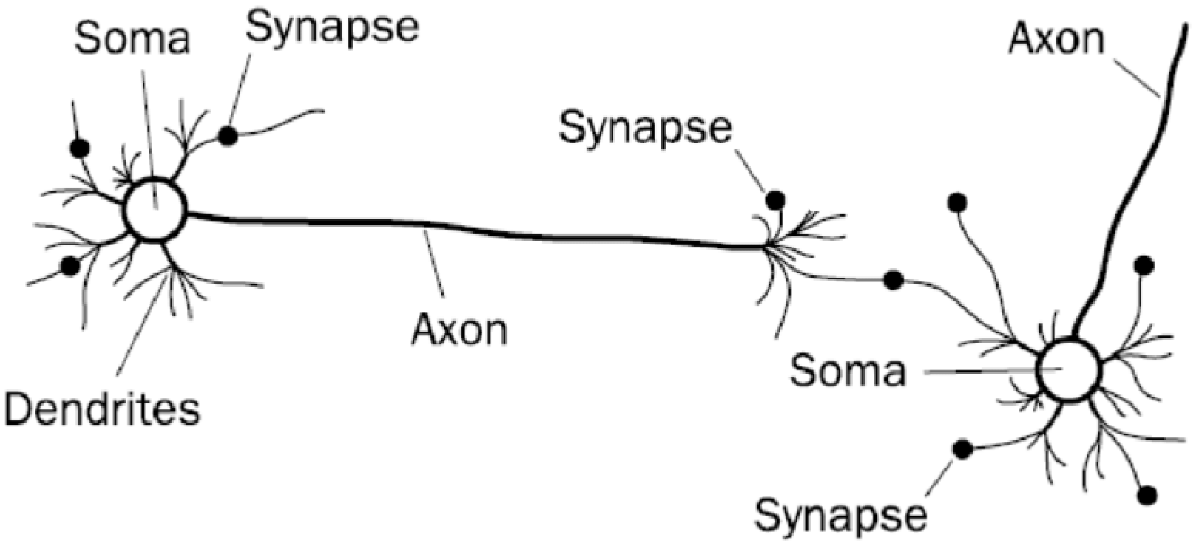
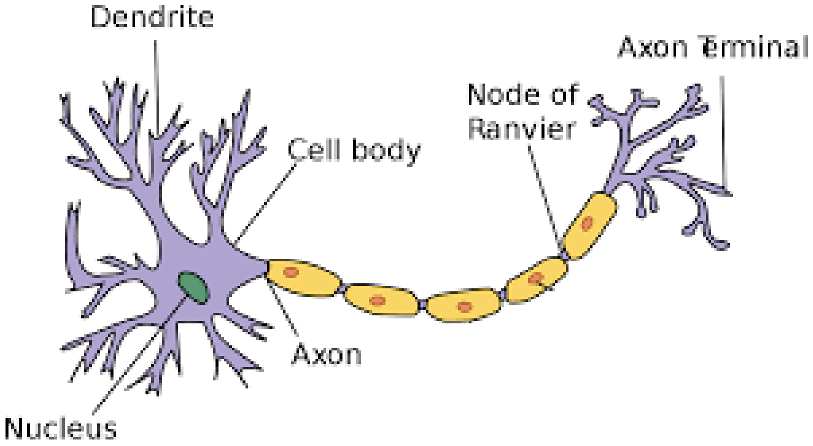
Neural Networks and Deep Learning

Dr. Fayyaz Minhas

Department of Computer Science
University of Warwick
https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs909/

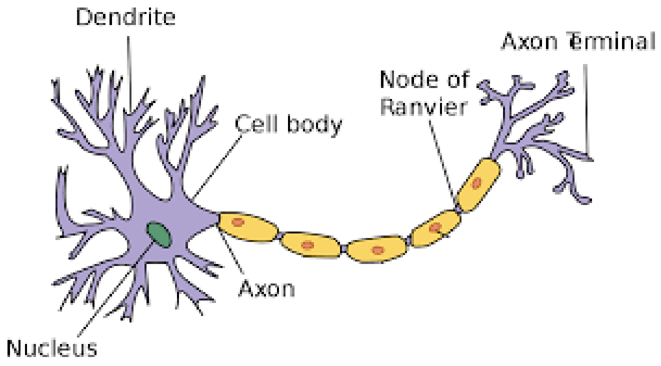
Biological Neurons and Networks

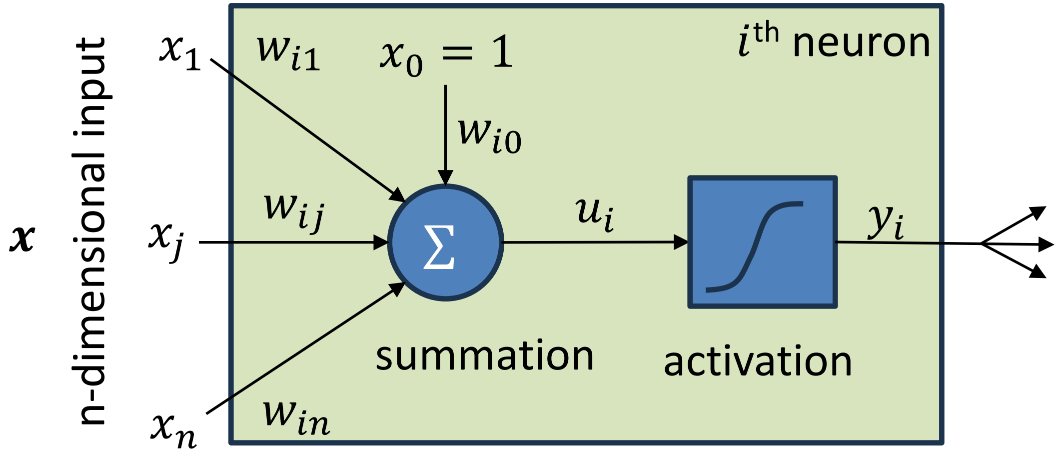




Single Neuron: Representation

- An abstraction of the biological neuron



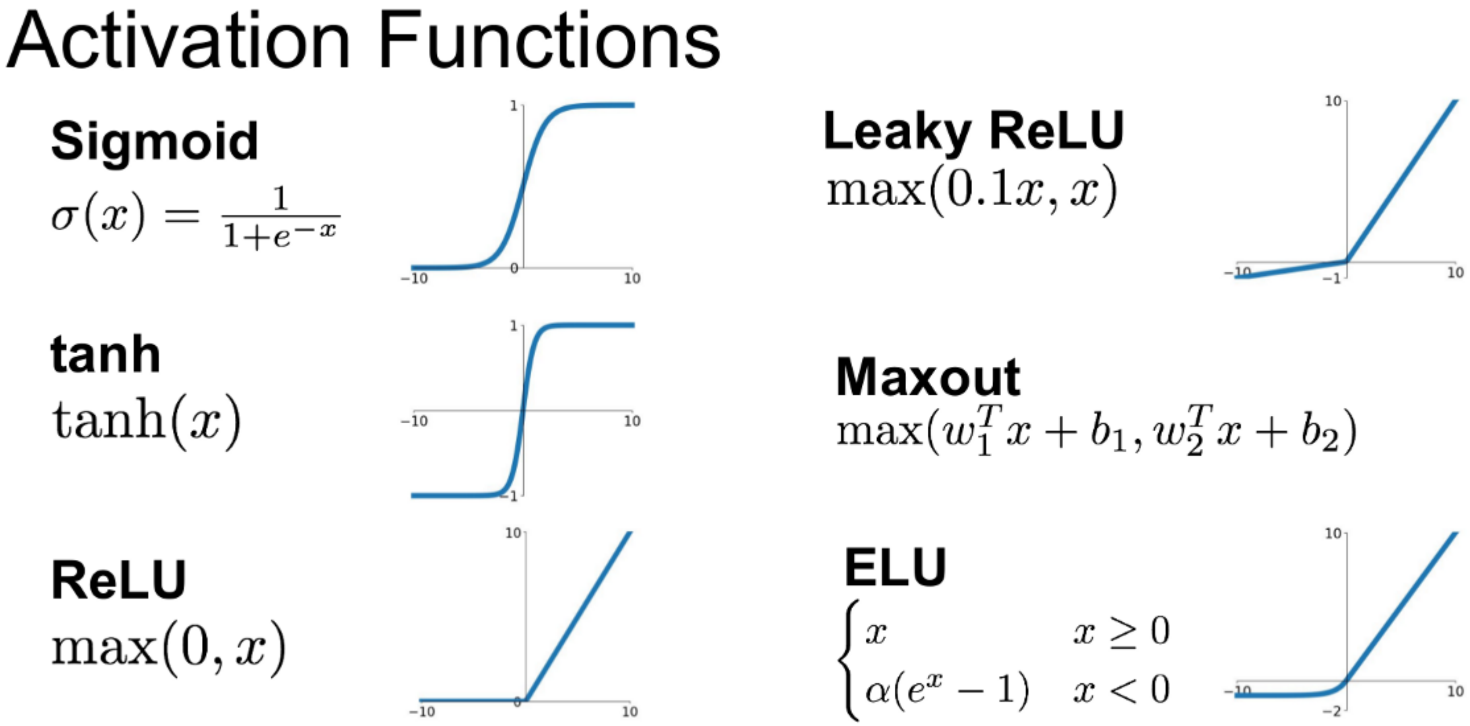


$$y_i = a(u_i) = a\left(\sum_j w_{ij} x_j\right)$$

$$u_i = \sum_j w_{ij} x_j = \boldsymbol{w}_{\boldsymbol{i}}^{\boldsymbol{T}} \boldsymbol{x}$$

Activation Functions

- Can use any activation function



Activation Functions

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$

Leaky ReLU

$\max(0.1x, x)$

Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Neural Networks

- Evaluation
  - – Error between predicted and target output
    - Predicted output: $y = a(u) = a(w^T x)$
    - Target output: t
    - Error: $(t - y)^2$
- Optimization
  - – Whenever the weights change, the output will change
  - – Optimize the weights so that the output matches the target
  - – Gradient Descent

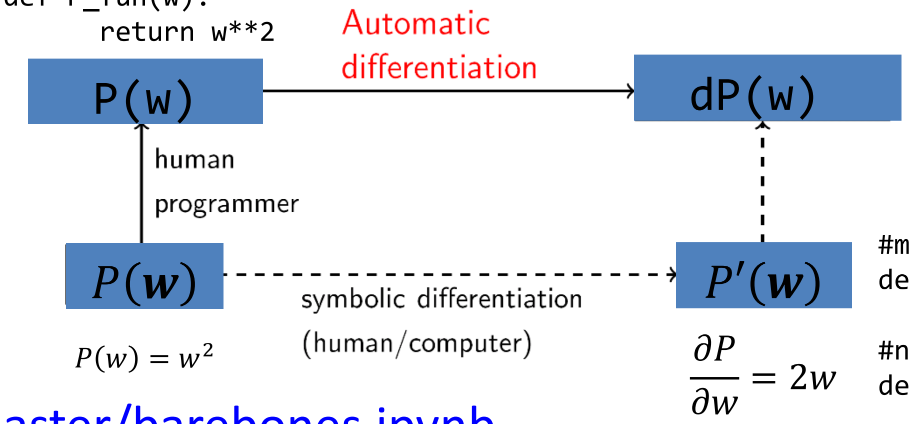
How to implement Neurons?

- Remember:
  - – If you can define a loss function
  - – And a regularizer
  - – The rest can be automated For any ML problem*!
    - Using Automatic Differentiation Libraries
      - – Autograd
      - – PyTorch
      - – TensorFlow
      - – JAX
      - – Zygote.jl

REO and SRM are all you need!

- **Representation**
  - *How does the model produce its output given its input*
    - $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x}$
- **Evaluation (SRM/Definition of Optimization Problem)**
  - Define a loss function and a regularization strategy write the optimization problem
  - $min_{\boldsymbol{w}} P(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} + \sum_{i=1}^{N} max(0, 1 - y_i f(\boldsymbol{x};\boldsymbol{w}))$
- **Optimization**
  - Obtain gradient $\nabla_w P(w) = \frac{\partial P(w)}{\partial x}$ through an automatic differentiation method
  - Apply gradient descent (or other optimization) updates until convergence
    - $w \leftarrow w - \alpha \nabla_w P(w)$
  - **Successful optimization is necessary for generalization (but not sufficient). Must check for successful optimization!**

```
def P_fun(w):
    return w**2
```



Automatic differentiation

P(w)

dP(w)

human programmer

$P(w)$

$P'(w)$

symbolic differentiation (human/computer)

$P(w) = w^2$

$\frac{\partial P}{\partial w} = 2w$

```
w = 1.0 # a value of w
out = P_fun(w)
out.backward() #generates
w.grad #equal to 2*w
```

```
#manual implement or use sympy
def dP_symbolic(w):
    return 2*w
#numeric differentiation
def dP_numeric(w,d):
    return (P_fun(w+d)-P_fun(w))/d
```

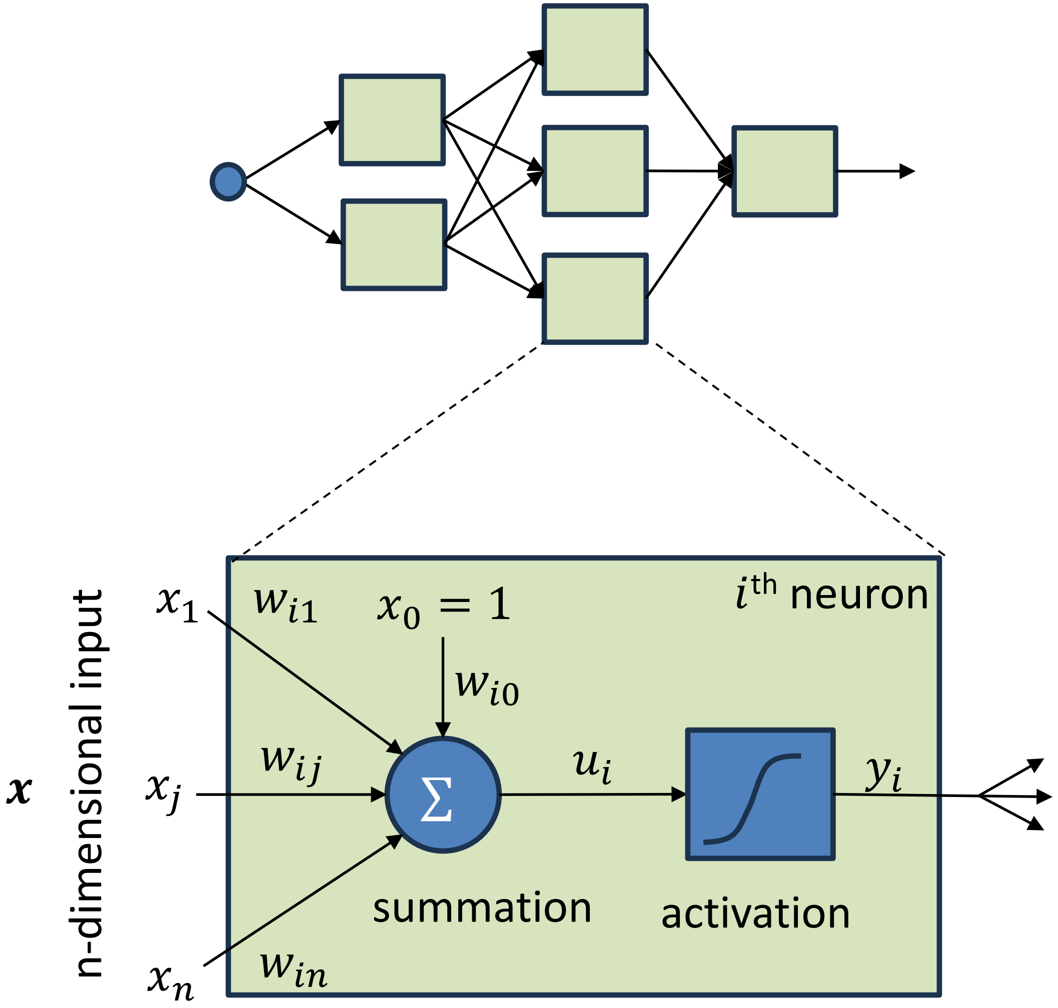
Go through this exercise:

https://github.com/foxtrotmike/CS909/blob/master/barebones.ipynb

*Terms and conditions apply

MULTILAYER PERCEPTRONS

A network of neurons



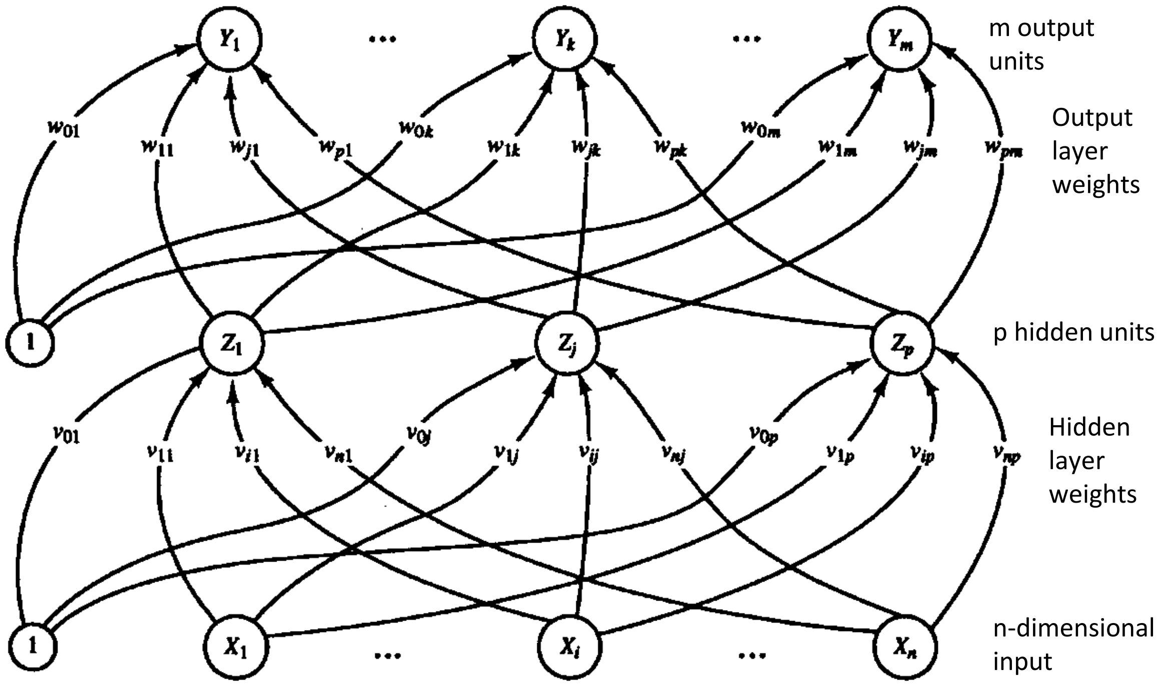
$$y_i = a(u_i) = a\left(\sum_j w_{ij}x_j\right)$$

$$u_i = \sum_j w_{ij}x_j = \boldsymbol{w}_{\boldsymbol{i}}^{\boldsymbol{T}}\boldsymbol{x}$$

Single to Multiple Neurons

Multilayer Perceptron: Representation

- Consists of multiple layers of neurons
  - Multi-Input Multi-Output
- Layers of units other than the input and output are called hidden units
- Unidirectional weight connections and biases (Feed-Forward)
- Activation functions
  - Use of activation functions
    - Sigmoidal activations
      - Nonlinear Operation: Ability to solve practical problems
      - Differentiable
      - Derivative can be expressed in terms of functions themselves: Computational Efficiency
    - Other activation functions also possible
  - Activation function is the same for all neurons in the same layer
    - Not a strict requirement though
  - Input layer just passes on the signal without processing (linear operation)



$$z_j = a(z\_in_j)$$

$$z\_in_j = \sum_{i=0}^{n} x_i v_{ij}\,,\ x_0 = 1, j = 1\ldots p$$

$$y_k = a(y\_in_k)$$

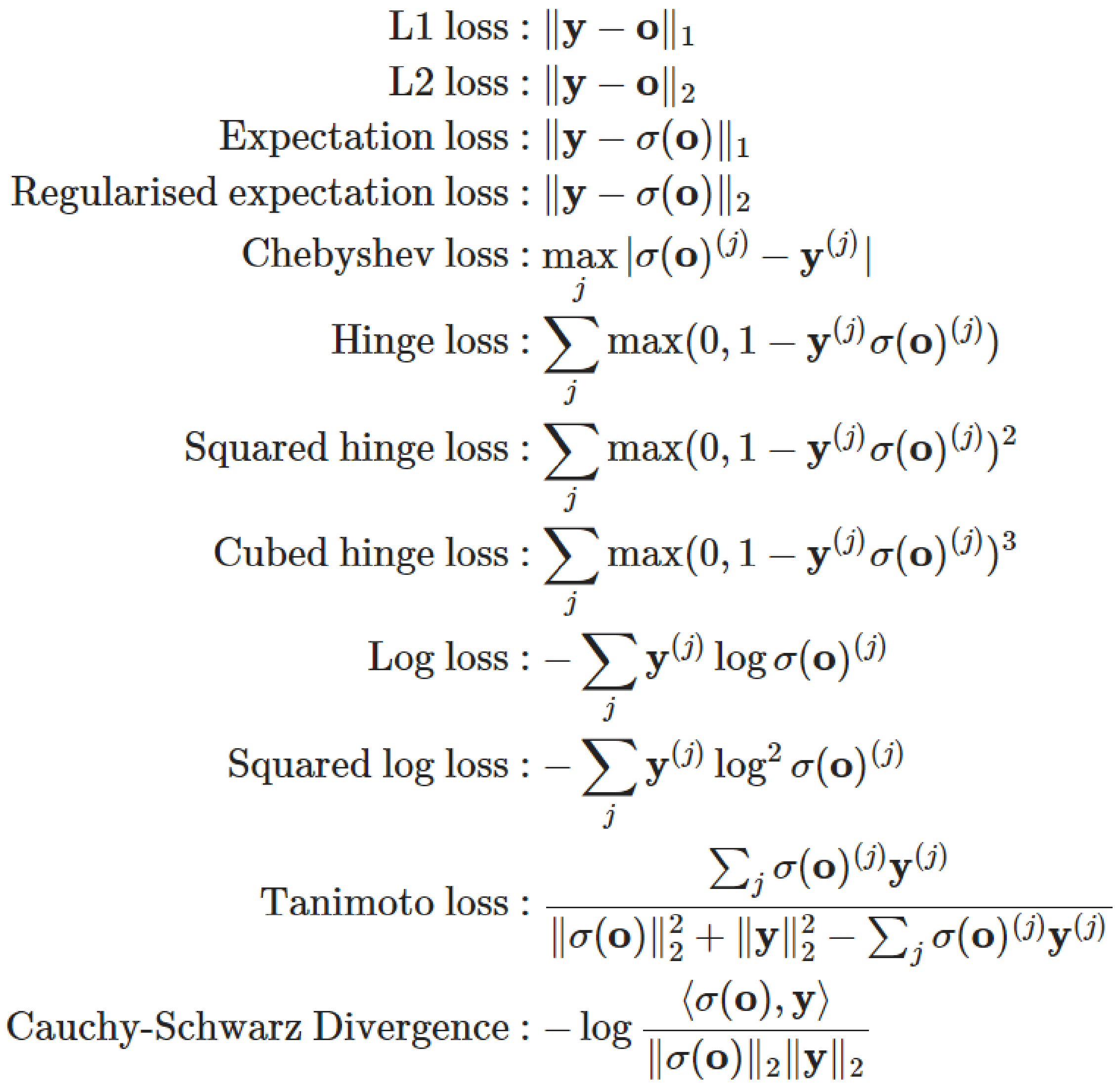
$$y\_in_k = \sum_{j=0}^{p} z_j w_{jk}\,, z_0 = 1, k = 1\ldots m$$

Multilayer Perceptron: Evaluation

- Compute the error between prediction and target
  - SSE Loss:

$$\text{loss} = \sum_{i} \sum_{k=1}^{m} \left(y_k^i - t_k^i\right)^2$$

Can use other loss terms.



$$\begin{aligned}
\text{L1 loss} &: \|\mathbf{y} - \mathbf{o}\|_1 \\
\text{L2 loss} &: \|\mathbf{y} - \mathbf{o}\|_2 \\
\text{Expectation loss} &: \|\mathbf{y} - \sigma(\mathbf{o})\|_1 \\
\text{Regularised expectation loss} &: \|\mathbf{y} - \sigma(\mathbf{o})\|_2 \\
\text{Chebyshev loss} &: \max_j |\sigma(\mathbf{o})^{(j)} - \mathbf{y}^{(j)}| \\
\text{Hinge loss} &: \sum_j \max(0, 1 - \mathbf{y}^{(j)} \sigma(\mathbf{o})^{(j)}) \\
\text{Squared hinge loss} &: \sum_j \max(0, 1 - \mathbf{y}^{(j)} \sigma(\mathbf{o})^{(j)})^2 \\
\text{Cubed hinge loss} &: \sum_j \max(0, 1 - \mathbf{y}^{(j)} \sigma(\mathbf{o})^{(j)})^3 \\
\text{Log loss} &: -\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)} \\
\text{Squared log loss} &: -\sum_j \mathbf{y}^{(j)} \log^2 \sigma(\mathbf{o})^{(j)} \\
\text{Tanimoto loss} &: \frac{\sum_j \sigma(\mathbf{o})^{(j)} \mathbf{y}^{(j)}}{\|\sigma(\mathbf{o})\|_2^2 + \|\mathbf{y}\|_2^2 - \sum_j \sigma(\mathbf{o})^{(j)} \mathbf{y}^{(j)}} \\
\text{Cauchy-Schwarz Divergence} &: -\log \frac{\langle \sigma(\mathbf{o}), \mathbf{y} \rangle}{\|\sigma(\mathbf{o})\|_2 \|\mathbf{y}\|_2}
\end{aligned}$$

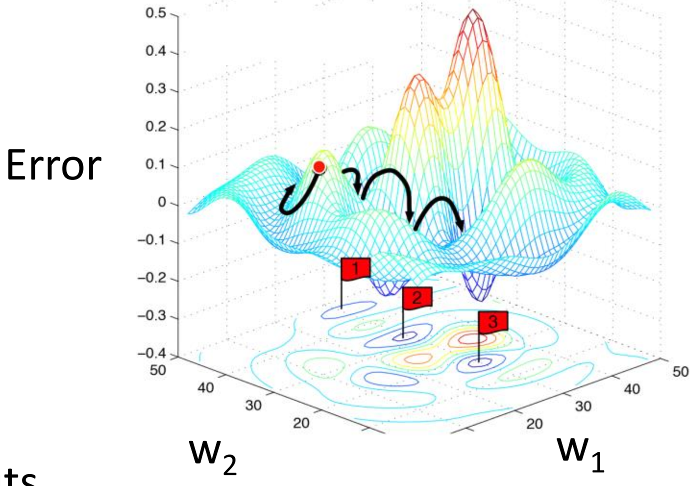
Multilayer Perceptron: Optimization

- Non-convex optimization
  - Because:

$$y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}$$

$$y_k = a(y\_in_k)$$

    - Weighted combination of activation function outputs



- Compute the gradient of the error/loss function with respect to each weight of the neural network

$$\frac{\partial l}{\partial w_{jk}} \qquad \frac{\partial l}{\partial v_{ij}}$$

- Update weights using gradient descent or other methods

$$w_{jk}^{new} \leftarrow w_{jk}^{old} - \alpha \frac{\partial l}{\partial w_{jk}^{old}} \quad \text{or} \quad \Delta w_{jk} = -\alpha \frac{\partial l}{\partial w_{jk}^{old}}$$

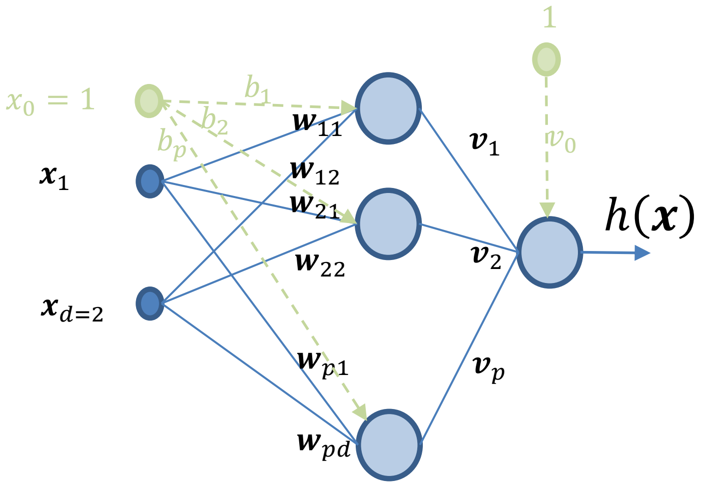
$$v_{ij}^{new} \leftarrow v_{ij}^{old} - \alpha \frac{\partial l}{\partial v_{ij}^{old}} \quad \text{or} \quad \Delta v_{ij} = -\alpha \frac{\partial l}{\partial v_{ij}^{old}}$$

REO for MLPs

- **Representation**
  - Defined by the architecture
    - Number of inputs and outputs, Interconnection of neurons, number of neurons in layers, activation functions, etc.

$$h(\boldsymbol{x}) = \sum_{i=1}^{P} \boldsymbol{v}_i a\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right) + v_0 = \boldsymbol{V} a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + v_0$$

  - Modern DL libraries require you to define “Representation”
- **Evaluation**
  - Defined by the ML problem
  - Can use any loss function
    - Square Error Loss
    - Hinge Loss
    - Cross-Entropy Loss



$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{V} a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + v_0$$

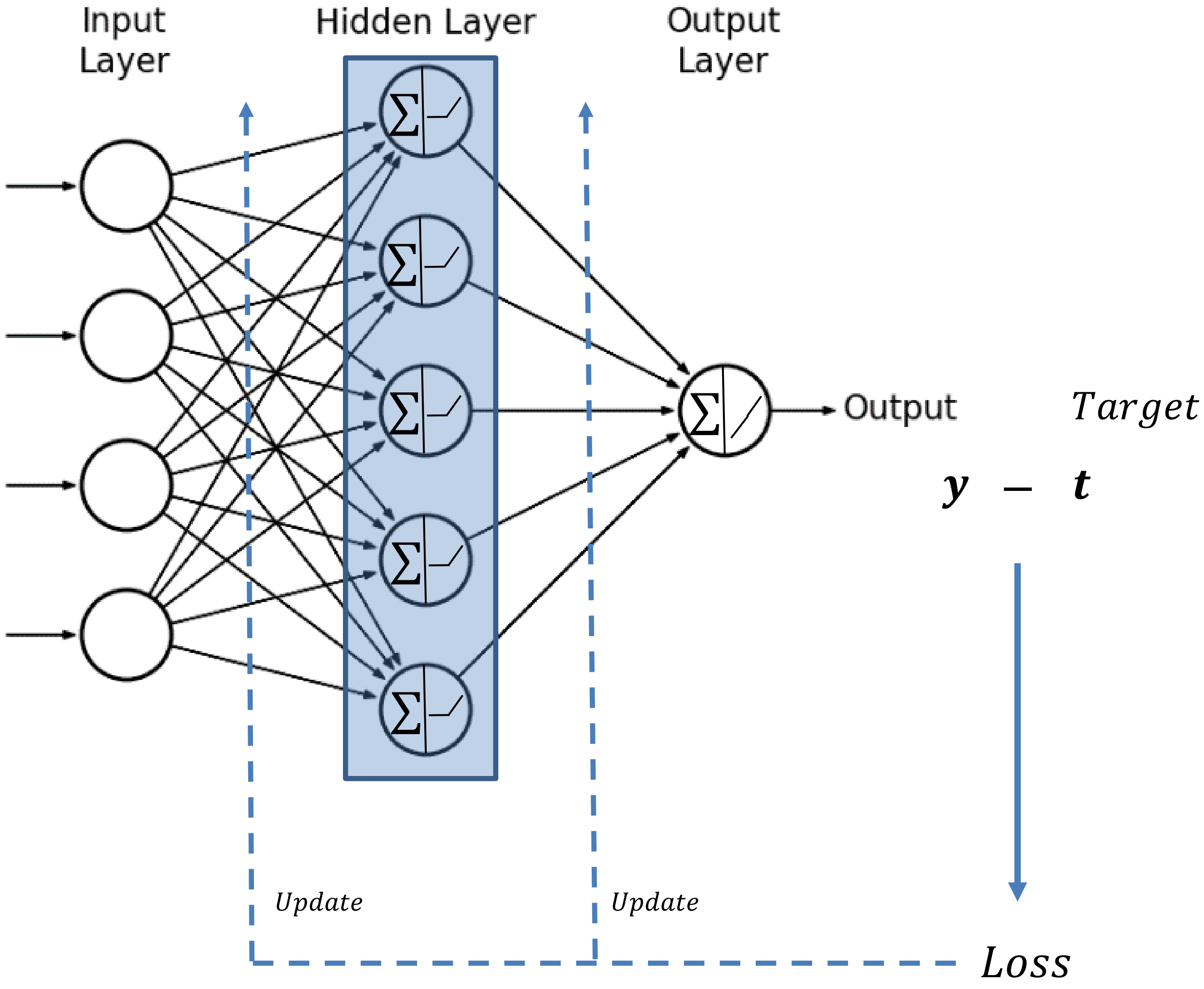
- **Optimization**
  - Solve for weights that reduce error over training data and (hopefully!) generalize to test data
  - Using any optimization method
    - Stochastic Gradient Descent
    - Adaptive Learning Rate with Momentum (Adam)
    - So many other

https://playground.tensorflow.org

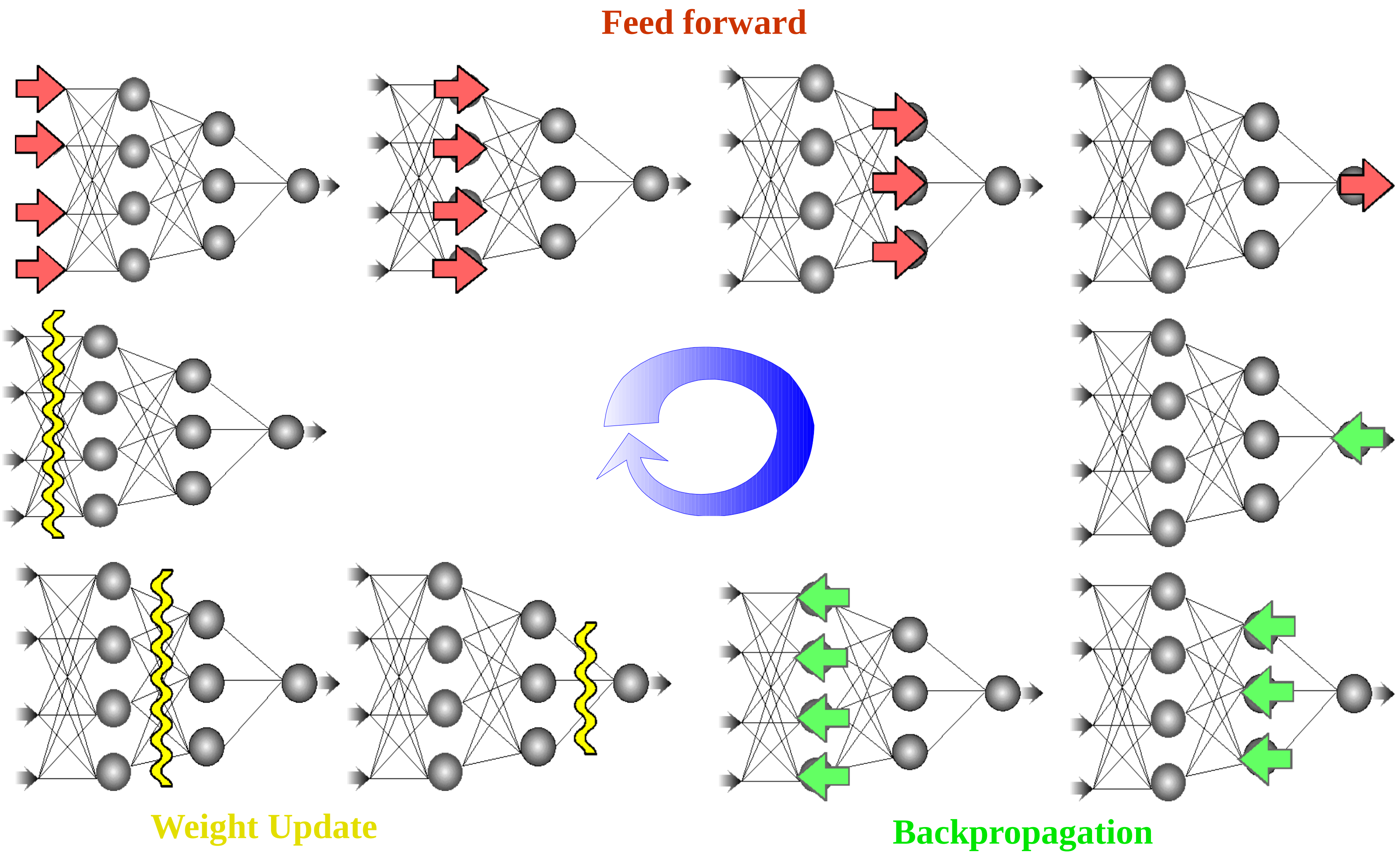
Important:
The output of a fully connected layer of weights $\boldsymbol{W}$ can be viewed as a transformation $z\colon \mathrm{R}^{\mathrm{d}} \to \mathrm{R}^{\mathrm{p}}$ involving a matrix-vector product and an activation function

$$z(\boldsymbol{x}) = a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

Multilayer Perceptron



Backpropagation training cycle



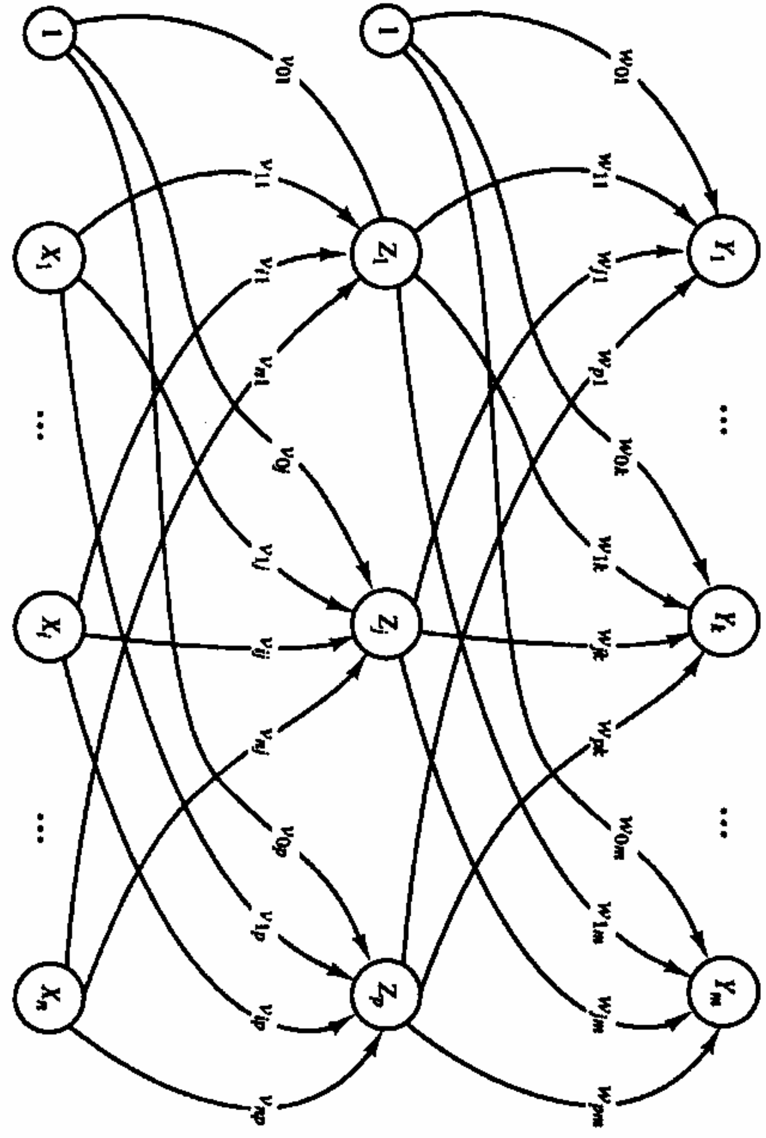
Feed forward

Weight Update

Backpropagation

Training

- During training we are presented with input patterns and their targets
- At the output layer we can compute the error between the targets and actual output and use it to compute weight updates through the Delta Rule
- But the Error cannot be calculated at the hidden input as their targets are not known
- Therefore we propagate the error at the output units to the hidden units to find the required weight changes (Backpropagation)
- 3 Stages
  - Feed-forward of the input training pattern
  - Calculation and Backpropagation of the associated error
  - Weight Adjustment
- Based on minimization of SSE (Sum of Square Errors)



Proof for the Learning Rule

We can use the chain rule to compute the gradient of E

$$E = 0.5 \sum_{k} (t_k - y_k)^2$$

How much does E change with change in w_{jk}

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} 0.5 \sum_{k} (t_k - y_k)^2 = \frac{\partial}{\partial w_{jk}} 0.5 (t_k - y_k)^2$$

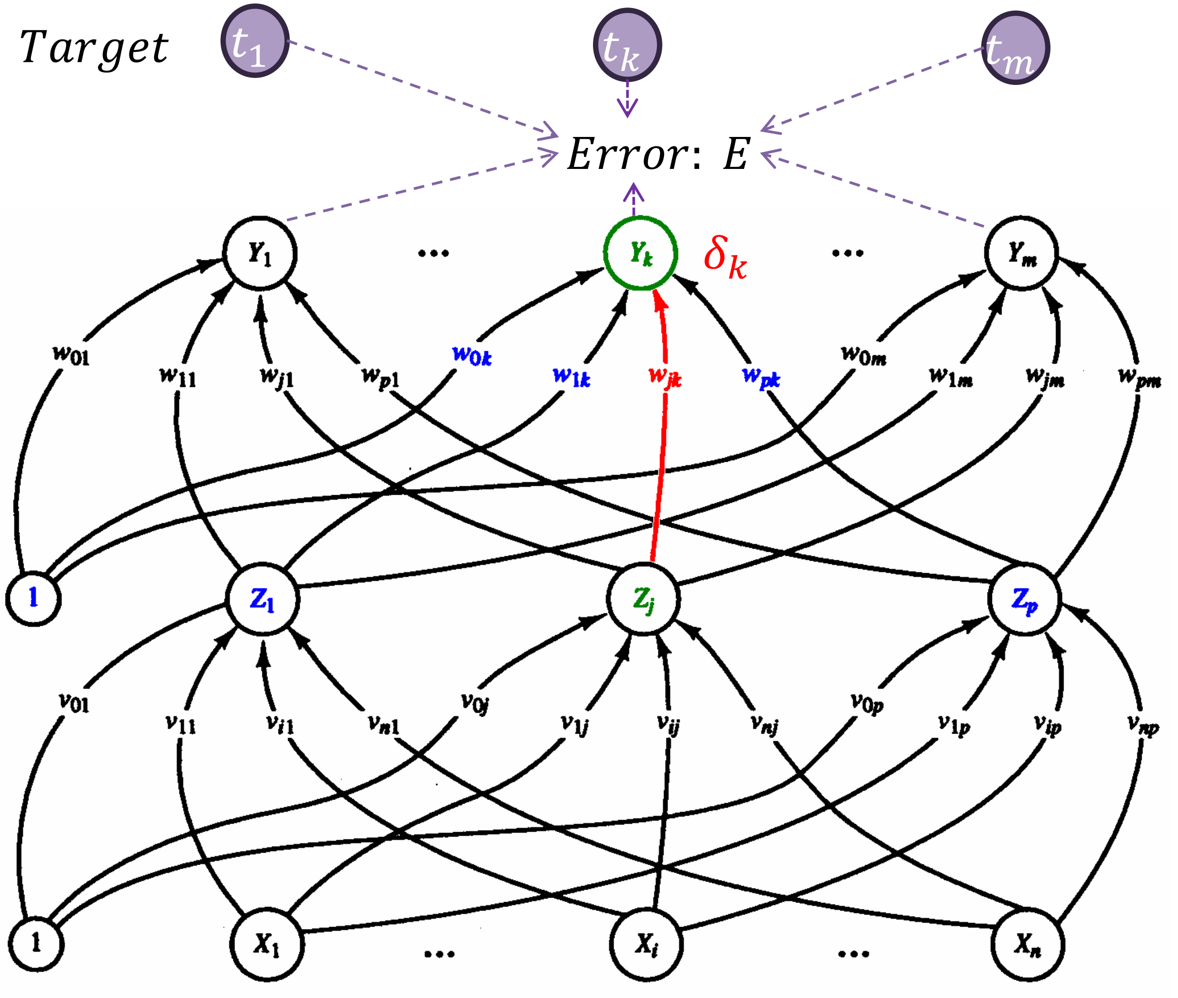
$$= -(t_k - y_k) \frac{\partial}{\partial w_{jk}} y_k = -(t_k - y_k) \frac{\partial}{\partial w_{jk}} a(y\_in_k)$$

$$= -(t_k - y_k) a'(y\_in_k) \frac{\partial}{\partial w_{jk}} y\_in_k$$

$$= -(t_k - y_k) a'(y\_in_k) \frac{\partial}{\partial w_{jk}} \sum_{j=0}^{p} z_j w_{jk}$$

$$= -(t_k - y_k) a'(y\_in_k) z_j = -\delta_k z_j$$

Change in w_{jk} affects only y_k



$$z_j = a(z\_in_j), z_{in_j} = \sum_{i=0}^{n} x_i v_{ij}, x_0 = 1, j = 1 \ldots p$$

$$y_k = a(y\_in_k), y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}, z_0 = 1, k = 1 \ldots m$$

Take away lesson:

The change in w_{jk} is proportional to

- The error $t_k - y_k$
- Output z_j
- The derivative of the activation function $a'(y\_in_k)$

Weight update will be zero if any of these terms is zero!

Use of Gradient Descent Minimization

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j$$

With $\delta_k = (t_k - y_k) a'(y\_in_k)$

The Learning Rule…

How much does E change with change in v_{ij} :

$$\frac{\partial E}{\partial v_{ij}} = \frac{\partial}{\partial v_{ij}} 0.5 \sum_k (t_k - y_k)^2 = 0.5 \sum_k \frac{\partial}{\partial v_{ij}} (t_k - y_k)^2$$

$$= \sum_k (t_k - y_k) \frac{\partial}{\partial v_{ij}} (-y_k) = -\sum_k (t_k - y_k) \frac{\partial}{\partial v_{ij}} a(y_{in_k})$$

$$= -\sum_k (t_k - y_k) a'(y_{in_k}) \frac{\partial}{\partial v_{ij}} y_{in_k}$$

$$= -\sum_k \delta_k \frac{\partial}{\partial v_{ij}} \sum_{j=0}^{p} z_j w_{jk} = -\sum_k \delta_k \frac{\partial}{\partial v_{ij}} z_j w_{jk}$$

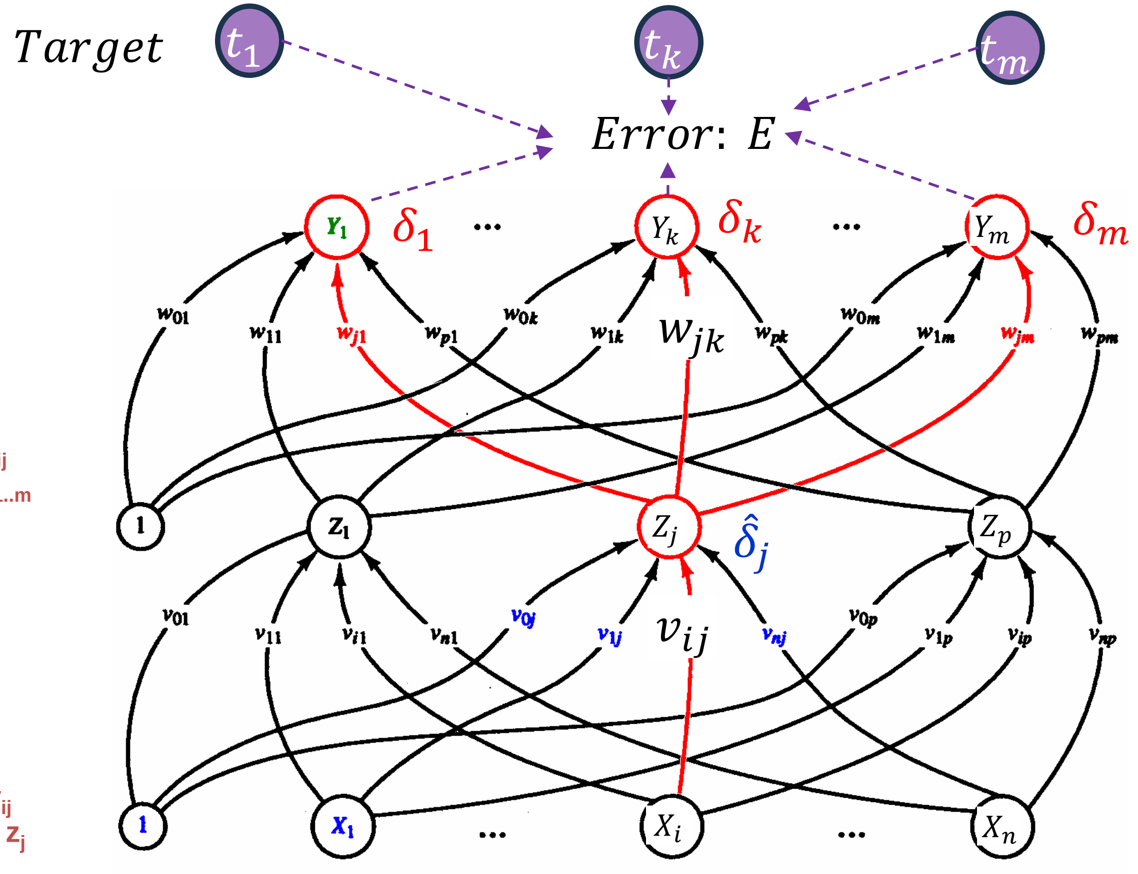
$$= -\sum_k \delta_k w_{jk} \frac{\partial}{\partial v_{ij}} a\left(z_{in_j}\right) = -\sum_k \delta_k w_{jk} a'\left(z_{in_j}\right) \frac{\partial}{\partial v_{ij}} z_{in_j}$$

$$= -\sum_k \delta_k w_{jk} a'\left(z_{in_j}\right) \frac{\partial}{\partial v_{ij}} \sum_{i=0}^{n} x_i v_{ij}$$

$$= -\sum_k \delta_k w_{jk} a'\left(z_{in_j}\right) x_i = -\hat{\delta}_j x_i$$

Change in v_{ij} affects all $Y_{1..m}$

Change in v_{ij} affects only z_j



$z_j = a(z\_in_j), z_{in_j} = \sum_{i=0}^{n} x_i v_{ij}, x_0 = 1, j = 1 \ldots p$

$y_k = a(y\_in_k), y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}, z_0 = 1, k = 1 \ldots m$

With $\delta_k = (t_k - y_k) a'(y\_in_k)$

Use of Gradient Descent Minimization

$$\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \hat{\delta}_j x_i$$

With: $\hat{\delta}_j = \sum_k \delta_k w_{jk} a'\left(z_{in_j}\right) x_i$ or

$$\hat{\delta}_j = \sum_k (t_k - y_k) w_{jk} a'(y\_in_k) a'\left(z_{in_j}\right) x_i$$

Take away message: The change in v_{ij} is proportion to:

- The input x_i
- $\hat{\delta}_j$: The backprop term which contains product of activation function derivatives

Understanding Backpropagation

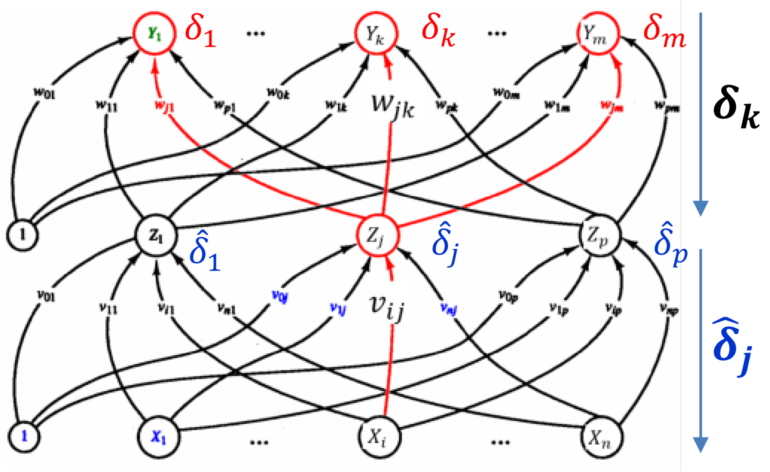
- Pass the input and compute the output
- Compute Error
- Compute Gradient of error wrt weights
- Compute weight updates
  - Compute δ_k
  - "Backpropagate" these δ_k through the network to Compute $\hat{\delta}_j$
  - Compute Δw_{jk} and Δv_{ij}
- Update weight updates

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j$$

$$\delta_k = (t_k - y_k) f'(y\_in_k)$$

$$\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \delta_j x_i$$

$$\hat{\delta}_j = \sum\nolimits_k \delta_k w_{jk} a'\left(z_{in_j}\right) x_i$$



Training Algorithm

Step 0.

Initialize weights. (Set to small random values).

Step 1.

While stopping condition is false, do Steps 2–9.

Step 2.

For each training pair, do Steps 3–8.

Feedforward:

Step 3.

Each input unit $(X_i, i = 1, \ldots, n)$ receives input signal x_i and broadcasts this signal to all units in the layer above (the hidden units).

Step 4.

Each hidden unit $(Z_j, j = 1, \ldots, p)$ sums its weighted input signals,

$$z_{\text{in}_j} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

applies its activation function to compute its output signal,

$$z_j = a(z_{\text{in}_j})$$

and sends this signal to all units in the layer above (output units).

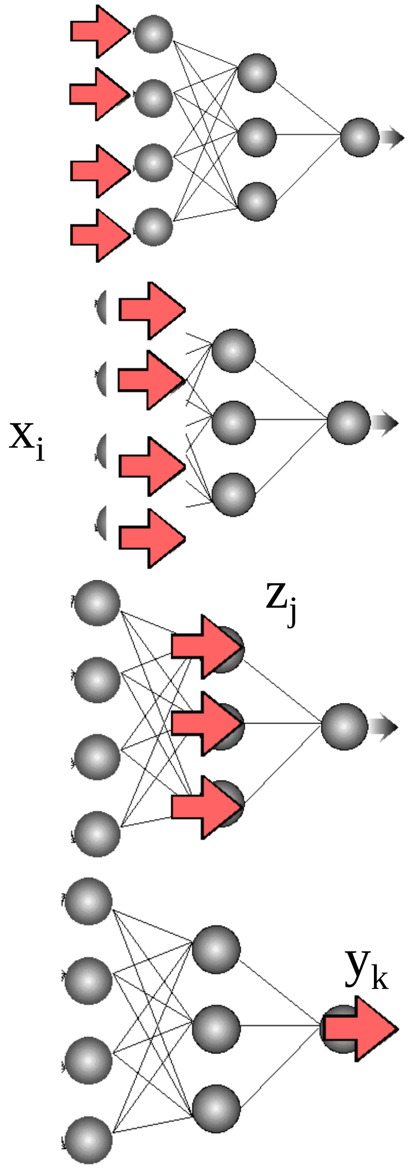
Step 5.

Each output unit $Y_k, k = 1, \ldots, m$ sums its weighted input signals,

$$y_{\text{in}_k} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk},$$

and applies its activation function to compute its output signal,

$$y_k = a(y_{\text{in}_k}).$$



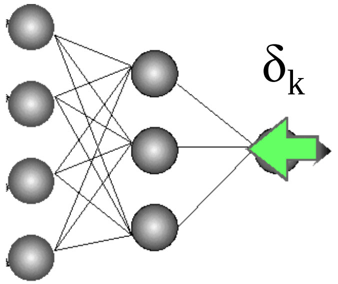
Training Algorithm…

Backpropagation of error:

Step 6.

Each output unit $Y_k, k = 1, \ldots, m$ receives a target pattern corresponding to the input training pattern, computes its error information term,

$$\delta_k = (t_k - y_k)a'(y_{\text{in}_k})$$



calculates its weight correction term (used to update w_{jk} later),

$$\Delta w_{jk} = \alpha \delta_k z_j,$$

calculates its bias correction term (used to update w_{0k} later),

$$\Delta w_{0k} = \alpha \delta_k,$$

and sends δ_k to units in the layer below.

Training Algorithm…

Step 7.

Each hidden unit $Z_j, j = 1, \ldots, p$ sums its delta inputs (from units in the layer above),

$$\delta_{\text{in}_j} = \sum_{k=1}^{m} \delta_k w_{jk},$$

multiplies by the derivative of its activation function to calculate its error information term,

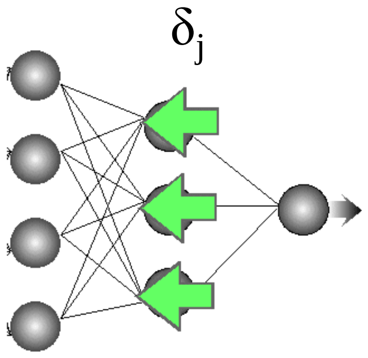
$$\hat{\delta}_j = \delta_{\text{in}_j} a'(z_{\text{in}j}),$$

calculates its weight correction term (used to update w_{ij} later),

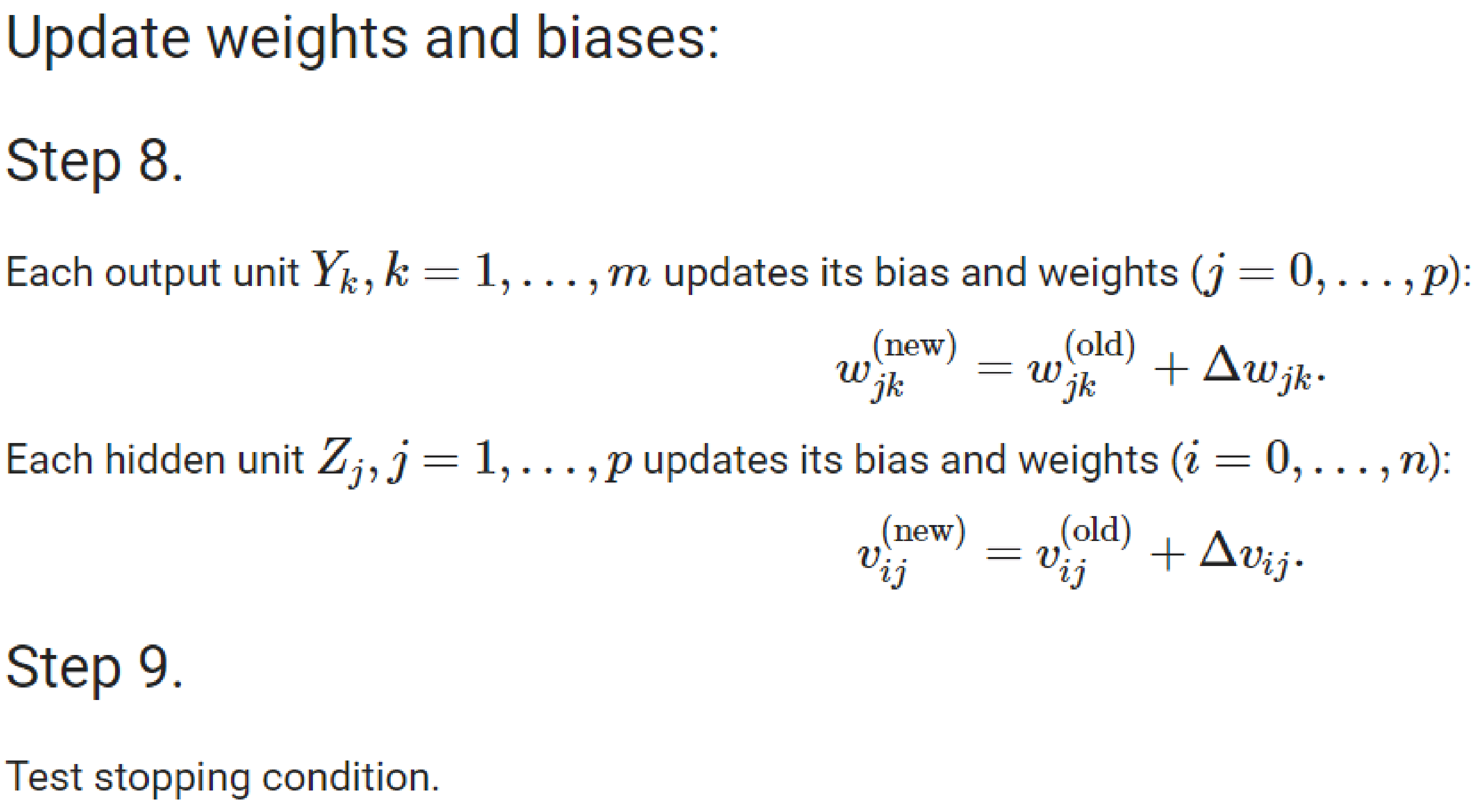
$$\Delta v_{ij} = \alpha \hat{\delta}_j x_i,$$

and calculates its bias correction term (used to update v_{0j} later),

$$\Delta v_{0j} = \alpha \hat{\delta}_j.$$



Training Algorithm…



Update weights and biases:

Step 8.

Each output unit $Y_k, k = 1, \ldots, m$ updates its bias and weights ($j = 0, \ldots, p$):

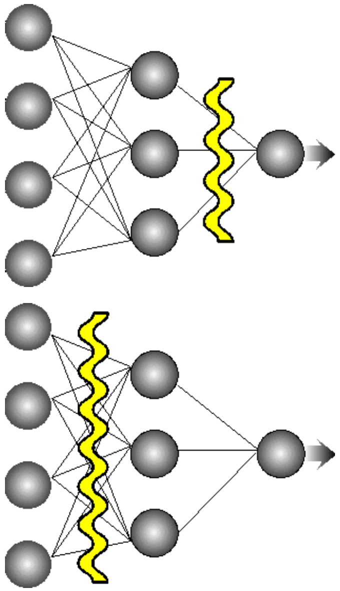
$$w_{jk}^{(\text{new})} = w_{jk}^{(\text{old})} + \Delta w_{jk}.$$

Each hidden unit $Z_j, j = 1, \ldots, p$ updates its bias and weights ($i = 0, \ldots, n$):

$$v_{ij}^{(\text{new})} = v_{ij}^{(\text{old})} + \Delta v_{ij}.$$

Step 9.

Test stopping condition.



Taken from:
Fausett, Laurene V. Fundamentals of Neural Networks: Architectures, Algorithms And Applications: United States Edition. US Ed edition. Englewood Cliffs, NJ: Pearson, 1993.

Optimization in minibatches

- We can do a full-scale optimization across all examples in each step or take a few examples at a time to determine the gradients and perform an update
  - Mini-batches
    - Stochastic gradient descent
    - Reduces memory consumption
    - Faster convergence

https://playground.tensorflow.org

Coding

- Using Keras
- https://github.com/foxtrotmike/CS909/blob/master/keras_barebones.ipynb

- PyTorch
- Barebones code in PyTorch
- https://github.com/foxtrotmike/CS909/blob/master/barebones.ipynb

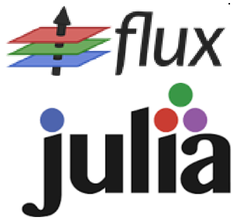
- Using nn-module
- https://github.com/foxtrotmike/CS909/blob/master/pytorch_nn_barebones.ipynb

- Universal Approximation code:
- https://github.com/foxtrotmike/CS909/blob/master/uniapprox.ipynb

- Digit Classification Exercise
- https://github.com/foxtrotmike/CS909/blob/master/pytorch_mlp_mnist.ipynb

Libraries

- All Neural Network/Deep Learning Libraries <u>Do three things</u>
  - Automatic Differentiation (Efficient Algorithms such as Reverse mode autodiff!)
  - Implement Optimizers
  - Use efficient hardware for multiprocessing (GPUs)
- Support efficient representation / abstraction



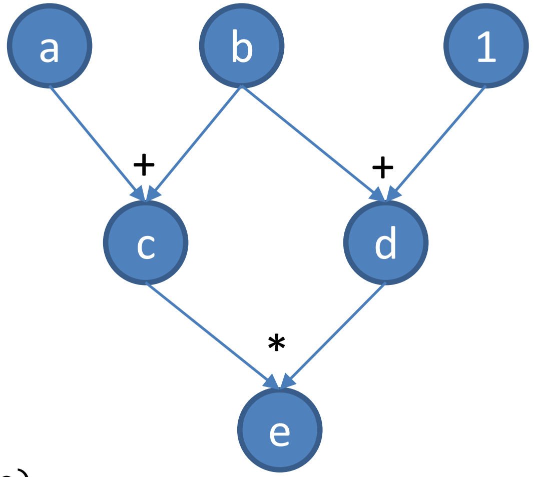
TensorFlow
- Static Computing Graphs
  - Build before you go (new version has dynamic graphs too!)
  - Compile then run/fit
- Good Documentation
- Distributed Computing / Delivery
- TensorFlow.js

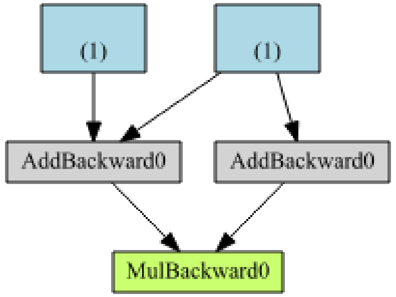
pyTorch
- Dynamic Computing Graphs
  - Graph built at run time
  - Build as you go
- Good for research

```
using Zygote

# Define a simple function
f(x) = 3x^2 + 2x + 1
# derivative of f at x = 2
gradient(f, 2)
```

NN/Deep Learning Libraries

- Essentially Automatic Differentiation Tools with optimization packages
  - Represent a neural network loss calculation as a computational graph and then compute the gradients
    - Have rules for each operator on how to differentiate “through” that operator
- Can use GPU
    - $e = (a+b)(b+1) = ab + a + b^2 + b$
    - $\left.\frac{\partial e}{\partial a}\right|_{(a=2,b=1)} = b + 1 = 2$
    - $\left.\frac{\partial e}{\partial b}\right|_{(a=2,b=1)} = a + 2b + 1 = 5$





```
import torch
import numpy as np
from torchviz import make_dot
a = torch.from_numpy(np.array([2.0])); a.requires_grad_(True)
b = torch.from_numpy(np.array([1.0])); b.requires_grad_(True)
e = (a+b)*(b+1)
e.backward()
print(a.grad) # 2
print(b.grad) # 5
make_dot(e)
```

```
!pip install torchviz
from torchviz import make_dot
make_dot(tloss,params=dict(model.named_parameters()))
```

Computation Graph of a two-layer network

Representation

```
model = torch.nn.Sequential(
          torch.nn.Linear(2, 2),
          torch.nn.Sigmoid(),
          torch.nn.Linear(2, 1),
          torch.nn.Sigmoid()
        ).to(device)

z = model(x)
```

Evaluation

```
e = loss_fn(z, y)
```

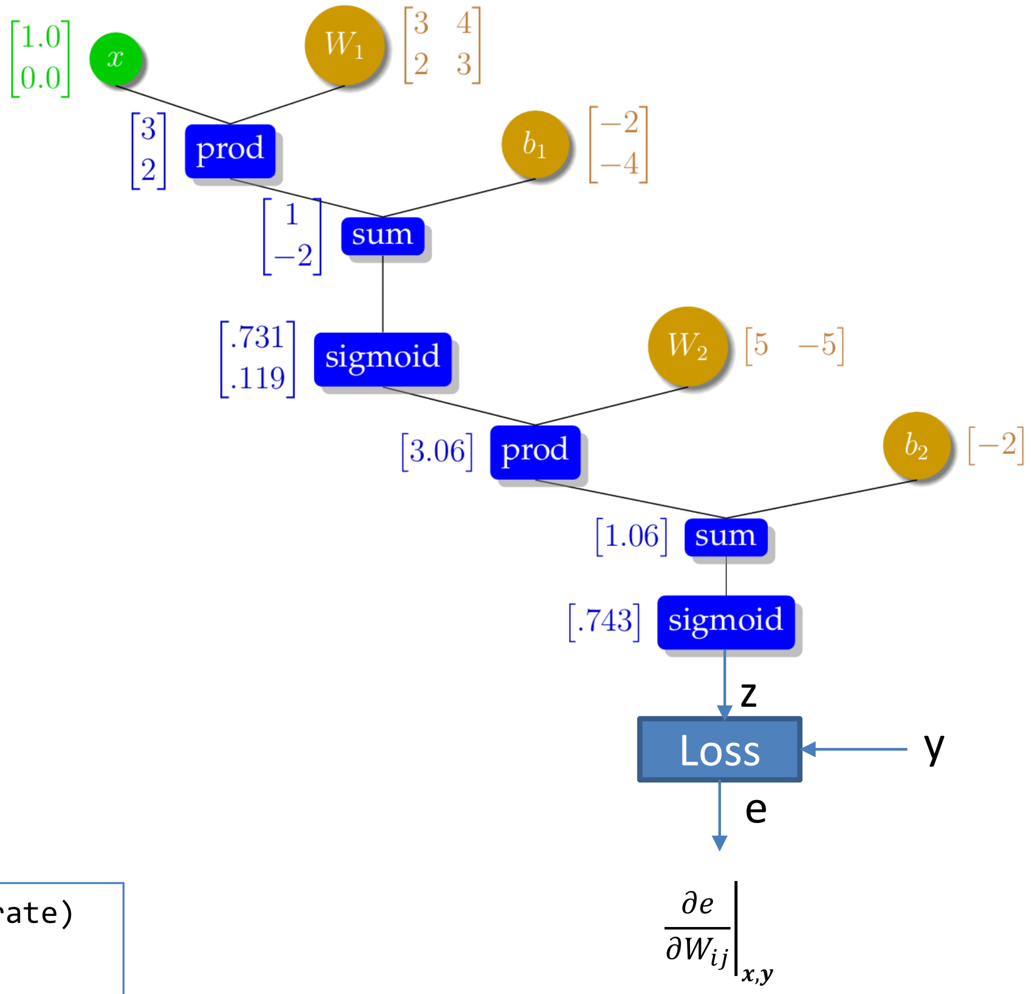
Optimization

Manual Gradient Descent

```
model.zero_grad()
e.backward()


with torch.no_grad():
for param in model.parameters():
  param.data -= learning_rate * param.grad
```

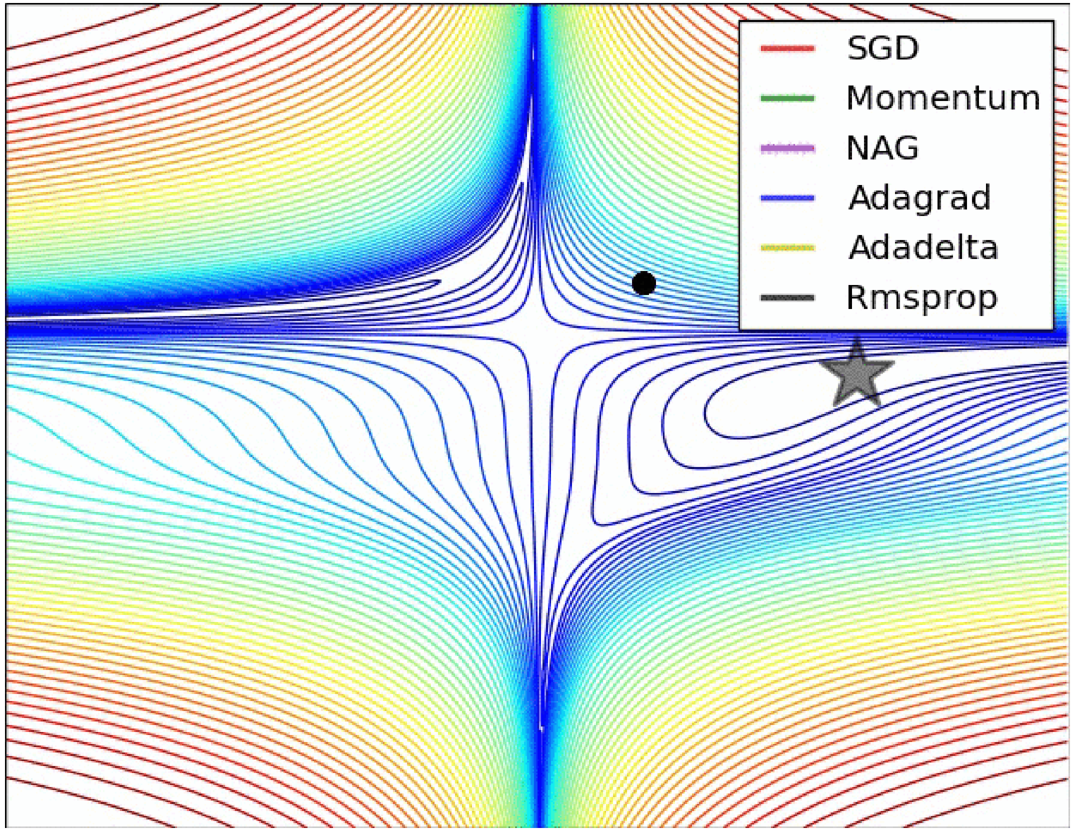
Using Built-in Optimizer

```
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
model.zero_grad()
e.backward()
optimizer.step()
```

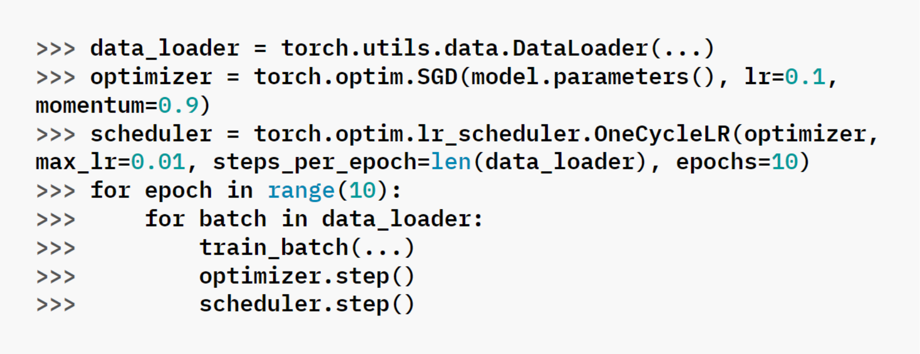


Optimization Methods

- Gradient Descent: Go down! $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$
- Stochastic Gradient Descent $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$
- Mini-batch Gradient Descent $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
- SGD with momentum: accelerate if going downhill for a long time
- Nesterov momentum: accelerate but not indefinitely
- Adagrad: Adaptive Learning Rate by accumulating past gradients
- AdaDelta/RMSProp: Adaptive Learning rate but does not accumulate all past gradients
- Adam: Adaptive learning rate with momentum

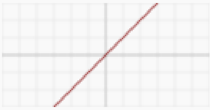
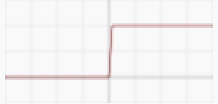
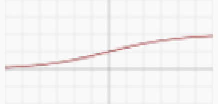
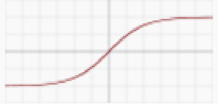
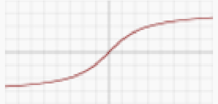
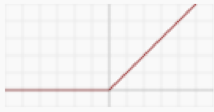
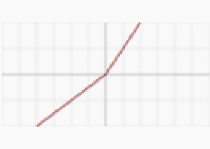
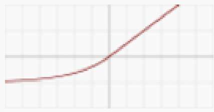
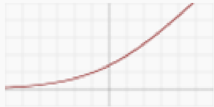


- Learning rate scheduling
  - Changing Learning rates at different times in the learning
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.OneCycleLR.html



```
>>> data_loader = torch.utils.data.DataLoader(...)
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1,
momentum=0.9)
>>> scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer,
max_lr=0.01, steps_per_epoch=len(data_loader), epochs=10)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         optimizer.step()
>>>         scheduler.step()
```

An overview of gradient descent optimization algorithms by Sebastian Ruder, 20-16
http://sebastianruder.com/optimizing-gradient-descent/ , https://arxiv.org/abs/1609.04747

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$ sigmoid	$f'(x) = f(x)(1 - f(x))$
TanH Bipolar sigmoid		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$
Softmax	Used for multi-class classification	$f(x_i) = p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$	$\frac{\partial f(x_i)}{\partial x_k} = \begin{cases} p_i(1-p_i) & i = k \\ -p_i p_k & else \end{cases}$

Minhas, Fayyaz ul Amir Afsar, and Amina Asif. "**Learning Neural Activations**." *arXiv:1912.12187 [Cs, Stat]*, December 27, 2019. http://arxiv.org/abs/1912.12187.

Here x is not an example, rather the input to an activation function f

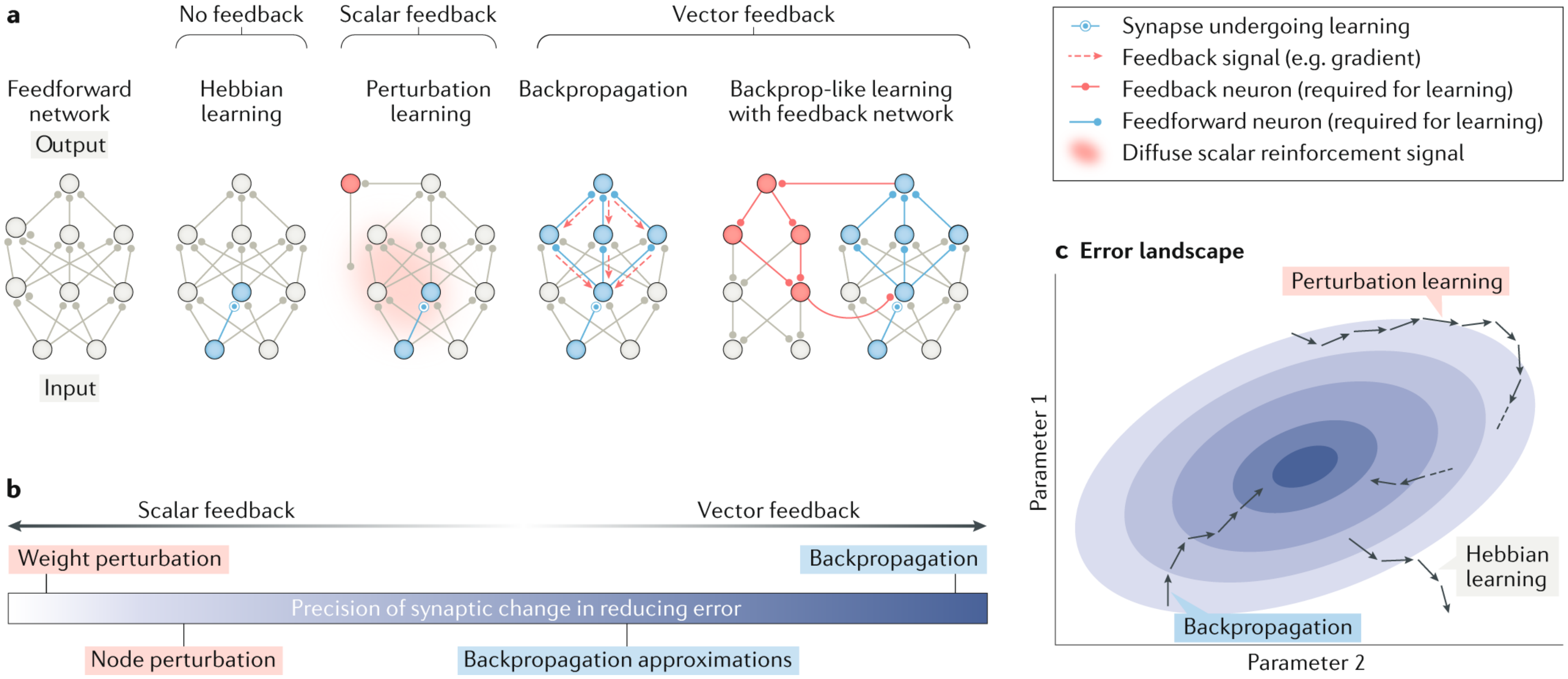
Some functions like the "**Softmax**" take a vector as input and produce a vector output. The softmax function takes a vector of "logits" as input and produces pseudo-probability values as output.

Readmore: https://en.wikipedia.org/wiki/Softmax_function

Does the brain do backpropagation?

- Short answer:
  - No

- Long answer:
  - Not enough evidence



Lillicrap, Timothy P., et al. "Backpropagation and the brain." *Nature Reviews Neuroscience* 21.6 (2020): 335-346.

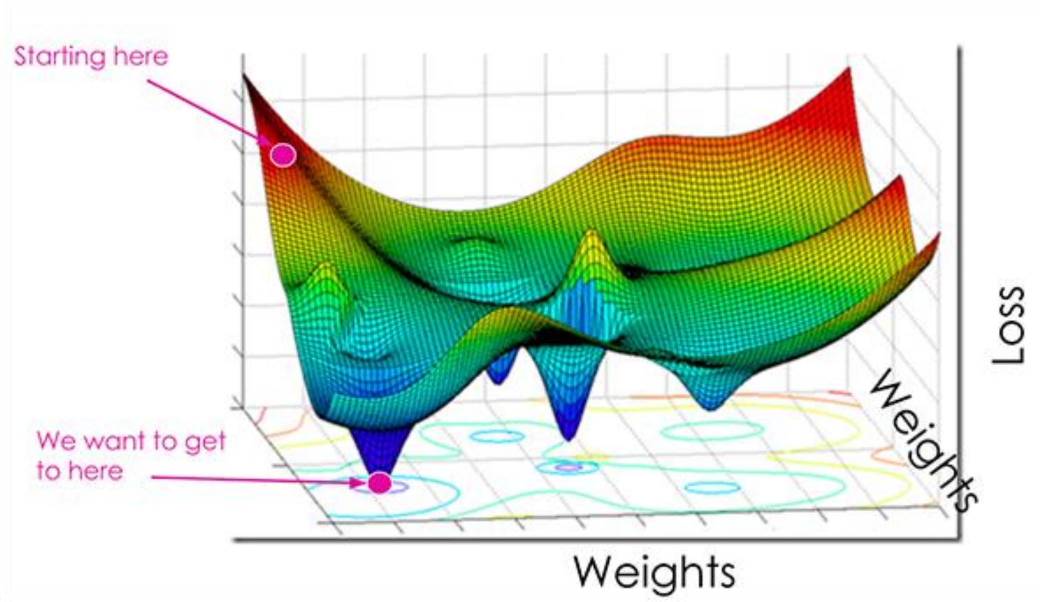
HOW TO IMPROVE NEURAL NETWORK TRAINING

Parameter Selection

- A MLP has a large number of parameters
  - Number of Neurons in Each Layer
  - Number of Layers
  - Activation Function for each neuron: ReLU, logsig…
  - Layer Connectivity: Dense, Dropout…
- Objective function
  - Loss Function: MSE, Entropy, Hinge loss, …
  - Regularization: L1, L2…
- Optimization Method
  - SGD, ADAM, RMSProp, LM …
  - Parameters for the Optimization method
    - Weight initialization
    - Momentum, weight decay, etc.

Issues with Neural Networks with non-linear activations

- Unlike an SVM, which has a single global optimum due to its convex loss function, the error surface of a neural network is not as smooth
- This complicates the optimization
- A number of "tricks" are used to make the neural network learn



Loss Landscape of a neural network

Examples showing that combinations and compositions (such as those that can arise in a multilayer perceptron) of even convex functions are not convex

Given convex functions

$$g_1(x) = -x$$
$$g_2(x) = x^2$$

Following are NOT convex:

$$g_1(x) - g_2(x) = -x - x^2$$
$$g_1(g_2(x)) = -x^2$$

Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. "Visualizing the Loss Landscape of Neural Nets." In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc., 2018. https://papers.nips.cc/paper/2018/hash/a41b3bb3e6b050b6c9067c67f663b915-Abstract.html.

How to improve MLP?

- For successful optimization
  - **Don't let the network stop learning prematurely!**
    - For example: Don't let the neurons saturate!
      - If the input or the gradient goes to zero, the learning stops!
  - Here is the gradient descent based weight update formula for a 2 layer MLP

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} = \alpha \delta_k z_j$$

Final layer weight update

$$\delta_k = (t_k - y_k) f'(y\_in_k)$$

Final layer backprop term

$$\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \alpha \hat{\delta}_j x_i$$

Hidden layer weight update

$$\hat{\delta}_j = \delta_{in_j} a'\left(z_{in_j}\right)$$

Hidden layer backprop

$$\delta_{in_j} = \sum_k \delta_k w_{jk}$$

$$z_j = a(z\_in_j), z_{in_j} = \sum_{i=0}^{n} x_i v_{ij}$$

Hidden layer output

$$y_k = a(y\_in_k), y_{in_k} = \sum_{j=0}^{p} z_j w_{jk}$$

Final layer output

$$\Delta v_{ij} = \alpha x_i a'(\boldsymbol{v}_j^T \boldsymbol{x}) \sum_{k=1}^{m} w_{jk} \left(t_k - a\left(\sum_{j=0}^{p} w_{jk} a(\boldsymbol{v}_j^T \boldsymbol{x}) \right) \right) a'\left(\sum_{j=0}^{p} w_{jk} a(\boldsymbol{v}_j^T \boldsymbol{x}) \right)$$

Understanding optimization stalls in neural networks

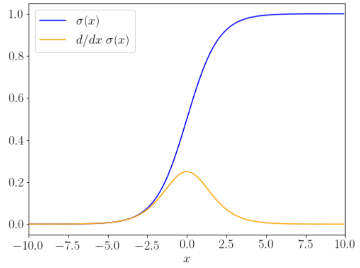
$$\Delta v_{ij} = \alpha x_i a'(\boldsymbol{v}_j^T \boldsymbol{x}) \sum_{k=1}^{m} w_{jk} \left(t_k - a\left(\sum_{j=0}^{p} w_{jk} a(\boldsymbol{v}_j^T \boldsymbol{x})\right)\right) a'\left(\sum_{j=0}^{p} w_{jk} a(\boldsymbol{v}_j^T \boldsymbol{x})\right)$$

Weight updates:
- When do we want them to be zero?
  - When all outputs target: $t_k - y_k = 0$
- When can they unwantedly be zero?
  - Leading to learning stall!

Why can optimization stall or slow down

1. When $x_i = 0$ (input is zero or too small)
2. Activation gradient $a'(\cdot)$ is small for a given input

Sigmoid activation and its derivative



3. Weights are close to zero $w_{jk} = 0$
4. When weight updates get too large, the next weights are going to be large leading to saturation (exploding gradients)
5. When the neural network output range cannot match the range of the target

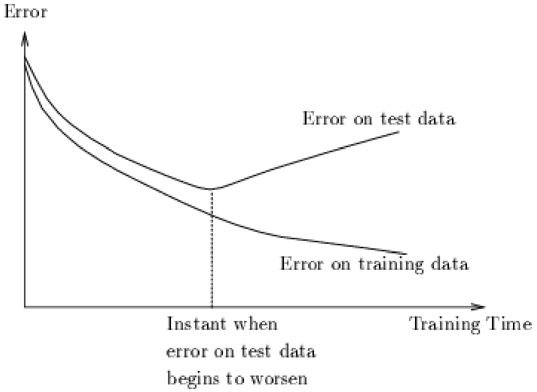
How to fix / Good practice

1. Don't use zero inputs (scale neuron inputs appropriately)
   Scale neuron outputs appropriately too as they become inputs to other neurons.
2. Either large inputs or large weights can push the activation function into saturation
   - Don't use "saturating" activation functions (leaky-RelU better than ReLU or sigmoid)
   - Don't use very large inputs (use appropriate input and output scaling)
   - Don't let weights get large
   - Each layer in a neural network introduces an additional product term of gradients of the activation function. If a neural network has many layers, there will be many products of activation function gradients and as the product of small numbers is even smaller, small gradients will just vanish and lead to a learning stall
     - **Vanishing gradients problem**
     - Don't use too many layers!
3. Don't start with zero weights (use proper weight initialization with small random weights – implicit regularization)
4. Choose the learning rate/optimizer appropriately. Plot the convergence plot. Use gradient clipping.
5. Choose an appropriate activation in the output layer

Glorot & Bengio 2010 "Understanding the difficulty of training deep feedforward neural networks".

Improving MLP

- Improving optimization
    - Different optimizers
        - Adaptive Momentum based optimization
        - Learning rate cycling strategies
- Improving generalization
    - Use Early Stopping
        - Keep track of generalization error and stop if the generalization error does not improve enough even when the error on training data is going down
    - Using regularization
        - Explicit regularization
            - Weight norms
            - Gradient clipping
        - Data Augmentation
            - Create artificial examples
                - Addition of noise
                - Translation of images or other transforms
        - Drop-Off
        - Batch Normalization
- The loss function has a significant impact on learning (both optimization and regularization)
    - For example cross-entropy loss and softmax work well for classification tasks



```
# Early stopping parameters
patience = 10  # How many epochs to wait after last time validation
loss improved.
best_loss = None
epochs_no_improve = 0
early_stop = False

for epoch in range(100):  # epochs
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    model.eval()
    val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            val_loss += criterion(outputs, labels).item()

    val_loss /= len(val_loader)
    print(f'Epoch {epoch}, Validation Loss: {val_loss}')

    # Check for early stopping
    if best_loss is None:
        best_loss = val_loss
    elif val_loss < best_loss:
        best_loss = val_loss
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        if epochs_no_improve == patience:
            print('Early stopping!')
            early_stop = True
            break  # Exit from the loop

if not early_stop:
    print('Training completed without early stopping.')
```

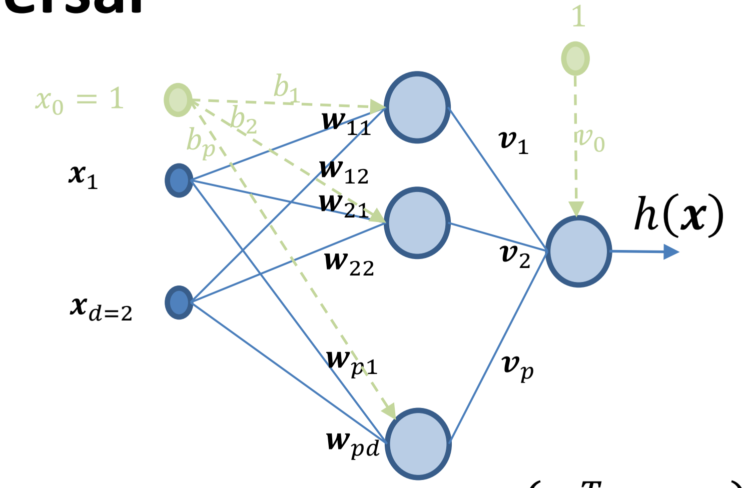
Home/Lab Exercise!

- Solve the XOR using a single hidden layer BPNN with sigmoid activations
  - See what is the effect of different parameters on the convergence characteristics of the neural network

UNIVERSAL FUNCTION APPROXIMATION WITH NEURAL NETWORKS

Universal Function Approximation

- **A neural network with a single hidden layer is a universal approximator**



$$z_i = a\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right)$$

- Universal Approximation
  - Any function $\mathrm{g}(\boldsymbol{x})$ over $\boldsymbol{x} \in \boldsymbol{R}^{\boldsymbol{m}}$ can be represented as follows:

$$h(\boldsymbol{x}) = \sum_{i=1}^{P} \boldsymbol{v}_i a\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right) + v_0 = \sum_{i=1}^{P} \boldsymbol{v}_i z_i + v_0$$

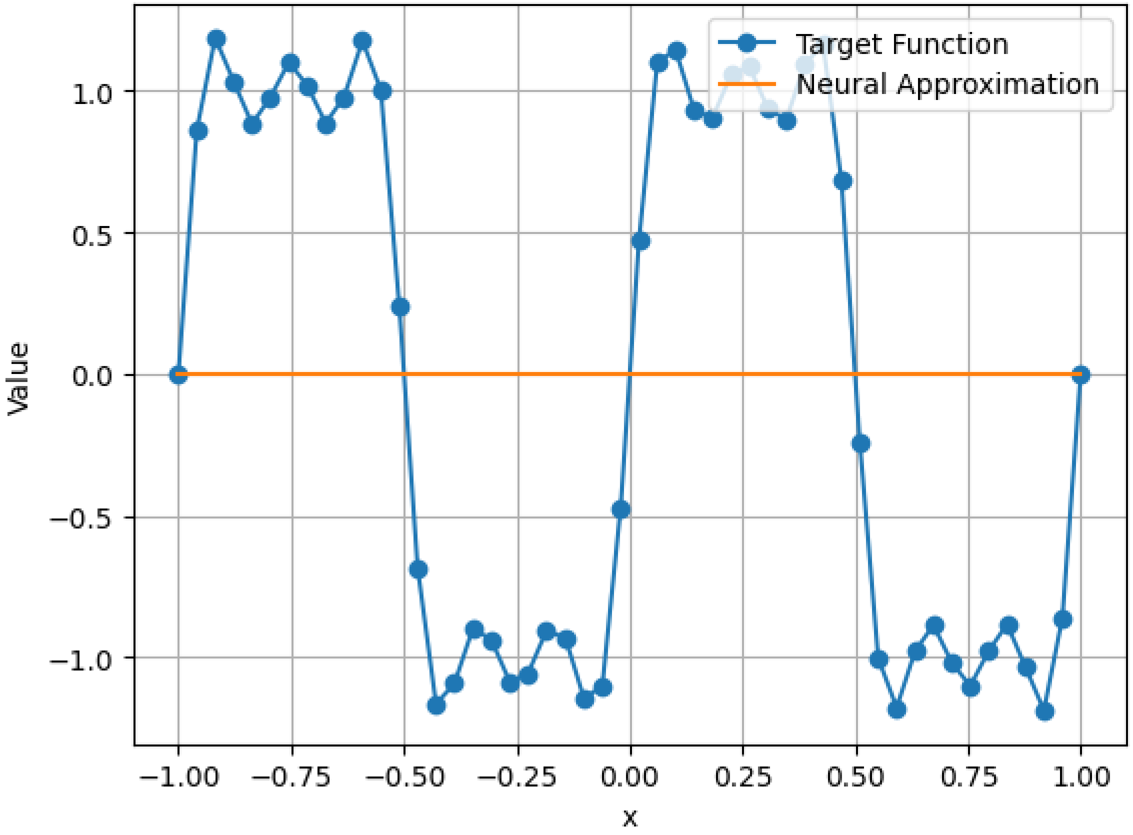
    - $a(\cdot)$ is a non-constant, bounded and monotonically-increasing continuous "basis" function
    - P is the number of functions
    - $h(\boldsymbol{x})$ is an approximation of $g(\boldsymbol{x})$, i.e., $|g(\boldsymbol{x}) - h(\boldsymbol{x})| < \epsilon$

https://en.wikipedia.org/wiki/Universal_approximation_theorem

Universal Function Approximation Example

- Let's try to approximate the function g(x) by a NN
- Let's build a neural network with sigmoid activations in the hidden layer
- The output of a single neuron depends on its net input which is a weighted summation of its inputs (with bias)
- The output is the sum of the outputs of all hidden neurons
- We want to find weights which sum up to produce the target function

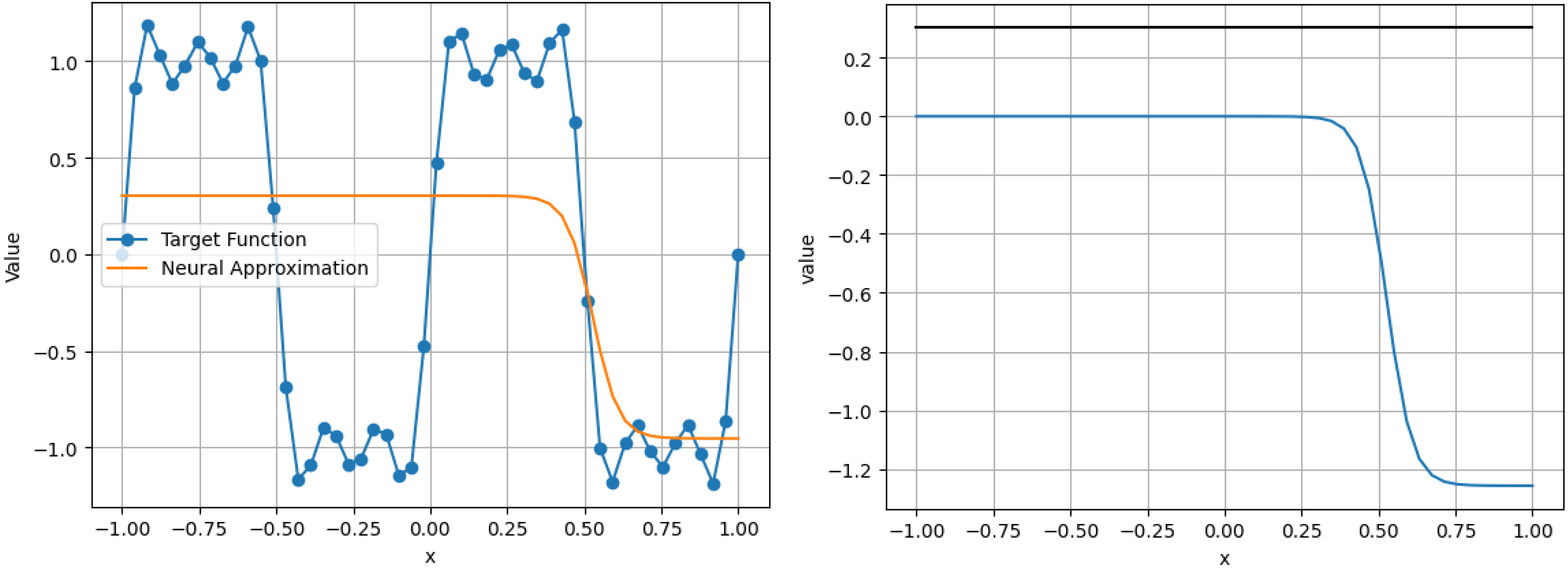
With no hidden layer neuron (P=0)



[CODE: https://github.com/foxtrotmike/CS909/blob/master/uniapprox.ipynb](https://github.com/foxtrotmike/CS909/blob/master/uniapprox.ipynb)

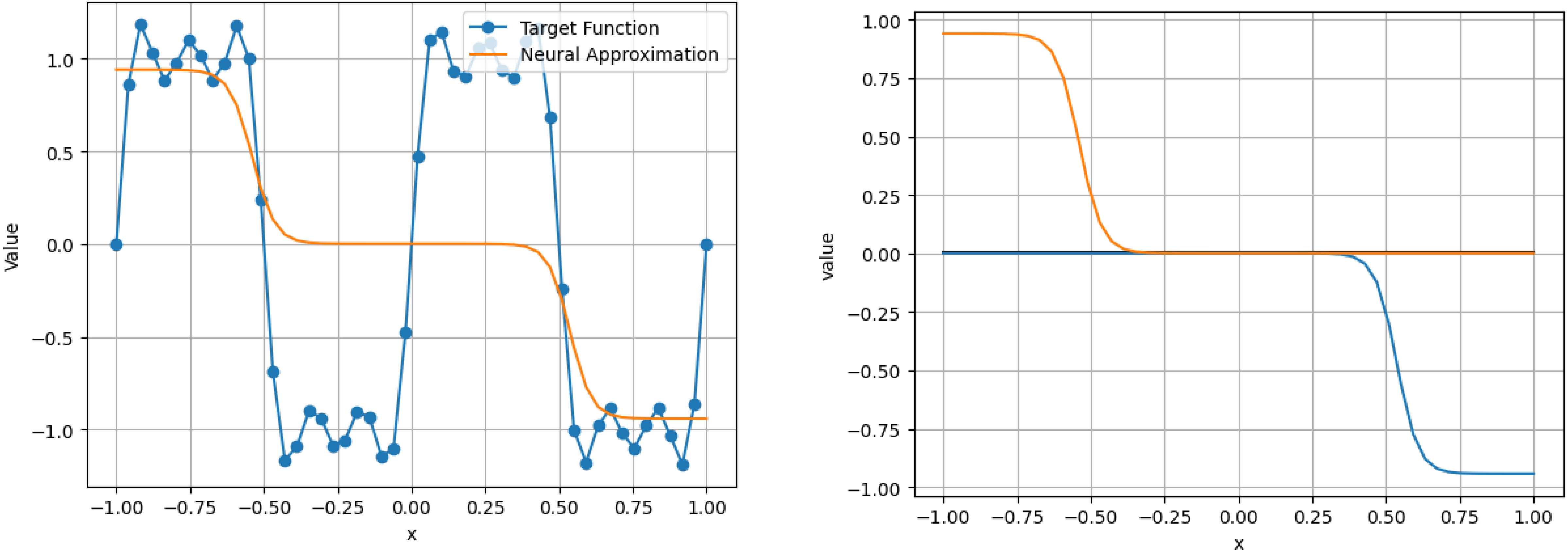
Universal Function Approximation Example

- With no hidden layer neuron (P=1)



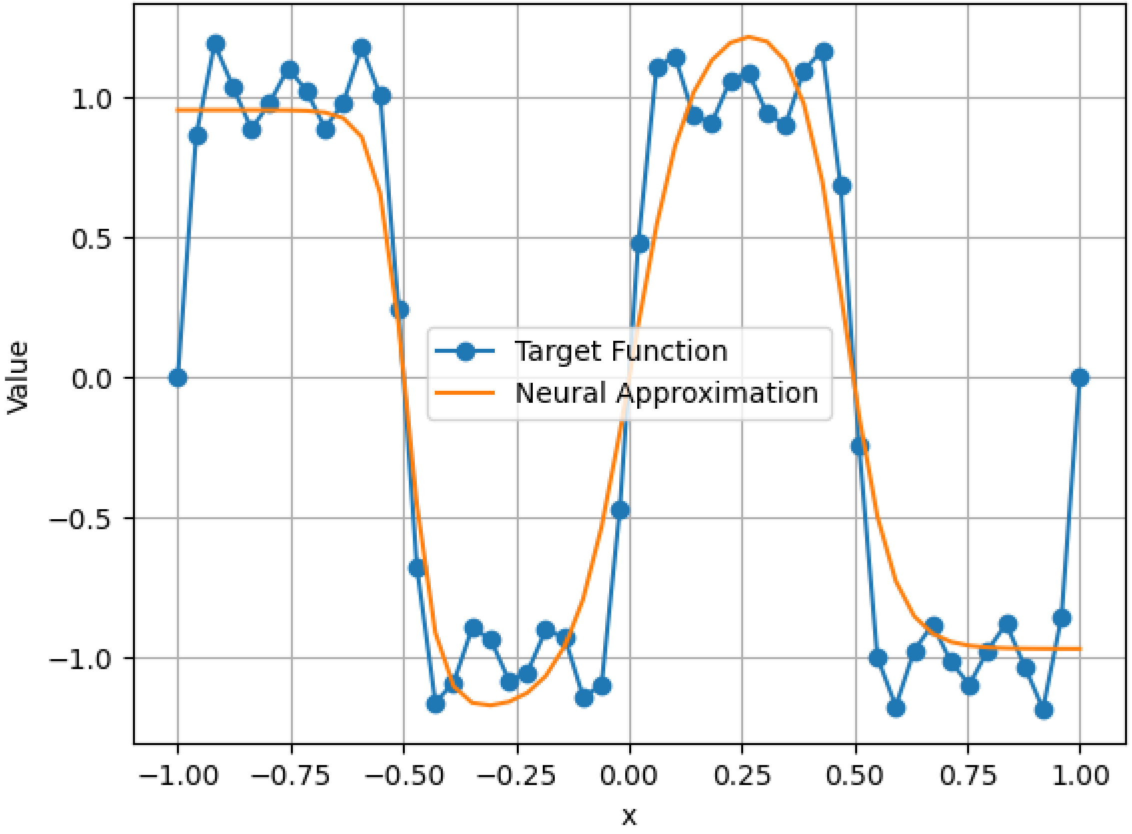
Universal Function Approximation Example

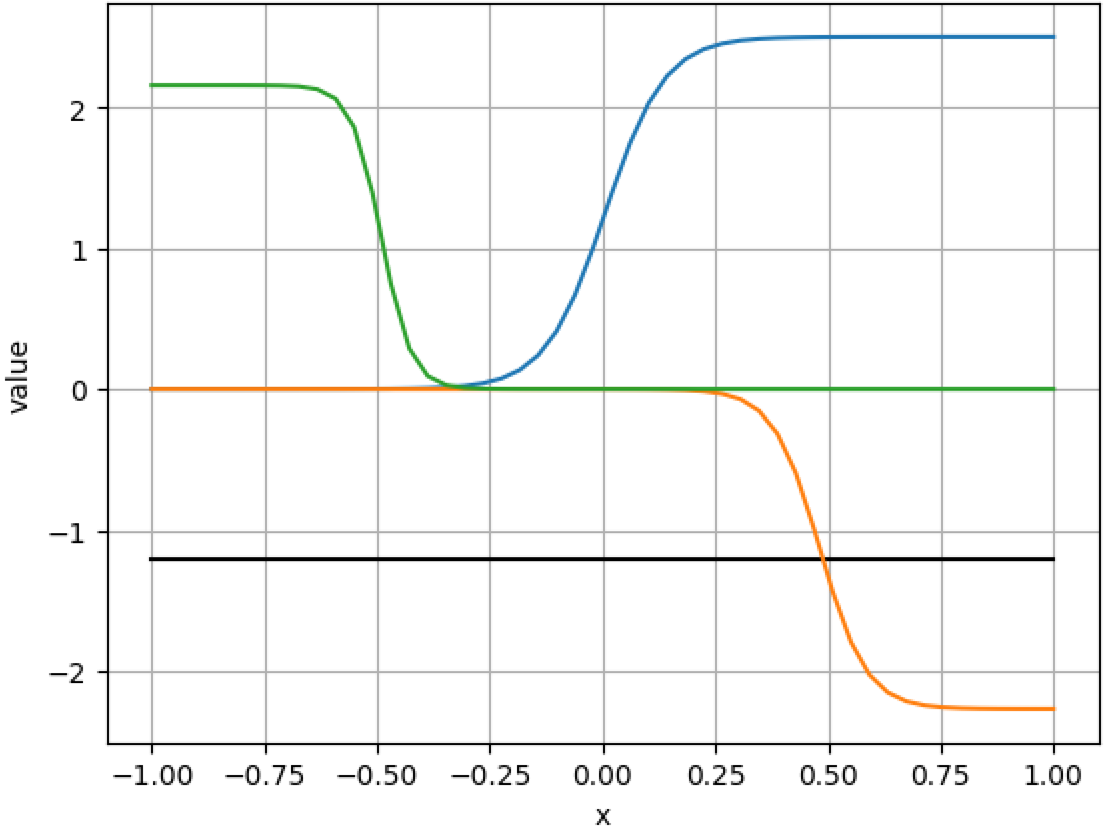
- With no hidden layer neuron (P=2)



Universal Function Approximation Example

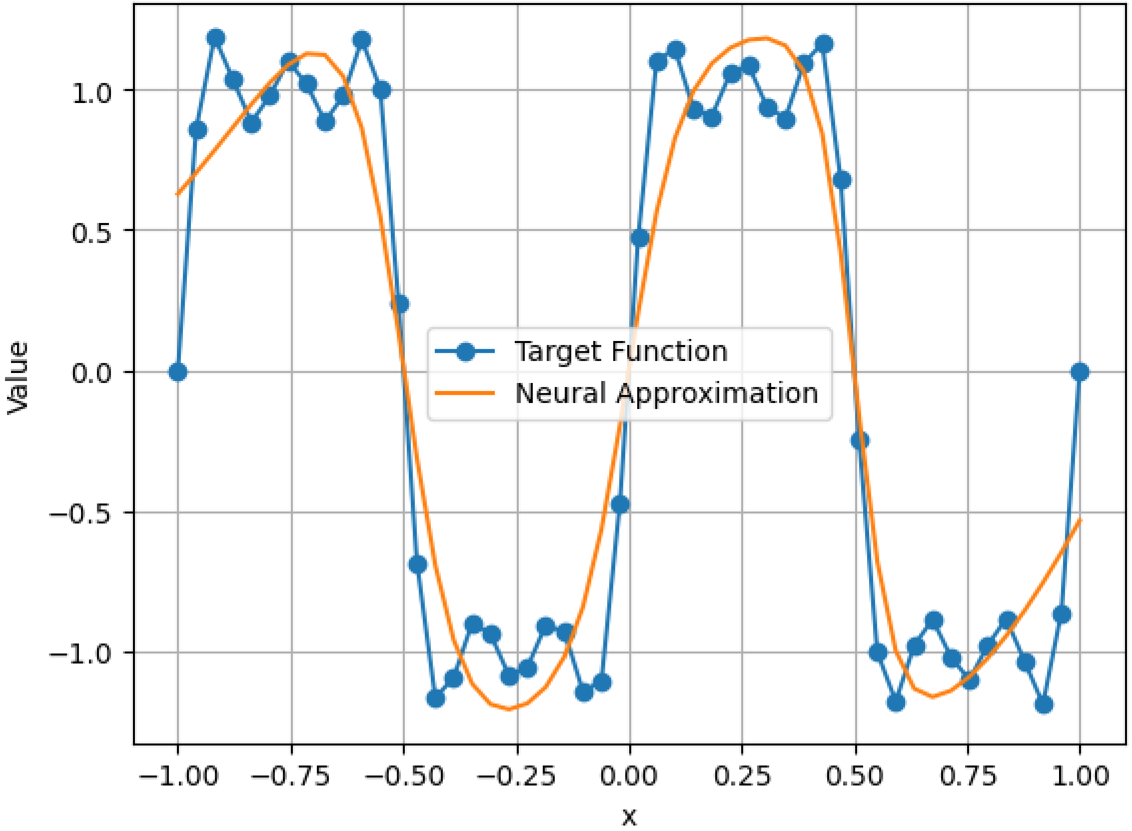
- With no hidden layer neuron (P=3)

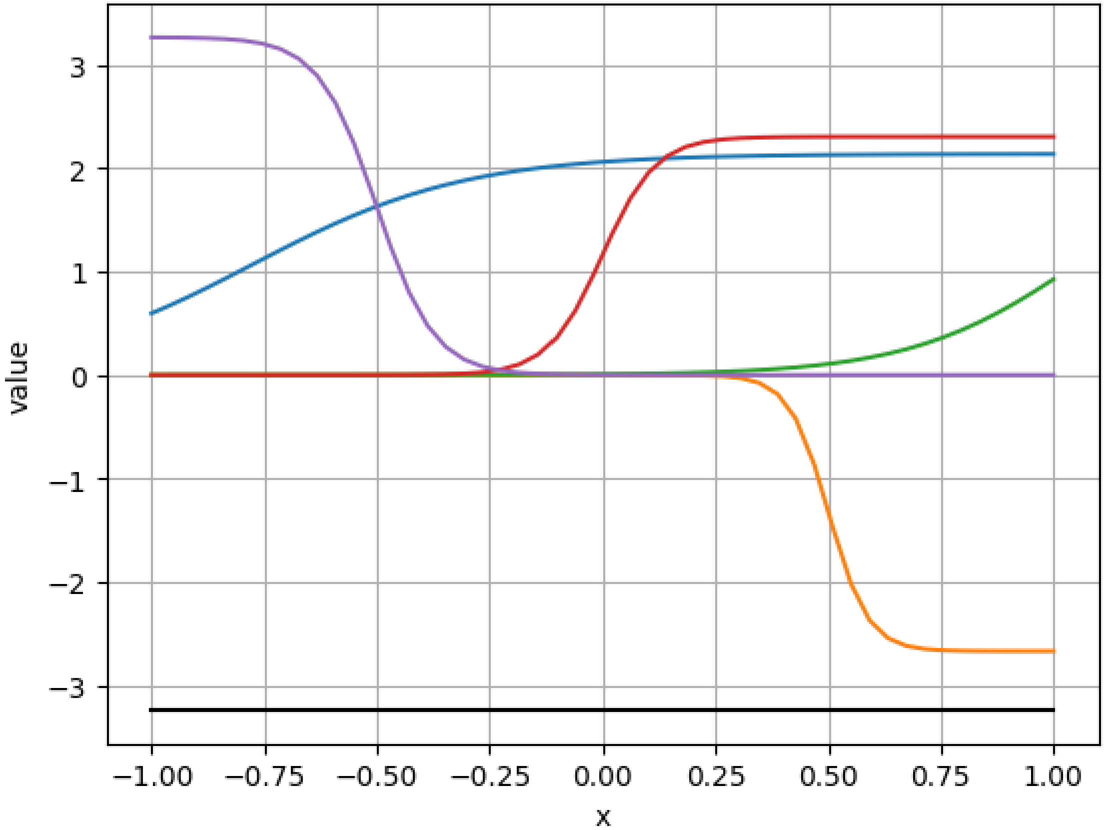




Universal Function Approximation Example

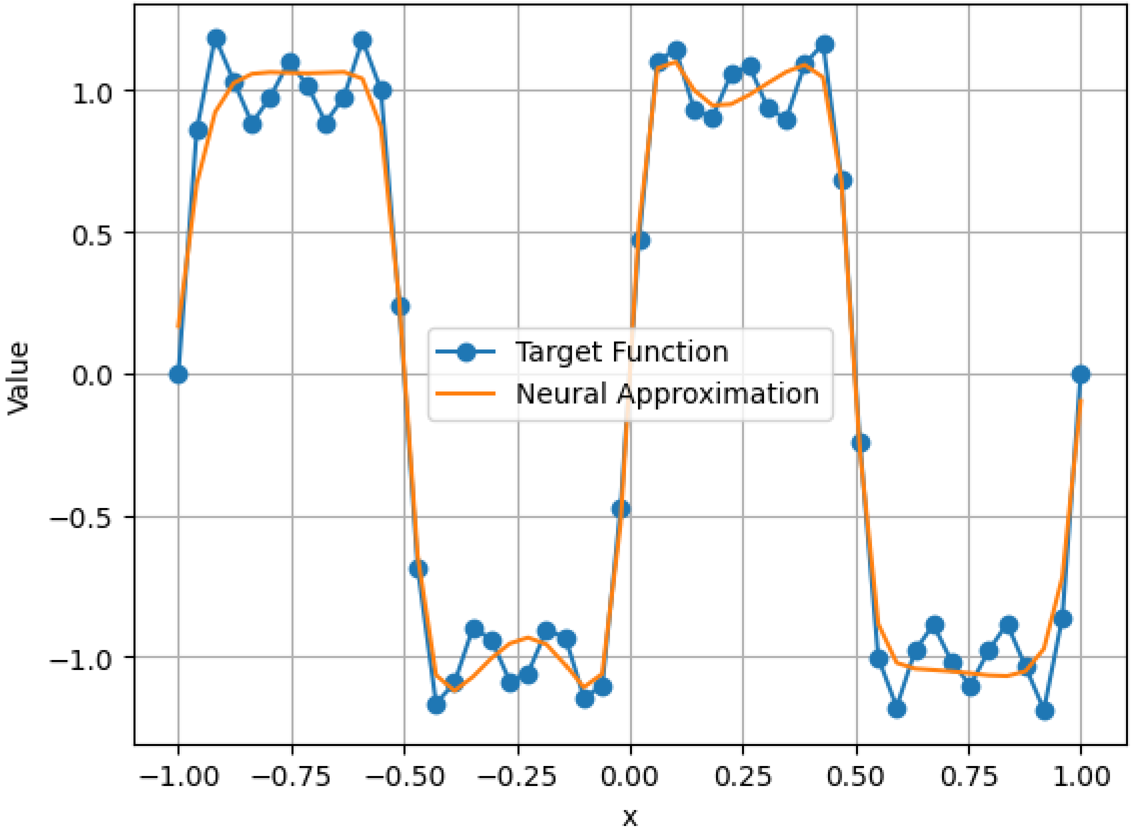
- With no hidden layer neuron (P=5)

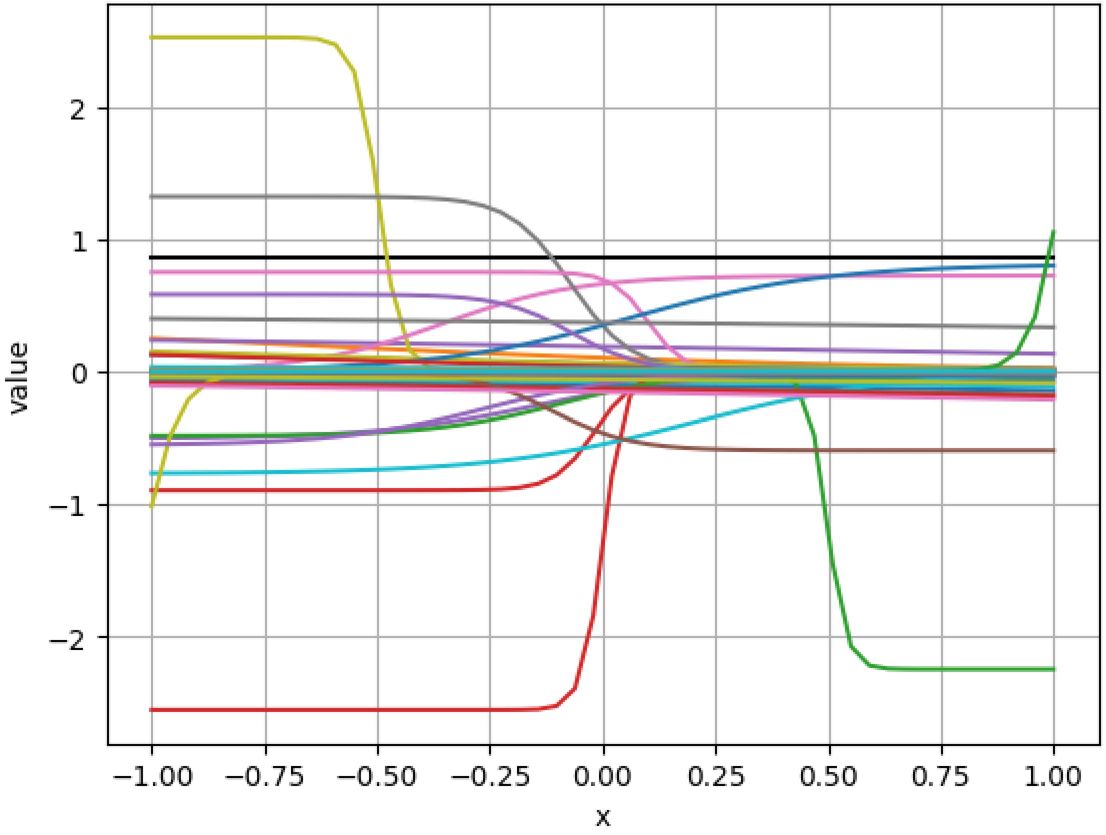




Universal Function Approximation Example

- With no hidden layer neuron (P=50)





Practical Issues in Universal Approximation

- The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function.
- However, we are not guaranteed that the training algorithm will be able to "learn" that function.
  - Optimization can fail
  - Learning is different from optimization
    - The primary requirement for learning is generalization

  - Representability alone does not guarantee learning

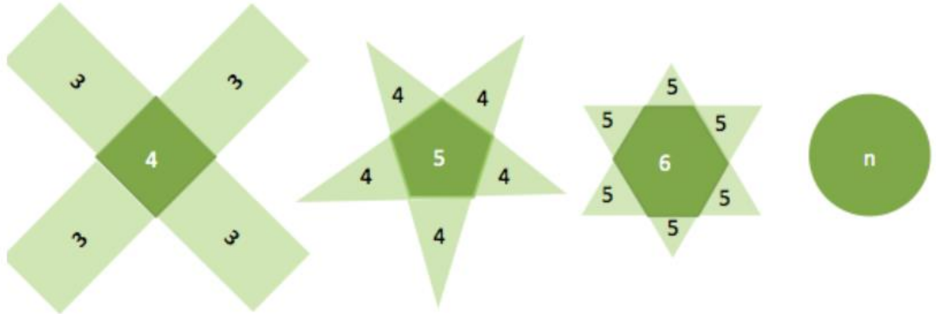
Universal Function Approximation

- A neural network with one hidden layer can be used to approximate any shape
  - However, the approximation might require exponentially many neurons
  - How can we reduce the number of computations?

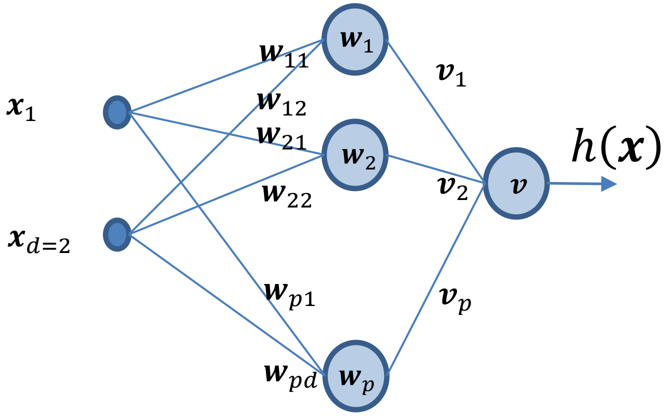
$$h(\boldsymbol{x}) = \sum_{i=1}^{p} \boldsymbol{v}_i a\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right)$$

A single hidden layer NN with step activation is a combination of straight cuts

Total number of learnable parameters: pd+p+p



The number of required straight cuts to approximate a given shape

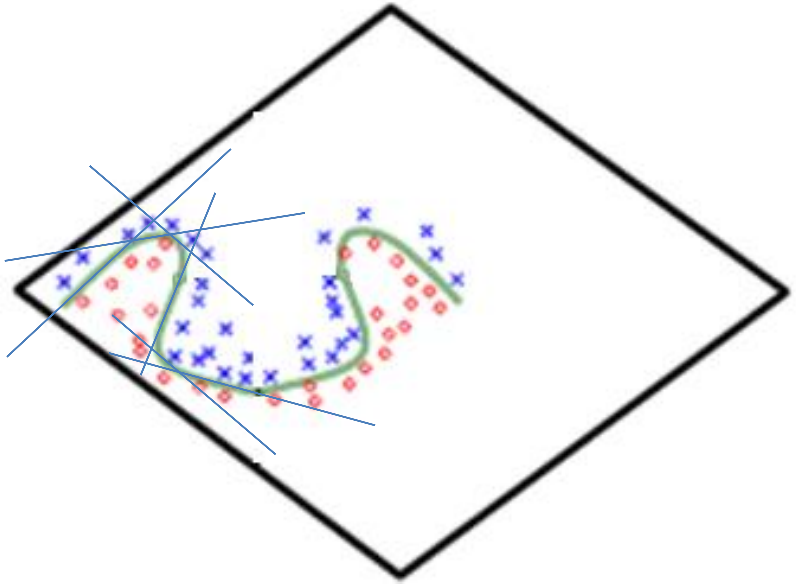


Wang, Haohan, and Bhiksha Raj. "On the Origin of Deep Learning." *arXiv:1702.07800 [Cs, Stat]*, February 24, 2017. http://arxiv.org/abs/1702.07800.

WHY GO DEEP?

How many cuts?

- Remember: Classification can be thought of as partitioning of the feature space
- How can we reduce the number of required cuts?
  - By folding: which is equivalent to:
    - Applying a transformation $\boldsymbol{\phi}(\boldsymbol{x})$
      - Neural networks
    - Changing the distance metric
      - Distance metric learning
    - Kernelization
      - SVM



Each layer is a transformation of the input data

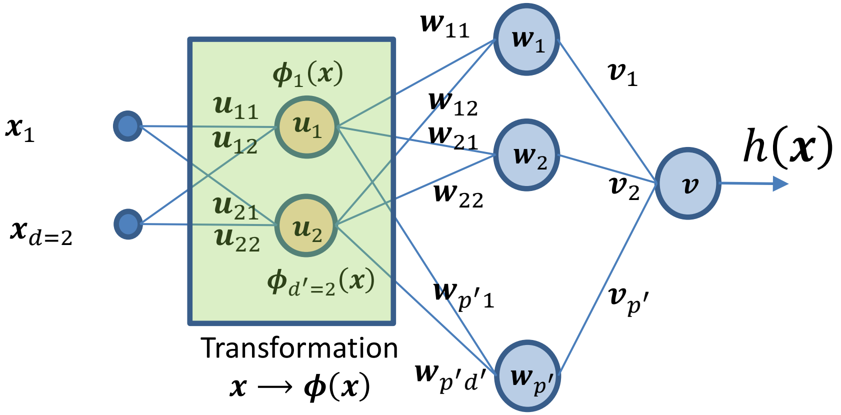
- In the transformed space

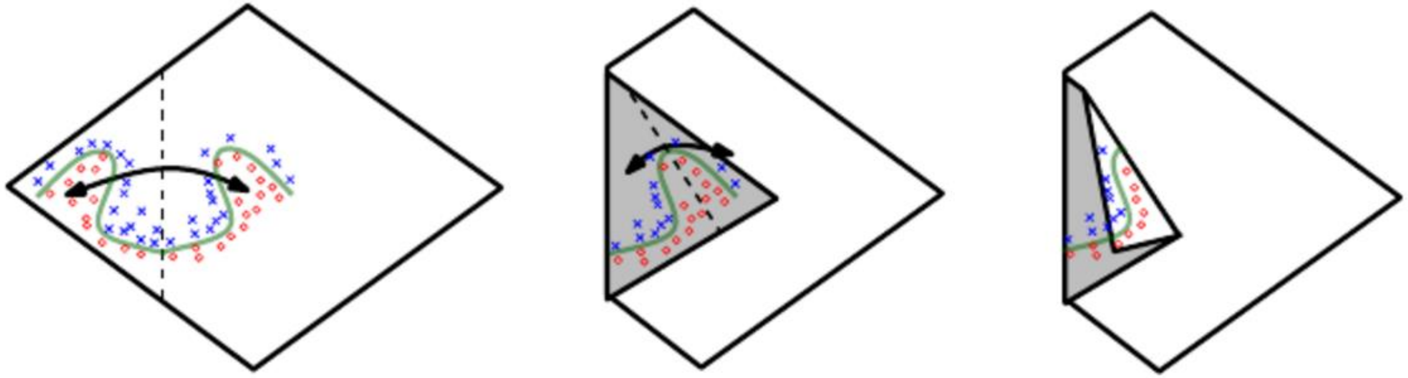
$$h(\boldsymbol{x}) = \sum_{i=1}^{p'} \boldsymbol{v}_i a\left(\boldsymbol{w}_i^T \boldsymbol{\phi}(\boldsymbol{x}) + b_i\right)$$

- We can implement a learnable feature transformation through neurons!

$$h(\boldsymbol{x}) = \sum_{i=1}^{p'} \boldsymbol{v}_i a\left(\sum_{j=1}^{d'} \boldsymbol{w}_{ij} g\left(\boldsymbol{u}_j^T \boldsymbol{x} + c_j\right) + b_i\right)$$

Total number of learnable parameters: dd’+d’+p’d’+p’+p’





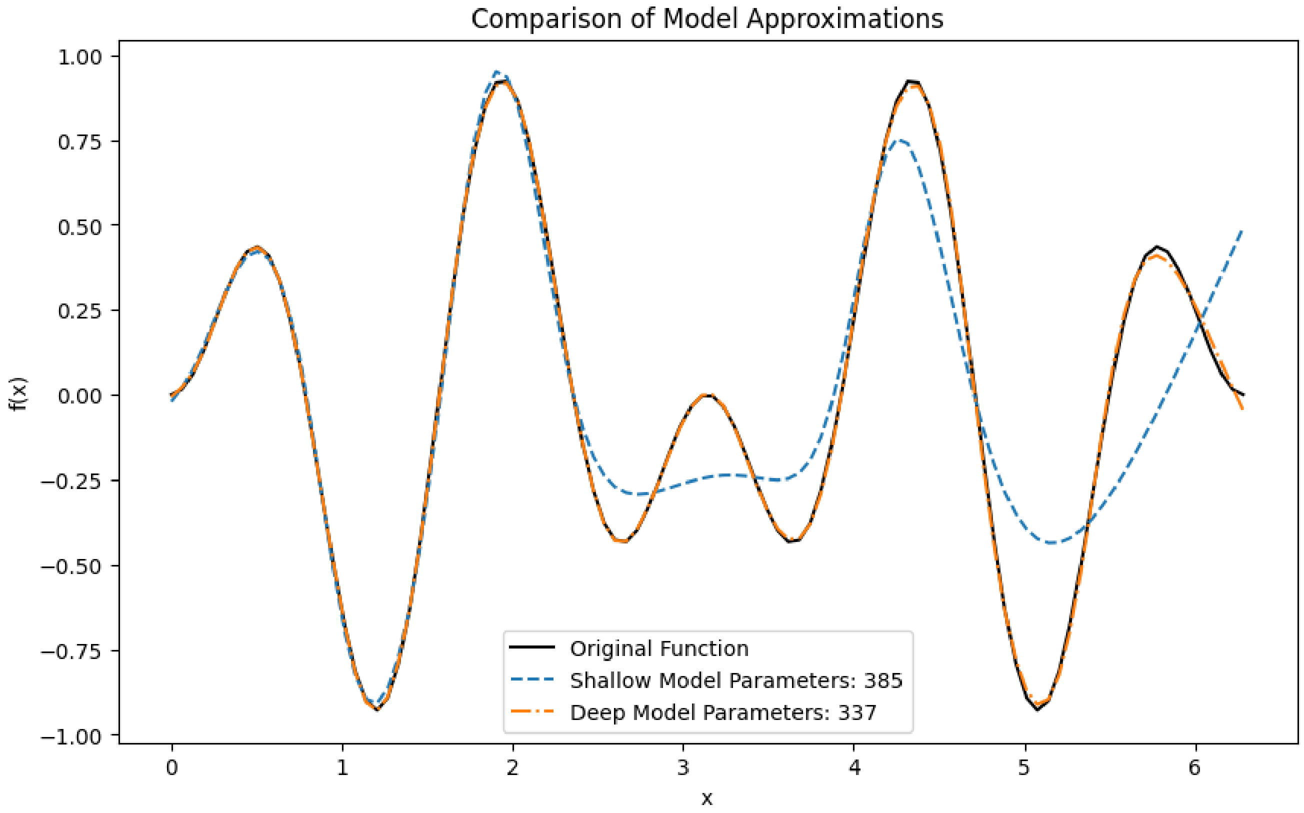
Montufar (2014)

Fold and Cut Theorem: https://www.youtube.com/watch?v=ZREp1mAPKTM

Width vs. Depth

- An MLP with a single hidden layer is sufficient to represent any function
  - But the layer may be infeasibly large
  - May fail to learn and generalize correctly
- Using a deeper model can reduce the number of units required to represent the desired function and can reduce the amount of generalization error
  - **Thus a deeper representation is more efficient!**
  - **A function that could be expressed with O(n) neurons on a network of depth k required at least $O(2^{\sqrt{n}})$ and $O((n-1)^k)$ neurons on a two-layer neural network**: Delalleau and Bengio (2011)
  - Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network: Montufar (2014)
  - For a shallow network, the representation power can only grow polynomially with respect to the number of neurons, but for deep architecture, the representation can grow exponentially with respect to the number of neurons: Bianchini and Scarselli (2014)
  - Depth of a neural network is exponentially more valuable than the width of a neural network, for a standard MLP with any popular activation functions: Eldan and Shamir (2015)

Comparison of Depth



```
nn.Sequential(
    nn.Linear(input_size, 128),
    nn.Sigmoid(),
    nn.Linear(128, output_size))

nn.Sequential(
    nn.Linear(input_size, 32),
    nn.Sigmoid(),
    nn.Linear(32, 8),
    nn.Sigmoid(),
    nn.Linear(8, output_size))
```

- Both have approximately the same number of parameters (tunable weigths)
  - Deeper is better
  - But is difficult to optimize

CODE: https://github.com/foxtrotmike/CS909/blob/master/uniapprox.ipynb

Width vs. Depth

- Empirical results for some data showed that depth increases generalization performance in a variety of applications

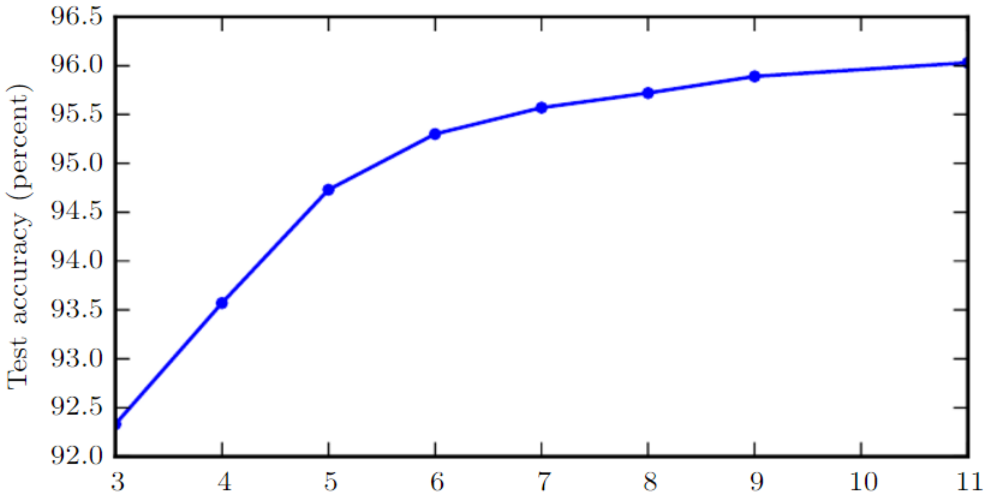


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

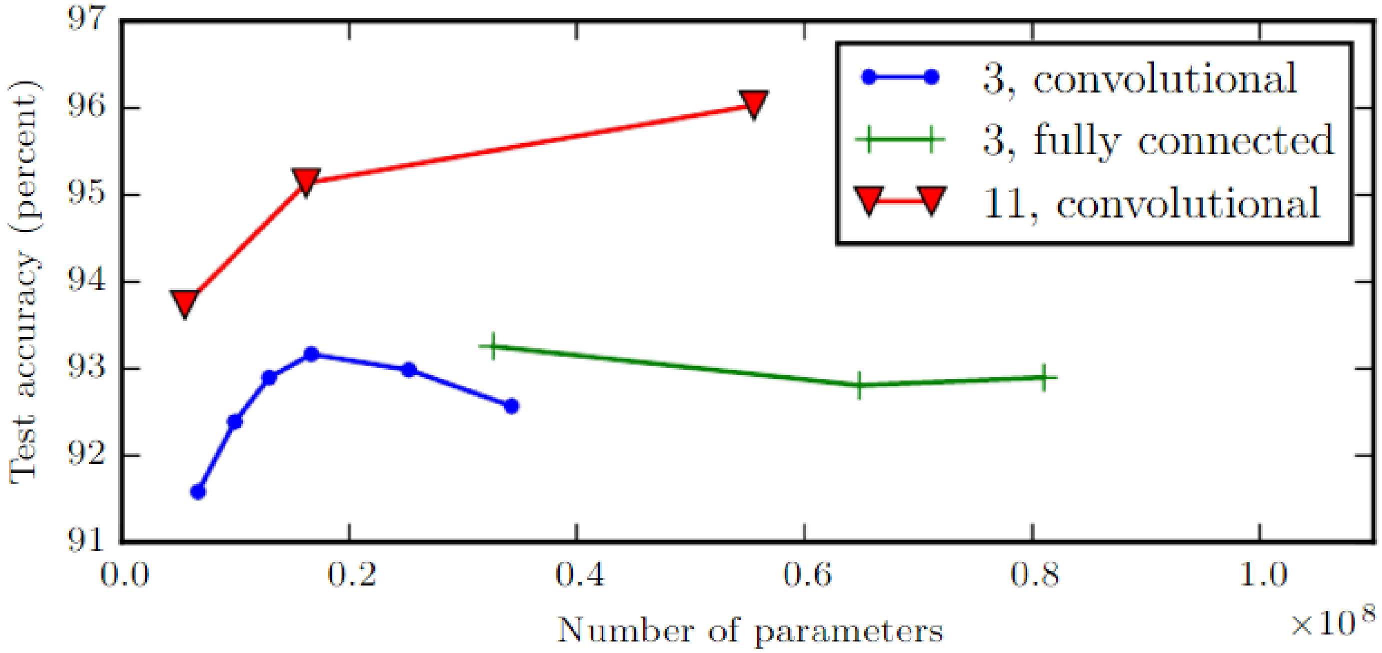


Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

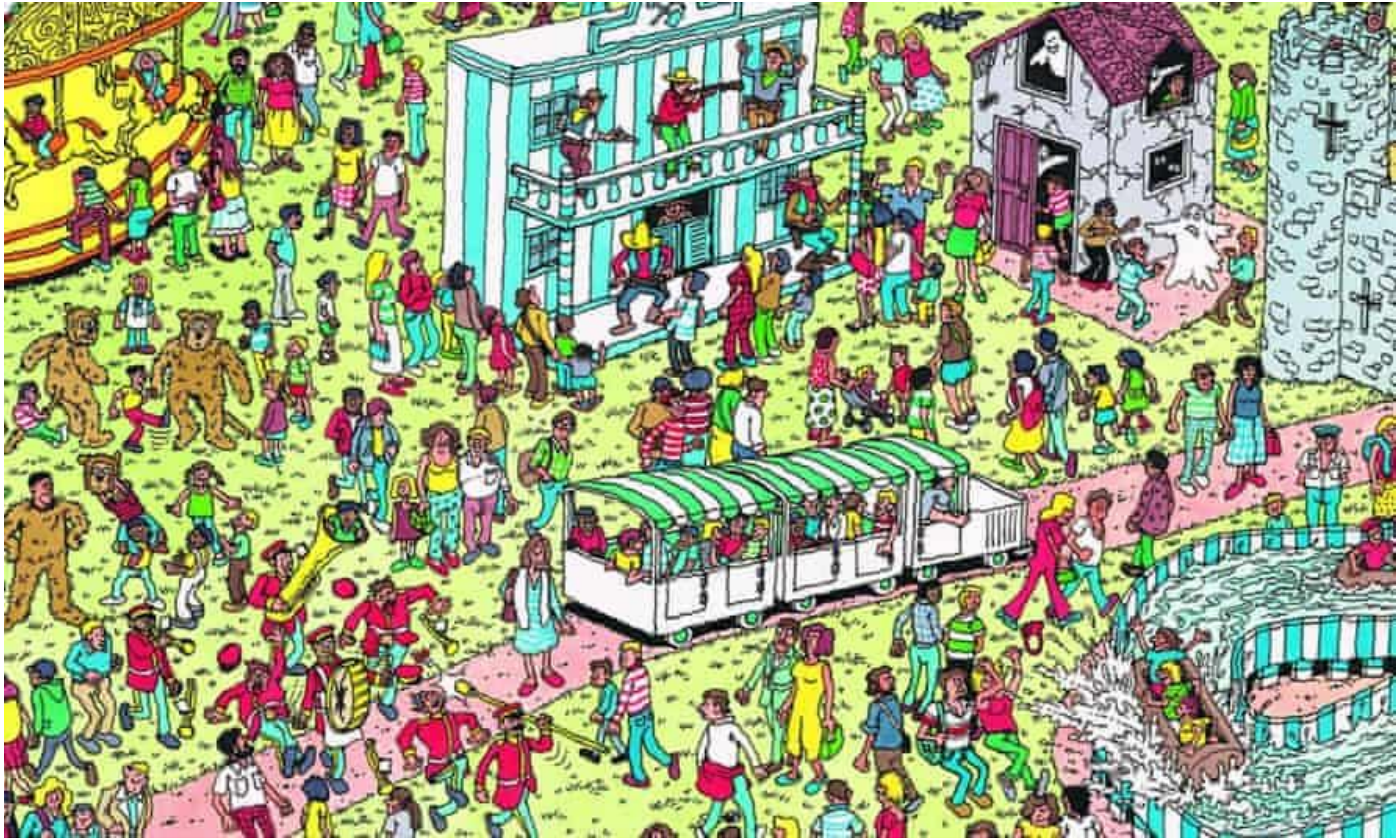
Shallow vs. Deep Networks

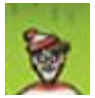
- Adding more layers increases the representation power of the neural network
- A deep network requires exponentially fewer parameters to get to the same error rate in comparison to a wide neural network
  - More efficient
- However, adding layers leads to a more difficult optimization problem
  - Vanishing and Exploding Gradients

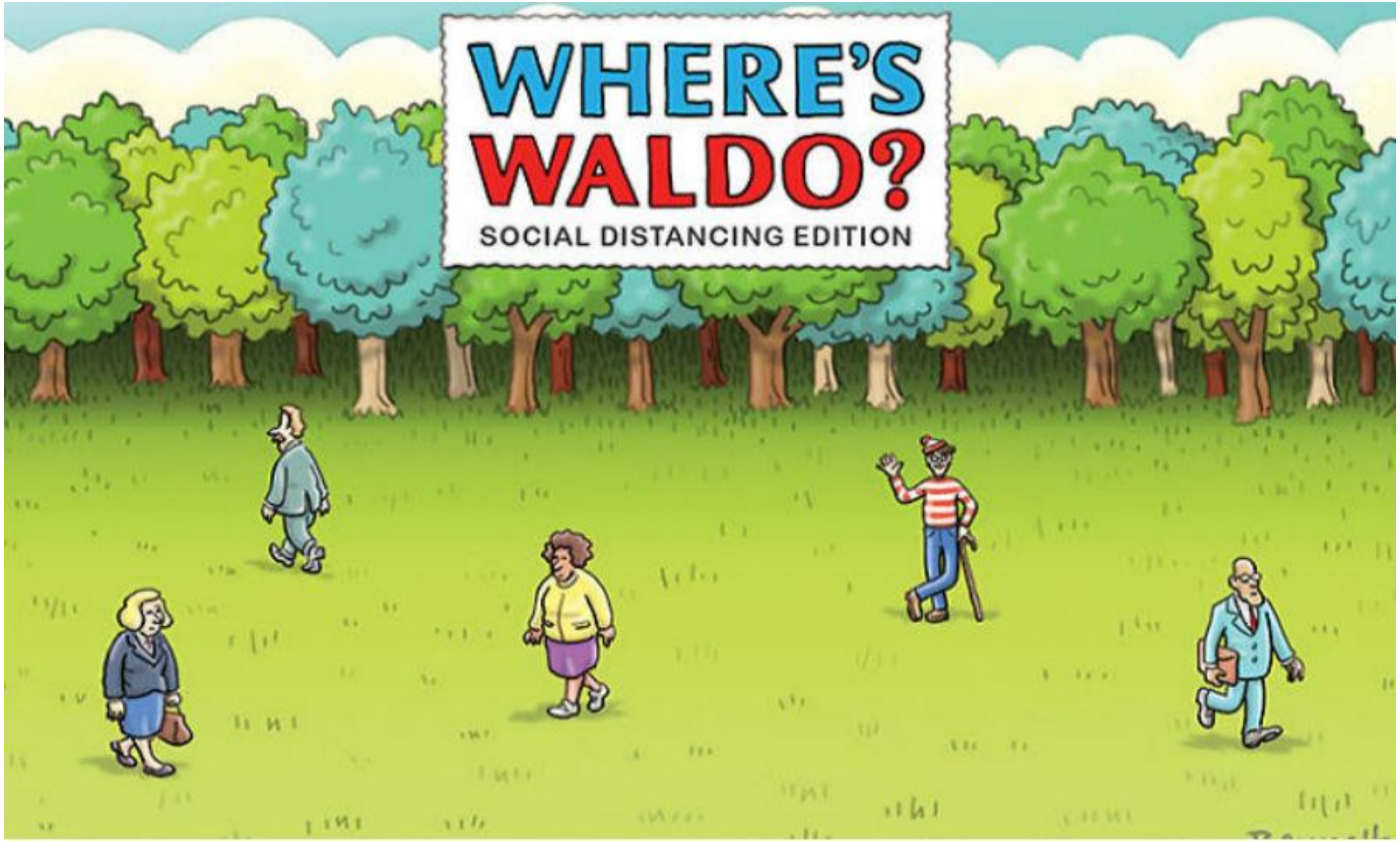
$$\Delta v_{ij} = \alpha x_i a'\left(\boldsymbol{v_j^T x}\right) \sum_{k=1}^{m} w_{jk} \left(t_k - a\left(\sum_{j=0}^{p} w_{jk} f\left(\boldsymbol{v_j^T x}\right) \right) \right) a'\left(\sum_{j=0}^{p} w_{jk} f\left(\boldsymbol{v_j^T x}\right) \right)$$

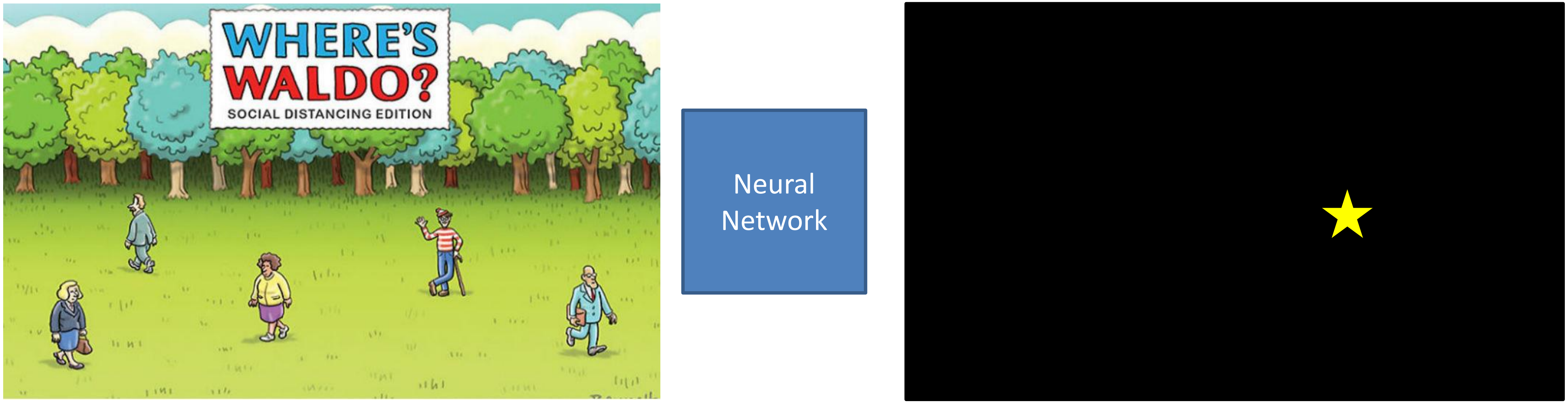
CONVOLUTIONAL NEURAL NETWORKS

Where’s Waldo?






WHERE'S
WALDO?
SOCIAL DISTANCING EDITION



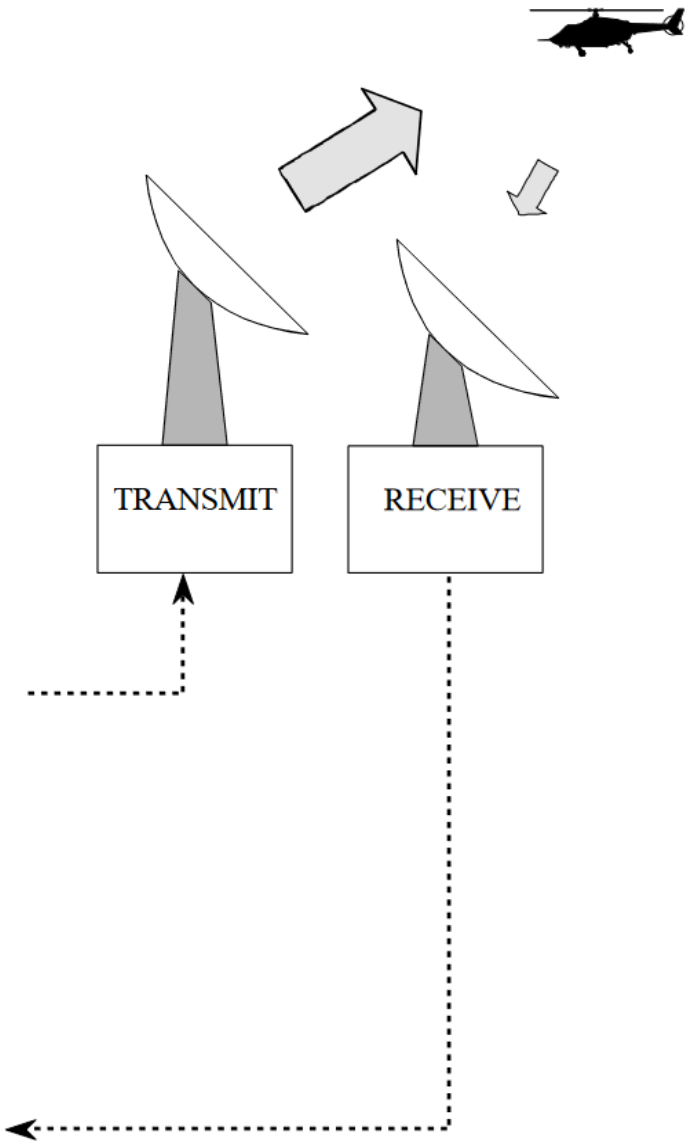
Neural Network

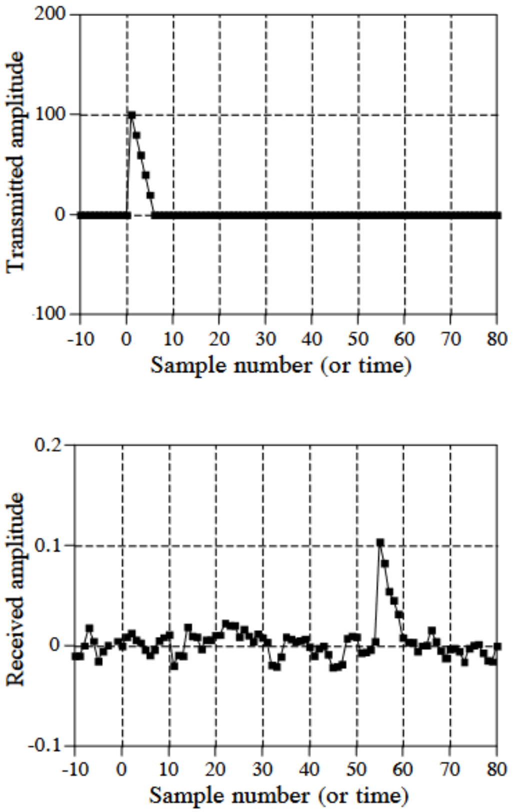
Let’s solve it using a neural network

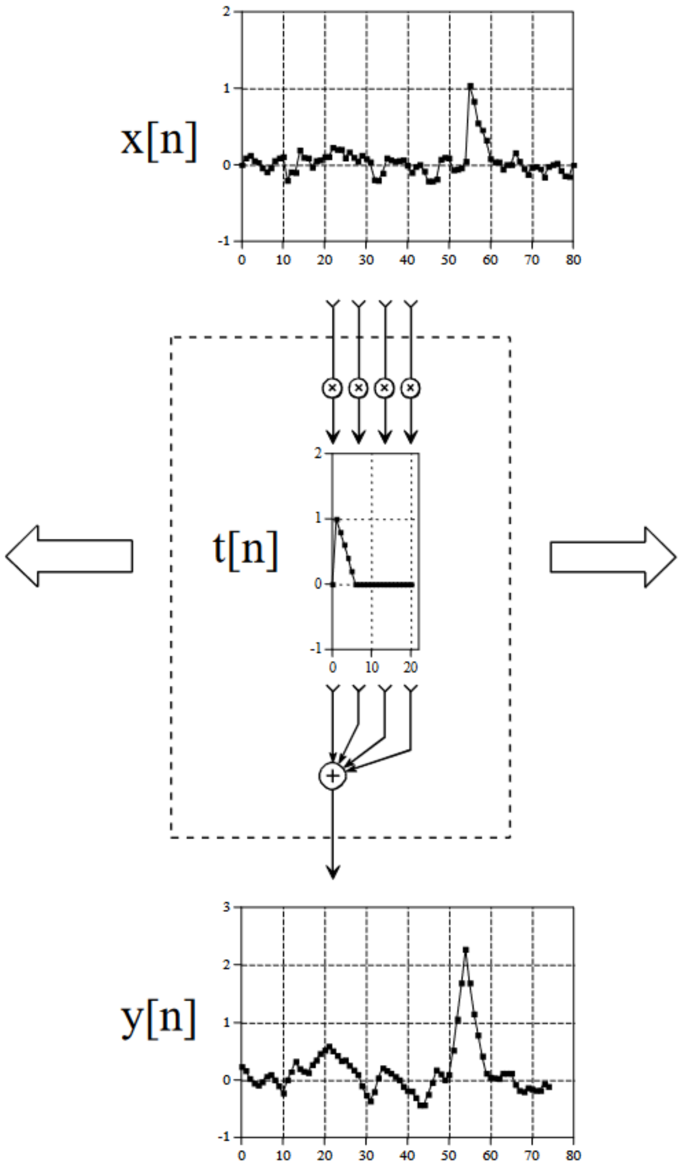
- Input image: 256x256x3
  - Flatten it: 196, 608 dimensional input
- Target: 256x256x3
  - Flatten it: 196, 608 dimensional output

- Let’s use a single hidden layer network
  - Very large number of parameters will be needed
- Let’s use a deep(er) network
  - Still a very large number of parameters will be needed

Key elements of a radar system. Like other echo location systems, radar transmits a short pulse of energy that is reflected by objects being examined. This makes the received waveform a shifted version of the transmitted waveform, plus random noise. Detection of a known waveform in a noisy signal is the fundamental problem in echo location. The answer to this problem is *correlation*.



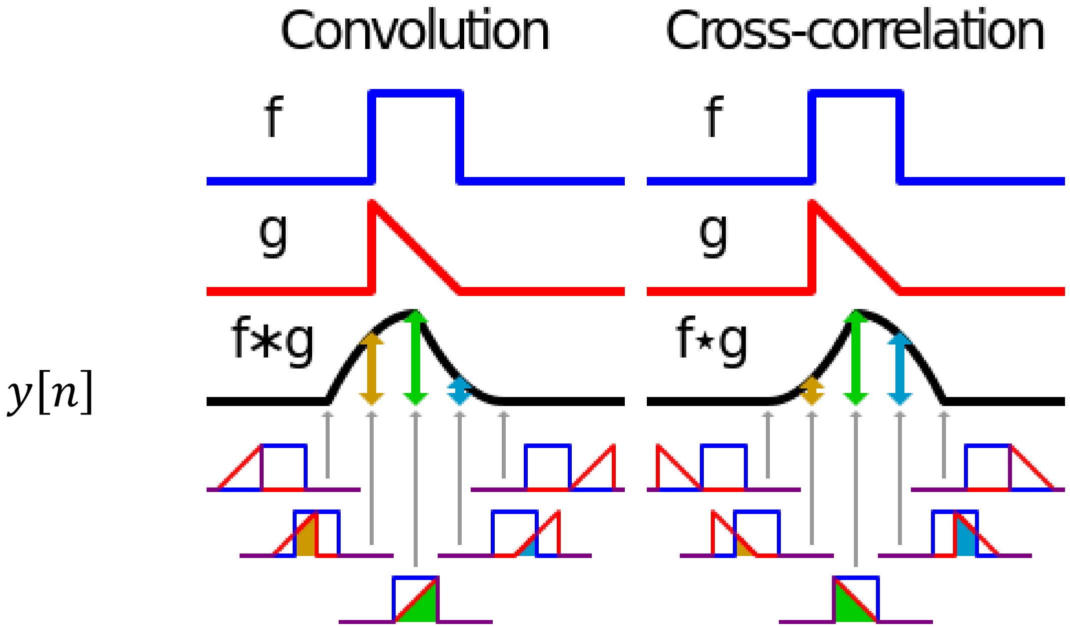




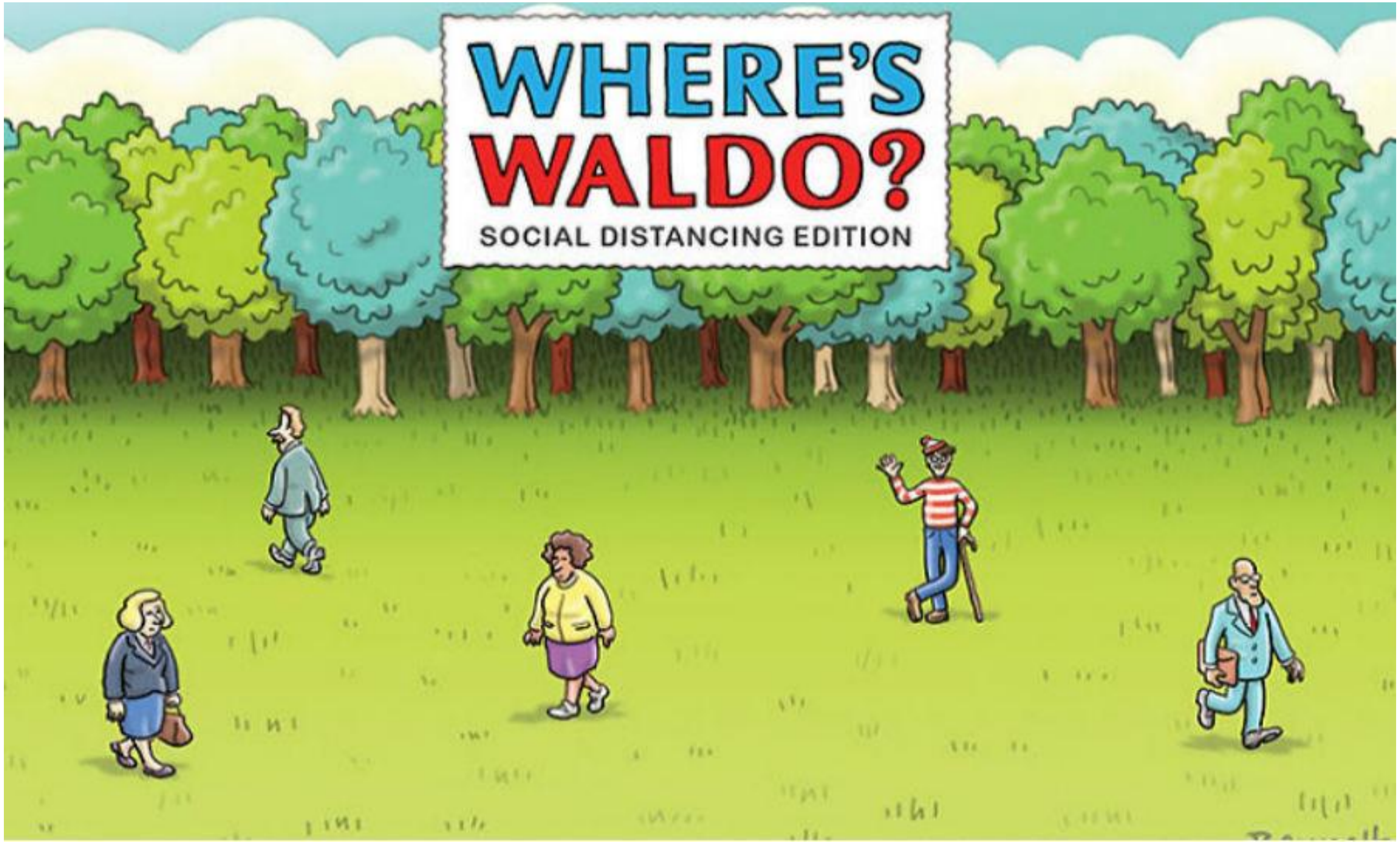
The correlation machine. This is a flowchart showing how the cross-correlation of two signals is calculated. In this example, $y[n]$ is the cross-correlation of $x[n]$ and $t[n]$. The dashed box is moved left or right so that its output points at the sample being calculated in $y[n]$. The indicated samples from $x[n]$ are multiplied by the corresponding samples in $t[n]$, and the products added.

Important conceptual note

- Correlation vs. convolution



$$y[n] = f * g = \sum_{k=-\infty}^{+\infty} f[k]g[n-k] \qquad y[n] = f \star g = \sum_{k=-\infty}^{+\infty} f[k]g[n+k]$$

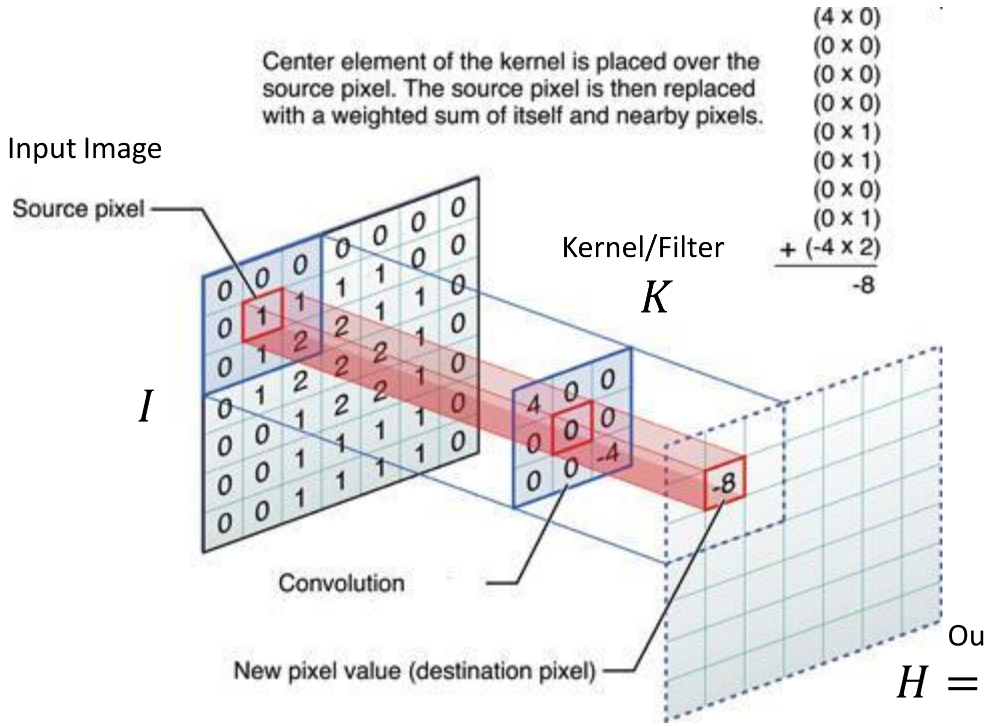

WHERE'S
WALDO?
SOCIAL DISTANCING EDITION

Convolutional Networks

- A feed-forward network inspired from visual cortex and the ideas of correlation
- Used for image or signal recognition tasks
- Objective
  - Find a set of filters which, when convolved with image, lead to the solution of the desired image recognition task
    - Invariant wrt translation
    - Hierarchical
      - Increasing feature complexity
      - Increasing “Globality”

Basics

- The convolution operation
  - Shows how a function (image) is modified by another (filter)



Output
$H = I \star K$

Technically, this is "correlation" and not convolution but we can ignore this for now. You can also use different edge handling or padding strategies.

$$H(i,j) = \sum_{k=-m/2}^{m/2} \sum_{l=-n/2}^{n/2} I(i+k, j+l)K(k,l)$$

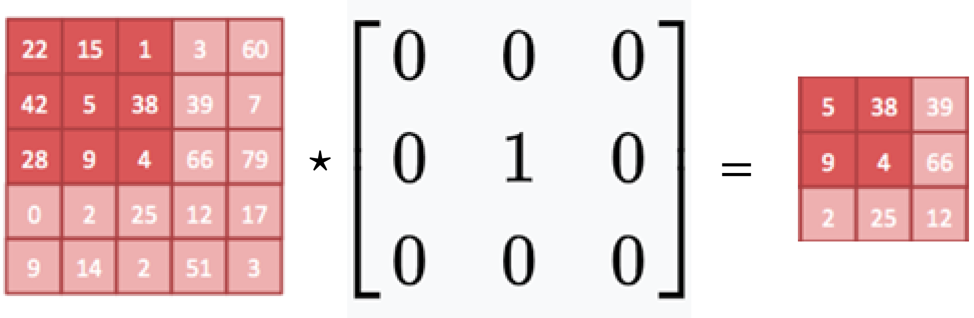
https://en.wikipedia.org/wiki/Kernel_(image_processing)

Examples of filters

- Identity Filter

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

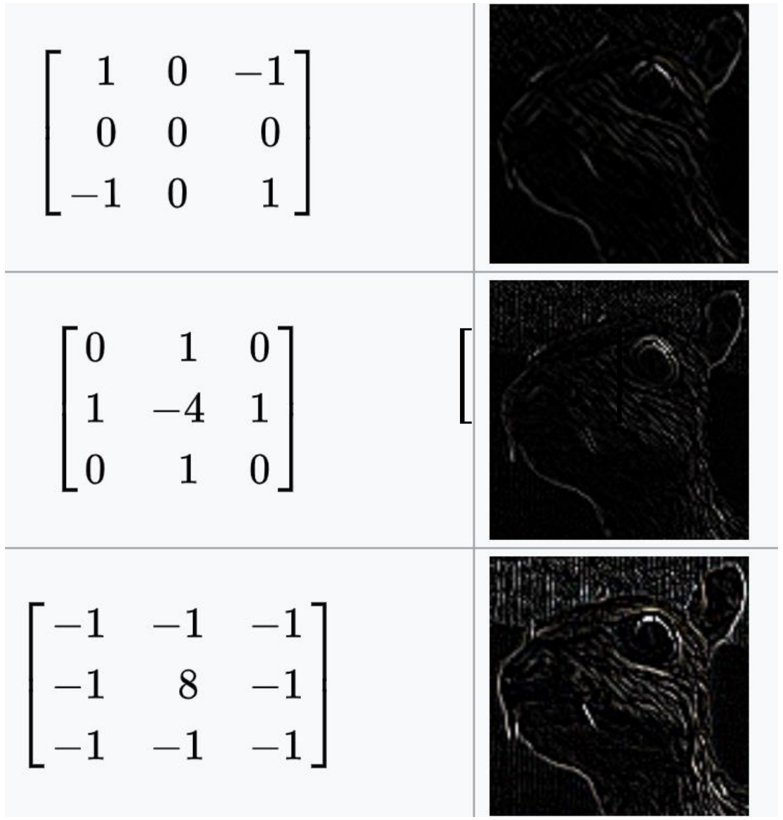
Muntjac



$$\begin{bmatrix} 22 & 15 & 1 & 3 & 60 \\ 42 & 5 & 38 & 39 & 7 \\ 28 & 9 & 4 & 66 & 79 \\ 0 & 2 & 25 & 12 & 17 \\ 9 & 14 & 2 & 51 & 3 \end{bmatrix} \star \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 38 & 39 \\ 9 & 4 & 66 \\ 2 & 25 & 12 \end{bmatrix}$$

Examples of filters

- Edge filters



$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

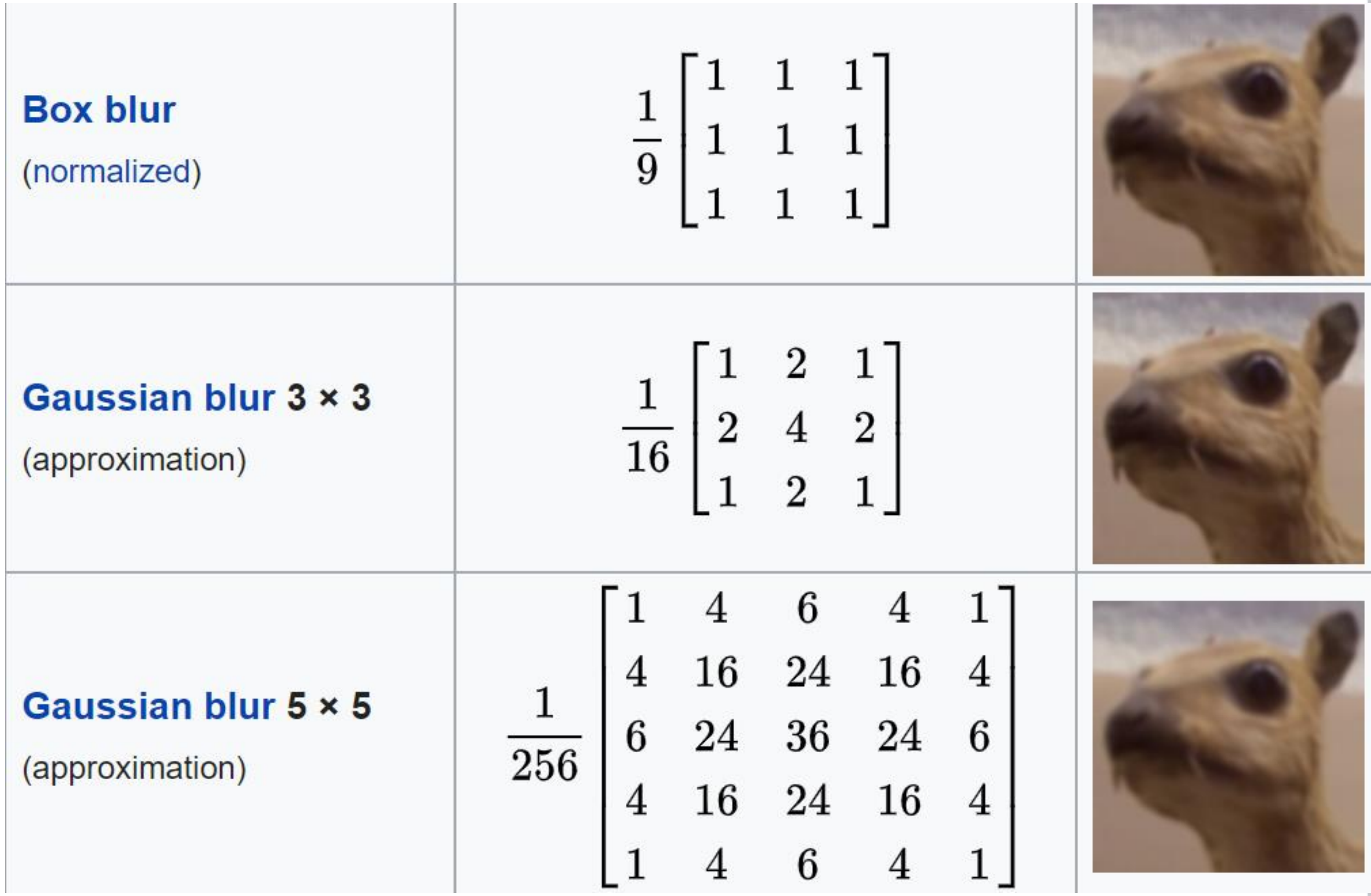
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 10 & 10 & 10 \\ 1 & 1 & 10 & 10 & 10 \\ 1 & 1 & 10 & 10 & 10 \\ 1 & 1 & 10 & 10 & 10 \end{bmatrix} * \mathrm{K} = \begin{bmatrix} 9 & -18 & -9 \\ 9 & -9 & 0 \\ 9 & -9 & 0 \end{bmatrix}$$

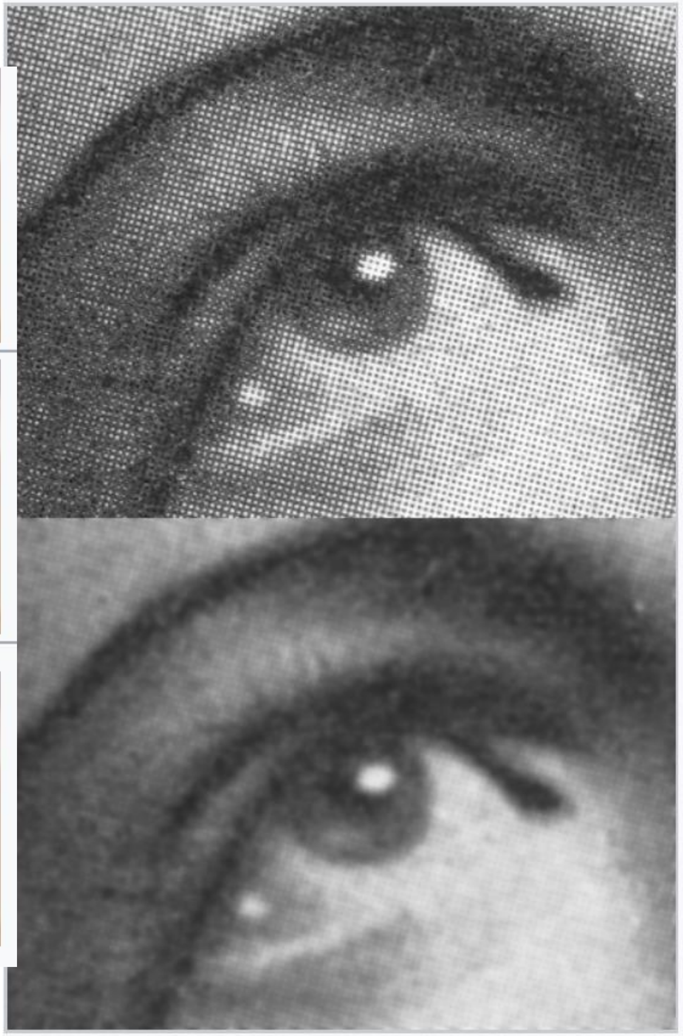
```
import numpy as np
I = np.array([[1,1,1,1,1],[1,1,10,10,10],[1,1,10,10,10],[1,1,10,10,10],[1,1,10,10,10]])
from scipy.ndimage.filters import convolve
K = np.array([[0,1,0],[1,-4,1],[0,1,0]])
H = convolve(I,K)
```

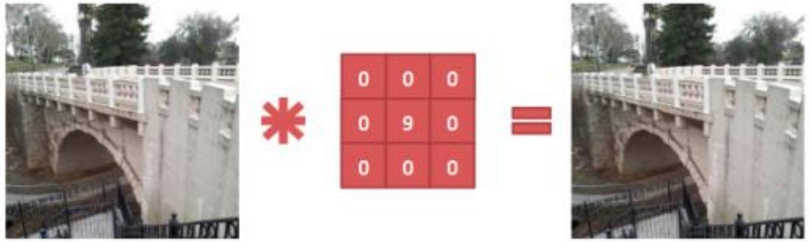
Example Filters

- Reducing noise using a smoothing filter

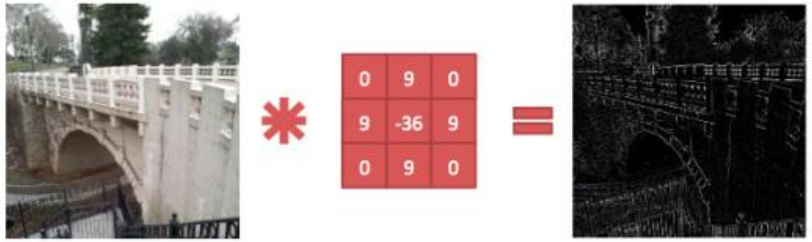


Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\4 & 16 & 24 & 16 & 4\\6 & 24 & 36 & 24 & 6\\4 & 16 & 24 & 16 & 4\\1 & 4 & 6 & 4 & 1\end{bmatrix}$	

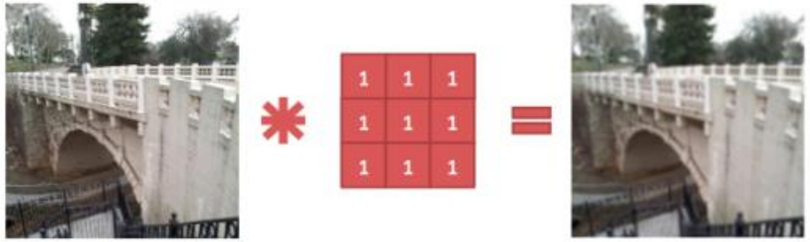




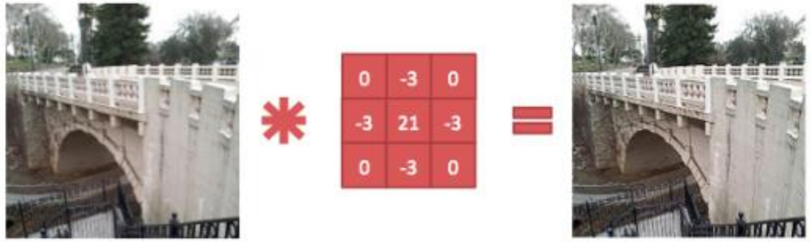
(a) Identity kernel



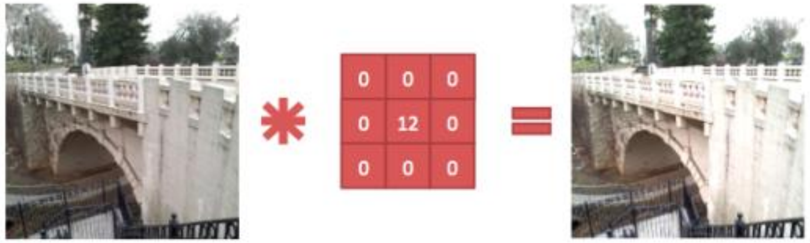
(b) Edge detection kernel



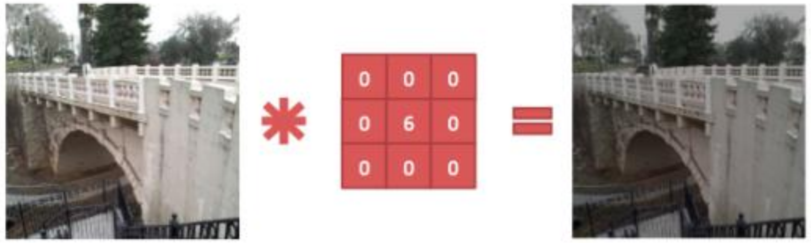
(c) Blur kernel



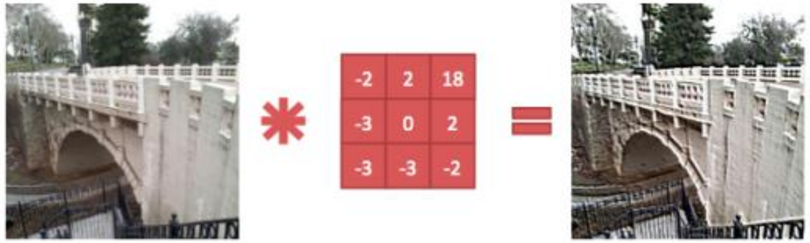
(d) Sharpen kernel



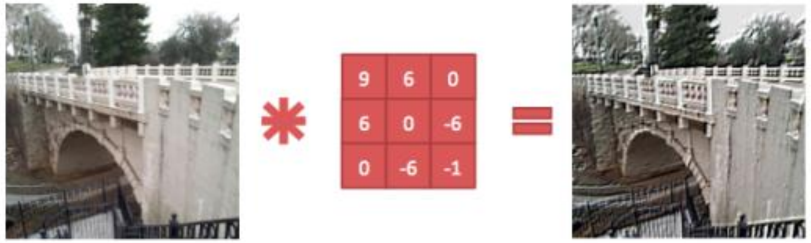
(e) Lighten kernel



(f) Darken kernel

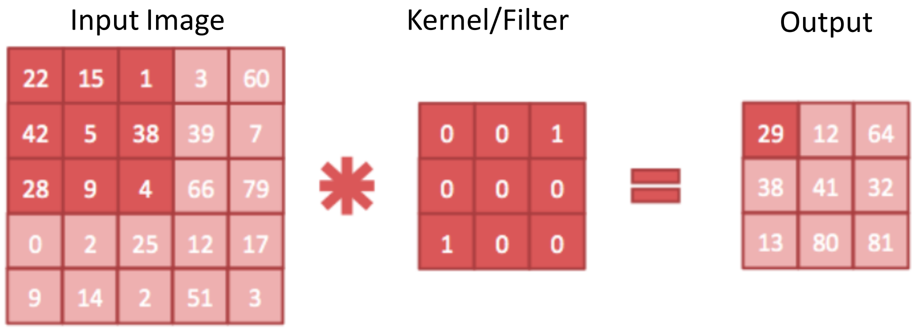


(g) Random kernel 1



(h) Random kernel 2

Convolution*



Input Image

22	15	1	3	60
42	5	38	39	7
28	9	4	66	79
0	2	25	12	17
9	14	2	51	3

$*$

Kernel/Filter

0	0	1
0	0	0
1	0	0

$=$

Output

29	12	64
38	41	32
13	80	81

I K H

$$H(i,j) = \sum_{k=-m/2}^{m/2} \sum_{l=-n/2}^{n/2} I(i+k, j+l)K(k,l)$$

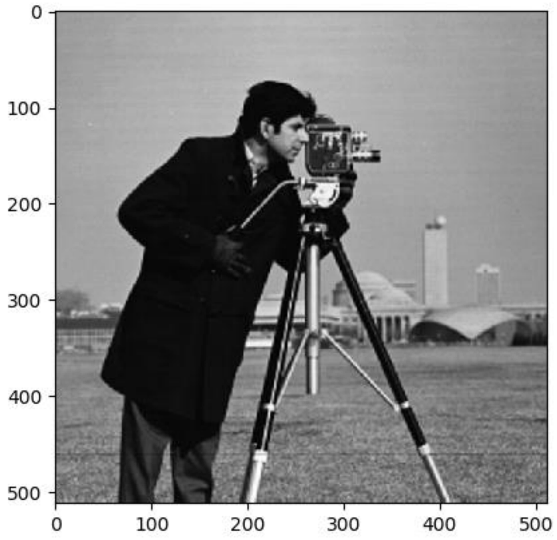
- If you think about it
  - Convolution is a sum of products
    - Can be expressed as a dot product

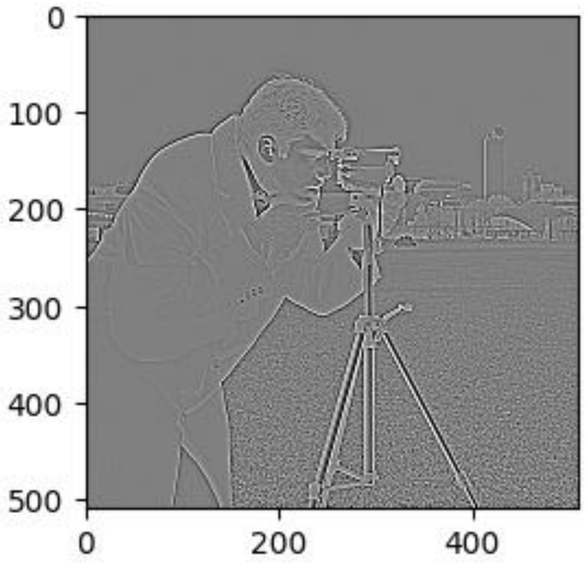
*Strictly speaking, this is cross-correlation.

How to apply filters?

- The easy way
  - Use skimage filters

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.data import camera
from scipy.ndimage import convolve
K = np.array([[0,1,0],[1,-4,1],[0,1,0]])/4 # our filter.
I = camera()/255.0 #so that values are in the range 0-255
H = convolve(I,K)
plt.figure();plt.subplot(1,2,1); plt.imshow(I,cmap='gray')
plt.subplot(1,2,2); plt.imshow(H,vmin=-0.05,vmax=+0.05,cmap = 'gray')
print(f"sizes of images are: {I.shape} and {H.shape}")
```

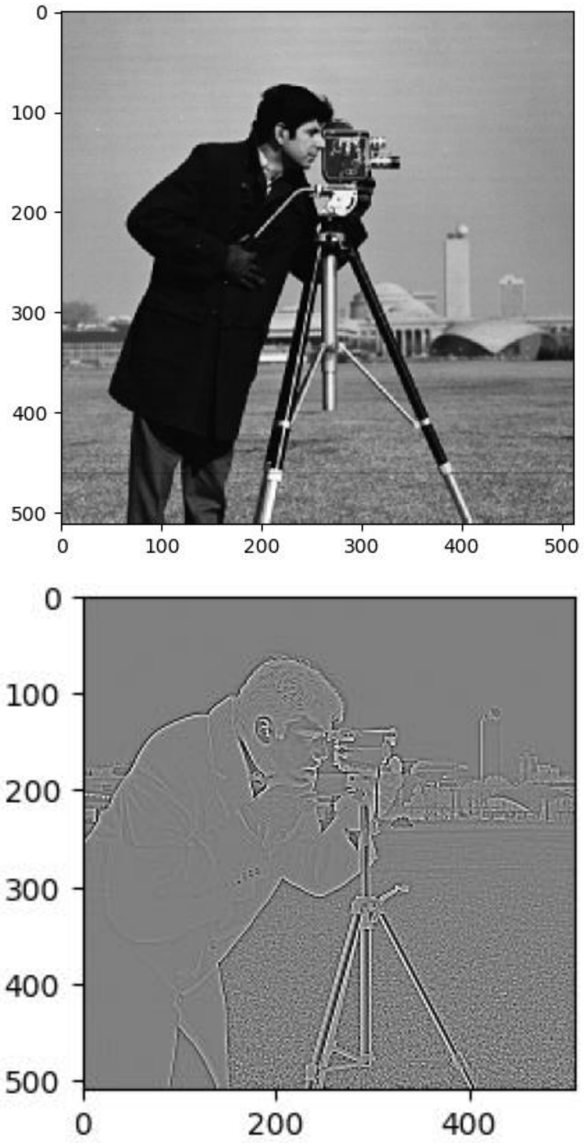




https://github.com/foxtrotmike/CS909/blob/master/learn_filters.ipynb

But what’s the fun in that?

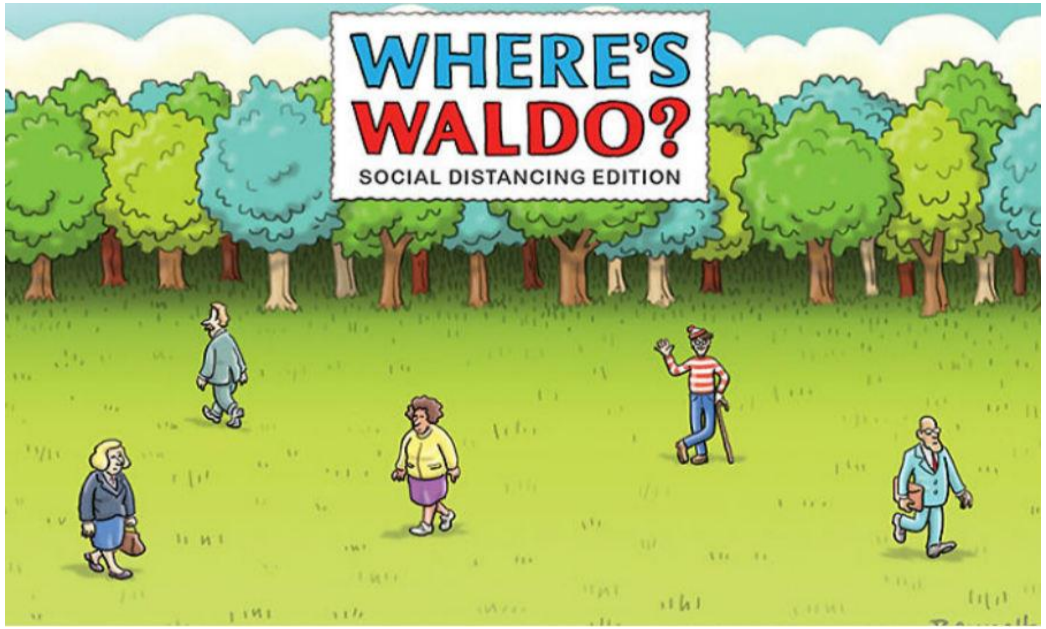
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

class Filter(nn.Module):
    def __init__(self,K):
        super(Filter, self).__init__()
        K = torch.from_numpy(K).float()
        self.K = K.unsqueeze(0).unsqueeze(0) #convert image to NCHW from HW by adding two
extra dimensions in the beginning
    def forward(self, x):
        return F.conv2d(x, self.K) #this is the convolution of the kernel
    def __repr__(self):
      return f"Convolution filter of dimensions: {self.K.shape}"

plt.close('all')
from skimage import data
X = data.camera()/255.0;
plt.subplot(1,2,1); plt.imshow(X,cmap='gray')
K = np.array([[0 ,1, 0],[1,-4,1], [0, 1 ,0]])/4.0
X_torch = torch.from_numpy(X).float().unsqueeze(0).unsqueeze(0) #convert image to NCHW from
HW by adding two extra dimensions in the beginning
#move image to torch
f = Filter(K)
#set the kernel in Filter object
Z_torch = f(X_torch)
#convolution
Z = Z_torch.squeeze().detach().numpy()
#move back to numpy
plt.subplot(1,2,2); plt.imshow(Z,vmin=-0.05,vmax=+0.05,cmap = 'gray')
print(f)
print(f"sizes of images are: {X.shape} and {Z.shape}")
```



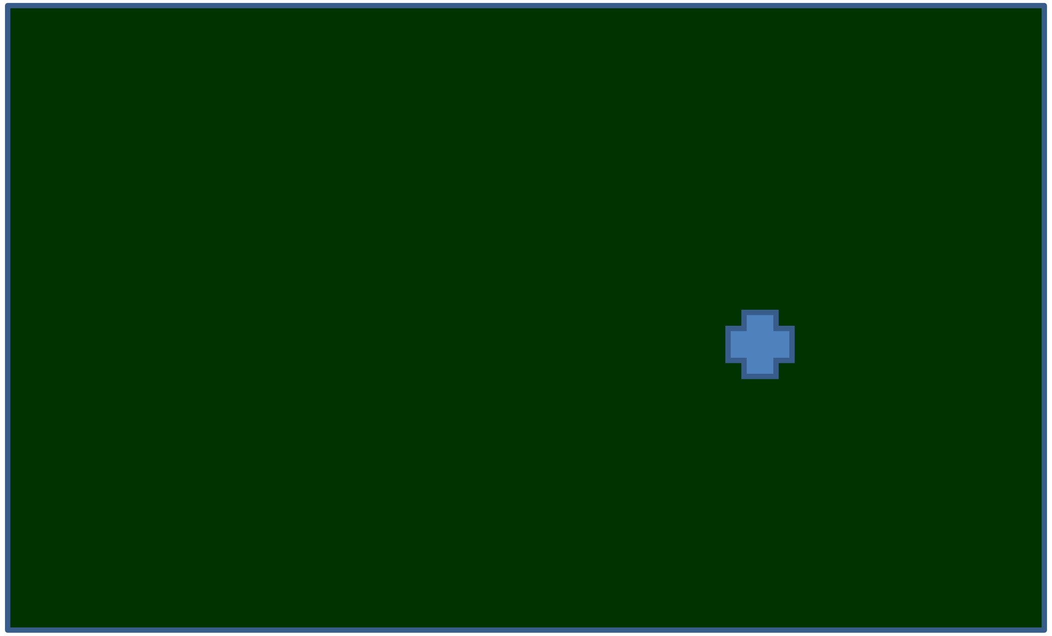
https://github.com/foxtrotmike/CS909/blob/master/learn_filters.ipynb
https://github.com/foxtrotmike/CS909/blob/master/pytorch_conv.py

Now the interesting question

- Can we learn filters to do something we want to do?
  - Let’s say we have an image and it’s output after a certain operation
  - Can we learn a filter that produces the output given the input?

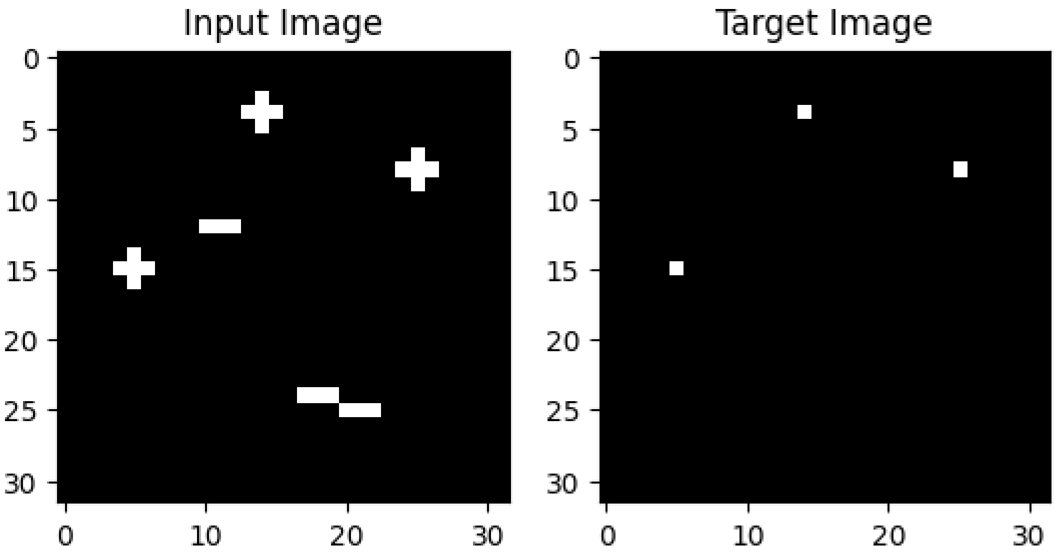


Filter?



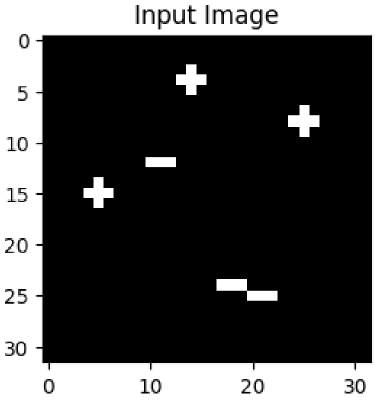
Example

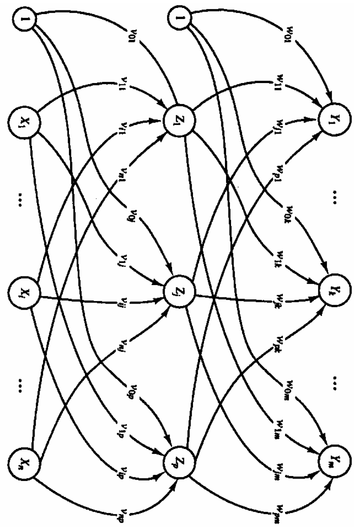
- Let’s say, we have an image and we want to design a filter that when convolved with the image leads to the desired output. How?

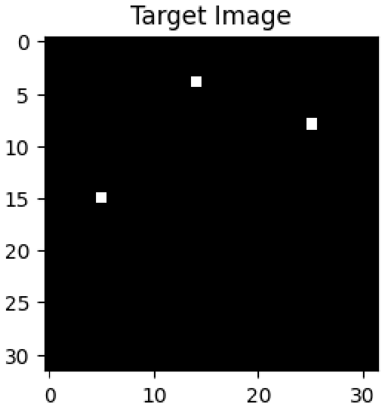


How can this be done?

- Let's try to build a multi-layer perceptron
  - Input image size: (32,32)
    - This means the number of input neurons will be 1024
  - Target image size: (32,32)
    - This means the number of output neurons will be 1024
  - Number of weights:
    - 1024*1024 = 1,048,576
  - Add hidden layers!
  - Good luck!







Let’s try to learn a 3x3 filter

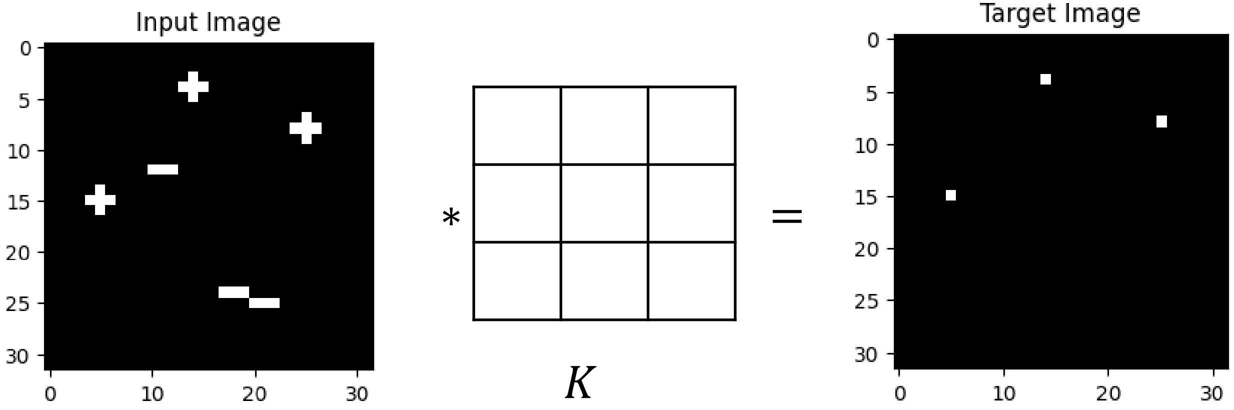
- Representation

$$H = I \star K \qquad H(i,j) = \sum_{k=-1}^{m=1} \sum_{l=-1}^{n=1} I(i+k, j+l) K(k,l)$$

- Evaluation

$$E(K) = \sum_{k=1}^{M} \sum_{l=1}^{N} \big(H(i,j) - T(i,j)\big)^2$$

- Optimization
  - Solve the following problem: $\min_{K} E(K)$

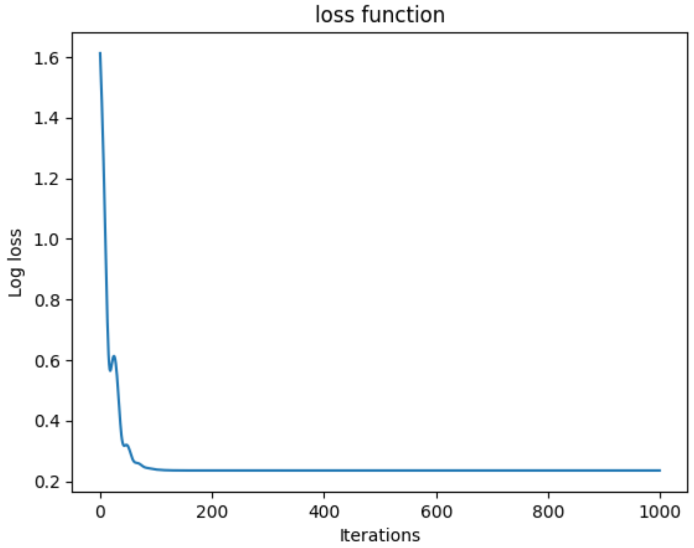


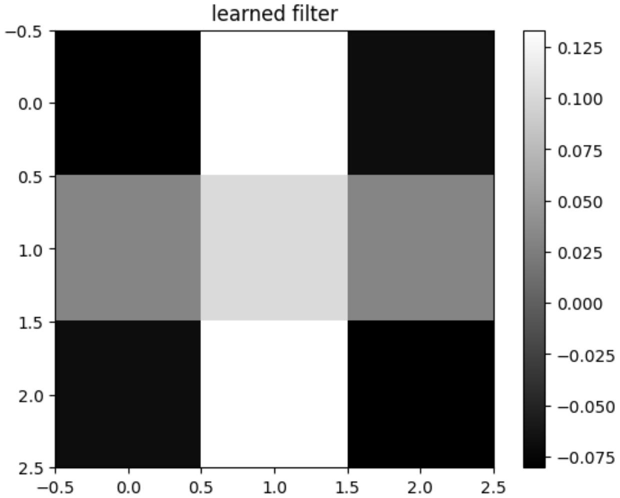
Let’s solve this

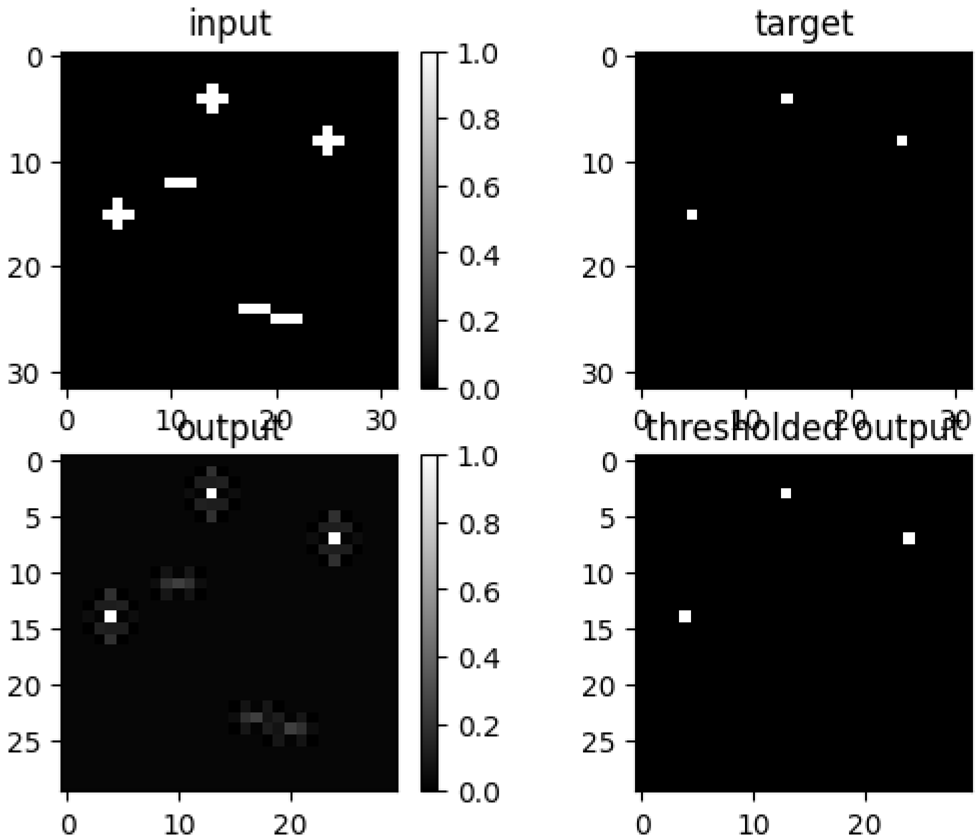
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
class Filter(nn.Module):
    def __init__(self, ksize = 3):
        super(Filter, self).__init__()
        self.conv1 = nn.Conv2d(1,1, ksize) #torch allows creating a convolution filter using a conv2d layer object which applies
conv2d internally for a given input
    def forward(self, x):
        x = self.conv1(x) #perform convolution
        x = torch.tanh(x) #apply activation
        return x
# let's use a convolution filter of size ksize
ksize = 3
bsize = int(ksize/2) #size of broder region
f = Filter(ksize)
optimizer = torch.optim.Adam(f.parameters(), lr=1e-2)
T_torch = torch.from_numpy(T[bsize:-bsize,bsize:-bsize]).float()# reduce target filter size to compensate for border loss in
convolution
X_torch = torch.from_numpy(X).float().unsqueeze(0).unsqueeze(0) #convert image to NCHW from HW by adding two extra dimensions in
the beginning
L = []
for _ in range(1000):
    optimizer.zero_grad() #optimization
    Z_torch = f(X_torch).squeeze()
    loss = torch.sum(torch.abs((T_torch-Z_torch)**2)) #error
    loss.backward()
    optimizer.step()
    L.append(loss.item())

output = Z_torch.squeeze().detach().numpy()
output = (output-np.min(output))/(np.max(output)-np.min(output)) #rescale so that the lowest value in the input image is 0 and the
highest is 1 so we can threshold it
```

https://github.com/foxtrotmike/CS909/blob/master/learn_filters.ipynb

Results



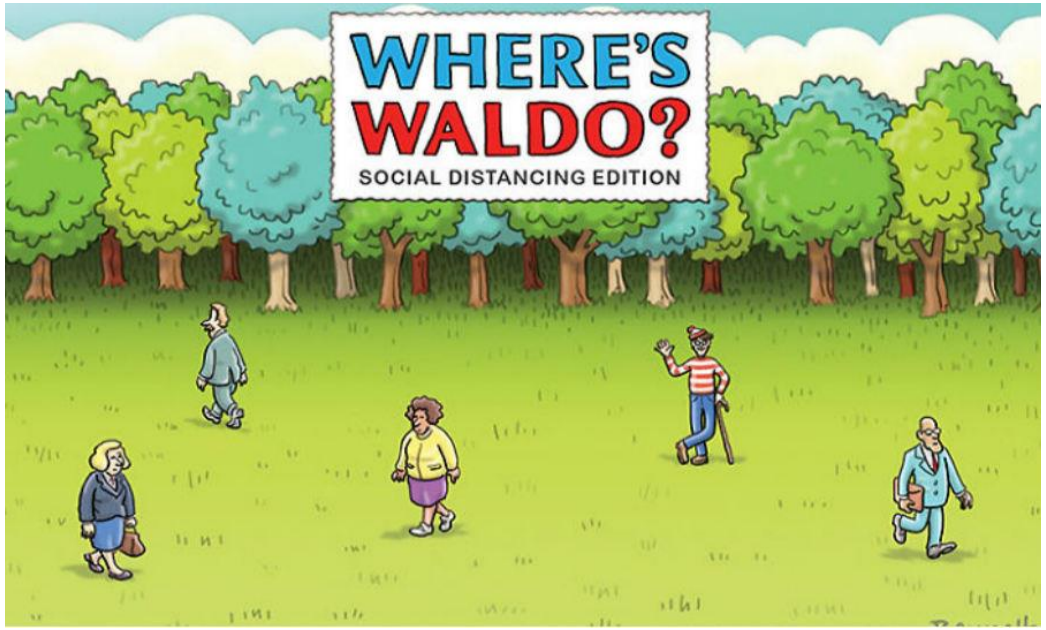




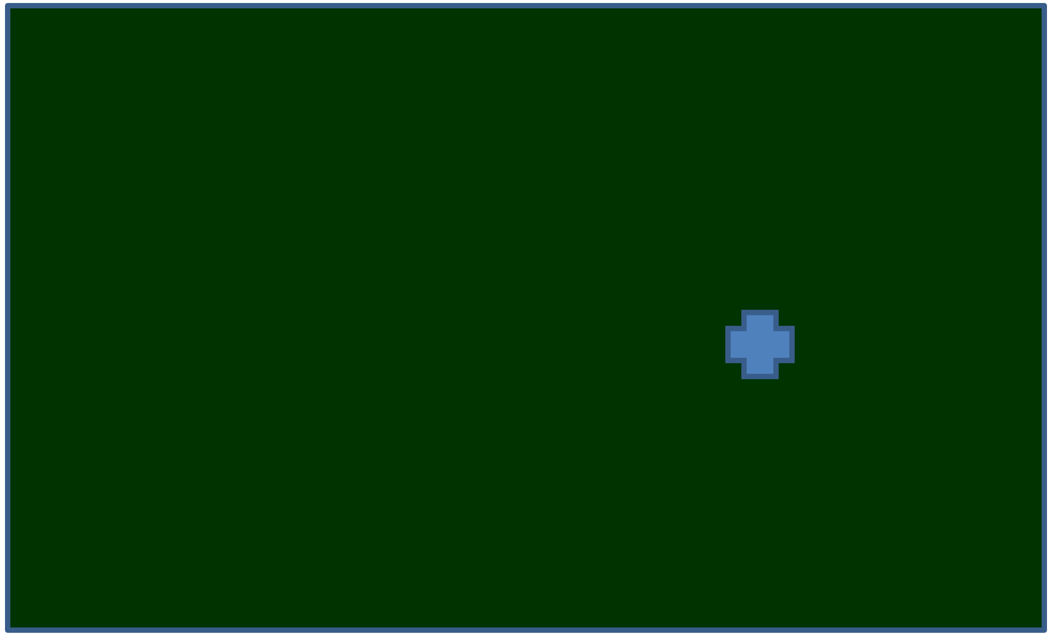
https://github.com/foxtrotmike/CS909/blob/master/learn_filters.ipynb
https://github.com/foxtrotmike/CS909/blob/master/learn_filters.py

Another way of looking at this

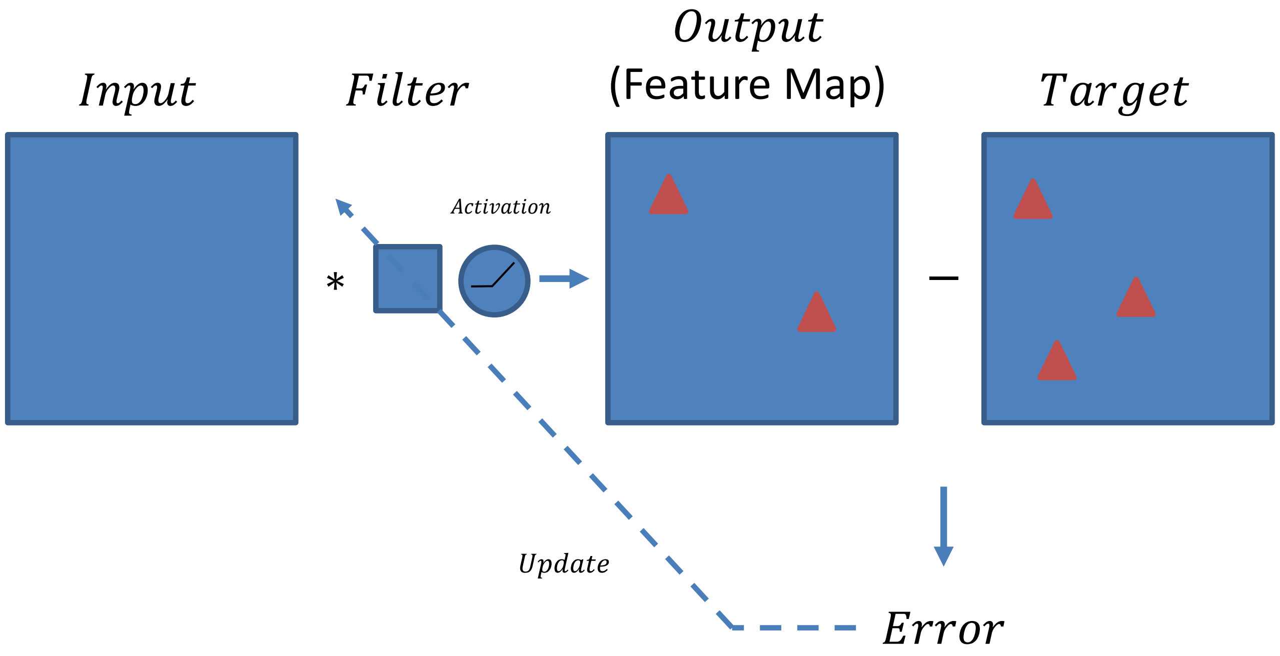
- We learned a convolution filter kernel based on an input and a target image
- The filter will act as a + detector when convolved with a new image (hopefully!)



Filter?

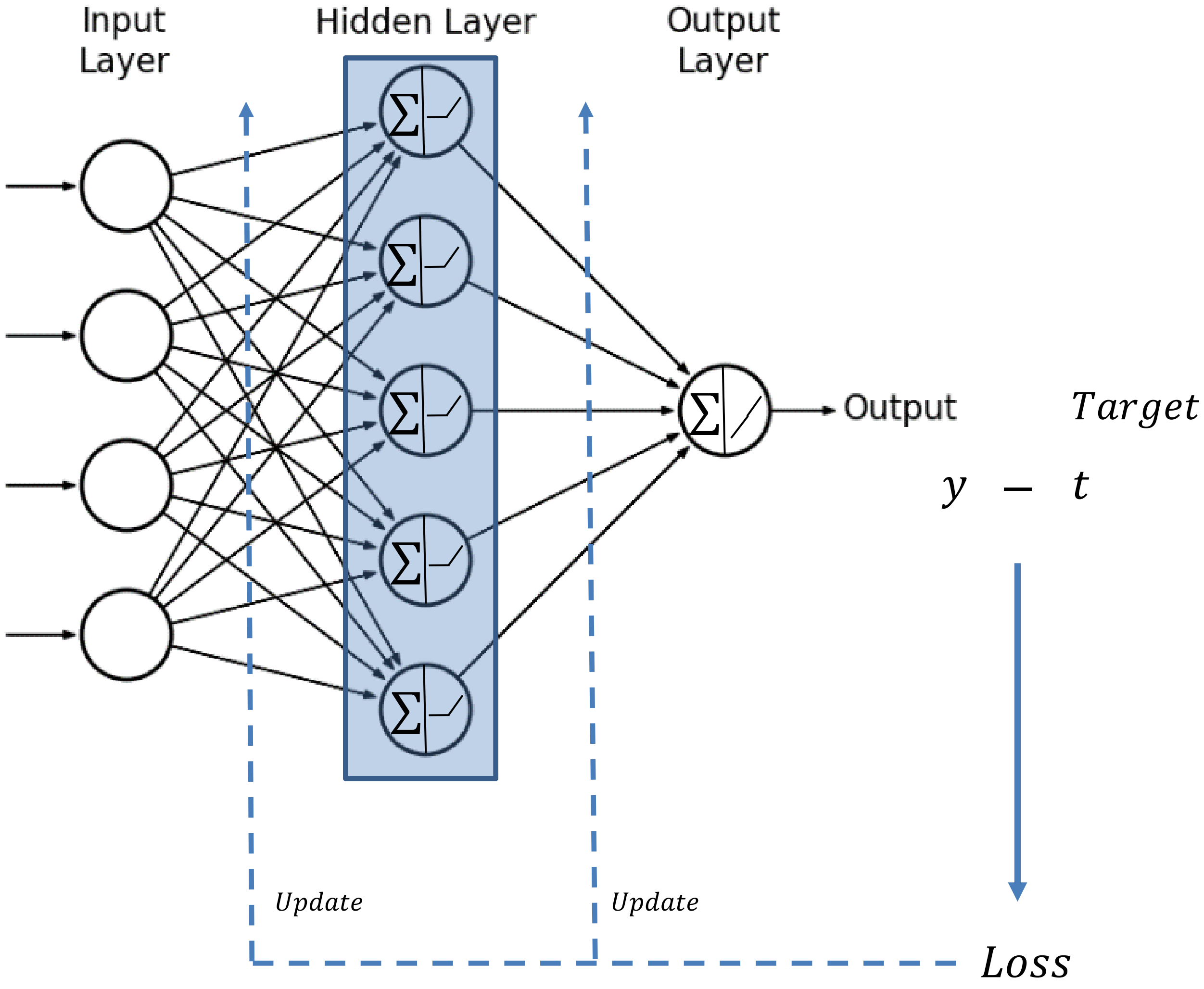


Most basic convolutional neural network

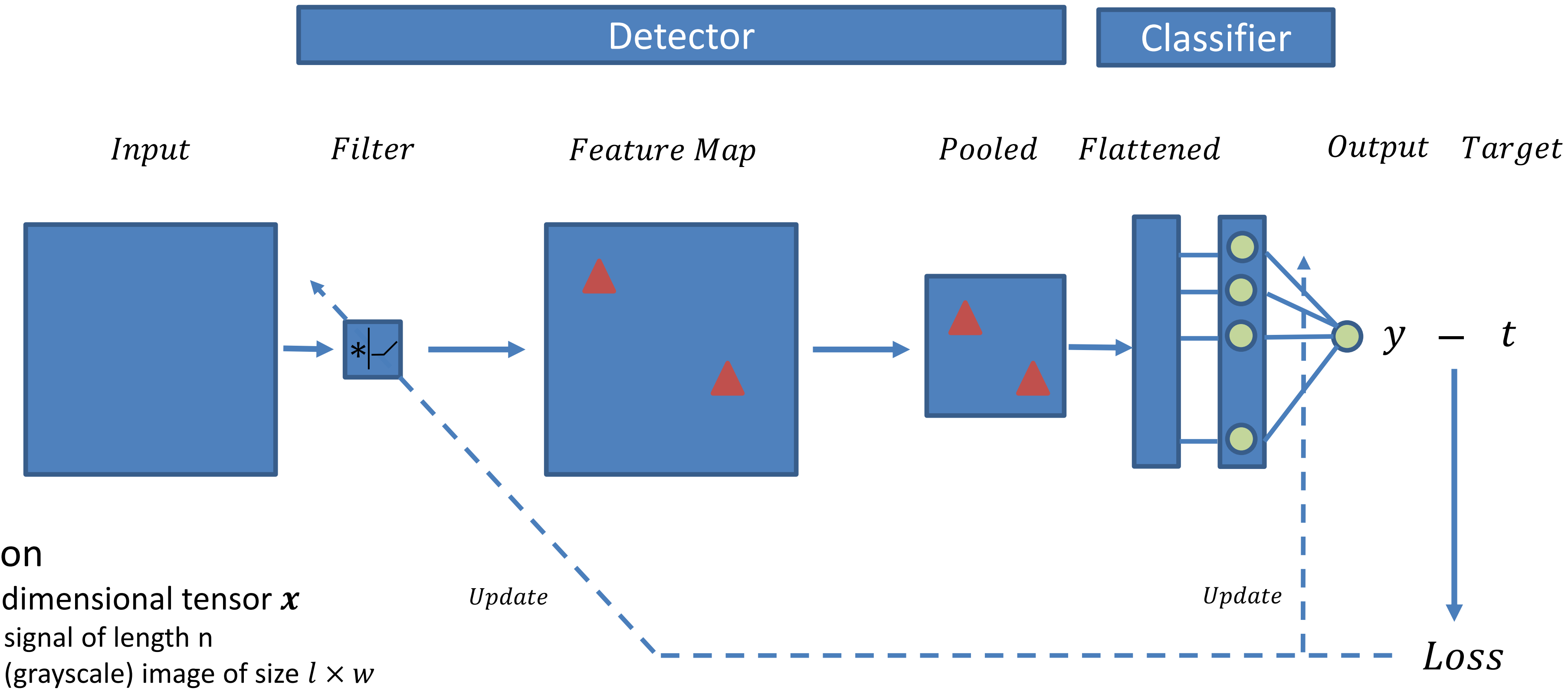


- Acts as a “detection” or “feature extraction” unit

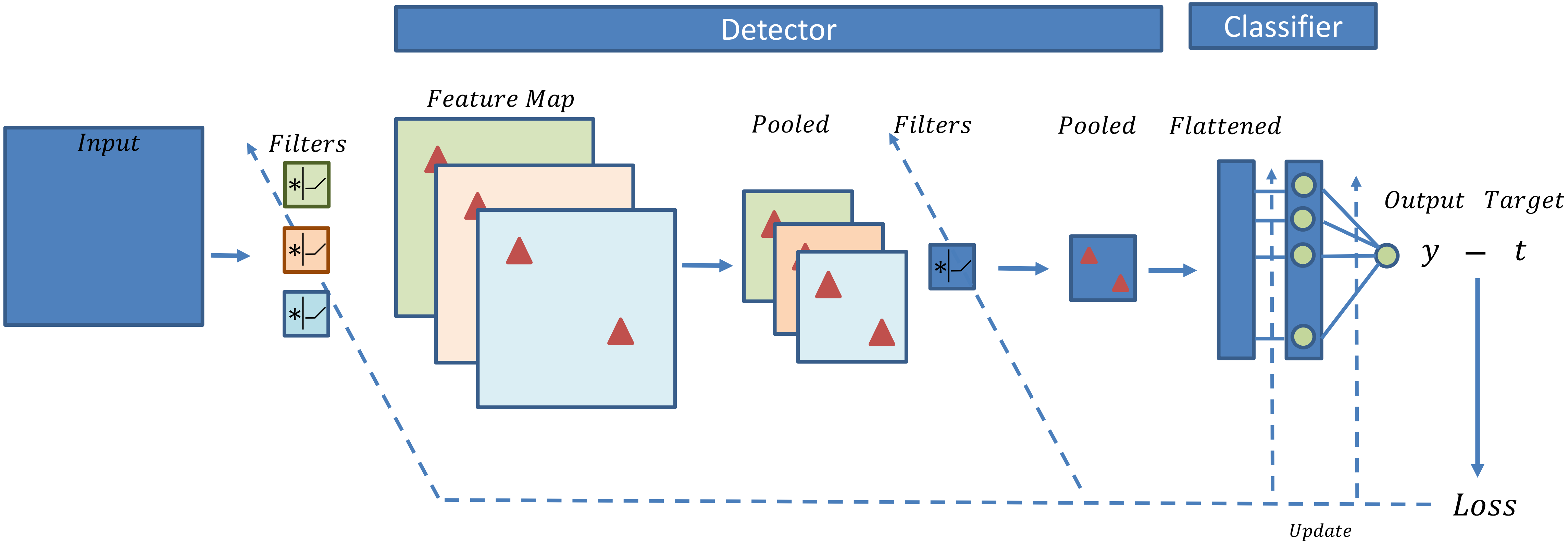
Classification with Multilayer Perceptron



REO for a convolution neural network

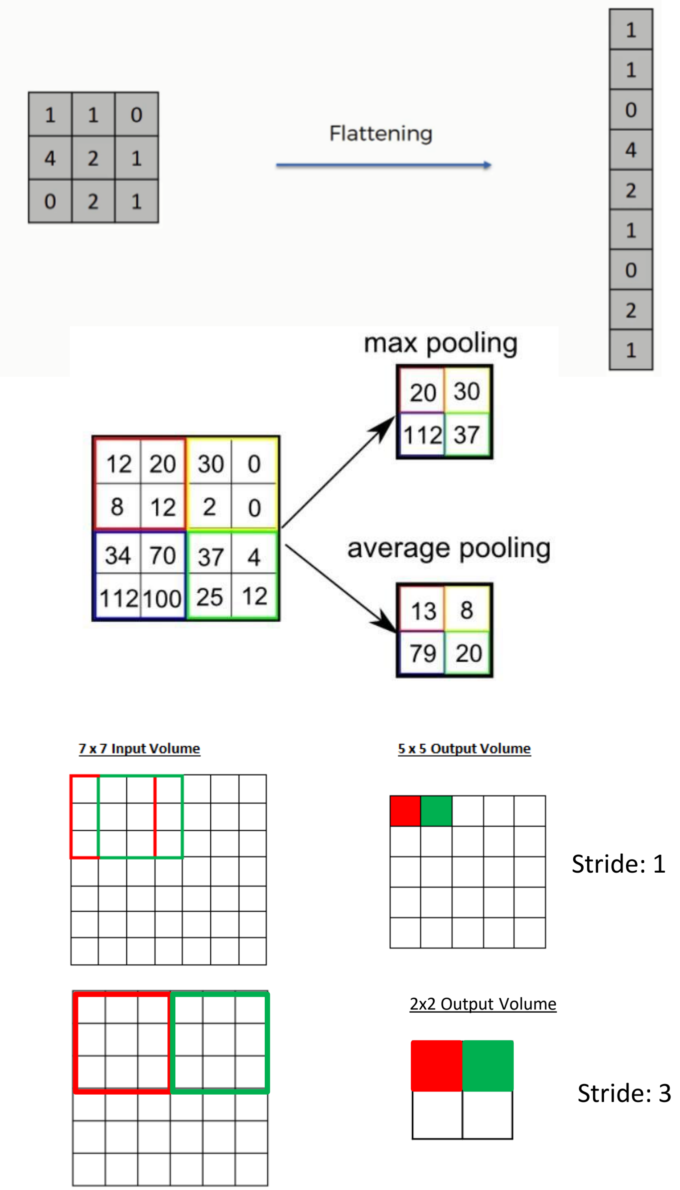


- Representation
    - Input: a k-dimensional tensor $\boldsymbol{x}$
        - k = 1: signal of length n
        - k = 2: (grayscale) image of size $l \times w$
            - RGB channel image: $l \times w \times 3$
        - k = 3: $l \times w \times t$ video of frame size $l \times w$ with duration t
    - Output: A decision score $\mathrm{y} = f(x; \boldsymbol{\theta})$ (can be multi-dimensional as well)
    - Structure
        - **Layers of Learnable filters each of which is correlated (or convolved) with the input tensor in parallel followed by convolution with other filters**
            - A single convolution is indicated by $\boldsymbol{z} = \boldsymbol{a}(\boldsymbol{x} \star \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the representation of a single filter and $\boldsymbol{a}(\cdot)$ is an activation function. Filters are much smaller than $\boldsymbol{x}$.
            - Implemented as layers: Conv1d, Conv2d, Conv3d (in PyTorch)
        - The correlation output is then pooled (optional)
        - Nonlinear activation functions are applied
        - Aggregated to produce the final output (depending upon application)


Detector
Classifier
Input
Filters
Feature Map
Pooled
Filters
Pooled
Flattened
Output
Target
$y - t$
Loss
Update

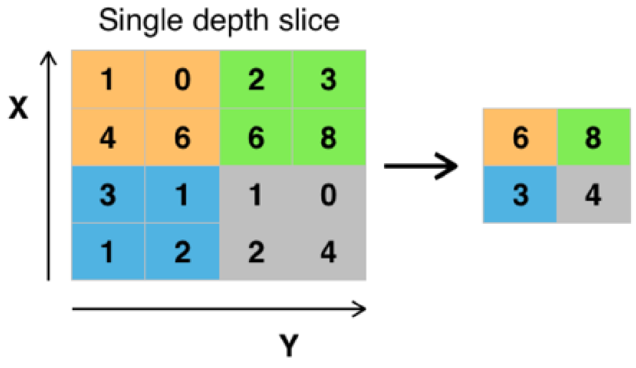
Convolutional Neural Networks for ML

- If we want to use the output of convolution filters for learning to classify or regress or rank or for any other task
  - We can use a multilayer perceptron but:
    - We will need to “flatten” the output of the correlation filter (aka feature/filter map)
      - Convert an image to a vector e.g., (8x8 to 64)
    - We will also need to reduce the dimensions of the output
      - Done through “Pooling”
        - » Average or max
      - And/Or “Striding”
        - » How we move the convolution filter



Structure

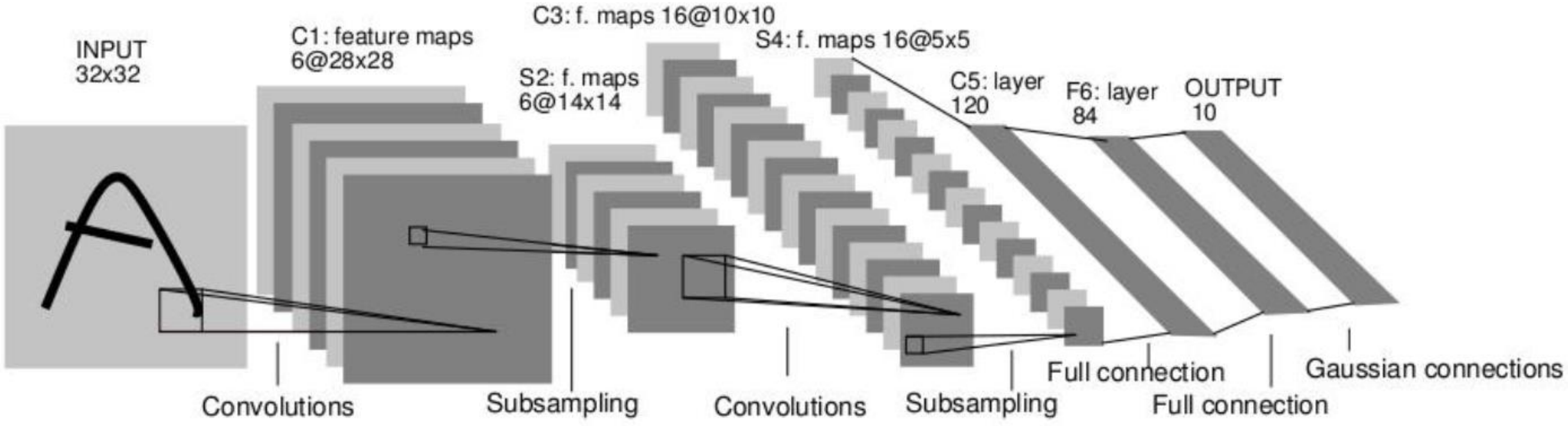
- Increasing "globality"
  - Input → Convolution → Non-linearity → Sub-sampling … → Fully Connected Layer (for classification)



1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

→

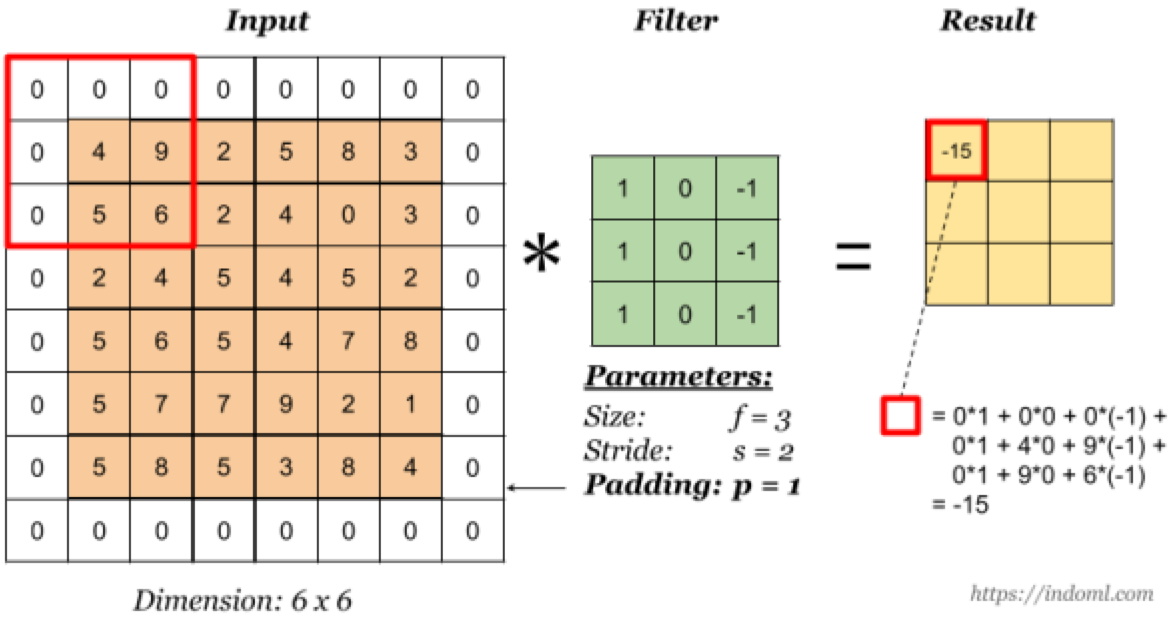
6	8
3	4



See Coding

- https://github.com/foxtrotmike/CS909/blob/master/learn_filters.ipynb
- https://github.com/foxtrotmike/CS909/blob/master/cnn_mnist_pytorch.ipynb

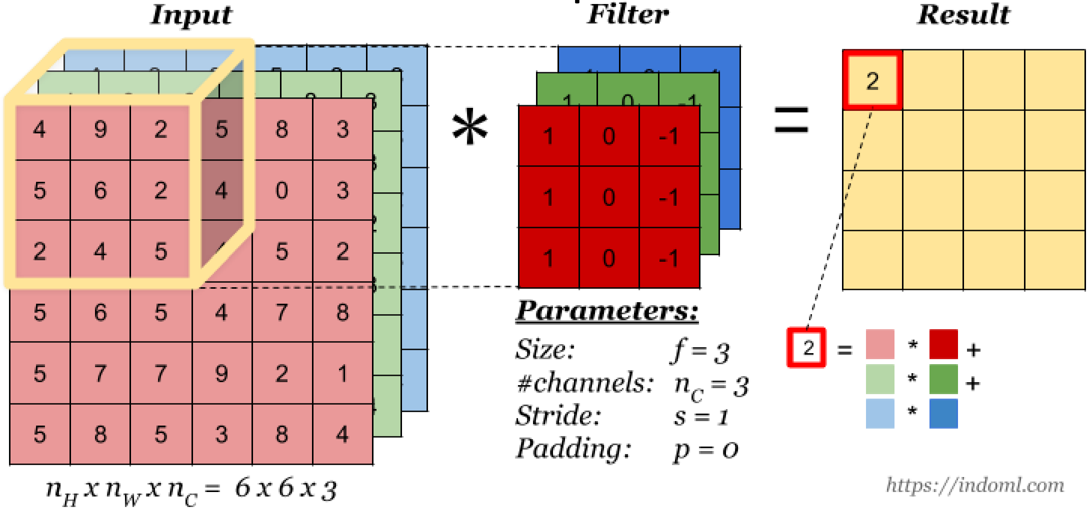
Padding



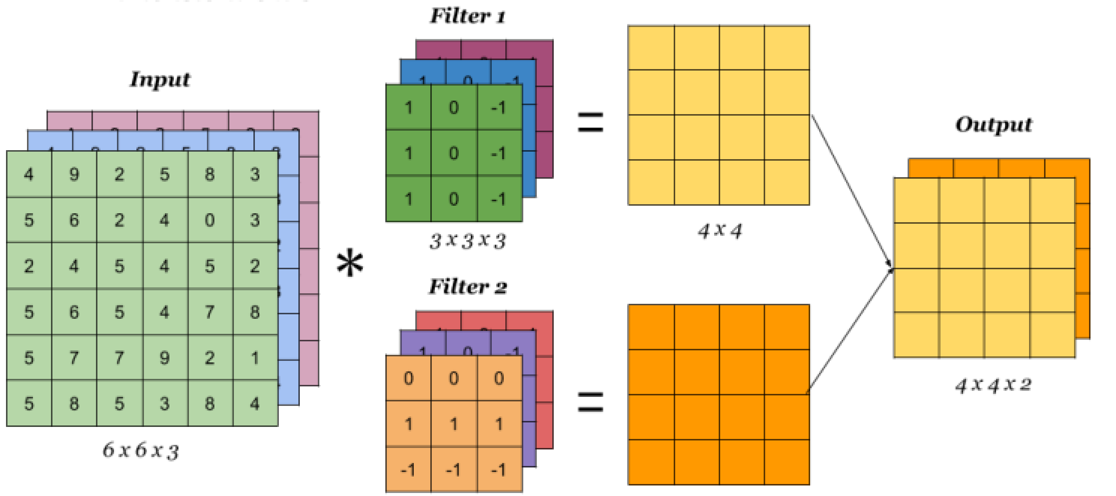
Parameters:
Size: $f = 3$
Stride: $s = 2$
Padding: $p = 1$

= 0*1 + 0*0 + 0*(-1) +
0*1 + 4*0 + 9*(-1) +
0*1 + 9*0 + 6*(-1)
= -15

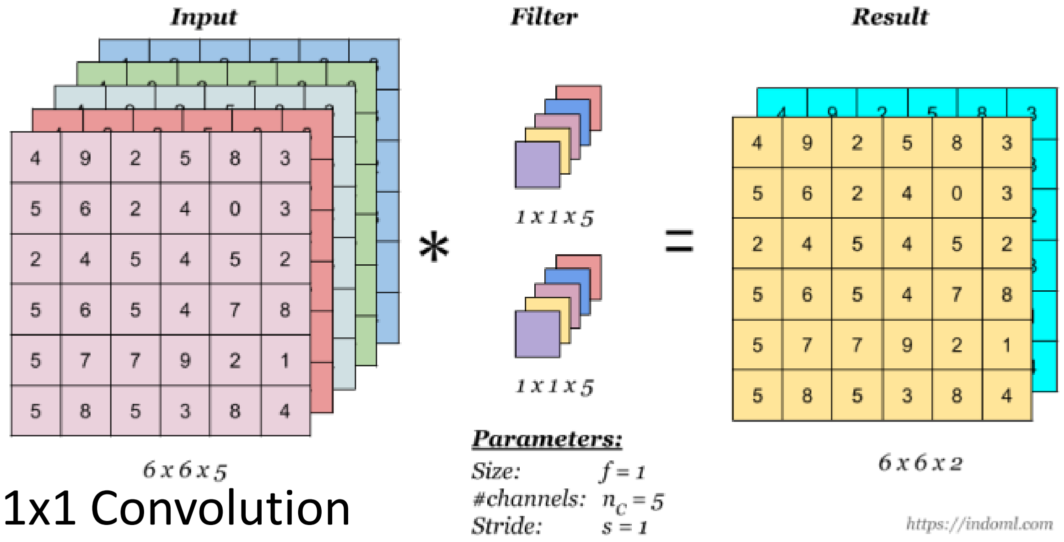
Multiple Channels



Parameters:
Size: $f = 3$
#channels: $n_C = 3$
Stride: $s = 1$
Padding: $p = 0$



Multiple IO Channels



Parameters:
Size: $f = 1$
#channels: $n_C = 5$
Stride: $s = 1$

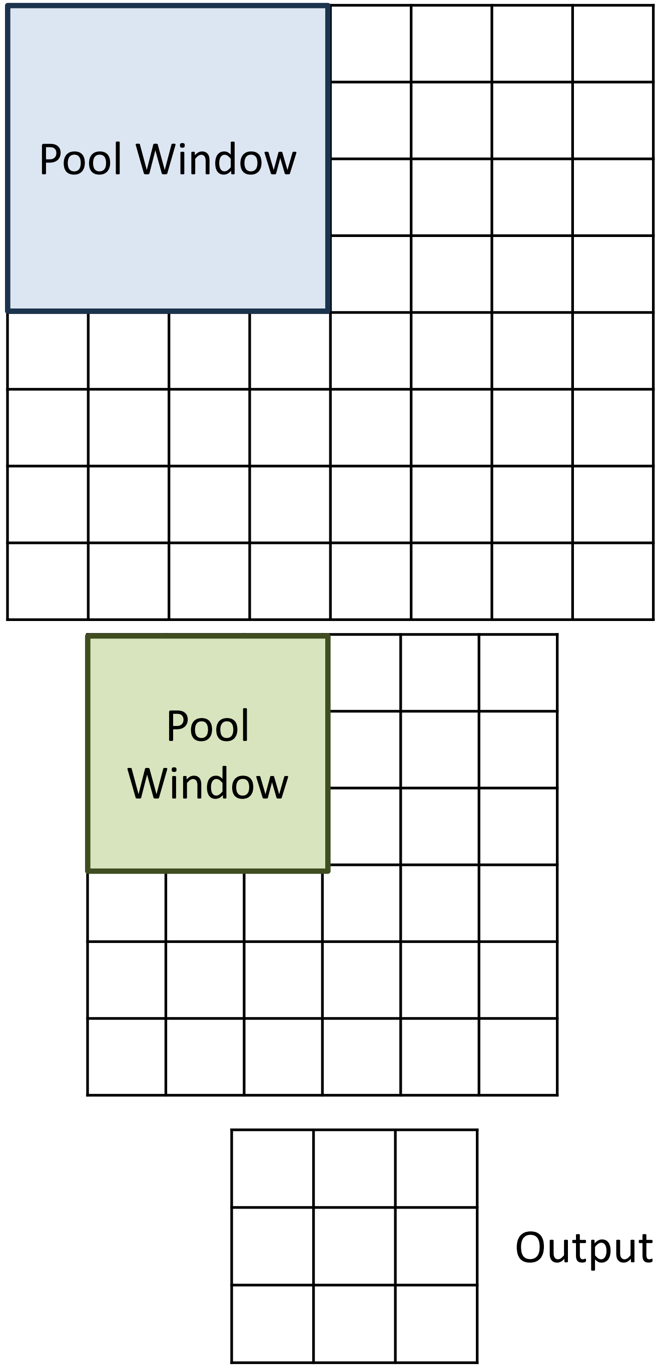
1x1 Convolution

https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/

Pooling

- Average
- Max
- Adaptive Pooling
  - Produces a fixed (specified) sized output despite the size of the input by changing the window size adaptively
  - Allows us to have convolutional neural networks take arbitrary image sizes as input
    - nn.AdaptiveMaxPool2d
    - nn.AdaptiveAvgPool2d
- Learnable pooling

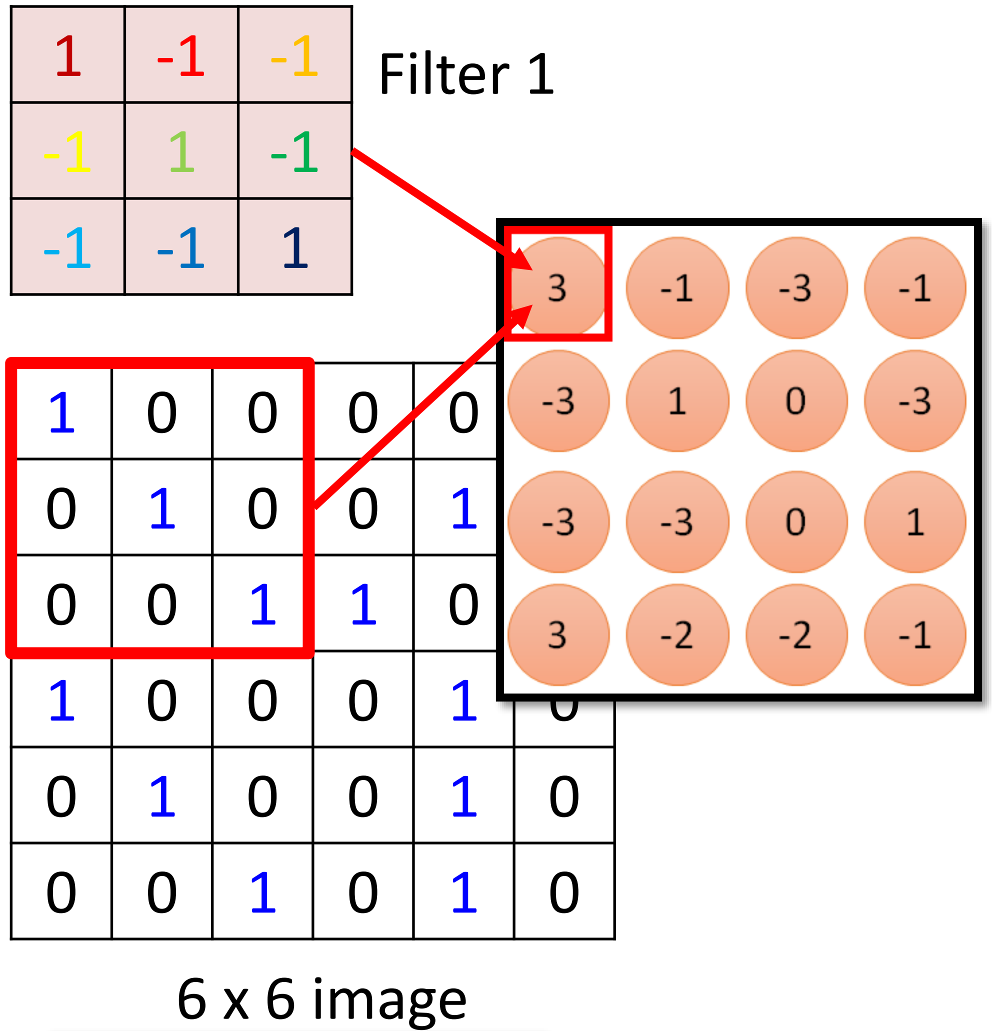
```
pool = nn.AdaptiveAvgPool2d(3)
input = torch.randn(1, 64, 8, 8)
output = pool(input)
print(output.shape)#3,3
input = torch.randn(1, 64, 6, 6)
output = pool(input)
print(output.shape) #3,3
```



“Learning Pooling for Convolutional Neural Network.” *Neurocomputing* 224 (February 8, 2017): 96–104. https://doi.org/10.1016/j.neucom.2016.10.049.

Why do CNNs work?

- There are three major reasons why CNN’s work better than fully connected MLPs
  - Local weight connectivity
    - In contrast to a fully connected neural network like a multilayer perceptron, a filter in a CNN operates over an image at the local level
  - Shared weights
    - No separate weights for each pixel
  - Hierarchical representations



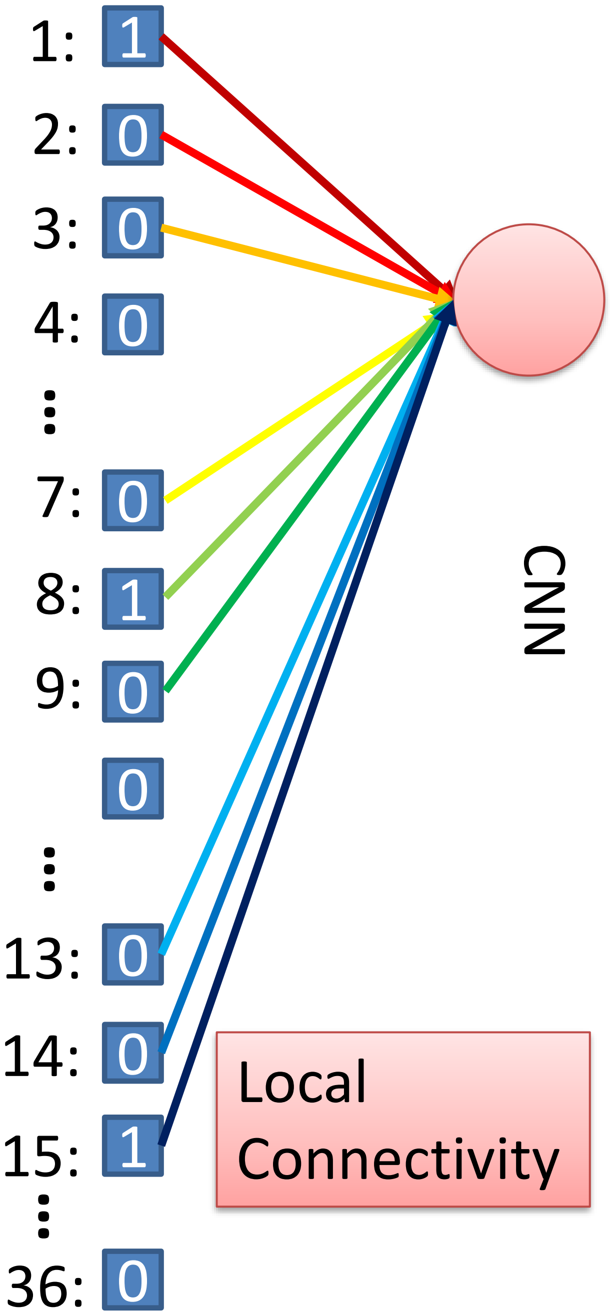
1	-1	-1
-1	1	-1
-1	-1	1

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

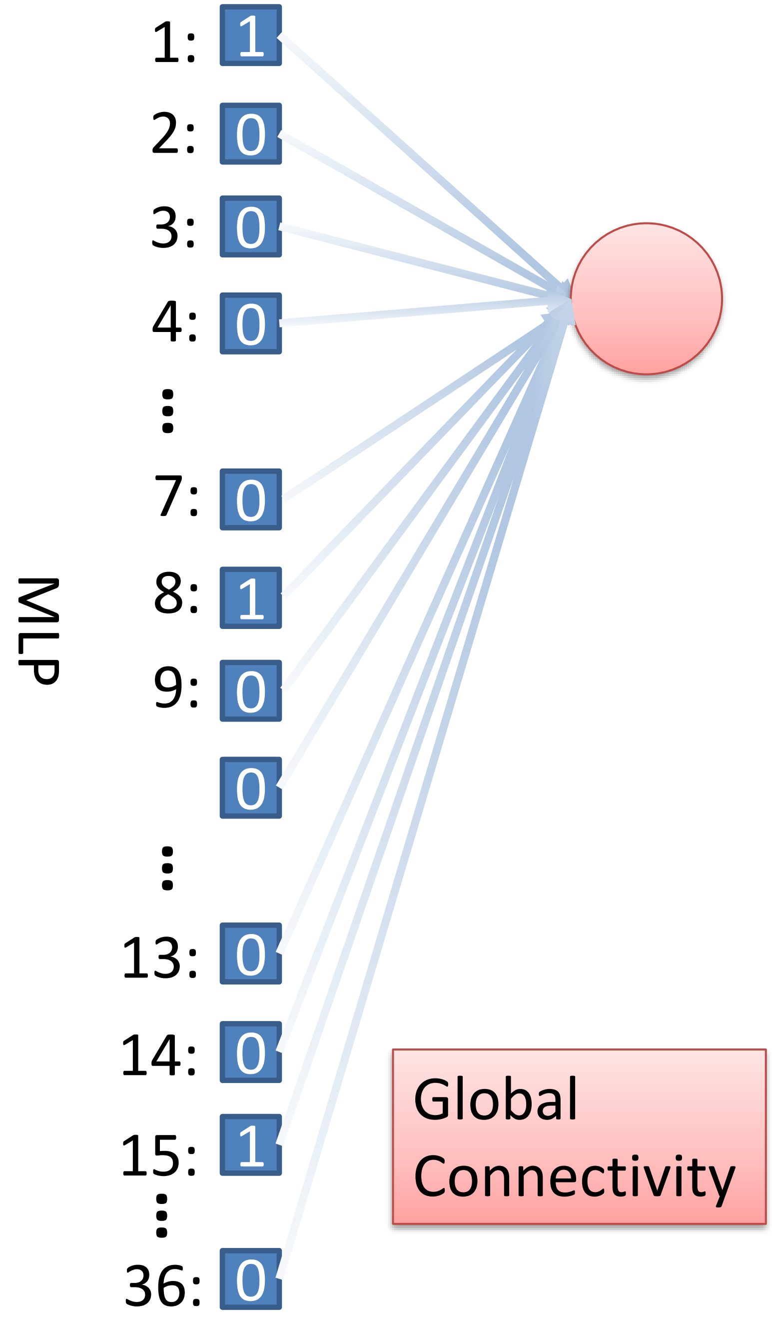
6 x 6 image

Only connect to 9 input, not fully connected

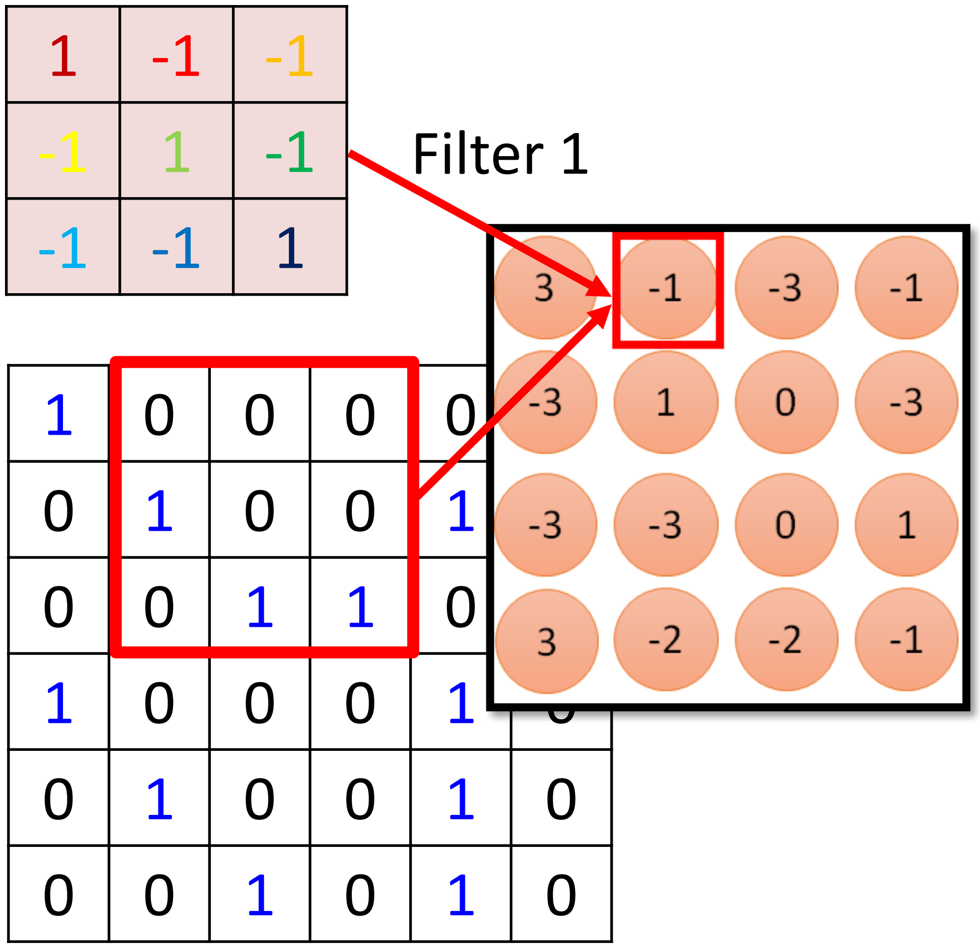
Less parameters!



Local Connectivity



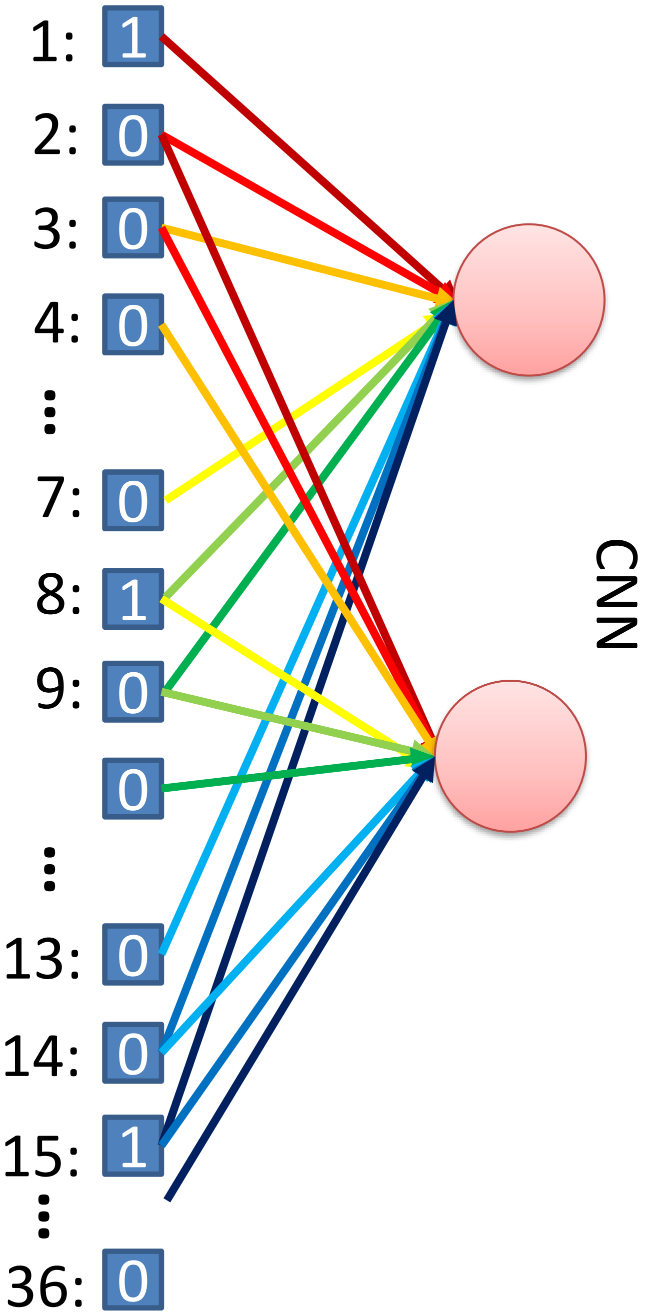
Global Connectivity

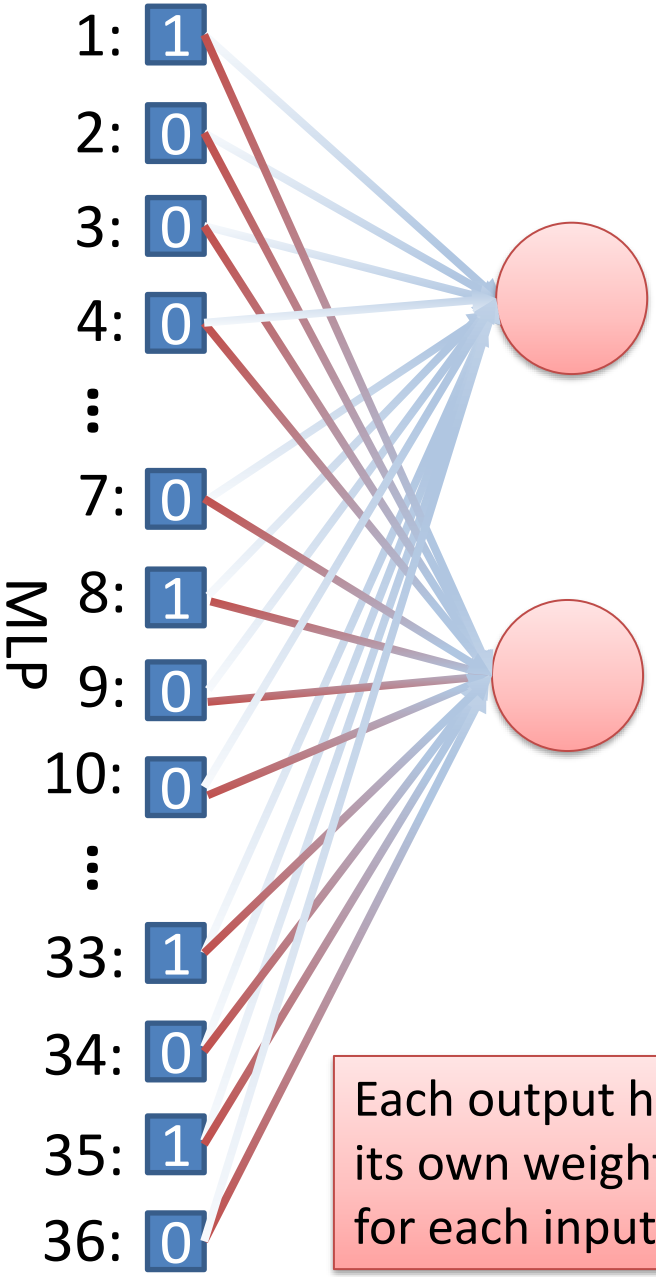


6 x 6 image

Shared weights

Even less parameters!

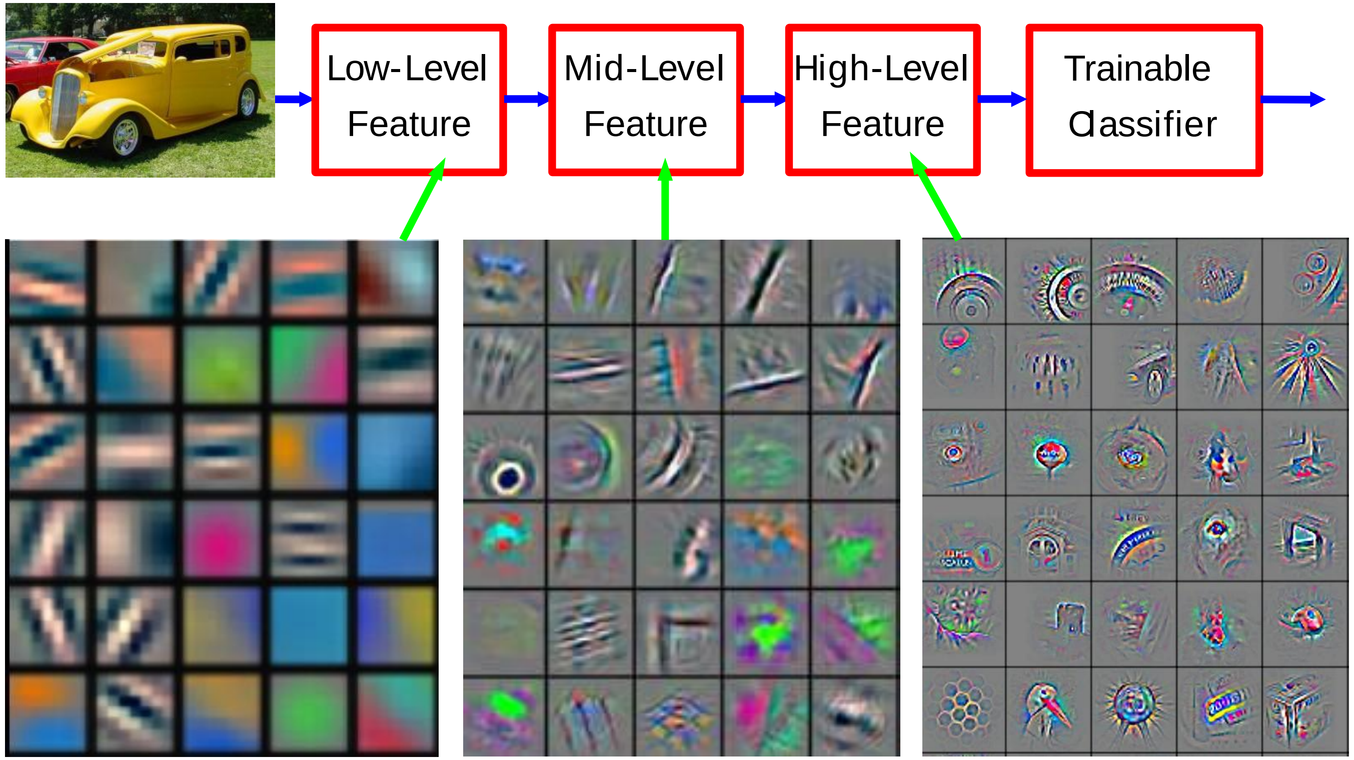




Each output has its own weights for each input

Deep Learning: Learning Hierarchical Representations

It's deep if it has more than one stage of non-linear feature transformation



Low-Level Feature

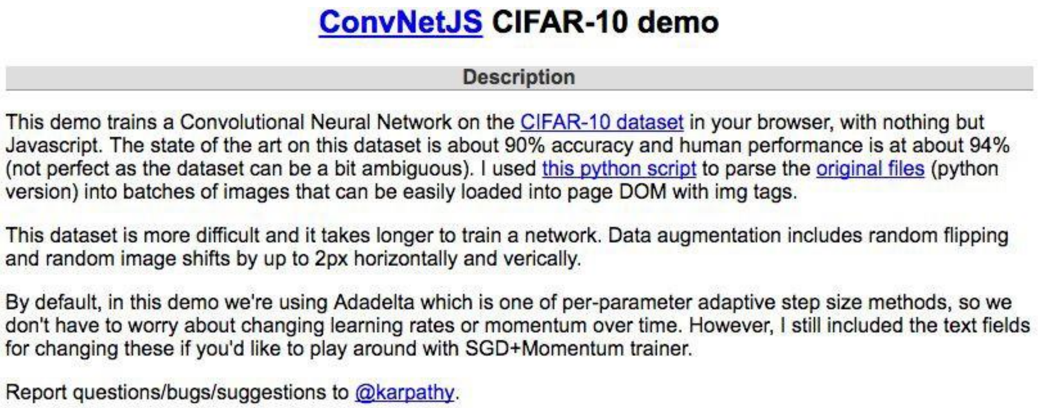
Mid-Level Feature

High-Level Feature

Trainable Classifier

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

[ConvNetJS demo: training on CIFAR-10]



ConvNetJS CIFAR-10 demo

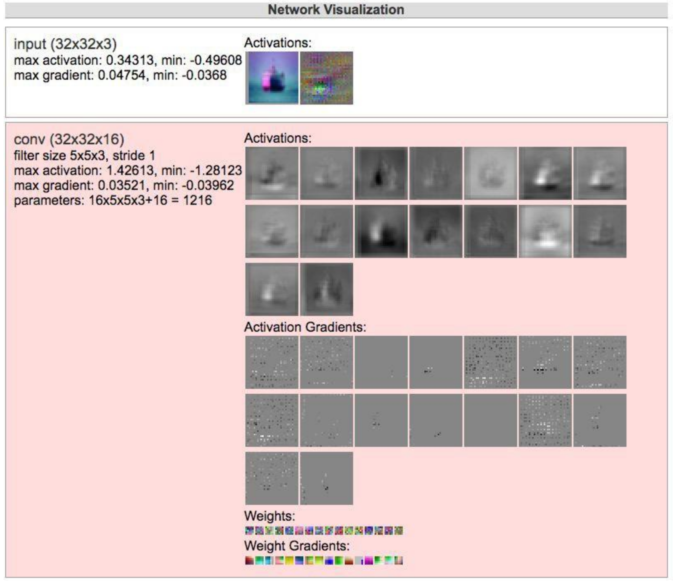
Description

This demo trains a Convolutional Neural Network on the CIFAR-10 dataset in your browser, with nothing but Javascript. The state of the art on this dataset is about 90% accuracy and human performance is at about 94% (not perfect as the dataset can be a bit ambiguous). I used this python script to parse the original files (python version) into batches of images that can be easily loaded into page DOM with img tags.

This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and vertically.

By default, in this demo we're using Adadelta which is one of per-parameter adaptive step size methods, so we don't have to worry about changing learning rates or momentum over time. However, I still included the text fields for changing these if you'd like to play around with SGD+Momentum trainer.

Report questions/bugs/suggestions to @karpathy.



Network Visualization

input (32x32x3)
max activation: 0.34313, min: -0.49608
max gradient: 0.04754, min: -0.0368

conv (32x32x16)
filter size 5x5x3, stride 1
max activation: 1.42613, min: -1.28123
max gradient: 0.03521, min: -0.03962
parameters: 16x5x5x3+16 = 1216

http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

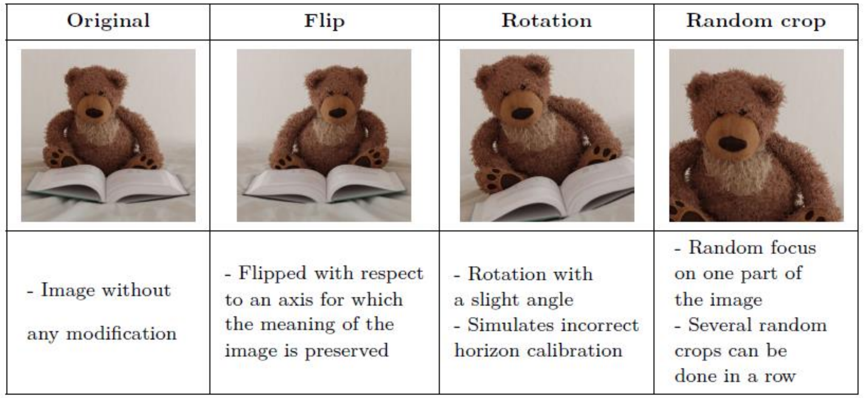
RISK MINIMIZATION AND GENERALIZATION

Risk Minimization in Neural Networks

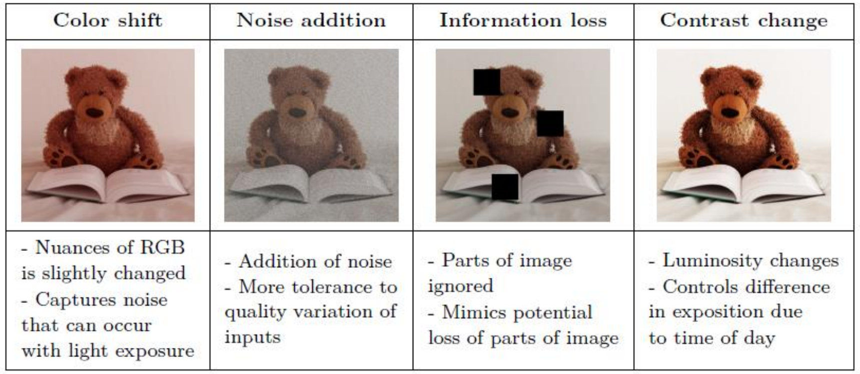
- Structural Risk
  - Empirical Error Minimization via Loss minimization
  - Regularization

Important Concepts

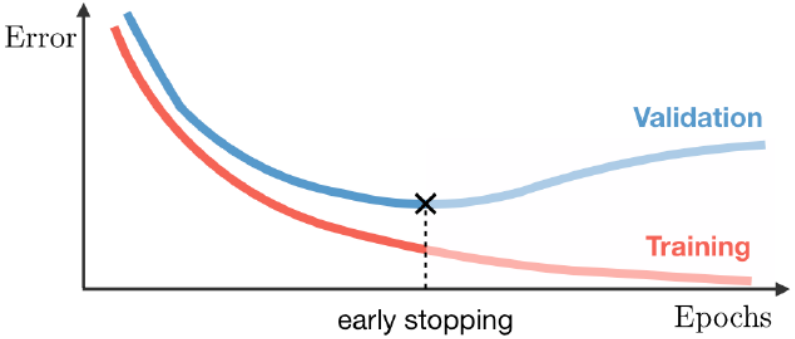
- Differences from fully connected nets
  - 3D volume of neurons
  - Local connectivity
  - Shared weights
- Hyper-parameter
  - Number of filters
  - Filter shape (receptive field)
  - Pooling type and shape
  - **Regularization**
    - Dropout
    - Early Stopping
    - Data Augmentation
    - Early Stopping
    - Norm constraints
    - L1/L2 regularization
  - Use performance over a validation set to pick hyperparameters



Original	Flip	Rotation	Random crop
- Image without any modification	- Flipped with respect to an axis for which the meaning of the image is preserved	- Rotation with a slight angle - Simulates incorrect horizon calibration	- Random focus on one part of the image - Several random crops can be done in a row



Color shift	Noise addition	Information loss	Contrast change
- Nuances of RGB is slightly changed - Captures noise that can occur with light exposure	- Addition of noise - More tolerance to quality variation of inputs	- Parts of image ignored - Mimics potential loss of parts of image	- Luminosity changes - Controls difference in exposition due to time of day



Regularization Mechanisms

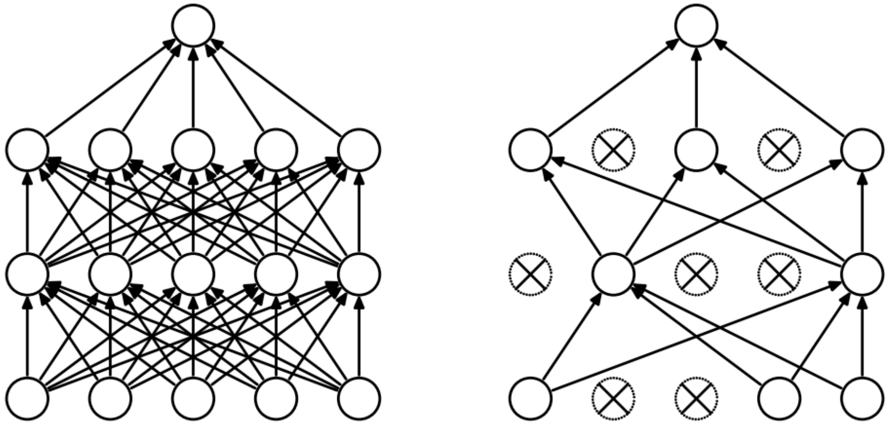
- L2 penalty to weights
  - Weight_decay parameter
    - sgd = torch.optim.SGD([w_torch], lr=lr, weight_decay=0.9)
- Handling vanishing (or exploding) gradients
  - Pre-training (old!)
  - Layerwise training
  - Drop-out
  - Batch Normalization
  - Normalization free architectures with weight and gradient clipping

`nn.Dropout(0.5)`

`nn.BatchNorm2d(6)`

Understanding Drop-out in training

- "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" by Srivastava et al., 2014.
  - Randomly drop units (along with their connections) from the neural network during training
  - Average weights across all "thinned" networks
  - Replaces explicit regularization and produces faster learning

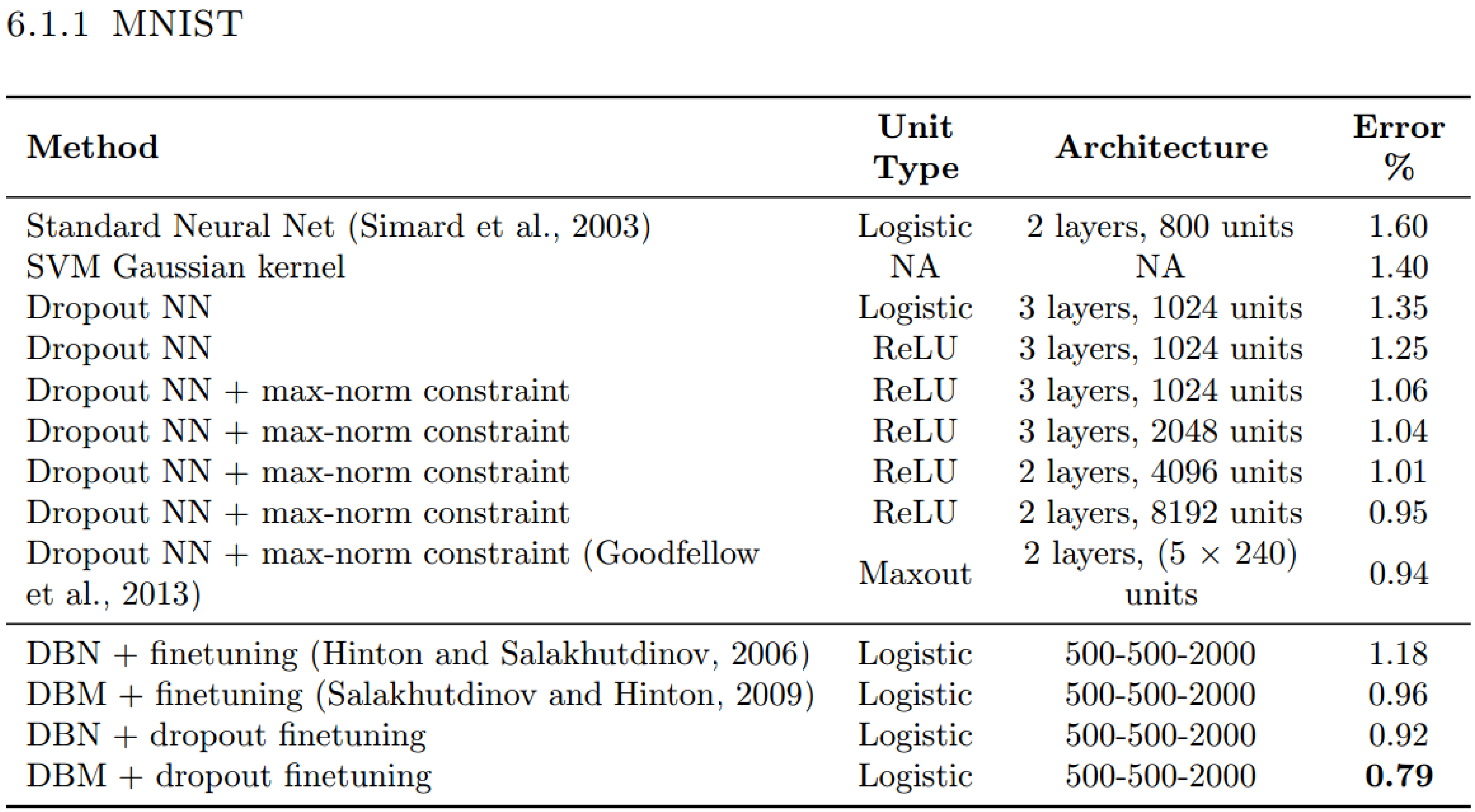


(a) Standard Neural Net

(b) After applying dropout.

Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Effect of Dropout



6.1.1 MNIST

Method	**Unit Type**	**Architecture**	**Error %**
Standard Neural Net (Simard et al., 2003)	Logistic	2 layers, 800 units	1.60
SVM Gaussian kernel	NA	NA	1.40
Dropout NN	Logistic	3 layers, 1024 units	1.35
Dropout NN	ReLU	3 layers, 1024 units	1.25
Dropout NN + max-norm constraint	ReLU	3 layers, 1024 units	1.06
Dropout NN + max-norm constraint	ReLU	3 layers, 2048 units	1.04
Dropout NN + max-norm constraint	ReLU	2 layers, 4096 units	1.01
Dropout NN + max-norm constraint	ReLU	2 layers, 8192 units	0.95
Dropout NN + max-norm constraint (Goodfellow et al., 2013)	Maxout	2 layers, (5 × 240) units	0.94
DBN + finetuning (Hinton and Salakhutdinov, 2006)	Logistic	500-500-2000	1.18
DBM + finetuning (Salakhutdinov and Hinton, 2009)	Logistic	500-500-2000	0.96
DBN + dropout finetuning	Logistic	500-500-2000	0.92
DBM + dropout finetuning	Logistic	500-500-2000	**0.79**

Does drop out help with overfitting and underfitting?

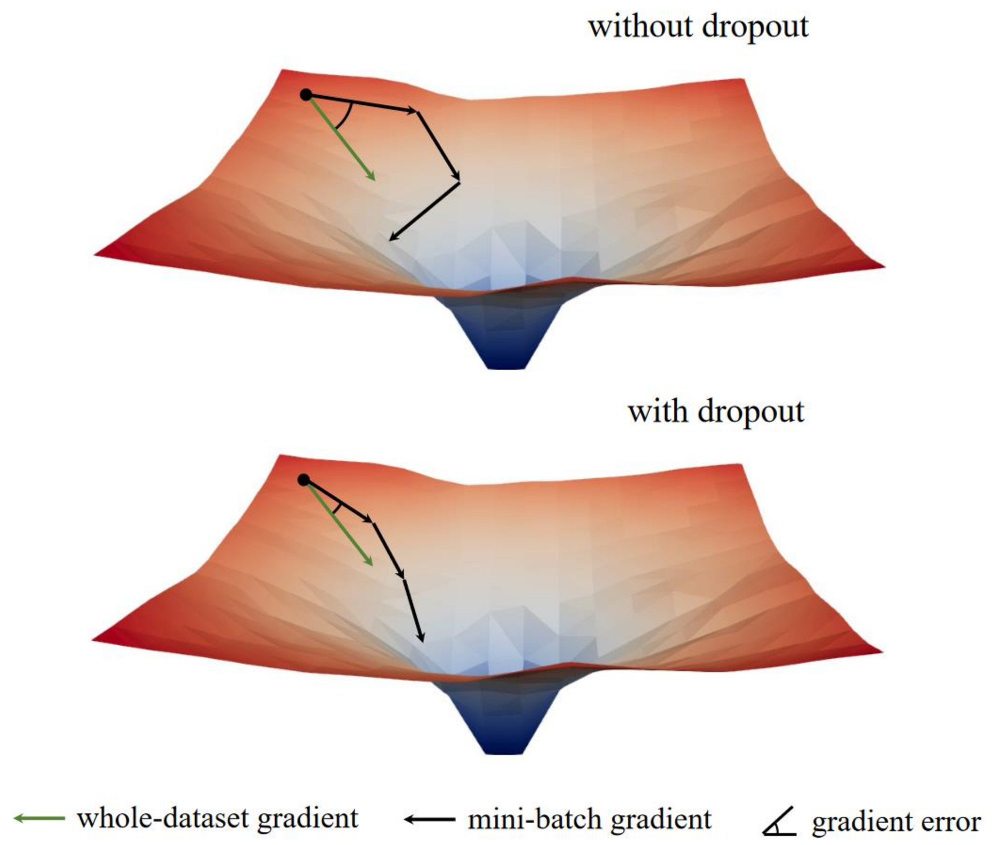
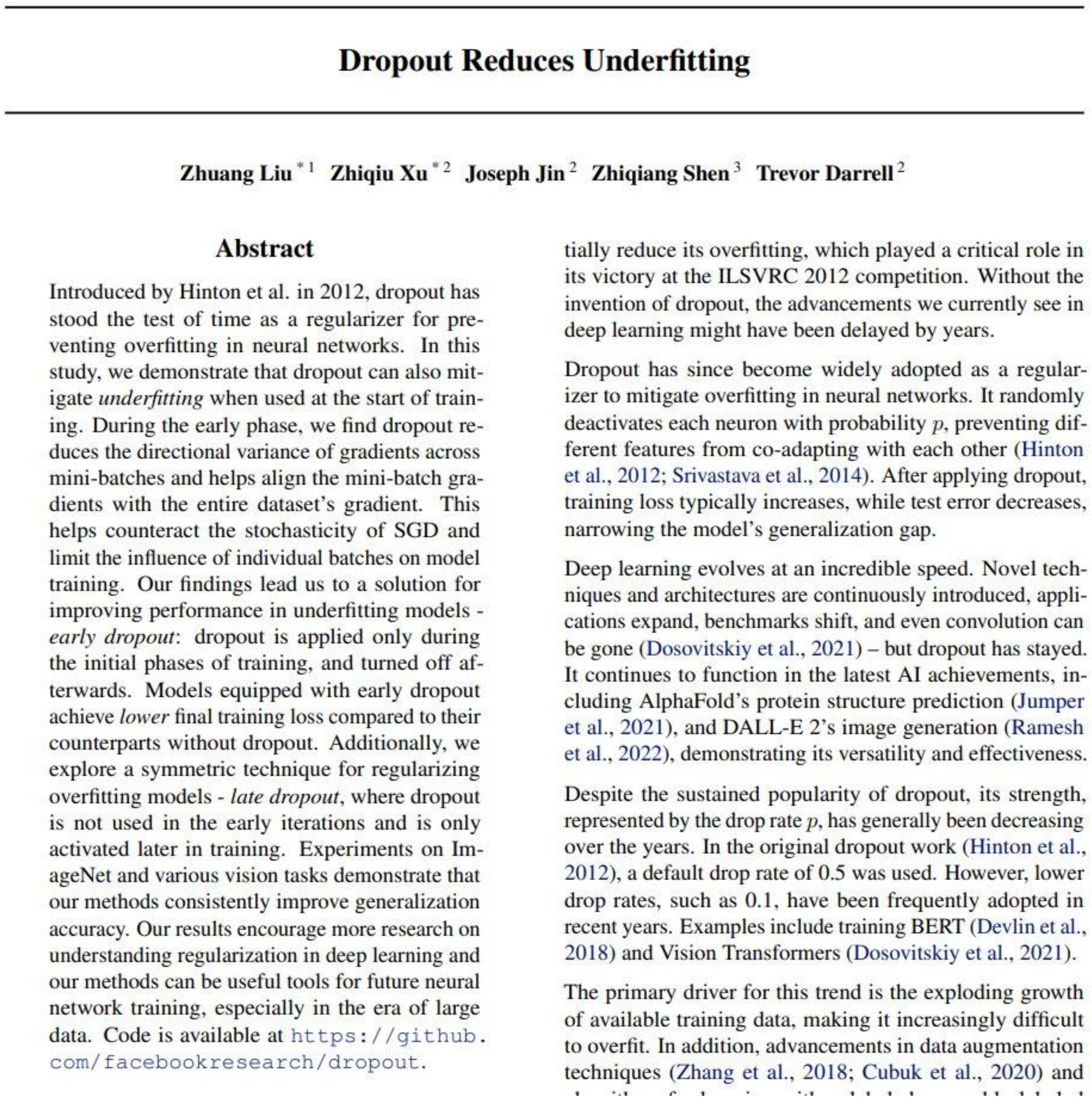


Figure 1. **Dropout in early training** helps the model produce mini-batch gradient directions that are more consistent and aligned with the overall gradient of the entire dataset.



Dropout Reduces Underfitting

Zhuang Liu [* 1] **Zhiqiu Xu** [* 2] **Joseph Jin** [2] **Zhiqiang Shen** [3] **Trevor Darrell** [2]

Abstract

Introduced by Hinton et al. in 2012, dropout has stood the test of time as a regularizer for preventing overfitting in neural networks. In this study, we demonstrate that dropout can also mitigate *underfitting* when used at the start of training. During the early phase, we find dropout reduces the directional variance of gradients across mini-batches and helps align the mini-batch gradients with the entire dataset's gradient. This helps counteract the stochasticity of SGD and limit the influence of individual batches on model training. Our findings lead us to a solution for improving performance in underfitting models - *early dropout*: dropout is applied only during the initial phases of training, and turned off afterwards. Models equipped with early dropout achieve *lower* final training loss compared to their counterparts without dropout. Additionally, we explore a symmetric technique for regularizing overfitting models - *late dropout*, where dropout is not used in the early iterations and is only activated later in training. Experiments on ImageNet and various vision tasks demonstrate that our methods consistently improve generalization accuracy. Our results encourage more research on understanding regularization in deep learning and our methods can be useful tools for future neural network training, especially in the era of large data. Code is available at `https://github.com/facebookresearch/dropout`.

tially reduce its overfitting, which played a critical role in its victory at the ILSVRC 2012 competition. Without the invention of dropout, the advancements we currently see in deep learning might have been delayed by years.

Dropout has since become widely adopted as a regularizer to mitigate overfitting in neural networks. It randomly deactivates each neuron with probability p, preventing different features from co-adapting with each other (Hinton et al., 2012; Srivastava et al., 2014). After applying dropout, training loss typically increases, while test error decreases, narrowing the model's generalization gap.

Deep learning evolves at an incredible speed. Novel techniques and architectures are continuously introduced, applications expand, benchmarks shift, and even convolution can be gone (Dosovitskiy et al., 2021) – but dropout has stayed. It continues to function in the latest AI achievements, including AlphaFold's protein structure prediction (Jumper et al., 2021), and DALL-E 2's image generation (Ramesh et al., 2022), demonstrating its versatility and effectiveness.

Despite the sustained popularity of dropout, its strength, represented by the drop rate p, has generally been decreasing over the years. In the original dropout work (Hinton et al., 2012), a default drop rate of 0.5 was used. However, lower drop rates, such as 0.1, have been frequently adopted in recent years. Examples include training BERT (Devlin et al., 2018) and Vision Transformers (Dosovitskiy et al., 2021).

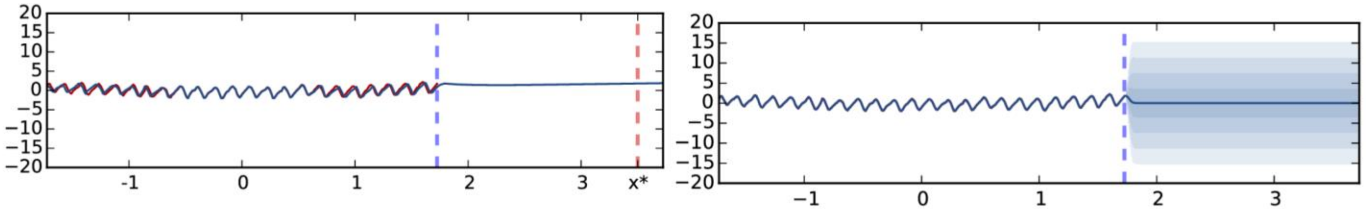
The primary driver for this trend is the exploding growth of available training data, making it increasingly difficult to overfit. In addition, advancements in data augmentation techniques (Zhang et al., 2018; Cubuk et al., 2020) and

Dropout in testing: MCDropout

- Quantifying uncertainty in neural network predictions
  - Use **drop-out at test time** and average the results (and compute error bounds)

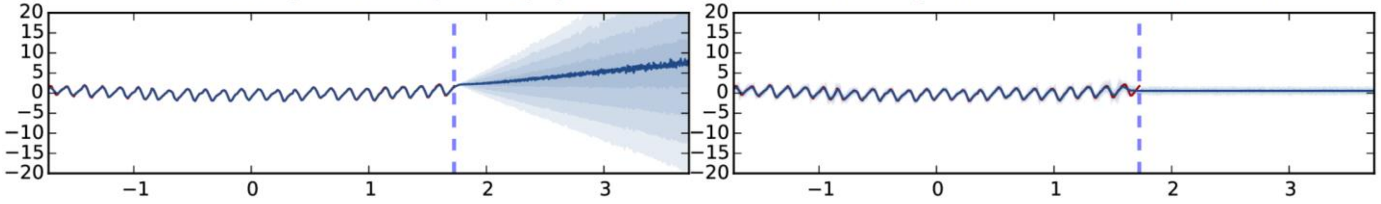
$$\mathbb{E}_{q(\mathbf{y}^*|\mathbf{x}^*)}(\mathbf{y}^*) \approx \frac{1}{T}\sum_{t=1}^{T}\widehat{\mathbf{y}}^*(\mathbf{x}^*, \mathbf{W}_1^t, ..., \mathbf{W}_L^t)$$

Consider a model with L layers with the weights of each obtained through a drop-out in T trials



(a) Standard dropout with weight averaging

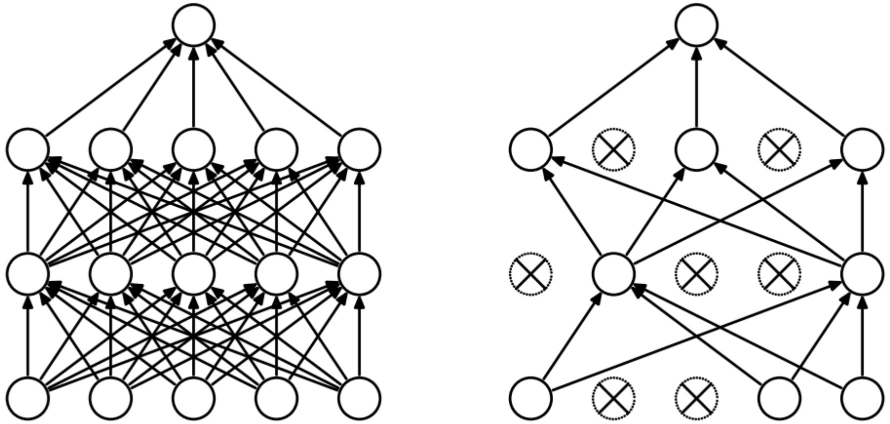
(b) Gaussian process with SE covariance function



(c) MC dropout with ReLU non-linearities

(d) MC dropout with TanH non-linearities

Figure 2. **Predictive mean and uncertainties on the Mauna Loa CO$_2$ concentrations dataset, for various models.** In red is the observed function (left of the dashed blue line); in blue is the predictive mean plus/minus two standard deviations (8 for fig. 2d). Different shades of blue represent half a standard deviation. Marked with a dashed red line is a point far away from the data: standard dropout confidently predicts an insensible value for the point; the other models predict insensible values as well but with the additional information that the models are uncertain about their predictions.



(a) Standard Neural Net

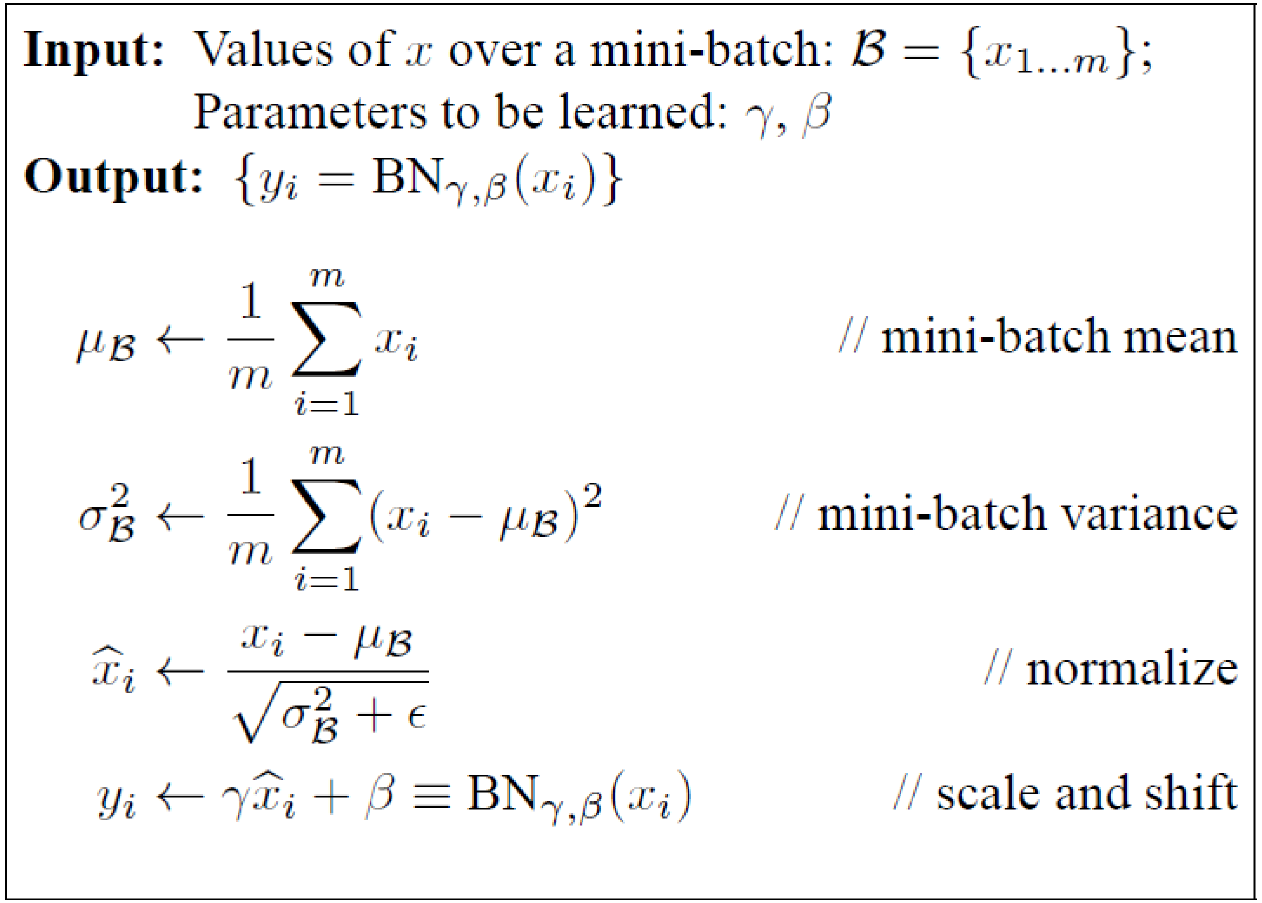
(b) After applying dropout.

Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Gal, Yarin, and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." arXiv, October 4, 2016. https://doi.org/10.48550/arXiv.1506.02142.

Understanding Batch-Normalization

- Given a batch of N examples, each dimension of each example is normalized to zero mean and unit variance
- Minimizes “covariate shift”
  - a change in the distribution of a function’s domain
  - Input changes and now the function cannot deal with it
  - Layer to layer changes
- Accelerates learning by preventing learning stalls
- Important Note: Keep batch norm parameter learning active only in training



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167v3*, 2015.

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

Effect of Batch Normalization

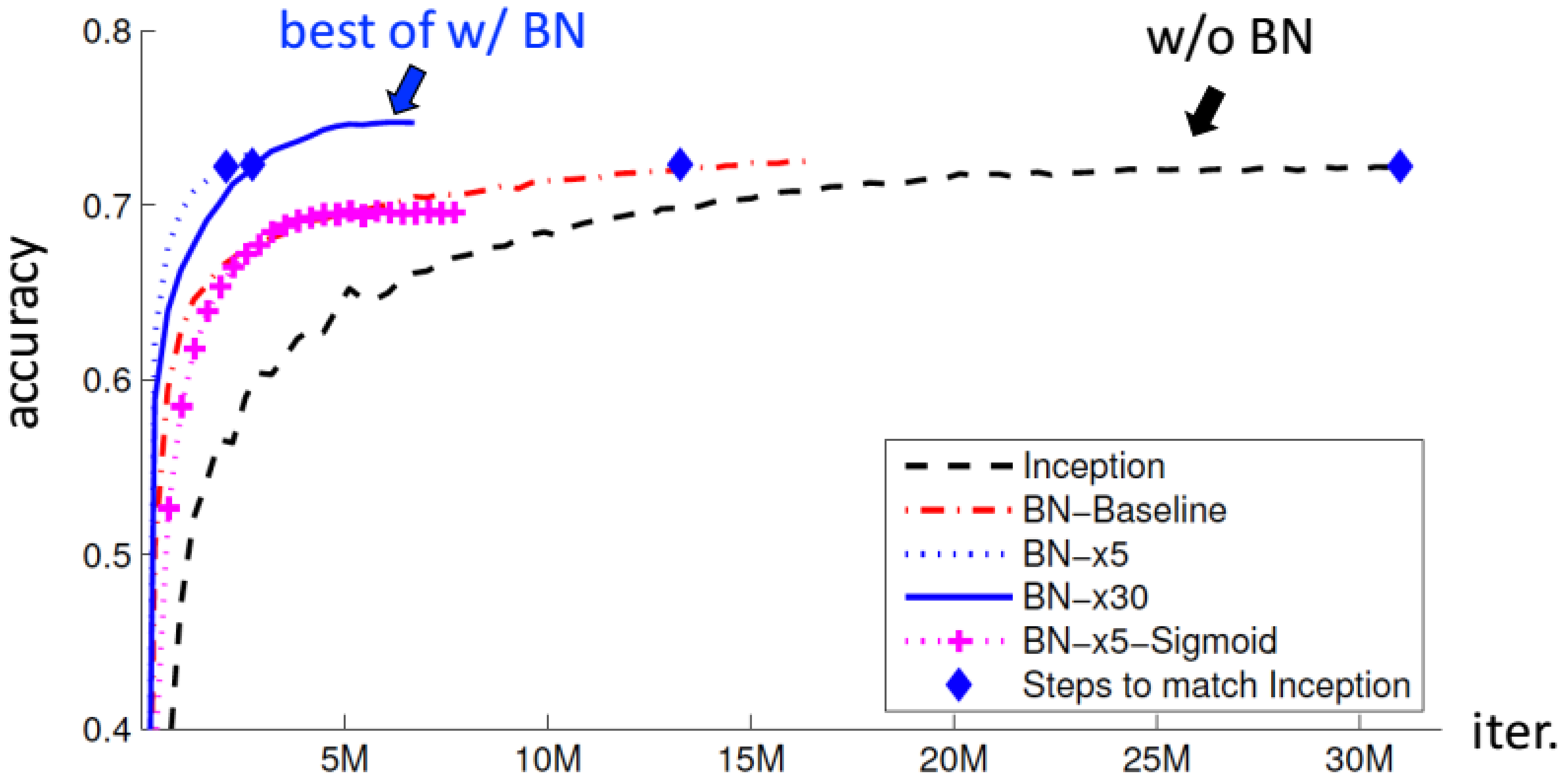
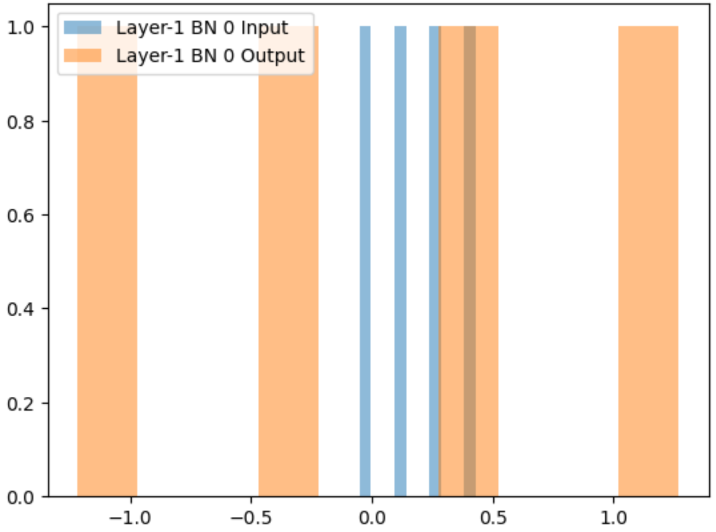
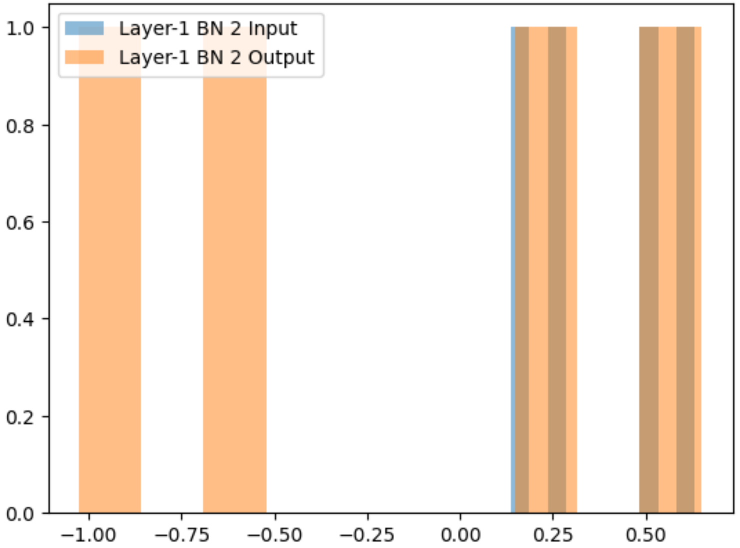


Figure taken from [S. Ioffe & C. Szegedy]

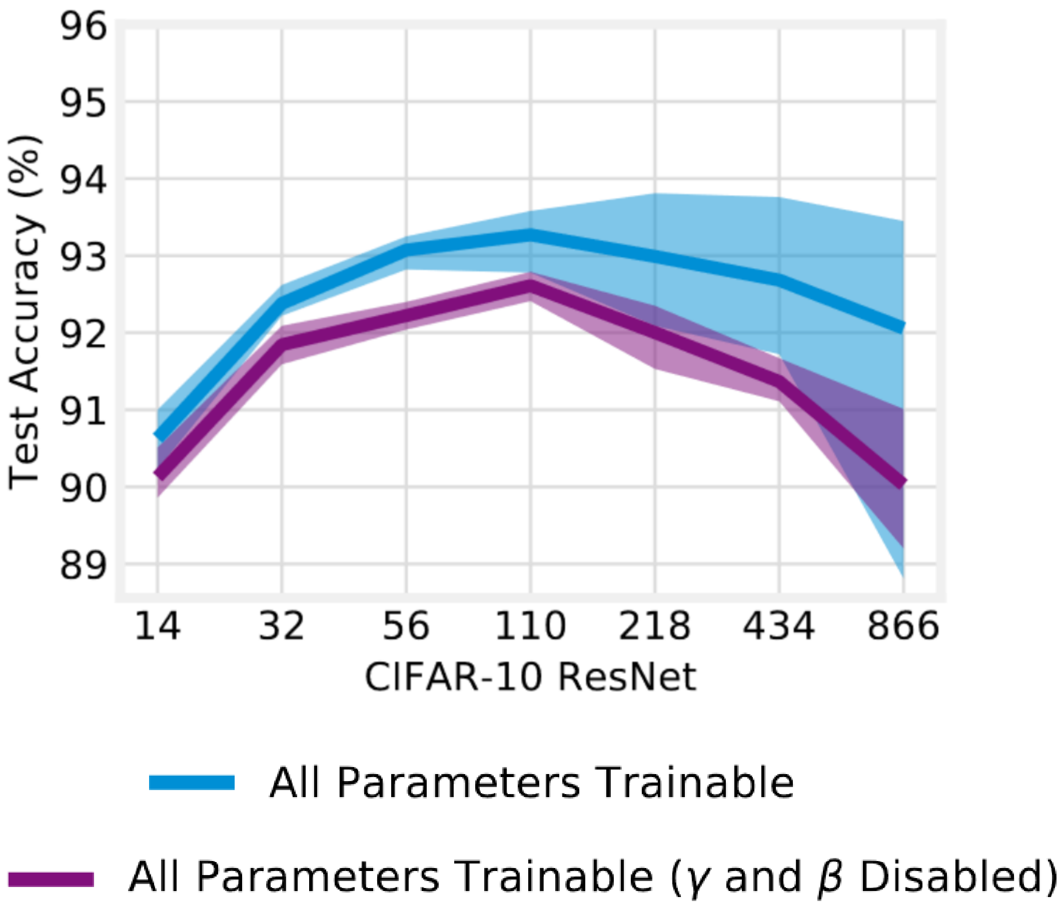
Batch Normalization Coding

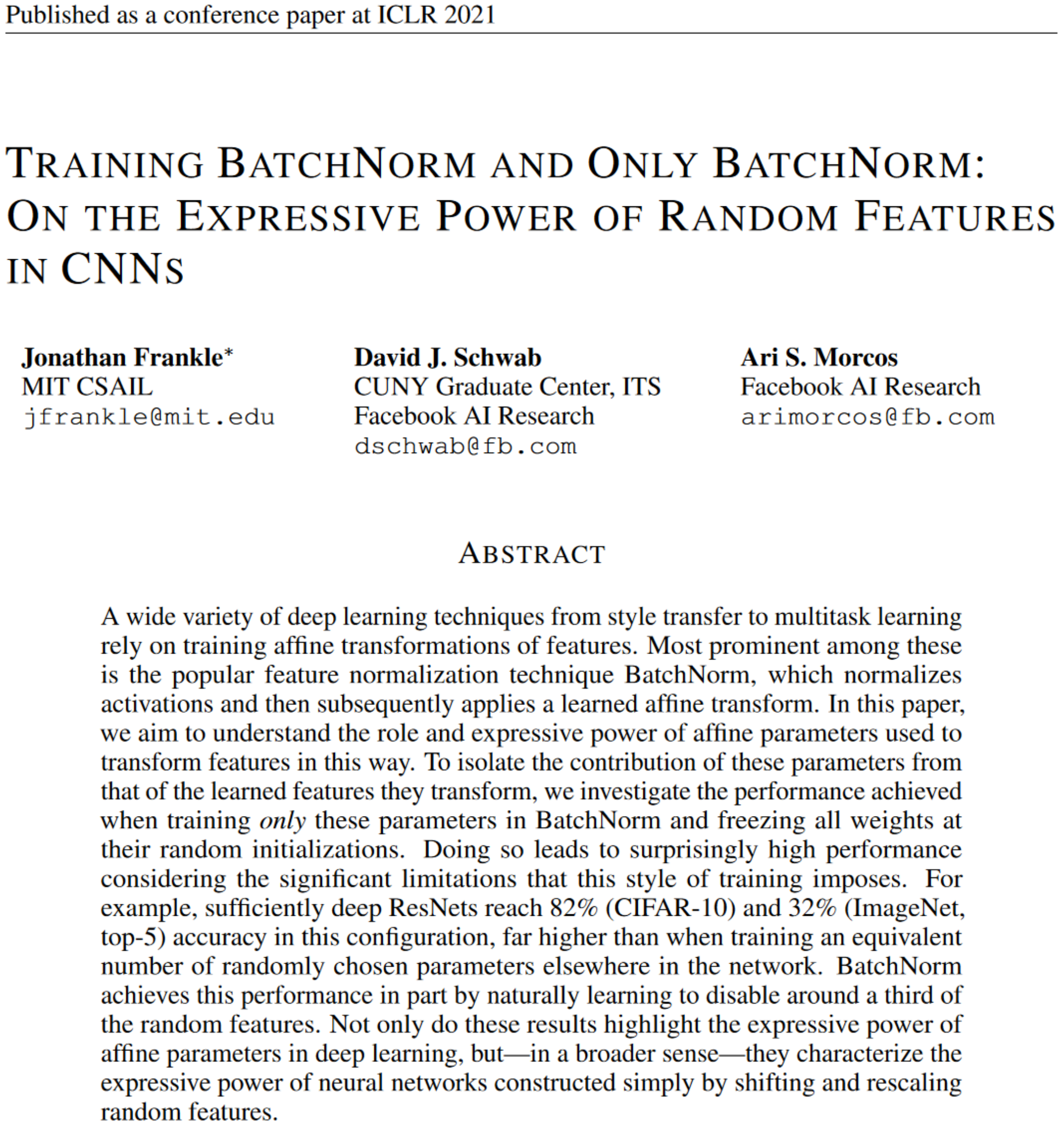
- See: https://github.com/foxtrotmike/CS909/blob/master/xornet_batch_normalization.ipynb
- Compare the distributions of data before and after batch normalization: Better range of data after batch normalization
  - Both positive and negative values in outputs





What can you do with just training batch norm parameters?




Published as a conference paper at ICLR 2021

TRAINING BATCHNORM AND ONLY BATCHNORM: ON THE EXPRESSIVE POWER OF RANDOM FEATURES IN CNNS

Jonathan Frankle*
MIT CSAIL
jfrankle@mit.edu

David J. Schwab
CUNY Graduate Center, ITS
Facebook AI Research
dschwab@fb.com

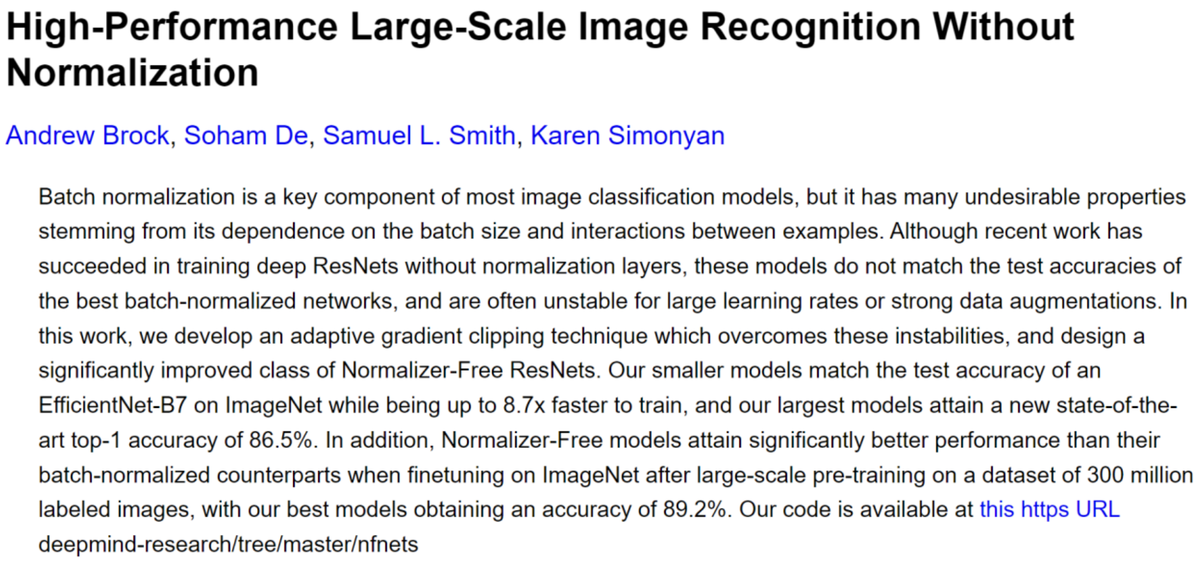
Ari S. Morcos
Facebook AI Research
arimorcos@fb.com

ABSTRACT

A wide variety of deep learning techniques from style transfer to multitask learning rely on training affine transformations of features. Most prominent among these is the popular feature normalization technique BatchNorm, which normalizes activations and then subsequently applies a learned affine transform. In this paper, we aim to understand the role and expressive power of affine parameters used to transform features in this way. To isolate the contribution of these parameters from that of the learned features they transform, we investigate the performance achieved when training *only* these parameters in BatchNorm and freezing all weights at their random initializations. Doing so leads to surprisingly high performance considering the significant limitations that this style of training imposes. For example, sufficiently deep ResNets reach 82% (CIFAR-10) and 32% (ImageNet, top-5) accuracy in this configuration, far higher than when training an equivalent number of randomly chosen parameters elsewhere in the network. BatchNorm achieves this performance in part by naturally learning to disable around a third of the random features. Not only do these results highlight the expressive power of affine parameters in deep learning, but—in a broader sense—they characterize the expressive power of neural networks constructed simply by shifting and rescaling random features.

What can you do without batch normalization?

- Batch normalization requires a sufficient large batch size to allow effective estimation of mean and variance of each batch which can be a problem for large input data or low memory machines



High-Performance Large-Scale Image Recognition Without Normalization

Andrew Brock, Soham De, Samuel L. Smith, Karen Simonyan

Batch normalization is a key component of most image classification models, but it has many undesirable properties stemming from its dependence on the batch size and interactions between examples. Although recent work has succeeded in training deep ResNets without normalization layers, these models do not match the test accuracies of the best batch-normalized networks, and are often unstable for large learning rates or strong data augmentations. In this work, we develop an adaptive gradient clipping technique which overcomes these instabilities, and design a significantly improved class of Normalizer-Free ResNets. Our smaller models match the test accuracy of an EfficientNet-B7 on ImageNet while being up to 8.7x faster to train, and our largest models attain a new state-of-the-art top-1 accuracy of 86.5%. In addition, Normalizer-Free models attain significantly better performance than their batch-normalized counterparts when finetuning on ImageNet after large-scale pre-training on a dataset of 300 million labeled images, with our best models obtaining an accuracy of 89.2%. Our code is available at this https URL deepmind-research/tree/master/nfnets

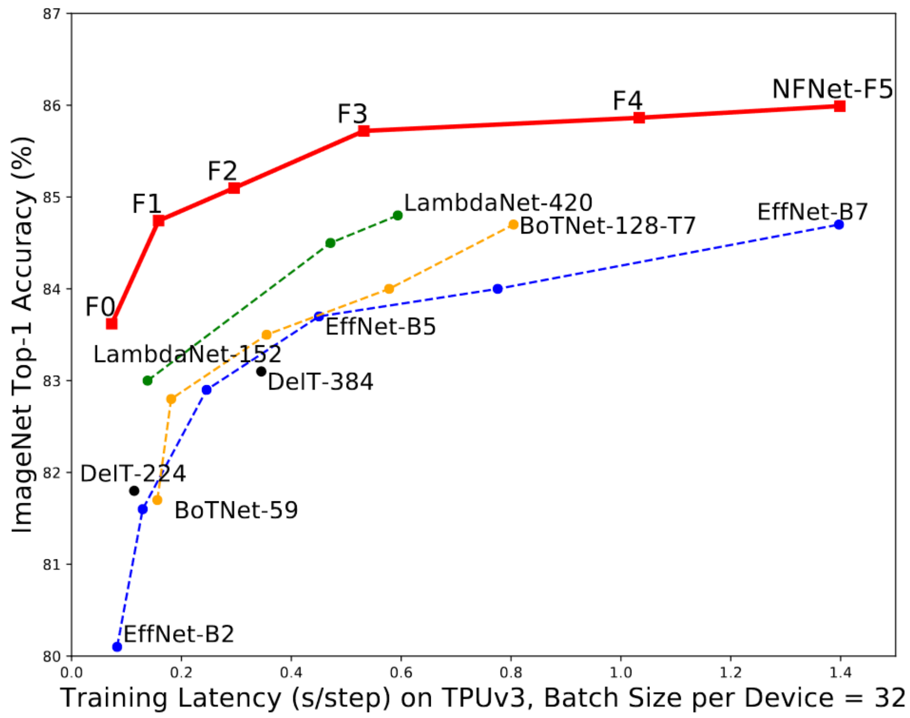
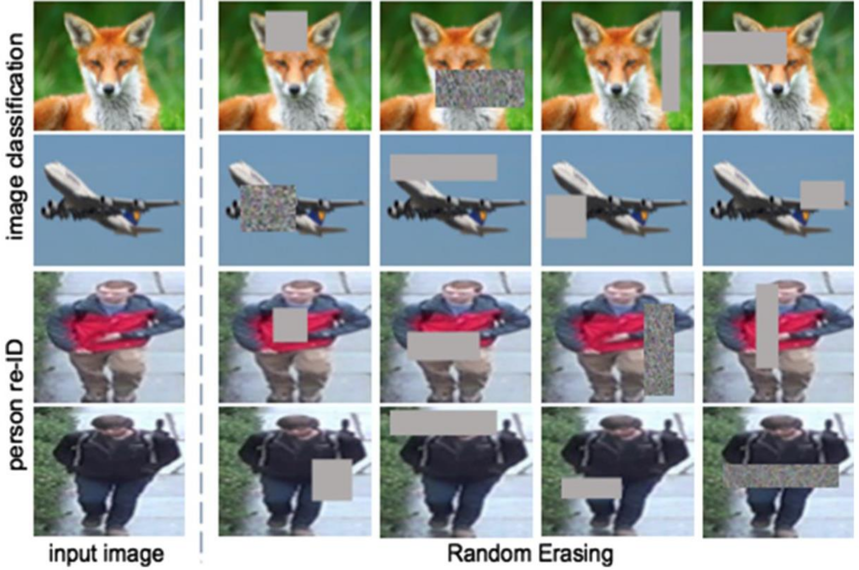


Figure 1. **ImageNet Validation Accuracy vs Training Latency.** All numbers are single-model, single crop. Our NFNet-F1 model achieves comparable accuracy to an EffNet-B7 while being 8.7× faster to train. Our NFNet-F5 model has similar training latency to EffNet-B7, but achieves a state-of-the-art 86.0% top-1 accuracy on ImageNet. We further improve on this using Sharpness Aware Minimization (Foret et al., 2021) to achieve 86.5% top-1 accuracy.

Data Augmentation


Image Data Augmentation
Basic Image Manipulations
Deep Learning Approaches
Kernel Filters
Color Space Transformations
Random Erasing
Geometric Transformations
Mixing Images
Adversarial Training
Neural Style Transfer
GAN Data Augmentation
Meta Learning
Neural Augmentation
AutoAugment
Smart Augmentation



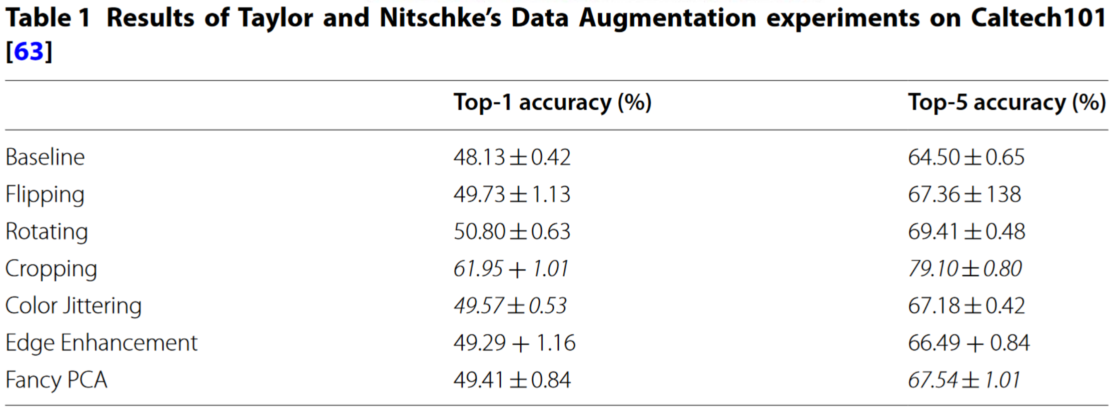


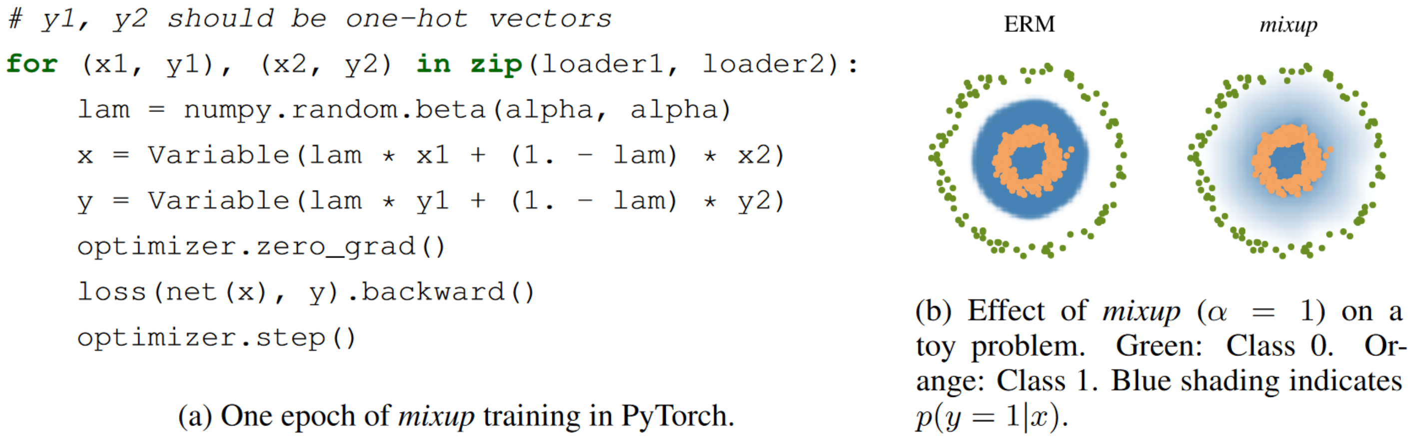
Table 1 Results of Taylor and Nitschke's Data Augmentation experiments on Caltech101 [63]

	Top-1 accuracy (%)	Top-5 accuracy (%)
Baseline	48.13 ± 0.42	64.50 ± 0.65
Flipping	49.73 ± 1.13	67.36 ± 138
Rotating	50.80 ± 0.63	69.41 ± 0.48
Cropping	*61.95 + 1.01*	*79.10 ± 0.80*
Color Jittering	*49.57 ± 0.53*	67.18 ± 0.42
Edge Enhancement	49.29 + 1.16	66.49 + 0.84
Fancy PCA	49.41 ± 0.84	*67.54 ± 1.01*

Shorten, Connor, and Taghi M. Khoshgoftaar. "A Survey on Image Data Augmentation for Deep Learning." *Journal of Big Data* 6, no. 1 (July 6, 2019): 60. https://doi.org/10.1186/s40537-019-0197-0.

Data Augmentation

MixUp

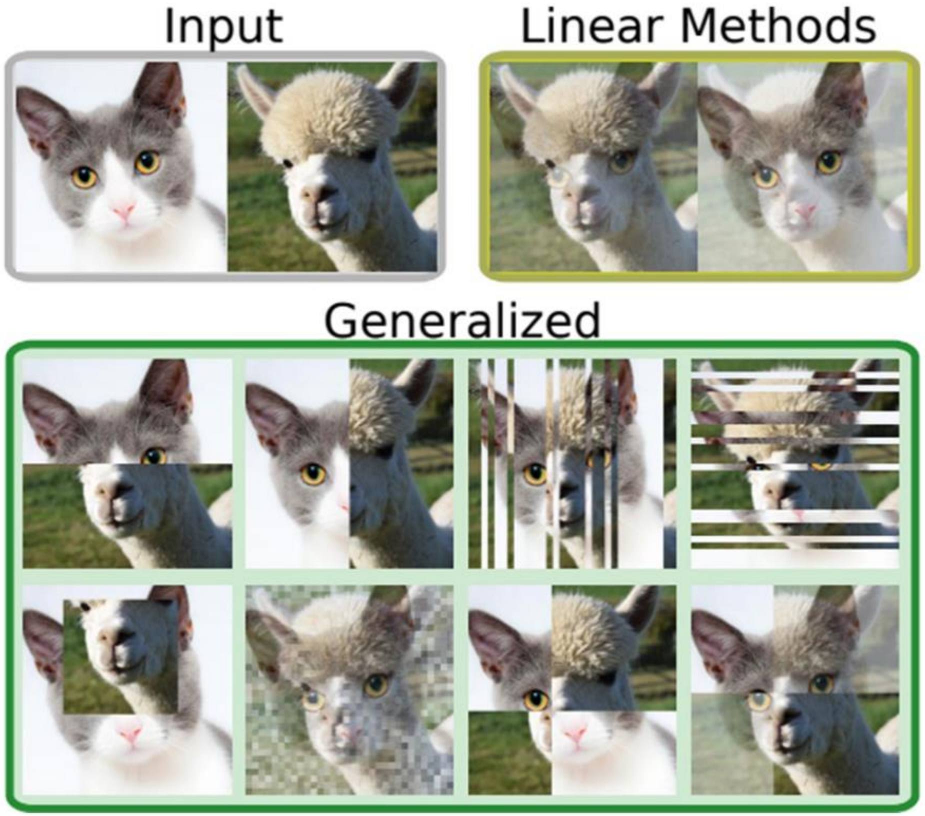


```
# y1, y2 should be one-hot vectors
for (x1, y1), (x2, y2) in zip(loader1, loader2):
    lam = numpy.random.beta(alpha, alpha)
    x = Variable(lam * x1 + (1. - lam) * x2)
    y = Variable(lam * y1 + (1. - lam) * y2)
    optimizer.zero_grad()
    loss(net(x), y).backward()
    optimizer.step()
```

(a) One epoch of *mixup* training in PyTorch.

(b) Effect of *mixup* ($\alpha = 1$) on a toy problem. Green: Class 0. Orange: Class 1. Blue shading indicates $p(y = 1|x)$.

Figure 1: Illustration of *mixup*, which converges to ERM as $\alpha \to 0$.



Libraries

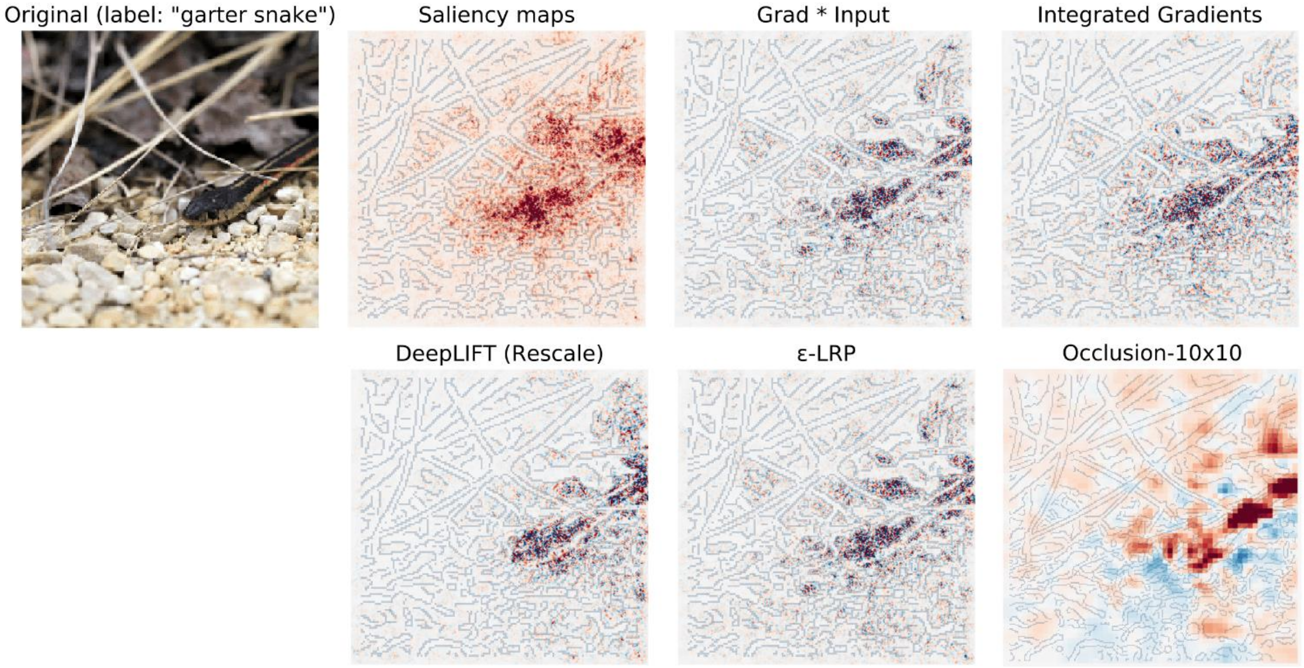
https://pytorch.org/vision/stable/transforms.html
https://albumentations.ai/
https://kornia.readthedocs.io/en/latest/augmentation.html

Zhang, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. “Mixup: Beyond Empirical Risk Minimization.” arXiv, April 27, 2018. https://doi.org/10.48550/arXiv.1710.09412.

What is my model doing? What is my model learning?

- Interpretability
  - Interpret why a certain model is producing a certain output for a given input
  - “What is the model doing?”
- Explainable
  - Explaining the “behavior” of the model or “What is the model learnin?”

- Model Agnostic Methods
    - Permutation Feature Invariance
    - LIME Analysis
    - SHAP Analysis
- For CNNs
  - Pixel Attribution (Saliency Maps)
    - Score-CAM
    - Grad-CAM
  - Testing with Concept Activation Vectors (TCAV)
  - DeepSHAP

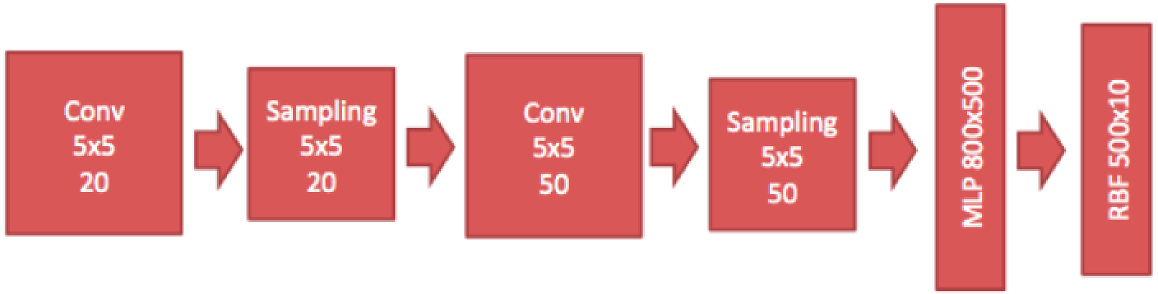


https://github.com/marcoancona/DeepExplain

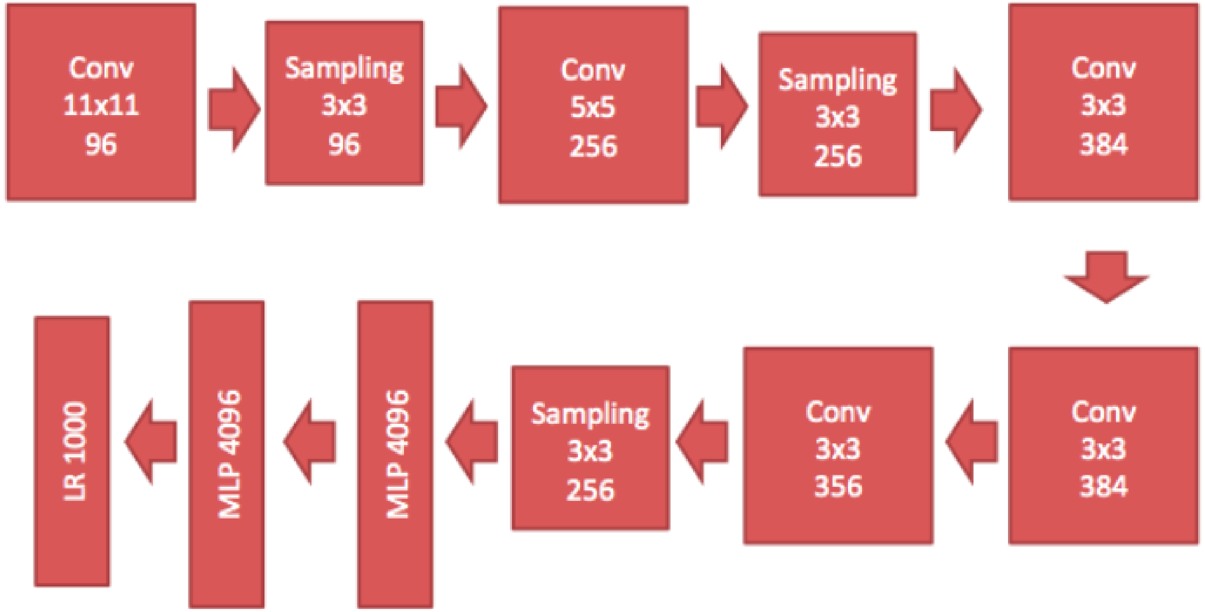
Great Resource on interpretable machine learning:
https://christophm.github.io/interpretable-ml-book/

Famous CNN

- LeNet (Le Cunn 1990, 1998)


Conv 5x5 20
Sampling 5x5 20
Conv 5x5 50
Sampling 5x5 50
MLP 800x500
RBF 500x10

- AlexNet
- VGG19
- Inception
- Xception
- EfficientNet

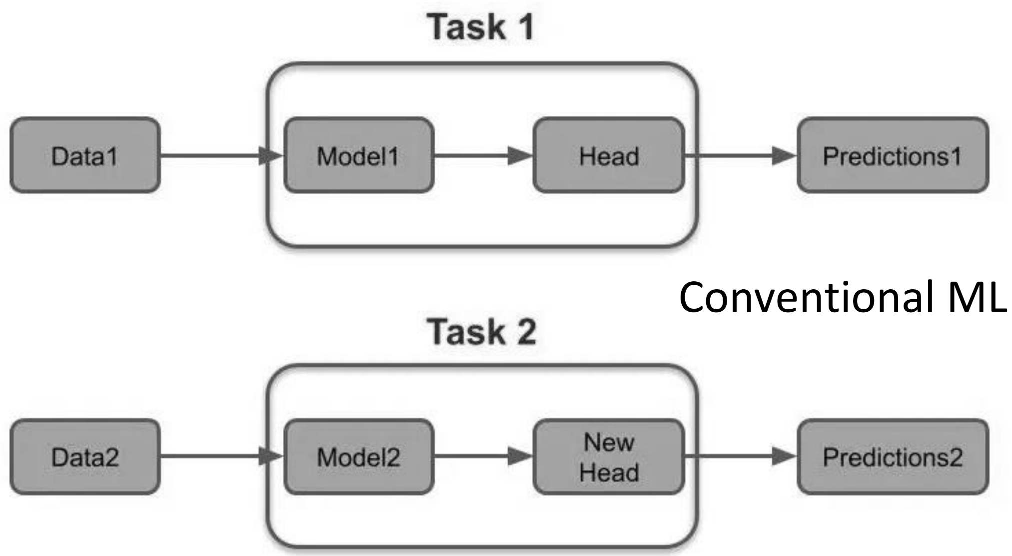

Conv 11x11 96
Sampling 3x3 96
Conv 5x5 256
Sampling 3x3 256
Conv 3x3 384
Conv 3x3 384
Conv 3x3 356
Sampling 3x3 256
MLP 4096
MLP 4096
LR 1000

Transfer Learning and Fine Tuning

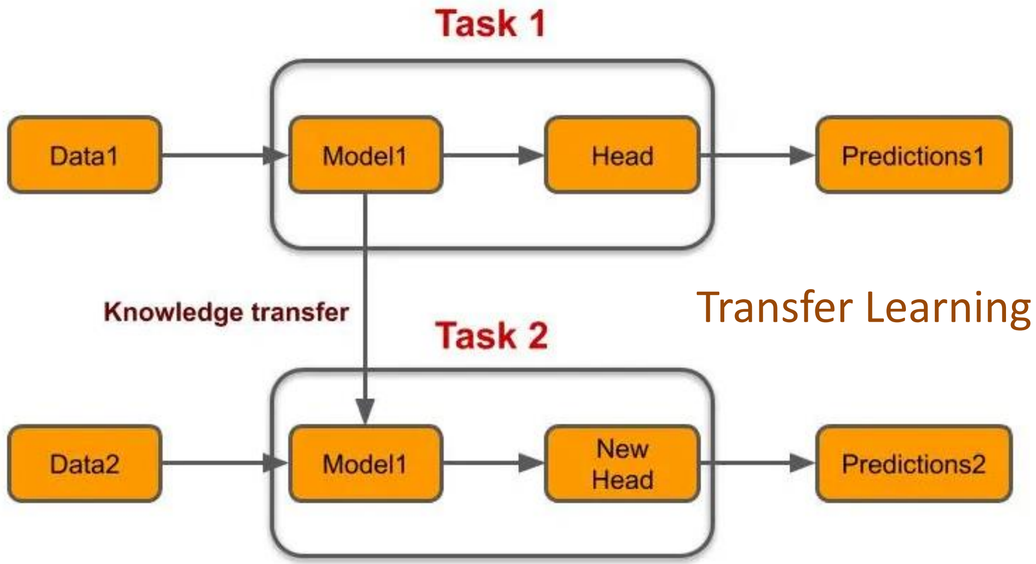
- Use a pretrained network for one task
- Keep the convolutional layers fixed (frozen)
- Freezing layers

  -
    ```
    for param in vgg.features.parameters():
    param.requires_grad = False
    ```
- **Transfer Learning:** Train the last layers (fully connected) for your task and/or add more layers as needed
- **Fine tuning:** Modify the weights of a few convolutional layers too

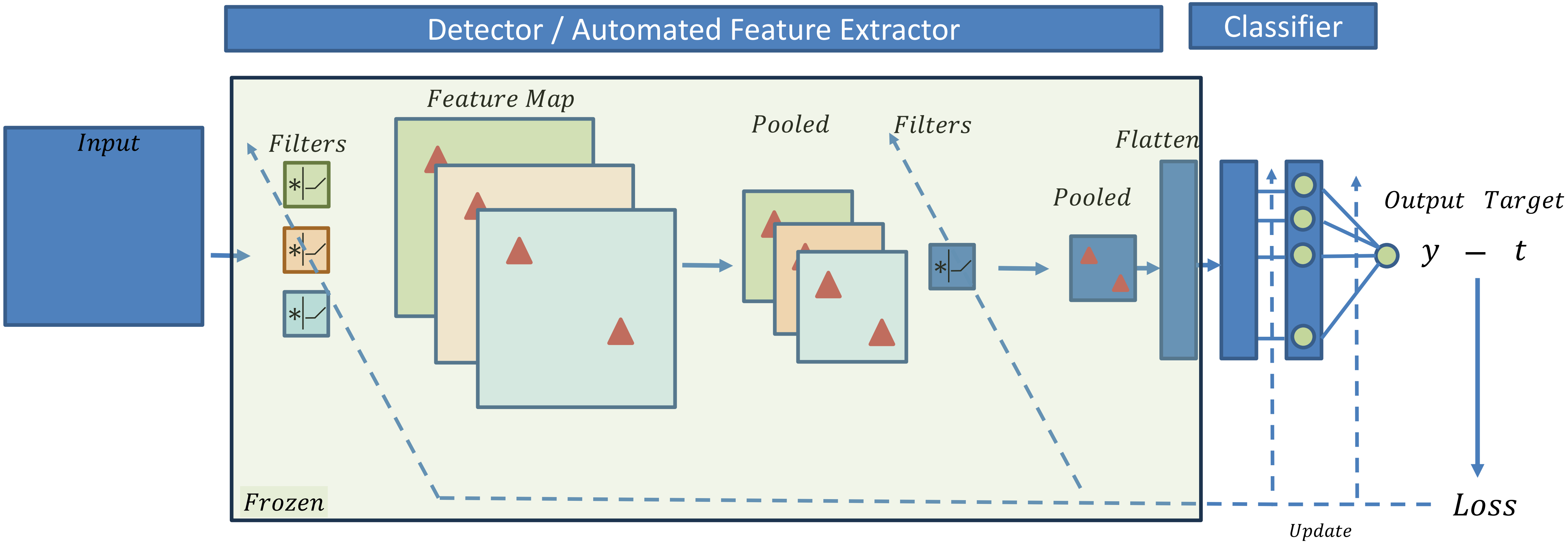
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
https://jimmy-shen.medium.com/pytorch-freeze-part-of-the-layers-4554105e03a6#



Conventional ML



Transfer Learning



We can choose which layers to freeze depending upon the application and the level of similarity between tasks

Advanced: Adapters

- Generalize the concept of transfer learning

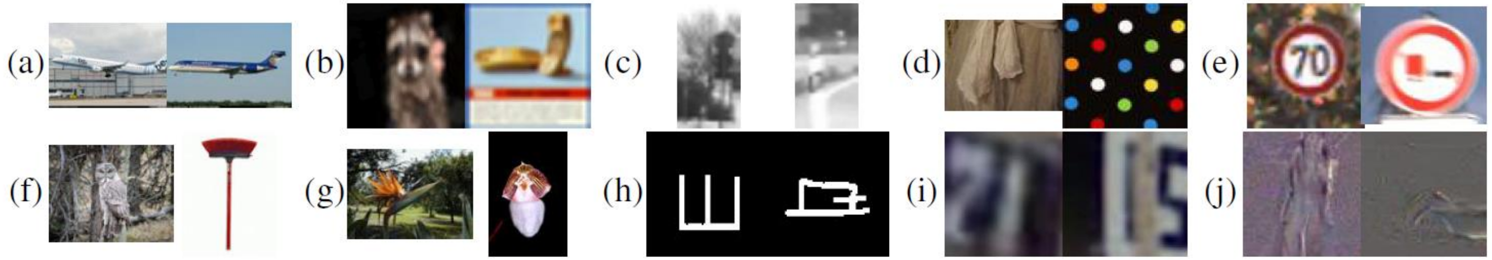


Figure 1: **Visual Decathlon.** We explore deep architectures that can learn simultaneously different tasks from very different visual domains. We experiment with ten representative ones: (a) Aircraft, (b) CIFAR-100, (c) Daimler Pedestrians, (d) Describable Textures, (e) German Traffic Signs, (f) ILSVRC (ImageNet) 2012, (g) VGG-Flowers, (h) OmniGlot, (i) SVHN, (j) UCF101 Dynamic Images.

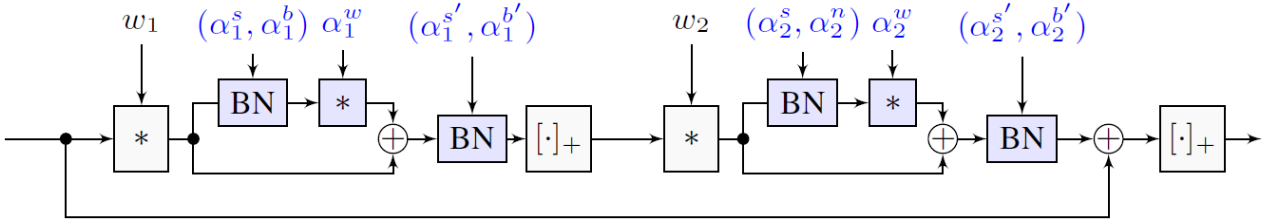
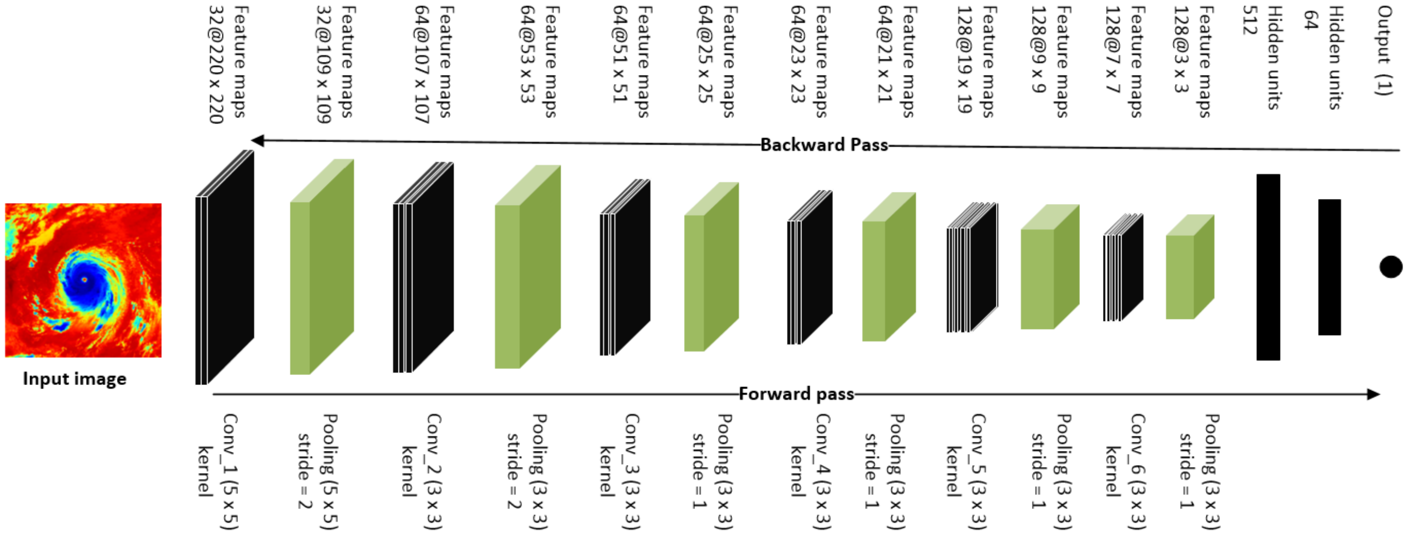


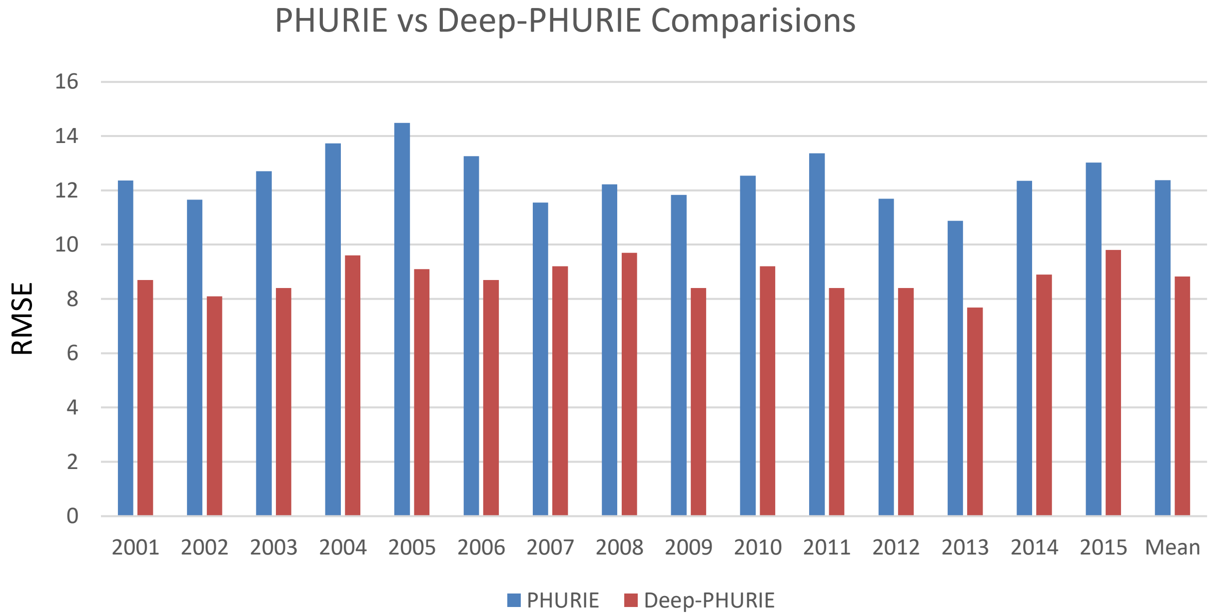
Figure 2: **Residual adapter modules.** The figure shows a standard residual module with the inclusion of adapter modules (in blue). The filter coefficients (w_1, w_2) are domain-agnostic and contains the vast majority of the model parameters; (α_1, α_2) contain instead a small number of domain-specific parameters.

Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi. "Learning Multiple Visual Domains with Residual Adapters." arXiv, November 27, 2017. https://doi.org/10.48550/arXiv.1705.08045.

Predicting Hurricane Intensities

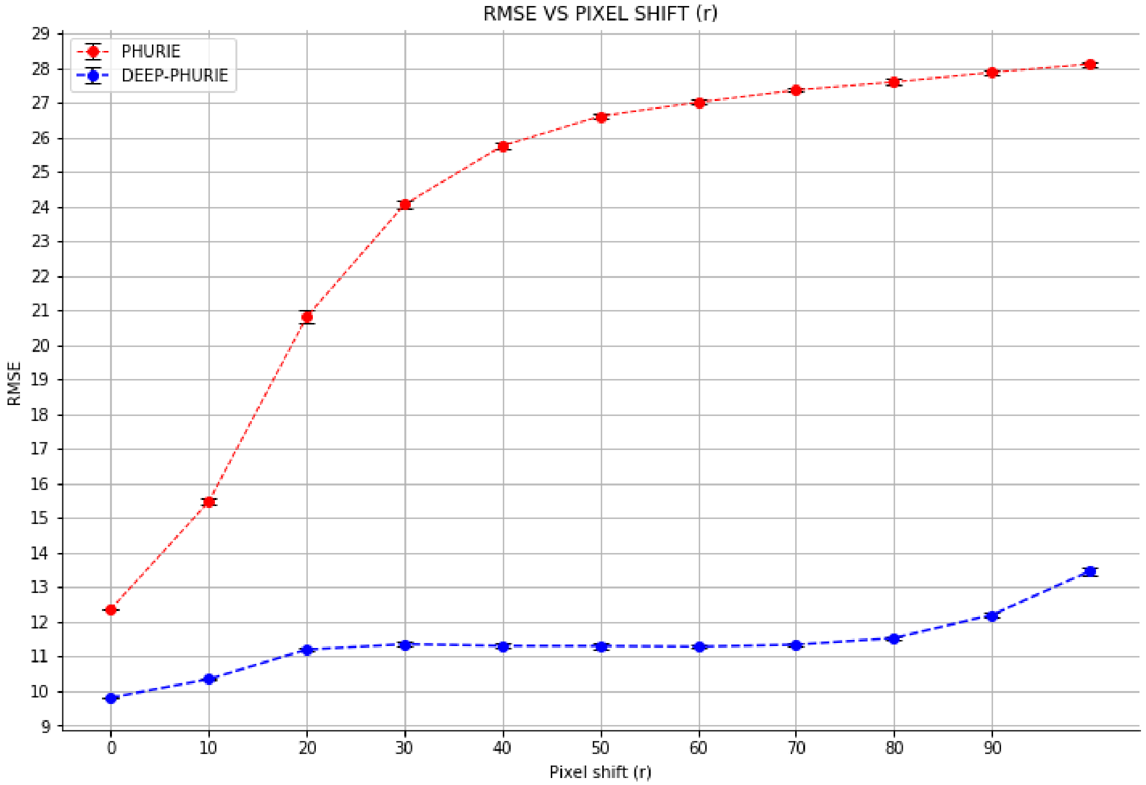
- Deep-PHURIE

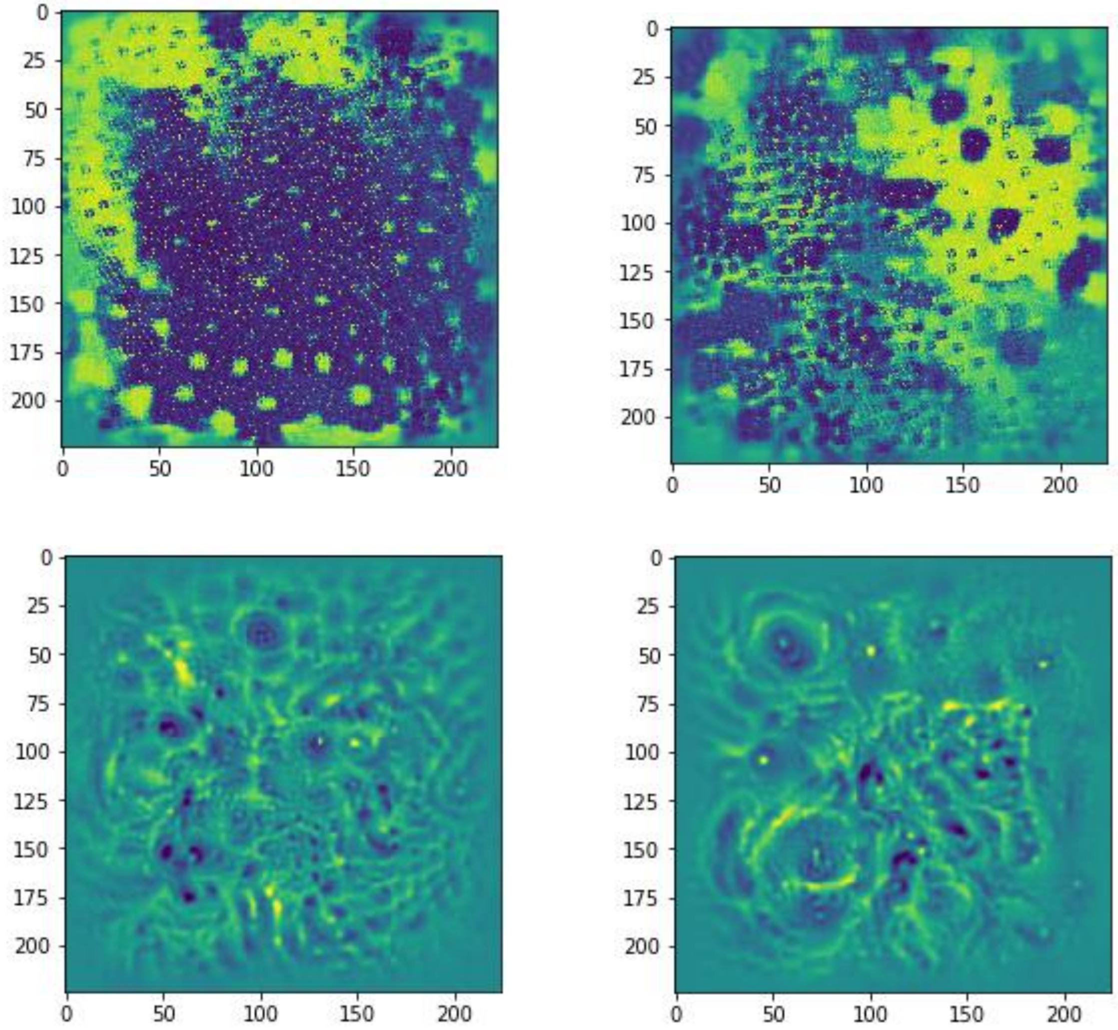




https://link.springer.com/article/10.1007/s00521-019-04410-7

Deep-PHURIE Robustness Analysis





Activation Maps for Deep PHURIE

Types of Neural Networks

- “Fully Connected”/Dense Feed Forward Backpropagation **multi-layer perceptrons**
- Convolutional neural networks

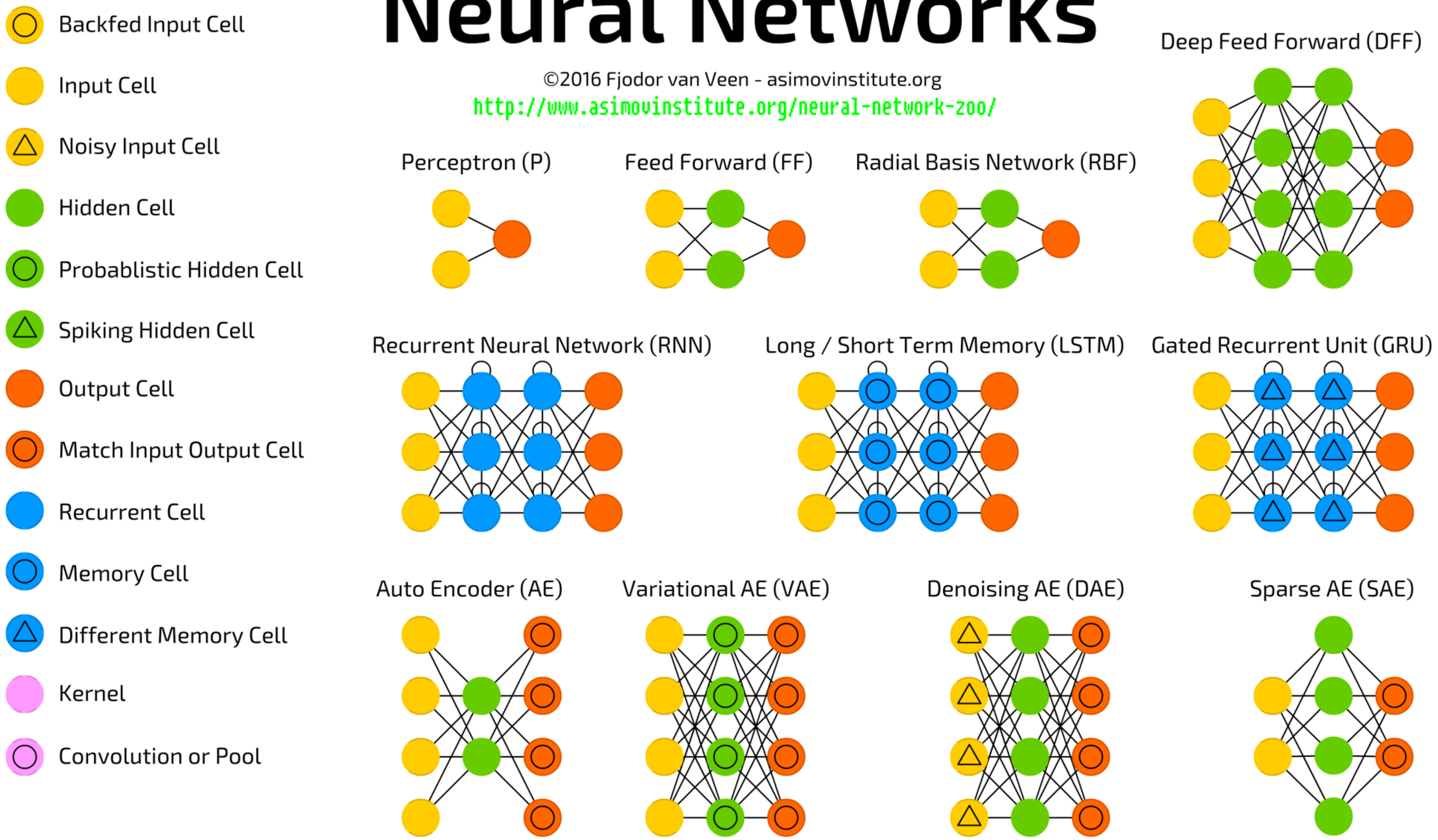
- Residual Neural networks
- Recurrent neural networks
- Auto-encoders
- Adversarial Networks
- Transformers
- Graph Neural Networks

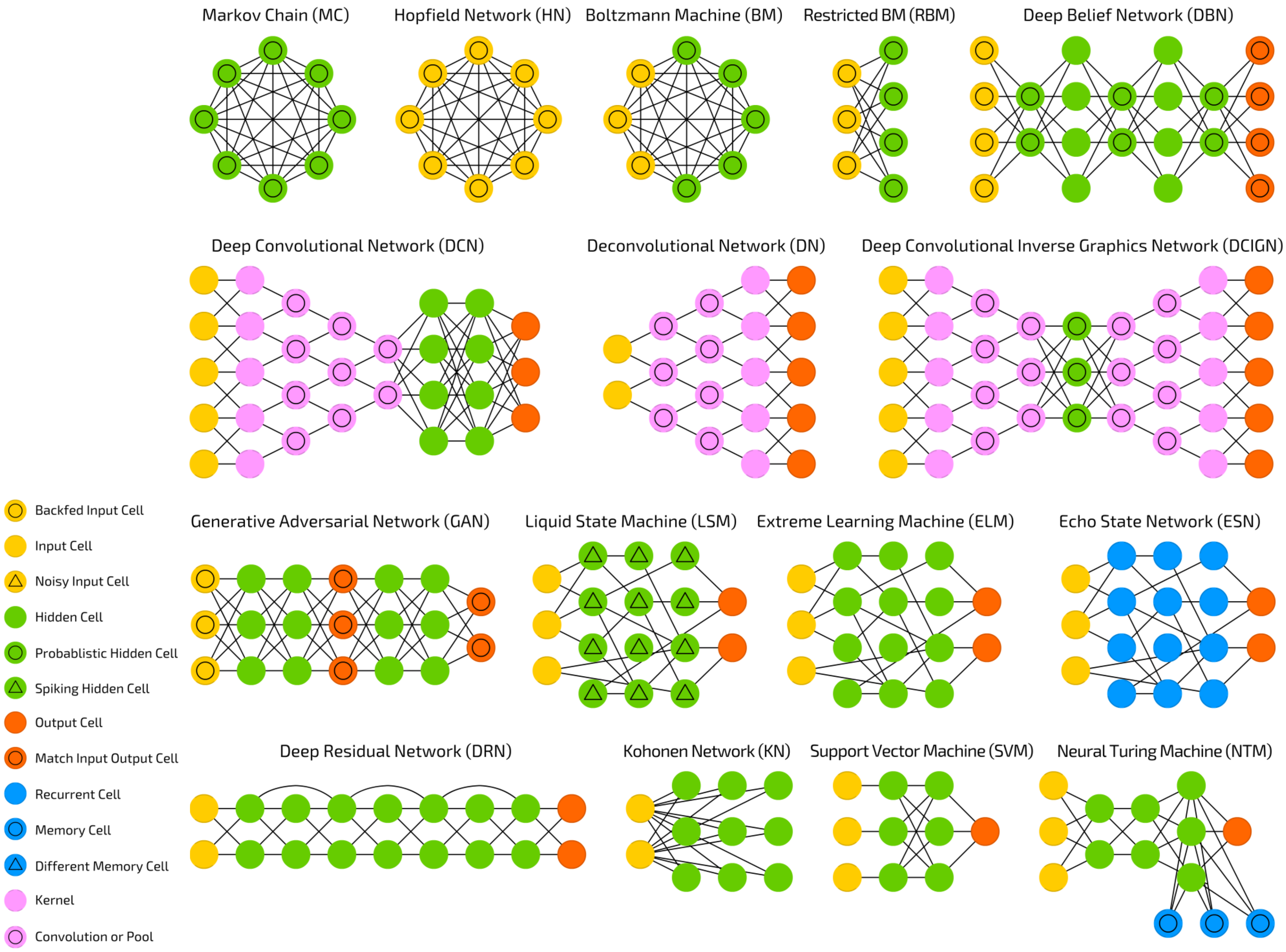
A mostly complete chart of

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

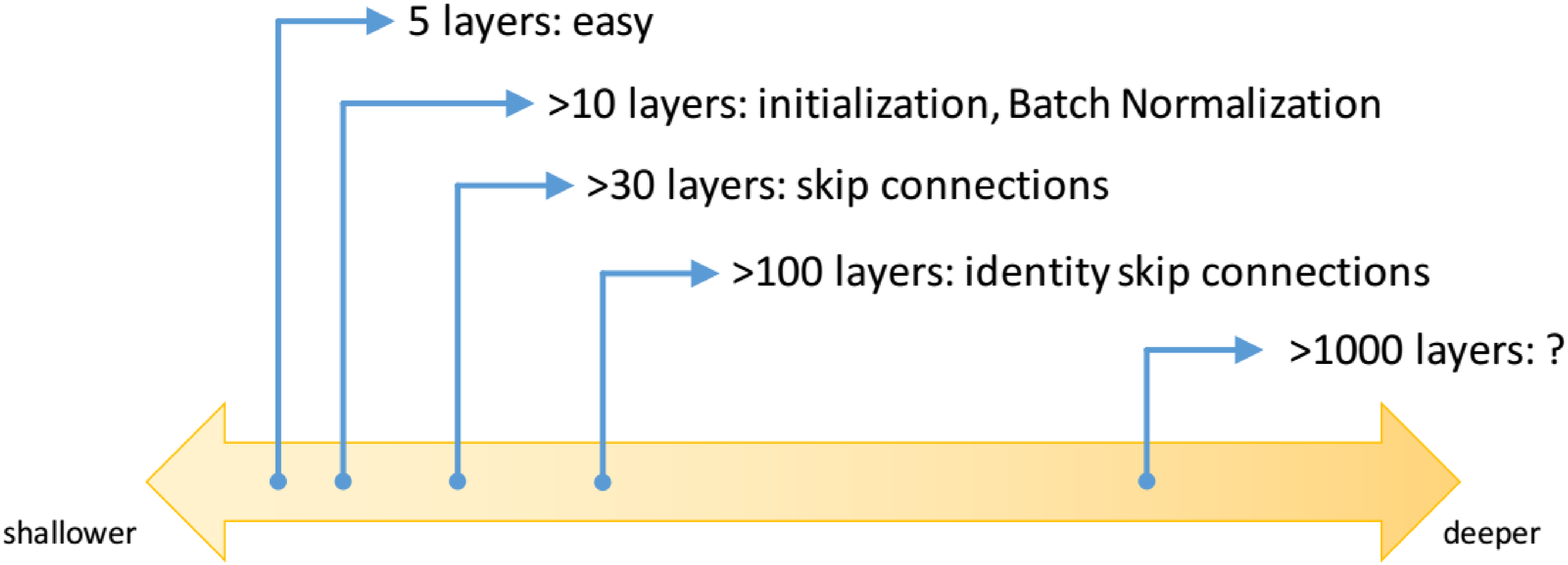
http://www.asimovinstitute.org/neural-network-zoo/




Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

NETWORKS WITH SKIP CONNECTIONS

Spectrum of Depth



5 layers: easy

>10 layers: initialization, Batch Normalization

>30 layers: skip connections

>100 layers: identity skip connections

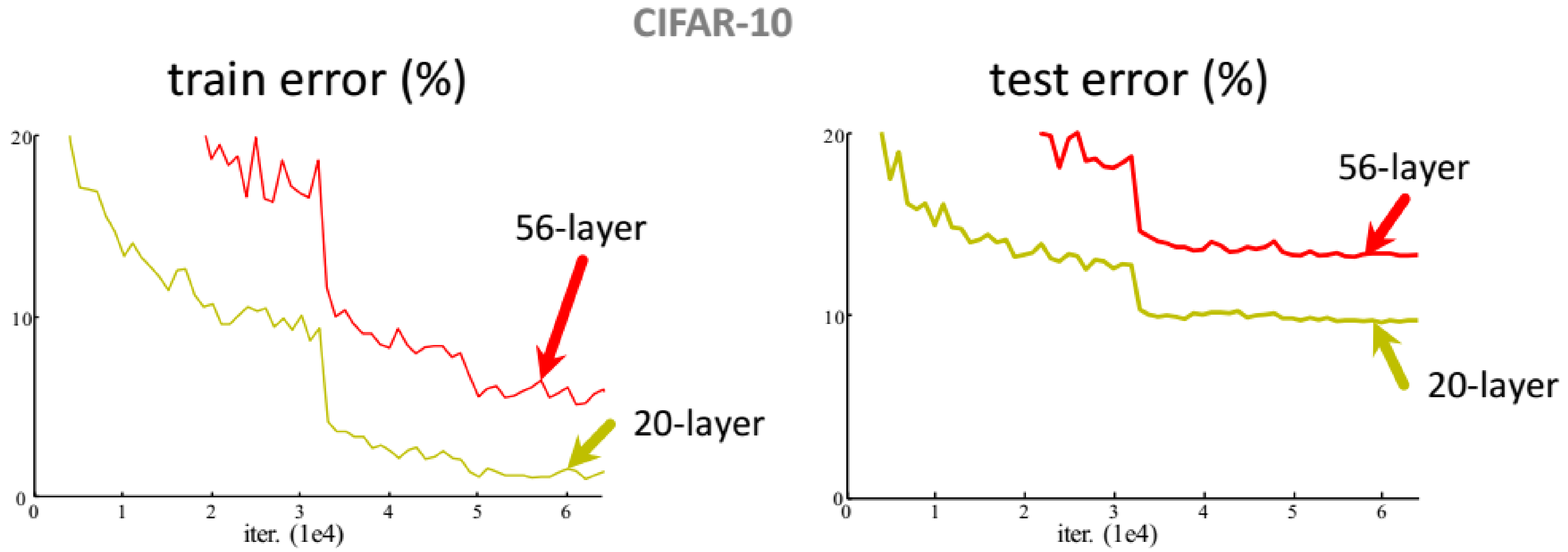
>1000 layers: ?

shallower

deeper

Increasing Depth (10-100 Layers)

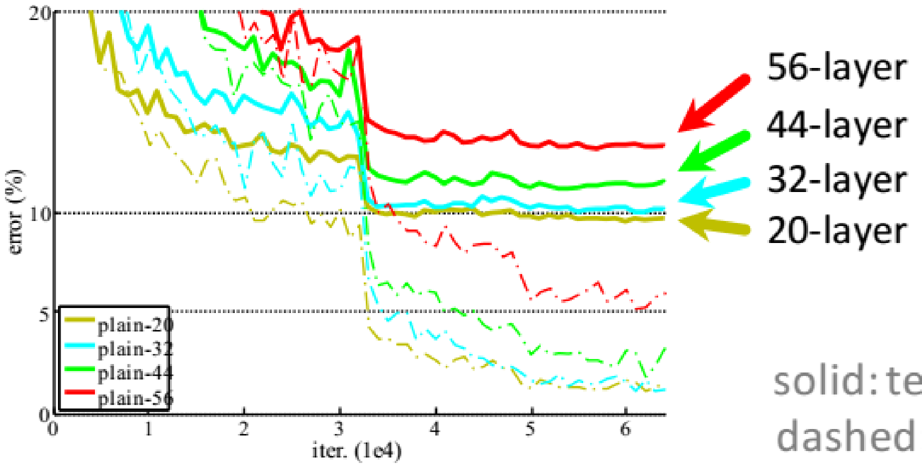
- What if we keep on stacking layers?
  - 56-layer net has **higher training error** and test error than 20-layer net



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016

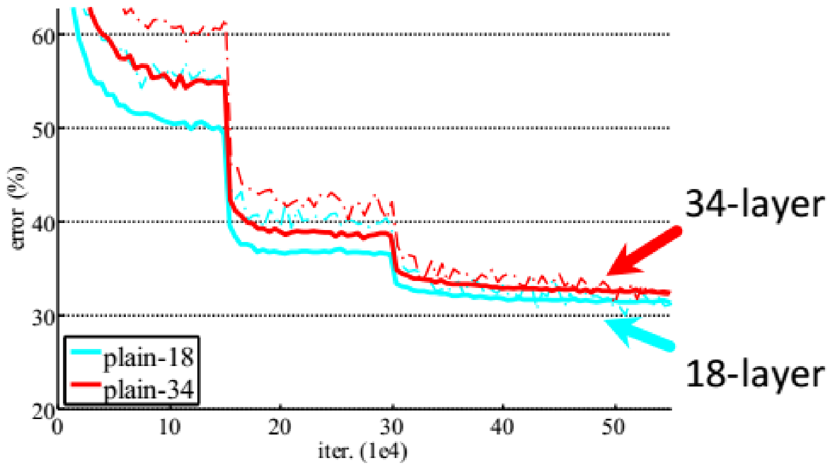
Simply Stacking Layers?

- “Overly deep” plain nets have **higher training error**
- A general phenomenon, observed in many datasets
- Reasons
  - Optimization failure



CIFAR-10

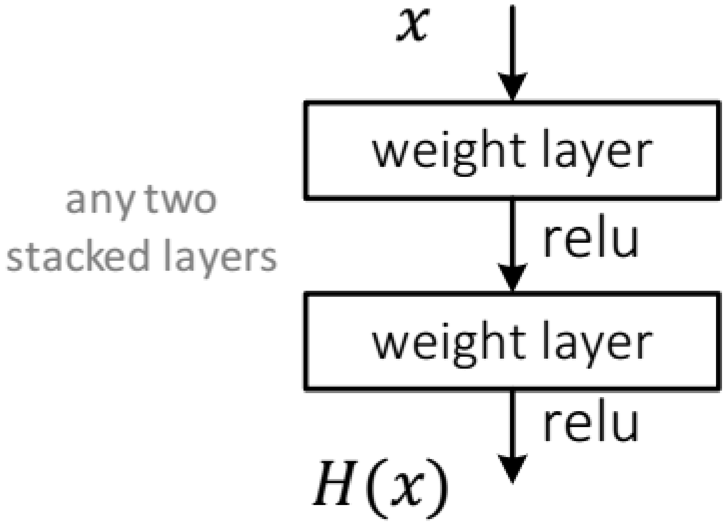
solid: test/val
dashed: train



ImageNet-1000

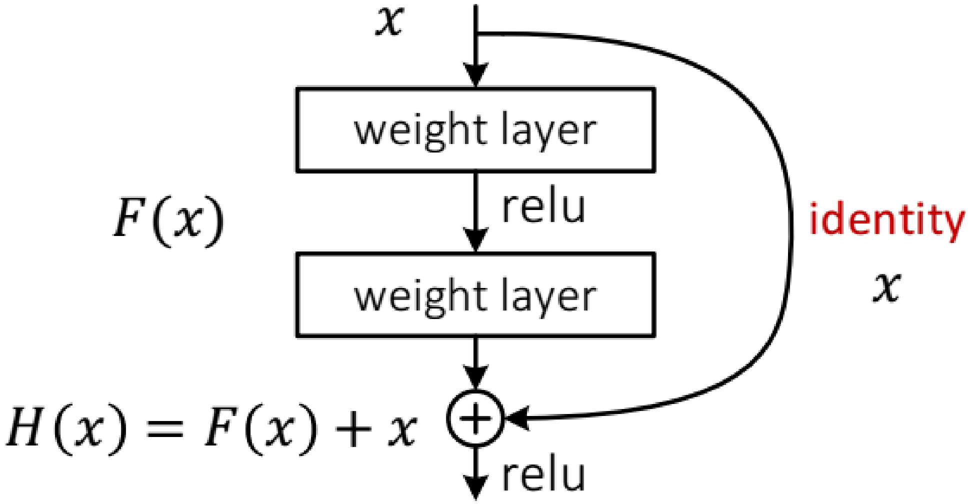
Residual Learning: skip connections

Plain Network



H(x) is any desired mapping
Hope the 2 weight layers fit *H(x)*

Residual Network



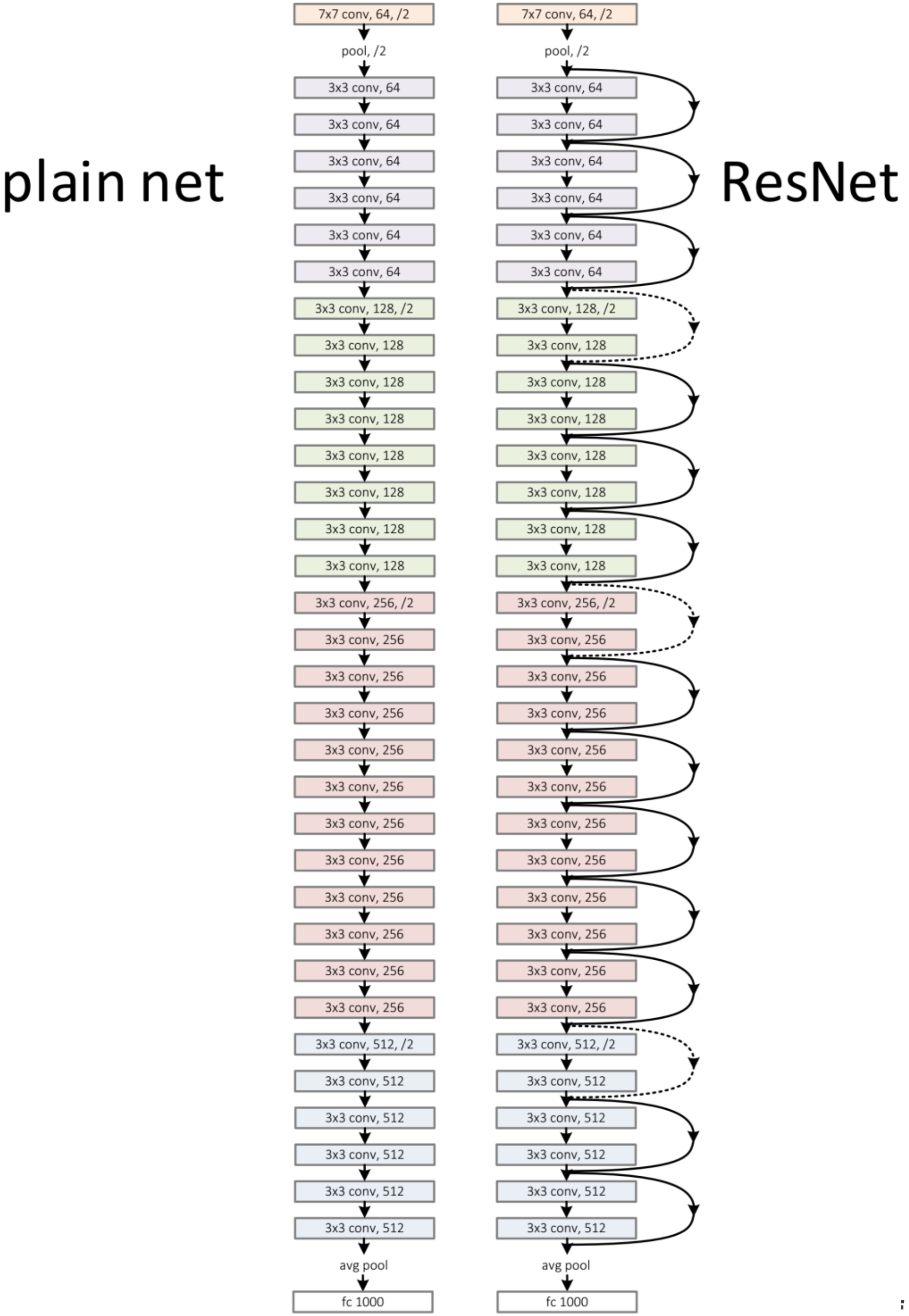
H(x) is any desired mapping
Hope the 2 weight layers fit *F(x)*

The network learns fluctuations *F(x)=H(x)-x*
Easier!

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.

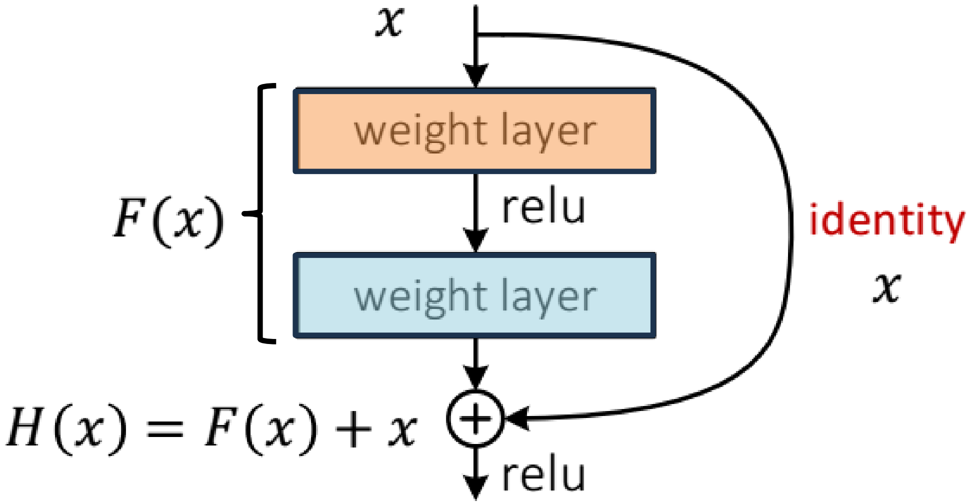
ResNet Models

- No Dropout
- With Batch Normalization
- Use Data Augmentation



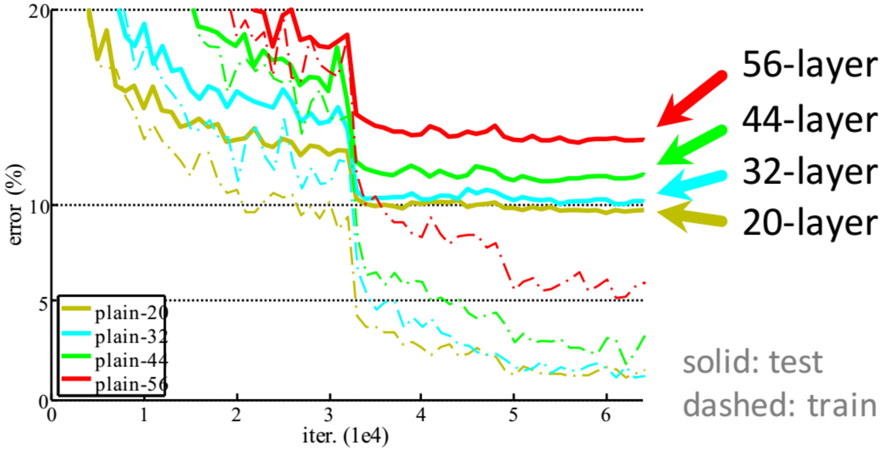
A residual/skip block in code

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        # downsample only if dimensions of x and F(x) don’t match
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

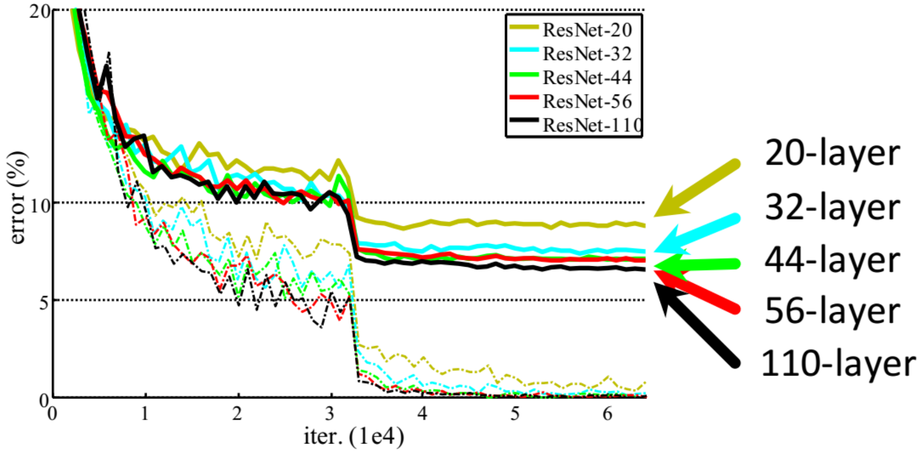


Strongly recommended: How to use a minimalistic residual network for MNIST Classification
https://github.com/foxtrotmike/CS909/blob/master/resnet_mnist.ipynb

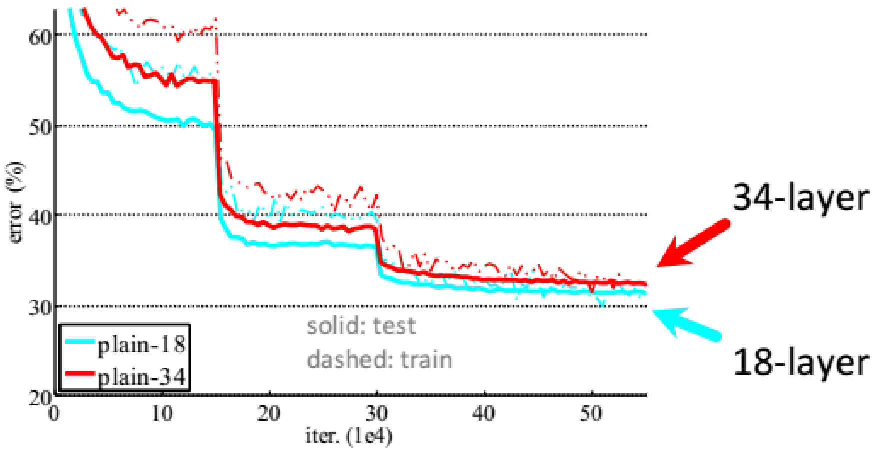
CIFAR-10 plain nets



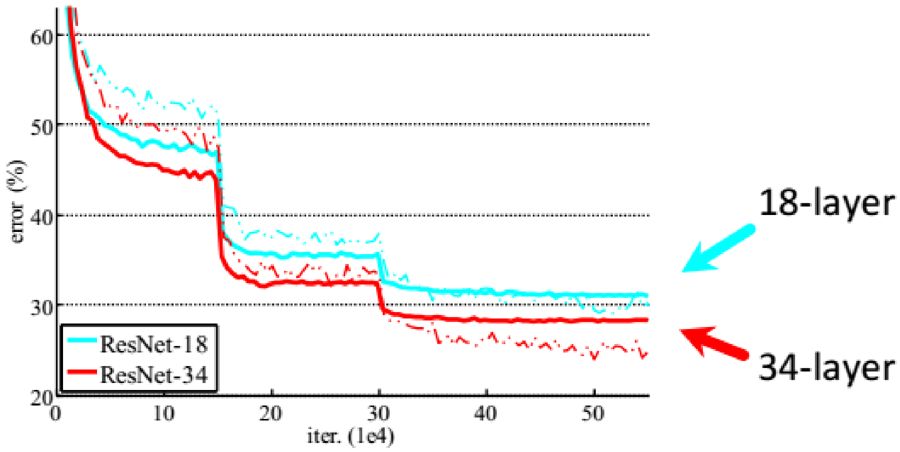
CIFAR-10 ResNets



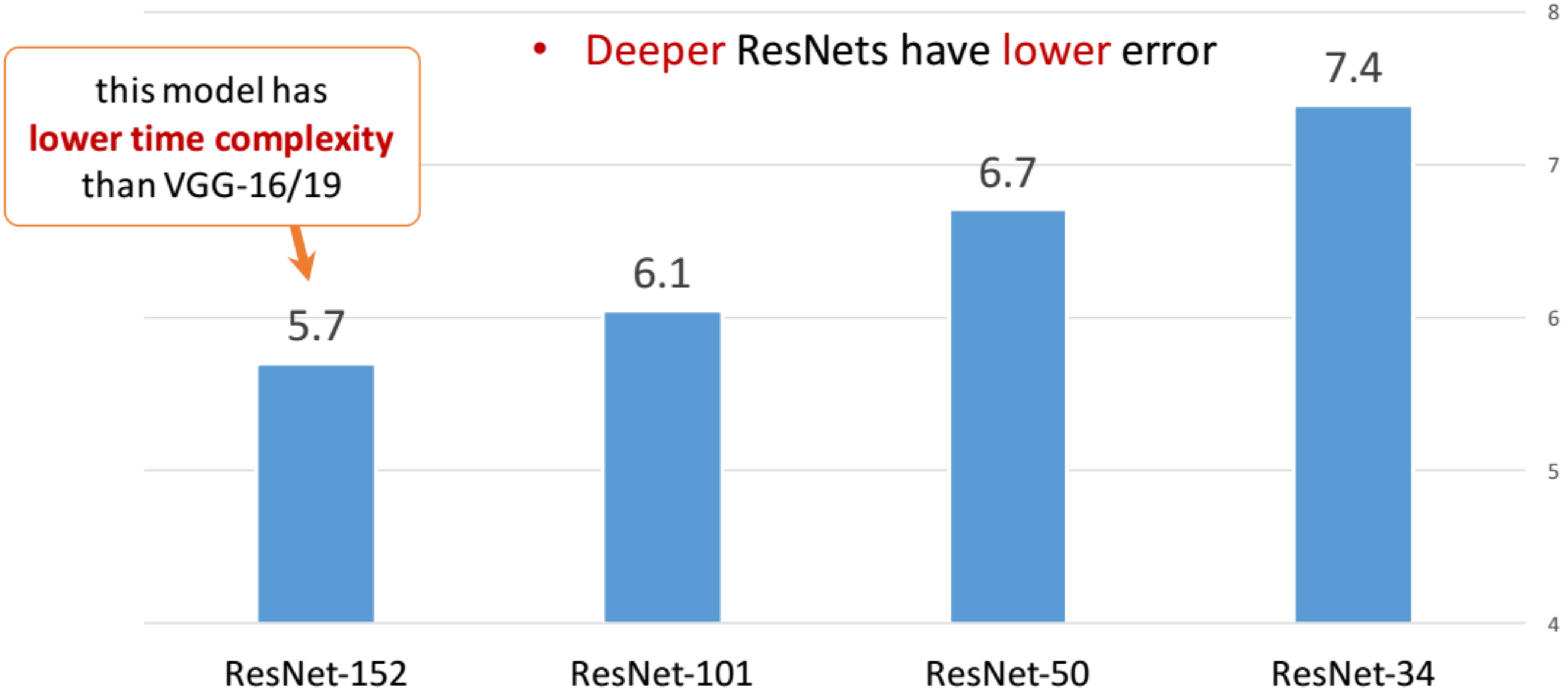
ImageNet plain nets



ImageNet ResNets



- Deep ResNets can be trained without difficulties
- Deeper ResNets have **lower training error**, and also lower test error

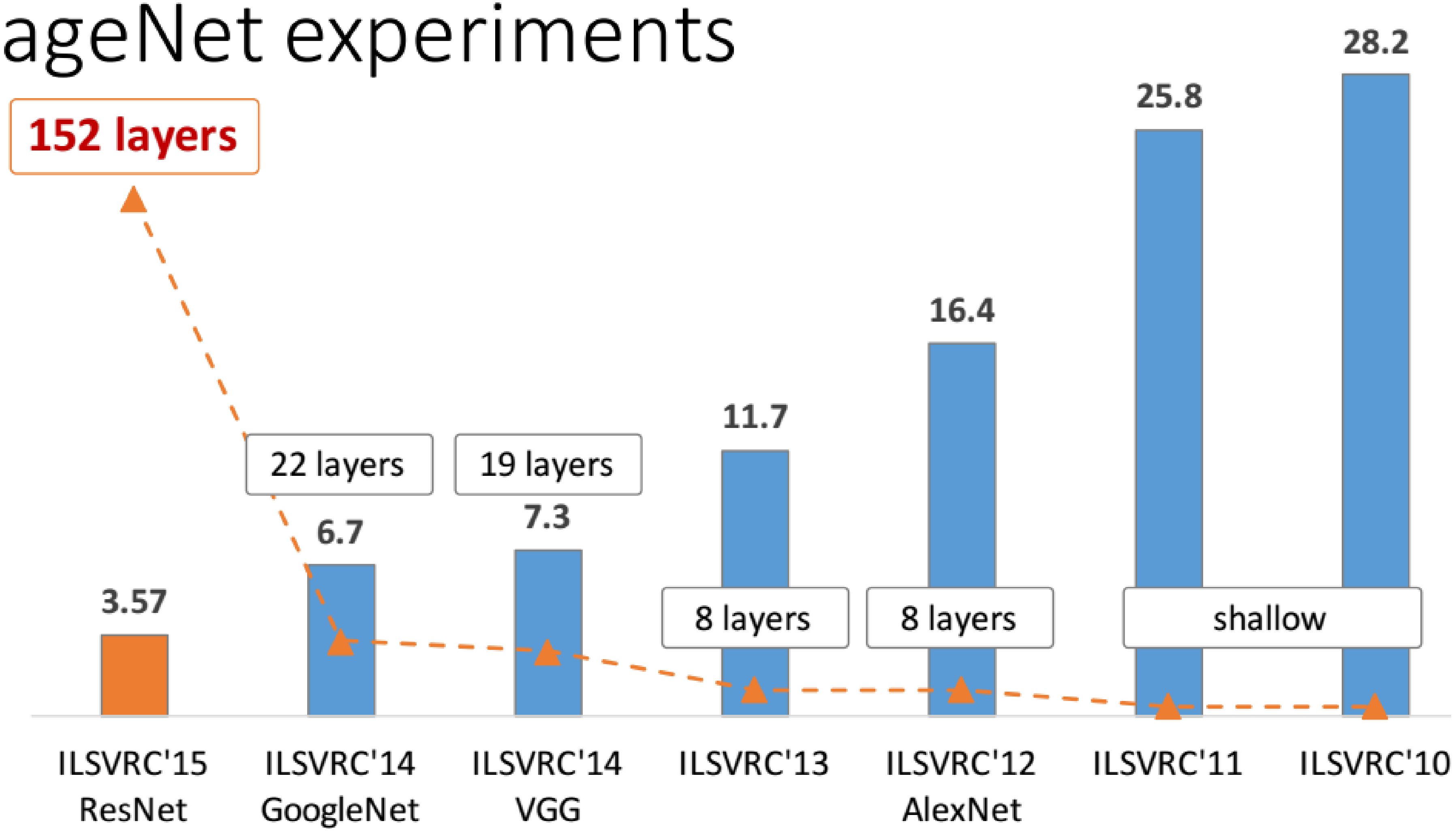


- Deeper ResNets have lower error

this model has
lower time complexity
than VGG-16/19

10-crop testing, top-5 val error (%)

ImageNet experiments

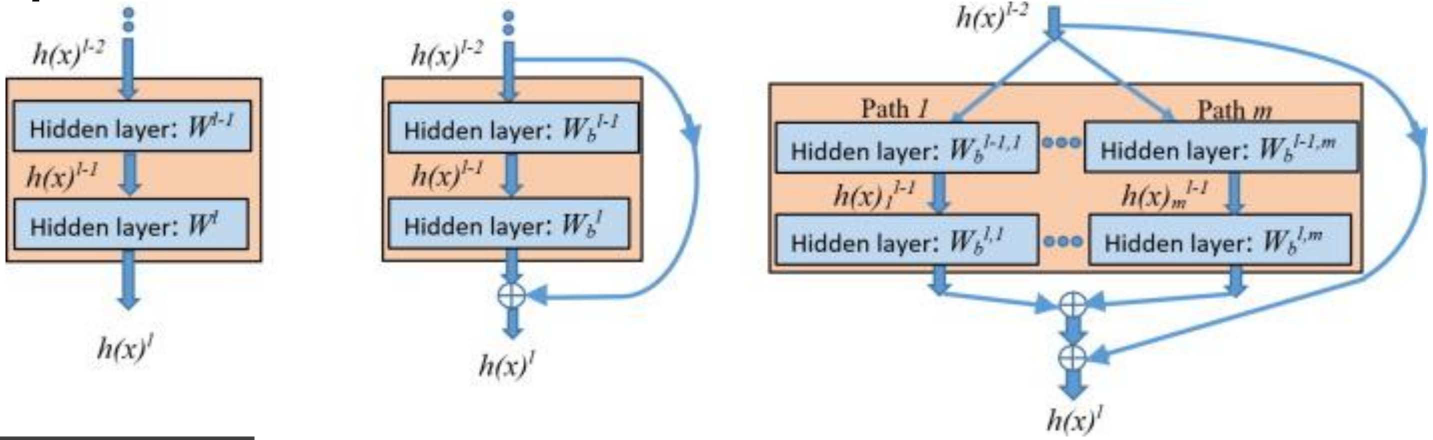


ImageNet Classification top-5 error (%)

ResNet Results

- **1st places in all five main tracks**
  - ImageNet Classification: “*Ultra-deep*” 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

- Can also concatenate outputs rather than sum
  - – ResNeXT

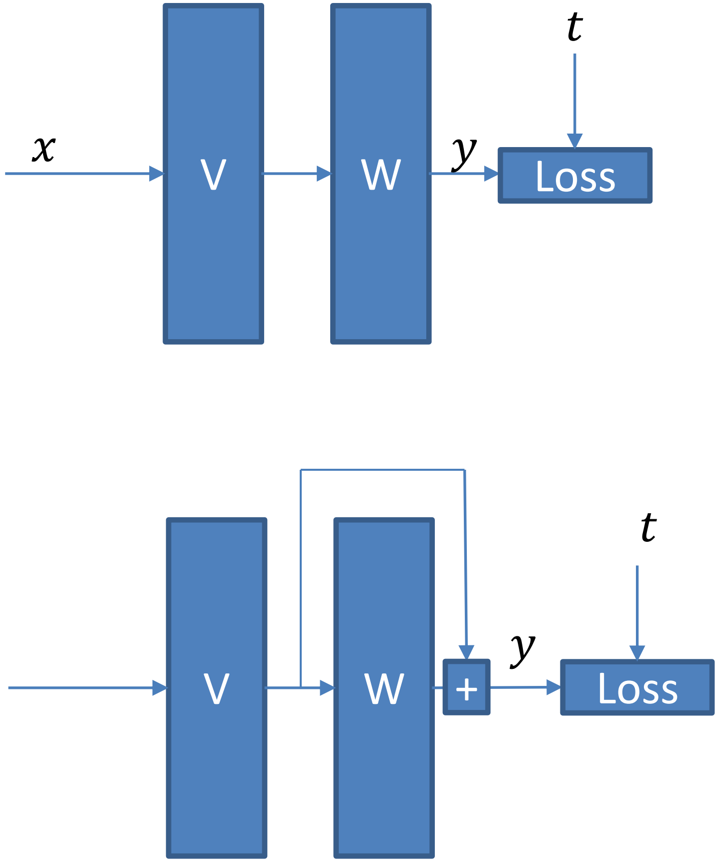


Reasons for adding skip connections

- Making gradients flow more easily

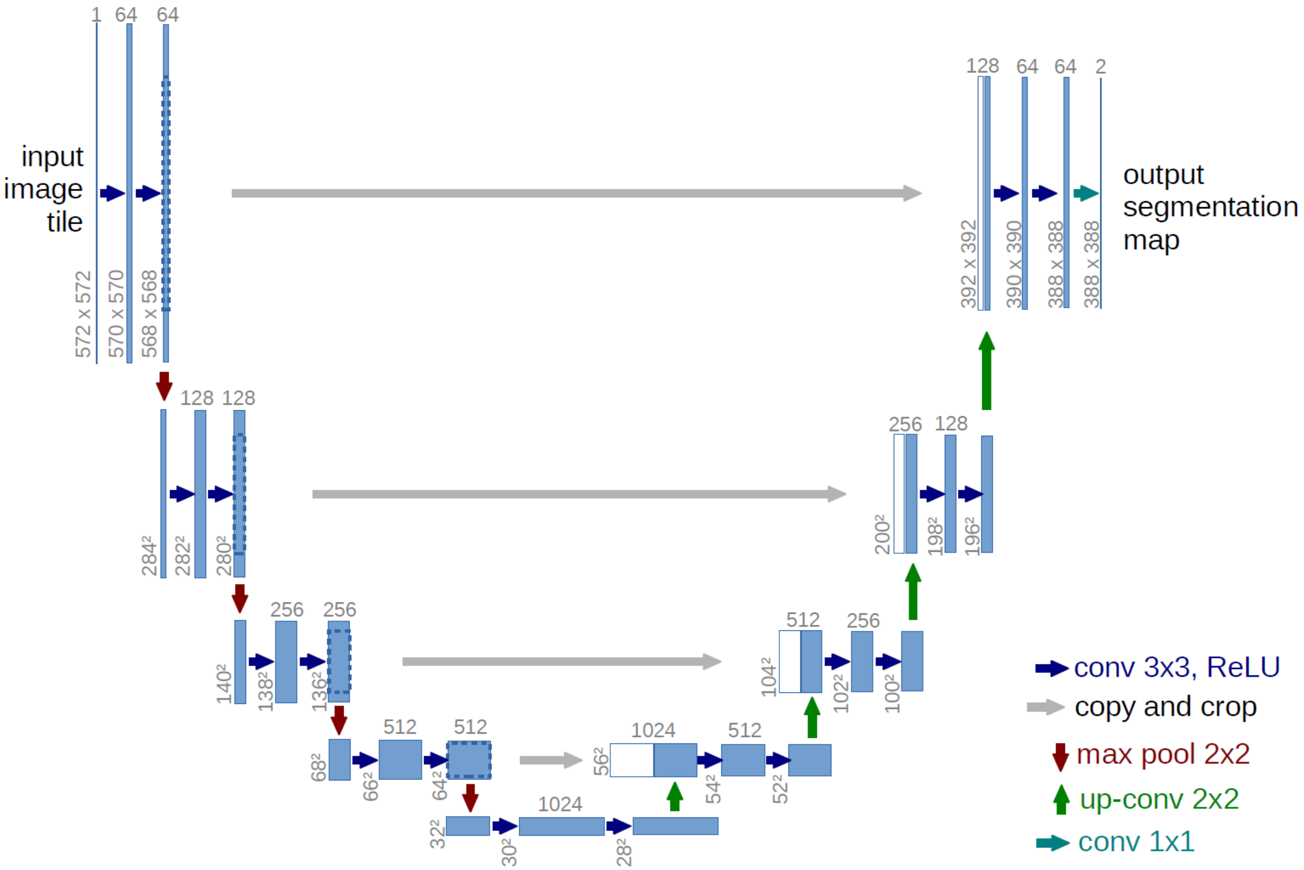
$$\Delta v_{ij} = \alpha x_i f'(\boldsymbol{v}_j^T \boldsymbol{x}) \sum_{k=1}^{m} w_{jk} \left(t_k - f\left(\sum_{j=0}^{p} w_{jk} f(\boldsymbol{v}_j^T \boldsymbol{x})\right)\right) f'\left(\sum_{j=0}^{p} w_{jk} f(\boldsymbol{v}_j^T \boldsymbol{x})\right)$$

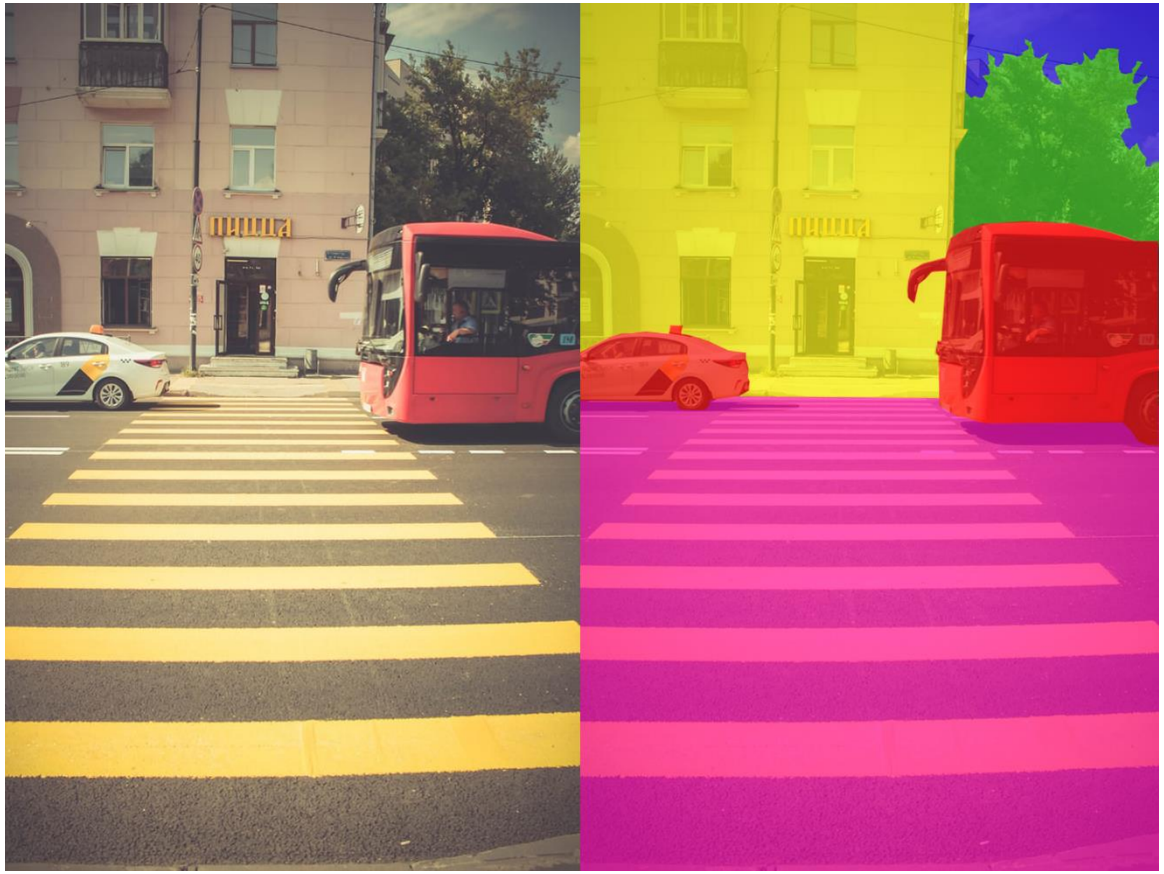
  - If you work out the weight update equation for the neural network with skip connections, it will have fewer multiplicative terms of gradients thus reducing the chances of gradient based problems
- Making information flow more easily
  - Directly Preserving information learned in earlier layers
- Have a regularization effect



Oyedotun, Oyebade K., Kassem Al Ismaeil, and Djamila Aouada. “Training Very Deep Neural Networks: Rethinking the Role of Skip Connections.” *Neurocomputing* 441 (June 21, 2021): 105–17. https://doi.org/10.1016/j.neucom.2021.02.004.

U-Net for Segmentation





YOLO

- Convolution
- Residual Architecture
- Reversible function to allow preservation information
- Programmable gradient information

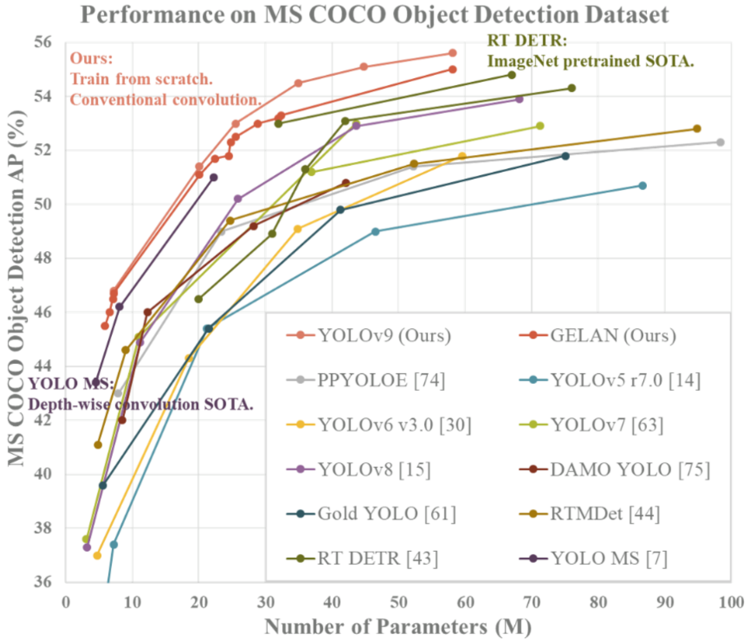


Figure 1. Comparisons of the real-time object detecors on MS COCO dataset. The GELAN and PGI-based object detection method surpassed all previous train-from-scratch methods in terms of object detection performance. In terms of accuracy, the new method outperforms RT DETR [43] pre-trained with a large dataset, and it also outperforms depth-wise convolution-based design YOLO MS [7] in terms of parameters utilization.

Wang, Chien-Yao, I.-Hau Yeh, and Hong-Yuan Mark Liao. "YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information." arXiv, February 21, 2024. https://doi.org/10.48550/arXiv.2402.13616.

Residual Networks

- Required Reading
    - Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.
- Many third-party implementations
    - list in https://github.com/KaimingHe/deep-residual-networks
    - Torch ResNet: https://github.com/pytorch/examples/tree/master/imagenet
    - Transfer Learning with ResNet: https://www.pluralsight.com/guides/introduction-to-resnet

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Identity Mappings in Deep Residual Networks”. arXiv 2016.

TRANSFORMERS

Transformers

- Very useful and popular architecture for vision tasks though originally built for natural language processing
- Use “attention mechanism” to integrate information from different components of an input in a weighted manner to produce an output representation for the input that can be passed to predictor to generate predictions

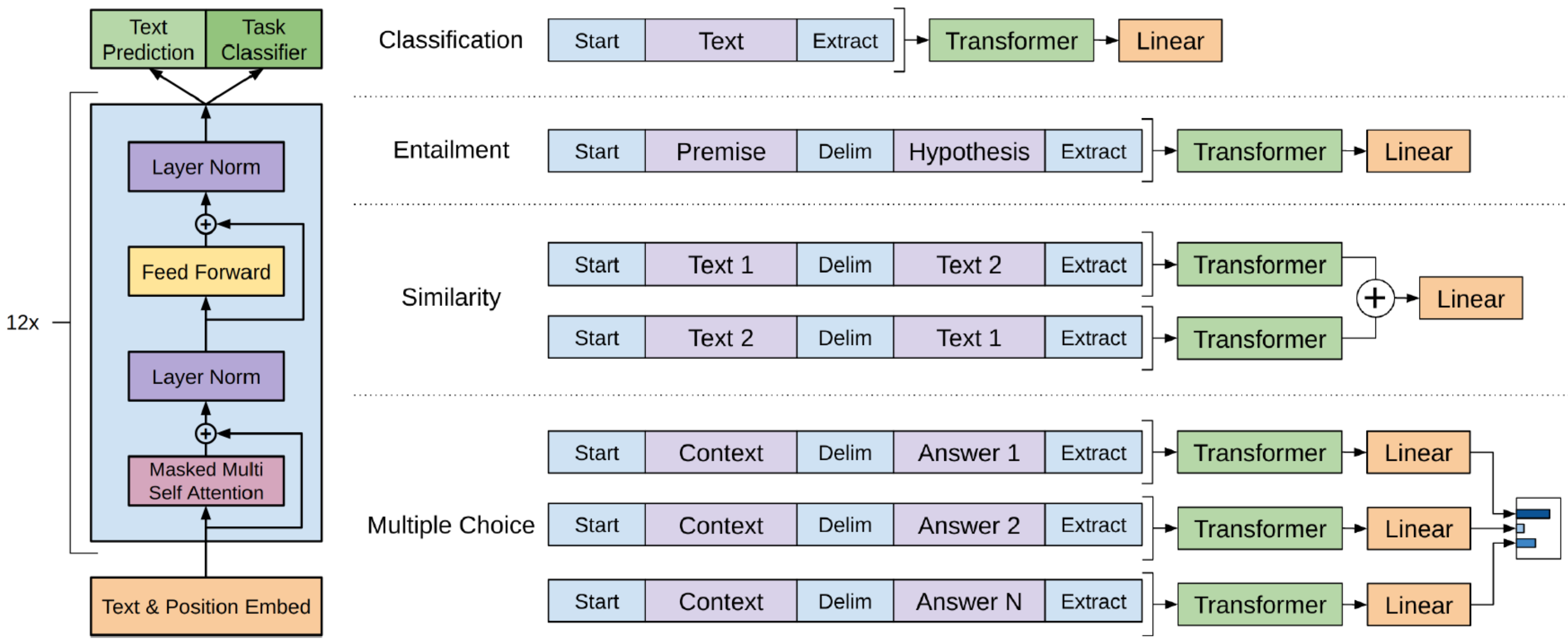
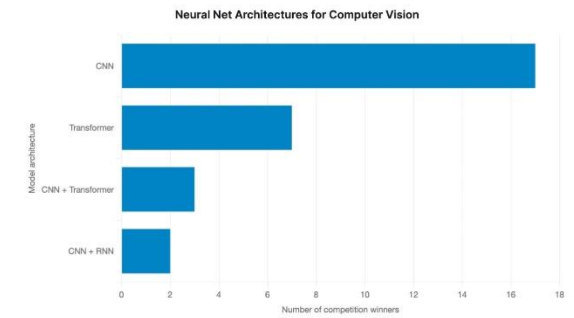
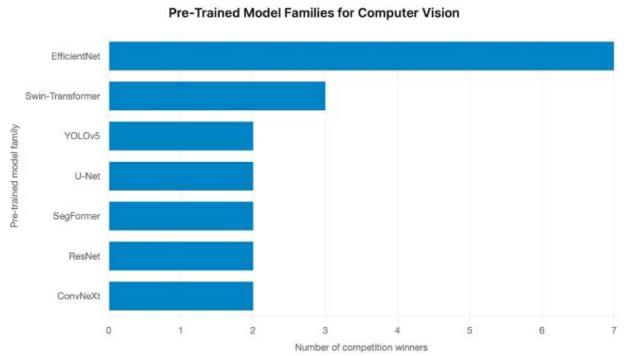


Figure from the **Generative Pre-trained Transformer (GPT)** paper

Radford, Alec, et al (OpenAI). “Improving Language Understanding by Generative Pre-Training,” 2018.

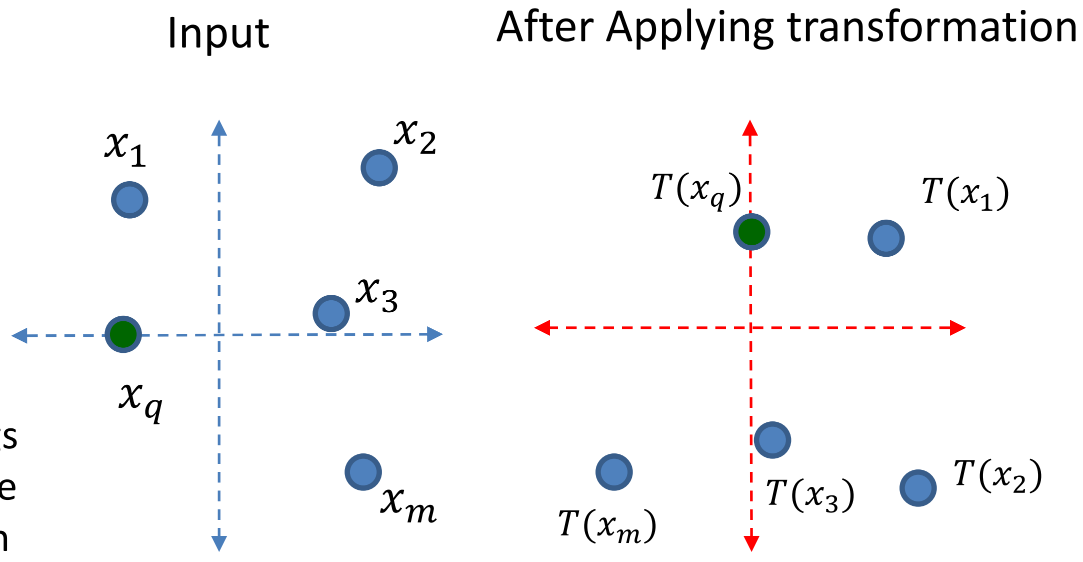

Convolutional neural networks still dominate computer vision


EfficientNet is the most popular pretrained architecture for computer vision

https://twitter.com/rasbt/status/1634564282535878661/photo/1

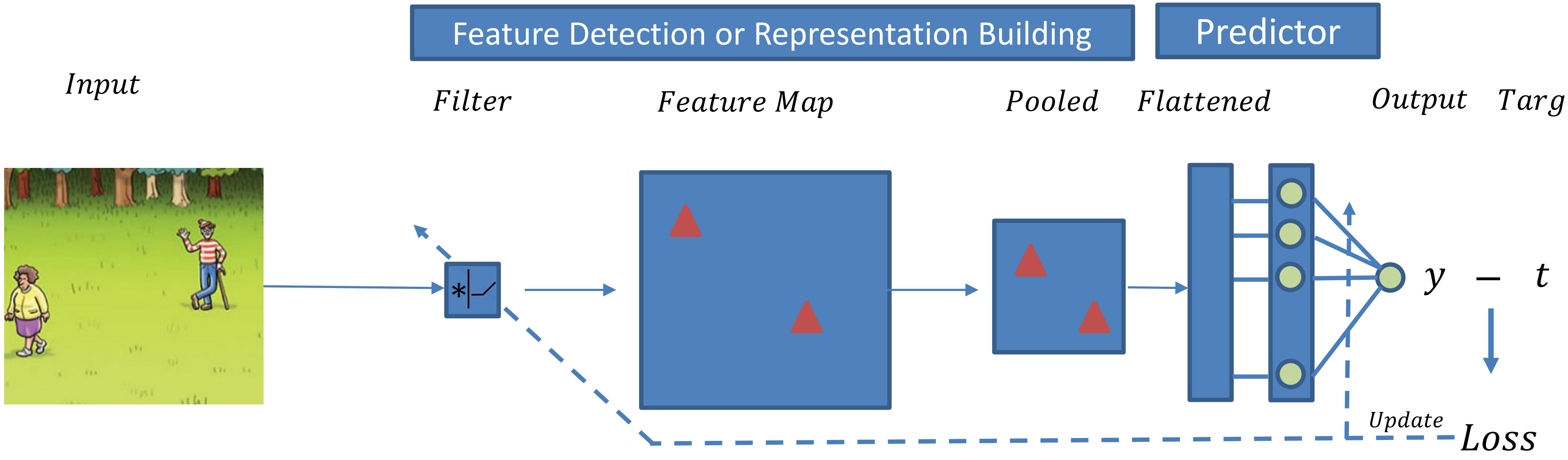
Background

- Transformations: $T(x;\theta)$
  - Explicitly transform a point to a different feature space
- A kernel $k(\boldsymbol{a},\boldsymbol{b})$ is a generalized dot-product or a way of quantifying the degree of similarity between two examples or objects
  - If we can change the definition of how similar (or distant) two things are (by switching to a different kernel), this results in a folding of the feature space which is the same effect as we would achieve from an explicit transformation of the feature space



Kernel	**Transform (for 2D Input)**
Linear: $k(\boldsymbol{a},\boldsymbol{b}) = \boldsymbol{a}^T\boldsymbol{b}$	$\phi(\boldsymbol{u}) = \boldsymbol{u} = [u^{(1)} \quad u^{(2)}]^T$
Polynomial degree 2: $k(\boldsymbol{a},\boldsymbol{b}) = (\boldsymbol{a}^T\boldsymbol{b})^2$ (Homogeneous)	$\phi(\boldsymbol{u}) = [u^{(1)^2} \quad u^{(2)^2} \quad \sqrt{2}u^{(1)}u^{(2)}]^T$
Polynomial degree 2: $k(\boldsymbol{a},\boldsymbol{b}) = (\boldsymbol{a}^T\boldsymbol{b}+1)^2$	$\phi(\boldsymbol{u}) = [1 \quad \sqrt{2}u^{(1)} \quad \sqrt{2}u^{(2)} \quad u^{(1)^2} \quad u^{(2)^2} \quad \sqrt{2}u^{(1)}u^{(2)}]^T$
RBF Kernel: $k(\boldsymbol{a},\boldsymbol{b}) = \exp(-\gamma\|\boldsymbol{a}-\boldsymbol{b}\|^2)$	Infinite dimensional (depending upon hyperpameter $\gamma > 0$ See: https://en.wikipedia.org/wiki/Radial_basis_function_kernel

For Review see notes on Kernels in SVMs



Feature Detection or Representation Building

Predictor

Input

Filter

Feature Map

Pooled

Flattened

Output

Target

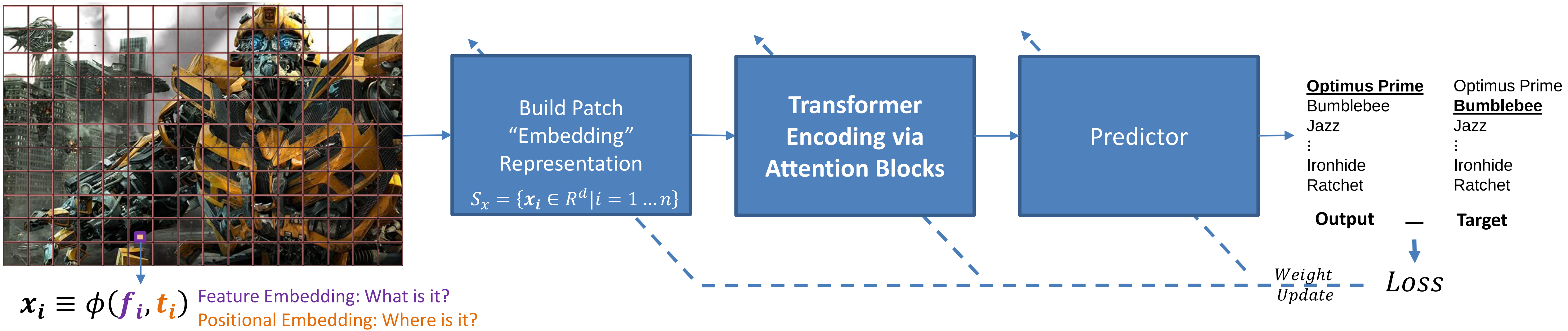
$y - t$

Update

Loss

Convolutional Neural Network

(Vision) Transformers



Build Patch "Embedding" Representation

$S_x = \{\boldsymbol{x_i} \in R^d | i = 1 \dots n\}$

Transformer Encoding via Attention Blocks

Predictor

Output	Target
Optimus Prime	Optimus Prime
Bumblebee	Bumblebee
Jazz	Jazz
⋮	⋮
Ironhide	Ironhide
Ratchet	Ratchet

Output — **Target**

Loss

Weight Update

$\boldsymbol{x_i} \equiv \phi(\boldsymbol{f_i}, \boldsymbol{t_i})$

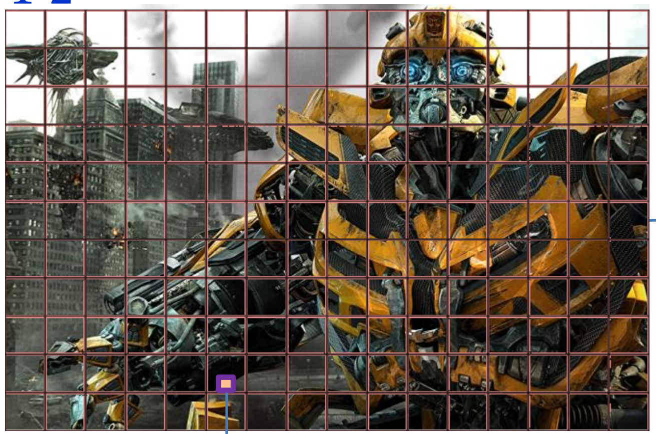
Feature Embedding: What is it?

Positional Embedding: Where is it?

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale." arXiv, June 3, 2021. https://doi.org/10.48550/arXiv.2010.11929.

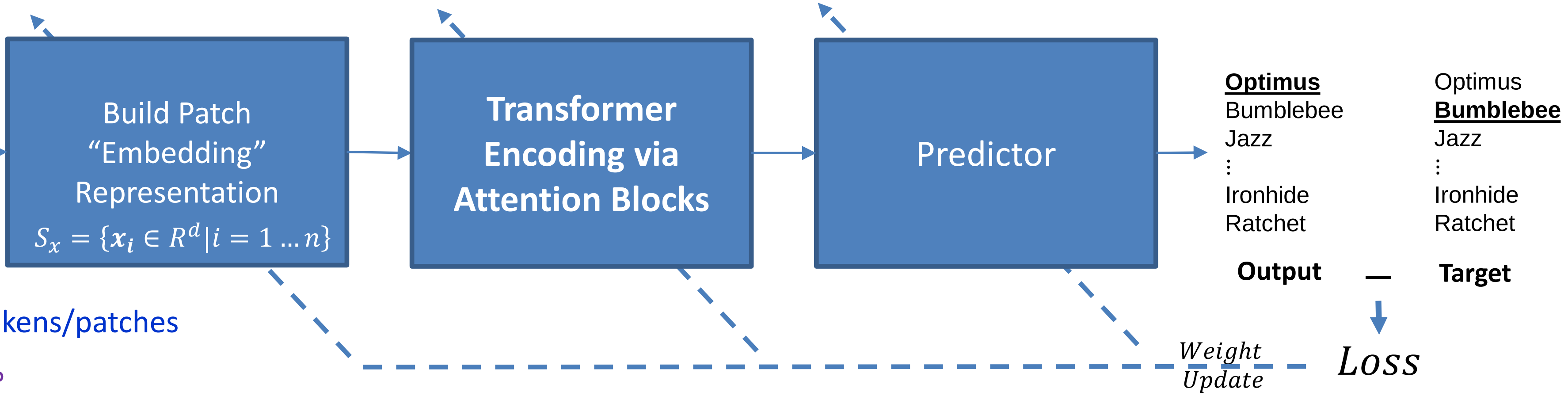
(Vision) Transformers (for classification)

$1\ 2\ \cdots$



n tokens/patches

$\boldsymbol{x_i} \equiv \phi(\boldsymbol{f}_i, \boldsymbol{t}_i)$ Feature Embedding: What is it?
Positional Embedding: Where is it?



Build Patch “Embedding” Representation $S_x = \{\boldsymbol{x_i} \in R^d | i = 1 \dots n\}$

Transformer Encoding via Attention Blocks

Predictor

Output		**Target**
Optimus Bumblebee Jazz ⋮ Ironhide Ratchet	—	Optimus **Bumblebee** Jazz ⋮ Ironhide Ratchet

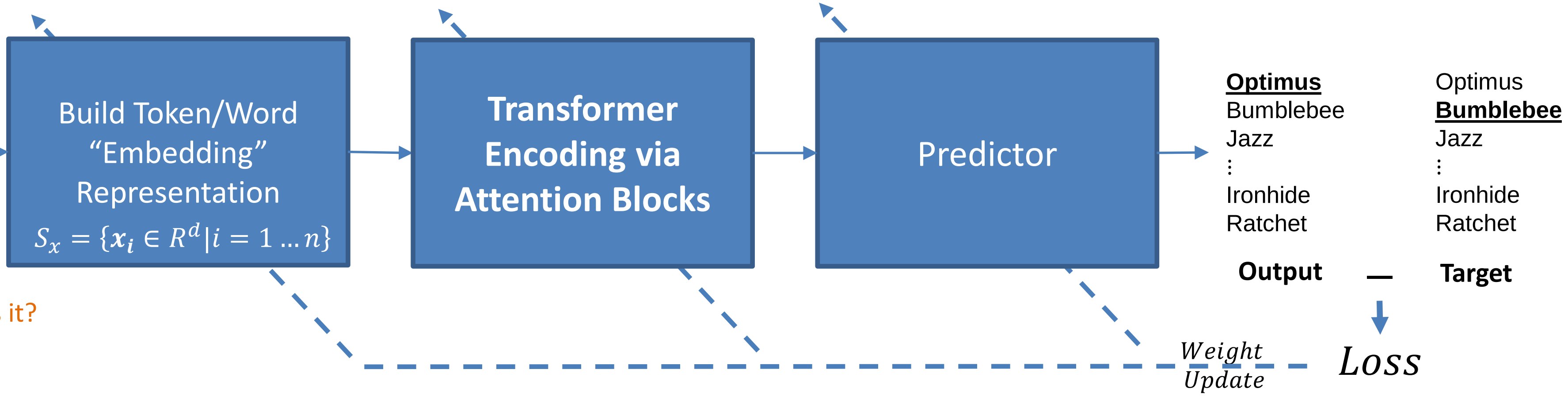
$Loss$

Weight Update

Building an integrated representation of how components form the overall object

A transformer that can transform into a yellow car is called _____________.

$\boldsymbol{x_i} \equiv \phi(\boldsymbol{f}_i, \boldsymbol{t}_i)$ Feature Embedding: What is it?
Positional Embedding: Where is it?



Build Token/Word “Embedding” Representation $S_x = \{\boldsymbol{x_i} \in R^d | i = 1 \dots n\}$

Transformer Encoding via Attention Blocks

Predictor

Output		**Target**
Optimus Bumblebee Jazz ⋮ Ironhide Ratchet	—	Optimus **Bumblebee** Jazz ⋮ Ironhide Ratchet

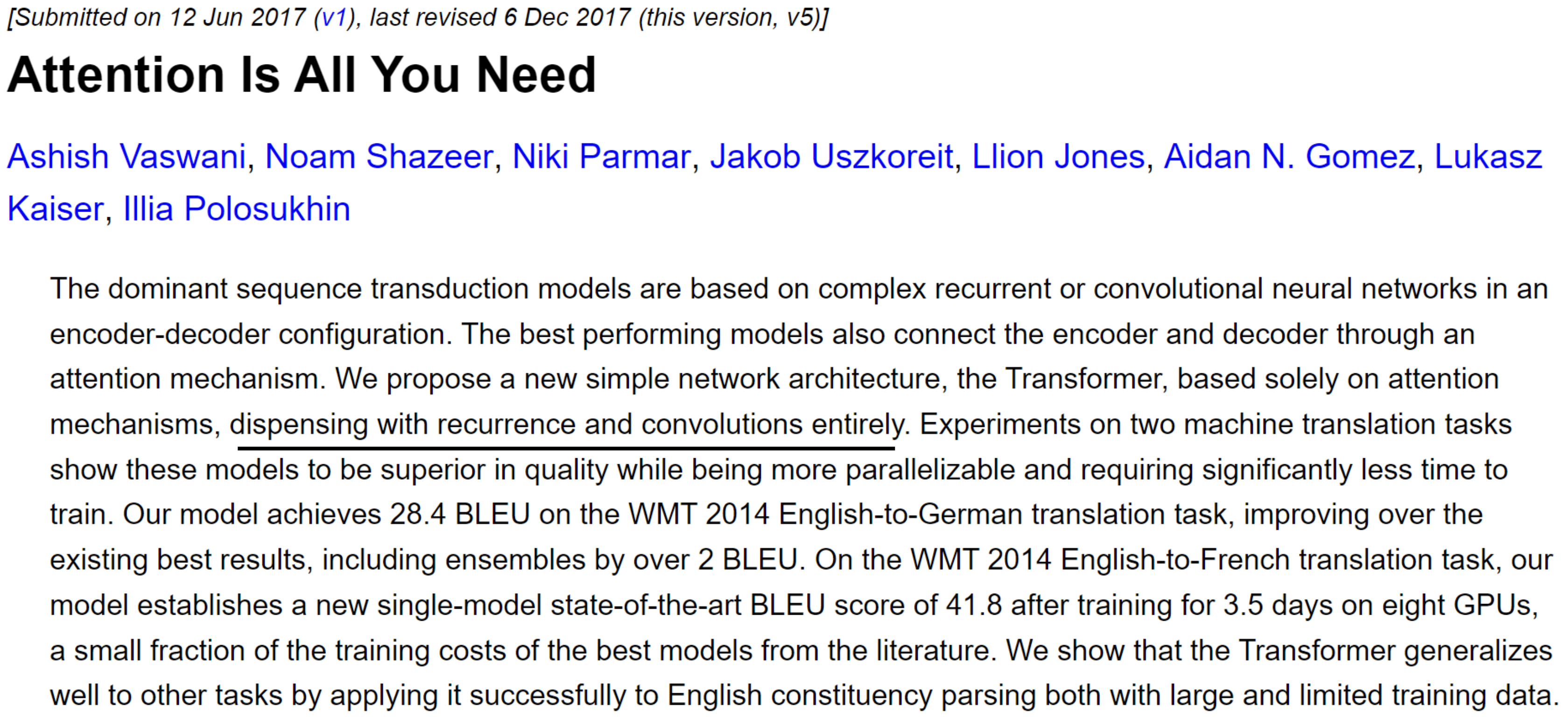
$Loss$

Weight Update

(NLP) Transformers (for next word prediction)

Simplest: $\phi(\boldsymbol{f}_i, \boldsymbol{t}_i) = \boldsymbol{f}_i + \boldsymbol{t}_i$

What is attention and why do you need it?



[Submitted on 12 Jun 2017 (v1), last revised 6 Dec 2017 (this version, v5)]

Attention Is All You Need

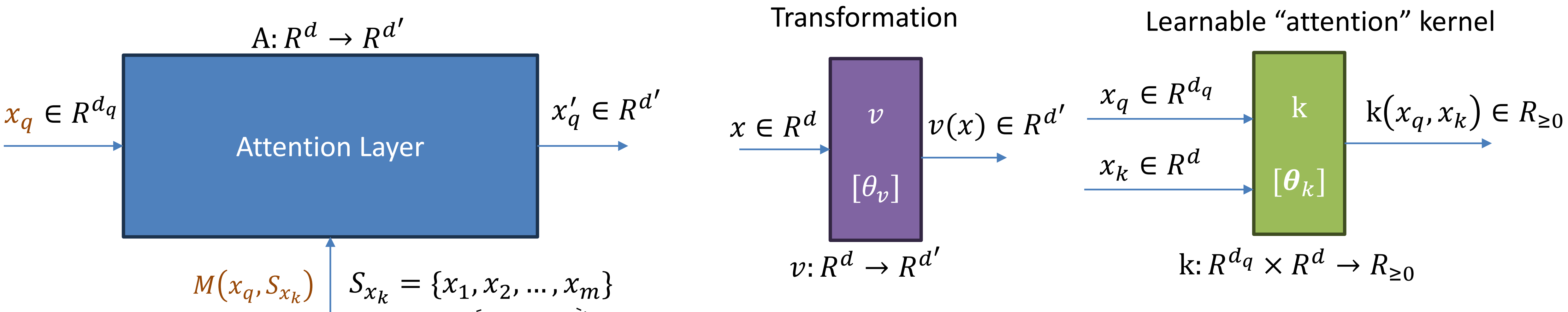
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder-decoder configuration. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

We are going to have a no **gobbledygook** introduction to attention (using the paper below)!

Tsai, Yao-Hung Hubert, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. “**Transformer Dissection: A Unified Understanding of Transformer’s Attention via the Lens of Kernel**.” *ArXiv:1908.11775 [Cs, Stat]*, November 11, 2019. http://arxiv.org/abs/1908.11775.

General Attention Building Blocks



Let simplify and understand transformers

(Masked out)

How similar are x_q and x_k relative to similarity of x_q to all tokens

$$x'_q = A(x_q; M(x_q, S_{x_k}); \boldsymbol{\theta}) = \sum_{x_k \in M(x_q, S_{x_k})} a(x_q, x_k; M(x_q, S_{x_k}), \boldsymbol{\theta}_k)\, v(x_k; \boldsymbol{\theta}_v) = \sum_{x_k \in M(x_q, S_{x_k})} \frac{k(x_q, x_k; \boldsymbol{\theta}_k)}{\sum_{x_{k'} \in M(x_q, S_{x_k})} k(x_q, x_{k'}; \boldsymbol{\theta}_k)} v(x_k; \boldsymbol{\theta}_v)$$

Input:

A "query" token $x_q \in R^{d_q}$ representation of a component (patch or token) which will be transformed. In turn, all tokens will take the role of the query token in classical attention.

A set of "key" tokens $S_{x_k} = \{x_1, x_2, \ldots, x_m\}$. Can come from a different source (e.g., as in cross-attention).

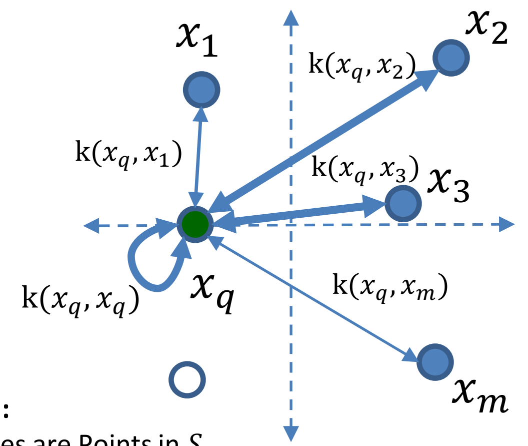
Output:

$x'_q \in R^{d'}$ Transformed representation of x_q which is based on the transformed representations of other tokens and the degree of association of x_q to those other tokens

Attention Parameters

- A "value" or transformation function $v(x): R^d \to R^{d'}$ that produces a vector for a given token (For simplicity, assume, $d = d'$)
- A "Masking" function $M(x_q, S_{x_k})$ which gives a subset of tokens from S_{x_k} to which a given query can be compared, e.g., text upto a certain point. For simplicity, assume, for all x_q, $M(x_q, S_{x_k}) = S_{x_k}$)
- A "kernel" function $k(x_i, x_j)$ that gives us the association between two tokens. Used to determine the **attention scores** that tell us how associated are x_q and x_k relative to similarity of x_q to all tokens
- Different formulations for $k(x_i, x_j)$, $M(x_q, S_{x_k})$ and $v(x)$ give you different flavours of attentions. Learnable parameters denoted by $\boldsymbol{\theta}$.

Input and defining attention scores



Legend:

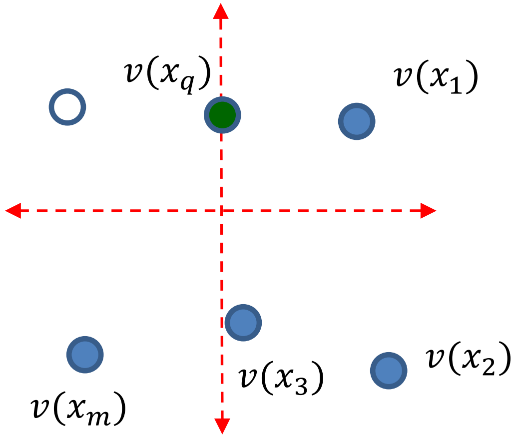
All circles are Points in S_{x_k}

Filled circles are in $M(x_q, S_{x_k})$ and will be used in the layer

Note that points in $M(x_q, S_{x_k})$ will change depending upon x_q

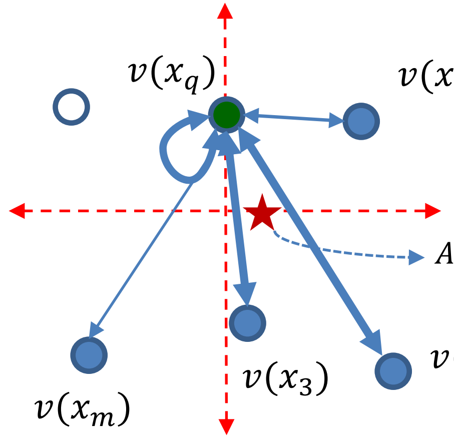
Thickness of solid lines indicates attention scores $a_{qk} \in [0,1]$ which is a obtained by dividing $k(x_q, x_k)$ by the sum of all kernel values involving x_q.

After transformation/value function v(x)



Note change in space (blue to red dotted arrows and shifting of the points)

Output of Attention



attention values

$$A\left(x_q; M\left(x_q, S_{x_k}\right)\right) = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} a_{qk} v(x_k)$$

$$a_{qk} = \frac{k(x_q, x_k)}{\sum_{x_{k'} \in M\left(x_q, S_{x_k}\right)} k(x_q, x_{k'})}$$

The new representation of the token (indicated by star) is based on the "pulls" (attention values a_{qk}) of different points on the query token or the weighted combination of all transformed points.

This process can be applied for all tokens in the input one by one so if there are n tokens in the input, there would be n tokens in the output (with transformed representation).

- Input:
  - A "query" token $x_q \in R^d$ representation of a component (patch or token)
  - A set of "key" tokens S_{x_k}
- Attention Parameters
  - A value function $v(x): R^d \rightarrow R^{d'}$ that produces a vector for a given token (For simplicity, assume, $d = d' = d_q$)
  - A "Masking" function $M(x_q, S_{x_k})$ which gives a subset of tokens from S_{x_k} to which a given query can be compared (For simplicity, assume, for all x_q, $M(x_q, S_{x_k}) = S_{x_k}$)
  - A kernel function $k(x_i, x_j)$ that can give us a degree of similarity between two tokens
  - Different formulations for $k(x_i, x_j)$, $M(x_q, S_{x_k})$ and $v(x)$ give you different flavours of attentions but once chosen they remain the same for a given attention block

- Output:
  - A new representation for the query token (patch)

$$x_q' = A\left(x_q; M\left(x_q, S_{x_k}\right); \boldsymbol{\theta}\right)$$

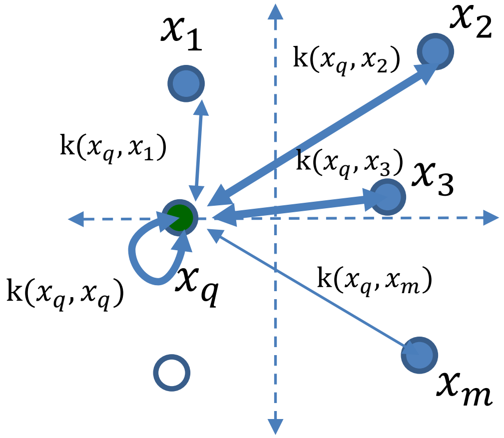
$$= \sum_{x_k \in M\left(x_q, S_{x_k}\right)} a\left(x_q, x_k; \boldsymbol{\theta_a}\right) v\left(x_k; \boldsymbol{\theta_v}\right)$$

$$= \sum_{x_k \in M\left(x_q, S_{x_k}\right)} \frac{k\left(x_q, x_k; \boldsymbol{\theta_k}\right)}{\sum_{x_{k'} \in M\left(x_q, S_{x_k}\right)} k\left(x_q, x_{k'}; \boldsymbol{\theta_k}\right)} v\left(x_k; \boldsymbol{\theta_v}\right)$$

Non-learnable attention

$$x'_q = A\left(x_q; M\left(x_q, S_{x_k}\right); \boldsymbol{\theta}\right) = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} \mathrm{a}\left(x_q, x_k; \boldsymbol{\theta_a}\right) v\left(x_k; \boldsymbol{\theta_v}\right) = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} \frac{\mathrm{k}\left(x_q, x_k; \boldsymbol{\theta_k}\right)}{\sum_{x_{k'} \in M\left(x_q, S_{x_k}\right)} \mathrm{k}\left(x_q, x_{k'}; \boldsymbol{\theta_k}\right)} v\left(x_k; \boldsymbol{\theta_v}\right)$$

- Note that if we pick a fixed $\mathrm{k}\left(x_q, x_k\right)$ and $v\left(x_k\right)$, such as:
  - $\mathrm{k}\left(x_q, x_k; \boldsymbol{\theta_k}\right) = \boldsymbol{exp}\left(-\boldsymbol{\theta}^2 \left\|\boldsymbol{x_q} - \boldsymbol{x_k}\right\|^2\right)$
  - $v\left(\boldsymbol{x_k}; \boldsymbol{\theta_v}\right) = \boldsymbol{x_k}$
- This leads to the following expression which expresses x_q in terms of other points in $M\left(x_q, S_{x_k}\right)$. This is similar, in concept, to locally linear embeddings.

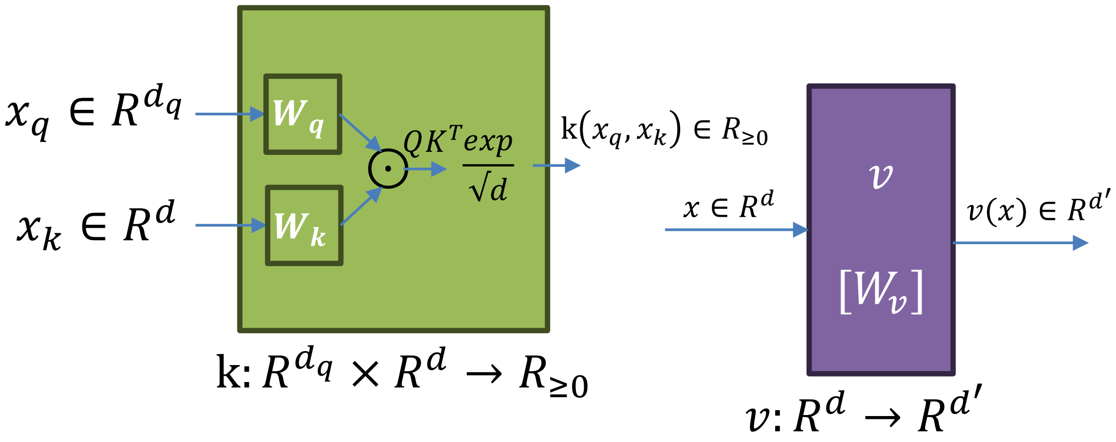


$$x'_q = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} \frac{\boldsymbol{exp}\left(-\boldsymbol{\theta}^2 \left\|\boldsymbol{x_q} - \boldsymbol{x_k}\right\|^2\right)}{\sum_{x_{k'} \in M\left(x_q, S_{x_k}\right)} \boldsymbol{exp}\left(-\boldsymbol{\theta}^2 \left\|\boldsymbol{x_q} - \boldsymbol{x_{k'}}\right\|^2\right)} \boldsymbol{x_k} = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} softmax\left(-\boldsymbol{\theta}^2 \left\|\boldsymbol{x_q} - \boldsymbol{x_k}\right\|^2\right) \boldsymbol{x_k}$$

Learnable Attention as (asymmetric, non-Mercer) kernel transformations

$$x'_q = A\big(x_q; M(x_q, S_{x_k}); \boldsymbol{\theta}\big) = \sum_{x_k \in M(x_q, S_{x_k})} \mathrm{a}(x_q, x_k; \boldsymbol{\theta_a})\, v(x_k; \boldsymbol{\theta_v}) = \sum_{x_k \in M(x_q, S_{x_k})} \frac{\mathrm{k}(x_q, x_k; \boldsymbol{\theta_k})}{\sum_{x_{k'} \in M(x_q, S_{x_k})} \mathrm{k}(x_q, x_{k'}; \boldsymbol{\theta_k})}\, v(x_k; \boldsymbol{\theta_v})$$

- We can introduce learnable parameters
  - We can learn **which input tokens should associate more with other tokens** to produce a representation that when passed to the predictor should produce the target output
  - For example, **"Attention Is All You Need"** paper uses the following functions with three learnable weight matrices $\boldsymbol{W_q}$, $\boldsymbol{W_k}$ and $\boldsymbol{W_v}$



$$\mathrm{k}(x_q, x_k) = exp\left(\frac{1}{\sqrt{d}} \overbrace{\langle x_q \boldsymbol{W_q}, x_k \boldsymbol{W_k} \rangle}^{\text{Dot Product}}\right)$$

$$v(x_k) = x_k \boldsymbol{W_v}$$

$$x'_q = A\big(x_q; M(x_q, S_{x_k}); \boldsymbol{\theta}\big)$$

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \longrightarrow$$

https://en.wikipedia.org/wiki/Softmax_function

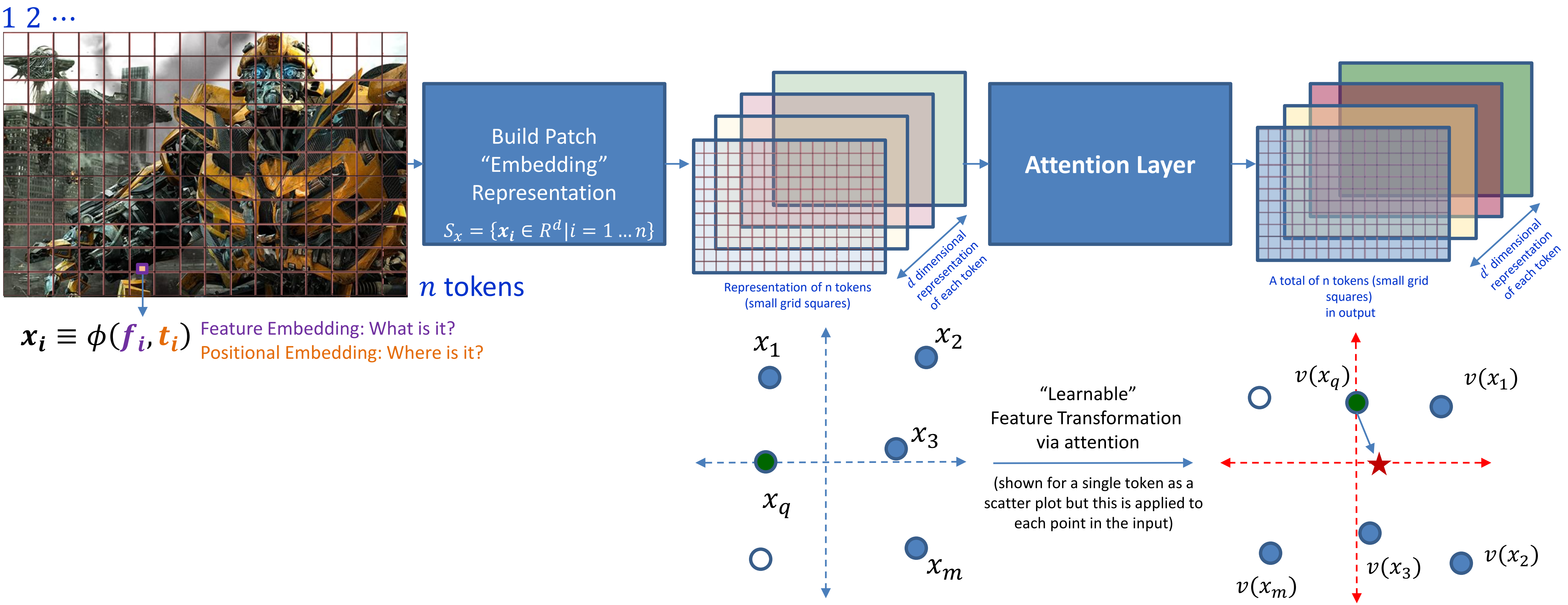
$$x'_q = A\Big(x_q; M(x_q, S_{x_k})\Big) =$$

$$\sum_{x_k \in M(x_q, S_{x_k})} \frac{exp\left(\frac{1}{\sqrt{d}} \langle x_q \boldsymbol{W_q}, x_k \boldsymbol{W_k} \rangle\right)}{\sum_{x_{k'} \in M(x_q, S_{x_k})} exp\left(\frac{1}{\sqrt{d}} \langle x_q \boldsymbol{W_q}, x_{k'} \boldsymbol{W_k} \rangle\right)} x_k \boldsymbol{W_v} =$$

$$softmax\left(\frac{1}{\sqrt{d}} x_q \boldsymbol{W_q} (x_k \boldsymbol{W_k})^T\right) x_k \boldsymbol{W_v} =$$

$$softmax\left(\frac{1}{\sqrt{d}} qK^T\right) V$$

Output of a single attention layer



$1\ 2\ \cdots$

n tokens

$x_i \equiv \phi(f_i, t_i)$ Feature Embedding: What is it? Positional Embedding: Where is it?

Build Patch "Embedding" Representation

$S_x = \{x_i \in R^d | i = 1 \ldots n\}$

Representation of n tokens (small grid squares)

d dimensional representation of each token

Attention Layer

A total of n tokens (small grid squares) in output

d' dimensional representation of each token

"Learnable" Feature Transformation via attention

(shown for a single token as a scatter plot but this is applied to each point in the input)

Note that the representation of each patch (or token) at the output of attention is dependent upon the representation of all other patches in a end-to-end learnable manner so that when this representation is used for a prediction task, the loss is minimized

Attention gives transformations

- Another way of looking at an attention operation

Learnable “attention” values

$$A\left(x_q; M\left(x_q, S_{x_k}\right)\right) = \sum_{x_k \in M\left(x_q, S_{x_k}\right)} \frac{\mathrm{k}\left(x_q, x_k\right)}{\sum_{x_{k'} \in M\left(x_q, S_{x_k}\right)} \mathrm{k}\left(x_q, x_k\right)} v(x_k) = \sum_{x_k \in S_{x_k}} a(x_q, x_k; W)\, v(x_k; W')$$

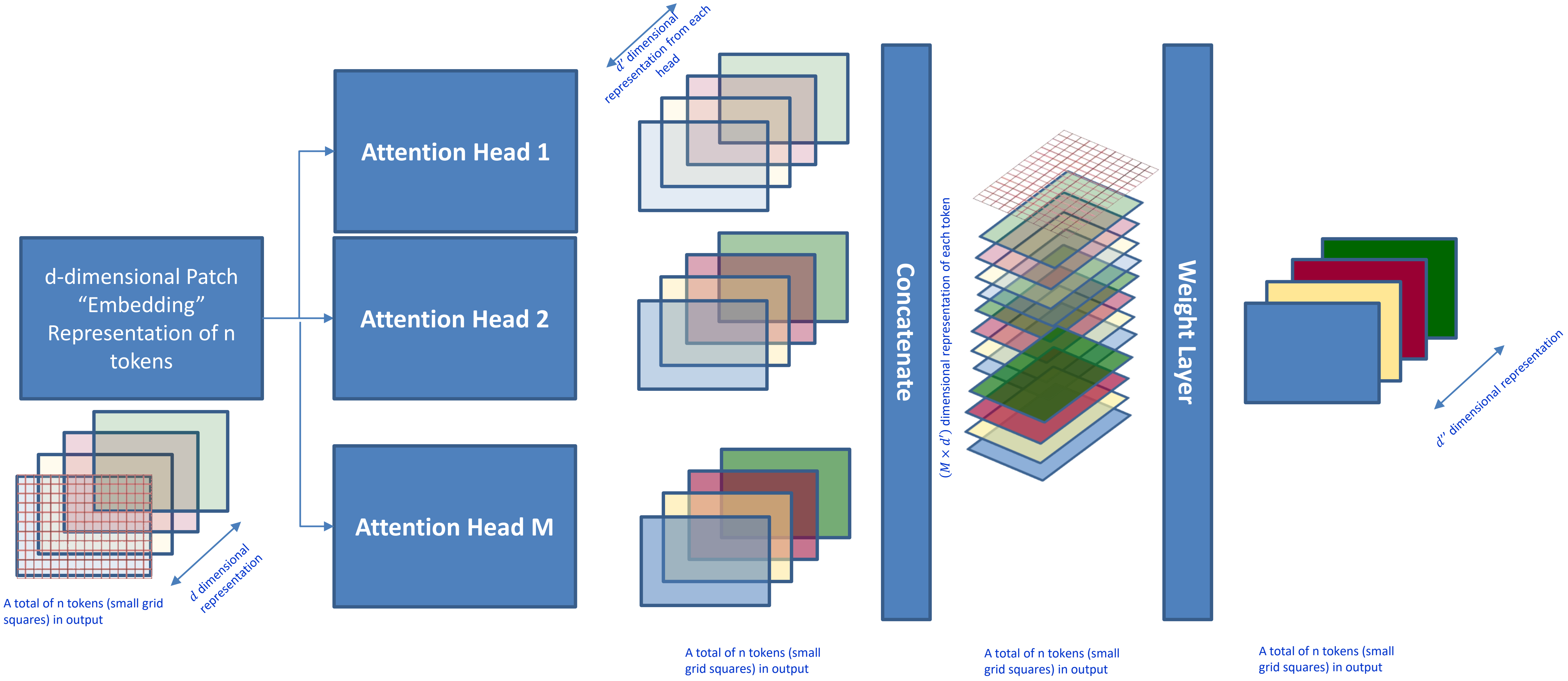
Learnable data transformation

- But a classic neural network layer also “learns” to “transform”

$$F(x; \theta) = activation(\theta_{d' \times d} x_{d \times 1})$$

- Where is the extra information coming from?
  - From <u>comparing against all tokens</u> and using a supervisory signal to learn the transform
  - Weight sharing across all patches is still there like in a convolutional neural network

Multi-headed Attention



d-dimensional Patch “Embedding” Representation of n tokens

A total of n tokens (small grid squares) in output

d dimensional representation

Attention Head 1

Attention Head 2

Attention Head M

d' dimensional representation from each head

A total of n tokens (small grid squares) in output

Concatenate

$(M \times d')$ dimensional representation of each token

A total of n tokens (small grid squares) in output

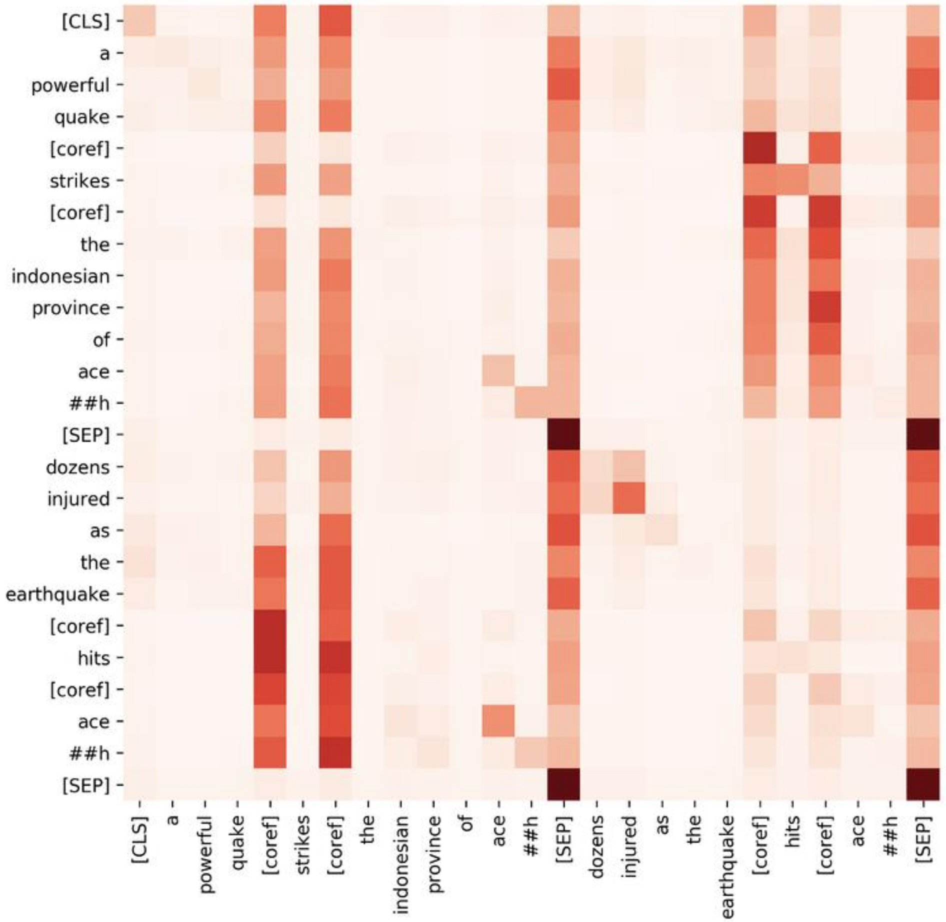
Weight Layer

d'' dimensional representation

A total of n tokens (small grid squares) in output

What happens at the end of the Training phase?

- We learn
  - [For NLP] The representation or embedding of different tokens only in reference to representations of other tokens
  - The association between different tokens
  - How to transform different tokens

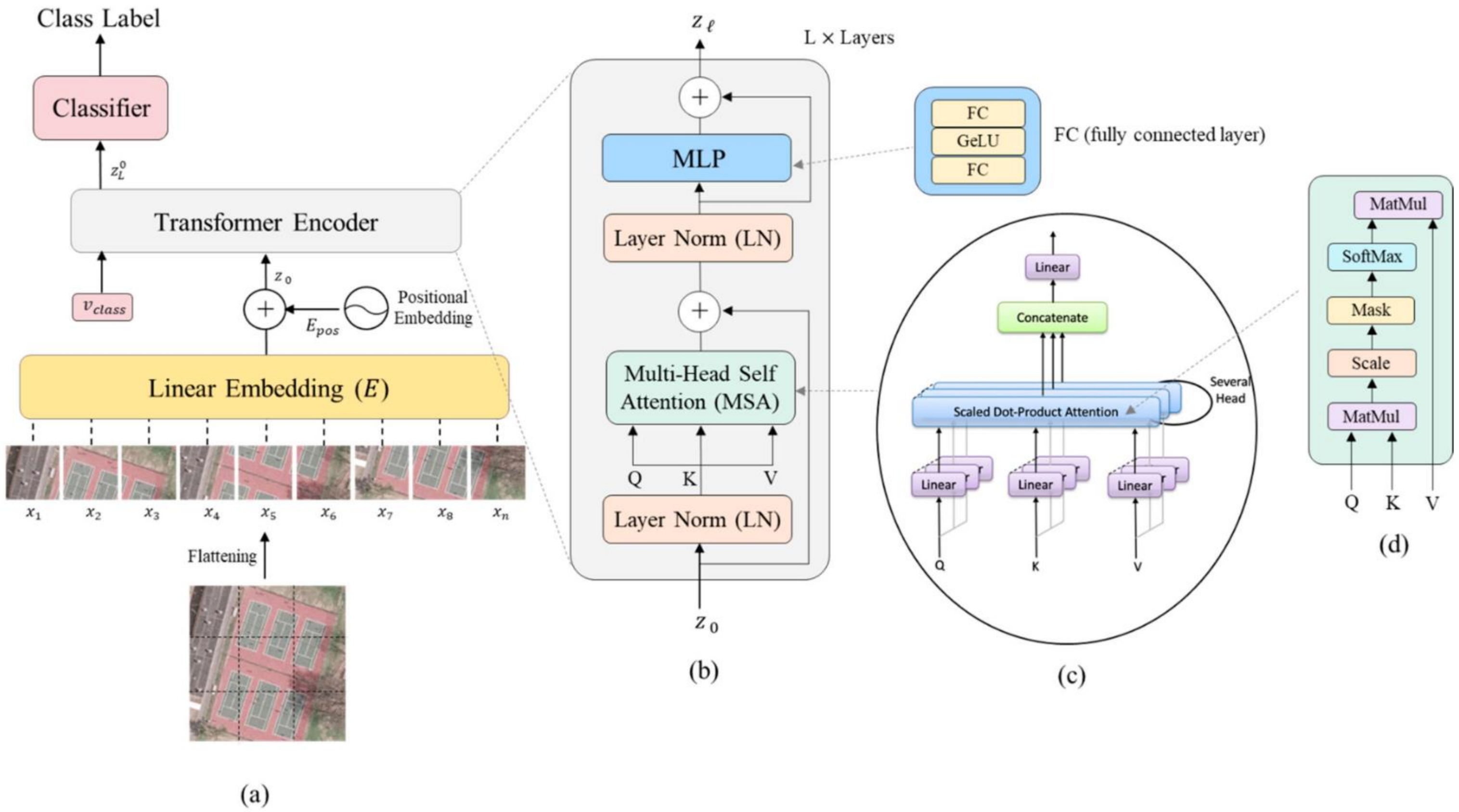


https://github.com/jessevig/bertviz

Transformers

Key Components

- Multiple Multi-headed Attention Blocks
- Layer Normalization
  - standardization across features of the same input
- Skip Connections
- Various types of positional encodings
- “Class” tokens
  - Add global features to each example to enable global sharing of information across examples
- Masking strategies
  - Needed for training in sentence completion or related problems where the next work cannot be used for generating the output
- Computational Complexity
  - As we compare each token against every other, transformers can be quite complex
  - **Performer** architectures
    - Uses kernel approximation to reduce complexity



My PyTorch tutorial: https://github.com/foxtrotmike/CS909/blob/master/mnist_transformer.ipynb

Another Tutorial: https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c

Figure from : Bazi, Yakoub, Laila Bashmal, Mohamad M. Al Rahhal, Reham Al Dayil, and Naif Al Ajlan. “Vision Transformers for Remote Sensing Image Classification.” *Remote Sensing* 13, no. 3 (January 2021): 516. https://doi.org/10.3390/rs13030516.

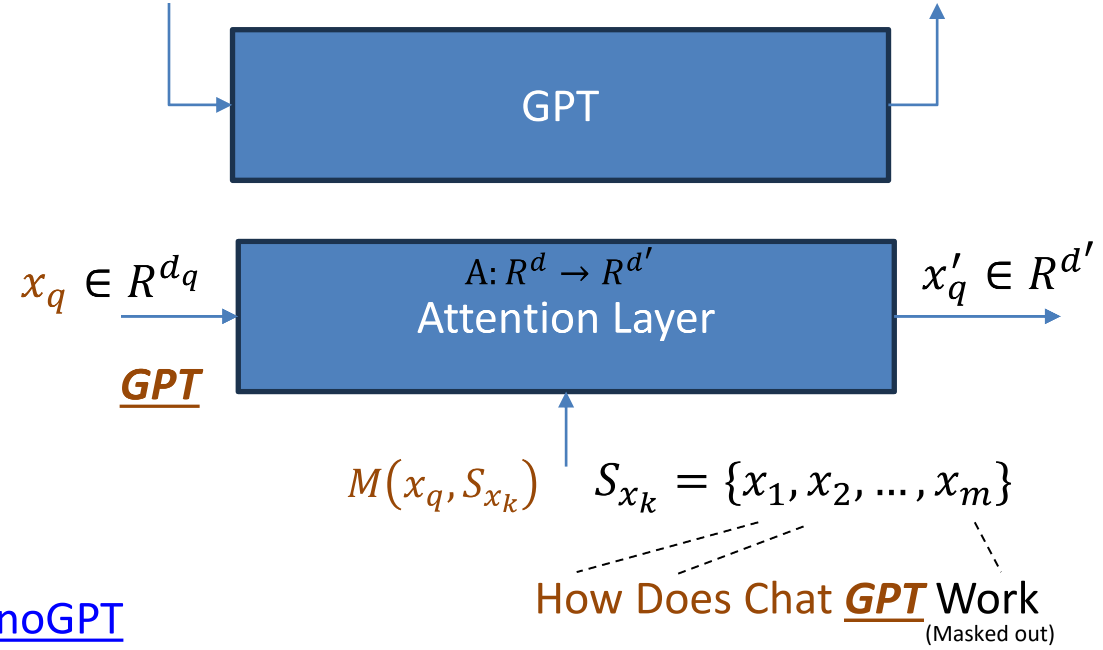
Krzysztof, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, et al. “Rethinking Attention with Performers.” arXiv, November 19, 2022. https://doi.org/10.48550/arXiv.2009.14794

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. “**An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale**.” arXiv, June 3, 2021. https://doi.org/10.48550/arXiv.2010.11929.

How is an attention layer used in Chat-GPT?

- GPTs are essentially sophisticated auto-complete mechanisms
  - Predict next word
- **Training Principle**
- Taken each “document” as a set of tokens: $S_{x_k} = \{x_1, x_2, \dots, x_m\}$
  1. Take a single next-word prediction task from the document (see **bold** text on the right)
     a. For each token in the input, apply the attention layers to a single token
        i. Take a single “query” token x_q in the input for which we want to generate a representation
        ii. For the given “example” input, mask the next token, i.e., set $M(x_q, S_{x_k})$ to be a subset of only those tokens that are available as inputs
        iii. Pass x_q and $M(x_q, S_{x_k})$ to the attention layer to generate x_q'
  2. Pass the updated representation through other downstream layers until you generate the output probability of the target token
  3. Maximize the probability of the target token while minimizing the probability of all other (non-target) tokens

Input	Next Token
How	Does
How does	Chat
How does Chat	GPT
How does Chat GPT	**Work**
… all setences in the internet corpus …	



GPT

$x_q \in R^{d_q}$

$A: R^d \rightarrow R^{d'}$

Attention Layer

$x_q' \in R^{d'}$

GPT

$M(x_q, S_{x_k})$

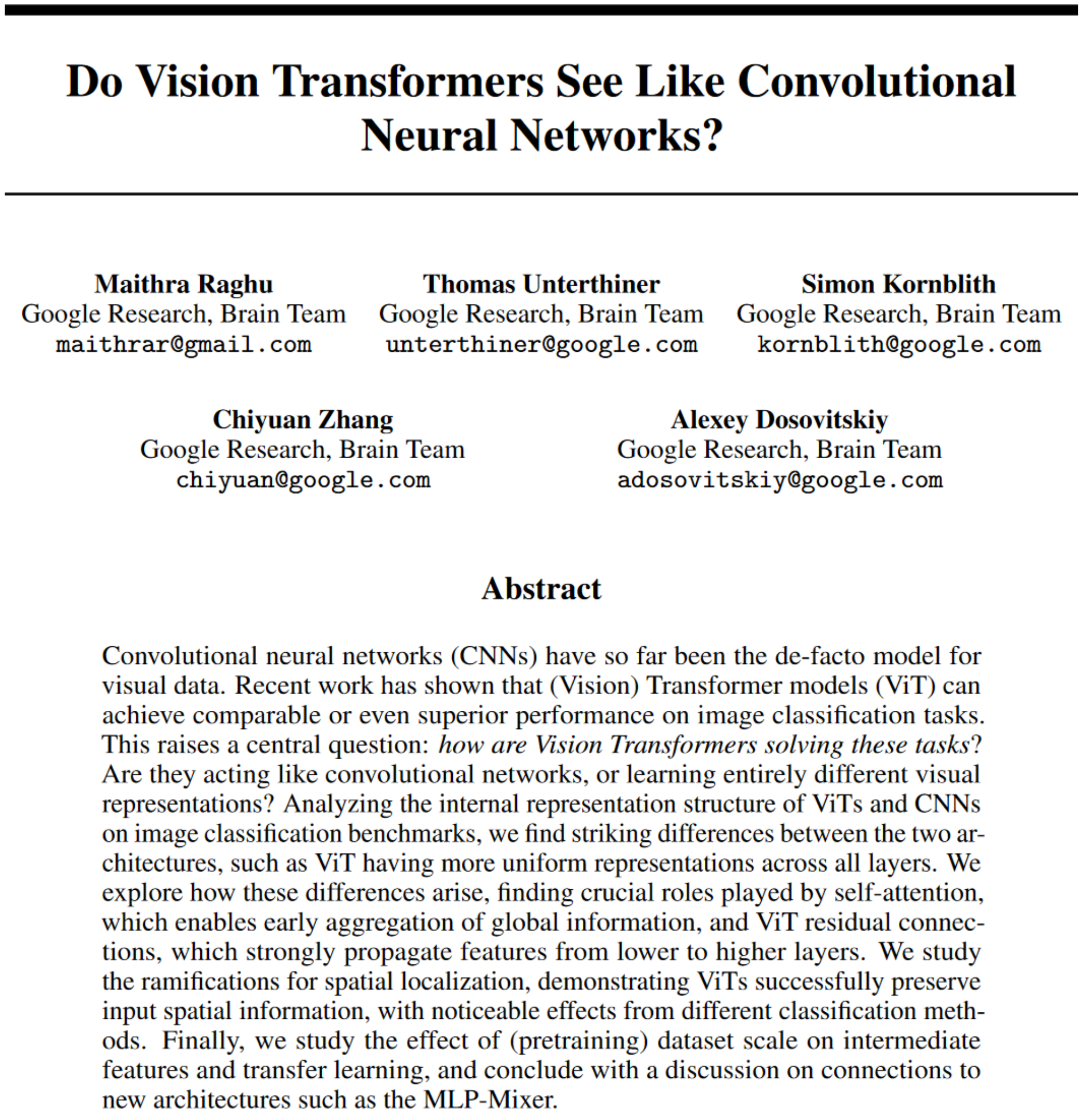
$S_{x_k} = \{x_1, x_2, \dots, x_m\}$

How Does Chat ***GPT*** Work

(Masked out)

https://github.com/karpathy/nanoGPT

What do transformers see?



Do Vision Transformers See Like Convolutional Neural Networks?

Maithra Raghu
Google Research, Brain Team
`maithrar@gmail.com`

Thomas Unterthiner
Google Research, Brain Team
`unterthiner@google.com`

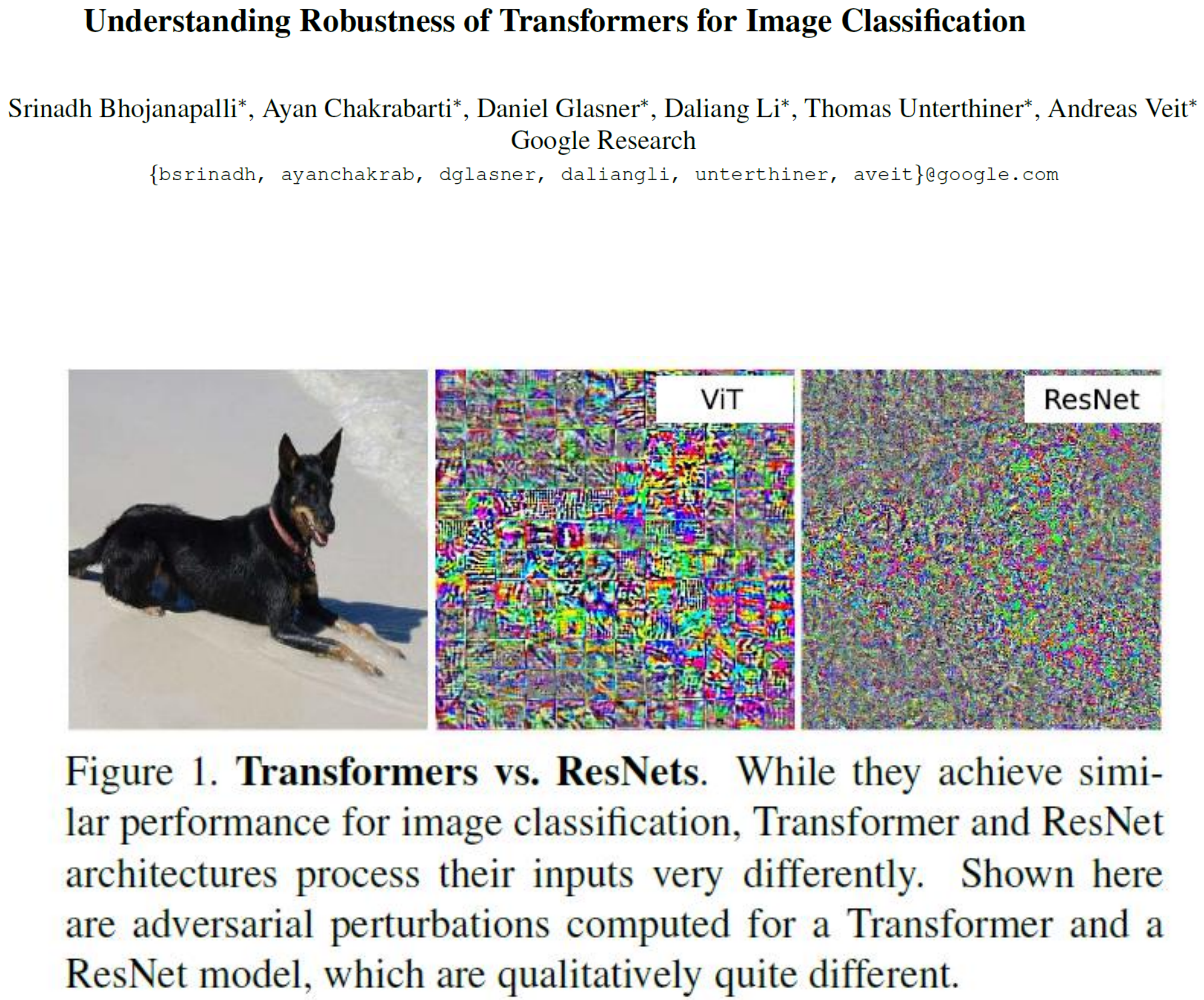
Simon Kornblith
Google Research, Brain Team
`kornblith@google.com`

Chiyuan Zhang
Google Research, Brain Team
`chiyuan@google.com`

Alexey Dosovitskiy
Google Research, Brain Team
`adosovitskiy@google.com`

Abstract

Convolutional neural networks (CNNs) have so far been the de-facto model for visual data. Recent work has shown that (Vision) Transformer models (ViT) can achieve comparable or even superior performance on image classification tasks. This raises a central question: *how are Vision Transformers solving these tasks*? Are they acting like convolutional networks, or learning entirely different visual representations? Analyzing the internal representation structure of ViTs and CNNs on image classification benchmarks, we find striking differences between the two architectures, such as ViT having more uniform representations across all layers. We explore how these differences arise, finding crucial roles played by self-attention, which enables early aggregation of global information, and ViT residual connections, which strongly propagate features from lower to higher layers. We study the ramifications for spatial localization, demonstrating ViTs successfully preserve input spatial information, with noticeable effects from different classification methods. Finally, we study the effect of (pretraining) dataset scale on intermediate features and transfer learning, and conclude with a discussion on connections to new architectures such as the MLP-Mixer.



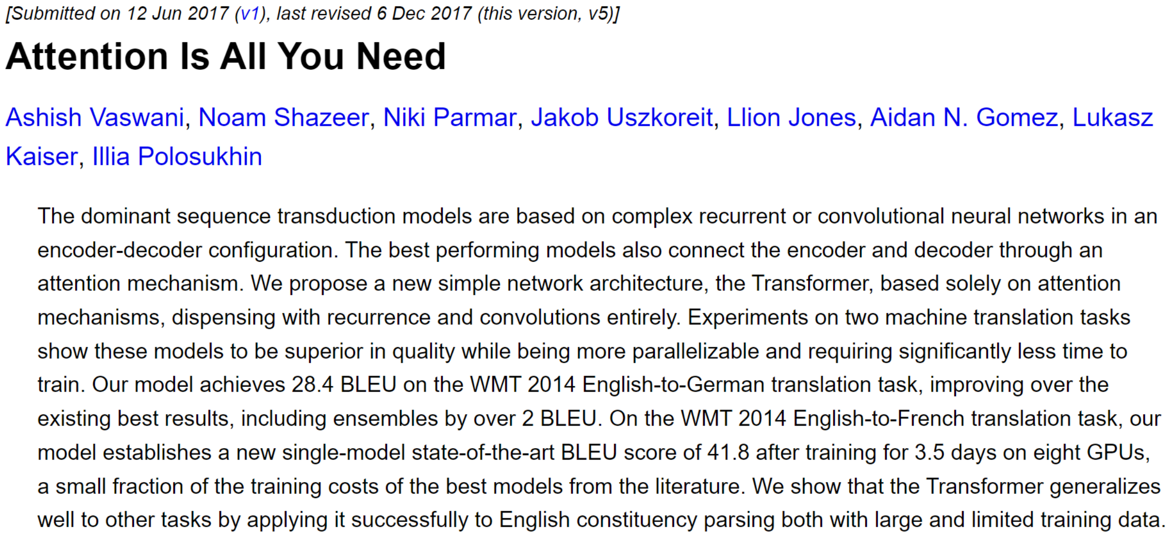
Understanding Robustness of Transformers for Image Classification

Srinadh Bhojanapalli*, Ayan Chakrabarti*, Daniel Glasner*, Daliang Li*, Thomas Unterthiner*, Andreas Veit*
Google Research
`{bsrinadh, ayanchakrab, dglasner, daliangli, unterthiner, aveit}@google.com`

Figure 1. **Transformers vs. ResNets**. While they achieve similar performance for image classification, Transformer and ResNet architectures process their inputs very differently. Shown here are adversarial perturbations computed for a Transformer and a ResNet model, which are qualitatively quite different.

Are convolutions and attention really necessary?

- MLP Mixer Paper: “In this paper we show that while convolutions and attention are both sufficient for good performance, neither of them are necessary. ”
- gMLP: “self-attention is not critical for Vision Transformers”
- Attention with Convolution may be more useful ☺



[Submitted on 12 Jun 2017 (v1), last revised 6 Dec 2017 (this version, v5)]

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder-decoder configuration. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

“Similar to fully-connected networks, the ViT architecture (and transformer architecture in general) lacks the inductive bias for spatial invariance/equivariance that convolutional networks have. Consequently, ViTs require more data for pretraining to acquire useful "priors" from the training data.” (S. Raschka)

https://twitter.com/rasbt/status/1636371712467177472

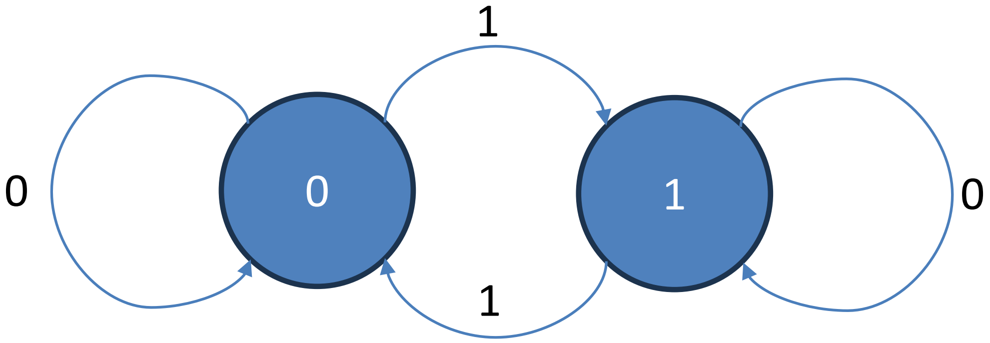
Tolstikhin, Ilya, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, et al. “MLP-Mixer: An All-MLP Architecture for Vision.” arXiv, June 11, 2021. https://doi.org/10.48550/arXiv.2105.01601.
Liu, Hanxiao, Zihang Dai, David R. So, and Quoc V. Le. “Pay Attention to MLPs.” arXiv, June 1, 2021. https://doi.org/10.48550/arXiv.2105.08050.

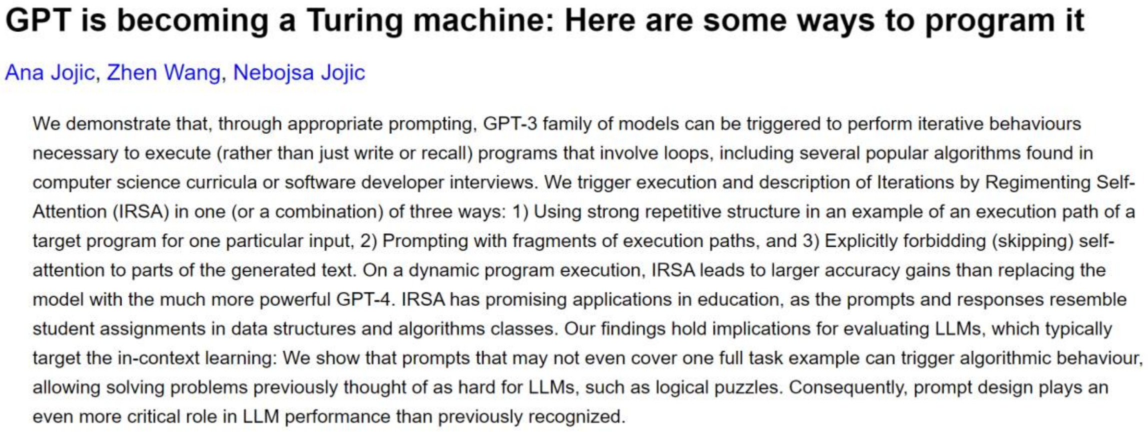
Using Transformers

- Hugging Face Transformers Library
  - Examples: https://huggingface.co/docs/transformers/model_doc/vit
  - Tutorial notebook on finetuning: https://github.com/NielsRogge/Transformers-Tutorials/blob/master/VisionTransformer/Fine_tuning_the_Vision_Transformer_on_CIFAR_10_with_the_%F0%9F%A4%97_Trainer.ipynb

Another way of thinking about GPTs

Modeling XOR as a “next token” problem or FSM or turing machine





GPT is becoming a Turing machine: Here are some ways to program it

Ana Jojic, Zhen Wang, Nebojsa Jojic

We demonstrate that, through appropriate prompting, GPT-3 family of models can be triggered to perform iterative behaviours necessary to execute (rather than just write or recall) programs that involve loops, including several popular algorithms found in computer science curricula or software developer interviews. We trigger execution and description of Iterations by Regimenting Self-Attention (IRSA) in one (or a combination) of three ways: 1) Using strong repetitive structure in an example of an execution path of a target program for one particular input, 2) Prompting with fragments of execution paths, and 3) Explicitly forbidding (skipping) self-attention to parts of the generated text. On a dynamic program execution, IRSA leads to larger accuracy gains than replacing the model with the much more powerful GPT-4. IRSA has promising applications in education, as the prompts and responses resemble student assignments in data structures and algorithms classes. Our findings hold implications for evaluating LLMs, which typically target the in-context learning: We show that prompts that may not even cover one full task example can trigger algorithmic behaviour, allowing solving problems previously thought of as hard for LLMs, such as logical puzzles. Consequently, prompt design plays an even more critical role in LLM performance than previously recognized.

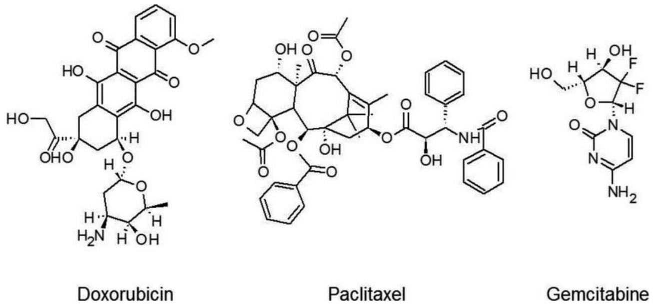
Input	Next “Target” Token	Target Probability	
		P(0)	P(1)
0,0	0	1	0
0,1	1	0	1
1,0	1	0	1
1,1	0	1	0

https://github.com/foxtrotmike/CS909/blob/master/gpt_finite_state.ipynb

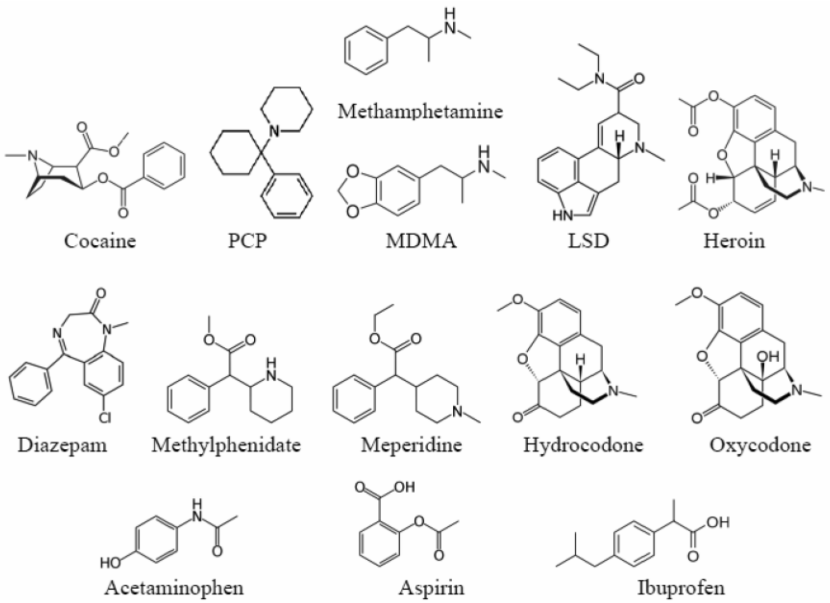
GRAPH NEURAL NETWORKS

Graph Neural Networks

- The Need
  - Example
    - Classifying chemical compounds
  - It is difficult to model arbitrary input data structures with SVMs, MLPs, CNNs and Transformers
    - Images and text have “Linear Structure”
      - Text is 1-dimensional
      - Image is 2-dimensional
      - But each can be mapped onto a grid



Cancer Drugs

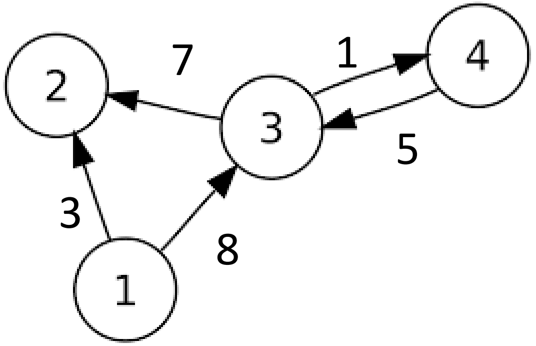


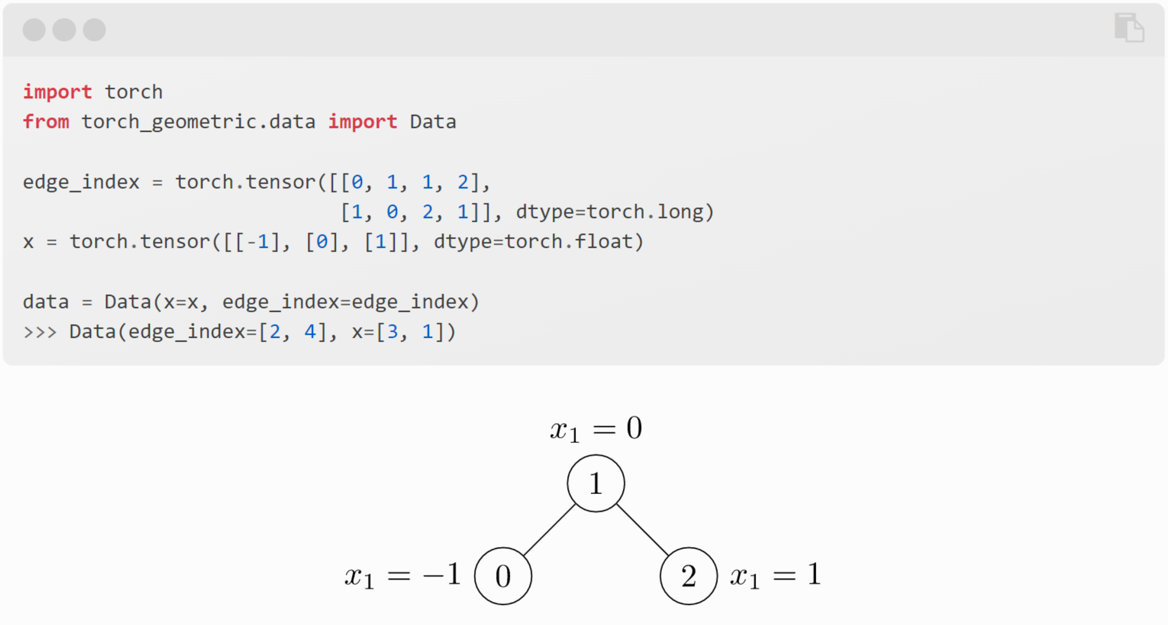
Other Drugs

Graphs

- Graph Modelling
  - Very flexible data structure

- Components of a graph
  - Vertices/Node Set: $V = \{x_1, x_2, x_3, x_4\}$
    - Each element of the set can have a vector descriptor of its properties
  - Edge Set: $E = \{e_{1,2}, e_{1,3}, e_{3,2}, e_{3,4}, e_{4,3}\} \subseteq V \times V$
    - Each element of the set can have a vector descriptor of its properties



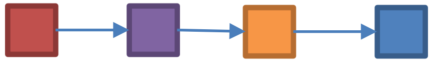


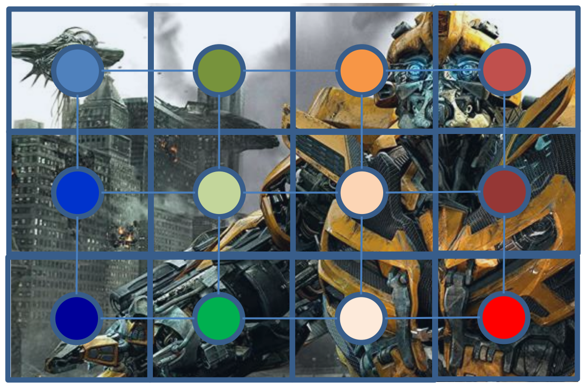
```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
>>> Data(edge_index=[2, 4], x=[3, 1])
```

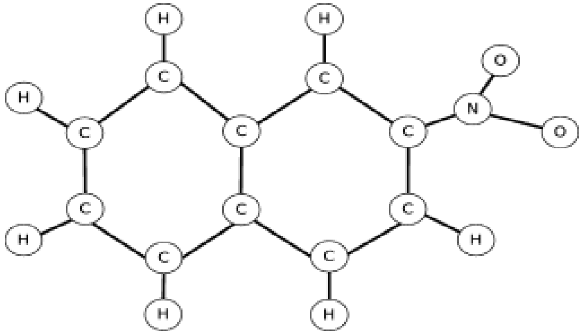
https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
https://en.wikipedia.org/wiki/Adjacency_matrix

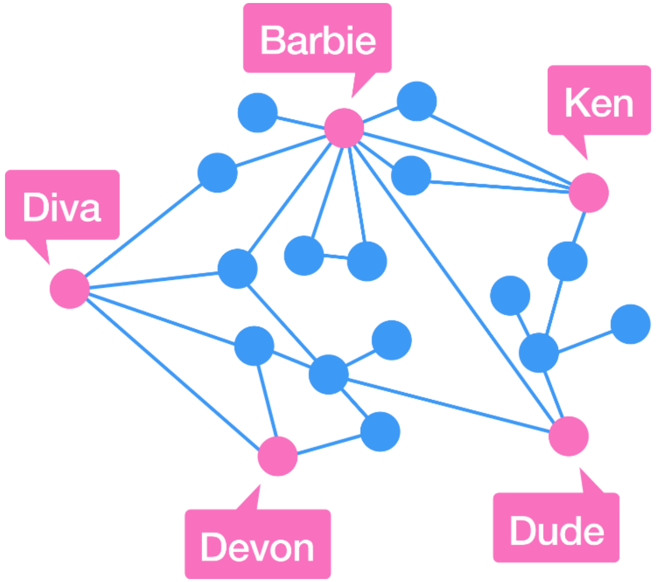
Examples of graphs

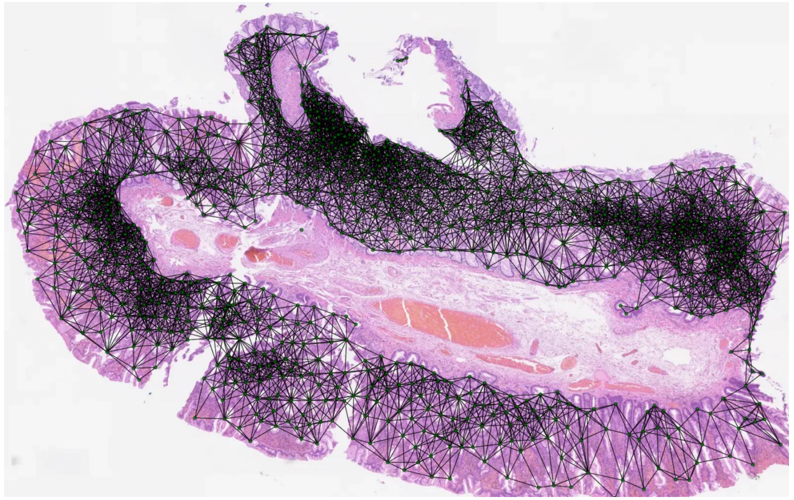
This is a graph

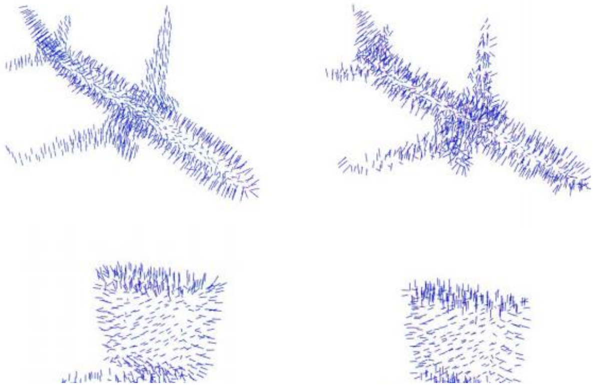






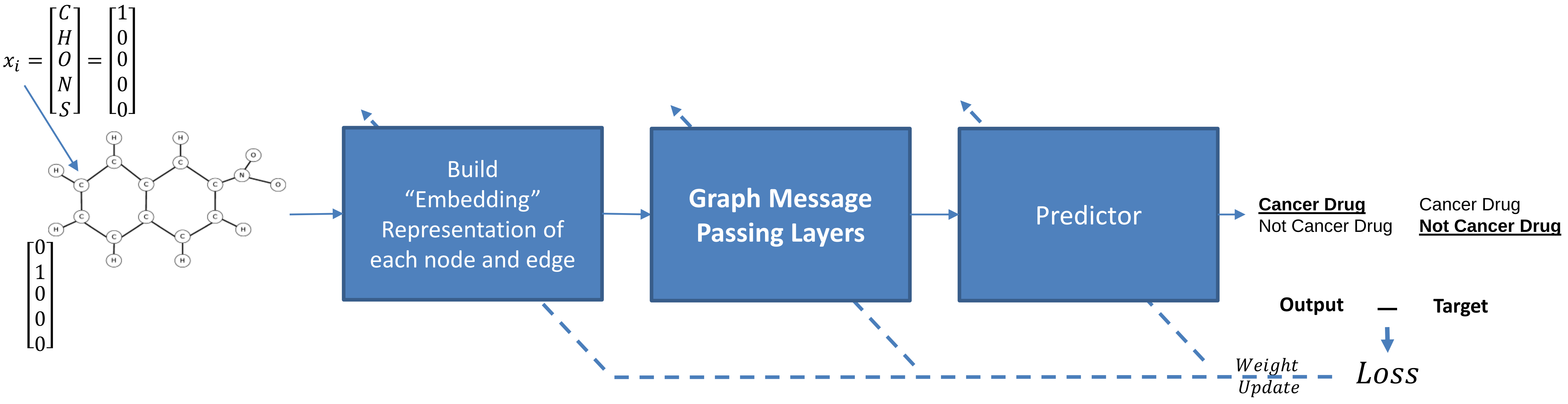






Graph Neural Networks

- Simple Graph Classification Example
  - Node and edge level prediction problems also possible



$$x_i = \begin{bmatrix} C \\ H \\ O \\ N \\ S \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Build “Embedding” Representation of each node and edge

Graph Message Passing Layers

Predictor

Cancer Drug
Not Cancer Drug

Cancer Drug
Not Cancer Drug

Output — **Target**

$Loss$

$Weight$ $Update$

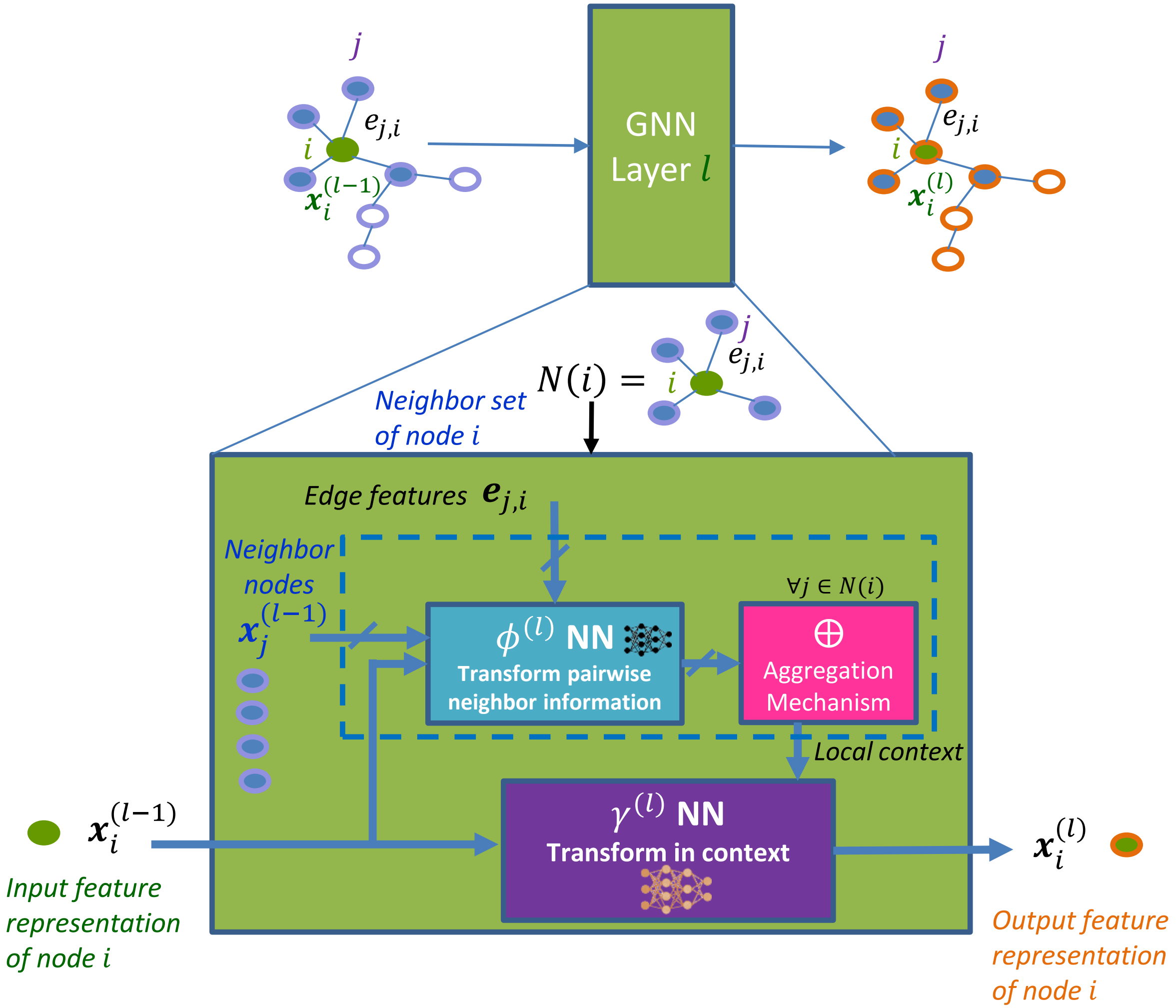
Input: Graph consisting of

Node set: what are things (each node has feature representation)

Edge set: how are they connected (each edge can have a feature representation but, in the very least, it tells us what nodes are connected by an edge)

How does a graph neural network layer work?

- Just like any other neural network layer, the goal of a graph layer is to transform the representation of the input to a new representation in a learnable/trainable fashion so that we can optimize the parameters in the layer to reduce our loss or error function
- Input: A Graph with node and edge level features
- Output: A Graph with (transformed) node and edge features
- The GNN layer transforms the feature representation of each node as follows:
  - **Where am I?** Generate context for each input node
    - **Node pair transform:** Transform features of each node connected to an input node while taking pairwise edge information into account (using a neural network)
    - **Aggregation:** Aggregate information of neighbors of the node to provide the local context in the form of a fixed dimensional feature vector (max, sum, average, etc.)
  - **What should I become?** Transform each node in the context of its neighbors (using a neural network)
- Each GNN layer thus incorporates information from one hop away of each node thus multiple GNN layers in series can be used to incorporate information from multiple layers

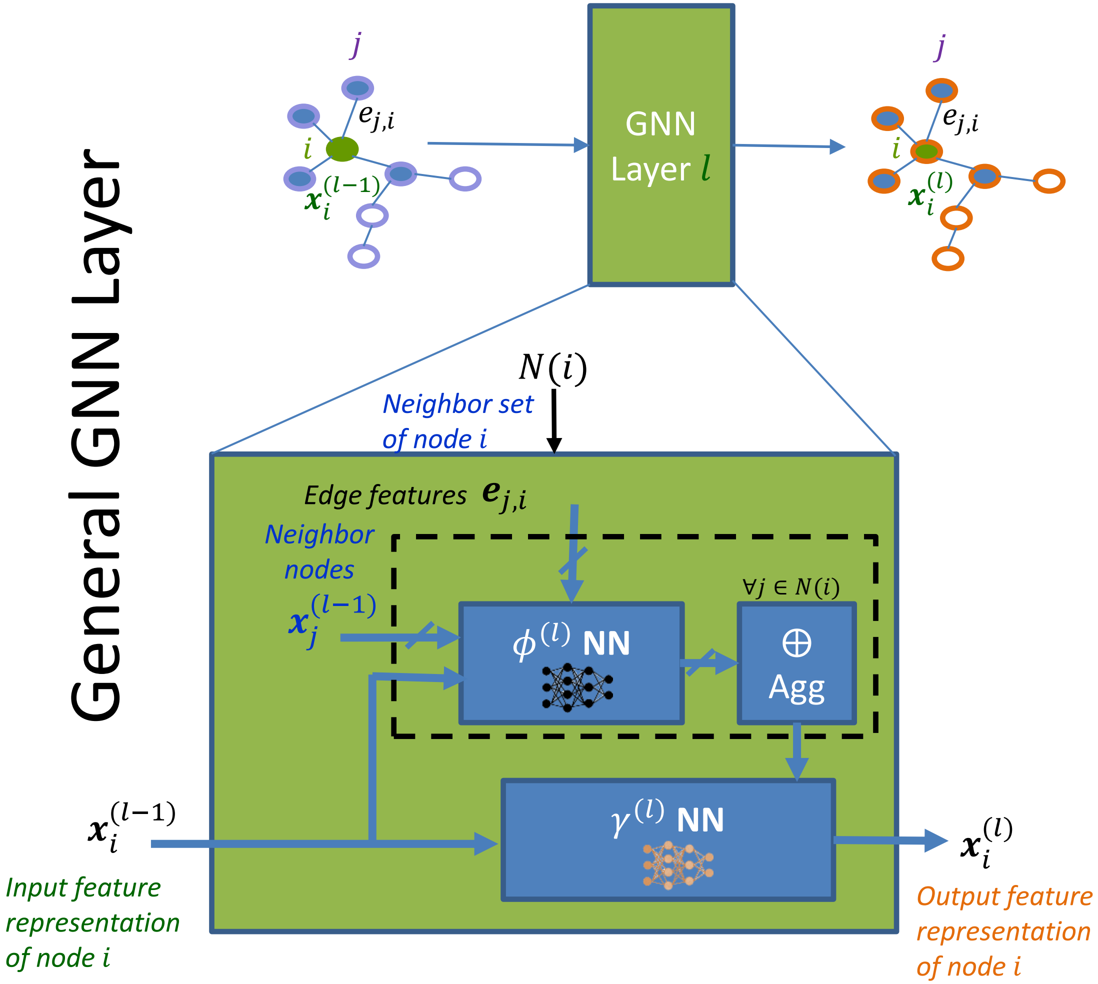


$$x_i^{(l)} = \gamma^{(l)}\left(x_i^{(l-1)}, \oplus_{j\in N(i)}\phi^{(l)}\left(x_i^{(l-1)}, x_j^{(l-1)}, e_{j,i}\right)\right)$$

https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html

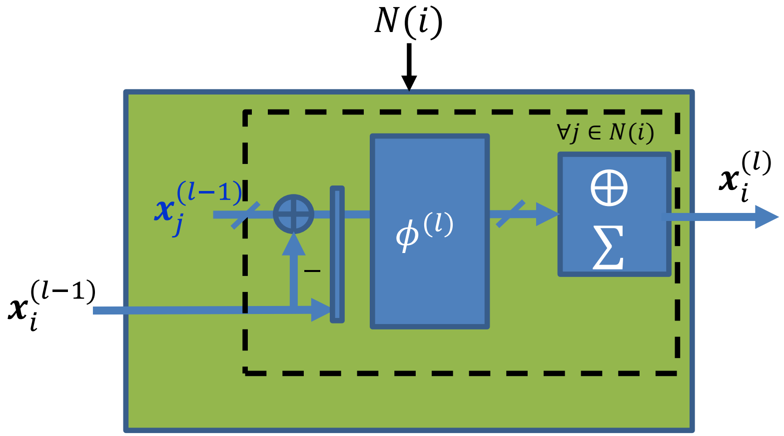
Implementing Different Graph Neural Network Layers

General GNN Layer



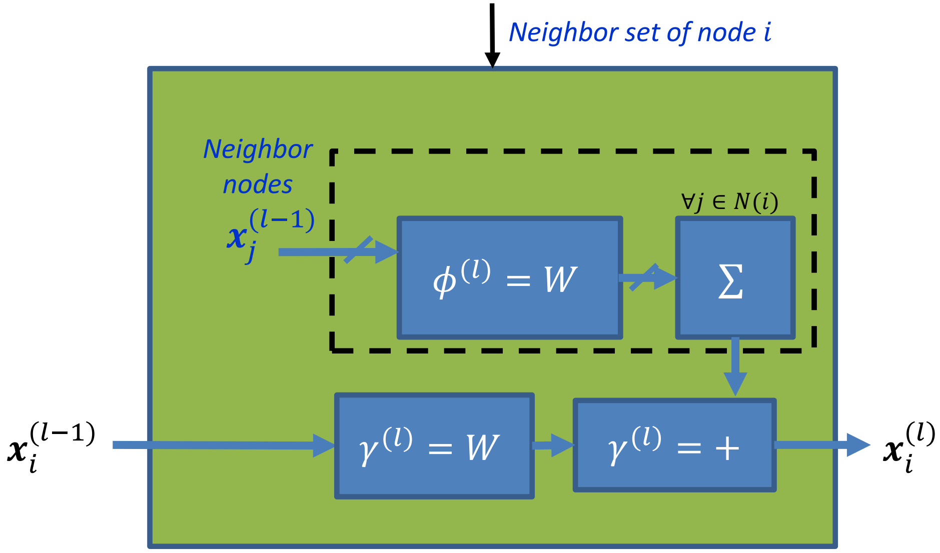
$$\mathbf{x}_i^{(l)} = \gamma^{(l)}\left(\mathbf{x}_i^{(l-1)}, \oplus_{j\in N(i)}\phi^{(l)}\left(\mathbf{x}_i^{(l-1)}, \mathbf{x}_j^{(l-1)}, \mathbf{e}_{j,i}\right)\right)$$

https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html



Edge Convolution Layer (GCNConv)

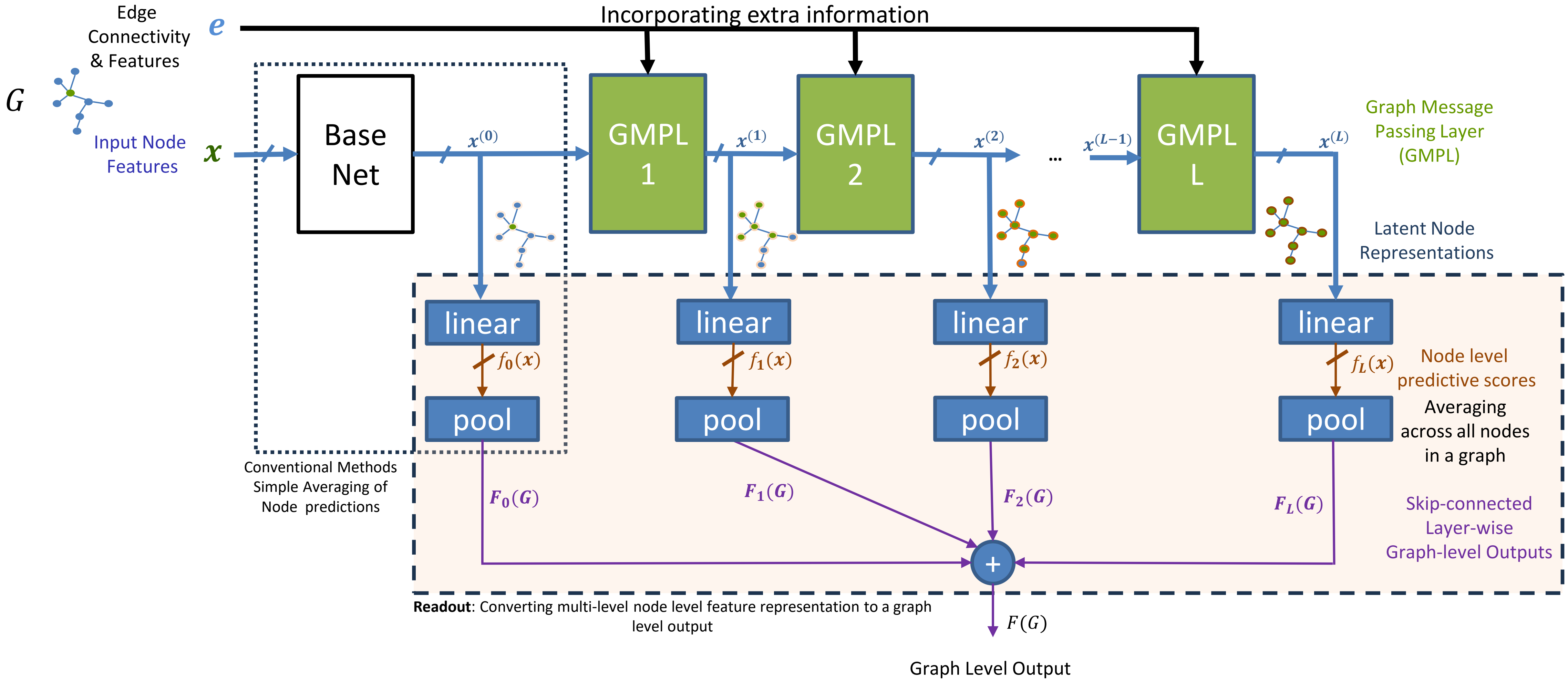
$$\mathbf{x}_i^{(l)} = \sum_{j\in N(i)} \phi^{(l)}\left(\mathbf{x}_i^{(l-1)} \,\Big\|\, \mathbf{x}_j^{(l-1)} - \mathbf{x}_i^{(l-1)}\right)$$



Graph Convolution Layer (GCNConv)

$$\mathbf{x}_i^{(l)} = \sum_{j\in N(i)\cup\{\mathrm{i}\}} \frac{1}{\sqrt{\deg(i)\deg(j)}}\left(\mathbf{W}\mathbf{x}_{\mathbf{j}} + \mathbf{b}\right)$$

Message Passing Based Graph Neural Networks



Edge Connectivity & Features

Input Node Features

Incorporating extra information

Graph Message Passing Layer (GMPL)

Latent Node Representations

Node level predictive scores

Averaging across all nodes in a graph

Skip-connected Layer-wise Graph-level Outputs

Conventional Methods Simple Averaging of Node predictions

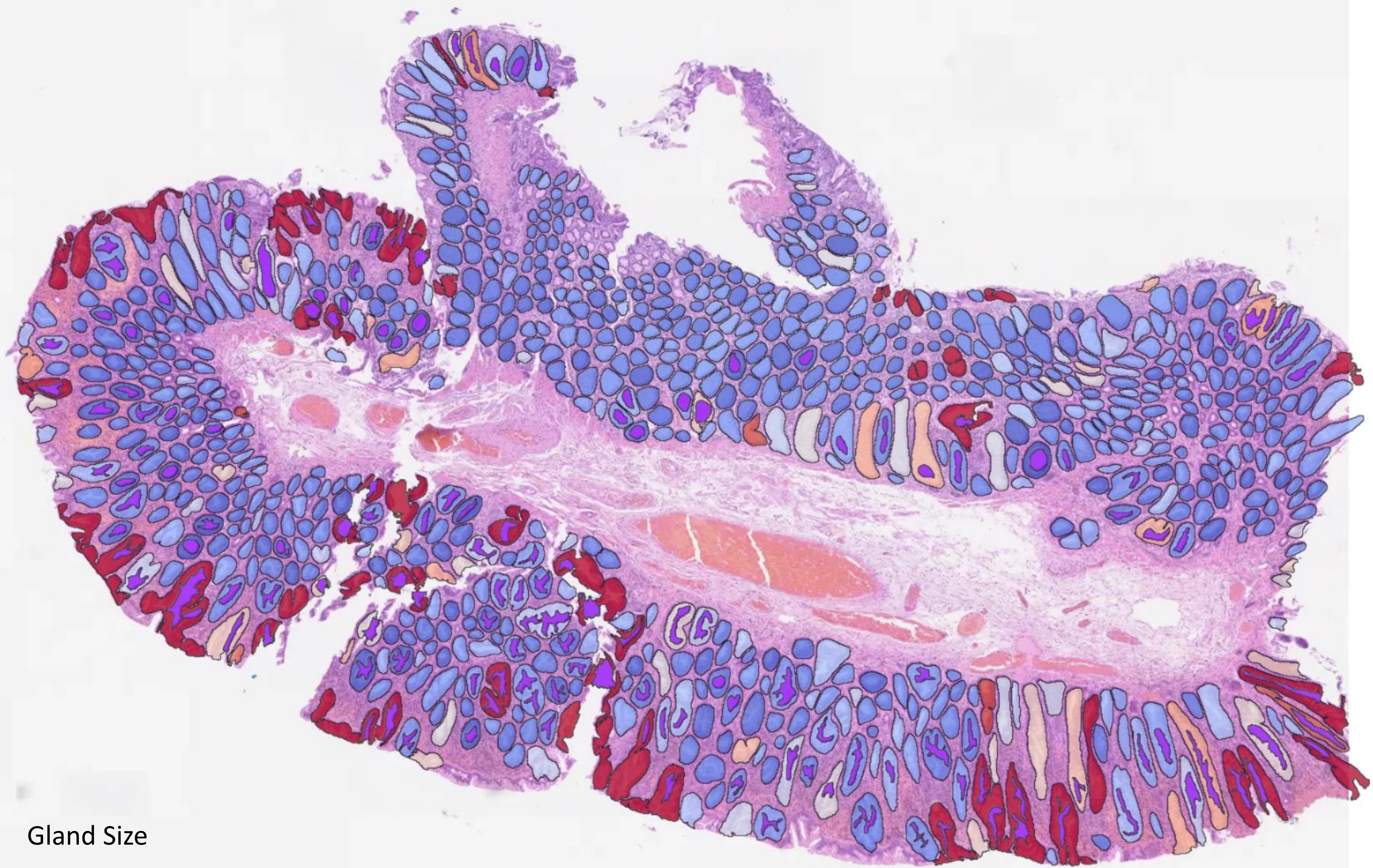
Readout: Converting multi-level node level feature representation to a graph level output

Graph Level Output

Code: https://tia-toolbox.readthedocs.io/en/latest/_notebooks/jnb/inference-pipelines/slide-graph.html

Fayyaz Minhas, Whole Slide Images Are Graphs, 2020. https://www.youtube.com/watch?v=Of1u0i7roS0.

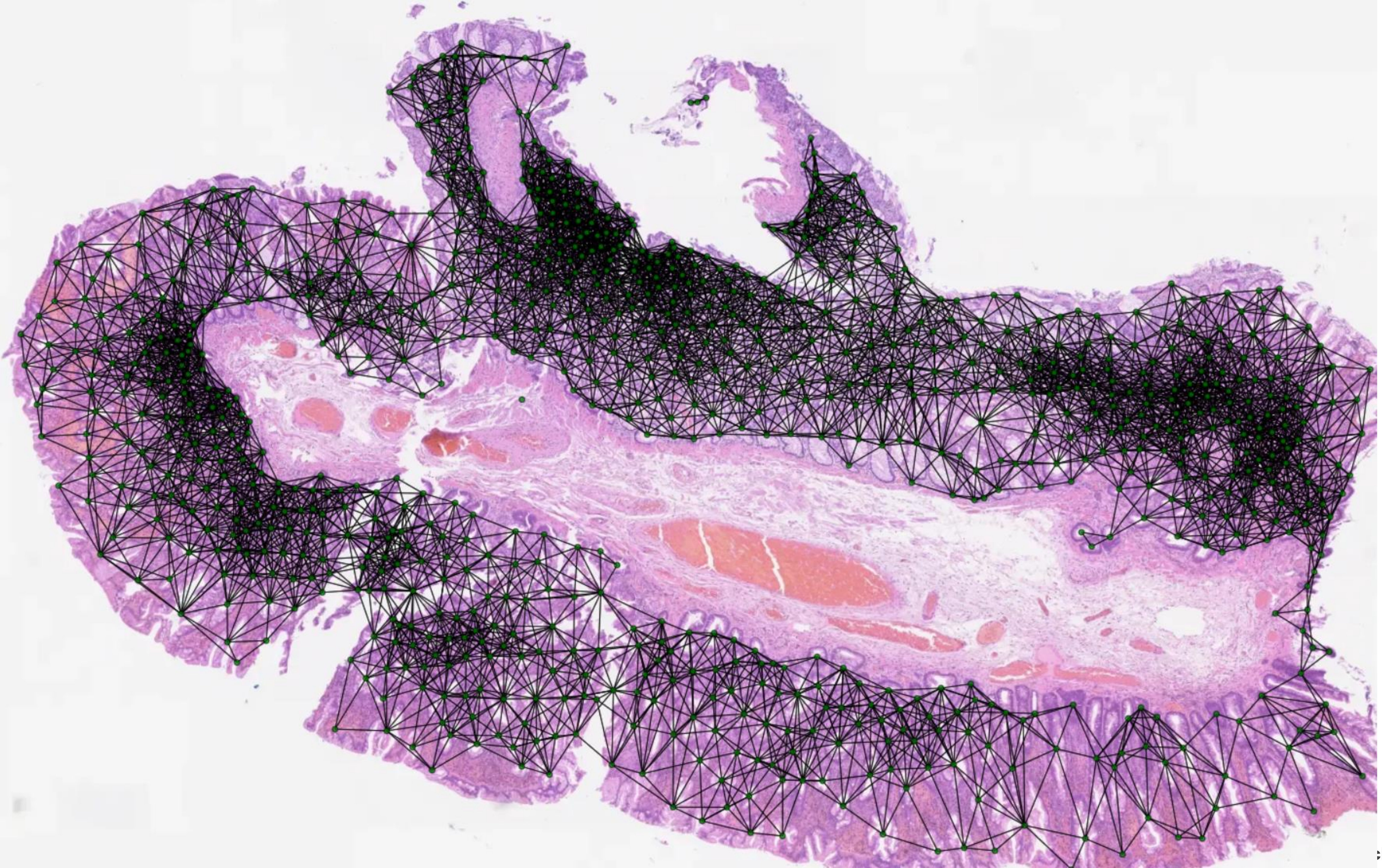
IGUANA

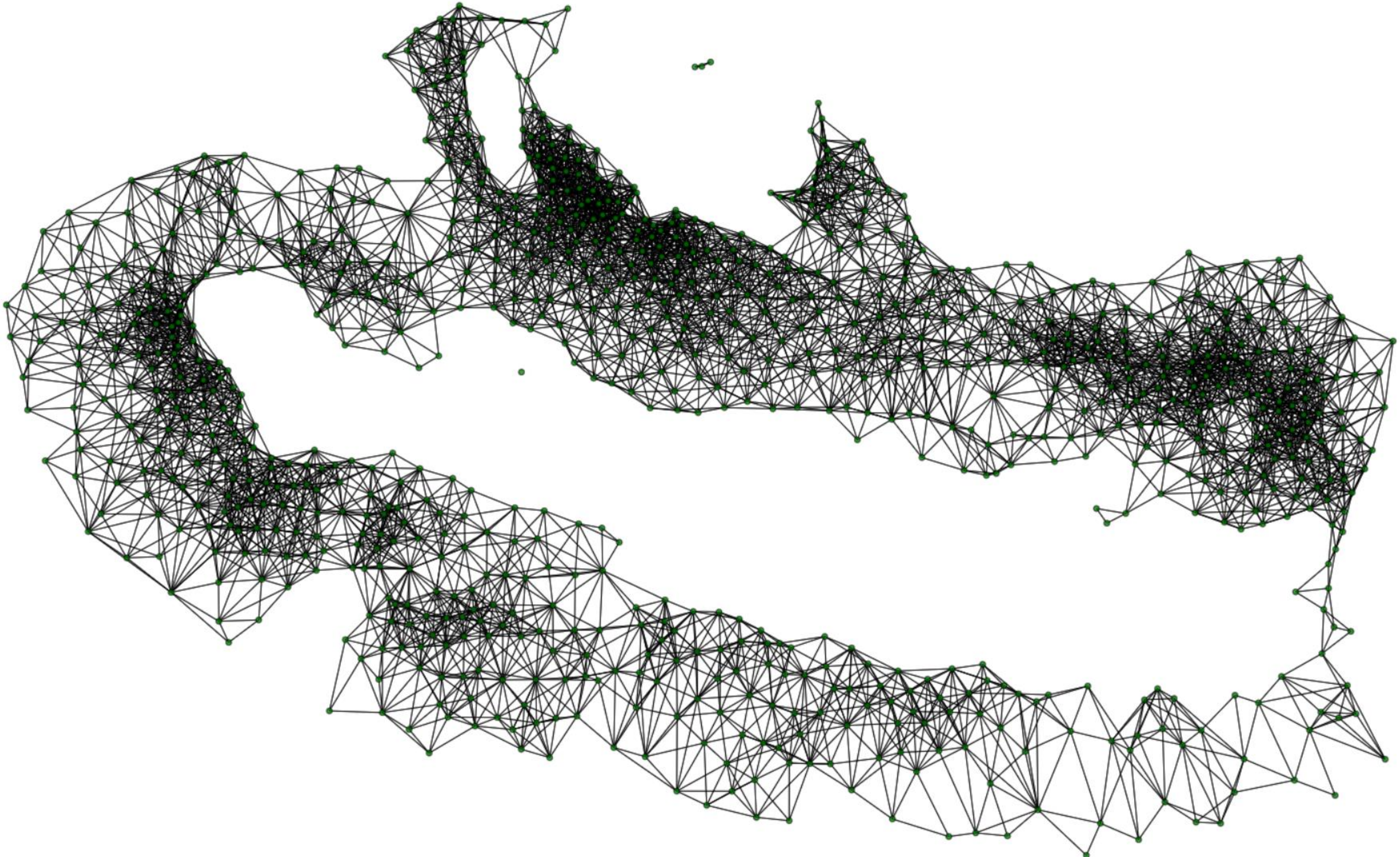


Gland Size

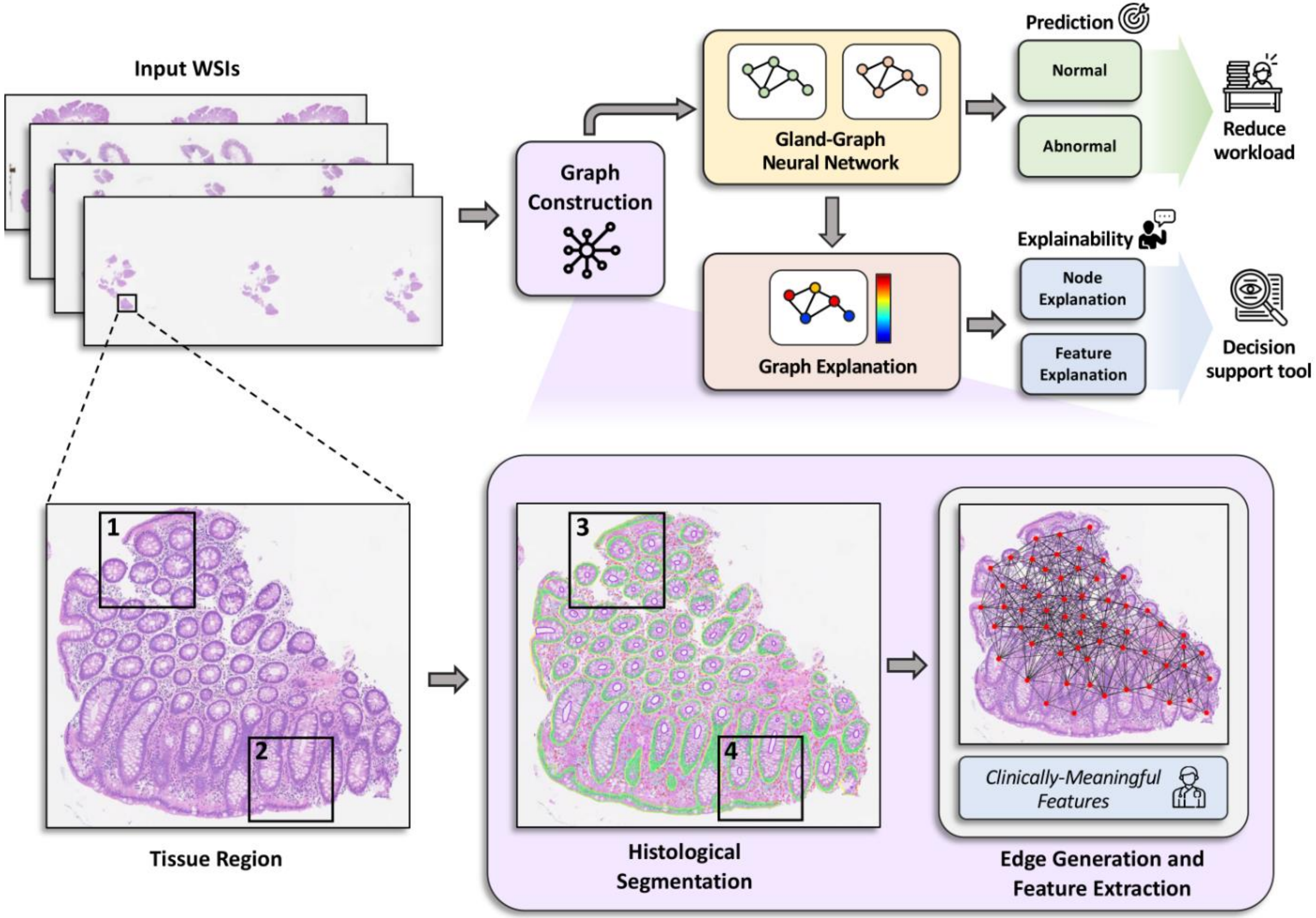
ick

Lumen Shape Irregularity





IGUANA



Input WSIs

Graph Construction

Gland-Graph Neural Network

Graph Explanation

Prediction

Normal

Abnormal

Reduce workload

Explainability

Node Explanation

Feature Explanation

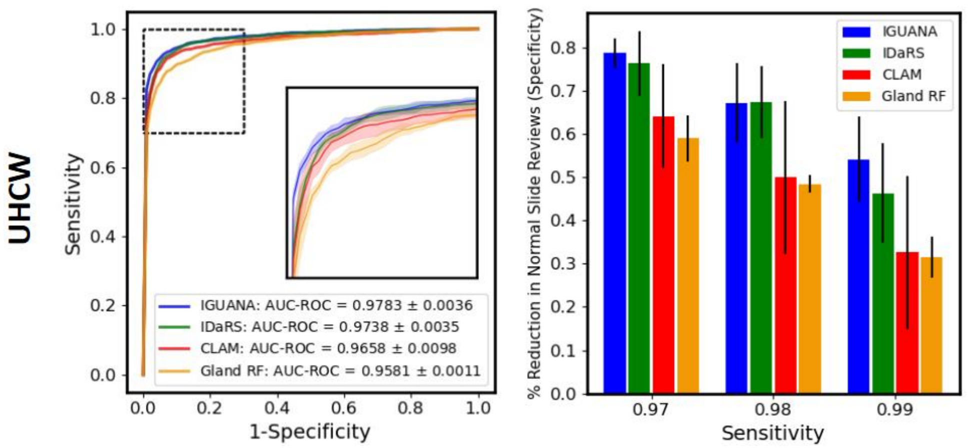
Decision support tool

Tissue Region

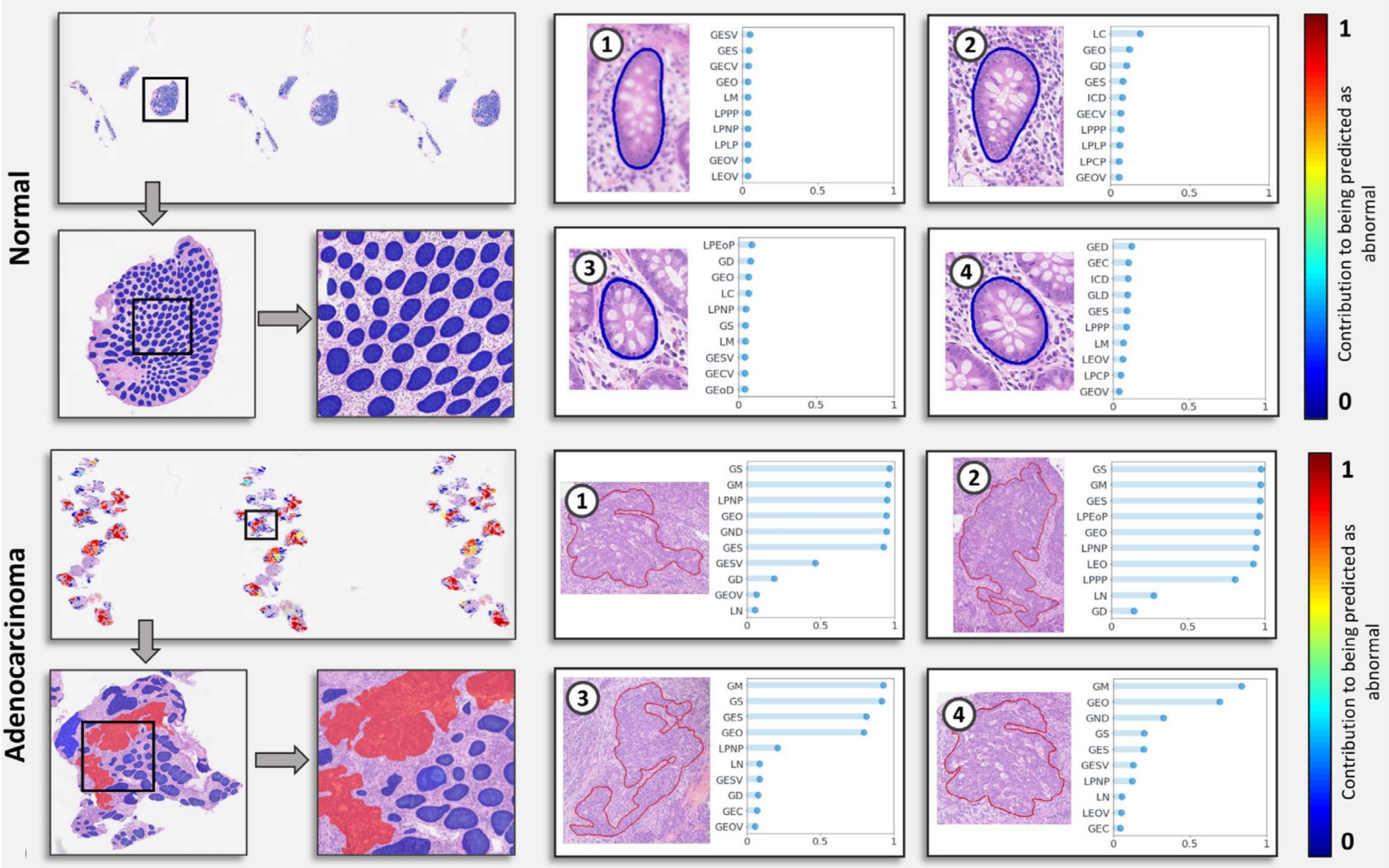
Histological Segmentation

Clinically-Meaningful Features

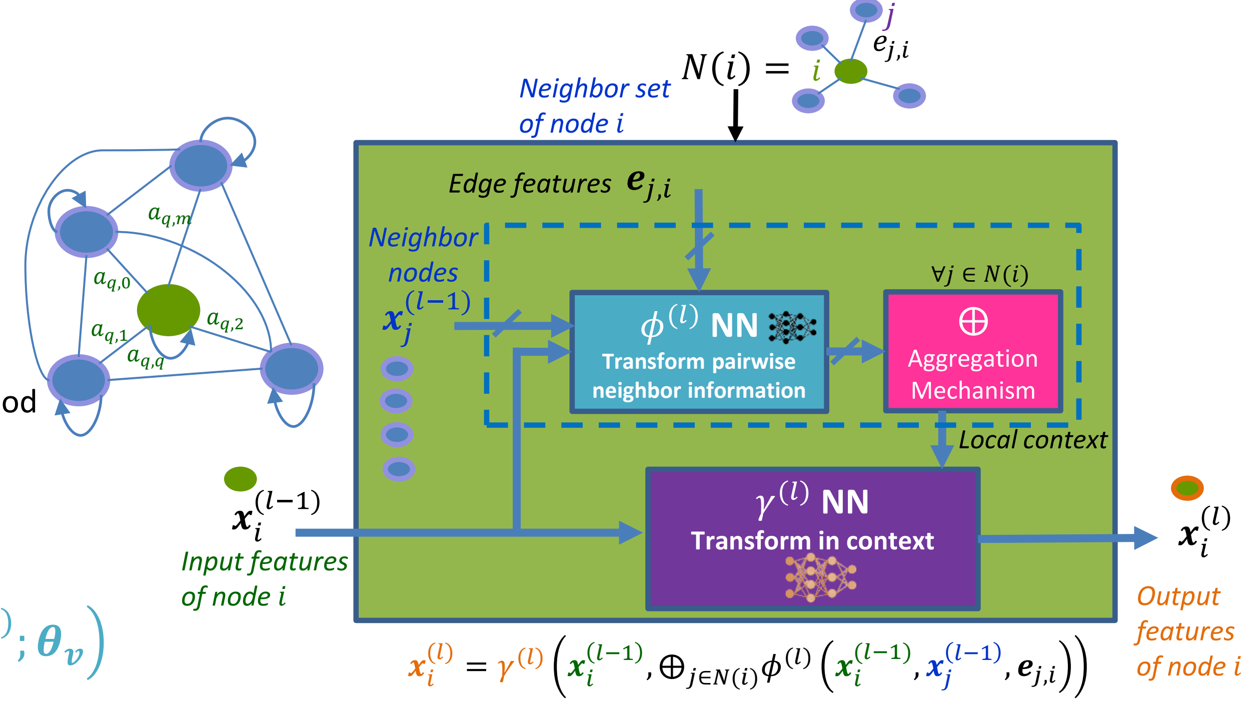
Edge Generation and Feature Extraction



Graham, Simon, Fayyaz Minhas, Mohsin Bilal, Mahmoud Ali, Yee Wah Tsang, Mark Eastwood, Noorul Wahab, et al. “Screening of Normal Endoscopic Large Bowel Biopsies with Interpretable Graph Learning: A Retrospective Study.” *Gut*, May 12, 2023. https://doi.org/10.1136/gutjnl-2023-329512.
Demo: https://tiademos.dcs.warwick.ac.uk/bokeh_app?demo=iguana


Normal
1
GESV
GES
GECV
GEO
LM
LPPP
LPNP
LPLP
GEOV
LEOV
0
0.5
1
2
LC
GEO
GD
GES
ICD
GECV
LPPP
LPLP
LPCP
GEOV
0
0.5
1
3
LPEoP
GD
GEO
LC
LPNP
GS
LM
GESV
GECV
GEoD
0
0.5
1
4
GED
GEC
ICD
GLD
GES
LPPP
LM
LEOV
LPCP
GEOV
0
0.5
1
1
Contribution to being predicted as abnormal
0
Adenocarcinoma
1
GS
GM
LPNP
GEO
GND
GES
GESV
GD
GEOV
LN
0
0.5
1
2
GS
GM
GES
LPEoP
GEO
LPNP
LEO
LPPP
LN
GD
0
0.5
1
3
GM
GS
GES
GEO
LPNP
LN
GESV
GD
GEC
GEOV
0
0.5
1
4
GM
GEO
GND
GS
GES
GESV
LPNP
LN
LEOV
GEC
0
0.5
1
1
Contribution to being predicted as abnormal
0

Are Transformers (secretly) GNNs?



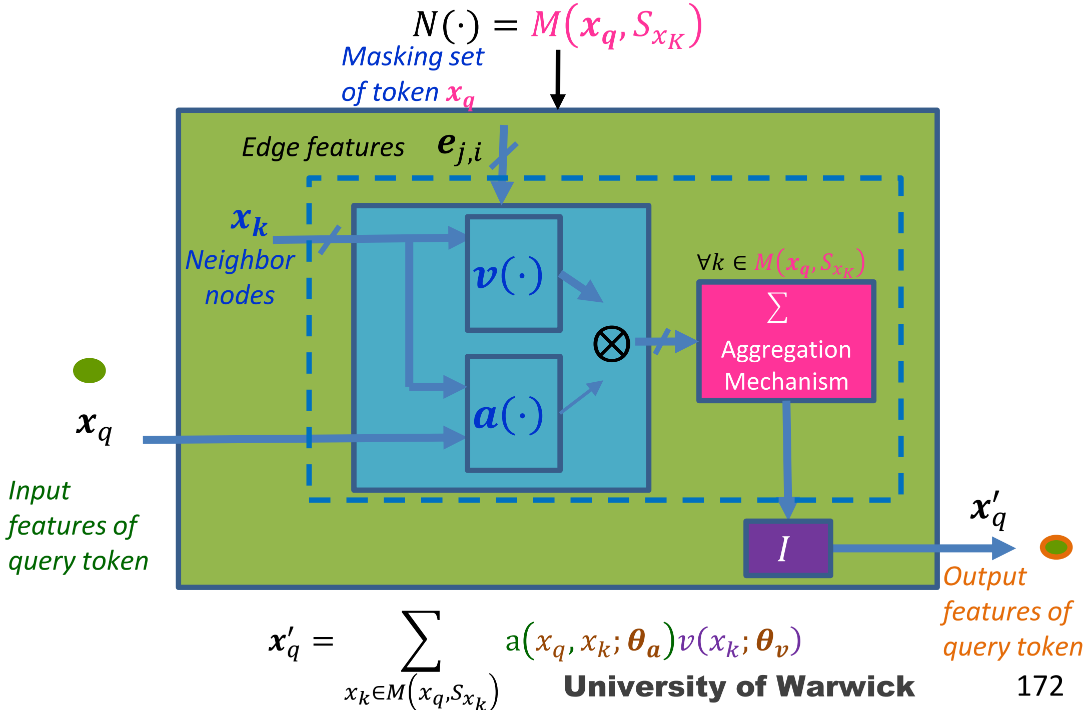
Assume

- We have a set of "nodes" S_{x_K} and for a given "query" node x_q, we have a masking set set $M(x_q, S_{x_K})$ (for simplicity assume $M(x_q, S_{x_K}) = S_{x_K}$)
- Each node is connected to all other nodes including itself (i.e., neighborhood $N(x_q) = M(x_q, S_{x_K})$) (Fully Connected Graph)

Now consider a specific graph neural network layer in which

- $\gamma^{(k)}(\mathrm{a}, \mathrm{b}) = \mathrm{b}$
- $\phi^{(l)}\left(x_i^{(l-1)}, x_j^{(l-1)}, e_{j,i}\right) = \mathrm{a}\left(x_i^{(l-1)}, x_j^{(l-1)}; \theta_a\right) v\left(x_j^{(l-1)}; \theta_v\right)$
- $\bigoplus_{j\in N(q)}(\cdot) = \sum_{x_k \in M(x_q, S_{x_K})}(\cdot)$

- Then the output of the GNN layer:
  - $x_i^{(l)} = \gamma^{(l)}\left(x_i^{(l-1)}, \oplus_{j\in N(i)}\phi^{(l)}\left(x_i^{(l-1)}, x_j^{(l-1)}, e_{j,i}\right)\right)$
- Becomes (with notation $x_i^{(l-1)} = x_q$, $x_j^{(l-1)} = x_k$ and $x_i^{(l)} = x'_q$)
  - $x'_q = \sum_{x_k \in M(x_q, S_{x_K})} \mathrm{a}(x_q, x_k; \theta_a) v(x_k; \theta_v)$
- Which is an attention layer (assuming position encoding is built into node features)
- **An attention layer is a special case of a GNN layer!**
  - **Attention scores can be viewed as pairwise weights of edges between nodes**



Reading on Graph Neural Networks

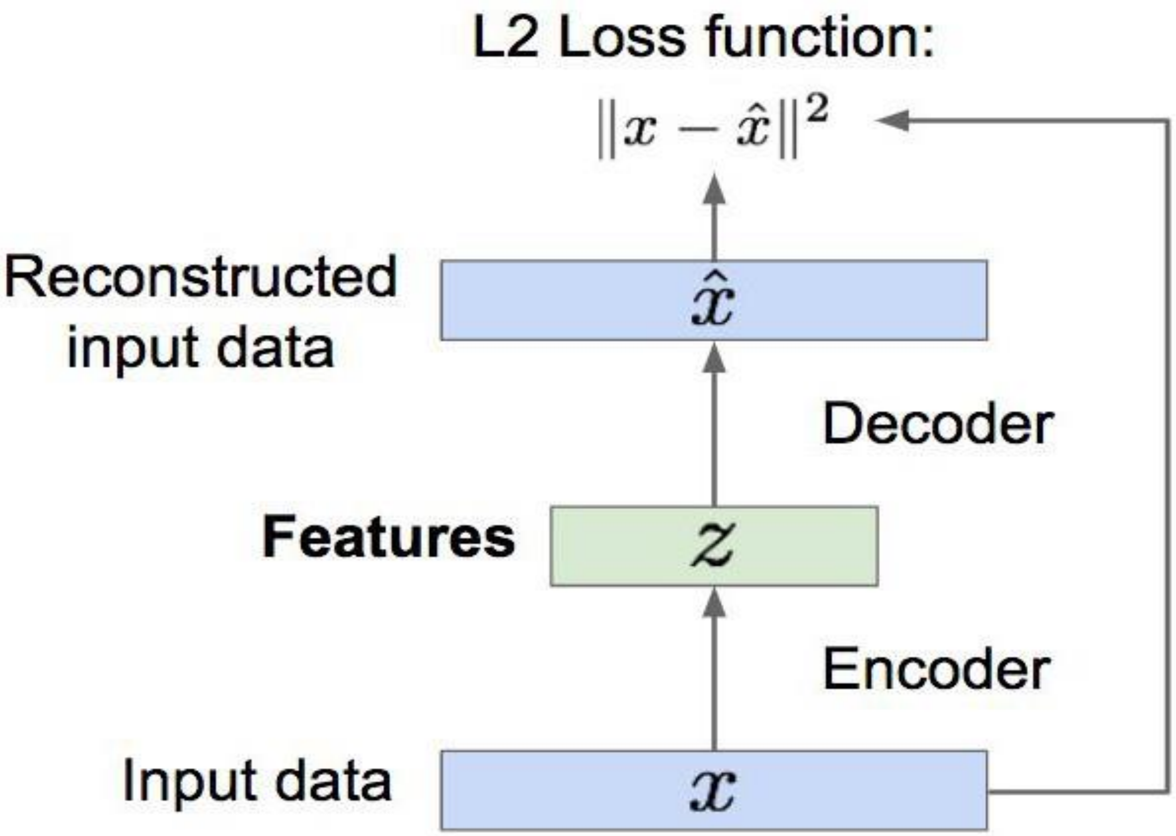
- Xu*, Keyulu, Weihua Hu*, Jure Leskovec, and Stefanie Jegelka. “How Powerful Are Graph Neural Networks?,” 2023. https://openreview.net/forum?id=ryGs6iA5Km.
- Kanatsoulis, Charilaos I., and Alejandro Ribeiro. “Graph Neural Networks Are More Powerful Than We Think.” arXiv, October 2, 2022. https://doi.org/10.48550/arXiv.2205.09801.
- Bronstein, Michael M., Joan Bruna, Taco Cohen, and Petar Veličković. “Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.” arXiv, May 2, 2021. https://doi.org/10.48550/arXiv.2104.13478.
- http://web.stanford.edu/class/cs224w/
- Libraries
  - PyTorch Geometric
  - DGL
  - Topological Neural Networks

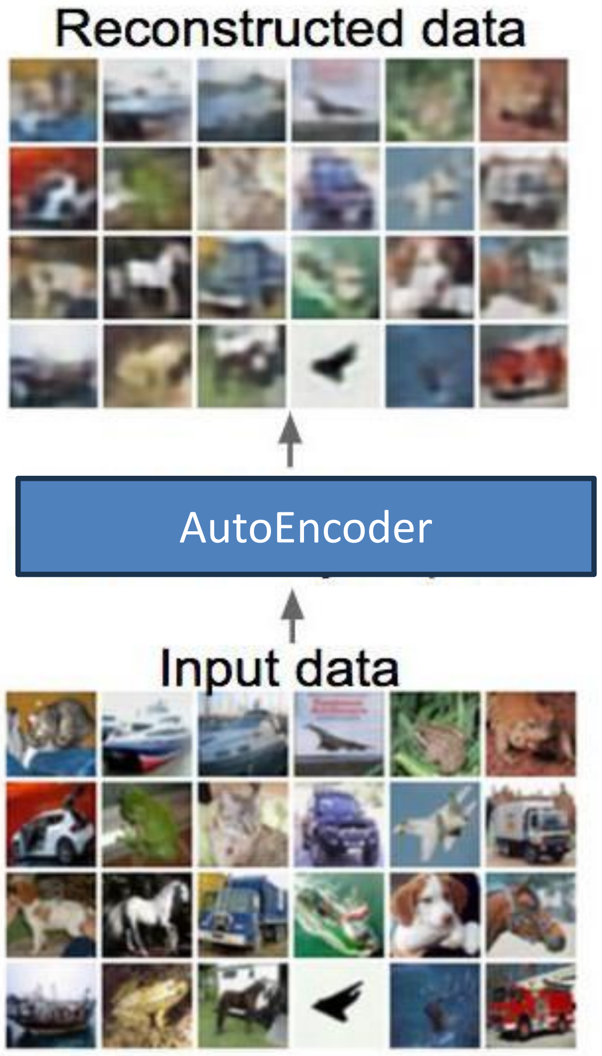
AUTOENCODERS

REO For Auto-Encoders

- Goal
  - Get an embedding (usually a compressed encoding) of a data sample such that the embedding can be used for reconstruction of data.
  - Used for dimensionality reduction, feature extraction, compression, visualization and generative learning
- Representation
  - Input: $\boldsymbol{x} \in R^{\boldsymbol{d}}$ Output: Reconstruction $\widehat{\boldsymbol{x}} = \boldsymbol{D}(\boldsymbol{E}(\boldsymbol{x};\boldsymbol{\theta_E});\boldsymbol{\theta_D})$
    - Encoder $\boldsymbol{E}(\boldsymbol{x};\boldsymbol{\theta_E})$: $R^d \to R^{d_E}$ (Usually $d_E < d$)
    - Decode $\boldsymbol{D}(\boldsymbol{x}';\boldsymbol{\theta_E})$:$R^{d_E} \to R^d$
- Evaluation:
  - Mean Square Error Loss (Other losses such as KL Divergence etc)
    - $\boldsymbol{min}_{\boldsymbol{\theta_D},\boldsymbol{\theta_E}} \frac{1}{N}\sum_{\boldsymbol{i}}\|\boldsymbol{x_i} - \widehat{\boldsymbol{x}}_{\boldsymbol{i}}\|^{\boldsymbol{2}} = \frac{1}{N}\sum_{\boldsymbol{i}}\|\boldsymbol{x_i} - \boldsymbol{D}(\boldsymbol{E}(\boldsymbol{x};\boldsymbol{\theta_E});\boldsymbol{\theta_D})\ \|^{\boldsymbol{2}}$
- Optimization

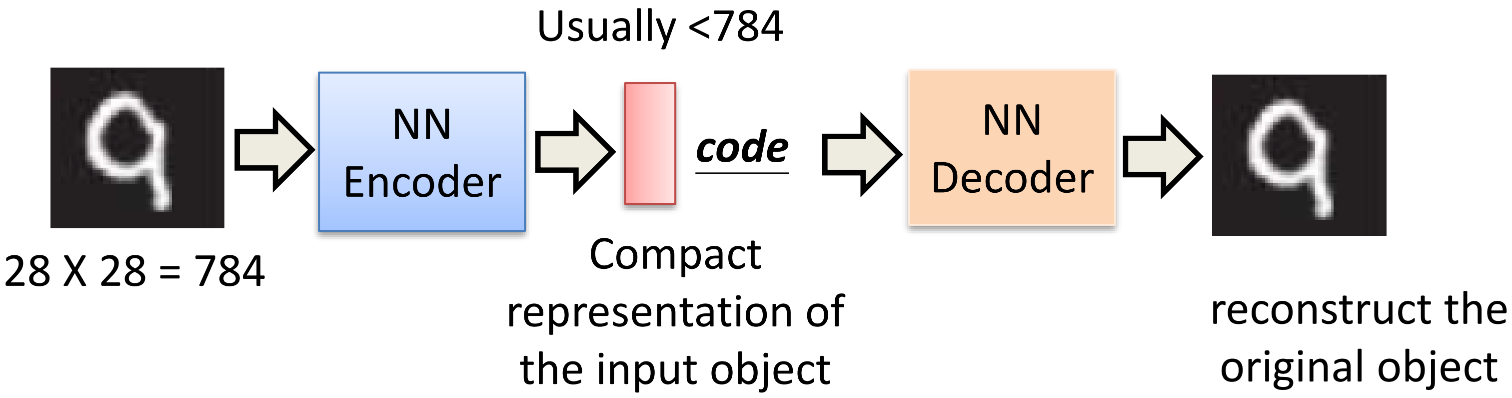
Unsupervised Learning - Autoencoders





Autoencoder

Unsupervised approach for learning a lower-dimensional feature representation from unlabelled training data



Usually <784

NN Encoder

code

NN Decoder

28 X 28 = 784

Compact representation of the input object

reconstruct the original object

Q: Why dimensionality reduction?

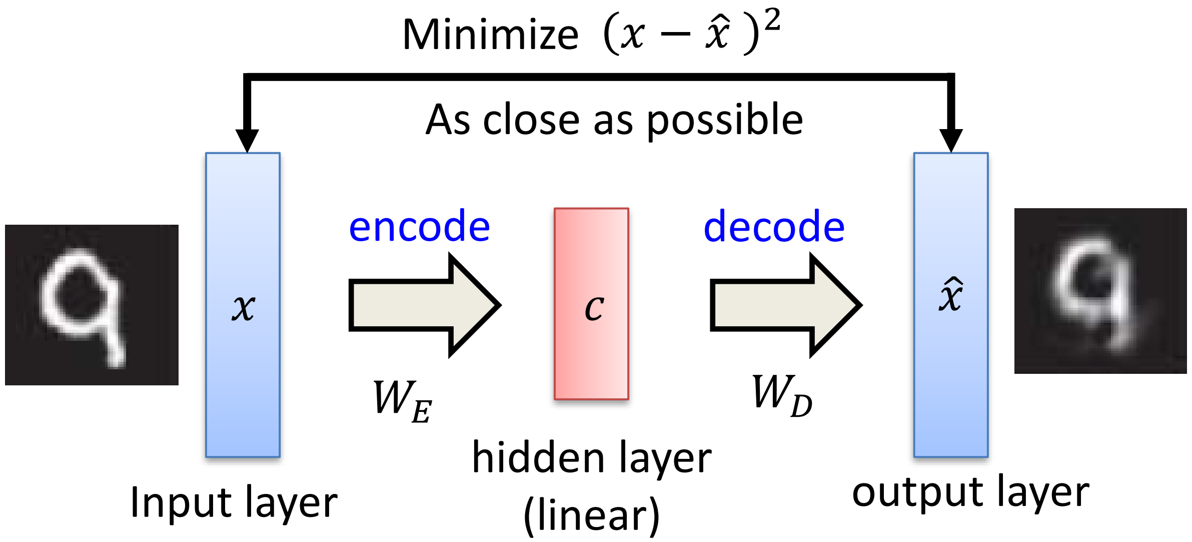
A: Want features to capture meaningful factors of variation in data

How to Train Autoencoders?

Train such that features can be used to reconstruct original data

“Autoencoding” – encoding itself

Equivalent to PCA*



Minimize $(x - \hat{x})^2$

As close as possible

Bottleneck later

Output of the hidden layer is the code

*Under the assumptions that the data is mean-centered and mean squared error is used as a loss function along with an orthogonality constraint $W_E W_D = I$

Refresher PCA: Reconstruction

- We know that $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x}$ (assuming $\boldsymbol{x}$ is centered) therefore

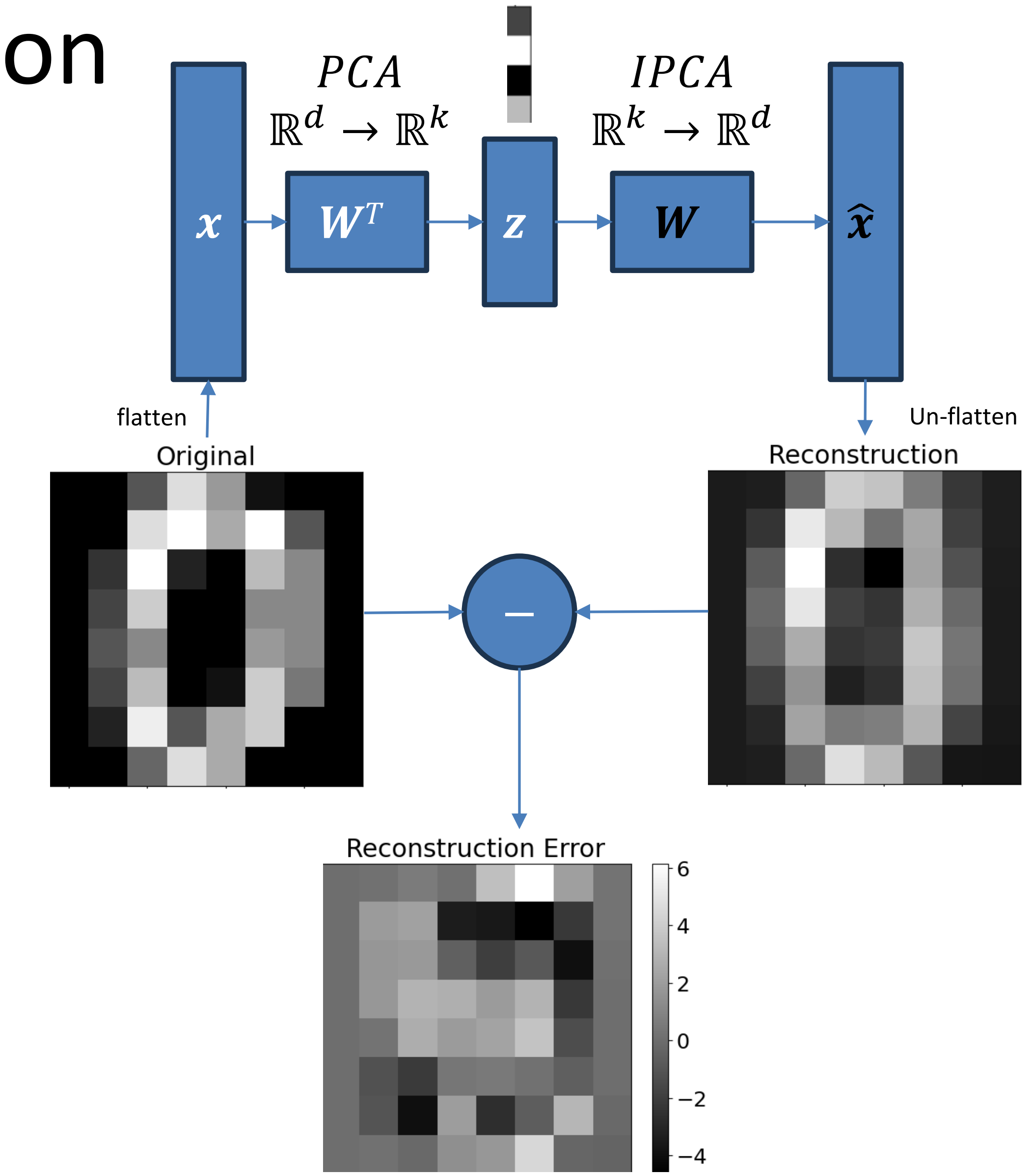
$$\widehat{\boldsymbol{x}} = (\boldsymbol{W}^T)^{-1}\boldsymbol{z}$$
$$\Rightarrow \widehat{\boldsymbol{x}} = \boldsymbol{W}_{(d\times k)}\boldsymbol{z} \quad \because \boldsymbol{W}\boldsymbol{W}^T = \boldsymbol{I}$$

- The reconstruction error is given by

$$E_{rec} = \sum_{i=1}^{N} \left\| \widehat{\boldsymbol{x}}^{\boldsymbol{i}} - \boldsymbol{x}^{\boldsymbol{i}} \right\|$$

- Another way of interpreting PCA is that it finds orthogonal direction vectors such that after projecting data onto to them, the reconstruction error is minimal.

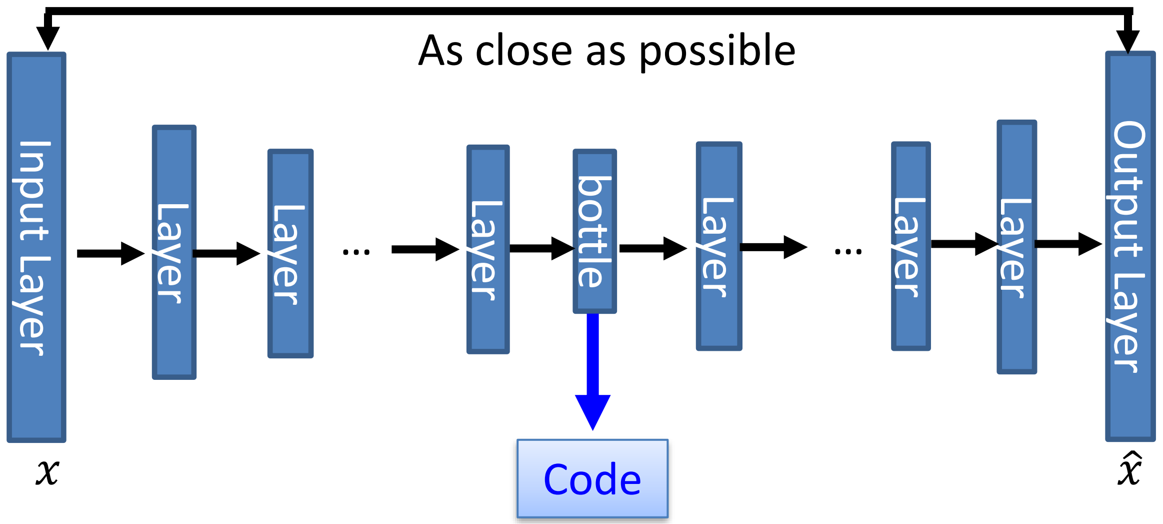
$$\min_{\boldsymbol{W}} \sum_{i=1}^{N} \left\| \widehat{\boldsymbol{x}}^{\boldsymbol{i}} - \boldsymbol{x}^{\boldsymbol{i}} \right\| \; \boldsymbol{s.th.} \boldsymbol{W}\boldsymbol{W}^T = \boldsymbol{I}$$



See Tutorial: https://github.com/foxtrotmike/PCA-Tutorial/blob/master/pca-lagrange.ipynb

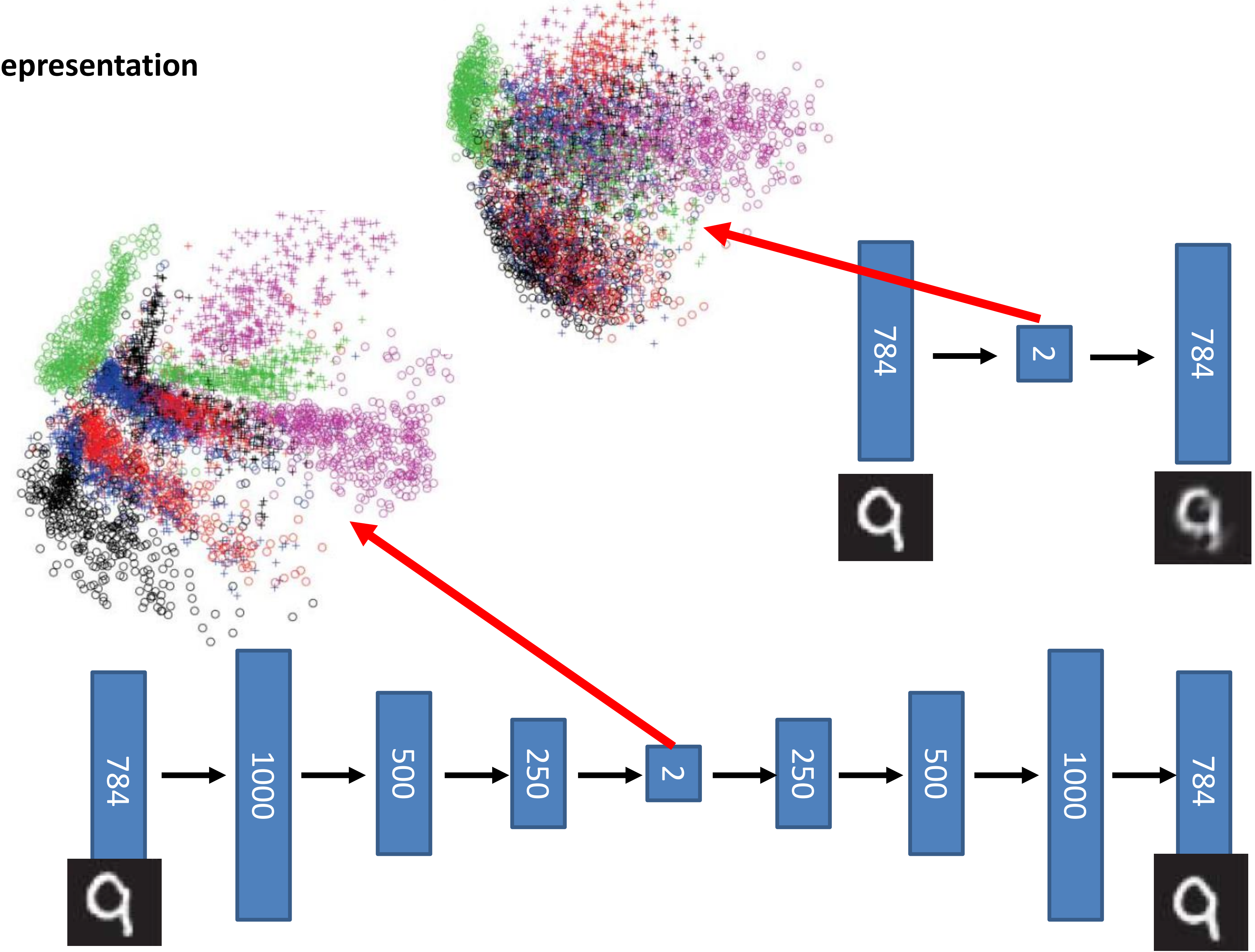
Deep Auto-encoder

- Of course, the auto-encoder can be deep



Reference: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507

"Latent Space" Representation



Tutorial Implementation: https://github.com/foxtrotmike/CS909/blob/master/autoenncoders.ipynb

Other types of autoencoders

- Vanilla Auto-encoder
- Denoising Auto-encoders
- Variational Auto-encoder (VAE)
- Vector-Quantized Variational Autoencoders (VQ-VAE)

Further notes: https://github.com/foxtrotmike/CS909/blob/master/autoenncoders.ipynb

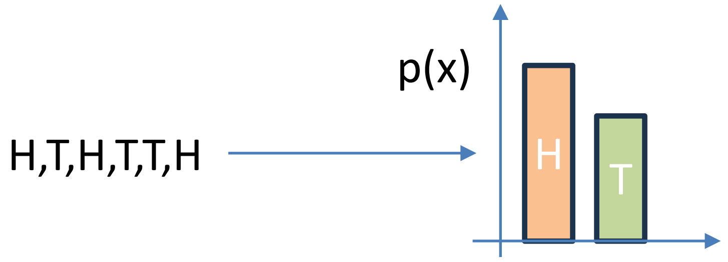
GENERATIVE MACHINE LEARNING

Creating noise from data is easy; creating data from noise is generative modeling.*

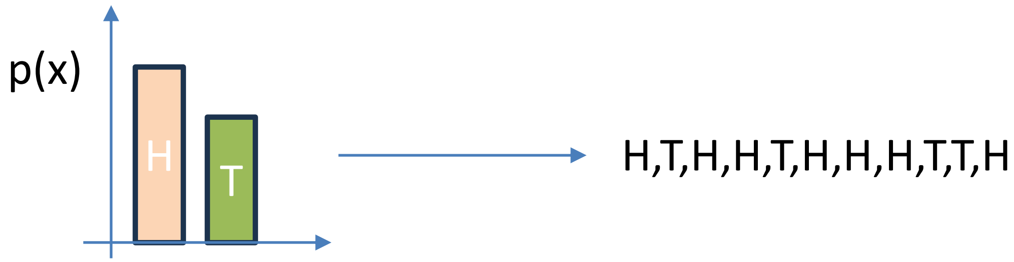
[*] Song, Yang, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. “Score-Based Generative Modeling through Stochastic Differential Equations.” arXiv, February 10, 2021. https://doi.org/10.48550/arXiv.2011.13456.

Background: Introduction to Sampling

- **Empirical distribution Modelling**: Making a distribution from observations (Density Estimation)
  - Example:
    - Observations: {H,T,H,T,H}
    - P(H) = 3/5 = 0.6, P(T) = 2/5 = 0.4
    - Shown as probability distribution (normalized histogram)

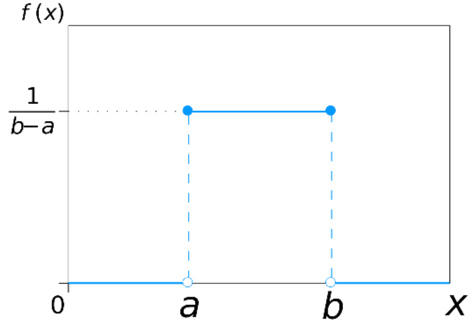


- **Sampling** from a distribution
  - Assume you are given a probability distribution p(x), then if you “sample” from it, you will be generating samples x which when observed will give you p(x)
  - Example
    - Given: P(H) = 0.6, P(T) = 0.4
    - Generated Samples: {H,T,H,T,H,T,H,H,T,H}



Background: Generating samples

- Can we **generate** samples of a target distribution using samples from a source distribution as input?



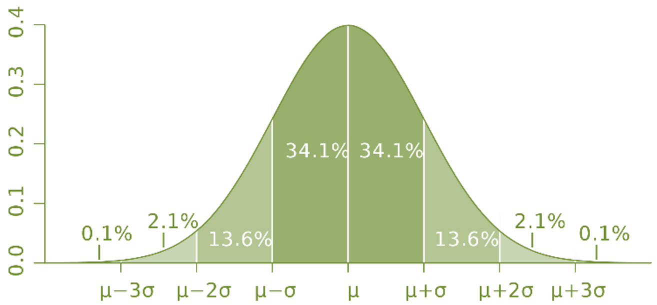
$x \sim U(a=0, b=1)$

X = np.random.rand(N)

$x \sim S(x)$

Generator

$z \sim T(z)$

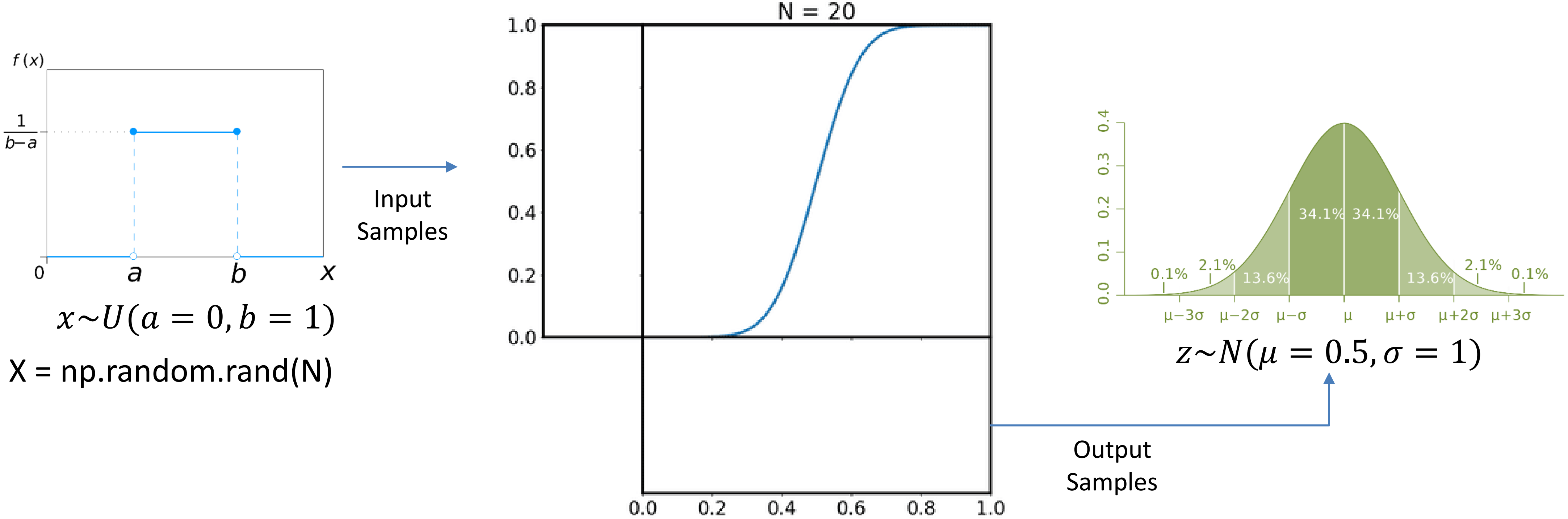


$z \sim N(\mu = 0.5, \sigma = 1)$

Z = np.random.randn(N)+0.5

Background: Generating samples

- We can use **inverse transform sampling**
  - – But that requires the knowledge of the formula for both probability distributions which may not be available for the target distribution



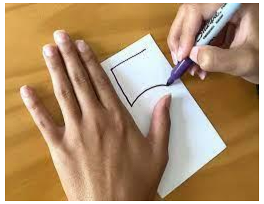
$x \sim U(a=0, b=1)$

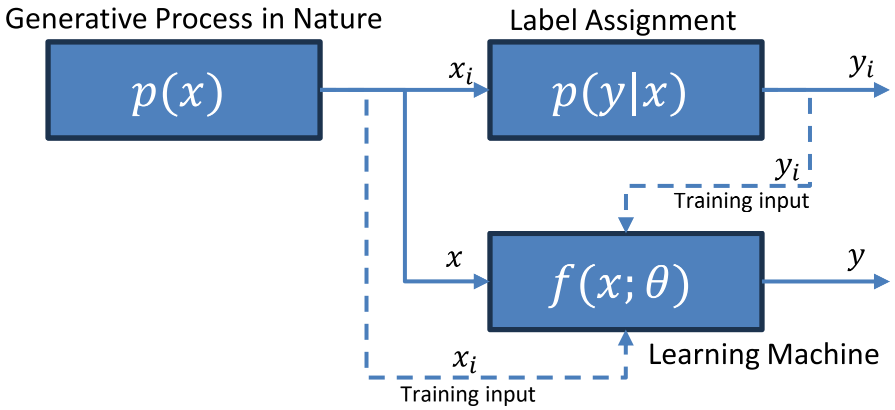
X = np.random.rand(N)

$z \sim N(\mu = 0.5, \sigma = 1)$

https://en.wikipedia.org/wiki/Inverse_transform_sampling

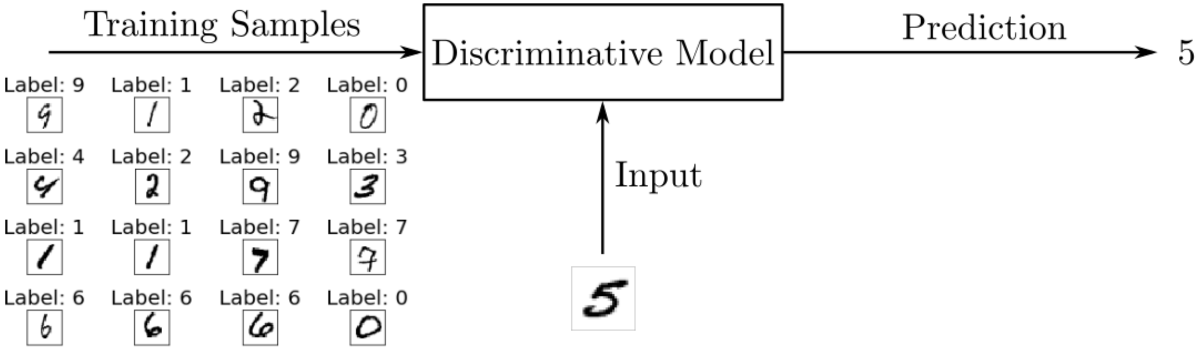
A generative look at Machine Learning





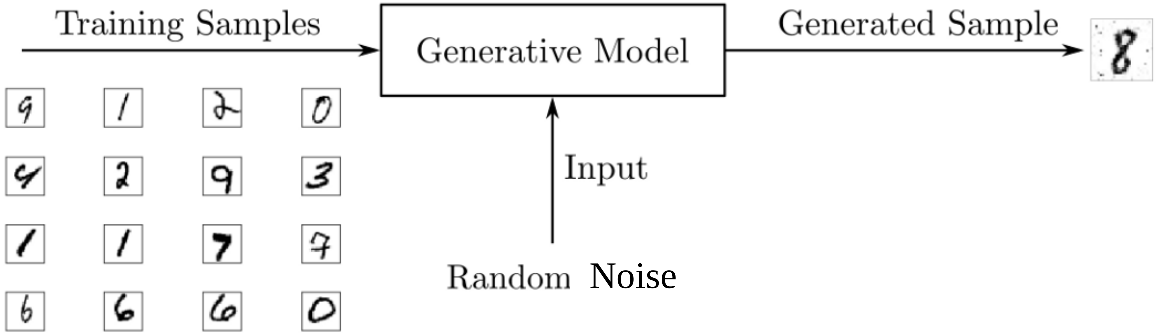
Fundamental aim of a discriminative model

Learn a model of p(y|x) from observations



Fundamental aim of a Generative Model

Learn a model of $p(x)$ or $p(x|y)$ from observations to generate samples from random noise input

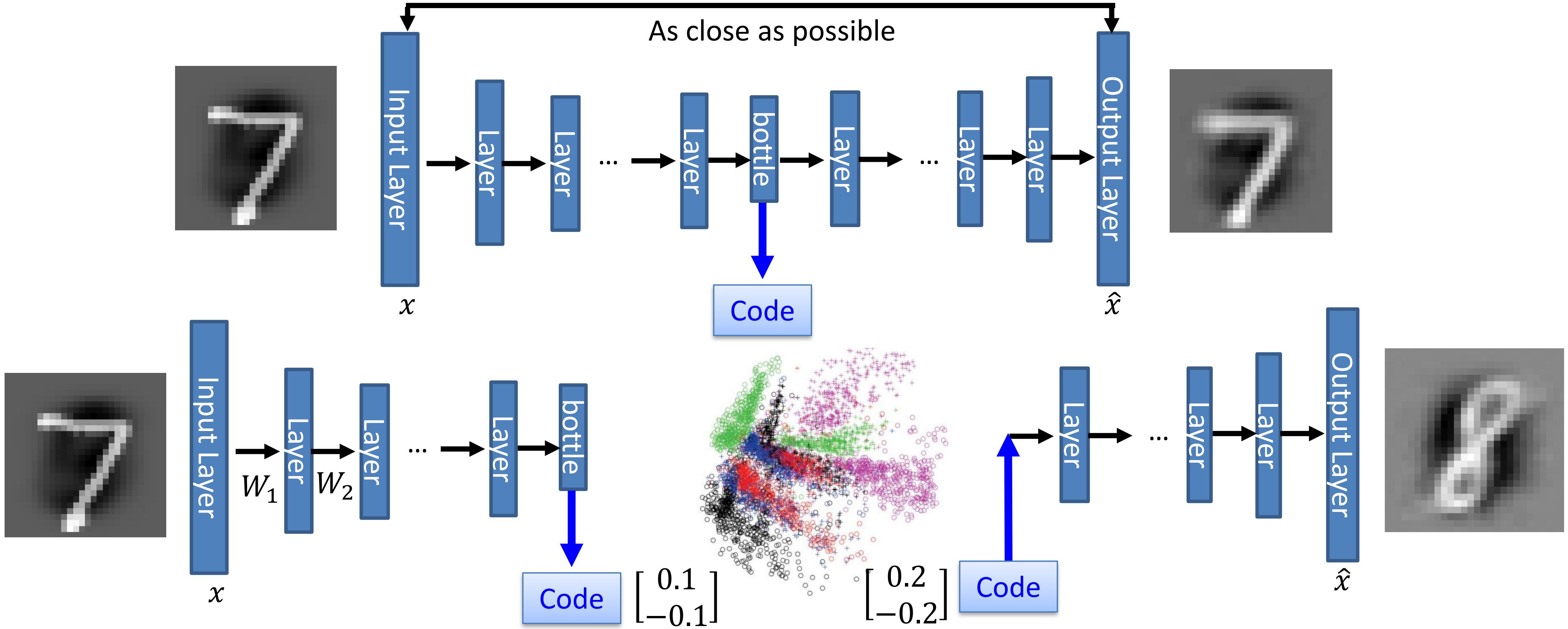


Lex Fridman. *Complete Statistical Theory of Learning (Vladimir Vapnik) | MIT Deep Learning Series*, 2020. https://www.youtube.com/watch?v=Ow25mjFjSmg.

Generating data with machine learning

- Can we generate examples that follow the same distribution as a given set of examples using noise as input?
- Sampling from the multi-dimensional distribution of data
- How?
  - Density Modelling
    - Modelling the Probability of observing a given point $p(x)$
    - Once I have an explicit or implicit $p(x)$, I can sample from that distribution to generate an example

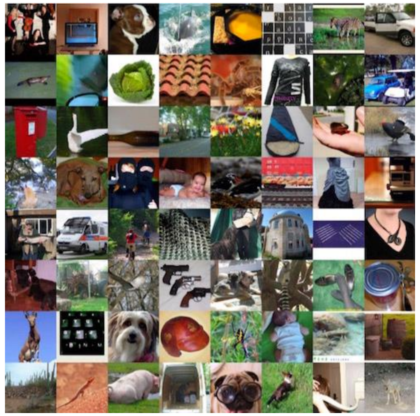
Generating Data with Autoencoders



Tutorial Implementation: https://github.com/foxtrotmike/CS909/blob/master/autoenncoders.ipynb

Generative Models

- Can we build a model to approximate a data distribution from given examples?



Real image (training data) ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Density estimation: a core problem in unsupervised learning

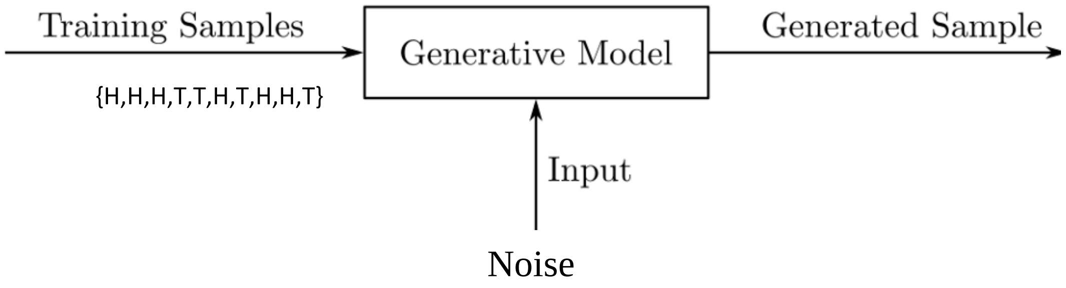
Several flavors:

- **Explicit density estimation**: explicitly define and solve for $p_{model}(x)$
  - Algorithms: Gaussian Mixture Models, Kernel Density Estimation, Variational Autoencoders
- **Implicit density estimation**: learn model that can sample from $p_{model}(x)$ w/o explicitly defining it
  - Algorithms: Vanilla autoencoder, Generative adversarial networks (GANs), Diffusion Models, Normalizing Flows

https://openai.com/blog/generative-models/

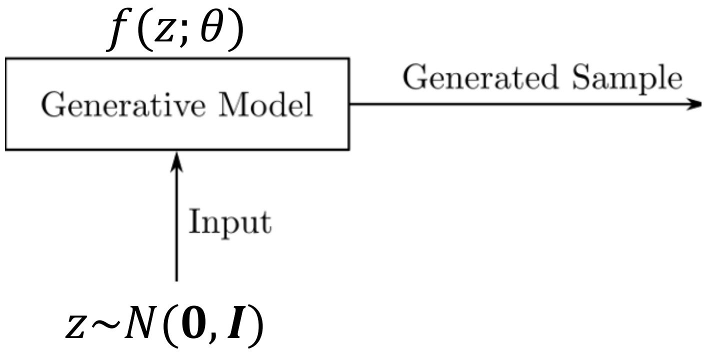
A Simple Generative Machine Learning Example

- Nature
  - A coin with p(x=H)=0.7 and p(x=T)=0.3
  - Generates data
- Given Data
  - {H,H,H,T,T,H,T,H,H,T}
- Goal of Generative Learning
  - Make a machine learning model that can generate data (heads or tails) that follows the same distribution as data from the real world or natural process.
  - The difference between the probability distributions of real and generated samples should be small

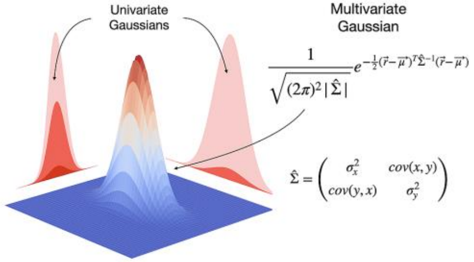


REO for Generative Models

- Goal
  - Given a set of real-world examples: $x \sim p(x)$. $p(x)$ is not explicitly known.
  - Learn parameters θ of the model $f(z;\theta)$ so that the examples generated by the model follow the same distribution as the real-world examples $x \sim p(x)$

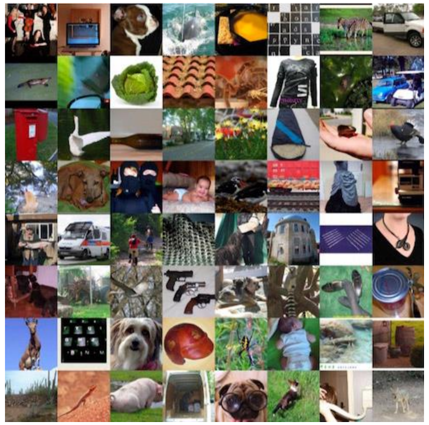


- Representation: $x = f(z;\theta)$ with $z \sim Noise$
  - Let's denote the distribution of examples generated by this model as $p_\theta(x)$.
  - Note that the model may not have an explicit internal formula for this distribution.



- Evaluation:
  - Differences between the probability distribution of x in nature $p(x)$ and of the generated samples $p_\theta(x)$ from $f(z;\theta)$
    - That is, if I sample from $p(x)$ or if I sample from$p_\theta(x)$, the real and generated samples are similar

- Optimization
  - Use gradient descent to optimize for θ

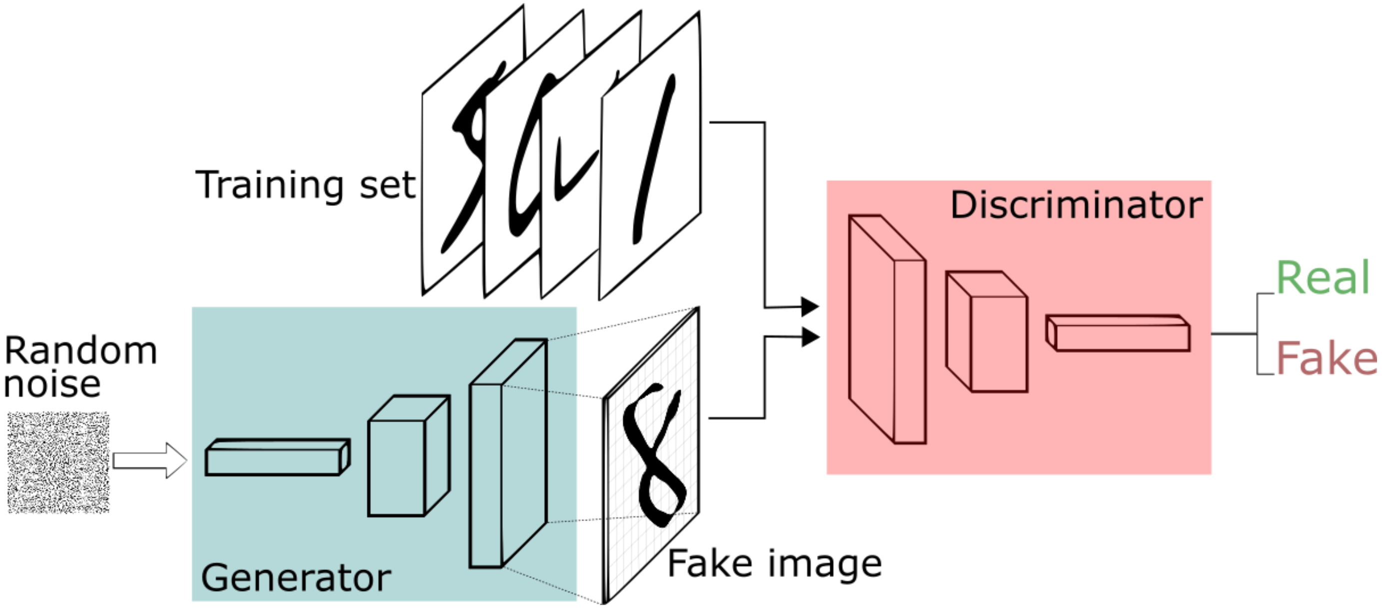


Real image (training data) ~ $p(x)$
$p(x)$ is not given.

Generated samples ~ $p_\theta(x)$
$p_\theta(x)$ may be implicit.

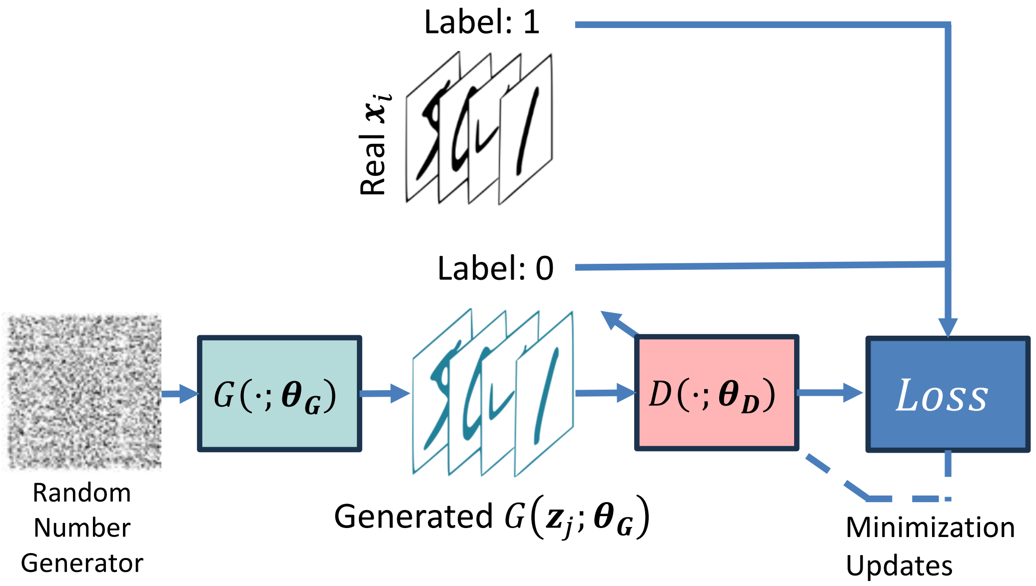
Generative Adversarial Networks

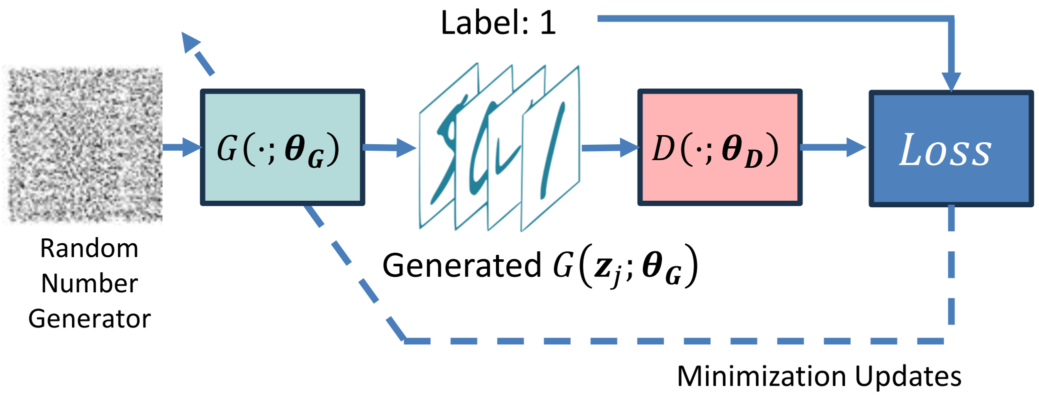
- Use “Adversarial Training” to train a generator and discriminator simultaneously
- Generator: Generate samples from noise
- Discriminator: Detect “fake” or generated samples



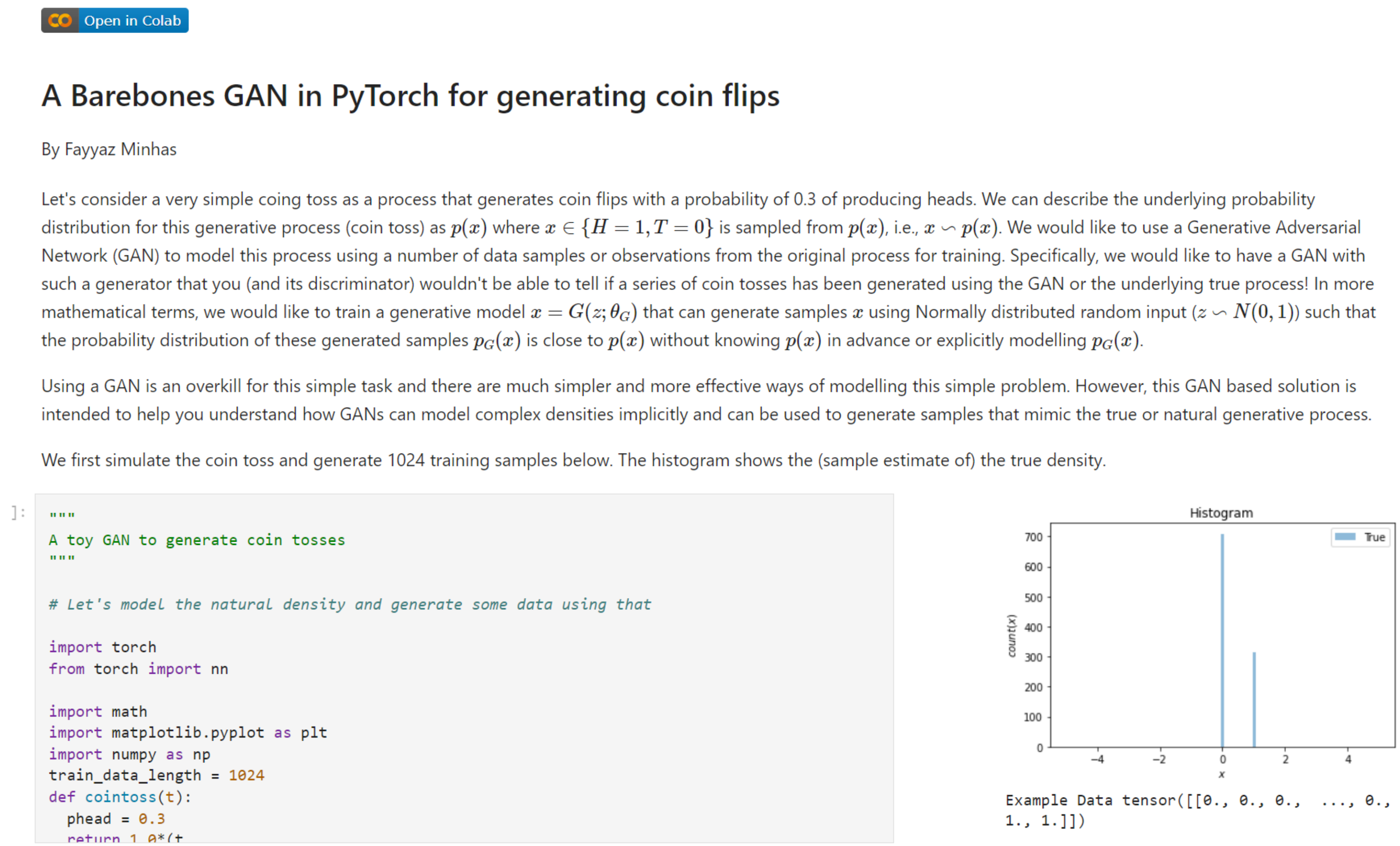
Adversarial Training in a GAN

- GAN Training the goal is to:
  - **Train the discriminator** to be good at detecting fakes
    - Simple classification: Discriminator should produce 1 for real and 0 for generated
      - $\min_{\boldsymbol{\theta_D}} \sum_{x_i \in R} l(D(\boldsymbol{x}_i;\boldsymbol{\theta_D}),1) + \sum_{z_j \sim N} l(D(G(\boldsymbol{z}_j;\boldsymbol{\theta_G});\boldsymbol{\theta_D}),0)$
  - **Train the generator** to be so good that the discriminator labels generated samples as “Real”
    - The generator exploits the discriminator’s ability or knowledge to distinguish between real and generated samples to its advantage
    - The generator is optimized such that the discriminator produces 1 for generated examples
      - $\min_{\boldsymbol{\theta_G}} \sum_{\boldsymbol{z}_j \sim N} l(D(G(\boldsymbol{z}_j;\boldsymbol{\theta_G});\boldsymbol{\theta_D}),1)$
      - OR equivalently, the generator is optimized such that the discriminator generates errors in classifying generated examples (note the max below)
      - $\max_{\boldsymbol{\theta_G}} \sum_{\boldsymbol{z}_j \sim N} l(D(G(z_j;\boldsymbol{\theta_G});\boldsymbol{\theta_D}),0)$
  - Can also add additional loss terms for quality/realism etc.





GAN Tutorial



A Barebones GAN in PyTorch for generating coin flips

By Fayyaz Minhas

Let's consider a very simple coing toss as a process that generates coin flips with a probability of 0.3 of producing heads. We can describe the underlying probability distribution for this generative process (coin toss) as $p(x)$ where $x \in \{H = 1, T = 0\}$ is sampled from $p(x)$, i.e., $x \backsim p(x)$. We would like to use a Generative Adversarial Network (GAN) to model this process using a number of data samples or observations from the original process for training. Specifically, we would like to have a GAN with such a generator that you (and its discriminator) wouldn't be able to tell if a series of coin tosses has been generated using the GAN or the underlying true process! In more mathematical terms, we would like to train a generative model $x = G(z;\theta_G)$ that can generate samples x using Normally distributed random input ($z \backsim N(0,1)$) such that the probability distribution of these generated samples $p_G(x)$ is close to $p(x)$ without knowing $p(x)$ in advance or explicitly modelling $p_G(x)$.

Using a GAN is an overkill for this simple task and there are much simpler and more effective ways of modelling this simple problem. However, this GAN based solution is intended to help you understand how GANs can model complex densities implicitly and can be used to generate samples that mimic the true or natural generative process.

We first simulate the coin toss and generate 1024 training samples below. The histogram shows the (sample estimate of) the true density.

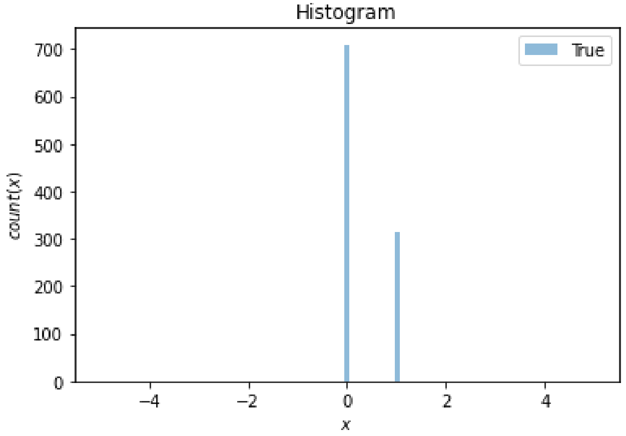
```
"""
A toy GAN to generate coin tosses
"""

# Let's model the natural density and generate some data using that

import torch
from torch import nn

import math
import matplotlib.pyplot as plt
import numpy as np
train_data_length = 1024
def cointoss(t):
  phead = 0.3
  return 1.0*(t
```

```
Example Data tensor([[0., 0., 0.,  ..., 0.,
1., 1.]])
```

https://github.com/foxtrotmike/CS909/blob/master/simpleGAN.ipynb

How to go from generating coin flips to images?

- Assume you are given B&W images for training a GAN to generate more images like that.
- Let’s look at a single pixel location in each image
  - We have a distribution of pixel values across all images at that location
    - We would like our GAN to generate data according to that distribution at that pixel location
    - Naïve idea: Have multiple GANs – one for each pixel location
      - Assumes each pixel is independent of the other
      - Computationally intensive
    - We can train a single GAN to generate a multi-dimensional probability distribution by using a multi-output generator.

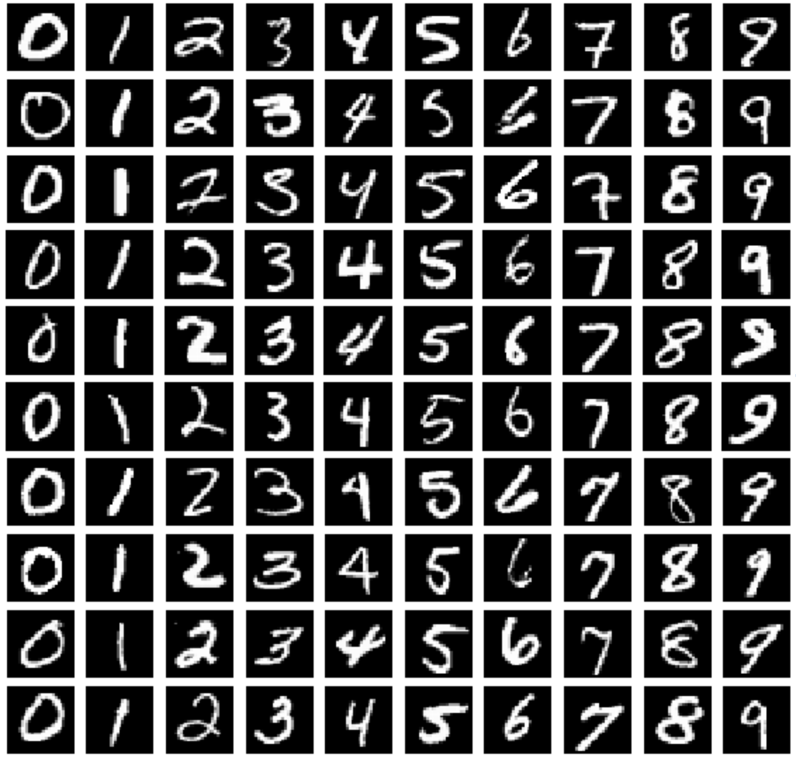


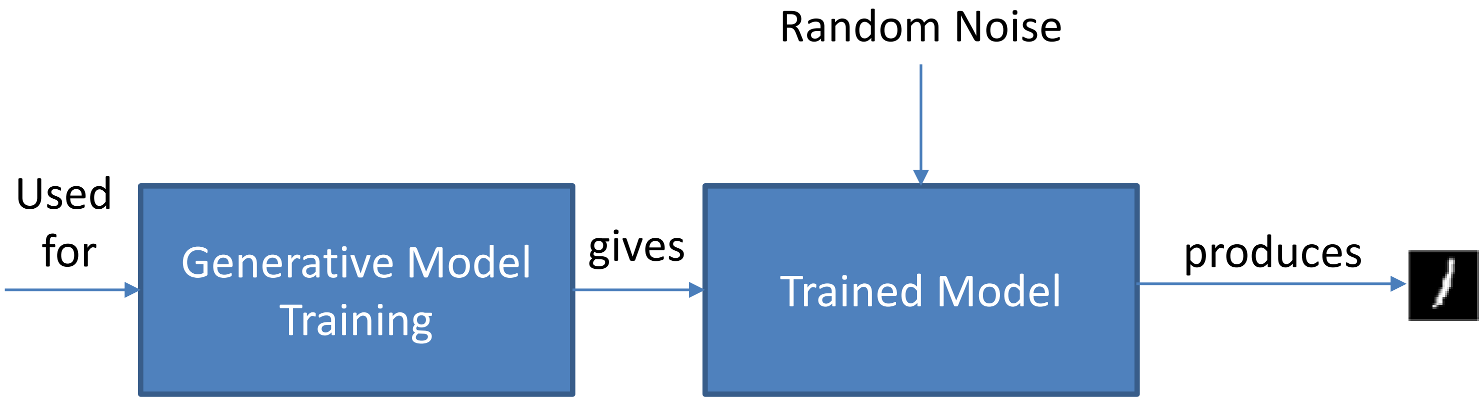
Performance Assessment of Generative Models

Goal	Metrics
Measure difference in probability distribution of generated and real samples	Earth Mover Distances Maximum Mean Discrepancy Kernel Inception Distance (KID) Wasserstein Distance
Diversity: Evaluate whether the model can generate a wide variety of outputs:	Diversity Score Mode Score
Coverage: Measure how well the generated samples cover the variety of the dataset	Coverage Score
Stability and Robustness:	Consistency of good results Adversarial robustness measures
Quality	Inception score Fréchet Inception Distance and KID Structural Similarity Index Measure Learned Perceptual Image Patch Similarity (LPIPS)
Task Specific metrics	NLP: BLEU, ROUGE Drug Discovery: Quantitative Structure-Activity Relationship (QSAR) Metrics Subjective Assessment

Unconditional vs Conditional Generation

- Unconditional Generative Modelling
  - Simple model the probability distribution of the data p(x)
    - Example: Generating images without paying any regard to the digit





Used for

Generative Model Training

gives

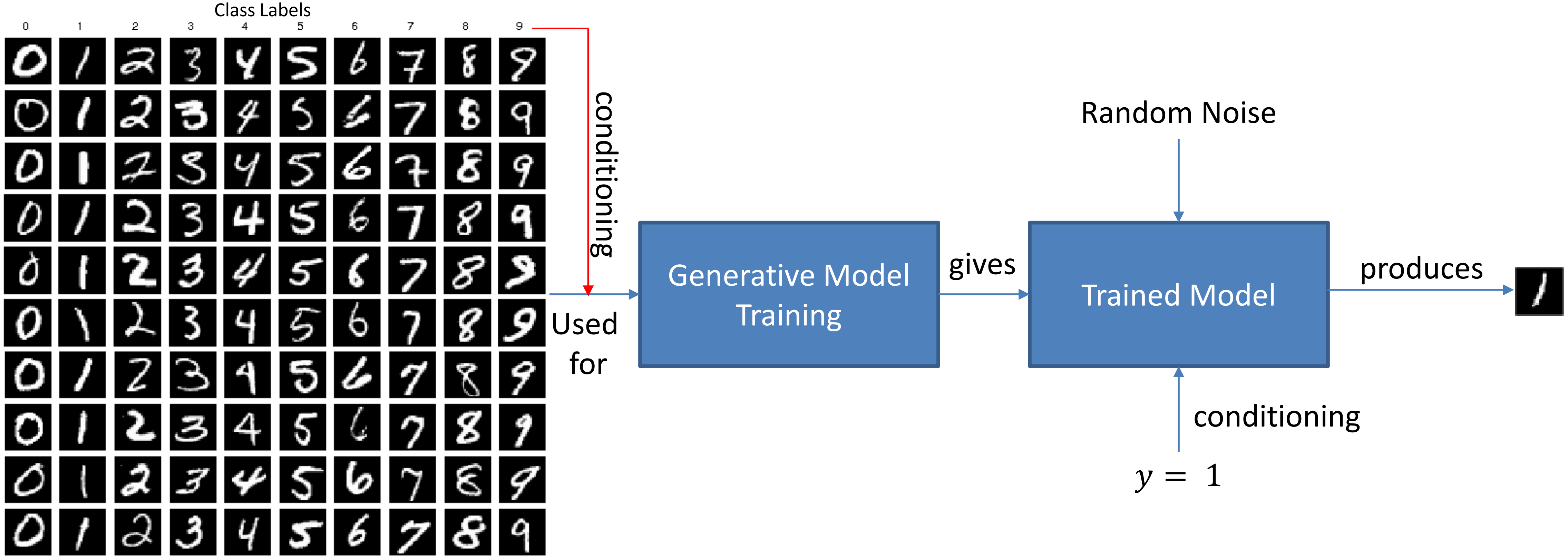
Random Noise

Trained Model

produces

Unconditional vs Conditional Generation

- Conditional Generative Modelling
  - Model the distribution p(x|y) of data x conditioned on a variable y
    - Example: Generating images for a given digit



GANs Applications

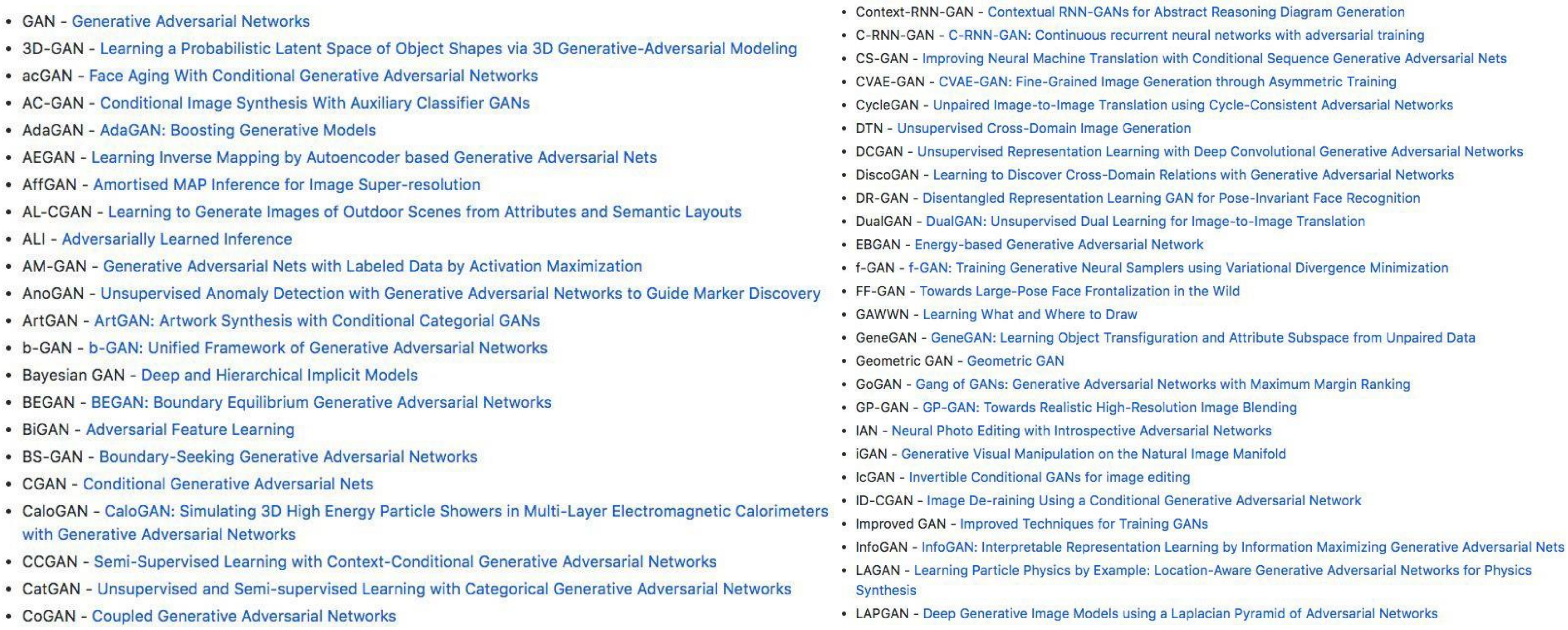
- GANs have some impressive applications
  - Synthetic Image Generation
  - Speech Generation
  - Image to Image Translation
  - Style Transfer
  - Deep Fakes

Barebones GAN
https://github.com/foxtrotmike/CS909/blob/master/simpleGAN.ipynb

Raevskiy, Mikhail. "Write Your First Generative Adversarial Network Model on PyTorch." Medium, August 31, 2020. https://medium.com/dev-genius/write-your-first-generative-adversarial-network-model-on-pytorch-7dc0c7c892c7.

Ian Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. https://arxiv.org/abs/1701.00160
https://github.com/eriklindernoren/PyTorch-GAN
https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/
https://affinelayer.com/pixsrv/

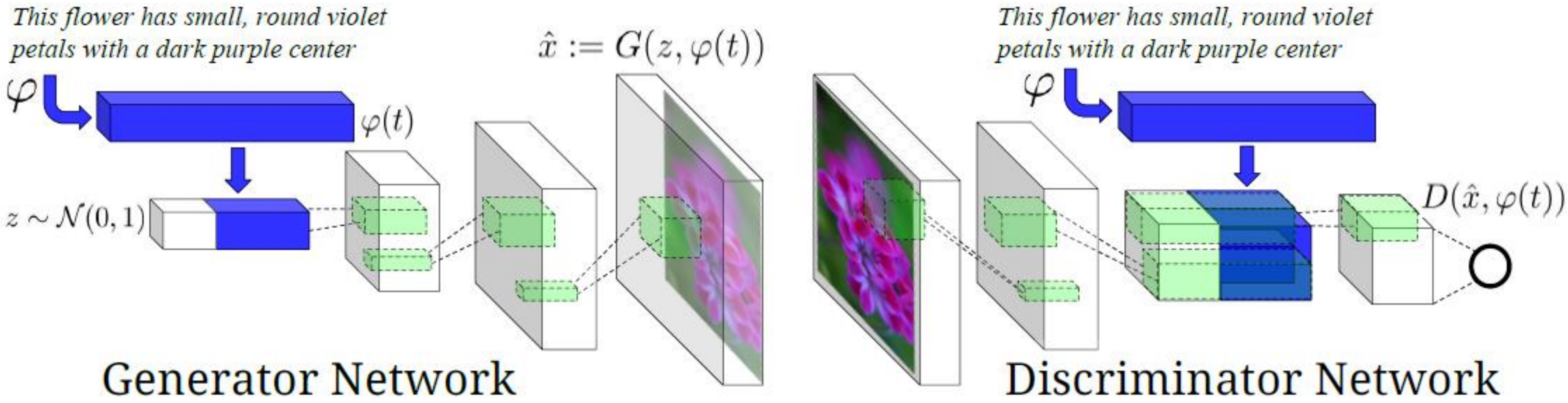
The GAN Zoo



- GAN - Generative Adversarial Networks
- 3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- acGAN - Face Aging With Conditional Generative Adversarial Networks
- AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
- AdaGAN - AdaGAN: Boosting Generative Models
- AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- AffGAN - Amortised MAP Inference for Image Super-resolution
- AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- ALI - Adversarially Learned Inference
- AM-GAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
- AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorial GANs
- b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
- Bayesian GAN - Deep and Hierarchical Implicit Models
- BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- BiGAN - Adversarial Feature Learning
- BS-GAN - Boundary-Seeking Generative Adversarial Networks
- CGAN - Conditional Generative Adversarial Nets
- CaloGAN - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- CoGAN - Coupled Generative Adversarial Networks
- Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- CS-GAN - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- DTN - Unsupervised Cross-Domain Image Generation
- DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- EBGAN - Energy-based Generative Adversarial Network
- f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- FF-GAN - Towards Large-Pose Face Frontalization in the Wild
- GAWWN - Learning What and Where to Draw
- GeneGAN - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- Geometric GAN - Geometric GAN
- GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
- IAN - Neural Photo Editing with Introspective Adversarial Networks
- iGAN - Generative Visual Manipulation on the Natural Image Manifold
- IcGAN - Invertible Conditional GANs for image editing
- ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
- Improved GAN - Improved Techniques for Training GANs
- InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- LAGAN - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

https://github.com/hindupuravinash/the-gan-zoo

Text-to-Image Synthesis

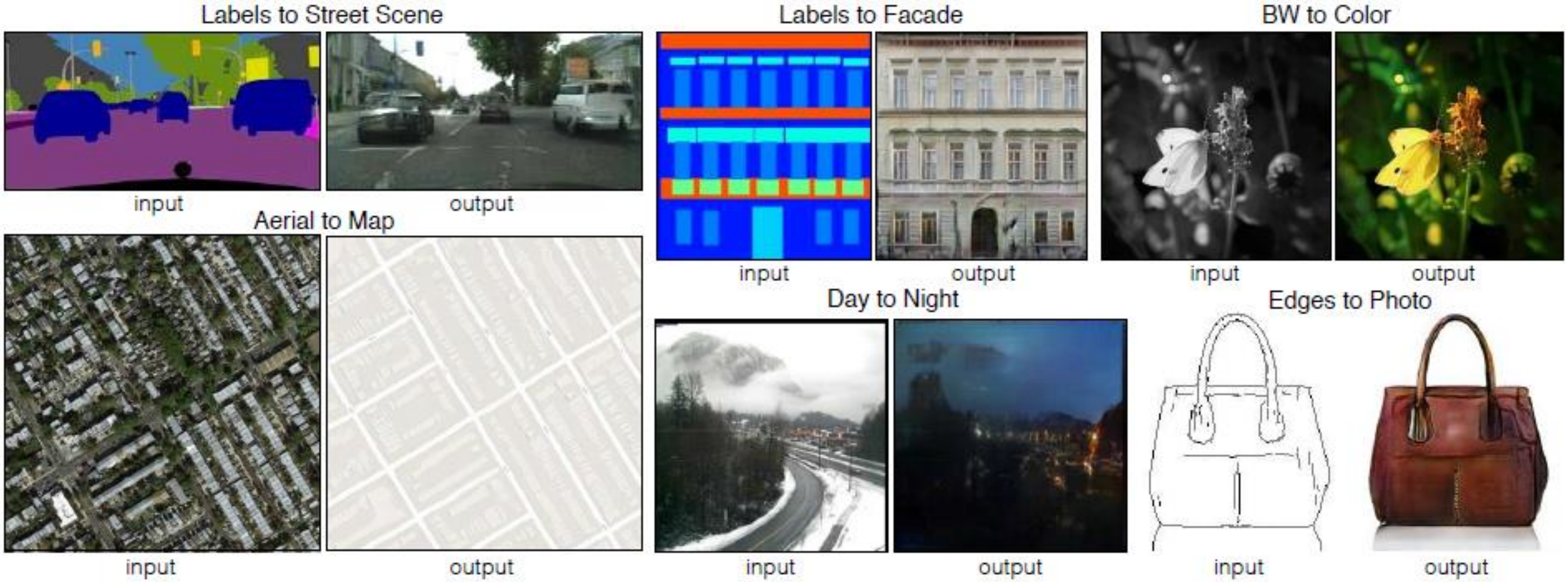


S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, H. Lee, “Generative Adversarial Text-to-Image Synthesis”, ICML 2016
H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, D. Metaxas, “StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks”, arXiv prepring, 2016
S. Reed, Z. Akata, S. Mohan, S. Tenka, B. Schiele, H. Lee, “Learning What and Where to Draw”, NIPS 2016

Text to Image – Results

Caption	**Image**
a pitcher is about to throw the ball to the batter	
a group of people on skis stand in the snow	
a man in a wet suit riding a surfboard on a wave	

Image-to-image Translation



P. Isola, J.-Y. Zhu, T. Zhou, A.A. Efros, “Image-to-Image Translation with Conditional Adversarial Networks”, arXiv preprint, 2016

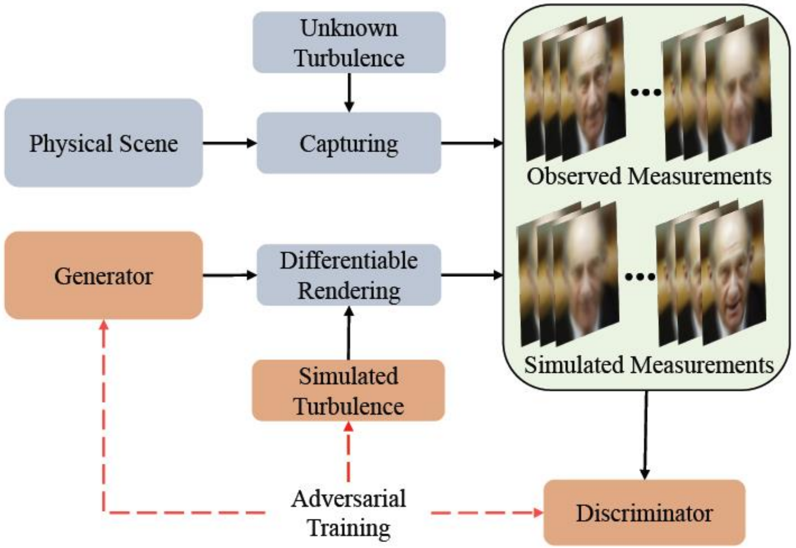
Unpaired Transformation – Cycle GAN, Disco GAN

Transform an object from one domain to another ***without paired data***

Domain X

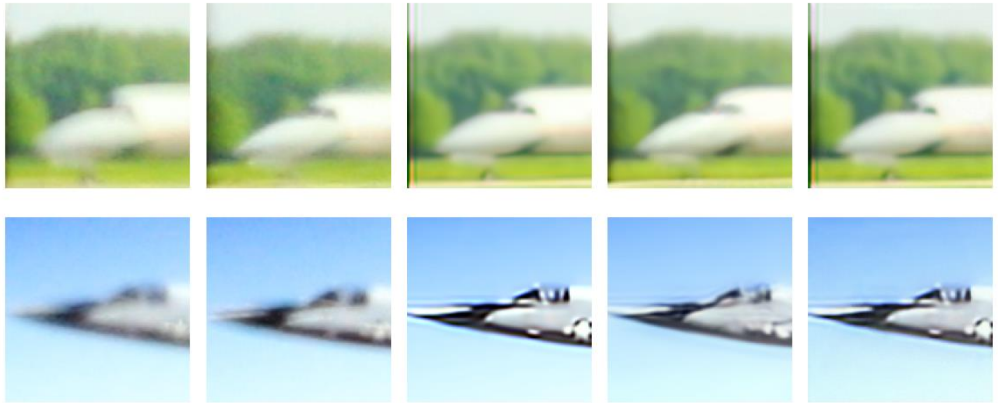
Domain Y

TurbuGAN



(a) Typical Observation (b) Lucky Frame

(c) Mao et al. (d) Ours

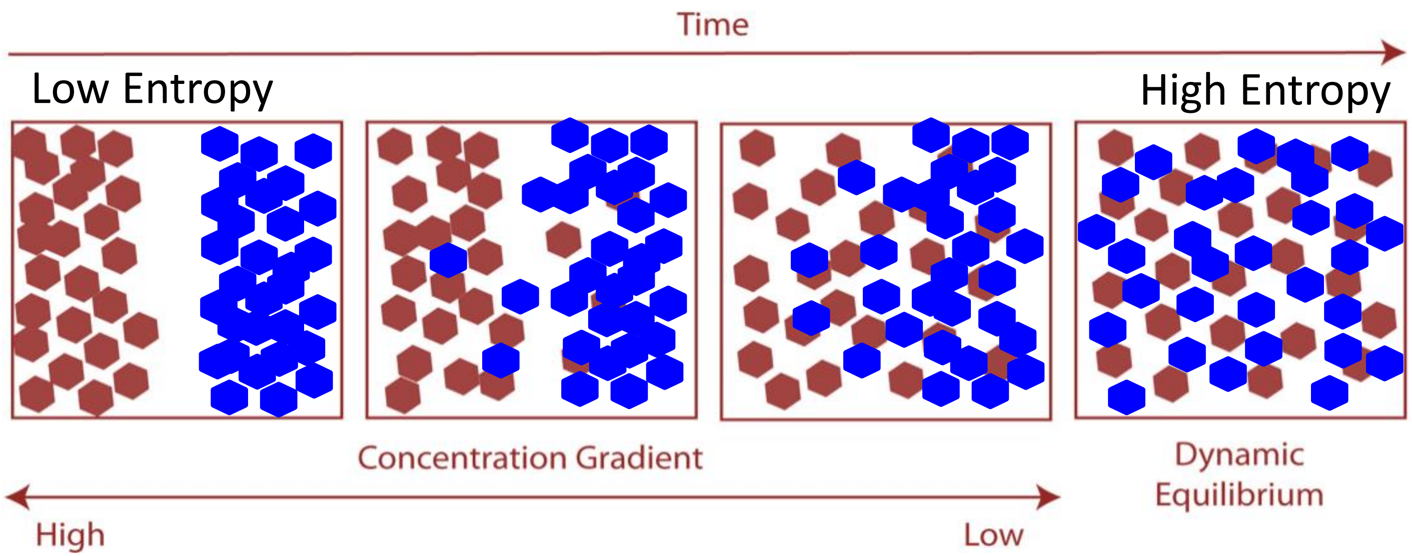


(a) Typical (b) Lucky (c) Mao et al. (d) Ours-IN (e) Ours-DIP

Feng, Brandon Yushan, Mingyang Xie, and Christopher A. Metzler. “TurbuGAN: An Adversarial Learning Approach to Spatially-Varying Multiframe Blind Deconvolution with Applications to Imaging Through Turbulence.” arXiv, August 22, 2022. https://doi.org/10.48550/arXiv.2203.06764.

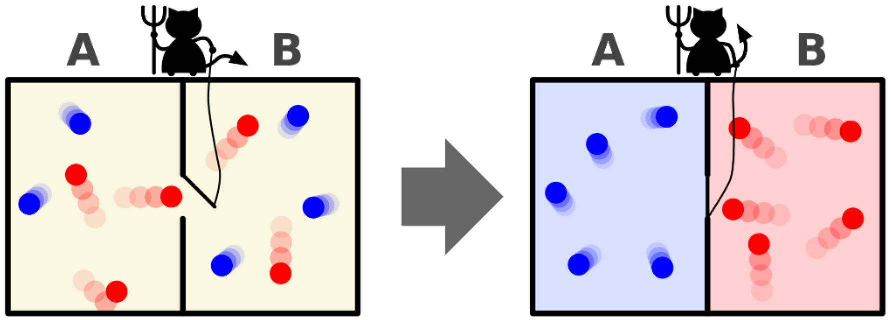
Diffusion Models

- What is diffusion?

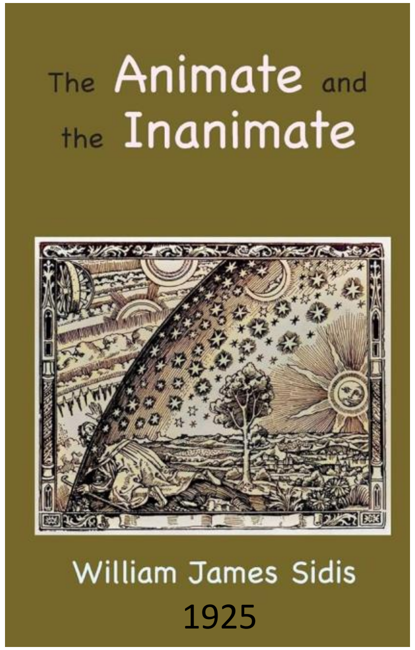


- Can we learn to reverse it?

Maxwell's Demon



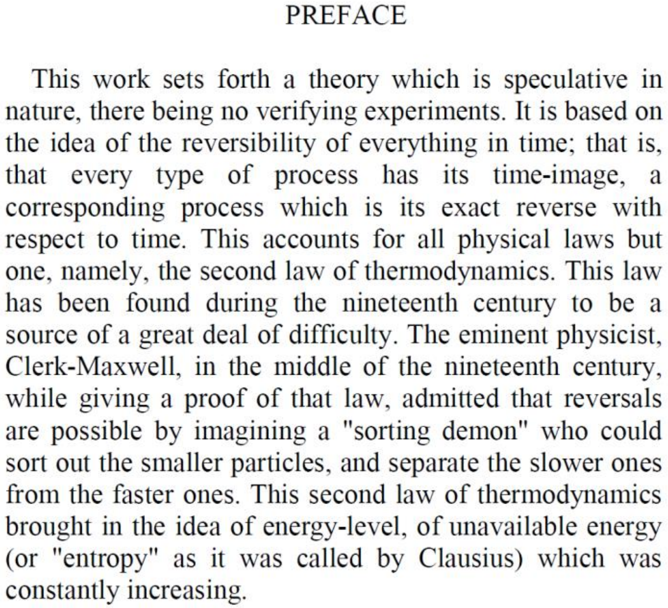
https://en.wikipedia.org/wiki/Maxwell%27s_demon



The Animate and the Inanimate

William James Sidis

1925

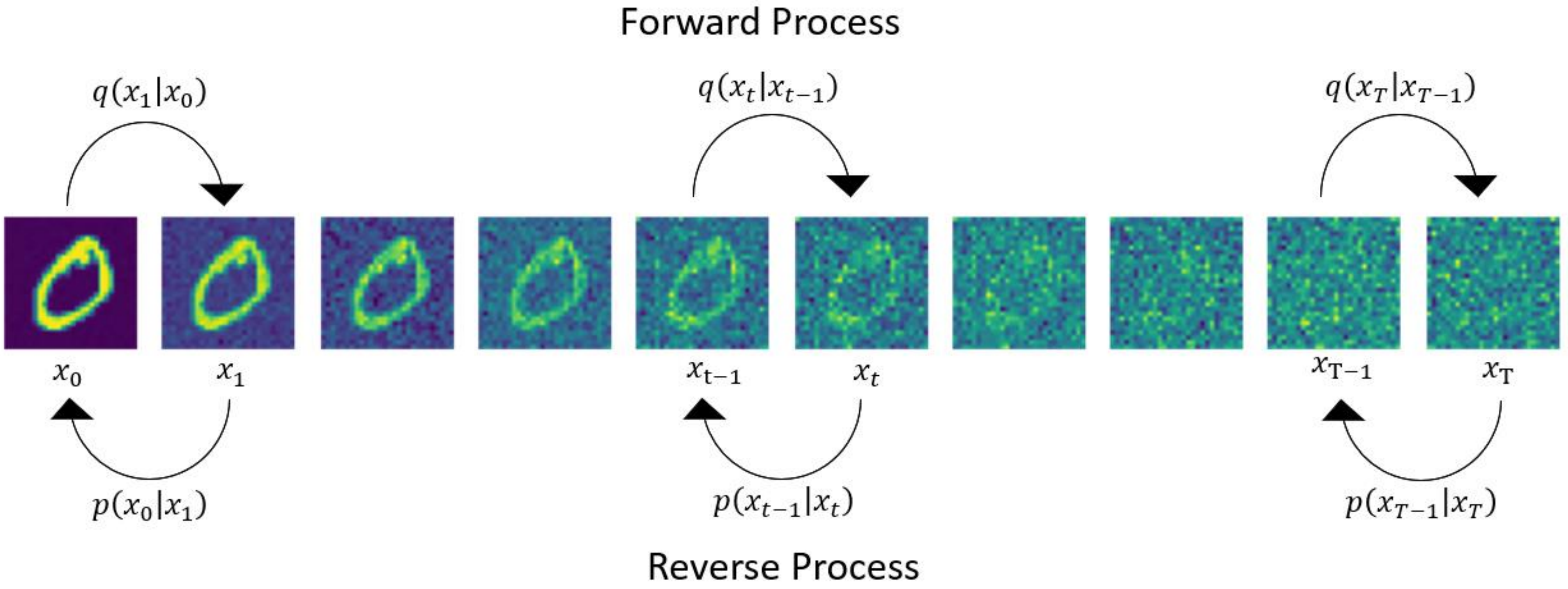


PREFACE

This work sets forth a theory which is speculative in nature, there being no verifying experiments. It is based on the idea of the reversibility of everything in time; that is, that every type of process has its time-image, a corresponding process which is its exact reverse with respect to time. This accounts for all physical laws but one, namely, the second law of thermodynamics. This law has been found during the nineteenth century to be a source of a great deal of difficulty. The eminent physicist, Clerk-Maxwell, in the middle of the nineteenth century, while giving a proof of that law, admitted that reversals are possible by imagining a "sorting demon" who could sort out the smaller particles, and separate the slower ones from the faster ones. This second law of thermodynamics brought in the idea of energy-level, of unavailable energy (or "entropy" as it was called by Clausius) which was constantly increasing.

Diffusion Models

- Main idea: Learn to reverse a “diffusion” process



[Tutorial: https://github.com/wgrgwrght/Simple-Diffusion/blob/main/SimpleDiffusion.ipynb](https://github.com/wgrgwrght/Simple-Diffusion/blob/main/SimpleDiffusion.ipynb)

Dhariwal, Prafulla, and Alex Nichol. “Diffusion Models Beat GANs on Image Synthesis.” arXiv, June 1, 2021. https://doi.org/10.48550/arXiv.2105.05233.

Nichol, Alex, and Prafulla Dhariwal. “Improved Denoising Diffusion Probabilistic Models.” arXiv, February 18, 2021. https://doi.org/10.48550/arXiv.2102.09672.

Diffusion Models

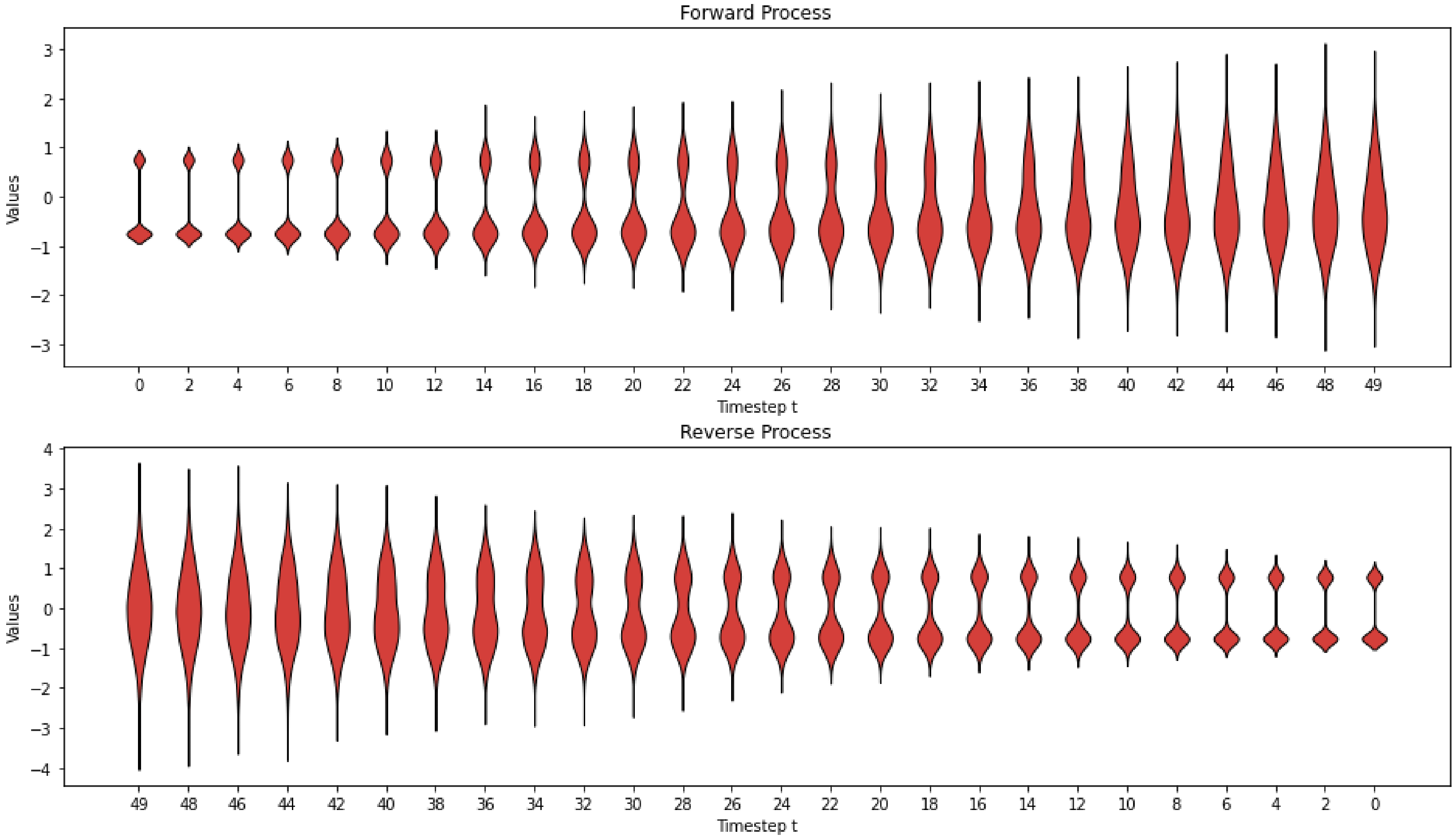
- **Generation by learning to reverse entropy**
- **Forward Process:** Generate noisy signals from data
  - Data distribution gets gradually converted to noise
- **Reverse Process:** Learn to denoise
  - Using a neural network $\epsilon_\theta(x_t, t)$ with weights θ which takes the noisy data x_t as input along with the time step t (and possibly other "conditioning" variables) to output an estimate of the noise ϵ_t that has been added to x_0 to generate x_t. This is achieved by solving the following optimization problem:

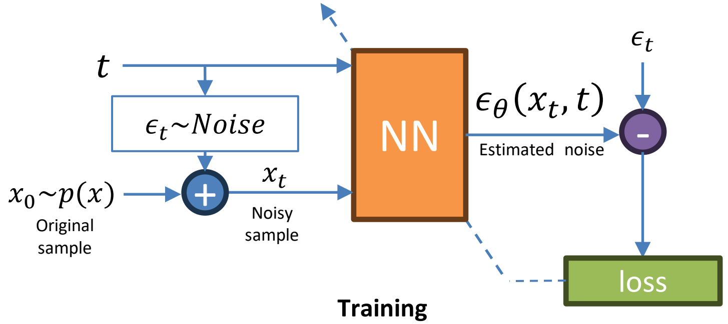
$$\min_\theta E_{t, x_0 \epsilon} |\epsilon_t - \epsilon_\theta(x_t, t)|^2$$

- **Generation:** Once the neural network is trained, we can generate data using:

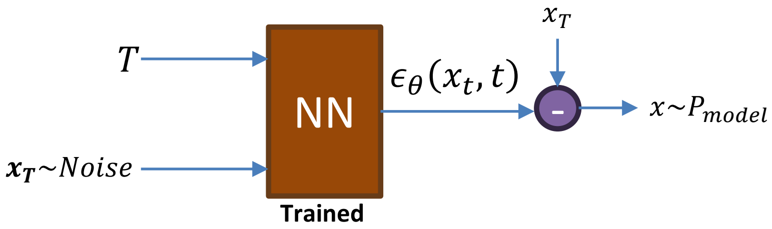
$$x = x_T - \epsilon_\theta(x_T, T) \text{ with } x_T \sim N(0,1)$$

- Can be improved by operating in a compressed or latent space: **Latent diffusion**





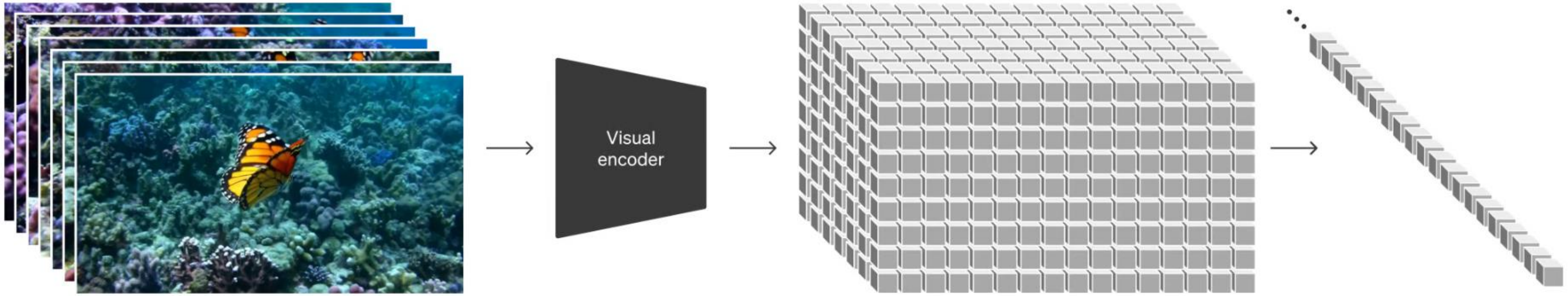
Training

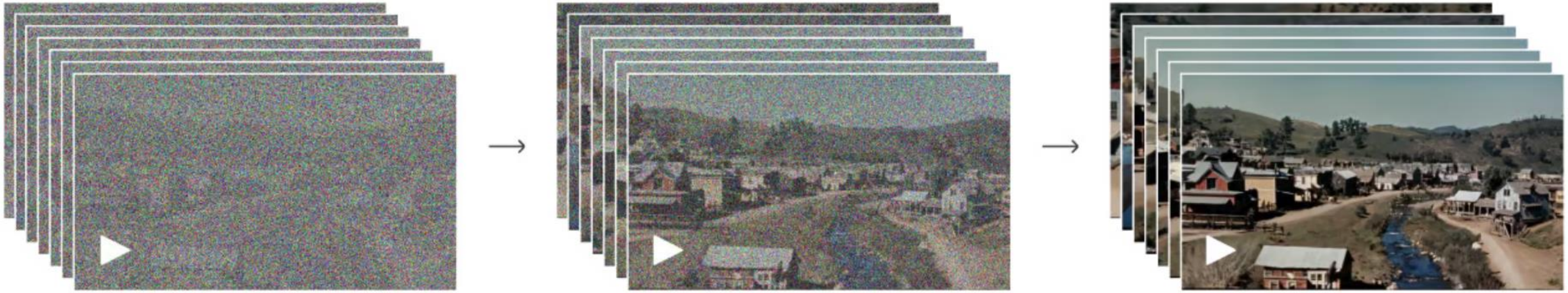


Generation

[Simplest Diffusion Tutorial: https://github.com/wgrgwrght/Simple-Diffusion/blob/main/SimpleDiffusion.ipynb](https://github.com/wgrgwrght/Simple-Diffusion/blob/main/SimpleDiffusion.ipynb)

SORA: Diffusion Transformer

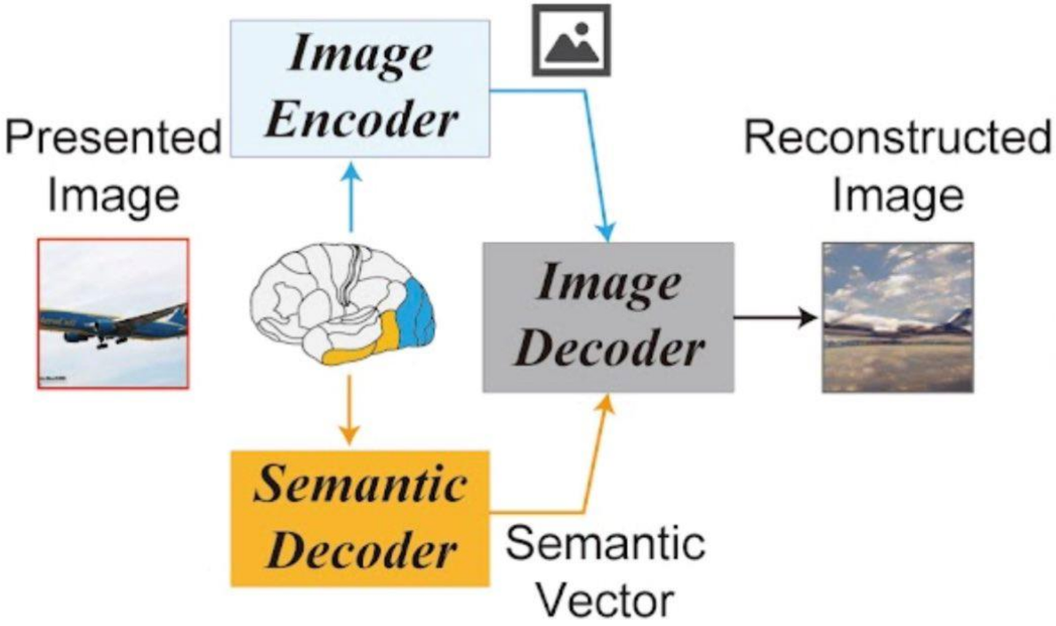




https://openai.com/research/video-generation-models-as-world-simulators

Seeing without seeing

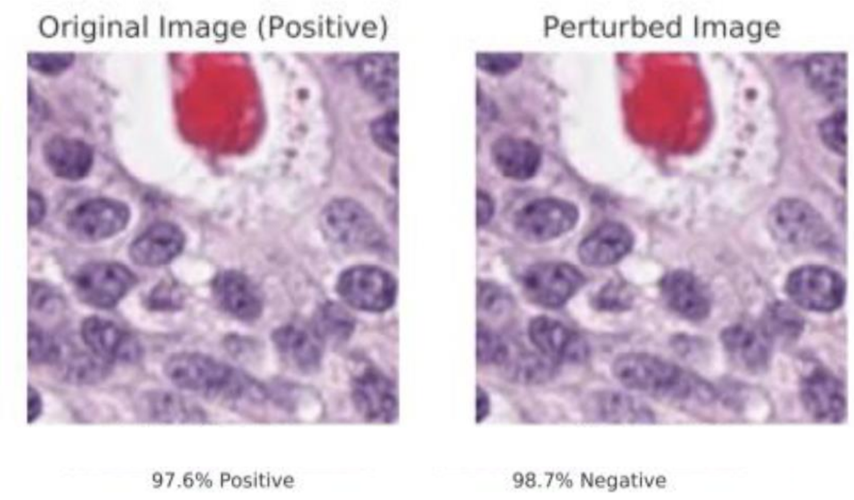
- Takagi, Yu, and Shinji Nishimoto. “High-Resolution Image Reconstruction with Latent Diffusion Models from Human Brain Activity.” bioRxiv, December 1, 2022. https://doi.org/10.1101/2022.11.18.517004.

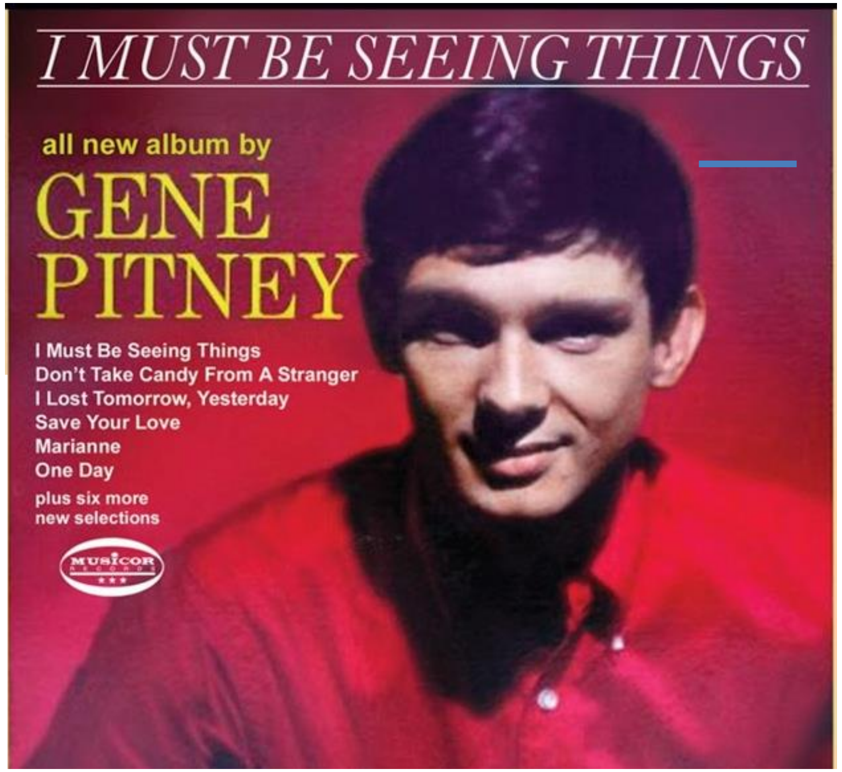


CONCLUSIONS

Issues

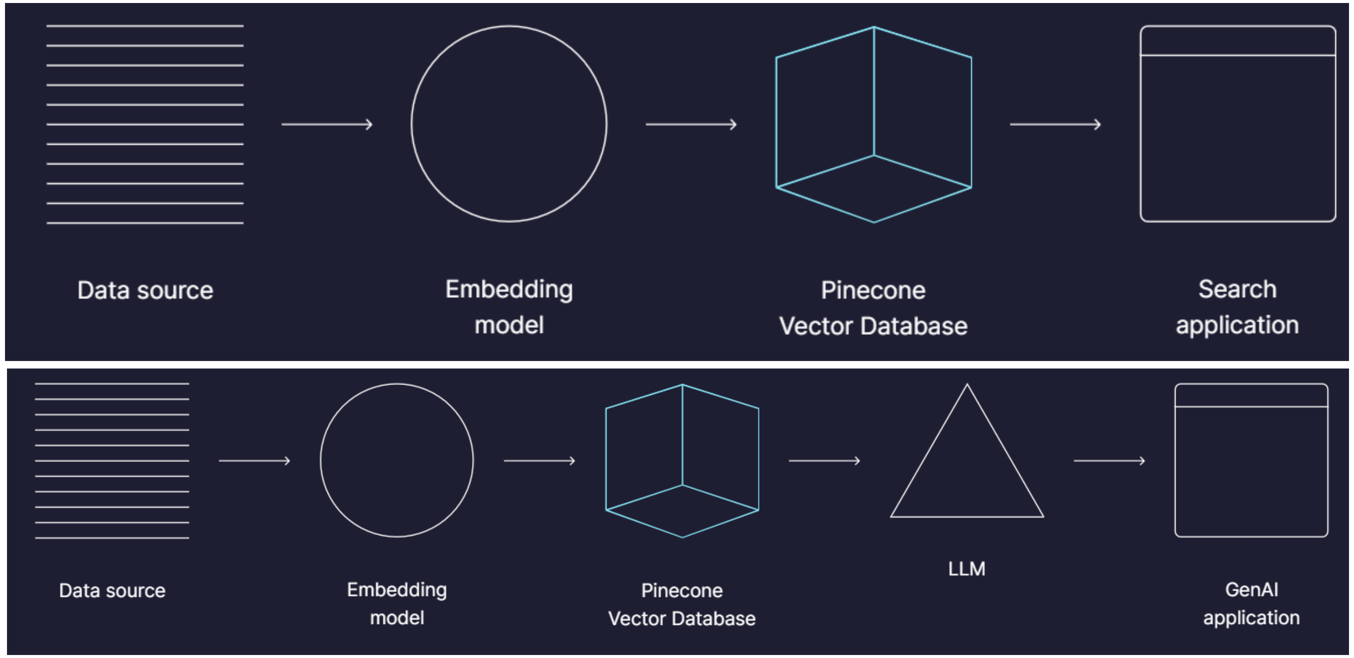
- Deep Neural Networks are Easily Fooled
  - https://arxiv.org/abs/1412.1897v4
- Failures of deep learning
  - https://arxiv.org/abs/1703.07950
- To understand deep learning we need to understand kernel learning
  - https://arxiv.org/abs/1802.01396
- Understanding deep learning requires rethinking generalization
- Steps toward deep kernel methods from infinite neural networks
  - https://arxiv.org/abs/1508.05133
- Do Deep Neural Networks Really Need to be Deep?
- One pixel attack for fooling deep neural networks
  - https://www.youtube.com/watch?v=SA4YEAWVpbk
  - https://github.com/Hyperparticle/one-pixel-attack-keras
- Adversarial Examples that Fool both Computer Vision and Time-Limited Humans
- Alchemy? https://www.youtube.com/watch?v=ORHFOnaEzPc
  - Ali Rahimi





The Rise of Vector Databases

- Flowise, langchain



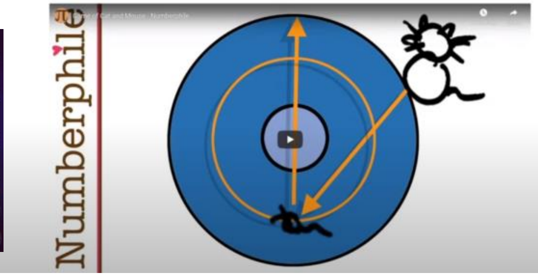
https://www.pinecone.io/

https://github.com/pinecone-io/examples/blob/master/docs/gpt-4-langchain-docs.ipynb

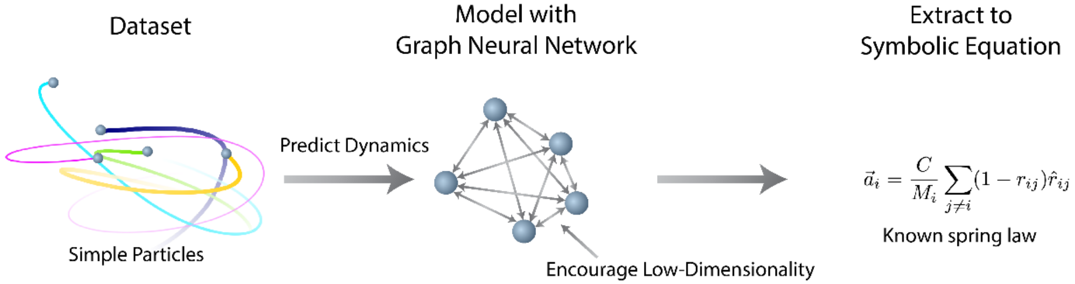
Other Topics

- Recurrent Neural Networks
- Reinforcement Learning
  - Learning from experience
  - Example: Learning to levitate or helping a mouse escape from a cat
  - https://github.com/foxtrotmike/RL-MagLev/blob/master/RL.ipynb
  - https://github.com/foxtrotmike/RL-MagLev/blob/master/cat_mouse.ipynb
- Learning Paradigms
  - Multi-task Learning
  - Multi-Label Learning
  - Self-Supervised Learning
    - Learn a task to learn a feature representation and adapt it to other tasks
    - Contrastive Learning
  - Zero Shot and Few Shot Learning
- Bayesian Neural Networks and Uncertainty Quantification (Conformal Prediction)
- Neural Ordinary Differential Equations (NODE)
  - https://github.com/foxtrotmike/NODE-Tutorial/blob/main/node_tutorial%20(2).ipynb
- Data Efficient Learning
- Symbolic Regression
- Learning to Learn
- Quantum ML
- Domain Generalization
- Robustness
- Building invariances into machine learning models
- Link between Causality, Symmetry, Invariance and Generalization
- Prompt Engineering, Retrieval Augmented Generation

The Cat and Mouse Puzzle
A Neural Solution



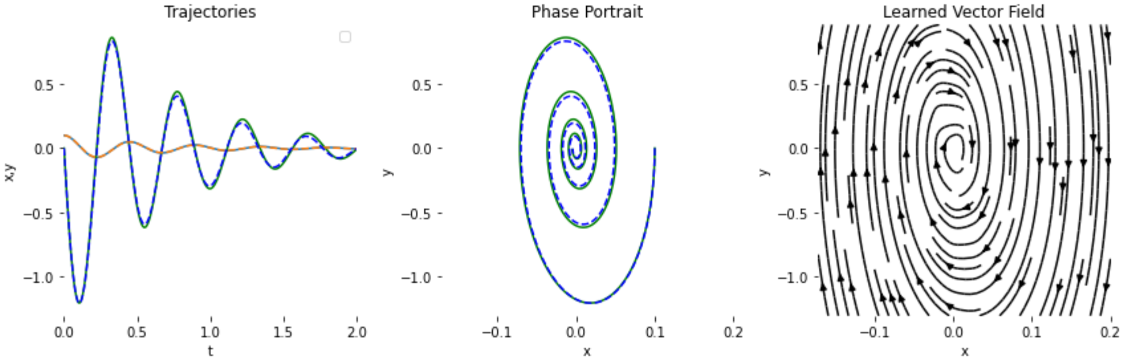
My RL Tutorial Video: https://youtu.be/N20h6vpR13Y



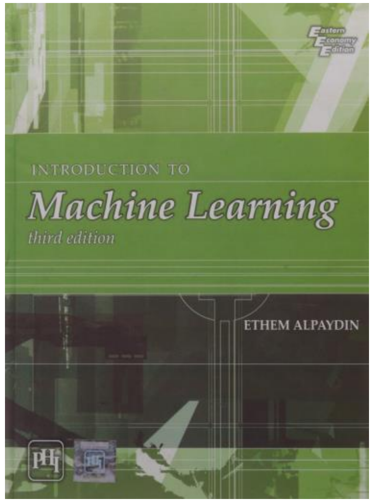
https://astroautomata.com/paper/symbolic-neural-nets/

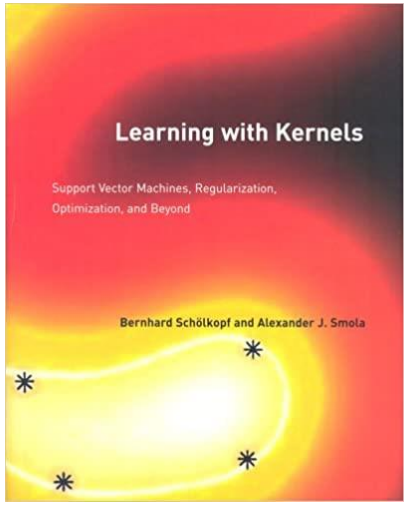
$$\dot{x_1}(t) = v(t) = x_2(t)$$

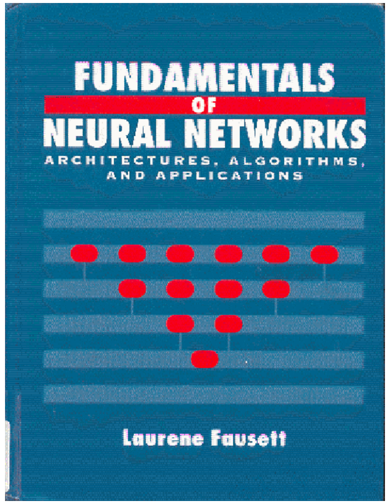
$$\dot{x_2}(t) = \dot{v}(t) = -\frac{g}{l}sin(x(t)) - \frac{k}{ml}v(t) = -\frac{g}{l}sin(x_1(t)) - \frac{k}{ml}x_2(t)$$

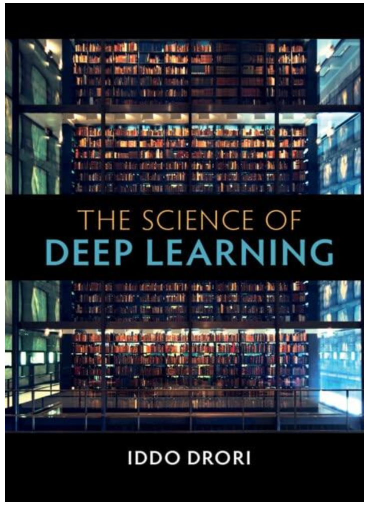


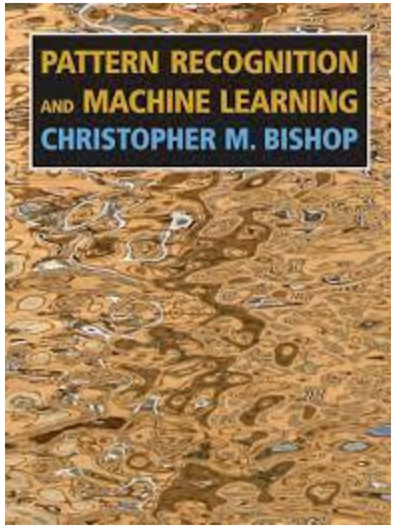
Books

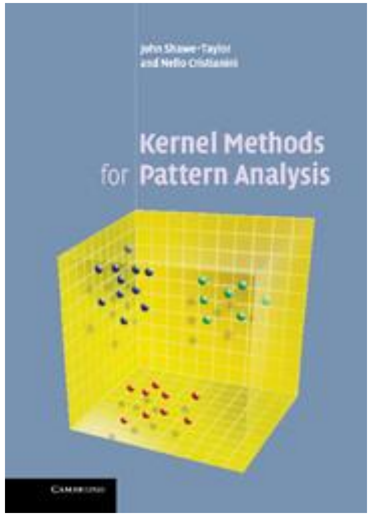


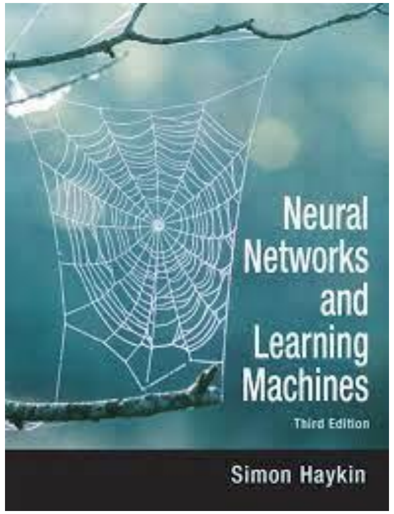


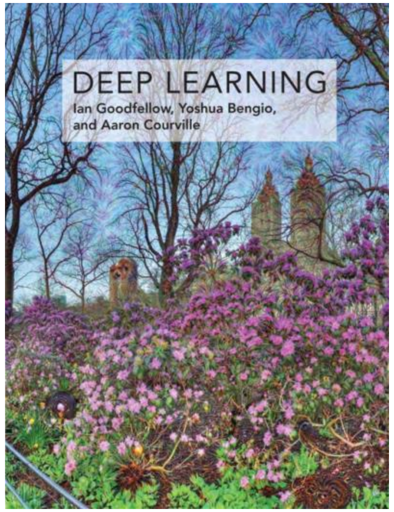


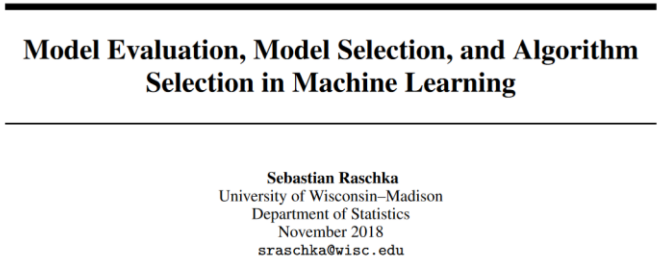












Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning

Sebastian Raschka
University of Wisconsin–Madison
Department of Statistics
November 2018
sraschka@wisc.edu

Understanding Deep Learning
by Simon J.D. Prince
https://udlbook.github.io/udlbook/

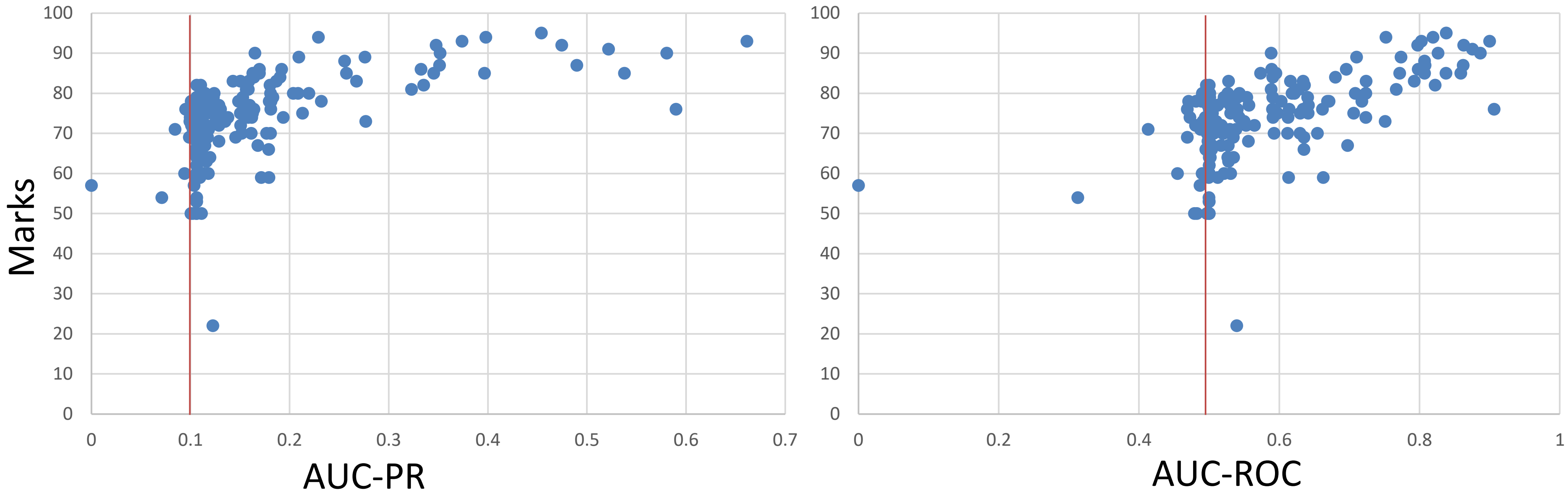
Foundations

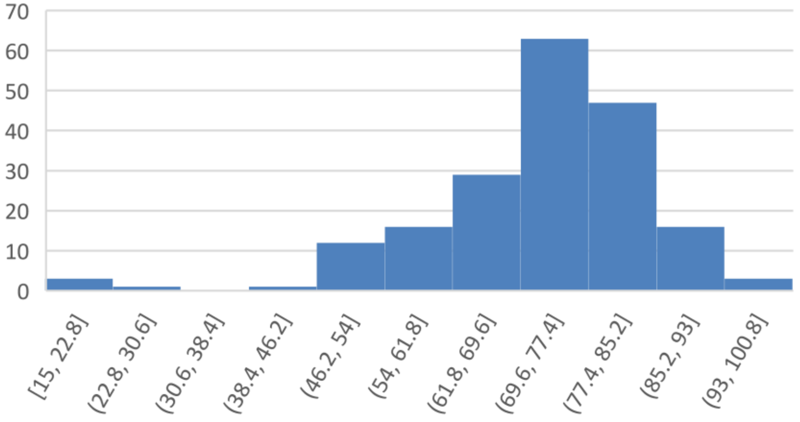
SVMs and Kernels

Backpropagation and MLPs

Deep Learning

Assignment 1 Grades




Marks Distribution

THE END (2024)

Any Slides After This Are Optional and not included in the 2024 exam