



Computer  
Science

# **CSC 480/580: Principles of Machine Learning**

## **Feed Forward Neural Networks**

Chicheng Zhang

# Basis Functions

Basis functions transform linear models into nonlinear ones...

**Linear Regression**

$$y = w^T x$$



$$y = w^T \phi(x)$$

**Classification  
( Logistic Regression )**

$$y = \sigma(w^T x)$$



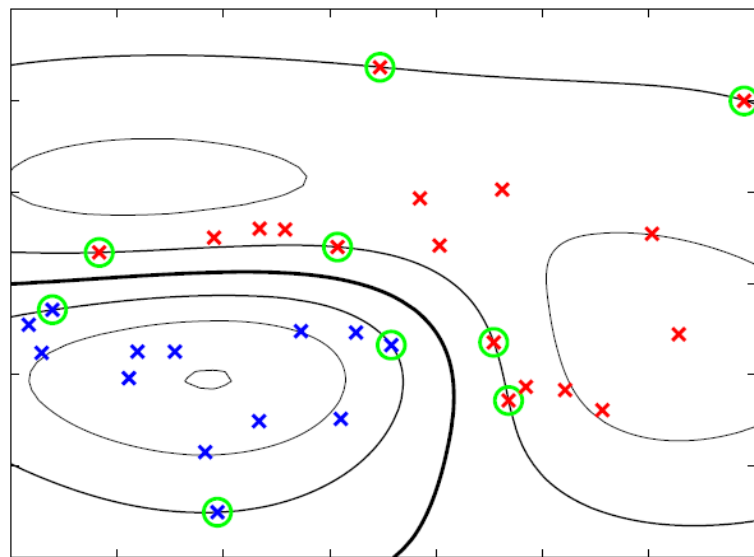
$$y = \sigma(w^T \phi(x))$$

...but it is often difficult to find a good basis transformation

# Learning Basis Functions

What if we could learn a basis function so that a simple linear model performs well...

**Data Space**

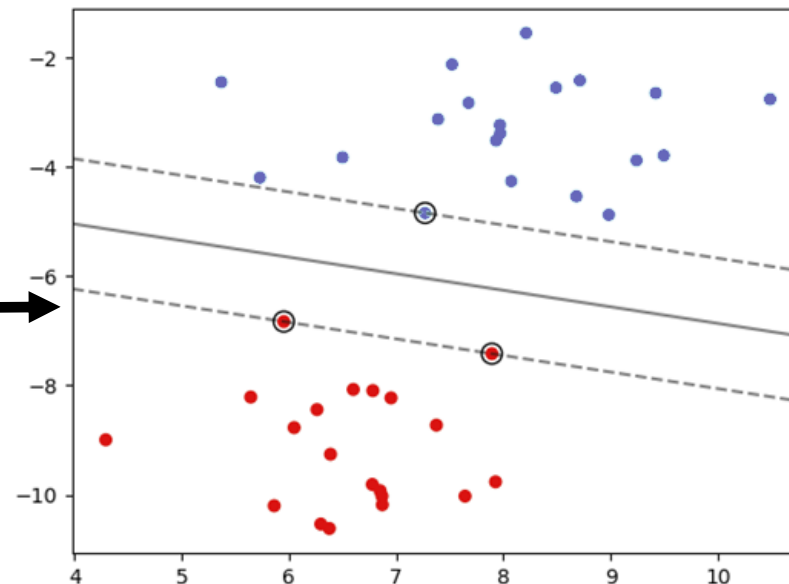


Ignore the circled points...  
reused these from the SVM slides

**Neural Net**  
 $\phi(x)$



**Warped Space**



...this is essentially what standard neural networks do...

# Neural Networks

- Flexible nonlinear transformations of data
- Resulting transformation is easily fit with a linear model
- Relatively efficient learning procedure scales to massive data
- Apply to many Machine Learning / Data Science problems
  - Regression
  - Classification
  - Dimensionality reduction
  - Function approximation
  - Many application-specific problems



# Neural Networks

Forms of NNs are used all over the place nowadays...



**AI Chat Bots**

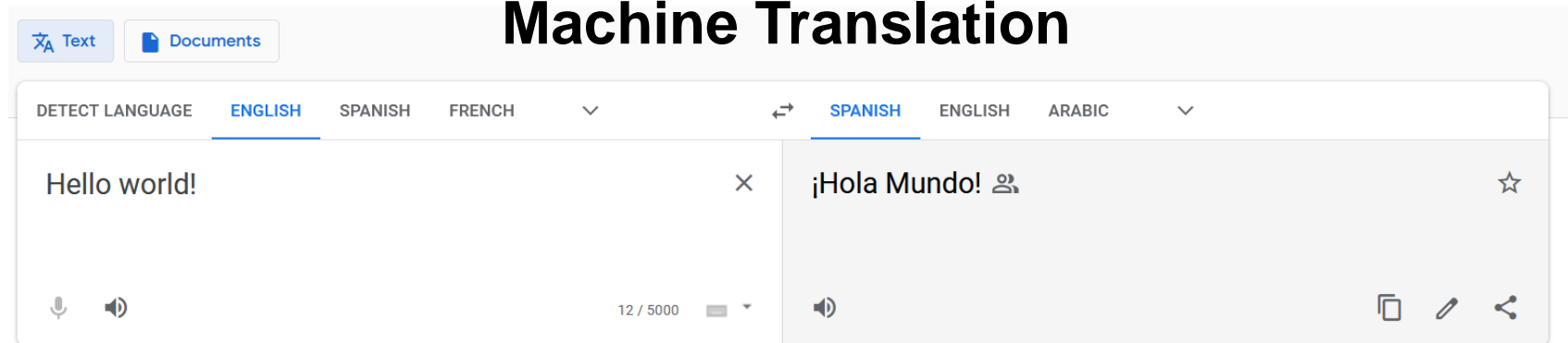


**Self-Driving Cars**



**Creepy Robots**

## Machine Translation

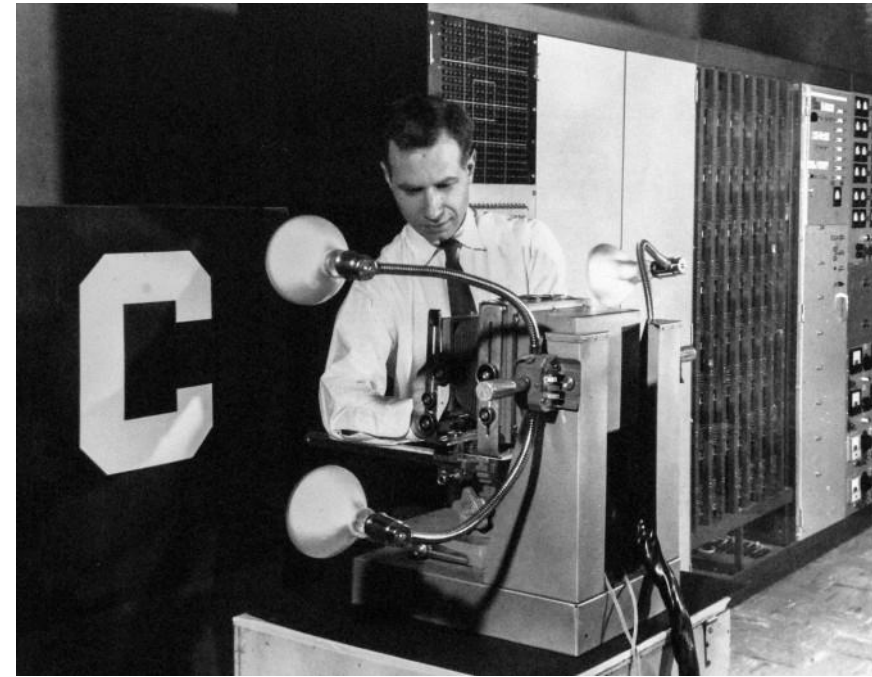


[Send feedback](#)

# Rosenblatt's Perceptron

Despite recent attention, neural networks are fairly old

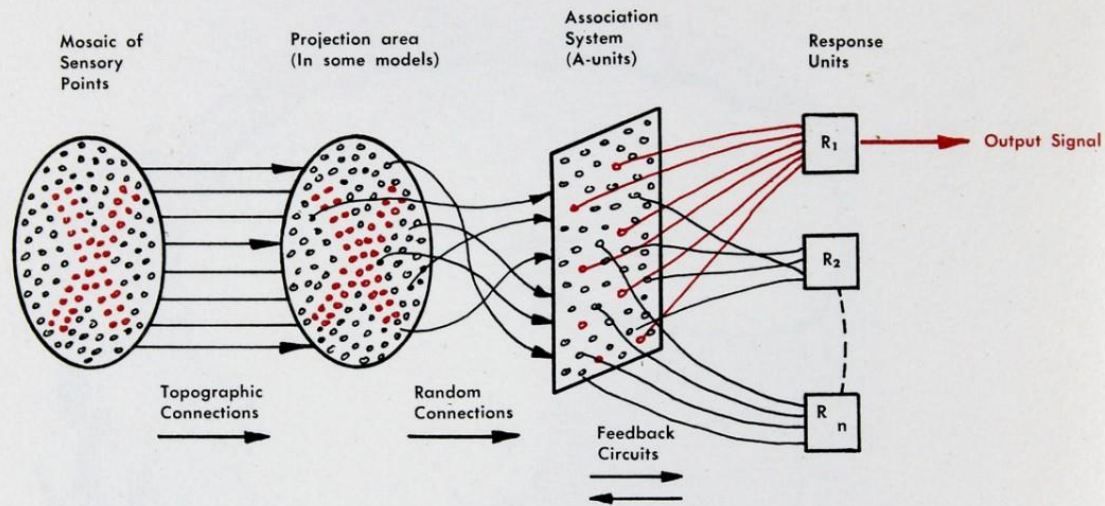
In 1957 Frank Rosenblatt constructed the first (single layer) neural network known as a "perceptron"



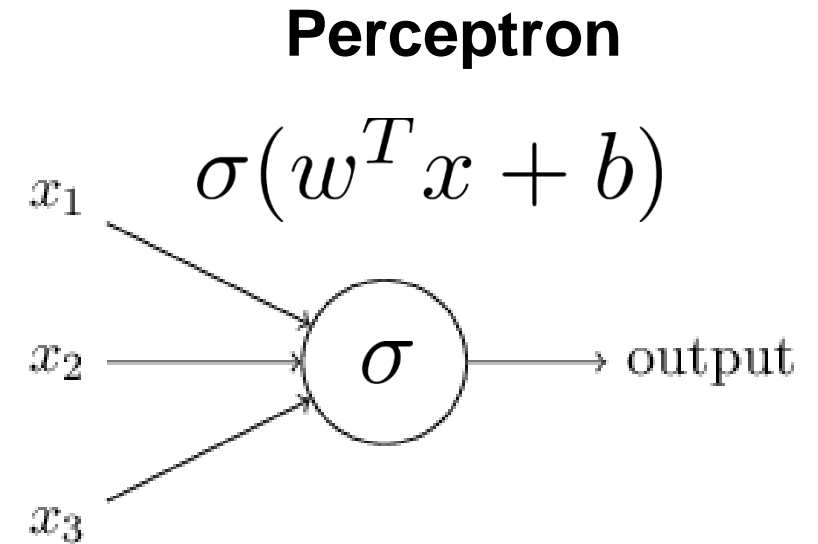
He demonstrated that it is capable of recognizing characters projected onto a 20x20 "pixel" array of photosensors

# Rosenblatt's Perceptron

**FIG. 1** — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

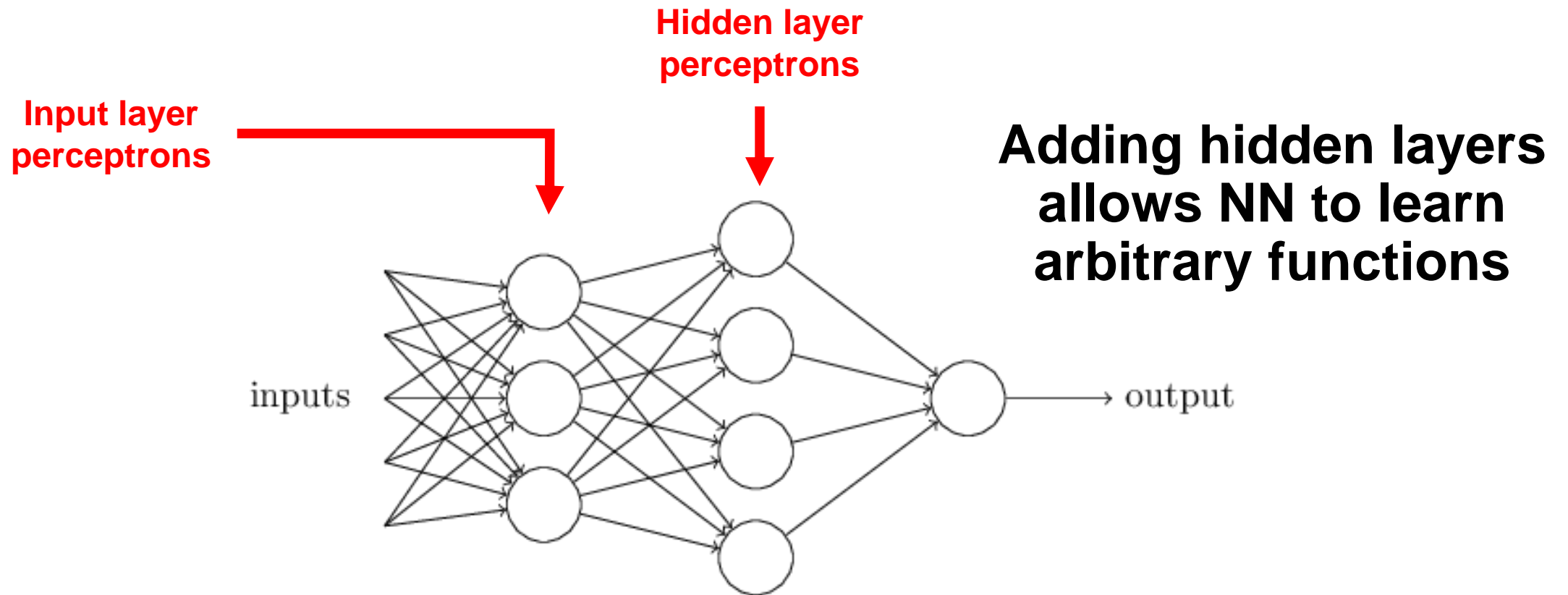


**FIG. 2** — Organization of a perceptron.



- In Rosenblatt's perceptron, the inputs are tied directly to output
- "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanics" (1962)
- Criticized by Marvin Minsky in book "Perceptrons" since can only learn linearly-separable functions
- **The perceptron is just linear classification in disguise**

# Multilayer Perceptron

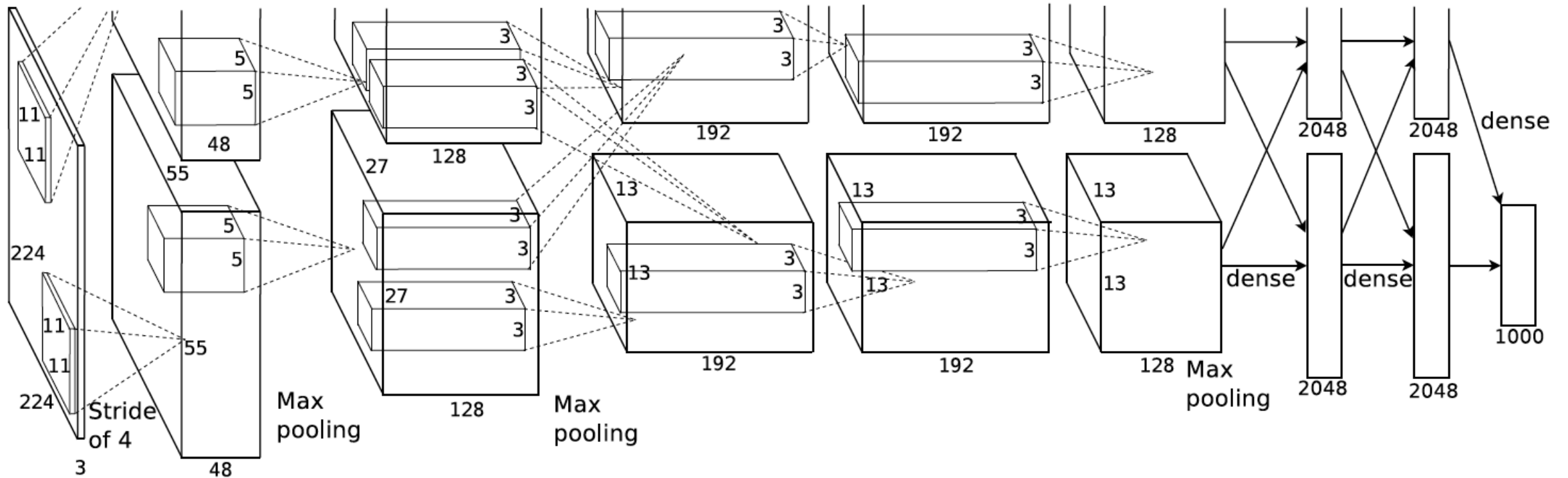


This is the quintessential *Neural Network...*  
...also called *Feed Forward Neural Net* or *Artificial Neural Net*

[ Source: <http://neuralnetworksanddeeplearning.com> ]

# Modern Neural Networks

Modern *Deep Neural networks* have many hidden layers



...and have many trillions of parameters to learn

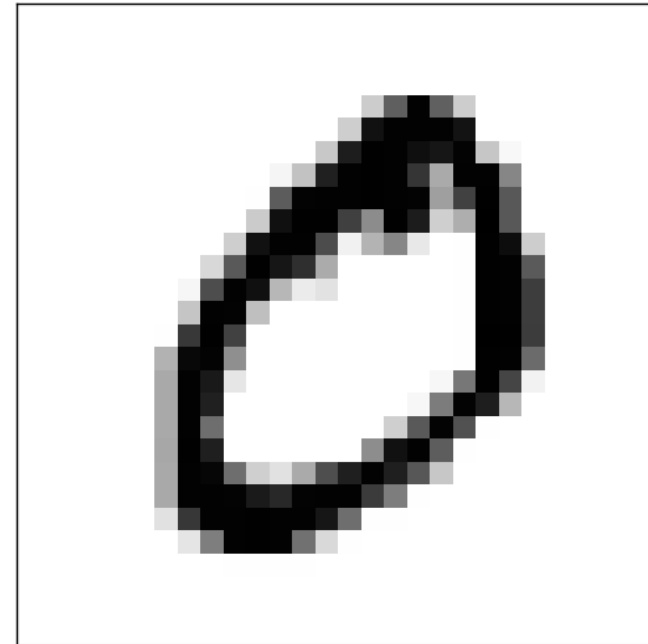


# Handwritten Digit Classification

Classifying handwritten digits is the “Hello World” of NNs



Each character is centered  
in a  $28 \times 28 = 784$  pixel  
grayscale image

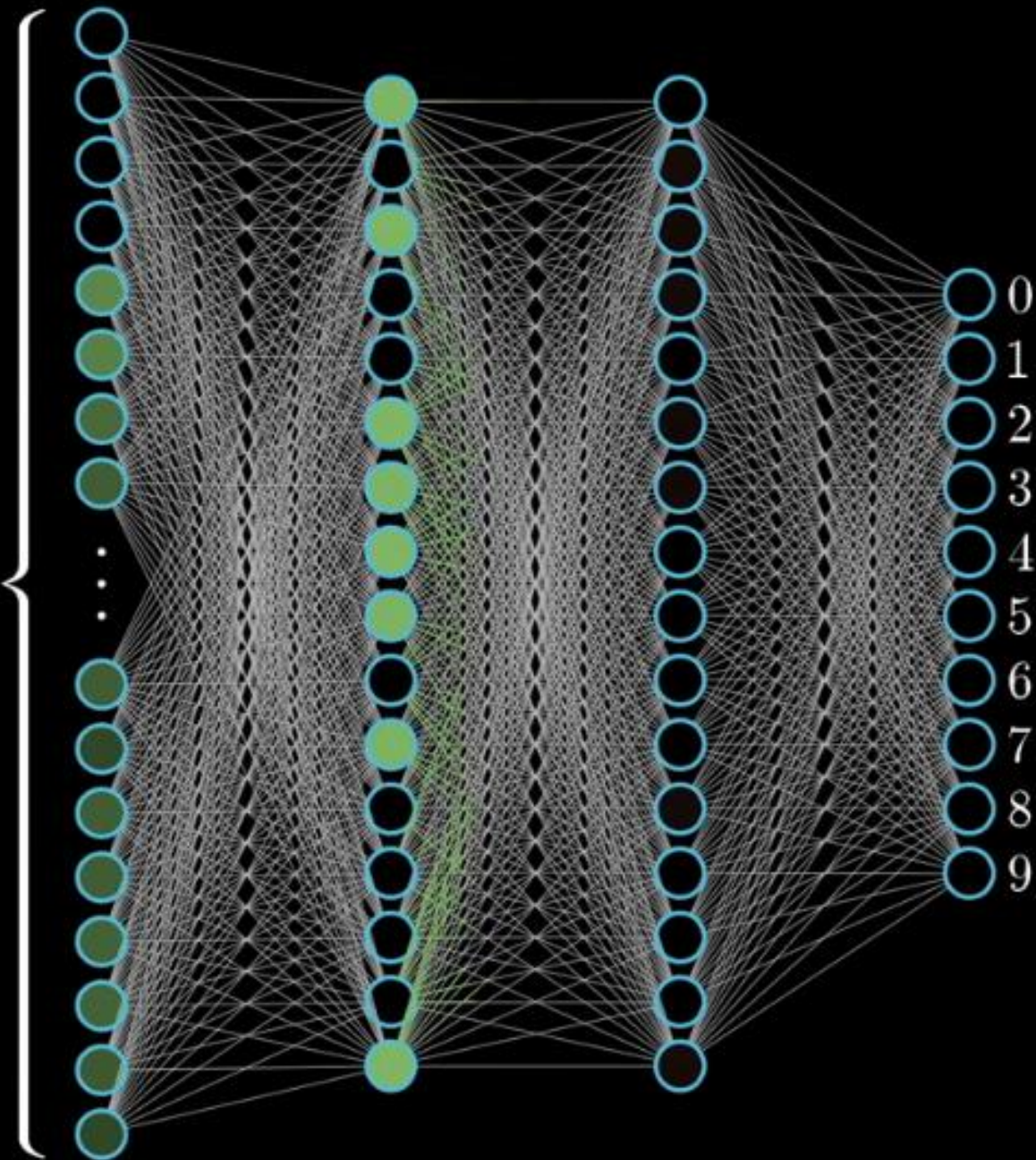


Modified National Institute of  
Standards and Technology  
(MNIST) database contains 60k  
training and 10k test images

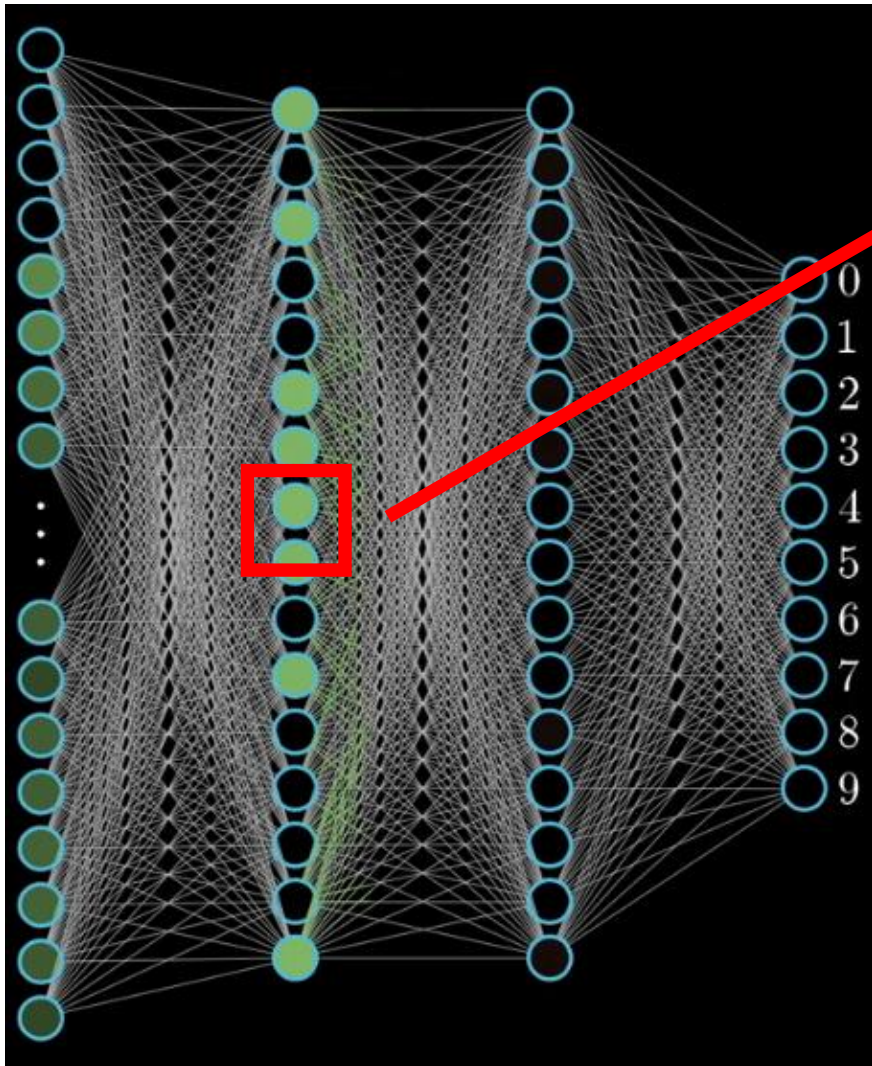


784

Each image pixel is a number in  $[0,1]$  indicated by highlighted color



# Feedforward Procedure



Each node computes a *weighted combination* of nodes at the previous layer...

$$w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Then applies a *nonlinear function* to the result

$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

**Often, we also introduce a constant *bias* parameter**



# Nonlinear Activation functions

We call this an *activation function* and typically write it in vector form,

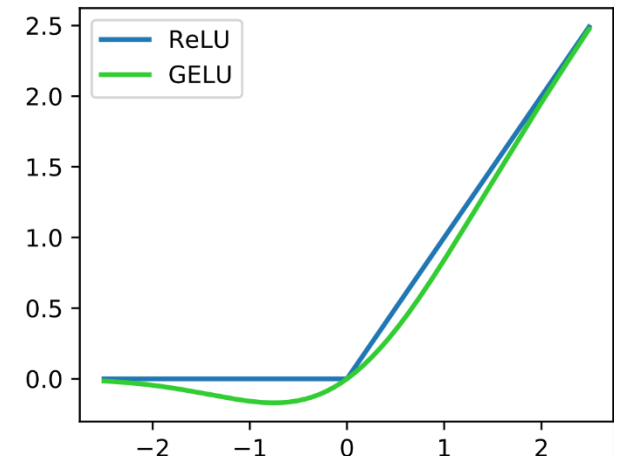
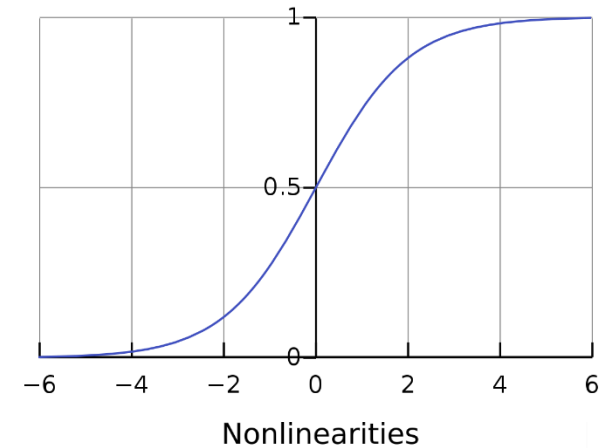
$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b) = \sigma(w^T x + b)$$

An early choice was the *logistic function*,

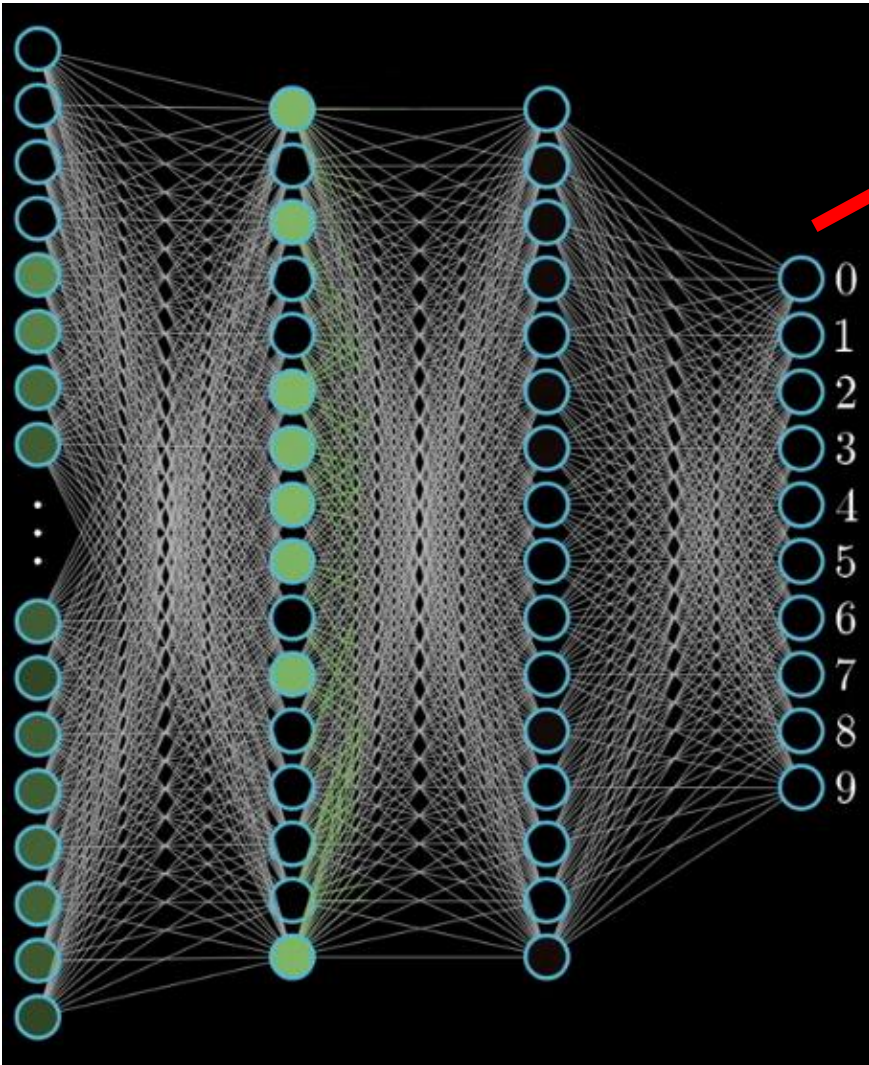
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Later found to lead to slow learning and the *rectified linear unit (ReLU)* become popular,

$$\sigma(w^T x + b) = \max(0, w^T x + b)$$



# Multilayer Perceptron



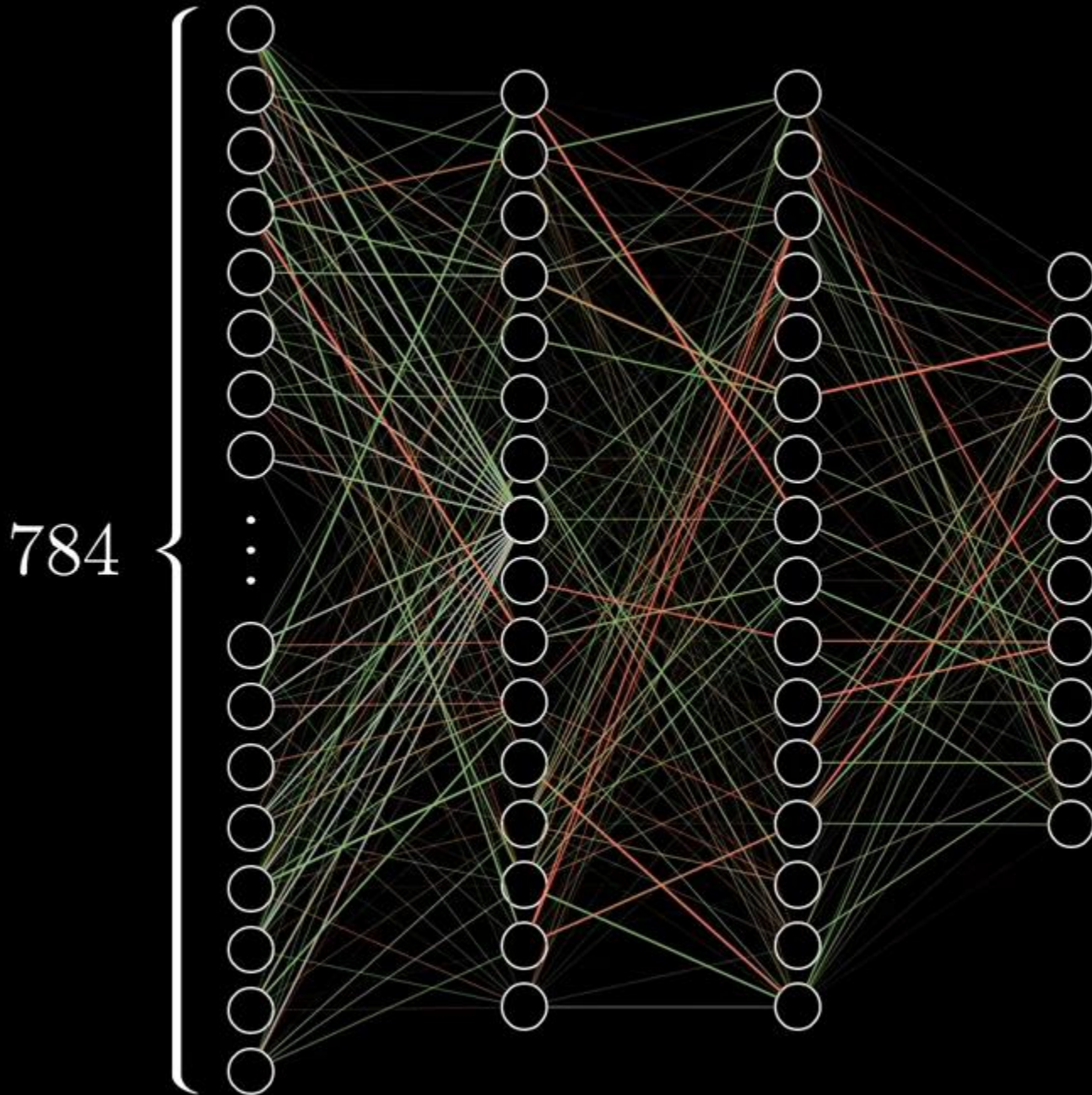
Final layer is typically a linear model... each output node is computed by

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

**Vector of activations from previous layer**

Recall that for binary logistic regression with 2 classes,

$$p(\text{Class} = 1 \mid x) \propto \sigma(w^T x + b)$$



$$784 \times 16 + 16 \times 16 + 16 \times 10$$

weights

$$16 + 16 + 10$$

biases

13,002

Each parameter has some impact on the output...need to tweak (learn) all parameters simultaneously to improve prediction accuracy

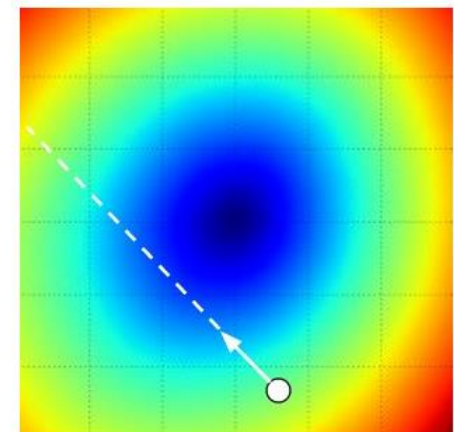


# Reading quiz

- <https://cs231n.github.io/optimization-1/>
- New concepts
  - Optimization methods:
    - random search,
    - random local search,
    - Cf. gradient descent
  - Gradient check: use *slow & easy numerical gradient* to check the correctness of implementation of *fast & error prone analytical gradient*

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- Effect of step size



# Training Multilayer Perceptron

$$X^{\text{Train}} = \begin{matrix} \begin{matrix} 0 & 4 & 1 & 9 & 2 & 1 & 3 & 1 & 4 & 3 \\ 5 & 3 & 6 & 1 & 7 & 2 & 8 & 6 & 9 & 4 \\ 0 & 9 & 1 & 1 & 2 & 4 & 3 & 2 & 7 & 3 \\ 8 & 6 & 9 & 0 & 5 & 6 & 0 & 7 & 6 & 1 \\ 8 & 7 & 9 & 3 & 9 & 8 & 5 & 9 & 3 & 3 \\ 0 & 7 & 4 & 9 & 8 & 0 & 9 & 4 & 1 & 4 \\ 4 & 6 & 0 & 4 & 5 & 6 & 1 & 0 & 0 & 1 \\ 7 & 1 & 6 & 3 & 0 & 2 & 1 & 1 & 7 & 9 \\ 0 & 2 & 6 & 7 & 8 & 3 & 9 & 0 & 4 & 6 \\ 7 & 4 & 6 & 8 & 0 & 7 & 8 & 3 & 1 & 5 \end{matrix} \end{matrix}$$

$$Y^{\text{Train}} = \begin{pmatrix} 0 & 4 & 1 & \dots & 3 \\ 5 & 3 & 6 & \dots & 4 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 7 & 4 & 6 & \dots & 5 \end{pmatrix}$$



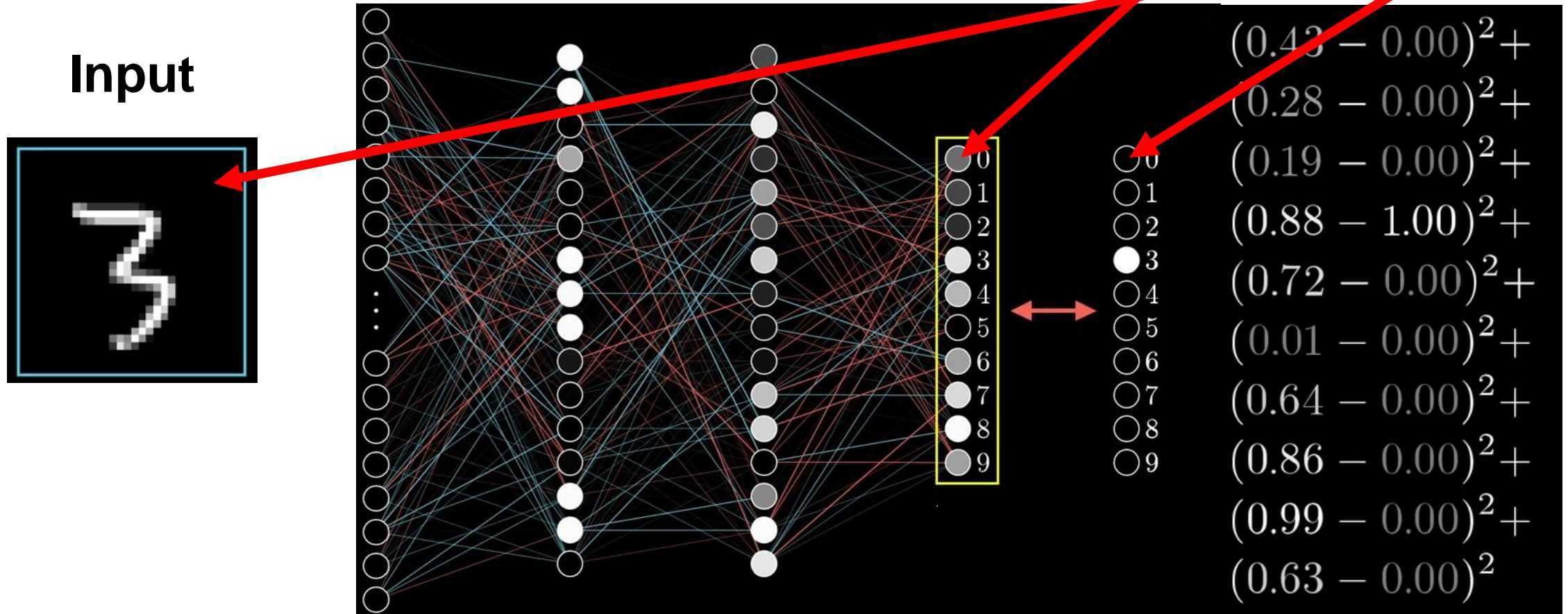
For each training example, predict label and adjust weights...



- How to score final layer output?
- How to adjust weights?

# Training Multilayer Perceptron

One way to score (square loss): based on difference between final layer and one-hot vector of true class...  $\ell_i(\theta) = \sum_j (f_j(x_i; \theta) - y_j)^2$



# Training Multilayer Perceptron: for classification

- For classification, it is most popular to use:

- A softmax layer as final output

$$\sigma(\vec{f})_c = \frac{e^{f_c}}{\sum_{j=1}^C e^{f_j}}, c = 1, \dots, K$$

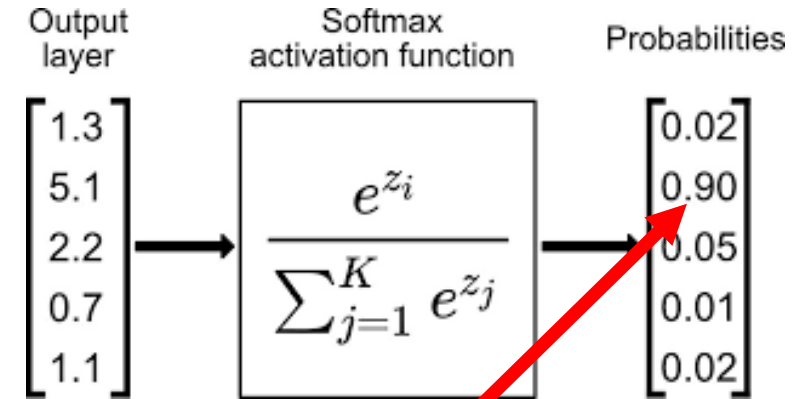
probability estimate of each class given example

- Cross-entropy loss for training

$$\ell(\vec{p}, y) = \log\left(\frac{1}{p_y}\right)$$

(if  $l$  represents one-hot encoding of label, often written as  $\sum_c l_c \log\left(\frac{1}{p_c}\right)$ )

measures the “surprise” of label being  $y$  based on current belief




E.g.  $y = 2$ ,  $\ell(\vec{p}, y) = \log\frac{1}{0.90}$



# Training Multilayer Perceptron

Our loss function for  $i^{\text{th}}$  example is error in terms of weights / biases...

$$\ell_i(\theta), \quad \theta := (w_1, \dots, w_n, b_1, \dots, b_n)$$


**13,002 Parameters  
in this network**

...minimize loss over all training data...

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^m \ell_i(\theta)$$

This is a super high-dimensional optimization (13,002 dimensions in this example)...how do we solve it?

**Gradient descent!**



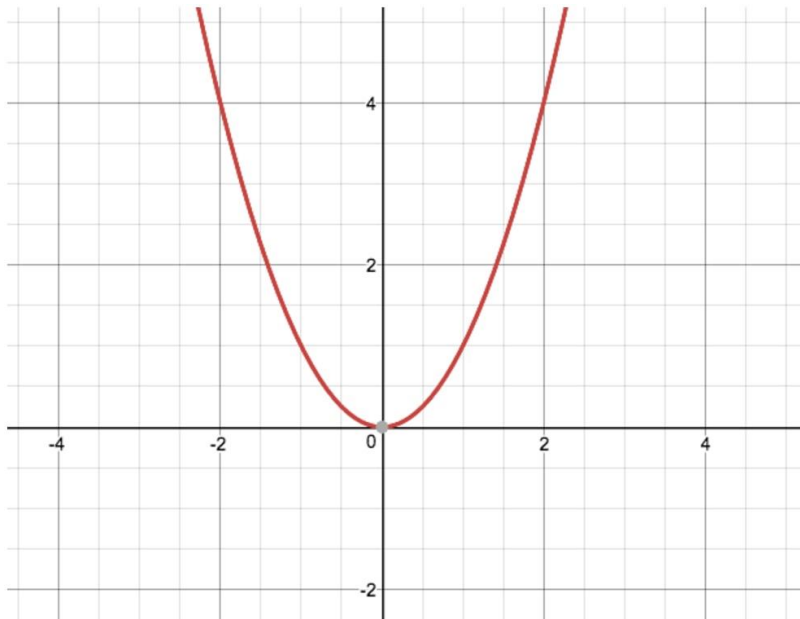
# Learning algorithm intuition

- Gradient descent: Move in direction of greatest local improvement (greedily)
- “Knob turning”
  - ”knob” = weight of an edge
  - If a neuron increases the probability of an incorrect prediction, its knobs will be turned down.
  - If a neuron increases the probability of a correct prediction, its knobs will be turned up.

# Training Multilayer Perceptron

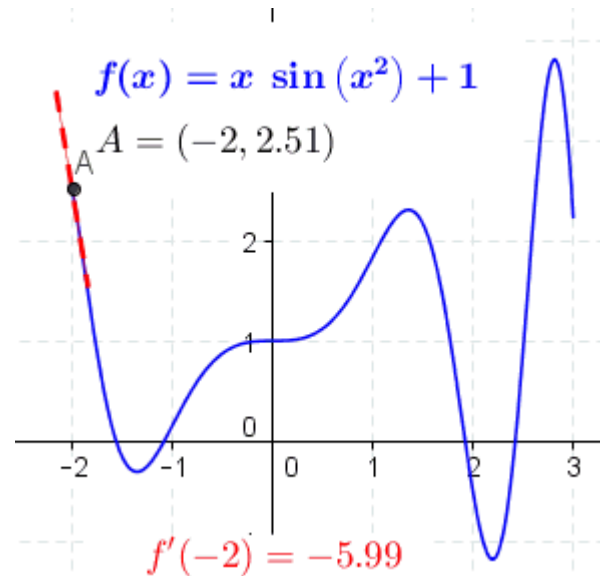
Need to find local / global optimal solution...

Convex Cost Function



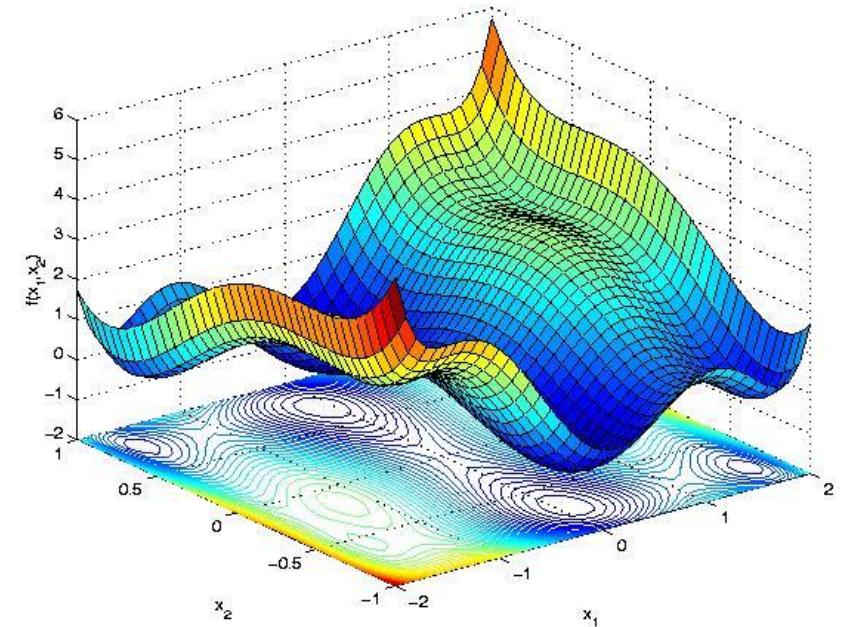
Easy!

Non-convex Cost Function



Hard!

High-Dimensional Non-convex

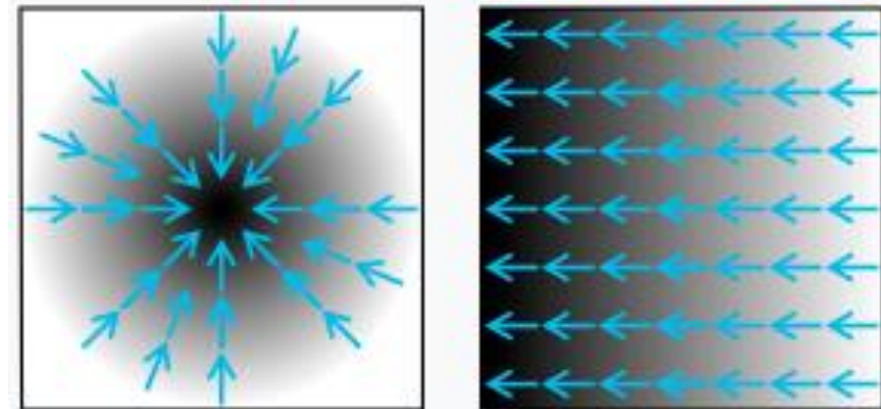


Hard!

Actually, the situation is much worse, since the cost is super 13,002(high)-dimensional...but we proceed as is...

# Gradient descent (GD)

- For  $t = 1, \dots, T$ :  $\theta_{t+1} \leftarrow \theta_t - \alpha \nabla \mathcal{L}(\theta_t)$
- $\nabla \mathcal{L}(\theta) = \left( \frac{\partial \mathcal{L}}{\partial \theta(1)}, \dots, \frac{\partial \mathcal{L}}{\partial \theta(d)} \right)$  is the *gradient* of  $\mathcal{L}$  at point  $\theta$
- Intuition:  $\nabla \mathcal{L}(\theta) / -\nabla \mathcal{L}(\theta)$  points to the direction  
In which  $\mathcal{L}$  increase / decreases the most
  - GD provides local improvement on  $\mathcal{L}$
- Special case (linear fn):  $\mathcal{L}(\theta) = x^T \theta$



# Gradient descent: example

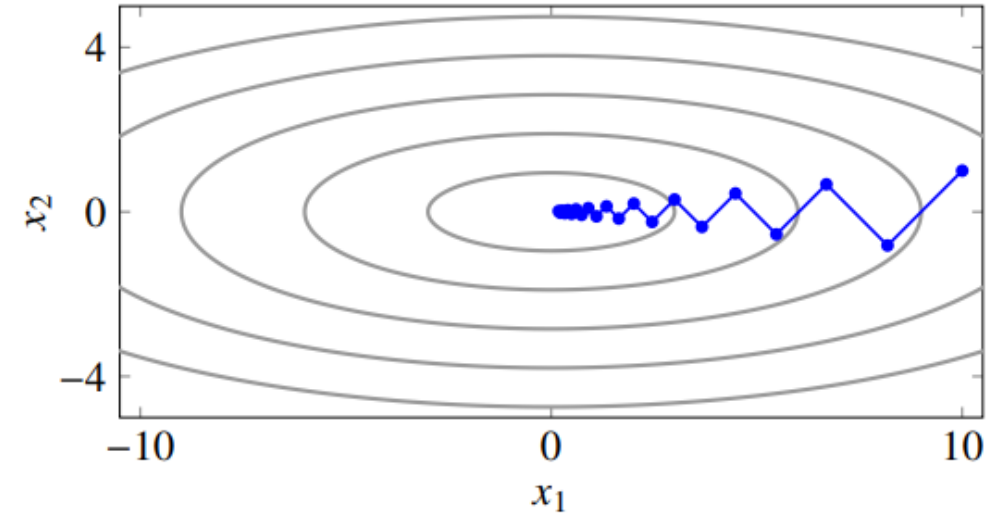
- Example:  $\mathcal{L}(\theta) = \frac{1}{2} (\theta(1)^2 + 4\theta(2)^2)$

- $\nabla \mathcal{L}(\theta) = (\theta(1), 4\theta(2))$

- $$\begin{pmatrix} \theta_{t+1}(1) \\ \theta_{t+1}(2) \end{pmatrix} = \begin{pmatrix} \theta_t(1) \\ \theta_t(2) \end{pmatrix} - \alpha \begin{pmatrix} \theta_t(1) \\ 4\theta_t(2) \end{pmatrix} = \begin{pmatrix} (1 - \alpha)\theta_t(1) \\ (1 - 4\alpha)\theta_t(2) \end{pmatrix} = \dots = \begin{pmatrix} (1 - \alpha)^t \theta_1(1) \\ (1 - 4\alpha)^t \theta_1(2) \end{pmatrix}$$

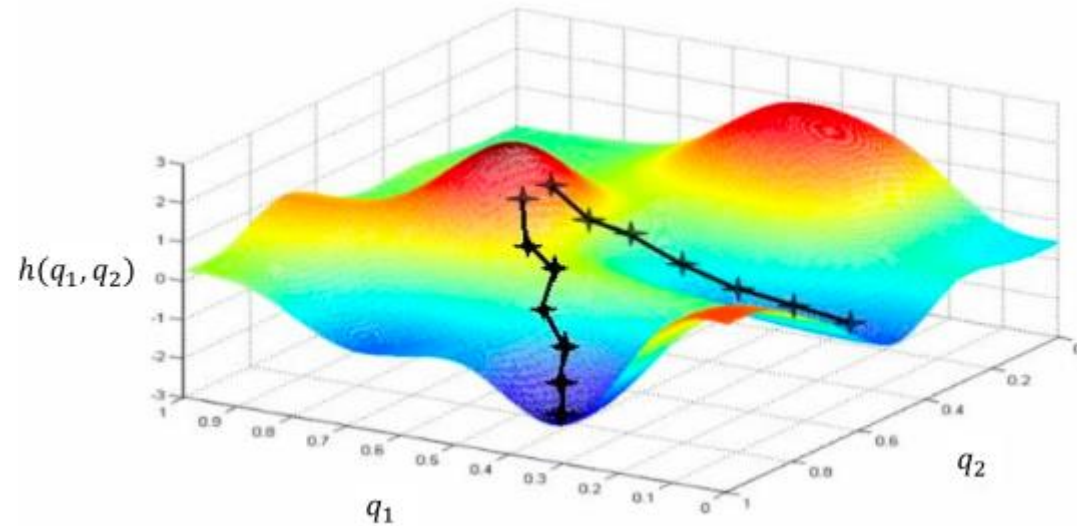
- When  $\alpha$  is small,  $\theta_t \rightarrow (0,0)$  - global minimizer of  $\mathcal{L}$

Vanishing as  $t \rightarrow \infty$



# GD: remarks

- Does not promise to converge to global optima



- In practice, people use it (or its variants) anyways and it performs well
  - This is especially true in training neural networks

# Stochastic gradient descent (SGD)

- Using GD to optimize training loss  $\mathcal{L}(\theta)$  can be expensive!
  - Full gradient  $\nabla\mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla\ell_i(\theta)$ , which takes  $O(m)$  time to evaluate

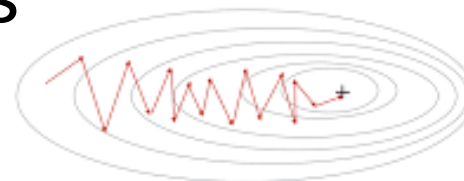
- To reduce computational cost: use instead

$$\theta_{t+1} \leftarrow \theta_t - \alpha g_t,$$

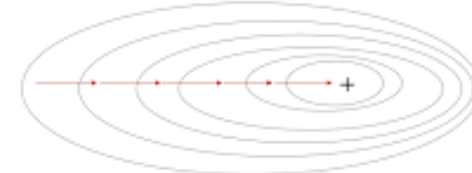
where  $g_t$  is an *unbiased estimate* of  $\nabla\mathcal{L}(\theta_t)$

- One choice:  $g_t = \nabla\ell_{i_t}(\theta_t)$ , where  $i_t \sim \text{Uniform}(\{1, \dots, m\})$ 
  - Only one gradient evaluation!
  - Well-established convergence guarantees

Stochastic Gradient Descent



Gradient Descent

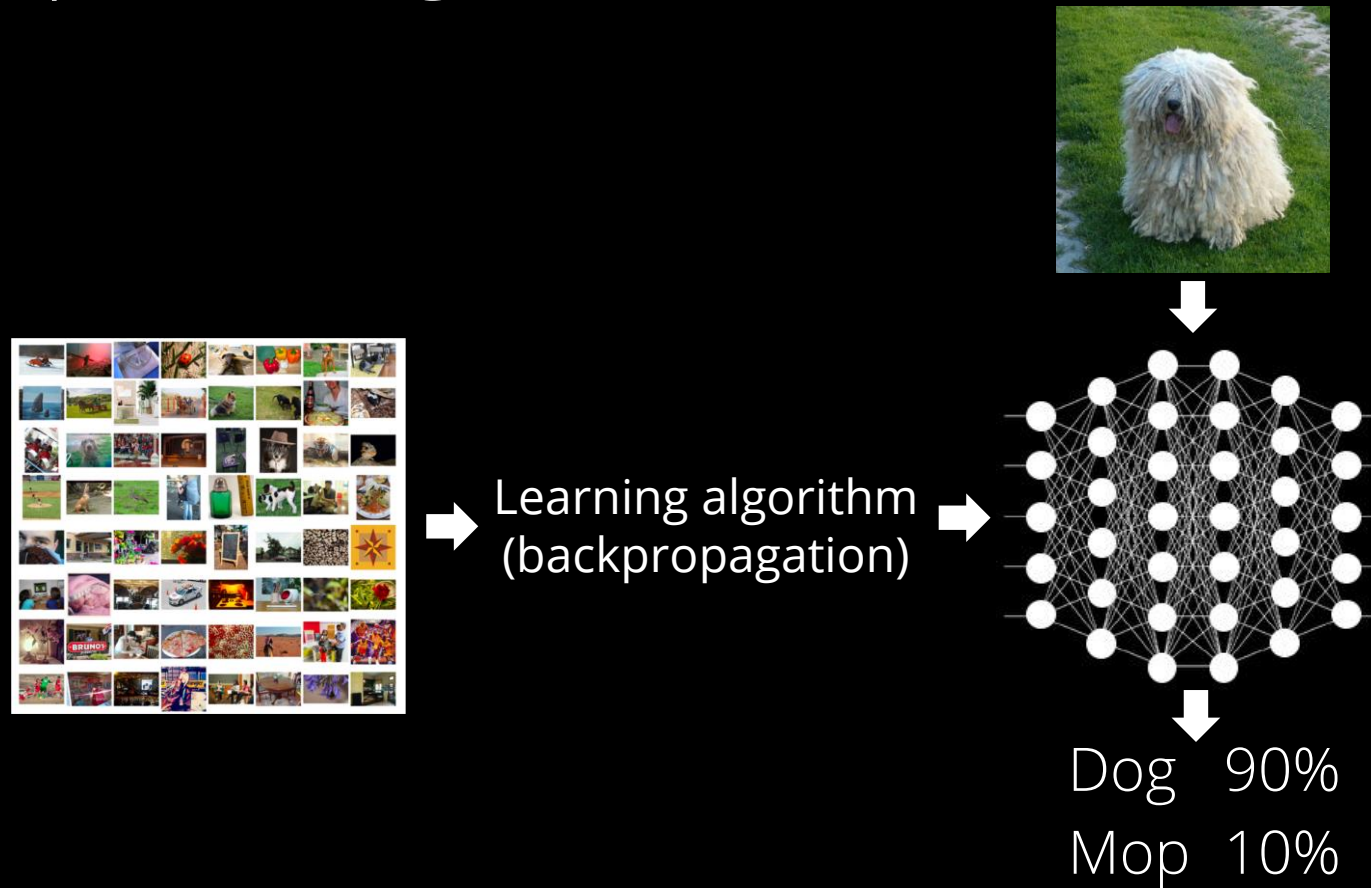


# Stochastic gradient descent (SGD)

- Claim:  $g_t = \nabla \ell_{i_t}(\theta_t)$ , where  $i_t \sim \text{Uniform}(\{1, \dots, m\})$  is an unbiased estimate of  $\nabla \mathcal{L}(\theta_t)$
- Justification: 
$$\begin{aligned}\mathbb{E}[g_t] &= \sum_{i=1}^m \mathbb{P}(i_t = i) \nabla \ell_i(\theta_t) && \text{(defn of expectation)} \\ &= \frac{1}{m} \sum_{i=1}^m \nabla \ell_i(\theta_t) && \text{(PMF of } i_t) \\ &= \nabla \left( \frac{1}{m} \sum_{i=1}^m \ell_i(\theta_t) \right) && \text{(linearity of derivative)} \\ &= \nabla \mathcal{L}(\theta_t)\end{aligned}$$
- Extension (Mini-batch SG):  $g_t = \frac{1}{k} \sum_{i \in S_t} \nabla \ell_i(\theta_t)$  ( $S_t$  is a random size- $k$  subset of  $\{1, \dots, m\}$ ) is also an unbiased estimate

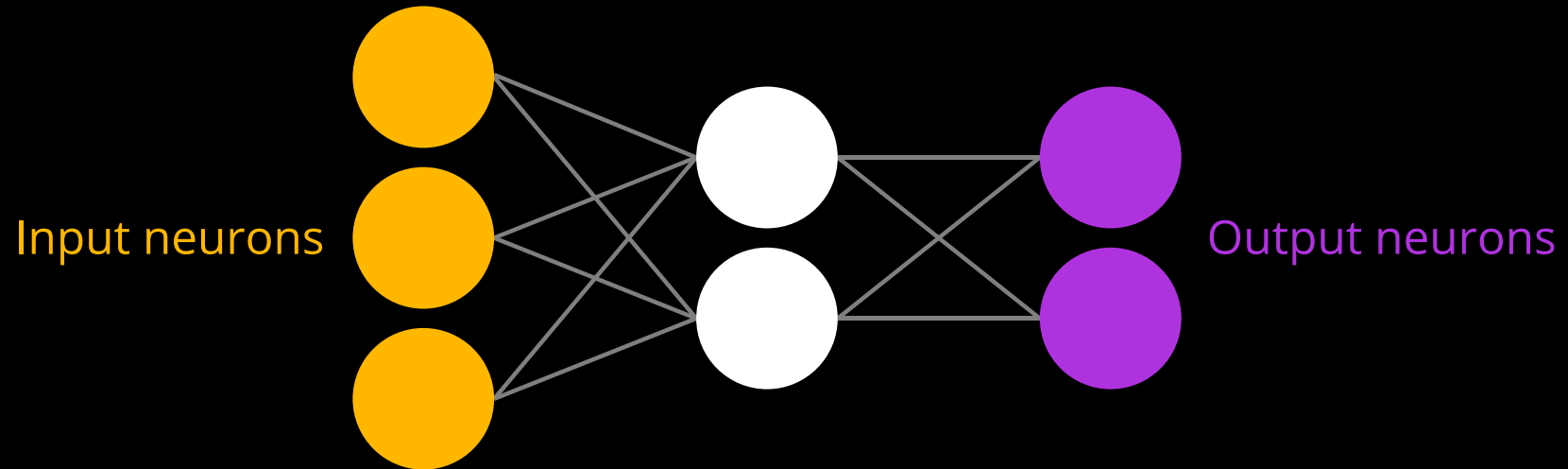


# Deep learning, a field of machine learning

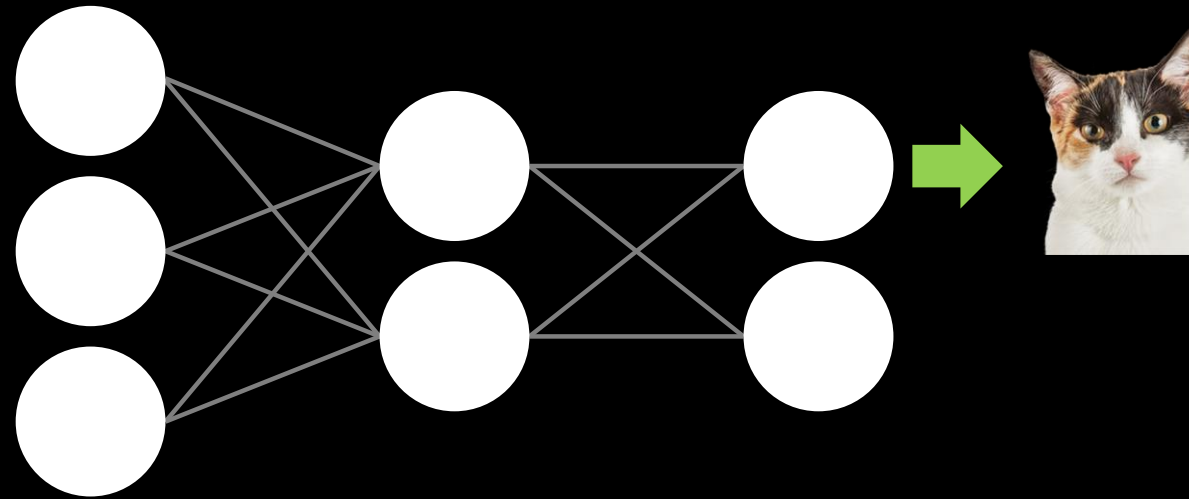




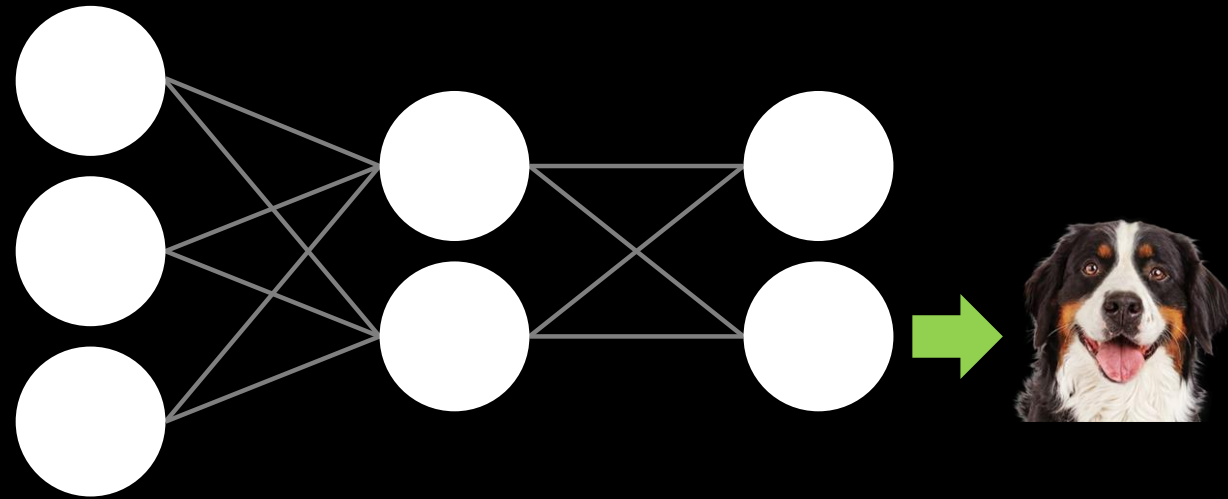
# Deep learning with backpropagation



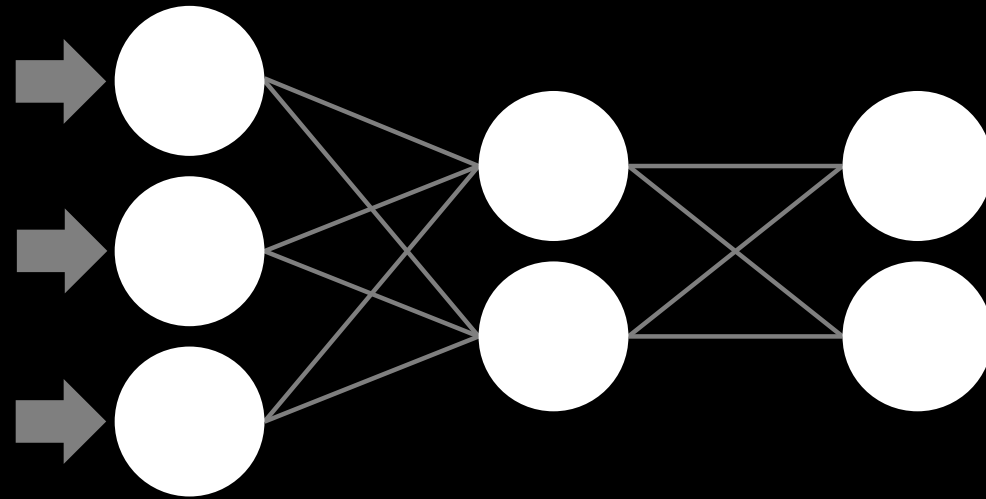
# Deep learning with backpropagation



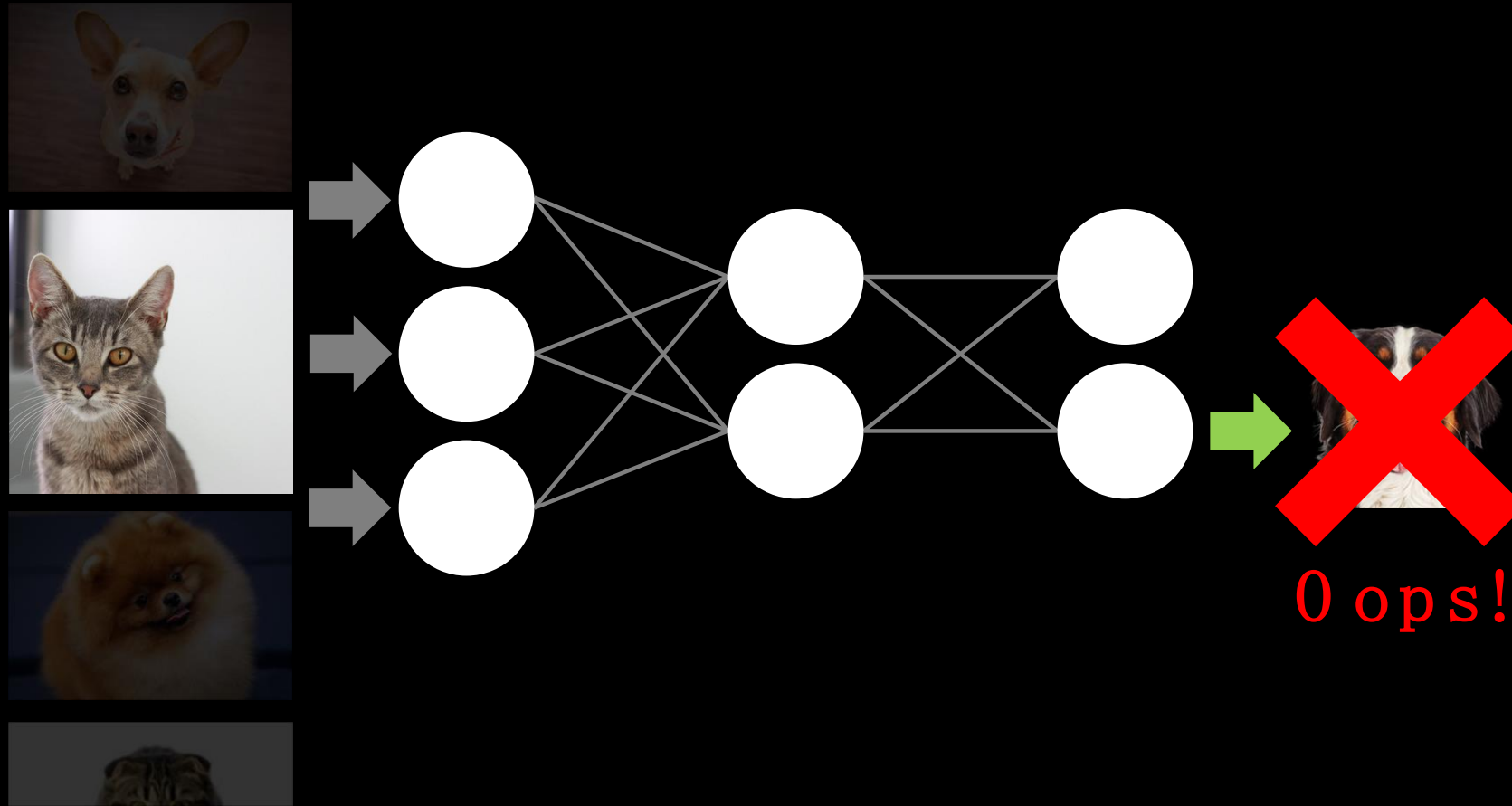
# Deep learning with backpropagation



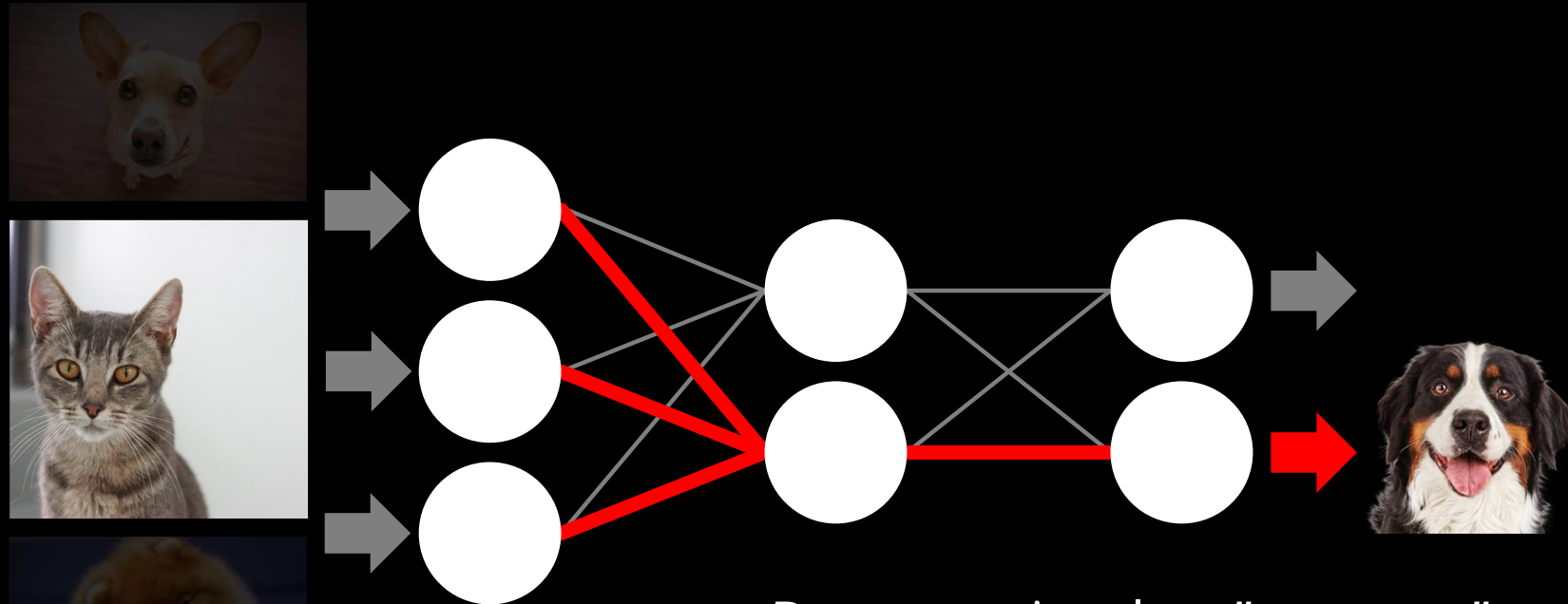
# Deep learning with backpropagation



# Deep learning with backpropagation

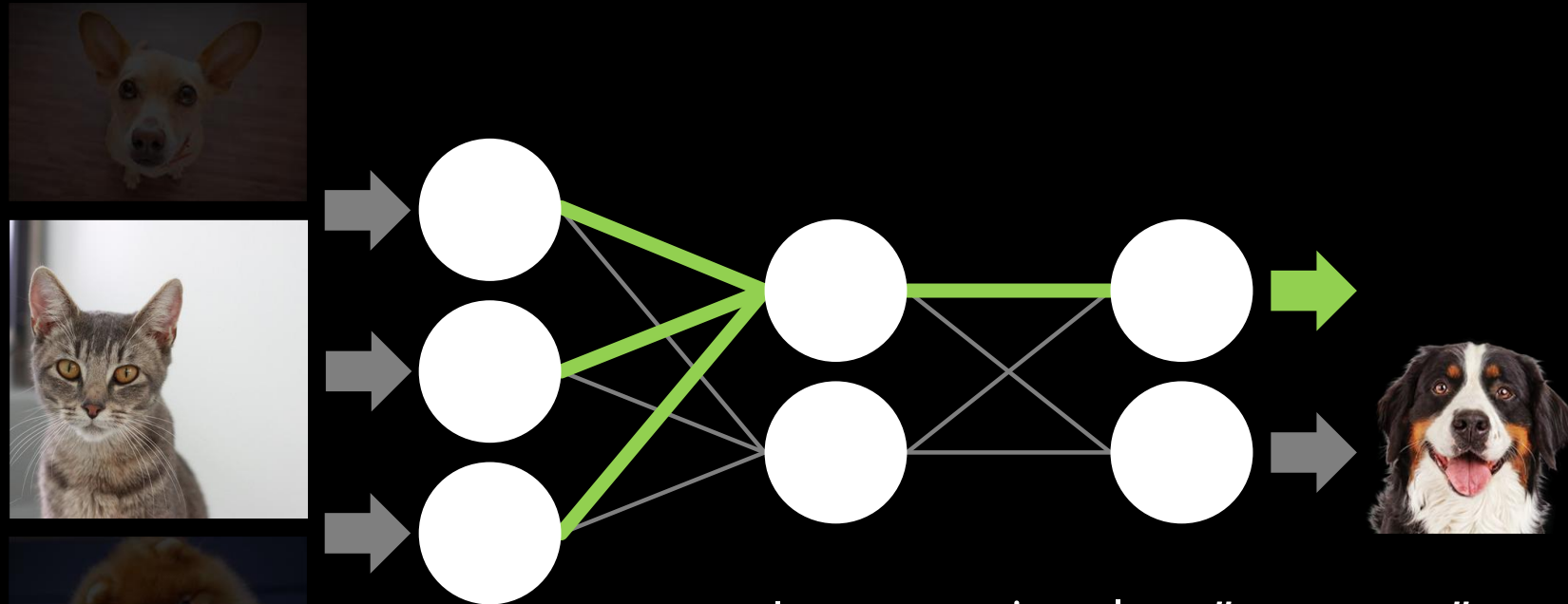


# Deep learning with backpropagation



Decrease signal on "synapses"  
that fired incorrectly!

# Deep learning with backpropagation



Increase signal on "synapses"  
that did not fire sufficiently!

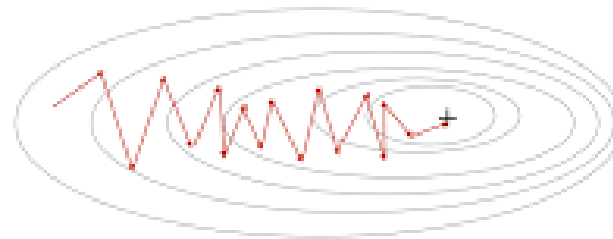
---

**Algorithm 7:** Stochastic gradient descent algorithm for the training of neural networks.

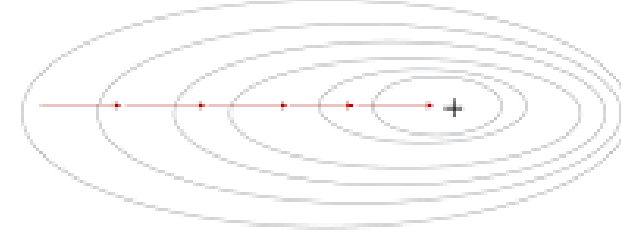
---

```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```

Stochastic Gradient Descent



Gradient Descent



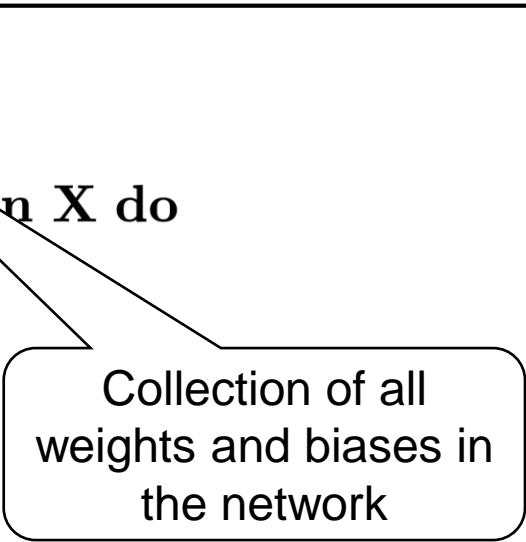


---

**Algorithm 7:** Stochastic gradient descent algorithm for the training of neural networks.

---

```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



Collection of all weights and biases in the network

---

**Algorithm 7:** Stochastic gradient descent algorithm for the training of neural networks.

---

```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $x_i$  in  $X$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



One training example

---

**Algorithm 7:** Stochastic gradient descent algorithm for the training of neural networks.

---

```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\Theta)$ 
6     end
7   end
8 end
```

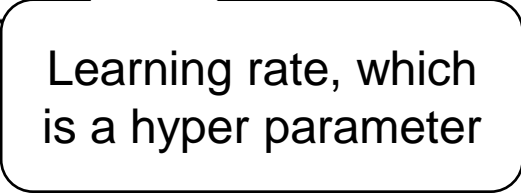
Partial derivative of the cost function  $C$  for each parameter (weight or bias) in the network

---

**Algorithm 7:** Stochastic gradient descent algorithm for the training of neural networks.

---

```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



Learning rate, which is a hyper parameter

# Computing the Derivative

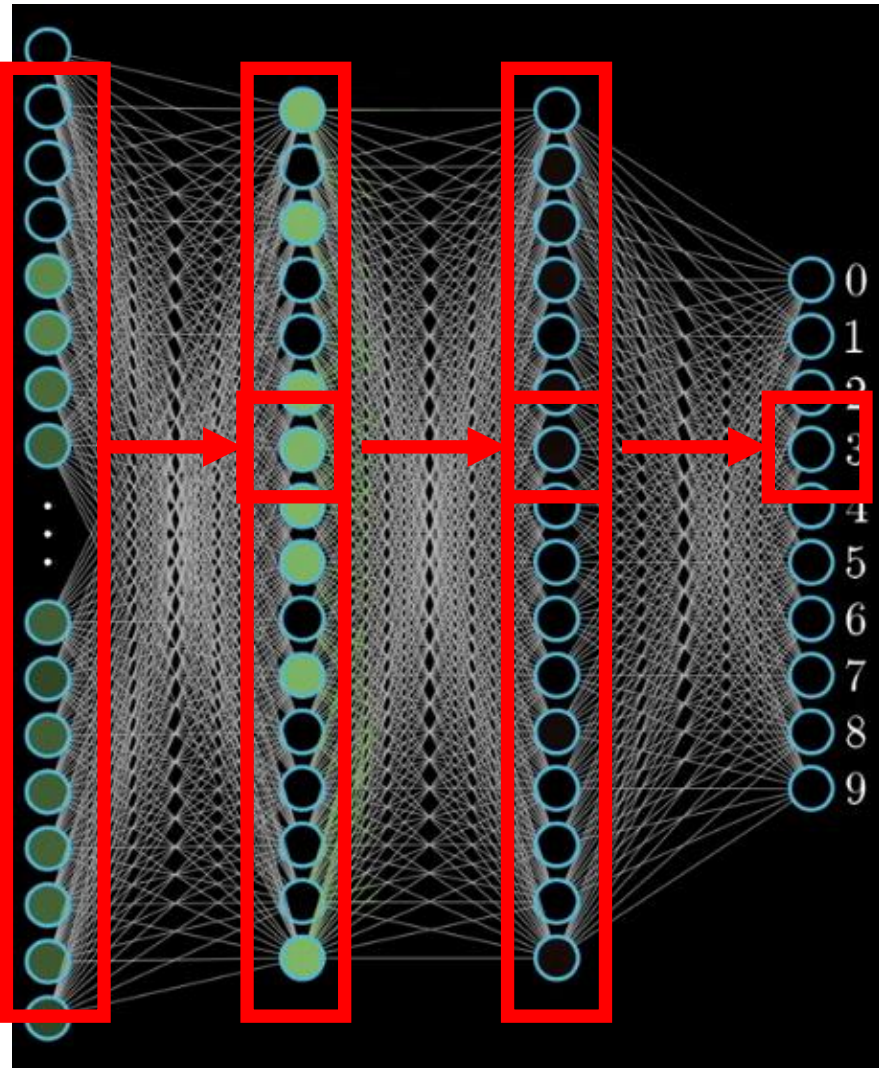
So we need to compute derivatives of a super complicated function...

$$\nabla \mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla \ell_i(\theta)$$

- Tells us how much to turn the “tuning knob” (i.e. weight)
- But how do we compute derivatives for edge weights not directly connected to the output layer?
- Key technique: Backpropagation

# Backpropagation

[ Source : 3Blue1Brown : <https://www.youtube.com/watch?v=aircAruvnKk> ]



Activation at final layer involves weighted combination of activations at previous layer...

$$z_n = \sigma(W_n z_{n-1})$$

Which involves a weighted combination of the layer before it...

$$z_{n-1} = \sigma(W_{n-1} z_{n-2})$$

And so on...

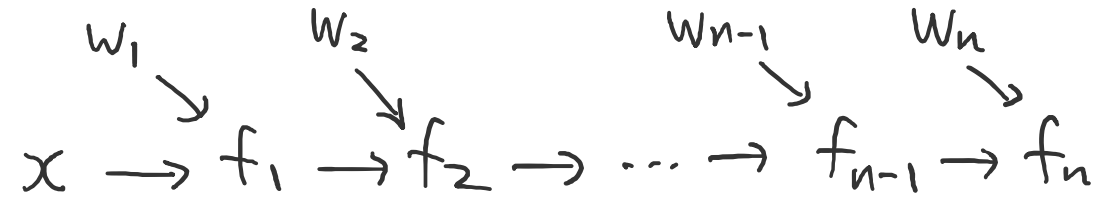
$$z_n = \sigma(W_n \sigma(W_{n-1} \sigma(\dots)))$$



# Key conceptual tool: Computation graph

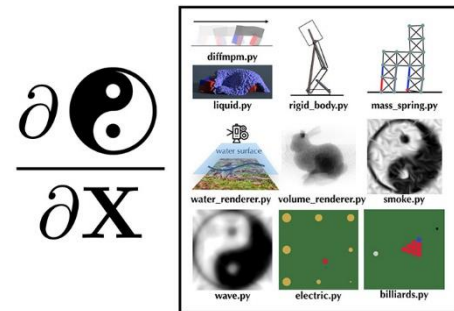
- A DAG that describes the order of computation in a general computational process

- Nodes: variables
- Edges: dependency of the variables
- $f_1 = F_1(W_1, x), \dots, f_n = F_n(W_n, f_{n-1})$



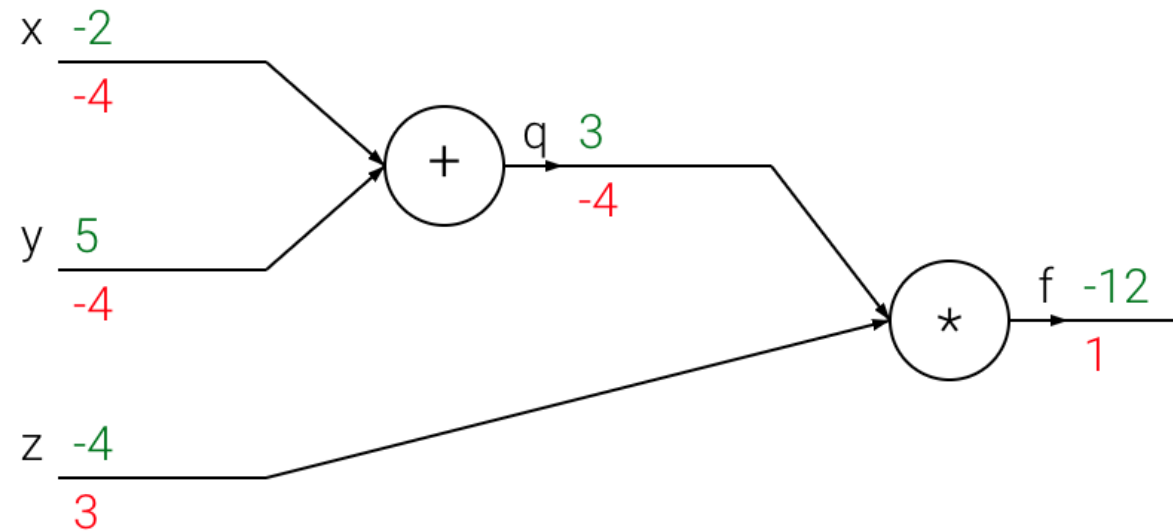
- Has more general application beyond training neural networks

- Differentiable physics simulation engine (Hu et al 2020)
- Differentiable ray-tracing (e.g. Yang et al 2022)
- ICML workshops on “differentiable almost everything”  
<https://differentiable.xyz/>



# Key tool: Computation graph

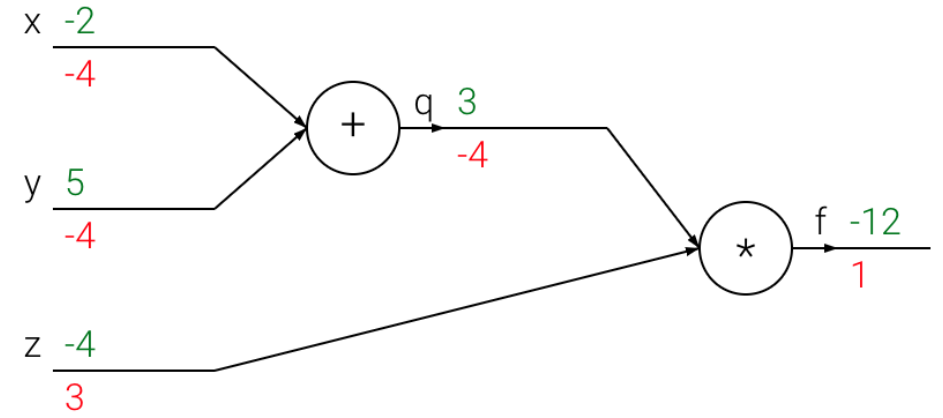
- Sometimes useful to highlight the *operation* that computes each variable as well
- E.g.  $q = F_q(x, y) = x + y$ ;  $f = F_f(q, z) = q \cdot z$



- **Backpropagation** is the procedure of repeatedly applying the derivative chain rule to compute the full derivative

# Chain rule in computation graphs: an example

- $q = F_1(x, y) = x + y$ ;  $f = F_2(q, z) = q \cdot z$
- Representing function  $F(x, y, z) = (x + y) \cdot z$



- How to calculate  $\nabla F = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$ ?
- Interpretation of  $\frac{\partial v}{\partial u}$ : if node  $u$  is changed by 1 unit *independently*, how much does  $v$  change?

- Using *reverse topological order*, go over all variables in the graph

- $\frac{\partial f}{\partial q} = -4, \quad \frac{\partial f}{\partial z} = 3$

- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = -4$

- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = -4$

green: node values  
red: derivatives

# Backpropagation

- In practice, neural network implementations use *auto differentiation* to compute the derivative on-the-fly
- Can do this efficiently on *graphical processing units (GPUs)* on extremely large training datasets

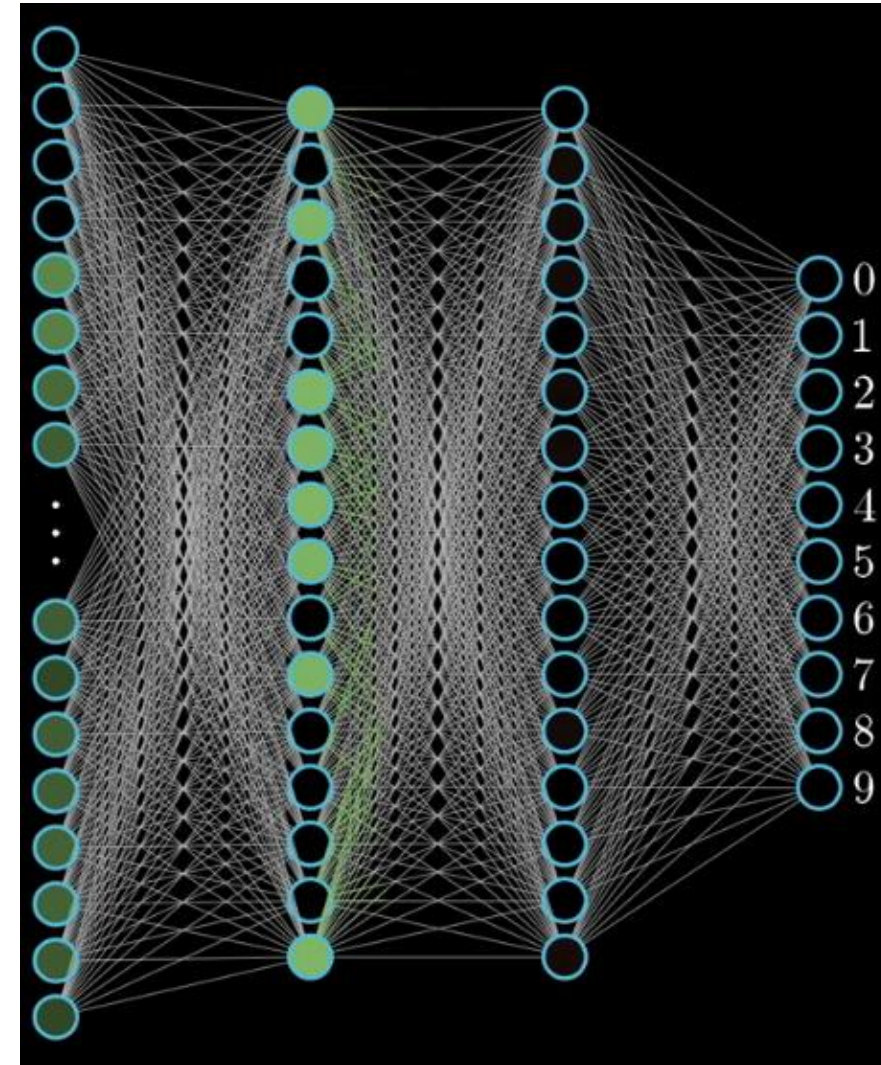
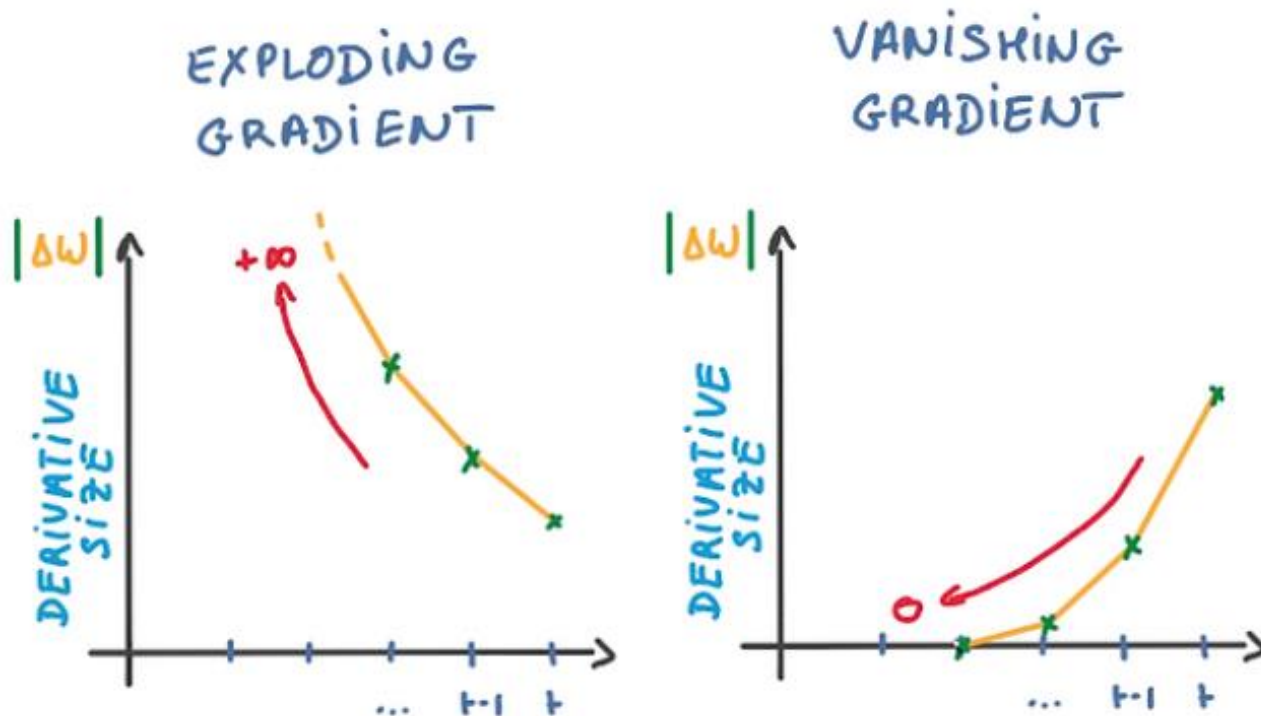


"Did you compute your gradients correctly?" — Leon Bottou.

(Taken from Matus Telgarsky's deep learning lecture: <https://mjt.cs.illinois.edu/ml/lec8.pdf>)

# Vanishing / exploding gradient problems

- Experimental observation:  $\frac{\partial f}{\partial w_i}$  for  $w_i$  in closer-to-input layers are more likely to be tiny (vanishing) / huge (exploding), leading to problematic updates
- We'll see NN designs that mitigates this





# Expressive power of neural networks

- (Cybenko, 1989; Hornik et al, 1989)

**Theorem 10** (Two-Layer Networks are Universal Function Approximators). *Let  $F$  be a continuous function on a bounded subset of  $D$ -dimensional space. Then there exists a two-layer neural network  $\hat{F}$  with a finite number of hidden units that approximate  $F$  arbitrarily well. Namely, for all  $\mathbf{x}$  in the domain of  $F$ ,  $|F(\mathbf{x}) - \hat{F}(\mathbf{x})| < \epsilon$ .*

- Does this mean that there is no benefit in learning deeper networks? No..

## The Power of Depth for Feedforward Neural Networks

Ronen Eldan, Ohad Shamir

We show that there is a simple (approximately radial) function on  $\mathbb{R}^d$ , expressible by a small 3-layer feedforward neural networks, which cannot be approximated by any 2-layer network, to more than a certain constant accuracy, unless its width is exponential in the dimension. The result holds for virtually all known activation functions, including rectified linear units, sigmoids and

## Benefits of depth in neural networks

Matus Telgarsky

For any positive integer  $k$ , there exist neural networks with  $\Theta(k^3)$  layers,  $\Theta(1)$  nodes per layer, and  $\Theta(1)$  distinct parameters which can not be approximated by networks with  $\mathcal{O}(k)$  layers unless they are exponentially large --- they must possess  $\Omega(2^k)$  nodes. This result is proved here for a class of nodes termed "semi-algebraic gates" which includes the common choices of ReLU, maximum,



# Regularization

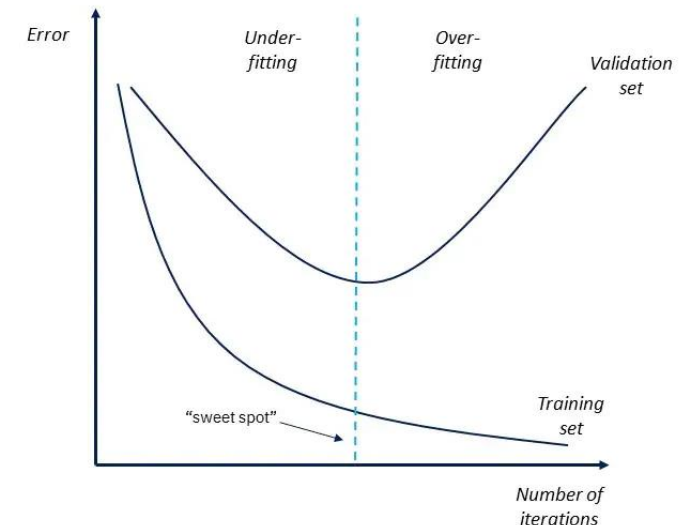
# Regularization

*With four parameters I can fit an elephant. With five I can make him wiggle his trunk.* - John von Neumann

$$w = \arg \min_w \text{Cost}(w) + \alpha \cdot \text{Regularizer}(\text{Model})$$

Our example model has 13,002 parameters...that's a lot of elephants!  
Regularization is useful...

...numerous regularization schemes are used in training neural networks



# L2 Regularization

Formalize the regularized cost function as,

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta})$$

Consider an L2 penalty,

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Gradient (derivative) with respect to  $\mathbf{w}$  is given by,

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Take a single step in the direction of the gradient,

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon (\alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}))$$

# L2 Regularization (Weight Decay)

Written another way, a single gradient step is:

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon \nabla_w J(w; \mathbf{X}, \mathbf{y})$$

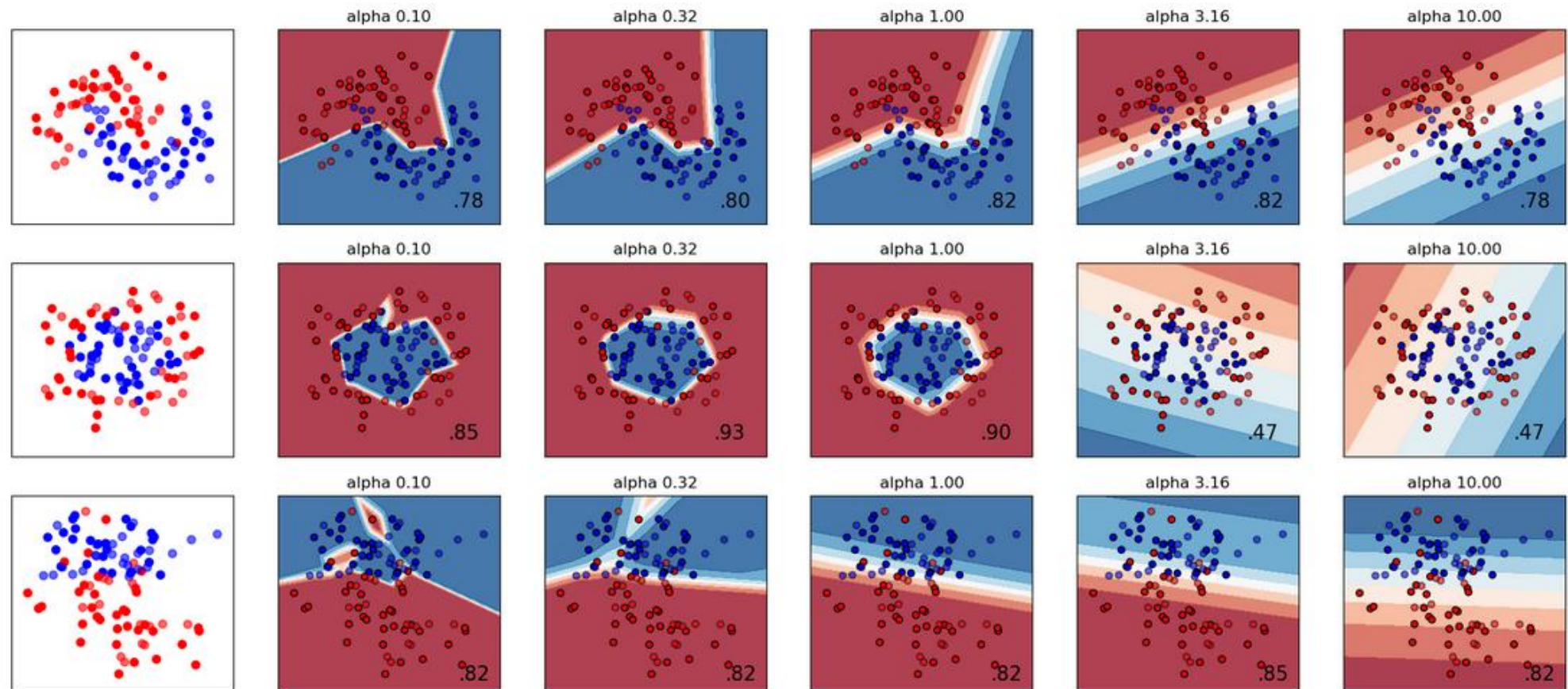
Learning Rate  
(how big of a step)

Regularization  
Strength (Coefficient)

- Can see this is a modification to the learning rule (gradient descent)
- “Shrinks” the weight by constant factor on each step
- Then perform usual gradient step

# Regularization : Weight Decay

$$w = \arg \min_w \text{Cost}(w) + \frac{\alpha}{2} \|w\|^2$$



# L1 Regularization

$$\tilde{J}(w) = J(w) + \alpha \|w\|_1$$

(Sub-)gradient given by,

$$\nabla_w \tilde{J}(w; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(w) + \nabla_w J(\mathbf{X}, \mathbf{y}; w)$$

- Very different effect from L2 weight decay
- Regularization contribution no longer scales linearly with each  $w$
- Constant addition with sign equal to  $\text{sign}(w)$
- Has a *sparsity-inducing property* (encourages some weights to be 0)



# Parameter Tying / Sharing

- Introduces inductive bias
  - There should be dependencies among parameters
  - Parameters should be close / similar
- Hard constraints force sets of parameters to be equal
  - Known as *parameter sharing*
  - Only subset of unique parameters needs to be stored in memory
- Example: convolutional neural network (we will discuss in detail)

# Dataset Augmentation

- Train on more data (always more data)
- What if we don't have more data? (Make up more)
- Easiest for classification
- Generate new  $(x,y)$  pairs by transforming an existing  $(x,y)$
- Particularly effective for object recognition
  - Translation
  - Scaling
  - Rotation
  - ...

# Dataset Augmentation



Affine Distortion



Noise



Elastic Deformation



Horizontal flip



Random Translation



Hue Shift



# Dataset Augmentation

- Need to avoid transformations that change class
- For example mirror “b” to produce “d”
- Rotation turns “6” into “9”
- Some transformations are not easy to perform, e.g. out-of-plane rotation

# Learning Curves – Early Stopping

Early stopping: terminate when validation set performance stops improving

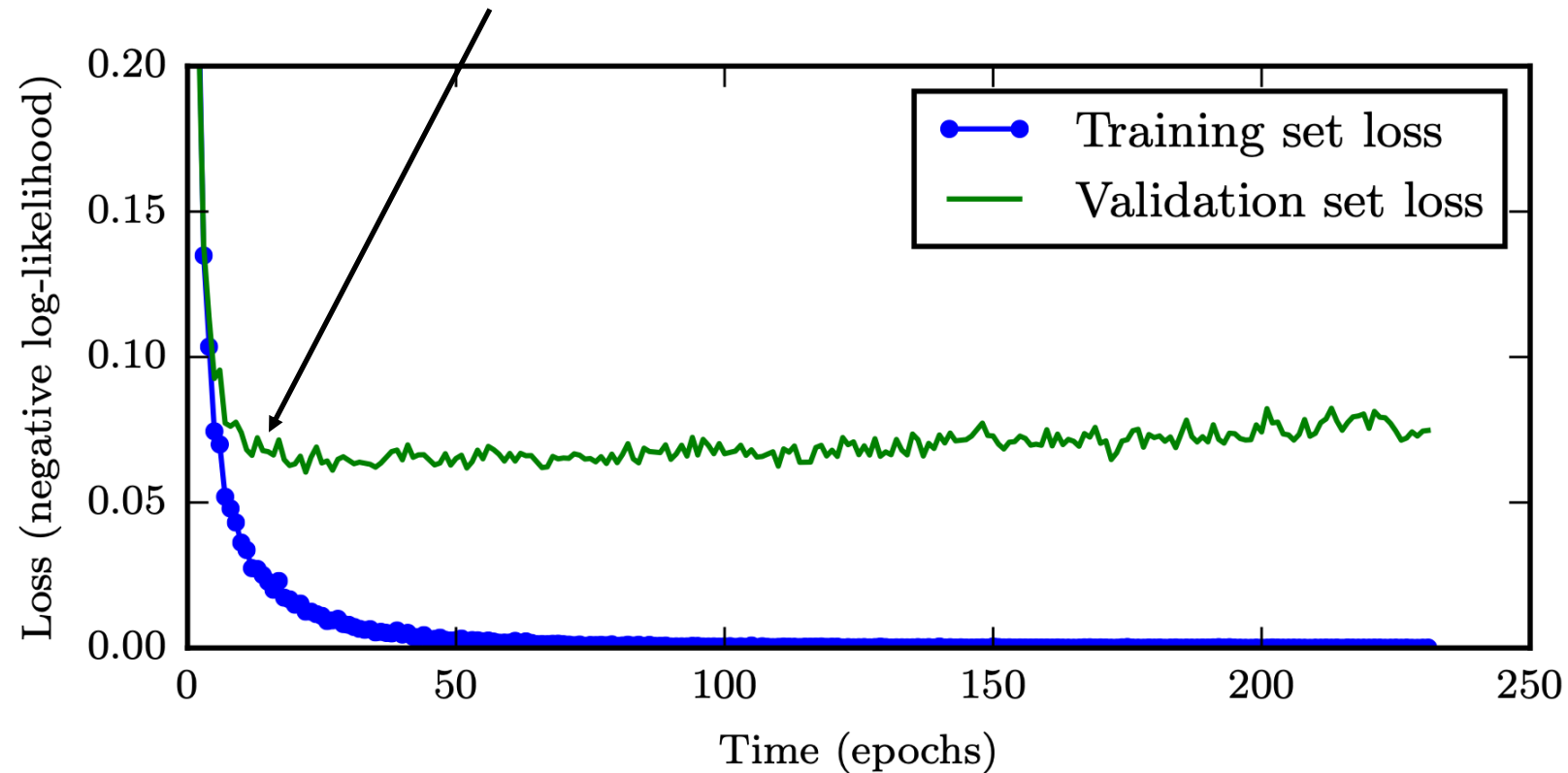


Figure 7.3

**Algorithm 7.1** The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

---

Let  $n$  be the number of steps between evaluations.

Let  $p$  be the “patience,” the number of times to observe worsening validation set error before giving up.

Let  $\theta_o$  be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

**while**  $j < p$  **do**

    Update  $\theta$  by running the training algorithm for  $n$  steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

**if**  $v' < v$  **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

**else**

$j \leftarrow j + 1$

**end if**

**end while**

Best parameters are  $\theta^*$ , best number of training steps is  $i^*$ .

---



# Early Stopping

- Think of it as efficient hyperparameter selection algorithm (hyperparameter = number of training steps)
- Requires almost no change to underlying training procedure
  - Contrast with weight decay that requires hyperparameter tuning

# Regularization

- L1+L2 (elastic net) regularization
- **Data Augmentation** Synthetically expand training data by applying random transformations
- **Early stopping** Just as it sounds...stop the network before reaching a local minimum...simple-but-effective
- **Dropout** Each iteration randomly selects a small number of edges to temporarily exclude from the network (weights=0)

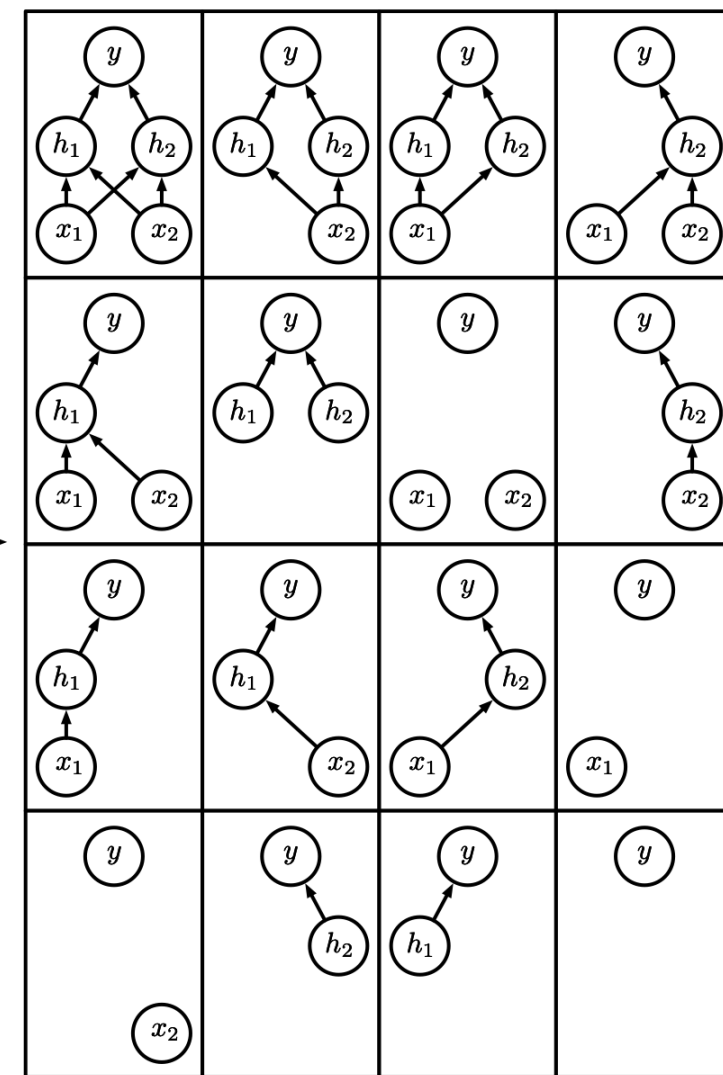
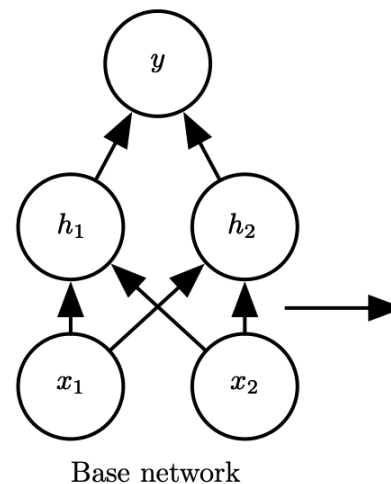
# Dropout (Srivastava et al, 2014)

Each time we load a minibatch to perform updates:

- randomly remove set of nodes (& associated edges)
- do updates on the remaining network

Includes input and hidden nodes – typically different probabilities of dropping each

Figure 7.6



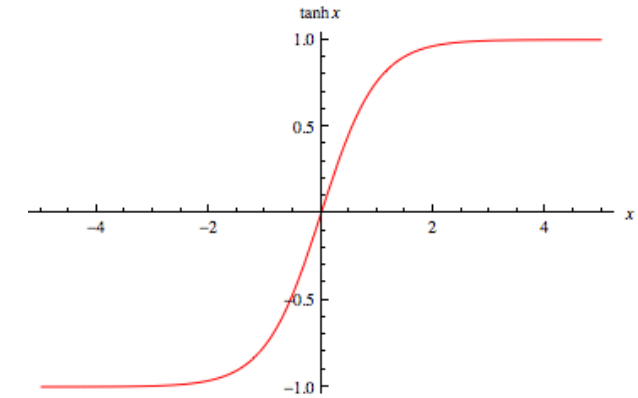
Ensemble of subnetworks

# Dropout

- Srivastava et al. (2014) showed more effective than weight decay and other “simple” regularization methods
- Computationally very cheap
- Doesn't significantly limit type of model that can be used
- Can slow training and require larger model sizes
- Less effective when very few training examples available

# Batch normalization (Ioffe and Szegedy, 2015)

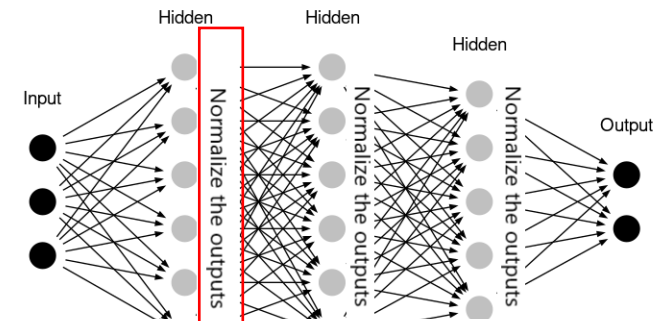
- **Fact:** optimization is easier when the inputs of each layer is within constant interval, say  $[-2,2]$
- Can we “enforce” this by modifying the NN’s design?
- Key idea: Let’s add a layer that normalizes (i.e., standardizes) the inputs!
- Recall minibatch SGD
  - Computes gradients for  $k$  data points, then updates the weights.
  - Can we ensure that within a batch, for a layer, most of the inputs are “standardized”?



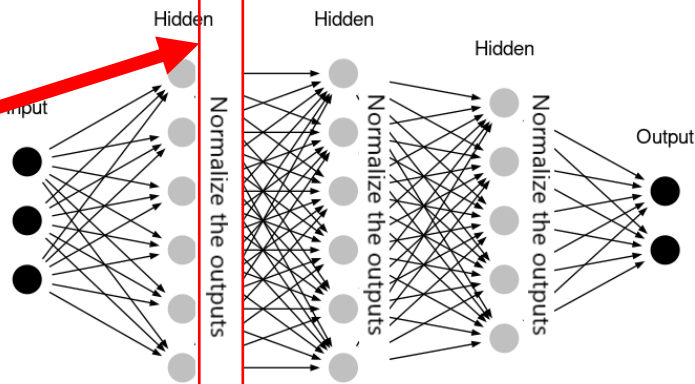
# Batch normalization layer

- Example: neural network making predictions on a batch of  $M=3$  examples with batch normalization

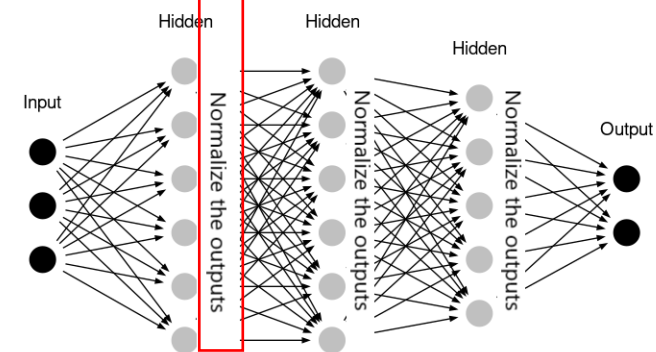
Example 1



Example 2



Example 3



Using all examples from the batch to normalize the activations

# Batch normalization layer

- Example: neural network making predictions on a batch of  $M=3$  examples with batch normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

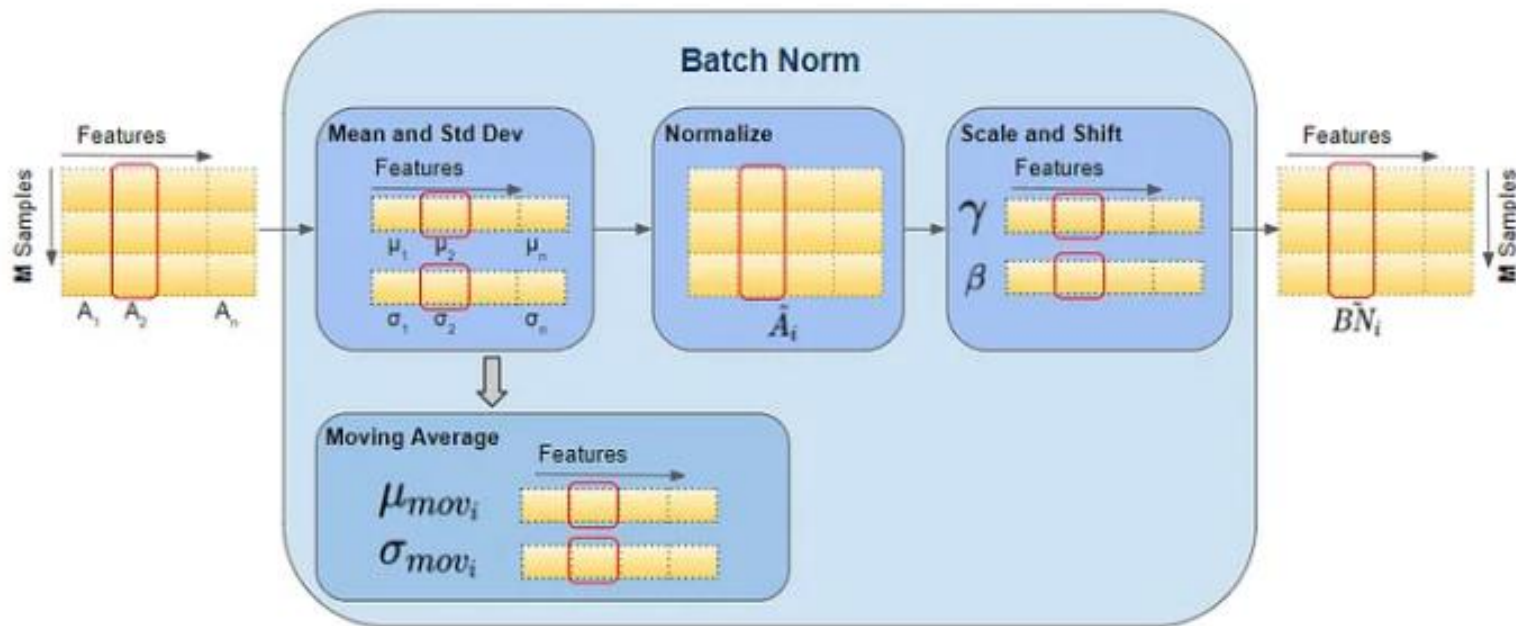
**Output:**  $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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



**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.



# Batch normalization

- One twist for the test time
  - The network is now well-trained to predict batches of  $M$  test examples
  - What if we want to use the network to predict a *single* test example  $x$ ?
  - Use (averaged)  $\mu_B$  and  $\sigma_B^2$  computed in training time
- Why does batch normalization help?
  - (Santurkar et al, 2018): it helps making the NN optimization landscape smoother
- Popular variants: layer normalization (Ba et al, 2016)
  - widely used in transformer/ LLMs

# Example

Play with a small multilayer perceptron on a binary classification task...

<https://playground.tensorflow.org/>

# sklearn.neural\_network.MLPClassifier

**hidden\_layer\_sizes** : *tuple, length = n\_layers - 2, default=(100,)*

The *i*th element represents the number of neurons in the *i*th hidden layer.

**activation** : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

**solver** : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

**alpha** : *float, default=0.0001*

L2 penalty (regularization term) parameter.

**learning\_rate** : *{'constant', 'invscaling', 'adaptive'}, default='constant'*

Learning rate schedule for weight updates.

**early\_stopping** : *bool, default=False*

Whether to use early stopping to terminate training when validation score is not improving. If set to true,

# Scikit-Learn : Multilayer Perceptron

Fetch MNIST data from [www.openml.org](http://www.openml.org) :

```
X, y = fetch_openml("mnist_784", version=1, return_X_y=True)
X = X / 255.0
```

Train test split (60k / 10k),

```
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]
```

Create MLP classifier instance,

- Single hidden layer (50 nodes)
- Use stochastic gradient descent
- Maximum of 10 learning iterations
- Small L2 regularization  $\alpha=1e-4$

```
mlp = MLPClassifier(
    hidden_layer_sizes=(50,),
    max_iter=10,
    alpha=1e-4,
    solver="sgd",
    verbose=10,
    random_state=1,
    learning_rate_init=0.1,
)
```

# Scikit-Learn : Multilayer Perceptron

Fit the MLP and print accuracy...

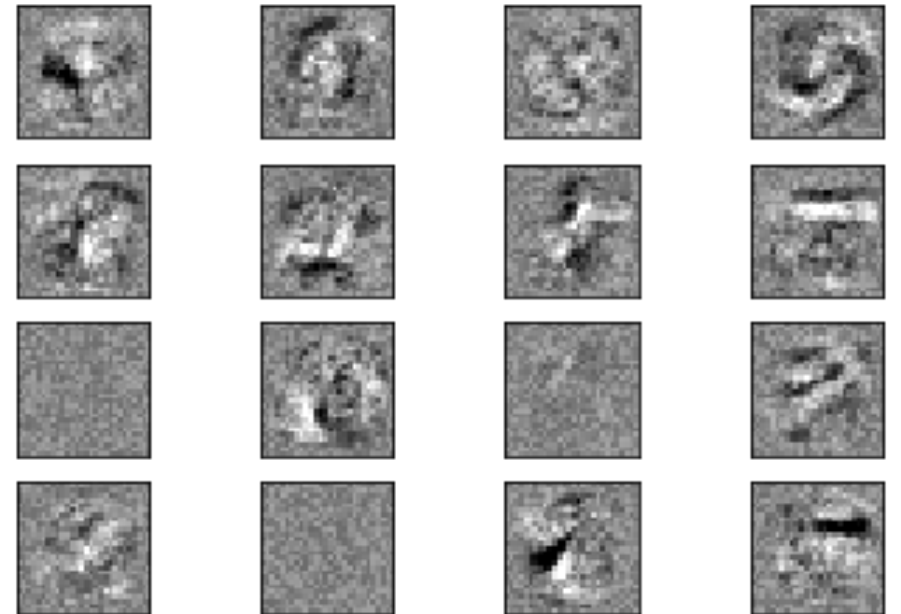
```
mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

Visualize the weights for each node...

```
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray,
               vmin=0.5 * vmin, vmax=0.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
```

...magnitude of weights indicates which input features are important in prediction



# Convolutional Neural Networks

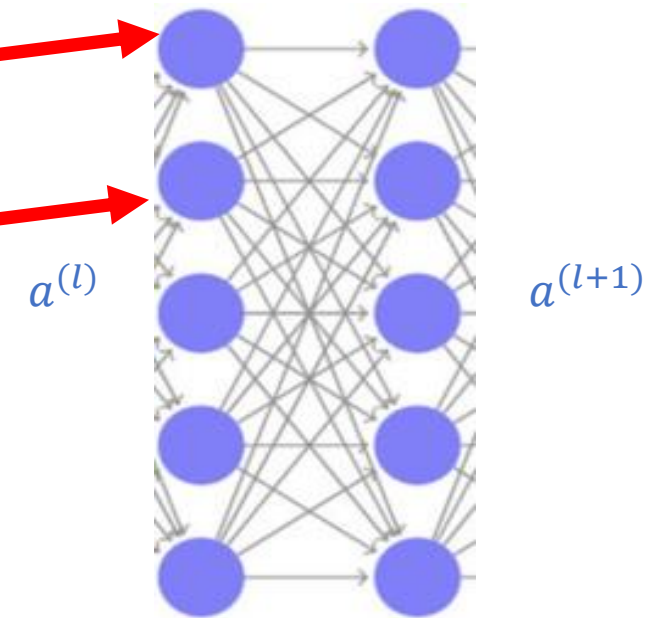
# NNs for images

- FCs can learn (pattern, location) combinations in images
  - The learned patterns do not generalize to different spatial locations.

Neuron 1: detects faces around (124, 236)

Neuron 2: detects flowers around (34, 301)

....





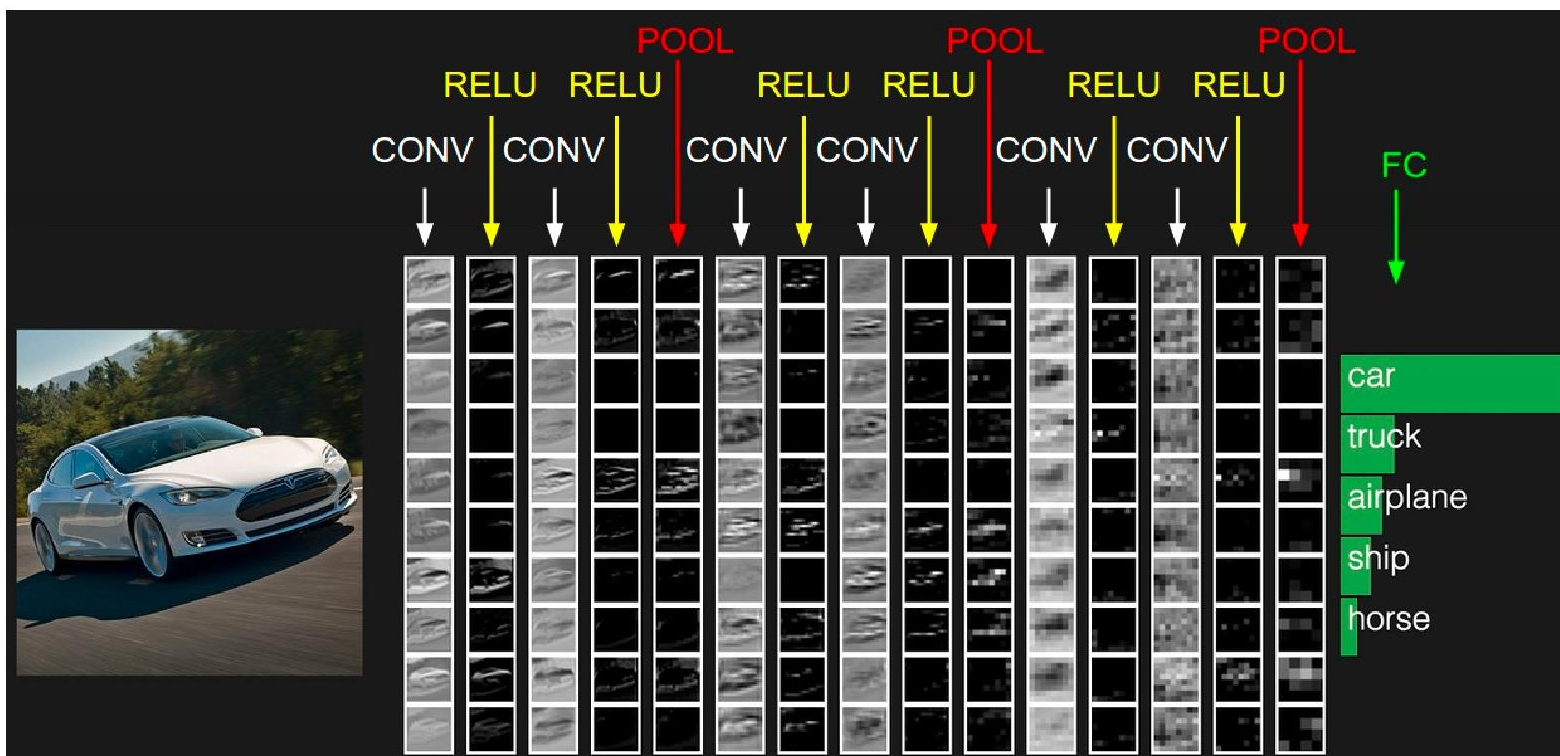
# NNs for images

- Can we learn a group of neurons that detect a certain pattern (e.g. *existence* of a wheel) at all locations?
  - low level: edge of some orientation, a patch of some color
  - high level: shape of a wheel, face
- Encodes inductive bias
  - Image classification: semantic of an image should be translation-invariant



# Convolutional neural networks (CNN)

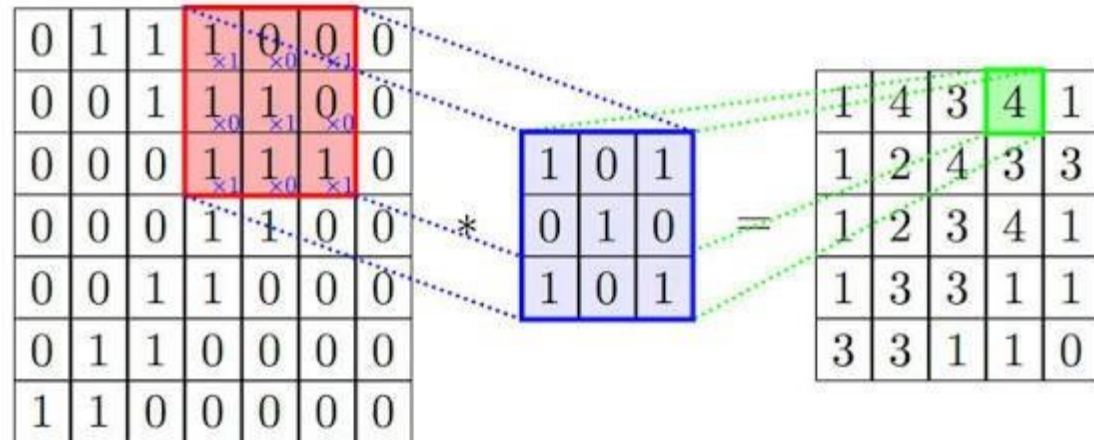
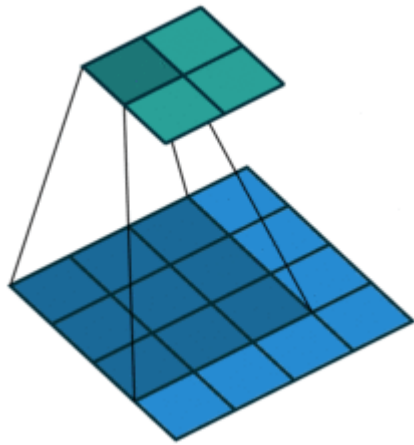
- A.K.A. ConvNet architecture
- A set of neural network architecture that consists of
  - convolutional layers
  - pooling layers
  - fully-connected (FC) layers



# Convolution for single-channel images

Consider one filter with weights  $\{w_{i,j}\}$  with size  $F \times F$

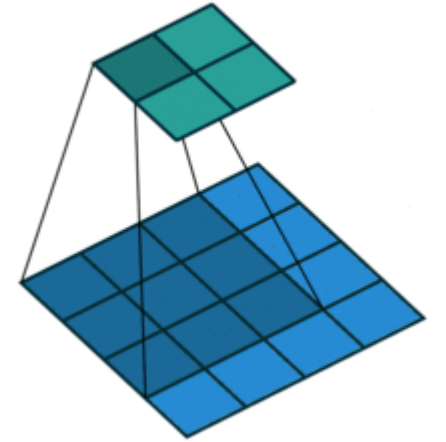
- For every  $F \times F$  region of the image, perform inner product (= element wise product, then sum them all)
- Q: given a  $w \times h$  image, after convolution with a  $F \times F$  filter, what is the size of the resulting image?
- Terminologies: filter size, receptive field size, kernel.



# Convolution: Some Intuition

Define the convolution of filter  $f$  on image  $I$  as:

$$(f * I)(x) = \sum_m \sum_n I(x + m, y + n) f(m, n)$$



In signal processing, people use another convention for defining convolution:

$$(f * I)(x) = \sum_m \sum_n I(x - m, y - n) f(m, n)$$

*A good filter detects interesting patterns in images*

*Learning finds good values for the convolution filter...*

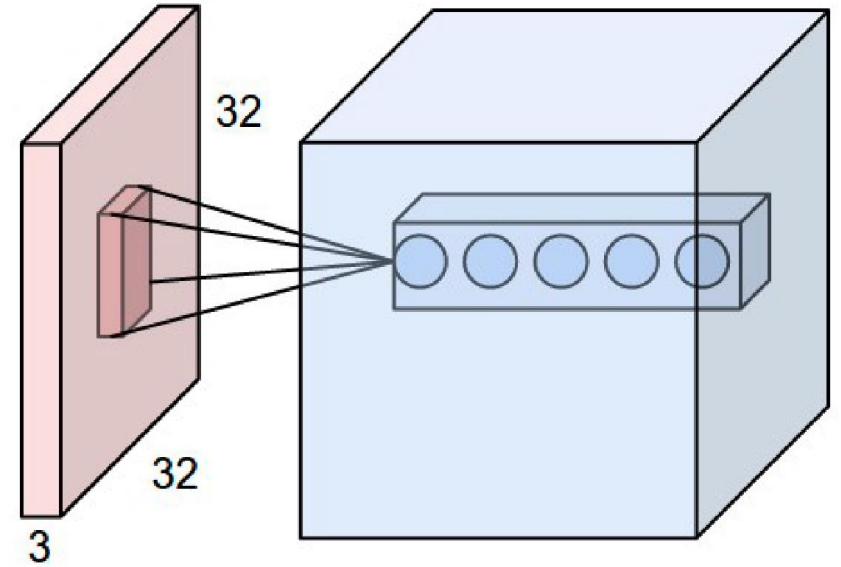
# Convolutional layer for multi-channel images

Input:  $w$  (width)  $\times$   $h$  (height)  $\times$   $c$  (#channels)

- E.g.  $32 \times 32 \times 3$
- $3$  channels: R, G, and B

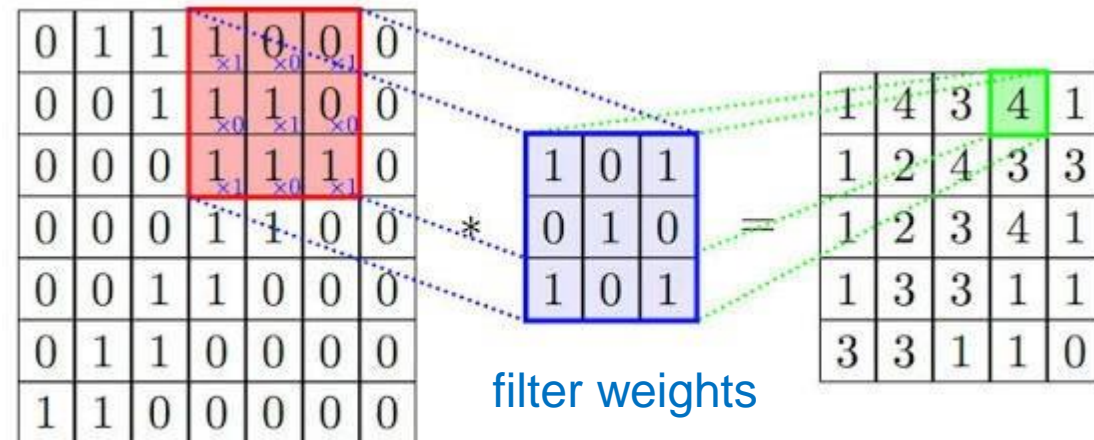
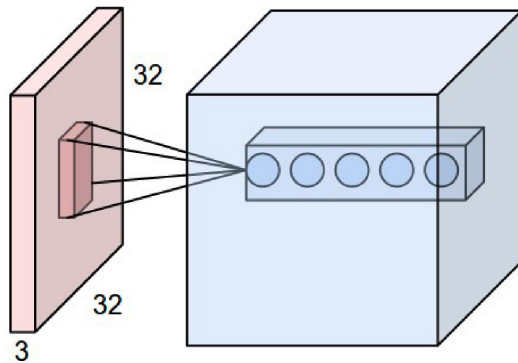
A convolutional filter on such image is of shape  $F \times F \times c$

- Only spatial structure in the first two dimensions
- Denoted by  $\{w_{i,j,k}\}$



# Convolutional layer: multichannel images

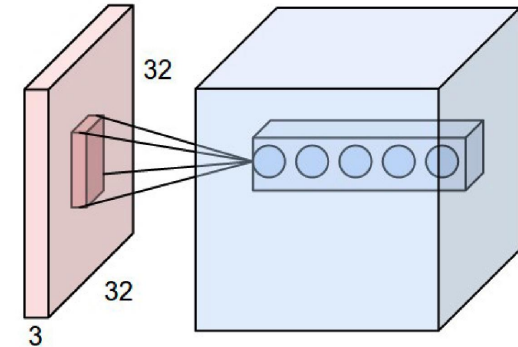
- Consider one filter with weights  $\{w_{i,j,k}\}$  with  $5 \times 5 \times 3$ 
  - Imagine a sliding 3D window.
  - **Convolution:** For every  $5 \times 5$  region of the image, perform inner product (= element wise product, then sum them all)
  - Then apply the activation function (e.g., ReLU)
- Results in  $28 \times 28 \times 1$  – called **activation map**.
- Now, we can do  $K$  of these filters but with different weights  $\{w_{i,j,k}^{(\ell)}\}$  for  $\ell \in [K] \Rightarrow$  output is  $28 \times 28 \times K$





# Convolutional layers act as feature extractors

- Example filters learned at the first conv layer
- Each filter has size  $11 \times 11 \times 3$
- Many filters look like edge detectors





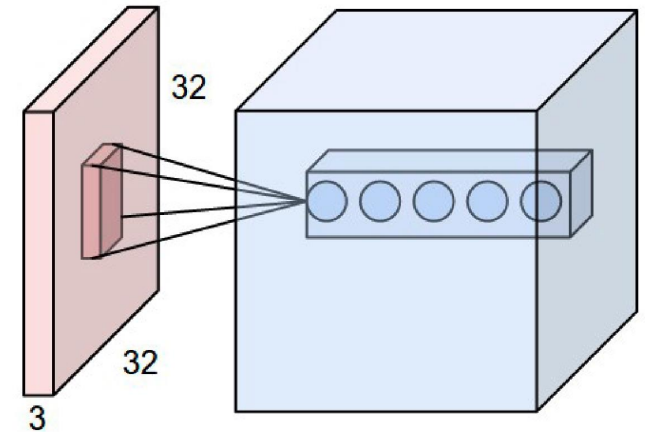
# Convolutional Layers beyond the First Layer

Generalization: conv layer as the 2<sup>nd</sup> layer or more

- Input **volume** (3d object with size  $w \times h \times d$ ):
  - the  $d$  (called **depth**) is not necessarily 3
- Output **volume**: size  $w' \times h' \times d'$ , where  $d'$  is the number of filters at the current layer.

Interpretation: patterns over the patterns.

- Each filter now convolves and combines  $d'$  activation maps for each spatial location.
- e.g., combinations of particular edges and textures



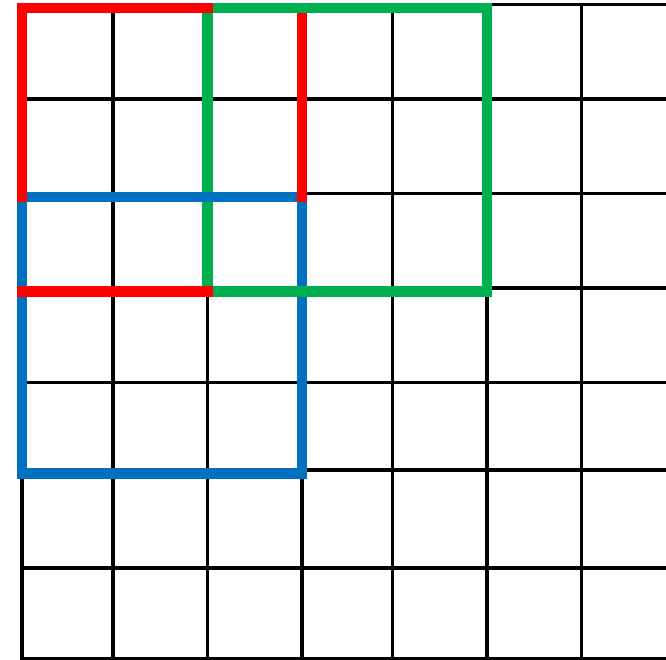
# Convolutional Layer: More Details

## Stride length $S$

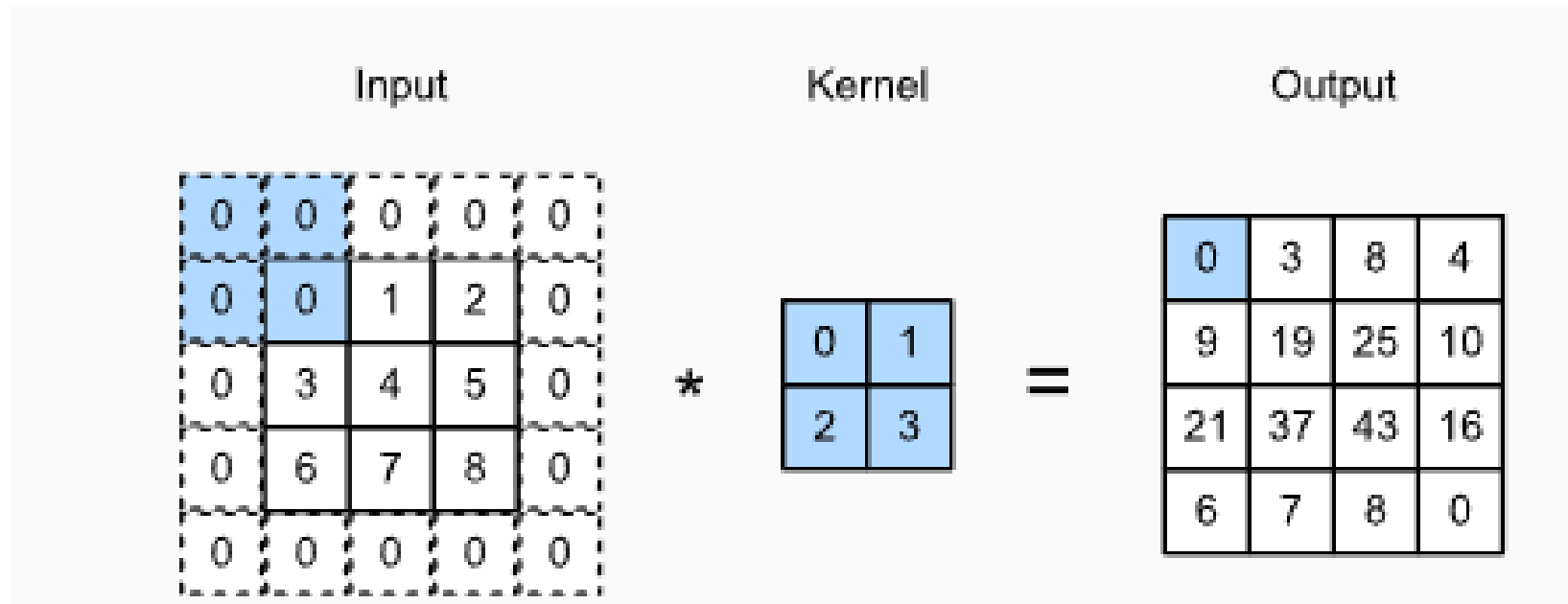
- Skip input regions; Move the sliding window of a filter not by 1 but by  $S$ .
- E.g.,  $S=2$  means skipping every other 5 by 5 region.

Zero-padding  $P$ : add  $P$  number of artificial pixels with value 0 around the input image on both sides

- E.g. given a 28 x 28 image,  $P=1 \Rightarrow$  pad it to a 30 x 30 image
- To ensure the spatial dimension is maintained (otherwise, patterns at the corners are not detected well)

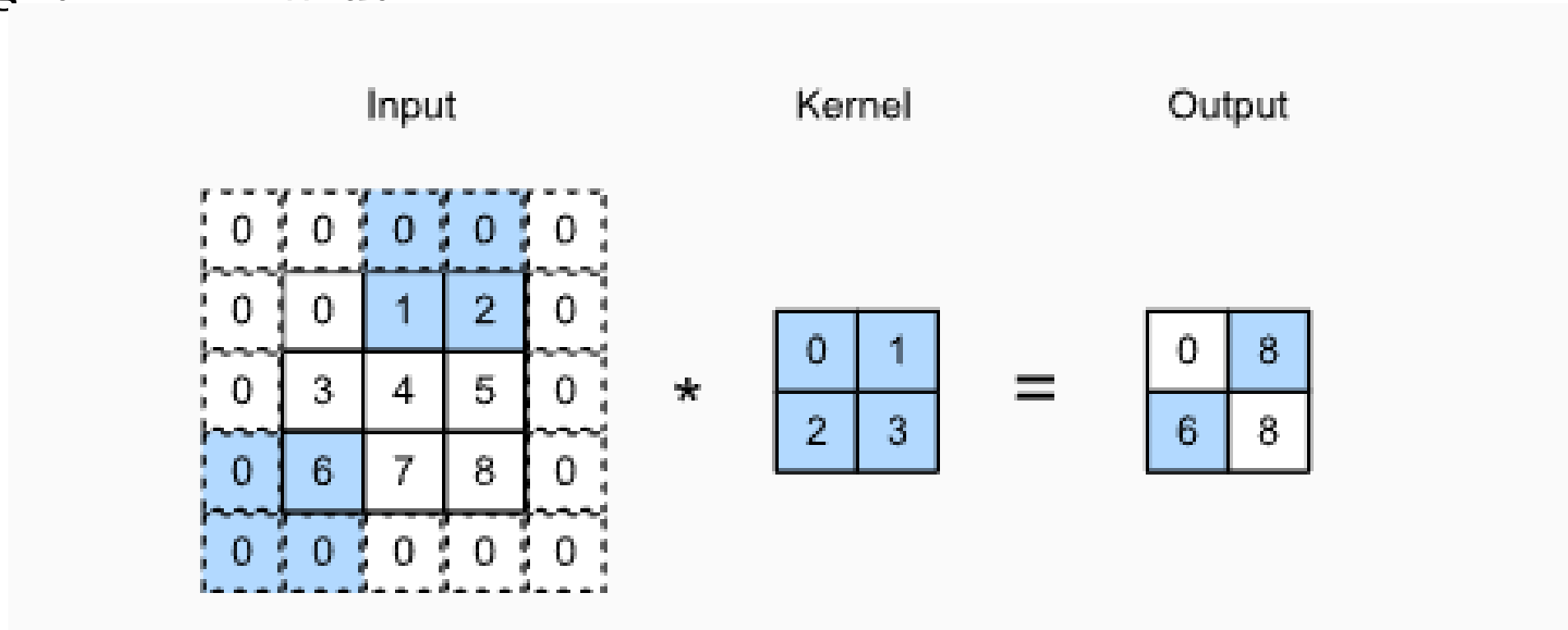


# Example: P=1, S=1



# Example: $P=1$ , different strides in height & width

- $S_{\text{height}} = 3, S_{\text{width}} = 2$



# Convolutional Layer: More Details

## Stride length $S$

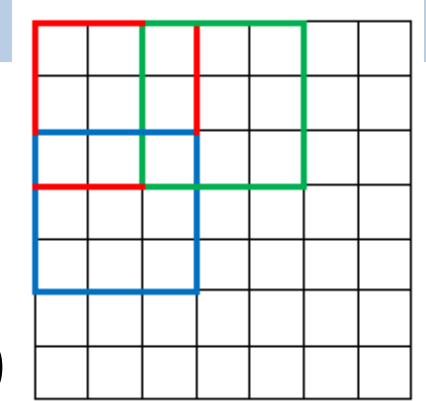
- Skip input regions; Move the sliding window of a filter not by 1 but by  $S$ .

Zero-padding  $P$ : add  $P$  number of artificial pixels with value 0 input image on both sizes

## Dimension Rules (same goes for **height**)

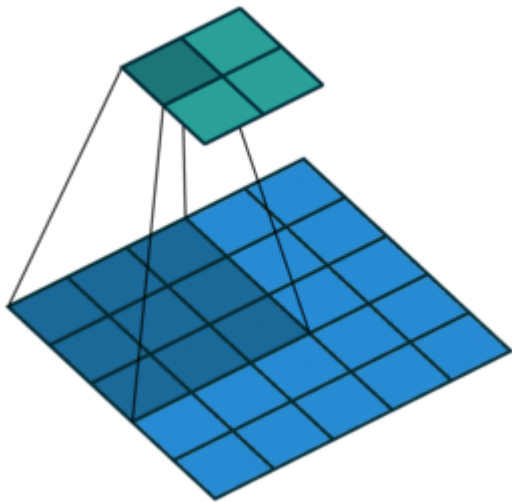
- $W$ : input volume **width**,  $F$ : filter **width**
- The output **width**  $\left\lfloor \frac{W-F+2P}{S} \right\rfloor + 1$
- E.g.,  $W=32, F=5, P=0, S=1 \Rightarrow K = 28$
- E.g.,  $W=32, F=5, P=2, S=1 \Rightarrow K = 32$

(usually, the filter has the same width and height)

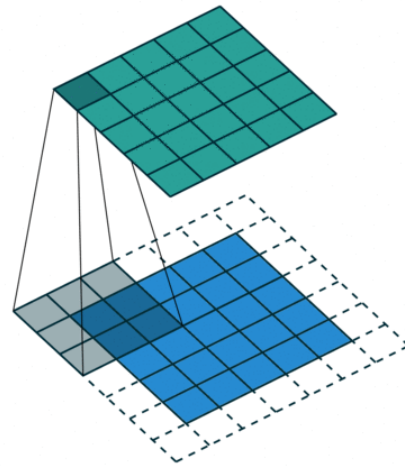


# Strides and padding: animations

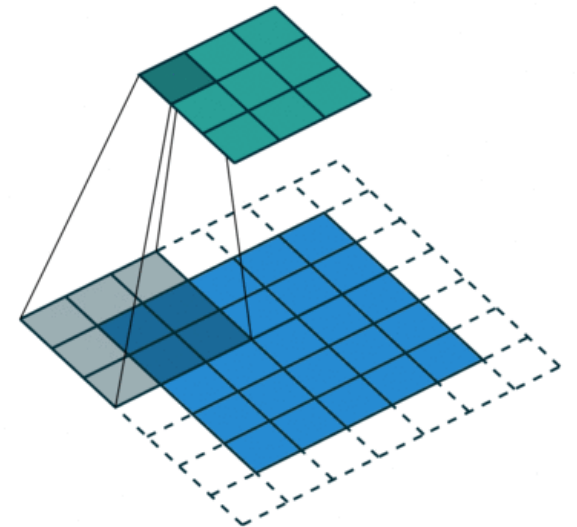
Strides only



Padding only



Strides + Padding



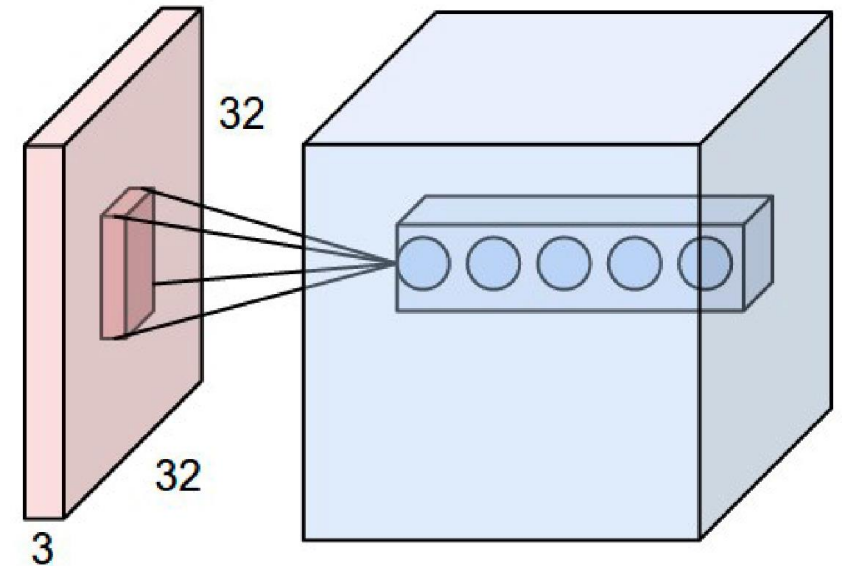
# Convolutional Layer: Summary

**Input**  $W_1 \times H_1 \times D_1$  (width, height, depth)

**Hyperparameters** # of filters  $K$ , filter size (=width=height)  $F$ , stride  $S$ , zero-padding  $P$

**Output**  $W_2 \times H_2 \times D_2$

$$W_2 = \left\lfloor \frac{W_1 - F + 2P}{S} \right\rfloor + 1,$$
$$H_2 = \left\lfloor \frac{H_1 - F + 2P}{S} \right\rfloor + 1,$$
$$D_2 = K$$



More terminology: depth slice (W by H by 1), depth column (1 by 1 by D)



# Final project report

- Credits will be given when you make and document your honest trials
- If you failed to implement something, please explain **what you have done to find an answer** and **where you get stuck**.
  - DON'T: "I don't know how to implement X"
  - DO: "I read material A, and there is this package B that seems to help, but when I tried to apply, C became an issue. ..."

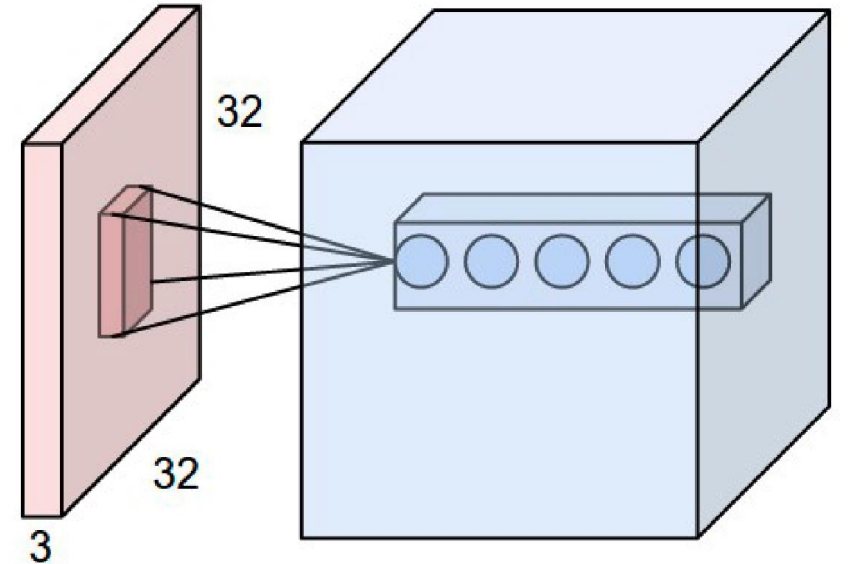
# Convolutional Layer: Summary

**Input**  $W_1 \times H_1 \times D_1$  (width, height, depth)

**Hyperparameters** # of filters  $K$ , filter size (=width=height)  $F$ , stride  $S$ , zero-padding  $P$

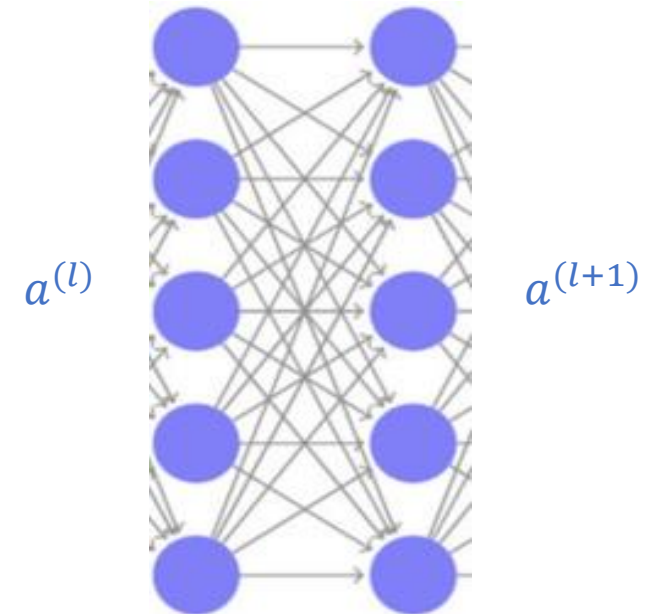
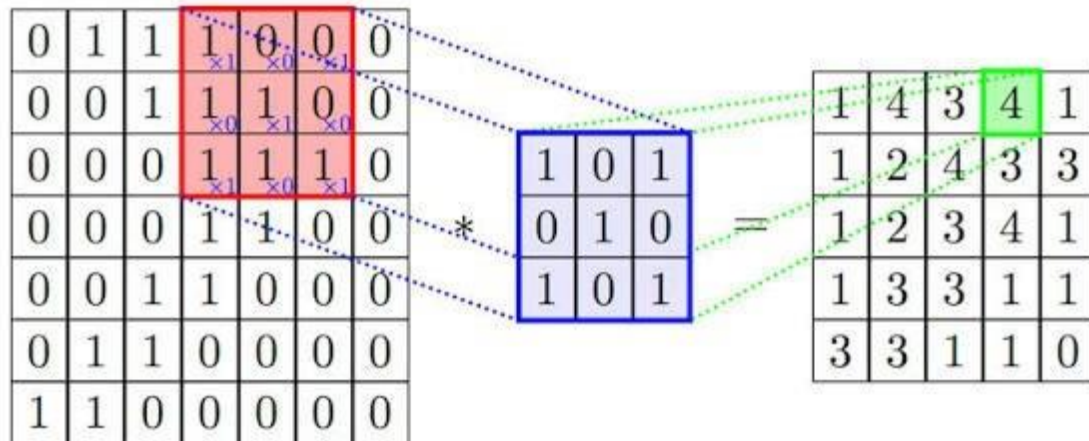
**Output**  $W_2 \times H_2 \times D_2$

$$W_2 = \left\lfloor \frac{W_1 - F + 2P}{S} \right\rfloor + 1,$$
$$H_2 = \left\lfloor \frac{H_1 - F + 2P}{S} \right\rfloor + 1,$$
$$D_2 = K$$



# Comparison: FC vs Conv

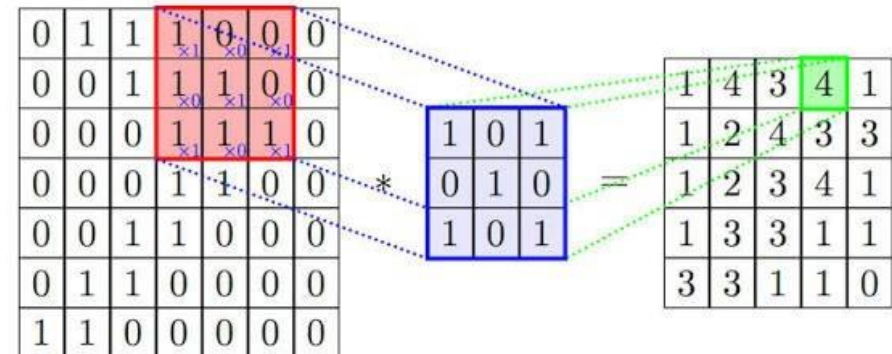
- Conv layer allows *parsimonious* representations:
  - Inter-layer connections are **local**
  - **parameter is shared** across spatial locations.
  - imposing **inductive bias** specialized for images



# Case study: first layer of AlexNet (Krizhevsky et al '12)

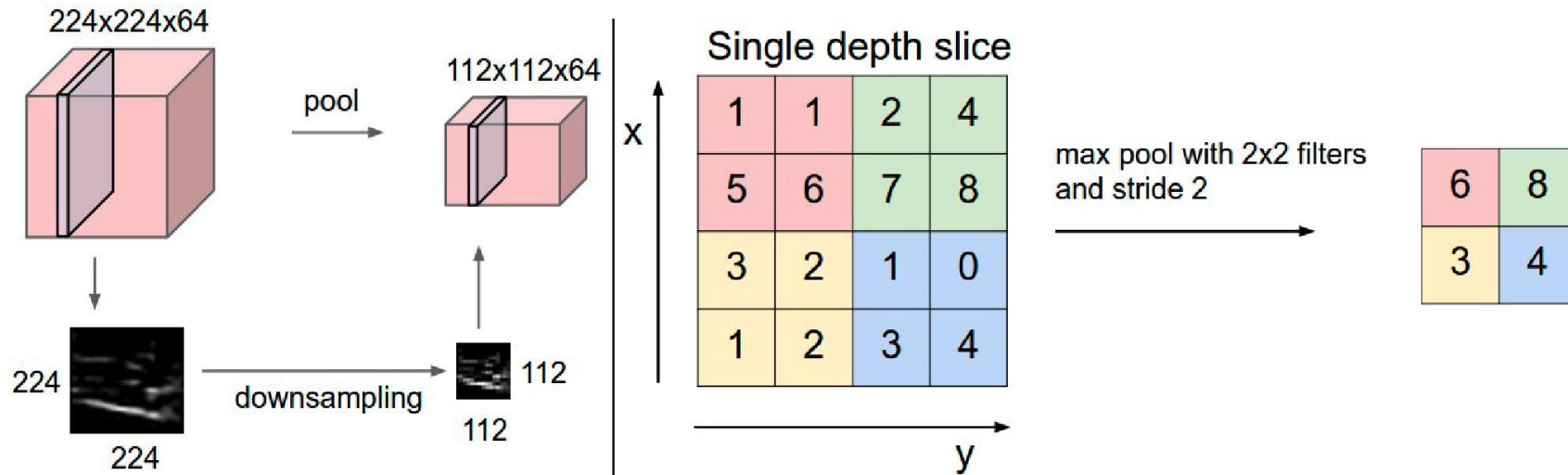
Input: 227x227x3, and the first conv layer output is 55x55x96 (96 filters)

- Each filter has  $11 \times 11 \times 3$  weights with 1 bias  $\Rightarrow$  364 parameters
- $364 \times 96 = \underline{34K}$  total parameters are used to compute the output  $55 \times 55 \times 96 = \underline{290,400}$
- What if we didn't do **parameter sharing**? I.e., for each location of image, use independent filter parameter  $w$ .
  - #params =  $290,400 \times 364 = 105M$
- What if we use FC to compute the same number of outputs? (the parsimony of **local connections**)
  - #params =  $230,187 \times 290,400 = 66B$



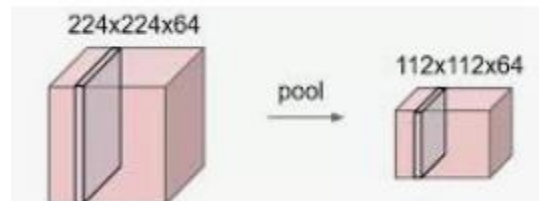
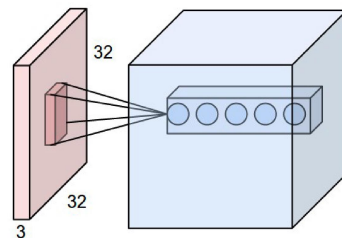
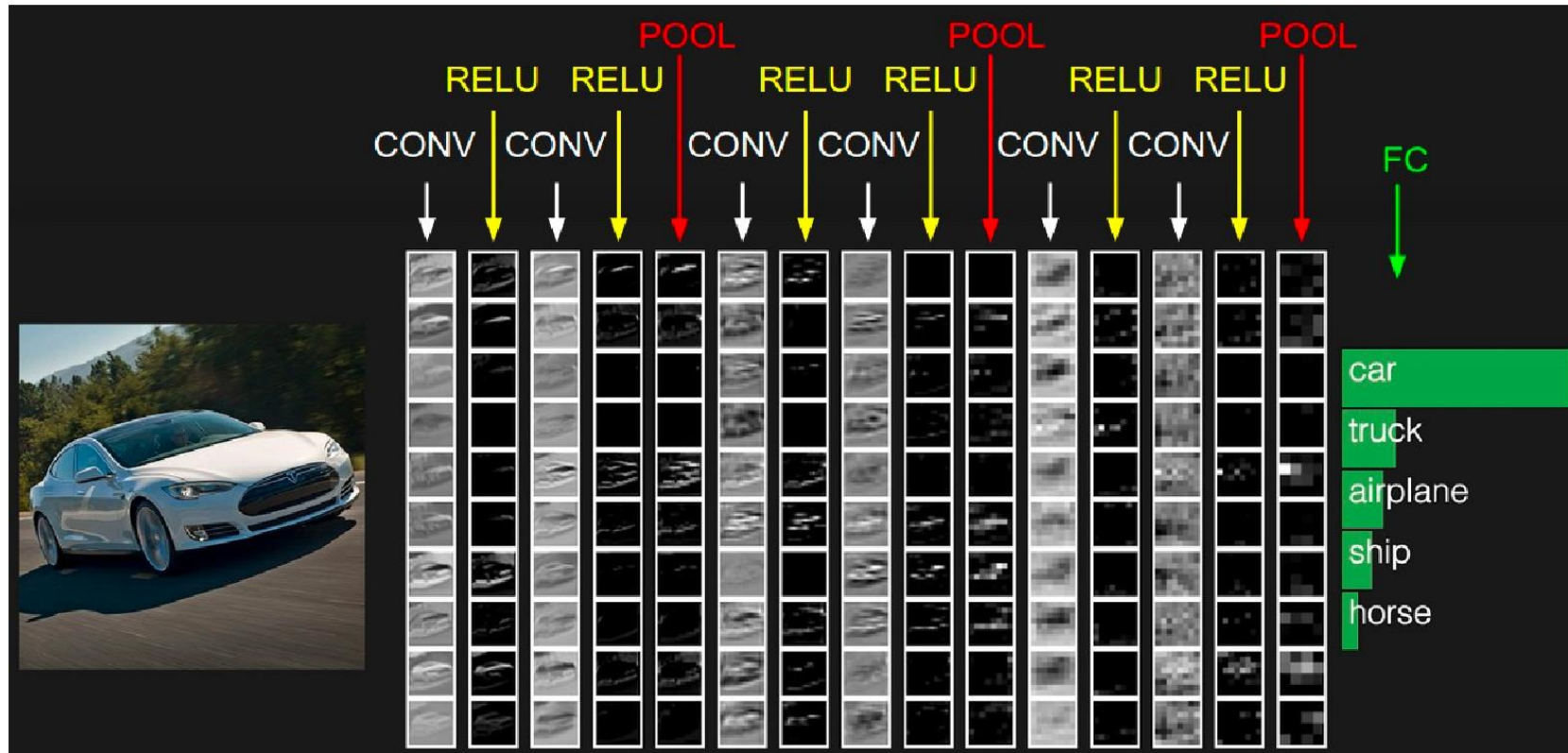
# Pooling layer

- The role: Summarize the input and scale down the spatial size.
  - has the effect of **routing** the region with the most activation.
- Recall depth slice: take the matrix at a particular depth.
- Max pooling: run a particular filter that computes maximum, for each depth slice.



- Variation: average pooling (but not popular).
- Recommended: Filter size  $F=2$ , stride length  $S=2$ . ( $F=3$ ,  $S=2$  is also commonly used – overlapping pooling).
- Note: There are **no parameters** for this layer!

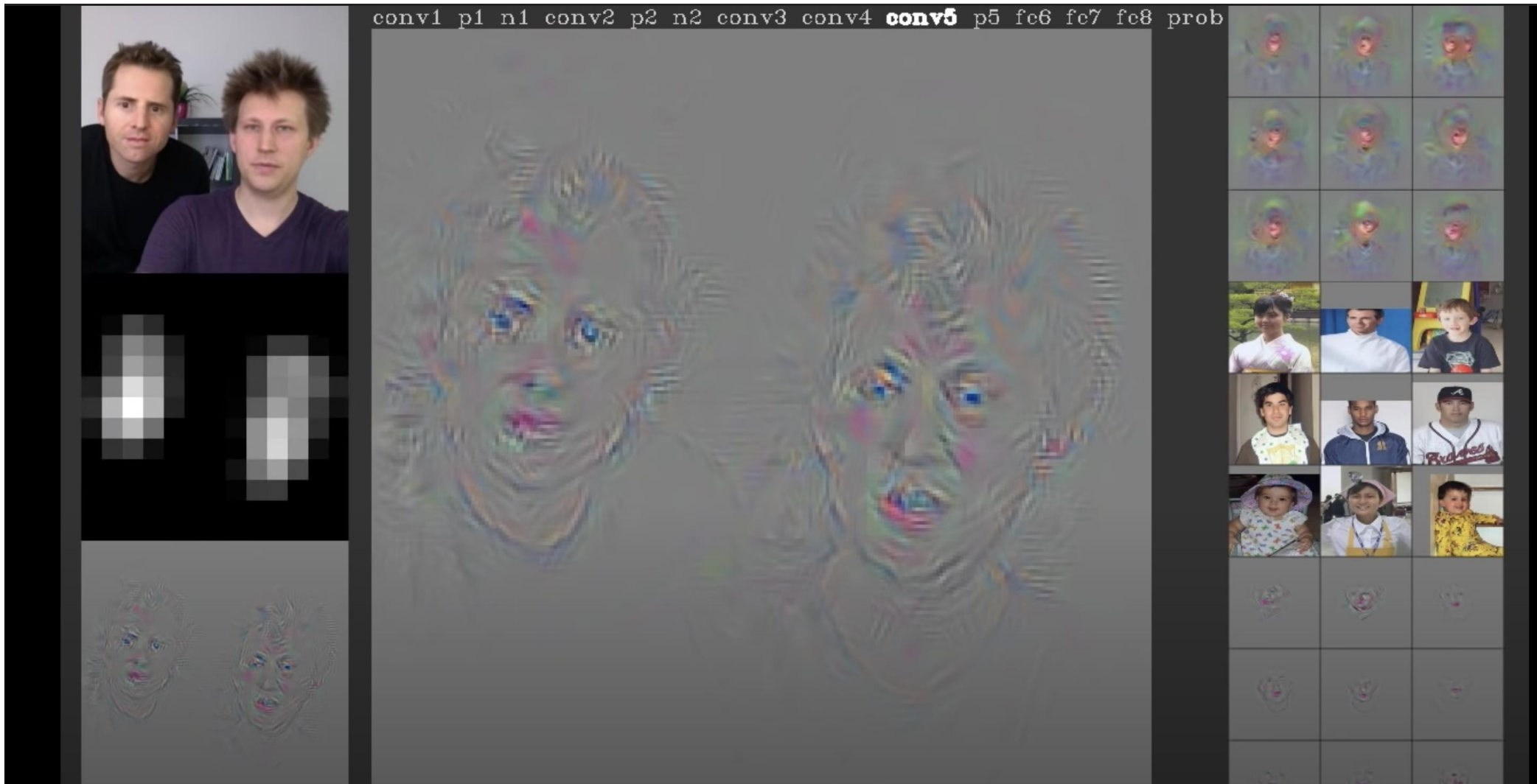
# Typical architectural patterns in CNN





# Seeing what happens in CNN

- <https://yosinski.com/deepvis#toolbox>



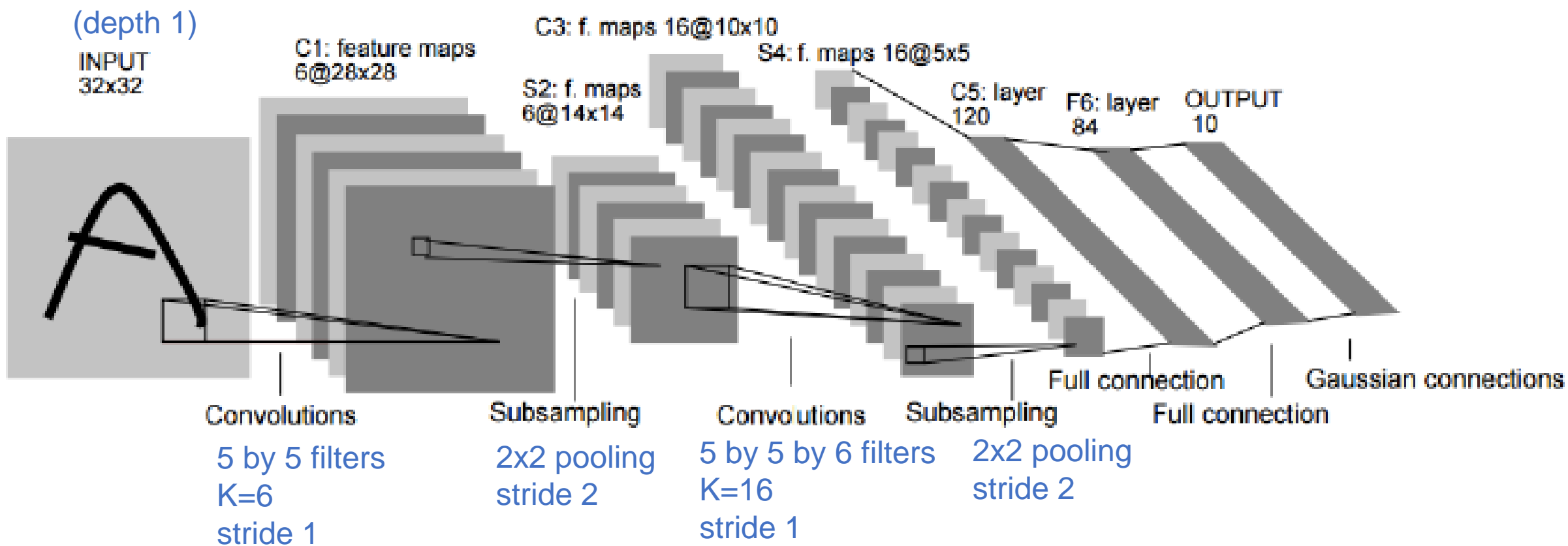


# CNN examples

# LeNet-5

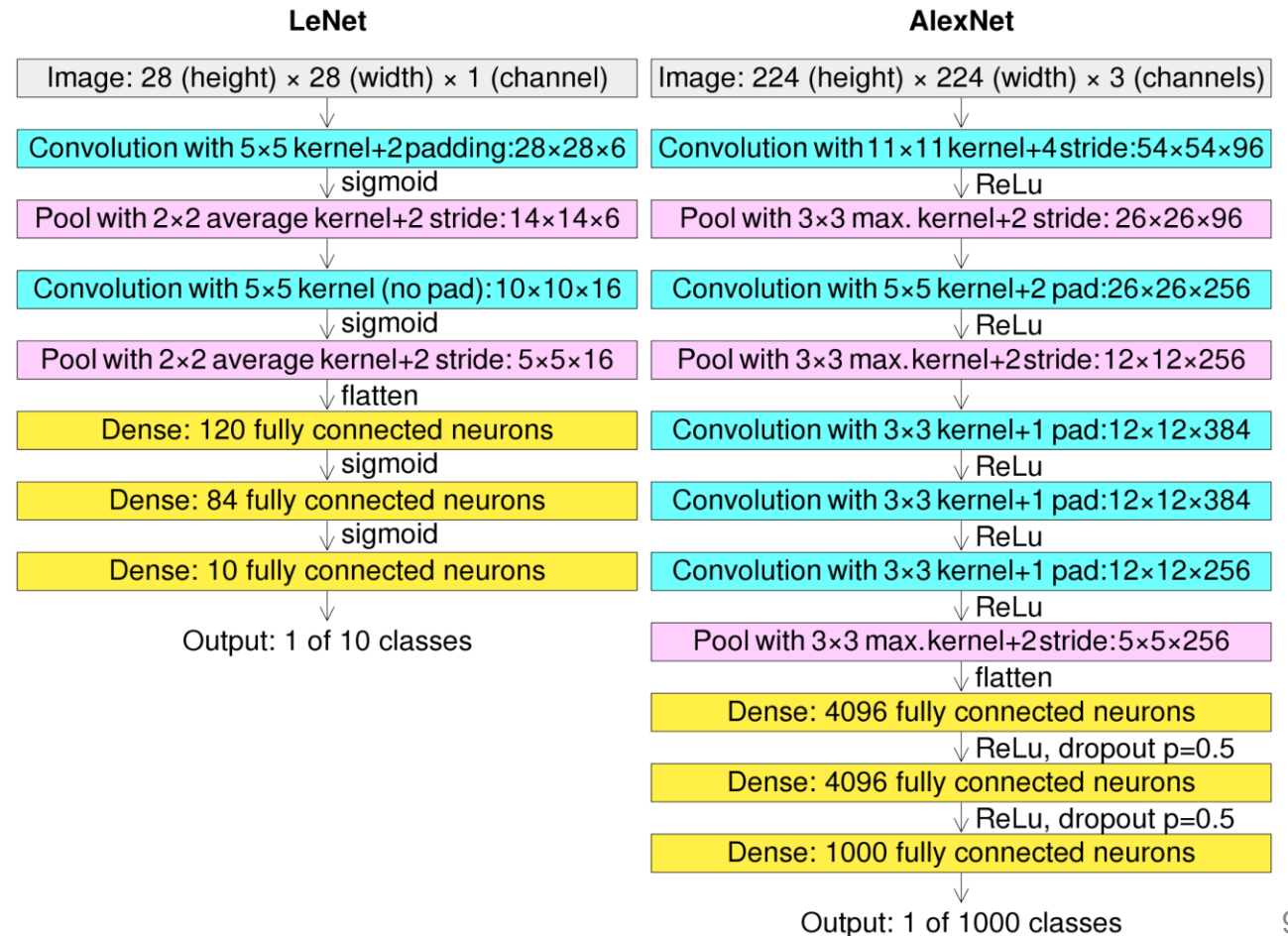
- Proposed in “Gradient-based learning applied to document recognition”, *by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in Proceedings of the IEEE, 1998*
- Apply convolution on 2D images (handwritten characters) and use backpropagation
- Structure: 2 convolutional layers (with pooling) + 3 fully connected layers
  - Input size: 32x32x1
  - Convolution kernel size: 5x5
  - Pooling: 2x2

# LeNet-5



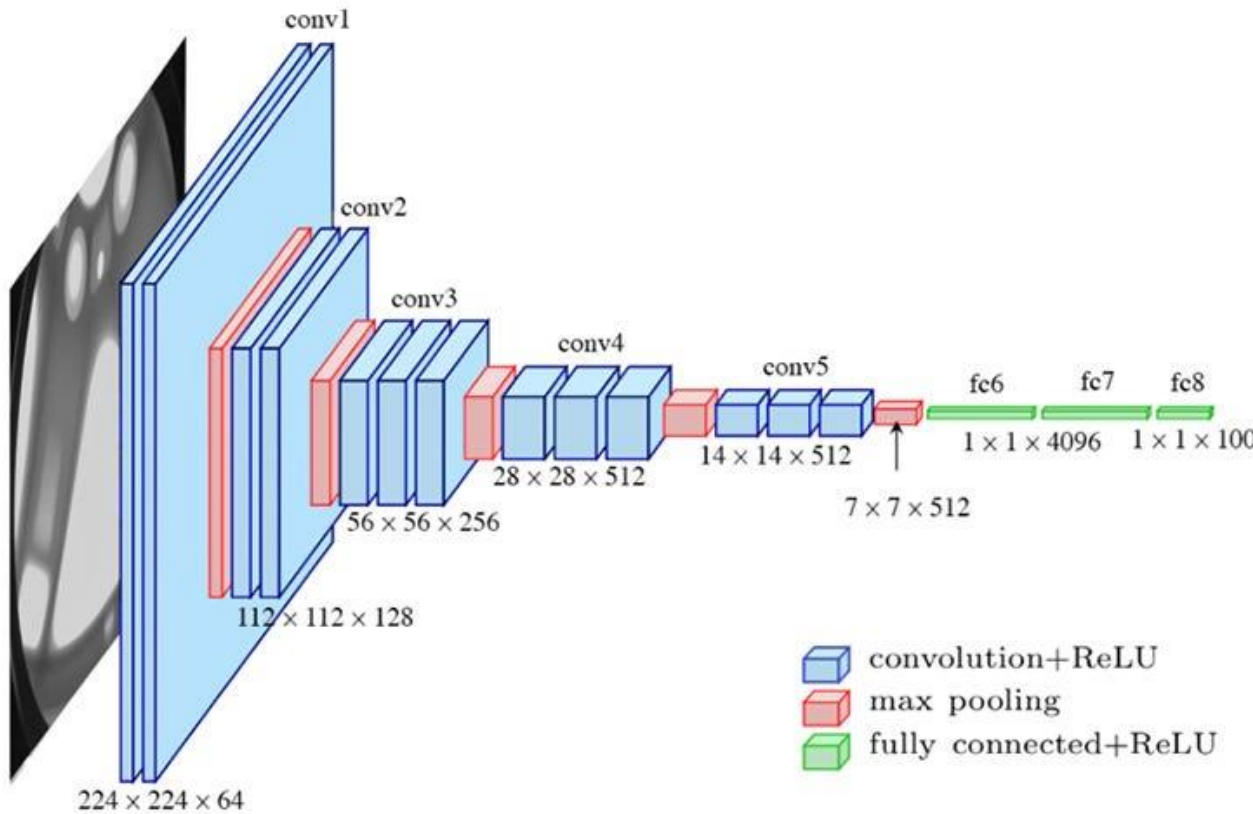
# AlexNet (2012)

- Won the ImageNet competition with top-5 test error rate of 16.4% (second place was 26.2%). (1000 classes)
- Almost just an extension of LeNet-5. But uses ReLU for the first time.



# VGGNet (2014): 7.3% error on ImageNet

- Mimic large convolutional filters with multiple small (3x3) convolutional filters
- Amortizing memory cost: every time it halves the spatial size, double the # of filters



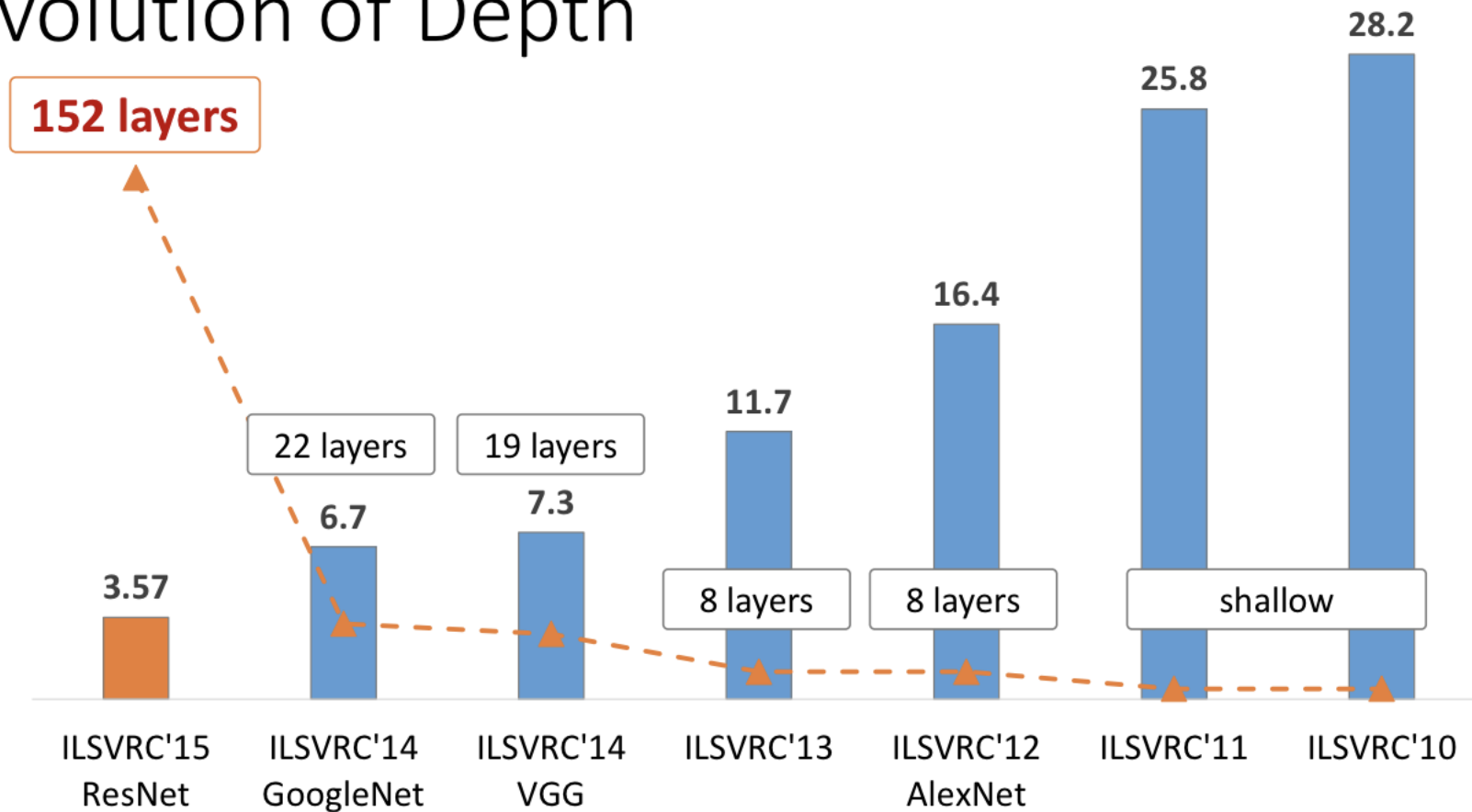
```
INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000
```

# ResNet (2016): 3.5% error on ImageNet

- Proposed in “Deep residual learning for image recognition” by *He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun*. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
- Apply very deep networks with repeated **residual blocks**.
- Structure: simply stacking residual blocks, but the network is very deep.



# Revolution of Depth



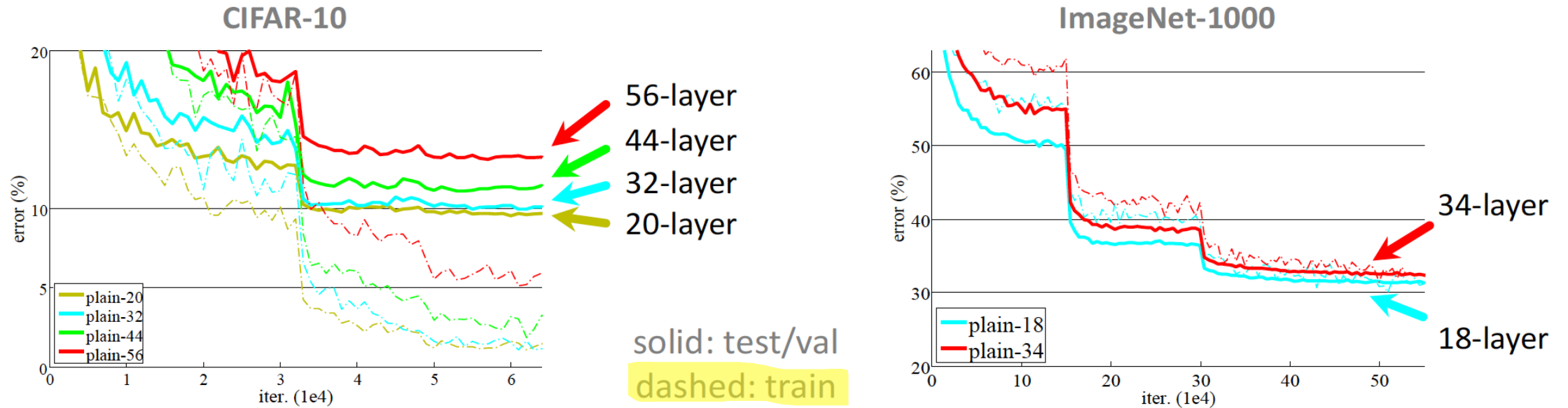
ImageNet Classification top-5 error (%)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.



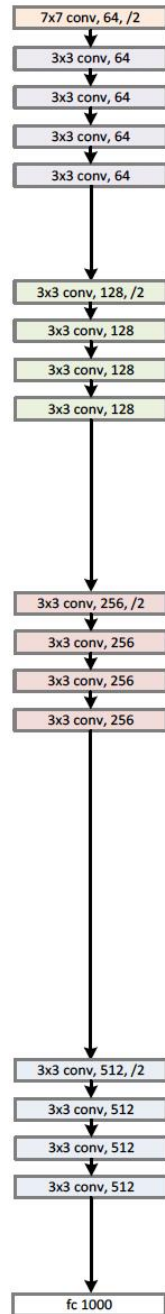


# Deep nets seem to suffer

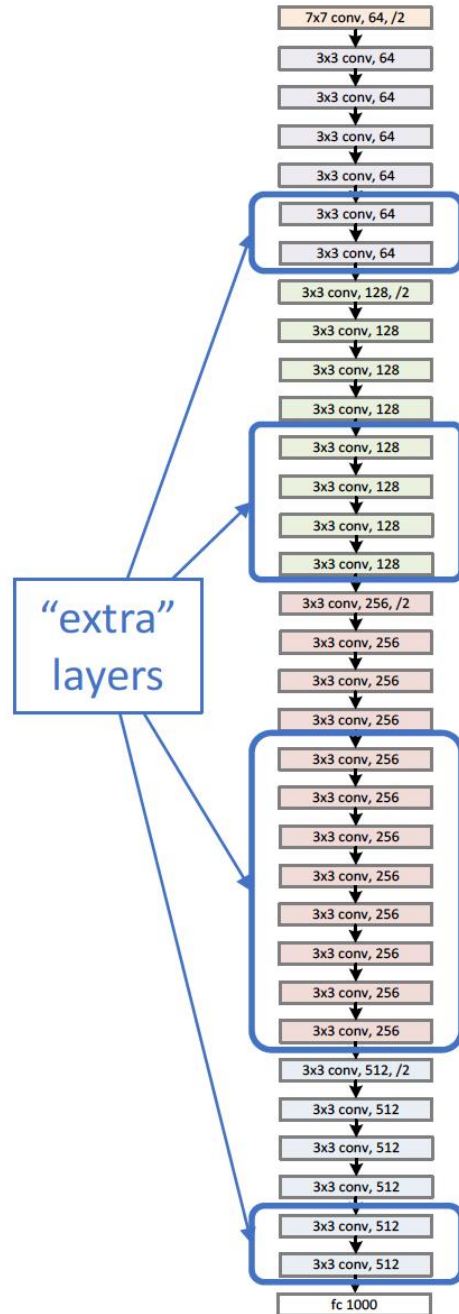


- “Overly deep” plain nets have **higher training error**
- A general phenomenon, observed in many datasets

a shallower model  
(18 layers)



a deeper counterpart  
(34 layers)



(slides from Kaiming He)

- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as **identity**
  - at least the same training error
- **Optimization difficulties**: solvers cannot find the solution when going deeper...

[http://image-net.org/challenges/talks/ilsvrc2015\\_deep\\_residual\\_learning\\_kaiminghe.pdf](http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf)

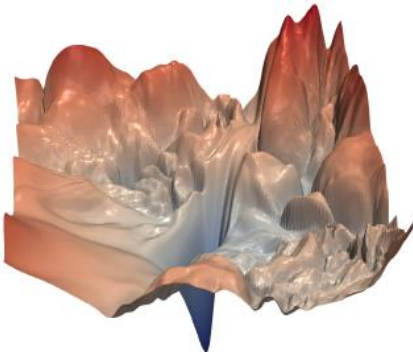
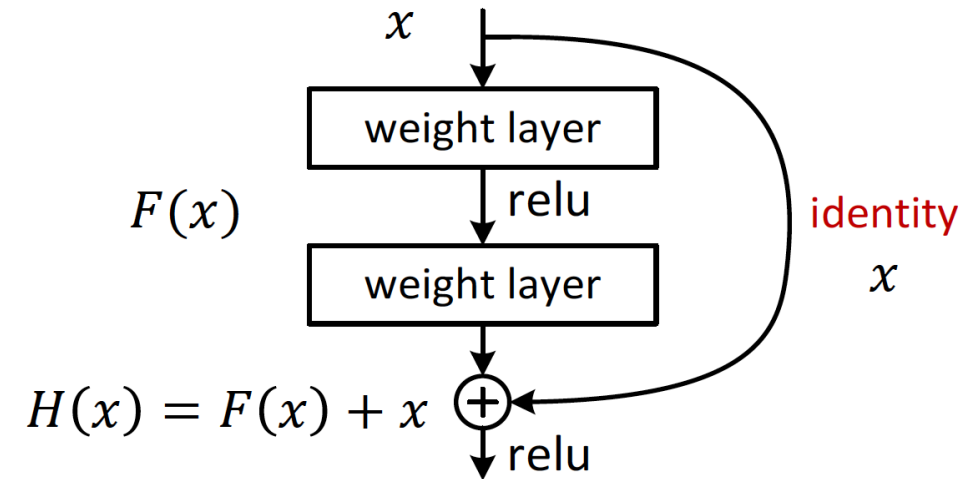
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

# Key idea: skip connections

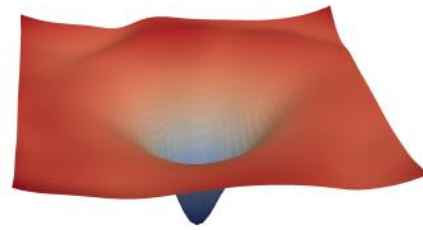
## Skip connections

- $F(x)$  encodes residual representations, which has previously been explored in early works
- When backprop'ing, by the chain rule, gradients will 'flow' directly to the previous layer.
  - In contrast, plain CNNs suffer from vanishing gradient problem
- It makes the optimization landscape much better

- Residual net



(a) without skip connections



(b) with skip connections

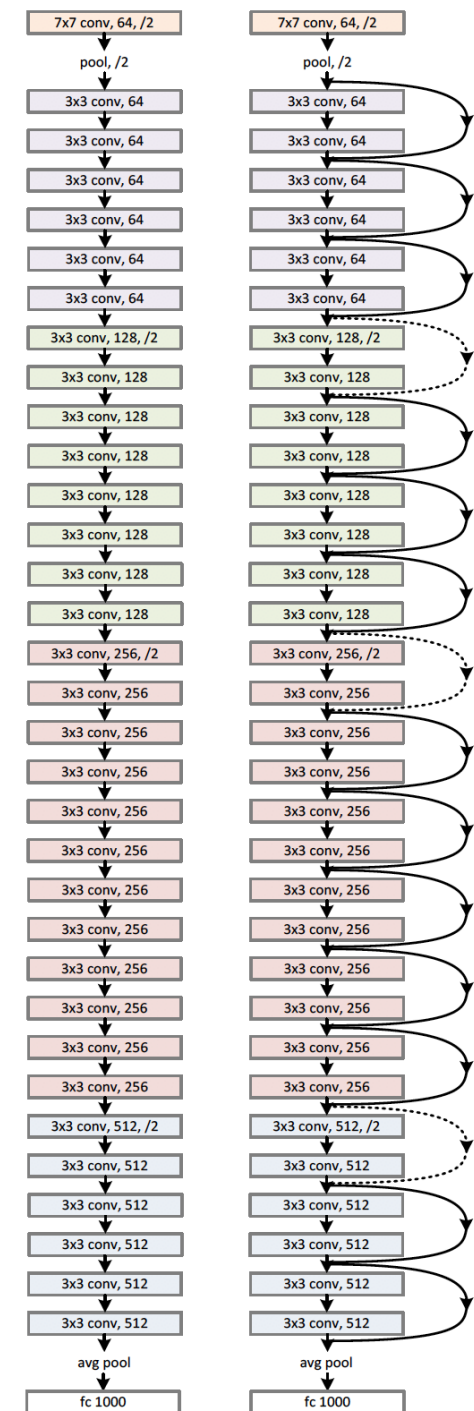
[http://image-net.org/challenges/talks/ilsvrc2015\\_deep\\_residual\\_learning\\_kaiminghe.pdf](http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf)

<https://www.cs.umd.edu/~tomg/projects/landscapes/>

# ResNet

- VGG-style scheme: halve the spatial size, double the # of filters
- Use conv layer with stride 2 occasionally to reduce the spatial dimension => called **bottleneck** blocks.

plain net



# ResNet in PyTorch

Torchvision implementation:

[https://pytorch.org/vision/0.8/\\_modules/torchvision/models/resnet.html](https://pytorch.org/vision/0.8/_modules/torchvision/models/resnet.html)

```
class Bottleneck(nn.Module):
    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

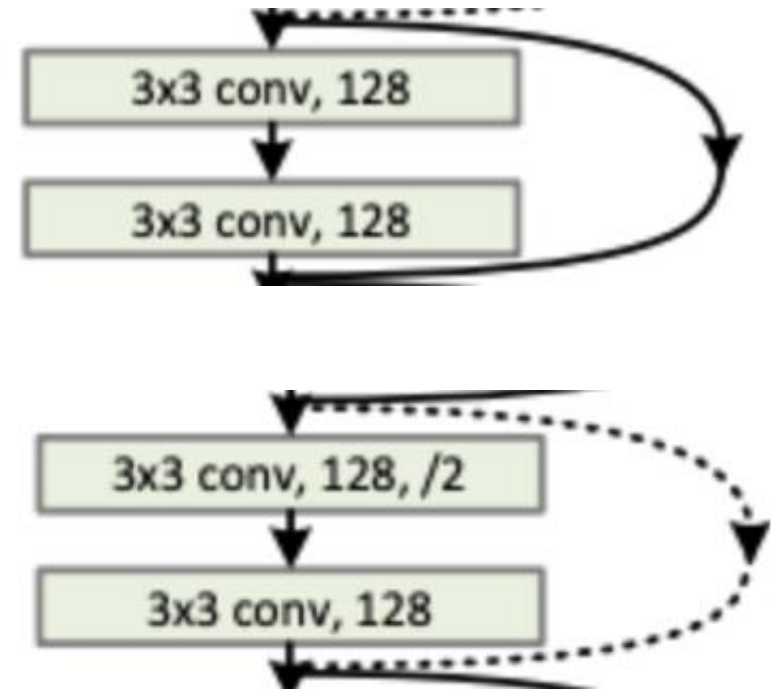
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

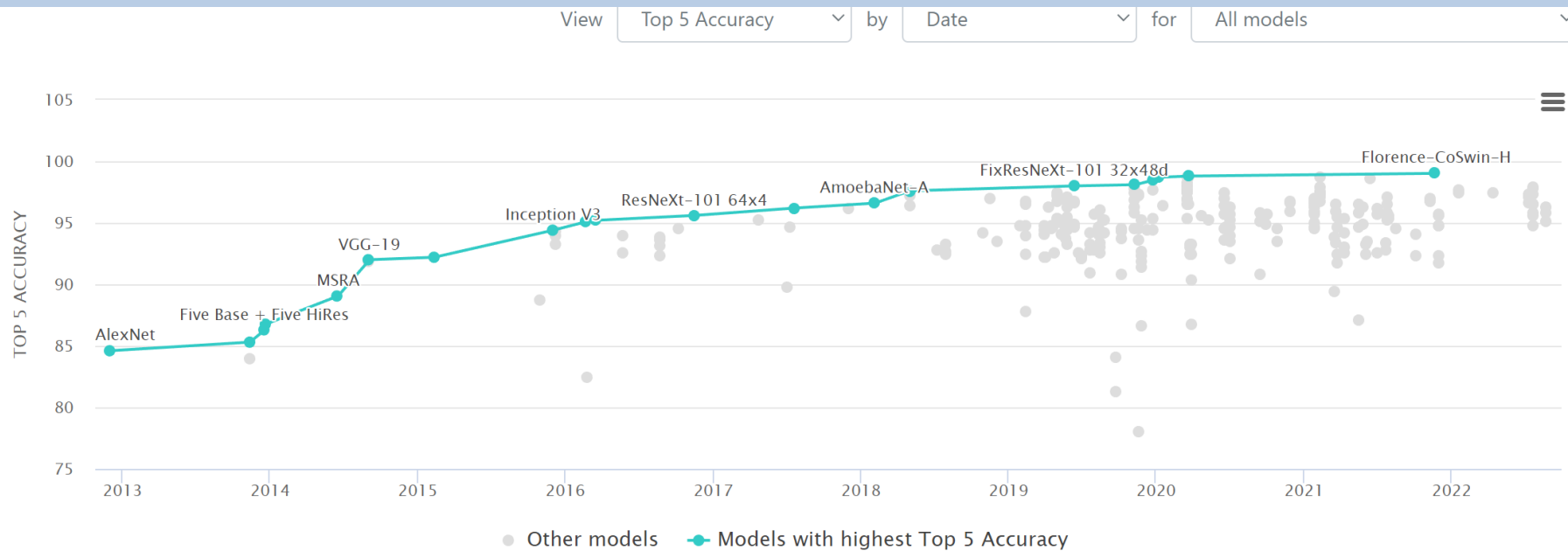
        out += identity
        out = self.relu(out)

        return out
```

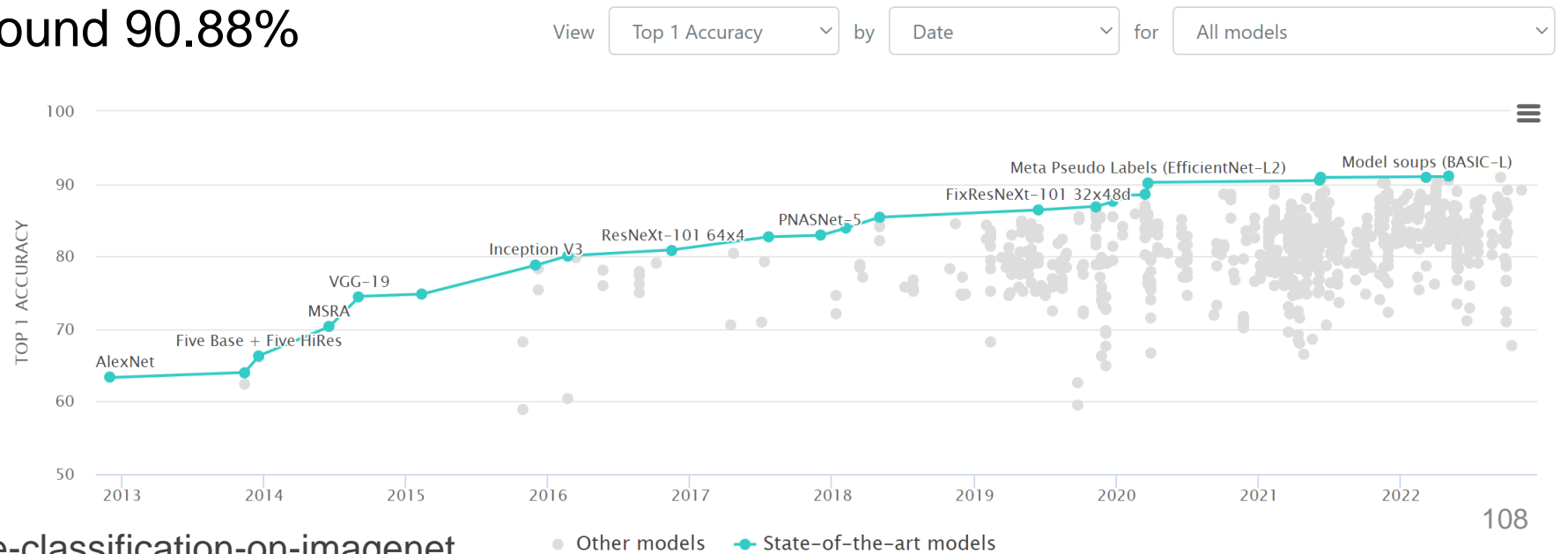


# ImageNet nowadays

Top-5 accuracy is boring



SoTA top-1 accuracy is around 90.88%

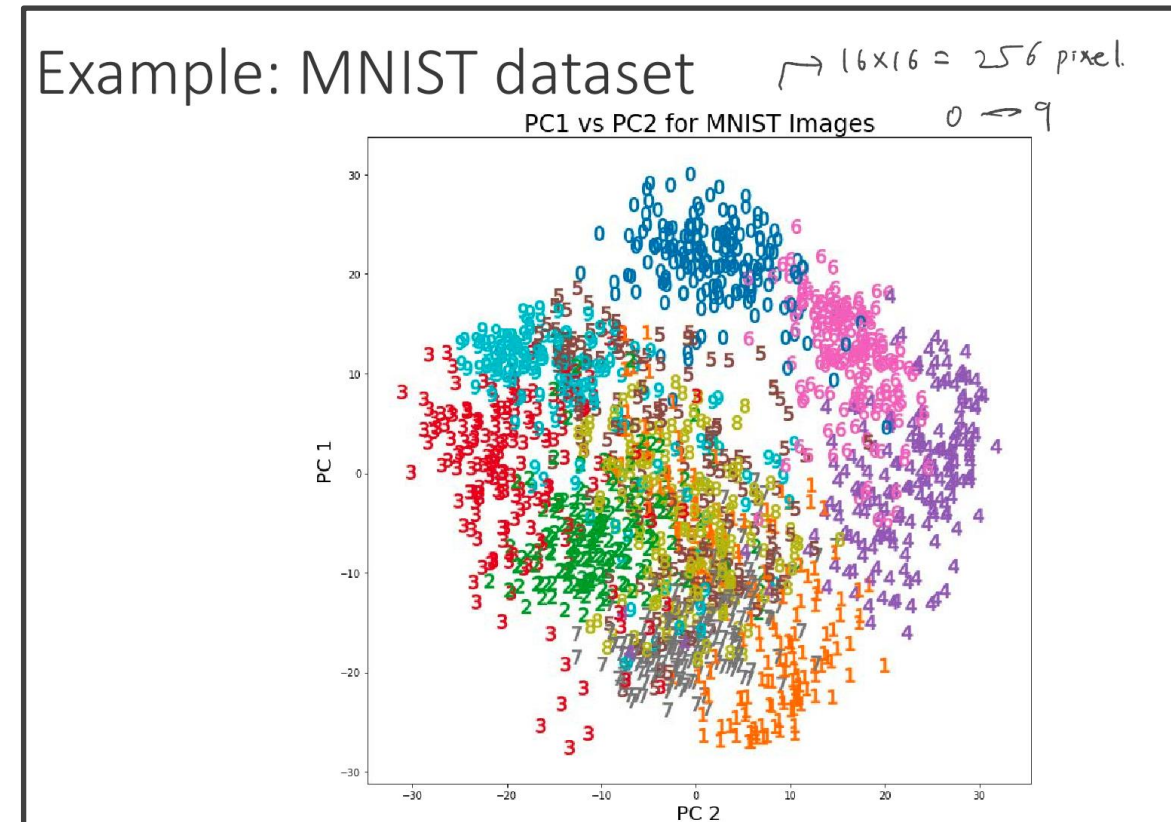


# Autoencoder



# Unsupervised Learning Review

- Recall: unlabeled data.
  - Q: what is the main goal of unsupervised learning?
  - Examples: clustering, PCA.
- 
- Recall PCA can be used for 'representation learning' = learning useful (and compact) features.  
(learned features = projected feature vector)
  - NNs can be used to do generalizations of PCA.



# Introductory Example

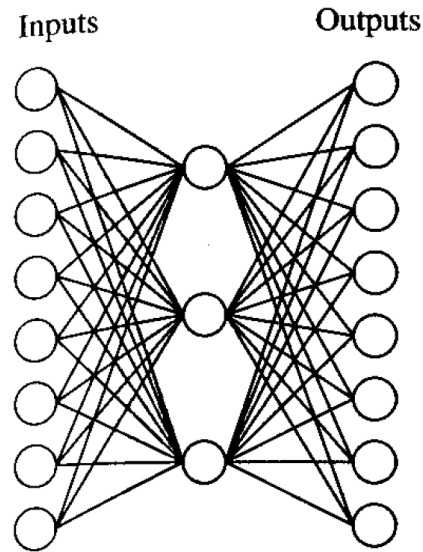
- Suppose you have a number in  $\{0,1,2,3,4,5,6,7\}$
- What would be a compact representation (say, for computers)?
  
- Q: how many bits do we need?

# Early Observations

Train a pair of (encoder E, decoder D) such that

- $D(E(x))$  recovers  $x$
- imposing squared loss on all the output units & backpropagation.

Q: What do the hidden values (codes)  $E(x)$  look like?

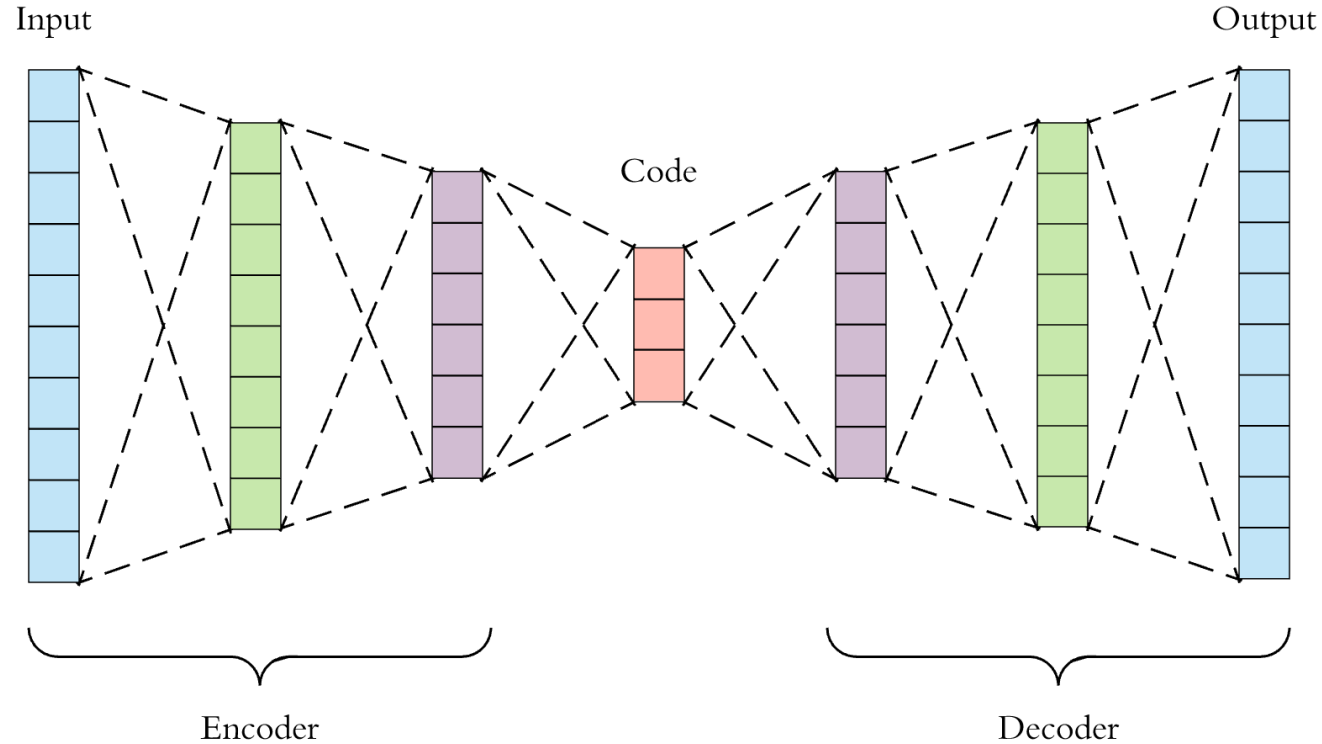


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

**FIGURE 4.7**

Learned Hidden Layer Representation. This  $8 \times 3 \times 8$  network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

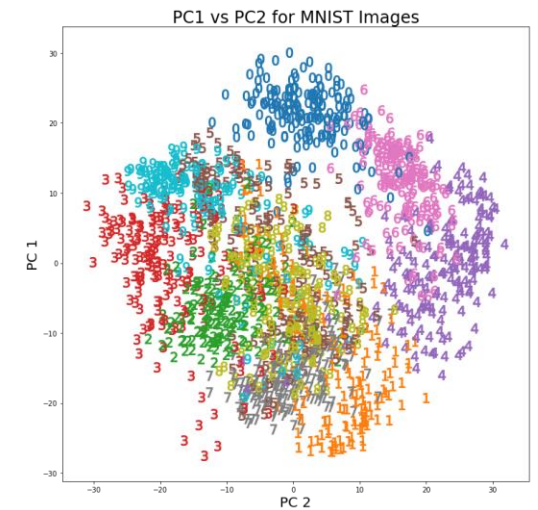
# Autoencoder using deep networks



We can do this for any data!

How to use it:

- Encoder: for dimensionality reduction
- Decoder: generate new samples from the distribution by varying the input 'code'



# PCA as a linear autoencoder

linear = no activation

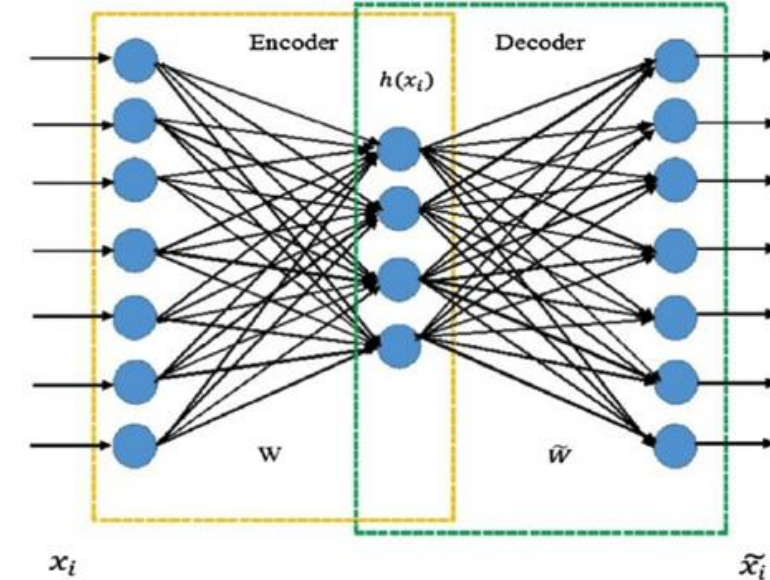
## PCA pseudocode

- Input: data matrix  $X \in \mathbb{R}^{n \times d}$
- Preprocess: Let  $\mu = \frac{1}{n} \sum_{i=1}^n x_i$ . Compute  $x'_i = x_i - \mu, \forall i \in [n]$
- Compute the top  $k$  eigenvectors  $V = [v_1, \dots, v_k]$  of  $\frac{1}{n} \sum_{i=1}^n x'_i (x'_i)^\top$
- Feature map:  $\phi(x) = (v_1^\top (x - \mu), \dots, v_k^\top (x - \mu)) \in \mathbb{R}^k$
- Decorrelating property: ("whitening").
  - $\frac{1}{n} \sum_{i=1}^n \phi(x_i) = 0$
  - $\frac{1}{n} \sum_{i=1}^n \phi(x_i) \phi(x_i)^\top = \text{Diag}(\lambda_1, \dots, \lambda_k)$
- Reconstruction (the actual projection): apply  $\mu + V\phi(x)$

← coefficient.

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_k \end{bmatrix}$$

↑  
( $v_1 \dots v_k$ )

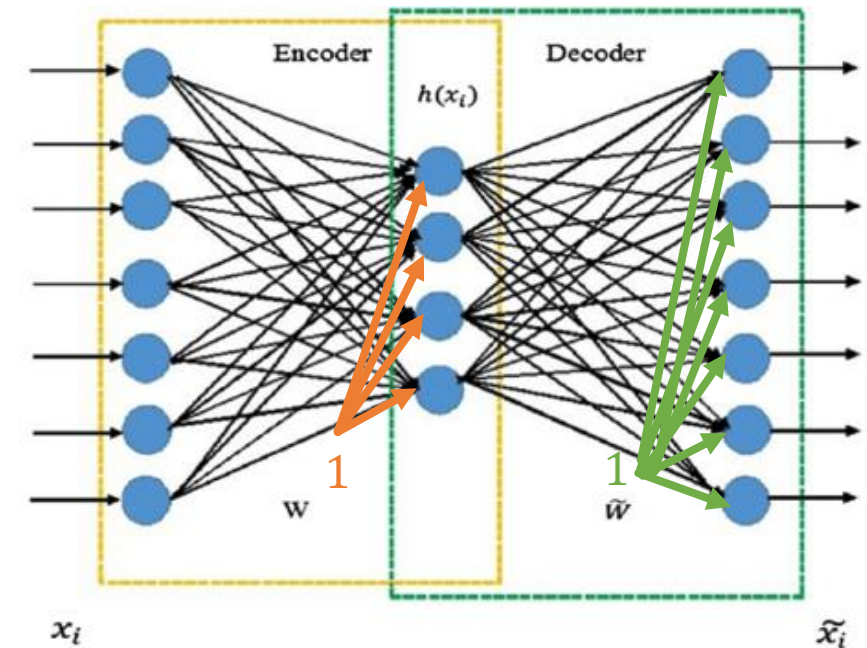


# PCA as a linear autoencoder

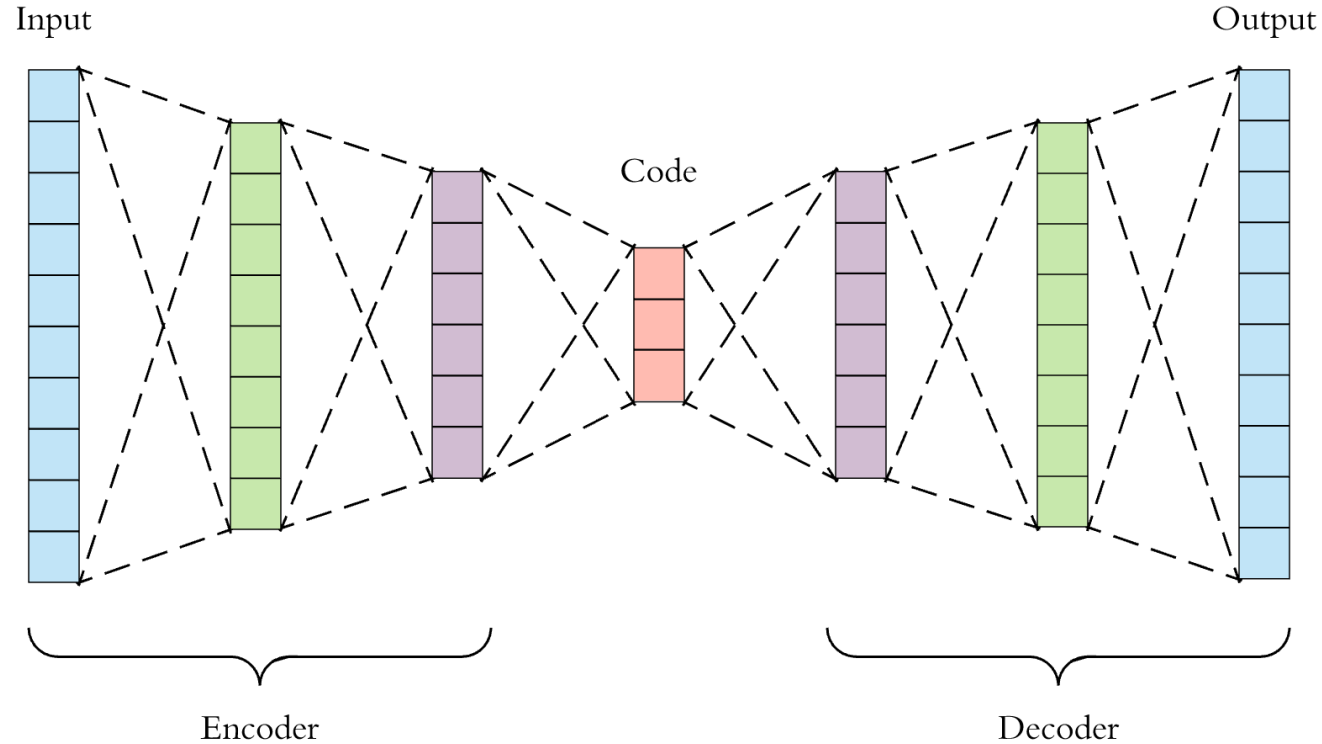
- The PCA can be represented as an autoencoder with  $k$  units in the hidden layer, constant bias added in each layer):

- Encoder: 
$$h = \begin{pmatrix} -v_1 & - \\ \dots & \\ -v_k & - \end{pmatrix} \cdot x + \begin{pmatrix} -v_1^\top \mu \\ \dots \\ -v_k^\top \mu \end{pmatrix}$$

- Decoder: 
$$\tilde{x} = \begin{pmatrix} | & & | \\ v_1 & \dots & v_k \\ | & & | \end{pmatrix} \cdot h + \begin{pmatrix} | \\ \mu \\ | \end{pmatrix}$$



# Autoencoder using deep networks



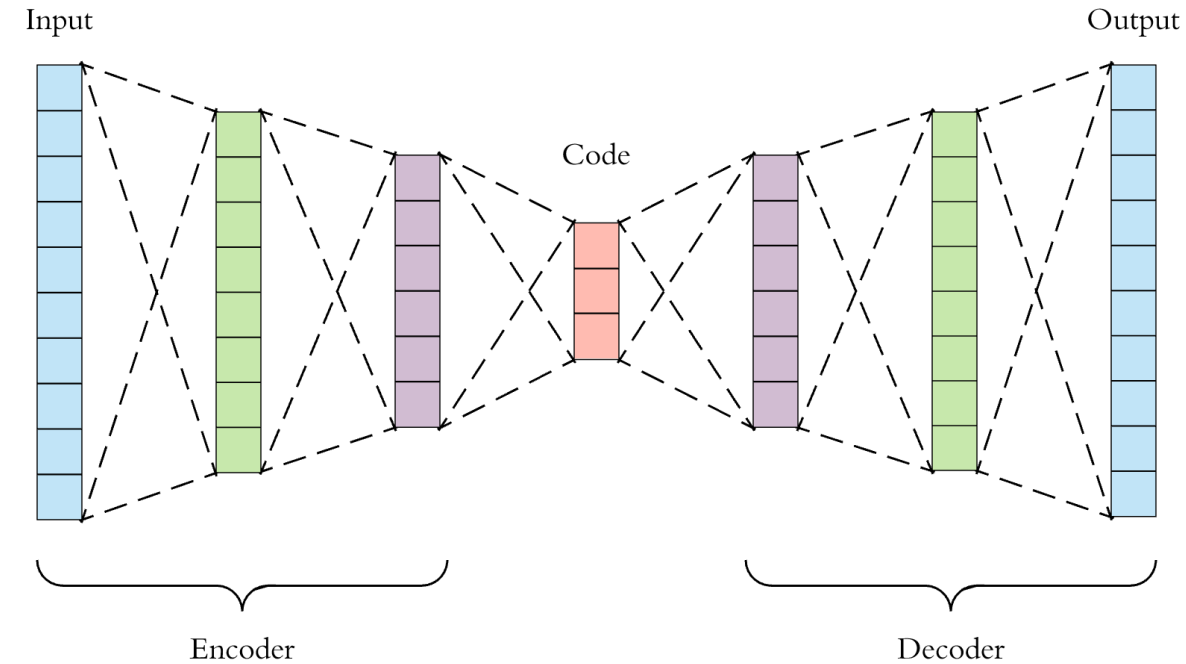
We can do this for any data!

What about images?



# Training autoencoders

- **Given:**
  - data  $x_1, \dots, x_n \in \mathbb{R}^d$ ,
  - Embedding dimension  $k$  ( $k \ll d$ )
- **Goal: obtain**
  - Encoder network  $f_\theta: \mathbb{R}^d \rightarrow \mathbb{R}^k$
  - Decoder network  $g_\phi: \mathbb{R}^k \rightarrow \mathbb{R}^d$
  - Such that for every  $i$ ,  $x_i \approx g_\phi(f_\theta(x_i))$



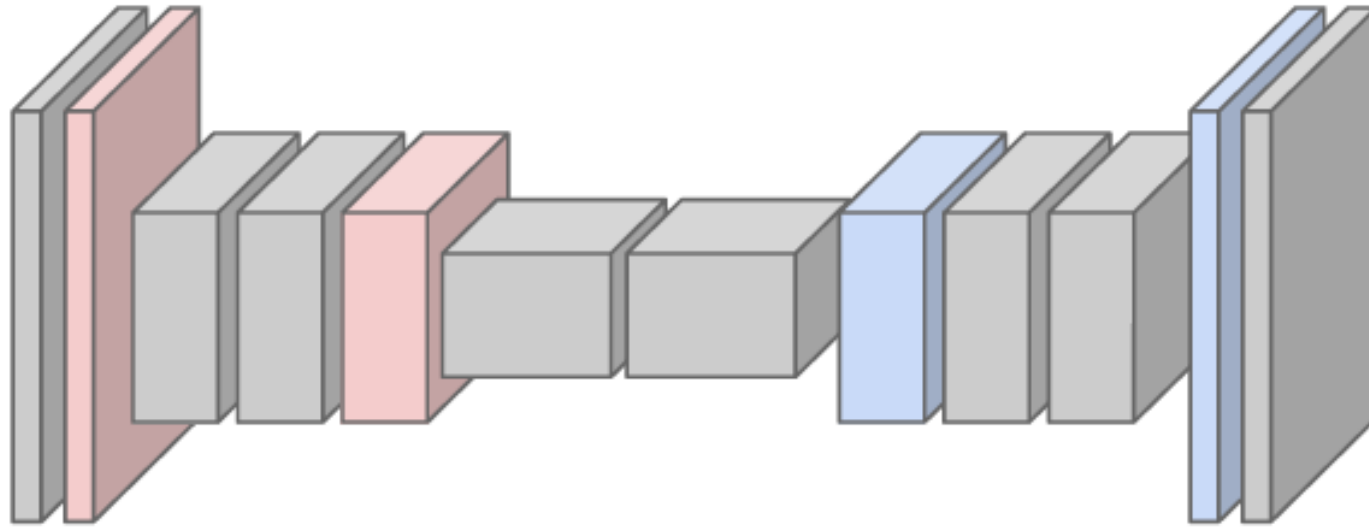
- Most commonly used formulation (can be straightforwardly trained by gradient-based methods):

$$\text{minimize}_{\theta, \phi} \sum_{i=1}^n \underbrace{\|x_i - g_\phi(f_\theta(x_i))\|^2}_{\text{Reconstruction error}}$$

Reconstruction error

# Autoencoder for images

- Encoder: conv-conv-pool-conv-conv-pool-....,
- Decoder: conv-conv-pool-...?? It will reduce the spatial dimension rather than increasing it.
- How to do the opposite of pooling (or conv with stride length  $\geq 2$ )?



Following slides largely based on Stanford cs231n <https://youtu.be/nDPWyywWRIRo?t=1109>

[http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture11.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf) 118

# “Un”pooling

**Nearest Neighbor**

1	2
3	4



1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Input: 2 x 2

Output: 4 x 4

**“Bed of Nails”**

1	2
3	4



1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

Input: 2 x 2

Output: 4 x 4

# Max unpooling

## Max Pooling

Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

Input: 4 x 4



5	6
7	8

Output: 2 x 2



Rest of the network

## Max Unpooling

Use positions from pooling layer

1	2
3	4

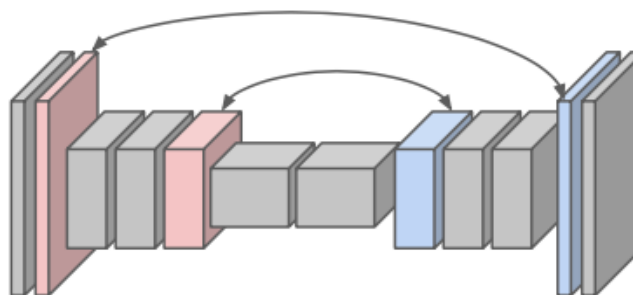
Input: 2 x 2



0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

Output: 4 x 4

Corresponding pairs of downsampling and upsampling layers

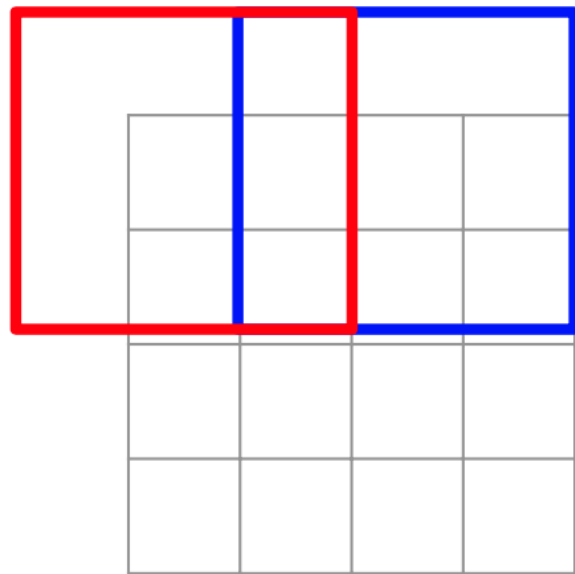


The dimensions in all layers of the network must be symmetric!

(fig from Stanford cs231n)<sup>120</sup>

# Transposed convolution

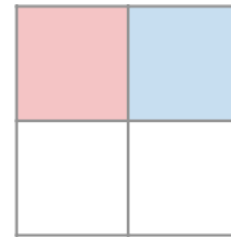
- Other names: upconvolution, fractionally strided convolution, backward strided convolution, deconvolution (don't use this name)
- Recall: 3 x 3 convolution with stride 2 pad 1.



Input: 4 x 4



Dot product  
between filter  
and input



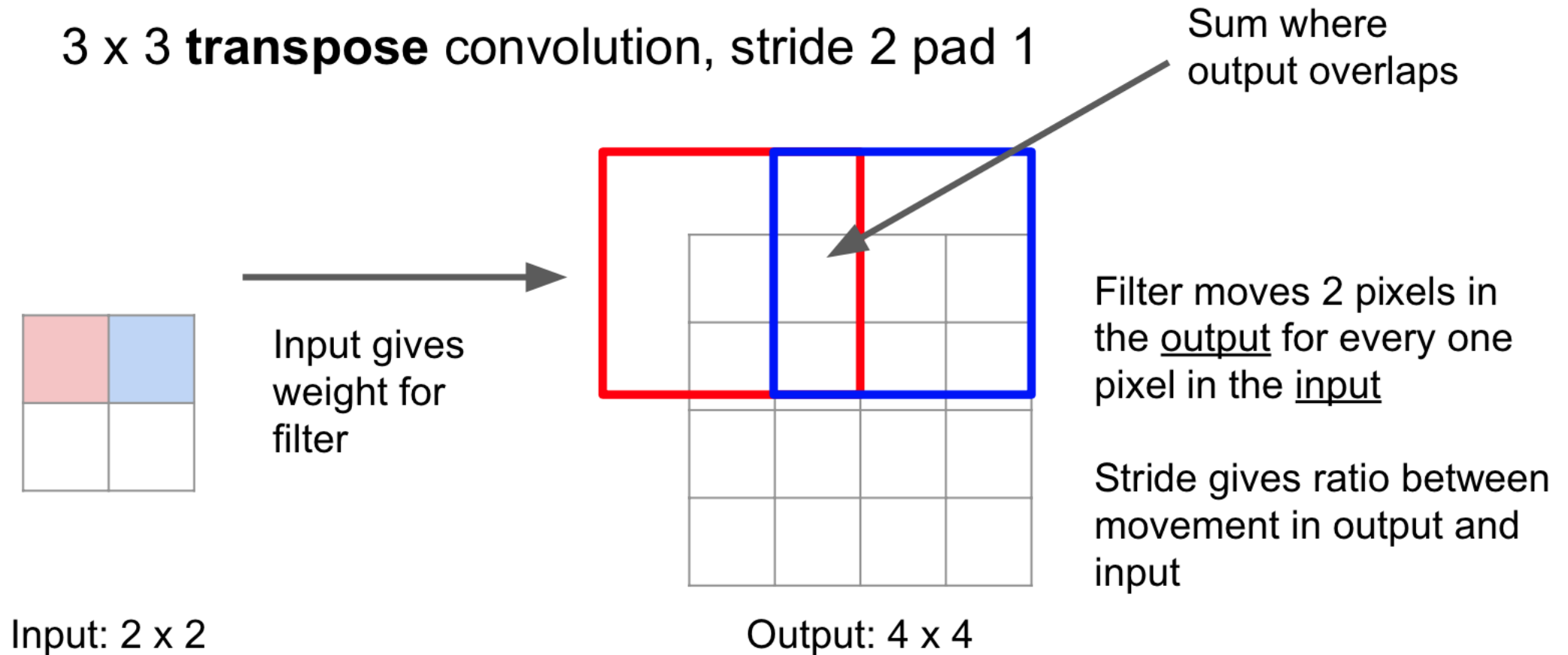
Output: 2 x 2

Filter moves 2 pixels in  
the input for every one  
pixel in the output

Stride gives ratio between  
movement in input and  
output

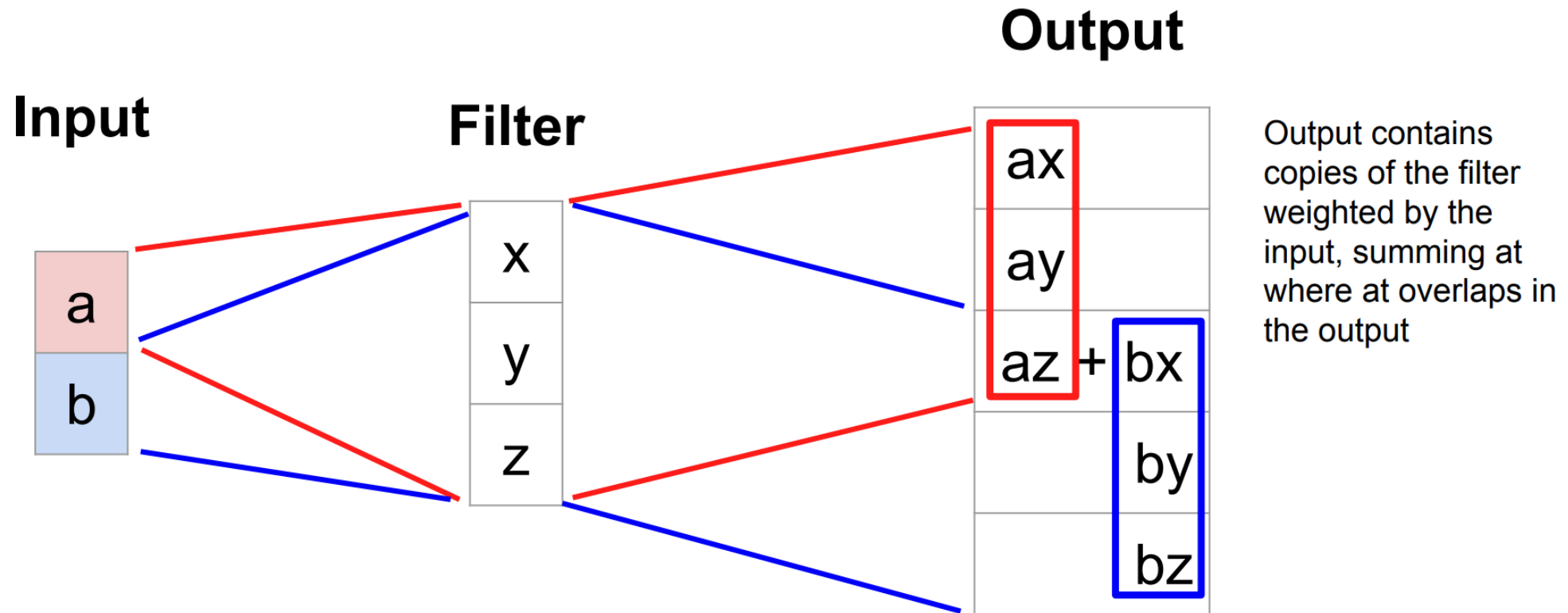
(fig from Stanford cs231n)

# Transposed convolution



**Disclaimer:** this is not the inverse of convolution!  
Rather, it's just a variation of the convolution.

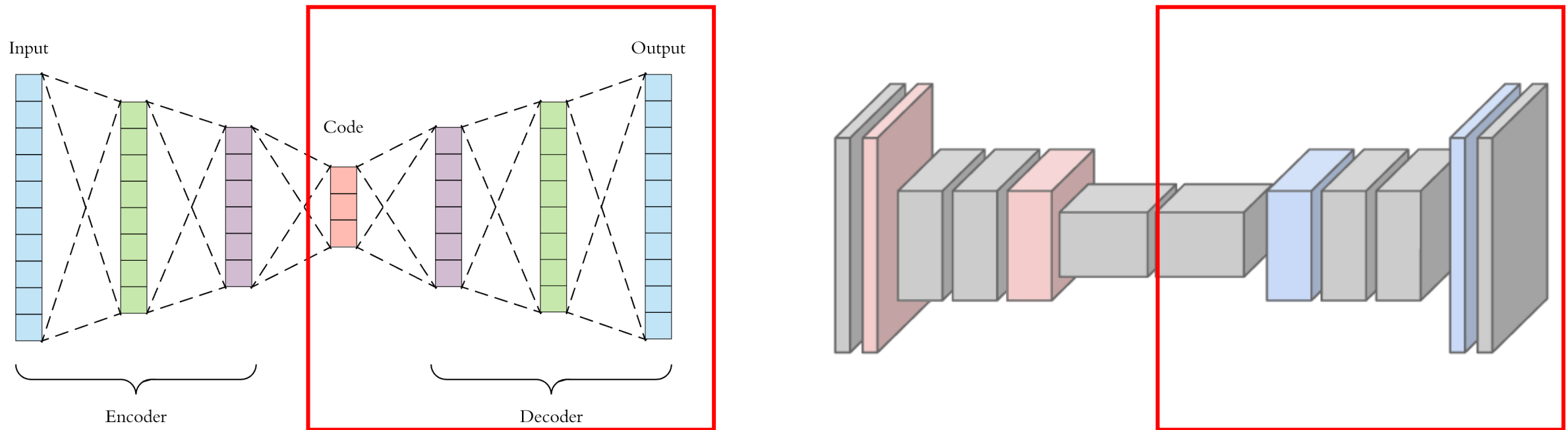
# 1D transposed convolution





# Decoders: additional remarks

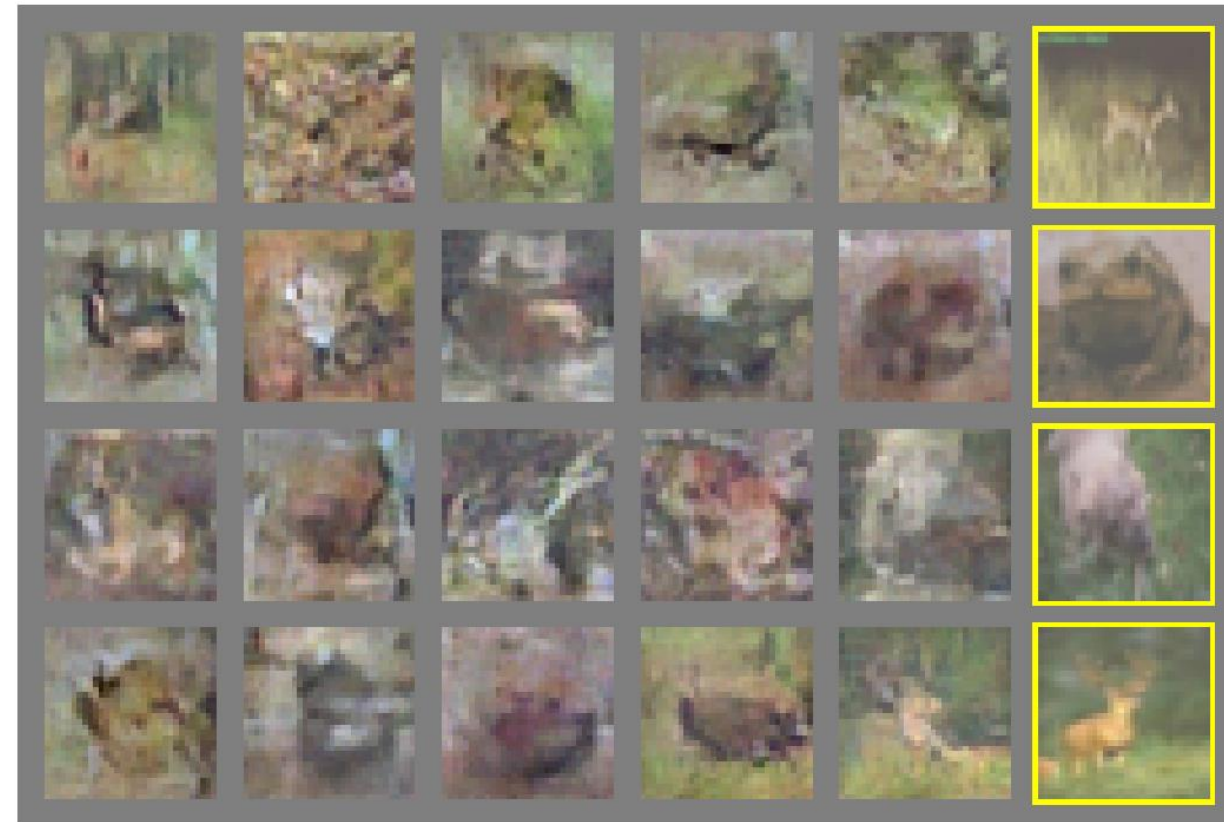
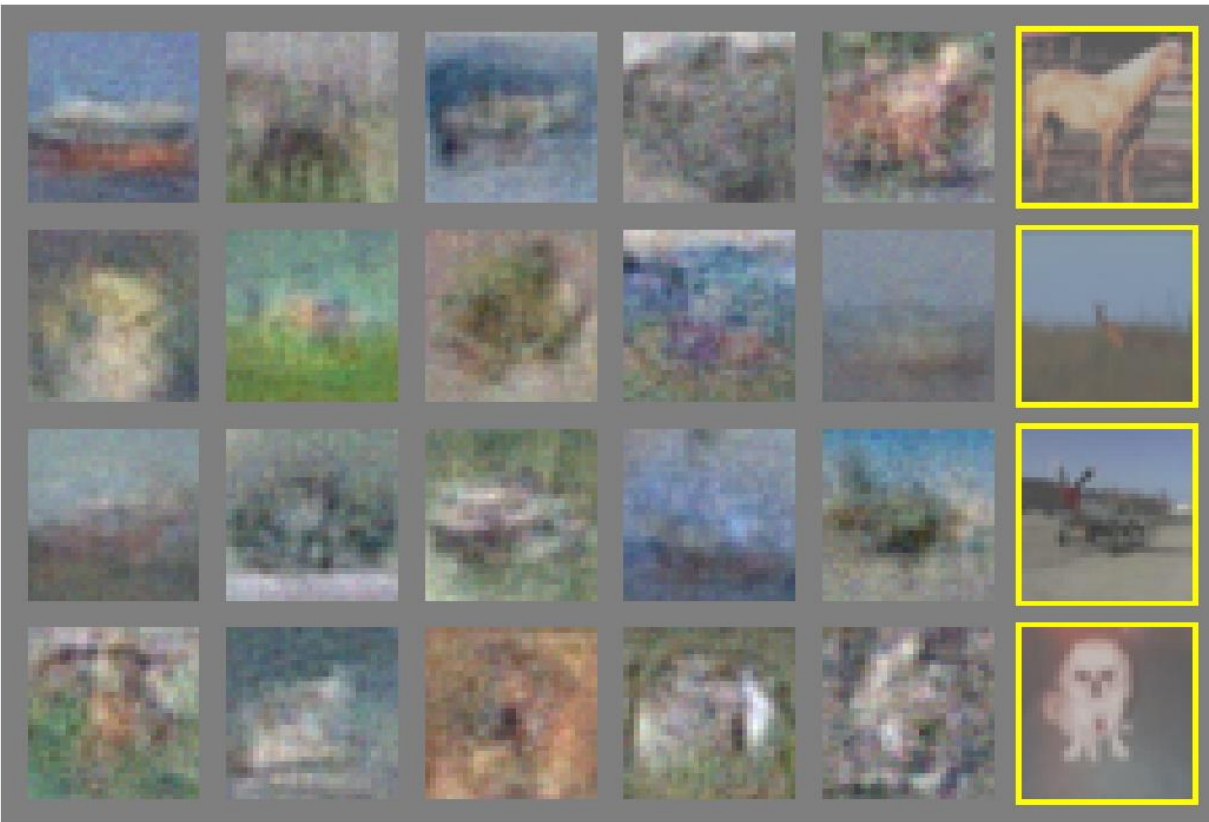
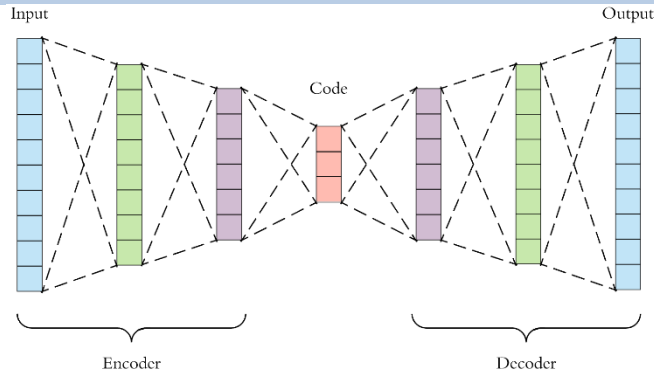
- Decoders: can be directly used to generate data from the original data distribution
- There are many “modern” ways to train decoders, e.g. generative adversarial network (GAN)



Following slides largely based on Stanford cs231n <https://youtu.be/nDPWyywWRIRo?t=1109>

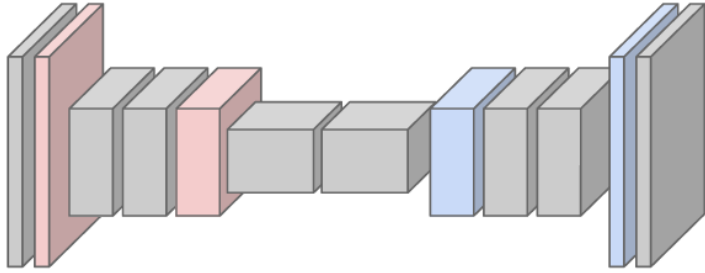
# Samples from fully-connected decoders

Ian Goodfellow et al., "Generative Adversarial Nets", NeurIPS 2014



(not using conv layers)

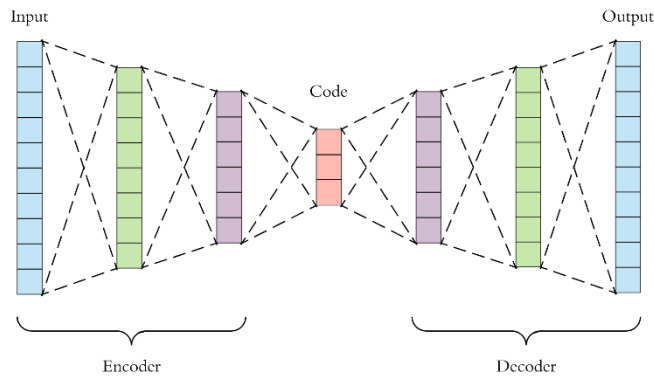
# Samples from convolutional decoders





# Autoencoders: more usages

- Interpolate between two samples by interpolating their codes



$D(z_1)$

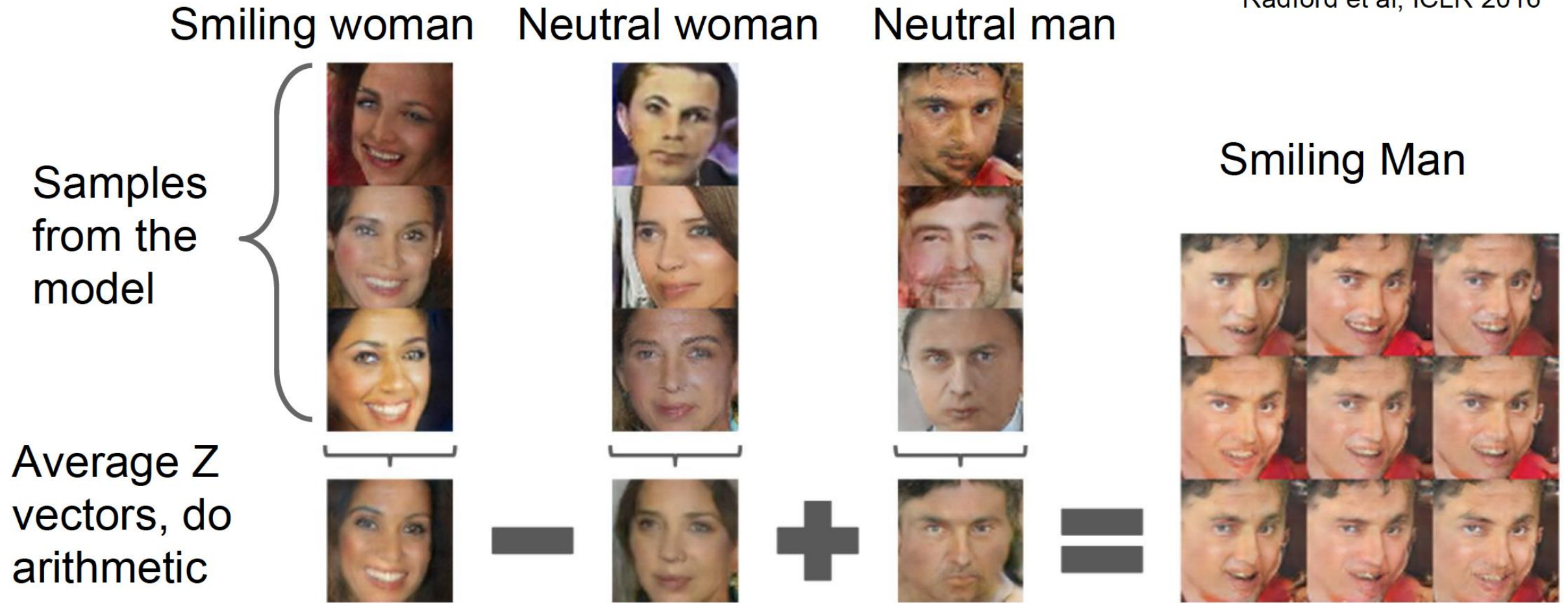
$D(tz_1 + (1 - t)z_2)$

$D(z_2)$



# Learned 'code' is interpretable

Radford et al, ICLR 2016



- This means that there are 'directions' in the latent code  $z$  that have particular meanings!

(slide from Stanford CS231n)

# Learned 'codes' is interpretable

Glasses man



No glasses man



No glasses woman



-

+

=

Woman with glasses



Radford et al,  
ICLR 2016



# Resources

“The Deep Learning Book” by Goodfellow et al.

<https://www.deeplearningbook.org/>

3Blue1Brown Youtube channel has a nice four-part intro:

<https://www.youtube.com/watch?v=aircAruvnKk>

Free book by Michael Nielson uses MNIST example in Python:

<http://neuralnetworksanddeeplearning.com/>

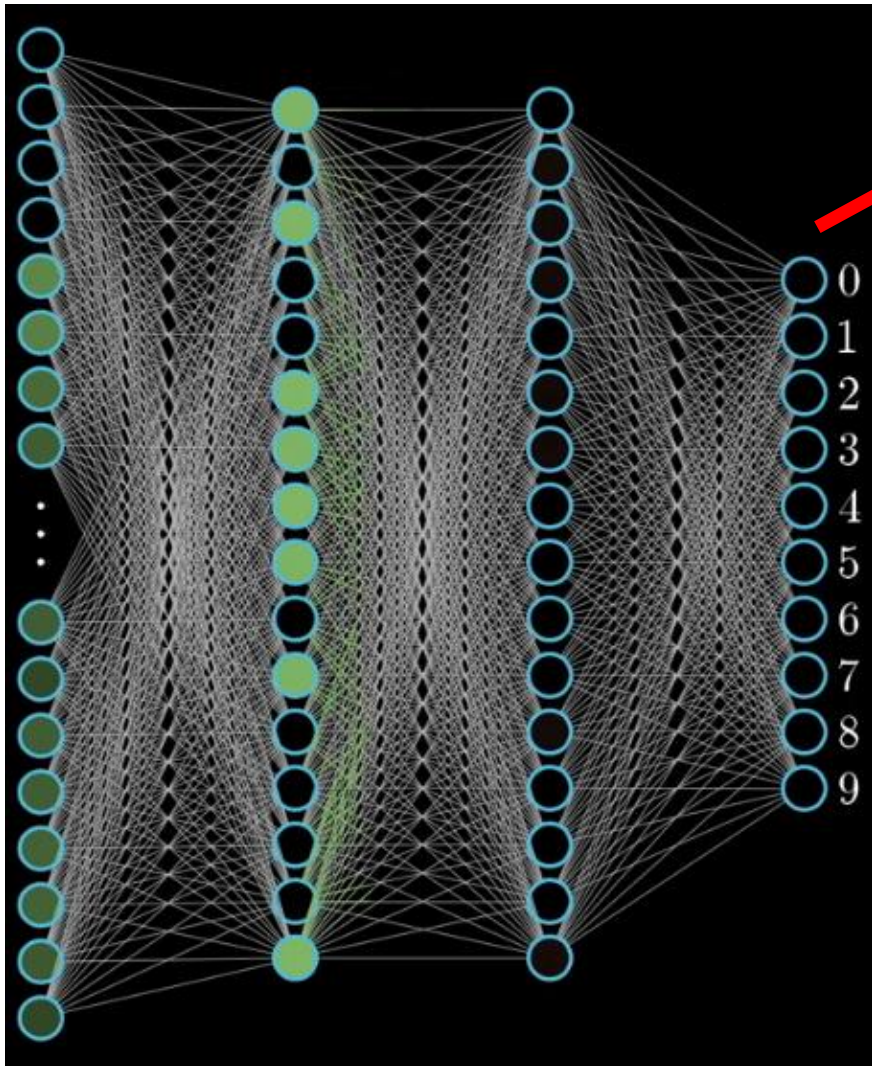
“Dive into deep learning”: an online textbook with interactive notebook

<https://d2l.ai/index.html>



Backup

# Multilayer Perceptron



Final layer is typically a linear model... each output node is computed by

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

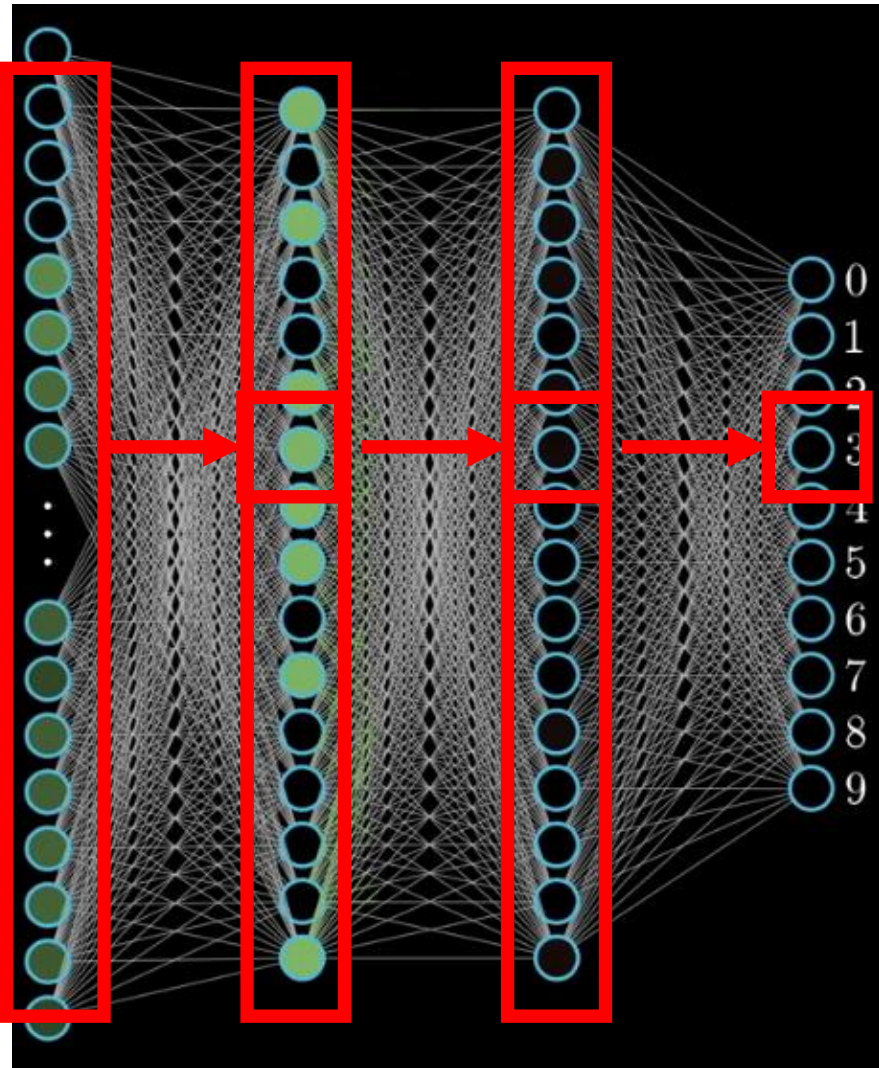
**Vector of activations from previous layer**

Recall that for multiclass logistic regression with K classes,

$$p(\text{Class} = k \mid x) \propto \sigma(w_k^T x + b_k)$$

# Backpropagation

[ Source : 3Blue1Brown : <https://www.youtube.com/watch?v=aircAruvnKk> ]



Activation at final layer involves weighted combination of activations at previous layer...

$$z_n = \sigma(W_n z_{n-1})$$

Which involves a weighted combination of the layer before it...

$$\sigma(w_n^T \sigma(w_{n-1}^T x))$$

And so on...

$$\sigma(w_n^T \sigma(w_{n-1}^T \sigma(w_{n-2}^T \sigma(\dots))))$$

# Computing the Derivative

Recall the **derivative chain rule**

$$\frac{d}{dw} f(g(w)) = \underbrace{\frac{d}{dz} f(z) \Big|_{z=g(w)}}_{\substack{\text{Derivative of } f \text{ at its} \\ \text{argument } g(w) \\ \text{e.g. treat } g(w) \text{ as a variable}}} \underbrace{\left( \frac{d}{dw} g(w) \right)}_{\substack{\text{Differentiate } g \text{ with} \\ \text{respect to } w}}$$

Alternatively we can write this as...

$$\frac{d}{dw} f(g(w)) = f'(g(w))g'(w)$$

# Derivative Chain Rule

**Example** Derivative of the logistic function,

$$\frac{d}{dz}\sigma(z) = \frac{d}{dz} \frac{1}{1 + e^{-z}}$$

$$f(x) = \frac{1}{x}$$

$$f'(x) = -\frac{1}{x^2}$$


$$g(z) = 1 + e^{-z}$$

$$g'(z) = -e^{-z}$$

$$\begin{aligned}\sigma'(z) &= f'(g(z))g'(z) \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= \sigma(z)(1 - \sigma(z))\end{aligned}$$

# Backpropagation

## Example

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$


$$\frac{d}{dz}\sigma(\sigma(z)) = \sigma(\sigma(z))(1 - \sigma(\sigma(z)))\frac{d}{dz}\sigma(z)$$

This is simply the derivative chain rule applied through the entire network, from the output to the input

# Backpropagation

- Implementation-wise all we need is a function that computes the derivative of each nonlinear activation
- We can repeatedly call this function, starting at the end of the network and moving backwards
- In practice, neural network implementations use *auto differentiation* to compute the derivative on-the-fly
- Can do this efficiently on *graphical processing units (GPUs)* on extremely large training datasets



# L1 Regularization

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}$$

Consider the case where  $w_i^* > 0$  for all  $i$ . There are two possible cases,

$$w_i^* \leq \frac{\alpha}{H_{i,i}} :$$

- Optimal value is just  $w_i=0$
- Contribution of  $J(w;X,y)$  is “overwhelmed” by L1 regularizer

$$w_i^* > \frac{\alpha}{H_{i,i}} :$$

- Shifts  $w_i$  in the direction of 0 by distance equal to  $\alpha/H$

Similar process for  $w < 0$  but in opposite direction.

# Sparse Representations

L1 regularization induces **sparse parameterization** – many parameters 0

**Representational sparsity** enforces many data elements 0 (or close to it)

Accomplished by same set of mechanisms as sparse param – norm penalty on representation

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\mathbf{h})$$

e.g. L1 penalty

## Sparse Parameterization

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}$$

$\mathbf{y} \in \mathbb{R}^m$                        $\mathbf{A} \in \mathbb{R}^{m \times n}$                        $\mathbf{x} \in \mathbb{R}^n$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

$\mathbf{y} \in \mathbb{R}^m$                        $\mathbf{B} \in \mathbb{R}^{m \times n}$                        $\mathbf{h} \in \mathbb{R}^n$

## Sparse Representation

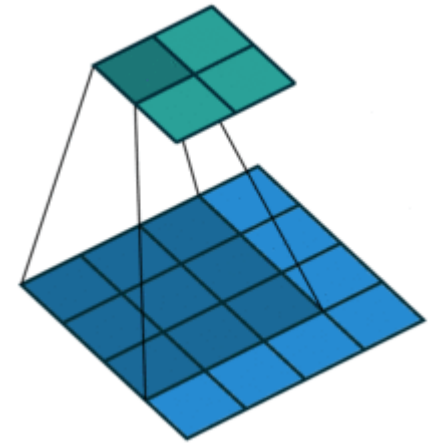
# Label Smoothing

- Many datasets have some mistakes in labels  $y$
- Inject noise in labels at output
  - Assume label is correct with probability  $1-e$  (for some small  $e$ )
  - Otherwise any other label is assigned
- Can incorporate this into cost function analytically
- Label smoothing regularizes model based on softmax
  - Replaces hard assignment with  $1-e$  and  $e/(k-1)$  ; for  $k$  labels
  - Can use standard cross-entropy loss with soft targets

# Convolution: Some Intuition

Define the convolution of filter  $f$  on image  $I$  as:

$$(I * f)(x) = \sum_m \sum_n I(x - m, y - n) f(m, n)$$



Many ML libraries *actually* implement cross-correlation:

$$(f * I)(x) = \sum_m \sum_n I(x + m, y + n) f(m, n)$$

*Learning finds good values for the convolution filter...*

# VGGNet (2014): 7.3% error on ImageNet

- Mimic large convolutional filters with multiple small (3x3) convolutional filters
- Every time it halves the spatial size, double the # of filters

INPUT: [224x224x3] memory:  $224*224*3=150K$  params: 0 (not counting biases)

CONV3-64: [224x224x64] memory:  $224*224*64=3.2M$  params:  $(3*3*3)*64 = 1,728$

CONV3-64: [224x224x64] memory:  $224*224*64=3.2M$  params:  $(3*3*64)*64 = 36,864$

POOL2: [112x112x64] memory:  $112*112*64=800K$  params: 0

CONV3-128: [112x112x128] memory:  $112*112*128=1.6M$  params:  $(3*3*64)*128 = 73,728$

CONV3-128: [112x112x128] memory:  $112*112*128=1.6M$  params:  $(3*3*128)*128 = 147,456$

POOL2: [56x56x128] memory:  $56*56*128=400K$  params: 0

CONV3-256: [56x56x256] memory:  $56*56*256=800K$  params:  $(3*3*128)*256 = 294,912$

CONV3-256: [56x56x256] memory:  $56*56*256=800K$  params:  $(3*3*256)*256 = 589,824$

CONV3-256: [56x56x256] memory:  $56*56*256=800K$  params:  $(3*3*256)*256 = 589,824$

POOL2: [28x28x256] memory:  $28*28*256=200K$  params: 0

CONV3-512: [28x28x512] memory:  $28*28*512=400K$  params:  $(3*3*256)*512 = 1,179,648$

CONV3-512: [28x28x512] memory:  $28*28*512=400K$  params:  $(3*3*512)*512 = 2,359,296$

CONV3-512: [28x28x512] memory:  $28*28*512=400K$  params:  $(3*3*512)*512 = 2,359,296$

POOL2: [14x14x512] memory:  $14*14*512=100K$  params: 0

CONV3-512: [14x14x512] memory:  $14*14*512=100K$  params:  $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory:  $14*14*512=100K$  params:  $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory:  $14*14*512=100K$  params:  $(3*3*512)*512 = 2,359,296$

POOL2: [7x7x512] memory:  $7*7*512=25K$  params: 0

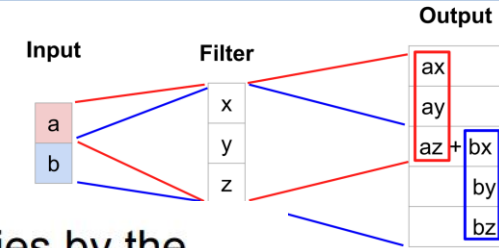
FC: [1x1x4096] memory: 4096 params:  $7*7*512*4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params:  $4096*4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params:  $4096*1000 = 4,096,000$

ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
Input (224 × 224 RGB image)			
conv3-64	conv3-64	conv3-64	conv3-64
<b>conv3-64</b>	conv3-64	conv3-64	conv3-64
maxpool			
conv3-128	conv3-128	conv3-128	conv3-128
<b>conv3-128</b>	conv3-128	conv3-128	conv3-128
maxpool			
conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256
	<b>conv1-256</b>	<b>conv3-256</b>	conv3-256
maxpool			
conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512
	<b>conv1-512</b>	<b>conv3-512</b>	conv3-512
maxpool			
conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512
	<b>conv1-512</b>	<b>conv3-512</b>	conv3-512
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

# 1D transposed convolution: matrix form



We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X \vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Transposed convolution multiplies by the transpose of the same matrix:

$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

Example: 1D transposed conv, kernel size=3, stride=2, padding=0