



Computer
Science

CSC 480/580: Principles of Machine Learning

Neural Networks & Deep Learning

Jason Pacheco

Basis Functions

Basis functions transform linear models into nonlinear ones...

Linear Regression

$$y = w^T x$$



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

**Classification
(Logistic Regression)**

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



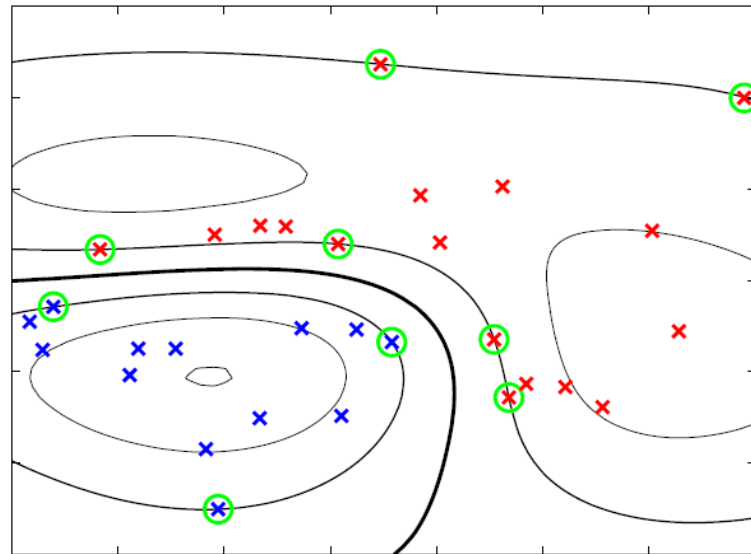
$$y = \sigma(w^T \phi(x))$$

...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

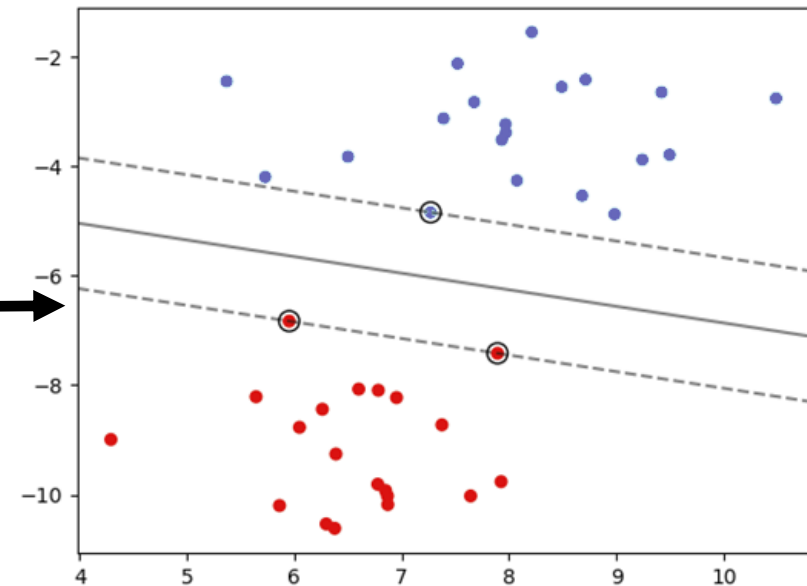


Ignore the circled points...I
reused these from the SVM slides

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...



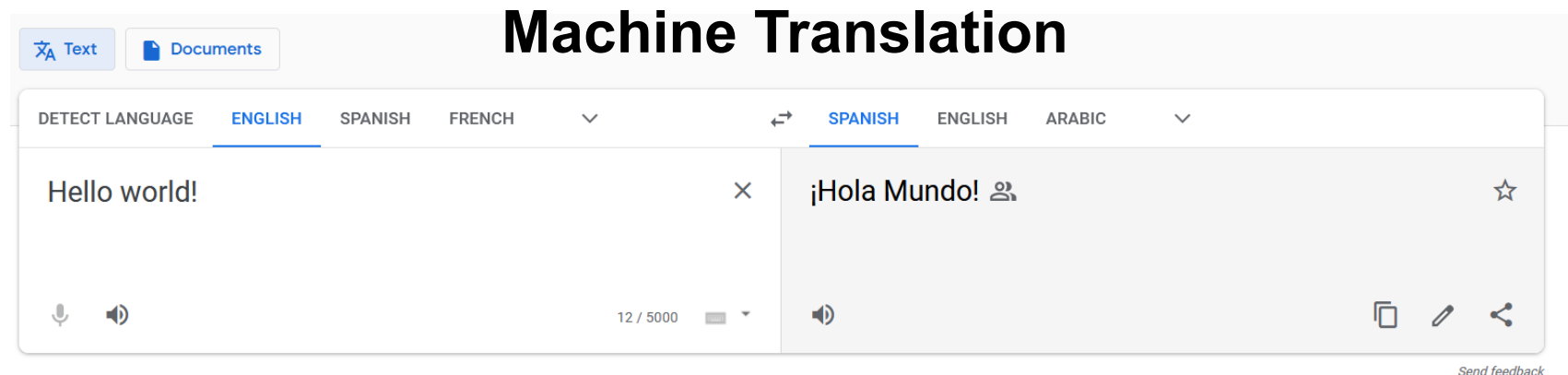
AI Chat Bots



Self-Driving Cars



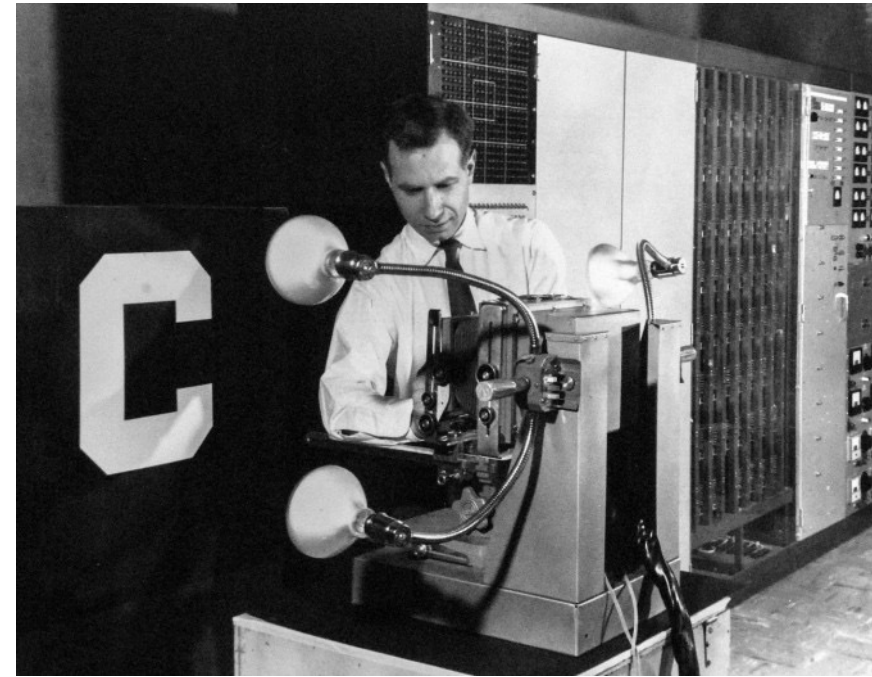
Creepy Robots



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.)

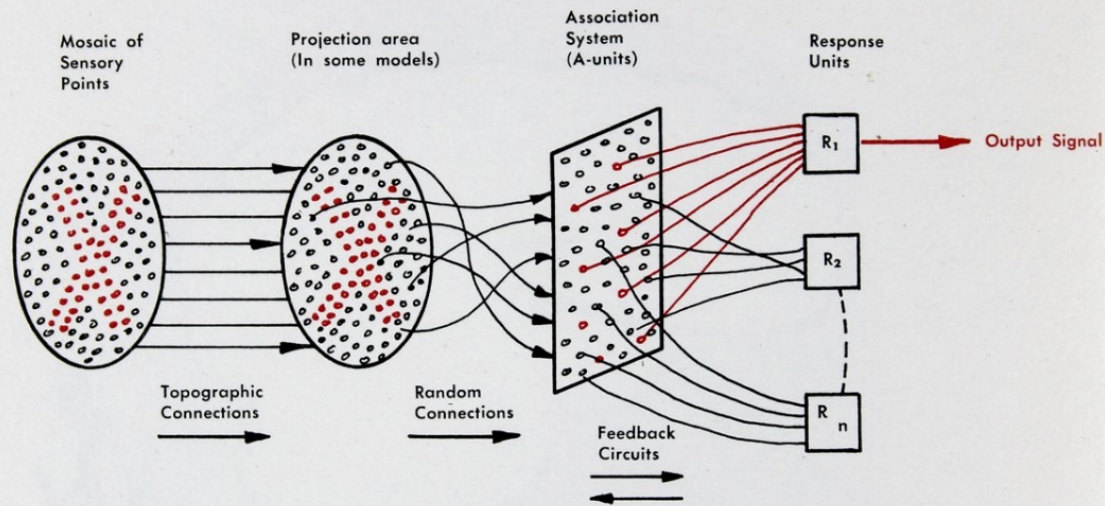
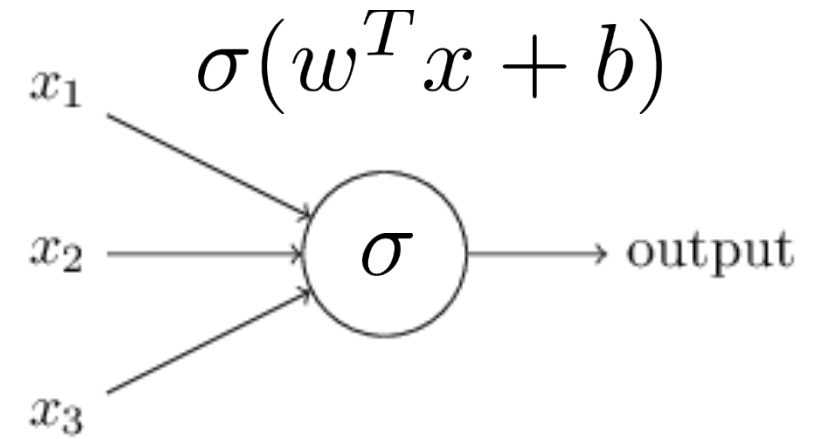


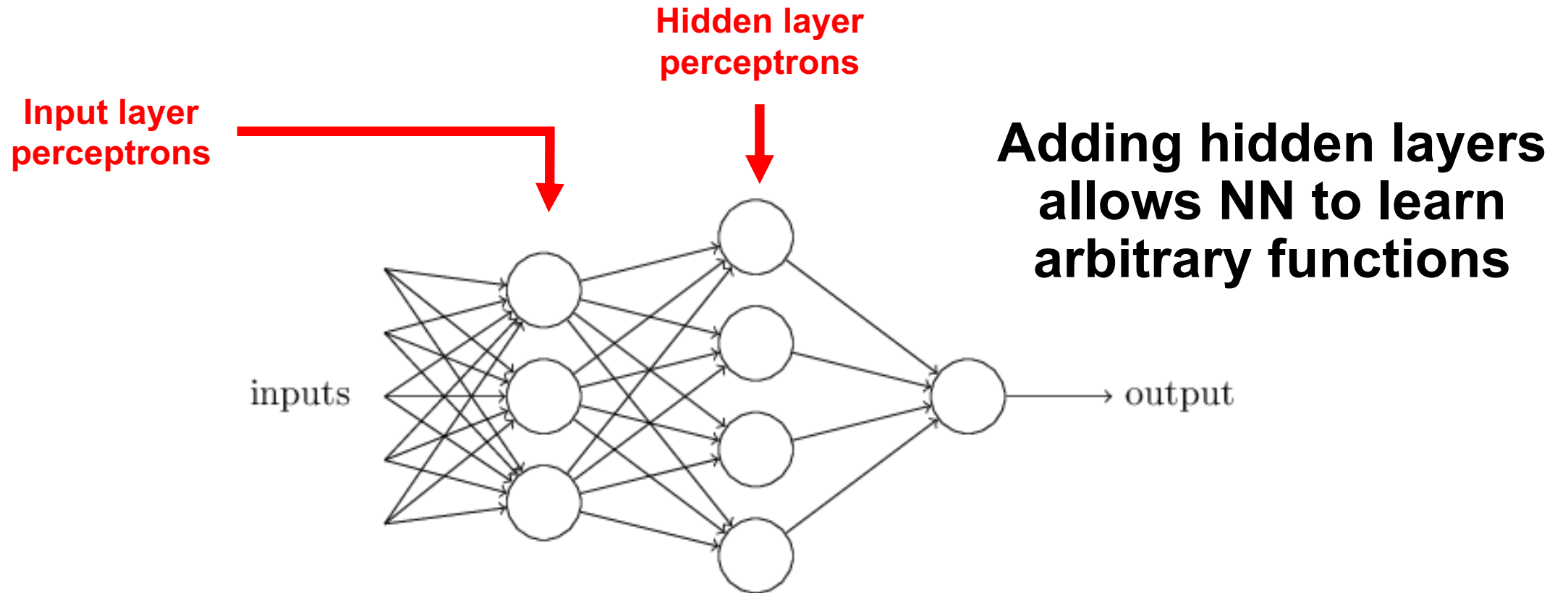
FIG. 2 — Organization of a perceptron.

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 (MLP)

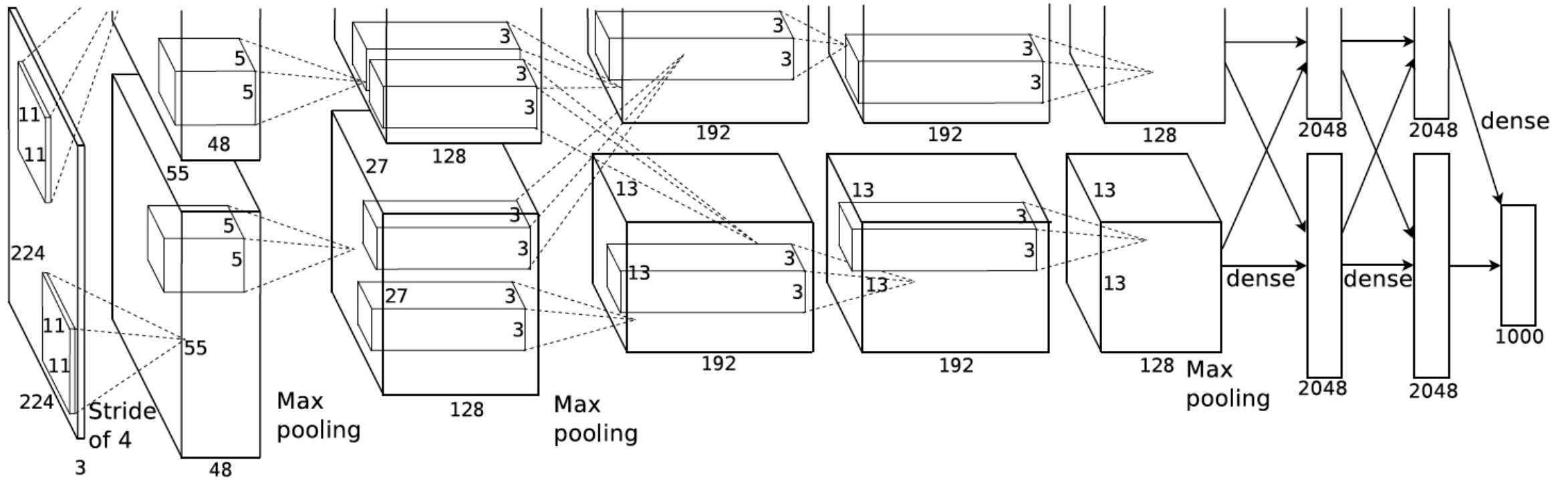


This is the quintessential *Neural Network*...
...also called *Feed Forward Neural Net* or *Artificial Neural Net*

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

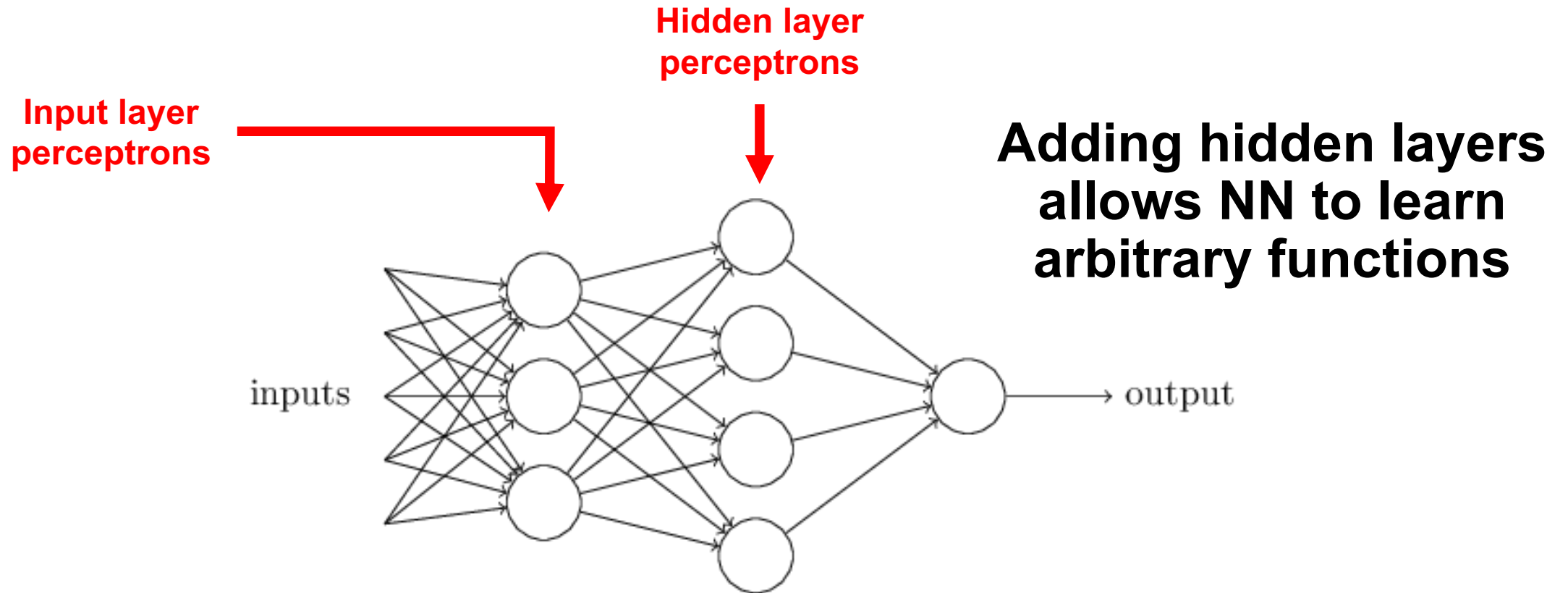
Modern Neural Networks

Modern *Deep Neural networks* have many hidden layers



...and have billions, if not trillions of parameters to learn

Recap: Neural Networks



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

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

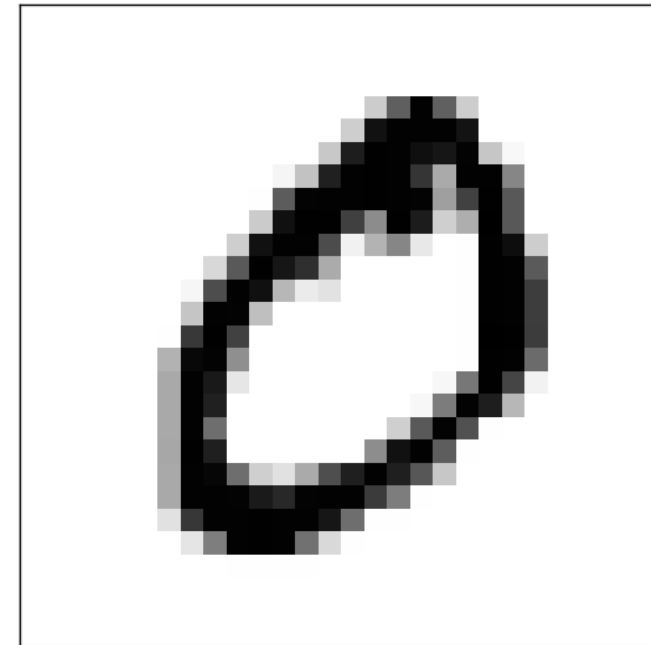
Handwritten Digit Classification

Classifying handwritten digits is the “Hello World” of NNs



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

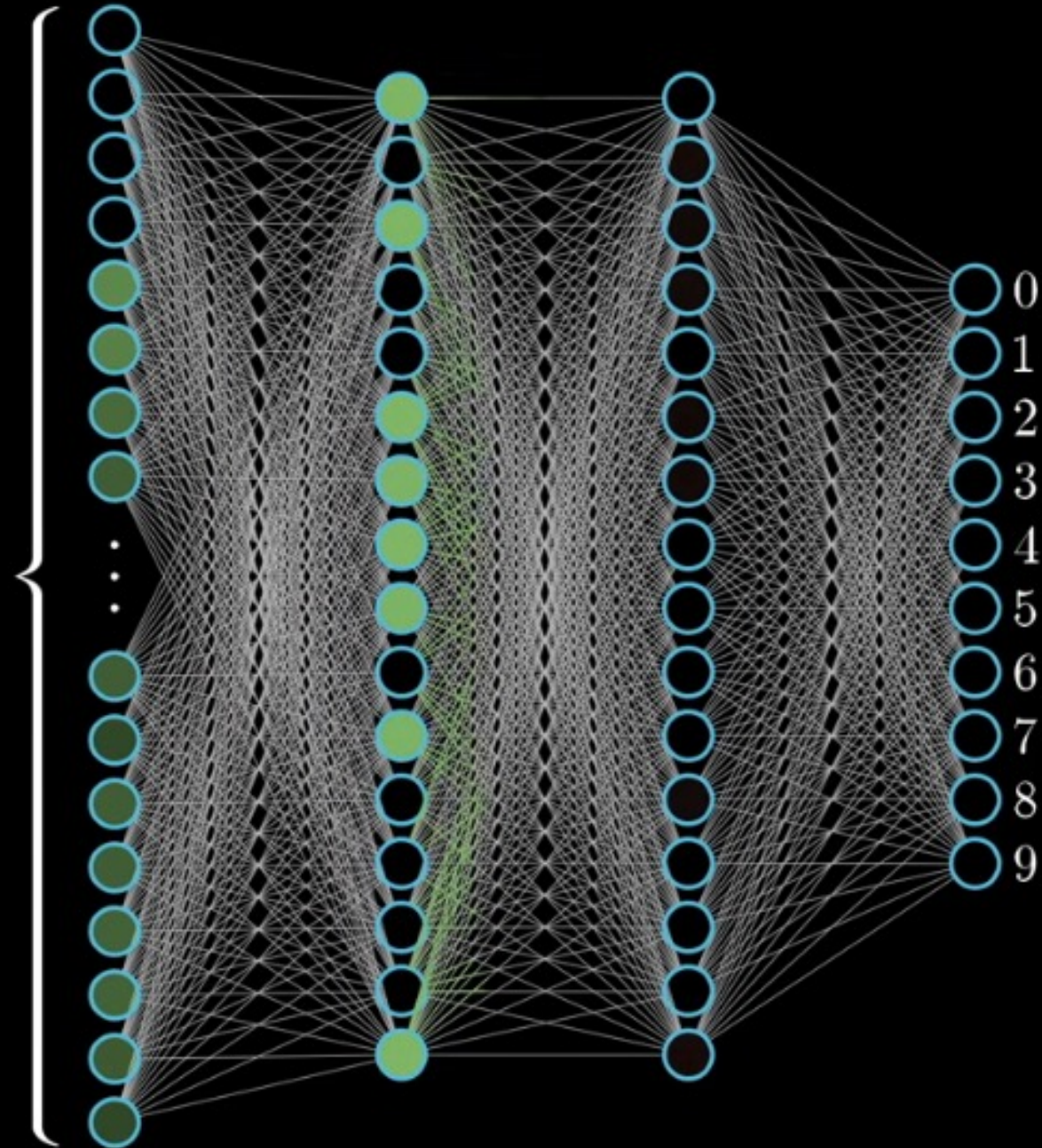
Each character is centered in a $28 \times 28 = 784$ pixel grayscale image



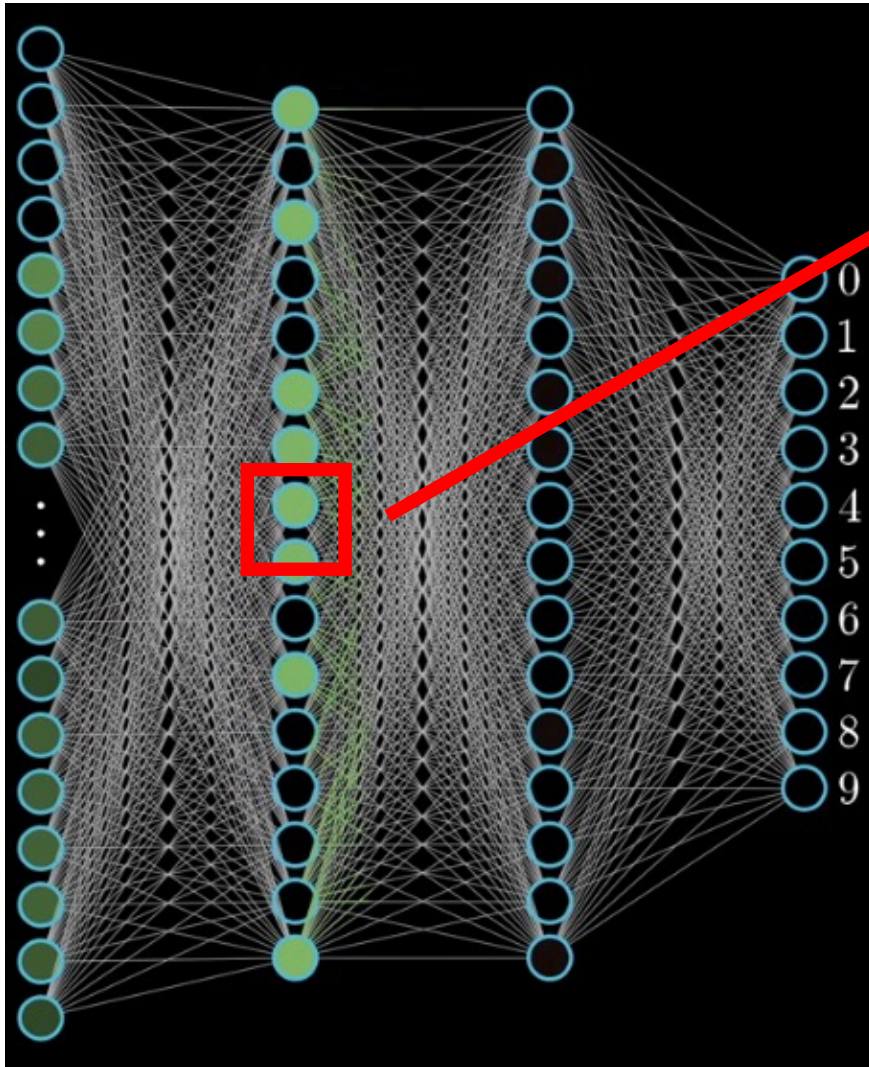


784

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,

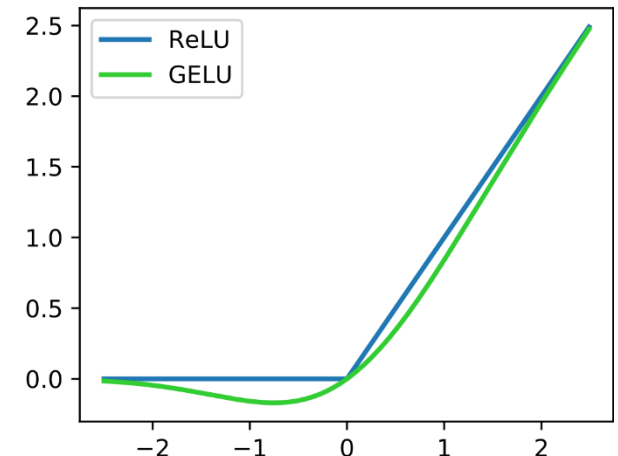
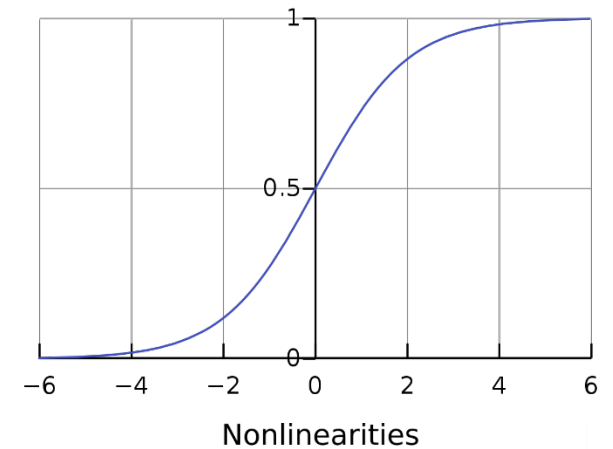
$$\sigma(w_1x_1 + w_2x_2 + \dots + w_nx_n + b) = \sigma(w^T x + b)$$

An early choice was the *logistic function*,

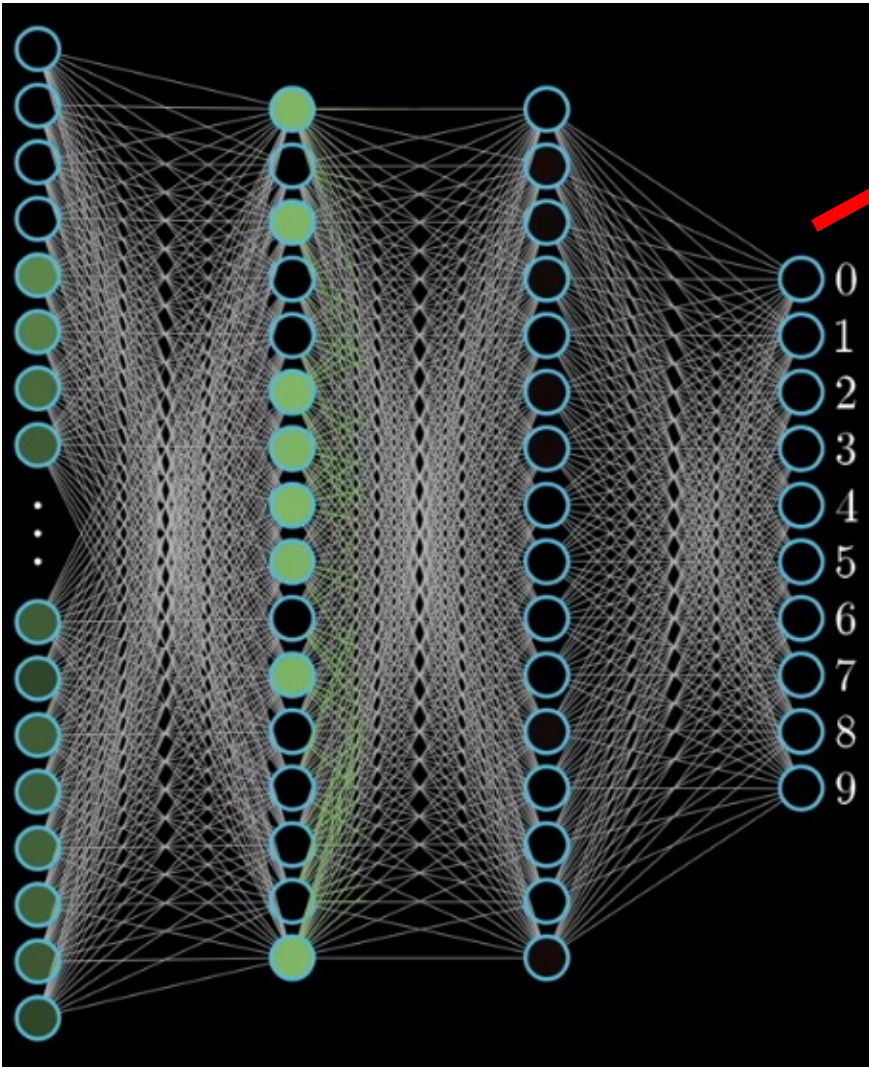
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

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



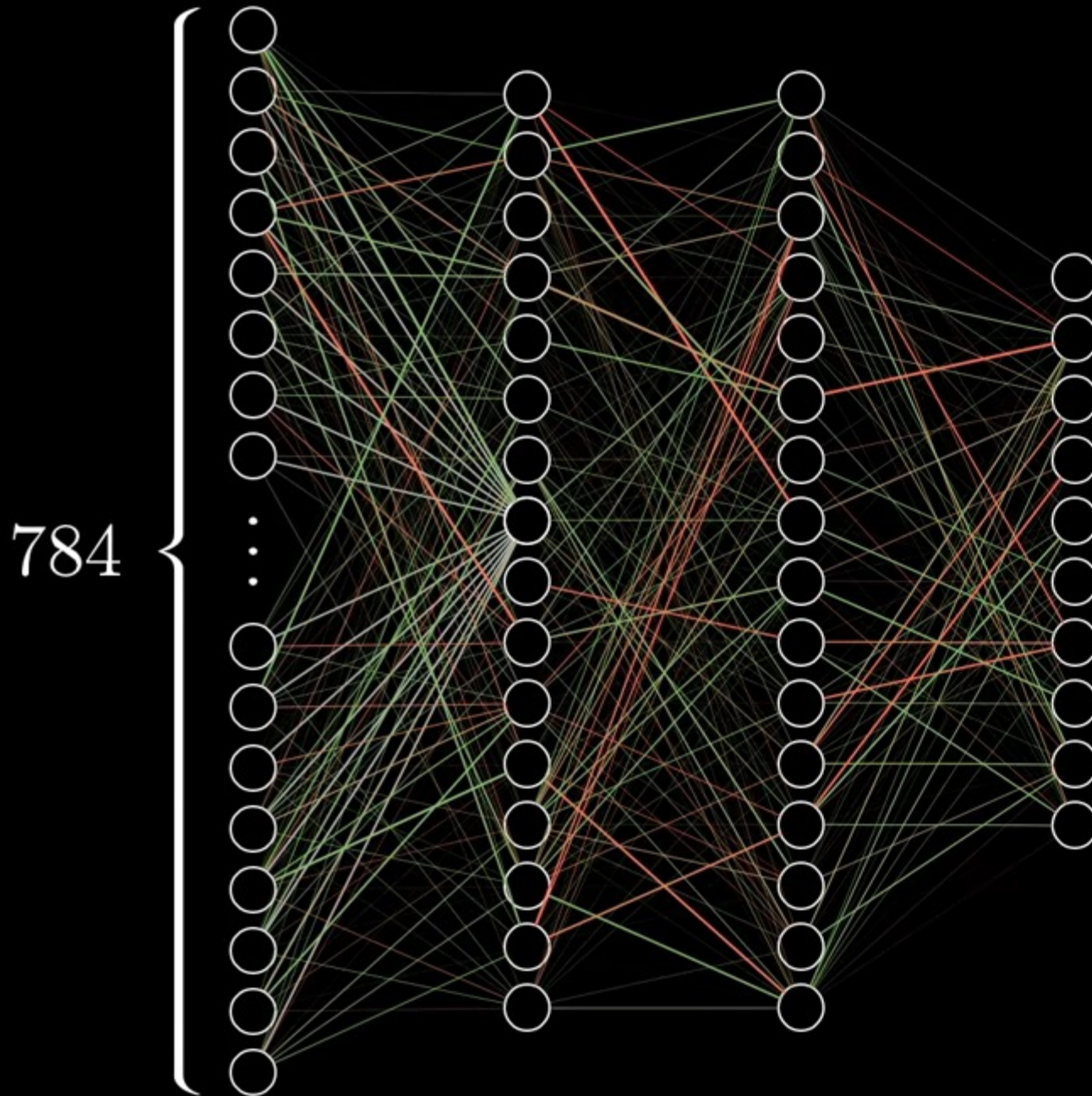
Final layer is typically a linear model... each output node is computed by

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

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

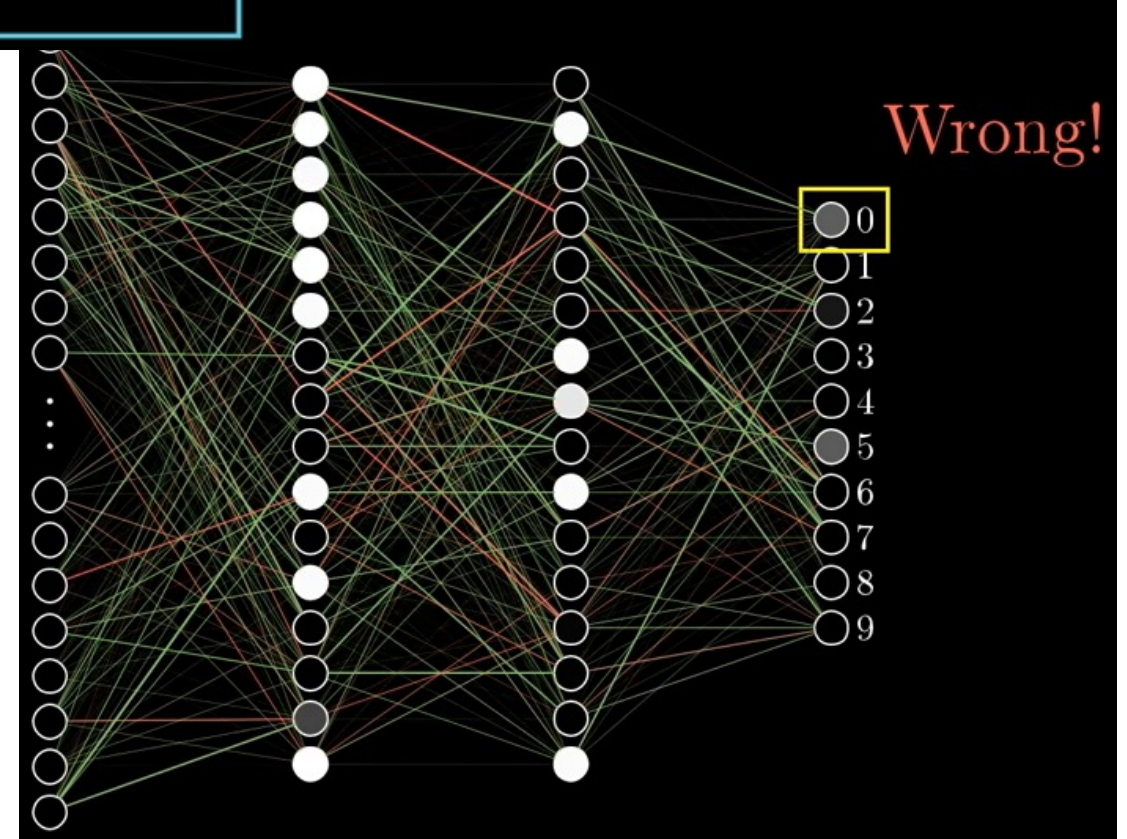
Training Multilayer Perceptron

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

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



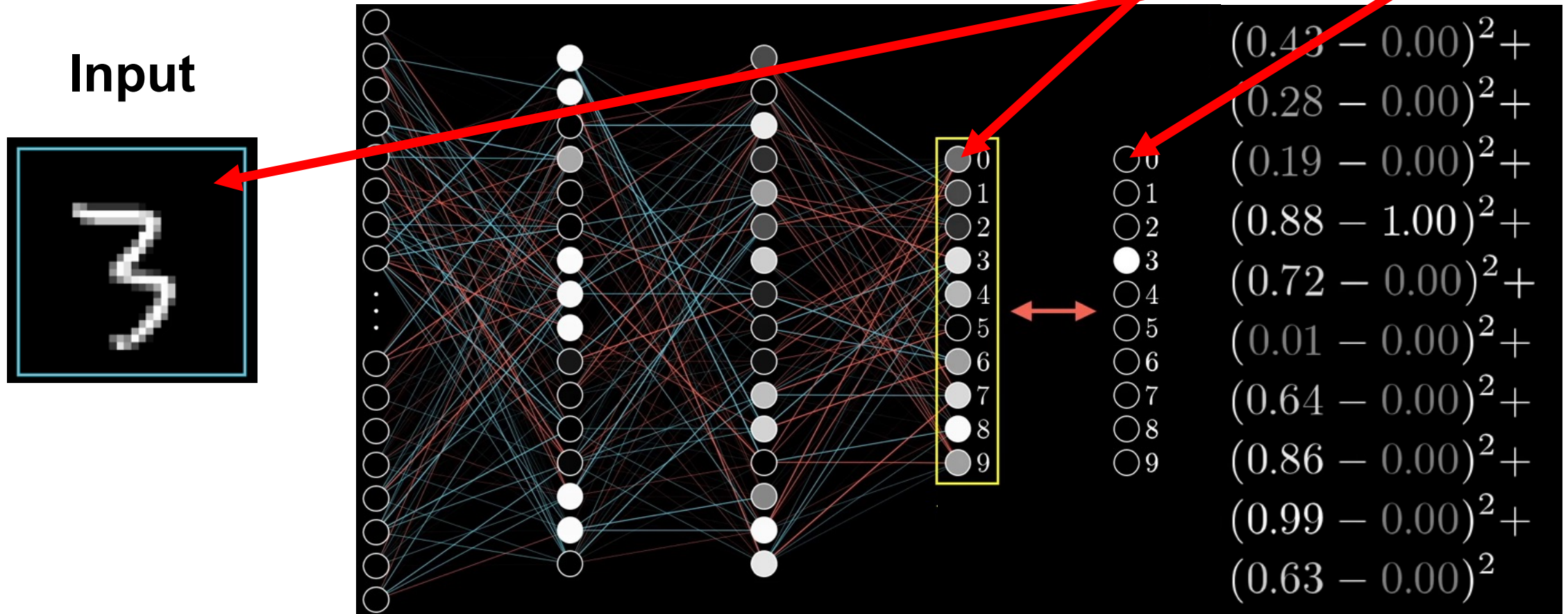
For each training example, predict label and adjust weights according to how it is doing...



- 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

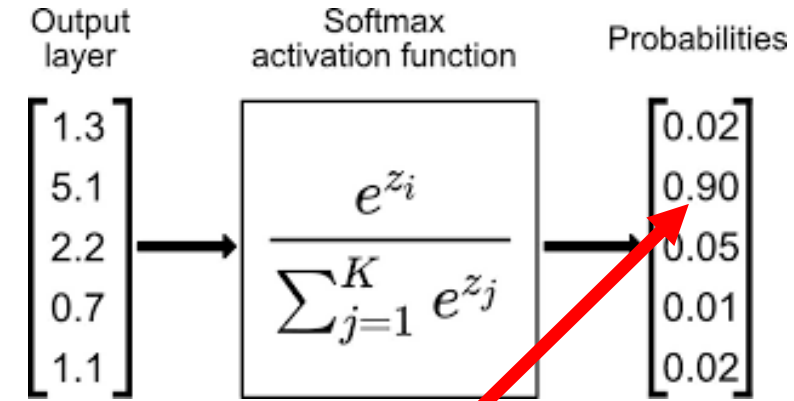
$$\sigma(\vec{f})_c = \frac{e^{f_c}}{\sum_{j=1}^C e^{f_j}}, c = 1, \dots, C$$

probability estimate of each class given example

- Cross-entropy loss for training

$$\ell(\vec{p}, y) = \log\left(\frac{1}{p_y}\right) \quad \text{Smaller if } p_y \text{ is larger}$$

measures the model's "surprise" about seeing this example having label y



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

Training Multilayer Perceptron

Our loss function for i^{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...

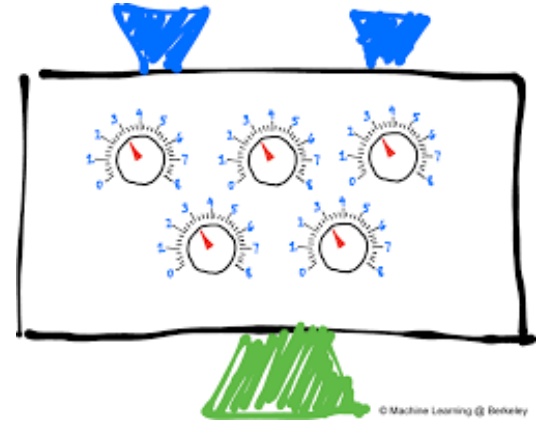
$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^m \ell_i(\theta)$$

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

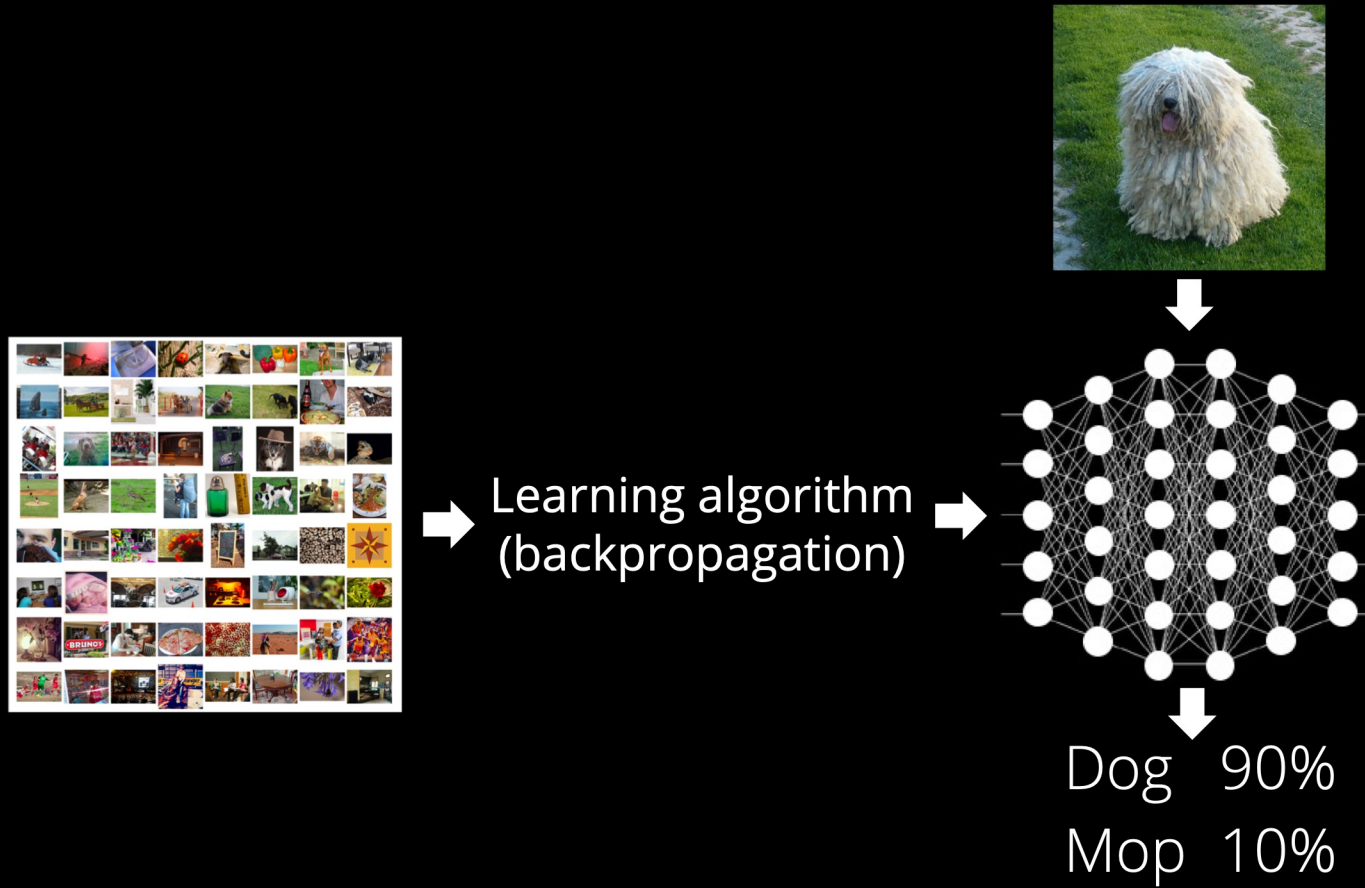
Gradient descent!

Learning algorithm intuition

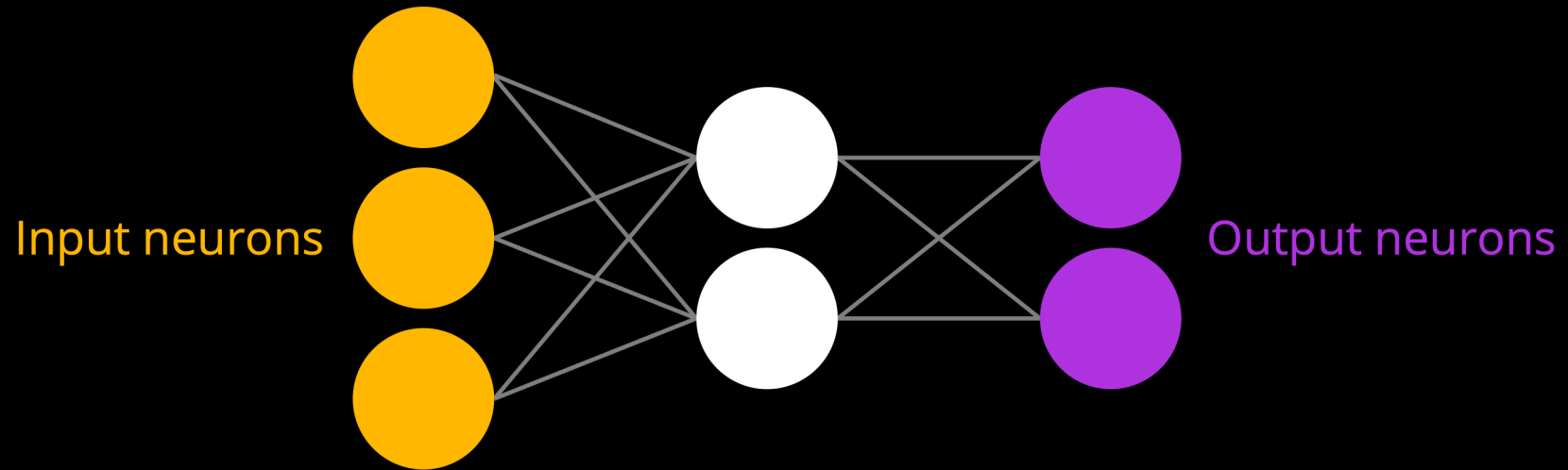
- Gradient descent: Move in direction of greatest local improvement (greedily)
- “Knob turning”
 - “knob” = weight of an edge
 - If a neuron increases the probability of an incorrect prediction, its knobs will be turned down.
 - If a neuron increases the probability of a correct prediction, its knobs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



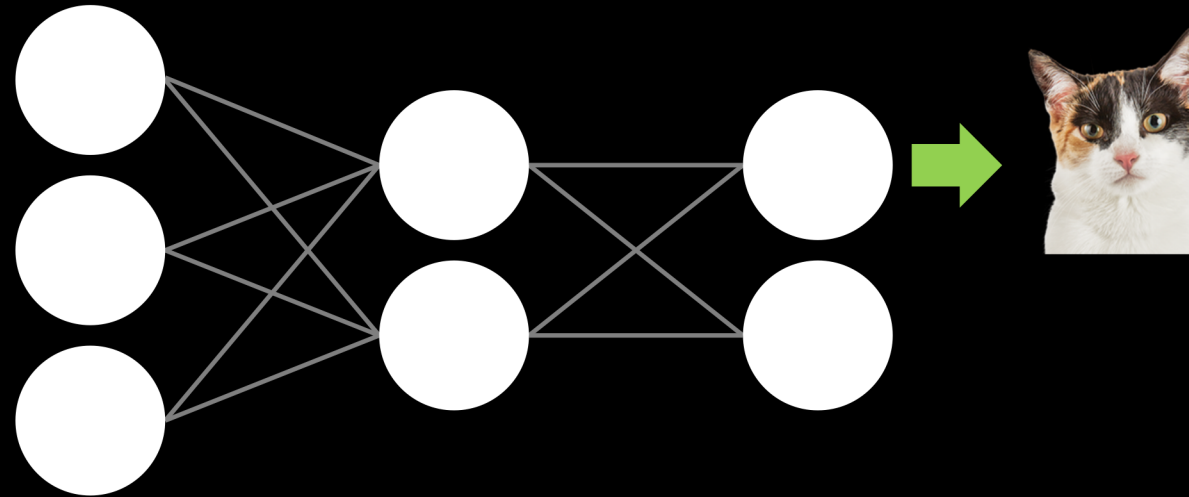
Deep learning, a field of machine learning



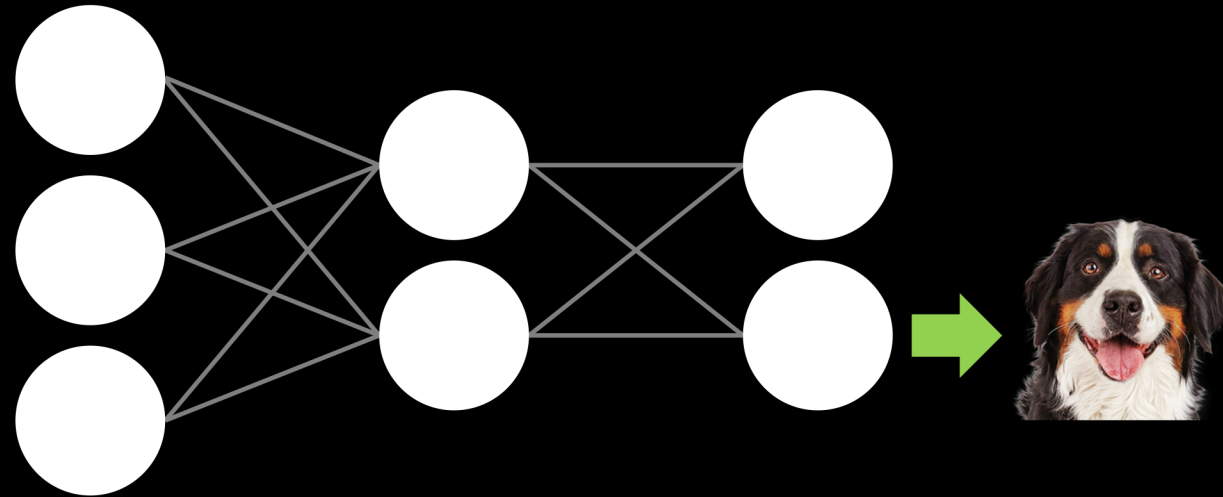
Deep learning with backpropagation



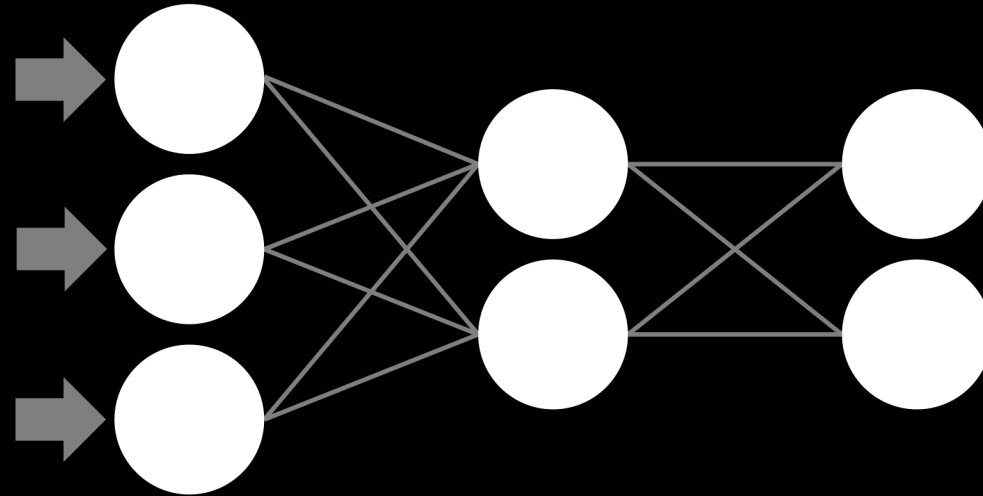
Deep learning with backpropagation



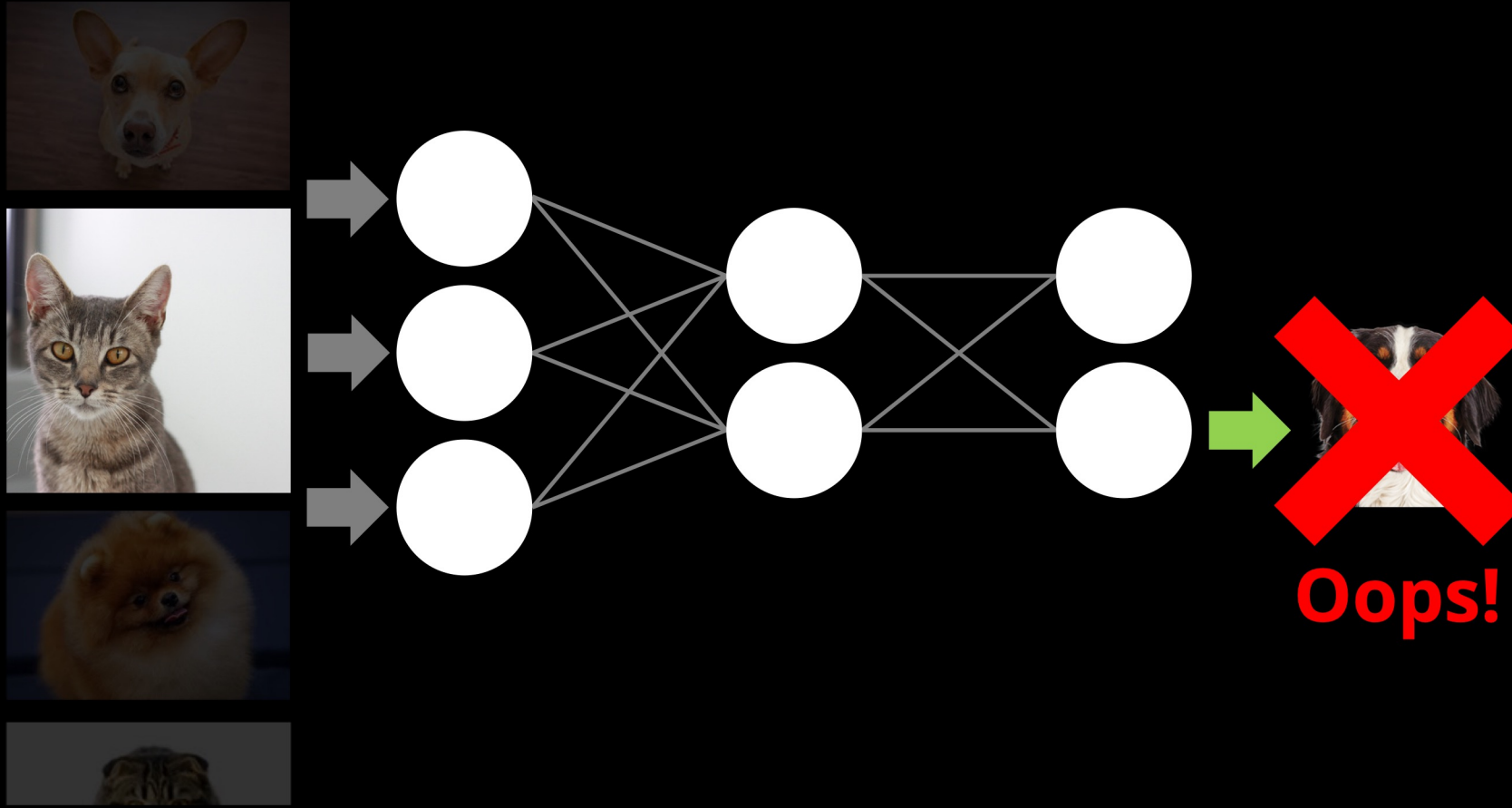
Deep learning with backpropagation



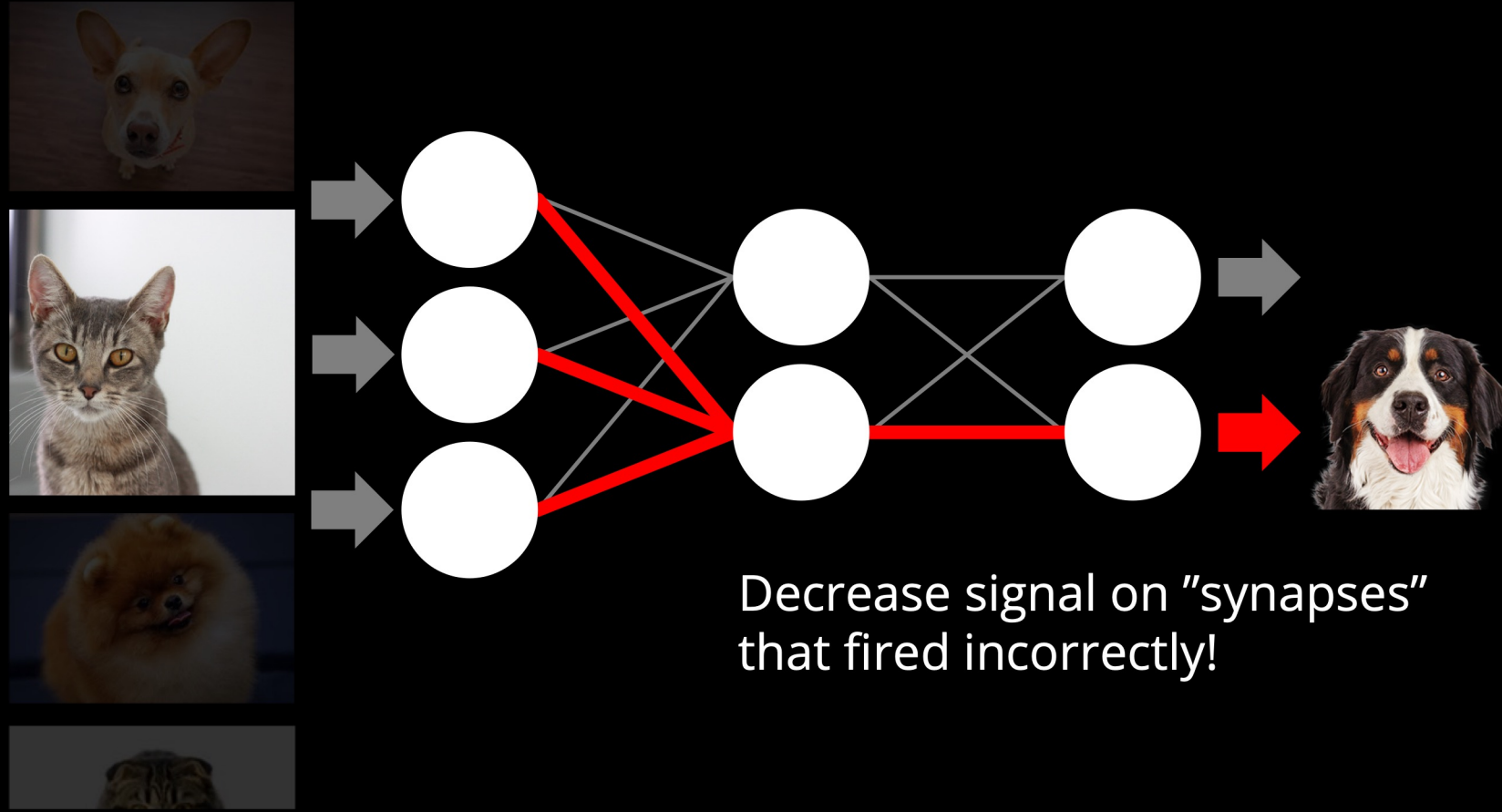
Deep learning with backpropagation



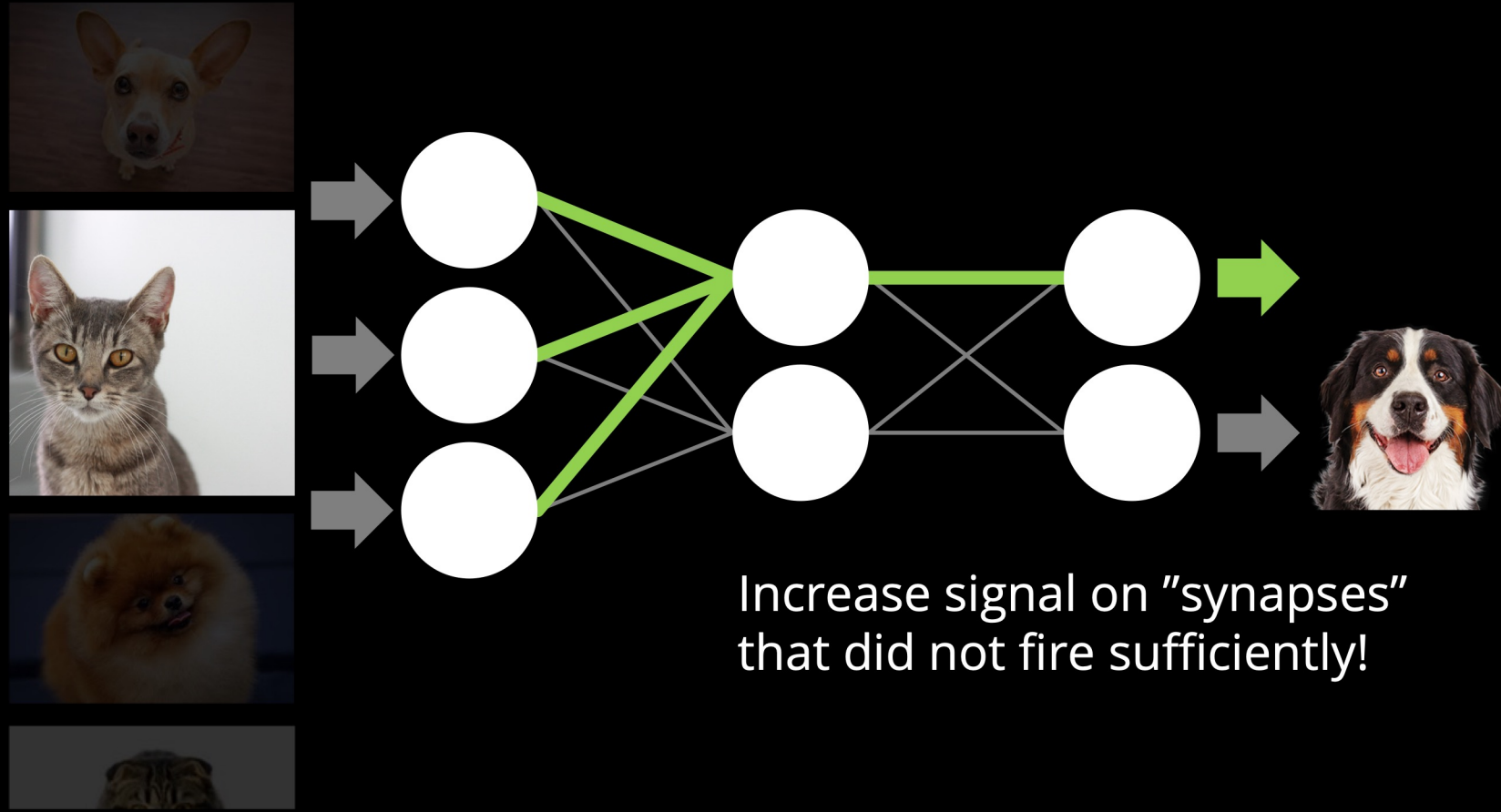
Deep learning with backpropagation



Deep learning with backpropagation



Deep learning with backpropagation



xkcd: Machine Learning

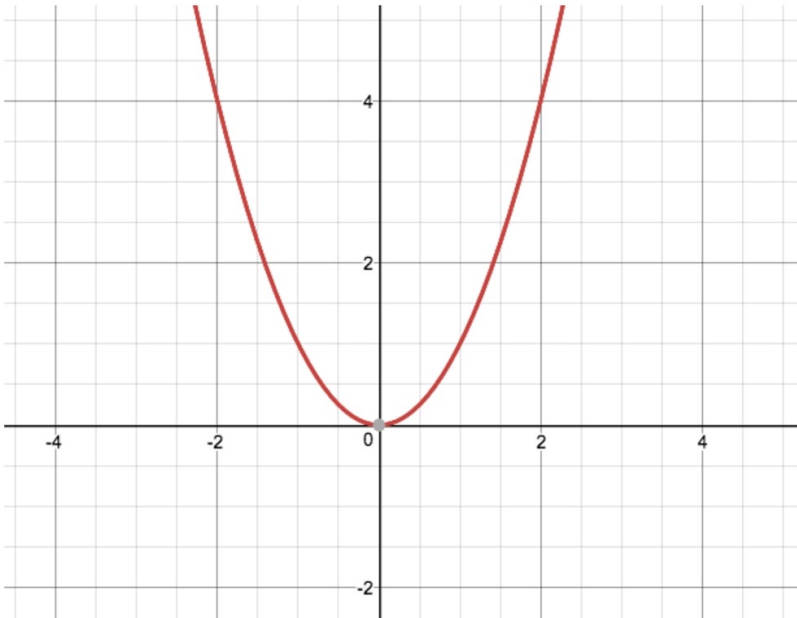
- Optimizing ML models can sometimes feel like this..
- Gradient descent automates a big part of it
- What remains for us to do?
 - Hyperparameter optimization
 - Neural network architecture design
 - Learning algorithm selection
 - ..



Training Multilayer Perceptron

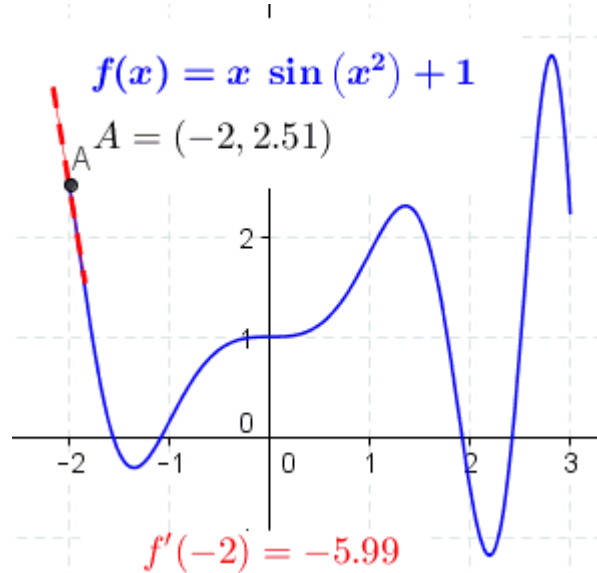
Need to find local / global optimal solution...

Convex Cost Function



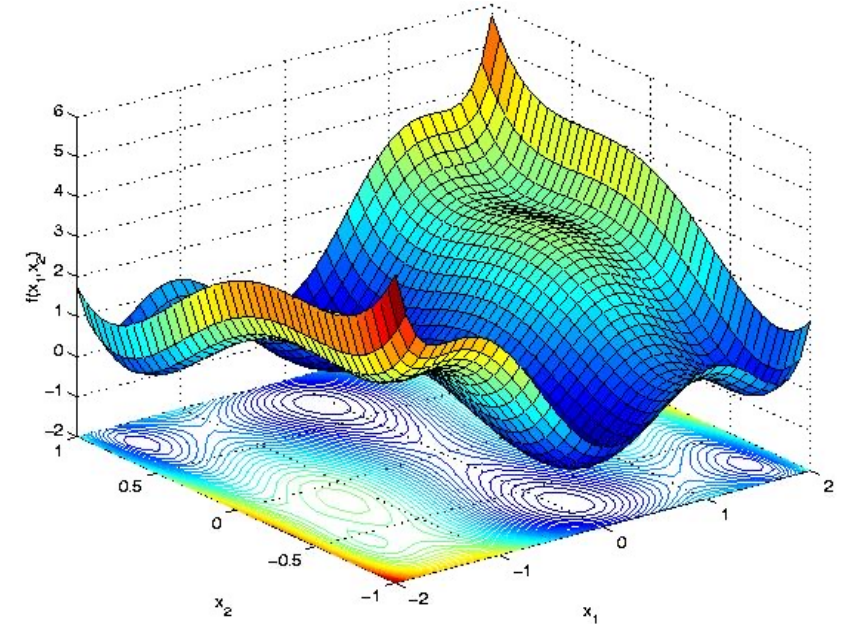
Easy!

Non-convex Cost Function



Hard!

High-Dimensional Non-convex

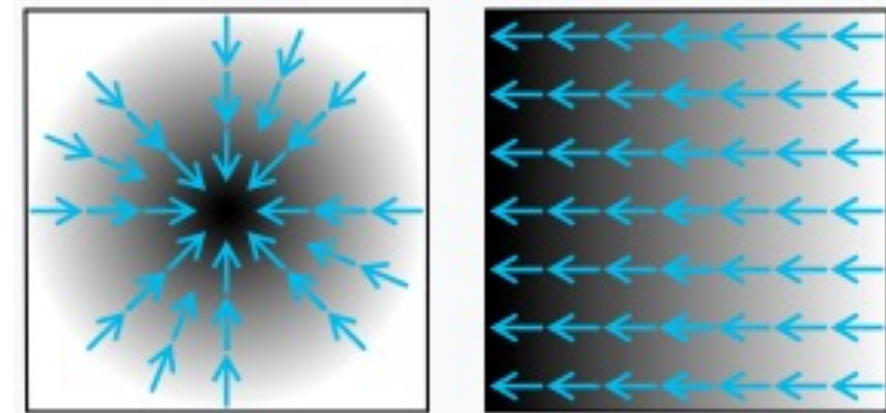
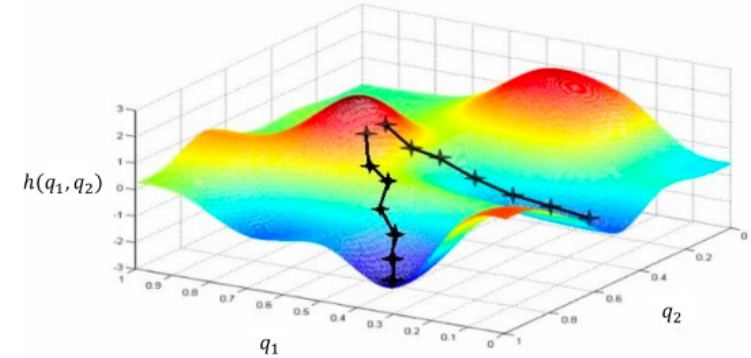


Hard!

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 θ
- 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 $\mathcal{L}$
- Examples: $\mathcal{L}(\theta) = ||\theta||^2$ (left),
 $\mathcal{L}(\theta) = x^T \theta$ (right)



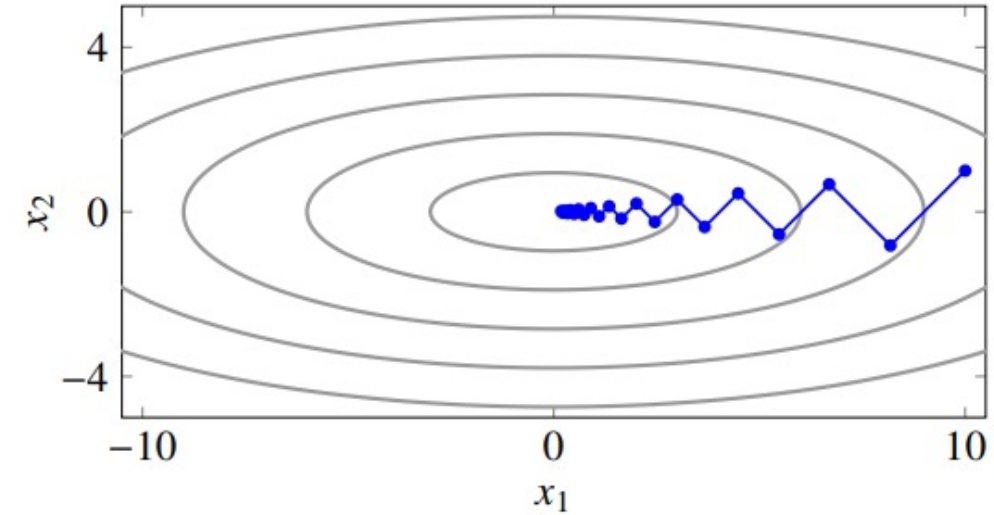
Gradient descent: example

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

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

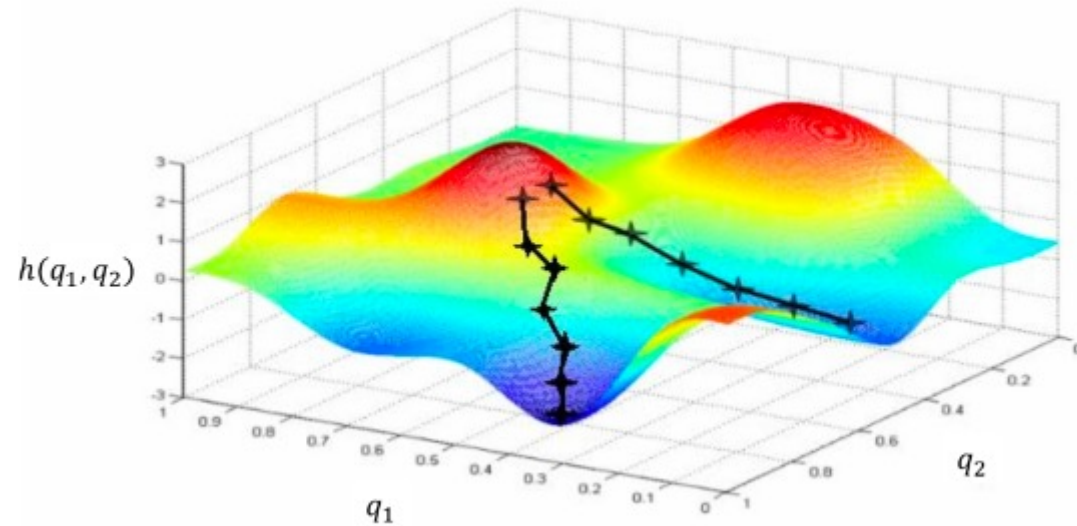
- $$\begin{pmatrix} \theta_{t+1}(1) \\ \theta_{t+1}(2) \end{pmatrix} = \begin{pmatrix} \theta_t(1) \\ \theta_t(2) \end{pmatrix} - \alpha \begin{pmatrix} \theta_t(1) \\ 4\theta_t(2) \end{pmatrix} = \begin{pmatrix} (1 - \alpha)\theta_t(1) \\ (1 - 4\alpha)\theta_t(2) \end{pmatrix} = \dots = \begin{pmatrix}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

- When α is small, $\theta_t \rightarrow (0,0)$ - global minimizer of $\mathcal{L}$ **Vanishing as $t \rightarrow \infty$**



GD: remarks

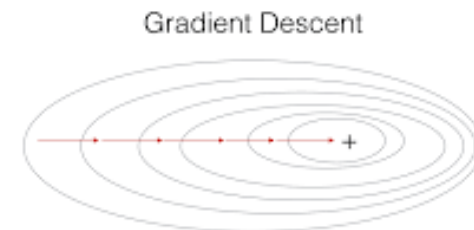
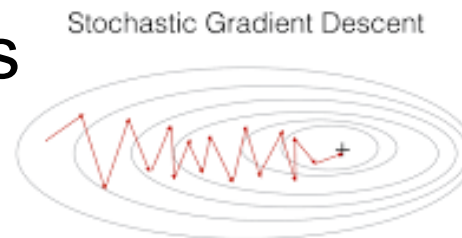
- Does not promise to converge to global optima



- 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



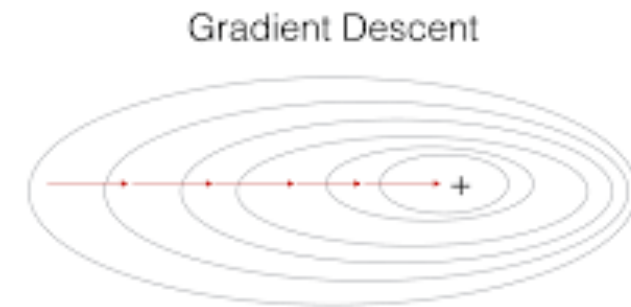
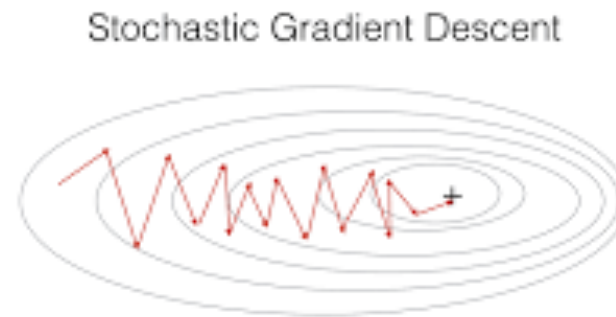
Stochastic gradient descent (SGD)

- 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: $\mathbb{E}[g_t] = \nabla \mathcal{L}(\theta_t)$
- More popular choice (Mini-batch SGD): $g_t = \frac{1}{k} \sum_{i \in S_t} \nabla \ell_i(\theta_t)$ (S_t is a random size- k subset of $\{1, \dots, m\}$) is also an unbiased estimate

Thanks to modern GPU's parallel processing capability

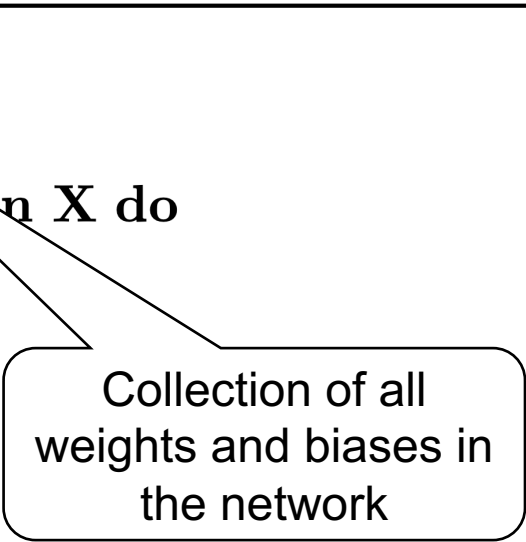
Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
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
```



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

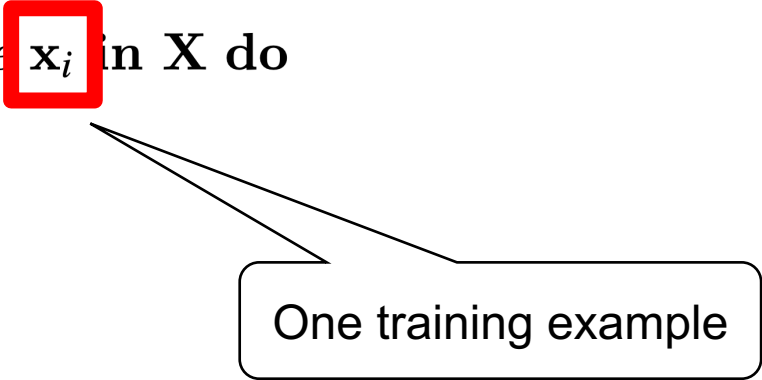
```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



Collection of all
weights and biases in
the network

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

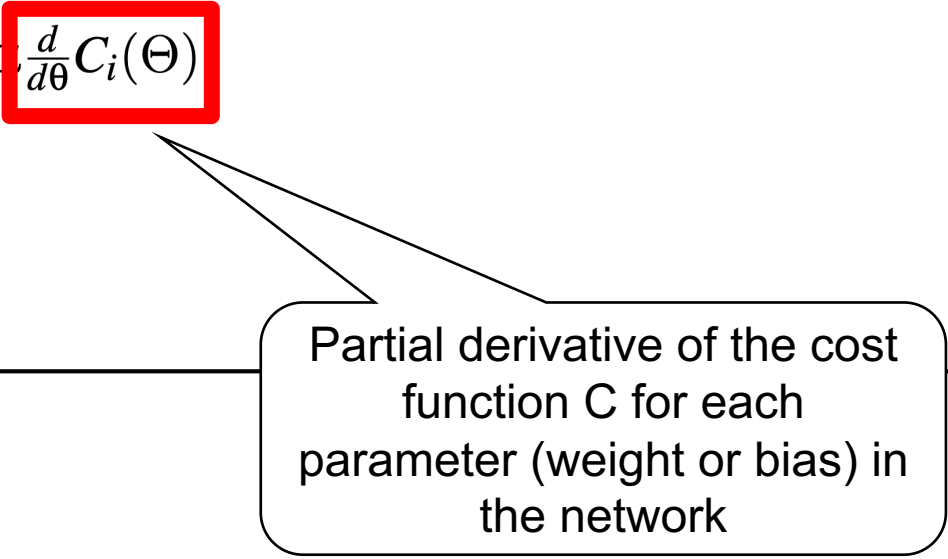
```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



One training example

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

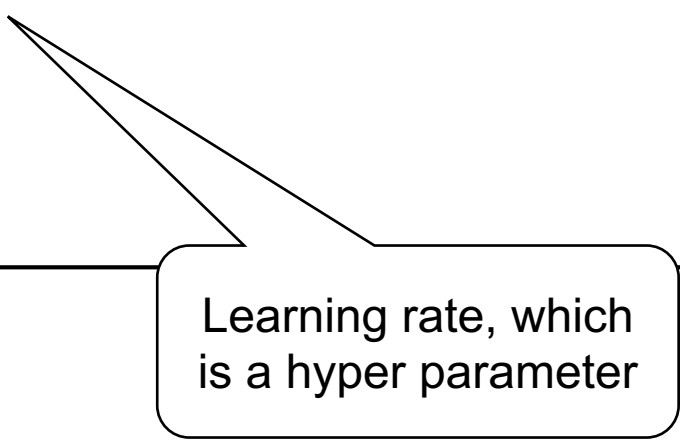
```
1 initialize parameters in  $\Theta$ 
2 while not converged do
3   for each training example  $\mathbf{x}_i$  in  $\mathbf{X}$  do
4     for each  $\theta$  in  $\Theta$  do
5        $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\theta)$ 
6     end
7   end
8 end
```



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

Neural network demo

- [Tensorflow neural network playground](#)

Visualizes:

- learning curves (training & test losses vs number of iterations)

Let's try:

- using no hidden layers
- using 1 hidden layer with 4 nodes
- playing with a harder dataset

Computing the Derivative

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

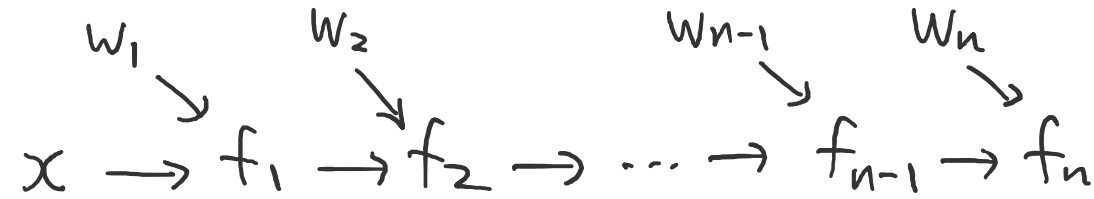
$$\nabla \mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla \ell_i(\theta)$$

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

Key conceptual tool: Computation graph

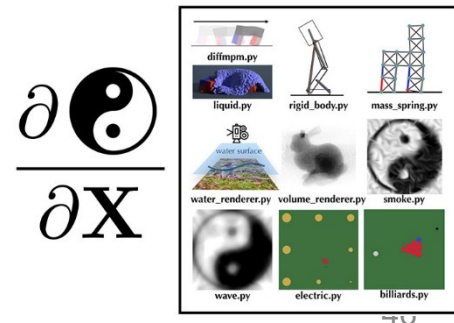
- A DAG that describes the order of computation in a general computational process

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



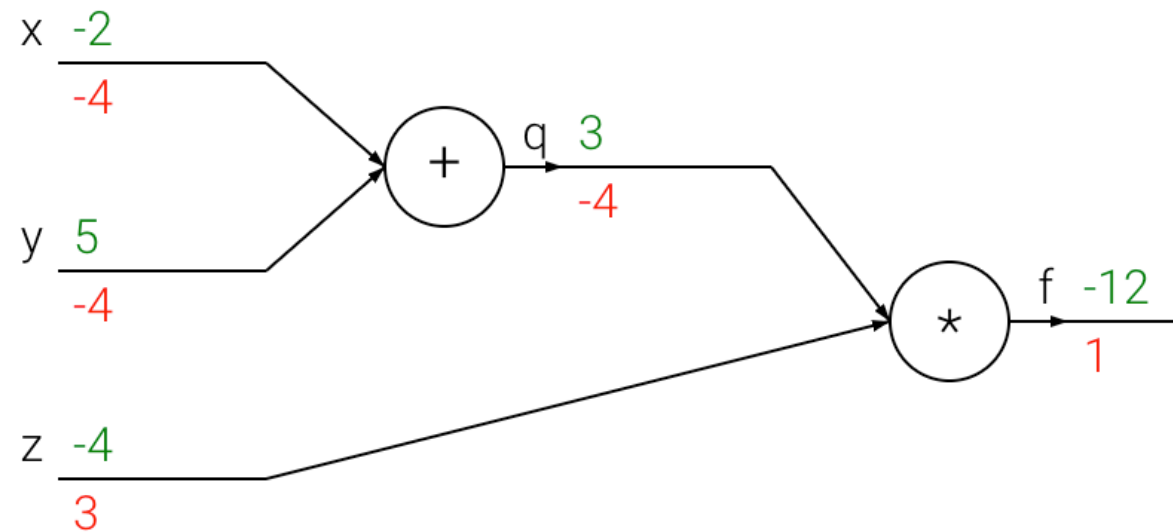
- Has more general application beyond training neural networks
 - Differentiable physics simulation engine (Hu et al 2020)
 - Differentiable ray-tracing (e.g. Yang et al 2022)
 - ICML workshops on “differentiable almost everything”

<https://differentiable.xyz/>



Key tool: Computation graph

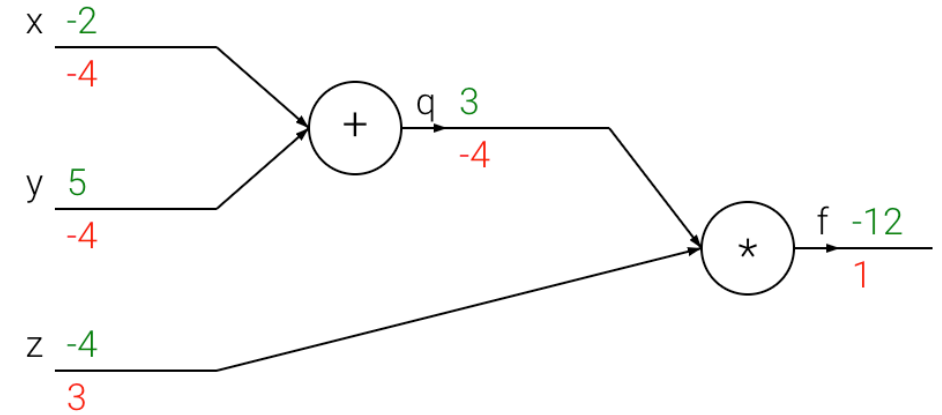
- Sometimes useful to highlight the *operation* that computes each variable as well
- E.g. $q = F_q(x, y) = x + y$; $f = F_f(q, z) = q \cdot z$



- **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

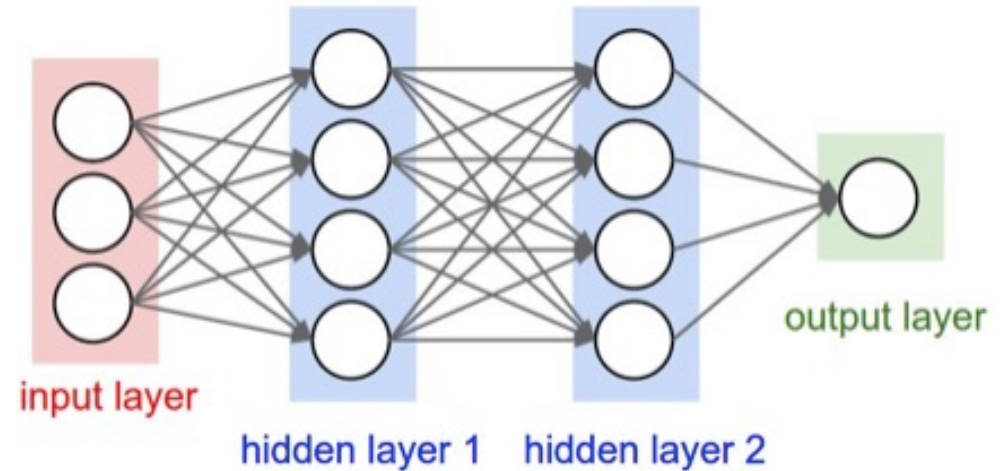
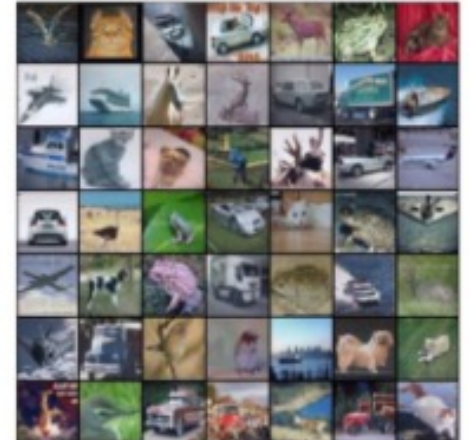
Recap: neural networks for supervised learning

- Step 1: setting up data
- Step 2: setting up neural network architecture
- Step 3: setting up the loss
 - square loss
 - cross-entropy loss
- Step 4: optimize the loss, i.e., solving

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

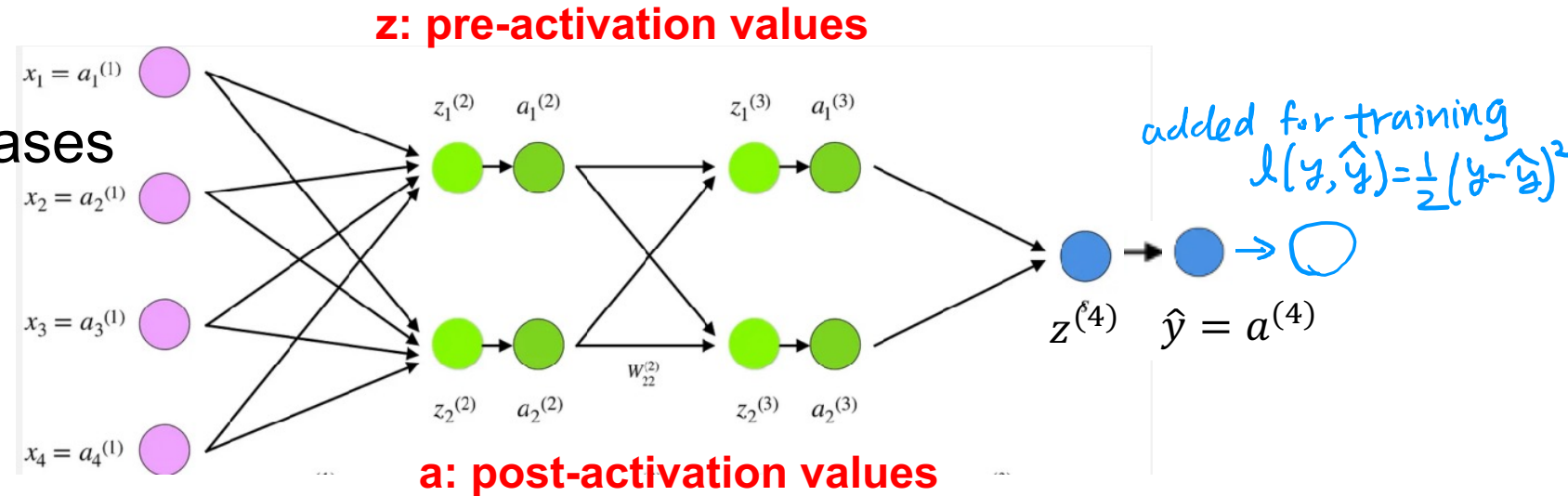
θ = parameters of the network

original images

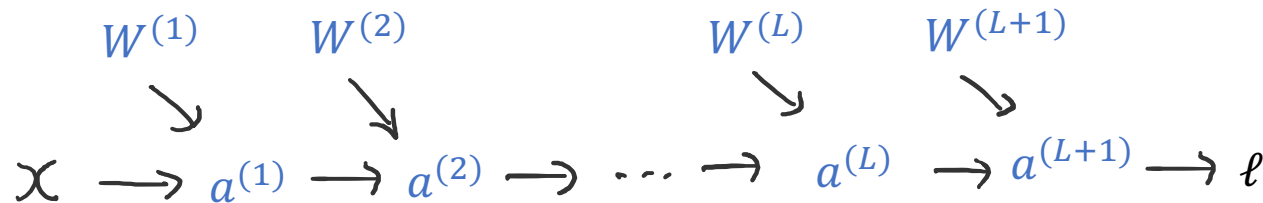


Important case study: loss gradients for MLP learning

- Input $x = (x_1, \dots, x_d)$, label y
- Assume L hidden layers, layer l has n_l nodes
- Assume regression task
- Assume nodes have no biases



- Goal: compute the gradients of $\ell(y, \hat{y})$ w.r.t. all weights
- Treating layers / weights as variables, getting equivalent computation graph:



skipping showing z's for now

This will be the basis for HW3 P3..

Math interlude: Jacobian matrix

Example Given a vector-valued function

$$\vec{f}(x_1, x_2) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 3x_1 + 2x_2 \\ x_1 - x_2 \end{bmatrix}$$

How does each input affect each coordinate of the output?

Sensitivity	x_1	x_2
Affecting f_1	3	2
Affecting f_2	1	-1

Let's organize these sensitivities in a matrix:

$$J = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} \quad \text{the Jacobian of } \mathbf{f}$$

Informally: change in output vector $\vec{f} \approx J \times$ change in input vector x



Carl Jacobi (1804-1851)

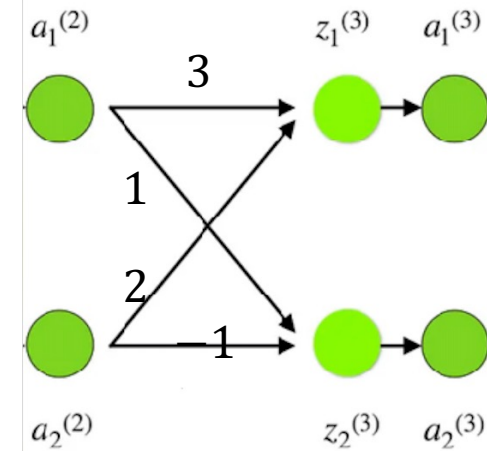
Math interlude: Jacobian matrix

a: post-activation values

z: pre-activation values

Example Suppose neural network layer l is connected with layer $l + 1$ with weight $W^{(l+1)}$. What's $\frac{\partial z^{(l+1)}}{\partial a^{(l)}}$?

Sensitivity	$a_1^{(l)}$	$a_2^{(l)}$
Affecting $z_1^{(l+1)}$	3	2
Affecting $z_2^{(l+1)}$	1	-1



Conclusion: $\frac{\partial z^{(l+1)}}{\partial a^{(l)}} = W^{(l+1)}$

**The weight matrix
connecting layers l and $l+1$**

Math interlude: Jacobian matrix

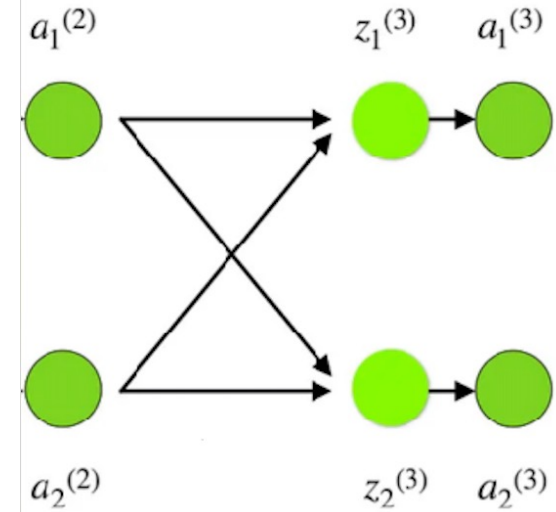
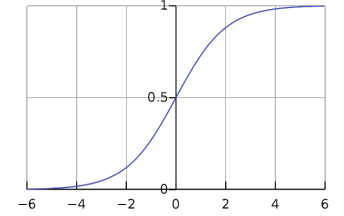
Example Suppose the neural net uses activation function σ .

What's $\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}}$?

Sensitivity	$z_1^{(l+1)}$	$z_2^{(l+1)}$
Affecting $a_1^{(l+1)}$	$\sigma'(z_1^{(l+1)})$	0
Affecting $a_2^{(l+1)}$	0	$\sigma'(z_2^{(l+1)})$

Conclusion: $\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} = \text{diag}(\sigma'(z^{(l+1)}))$

Diagonal matrix with diagonal entries being $\sigma'(z_i^{(l+1)})$'s



Math interlude: generalized chain rule of derivatives

Suppose we have dependency graph $x \xrightarrow{f} y \xrightarrow{g} z$ on vectors x, y, z . Then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

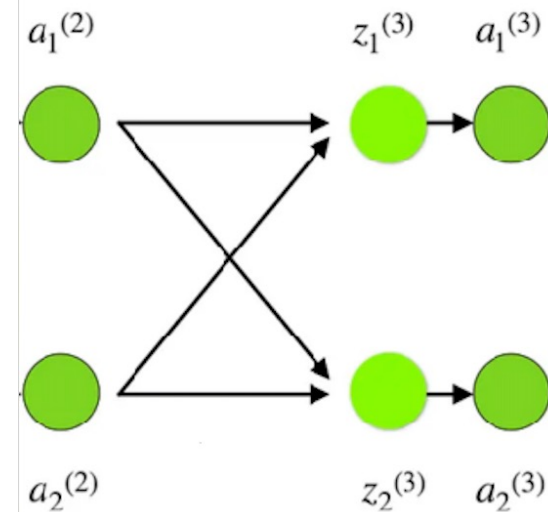
“When you compose functions, multiply their Jacobian matrices in order.”

generalizes the univariate chain rule of derivatives

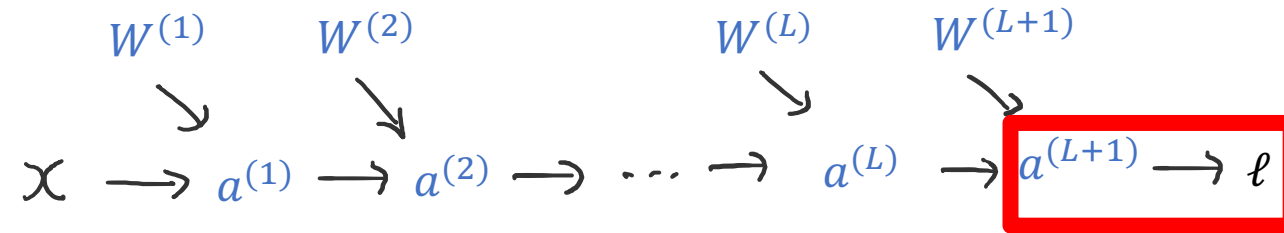
Example Suppose neural net layer l is connected with layer $l + 1$ with weight $W^{(l+1)}$ with activation σ . What is $\frac{\partial a^{(l+1)}}{\partial a^{(l)}}$?

Use chain rule:

$$\frac{\partial a^{(l+1)}}{\partial a^{(l)}} = \underbrace{\text{diag}\left(\sigma'(z^{(l+1)})\right)}_{\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}}} \cdot \underbrace{W^{(l+1)}}_{\frac{\partial z^{(l+1)}}{\partial a^{(l)}}}$$



Loss gradients for MLP learning: last layer



- Find the gradient of ℓ for all variables in *reverse topological order*

- First variable: $a^{(L+1)}$

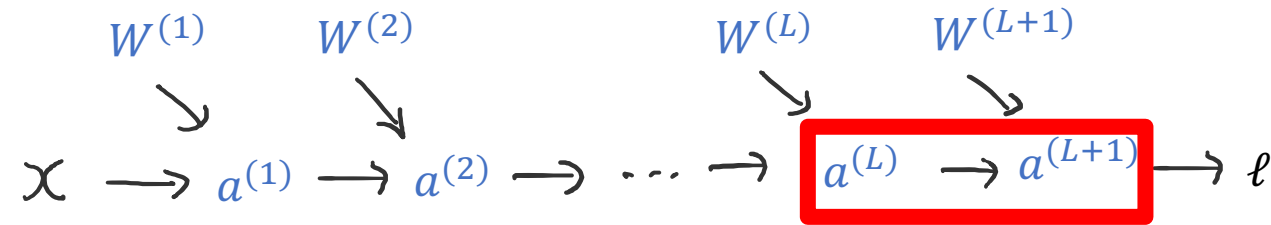
$$\bullet \frac{\partial l}{\partial a^{(L+1)}} = \frac{\partial l}{\partial \hat{y}} = \frac{\partial \left(\frac{1}{2} (\hat{y} - y)^2 \right)}{\partial \hat{y}} = (\hat{y} - y)$$

1 dimensional scalar

added for training
 $\mathcal{L}(y, \hat{y}) = \frac{1}{2} (y - \hat{y})^2$

The diagram shows a circle labeled $\hat{y} = a^{(L+1)}$ with an arrow pointing to another circle, representing the loss function $\mathcal{L}$.

Loss gradients for MLP learning: intermediate layers



- Calculating the gradient of ℓ in *reverse topological order*

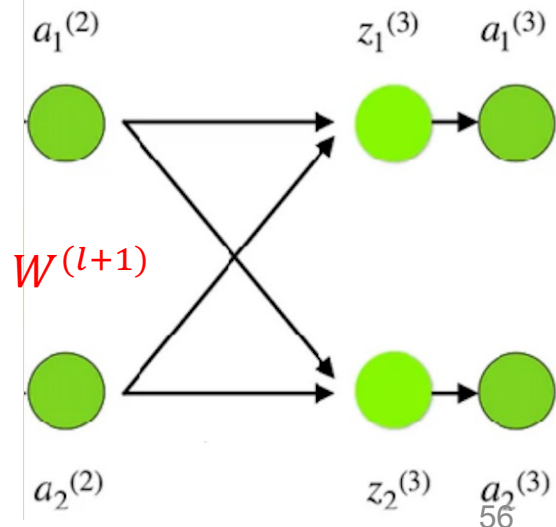
- For each layer l , find $\frac{\partial \ell}{\partial a^{(l)}}$ from $\frac{\partial \ell}{\partial a^{(l+1)}}$:

**generalized chain rule
from calculus**

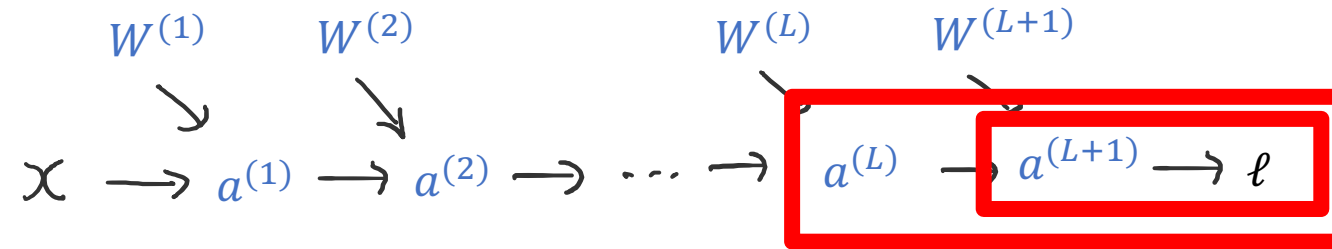
Matrix multiplication

$$\frac{\partial \ell}{\partial a^{(l)}} = \frac{\partial \ell}{\partial a^{(l+1)}} \cdot \frac{\partial a^{(l+1)}}{\partial a^{(l)}} \text{diag}(\sigma'(z^{(l+1)})) \cdot W^{(l+1)}$$

$1 \times n_l$ $1 \times n_{l+1}$ $n_{l+1} \times n_l$

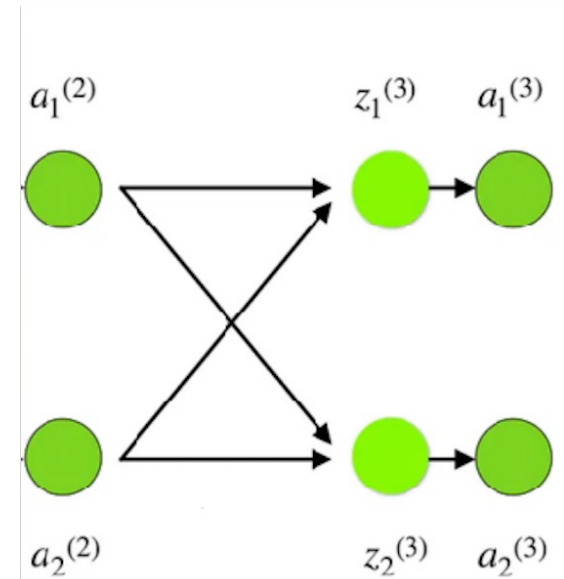


Loss gradients for MLP learning: intermediate layers

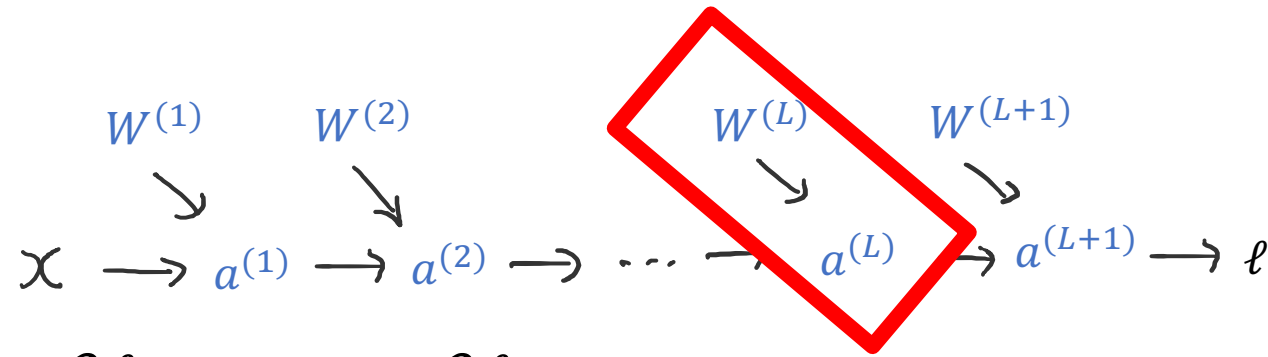


Summary for each layer l :

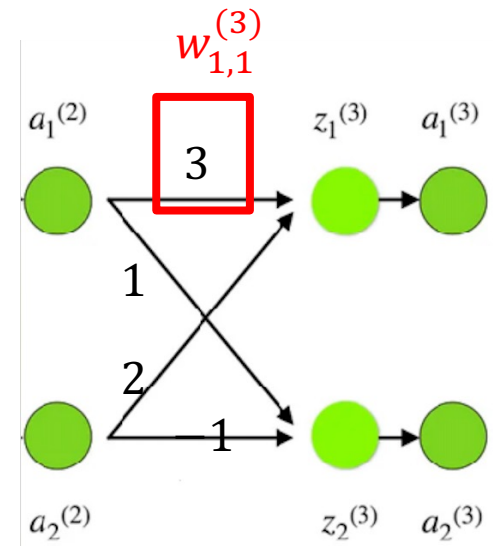
$$\frac{\partial \ell}{\partial a^{(l)}} = \frac{\partial \ell}{\partial a^{(l+1)}} \cdot \text{diag}(\sigma'(z^{(l+1)})) \cdot W^{(l+1)}$$



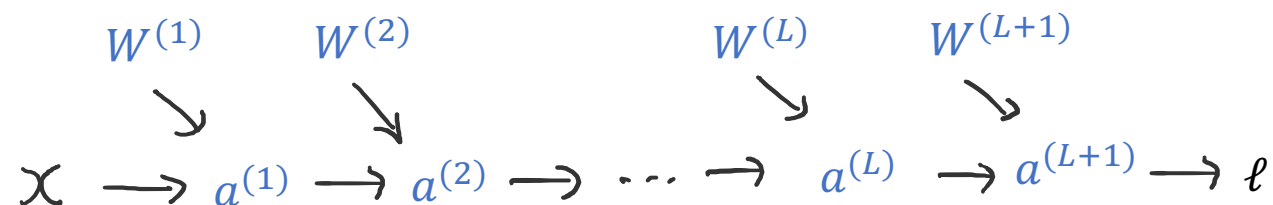
Loss gradients for MLP learning: weight for each layer



- Finally, let's find $\frac{\partial \ell}{\partial W^{(l)}}$ from $\frac{\partial \ell}{\partial a^{(l)}}$
- Specifically, let's find $\frac{\partial \ell}{\partial w_{j,i}^{(l)}}$ for every j, i
- Example: find $\frac{\partial \ell}{\partial w_{1,1}^{(3)}}$



Loss gradients for MLP learning: weight for each layer

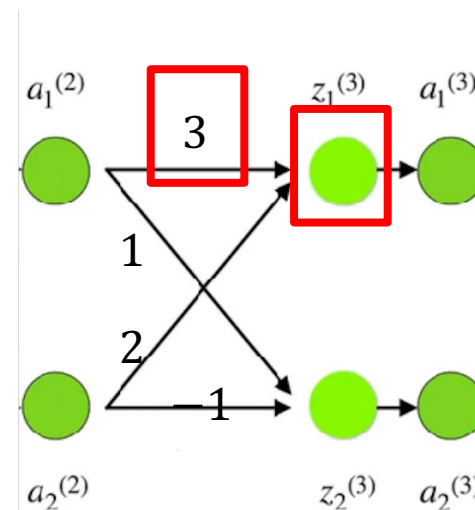


- Observe: $w_{ji}^{(l)}$ only directly influences $z_j^{(l)}$ and no other nodes
- Thus,

$$\frac{\partial \ell}{\partial w_{ji}^{(l)}} = \frac{\partial \ell}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}}$$

Diagram illustrating the chain rule for the gradient of the loss ℓ with respect to the weight $w_{ji}^{(l)}$. The expression is shown as a product of two terms, each enclosed in a red box. Red arrows point from the boxes to their corresponding terms in the chain rule:

- The first term, $\frac{\partial \ell}{\partial z_j^{(l)}}$, is associated with $\frac{\partial \ell}{\partial a_j^{(l)}} \cdot \sigma'(z_j^{(l)})$.
- The second term, $\frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}}$, is associated with $a_i^{(l-1)}$.



Math interlude: outer product of vectors

Example Suppose have two vectors $a = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $b = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$, what if we multiply every entry of a with every entry of b ?

This gives

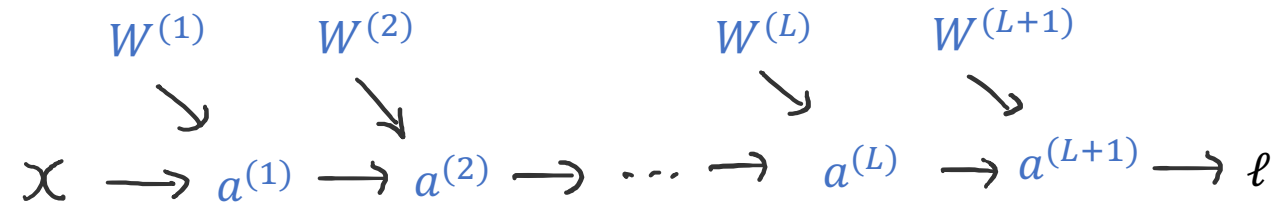
$$\begin{bmatrix} 1 \times 3 & 1 \times 4 \\ 2 \times 3 & 2 \times 4 \end{bmatrix} = ab^\top$$

$2 \times 1 \quad 1 \times 2$

The matrix is called *outer product*

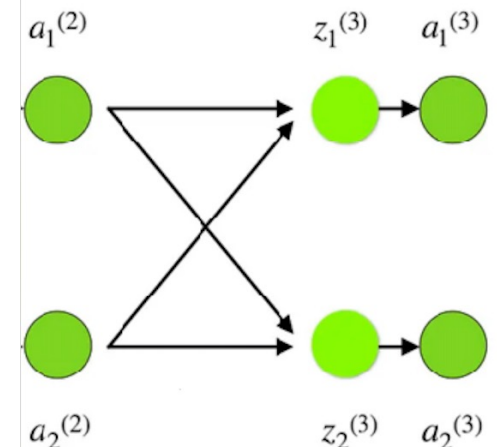
Property for every j, i , $(ab^\top)_{j,i} = a_j b_i$

Loss gradients for MLP learning: weight for each layer



Summary we can write $\frac{\partial \ell}{\partial W^{(l)}}$ as an outer product:

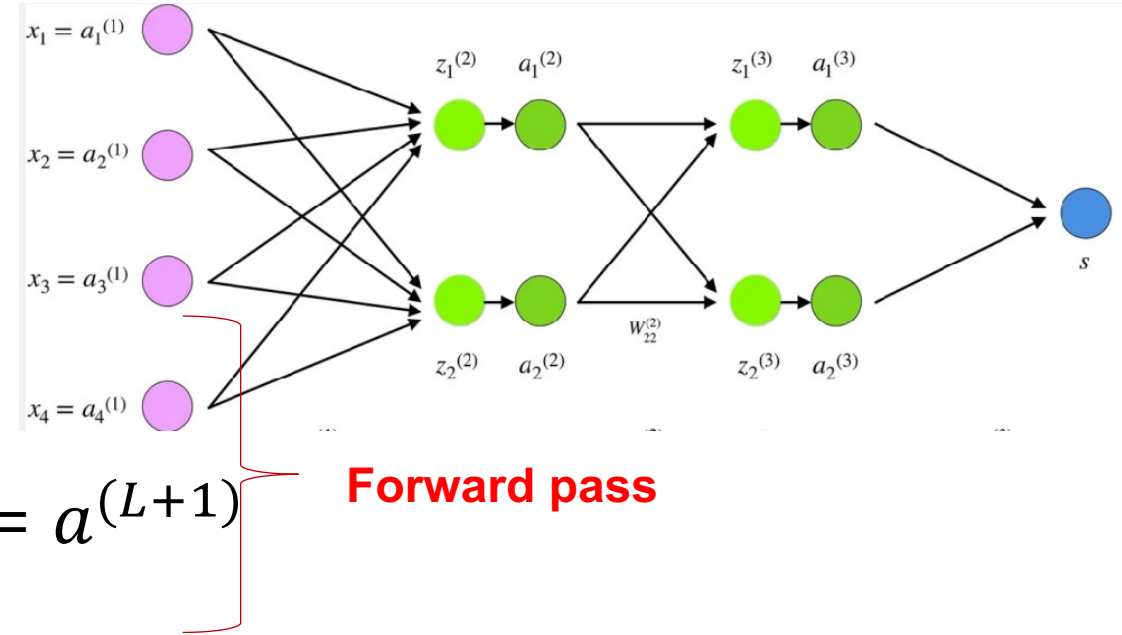
$$\frac{\partial \ell}{\partial W^{(l)}} = \underbrace{\left(\frac{\partial \ell}{\partial z^{(l)}} \right)^{\top}}_{\mathbf{1} \times \mathbf{n}_{l-1}} \underbrace{a^{(l-1)}}_{\mathbf{n}_l \times \mathbf{1}} \quad \mathbf{n}_l \times \mathbf{n}_{l-1}$$



Backprop for MLP learning: Summary

To calculate $\frac{\partial \ell}{\partial W}$ given example (x, y)

- For $l = 1, \dots, L$:
 - Compute $a^{(l+1)}$ based on $a^{(l)}$
- Compute loss $\ell = \frac{1}{2} (\hat{y} - y)^2$, where $\hat{y} = a^{(L+1)}$



- Last layer: $\frac{\partial \ell}{\partial a^{(L+1)}} = \frac{\partial \ell}{\partial \hat{y}} = (\hat{y} - y)$

a: post-activation

- For $l = L, \dots, 1$:

- $\frac{\partial \ell}{\partial z^{(l+1)}} = \frac{\partial \ell}{\partial a^{(l+1)}} \cdot \text{diag} \left(\sigma' \left(z^{(l+1)} \right) \right), \frac{\partial \ell}{\partial a^{(l)}} = \frac{\partial \ell}{\partial z^{(l+1)}} \cdot W^{(l+1)}$

- $\frac{\partial \ell}{\partial W^{(l+1)}} = \left(\frac{\partial \ell}{\partial z^{(l+1)}} \right)^\top a^{(l)}$

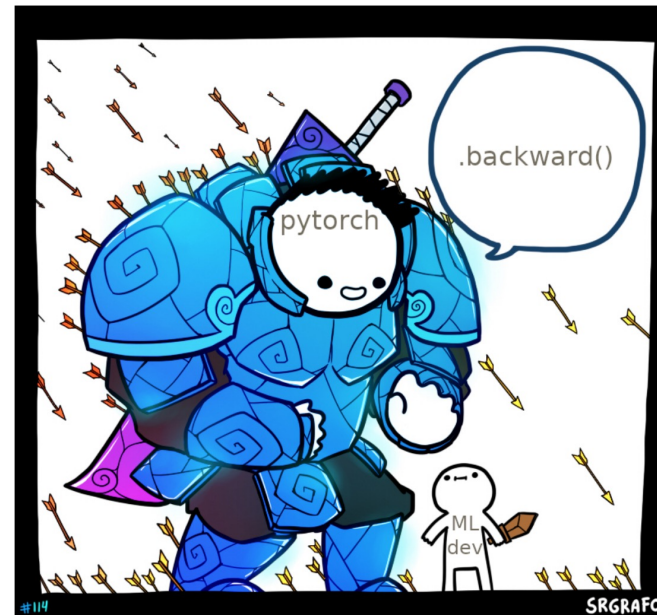
Backward pass

Backprop for MLP learning: resources

- <https://www.donnybertucci.com/project/backprop-explainer>
- [Stanford CS 231N lecture by Andrej Karpathy](#)

Backpropagation

- In practice, neural network implementations use *auto differentiation* to compute the gradients 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.

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

Backpropagation in pytorch

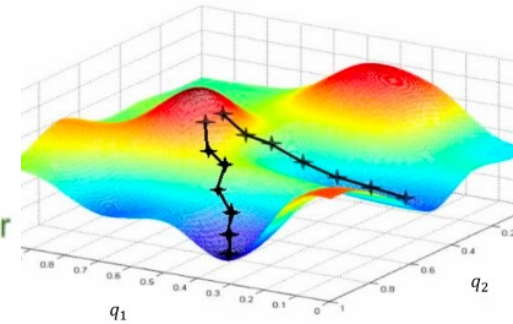
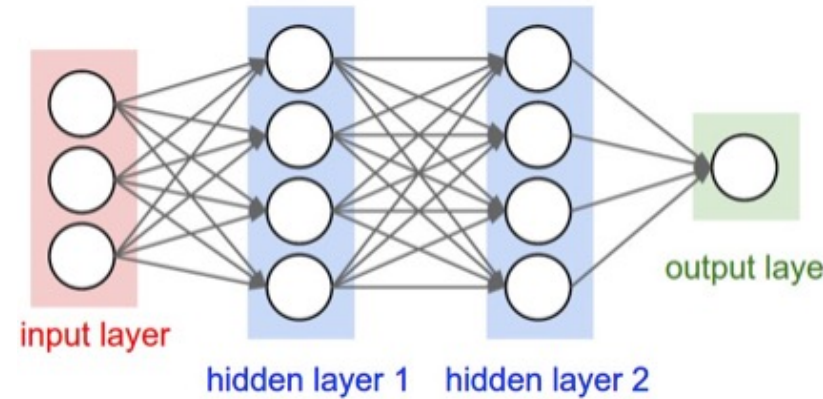
- Jupyter notebook [here](#)
- ‘autodifferentiation’ section
- In HW3, you are asked to implement a “baby pytorch” and train a neural network with it

Recap: neural network optimization

- Mini-batch gradient descent

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

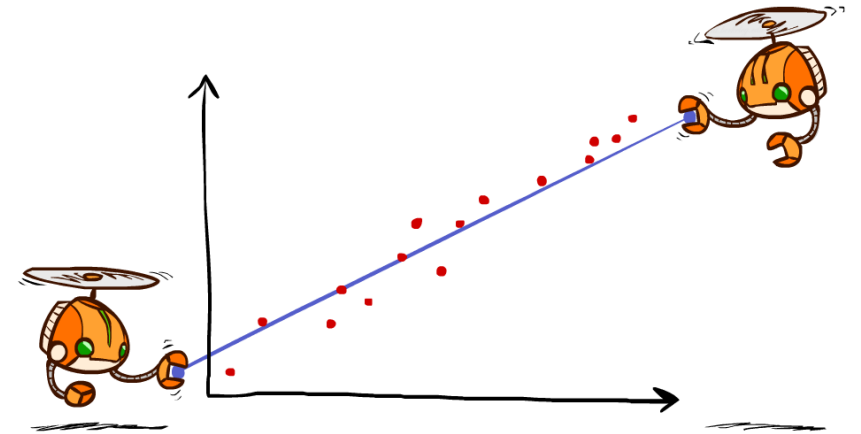
θ = parameters of the network



- How to compute the gradient?
 - Form a computational graph that computes loss in terms of data and neural network weights
 - Backpropagation: compute the derivative of the loss with respect to neural network weights in a backward fashion
 - Key tool: chain rule of derivatives (Jacobians)
 - Modern tools such as PyTorch provide autodifferentiation

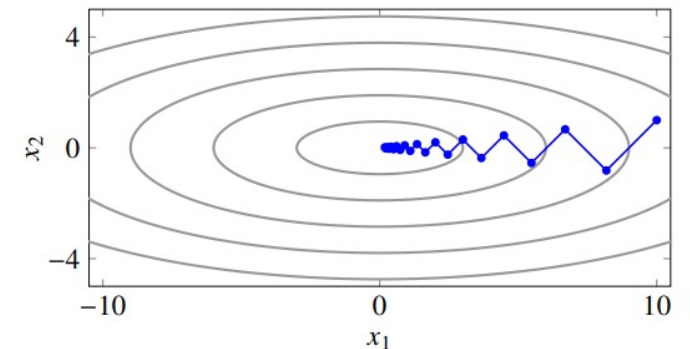
Recall: Linear regression

- Find w that minimize MSE
- $R(w) = \frac{1}{7} \cdot (w_1 \cdot 1 + w_2 \cdot 1 - 1)^2 + \dots$
 $+ \frac{1}{7} \cdot (w_1 \cdot 4 + w_2 \cdot 1 - 1)^2$



- $\nabla R(w) = \left(\frac{\partial R}{\partial w_1}, \frac{\partial R}{\partial w_2} \right) = \left(\frac{2}{7} (78w_1 + 46w_2 - 38), \frac{2}{7} (46w_1 + 33w_2 - 13) \right)$

- Find optimal w by gradient descent: repeatedly,
$$w \leftarrow w - \alpha \nabla R(w)$$



Linear regression in pytorch

```
1 w = torch.zeros(2, requires_grad = True)
2 stepsize = 0.1 # a small one to help visualize
3 w_optimizer = torch.optim.SGD([w], lr = stepsize)
4 n_iters = 10
5 # we'll use pytorch's sgd for z:
6 for i in range(n_iters):
7     w_optimizer.zero_grad() # the optimizer handles this now
8     # creating function R(w)
9     risk = ((X @ w - y) ** 2 / 2).mean()
10    risk.backward()
11    print(risk.item())
12    # now do the update. torch.no_grad() is unnecessary
13    # because .step() invokes it internally.
14    w_optimizer.step() # thanks, pytorch
```

takes control in updating the iterate w

Backprop to compute the gradient of risk
wrt all quantities involved (including w)

4.102662563323975
3.3636224269866943
2.7586870193481445
2.2633352279663086
1.8575656414031982
1.525056004524231
1.2524806261062622
1.0289573669433594
0.8455942273139954
0.6951242089271545

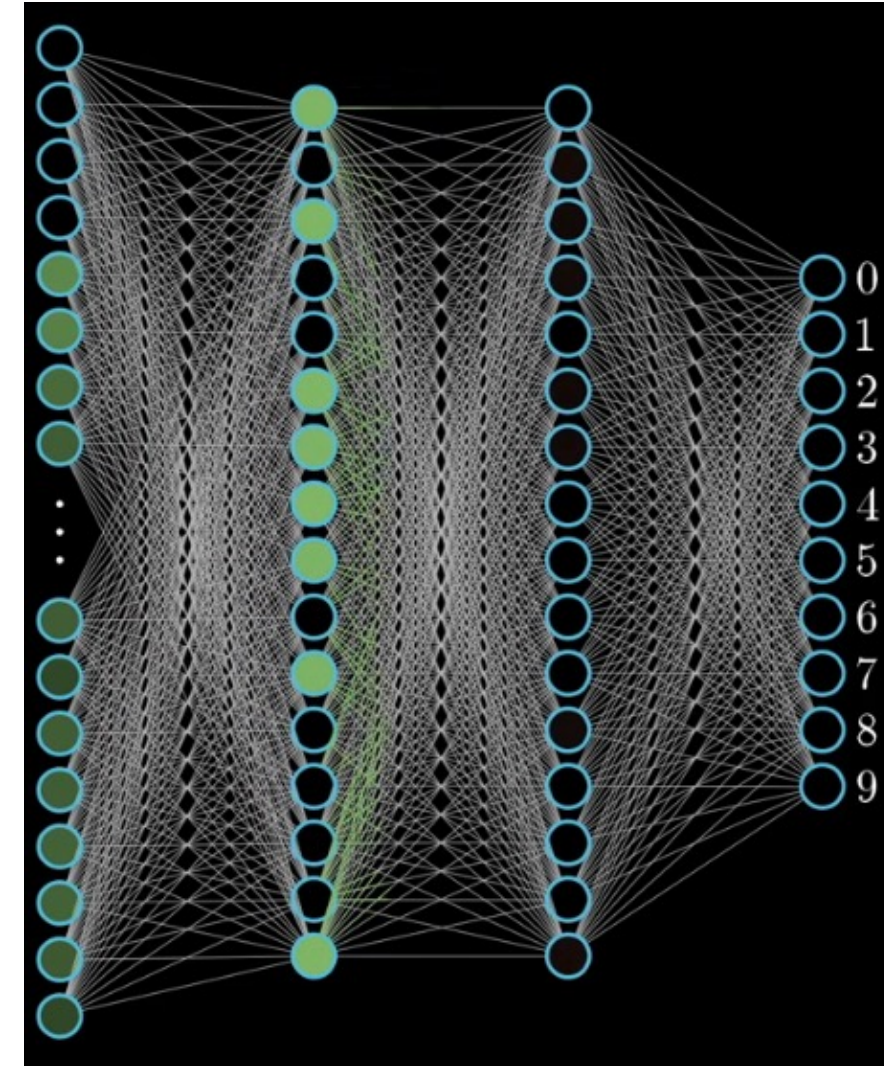
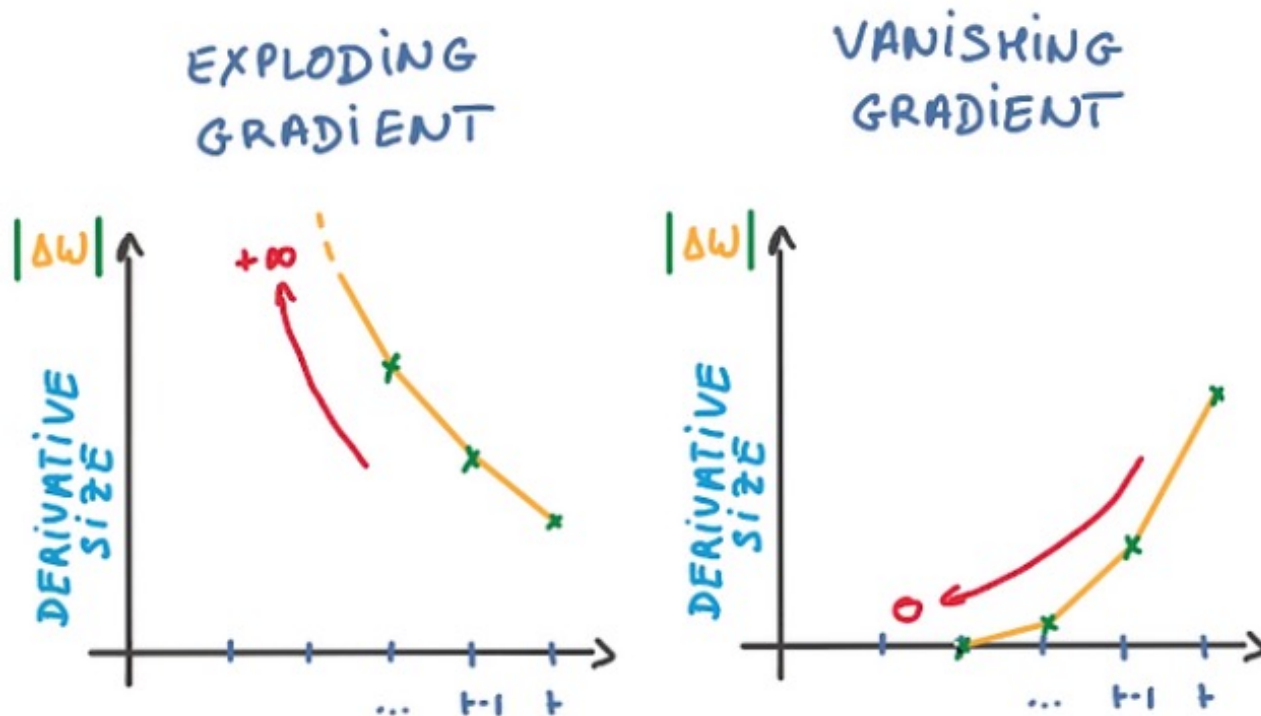
Pytorch: a Jupyter notebook tutorial

- Jupyter notebook [here](https://mjt.cs.illinois.edu/ml/) (Credit: Matus Telgarsky <https://mjt.cs.illinois.edu/ml/>)
- In the interest of time, let's jump to 'Digit Experiment'
- This tries two neural network architectures for digit classification
 - Multi-layer Perceptron
 - Convolutional neural network (What's this? We'll see)



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 x in the domain of F , $|F(x) - \hat{F}(x)| < \epsilon$.*

- 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

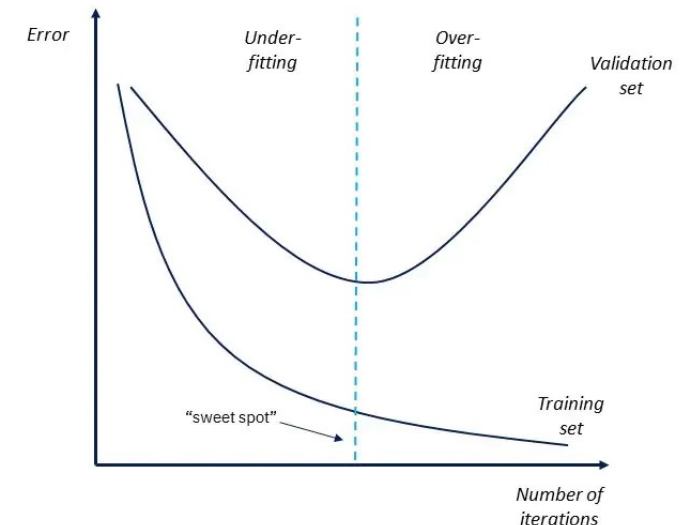
Regularization

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:

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

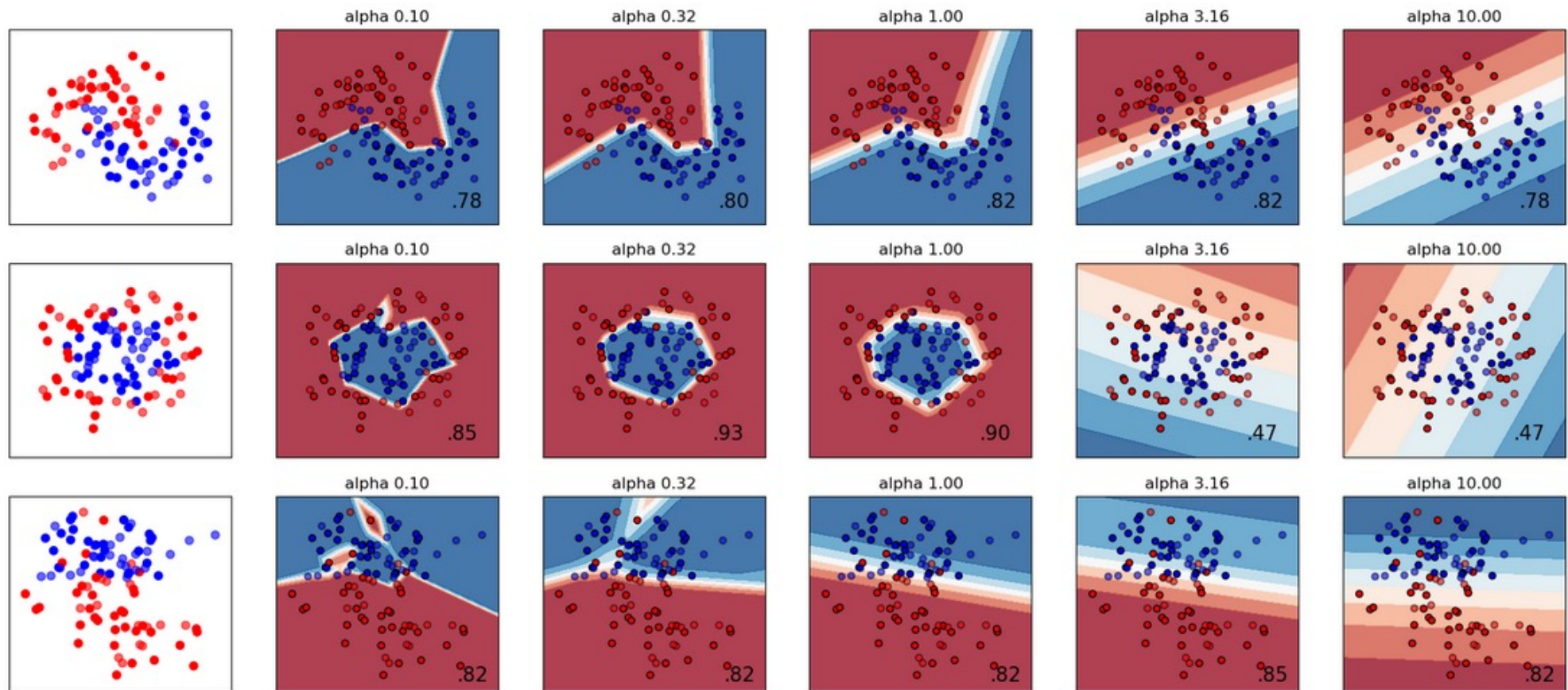
Learning Rate
(how big of a step)

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

L2 Regularization: Weight Decay

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



L1 Regularization

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

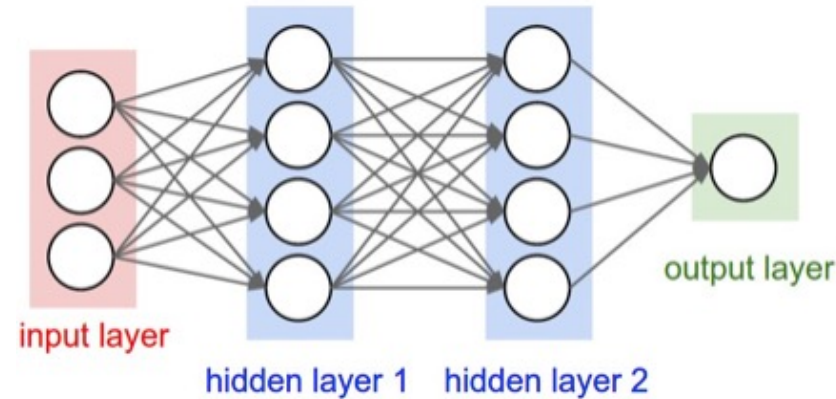
Gradient given by,

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

- 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)

Weight Tying / Sharing

- Hard constraints force sets of parameters to be equal
 - Known as *parameter sharing*
 - Only subset of unique parameters needs to be stored in memory

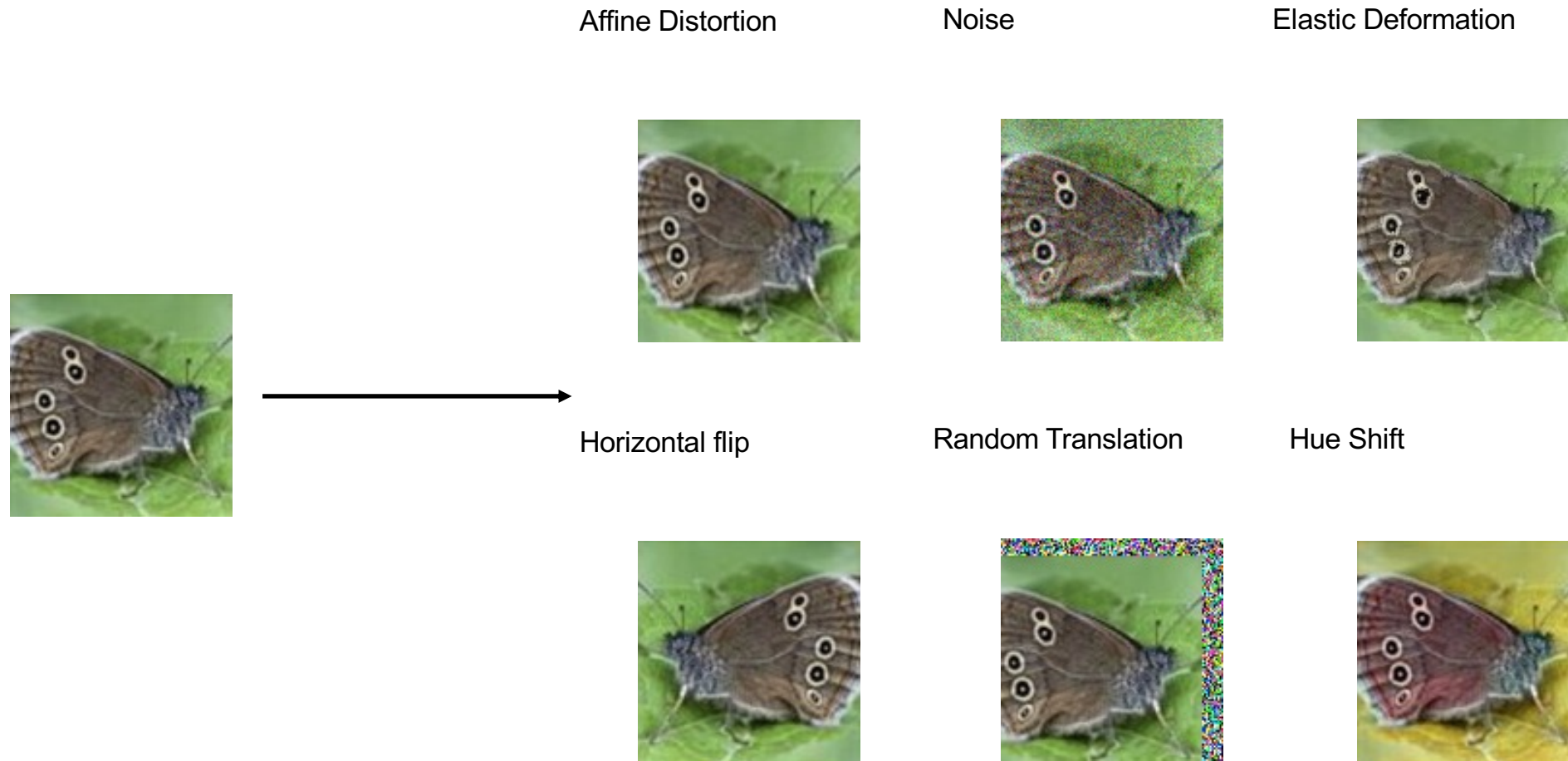


- Example: convolutional neural network (we will discuss in detail)

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)
- Example application: object recognition
 - Translation
 - Scaling
 - Rotation
 - ...

Dataset Augmentation



Dataset Augmentation

- Need to avoid transformations that change class
- For example:
 - Mirroring “b” to produce “d”
 - Rotation turns “6” into “9”

Learning Curves – Early Stopping

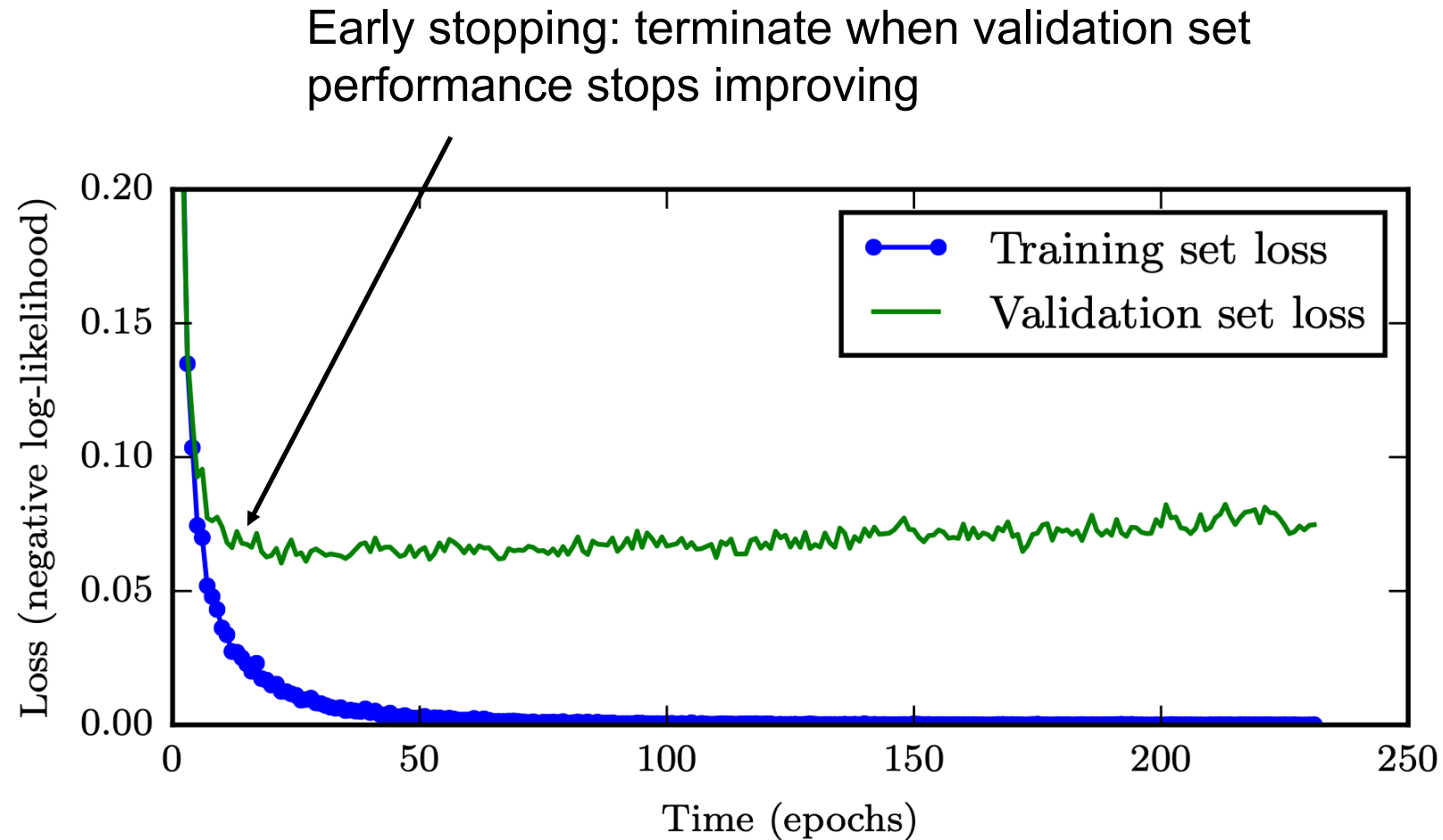


Figure 7.3

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 θ_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 θ 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 θ^* , best number of training steps is i^* .

Validation error going up

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, L1+L2 (elastic net) regularization
- **Data Augmentation** Synthetically expand training data by applying random transformations
- **Early stopping** Just as it sounds...stop the training if the validation error does not keep decreasing...simple-but-effective
- **Dropout** Each iteration randomly selects a small number of edges to temporarily exclude from the network (weights=0)

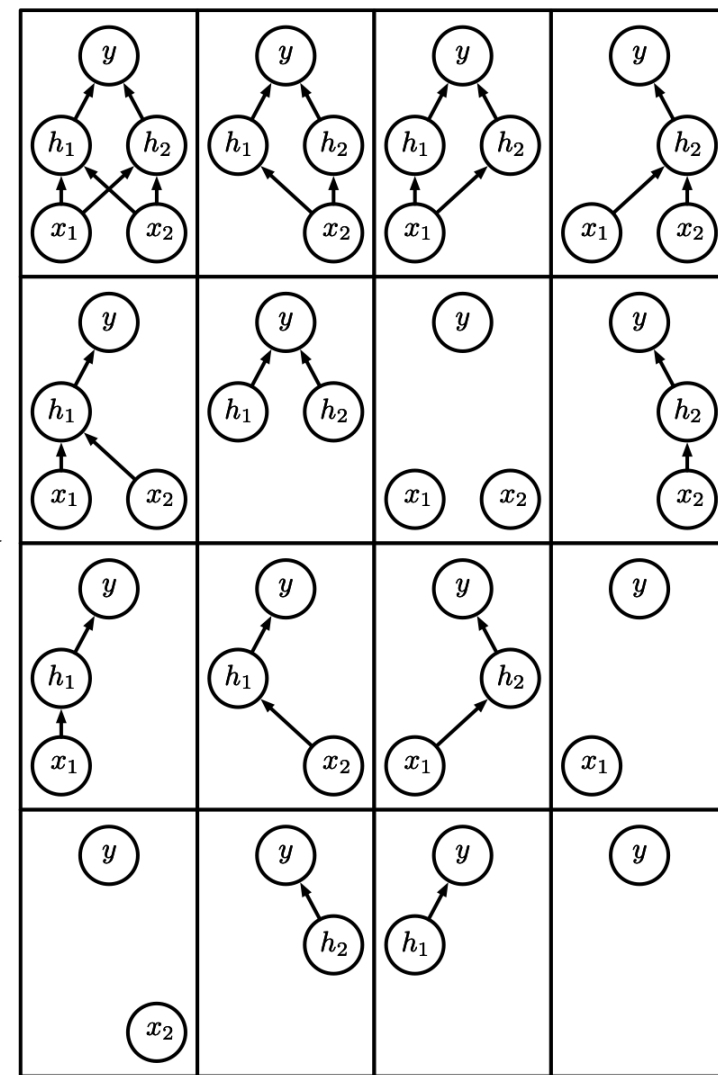
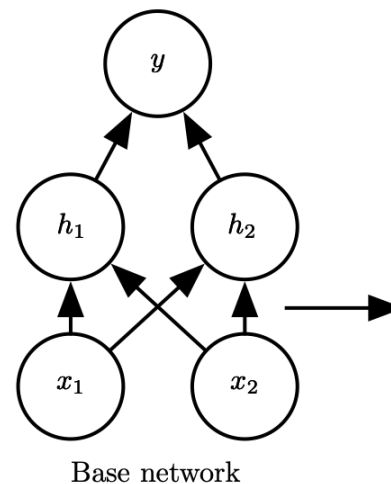
Dropout (Srivastava et al, 2014)

Each time we load a minibatch to perform updates:

- randomly remove set of nodes & associated edges
- do backprop on the remaining network

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

Figure 7.6



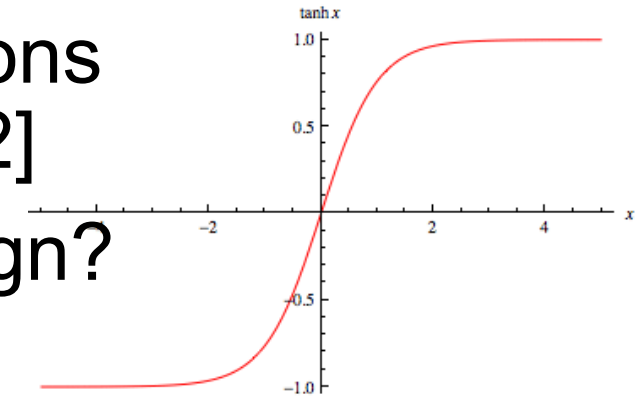
Ensemble of subnetworks

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 down 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 pre-activations of the nodes are 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 pre-activations!
- Recall minibatch SGD
 - Computes gradients for $k = 32$ (for example) data points, then updates the weights
 - Can we ensure that within a batch, for a layer, most of the inputs are “standardized”?



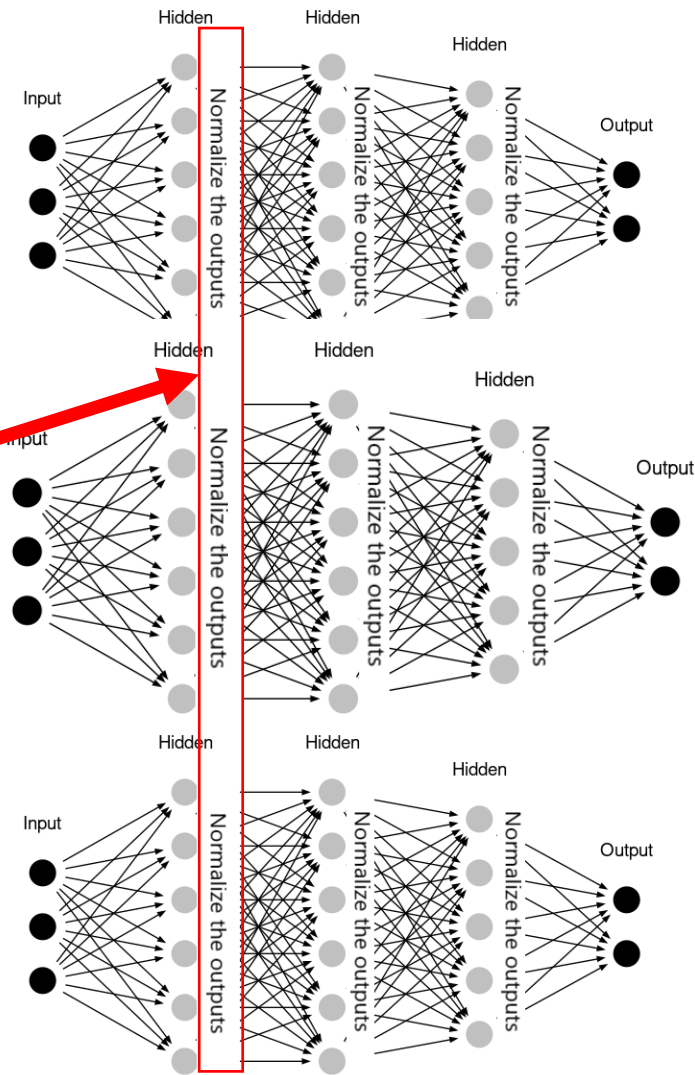
Batch normalization layer

- 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 pre-activations

Batch normalization layer

- Example: neural network making predictions on a batch of $M=3$ examples with batch normalization on 4 neurons

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

Parameters to be learned: γ, β

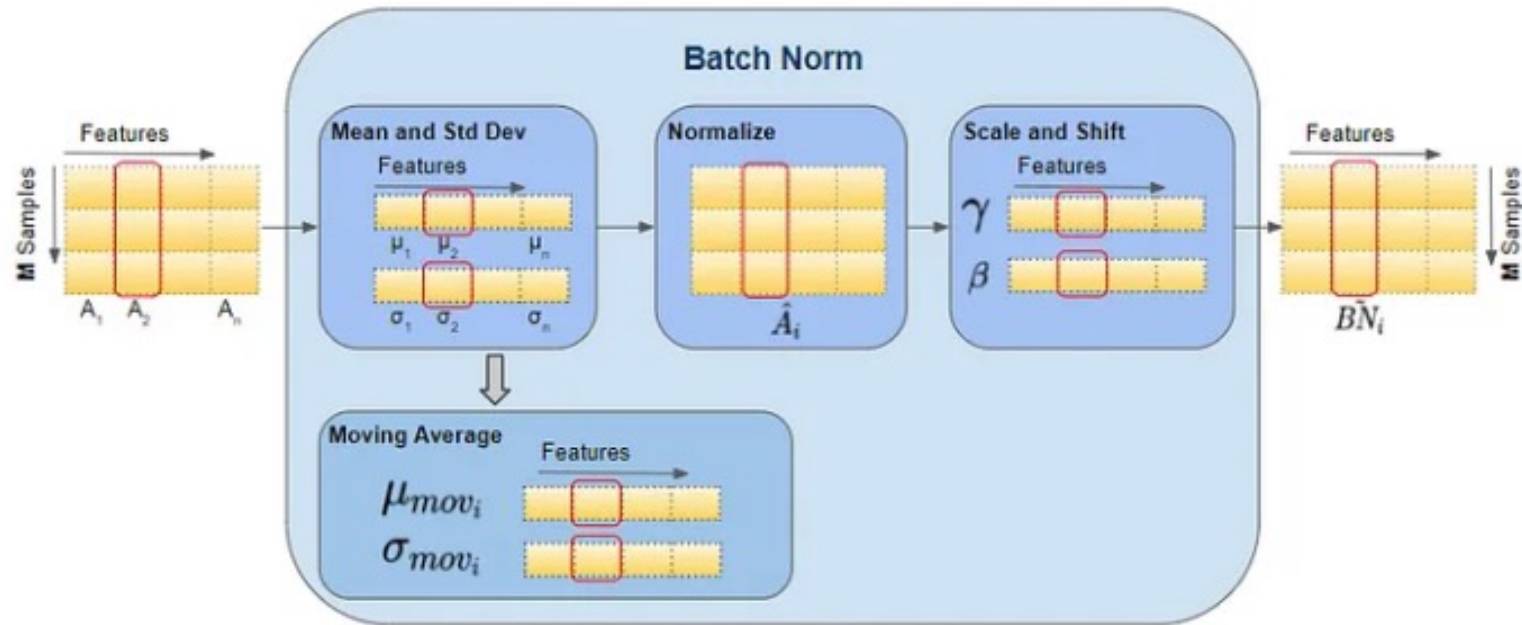
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

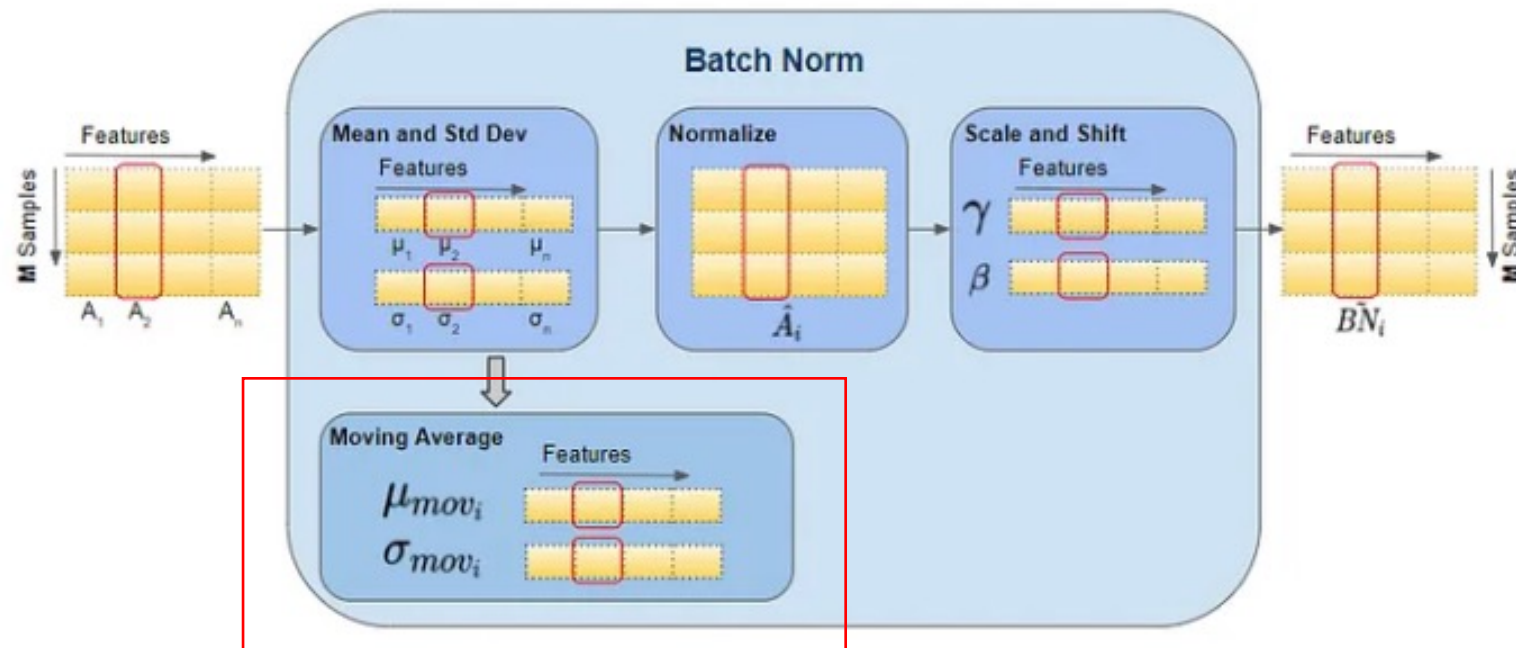
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$



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

Batch normalization

- One twist for the test time
 - It is now natural to use this network to predict batches of M examples
 - What if we want to use the network to predict a *single* test example x ?
 - Use (averaged) μ_B and σ_B^2 computed in training time



Batch normalization

- 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 Transformers / LLMs

Recap: learning neural networks

PyTorch training loop

```
1 # Pass the data through the model for a number of epochs (e.g. 100)
2 for epoch in range(epochs):
3
4     # Put model in training mode (this is the default state of a model)
5     model.train()
6
7     # 1. Forward pass on train data using the forward() method inside
8     y_pred = model(X_train)
9
10    # 2. Calculate the loss (how different are the model's predictions to the true values)
11    loss = loss_fn(y_pred, y_true)
12
13    # 3. Zero the gradients of the optimizer (they accumulate by default)
14    optimizer.zero_grad()
15
16    # 4. Perform backpropagation on the loss
17    loss.backward()
18
19    # 5. Progress/step the optimizer (gradient descent)
20    optimizer.step()
```

Note: all of this can be turned into a function

Pass the data through the model for a number of **epochs** (e.g. 100 for 100 passes of the data)

Pass the data through the model, this will perform the **forward()** method located within the model object

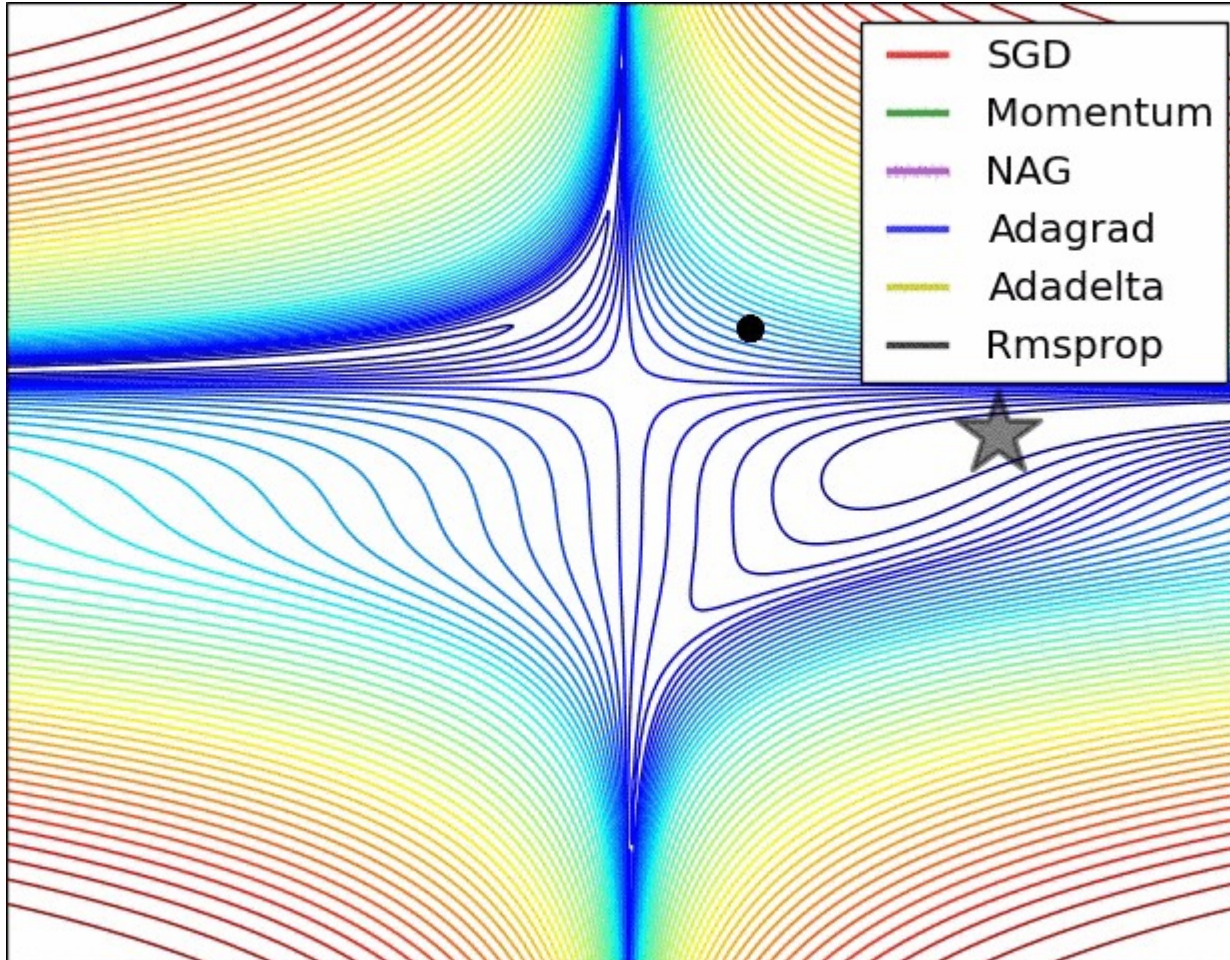
Calculate the loss value (how wrong the model's predictions are)

Zero the optimizer gradients (they accumulate every epoch, zero them to start fresh each forward pass)

Perform **backpropagation** on the loss function (compute the gradient of every parameter with `requires_grad=True`)

Step the optimizer to update the model's parameters with respect to the gradients calculated by `loss.backward()`

The importance of stepsize in NN optimization



Gradient-based optimization:

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

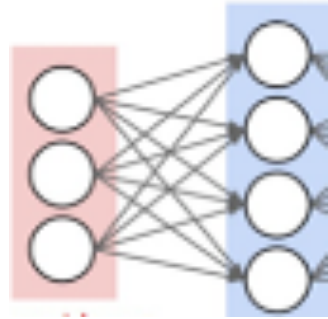
Stochastic Gradient Descent (SGD) sets $\eta_t =$ some constant.

Advanced optimizers (adaptive gradient methods) have different ways to set η_t !

- **Adam** with default step size typically works well.
- In PyTorch:
`optimizer = optim.Adam(model.parameters(), lr=1e-3)`

Weight initialization

- Never start from 0 -- this will induce undesirable symmetries in weights
- Xavier initialization (2010): $\text{weight} \leftarrow (\text{unit gaussian}) / \sqrt{\text{fan_in}}$
 - **fan_in**: #nodes in the previous layer
 - ensures that the output node values have variance ≈ 1
 - works for $\tanh()$ activation but not for ReLU
- Kaiming Initialization (2015):
 - ReLU “kills” half of the activations
 - for ReLU, do $(\text{unit gaussian}) / \sqrt{\text{fan_in}/2}$



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 :

```
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)

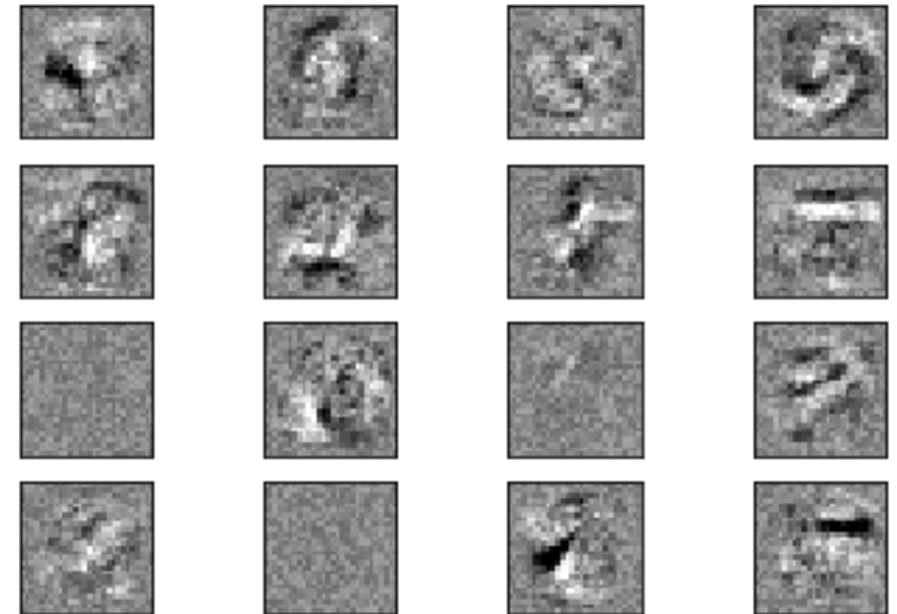
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
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

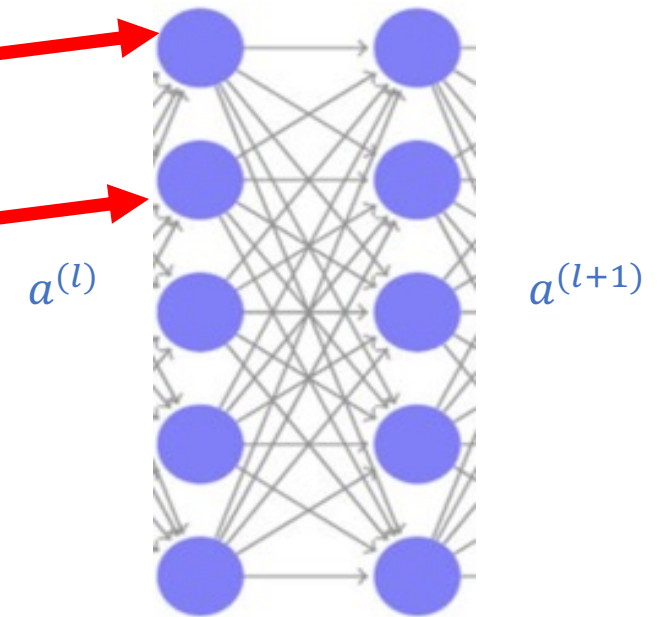
NNs for images

- MLPs with fully connected layers can learn (pattern, location) combinations in images
 - The learned patterns do not generalize to different spatial locations

Neuron 1: detects deer around (124, 236)

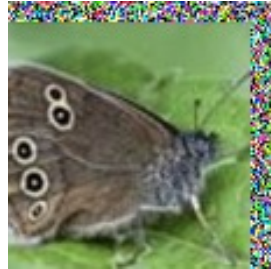
Neuron 2: detects flowers around (34, 301)

....



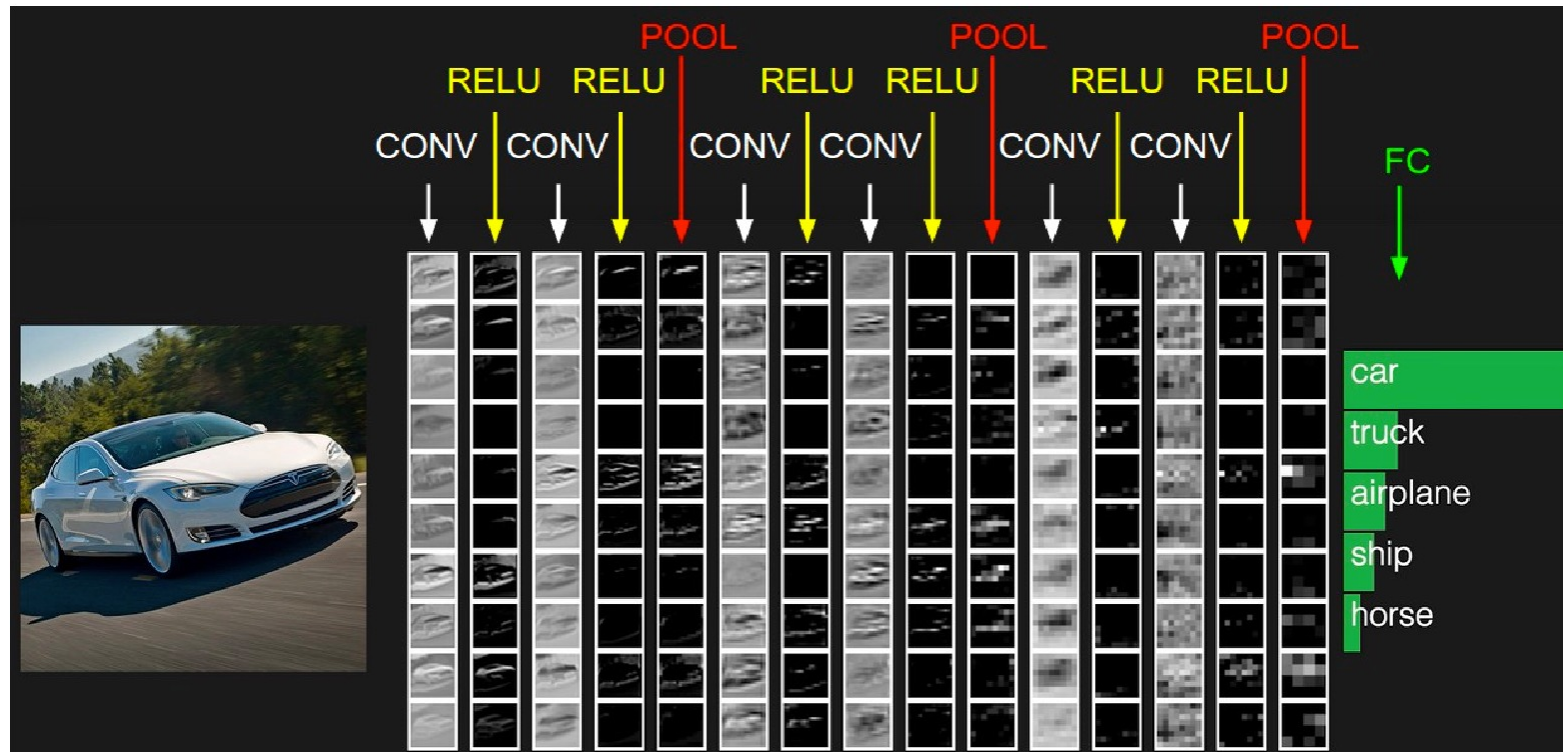
NNs for images

- Can we learn a group of neurons that detect a certain pattern (e.g. *existence* of a face) at *all locations*?
 - low level: edge of some orientation, a patch of some color
 - high level: face, shape of a wheel
- Encodes inductive bias
 - Image classification: the label of an image should not change if the image is cropped



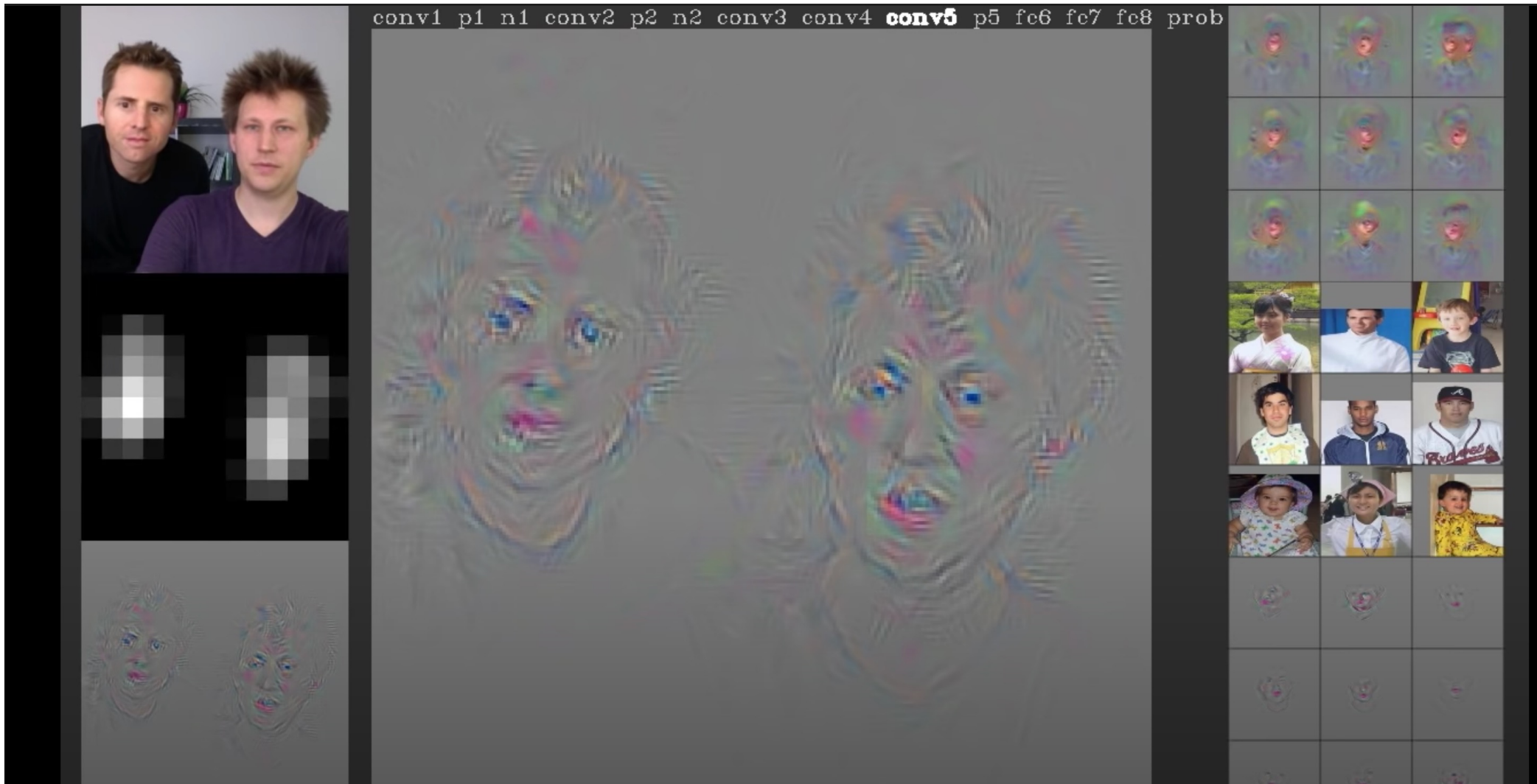
Convolutional neural networks (CNN)

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



Seeing what happens in CNN

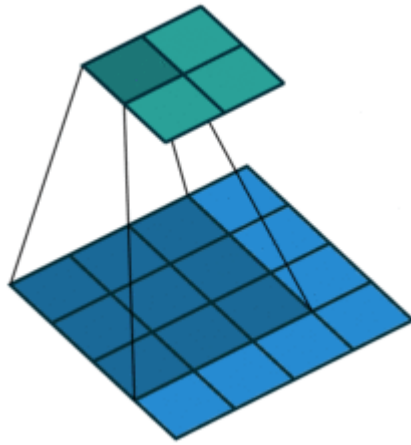
- <https://yosinski.com/deepvis#toolbox>



Convolution for single-channel images

One convolutional 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.



0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	1	0	0	0	0
1	1	0	0	0	0	0

$\times$

1	0	1
0	1	0
1	0	1

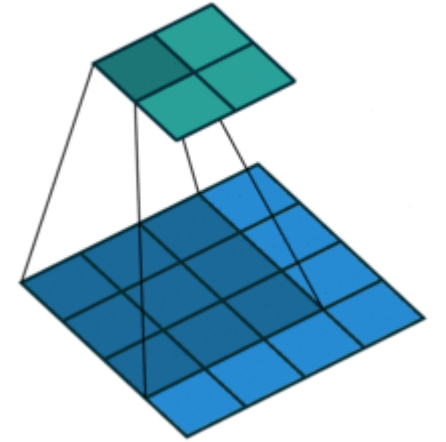
=

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

Convolution: signal processing perspective

The convolution of filter f on image I is:

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



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...

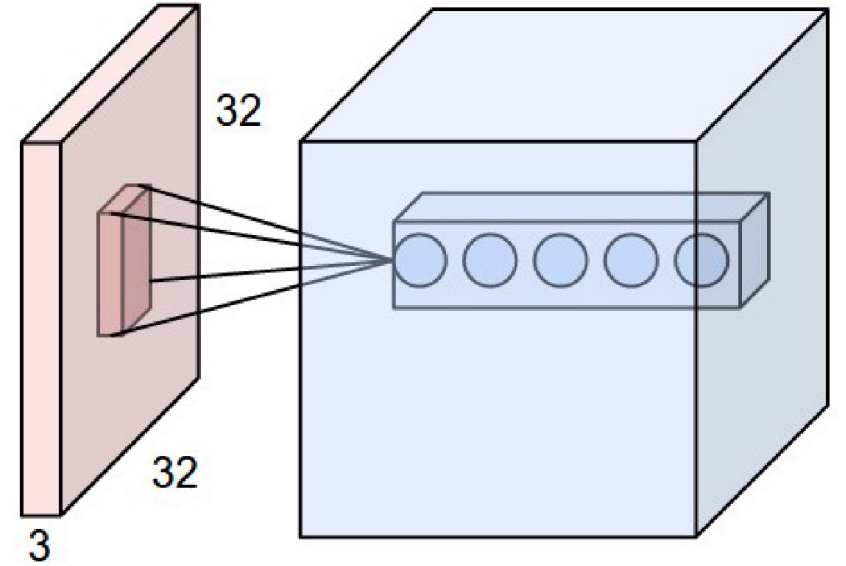
Convolutional layer for multi-channel images

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

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

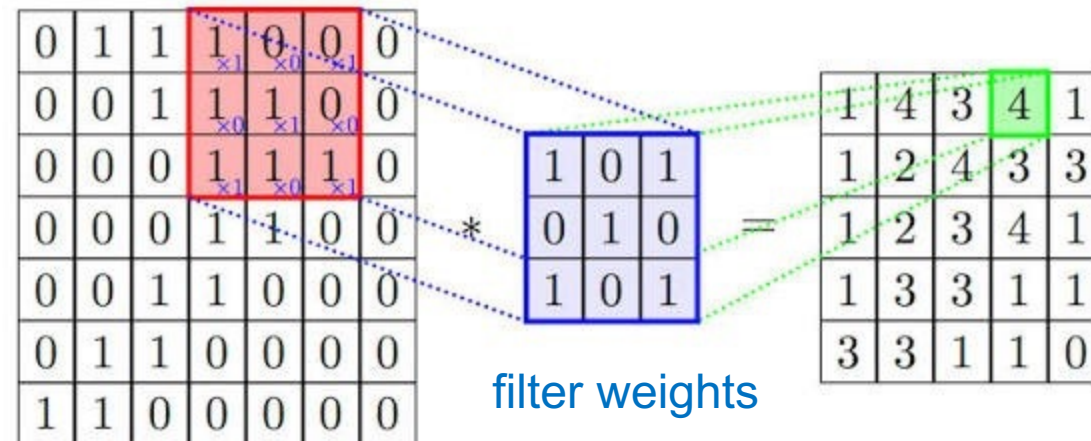
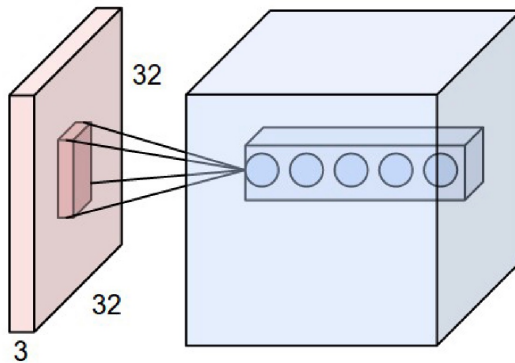
A convolutional filter on such image is of shape $F \times F \times c$

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



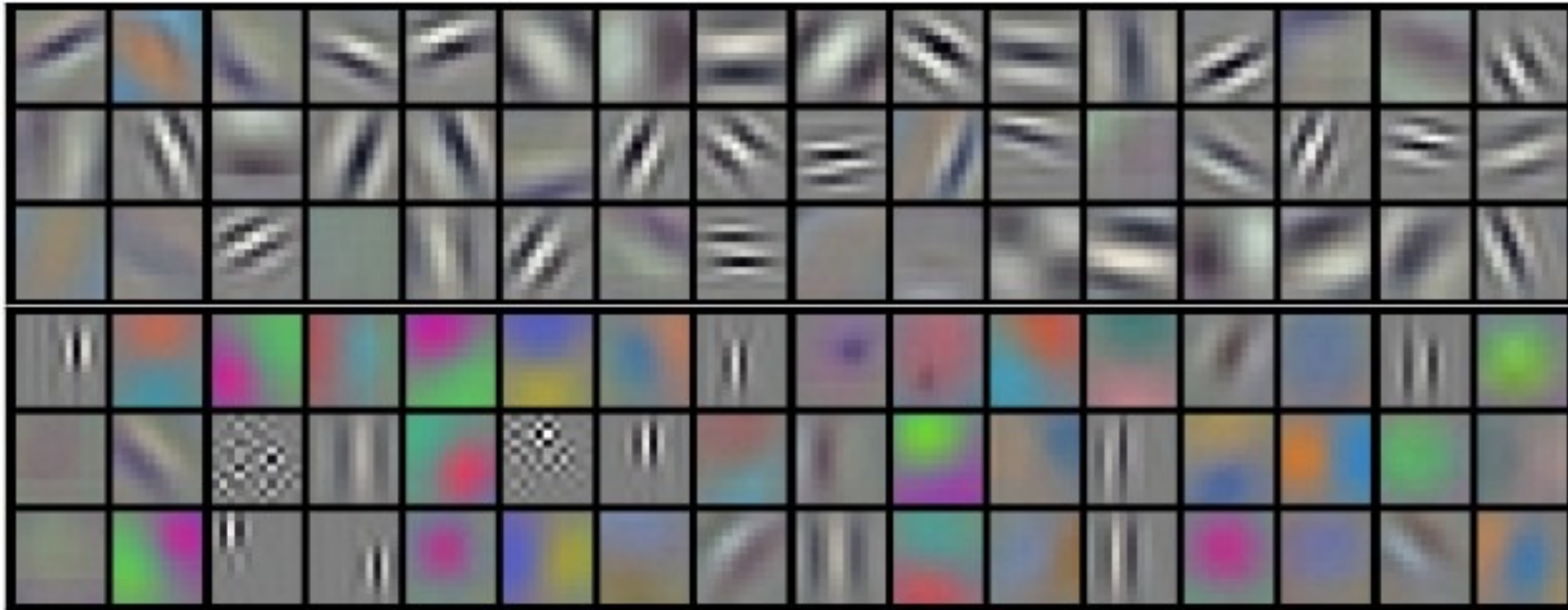
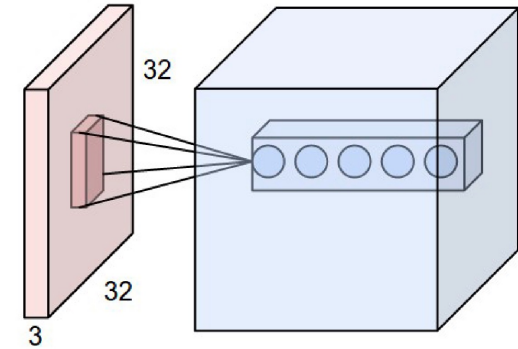
Convolutional layer: multichannel images

- Consider one filter with weights $\{w_{i,j,k}\}$ with $5 \times 5 \times 3$
 - Imagine a sliding 3D window, but only sliding along the two spatial dimensions
 - **Convolution:** For every 5×5 region of the image, perform inner product (= element wise product, then sum them all)
- 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

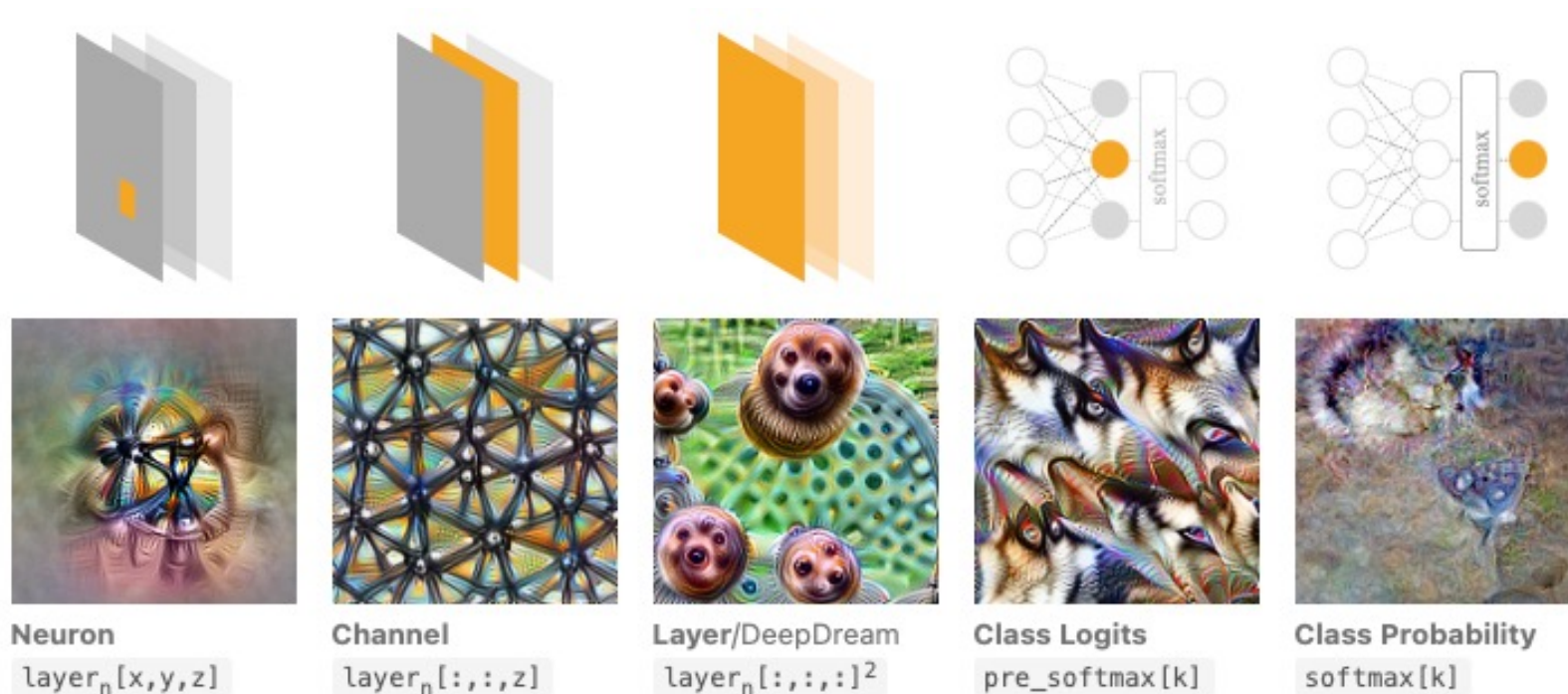
- Example filters learned at the first conv layer
- Each filter has size 11x11x3
- Many filters look like edge detectors



Convolutional layers act as feature extractors

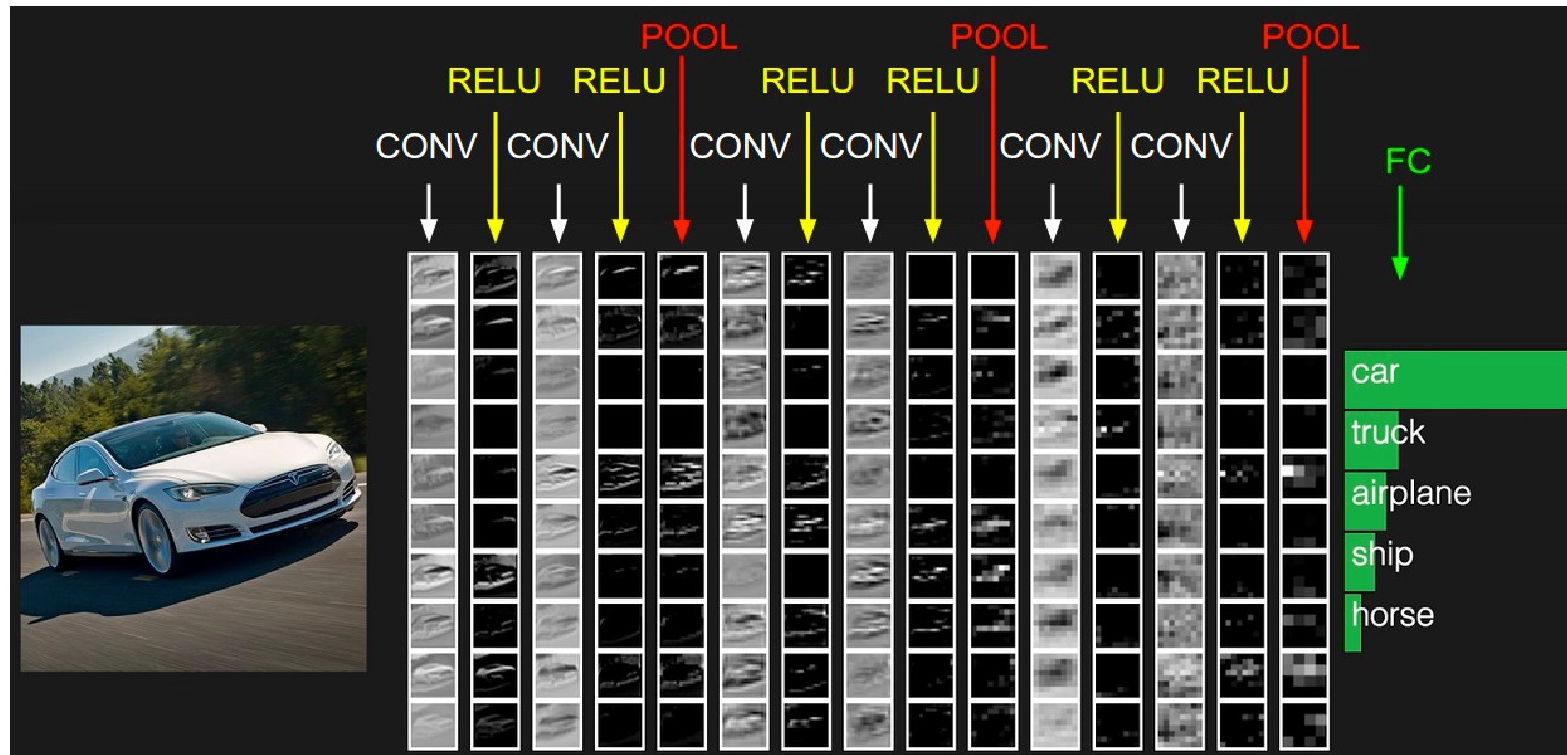
Finding inputs that maximize activations in different parts of the neuron network

- <https://distill.pub/2017/feature-visualization/>



Convolutional Layers beyond the First Layer

- Conv layer can well be a non-input layer:



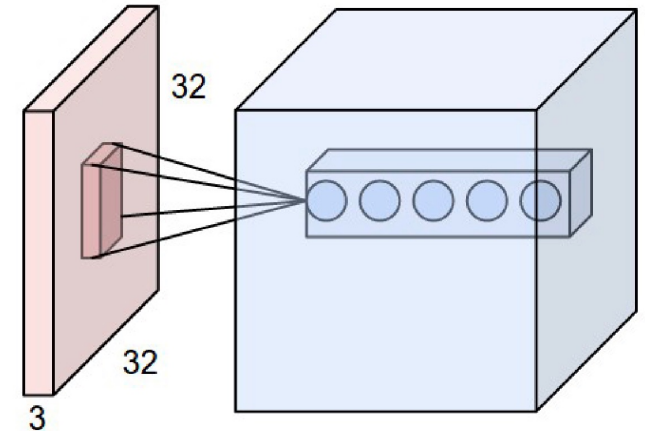
Convolutional Layers beyond the First Layer

Generalization: conv layer as non-input layer

- 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 patterns.

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



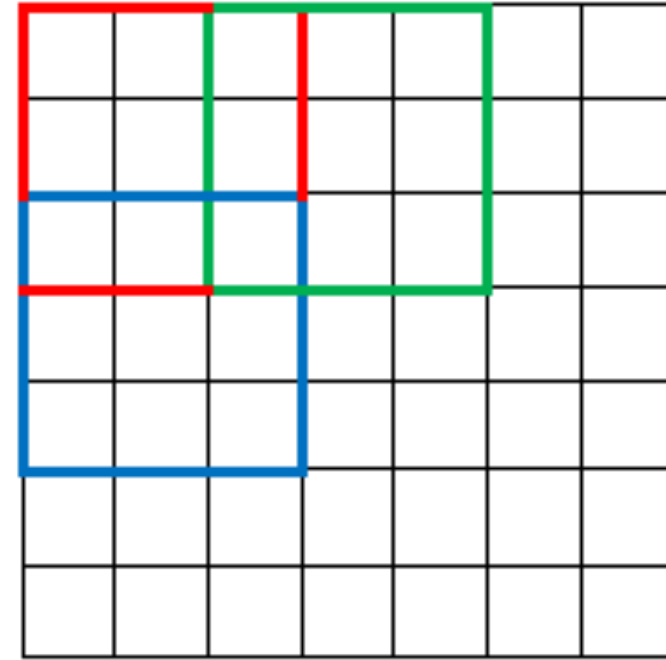
Convolutional Layer: more variants

Stride length S

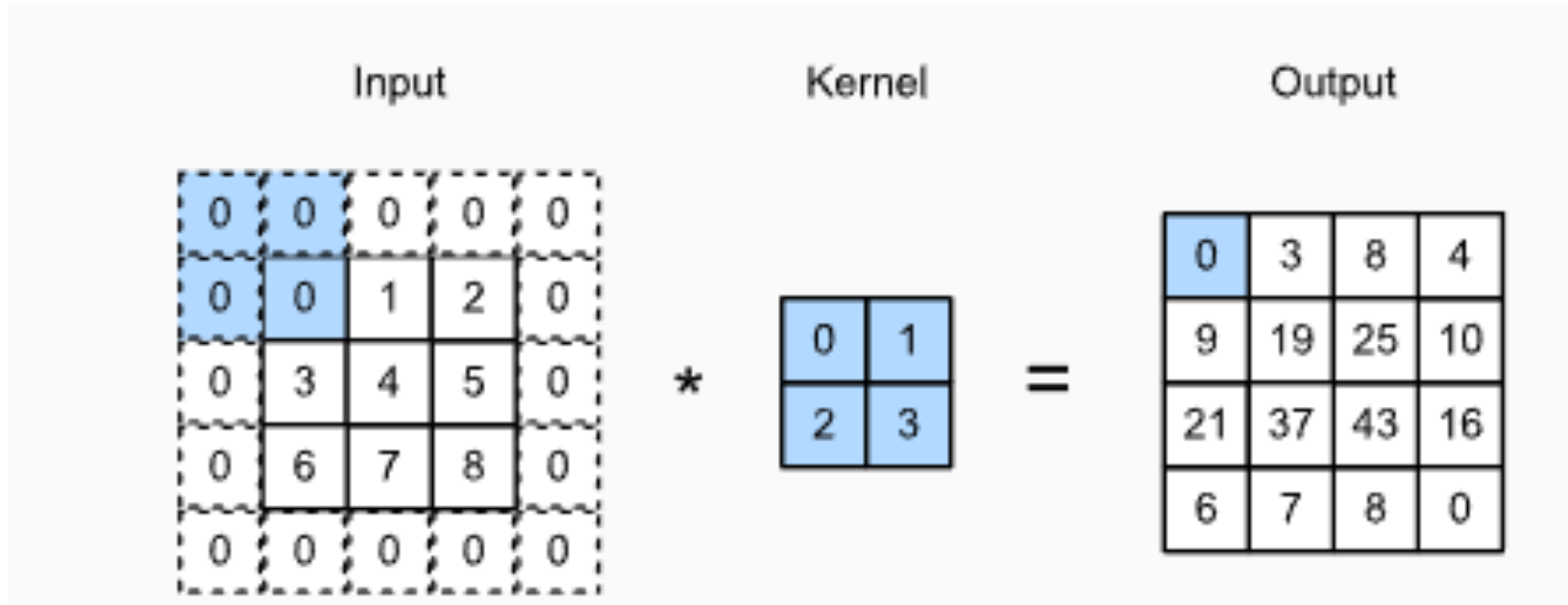
- Skip input regions; Move the sliding window of a filter not by 1 but by S .
- E.g., $S=2$ means skipping every other 5 by 5 region.

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)

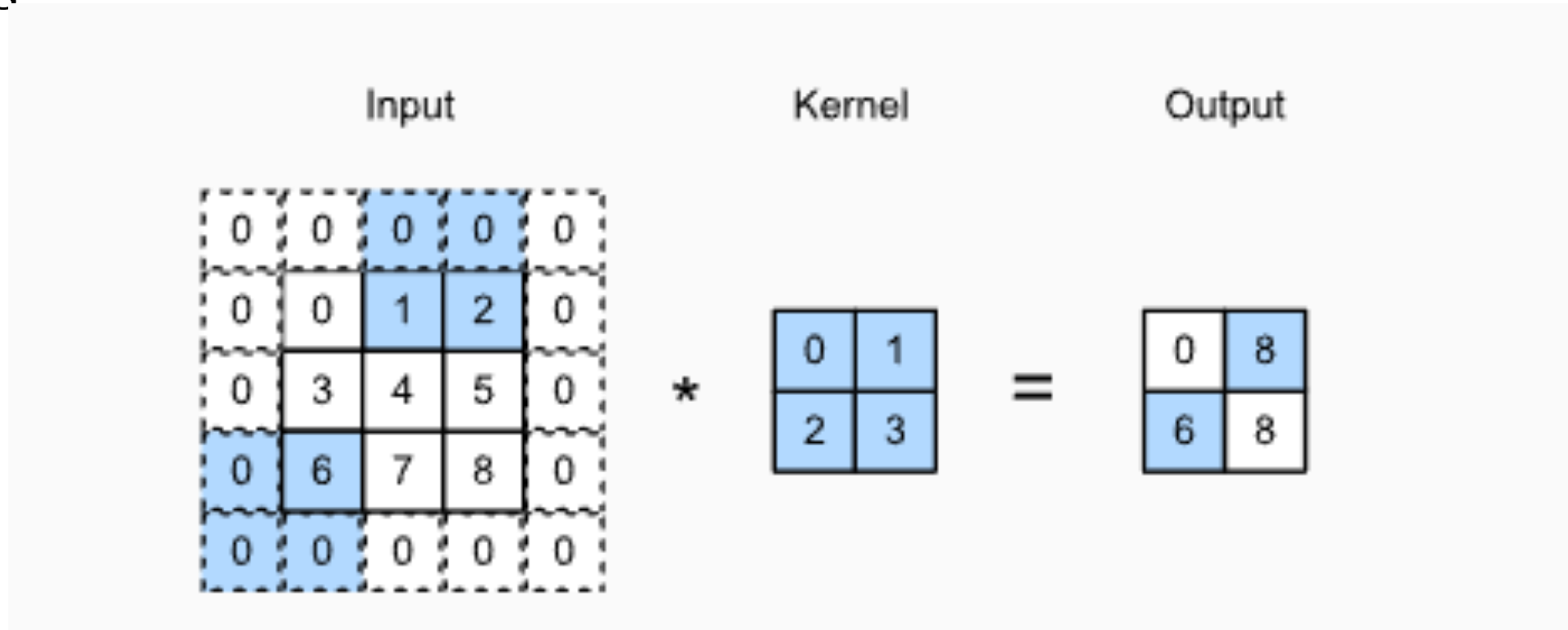


Example: P=1, S=1



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

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



Convolutional Layer: More variants

Stride length S

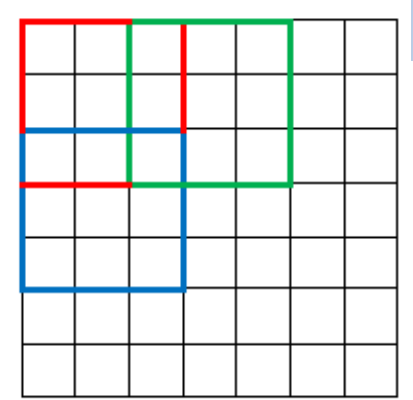
- Skip input regions; Move the sliding window of a filter not by 1 but by S .

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)

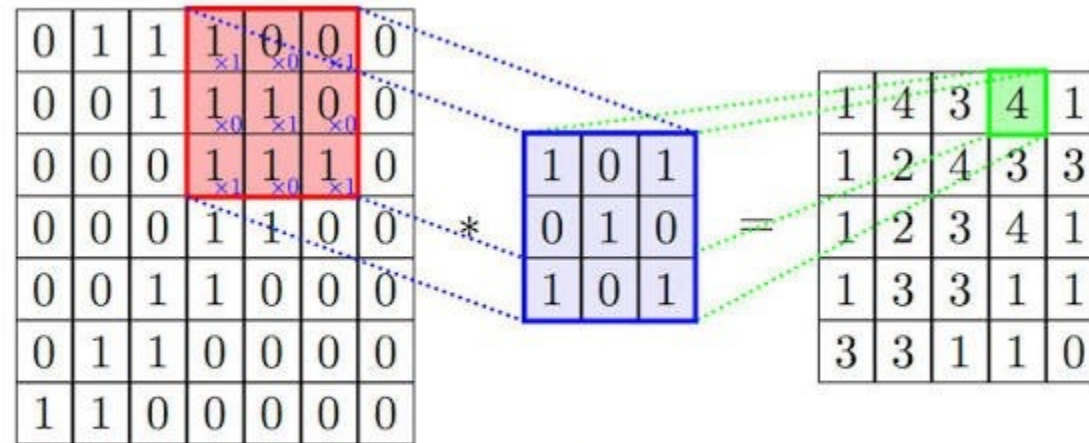


Announcements

- Mid-project presentation next Monday: schedule [here](#)
 - Alphabetical order
 - Tried to accommodate special requests
 - Feedback is welcome!
- I will give time cues (1-min, time up) for each talk
- Peer feedback is highly welcome 😊

Recap: Convolutional neural networks

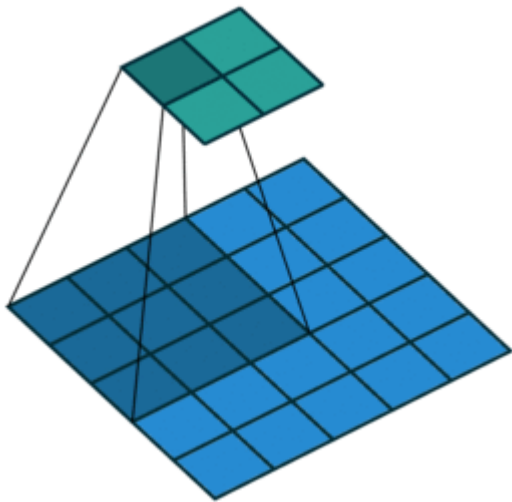
- Convolution with a $F \times F$ filter: for every $F \times F$ region of the original image, compute the inner product with the filter



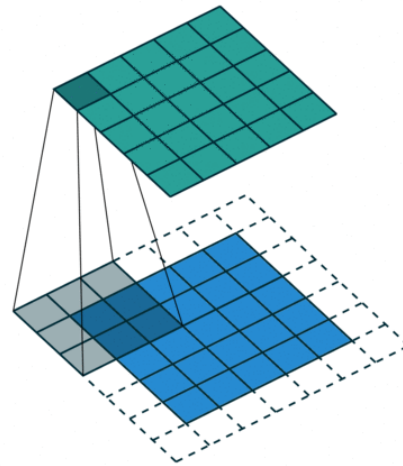
- Stride S : move the filter S pixels at a time
- Padding P : add extra P pixels around the border

Strides and padding: animations

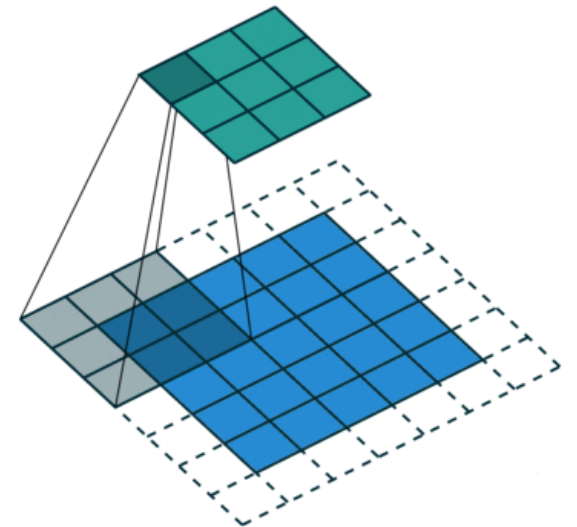
Strides only



Padding only

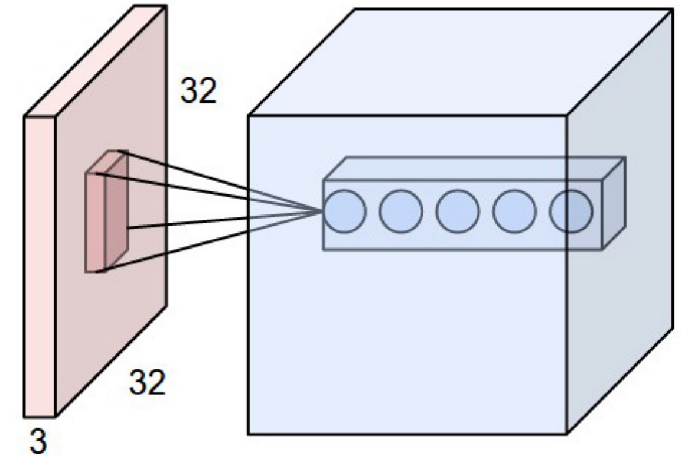


Strides + Padding



Convolution: multi-channel input & output maps

- In general, input map can have $c \geq 1$ channels (e.g. $c = 3$ for RGB images)
- Convolution needs to be defined with a $F \times F \times c$ filter
- In general, $D \geq 1$ filters can be applied simultaneously to the input (e.g. $D=5$ in the picture)
 - Output map has D channels



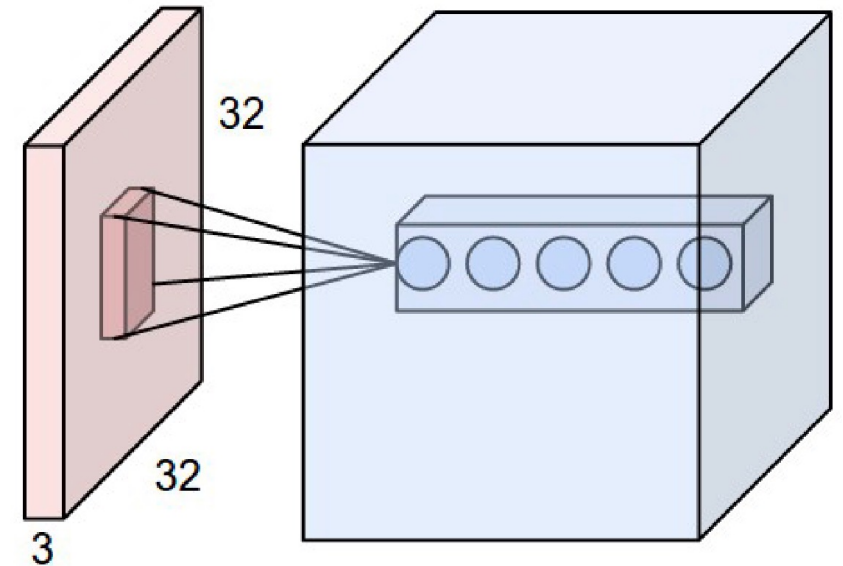
Convolutional Layer: Summary

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

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

Output $W_2 \times H_2 \times D_2$

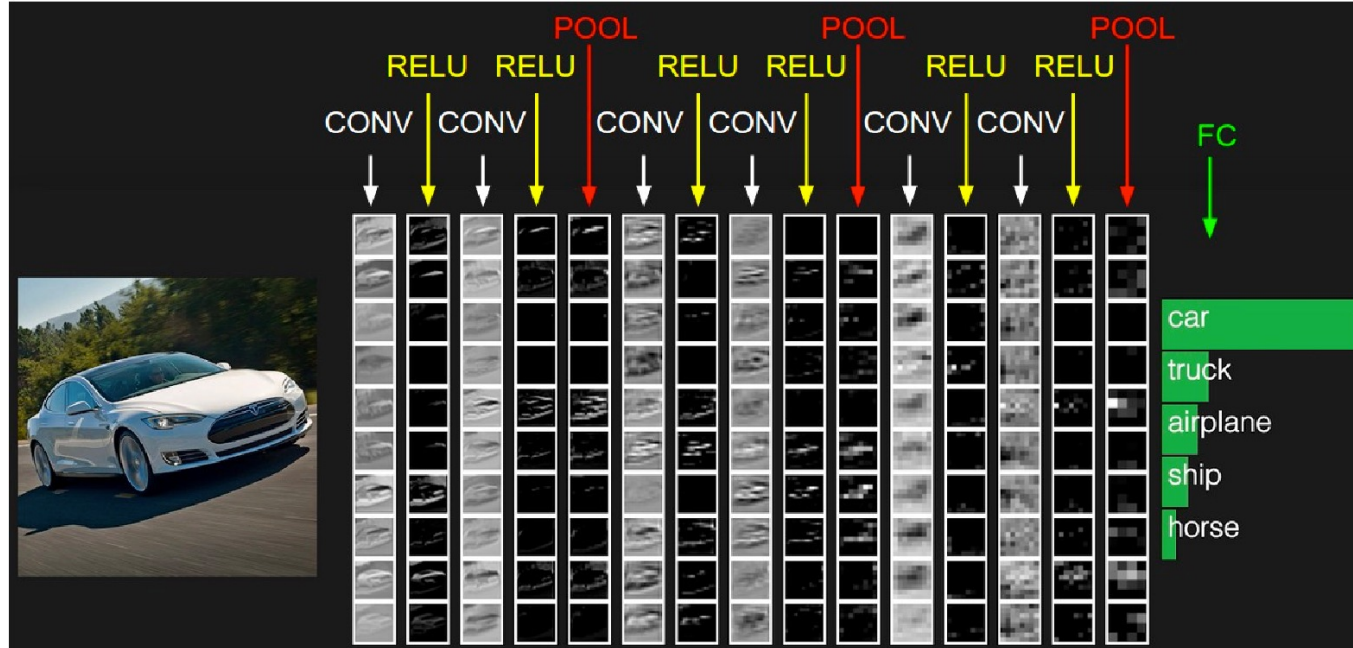
$$W_2 = \left\lfloor \frac{W_1 - F + 2P}{S} \right\rfloor + 1,$$
$$H_2 = \left\lfloor \frac{H_1 - F + 2P}{S} \right\rfloor + 1,$$
$$D_2 = K$$



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

Quick questions

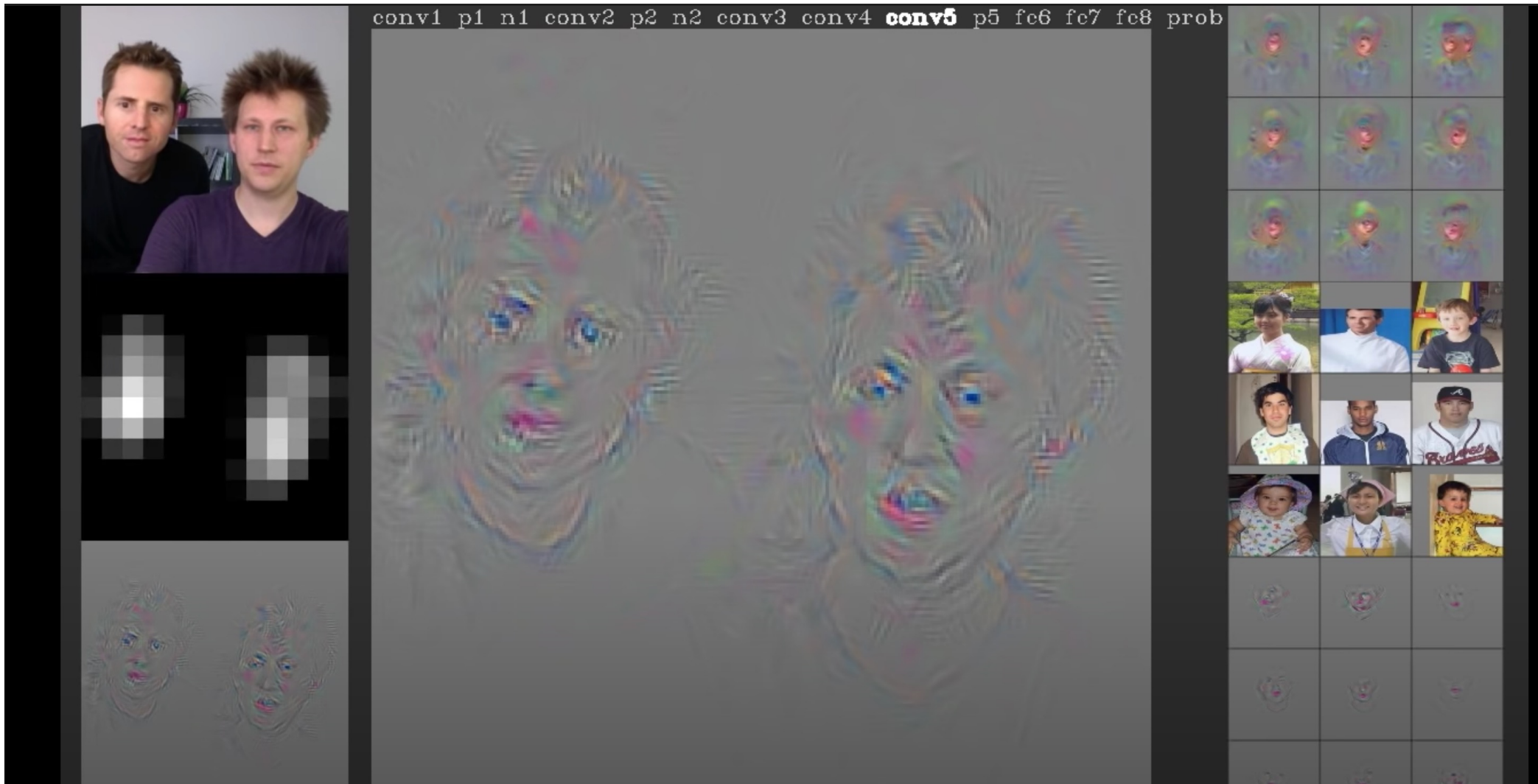
- How many filters does the first convolutional layer use? 10



- Suppose all filters are of spatial size 5×5 , how many parameters does the first convolutional layer have? $3 \times 5 \times 5 \times 10$
- What about the second convolutional layer? $10 \times 5 \times 5 \times 10$

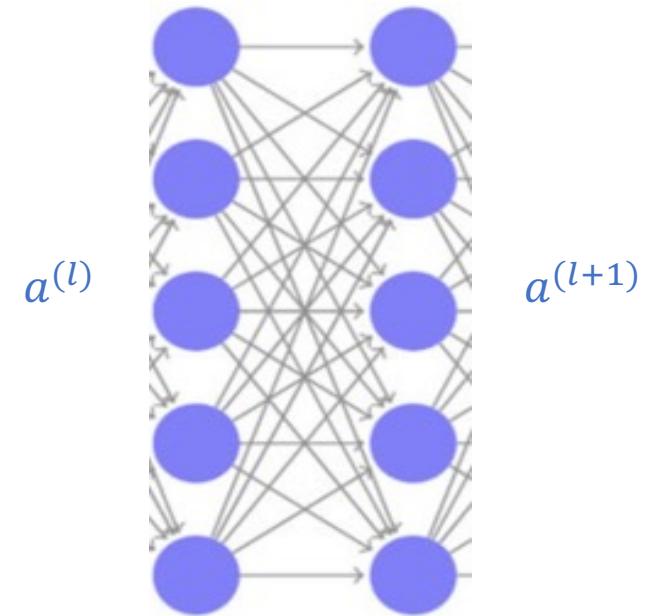
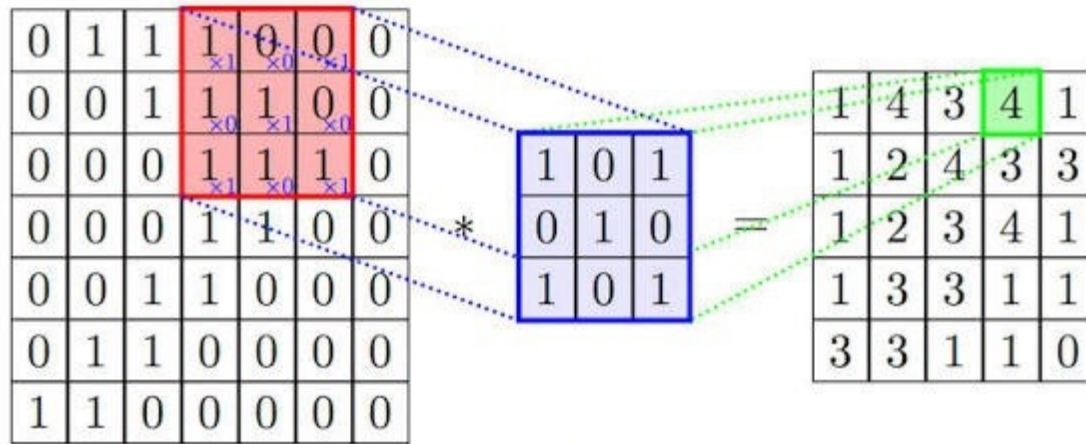
Seeing what happens in CNN

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



Comparison: FC vs Conv

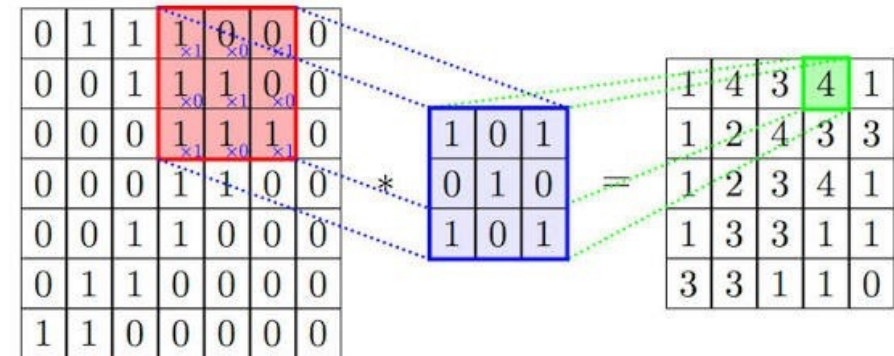
- Conv layer allows *parsimonious* representations:
 - Inter-layer connections are **local**
 - **parameter is shared** across spatial locations.
 - imposing **inductive bias** specialized for images



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

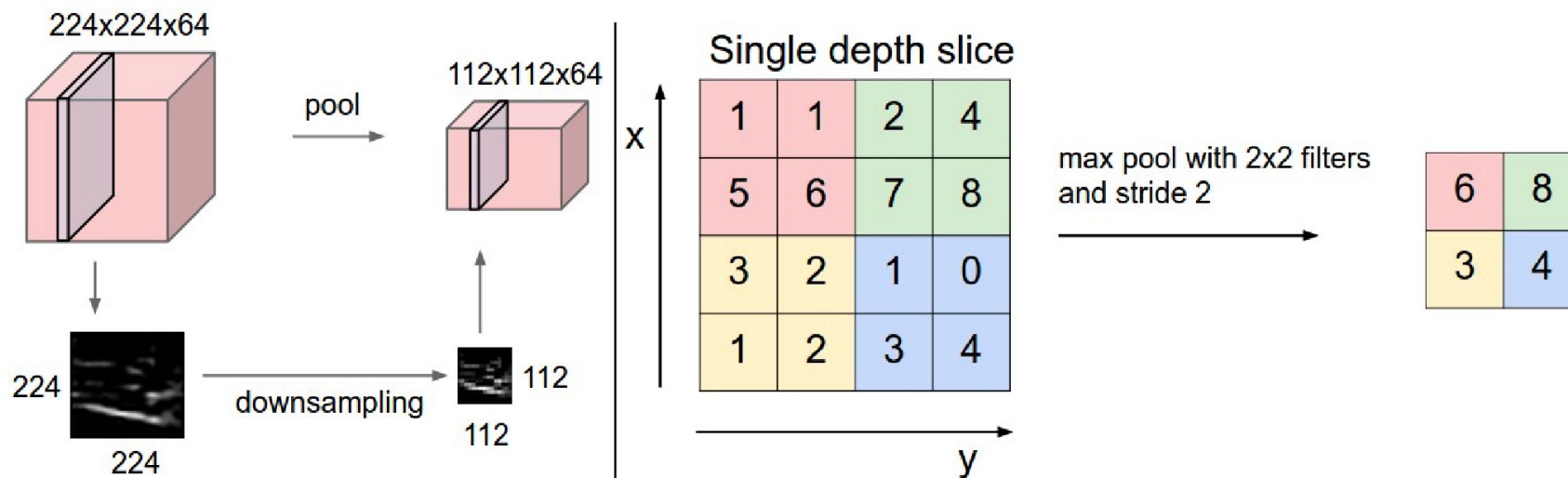
Input: 227x227x3, and the first conv layer output is 55x55x96 (96 filters)

- 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 output, 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 = $230,187 \times 290,400 = 66B$



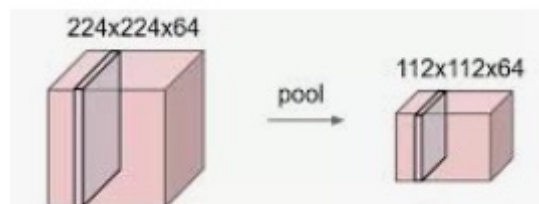
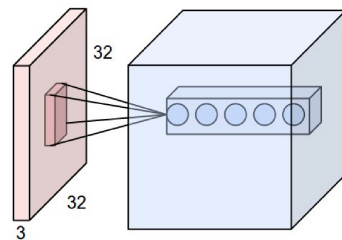
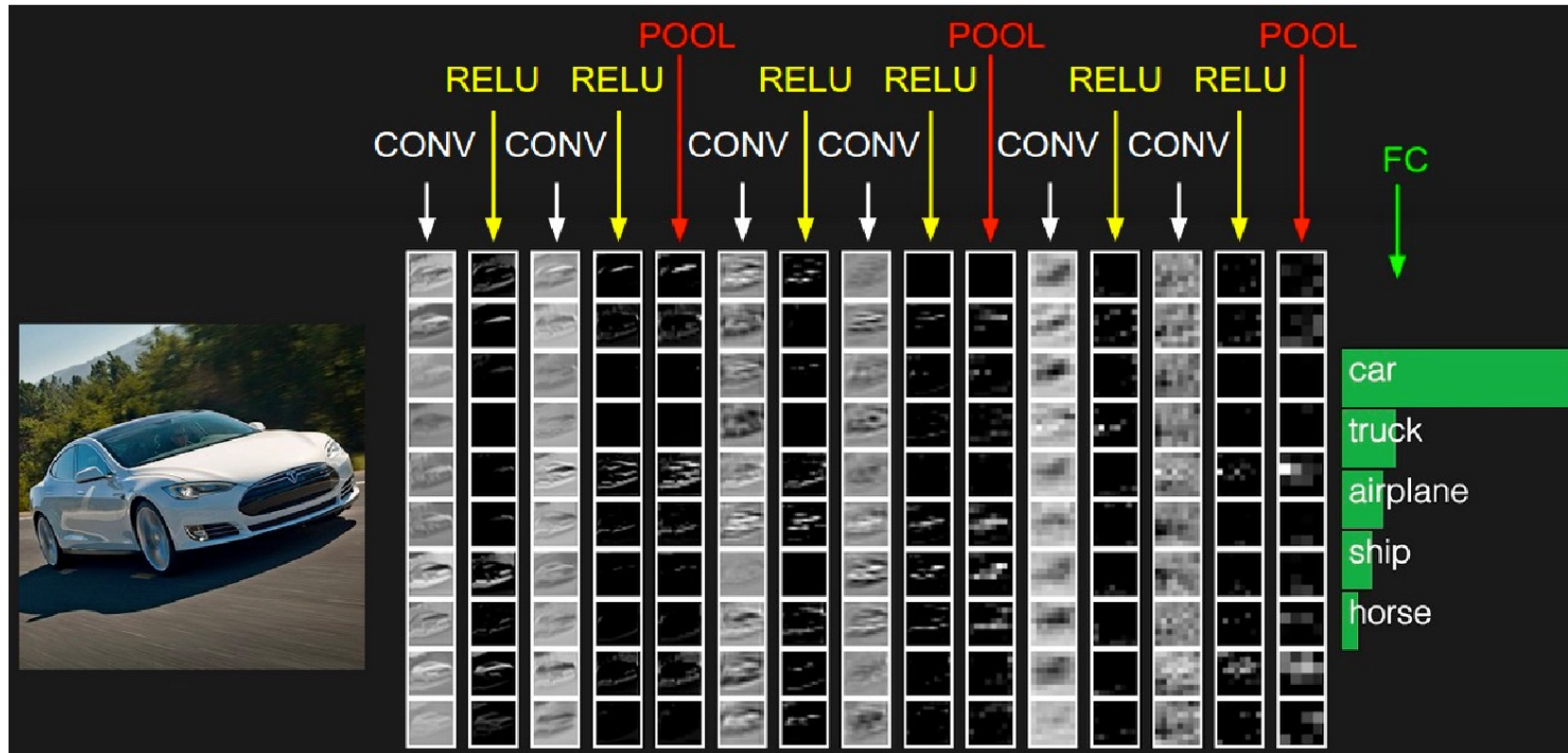
Pooling layer

- Usage: 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 tensor at a particular depth.
- Max pooling: run a particular filter that computes maximum, for each depth slice.



- Variation: average pooling (not popular).
- Recommended: Filter size $F=2$, stride length $S=2$.
- Note: **no parameters** to learn for this layer!

Typical architectural patterns in CNN

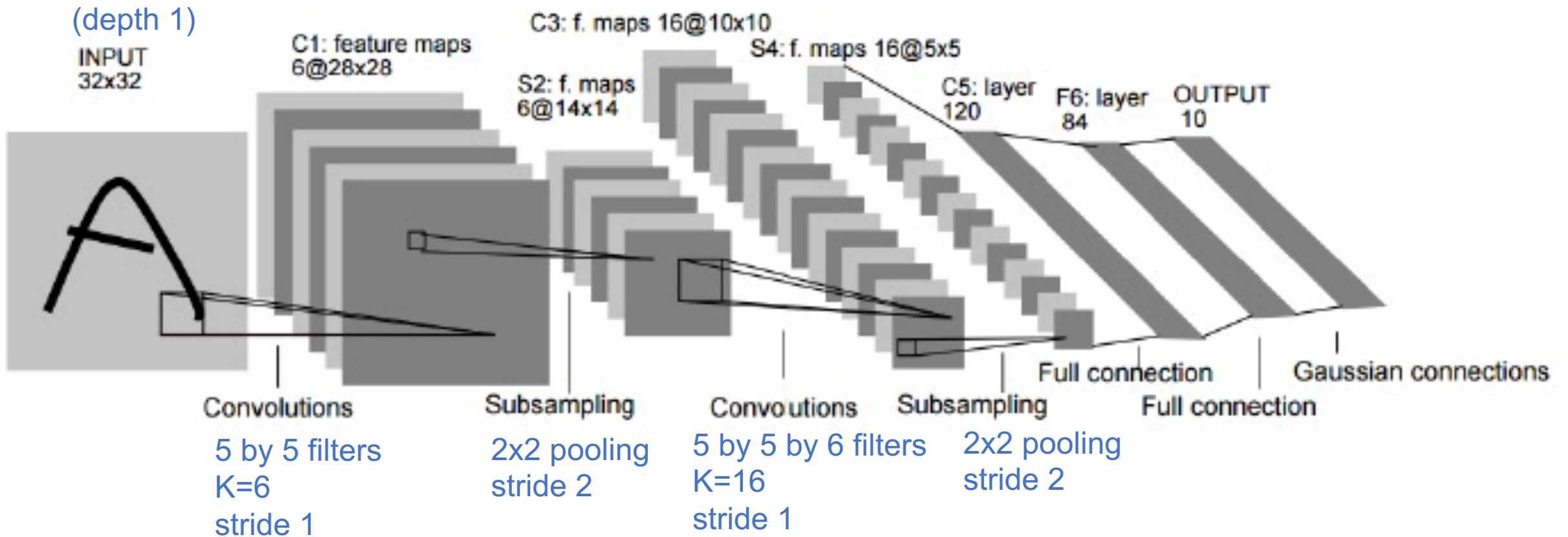


CNN examples

LeNet-5

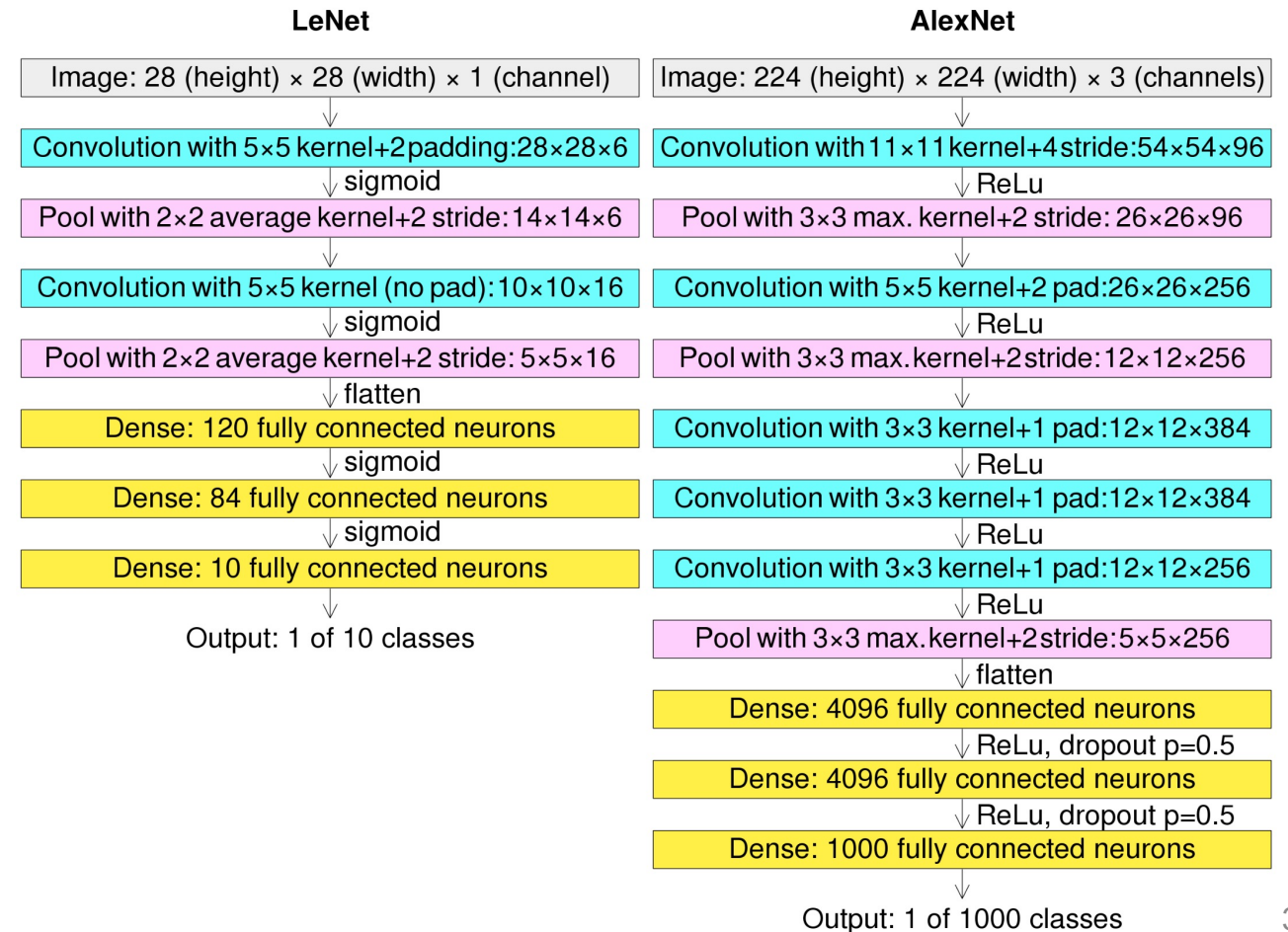
- Proposed in “Gradient-based learning applied to document recognition”, *by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in Proceedings of the IEEE, 1998*
- Apply convolution on 2D images (handwritten characters) and use backpropagation
- Structure: 2 convolutional layers (with pooling)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

LeNet-5



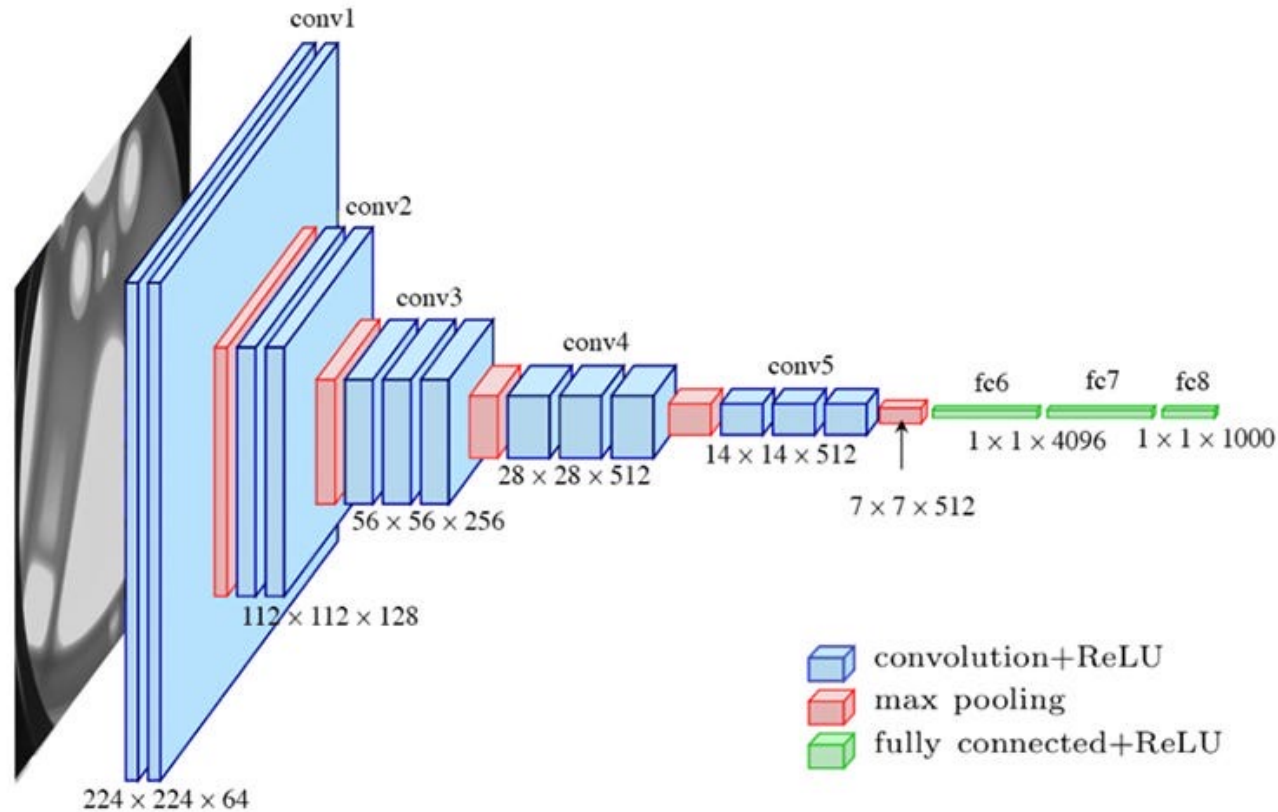
AlexNet (2012)

- 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

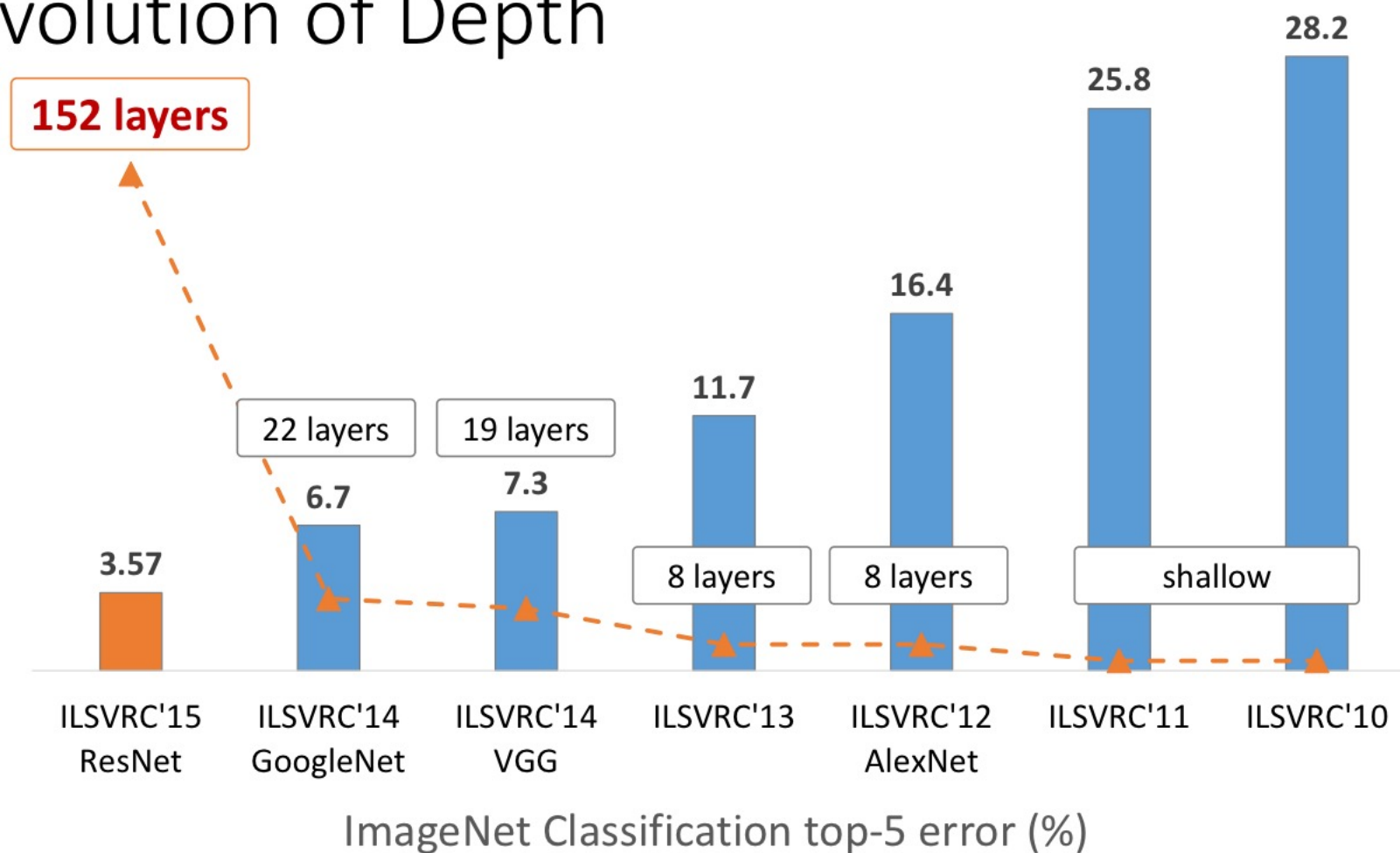
- Mimic large convolutional filters with multiple small (3x3) convolutional filters
- Amortizing memory cost: every time it halves the spatial size, double the # of filters



ResNet (2016): 3.5% error on ImageNet

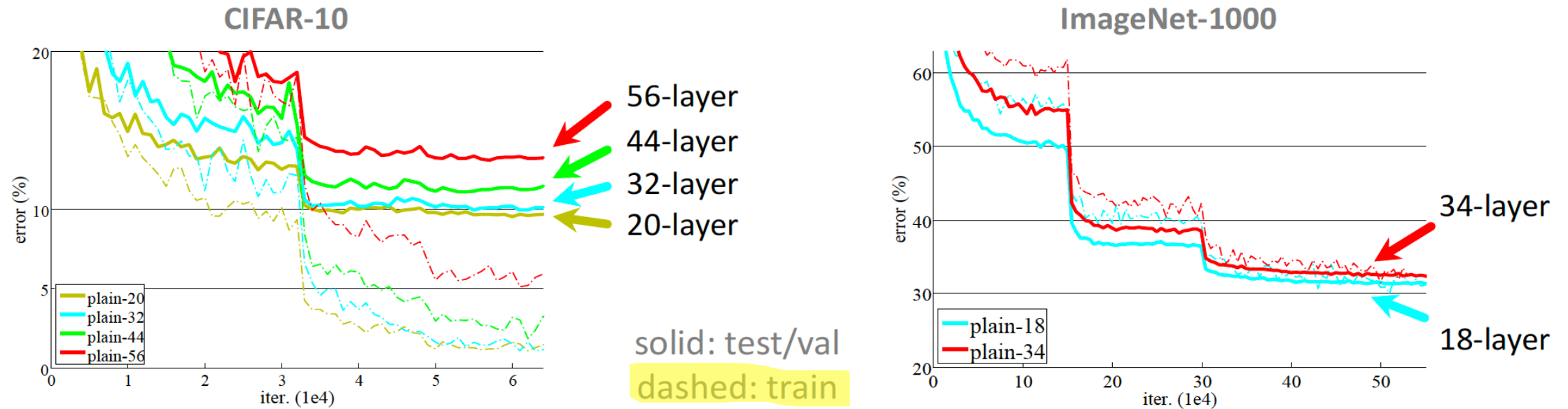
- Proposed in “Deep residual learning for image recognition” by *Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun*. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.
- Apply very deep networks with repeated **residual blocks**.
- Structure: simply stacking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

Revolution of Depth



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

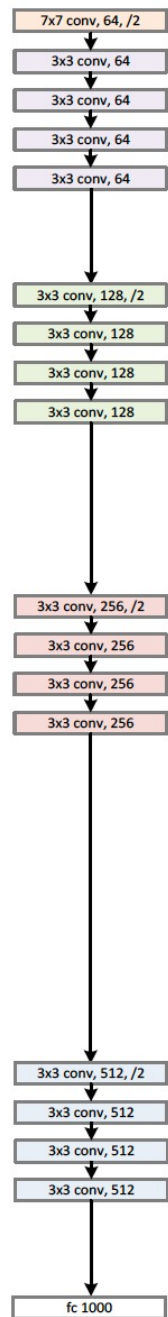
Deep nets seem to suffer



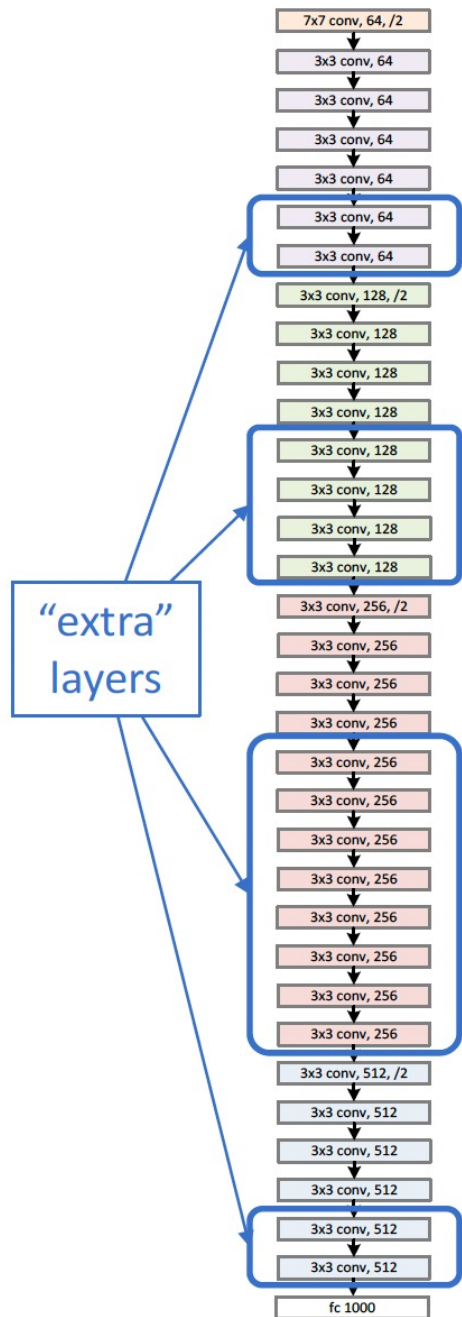
- “Overly deep” plain nets have **higher training error**
- A general phenomenon, observed in many datasets

(slides from Kaiming He)

a shallower
model
(18 layers)



a deeper
counterpart
(34 layers)

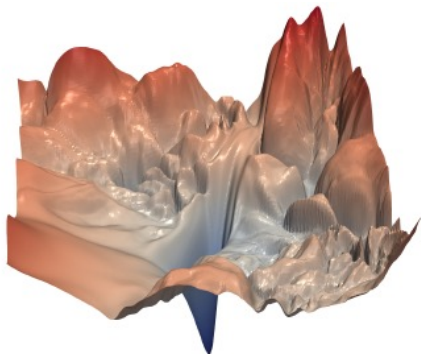


- A deeper model should not have **higher training error**
- A solution *by construction*:
 - original layers: copied from a learned shallower model
 - extra layers: set as **identity**
 - at least the same training error
- **Optimization difficulties**: solvers cannot find the solution when going deeper...

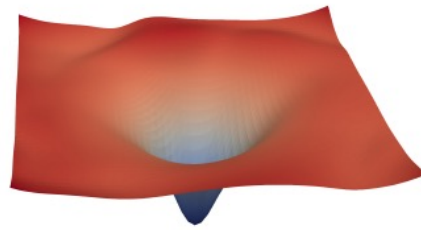
Key idea: skip connections

Skip connections

- $F(x)$ encodes residual representations
- When calculating gradients using backprop, 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 nicer

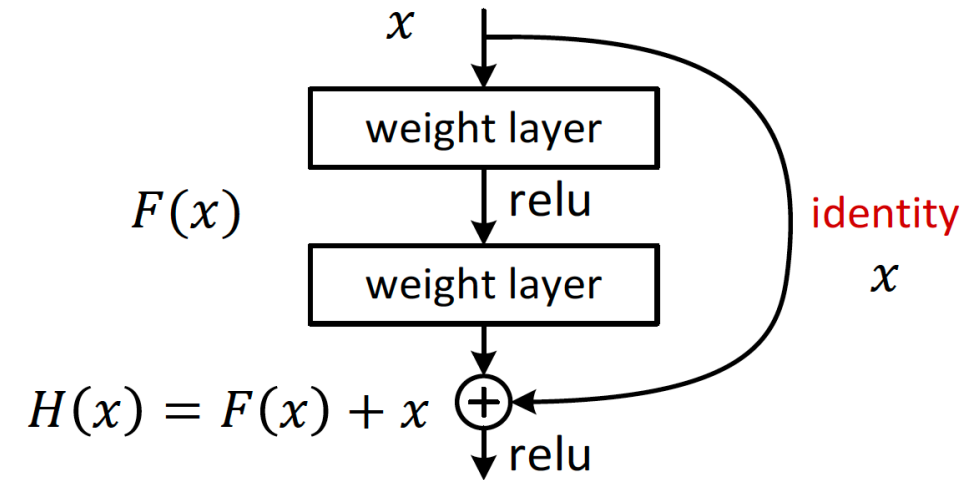


(a) without skip connections



(b) with skip connections

• Residual net



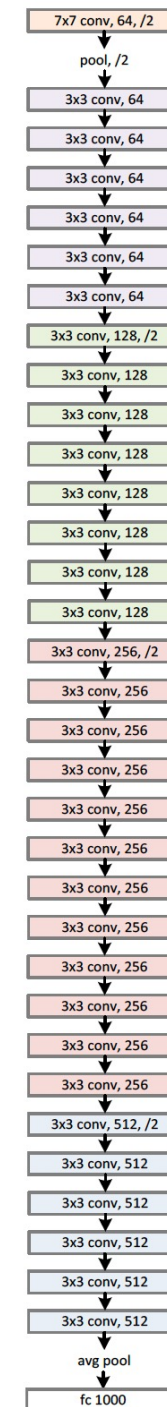
http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

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

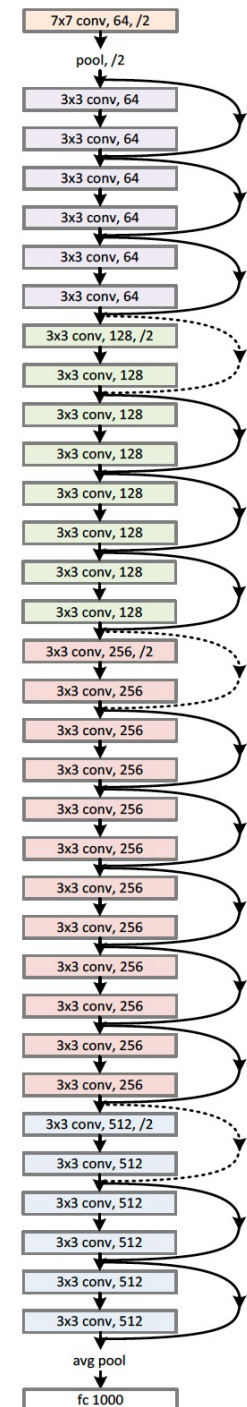
ResNet

- VGG-style scheme: halve the spatial size, double the # of filters
- Use conv layer with stride 2 occasionally to reduce the spatial dimension

plain net



ResNet



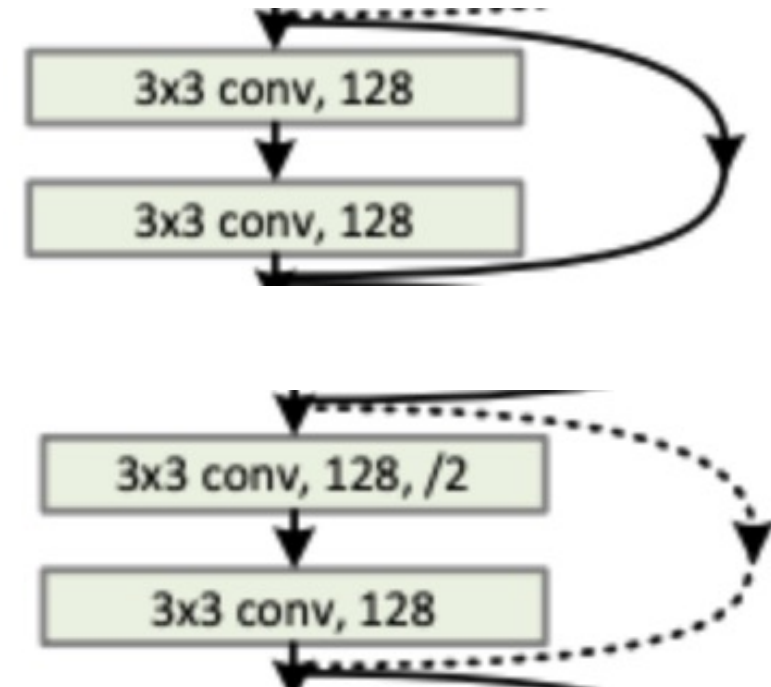
ResNet in PyTorch

Torchvision implementation:

https://pytorch.org/vision/0.8/_modules/torchvision/models/resnet.html

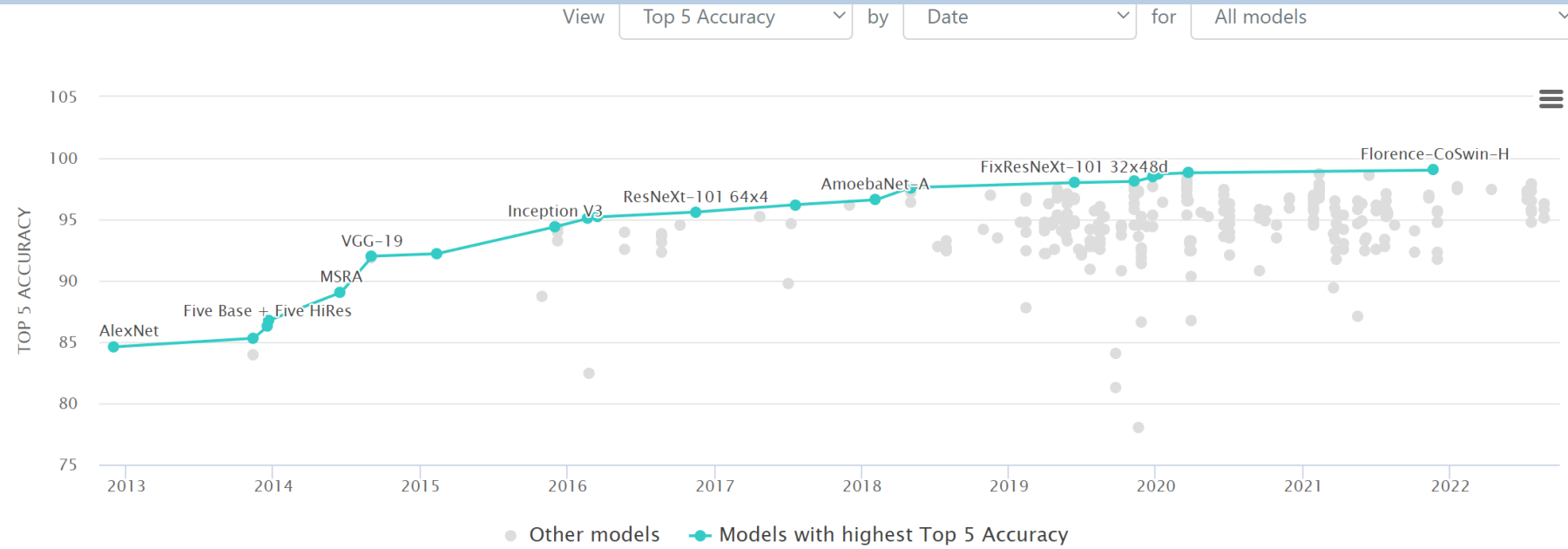
```
class BasicBlock(nn.Module):  
    def forward(self, x):  
        identity = x  
  
        out = self.conv1(x)  
        out = self.bn1(out)  
        out = self.relu(out)  
  
        out = self.conv2(out)  
        out = self.bn2(out)  
  
        if self.downsample is not None:  
            identity = self.downsample(x)  
  
        out += identity  
        out = self.relu(out)  
  
        return out
```

Batch normalization
(applied before activation)

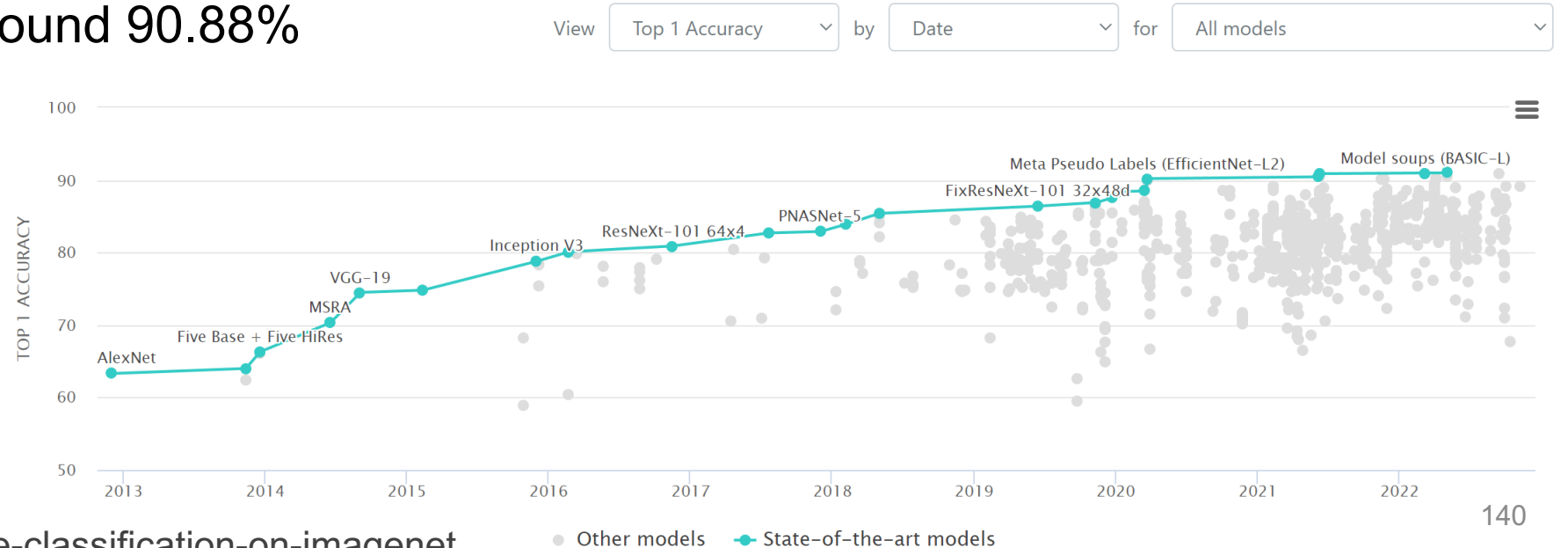


ImageNet nowadays

Top-5 accuracy is boring



SoTA top-1 accuracy is around 90.88%



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>