

# TUM - I2DL - Matrix derivatives

Dan Halperin - Tutor

November 2022

## 1 Affine layer

$$y = XW + b \quad (1)$$

Where  $X_{NxD}$     $W_{DxM}$     $b_{1xM}$ .

A known use case is the 1-dim case of a line equation:

$$y = ax + b$$

### 1.1 What is X?

- In the affine layer context, the matrix  $X$  is considered to be the input.
- In neural networks, we almost always refer to it as a **batch** of input elements (e.g. images).
- In some deep learning applications (e.g. "style-transfer"), it is also trained by backpropogation.
- Besides being the input of each layer of the network,  $X$  is also the **output** of the previous layer.

Let us take for example this one input instance (image) from the **MNIST** handwritten digits' dataset. Each **grayscale** image in this dataset is a  $1 \times 8 \times 8$  tensor: 1 for the channels, 8 for the height and 8 for the width.

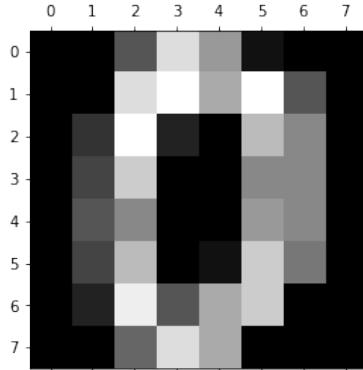


Figure 1: Mnist handwritten  $8 \times 8$  image of the digit 0

For the affine layer, as phrased in (1), each input instance is **flattened** to be a row vector inside  $X$ . Let us take a batch of 2 images from the MNIST dataset.

$$X = \begin{bmatrix} [x_{111} \dots x_{118}] & [x_{211} \dots x_{218}] \\ \vdots & \ddots & \vdots \\ [x_{181} \dots x_{188}] & [x_{281} \dots x_{288}] \end{bmatrix} \rightarrow \begin{bmatrix} [x_{111}, \dots, x_{118}, x_{121}, \dots, x_{181}, \dots, x_{188}] \\ [x_{211}, \dots, x_{218}, x_{221}, \dots, x_{281}, \dots, x_{288}] \end{bmatrix} \quad (2)$$

Here, the batch shape is  $2 \times 1 \times 8 \times 8$

**Question:** What if we had a 3-channels RGB images?

**Answer:** The images are flattened the same, row by row and channel by channel. The actual order doesn't matter, but it is important that it will remain consistent among all input instances, so the weights will correspond to the correct entries.

## 1.2 What is $W$ ?

- The coefficient matrix.
- In a learning model, they represent the **learnable weights**, and modified during the backpropagation step.
- If in  $X$  each row represents one input inside the batch, in  $W$  each column represents the weights that are attached from all input neurons (cells in the input vector) to one neuron in the next layer, which is the input to the next layer, as could be seen in [Figure 2](#).

## 1.3 Notes

- **Note:** It is not a linear function, but we treat it as an approximation. Why not? It doesn't follow the rules of linearity, where

$$f(x + y) = f(x) + f(y)$$

or

$$f(ax) = af(x)$$

- Another common notation of an affine layer is

$$y = Wx + b = ((Wx + b)^T)^T = (X^T W^T + b^T)^T \quad (3)$$

Which calculates the exact same thing, but results in a column vector and not a row.  $W$  weight vectors are now row vectors and  $X$  inputs are now column vectors. It is just a matter of how we construct our inputs and weights.

## 2 Derivatives

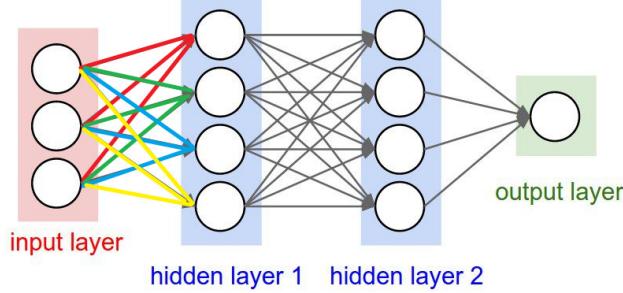


Figure 2: A neural network computational graph. Note: Although we always deal with batches of inputs, in the sketch, the input layer represents only one input instance (e.g one flattened image). Each colour represents a different weights column vector in  $W$ . Also, each neuron in the input layer (true to any neuron in the network) will collect the gradients from the flow on the colourful edges that are attached to it.

### 2.1 What is a gradient

It all depends on the function!

- 

$$f : \mathbb{R} \rightarrow \mathbb{R}, \quad x \in \mathbb{R}, \quad \frac{\partial f}{\partial x} = a \in \mathbb{R} \quad (4)$$

- Gradient:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}, \quad x \in \mathbb{R}^n, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} \in \mathbb{R}^n \quad (5)$$

- Gradient:

$$f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}, \quad x \in \mathbb{R}^{n \times m}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_{1,1}} & \cdots & \frac{\partial f}{\partial x_{1,m}} \\ \vdots & \ddots & \cdots \\ \frac{\partial f}{\partial x_{n,1}} & \cdots & \frac{\partial f}{\partial x_{n,m}} \end{bmatrix} \in \mathbb{R}^{n \times m} \quad (6)$$

- Jacobian:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad f(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}, \quad \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \quad (7)$$

- Note that if  $x$  was a row vector and so was the function 'image' (result), then this Jacobian matrix would have been transposed.

- An ugly Jacobian (Tensor - A multidimensional matrix):

$$f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n, \quad f \left( \begin{bmatrix} w_{11} & \dots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{mn} \end{bmatrix} \right) = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \frac{\partial f}{\partial w} = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \dots & \frac{\partial f_1}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial w_{n1}} & \dots & \frac{\partial f_1}{\partial w_{mn}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial w_{11}} & \dots & \frac{\partial f_n}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial w_{n1}} & \dots & \frac{\partial f_n}{\partial w_{mn}} \end{bmatrix} \quad (8)$$

- What is a gradient? It is the derivative scalar-valued differentiable function by a vector or a matrix input.
- **Super important:** Neural networks in general could take any shape of input, but they all result in a loss function, that gives a scalar  $L \in \mathbb{R}$ . That means:

- In the backpropagation step, the derivative of a learnable weight  $w_{ij}$  is to be calculated as a **scalar derivative**:

$$\frac{\partial L}{\partial w_{u,v}} = \sum_i \sum_j \frac{\partial L}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{u,v}} \quad (9)$$

Where  $i, j$  correspond to the rows and columns of  $\frac{\partial L}{\partial Y}$  in some function that utilizes  $w$ , such in  $Y = f(X) = XW + b$ .

- In the neural network backpropagation algorithm, we observe only the **current** layer at a time, as an abstraction. We do not try to think of the entire network at once, but step-by-step. Example:

Toy-Network:

- \* Affine()
- \* ReLU()
- \* **Affine()**
- \* Sigmoid()
- \* Loss()

When it comes to think of how to derive the current **Affine()** layer, we observe it as if it was a function with its own scope.

```
def affine_backward(dout, cache):
    x, w, b = cache
    dx, dw, db = None, None, None
    # some math
    return dx, dw, db
```

Figure 3: Scope of a function

According to the chain rule, the derivative of the loss function value  $L$  according to the weight matrix of our current affine layer  $W$ , would be:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \sigma(Y)} \oplus \frac{\partial \sigma(Y)}{\partial Y} \oplus \frac{\partial Y}{\partial W} \quad (10)$$

In this case,  $\frac{\partial L}{\partial \sigma(Y)} \frac{\partial \sigma(Y)}{\partial Y}$  is what we call **dout**, or the **upstream gradient**, and we assume it is already calculated before, as in our current scope (according to the relevant functions, of course). Now it is sent to our current scope, to be calculated as a part of the chain-rule, and sent up the stream to the next layer.

Also,  $\oplus$  in scalar derivatives represents a simple multiplication. However, in multidimensional derivatives,  $\oplus$  represents some unknown function, which we need to figure out.

Note the following abstraction. Each layer is calculated in turn according to the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \sigma} \oplus \frac{\partial \sigma}{\partial y} \oplus \frac{\partial y}{\partial W} \rightarrow \left( \frac{\partial L}{\partial \sigma} \right) \oplus \frac{\partial \sigma}{\partial y} \oplus \frac{\partial y}{\partial W} \rightarrow \left( dout_{\sigma} \oplus \frac{\partial \sigma}{\partial y} \right) \oplus \frac{\partial y}{\partial W} \rightarrow \left( dout_y \oplus \frac{\partial y}{\partial W} \right) \rightarrow dW \quad (11)$$

### 3 Stanford Article

- Original article by Stanford (cs231n): [Link](#)

Let's follow their example:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2 \times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2 \times 3} \quad (12)$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \quad (13)$$

Given a loss function  $Loss(Y) = L$ , we want to calculate  $\frac{\partial L}{\partial X}$  or  $\frac{\partial L}{\partial W}$ .

As seen in (6), the derivative of a scalar by a matrix, is a gradient / Jacobian matrix that has the same shape as the input. Moreover, we saw in (9), that the final derivative of the loss value  $L$  by **any** entry of any matrix in the whole neural network is just a **scalar**. For better understanding we could look at the computational graph of the network in, [Figure 2](#), to clearly see that each neuron collects and sums the upstream derivatives (from the loss up to it) - that it took part in calculation of, during the forward pass.

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \quad (14)$$

So, from (6) we know that the gradient of  $Y$  will have the same shape of  $Y$ , because  $L$  is a scalar, and it is calculated as a part of the chain-rule. This is the abstraction notion that is discussed above.

Let's derive  $W$ . Eventually, after the chain-rule, the derivative of  $W$  would have the same shape:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} & \frac{\partial L}{\partial w_{1,3}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} & \frac{\partial L}{\partial w_{2,3}} \end{pmatrix} \quad (15)$$

Now, this is important. We **do not** (!!) want to calculate the Jacobians. For a better explanation why, refer to the attached article. We have also learned that each entry of  $\frac{\partial L}{\partial W}$  is a scalar, that is computed as in (9).

So let's divide and conquer. It is always a better practice, because it's hard to wrap our minds on something bigger than scalars.

$$\frac{\partial L}{\partial w_{11}} = \sum_{i=1}^2 \sum_{j=1}^3 \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{11}} \quad (16)$$

For better visualization, we could look at it as a **dot product**, which is elementwise multiplication and then summation off all cells (Not what we know as np.dot() - this is confusing). Remember: when deriving a function, it is by the input variable (at least one):

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \begin{pmatrix} \frac{\partial y_{1,1}}{\partial w_{1,1}} & \frac{\partial y_{1,2}}{\partial w_{1,1}} & \frac{\partial y_{1,3}}{\partial w_{1,1}} \\ \frac{\partial y_{2,1}}{\partial w_{1,1}} & \frac{\partial y_{2,2}}{\partial w_{1,1}} & \frac{\partial y_{2,3}}{\partial w_{1,1}} \end{pmatrix} \quad (17)$$

If we go back to (24), we get:

$$\frac{\partial L}{\partial w_{11}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} \cdot \begin{pmatrix} x_{1,1} & 0 & 0 \\ x_{2,1} & 0 & 0 \end{pmatrix} \quad (18)$$

Now let's perform the dot product, and we get:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial y_{1,1}} x_{1,1} + \frac{\partial L}{\partial y_{2,1}} x_{2,1} \quad (19)$$

We can do that for every entry  $w_{i,j}$  in  $W$ , and we get:

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} x_{1,1} + \frac{\partial L}{\partial y_{2,1}} x_{2,1} & \frac{\partial L}{\partial y_{1,2}} x_{1,1} + \frac{\partial L}{\partial y_{2,2}} x_{2,1} & \frac{\partial L}{\partial y_{1,3}} x_{1,1} + \frac{\partial L}{\partial y_{2,3}} x_{2,1} \\ \frac{\partial L}{\partial y_{1,1}} x_{1,2} + \frac{\partial L}{\partial y_{2,1}} x_{2,2} & \frac{\partial L}{\partial y_{1,2}} x_{1,2} + \frac{\partial L}{\partial y_{2,2}} x_{2,2} & \frac{\partial L}{\partial y_{1,3}} x_{1,2} + \frac{\partial L}{\partial y_{2,3}} x_{2,2} \end{pmatrix} \quad (20)$$

From this matrix, with a little experience, we could derive

$$\frac{\partial L}{\partial W} = \begin{pmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_{1,1}} & \frac{\partial L}{\partial y_{1,2}} & \frac{\partial L}{\partial y_{1,3}} \\ \frac{\partial L}{\partial y_{2,1}} & \frac{\partial L}{\partial y_{2,2}} & \frac{\partial L}{\partial y_{2,3}} \end{pmatrix} = X^T \cdot \frac{\partial L}{\partial Y} \quad (21)$$

We could, of course, do the exact same thing in order to derive  $X$ , and we will see that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T \quad (22)$$

**Note:** This is only true, because  $L$  is a scalar. If we just looked at  $Y = XW \rightarrow \frac{\partial Y}{\partial W}$  would be a Jacobian.

### 3.1 Example

Example:

$$L \left( \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} x_{11}w_1 + x_{12}w_2 \\ x_{21}w_1 + x_{22}w_2 \end{bmatrix} \right) = y_1 + y_2 = x_{11}w_1 + x_{12}w_2 + x_{21}w_1 + x_{22}w_2$$

Where  $L()$  just sums them, for simplicity.

So,  $\frac{\partial L}{\partial x_{11}} = w_1$  is just a scalar

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_1} = \sum_i^n \sum_j^m \frac{\partial L}{\partial y_{ij}} \cdot \frac{\partial y_{ij}}{\partial w_1} = \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial w_1} + \frac{\partial L}{\partial y_2} \frac{\partial y_2}{\partial w_1} = 1 \cdot x_{11} + 1 \cdot x_{21}$$

### 3.2 What about the bias $b$ term in the affine layer?

We could, or course, do the trick of merging it into  $X$  and  $W$ , as we saw in the lecture.

If not:

1.

$$Y = XW + b$$

where  $X_{NxD}$ ,  $W_{DxM}$ ,  $b_{1xM}$ ,  $XW_{NxD}$

That means that each  $b_i$  in  $b$  corresponds to one feature in a row of  $XW$  - but to add them like that, it is quite impossible mathematically, right?

- NumPy uses **broadcasting** to duplicate the row vector  $b$  to be a matrix  $B_{NxD}$ , by simply copying  $b$   $N$  times and stacking them together along the rows, or the 0-axis. But that's just the programming application.
- Mathematically speaking, it's not  $Y = XW + b$ , but  $Y = XW + 1^N b$ , where  $1^N$  is a column vector, such that

$$\begin{bmatrix} 1_1 \\ \vdots \\ 1_N \end{bmatrix} [b_1 \ \dots \ b_M] = \begin{bmatrix} b_1 & \dots & b_M \\ \vdots & \ddots & \vdots \\ b_1 & \dots & b_M \end{bmatrix}_{N \times M}$$

That gives us the broadcast that python does by itself, and allows us to actually sum those matrices together.

Now, one can simply follow the exact same paradigm that we've shown above to solve for  $b$ , or we could just look at  $1^N b$  as another  $XW$ , and do the exact same thing as you did for  $XW$ , where  $1^N$  was  $X$  and  $b$  was  $W$ .

2. We see that the derivative of  $\frac{\partial L}{\partial b}$  is,

$$\frac{\partial L}{\partial b} = (1^N)^T \cdot \frac{\partial L}{\partial Y} = \left[ \sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}}, \dots, \sum_{i=1}^N \frac{\partial L}{\partial y_{i,M}} \right]$$

Which in NumPy translates into:

`np.sum(dout, axis = 0)`

## 4 Exercise

Given a simple neural network as above:

Toy-Network:

- **Affine()**
- **Sigmoid()**
- **Loss()**

Given again that:

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}_{2 \times 2} \quad W = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}_{2 \times 3} \quad (23)$$

$$Y = XW = \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{pmatrix} \quad (24)$$

1. Show a solution to compute the gradient of the **Sigmoid** layer, w.r.t to the upstream gradient. Hint: have a look again at the abstraction of equation (11)
2. Replace the sigmoid layer with  $f(x) = x^2$ . For the following input  $X$  and weight matrix  $W$ :

$$X = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}_{2 \times 2} \quad W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}_{2 \times 2}$$

$$y = XW$$

$$\text{Loss}(X) = \sum_i^N x_i$$

Find:

- (a) loss value.
- (b)  $\frac{\partial L}{\partial f}$
- (c)  $\frac{\partial L}{\partial y}$
- (d)  $\frac{\partial L}{\partial X}$
- (e)  $\frac{\partial L}{\partial W}$

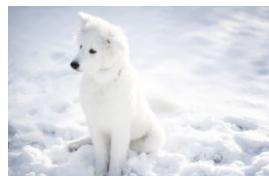
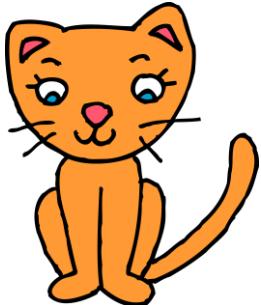
Solutions:

- (a) 52
- (b)  $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$
- (c)  $\begin{pmatrix} 2 & 2 \\ 10 & 10 \end{pmatrix}$
- (d)  $\begin{pmatrix} 4 & 4 \\ 20 & 20 \end{pmatrix}$

$$(e) \begin{pmatrix} 20 & 20 \\ 32 & 32 \end{pmatrix}$$

# Machine Learning Basics

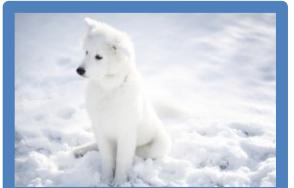
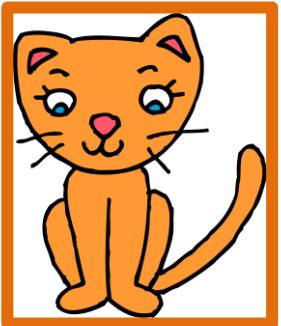
# Machine Learning



Task



# Image Classification



All

Images

Videos

News

Shopping

More

Settings

Tools

SafeSearch ▾



Cute



And Kittens



Clipart



Drawing



Cute Baby



White Cats And Kittens



Pose

Illumination

Appearance

# Image Classification



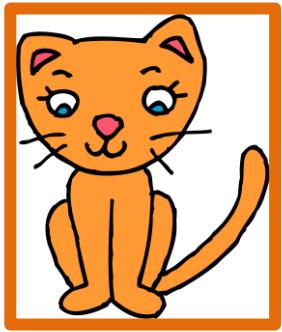
Occlusions



# Image Classification

Background clutter





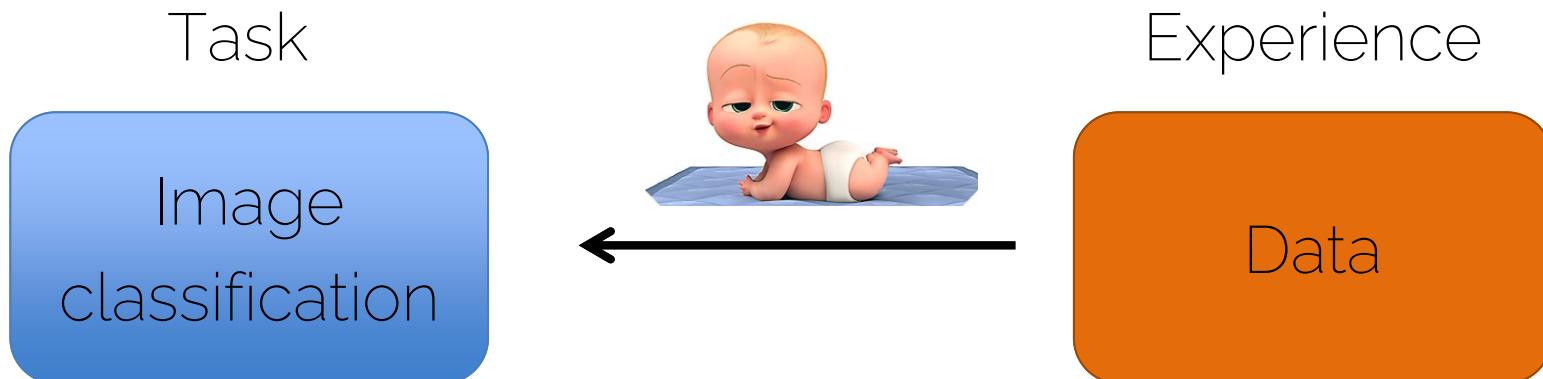
# Image Classification

Representation



# Machine Learning

- How can we learn to perform image classification?



# Machine Learning

Unsupervised learning

- No label or target class
- Find out properties of the structure of the data
- Clustering (k-means, PCA, etc.)

Supervised learning

# Machine Learning

Unsupervised learning

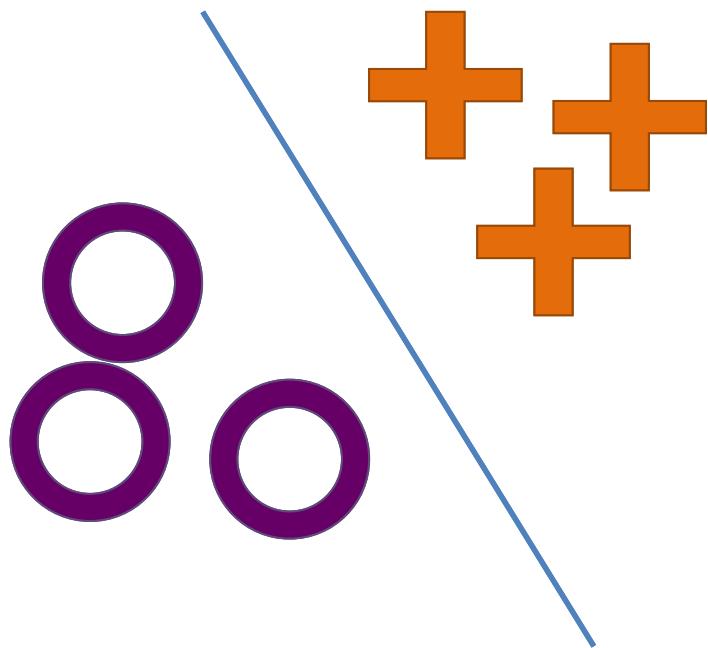


Supervised learning



# Machine Learning

Unsupervised learning

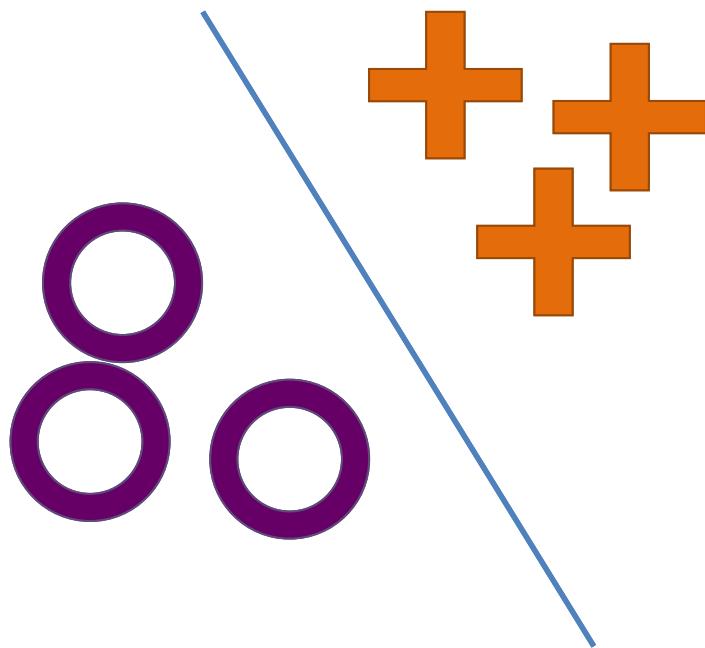


Supervised learning

- Labels or target classes

# Machine Learning

Unsupervised learning

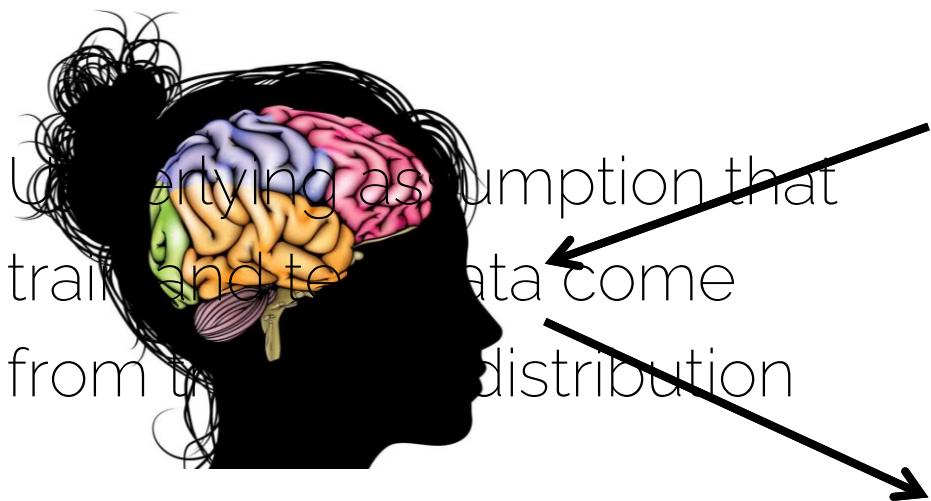


Supervised learning



# Machine Learning

- How can we learn to perform image classification?



Experience

Test data

# Machine Learning

- How can we learn to perform image classification?

Task

Image  
classification

Experience

Performance  
measure

Accuracy

Data

# Machine Learning

Unsupervised learning



Supervised learning



Reinforcement learning



# Machine Learning

Unsupervised learning



Supervised learning



Reinforcement learning



# Machine Learning

Unsupervised learning



Supervised learning

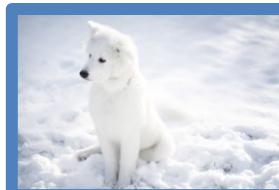
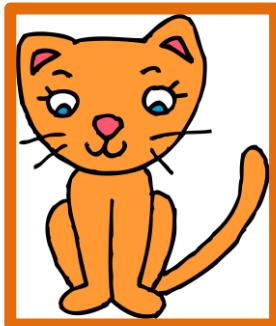


Reinforcement learning



# A Simple Classifier

# Nearest Neighbor



# Nearest Neighbor

NN classifier = dog



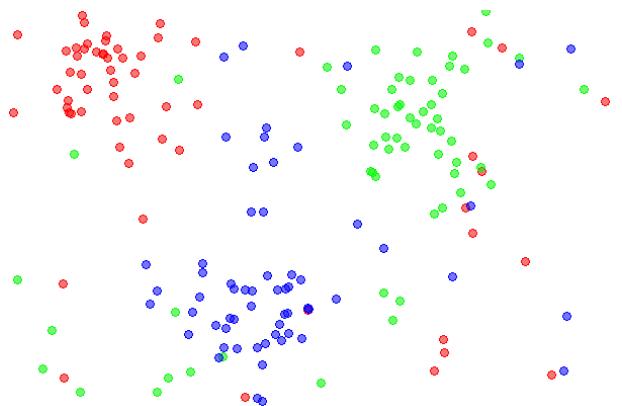
# Nearest Neighbor



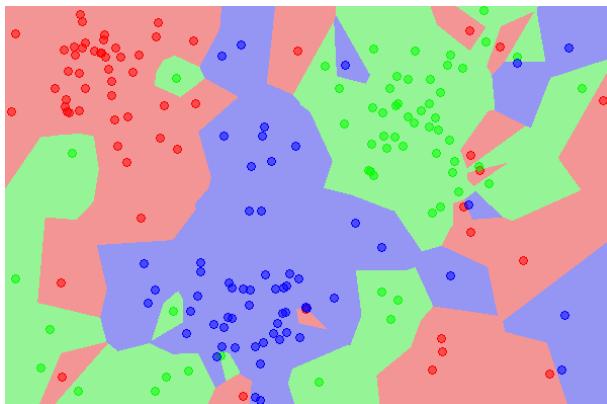
$k$ -NN classifier = cat

# Nearest Neighbor

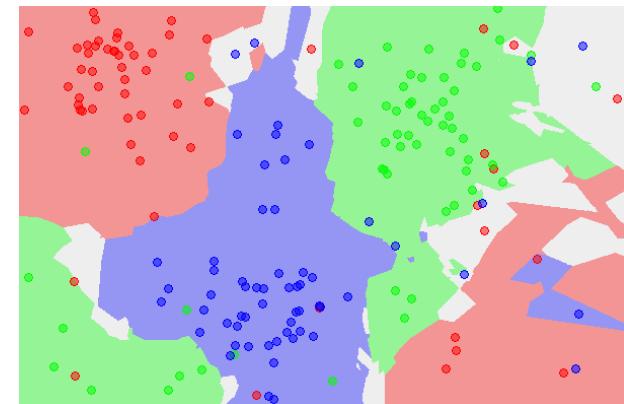
The Data



NN Classifier



5NN Classifier



How does the NN classifier perform on training data?

What classifier is more likely to perform best on test data?

Source: <https://commons.wikimedia.org/wiki/File:Data3classes.png>

# Nearest Neighbor

- Hyperparameters
  - L1 distance :  $|x - c|$
  - L2 distance :  $\|x - c\|_2$
  - No. of Neighbors:  $k$
- These parameters are problem dependent.
- How do we choose these hyperparameters?

# Basic Recipe for Machine Learning

- Split your data



Find your hyperparameters

Other splits are also possible (e.g., 80%/10%/10%)

# Basic Recipe for Machine Learning

- Split your data



# Cross Validation

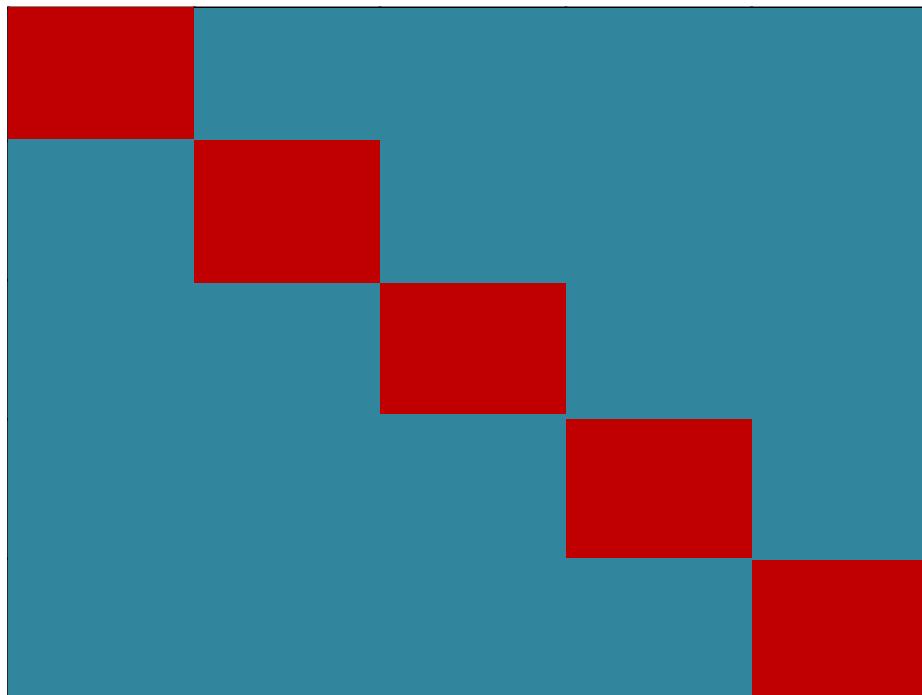
Run 1

Run 2

Run 3

Run 4

Run 5



train

validation

Split the **training data** into N folds

# Cross Validation



Find your hyperparameters

Why do cross validation?  
Why not just train and test?

# Cross Validation

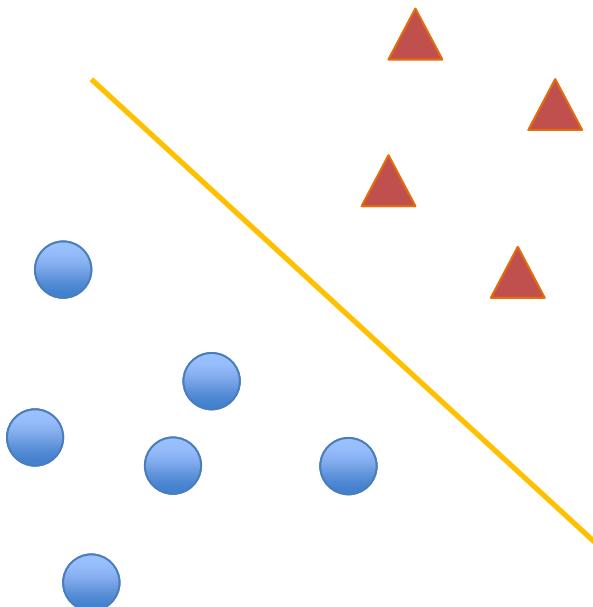


Test set is only used once!

Why do cross validation? Why not just train and test?

# Linear Decision Boundaries

This lecture

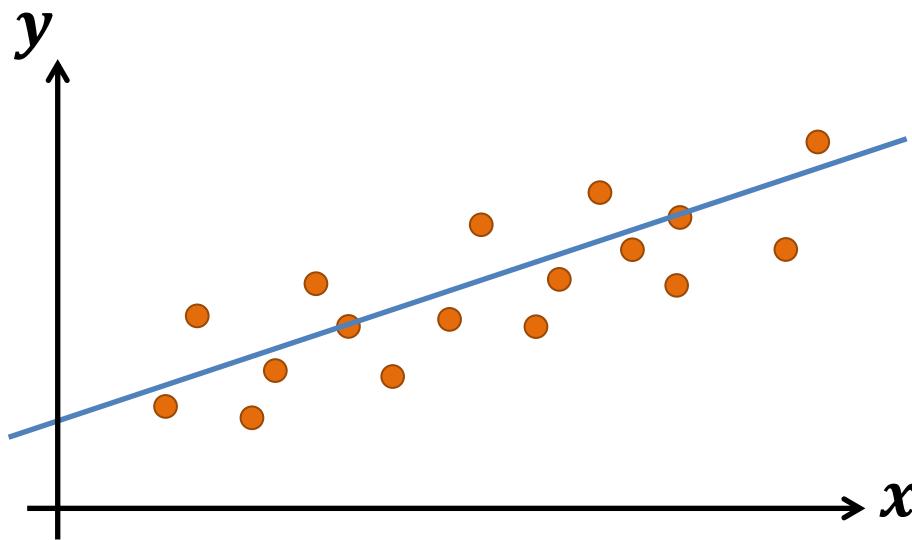


What are the pros  
and cons for using  
linear decision  
boundaries?

# Linear Regression

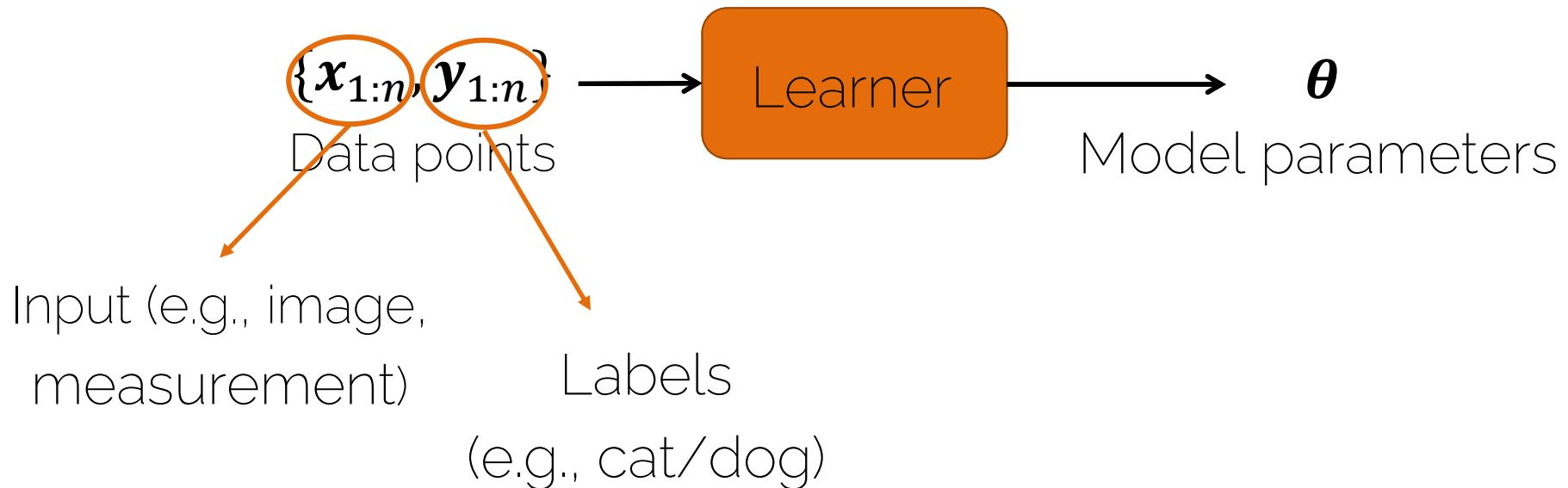
# Linear Regression

- Supervised learning
- Find a linear model that explains a target  $\mathbf{y}$  given inputs  $\mathbf{x}$



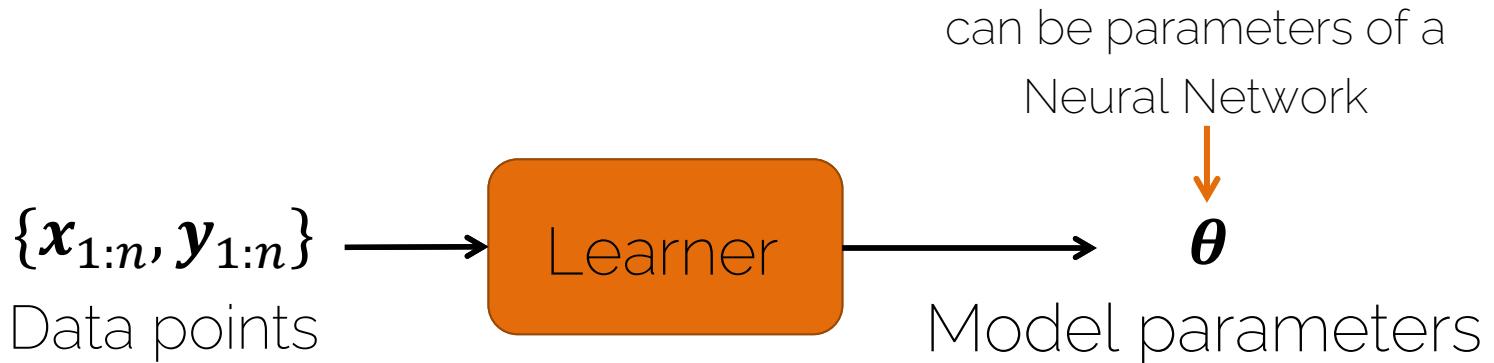
# Linear Regression

Training



# Linear Regression

Training



Testing



# Linear Prediction

- A linear model is expressed in the form

$$\hat{y}_i = \sum_{j=1}^d x_{ij} \theta_j$$

input dimension

weights (i.e., model parameters)

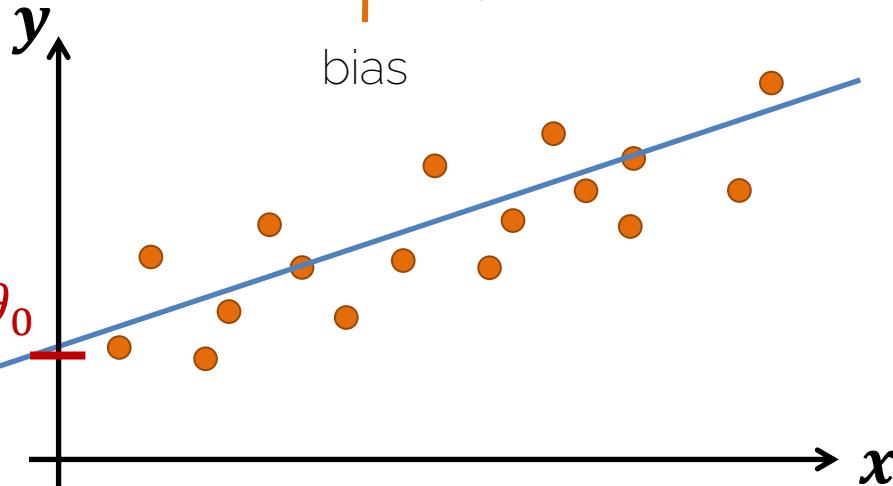
Input data, features

The diagram illustrates the components of a linear prediction equation. The equation is  $\hat{y}_i = \sum_{j=1}^d x_{ij} \theta_j$ . A purple arrow points from the label "input dimension" to the superscript "d" above the summation symbol. Two orange circles highlight the terms  $x_{ij}$  and  $\theta_j$ , with a blue arrow pointing from the label "weights (i.e., model parameters)" to the  $\theta_j$  term. An orange arrow points from the label "Input data, features" to the  $x_{ij}$  term.

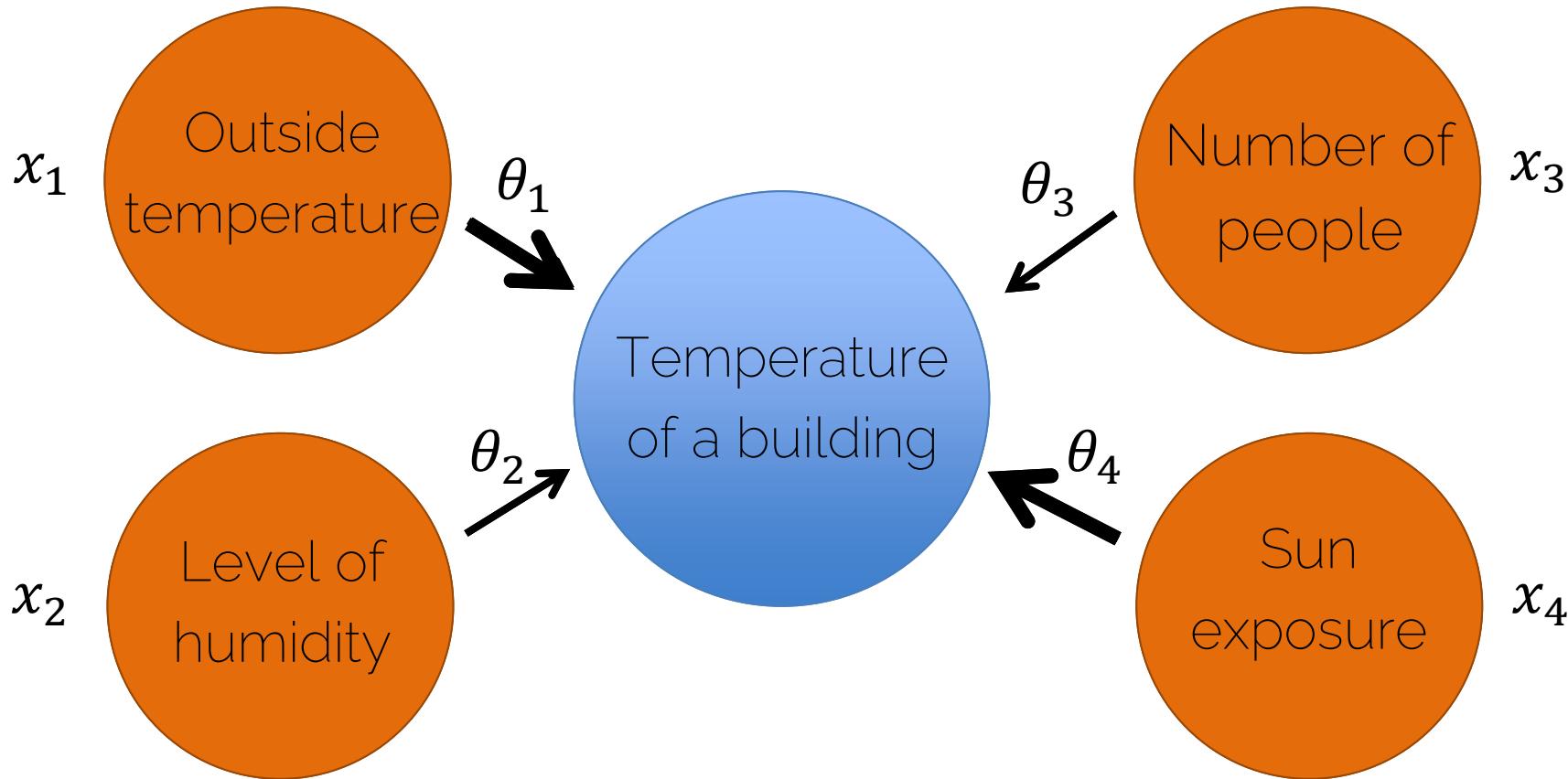
# Linear Prediction

- A linear model is expressed in the form

$$\hat{y}_i = \theta_0 + \sum_{j=1}^d x_{ij} \theta_j = \theta_0 + x_{i1} \theta_1 + x_{i2} \theta_2 + \dots + x_{id} \theta_d$$



# Linear Prediction



# Linear Prediction

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \theta_0 + \begin{bmatrix} x_{11} & \cdots & x_{1d} \\ x_{21} & \cdots & x_{2d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix} \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$



$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \Rightarrow \hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

# Linear Prediction

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

Prediction

Input features  
(one sample has  $d$  features)

Model parameters  
( $d$  weights and 1 bias)

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

The diagram illustrates the linear prediction equation  $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$ . It shows the prediction vector  $\hat{\mathbf{y}}$ , the input feature matrix  $\mathbf{X}$ , and the model parameter vector  $\boldsymbol{\theta}$ . Arrows point from the text labels to the corresponding parts of the equation: an arrow from 'Prediction' to  $\hat{\mathbf{y}}$ , an arrow from 'Input features (one sample has  $d$  features)' to the matrix  $\mathbf{X}$ , and an arrow from 'Model parameters ( $d$  weights and 1 bias)' to the vector  $\boldsymbol{\theta}$ .

# Linear Prediction

Temperature  
of the building

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} 1 & -25 & 50 & 2 & 50 \\ 1 & -10 & 50 & 0 & 10 \end{bmatrix} \cdot \begin{bmatrix} 0.2 \\ 0.64 \\ 0 \\ 1 \\ 0.14 \end{bmatrix}$$

Bias      Outside temperature      Humidity      Number people      Sun exposure (%)

MODEL

The diagram illustrates the calculation of linear predictions for two variables,  $\hat{y}_1$  and  $\hat{y}_2$ . It shows the input feature vector and the weight matrix being multiplied to produce the output vector. The feature vector includes a bias term (1) and four other features: Outside temperature, Humidity, Number people, and Sun exposure (%). The weight matrix has four rows corresponding to these features. The resulting prediction vector is  $\begin{bmatrix} 0.2 \\ 0.64 \\ 0 \\ 1 \\ 0.14 \end{bmatrix}$ .

# Linear Prediction



Temperature  
of the building

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} 1 & 25 & 50 & 2 & 50 \\ 1 & -10 & 50 & 0 & 10 \end{bmatrix} \cdot \begin{bmatrix} 0.2 \\ 0.64 \\ 0 \\ 1 \\ 0.14 \end{bmatrix}$$

Bias      Outside temperature      Humidity      Number people      Sun exposure (%)

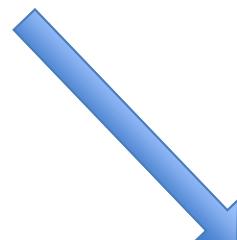
MODEL

How do we  
obtain the  
model?

# How to Obtain the Model?

Data points

$\mathbf{x}$



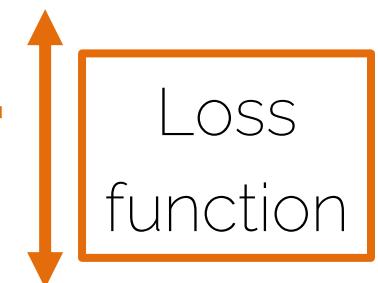
Optimization

Model parameters

$\theta$

Labels (ground truth)

$y$



Estimation

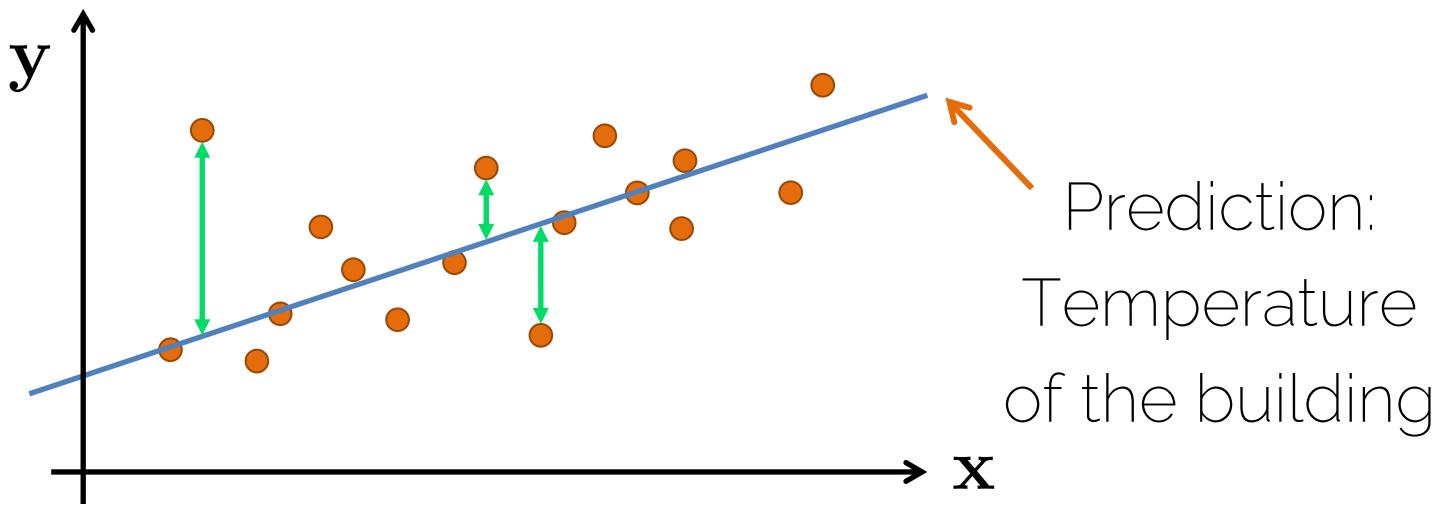
$\hat{y}$



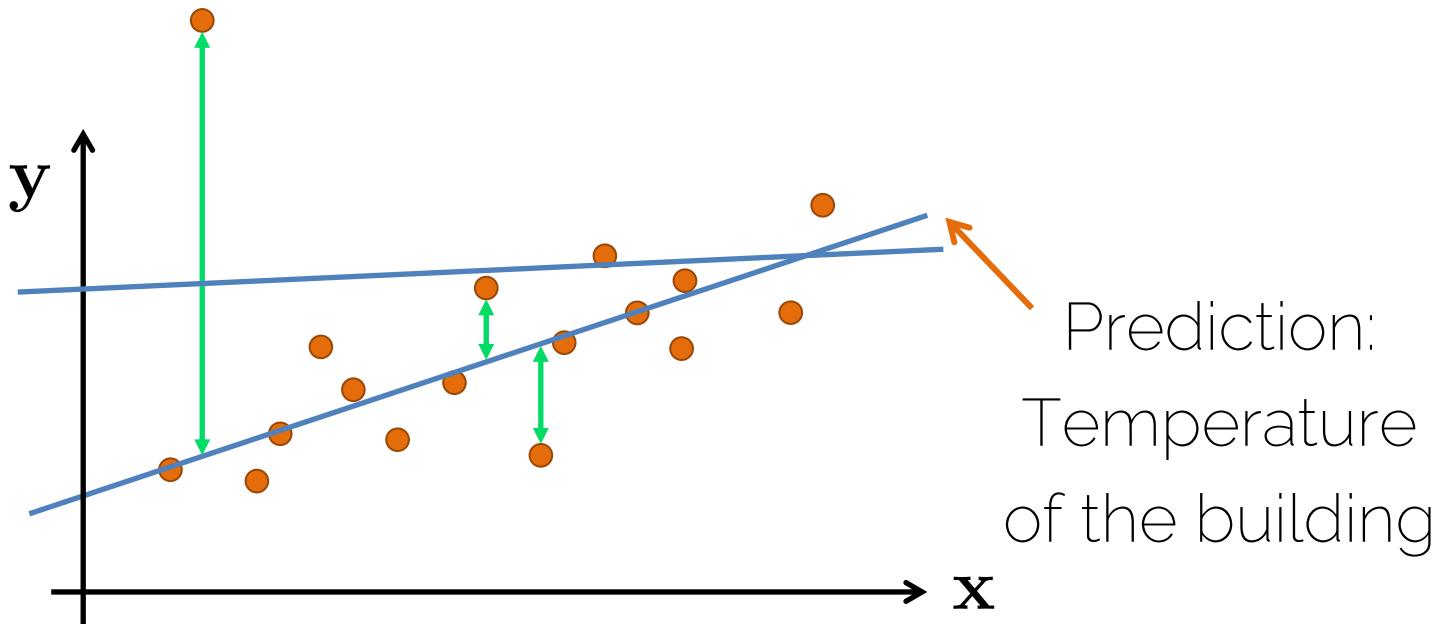
# How to Obtain the Model?

- **Loss function:** measures how good my estimation is (how good my model is) and tells the optimization method how to make it better.
- **Optimization:** changes the model in order to improve the loss function (i.e., to improve my estimation).

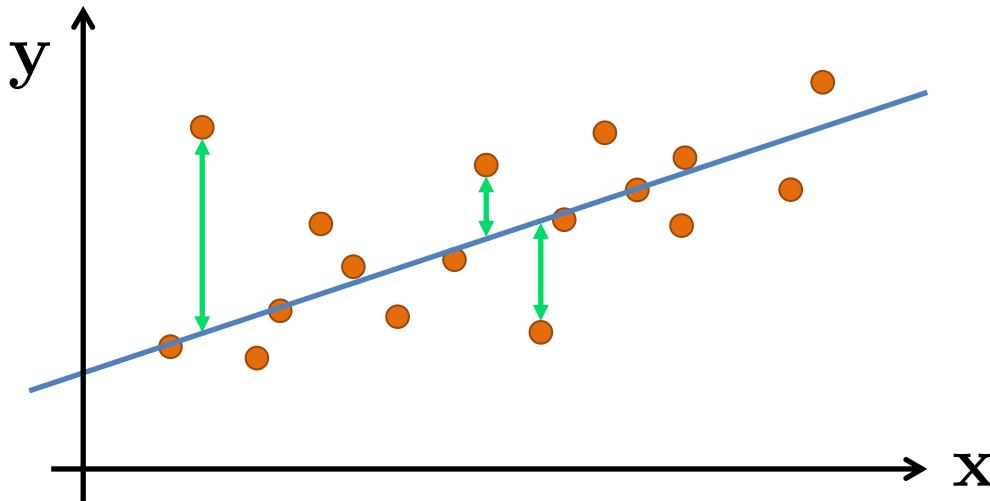
# Linear Regression: Loss Function



# Linear Regression: Loss Function



# Linear Regression: Loss Function



Minimizing

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Objective function

Energy

Cost function

# Optimization: Linear Least Squares

- Linear least squares: an approach to fit a linear model to the data

$$\min_{\theta} J(\theta) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

- Convex problem, there exists a closed-form solution that is unique.

# Optimization: Linear Least Squares

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i \boldsymbol{\theta} - y_i)^2$$



$n$  training samples



The estimation comes  
from the linear model

# Optimization: Linear Least Squares

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i \boldsymbol{\theta} - y_i)^2$$

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Matrix notation

$n$  training samples,  
each input vector has  
size  $d$

$n$  labels

# Optimization: Linear Least Squares

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i \boldsymbol{\theta} - y_i)^2$$

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) \quad \text{Matrix notation}$$

More on matrix notation in the next exercise session

# Optimization: Linear Least Squares

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}_i \boldsymbol{\theta} - y_i)^2$$

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0$$

Convex

Optimum



# Optimization

$$\frac{\partial J(\theta)}{\partial \theta} = 2\mathbf{X}^T \mathbf{X} \boldsymbol{\theta} - 2\mathbf{X}^T \mathbf{y} = 0$$

Details in the  
exercise  
session!

$$\boldsymbol{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

We have found  
an analytical  
solution to a  
convex problem

Inputs: Outside  
temperature,  
number of people,  
...

True output:  
Temperature of  
the building

# Is this the best Estimate?

- Least squares estimate

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

# Maximum Likelihood

# Maximum Likelihood Estimate

$$p_{data}(\mathbf{y}|\mathbf{X})$$

True underlying distribution



$$p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

Parametric family of distributions



Controlled by parameter(s)

# Maximum Likelihood Estimate

- A method of estimating the parameters of a statistical model given observations,

$$p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$


Observations from  $p_{data}(\mathbf{y}|\mathbf{X})$

# Maximum Likelihood Estimate

- A method of estimating the parameters of a statistical model given observations, by finding the parameter values that **maximize the likelihood** of making the observations given the parameters.

$$\boldsymbol{\theta}_{ML} = \arg \max_{\boldsymbol{\theta}} p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

# Maximum Likelihood Estimate

- MLE assumes that the training samples are independent and generated by the same probability distribution

$$p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^n p_{model}(y_i|\mathbf{x}_i, \boldsymbol{\theta})$$



"i.i.d." assumption

# Maximum Likelihood Estimate

$$\theta_{ML} = \arg \max_{\theta} \prod_{i=1}^n p_{model}(y_i | \mathbf{x}_i, \theta)$$

$$\theta_{ML} = \arg \max_{\theta} \sum_{i=1}^n \log p_{model}(y_i | \mathbf{x}_i, \theta)$$

Logarithmic property  $\log ab = \log a + \log b$

# Back to Linear Regression

$$\theta_{ML} = \arg \max_{\theta} \sum_{i=1}^n \log p_{model}(y_i | \mathbf{x}_i, \theta)$$

What shape does our probability distribution have?

# Back to Linear Regression

$$p(y_i | \mathbf{x}_i, \boldsymbol{\theta})$$

What shape does our probability distribution have?

# Back to Linear Regression

$$p(y_i | \mathbf{x}_i, \boldsymbol{\theta})$$

Gaussian / Normal distribution

Assuming  $y_i = \mathcal{N}(\mathbf{x}_i \boldsymbol{\theta}, \sigma^2) = \mathbf{x}_i \boldsymbol{\theta} + \mathcal{N}(0, \sigma^2)$

mean

Gaussian:

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i - \mu)^2}$$

$$y_i \sim \mathcal{N}(\mu, \sigma^2)$$

# Back to Linear Regression

$$p(y_i | \mathbf{x}_i, \boldsymbol{\theta}) = ?$$

Assuming  $y_i = \mathcal{N}(\mathbf{x}_i \boldsymbol{\theta}, \sigma^2) = \mathbf{x}_i \boldsymbol{\theta} + \mathcal{N}(0, \sigma^2)$

Gaussian:

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i - \mu)^2}$$

$$y_i \sim \mathcal{N}(\mu, \sigma^2)$$

# Back to Linear Regression

$$p(y_i | \mathbf{x}_i, \boldsymbol{\theta}) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i - \mathbf{x}_i \boldsymbol{\theta})^2}$$

Assuming  $y_i = \mathcal{N}(\mathbf{x}_i \boldsymbol{\theta}, \sigma^2) = \mathbf{x}_i \boldsymbol{\theta} + \mathcal{N}(0, \sigma^2)$

Gaussian:

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i - \mu)^2}$$

$$y_i \sim \mathcal{N}(\mu, \sigma^2)$$

mean

# Back to Linear Regression

$$p(y_i | \mathbf{x}_i, \boldsymbol{\theta}) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i - \mathbf{x}_i \boldsymbol{\theta})^2}$$

Original  
optimization  
problem

$$\boldsymbol{\theta}_{ML} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^n \log p_{model}(y_i | \mathbf{x}_i, \boldsymbol{\theta})$$

# Back to Linear Regression

$$\sum_{i=1}^n \log \left[ (2\pi\sigma^2)^{-\frac{1}{2}} e^{-\frac{1}{2\sigma^2}(y_i - \mathbf{x}_i \boldsymbol{\theta})^2} \right]$$

Canceling  $\log$  and  $e$

$$\sum_{i=1}^n -\frac{1}{2} \log (2\pi\sigma^2) + \sum_{i=1}^n \left( -\frac{1}{2\sigma^2} \right) (y_i - \mathbf{x}_i \boldsymbol{\theta})^2$$

Matrix notation

$$-\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

# Back to Linear Regression

$$\theta_{ML} = \arg \max_{\theta} \sum_{i=1}^n \log p_{model}(y_i | \mathbf{x}_i, \theta)$$

$$-\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$



Details in the  
exercise session!

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0$$

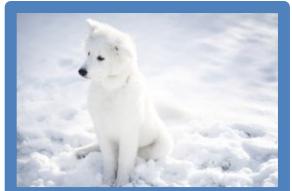
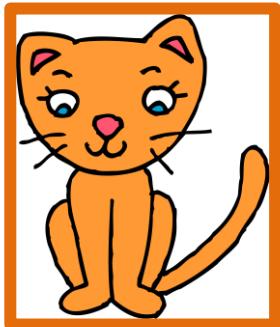
$$\boldsymbol{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

How can we find  
the estimate of  
theta?

# Linear Regression

- Maximum Likelihood Estimate (MLE) corresponds to the Least Squares Estimate (given the assumptions)
- Introduced the concepts of loss function and optimization to obtain the best model for regression

# Image Classification



# Regression vs Classification

- Regression: predict a continuous output value (e.g., temperature of a room)
- Classification: predict a discrete value
  - Binary classification: output is either 0 or 1
  - Multi-class classification: set of N classes



# Logistic Regression



CAT classifier



# Sigmoid for Binary Predictions

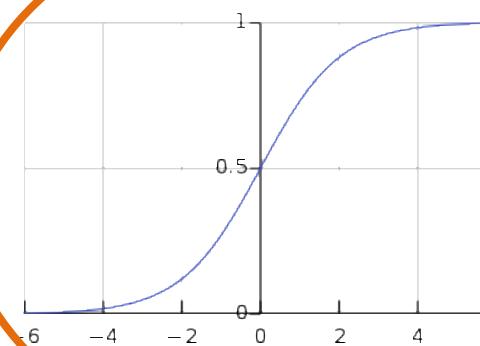
$$x_0 \quad x_1 \quad x_2$$

$$\theta_0$$

$$\theta_1$$

$$\theta_2$$

$$\Sigma$$



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

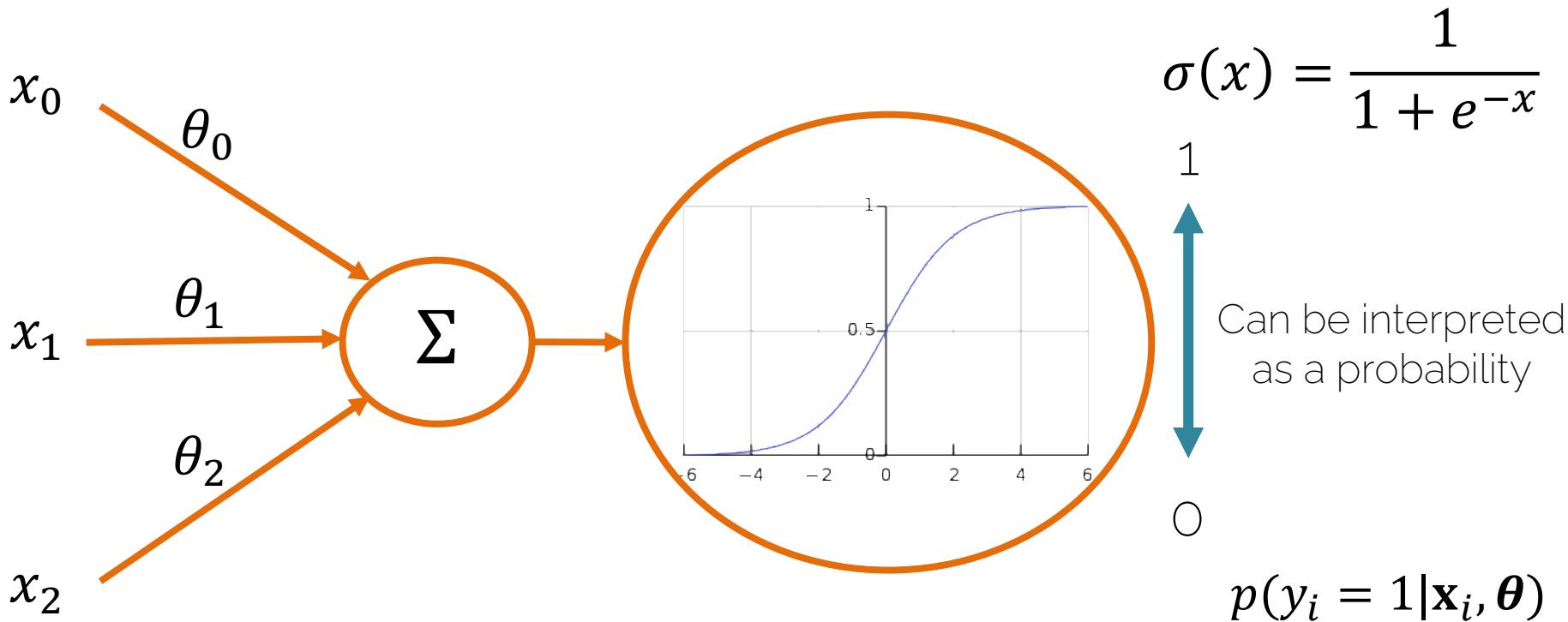
1

0

Can be interpreted  
as a probability

$$p(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta})$$

# Spoiler Alert: 1-Layer Neural Network



# Logistic Regression

- Probability of a binary output

$$\hat{\mathbf{y}} = p(\mathbf{y} = 1 | \mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^n p(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta})$$

The prediction of  
our sigmoid

$$\hat{y}_i = \sigma(\mathbf{x}_i \boldsymbol{\theta})$$

# Logistic Regression

- Probability of a binary output

$$\hat{\mathbf{y}} = p(\mathbf{y} = 1 | \mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^n p(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta})$$

Bernoulli trial

Model for coins

$$p(z|\phi) = \phi^z(1-\phi)^{1-z} = \begin{cases} \phi & , \text{ if } z = 1 \\ 1 - \phi & , \text{ if } z = 0 \end{cases}$$

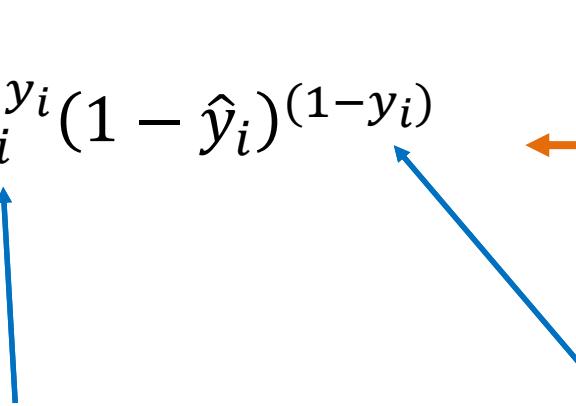
The prediction of our sigmoid

# Logistic Regression

- Probability of a binary output

$$\hat{\mathbf{y}} = p(\mathbf{y} = 1 | \mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^n p(y_i = 1 | \mathbf{x}_i, \boldsymbol{\theta})$$

$$\hat{\mathbf{y}} = \prod_{i=1}^n \hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)}$$

A blue arrow points from the term  $\hat{y}_i^{y_i}$  to the text "Prediction of the Sigmoid: continuous". An orange arrow points from the term  $(1 - \hat{y}_i)^{(1-y_i)}$  to the text "True labels: 0 or 1".

Prediction of the Sigmoid: continuous

True labels: 0 or 1

Model for coins

# Logistic Regression: Loss Function

- Probability of a binary output

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \hat{\mathbf{y}} = \prod_{i=1}^n \hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)}$$

- Maximum Likelihood Estimate

$$\boldsymbol{\theta}_{ML} = \arg \max_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

# Logistic Regression: Loss Function

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \hat{\mathbf{y}} = \prod_{i=1}^n \hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)}$$

$$\sum_{i=1}^n \log (\hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)})$$

$$\sum_{i=1}^n y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

# Logistic Regression: Loss Function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

$$y_i = 1 \longrightarrow \mathcal{L}(\hat{y}_i, 1) = \log \hat{y}_i$$

Maximize!

$$\boldsymbol{\theta}_{ML} = \arg \max_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

# Logistic Regression: Loss Function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

$$y_i = 1 \longrightarrow \mathcal{L}(\hat{y}_i, 1) = \log \hat{y}_i$$

We want  $\log \hat{y}_i$  large; since logarithm is a monotonically increasing function, we also want large  $\hat{y}_i$ .

(1 is the largest value our model's estimate can take!)

# Logistic Regression: Loss Function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

$$y_i = 1 \longrightarrow \mathcal{L}(\hat{y}_i, 1) = \log \hat{y}_i$$

$$y_i = 0 \longrightarrow \mathcal{L}(\hat{y}_i, 0) = \log(1 - \hat{y}_i)$$

We want  $\log(1 - \hat{y}_i)$  large; so we want  $\hat{y}_i$  to be small

(0 is the smallest value our model's estimate can take!)

# Logistic Regression: Loss Function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

Referred to as *binary cross-entropy* loss (BCE)

- Related to the multi-class loss you will see in this course (also called *softmax loss*)

# Logistic Regression: Optimization

- Loss function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

- Cost function

$$C(\theta) = -\frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}_i, y_i)$$

Minimization

$$= -\frac{1}{n} \sum_{i=1}^n y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

$$\hat{y}_i = \sigma(\mathbf{x}_i \boldsymbol{\theta})$$

# Logistic Regression: Optimization

- No closed-form solution
- Make use of an iterative method → gradient descent

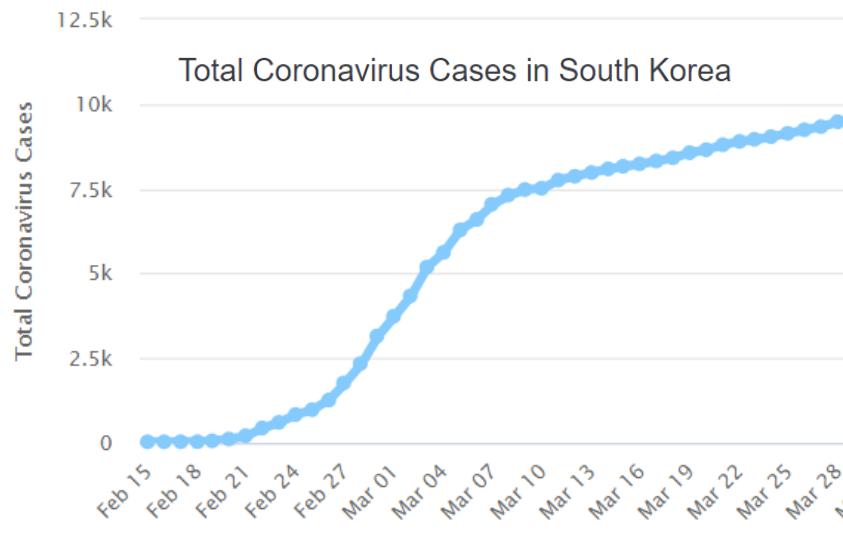
Gradient descent –  
later on!

# Why Machine Learning so Cool

- We can learn from experience
  - > Intelligence, certain ability to infer the future!
- Even linear models are often pretty good for complex phenomena: e.g., weather:
  - Linear combination of day-time, day-year etc. is often pretty good

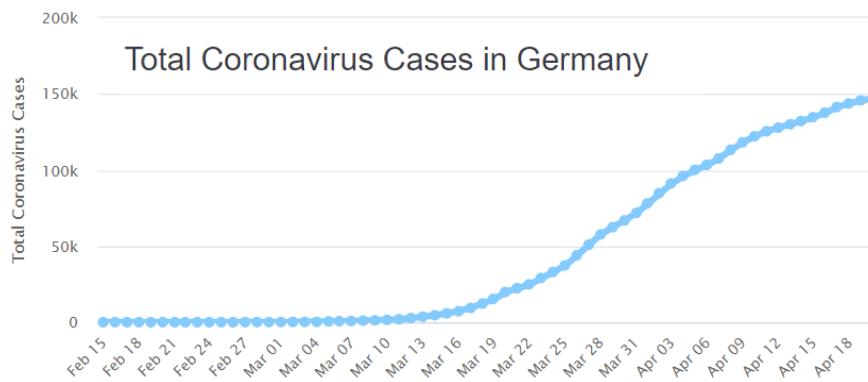
# Many Examples of Logistic Regression

- Coronavirus models behave like logistic regressions
  - Exponential spread at beginning
  - Plateaus when certain portion of pop. is infected/immune



# Many Examples of Logistic Regression

- Coronavirus models behave like logistic regressions
  - Exponential spread at beginning
  - Plateaus when certain portion of pop. is infected/immune



Think about good features:

- Total population
- Population density
- Implementation of Measures
- Reasonable government ☺ ?
- Etc. (many more of course)

# The Model Matters

- Each case requires different models; linear vs logistic
- Many models:
  - #coronavirus\_infections cannot be  $> \#total\_population$
  - Munich housing prizes seem exponential though
    - No hard upper bound -> prizes can always grow!

# Next Lectures

- Next exercise session: Math Recap II
- Next Lecture: Lecture 3:
  - Jumping towards our first Neural Networks and Computational Graphs

# References for further Reading

- Cross validation:
  - <https://medium.com/@zstern/k-fold-cross-validation-explained-5aebagoebb3>
  - <https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6>
- General Machine Learning book:
  - Pattern Recognition and Machine Learning. C. Bishop.

See you next week ☺

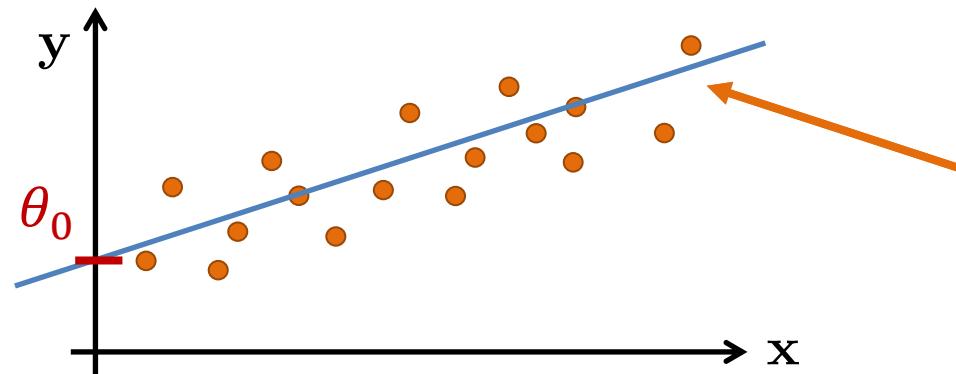
# Introduction to Neural Networks

# Lecture 2 Recap

# Linear Regression

= a supervised learning method to find a linear model of the form

$$\hat{y}_i = \theta_0 + \sum_{j=1}^d x_{ij}\theta_j = \theta_0 + x_{i1}\theta_1 + x_{i2}\theta_2 + \dots + x_{id}\theta_d$$



Goal: find a model that explains a target  $y$  given the input  $x$

# Logistic Regression

- Loss function

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

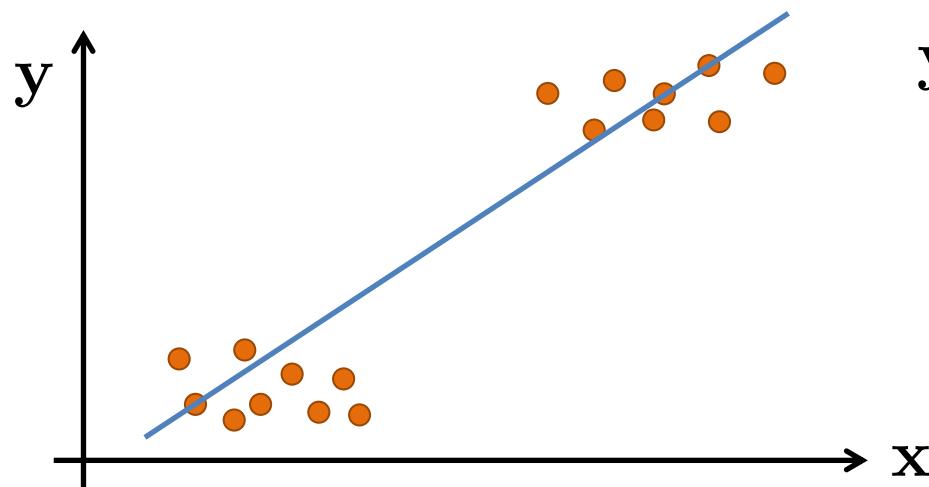
- Cost function

$$\mathcal{C}(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^n (y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log[1 - \hat{y}_i])$$

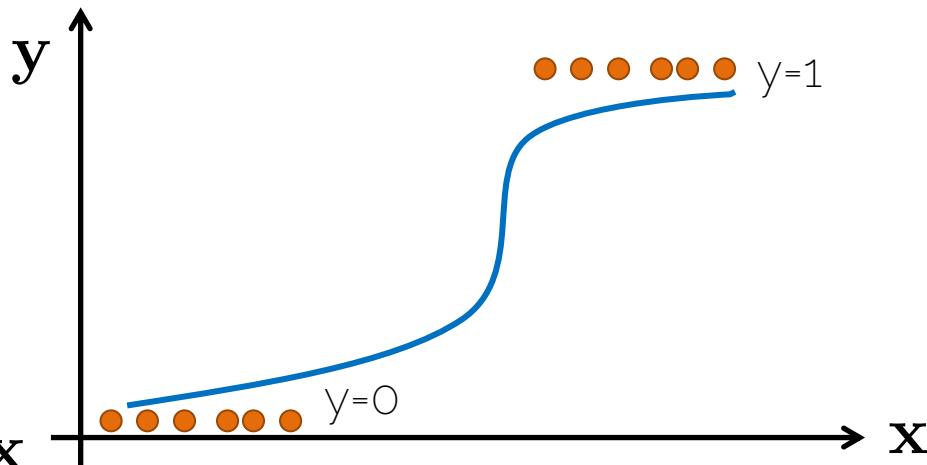
Minimization

$\hat{y}_i = \sigma(x_i \boldsymbol{\theta})$

# Linear vs Logistic Regression

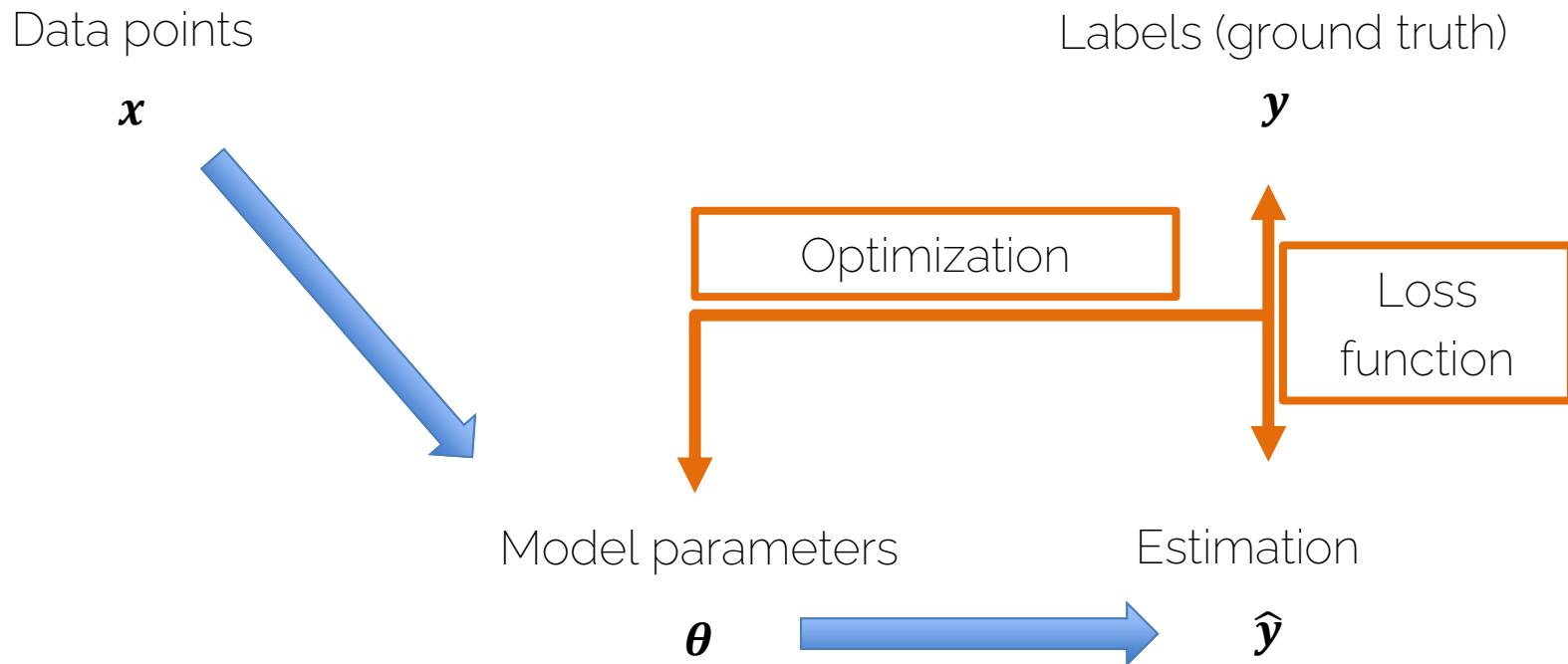


Predictions can exceed the range of the training samples  
→ in the case of classification [0;1] this becomes a real issue



Predictions are guaranteed to be within [0;1]

# How to obtain the Model?



# Linear Score Functions

- Linear score function as seen in linear regression

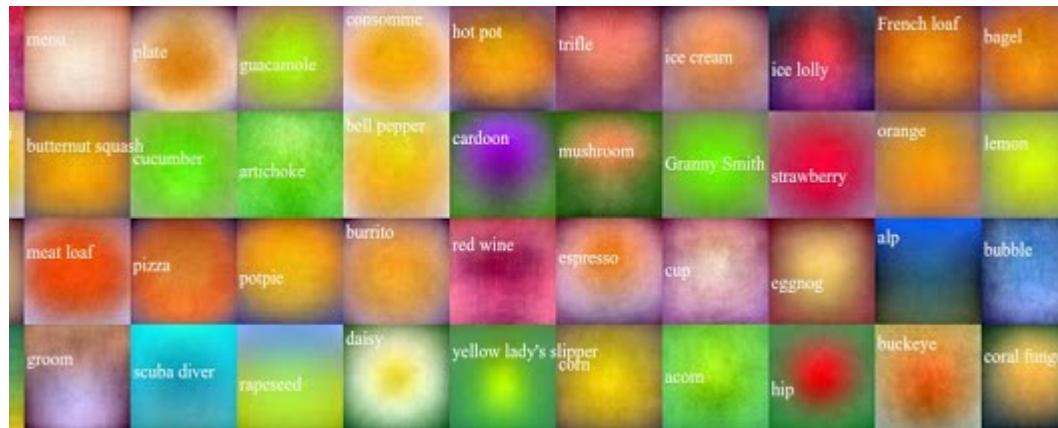
$$\mathbf{f}_i = \sum_j w_{k,j} x_{j,i}$$
$$\mathbf{f} = \mathbf{W} \mathbf{x} \quad (\text{Matrix Notation})$$

# Linear Score Functions on Images

- Linear score function  $f = \mathbf{W}\mathbf{x}$



On CIFAR-10

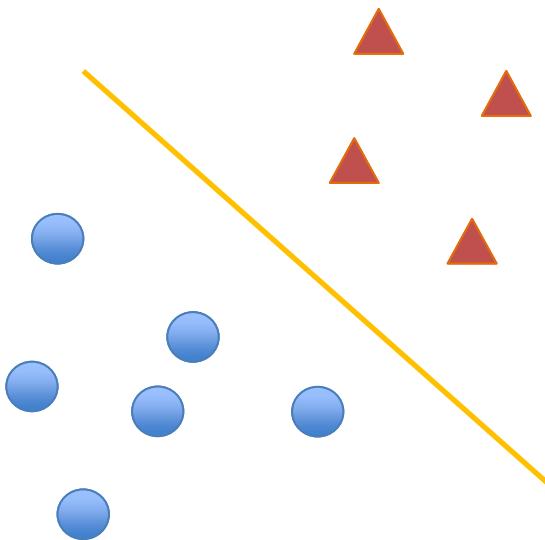


On ImageNet

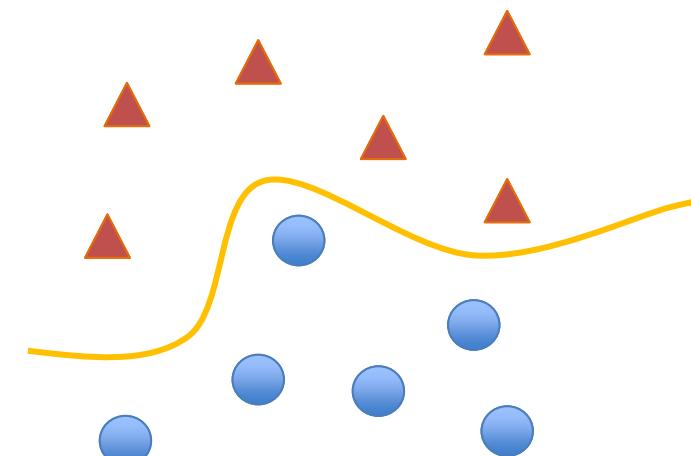
Source: Li/Karpathy/Johnson

# Linear Score Functions?

Logistic Regression



Linear Separation Impossible!



# Linear Score Functions?

- Can we make linear regression better?
  - Multiply with another weight matrix  $\mathbf{W}_2$

$$\begin{aligned}\hat{\mathbf{f}} &= \mathbf{W}_2 \cdot \mathbf{f} \\ \hat{\mathbf{f}} &= \mathbf{W}_2 \cdot \mathbf{W} \cdot \mathbf{x}\end{aligned}$$

- Operation is still linear.

$$\begin{aligned}\widehat{\mathbf{W}} &= \mathbf{W}_2 \cdot \mathbf{W} \\ \hat{\mathbf{f}} &= \widehat{\mathbf{W}} \mathbf{x}\end{aligned}$$

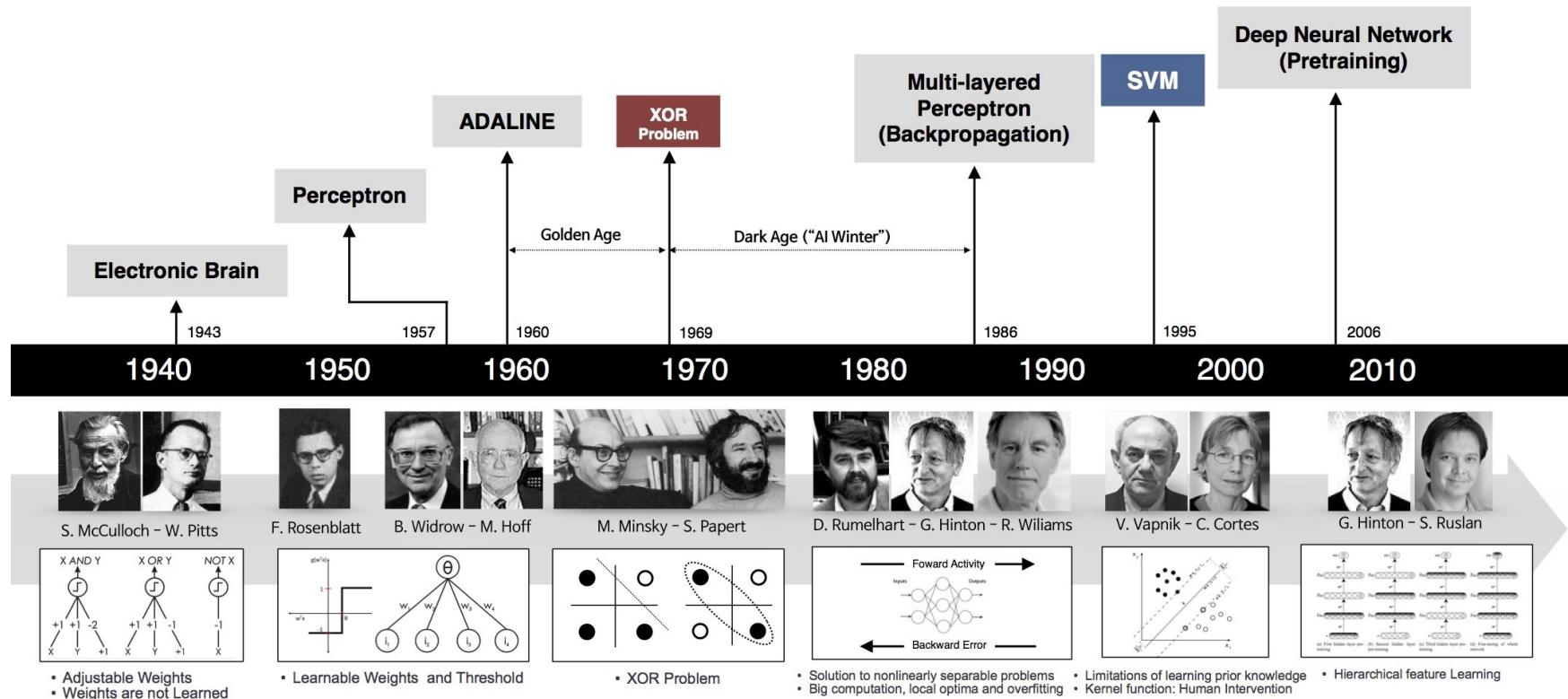
- Solution → add non-linearity!!

# Neural Network

- Linear score function  $f = \mathbf{W}\mathbf{x}$
- Neural network is a nesting of 'functions'
  - 2-layers:  $f = \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x})$
  - 3-layers:  $f = \mathbf{W}_3 \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}))$
  - 4-layers:  $f = \mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x})))$
  - 5-layers:  $f = \mathbf{W}_5 \sigma(\mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}))))$
  - ... up to hundreds of layers

# Introduction to Neural Networks

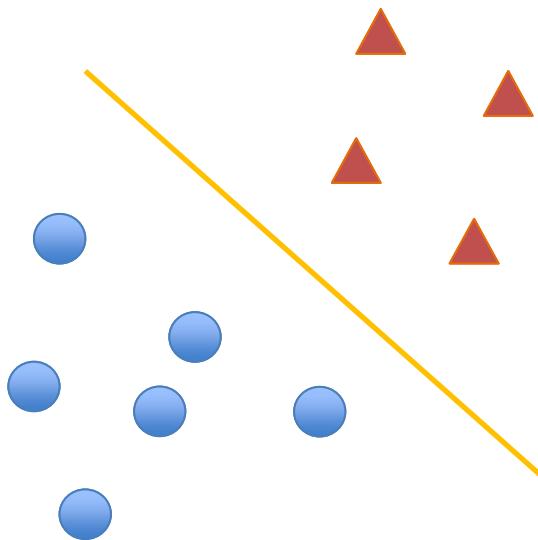
# History of Neural Networks



Source: [http://beamlab.org/deeplearning/2017/02/23/deep\\_learning\\_101\\_part1.html](http://beamlab.org/deeplearning/2017/02/23/deep_learning_101_part1.html)

# Neural Network

Logistic Regression



Neural Networks

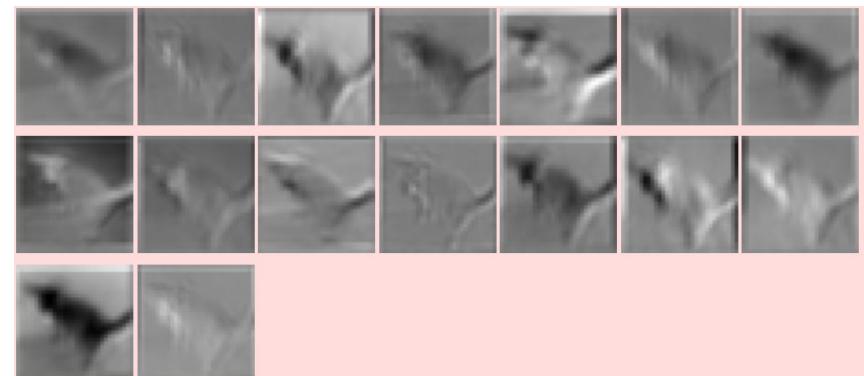


# Neural Network

- Non-linear score function  $f = \dots (\max(0, W_1 x))$



On CIFAR-10

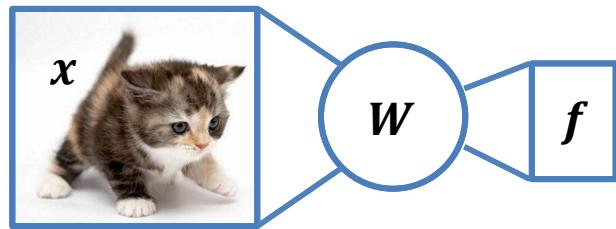


Visualizing activations of the first layer.

Source: ConvNetJS

# Neural Network

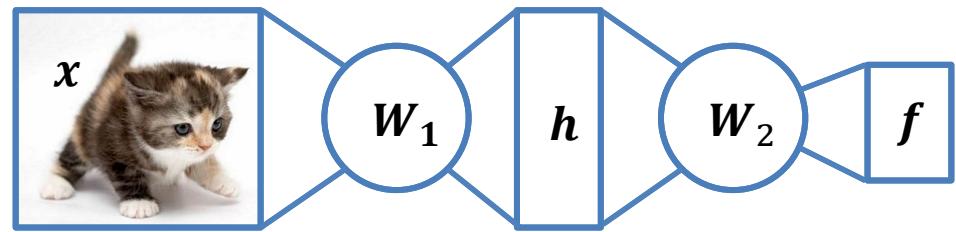
1-layer network:  $f = Wx$



$$128 \times 128 = 16384$$

$$10$$

2-layer network:  $f = W_2 \max(0, W_1 x)$



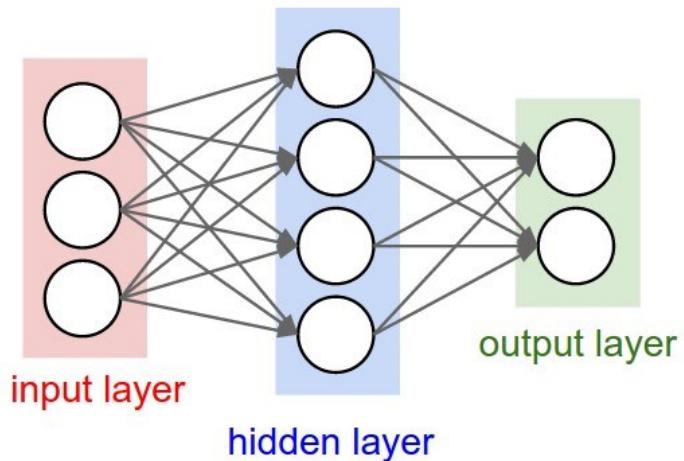
$$128 \times 128 = 16384$$

$$1000$$

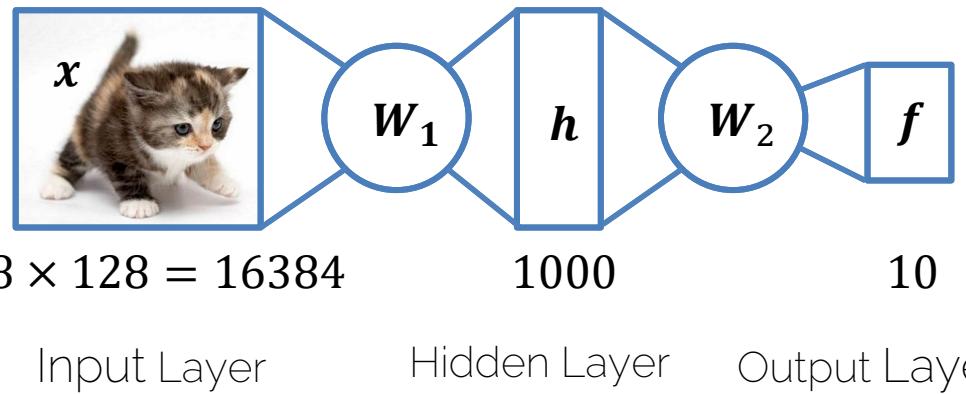
$$10$$

Why is this structure useful?

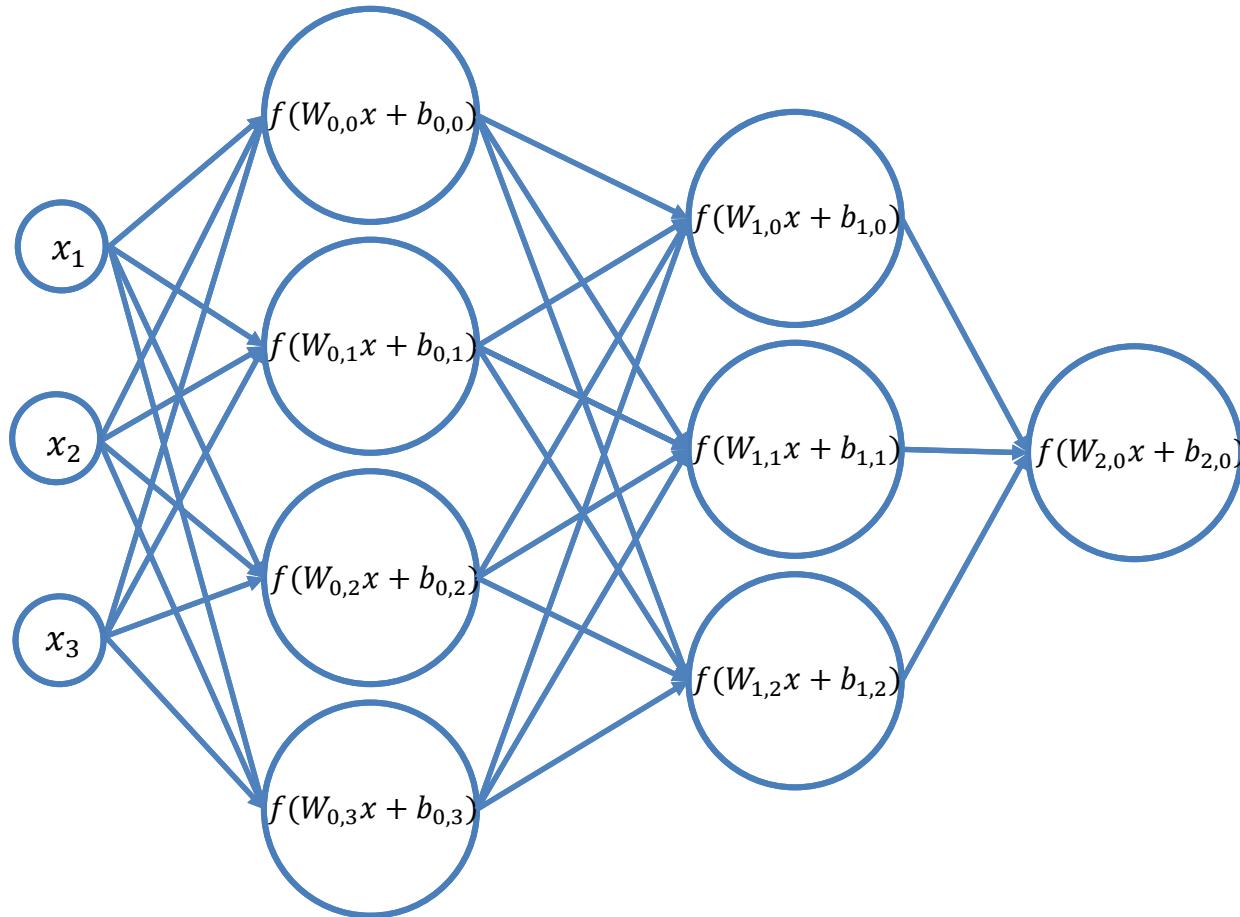
# Neural Network



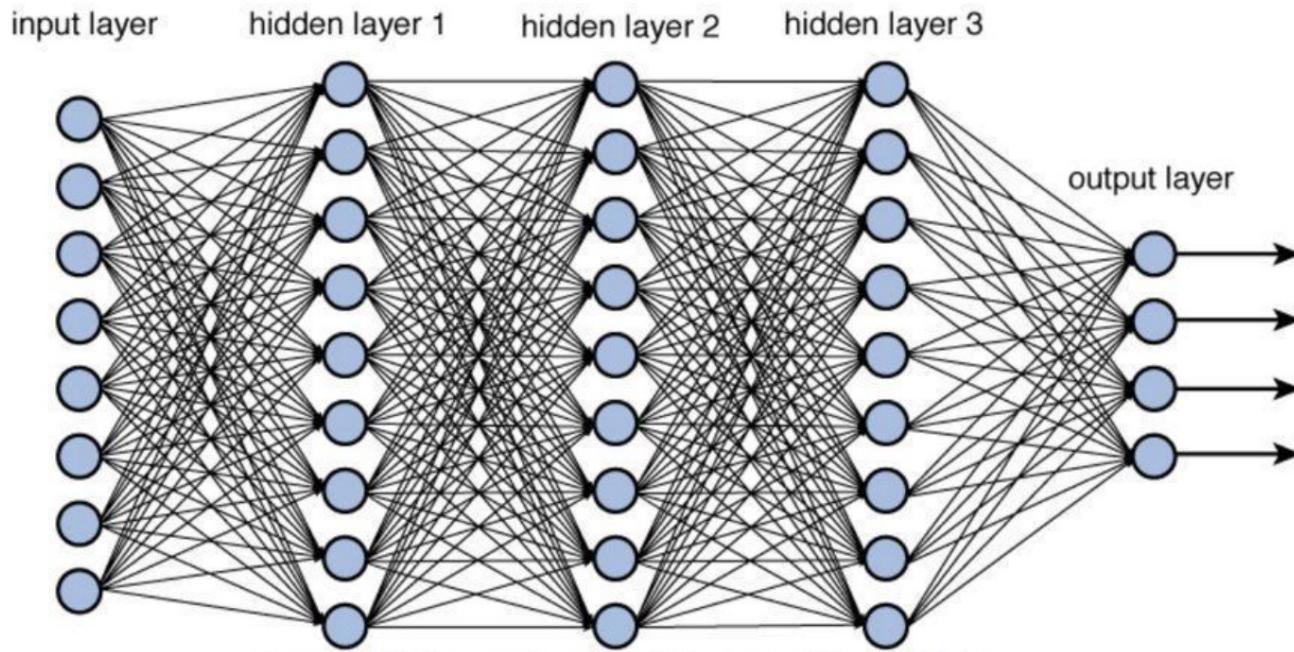
2-layer network:  $f = \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x)$



# Net of Artificial Neurons



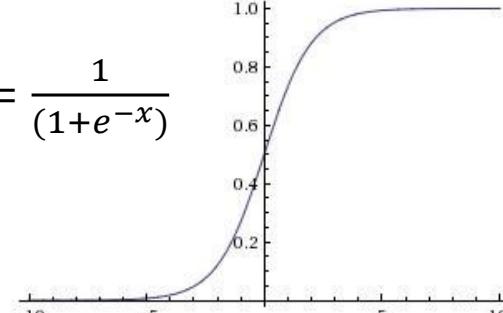
# Neural Network



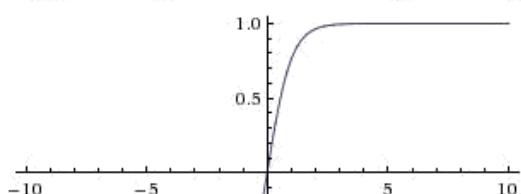
Source: <https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964>

# Activation Functions

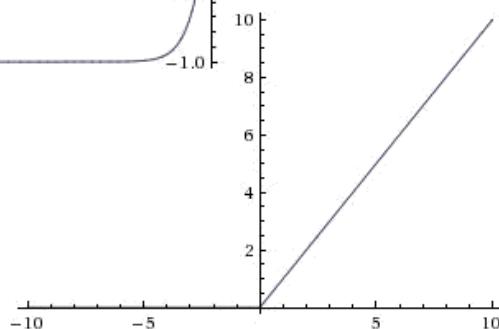
Sigmoid:  $\sigma(x) = \frac{1}{(1+e^{-x})}$



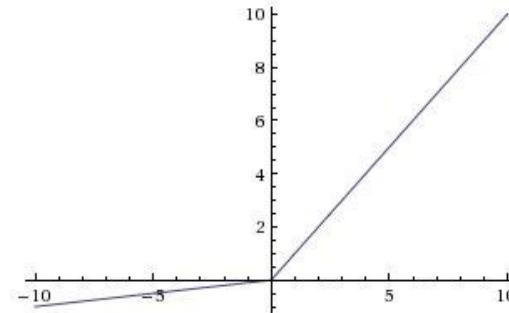
tanh:  $\tanh(x)$



ReLU:  $\max(0, x)$



Leaky ReLU:  $\max(0.1x, x)$



Parametric ReLU:  $\max(\alpha x, x)$

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

ELU f(x) =  $\begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$

# Neural Network

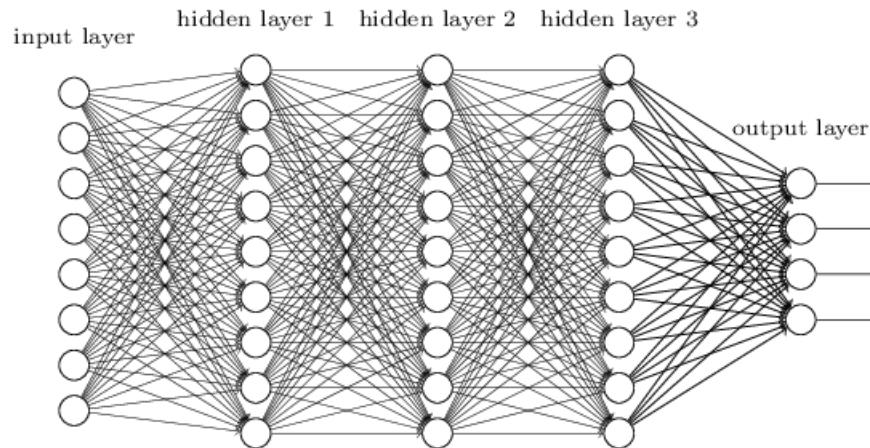
$$f = W_3 \cdot (W_2 \cdot (W_1 \cdot x)))$$

Why activation functions?

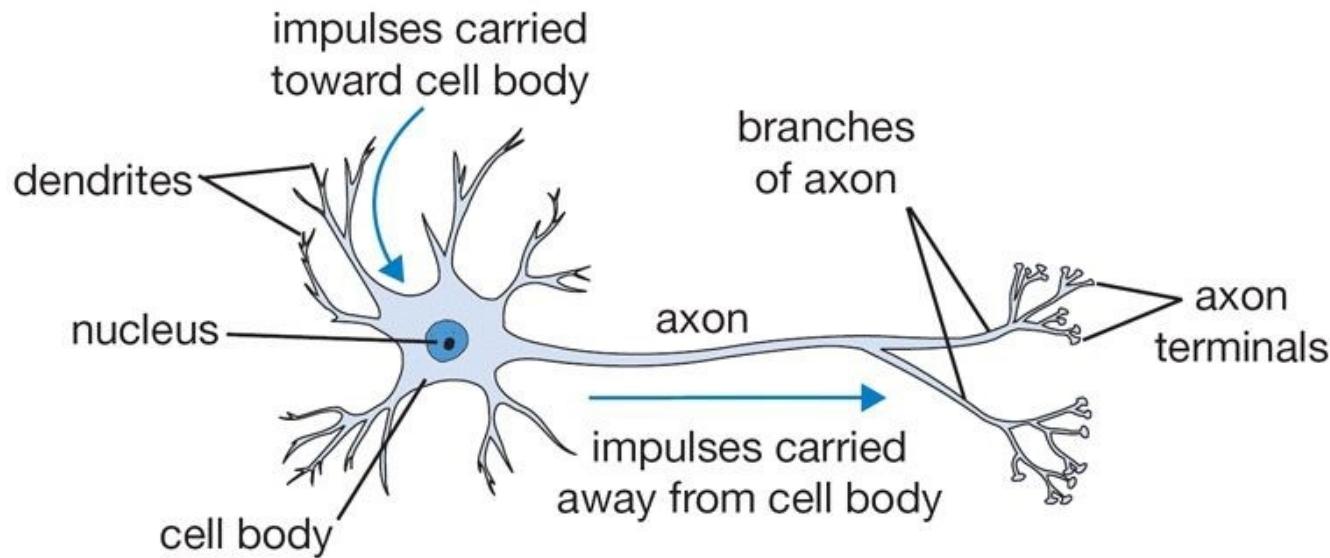
Simply concatenating linear layers would be so much cheaper...

# Neural Network

Why organize a neural network into layers?

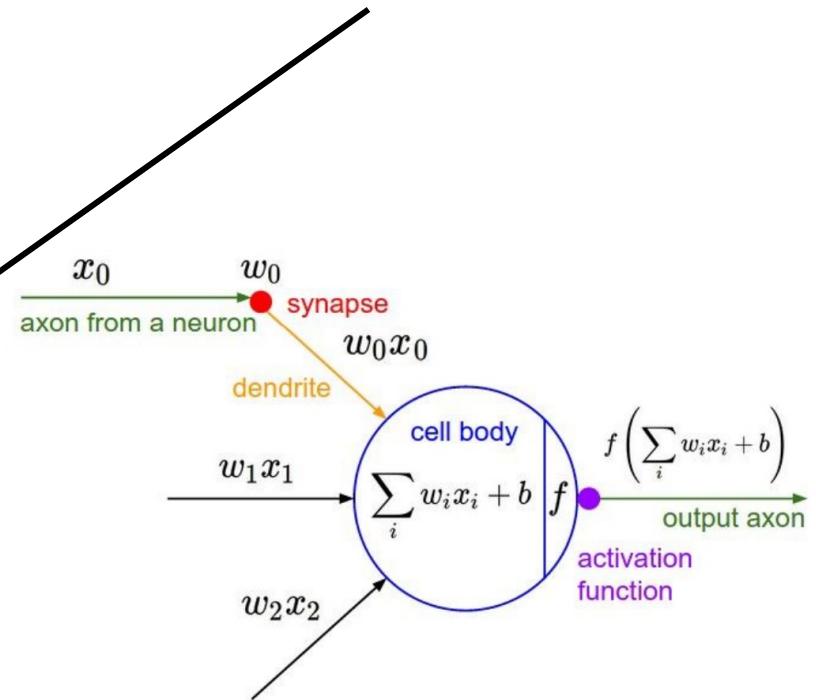
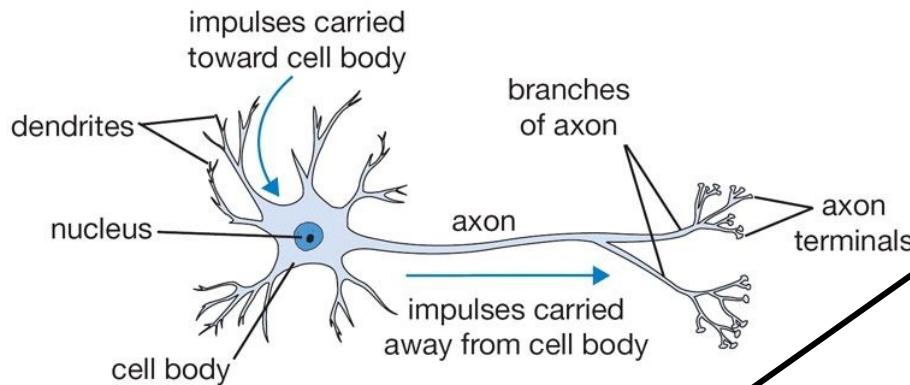


# Biological Neurons



Credit: Stanford CS 231n

# Biological Neurons



Credit: Stanford CS 231n

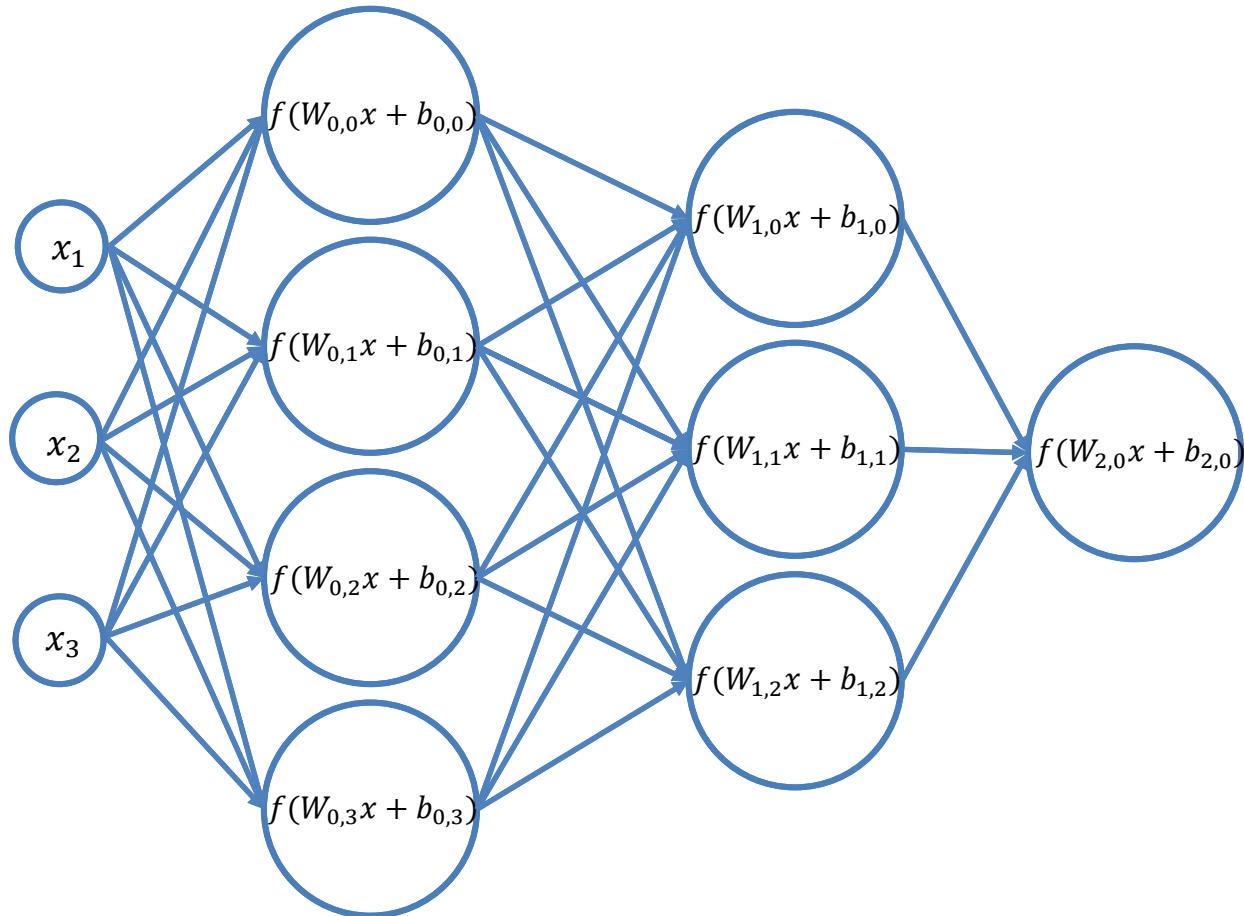
# Artificial Neural Networks vs Brain



Artificial neural networks are **inspired** by the brain,  
but not even close in terms of complexity!

The comparison is great for the media and news articles though... ☺

# Artificial Neural Network



# Neural Network

- Summary
  - Given a dataset with ground truth training pairs  $[x_i; y_i]$ .
  - Find optimal weights and biases  $\mathbf{W}$  using stochastic gradient descent, such that the loss function is minimized
    - Compute gradients with backpropagation (use batch-mode; more later)
    - Iterate many times over training set (SGD; more later)

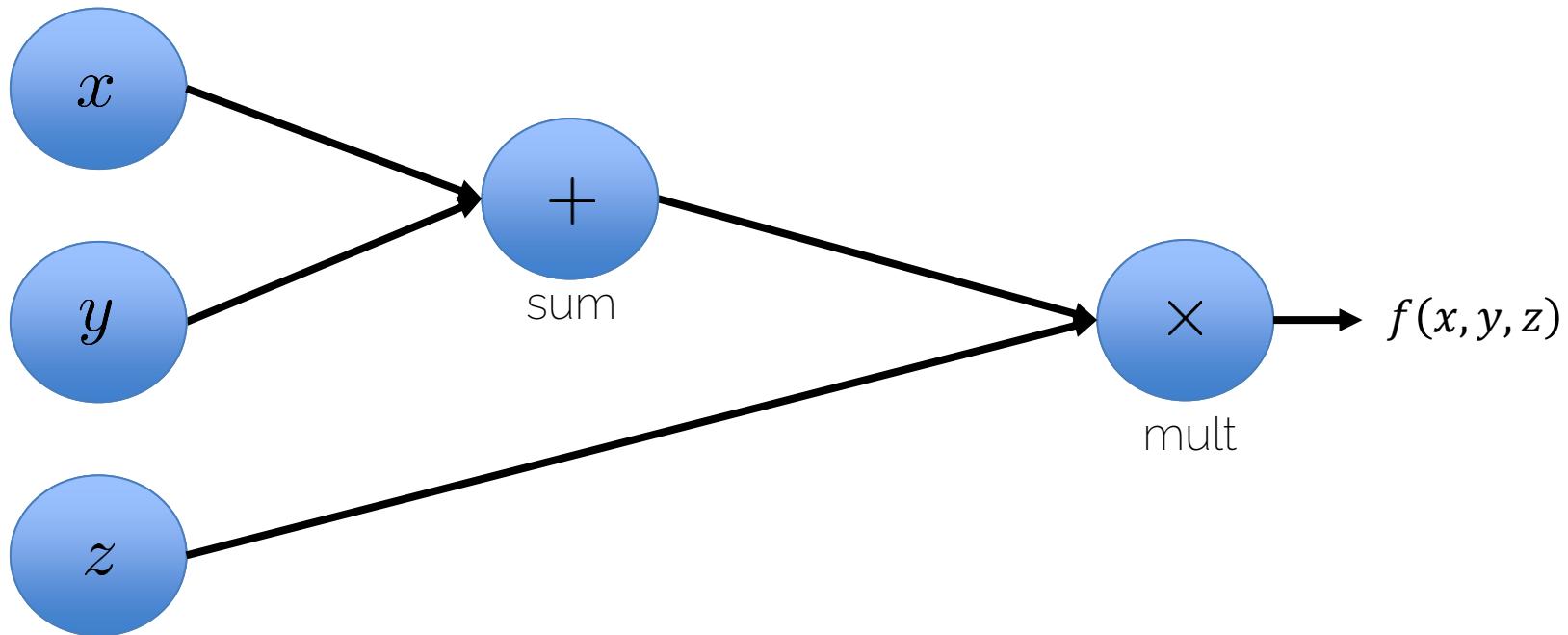
# Computational Graphs

# Computational Graphs

- Directional graph
- Matrix operations are represented as compute nodes.
- Vertex nodes are variables or operators like  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\log()$ ,  $\exp()$  ...
- Directional edges show flow of inputs to vertices

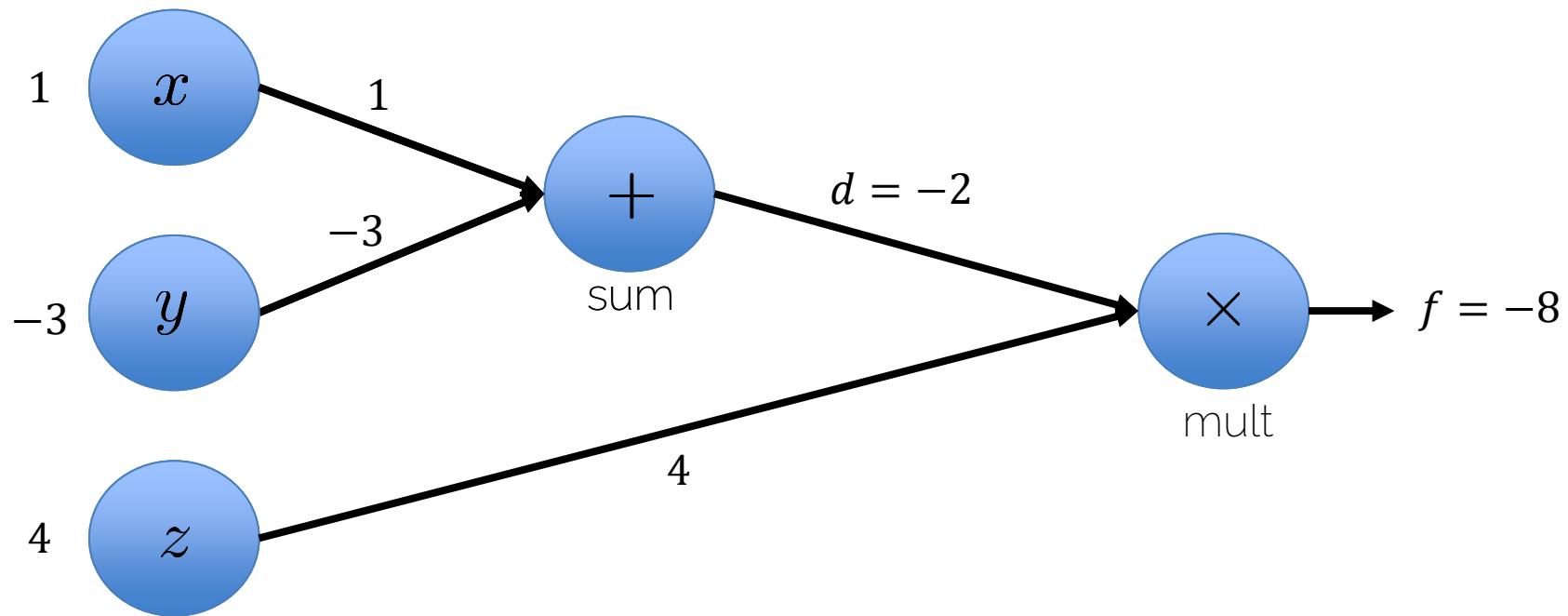
# Computational Graphs

- $f(x, y, z) = (x + y) \cdot z$



# Evaluation: Forward Pass

- $f(x, y, z) = (x + y) \cdot z$  Initialization  $x = 1, y = -3, z = 4$



# Computational Graphs

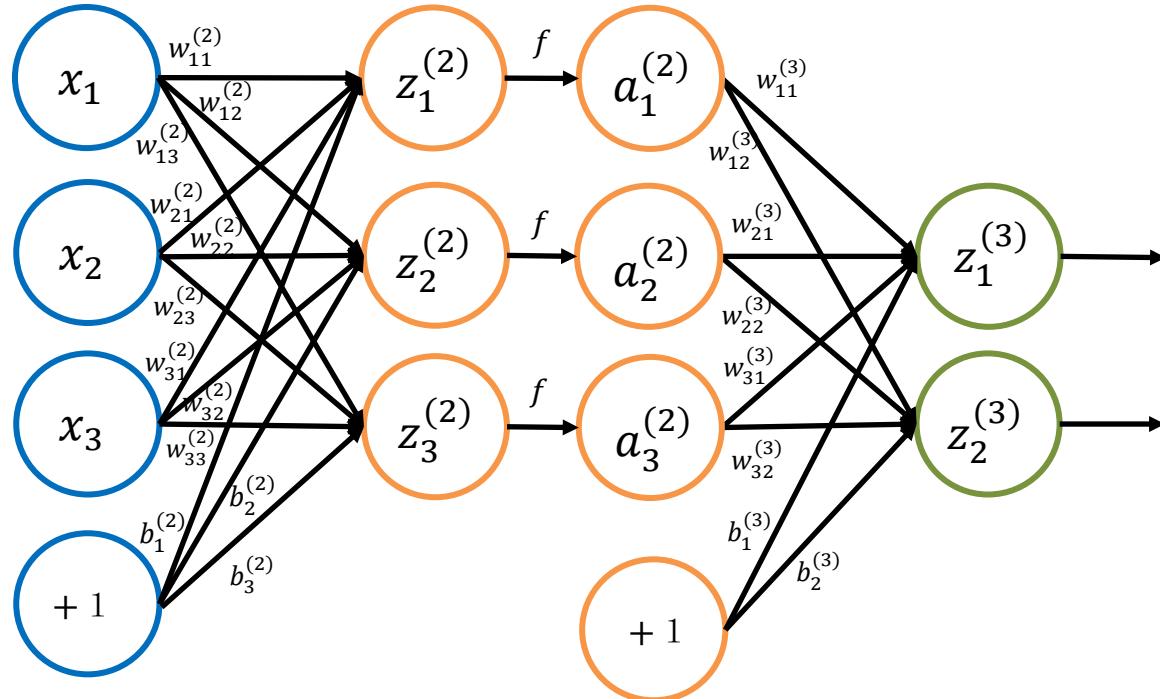
- Why discuss compute graphs?
- Neural networks have complicated architectures  
$$f = \mathbf{W}_5 \sigma(\mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x))))$$
- Lot of matrix operations!
- Represent NN as computational graphs!

# Computational Graphs

A neural network can be represented as a computational graph...

- it has compute nodes (operations)
- it has edges that connect nodes (data flow)
- it is directional
- it can be organized into 'layers'

# Computational Graphs



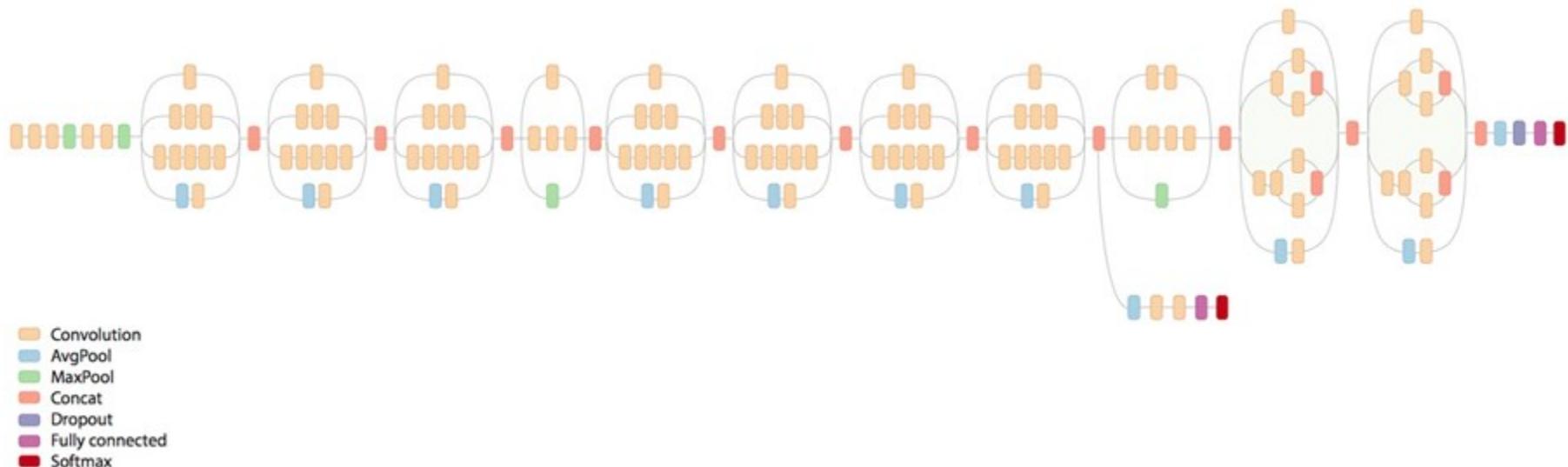
$$z_k^{(2)} = \sum_i x_i w_{ik}^{(2)} + b_k^{(2)}$$

$$a_k^{(2)} = f(z_k^{(2)})$$

$$z_k^{(3)} = \sum_i a_i^{(2)} w_{ik}^{(3)} + b_k^{(3)}$$

# Computational Graphs

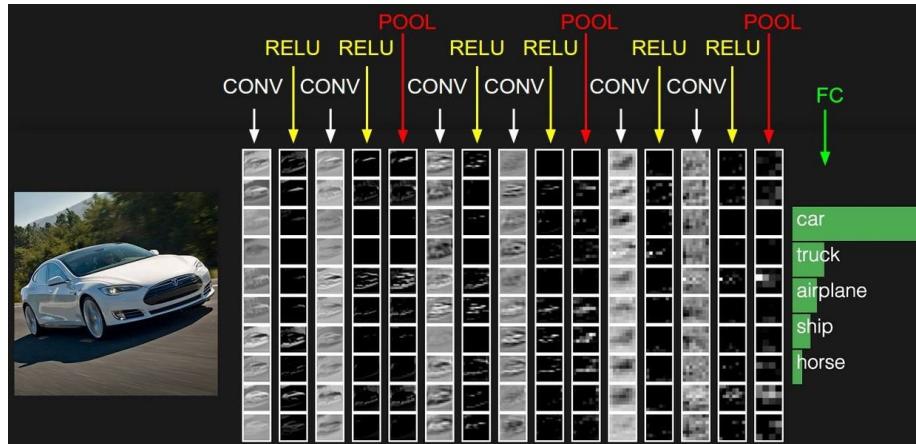
- From a set of neurons to a Structured Compute Pipeline



[Szegedy et al., CVPR'15] Going Deeper with Convolutions

# Computational Graphs

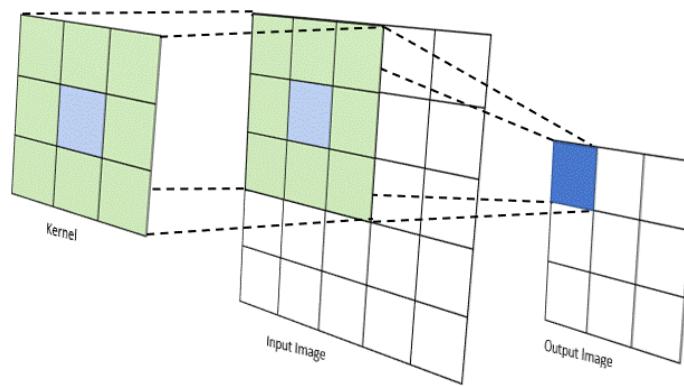
- The computation of Neural Network has further meanings:
  - The multiplication of  $\mathbf{W}_i$  and  $\mathbf{x}$ : encode input information
  - The activation function: select the key features



Source: <https://www.zybuluo.com/liuhui0803/note/981434>

# Computational Graphs

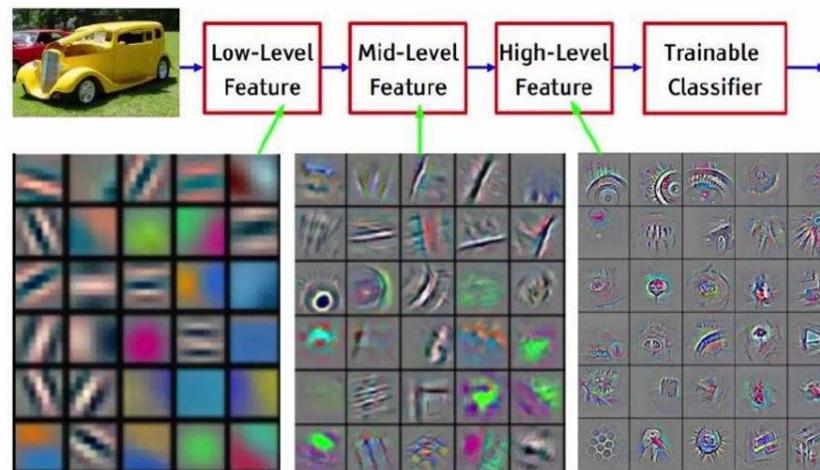
- The computations of Neural Networks have further meanings:
  - The convolutional layers: extract useful features with shared weights



Source: [https://medium.com/@timothy\\_terati/image-convolution-filtering-a54dce7c786b](https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b)

# Computational Graphs

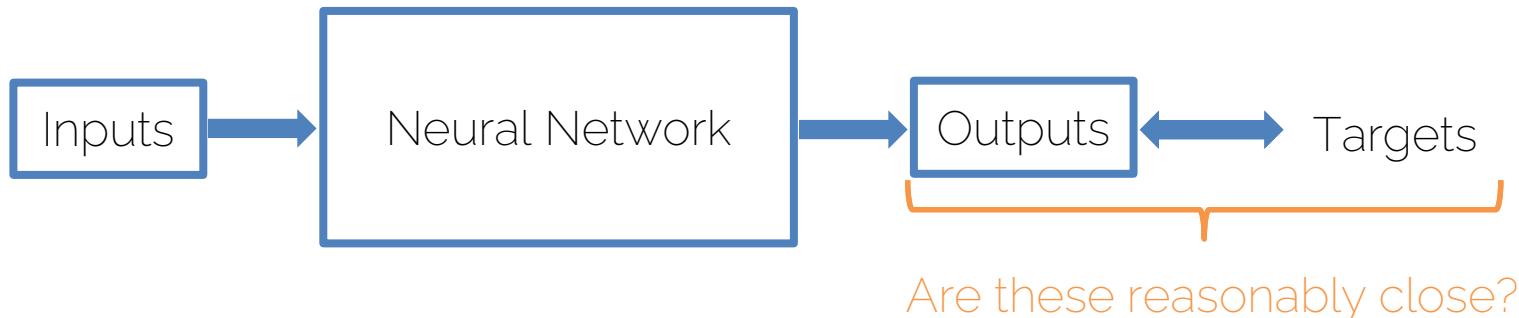
- The computations of Neural Networks have further meanings:
  - The convolutional layers: extract useful features with shared weights



Source: <https://www.zybuluo.com/liuhui0803/note/981434>

# Loss Functions

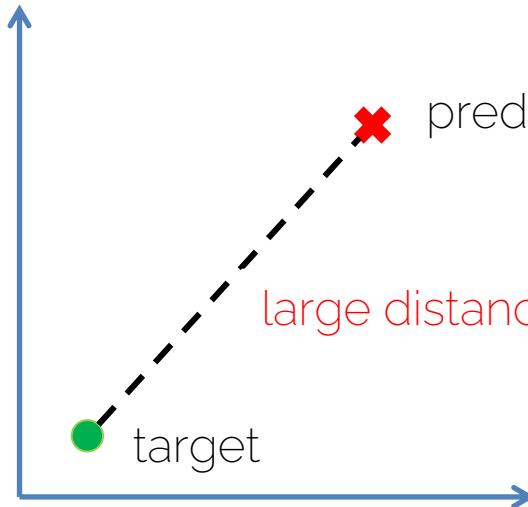
# What's Next?



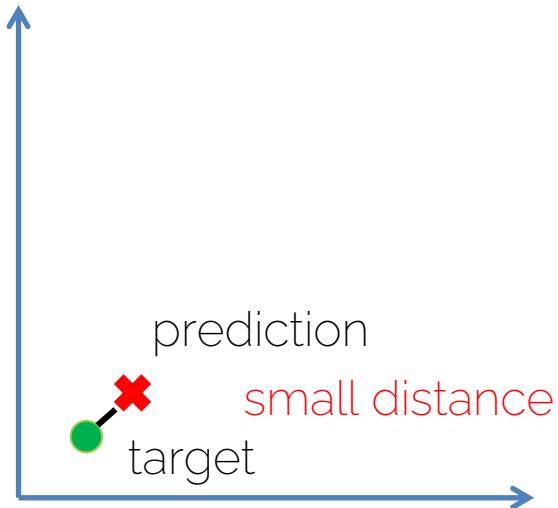
We need a way to describe how close the network's outputs (= predictions) are to the targets!

# What's Next?

Idea: calculate a 'distance' between prediction and target!



bad prediction



good prediction

# Loss Functions

- A function to **measure the goodness of the predictions** (or equivalently, the network's performance)

Intuitively, ...

- a large loss indicates bad predictions/performance  
(→ performance needs to be improved by training the model)
- the choice of the loss function depends on the concrete problem or the distribution of the target variable

# Regression Loss

- L1 Loss:

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n ||\mathbf{y}_i - \hat{\mathbf{y}}_i||_1$$

- MSE Loss:

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n ||\mathbf{y}_i - \hat{\mathbf{y}}_i||_2^2$$

# Binary Cross Entropy

- Loss function for binary (yes/no) classification

$$L(y, \hat{y}; \theta) = -\frac{1}{n} \sum_{i=1}^n (y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log[1 - \hat{y}_i])$$



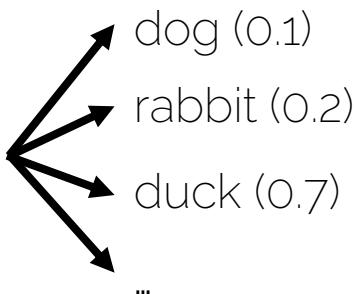
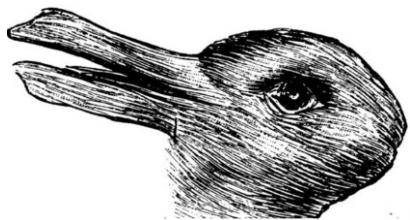
Yes! (0.8)  
No! (0.2)

The network predicts  
the probability of the input  
belonging to the "yes" class!

# Cross Entropy

= loss function for multi-class classification

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = - \sum_{i=1}^n \sum_{k=1}^k (y_{ik} \cdot \log \hat{y}_{ik})$$

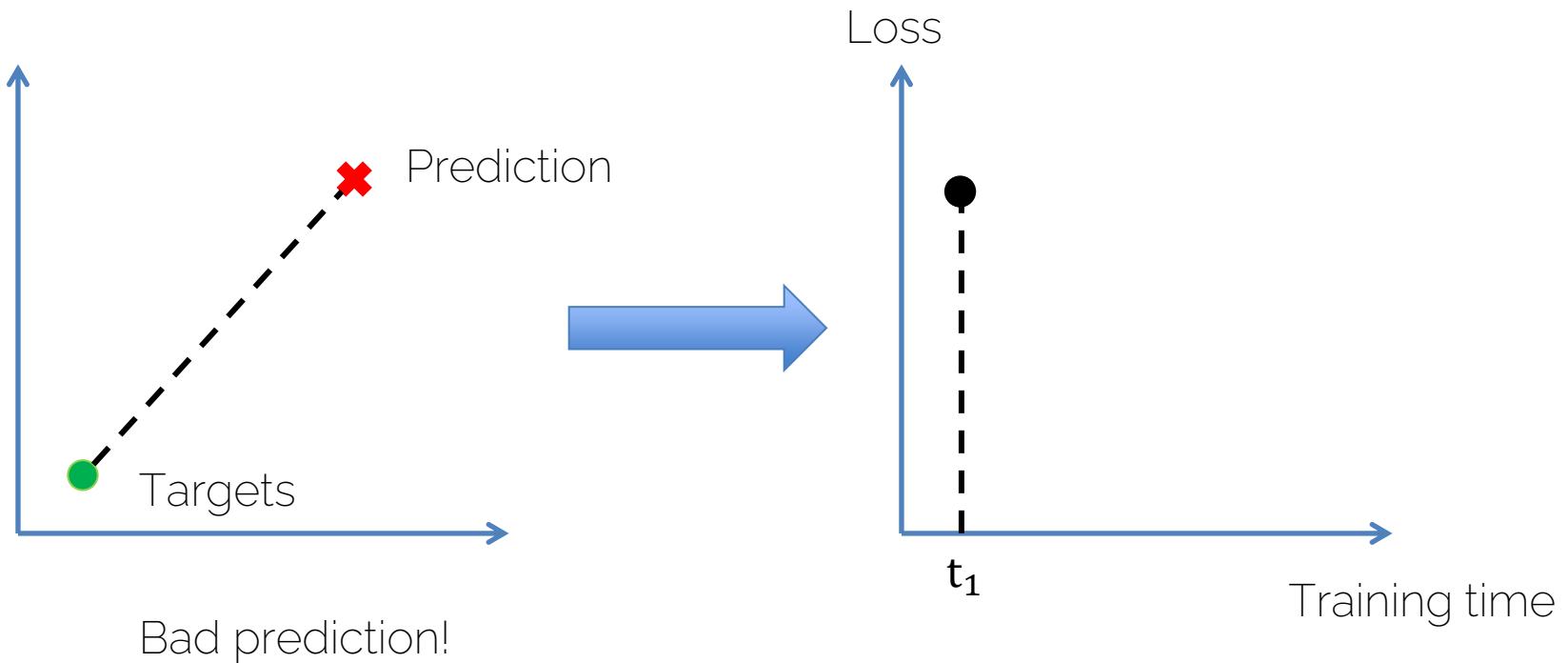


This generalizes the binary case from the slide before!

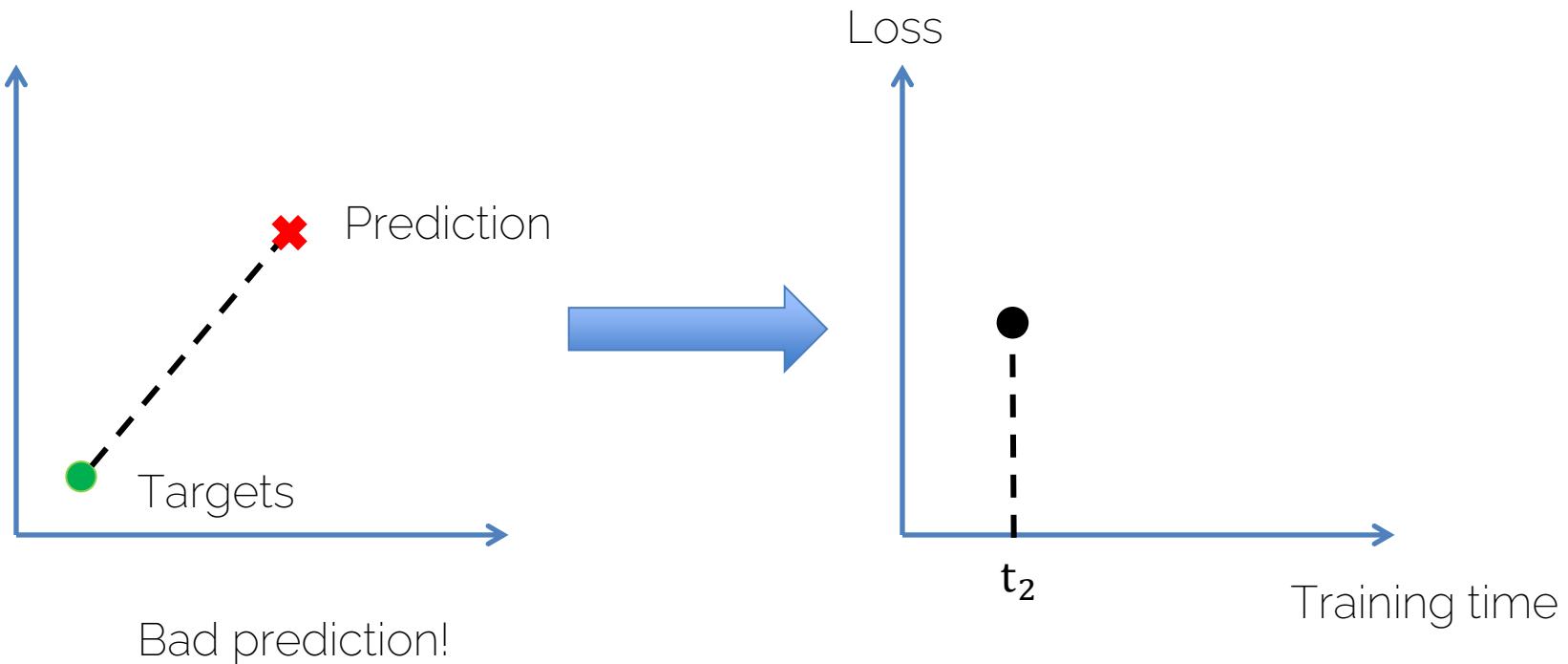
# More General Case

- Ground truth:  $\mathbf{y}$
- Prediction:  $\hat{\mathbf{y}}$
- Loss function:  $L(\mathbf{y}, \hat{\mathbf{y}})$
- Motivation:
  - minimize the loss  $\Leftrightarrow$  find better predictions
  - predictions are generated by the NN
  - find better predictions  $\Leftrightarrow$  find better NN

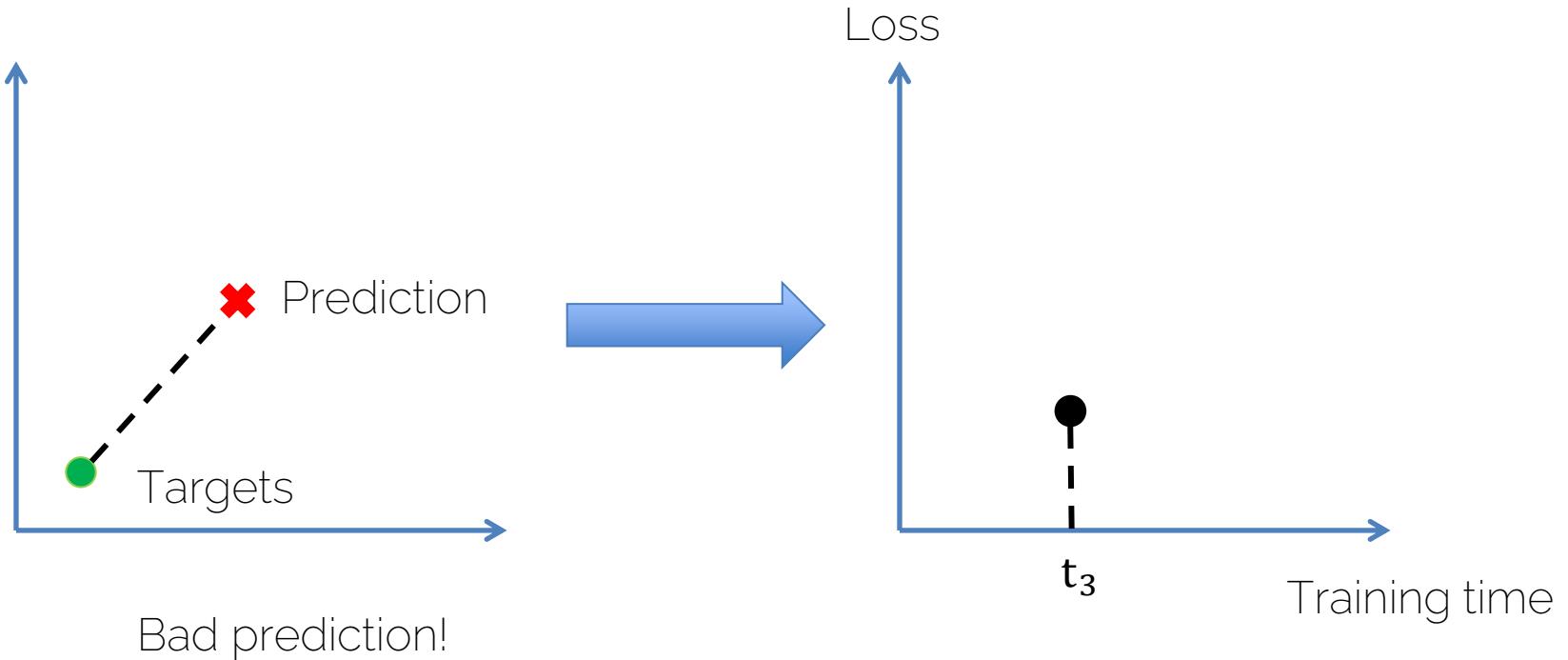
# Initially



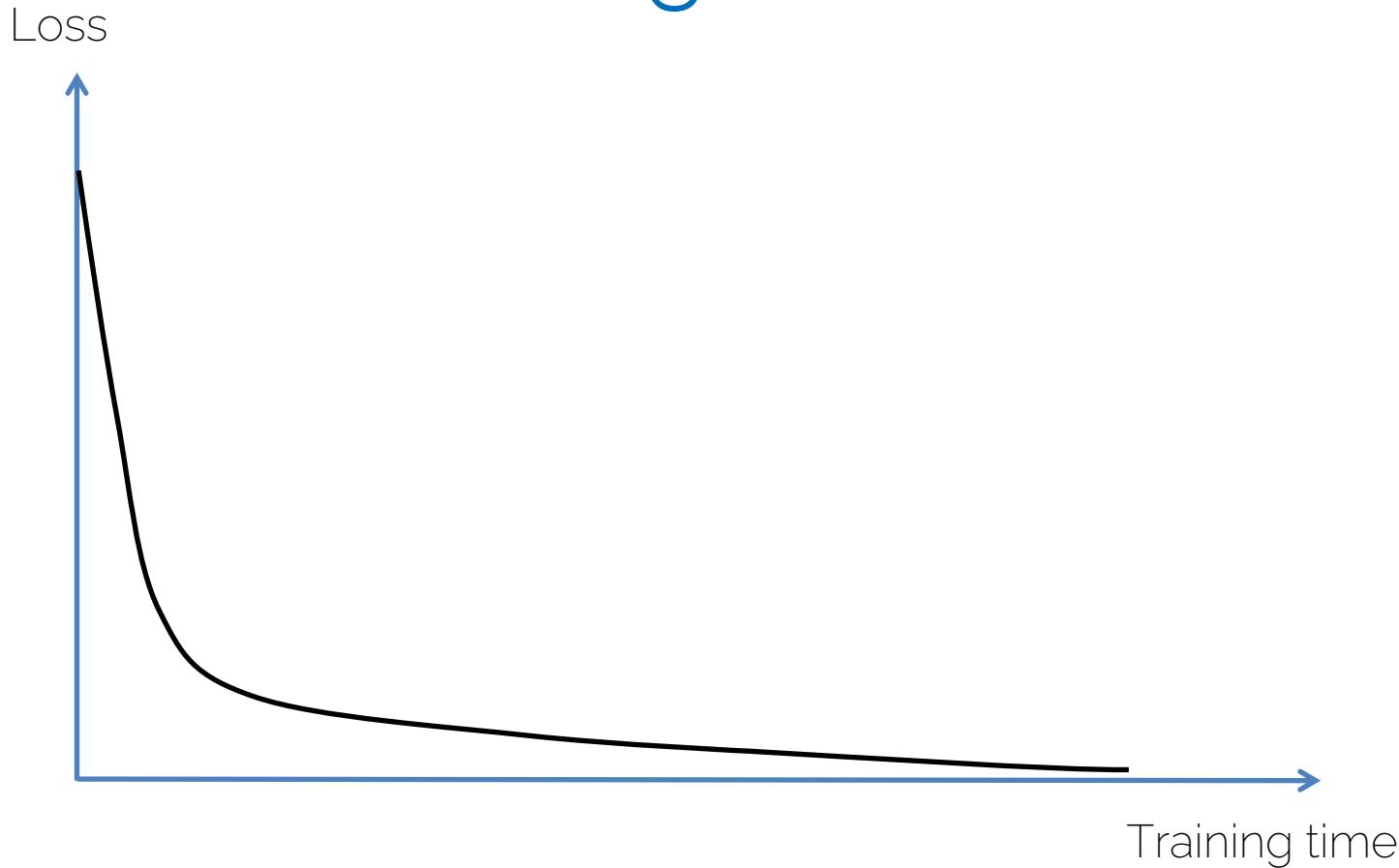
# During Training...



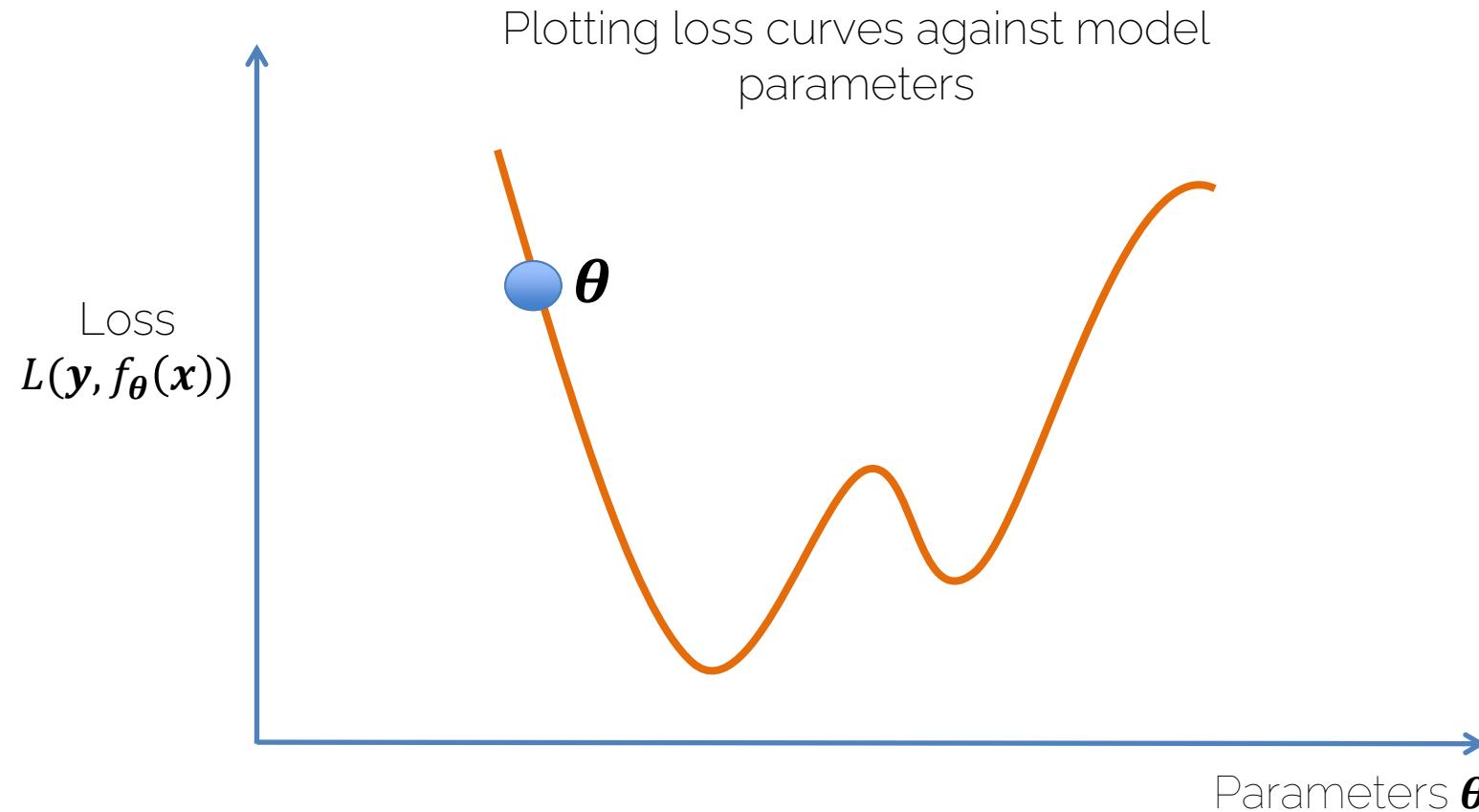
# During Training...



# Training Curve



# How to Find a Better NN?



# How to Find a Better NN?

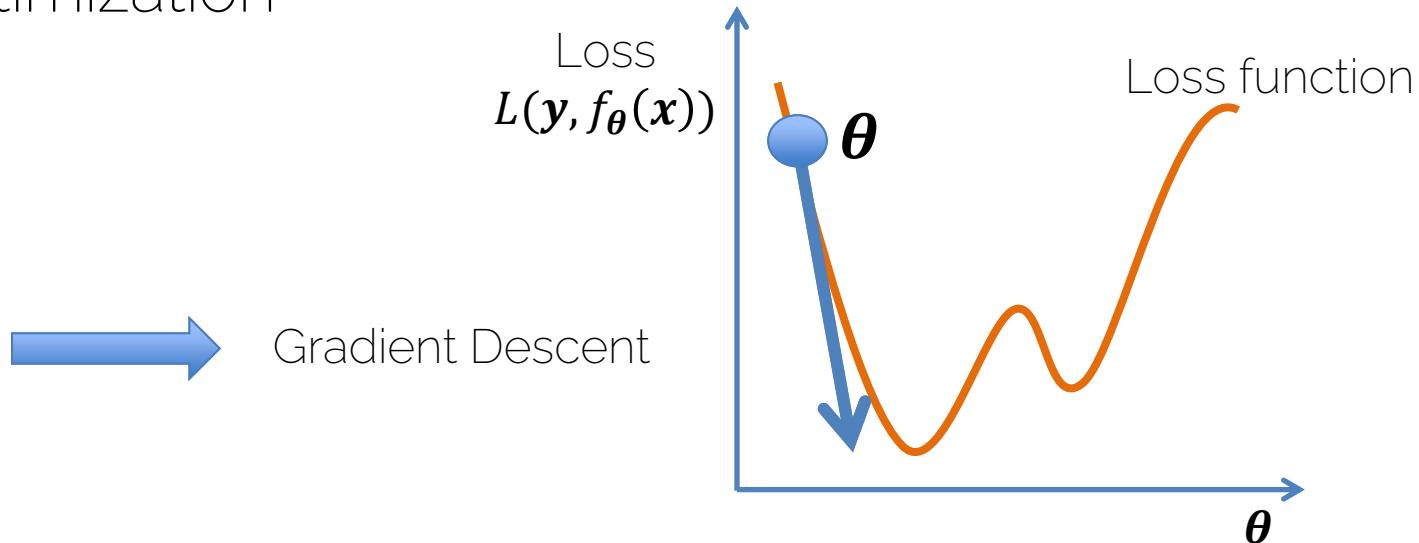
- Loss function:  $L(\mathbf{y}, \hat{\mathbf{y}}) = L(\mathbf{y}, f_{\boldsymbol{\theta}}(\mathbf{x}))$
- Neural Network:  $f_{\boldsymbol{\theta}}(\mathbf{x})$
- Goal:
  - minimize the loss w. r. t.  $\boldsymbol{\theta}$



Optimization! We train compute graphs  
with some optimization techniques!

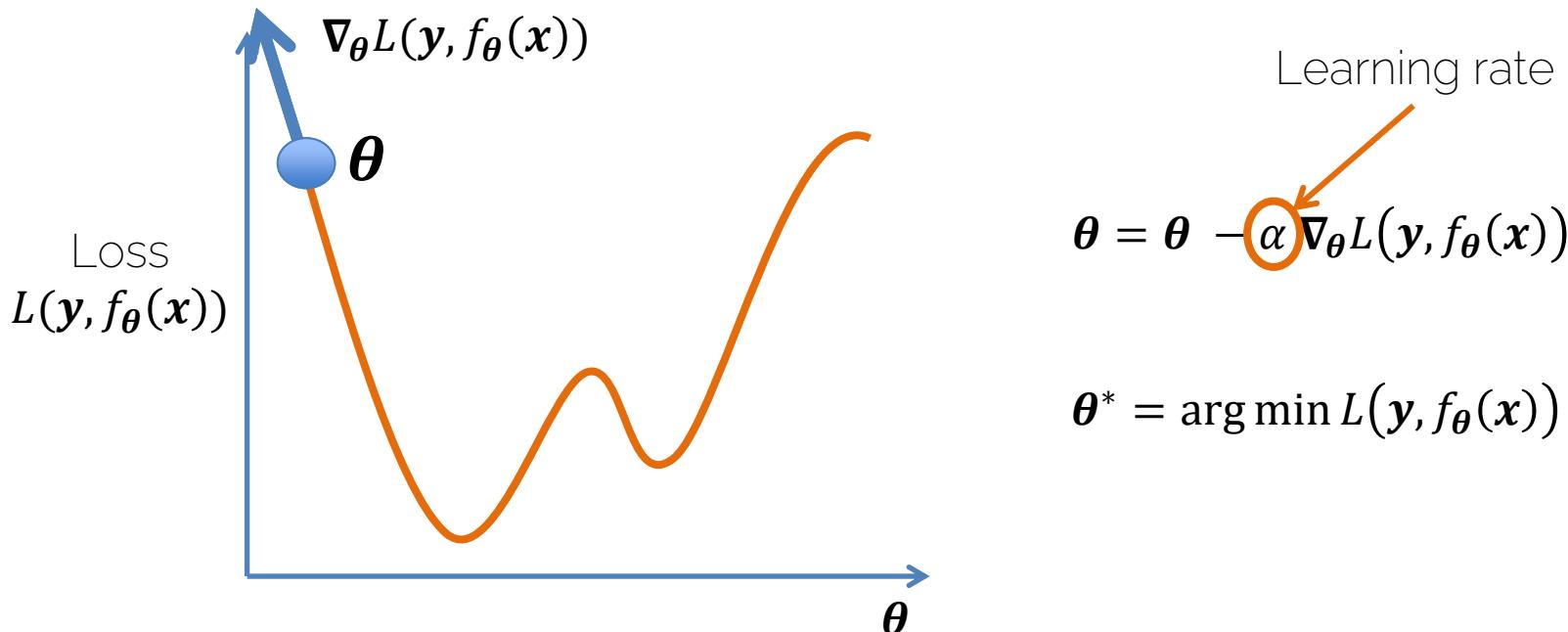
# How to Find a Better NN?

- Minimize:  $L(\mathbf{y}, f_{\theta}(\mathbf{x}))$  w.r.t.  $\theta$
- In the context of NN, we use gradient-based optimization



# How to Find a Better NN?

- Minimize:  $L(y, f_{\theta}(x))$  w.r.t.  $\theta$



# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given one layer NN with no activation function

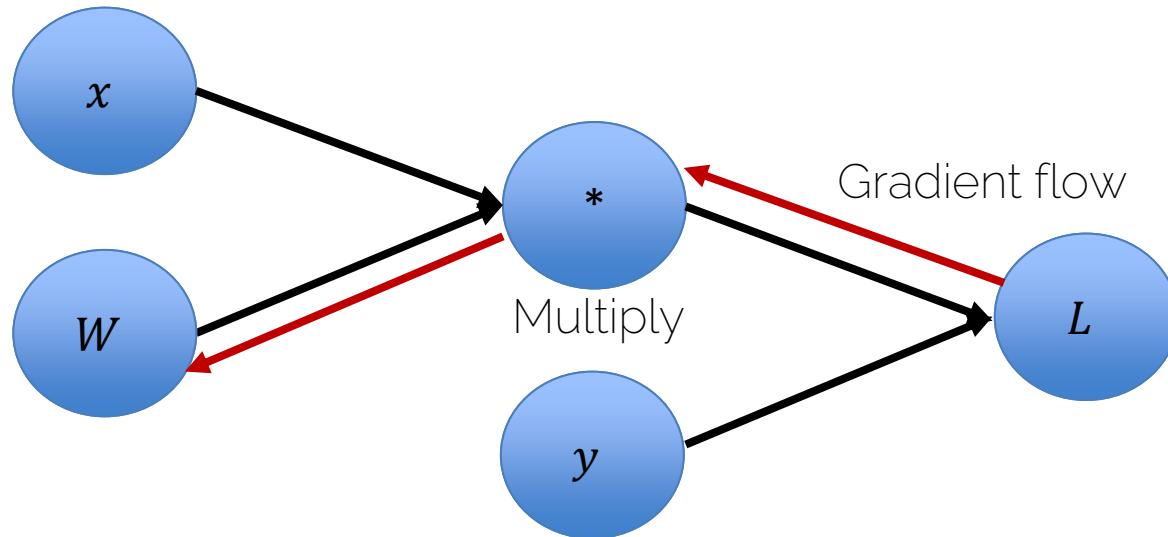
$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x}, \quad \boldsymbol{\theta} = \mathbf{W}$$

Later  $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$

- Given MSE Loss:  $L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n \|y_i - \hat{y}_i\|_2^2$

# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given one layer NN with no activation function
- Given MSE Loss:  $L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n \|\mathbf{y}_i - \mathbf{W} \cdot \mathbf{x}_i\|_2^2$



# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given one layer NN with no activation function

$$f_{\theta}(\mathbf{x}) = \mathbf{W}\mathbf{x}, \quad \boldsymbol{\theta} = \mathbf{W}$$

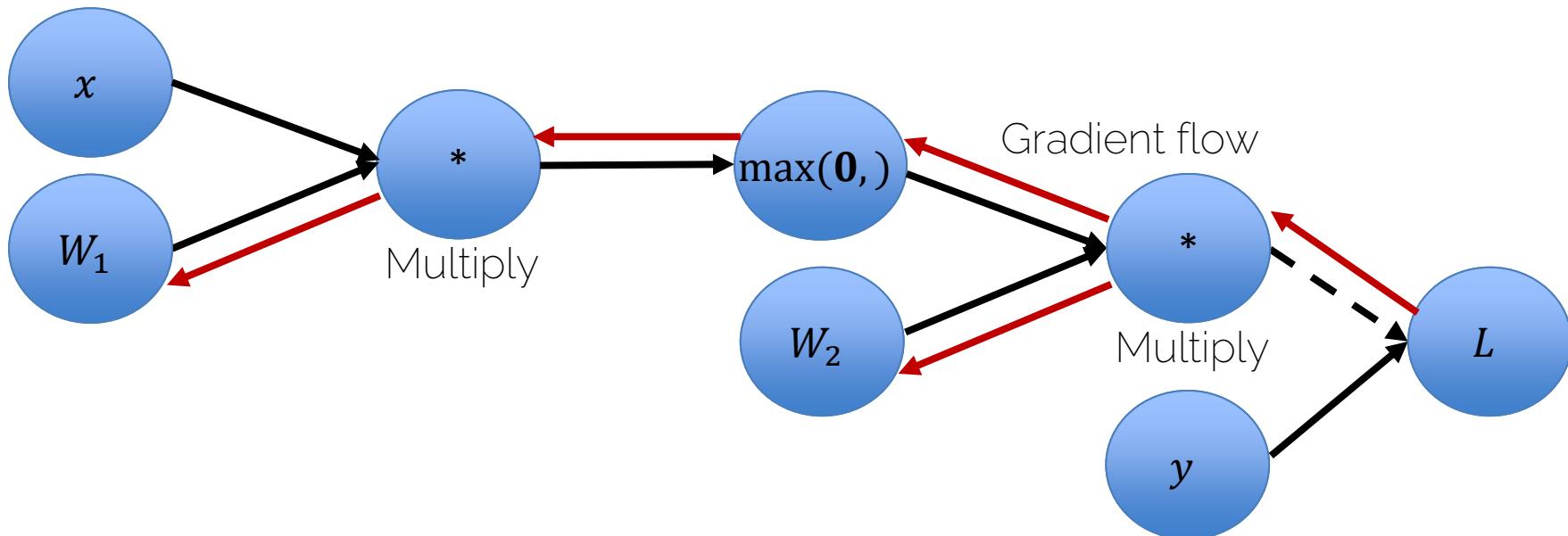
- Given MSE Loss:  $L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n \|\mathbf{W} \cdot \mathbf{x}_i - y_i\|_2^2$
- $\nabla_{\theta} L(\mathbf{y}, f_{\theta}(\mathbf{x})) = \frac{2}{n} \sum_i^n (\mathbf{W} \cdot \mathbf{x}_i - y_i) \cdot \mathbf{x}_i^T$

# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given a multi-layer NN with many activations  
$$f = \mathbf{W}_5 \sigma(\mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}))))$$
- Gradient descent for  $L(\mathbf{y}, f_{\theta}(\mathbf{x}))$  w. r. t.  $\theta$ 
  - Need to propagate gradients from end to first layer ( $\mathbf{W}_1$ ).

# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given multi-layer NN with many activations



# How to Find a Better NN?

- Given inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Given multilayer layer NN with many activations  
$$\mathbf{f} = \mathbf{W}_5 \sigma(\mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}))))$$
- Gradient descent solution for  $L(\mathbf{y}, f_{\theta}(\mathbf{x}))$  w. r. t.  $\theta$ 
  - Need to propagate gradients from end to first layer ( $\mathbf{W}_1$ )
- Backpropagation: Use chain rule to compute gradients
  - Compute graphs come in handy!

# How to Find a Better NN?

- Why gradient descent?
  - Easy to compute using compute graphs
- Other methods include
  - Newtons method
  - L-BFGS
  - Adaptive moments
  - Conjugate gradient

# Summary

- Neural Networks are computational graphs
- Goal: for a given train set, find optimal weights
- Optimization is done using gradient-based solvers
  - Many options (more in the next lectures)
- Gradients are computed via backpropagation
  - Nice because can easily modularize complex functions

# Next Lectures

- Next Lecture:
  - Backpropagation and optimization of Neural Networks
- Check for updates on website/piazza regarding exercises

See you next week ☺

# Further Reading

- Optimization:
  - <http://cs231n.github.io/optimization-1/>
  - <http://www.deeplearningbook.org/contents/optimization.html>
- General concepts:
  - Pattern Recognition and Machine Learning – C. Bishop
  - <http://www.deeplearningbook.org/>

# Optimization and Backpropagation

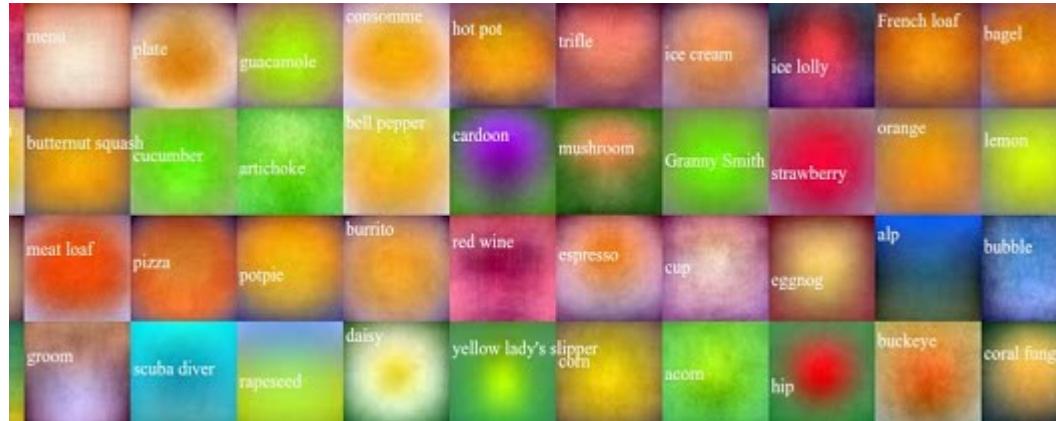
# Lecture 3 Recap

# Neural Network

- Linear score function  $f = Wx$



On CIFAR-10

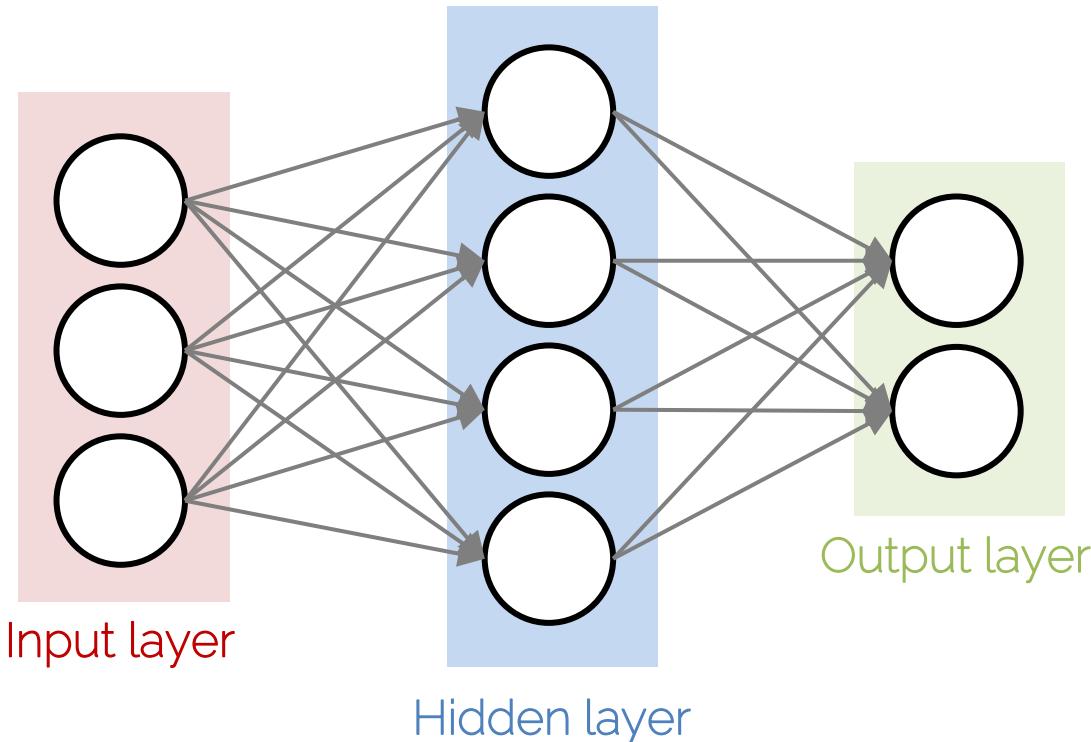


On ImageNet

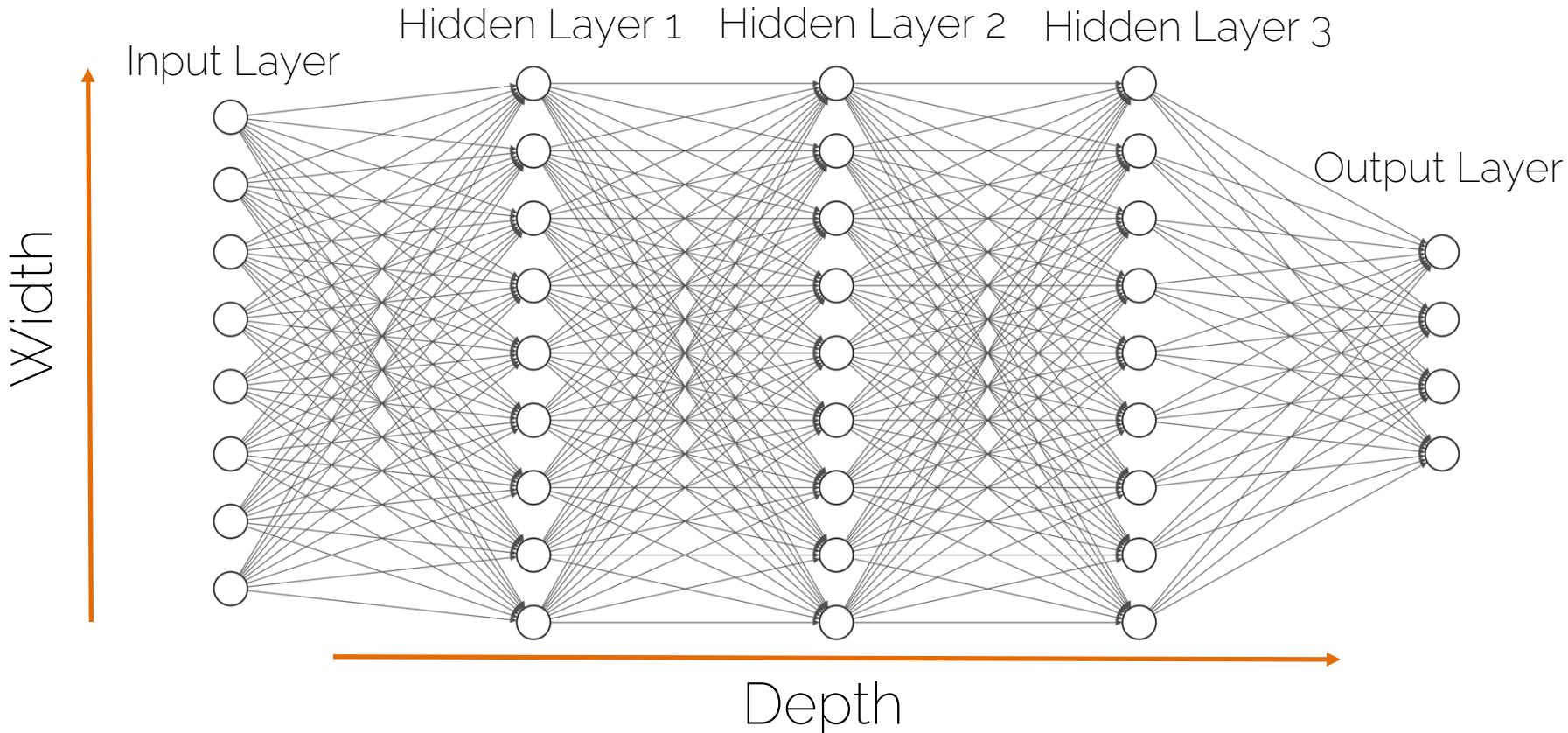
# Neural Network

- Linear score function  $f = \mathbf{W}x$
- Neural network is a nesting of 'functions'
  - 2-layers:  $f = \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x)$
  - 3-layers:  $f = \mathbf{W}_3 \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x))$
  - 4-layers:  $f = \mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x)))$
  - 5-layers:  $f = \mathbf{W}_5 \sigma(\mathbf{W}_4 \tanh(\mathbf{W}_3, \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 x))))$
  - ... up to hundreds of layers

# Neural Network

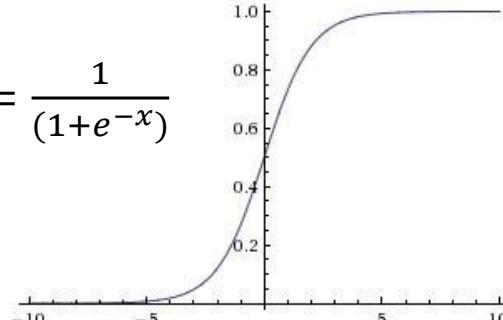


# Neural Network

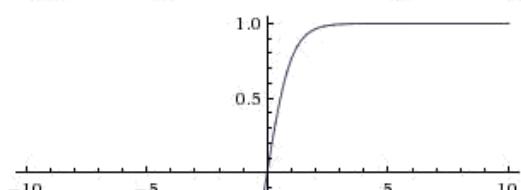


# Activation Functions

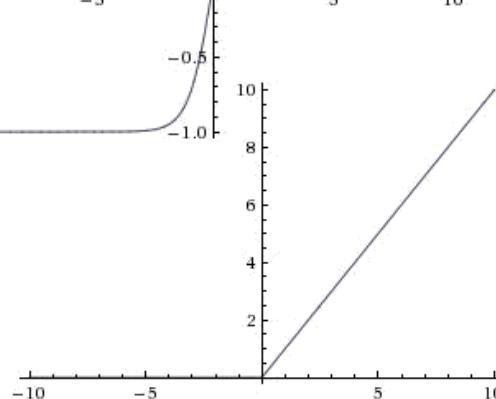
Sigmoid:  $\sigma(x) = \frac{1}{(1+e^{-x})}$



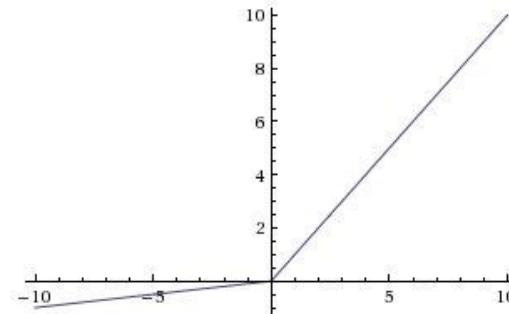
tanh:  $\tanh(x)$



ReLU:  $\max(0, x)$



Leaky ReLU:  $\max(0.1x, x)$



Parametric ReLU:  $\max(\alpha x, x)$

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

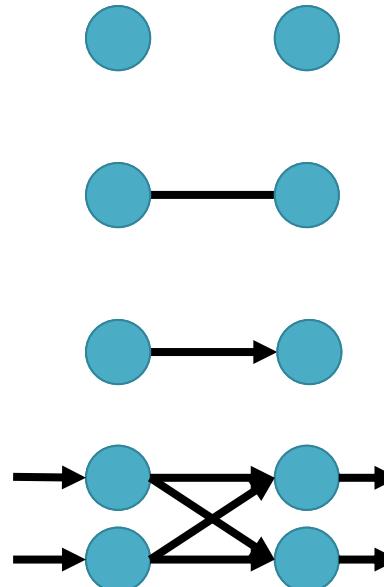
$$\text{ELU } f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

# Loss Functions

- Measure the goodness of the predictions (or equivalently, the network's performance)
- Regression loss
  - L<sub>1</sub> loss  $\mathbf{L}(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n \|y_i - \hat{y}_i\|_1$
  - MSE loss  $\mathbf{L}(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n \|y_i - \hat{y}_i\|_2^2$
- Classification loss (for multi-class classification)
  - Cross Entropy loss  $E(y, \hat{y}; \theta) = - \sum_{i=1}^n \sum_{k=1}^K (y_{ik} \cdot \log \hat{y}_{ik})$

# Computational Graphs

- Neural network is a computational graph
  - It has compute nodes
  - It has edges that connect nodes
  - It is directional
  - It is organized in 'layers'



# Backprop

# The Importance of Gradients

- Our optimization schemes are based on computing gradients

$$\nabla_{\theta} L(\theta)$$

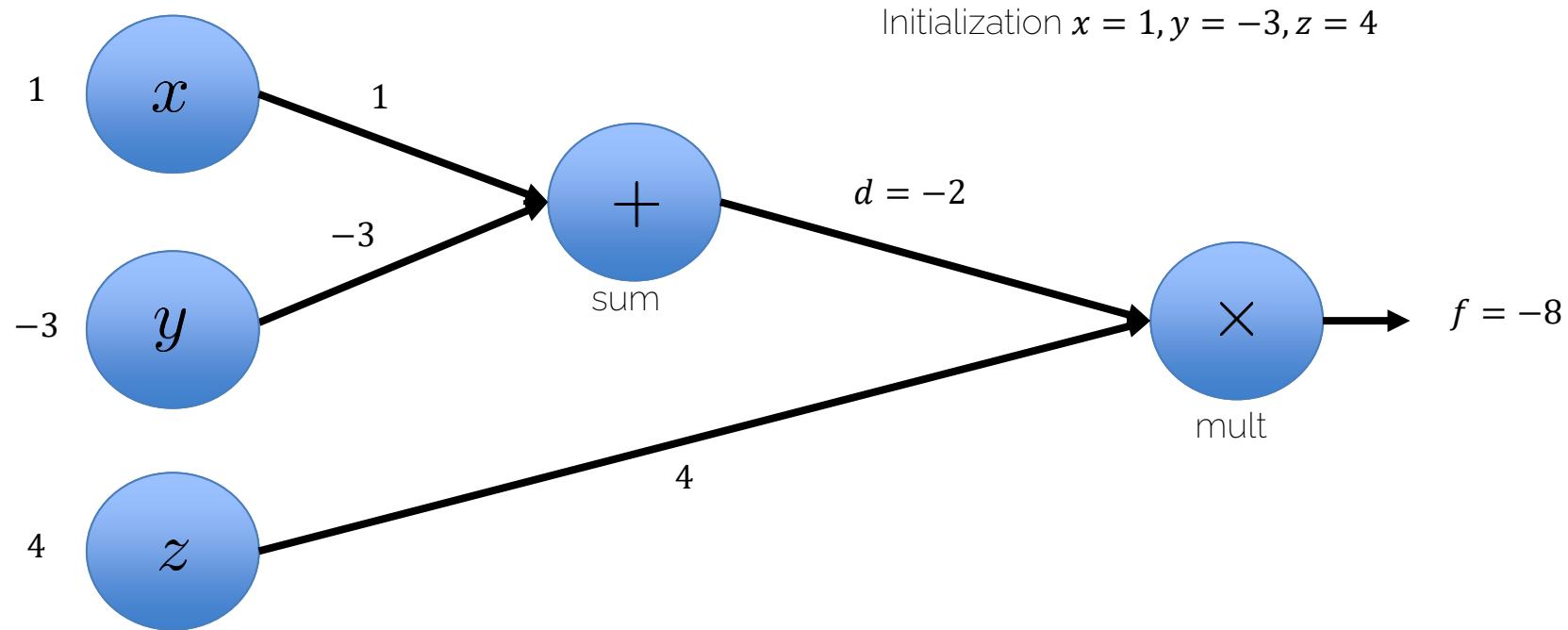
- One can compute gradients analytically but what if our function is too complex?
- Break down gradient computation

Backpropagation

Done by many people before, but often credited to Rumelhart 1986

# Backprop: Forward Pass

- $f(x, y, z) = (x + y) \cdot z$



# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

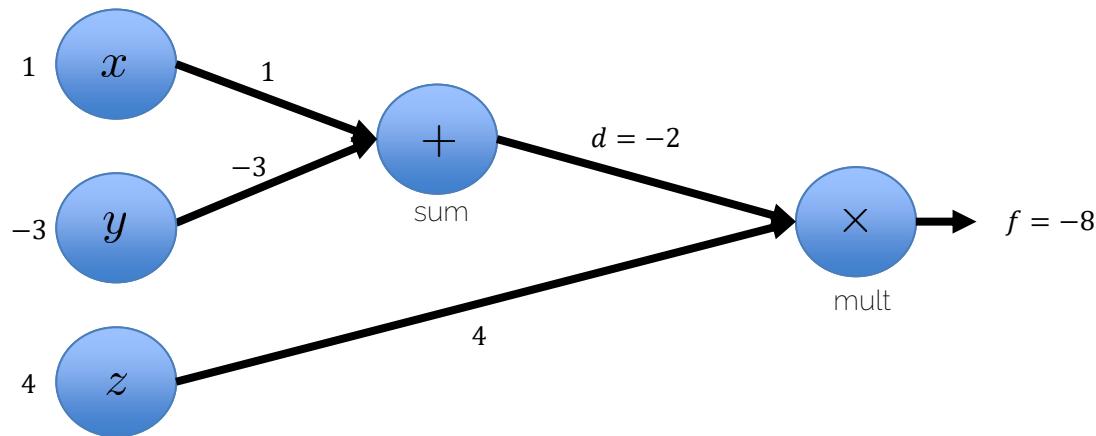
$$d = x + y$$

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z$$

$$\frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$  ?



# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

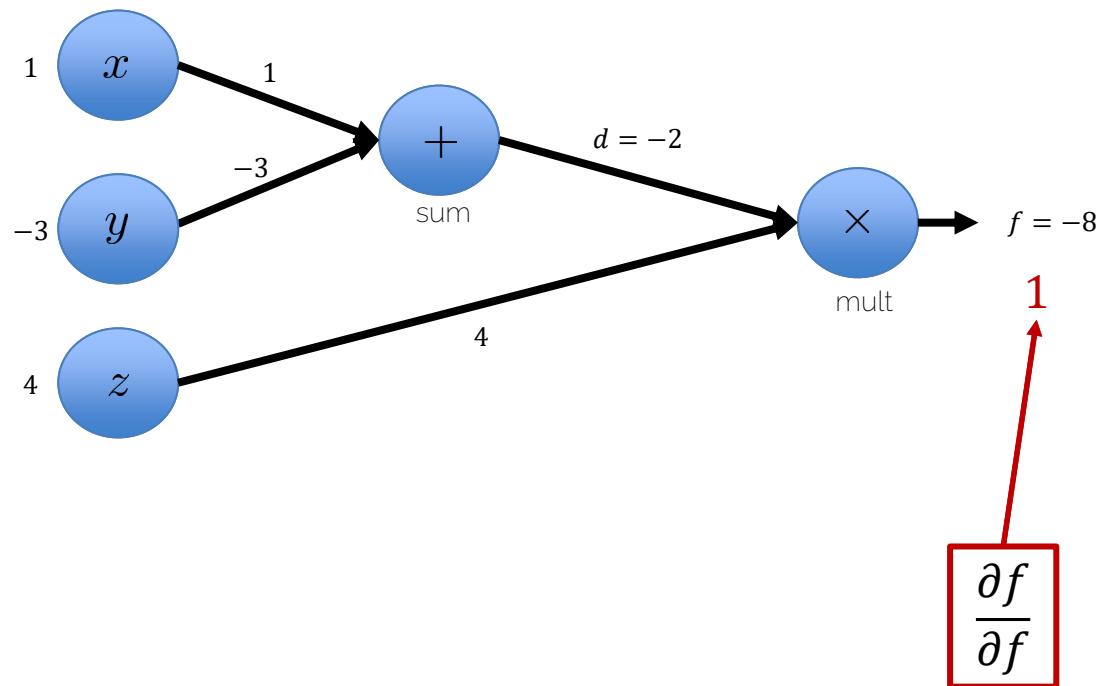
$$d = x + y$$

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z$$

$$\frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$  ?



# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

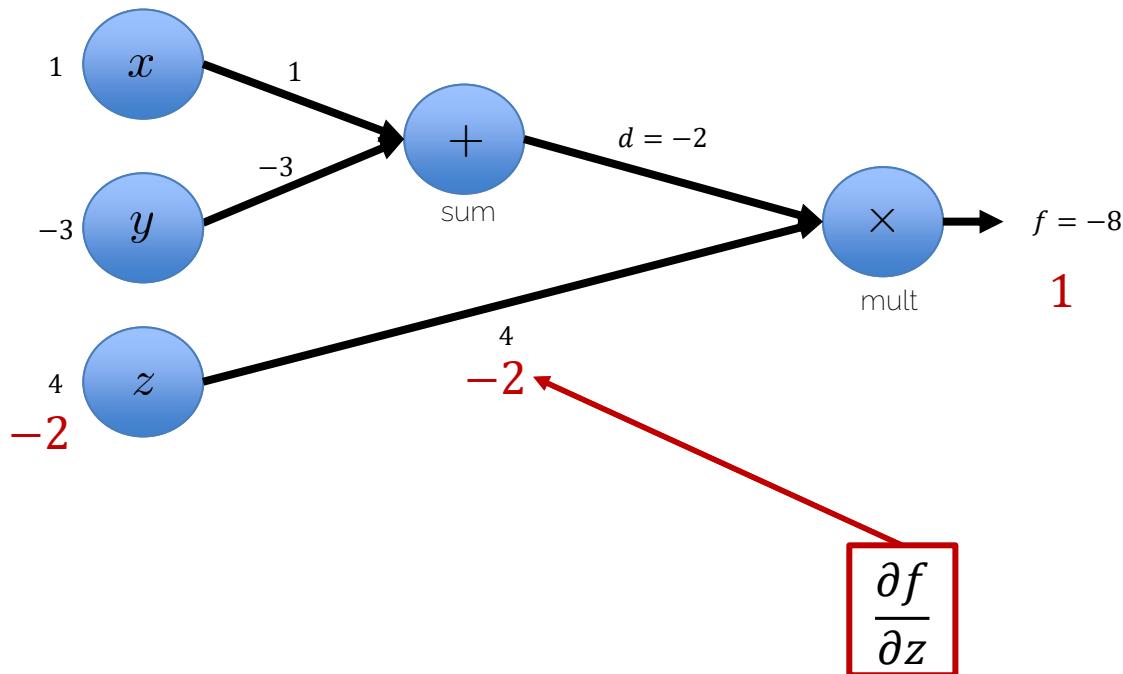
$$d = x + y$$

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z$$

$$\frac{\partial f}{\partial d} = z, \boxed{\frac{\partial f}{\partial z} = d}$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$  ?



# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

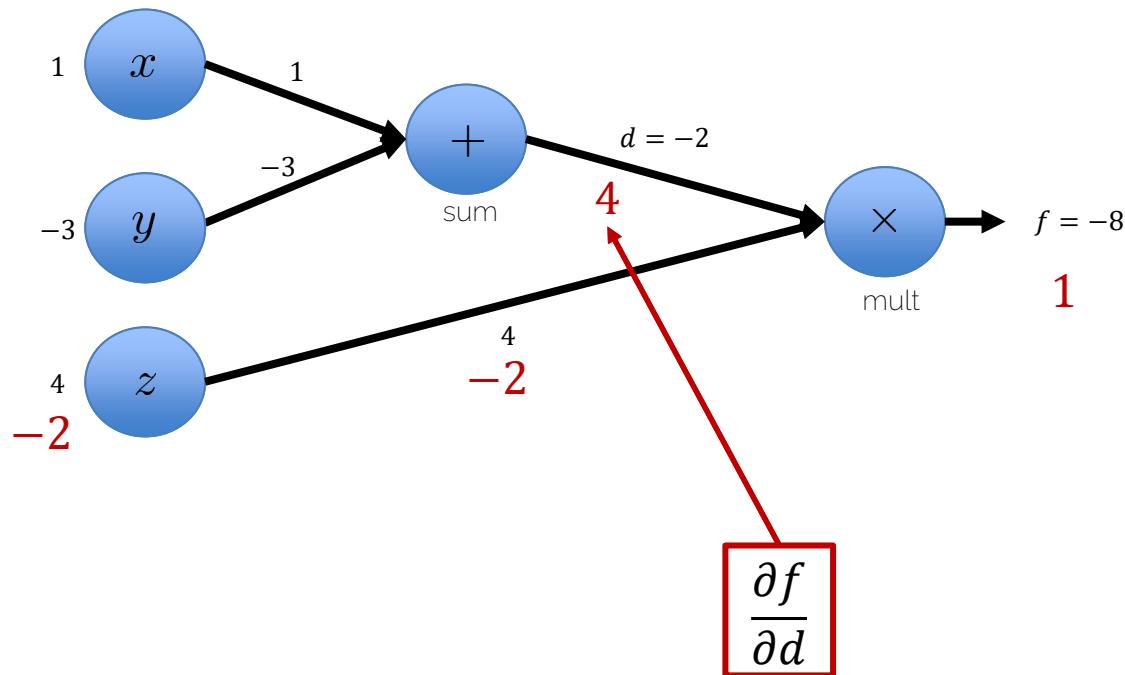
$$d = x + y$$

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z$$

$$\boxed{\frac{\partial f}{\partial d} = z} \quad \frac{\partial f}{\partial z} = d$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$  ?



# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

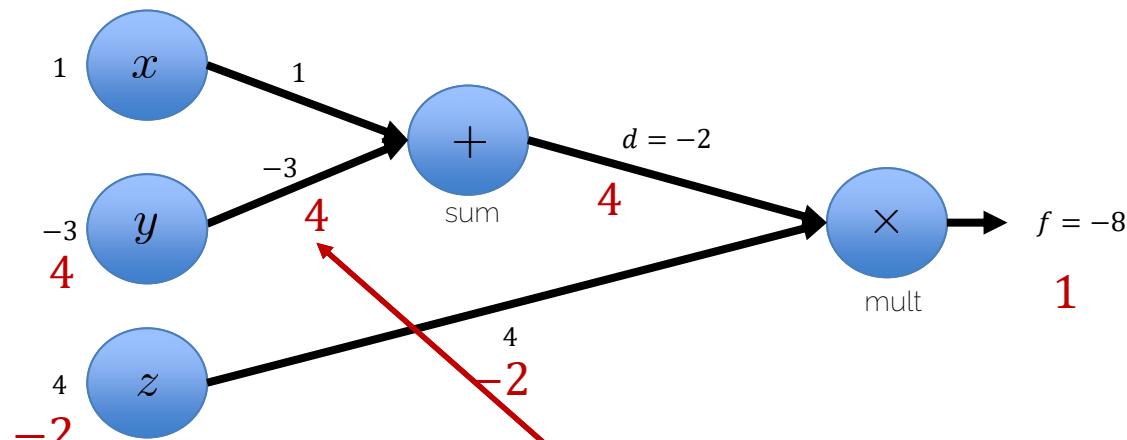
$$d = x + y$$

$$\frac{\partial d}{\partial x} = 1, \boxed{\frac{\partial d}{\partial y} = 1}$$

$$f = d \cdot z$$

$$\frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ ?



Chain Rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial d} \cdot \frac{\partial d}{\partial y}}$$

$$\boxed{\frac{\partial f}{\partial y}}$$

$$\rightarrow \frac{\partial f}{\partial y} = 4 \cdot 1 = 4$$

# Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with  $x = 1, y = -3, z = 4$

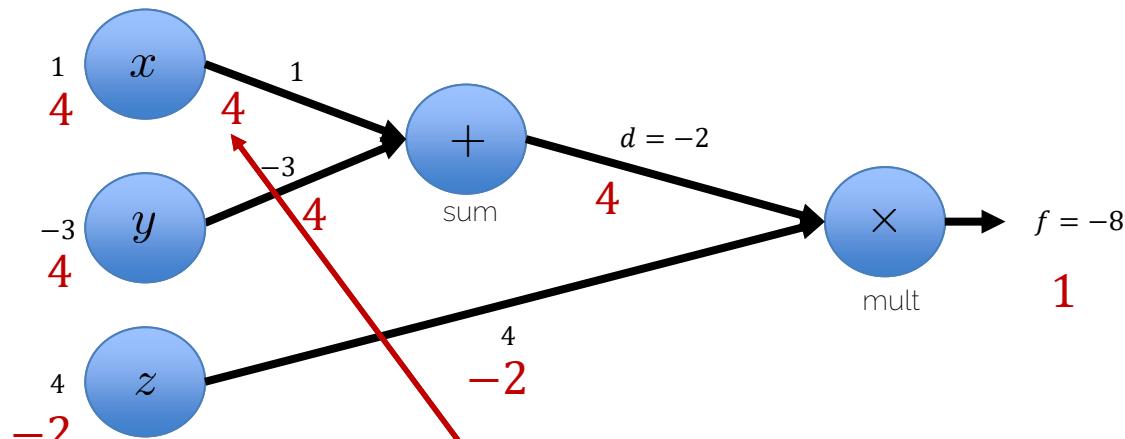
$$d = x + y$$

$$\boxed{\frac{\partial d}{\partial x} = 1}, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z$$

$$\frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

What is  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ ?



Chain Rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial d} \cdot \frac{\partial d}{\partial y}}$$

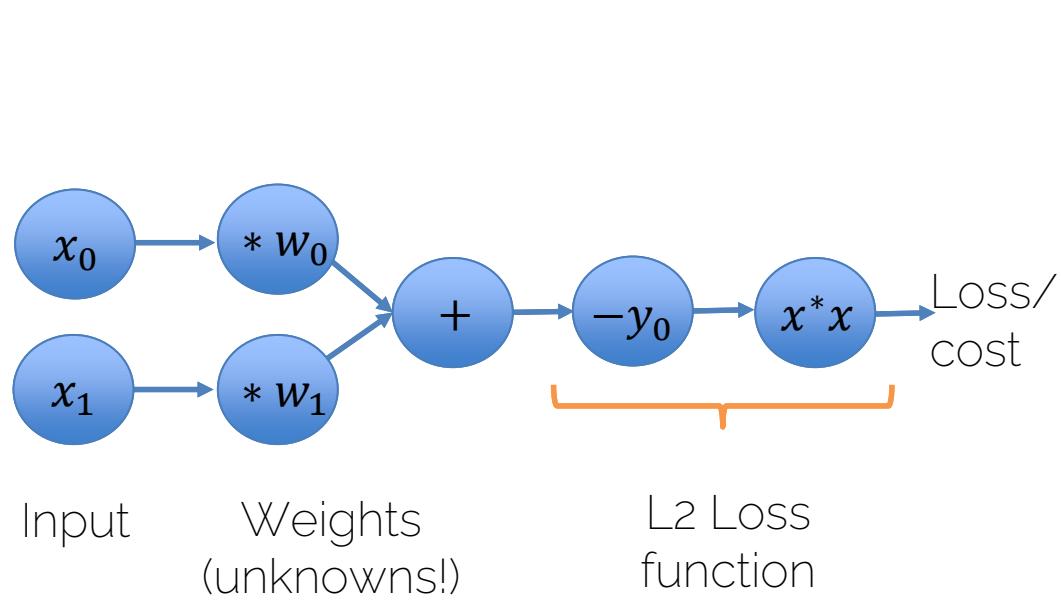
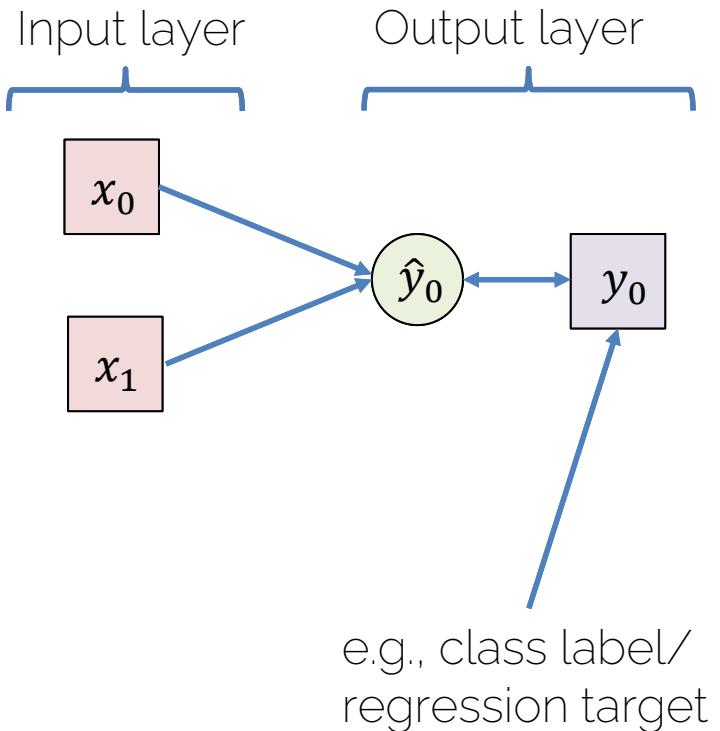
$$\boxed{\frac{\partial f}{\partial x}}$$

$$\rightarrow \frac{\partial f}{\partial x} = 4 \cdot 1 = 4$$

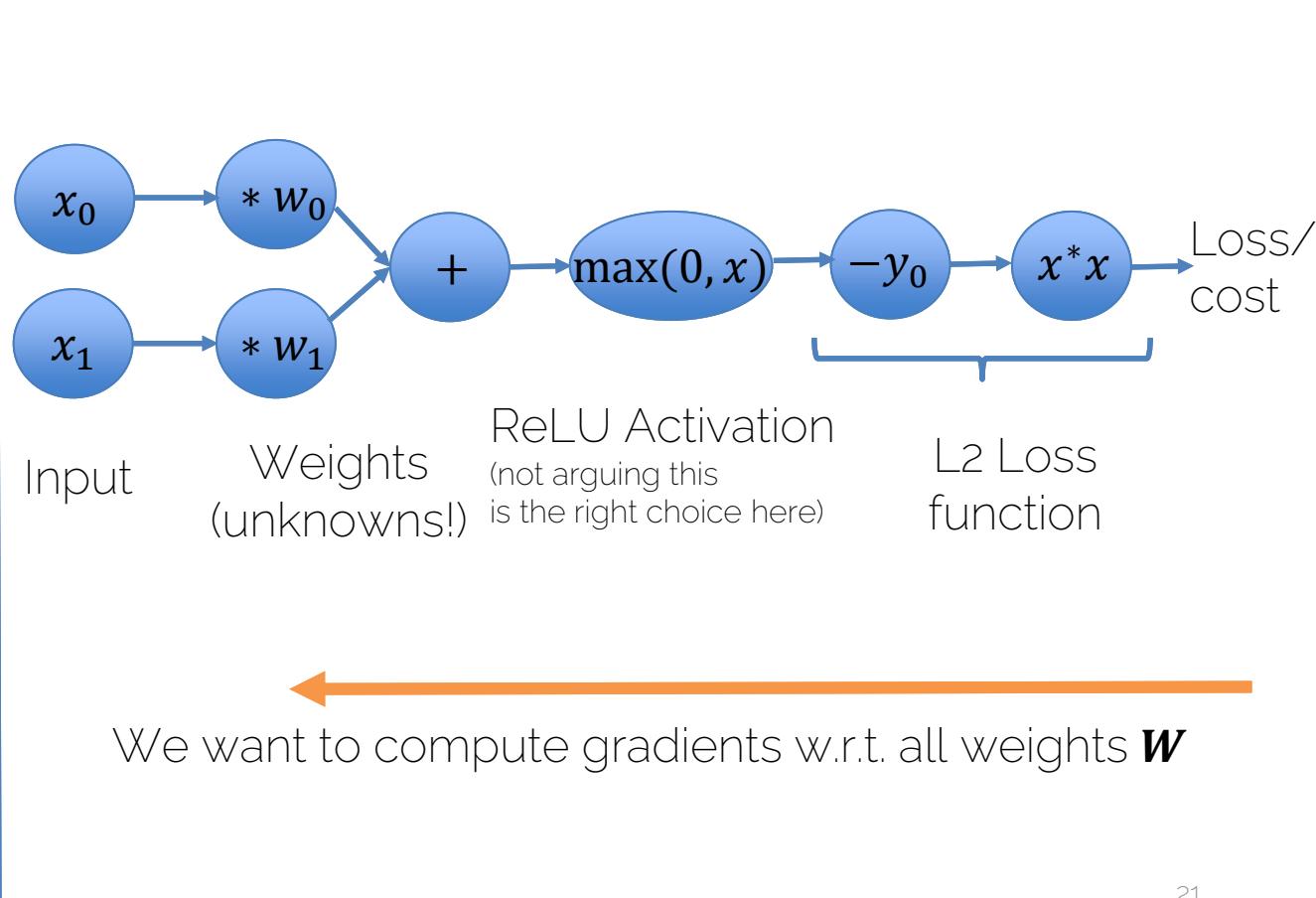
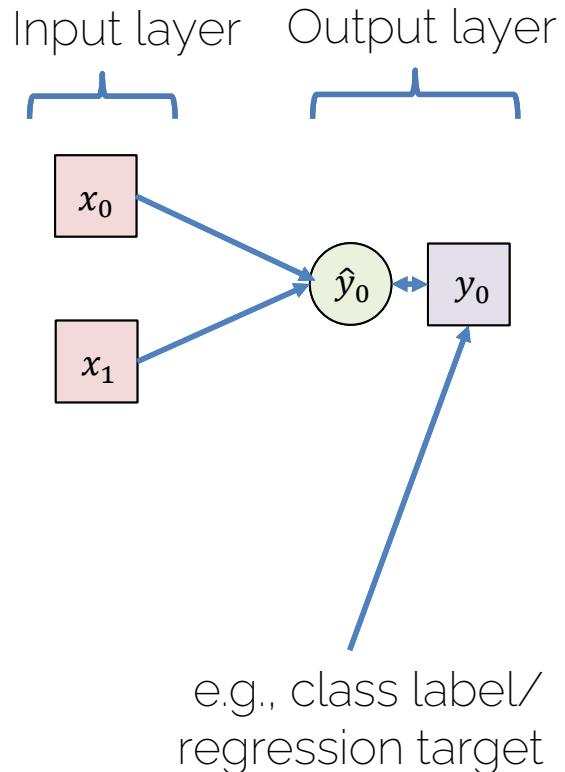
# Compute Graphs -> Neural Networks

- $x_k$  input variables
- $w_{l,m,n}$  network weights (note 3 indices)
  - $l$  which layer
  - $m$  which neuron in layer
  - $n$  which weight in neuron
- $\hat{y}_i$  computed output ( $i$  output dim;  $n_{out}$ )
- $y_i$  ground truth targets
- $L$  loss function

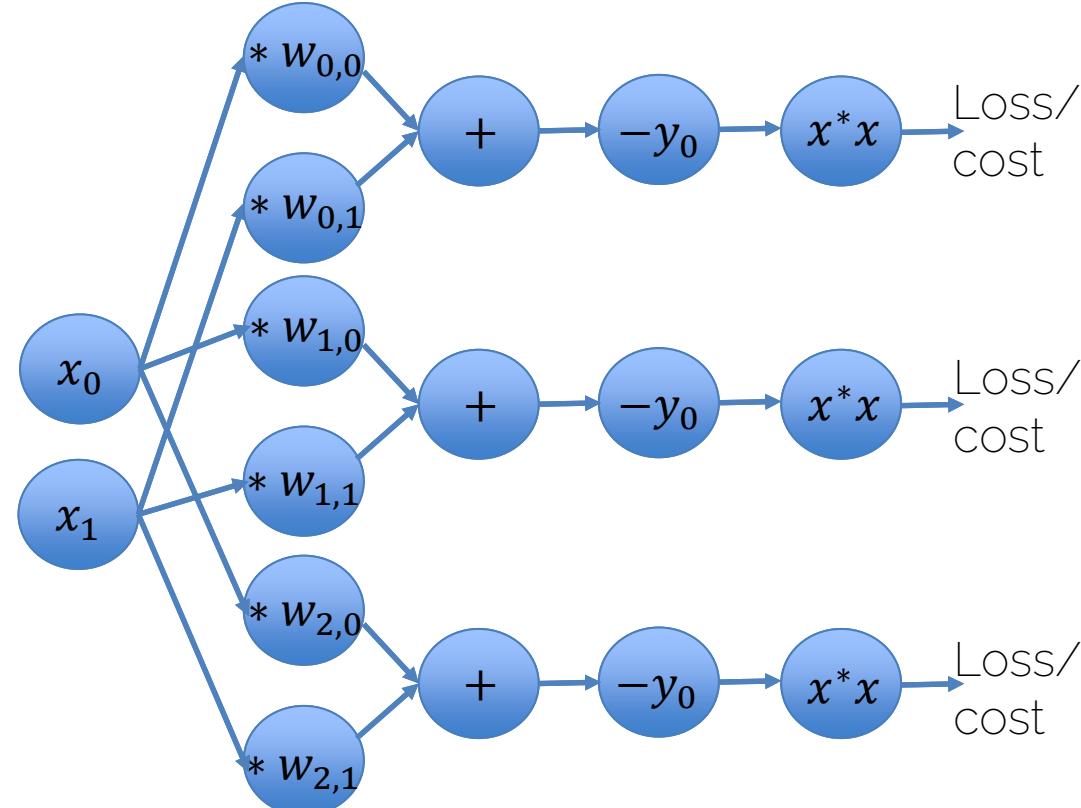
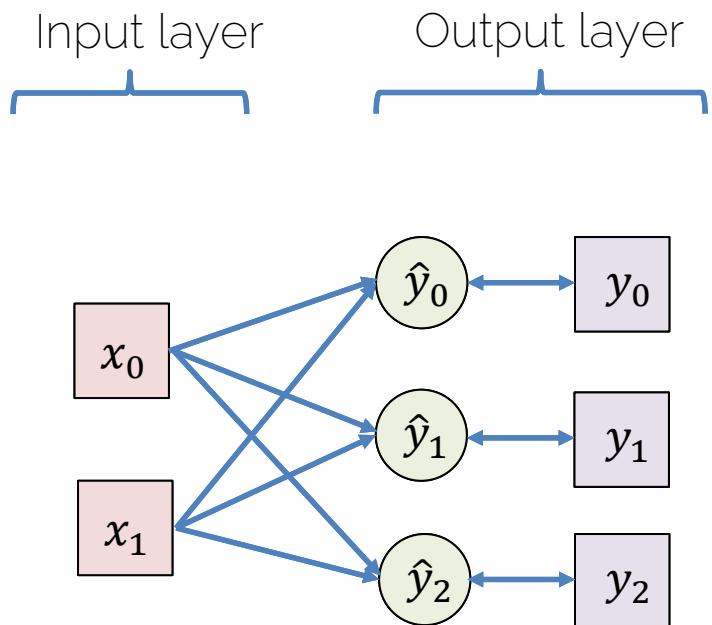
# Compute Graphs → Neural Networks



# Compute Graphs -> Neural Networks



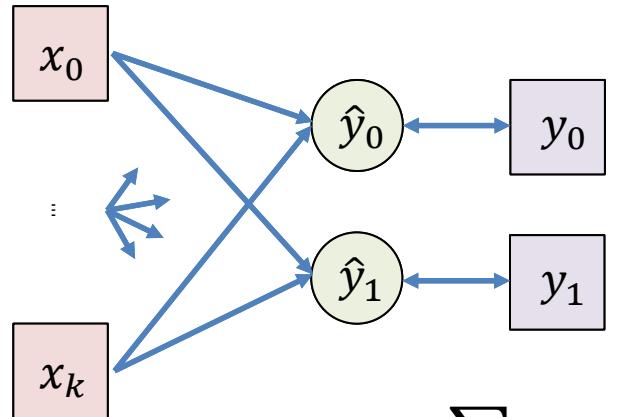
# Compute Graphs -> Neural Networks



We want to compute gradients w.r.t. all weights  $\mathbf{W}$

# Compute Graphs -> Neural Networks

Input layer                      Output layer



$$\hat{y}_i = A(b_i + \sum_k x_k w_{i,k})$$

Activation function      bias

Goal: We want to compute gradients of the loss function  $L$  w.r.t. all weights  $\mathbf{W}$

$$L = \sum_i L_i$$

$L$ : sum over loss per sample, e.g.  
L2 loss → simply sum up squares:

$$L_i = (\hat{y}_i - y_i)^2$$

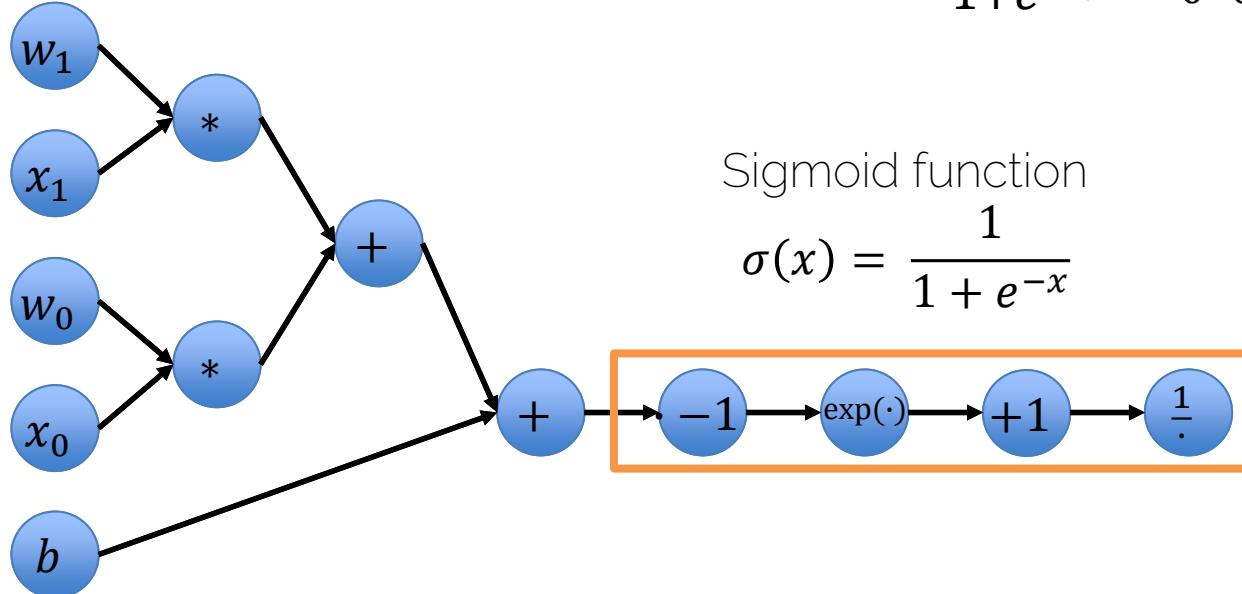
→ use chain rule to compute partials

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

We want to compute gradients w.r.t. all weights  $\mathbf{W}$  AND all biases  $\mathbf{b}$

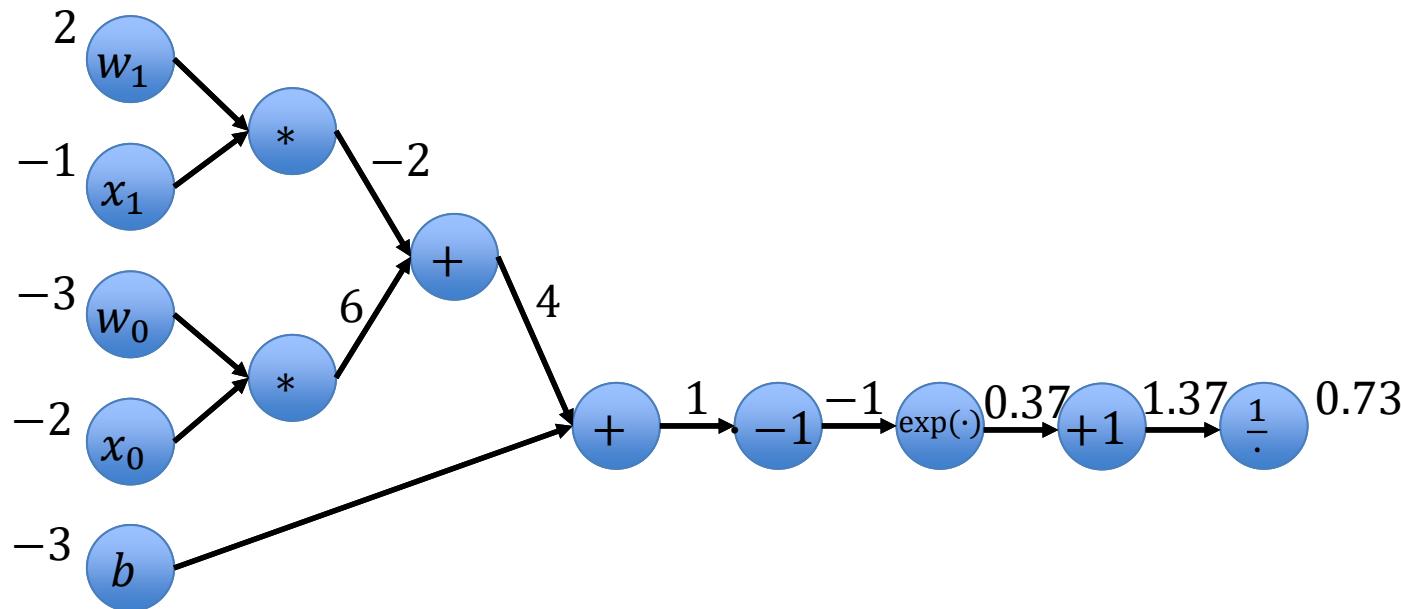
# NNs as Computational Graphs

- We can express any kind of functions in a computational graph, e.g.  $f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$



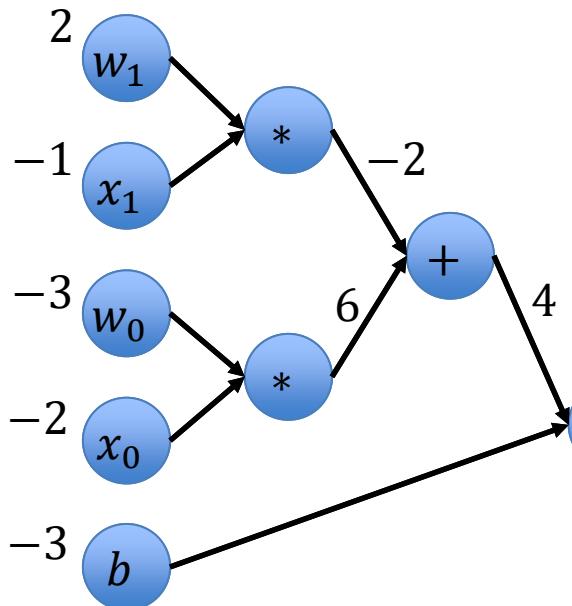
# NNs as Computational Graphs

- $f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$



# NNs as Computational Graphs

- $f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$



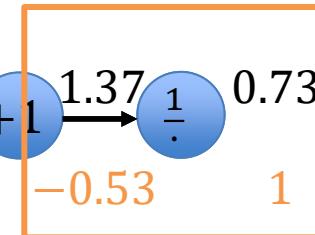
$$g(x) = \frac{1}{x} \Rightarrow \frac{\partial g}{\partial x} = -\frac{1}{x^2}$$

$$g_\alpha(x) = \alpha + x \Rightarrow \frac{\partial g}{\partial x} = 1$$

$$g(x) = e^x \Rightarrow \frac{\partial g}{\partial x} = e^x$$

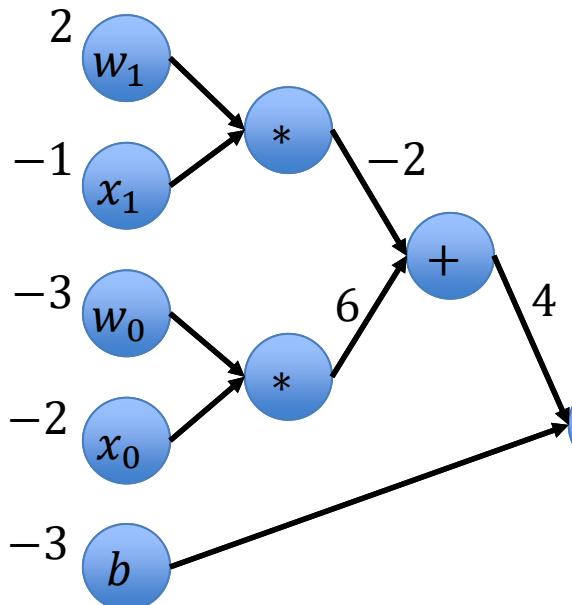
$$g_\alpha(x) = \alpha x \Rightarrow \frac{\partial g}{\partial x} = \alpha$$

$$1 \cdot -\frac{1}{1.37^2} = -0.53$$



# NNs as Computational Graphs

- $$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$$

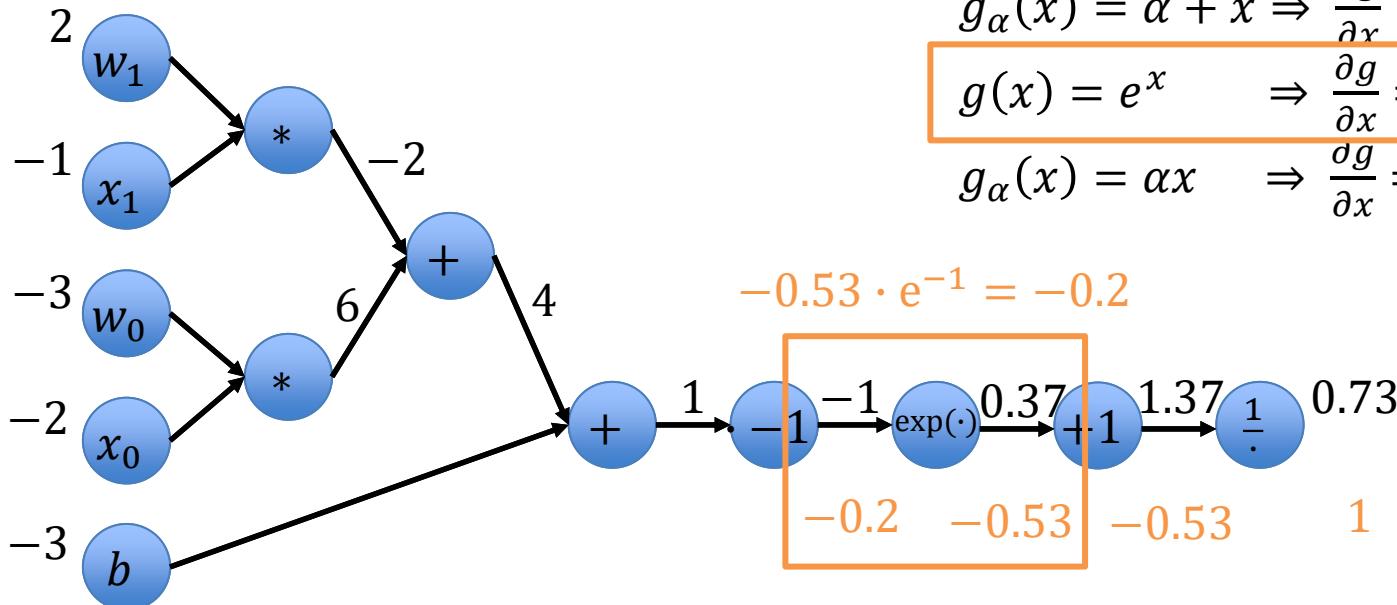


$$\begin{aligned} g(x) &= \frac{1}{x} & \Rightarrow \frac{\partial g}{\partial x} &= -\frac{1}{x^2} \\ g_\alpha(x) &= \alpha + x & \Rightarrow \frac{\partial g}{\partial x} &= 1 \\ g(x) &= e^x & \Rightarrow \frac{\partial g}{\partial x} &= e^x \\ g_\alpha(x) &= \alpha x & \Rightarrow \frac{\partial g}{\partial x} &= \alpha \end{aligned}$$

$$\begin{aligned} -0.53 \cdot 1 &= -0.53 \\ -0.53 & \quad \quad \quad 0.37 \\ & \quad \quad \quad +1 \\ & \quad \quad \quad 1.37 \\ & \quad \quad \quad \frac{1}{\cdot} \\ & \quad \quad \quad 0.73 \\ & \quad \quad \quad 1 \end{aligned}$$

# NNs as Computational Graphs

- $$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$$



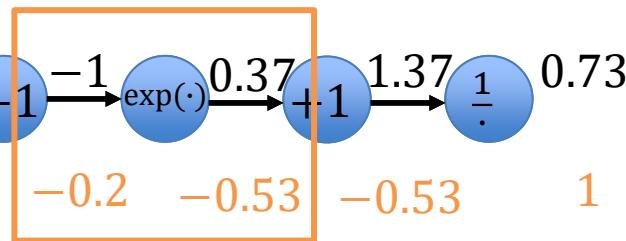
$$g(x) = \frac{1}{x} \Rightarrow \frac{\partial g}{\partial x} = -\frac{1}{x^2}$$

$$g_\alpha(x) = \alpha + x \Rightarrow \frac{\partial g}{\partial x} = 1$$

$$g(x) = e^x \Rightarrow \frac{\partial g}{\partial x} = e^x$$

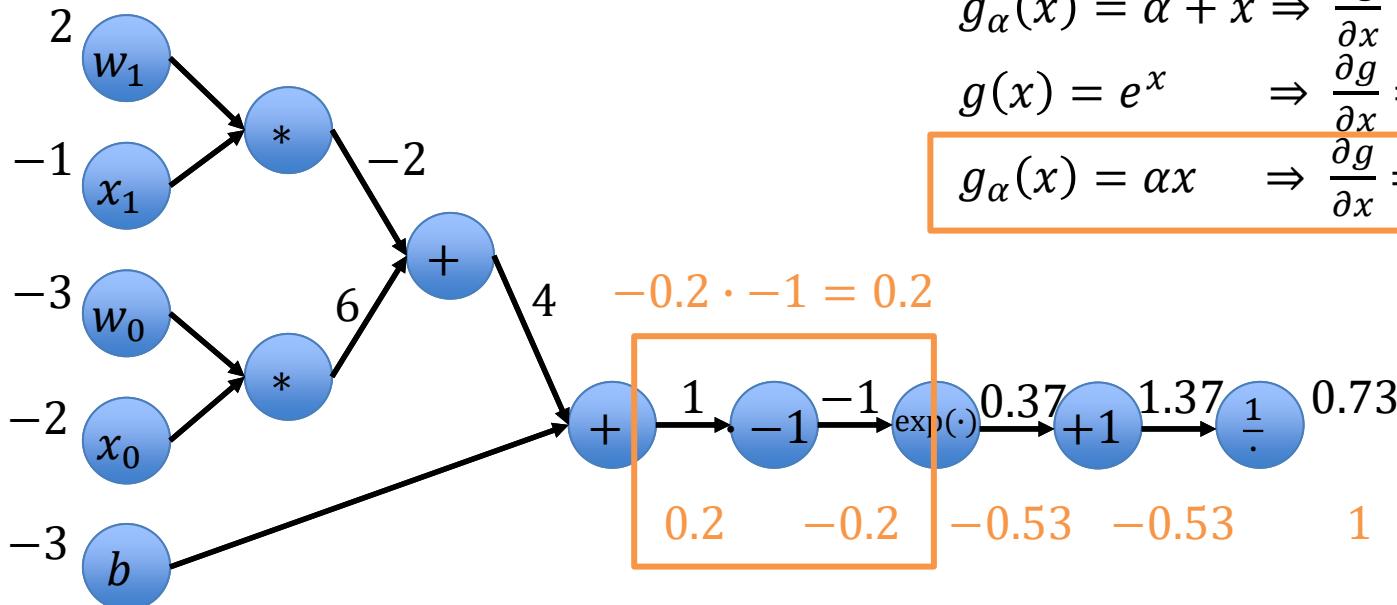
$$g_\alpha(x) = \alpha x \Rightarrow \frac{\partial g}{\partial x} = \alpha$$

$$-0.53 \cdot e^{-1} = -0.2$$



# NNs as Computational Graphs

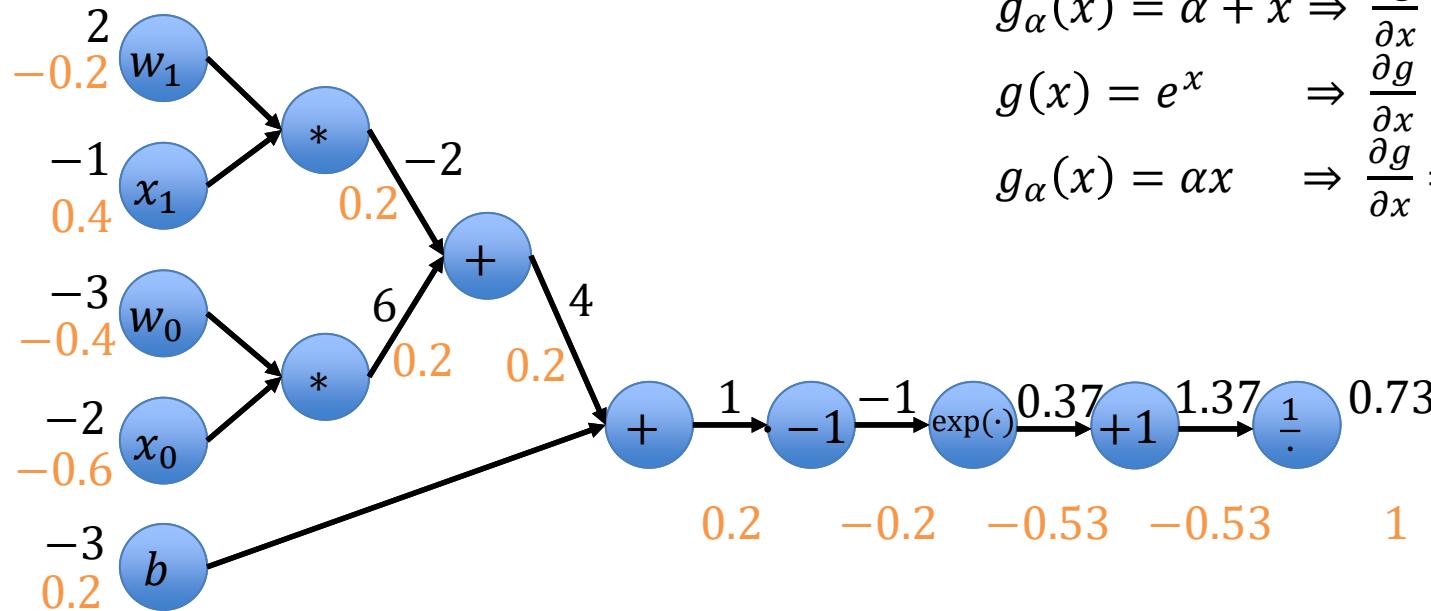
- $f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$



$$\begin{aligned} g(x) &= \frac{1}{x} & \Rightarrow \frac{\partial g}{\partial x} &= -\frac{1}{x^2} \\ g_\alpha(x) &= \alpha + x & \Rightarrow \frac{\partial g}{\partial x} &= 1 \\ g(x) &= e^x & \Rightarrow \frac{\partial g}{\partial x} &= e^x \\ g_\alpha(x) &= \alpha x & \Rightarrow \frac{\partial g}{\partial x} &= \alpha \end{aligned}$$

# NNs as Computational Graphs

- $f(\mathbf{w}, \mathbf{x}) = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$

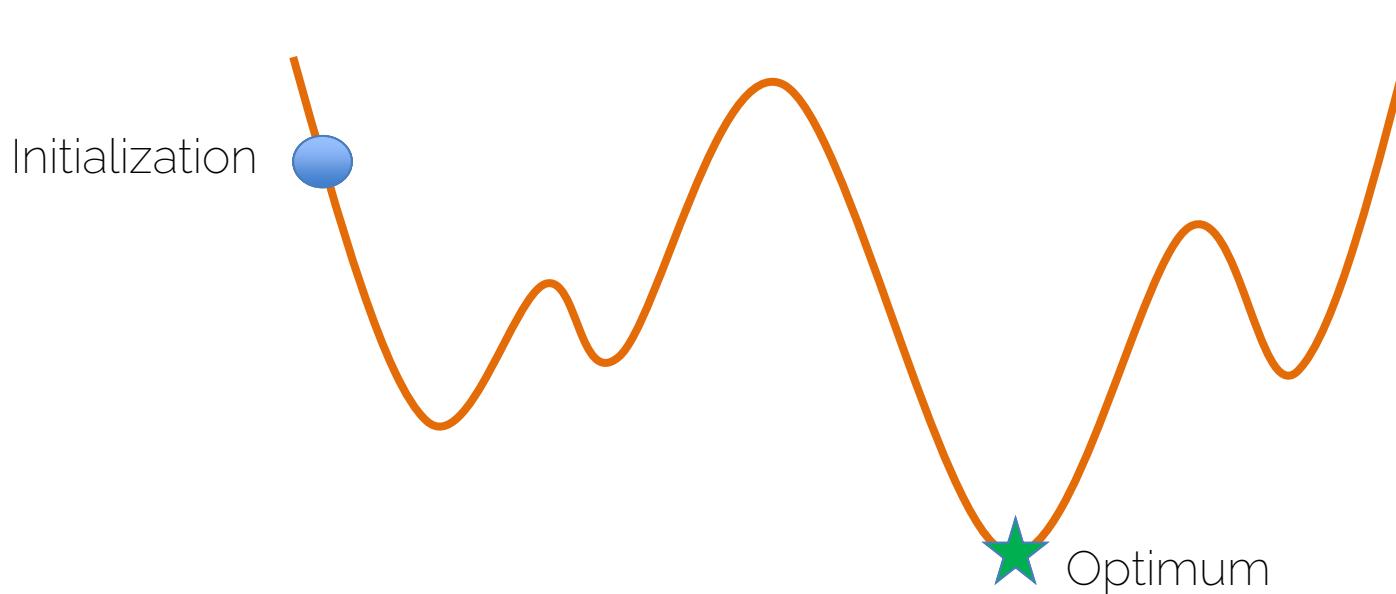


$$\begin{aligned} g(x) &= \frac{1}{x} & \Rightarrow \frac{\partial g}{\partial x} &= -\frac{1}{x^2} \\ g_\alpha(x) &= \alpha + x & \Rightarrow \frac{\partial g}{\partial x} &= 1 \\ g(x) &= e^x & \Rightarrow \frac{\partial g}{\partial x} &= e^x \\ g_\alpha(x) &= \alpha x & \Rightarrow \frac{\partial g}{\partial x} &= \alpha \end{aligned}$$

# Gradient Descent

# Gradient Descent

$$\boldsymbol{x}^* = \arg \min f(\boldsymbol{x})$$



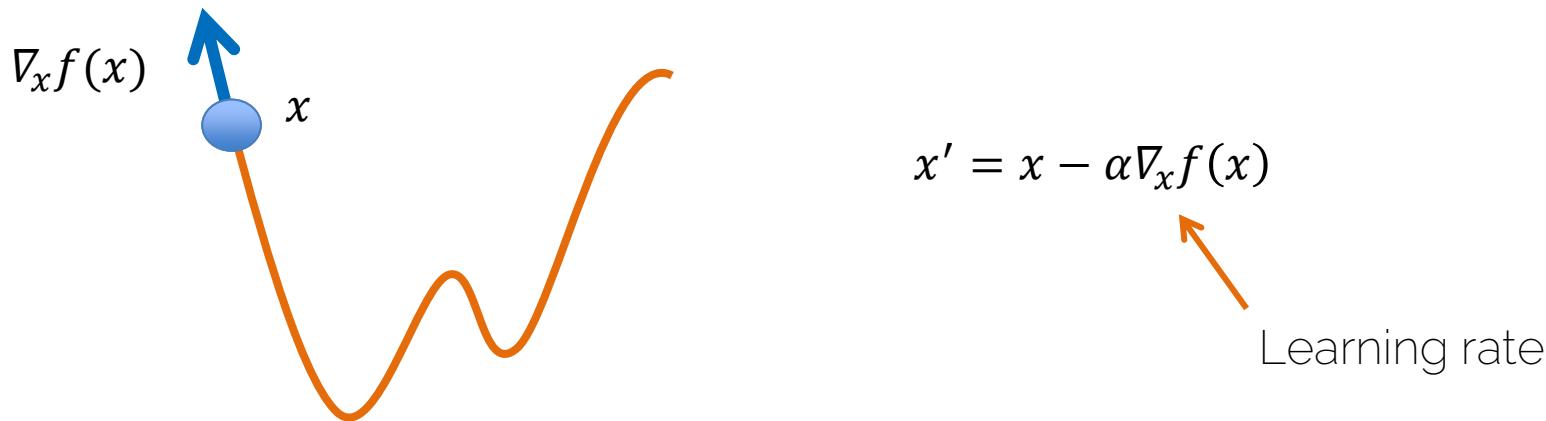
# Gradient Descent

- From derivative to gradient

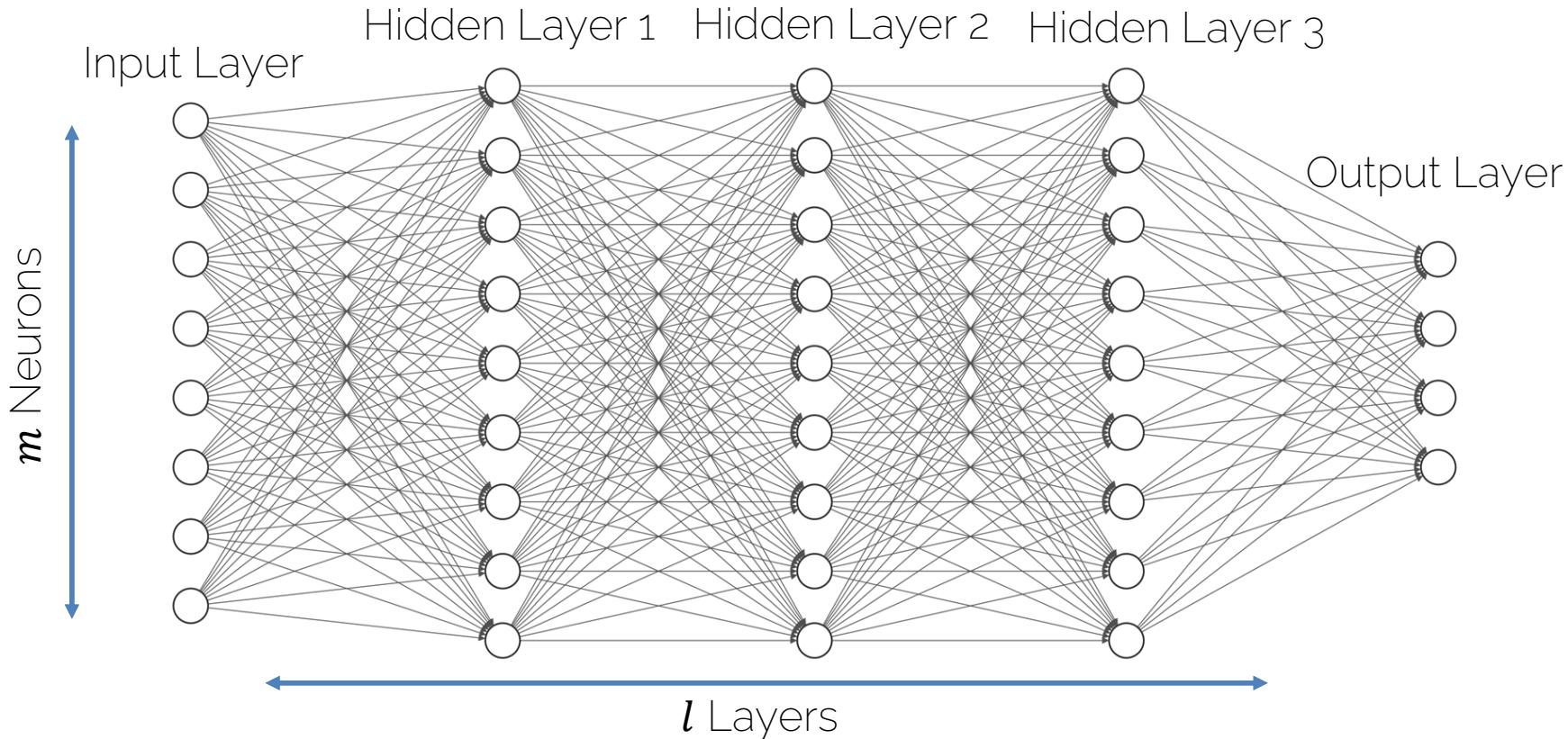
$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



# Gradient Descent for Neural Networks



# Gradient Descent for Neural Networks

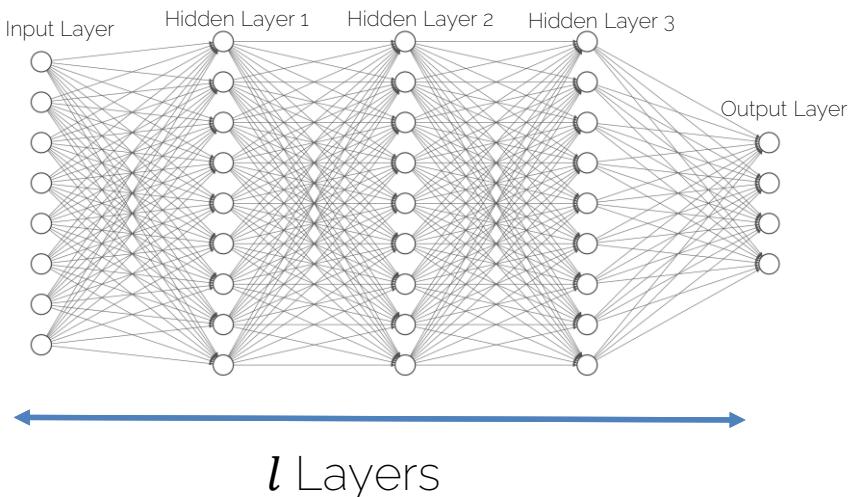
For a given training pair  $\{\mathbf{x}, \mathbf{y}\}$ , we want to update all weights, i.e., we need to compute the derivatives w.r.t. to all weights:

$$\nabla_{\mathbf{W}} f_{\{\mathbf{x}, \mathbf{y}\}}(\mathbf{W}) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \vdots \\ \vdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{bmatrix}$$

*m* Neurons

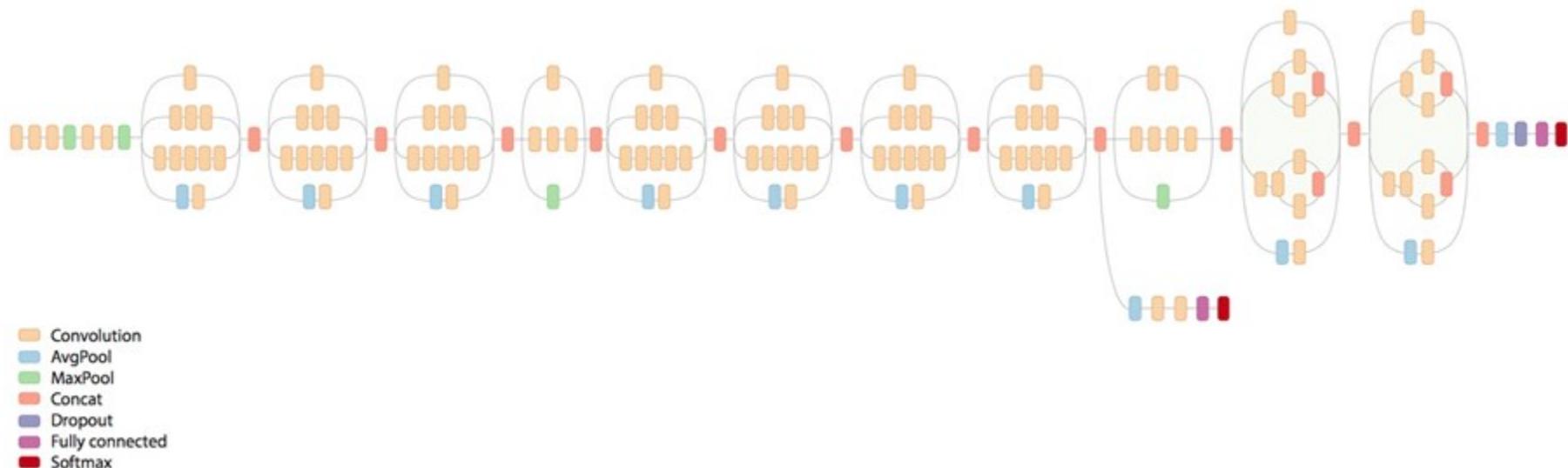
Gradient step:

$$\mathbf{W}' = \mathbf{W} - \alpha \nabla_{\mathbf{W}} f_{\{\mathbf{x}, \mathbf{y}\}}(\mathbf{W})$$



# NNs can Become Quite Complex...

- These graphs can be huge!



[Szegedy et al., CVPR'15] Going Deeper with Convolutions

# The Flow of the Gradients

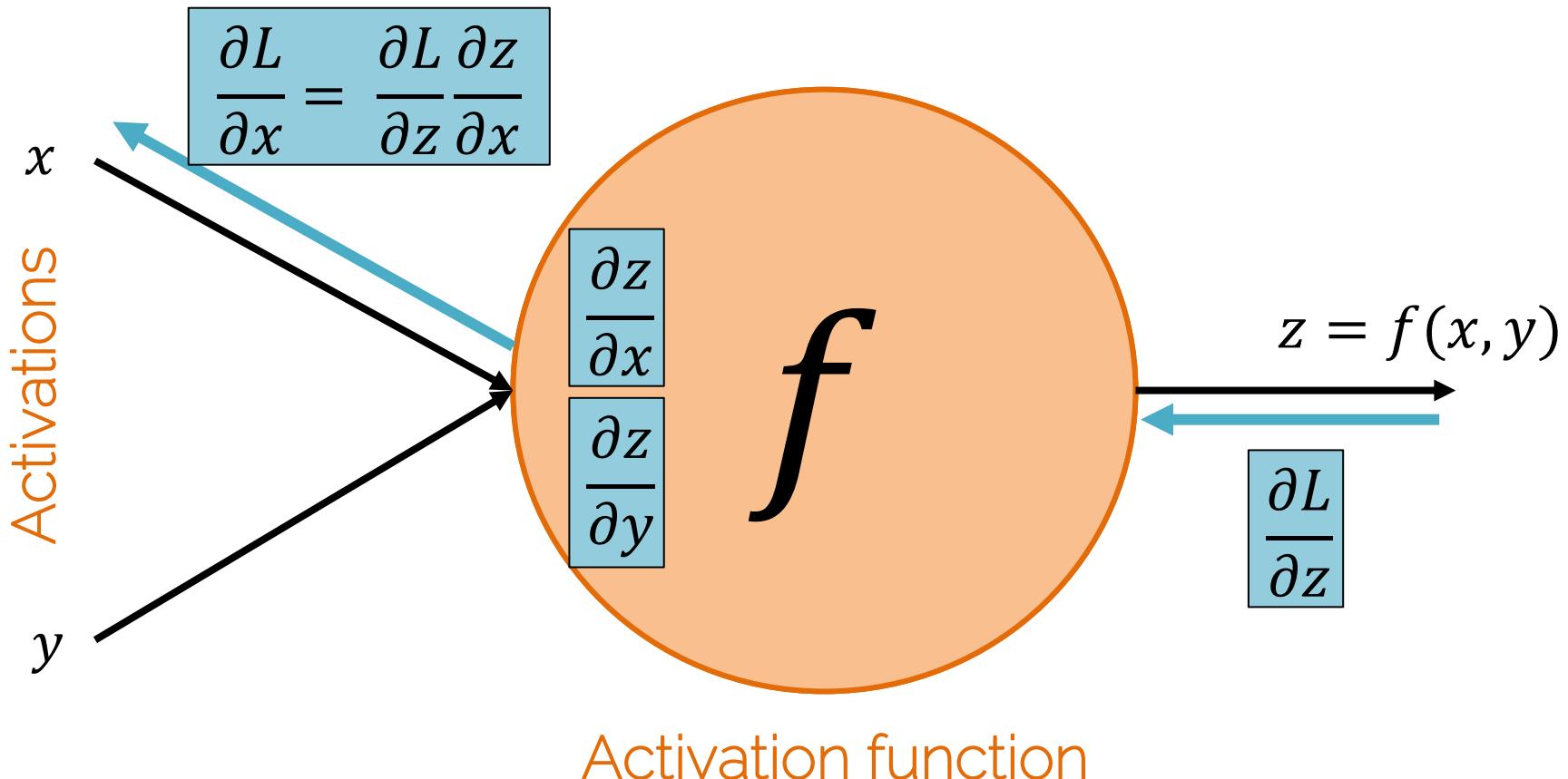
- Many many many many of these nodes form a neural network

NEURONS

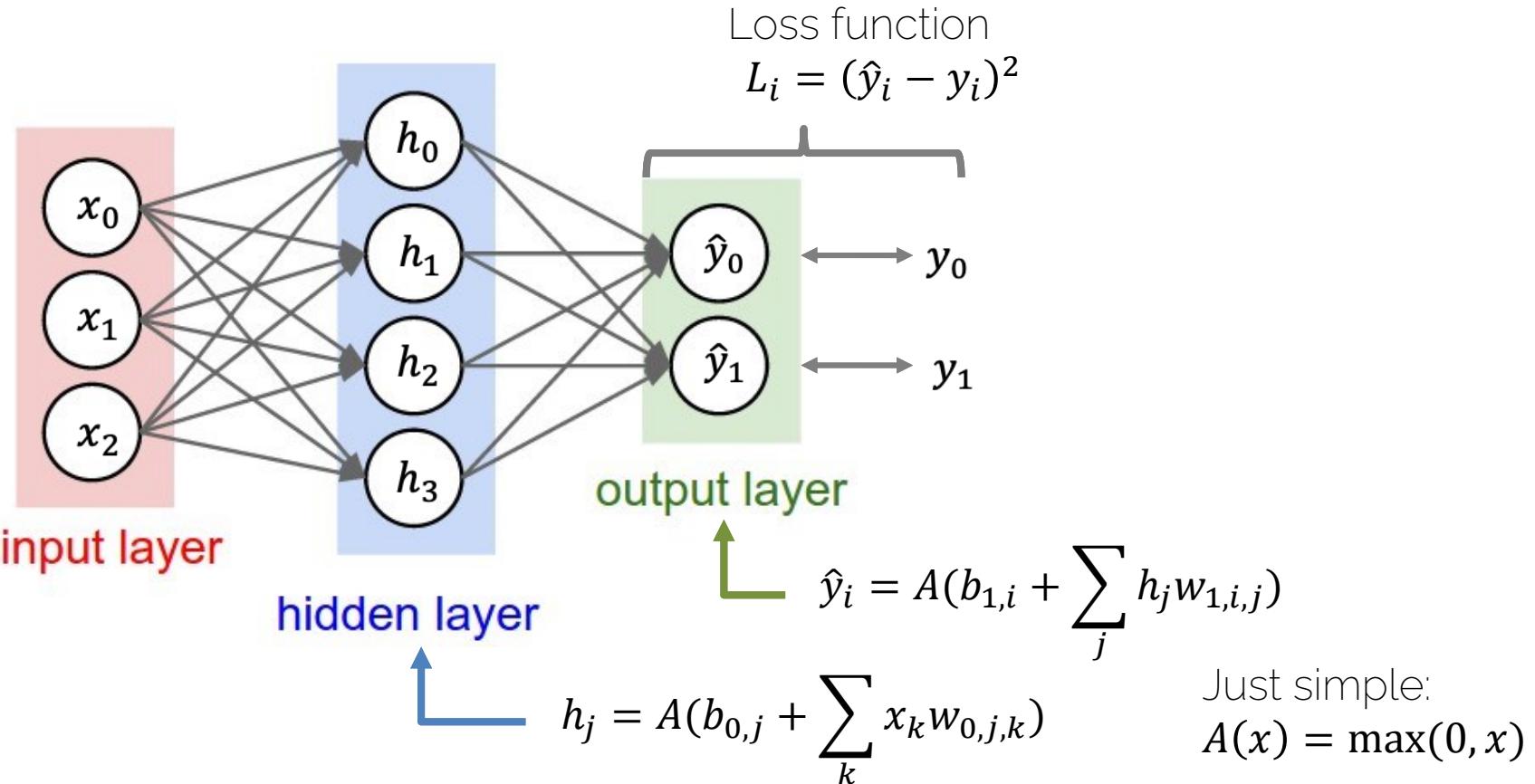
- Each one has its own work to do

FORWARD AND BACKWARD PASS

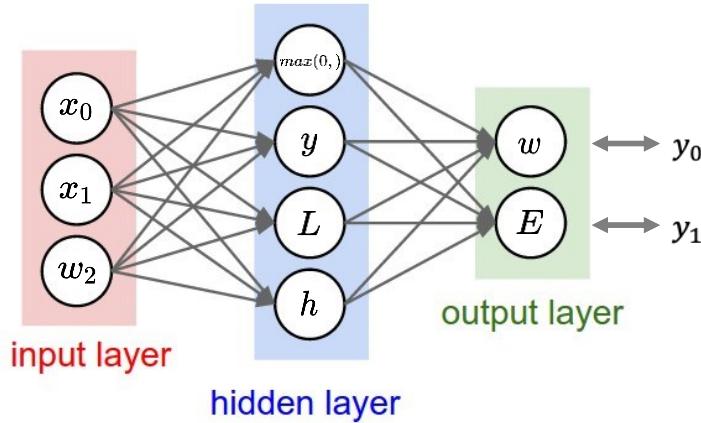
# The Flow of the Gradients



# Gradient Descent for Neural Networks



# Gradient Descent for Neural Networks



$$h_j = A(b_{0,j} + \sum_k x_k w_{0,j,k})$$

$$\hat{y}_i = A(b_{1,i} + \sum_j h_j w_{1,i,j})$$

$$L_i = (\hat{y}_i - y_i)^2$$

Just go through layer by layer

Backpropagation

$$\frac{\partial L}{\partial w_{1,i,j}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{1,i,j}}$$

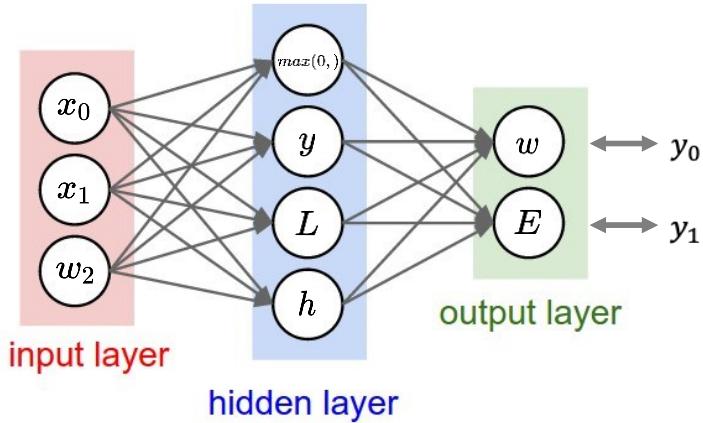
$$\frac{\partial L_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

$$\frac{\partial \hat{y}_i}{\partial w_{1,i,j}} = h_j \quad \text{if } > 0, \text{ else } 0$$

$$\frac{\partial L}{\partial w_{0,j,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial h_j} \cdot \frac{\partial h_j}{\partial w_{0,j,k}}$$

...

# Gradient Descent for Neural Networks



$$h_j = A(b_{0,j} + \sum_k x_k w_{0,j,k})$$

$$\hat{y}_i = A(b_{1,i} + \sum_j h_j w_{1,i,j})$$

$$L_i = (\hat{y}_i - y_i)^2$$

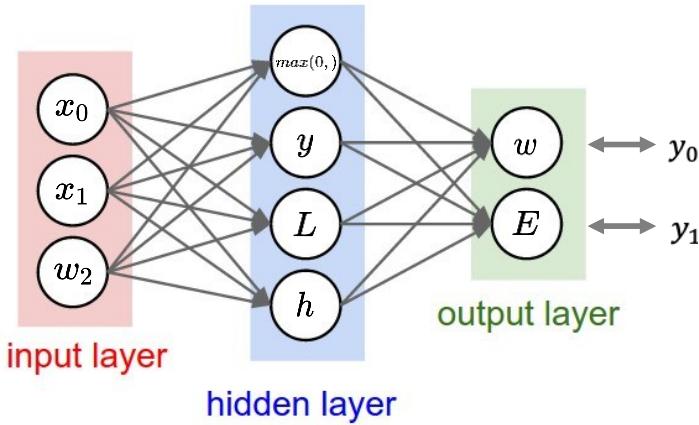
How many unknown weights?

- Output layer:  $2 \cdot 4 + 2$
- Hidden Layer:  $4 \cdot 3 + 4$

#neurons  $\cdot$  #input channels + #biases

Note that some activations have also weights

# Derivatives of Cross Entropy Loss



Binary Cross Entropy loss

$$L = - \sum_{i=1}^{n_{out}} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

$$\hat{y}_i = \frac{1}{1 + e^{-s_i}} \quad s_i = \sum_j h_j w_{ji}$$

output                            scores

Gradients of weights of last layer:

$$\frac{\partial L}{\partial w_{ji}} = \boxed{\frac{\partial L}{\partial \hat{y}_i}} \cdot \boxed{\frac{\partial \hat{y}_i}{\partial s_i}} \cdot \boxed{\frac{\partial s_i}{\partial w_{ji}}}$$

$$\boxed{\frac{\partial L}{\partial \hat{y}_i}} = \frac{-y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)},$$

$$\boxed{\frac{\partial \hat{y}_i}{\partial s_i}} = \hat{y}_i (1 - \hat{y}_i),$$

$$\boxed{\frac{\partial s_i}{\partial w_{ji}}} = h_j$$

$$\Rightarrow \frac{\partial L}{\partial w_{ji}} = (\hat{y}_i - y_i)h_j, \quad \frac{\partial L}{\partial s_i} = \hat{y}_i - y_i$$

# Derivatives of Cross Entropy Loss

Gradients of weights of first layer:

$$\frac{\partial L}{\partial h_j} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_j} \frac{\partial s_j}{\partial h_j} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i} \hat{y}_i (1 - \hat{y}_i) w_{ji} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji}$$

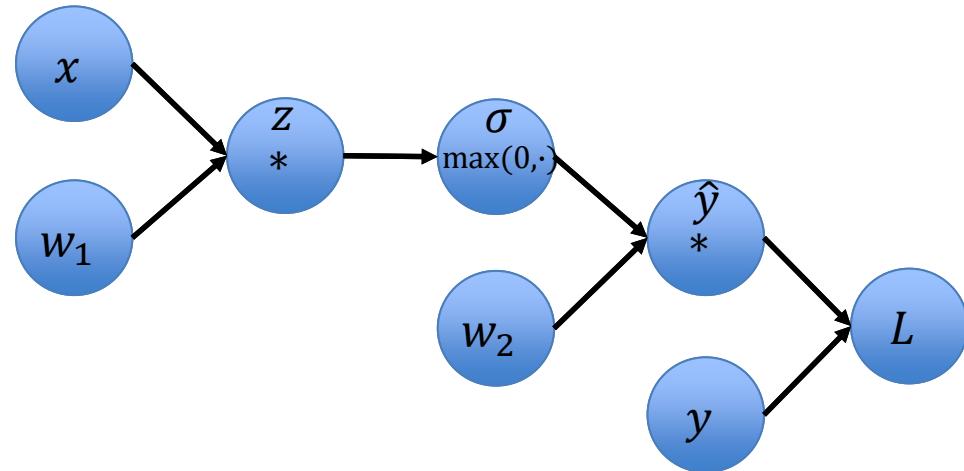
$$\frac{\partial L}{\partial s_j^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_i} \frac{\partial s_i}{\partial h_j} \frac{\partial h_j}{\partial s_j^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji} (h_j (1 - h_j))$$

$$\frac{\partial L}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_j^1} \frac{\partial s_j^1}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji} (h_j (1 - h_j)) x_k$$

# Back to Compute Graphs & NNs

- Inputs  $\mathbf{x}$  and targets  $\mathbf{y}$
- Two-layer NN for regression with ReLU activation
- Function we want to optimize:

$$\sum_{i=1}^n \|w_2 \max(0, w_1 x_i) - y_i\|_2^2$$



# Gradient Descent for Neural Networks

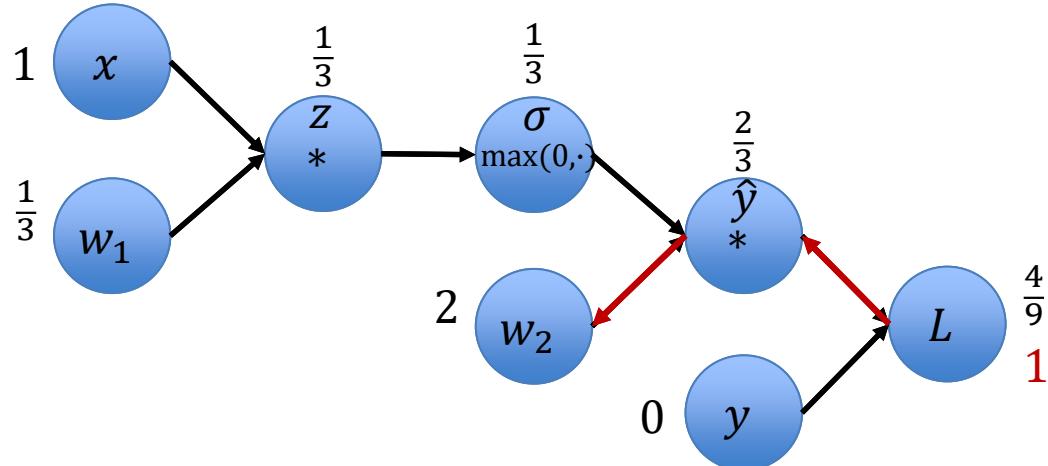
Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n ||\hat{y}_i - y_i||^2$$

In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \quad \Rightarrow \frac{\partial \hat{y}}{\partial w_2} = \sigma$$



Backpropagation

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}$$

# Gradient Descent for Neural Networks

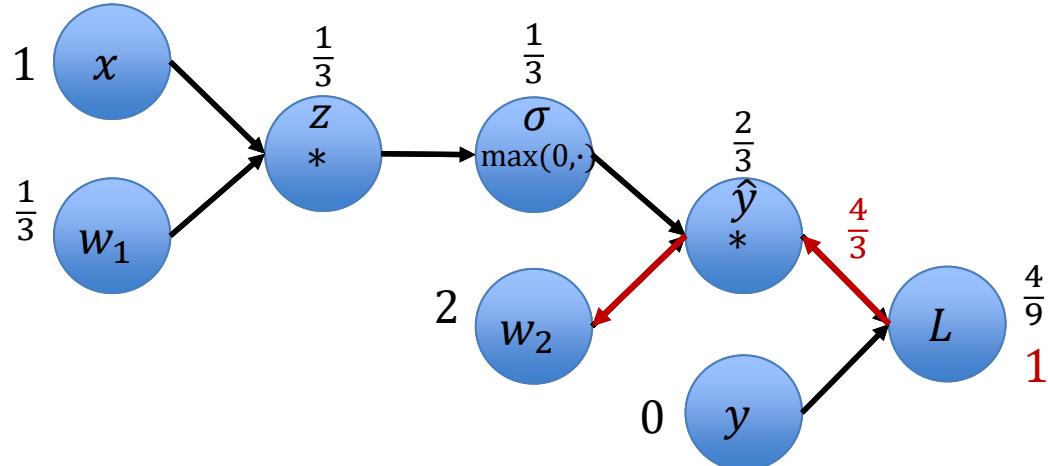
Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n ||\hat{y}_i - y_i||^2$$

In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \quad \Rightarrow \frac{\partial \hat{y}}{\partial w_2} = \sigma$$



Backpropagation

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}$$

$$2 \cdot \frac{2}{3}$$

# Gradient Descent for Neural Networks

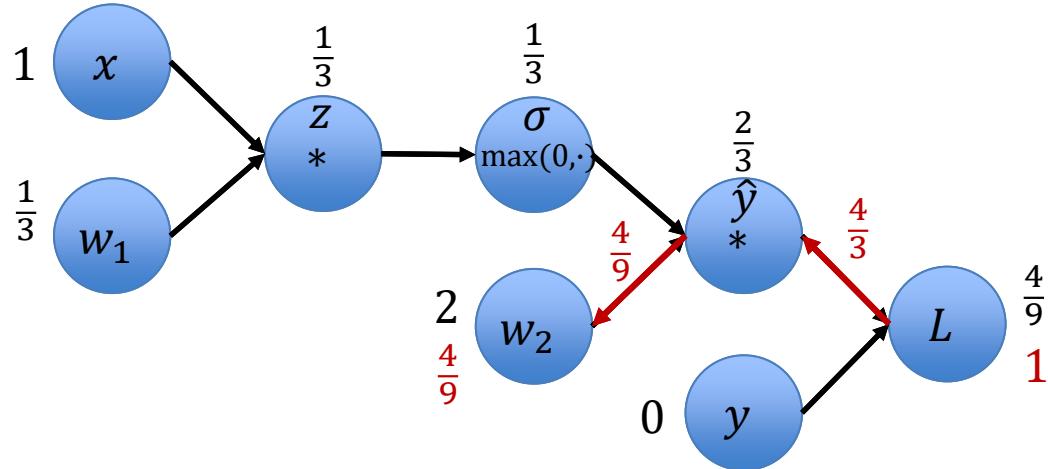
Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

$$L(\mathbf{y}, \hat{\mathbf{y}}; \boldsymbol{\theta}) = \frac{1}{n} \sum_i^n ||\hat{y}_i - y_i||^2$$

In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \quad \Rightarrow \boxed{\frac{\partial \hat{y}}{\partial w_2} = \sigma}$$



Backpropagation

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}$$

$$2 \cdot \frac{2}{3} \cdot \frac{1}{3}$$

# Gradient Descent for Neural Networks

Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

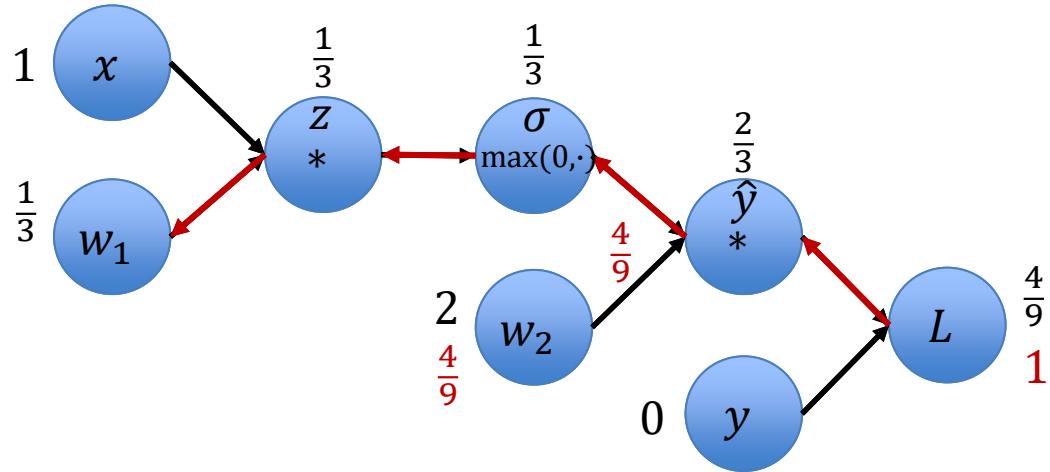
In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \Rightarrow \frac{\partial \hat{y}}{\partial \sigma} = w_2$$

$$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{else} \end{cases}$$

$$z = x \cdot w_1 \Rightarrow \frac{\partial z}{\partial w_1} = x$$



Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

# Gradient Descent for Neural Networks

Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

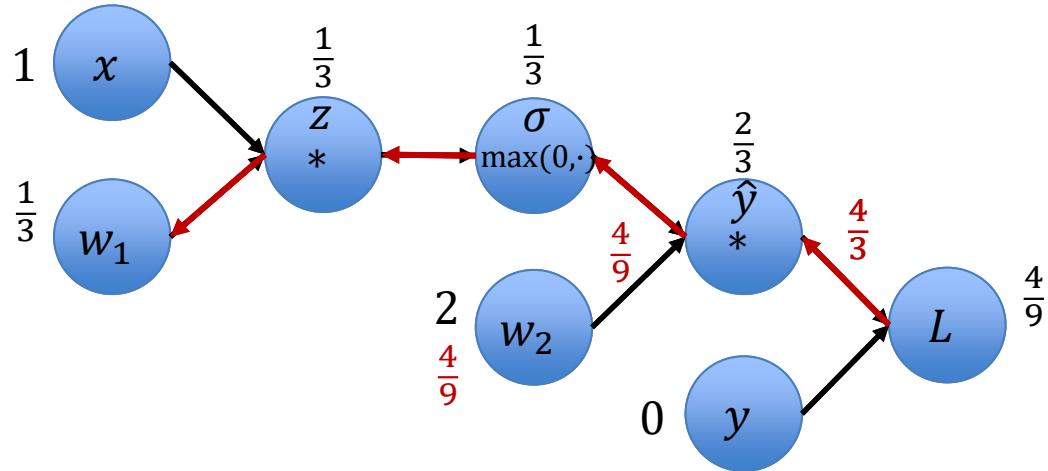
In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \Rightarrow \frac{\partial \hat{y}}{\partial \sigma} = w_2$$

$$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

$$z = x \cdot w_1 \Rightarrow \frac{\partial z}{\partial w_1} = x$$



Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$2 \cdot \frac{2}{3}$$

# Gradient Descent for Neural Networks

Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

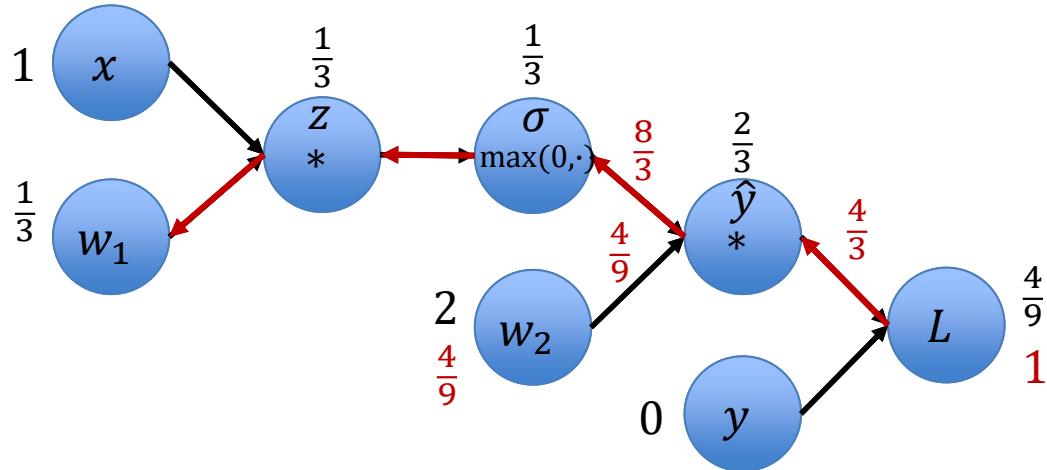
In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \Rightarrow \boxed{\frac{\partial \hat{y}}{\partial \sigma} = w_2}$$

$$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{else} \end{cases}$$

$$z = x \cdot w_1 \Rightarrow \frac{\partial z}{\partial w_1} = x$$



Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$2 \cdot \frac{2}{3} \cdot 2$$

# Gradient Descent for Neural Networks

Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

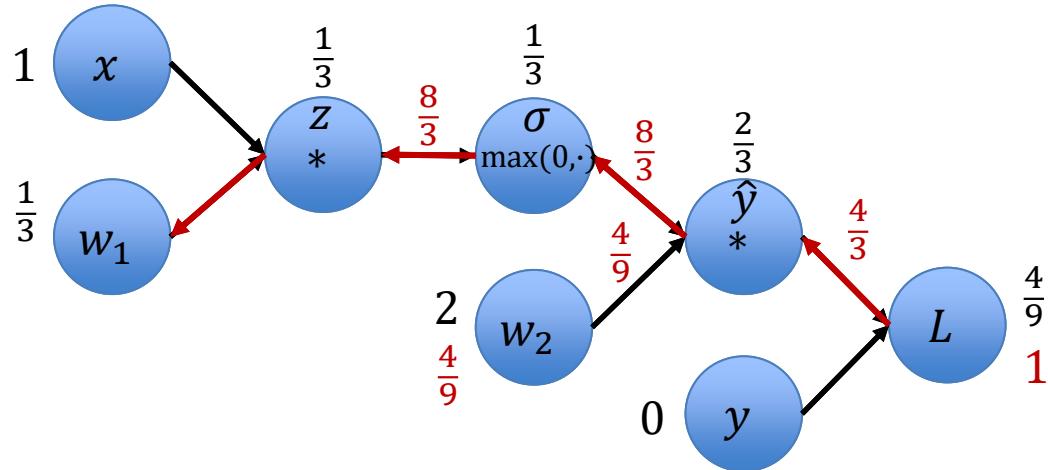
In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \Rightarrow \frac{\partial \hat{y}}{\partial \sigma} = w_2$$

$$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{else} \end{cases}$$

$$z = x \cdot w_1 \Rightarrow \frac{\partial z}{\partial w_1} = x$$



Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$2 \cdot \frac{2}{3} \cdot 2 \cdot 1$$

# Gradient Descent for Neural Networks

Initialize  $x = 1$ ,  $y = 0$ ,  
 $w_1 = \frac{1}{3}$ ,  $w_2 = 2$

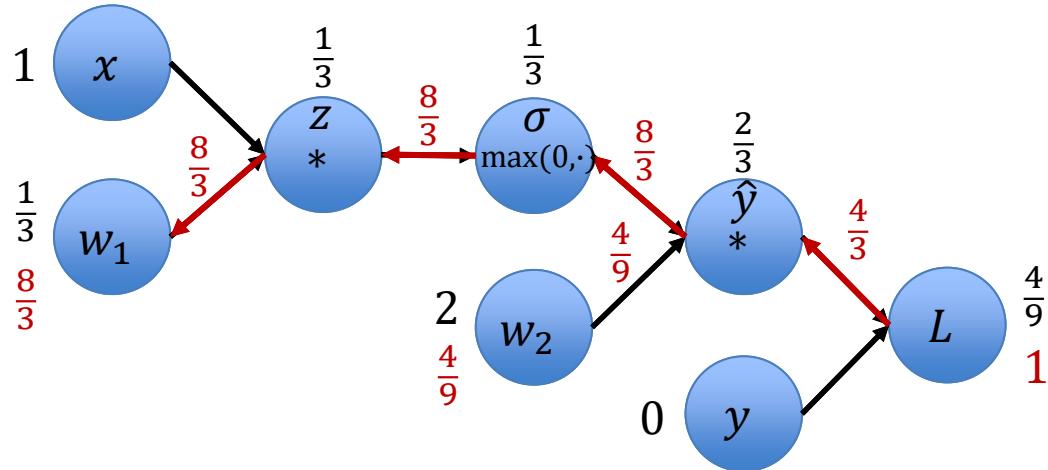
In our case  $n, d = 1$ :

$$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\hat{y} = w_2 \cdot \sigma \Rightarrow \frac{\partial \hat{y}}{\partial \sigma} = w_2$$

$$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

$$z = x \cdot w_1 \Rightarrow \boxed{\frac{\partial z}{\partial w_1} = x}$$



Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$2 \cdot \frac{2}{3} \cdot 2 \cdot 1 \cdot 1$$

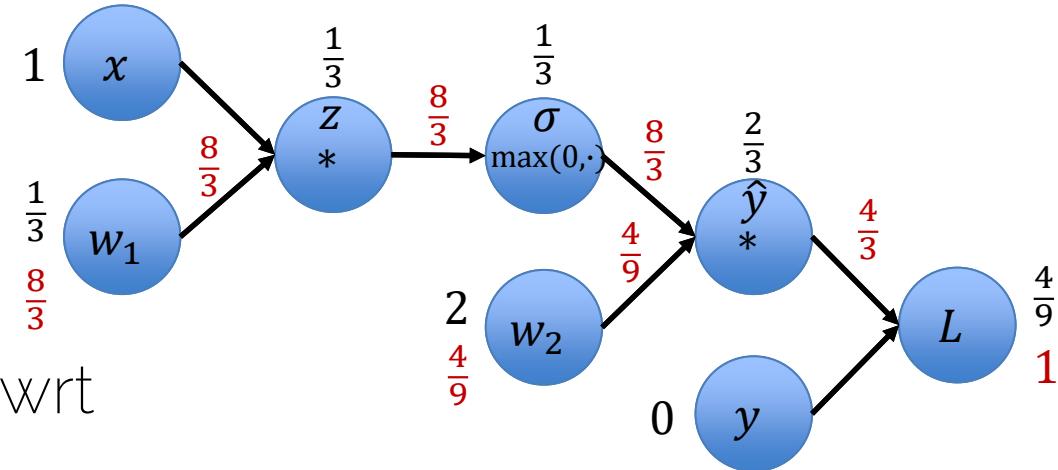
# Gradient Descent for Neural Networks

- Function we want to optimize:

$$f(x, \mathbf{w}) = \sum_{i=1}^n \|w_2 \max(0, w_1 x_i) - y_i\|_2^2$$

- Computed gradients wrt to weights  $\mathbf{w}_1$  and  $\mathbf{w}_2$
- Now: update the weights

$$\begin{aligned}\mathbf{w}' &= \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} f = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \alpha \cdot \begin{pmatrix} \nabla_{w_1} f \\ \nabla_{w_2} f \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{3} \\ 2 \end{pmatrix} - \alpha \cdot \begin{pmatrix} \frac{8}{3} \\ \frac{4}{9} \end{pmatrix}\end{aligned}$$



But: how to choose a good learning rate  $\alpha$  ?

# Gradient Descent

- How to pick good learning rate?
- How to compute gradient for single training pair?
- How to compute gradient for large training set?
- How to speed things up? More to see in next lectures...

# Regularization

# Recap: Basic Recipe for ML

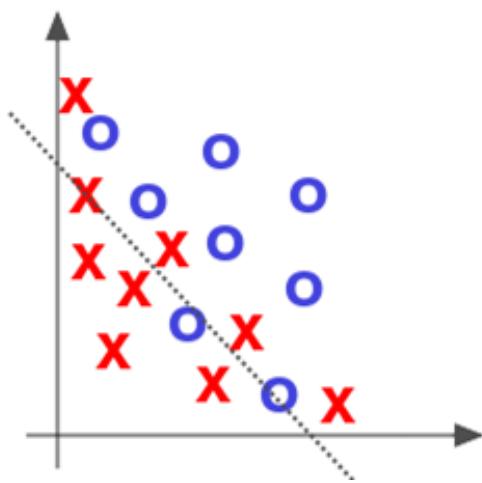
- Split your data



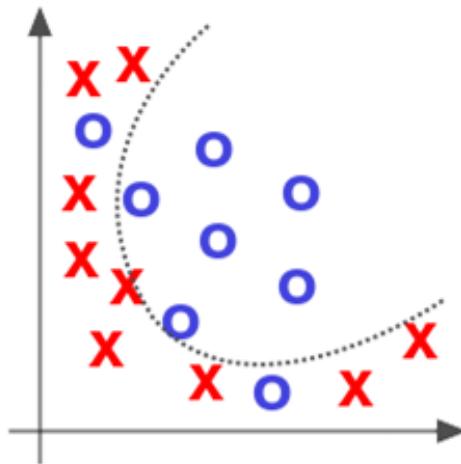
Find your hyperparameters

Other splits are also possible (e.g., 80%/10%/10%)

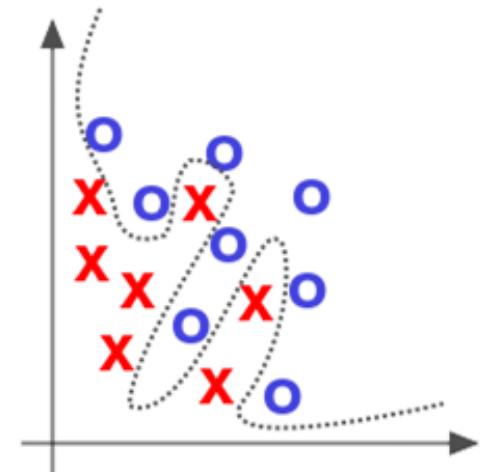
# Over- and Underfitting



Underfitted



Appropriate

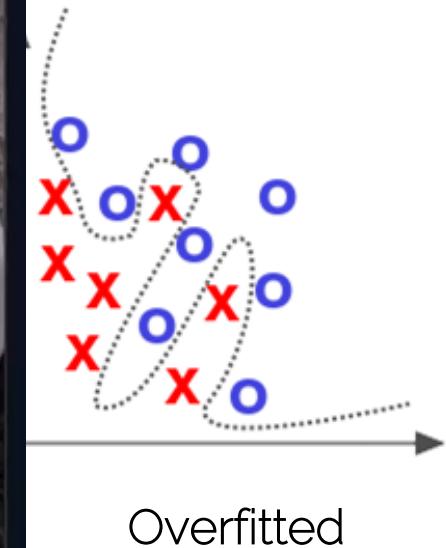
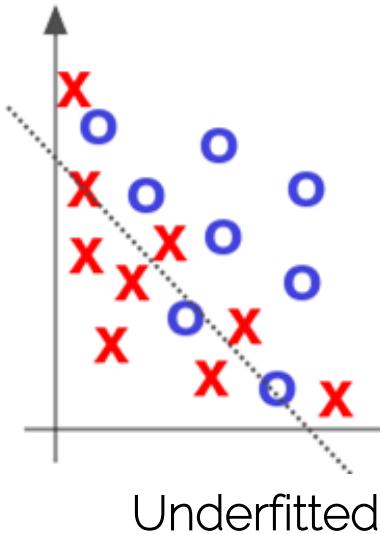


Overfitted

Source: Deep Learning by Adam Gibson, Josh Patterson, O'Reilly Media Inc., 2017

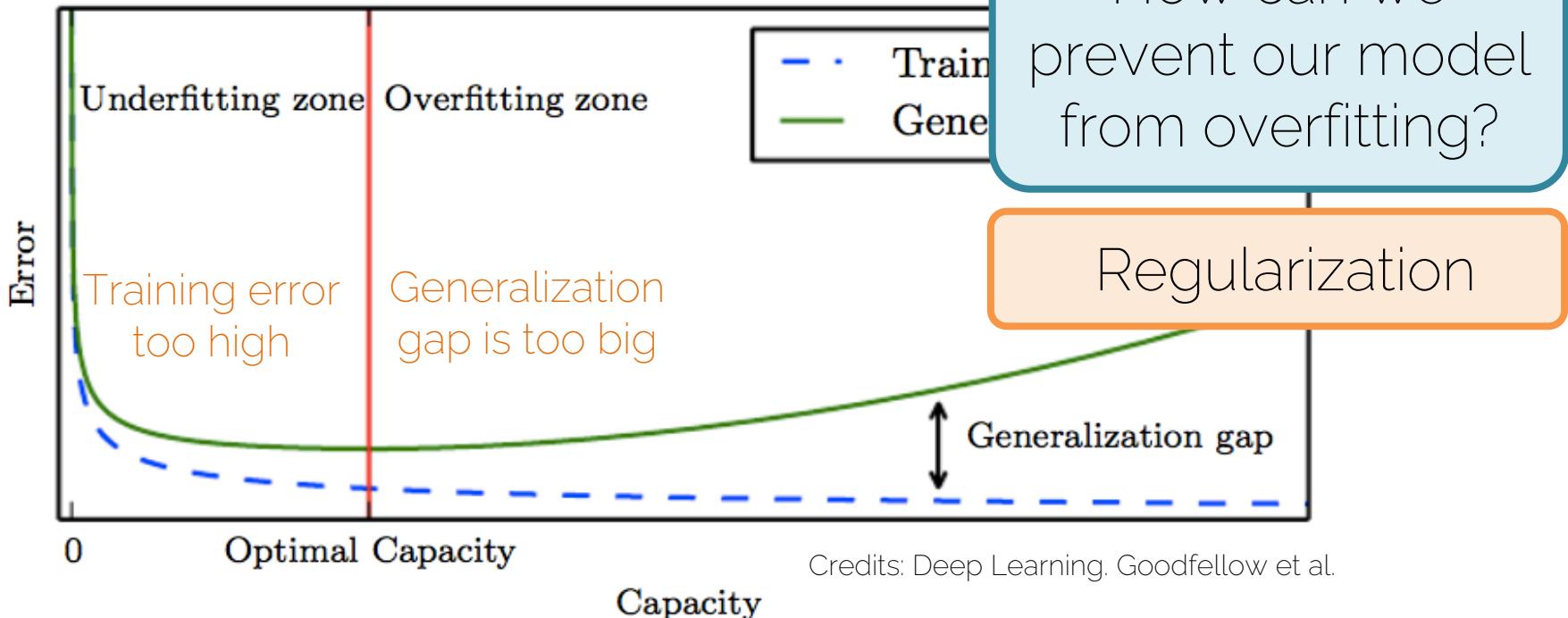
# Over- and Underfitting

ML Engineers looking at their classification model running on the test set.



# Training a Neural Network

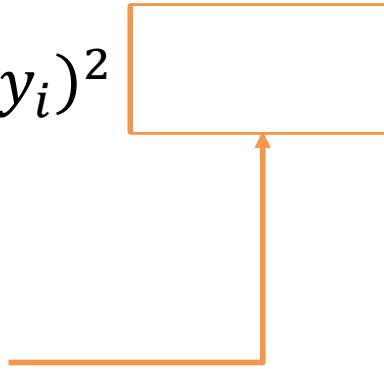
- Training/ Validation curve



# Regularization

- Loss function  $L(\mathbf{y}, \hat{\mathbf{y}}, \boldsymbol{\theta}) = \sum_{i=1}^n (\hat{y}_i - y_i)^2$
- Regularization techniques
  - L<sub>2</sub> regularization
  - L<sub>1</sub> regularization
  - Max norm regularization
  - Dropout
  - Early stopping
  - ...

Add regularization term to loss function



# Regularization

- Loss function  $L(\mathbf{y}, \hat{\mathbf{y}}, \boldsymbol{\theta}) = \sum_{i=1}^n (\hat{y}_i - y_i)^2 + \lambda R(\boldsymbol{\theta})$
  - Regularization techniques
    - L2 regularization
    - L1 regularization
    - Max norm regularization
    - Dropout
    - Early stopping
    - ...
- 
- The diagram features a large orange bracket on the right side of the slide. It spans from the 'Add regularization term to loss function' label down to the ellipsis '...'. An orange arrow points upwards from the bottom of this bracket to the  $\lambda R(\boldsymbol{\theta})$  term in the loss function equation.

# Regularization: Example

- Input: 3 features  $\mathbf{x} = [1, 2, 1]$
- Two linear classifiers that give the same result:
  - $\theta_1 = [0, 0.75, 0]$   Ignores 2 features
  - $\theta_2 = [0.25, 0.5, 0.25]$   Takes information from all features

# Regularization: Example

- Loss  $L(\mathbf{y}, \hat{\mathbf{y}}, \boldsymbol{\theta}) = \sum_{i=1}^n (x_i \theta_{ji} - y_i)^2 + \lambda R(\boldsymbol{\theta})$

- L2 regularization  $R(\boldsymbol{\theta}) = \sum_{i=1}^n \theta_i^2$

$$\theta_1 \longrightarrow 0 + 0.75^2 + 0 = 0.5625$$

$$\theta_2 \longrightarrow 0.25^2 + 0.5^2 + 0.25^2 = 0.375 \quad \text{Minimization}$$

$$x = [1, 2, 1], \theta_1 = [0, 0.75, 0], \theta_2 = [0.25, 0.5, 0.25]$$

# Regularization: Example

- Loss  $L(\mathbf{y}, \hat{\mathbf{y}}, \boldsymbol{\theta}) = \sum_{i=1}^n (x_i \theta_{ji} - y_i)^2 + \lambda R(\boldsymbol{\theta})$

- L1 regularization  $R(\boldsymbol{\theta}) = \sum_{i=1}^n |\theta_i|$

$$\theta_1 \longrightarrow 0 + 0.75 + 0 = 0.75 \quad \text{Minimization}$$

$$\theta_2 \longrightarrow 0.25 + 0.5 + 0.25 = 1$$

$$x = [1, 2, 1], \theta_1 = [0, 0.75, 0], \theta_2 = [0.25, 0.5, 0.25]$$

# Regularization: Example

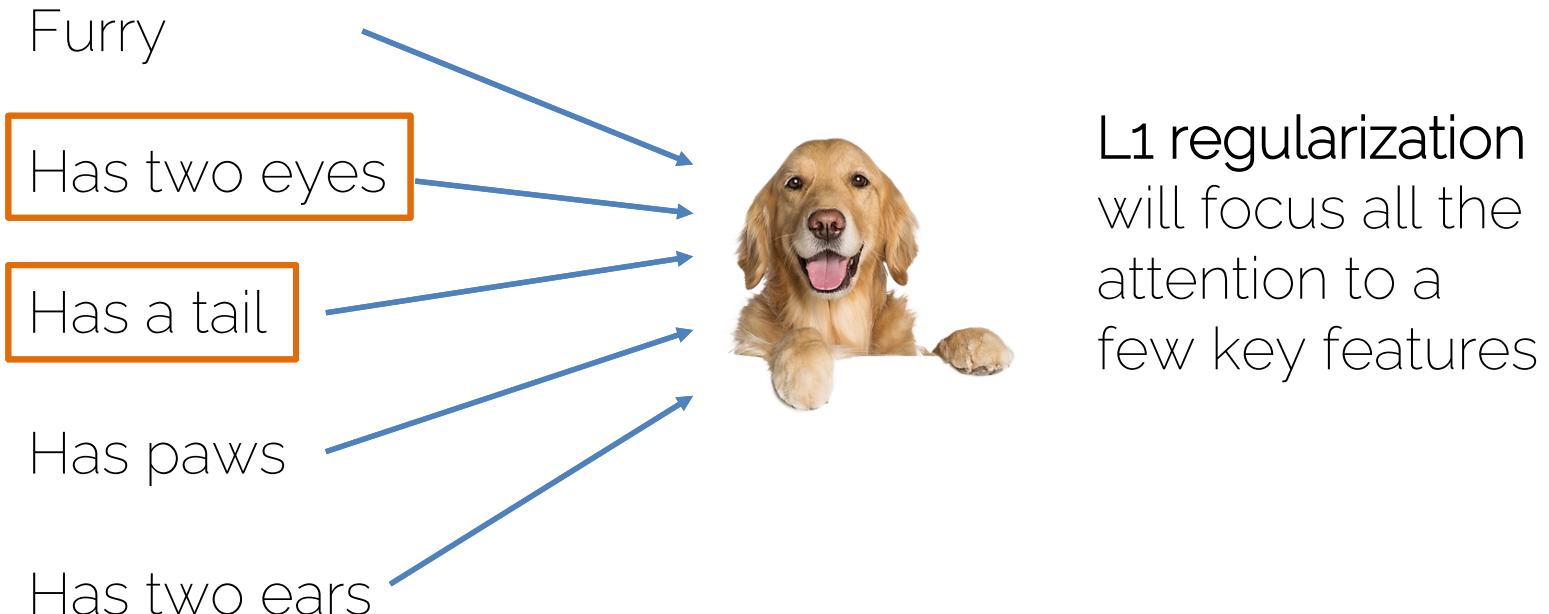
- Input: 3 features  $\mathbf{x} = [1, 2, 1]$
- Two linear classifiers that give the same result:
- $\theta_1 = [0, 0.75, 0]$   Ignores 2 features
- $\theta_2 = [0.25, 0.5, 0.25]$   Takes information from all features

# Regularization: Example

- Input: 3 features  $\mathbf{x} = [1, 2, 1]$
- Two linear classifiers that give the same result:
- $\theta_1 = [0, 0.75, 0]$   L1 regularization enforces **sparsity**
- $\theta_2 = [0.25, 0.5, 0.25]$   L2 regularization enforces that the weights have **similar values**

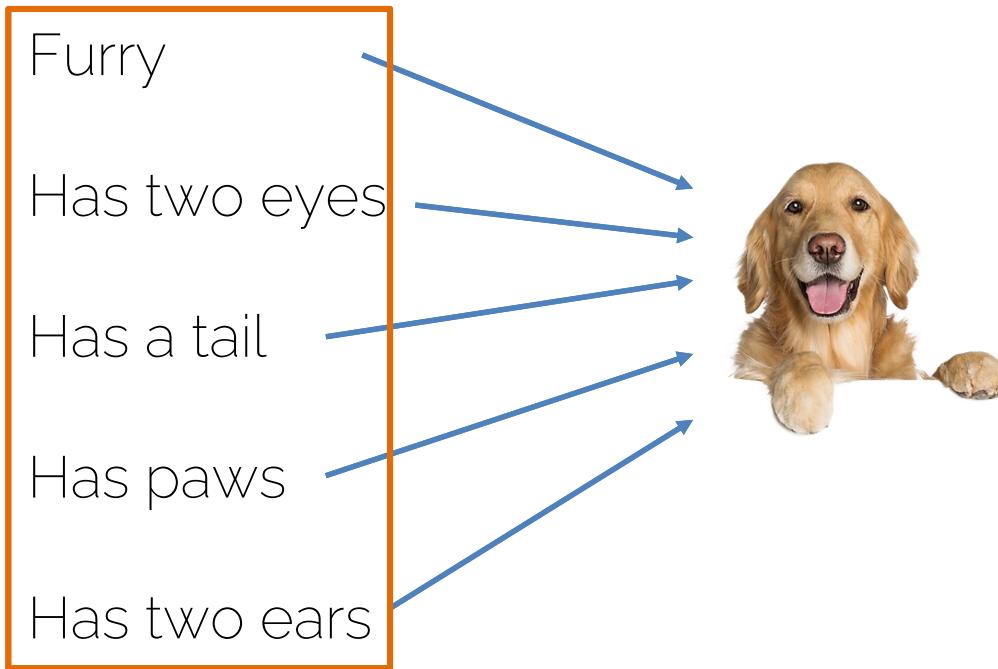
# Regularization: Effect

- Dog classifier takes different inputs



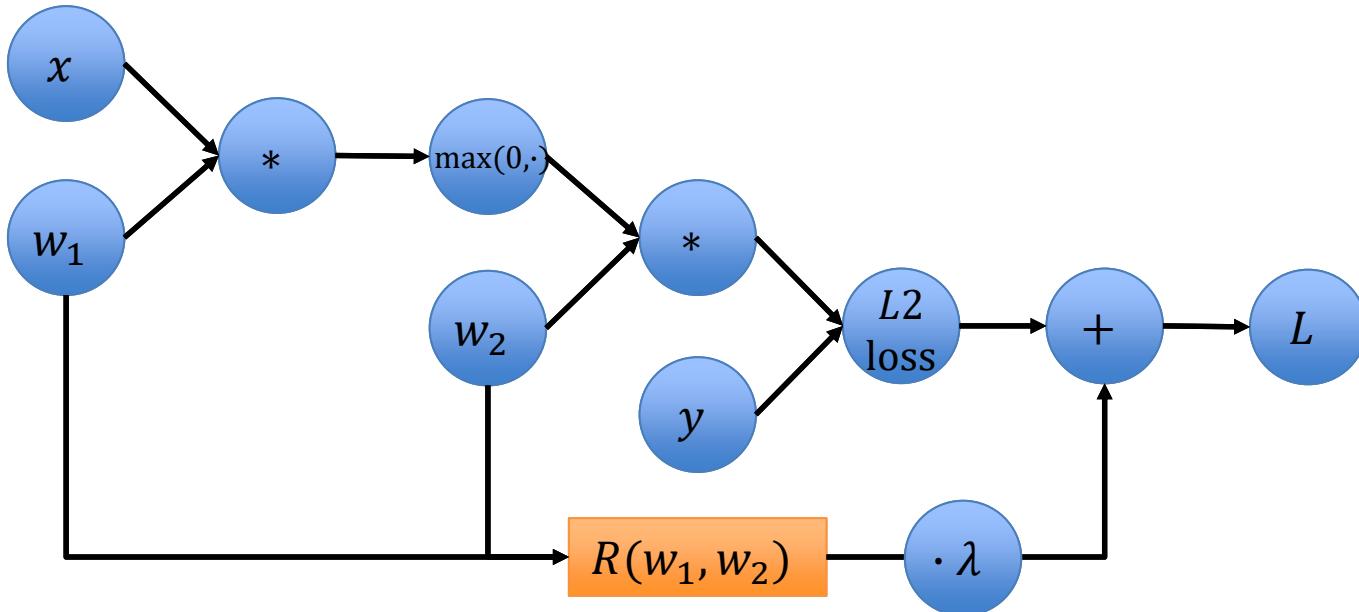
# Regularization: Effect

- Dog classifier takes different inputs



L2 regularization will take all information into account to make decisions

# Regularization for Neural Networks

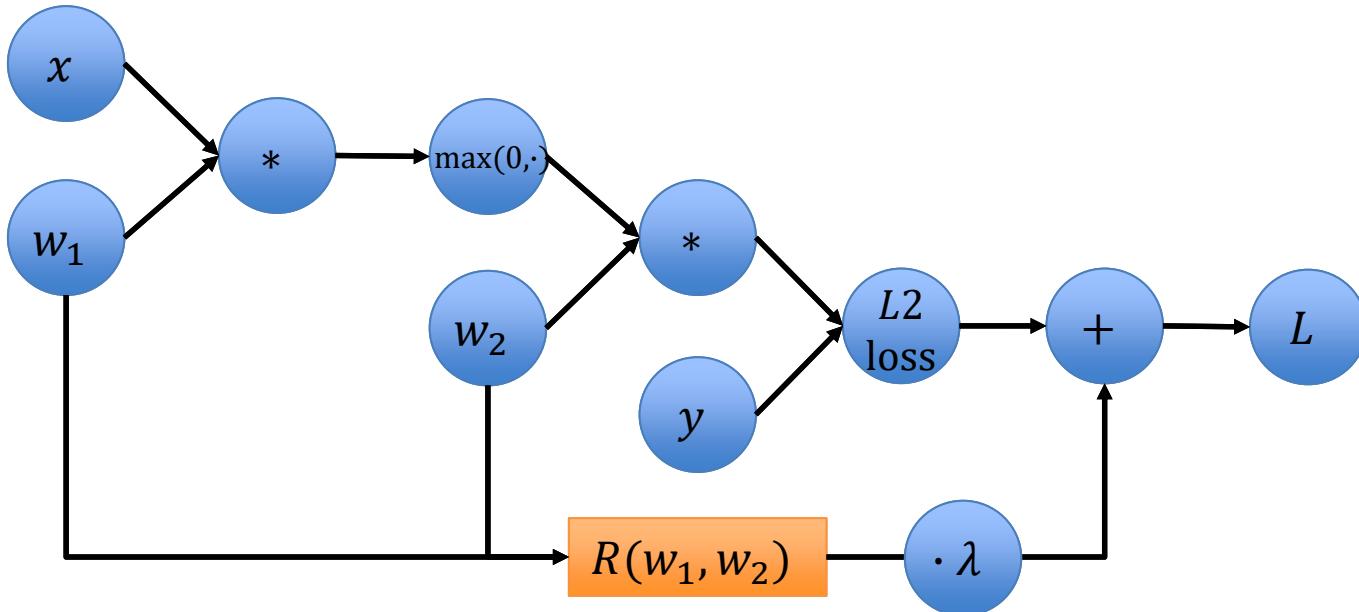


Combining nodes:

Network output + L2-loss +  
regularization

$$\sum_{i=1}^n \|w_2 \max(0, w_1 x_i) - y_i\|_2^2 + \lambda R(w_1, w_2)$$

# Regularization for Neural Networks

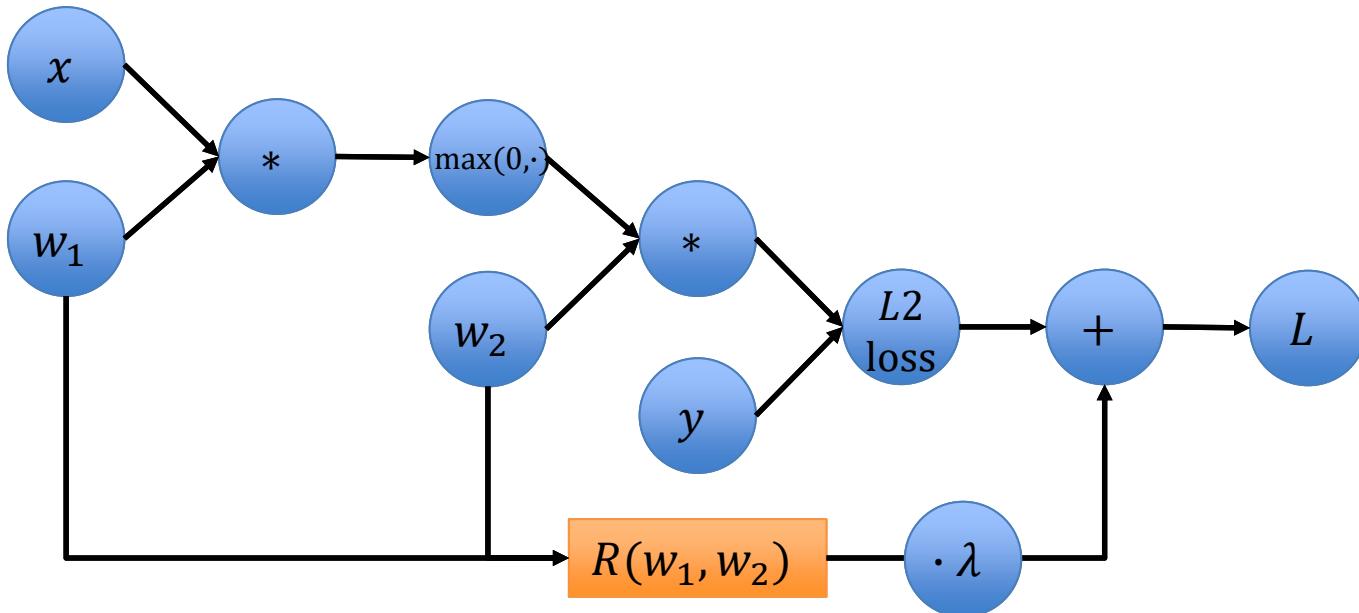


Combining nodes:

Network output + L2-loss +  
regularization

$$\sum_{i=1}^n \|w_2 \max(0, w_1 x_i) - y_i\|_2^2 + \lambda \left\| \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \right\|_2^2$$

# Regularization for Neural Networks

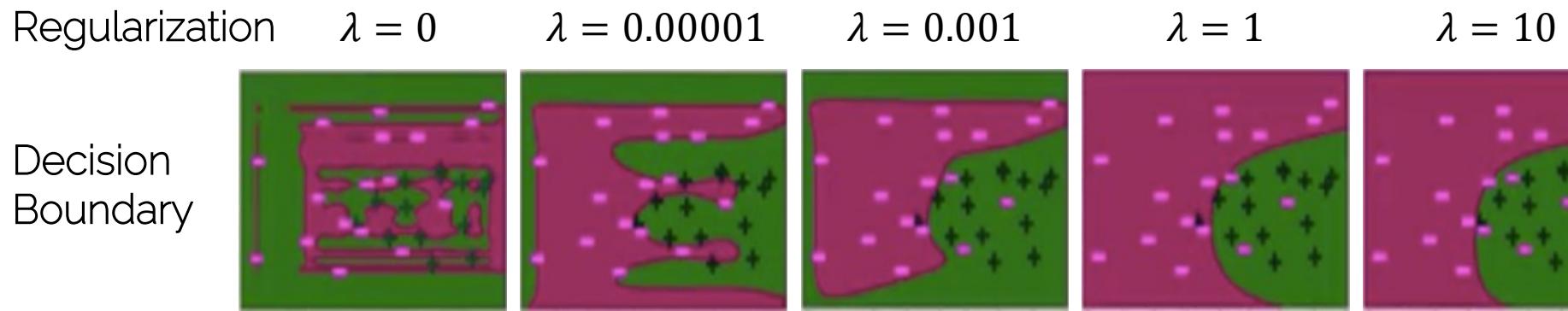


Combining nodes:

Network output + L2-loss +  
regularization

$$\sum_{i=1}^n \|w_2 \max(0, w_1 x_i) - y_i\|_2^2 + \lambda(w_1^2 + w_2^2)$$

# Regularization



Credit: University of Washington

What happens to the training error?

What is the goal of regularization?

# Regularization

- Any strategy that aims to



Lower  
validation error



Increasing  
training error

# Next Lecture

- This week:
  - Check exercises!
  - Check piazza / post questions ☺
- Next lecture
  - Optimization of Neural Networks
  - In particular, introduction to SGD (our main method!)

See you next week 😊

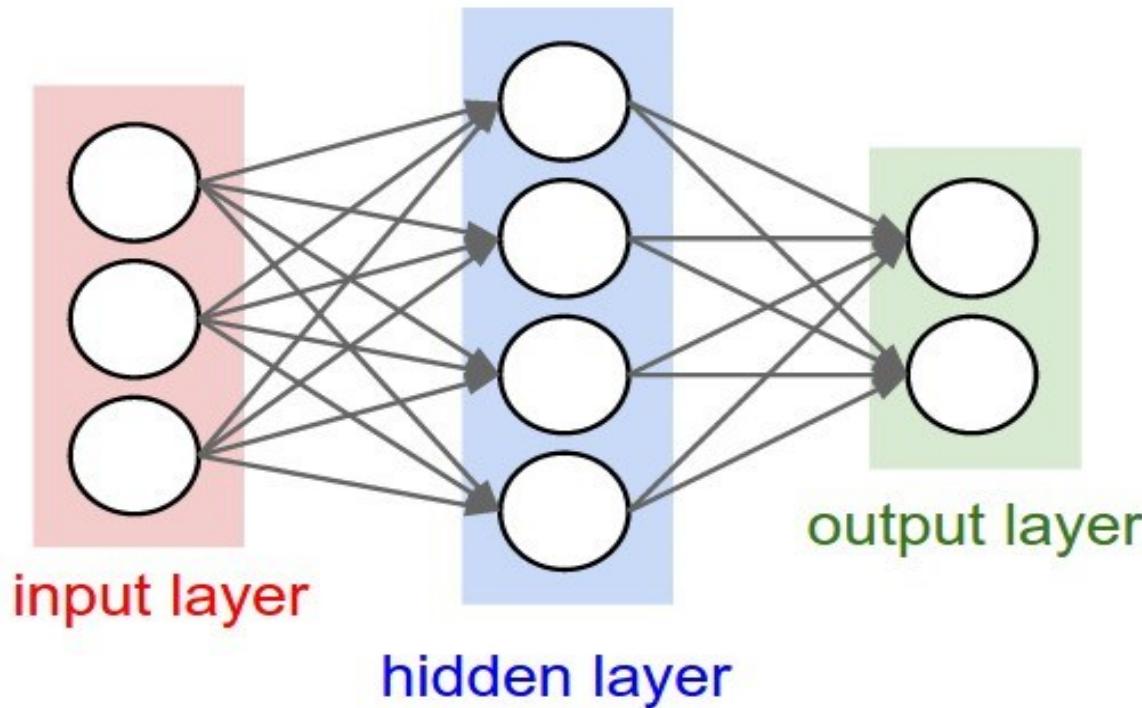
# Further Reading

- Backpropagation
  - Chapter 6.5 (6.5.1 - 6.5.3) in  
<http://www.deeplearningbook.org/contents/mlp.html>
  - Chapter 5.3 in Bishop, Pattern Recognition and Machine Learning
  - <http://cs231n.github.io/optimization-2/>
- Regularization
  - Chapter 7.1 (esp. 7.1.1 & 7.1.2)  
<http://www.deeplearningbook.org/contents/regularization.html>
  - Chapter 5.5 in Bishop, Pattern Recognition and Machine Learning

# Scaling Optimization

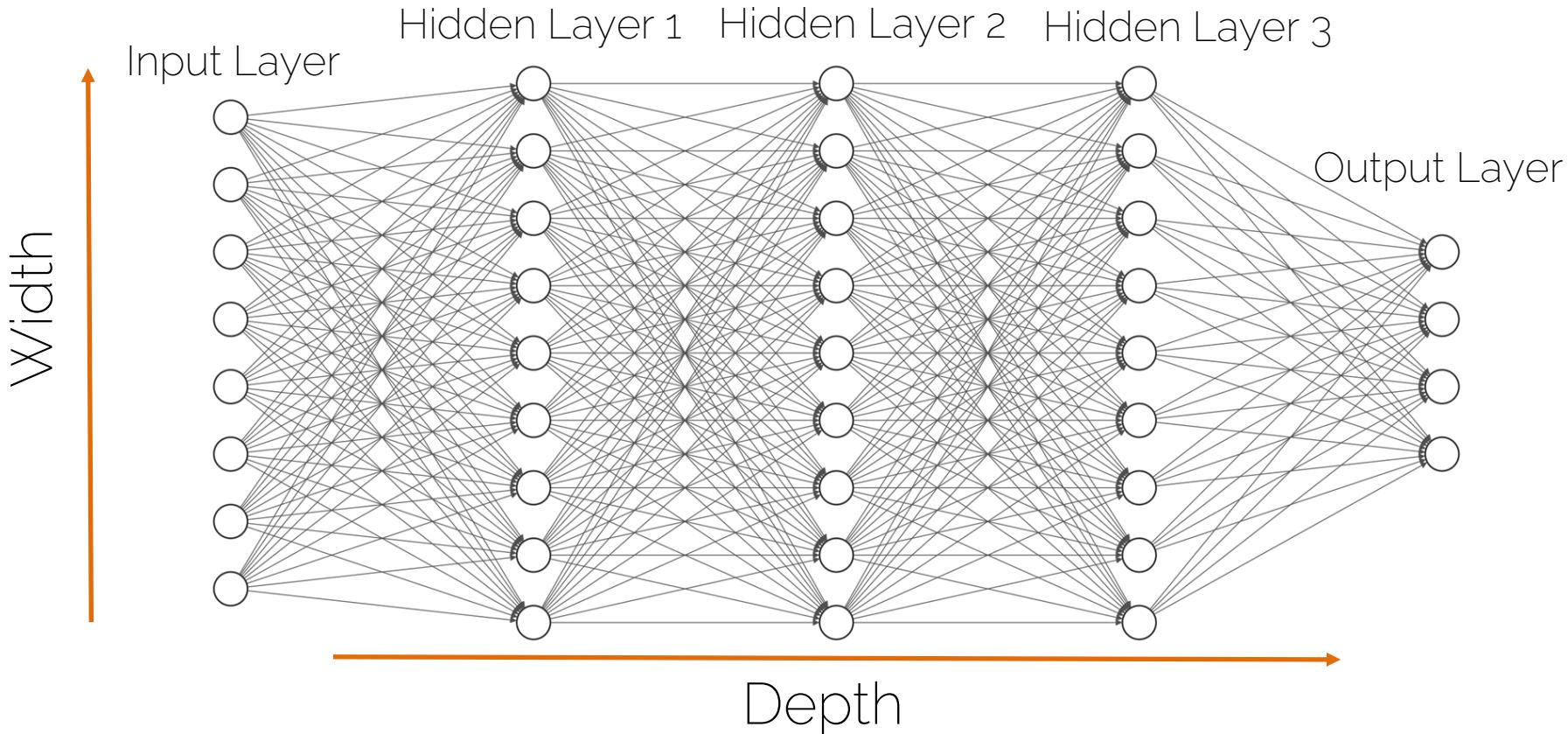
# Lecture 4 Recap

# Neural Network

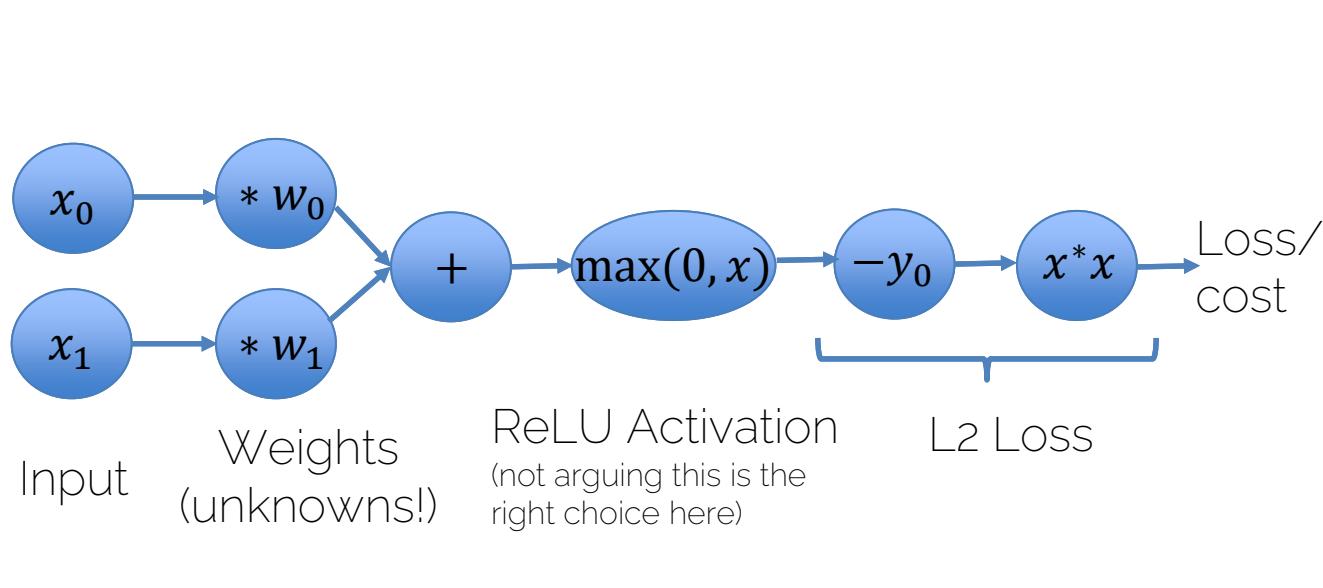
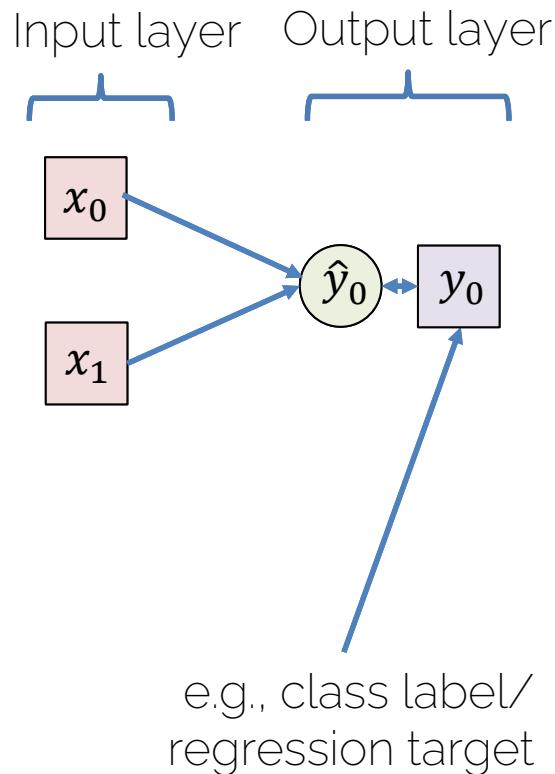


Source: <http://cs231n.github.io/neural-networks-1/>

# Neural Network

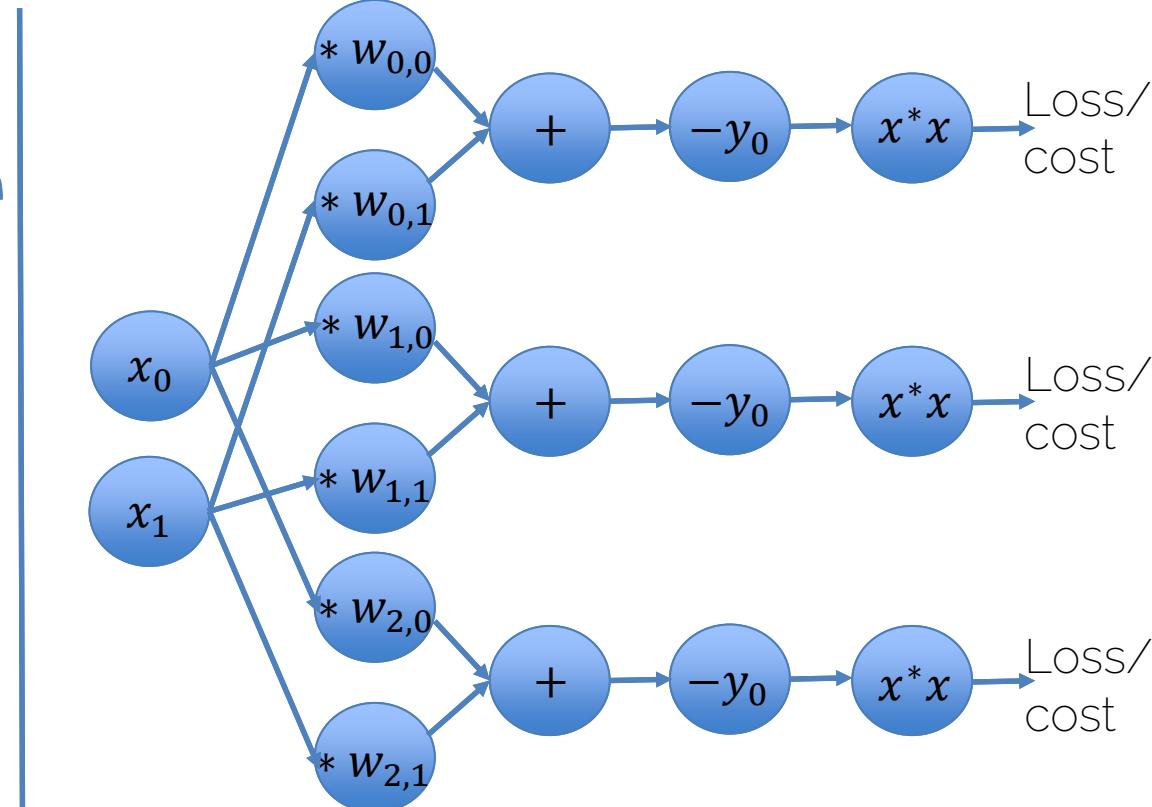
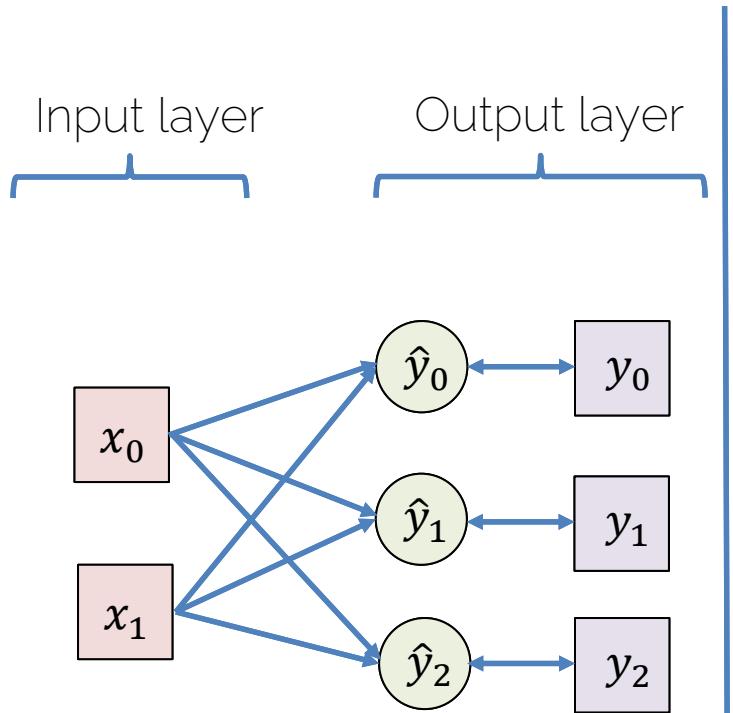


# Compute Graphs → Neural Networks



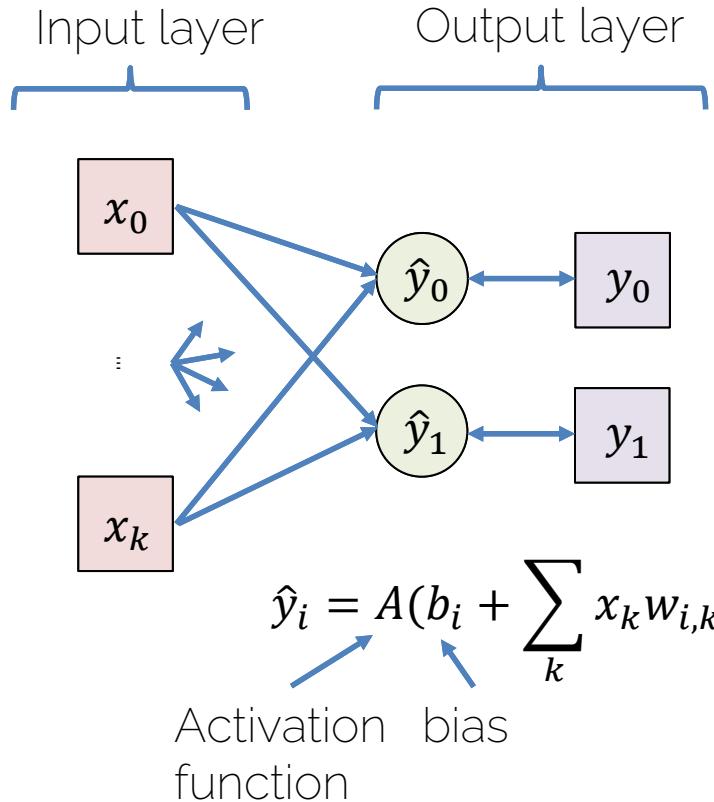
We want to compute gradients w.r.t. all weights  $\mathbf{w}$

# Compute Graphs → Neural Networks



We want to compute gradients w.r.t. all weights  $\mathbf{w}$

# Compute Graphs → Neural Networks



Goal: We want to compute gradients of the loss function  $L$  w.r.t. all weights  $w$

$$L = \sum_i L_i$$

$L$ : sum over loss per sample, e.g.  
L2 loss → simply sum up squares:

$$L_i = (\hat{y}_i - y_i)^2$$

→ use chain rule to compute partials

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

We want to compute gradients w.r.t. all weights  $W$  AND all biases  $b$

# Summary

- We have
  - (Directional) compute graph
  - Structure graph into layers
  - Compute partial derivatives w.r.t. weights (unknowns)
- Next
  - Find weights based on gradients

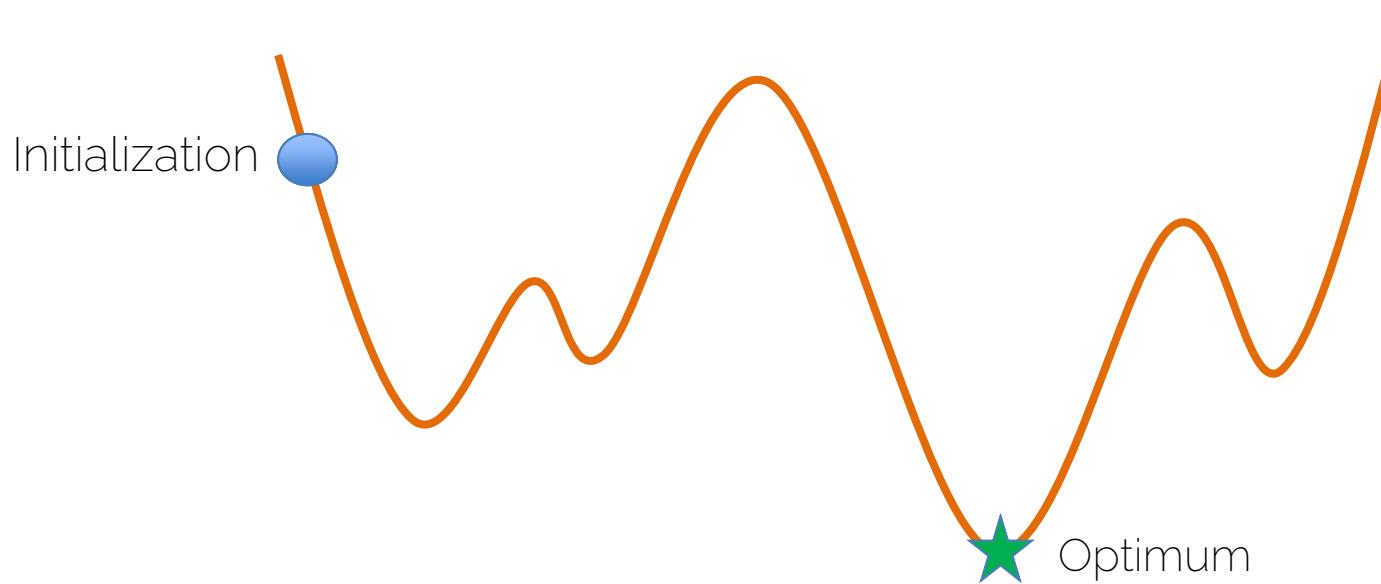
$$\nabla_{\mathbf{W}} f_{\{x,y\}}(\mathbf{W}) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \vdots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \vdots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

Gradient step:  
 $\mathbf{W}' = \mathbf{W} - \alpha \nabla_{\mathbf{W}} f_{\{x,y\}}(\mathbf{W})$

# Optimization

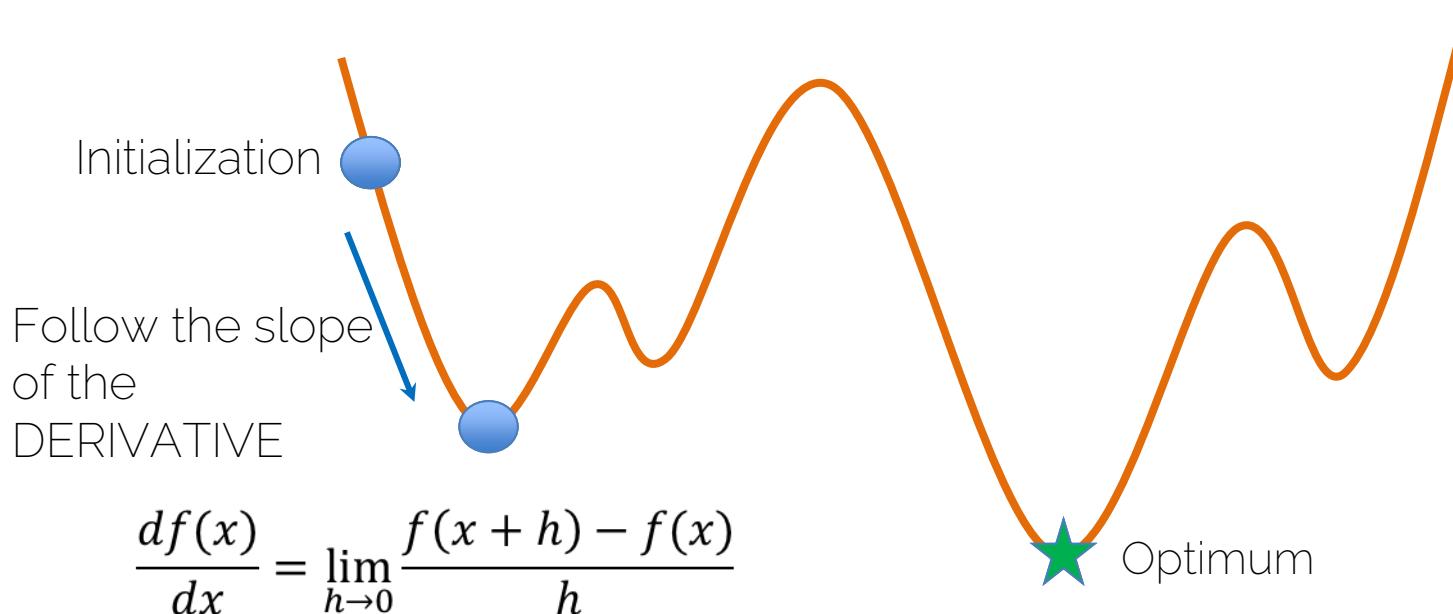
# Gradient Descent

$$x^* = \arg \min f(x)$$



# Gradient Descent

$$x^* = \arg \min f(x)$$



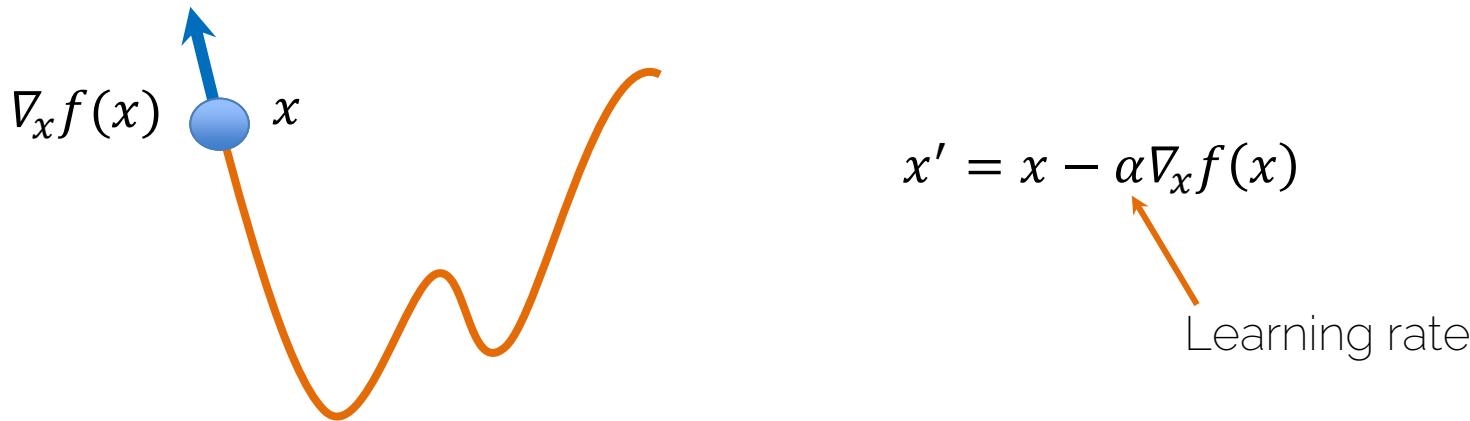
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



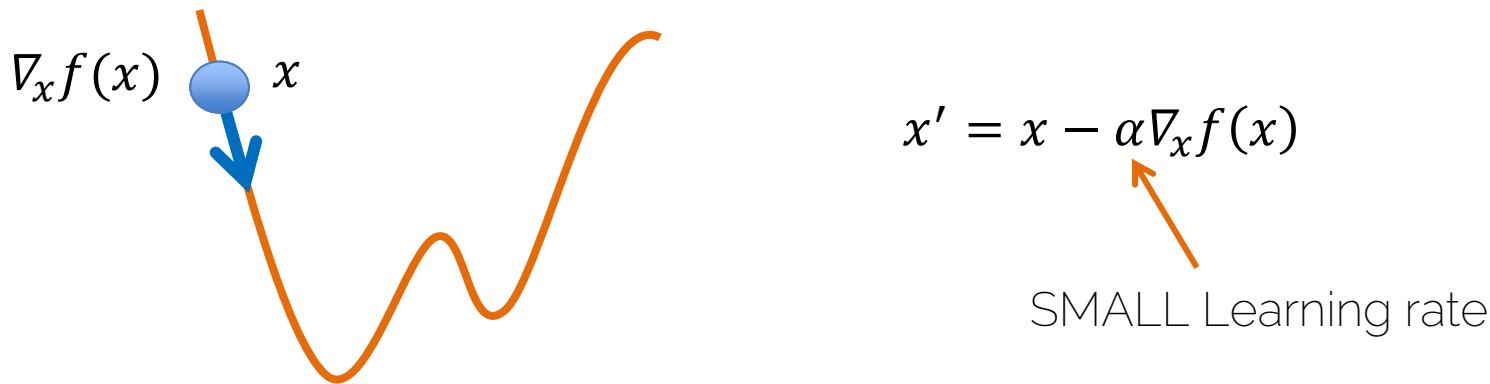
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



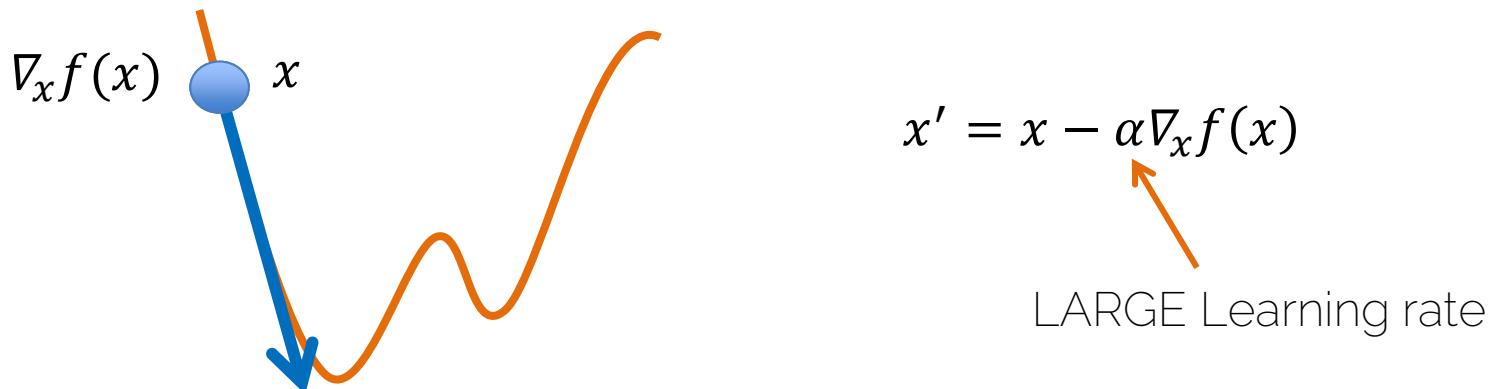
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



# Gradient Descent

$$\mathbf{x}^* = \arg \min f(\mathbf{x})$$

Initialization

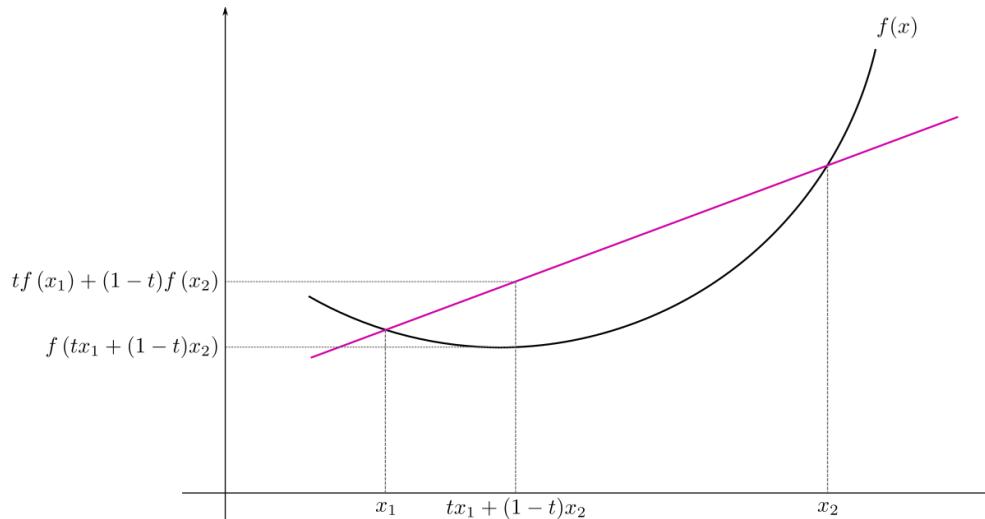
What is the gradient when we reach this point?

Optimum

Not guaranteed to reach the global optimum

# Convergence of Gradient Descent

- Convex function: all local minima are global minima

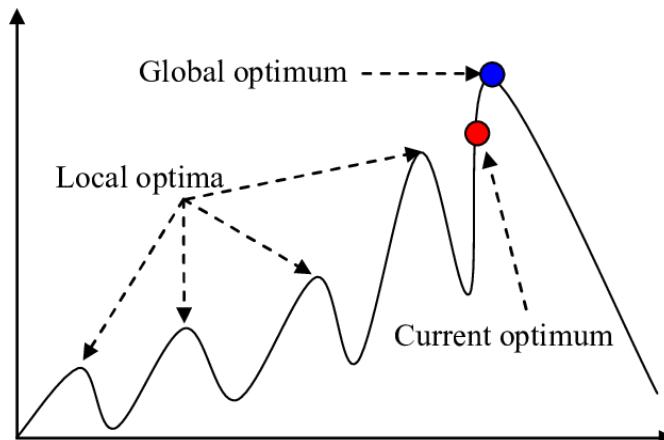


Source: [https://en.wikipedia.org/wiki/Convex\\_function#/media/File:ConvexFunction.svg](https://en.wikipedia.org/wiki/Convex_function#/media/File:ConvexFunction.svg)

If line/plane segment between any two points lies above or on the graph

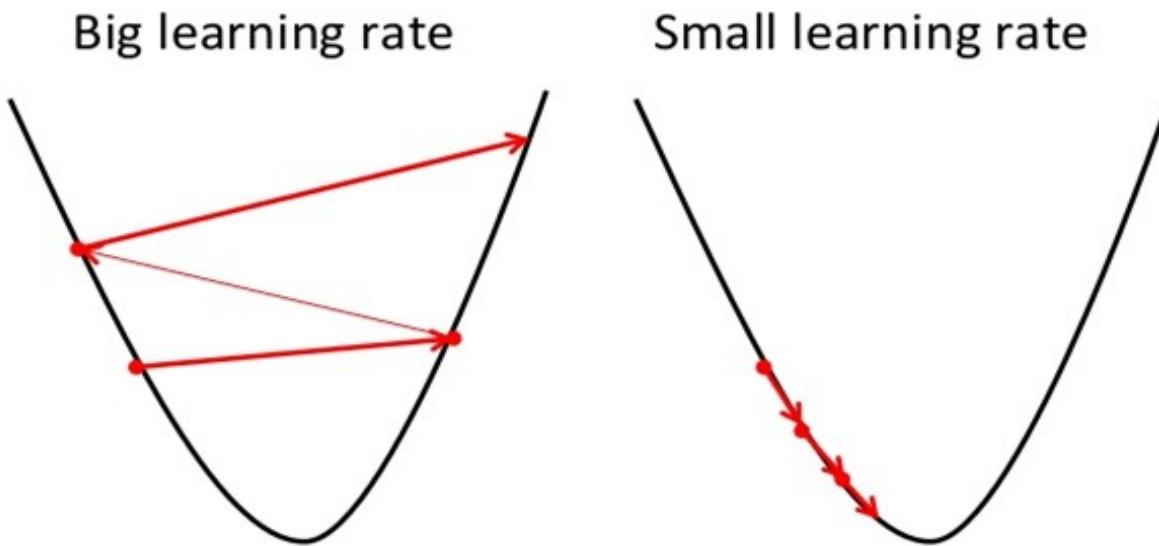
# Convergence of Gradient Descent

- Neural networks are non-convex
  - many (different) local minima
  - no (practical) way to say which is globally optimal



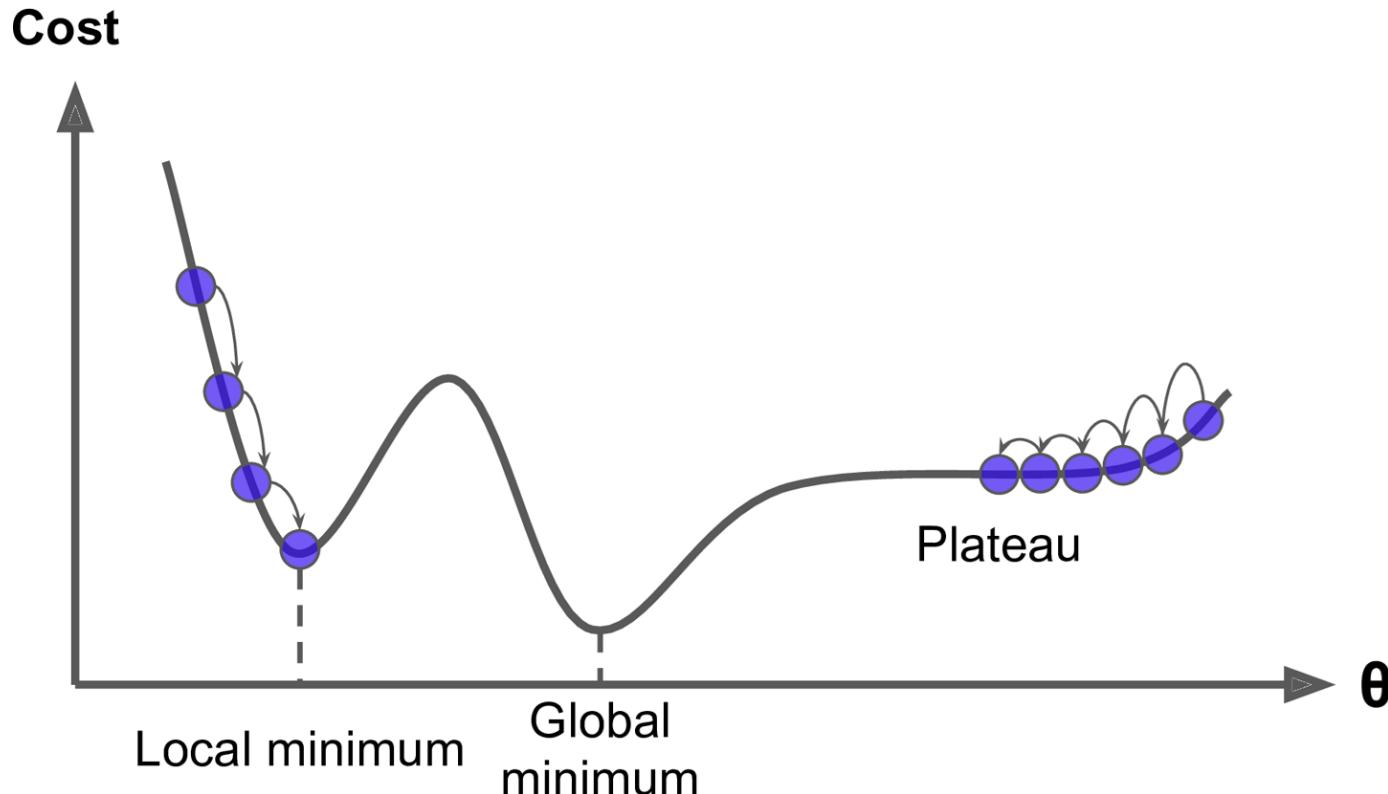
Source: Li, Qi. (2006). Challenging Registration of Geologic Image Data

# Convergence of Gradient Descent



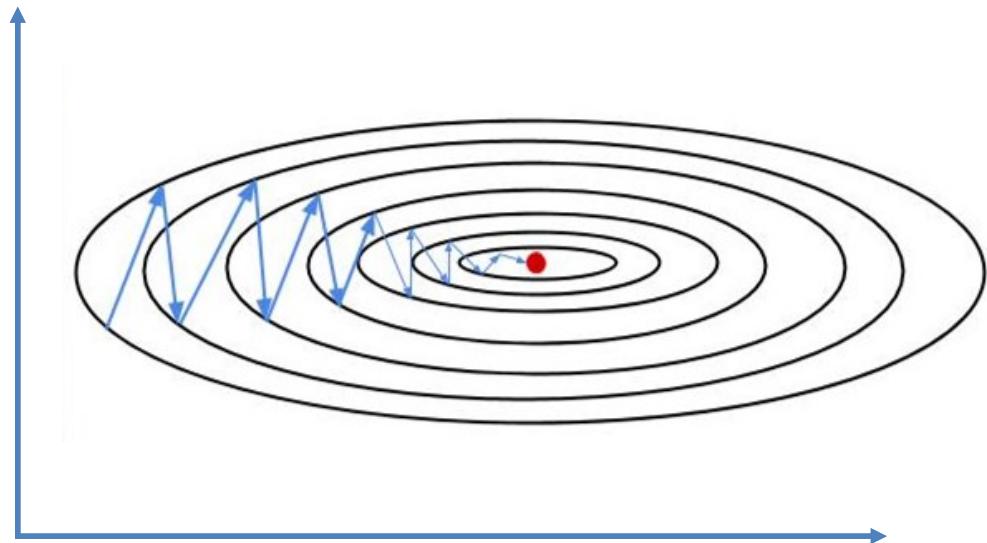
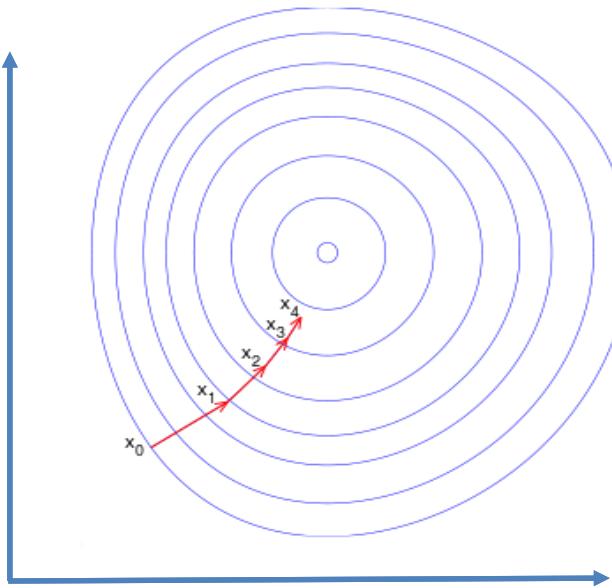
Source: <https://builtin.com/data-science/gradient-descent>

# Convergence of Gradient Descent



Source: A. Geron

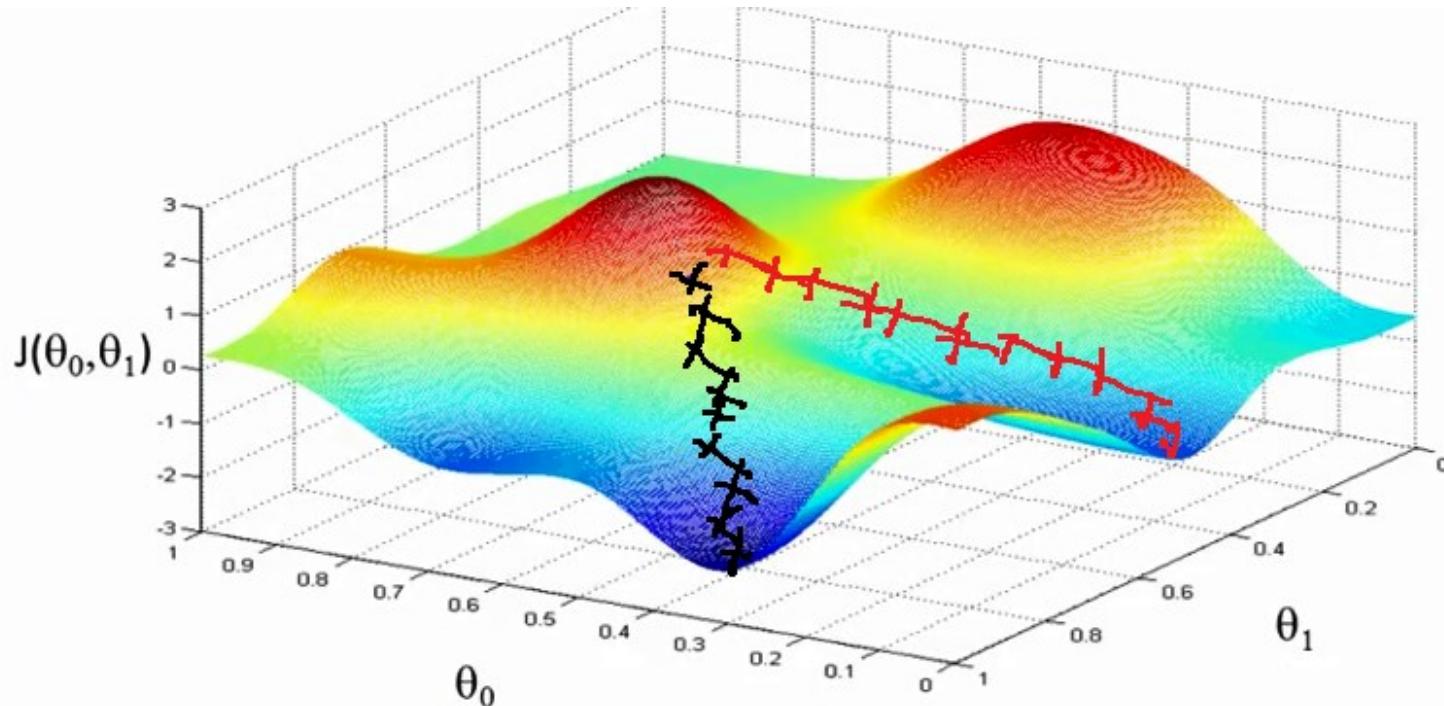
# Gradient Descent: Multiple Dimensions



Source: [builtin.com/data-science/gradient-descent](https://builtin.com/data-science/gradient-descent)

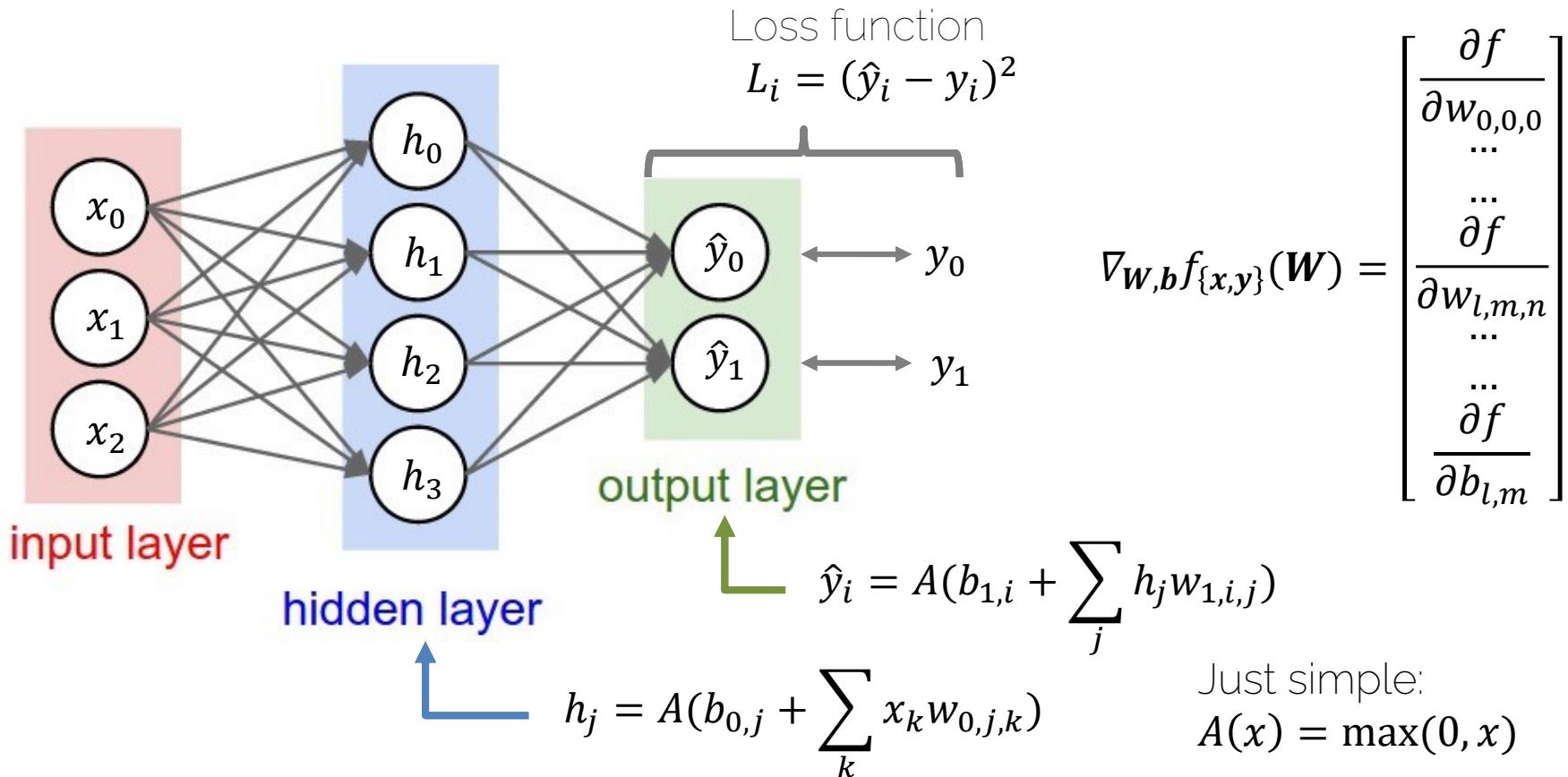
Various ways to visualize...

# Gradient Descent: Multiple Dimensions



Source: <http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png>

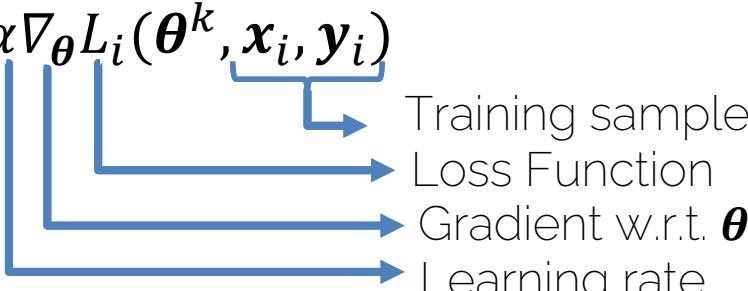
# Gradient Descent for Neural Networks



# Gradient Descent: Single Training Sample

- Given a loss function  $L$  and a single training sample  $\{\mathbf{x}_i, \mathbf{y}_i\}$
- Find best model parameters  $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$
- Cost  $L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)$ 
  - $\boldsymbol{\theta} = \arg \min L_i(\mathbf{x}_i, \mathbf{y}_i)$
- Gradient Descent:
  - Initialize  $\boldsymbol{\theta}^1$  with 'random' values (more on that later)
  - $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \mathbf{x}_i, \mathbf{y}_i)$
  - Iterate until convergence:  $|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k| < \epsilon$

# Gradient Descent: Single Training Sample

- $\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L_i(\theta^k, x_i, y_i)$ 
    - Weights, biases after update step
    - Weights, biases at step k (current model)
  - $\nabla_{\theta} L_i(\theta^k, x_i, y_i)$  computed via backpropagation
  - Typically:  $\dim(\nabla_{\theta} L_i(\theta^k, x_i, y_i)) = \dim(\theta) \gg 1 \text{ million}$
- 

# Gradient Descent: Multiple Training Samples

- Given a loss function  $L$  and multiple ( $n$ ) training samples  $\{\mathbf{x}_i, \mathbf{y}_i\}$
- Find best model parameters  $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$
- Cost  $L = \frac{1}{n} \sum_{i=1}^n L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)$ 
  - $\boldsymbol{\theta} = \arg \min L$

# Gradient Descent: Multiple Training Samples

- Update step for multiple samples

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k, \mathbf{x}_{\{1..n\}}, \mathbf{y}_{\{1..n\}})$$

- Gradient is average / sum over residuals

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k, \mathbf{x}_{\{1..n\}}, \mathbf{y}_{\{1..n\}}) = \frac{1}{n} \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \mathbf{x}_i, \mathbf{y}_i)$$

Reminder: this comes from backprop.

- Often people are lazy and just write:  $\nabla L = \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} L_i$ 
  - omitting  $\frac{1}{n}$  is not 'wrong', it just means rescaling the learning rate

# Side Note: Optimal Learning Rate

Can compute optimal learning rate  $\alpha$  using Line Search  
(optimal for a given set)

1. Compute gradient:  $\nabla_{\theta} L = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L_i$
2. Optimize for optimal step  $\alpha$ :

$$\arg \min_{\alpha} L(\underbrace{\theta^k - \alpha \nabla_{\theta} L}_{\theta^{k+1}})$$

3.  $\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L$

Not that practical for DL since we need to solve huge system every step...

# Gradient Descent on Train Set

- Given large train set with  $n$  training samples  $\{\mathbf{x}_i, \mathbf{y}_i\}$ 
  - Let's say 1 million labeled images
  - Let's say our network has 500k parameters
- Gradient has 500k dimensions
- $n = 1 \text{ million}$
- Extremely expensive to compute

# Stochastic Gradient Descent (SGD)

- If we have  $n$  training samples we need to compute the gradient for all of them which is  $\mathcal{O}(n)$
- If we consider the problem as empirical risk minimization, we can express the total loss over the training data as the expectation of all the samples

$$\frac{1}{n} \left( \sum_{i=1}^n L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i) \right) = \mathbb{E}_{i \sim [1, \dots, n]} [L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)]$$

# Stochastic Gradient Descent (SGD)

- The expectation can be approximated with a small subset of the data

$$\mathbb{E}_{i \sim [1, \dots, n]} [L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)] \approx \frac{1}{|S|} \sum_{j \in S} (L_j(\boldsymbol{\theta}, \mathbf{x}_j, \mathbf{y}_j)) \text{ with } S \subseteq \{1, \dots, n\}$$

Minibatch  
choose subset of trainset  $m \ll n$

$$B_i = \{\{\mathbf{x}_1, \mathbf{y}_1\}, \{\mathbf{x}_2, \mathbf{y}_2\}, \dots, \{\mathbf{x}_m, \mathbf{y}_m\}\}$$
$$\{B_1, B_2, \dots, B_{n/m}\}$$

# Stochastic Gradient Descent (SGD)

- Minibatch size is hyperparameter
  - Typically power of 2 → 8, 16, 32, 64, 128...
  - Smaller batch size means greater variance in the gradients
    - noisy updates
  - Mostly limited by GPU memory (in backward pass)
  - E.g.,
    - Train set has  $\mathbf{n} = 2^{20}$  (about 1 million) images
    - With batch size  $\mathbf{m} = 64$ :  $B_1 \dots n/m = B_1 \dots 16,384$  minibatches  
(Epoch = complete pass through training set)

# Stochastic Gradient Descent (SGD)

$$\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L(\theta^k, x_{\{1..m\}}, y_{\{1..m\}})$$

$\nabla_{\theta} L = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L_i$

$k$  now refers to  $k$ -th iteration

$m$  training samples in the current minibatch

Gradient for the  $k$ -th minibatch

Note the terminology: iteration vs epoch

# Convergence of SGD

Suppose we want to minimize the function  $F(\theta)$  with the stochastic approximation

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$$

where  $\alpha_1, \alpha_2 \dots \alpha_n$  is a sequence of positive step-sizes and  $H(\theta^k, X)$  is the unbiased estimate of  $\nabla F(\theta^k)$ , i.e.

$$\mathbb{E}[H(\theta^k, X)] = \nabla F(\theta^k)$$

Robbins, H. and Monro, S. "A Stochastic Approximation Method" 1951.

# Convergence of SGD

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$$

converges to a local (**global**) minimum if the following conditions are met:

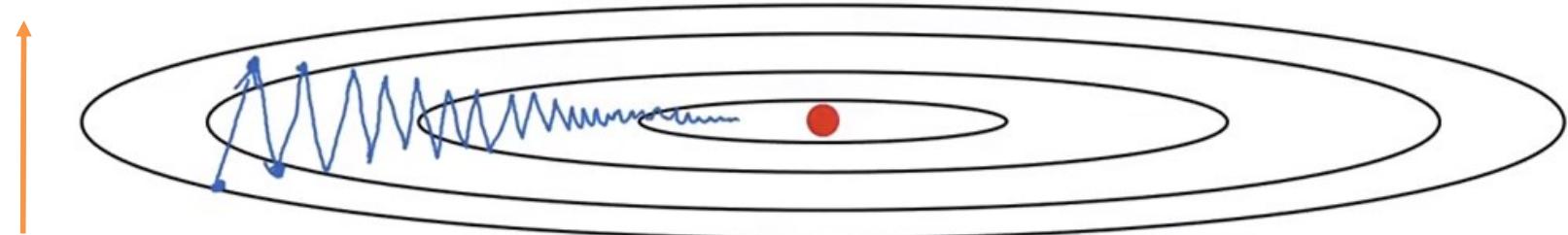
- 1)  $\alpha_n \geq 0, \forall n \geq 0$
- 2)  $\sum_{n=1}^{\infty} \alpha_n = \infty$
- 3)  $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$
- 4)  **$F(\theta)$  is strictly convex**

The proposed sequence by Robbins and Monro is  $\alpha_n \propto \frac{\alpha}{n}, for n > 0$

# Problems of SGD

- Gradient is scaled equally across all dimensions
  - i.e., cannot independently scale directions
  - need to have conservative min learning rate to avoid divergence
  - Slower than 'necessary'
- Finding good learning rate is an art by itself
  - More next lecture

# Gradient Descent with Momentum



Source: A. Ng

We're making many steps back and forth along this dimension. Would love to track that this is averaging out over time.

Would love to go faster here...  
i.e., accumulated gradients over time

# Gradient Descent with Momentum

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)$$

accumulation rate ('friction', momentum)      velocity      learning rate      Gradient of current minibatch

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

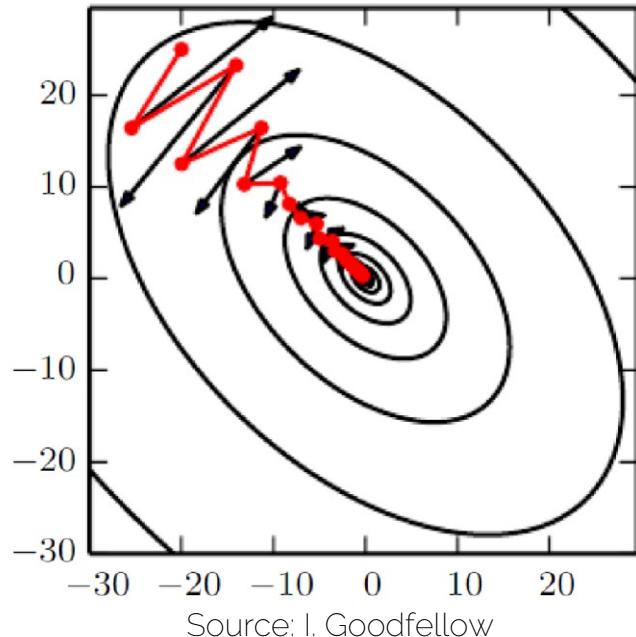
weights of model      velocity

```
graph TD; subgraph Top [ ]; v1["vk+1 = β · vk - α · ∇θ L(θk)"]; end; subgraph Bottom [ ]; v2["θk+1 = θk + vk+1"]; end; Top -- "learning rate" --> v1; Top -- "Gradient of current minibatch" --> v1; Bottom -- "velocity" --> v2; Top -- "velocity" --> v2;
```

Exponentially-weighted average of gradient

Important: velocity  $\boldsymbol{v}^k$  is vector-valued!

# Gradient Descent with Momentum



Step will be largest when a sequence of gradients all point to the same direction

Hyperparameters are  $\alpha, \beta$   
 $\beta$  is often set to 0.9

$$\theta^{k+1} = \theta^k + v^{k+1}$$

# Gradient Descent with Momentum

- Can it overcome local minima?



$$\theta^{k+1} = \theta^k + v^{k+1}$$

# Nesterov Momentum

- Look-ahead momentum

$$\tilde{\boldsymbol{\theta}}^{k+1} = \boldsymbol{\theta}^k + \beta \cdot \boldsymbol{v}^k$$

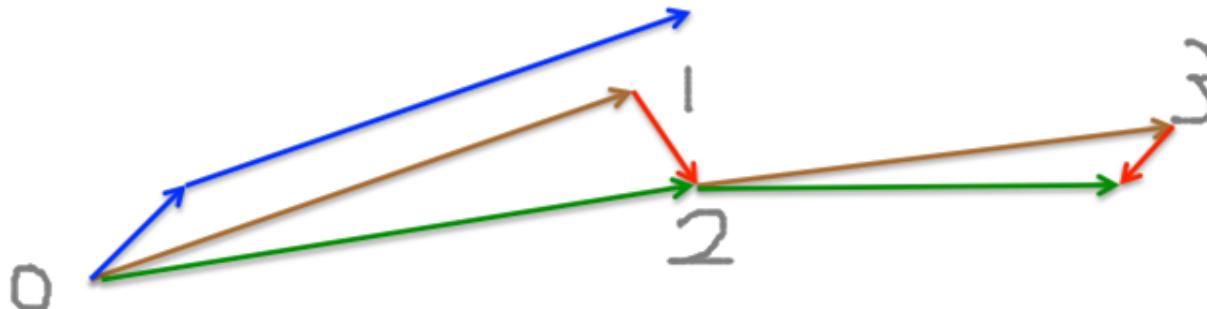
$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\tilde{\boldsymbol{\theta}}^{k+1})$$

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

Nesterov, Yurii E. "A method for solving the convex programming problem with convergence rate  $O(1/k^2)$ ." *Dokl. akad. nauk Sssr.* Vol. 269. 1983.

# Nesterov Momentum

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



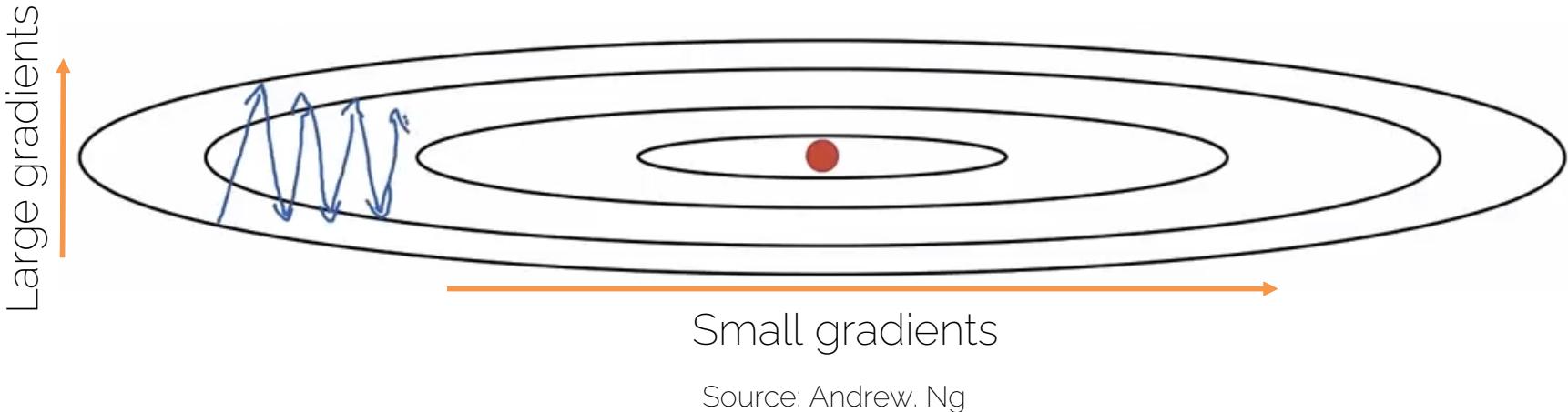
brown vector = jump,      red vector = correction,      green vector = accumulated gradient

blue vectors = standard momentum

Source: G. Hinton

$$\begin{aligned}\tilde{\theta}^{k+1} &= \theta^k + \beta \cdot v^k \\ v^{k+1} &= \beta \cdot v^k - \alpha \cdot \nabla_{\theta} L(\tilde{\theta}^{k+1}) \\ \theta^{k+1} &= \theta^k + v^{k+1}\end{aligned}$$

# Root Mean Squared Prop (RMSProp)



Source: Andrew. Ng

- RMSProp divides the learning rate by an exponentially-decaying average of squared gradients.

Hinton et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural networks for machine learning 4.2 (2012): 26-31.

# RMSProp

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_{\theta} L}{\sqrt{s^{k+1}} + \epsilon}$$

Element-wise multiplication

$$s^{k+1} = \beta \cdot s^k + (1 - \beta) [\nabla_{\theta} L \circ \nabla_{\theta} L]$$

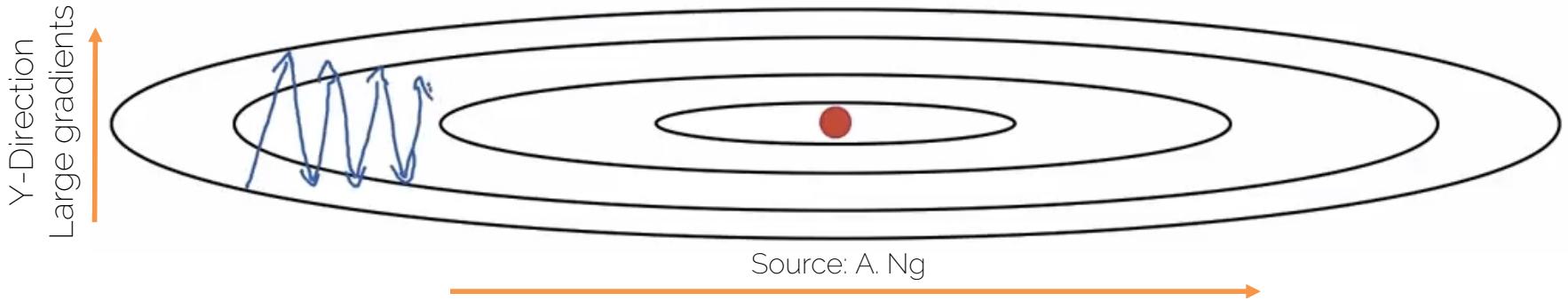
Hyperparameters:  $\alpha, \beta, \epsilon$

Needs tuning!

Often 0.9

Typically  $10^{-8}$

# RMSProp



(Uncentered) variance of gradients

→ second momentum

$$\boxed{\mathbf{s}^{k+1} = \beta \cdot \mathbf{s}^k + (1 - \beta)[\nabla_{\theta} L \circ \nabla_{\theta} L]}$$

We're dividing by square gradients:

- Division in Y-Direction will be large
- Division in X-Direction will be small

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\nabla_{\theta} L}{\sqrt{\mathbf{s}^{k+1}} + \epsilon}$$

Can increase learning rate!

# RMSProp

- Dampening the oscillations for high-variance directions
- Can use faster learning rate because it is less likely to diverge
  - Speed up learning speed
  - Second moment

# Adaptive Moment Estimation (Adam)

Idea : Combine Momentum and RMSProp

$$\mathbf{m}^{k+1} = \beta_1 \cdot \mathbf{m}^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k)$$

First momentum:  
mean of gradients

$$\mathbf{v}^{k+1} = \beta_2 \cdot \mathbf{v}^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\mathbf{m}^{k+1}}{\sqrt{\mathbf{v}^{k+1}} + \epsilon}$$

Note : This is not the  
update rule of  
Adam

Second momentum:  
variance of gradients

Q. What happens at  $k = 0$ ?

A. We need bias correction as  $\mathbf{m}^0 = \mathbf{0}$  and  $\mathbf{v}^0 = \mathbf{0}$

# Adam : Bias Corrected

- Combines Momentum and RMSProp

$$\mathbf{m}^{k+1} = \beta_1 \cdot \mathbf{m}^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k) \quad \mathbf{v}^{k+1} = \beta_2 \cdot \mathbf{v}^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$

- $\mathbf{m}^k$  and  $\mathbf{v}^k$  are initialized with zero
  - bias towards zero
  - Need bias-corrected moment updates

Update rule of Adam

$$\hat{\mathbf{m}}^{k+1} = \frac{\mathbf{m}^{k+1}}{1 - \beta_1^{k+1}} \quad \hat{\mathbf{v}}^{k+1} = \frac{\mathbf{v}^{k+1}}{1 - \beta_2^{k+1}} \quad \longrightarrow \quad \theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{\mathbf{m}}^{k+1}}{\sqrt{\hat{\mathbf{v}}^{k+1}} + \epsilon}$$

# Adam

- Exponentially-decaying mean and variance of gradients (combines first and second order momentum)
- Hyperparameters:  $\alpha, \beta_1, \beta_2, \epsilon$

Needs tuning!

Often 0.9

Often 0.999

Typically  $10^{-8}$

Defaults in PyTorch

$$\mathbf{m}^{k+1} = \beta_1 \cdot \mathbf{m}^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k)$$
$$\mathbf{v}^{k+1} = \beta_2 \cdot \mathbf{v}^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$
$$\hat{\mathbf{m}}^{k+1} = \frac{\mathbf{m}^{k+1}}{1 - \beta_1^{k+1}}$$
$$\hat{\mathbf{v}}^{k+1} = \frac{\mathbf{v}^{k+1}}{1 - \beta_2^{k+1}}$$
$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{\mathbf{m}}^{k+1}}{\sqrt{\hat{\mathbf{v}}^{k+1}} + \epsilon}$$

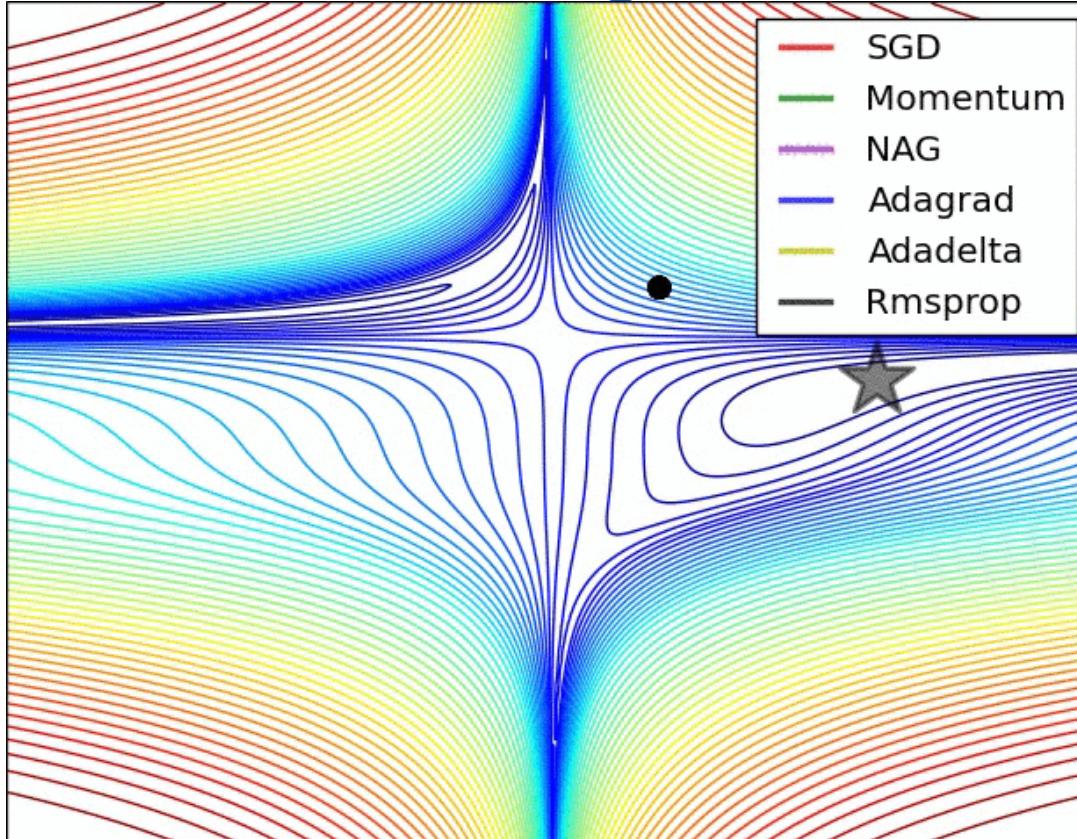
# There are a few others...

- 'Vanilla' SGD
- Momentum
- RMSProp
- Adagrad
- Adadelta
- AdaMax
- Nada
- AMSGrad

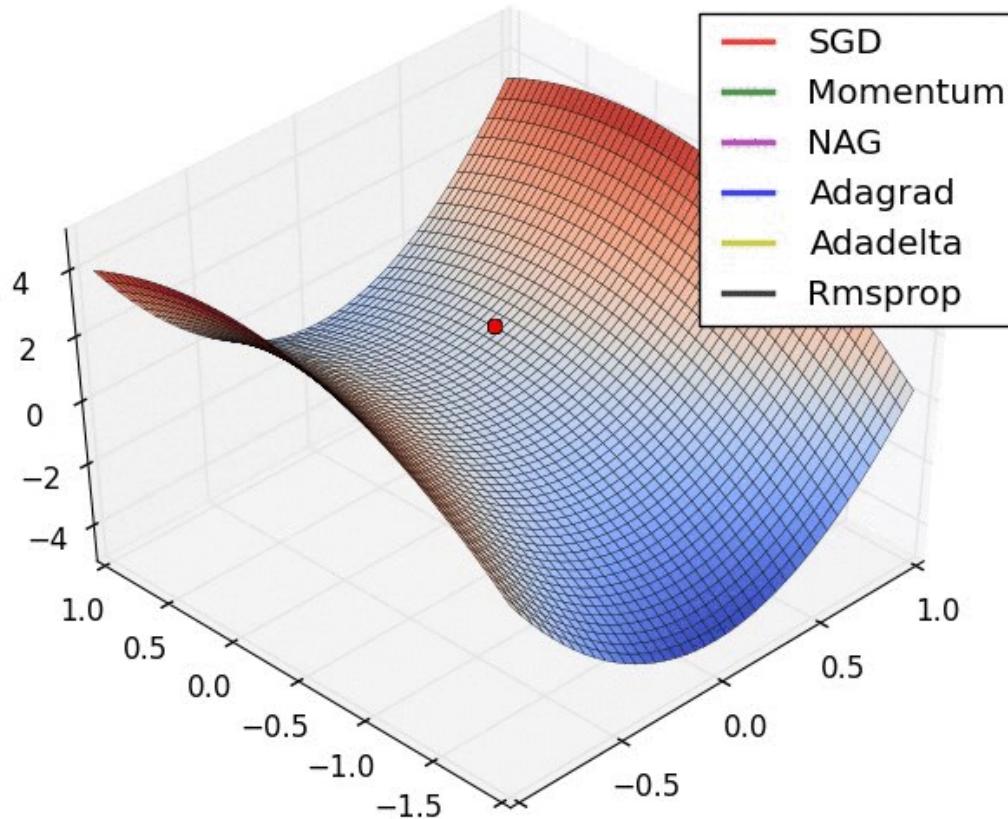
Adam is mostly method  
of choice for neural networks!

It's actually fun to play around with SGD  
updates.  
It's easy and you get pretty immediate  
feedback ☺

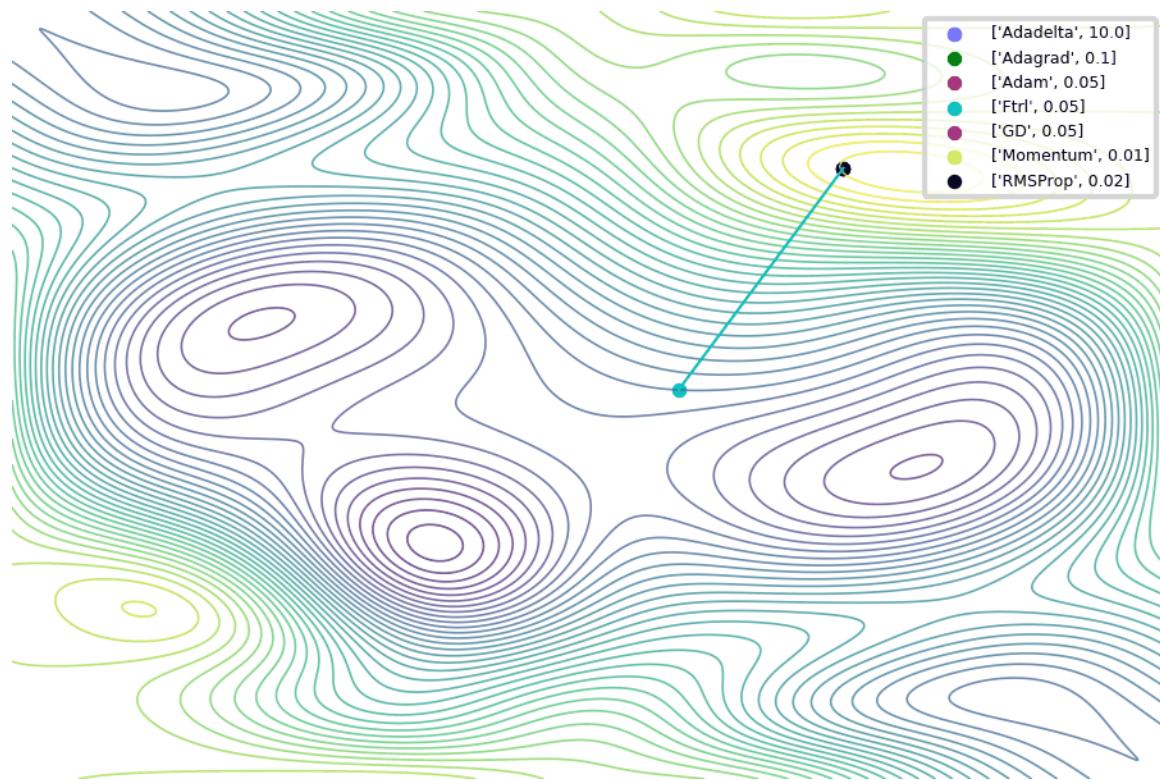
# Convergence



# Convergence



# Convergence



Source: <https://github.com/Jaewan-Yun/optimizer-visualization>

# Jacobian and Hessian

- Derivative

$$\mathbf{f}: \mathbb{R} \rightarrow \mathbb{R}$$

$$\frac{df(x)}{dx}$$

- Gradient

$$\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}$$

$$\nabla_{\mathbf{x}} f(\mathbf{x})$$

$$\left( \frac{df(x)}{dx_1}, \frac{df(x)}{dx_2} \right)$$

- Jacobian

$$\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}^n$$

$$\mathbf{J} \in \mathbb{R}^{n \times m}$$

- Hessian

$$\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}$$

$$\mathbf{H} \in \mathbb{R}^{m \times m}$$

SECOND  
DERIVATIVE

# Newton's Method

- Approximate our function by a second-order Taylor series expansion

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

First derivative                                  Second derivative (curvature)

More info:  
[https://en.wikipedia.org/wiki/Taylor\\_series](https://en.wikipedia.org/wiki/Taylor_series)

# Newton's Method

- Differentiate and equate to zero

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

Update step



We got rid of the learning rate!

SGD       $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \mathbf{x}_i, \mathbf{y}_i)$

# Newton's Method

- Differentiate and equate to zero

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

Update step

Parameters of a network (millions)

$k$

Number of elements in the Hessian

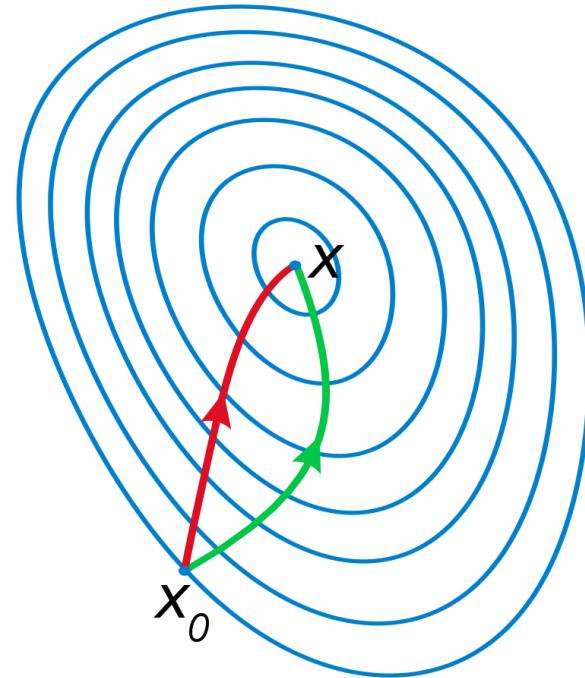
$k^2$

Computational complexity of 'inversion' per iteration

$\mathcal{O}(k^3)$

# Newton's Method

- Gradient Descent (green)
- Newton's method exploits the curvature to take a more direct route



Source: [https://en.wikipedia.org/wiki/Newton%27s\\_method\\_in\\_optimization](https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization)

# Newton's Method

$$J(\boldsymbol{\theta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

Can you apply Newton's method for linear regression?  
What do you get as a result?

# BFGS and L-BFGS

- Broyden-Fletcher-Goldfarb-Shanno algorithm
- Belongs to the family of quasi-Newton methods
- Have an approximation of the inverse of the Hessian

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boxed{\mathbf{H}^{-1}} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

- BFGS       $\mathcal{O}(n^2)$
- Limited memory: L-BFGS       $\mathcal{O}(n)$

# Gauss-Newton

- $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$ 
  - 'true' 2<sup>nd</sup> derivatives are often hard to obtain (e.g., numerics)
  - $H_f \approx 2J_F^T J_F$
- Gauss-Newton (GN):  
$$x_{k+1} = x_k - [2J_F(x_k)^T J_F(x_k)]^{-1} \nabla f(x_k)$$
- Solve linear system (again, inverting a matrix is unstable):

$$2(J_F(x_k)^T J_F(x_k)) \underbrace{(x_k - x_{k+1})}_{\text{Solve for delta vector}} = \nabla f(x_k)$$

# Levenberg

- Levenberg
  - "damped" version of Gauss-Newton:
  - $(J_F(x_k)^T J_F(x_k) + \lambda \cdot I) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$  Tikhonov regularization
  - The damping factor  $\lambda$  is adjusted in each iteration ensuring:  
$$f(x_k) > f(x_{k+1})$$
    - if the equation is not fulfilled increase  $\lambda$
    - → Trust region
- → "Interpolation" between Gauss-Newton (small  $\lambda$ ) and Gradient Descent (large  $\lambda$ )

# Levenberg-Marquardt

- Levenberg-Marquardt (LM)

$$(J_F(x_k)^T J_F(x_k) + \lambda \cdot \text{diag}(J_F(x_k)^T J_F(x_k))) \cdot (x_k - x_{k+1}) \\ = \nabla f(x_k)$$

- Instead of a plain Gradient Descent for large  $\lambda$ , scale each component of the gradient according to the curvature.
  - Avoids slow convergence in components with a small gradient

# Which, What, and When?

- Standard: Adam
- Fallback option: SGD with momentum
- Newton, L-BFGS, GN, LM only if you can do full batch updates (doesn't work well for minibatches!!)



This practically never happens for DL  
Theoretically, it would be nice though due to fast convergence

# General Optimization

- Linear Systems ( $Ax = b$ )
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, etc.
- Non-linear (gradient-based)
  - Newton, Gauss-Newton, LM, (L)BFGS                     $\leftarrow$  second order
  - Gradient Descent, SGD                                       $\leftarrow$  first order
- Others
  - Genetic algorithms, MCMC, Metropolis-Hastings, etc.
  - Constrained and convex solvers (Langrange, ADMM, Primal-Dual, etc.)

# Please Remember!

- Think about your problem and optimization at hand
- SGD is specifically designed for minibatch
- When you can, use 2<sup>nd</sup> order method → it's just faster
- GD or SGD is not a way to solve a linear system!

# Next Lecture

- This week:
  - Check exercises
  - Check office hours ☺
- Next lecture
  - Training Neural networks

See you next week 😊

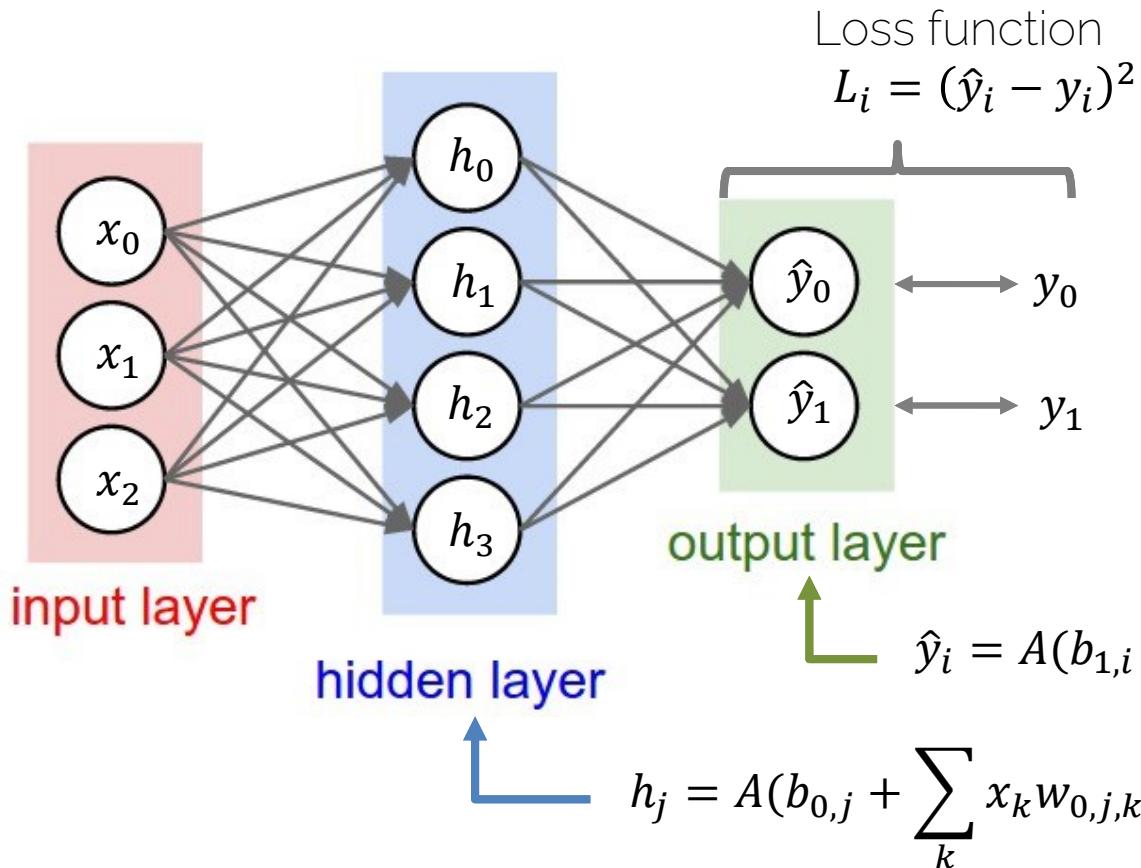
# Some References to SGD Updates

- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 8: Optimization
- Bishop "Pattern Recognition and Machine Learning" (2006),
  - Chapter 5.2: Network training (gradient descent)
  - Chapter 5.4: The Hessian Matrix (second order methods)
- <https://ruder.io/optimizing-gradient-descent/index.html>
- PyTorch Documentation (with further readings)
  - <https://pytorch.org/docs/stable/optim.html>

# Training Neural Networks

# Lecture 5 Recap

# Gradient Descent for Neural Networks



$$\nabla_{W,b} f_{\{x,y\}}(\mathbf{W}) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \dots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \dots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

Just simple:  
 $A(x) = \max(0, x)$

# Stochastic Gradient Descent (SGD)

$$\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L(\theta^k, x_{\{1..m\}}, y_{\{1..m\}})$$

$\nabla_{\theta} L = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L_i$

$m$  training samples in the current minibatch

Gradient for the  $k$ -th minibatch

$k$  now refers to  $k$ -th iteration

# Gradient Descent with Momentum

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k + \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)$$

accumulation rate ('friction', momentum)      velocity      Gradient of current minibatch

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \boldsymbol{v}^{k+1}$$

model

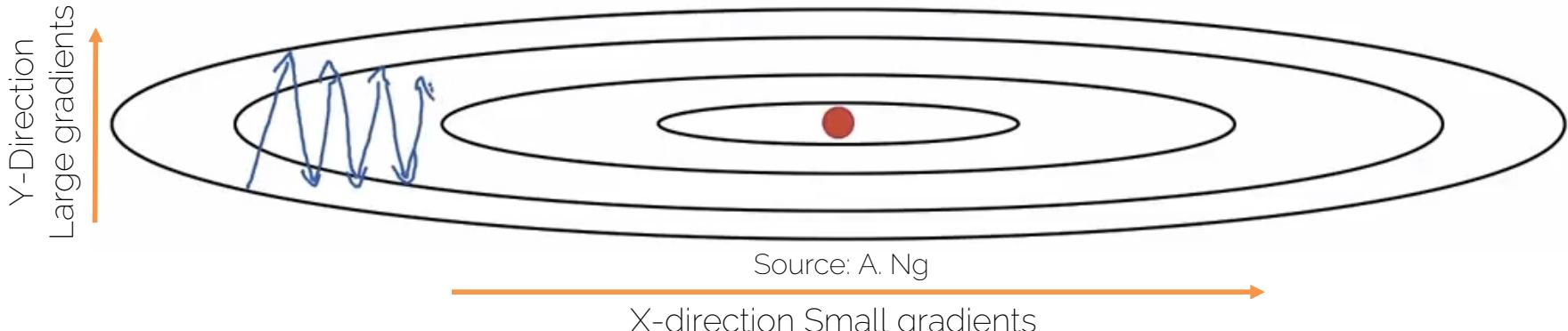
learning rate

velocity

Exponentially-weighted average of gradient

Important: velocity  $\boldsymbol{v}^k$  is vector-valued!

# RMSProp



(Uncentered) variance of gradients

→ second momentum

$$\boxed{s^{k+1} = \beta \cdot s^k + (1 - \beta)[\nabla_{\theta} L \circ \nabla_{\theta} L]}$$

We're dividing by square gradients:

- Division in Y-Direction will be large
- Division in X-Direction will be small

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_{\theta} L}{\sqrt{s^{k+1}} + \epsilon}$$

Can increase learning rate!

# Adam

- Combines Momentum and RMSProp

$$\mathbf{m}^{k+1} = \beta_1 \cdot \mathbf{m}^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k) \quad \mathbf{v}^{k+1} = \beta_2 \cdot \mathbf{v}^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$

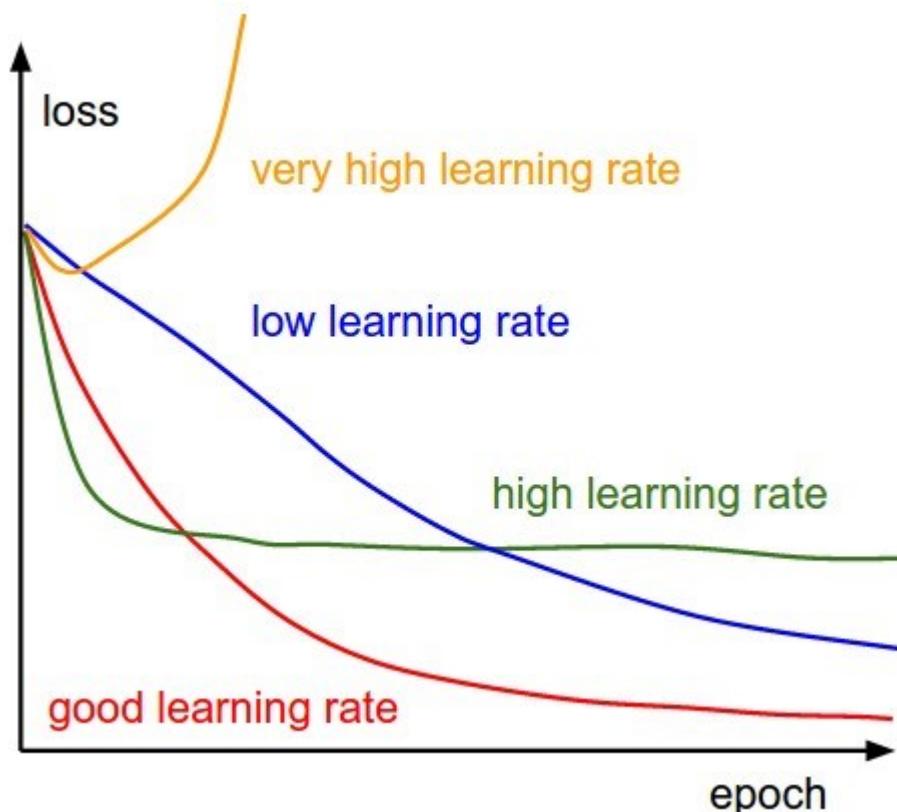
- $\mathbf{m}^{k+1}$  and  $\mathbf{v}^{k+1}$  are initialized with zero
  - bias towards zero
  - Typically, bias-corrected moment updates

$$\hat{\mathbf{m}}^{k+1} = \frac{\mathbf{m}^{k+1}}{1 - \beta_1^{k+1}} \quad \hat{\mathbf{v}}^{k+1} = \frac{\mathbf{v}^{k+1}}{1 - \beta_2^{k+1}} \quad \longrightarrow \quad \theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{\mathbf{m}}^{k+1}}{\sqrt{\hat{\mathbf{v}}^{k+1}} + \epsilon}$$

# Training Neural Nets

# Learning Rate: Implications

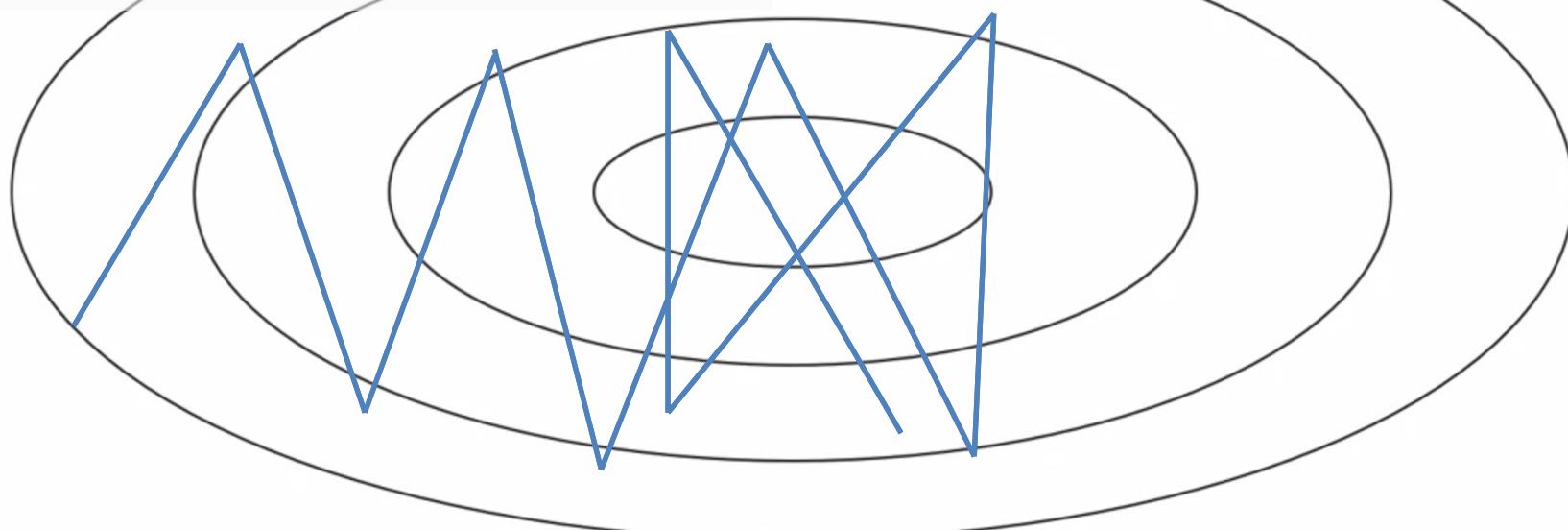
- What if too high?
- What if too low?



Source: <http://cs231n.github.io/neural-networks-3/>

# Learning Rate

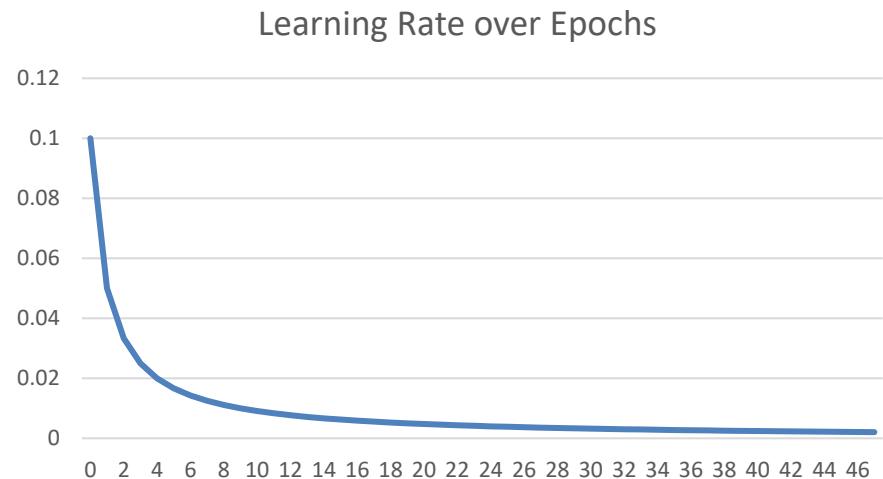
Need high learning rate when far away



Need low learning rate when close

# Learning Rate Decay

- $\alpha = \frac{1}{1+decay\_rate*epoch} \cdot \alpha_0$ 
  - E.g.,  $\alpha_0 = 0.1$ ,  $decay\_rate = 1.0$ 
    - Epoch 0: 0.1
    - Epoch 1: 0.05
    - Epoch 2: 0.033
    - Epoch 3: 0.025
    - ...



# Learning Rate Decay

Many options:

- Step decay  $\alpha = \alpha - t \cdot \alpha$  (only every n steps)
  - T is decay rate (often 0.5)
- Exponential decay  $\alpha = t^{epoch} \cdot \alpha_0$ 
  - t is decay rate ( $t < 1.0$ )
- $\alpha = \frac{t}{\sqrt{epoch}} \cdot \alpha_0$ 
  - t is decay rate
- Etc.

# Training Schedule

Manually specify learning rate for entire training process

- Manually set learning rate every n-epochs
- How?
  - Trial and error (the hard way)
  - Some experience (only generalizes to some degree)

Consider: #epochs, training set size, network size, etc.

# Basic Recipe for Training

- Given a dataset with labels
  - $\{x_i, y_i\}$ 
    - $x_i$  is the  $i^{th}$  training image, with label  $y_i$
    - Often  $\text{dim}(x) \gg \text{dim}(y)$  (e.g., for classification)
    - $i$  is often in the 100-thousands or millions
  - Take network  $f$  and its parameters  $w, b$
  - Use SGD (or variation) to find optimal parameters  $w, b$ 
    - Gradients from backpropagation

# Gradient Descent on Train Set

- Given large train set with ( $n$ ) training samples  $\{\mathbf{x}_i, \mathbf{y}_i\}$ 
  - Let's say 1 million labeled images
  - Let's say our network has 500k parameters
- Gradient has 500k dimensions
- $n = 1 \text{ million}$
- Extremely expensive to compute

# Learning

- Learning means generalization to unknown dataset
  - (So far no 'real' learning)
  - i.e., train on known dataset → test with optimized parameters on unknown dataset
- Basically, we hope that based on the train set, the optimized parameters will give similar results on different data (i.e., test data)

# Learning

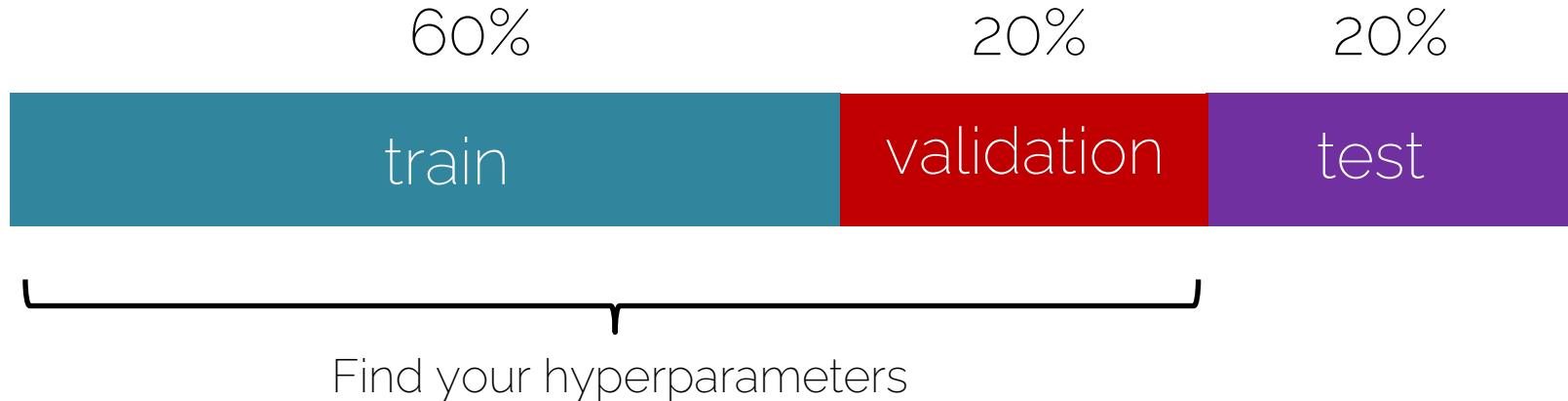
- Training set ('*train*'):
  - Use for training your neural network
- Validation set ('*val*'):
  - Hyperparameter optimization
  - Check generalization progress
- Test set ('*test*'):
  - Only for the very end
  - NEVER TOUCH DURING DEVELOPMENT OR TRAINING

# Learning

- Typical splits
  - Train (60%), Val (20%), Test (20%)
  - Train (80%), Val (10%), Test (10%)
- During training:
  - Train error comes from average minibatch error
  - Typically take subset of validation every n iterations

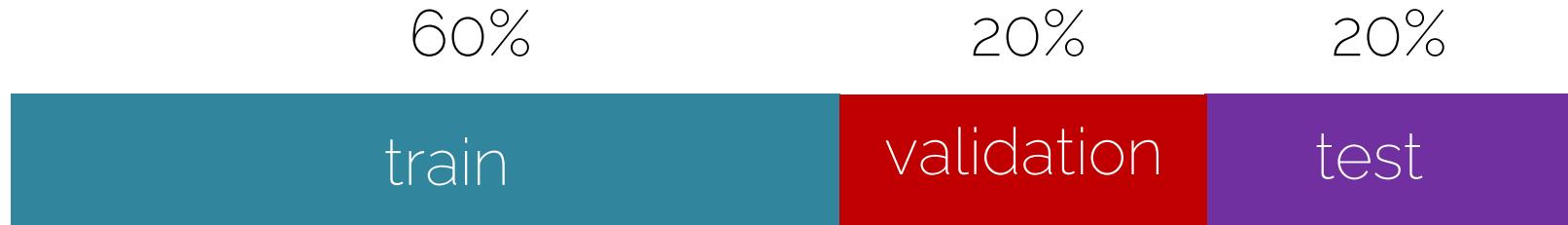
# Basic Recipe for Machine Learning

- Split your data



# Basic Recipe for Machine Learning

- Split your data



Example scenario

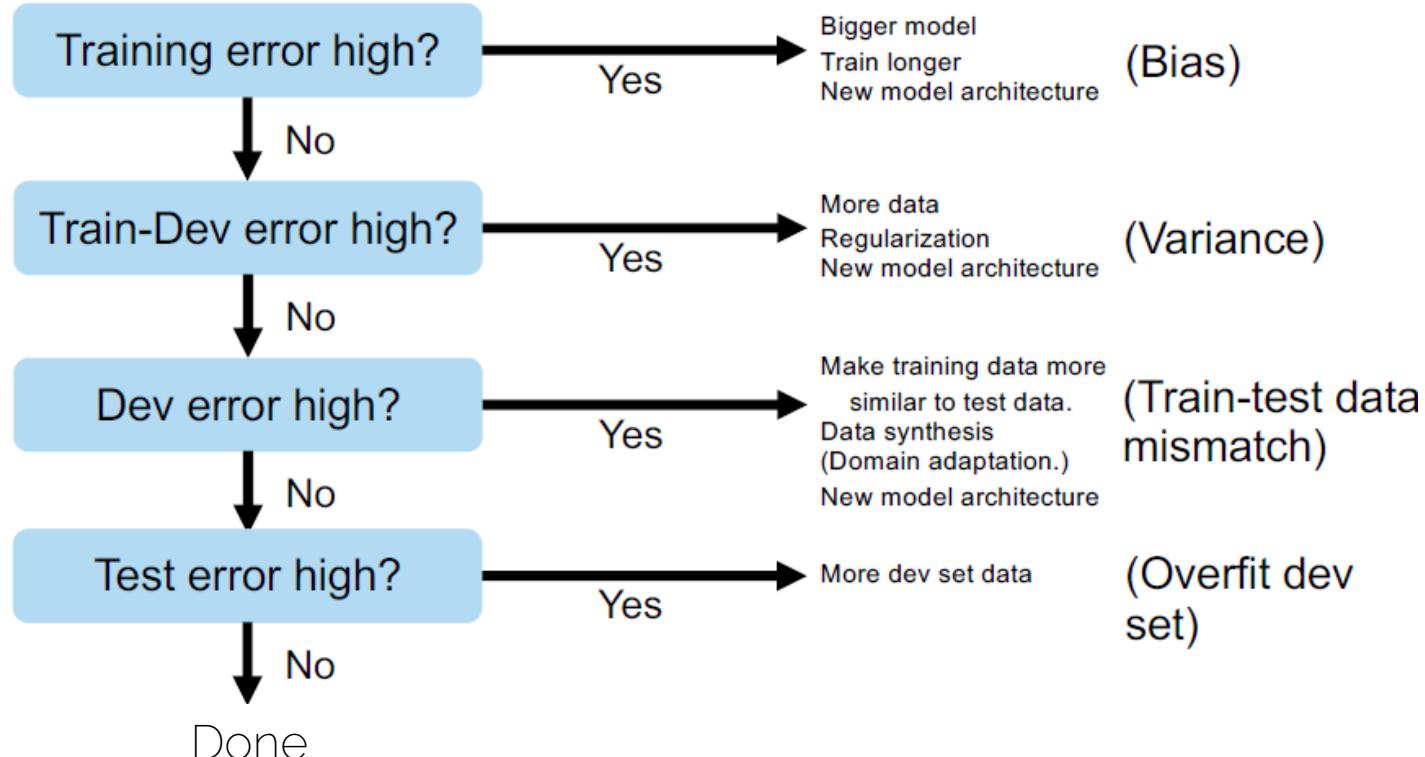
Ground truth error ..... 1%

Training set error ..... 5%

Val/test set error ..... 8%

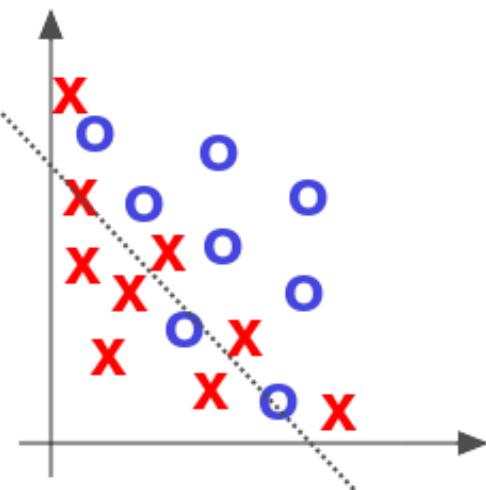
*Bias*  
(underfitting)  
*Variance*  
(overfitting)

# Basic Recipe for Machine Learning

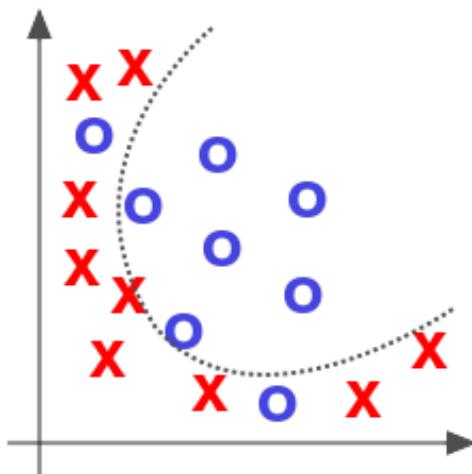


Credits: A. Ng

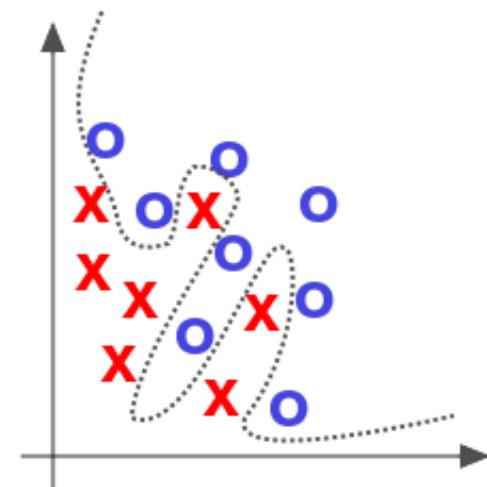
# Over- and Underfitting



Underfitted



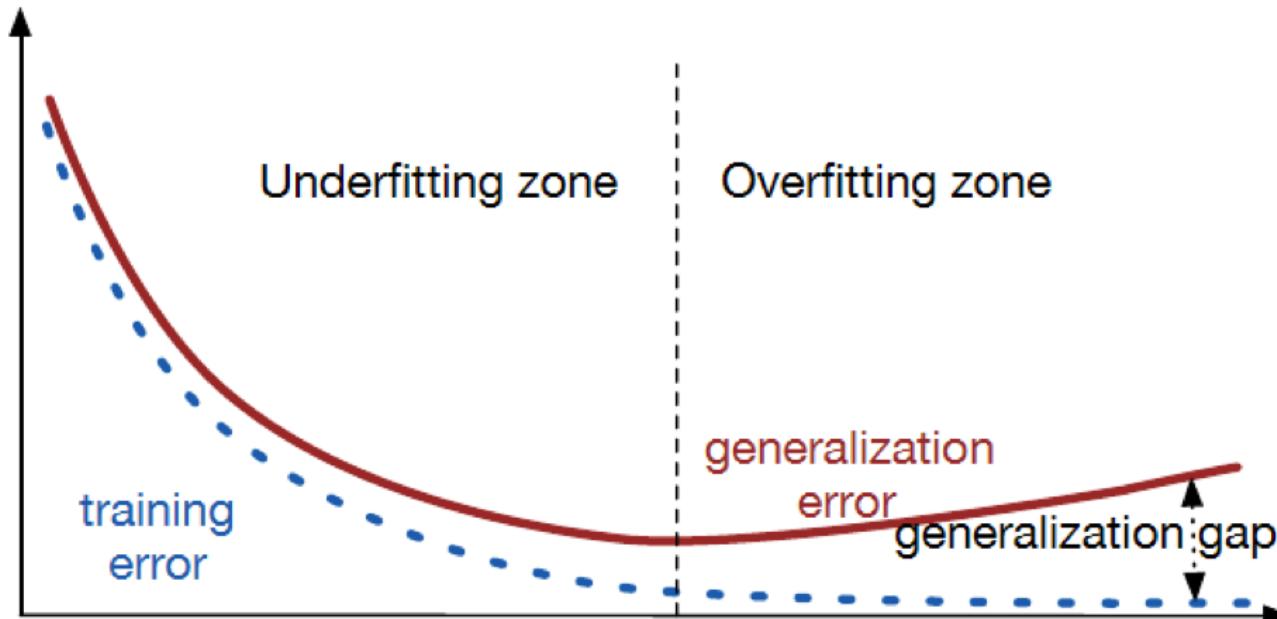
Appropriate



Overfitted

Source: Deep Learning by Adam Gibson, Josh Patterson, O'Reilly Media Inc., 2017

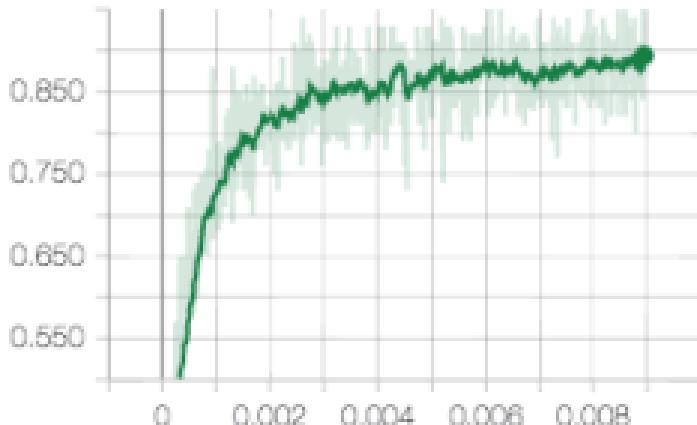
# Over- and Underfitting



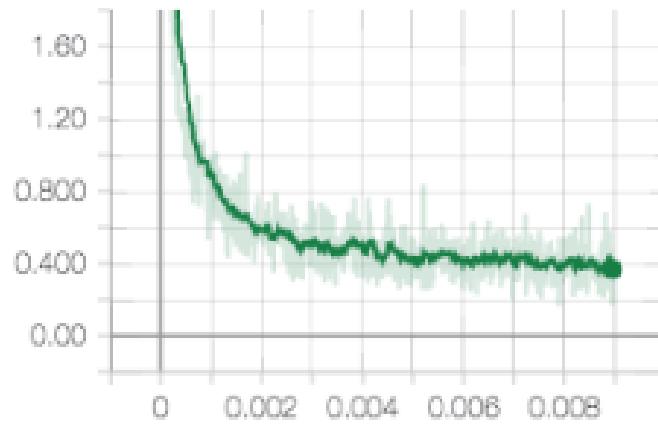
Source: <https://srdas.github.io/DLBook/ImprovingModelGeneralization.html>

# Learning Curves

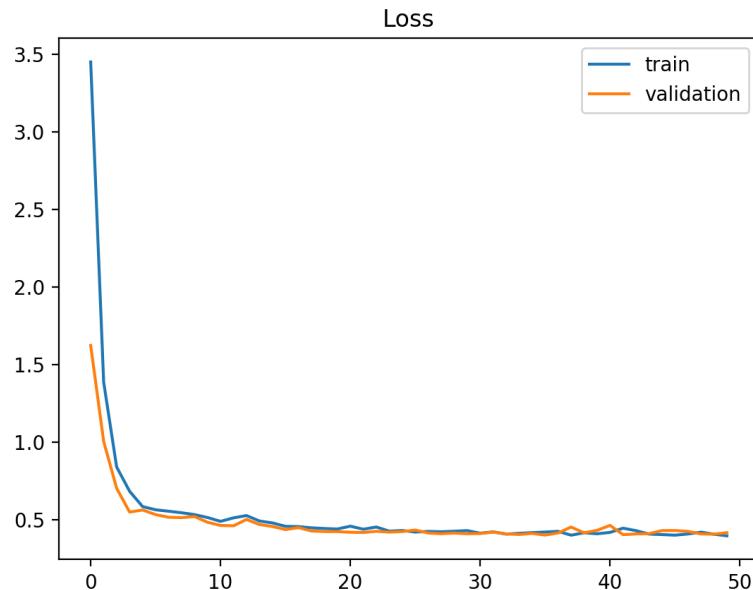
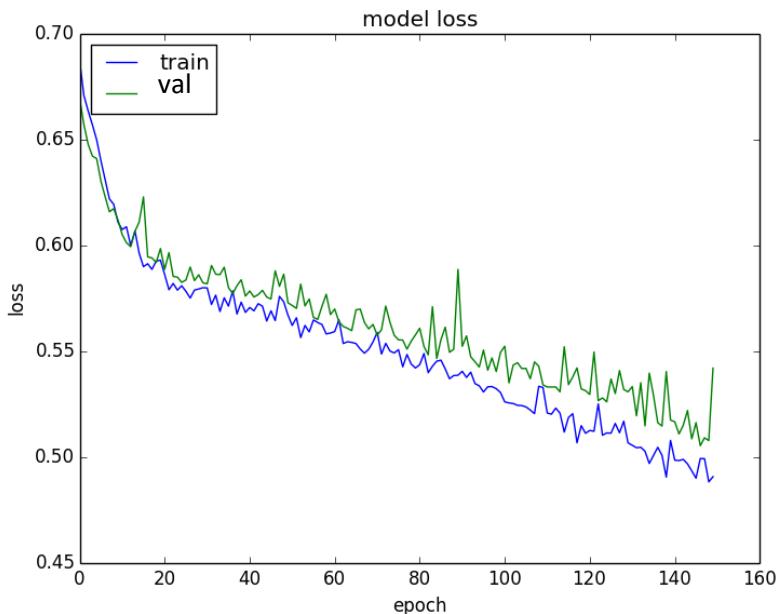
- Training graphs
  - Accuracy



- Loss

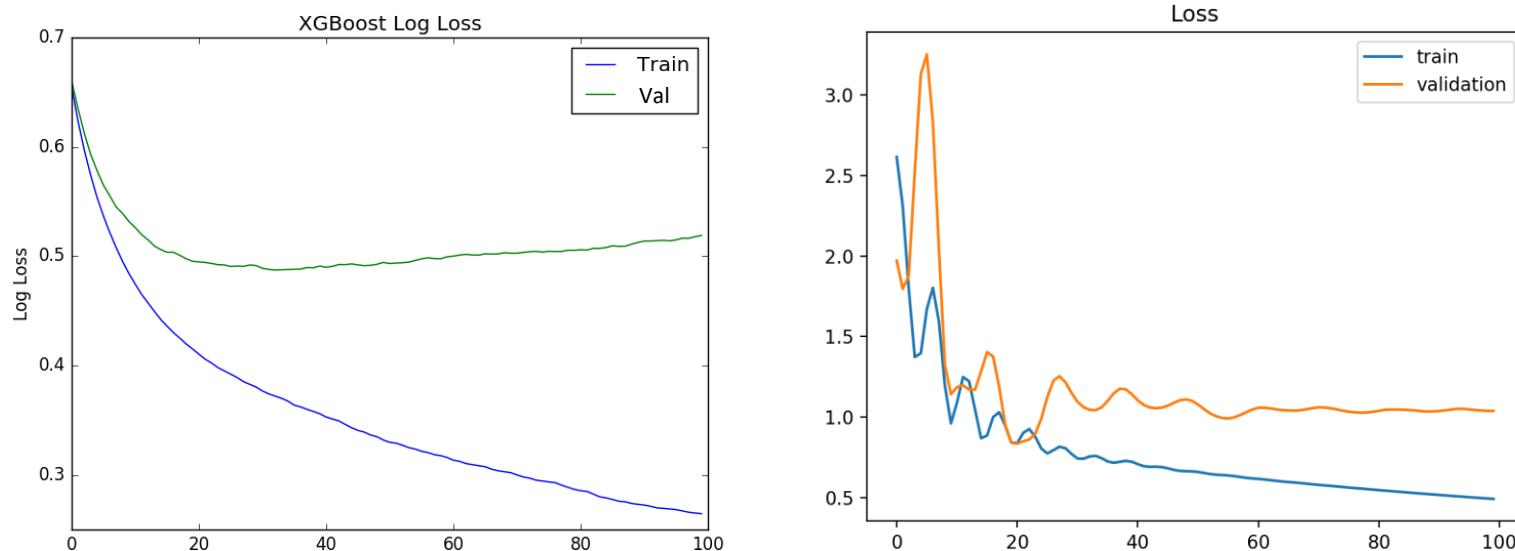


# Learning Curves



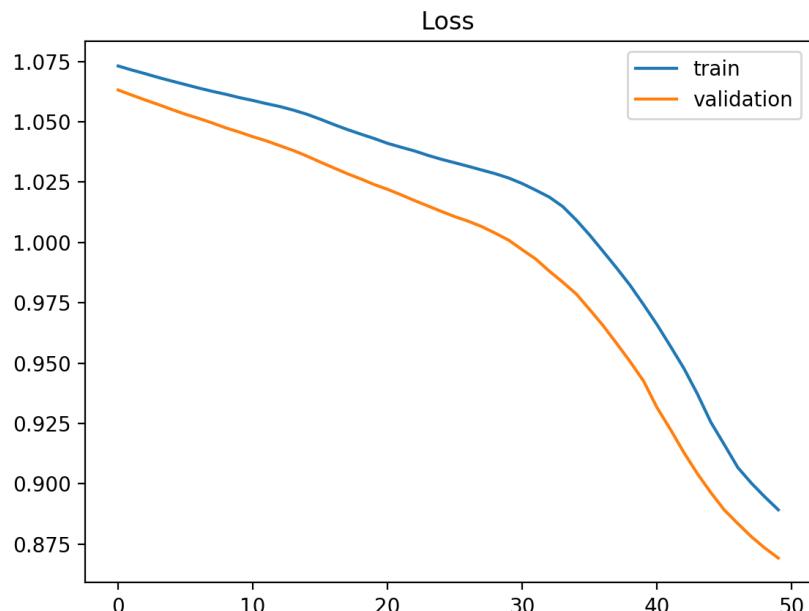
Source: <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>

# Overfitting Curves



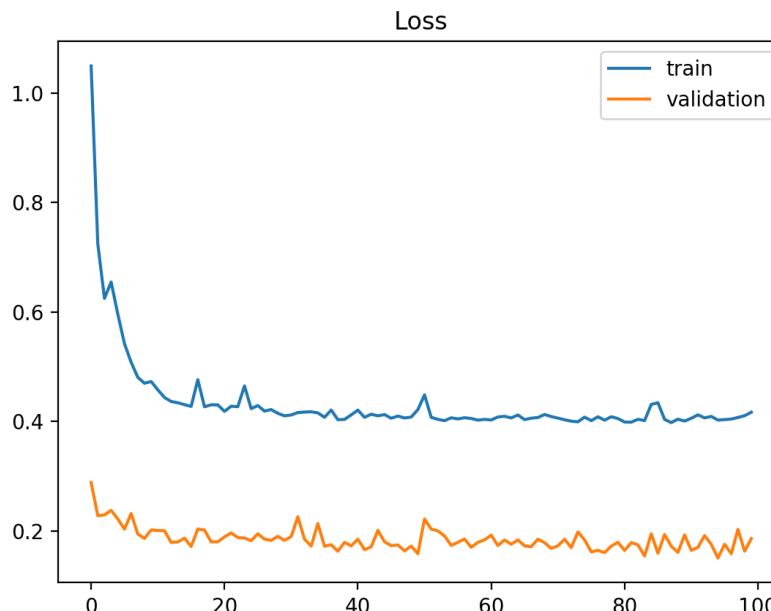
Source: <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>

# Other Curves



Underfitting (loss still decreasing)

Source: <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>



Validation Set is easier than Training set

# To Summarize

- Underfitting
  - Training and validation losses decrease even at the end of training
- Overfitting
  - Training loss decreases and validation loss increases
- Ideal Training
  - Small gap between training and validation loss, and both go down at same rate (stable without fluctuations).

# To Summarize

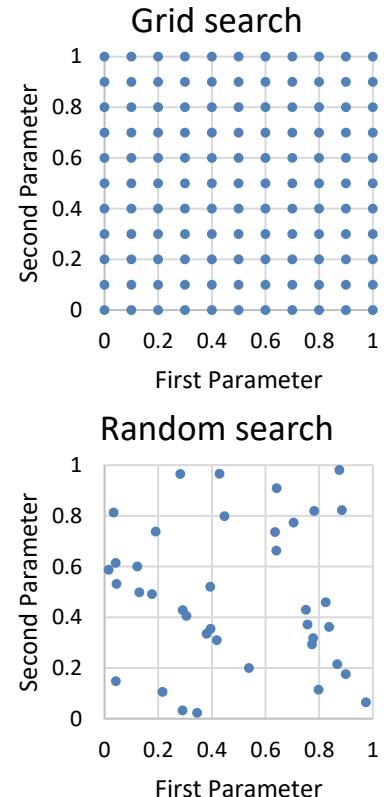
- Bad Signs
    - Training error not going down
    - Validation error not going down
    - Performance on validation better than on training set
    - Tests on train set different than during training
  - Bad Practice
    - Training set contains **test data**
    - Debug algorithm on **test data**
- 
- Never touch during development or training

# Hyperparameters

- Network architecture (e.g., num layers, #weights)
- Number of iterations
- Learning rate(s) (i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size
- ...
- Overall:  
learning setup + optimization = hyperparameters

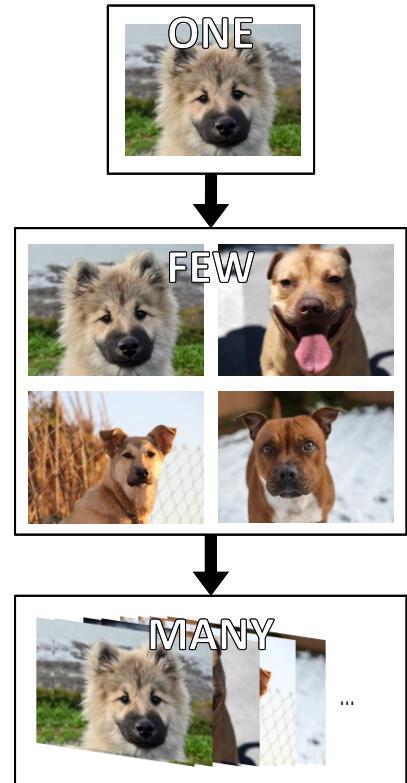
# Hyperparameter Tuning

- Methods:
  - Manual search:
    - most common 😊
  - Grid search (structured, for 'real' applications)
    - Define ranges for all parameters spaces and select points
    - Usually pseudo-uniformly distributed
      - Iterate over all possible configurations
  - Random search:
    - Like grid search but one picks points at random in the predefined ranges



# How to Start

- Start with single training sample
  - Check if output correct
  - Overfit → train accuracy should be 100% because input just memorized
- Increase to handful of samples (e.g., 4)
  - Check if input is handled correctly
- Move from overfitting to more samples
  - 5, 10, 100, 1000, ...
  - At some point, you should see generalization



# Find a Good Learning Rate

# Karpathy's constant



Andrej Karpathy   
@karpathy

...

3e-4 is the best learning rate for Adam, hands down.

4:01 AM · Nov 24, 2016 · Twitter Web Client

---

123 Retweets   30 Quote Tweets   562 Likes

# Karpathy's constant



Andrej Karpathy ✅

@karpathy

...

3e-4 is the best learning rate for Adam, hands down.

4:01 AM · Nov 24, 2016 · Twitter Web Client

---

123 Retweets 30 Quote Tweets 562 Likes



Andrej Karpathy ✅ @karpathy · Nov 24, 2016

...

Replying to @karpathy

(i just wanted to make sure that people understand that this is a joke...)

9

6

144



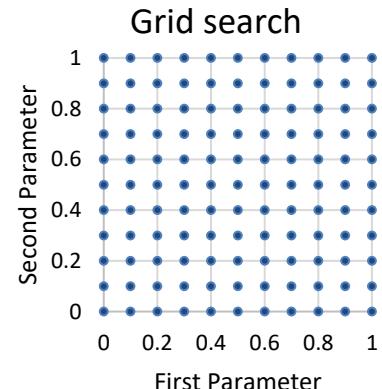
# Find a Good Learning Rate

- Use all training data with small weight decay
- Perform initial loss sanity check e.g.,  $\log(C)$  for softmax with  $C$  classes
- Find a learning rate that makes the loss drop significantly (exponentially) within 100 iterations
- Good learning rates to try:  
 $1e-1, 1e-2, 1e-3, 1e-4$



# Coarse Grid Search

- Choose a few values of learning rate and weight decay around what worked from
- Train a few models for a few epochs.
- Good weight decay to try:  $1e-4$ ,  $1e-5$ , 0

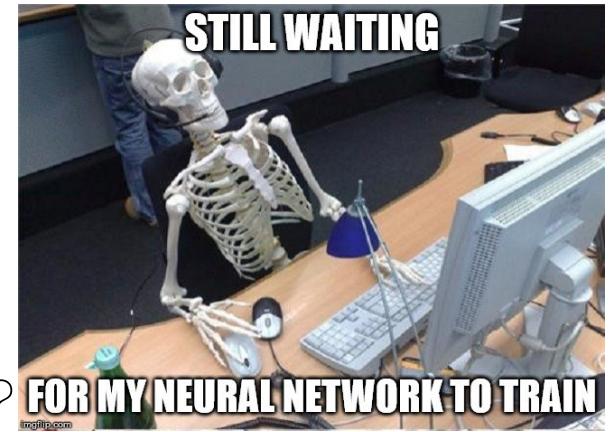


# Refine Grid

- Pick best models found with coarse grid.
- Refine grid search around these models.
- Train them for longer (10-20 epochs) without learning rate decay
- Study loss curves <- most important debugging tool!

# Timings

- How long does each iteration take?
  - Get **precise** timings!
  - If an iteration exceeds **500ms**, things get dicey
- Look for bottlenecks
  - Dataloading: smaller resolution, compression, train from SSD
  - Backprop
- Estimate total time
  - How long until you see some pattern?
  - How long till convergence?



# Network Architecture

- Frequent mistake: “*Let's use this super big network, train for two weeks and we see where we stand.*”
- Instead: start with simplest network possible
  - Rule of thumb divide #layers you started with by 5
- Get debug cycles down
  - Ideally, minutes



# Debugging

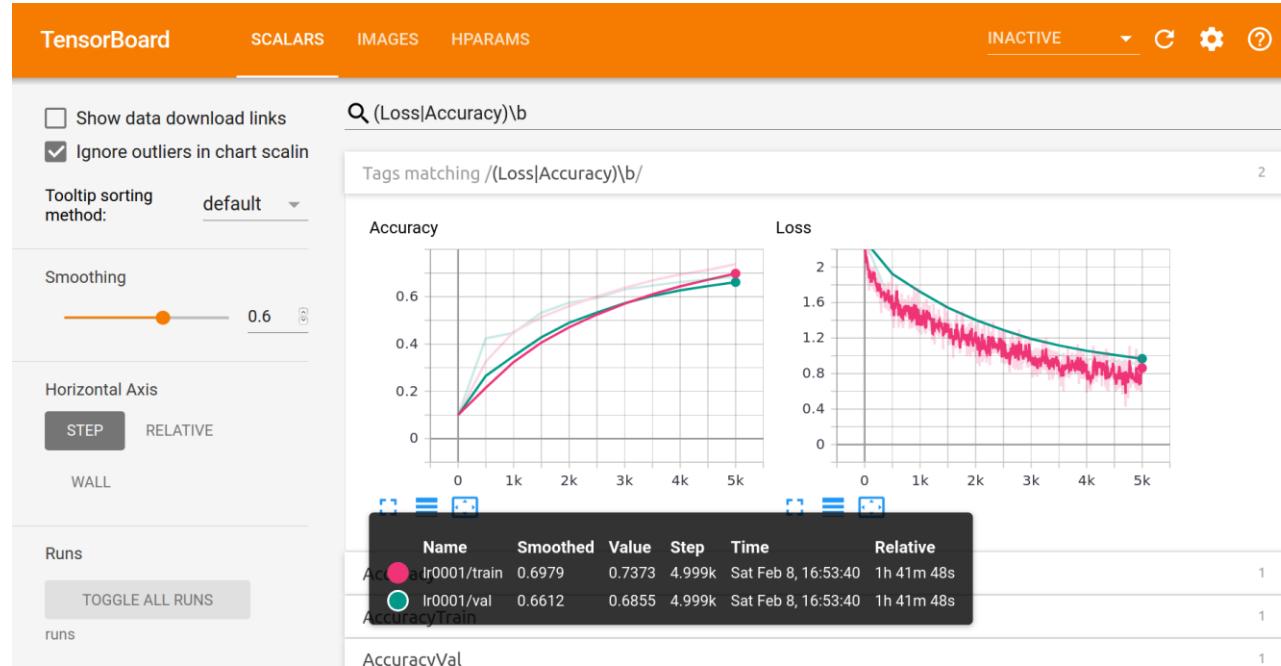
- Use train/validation/test curves
  - Evaluation needs to be consistent
  - Numbers need to be comparable
- Only make **one change at a time**
  - "I've added 5 more layers and double the training size, and now I also trained 5 days longer. Now it's better, but why?"
- Visualize input, prediction, ground truth

# Common Mistakes in Practice

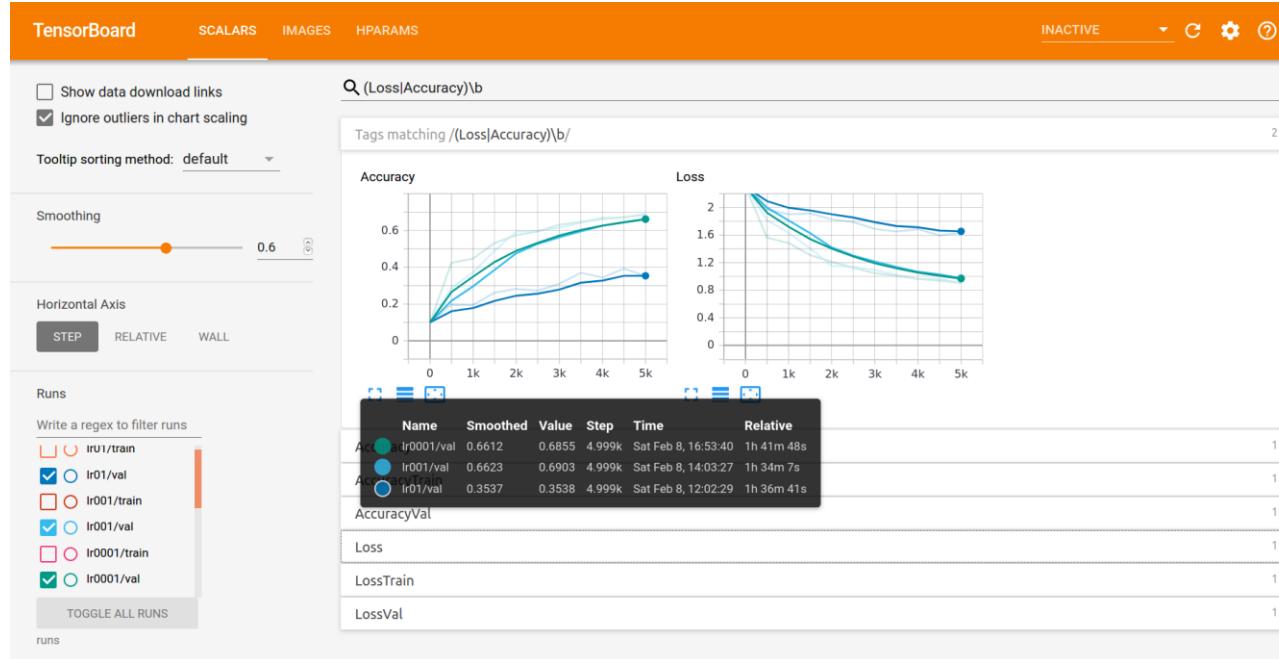
- Did not overfit to single batch first
- Forgot to toggle train/eval mode for network
  - Check later when we talk about dropout...
- Forgot to call `.zero_grad()` (*in PyTorch*) before calling `.backward()`
- Passed softmaxed outputs to a loss function that expects raw logits

# Tensorboard: Visualization in Practice

# Tensorboard: Compare Train/Val Curves



# Tensorboard: Compare Different Runs



# Tensorboard: Visualize Model Predictions

TensorBoard SCALARS IMAGES HPARAMS INACTIVE ⚙️ 🌐 🛡️ ?

Show actual image size

Brightness adjustment  RESET

Contrast adjustment  RESET

Runs  
Write a regex to filter runs

- Ir01/train
- Ir01/val
- Ir001/train
- Ir001/val
- Ir0001/train
- Ir0001/val
- Ir0001
- Ir0001/158118684 4.0108936
- Ir001
- Ir001/1581188042. 4137468
- Ir01
- Ir01/1581100000 0

[TOGGLE ALL RUNS](#)

runs

Misclassifications

Misclassifications/car step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/cat step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/deer step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/dog step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/frog step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

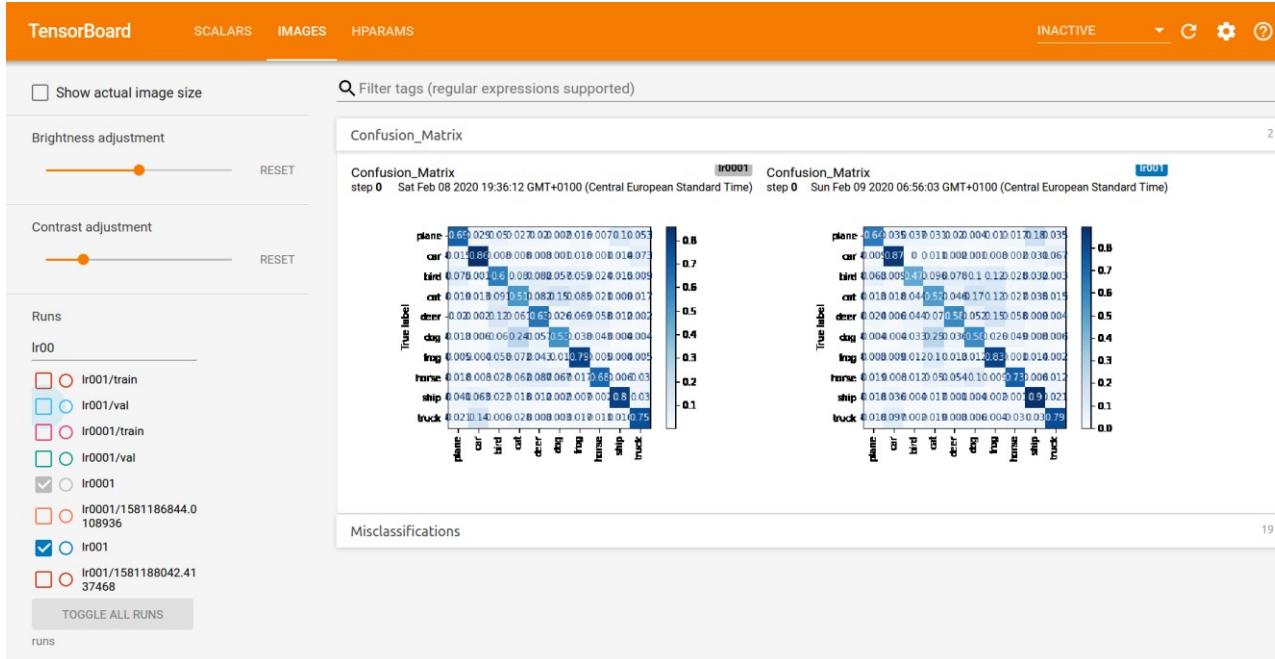
Misclassifications/horse step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/plane step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

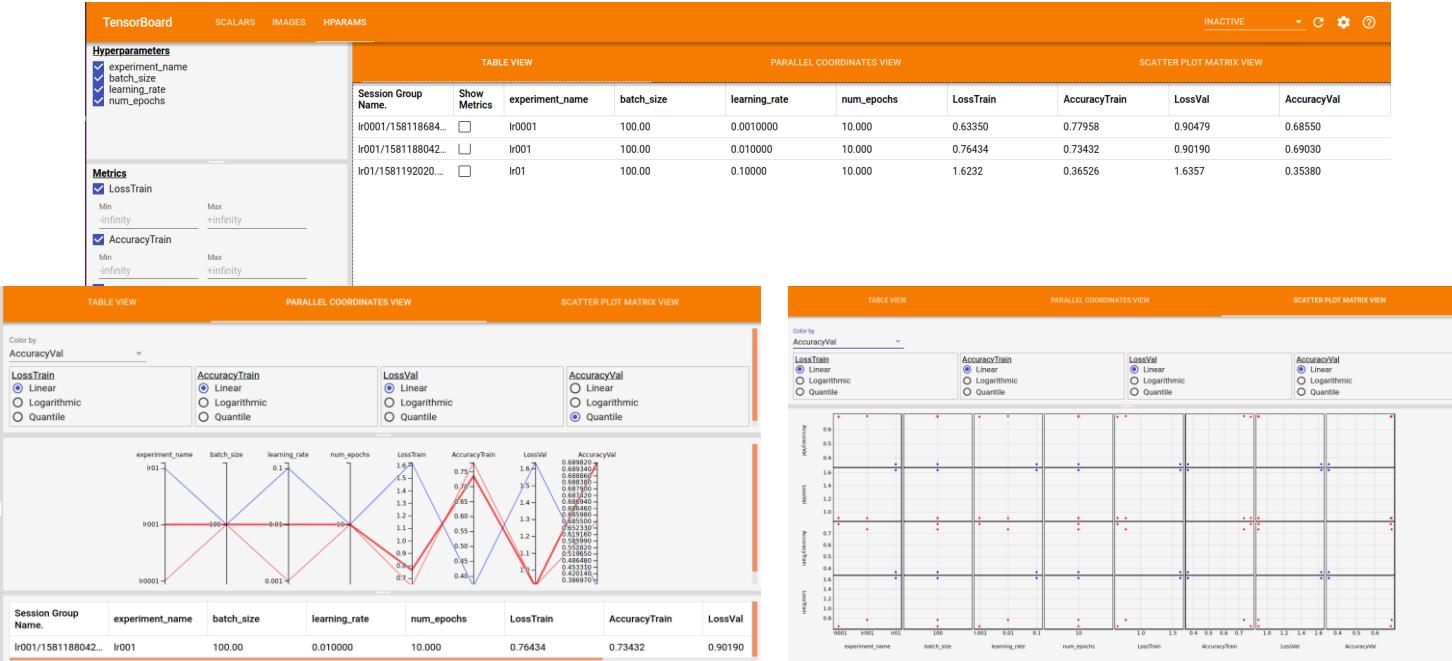
Misclassifications/ship step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

Misclassifications/truck step 0 Sun Feb 09 2020 06:56:04 GMT+0100 (Central European Standard Time)

# Tensorboard: Visualize Model Predictions



# Tensorboard: Compare Hyperparameters



# Next Lecture

- Next lecture
  - More about training neural networks: output functions, loss functions, activation functions
- Check the exercises ☺

See you next week ☺

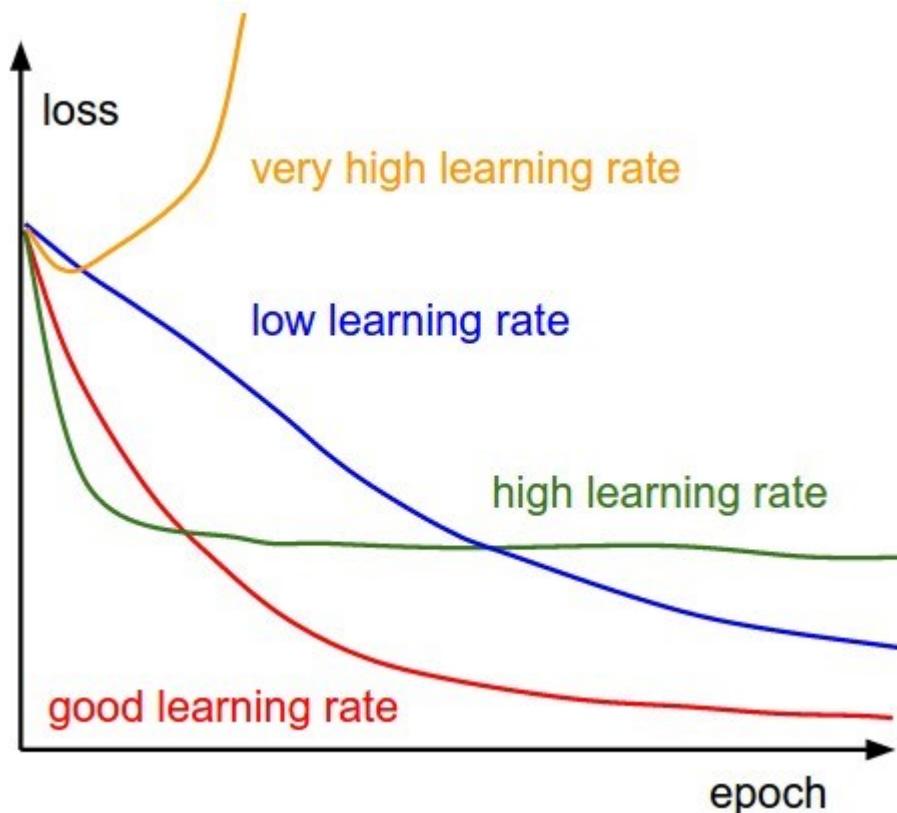
# References

- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 6: Deep Feedforward Networks
- Bishop "Pattern Recognition and Machine Learning" (2006),
  - Chapter 5.5: Regularization in Network Nets
- <http://cs231n.github.io/neural-networks-1/>
- <http://cs231n.github.io/neural-networks-2/>
- <http://cs231n.github.io/neural-networks-3/>

# Lecture 6 Recap

# Learning Rate: Implications

- What if too high?
- What if too low?



Source: <http://cs231n.github.io/neural-networks-3/>

# Training Schedule

Manually specify learning rate for entire training process

- Manually set learning rate every  $n$ -epochs
- How?
  - Trial and error (the hard way)
  - Some experience (only generalizes to some degree)

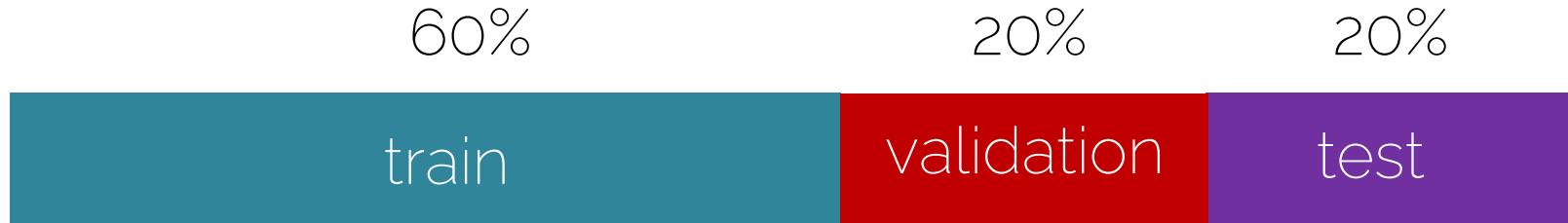
Consider: #epochs, training set size, network size, etc.

# Basic Recipe for Training

- Given dataset with ground truth labels
  - $\{\mathbf{x}_i, \mathbf{y}_i\}$ 
    - $\mathbf{x}_i$  is the  $i^{th}$  training image, with label  $\mathbf{y}_i$
    - Often  $\text{dim}(\mathbf{X}) \gg \text{dim}(\mathbf{y})$  (e.g., for classification)
    - $i$  is often in the 100-thousands or millions
  - Take network  $\mathbf{f}$  and its parameters  $\mathbf{W}, \mathbf{b}$
  - Use SGD (or variation) to find optimal parameters  $\mathbf{W}, \mathbf{b}$ 
    - Gradients from backprop

# Basic Recipe for Machine Learning

- Split your data



Example scenario

Ground truth error ..... 1%

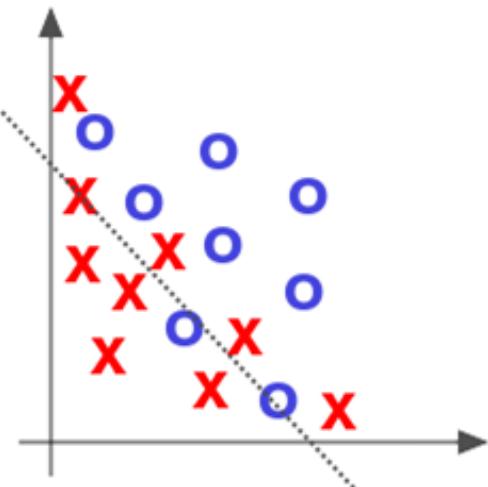
Training set error ..... 5%

Val/test set error ..... 8%

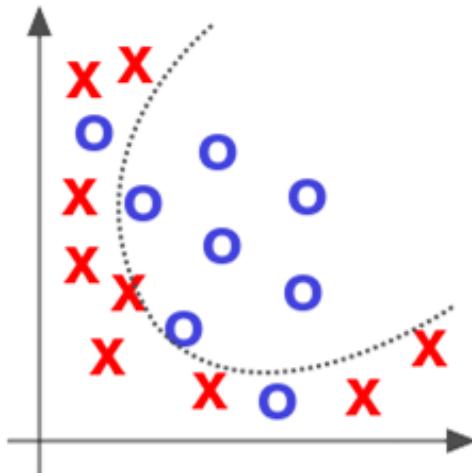
*Bias* (or underfitting)

*Variance*  
(overfitting)

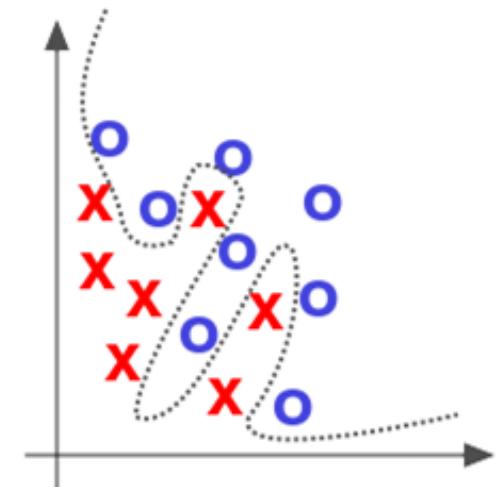
# Over and Underfitting



Underfitted  
Overfitted

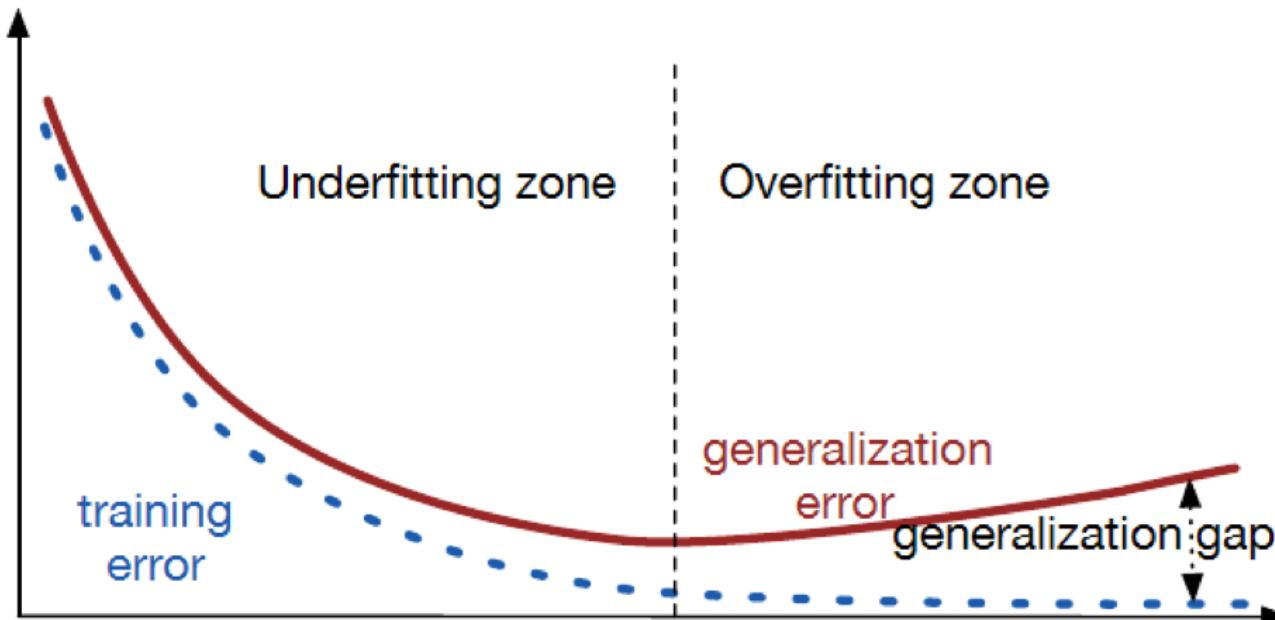


Appropriate



Source: Deep Learning by Adam Gibson, Josh Patterson, O'Reilly Media Inc., 2017

# Over and Underfitting



Source:

<https://srdas.github.io/DLBook/ImprovingModelGeneralization.html>

# Hyperparameters

- Network architecture (e.g., num layers, #weights)
- Number of iterations
- Learning rate(s) (i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size
- ...
- Overall: learning setup + optimization = hyperparameters

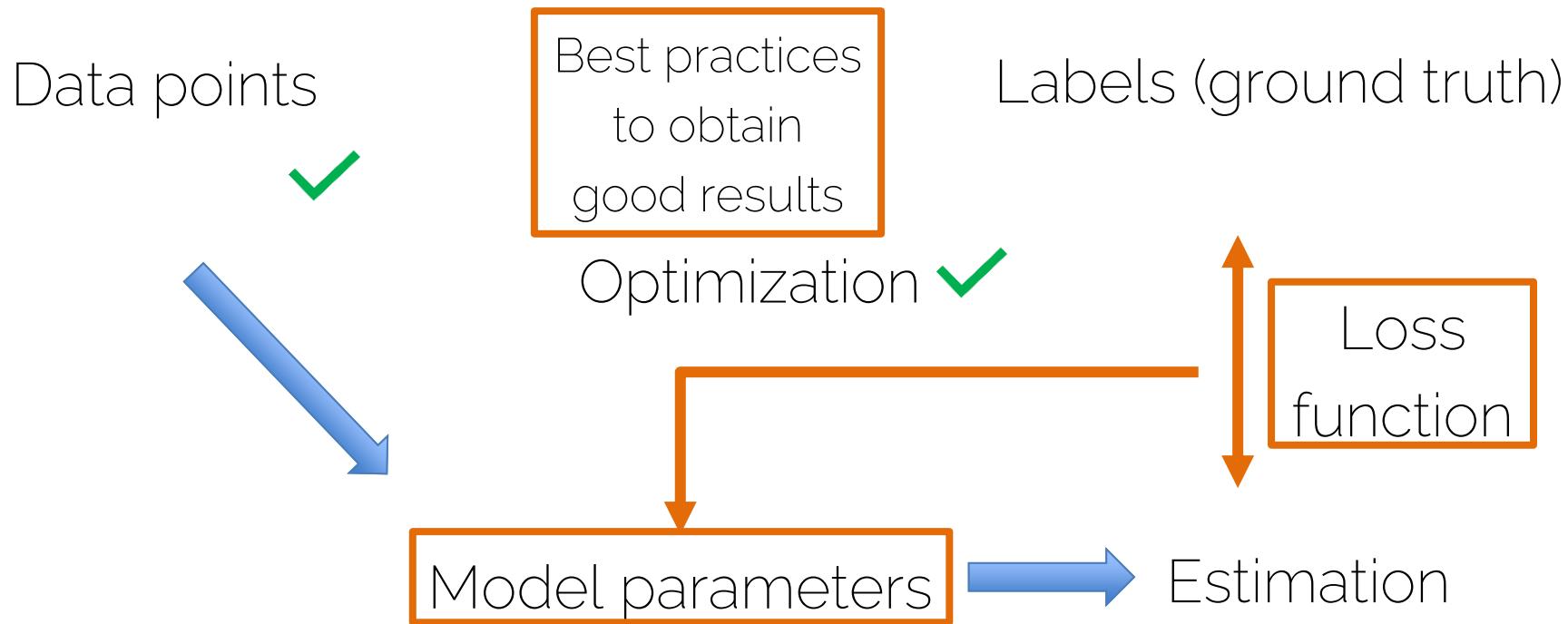
# Hyperparameter Tuning

- Methods:
  - Manual search: most common 😊
  - Grid search (structured, for 'real' applications)
    - Define ranges for all parameters spaces and select points
    - Usually pseudo-uniformly distributed
      - Iterate over all possible configurations
  - Random search:
    - Like grid search but one picks points at random in the predefined ranges
  - Auto-ML:
    - Bayesian, gradient-based etc

# Lecture 7

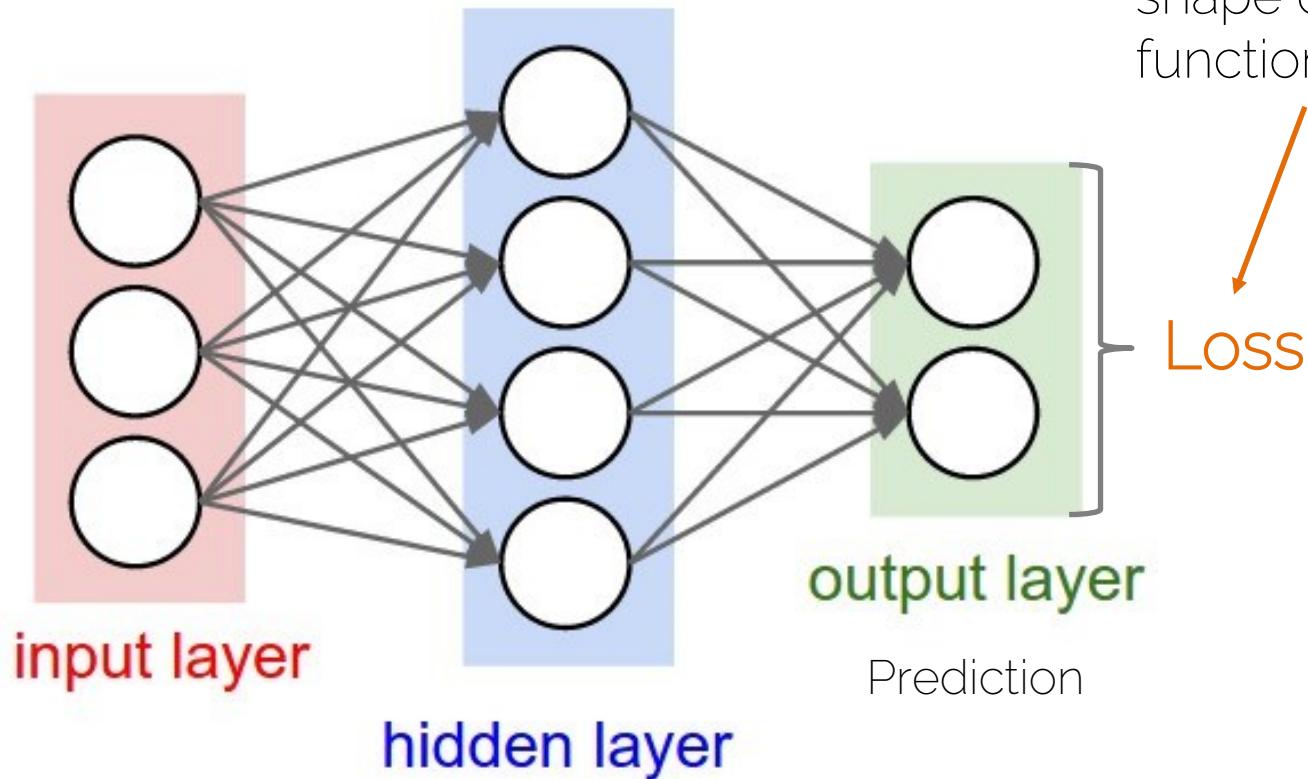
# Training NN (part 2)

# What we have seen so far



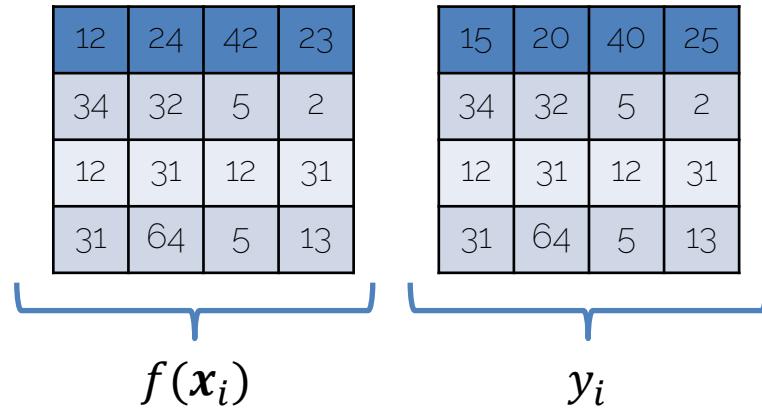
# Output and Loss Functions

# Neural Networks



# Regression Losses

- L2 Loss:  $L^2 = \sum_{i=1}^n (y_i - f(x_i))^2$  training pairs  $[x_i; y_i]$   
(input and labels)
- L1 Loss:  $L^1 = \sum_{i=1}^n |y_i - f(x_i)|$



$$L^2(x, y) = 9 + 16 + 4 + 4 + 0 + \dots + 0 = 33$$

$$L^1(x, y) = 3 + 4 + 2 + 2 + 0 + \dots + 0 = 11$$

# Regression Losses: L2 vs L1

- L2 Loss:

$$L^2 = \sum_{i=1}^n (y_i - f(\mathbf{x}_i))^2$$

- Sum of squared differences (SSD)
- Prone to outliers
- Compute-efficient optimization
- Optimum is the mean

- L1 Loss:

$$L^1 = \sum_{i=1}^n |y_i - f(\mathbf{x}_i)|$$

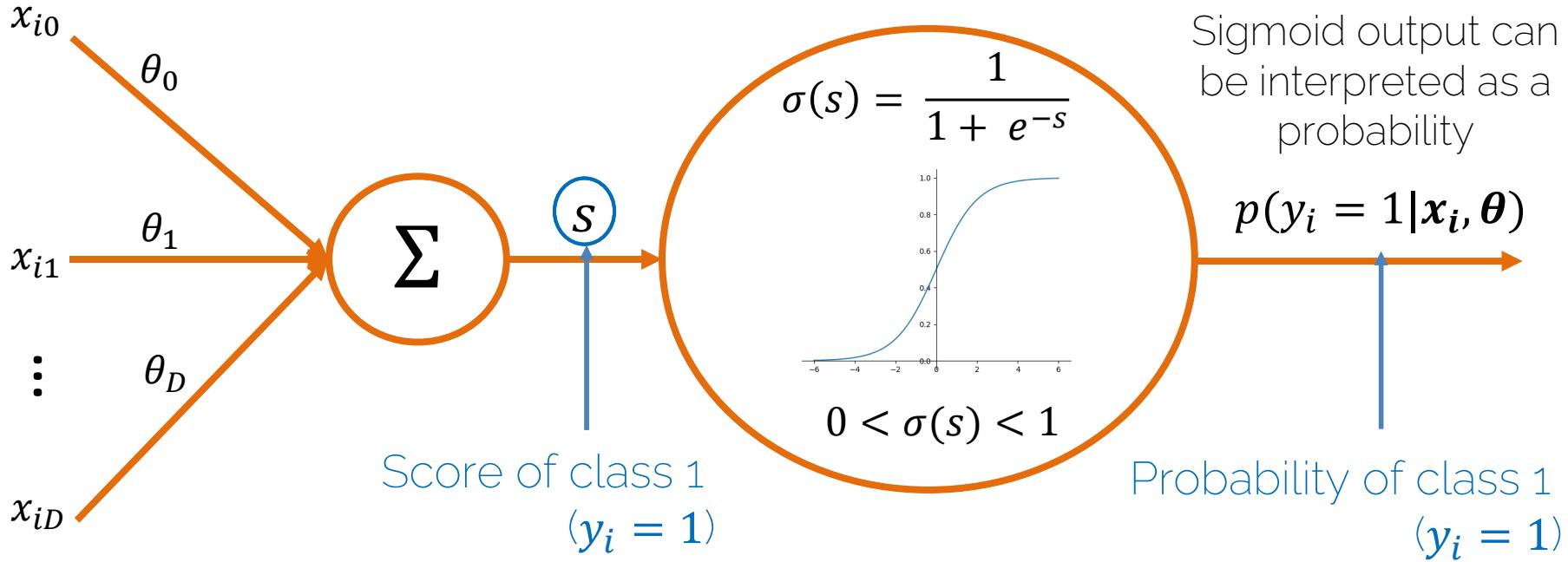
- Sum of absolute differences
- Robust (cost of outliers is linear)
- Costly to optimize
- Optimum is the median

# Binary Classification: Sigmoid

training pairs  $[x_i; y_i]$ ,

$x_i \in \mathbb{R}^D, y_i \in \{1, 0\}$  (2 classes)

$$p(y_i = 1 | x_i, \theta) = \sigma(s) = \frac{1}{1 + e^{-\sum_{d=0}^D \theta_d x_{id}}}$$



Sigmoid output can  
be interpreted as a  
probability

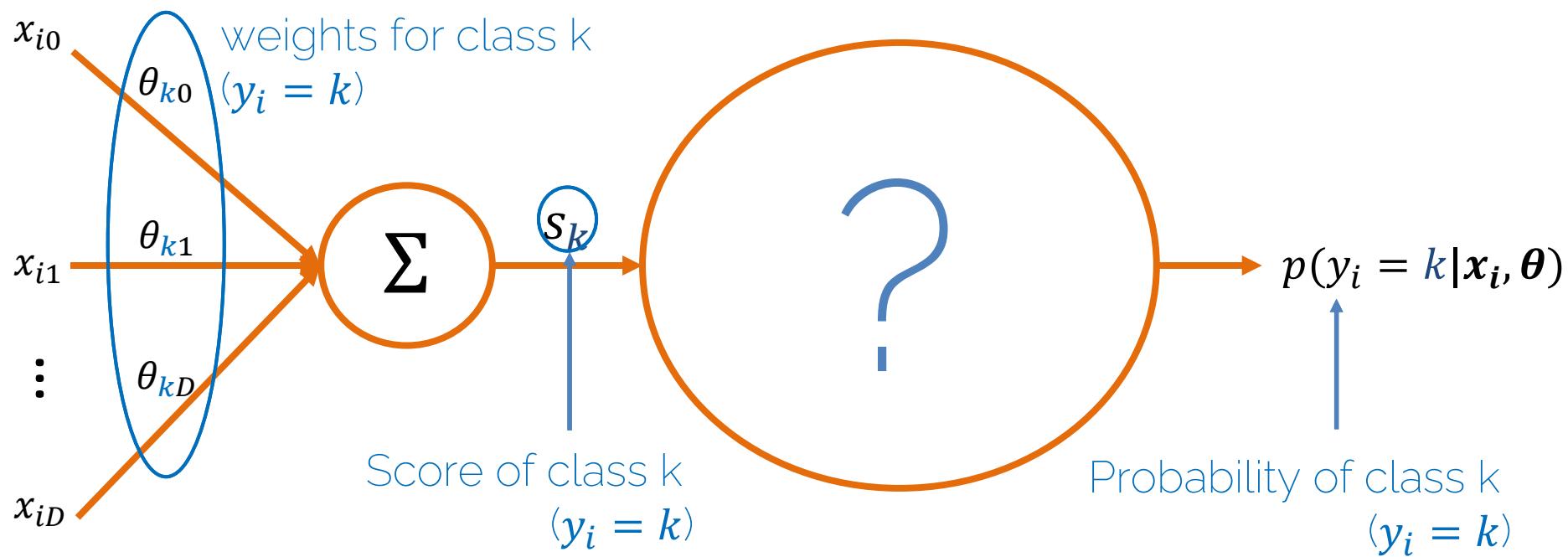
$$p(y_i = 1 | x_i, \theta)$$

Probability of class 1  
( $y_i = 1$ )

# Multiclass Classification: Softmax

training pairs  $[x_i; y_i]$ ,

$x_i \in \mathbb{R}^D, y_i \in \{1, 2, \dots, C\}$  ( $C$  classes)

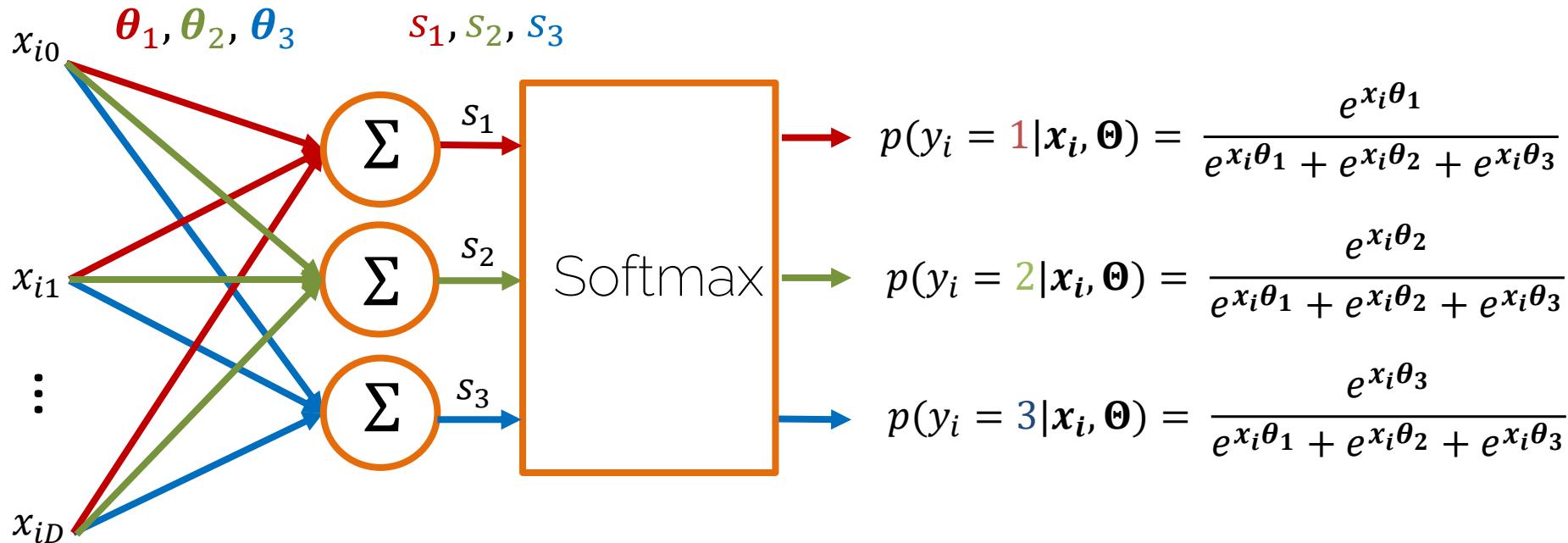


# Multiclass Classification: Softmax

Weights for each class

Scores for each class

Probabilities for each class



# Multiclass Classification: Softmax

- Softmax

$$p(y_i | \mathbf{x}_i, \Theta) = \frac{e^{s_{y_i}}}{\sum_{k=1}^C e^{s_k}} = \frac{e^{x_i \boldsymbol{\theta}_{y_i}}}{\sum_{k=1}^C e^{x_i \boldsymbol{\theta}_k}}$$

Probability of  
the true class

Exp

normalize

training pairs  $[\mathbf{x}_i; y_i]$ ,  
 $\mathbf{x}_i \in \mathbb{R}^D, y_i \in \{1, 2, \dots, C\}$   
 $y_i$ : label (true class)

Parameters:

$$\Theta = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_C]$$

$C$ : number of classes

$s$ : score of the class

- Exponential operation: make sure probability > 0
- Normalization: make sure probabilities sum up to 1.

# Multiclass Classification: Softmax

- Numerical Stability

$$p(y_i | \mathbf{x}_i, \Theta) = \frac{e^{s_{y_i}}}{\sum_{k=1}^C e^{s_k}} = \frac{e^{s_{y_i} - s_{max}}}{\sum_{k=1}^C e^{s_k - s_{max}}}$$

Try to prove it  
by yourself ☺

- Cross-Entropy Loss (Maximum Likelihood Estimate)

$$L_i = -\log(p(y_i | \mathbf{x}_i, \Theta)) = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$$

# Example: Cross-Entropy Loss

Cross Entropy

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$$

Score function

$$\mathbf{s} = \mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta})$$

e.g.,  $\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1

Given a function with weights  $\boldsymbol{\theta}$ , training pairs  $[\mathbf{x}_i; \mathbf{y}_i]$  (input and labels)  $\boldsymbol{\theta}_k = [w_k^b]$  parameters for each class with  $C$  classes

# Example: Cross-Entropy Loss

Cross Entropy

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_k e^{s_k}}\right)$$

Score function

$$\mathbf{s} = \mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta})$$

e.g.,  $\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_C]$

Suppose: 3 training examples and 3 classes

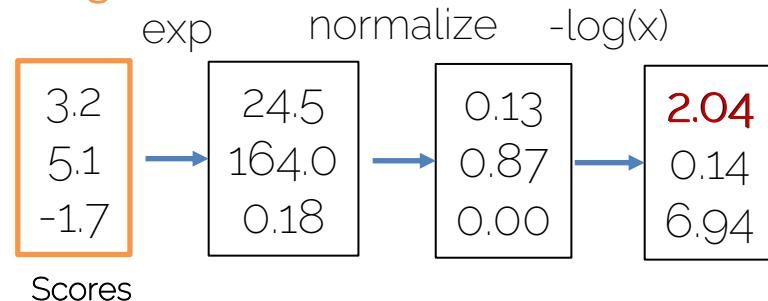


scores	cat	chair	car
	3.2	1.3	2.2
	5.1	4.9	2.5
	-1.7	2.0	-3.1

Loss 2.04

Given a function with weights  $\boldsymbol{\theta}$ , training pairs  $[\mathbf{x}_i; y_i]$  (input and labels)  $\boldsymbol{\theta}_k = [\mathbf{w}_k^T, b_k]$  parameters for each class with  $C$  classes

Image 1



# Example: Cross-Entropy Loss

Cross Entropy

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_k e^{s_k}}\right)$$

Score function

$$\mathbf{s} = \mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta})$$

e.g.,  $\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1
Loss		2.04	0.079	6.156

Given a function with weights  $\boldsymbol{\theta}$ , training pairs  $[\mathbf{x}_i; y_i]$  (input and labels)  $\boldsymbol{\theta}_k = [\mathbf{w}_k^T, b_k]$  parameters for each class with  $C$  classes

$$L = \frac{1}{N} \sum_{i=1}^N L_i = \frac{L_1 + L_2 + L_3}{3}$$

$$= \frac{2.04 + 0.079 + 6.156}{3} =$$

$$= 2.76$$

# Hinge Loss (SVM Loss)

- Score Function  $s = f(x_i, \theta)$ 
  - e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_c]$
- Hinge Loss (Multiclass SVM Loss)

$$L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

# Example: Hinge Loss (SVM Loss)

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Score function  $s = f(x_i, \theta)$

e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1

Given a function with weights  $\theta$ , training pairs  $[x_i; y_i]$  (input and labels)  $\theta_k = [w_k^b]$  parameters for each class with  $C$  classes

Loss

# Example: Hinge Loss (SVM Loss)

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Score function  $s = f(x_i, \theta)$

e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1

---

Loss 2.9

Given a function with weights  $\theta$ , training pairs  $[x_i; y_i]$  (input and labels)  $\theta_k = [w_k]$  parameters for each class with  $C$  classes

$$\begin{aligned} &= \max(0, 5.1 - 3.2 + 1) + \\ &\quad \max(0, -1.7 - 3.2 + 1) \\ &= \max(0, 2.9) + \max(0, -3.9) \\ &= 2.9 + 0 \\ &= \mathbf{2.9} \end{aligned}$$

# Example: Hinge Loss (SVM Loss)

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Score function  $s = f(x_i, \theta)$

e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1

Loss	2.9	0

Given a function with weights  $\theta$ , training pairs  $[x_i; y_i]$  (input and labels)  $\theta_k = [w_k^b]$  parameters for each class with  $C$  classes

$$\begin{aligned}L_2 &= \max(0, 1.3 - 4.9 + 1) + \\&\quad \max(0, 2.0 - 4.9 + 1) \\&= \max(0, -2.6) + \max(0, -1.9) \\&= 0 + 0 = 0\end{aligned}$$

# Example: Hinge Loss (SVM Loss)

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Score function  $s = f(x_i, \theta)$

e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_C]$

Suppose: 3 training examples and 3 classes



scores	cat	chair	car
	3.2	5.1	-1.7
	1.3	4.9	2.0
	2.2	2.5	-3.1

---

Loss	2.9	0	12.9
------	-----	---	------

Given a function with weights  $\theta$ , training pairs  $[x_i; y_i]$  (input and labels)  $\theta_k = [w_k]$  parameters for each class with  $C$  classes

$$\begin{aligned}L_3 &= \max(0, 2.2 - (-3.1) + 1) + \\&\quad \max(0, 2.5 - (-3.1) + 1) \\&= \max(0, 6.3) + \max(0, 6.6) \\&= 6.3 + 6.6 \\&= \mathbf{12.9}\end{aligned}$$

# Example: Hinge Loss (SVM Loss)

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Score function  $s = f(x_i, \theta)$

e.g.,  $f(x_i, \theta) = [x_{i0}, x_{i2}, \dots, x_{id}] \cdot [\theta_1, \theta_2, \dots, \theta_C]$

Suppose: 3 training examples and 3 classes



scores	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	car	-1.7	2.0	-3.1

Loss	2.9	0	12.9
------	-----	---	------

Given a function with weights  $\theta$ , training pairs  $[x_i; y_i]$  (input and labels)  $\theta_k = [w_k^k]$  parameters for each class with  $C$  classes

$$L = \frac{1}{N} \sum_{i=1}^N L_i = \frac{L_1 + L_2 + L_3}{3}$$

$$= \frac{2.9 + 0 + 12.9}{3} \\ = 5.3$$

# Multiclass Classification: Hinge vs Cross-Entropy

- Hinge Loss:  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$
- Cross Entropy Loss:  $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}})$

# Example: Hinge vs Cross-Entropy

$$\text{Hinge Loss: } L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

$$\text{Cross Entropy: } L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$$

For image  $\mathbf{x}_i$  (assume  $y_i = 0$ ):

	Scores	Hinge loss:	Cross Entropy loss:
Model 1	$s = [5, -3, 2]$		
Model 2	$s = [5, 10, 10]$		
Model 3	$s = [5, -20, -20]$		

# Example: Hinge vs Cross-Entropy

$$\text{Hinge Loss: } L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

$$\text{Cross Entropy: } L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$$

For image  $\mathbf{x}_i$  (assume  $y_i = 0$ ):

	Scores	Hinge loss:	Cross Entropy loss:
Model 1	$s = [5, -3, 2]$	$\max(0, -3 - 5 + 1) + \max(0, 2 - 5 + 1) = 0$	
Model 2	$s = [5, 10, 10]$	$\max(0, 10 - 5 + 1) + \max(0, 10 - 5 + 1) = 12$	
Model 3	$s = [5, -20, -20]$	$\max(0, -20 - 5 + 1) + \max(0, -20 - 5 + 1) = 0$	

Apparently Model 3 is better, but losses show no difference between Model 1&3, since they all have same loss=0.

# Example: Hinge vs Cross-Entropy

Hinge Loss:  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Cross Entropy :  $L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$

For image  $\mathbf{x}_i$  (assume  $y_i = 0$ ):

	Scores	Hinge loss:	Cross Entropy loss:
Model 1	$s = [5, -3, 2]$	$\max(0, -3 - 5 + 1) + \max(0, 2 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^3 + e^2}\right) = 0.05$
Model 2	$s = [5, 10, 10]$	$\max(0, 10 - 5 + 1) + \max(0, 10 - 5 + 1) = 12$	
Model 3	$s = [5, -20, -20]$	$\max(0, -20 - 5 + 1) + \max(0, -20 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^{-20} + e^{-20}}\right) = 2 * 10^{-11}$

Model 3 has a clearly smaller loss now.

# Example: Hinge vs Cross-Entropy

Hinge Loss:  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

Cross Entropy :  $L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$

For image  $\mathbf{x}_i$  (assume  $y_i = 0$ ):

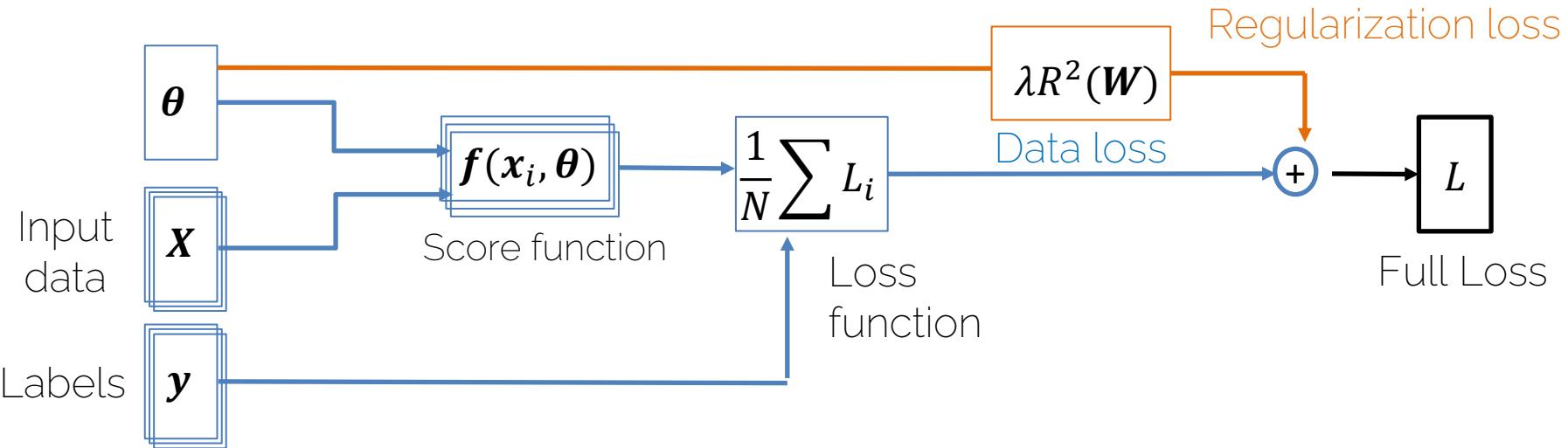
	Scores	Hinge loss:	Cross Entropy loss:
Model 1	$s = [5, -3, 2]$	$\max(0, -3 - 5 + 1) + \max(0, 2 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^3 + e^2}\right) = 0.05$
Model 2	$s = [5, 10, 10]$	$\max(0, 10 - 5 + 1) + \max(0, 10 - 5 + 1) = 12$	$-\ln\left(\frac{e^5}{e^5 + e^{10} + e^{10}}\right) = 5.70$
Model 3	$s = [5, -20, -20]$	$\max(0, -20 - 5 + 1) + \max(0, -20 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^{-20} + e^{-20}}\right) = 2 * 10^{-11}$

- Cross Entropy \*always\* wants to improve! (loss never 0)
- Hinge Loss saturates.

# Loss in Compute Graph

- How do we combine loss functions with weight regularization?
- How to optimize parameters of our networks according to multiple losses?

# Loss in Compute Graph



Want to find optimal  $\theta$ . (weights are unknowns of optimization problem)

- Compute gradient w.r.t.  $\theta$ .
- Gradient  $\nabla_{\theta} L$  is computed via backpropagation

# Loss in Compute Graph

- Score function  $s = f(\mathbf{x}_i, \theta)$
- Data Loss
  - Cross Entropy  $L_i = -\log(\frac{e^{sy_i}}{\sum_k e^{sk}})$
  - SVM  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$
- Regularization Loss: e.g.,  $L2\text{-Reg: } R^2(\mathbf{W}) = \sum \mathbf{w}_i^2$
- Full Loss  $L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R^2(\mathbf{W})$
- Full Loss = Data Loss + Reg Loss

Given a function with weights  $\theta$ ,

Training pairs  $[\mathbf{x}_i; y_i]$  (input and labels)

# Example: Regularization & SVM Loss

Multiclass SVM loss  $L_i = \sum_{k \neq y_i} \max(0, f(x_i; \theta)_k - f(x_i; \theta)_{y_i} + 1)$

Full loss  $L = \frac{1}{N} \sum_{i=1}^N \sum_{k \neq y_i} \max(0, f(x_i; \theta)_k - f(x_i; \theta)_{y_i} + 1) + \lambda R(\mathbf{W})$

$$L1\text{-Reg}: R^1(\mathbf{W}) = \sum_{i=1}^D |\mathbf{w}_i|$$

$$L2\text{-Reg}: R^2(\mathbf{W}) = \sum_{i=1}^D \mathbf{w}_i^2$$

Example:

$$\mathbf{x} = [1, 1, 1, 1]^T$$

$$R^2(\mathbf{w}_1) = 1$$

$$\mathbf{w}_1 = [1, 0, 0, 0]^T$$

$$R^2(\mathbf{w}_2) = 0.25^2 + 0.25^2 + 0.25^2 + 0.25^2$$

$$\mathbf{w}_2 = [0.25, 0.25, 0.25, 0.25]^T$$

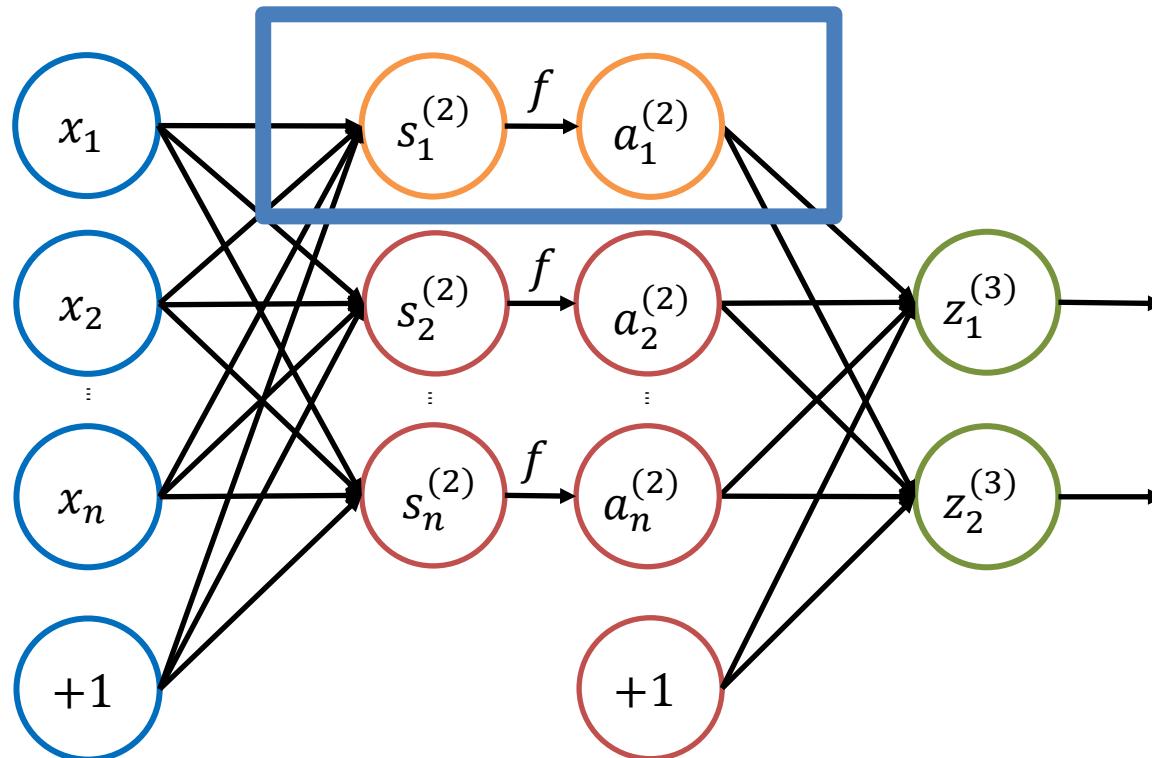
$$= 0.25$$

$$\mathbf{x}^T \mathbf{w}_1 = \mathbf{x}^T \mathbf{w}_2 = 1$$

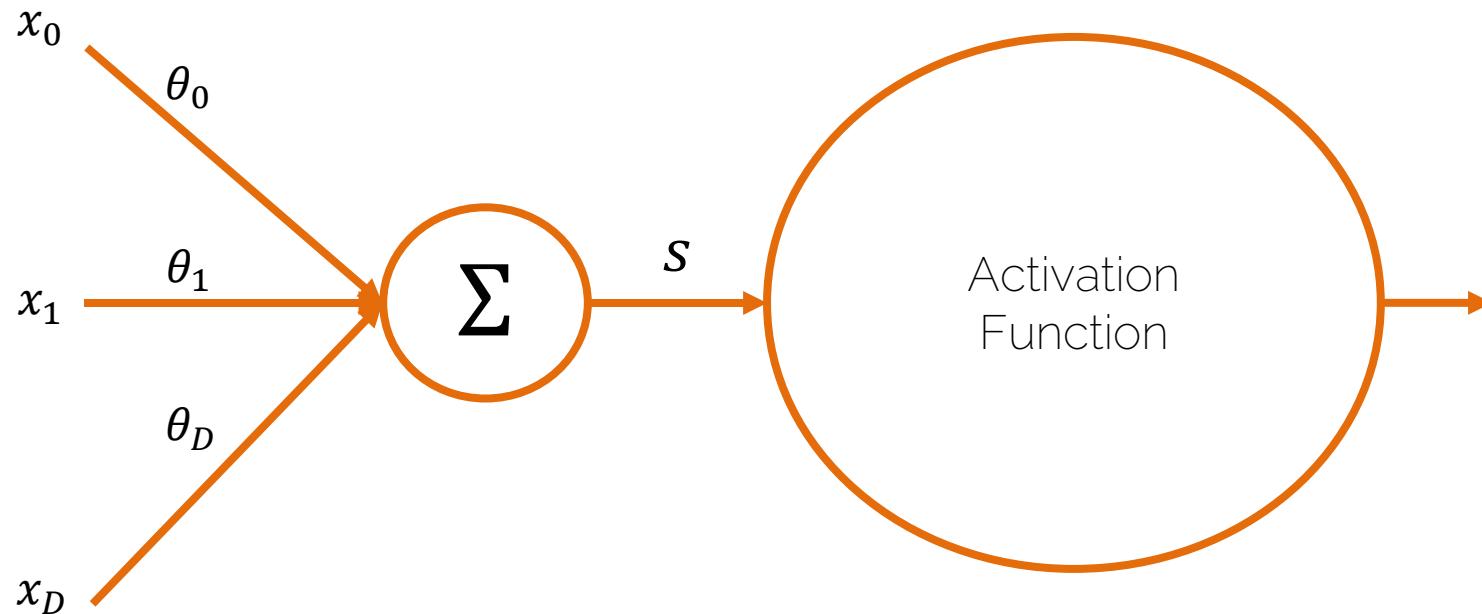
$$R^2(\mathbf{W}) = 1 + 0.25 = 1.25$$

# Activation Functions

# Neural Networks

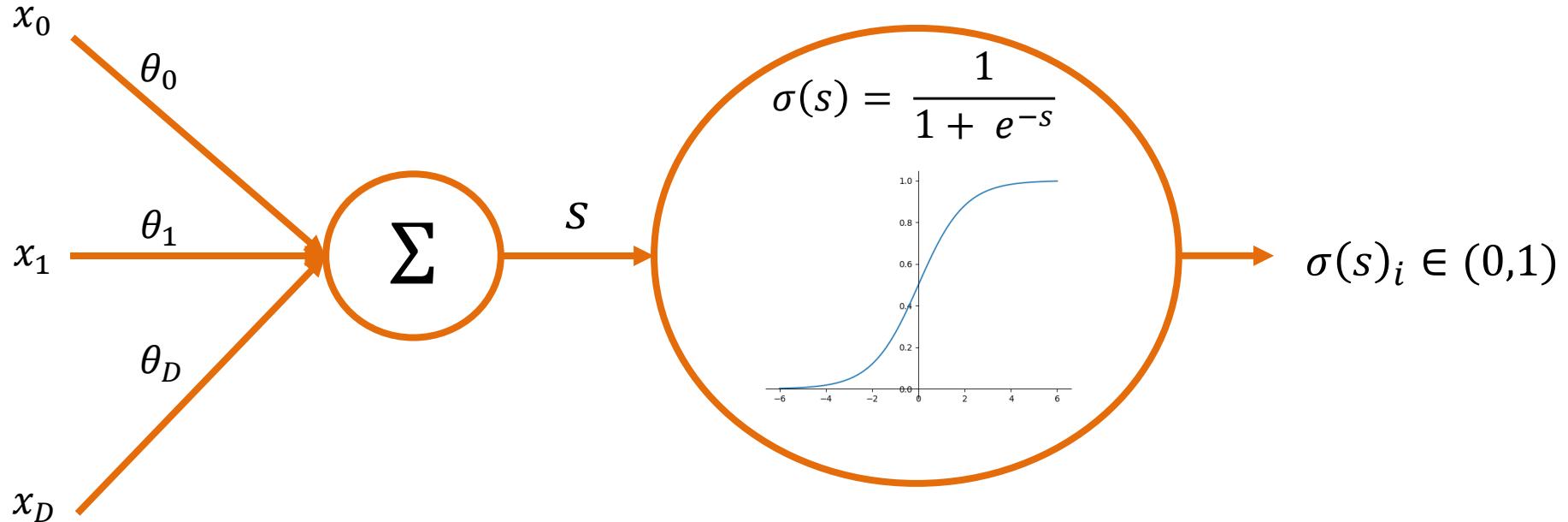


# Activation Functions or Hidden Units



# Sigmoid Activation

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



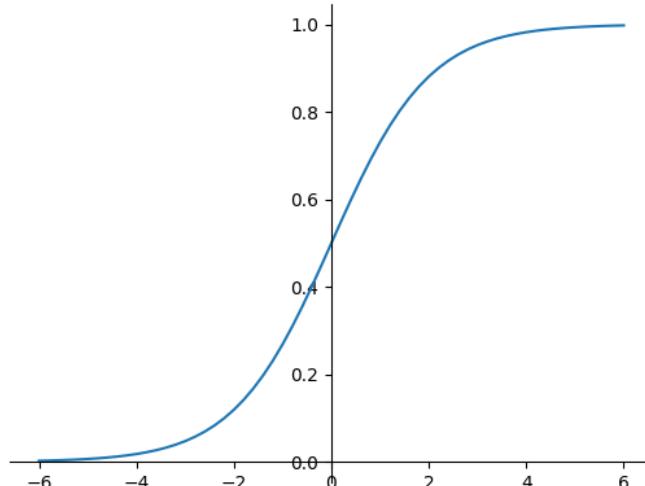
# Sigmoid Activation

Forward

$$\frac{\partial L}{\partial w} = \frac{\partial s}{\partial w} \frac{\partial L}{\partial s}$$

$\uparrow$        $\uparrow$   
 $x^T$  ?

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

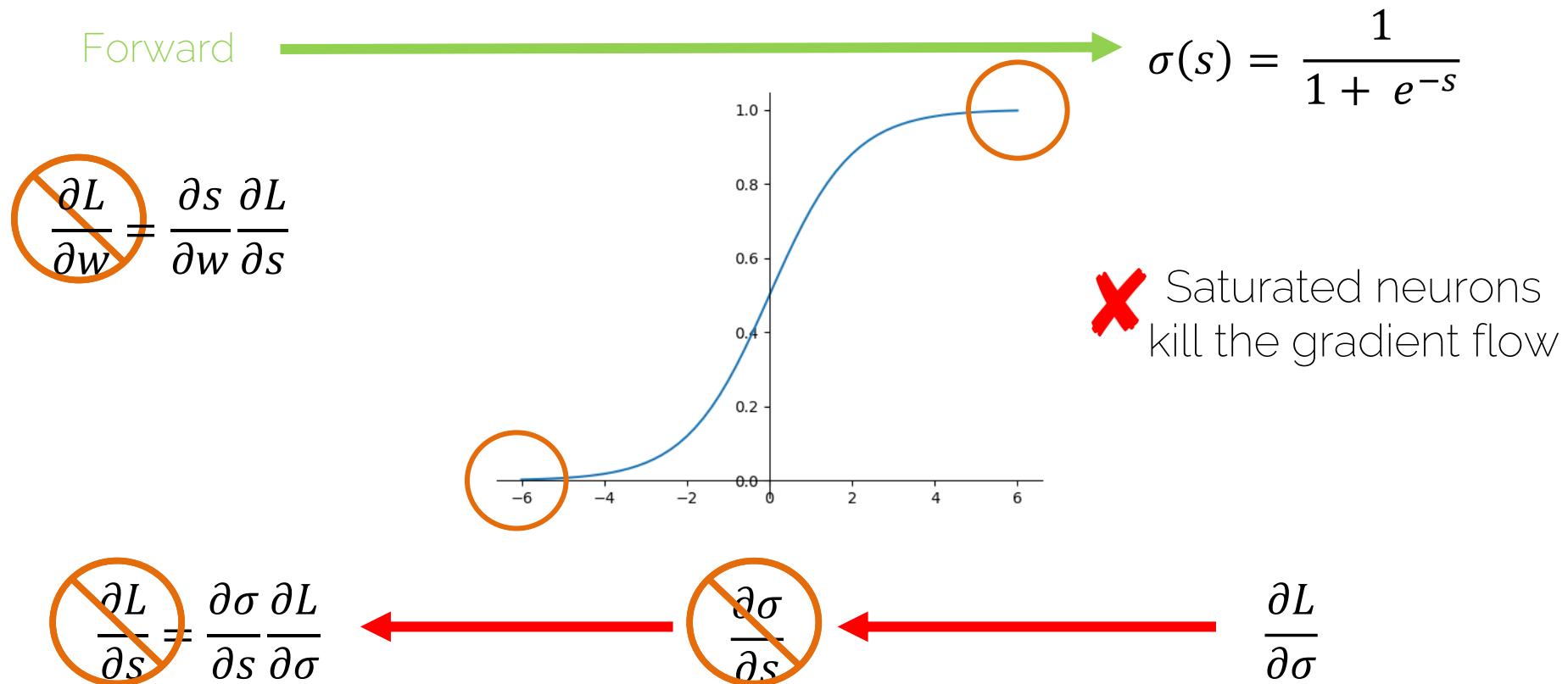


$$\frac{\partial L}{\partial s} = \frac{\partial \sigma}{\partial s} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial \sigma}{\partial s}$$

$$\frac{\partial L}{\partial \sigma}$$

# Sigmoid Activation

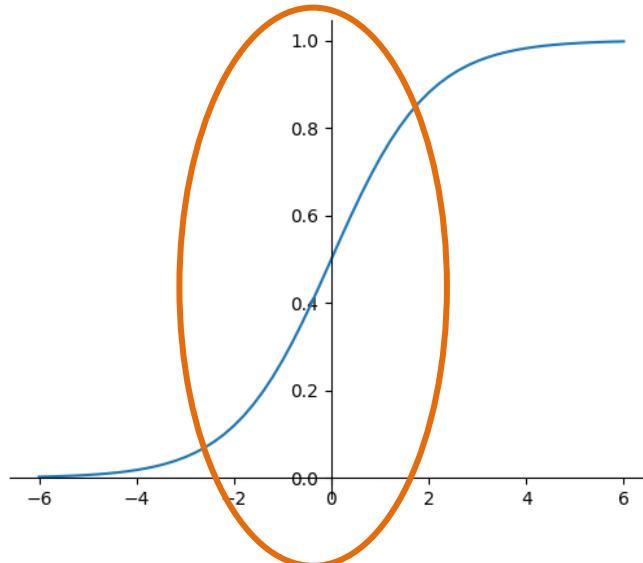


# Sigmoid Activation

Forward

$$\frac{\partial L}{\partial w} = \frac{\partial s}{\partial w} \frac{\partial L}{\partial s}$$

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



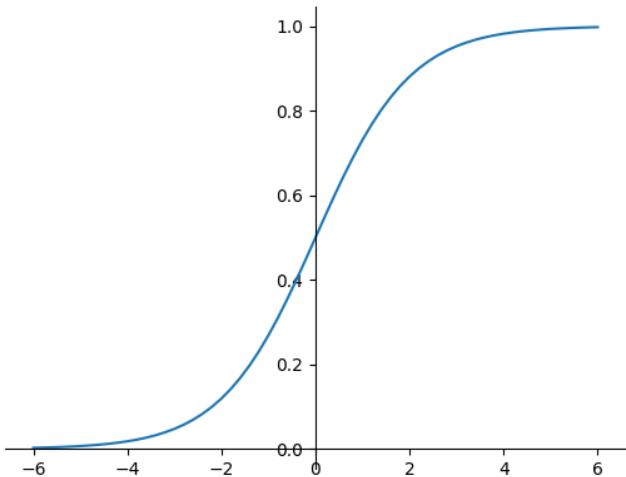
Active region for  
gradient descent

$$\frac{\partial L}{\partial s} = \frac{\partial \sigma}{\partial s} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial \sigma}{\partial s}$$

$$\frac{\partial L}{\partial \sigma}$$

# Sigmoid Activation



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

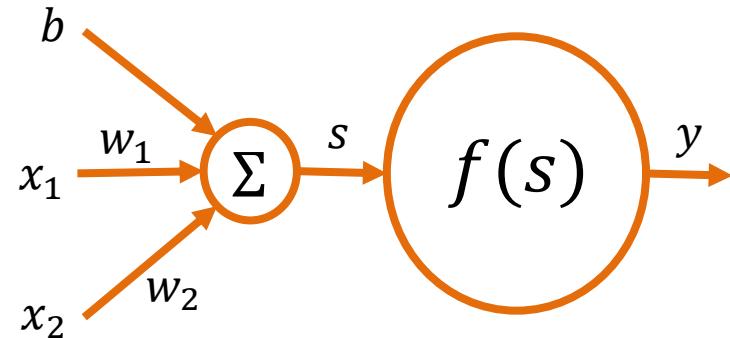
Output is always positive!

- Sigmoid output provides positive input for the next layer

What is the disadvantage of this?

# Sigmoid Output not Zero-centered

- We want to compute the gradient w.r.t. the weights



Assume we have all positive data:

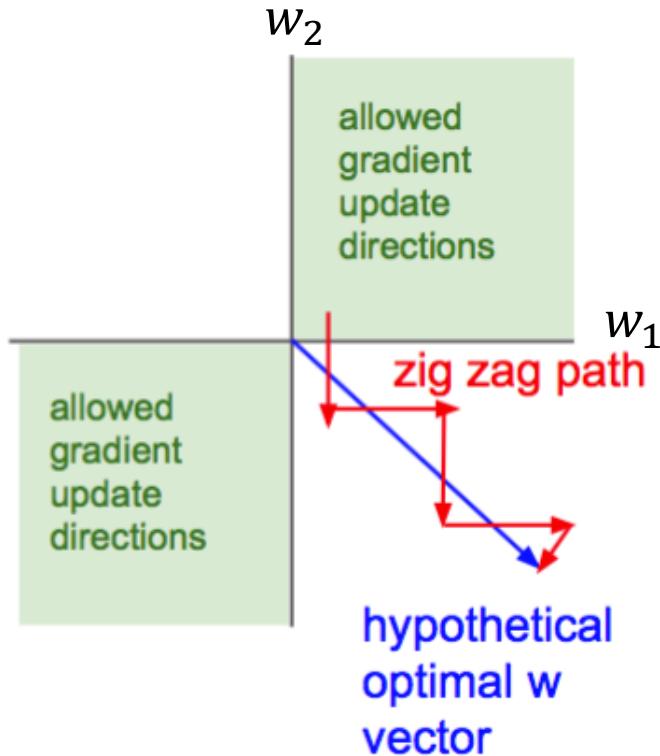
$$\mathbf{x} = (x_1, x_2)^T > 0$$

either positive  
or negative

$$\frac{\partial L}{\partial w_1} = \boxed{\frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s}} \cdot \underbrace{\frac{\partial s}{\partial w_1}}_{x_1 > 0}$$
$$\frac{\partial L}{\partial w_2} = \boxed{\frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s}} \cdot \underbrace{\frac{\partial s}{\partial w_2}}_{x_2 > 0}$$

It is going to be either positive or negative for all weights' update. 😞

# Sigmoid Output not Zero-centered



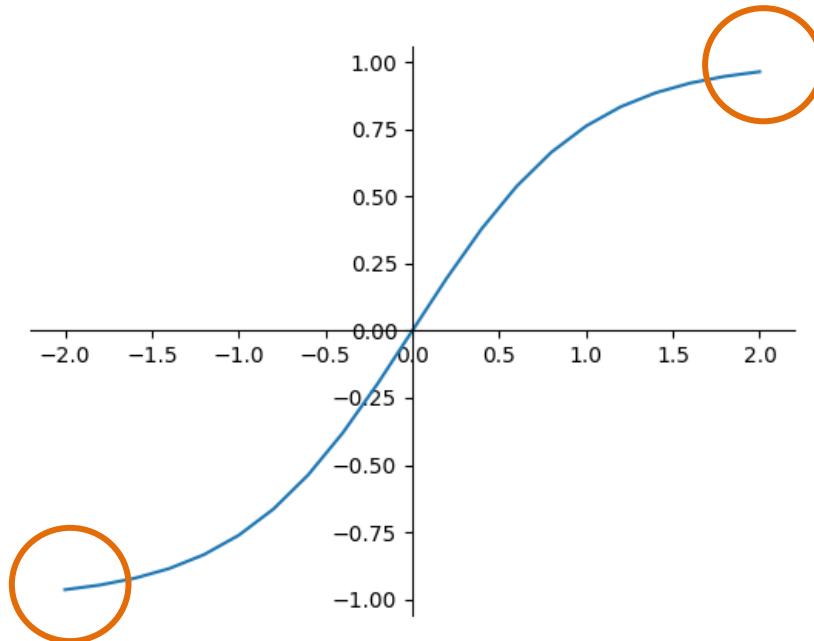
$w_1, w_2$  can only be increased or decreased at the same time, which is not good for update.

That is also why you need zero-centered data.

Source :

[http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture6.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf)

# TanH Activation



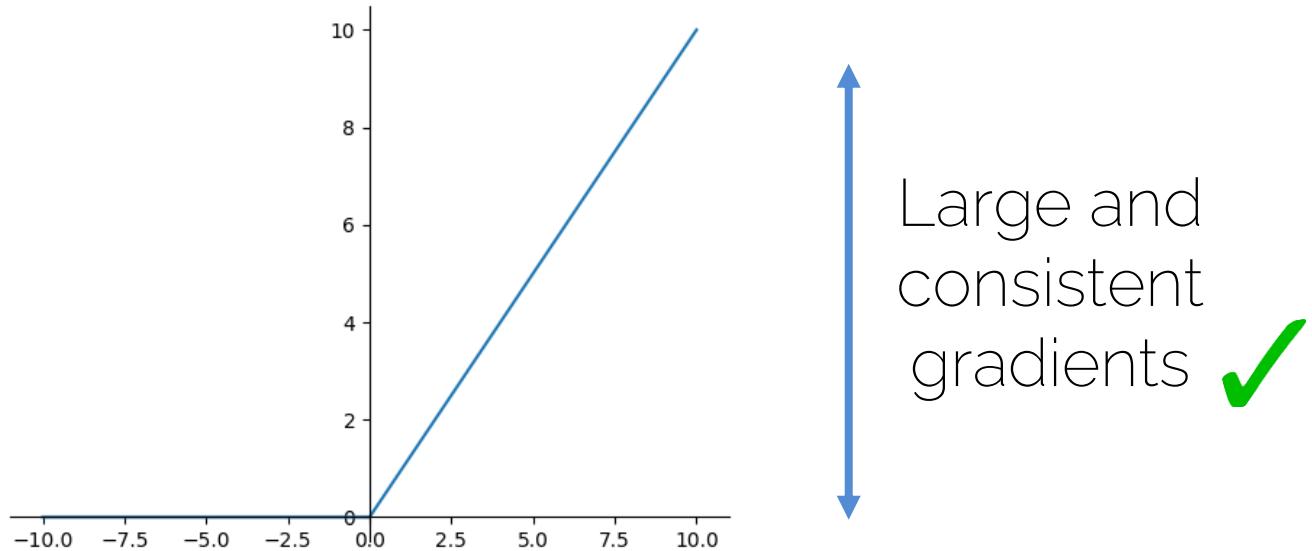
✗ Still saturates

✓ Zero-centered

[LeCun et al. 1991] Improving Generalization Performance in Character Recognition

# Rectified Linear Units (ReLU)

$$\sigma(x) = \max(0, x)$$



✓ Fast convergence

✓ Does not saturate

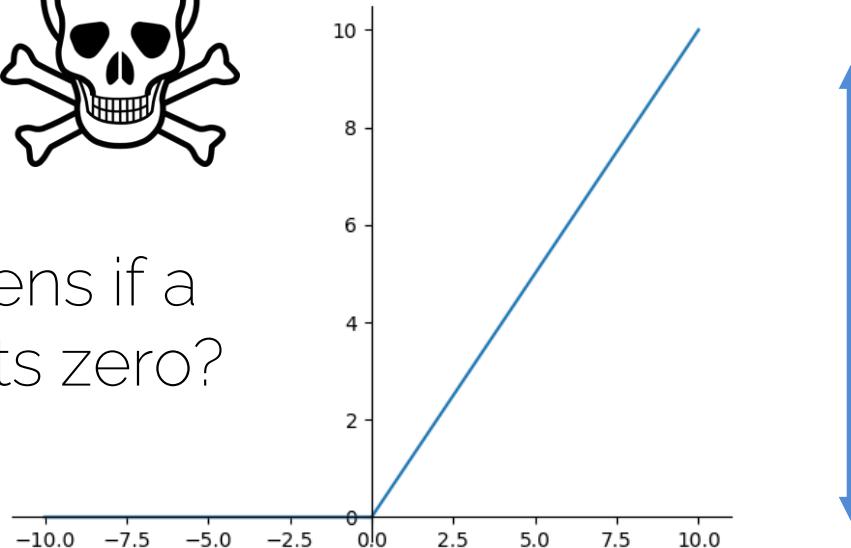
[Krizhevsky et al. NeurIPS 2012] ImageNet Classification with Deep Convolutional Neural Networks

# Rectified Linear Units (ReLU)

✗ Dead ReLU



What happens if a  
ReLU outputs zero?



Large and  
consistent  
gradients



✓ Fast convergence

✓ Does not saturate

[Krizhevsky et al. NeurIPS 2012] ImageNet Classification with Deep Convolutional Neural Networks

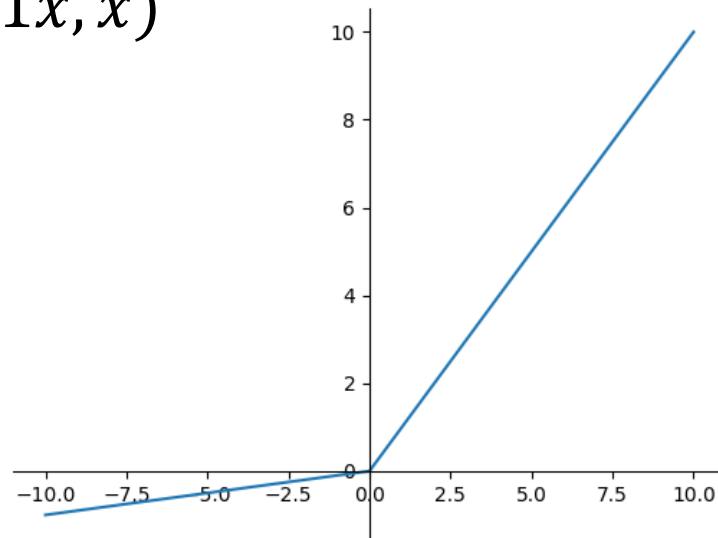
# Rectified Linear Units (ReLU)

- Initializing ReLU neurons with slightly positive biases (0.01) makes it likely that they stay active for most inputs

$$f\left(\sum_i w_i x_i + b\right)$$

# Leaky ReLU

$$\sigma(x) = \max(0.01x, x)$$



Does not die

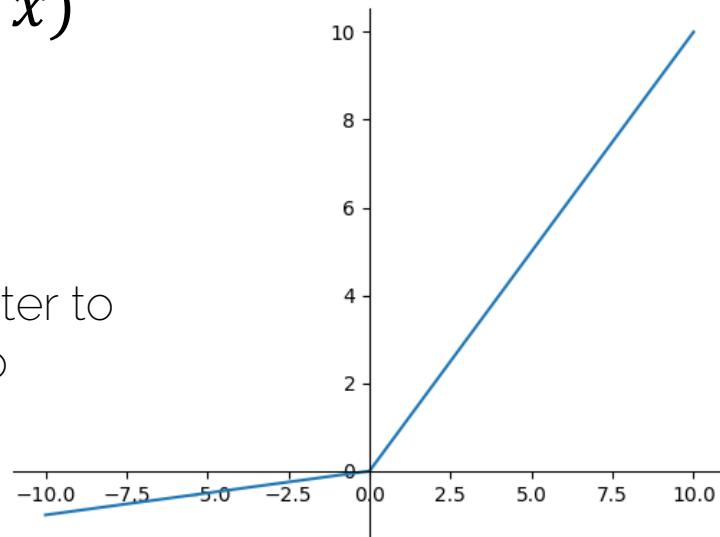
[Mass et al., ICML 2013] Rectifier Nonlinearities Improve Neural Network Acoustic Models

# Parametric ReLU

$$\sigma(x) = \max(\alpha x, x)$$



One more parameter to  
backprop into

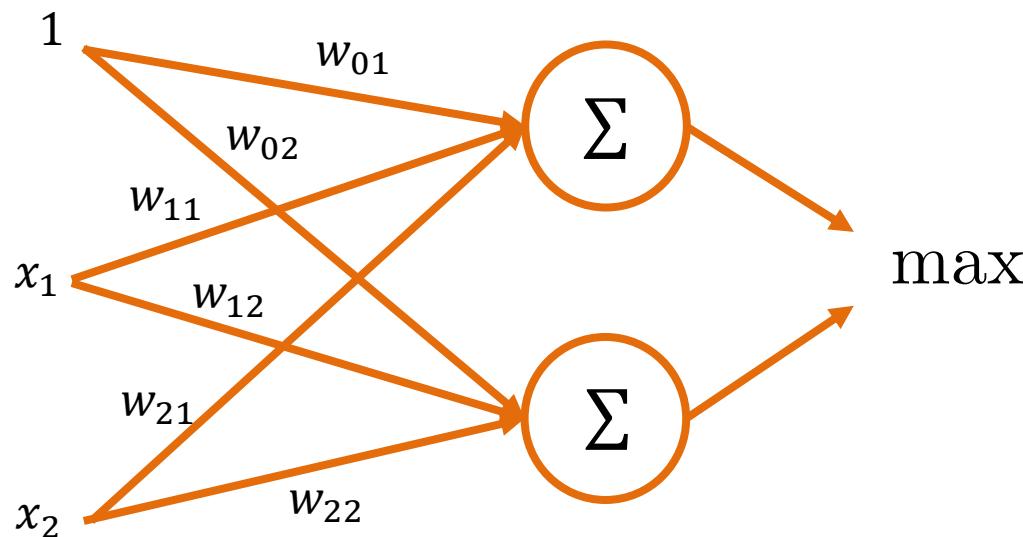


Does not die

[He et al. ICCV 2015] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

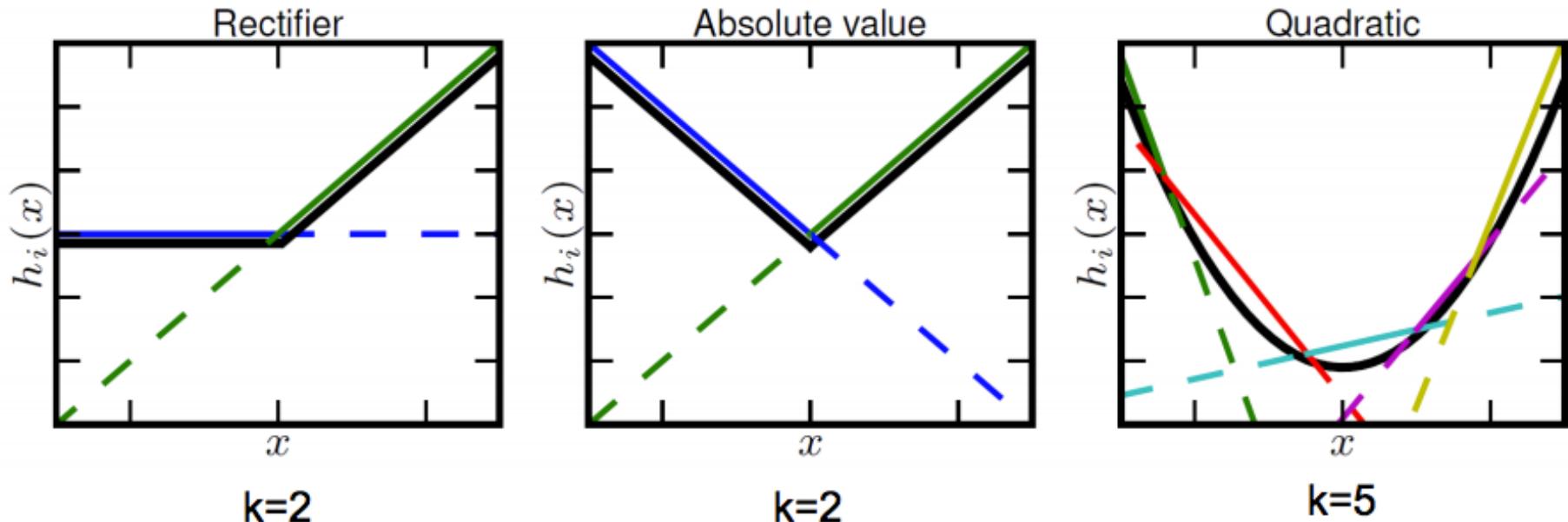
# Maxout Units

$$\text{Maxout} = \max(w_1^T x + b_1, w_2^T x + b_2)$$



[Goodfellow et al. ICML 2013] Maxout Networks

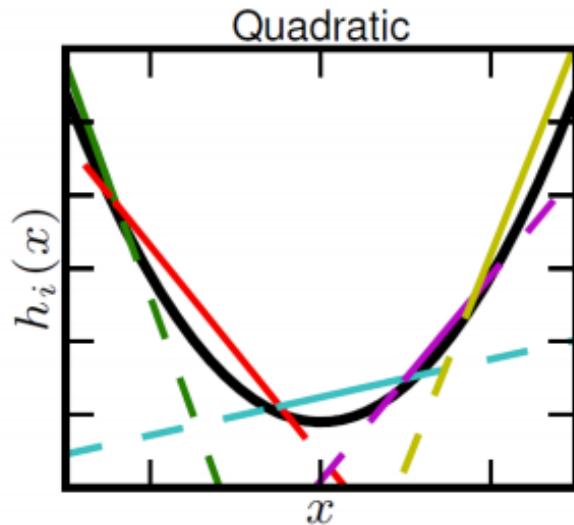
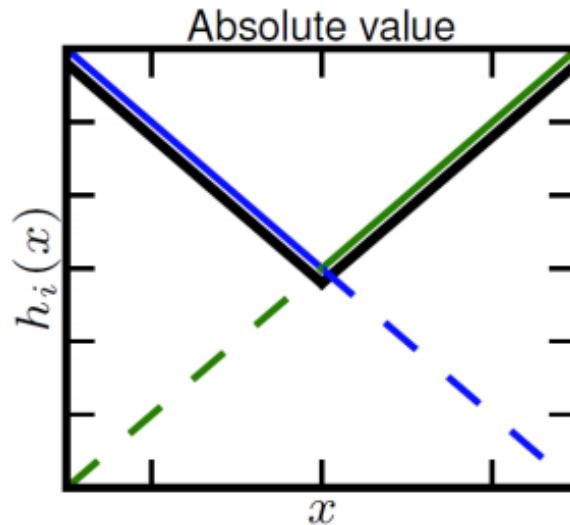
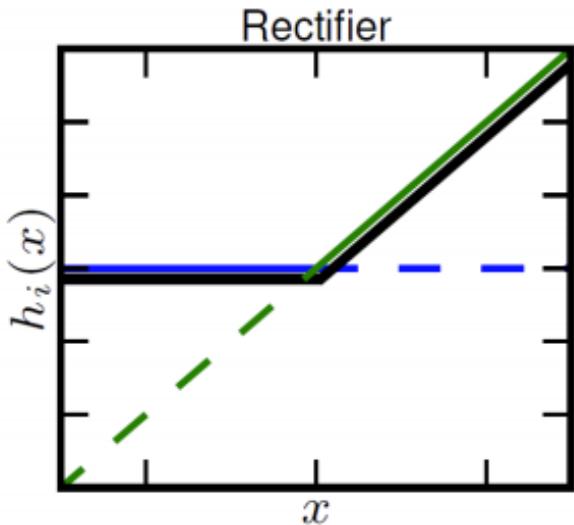
# Maxout Units



Piecewise linear approximation of  
a convex function with N pieces

[Goodfellow et al. ICML 2013] Maxout Networks

# Maxout Units



**k=2**

✓ Generalization  
of ReLUs

✓ Linear  
regimes

**k=2**

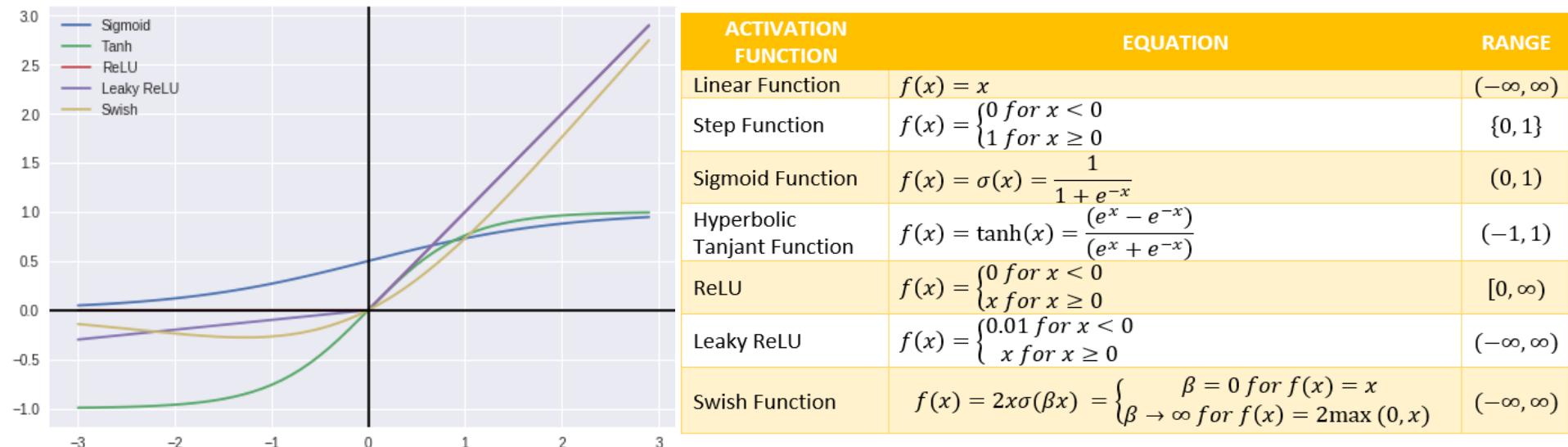
✓ Does not  
die

**k=5**

✓ Does not  
saturate

✗ Increases of the number of parameters

# In a Nutshell



Source : <https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a>

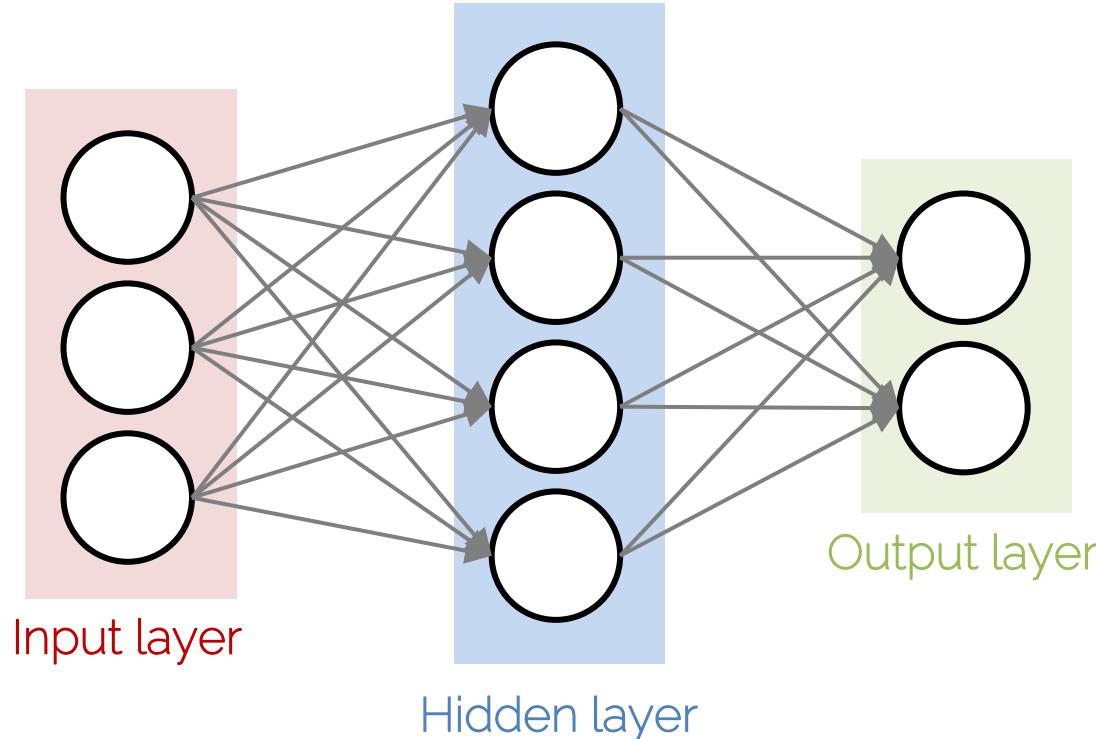
# Quick Guide

- Sigmoid/TanH are not really used in feedforward nets.
- ReLU is the standard choice.
- Second choice are the variants of ReLU or Maxout.
- Recurrent nets will require Sigmoid/TanH or similar.

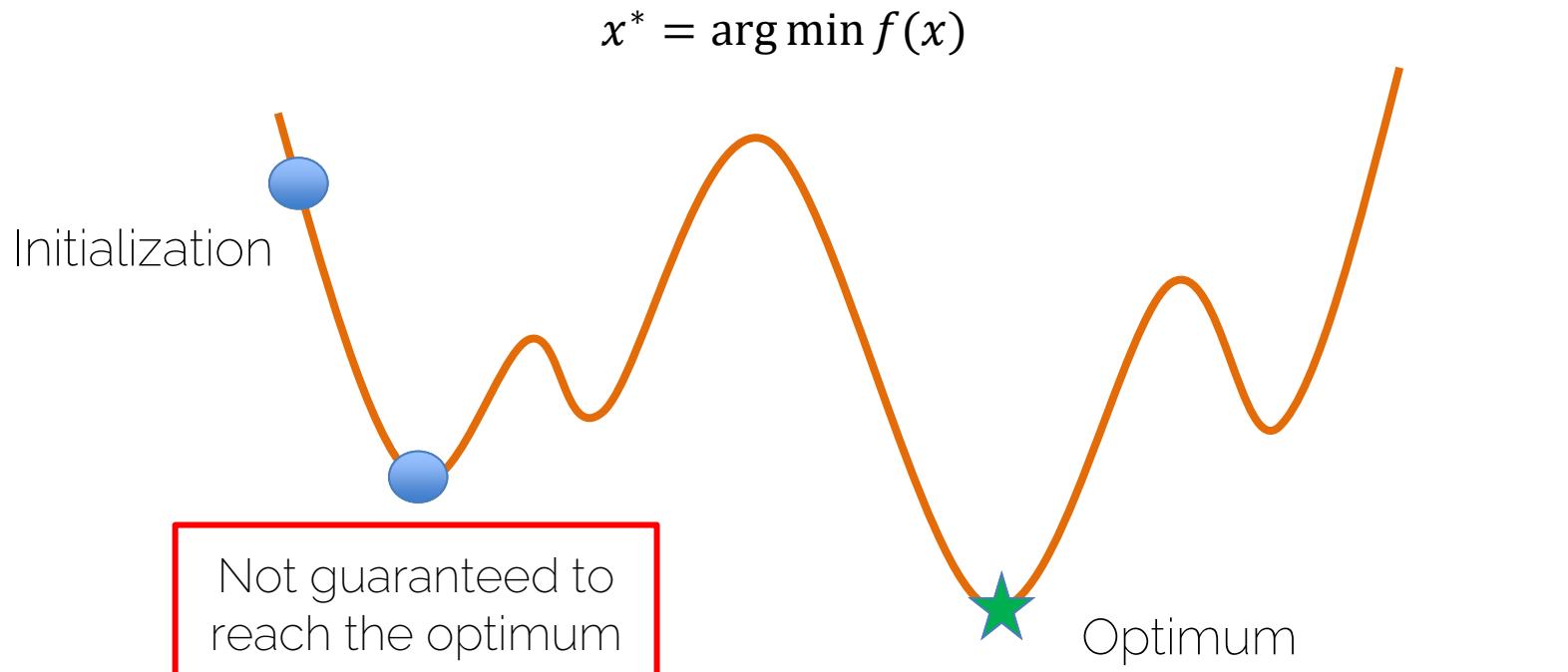
# Weight Initialization

# How do I start?

Forward

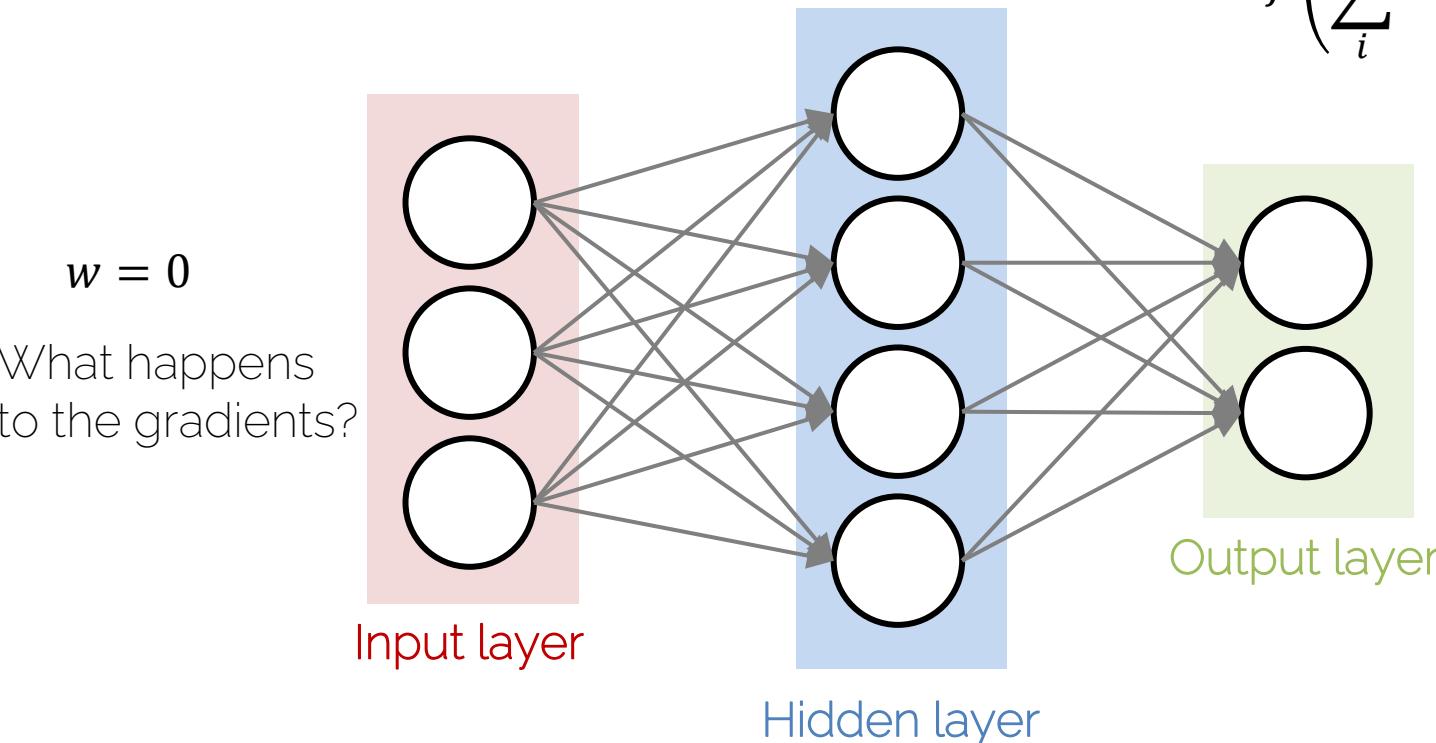


# Initialization is Extremely Important!



# How do I start?

Forward   $f\left(\sum_i w_i x_i + b\right)$



# All Weights Zero

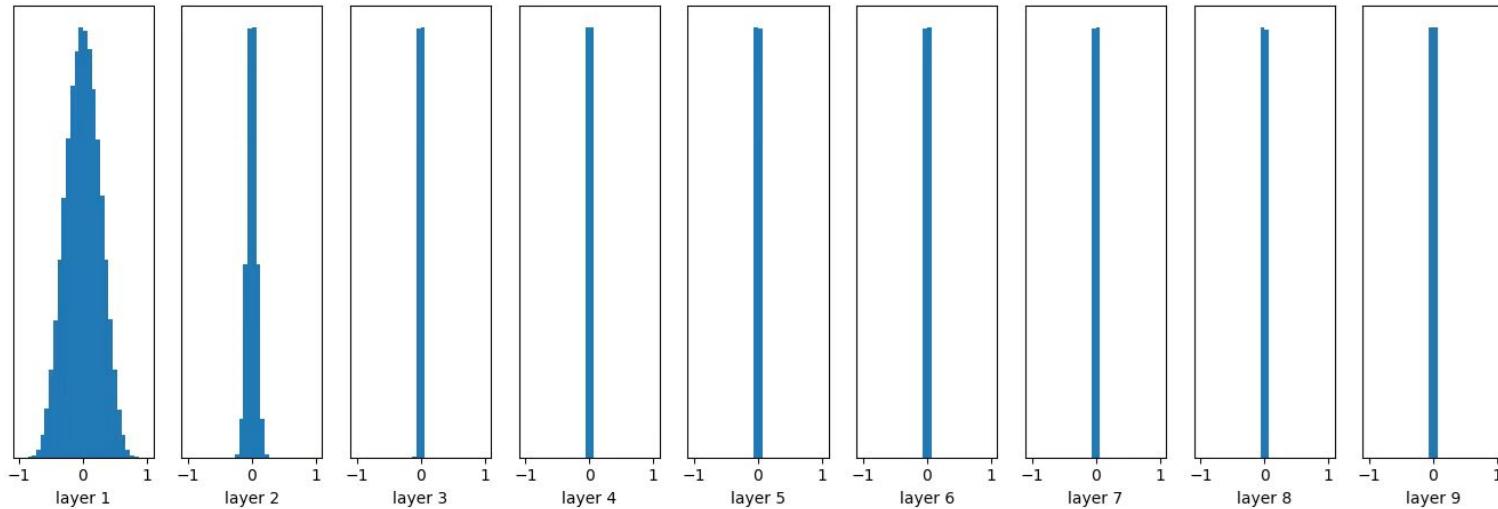
- What happens to the gradients?
- The hidden units are all going to compute the same function, gradients are going to be the same
  - No symmetry breaking

# Small Random Numbers

- Gaussian with zero mean and standard deviation 0.01
- Let's see what happens:
  - Network with 10 layers with 500 neurons each
  - Tanh as activation functions
  - Input unit Gaussian data

# Small Random Numbers

tanh as activation functions

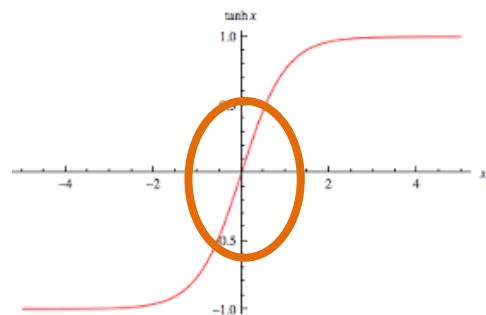
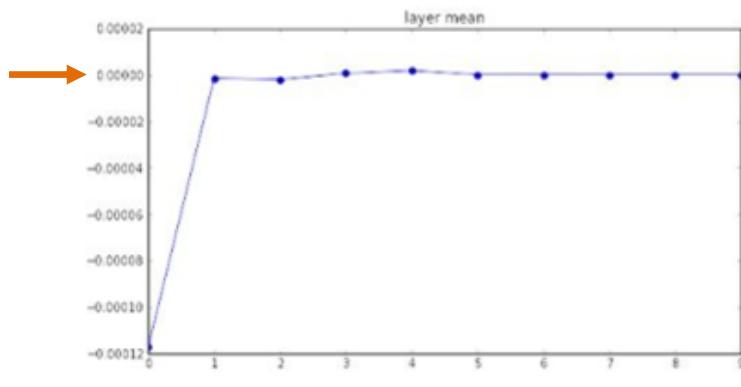


Output goes to zero

Forward



# Small Random Numbers



Small  $w_i^l$  cause small output for layer  $l$ :

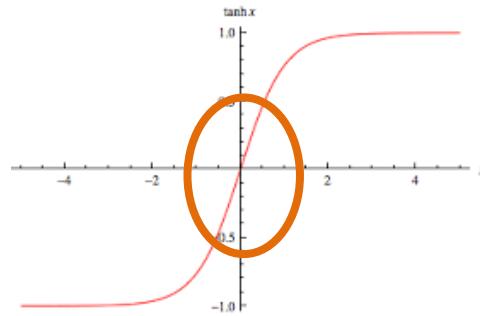
$$f_l \left( \sum_i w_i^l x_i^l + b^l \right) \approx 0$$

Forward



# Small Random Numbers

Even activation function's gradient is ok, we still have vanishing gradient problem.



Small outputs of layer  $l$  (input of layer  $l + 1$ ) cause small gradient w.r.t to the weights of layer  $l + 1$ :

$$f_{l+1} \left( \sum_i w_i^{l+1} x_i^{l+1} + b^{l+1} \right)$$

$$\frac{\partial L}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot \frac{\partial f_{l+1}}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot x_i^{l+1} \approx 0$$

Vanishing gradient, caused by small output

Backward

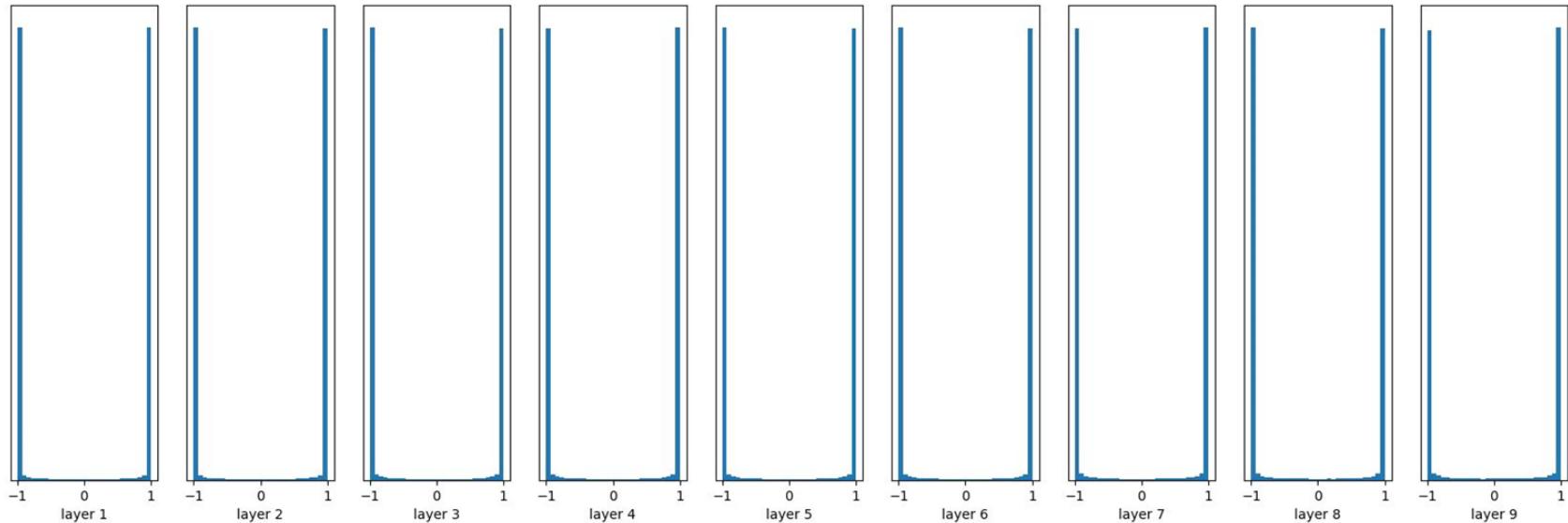


# Big Random Numbers

- Gaussian with zero mean and standard deviation 1
- Let us see what happens:
  - Network with 10 layers with 500 neurons each
  - Tanh as activation functions
  - Input unit Gaussian data

# Big Random Numbers

tanh as activation functions



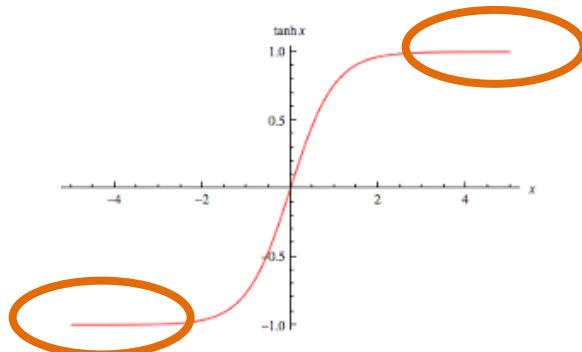
Output saturated to  
-1 and 1

# Big Random Numbers

Output saturated to -1 and 1.

Gradient of the activation function becomes close to 0.

$$f(s) = f\left(\sum_i w_i x_i + b\right)$$



$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial s} \cdot \frac{\partial s}{\partial w_i} \approx 0$$

Vanishing gradient, caused by saturated activation function.

# How to solve this?

- Working on the initialization
- Working on the output generated by each layer

# Xavier Initialization

- Gaussian with zero mean, but what standard deviation?

$$Var(s) = Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i)$$

Notice:  $n$  is the number of input neurons for the layer of weights you want to initialized. This  $n$  is not the number  $N$  of input data  $X \in R^{N \times D}$ . For the first layer  $n = D$ .

Tips:

$$E[X^2] = Var[X] + E[X]^2$$

If X, Y are independent:

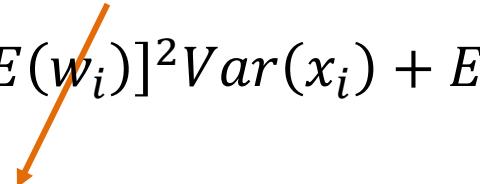
$$Var[XY] = E[X^2Y^2] - E[XY]^2$$

$$E[XY] = E[X]E[Y]$$

# Xavier Initialization

- Gaussian with zero mean, but what standard deviation?

$$\begin{aligned}Var(s) &= Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i) \\&= \sum_i^n [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i) Var(w_i)\end{aligned}$$

  
Zero mean                      Zero mean

# Xavier Initialization

- Gaussian with zero mean, but what standard deviation?

$$\begin{aligned}Var(s) &= Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i) \\&= \sum_i^n [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i) \\&= \sum_i^n Var(x_i)Var(w_i) = n(Var(w)Var(x))\end{aligned}$$

↑  
Identically distributed  
(each random variable has the same distribution)

# Xavier Initialization

- How to ensure the variance of the output is the same as the input?

Goal:

$$Var(s) = Var(x) \longrightarrow n \cdot \underbrace{Var(w)}_{= 1} Var(x) = Var(x)$$

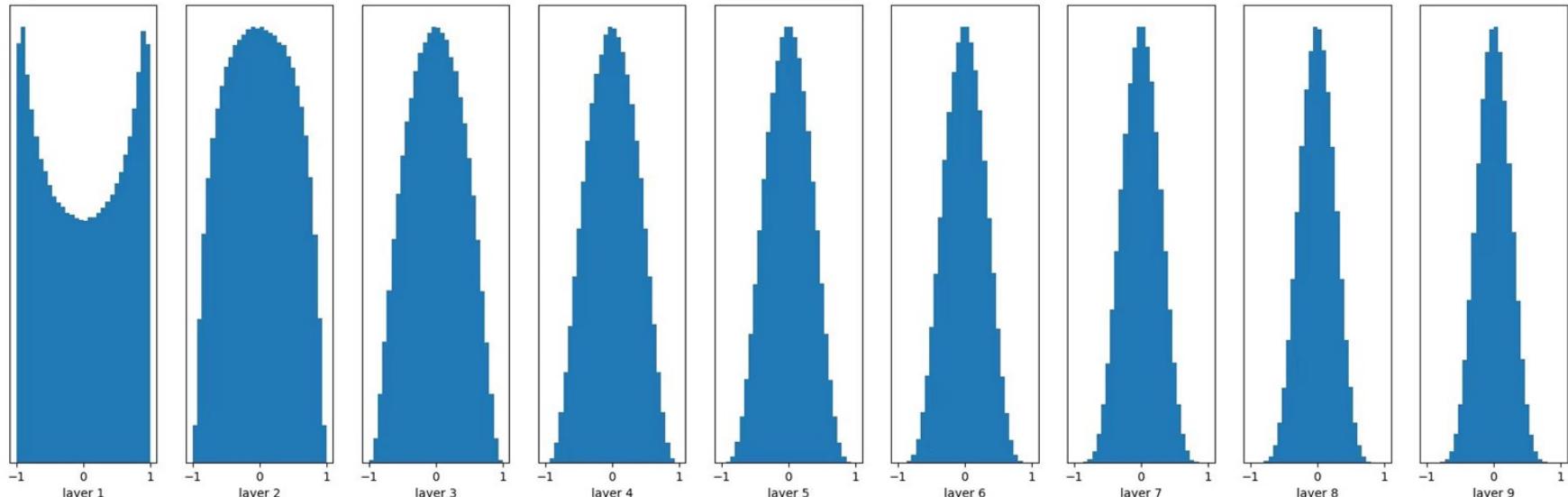
$$\longrightarrow Var(w) = \frac{1}{n}$$

$n$ : number of input neurons

# Xavier Initialization

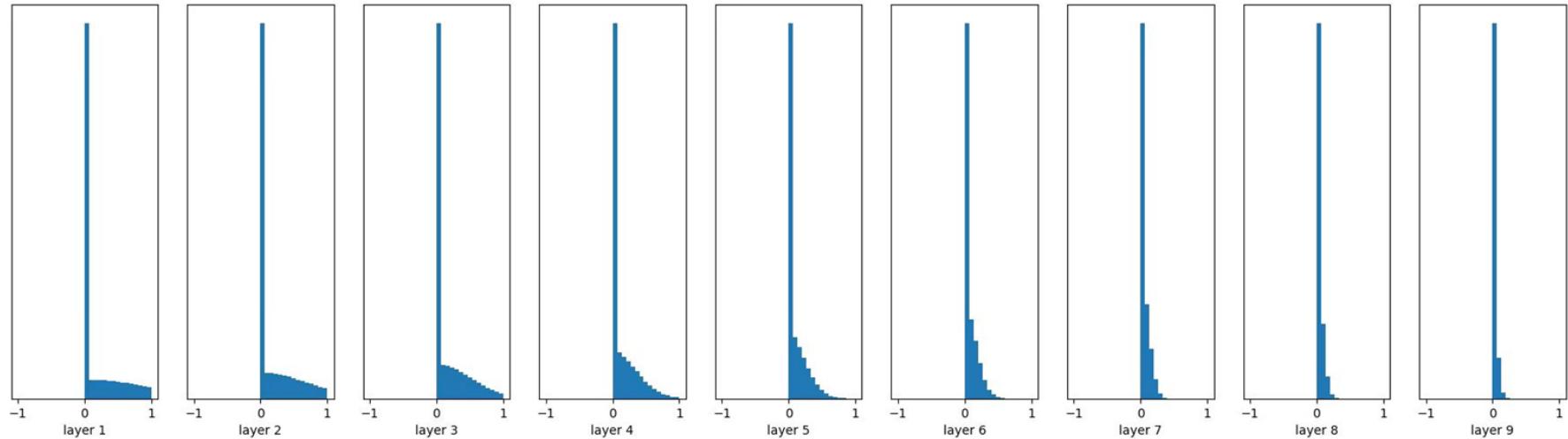
$$Var(w) = \frac{1}{n}$$

tanh as activation functions



# Xavier Initialization with ReLU (Kaiming Initialization)

$$Var(w) = \frac{1}{n}$$

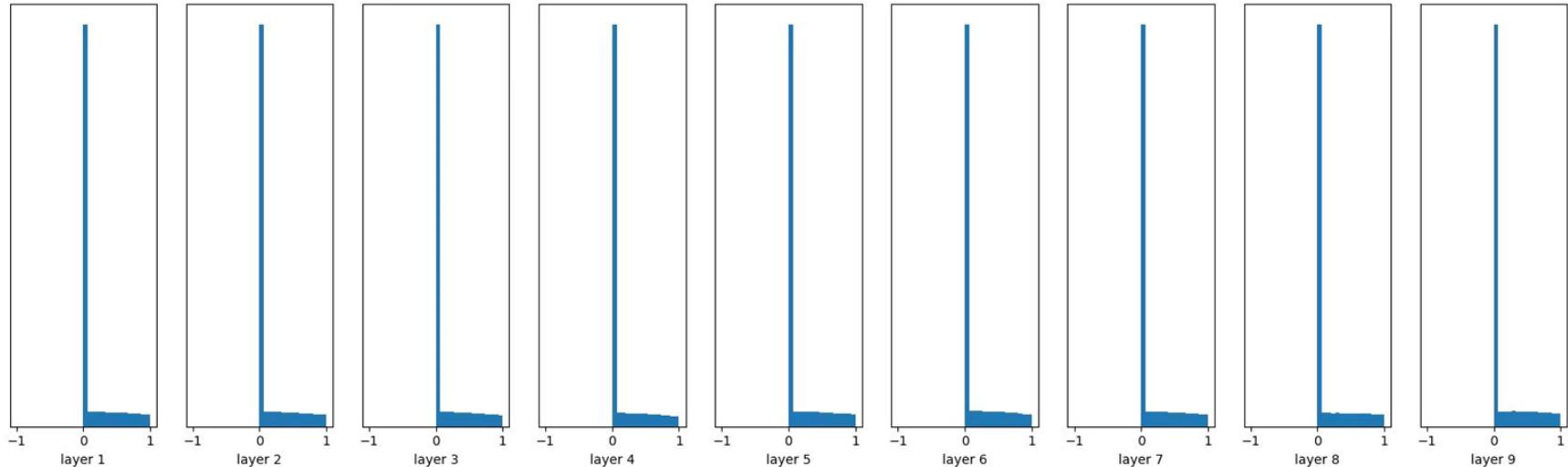


ReLU kills Half of the Data  
What's the solution?

When using ReLU, output  
close to zero again 😞

# Kaiming Initialization with ReLU

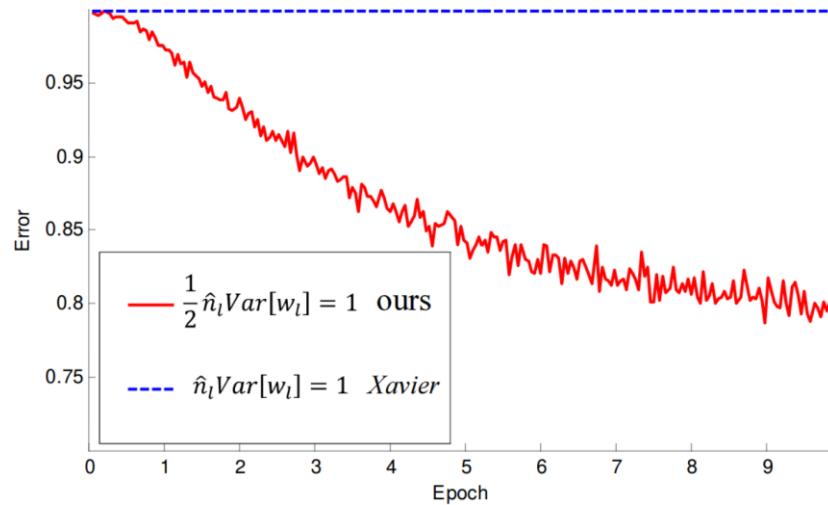
$$Var(w) = \frac{1}{n/2} = \frac{2}{n}$$



# Kaiming Initialization with ReLU

$$Var(w) = \frac{2}{n}$$

It makes a huge difference!



- Use ReLU and Xavier/2 initialization

# Summary

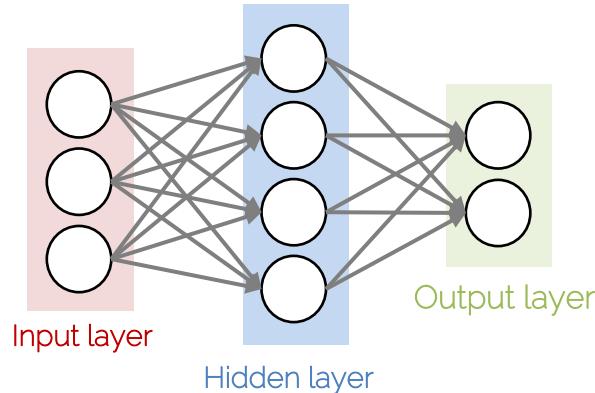


Image Classification	Output Layer	Loss function
Binary Classification	Sigmoid	Binary Cross entropy
Multiclass Classification	Softmax	Cross entropy

Other Losses:

SVM Loss (Hinge Loss), L1/L2-Loss

Initialization of optimization

- How to set weights at beginning

# Next Lecture

- Next lecture
  - More about training neural networks: regularization, dropout, data augmentation, batch normalization, etc.
  - Followed by CNNs

See you next week!

# References

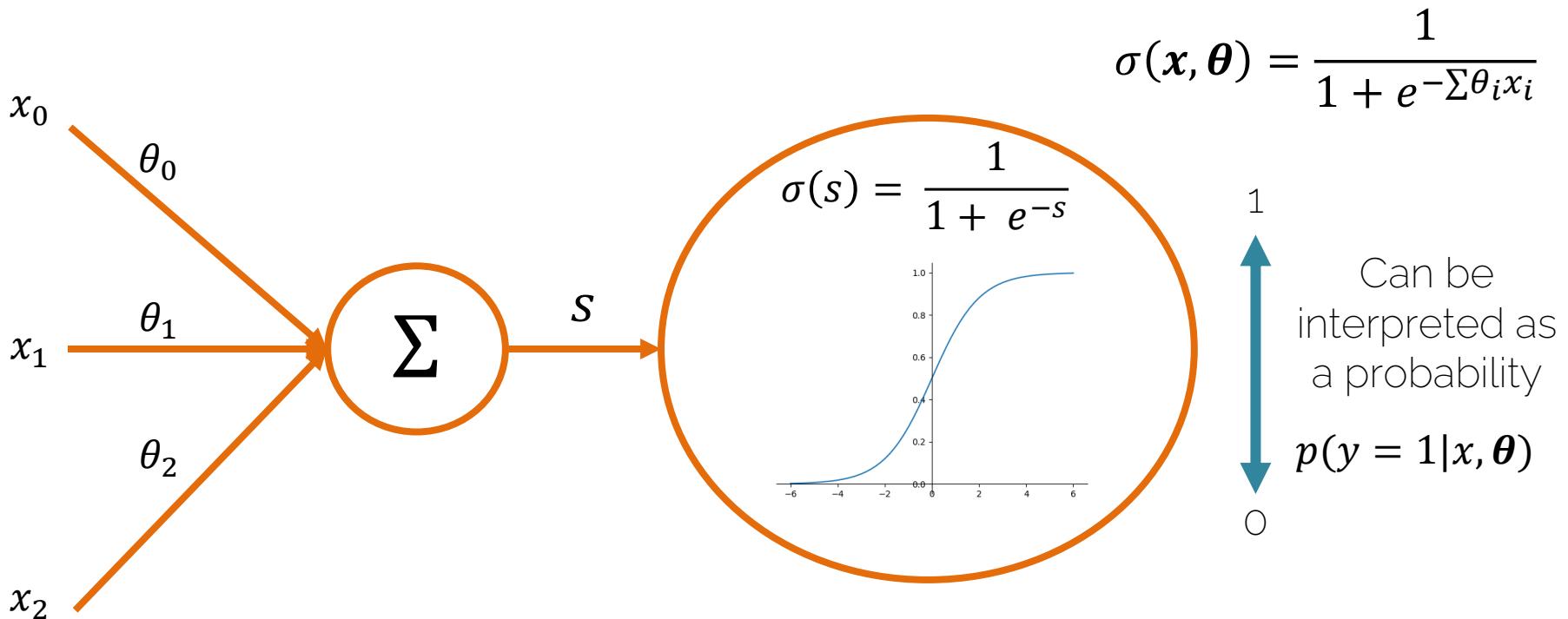
- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 6: Deep Feedforward Networks
- Bishop "Pattern Recognition and Machine Learning" (2006),
  - Chapter 5.5: Regularization in Network Nets
- <http://cs231n.github.io/neural-networks-1/>
- <http://cs231n.github.io/neural-networks-2/>
- <http://cs231n.github.io/neural-networks-3/>

# Lecture 7 Recap

# Regression Losses: L2 vs L1

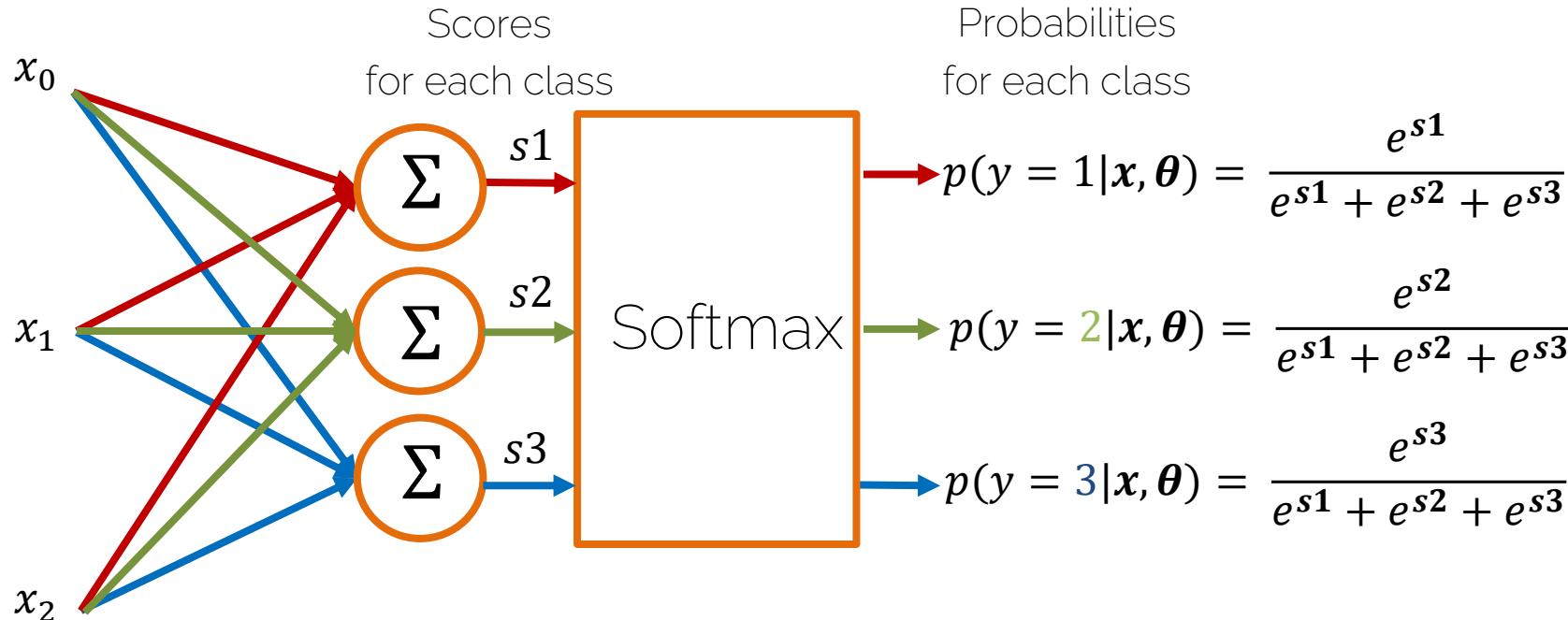
- L2 Loss:
  - $L^2 = \sum_{i=1}^n (y_i - f(x_i))^2$
  - Sum of squared differences (SSD)
  - Prone to outliers
  - Compute-efficient (optimization)
  - Optimum is the mean
- L1 Loss:
  - $L^1 = \sum_{i=1}^n |y_i - f(x_i)|$
  - Sum of absolute differences
  - Robust
  - Costly to compute
  - Optimum is the median

# Binary Classification: Sigmoid



# Softmax Formulation

- What if we have multiple classes?



# Example: Hinge vs Cross-Entropy

Hinge Loss:  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$

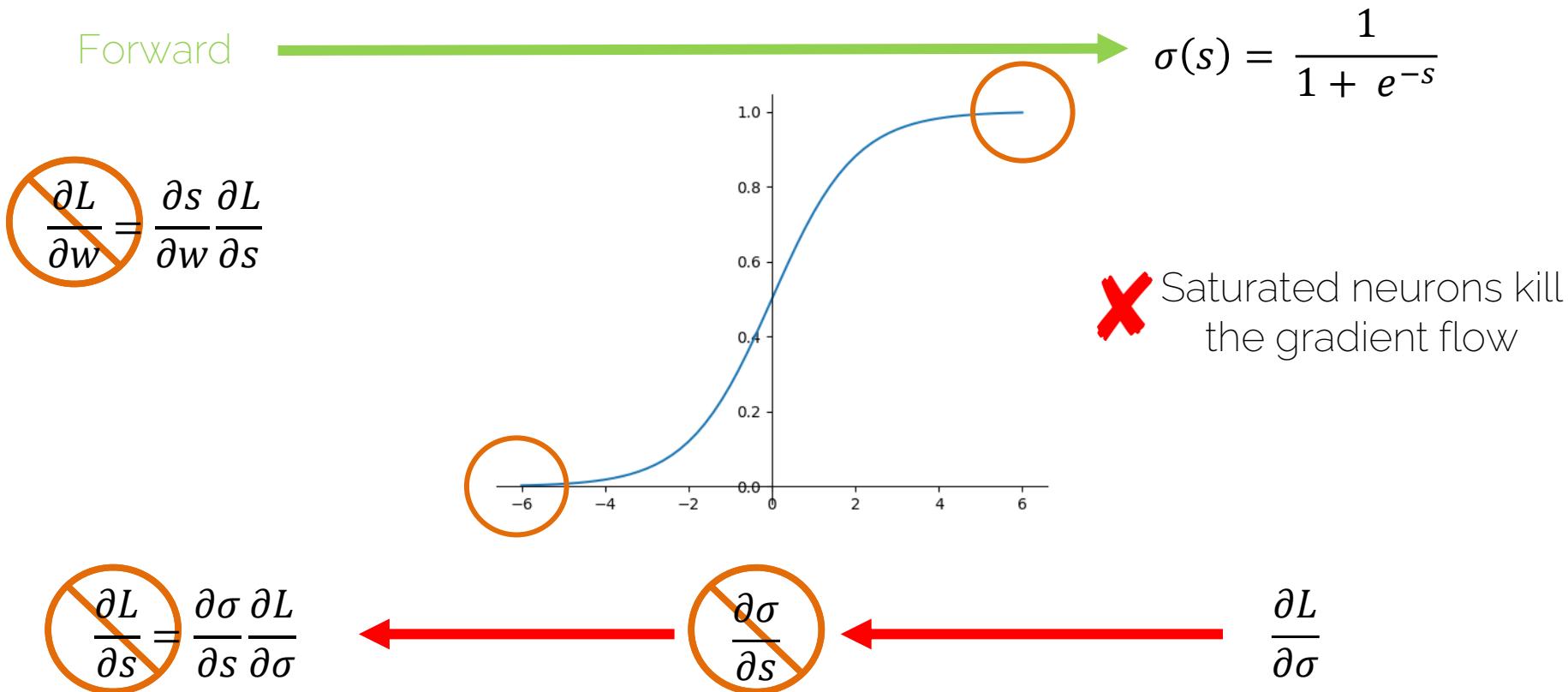
Cross Entropy :  $L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$

Given the following scores for  $\mathbf{x}_i$  : Hinge loss: Cross Entropy loss:

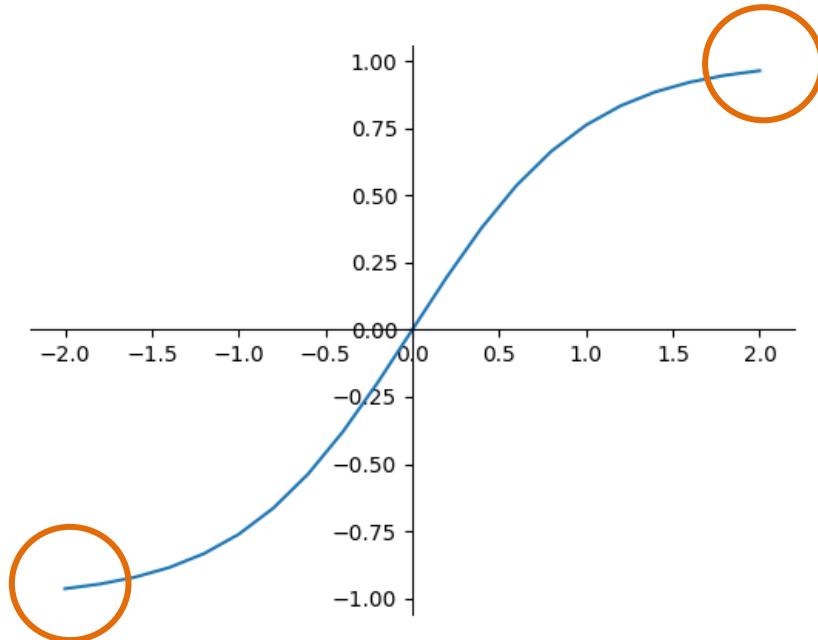
Model 1	$s = [5, -3, 2]$	$\max(0, -3 - 5 + 1) + \max(0, 2 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^{-3} + e^2}\right) = 0.05$
Model 2	$s = [5, 10, 10]$	$\max(0, 10 - 5 + 1) + \max(0, 10 - 5 + 1) = 12$	$-\ln\left(\frac{e^5}{e^5 + e^{10} + e^{10}}\right) = 5.70$
Model 3	$s = [5, -20, -20]$ $y_i = 0$	$\max(0, -20 - 5 + 1) + \max(0, -20 - 5 + 1) = 0$	$-\ln\left(\frac{e^5}{e^5 + e^{-20} + e^{-20}}\right) = 2 * 10^{-11}$

- Cross Entropy \*always\* wants to improve! (loss never 0)
- Hinge Loss saturates.

# Sigmoid Activation



# TanH Activation



✗ Still saturates

✓ Zero-centered

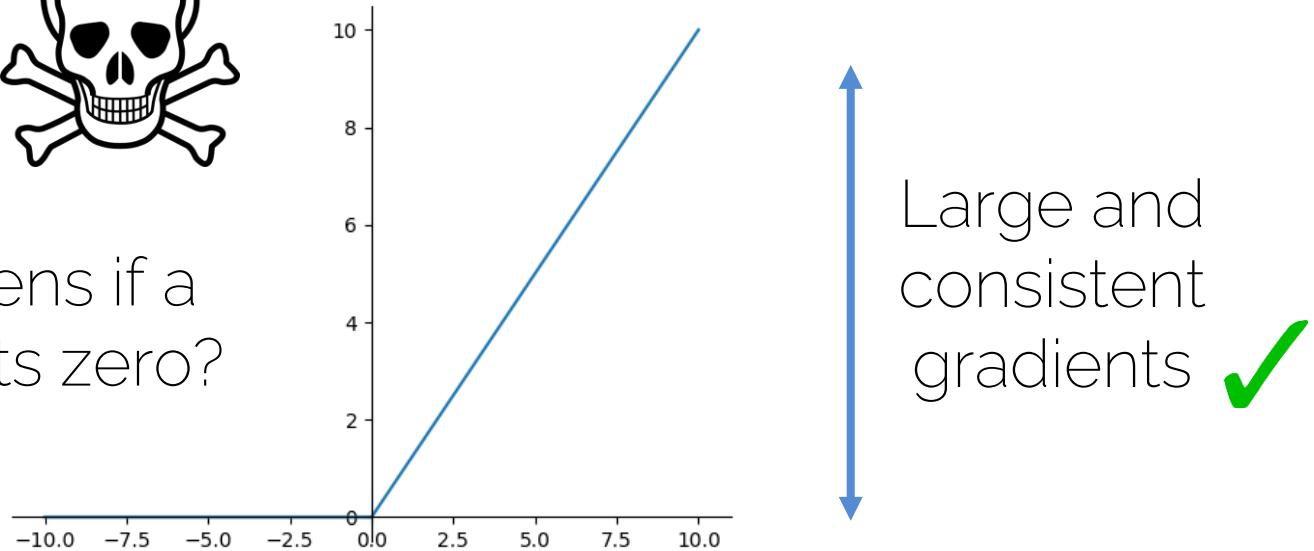
# Rectified Linear Units (ReLU)



Dead ReLU



What happens if a  
ReLU outputs zero?



Fast convergence



Does not saturate

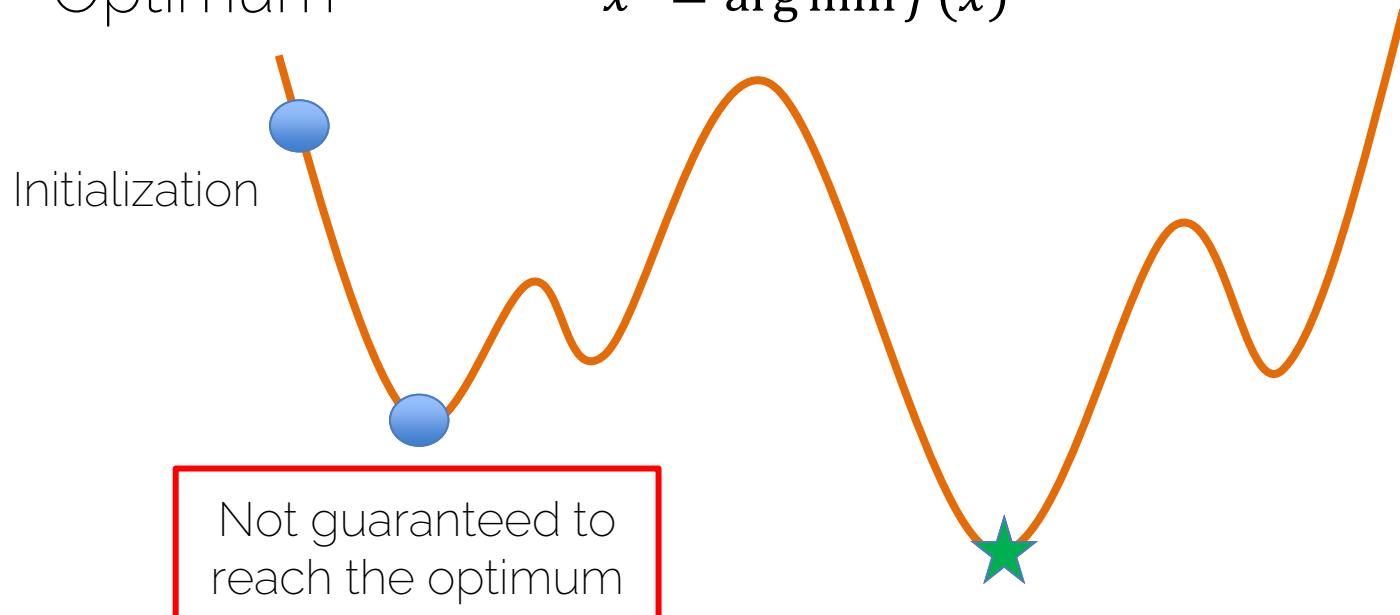
# Quick Guide

- Sigmoid is not really used.
- ReLU is the standard choice.
- Second choice are the variants of ReLU or Maxout.
- Recurrent nets will require TanH or similar.

# Initialization is Extremely Important!

- Optimum

$$x^* = \arg \min f(x)$$



# Xavier/Kaiming Initialization

- How to ensure the variance of the output is the same as the input?

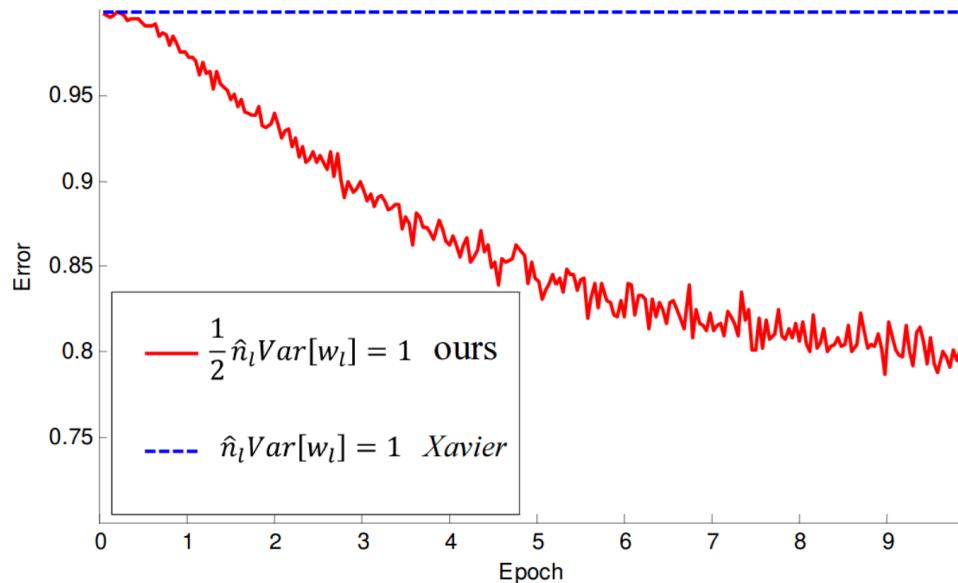
$$\underbrace{(nVar(w)Var(x))}_{= 1}$$

$$Var(w) = \frac{1}{n}$$

# ReLU Kills Half of the Data

$$Var(w) = \frac{2}{n}$$

It makes a huge difference!

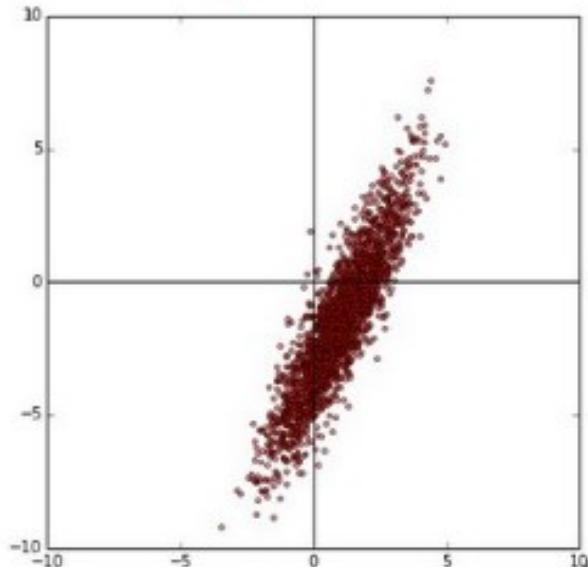


# Lecture 8

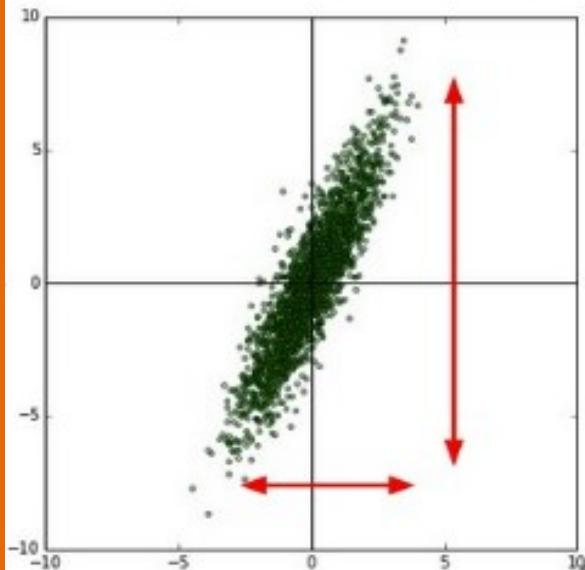
# Data Augmentation

# Data Pre-Processing

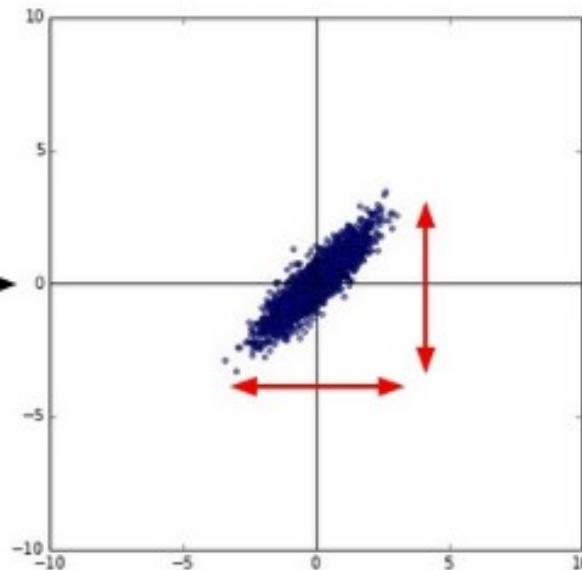
original data



zero-centered data



normalized data



For images subtract the mean image (AlexNet) or per-channel mean (VGG-Net)

# Data Augmentation

- A classifier has to be invariant to a wide variety of transformations



All

Images

Videos

News

Shopping

More

Settings

Tools

SafeSearch ▾



Cute



And Kittens



Clipart



Drawing



Cute Baby



White Cats And Kittens



Pose

Appearance

Illumination

# Data Augmentation

- A classifier has to be invariant to a wide variety of transformations
- Helping the classifier: synthesize data simulating plausible transformations

# Data Augmentation

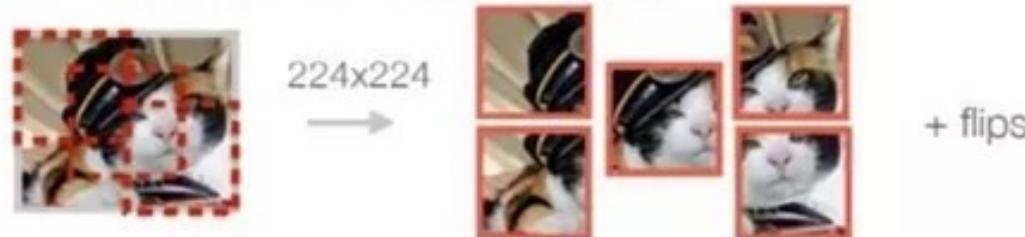
a. No augmentation (= 1 image)



b. Flip augmentation (= 2 images)

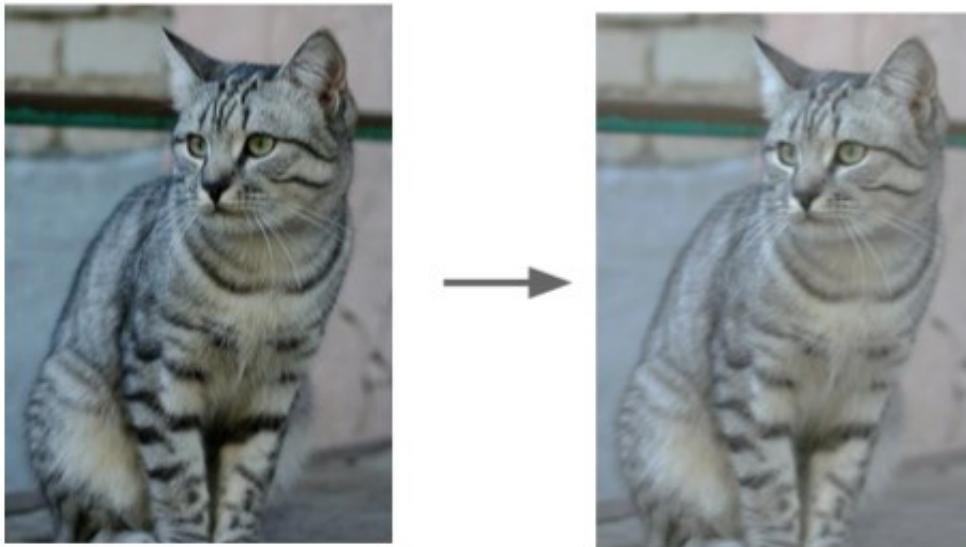


c. Crop+Flip augmentation (= 10 images)



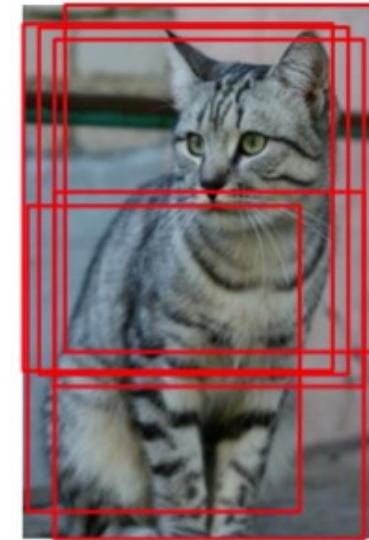
# Data Augmentation: Brightness

- Random brightness and contrast changes

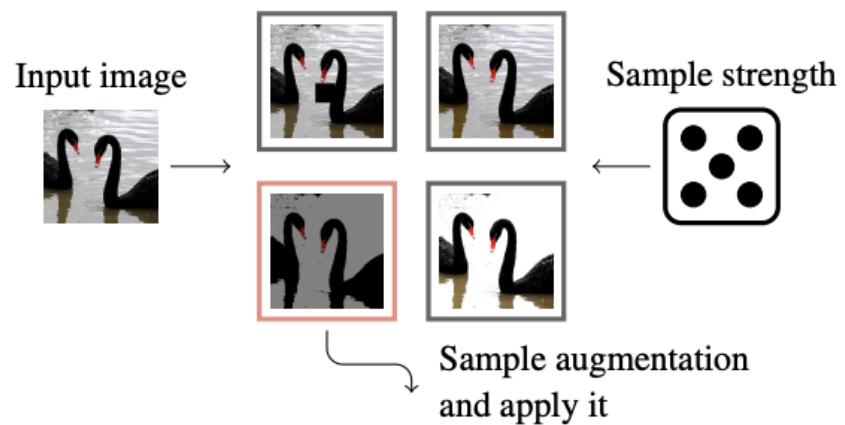
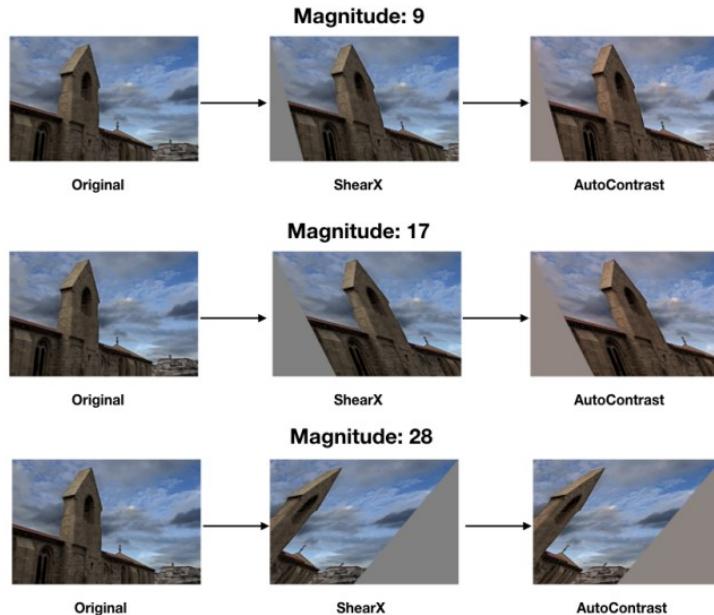


# Data Augmentation: Random Crops

- Training: random crops
  - Pick a random  $L$  in  $[256, 480]$
  - Resize training image, short side  $L$
  - Randomly sample crops of  $224 \times 224$
- Testing: fixed set of crops
  - Resize image at  $N$  scales
  - 10 fixed crops of  $224 \times 224$ : (4 corners + 1 center)  $\times$  2 flips



# Data Augmentation: Advanced



Cubuk et al., RandAugment, CVPRW 2020

Muller et al., Trivial Augment, ICCV 2021

# Data Augmentation

- When comparing two networks make sure to use the same data augmentation!
- Consider data augmentation a part of your network design

# Advanced Regularization

# L2 regularization, also (wrongly) called weight decay

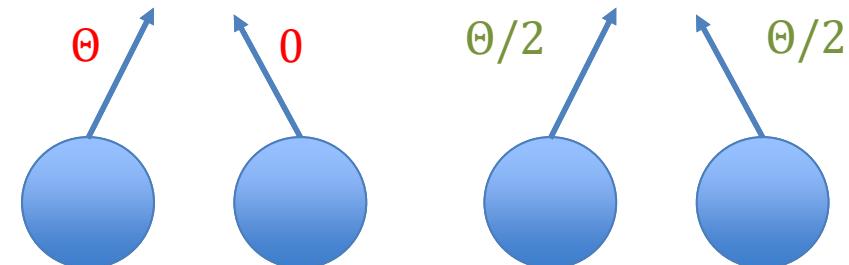
- L2 regularization

$$\Theta_{k+1} = \Theta_k - \epsilon \nabla_{\Theta}(\Theta_k, x, y) - \lambda \Theta_k$$

Learning rate      Gradient      Gradient of L2-regularization

The diagram illustrates the L2 regularization update rule. It shows the formula  $\Theta_{k+1} = \Theta_k - \epsilon \nabla_{\Theta}(\Theta_k, x, y) - \lambda \Theta_k$  with three arrows pointing to its terms. The first arrow points to  $\epsilon \nabla_{\Theta}(\Theta_k, x, y)$  and is labeled "Learning rate". The second arrow points to  $\lambda \Theta_k$  and is labeled "Gradient". The third arrow points to  $\Theta_k$  and is labeled "Gradient of L2-regularization".

- Penalizes large weights
- Improves generalization



# L2 regularization, also (wrongly) called weight decay

- Weight decay regularization

$$\Theta_{k+1} = (1 - \lambda)\Theta_k - \alpha \nabla f_k(\Theta_k)$$

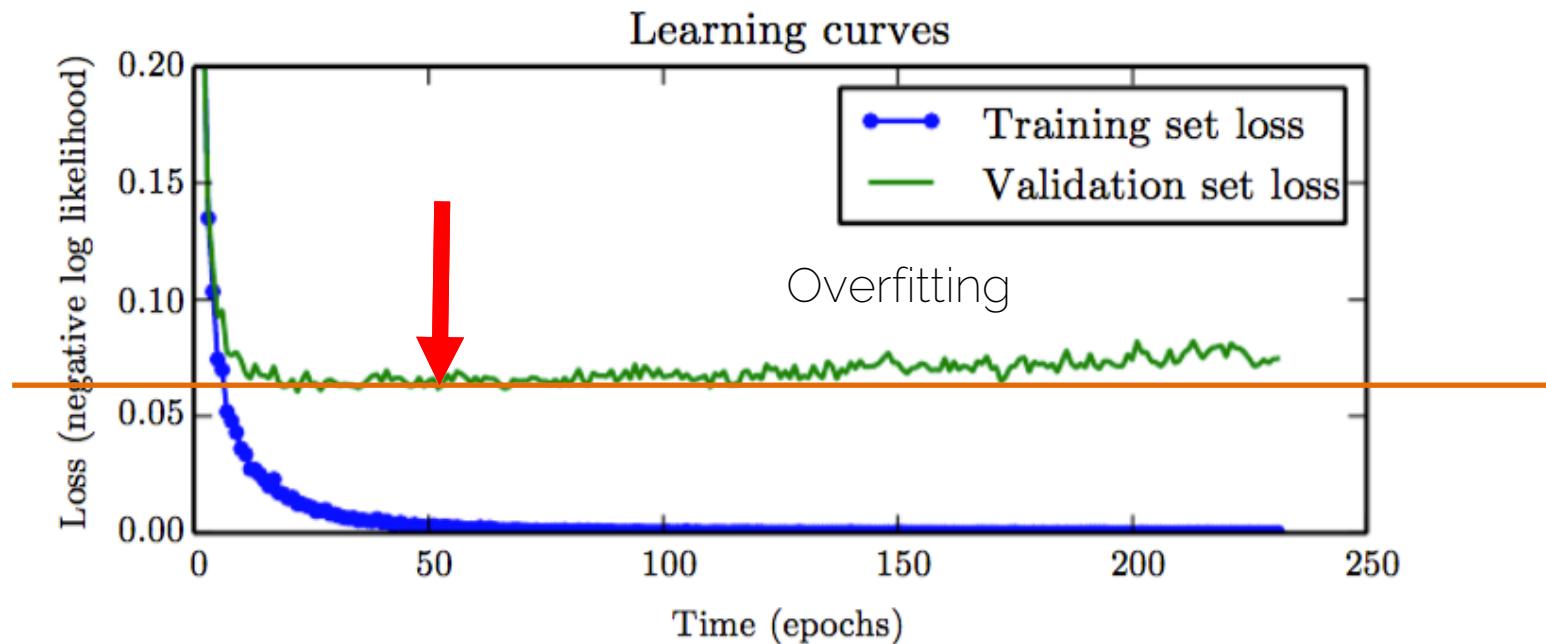
Learning rate of weight decay

Learning rate of the optimizer

- Equivalent to L2 regularization in GD, but not in Adam.

Loshchilov and Hutter, Decoupled Weight Decay Regularization, ICLR 2019

# Early Stopping

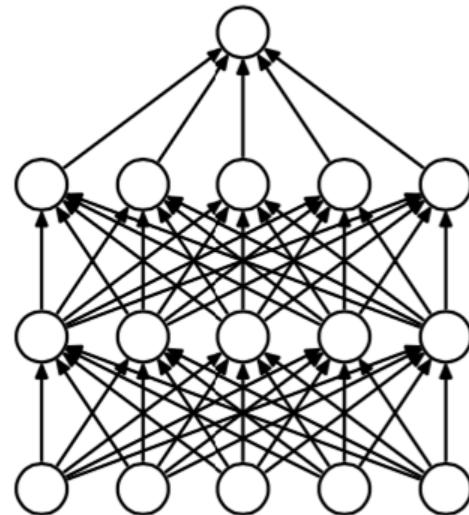


# Bagging and Ensemble Methods

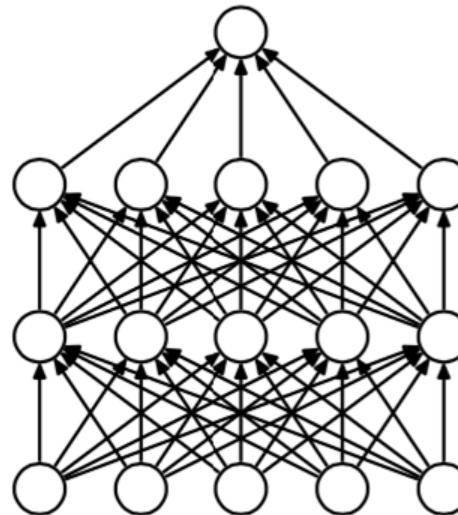
- Train multiple models and average their results
- E.g., use a different algorithm for optimization or change the objective function / loss function.
- If errors are uncorrelated, the expected combined error will decrease linearly with the ensemble size

# Bagging and Ensemble Methods

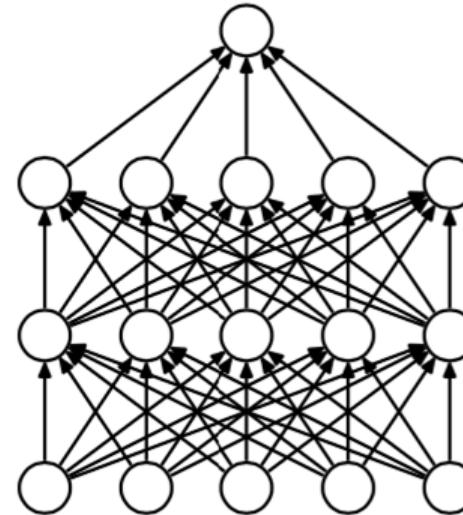
- Bagging: uses  $k$  different datasets (or SGD/init noise)



Training Set 1



Training Set 2

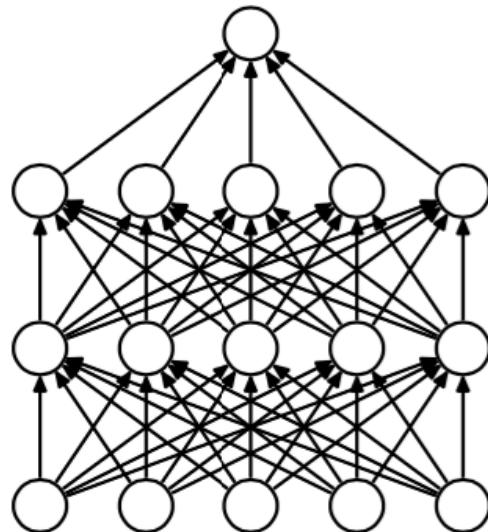


Training Set 3

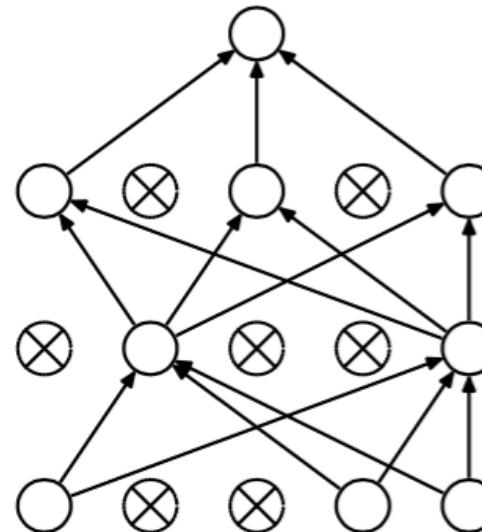
# Dropout

# Dropout

- Disable a random set of neurons (typically 50%)



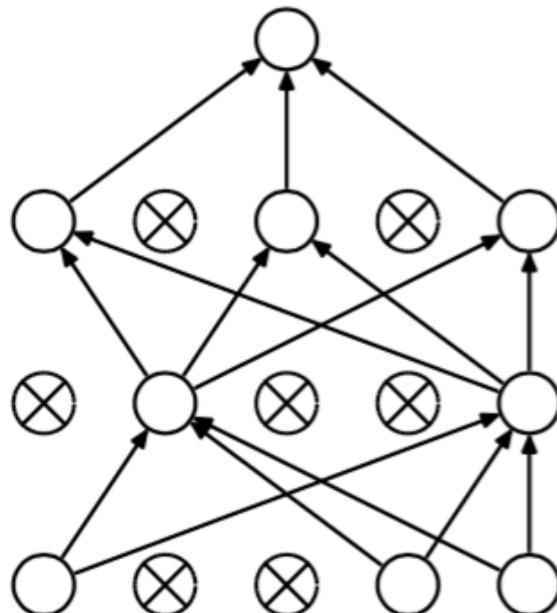
(a) Standard Neural Net



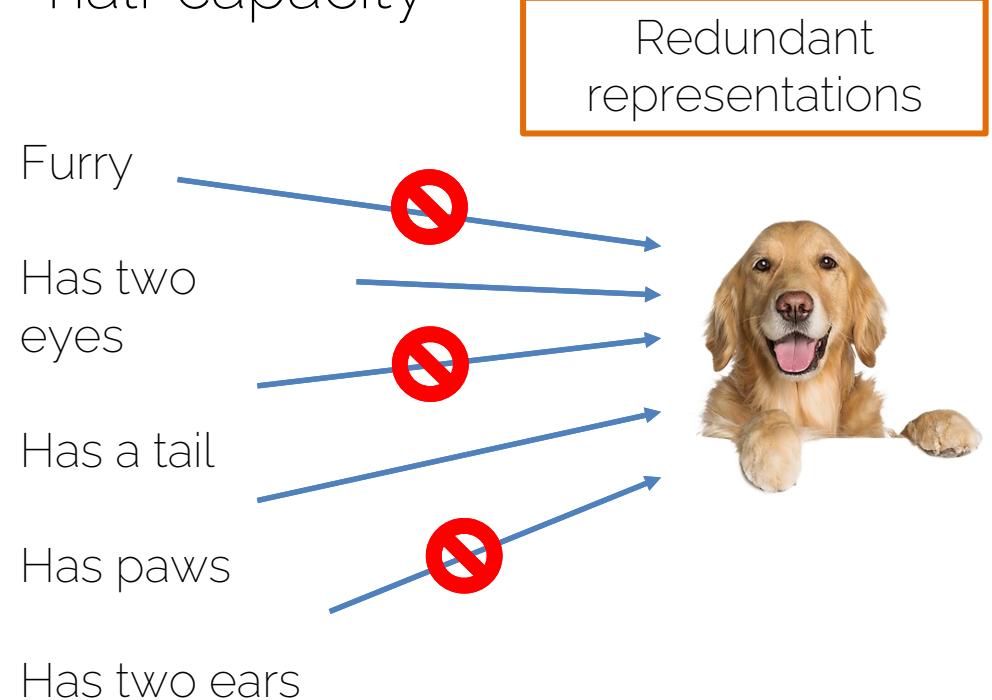
(b) After applying dropout.

# Dropout: Intuition

- Using half the network = half capacity



(b) After applying dropout.

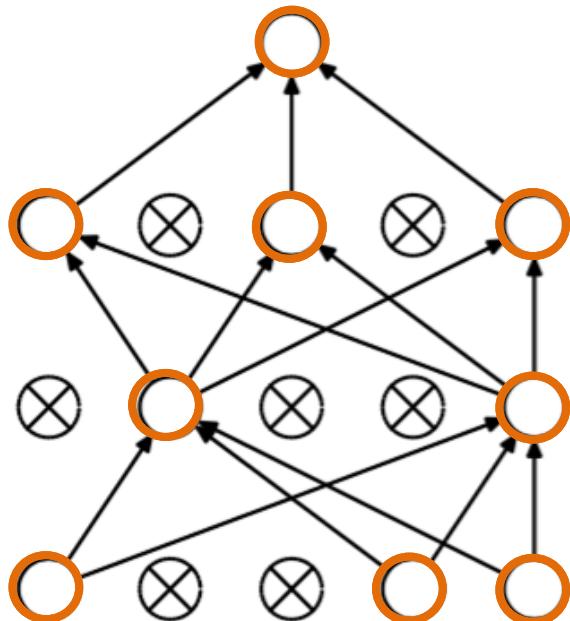


# Dropout: Intuition

- Using half the network = half capacity
  - Redundant representations
  - Base your scores on more features
- Consider it as a model ensemble

# Dropout: Intuition

- Two models in one

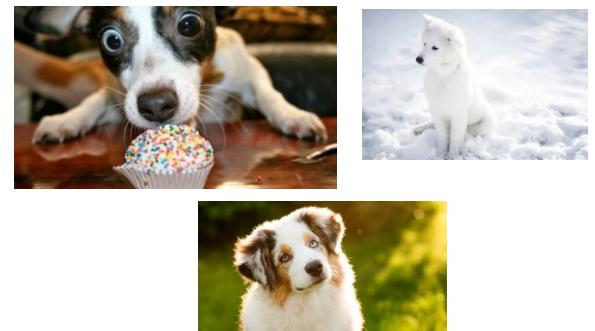


(b) After applying dropout.

○ Model 1



⊗ Model 2



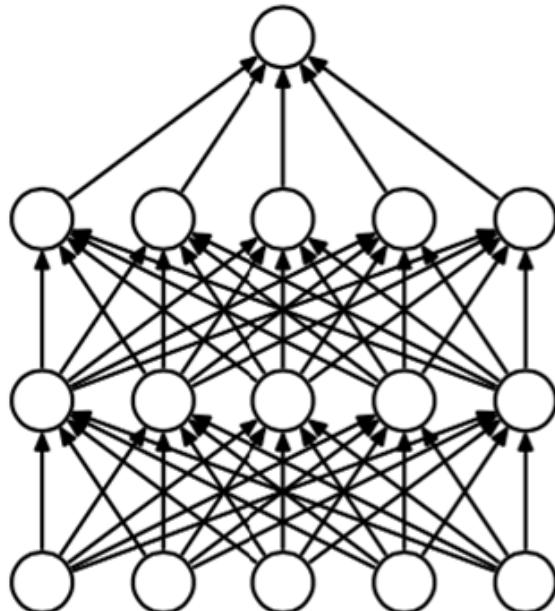
# Dropout: Intuition

- Using half the network = half capacity
  - Redundant representations
  - Base your scores on more features
- Consider it as two models in one
  - Training a large ensemble of models, each on different set of data (mini-batch) and with SHARED parameters

Reducing co-adaptation between neurons

# Dropout: Test Time

- All neurons are “turned on” – no dropout



Conditions at train and test time are not the same

PyTorch: `model.train()` and `model.eval()`

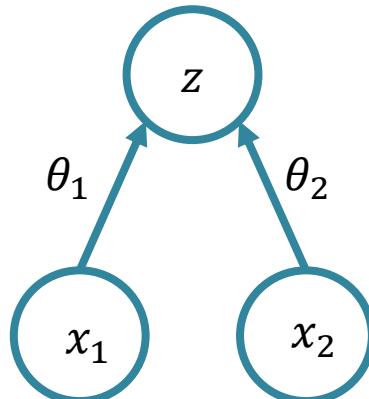
# Dropout: Test Time

- Test:

$$z = (\theta_1 x_1 + \theta_2 x_2) \cdot p$$

Dropout probability  
 $p = 0.5$

- Train:



Weight scaling  
inference rule

$$\begin{aligned} E[z] &= \frac{1}{4}(\theta_1 0 + \theta_2 0 \\ &\quad + \theta_1 x_1 + \theta_2 0 \\ &\quad + \theta_1 0 + \theta_2 x_2 \\ &\quad + \theta_1 x_1 + \theta_2 x_2) \\ &= \frac{1}{2}(\theta_1 x_1 + \theta_2 x_2) \end{aligned}$$

# Dropout: Before

- Efficient bagging method with parameter sharing
- Try it!
- Dropout reduces the effective capacity of a model, but needs more training time
- Efficient regularization method, can be used with L2

# Dropout: Nowadays

- Usually does not work well when combined with batch-norm.
- Training takes a bit longer, usually 1.5x
- But, can be used for uncertainty estimation.
- Monte Carlo dropout (Yarin Gal and Zoubin Ghahramani series of papers).

# Monte Carlo Dropout

- Neural networks are massively overconfident.
- We can use dropout to make the softmax probabilities more calibrated.
- Training: use dropout with a low  $p$  (0.1 or 0.2).
- Inference, run the same image multiple times (25-100), and average the results.

Gal et al., Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference, ICLRW 2015

Gal and Ghahramani, Dropout as a Bayesian approximation, ICML 2016

Gal et al., Deep Bayesian Active Learning with Image Data, ICML 2017

Gal, Uncertainty in Deep Learning, PhD thesis 2017

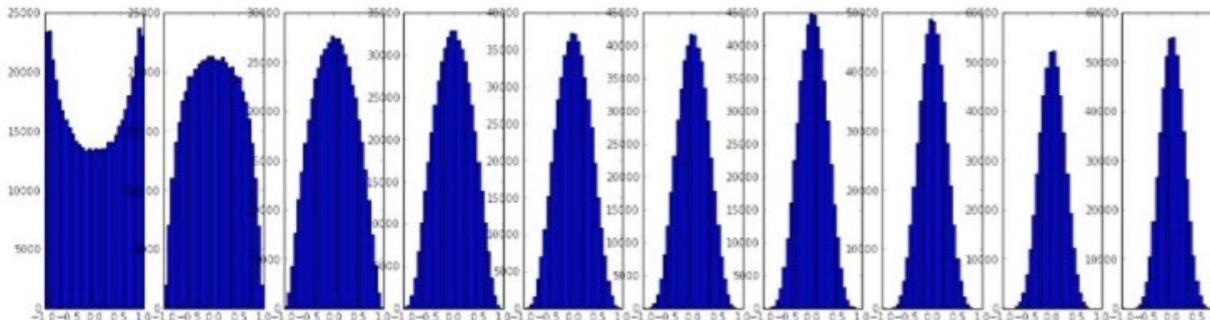
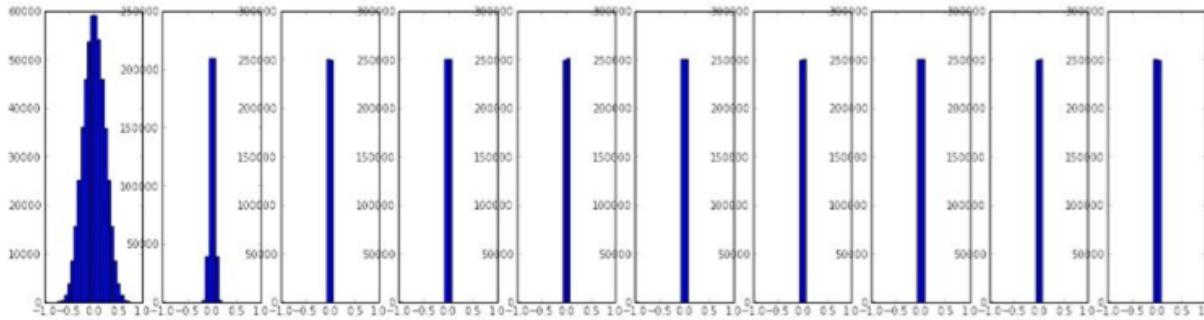
# Batch Normalization: Reducing Internal Covariate Shift

# Batch Normalization: Reducing Internal Covariate Shift

What is internal covariate shift, by the way?

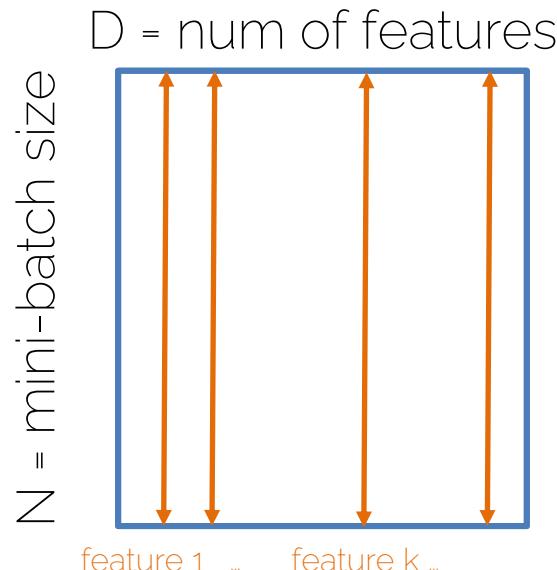
# Our Goal

- All we want is that our activations do not die out



# Batch Normalization

- Wish: Unit Gaussian activations (in our example)
- Solution: let's do it

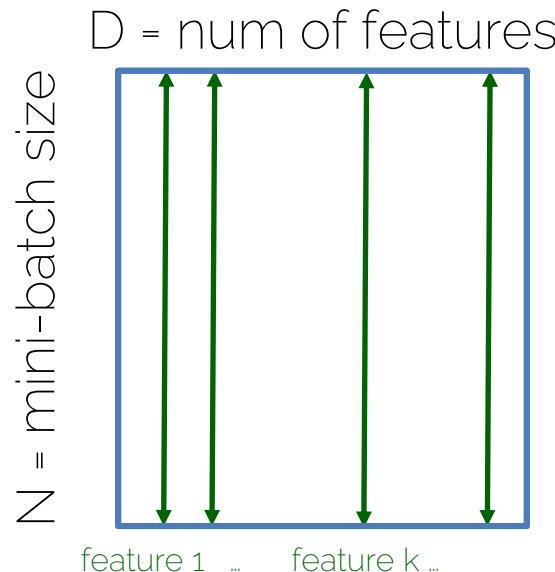


Mean of your mini-batch examples over feature k

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

# Batch Normalization

- In each dimension of the features, you have a unit gaussian (in our example)



Mean of your mini-batch examples over feature k

$$\hat{x}^{(k)} = \frac{\mathbf{x}^{(k)} - E[\mathbf{x}^{(k)}]}{\sqrt{Var[\mathbf{x}^{(k)}]}}$$

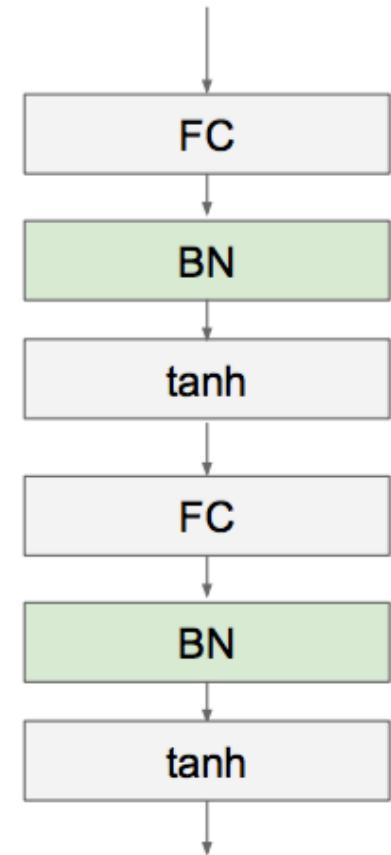
Unit gaussian

# Batch Normalization

- In each dimension of the features, you have a unit gaussian (in our example)
- For NN in general, BN normalizes the mean and variance of the inputs to your activation functions

# BN Layer

- A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions



# Batch Normalization

- 1. Normalize

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - E[\mathbf{x}^{(k)}]}{\sqrt{Var[\mathbf{x}^{(k)}]}}$$



Differentiable function so we can backprop through it....

- 2. Allow the network to change the range

$$\mathbf{y}^{(k)} = \gamma^{(k)} \hat{\mathbf{x}}^{(k)} + \beta^{(k)}$$



These parameters will be optimized during backprop

# Batch Normalization

- 1. Normalize

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - E[\mathbf{x}^{(k)}]}{\sqrt{Var[\mathbf{x}^{(k)}]}}$$

- 2. Allow the network to change the range

$$\mathbf{y}^{(k)} = \gamma^{(k)} \hat{\mathbf{x}}^{(k)} + \beta^{(k)}$$

backprop

The network can learn to undo the normalization

$$\gamma^{(k)} = \sqrt{Var[\mathbf{x}^{(k)}]}$$

$$\beta^{(k)} = E[\mathbf{x}^{(k)}]$$

# Batch Normalization

- Ok to treat dimensions separately?

Shown empirically that even if features are not correlated, convergence is still faster with this method

# BN: Train vs Test

- Train time: mean and variance is taken over the mini-batch

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - E[\mathbf{x}^{(k)}]}{\sqrt{Var[\mathbf{x}^{(k)}]}}$$

- Test-time: what happens if we can just process one image at a time?
  - No chance to compute a meaningful mean and variance

# BN: Train vs Test

**Training:** Compute mean and variance from mini-batch  
1,2,3 ...

**Testing:** Compute mean and variance by running an exponentially weighted averaged across training mini-batches. Use them as  $\sigma_{test}^2$  and  $\mu_{test}$ .

$$Var_{running} = \beta_m * Var_{running} + (1 - \beta_m) * Var_{minibatch}$$

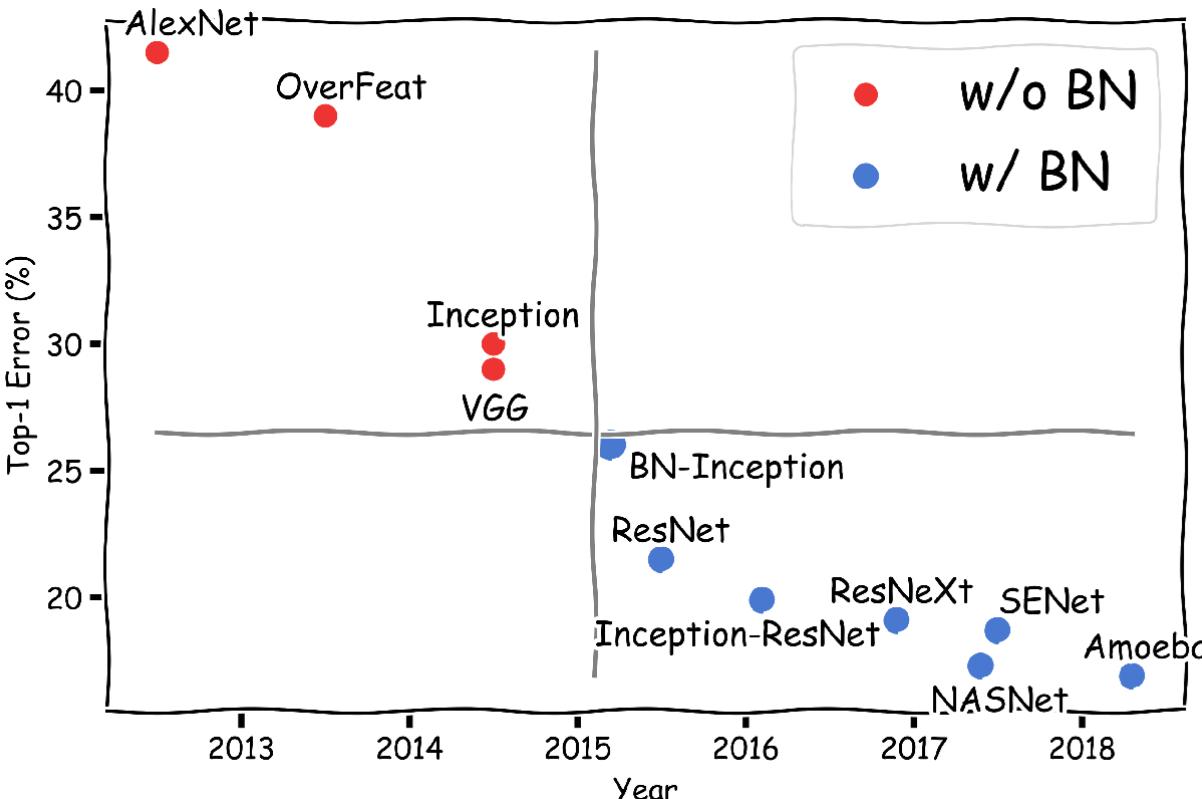
$$\mu_{running} = \beta_m * \mu_{running} + (1 - \beta_m) * \mu_{minibatch}$$

$\beta_m$ : momentum (hyperparameter)

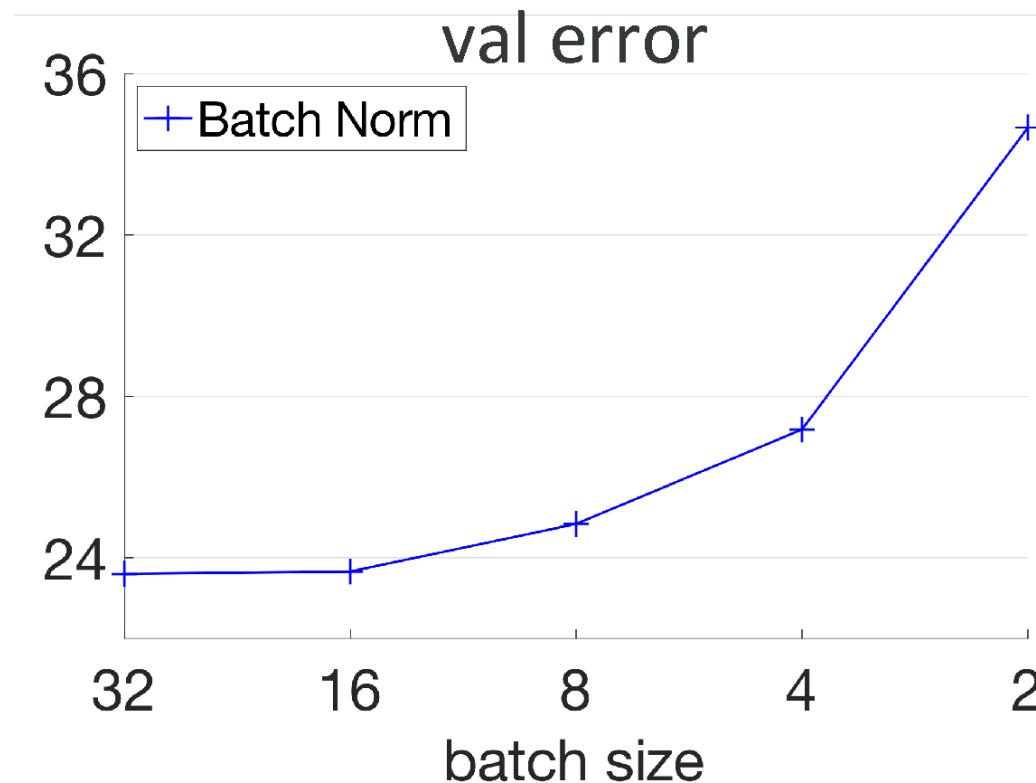
# BN: What do you get?

- Very deep nets are much easier to train, more stable gradients
- A much larger range of hyperparameters works similarly when using BN

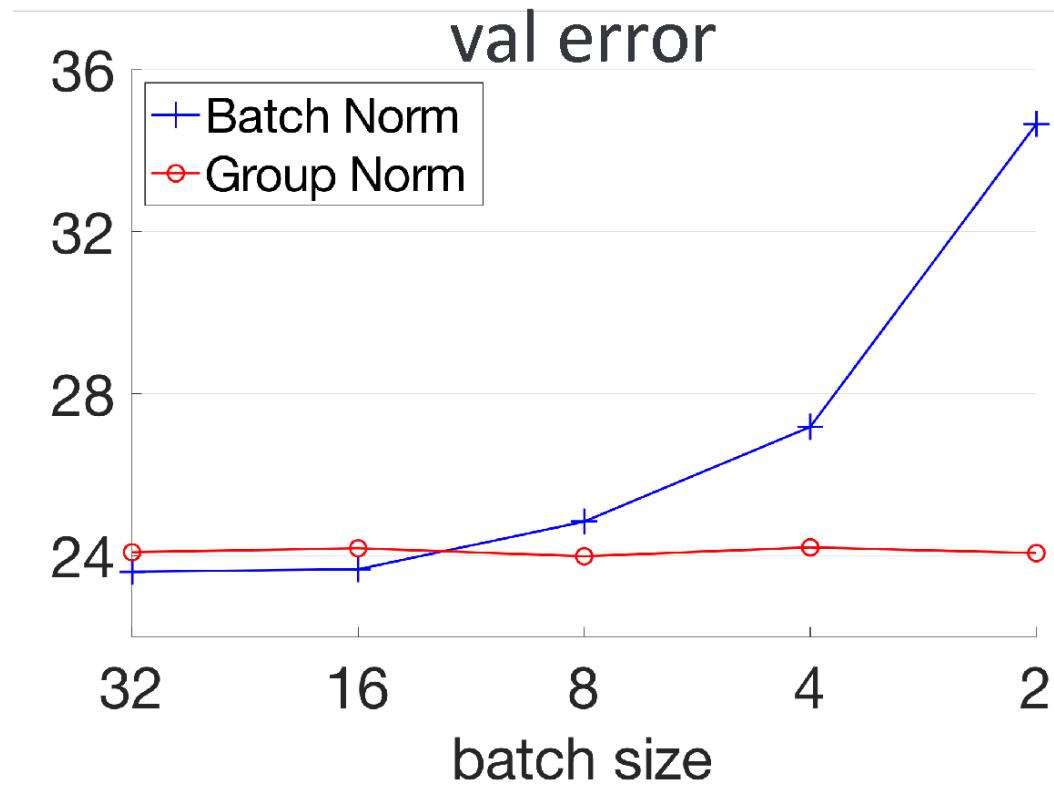
# BN: A Milestone



# BN: Drawbacks

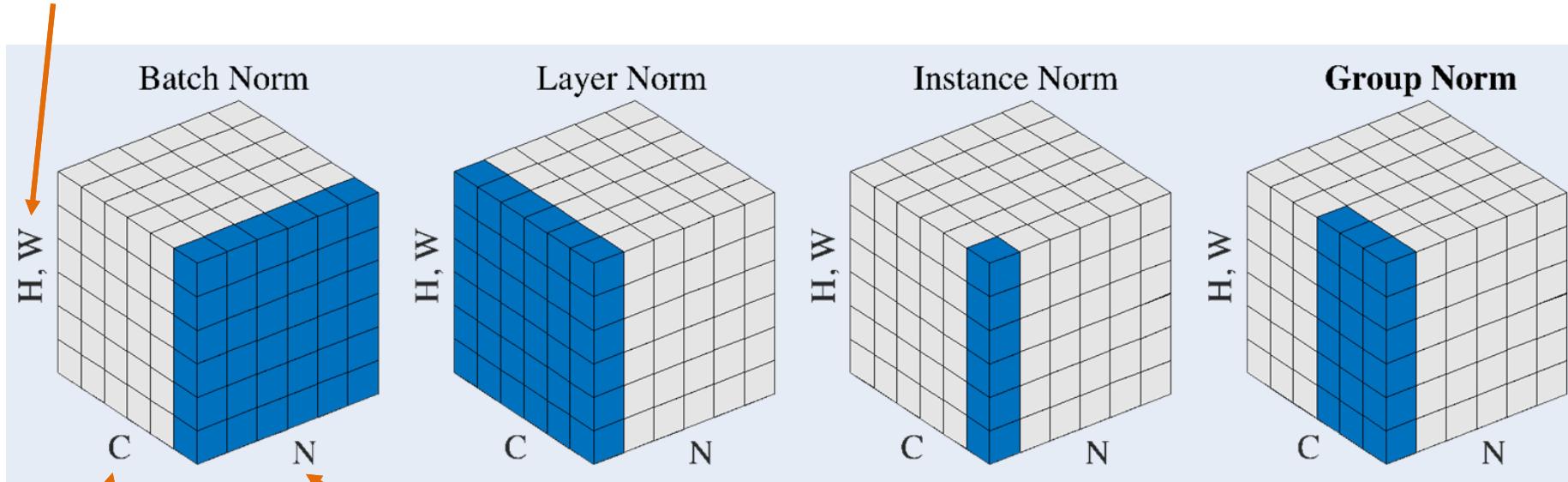


# Other Normalizations



# Other Normalizations

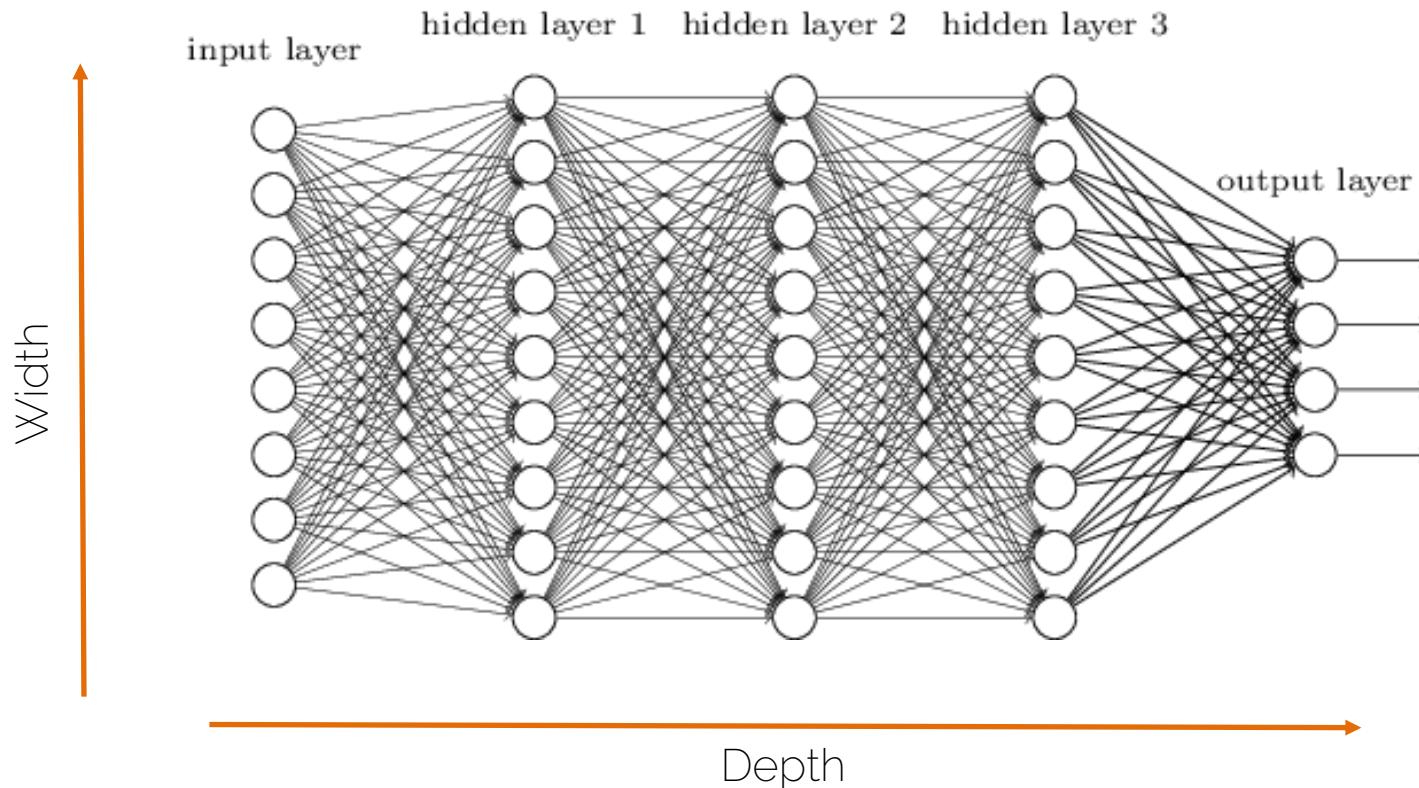
Image size



Number of channels

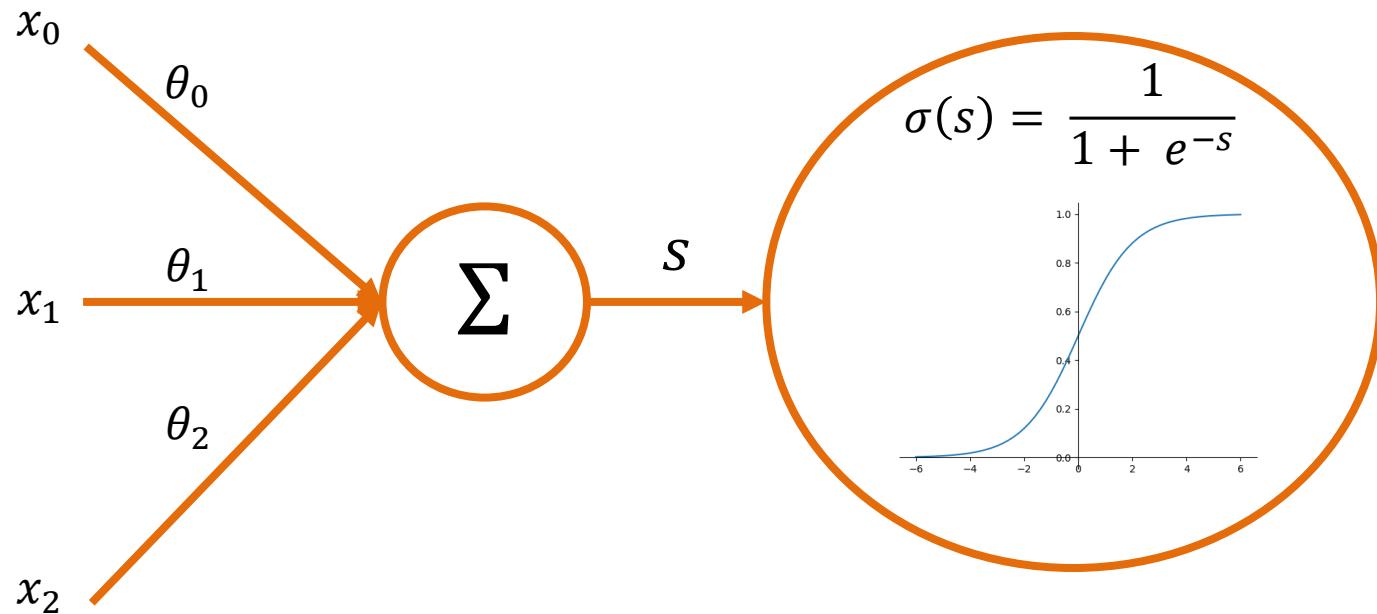
# What We Know

# What do we know so far?



# What do we know so far?

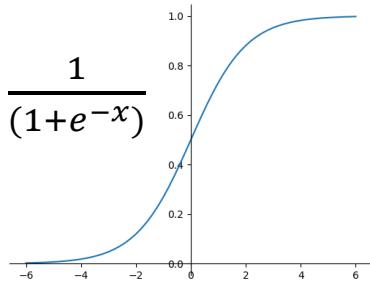
Concept of a 'Neuron'



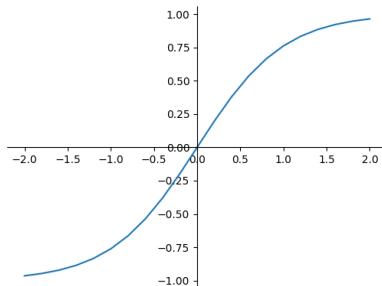
# What do we know so far?

## Activation Functions (non-linearities)

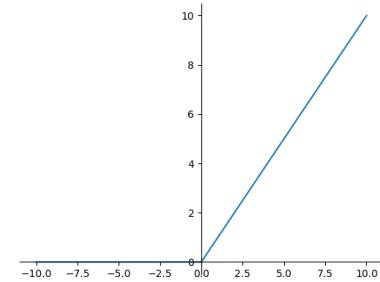
- Sigmoid:  $\sigma(x) = \frac{1}{(1+e^{-x})}$



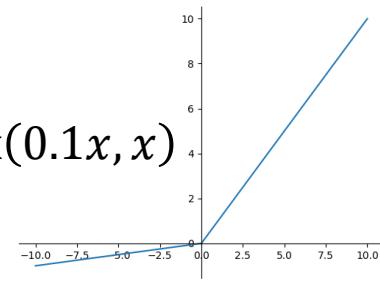
- TanH:  $\tanh(x)$



- ReLU:  $\max(0, x)$

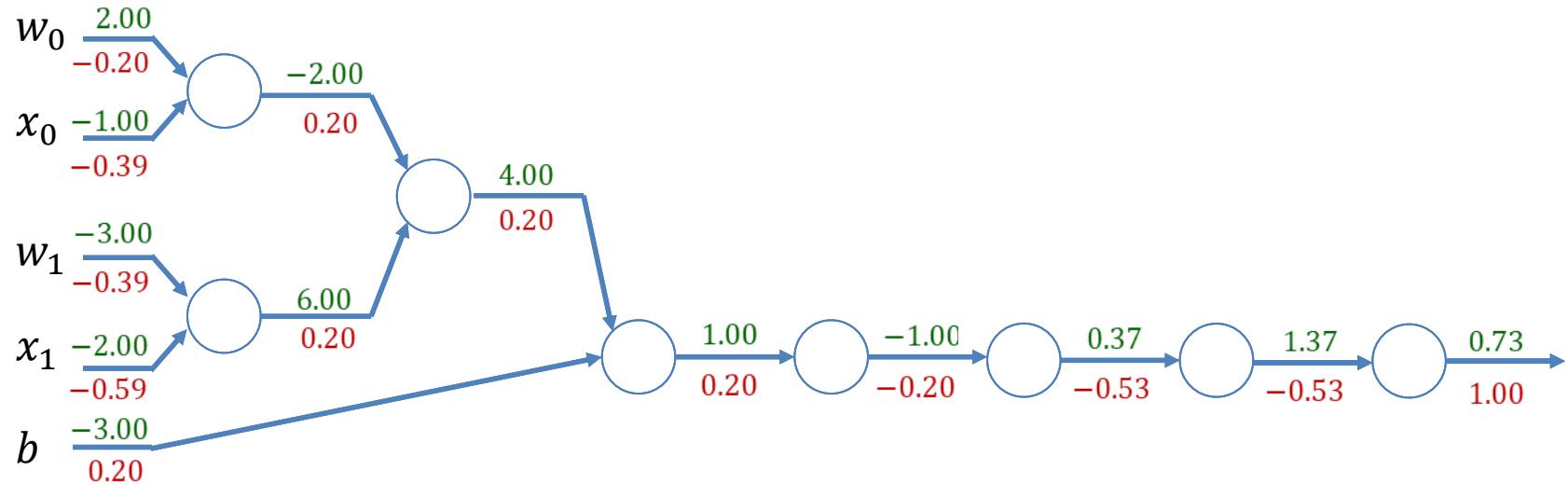


- Leaky ReLU:  $\max(0.1x, x)$



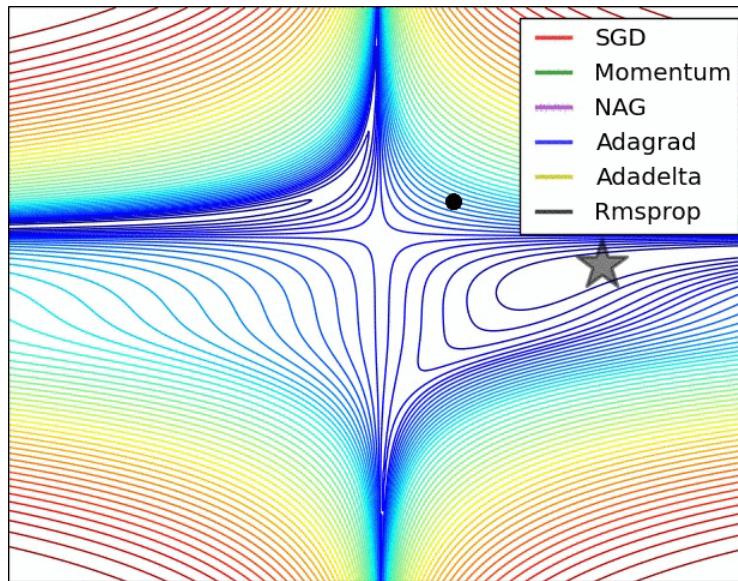
# What do we know so far?

Backpropagation



# What do we know so far?

SGD Variations (Momentum, etc.)



# What do we know so far?

## Data Augmentation

a. No augmentation (= 1 image)



b. Flip augmentation (= 2 images)



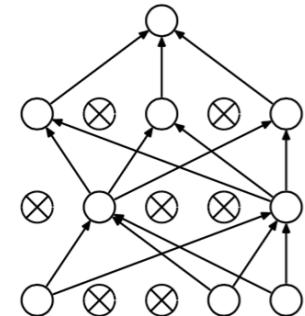
## Weight Regularization

e.g.,  $L^2$ -reg:  $R^2(\mathbf{W}) = \sum_{i=1}^N w_i^2$

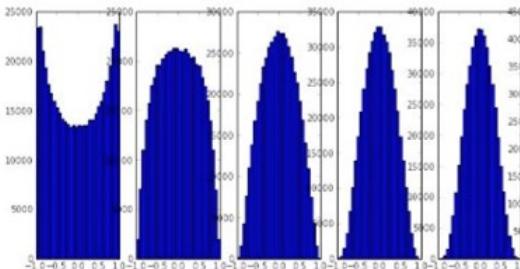
## Batch-Norm

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - E[\mathbf{x}^{(k)}]}{\sqrt{Var[\mathbf{x}^{(k)}]}}$$

## Dropout



Weight Initialization  
(e.g., Kaiming)



(b) After applying dropout.

# Why not simply more layers?

- Neural nets with at least one hidden layer are universal function approximators.
- But generalization is another issue.
- Why not just go deeper and get better?
  - No structure!!
  - It is just brute force!
  - Optimization becomes hard
  - Performance plateaus / drops!
- We need more! More means CNNs, RNNs and eventually Transformers.

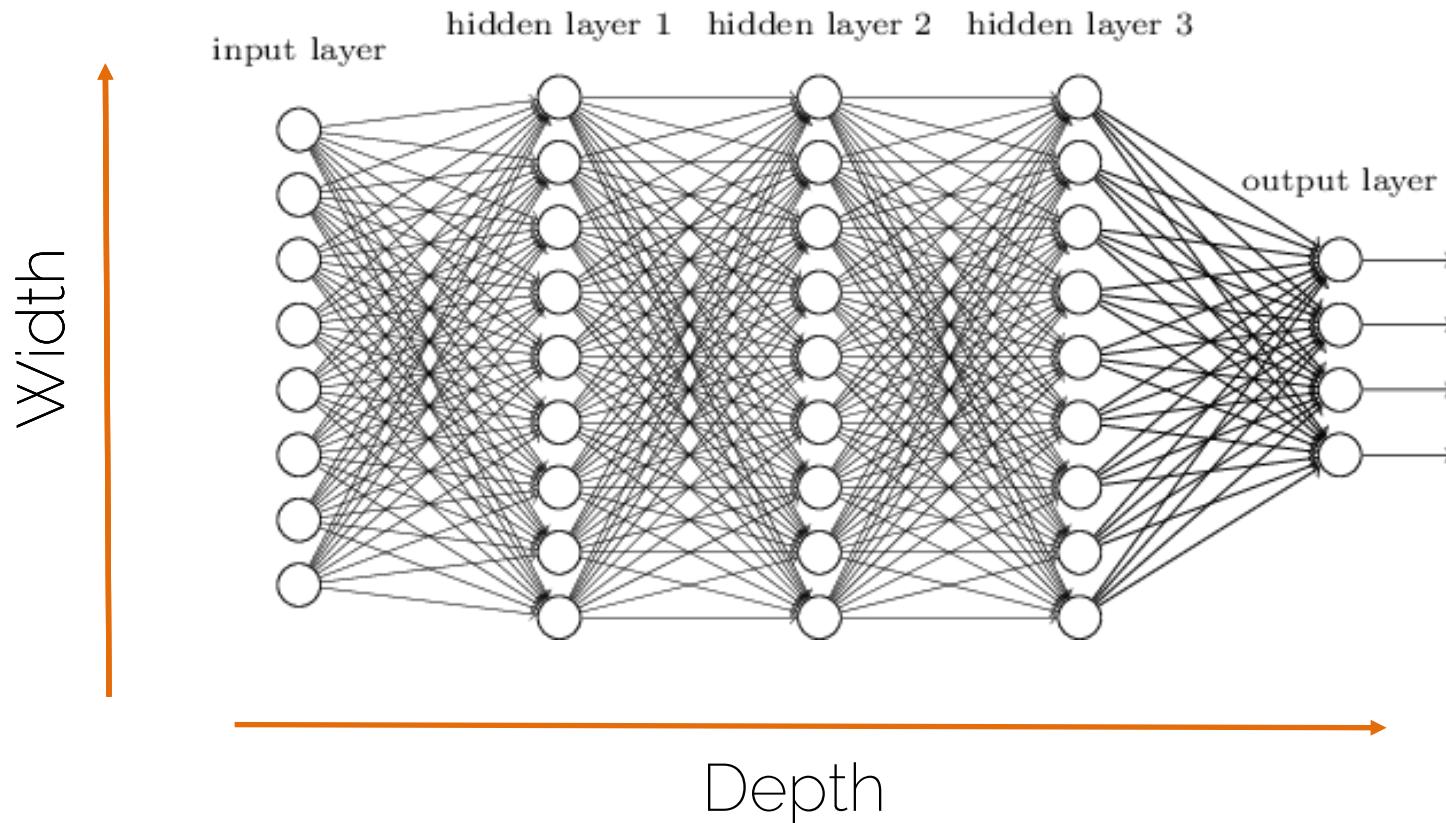
See you next week!

# References

- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 6: Deep Feedforward Networks
- Bishop "Pattern Recognition and Machine Learning" (2006),
  - Chapter 5.5: Regularization in Network Nets
- <http://cs231n.github.io/neural-networks-1/>
- <http://cs231n.github.io/neural-networks-2/>
- <http://cs231n.github.io/neural-networks-3/>

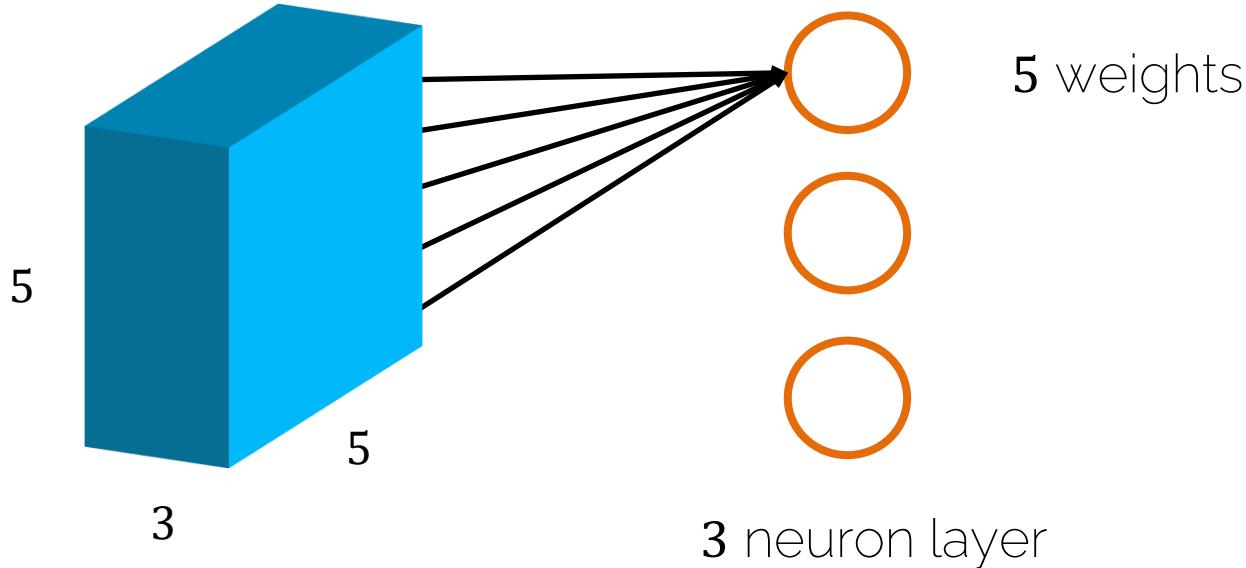
# Lecture 9 - Convolutional Neural Networks

# Fully Connected Neural Network



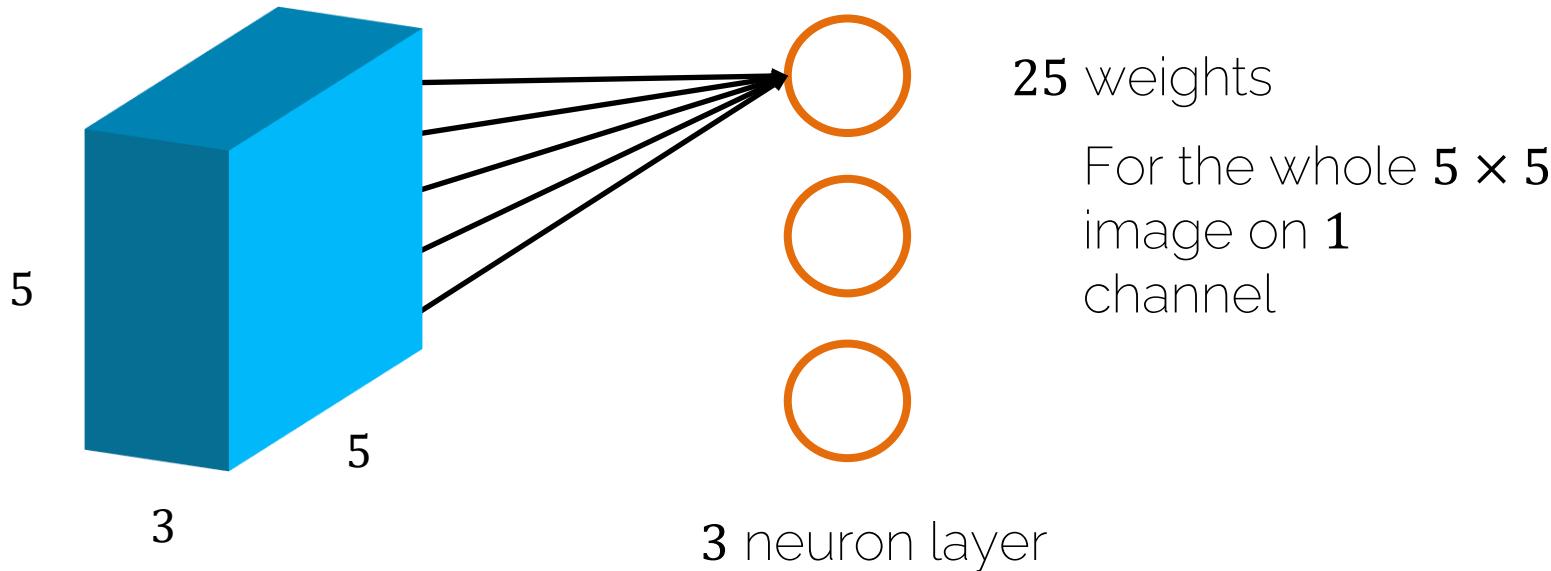
# Problems using FC Layers on Images

- How to process a tiny image with FC layers



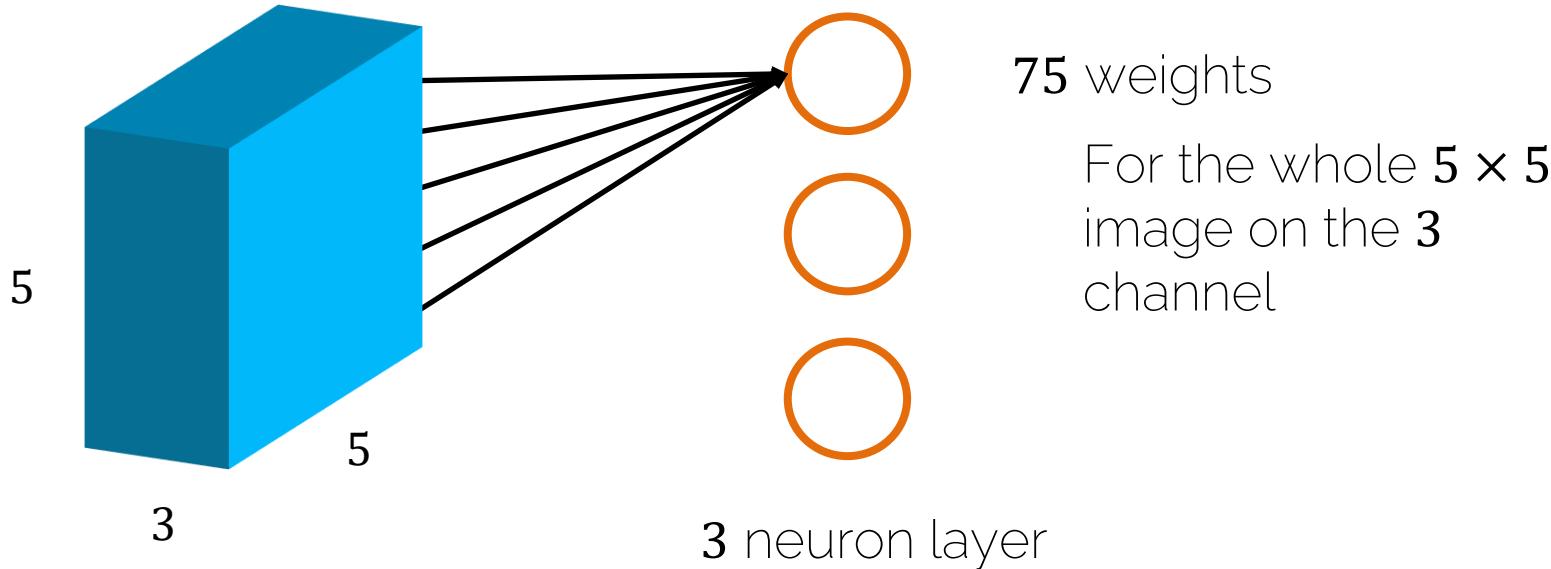
# Problems using FC Layers on Images

- How to process a tiny image with FC layers



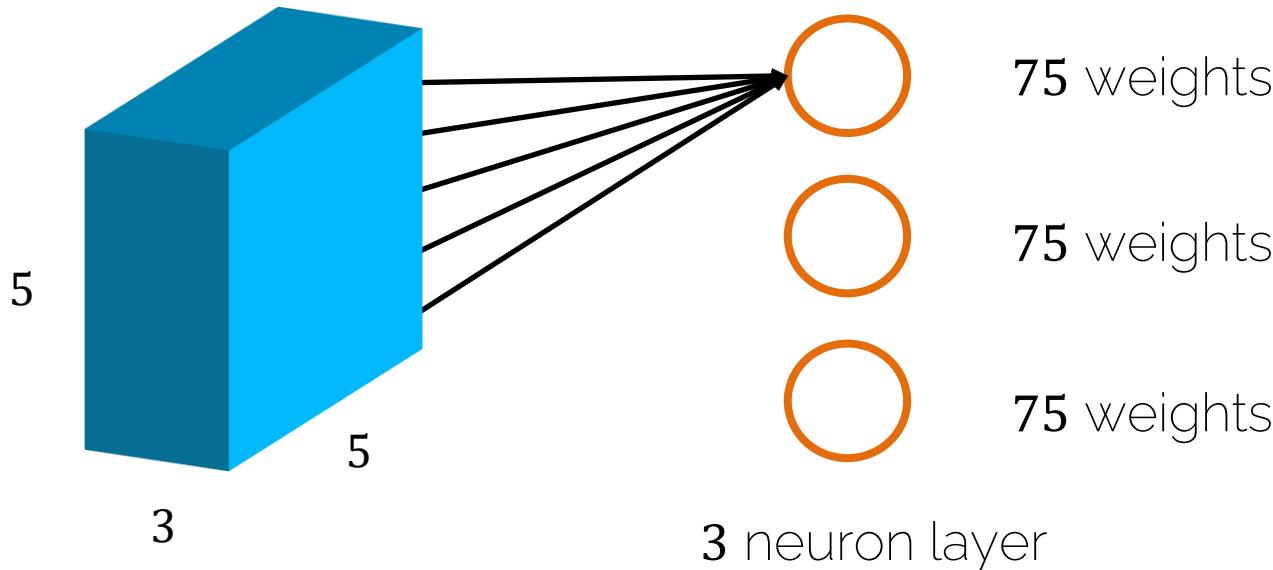
# Problems using FC Layers on Images

- How to process a tiny image with FC layers



# Problems using FC Layers on Images

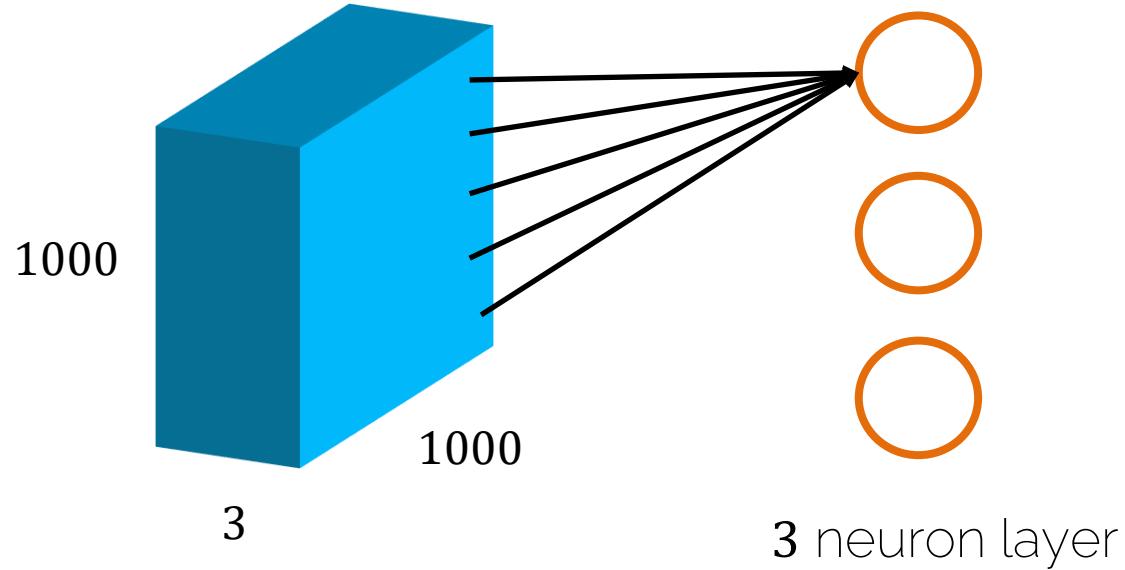
- How to process a tiny image with FC layers



For the whole  
 $5 \times 5$  image on  
the three  
channels per  
neuron

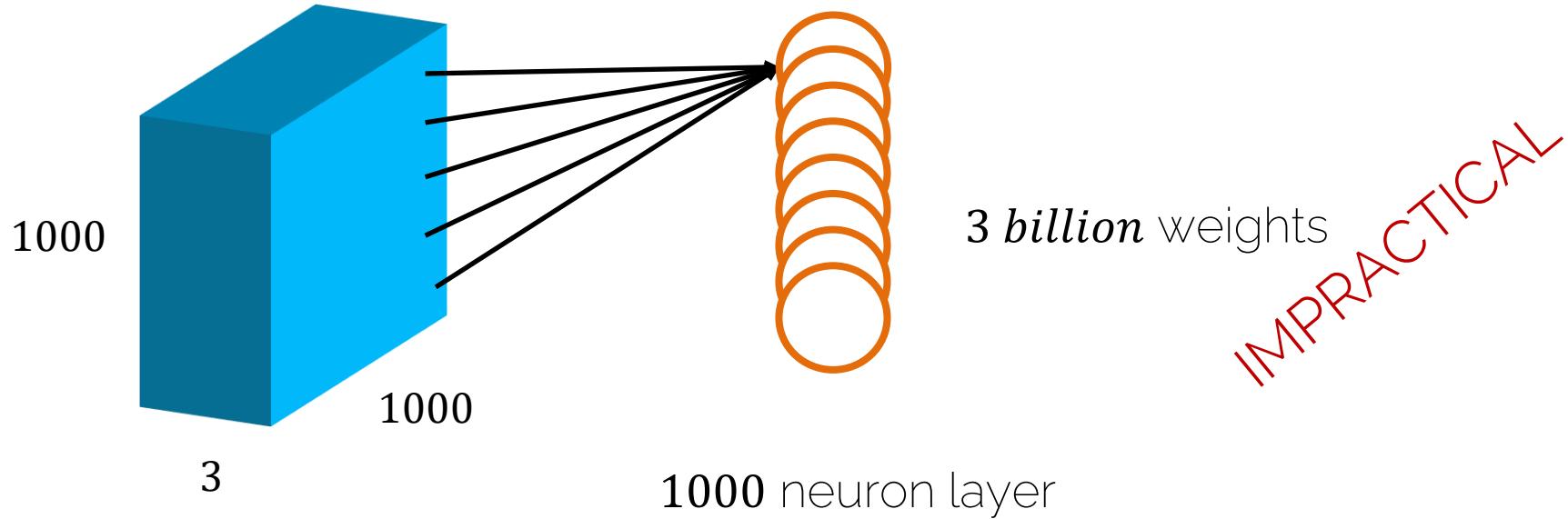
# Problems using FC Layers on Images

- How to process a **normal** image with FC layers



# Problems using FC Layers on Images

- How to process a **normal** image with FC layers



# Why not simply more FC Layers?

We cannot make networks arbitrarily complex

- Why not just go deeper and get better?
  - No structure!!
  - It is just brute force!
  - Optimization becomes hard
  - Performance plateaus / drops!

# Better Way than FC ?

- We want to restrict the degrees of freedom
  - We want a layer with structure
  - Weight sharing → using the same weights for different parts of the image

# Using CNNs in Computer Vision

**Classification**



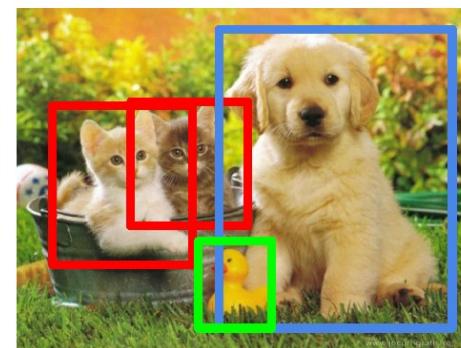
CAT

**Classification + Localization**



Single object

**Object Detection**



CAT, DOG, DUCK

**Instance Segmentation**

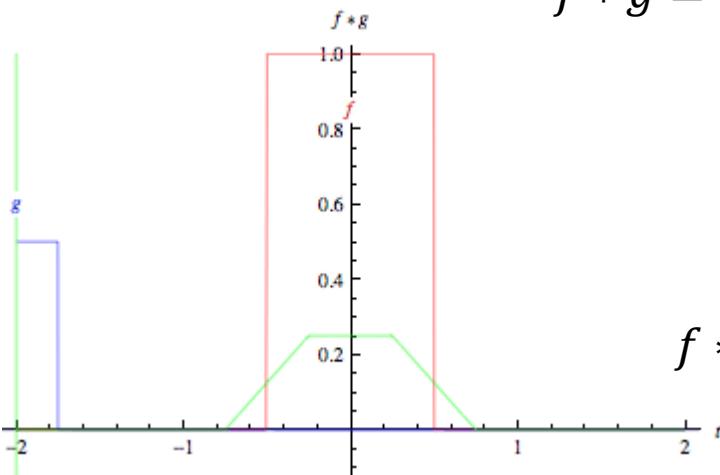


CAT, DOG, DUCK

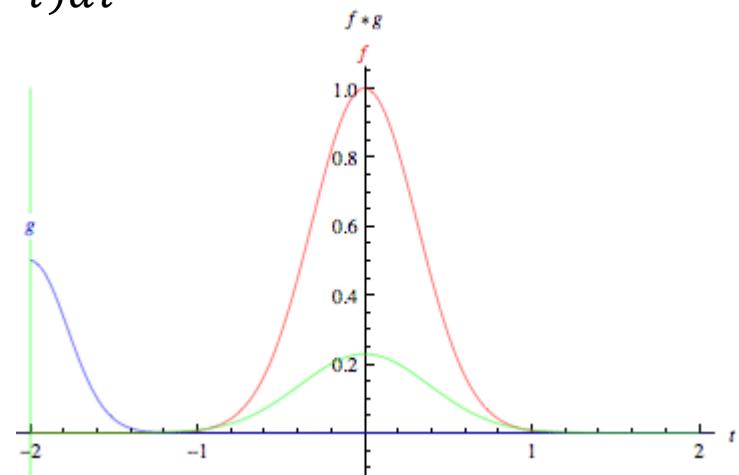
# Convolutions

# What are Convolutions?

$$f * g = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$



Convolution of two box functions



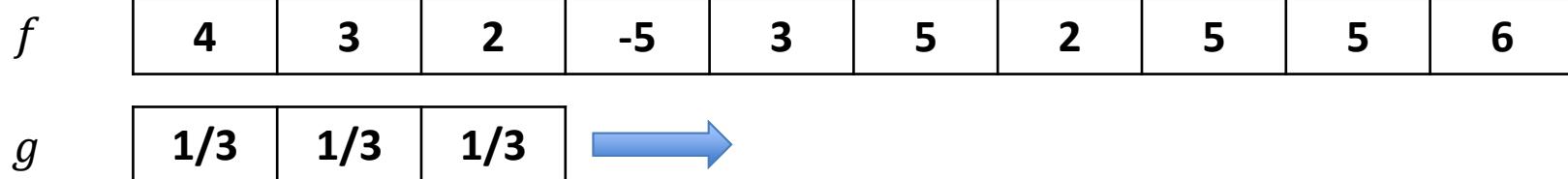
Convolution of two Gaussians

Application of a filter to a function

- The 'smaller' one is typically called the filter kernel

# What are Convolutions?

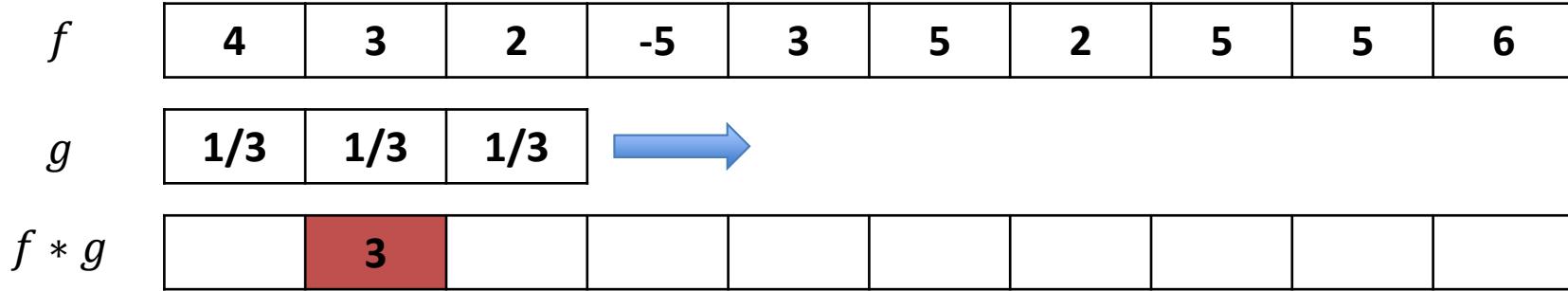
Discrete case: box filter



'Slide' filter kernel from left to right; at each position,  
compute a single value in the output data

# What are Convolutions?

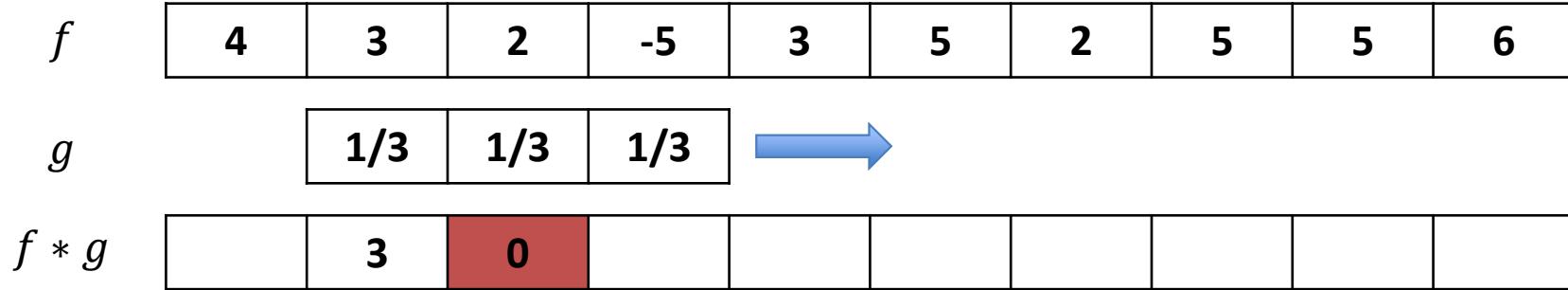
Discrete case: box filter



$$4 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} = 3$$

# What are Convolutions?

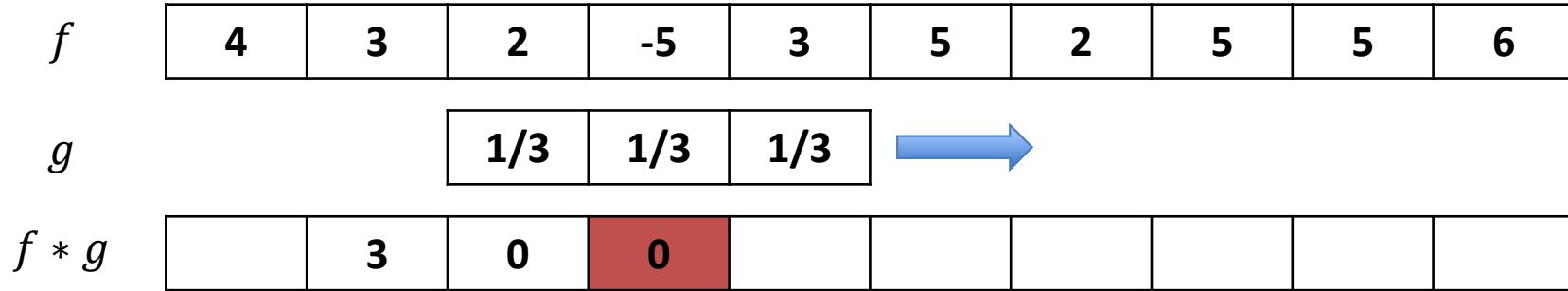
Discrete case: box filter



$$3 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} + (-5) \cdot \frac{1}{3} = 0$$

# What are Convolutions?

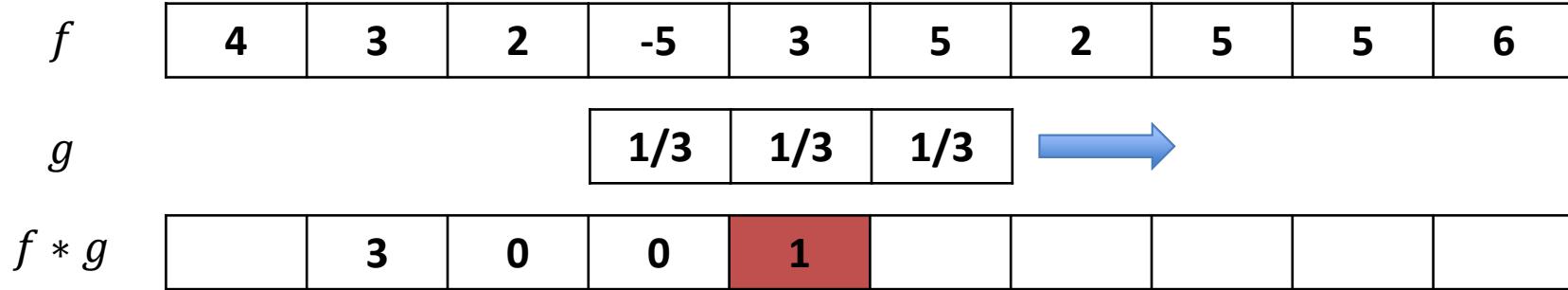
Discrete case: box filter



$$2 \cdot \frac{1}{3} + (-5) \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} = 0$$

# What are Convolutions?

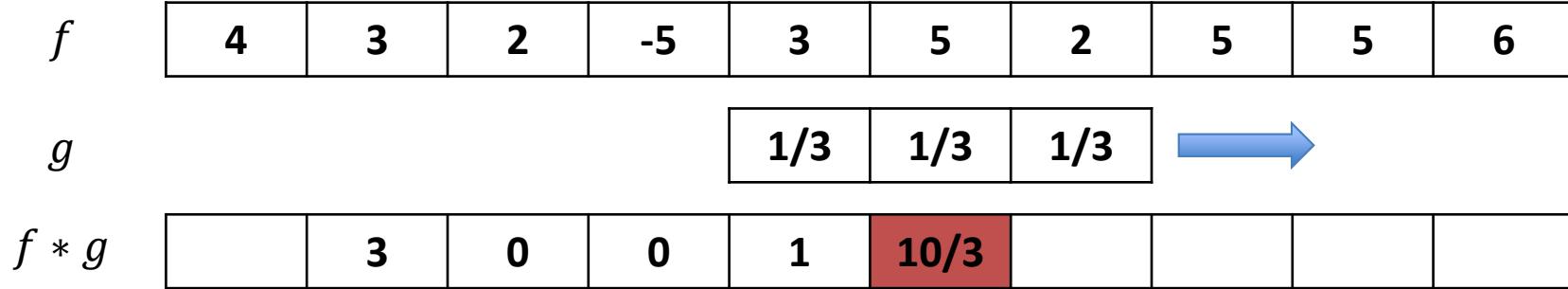
Discrete case: box filter



$$(-5) \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} = 1$$

# What are Convolutions?

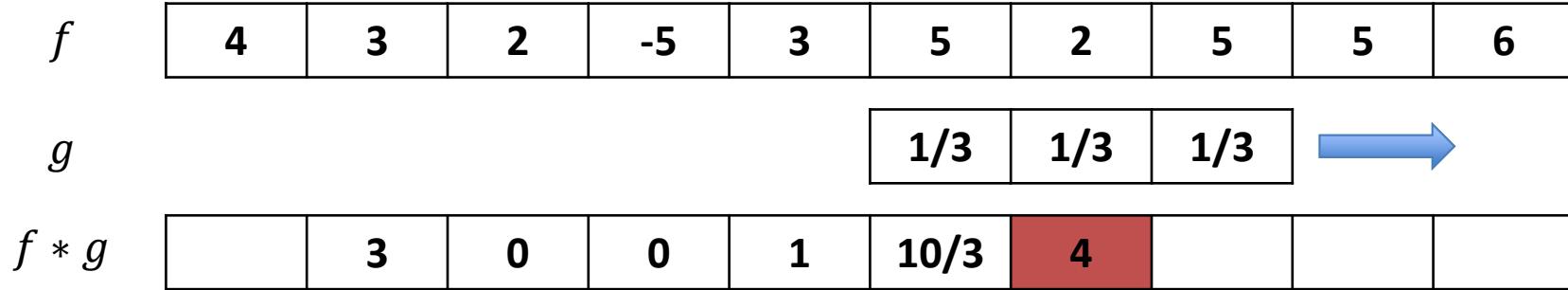
Discrete case: box filter



$$3 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} = \frac{10}{3}$$

# What are Convolutions?

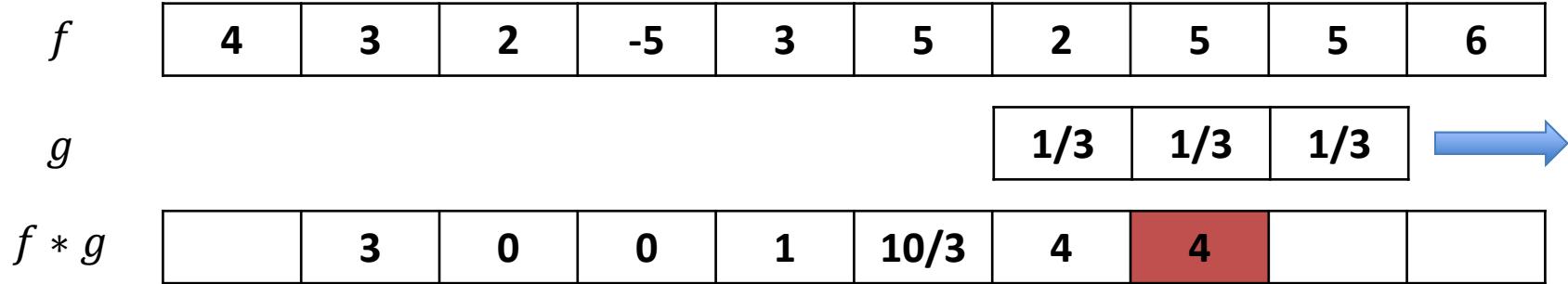
Discrete case: box filter



$$5 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} = 4$$

# What are Convolutions?

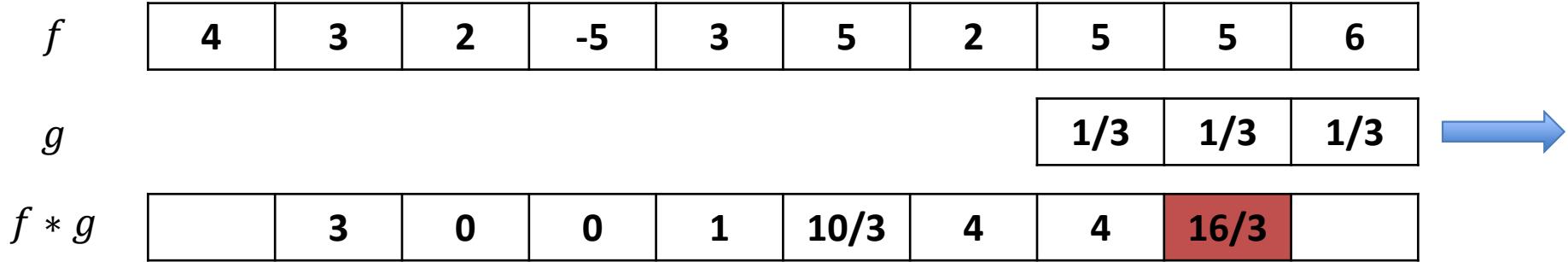
Discrete case: box filter



$$2 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} = 4$$

# What are Convolutions?

Discrete case: box filter



$$5 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} + 6 \cdot \frac{1}{3} = \frac{16}{3}$$

# What are Convolutions?

Discrete case: box filter

4	3	2	-5	3	5	2	5	5	6
---	---	---	----	---	---	---	---	---	---

1/3	1/3	1/3
-----	-----	-----

??	3	0	0	1	10/3	4	4	16/3	??
----	---	---	---	---	------	---	---	------	----

What to do at boundaries?

# What are Convolutions?

Discrete case: box filter

4	3	2	-5	3	5	2	5	5	6
---	---	---	----	---	---	---	---	---	---

1/3	1/3	1/3
-----	-----	-----

??	3	0	0	1	10/3	4	4	16/3	??
----	---	---	---	---	------	---	---	------	----

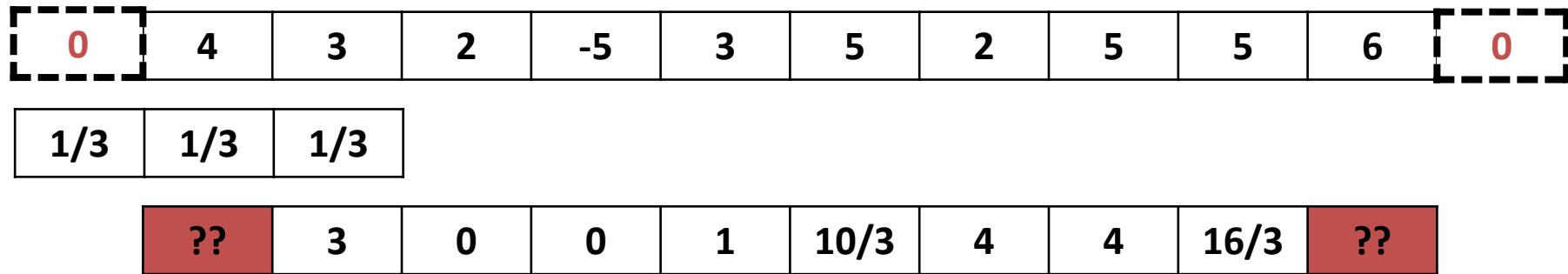
What to do at boundaries?

Option 1: Shrink

3	0	0	1	10/3	4	4	16/3
---	---	---	---	------	---	---	------

# What are Convolutions?

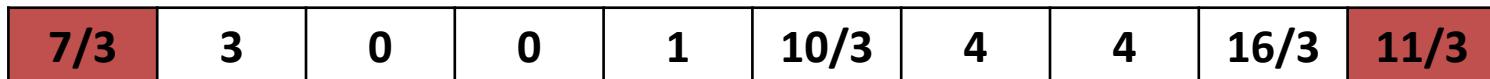
Discrete case: box filter



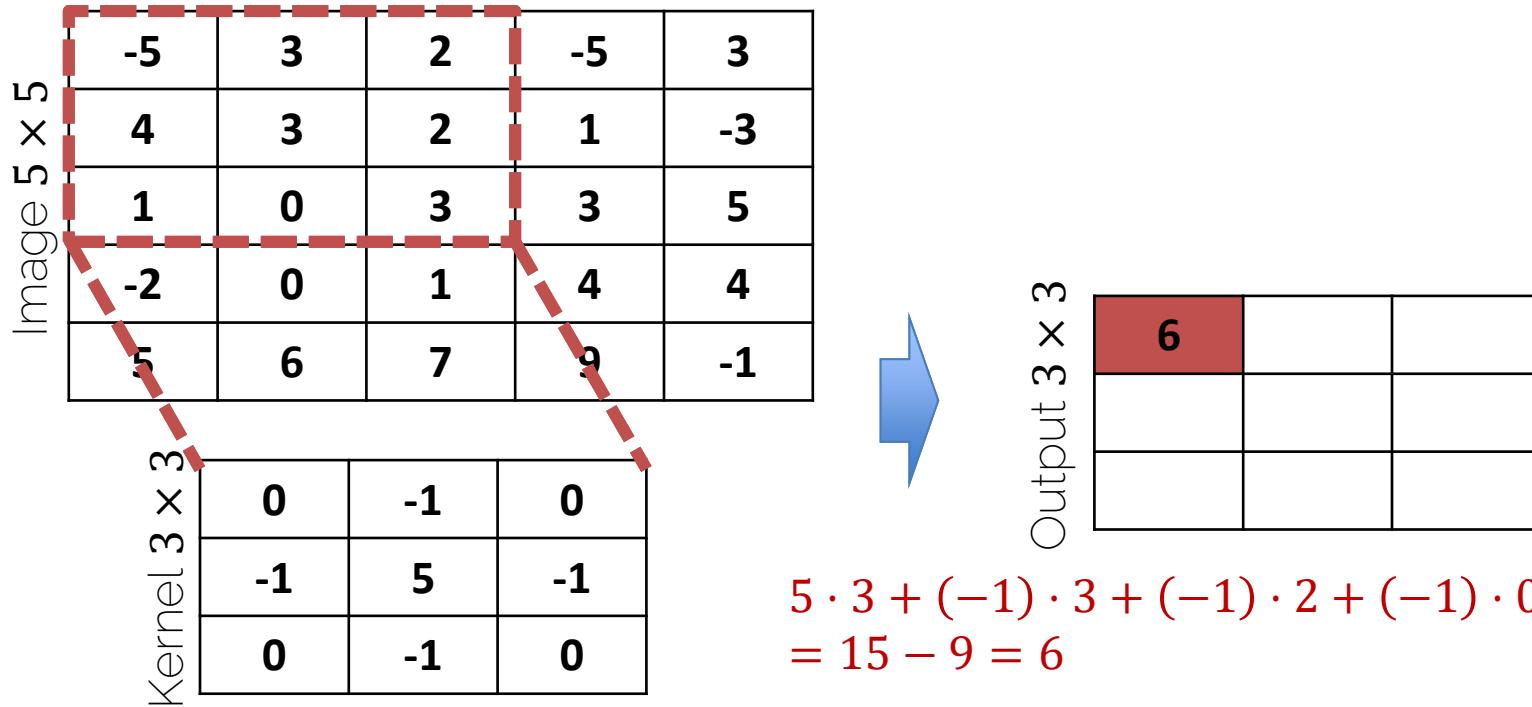
$$0 \cdot \frac{1}{3} + 4 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} = \frac{7}{3}$$

What to do at boundaries?

Option 2: Pad (often 0's)



# Convolutions on Images



# Convolutions on Images

Image  $5 \times 5$

-5	3	2	-5	3
4	3	2	1	-3
1	0	3	3	5
-2	0	1	4	4
5	6	7	9	-1

Kernel  $3 \times 3$

0	-1	0
-1	5	-1
0	-1	0

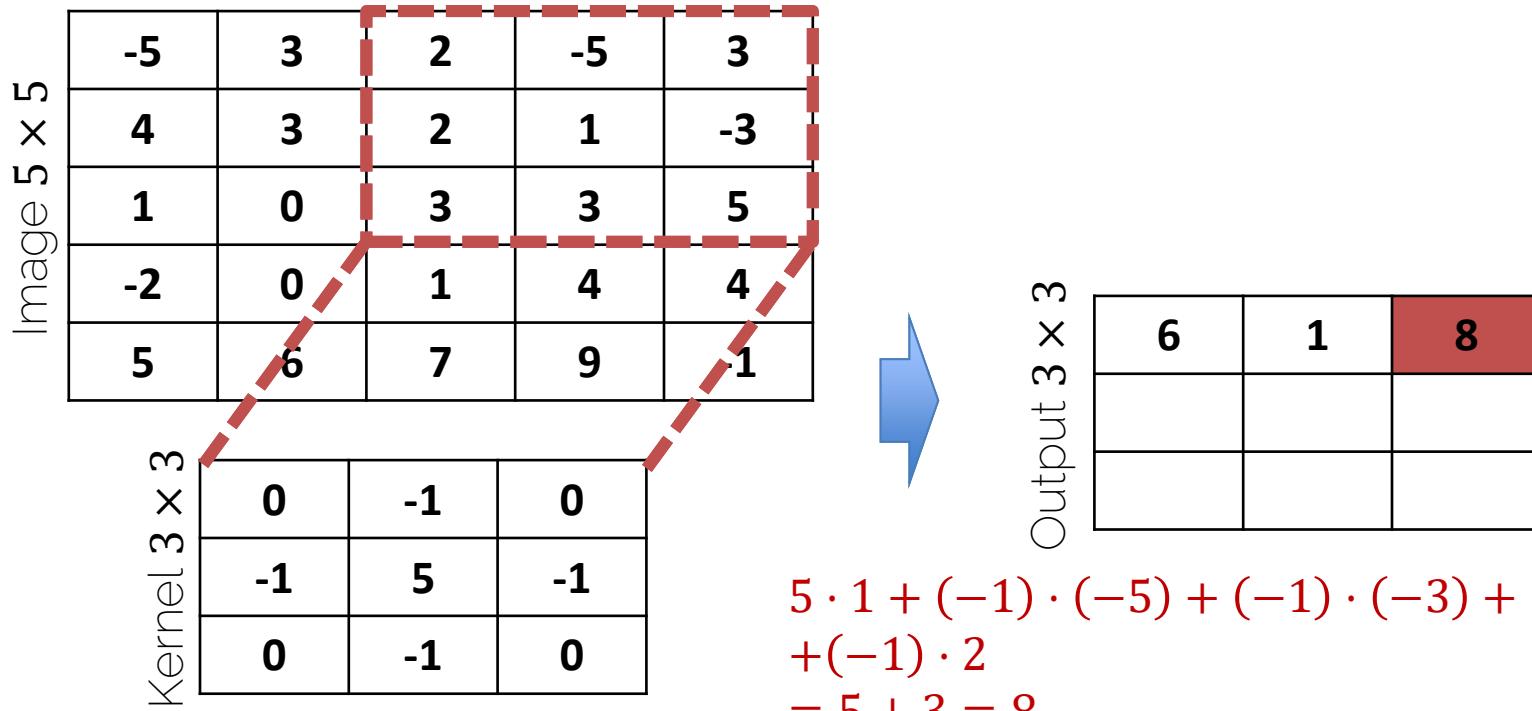


Output  $3 \times 3$

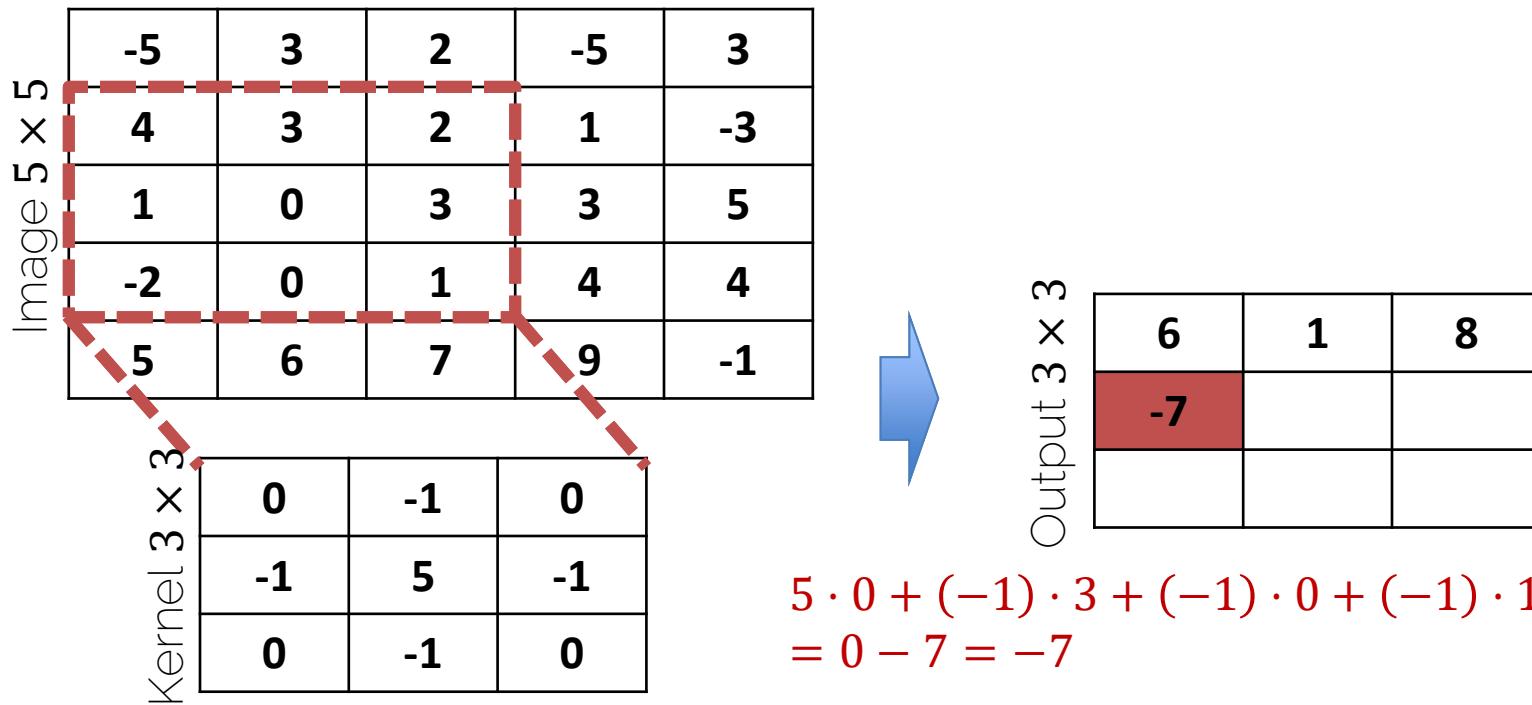
6	1	

$$5 \cdot 2 + (-1) \cdot 2 + (-1) \cdot 1 + (-1) \cdot 3 + (-1) \cdot 3 \\ = 10 - 9 = 1$$

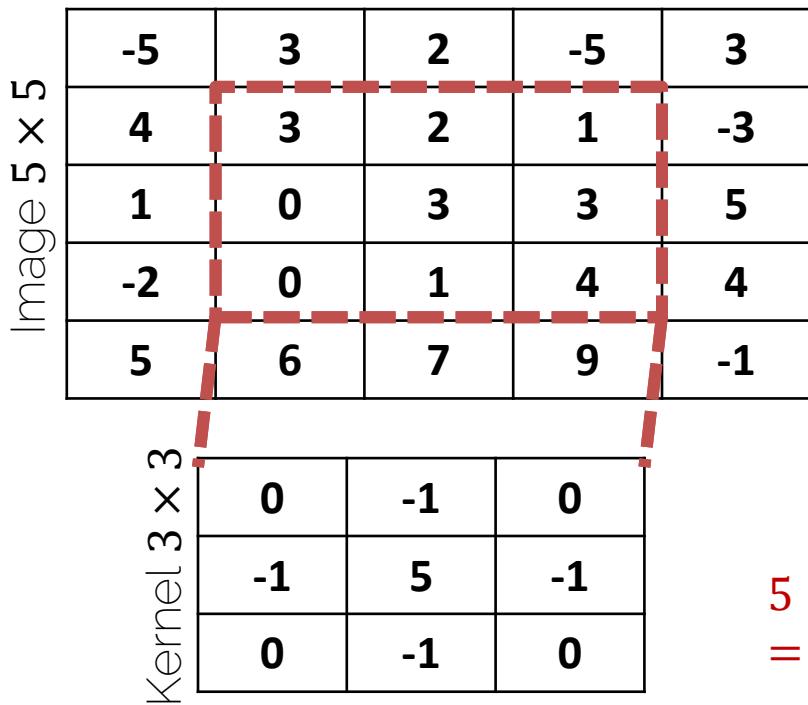
# Convolutions on Images



# Convolutions on Images



# Convolutions on Images

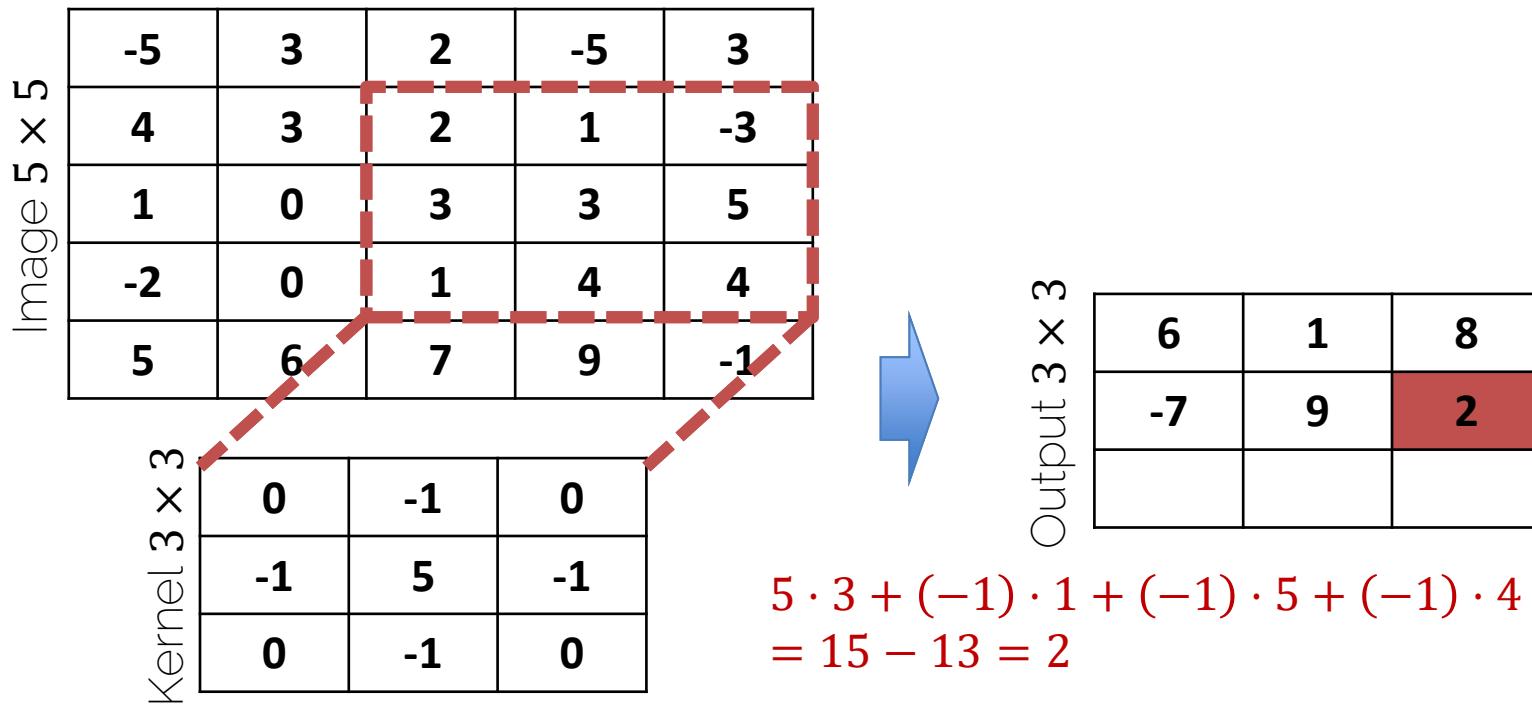


Output  $3 \times 3$

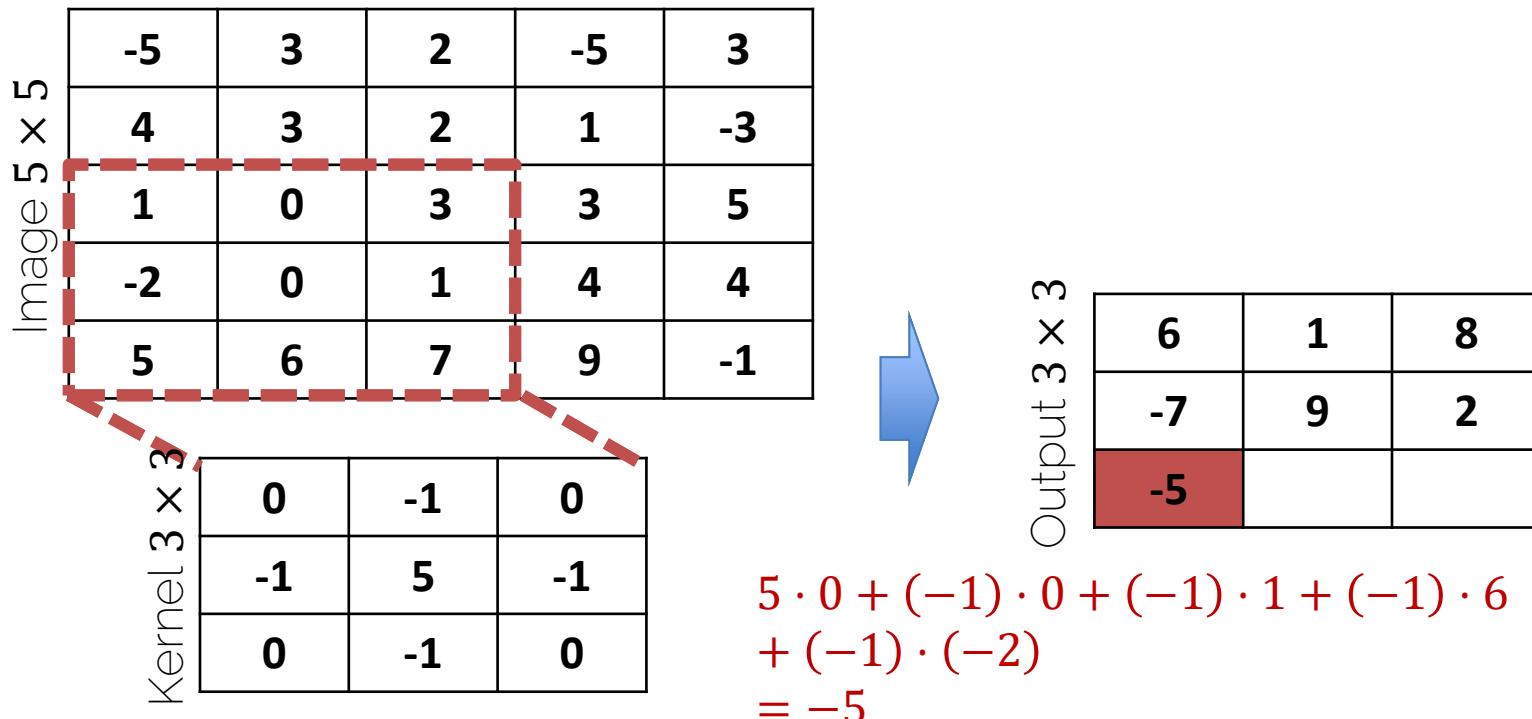
6	1	8
-7	9	

$$\begin{aligned} & 5 \cdot 3 + (-1) \cdot 2 + (-1) \cdot 3 + (-1) \cdot 1 + (-1) \cdot 0 \\ & = 15 - 6 = 9 \end{aligned}$$

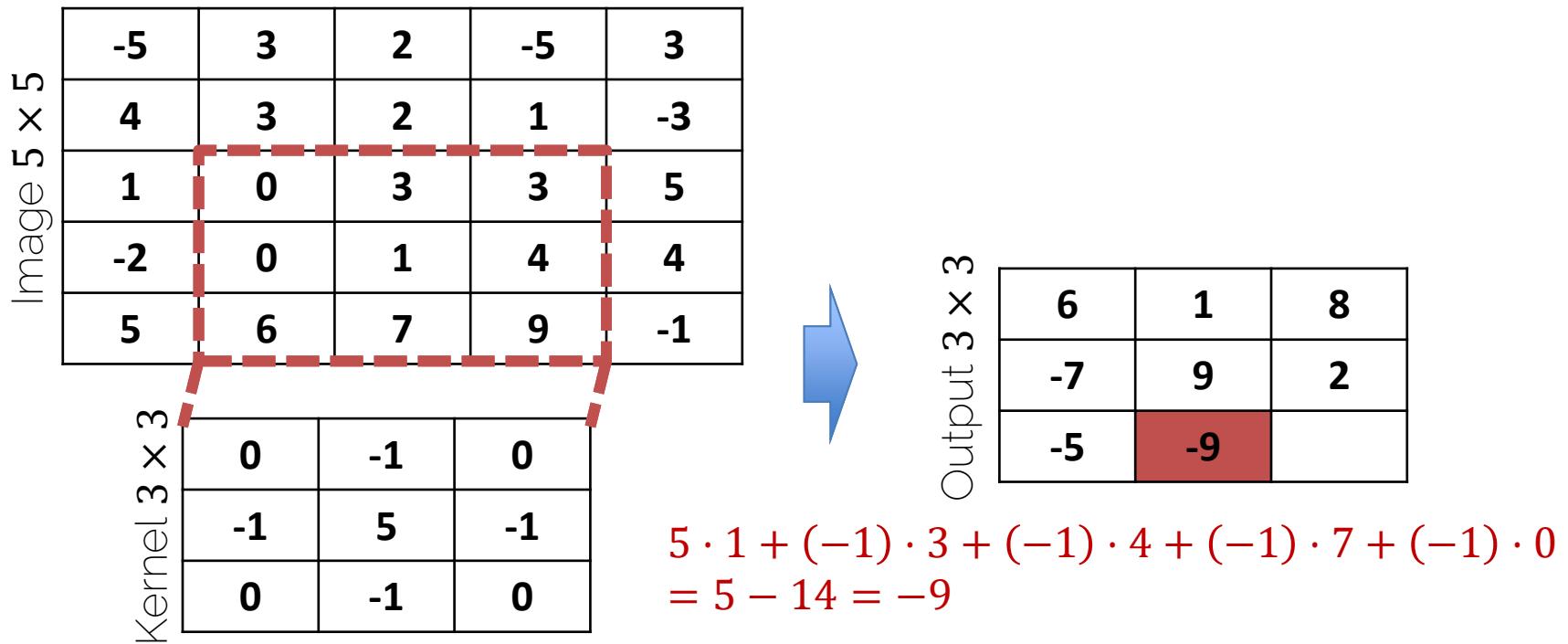
# Convolutions on Images



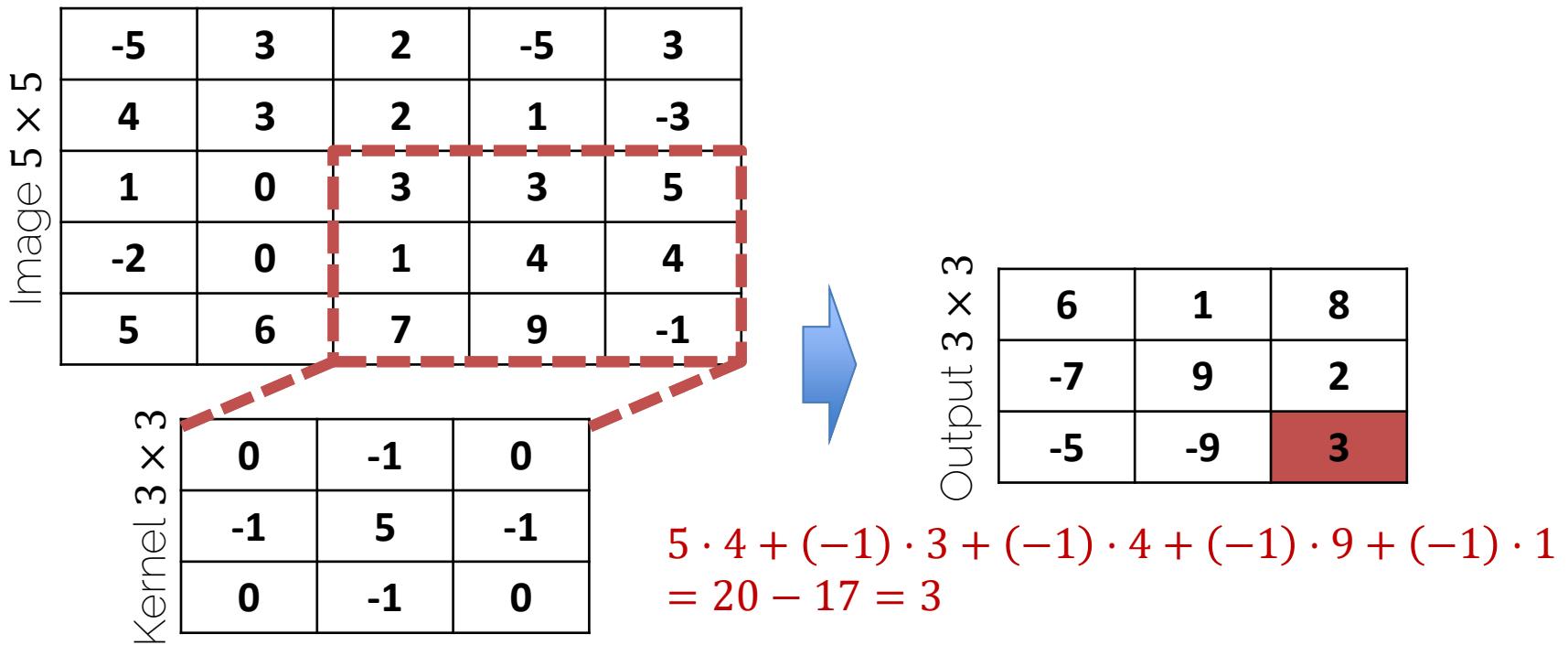
# Convolutions on Images



# Convolutions on Images



# Convolutions on Images



# Image Filters

- Each kernel gives us a different image filter

Input



LET'S LEARN THESE FILTERS!



Edge detection

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



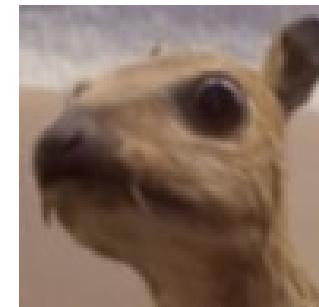
Box mean

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



Sharpen

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

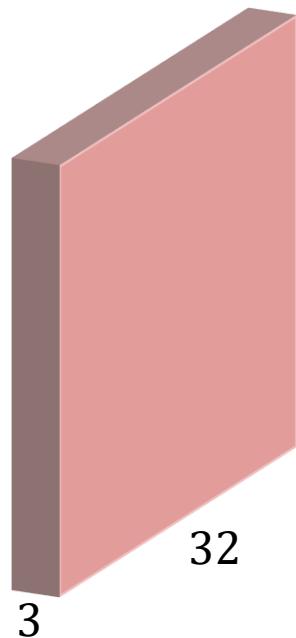


Gaussian blur

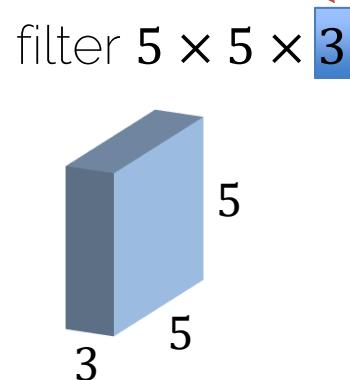
$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

# Convolutions on RGB Images

width      height      depth  
image  $32 \times 32 \times 3$



Depth dimension **\*must\*** match;  
i.e., filter extends the full depth of the input



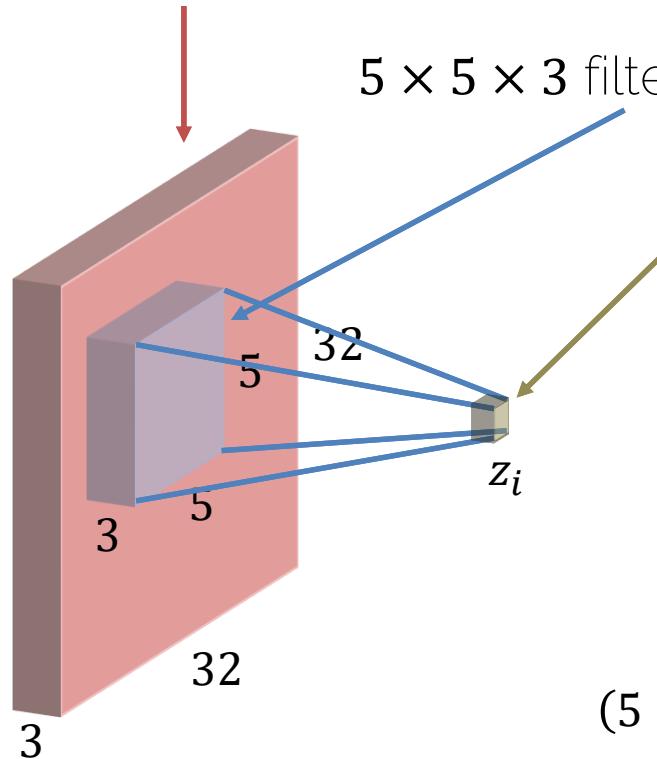
Convolve filter with image  
i.e., 'slide' over it and:  

- apply filter at each location
- dot products

Images have depth: e.g. RGB  $\rightarrow$  3 channels

# Convolutions on RGB Images

$32 \times 32 \times 3$  image (pixels  $\mathbf{X}$ )



$5 \times 5 \times 3$  filter (weights vector  $\mathbf{w}$ )

1 number at a time:

equal to dot product between  
filter weights  $\mathbf{w}$  and  $\mathbf{x}_i - \mathbf{th}$  chunk of  
the image. Here:  $5 \cdot 5 \cdot 3 = 75$ -dim  
dot product + bias

$$z_i = \mathbf{w}^T \mathbf{x}_i + b$$

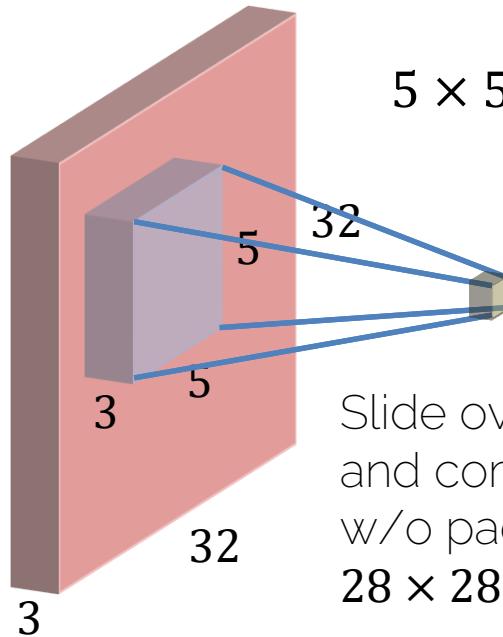
$(5 \times 5 \times 3) \times 1$

$(5 \times 5 \times 3) \times 1$

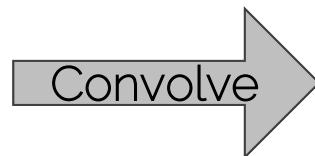
1

# Convolutions on RGB Images

$32 \times 32 \times 3$  image

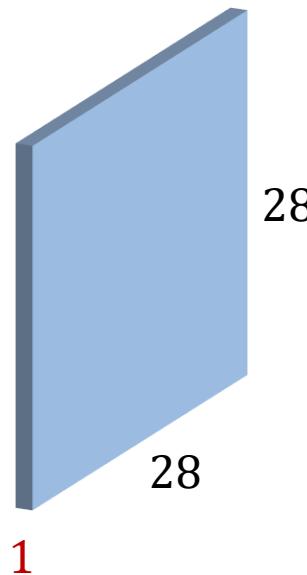


$5 \times 5 \times 3$  filter



Slide over all spatial locations  $x_i$   
and compute all output  $z_i$ :  
w/o padding, there are  
 $28 \times 28$  locations

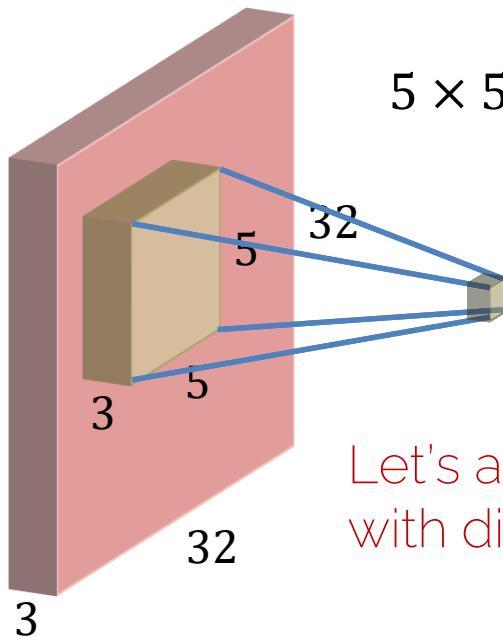
Activation map  
(also feature map)



# Convolution Layer

# Convolution Layer

$32 \times 32 \times 3$  image

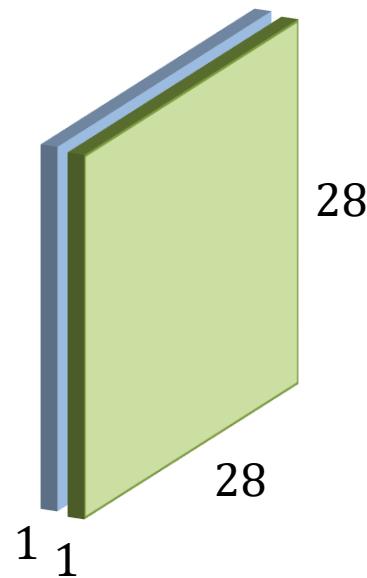


$5 \times 5 \times 3$  filter

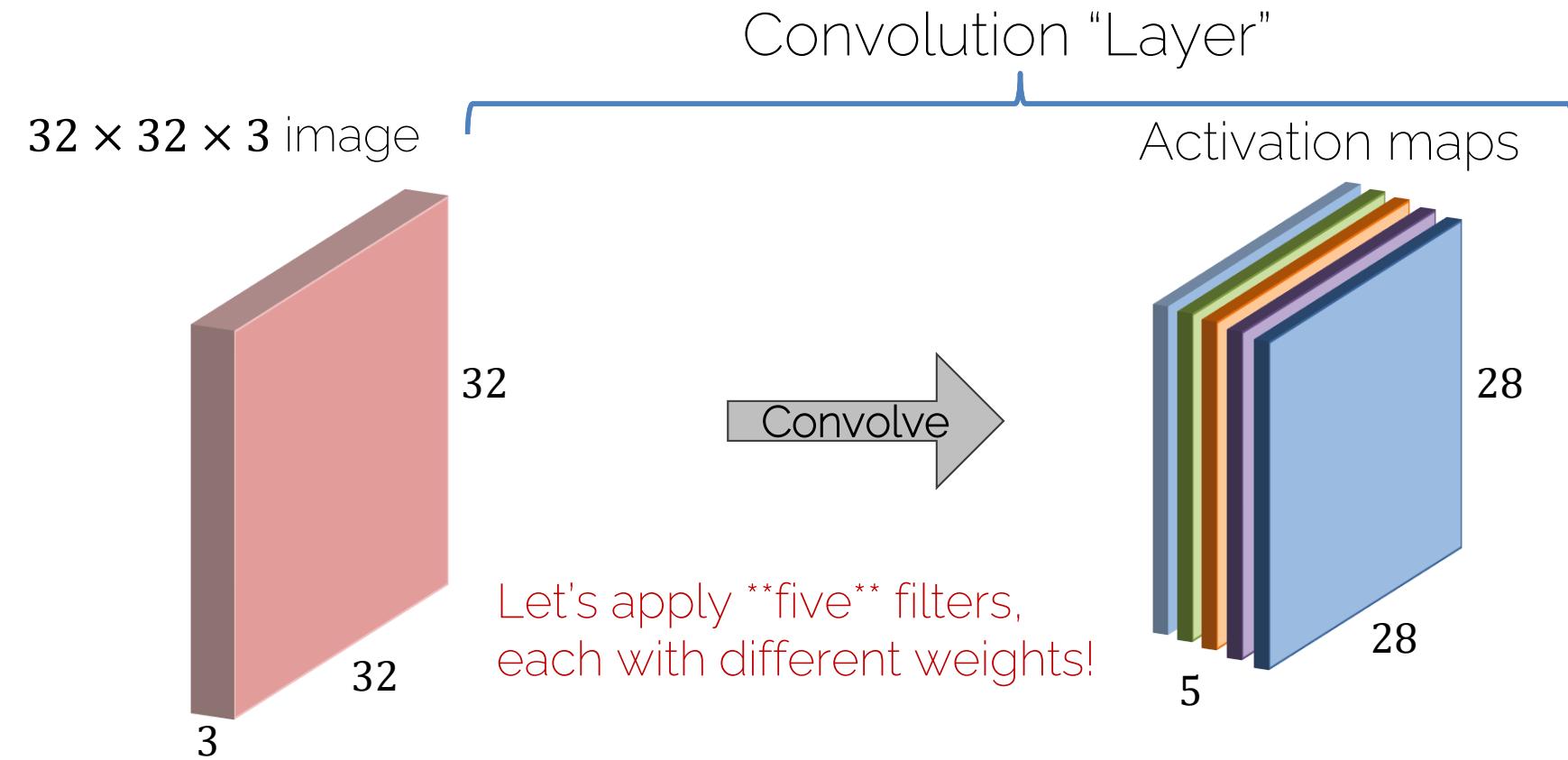


Let's apply a different filter  
with different weights!

Activation maps



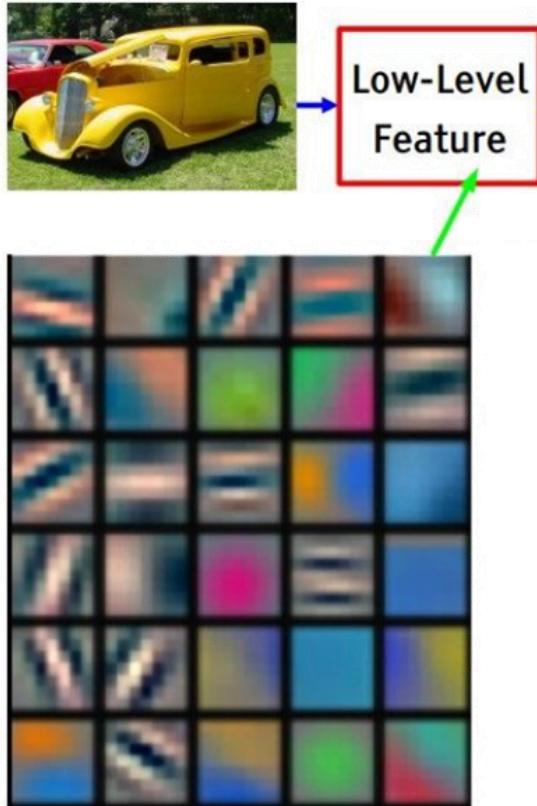
# Convolution Layer



# Convolution Layer

- A basic layer is defined by
  - Filter width and height (depth is implicitly given)
  - Number of different filter banks (#weight sets)
- Each filter captures a different image characteristic

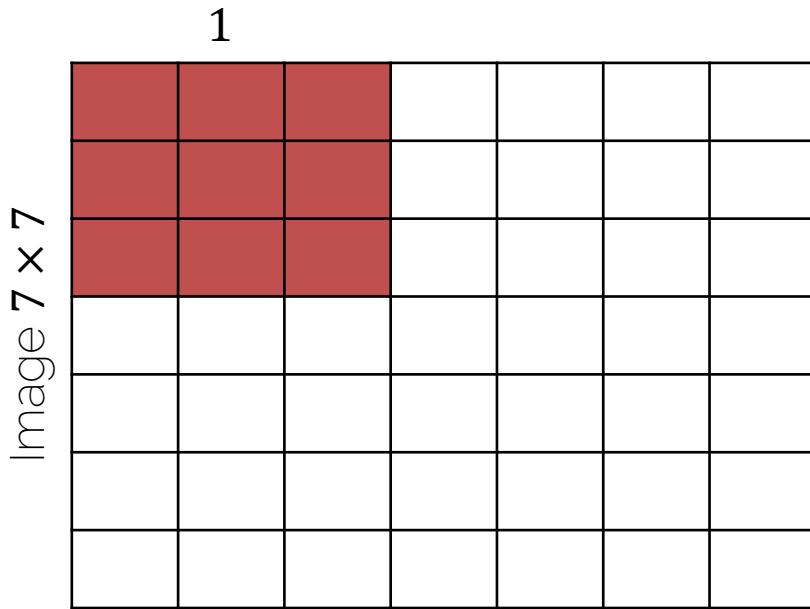
# Different Filters



- Each filter captures different image characteristics:
  - Horizontal edges
  - Vertical edges
  - Circles
  - Squares
  - ...

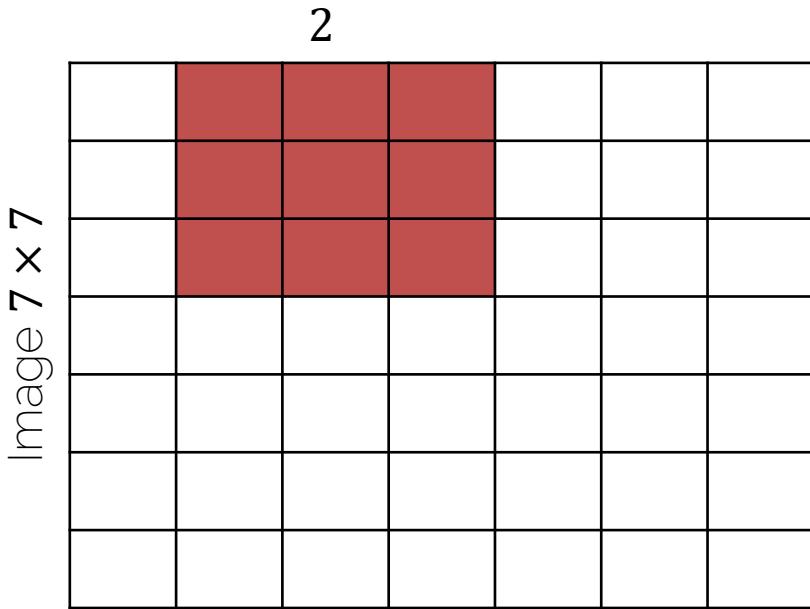
# Dimensions of a Convolution Layer

# Convolution Layers: Dimensions



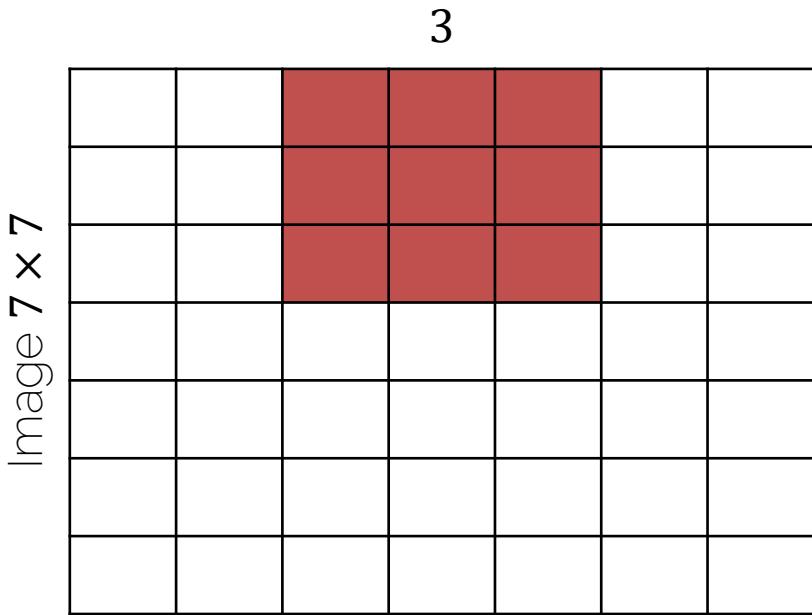
Input:  $7 \times 7$   
Filter:  $3 \times 3$   
Output:  $5 \times 5$

# Convolution Layers: Dimensions



Input:  $7 \times 7$   
Filter:  $3 \times 3$   
Output:  $5 \times 5$

# Convolution Layers: Dimensions



Input:

$7 \times 7$

Filter:

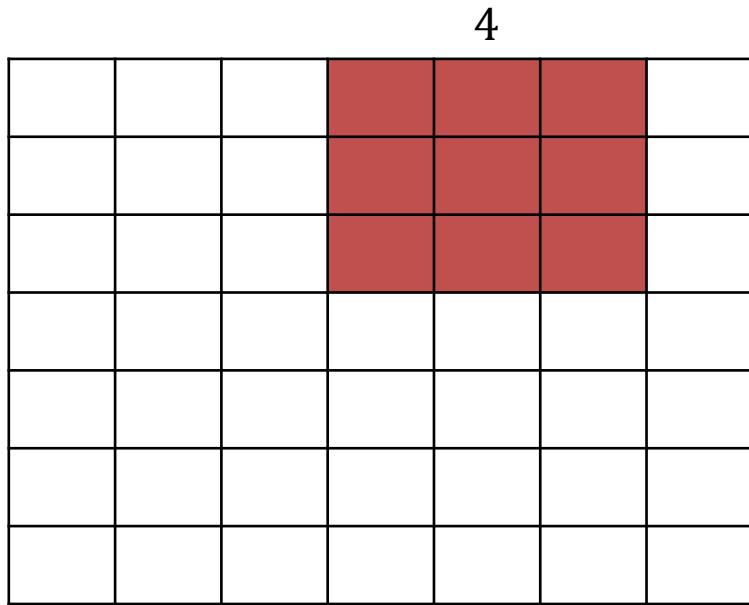
$3 \times 3$

Output:

$5 \times 5$

# Convolution Layers: Dimensions

Image  $7 \times 7$



Input:

$7 \times 7$

Filter:

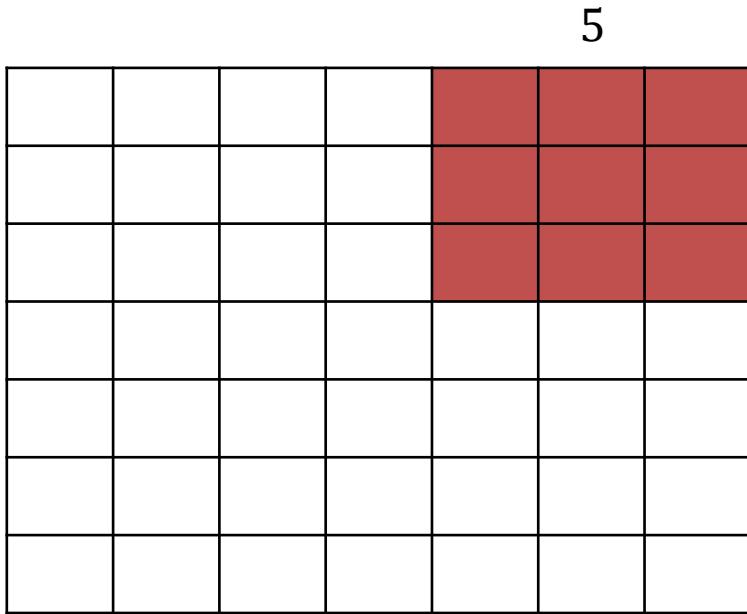
$3 \times 3$

Output:

$5 \times 5$

# Convolution Layers: Dimensions

Image  $7 \times 7$



Input:

$$7 \times 7$$

Filter:

$$3 \times 3$$

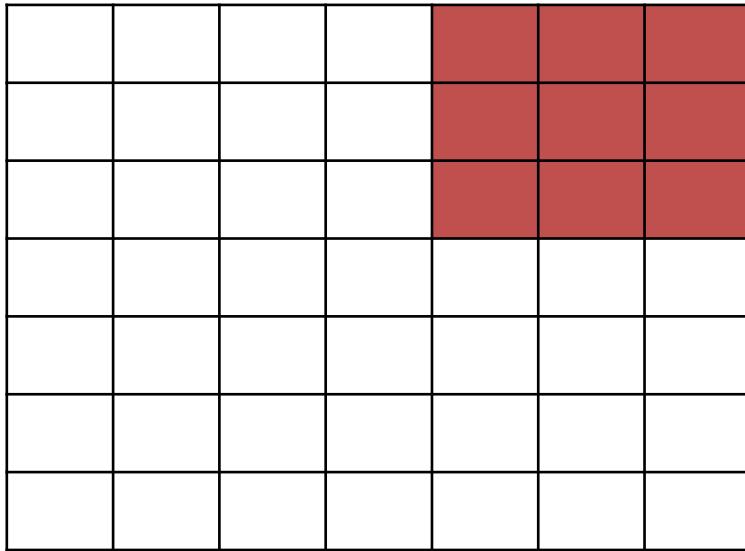
Output:

$$5 \times 5$$

# Convolution Layers: Stride

With a stride of 1

Image  $7 \times 7$

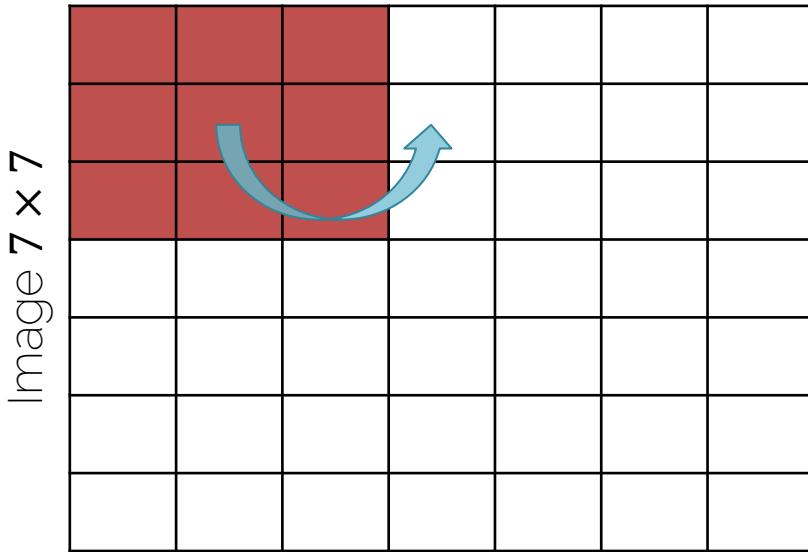


Input:	$7 \times 7$
Filter:	$3 \times 3$
Stride:	1
Output:	$5 \times 5$

Stride of  $S$ : apply filter every  $S$ -th spatial location;  
i.e. subsample the image

# Convolution Layers: Stride

With a stride of 2

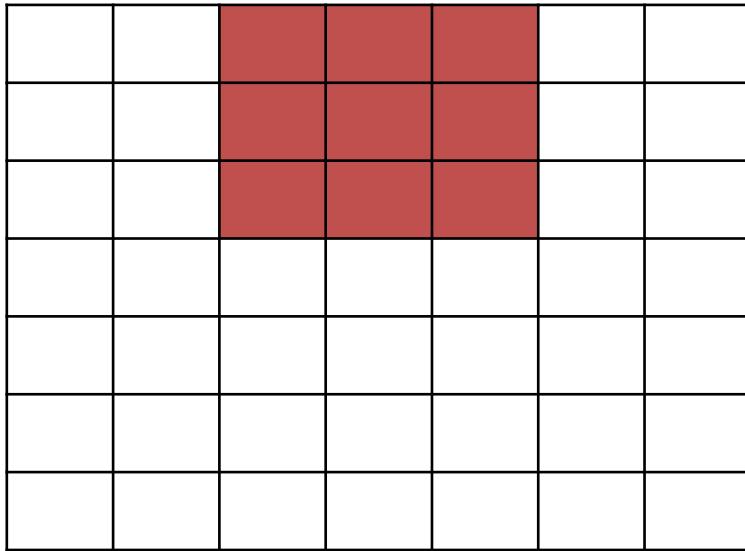


Input:	$7 \times 7$
Filter:	$3 \times 3$
Stride:	2
Output:	$3 \times 3$

# Convolution Layers: Stride

With a stride of 2

Image  $7 \times 7$

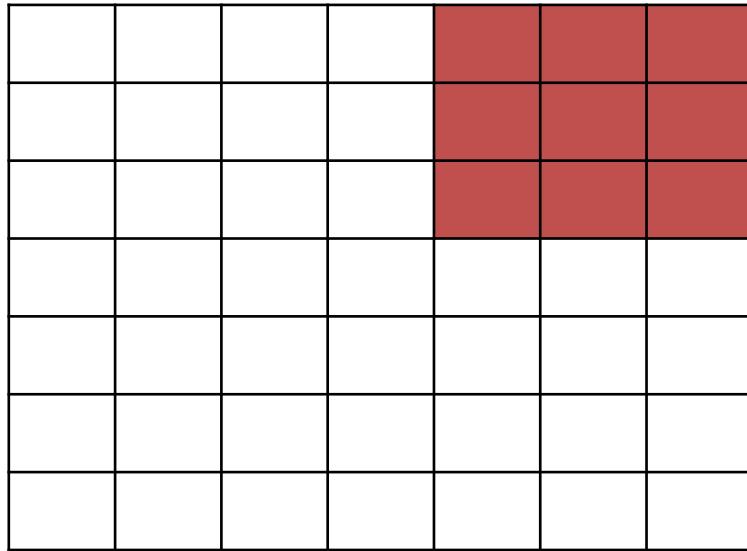


Input:	$7 \times 7$
Filter:	$3 \times 3$
Stride:	2
Output:	$3 \times 3$

# Convolution Layers: Stride

With a stride of 2

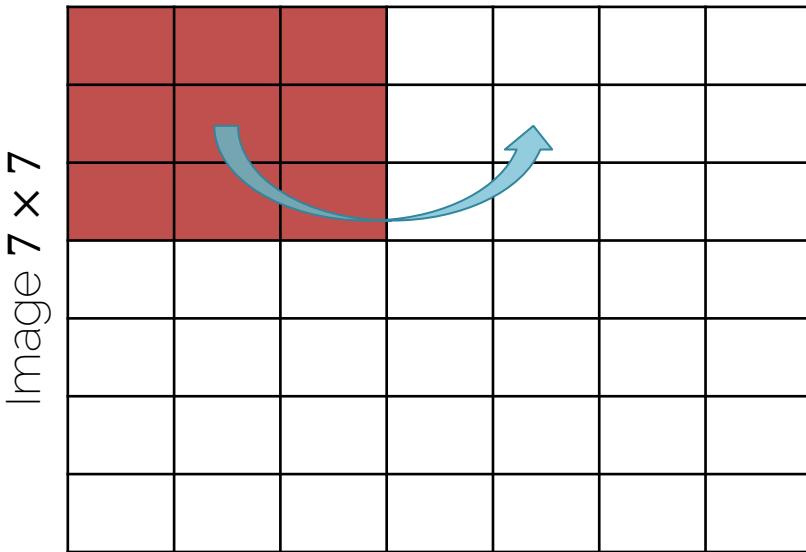
Image  $7 \times 7$



Input:	$7 \times 7$
Filter:	$3 \times 3$
Stride:	2
Output:	$3 \times 3$

# Convolution Layers: Stride

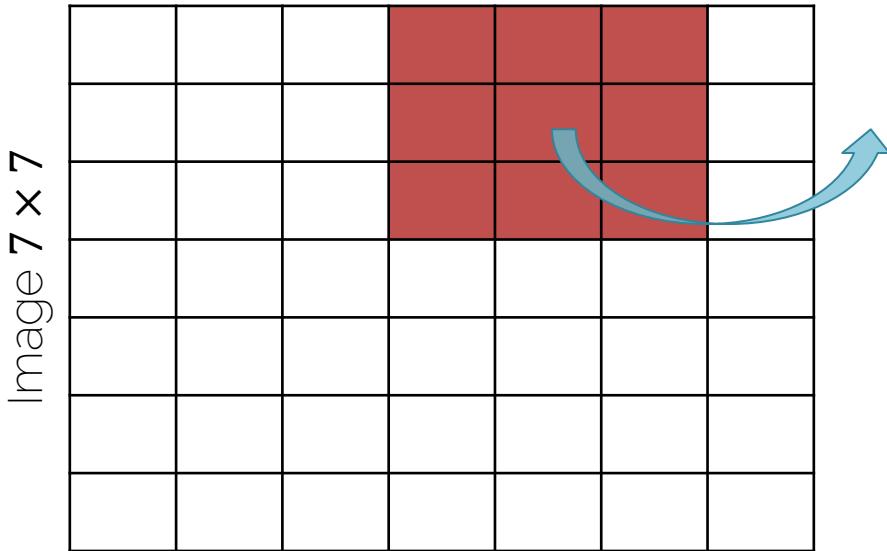
With a stride of 3



Input:	$7 \times 7$
Filter:	$3 \times 3$
Stride:	3
Output:	? $\times$ ?

# Convolution Layers: Stride

With a stride of 3



Input:

$7 \times 7$

Filter:

$3 \times 3$

Stride:

3

Output:

?  $\times$  ?

# Convolution Layers: Stride

With a stride of 3

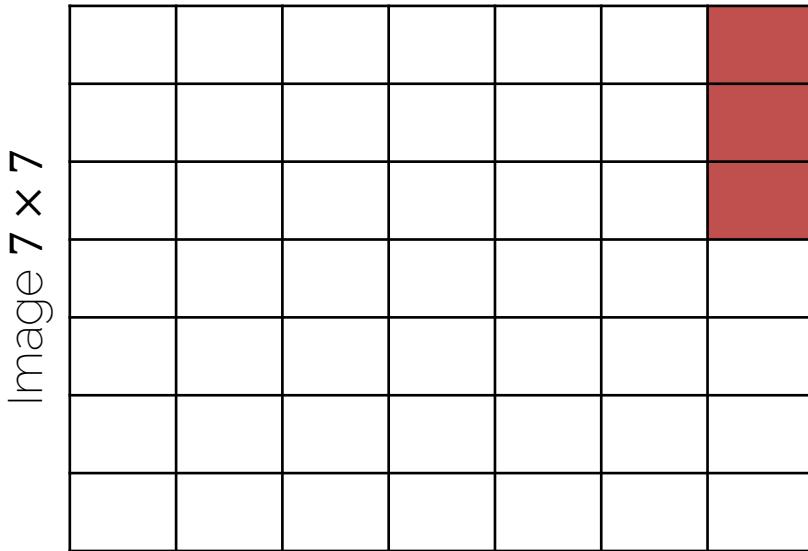


Image  $7 \times 7$

Input:

$7 \times 7$

Filter:

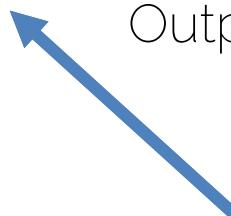
$3 \times 3$

Stride:

3

Output:

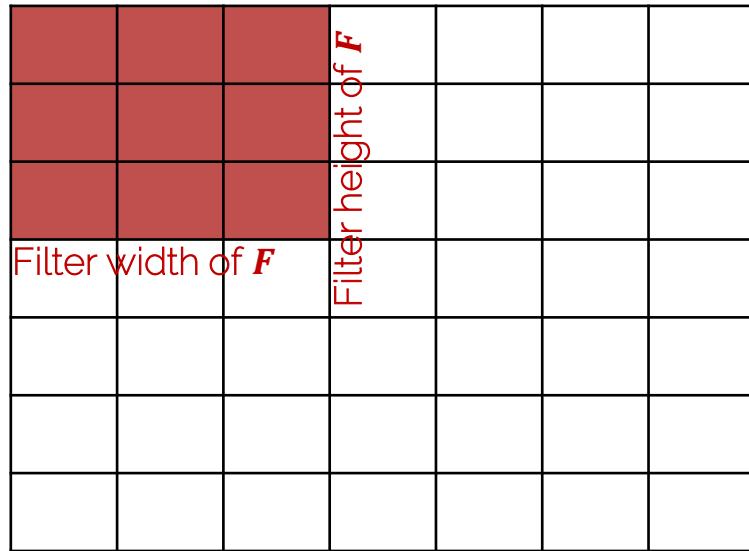
?  $\times$  ?



Does not really fit (remainder left)  
→ Illegal stride for input & filter size!

# Convolution Layers: Dimensions

Input width of  $N$



Input:  $N \times N$

Filter:  $F \times F$

Stride:  $S$

Output:  $\left(\frac{N-F}{S} + 1\right) \times \left(\frac{N-F}{S} + 1\right)$

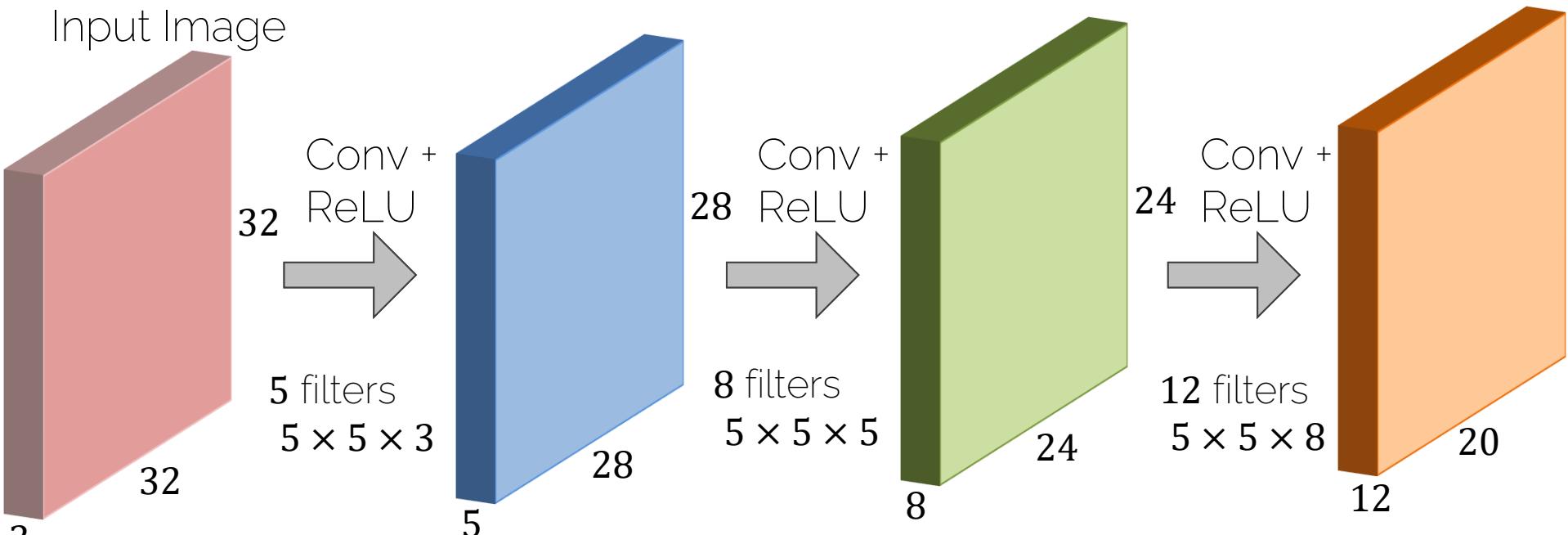
$$N = 7, F = 3, S = 1: \frac{7-3}{1} + 1 = 5$$

$$N = 7, F = 3, S = 2: \frac{7-3}{2} + 1 = 3$$

$$N = 7, F = 3, S = 3: \frac{7-3}{3} + 1 = 2.\bar{3}$$

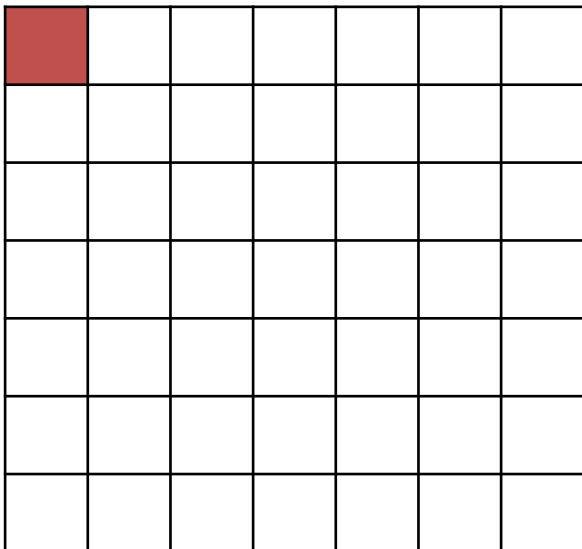
Fractions are illegal

# Convolution Layers: Dimensions



Shrinking down so quickly ( $32 \rightarrow 28 \rightarrow 24 \rightarrow 20$ ) is typically not a good idea...

# Convolution Layers: Padding



Why padding?

- Sizes get small too quickly
- Corner pixel is only used once

# Convolution Layers: Padding

Image  $7 \times 7$  + zero padding

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Why padding?

- Sizes get small too quickly
- Corner pixel is only used once

# Convolution Layers: Padding

Image  $7 \times 7$  + zero padding

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input ( $N \times N$ ):  $7 \times 7$

Filter ( $F \times F$ ):  $3 \times 3$

Padding ( $P$ ): 1

Stride ( $S$ ): 1

Output  $7 \times 7$  

Most common is 'zero' padding

Output Size:

$$\left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right)$$

$\lfloor \cdot \rfloor$  denotes the floor operator (as in practice an integer division is performed)

# Convolution Layers: Padding

Image  $7 \times 7$  + zero padding

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Types of convolutions:

- Valid convolution: using no padding
- Same convolution: output = input size

Set padding to  $P = \frac{F-1}{2}$

# Convolution Layers: Dimensions

Example

Input image:  $32 \times 32 \times 3$

10 filters  $5 \times 5$

Stride 1

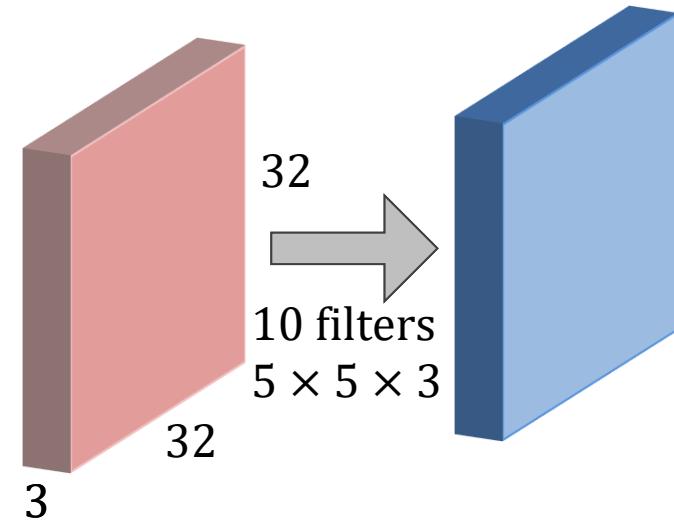
Pad 2

Depth of 3 is implicitly given

Output size is:

$$\frac{32 + 2 \cdot 2 - 5}{1} + 1 = 32$$

i.e.  $32 \times 32 \times 10$



Remember

$$\text{Output: } \left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right)$$

# Convolution Layers: Dimensions

Example

Input image:  $32 \times 32 \times 3$

10 filters  $5 \times 5$

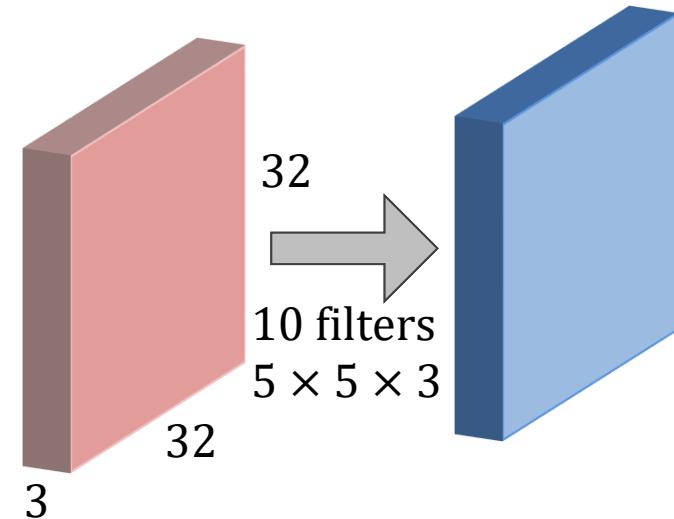
Stride 1

Pad 2

Output size is:

$$\frac{32 + 2 \cdot 2 - 5}{1} + 1 = 32$$

i.e.  $32 \times 32 \times 10$



Remember

$$\text{Output: } \left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{N+2 \cdot P - F}{S} \right\rfloor + 1\right)$$

# Convolution Layers: Dimensions

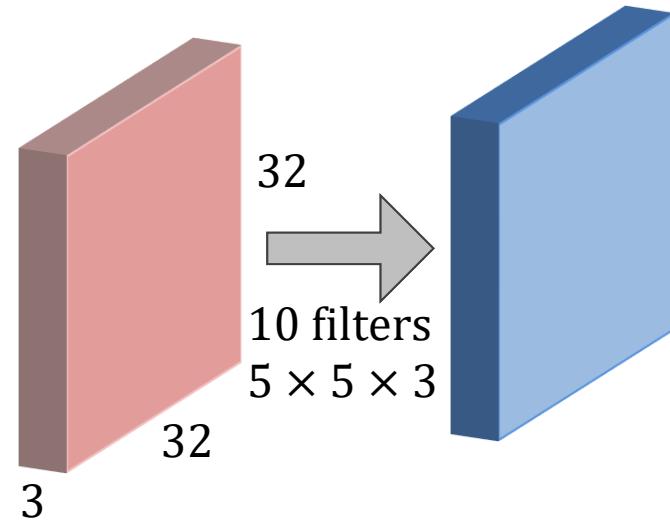
Example

Input image:  $32 \times 32 \times 3$

10 filters  $5 \times 5$

Stride 1

Pad 2



Number of parameters (weights):

Each filter has  $5 \times 5 \times 3 + 1 = 76$  params    (+1 for bias)  
→  $76 \cdot 10 = 760$  parameters in layer

# Example

- You are given a convolutional layer with **4** filters, kernel size **5**, stride **1**, and no padding that operates on an RGB image.
- Q1: What are the dimensions and the shape of its weight tensor?
  - ❑ A1: **(3, 4, 5, 5)**
  - ❑ A2: **(4, 5, 5)**
  - ❑ A3: depends on the width and height of the image

# Example

- You are given a convolutional layer with **4** filters, kernel size **5**, stride **1**, and no padding that operates on an RGB image.
- Q1: What are the dimensions and the shape of its weight tensor?

A1:  $(3, 4, 5, 5)$

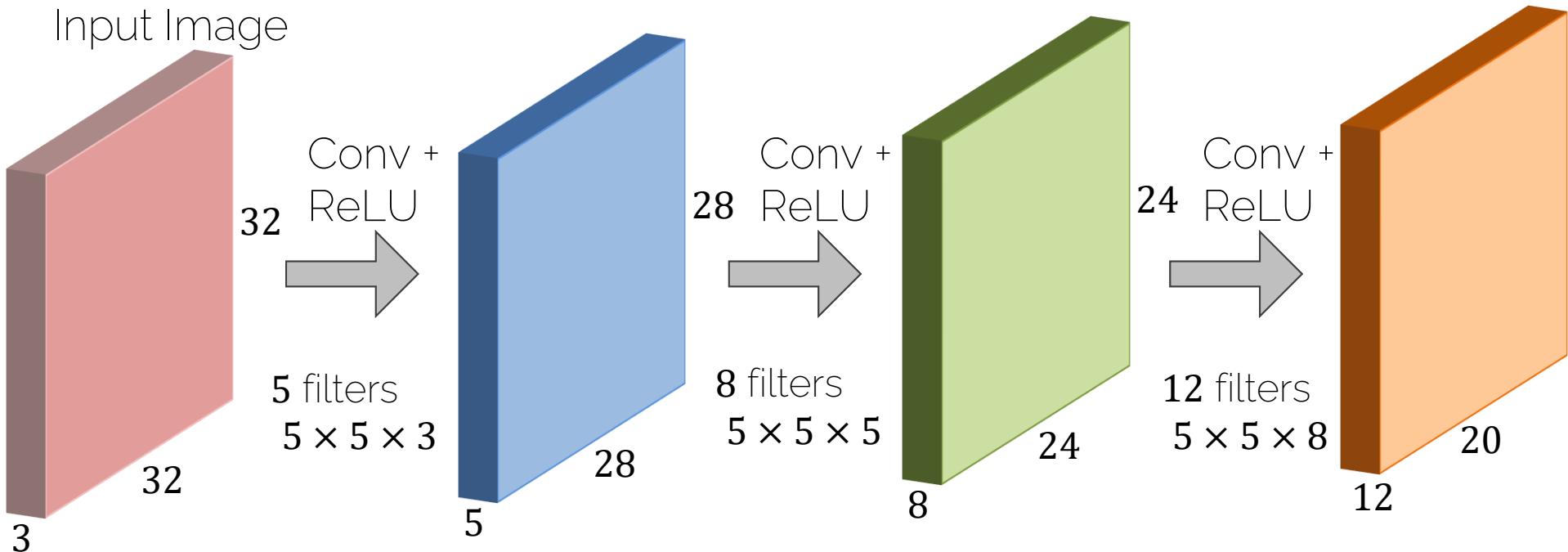
Input channels (RGB = 3)      Filter size =  $5 \times 5$

Output size =  
4 filters

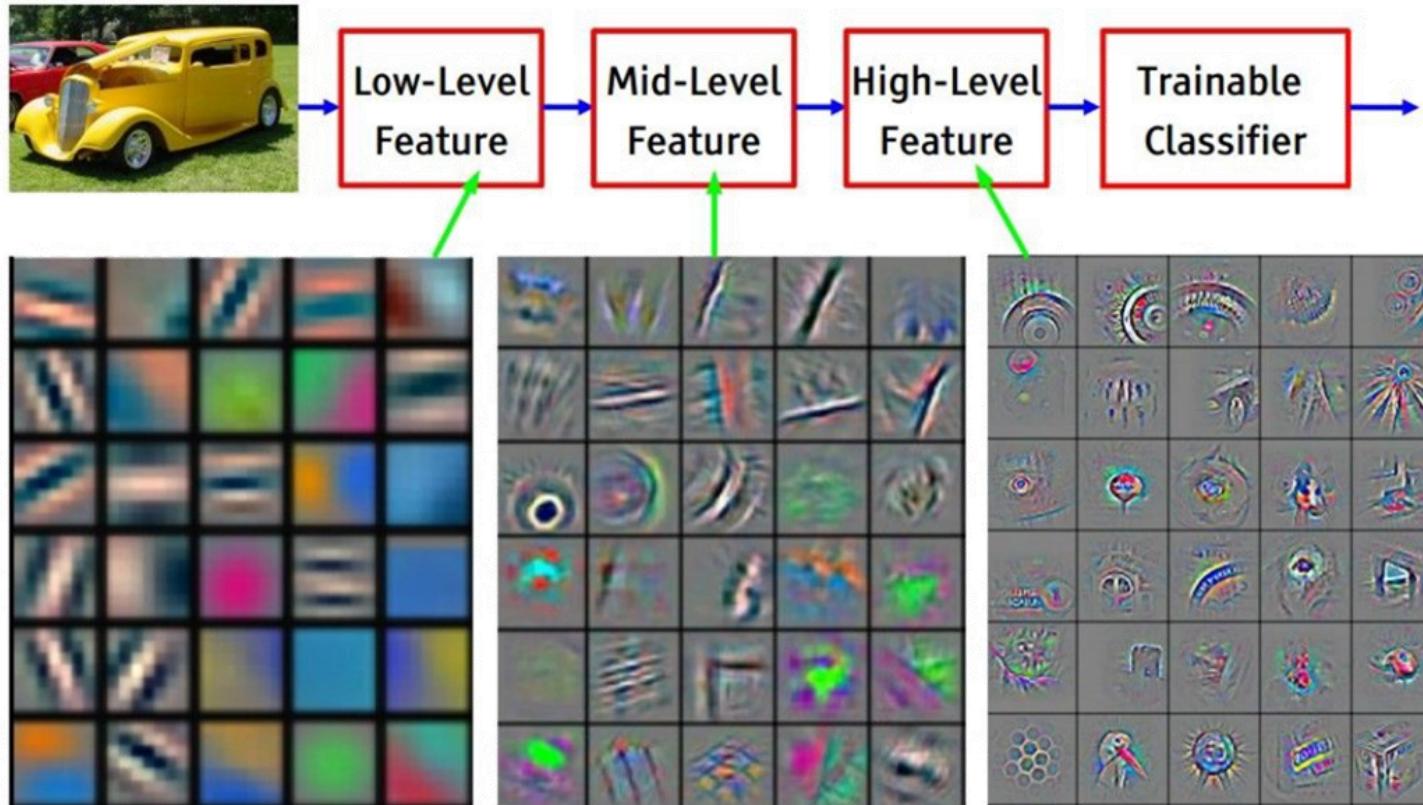
# Convolutional Neural Network (CNN)

# CNN Prototype

ConvNet is concatenation of Conv Layers and activations



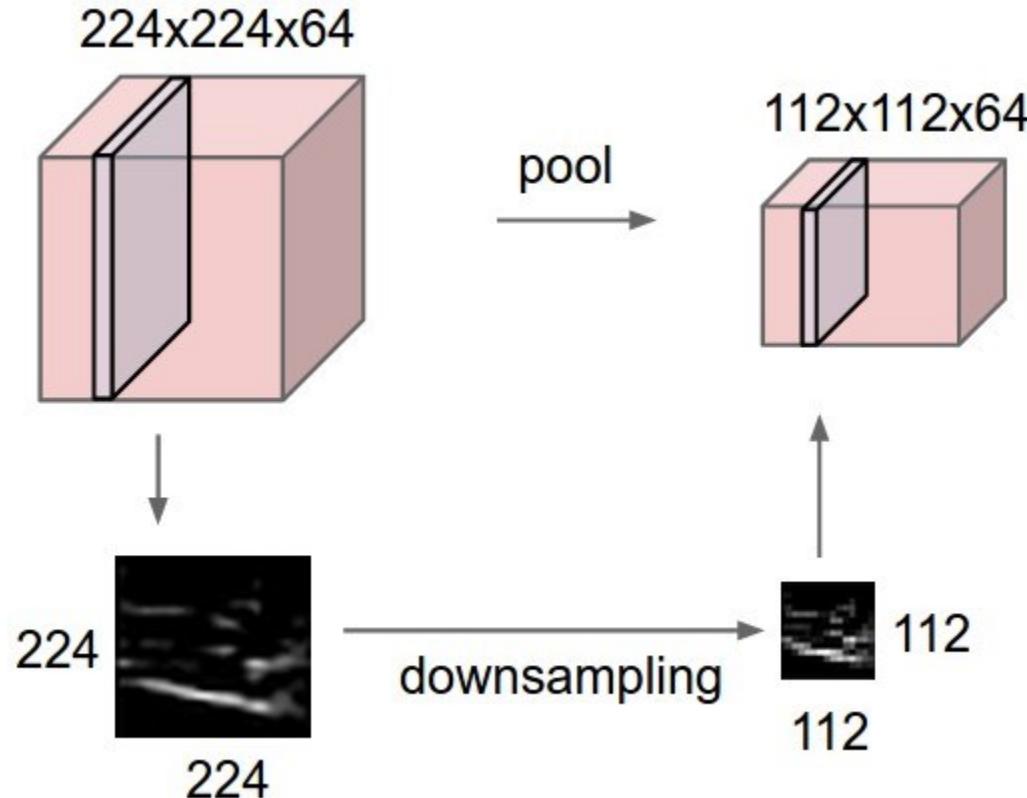
# CNN Learned Filters



[Zeiler & Fergus, ECCV'14] Visualizing and Understanding Convolutional Networks

# Pooling

# Pooling Layer

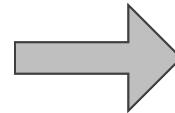


# Pooling Layer: Max Pooling

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

Max pool with  
 $2 \times 2$  filters and stride 2



'Pooled' output

6	9
3	4

# Pooling Layer

- Conv Layer = 'Feature Extraction'
  - Computes a feature in a given region
- Pooling Layer = 'Feature Selection'
  - Picks the strongest activation in a region

# Pooling Layer

- Input is a volume of size  $W_{in} \times H_{in} \times D_{in}$
- Two hyperparameters
  - Spatial filter extent  $F$
  - Stride  $S$
- Output volume is of size  $W_{out} \times H_{out} \times D_{out}$ 
  - $W_{out} = \frac{W_{in}-F}{S} + 1$
  - $H_{out} = \frac{H_{in}-F}{S} + 1$
  - $D_{out} = D_{in}$
- Does not contain parameters; e.g. it's fixed function

# Pooling Layer

- Input is a volume of size  $W_{in} \times H_{in} \times D_{in}$
- Two hyperparameters
  - Spatial filter extent  $F$
  - Stride  $S$
- Output volume is of size  $W_{out} \times H_{out} \times D_{out}$ 
  - $W_{out} = \frac{W_{in}-F}{S} + 1$
  - $H_{out} = \frac{H_{in}-F}{S} + 1$
  - $D_{out} = D_{in}$
- Does not contain parameters; e.g. it's fixed function

Common settings:

$$F = 2, S = 2$$

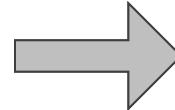
$$F = 3, S = 2$$

# Pooling Layer: Average Pooling

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

Average pool with  
 $2 \times 2$  filters and stride 2

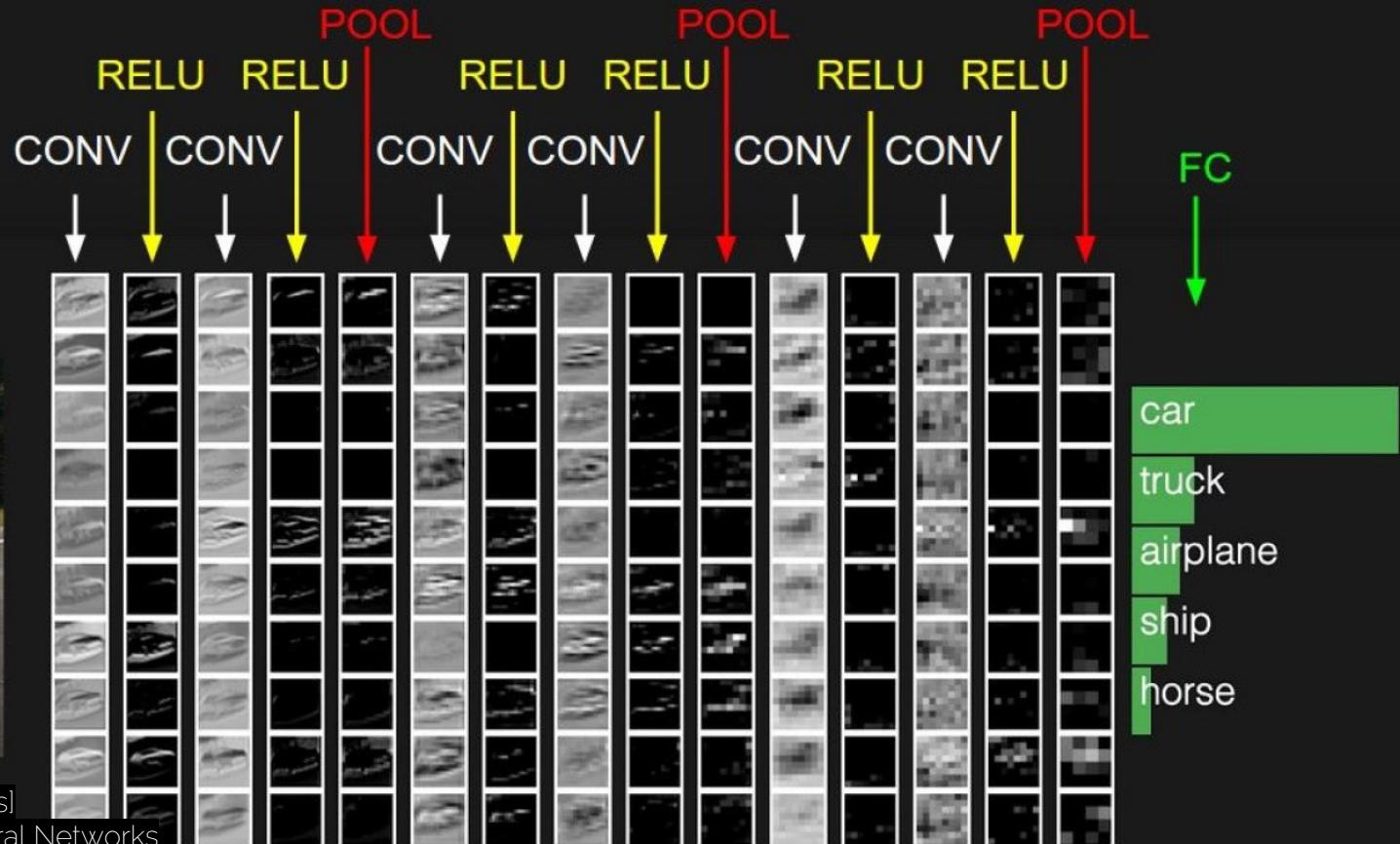


'Pooled' output

2.5	6
1.75	3

- Typically used deeper in the network

# CNN Prototype



[Li et al., CS231n Course Slides]  
Lecture 5: Convolutional Neural Networks

# Final Fully-Connected Layer

- Same as what we had in ‘ordinary’ neural networks
  - Make the final decision with the extracted features from the convolutions
  - One or two FC layers typically

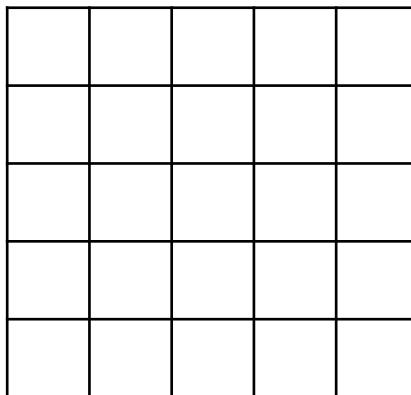
# Convolutions vs Fully-Connected

- In contrast to fully-connected layers, we want to restrict the degrees of freedom
  - FC is somewhat brute force
  - Convolutions are **structured**
- Sliding window to with the same filter parameters to extract image features
  - Concept of weight sharing
  - Extract same features independent of location

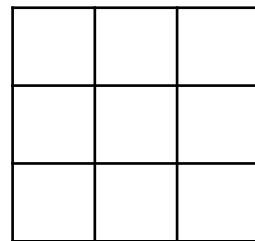
# Receptive field

# Receptive Field

- Spatial extent of the connectivity of a convolutional filter

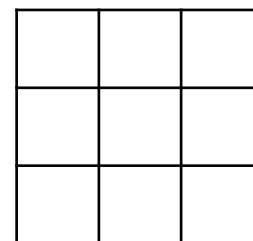


5x5 input



3x3 filter

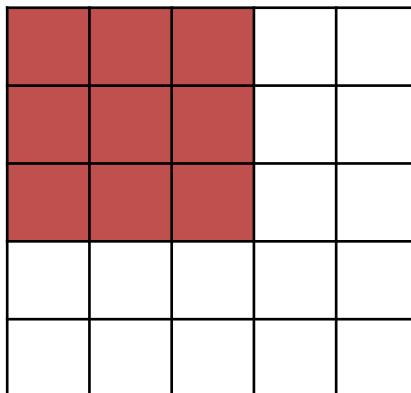
=



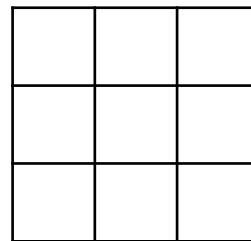
3x3 output

# Receptive Field

- Spatial extent of the connectivity of a convolutional filter

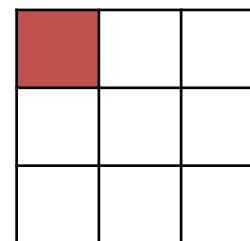


5x5 input



3x3 filter

=

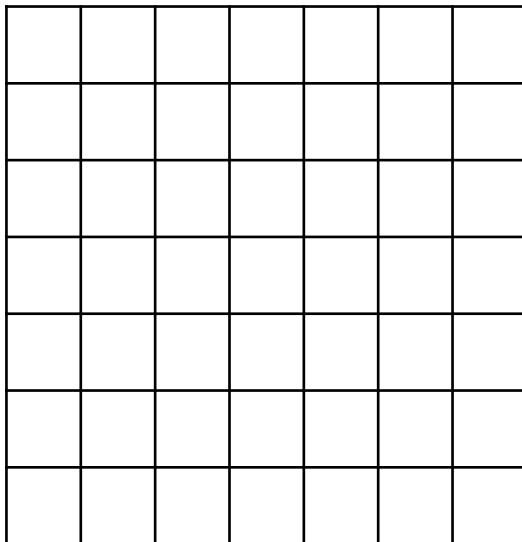


3x3 output

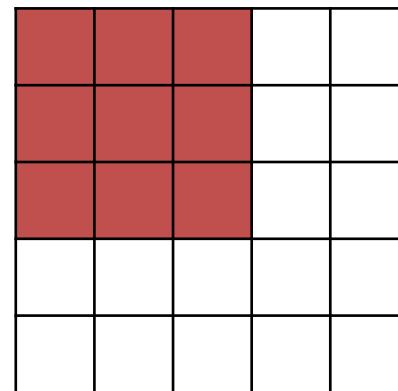
3x3 receptive field = 1 output pixel is connected to 9 input pixels

# Receptive Field

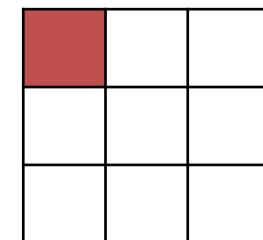
- Spatial extent of the connectivity of a convolutional filter



7x7 input



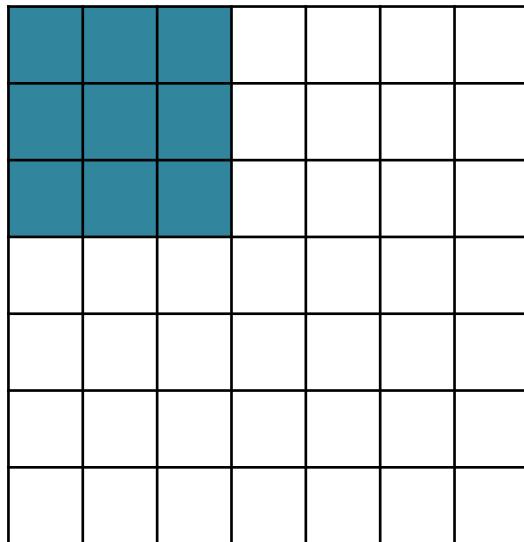
3x3 output



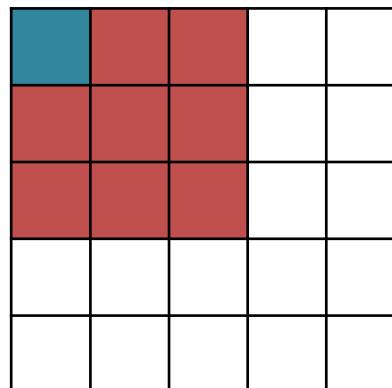
3x3 receptive field = 1 output pixel is connected to 9 input pixels

# Receptive Field

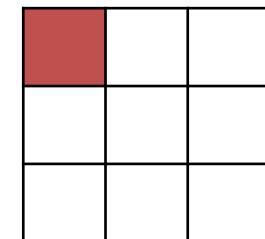
- Spatial extent of the connectivity of a convolutional filter



7x7 input



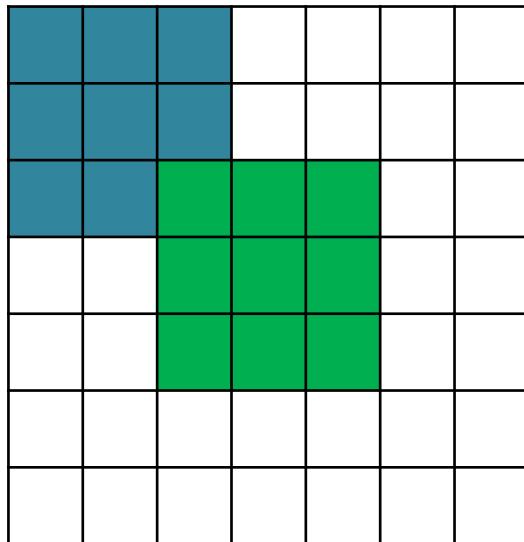
3x3 output



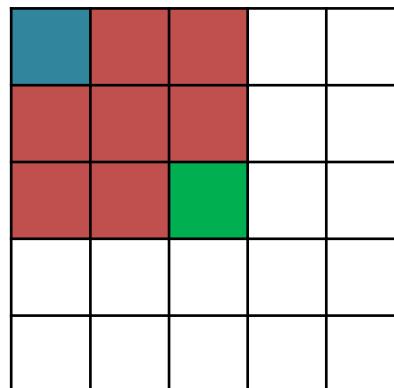
3x3 receptive field = 1 output pixel is connected to 9 input pixels

# Receptive Field

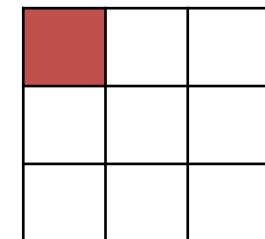
- Spatial extent of the connectivity of a convolutional filter



7x7 input



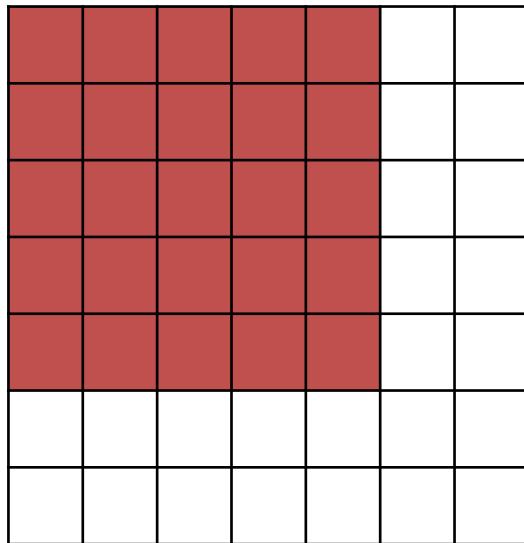
3x3 output



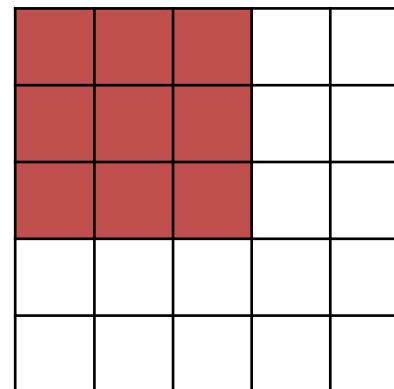
3x3 receptive field - 1 output pixel is connected to 9 input pixels

# Receptive Field

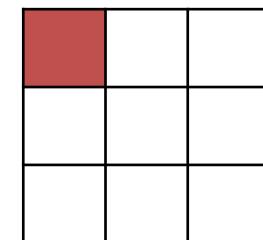
- Spatial extent of the connectivity of a convolutional filter



7x7 input



3x3 output



5x5 receptive field on the original input:  
one output value is connected to 25 input pixels

See you next time!

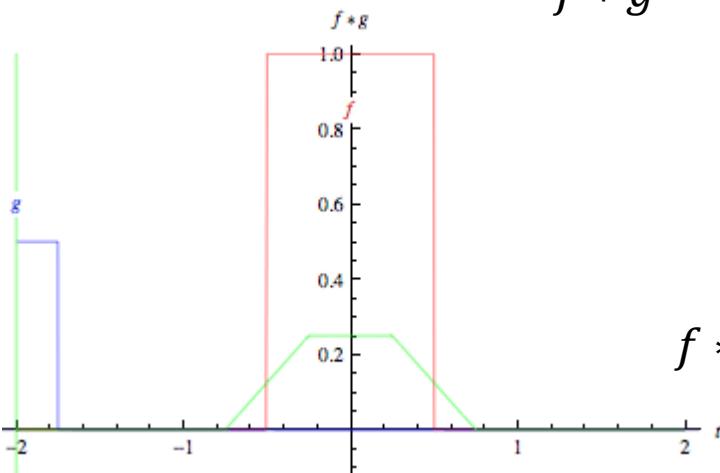
# References

- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 9: Convolutional Networks
- <http://cs231n.github.io/convolutional-networks/>

# Lecture 9 Recap

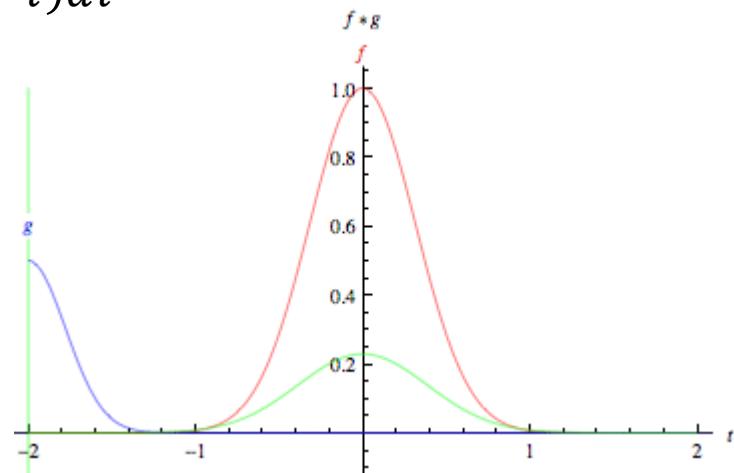
# What are Convolutions?

$$f * g = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$



Convolution of two box functions

$f$  = red  
 $g$  = blue  
 $f * g$  = green



Convolution of two Gaussians

application of a filter to a function  
the 'smaller' one is typically called the filter kernel

# What are Convolutions?

Discrete case: box filter

4	3	2	-5	3	5	2	5	5	6
---	---	---	----	---	---	---	---	---	---

1/3	1/3	1/3
-----	-----	-----

??	3	0	0	1	10/3	4	4	16/3	??
----	---	---	---	---	------	---	---	------	----

*What to do at boundaries?*

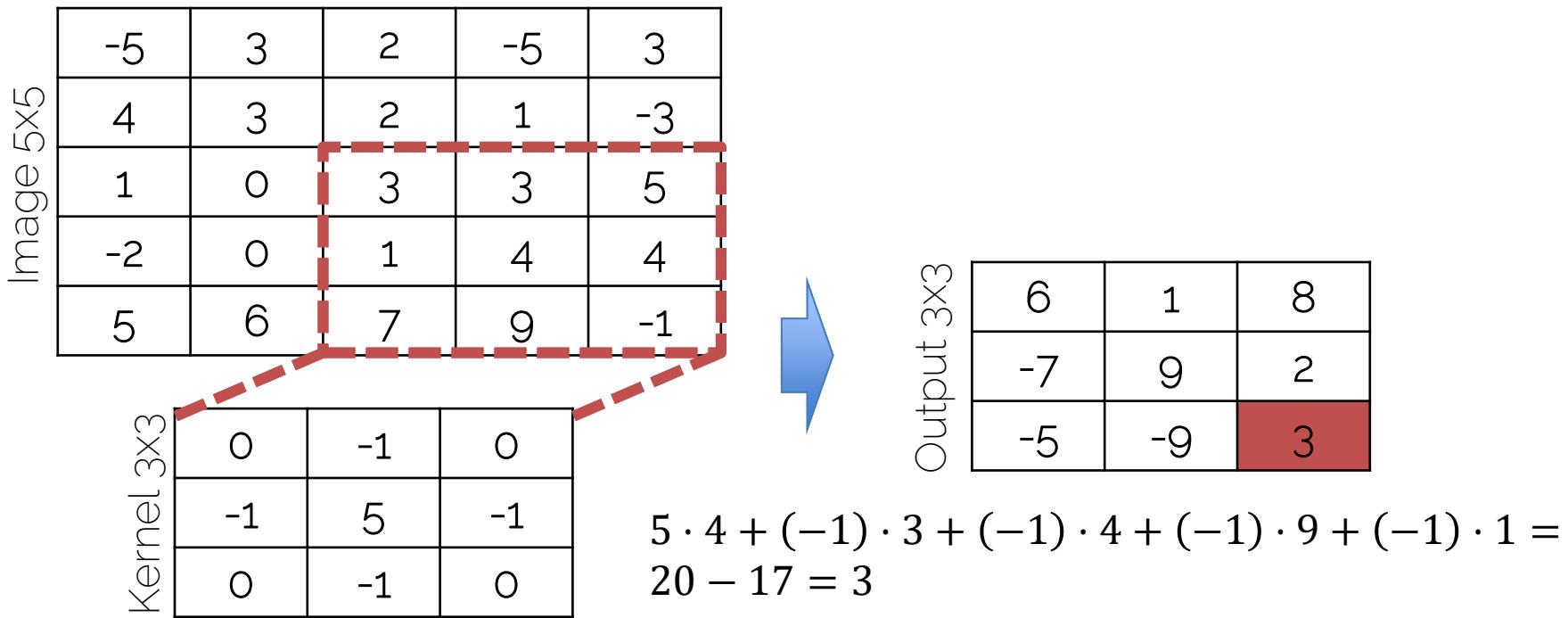
1) Shrink

3	0	0	1	10/3	4	4	16/3
---	---	---	---	------	---	---	------

2) Pad  
often '0'

7/3	3	0	0	1	10/3	4	4	16/3	11/3
-----	---	---	---	---	------	---	---	------	------

# Convolutions on Images



# Image Filters

- Each kernel gives us a different image filter

Input



Edge detection

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



Box mean

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



Sharpen

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

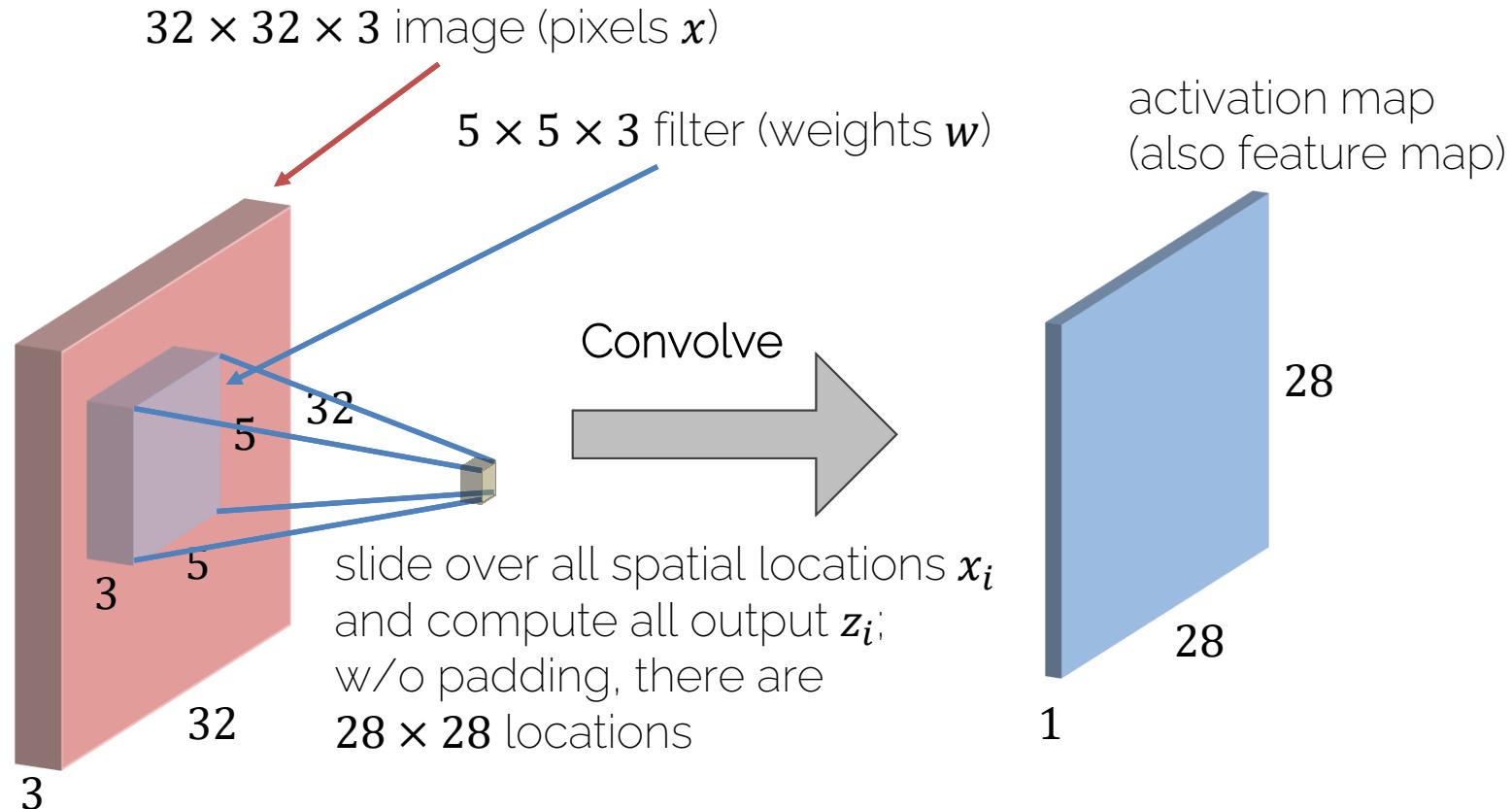


Gaussian blur

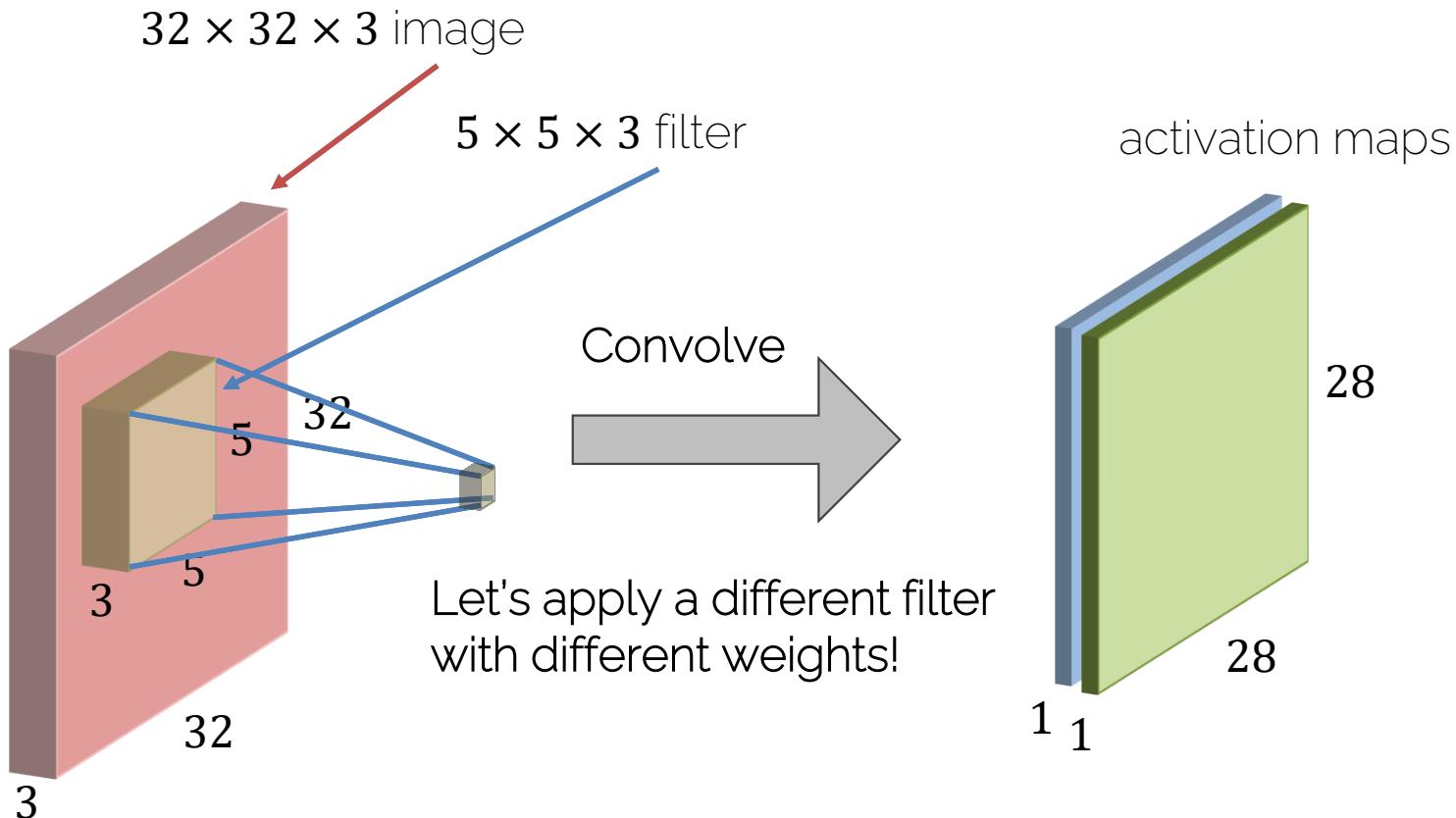
$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

LET'S LEARN THESE FILTERS!

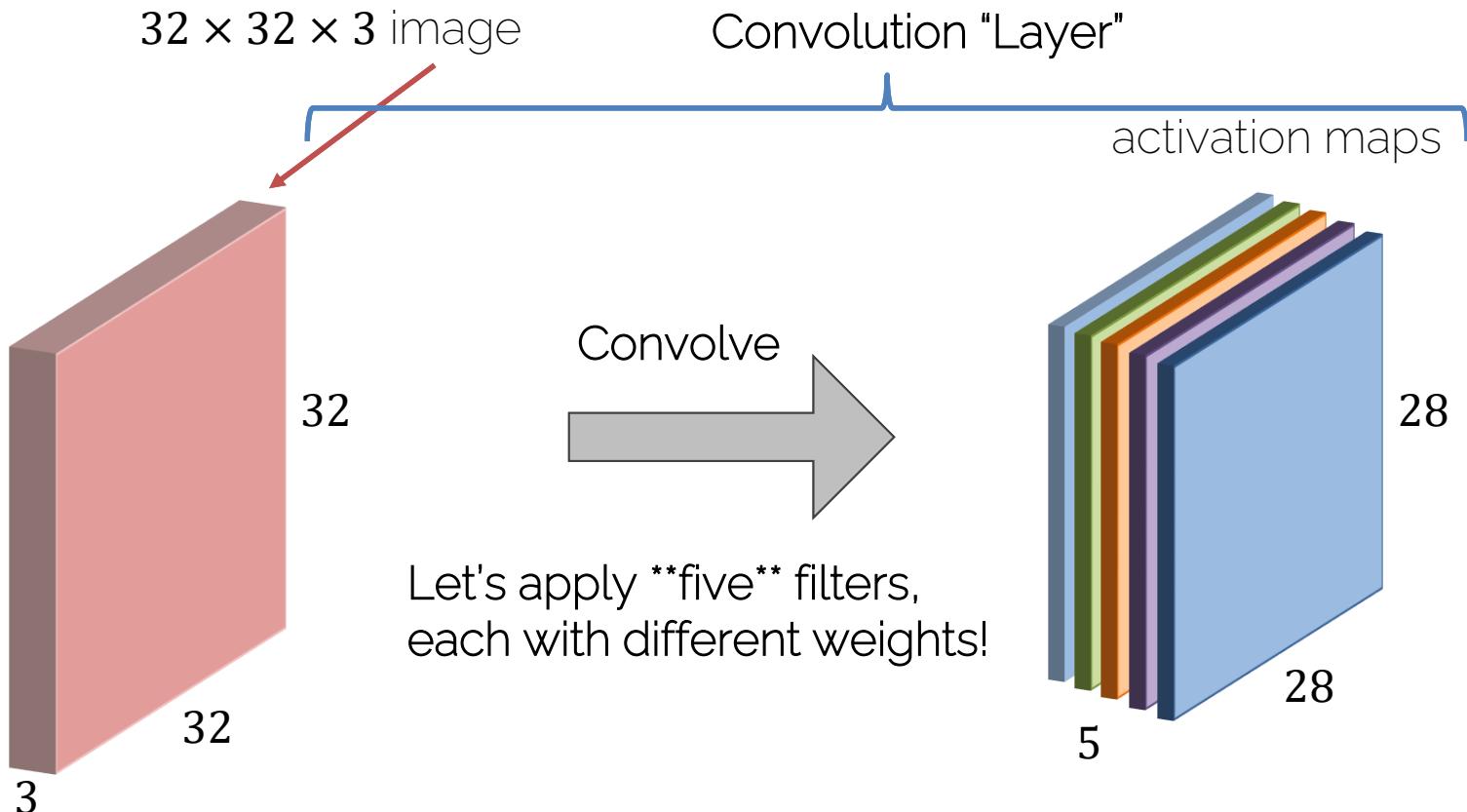
# Convolutions on RGB Images



# Convolution Layer

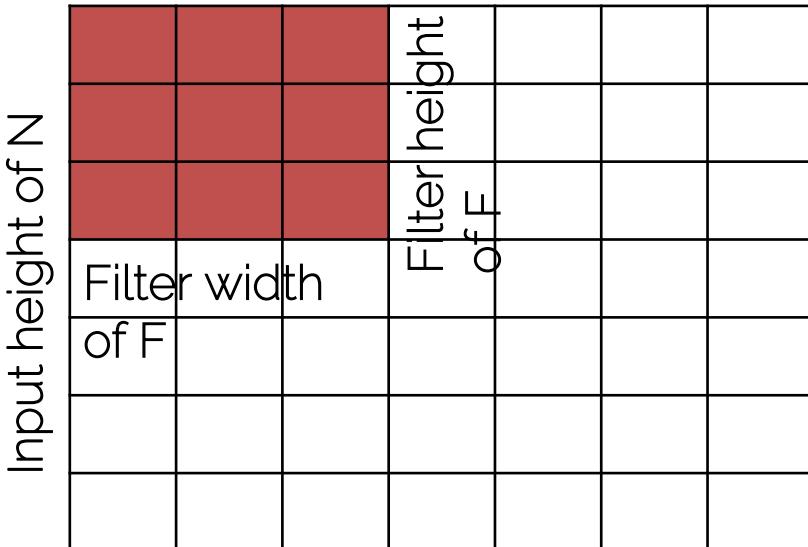


# Convolution Layer



# Convolution Layers: Dimensions

Input width of  $N$



Input:  $N \times N$

Filter:  $F \times F$

Stride:  $S$

Output:  $(\frac{N-F}{S} + 1) \times (\frac{N-F}{S} + 1)$

$$N = 7, F = 3, S = 1: \quad \frac{7-3}{1} + 1 = 5$$

$$N = 7, F = 3, S = 2: \quad \frac{7-3}{2} + 1 = 3$$

$$N = 7, F = 3, S = 3: \quad \frac{7-3}{3} + 1 = 2.3333$$



Fractions are illegal

# Convolution Layers: Padding

Image 7x7 + zero padding

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Types of convolutions:

- **Valid convolution:** using no padding
- **Same convolution:** output=input size

$$\text{Set padding to } P = \frac{F-1}{2}$$

# Convolution Layers: Dimensions

Remember: Output =  $\left(\frac{N+2\cdot P-F}{S} + 1\right) \times \left(\frac{N+2\cdot P-F}{S} + 1\right)$

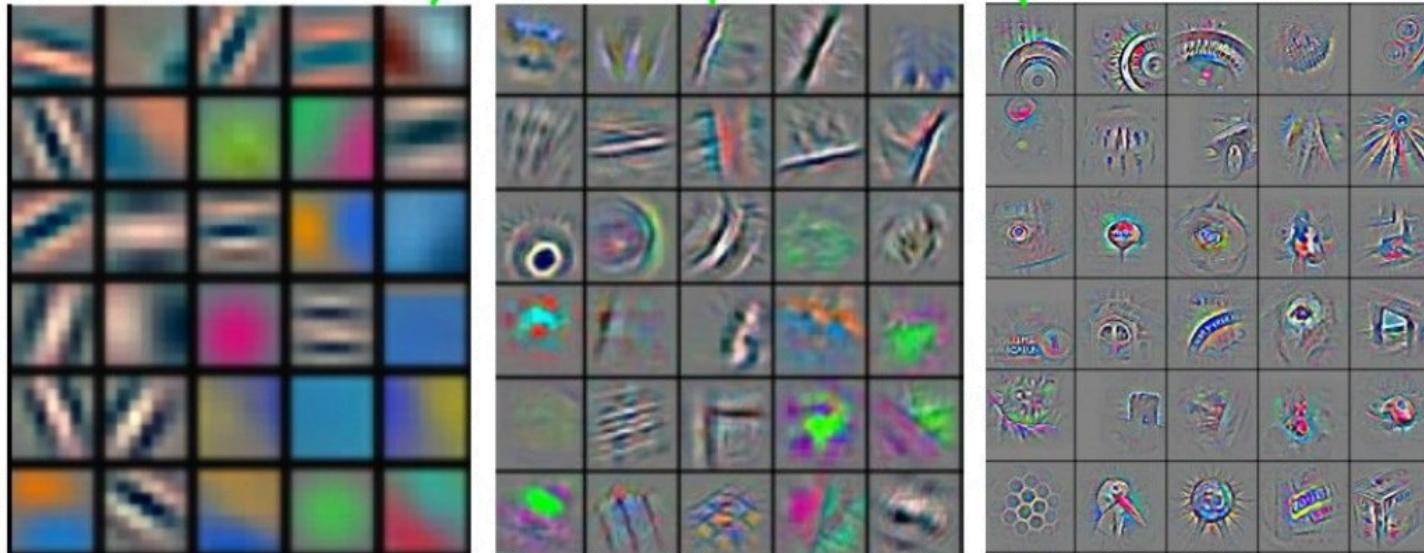


**REMARK:** in practice, typically **integer division** is used  
(i.e., apply the **floor-operator!**)

*Example: 3x3 conv with same padding and strides of 2  
on an 64x64 RGB image -> N = 64, F = 3, P = 1, S = 2*

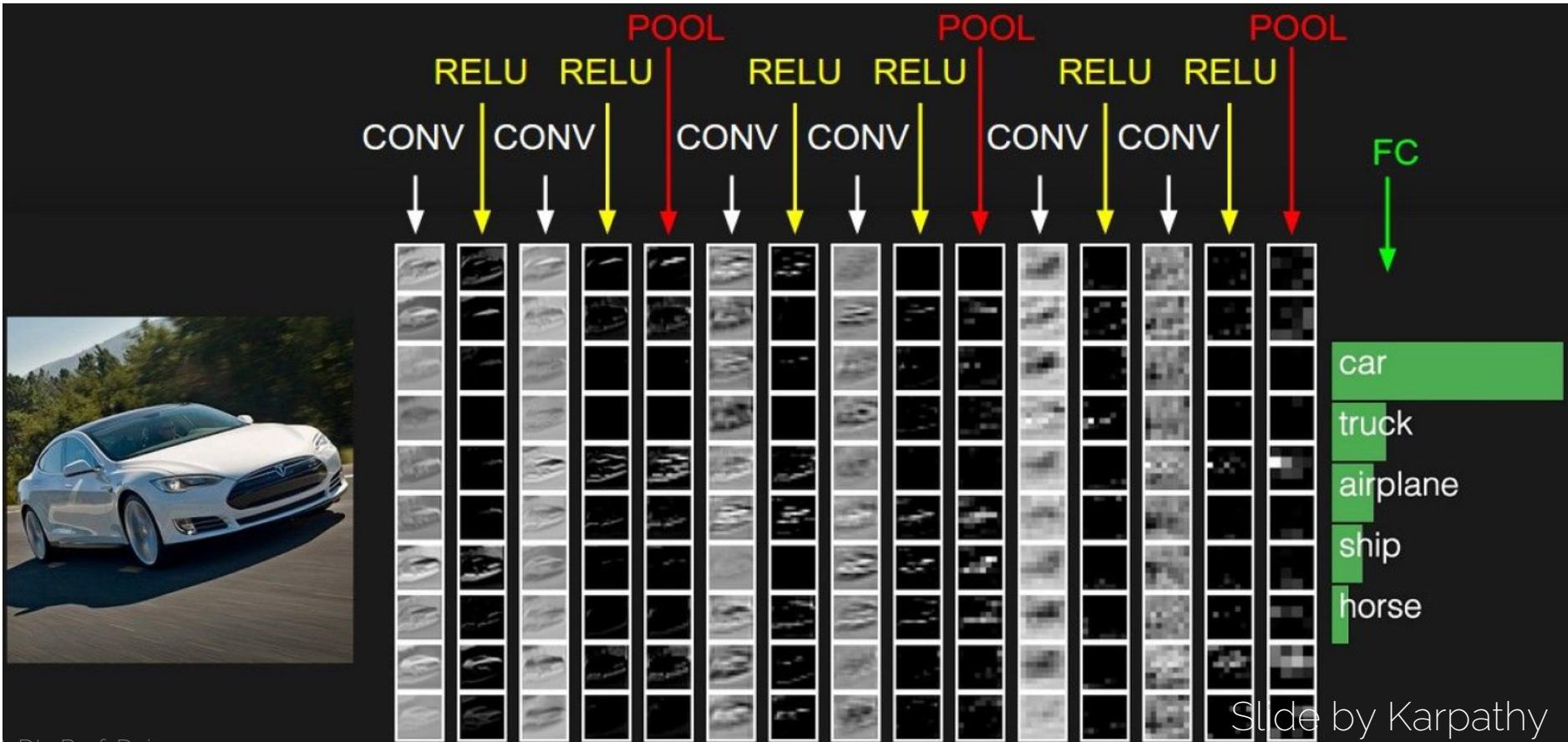
$$\begin{aligned}\text{Output: } & \left(\frac{64+2\cdot 1-3}{2} + 1\right) \times \left(\frac{64+2\cdot 1-3}{2} + 1\right) \\ &= \textcolor{brown}{floor}(32.5) \times \textcolor{brown}{floor}(32.5) \\ &= 32 \times 32\end{aligned}$$

# CNN Learned Filters



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

# CNN Prototype

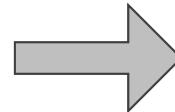


# Pooling Layer: Max Pooling

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

Max pool with  
 $2 \times 2$  filters and stride 2

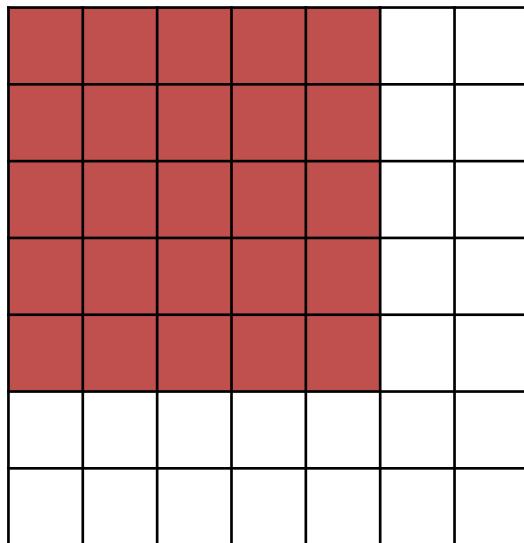


'Pooled' output

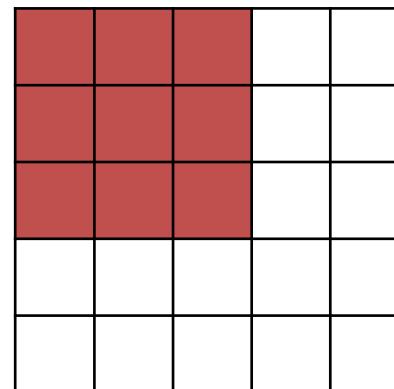
6	9
3	4

# Receptive Field

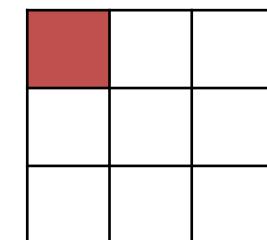
- Spatial extent of the connectivity of a convolutional filter



7x7 input



3x3 output



5x5 receptive field on the original input:  
one output value is connected to 25 input pixels

# Lecture 10 – CNNs (part 2)

# Classic Architectures

# LeNet

- Digit recognition: 10 classes



$32 \times 32 \times 1$

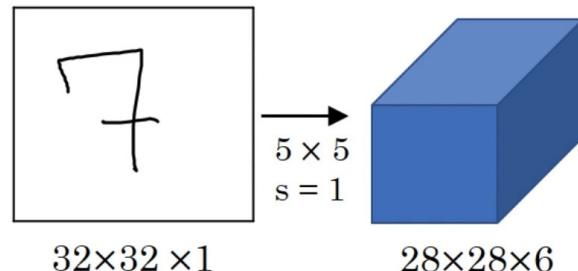
Input:  $32 \times 32$  grayscale images

*This one: Labeled as class "7"*

[LeCun et al. '98] LeNet

# LeNet

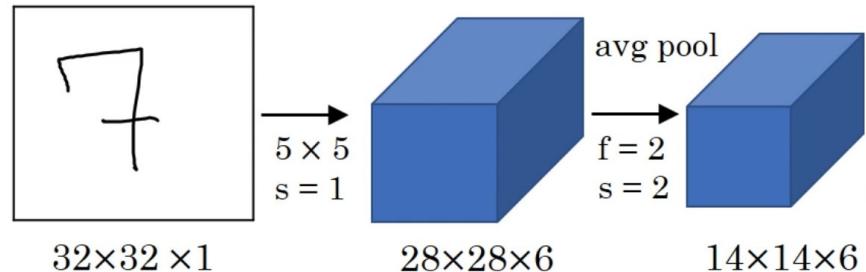
- Digit recognition: 10 classes



- Valid convolution: size shrinks
- How many conv filters are there in the first layer? 6

# LeNet

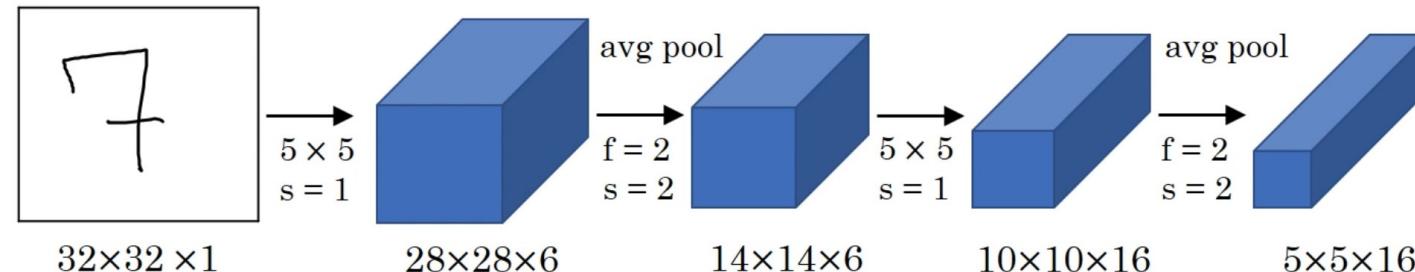
- Digit recognition: 10 classes



- At that time average pooling was used, now max pooling is much more common

# LeNet

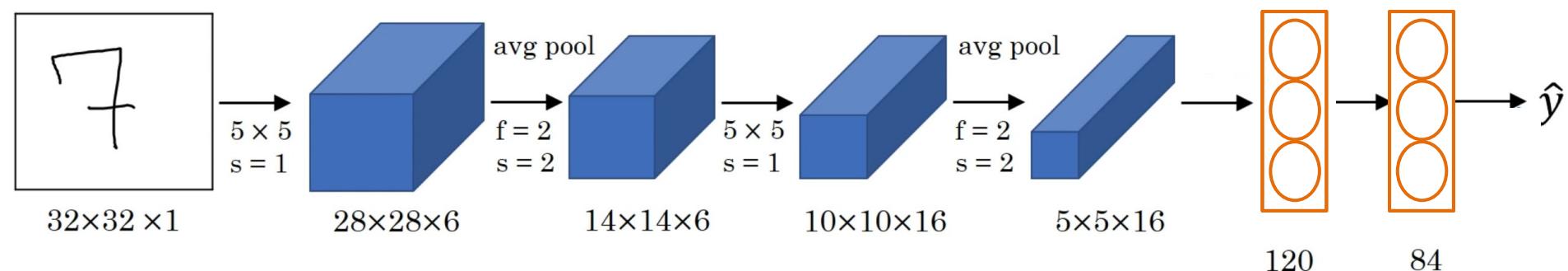
- Digit recognition: 10 classes



- Again valid convolutions, how many filters?

# LeNet

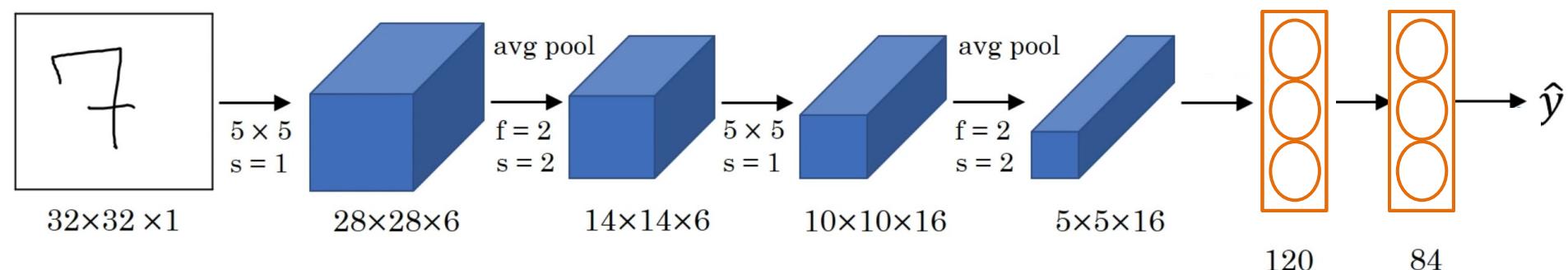
- Digit recognition: 10 classes



- Use of tanh/sigmoid activations → not common now!

# LeNet

- Digit recognition: 10 classes

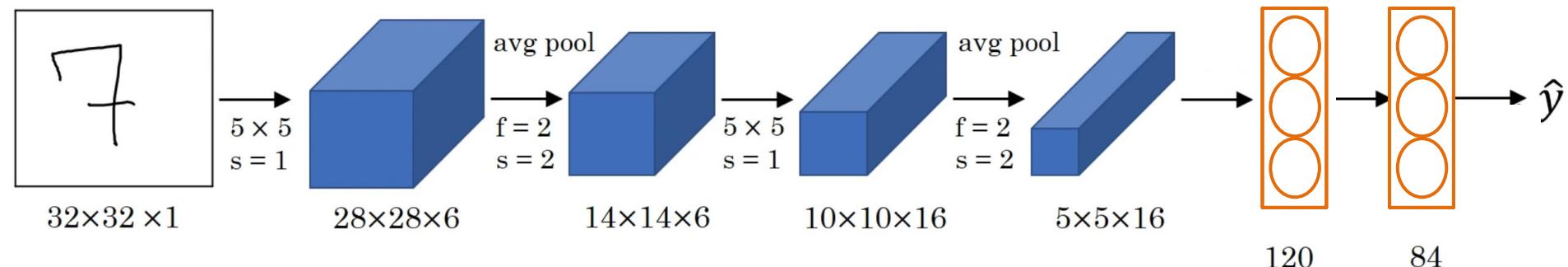


- Conv -> Pool -> Conv -> Pool -> Conv -> FC

# LeNet

- Digit recognition: 10 classes

60k parameters

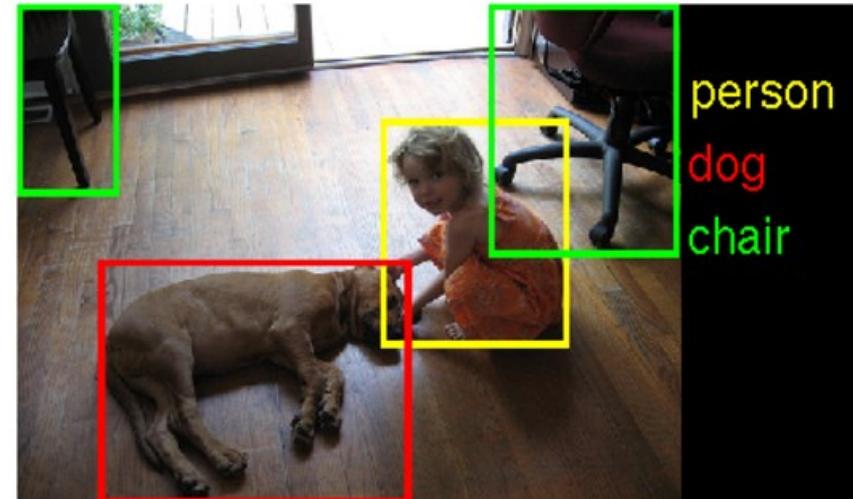
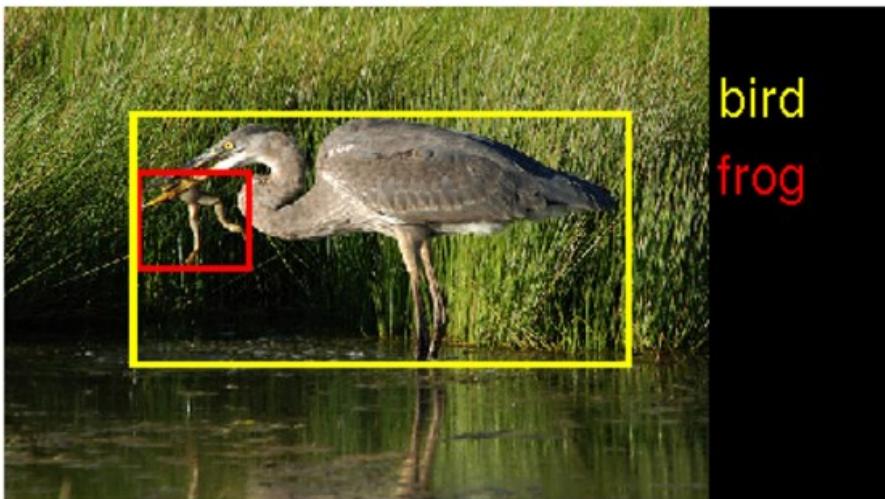


- Conv -> Pool -> Conv -> Pool -> Conv -> FC
- As we go deeper: Width, Height  $\downarrow$  Number of Filters  $\uparrow$

# Test Benchmarks

- ImageNet Dataset:

*ImageNet Large Scale Visual Recognition Competition (ILSVRC)*



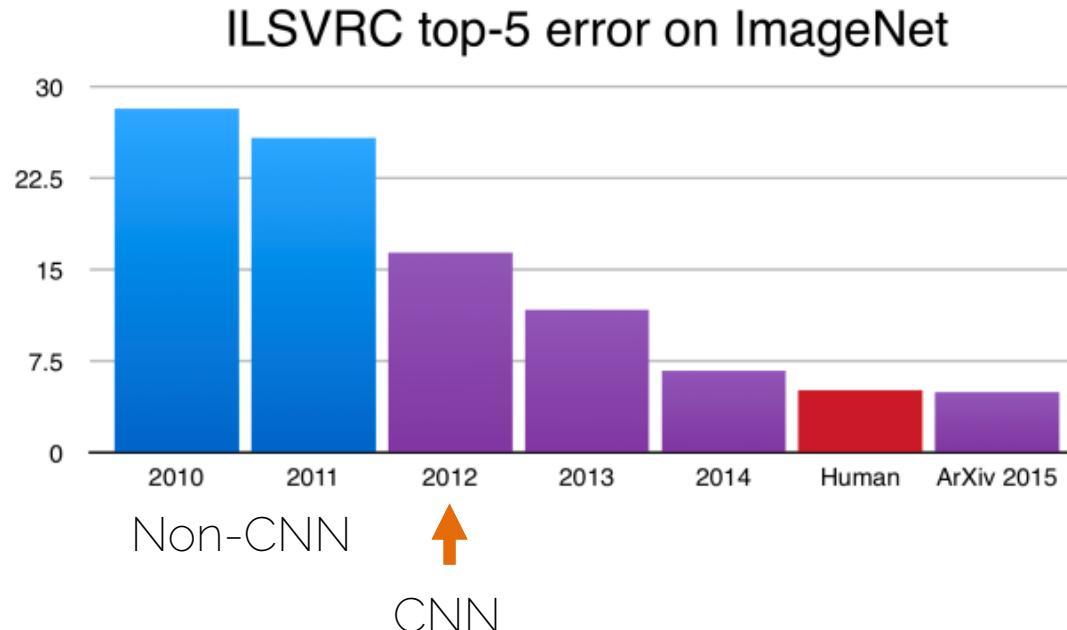
[Russakovsky et al., IJCV'15] "ImageNet Large Scale Visual Recognition Challenge."

# Common Performance Metrics

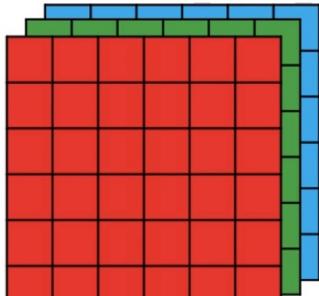
- **Top-1 score:** check if a sample's top class (i.e. the one with highest probability) is the same as its target label
- **Top-5 score:** check if your label is in your 5 first predictions (i.e. predictions with 5 highest probabilities)
- → **Top-5 error:** percentage of test samples for which the correct class was not in the top 5 predicted classes

# AlexNet

- Cut ImageNet error down in half



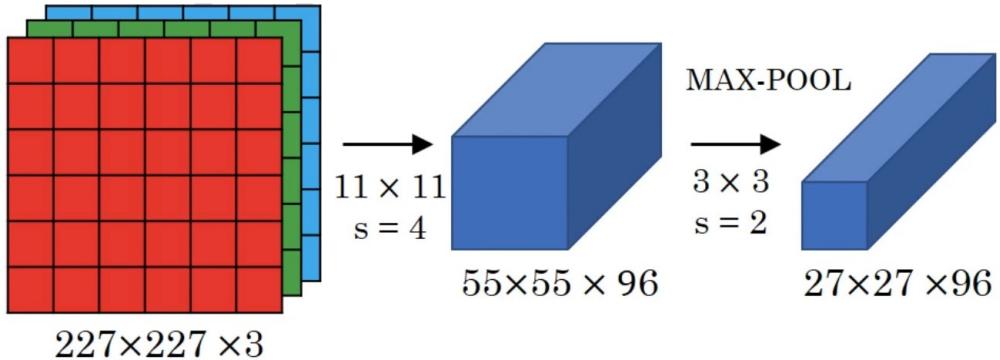
# AlexNet



227×227 ×3

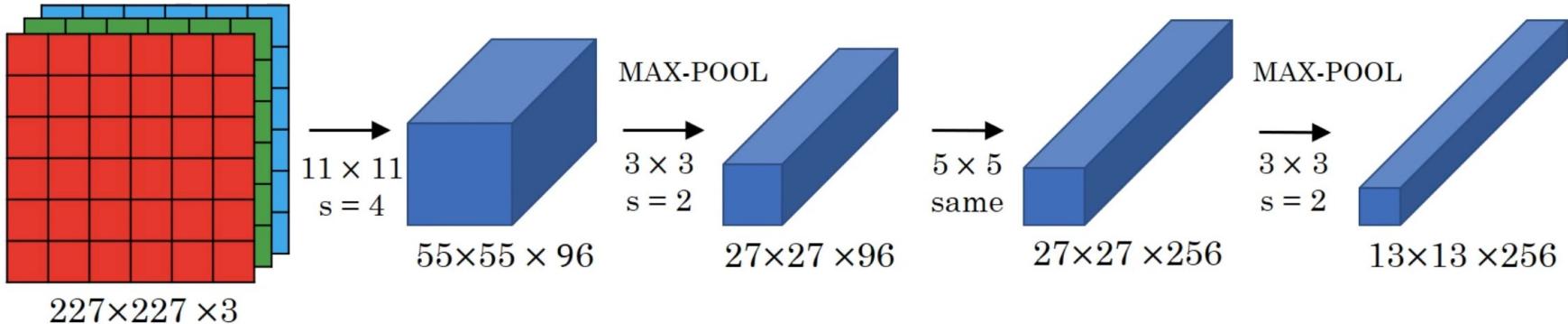
[Krizhevsky et al. NIPS'12] AlexNet

# AlexNet



[Krizhevsky et al. NIPS'12] AlexNet

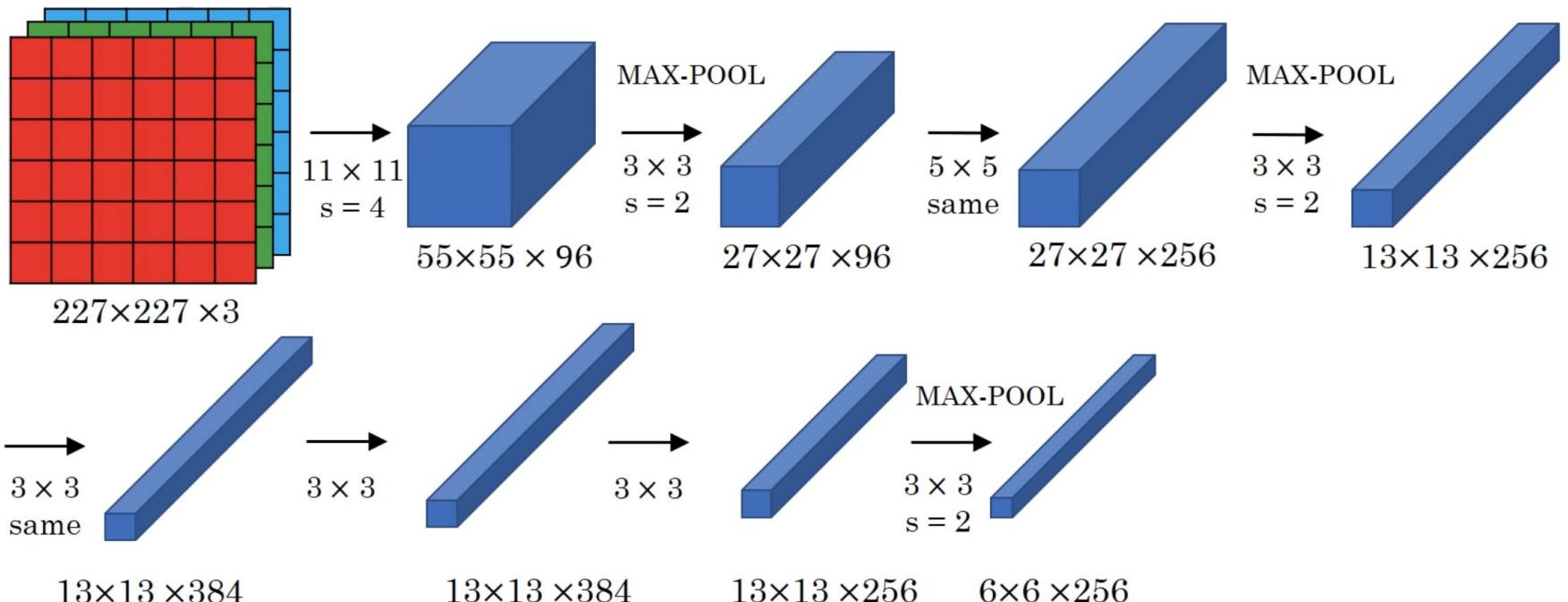
# AlexNet



- Use of same convolutions
- As with LeNet: Width, Height Number of Filters

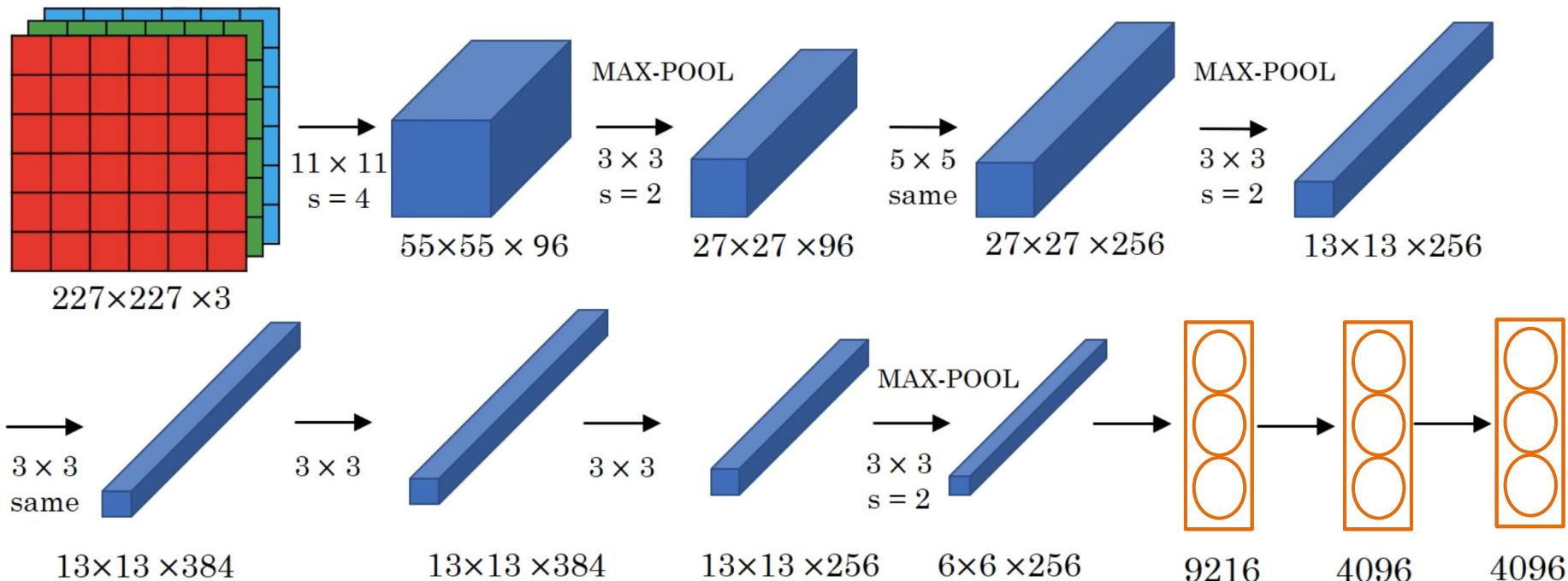
[Krizhevsky et al. NIPS'12] AlexNet

# AlexNet



[Krizhevsky et al. NIPS'12] AlexNet

# AlexNet



- Softmax for 1000 classes

[Krizhevsky et al. NIPS'12] AlexNet

# AlexNet

- Similar to LeNet but much bigger (~1000 times)
- Use of ReLU instead of tanh/sigmoid

60M parameters

[Krizhevsky et al. NIPS'12] AlexNet

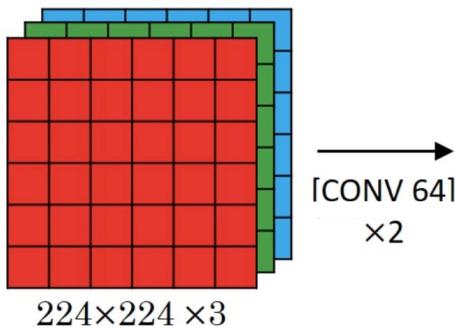
# VGGNet

- Striving for simplicity
- CONV =  $3 \times 3$  filters with stride 1, same convolutions
- MAXPOOL =  $2 \times 2$  filters with stride 2

[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

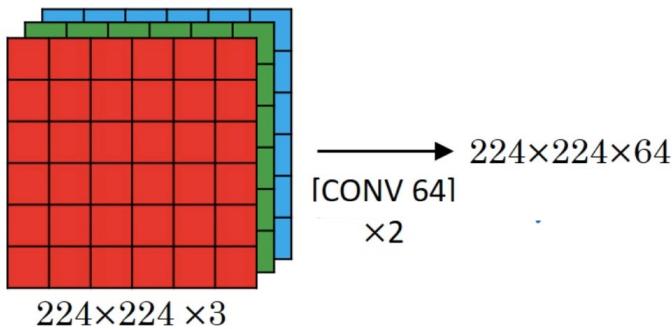
Conv=3x3,s=1,same  
Maxpool=2x2,s=2



[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

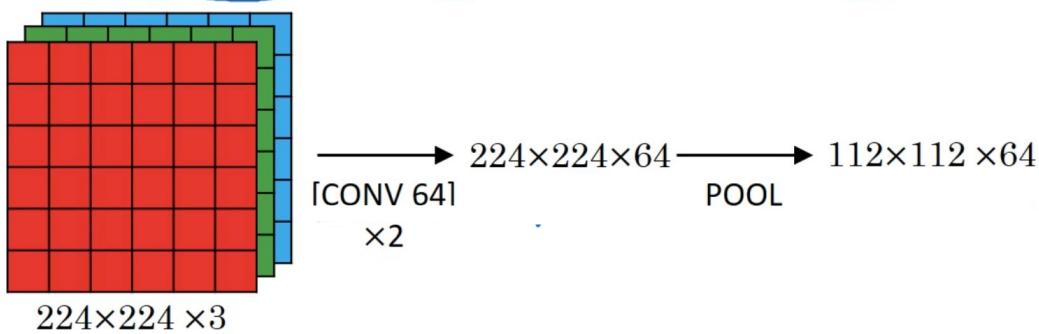
Conv=3x3,s=1,same  
Maxpool=2x2,s=2



[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

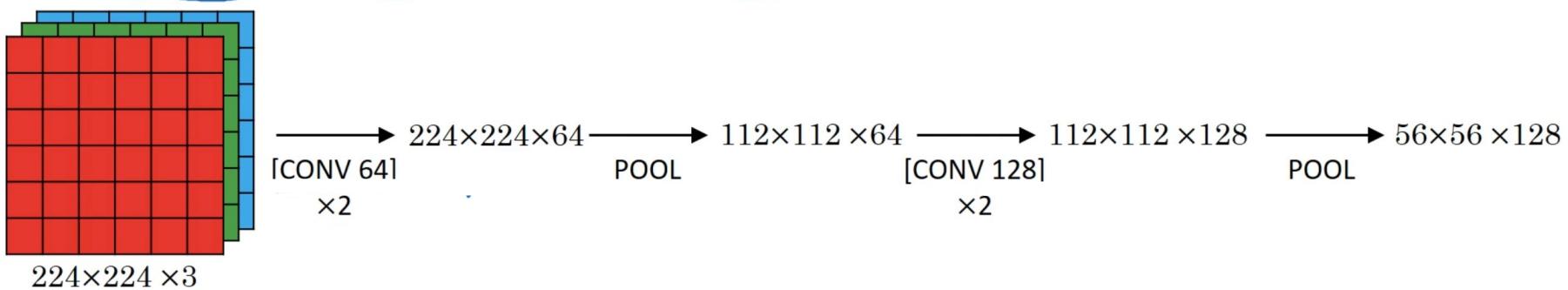
Conv=3x3,s=1,same  
Maxpool=2x2,s=2



[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

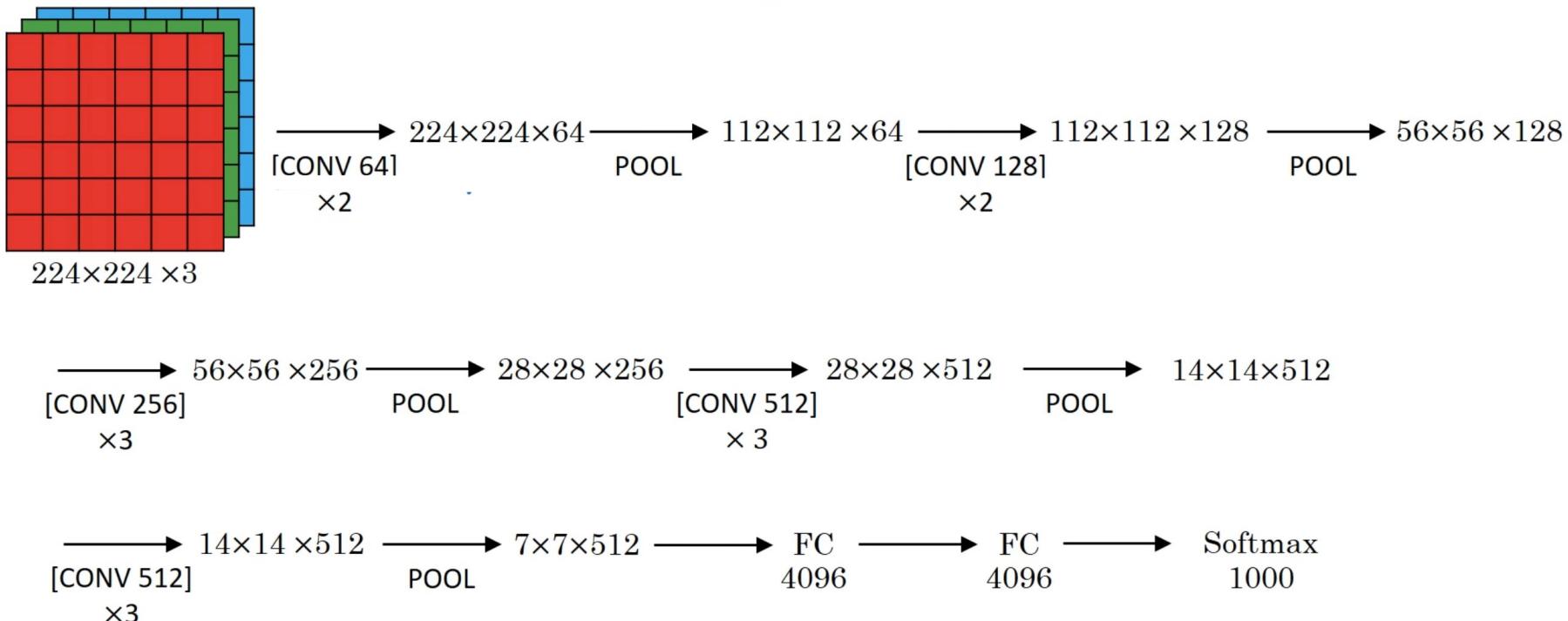
Conv=3x3,s=1,same  
Maxpool=2x2,s=2



[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

Conv=3x3,s=1,same  
Maxpool=2x2,s=2



[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

- Conv -> Pool -> Conv -> Pool -> Conv -> FC
- As we go deeper: Width, Height  Number of Filters 
- Called VGG-16: 16 layers that have weights  

138M parameters
- Large but simplicity makes it appealing

[Simonyan and Zisserman ICLR'15] VGGNet

# VGGNet

- A lot of architectures were analyzed

ConvNet Configuration						
A	A-LRN	B	C	D	E	
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers	
input (224 × 224 RGB image)						
conv3-64	conv3-64 <b>LRN</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	
maxpool						
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 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>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool						
conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool						
conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool						
FC-4096						
FC-4096						
FC-1000						
soft-max						

[Simonyan and Zisserman 2014]

Table 2: Number of parameters (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

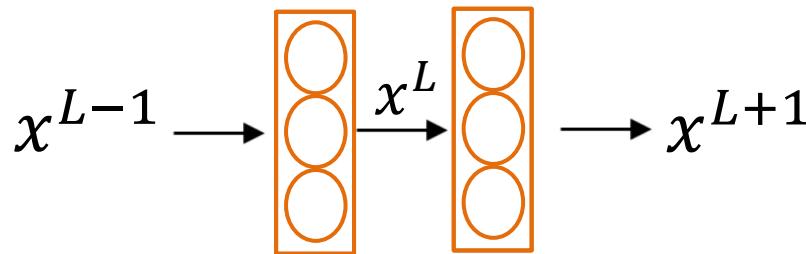
# Skip Connections

# The Problem of Depth

- As we add more and more layers, training becomes harder
- Vanishing and exploding gradients
- How can we train very deep nets?

# Residual Block

- Two layers



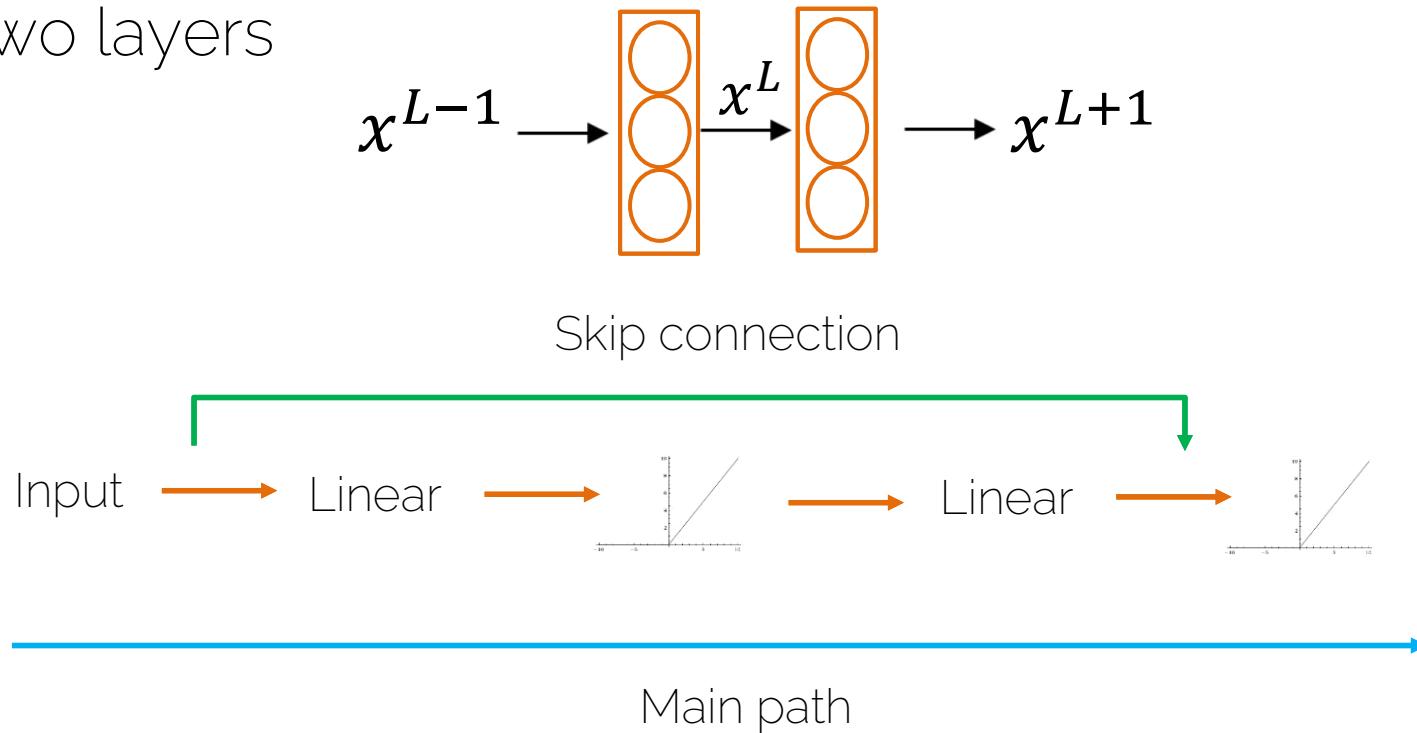
$$\text{Input} \longrightarrow W^L x^{L-1} + b^L \xrightarrow{\text{Linear}} x^L = f(W^L x^{L-1} + b^L) \xrightarrow{\text{Non-linearity}}$$

A graph showing a linear function  $f(x) = x$  plotted on a coordinate system. The x-axis ranges from -10 to 10, and the y-axis ranges from -10 to 10. The function is represented by a straight line passing through the origin (0,0).

$$\longrightarrow x^{L+1} = f(W^{L+1} x^L + b^{L+1})$$

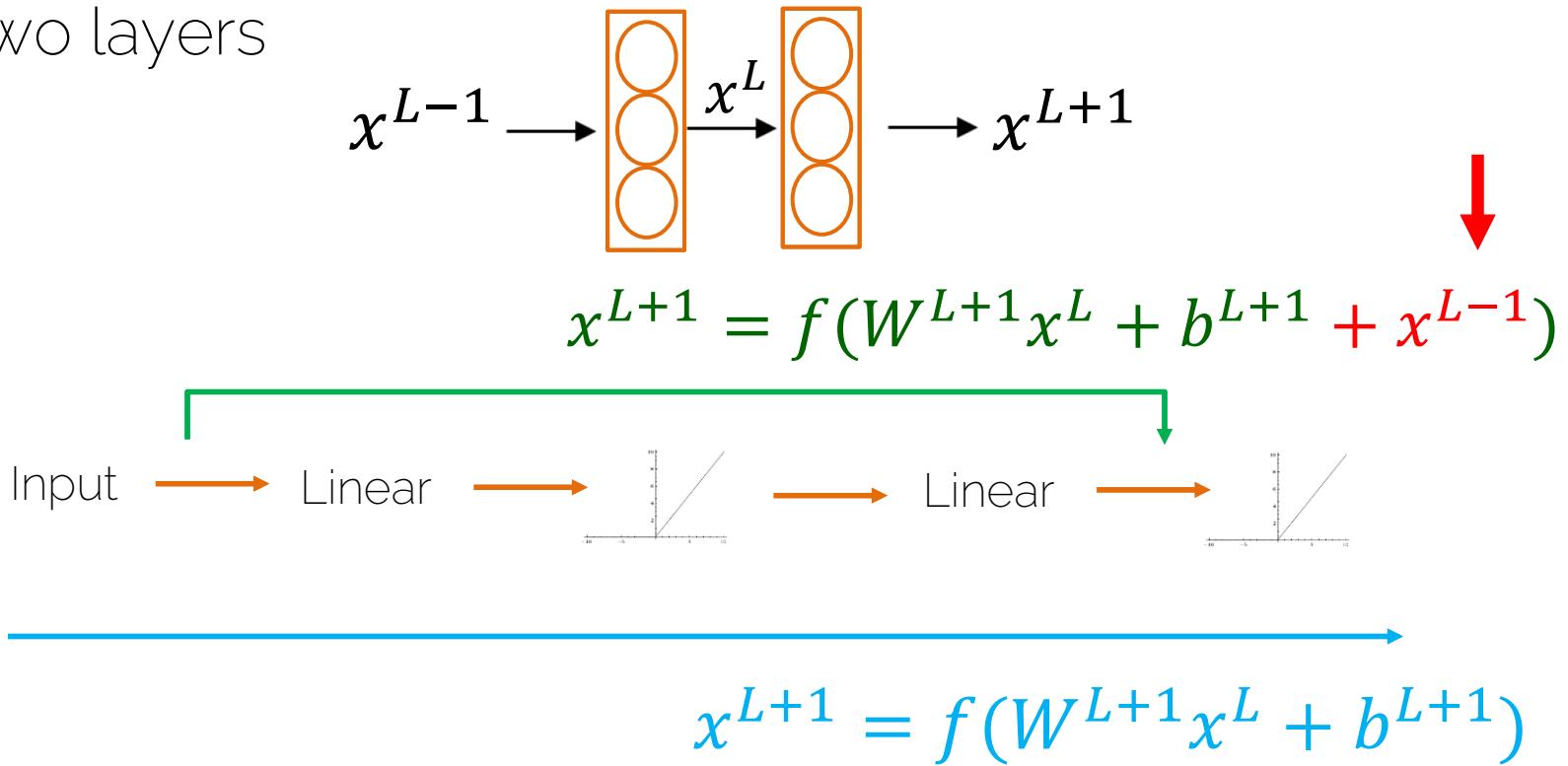
# Residual Block

- Two layers



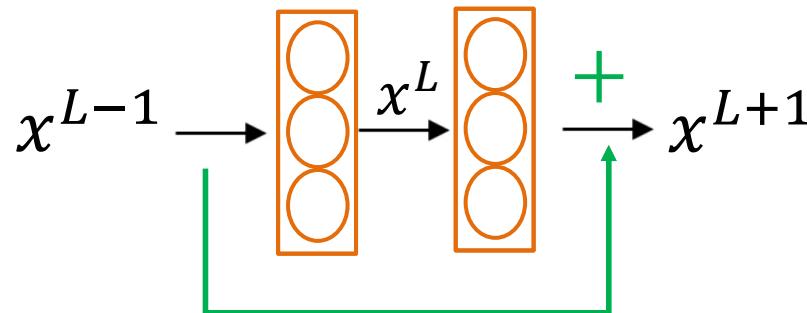
# Residual Block

- Two layers



# Residual Block

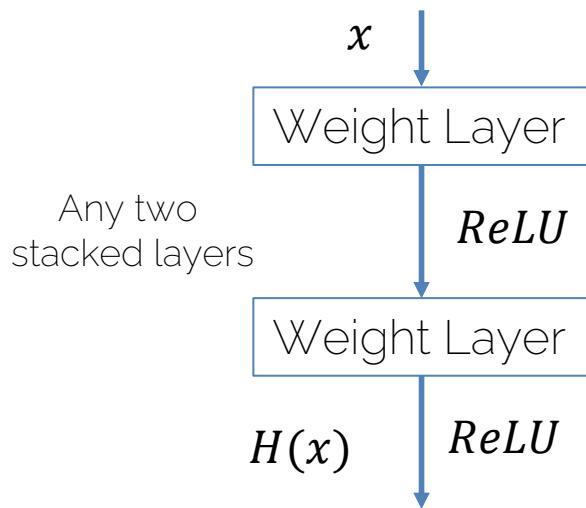
- Two layers



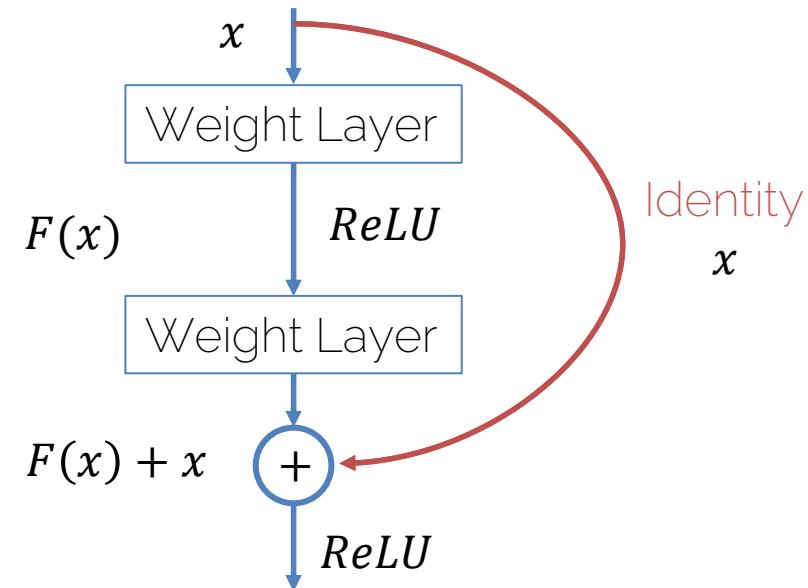
- Usually use a same convolution since we need same dimensions
- Otherwise we need to convert the dimensions with a matrix of learned weights or zero padding

# ResNet Block

Plain Net

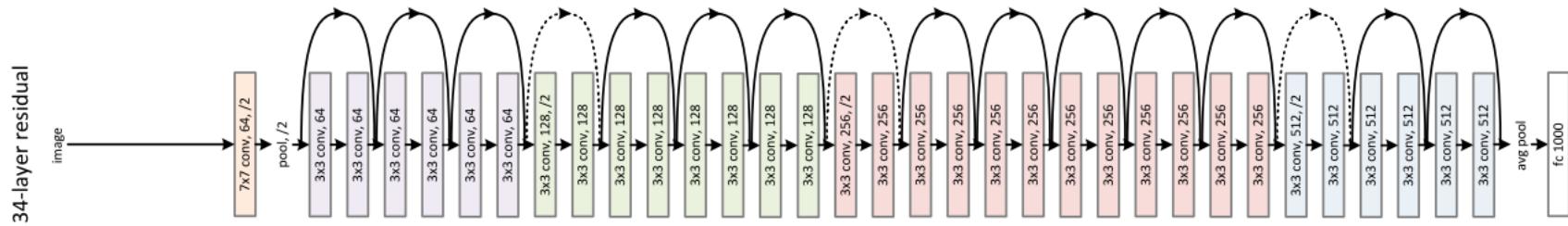


Residual Net



[He et al. CVPR'16] ResNet

# ResNet

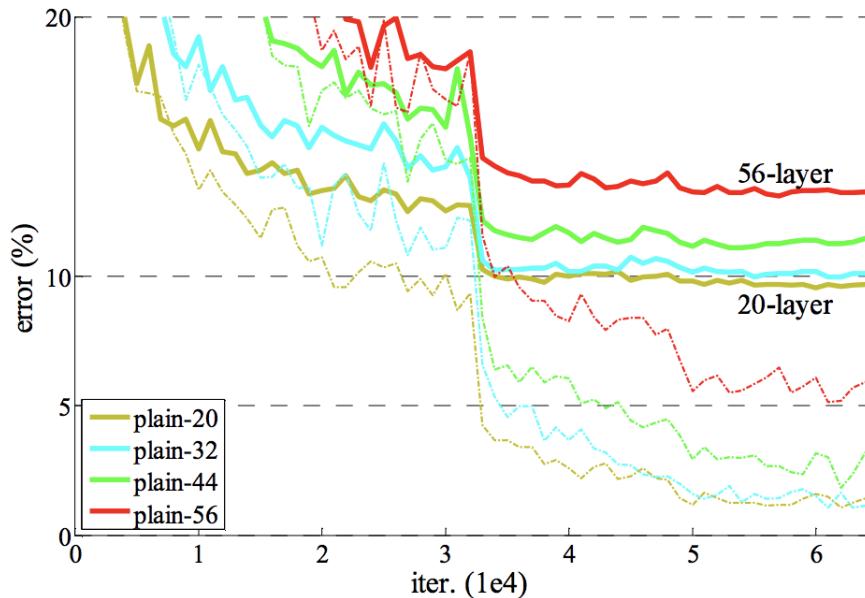


- Xavier/2 initialization
- SGD + Momentum (0.9)
- Learning rate 0.1, divided by 10 when plateau
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout

ResNet-152:  
60M parameters

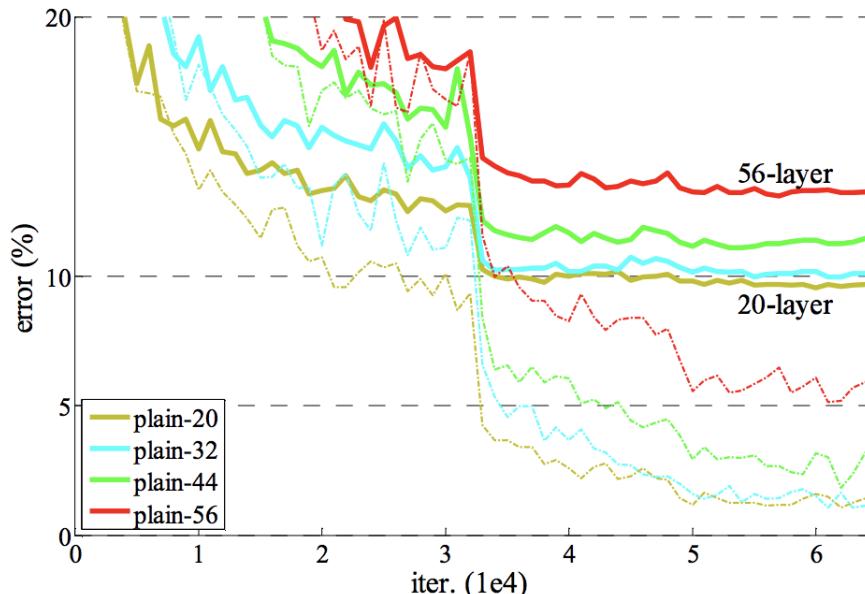
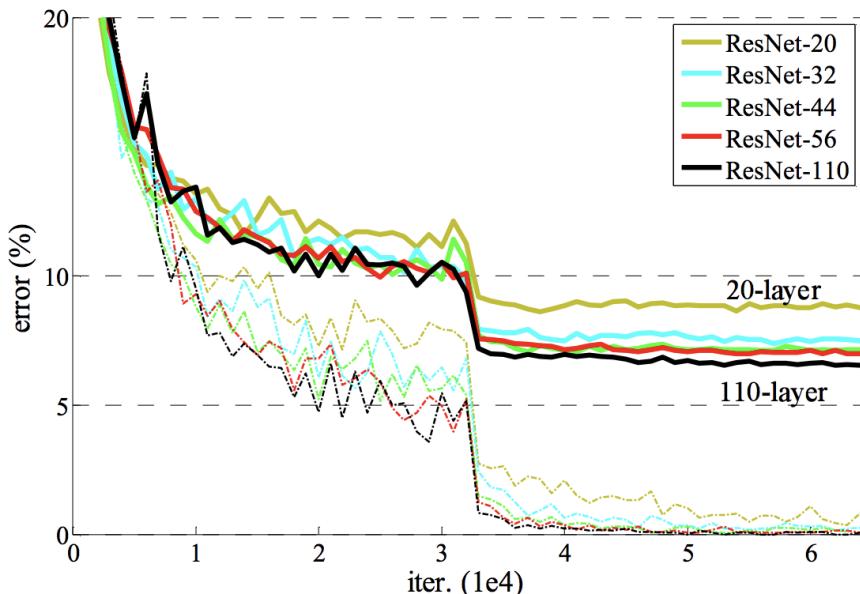
# ResNet

- If we make the network deeper, at some point performance starts to degrade

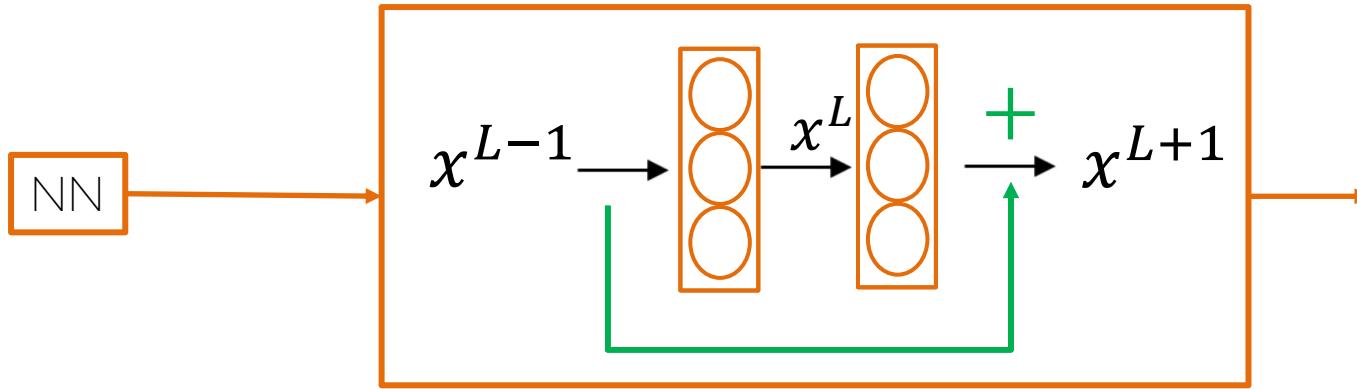


# ResNet

- If we make the network deeper, at some point performance starts to degrade

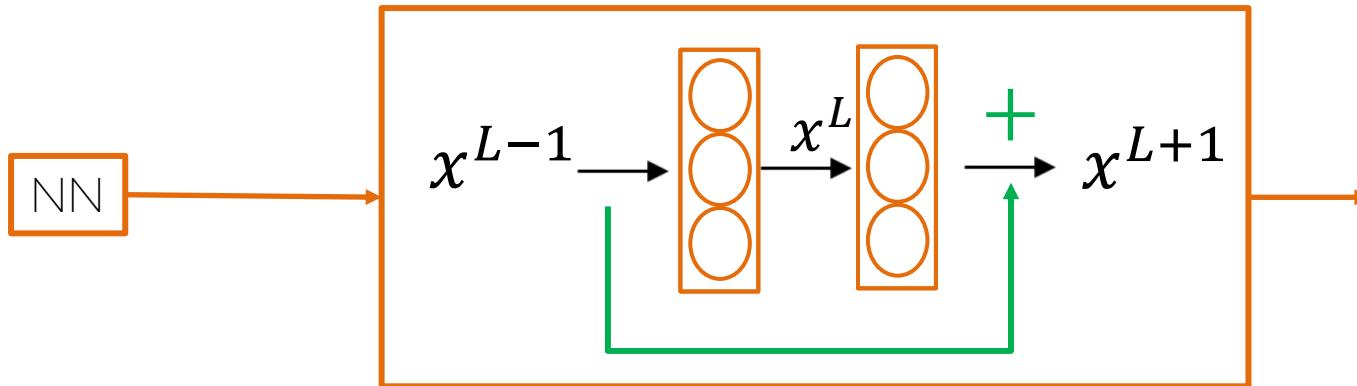


# Why do ResNets Work?



- How is this block really affecting me?

# Why do ResNets Work?

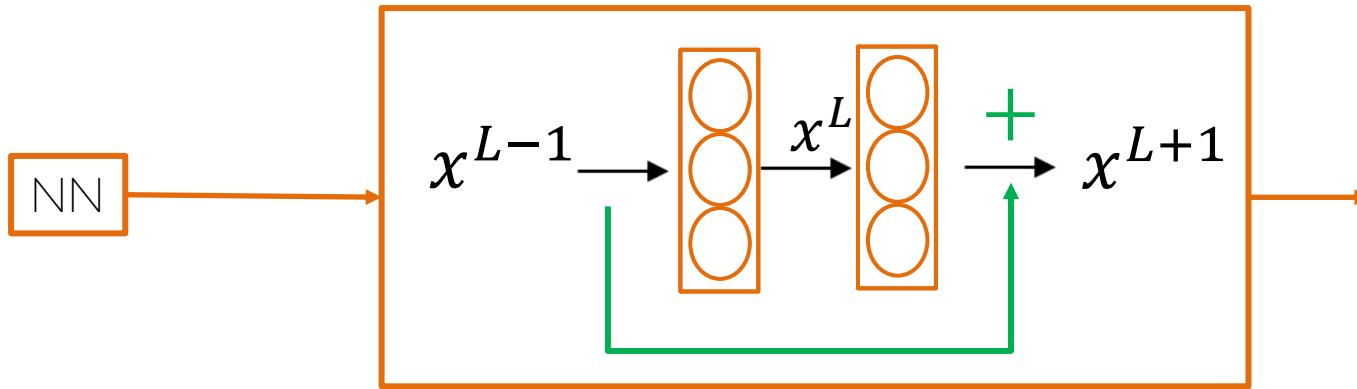


$$x^{L+1} = f(W^{L+1}x^L + b^{L+1} + x^{L-1})$$

$\sim \text{zero}$        $\sim \text{zero}$

$$x^{L+1} = f(x^{L-1})$$

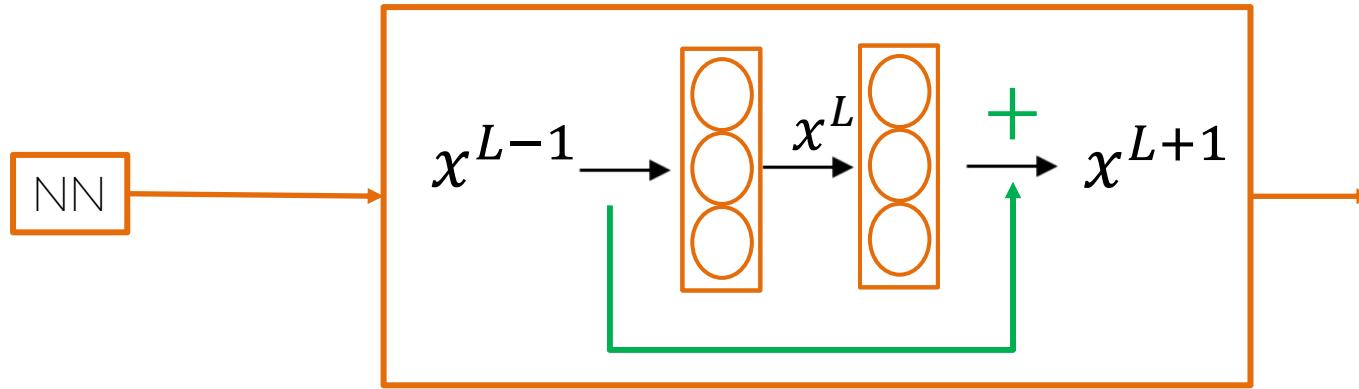
# Why do ResNets Work?



- We kept the same values and added a non-linearity

$$x^{L+1} = f(x^{L-1})$$

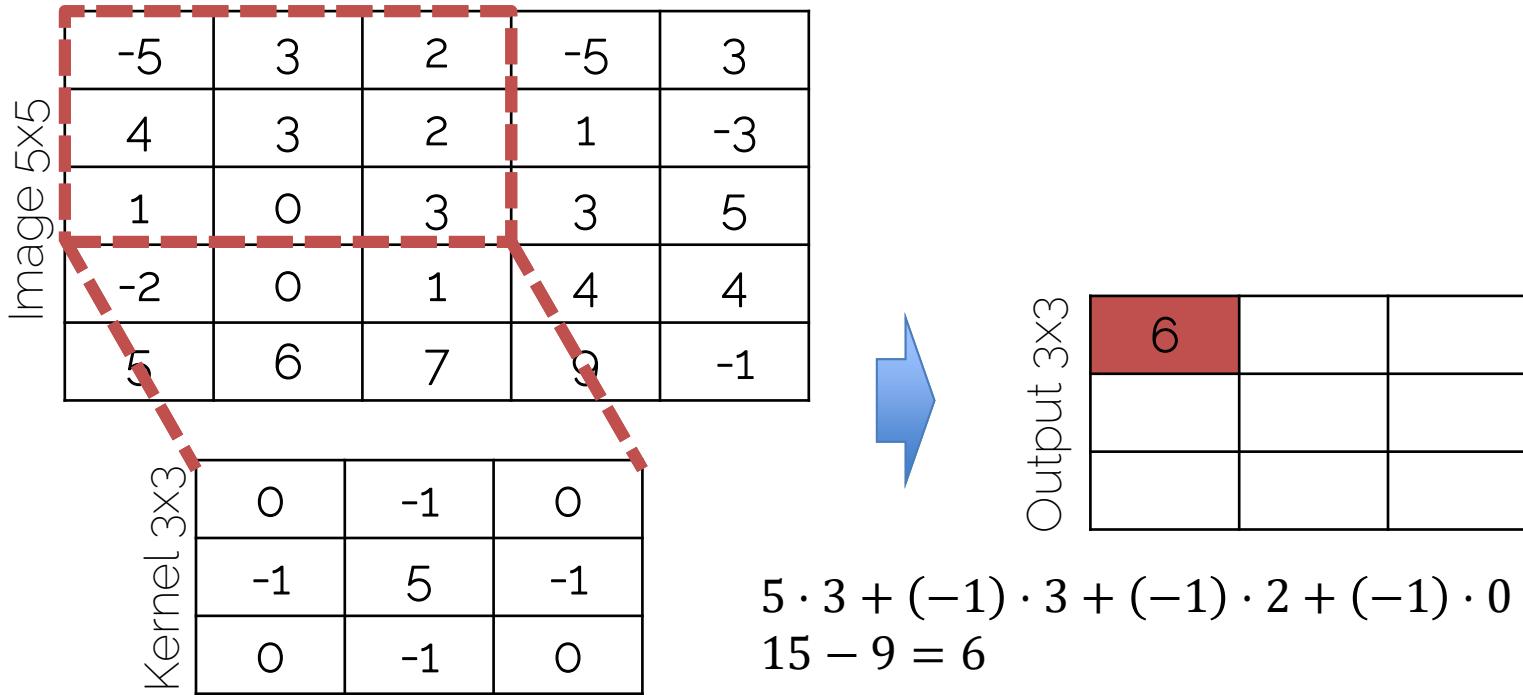
# Why do ResNets Work?



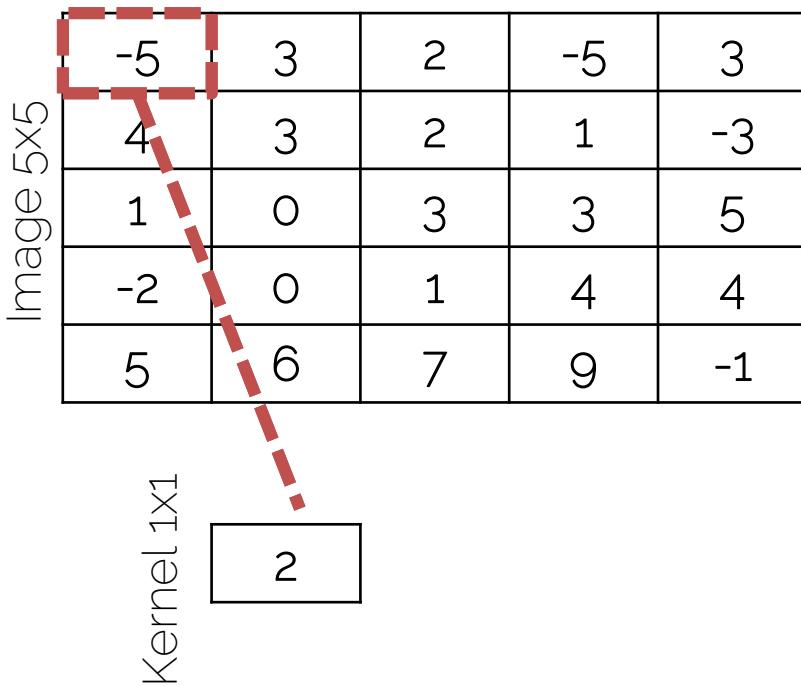
- The identity is easy for the residual block to learn
- Guaranteed it will not hurt performance, can only improve

# 1x1 Convolutions

# Recall: Convolutions on Images

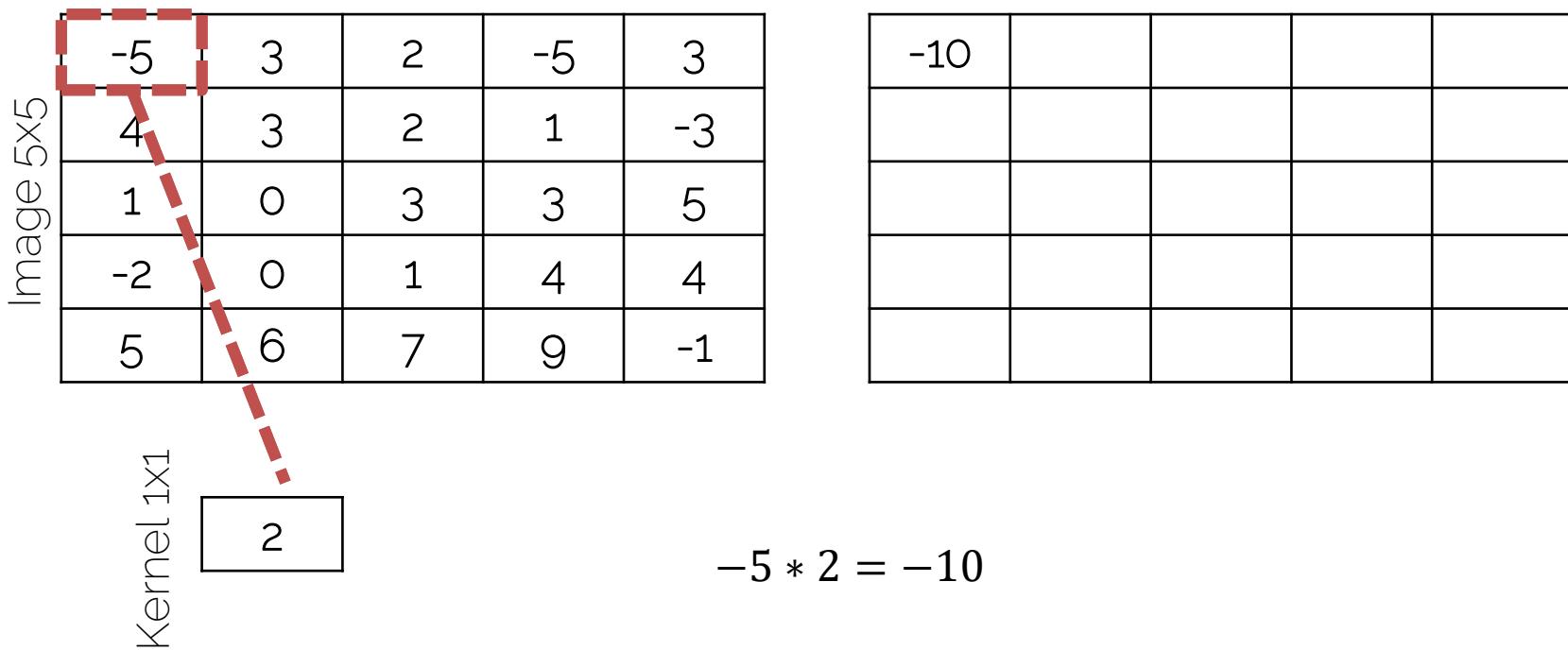


# 1x1 Convolution



What is the output size?

# 1x1 Convolution



# 1x1 Convolution

Image 5x5

-5	3	2	-5	3
4	3	2	1	-3
1	0	3	3	5
-2	0	1	4	4
5	6	7	9	-1

Kernel 1x1

2
---

-10	6	4	-10	6
8	6	4	2	-6
2	0	6	6	10
-4	0	2	8	8
10	12	14	18	-2

$$-1 * 2 = -2$$

# 1x1 Convolution

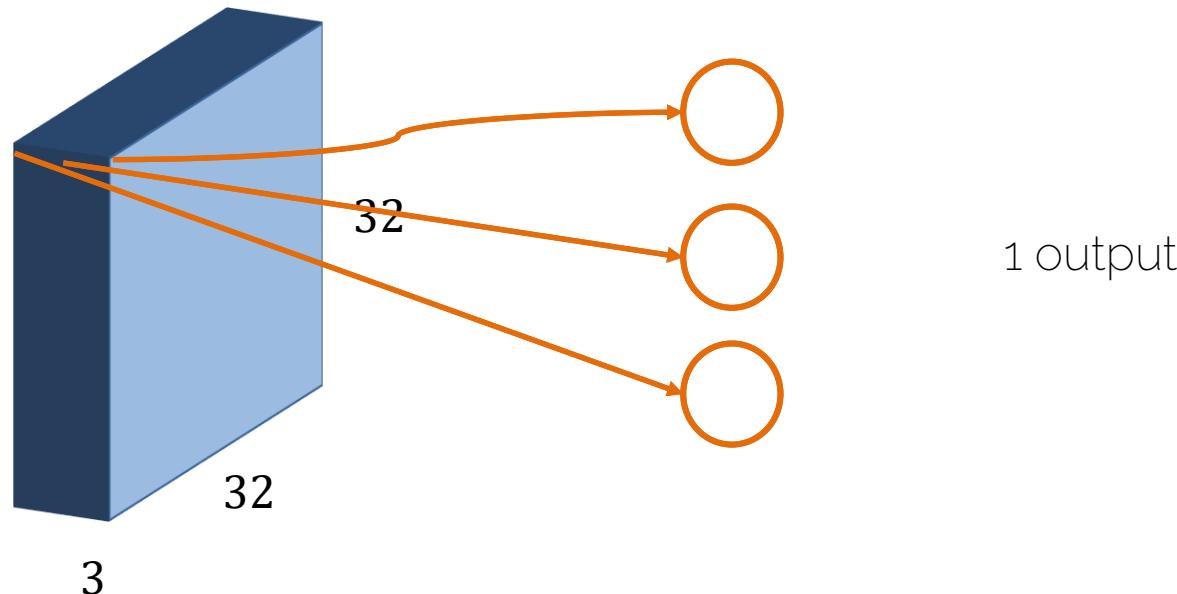
Image 5x5

-5	3	2	-5	3
4	3	2	1	-3
1	0	3	3	5
-2	0	1	4	4
5	6	7	9	-1

-10	6	4	-10	6
8	6	4	2	-6
2	0	6	6	10
-4	0	2	8	8
10	12	14	18	-2

- 1x1 kernel: keeps the dimensions and scales input

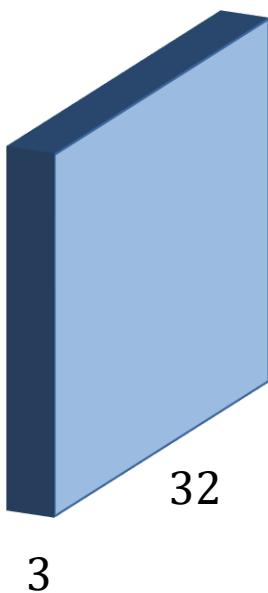
# 1x1 Convolution



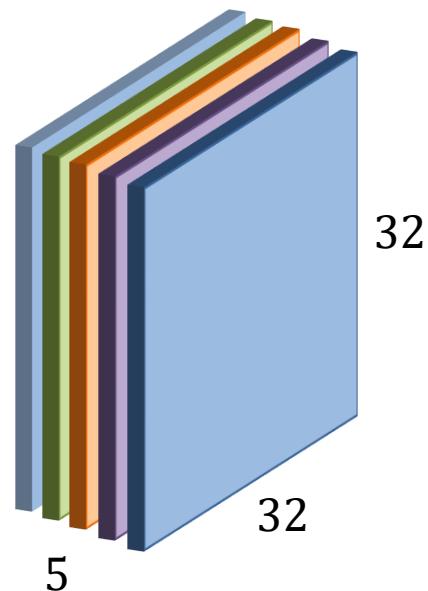
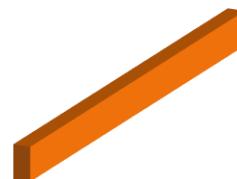
- Same as having a 3 neuron fully connected layer

# 1x1 Convolution

[Li et al. 2013]



*5 filters 1x1x3*



- As always we use more convolutional filters

# Using 1x1 Convolutions

- Use it to shrink the number of channels
- Further adds a non-linearity → one can learn more complex functions



# Inception Layer

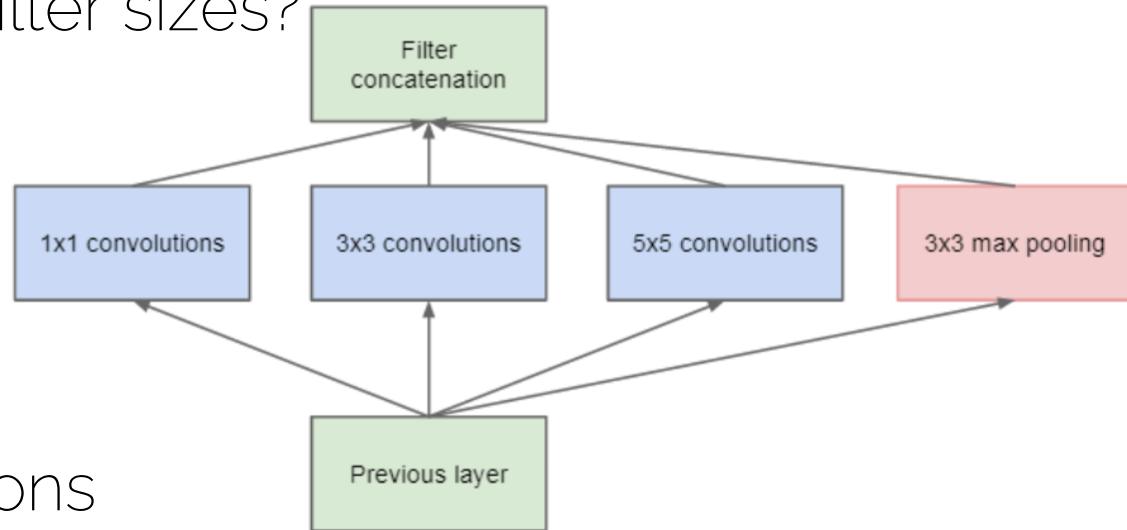
# Inception Layer

- Tired of choosing filter sizes?

- Use them all!

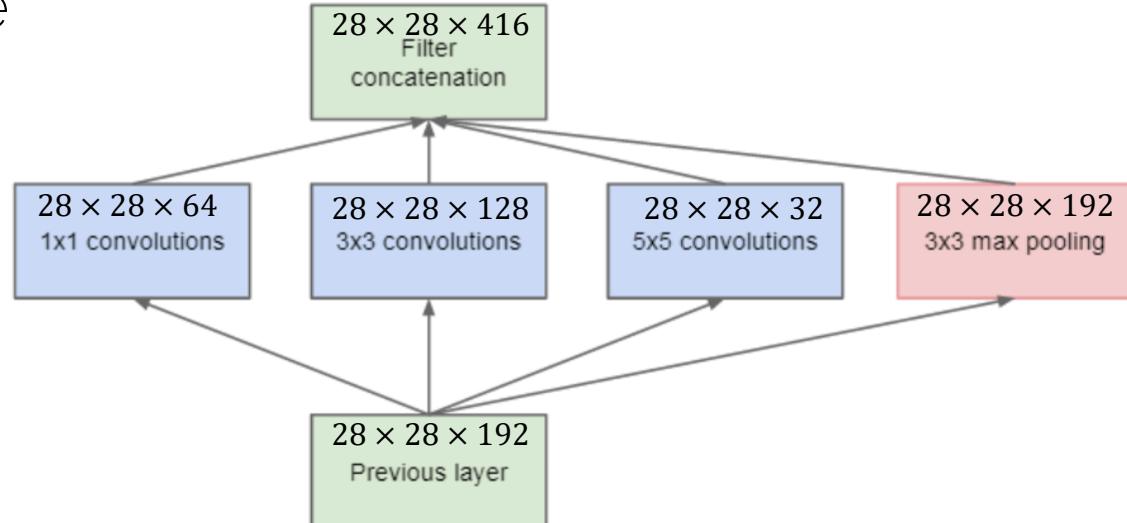
- All same convolutions

- 3x3 max pooling is with stride 1



# Inception Layer

- Possible size of the output



- Not sustainable!

# Inception Layer: Computational Cost

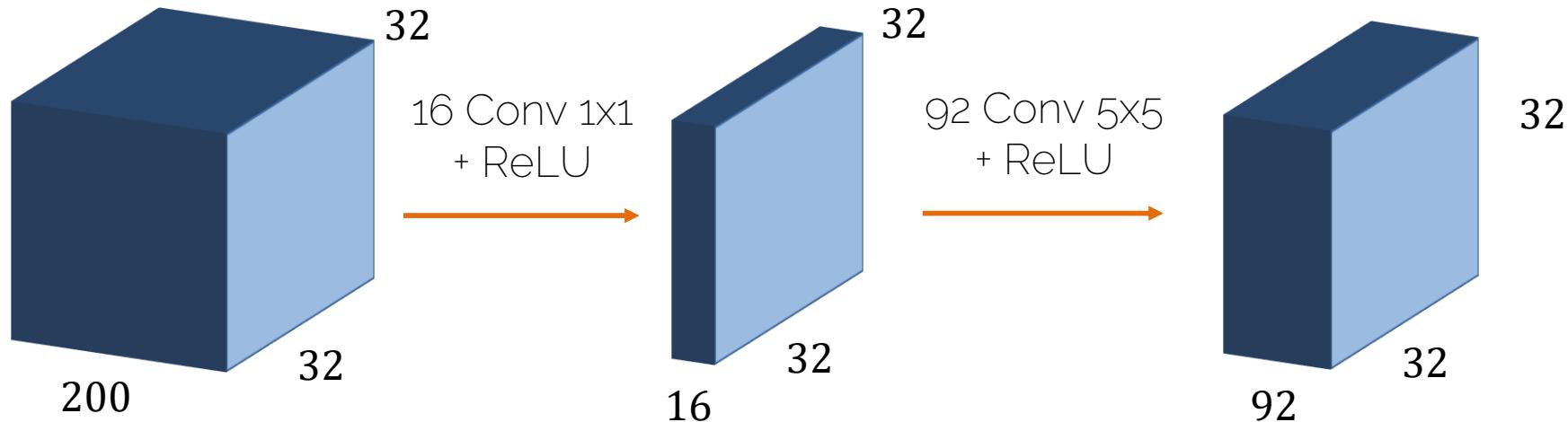


Multiplications:  $5 \times 5 \times 200 \times 32 \times 32 \times 92 \sim 470 \text{ million}$



1 value of the output volume

# Inception Layer: Computational Cost



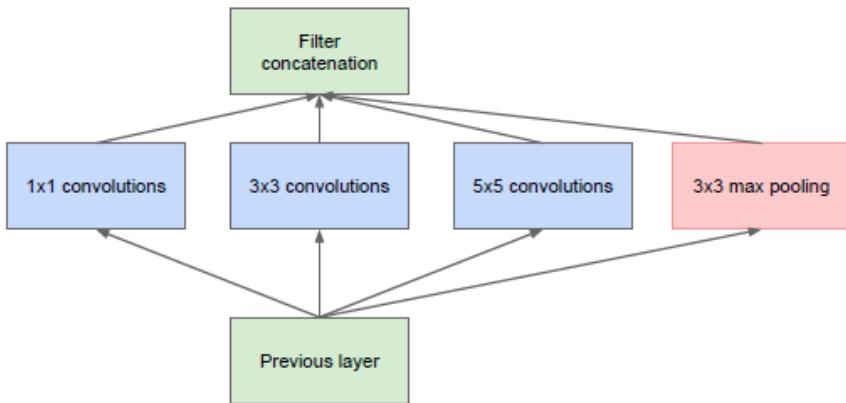
Multiplications:  $1 \times 1 \times 200 \times 32 \times 32 \times 16$

$5 \times 5 \times 16 \times 32 \times 32 \times 92$

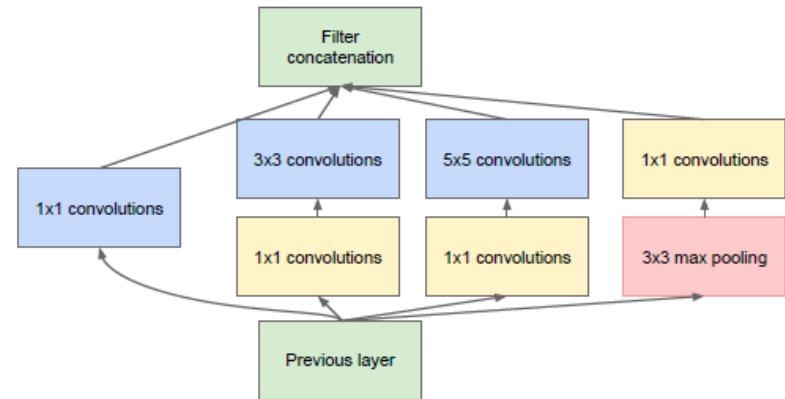
~ 40 million

Reduction of multiplications by 1/10

# Inception Layer



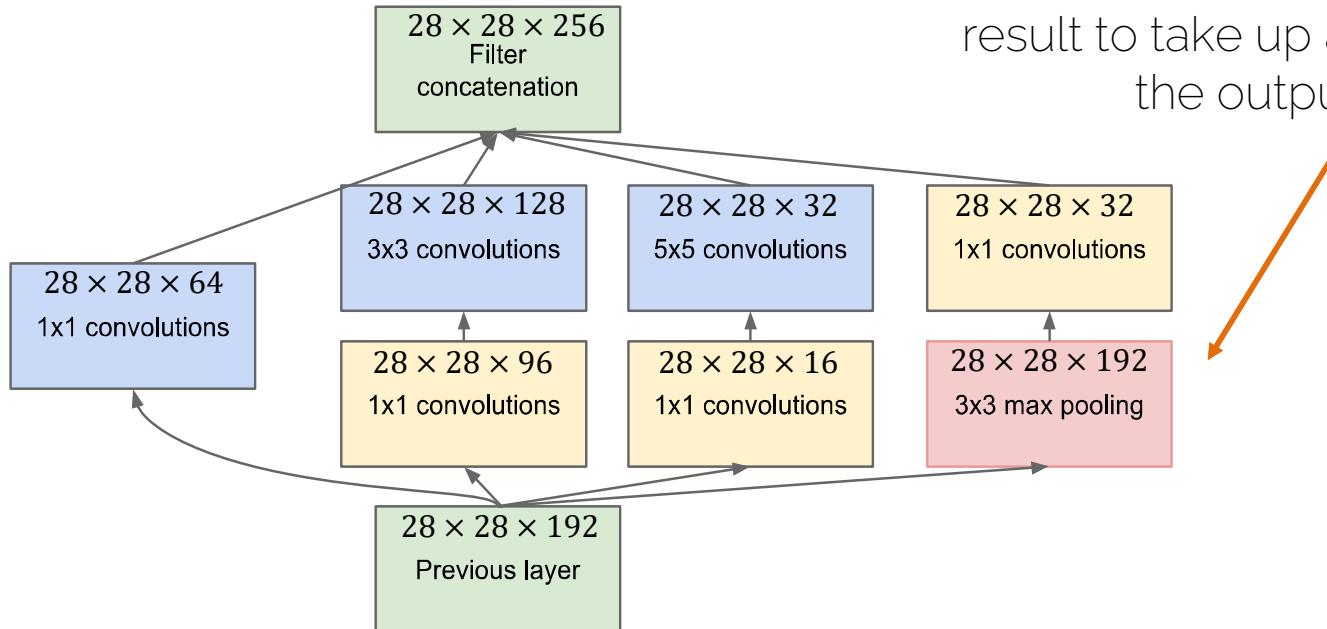
(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

[Szegedy et al CVPR'15] GoogLeNet

# Inception Layer: Dimensions

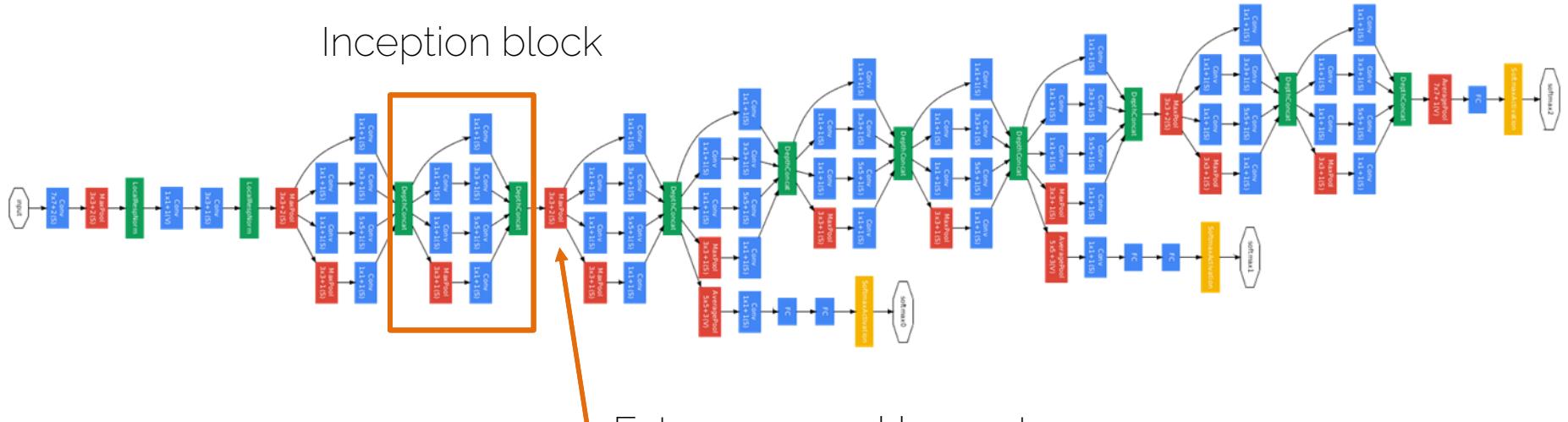


We do not want max pool result to take up almost all the output

[Szegedy et al CVPR'15] GoogLeNet

# GoogLeNet: Using the Inception Layer

## Inception block



Extra max pool layers to reduce dimensionality

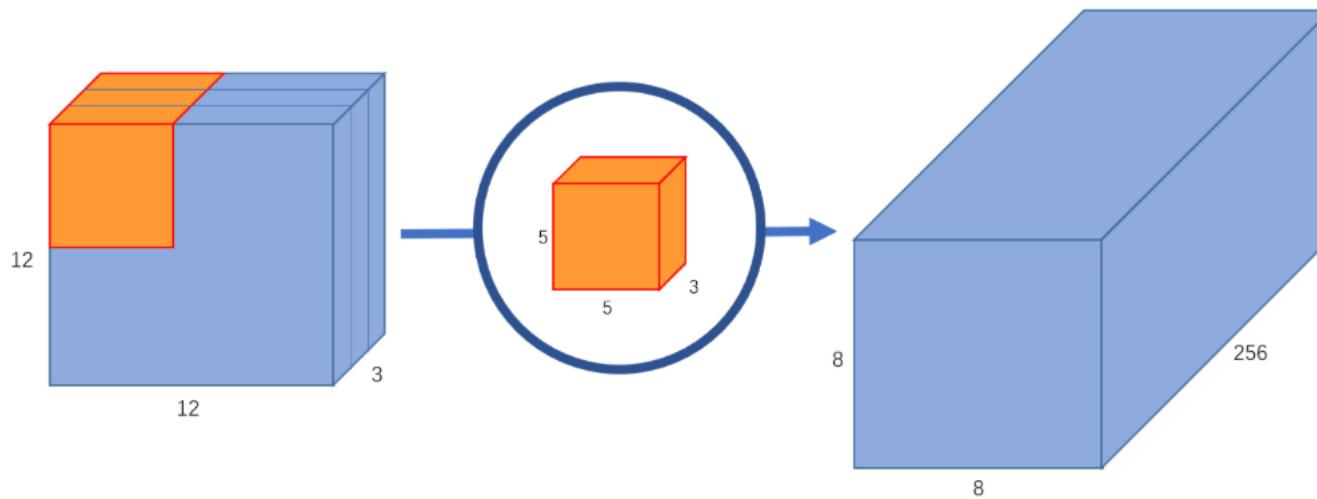
[Szegedy et al CVPR'15] GoogLeNet

# Xception Net

- “Extreme version of Inception”: applying (modified) **Depthwise Separable Convolutions** instead of normal convolutions
- 36 conv layers, structured into several modules with **skip connections**
- outperforms Inception Net V3

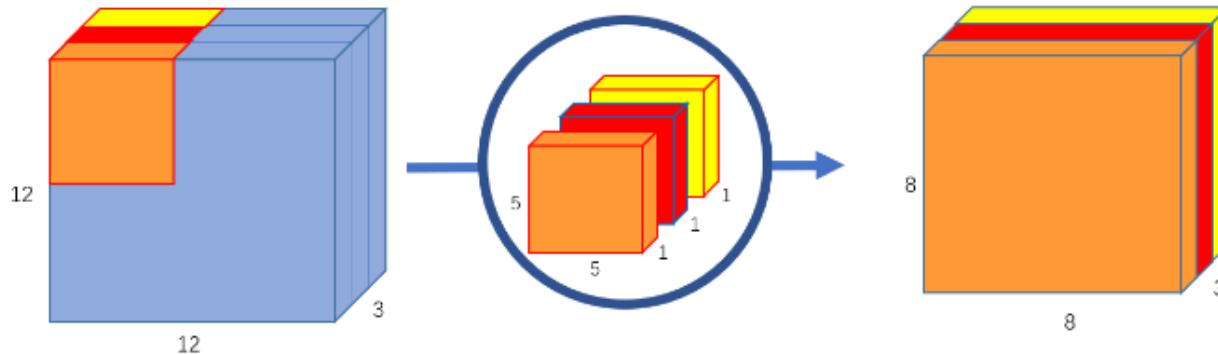
[Chollet CVPR'17] Xception

# Depth-wise separable convolutions



Normal convolutions act on all channels.

# Depth-wise separable convolutions

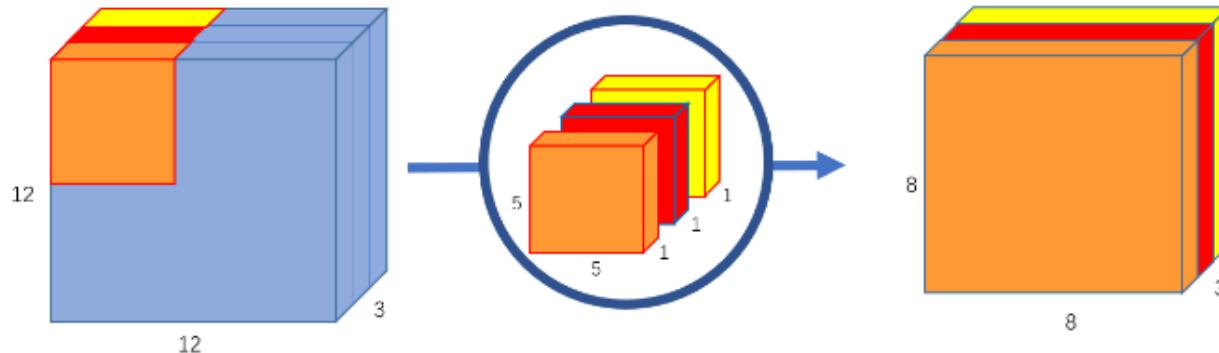


Filters are applied only at certain depths of the features.  
Normal convolutions have groups set to 1, the convolutions used in this image have groups set to 3.

```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)
```

```
class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)
```

# Depth-wise separable convolutions

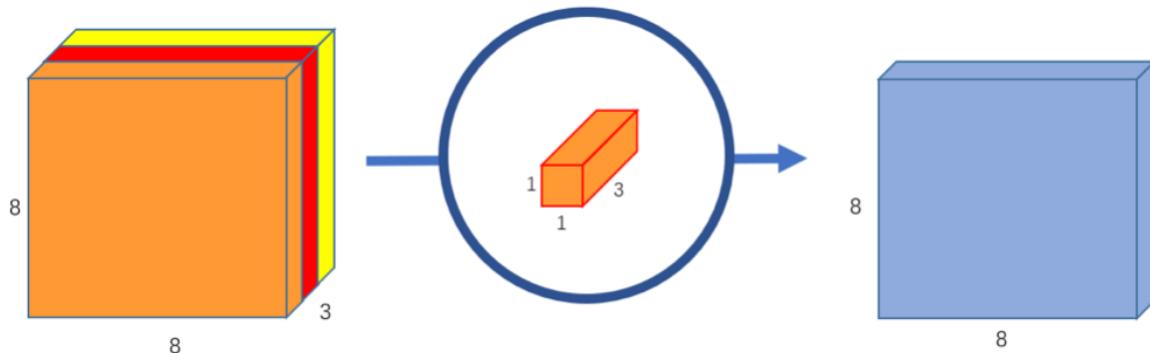
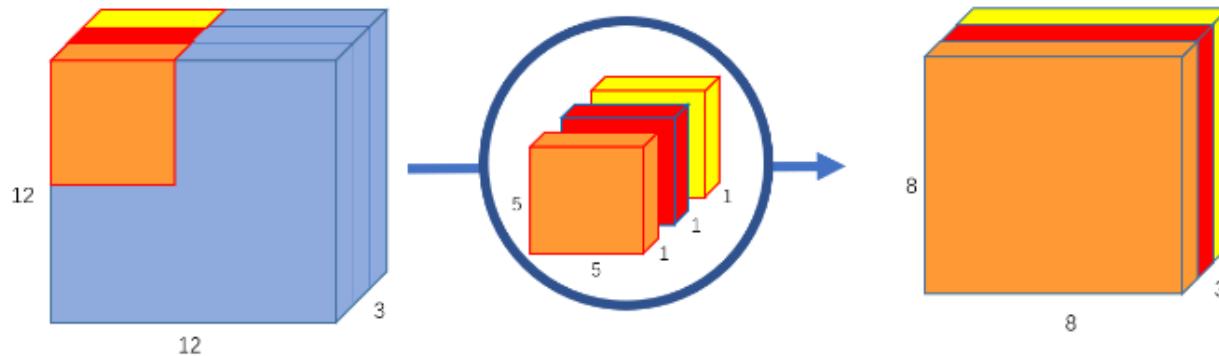


But the depth size is always the same!

```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)
```

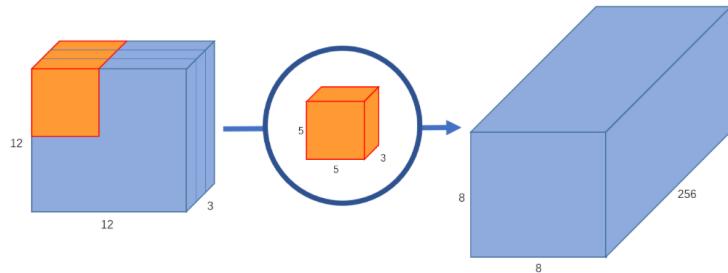
```
class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)
```

# Depth-wise separable convolutions



Solution:  
1x1 convs!

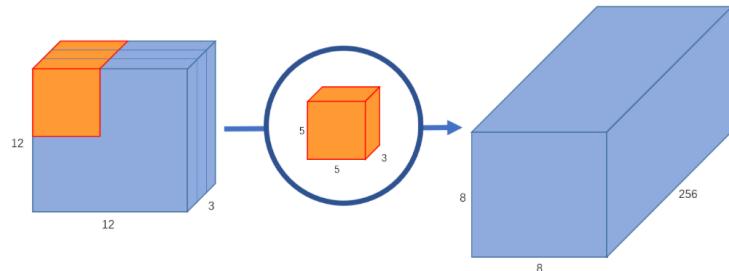
# But why?



Original convolution  
256 kernels of size 5x5x3

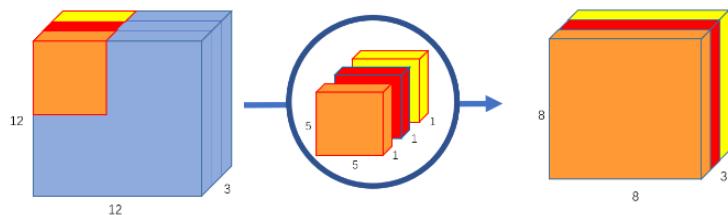
Multiplications:  
 $256 \times 5 \times 5 \times 3 \times (8 \times 8 \text{ locations}) = 1.228.800$

# But why?



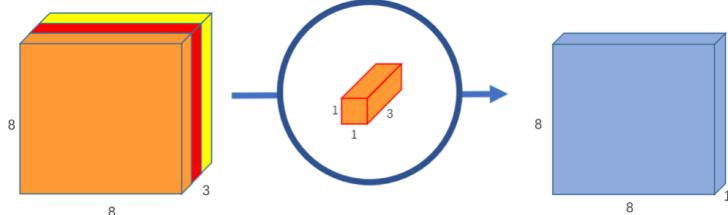
Original convolution  
256 kernels of size 5x5x3

Multiplications:  
 $256 \times 5 \times 5 \times 3 \times (8 \times 8 \text{ locations}) = 1.228.800$



Depth-wise convolution  
3 kernels of size 5x5x1

Multiplications:  
 $5 \times 5 \times 3 \times (8 \times 8 \text{ locations}) = 4800$

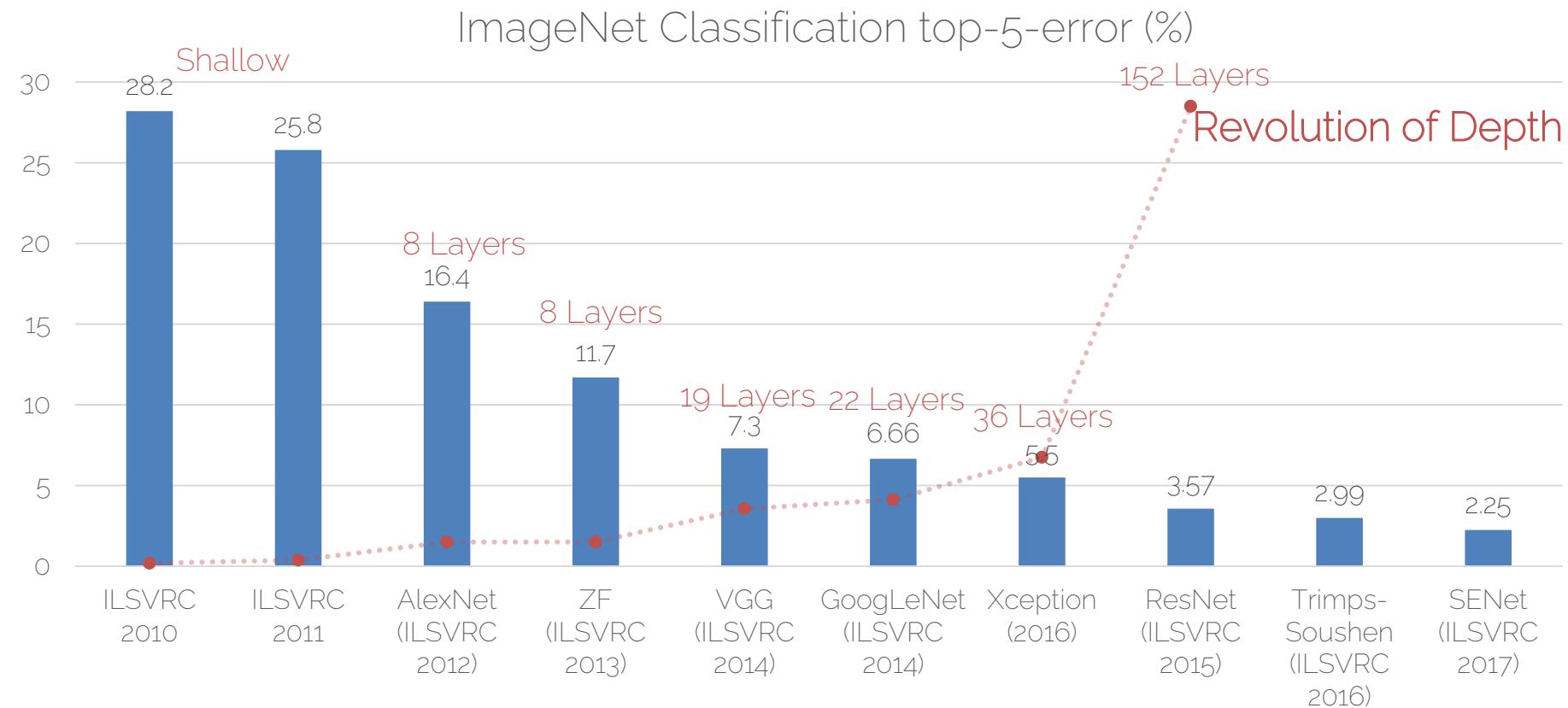


1x1 convolution  
256 kernels of size 1x1x3

Multiplications:  
 $256 \times 1 \times 1 \times 3 \times (8 \times 8 \text{ locations}) = 49152$

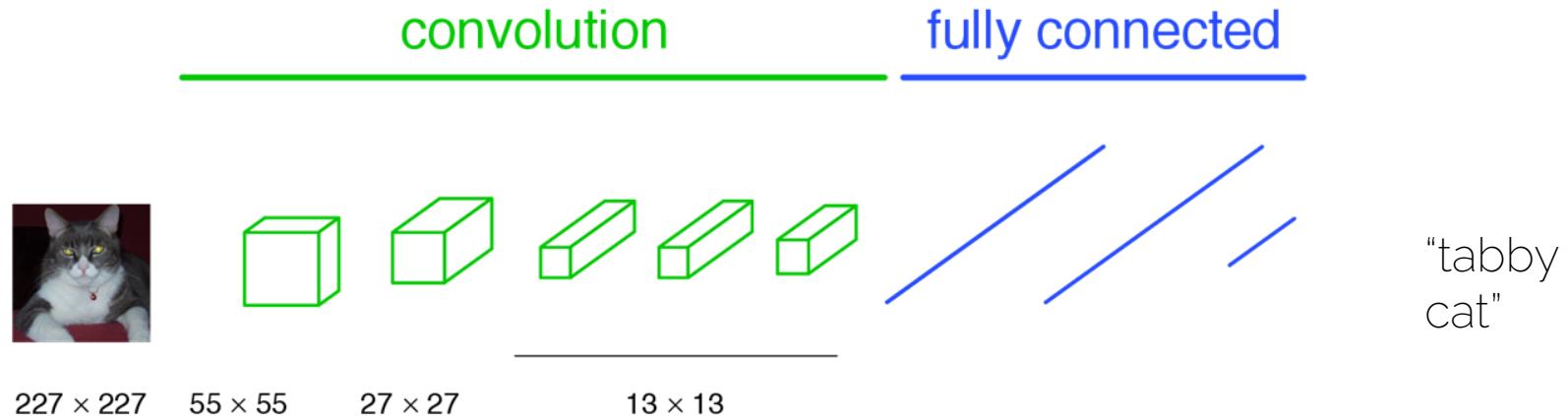
Less computation!

# ImageNet Benchmark

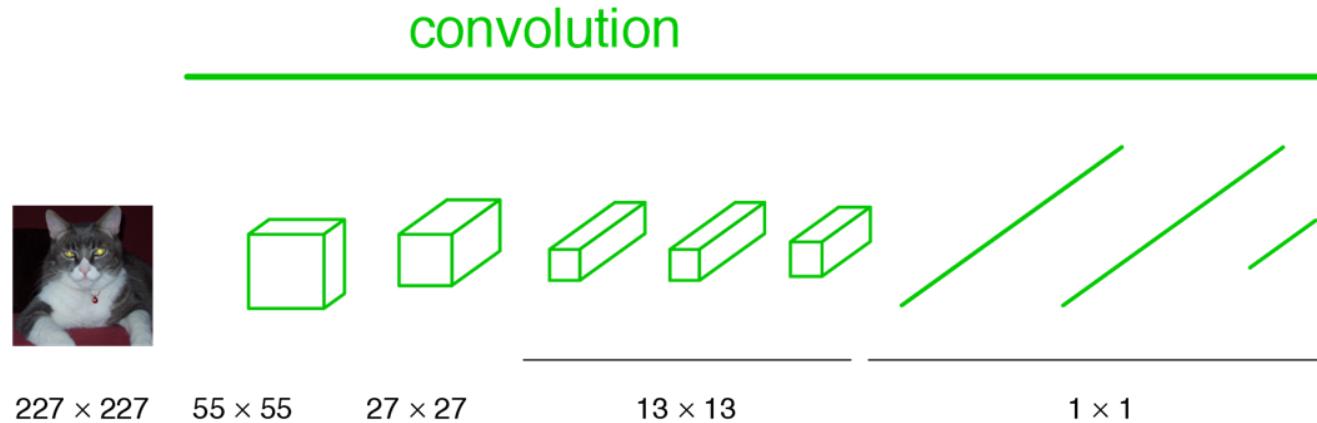


# Fully Convolutional Network

# Classification Network

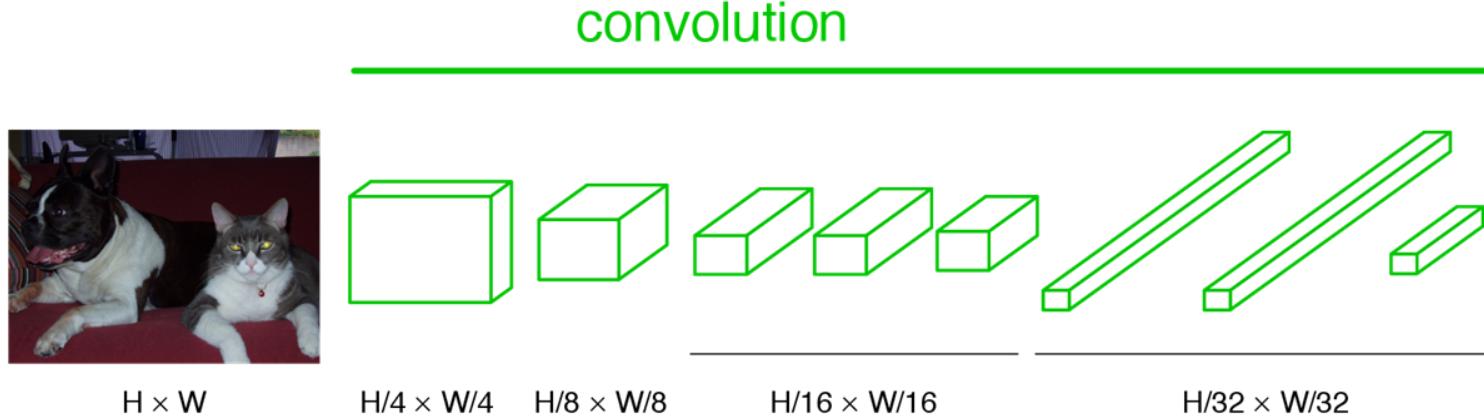


# FCN: Becoming Fully Convolutional

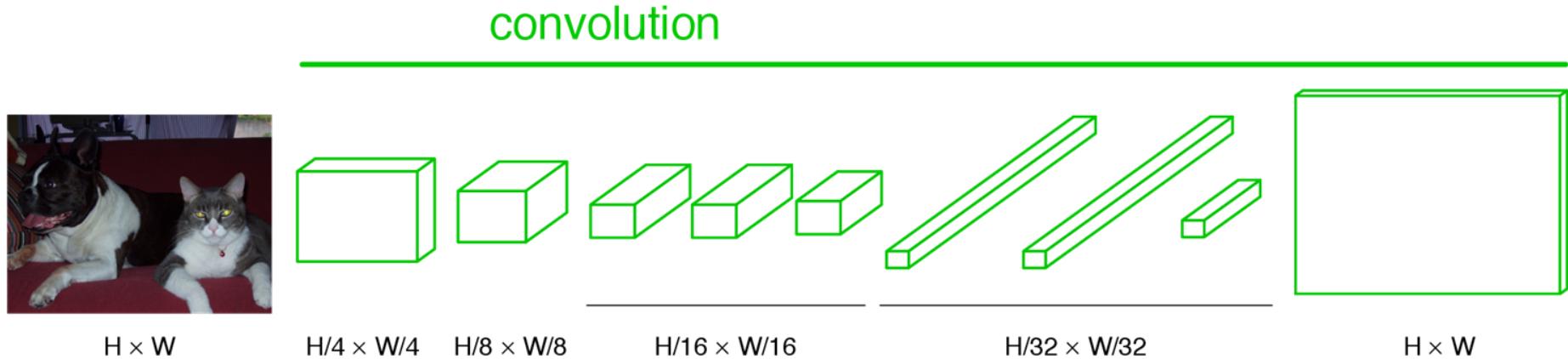


Convert fully connected layers to convolutional layers!

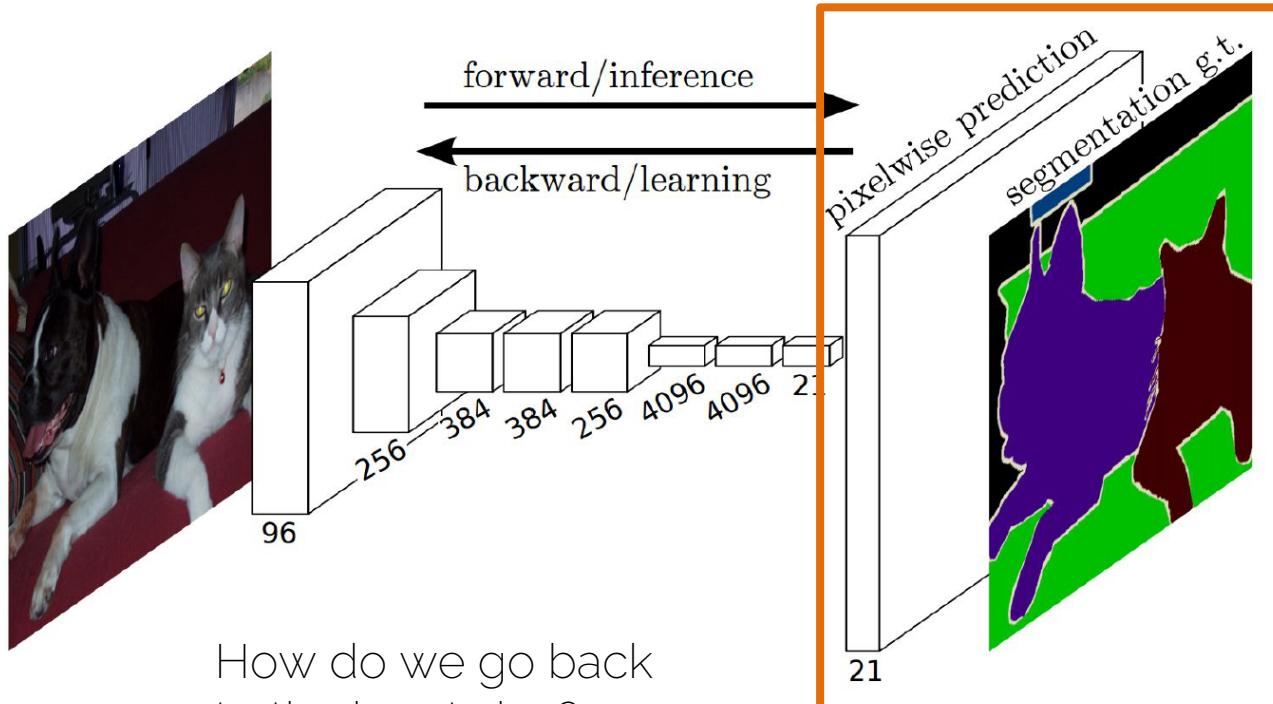
# FCN: Becoming Fully Convolutional



# FCN: Upsampling Output



# Semantic Segmentation (FCN)

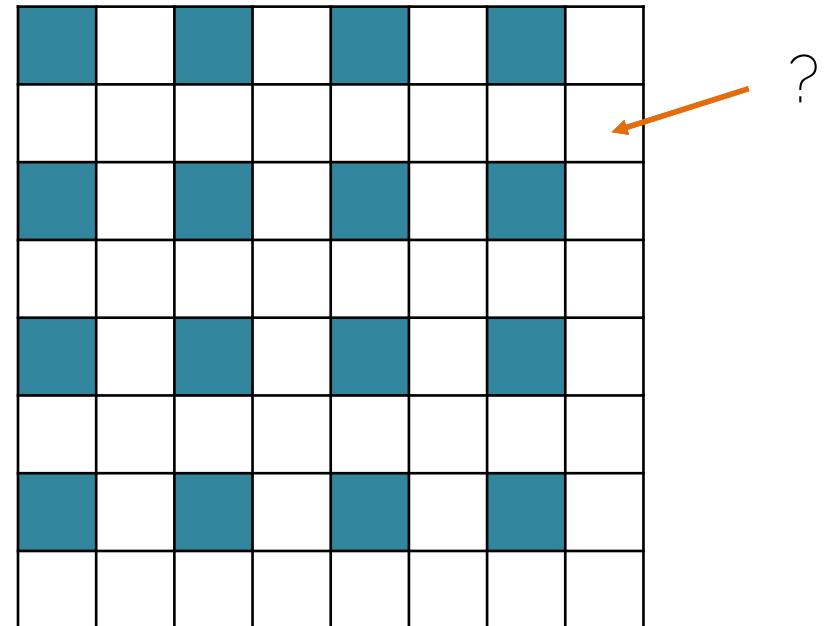
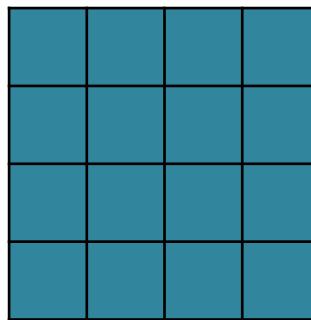


How do we go back  
to the input size?

[Long and Shelhamer. 15] FCN

# Types of Upsampling

- 1. Interpolation



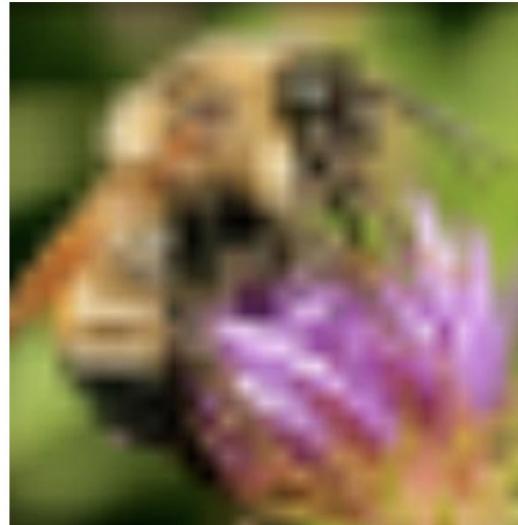
# Types of Upsampling

- 1. Interpolation

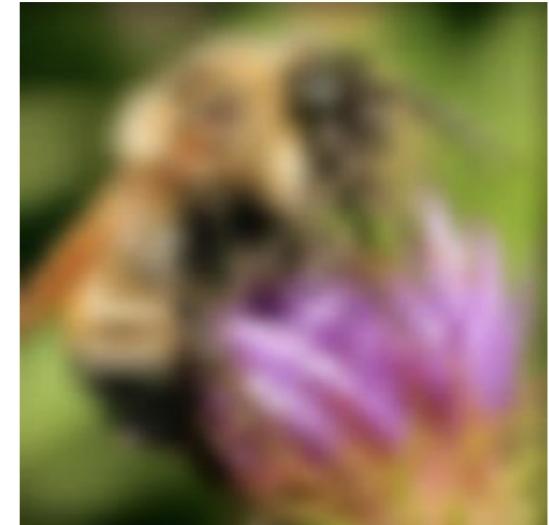
Original image  **x 10**



Nearest neighbor interpolation



Bilinear interpolation



Bicubic interpolation

# Types of Upsampling

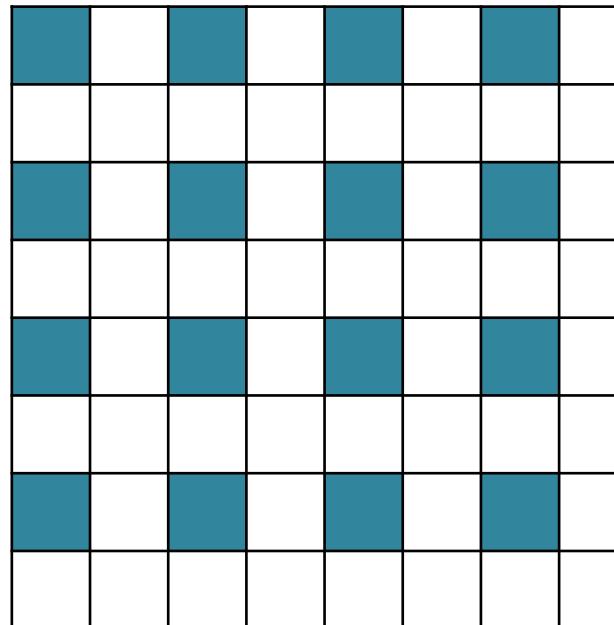
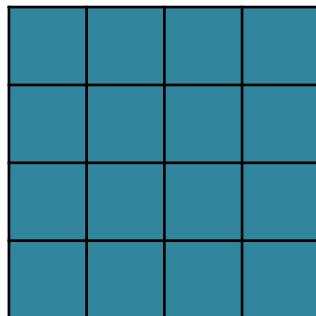
- 1. Interpolation



Few artifacts

# Types of Upsampling

- 2. Transposed conv



+ CONVS

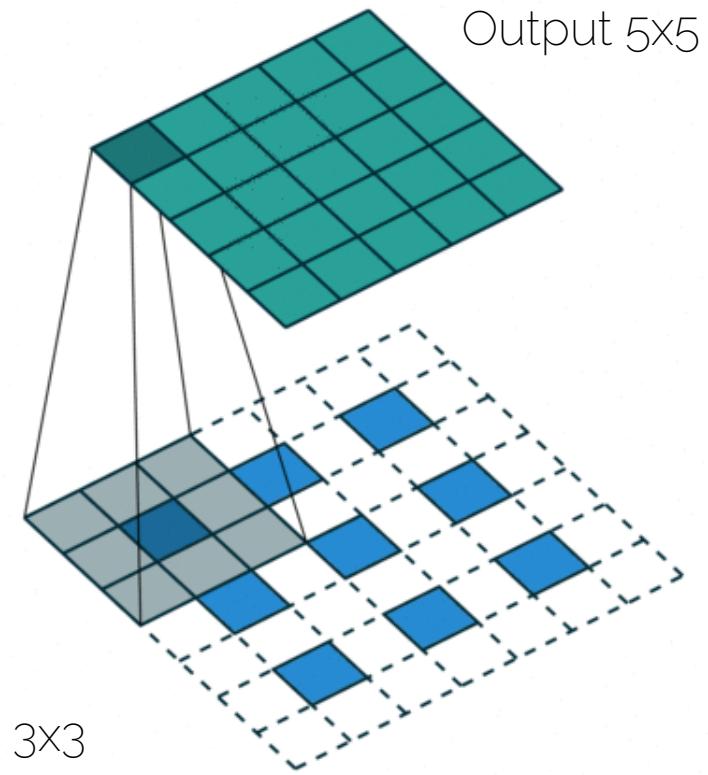
efficient

[A. Dosovitskiy, TPAMI 2017] "Learning to Generate Chairs, Tables and Cars with Convolutional Networks"

# Types of Upsampling

- 2. Transposed convolution

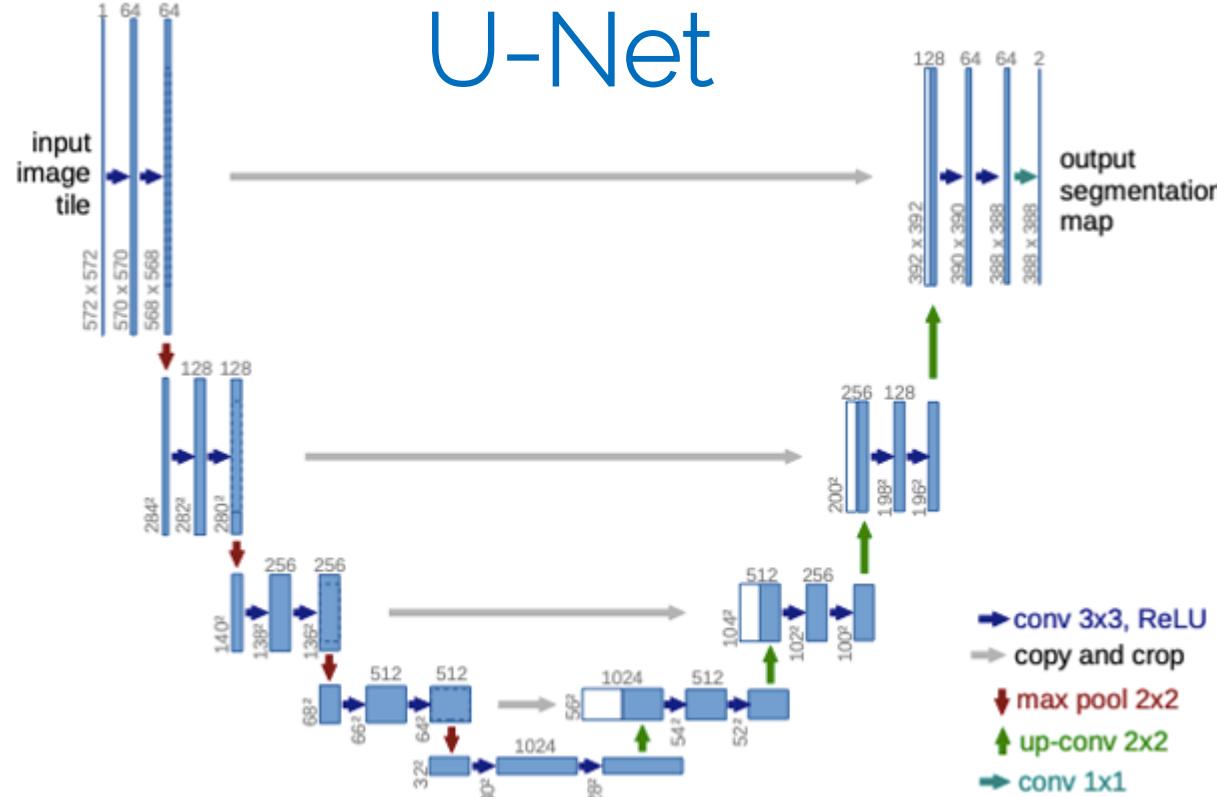
- Unpooling
  - Convolution filter (learned)
- Also called up-convolution  
**(never deconvolution)**



# Refined Outputs

- If one does a cascade of unpooling + conv operations, we get to the encoder-decoder architecture
- Even more refined: Autoencoders with skip connections (aka U-Net)

# U-Net



U-Net architecture: Each blue box is a multichannel feature map. Number of channels denoted at the top of the box. Dimensions at the top of the box. White boxes are the copied feature maps.

# U-Net: Encoder

Left side: Contraction Path (Encoder)

- Captures context of the image
  - Follows typical architecture of a CNN:
    - Repeated application of 2 unpadded 3x3 convolutions
    - Each followed by ReLU activation
    - 2x2 maxpooling operation with stride 2 for downsampling
    - At each downsampling step, # of channels is doubled
- as before: Height, Width , Depth: 

[Ronneberger et al. MICCAI'15] U-Net

# U-Net: Decoder

Right Side: Expansion Path (Decoder):

- Upsampling to recover spatial locations for assigning class labels to each pixel
  - 2x2 up-convolution that halves number of input channels
  - **Skip Connections:** outputs of up-convolutions are concatenated with feature maps from encoder
  - Followed by 2 ordinary 3x3 convs
  - final layer: 1x1 conv to map 64 channels to # classes
- Height, Width:  , Depth: 

[Ronneberger et al. MICCAI'15] U-Net

See you next time!

# References

*We highly recommend to read through these papers!*

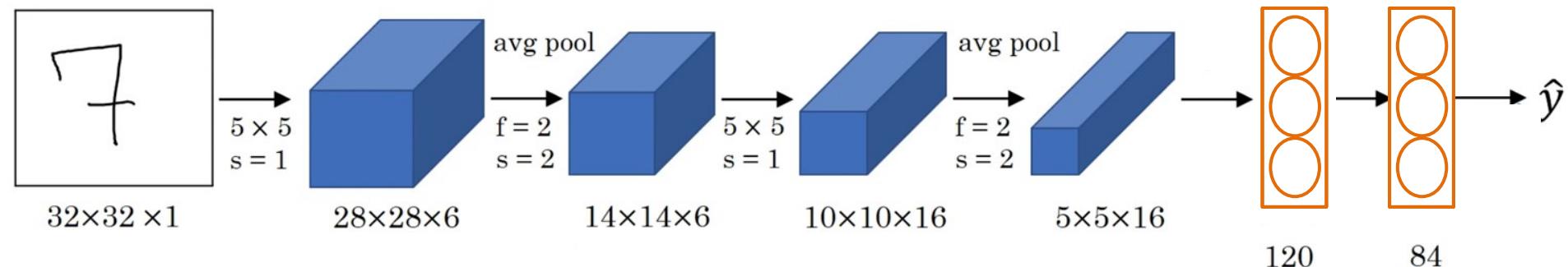
- [AlexNet](#) [Krizhevsky et al. 2012]
- [VGGNet](#) [Simonyan & Zisserman 2014]
- [ResNet](#) [He et al. 2015]
- [GoogLeNet](#) [Szegedy et al. 2014]
- [Xception](#) [Chollet 2016]
- [Fast R-CNN](#) [Girshick 2015]
- [U-Net](#) [Ronneberger et al. 2015]
- [EfficientNet](#) [Tan & Le 2019]

# Lecture 10 Recap

# LeNet

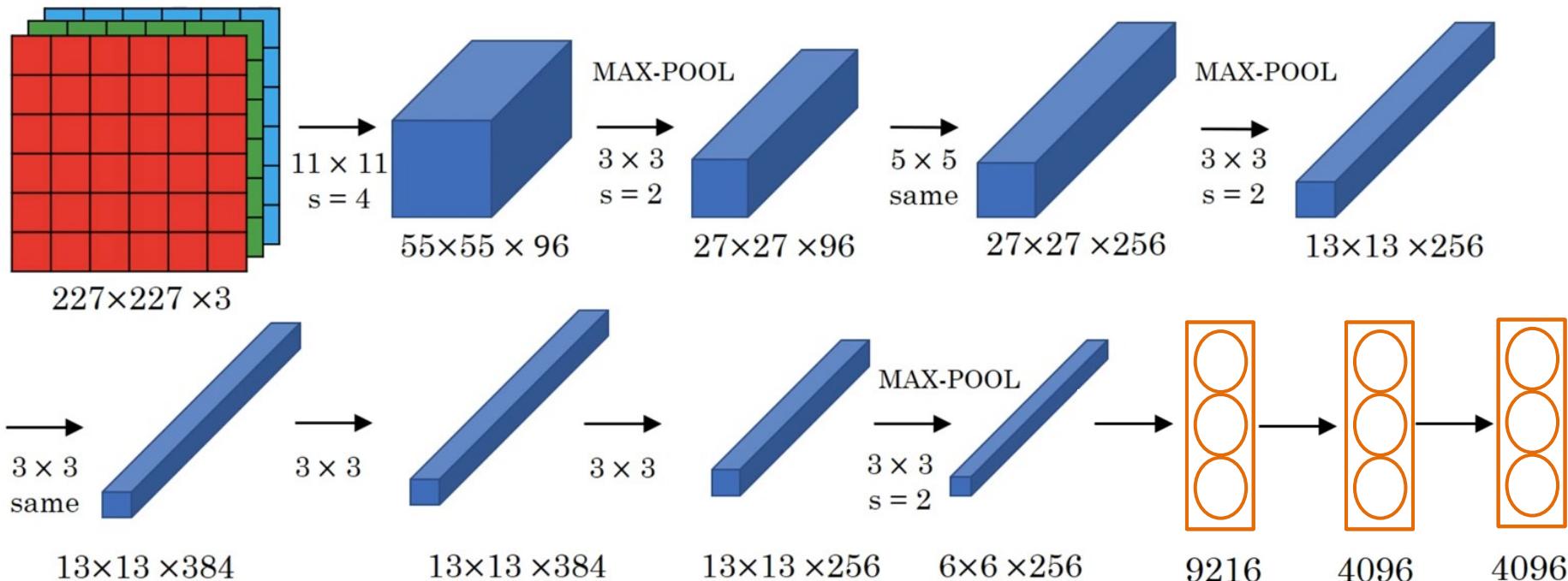
- Digit recognition: 10 classes

60k parameters



- Conv -> Pool -> Conv -> Pool -> Conv -> FC
- As we go deeper: Width, Height  $\downarrow$  Number of Filters  $\uparrow$

# AlexNet



- Softmax for 1000 classes

[Krizhevsky et al., ANIPS'12] AlexNet

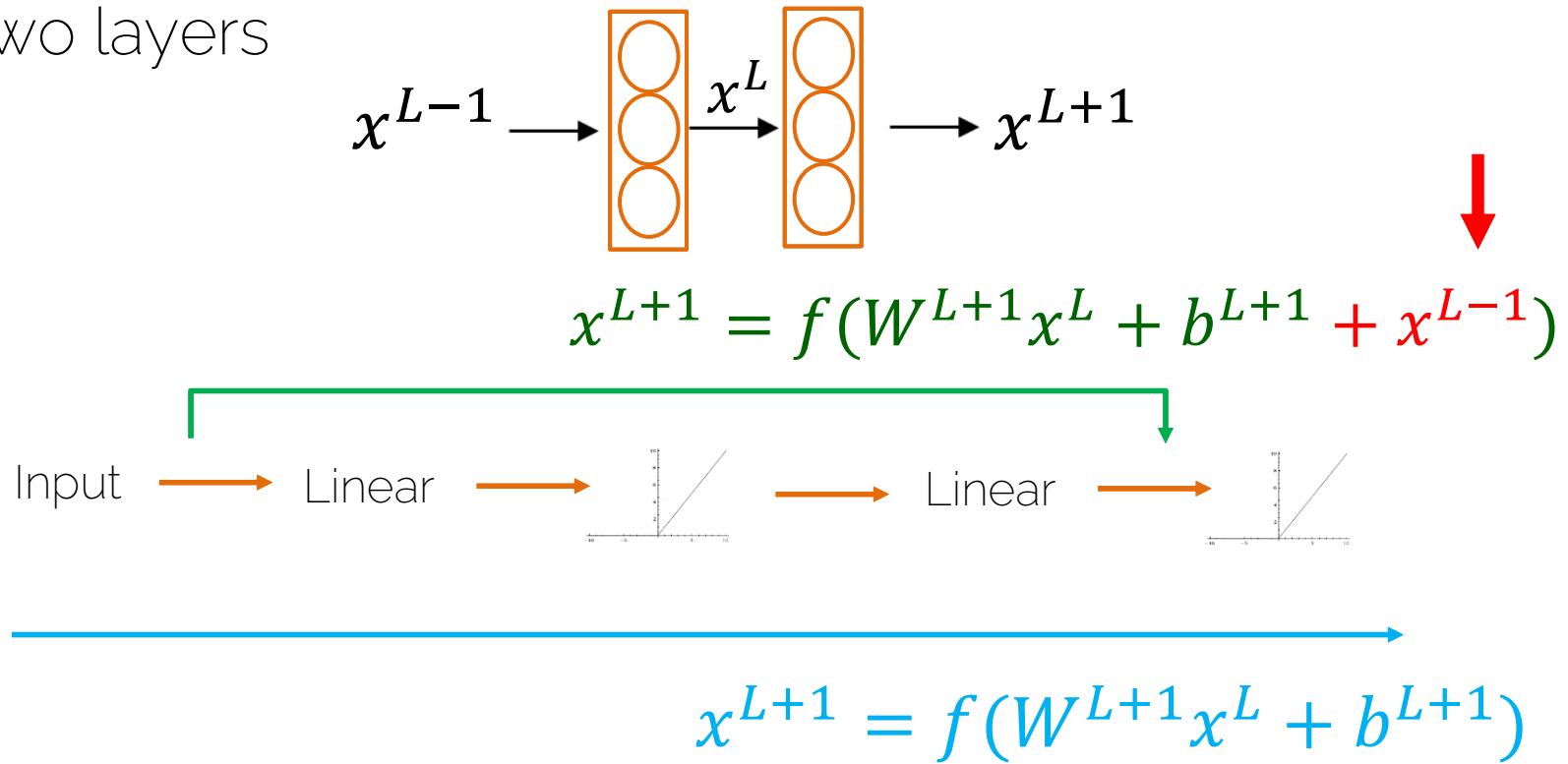
# VGGNet

- Striving for **simplicity**
  - Conv -> Pool -> Conv -> Pool -> Conv -> FC
  - Conv=3x3, s=1, same; Maxpool=2x2, s=2
- As we go deeper: Width, Height  Number of Filters 
- Called VGG-16: 16 layers that have weights
  - 138M parameters
- Large but simplicity makes it appealing

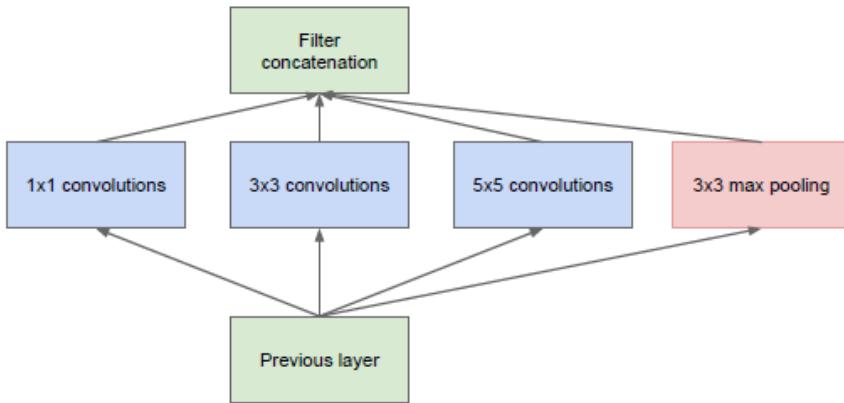
[Simonyan et al., ICLR'15] VGGNet

# Residual Block

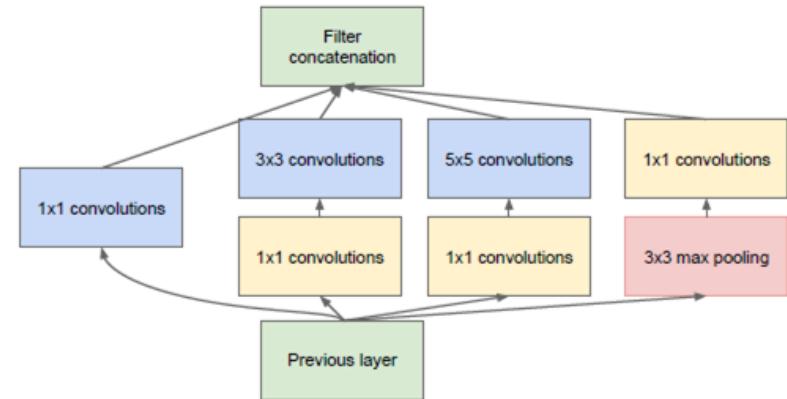
- Two layers



# Inception Layer



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

[Szegedy et al., CVPR'15] GoogleNet

# Lecture 11

# Transfer Learning

# Transfer Learning

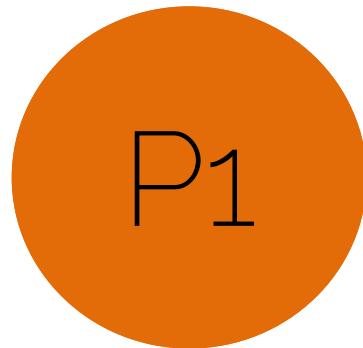
- Training your own model can be difficult with limited data and other resources

e.g.,

- It is a laborious task to manually annotate your own training dataset
- Why not reuse already pre-trained models?

# Transfer Learning

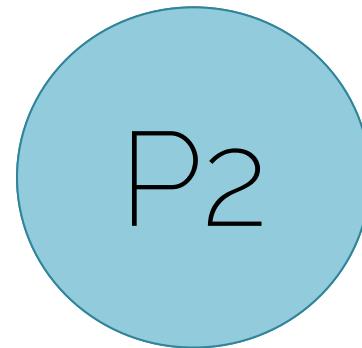
Distribution



Large dataset



Distribution

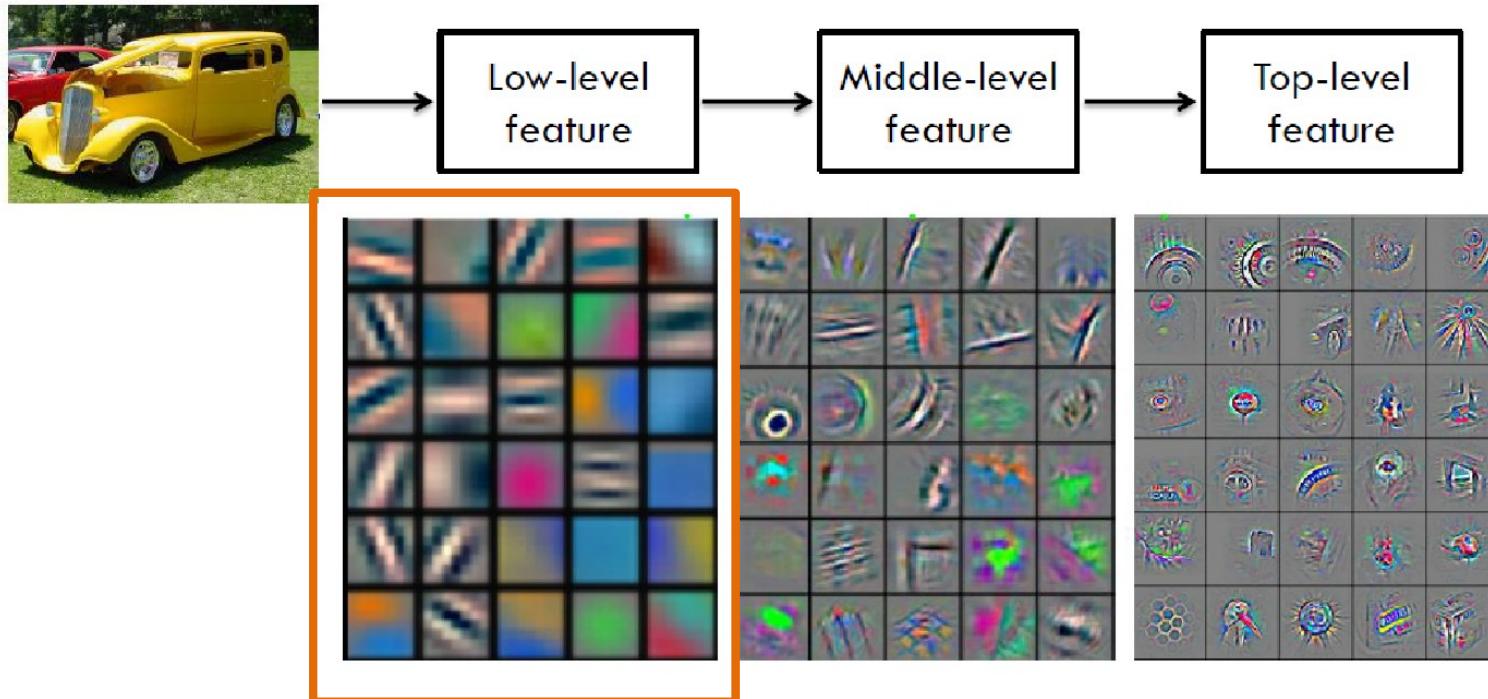


Small dataset



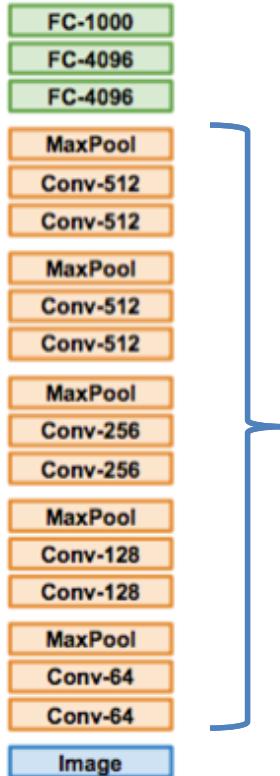
Use what has been  
learned for another  
setting

# Transfer Learning for Images



[Zeiler al., ECCV'14] Visualizing and Understanding Convolutional Networks

Trained on  
ImageNet

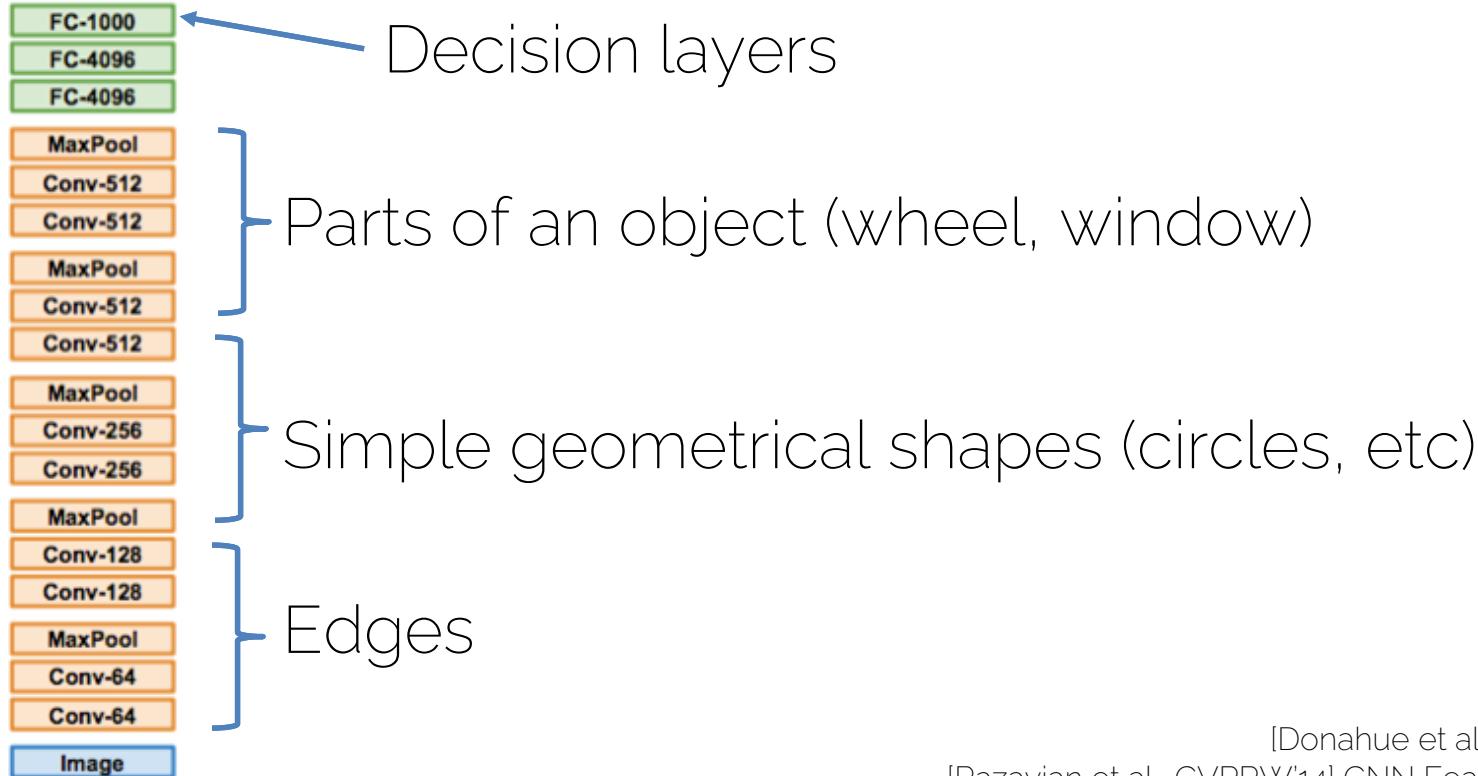


Feature  
extraction

[Donahue et al., ICML'14] DeCAF,  
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

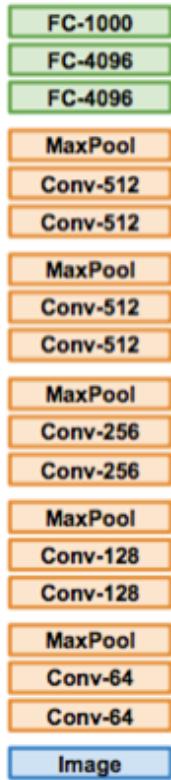
Trained on  
ImageNet

# Transfer Learning

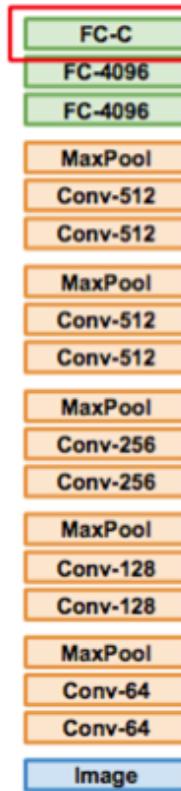


[Donahue et al., ICML'14] DeCAF,  
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

Trained on  
ImageNet



TRAIN



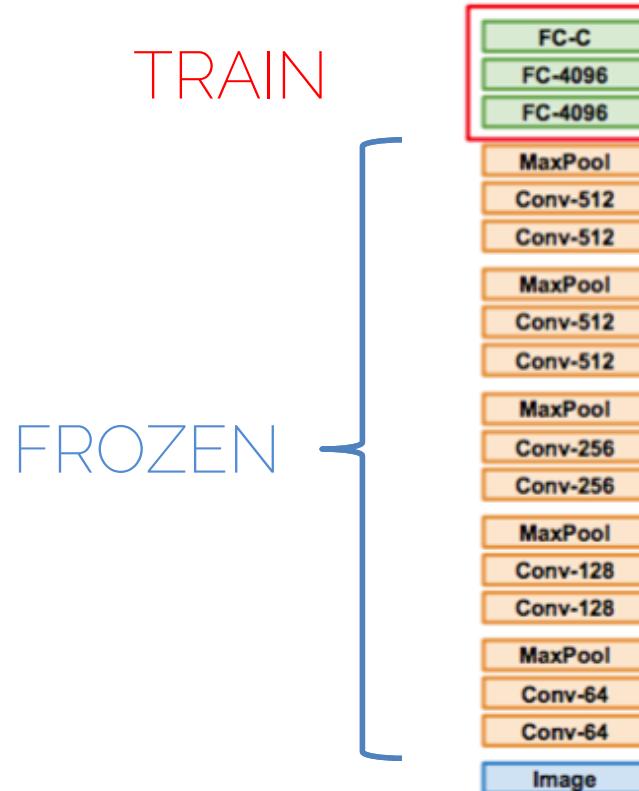
New dataset  
with C classes

FROZEN

[Donahue et al., ICML'14] DeCAF,  
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

# Transfer Learning

If the dataset is big enough train more layers with a low learning rate



# When Transfer Learning Makes Sense

- When task T1 and T2 have the same input (e.g. an RGB image)
- When you have more data for task T1 than for task T2
- When the low-level features for T1 could be useful to learn T2

# Now you are:

- Ready to perform image classification on any dataset
- Ready to design your own architecture
- Ready to deal with other problems such as semantic segmentation (Fully Convolutional Network)

# Representation Learning

# Learning Good Features

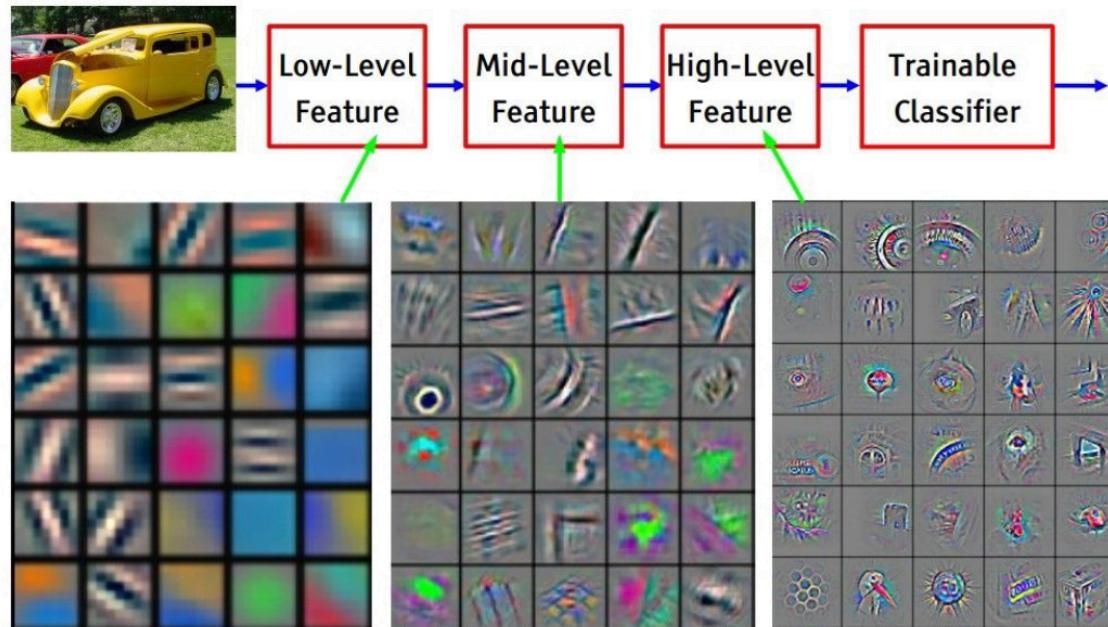
- Good features are essential for successful machine learning
- (Supervised) deep learning depends on training data used: input/target labels
- Change in inputs (noise, irregularities, etc) can result in drastically different results

# Representation Learning

- Allows for discovery of representations required for various tasks
- Deep representation learning: model maps input  $\mathbf{X}$  to output  $\mathbf{Y}$

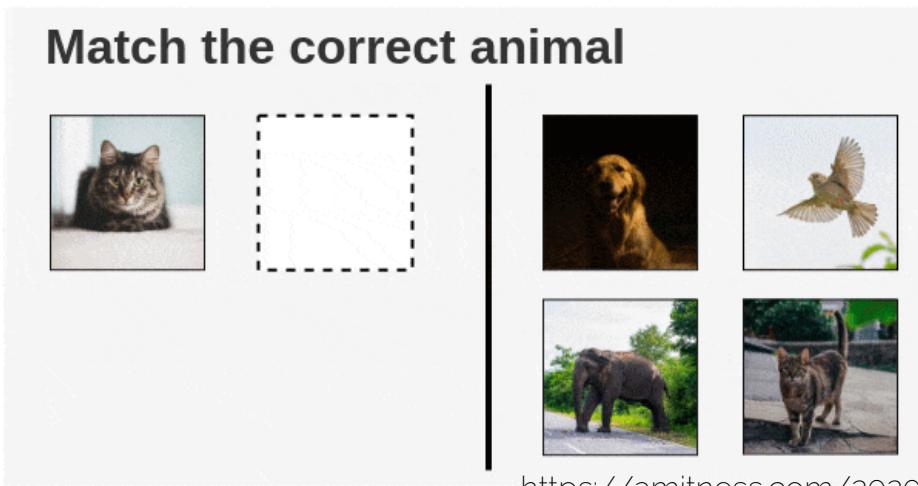
# Deep Representation Learning

- Intuitively, deep networks learn multiple levels of abstraction

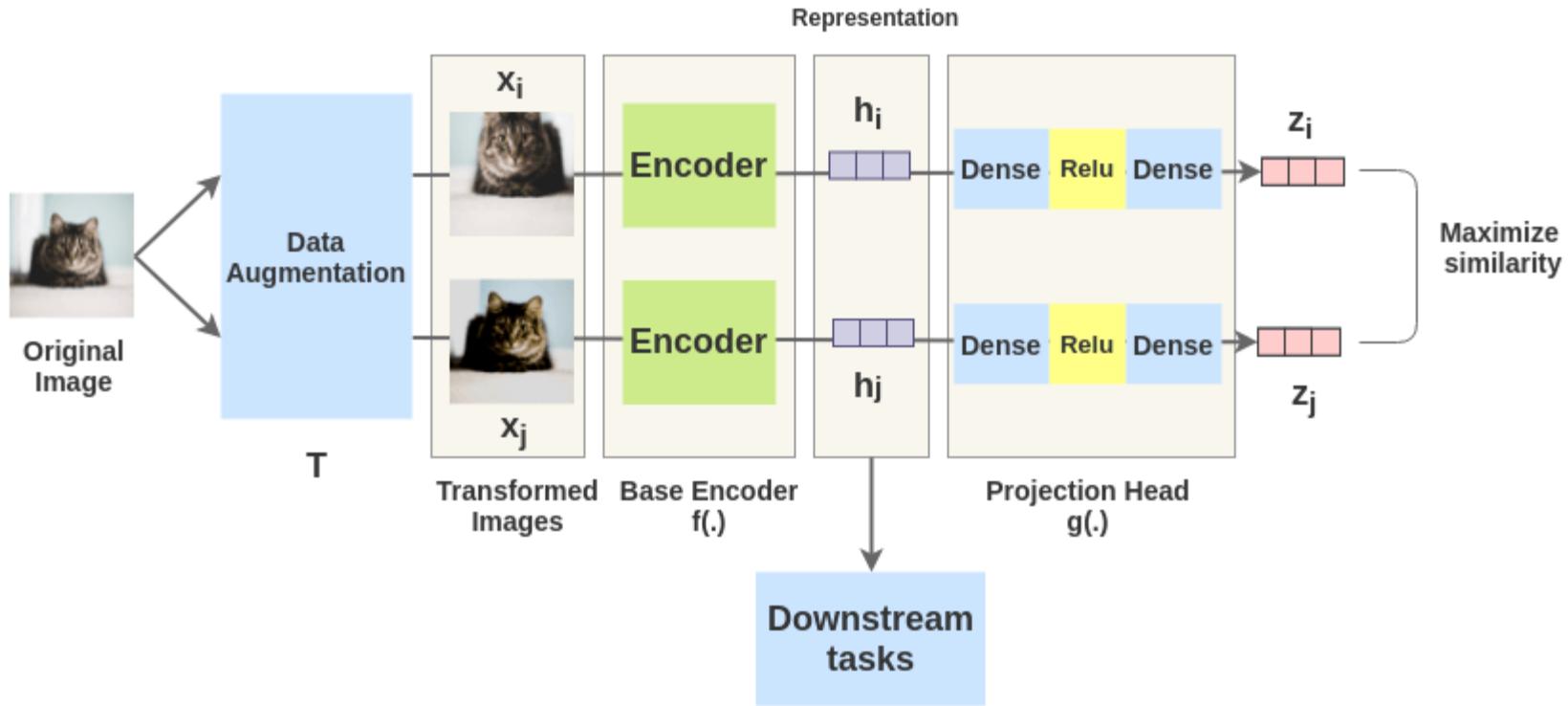


# How to Learn Good Features?

- Determine desired feature invariances
- Teach machines to distinguish between similar and dissimilar things

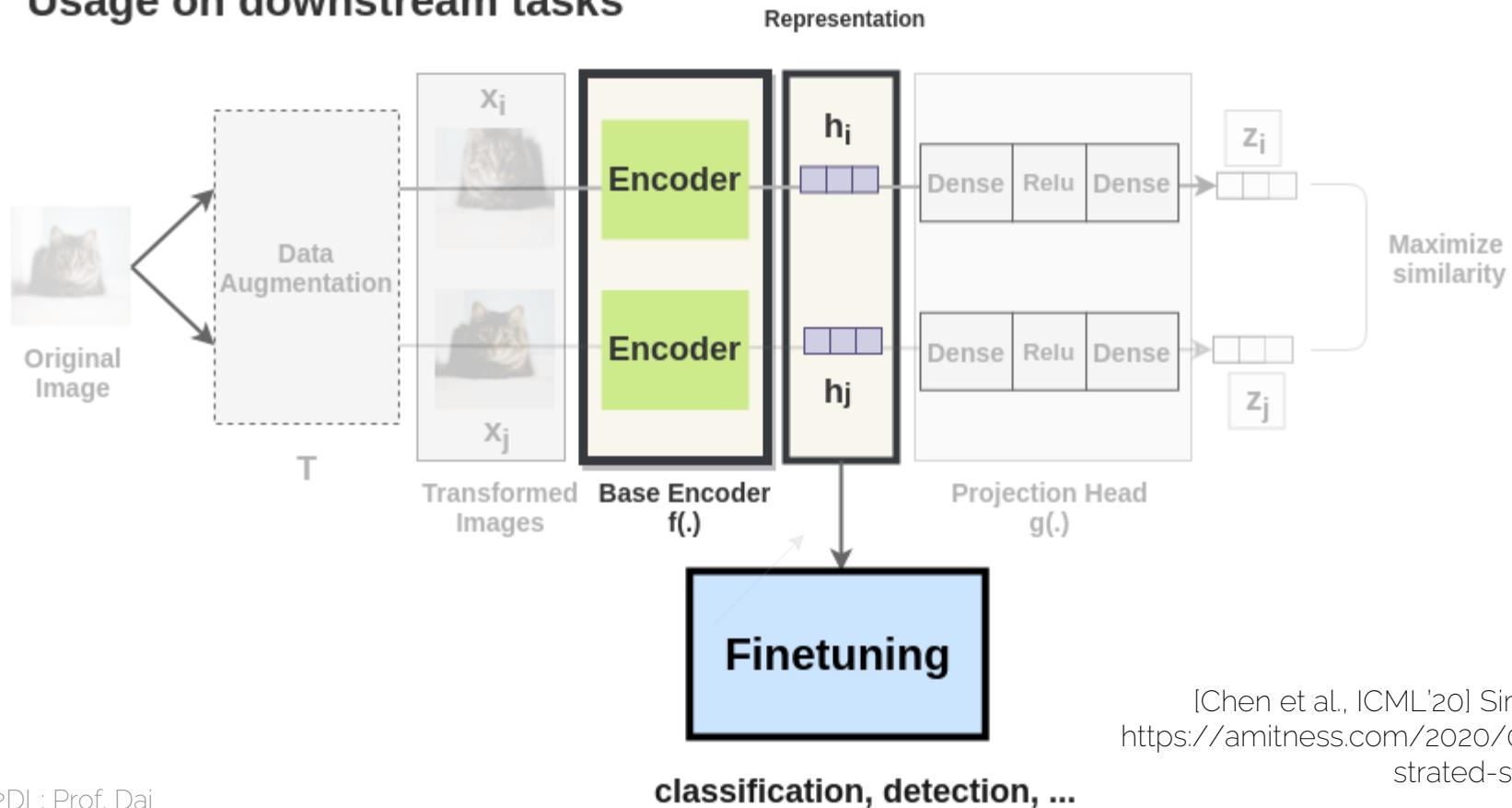


# How to Learn Good Features?



# Apply to Downstream Tasks

## Usage on downstream tasks



[Chen et al., ICML'20] SimCLR,  
<https://amitness.com/2020/03/illustrated-simclr/>  
25

# Transfer & Representation Learning

- Transfer learning can be done via representation learning
- Effectiveness of representation learning often demonstrated by transfer learning performance (but also other factors, e.g., smoothness of the manifold)

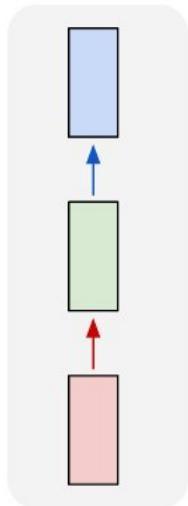
# Recurrent Neural Networks

# Processing Sequences

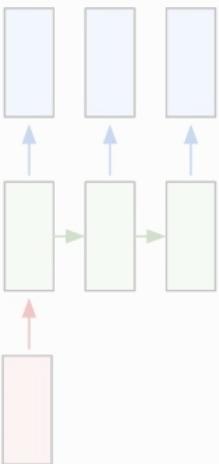
- Recurrent neural networks process sequence data
- Input/output can be sequences

# RNNs are Flexible

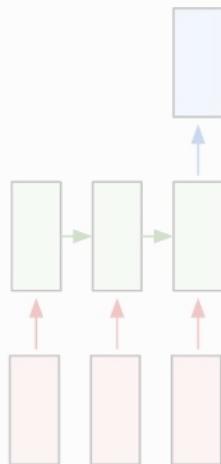
one to one



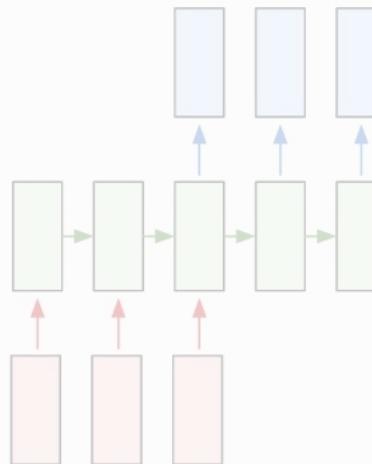
one to many



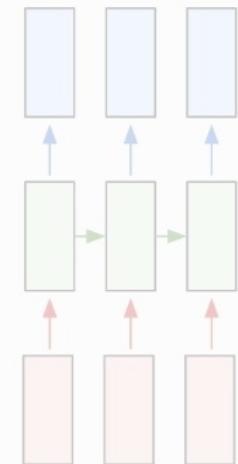
many to one



many to many



many to many



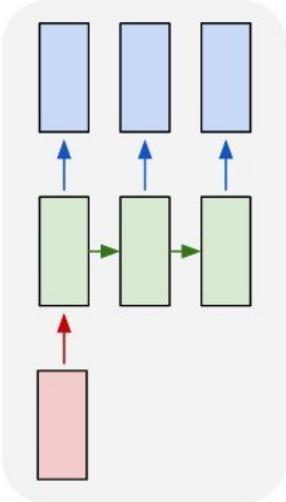
Classical neural networks for image classification

# RNNs are Flexible

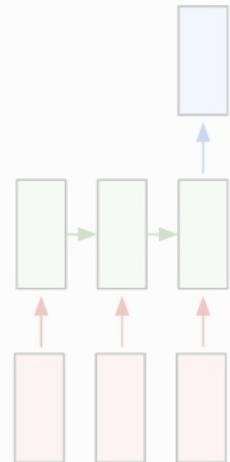
one to one



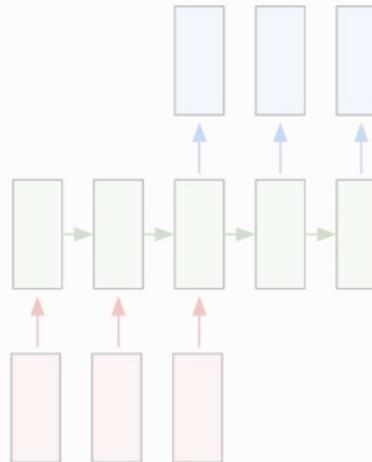
one to many



many to one



many to many



many to many

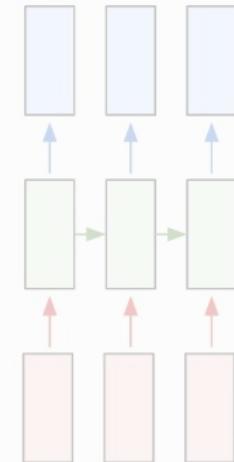


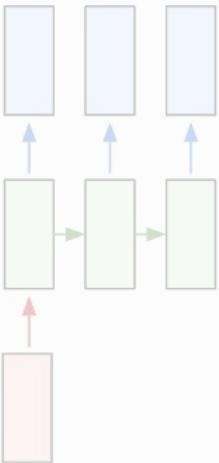
Image captioning

# RNNs are Flexible

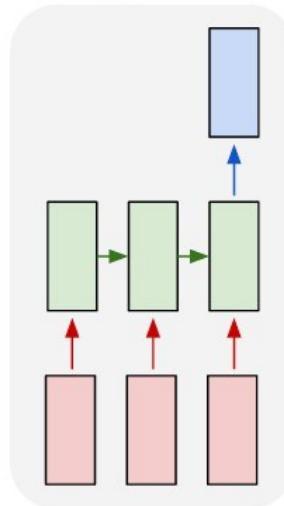
one to one



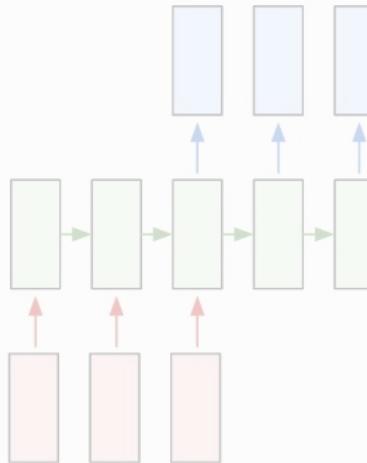
one to many



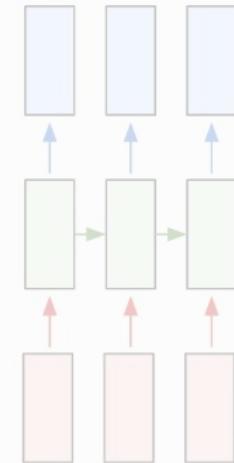
many to one



many to many



many to many



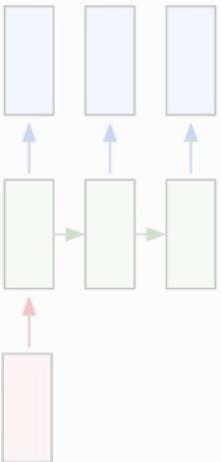
Language recognition

# RNNs are Flexible

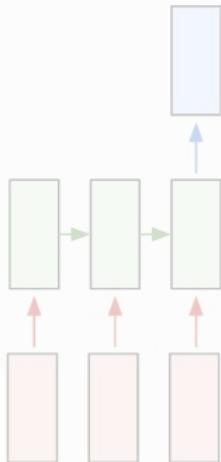
one to one



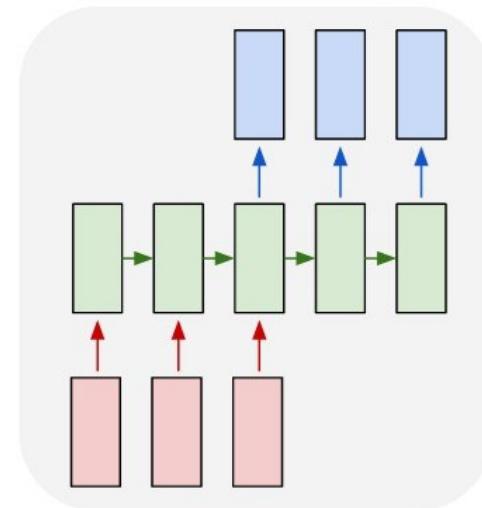
one to many



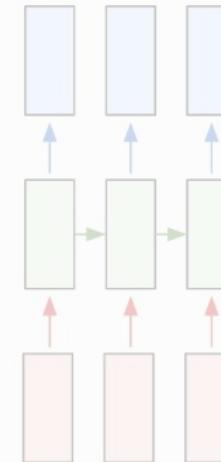
many to one



many to many



many to many



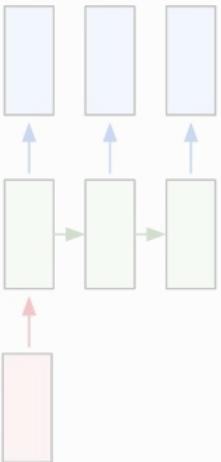
Machine translation

# RNNs are Flexible

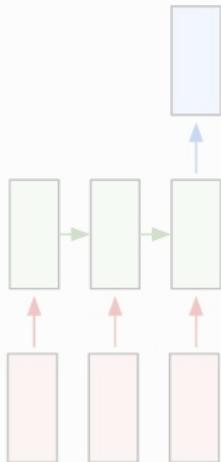
one to one



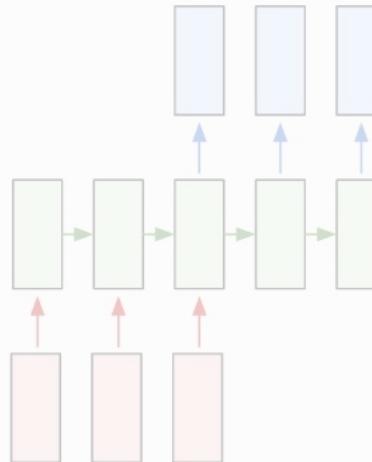
one to many



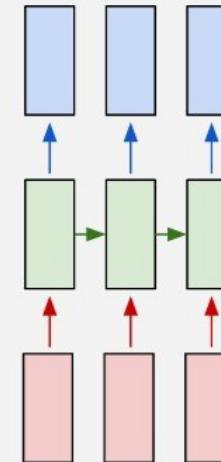
many to one



many to many



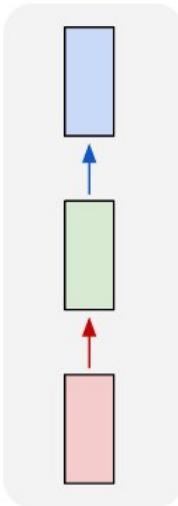
many to many



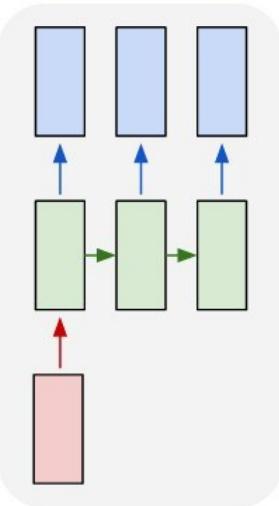
Event classification

# RNNs are Flexible

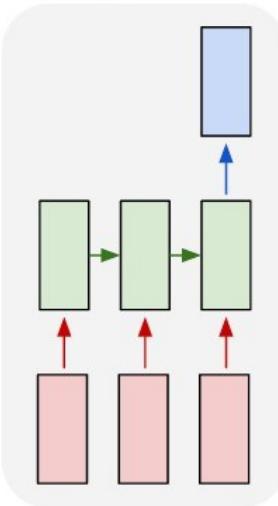
one to one



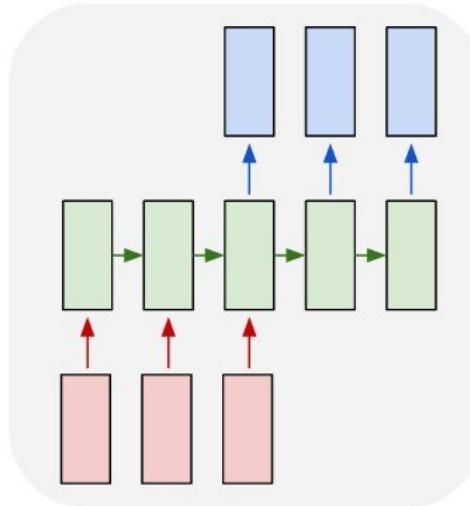
one to many



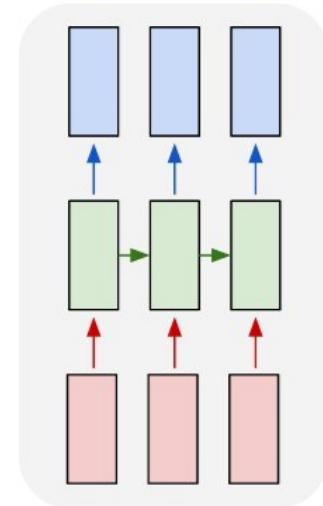
many to one



many to many



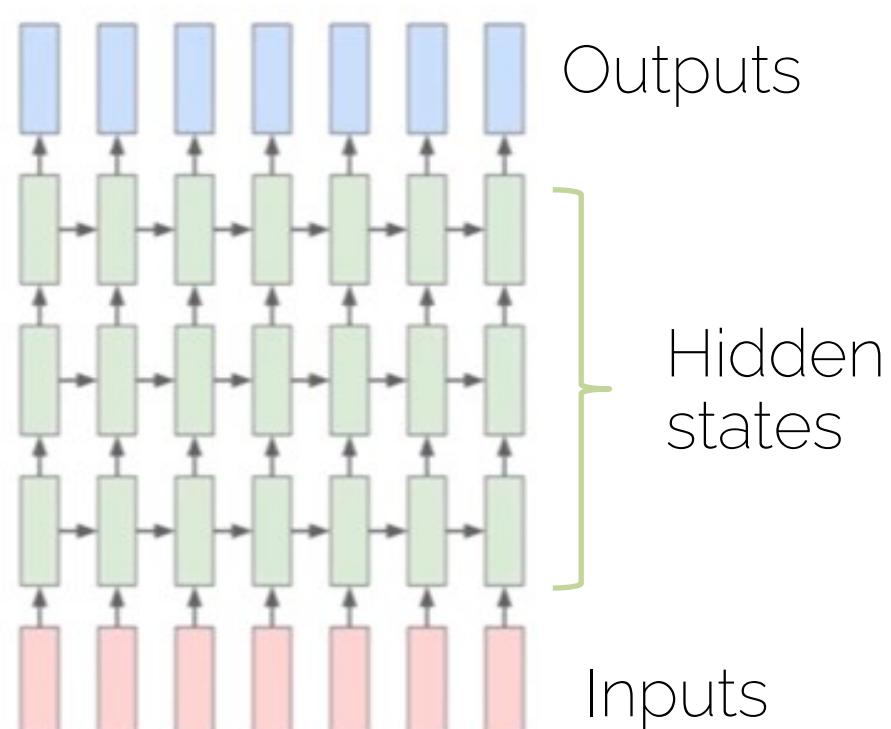
many to many



Event classification

# Basic Structure of an RNN

- Multi-layer RNN



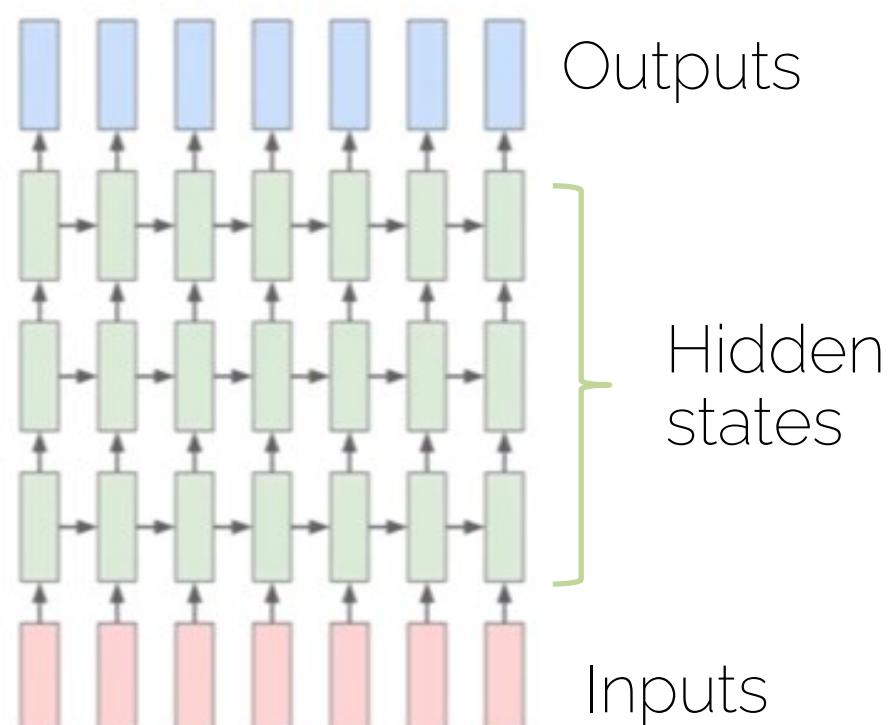
# Basic Structure of an RNN

- Multi-layer RNN

The hidden state  
will have its own  
internal dynamics

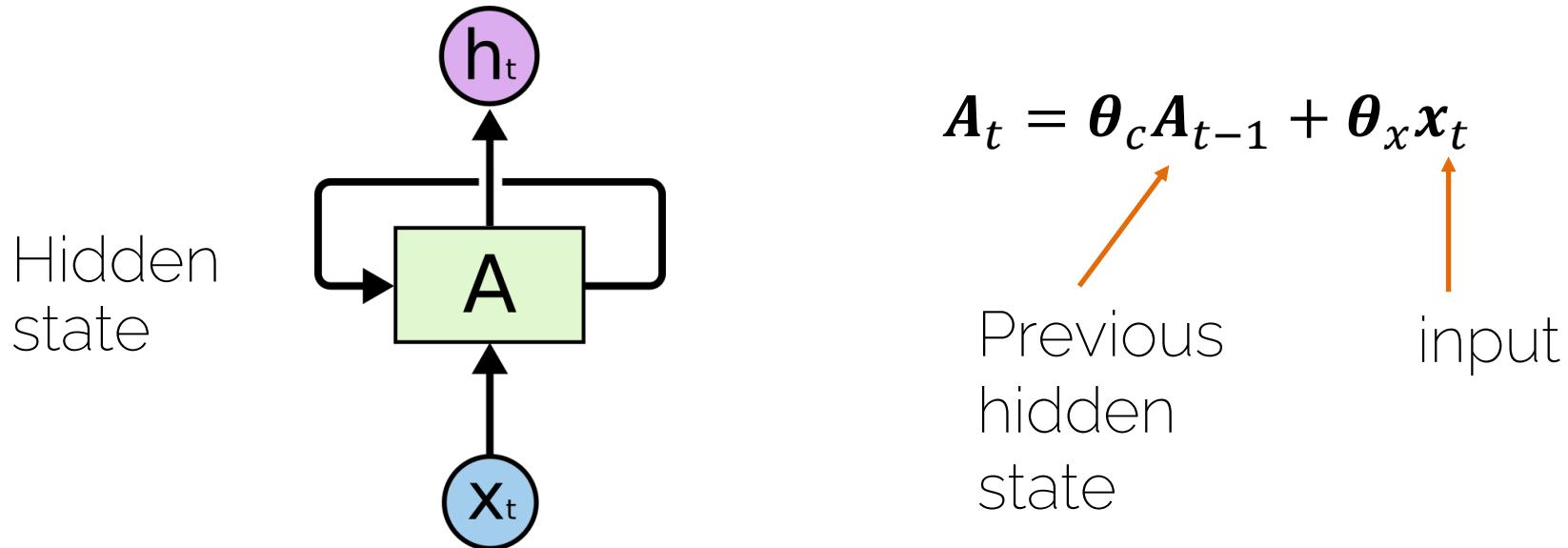


More expressive  
model!



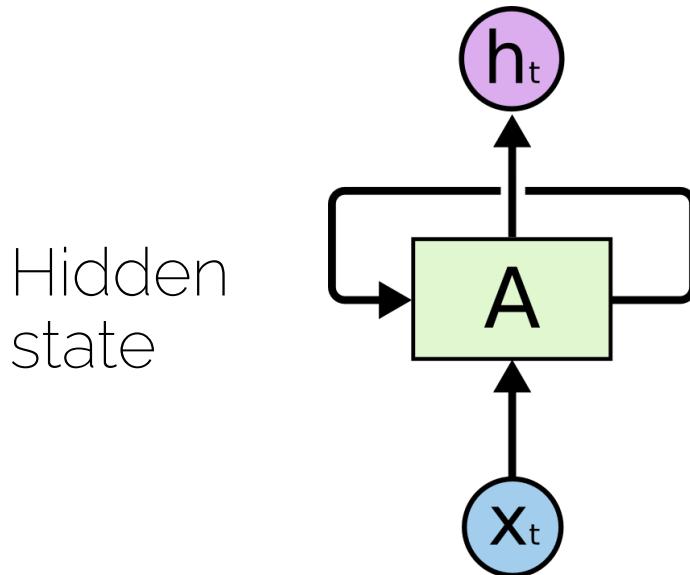
# Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



# Basic Structure of an RNN

- We want to have notion of “time” or “sequence”

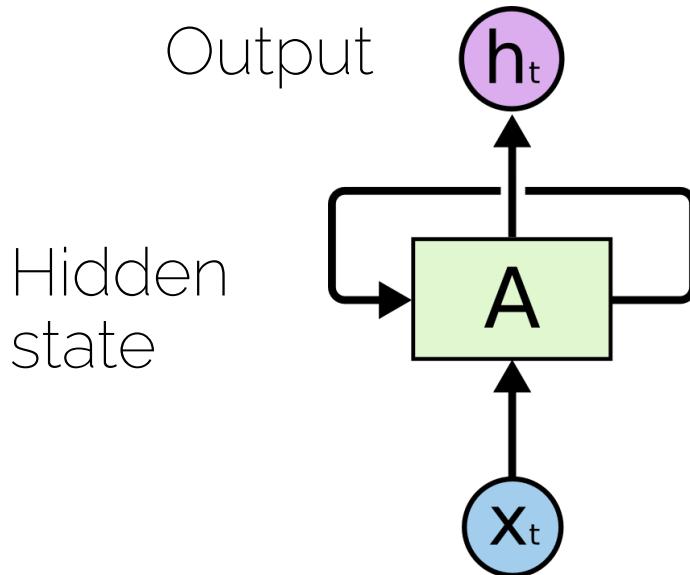


$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

Parameters to be learned

# Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



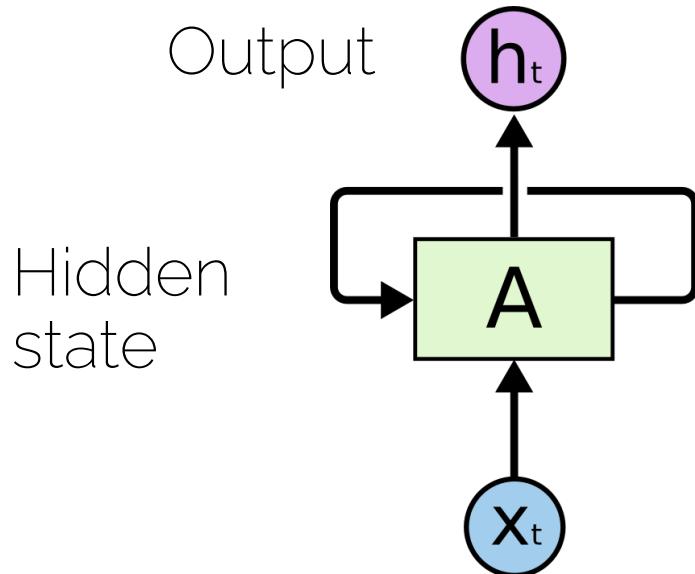
$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

$$h_t = \theta_h A_t$$

Note: non-linearities  
ignored for now

# Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

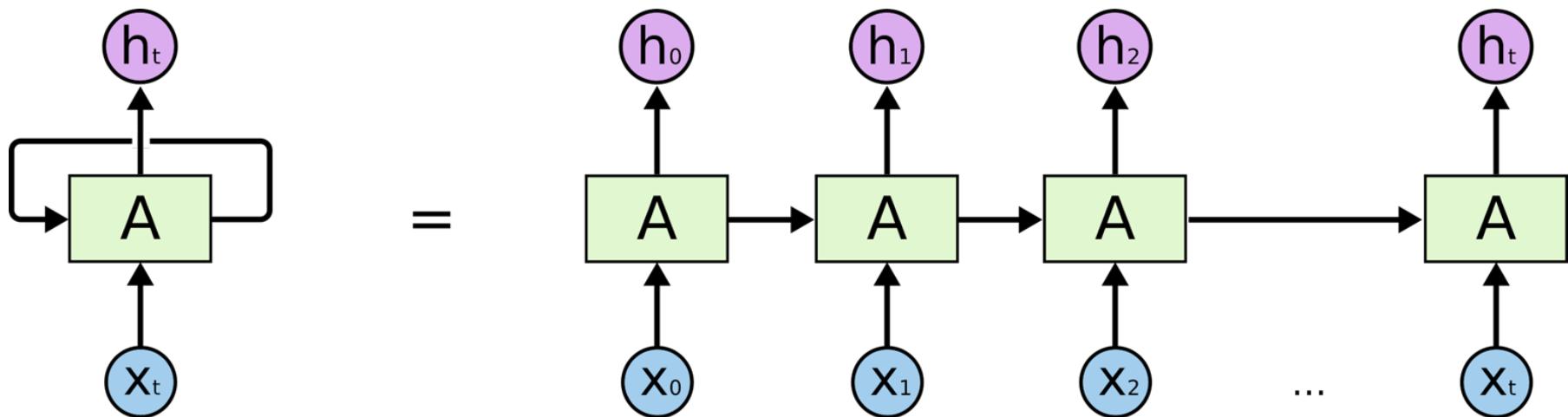
$$h_t = \theta_h A_t$$

Same parameters for each time step = generalization!

# Basic Structure of an RNN

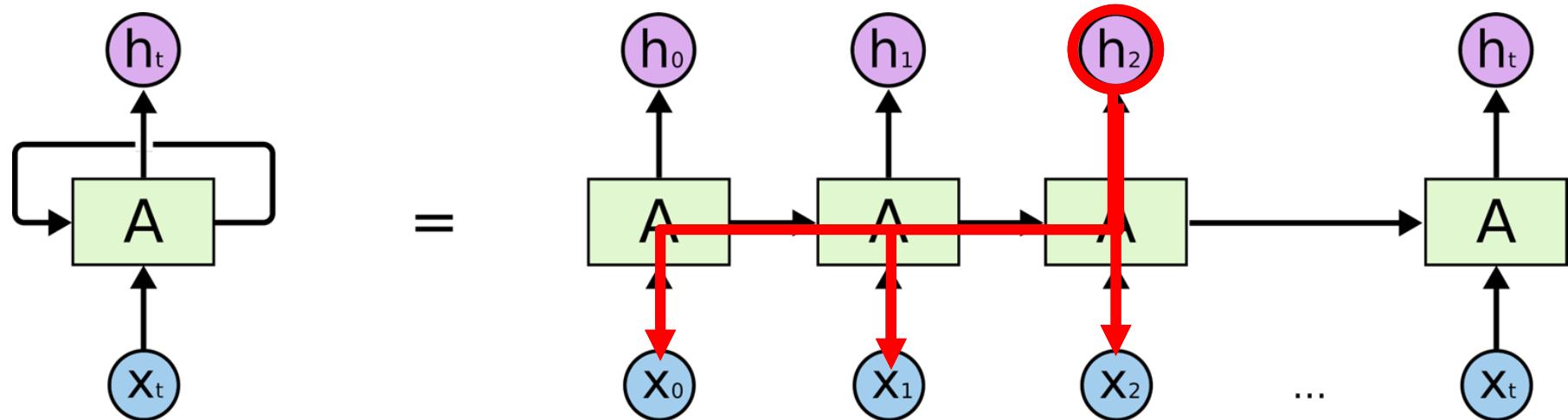
- Unrolling RNNs

Same function for the hidden layers



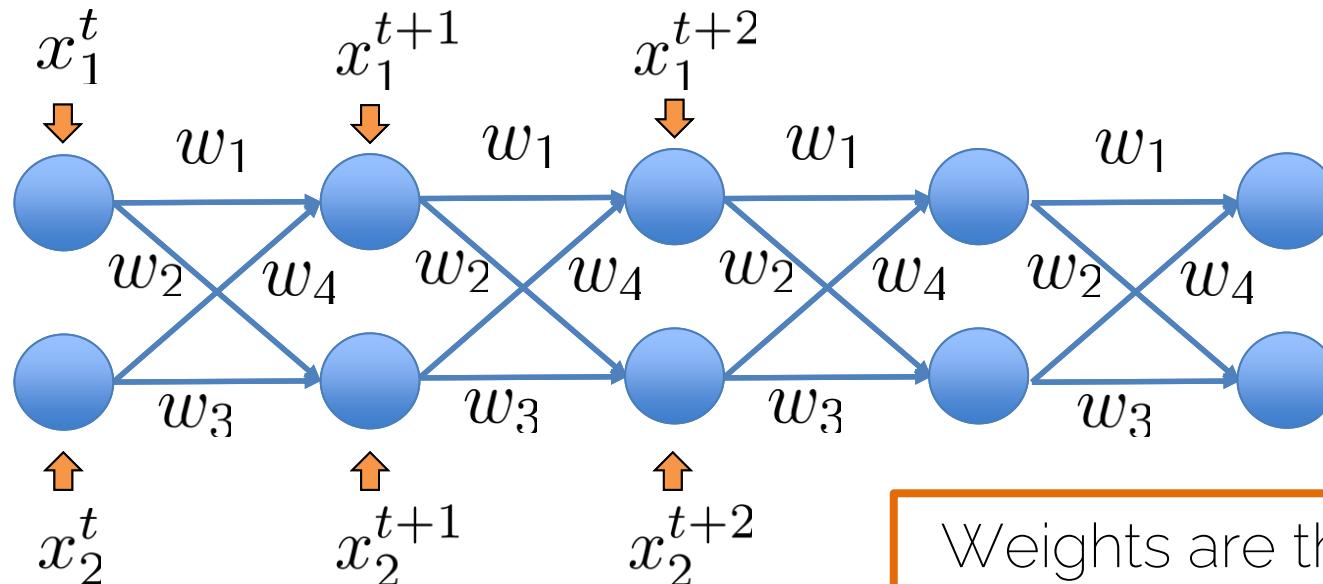
# Basic Structure of an RNN

- Unrolling RNNs



# Basic Structure of an RNN

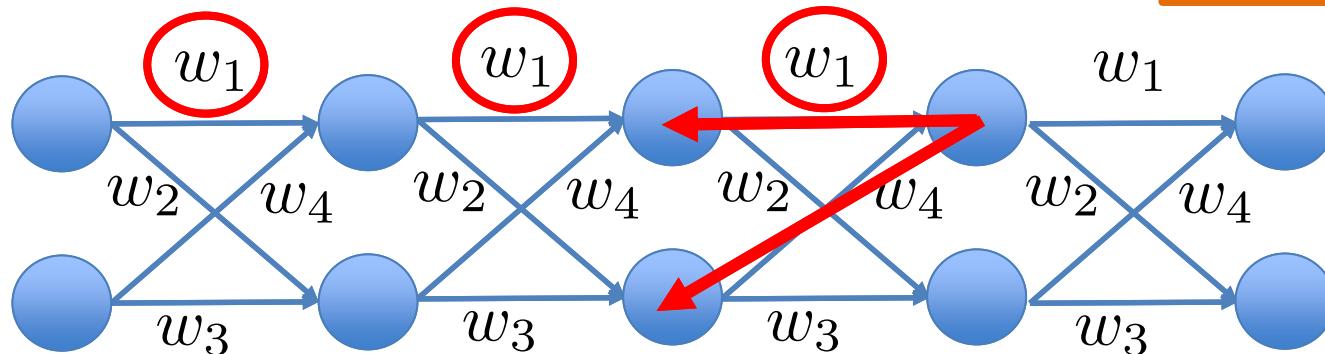
- Unrolling RNNs as feedforward nets



# Backprop through an RNN

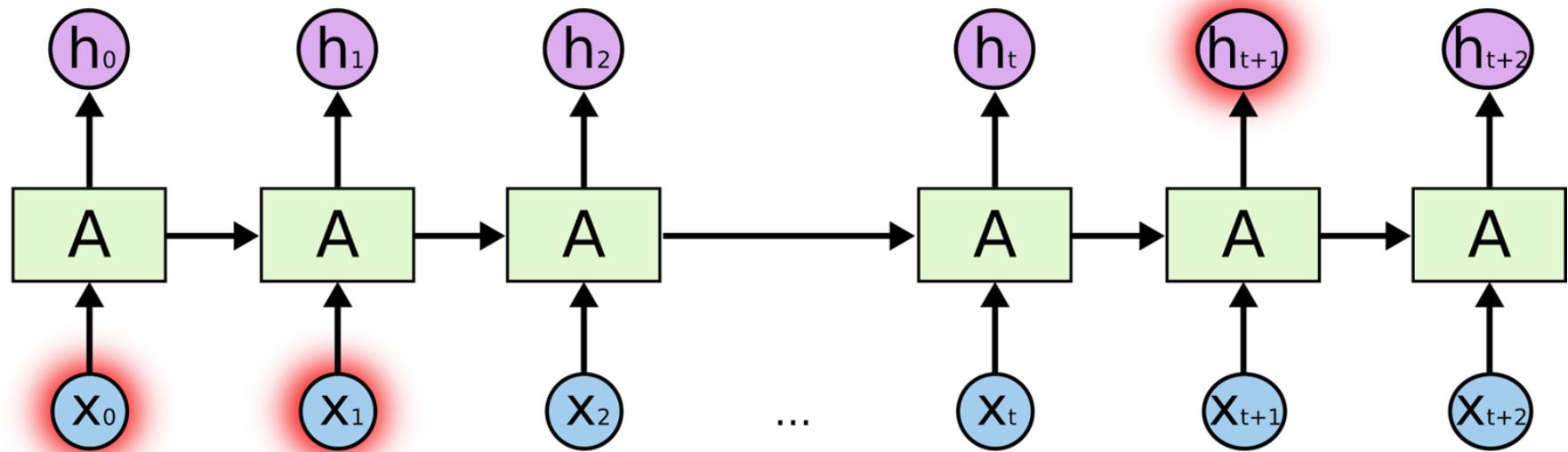
- Unrolling RNNs as feedforward nets

Chain rule



Add the derivatives at different times for each weight

# Long-term Dependencies



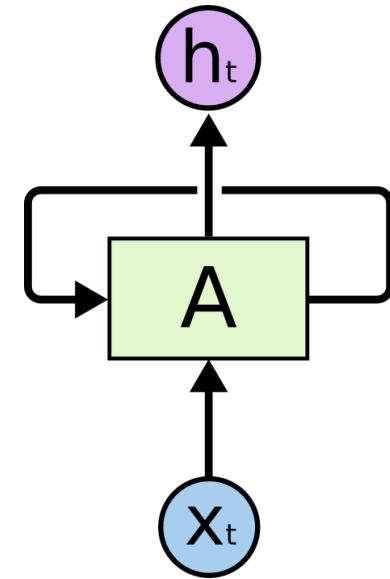
I moved to Germany ...

so I speak German fluently.

# Long-term Dependencies

- Simple recurrence  $\mathbf{A}_t = \theta_c \mathbf{A}_{t-1} + \theta_x \mathbf{x}_t$
- Let us forget the input  $\mathbf{A}_t = \theta_c^t \mathbf{A}_0$

Same weights are multiplied over and over again



# Long-term Dependencies

- Simple recurrence  $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$

What happens to small weights?

Vanishing gradient

What happens to large weights?

Exploding gradient

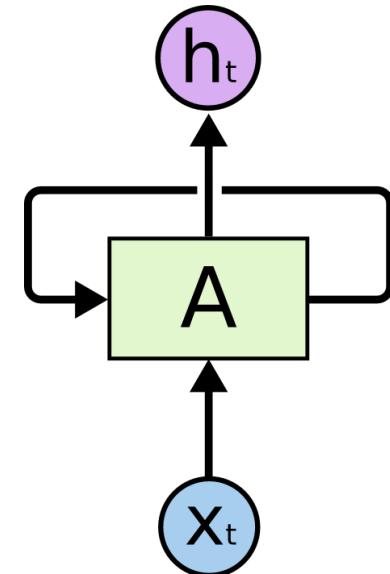
# Long-term Dependencies

- Simple recurrence  $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$
- If  $\boldsymbol{\theta}$  admits eigendecomposition

$$\boldsymbol{\theta} = \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^T$$

Matrix of  
eigenvectors

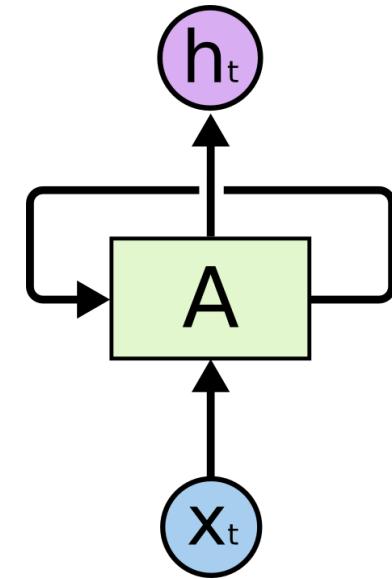
Diagonal of this  
matrix are the  
eigenvalues



# Long-term Dependencies

- Simple recurrence  $\mathbf{A}_t = \boldsymbol{\theta}^t \mathbf{A}_0$
- If  $\boldsymbol{\theta}$  admits eigendecomposition

$$\boldsymbol{\theta} = \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^T$$



- Orthogonal  $\boldsymbol{\theta}$  allows us to simplify the recurrence

$$\mathbf{A}_t = \mathbf{Q} \boldsymbol{\Lambda}^t \mathbf{Q}^T \mathbf{A}_0$$

# Long-term Dependencies

- Simple recurrence  $\mathbf{A}_t = \mathbf{Q}\boldsymbol{\Lambda}^t\mathbf{Q}^T\mathbf{A}_0$

What happens to eigenvalues with magnitude less than one?

Vanishing gradient

What happens to eigenvalues with magnitude larger than one?

Exploding gradient



Gradient clipping

# Long-term Dependencies

- Simple recurrence

$$\mathbf{A}_t = \theta_c^t \mathbf{A}_0$$

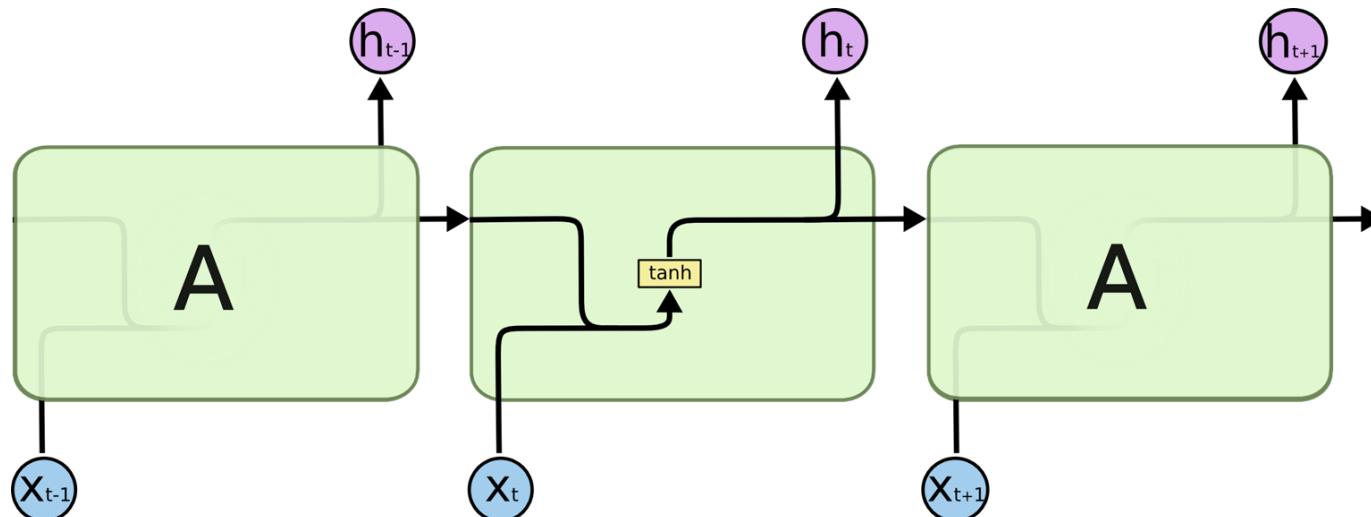
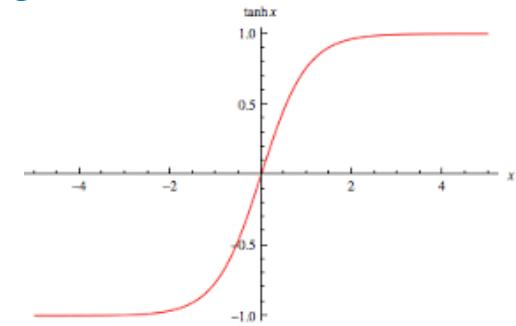


Let us just make a matrix with eigenvalues = 1

Allow the **cell** to maintain its "state"

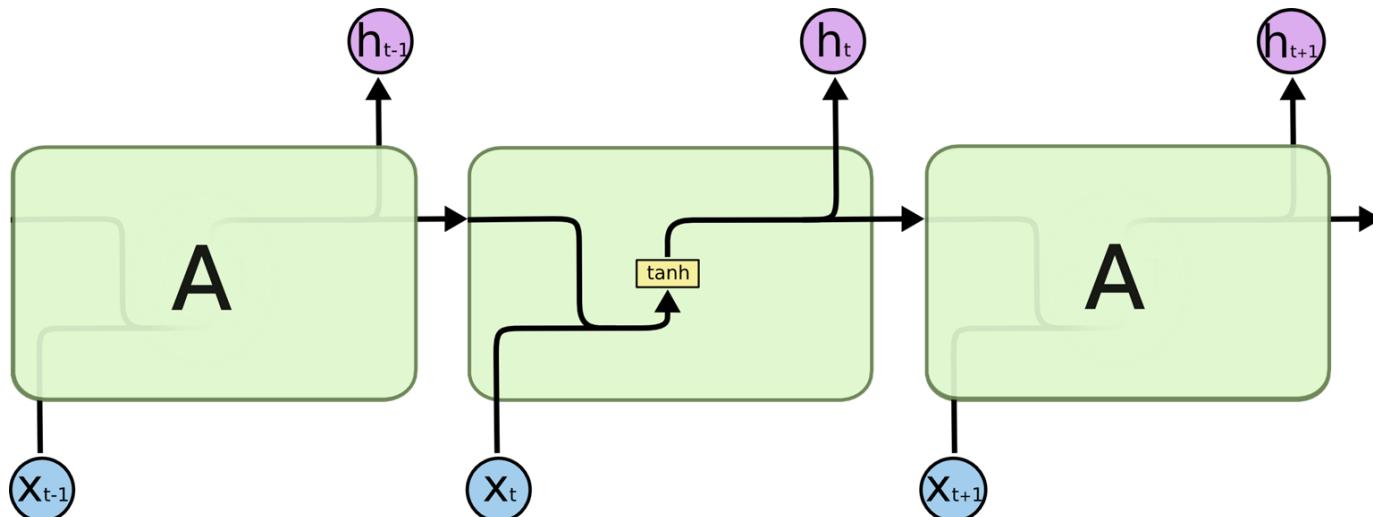
# Vanishing Gradient

- 1. From the weights  $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$
- 2. From the activation functions (*tanh*)



# Vanishing Gradient

- 1. From the weights  $A_t = \cancel{\theta^t} A_0$   
1
- 2. From the activation functions ( $\tanh$ ) ?

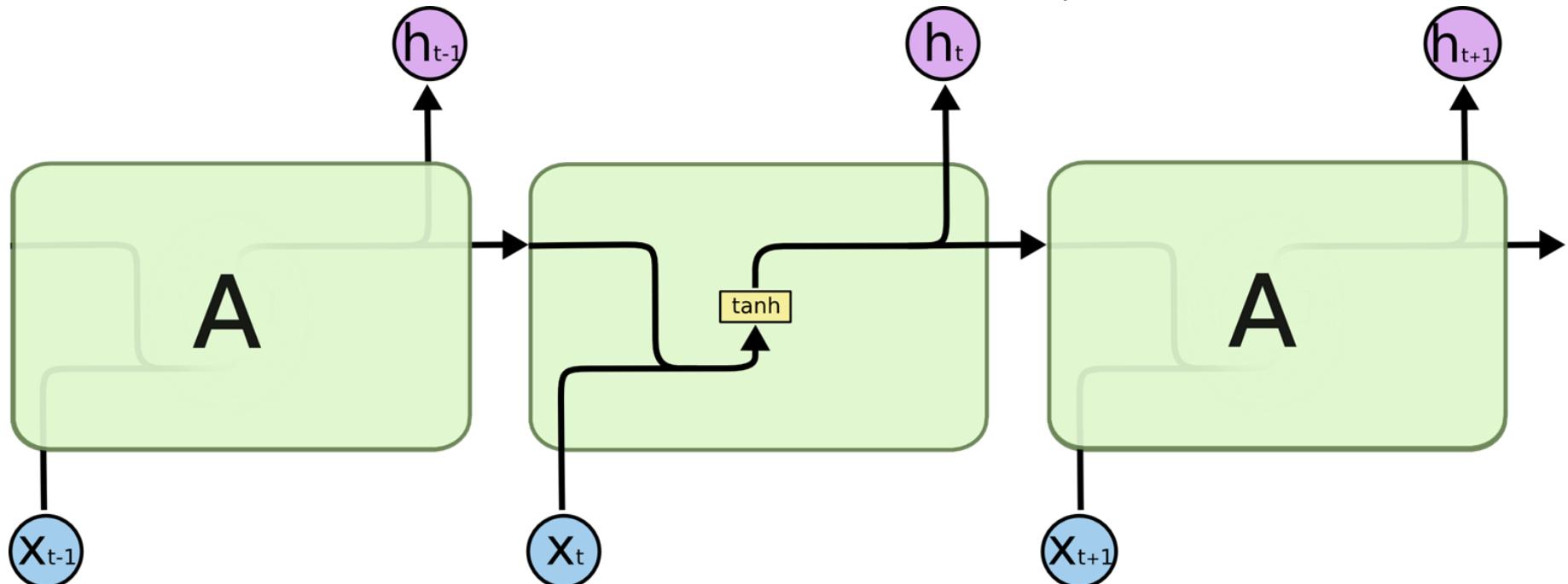


# Long Short Term Memory

[Hochreiter et al., Neural Computation'97] Long Short-Term Memory

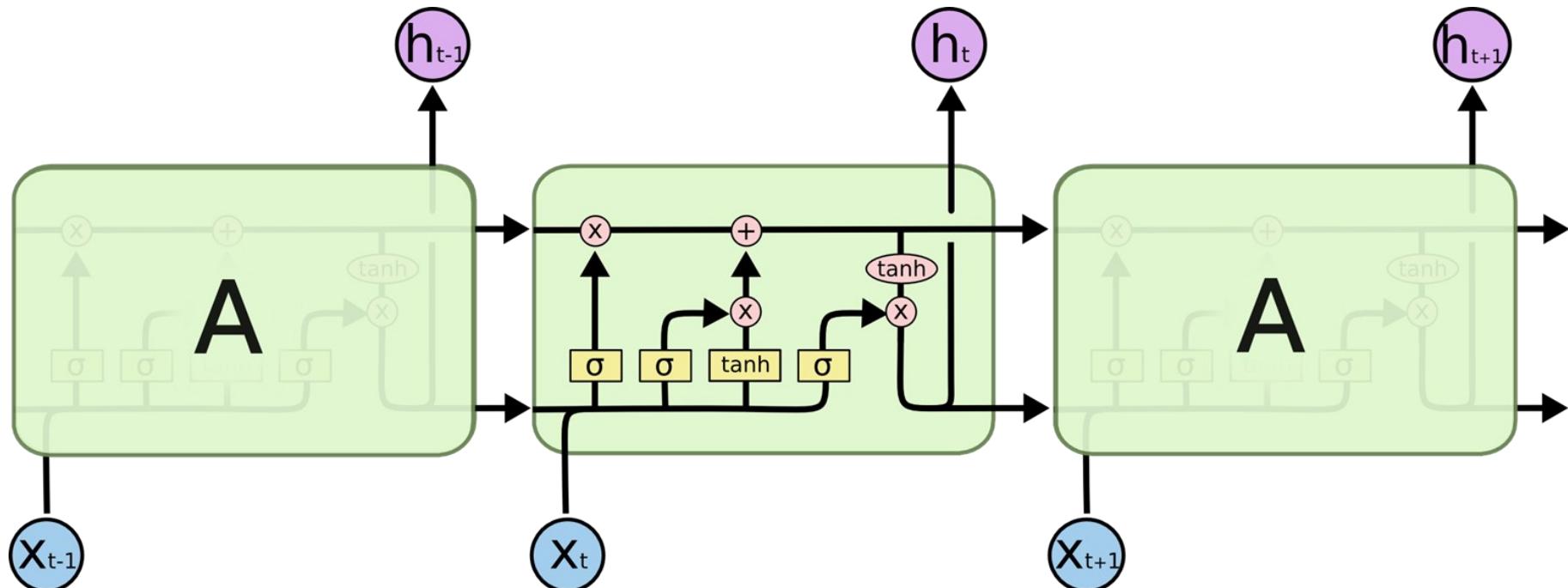
# Long-Short Term Memory Units

- Simple RNN has **tanh** as non-linearity



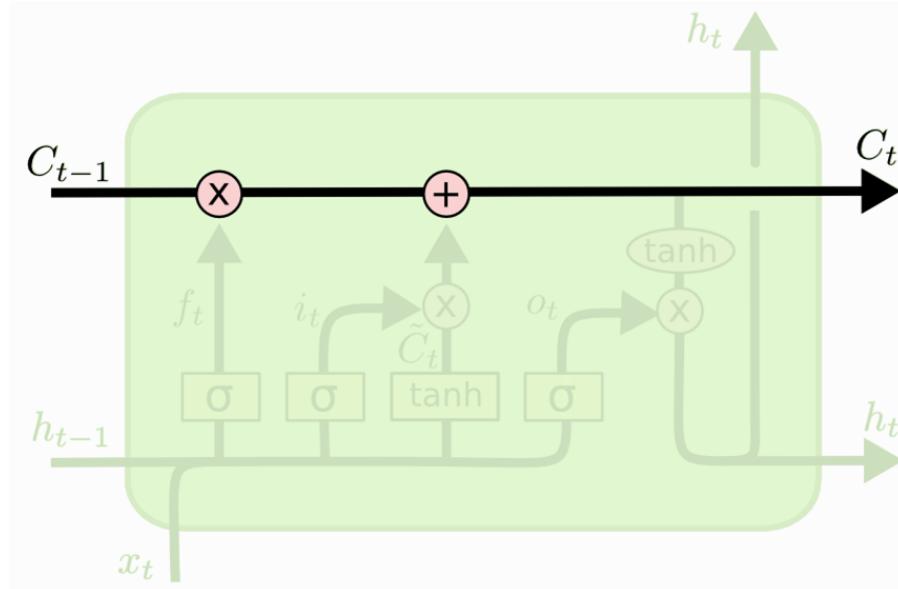
# Long-Short Term Memory Units

LSTM



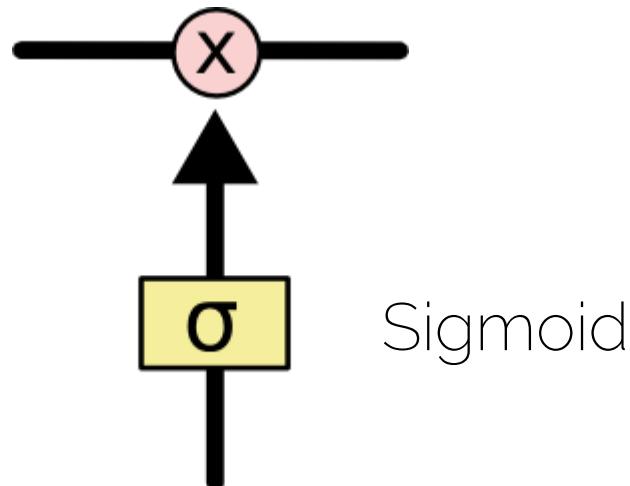
# Long-Short Term Memory Units

- Key ingredients
- Cell = transports the information through the unit



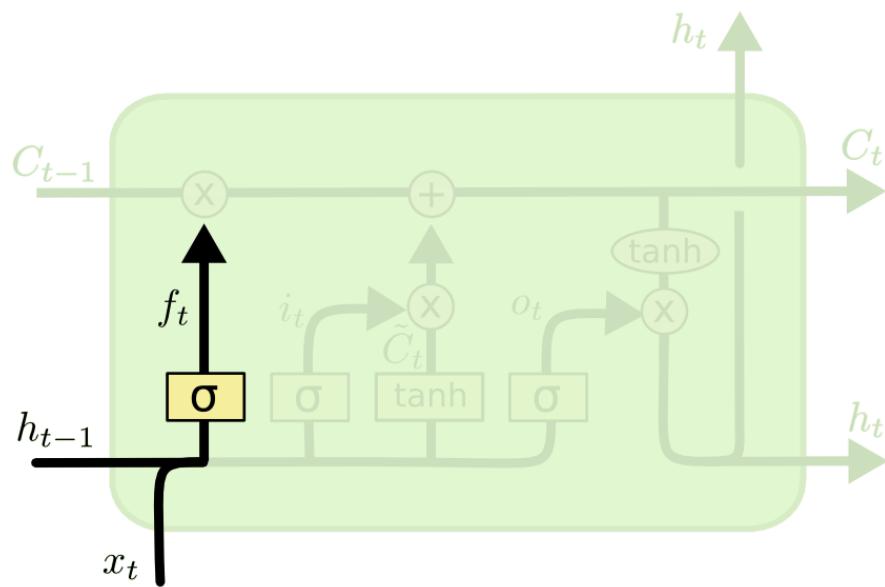
# Long-Short Term Memory Units

- Key ingredients
- Cell = transports the information through the unit
- Gate = remove or add information to the cell state



# LSTM: Step by Step

- Forget gate  $f_t = \text{sigm}(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$

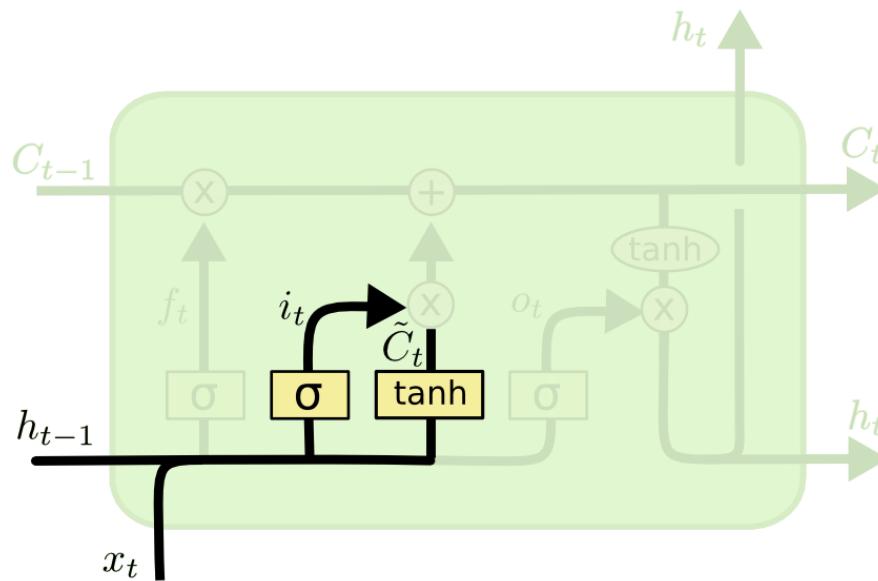


Decides when to  
erase the cell state

Sigmoid = output  
between **0** (forget)  
and **1** (keep)

# LSTM: Step by Step

- Input gate  $i_t = \text{sigm}(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$

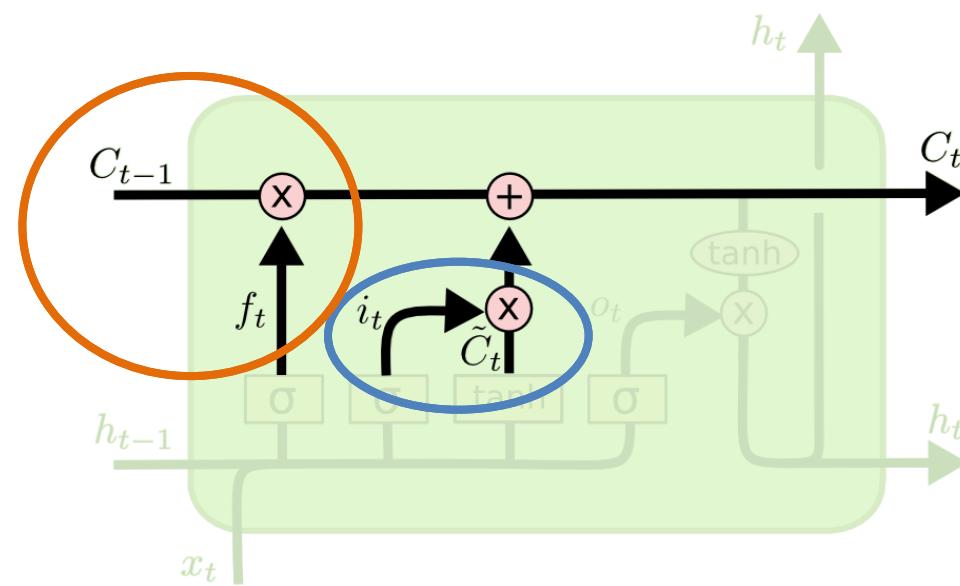


Decides which values will be updated

New cell state, output from a **tanh** (-1,1)

# LSTM: Step by Step

- Element-wise operations



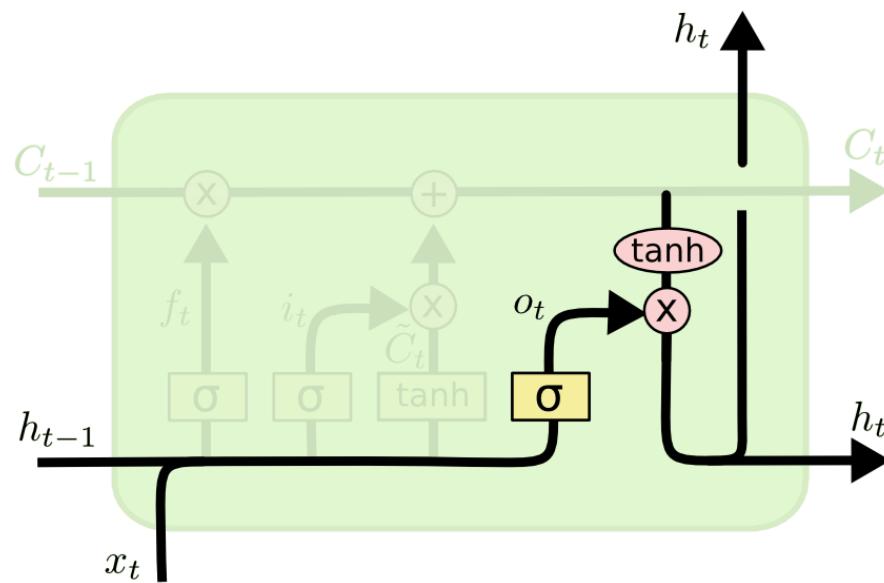
$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

Previous states

Current state

# LSTM: Step by Step

- Output gate  $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$



Decides which values will be outputted

Output from a tanh  $(-1, 1)$

# LSTM: Step by Step

- Forget gate  $f_t = \text{sigm}(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$
- Input gate  $i_t = \text{sigm}(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$
- Output gate  $o_t = \text{sigm}(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o)$
- Cell update  $g_t = \tanh(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g)$
- Cell  $C_t = f_t \odot C_{t-1} + i_t \odot g_t$
- Output  $h_t = o_t \odot \tanh(C_t)$

# LSTM: Step by Step

- Forget gate
- Input gate
- Output gate
- Cell update
- Cell
- Output

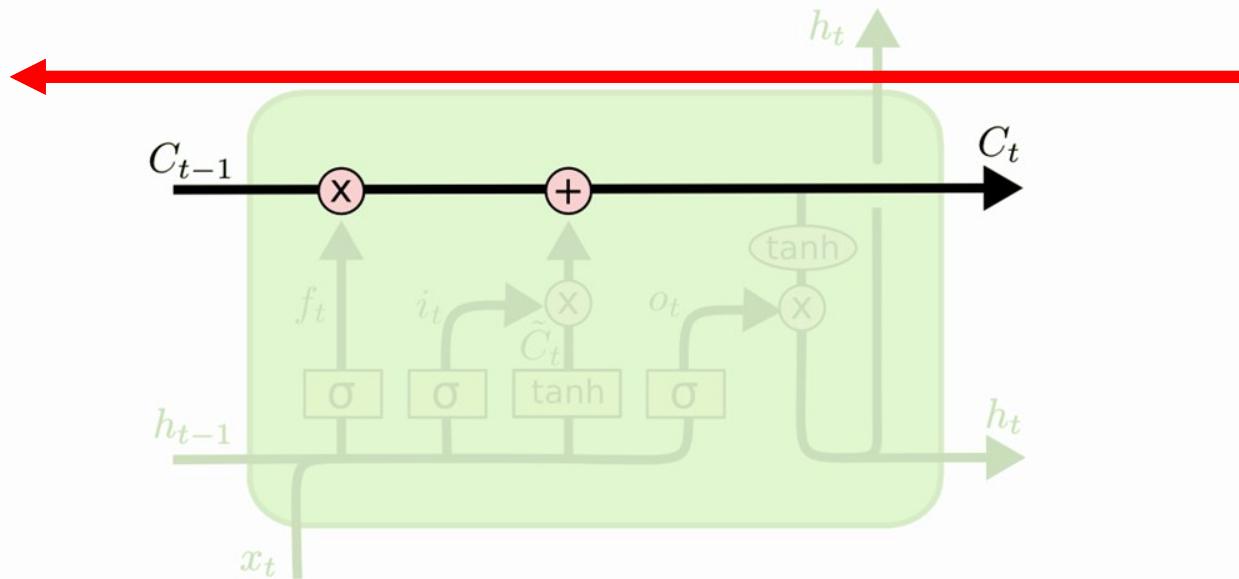
$$\begin{aligned} f_t &= \text{sigm}(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f) \\ i_t &= \text{sigm}(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i) \\ o_t &= \text{sigm}(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o) \\ g_t &= \tanh(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g) \\ C_t &= f_t \odot C_{t-1} + i_t \odot g_t \end{aligned}$$

$$h_t = o_t \odot \tanh(C_t)$$

Learned through  
backpropagation

# LSTM

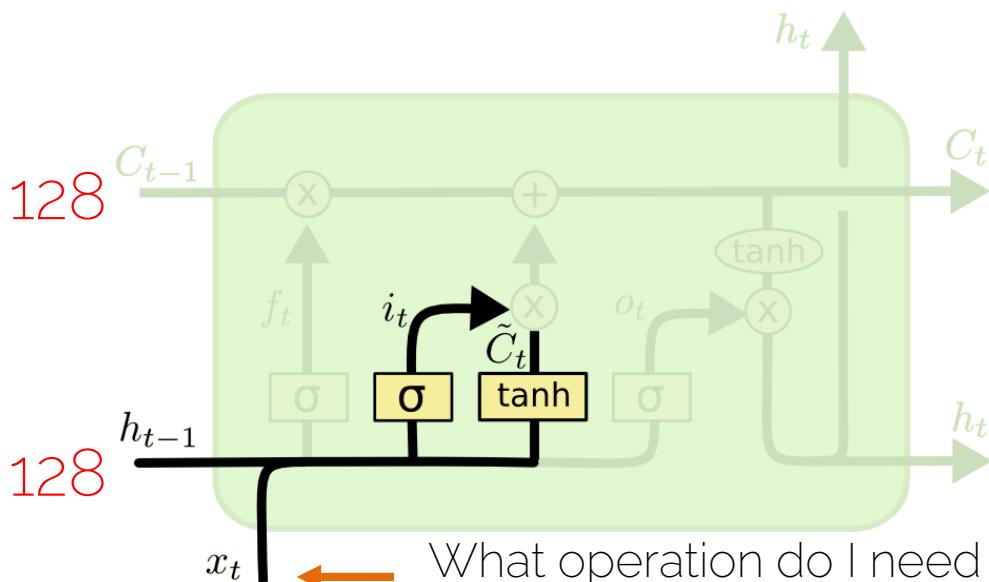
- Highway for the gradient to flow



# LSTM: Dimensions

- Cell update

$$128 \quad 128 \quad 128 \\ \mathbf{g}_t = \tanh(\theta_{xg} \mathbf{x}_t + \theta_{hg} \mathbf{h}_{t-1} + \mathbf{b}_g)$$



When coding an LSTM, we have to define the size of the hidden state

Dimensions need to match

What operation do I need to do to my input to get a 128 vector representation?

# LSTM in code

```
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
    """
    Forward pass for a single timestep of an LSTM.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data, of shape (N, D)
    - prev_h: Previous hidden state, of shape (N, H)
    - prev_c: previous cell state, of shape (N, H)
    - Wx: Input-to-hidden weights, of shape (D, 4H)
    - Wh: Hidden-to-hidden weights, of shape (H, 4H)
    - b: Biases, of shape (4H)

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    - next_c: Next cell state, of shape (N, H)
    - cache: Tuple of values needed for backward pass.
    """
    next_h, next_c, cache = None, None, None

    N, H = prev_h.shape
    # 1
    a = np.dot(x, Wx) + np.dot(prev_h, Wh) + b

    # 2
    ai = a[:, :H]
    af = a[:, H:2*H]
    ao = a[:, 2*H:3*H]
    ag = a[:, 3*H:]

    # 3
    i = sigmoid(ai)
    f = sigmoid(af)
    o = sigmoid(ao)
    g = np.tanh(ag)

    # 4
    next_c = f * prev_c + i * g

    # 5
    next_h = o * np.tanh(next_c)

    cache = i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h
    return next_h, next_c, cache|
```

```
def lstm_step_backward(dnext_h, dnext_c, cache):
    """
    Backward pass for a single timestep of an LSTM.

    Inputs:
    - dnext_h: Gradients of next hidden state, of shape (N, H)
    - dnext_c: Gradients of next cell state, of shape (N, H)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient of input data, of shape (N, D)
    - dprev_h: Gradient of previous hidden state, of shape (N, H)
    - dprev_c: Gradient of previous cell state, of shape (N, H)
    - dWx: Gradient of input-to-hidden weights, of shape (D, 4H)
    - dWh: Gradient of hidden-to-hidden weights, of shape (H, 4H)
    - db: Gradient of biases, of shape (4H)
    """
    dx, dh, dc, dWx, dWh, db = None, None, None, None, None, None
    i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h = cache

    # backprop into step 5
    do = np.tanh(next_c) * dnext_h
    dnext_c += o * (1 - np.tanh(next_c) ** 2) * dnext_h

    # backprop into 4
    df = prev_c * dnext_c
    dprev_c = f * dnext_c
    di = g * dnext_c
    dg = i * dnext_c

    # backprop into 3
    dai = sigmoid(ai) * (1 - sigmoid(ai)) * di
    daf = sigmoid(af) * (1 - sigmoid(af)) * df
    dao = sigmoid(ao) * (1 - sigmoid(ao)) * do
    dag = (1 - np.tanh(ag) ** 2) * dg

    # backprop into 2
    da = np.hstack((dai, daf, dao, dag))

    # backprop into 1
    db = np.sum(da, axis = 0)
    dprev_h = np.dot(Wh, da.T).T
    dWh = np.dot(prev_h.T, da)
    dx = np.dot(da, Wx.T)
    dWx = np.dot(x.T, da)

    return dx, dprev_h, dprev_c, dWx, dWh, db|
```

# Attention

# Attention is all you need

---

## Attention Is All You Need

---

**Ashish Vaswani\***

Google Brain

avaswani@google.com

**Noam Shazeer\***

Google Brain

noam@google.com

**Niki Parmar\***

Google Research

nikip@google.com

**Jakob Uszkoreit\***

Google Research

usz@google.com

**Llion Jones\***

Google Research

llion@google.com

**Aidan N. Gomez\* †**

University of Toronto

aidan@cs.toronto.edu

**Lukasz Kaiser\***

Google Brain

lukaszkaiser@google.com

**Illia Polosukhin\* ‡**

illia.polosukhin@gmail.com

# Attention is all you need

---

## Attention Is All You Need

---

~62,000 citations in  
5 years!

**Ashish Vaswani\***

Google Brain

avaswani@google.com

**Noam Shazeer\***

Google Brain

noam@google.com

**Niki Parmar\***

Google Research

nikip@google.com

**Jakob Uszkoreit\***

Google Research

usz@google.com

**Llion Jones\***

Google Research

llion@google.com

**Aidan N. Gomez\* †**

University of Toronto

aidan@cs.toronto.edu

**Lukasz Kaiser\***

Google Brain

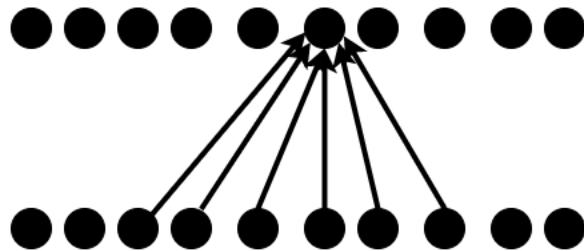
lukaszkaiser@google.com

**Illia Polosukhin\* ‡**

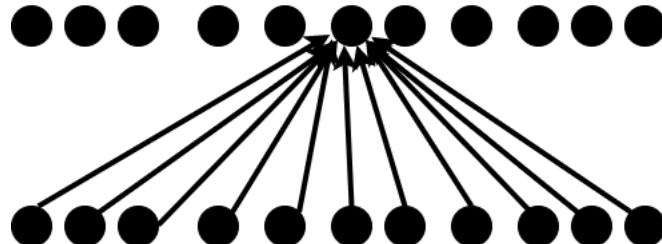
illia.polosukhin@gmail.com

# Attention vs convolution

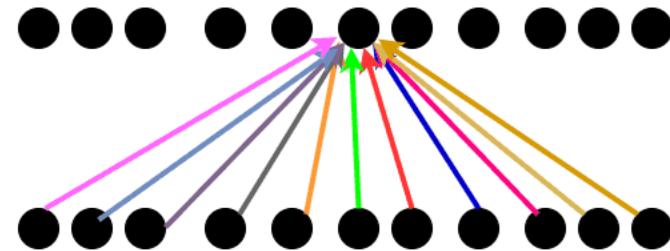
Convolution



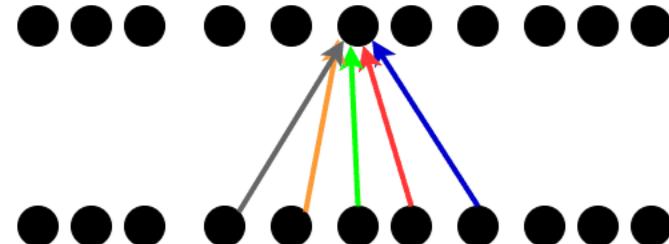
Fully Connected layer



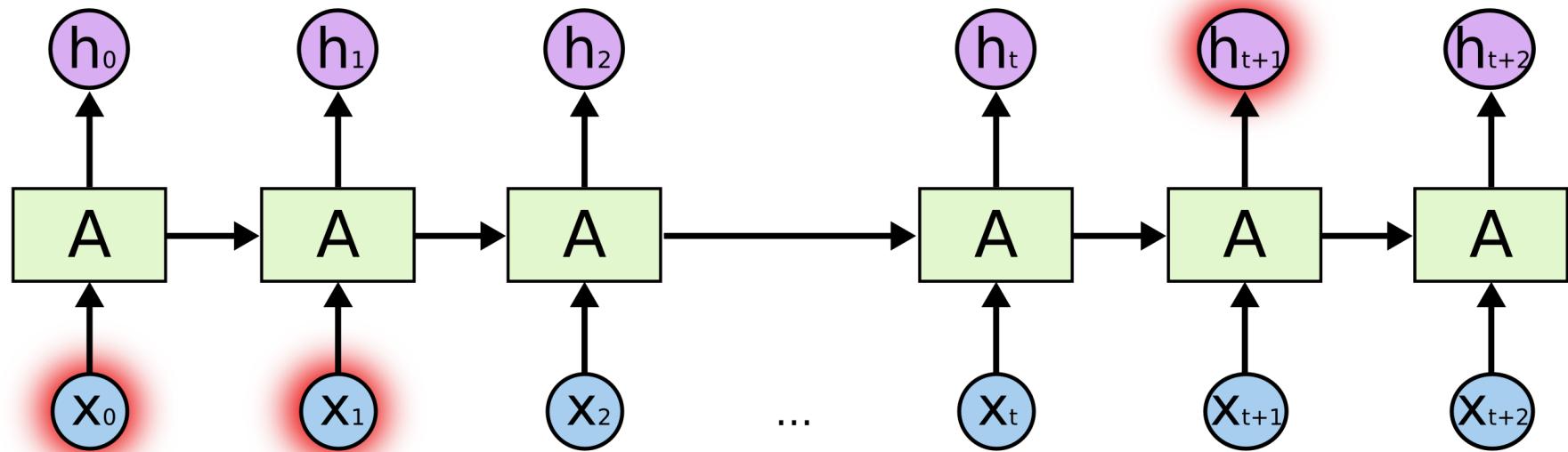
Global attention



Local attention



# Long-Term Dependencies

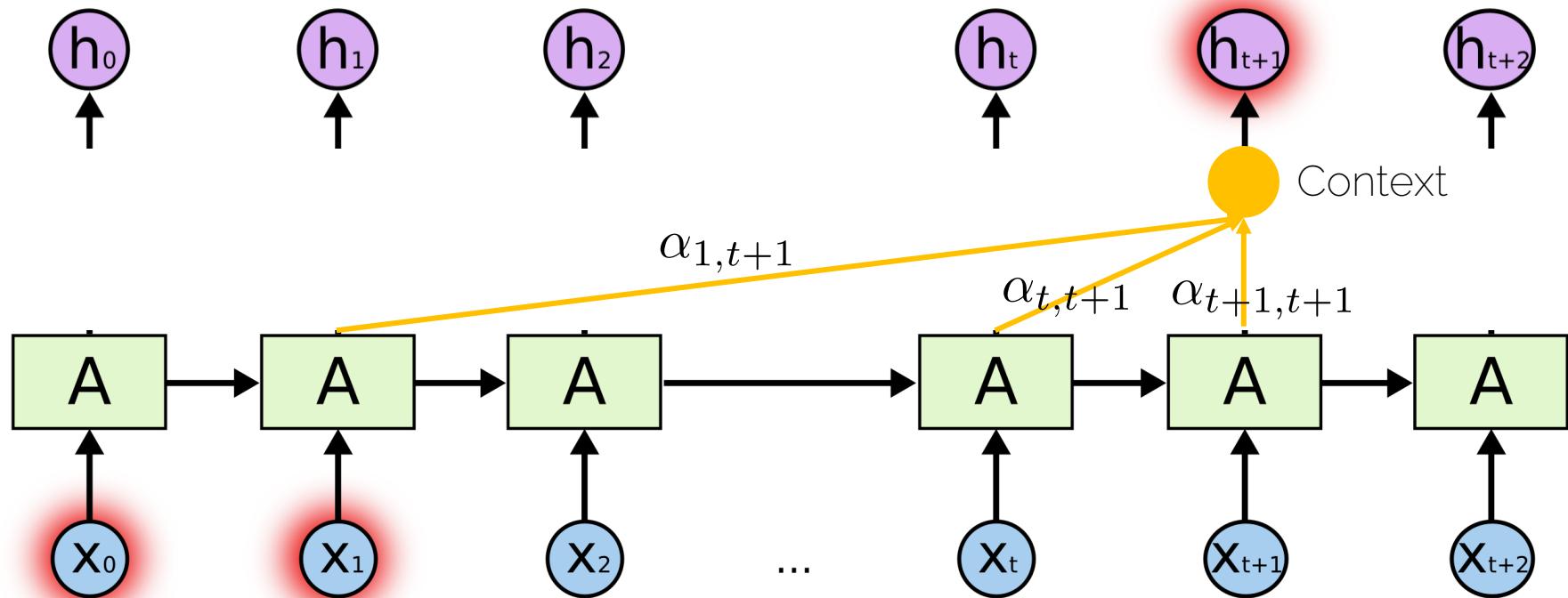


I moved to Germany ...

so I speak German fluently.

Source: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

# Attention: Intuition



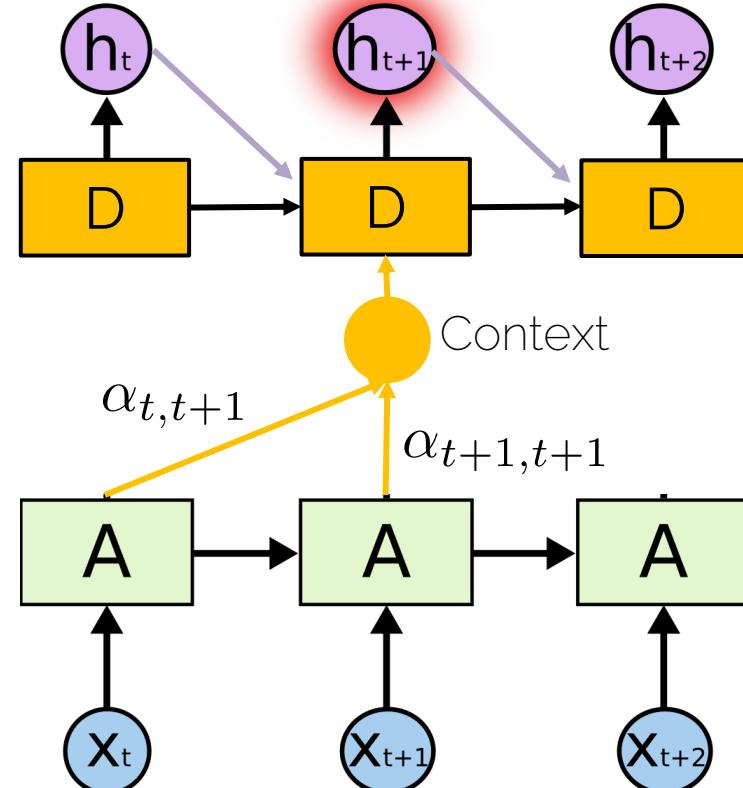
I moved to Germany ...

so I speak German fluently

# Attention: Architecture

- A decoder processes the information

- Decoders take as input:
  - Previous decoder hidden state
  - Previous output
  - Attention



# Transformers

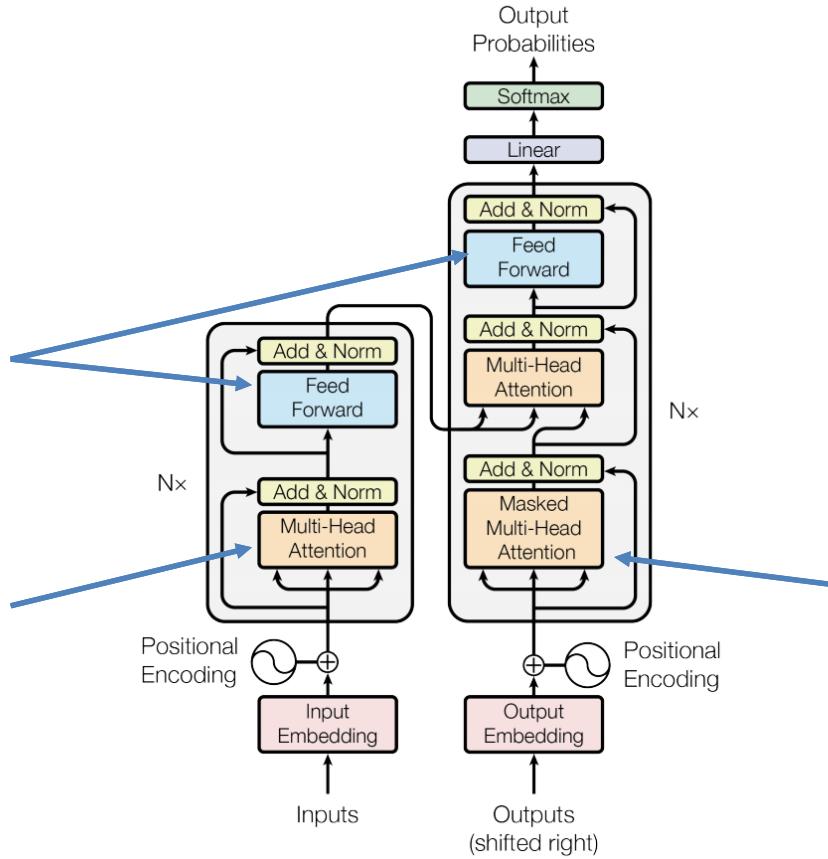
# Deep Learning Revolution

	Deep Learning	Deep Learning 2.0
Main idea	Convolution	Attention
Field invented	Computer vision	NLP
Started	NeurIPS 2012	NeurIPS 2017
Paper	AlexNet	Transformers
Conquered vision	Around 2014-2015	Around 2020-2021
Replaced (Augmented)	Traditional ML/CV	CNNs, RNNs

# Transformers

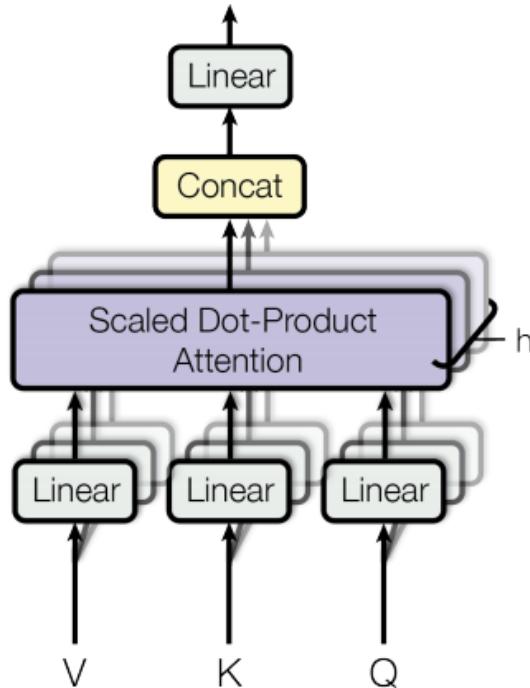
Fully connected layer

Multi-Head Attention on the "encoder"



Masked Multi-Head Attention on the "decoder"

# Multi-Head Attention



Intuition: Take the query  $Q$ , find the most similar key  $K$ , and then find the value  $V$  that corresponds to the key.

In other words, learn  $V$ ,  $K$ ,  $Q$  where:

- $V$  – here is a bunch of interesting things.
- $K$  – here is how we can index some things.
- $Q$  – I would like to know this interesting thing.

Loosely connected to Neural Turing Machines (Graves et al.).

# Multi-Head Attention

Index the values via a differentiable operator.

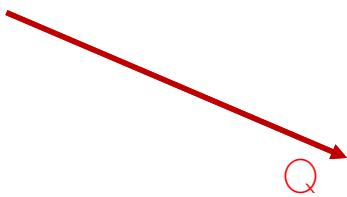
Multiply queries with keys

Get the values

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

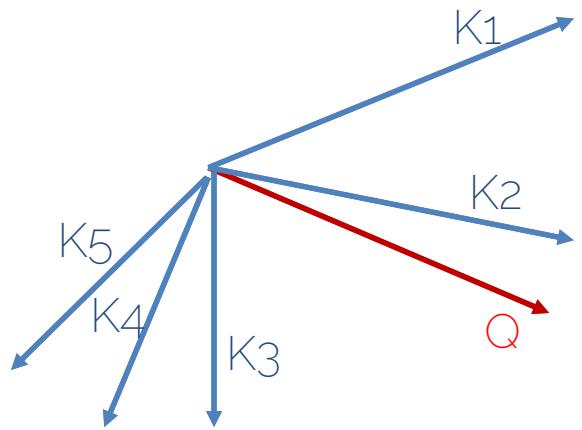
To train them well, divide by  $\sqrt{d_k}$ , "probably" because for large values of the key's dimension, the dot product grows large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

# Multi-Head Attention

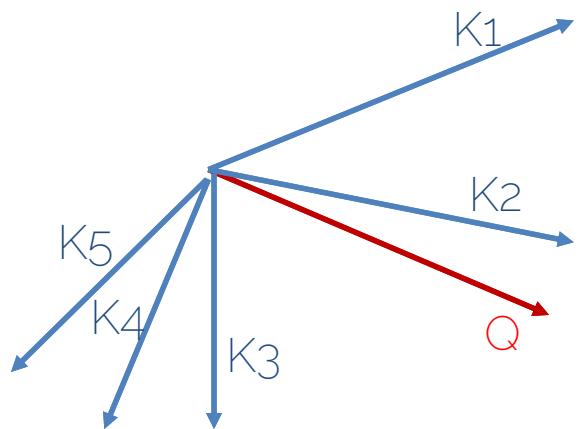


Adapted from Y. Kilcher

# Multi-Head Attention

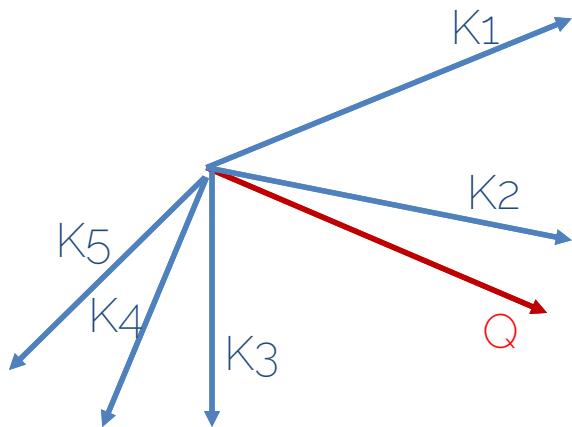


# Multi-Head Attention



Values
V1
V2
V3
V4
V5

# Multi-Head Attention

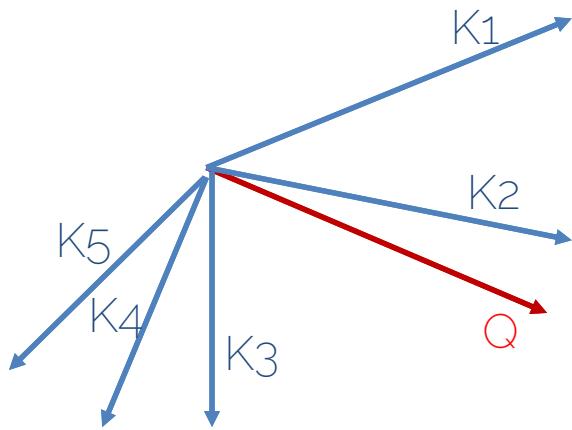


Values
V1
V2
V3
V4
V5

$$QK^T$$

Essentially, dot product between ( $\langle Q, K_1 \rangle$ ), ( $\langle Q, K_2 \rangle$ ), ( $\langle Q, K_3 \rangle$ ), ( $\langle Q, K_4 \rangle$ ), ( $\langle Q, K_5 \rangle$ ).

# Multi-Head Attention

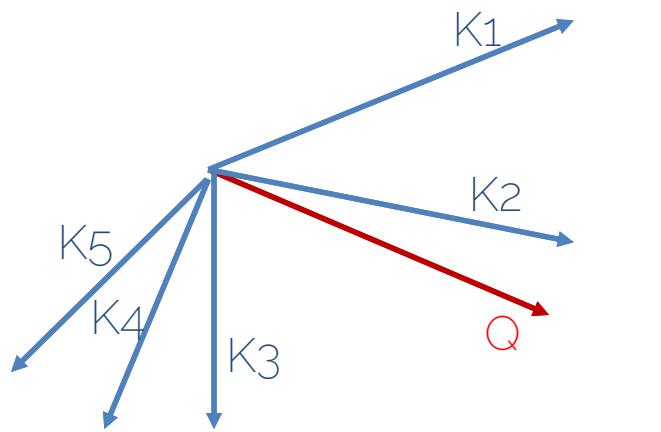


Values
V <sub>1</sub>
V <sub>2</sub>
V <sub>3</sub>
V <sub>4</sub>
V <sub>5</sub>

$$\text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right)$$

Is simply inducing a distribution over the values.  
The larger a value is, the higher is its softmax value.  
Can be interpreted as a differentiable soft indexing.

# Multi-Head Attention

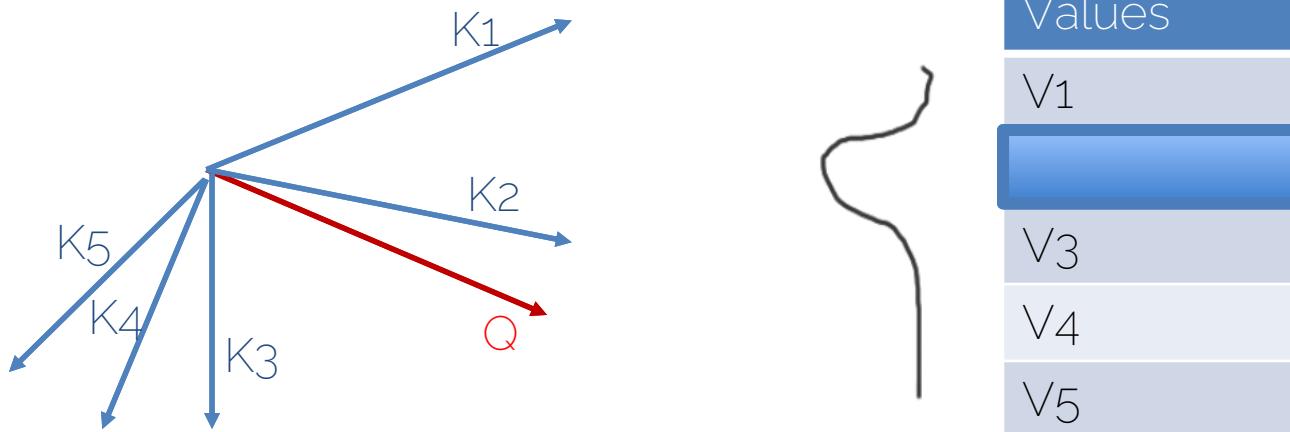


Values
V1
V2
V3
V4
V5

$$\text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right)$$

Is simply inducing a distribution over the values.  
The larger a value is, the higher is its softmax value.  
Can be interpreted as a differentiable soft indexing.

# Multi-Head Attention

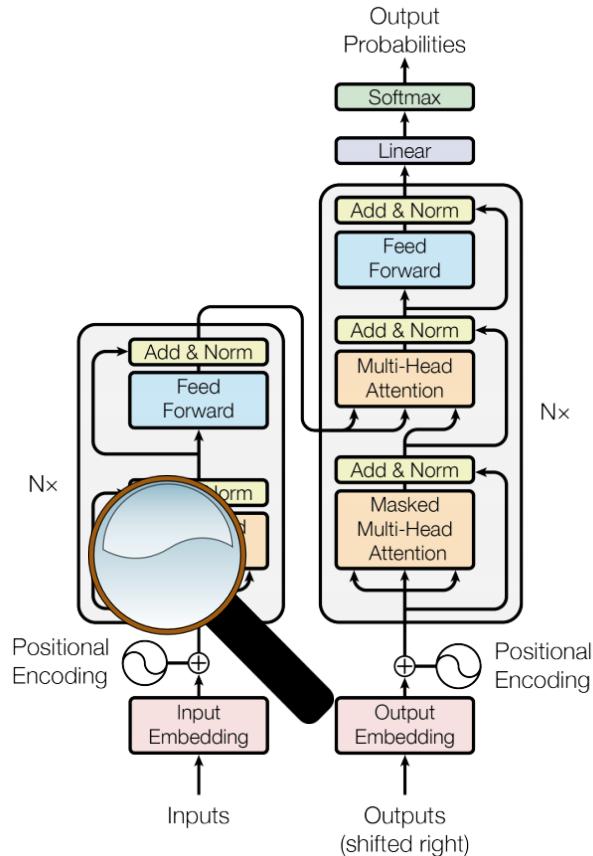
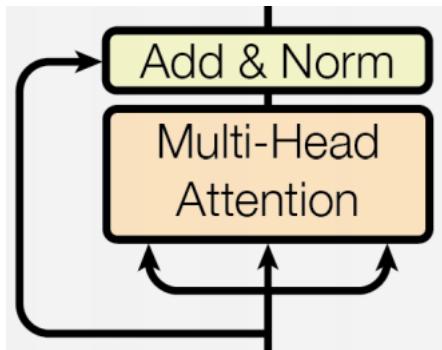


$$\text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right)$$

Selecting the value V where  
the network needs to attend..

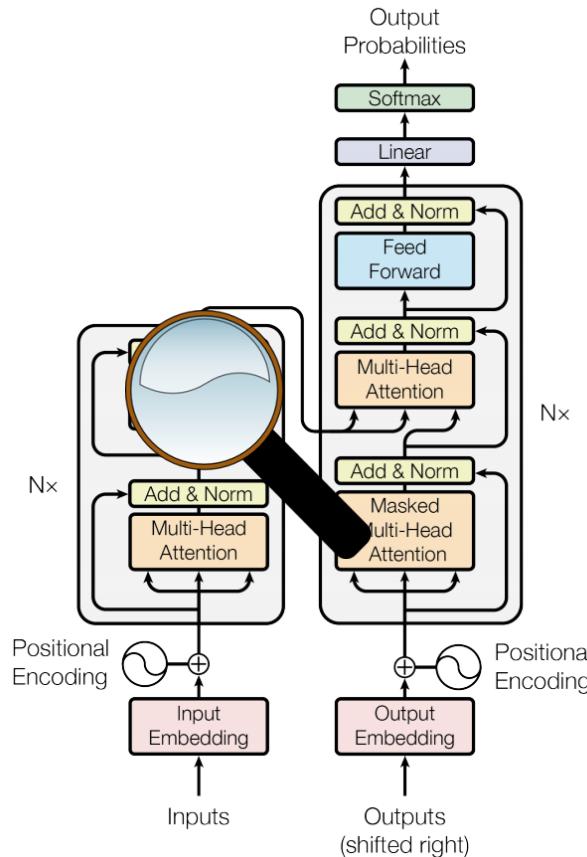
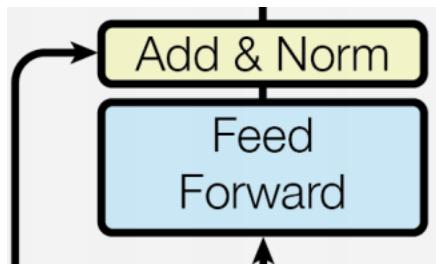
# Transformers – a closer look

K parallel  
attention heads.



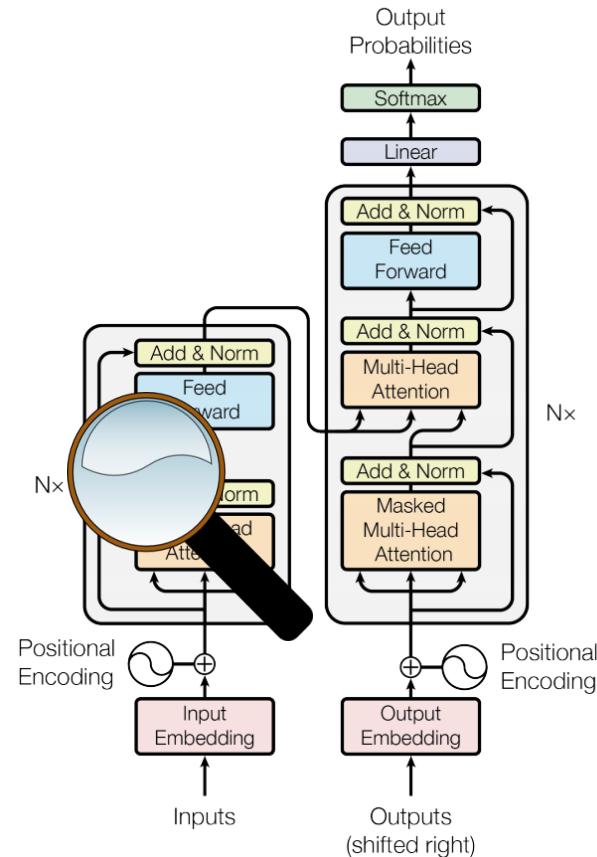
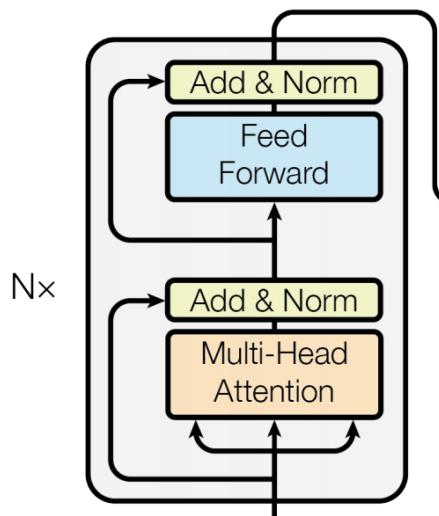
# Transformers – a closer look

Good old fully-connected layers.



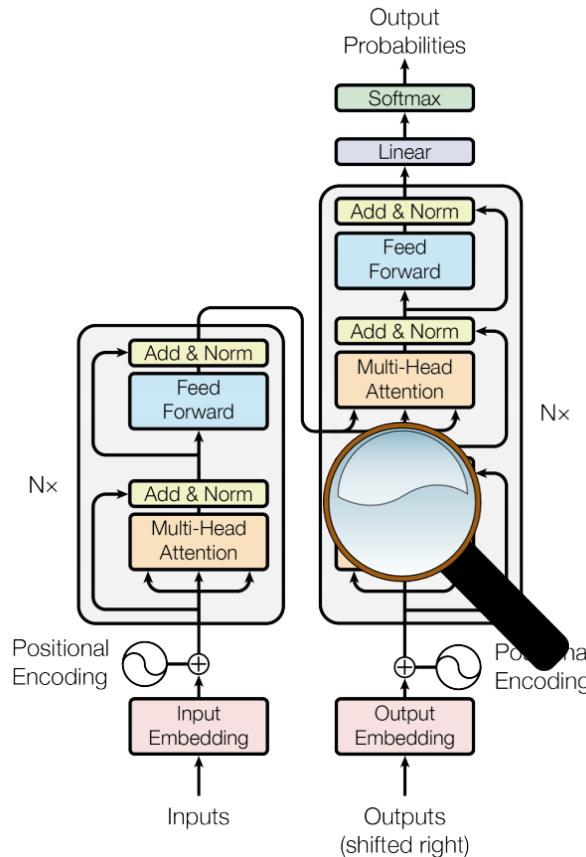
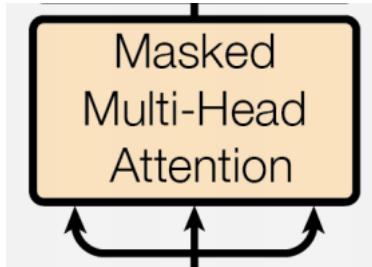
# Transformers – a closer look

N layers of  
attention  
followed by FC



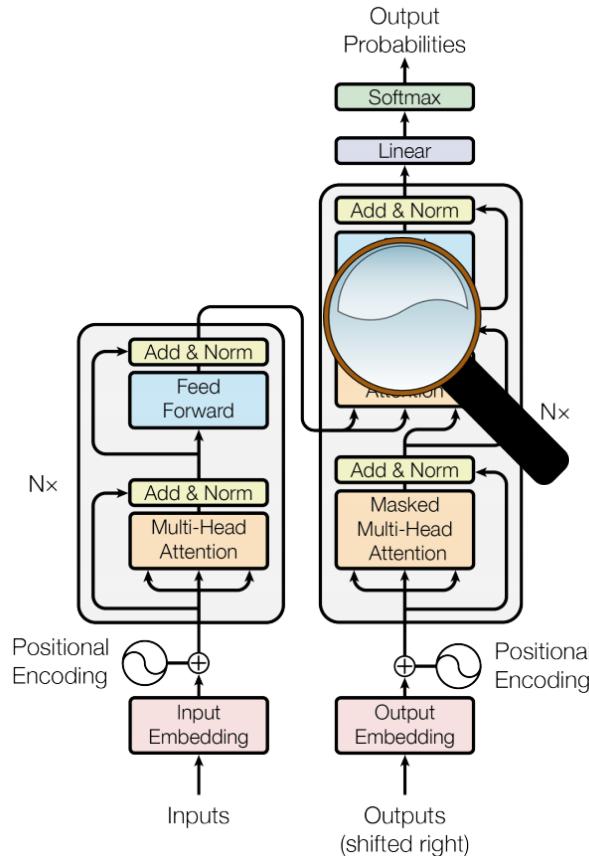
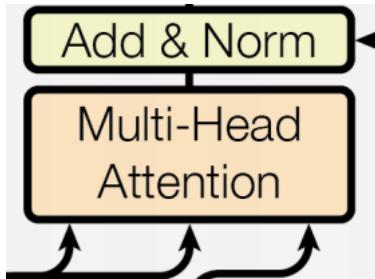
# Transformers – a closer look

Same as multi-head attention, but masked. Ensures that the predictions for position  $i$  can depend only on the known outputs at positions less than  $i$ .



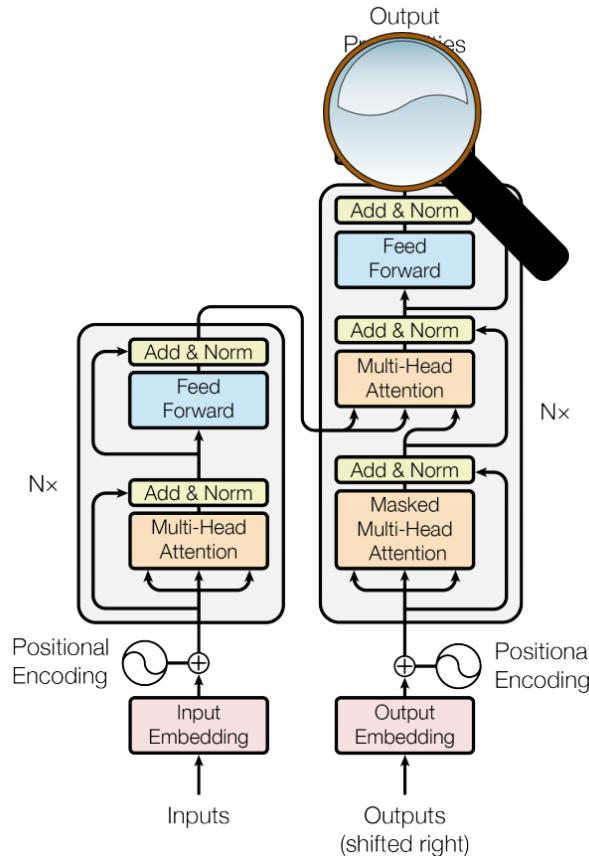
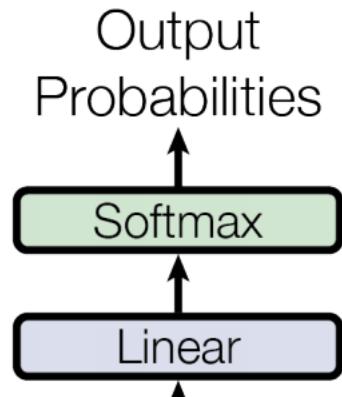
# Transformers – a closer look

Multi-headed attention between encoder and the decoder.



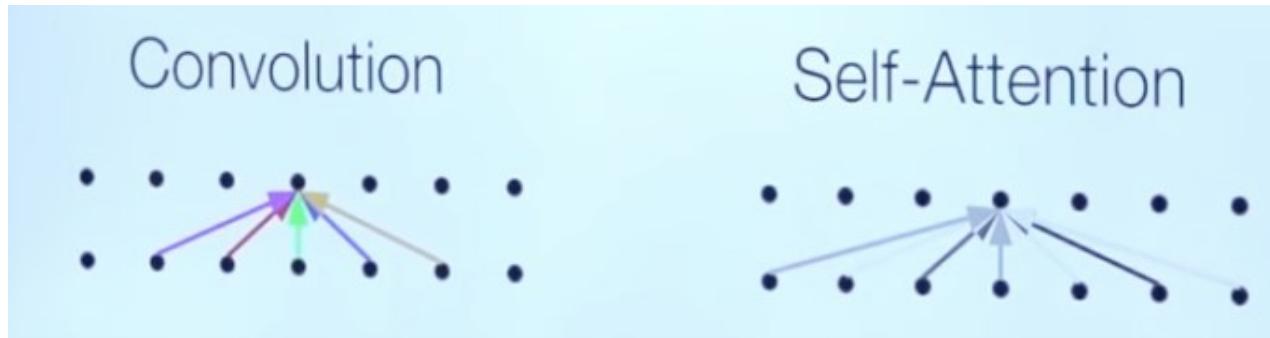
# Transformers – a closer look

Projection and prediction.



# What is missing from self-attention?

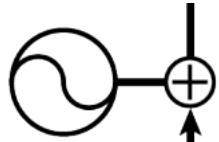
- Convolution: a different linear transformation for each relative position. Allows you to distinguish what information came from where.
- Self-attention: a weighted average.



# Transformers – a closer look

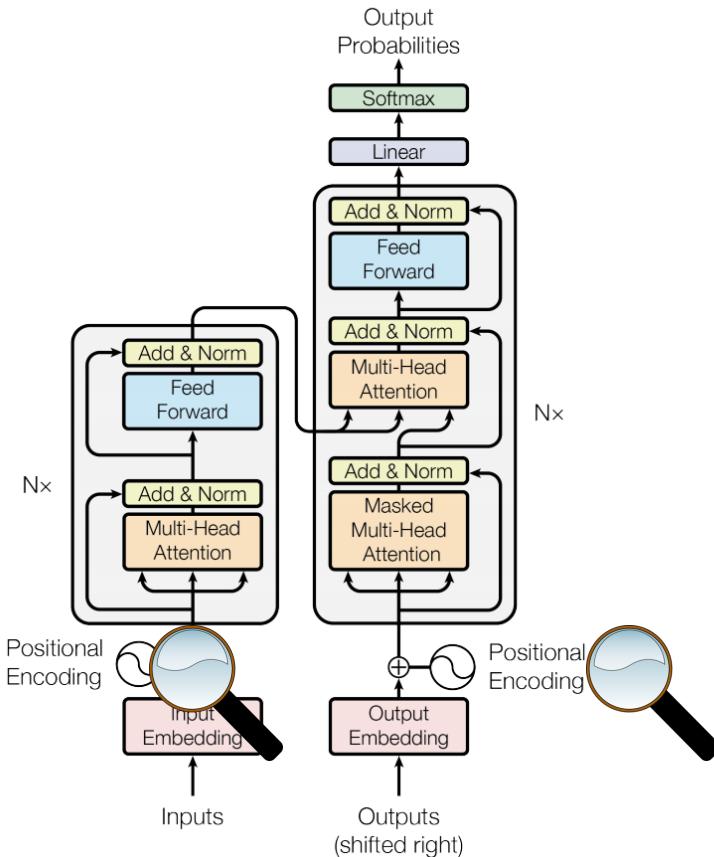
Uses fixed positional encoding based on trigonometric series, in order for the model to make use of the order of the sequence

Positional Encoding

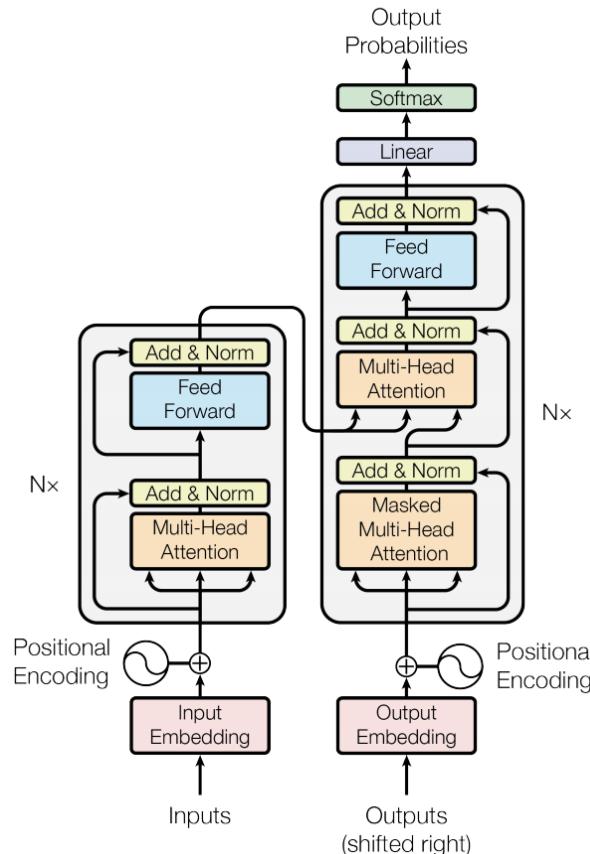


$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$



# Transformers – a final look



# Self-attention: complexity

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

where  $n$  is the sequence length,  $d$  is the representation dimension,  $k$  is the convolutional kernel size, and  $r$  is the size of the neighborhood.

# Self-attention: complexity

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

where  $n$  is the sequence length,  $d$  is the representation dimension,  $k$  is the convolutional kernel size, and  $r$  is the size of the neighborhood.

Considering that most sentences have a smaller dimension than the representation dimension (in the paper, it is 512), self-attention is very efficient.

# Transformers – training tricks

- ADAM optimizer with proportional learning rate:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

- Residual dropout
- Label smoothing
- Checkpoint averaging

# Transformers - results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1		<b><math>3.3 \cdot 10^{18}</math></b>
Transformer (big)	<b>28.4</b>	<b>41.0</b>		$2.3 \cdot 10^{19}$

# Transformers - summary

- Significantly improved SOTA in machine translation
- Launched a new deep-learning revolution in MLP
- Building block of NLP models like BERT (Google) or GPT/ChatGPT (OpenAI)
- BERT has been heavily used in Google Search
- And eventually made its way to computer vision (and other related fields)

See you next time!

TECHNISCHE UNIVERSITÄT MÜNCHEN

SUMMARY OF THE LECTURE I2DL

## **Introduction to Deep Learning**

*according to the Lecture of Prof. Niessner and Prof. Leal-Taixé*

# Contents

<b>I</b>	<b>Introduction</b>	<b>5</b>
I.1	The history of Computer Vision	5
I.2	The summer vision project 1966	5
I.3	Image classification	5
I.4	History of Deep Learning	6
I.5	What made this [Deep Learning] possible?	6
I.6	Different Tasks in DL	6
<b>II</b>	<b>Machine Learning Basics</b>	<b>7</b>
II.1	Linear Regression	7
II.2	Maximum Likelihood	8
II.3	Logistic Regression	9
<b>III</b>	<b>Neural Networks</b>	<b>12</b>
III.1	Computational Graphs	13
III.2	Loss Functions	13
<b>IV</b>	<b>Optimization and backpropagation</b>	<b>15</b>
IV.1	Backpropagation	15
IV.2	Gradient Descent	16
IV.3	Regularization	17
<b>V</b>	<b>Scaling Optimization and Stochastic Gradient Descent</b>	<b>19</b>
V.1	Gradient Descent	19
V.2	The Newton Method	22
V.3	Other Optimization methods	23
<b>VI</b>	<b>Training Neural Networks I</b>	<b>24</b>
VI.1	Learning rate	24
VI.2	Training	24
VI.3	Hyperparameter tuning	26
<b>VII</b>	<b>Training Neural Networks II</b>	<b>28</b>
VII.1	Output and Loss Function	28
VII.2	Activation Functions	29
VII.3	Weight Initialization	31
<b>VIII</b>	<b>Training Neural Networks III</b>	<b>33</b>
VIII.1	Data Augmentation	33
VIII.2	Advanced Regularization	33
VIII.3	Batch Normalization	34
VIII.4	Other Normalizations	35
VIII.5	Recap	35

<b>IX</b>	<b>Introduction to CNNs</b>	<b>36</b>
IX.1	Convolution . . . . .	36
IX.2	Convolution Layer . . . . .	37
IX.3	Dimensions of a Convolution Layer . . . . .	37
IX.4	Convolutional Neural Network (CNN): Pooling . . . . .	38
IX.5	Receptive Field . . . . .	38
IX.6	Dropout for CNN . . . . .	39
IX.7	Transfer Learning . . . . .	39
<b>X</b>	<b>Popular CNN Architectures</b>	<b>41</b>
X.1	LeNet . . . . .	41
X.2	AlexNet . . . . .	41
X.3	VGGNet . . . . .	42
X.4	Skip Connections: Residual Block . . . . .	42
X.5	ResNet (Residual Networks) . . . . .	43
X.6	Inception Layer: 1x1 Convolution . . . . .	44
X.7	XceptionNet . . . . .	44
X.8	Fully Convolutional Network . . . . .	45
<b>XI</b>	<b>Recurrent Neural Networks</b>	<b>48</b>
XI.1	Transfer Learning . . . . .	48
XI.2	Recurrent Neural Networks . . . . .	48
XI.3	Long Short Term Memory . . . . .	50
<b>XII</b>	<b>Advanced Deep Learning Topics</b>	<b>51</b>
XII.1	Attention . . . . .	51
XII.2	Graph Neural Networks . . . . .	51
XII.3	Generative Models . . . . .	52
XII.4	Variational Autoencoder . . . . .	52
XII.5	Generative Adversarial Networks (GAN) . . . . .	53
XII.6	Reinforcement Learning . . . . .	54
XII.7	Autoencoder . . . . .	55

## Preface

Hey guys, Eva created this great summary and I added some of my notes, and we would like to share it with you, so everyone can add their notes. As we share the editable link, please make sure not to delete it ;) and you are very welcome to add your notes / pictures / improve the latex file (I am not a pro, just used a good template). If you add your note just make sure not to write anything that is already in the document — it is better to improve and expand the correct paragraphs. Have fun!!

Munich 2021-2023

# I Introduction

Computer Vision: We're trying to make sure that machines are learning to see in a similar way that humans are doing. That is why we need Machine Learning methods, to get there. CV is the center for robotics so that you understand the environment and what it does for you. There's a lot of images/videos processing done by CV as well.

## I.1 The history of Computer Vision

### Hubel and Wiesel Experiment

Hubel and Wiesel (neurobiologists) experimented on cats' brains by putting electrodes in it and recording them while the cat was being shown stimuli through a screen (mostly edges). They were able to find out that visual cortex cells are sensitive to the orientation of edges, yet they were insensitive to the position of the edges. Something that we will see later in convolutional networks.

## I.2 The summer vision project 1966

They tried to construct a significant part of a visual system, and it was the time when pattern recognition was coined.

CV is a very core element to other areas as well such as robotics, NLP, optics and image processing, algo optimization, neuroscience, AI and ML.

## I.3 Image classification

Previously DL was not used for Image classification yet it became popular later. Earlier they did pre-processing (i.e. normalizing the colors of images), then came the feature descriptor which functioned kind of the same as the Hubel Wiesel Experiment in the sense that certain properties were not important such as position of edges.

Different types of feature descriptor are HAAR, HOG, SIFT, SURF. In order to get to these feature descriptor they had to hand engineer it, since most are gradient based. After that you have aggregators such as svm rf, ann etc which would aggregate the features and give the label.

Instead of feature extraction+accumulation we have a magic box that does that for us. That magic box is deep learning. We do not have to hand engineer the feature descriptors. We are letting a data set decide what the best possible descriptor might be that will give us the best results.

### Image Classification Issues:

- Occlusion.
- Background cluttering: Background and foreground (object) similar colors
- Representation: Ex: cat drawing vs cat photo

## I.4 History of Deep Learning

Started in 1940 with the electronic brain. Each cell has a certain pattern in them. They accumulated weights/impulses and eventually made a decision.

1960 we saw the perceptron. Instead of fixed weights, we could learn the weights. We showed the system a couple of example and we hope to essentially learn certain parameters of these perceptrons. We learn the feature extraction (weights) and the threshold of learning. This was all hardwired.

Then we had Adaline (the golden age of deep learning). There was a lot of hype and progress being made.

Then in 1969, people realized the problems with perceptrons, specifically the xor problem. The problem was that a linear model (a single perceptron) cannot separate the two classes. This era was called the AI winter.

In 1986, the multi-layer perceptron came to light. We have several layers that can be trained (optimized for the weights of the multi-layer perceptron). This is called backpropagation. Gradient based method.

In 1995 there was the SVM. Since it was successful, it put a halt to deep learning.

In 2006, Hinton and Ruslan developed Deep Belief Networks. The idea of pretraining came around. So you train an nn and then you train it again for a specific task. The idea of pretraining is still one of the most relevant today (for example transfer learning with ImageNet weights). Despite of this, neural networks were still not a mainstream method.

In 2012 : the AlexNet architecture (see Section X.2) was the first neural network based architecture that won the ImageNet competition based on the lowest top 5-error.

Definition of top 5 error: Give me an image, ask the method what class it is and see if the top five predictions include the correct class.

## I.5 What made this [Deep Learning] possible?

- Big Data: When we have big data, models learn where to learn from and we have so much more data today than we did back then. The datasets are also online.
- Better Hardware: Not only has the data changed, the hardware has changes as well (i.e. GPU). Hardware was developed for the rendering of images in games, and it is now used as well for deep learning, to train models faster.
- Models are more complex

## I.6 Different Tasks in DL

- Object Detection
- Self-Driving Cars
- Gaming (i.e. AlphaGO, AlphaStar)
- Machine Translation
- Automated Text Generation (ChatBots)
- Healthcare, cancer Detection

## II Machine Learning Basics

There are a large variety of tasks, such as image classification, which is sometimes (depending on the number of classes) called binary classification. (key words: background clutter, semantic differences). The task of image classification should be done by using data, we want to train a model and make it learn from the data. From here we can say that we distinguish ML methods into two types: unsupervised learning methods and supervised learning methods.

**Unsupervised Learning:** doesn't have any labels or target classes. We just want to find out the properties of the structure of the data. An example of this can be clustering, kmeans, pca.

**Supervised Learning:** in supervised data we have the respective labels or target classes(done by annotators).

But how do we actually learn image classification, and how do we know we are learning and not memorizing things? First, we need to separate the data into training data and test data. Take the train data and train a model. Then you test on the test data to see how well the model is doing. We measure the performance, to see how good the model is doing. A metric would be accuracy. An underlying assumption is that train and test data come from the same distribution.

**Nearest neighbor Model:** Unsupervised learning method<sup>ii</sup>, labels the sample based on the majority label of its neighboring samples. The hyper-parameters to be tweaked in KNN are: k, L1 or L2 distances. These hyperparameters are found by using the validation data. (Train-validation-test: 60-20-20 or 80-10-10). The test set is used only once at the end to display the model's performance. It is not used to 'fix' the accuracy. The latter is done in the validation set.

**Cross Validation:** Split the data in K folds and iterate through the permutations; test on one of the folds and train on the rest.

**Decision Boundaries** are boundaries where the data is separated into classes.

The pros and cons of using linear decision boundaries:

- + It's very easy to implement and derive
- + It's easy to find the hyperparameters
- The distribution must be clearly separated
- Harder to use for multi classes (?)

### II.1 Linear Regression

Linear Regression is a supervised learning method that finds a linear model that explains a target  $y$  given inputs  $x$  with weights  $\theta$ :

---

<sup>ii</sup>Well, I disagree with that. If you just want to list the neighbors, it is unsupervised. But as soon as you want to make predictions (e.g. average the neighbors), you need the true labels and it is supervised.

$$\hat{y}_i = \sum_{j=1}^d x_{ij}\theta_j$$

and the prediction looks like:

$$\hat{y}_i = \theta_0 + \sum_{j=1}^d x_{ij}\theta_j = \theta_0 + x_{i1}\theta_1 + x_{i2}\theta_2 + \dots + x_{id}\theta_d \Rightarrow \hat{\mathbf{y}} = \mathbf{X}\theta$$

$x_{ij}$  are the features;  $\theta$  are the weights (model parameters).  $\theta_0$  is a bias. In linear data it is where the decision line intersects with  $y$ .

**Loss Function** measures the goodness of the estimation and tells the optimization method how to make it better.

**Optimization** changes the model to improve the estimates and minimize the loss. The goal is to reduce the loss function.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

$$\min_{\theta} J(\theta) = \frac{1}{n} \sum_{i=1}^n (x_i\theta - y_i)^2$$

$J$  is the function that we want to minimize. Basically we want the find  $\theta$ s that minimize  $J$ .

To find the optimum we need to find  $\frac{\partial J}{\partial \theta} = 0$

This loss function is convex which means it has a minimum which means there exists a solution i.e an optimal theta.

Matrix notation:  $\min_{\theta} J(\theta) = (\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y})$

There exists a closed form solution for this loss function:  $\theta = (X^T X)^{-1} X^T y = X^\dagger y$

Is least squares estimate the best estimate? If linear: Yes.

## II.2 Maximum Likelihood

MLE is a method of estimating the parameters of a statistical model given observations. This is done by finding the parameter values that maximize the likelihood of making the observations given by the parameters.

$$\begin{aligned} \theta_{ML} &= \arg \max_{\theta} p_{\text{model}}(Y|X, \theta) \\ &= \prod_{i=1}^n p_{\text{model}}(y_i|x_i, \theta) \\ &= \sum_{i=1}^n \log p_{\text{model}}(y_i|x_i, \theta) \end{aligned}$$

**MLE assumes that the training samples are independent and generated by the same distribution.**

What shape does our probability distribution have? Assuming Gaussian distribution:  $y_i = \mathcal{N}((x_i)\theta, \sigma^2) = x_i\theta + \mathcal{N}(0, \sigma^2)$

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i - \mu)^2}$$

$$p(y_i|x_i, \theta) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i - x_i\theta)^2}$$

then after more matrix calculations we get:

$$\theta_{ML} = -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(y - X\theta)^T(y - X\theta)$$

$$\theta = (X^T X)^{-1} X^T y$$

So the MLE is the same as the least squares estimate we found previously.

## II.3 Logistic Regression

**Sigmoid function** maps the values into 0 and 1 and its formula is below:

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

**Regression:** predict a continuous output value (e.g. temperature of a room)

**Classification:** predict a discrete value, Binary classification (output either 0 or 1) and Multi-class classification (set of N classes)

Probability of a binary output:

$$\hat{y} = p(y = 1|X, \theta) = \prod_{i=1}^n p(y_i = 1|x_i, \theta)$$

$$p(y|X, \theta) = \hat{y} = \prod_{i=1}^n \hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)}$$

$$\hat{y}_i = \sigma(x_i\theta)$$

Maximum Likelihood Estimate:  $\theta_{ML} = \arg \max_{\theta} \log p(y | \mathbf{X}, \theta)$

We are interested in maximizing the likelihood quantity  $\theta_{ML} = \arg \max_{\theta} p(y = 1|X, \theta)$ . Since log simplifies and maintains our maximum, we get  $\theta_{ML} = \arg \max_{\theta} \log p(y = 1|X, \theta)$ . Taking the logarithm, we had to add a negative, which ended up with a minimization. Here are the steps:

1. let  $\hat{y} = p(y = 1|X, \theta)$
2. Since Binary Problem, we use Bernoulli, which can be defined as either :
  - a)  $P(k) = p^k(1-p)^{(1-k)}, k = \{0, 1\}$  [we stick to this]
  - b)  $P(k) = pk + (1-p)(1-k), k = \{0, 1\}$

As Both will produce :

- a)  $P(k = 1) = p$
  - b)  $P(k = 0) = (1 - p)$
  - 3. Our prob is represented by sigmoid function.  $\sigma(x) = \frac{1}{1+e^{-(x_i\theta)}} = \hat{y}$  (prediction)  
 Note: The fraction in the sigmoid function with log will cause a negative later. Since  $\log(\frac{a}{b}) = \log(a) - \log(b)$
  - 4.  $\hat{y}_i = p(y = 1|X, \theta) = \prod_{i=1}^N \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$ . (going  $\theta \rightarrow \hat{y}_i$ )
  - 5. Goal is to  $\max_{\theta} p(y = 1|X, \theta) = \max \hat{y}_i = \max \prod_{i=1}^N \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$
- This maximization boils down to 2 cases in Binary settings:
- a)  $y_i = 1, \max \prod_{i=1}^N \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$  reduces to  $\max \prod_{i=1}^N \hat{y}_i \rightarrow \text{Goal } \max \hat{y}_i$
  - b)  $y_i = 0, \max \prod_{i=1}^N \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$  reduces to  $\max \prod_{i=1}^N 1 - \hat{y}_i \rightarrow \text{Goal } \min \hat{y}_i$
6. Taking  $\log(p(y = 1|X, \theta))$ :

Taking Log is tricky. As to Maintain Same Goals and to make sense :

- a)  $\log(p)$  Always **-ve**  $\iff -\log(p)$  Always **+ve**

As  $\log(p)$  where  $p$  is a probability,  $p \in [0, 1]$ . Therefore, the logarithm function produces negative values

- b) We can't accept a -ve likelihood prob. , so we must Mult. by **-1** with every  $\log(\text{prob})$ .

Therefore:

$$\log(p(y = 1|X, \theta)) = \log(\prod_{i=1}^N \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}) \Rightarrow -1 \sum_{i=1}^N y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) = -\sum_{i=1}^N L(\hat{y}_i, y_i)$$

Goal is to:

$$\max_{\theta} \log(p(y = 1|X, \theta)) \Rightarrow -\sum_{i=1}^N y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

This maximization boils down to 2 cases in Binary settings:

- a)  $y_i = 1, \max -1 \sum_{i=1}^N \log(\hat{y}_i) = \min \sum_{i=1}^N \log(\hat{y}_i) \rightarrow \text{Same Goal } \max \hat{y}_i \checkmark$
- b)  $y_i = 0, \max -1 \sum_{i=1}^N \log(1 - \hat{y}_i) = \min \sum_{i=1}^N \log(1 - \hat{y}_i) \rightarrow \text{Same Goal } \min \hat{y}_i \checkmark$
- c) These Goals also match from the perspective of NN weights. when you expand  $\hat{y} = \sigma(x_i\theta)$ :

- i.  $y_i = 1 : \max \sum_{i=1}^N \log(\hat{y}_i) \Rightarrow \max \sum_{i=1}^N (x_i\theta) \rightarrow \text{Goal } \max (x_i\theta)$
- ii.  $y_i = 0 : \max \sum_{i=1}^N \log(1 - \hat{y}_i) \Rightarrow \max \sum_{i=1}^N (1 - x_i\theta) \rightarrow \text{Goal } \max (1 - x_i\theta) = \min x_i\theta$

- 7. Taking log transformed  $\max_{\theta} \log(p(y = 1|X, \theta)) \rightarrow \min \sum_{i=1}^N L(\hat{y}_i, y_i)$

Going from **max**  $\rightarrow$  **min** feels more like a loss function that we hope to minimize .

- 8. Finally Cost func:

$$L(\hat{y}_i, y_i) = y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

$$C(\theta) = \frac{-1}{N} \sum_i^N L(\hat{y}_i, y_i) \text{ which we hope to } \min.$$

This is called binary cross-entropy loss or BCE.

In the more general case (number of classes  $> 2$ ), the cross entropy loss can be written as :

$$\mathcal{L}(\hat{y}_i, y_i) = \sum_{i=1}^N \sum_{j=1}^C y_{i,j} \log \hat{y}_{i,j}$$

Cost function:  $C(\theta) = -\frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}_i, y_i)$ . We want to minimize the cost function (minimizing the negative log-likelihood is equivalent to maximizing the log-likelihood).

To optimize there is no closed form solution so we need to use iteration to solve it. This is done by using gradient descent. Gradient Descent based optimizers are explained in detail in section V.

L1 norm:  $\|v\|_1 = \sum_{i=1}^n |v_i|$

L2 norm:  $\|v\|_2 = \sqrt{\sum_{i=1}^n (v_i)^2}$

### III Neural Networks

Linear score functions are defined as:  $1$  where  $W$  is the weights and  $x$  is our inputs.

The weights of the linear score would essentially be the mean of the loss function that's L2. Yet when the data has a lot of variety, the linear score function does not work anymore.

Adding more weight matrices does not work because the function is still linear. We need to add some non-linearity and we do that like the following:

$$\text{Lineare score function: } f = Wx$$

$$2 \text{ layers: } f = W_2 \max(0, W_1 x)$$

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$f = W_4 \tanh(W_3 \max(0, W_2 \max(0, W_1 x)))$$

$$f = W_5 \sigma(W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))$$

Basically we have layers of non-linearity stacked on top of each other. These non linear functions are called activation functions.

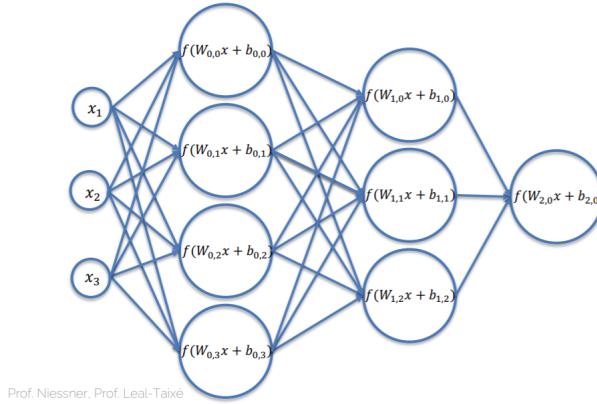


Figure III.1: Layers of a neural network: 3-dimensional input, first layer: 12 weights in total (3 inputs x 4 neurons) and 4 bias

As seen in the image above, a neural network is made of the input layer, the hidden layers and the output layer. The output layer size is the number of classes. The graph is fully connected, meaning that the previous layer's nodes are connected to all of the next layer's node.

#### Activation Functions

- Sigmoid:  $\sigma(x) = \frac{1}{1+e^{-x}}$
- tanH:  $\tanh(x)$
- ReLU:  $\max(0, x)$
- Leaky ReLU:  $\max(0.1x, x)$
- Parametric ReLU:  $\max(ax, x)$

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

- ELU:  $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$

**Why use activation functions?** Without activation functions network would collapse and would have a linear model. Non-linearity  $\rightarrow$  more capacity and better learning

**Why organize a neural network into layers?** Good to use with matrix calculations  $\rightarrow$  easy to abstract math (e.g. gradient descent), every input layer corresponds to a picture.

Neural networks are inspired by biological neurons. Yet we are still at the beginning and not even close to what the human brain can do.

### Summary

Given a dataset with ground truth training pairs  $[x_i; y_i]$

Find  $W$ s using stochastic gradient descent, that minimize the loss function. We need to compute gradients with backpropagation. Iterate many times over training set.

## III.1 Computational Graphs

Computational graphs are directional graphs where matrix operations are represented as compute nodes; vertex nodes are variable or operators like  $+ - * /$ ; directional edges show the flow of input to vertices.

**Why?** NN have complicated architectures, represent as computational graph because it has compute nodes (operations), edges that connect nodes (data flow), is directional (input from left to right, without loops), can be organized into layers.

### Further meanings

- Multiplication of  $W_i$  and  $x$ : encode input information
- activation function: select key features
- convolutional layers: extract useful features with shared weights

## III.2 Loss Functions

The loss functions shows how close the predictions are to the targets. It's a measure of the goodness of predictions. A large loss function means bad predictions and the way we choose the loss functions is dependent on the problem at hand or the distribution of the target variable.

L1 Loss (Mean-absolute error Loss, MAE)

$$L(y, \hat{y}, \theta) = \frac{1}{n} \sum_{i=1}^n \|y_i - \hat{y}_i\|_1$$

Mean-squared error (MSE) Loss

$$L(y, \hat{y}, \theta) = \frac{1}{n} \sum_{i=1}^n \|y_i - \hat{y}_i\|_2^2$$

Binary Cross Entropy (BCE) for yes/no classification

$$L(y, \hat{y}, \theta) = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log[1 - \hat{y}_i])$$

Cross Entropy for multi-class classification

$$L(y, \hat{y}, \theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^k (y_{ik} * \log \hat{y}_{ik})$$

### Notes:

- The minus sign is added thanks to the fact that we use  $\log(p)$  where  $p$  is a probability, a value in the range of  $[0, 1]$ . Therefore, the logarithm function produces negative values. To overcome this, we add the minus.
- Note** That for the **multi-class** classification we use the **one-hot-encoding** form of the ground-truth  $y_{ik}$

### Notations

- Ground truth:  $y$
- Prediction:  $\hat{y}$
- Loss function:  $L(y, \hat{y}) = L(y, f_\theta(x))$
- Neural Network:  $f_\theta(x)$   $\Rightarrow$  Goal: minimize the loss wrt  $\theta \rightarrow$  Gradient Descent

The loss curve starts high and ideally goes lower and lower. The loss function can also be plotted against  $\theta$ . And then we use gradient descent to find the optimal  $\theta$ .

### Gradient Descent

The updated  $\theta$  where  $\alpha$  is the step size or the learning rate (how fast we are going down the loss function).

$$\theta = \theta - \alpha \nabla_\theta L(y, f_\theta(x))$$

$$\theta^* = \arg \min L(y, f_\theta(x))$$

In order to compute the gradient for multi layers, we start from the nth layer and backpropagate to the first one using the chain rule.

Given the MSE Loss, we can derive the closed form formula for the derivative:

$$\nabla_\theta L(y, f_\theta(x)) = \frac{1}{n} \sum_i^n (W \cdot x_i - y_i) \cdot x_i^T$$

**Why gradient descent?** Not always the best option, but easy to compute using compute graphs. Other methods: Newtons method, Adaptive moments, Conjugate gradient.

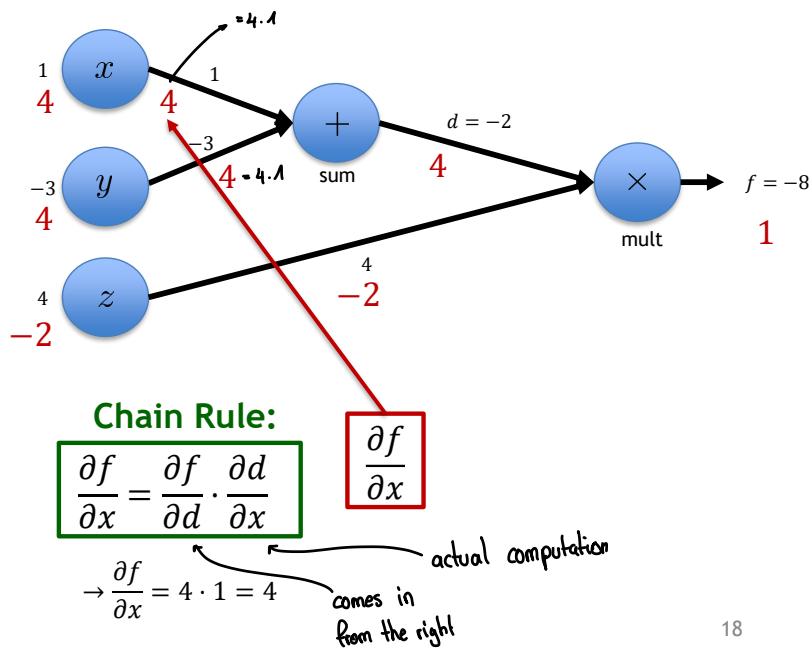
### Summary

- NN are computational graphs, Goal: for a given train set, find optimal weights
- Optimization is done using gradient-based solvers: many options
- Gradients are computed via backpropagation: Nice because can easily modularize complex functions

# IV Optimization and backpropagation

Optimization schemes are based on computing gradients  $\nabla_{\theta} L(\theta)$ . Compute gradients analytically, but what if function is too complex → break down gradient computation: Backpropagation

## IV.1 Backpropagation



18

Figure IV.1: Exemplary computational graph for  $f(x, y, z) = (x + y) \cdot z$   
Forward pass in black and Backward pass in red

**Forward pass** Just calculate  $f$ . takes output from previous layer, performs operation, returns result. caches values needed for gradient computation during backprop

**Backward pass** Start from the end and compute the partial derivative for every node:  $d = x + y$  and  $f = d \cdot z$ . takes upstream gradient, returns all partial derivatives.

$$\frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1, \frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

Compute values with the chain rule.

### Notations

- $x_k$  input variables
- $w_{l,m,n}$  network weight:  $l$  which layer,  $m$  which neuron in layer,  $n$  which weight in neuron
- $\hat{y}_i$  computed output ( $i$  output dimension:  $n_{out}$ )
- $y_i$  ground truth targets

- L loss function

The bias is given as  $b_i$  in  $\hat{y}_i = A(b_i + \sum_k^k x_k W_{i,j})$ . If we want to compute the gradient of the loss function  $L$  wrt all weights  $W$ , we use the chain rule:

$$\begin{aligned} \text{Loss function: } L &= \sum_i L_i \\ \text{L2 Loss: } L_i &= (\hat{y}_i - y_i)^2 \\ \text{Chain rule to compute partials: } \frac{\partial L}{\partial w_{i,k}} &= \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_{i,k}} \end{aligned}$$

⇒ We compute the gradients wrt all the weights and all the biases.

## IV.2 Gradient Descent

For a given training pair  $\{x, y\}$ , we want to update all weights, i.e. we need to compute the derivatives wrt all weights:

$$\nabla_W f_{\{x,y\}}(W) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \vdots \\ \vdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{bmatrix}$$

Gradient steps in direction of negative gradient:

$$W' = W - \alpha \nabla_W f_{\{x,y\}}(W),$$

where  $\alpha$  is the learning rate and  $\nabla$  is the gradient w.r.t the weights.

Gradients can be computed analytically yet it is computationally expensive therefore it is easier to break down the gradient computation and utilize backpropagation.

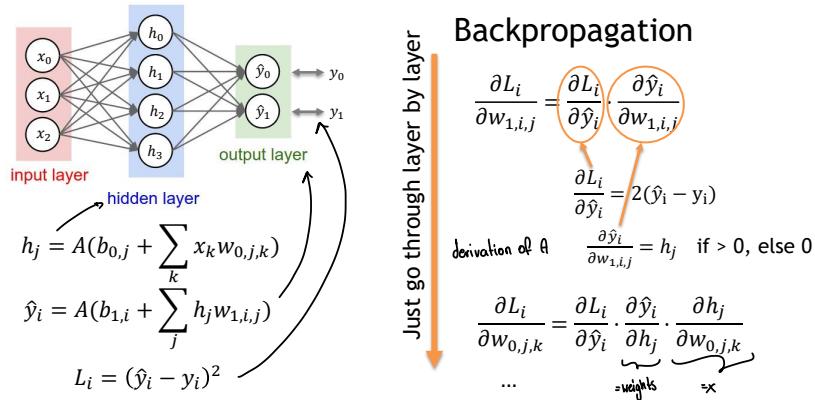


Figure IV.2: Gradient Descent for NN

How many unknown weights are there in input layer (3) hidden layer (4) and output (2)?

The output layer has:  $2 \cdot 4 + 2$  weights → #neurons \* # inputchannels + #biases

So the hidden layer will have:  $4 \cdot 3 + 4$  weights

### Derivatives of Cross Entropy Loss

Loss:  $L = - \sum_{i=1}^{n_{out}} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$

Output:  $\hat{y}_i = \frac{1}{1+e^{-s_i}}$ , Scores:  $s_i = \sum_j h_j w_{ji}$

Gradients of weights of last layer:  $\frac{\partial L_i}{\partial w_{ji}} = \frac{\partial L_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_i} \frac{\partial s_i}{\partial w_{ji}}$  with  $\frac{\partial L_i}{\partial \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)}$ ,  $\frac{\partial \hat{y}_i}{\partial s_i} = \hat{y}_i(1 - \hat{y}_i)$ ,  $\frac{\partial s_i}{\partial w_{ji}} = h_j$   
 $\Rightarrow \frac{\partial L_i}{\partial w_{ji}} = (\hat{y}_i - y_i)h_j$ ,  $\frac{\partial L_i}{\partial s_i} = \hat{y}_i - y_i$

Gradients of the first layer are:

$$\begin{aligned}\frac{\partial L}{\partial h_j} &= \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_j} \frac{\partial s_j}{\partial h_j} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji} \\ \frac{\partial L}{\partial s_j^1} &= \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_i} \frac{\partial s_i}{\partial h_j} \frac{\partial h_j}{\partial s_j^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji}(h_j(1 - h_j)) \\ \frac{\partial L}{\partial w_{kj}^1} &= \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_j^1} \frac{\partial s_j^1}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i)w_{ji}(h_j(1 - h_j))x_k\end{aligned}$$

Example for Two-layer NN for regression with ReLU activation on Lecture 4, slide 44.

**How to pick a good learning rate?** If the learning rate is too high, then the model may not be able to converge to an optimal solution or it may converge too fast to a suboptimal solution and if the learning rate is too low, then the model will converge very slowly. We should find some sort of sweet spot so that the function converges fast and is not locked in saddle points.

**How to compute the gradient for large training set?** We can compute the gradient for individual training pairs and or training batches and then we take the average.

## IV.3 Regularization

To close the generalization gap we can add regularization terms to the loss function such as the L2 reg or L1 reg → prevention from overfitting, make training harder, less likely to remember samples (Training error goes down, Validation error goes down and up, Difference = Generalization gap) (We can also use early stopping, dropout or max norm reg.)

$$\begin{aligned}\text{Loss function: } L(y, \hat{y}, \theta) &= \sum_{i=1}^n (x_i \theta_{ji} - y_i)^2 + \lambda R(\theta) \quad \text{Regularization term} \\ \text{L2 Reg: } R(\theta) &= \sum_{i=1}^n \theta_i^2 \\ \text{L1 Reg: } R(\theta) &= \sum_{i=1}^n |\theta_i|\end{aligned}$$

L1: enforces sparsity (= Seltenheit), focus the attention to a few key features (probably more overfitting)

L2: enforces that the weights have similar values, will take all information into account to make decisions

The goal of regularization is to not overfit the data and it will make the training error higher, basically to generalize. It wants to make the training harder so that the network has to learn better features. The reg term should also not be too strong. We should also find a good balance for this term.

Regularization is any strategy that aims to lower validation error and increase training error.

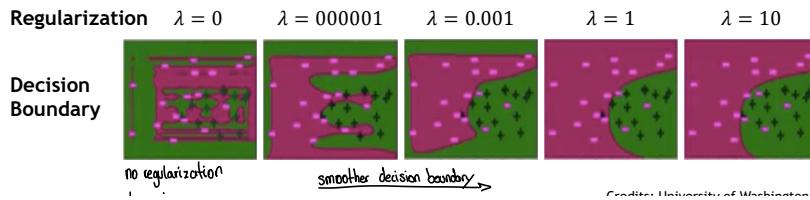


Figure IV.3: Different lambdas for Regularization

## Forward and Backward passes of Loss function

---

```
1 # L1: Forward
2 result = np.abs(y_out - y_truth)
3
4 # L1: Backward
5 gradient = y_out - y_truth
6
7 zero_loc = np.where(gradient == 0)
8 negative_loc = np.where(gradient < 0)
9 positive_loc = np.where(gradient > 0)
10
11 gradient[zero_loc] = 0
12 gradient[positive_loc] = 1
13 gradient[negative_loc] = -1
14
15 # MSE: Forward
16 result = (y_out - y_truth)**2
17
18 # MSE: Backward
19 gradient = 2 * (y_out - y_truth)
20
21 # BCE: Forward
22 result = -y_truth*np.log(y_out)-(1-y_truth)*np.log(1-y_out)
23
24 # BCE: Backward
25 gradient = -(y_truth/y_out) + (1-y_truth)/(1-y_out)
```

---

In the backward pass of the models always multiply with `d_out` or `self.cache`.

# V Scaling Optimization and Stochastic Gradient Descent

## V.1 Gradient Descent

As explained in the previous lecture, having  $w$  to optimize over, first we initialize it and then find the slope of the derivative of the function (in this case: the loss function) and we take gradient steps in the direction of the negative gradient. These steps that we take are dependent on the learning rate ( $\alpha$ ). When  $\alpha$  is too large the step is bigger and vice versa. That is why  $\alpha$  should be appropriately set.

### Convergence

For a convex function: local minimum = global minimum BUT NN are non-convex (many different local minima, no practical way to say which is globally optimal).

Plateau is when the learning rate is significantly small and are the gradient steps which leads to slow convergence or no convergence at all.

Given a loss function  $L$  and a single training sample  $\{x_i, y_i\}$ . Find the best model parameters  $\theta = \{W, b\}$  such that the cost function  $L_i(\theta, x_i, y_i)$  is minimised. The gradient descent steps for single training sample:

1. Initialize  $\theta^1$  with random values
2.  $\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L_i(\theta^k, x_i, y_i)$  (Computed via backpropagation)
3. Iterate until convergence  $|\theta^{k+1} - \theta^k| < \epsilon$

Multiple training samples

1. Take the average cost function  $L = \frac{1}{n} \sum_{i=1}^n L_i(\theta, x_i, y_i)$
2. Update step for multiple samples:  $\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L(\theta^k, x_{1..n}, y_{1..n})$  with  $\nabla_{\theta} L(\theta^k, x_{\{1..n\}}, y_{\{1..n\}}) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L_i(\theta^k, x_i, y_i)$
3. Often written as  $\nabla L = \sum_{i=1}^n \nabla_{\theta} L_i$  (omitting  $1/n$  not wrong, but means rescaling the learning rate)

### Optimal Learning Rate with Line search

1. Compute gradient  $\nabla_{\theta} L = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L_i$
2. Optimize for optimal step  $\alpha$ :  $\arg \min_{\alpha} L \left( \underbrace{\theta^k - \alpha \nabla_{\theta} L}_{\theta^{k+1}} \right)$
3.  $\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} L$

Problem: not practical for DL since need to solve huge system every step.

### Stochastic Gradient Descent (SGD)

Since the gradient descent is very computationally complex, we can also use stochastic gradient descent which takes only a random portion of the data to train on.

We can consider the problem as empirical risk minimization where the loss over the training data is expressed as the expectation of all samples:  $\frac{1}{n} \sum_{i=1}^n L_i(\theta, x_i, y_i) = E_{[1..n]}[L_i(\theta, x_i, y_i)]$  where the expectation can be approximated with a small subset of the data:

$$E_{[1..n]}[L_i(\theta, x_i, y_i)] = \frac{1}{|S|} \sum_{j \in S} (L_i(\theta, x_j, y_j)) \quad \text{with } S \subseteq \{1..n\}$$

A minibatch is a smaller subset of the data where  $m << n$  and the minibatch size is also a hyperparameter typically of a power of 2:

$$B_i = \left\{ \{\mathbf{x}_1, \mathbf{y}_1\}, \{\mathbf{x}_2, \mathbf{y}_2\}, \dots, \{\mathbf{x}_m, \mathbf{y}_m\} \right\} \\ \left\{ B_1, B_2, \dots, B_{n/m} \right\} \quad (\text{V.1})$$

n: number of total samples, m: number of samples in batch, n/m: number of batches.

Smaller batch size means greater variance in the gradients (noisy updates).

An epoch is a complete pass through the train set. So after an epoch, we have seen all the samples from the training set.

- one epoch = one forward pass and one backward pass of all the training examples.
- number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Every time we do an iteration the mini batch is different since it takes random data from the training set on each run. Compute the gradient for one minibatch, then update the weights.

When want to minimize the function  $F(\theta)$  with the stochastic approximation:

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X))$$

where  $\alpha_1, \alpha_2 \dots \alpha_n$  is a sequence of positive step-sizes and  $H(\theta^k, X)$  is the unbiased estimate of  $\nabla F(\theta^k)$ , i.e.  $\mathbb{E}[H(\theta^k, X)] = \nabla F(\theta^k)$ .

It will converge to a local (global) minimum if the following conditions are met:

- $a_n \geq 0, \forall n \geq 0$
- $\sum_{n=1}^{\infty} \alpha_n = \infty$
- $\sum_{n=1}^{\infty} (\alpha_n)^2 < \infty$
- $F(\theta)$  is strictly convex

Robbins and Monro sequence is:  $a_n \propto \frac{a}{n}$ , for  $n > 0$

**Problems** SGD has two main problems:

- The gradient is scaled equally across all dimensions. Cannot independently scale directions, need to have conservative min learning rate to avoid divergence, slower than necessary
- Finding a good learning rate is quite difficult

**Gradient Descent with Momentum:** With momentums we want to accumulate gradients over time, we want it go slower on the vertical jumps and faster on the horizontal ones (see Fig. V.1). So we want to take the history of gradients. Compute velocity

$$v^{k+1} = \beta v^k - \alpha \nabla_{\theta} L(\theta^k) \quad \theta^{k+1} = \theta^k + v^{k+1}$$

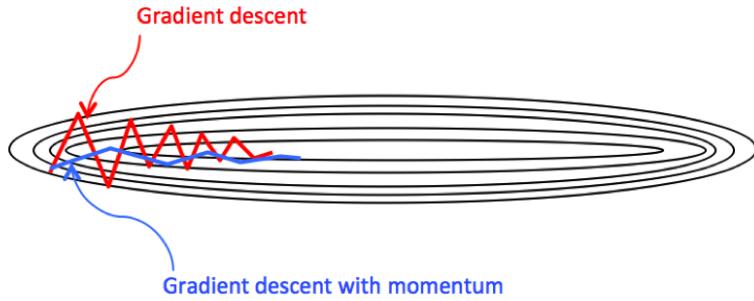


Figure V.1: Gradient Descent with Momentum

where  $\beta$  is the accumulation rate/momentum often set to 0.9 and  $\alpha$  is the learning rate. This velocity is an exponential weights average of the gradients and it is vector valued. Steps will be largest when a sequence of gradients all point to the same direction. If  $\beta = 0$  no momentum, if  $\beta$  is high only about previous gradient descents → Overshooting.

**Important note:** Momentum speeds up movement along directions of strong improvement (loss decrease) and also helps the network avoid local minima.

**Nesterov momentum** is when we use look ahead momentum, which means update theta over v then update the next v over the current updated theta and update the theta again → overcome local minima (not always good)

$$\begin{aligned}\tilde{\theta}^{k+1} &= \theta^k + \beta v^k \\ v^{k+1} &= \beta v^k - \alpha \nabla_{\theta} L(\tilde{\theta}^{k+1}) \\ \theta^{k+1} &= \theta^k + v^{k+1}\end{aligned}$$

First you make a big jump in the direction of the previous accumulated gradient. Then measure the gradient where you end up and make a correction ⇒ Faster if the gradient always in the same direction, but risk of overshooting

**RMSProp (Root Mean Squared Prop)** divides the learning rate by an exponentially-decaying average of squared gradients. Which in other terms means: if we're having fluctuations, don't go so far in that direction.

$$\begin{aligned}s^{k+1} &= \beta s^k + (1 - \beta)[\nabla_{\theta} L \circ \nabla_{\theta} L] \\ \theta^{k+1} &= \theta^k - \alpha \frac{\nabla_{\theta} L}{\sqrt{s^{k+1}} + \epsilon}\end{aligned}$$

(Uncentered) variance of gradients:  $s_{k+1}$  is a second moment. Division by square gradients, if you divide in y-direction it will be large and if you divide in x-direction it will be small for Fig. V.1.

RMS-prop does not use momentum (the mechanism).

⇒ Damping the oscillations for high-variance directions. RMSProp can use faster learning rate since it is less likely to diverge (speed up learning rate, second moment)

**Adam Moment Estimation (ADAM)** uses both the momentum and RMSProp.

$$m^{k+1} = \beta_1 \cdot m^k + (1 - \beta_1)\nabla_{\theta} L(\theta^k)$$

$$v^{k+1} = \beta_2 \cdot v^k + (1 - \beta_2)[\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{m^{k+1}}{\sqrt{v^{k+1}} + \epsilon}$$

However if  $k=0$ , the first estimates are very bad therefore we need a bias correction since  $m^0 = 0$  and  $v^0 = 0$ .

Final update rules:

$$\begin{aligned} m^{k+1} &= \beta_1 \cdot m^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k) \\ v^{k+1} &= \beta_2 \cdot v^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)] \\ \hat{m}^{k+1} &= \frac{m^{k+1}}{1 - \beta_1^{k+1}} \\ \hat{v}^{k+1} &= \frac{v^{k+1}}{1 - \beta_2^{k+1}} \\ \theta^{k+1} &= \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \epsilon} \end{aligned}$$

The  $m^{k+1}$  is the **first order momentum**, because it utilizes the **first moment**, which is the mean. It accumulates a "running mean" of the gradients, not because it averages over each individual batch. Hence, the notation  $m$  - mean.

The  $v^{k+1}$  is the **second order momentum**, because it utilizes the **second moment**, which is the variance. More accurately, the "**uncentered** variance".

Recall that the variance is

$$Var(X) = E[(X - \mu)^2]$$

Here, we drop the mean  $\mu$ , or just assume it is zero. Therefore, it is **uncentered**. So

$$\begin{aligned} \nabla_{\theta} L(\theta^k)^2 &= [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)] \\ Var(\nabla_{\theta} L(\theta^k)) &\sim E[\nabla_{\theta} L(\theta^k)^2] \sim \beta_2 \cdot v^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)] \end{aligned}$$

Which is accumulated iteratively.

It accumulates a "running variance" of the gradients. Hence, the notation  $v$  - variance.

#### Notes:

- Momentum is a technique to utilize previous gradient knowledge, to boost the training steps in the correct direction, while **moment** is a statistical term, representing the mean, variance and more.
- The moment term refers to a mean over some random-variable, therefore the momentum mechanism doesn't use the first order moment.

## V.2 The Newton Method

We can approximate the loss function by a second order Taylor series expansion.

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} L(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H(\theta - \theta_0)$$

Differentiate and equate to 0: Update step:  $\theta^* = \theta_0 - H^{-1} \nabla_{\theta} L(\theta)$

The computational complexity is  $\mathcal{O}(k^3)$ .

The Newton's method exploits the curvature to take a more direct route. We also got rid of the learning rate.

potential downsides: Faster convergence in terms of number of iterations "mathematical view" Approximating the inverse Hessian is highly computationally costly, not feasible for high-dimensional datasets

## V.3 Other Optimization methods

- Broyden-Fletcher-Goldfarb-Shanno algorithm: family of quasi-Newton methods, approximation of the inverse of the Hessian:  $\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$
- Gauss-Newton:  $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$  BUT second derivatives are often hard to obtain
- Levenberg: damped version of Gauss-Newton  $(J_F(x_k)^T J_F(x_k) + \lambda I)(x_k - x_{k+1}) = \nabla f(x_k)$
- Levenberg-Marquardt:  $(J_F(x_k)^T J_F(x_k) + \lambda \cdot \text{diag}(J_F(x_k)^T J_F(x_k))) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$  Instead of a plain Gradient Descent for large  $\lambda$ , scale each component of the gradient according to the curvature: Avoids slow convergence in components with a small gradient

Standard: Adam

Fallback option: SGD with momentum

Newton, L-BFGS, GN, LM only if you can do full batch updates (doesn't work well for minibatches!!)

SGD is specifically designed for minibatch

When you can, use 2nd order method (it's just faster)

GD or SGD is not a way to solve a linear system!

---

```

1 #ReLU: Forward
2     outputs = np.maximum(x, 0)
3     cache = x
4
5 #ReLU: Backward
6     x = cache
7     dx = dout
8     dx[x < 0] = 0
9
10 #Affine layer: Forward
11    x_reshaped = np.reshape(x, (x.shape[0], -1))
12    out = x_reshaped.dot(w) + b
13
14 #Affine layer: Backward
15    n = x.shape[0]
16    dw = (np.reshape(x, (x.shape[0], -1)).T).dot(dout) / n
17    dw = np.reshape(dw, w.shape)
18
19    db = np.mean(dout, axis=0, keepdims=False)
20
21    dx = dout.dot(w.T)
22    dx = np.reshape(dx, x.shape)

```

---

# VI Training Neural Networks I

## VI.1 Learning rate

Learning rate: If too high, the model will overshoot during training and will perform poorly on any sample (both train and test data). If it is too low, the model will underfit the data. Underfitting also happens if you try to fit a linear model to non-linear data.

When you're far away from the optimum, the learning rate should be high and then decrease with time, that's why the learning rate decay is introduced.

Possible variants:

- **Fractional Decay:**  $\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch}} \cdot \alpha_0$
- **Step decay:** only every n steps, t is decay rate (often 0.5)  $\alpha = \alpha - t \cdot \alpha$
- **Exponential Decay:** t is decay rate ( $t < 1.0$ )  $\alpha = t^{\text{epoch}} \cdot \alpha_0$

## VI.2 Training

### Definitions

- The training error comes from average minibatch error.
- Bias and Variance: Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. A model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on the train and test data.
- Bias: Underfitting, error caused by model being too simple
- Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. A model with high variance pays a lot of attention to the training data and does not generalize (Overfitting). These kinds of model perform very well on the training set, yet very poorly on the test set.
- Learning means generalization to unknown dataset (i.e., train on known dataset → test with optimized parameters on unknown dataset). Basically, we hope that based on the train set, the optimized parameters will give similar results on different data (i.e., test data).

**Training schedule:** Manually specify learning rate for entire training process

Manually set learning rate every n-epochs

How? Trial and error (the hard way) or Some experience (only generalizes to some degree)

Consider: #epochs, training set size, network size, etc.

### Basic Recipe for Training

Given ground dataset with ground labels

-  $\{x_i, y_i\}$

- $x_i$  is the  $i^{th}$  training image, with label  $y_i$
- Often  $\text{dim}(x) \gg \text{dim}(y)$  (e.g., for classification)
- i is often in the 100-thousands or millions

- Take network  $f$  and its parameters  $w, b$
- Use SGD (or variation) to find optimal parameters  $w, b$  (Gradients from back propagation)

- **Training set ('train'):**
  - Use for training your neural network
  - During training: Train error comes from average minibatch error, Typically take subset of validation every n iterations (backward pass dominates forward pass regarding costs)
- **Validation set ('val'):**
  - Hyperparameter optimization – Check generalization progress
- **Test set ('test'):**
  - Only for the very end
  - NEVER TOUCH DURING DEVELOPMENT OR TRAINING
- **Typical splits**
  - Train (60%), Val (20%), Test (20%)
  - Train (80%), Val (10%), Test (10%)

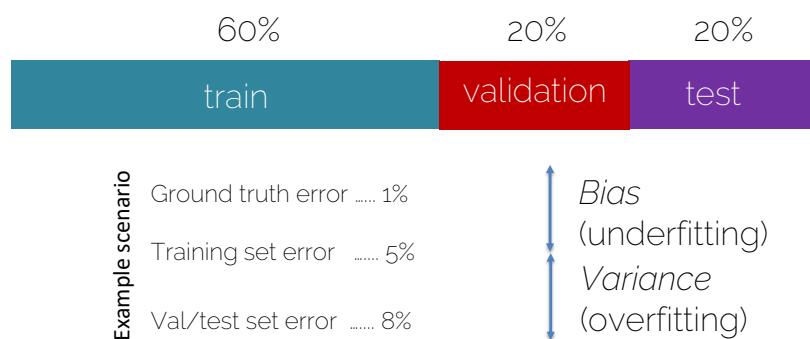


Figure VI.1: Ground truth error: errors in the labels (annotated manually), Training set error: underfitting issue, Val/test error: overfitting issue

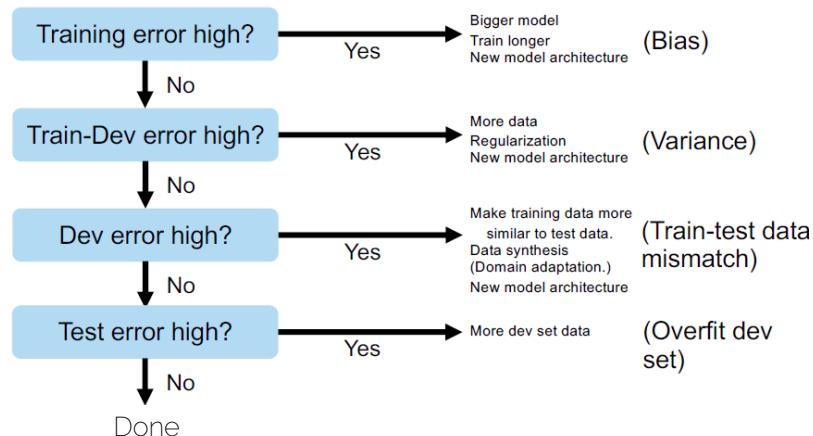


Figure VI.2: Decision tree for finding the error

Training error will go down during training, but validation error can go up because model will memorize data → stop in the middle and other methods

### What does a good and bad training curve look like?

Source: <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>

- Train Learning Curve: Learning curve calculated from the training dataset that gives an idea of how well the model is learning.

- Validation Learning Curve: Learning curve calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

Accuracy and Loss curve will fluctuate: Apply smoothing function

Accuracy should go up / Loss should go down in the beginning very quickly → know quickly if model is working

gets slower at the end → change learning rate (smaller)

## Summary

- Underfitting: Training and validation losses decrease even at the end of training
- Overfitting: Training loss decreases and validation loss increases
- Ideal Training: Small gap between training and validation loss, and both go down at same rate (stable without fluctuations)
- Bad Signs:
  - Training error not going down
  - Validation error not going down
  - Performance on validation better than on training set
  - Tests on train set different than during training
- Bad Practice:
  - Training set contains test data
  - Debug algorithm on test data
  - Never touch during development or training

## VI.3 Hyperparameter tuning

### Methods:

- Manually (most common)
- Grid search (define ranges for the hyperparameters)
- Random search (like grid search, but the points are picked at random and not sequentially)

### Advice:

- If an iteration exceeds 500ms, things get dicey
- look for bottlenecks and estimate total time
- Start small, you can start with #layers / 5
- Check the loss and accuracy curves
- Only make one change at a time

### Find a good Learning Rate

- Use all training data with small weight decay
- Perform initial loss sanity check e.g.,  $\log(C)$  for softmax with  $C$  classes
- Find a learning rate that makes the loss drop significantly (exponentially) within 100 iterations
- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

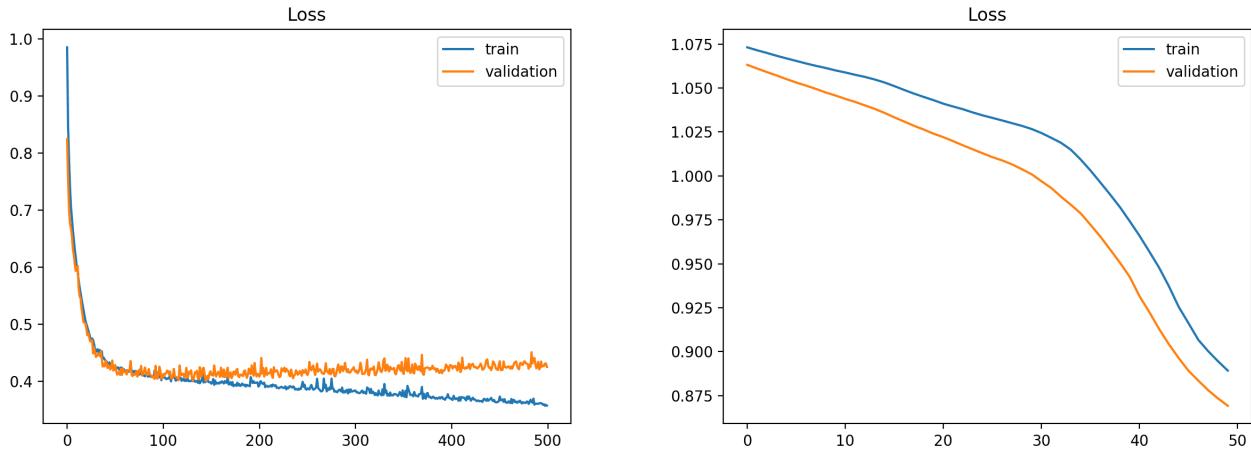


Figure VI.3: **Overfitting** (left): The plot of validation loss decreases to a point and begins increasing again.  
**Underfitting** (right): An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot. This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely.

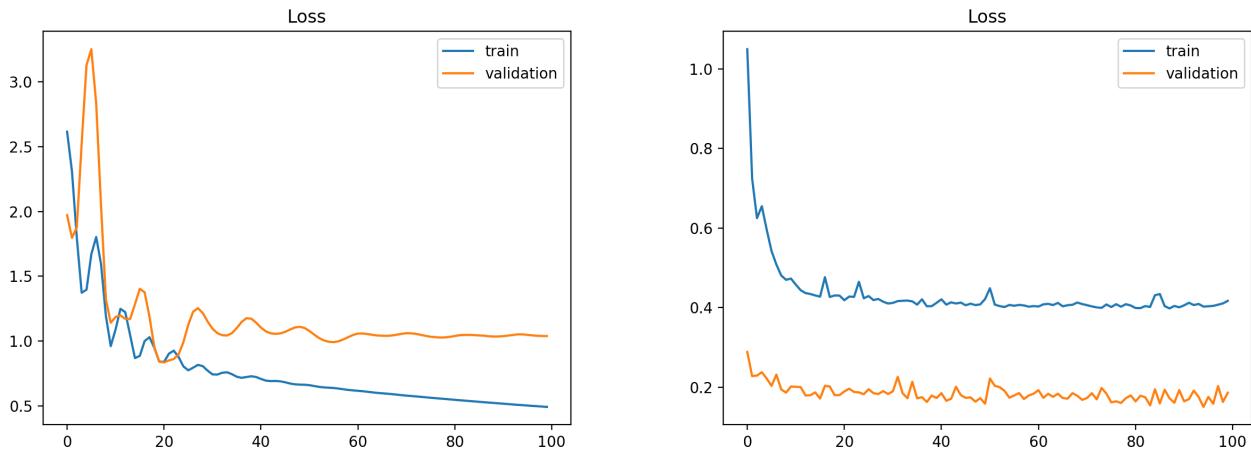


Figure VI.4: **Unrepresentative Train Dataset** (left): This situation can be identified by a learning curve for training loss that shows improvement and similarly a learning curve for validation loss that shows improvement, but a large gap remains between both curves.  
**Unrepresentative Validation Dataset** (right): It may be identified by a validation loss that is lower than the training loss. In this case, it indicates that the validation dataset may be easier for the model to predict than the training dataset.

# VII Training Neural Networks II

## VII.1 Output and Loss Function

Loss Function is calculated on Output Layer (=Prediction) → What shape should the loss function have?

### Naïve Losses

$$L_1 = \sum_{i=1}^n |y_i - f(x_i)|$$

$$L_2 = \sum_{i=1}^n (y_i - f(x_i))^2$$

L2 Loss	L1 Loss
Squared distances	Absolute differences
Prone to outliers	Robust (cost of outliers is linear)
Compute-efficient optimization	Costly to optimize
Optimum is the mean	Optimum is the median

**Sigmoid:** two classes, turns score into a probability

$$p(y_i = 1|x_i, \theta) = \sigma(s) = \frac{1}{1 + e^{-\sum_{d=0}^D \theta_d x_{id}}}$$

**Softmax:** C classes, what is the score of a certain class k?

$$p(y_i|x_i, \theta) = \frac{e^{s_{yi}}}{\sum_{k=1}^C e^{s_k}} = \frac{e^{x_i \theta_{yi}}}{\sum_{k=1}^C e^{x_i \theta_k}}$$

$y_i$ : label (true class),  $C$ : number of classes,  $s$ : score of the class,  $s_{max}$ : maximal score  
For numerical stability it's better to use:

$$p(y_i|x_i, \theta) = \frac{e^{s_{yi}}}{\sum_{k=1}^C e^{s_k}} = \frac{e^{s_{yi} - s_{max}}}{\sum_{k=1}^C e^{s_k - s_{max}}}$$

**Cross Entropy Loss (Maximum Likelihood Estimate):** Create a loss that pushes the probability into the right direction.

How wrong is my neural network? Loss function is low if prediction is correct.

$$L_i = -\log(p(y_i|x_i, \theta)) = -\log\left(\frac{e^{s_{yi}}}{\sum_k e^{s_k}}\right)$$

Total loss is averaged sum of losses.

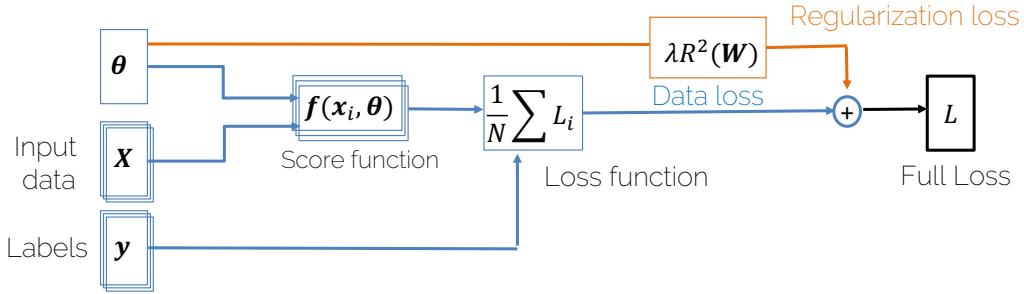
**Hinge Loss (Multiclass SVM Loss):** Subtract score of the ground truth label from score of false predicted labels. Hinge Loss will be 0 if label is predicted correctly.

$$L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

### Multiclass Losses: Hinge vs Cross-Entropy

Both Hinge Losses could be the same, even if one model is better. Cross-Entropy shows the difference. The CE loss always wants to improve (it is never 0, due to the  $\exp()$  effect) whereas the Hinge loss saturates.

## Loss in Compute Graph



Want to find optimal  $\theta$ . (weights are unknowns of optimization problem)

- Compute gradient w.r.t.  $\theta$ .
- Gradient  $\nabla_{\theta} L$  is computed via backpropagation

IzDL: Prof. Niessner, Prof. Leal-Taixe

36

### Summary

- Score Function:  $s = f(x_i, \theta)$
- Data Loss:
  - Cross Entropy:  $L_i = -\log \left( \frac{e^{sy_i}}{\sum_k e^{sk}} \right)$
  - SVM:  $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$
- Regularization Loss: e.g.  $L_2$  -Reg:  $R^2(\mathbf{W}) = \sum w_i^2$
- Full Loss:  $L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R^2(\mathbf{W})$  ( $\lambda$ : how much weight do I give to my regularization)
- Full Loss = Data Loss + Reg Loss

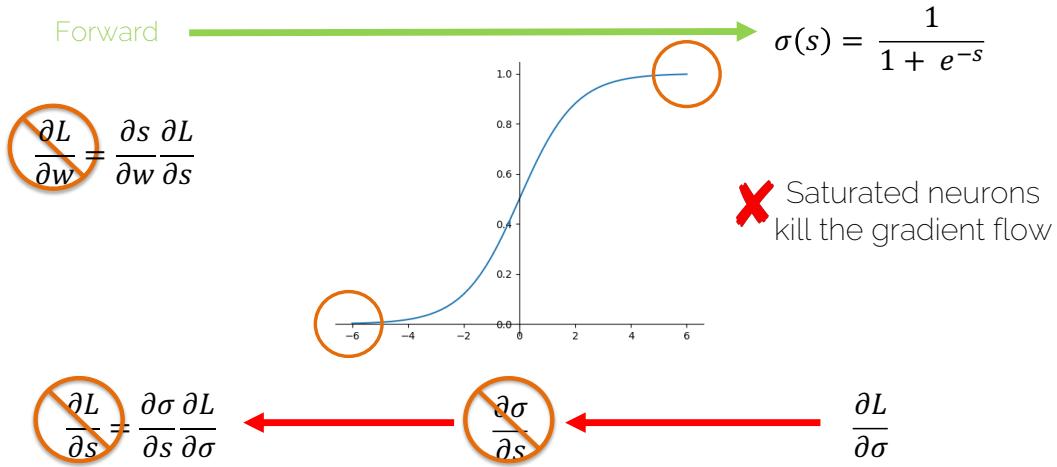
## VII.2 Activation Functions

Bring non-linear motion in hidden units

### Sigmoid Activation

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

This type of activation function is not advised to use since it has a really narrow active region. If value is really high, gradient is very low and value will be ‘killed’ during backpropagation. The sigmoid output is



not zero centered. All the weight updates will be either both positive or both negative.  $w_1$  and  $w_2$  can only be increased or decreased at the same, which would mean that the path would follow a zig-zag manner. We need zero-centered data.

**Tanh Activation:** This function is zero-centered yet it saturates.

$$\sigma(s) = \tanh(s)$$

**ReLU: Rectified Linear Unit:** It does not saturate, it converges fast and has larger and consistent gradients.

$$\sigma(s) = \max(0, x)$$

**Dead ReLU:** Output = 0 all the time, we can't get out of that problem. That's why it's advised to add the slightly positive biases term (makes it likely that they stay active for most inputs:  $f(\sum_i w_i x_i + b))$ ), or use Leaky ReLU instead:

$$\sigma(s) = \max(0.01x, x)$$

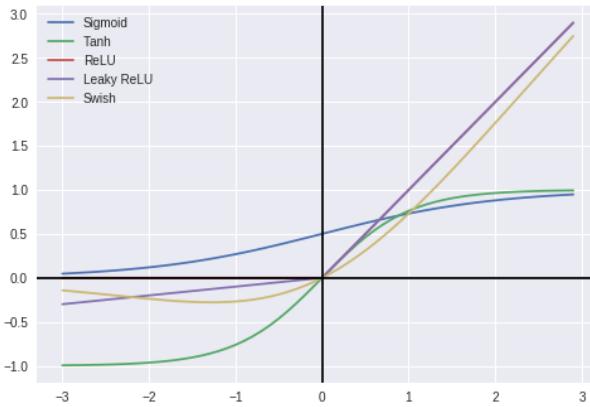
**Parametric ReLU:** Parameter  $\alpha$  instead of 0.01, that is trained of neural network (One more parameter to backprop into)

**Maxout units:** Train parameters with other parameters of the network. Piecewise linear approximation of a convex function with N pieces → Generalization of ReLUs, Linear Regimes, Does not die, does not saturate BUT Increases number of parameters

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

## Summary

- Sigmoid is not really used.
- ReLU is the standard choice.
- Second choice are the variants of ReLU or Maxout
- Recurrent nets will require TanH or similar.



ACTIVATION FUNCTION	EQUATION	RANGE
Linear Function	$f(x) = x$	$(-\infty, \infty)$
Step Function	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$\{0, 1\}$
Sigmoid Function	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Hyperbolic Tangent Function	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$(-1, 1)$
ReLU	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU	$f(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Swish Function	$f(x) = 2x\sigma(\beta x) = \begin{cases} \beta = 0 & \text{for } f(x) = x \\ \beta \rightarrow \infty & \text{for } f(x) = 2\max(0, x) \end{cases}$	$(-\infty, \infty)$

Figure VII.1: <https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-101>

## VII.3 Weight Initialization

Initialization is extremely important because it is not guaranteed to reach the optimum for different starting points.

If  $w=0$  at the start then all the hidden units are all going to compute the same function (grads will be the same) so it would be as if you'd have just one neuron (No symmetry breaking).

So, initialize weights randomly (Gaussian with zero mean and standard deviation 0.01). Yet if they are small numbers then we would face vanishing gradients (from small outputs of layer 1) and if the weights would be big numbers we would also face vanishing gradients caused by saturated activation function.

Use **Xavier Initialization** which ensures the variance of the output is the same as the input  $\rightarrow$  Gaussian with

- Mean:  $\mu = 0$
- Variance:

$$\begin{aligned} \text{Var}(s) &= \text{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \text{Var}(w_i x_i) \\ &= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i) \text{Var}(w_i) \\ &= \sum_i^n \text{Var}(x_i) \text{Var}(w_i) = n(\text{Var}(w) \text{Var}(x)) \end{aligned}$$

- Ensure that the variance of the output is the same as the input:  $\text{Var}(s) = \text{Var}(x) \rightarrow n \cdot \text{Var}(w) \text{Var}(x) = \text{Var}(x) \rightarrow$ 
  - Tanh:  $\text{Var}(w) = \frac{1}{n}$
  - ReLU (Xavier/2 Initialization):  $\text{Var}(w) = \frac{1}{n/2} = \frac{2}{n}$
- Notice:  $n$  is the number of input neurons for the layer of weights you want to initialize. This  $n$  is not the number  $N$  of input data  $X \in R^{N \times D}$ . For the first layer  $n = D$
- Side note:  $E[X^2] = \text{Var}[X] + E[X]^2$  and if  $X, Y$  are independent:  $\text{Var}[XY] = E[X^2Y^2] - E[XY]^2$ ,  $E[XY] = E[X]E[Y]$

Image Classification	Output Layer	Loss function
Binary Classification	Sigmoid	Binary Cross entropy
Multiclass Classification	Softmax	Cross entropy

Other Losses:

SVM Loss (Hinge Loss), L1/L2-Loss

Initialization of optimization

- How to set weights at beginning

# VIII Training Neural Networks III

## VIII.1 Data Augmentation

A classifier has to be invariant to a wide variety of transformations. So to help it, we can synthesize data simulating plausible transformations.

Examples:

- Brightness and Contrast: More robust to illumination changes
- Random Crops (and Resizing): Sometimes image only half on the picture → not showing the full image helps network to handle cropped images
  - Training random crops: Pick a random  $L$ , resize training image: short side  $L$ , randomly sample crops
  - Testing fixed set of crops: Resize image at  $N$  scales, 10 fixed crops
- Flips
- Rotations
- Combinations of the above

Use the same data augmentation when comparing two networks (part of your network design).

We have online and offline data augmentation. The former is done as a pre-processing step to increase the size of the dataset. The latter happens when we apply transformations in mini-batches and then feed it to the model.

## VIII.2 Advanced Regularization

### Early Stopping

Stop at the point where the validation error stops decreasing. You can have an impatience parameter set. For example if impatience is 3, then the iterative training will stop when there's 3 consecutive losses that are increasing (overfitting).

### Bagging and Ensemble Methods

Ensemble: Train multiple models average their results (e.g. different algorithm for optimization or change the objective function/loss function). If errors are uncorrelated, the expected combined error will decrease linearly with the ensemble size.

Bagging: Uses  $k$  different datasets (can be overlapping, drawn from the bigger dataset). Models specialize on a specific thing what is common in the dataset (e.g. white and black dogs). Problem: need to train 3 networks.

### Dropout

- Forward: Disable a random set of neurons (typically  $p=0.5$ ). For each neuron there's a  $p$  possibility that it's dropped (nodes are not used in the forward pass)  $\rightarrow$  only  $p$  of the capacity, only half of the features are used, decision needs to be made with fewer information.
- Backward: reprogram some of the active neurons to detect some of the ignored features  $\rightarrow$  other neurons are responsible of detecting the features, one than more way to detect features  $\rightarrow$  redundant representations, base your scores on more features  $\rightarrow$  considerable as a model ensemble (model of inactive neurons and model of active neurons, trained on a different set of data (mini-batch) and with SHARED parameters)
  - $\Rightarrow$  Reducing the co-adaption between neurons (neurons depending on other neurons doing their job)
- Testing: All neurons are turned on, no dropout  $\rightarrow$  Conditions at train and test time are not the same say  $z = (\theta_1 x_1 + \theta_2 x_2) \cdot p$  (dropout probability)
  - Expectation value for  $z$ :  $E[z] = \frac{1}{4}(\theta_1 0 + \theta_2 0 + \theta_1 x_1 + \theta_2 0 + \theta_1 0 + \theta_2 x_2 + \theta_1 x_1 + \theta_2 x_2) = \frac{1}{2}(\theta_1 x_1 + \theta_2 x_2)$
  - 1/2 corresponds to the Dropout probability  $p = 0.5 \rightarrow$  Weight scaling inference rule
- Efficient bagging method with parameter sharing, dropout reduces the effective capacity of a model  $\rightarrow$  larger models, more training time

## VIII.3 Batch Normalization

Normally used as Regularization technique

Goal: Activation do not die out

Wish: Unit Gaussian activations (in our example)

Mean of your mini-batch examples over feature  $k \rightarrow$  force unit Gaussian in each dimension of the features (Gaussian as input and as output):  $\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$

For NN in general: BN normalizes the mean and the variance of the inputs to your activation functions

A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions.

Implementation

1. Normalize
2. Allow network to change the range:  $y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$  where  $\gamma$  and  $\beta$  get optimized during backprop  $\rightarrow$  learned as any other hyperparameter, the network can learn to undo the normalization:  $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$  and  $\beta^{(k)} = E[x^{(k)}]$

Is it okay to treat dimensions separately? (Computed for each  $k$ )

- Shown empirically that even if features are not correlated, convergence is still faster with this method
- All the biases of the layers before BN can be set to zero since they will be canceled out by BN anyway. BN shifts the activation by their mean values and any constant will get canceled.

Train-time: mean and variance is taken over the mini-batch

Test-time: compute the mean and variance by running an exponentially weighted averaged across training mini-batches.  $\sigma_{test}^2$  and  $\mu_{test}$ :

$$Var_{running} = \beta_m * Var_{running} + (1 - \beta_m) * Var_{minibatch}$$

$$\mu_{running} = \beta_m * \mu_{running} + (1 - \beta_m) * \mu_{minibatch}$$

- Sometimes,  $\beta_m$  is called as "momentum", which I find falsely so.
- By default, but could be manually changed as a hyperparameter  $\beta_m = 0.99$ . Therefore, the new calculated mean and variance over the current batch have but little effect on the running variables.

⇒ Very deep nets are much easier to train: more stable gradients, a much larger range of hyperparameters works similarly when using BN

Drawbacks: As we reduce the batch size (< 8), the statistics for mean and variance get less and less accurate and BN starts to deliver not so good results → use Layer Norm, Instance Norm, Group Norm (the error stays quite constant despite the batch size)

The non-linearity of the Batchnorm during training is due to the fact that each batch out of the whole dataset is normalized by different values (mean and std). During the inference time (testing), it is considered as a linear function, since those values are a constant (running-mean and running-var), that do not depend on the test input.

## VIII.4 Other Normalizations

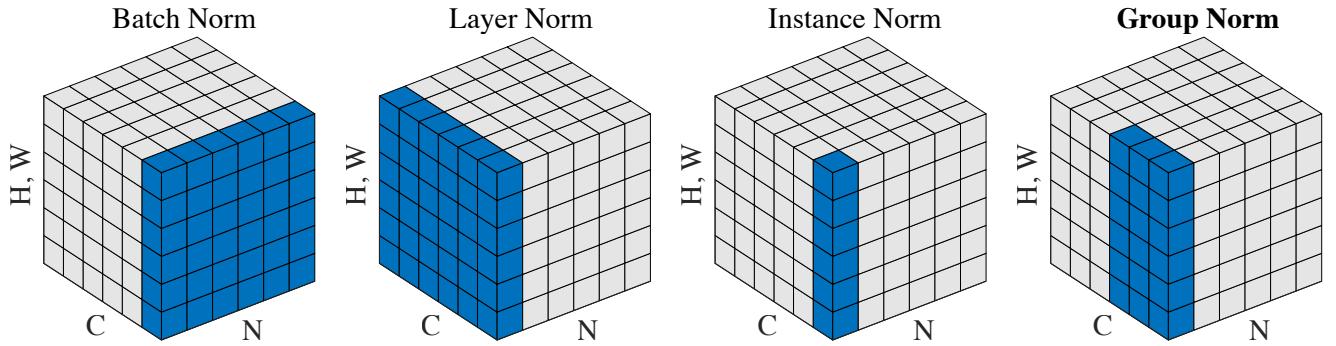


Figure VIII.1: Feature map: spatial sizes H and W, Number of channels C, Number of training samples N.  
The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

## VIII.5 Recap

Why not simply more Layers?

- We cannot make networks arbitrarily complex
- Why not just go deeper and get better? – No structure, It is just brute force, Optimization becomes hard, Performance plateaus/drops

# IX Introduction to CNNs

Using only FC layers to build an architecture is impractical since it is close to brute-force and there's a lot of weights that need to be trained:

- [3, 5, 5] image and 3 neuron in FC 75x3 weights need to be trained
- image sizes are 1k x 1k then we have 3 billion weights

Also there's no structure, optimization becomes hard and the performance drops.

Instead use a different structure: layer with structure, weight sharing (share the same weights for different parts of the image)

## IX.1 Convolution

A convolution is an application of a filter to a function:  $f * g = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$  (The ‘smaller’ function is typically called the filter kernel)

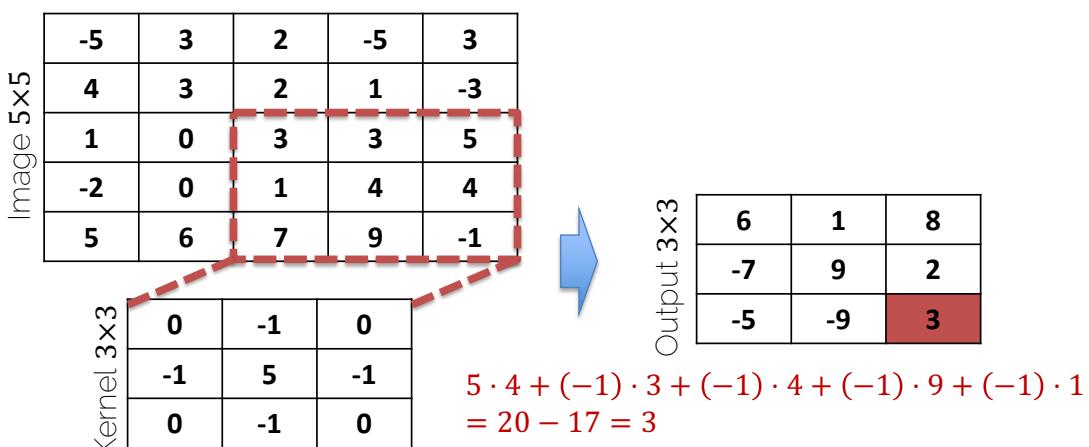


Figure IX.1: The filter kernel ( $g$ ) is slid throughout the image ( $f$ ) and computes the single values in the output data ( $f * g$ )

In the discrete case, there's unassigned values at the boundaries and for those we can: shrink (smaller  $f * g$ ) or pad (often with 0s).

Each kernel gives us a different image filter → learn these filters

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \text{ Edge Detection; } \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ Box Mean; } \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \text{ Gaussian blur; } \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

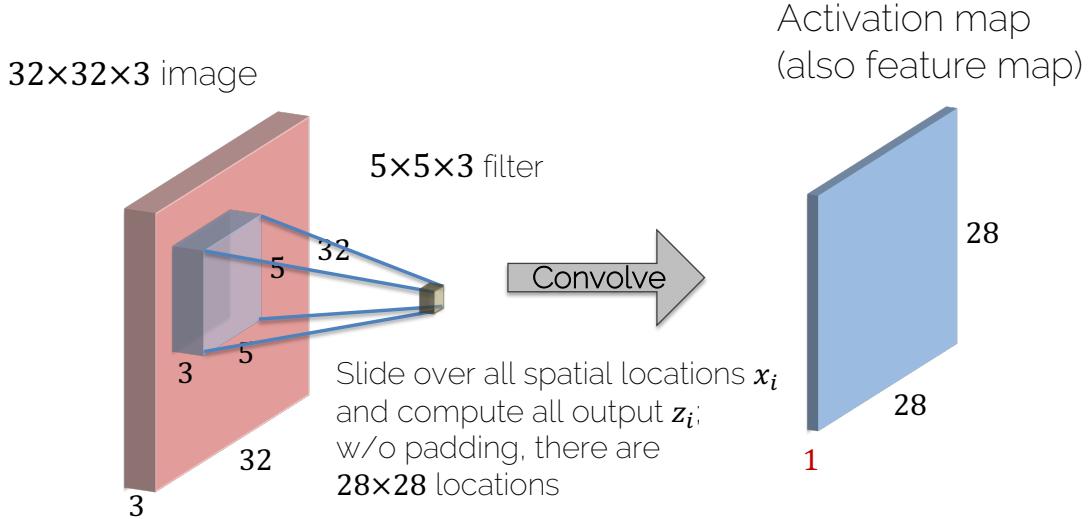
Sharpen

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \text{ Sobel Filter, vertical edge detection; } \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \text{ Scharr filter, vertical edge detection}$$

### Convolutions on RGB Images

The depth of the image must match the depth of the filter (3 for a RGB image)

Create one output for every convolution:  $z_i = \mathbf{w}^T \mathbf{x}_i + b$ , where  $\mathbf{w}^T$  and  $\mathbf{x}_i$  are of size [width \* height \* depth] x 1



## IX.2 Convolution Layer

In a convolution layer we can apply many filters with different weights, so we can have many activation maps → form a Convolution Layer. So a basic layer would be {filter width, filter height, number of different filter banks}.

Each filter captures a different characteristic such as: edges, circles, squares, etc.

## IX.3 Dimensions of a Convolution Layer

N: Input width/height

F: Filter/Kernel size

S: Stride

P: Padding

### Stride

- is the jump the filter does while sliding
- Output is:  $(\frac{N-F}{S} + 1) \times (\frac{N-F}{S} + 1)$
- **Fractions are illegal (Filter must not be outside the image partly). So check the stride.**

### Padding

- Since the sizes get too small too quickly, we use padding. Also like this we can use the corner pixels more than once.
- Pad the whole picture with numbers (mostly zeros) around
- Output:  $(\left\lfloor \frac{N+2 \cdot P-F}{S} \right\rfloor + 1) \times (\left\lfloor \frac{N+2 \cdot P-F}{S} \right\rfloor + 1)$
- Different paddings:
  - Valid: no padding
  - Same: output size = input size, set  $P = \frac{F-1}{2}$

Number of parameters =  $F \cdot F \cdot \text{Depth} \cdot \text{Number of Filters}$

Weight tensor: (Depth, Number of Filters,  $F, F$ )

**Note:** Each kernel filter gets 1 bias.

## IX.4 Convolutional Neural Network (CNN): Pooling

Used as a downsampling layer. It picks the strongest activation in a region. Pooling layer goes directly after convolution layer:

- Conv Layer = ‘Feature Extraction’: Computes a feature in a given region
- Pooling Layer = ‘Feature Selection’: Picks the strongest activation in a region

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

Max pool with  
2x2 filters and stride 2



‘Pooled’ output

6	9
3	4

Input:  $W_{in} \times H_{in} \times D_{in}$  Two hyperparameters: Spatial filter extend F and Stride S

Output:  $W_{out} \times H_{out} \times D_{out}$  where  $W_{out} = \frac{W_{in}-F}{S} + 1$  and  $H_{out} = \frac{H_{in}-F}{S} + 1$  and  $D_{out} = D_{in}$ .  
This is a fixed function, so it does not contain parameters.

Various pooling types:

- Max Pooling
- Average Pooling
- Global Average Pooling

To finalize the CNN we add a FC layer (One or two FC layers typically)

## IX.5 Receptive Field

Receptive field is the size of the region in the input space that a pixel in the output space is affected by; the spatial extent of the connectivity of a convolutional filter. What is the relationship between the input and the output? For example: 3x3 receptive field: 1 output pixel is connected to 9 input pixels.

$$r_k = r_{k-1} + \left( \prod_{i=1}^{k-1} s_i \right) \cdot (f_k - 1) \text{ for } k \geq 2$$

Where  $s$  is the stride and  $f$  is the kernel size. Also,  $r_1$  is the kernel size of the first layer. **Note** that we start from the input and deeper, and not from the requested layer back!

For FC layers, the receptive field is the ENTIRE image. Since each output neuron of that layer is connected to each neuron of the previous one, where each layer represents the entire output, whether it is downsampled or upsampled.

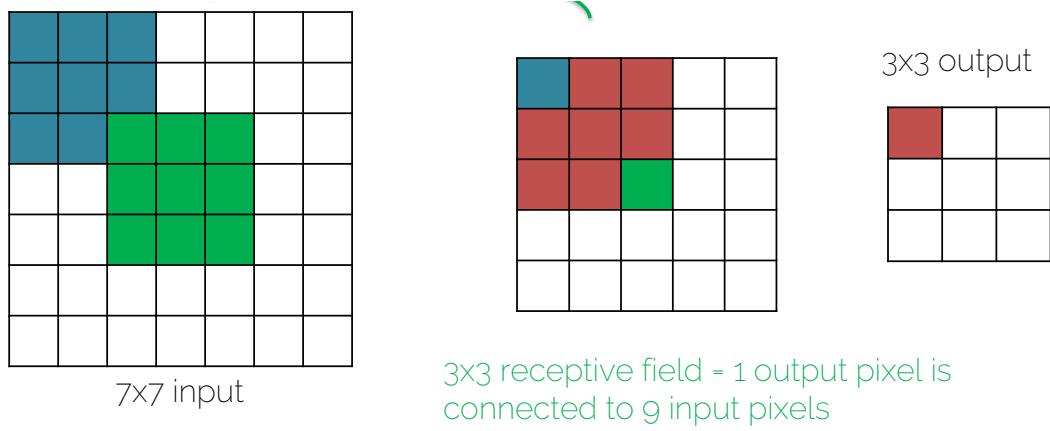


Figure IX.2: Finally: 5x5 receptive field on the original input: one output value is connected to 25 input pixels

Batch Norm for CNN:

$$\begin{aligned}\mu_j &= \frac{1}{N} \sum_{i=1}^N x_{i,j} \\ \sigma_j^2 &= \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \\ \hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \\ y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j\end{aligned}$$

We compute minibatch mean and variance for each channel C.

Spatial batch norm: Compute mean and variance of each channel.

## IX.6 Dropout for CNN

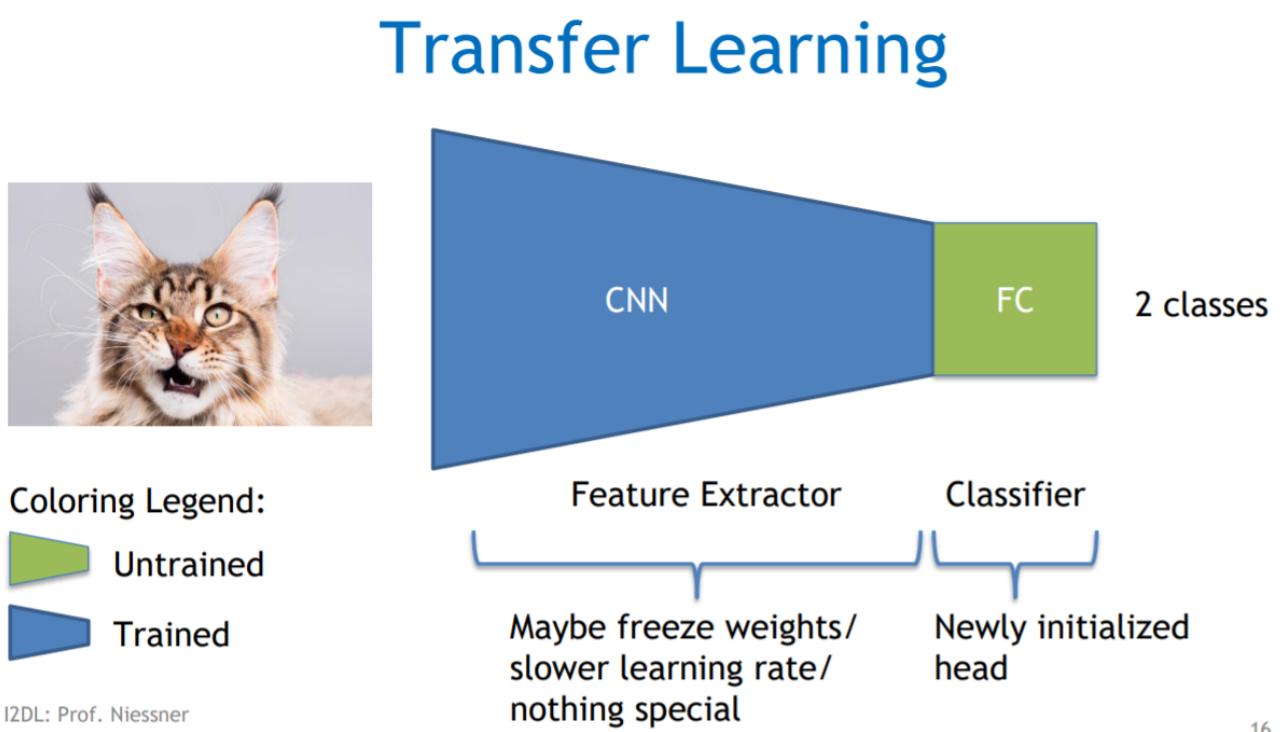
Standard: randomly drop a unit with a certain probability. This does not improve performance in CNN.

Spatial: randomly sets entire feature maps to zero.

## IX.7 Transfer Learning

- The main idea is to **train a new classifier for a new problem, with an existing model.**
- The problems we try to overcome:
  - Training a Deep Neural Network requires a **HUGE** amount of data.
  - Much data might not be available, or it is expensive.
- What are the basic guidelines for such thing?
  - The new problem must be similar to the one the current model already aims to solve. (i.e - taking an already trained model that recognizes dogs, and make it also recognize cats).
  - Can we transfer knowledge from the previous problem to the new one?
  - can we re-use parts of the already trained network?

- The last bullet point is the key idea of this method. A good modern classifier is constructed from two parts - CNN and FC, where the first one is referred as a **Features Extractor** and the latter as **The Classifier**.
- In order to use the existing knowledge:
  - Let us drop the FC part, and replace it with a new fresh untrained one.
  - For the CNN part we might want to:
    - \* Freeze the CNN weights completely during the new training session.
    - \* Slower the learning rate of the CNN only, so it learns a bit more from the new data, but doesn't make a significant impact, that might hurt the efficiency of the model.
    - \* let it train as usual.



# X Popular CNN Architectures

Performance Metrics in ImageNet:

- **Top-1 Score:** check if a sample's top class is the same as its target label
- **Top-5 Score:** check if the label is in the first 5 predictions.
- **Top-5 Error:** percentage of test samples for which the correct class was not in the top 5 predicted classes.

## X.1 LeNet

Firstly used for handwritten digits. Has the following architecture:

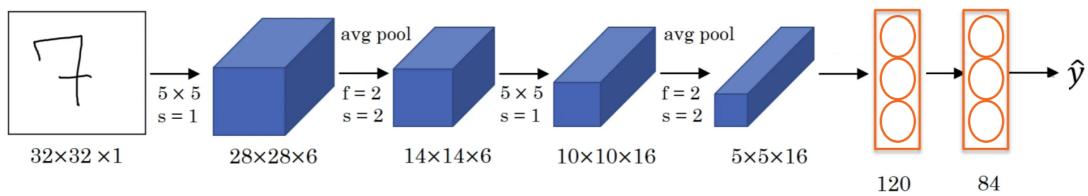


Figure X.1: LeNet Architecture: Conv -> Pool -> Conv -> Pool -> Conv -> FC

- Apply valid convolution: size shrinks (reduced by two pixels on each side)
- 6  $5 \times 5$  convolution filters used in first layer (6 filters as the depth of the convolution obtained is 6)
- Average pooling is used (now: Max pooling much more common)
- Reduce first to 120, then to 84
- Tanh/sigmoid activation is used (not common now)
- Has 60k parameters
- As we go deeper: width, height go down and number of filters go up

## X.2 AlexNet

- Same convolutions and paddings
- Similar to LeNet but  $\sim 1000$  times bigger
- ReLU instead of tanh/sigmoid
- 60m parameters

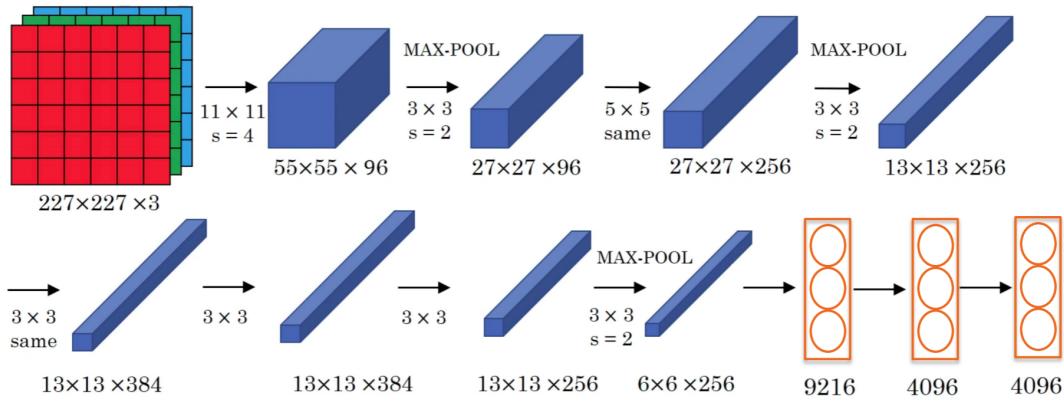


Figure X.2: AlexNet Architecture

### X.3 VGGNet

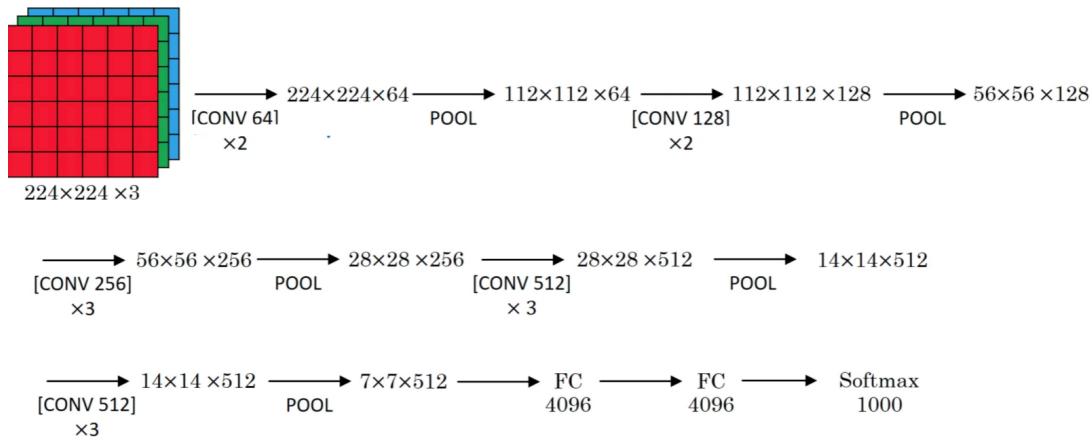


Figure X.3: VGGNet Architecture: Conv -&gt; Pool -&gt; Conv -&gt; Pool -&gt; Conv -&gt; FC

- Same padding
- CONV =  $3 \times 3$  filters with stride 1, same convolutions
- MAXPOOL =  $2 \times 2$  filters with stride 2
- 138m parameters
- It's large but simplicity makes it appealing
- As we go deeper, width and height go down and number of filters up

### X.4 Skip Connections: Residual Block

Since we add more and more layers, training becomes harder and we face vanishing or exploding gradients. Here skip connections are useful to train very deep models. Usually same padding is used since we need same dimensions (you can convert the dims with a matrix of learned weights or zero padding).

**Note**, that since we add the input of  $x^{L-1}$  into  $x^{L+1}$ , this addition **can not** make the neural-network go worse, since it could just learn to skip the next layers, if they do not contribute. That way, as it appeared in **ResNet**, we can harness the power of depth, while usually a deeper network doesn't generalize well.

- Two layers:  $x^{L-1} \rightarrow x^L \rightarrow x^{L+1}$
- Main path:  $x^{L+1} = f(W^{L+1}x^L + b^{L+1})$
- Add skip connection:  $x^{L+1} = f\left(W^{L+1}x^L + b^{L+1} + x^{L-1}\right)$

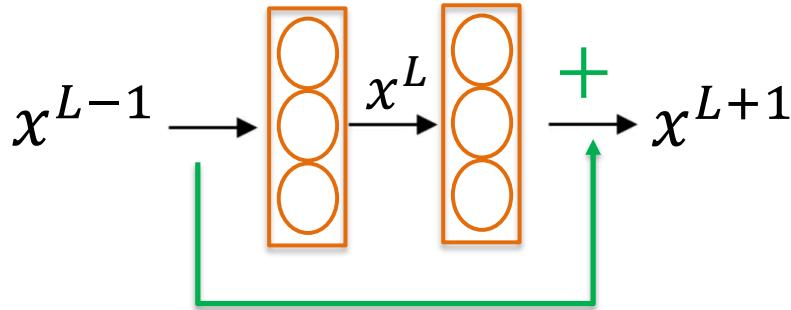


Figure X.4: Skip connection: Usually use a same convolution since we need same dimensions (Otherwise we need to convert the dimensions with a matrix of learned weights or zero padding)

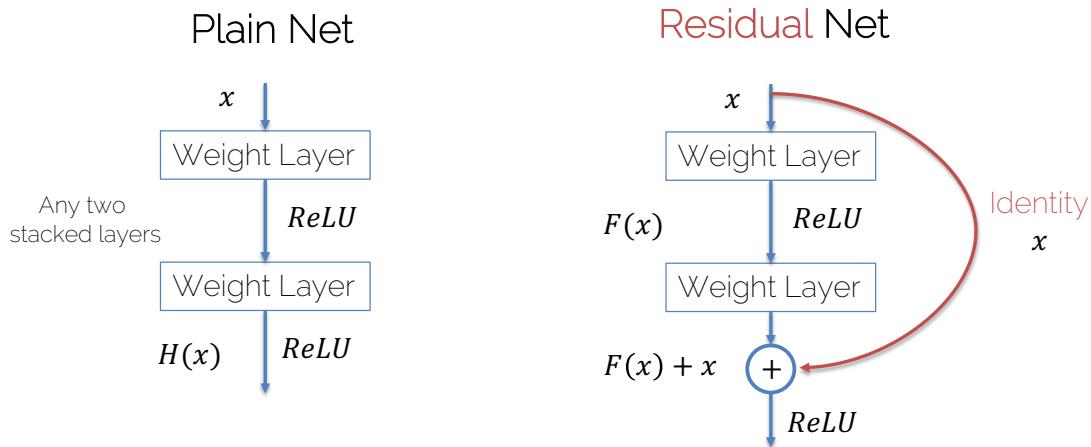


Figure X.5: Residual Net

We wouldn't call the skip connection itself as nonlinear then, because its not introducing new non-linearity.

## X.5 ResNet (Residual Networks)

- Xavier/2 initialization
- SGD + Momentum(0.9)
- lr=0.1 (lr/10 when plateau)
- Minibatch: 256
- Weight decay: 1e-5
- No dropout
- 60m parameters

- If we go deeper, the performance starts to degrade at some point. (Deeper ResNets perform better than not so deep ResNets, not the case with PlainNets)

Why do ResNets (skip connection) work?

- The same values are kept and a non-linearity is added (good if previous values are 0)
- The identity is easy for the residual block to learn
- Guaranteed it will not hurt the performance, can only improve

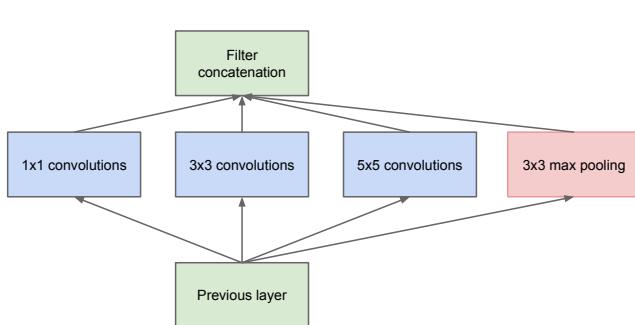
## X.6 Inception Layer: 1x1 Convolution

1x1 Convolution: No padding needed, same output size. Just scales the input. (Same as having a 3 neuron fully connected layer)

Use more convolutional layers: Use it to shrink the number of channels and adds a non-linearity → one can learn more complex functions

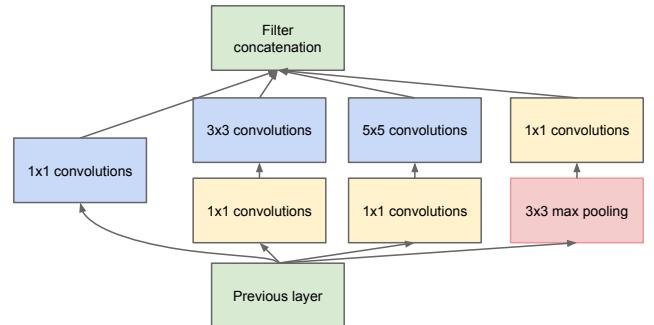
Inception layer:

- You can use all filter sizes instead of one at the same time → Extreme computational
- Implementation: All same convolutions → output spatial size is the same. For 3x3 max pooling stride of 1. Then concatenate at the end. This is not sustainable.
- That's why we use 1x1 convolutions. First do 1x1 convolution then the filter. It will reduce the number of multiplications needed
- Computational cost: Reduction of multiplications by 1/10 with 1x1 convolution
- Using the Inception Layer creates the GoogLeNet



(a) Inception module, naïve version

Figure X.6: Even a modest number of 5x5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters.



(b) Inception module with dimensionality reduction

Figure X.7: 1x1 convolutions are used to compute reductions before the expensive 3x3 and 5x5 convolutions. No blow-up in computational complexity.

## X.7 XceptionNet

- „Extreme version of Inception“: applying (modified) Depthwise Separable Convolutions instead of normal convolutions

- 36 conv layers, structured into several modules with skip connections
- Outperforms Inception Net V3
- Normal convolutions act on all channels

**Depth-wise separable convolutions (DSC):** Filters are applied only at certain depths of the features. Normal convolutions have groups set to 1 (act on all depths), the convolutions used in this image have groups set to 3

```

1 class torch.nn.Conv2d(in_channels, out_channels, kernel_size,
2   stride=1, padding=0, groups=3)
3 class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
4   stride=1, padding=0, groups=3)

```

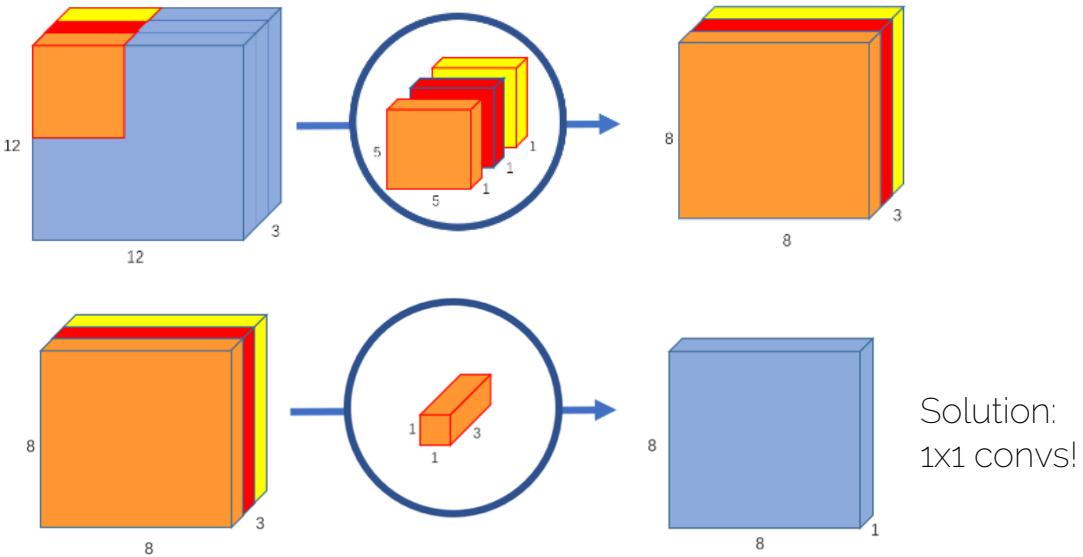


Figure X.8: Above: The depth size will always be the same.

Below: First DSC, then 1x1x3 convolution filter to change the output depth to 1

**Why?** DSC plus 1x1 Convolutions reduces the number of computations

## X.8 Fully Convolutional Network

Convert fully connected layers to convolutional layers! (1x1 convolutions = FC layers) → Has a size of H/32 x W/32 instead of 1x1

Want output of the same size as input image → go back to original size with upsampling

### Types of Upsampling

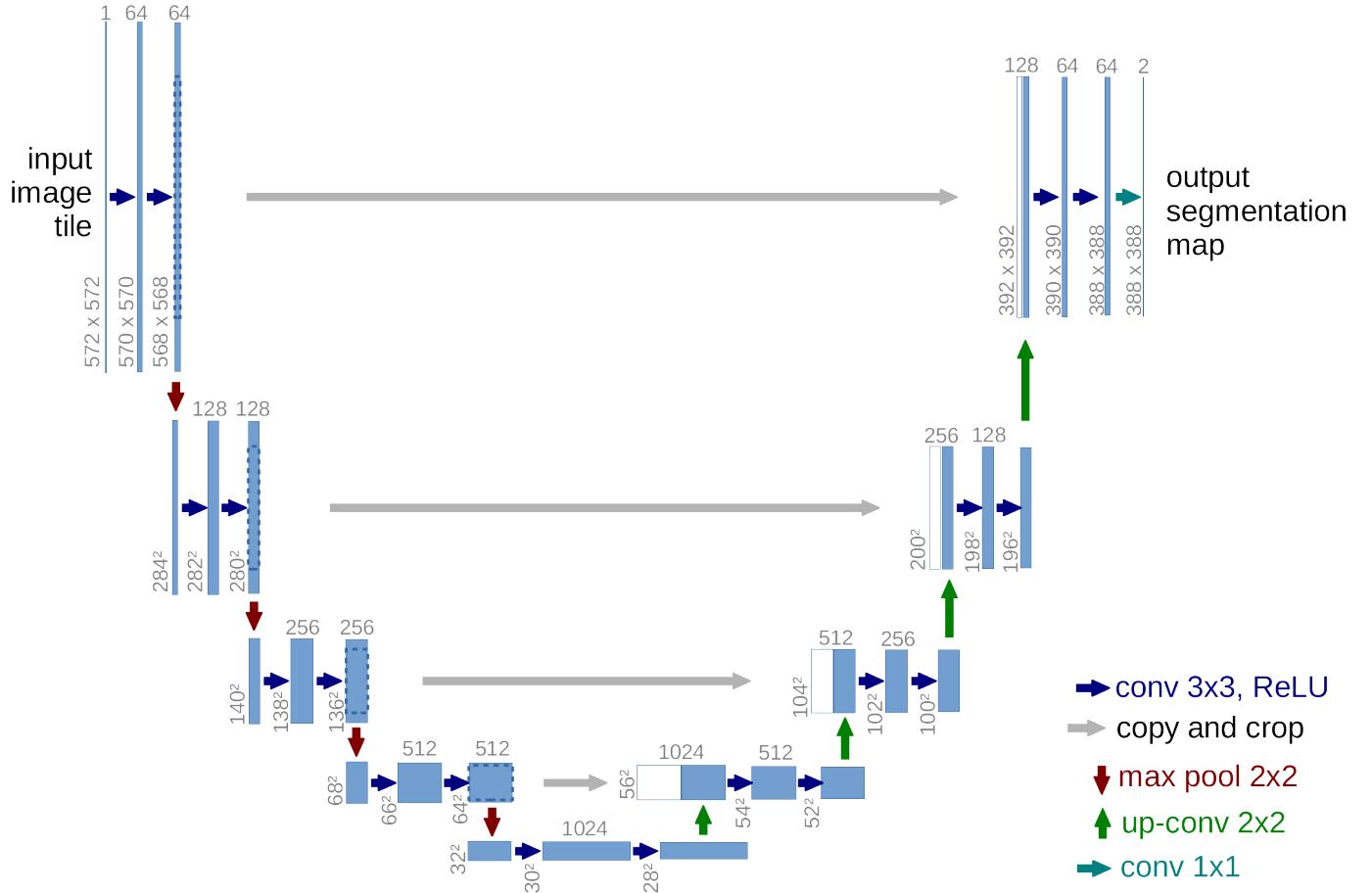
1. Interpolation: Few artifacts

- Nearest neighbor interpolation: average of the pixels intensity around the pixel (with padding).
- Bilinear interpolation: weighted average - give the surrounding pixels weights, and then sum. Could be of different kernel sizes. Results with a bigger, but blurry new image.
- Bicubic interpolation: accomplished using either Lagrange polynomials, cubic splines, or cubic convolution algorithm

2. Transposed Convolution (Up-convolution): Unpooling (Blow image up: spread pixels and fill values with 0) + Convolution → efficient (If one does a cascade of unpooling + conv operations, we get to the encoder-decoder architecture)

**U-Net:** Even more refined: Autoencoders with skip connections (aka U-Net)

- Fully convolutional networks, that utilizes skip connections between the encoder and decoder units, for fine-grained details, boosting performance.
- Usually used for **segmentation** of the image to its objects, by classifying each single pixel among a veracity of classes.
- For every layer of the decoder part we concatatae the output of the corresponding layer from the decoder (Which means that both encoder and decoder parts of the module have the same number of levels). Thus, we use the learnt features from the encoder as an input, a guidline of a sort, to the decoder on it's way to the target.
- **Left side:** Contraction Path (Encoder)
  - Captures context of the image
  - Follows typical architecture of a CNN: Repeated application of 2 unpadded 3x3 convolutions, Each followed by ReLU activation, 2x2 maxpooling operation with stride 2 for downsampling – At each downsampling step, # of channels is doubled
  - As before: Height and Width go down, Depth goes up
- **Right side:** Expansion Path (Decoder)
  - Upsampling to recover spatial locations for assigning class labels to each pixel: 2x2up-convolution that halves number of input channels, Skip Connections: outputs of up-convolutions are concatenated with feature maps from encoder, Followed by 2 ordinary 3x3 convs, final layer :1x1 conv to map 64 channels to # classes
  - As before: Heigh and Width go up, Depth goes down
- **Applications:**
  - Labeling pixels in the original images - useful for solving segmentation problems.
- **Note:** On the upscaling-stage we concatenate the corresponding layers from the down-scaling stage to the the new created layers by the **up-conv** layers. Then they are processed together.



# XI Recurrent Neural Networks

## XI.1 Transfer Learning

Training can be difficult with limited data and other resources (laborious task to manually annotate) → use pretrained models and add your own classifier on top of it

Two Distributions:

- Distribution P1: Large dataset
- Distribution P2: Small dataset

Slightly different task, but use what has been learned for another setting.

Only change and the last decision class (fully connected layer). Reuse all the previous layers. Alternative: If the own dataset is big enough train more layers with a low learning rate.

**When does it make sense?** It makes sense when task T1 and T2 have the same input (e.g. an RGB image), when there's more data for T1 than for T2 or when the low level features for T1 can be useful to learn T2.

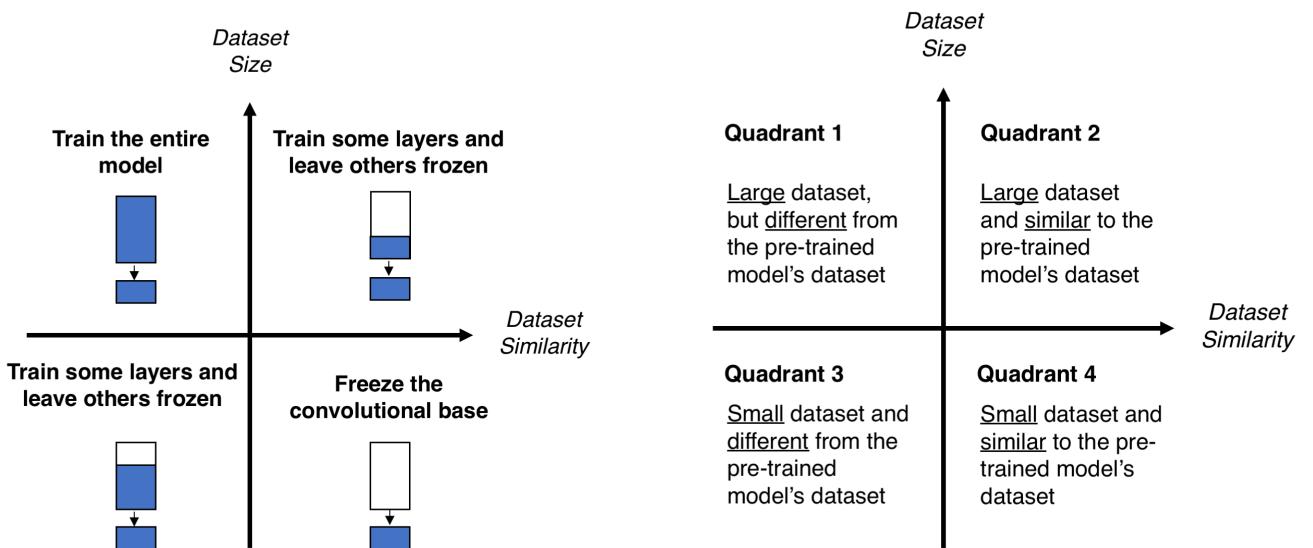


Figure XI.1: Size-similarity matrix and decision map for fine tuning pre-trained models

## XI.2 Recurrent Neural Networks

RNN processes sequence data. The input and/or output can be sequences, whose components are related. They are flexible, can be one-one, one-many (Image captioning: Sentence for a single image), many-one (Language recognition), many-many (Machine translation (time shift), event classification (check if frames

show a dog)).

⇒ Learning process is not independent: remember things from processing training data, and remember things learnt from prior inputs, prior inputs influence decision

### Basic Structure

Multi-layer RNN: Inputs, Hidden States, Outputs

Hidden states will have its own internal dynamics → more expressive model

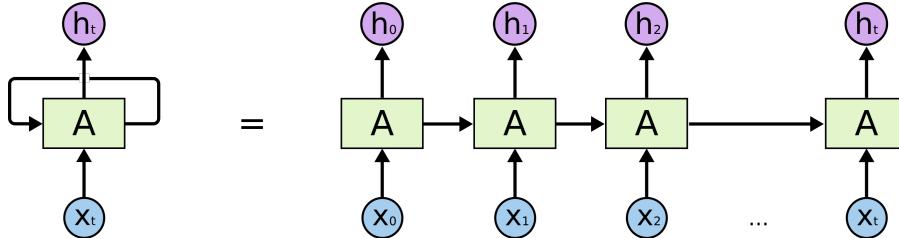


Figure XI.2: Basic structure of an RNN: From the current state, the output  $h$  is predicted.  $x_1, x_2, \dots$  are processed with the same parameters. Output  $h_2$  depends on  $x_2, x_1$  and  $x_0$

We want to have notion of time or sequence. Take current input, weight it and compute the next state:  $A_t = \theta_C A_{t-1} + \theta_x x_t$  where the former part is the previous hidden state and the latter is the input. The  $\theta$ s are parameters to be learned. The following formulas are used (ignoring activation functions):

$$A_t = \theta_C A_{t-1} + \theta_x x_t$$

$$h_t = \theta_h A_t$$

The parameters  $\theta$  are the same for each time step. The connection between  $x^t$  and  $x^{t+1}$  is the same as between  $x^{t+1}$  and  $x^{t+2}$  (same weights). Backpropagation with chain rule all the way back  $t=0$ . Add the derivatives at different times for each weight. Not so straightforward as in other networks because weight appears more often.

**Problem:** Long-term dependencies (e.g. in sentences) → Important to keep enough information  
Ignoring the input, same weight are multiplied over and over again:  $A_t = \theta_c^t A_0$

- small weights → vanishing gradients
- large weights → exploding gradients

$\theta$  is a matrix, we can write it by its eigendecomposition:

$$\theta = Q \Lambda Q^T \xrightarrow{\text{Orthogonal theta}} A_t = Q \Lambda^t Q^T A_0$$

- eigenvalues with magnitude  $< 1$  → vanishing gradients
- eigenvalues with magnitude  $> 1$  → exploding gradients (Gradient clipping: cut gradients larger than a threshold)

So in order to fix the exploding and vanishing gradients caused by  $\theta$ , we enforce that the matrix has eigenvalues = 1. (Allows the cell to maintain its state).

But the vanishing gradients could also come from the activation function being used, tanh typically in this case.

## XI.3 Long Short Term Memory

LSTMs try to solve those problems once and for all by setting the eigenvalues to 1 and ensuring that tanh does not cause vanishing gradients. Basically, they are an iteration of a simple RNN with tanh as non-linearity. It's a special kind of RNN capable of learning long-term dependencies.

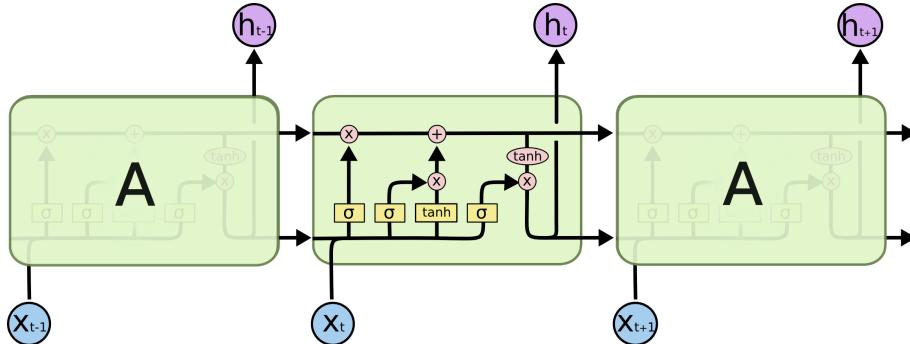


Figure XI.3: LSTM

Key ingredients:

- The cell transports the information through the unit (highway for the information) Add information through multiplication and sums. Just like the skip connections in the ResNet, this ensures that there is always a path we can take (i.e. longer sequence cannot cause problems).
- The gate removes or adds information to the cell state.
  - Forget gate:  $f_t = \sigma(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$ . Decides when to erase the cell state.  
Sigmoid = output between 0 (forget) and 1 (keep)
  - Input gate:  $i_t = \sigma(\theta_{xi}x_t + \theta_{hi}h_{t-1} + b_i)$ . Decides which values will be updated.  
New cell state from  $\tanh(-1, 1)$ .
  - Interaction between current and previous time step: element-wise operations:  $C_t = f_t \odot C_{t-1} + i_t \odot g_t$
  - Output gate:  $o_t = \tanh(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o)$  Decides which values will be outputted. Output from a  $\tanh(-1, 1)$
- Actual output:  $h_t = o_t \odot \tanh(C_t)$
- Cell update:  $g_t = \tanh(\theta_{xg}x_t + \theta_{hg}h_{t-1} + b_g)$

LSTM solved the vanishing gradient problem because...

...  $f_t$  are outputs of a sigmoid and therefore 1 for important information.

... because activation functions act through a summation, therefore vanishing gradients are not propagated through the whole cell state.

**Dimensions:** When coding an LSTM, we have to define the size of the hidden state. Dimensions need to match, but are not limited to vectors.

**General LSTM Units:** Input, states, and gates are not limited to 1st-order tensors. Gate functions can consist of FC and CNN layers

E.g.: ConvLSTM for video sequences: Input, hidden, and cell states are higher order tensors (i.e. images). Gates have CNN instead of FC layers

RNNs in Computer Vision: Caption generation and Instance segmentation (separate different instances of a class)

# XII Advanced Deep Learning Topics

## XII.1 Attention

Same goal as LSTM

Attention is the intuition between the hidden state and the output. We want to give the unit a context, and what it does is that it gives the current time stamps and all the other one a relationship. We have  $\alpha$  which are variables (attention variables) which are weights that represent how important the time stamp  $t+1$  is to predict the output at  $t+1$ . All these variables are processed together to what it's called a context.

A decoder processes the information

Decoders take as input: Previous decoder hidden state, Previous output, Attention

**Transformers:** it gets rid of the recurrent architecture (memory problems of RNNs disappear). There are direct connections. It's not RNN or CNN. They just use attention.

Difference: in RNN the words are processed sequentially; transformers have connections all over the place and the alphas say how strongly the words are connected to each other.

Transformers have a lot of parameters too and are very related to Graph Neural Networks.

## XII.2 Graph Neural Networks

Node: a concept (e.g. word, image)

Edge: a connection between concepts

Generalizations of neural networks that can operate on graph-structured domains

Key challenges: The number of nodes and edges changes as we move from one graph to the other. We need invariance to node permutations.

There can be node and edge feature vectors, which go to a series of hidden layers where the propagation of information takes place. Propagate information over several iterations. The operation here is not a convolution. At the end the feature vectors have been changed. It represents the nodes but also the neighbors  $\Rightarrow$  graph with updated context-aware node and (possibly) edge feature vectors

### Message passing networks

We can divide the propagation process in two steps: 'node to edge' and 'edge to node' updates (alternating). See [XII.1](#).

### Multi-object tracking with graphs

Step 1: Object detection. Each node can represent a detection and the edges can represent the trajectory.  
Step 2: Use the graph to perform learning. Not solve the problem directly, but perform neural message passing steps (sharing of information between node and edges). Classify edges: feature vector as input and decides if active or not. Extract trajectories.

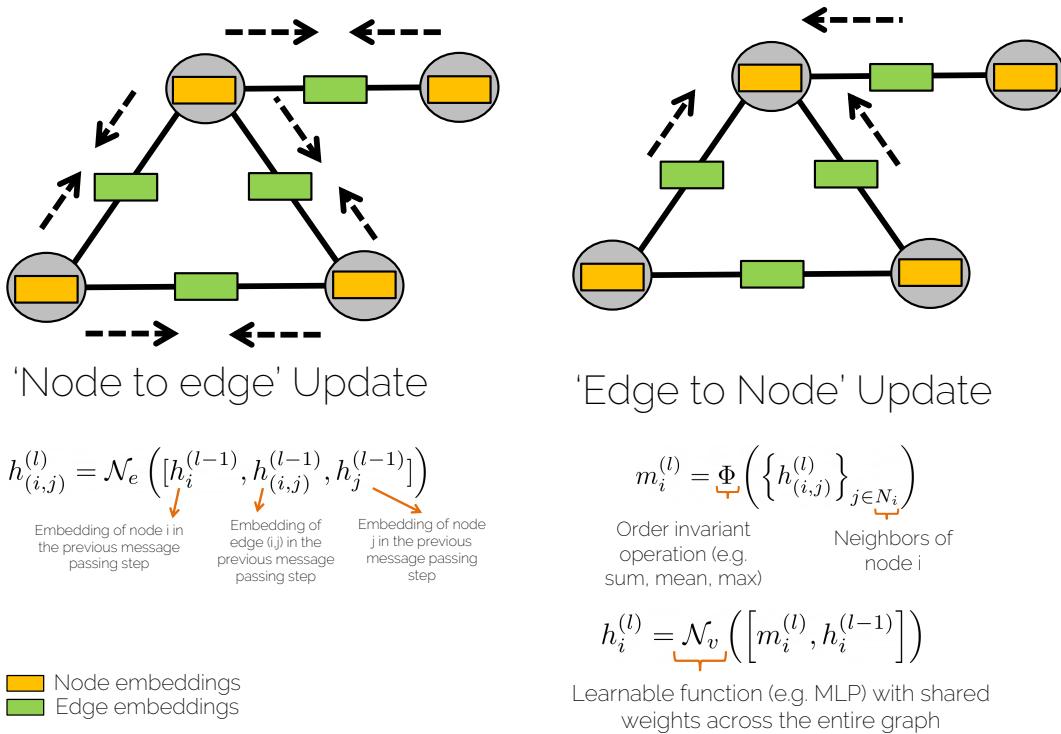


Figure XII.1: Node to edge: At every message passing step  $l$ ,  $\mathcal{N}_e$  is learnable (small) neural net with shared weight across the entire graph. Every edge is updated by applying the neural net to the old value, and the value of the two nodes of the edge.

Edge to node: This update is a bit trickier, as we don't know the number of edges a node has (compared to the previous case where an edge always has two nodes). To solve this, we need an order invariant function  $\Phi$  that aggregates all edge values of a node to a single value. Then we again have a shared neural net that process this single combined edge information and the current node value to produce the new node value. The aggregation provides each node embedding with contextual information about its neighbors.

Repeating those steps allows the the nodes to communicate over the edges and the neural networks decide how important an information is.

## XII.3 Generative Models

Outputs of the NN with same width and height as input (e.g. Semantic Segmentation (FCN) with upsampling) but there are better ways as the SegNet (Convolutional Encoder-Decoder)

Given training data, how to generate new samples from the same distribution: Various methods

## XII.4 Variational Autoencoder

Autoencoder: Encoding the input into a representation (bottleneck) and reconstruct it with the decoder  $\Rightarrow$  Reconstruct input, Unsupervised learning

Variational Autoencoder:

- Goal: Sample from the latent distribution to generate the new outputs
- Training: The loss makes sure the latent space is close to a unit Gaussian and the reconstructed output is close to the input
- Test: Sample from the latent space

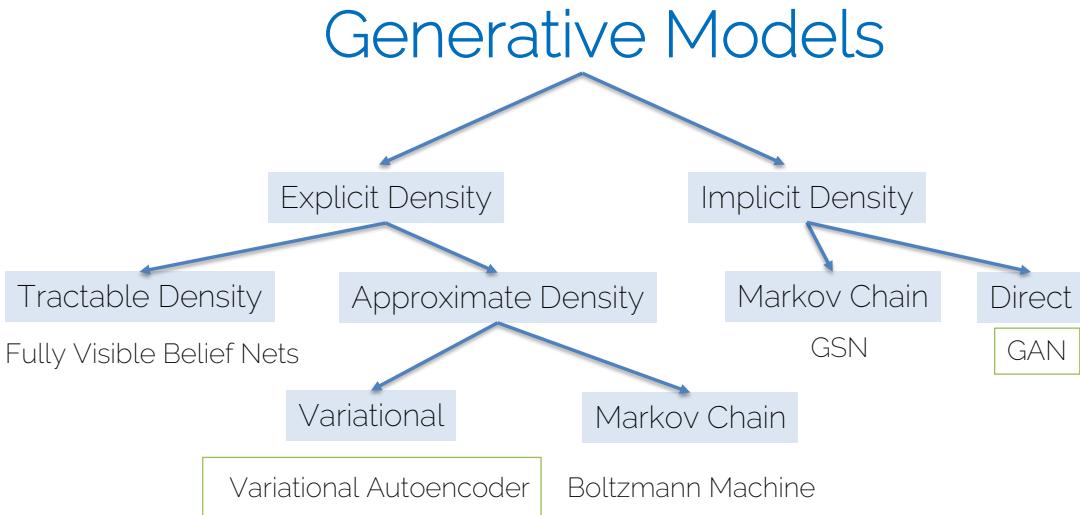


Figure XII.2: Explicit: Model explicitly the density. Plug in the model density function to likelihood. Then maximize the likelihood

Implicit: No explicit probability density function (pdf) needed. Instead, a sampling mechanism to draw samples from the pdf without knowing the pdf

- $\Rightarrow$  Probability distribution in latent space (e.g., Gaussian), Interpretable latent space (head pose, smile), Sample from model to generate output

## XII.5 Generative Adversarial Networks (GAN)

Huge increase in the usage

Reconstruction with Loss (often L2) gives blurry output because distributes error equally, mean is the optimum.  $\rightarrow$  Learn the loss function

We have generators G and discriminators D (these tell you if the images are real or fake). G and D are neural networks so they improve each other with backpropagation. Train with real world images. Generator is trained to better generate images and discriminator is trained to detect real and fake images:  $G \Leftrightarrow D$

Real data: Sample from the data, pass it through D (tries to be near 1)

Fake data: Input noise passes through G which generates an image. D tries to identify the image  $\rightarrow$  D tries to make  $D(G(z))$  near 0, G tries to make  $D(G(z))$  near 1.

Discriminator Loss:  $J^{(D)} = -\frac{1}{2}E_x p_{data} \log D(x) - \frac{1}{2}E_z \log(1 - D(G(z)))$  (binary cross entropy)

Generator Loss:  $J^{(G)} = -J^{(D)}$

Minimax Game: G minimizes probability that D is correct. Equilibrium is saddle point of discriminator loss (D provides supervision (i.e., gradients) for G)

### GAN Applications

- Generate high quality (definition) images
- Generate new faces from scratch
- Conversions: a horse to a zebra (zebrify), day to night
- GAN based image editing: From drawing to real photo, change faces in videos

## XII.6 Reinforcement Learning

Learning by interacting with the environment. Basically we get the labels from the environment.

- Environment sends an Observation  $o_t$  to the Agent
- The agent makes decisions and performs an action  $a_t$  based on the history  $h$  of observations, actions and rewards up to time-step  $t$
- Based on the action, Agent gets a reward  $r_t$  (positive or negative)

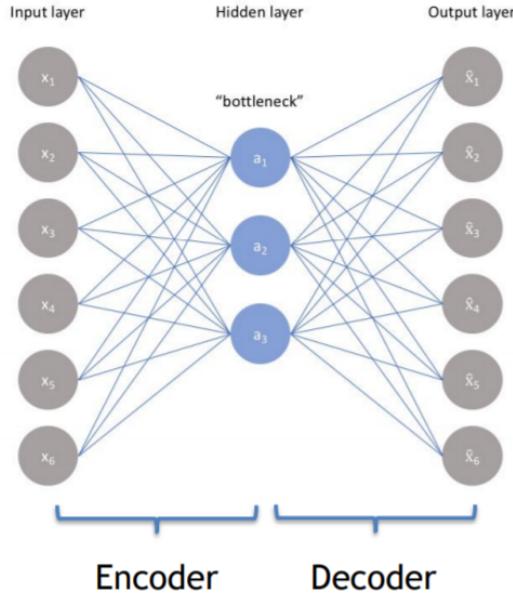
**Characteristics:** Sequential, non i.i.d. data (time matters), Actions have an effect on the environment → Change future input. No supervisor, target is approximated by the reward signal.

**Problem:** Your actions are changing the future, so you need to take into account and keep in a state all history. Therefore there needs to be a mapping between the history and the state  $s$ . But this is really hard to maintain. History grows linearly over time. So use a Markov assumption. A state  $t$  is Markov if it only depends on the past state. Reward and next state are functions of current observation  $o_t$  and action at only

**Components of an RL agent:** Policy (Behavior of the agent → Mapping from state to action) and value, q-function (How good is a state or (state, action) pair → Expected future reward)

## XII.7 Autoencoder

One of the most interesting network architectures. An autoencoder is such a powerful tool, that later on was perfected, and it is being used intensively in the industry and research, for the tasks of segmentation, 3D reconstruction, generative models, style transfers, etc. Basically, endless applications.



1. The basic architecture is still based on fully connected layers (Of course with activations layers).
2. It consists of two main parts:
  - Encoder: Reducing the dimensionality towards the "latent" space forces the network to focus on collecting the most meaningful features from the input data, so the loss of information would be as small as possible. As stated in the tutorial session, in similar notion to PCA. Therefore, we could say that the Encoder is used as a \*\*features extractor\*\*, much alike the upcoming convolutional layers (But yet, not as strong, and still much more expansive).
  - Decoder: Receives the dimensionality-reduced latent space, and aims to \*\*reconstruct\*\* the input of the Encoder. The output has the same shape as the input. A crucial note, for more advanced usage, of semantic segmentation.
3. For the loss function, we could use the  $L_1$  loss or the MSE, between each pixel in the input and its corresponding pixel in the output.
4. Latent space:
  - If the size of is too small, not much information could eventually pass through to the decoder, and the reconstruction would be very hard. The result would be very blurry.
  - If too big, the network could basically learn to copy the image, without learning any meaningful features.
5. Without non-linearities, it is very similar to PCA.

But why do we even want to use that? Auto encoders, as used in exercise 08, is an excellent solution to a state where our dataset is very big, but only just a small part of it is actually labeled, like medical a CT dataset, for example. So, we will have 2 steps:

## 1. Autoencoder

 $\rightarrow$ 

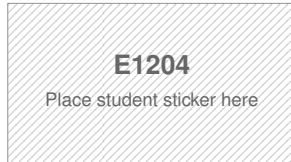
reconstruct the input. Let the Encoder learn the relevant features about the **unlabeled** data. This part can be referred to as **unsupervised learning**.

2. After the training has converged, remove the decoder and discard it. Then, plug in instead, just after the latent-space, a very simple fully-connected classifier, and train on the **labeled** data, given the fact that the remaining Encoder is already trained as a good features extractor. This part can be referred to as **supervised learning**.



# Introduction to Deep Learning endterm 2022SS exam

Introduction to Deep Learning (Technische Universität München)

**Note:**

- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

## Introduction to Deep Learning

**Exam:** IN2346 / endterm

**Examiner:** Prof. Leal-Taixé

**Date:** Monday 1<sup>st</sup> August, 2022

**Time:** 08:15 – 09:45

P 1	P 2	P 3	P 4	P 5	P 6	P 7	P 8	P 9

### Working instructions

- This exam consists of **24 pages** with a total of **9 problems**.  
Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 90 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources:
  - one **non-programmable pocket calculator**
  - one **analog dictionary** English ↔ native language
- Subproblems marked by \* can be solved without results of previous subproblems.
- **Answers are only accepted if the solution approach is documented.** Give a reason for each answer unless explicitly stated otherwise in the respective subproblem.
- Do not write with red or green colors nor use pencils.
- Physically turn off all electronic devices, put them into your bag and close the bag.

Left room from \_\_\_\_\_ to \_\_\_\_\_ / Early submission at \_\_\_\_\_

## Problem 1 Multiple Choice (18 credits)

Mark correct answers with a cross



To undo a cross, completely fill out the answer option



To re-mark an option, use a human-readable marking



Please note:

- For all multiple choice questions any number of answers, i.e. either zero (!), one or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

1.1 You're training a neural network with 3 fully-connected layers, ReLU as an activation function and a keeping dropout probability of 0.2. Your task is to classify a dataset that is made of 2 classes - what is a good choice of (activation layer, loss function) at the end of the architecture?

- (TanH, Binary Cross Entropy)
- (ReLU, Mean Squared Error ( $L_2$ ))
- (Sigmoid, Binary Cross Entropy)
- (Softmax, (Multi-class) Cross Entropy)

1.2 What is true about fully-connected layers?

- They are non-linear functions.
- They can be represented as a convolutional layer.
- Given the input  $X_{N \times D}$  and  $W_{D \times M}$ , the length of the learnable parameters  $\beta$  and  $\gamma$  in a following batch normalization layer is  $D$ .
- Once initialized, they can accept any input size.

1.3 If your input batch of images is  $16 \times 32 \times 64 \times 64$  ( $N \times C \times H \times W$ ), how many parameters are there in a single  $1 \times 1$  convolution filter operating on this input, including bias?

- 65
- 2,097,153
- 33
- 17

1.4 You're building a neural network consisting of convolutional layers and TanH as the activation function. What could mitigate / reduce the likelihood for the gradient to vanish during training?

- Use Xavier initialization for your convolutional layers.
- Organize the layers in residual blocks.
- Replace TanH with Leaky ReLU with  $\alpha = 0.2$ .
- Reduce the number of layers.

1.5 What is the benefit of using Momentum in optimization?

- It introduces just a single learnable parameter.
- It effectively scales the learning rate to act the same amount across all dimensions.
- It combines the benefits of multiple optimization methods.
- It is more likely to avoid local minima when used with stochastic gradient descent.

1.6 A sigmoid layer ...

- ... could be used only on the logits of a classification neural network.
- ... could boost performance when used in linear regression on normalized data with mean=0 and standard deviation=1.
- ... maps all values into the interval  $[-0.5, 0.5]$ .
- ... is a zero-centered non-linear activation function.

1.7 Given a Max-Pool layer with kernel size of  $2 \times 2$  on a window with all unique positive values:

- In order to backpropagate through the layer, one needs to pass information about the positions of the max values from the forward pass.
- 75% of the derivatives differ from zero.
- The layer's weights are updated with a chosen optimizer method.
- It performs features selection.

1.8 You are given the following fully-connected network. What is true about the Xavier initialization? **Note:** Each fully-connected layer has  $X$  as the input,  $W$  as the weight matrix and a vector  $b$  as the bias.

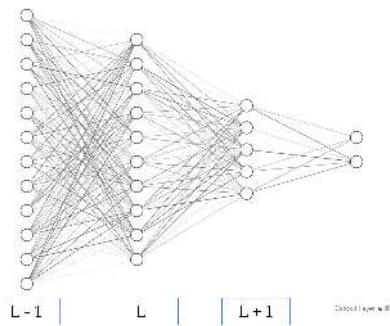


Figure 1.1: A fully-connected network.

- $\text{Var}(X_L) = \text{Var}(X_{L+1})$
- $\text{Var}(W_{L-1}) = \text{Var}(W_L)$
- $\text{Var}(X_L) = \text{Var}(W_L)$
- $\text{Var}(W_L) = \frac{1}{m}$ , where  $m$  is the number of columns of  $W_L$ .

1.9 You're given the following fully-connected toy network with:

1. All weights are initialized with the value 1.
2. The optimizer is stochastic gradient descent (SGD).
3. The learning rate is  $\alpha = 1$ .
4. The same input  $X = [1, 1, 1, 1]$  is fed time after time, for 3 iterations.

What is the sign of the weights of the first layer,  $L_1$ , by the end of the loop?

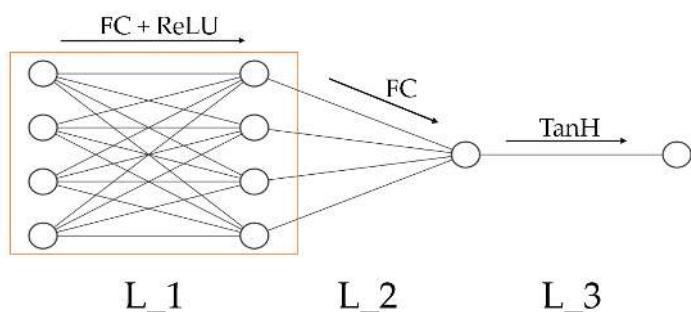


Figure 1.2: Toy network

- All negative.
- All zeros.
- All positive.
- Both negative and positive.

## Problem 2 Unsorted Questions (14 credits)

2.1 List one advantage and one disadvantage of having a small batch size for training.

(1p): Advantages: Less memory consumption of a batch. Faster computation for one iteration/for updates/ for a batch. Escape local minima.

(1p): Disadvantage: Converges slower. Small batches can offer a regularizing effect that provides better generalization (Noise). More iterations needed.

0  
1  
2

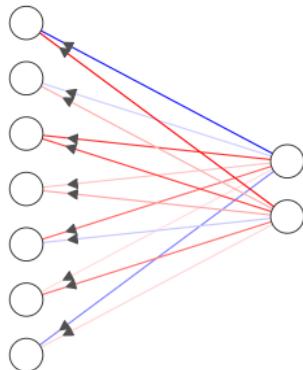


Figure 2.1: Backpropagation of a fully-connected layer

Given a fully-connected layer  $Y = XW + b$ , where  $X_{N \times D}$ ,  $W_{D \times M}$ , an upstream gradient  $dout = \frac{\partial L}{\partial Y}$  during the backpropagation process and a loss function  $L : \mathbb{R}^M \rightarrow \mathbb{R}$ .

2.2 What are the dimensions of  $b$  and  $dout$ ?

(0.5p):  $1 \times M$

(0.5p):  $N \times M$

0  
1

2.3 What is the derivative of  $\frac{\partial L}{\partial X}$  and what are its dimensions?

(0.5p):  $dout \cdot W^T$

(0.5p):  $N \times D$

0  
1

2.4 Consider the  $L_2$  - regularization mechanism with a corresponding coefficient  $\lambda$ . Write down the update rule of gradient descent with weight decay, for the iteration  $k + 1$ , a weight matrix  $\theta$  and a learning rate coefficient  $\alpha$ . Use  $\nabla L(\theta_k)$  as the derivative of  $\frac{\partial L}{\partial \theta}$ .

0  
1

The  $L_2$  regularization term, that is added to the scores of a linear layer is  $Y = f(x, \theta, b) + \frac{1}{2} \sum_i \sum_j |\theta_{i,j}|_2^2$ . So after taking the derivatives, the optimizer step is:

(1p):  $\theta_{k+1} = (1 - \alpha\lambda)\theta_k - \alpha\nabla L(\theta_k) = \theta_k - \alpha(\nabla L(\theta_k) + \lambda\theta_k)$

0  2.5 What is the purpose of weight decay?

1 (1p): Regularization.

0  2.6 What can we expect to see in terms of the error curves in a graph, when the corresponding coefficient  $\lambda$  in weight decay is too high while training?

1 (1p): Underfitting. Both training and validation loss will plateau on a relatively high value.

Linear Regression is a well-known problem in Machine Learning, where we aim to find the best coefficient matrix  $W$ , given a function  $\hat{Y} = f(X) \sim XW$ , where  $X = \{x_1, \dots, x_n\}$  and such that

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{2n} \sum_{i=1}^n (y_t - \hat{y}_i)^2$$

Consider that we have a classical problem setting with a low amount of data samples.

0  2.7 If our algorithm converges to a minimum, why is it guaranteed to be the global minimum?

1 (1p): Linear regression is a convex function.

0  2.8 What advantage does the linear regression method has over the iterative deep-learning methods, given a linearly distributed dataset?

1 (1p): It has a closed-form solution. Mathematically computing the solution  $W = (X^T X)^{-1} X^T Y$  is superior.

0  2.9 Mention 2 drawbacks of ReLU. Explain how they affect the optimization problem.

1 (0.5p): Not zero centered/ All weights can change only with the same sign  $\rightarrow$  (0.5p) The zigzag effect, slower training.

(1p): Dying ReLU  $\rightarrow$  Vanishing gradient.

(1p): ReLU not differentiable at zero.

You're working on a group project, and define some deep-learning architecture. You've found a good set of hyperparameters and you are ready to start your training. Now, your partner suggests that you could merge the validation set into the training set, so it will be bigger.

0  2.10 What is the purpose of the validation set while training?

1 (1p): Sanity check. The validation set is unseen data, that helps to determine whether the model has overfitted or underfitted.

2.11 You take their advice, and see that your training loss is converging at a very low error rate for a long time. However, when testing, your error rate is really high. Name the problem, and how can we avoid training for such a long time **by using the validation set**.

0  
1  
2

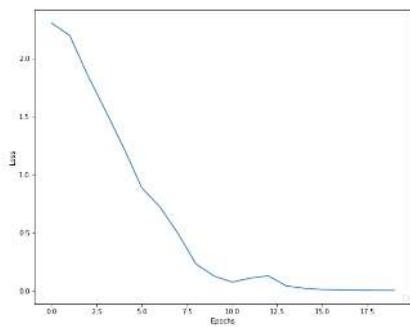


Figure 2.2: Low training error rate

(1p): Overfitting. (1p): Early stopping.

Sample Solution

### Problem 3 Data Augmentation (4 credits)

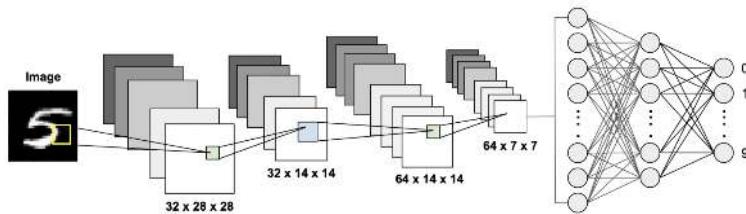


Figure 3.1: MNIST classification architecture

You have a classification task on the MNIST dataset. After training, your model achieves great results. In order to further improve, a friend has suggested you to apply data augmentation.

0  3.1 Explain the concept of data augmentation. (When would one use it? What is crucial to keep?)

1   
Data augmentation is a regularization technique in which we modify the training set inputs to introduce a larger variety of training data.  
(0.5p): One uses data augmentation for: Regularize the training/minimize the generalization gap/reduce overfitting/one use it when the dataset is too small.  
(0.5p): One should keep in mind that augmented data must come from the same distribution as original data.

0  3.2 On which dataset out of the split training, validation, test would you apply data augmentation?

1   
Training.

You remember from class that data augmentation improves performance. In order to optimize your model, you decide to apply "Horizontal Flip" and "Gaussian Blur". Surprisingly, you notice that training and testing errors have both gone higher.

0  3.3 Why is the training error higher?

1   
(1p): Training error is higher because: There are now instances that present cases not seen before in original training dataset / data augmentation increases the diversity of training data / more variance in train data.

0  3.4 Why is the test error higher?

1   
(1p): The augmented data we create (horizontal flipped one) do not belong to the MNIST dataset distribution, hence do not come from same distribution as test data.

## Problem 4 An application (16 credits)



Figure 4.1: Example images of the MNIST dataset.

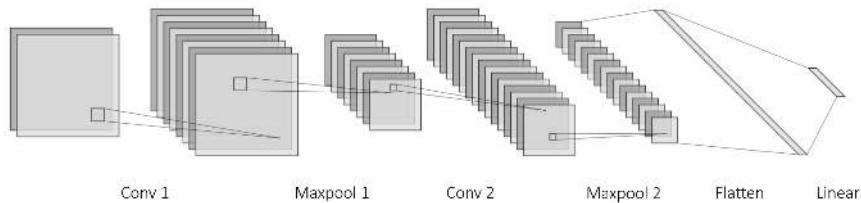


Figure 4.2: Sketch of the given architecture

We train a network which calculates **the sum of two digits** as a classification problem, represented by images from the MNIST dataset.

The MNIST dataset comprises 50,000 unique  $1 \times 28 \times 28$  grayscale images of handwritten digits, in the range of [0...9], equally distributed.

### Notes:

- The input of our network are two images stacked in feature space, with channels=2, height=28 and width=28.
- The output of the network is a vector of length  $n$ , corresponding to the number of possible sums.
- Note that if the original MNIST contains a balanced dataset of 50,000 images and 10 labels, now it holds  $50,000 \cdot 50,000 = 2.5\text{B}$  different possible pairs.
- Convolutional layers are defined as  $\text{Conv2d}(<\text{input channels}>, <\text{output channels}>)$
- Maxpool layers are defined as  $\text{MaxPool2d}(<\text{input channels}>)$

We give the following network architecture that classifies into  $n$  possible output classes:

- $\text{Conv2d}(2, 8) \rightarrow \text{MaxPool2d}(2) \rightarrow \text{BatchNorm2d}() \rightarrow \text{ReLU}()$
- $\text{Conv2d}(8, 16) \rightarrow \text{MaxPool2d}(2) \rightarrow \text{BatchNorm2d}() \rightarrow \text{ReLU}()$
- $\text{Flatten}()$
- $\text{Linear}(k, n)$

where  $k \in \mathbb{N}$  is a variable defining the input dimension of the Linear layer after the Flatten operation.

4.1 Give a triplet of (kernel size, stride, padding) for the convolutional layers, that will keep the spatial dimensions of their input (i.e.,  $H \times W \rightarrow H \times W$ )

(2p): Example: (k=3, s=1, p=1).

0  
1  
2

For the following consider a batch size  $N$  and a triplet (kernel size, stride, padding) that keeps the spatial dimension of the input.

0 4.2 What is the shape of the input of our network? What is the shape of the network output?

1 (0.5p): Shape of input:  $N \times 2 \times 28 \times 28$ .  
(0.5p): Shape of output:  $N \times 19$ , since there are 19 different possible sums.

0 4.3 What is the value of  $k$  (the input dimension of the Linear Layer)? Show your calculation steps. You can provide the final answer using products.

1 (1p): The spatial size decreases:  $28 \rightarrow 28 \rightarrow 14 \rightarrow 14 \rightarrow 7$ . So  $X = 16 \cdot 7 \cdot 7$ .

0 4.4 What loss function should we use?

1 (1p): MCE - (Multiclass) Cross-Entropy.

0 4.5 In the exercises, we used a fully-connected network to classify the MNIST dataset.  
How can we resemble a fully-connected layer, from an input of an image with size  $(2 \times 28 \times 28)$  to an output of 100 neurons, by using convolutions?

2 (2p): Use 100 filters, kernel-size=28, padding=0, stride=0, and flatten the output.

### Receptive Field:

0 4.6 State the definition of the receptive field of a neuron in an intermediate layer of a neural network.

1 (1p): The size of the region in the input image that a single neuron in the intermediate layer is affected by.

0 4.7 What is the receptive field of a neuron after **Maxpool 1**?

We assume

1. Convolutions have a kernel size of 5 and a stride 1.
2. Maxpools have a kernel size of 2 and a stride 2.

For reference the optional formula  $r_k = r_{k-1} + (\prod_{i=1}^{k-1} s_i) \cdot (f_k - 1)$  for  $k \geq 2$   
(1p):  $r_1 = 3 \rightarrow r_2 = r_1 + s_1 \cdot s_2 \cdot (f_2 - 1) = 5 + 1 \cdot (2 - 1) = 6 \rightarrow (6 \times 6)$ .

0 4.8 What is the receptive field of a neuron after the **Linear Layer** using the same assumptions?

1 (1p): A linear layer corresponds to the entire input, i.e. the receptive field is  $(28 \times 28)$  (the whole input image).

4.9 We decided to sample the input images randomly, and not set a fixed split to the dataset. Each time we test our network we randomly sample two digit images. State one problem that can arise by doing so.

(2p): Mixes train and test set. Therefore our network trains on test samples. Thus our test would be biased and we can't check if our model performs well on unseen data.

OR

(2p): Impossible to compare two different runs/architectures (not deterministic).

4.10 After training, you observe 95% test accuracy for the class 12 but 35% test accuracy for class 18. What is the problem?

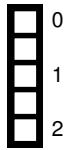
(2p): An imbalanced dataset. There are many more pairs that could sum to 12, than pairs summing to 18.

4.11 A friend suggests approaching the task in a different way, namely as a regression task. Name two required changes to the network.

(1p): 1 output neuron instead of 19, because regression is a continuous function.

(1p): Change the Loss function: MSE or L1.

Sample Solution



## Problem 5 Autoencoder (11 credits)

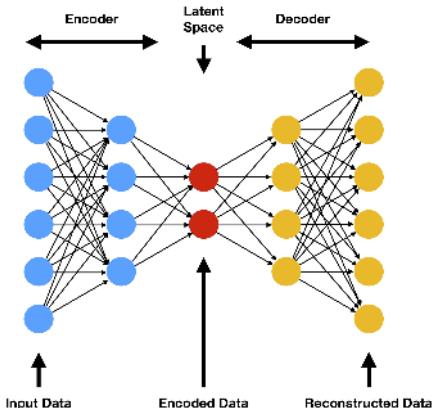


Figure 5.1: A Fully-Connected Autoencoder

0

5.1 The size of the bottleneck (latent space) of an autoencoder is an important hyperparameter. Explain the consequences, if the latent space is too small.

1   
(1p): Too much information is lost which means the signal cannot be reconstructed accurately.

0

5.2 And what if the latent space is too big?

1   
(1p): Model will learn the identity mapping / no compression / no meaningful features are learned.

0

5.3 Explain the main difference regarding data of training an autoencoder in comparison to training a classification network.

1   
(1P): No ground-truth labels necessary / unsupervised vs. supervised.

0

5.4 You train a classification network on a big dataset, but only a small proportion of your dataset is labeled. Explain the steps of using an autoencoder to deal with this issue.

1   
(1P): Train autoencoder on the whole (unlabelled) dataset for reconstruction.  
(1P): Take the encoder model, add classification layers and train/fine-tune it on the labeled data.

0

5.5 How is this technique called? As the classification dataset is quite small, suggest a reasonable approach to handle the weights of the network while training to avoid using the pretrained advantage of using an autoencoder as described above.

1   
(1.5p): (0.5p) Transfer learning. (1p) Freeze the weights / part of the weights of the pretrained encoder.

5.6 You now successfully trained an autoencoder on MNIST. You randomly sample a latent vector and decode it using the trained decoder, but the resulting image does not look like a number at all. Explain why this is to be expected.

0  
 1

(1p): During training we only map the input distribution to the latent space using an encoder. As this function does not need to be surjective, we can't expect randomly sampled vector's to be decoded into MNIST numbers.

5.7 Name a solution how we can train an autoencoder such that we can sample a latent vector and expect our decoder to generate MNIST images for randomly sampled latent vectors.

0  
 1

(1p): Variational Autoencoder / enforce a loss on the latent space so that it follows a known distribution, like a Gaussian.

We saw in the lecture that the Autoencoder architecture could be also used for the task of semantic segmentation.

0  
 1

5.8 What is the semantic segmentation of an image?

(1p): Semantic segmentation is the task of classification for each individual pixel.

0  
 1

5.9 Specify the output dimensions for an input image of size  $C \times H \times W$ .

(0.5p):  $\hat{C} \times H \times W$ , where  $\hat{C}$  is the number of classes in the dataset (in the setting of semantic segmentation).

(0.5p): Also accepted:  $C \times H \times W$ , where  $C$  is the number of input channels (in the setting of an autencoder architecture).

0  
 1

5.10 Why do convolutions fit better to the task of semantic segmentation than fully-connected layers?

(1p): Fully-connected layers process global information from the entire input image, while convolutions extract local features from the neighborhood of each pixel.

## Problem 6 Backpropagation (6 credits)

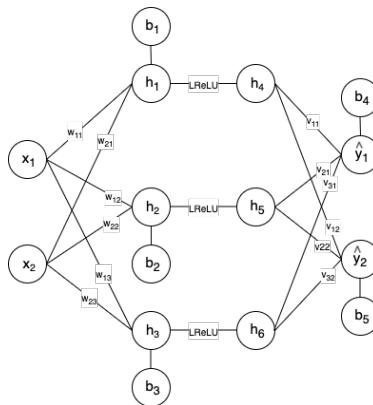


Figure 6.1: Simple network

Variable	$x_1$	$x_2$	$w_{11}$	$w_{12}$	$w_{13}$	$w_{21}$	$w_{22}$	$w_{23}$	$b_1$	$b_2$	$b_3$	$v_{11}$	$v_{12}$	$v_{21}$	$v_{22}$	$v_{31}$	$v_{32}$	$b_4$	$b_5$	$y_1$	$y_2$
Value	1.0	-2.0	-0.5	1.0	2.0	-1.0	0.5	1.5	0.5	0	-1.0	1.0	-0.5	-0.5	2.0	-1.0	1.0	2.0	1.0	1.0	1.0

Table 6.1: Values of the variables

In the diagram, a simple network is given with weights, biases and Leaky ReLU activation with  $\alpha = 0.5$ .

- 0 6.1 Compute the output ( $\hat{y}_1$  and  $\hat{y}_2$ ) of this network. Therefore, you will need to calculate the following variables:  $h_1, h_2, h_3, h_4, h_5, h_6, \hat{y}_1, \hat{y}_2$ .

1  
2

$$\begin{aligned}
 h_1 &= w_{11} * x_1 + w_{21} * x_2 + b_1 = -0.5 * 1 + (-1) * (-2) + 0.5 = 2 \\
 h_2 &= w_{12} * x_1 + w_{22} * x_2 + b_2 = 1 * 1 + 0.5 * (-2) + 0 = 0 \\
 h_3 &= w_{13} * x_1 + w_{23} * x_2 + b_3 = 2 * 1 + 1.5 * (-2) - 1 = -2 \\
 h_4 &= \max(0.5 * h_1, h_1) = \max(0.5 * 2, 2) = 2 \\
 h_5 &= \max(0.5 * h_2, h_2) = \max(0.5 * 0, 0) = 0 \\
 h_6 &= \max(0.5 * h_3, h_3) = \max(0.5 * (-2), -2) = -1 \\
 (1p): \hat{y}_1 &= v_{11} * h_4 + v_{21} * h_5 + v_{31} * h_6 + b_4 = 1 * 2 + (-0.5) * 0 + (-1) * (-1) - 1 = 2 \\
 (1p): \hat{y}_2 &= v_{12} * h_4 + v_{22} * h_5 + v_{32} * h_6 + b_5 = (-0.5) * 2 + 2 * 0 + 1 * (-1) + 2 = 0
 \end{aligned}$$

- 0 6.2 Calculate the Mean Squared Error Loss using your results in the previous question and the target values ( $y_1$  and  $y_2$ ).

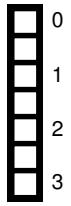
1

$$(1p): \text{MSE} = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^k (\hat{y}_{i,j} - y_{i,j})^2 = \frac{1}{2}(2-1)^2 + \frac{1}{2}(0-1)^2 = 1$$

Note:  $k$  is the number of neurons in the output layer.

6.3 Do one backward pass on this network to update  $w_{11}$  with Stochastic Gradient Descent using learning rate of 0.1.

$$(1.5p): \frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial h_4} \frac{\partial h_4}{\partial h_1} \frac{\partial h_1}{\partial w_{11}} + \frac{\partial L}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial h_4} \frac{\partial h_4}{\partial h_1} \frac{\partial h_1}{\partial w_{11}} = ((\hat{y}_1 - y_1) * v_{11} + (\hat{y}_2 - y_2) * v_{12}) * 1 * x_1 = (((2 - 1) * 1) + (0 - 1) * (-0.5)) * 1 * 1 = 1.5$$
$$(1.5p): w_{11}^* = w_{11} - \alpha \frac{\partial L}{\partial w_{11}} w_{11}^* = -0.5 - 0.1 * 1.5 = -0.65$$



Sample Solution

## Problem 7 Batch Normalization (6 credits)

Given a batch of inputs  $X$ , the batch normalization layers normalize the features according to the formula:

$$\hat{X} = \frac{X - E[X]}{\sqrt{\text{Var}[X]}}$$

$$y = \gamma \cdot \hat{X} + \beta$$

0

7.1 In general, why is it useful to apply a batch normalization layer after linear (fully connected / convolutional) layers?

1

(1p): Gradients are less varying in their magnitudes (stable gradients) and a smoothing of the gradients which leads to efficient backprop. For training that means they are less sensitive to initialisation, faster convergence and larger possible learning rates.

0

7.2 What are  $\gamma$  and  $\beta$ ? What is their purpose?

1

(1p):  $\gamma$  and  $\beta$  are learnable parameters.

2

(1p): It enables to undo the scaling and shifting in training.

0

7.3 Consider a BatchNorm2d() layer for convolutions that operates on an input  $X_{8 \times 3 \times 32 \times 64}$  ( $\text{BatchSize} \times \text{Channels} \times \text{Height} \times \text{Width}$ ). How many parameters do the running-mean and running-variance variables hold?

1

(1p):  $2 \cdot 3 = 6$ .

0

7.4 What is the consequence on a batch normalization layer when choosing a small batch size?

1

(1p): The running variables would not accurately represent the correct mean and variance of the distribution of the dataset (noisy or non-representative estimate).

2

(1p): Hence damaging the performance during inference (testing) time.

## Problem 8 Optimization (10 credits)

Remember that both Gradient descent (GD) and stochastic-gradient descent (SGD) could be performed by using batches of inputs.

8.1 What is the main difference in definition between GD and SGD?

(1p): GD performs the optimizer update every epoch (one pass over the entire training-set), while SGD performs that every iteration (batch).

0  
 1  
 2

8.2 Name two advantages of SGD over GD.

(1p): SGD introduces noise (regularization) to the optimization step, hence could escape saddle points.  
(1p): Requires more optimizer steps, but converges faster time-wise.

0  
 1  
 2

Revisit the formula of RMSProp:

$$s^{k+1} = \beta s^k + (1 - \beta)[\nabla_{\theta} L \circ \nabla_{\theta} L]$$
$$\theta^{k+1} = \theta^k - \alpha \frac{\nabla_{\theta} L}{\sqrt{s^{k+1}} + \epsilon}$$

Where  $\theta^k$  are the learnable weights at time step  $k$ ,  $\alpha$  is the learning rate,  $\beta$  is the exponential coefficient and  $\nabla_{\theta} L$  is the gradient of the loss w.r.t to  $\theta$  ( $\frac{\partial L}{\partial \theta}$ )

8.3 Which problem of SGD is addressed by RMSProp?

(1p): SGD converges slowly or SGD has a noisy optimization trajectory.

0  
 1

8.4 How does RMSProp solve this problem?

(1p): Dampening the oscillations for high-variance directions which enables a higher learning rate.

0  
 1

Adam is the state-of-the-art optimizer for deep learning optimization problems, and is being used widely. It is defined by the formula:

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \epsilon}$$

Including the bias-correction steps:

$$m^{k+1} = \beta_1 \cdot m^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k)$$

$$v^{k+1} = \beta_2 \cdot v^k + (1 - \beta_2) [\nabla_{\theta} L(\theta^k) \circ \nabla_{\theta} L(\theta^k)]$$

$$\hat{m}^{k+1} = \frac{m^{k+1}}{1 - \beta_1^{k+1}}$$

$$\hat{v}^{k+1} = \frac{v^{k+1}}{1 - \beta_2^{k+1}}$$

0  8.5 Which optimizers' concept does Adam combine?

- 1 (0.5p) RMSProp  
(0.5p) Momentum

0  8.6 Why do we apply the bias correction?

- 1 (1p): The first iterations are very biased towards the random initialization of the of  $m_0$  and  $v_0$ .  
2 (1p): The bias correction uses the entire magnitude of the gradient (before the learning rate) and allows much bigger steps towards the optimum.

0  8.7 Write the Netwon's method update step for  $w_{k+1}$  for a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ ,  $f(x, w) = wx$ , in terms of  $f(x, w)$ .

1 (1p):  $w_{k+1} = w_k - \frac{f'(x, w)}{f''(x, w)}$

0  8.8 Name one advantage of Newton's Method.

- 1 (1p): Converges much faster or removes the need of a learning rate.

## Problem 9 Advanced Topics (5 credits)

Consider the Recurrent Neural Network architecture (RNN):

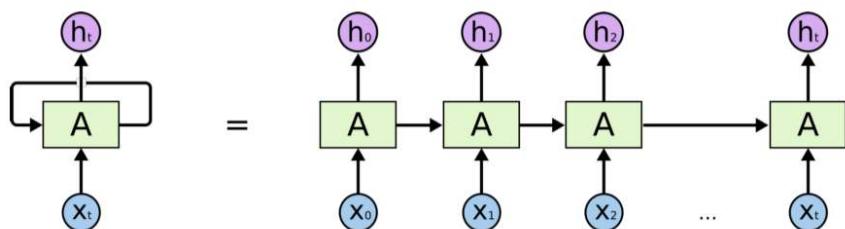


Figure 9.1: A Recurrent Neural Network

While training the model, we noticed that training loss does not drop.

9.1 What is the problem with RNNs that could cause this issue?

(1p): Vanishing / Exploding gradient (1p for either)

0  
1

9.2 Explain why it occurs.

(1p): The shared weights are multiplied continuously by themselves. If their eigenvalues are lower than 1, the gradient would vanish. If > 1, they will explode.

0  
1

9.3 Name one solution to the problem

(1p): If vanishing: Switch to LSTM/enforce eigenvalues are 1 during training (1p for either)  
If exploding: gradient clipping

0  
1

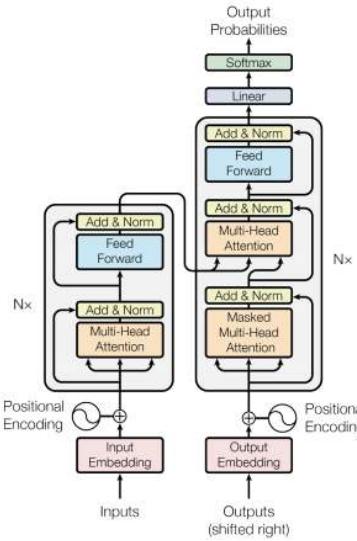


Figure 9.2: Transfomer Architecture

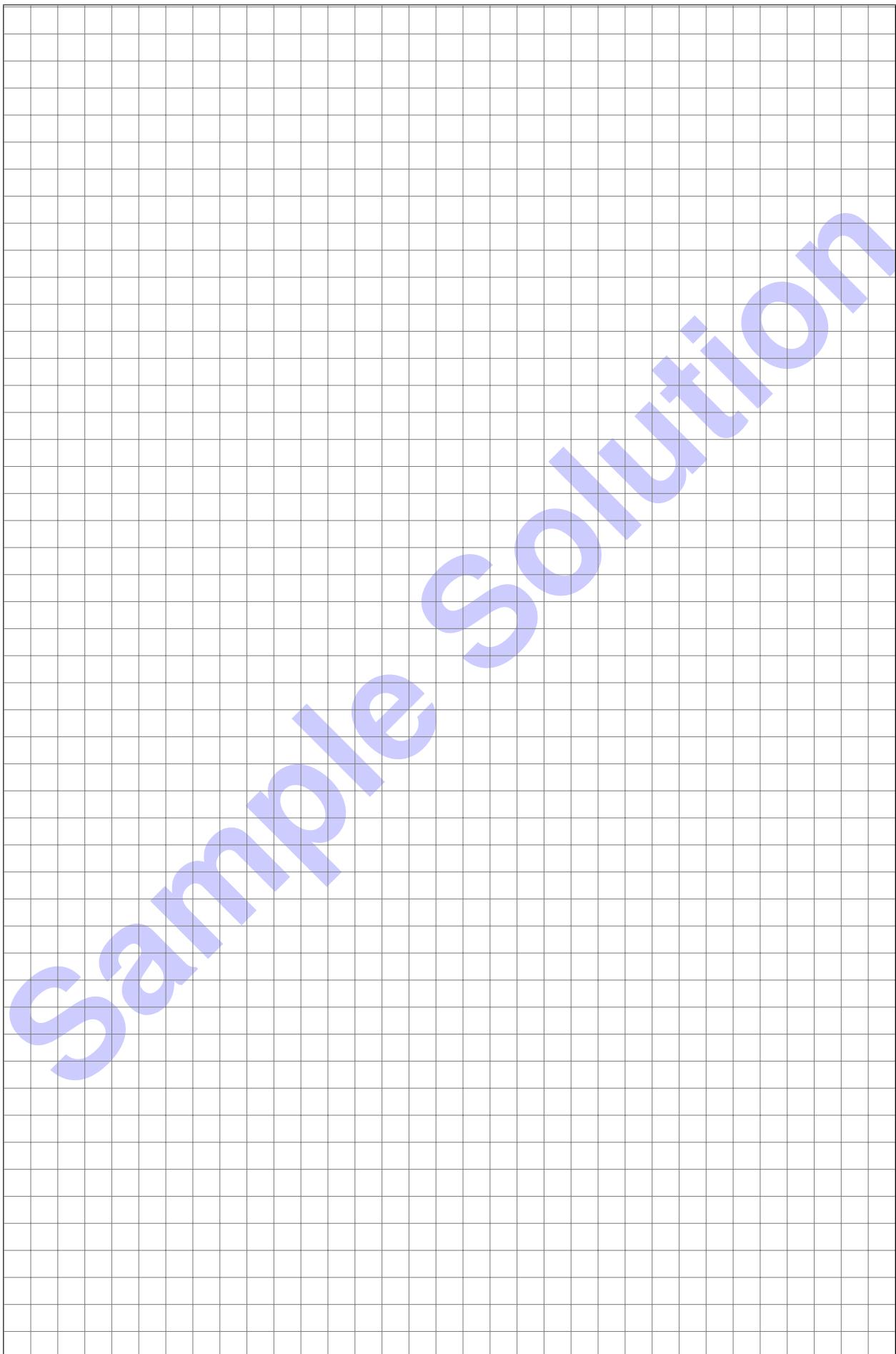
- 0 9.4 Consider the transformer architecture above which is used for machine translation by training to predict the next word in a sentence. The network gets provided with a full input and output sentence. Why do the outputs need to be masked when fed into the decoder?

(1p): Because the full output sentence is presented we have to mask out the part of the output sentence after our predicted word.

- 0 9.5 Can the transformer take an input sentence of arbitrary length? Explain why or why not.

(1p): No, because the attention layer (with fixed sized matrices K, Q, V) takes a fixed input.

**Additional space for solutions—clearly mark the (sub)problem your answers are related to and strike out invalid solutions.**



Sample Solution

Sample Solution

Sample Solution

# Introduction to Deep Learning (I2DL)

## Mock Exam

IN2346 - SoSe 2020

Technical University of Munich

Problem		Full Points	Your Score
1	Multiple Choice	10	
2	Short Questions	12	
3	Backpropagation	9	
<b>Total</b>		<b>31</b>	

Total Time: **31 Minutes**

Allowed Ressources: **None**

The purpose of this mock exam is to give you an idea of the type of problems and the structure of the final exam. The mock exam is not graded. The final exam will most probably be composed of 90 graded points with a total time of 90 minutes.

### Multiple Choice Questions:

- For all multiple choice questions any number of answers, i.e. either zero (!) or one or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

### How to Check a Box:

- Please **cross** the respective box:  (interpreted as **checked**)
- If you change your mind, please **fill** the box:  (interpreted as **not checked**)
- If you change your mind again, please **circle** the box:  (interpreted as **checked**)

## Part I: Multiple Choice (10 points)

1. (2 points) To avoid overfitting, you can...
  - increase the size of the network.
  - use data augmentation.
  - use Xavier initialization.
  - stop training earlier.
  
2. (2 points) What is true about Dropout?
  - The training process is faster and more stable to initialization when using Dropout.
  - You should not use weaky ReLu as non-linearity when using Dropout.
  - Dropout acts as regularization.
  - Dropout is applied differently during training and testing.
  
3. (2 points) What is true about Batch Normalization?
  - Batch Normalization uses two trainable parameters that allow the network to undo the normalization effect of this layer if needed.
  - Batch Normalization makes the gradients more stable so that we can train deeper networks.
  - At test time, Batch Normalization uses a mean and variance computed on training samples to normalize the data.
  - Batch Normalization has learnable parameters.
  
4. (2 points) Which of the following optimization methods use first order momentum?
  - Stochastic Gradient Descent
  - Adam
  - RMSProp
  - Gauss-Newton
  
5. (2 points) Making your network deeper by adding more parametrized layers will always...
  - slow down training and inference speed.
  - reduce the training loss.
  - improve the performance on unseen data.
  - (Optional: make your model sound cooler when bragging about it at parties.)

## Part II: Short Questions (12 points)

1. (2 points) You're training a neural network and notice that the validation error is significantly lower than the training error. Name two possible reasons for this to happen.

2. (2 points) You're working for a cool tech startup that receives thousands of job applications every day, so you train a neural network to automate the entire hiring process. Your model automatically classifies resumes of candidates, and rejects or sends job offers to all candidates accordingly. Which of the following measures is more important for your model? Explain.

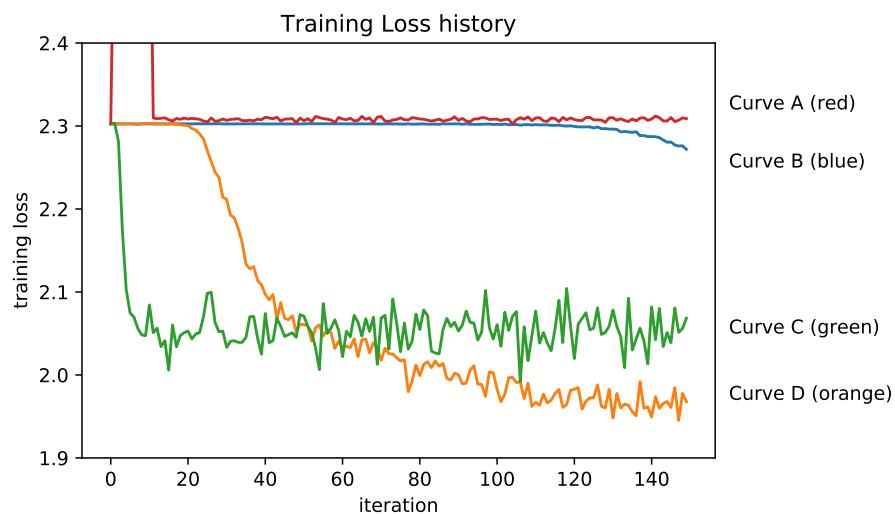
$$\text{Recall} = \frac{\text{True Positives}}{\text{Total Positive Samples}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{Total Predicted Positive Samples}}$$

3. (2 points) You're training a neural network for image classification with a very large dataset. Your friend who studies mathematics suggests: "If you would use Newton-Method for optimization, your neural network would converge much faster than with gradient descent!". Explain whether this statement is true (1p) and discuss potential downsides of following his suggestion (1p).

4. (2 points) Your colleague trained a neural network using standard stochastic gradient descent and L2 weight regularization with four different learning rates (shown below) and plotted the corresponding loss curves (also shown shown below). Unfortunately he forgot which curve belongs to which learning rate. Please assign each of the learning rate values below to the curve (A/B/C/D) it probably belongs to and explain your thoughts.

```
learning_rates = [3e-4, 4e-1, 2e-5, 8e-3]
```



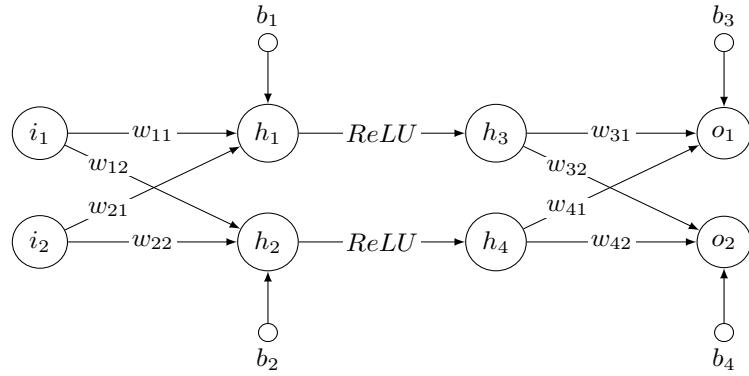
5. (1 point) Explain why we need activation functions.

6. (3 points) When implementing a neural network layer from scratch, we usually implement a ‘forward’ and a ‘backward’ function for each layer. Explain what these functions do, potential variables that they need to save, which arguments they take, and what they return.

7. (0 points) Optional: Given a Convolution Layer with 8 filters, a filter size of 6, a stride of 2, and a padding of 1. For an input feature map of  $32 \times 32 \times 32$ , what is the output dimensionality after applying the Convolution Layer to the input?

## Part III: Backpropagation (9 points)

1. (9 points) Given the following neural network with fully connection layer and ReLU activations, including two input units ( $i_1, i_2$ ), four hidden units ( $h_1, h_2$ ) and ( $h_3, h_4$ ). The output units are indicated as ( $o_1, o_2$ ) and their targets are indicated as ( $t_1, t_2$ ). The weights and bias of fully connected layer are called  $w$  and  $b$  with specific sub-descriptors.



The values of variables are given in the following table:

Variable	$i_1$	$i_2$	$w_{11}$	$w_{12}$	$w_{21}$	$w_{22}$	$w_{31}$	$w_{32}$	$w_{41}$	$w_{42}$	$b_1$	$b_2$	$b_3$	$b_4$	$t_1$	$t_2$
Value	2.0	-1.0	1.0	-0.5	0.5	-1.0	0.5	-1.0	-0.5	1.0	0.5	-0.5	-1.0	0.5	1.0	0.5

- (a) (3 points) Compute the output ( $o_1, o_2$ ) with the input ( $i_1, i_2$ ) and network parameters as specified above. Write down all calculations, including intermediate layer results.

- (b) (1 point) Compute the mean squared error of the output  $(o_1, o_2)$  calculated above and the target  $(t_1, t_2)$ .

- (c) (5 points) Update the weight  $w_{21}$  using gradient descent with learning rate 0.1 as well as the loss computed previously. (Please write down all your computations.)

**Additional Space for solutions.** Clearly mark the problem your answers are related to and strike out invalid solutions.

A large grid of squares, approximately 20 columns by 30 rows, intended for students to write their solutions. The grid is composed of thin black lines on a white background.

# Introduction to Deep Learning (I2DL)

## Mock Exam - *Solutions*

IN2346 - SoSe 2020

Technical University of Munich

Problem		Full Points	Your Score
1	Multiple Choice	10	
2	Short Questions	12	
3	Backpropagation	9	
<b>Total</b>		<b>31</b>	

Total Time: **31 Minutes**

Allowed Ressources: **None**

The purpose of this mock exam is to give you an idea of the type of problems and the structure of the final exam. The mock exam is not graded. The final exam will most probably be composed of 90 graded points with a total time of 90 minutes.

### Multiple Choice Questions:

- For all multiple choice questions any number of answers, i.e. either zero (!) or one or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

### How to Check a Box:

- Please **cross** the respective box:  (interpreted as **checked**)
- If you change your mind, please **fill** the box:  (interpreted as **not checked**)
- If you change your mind again, please **circle** the box:  (interpreted as **checked**)

## Part I: Multiple Choice (10 points)

1. (2 points) To avoid overfitting, you can...
  - increase the size of the network.
  - use data augmentation.**
  - use Xavier initialization.
  - stop training earlier.**
  
2. (2 points) What is true about Dropout?
  - The training process is faster and more stable to initialization when using Dropout.
  - You should not use weaky ReLu as non-linearity when using Dropout.
  - Dropout acts as regularization.**
  - Dropout is applied differently during training and testing.**
  
3. (2 points) What is true about Batch Normalization?
  - Batch Normalization uses two trainable parameters that allow the network to undo the normalization effect of this layer if needed.**
  - Batch Normalization makes the gradients more stable so that we can train deeper networks.**
  - At test time, Batch Normalization uses a mean and variance computed on training samples to normalize the data.**
  - Batch Normalization has learnable parameters.**
  
4. (2 points) Which of the following optimization methods use first order momentum?
  - Stochastic Gradient Descent
  - Adam**
  - RMSProp
  - Gauss-Newton
  
5. (2 points) Making your network deeper by adding more parametrized layers will always...
  - slow down training and inference speed.**
  - reduce the training loss.
  - improve the performance on unseen data.
  - (Optional: make your model sound cooler when bragging about it at parties.)**

## Part II: Short Questions (12 points)

1. (2 points) You're training a neural network and notice that the validation error is significantly lower than the training error. Name two possible reasons for this to happen.

**Solution:**

The model performs better on unseen data than on training data - this should not happen under normal circumstances. Possible explanations:

- Training and Validation data sets are not from the same distribution
- Error in the implementation
- ...

2. (2 points) You're working for a cool tech startup that receives thousands of job applications every day, so you train a neural network to automate the entire hiring process. Your model automatically classifies resumes of candidates, and rejects or sends job offers to all candidates accordingly. Which of the following measures is more important for your model? Explain.

$$\text{Recall} = \frac{\text{True Positives}}{\text{Total Positive Samples}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{Total Predicted Positive Samples}}$$

**Solution:**

Precision: High precision means low rate of false positives.

False Negatives are okay, since we get "thousands of applications" it's not too bad if we miss a few candidates even when they'd be a good fit. However, we don't want False Positives, i.e. offer a job to people who are not well suited.

3. (2 points) You're training a neural network for image classification with a very large dataset. Your friend who studies mathematics suggests: "If you would use Newton-Method for optimization, your neural network would converge much faster than with gradient descent!". Explain whether this statement is true (1p) and discuss potential downsides of following his suggestion (1p).

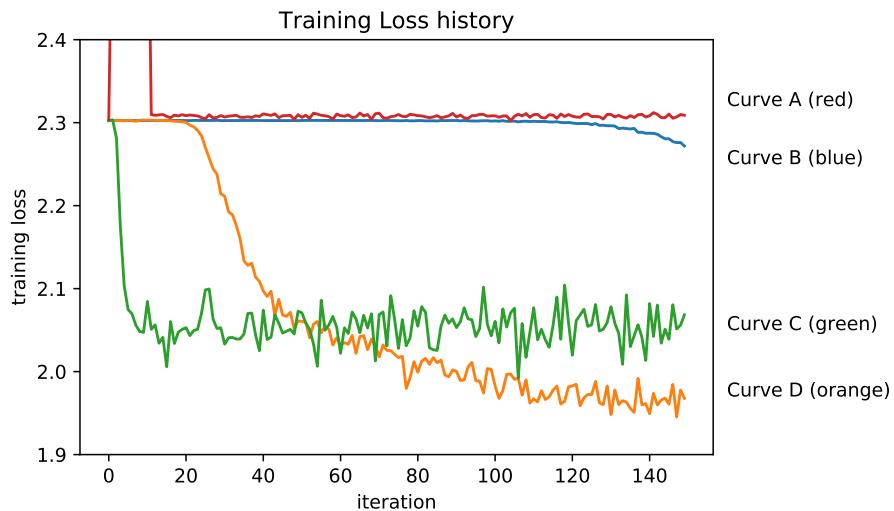
**Solution:**

Faster convergence in terms of number of iterations ("mathematical view"). (1 pt.)

However: Approximating the inverse Hessian is highly computationally costly, not feasible for high-dimensional datasets. (1 pt.)

4. (2 points) Your colleague trained a neural network using standard stochastic gradient descent and L2 weight regularization with four different learning rates (shown below) and plotted the corresponding loss curves (also shown shown below). Unfortunately he forgot which curve belongs to which learning rate. Please assign each of the learning rate values below to the curve (A/B/C/D) it probably belongs to and explain your thoughts.

`learning_rates = [3e-4, 4e-1, 2e-5, 8e-3]`

**Solution:**

Curve A:  $4e-1 = 0.4$  (Learning Rate is way too high)

Curve B:  $2e-5 = 0.00002$  (Learning Rate is too low)

Curve C:  $8e-3 = 0.008$  (Learning Rate is too high)

Curve D:  $3e-4 = 0.0003$  (Good Learning Rate)

5. (1 point) Explain why we need activation functions.

**Solution:**

Without non-linearities, our network can only learn linear functions, because the composition of linear functions is again linear.

6. (3 points) When implementing a neural network layer from scratch, we usually implement a ‘forward’ and a ‘backward’ function for each layer. Explain what these functions do, potential variables that they need to save, which arguments they take, and what they return.

**Solution:**

Forward Function:

- takes output from previous layer, performs operation, returns result (1 pt.)
- caches values needed for gradient computation during backprop (1 pt.)

Backward Function:

- takes upstream gradient, returns all partial derivatives (1 pt.)

7. (0 points) Optional: Given a Convolution Layer with 8 filters, a filter size of 6, a stride of 2, and a padding of 1. For an input feature map of  $32 \times 32 \times 32$ , what is the output dimensionality after applying the Convolution Layer to the input?

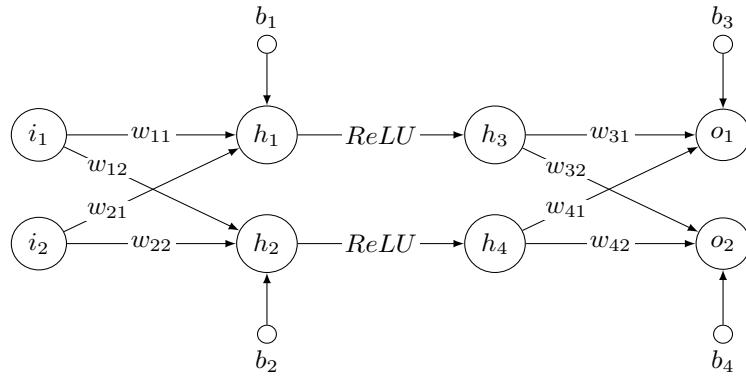
**Solution:**

$$\frac{32 - 6 + 2 \cdot 1}{2} + 1 = 14 + 1 = 15 \text{ (1 pt.)}$$

$15 \times 15 \times 8$  (1 pt.)

## Part III: Backpropagation (9 points)

1. (9 points) Given the following neural network with fully connection layer and ReLU activations, including two input units ( $i_1, i_2$ ), four hidden units ( $h_1, h_2$ ) and ( $h_3, h_4$ ). The output units are indicated as ( $o_1, o_2$ ) and their targets are indicated as ( $t_1, t_2$ ). The weights and bias of fully connected layer are called  $w$  and  $b$  with specific sub-descriptors.



The values of variables are given in the following table:

Variable	$i_1$	$i_2$	$w_{11}$	$w_{12}$	$w_{21}$	$w_{22}$	$w_{31}$	$w_{32}$	$w_{41}$	$w_{42}$	$b_1$	$b_2$	$b_3$	$b_4$	$t_1$	$t_2$
Value	2.0	-1.0	1.0	-0.5	0.5	-1.0	0.5	-1.0	-0.5	1.0	0.5	-0.5	-1.0	0.5	1.0	0.5

- (a) (3 points) Compute the output ( $o_1, o_2$ ) with the input ( $i_1, i_2$ ) and network parameters as specified above. Write down all calculations, including intermediate layer results.

**Solution:**

Forward pass:

$$h_1 = i_1 \times w_{11} + i_2 \times w_{21} + b_1 = 2.0 \times 1.0 - 1.0 \times 0.5 + 0.5 = 2.0$$

$$h_2 = i_1 \times w_{12} + i_2 \times w_{22} + b_2 = 2.0 \times -0.5 + -1.0 \times -1.0 - 0.5 = -0.5$$

$$h_3 = \max(0, h_1) = h_1 = 2$$

$$h_4 = \max(0, h_2) = 0$$

$$o_1 = h_3 \times w_{31} + h_4 \times w_{41} + b_3 = 2 \times 0.5 + 0 \times -0.5 - 1.0 = 0$$

$$o_2 = h_3 \times w_{32} + h_4 \times w_{42} + b_4 = 2 \times -1.0 + 0 \times 1.0 + 0.5 = -1.5$$

- (b) (1 point) Compute the mean squared error of the output  $(o_1, o_2)$  calculated above and the target  $(t_1, t_2)$ .

**Solution:**

$$MSE = \frac{1}{2} \times (t_1 - o_1)^2 + \frac{1}{2} \times (t_2 - o_2)^2 = 0.5 \times 1.0 + 0.5 \times 4.0 = 2.5$$

- (c) (5 points) Update the weight  $w_{21}$  using gradient descent with learning rate 0.1 as well as the loss computed previously. (Please write down all your computations.)

**Solution:**

Backward pass (Applying chain rule):

$$\begin{aligned} \frac{\partial MSE}{\partial w_{21}} &= \frac{\partial \frac{1}{2}(t_1 - o_1)^2}{\partial o_1} \times \frac{\partial o_1}{\partial h_3} \times \frac{\partial h_3}{\partial h_1} \times \frac{\partial h_1}{\partial w_{21}} + \frac{\partial \frac{1}{2}(t_2 - o_2)^2}{\partial o_2} \times \frac{\partial o_2}{\partial h_3} \times \frac{\partial h_3}{\partial h_1} \times \frac{\partial h_1}{\partial w_{21}} \\ &= (o_1 - t_1) \times w_{31} \times 1.0 \times i_2 + (o_2 - t_2) \times w_{32} \times 1.0 \times i_2 \\ &= (0 - 1.0) \times 0.5 \times -1.0 + (-1.5 - 0.5) \times -1.0 \times -1.0 \\ &= 0.5 + -2.0 = -1.5 \end{aligned}$$

Update using gradient descent:

$$w_{21}^+ = w_{21} - lr * \frac{\partial MSE}{\partial w_{21}} = 0.5 - 0.1 * -1.5 = 0.65$$

**Additional Space for solutions.** Clearly mark the problem your answers are related to and strike out invalid solutions.

A large grid of squares, approximately 20 columns by 30 rows, intended for students to write their solutions. The grid is composed of thin black lines on a white background.



**Note:**

- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

## Introduction to Deep Learning

**Exam:** IN2346 / Endterm

**Examiner:** Prof. Leal-Taixé and Prof. Nießner

**Date:** Tuesday 11<sup>th</sup> August, 2020

**Time:** 08:00 – 09:30

P 1

P 2

P 3

P 4

P 5

P 6

P 7

P 8

--	--	--	--	--	--	--	--

Left room from \_\_\_\_\_ to \_\_\_\_\_

from \_\_\_\_\_ to \_\_\_\_\_

Early submission at \_\_\_\_\_

Notes \_\_\_\_\_



## Endterm

# Introduction to Deep Learning

Prof. Leal-Taixé and Prof. Nießner  
Chair of Visual Computing & Artificial Intelligence  
Department of Informatics  
Technical University of Munich

**Tuesday 11<sup>th</sup> August, 2020**  
**08:00 – 09:30**

### Working instructions

- This exam consists of **20 pages** with a total of **8 problems**.  
Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 90 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources: **none**
- Do not write with red or green colors nor use pencils.
- Physically turn off all electronic devices, put them into your bag and close the bag.
- If you need additional space for a question, use the additional pages in the back and properly note that you are using additional space in the question's solution box.

## Problem 1 Multiple Choice Questions: (18 credits)

- For all multiple choice questions any number of answers, i.e. either zero (!), one, all or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

### How to Check a Box:

- Please **cross** the respective box:  (interpreted as **checked**)
- If you change your mind, please **fill** the box:  (interpreted as **not checked**)
- If you change your mind again, please place a cross to the left side of the box:   (interpreted as **checked**)

a) Which of the following statements regarding successful ImageNet-classification architectures are correct?

- ResNet18 has 11 million parameters more than VGG16.
- AlexNet uses filters of different kernel sizes.
- InceptionV3 uses filters of different kernel sizes.
- VGG16 only uses convolutional layers.

b) You train a neural network and the train loss diverges. What are reasonable things to do? (check all that apply)

- Decrease the learning rate.
- Add dropout.
- Increase the learning rate.
- Try a different optimizer.

c) What is the correct order of operations for an optimization with gradient descent?

- (a) Update the network weights to minimize the loss.
- (b) Calculate the difference between the predicted and target value.
- (c) Iteratively repeat the procedure until convergence.
- (d) Compute a forward pass.
- (e) Initialize the neural network weights.

- bcdea
- ebadc
- eadbc
- edbac

d) Consider a simple convolutional neural network with a single convolutional layer. Which of the following statements is true about this network?

- It is rotation invariant.
- It is translation equivariant.
- All input nodes are connected to all output nodes.
- It is scale-invariant.

e) Which of the following activation functions can lead to vanishing gradients?

- Tanh.
- ReLU.
- Sigmoid.
- Leaky Relu.

f) Logistic regression (check all that apply).

- Is a linear function.
- Is a supervised learning algorithm.
- Uses a type of cross-entropy loss.
- Allows to perform binary classification.

g) A sigmoid layer

- cannot be used during backpropagation.
- has a learnable parameter.
- maps surjectively to values in  $(-1, 1)$ , i.e., hits all values in that interval.
- is continuous and differentiable everywhere.

h) Your training loss does not decrease. What could be wrong?

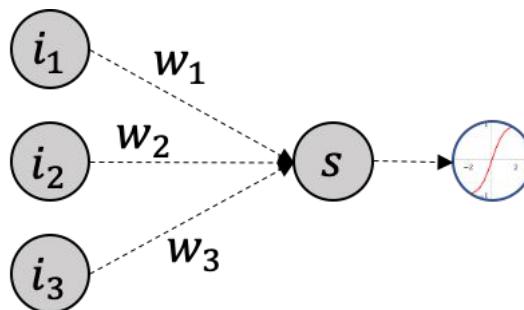
- Learning rate is too high.
- Too much regularization.
- Dropout probability not high enough.
- Bad initialization.

i) Which of the following have trainable parameters? (check all that apply)

- Leaky ReLU
- Batch normalization
- Dropout
- Max pooling

## Problem 2 Activation Functions and Weight Initialization (8 credits)

For your first job, you have to set up a neural network but you have some issue with its weight initialization. You remember from your I2DL lecture that you can sample the weights from a zero-centered normal distribution, but you can't remember which variance to use. Therefore, you set up a small network and try some numbers. You initialize the weights one time with  $\text{Var}(\mathbf{w}) = 0.02$  and one time with  $\text{Var}(\mathbf{w}) = 1.0$ :



Inputs:

- $i_1 = 2, i_2 = -4, i_3 = 1$

$\text{Var}(\mathbf{w}) = 0.02$ :

- $w_1 = 0.05, w_2 = 0.025, w_3 = -0.03$

$\text{Var}(\mathbf{w}) = 1.0$ :

- $w_1 = 1.0, w_2 = 0.5, w_3 = 1.5$

0 a) Compute a forward pass for each set of weights and draw the results of the linear layer in the Figure of the tanh plot. You don't need to compute the tanh.

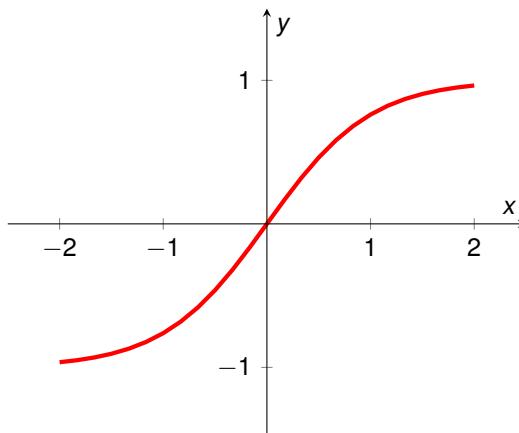
1  
2

$$\text{Var}(\mathbf{w}) = 0.02: s = (2 * 0.05) + (-4 * 0.025) + (1 * (-0.03)) = -0.03 \text{ (0.5p)}$$

$$\text{Var}(\mathbf{w}) = 1.0: s = (2 * 1.0) + (-4 * 0.5) + (1 * 1.5) = 1.5 \text{ (0.5p)}$$

0.5p for each of the calculated numbers drawn correctly into the plot.

Note: Inaccurate drawing is tolerated, but positive instead of negative isn't



b) Using the results above, explain what problems can arise during backpropagation of deep neural networks when initializing the weights with too small and too large variance. Also, explain the root of these problems.

Too small variance: The output in deeper layers is close to zero. Therefore, the gradient w.r.t the weights also becomes very small (vanishing gradient).

(0.5p for explanation, 0.5p for small gradients, i.e. only "vanishing gradient is 0.5p. Needs to mention gradients)

Too large variance: The tanh activation function saturates. Therefore, the gradient w.r.t the weights becomes very small (vanishing gradient).

Note: again, only vanishing gradient is 0.5p

0  
1  
2

c) Which initialization scheme did you learn in the lecture that tackles these problems? What does this initialization try to achieve in the activations of deep layers of the neural network?

Xavier initialization (1p)

Note: 0.5p for only writing down the formula

The goal is to keep the variance of the output is the same as the input. (1p)

0  
1  
2

d) After switching from tanh to ReLU activation functions, one of your initial problems occurs again. Why does this happen? How can you modify the initialization scheme proposed in c) to adjust it for this new non-linearity?

ReLU "kills" half of the outputs. Therefore, the output variance of a layer is halved. (1p)

Note: only mentioning that  $\text{relu}(x) = 0$  for  $x < 0$  is only 0.5p

Solution: Multiply Xavier initialization with 2/He initialization (1p)

Note: only name gives 0.5p

0  
1  
2

### Problem 3 Batch Normalization and Computation Graphs (6 credits)

For an input vector  $\mathbf{x}$  as well as variables  $\gamma$  and  $\beta$  the general formula of batch normalization is given by

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - E[\mathbf{x}]}{\sqrt{Var[\mathbf{x}]}}$$

$$\mathbf{y} = \gamma \hat{\mathbf{x}} + \beta.$$

0 a) Why would one want to apply batch normalization in a neural network?

1 Prevent covariate shift/ Mimics the normalization of data for each layer to receive more stable inputs (1p)

Note: Only 0.5p for stable gradient, regularization, able to train deeper network, stabilize training, faster training ...

0 b) Why are  $\gamma$  and  $\beta$  needed in the batch normalization formula?

1 Allows the network to "undo"/scale+shift the normalization. (1p)

Note: 0.5p if only one of scaling and shift are mentioned

Stating that gamma and beta are hyperparameters does not get any points

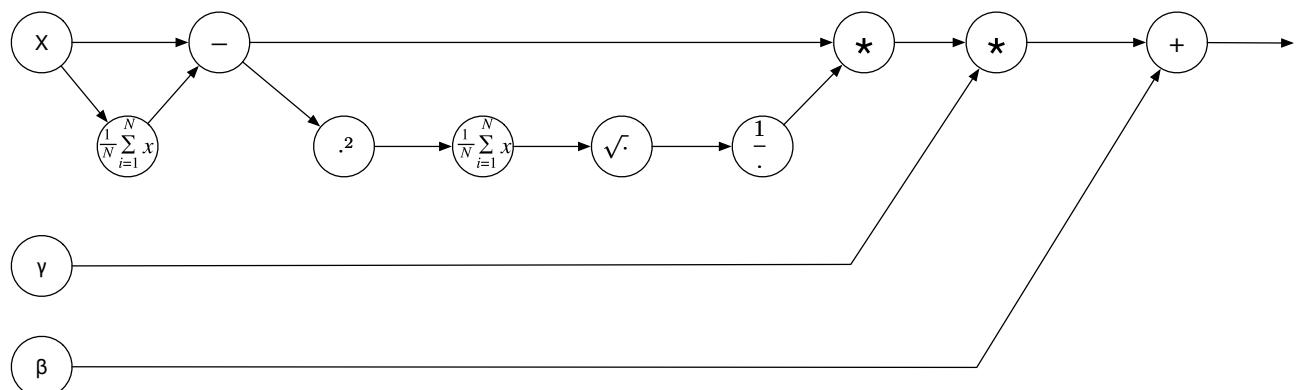
0 c) How is a batch normalization layer applied at training (1p) and at test (1p) time?

1 Train time: Calculate the mean and variance for each feature across the mini batch (0.5p) and store a weighted average across training mini-batches for the update at test time (0.5p)

Test time: Use exponentially weighted / moving / running average mean and variance that was computed at training time on the test samples (1p)

Note: 0.5p only if just simply mentioned acquiring statistics from the training set, but failed to illustrate which method is needed to acquire mean and var, i.e. exponentially weighted average

0 d) Computational graph of a batch normalization layer. Fill out the nodes (circles) of the following computational graph. Each node can consist of one of the following operations  $+$ ,  $-$ ,  $*$ ,  $^2$ ,  $\sqrt{\cdot}$ ,  $\frac{1}{\cdot}$ .



Note: -1p for each wrong operator

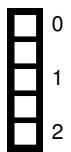
## Problem 4 Convolutional Neural Networks and Receptive Field (12 credits)

A friend of yours asked for a quick review of convolutional neural networks. As he has some background in computer graphics, you start by explaining previous uses of convolutional layers.

- a) You are given a two dimensional input (e.g., a grayscale image). Consider the following convolutional kernels

$$C_1 = \frac{1}{9} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

$$C_2 = \begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}.$$



What are the effects of the filter kernels  $C_1$  and  $C_2$  when applied to the image?

$C_1$ : Local/box (0.5p) blur/smoothing (0.5p) kernel. Note: If only mean/arg is mentioned instead of blur then 0.5p

$C_2$ : vertical (0.5p) edge detector (0.5p)

After showing him some results of a trained network, he immediately wants to use them and starts building a model in Pytorch. However, he is unsure about the layer sizes so you quickly help him out.

- b) Given a Convolution Layer in a network with 5 filters, filter size of 7, a stride of 3, and a padding of 1. For an input feature map of  $26 \times 26 \times 26$ , what is the output dimensionality after applying the Convolution Layer to the input?

$$8 \times 8 \times 5 \text{ (2p)} \quad 1\text{p for only kernel size computation } \lfloor \frac{26 - 7 + 2 * 1}{3} \rfloor + 1 = 7 + 1 = 8$$



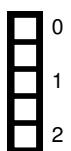
- c) You are given a convolutional layer with 4 filters, kernel size 5, stride 1, and no padding that operates on an RGB image.

1. What is the shape of its weight tensor?
2. Name all dimensions of your weight tensor.

Shape: (3, 4, 5, 5) or (4,3,5,5) (1p)

Reasoning: input size/rbg channels, output size/channels, width, height (1p)

Note: -1p if only 3 dimensions are mentioned, -1p if 5's are simply described as filter size instead of width,height

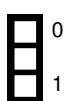


Now that he knows how to combine convolutional layers, he wonders how deep his network should be. After some thinking, you illustrate the concept of receptive field to him by these two examples. For the following two questions, consider a grayscale 224x224 image as network input.

- d) A convolutional neural network consists of 3 consecutive  $3 \times 3$  convolutional layers with stride 1 and no padding. How large is the receptive field of a feature in the last layer of this network?

$$1 \times 1 \rightarrow 3 \times 3 \rightarrow 5 \times 5 \rightarrow 7 \times 7 \text{ (1p)}$$

Note: -0.5p if no tuple



0  
1  
2

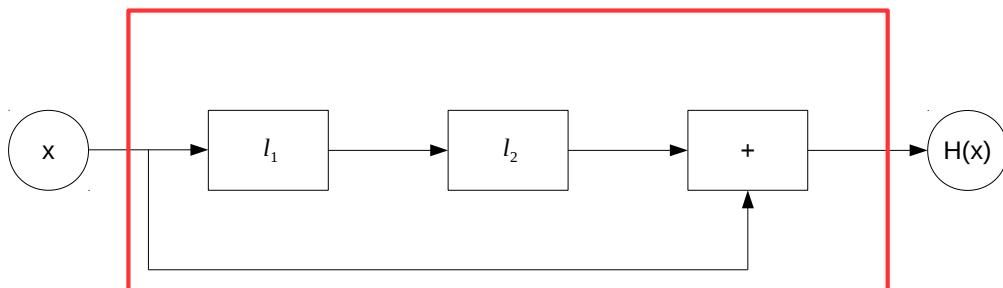
- e) Consider a network consisting of a single layer.
1. What layer choice has a receptive field of 1?
  2. What layer has a receptive field of the full image input?

1x1 convolution or identity (1p)

fully connected or conv/pooling layer with kernel size equals full input size (224x224) (1p)

If the answer is reasonable but incomplete: -0.5p

Blindly, he stacks 10 convolutional layers together to solve his task. However, the gradients seem to vanish and he can't seem to be able to train the network. You remember from your lecture that ResNet blocks were designed for these purposes.



- f) Draw a ResNet block in the image above (1p) containing two linear layers, which you can represent by  $l_1$  and  $l_2$ . For simplicity, you don't need to draw any non-linearities. Why does such a block improve the vanishing gradient problem in deep neural networks (1p)?

0  
1  
2

1p for correct drawing

Note: if image structure is correct but: i) arrow is missing or ii) "+" symbol is missing or not drawn correctly 0.5p

1p for highway of gradients

Note: if only forward pass is mentioned then no points

- g) For your above drawing, given the partial derivative of the residual block  $R(x) = l_2(l_1(x))$  as  $\frac{\partial R(x)}{\partial x} = r$ , calculate  $\frac{\partial H(x)}{\partial x}$ .

$$\frac{\partial H(x)}{\partial x} = \frac{\partial x + R(x)}{\partial x} = \frac{\partial x}{\partial x} + \frac{\partial R(x)}{\partial x} = 1 + r$$

1p for  $\frac{\partial H(x)}{\partial x} = 1 + r$

## Problem 5 Training a Neural Network (15 credits)

A team of architects approaches you for your deep learning expertise. They have collected nearly 5,000 hand-labeled RGB images and want to build a model to classify the buildings into their different architectural styles. Now they want to classify images of architectures into 3 classes depending on their style:



Islamic



Baroque



Soochow

- a) How would you split your dataset and give a meaningful percentage as answer.



0.5 pt: Train + Validation + Test (must mention all 3)  
0.5 pt: meaningful percentages (training split must be > 50%)

- b) After visually inspecting the different splits in the dataset, you realize that the training set only contains pictures taken during the day, whereas the validation set only has pictures taken at night. Explain what is the issue and how you would correct it.



Issue (1p):  
0.5 pt: data from different distributions / mismatch and therefore  
0.5 pt: bad generalization / high val error  
How to correct (1p): mix training + validation data, shuffle, split again => data in all sets from same distribution  
Note:  
- zero points for "cross validation" (without mentioning re-shuffling first)  
- zero points for data augmentation (simply changing brightness will not lead to realistic night images!)

- c) As you train your model, you realize that you do not have enough data. Unfortunately, the architects are unable to collect more data so you have to temper the data. Provide 4 data augmentation techniques that can be used to overcome the shortage of data.



Rotation, cropping, flipping, adding noise, mirroring, ... (0.5p each, max 2p)  
Note: if they named more than 4 actions: only grade first 4 ones

0 What is the saddle point and what is the problem with GD?

1

Problem: Gradient is zero at saddle point/ local minimum (0.5p) → Optimization can get stuck (0.5)

0 e) While training your classifier you experience that loss only slowly converges and always plateaus independent of the used learning rate. Now you want to use Stochastic Grading Descent (SGD) instead of Gradient Descent (GD). What is an advantage of SGD compared to GD in dealing with saddle points?

1

SGD has noisier updates (0.5p) → can help escape from a saddle point or local minima (0.5p)

0 f) Explain the concept behind momentum in SGD

1

Avoid getting stuck in local minima or accelerate optimization (1p)

Note: informal explanations: 0.5p if somewhat correct, 0p if too imprecise

0 g) Why would one want to use larger mini-batches in SGD?

1

1p: make the gradients less noisy  
only “less noisy” (not mentioning gradients) – > half a point

0 h) Why do we usually use small mini-batches in practice?

1

Limited GPU memory (1p)  
“faster computation” -> 0 points

0 i) There exists a whole zoo of different optimizers. Name an optimizer that uses both first and second order moment

1

Adam (1p)

0 j) Choosing a reasonable learning rate is not easy.

1

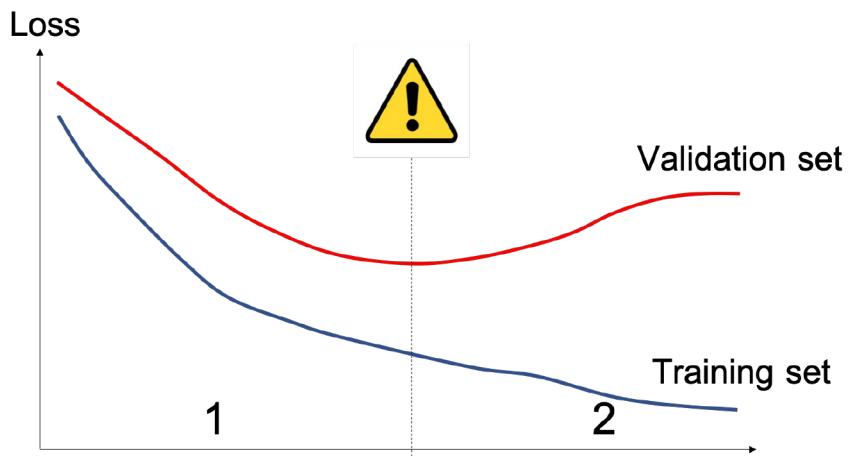
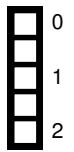
1. Name a problem that will result from using a learning rate that is too high (1p).
2. Name a problem that will arise from using a learning rate that is too low (1p)?

Too high: cost function does not converge to an optimal solution and can even diverge. (1p)

Note: if divergence (or no convergence) not mentioned: -0.5p

Too low: cost function may not converge to an optimal solution, or will converge after a very long time. (1p)

k) Finally you plot the loss curves with a suitable learning rate for both training data and validation data. What's the issue of period 2 called? Name a possible actions that you could do without changing the number of parameters in your network to counteract this problem.



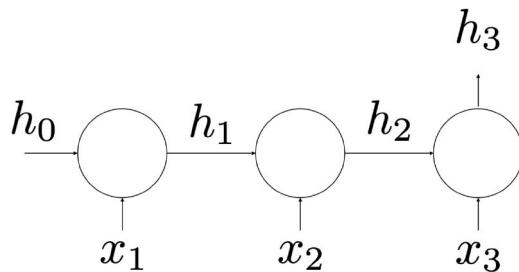
Issue: Model is overfitting (Note: generalization gap only is not enough) (1p).

Action: weight decay or drop out, data augmentation (1p)

Note: if only regularization 0.5p

## Problem 6 Recurrent Neural Networks and Backpropagation (9 credits)

Consider a vanilla RNN cell of the form  $h_t = \tanh(V \cdot h_{t-1} + W \cdot x_t + b)$ . The figure below shows the input sequence  $x_1, x_2$ , and  $x_3$ .



- 0 a) Given the dimensions  $x_t \in \mathbb{R}^3$  and  $h_t \in \mathbb{R}^5$ , what is the number of parameters in the RNN cell? (Calculate final number)

1

$3 \times 5 + 5 \times 5 + 5(\text{bias}) = 15 + 25 + 5 = 45$  (1p for 45, else 0)

- 0 b) If  $x_t$  and  $b$  are the 0 vector, then  $h_t = h_{t-1}$  for any value of  $h_t$ . Discuss whether this statement is correct.

1

False: (0.5p)

After transformation with  $V$  and non-linearity  $x_t = 0$  does not lead to  $h_t = h_{t-1}$  (0.5p), i.e.  $h_t$  can be changed.

Note: simply repeating the formula  $h_t = \tanh(V \cdot h_{t-1})$ ) does not give any points. If you only mention  $V$  or  $\tanh$  then this is also correct, though giving an incorrect formula invalidates that half point.

Now consider the following **one-dimensional** ReLU-RNN cell without bias  $b$ .

$$h_t = \text{ReLU}(V \cdot h_{t-1} + W \cdot x_t)$$

(Hidden state, input, and weights are scalars)

- 0 c) Calculate  $h_2$  and  $h_3$  where

1

$$V = -3, \quad W = 3, \quad h_0 = 0, \quad x_1 = 2, \quad x_2 = 3 \quad \text{and} \quad x_3 = 1.$$

2

$h_0 = 0$

$h_1 = \text{relu}(-3 \cdot 0 + 3 \cdot 2) = 6$

$h_2 = \text{relu}(-3 \cdot 6 + 3 \cdot 3) = 0 \quad (1 \text{ p})$

$h_3 = \text{relu}(-3 \cdot 0 + 3 \cdot 1) = 3 \quad (1 \text{ p})$

Note: Only points for correct solutions, no points for intermediate steps (even if you have an incorrect  $h_1$ )

d) Calculate the derivatives  $\frac{\partial h_3}{\partial V}$ ,  $\frac{\partial h_3}{\partial W}$ , and  $\frac{\partial h_3}{\partial x_1}$  for the forward pass of the ReLU-RNN where

$$V = -2, \quad W = 1, \quad h_0 = 2, \quad x_1 = 2, \quad x_2 = \frac{3}{2} \quad \text{and} \quad x_3 = 4.$$

for the forward outputs

$$h_1 = 0, \quad h_2 = \frac{2}{3}, \quad h_3 = 1.$$

Use that  $\left. \frac{\partial}{\partial x} \text{ReLU}(x) \right|_{x=0} = 0$ .

Generally:

$$\begin{aligned}\frac{\partial h_t}{\partial V} &= h_{t-1} + V \cdot \frac{\partial h_{t-1}}{\partial V} \\ \frac{\partial h_t}{\partial W} &= \frac{\partial \text{ReLU}(z_t)}{\partial z_t} \cdot \left( V \cdot \frac{\partial h_{t-1}}{\partial W} + x_t \right) \\ \frac{\partial h_t}{\partial x_\tau} &= \frac{\partial \text{ReLU}(z_t)}{\partial z_t} \cdot \left( V \cdot \frac{\partial h_t}{\partial x_\tau} + W \cdot \delta_{t\tau} \right)\end{aligned}$$

$$\frac{\partial h_3}{\partial V} = h_2 + V \cdot h_1 = \frac{2}{3} + 0 = \frac{2}{3} \quad (1p)$$

$$\frac{\partial h_3}{\partial W} = V \cdot x_2 + x_3 = -2 \cdot \frac{3}{2} + 4 = 1 \quad (1p)$$

$$\frac{\partial h_3}{\partial x_1} = 0 \quad (\text{dead ReLU}) \quad (1p)$$

Note: alternatively  $\frac{\partial h_3}{\partial V} = \frac{3}{2}$  if student correctly identified that  $h_2$  should have been flipped to be a correct forward pass.

For  $\frac{\partial h_3}{\partial x_1}$ , it's okay even if no formula, but some explanation is given (dead relu after first layer)

0 e) A Long-Short Term Memory (LSTM) unit is defined as

1

2

$$\begin{aligned} g_1 &= \sigma(W_1 \cdot x_t + U_1 \cdot h_{t-1}), \\ g_2 &= \sigma(W_2 \cdot x_t + U_2 \cdot h_{t-1}), \\ g_3 &= \sigma(W_3 \cdot x_t + U_3 \cdot h_{t-1}), \\ \tilde{c}_t &= \tanh(W_c \cdot x_t + u_c \cdot h_{t-1}), \\ c_t &= g_2 \circ c_{t-1} + g_3 \circ \tilde{c}_t, \\ h_t &= g_1 \circ c_t, \end{aligned}$$

where  $g_1$ ,  $g_2$ , and  $g_3$  are the gates of the LSTM cell.

- 1) Assign these gates correctly to the **forget**  $f$ , **update**  $u$ , and **output**  $o$  gates. (1p)
- 2) What does the value  $c_t$  represent in a LSTM? (1p)

$g_1$  = output gate

$g_2$  = forget gate

$g_3$  = update gate/input gate

(1p for all three, zero otherwise)

$c_t$ : cell state/memory (1p)

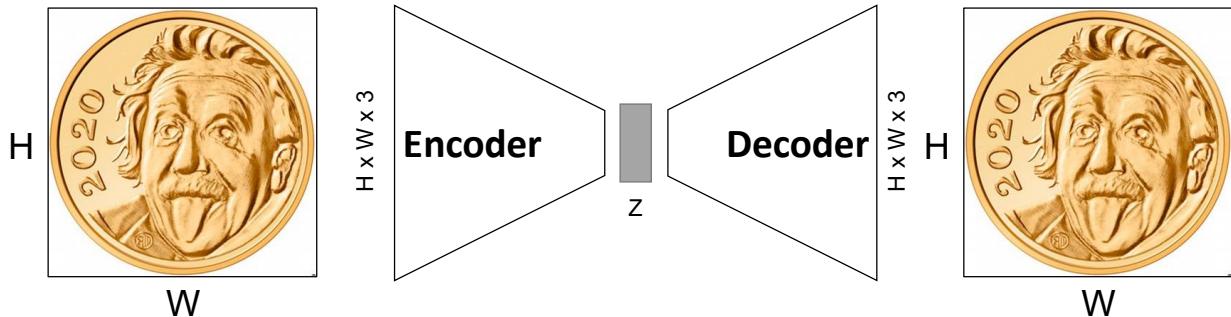
Note: if students interpreted  $c_t$  as "what does it do?" half a point was awarded. Possible half point: "Intermediate value, check what to forget and what to add from input"

## Problem 7 Autoencoder and Network Transfer (11 credits)

You are given a dataset containing 10,000 RGB images with height  $H$  and width  $W$  of single coins without any labels or additional information.



To work with the image dataset you build an autoencoder as depicted in the figure below:



The input of the encoder is the images of dimension  $(H \times W \times 3)$  which are transformed into a one-dimensional real vector with  $z$  entries. The latent code is used to decode the input image with the same dimension  $(H \times W \times 3)$ . Both encoder and decoder are neural networks and the combined network is trainable and uses the  $L_2$  loss as its optimization function.

- a) Is an autoencoder an example of unsupervised learning or supervised learning?

unsupervised learning (1p) if you give the wrong reasoning after correct choice then only 0.5p

0
1

- b) As the data gets scaled down from the original dimension to a lower-dimensional bottleneck, an autoencoder can be used for data compression. How does an autoencoder as described above differ from linear methods to reduce the dimensionality of the data such as PCA (principal component analysis)?

An autoencoder built of neural networks (containing non-linearities) is a non-linear function (1p)  
Note: Won't receive point if only mentioned "parameters", "conv layers"

0
1

- c) For an autoencoder we can vary the size of the bottleneck. Discuss briefly what may happen if

- i) the latent space is *too small* (1p).
- ii) the latent space is *too big* (1pt)

0
1
2

(i) too small: loss of information/bad reconstruction quality/too much compression/extraction/blurry/underfitting... (1p)  
(ii) too big: too little compression/extraction/overfitting/poor generalization/learned identity mapping/memorizes the inputs... (1p)

0  d) Now, you want to generate a random image of a coin. To do so, can you just randomly sample a vector from the latent space to generate a new coin image?

1

No (0.5p), network does not project input distribution surjectively into latent space → decoder will not project all latent space vectors to coin images (0.5)

Note: mentioning variational autoencoder does not give points unless it is specified what is different there -> “yes, variational autoencoder can do it” is 0 p, “no, you would need a variational autoencoder” is 0.5p, “no, you would need a variational autoencoder where you can sample from the gaussian latent space” is 1p



0  e) Now, someone gives you 1,000 images that are annotated for semantic segmentation of coin and background as shown in the image above. How would you change the architecture of the discussed autoencoder network to perform semantic segmentation?

1

Replace the last layer of the of the autoencoder/ decoder (0.5p) to output 1 or 2 channels (0.5p)

Note: “change output layer” is 1p with dimension otherwise 0.5p

“add conv layer” is 1p with dimensions, 0.5p for “add 1x1 conv” else 0p

“add/change/... FC layers” is 0p

mentioning optional improvements like “change to U-Net” does not lead to deduction

0  f) If you wanted to train the new semantic segmentation network what loss function would you use and how?

1

(Binary) Cross Entropy loss/hinge loss (1p) on pixel level/ over channels/ depth-wise (1p) for the two classes (coin and background)

Notes: “dice loss” 1p without explanation, 2p with explanation

“softmax loss” 0p

multiple losses listed: wrong loss + correct loss -> 50% deduction

0  g) How would you leverage your pretrained autoencoder for training a new segmentation network efficiently?

1

- use pretrained encoder of autoencoder and discard decoder (1p)

- (i) freeze weights of encoder or (ii) use very small learning rate for encoder during training for segmentation (1p)

Note: “use transfer learning” - only 1p, you have to mention “weight transfer” and “freeze” or “different learning rate” for encoder/first layers

“freeze all weights of autoencoder and train last layer only” - would not be efficient for segmentation -0.5p

0  h) Why do you expect the pretrained autoencoder variant to generalize more than a randomly initialized network?

1

access to much more data 1p

Note: can already detect features / features translate to other tasks 0.5p -> 1p for which features (encoder / low-level / shapes &edges / ...)

has prior knowledge of coins 0.5p

## Problem 8 Unsorted Short Questions (11 credits)

a) Why do we need activation functions in our neural networks?

Introduce non-linearity (1p), otherwise the network can be reduced to a single linear layer. (alternatively: to model non-linear functions).

0  
 1

b) You are solving the binary classification task of classifying images as cars vs. persons. You design a CNN with a single output neuron. Let the output of this neuron be  $z$ . The final output of your network,  $\hat{y}$  is given by:

$$\hat{y} = \sigma(\text{ReLU}(z)),$$

0  
 1

where  $\sigma$  denotes the sigmoid function. You classify all inputs with a final value  $\hat{y} \geq 0.5$  as car images. What problem are you going to encounter?

As  $\text{ReLU}(x) \geq 0$  we get  $\sigma(\text{ReLU}(z)) \geq 0.5 \quad \forall z$  (0.5p)  
→ classifier only predicts one class (1p)

0  
 1

c) Suggest a method to solve exploding gradients when training fully-connected neural networks.

Gradient clipping/ better weight initialization (e.g. Xavier initialization)/ Batch normalization

0  
 1

d)

Weight update with objective function  $J$  incl. weight decay:

$$W = W - \eta \nabla_W (J + \frac{1}{2} \lambda \sum_i W_i^2)$$
$$W = W(1 - \eta \lambda) - \eta \nabla_W J,$$

where  $\eta$  = learning rate and  $\lambda$  = regularization parameter (1p - an error in the equation or not taking the derivative of the L2 term subtracts 0.5 points.) With  $\eta \cdot \lambda \ll 1$ . (1p)

Value of  $W$  is pushed towards zero in each iteration since a constant amount is subtracted at each iteration. (1p - explanation gives one point if the derivation is correct or partially correct, but without derivation this does not give a point.)

0  
 1  
 2  
 3

- 0 f) You are given input samples  $\mathbf{x} = (x_1, \dots, x_n)$  for which each component  $x_j$  is drawn from a distribution with zero mean. For an input vector  $\mathbf{x}$  the output  $\mathbf{s} = (s_1, \dots, s_n)$  is given by

1

$$2 \quad s_i = \sum_{j=1}^n w_{ij} \cdot x_j,$$

3

4 where your weights  $w$  are initialized by a uniform random distribution  $U(-\alpha, \alpha)$ .

How do you have to choose  $\alpha$  such that the variance of the input data and the output is identical, hence  $\text{Var}(s) = \text{Var}(x)$ ?

**Hints:** For two statistically independent variables  $X$  and  $Y$  holds:

$$\text{Var}(X + Y) = [\text{E}(X)]^2 \text{Var}(Y) + [\text{E}(Y)]^2 \text{Var}(X) + \text{Var}(X)\text{Var}(Y)$$

Furthermore the PDF of an uniform distribution  $U(a, b)$  is

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise.} \end{cases}$$

The variance of a continuous distribution is calculated as

$$\text{Var}(X) = \int_R x^2 f(x) dx - \mu^2,$$

where  $\mu$  is the expected value of  $X$ .

$$\text{Var}(s) = \text{Var}\left(\sum_{j=0}^n w_{ij} \cdot x_j\right) = \sum_{j=0}^n \text{Var}(w_{ij} \cdot x_j) \quad (1p)$$

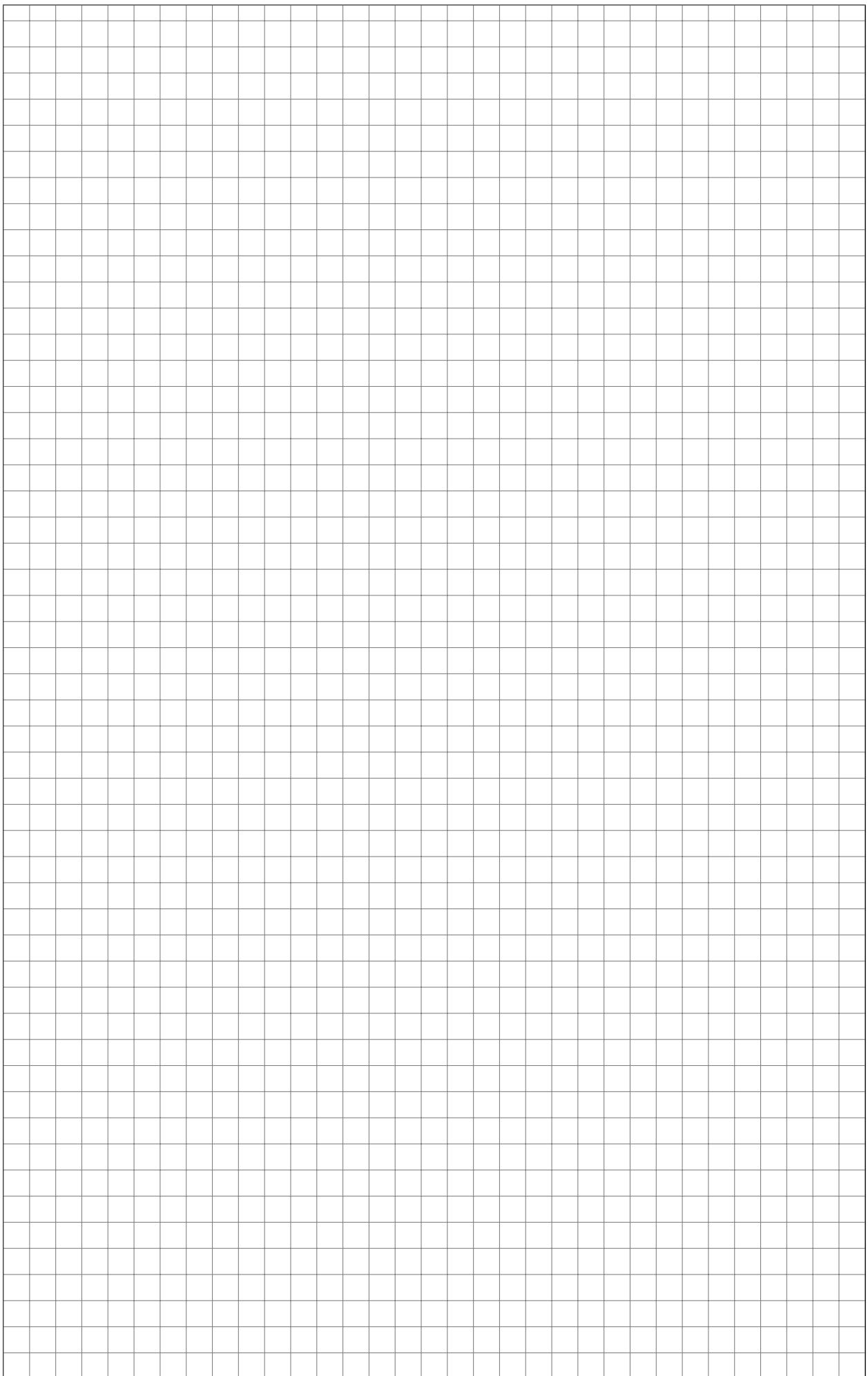
$$\sum_{j=0}^n \text{Var}(w_{ij} \cdot x_j) = n \cdot \text{Var}(w \cdot x) = n \cdot \text{Var}(w) \cdot \text{Var}(x) \quad (\text{independence}) \quad (1p)$$

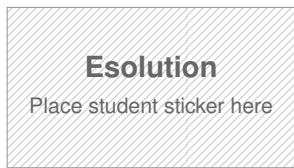
$$\text{Var}(w) = \frac{1}{3} \alpha^2 \quad (1p)$$

$$n \cdot \frac{1}{3} \alpha^2 = 1 \rightarrow \alpha = \sqrt{\frac{3}{n}} \quad (1p)$$

**Additional space for solutions—clearly mark the (sub)problem your answers are related to and strike out invalid solutions.**

A large grid of squares, approximately 20 columns by 25 rows, intended for students to write their solutions. The grid is composed of thin black lines on a white background.





**Note:**

- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

## Introduction to Deep Learning

**Exam:** IN2346 / endterm

**Date:** Tuesday 8<sup>th</sup> February, 2022

**Examiner:** Prof. Dr. Matthias Nießner

**Time:** 15:00 – 11:30

- The blackened exam has the same layout as the non-blackened exam with the acutal questions, which is going to be released once the working time starts.
- Only submit your personalized blackened exam. **DO NOT submit the non-blackened/non-personalized exam** (clearly indicated with “DO NOT SCAN/UPLOAD”).
- This final exam consists of **16 pages** with a total of **7 problems**. Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this simulation is **90 credits**.
- No additional resources are allowed.

## Problem 1 Multiple Choice (18 credits)

Mark correct answers with a cross



To undo a cross, completely fill out the answer option



To re-mark an option, use a human-readable marking



Please note:

- For all multiple choice questions any number of answers, i.e. either zero (!), one or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

1.1 You are training a network to classify images of handwritten digits in the range of [0,...,9] on the MNIST dataset. Which of the following data augmentation techniques are suitable to use for this task?

- Add Gaussian noise to the images
- Vertically flip the images
- Rotation of the images by 10 degrees
- Change the contrast of the images

1.2 What is true about Residual Blocks?

- Reduce the number of computations in the forward pass
- Act as a highway for gradient flow
- Enable a more stable training of larger networks
- Act as a regularizer

1.3 For a fully-convolutional 2D CNN, if we double the spatial dimensions of input images, ...

- ... the number of network parameters doubles
- ... the number of network parameters stays the same
- ... the receptive field of an arbitrary pixel in an intermediate activation map can decrease
- ... the dropout coefficient  $p$  must be corrected to  $\sqrt{p}$  in test time

1.4 What is true about Generative Adversarial Networks?

- The Generator minimizes the probability that the Discriminator is correct
- The Generator provides supervision for the Discriminator
- The Discriminator acts as a classifier
- The Discriminator samples from a latent space

1.5 Given input  $x$ , which of the following statements are always true? Note: For dropout, assume the same set of neurons are chosen.

- $\text{BatchNorm}(\text{ReLU}(x)) \equiv \text{ReLU}(\text{BatchNorm}(x))$
- $\text{Dropout}(\text{ReLU}(x)) \equiv \text{ReLU}(\text{Dropout}(x))$
- $\text{MaxPool}(\text{ReLU}(x)) \equiv \text{ReLU}(\text{MaxPool}(x))$
- $\text{ReLU}(\text{Sigmoid}(x)) \equiv \text{Sigmoid}(\text{ReLU}(x))$

1.6 When you are using a deep CNN to train a semantic segmentation model, which of the following can be chosen to help with overfitting issues?

- Decrease the weight decay parameter
- Increase the probability of switching off neurons in dropout
- Apply random Gaussian noise to the input images
- Remove parts of the validation set

1.7 In terms of (full-batch) gradient descent (GD) and (mini-batch) stochastic gradient descent (SGD), which of the following statements are true?

- The computed gradient of the loss w.r.t model parameters in SGD is equal to the computed gradient in GD
- The expected gradient of the loss w.r.t model parameters in SGD is equal to the expected gradient in GD
- There exists some batch size, for which the gradient of the loss w.r.t model parameters in SGD is equal to the gradient in GD
- SGD and GD will converge to the same model parameters, but SGD requires less memory at the expense of more iterations

1.8 What is true about batch normalization assuming your train and test set are sampled from the same distribution?

- Batch normalization cannot not be used together with dropout
- Batch normalization makes the gradients more stable, so we can train deeper networks
- At test time, Batch normalization uses a mean and variance computed on test set samples to normalize the data
- Batch normalization has learnable parameters

1.9 What is true for common architectures like VGG-16 or LeNet? (check all that apply)

- The number of filters tends to increase as we go deeper into the network
- The width and height of the activation maps tends to increase as we go deeper into the network
- The input can be an image of any size as long as its width and height are equal
- They follow the paradigm: Conv  $\rightarrow$  Pool ...  $\rightarrow$  Conv  $\rightarrow$  Pool  $\rightarrow$  FC ...  $\rightarrow$  FC  
(Conv = Conv + activation)

## Problem 2 Short Questions (18 credits)

0 In  $k$ -fold cross validation, choosing a larger value for  $k$  increases our confidence in the validation score. What could be a practical disadvantage in doing so? Explain how it arises.

1 (1p) Increases training time or more computations. (1p) Making use of more folds, will present the model with more data to train on, but will require way more time as it has to train and validate  $K$  separate times. (0p) Overfitting. (0p) High variance. (1p) Less data in validation set.

2.2 Consider the activation function  $f : \mathbb{R} \rightarrow \mathbb{R}$  and  $f(x) = \ln(1 + e^x)$ .

Which one of the following activation functions is most closely approximated by  $f$ ? Briefly justify your answer (2 points). What is the benefit of  $f$  over the activation function it closely approximates (2 points)?

- Tanh
- ReLU
- Sigmoid

(1p) ReLU. (1p) ReLU is the only function which is unbounded up or any other valid explanation why ReLU or correct drawing of softplus showing similarity to ReLU. (0p) Positive output since it is also valid for Sigmoid.

One of both benefits is sufficient: (2p) Unlike ReLU, this function (softplus) is smooth everywhere in  $\mathbb{R}$ , so its differentiable everywhere in  $\mathbb{R}$ . ReLU is not differentiable at 0. (2p) Softplus does not have a dead area for negative inputs or any explanation related to dead ReLU.

(0p) If not ReLU.

2.3 Explain the difference between the validation set and the test set. In your answer, explain the role of each subset and how they are used differently.

Validation set is used for testing **generalization** (0.5p) with different **hyperparameters/ hyperparameter tuning** (0.5p). Test set is only used **at the end/ Not touched during training** (0.5p) to test generalization **on unseen data once** (0.5p). For each missing keyword -0.5p.

2.4 You notice vanishing/exploding gradients in a deep network using the tanh activation function. Suggest two possible changes you can make to the network in order to diminish this issue, without changing the number of trainable parameters. Explain how each of these changes helps.

(0.5p) For naming a correct change. (0.5p) For correct explanation. (1p) Use ReLU activation, does not saturate, large consistent gradients. (1p) Add residual connections, highway for gradient flow, can learn to skip layers. (1p) Xavier initialization, improved weight initialization targets active area of the activation function. (0p) Gradient clipping (does not resolve vanishing gradients). (0p) BatchNorm. (0p) Regularization.

2.5 Can two consecutive dropout layers with probabilities  $q$  and  $p$  be replaced with one dropout operation? Explain.

(1p) Yes. (0p) No. (1p) Correct explanation. Neurons zero out independently, so dropout layer with probability  $p + q - 2pq$ . If dropout layer one already zeros out neurons those can't be considered by second layer anymore. Droping probability of  $pq$ .

0  
1  
2

2.6 Can one encounter overfitting in an unsupervised learning setting? If your answer is *no*, provide a mathematical reasoning. If your answer is *yes*, provide an example.

(1p) Yes. (0p) No. (2p) Valid example: clustering N datapoints with N clusters ( $k$ -means with  $k = N$ ), autoencoder with large bottleneck/ overfitted on one image, PCA with many components. (0.5p or 1p) For mentioning an unsupervised algorithm with vague explanation.

0  
1  
2  
3

2.7 For each of the following functions, describe one common problem when choosing them as the activation function for your deep neural network: (a) Sigmoid, (b) ReLU, (c) Identity

(1p) Sigmoid (not zero-centered or saturates), (1p) Identity (does not introduce non-linearity), (1p) ReLU (dead ReLU or not zero-centered)

0  
1  
2  
3

### Problem 3 Autoencoder (11 credits)

Consider a given **unlabeled** image dataset consisting of 10 distinct classes of animals.

0 To train an Autoencoder on images, which type of losses you would use? Name two suitable losses.

1 (1p each) Image reconstruction losses: L1, L2, SSIM, PSNR, MSE .... (-1p) For L2 together with  
2 MSE. (0p) For CE, Hinge, BCE, KL-divergence First two named losses count.

0 Explain the effect of choosing a bottleneck dimension which is too small, and the effect of a too large  
1 bottleneck dimension in Autoencoders.

2 (1p) Bottleneck dimension too small leads to poor reconstruction/underfitting, or loss of important  
information/ too much compression. (1p) Bottleneck dimension too big leads to no compression/overfitting/learning identity.

0 Having trained an Autoencoder on this dataset, how would you use the trained Autoencoder (without  
1 further training/fine-tuning) to partition the dataset into 10 subsets, where each subset consists only of  
2 images of a distinct type of animal?

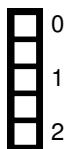
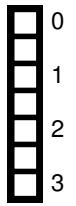
(1p) Use the trained encoder to get latent embedding for each unlabeled image. (1p) Do clustering  
(e.g k-means with  $k = 10$ ). Assign each image to its cluster centre. (0p) Adding FC layers. (0p)  
Using full autoencoder as feature extractor. (0p) Only mentioning clustering.

3.4 We want to use the same network architecture for de-noising and colorizing old, degraded, gray-scale images of animals. Given the dataset you already have, explain the steps you would take to train your model. In your answer, elaborate on your model's inputs, outputs, and losses.

- (1p) Augment input image by adding noise. (1p) Transform input images by converting to grayscale.
- (1p) Use original image as target, use L1/L2 as loss. (0.5p) Name correct loss. (0.5p) Loss between original RGB images and output of the network. (0p) Only loss name. (0p) Proposing another architecture.

3.5 Explain the differences between Autoencoders and Variational Autoencoders. How do they differ during training?

- (2p) Variational autoencoders will constrain the bottleneck distribution into a probability distribution but autoencoders don't constrain the latent space. (1p) Having a constraint in the latent space. (1p) Sampling from the latent space. (0.5p) KL - divergence. (-0.5p) Autoencoders generate images.



## Problem 4 CNNs (10 credits)

You are given the following network that classifies RGB images into one of 4 classes.

All Conv2d layers use `kernel = 3` , `padding = 1` , `stride = 1` , `bias = True` and are defined as `Conv2d(< channelsin >, < channelsout >)` .

All MaxPool2d layers use `stride = 2` , `padding = 0` , and are defined as `MaxPool(< kernel >)` .

The input dimension  $x$  of the Linear layer is unknown.

The network's architecture is as follows:

- `Conv2d(3, 8) → MaxPool2d(2) → BatchNorm2d() → ReLU() →`
- `Conv2d(8, 16) → MaxPool2d(2) → BatchNorm2d() → ReLU() →`
- `Conv2d(16, 32) → MaxPool2d(2) → BatchNorm2d() → ReLU() →`
- `Flatten() →`
- `Linear(  $x$  , 4) → Softmax()`

0

4.1 In terms of  $x$  , what is the total number of trainable parameters of the last linear layer? Include a bias term in your calculation.

1

2

$$4x + 4 = 4(x + 1).$$

(1p) Matrix is shape  $4 \times x$  (1p) plus 4 bias terms.

(-1p) Weight or bias wrong/missing.

0

4.2 Given RGB input images of size  $80 \times 80$  pixels, what should the value of  $x$  in the Linear layer be? Explain your calculation.

1

2

(1p) Each conv2d preserves spatial dimensions. Each maxpool reduces spatial dimensions by 2.

Height and width take shape  $80 / 2 / 2 / 2 = 10$  at linear layer. Depth is 32 as given by final conv2d.

$$(1p) x = 10 \times 10 \times 32 = 3200$$

(0.5p) Same convolution. (0.5p) Maxpool halves spatial dimension. (0.5p) Correct concept: channels x input x output.

4.3 Explain the main difference between the usage of a BatchNorm layer in a convolutional network in comparison to a fully connected network.

(2p) Normalization acts on channel dimension instead of per feature/different channels normalization/statistics. (1p) Only CNN or only FC. (0p) Over batch/all samples. (0p) Normalize weights. (0p) Normalize each pixel. (0p) Normalize input data.

0  
1  
2

4.4 Compute the total number of trainable parameters of the first convolutional layer, Conv2d(3,8).

$$(2p) 3 \times 3 \times 3 \times 8 + 8 = 216 + 8 = 224$$

$$k \times k \times \text{channels}_{in} \times N_{filters} + bias$$

(0.5p) Weights wrong, bias correct. (-1p) Bias missing/wrong. (-0.5p) Correct answer, additionally specified batchnorm.

0  
1  
2

4.5 Compute the total number of trainable parameters in all of the BatchNorm layers.

(2p) Each BatchNorm2d layer has two weights per channel. The number of channels it has is given by the output of the preceding conv2d layer. Therefore # trainable BatchNorm weights =  $2 * 8 + 2 * 16 + 2 * 32 = 2 * (56) = 112$  weights. (0.5p) Only 2 + 2 + 2 without channels. (1.5p) Correct expression, final answer wrong.

0  
1  
2

## Problem 5 Optimization and Gradients (16 credits)

You are training a large fully-connected neural network and select as an initial choice an SGD optimizer. In order to overcome the limitations of SGD, your colleague suggests adding momentum.

- 0  5.1 Name two limitations of SGD that momentum can potentially solve. Explain how momentum solves them.

1  
2 1) limitations: slow learning / small steps, Can't escape local minima, SGD is noisy, SGD only has one lr for all dimensions (1p each, 2p max)  
3 2) explanation: speeds up learning if gradient keeps pointing in the **same direction**, keeps direction of gradient to get out of local minimum, adjusts lr down if oscillating over local minimum, exponentially weighted moving average reduces noise (0.5p each, 1p max)

- 0  5.2 One can apply momentum, as shown in the formula:

$$\nu^{k+1} = \beta \cdot \nu^k - \alpha \cdot \nabla_{\theta} L(\theta^k)$$

1  
2 What do the hyperparameters  $\alpha$  and  $\beta$  represent?

1 in alpha = learning rate (1pt) beta = accumulation rate of velocity/friction (1pt) momentum (0.5pt),  
only accumulation rate (0.5pt)

- 0  5.3 How does Nesterov Momentum differ from standard momentum? Explain.

1  
2 Demonstrates understanding of Nesterov momentum but no further insights (only 0.5pt). A step in direction of previous momentum/accumulated gradient (only gradient is not enough) (1pt). Gradient term computed from position calculated with previous gradient i.e. look ahead step (1pt). Gradient corrects potential overshooting of momentum already in the same step (1pt).  
**Common mistakes:** formulas without explaining them, not mentioning that the “jump” is calculated using accumulated gradients / previous momentum

- 0  5.4 Is RMSProp considered a first or second order method (1p)? What is the main difference between RMSProp and SGD+Momentum?

1  
2 First order (1pt) **Explanation:** RMSProp dampens oscillation /exponentially decaying average of variance/ uses second moment (1pt) SGD + Momentum accumulates gradient / uses first moment (1pt)

For the following questions, consider the convex optimization objective:

$$\min_{x \in \mathbb{R}} x^2$$

5.5 What is the optimal solution of this optimization problem?

x\* = 0 (1p)

0  
 1

5.6 You are working with an initialization of  $x_0 = 5$  and a learning rate of  $\text{lr} = 1$ . How many iterations would gradient descent (without momentum) need in order to converge to the optimal solution? Explain.

Won't converge/ infinite iterations (0.5pt). Explanation as overshoot/ oscillate (0.5pt)

0  
 1

5.7 Assuming you instead start with a random initialization of  $x_0$ , how could you speed up the convergence of the gradient descent optimizer (without adding momentum) in this case?

Reduce lr/ adaptive lr/ dynamic lr/ any form of lr decay or gradient clipping or line search to get lr (1p). only change lr/ adjust lr/ play with lr/ suitable lr (0.5p)

**Common mistakes:** xavier initialization, second order method, adam

0  
 1

5.8 What is the main advantage of using a second order method such as Newton's Method? Why are second order methods not used often in practice for training deep neural networks?

Advantages: less iterations (1pt) if only mentioned "converge faster" without specifying in terms of iterations (0.5pt), only 1 step (0.5p), no need to choose learning rate.

Drawbacks: Hessian costly to compute, Second order methods don't work well with mini-batches (1pt each; 1pt max)

0  
 1  
 2

5.9 How many iterations would Newton's method need to converge (using the same initialization  $x_0 = 5$ ,  $\text{lr} = 1$ )? Explain.

Only takes 1 iteration(0.5pt). Jumps to minimum right away / convex problem / 2nd order taylor approximation exactly approximate quadratic problem / calculation that it converges after one step (0.5pt).

**Common mistakes:** uses second derivative, uses hessian instead of lr

0  
 1

## Problem 6 Derivatives (9 credits)

Consider the formula of the Sigmoid function  $\sigma(x) : \mathbb{R} \rightarrow \mathbb{R}$  :

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

0  6.1 Compute the derivative  $\frac{d\sigma(x)}{dx}$  in terms of  $x$ .

1

$$\frac{d\sigma}{dx} = \frac{0 \cdot (1 + e^{-x}) - 1 \cdot (-e^{-x})}{(1 + e^{-x})^2} = \frac{e^{-x}}{(1 + e^{-x})^2}$$

correct intermediate step (0.5p) & correct final answer (1p)

0  6.2 A special property of this function is that its derivative can be expressed in terms of the Sigmoid function itself. Denote  $y = \sigma(x)$ , and show how the derivative you computed can be re-written in terms of  $y$ , the output of the Sigmoid function. Hint: Your answer should only depend on  $y$ .

$$\begin{aligned}\frac{dy}{dx} &= y(1 - y) \\ y &= \frac{1}{(1 + e^{-x})} \\ 1 - y &= \frac{1 + e^{-x}}{(1 + e^{-x})} - \frac{1}{(1 + e^{-x})} = \frac{e^{-x}}{(1 + e^{-x})}\end{aligned}$$

Final correct answer (1p). Wrong answer with some correct derivation (0.5p).

An affine Layer is described by  $\mathbf{z} = \mathbf{XW} + \mathbf{b}$ .

Consider the following affine layer, which has 2 input neurons and 1 output neuron:

$$\mathbf{W} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}_{2 \times 1}$$

$$\mathbf{b} = 2 \in \mathbb{R}^1$$

and input:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}_{2 \times 2}$$

The forward pass of the network would be:

$$\sigma(\mathbf{z}) = \sigma(\mathbf{XW} + \mathbf{b}) = \sigma\left(\begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + 2\right) = \sigma\left(\begin{bmatrix} 3 \\ -2 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} 5 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \text{ (rounded up).}$$

Let's compute the backward pass of the network.

6.3 If  $\mathbf{y} = \sigma(\mathbf{z}) = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}$ , calculate the gradient of the output after the Sigmoid activation function w.r.t  $\mathbf{z}$ ,  $\frac{dy}{dz}$ :

$$\frac{\partial y}{\partial z} = \mathbf{y} \circ (1 - \mathbf{y})$$

$$\begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \circ (1 - \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}) = \begin{bmatrix} 0 \\ 0.25 \end{bmatrix}$$

writing the derivative correctly (element-wise multiplication and NOT matrix multiplication) (1p), correct intermediate calculation (dimensions are correct) (1p), correct final answer (1p)

6.4 We will use the computed gradient to perform back-propagation through the affine layer to the network's parameters.

Let  $dout$  be the upstream derivative of the Sigmoid that you have calculated in question 3. Calculate the derivatives  $\frac{dy}{dW}$  and  $\frac{dy}{db}$ .

*Hint:* Pay attention to the shapes of the results; they should be compatible for a gradient update.

*Note:* In case you skipped the previous question, you can get partial points by writing the correct formulas using  $dout$  symbolically.

$$dW = \mathbf{X}^T \cdot dout = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0.25 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.25 \end{bmatrix}$$

$$db \rightarrow \text{sum}(dout, \text{axis} = 0) = [1 \ 1] \begin{bmatrix} 0 \\ 0.25 \end{bmatrix} = 0.25$$

$dW$ :(2p)  $db$ :(2p). For case, chain rule (0.5p) writing the matrices correctly (e.g  $\mathbf{X}_T * dout$ ) (1p), correct answer (0.5p). if missed the correct answer by 1/n (-0.5p)

0  
1  
2  
3

0  
1  
2  
3  
4

## Problem 7 Model Evaluation (8 credits)

Two students, *Erika* and *Max* train a neural network for the task of image classification. They use a dataset which is divided into train and validation sets. They each train their own network for 25 epochs.

- 0 7.1 Erika selects a model and obtains the following curves. Interpret the model's behaviour from the curves. Then, suggest what could Erika do in order to improve its performance?

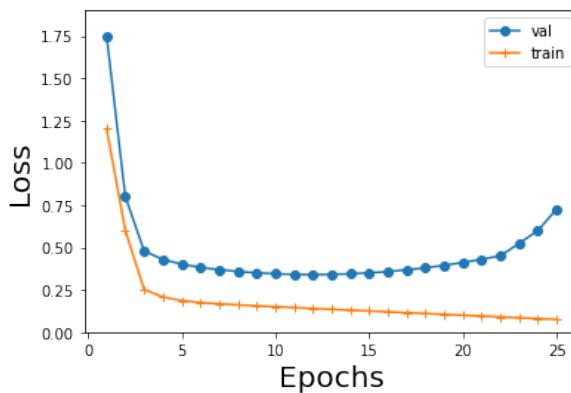


Figure 7.1: Training curves for Erika's model.

(0.5p) Overfitting , (0.5p) regularization (Dropout, L1, L2 weight decay, data augmentation), early stopping and reducing capacity.

**Common mistakes:** Stop training early. No mention of stopping training early based on validation error.

- 0 1 2 7.2 Max selects a different model and obtains the following curves. Interpret the model's behaviour from the curves. Then, suggest what change could Max make to his model in order to improve its performance?

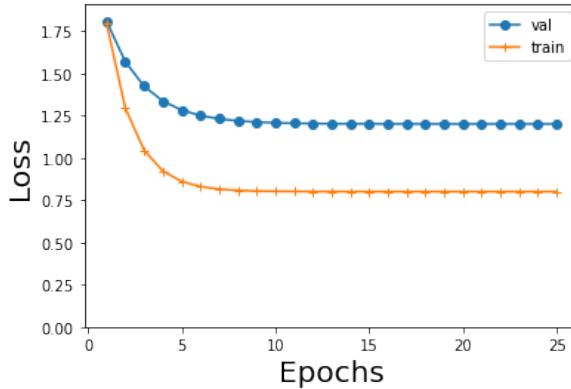


Figure 7.2: Training curves for Max's model.

Underfitting (0.5p), Increase model capacity (1.5p). OR Optimization is not optimal (0.5p), Decrease learning rate / learning rate decay / use optimizer that corrects a bad learning rate choice (e.g Adam) / BN (1.5p)

**Common mistakes:** Just describing what the graphs do. Generalization gap, add regularization

7.3 Both Max and Erika are able to agree on a model architecture and obtain the following curves. However, when deployed in real world, their model seems to perform poorly. What is a possible reason for such an observation and what should they do?

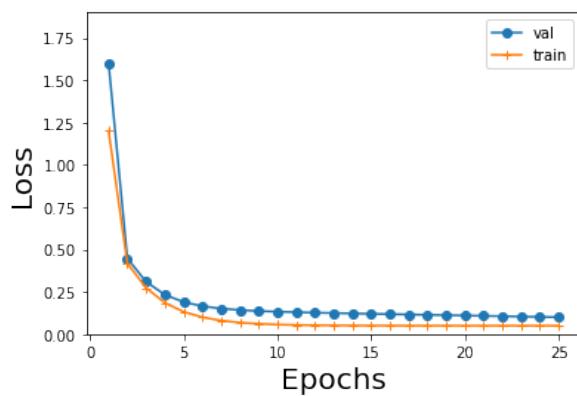


Figure 7.3: Training curves for the new model.

Possible reasoning: test and train/val data is sampled from different distributions / domain gap.(1p)  
Fix by trying to make test and train data more similar or from same distribution / augmentation / add another dataset to train / (1p).

**Common mistakes:** change the test set. Shuffle train and val dataset to train again. If train and val come from the same distribution (given in the question) shuffling will not help. Overfitting to “val” data.

After adapting the new network architecture, Max and Erika are training their own model, using the same architecture, with identical initial weights, using exactly the same hyperparameters. They also use the same SGD optimizer (no momentum), batch size, and learning rates. The only difference is that Max normalizes the loss by  $1/N$  (where  $N$  is the number of training samples in the dataset) while Erika does not.

7.4 How does this affect the optimal model weights that minimize this optimization objective? (1p) After 10 optimizer steps, will they arrive at the same model parameters? Explain.(2p)

It doesn't affect the optimal model's optima (1p) The weights will be different (0.5p) after 10 steps (0.5p). A good explanation why (Ir is scaled) (1p) Contradictory / Unclear explanation (-0.5p)

**Common mistakes:**  $1/N$  would make it independent of the size of the dataset. Irrelevant.

**Additional space for solutions—clearly mark the (sub)problem your answers are related to and strike out invalid solutions.**

A large grid of squares, approximately 20 columns by 30 rows, intended for students to write their solutions. The grid is composed of thin black lines on a white background.

**Note:**



- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

# Introduction to Deep Learning

**Exam:** IN2346 / Endterm

**Date:** Tuesday 13<sup>th</sup> July, 2021

**Examiner:** Prof. Dr. Matthias Nießner

**Time:** 17:30 – 19:00

## Working instructions

- This exam consists of **16 pages** with a total of **5 problems**.  
Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 91 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources: None
- Do not write with red or green colors

## Problem 1 Multiple Choice (18 credits)

Below you can see how you can answer multiple choice questions.

*Mark correct answers with a cross*



*To undo a cross, completely fill out the answer option*



*To re-mark an option, use a human-readable marking*



- For all multiple choice questions any number of answers, i.e. either zero (!), one or multiple answers can be correct.
- For each question, you'll receive 2 points if all boxes are answered correctly (i.e. correct answers are checked, wrong answers are not checked) and 0 otherwise.

1.1 Which of the following models are unsupervised learning methods?

- Auto-Encoder
- Maximum Likelihood Estimate
- K-means Clustering
- Linear regression

1.2 In which cases would you usually reduce the learning rate when training a neural network?

- When the training loss stops decreasing
- To reduce memory consumption
- After increasing the mini-batch size
- After reducing the mini-batch size

1.3 Which techniques will typically decrease your **training** loss?

- Add additional training data
- Remove data augmentation
- Add batch normalization
- Add dropout

1.4 Which techniques will typically decrease your **validation** loss?

- Add dropout
- Add additional training data
- Remove data augmentation
- Use ReLU activations instead of LeakyReLU

1.5 Which of the following are affected by multiplying the loss function by a constant positive value when using SGD?

- Memory consumption during training
- Magnitude of the gradient step
- Location of minima
- Number of mini-batches per epoch

1.6 Which of the following functions are not suitable as activation functions to add non-linearity to a network?

- $\sin(x)$
- $\text{ReLU}(x) - \text{ReLU}(-x)$
- $\log(\text{ReLU}(x) + 1)$
- $\log(\text{ReLU}(x + 1))$

1.7 Which of the following introduce non-linearity in the neural network?

- LeakyReLU with  $\alpha = 0$
- Convolution
- MaxPool
- Skip connection

1.8 Compared to the L1 loss, the L2 loss...

- is robust to outliers
- is costly to compute
- has a different optimum
- will lead to sparser solutions

1.9 Which of the following datasets are NOT i.i.d. (independent and identically distributed)?

- A sequence (toss number, result) of 10,000 coin flips using biased coins with  $p(\text{toss result} = 1) = 0.7$
- A set of (image, label) pairs where each image is a frame in a video and each label indicates whether that frame contains humans.
- A monthly sample of Munich's population over the past 100 years
- A set of (image, number) pairs where each image is a chest X-ray of a different human and each number represents the volume of their lungs.

## Problem 2 Short Questions (29 credits)

0  
1  
2  
3

2.1 Explain the idea of data augmentation (1p). Specify 4 different data augmentation techniques you can apply on a dataset of RGB images (2p).

Improve generalization by adding more data and preventing overfitting (1p)  
Rotation, cropping, color jittering, salt/paper, flipping, translation jitter ... (0.5p for each)

0.5 -> make training set larger  
1.0 -> generalization/prevent overfitting

0  
1  
2

2.2 You are training a deep neural network for the task of binary classification using the Binary Cross Entropy loss. What is the expected loss value for the first mini-batch with batch size  $N = 64$  for an untrained, randomly initialized network? Hint:  $BCE = -\frac{1}{N} \sum (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i))$

-log(0.5) or log(2)

-0.5 -> for 1/64  
-0.5 -> for minus

0  
1  
2

2.3 Explain the differences between *ReLU*, *LeakyReLU* and *Parametric ReLU*.

ReLU: constant 0 for negative values (0.5p) LeakyReLU: pre-defined slope for negative values (0.5p) Parametric ReLU: learnable value for slope, either 1 for all channels or 1 for each channels. (1p)

for relu and leaky relu -> full points to formula / drawing  
for parametric relu -> learnable slope

0  
1  
2

2.4 How will weights be initialized by Xavier initialization? Which mean and variance will the weights have? Which mean and variance will the output data have?

With Xavier initialization we initialize the weights to be Gaussian with zero mean and variance  $\text{Var}(w) = 1/n$  where n is the amount of neurons in the input.  
As a result, the output will have zero mean, and similar variance as the input

weights  
0.5 -> zero mean(with mentioning Gaussian)  
0.5 -> variance  
output  
0.5->mean  
0.5-> variance(same/similar)

2.5 Why do we often refer to L2-regularization as “weight decay”? Derive a mathematical expression that includes the weights  $W$ , the learning rate  $\eta$ , and the L2 regularization hyperparameter  $\lambda$  to explain your point.

$Reg = 0.5 \cdot \lambda \cdot \|W\|^2$

Upon a gradient update:

$$W_{new} = W - \eta \cdot \nabla Reg = W - \eta \cdot \lambda W = (1 - \eta \cdot \lambda) W$$

0.5/3.0 -> no formula, just correct explanation

1.0 -> formula of regularization

1.0 -> for gradient, inserting reg

1.0 -> weight decay

0
1
2
3

2.6 Given a Convolution Layer in a network with 6 filters, kernel size 5, a stride of 3, and a padding of 2. For an input feature map of shape  $28 \times 28 \times 28$ , what are the dimensions/shape of the output tensor after applying the Convolution Layer to the input?

Output width/height =  $(28 + 2 * 2 - 5) / 3 + 1 = 10$  (1 pt, it's ok if they do not have the calculation).

Output shape of channels x height x width =  $6 \times 10 \times 10$  or  $10 \times 10 \times 6$  (1 pt).

0.5 -> correct formula and wrong calculation

1.0 -> output shape

0
1
2
3

2.7 You are given a Convolutional Layer with: number of input channels 3, number of filters 5, kernel size 4, stride 2, padding 1. What is the total number of trainable parameters for this layer? Don't forget to consider the bias.

$(3 \times (4 \times 4)) \times 5$  for weights + 5 for bias =  $240 + 5 = 245$

(1pt for weights without correct bias)

1.0 for each correct calculation

2.0 for correct number(245)

1.5 -> wrong addition

0
1
2
3

2.8 You are given a fully-connected network with 2 hidden layers, the first of has 10 neurons, and the second hidden layer contains 5 neurons. Both layers use dropout with probability 0.5. The network classifies gray-scale images of size  $8 \times 8$  pixels as one of 3 different classes. All neurons include a bias. Calculate the total number of trainable parameters in this network.

Weights:  $(8 \times 8) \times 10 + 10 \times 5 + 5 \times 3 = 705$

Biases:  $10 + 5 + 3 = 18$

Total:  $705 + 18 = 723$

1 p -> weights

1 p -> bias

1.5 -> wrong addition

0
1
2
3

0  
1  
2

2.9 "Breaking the symmetry": Why is initializing all weights of a fully-connected layer to the same value problematic?

All neurons will learn the same thing / Gradient update will be the same for all neurons / they won't take on different values.

2p-> mentioning they all compute the same function/learn the same thing/ same gradient update

0  
1  
2

2.10 Explain the difference between *Auto-Encoders* and *Variational Auto-Encoders*.

A variational Autoencoder imposes (optional: Gaussian / KL-Divergence loss) constraints on the distribution of the bottleneck

no points for explanation of autoencoder  
2p -> constraint on latent space/distribution

0  
1  
2

2.11 Generative Adversarial Networks (GANs): What is the input to the generator network (1 pt)? What are the two inputs to the discriminator (1 pt)?

Generator: The input is a random noise vector (1 pt). [Wrong: labels] Discriminator: The inputs to the Discriminator are fake/generated images (0.5 pt) and real images (0.5 pt).

random input is fine, not mentioning fake is fine

0  
1  
2

2.12 Explain how LSTM networks often outperform traditional RNNs. What in their architecture enables this?

It is difficult for traditional RNNs to learn long-term dependencies due to vanishing gradients (1p).

The cell state (1p) in LSTMs improve the gradient flow and thereby allows the network to learn longer dependencies.

0.5 -> vanishing gradient  
0.5 p -> long term dep  
0.5 p -> highway for gradient/ improved gradient flow  
1.0-> cell state

0  
1  
2  
3

2.13 Explain how batch normalization is applied differently between a fully connected layer and a convolutional layer (1 pt). How many learnable parameters does batch normalization contain following (a) a single fully-connected layer (1 pt), and (b) a single convolutional layer with 16 filters (1 pt)?

Mini-batch is normalized over all neurons in a fully connected layer, while it is normalized over each channel in a convolutional layer (1 pt).

2 for a fully-connected layer (1 pt)

$2 \times 16 = 32$  for a convolutional layer (1 pt).

for the first point -> 0.5 p for the first part 0.5 for the second

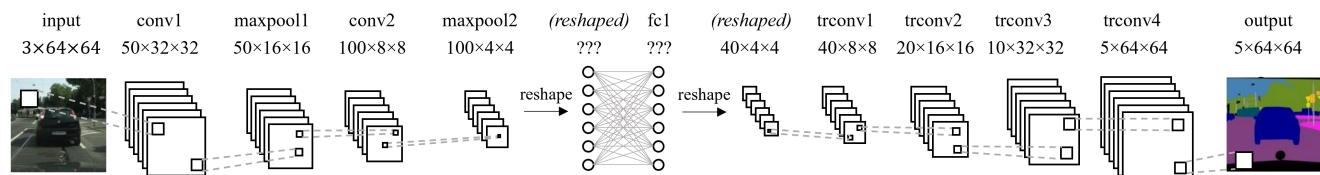
### Problem 3 Convolutions (13 credits)

You are asked to perform **per-pixel** semantic segmentation on the Cityscapes dataset, which consists of RGB images of European city streets, and you want to segment the images into 5 classes (vehicle, road, sky, nature, other). You have designed the following network, as seen in the illustration below:

For clarification of notation: The shape **after** having applied the operation ‘conv1’ (the first convolutional layer in the network) is  $50 \times 32 \times 32$ .

You are using 2D convolutions with: `stride = 2`, `padding = 1`, and `kernel_size = 4` .

For the MaxPool operation, you are using: `stride = 2`, `padding = 0`, and `kernel_size = 2` .



3.1 What is the shape of the weight matrix of the fully-connected layer ‘fc1’? (Ignore the bias)

input:  $100 \times 4 \times 4 = 1600$

output:  $40 \times 4 \times 4 = 640$

weight matrix:  $1600 \times 640$

0  
1  
2

3.2 Explain the term ‘receptive field’ (1p). What is the receptive field of one pixel of the activation map. after performing the operation ‘maxpool1’(1p)? What is the receptive field of a single neuron in the output of layer ‘fc1’ (1p)?

the size of the region in the input space that a pixel in the output space is affected by.

maxpool1:  $6 \times 6$ . One pixel after maxpool1 is affected by 4 pixels ( $2 \times 2$ ) in conv1. with  $4 \times 4$  kernel and stride 2, a  $2 \times 2$  output comes from a  $6 \times 6$  grid.

f1: whole image ( $64 \times 64$ ) (accept answer that takes into account padding)

0  
1  
2  
3

0 3.3 You now want to be able to classify finer-grained labels, which comprise of 30 classes. What is the **minimal** change in network architecture needed in order to support this without adding any additional layers?

1in - change output channels of trconv4 to 30 - NO: add 1x1 conv (with 30 output channels)  
- NO: or any conv that preserves the size (with 30 output channels)

0 3.4 Luckily, you found a pre-trained version of this network, which is trained on the original 5 labels. (It outputs a tensor of shape  $5 \times 64 \times 64$ ). How can you make use of/build upon this pre-trained network (as a black-box) to perform segmentation into 30 classes.

1 - add 1x1 conv at the end (with 30 output channels) - or any conv that preserves the size (with 30 output channels)

2 3.5 Luckily, you have gained access to a large dataset of city street images. Unfortunately, these images are not labelled, and you do not have the resources to annotate them. However, how can you still make use of these images to improve your network? Explain the architecture of any networks that you will use and explain how training will be performed. (Note: This question is independent of (3.3) and (3.4))

3 transfer learning, pre-train an AutoEncoder with the unlabeled / all images, use encoder or entire network (except last layer/layers) to initialize the segmentation network. Freeze (some) weights, change/add last layer to output segmentation.

0 3.6 Instead of taking  $64 \times 64$  images as input, you now want to be able to train the network to segment images of arbitrary size  $> 64$ . List, explicitly, two different approaches that would allow this. Your new network should support varying image sizes in run-time, without having to be re-trained.

- Resize layer/operation to downsample images to 64x64 (e.g bilinear)
- Make it fully convolutional by replacing the FC layer with convolutions

Wrong: explanation with RNNs (eigenvalues etc)

## Problem 4 Optimization (13 credits)

4.1 Explain the idea behind the RMSProp optimizer. How does it enable faster convergence than standard SGD? How does it make use of the gradient?

RMSProp is an adaptive learning rate method - It scales the learning rate (1 pt) based on (an exponentially decaying average of) magnitude/element-wise squared gradient. This enables faster convergence by e.g skipping through saddle points with high learning rates. Also possible arguments from the lecture:

- Dampening the oscillations for high-variance directions
- Can use faster learning rate because it is less likely to diverge: Speed up learning speed, Second moment RMSProp does not have momentum!

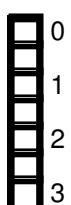
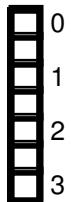
4.2 What is the *bias correction* in the ADAM optimizer? Explain the problem that it fixes.

When accumulating gradients in a weighted average fashion, the first gradient is initialized to zero. This biases all the accumulated gradients down towards zero. The Bias correction normalizes the magnitude of the accumulated gradient for early steps.

4.3 You read that when training deeper networks, you may suffer from the *vanishing gradients* problem. Explain what are vanishing gradients in the context of deep convolutional networks and the underlying cause of the problem.

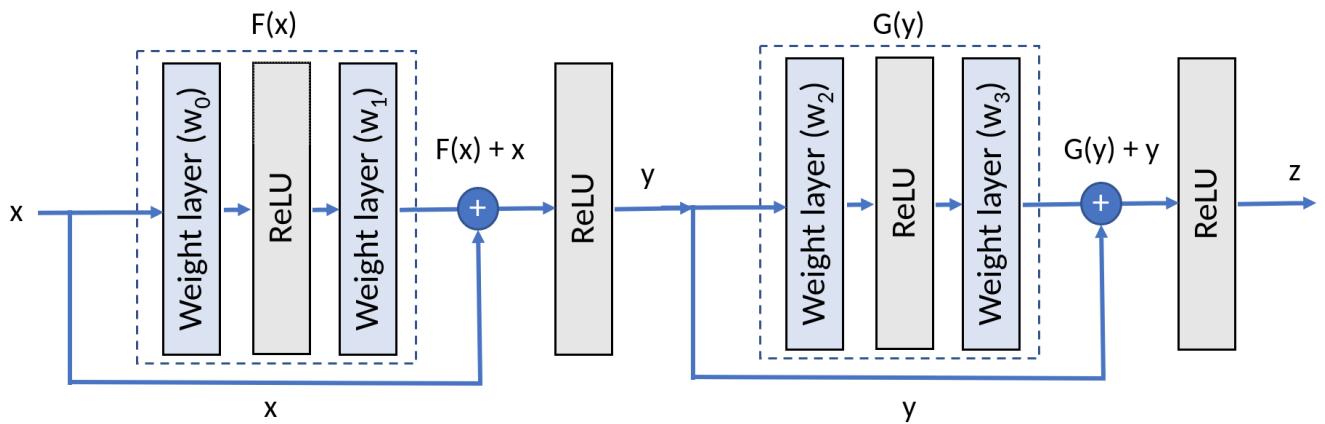
vanishing gradients are gradients with a very small magnitude (causing a meaningless update step) (1 pt) Caused by:

- saturated activations (e.g., tanh)
- chain rule - many multiplications on numbers in (0, 1) goes to 0. during backpropagation, the gradient of early layers (layers near to the input layer) are obtained by multiplying the gradients of later layers



0  
1  
2  
3  
4  
5

4.4 In the following image you can see a segment of a very deep architecture that uses residual connections. How are residual connections helpful against vanishing gradients? Demonstrate this mathematically by performing a weight update for  $w_0$ . Make sure to explain how this reduces the effect of vanishing gradients. Hint: Write the mathematical expression for  $\frac{\partial z}{\partial w_0}$  w.r.t all other weights.



Let,  $\tilde{z} = g(y) + y$   
 $\tilde{y} = F(x) + x$

$$z = \sigma(\tilde{z})$$

$$g(y) = \sigma(w_3 y) w_3$$

$$y = \sigma(\tilde{y})$$

$$F(x) = w_2 \sigma(w_1 x)$$

$$\frac{dz}{dw_0} = \frac{dz}{d\tilde{z}} \frac{d\tilde{z}}{dy} \frac{dy}{d\tilde{y}} \frac{d\tilde{y}}{dw_0}$$

$$\frac{dz}{dw_0} = (\sigma'_z)(\frac{dG(y)}{dy} + 1)(\sigma'_y)(\frac{dF(x)}{dw_0})$$

$$\frac{dz}{dw_0} = (\sigma'_z)(w_3 w_2 \sigma'_{w_2 y})(\sigma'_y)(w_1 x \sigma'_{w_1 x})$$

For ReLU's gradient we,

$$\frac{dz}{dw_0} = (w_3 w_2 + 1) w_1 x \rightarrow \text{gradient update for } w_3 \text{ has } +w_3 x \text{ additional term.}$$

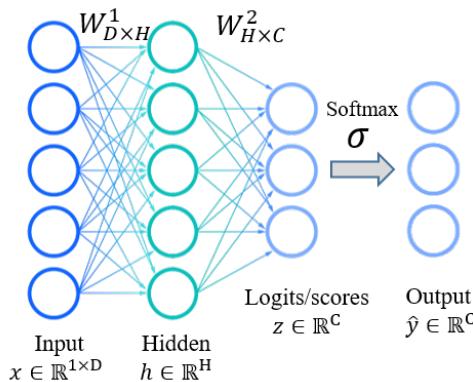
Propagation of gradient

## Problem 5 Multi-Class Classification (18 credits)

**Note:** If you cannot solve a sub-question and need its answer for a calculation in following sub-questions, mark it as such and use a symbolic placeholder (i.e., the mathematical expression you could not explicitly calculate + a note that it is missing from the previous question.)

Assume you are given a labeled dataset  $\{X, y\}$ , where each sample  $x_i$  belongs to one of  $C = 10$  classes. We denote its corresponding label  $y_i \in \{1, \dots, 10\}$ . In addition, you can assume each data sample is a row vector.

You are asked to train a classifier for this classification task, namely, a 2-layer fully-connected network. For a visualization of the setting, refer to the following illustration:



5.1 Why does one use a **Softmax** activation at the end of such a classification network? What property does it have that makes it a common choice for a classification task?

It normalizes the logits/scores to sum up to 1 / a probability distribution  
 Wrong: its derivative can be expressed in terms of the softmax function itself. This is not special for classification, say only output between [0,1] (0 pt)

5.2 For a vector of logits  $\vec{z}$ , the Softmax function  $\sigma : \mathbb{R}^C \rightarrow \mathbb{R}^C$ , is defined:

$$\hat{y}_i = \sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^C e^{z_j}}$$

where  $C$  is the number of classes and  $z_i$  is the  $i$ -th logit.

A special property of this function is that its derivative can be expressed in terms of the Softmax function itself. How could this be advantageous for training neural networks?

calculation of the backward pass is quick, immediate from saving the forward cache

0 5.3 Show explicitly how this can be done, by writing  $\frac{\partial \hat{y}_i}{\partial z_i}$  in terms of  $\hat{y}_i$ .

1  
2  
3

$$\begin{aligned}\frac{\partial}{\partial(z_i)}(\hat{y}_i) &= \frac{e^{z_i} \cdot \sum_j e^{z_j} - e^{z_i} \cdot e^{z_i}}{\left(\sum_j e^{z_j}\right)^2} = \\ &= \frac{e^{z_i} \cdot \left[\left(\sum_j e^{z_j}\right) - e^{z_i}\right]}{\left(\sum_j e^{z_j}\right) \cdot \left(\sum_j e^{z_j}\right)} = \frac{e^{z_i}}{\sum_j e^{z_j}} \cdot \frac{\sum_j e^{z_j} - e^{z_i}}{\sum_j e^{z_j}} = \\ &= \hat{y}_i \cdot \left(\frac{\sum_j e^{z_j}}{\sum_j e^{z_j}} - \frac{e^{z_i}}{\sum_j e^{z_j}}\right) = \hat{y}_i \cdot (1 - \hat{y}_i)\end{aligned}$$

0 5.4 Similarly, show explicitly how this can be done, by writing  $\frac{\partial \hat{y}_i}{\partial z_j}$  in terms of  $\hat{y}_i$  and  $\hat{y}_j$ , for  $i \neq j$ .

1  
2

$$\begin{aligned}\frac{\partial}{\partial(z_j)}(\hat{y}_i) &= \frac{0 \cdot \sum_j e^{z_j} - e^{z_j} \cdot e^{z_i}}{\left(\sum_j e^{z_j}\right)^2} = \\ &= \frac{-e^{z_j} \cdot e^{z_i}}{\left(\sum_j e^{z_j}\right) \cdot \left(\sum_j e^{z_j}\right)} = -\frac{e^{z_i}}{\sum_j e^{z_j}} \cdot \frac{e^{z_j}}{\sum_j e^{z_j}} = -\hat{y}_i \hat{y}_j\end{aligned}$$

5.5 Using the Softmax activation, what loss function  $\mathcal{L}(y, \hat{y})$  would you want to *minimize*, to train a network on such a multi-class classification task? Name this loss function (1 pt), and write down its formula (2 pt), for a single sample  $x$ , in terms of the network's prediction  $\hat{y}$  and its true label  $y$ . Here, you can assume the label  $y \in \{0, 1\}^C$  is a one-hot encoded vector:

$$y_i = \begin{cases} 1, & \text{if } i == \text{true class index} \\ 0, & \text{otherwise} \end{cases}$$

<input type="checkbox"/>	0
<input type="checkbox"/>	1
<input type="checkbox"/>	2
<input type="checkbox"/>	3

(not Binary! (0 pt for binary) Cross Entropy loss / softmax loss (colloquial term) .

$$CE(y, \hat{y}) = - \sum_{j=1}^C y_j \log \hat{y}_j$$

or, since labels are one-hot vectors here:

$$CE(y, \hat{y}) = - \log \hat{y}_j$$

Comments:

- forget minus - lose 0.5 pt
- normalize by  $1/C$  - OK
- formula has another sum over all data - OK

5.6 Having done a forward pass with our sample  $x$ , we will back-propagate through the network. We want to perform a gradient update for the weight  $w_{j,k}^2$  (the weight which is in row  $j$ , column  $k$  of the second weights' matrix  $W^2$ ). First, use the chain rule to write down the derivative  $\frac{\partial \mathcal{L}}{\partial w_{j,k}}$  as a product of 3 partial derivatives (no need to compute them). For convenience, you can ignore the bias and omit the  $^2$  superscript.

<input type="checkbox"/>	0
<input type="checkbox"/>	1
<input type="checkbox"/>	2

First, we write the Chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{j,k}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_{j,k}}$$

0  
1  
2  
3  
4  
5

5.7 Now, compute the gradient for the weight:  $w_{3,1}^2$ . For this, you will need to compute each of the partial derivatives you have written above, and perform the multiplication to get the final answer. You can assume the ground-truth label for the sample was `true_class = 3`. **Hint:** The derivative of the logarithm is  $(\log t)' = \frac{1}{t}$ .

For CE loss, the loss only depends on the prediction of  $\hat{y}_{\text{true}}$ , that is  $\hat{y}_3$  in this case.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_3} = \frac{\partial (-\log \hat{y}_3)}{\partial \hat{y}_3} = -\frac{1}{\hat{y}_3}$$

$\hat{y}_3$  is affected by all of the entries of the vector  $z$ , because of the softmax. Note that  $w_{3,1}$  only affects  $z_1$  ( $z = h \cdot W$ ), and from previous subquestions,

$$\frac{\partial \hat{y}_3}{\partial z_1} = -\hat{y}_3 \hat{y}_1$$

We are only missing  $\frac{\partial z_1}{\partial w_{3,1}}$ . That comes from matrix multiplication.

$$\frac{\partial z_1}{\partial w_{3,1}} = \sum_{k=1}^H h_k w_{k,1}$$

so  $\frac{\partial z_1}{\partial w_{3,1}} = h_3$ .

Finally, combining everything yields:

$$\frac{\partial \mathcal{L}}{\partial w_{1,3}} = -\frac{1}{\hat{y}_3} \cdot -\hat{y}_3 \hat{y}_1 \cdot h_3 = \hat{y}_1 h_3$$

wrong sign (lose 0.5 pt)

**Additional space for solutions—clearly mark the (sub)problem your answers are related to and strike out invalid solutions.**

A large grid of squares, approximately 20 columns by 30 rows, intended for students to write their solutions. The grid is composed of thin black lines on a white background.

