Applied Deep Learning for NLP

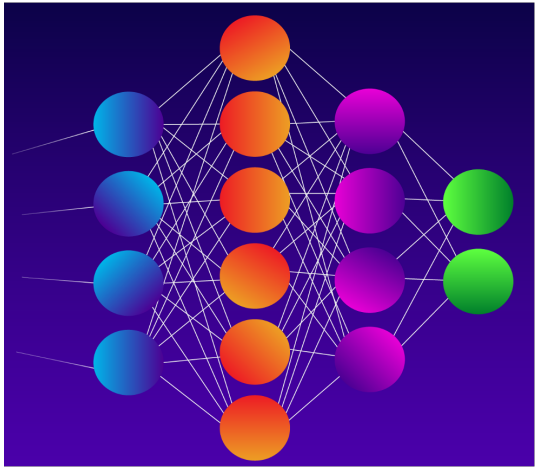
Week 2 - Deep Learning

Juan Carlos Medina Serrano

Technische Universität München
Hochschule für Politik
Political Data Science

Munich, 29. October 2020

political
data
science



Introduction to Deep Learning

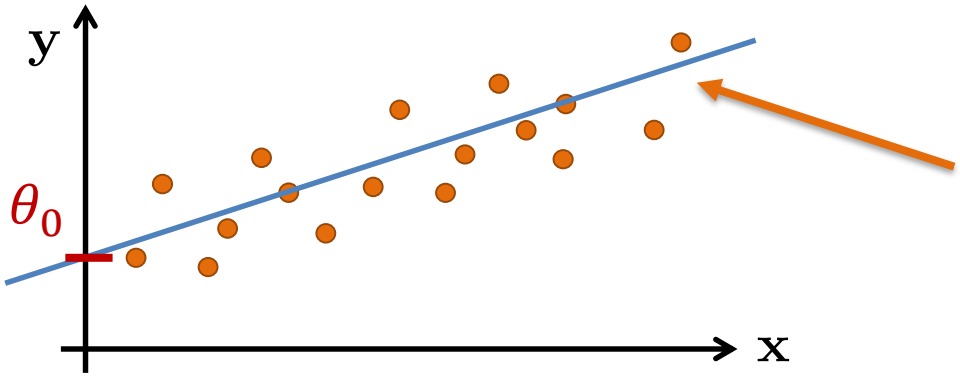
The slides for today are all from the amazing TUM lecture Introduction to Deep Learning.

videos: https://www.youtube.com/watch?v=QLOocPbztuc&list=PLQ8Y4kIIbzy_OaXv86lfbQwPHSomk2o2e&ab_channel=MatthiasNiessner

Linear Regression

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

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



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

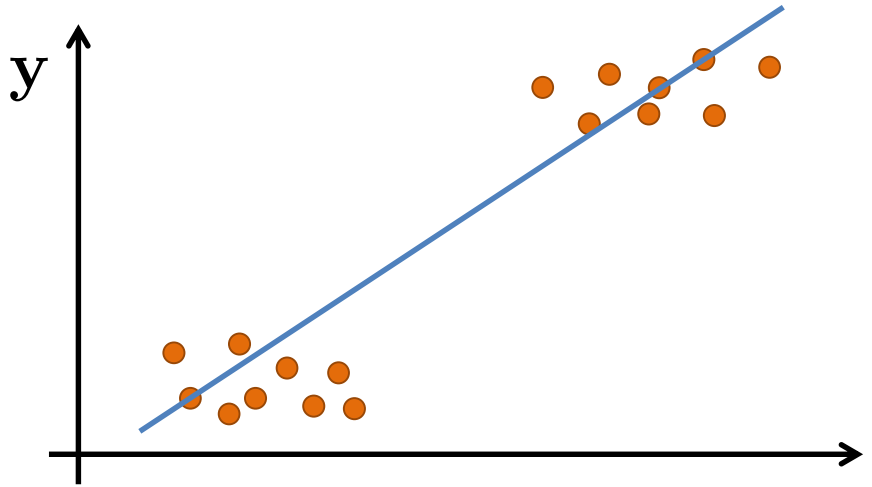
Linear Score Functions

- Linear score function as seen in linear regression

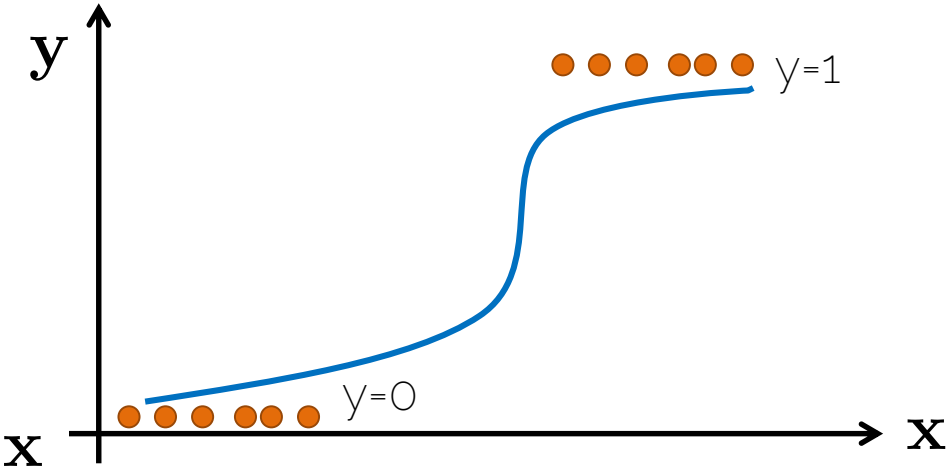
$$\boldsymbol{f_i} = \sum_{\boldsymbol{j}} \boldsymbol{W_{k,j}\, x_{j,i}}$$

$$\boldsymbol{f = W\, x} \qquad \text{(Matrix Notation)}$$

Linear vs Logistic Regression



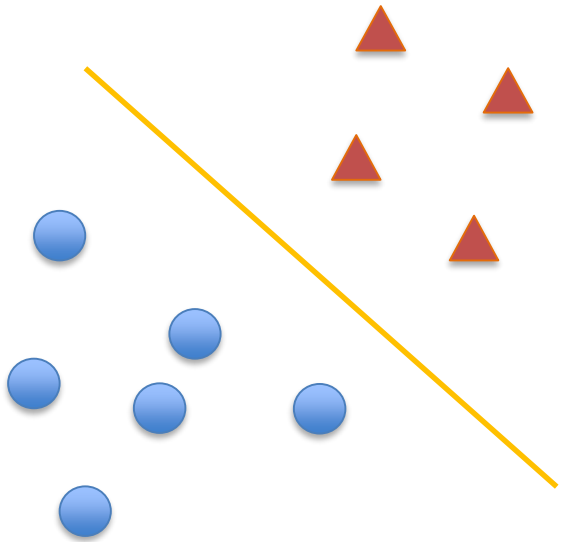
Predictions can exceed the range of the training samples
→ in the case of classification [0;1] this becomes a real issue



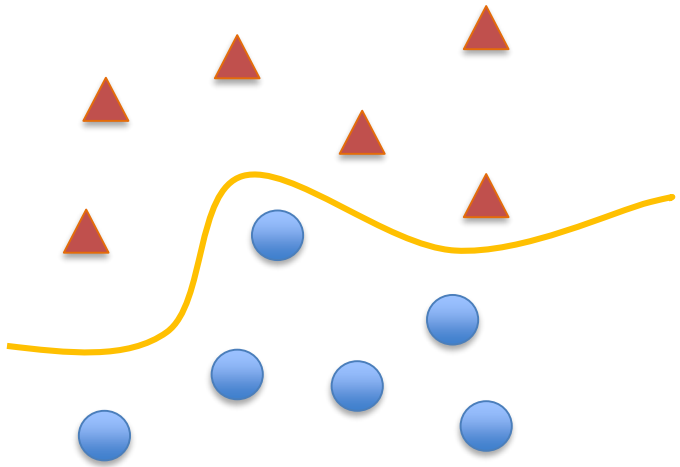
Predictions are guaranteed to be within [0;1]

Linear Score Functions?

Logistic Regression



Linear Separation Impossible!



Linear Score Functions?

- Can we make linear regression better?
  - Multiply with another weight matrix $\boldsymbol{W_2}$

$$\hat{\boldsymbol{f}} = \boldsymbol{W_2} \cdot \boldsymbol{f}$$
$$\hat{\boldsymbol{f}} = \boldsymbol{W_2} \cdot \boldsymbol{W} \cdot \boldsymbol{x}$$

- Operation is still linear.

$$\widehat{\boldsymbol{W}} = \boldsymbol{W_2} \cdot \boldsymbol{W}$$
$$\hat{\boldsymbol{f}} = \widehat{\boldsymbol{W}}\, \boldsymbol{x}$$

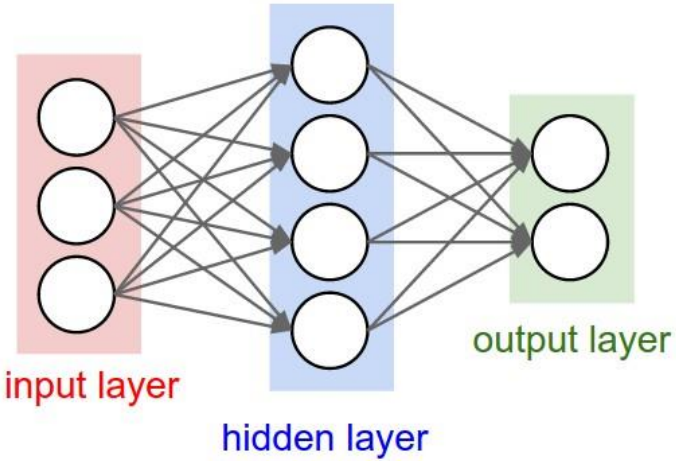
- Solution → add non-linearity!!

Neural Network

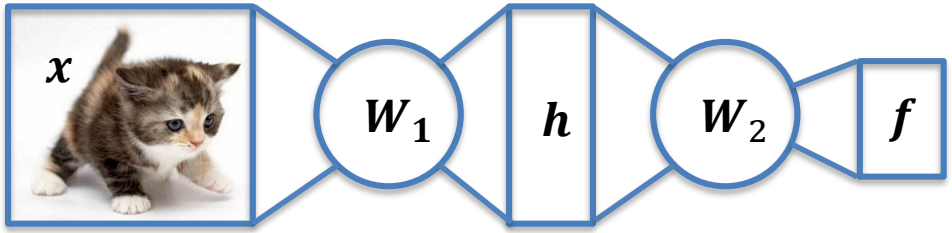
- Linear score function $\boldsymbol{f = Wx}$

- Neural network is a nesting of 'functions'
  - 2-layers: $\boldsymbol{f = W_2 \max(0, W_1 x)}$
  - 3-layers: $\boldsymbol{f = W_3 \max(0, W_2 \max(0, W_1 x))}$
  - 4-layers: $\boldsymbol{f = W_4 \tanh\,(W_3, \max(0, W_2 \max(0, W_1 x)))}$
  - 5-layers: $\boldsymbol{f = W_5 \sigma(W_4 \tanh(W_3, \max(0, W_2 \max(0, W_1 x))))}$
  - … up to hundreds of layers

Neural Network



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

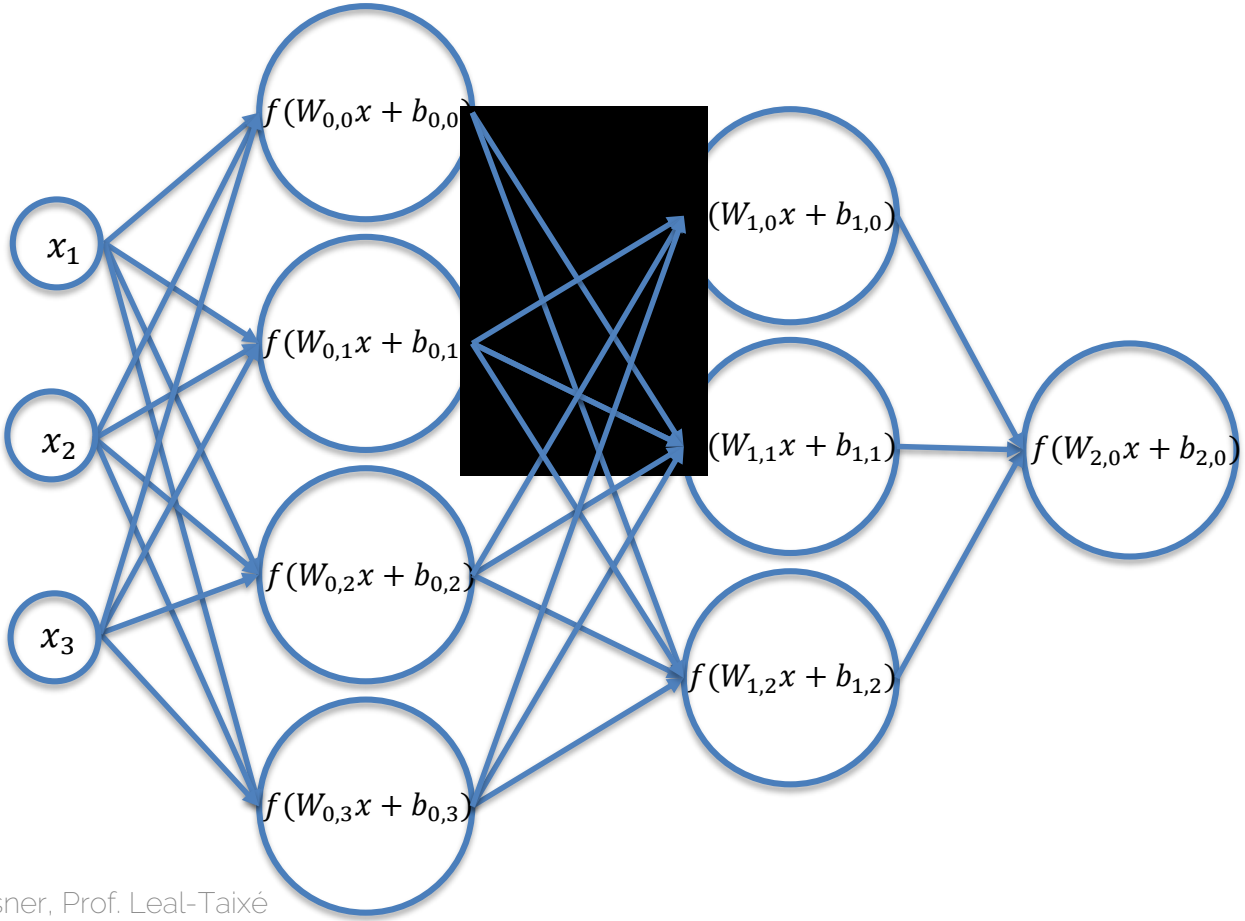


$128 \times 128 = 16384$ — Input Layer

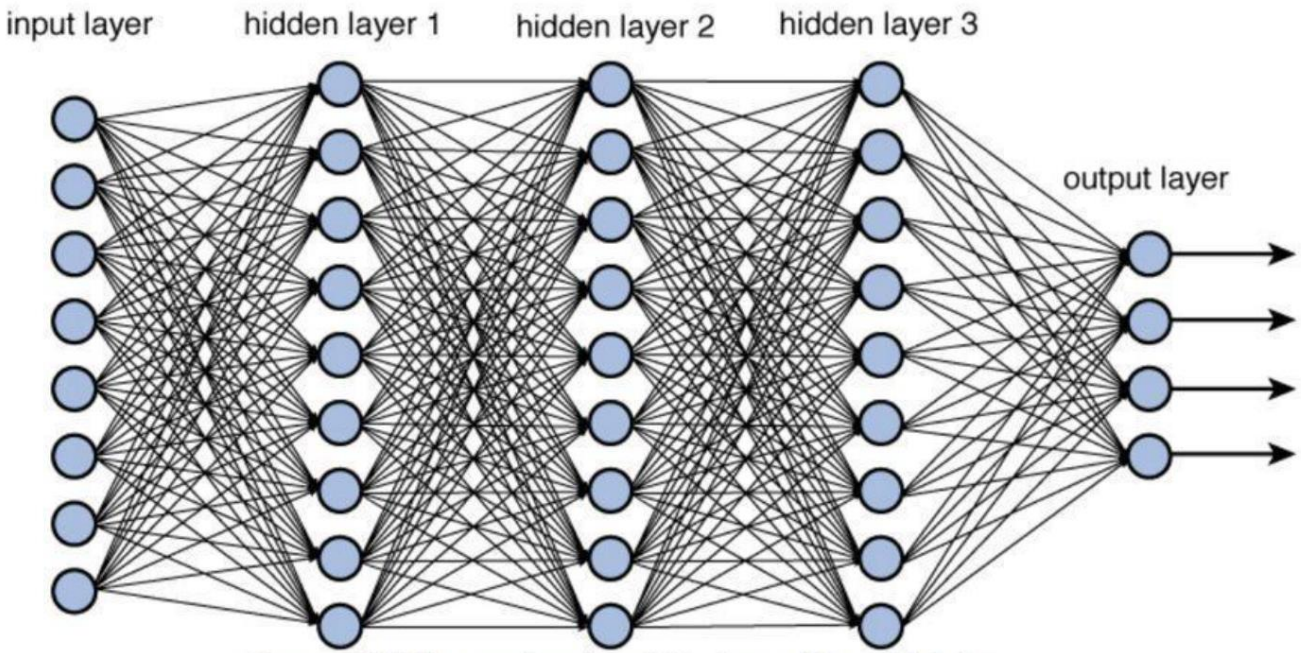
1000 — Hidden Layer

10 — Output Layer

Net of Artificial Neurons



Neural Network

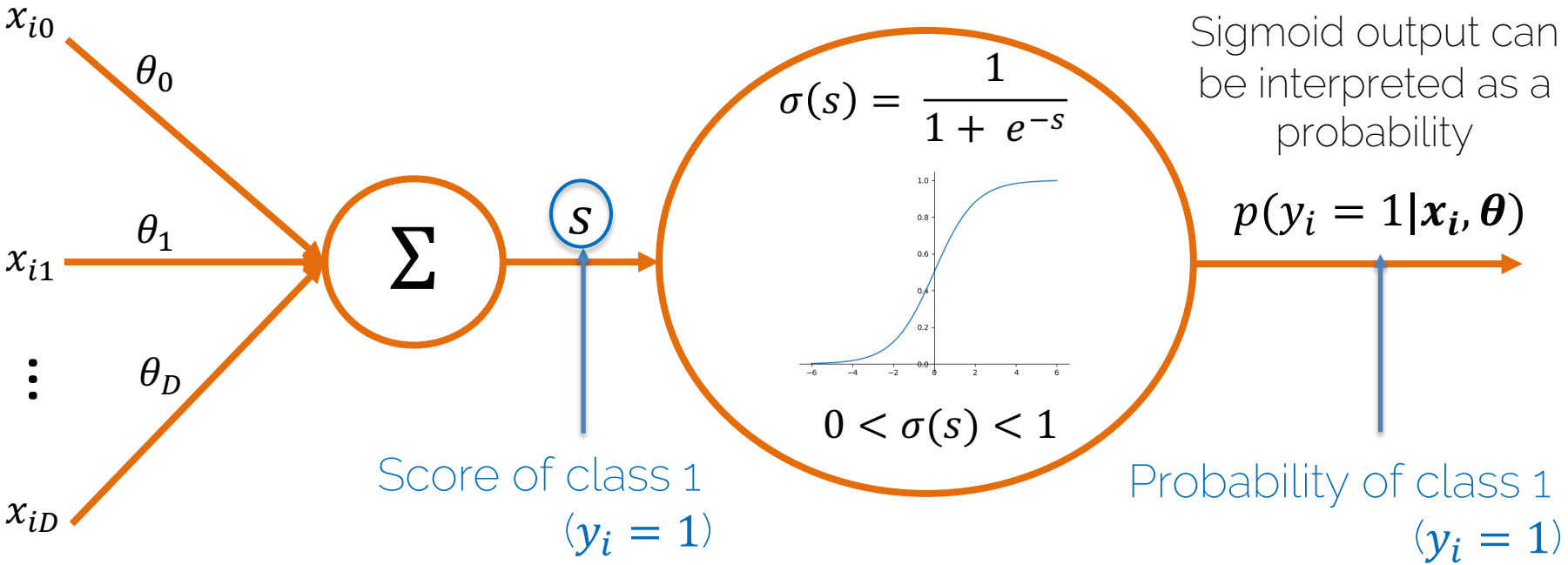


Source: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

Binary Classification: Sigmoid

training pairs $[\boldsymbol{x}_i; y_i]$,
$\boldsymbol{x}_i \in \mathbb{R}^D, y_i \in \{1,0\}$ (2 classes)

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



Multiclass Classification: Softmax

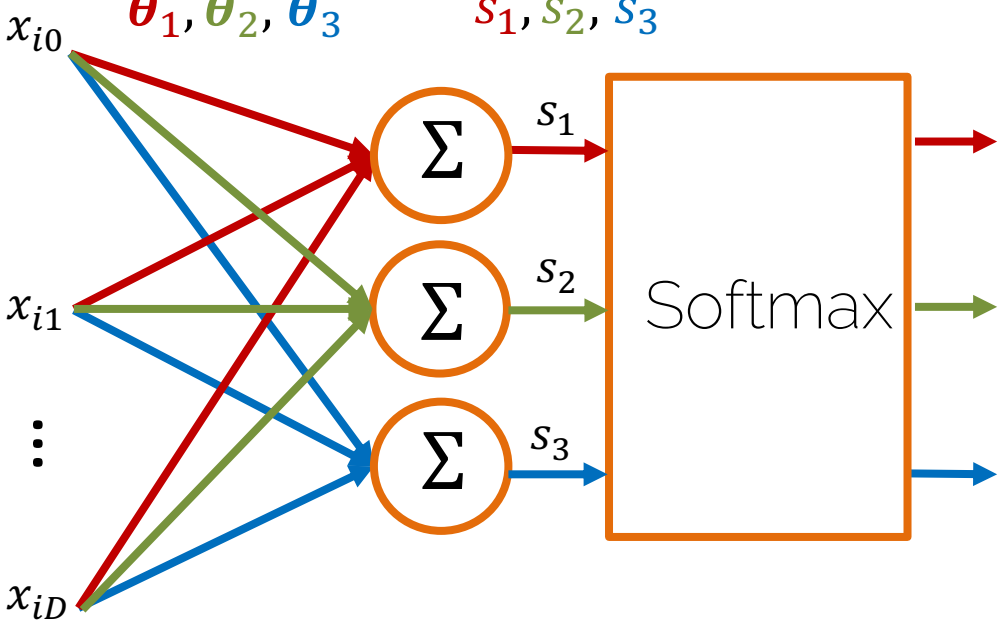
Weights for each class

$\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3$

Scores for each class

s_1, s_2, s_3

Probabilities for each class



$$p(y_i = 1|\boldsymbol{x_i}, \boldsymbol{\Theta}) = \frac{e^{\boldsymbol{x_i\theta_1}}}{e^{\boldsymbol{x_i\theta_1}} + e^{\boldsymbol{x_i\theta_2}} + e^{\boldsymbol{x_i\theta_3}}}$$

$$p(y_i = 2|\boldsymbol{x_i}, \boldsymbol{\Theta}) = \frac{e^{\boldsymbol{x_i\theta_2}}}{e^{\boldsymbol{x_i\theta_1}} + e^{\boldsymbol{x_i\theta_2}} + e^{\boldsymbol{x_i\theta_3}}}$$

$$p(y_i = 3|\boldsymbol{x_i}, \boldsymbol{\Theta}) = \frac{e^{\boldsymbol{x_i\theta_3}}}{e^{\boldsymbol{x_i\theta_1}} + e^{\boldsymbol{x_i\theta_2}} + e^{\boldsymbol{x_i\theta_3}}}$$

Multiclass Classification: Softmax

- Softmax

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

Exp

normalize

Probability of the true class

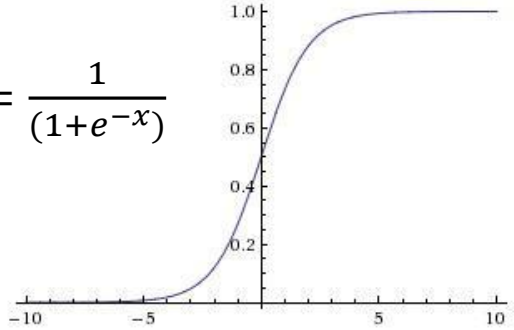
training pairs $[\boldsymbol{x_i}; y_i]$,
$\boldsymbol{x_i} \in \mathbb{R}^D, y_i \in \{1,2 \dots C\}$
y_i: label (true class)

Parameters:
$\boldsymbol{\Theta} = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_C]$
$\boldsymbol{C}$: number of classes
$\boldsymbol{s}$: score of the class

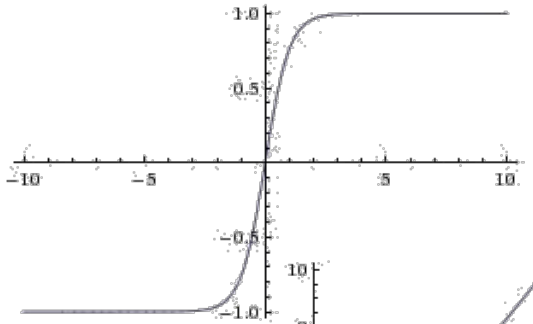
1. Exponential operation: make sure probability>0
2. Normalization: make sure probabilities sum up to 1.

Activation Functions

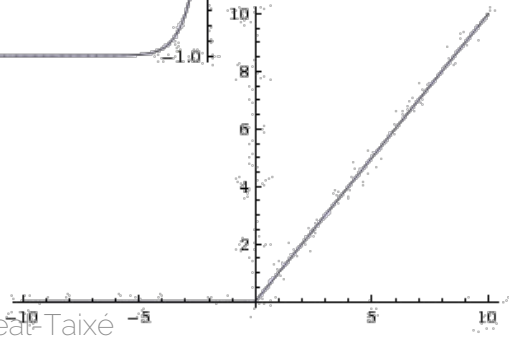
Sigmoid: $\sigma(x) = \frac{1}{(1+e^{-x})}$



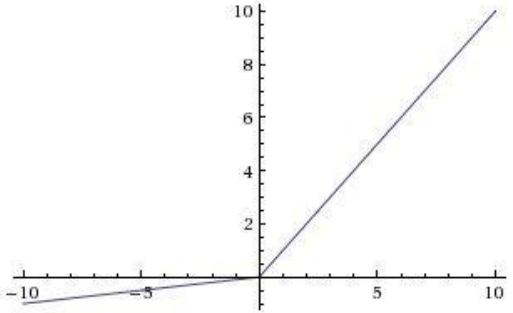
tanh: $\tanh(x)$



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

Loss Functions

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:

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

- MSE Loss:

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

Binary Cross Entropy

- Loss function for binary (yes/no) classification

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

Yes! (0.8)

No! (0.2)

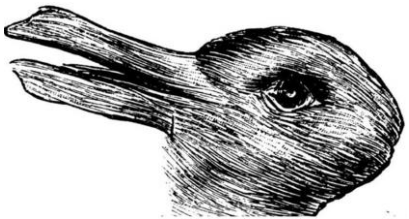
The network predicts the probability of the input belonging to the "yes" class!

Cross Entropy

= loss function for multi-class classification

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

This generalizes the binary case from the slide before!



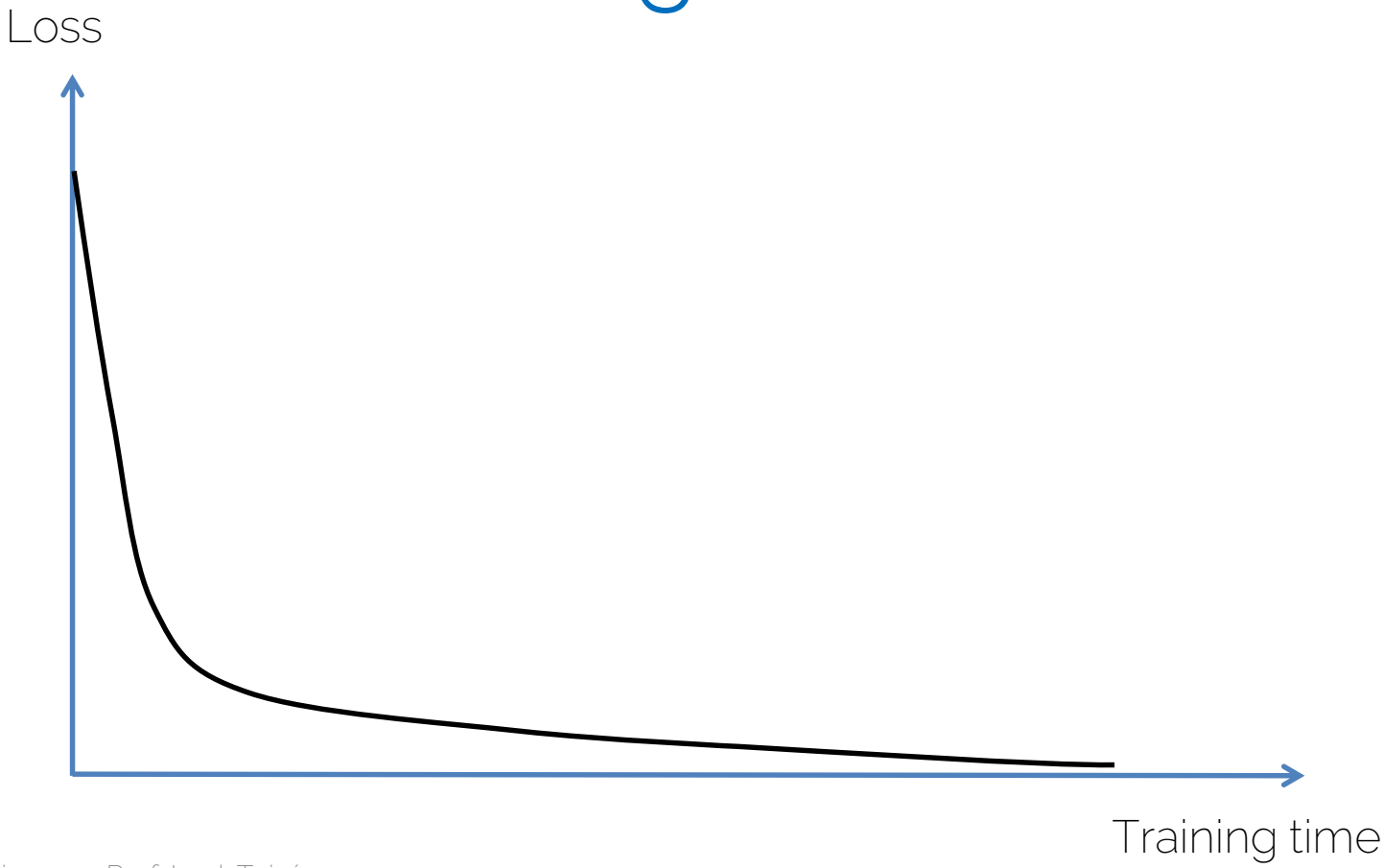
dog (0.1)

rabbit (0.2)

duck (0.7)

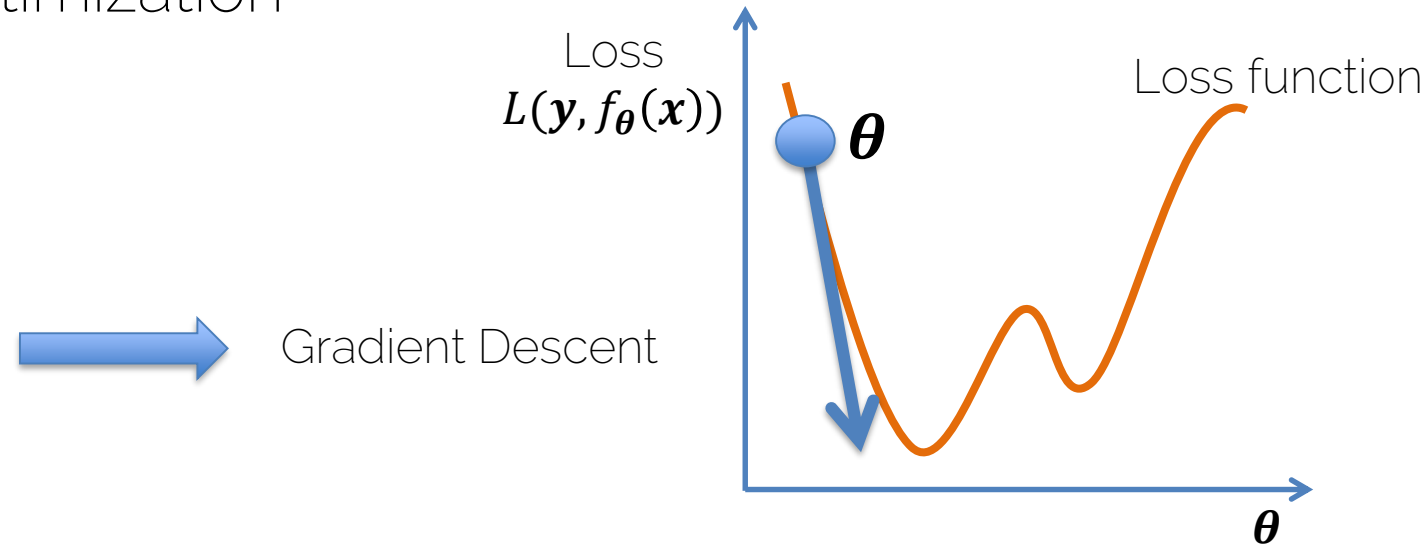
…

Training Curve



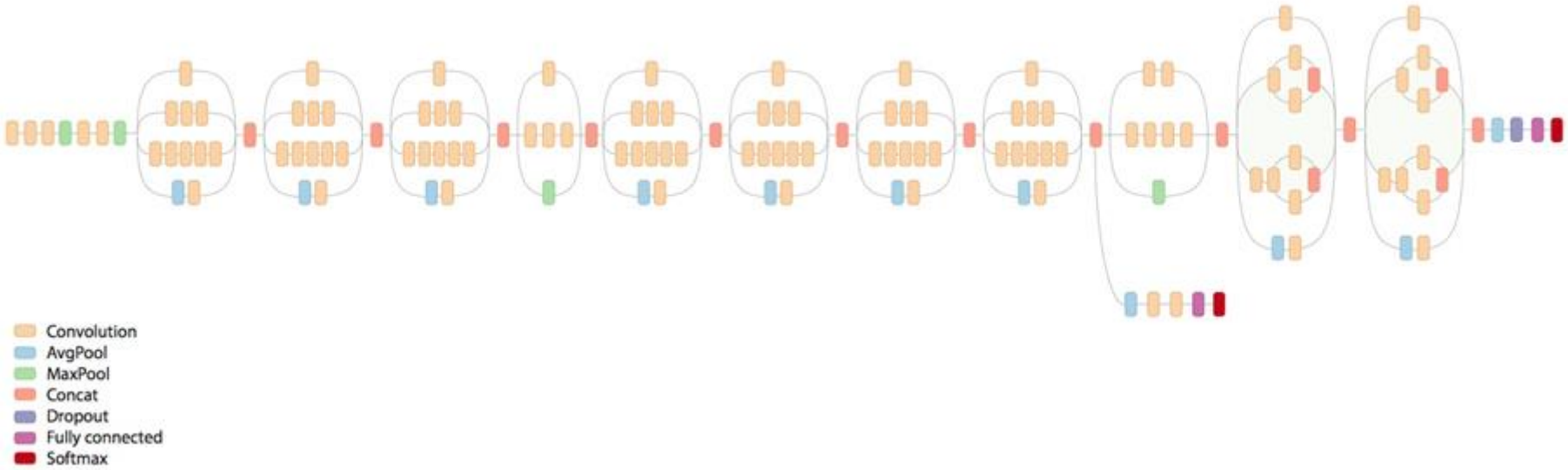
How to Find a Better NN?

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



NNs can Become Quite Complex…

- These graphs can be huge!



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

The Flow of the Gradients

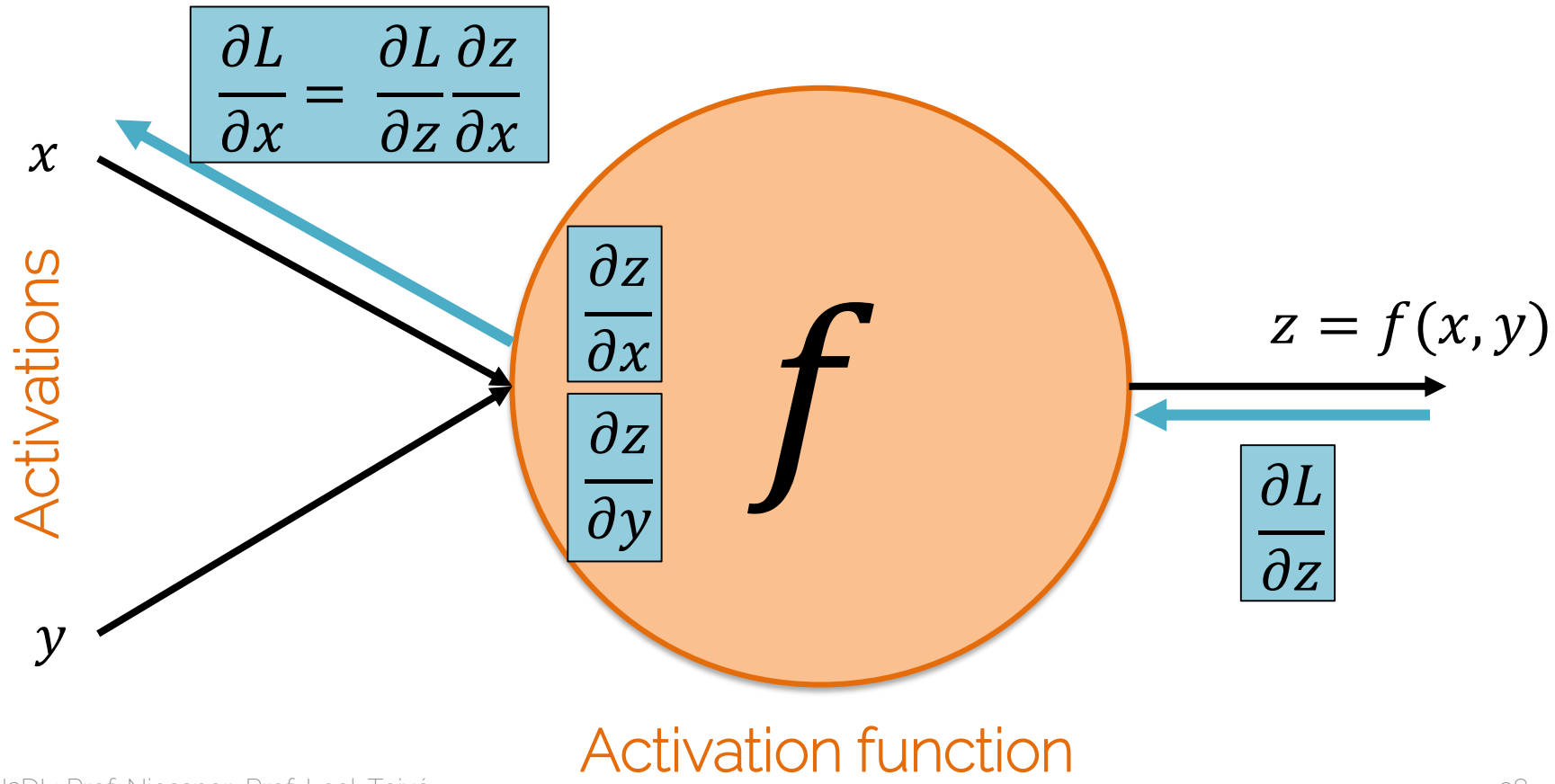
- Many many many many of these nodes form a neural network

NEURONS

- Each one has its own work to do

FORWARD AND BACKWARD PASS

The Flow of the Gradients



Stochastic Gradient Descent (SGD)

- If we have $\boldsymbol{n}$ training samples we need to compute the gradient for all of them which is $\boldsymbol{O(n)}$

- 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

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

Stochastic Gradient Descent (SGD)

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

$$\mathbb{E}_{i\sim[1,\dots,n]}[L_i(\boldsymbol{\theta}, \boldsymbol{x_i}, \boldsymbol{y_i})] \approx \frac{1}{|S|}\sum_{j\in S}\left(L_j(\boldsymbol{\theta}, \boldsymbol{x_j}, \boldsymbol{y_j})\right) \text{ with S} \subseteq \{1, \dots, n\}$$

Minibatch
choose subset of trainset $\boldsymbol{m} \ll \boldsymbol{n}$

$$B_i = \{\{\boldsymbol{x_1}, \boldsymbol{y_1}\}, \{\boldsymbol{x_2}, \boldsymbol{y_2}\}, \dots, \{\boldsymbol{x_m}, \boldsymbol{y_m}\}\}$$
$$\{B_1, B_2, \dots, B_{n/m}\}$$

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)

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

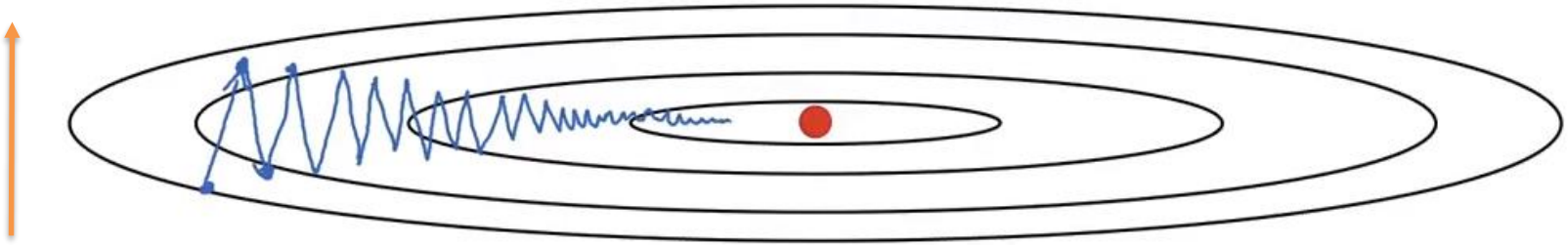
$\boldsymbol{k}$ now refers to $\boldsymbol{k}$-th iteration

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

$\boldsymbol{m}$ training samples in the current minibatch

Gradient for the $\boldsymbol{k}$-th minibatch

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

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

accumulation rate ('friction', momentum)

velocity

Gradient of current minibatch

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

model

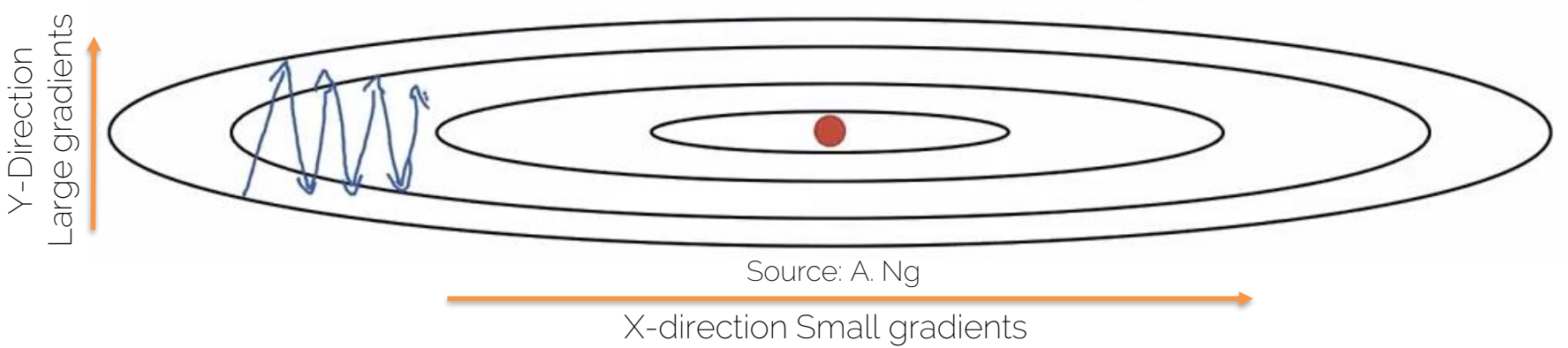
learning rate

velocity

Exponentially-weighted average of gradient

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

RMSProp



(Uncentered) variance of gradients
→ second momentum

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

We're dividing by square gradients:
- Division in Y-Direction will be large
- Division in X-Direction will be small

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

Can increase learning rate!

Adam

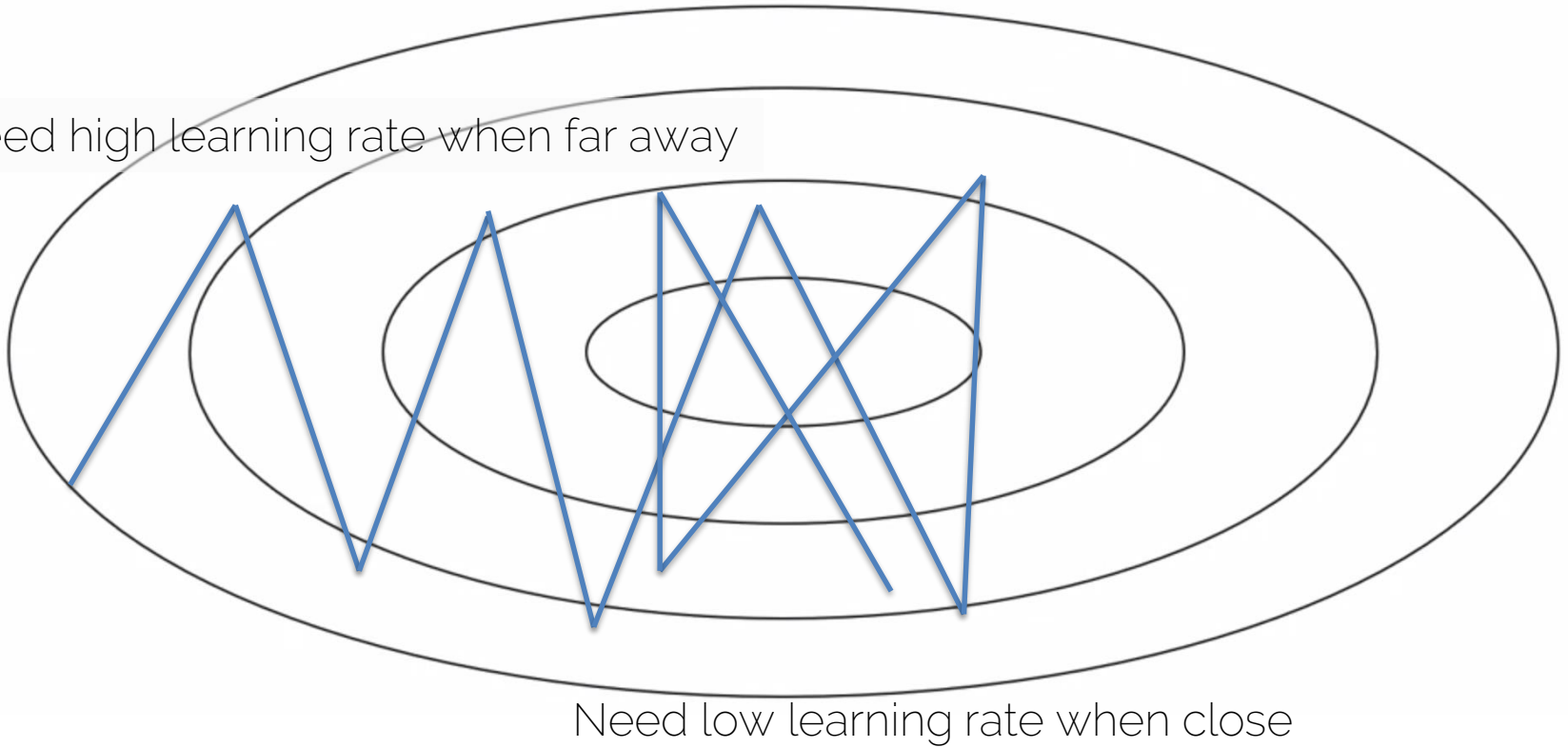
- Combines Momentum and RMSProp

$$\boldsymbol{m}^{k+1} = \beta_1 \cdot \boldsymbol{m}^k + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k\right) \qquad \boldsymbol{v}^{k+1} = \beta_2 \cdot \boldsymbol{v}^k + (1-\beta_2)[\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k\right) \circ \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k\right)$$

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

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

Learning Rate

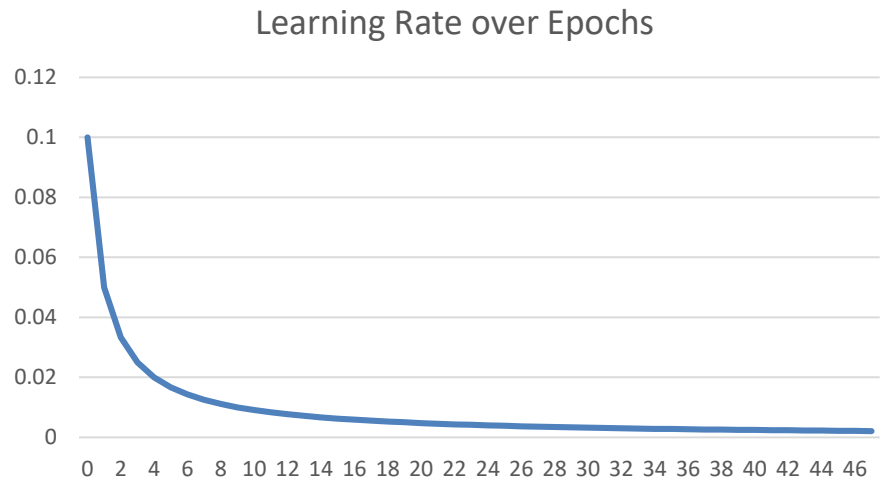


Learning Rate Decay

- $\alpha = \frac{1}{1+decay\_rate*epoch} \cdot \alpha_0$

  - 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

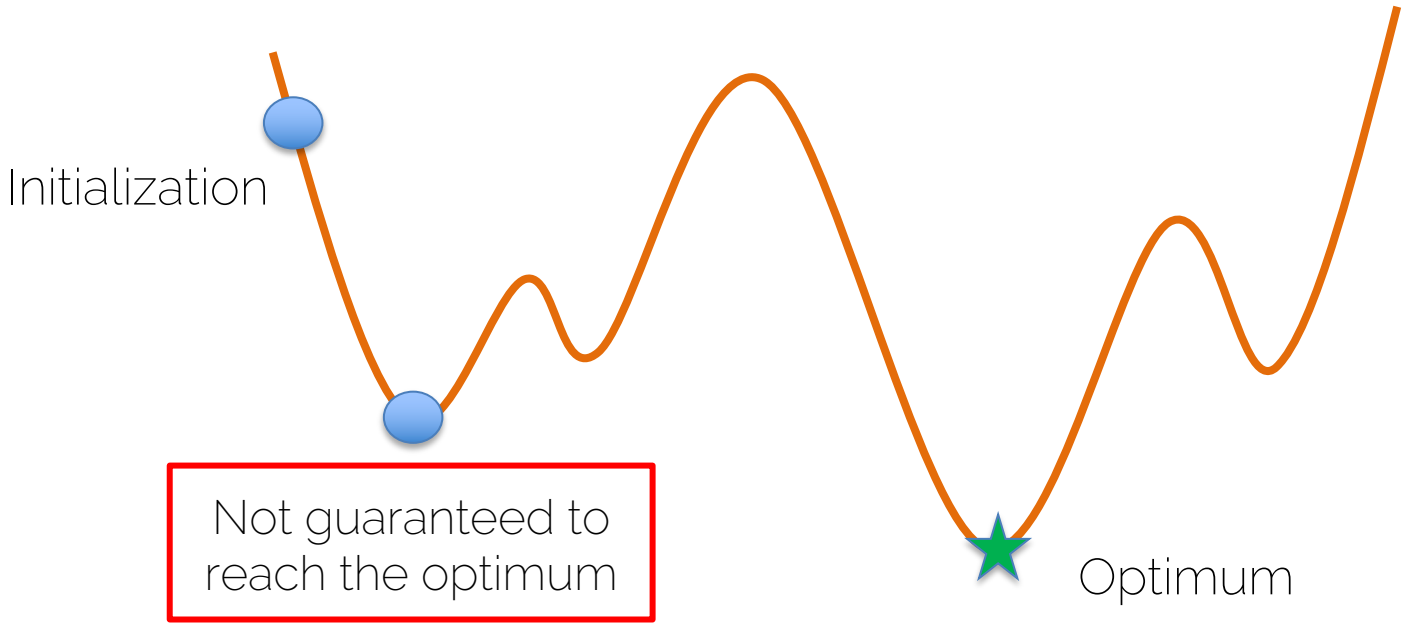
- Manually set learning rate every n-epochs
- How?
  - Trial and error (the hard way)
  - Some experience (only generalizes to some degree)

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

Weight Initialization

Initialization is Extremely Important

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



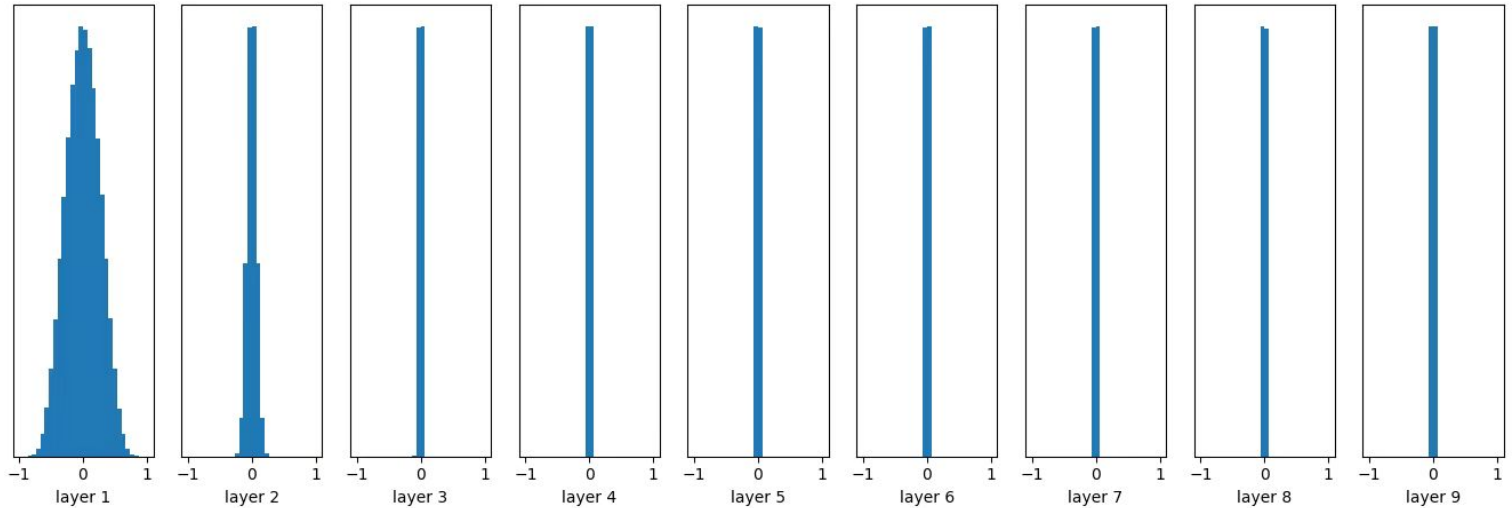
All Weights Zero

- 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

tanh as activation functions

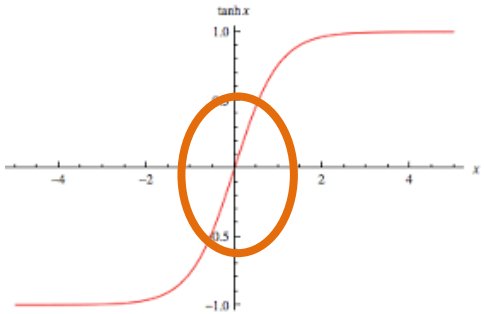


Output become to zero

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 $l+1$:

$$f_{l+1}\left(\sum_i w_i^{l+1} x_i^{l+1} + b^{l+1}\right)$$

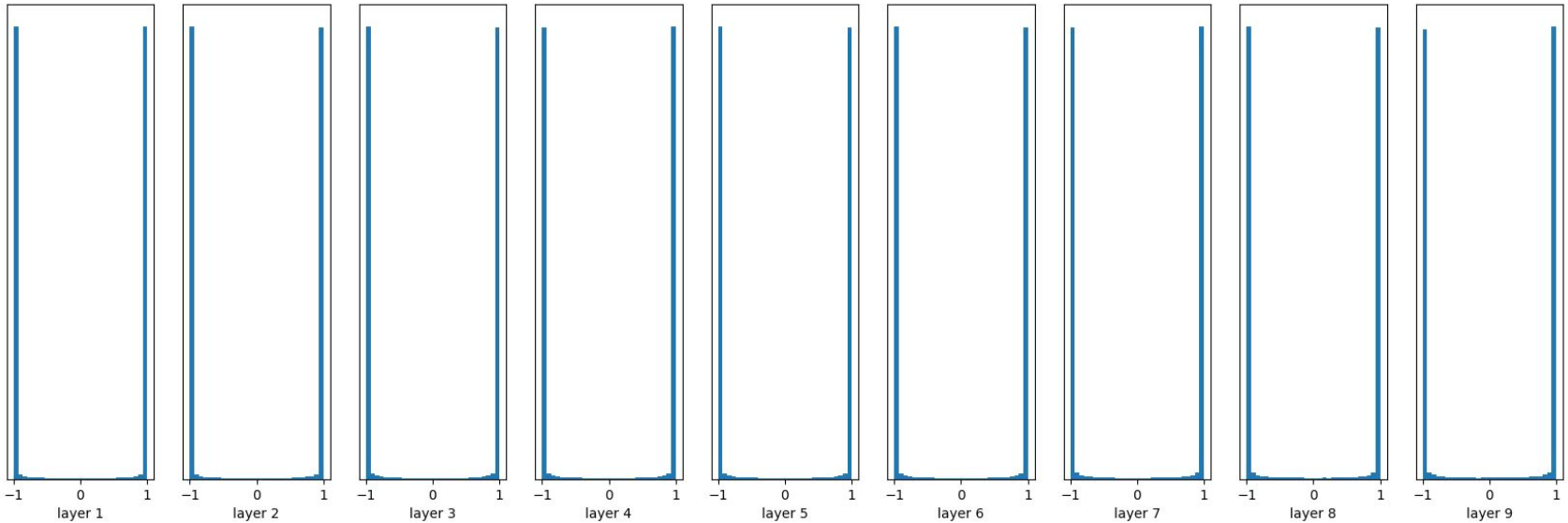
$$\frac{\partial L}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot \frac{\partial f_{l+1}}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot x_i^{l+1} \approx 0$$

Vanishing gradient, caused by small output

Backward

Big Random Numbers

tanh as activation functions



Output saturated to -1 and 1

Xavier Initialization

- How to ensure the variance of the output is the same as the input?

Goal:

$$Var(s) = Var(x) \longrightarrow \underbrace{n \cdot Var(w)}_{=1} Var(x) = Var(x)$$

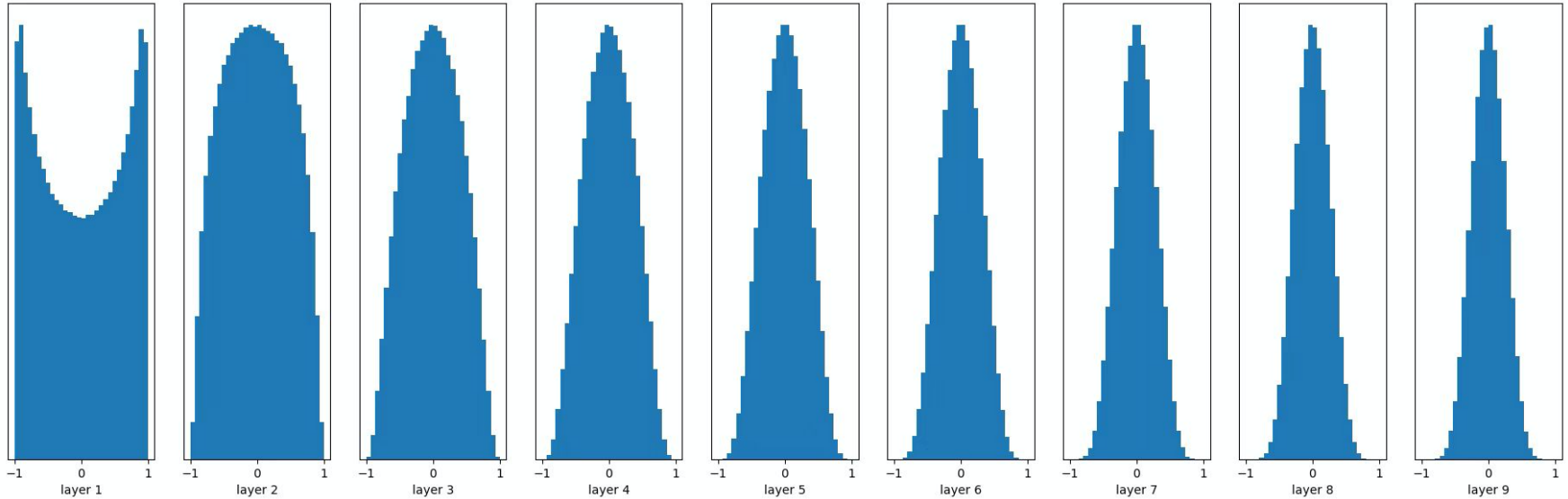
$$\longrightarrow Var(w) = \frac{1}{n}$$

n: number of input neurons

Xavier Initialization

$$Var(w) = \frac{1}{n}$$

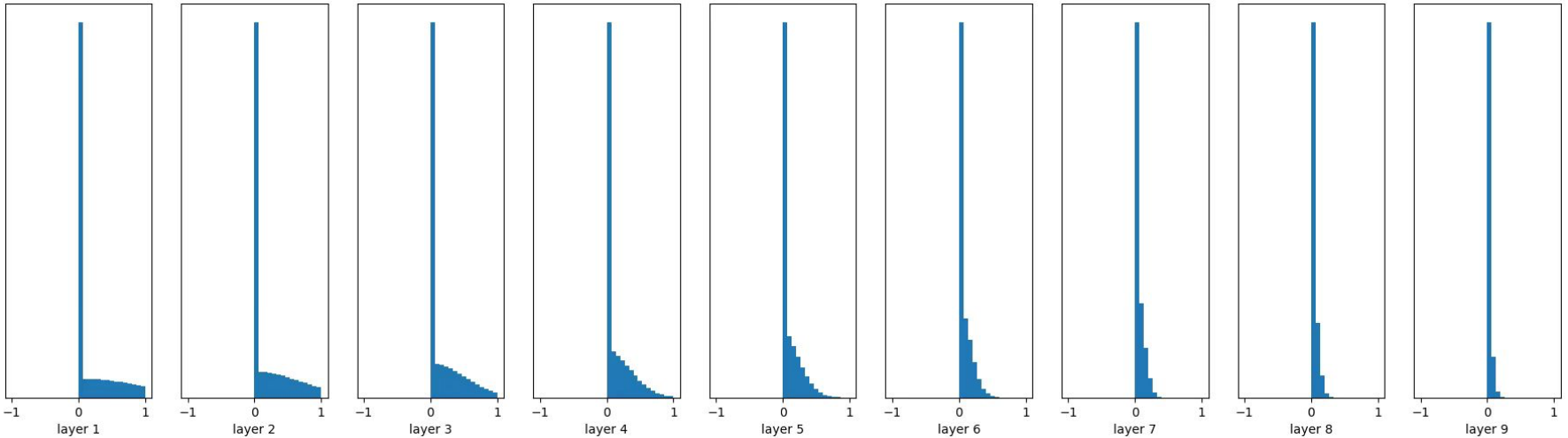
tanh as activation functions



Xavier Initialization with ReLU

$$Var(w) = \frac{1}{n}$$

tanh as activation functions



ReLU kills Half of the Data

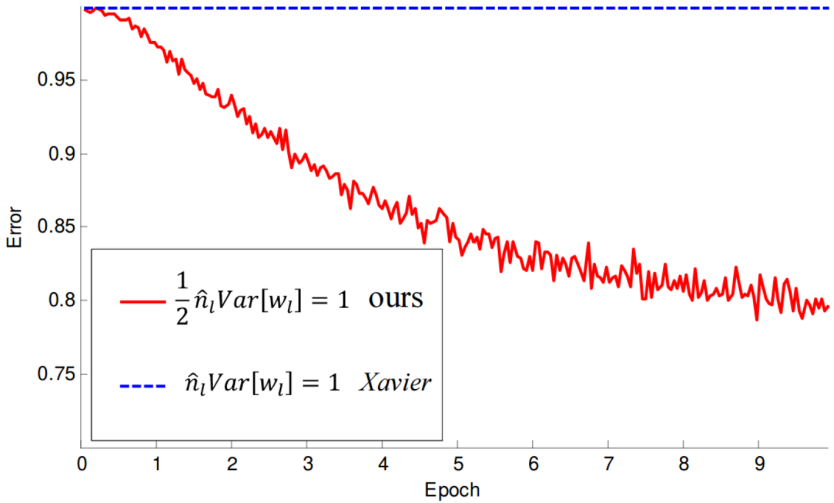
What's the solution?

When using ReLU, output close to zero again ☹

Xavier/2 Initialization with ReLU

$$Var(w) = \frac{2}{n}$$

It makes a huge difference!

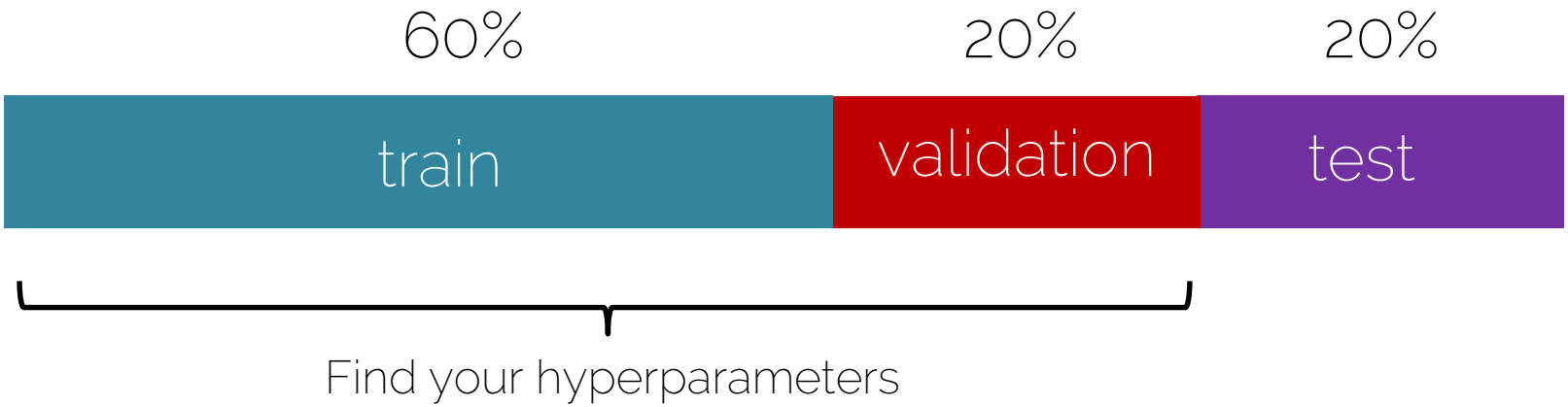


- Use ReLU and Xavier/2 initialization

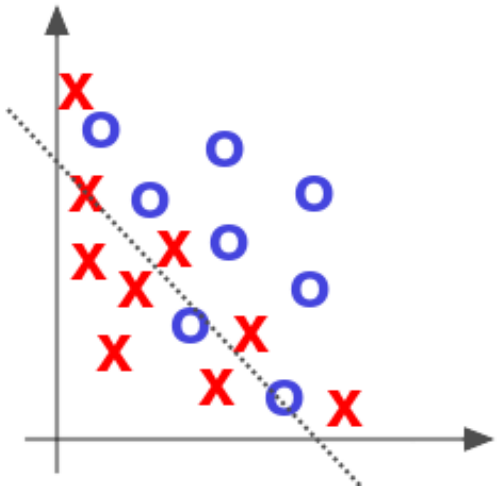
[He et al., ICCV'15] He Initialization

Basic Recipe for Machine Learning

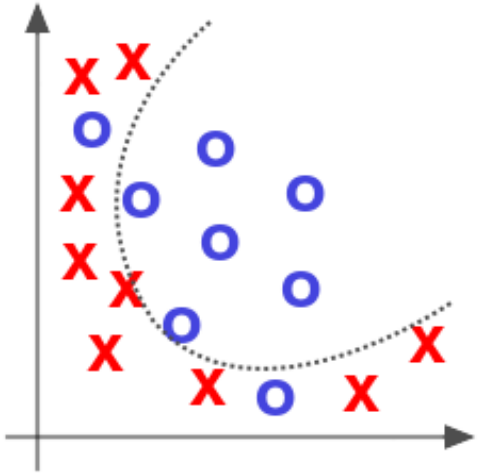
- Split your data



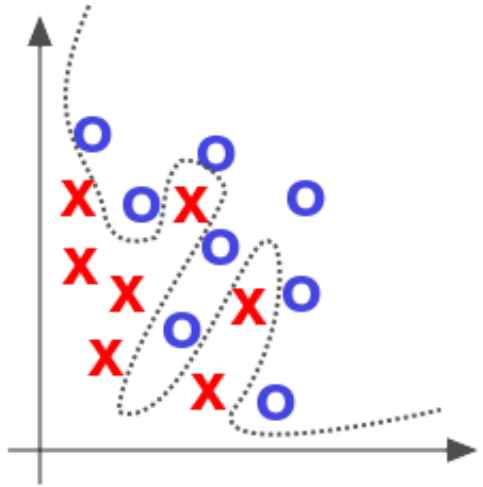
Over- and Underfitting



Underfitted



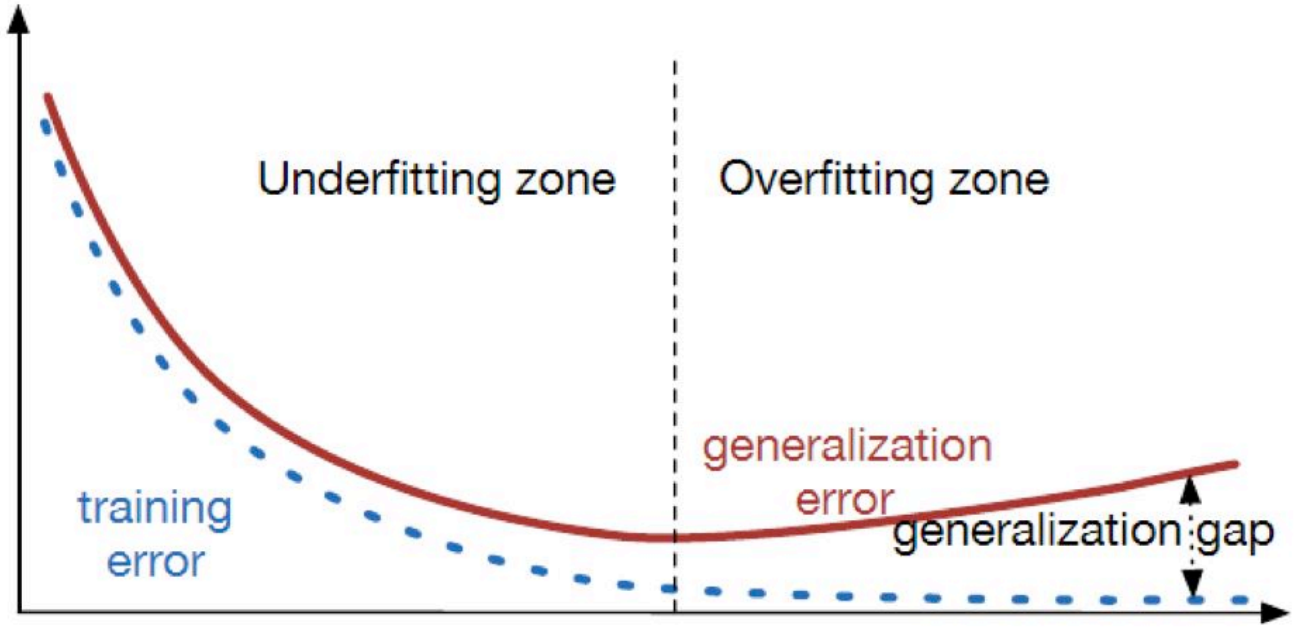
Appropriate



Overfitted

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

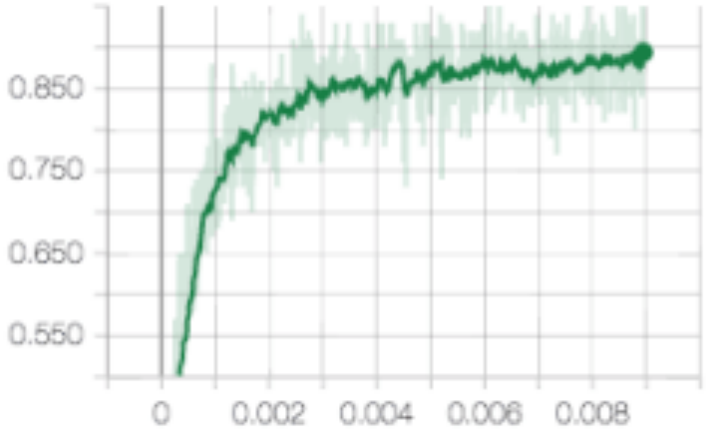
Over- and Underfitting

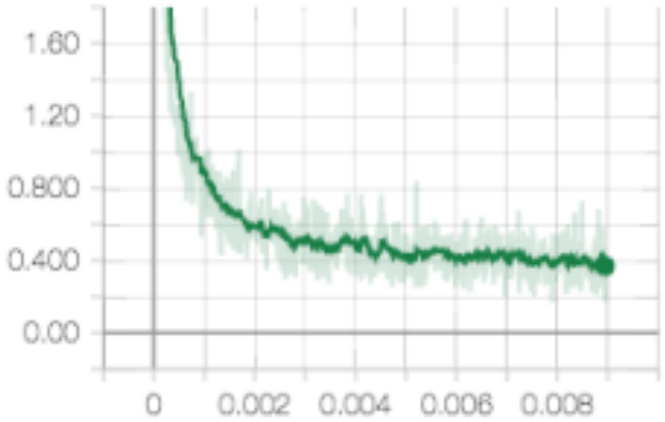


Source: https://srdas.github.io/DLBook/ImprovingModelGeneralization.html

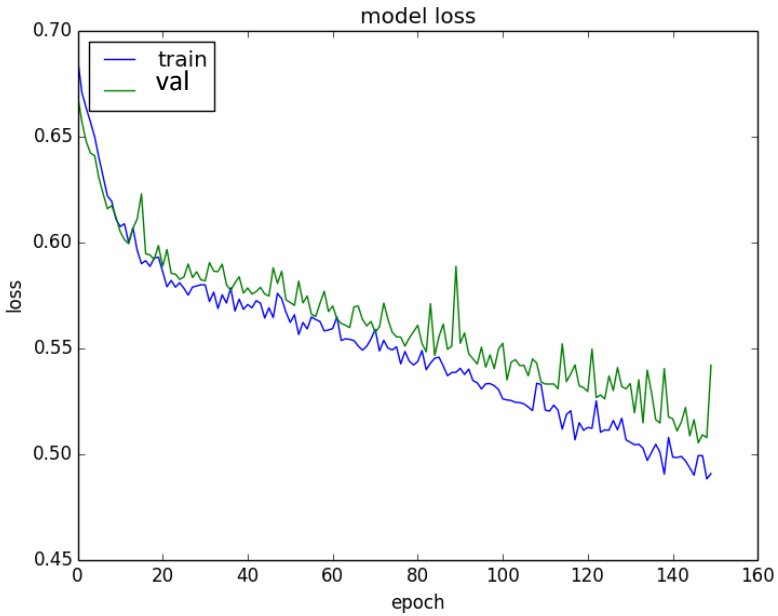
Learning Curves

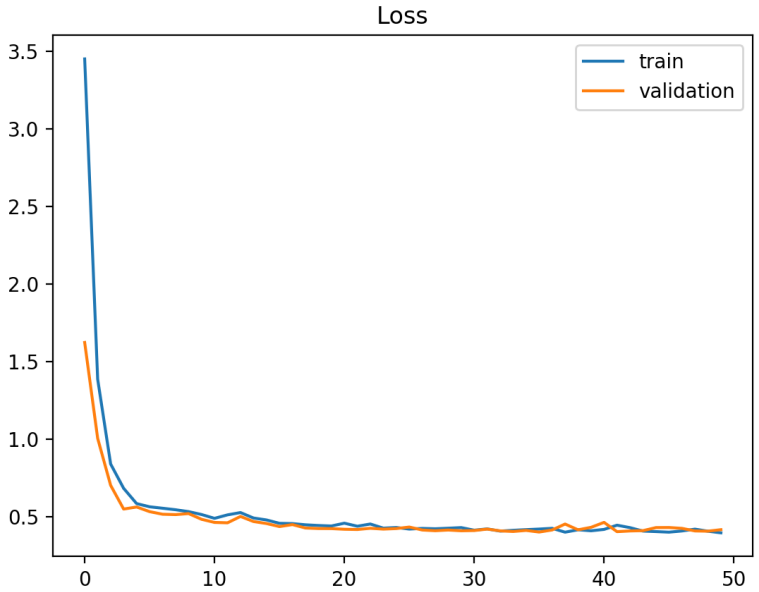
- Training graphs
  - Accuracy
  - Loss





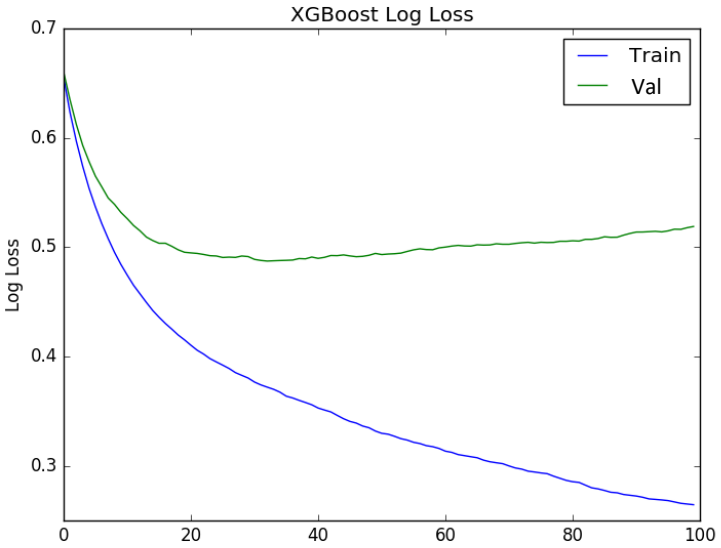
Learning Curves

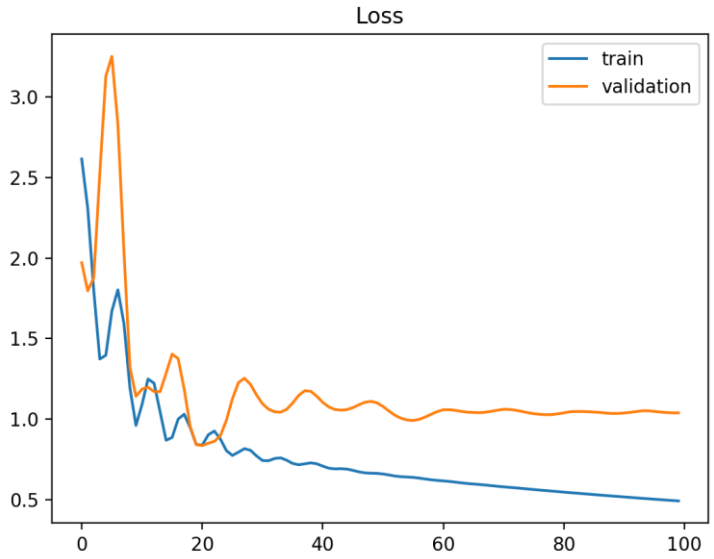




Source: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

Overfitting Curves





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

Regularization

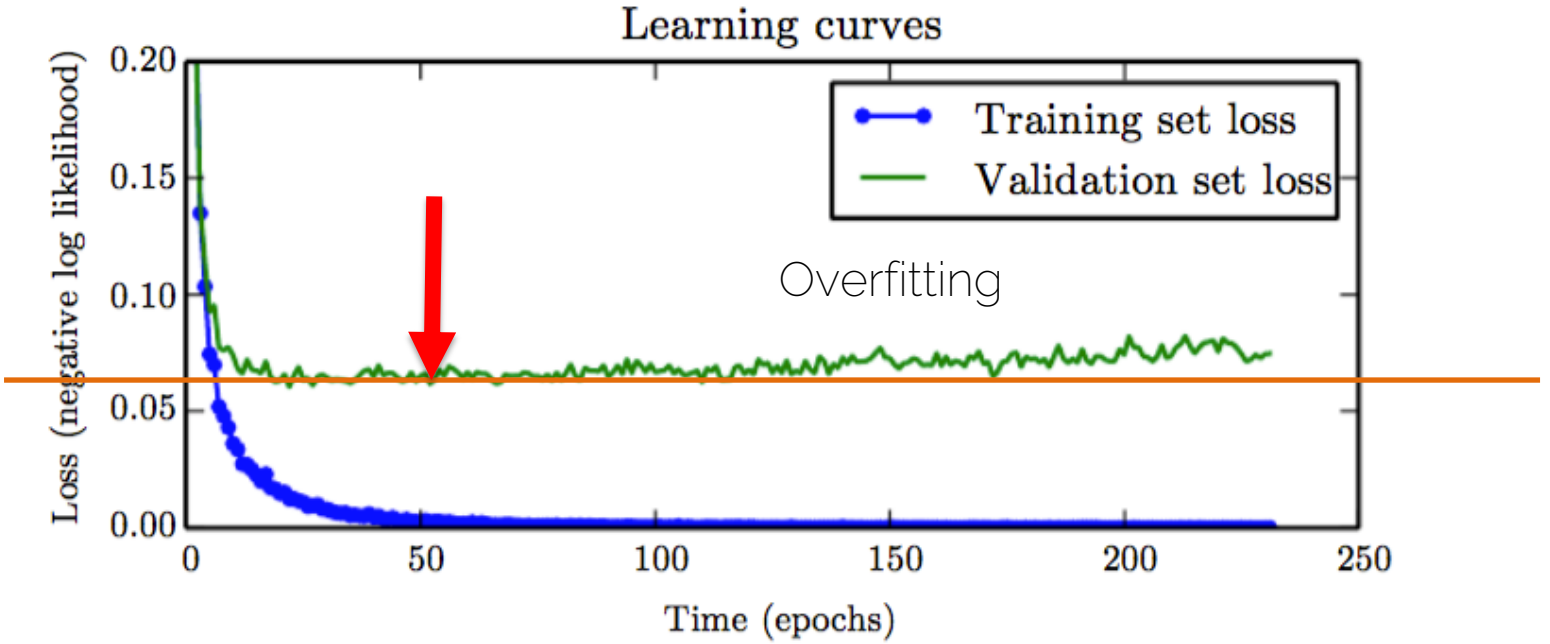
- Loss function $L(\boldsymbol{y}, \hat{\boldsymbol{y}}, \boldsymbol{\theta}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$

- Regularization techniques
  - L2 regularization
  - L1 regularization
  - Max norm regularization
  - Dropout
  - Early stopping
  - ...

Add **regularization term** to loss function

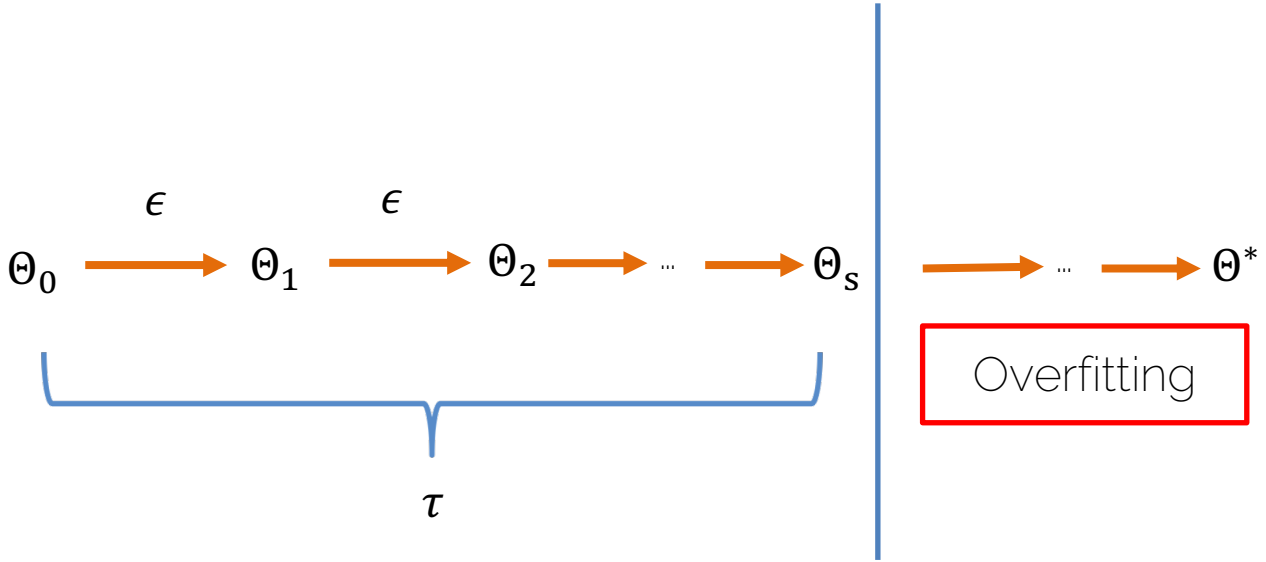
More details later

Early Stopping



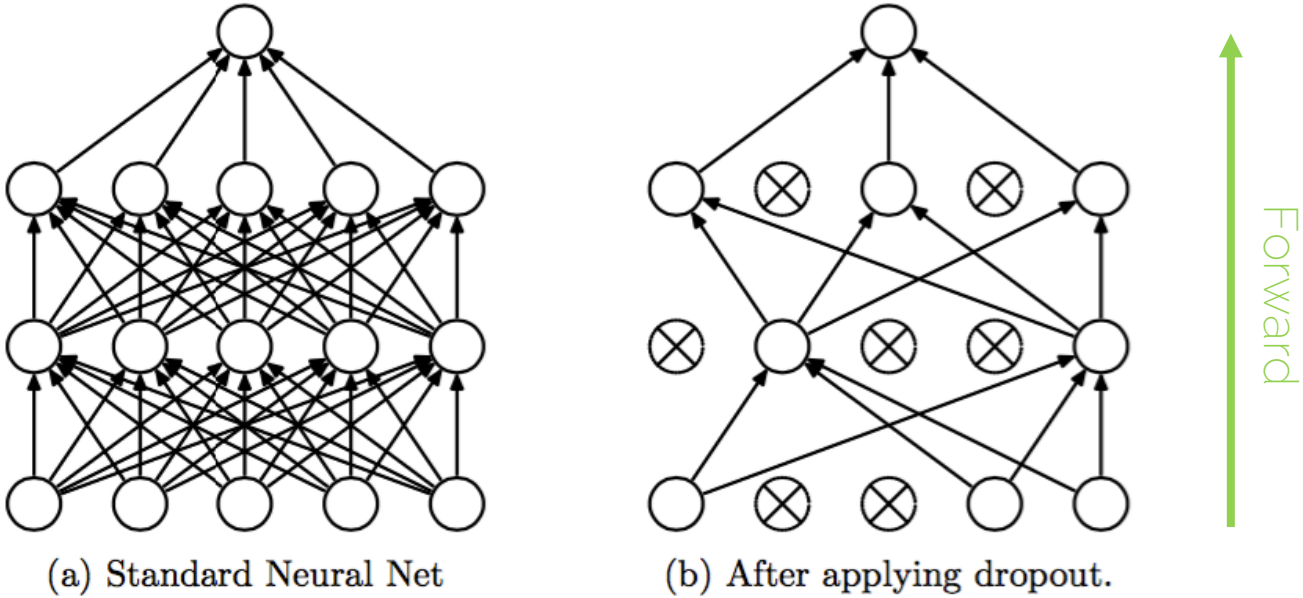
Early Stopping

- Easy form of regularization



Dropout

- Disable a random set of neurons (typically 50%)



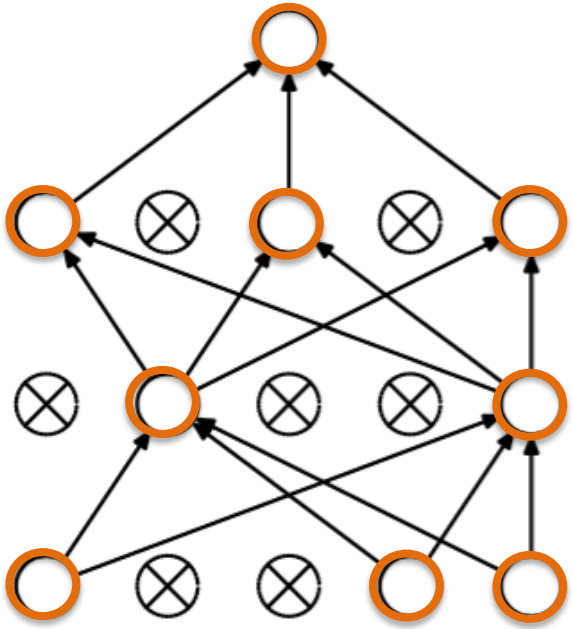
(a) Standard Neural Net

(b) After applying dropout.

[Srivastava et al., JMLR'14] Dropout

Dropout: Intuition

- Two models in one



(b) After applying dropout.

Model 1

Model 2

[Srivastava et al., JMLR'14] Dropout

Dropout: Intuition

- Using half the network = half capacity
  - Redundant representations
  - Base your scores on more features

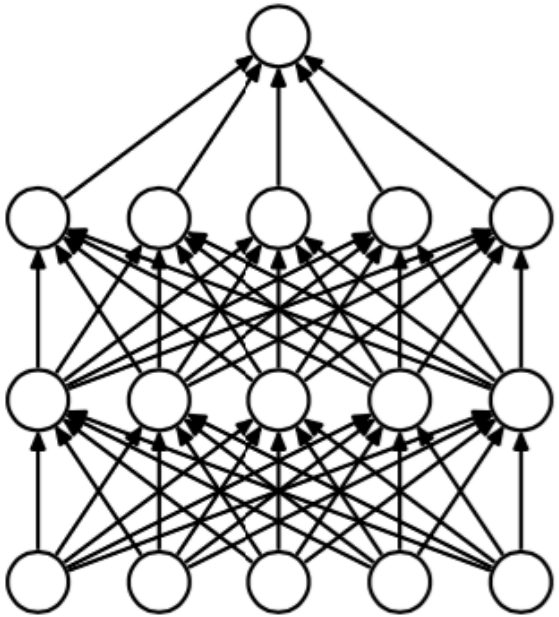
- Consider it as two models in one
  - Training a large ensemble of models, each on different set of data (mini-batch) and with SHARED parameters

Reducing co-adaptation between neurons

[Srivastava et al., JMLR'14] Dropout

Dropout: Test Time

- All neurons are "turned on" – no dropout



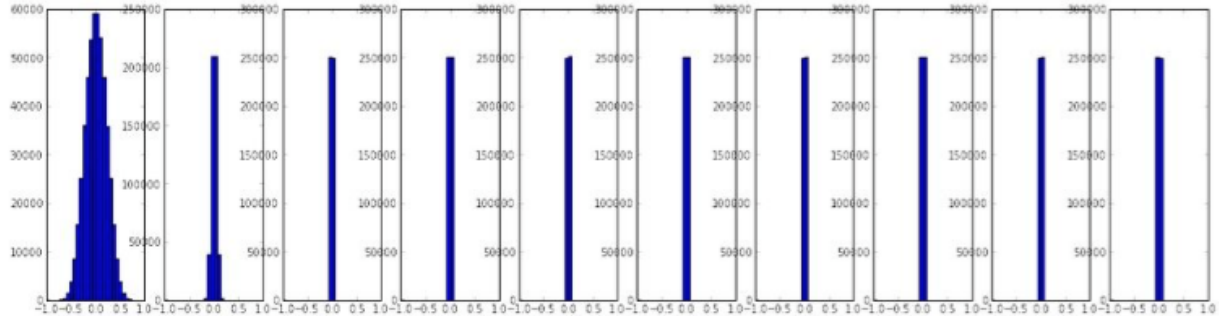
Conditions at train and test time are not the same

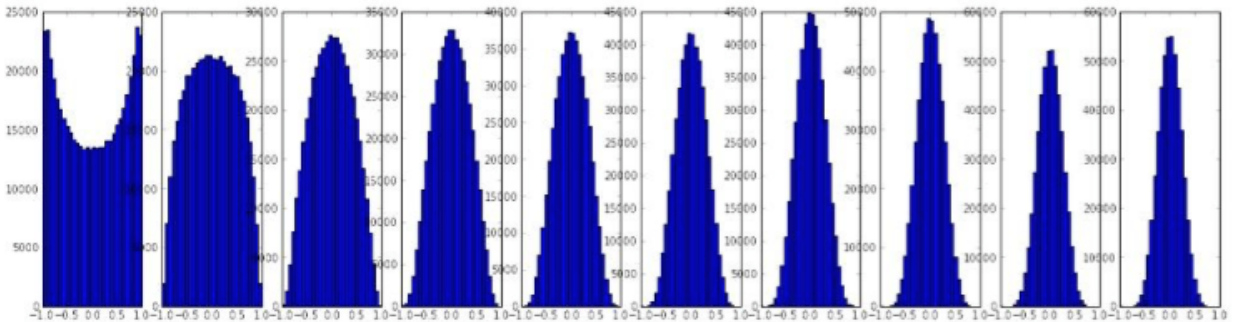
[Srivastava et al., JMLR'14] Dropout

Batch Normalization

Our Goal

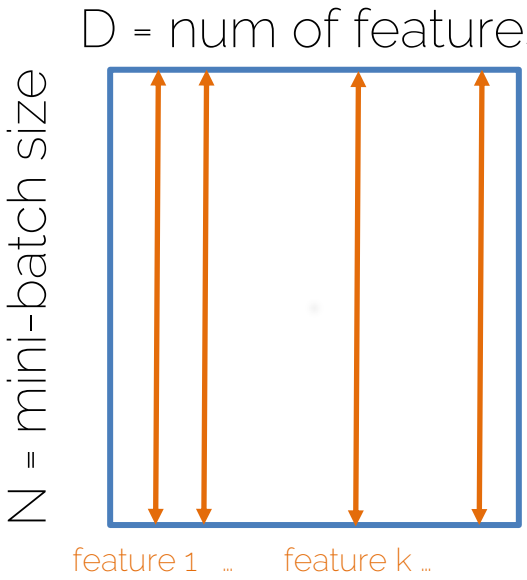
- All we want is that our activations do not die out





Batch Normalization

- Wish: Unit Gaussian activations (in our example)
- Solution: let's do it

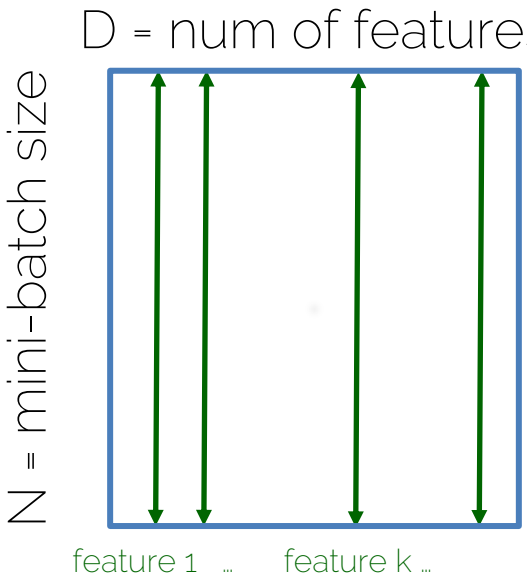


Mean of your mini-batch examples over feature k

$$\hat{\boldsymbol{x}}^{(k)} = \frac{\boldsymbol{x}^{(k)} - E[\boldsymbol{x}^{(k)}]}{\sqrt{Var[\boldsymbol{x}^{(k)}]}}$$

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

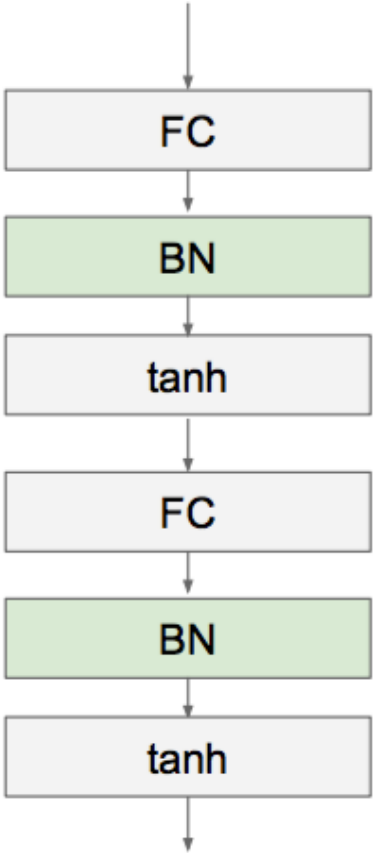
$$\hat{\boldsymbol{x}}^{(k)} = \frac{\boldsymbol{x}^{(k)} - E[\boldsymbol{x}^{(k)}]}{\sqrt{Var[\boldsymbol{x}^{(k)}]}}$$

Unit gaussian

[Ioffe and Szegedy, PMLR'15] Batch Normalization

BN Layer

- A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions



[Ioffe and Szegedy, PMLR'15] Batch Normalization

Batch Normalization

- 1. Normalize

$$\hat{\boldsymbol{x}}^{(k)} = \frac{\boldsymbol{x}^{(k)} - E[\boldsymbol{x}^{(k)}]}{\sqrt{Var[\boldsymbol{x}^{(k)}]}}$$

Differentiable function so we can backprop through it….

- 2. Allow the network to change the range

$$\boldsymbol{y}^{(k)} = \gamma^{(k)}\hat{\boldsymbol{x}}^{(k)} + \beta^{(k)}$$

These parameters will be optimized during backprop

Batch Normalization

- 1. Normalize

$$\hat{\boldsymbol{x}}^{(k)} = \frac{\boldsymbol{x}^{(k)} - E[\boldsymbol{x}^{(k)}]}{\sqrt{Var[\boldsymbol{x}^{(k)}]}}$$

- 2. Allow the network to change the range

$$\boldsymbol{y}^{(k)} = \gamma^{(k)} \hat{\boldsymbol{x}}^{(k)} + \beta^{(k)}$$

backprop

The network *can* learn to undo the normalization

$$\gamma^{(k)} = \sqrt{Var[\boldsymbol{x}^{(k)}]}$$

$$\beta^{(k)} = E[\boldsymbol{x}^{(k)}]$$