Lecture 6:
Deep Neural Networks and Training

Zerrin Yumak

Utrecht University

In this lecture

- Feedforward neural networks
- Activation functions
- Backpropagation
- Regularization
- Dropout
- Optimization algorithms
- Weight initialization
- Batch normalization
- Hyper parameter tuning

today

Deep Neural Network (Pretraining)

SVM

Multi-layered Perceptron (Backpropagation)

XOR Problem

ADALINE

Perceptron

Golden Age

Dark Age ("AI Winter")

Electronic Brain

1943 1957 1960 1969 1986 1995 2006

1940 1950 1960 1970 1980 1990 2000 2010

S. McCulloch - W. Pitts

F. Rosenblatt

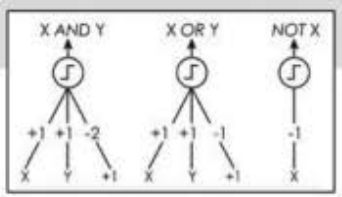
B. Widrow - M. Hoff

M. Minsky - S. Papert

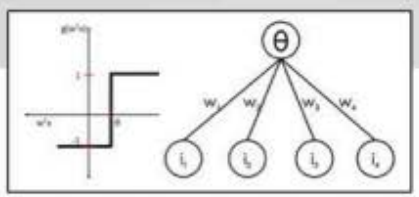
D. Rumelhart - G. Hinton - R. Wiliams

V. Vapnik - C. Cortes

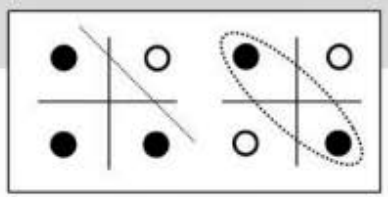
G. Hinton - S. Ruslan



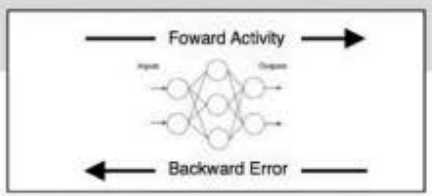
- Adjustable Weights
- Weights are not Learned



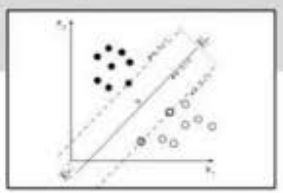
- Learnable Weights and Threshold



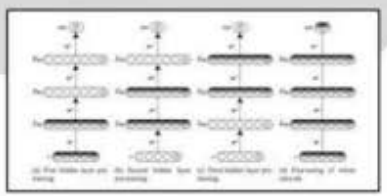
- XOR Problem



- Solution to nonlinearly separable problems
- Big computation, local optima and overfitting



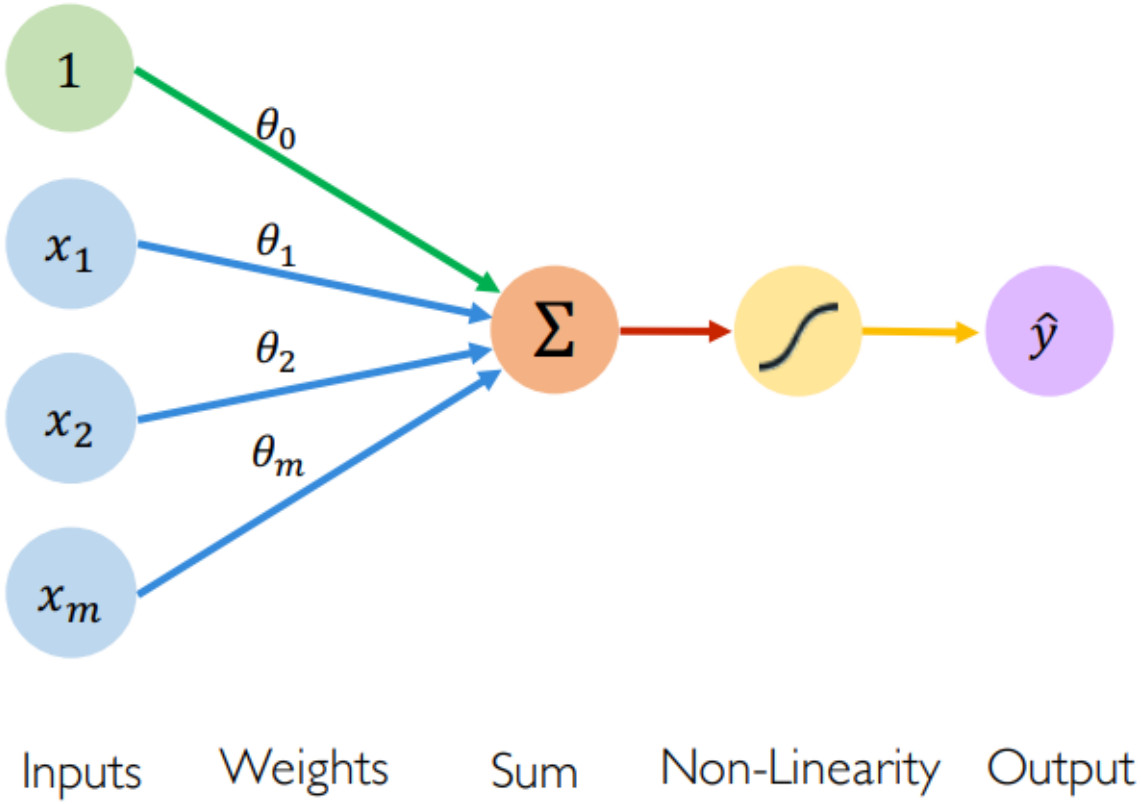
- Limitations of learning prior knowledge
- Kernel function: Human Intervention

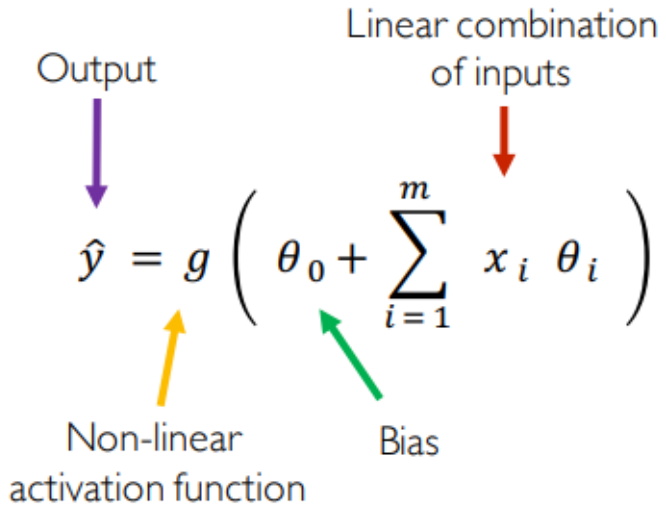


- Hierarchical feature Learning

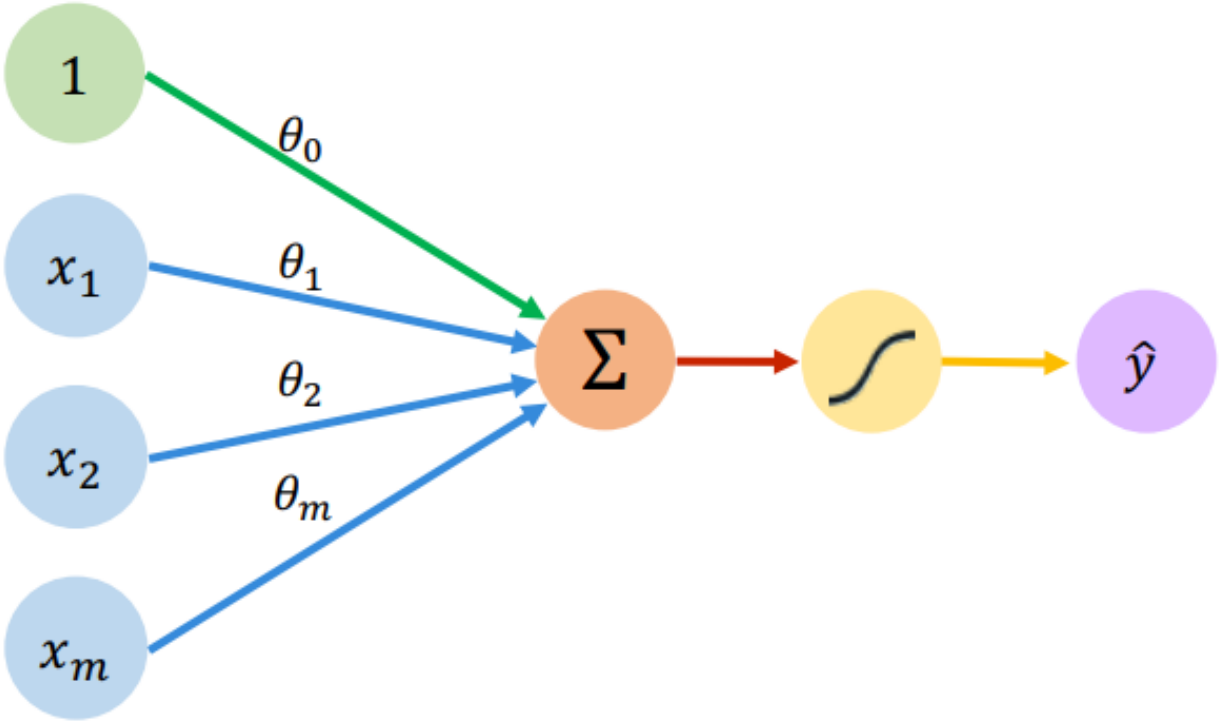
The Perceptron

- Building block of deep neural networks





The Perceptron



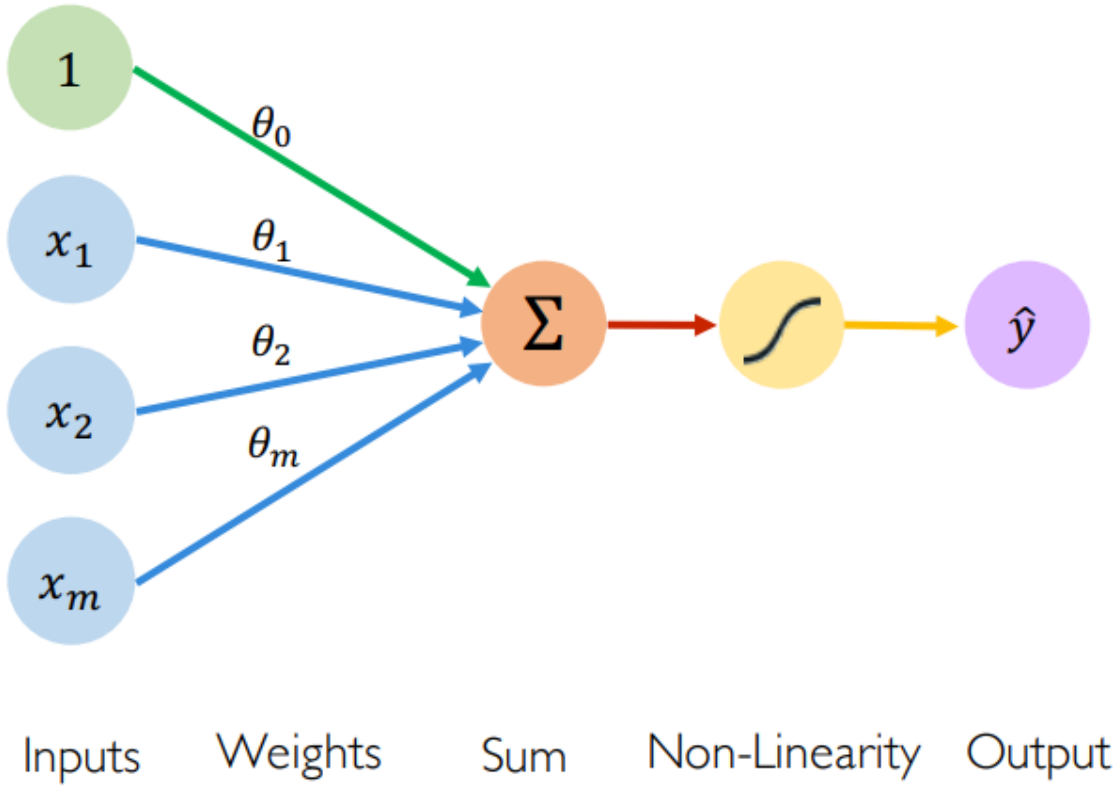
Inputs Weights Sum Non-Linearity Output

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\, \theta_i\right)$$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$

The Perceptron

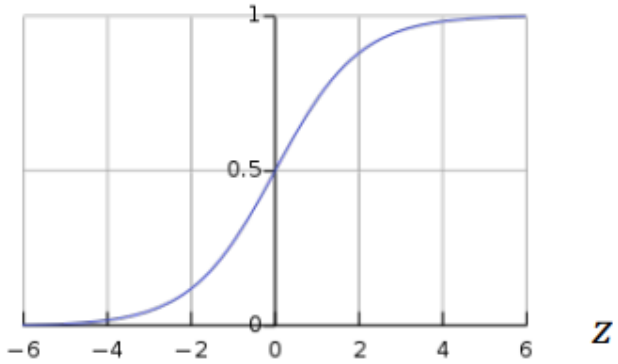


Activation Functions

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T\boldsymbol{\theta}\right)$$

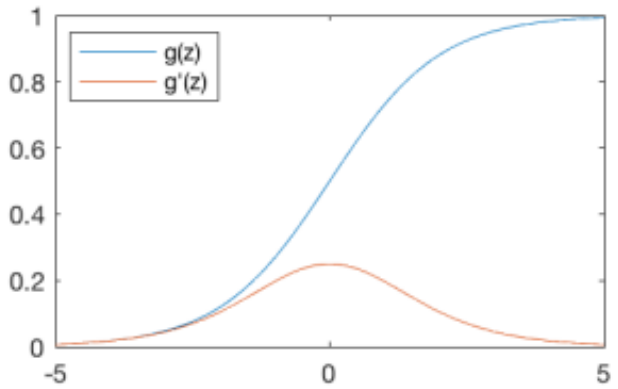
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

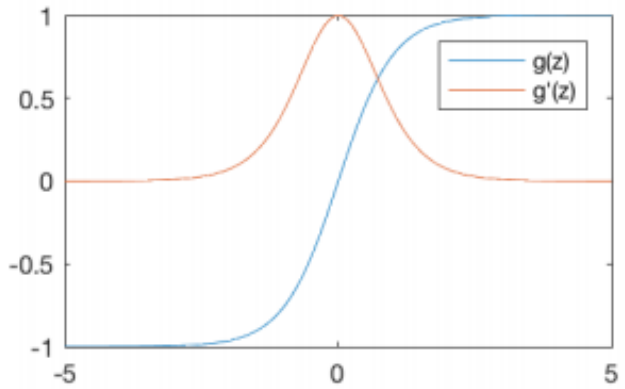


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

tf.nn.sigmoid(z)

Hyperbolic Tangent

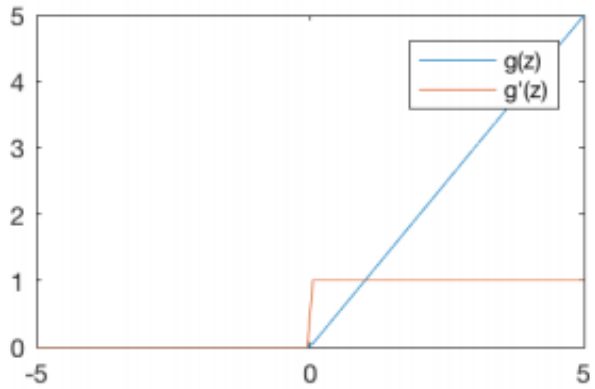


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

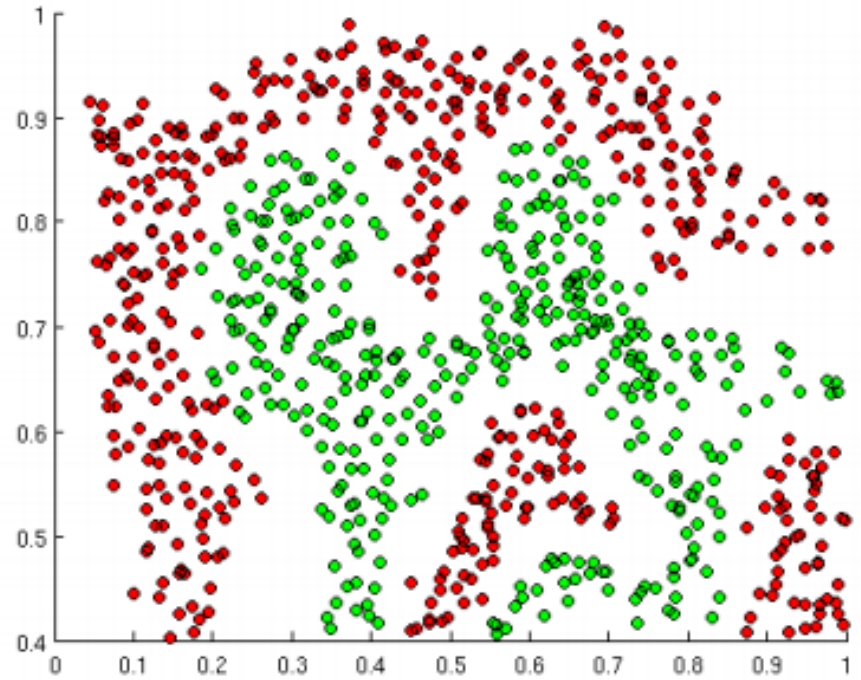
tf.nn.relu(z)

NOTE: All activation functions are non-linear

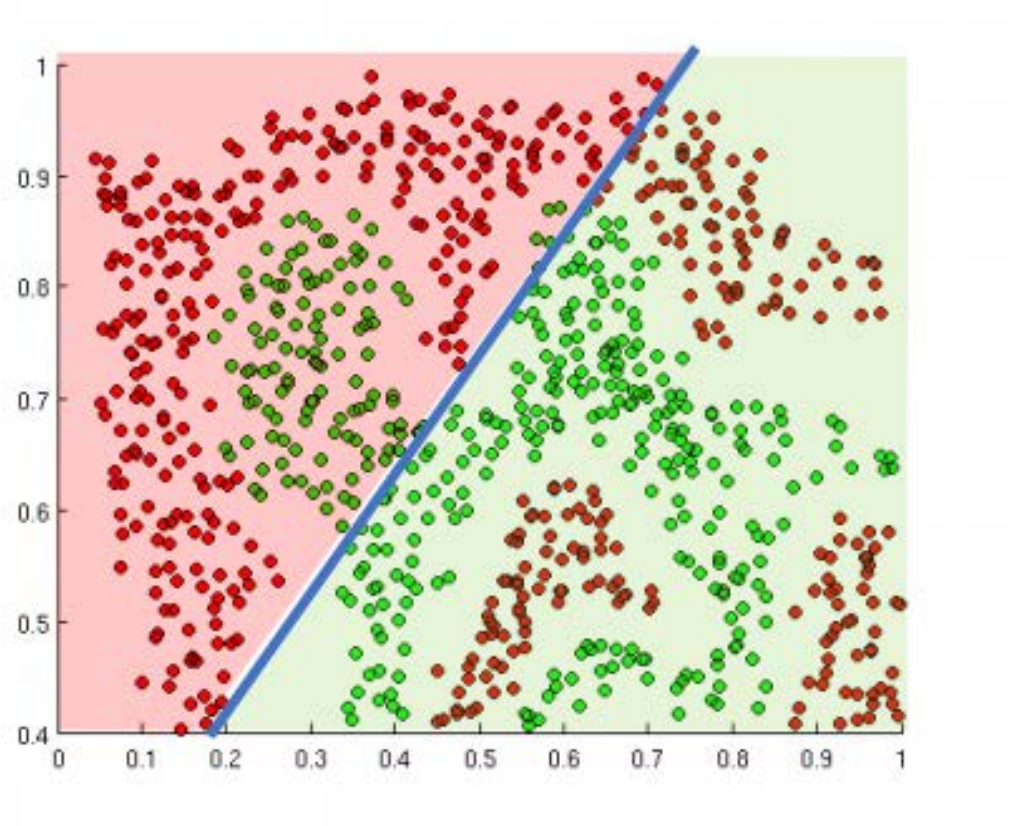
Why do we need activation functions?

- To introduce non-linearities into the network

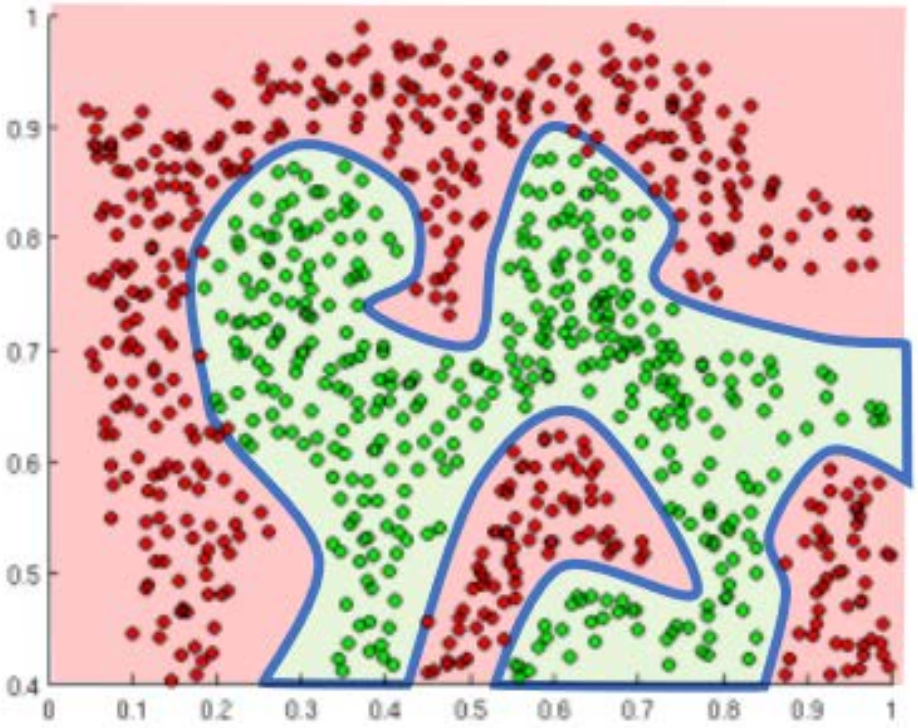
How to build a neural network to distinguish red and green points?



Linear vs Non-linear activation function

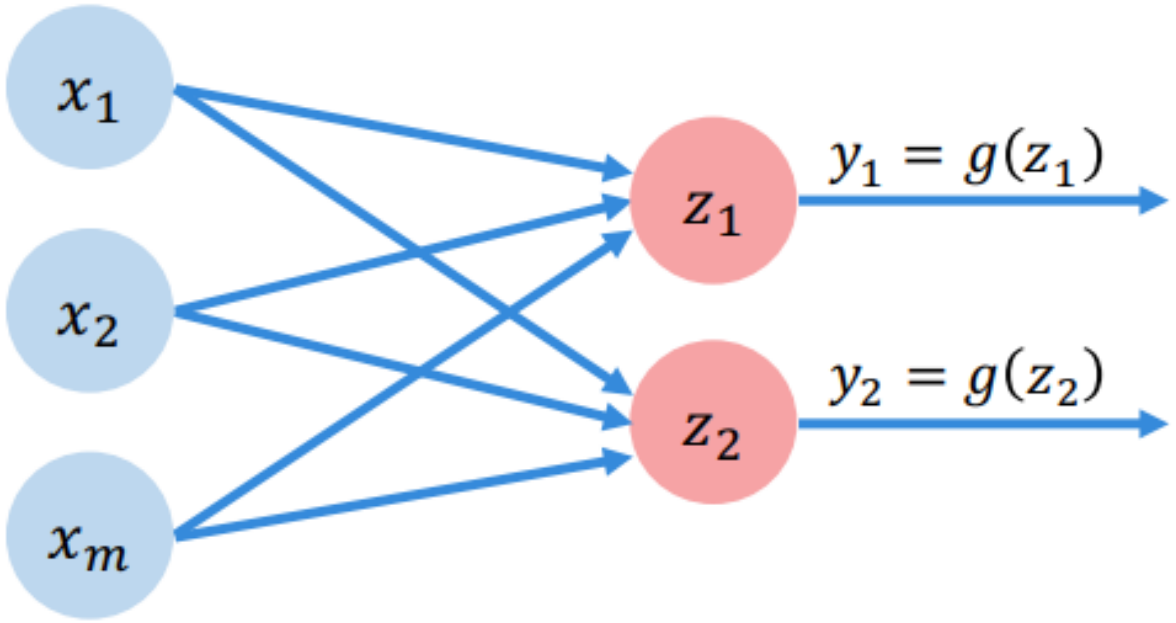


Linear activations produce linear decisions no matter the network size



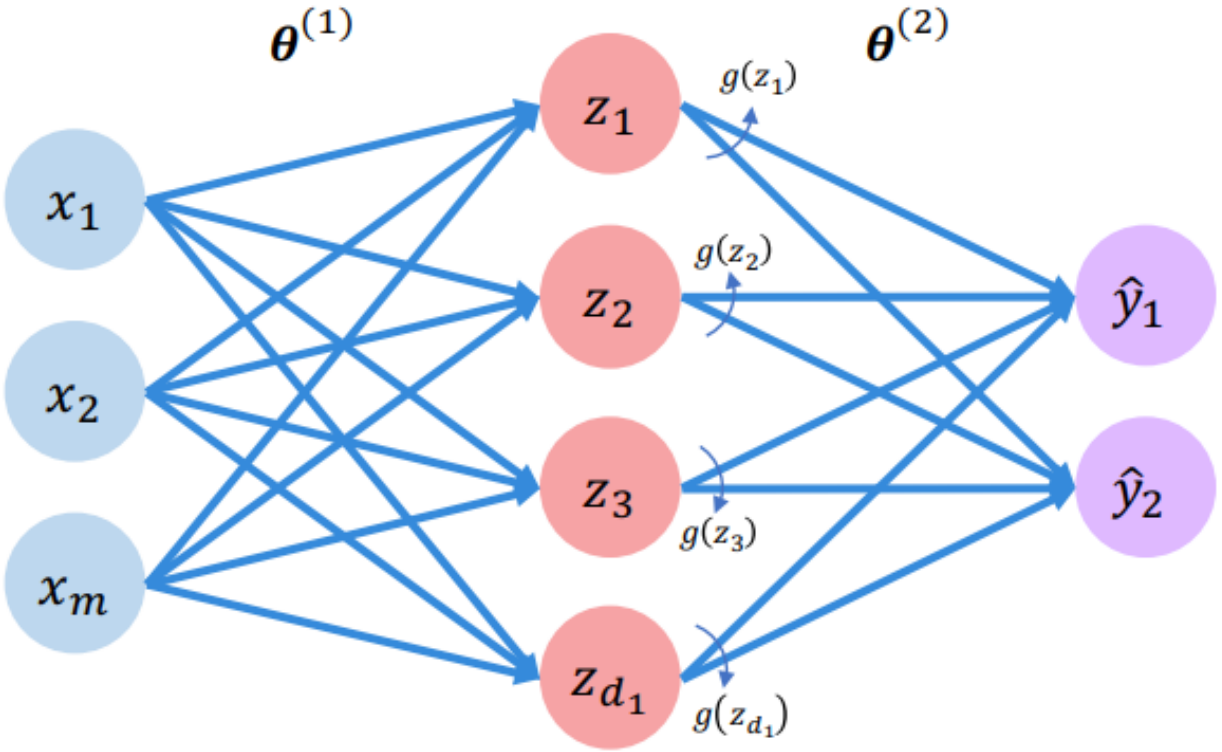
Non-linearities allow us to approximate arbitrarily complex functions

Multi-output perceptron



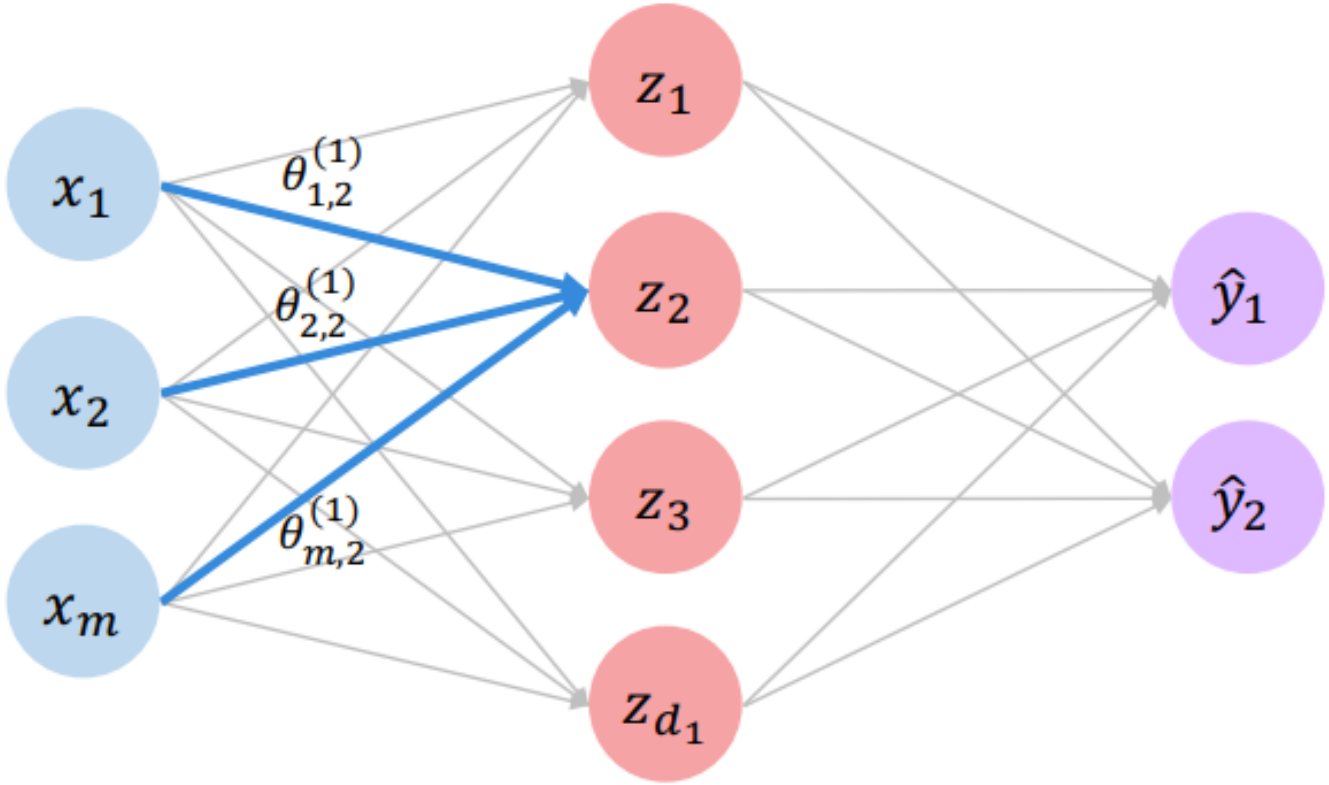
$$z_i = \theta_{0,i} + \sum_{j=1}^{m} x_j \, \theta_{j,i}$$

Single hidden layer neural network



$$z_i = \theta_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, \theta_{j,i}^{(1)} \qquad \hat{y}_i = \theta_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, \theta_{j,i}^{(2)}$$

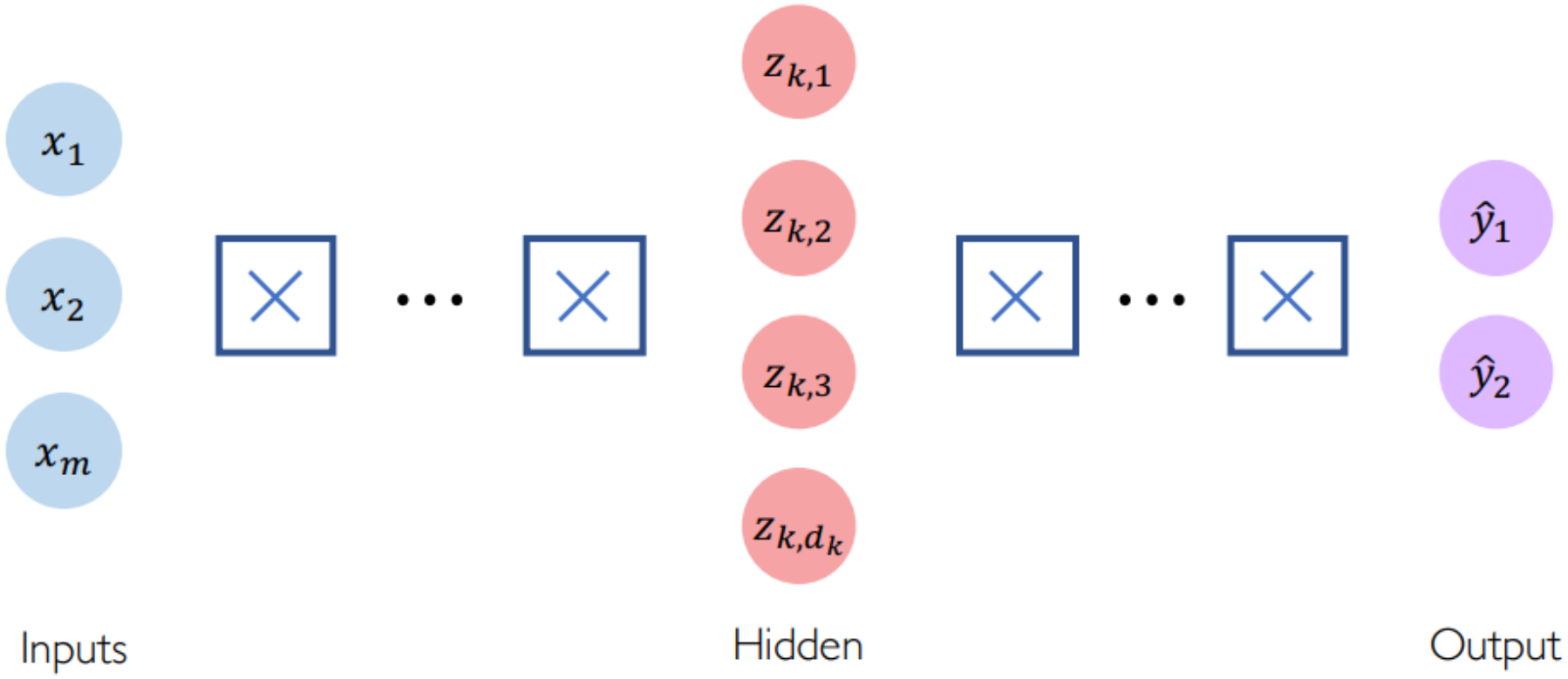
Single hidden layer neural network



$$z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, \theta_{j,2}^{(1)}$$

$$= \theta_{0,2}^{(1)} + x_1 \, \theta_{1,2}^{(1)} + x_2 \, \theta_{2,2}^{(1)} + x_m \, \theta_{m,2}^{(1)}$$

Deep Neural Network



$$z_{k,i} = \theta_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, \theta_{j,i}^{(k)}$$

Example Problem

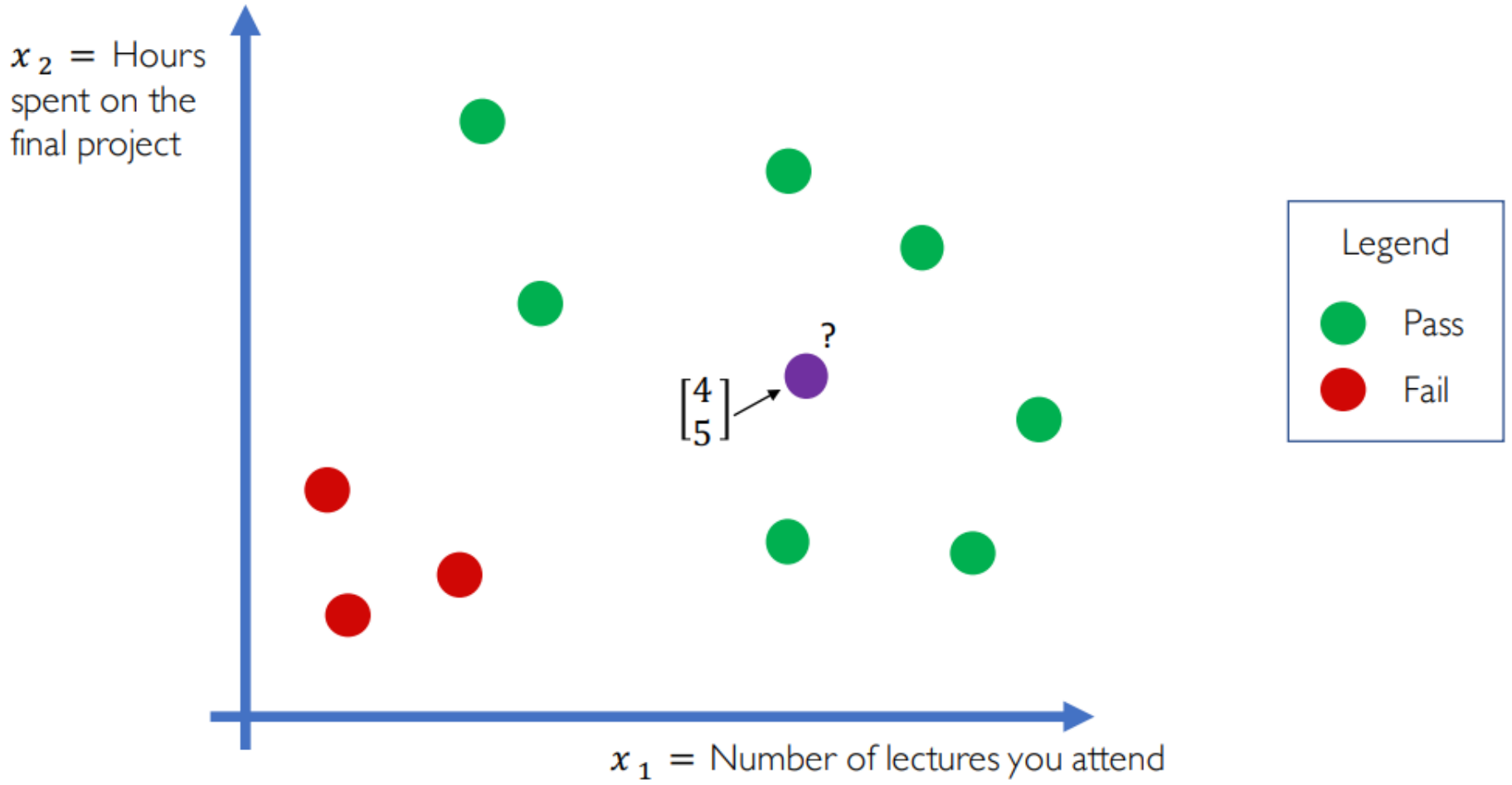
Will I pass this class?

Let's start with a simple two feature model

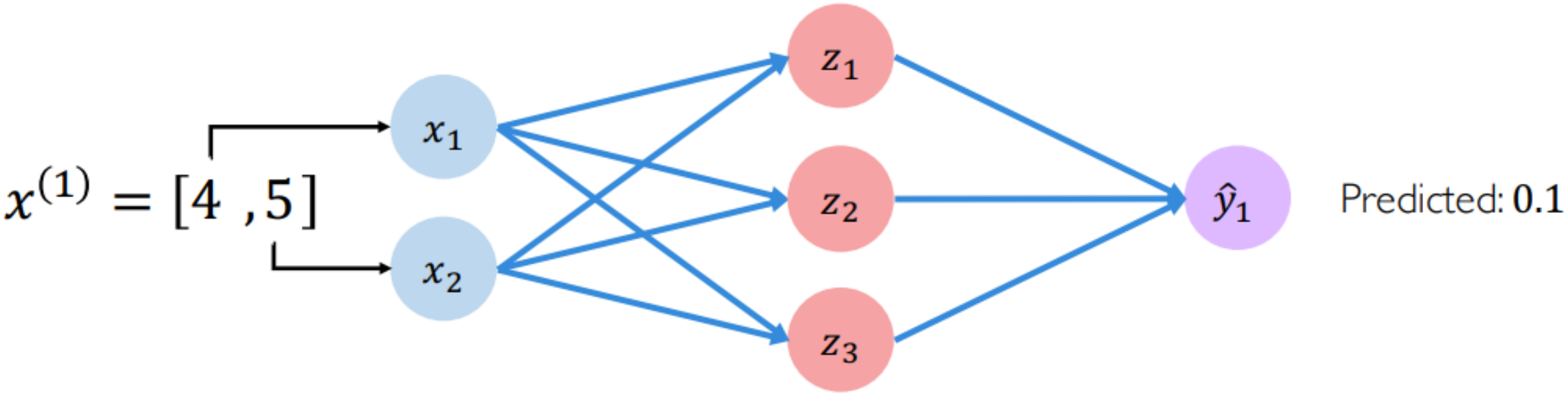
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

Example Problem

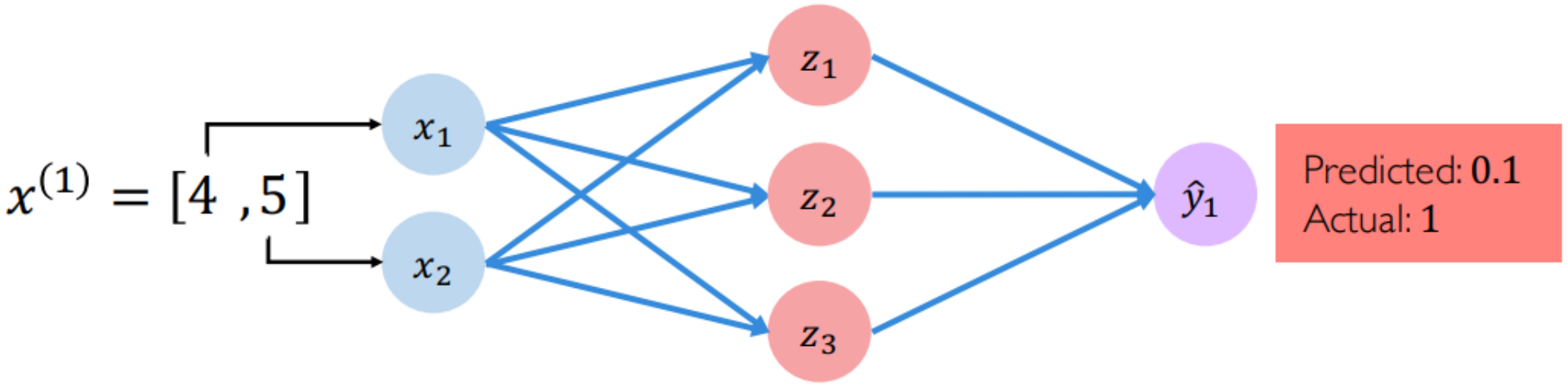


Example Problem



Quantifying loss

*The **loss** of our network measures the cost incurred from incorrect predictions*



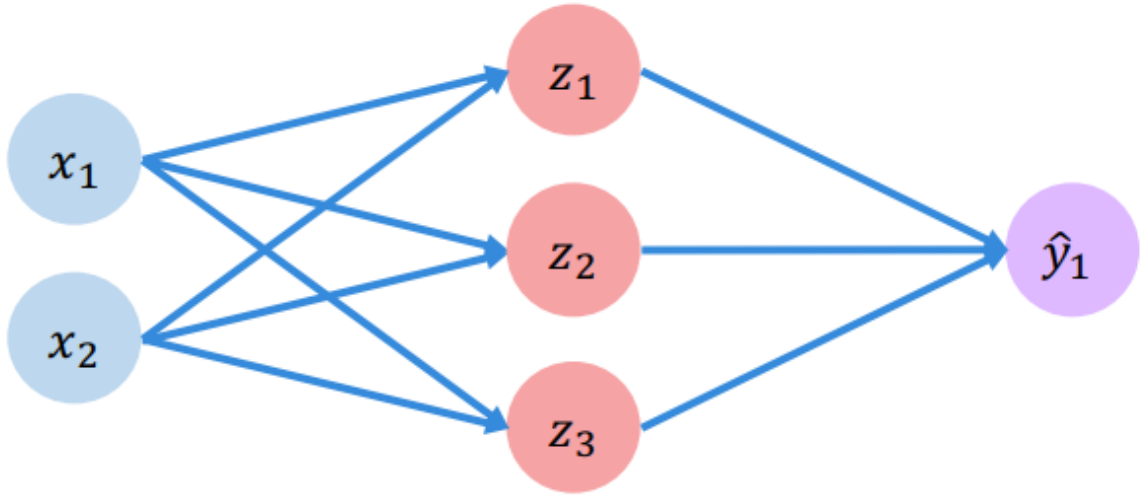
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

*The **empirical loss** measures the total loss over our entire dataset*

$$\boldsymbol{x} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$		y
0.1	✖	1
0.8	✖	0
0.6	✔	1
⋮		⋮

Also known as:
- Objective function
- Cost function
- Empirical Risk

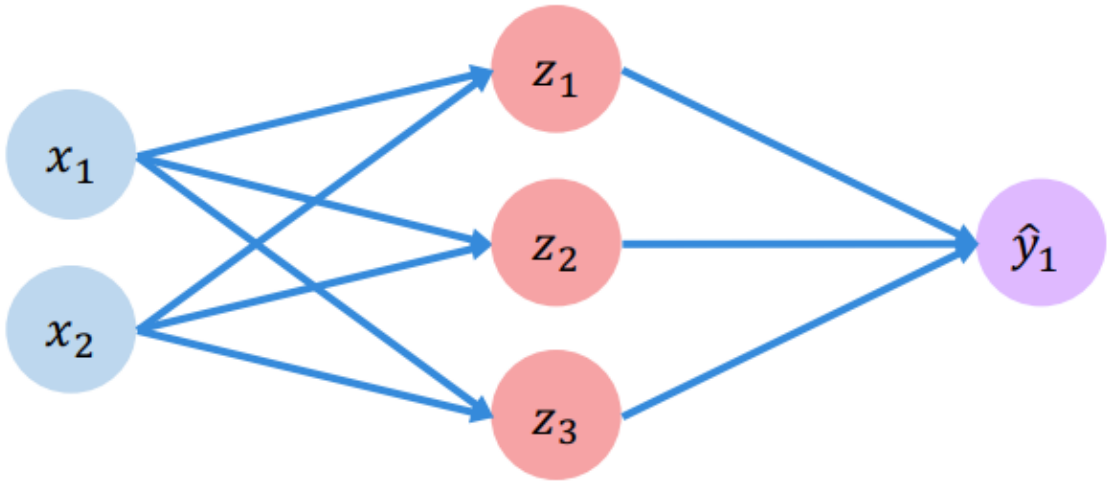
$$\boldsymbol{J(\theta)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

Predicted Actual

Binary Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

$$x = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

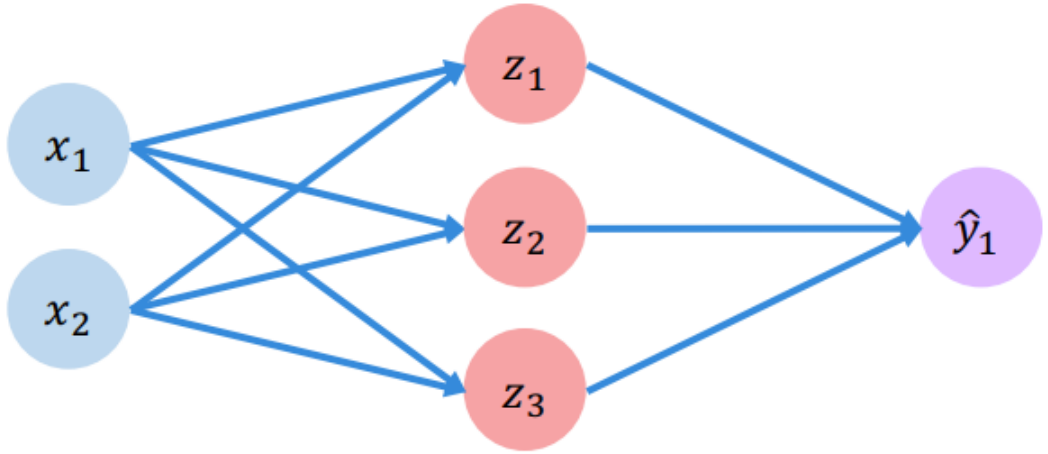
$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big)$$

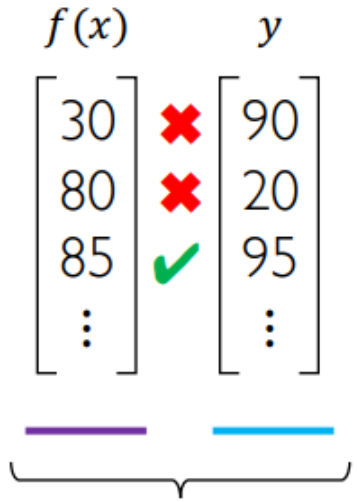
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*

$$\boldsymbol{x} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$





$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)};\theta\right)\right)^2$$

Actual Predicted

```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

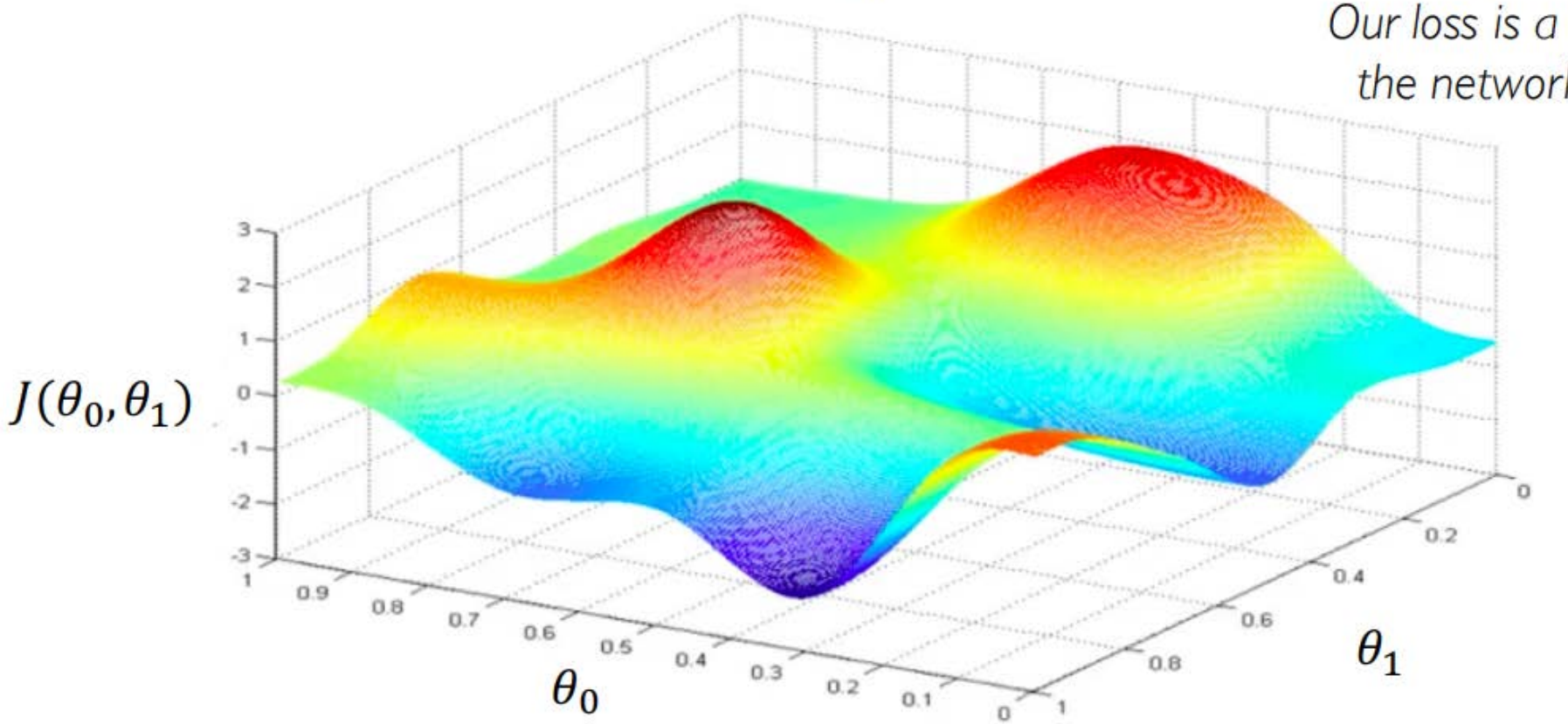
Remember:

$$\boldsymbol{\theta} = \left\{\theta^{(0)}, \theta^{(1)}, \dots\right\}$$

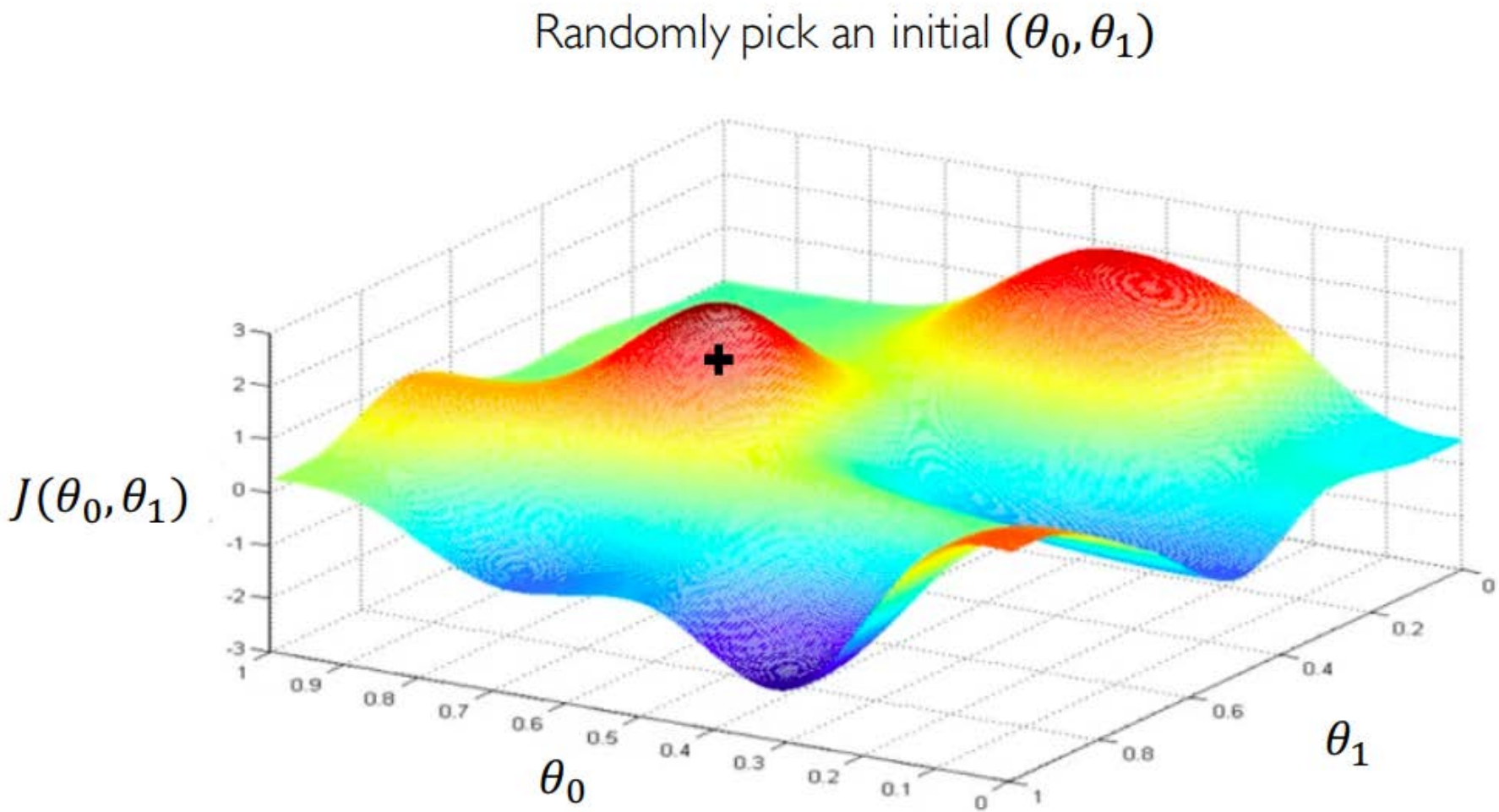
Loss Optimization

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

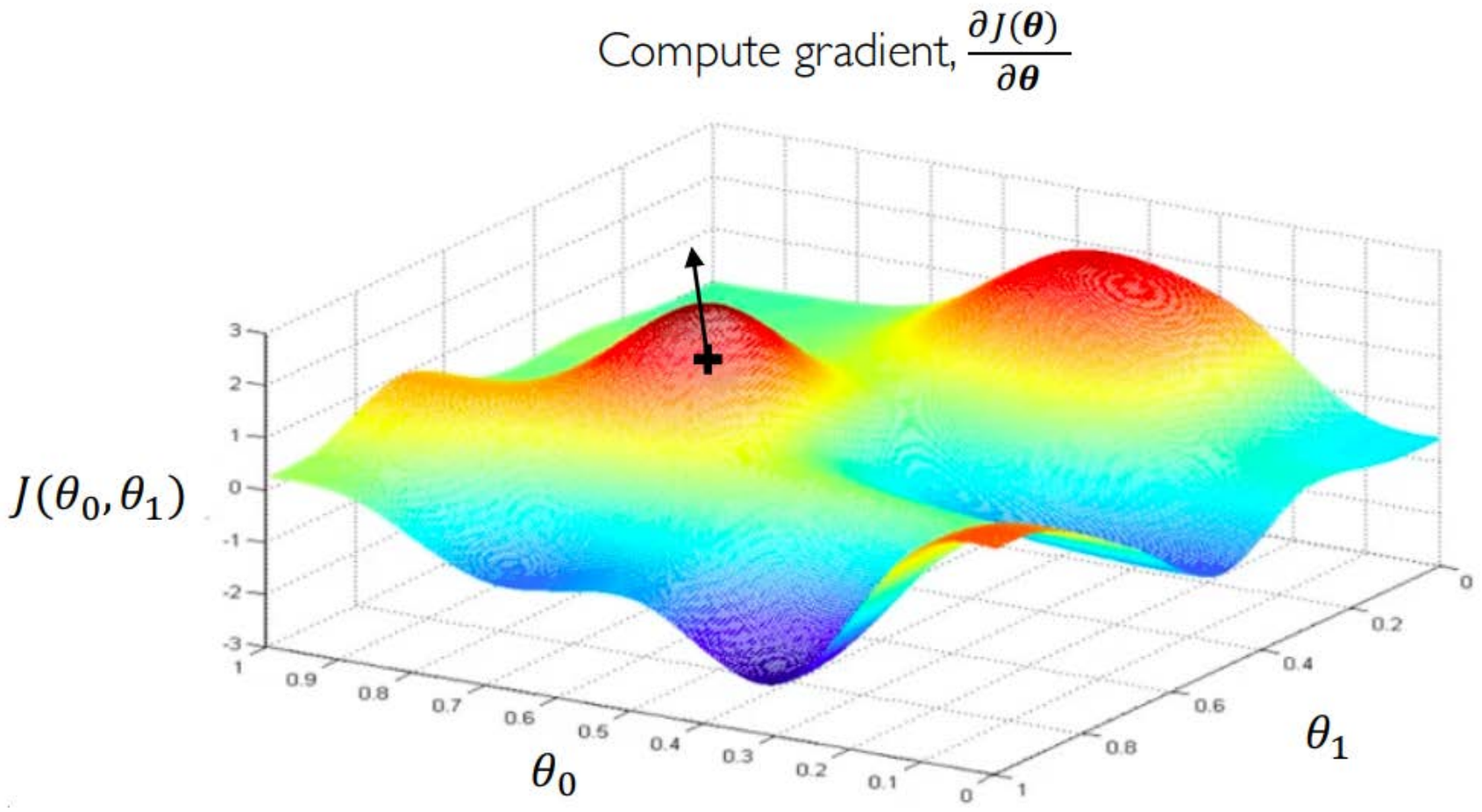
Remember:
Our loss is a function of the network weights!



Loss Optimization

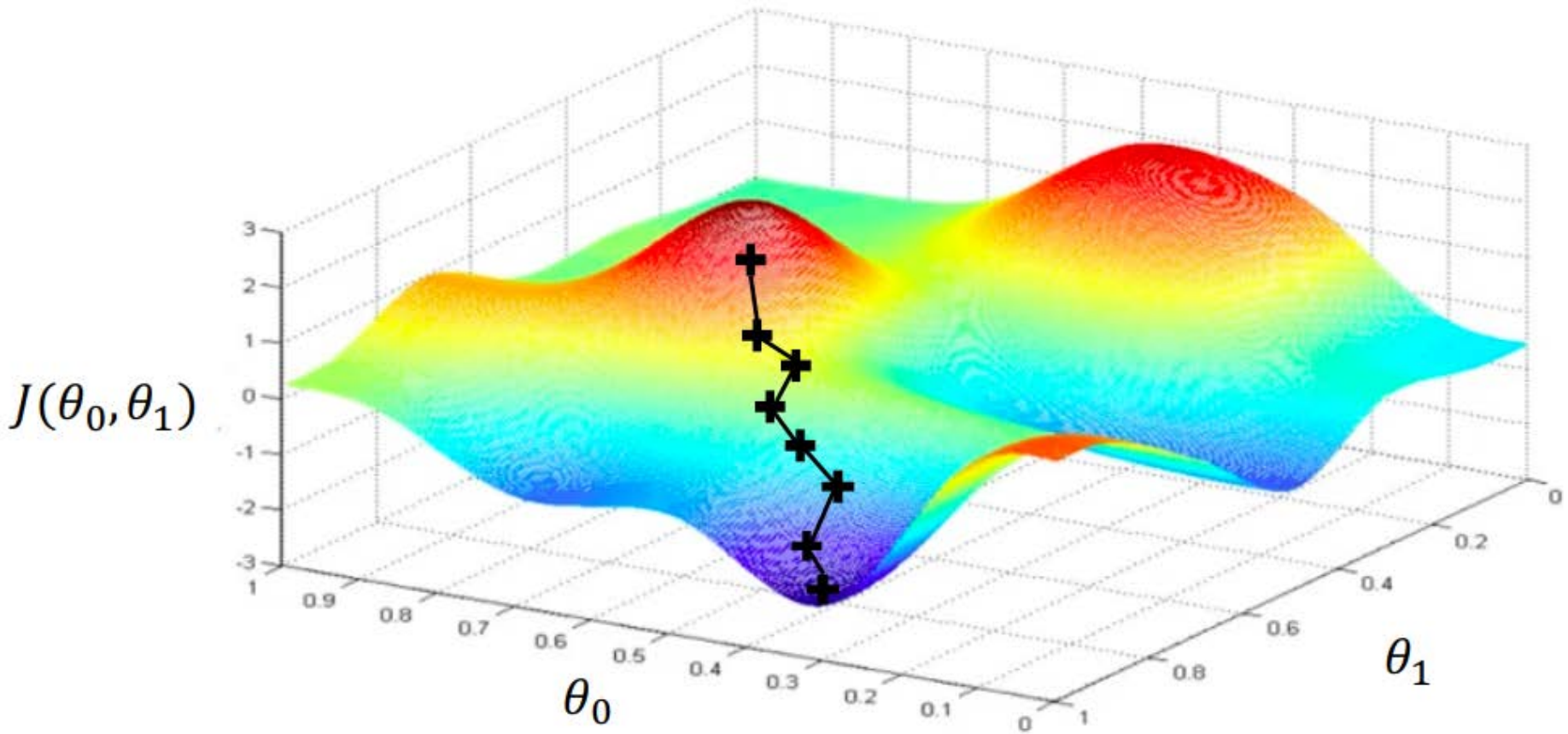


Loss Optimization



Gradient Descent

Repeat until convergence



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

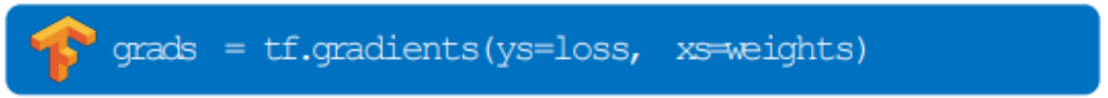
```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

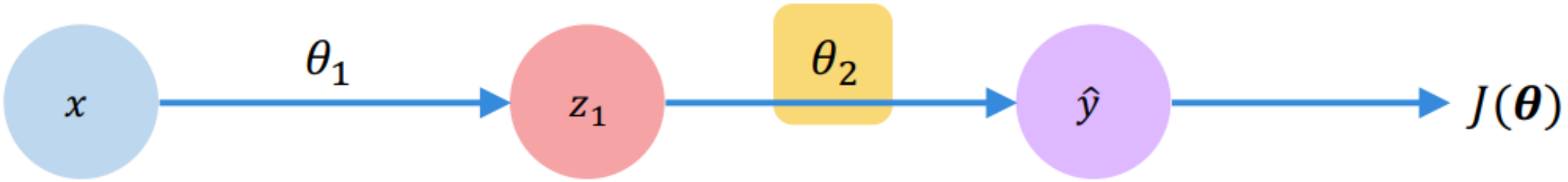
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

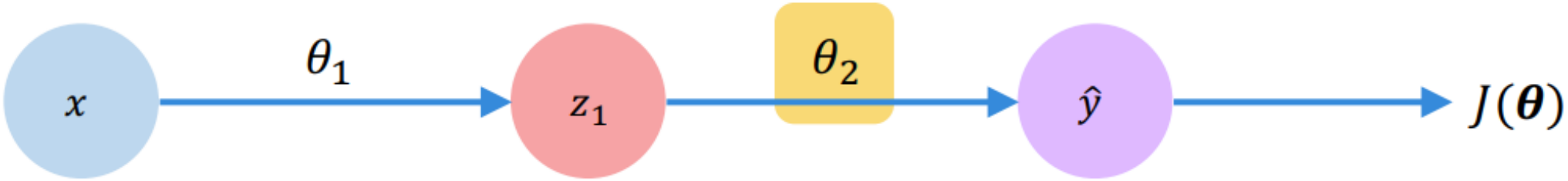


Computing Gradients: Backpropagation



How does a small change in one weight (ex. $\boldsymbol{\theta_2}$) affect the final loss $\boldsymbol{J(\theta)}$?

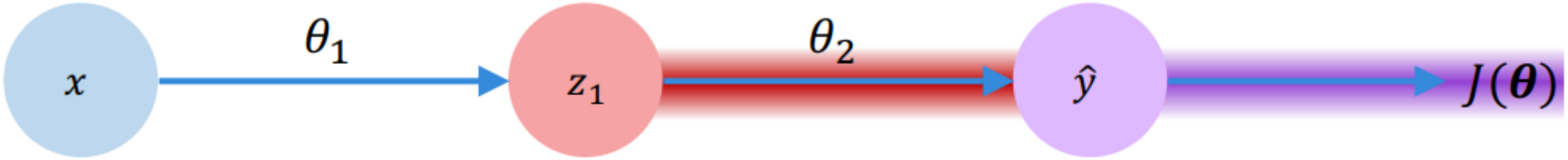
Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} =$$

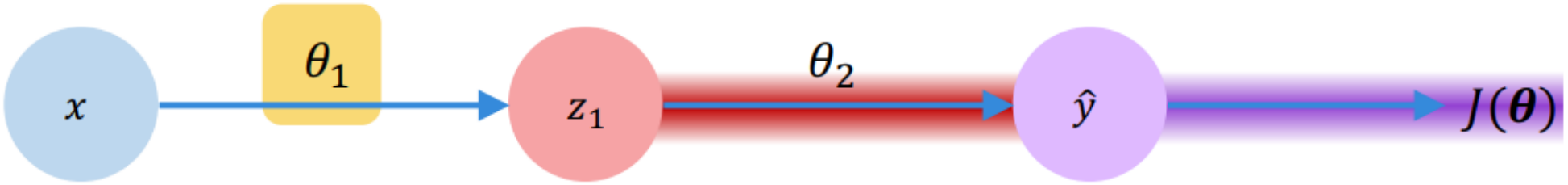
Let's use the chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_2}$$

Computing Gradients: Backpropagation

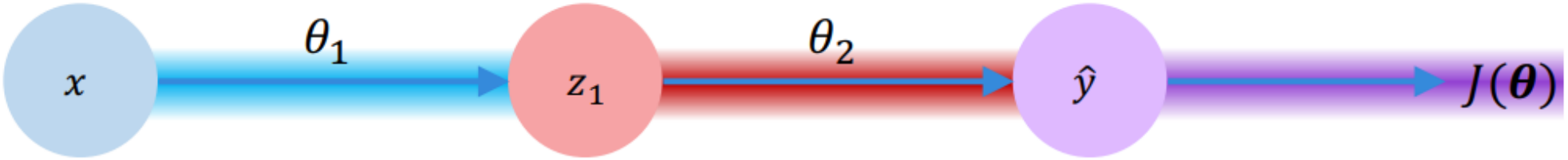


$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_1}$$

Apply chain rule!

Apply chain rule!

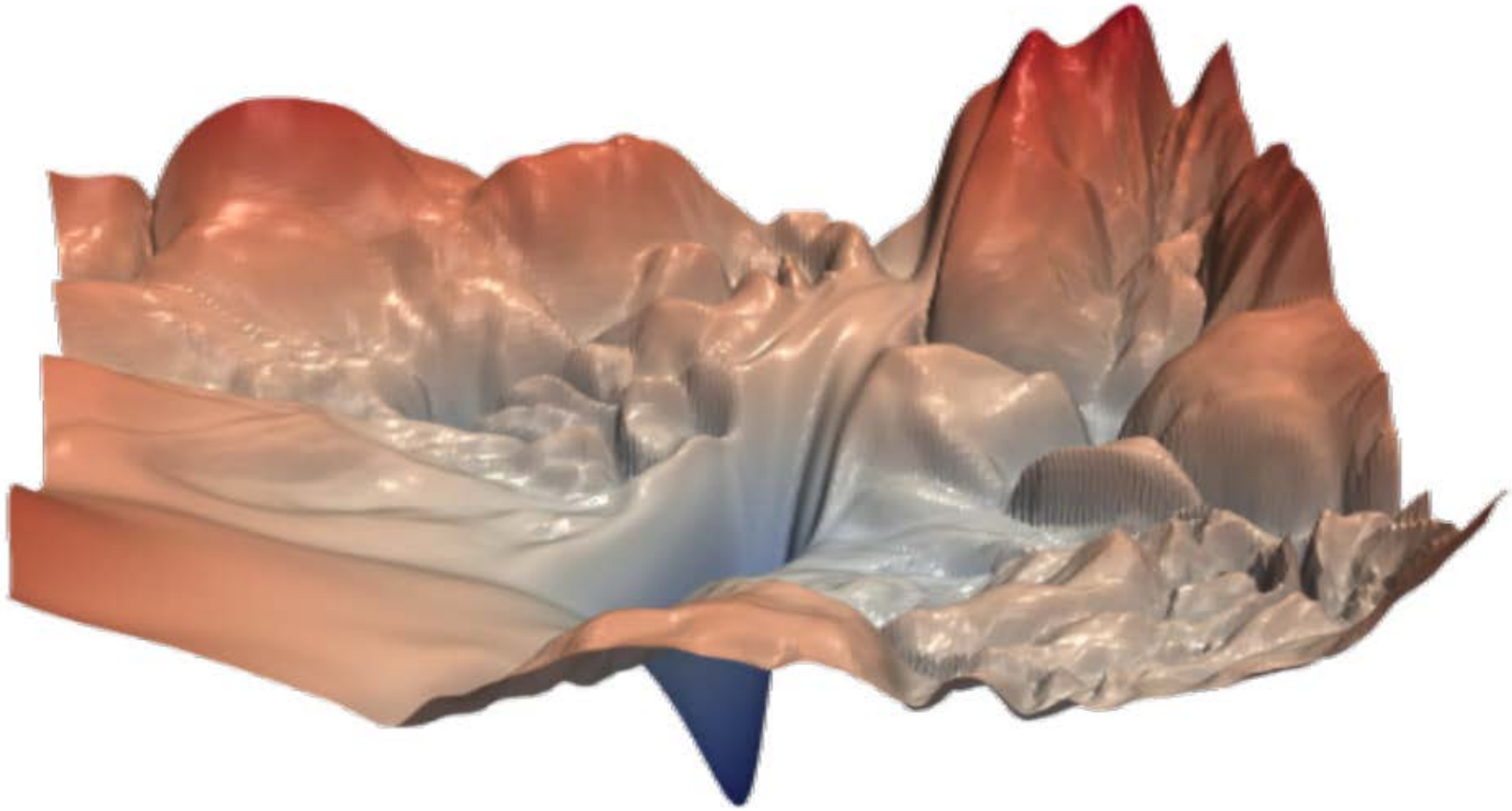
Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

Repeat this for ***every weight in the network*** *using gradients from later layers*

Training Neural Networks is Difficult



Hao Li, Zheng Xu, Gavin Taylor, Tom Goldstein, Visualizing the Loss Landscape of Neural Nets, 6th International Conference on Learning Representations, ICLR 2018

Loss functions can be difficult to optimize

Remember:

Optimization through gradient descent

$$\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$

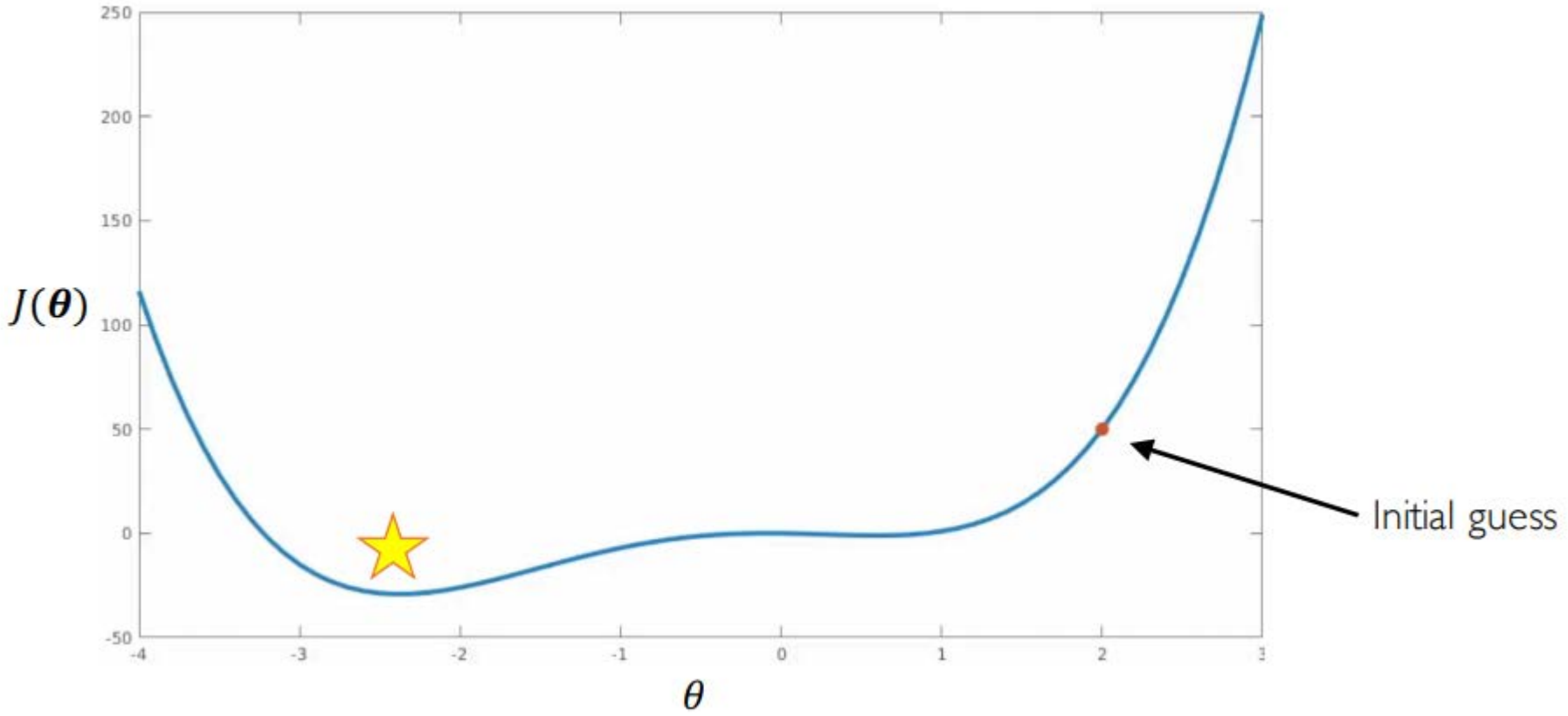
How can we set the learning rate?

Setting the learning rate

Small learning rates
converges slowly and gets stuck in false local minima

Large learning rates
overshoot, become unstable and diverge

Stable learning rates
converge smoothly and avoid local minima

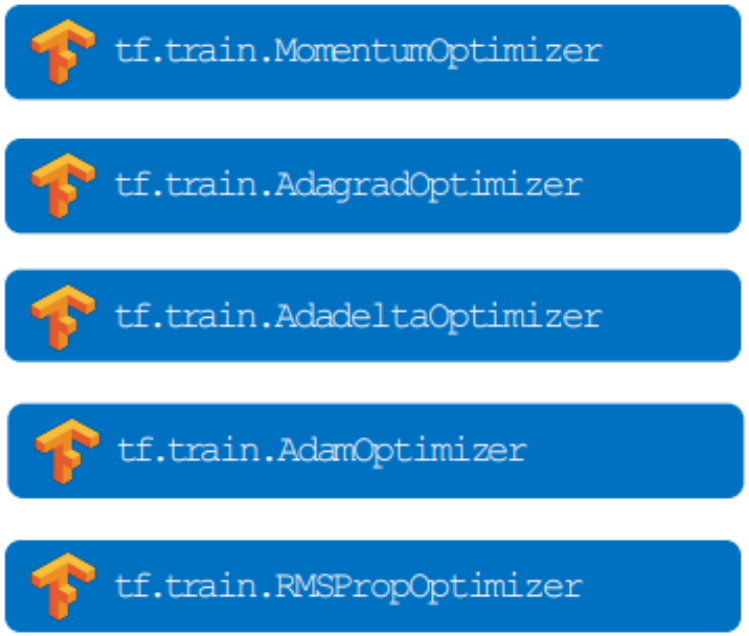


Adaptive Learning Rates

- Design an adaptive learning rate that adapts to the landscape
- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - How large the gradient is
  - How fast learning is happening
  - Etc..

Adaptive Learning Rate Algorithms

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Hinton's Coursera lecture (unpublished)

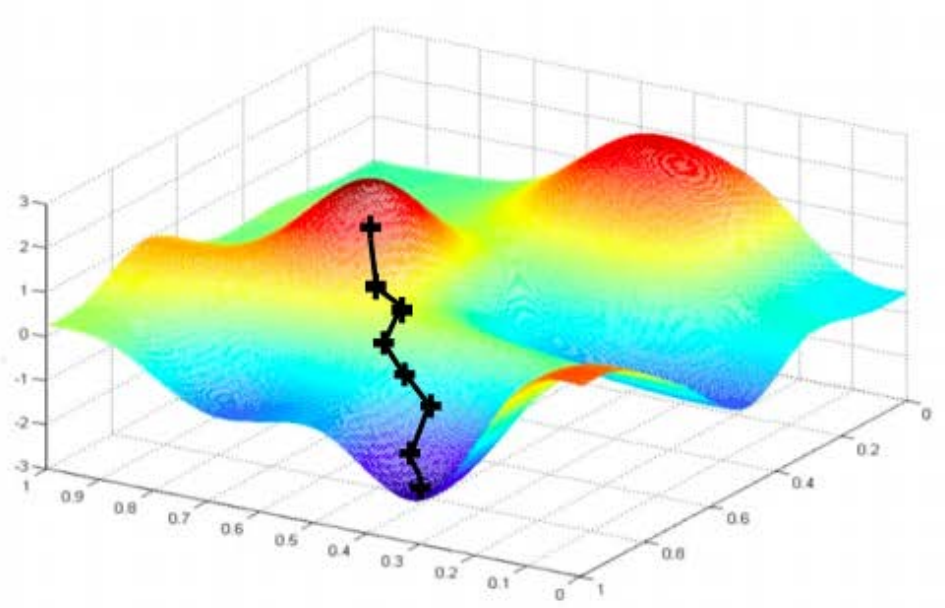
http://ruder.io/optimizing-gradient-descent/

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

Can be very computational to compute!

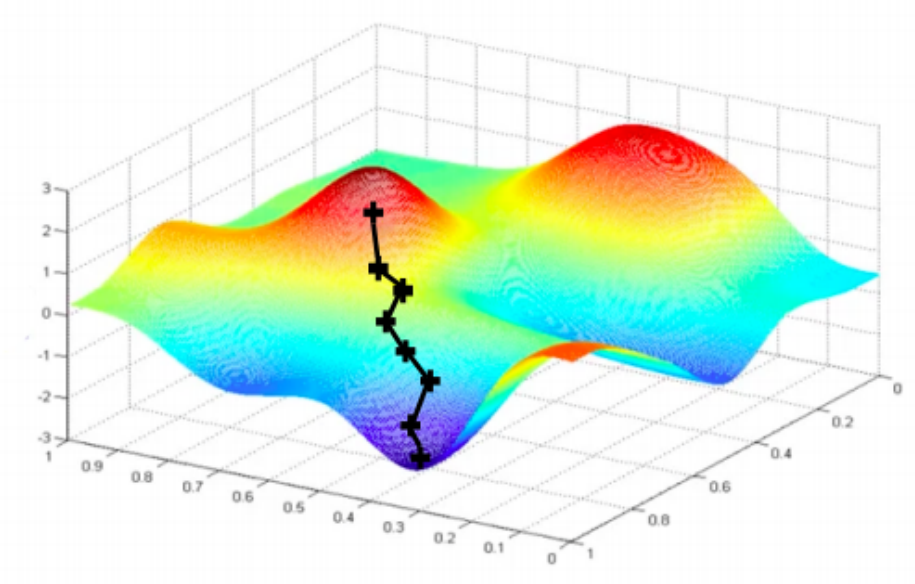


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights

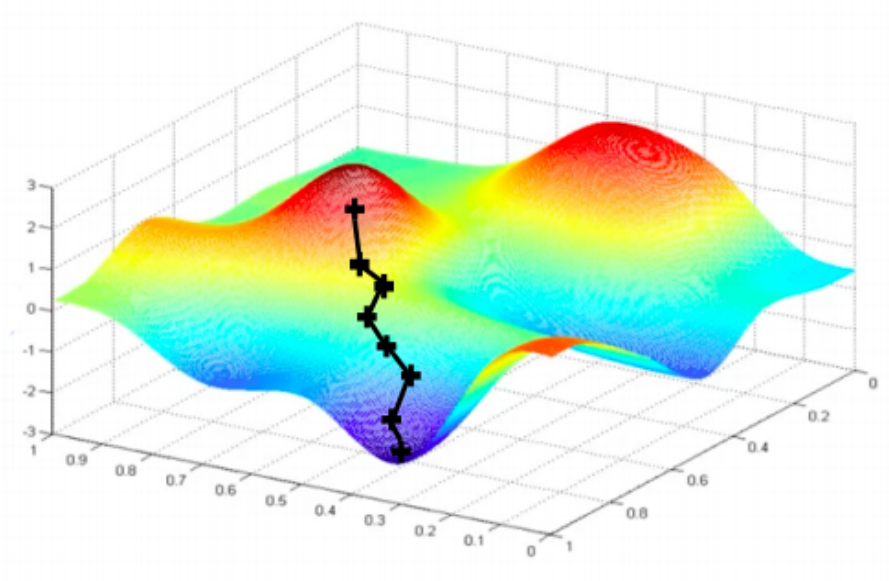
Easy to compute but **very noisy** (stochastic)!



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Mini-batches

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates

- Mini-batches lead to fast training
  - Can parallelize computation + achieve significant speed increases on GPU’s

Terminology

- **Number of iterations**: The number of times the gradient is estimated and the parameters of the neural network are updated using a batch of training instances
- **Batch size:** Number of training instances used in one iteration
- **Mini-batch:** When the total number of training instances N is large, a small number of training instances B<<N which constitute a mini-batch can be used in one iteration to estimate the gradient of the loss function and update the parameters of the network
- **Epoch:** It takes n = N/B iterations to use the entire training data once. That is called an epoch. The total number of times the parameters get updates is (N/B)*E, where E is the number of epochs.

https://www.quora.com/What-are-the-meanings-of-batch-size-mini-batch-iterations-and-epoch-in-neural-networks

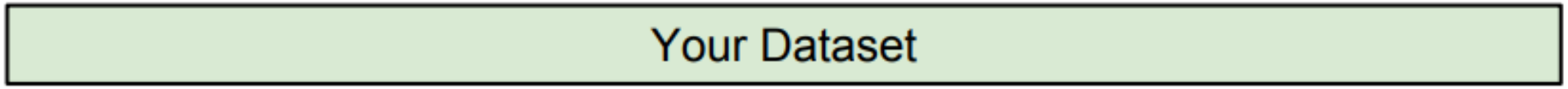
Three modes of gradient descent

- Batch mode: N=B, one epoch is same as one iteration.
- Mini-batch mode: 1<B<N, one epoch consists of N/B iterations.
- Stochastic mode: B=1, one epoch takes N iterations.

Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the data

BAD: always works perfectly on training data



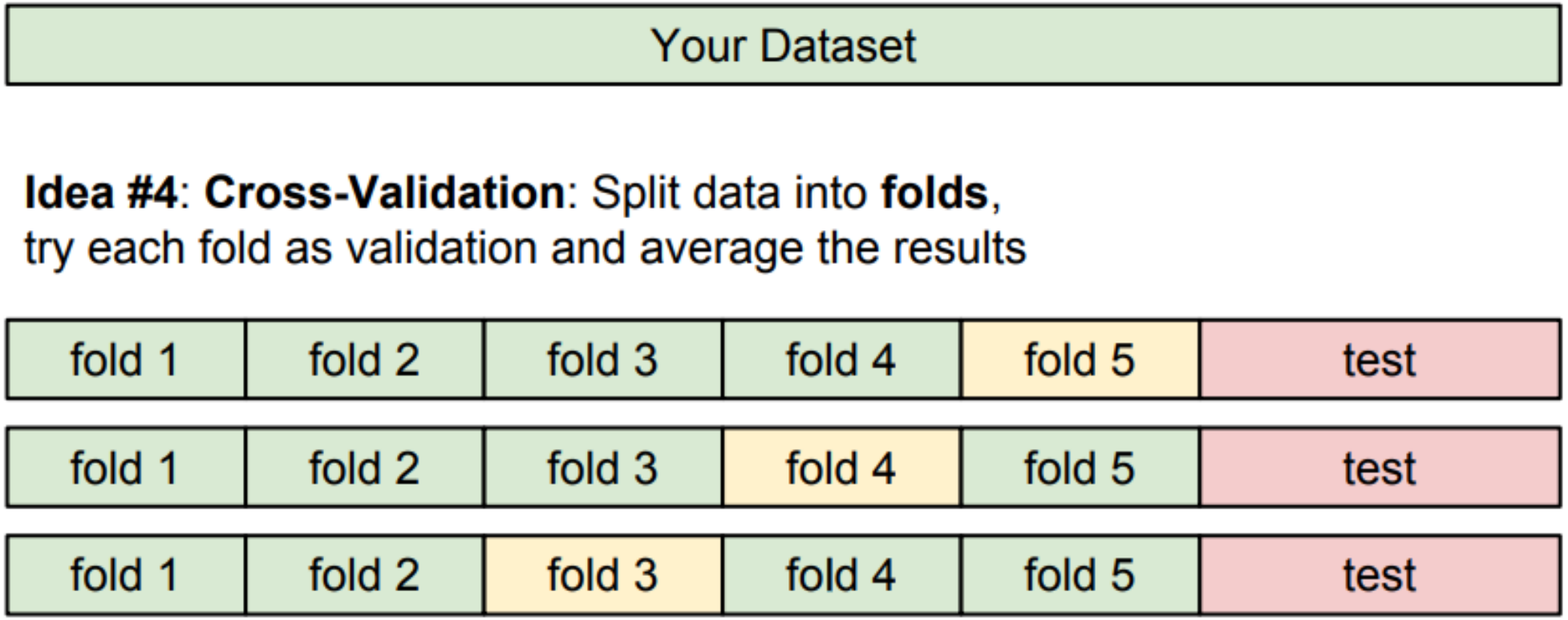
Idea #2: Split data into **train** and **test**, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data

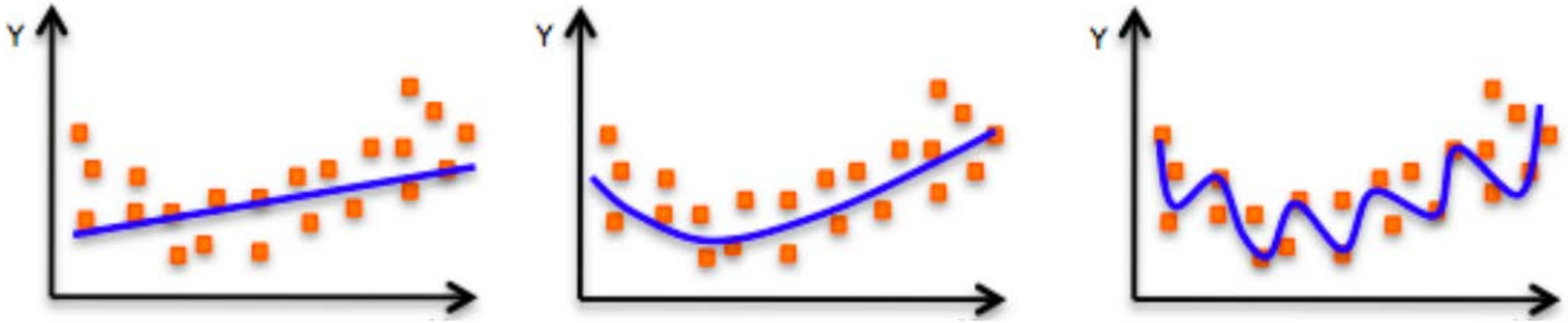
Idea #3: Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test

Better!

Setting Hyperparameters



The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

High bias

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

High variance

High Bias vs High Variance

- High Bias (high training set error)
  - Use a bigger network
  - Try different optimization algorithms
  - Train longer
  - Try different architecture

- High Variance (high validation set error)
  - Collect more data
  - Use regularization
  - Try different NN architecture

Regularization

- What is it?
  - Technique that constrains our optimization problem to discourage complex models
- Why do we need it?
  - Improve generalization of our model on unseen data

Regularization 1: Penalizing weights

- Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty)

- Neural networks have thousands (or millions of parameters)
  - Danger of overfitting

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \lambda\Omega(\theta)$$

Regularization 1: L1 and L2 regularization

- L2 regularization (most popular)

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2} \sum_{l} \|\theta_l\|^2$$

- L1 regularization

$$\theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2} \sum_{l} \|\theta_l\|$$

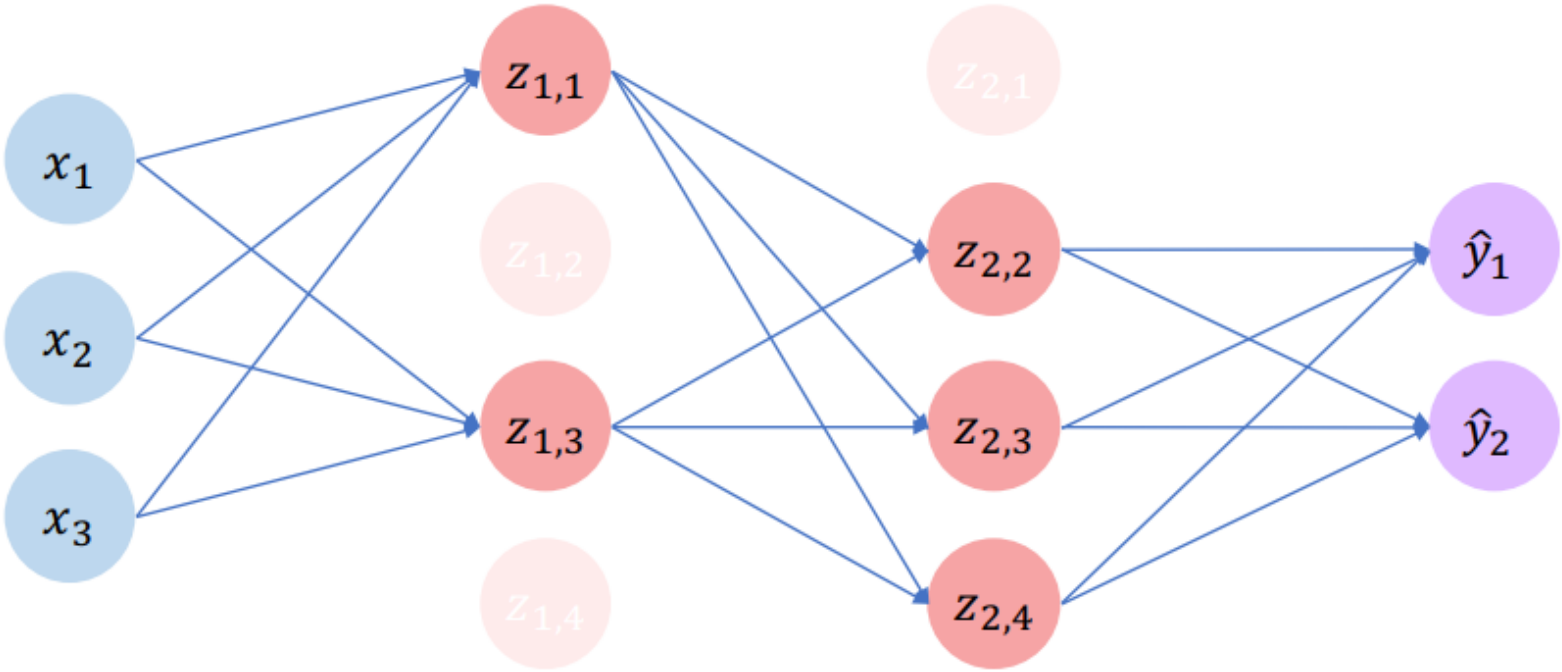
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Regularization 3: Dropout

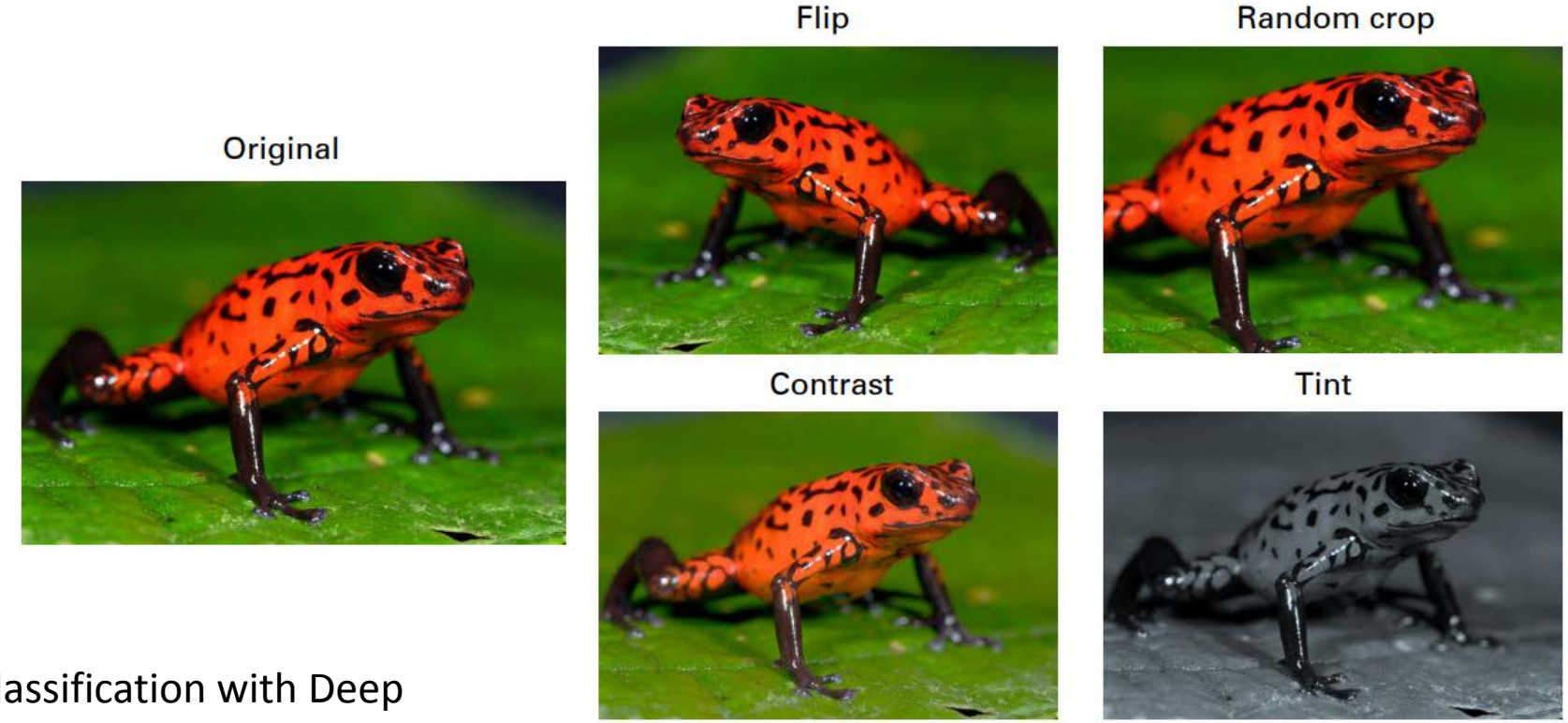
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.nn.dropout(hiddenLayer, p=0.5)
```



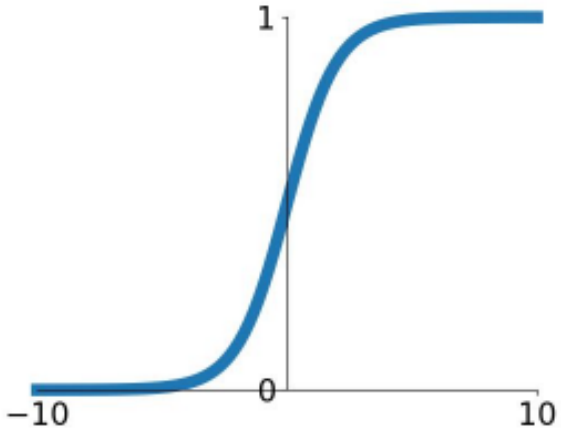
Regularization 4: Data Augmentation

- Adding more data reduces overfitting
- Data collection and labelling is expensive
- Solution: Synthetically increase training dataset



Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks, 2012

Difference between Activation Functions

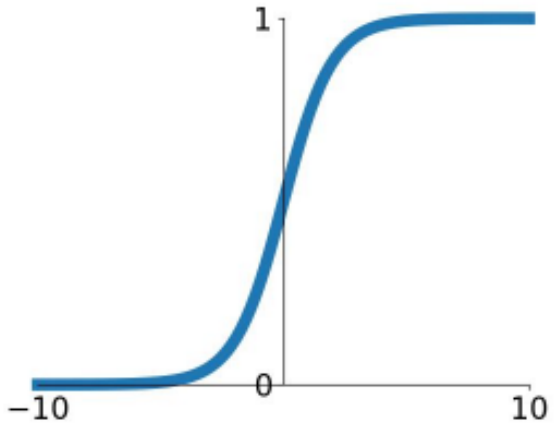


Sigmoid

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Difference between Activation Functions



Sigmoid

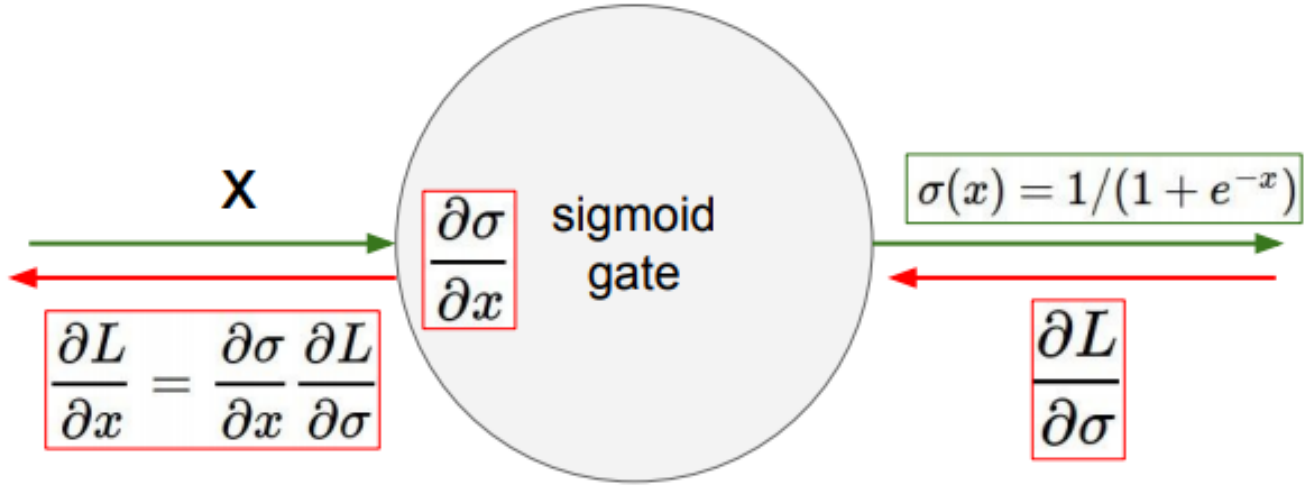
$$\sigma(x) = 1/(1 + e^{-x})$$

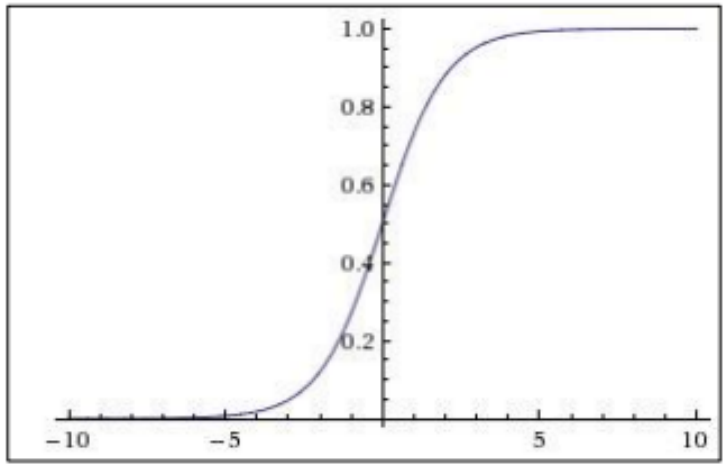
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

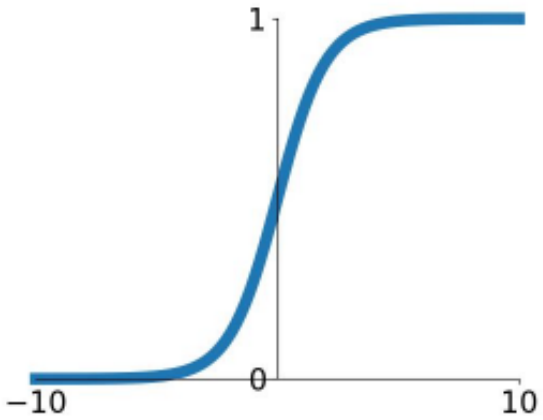
Difference between Activation Functions





What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Difference between Activation Functions



Sigmoid

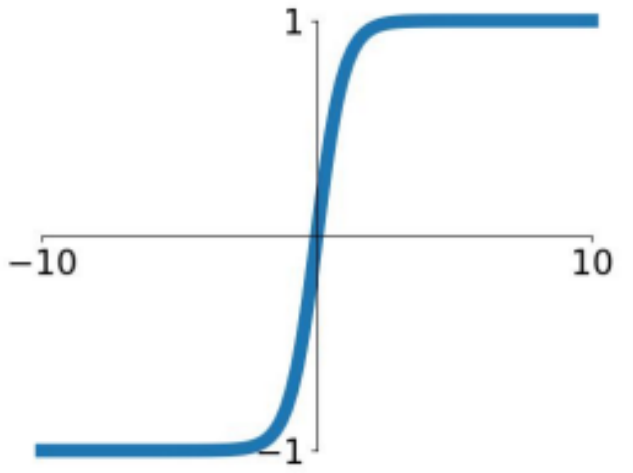
$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Difference between Activation Functions

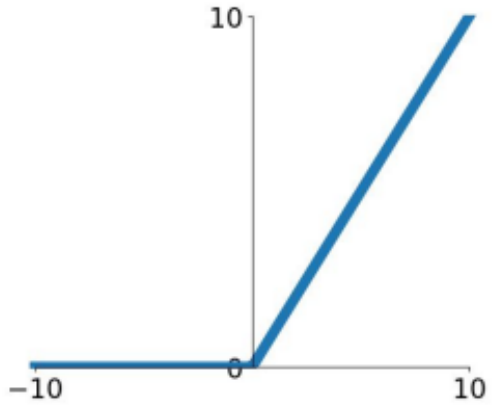


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Y. LeCun, I. Kanter, and S.A.Solla: "Second-order properties of error surfaces: learning time and generalization", Advances in Neural Information Processing Systems, vol. 3, pp. 918-924, 1991

Difference between Activation Functions



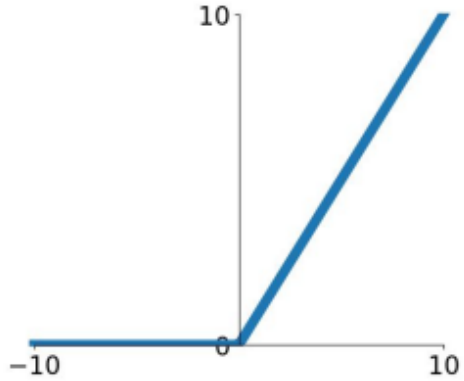
ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Difference between Activation Functions



ReLU
(Rectified Linear Unit)

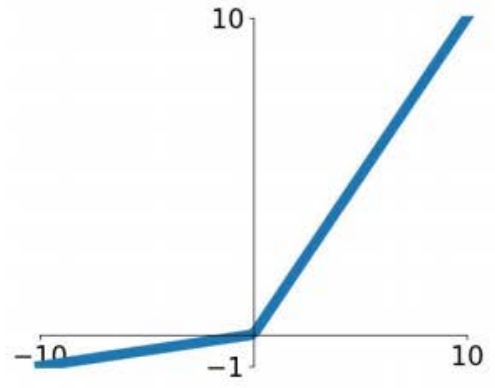
- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Difference between Activation Functions



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

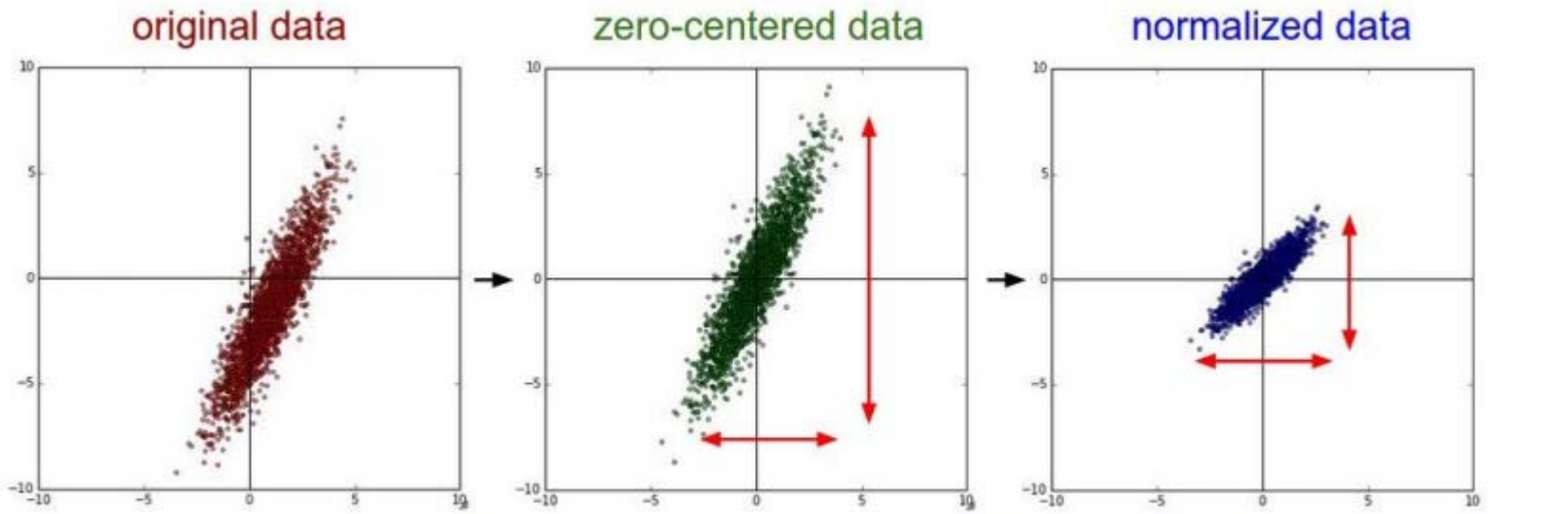
Leaky ReLU

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV) (ICCV '15). IEEE Computer Society, Washington, DC, USA, 1026-1034

Normalizing inputs

- Normalized inputs helps for the learning process
- Subtract mean and normalize variances
- Use the same mean and variance to normalize the test (you want them to go through the same transitions)



Batch Normalization

- Similar to input normalization, you can normalize the values in the hidden layer
- Two additional parameters to be trained

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

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

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

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

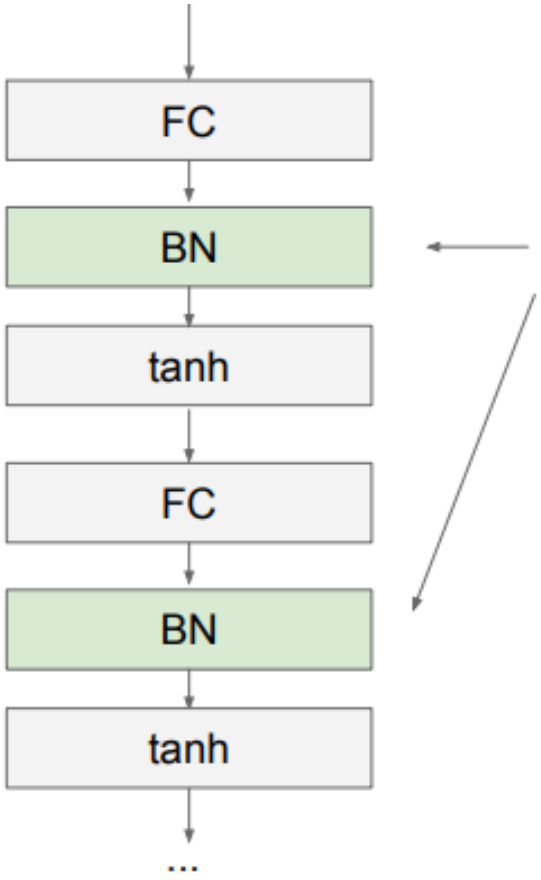
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

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

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15), Francis Bach and David Blei (Eds.), Vol. 37. JMLR.org 448-456

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Vanishing/exploding gradients

- Vanishing gradients: As we get back deep in the neural network, gradient tends to get smaller through hidden layers
  - In other words, neurons in the earlier layers learn much more slowly than neurons in later layers

- Exploding gradints: Gradients get much larger in earlier layers, unstable gradient

- How you initialize the network weights is important!!

Weight initialization

- Initialize with all 0s or 1s?
  - Behaves like a linear model, hidden units become symmetric
- Traditionally weights of a neural network were set to small random numbers
- Weight initialization is a whole field of study, careful weight initialization can speep up the learning process

https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/
https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94

Weight Initialization (Best practices)

- For tanh(z) (also called Xavier initialization)

$$\sqrt{\frac{1}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(1/size\_l\text{-}1)$$

- For RELU(z)

$$\sqrt{\frac{2}{size^{[l-1]}}}$$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2/size\_l\text{-}1)$$

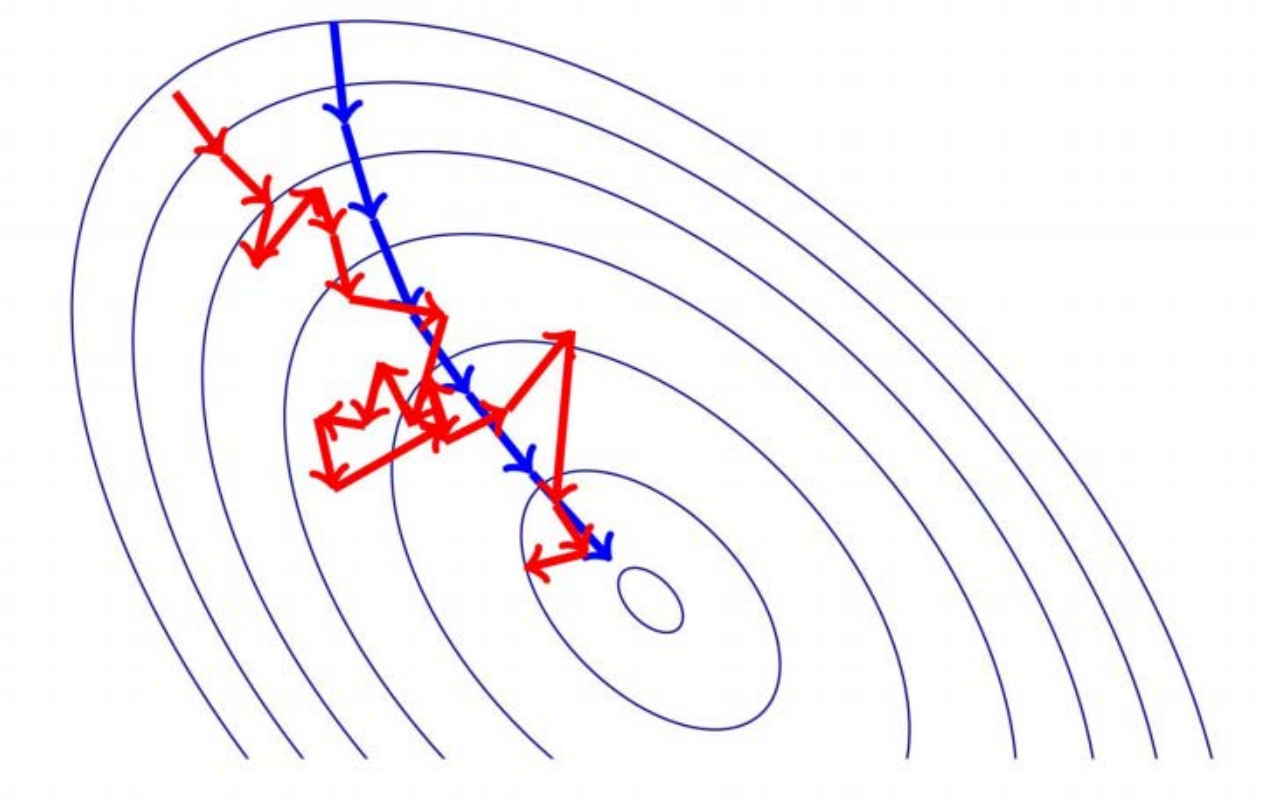
Understanding the difficulty of training deep feedforward neural networks Glorot and Bengio, 2010 (Xavier initialization)
Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification He et al., 2015

Proper initialization is an active area of research…

- **Understanding the difficulty of training deep feedforward neural networks** Glorot and Bengio, 2010
- **Exact solutions to the nonlinear dynamics of learning in deep linear neural networks** Saxe et al, 2013
- **Random walk initialization for training very deep feedforward networks** Sussillo and Abbott, 2014
- **Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification** He et al., 2015
- **Data-dependent Initializations of Convolutional Neural Networks** Krähenbühl et al., 2015
- **All you need is a good init**, Mishkin and Matas, 2015

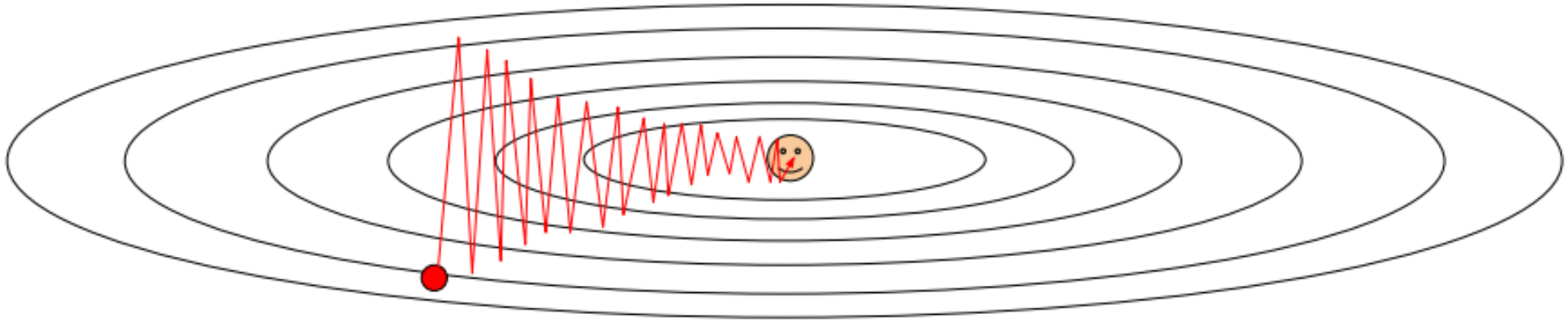
…

Stochastic Gradient Descent vs Gradient Descent



Optimization: Problems with SGD

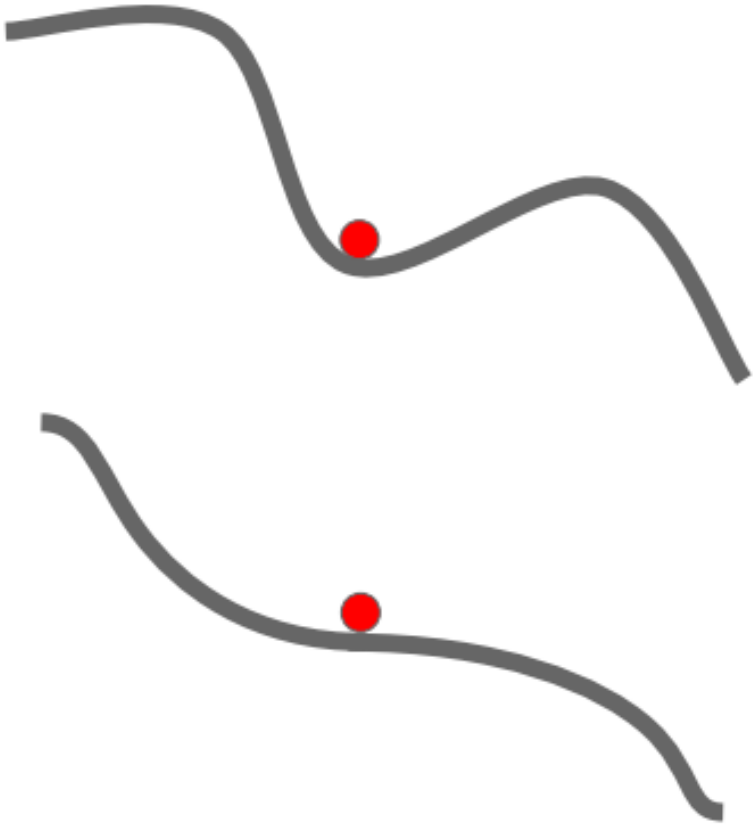
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

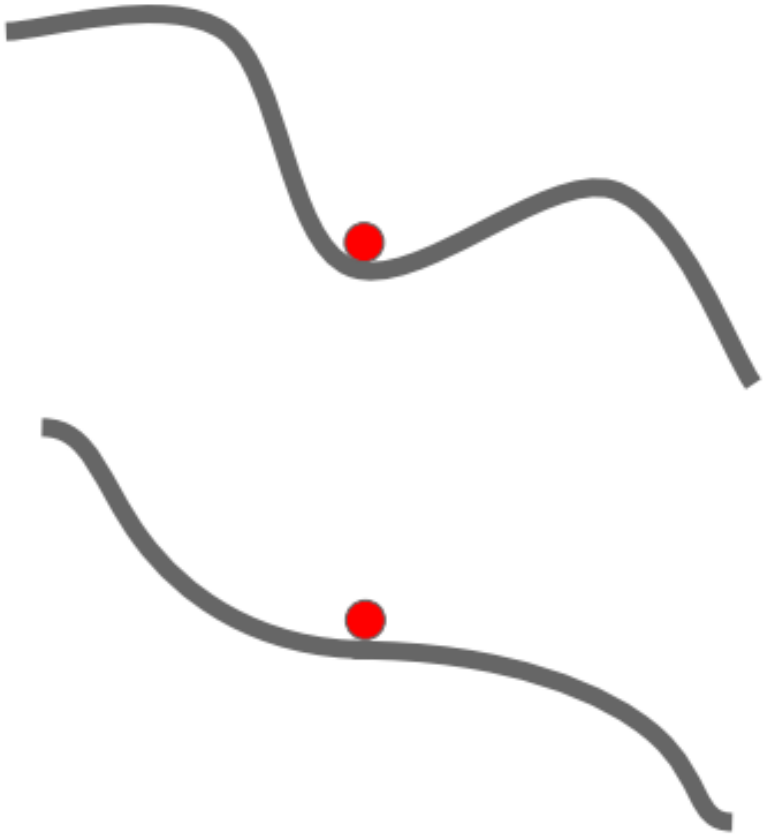
Zero gradient, gradient descent gets stuck



Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Saddle points much more common in high dimension

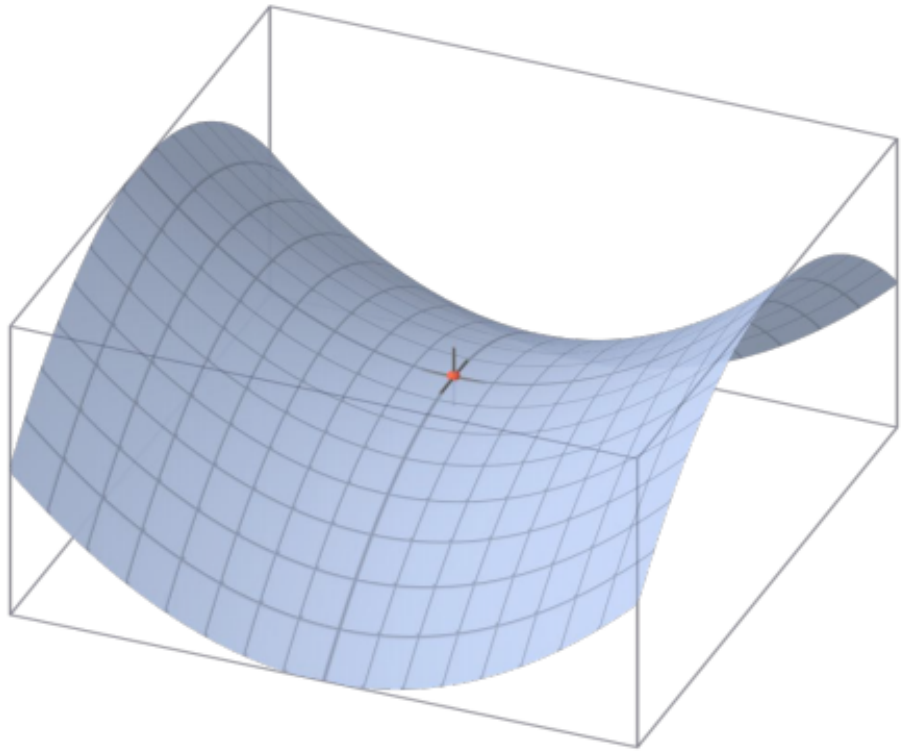


Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

Saddle points

- At a **saddle point**, $\frac{\partial L}{\partial W} = 0$ even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

- When would saddle points be a problem?
  - If we're exactly on the saddle point, then we're stuck.
  - If we're slightly to the side, then we can get unstuck.

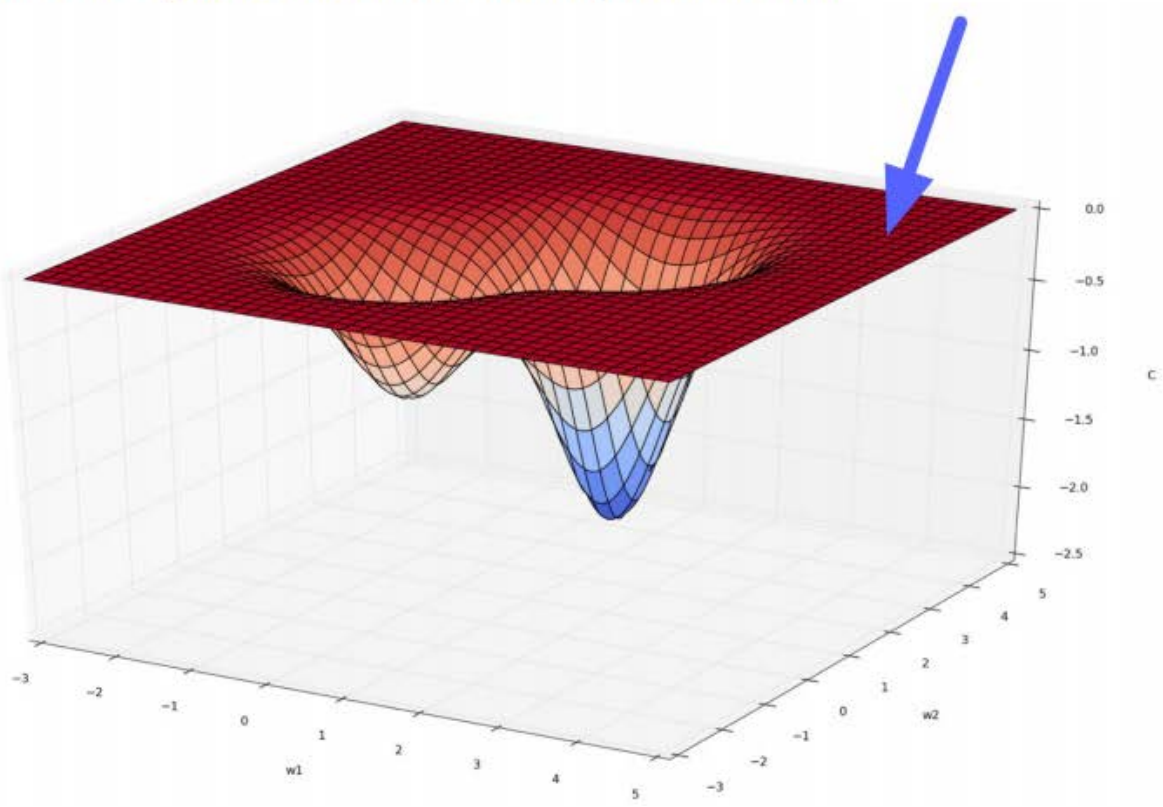


Saddle points much more common in high dimensions!

Y. Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS 2014

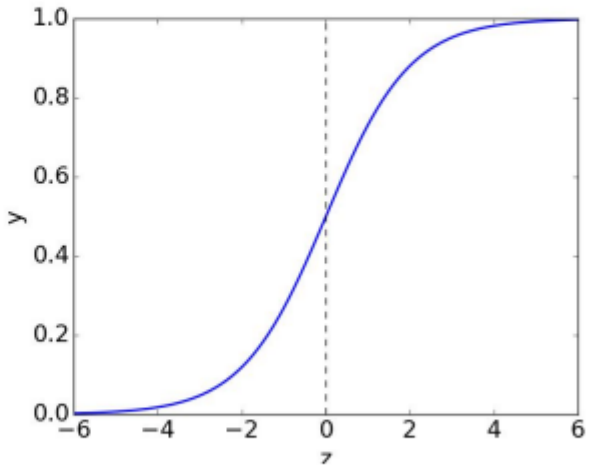
Plateaux

- A flat region is called a **plateau**. (Plural: plateaux)



Plateaux

- An important example of a plateau is a **saturated unit**. This is when it is in the flat region of its activation function.
- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be exactly 0. We call this a **dead unit**.



SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

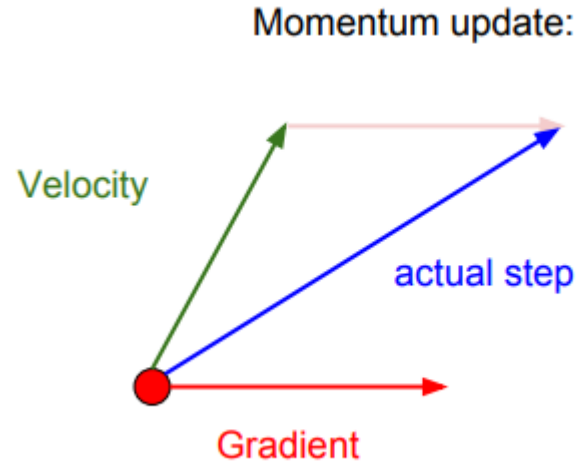
```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

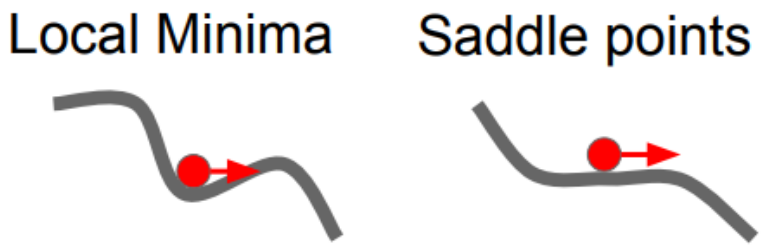
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```



- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99



Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

AdaGrad and RMSProp (Root Mean square prop)

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam (Adaptive Moment Estimation)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

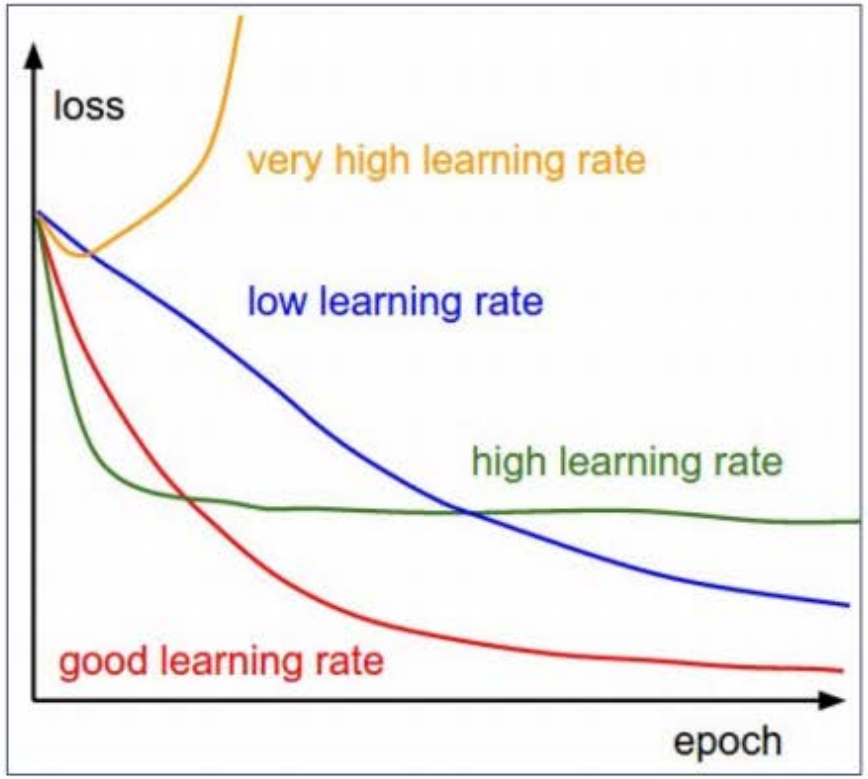
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

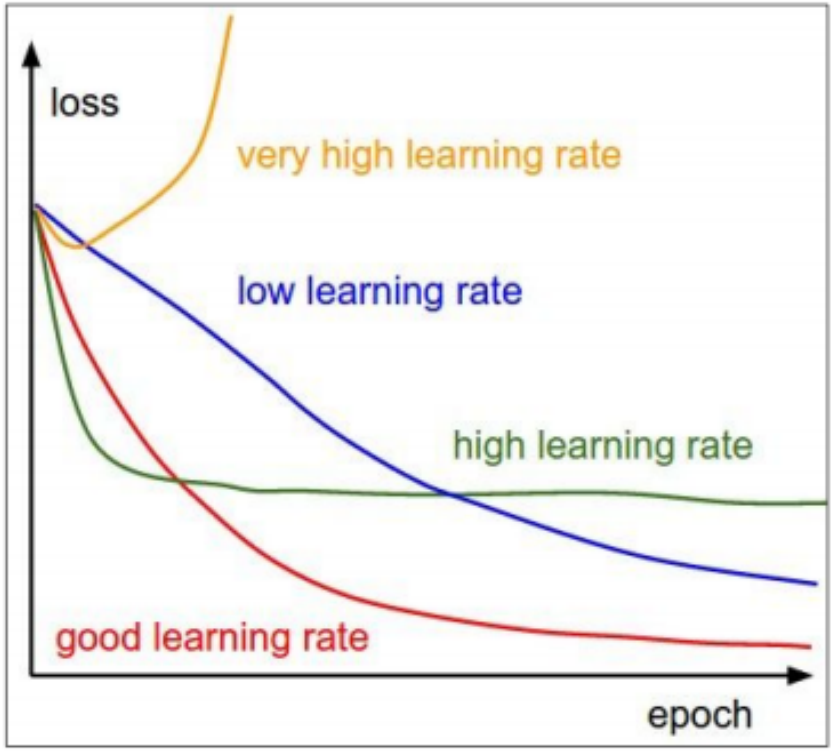
exponential decay:

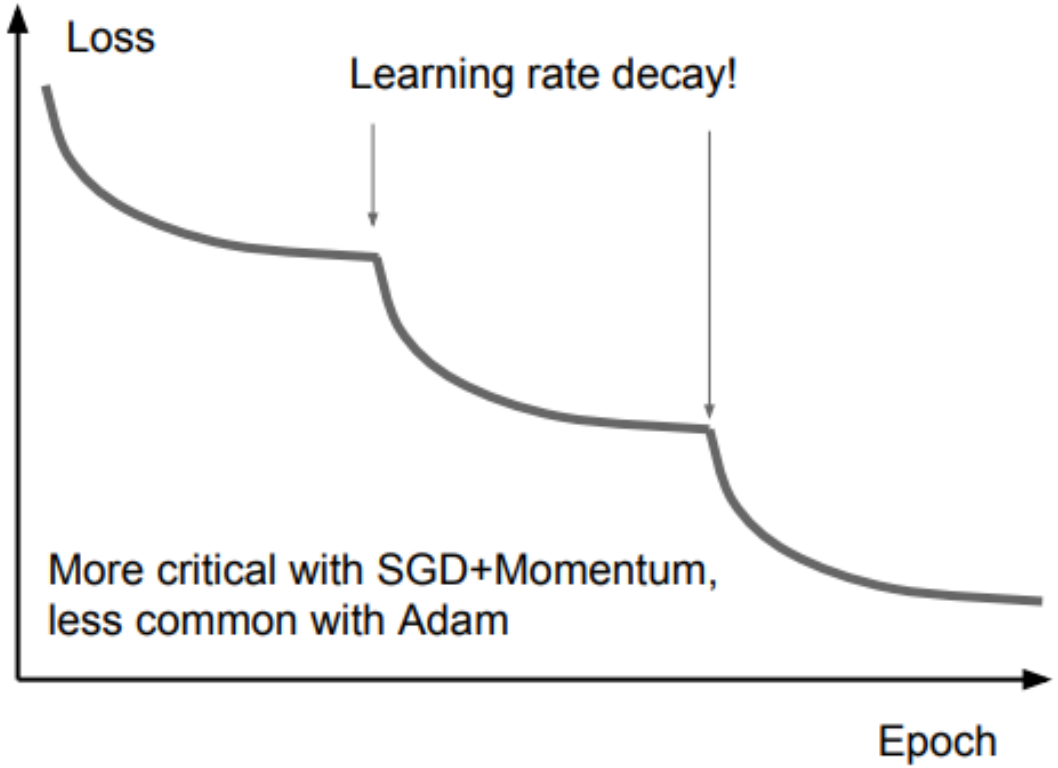
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter





Hyperparameters tuning

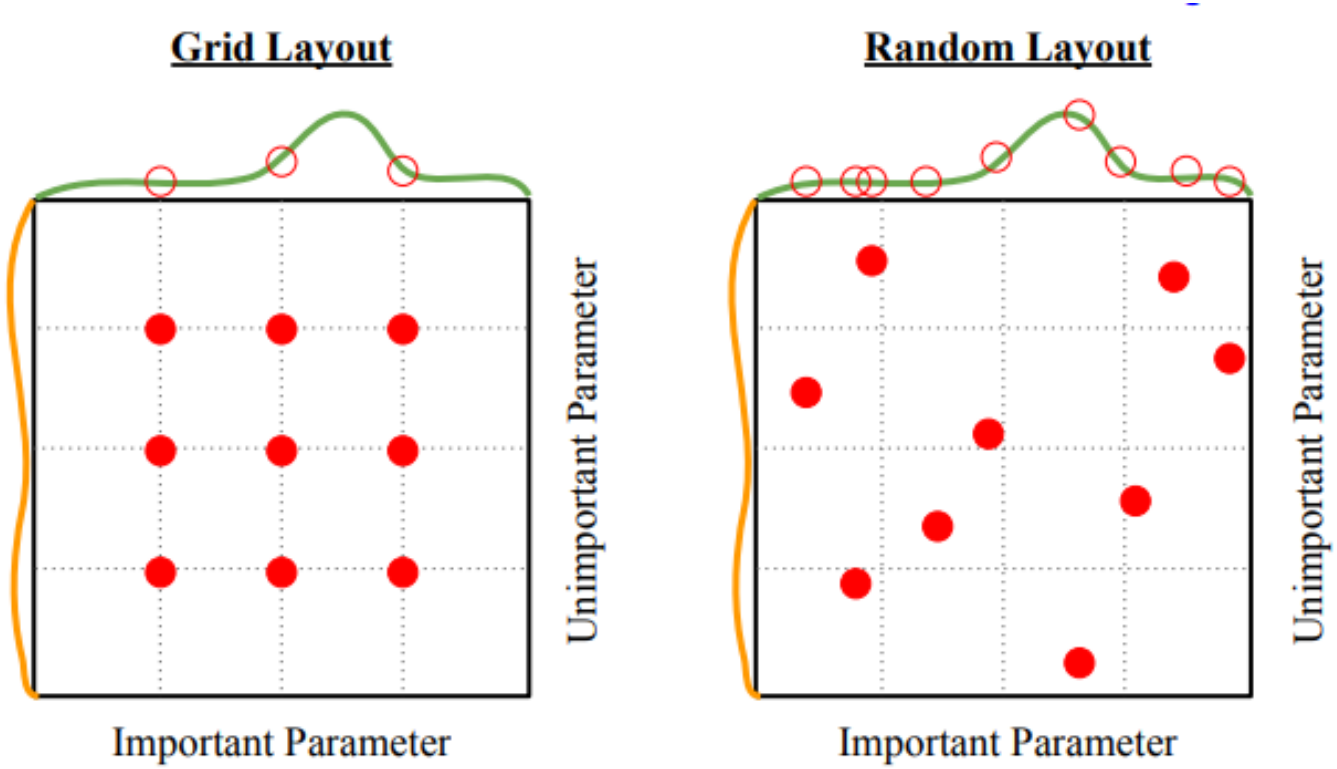


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

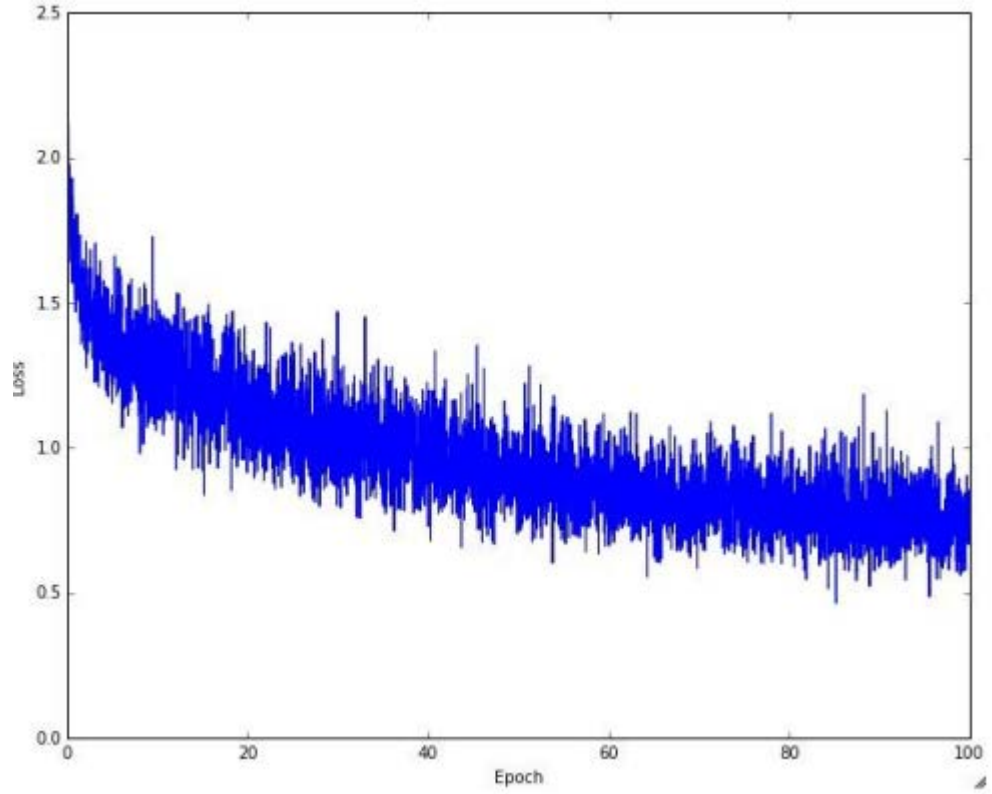
James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (February 2012), 281-305

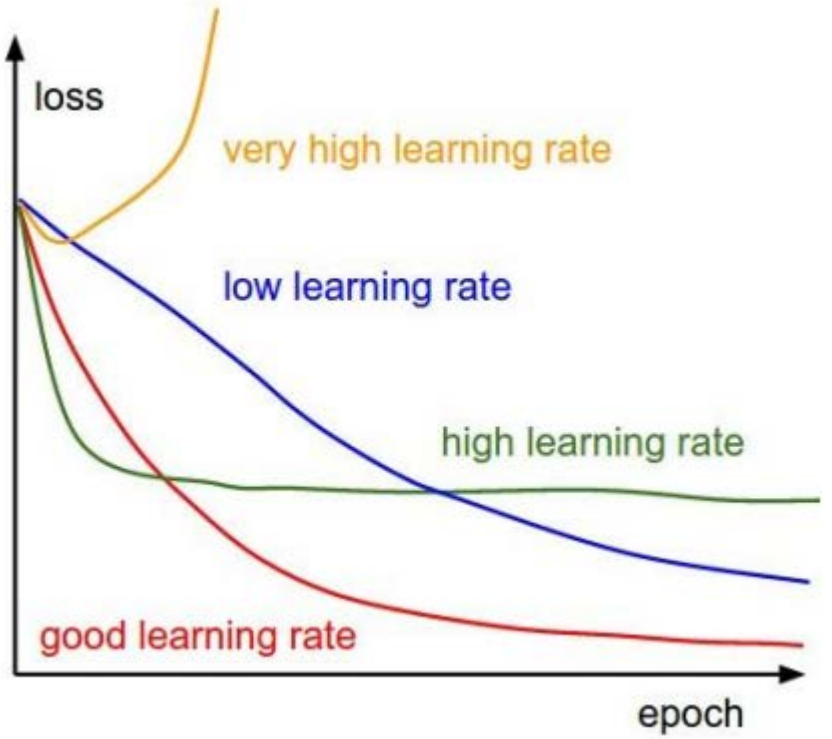
Hyperparameters tuning

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

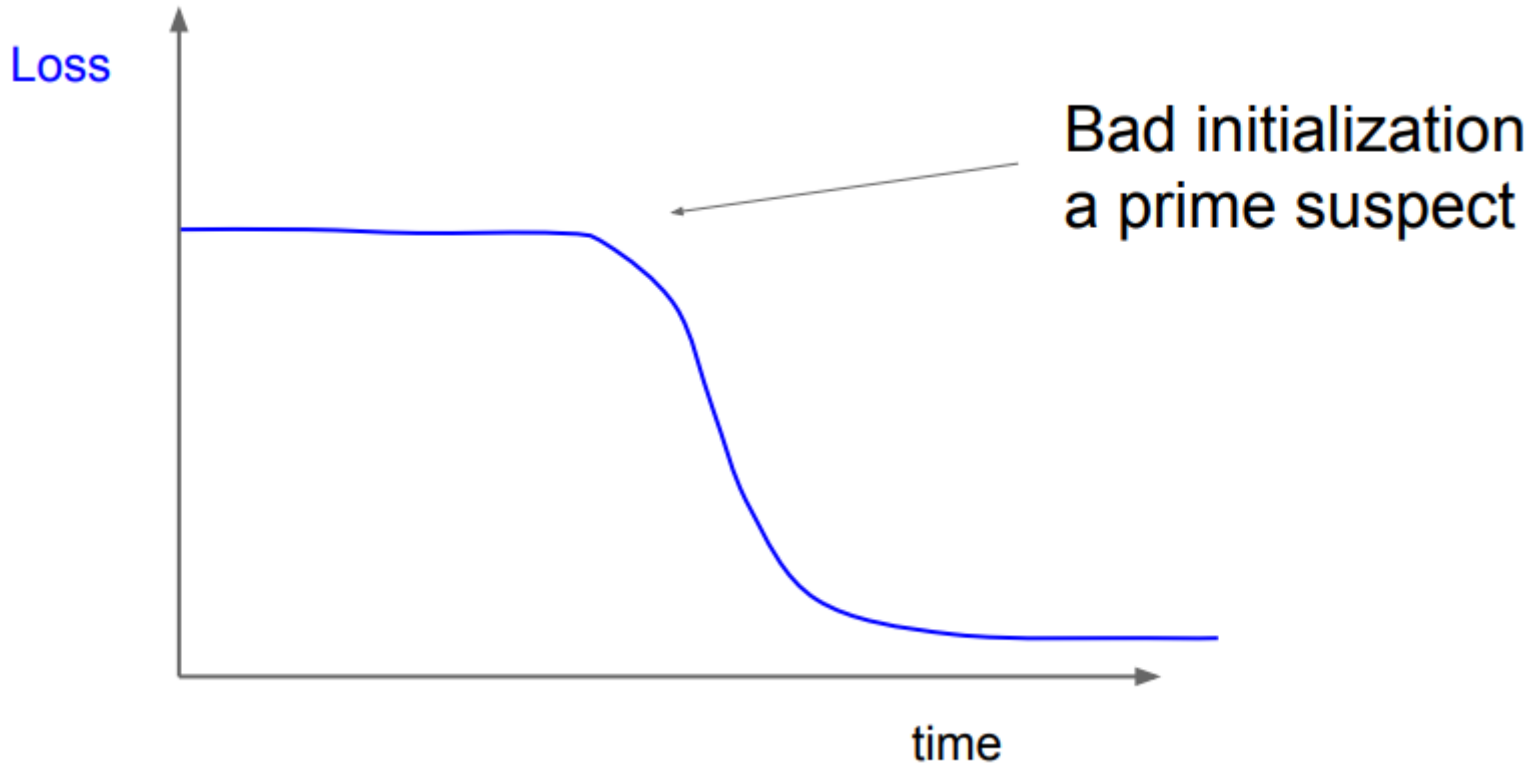
neural networks practitioner
music = loss function

Monitor and visualize the loss curve

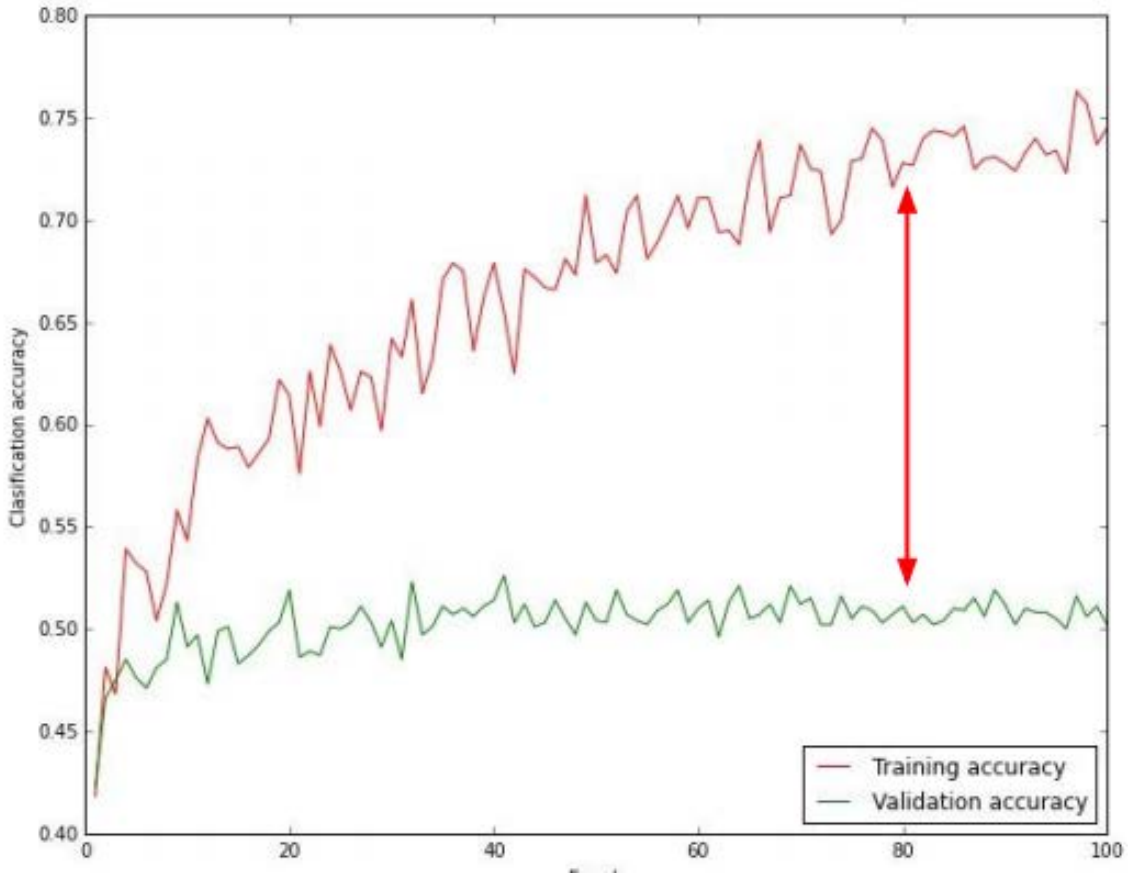




Monitor and visualize the loss curve



Monitor and visualize the accuracy



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

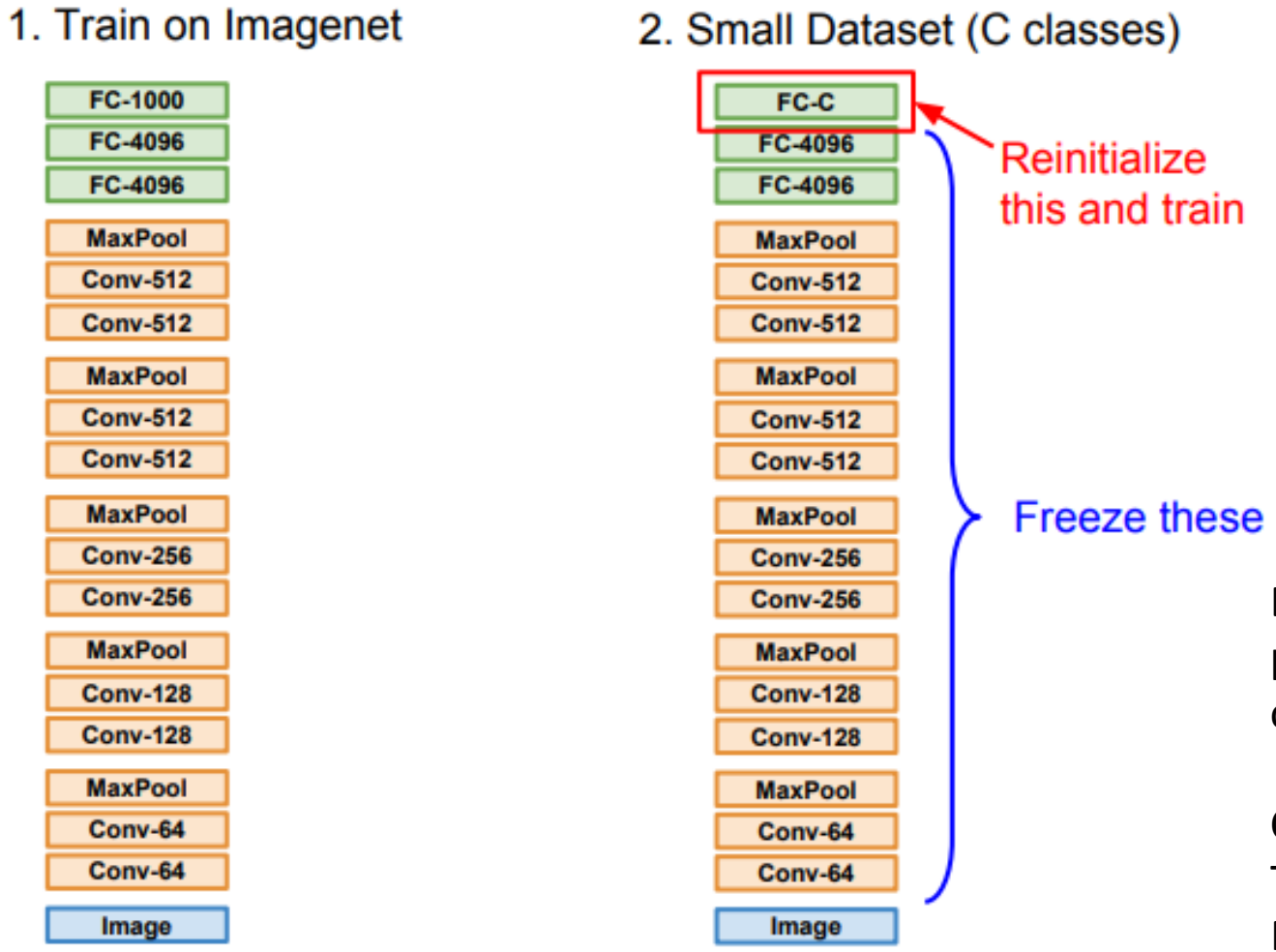
Babysitting one model vs training many models

- Model Ensembles

- 1. Train multiple independent models
- 2. At test time average their results

- Enjoy 2% extra performance

Transfer learning



Deep learning frameworks provide models of pretrained models so you might not need to train your own:

Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014 Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

Summary

- Many steps and parameters
  - Normalization
  - Weight initialization
  - Learning rate
  - Number of hidden units
  - Mini-batch size
  - Number of layers
  - Batch normalization
  - Optimization algorithms
  - Learning rate decay

In your projects..

- Describe the steps you went through? e.g.
  - What is the training, validation, test set? Why did you split the data like this?
  - Which hyperparameters did you test first, why?
- Compare and reason about the results by looking at the loss curve and accuracy, e.g.
  - Compare different weight initialization methods
  - Compare different activation functions
  - Compare different optimization functions
  - Try different learning rates
  - Compare with and without bacth normalization
  - Etc..
- Give also performance metrics
  - How much time it took for training?
  - How much time it took for testing?
  - On CPU, GPU? What are the machine specs?

Reading the research papers, critical thinking and in-depth analysis results into higher grades!
Avoid saying “We applied this and it worked well”. Try to explain why it worked!

Thoughts on research

- Scientific truth does not follow the fashion
  - Do not hesitate being a contrarian if you have good reasons
- Experiments are crucial
  - Do not aim at beating the state-of-the-art, aim at understanding the phenomena
- On the proper use of mathematics
  - A theorem is not like a subroutine that one can apply blindly
  - Theorems should not limit creativity

Olivier Bousquet, Google AI, NeurIPS2018

Supplementary reading and video

- Deep Learning book, Chapter 6, 7 and 8
- http://neuralnetworksanddeeplearning.com/, Michael Nielsen
- https://www.youtube.com/playlist?list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH, Hugo Larochelle's video lectures (1.1 to 2.7)
- https://webcolleges.uva.nl/Mediasite/Play/947ccbc9b11940c0ad5ab39ebb154c461d, Efstratios Gavves' Lecture 3
- Machine Learning and Deep Learning courses on Coursera by Andrew Ng

References

- MIT 6. S191 Introduction to Deep Learning
- CS231n: Convolutional Neural Networks
- CMP8784: Deep Learning, Hacettepe University
- (Slides mainly adopted from the above courses)

Tensorflow tutorial

- https://www.tensorflow.org/tutorials/