Basic Principles

of

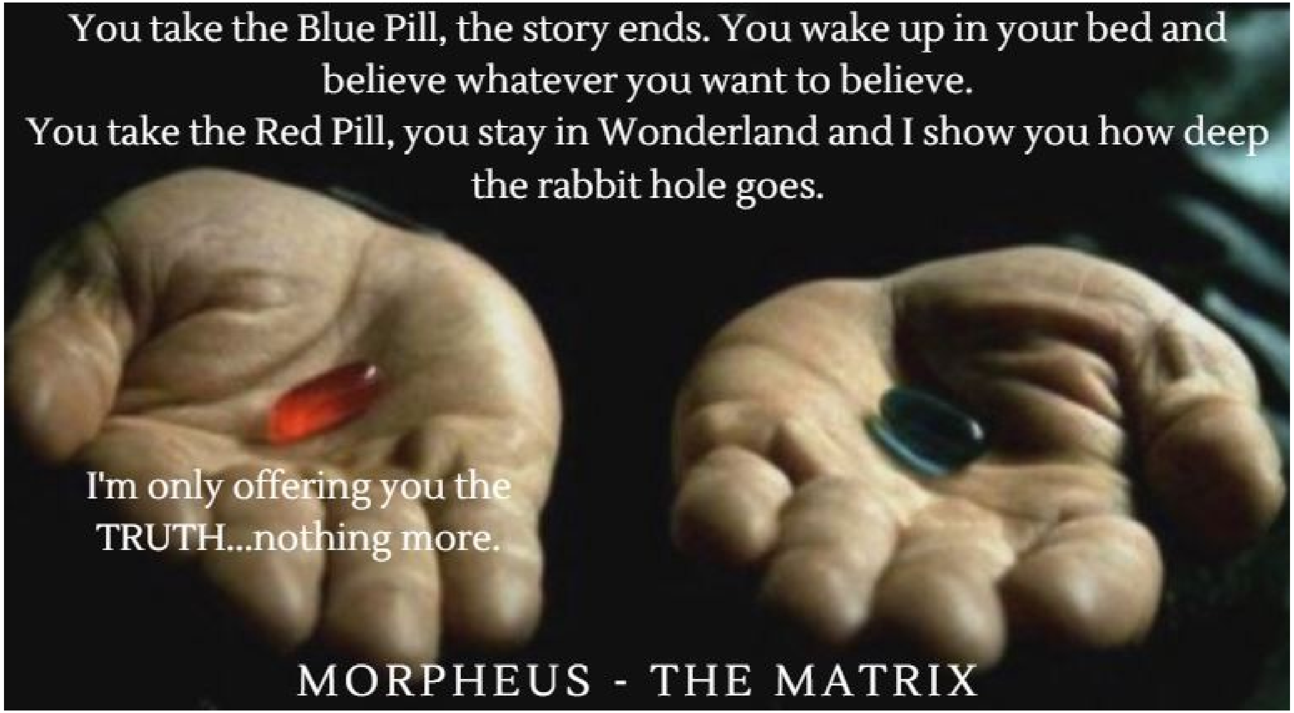
Neural Networks

Reloaded

Lecture 9

Fabian Märki / FHNW - I4DS

Introduction to Neural Networks Reloaded



Outline

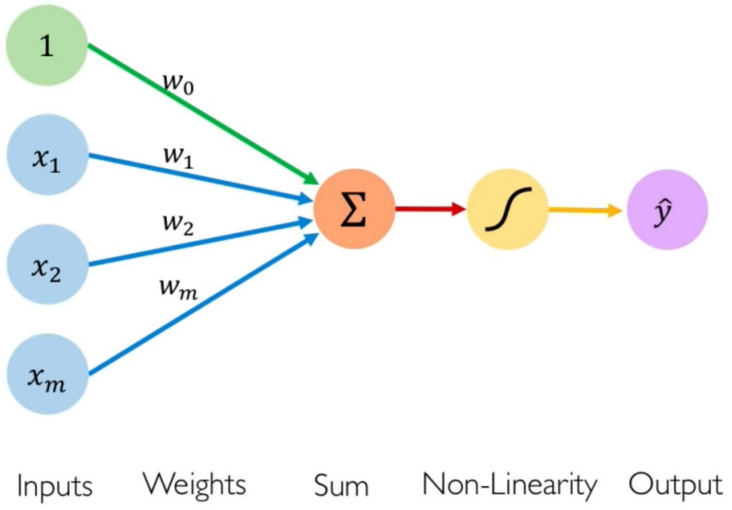
- Basics of Neural Networks
  - Basic building block – perceptron (neuron)
  - Activation functions
  - Training a Neural Network
  - Problem of overfitting (regularization)

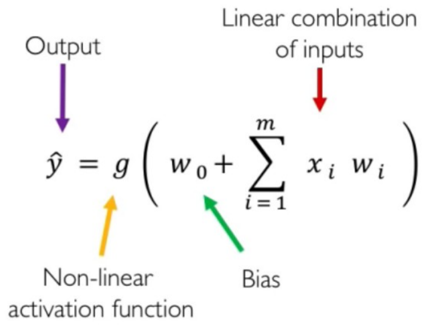
- Next Lectures
  - Special Neural Network based approaches used for Language Modeling

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Perceptron – Basic Building Block

- Perceptron (also called neuron)

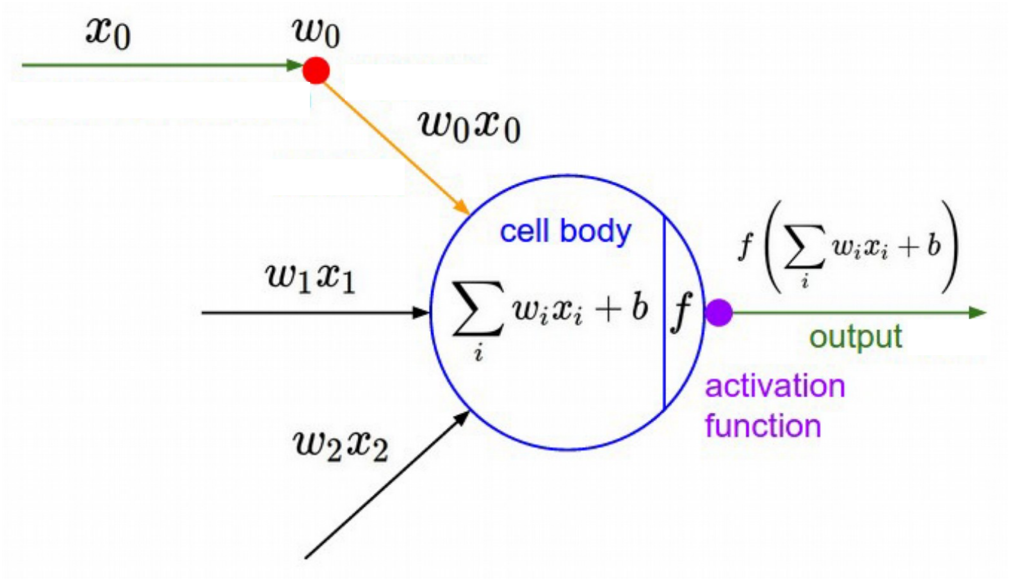




6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Perceptron – Basic Building Block

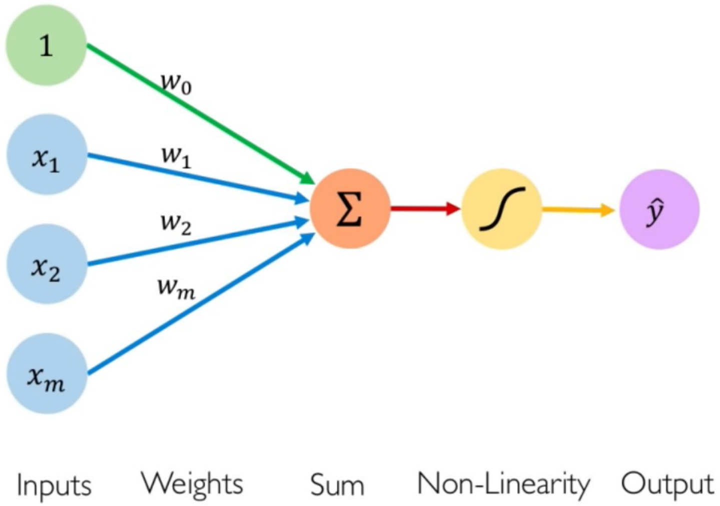
- Perceptron (alternative visualization)



CS224n: Natural Language Processing with Deep Learning, Stanford, Winter 2019 - http://www.youtube.com/watch?v=8CWyBNX6eDo&t=25m31s

Activation Function

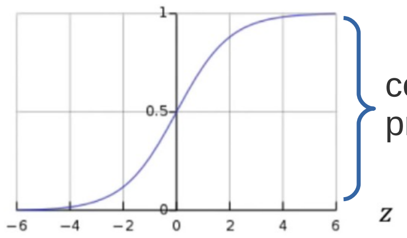
- Activation Function: Makes NN to Universal Function Approximator
  - Should be differentiable (for backpropagation)



$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

- Example: sigmoid function

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

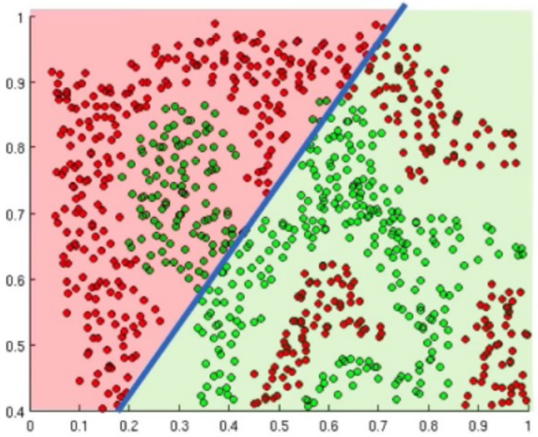


common use case:
probability distribution [0..1]

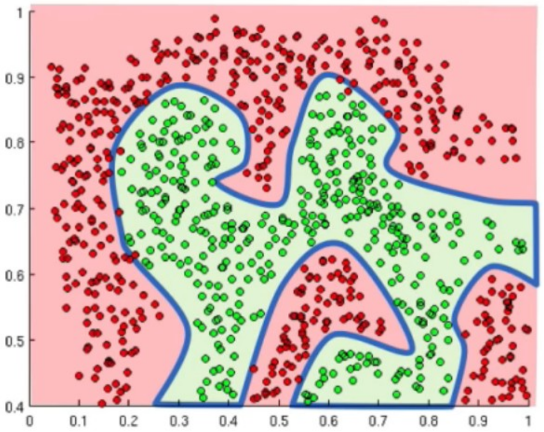
6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Purpose of Activation Functions

- Purpose of activation functions is to **introduce non-linearities**



Linear activation functions produce linear decision boundaries no matter the network size



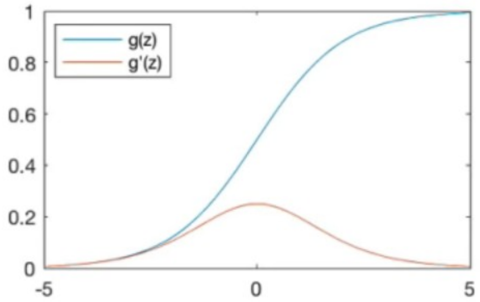
Non-linearities allow us to approximate arbitrarily complex functions

See this link for a nice visualization of decision boundaries using different activation functions and network capacities

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Common Activation Functions

Sigmoid Function

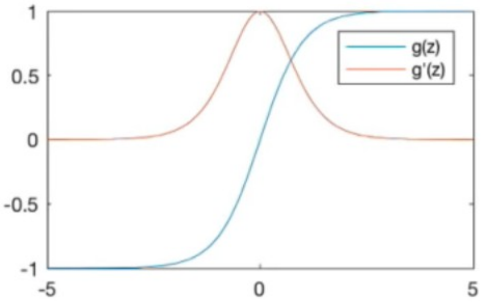


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

```
tf.math.sigmoid(z)
```

Hyperbolic Tangent

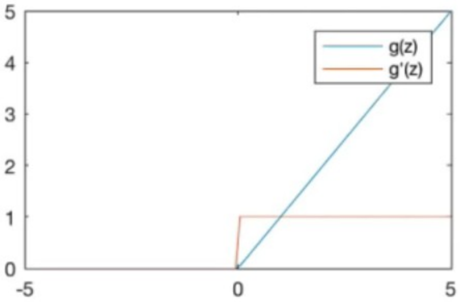


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

```
tf.math.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)
```

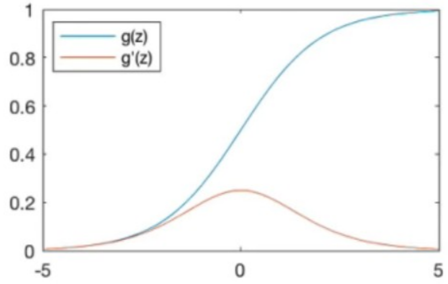

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Activation Function: Sigmoid

- Range [0..1]
- Output is not zero centered
  - Makes optimization difficult
  - Zero mean and normalize data
- Vanishing gradient problem (saturate and kill gradients)
- Sigmoids have slow convergence
- Computationally expensive (e^x)

➔ No practical relevance

Sigmoid Function

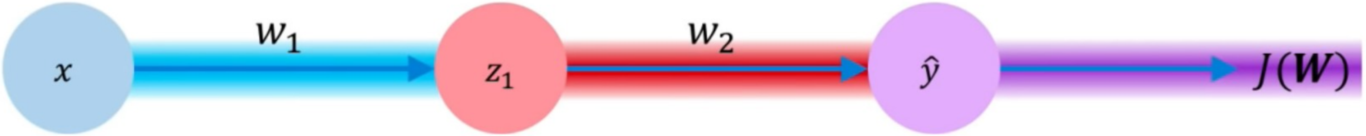


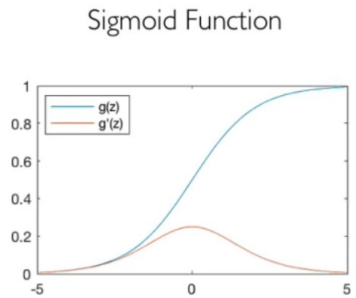
$$g(z) = \frac{1}{1 + e^{-z}}$$

https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f and https://adventuresinmachinelearning.com/vanishing-gradient-problem-tensorflow and https://www.learnopencv.com/understanding-activation-functions-in-deep-learning and https://rohanvarma.me/inputnormalization and http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf

Problems: Sigmoid

- Compute gradients through backpropagation





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

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Factors < 1 → vanishing gradients

Factors > 1 → exploding gradients

→ careful weight initialization

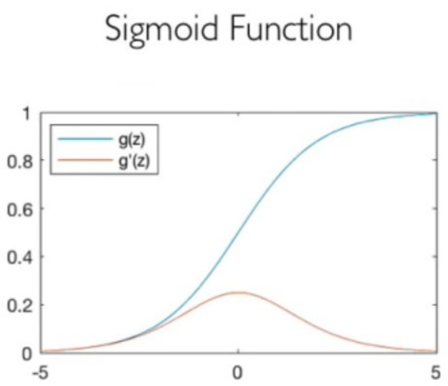
repeatedly apply chain rule
for every weight in the network using gradients from later layers

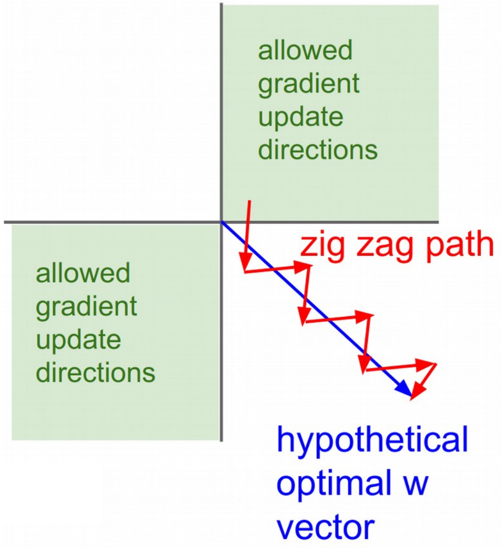
6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Problems: Sigmoid

- Why is it a problem when the output is not zero centered?
  - Makes optimization difficult
  - Gradients are either all positive or all negative
  - This is also why you want zero-mean data

See also here

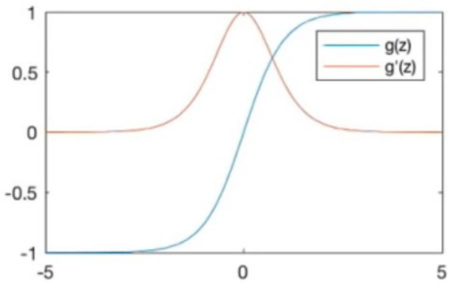




Activation Function: Tanh

- Range [-1..1]
- Output is zero centered
- Vanishing gradient problem (saturate and kill gradients)
- Computationally expensive (e^x)

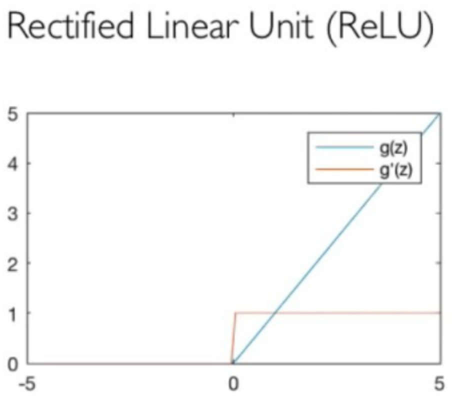
Hyperbolic Tangent



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

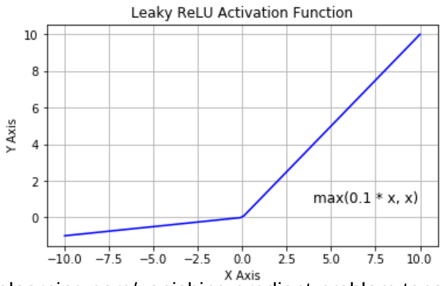
https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f and https://adventuresinmachinelearning.com/vanishing-gradient-problem-tensorflow and https://www.learnopencv.com/understanding-activation-functions-in-deep-learning

Activation Function: ReLU



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

- Converges faster (approx. 6 times faster than tanh)
- Resistant to the vanishing gradient problem
  - At least in the positive region
- Computationally very efficient
- Output is not zero centered
- If x < 0 during the forward pass, the neuron remains inactive and it kills the gradient during the backward pass (can result in Dead Neurons)
  - weights do not get updated, and the network cannot learn anymore
  - Leaky (parametric/randomized) ReLU

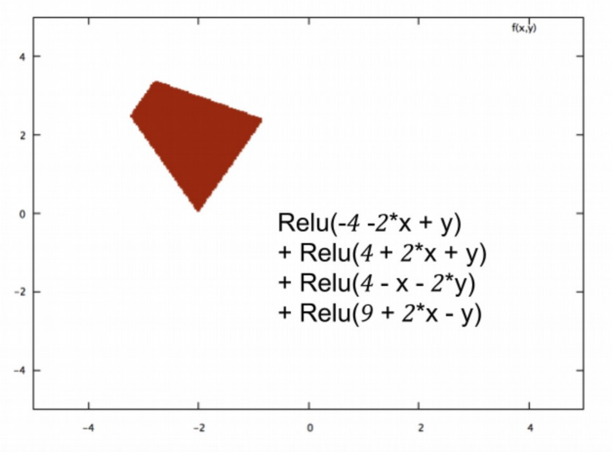


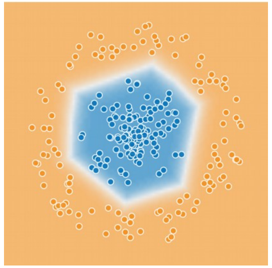
➔ Good default choice

https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f and https://adventuresinmachinelearning.com/vanishing-gradient-problem-tensorflow and https://www.learnopencv.com/understanding-activation-functions-in-deep-learning

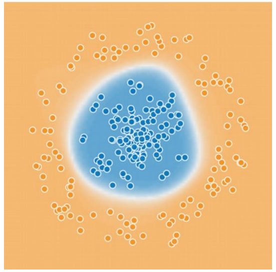
Activation Function: ReLU

- Why is ReLU non-linear?
  - It bends at the x-axis
  - Allows for building arbitrary shaped curves on the feature plane





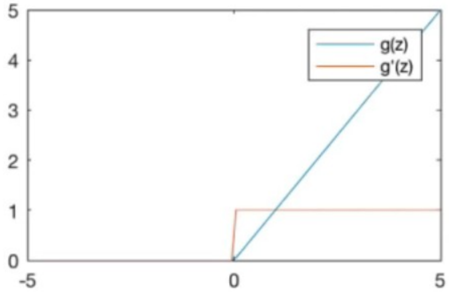
ReLU



Tanh

- A comparison of ReLU and Tanh shaped decision boundaries

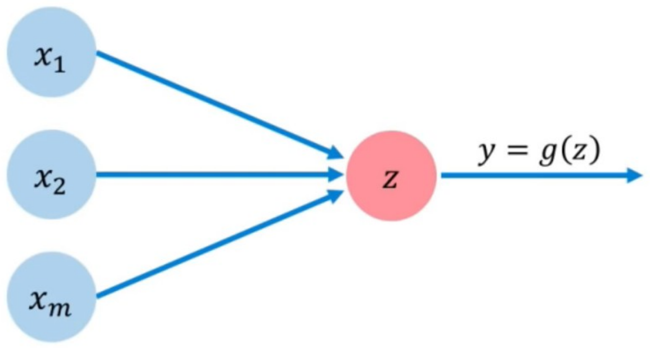
Rectified Linear Unit (ReLU)



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

https://medium.com/machine-intelligence-report/how-do-neural-networks-work-57d1ab5337ce or https://stats.stackexchange.com/a/299933

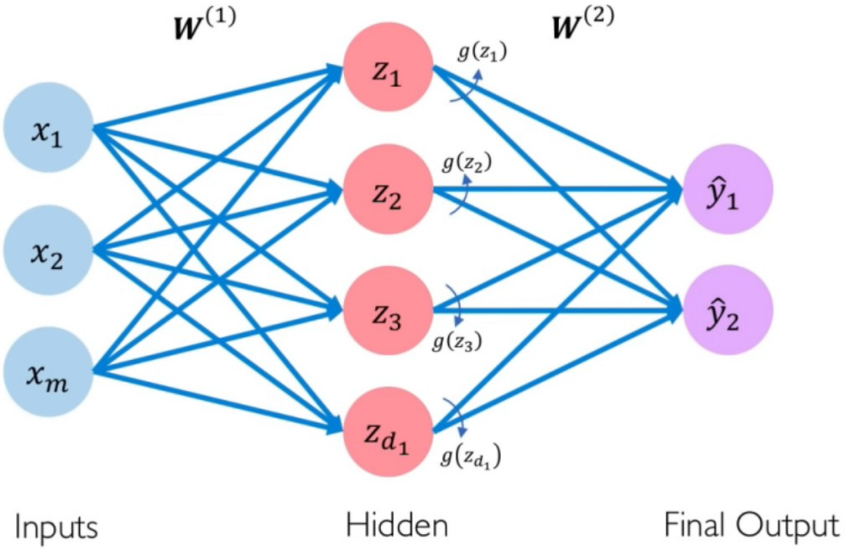
The Perceptron: Simplified



$$z = w_0 + \sum_{j=1}^{m} x_j\, w_j$$

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

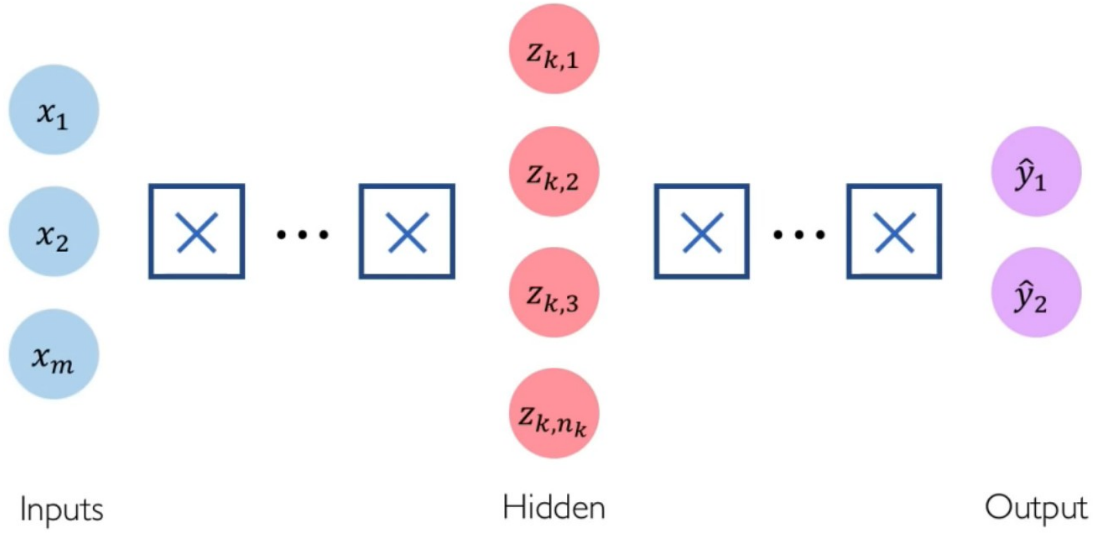
Neural Network with one Hidden Layer



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

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Deep Neural Network



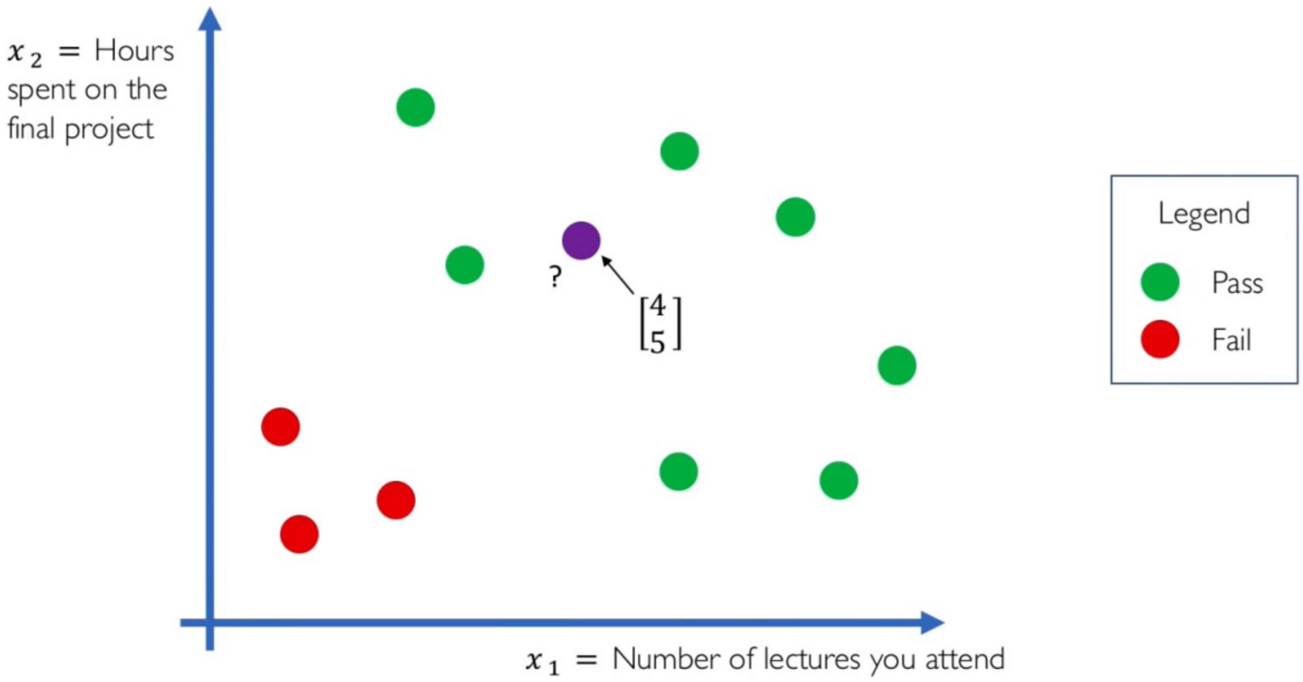

```
import tensorflow as tf

model = tf.keras.Sequential([
  tf.keras.layers.Dense(n₁),
  tf.keras.layers.Dense(n₂),
  ⋮
  tf.keras.layers.Dense(2)
])
```

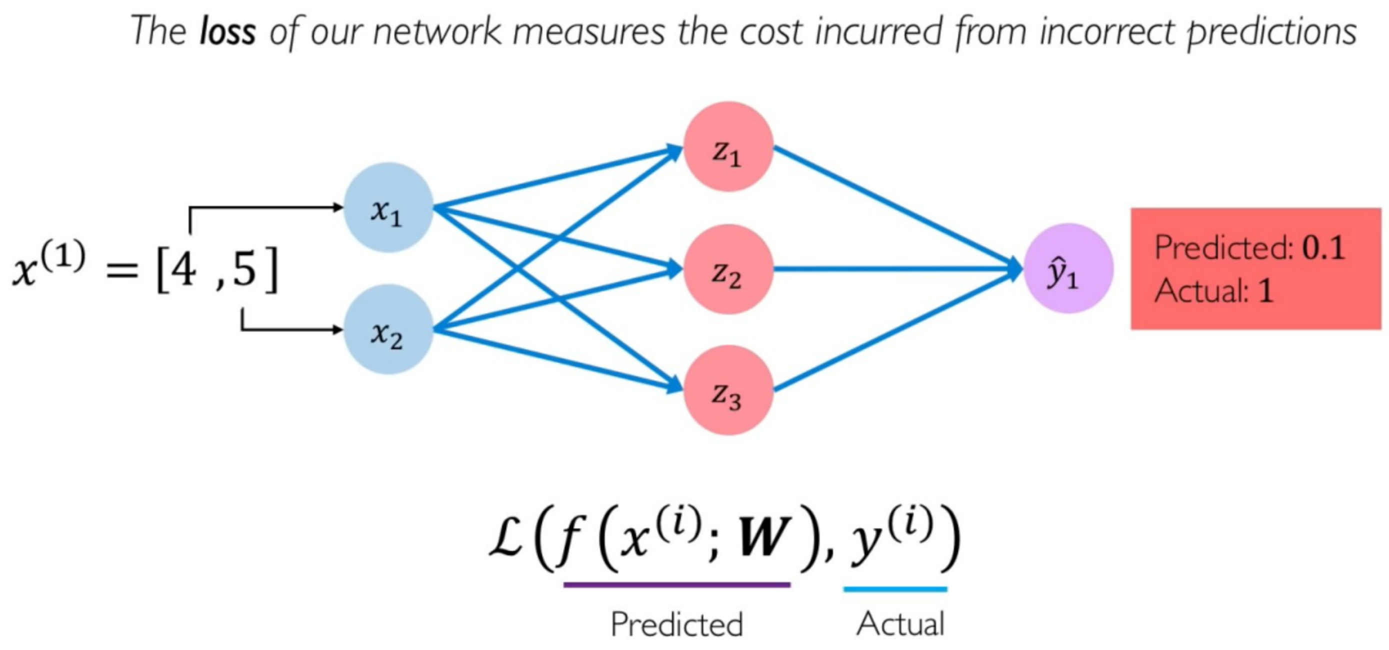
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Example Problem: Will I pass this class?



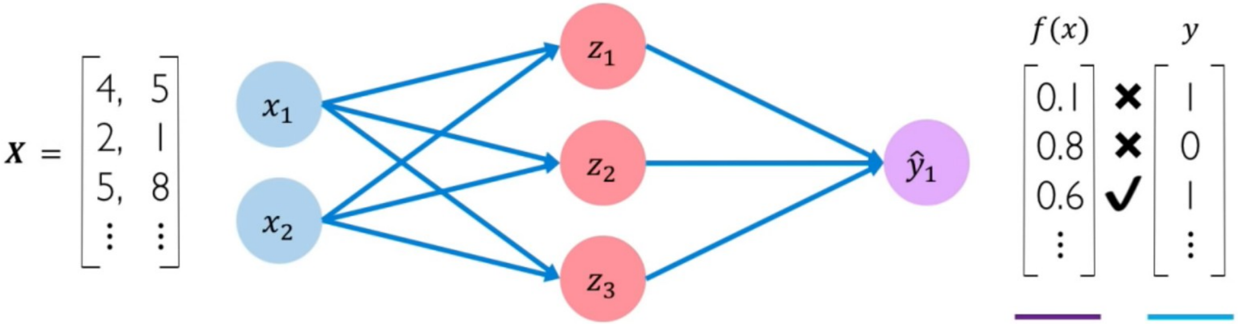
Example Problem: Will I pass this class?



6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Binary Cross Entropy Loss

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



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

a minus (-) is missing

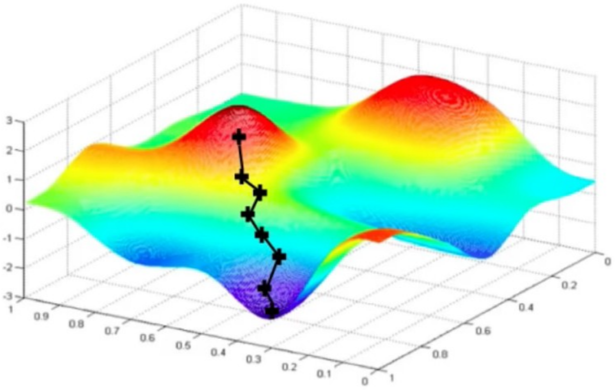
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```


6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Loss Optimization – Gradient Descent

Algorithm

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

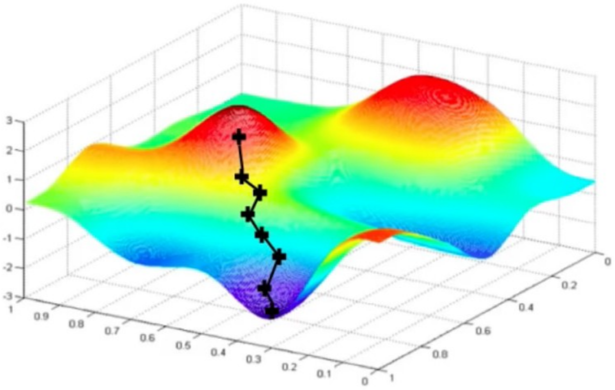


6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Loss Optimization – Gradient Descent

Algorithm

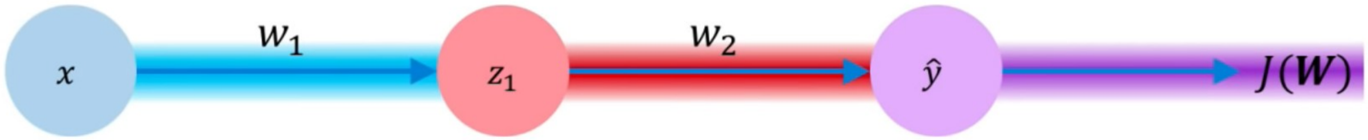
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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



6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Compute Gradients: Backpropagation

- How does a small change in one weight (e.g. w_1) affect the final loss J(W)?



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

repeatedly apply chain rule
for every weight in the network using gradients from later layers

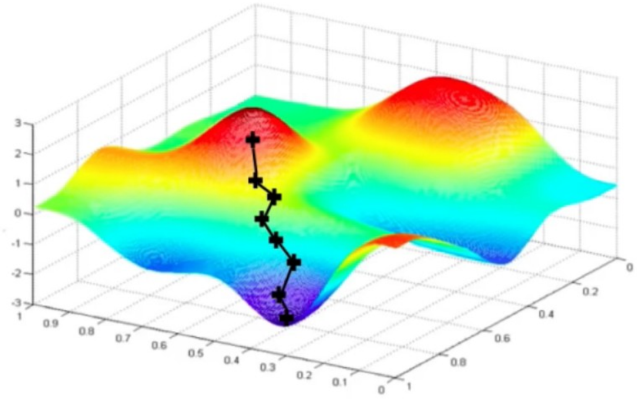
A lecture on backpropagation using the computational graph

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Gradient Descent

Algorithm

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

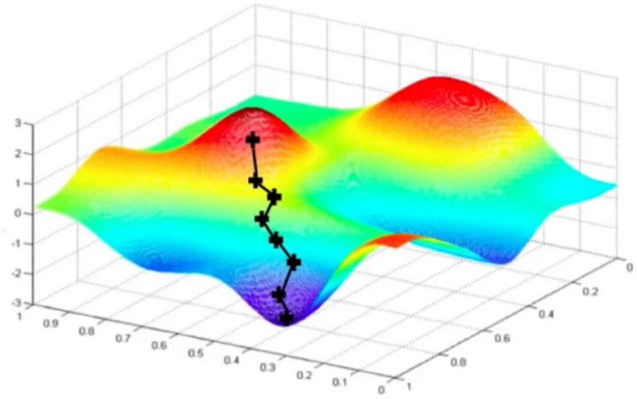


Can be very **computationally intensive** to compute!

Stochastic Gradient Descent

Algorithm

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



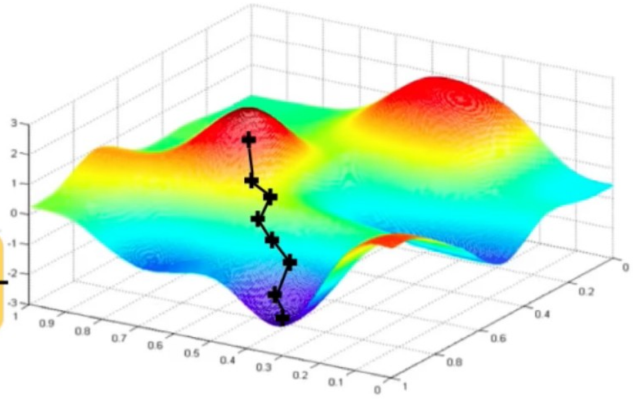
Easy to compute but
very noisy (stochastic)!

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Mini-Batch 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, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



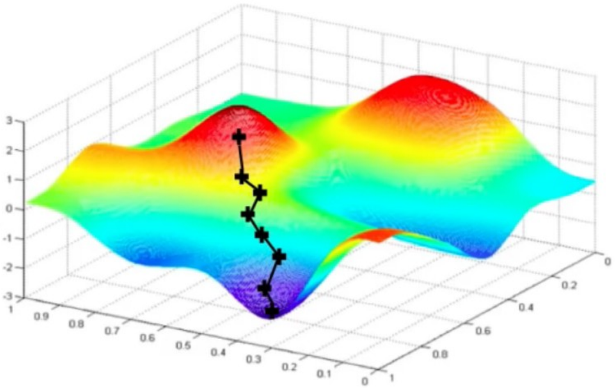
Fast to compute and a much better estimate of the true gradient!

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Loss Optimization – Gradient Descent

Algorithm

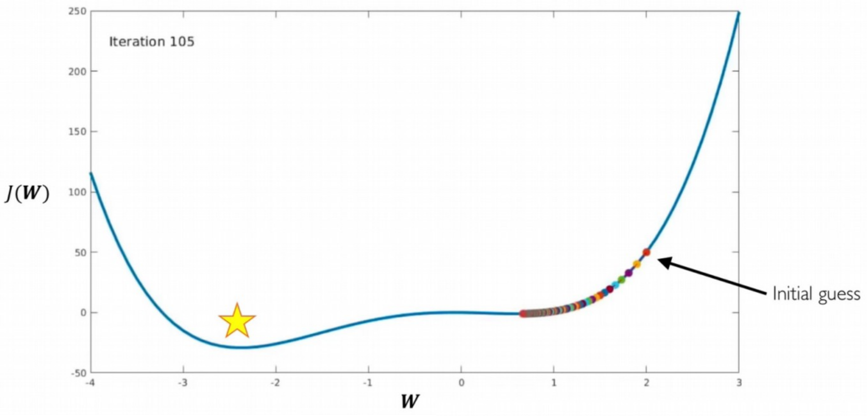
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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



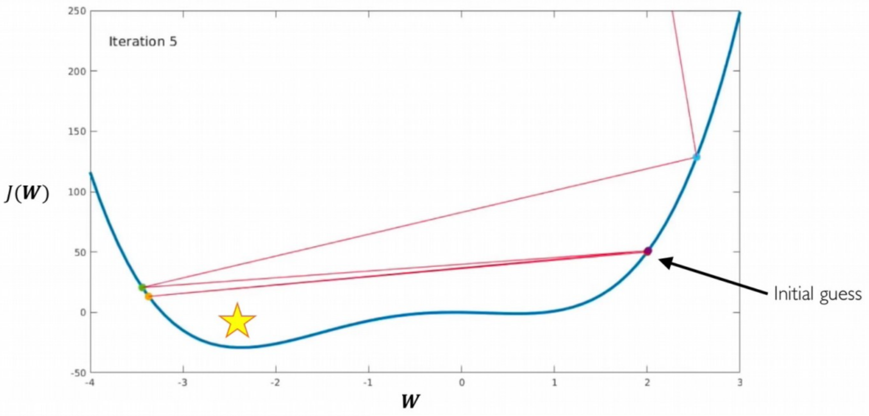
6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Setting the Learning Rate

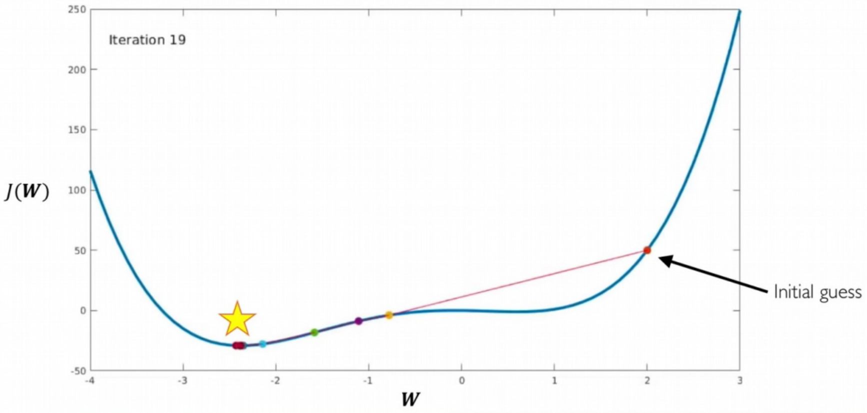
Small learning rate *converges slowly and gets stuck in false local minima*



Large learning rates *overshoot, become unstable and diverge*



Stable learning rates *converge smoothly and avoid local minima*



More on learning rate

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Gradient Descent Algorithm

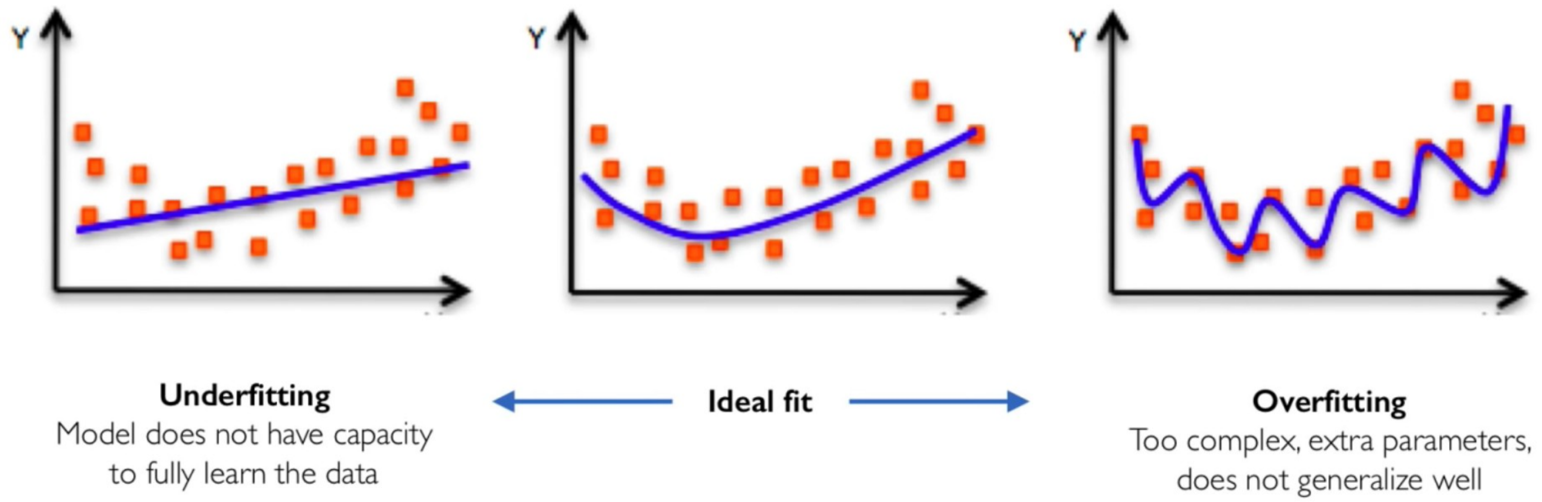
Algorithm	TF Implementation
SGD	`tf.keras.optimizers.SGD`
Adam	`tf.keras.optimizers.Adam`
Adadelta	`tf.keras.optimizers.Adadelta`
Adagrad	`tf.keras.optimizers.Adagrad`
RMSProp	`tf.keras.optimizers.RMSProp`

Adam is a good default choice

Additional details on Gradient Descent Algorithms (here and here)

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

The Problem of Overfitting

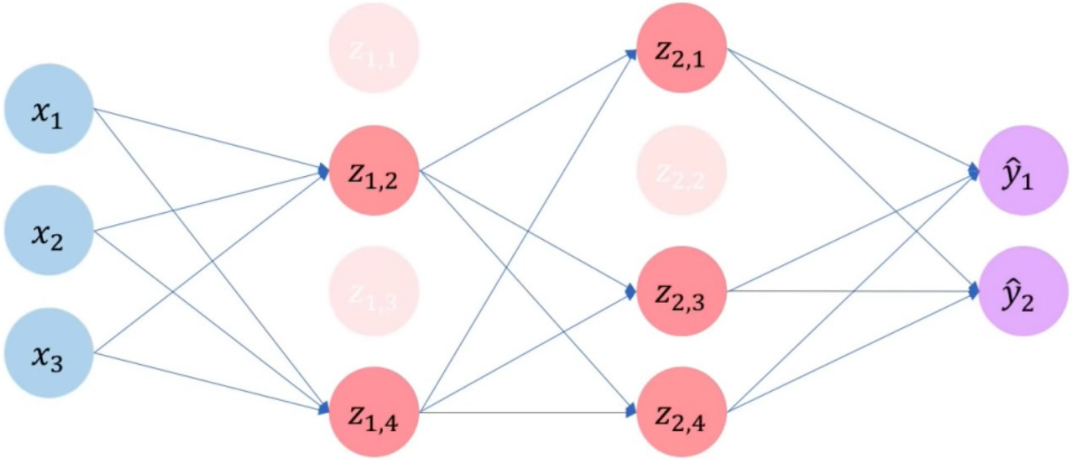


6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Regularization: Dropout

- During training (at every iteration), randomly set some activations to 0
  - E.g. dropout rate of 50%
  - Forces network to not rely on any node

```
tf.keras.layers.Dropout(p=0.5)
```



- More on dropout and other regularization techniques

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Early Stopping

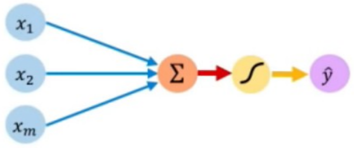
- Stop training before we have a chance to overfit

6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Wrap-Up

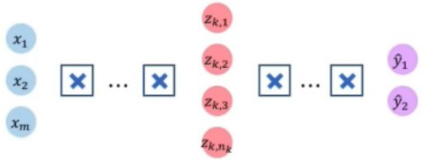
The Perceptron

- Structural building blocks
- Nonlinear activation functions



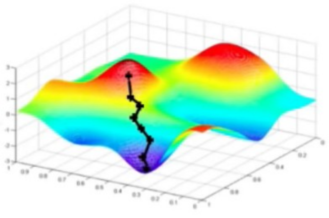
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



6.S191 Introduction to Deep Learning - https://www.youtube.com/watch?v=njKP3FqW3Sk&t=11m2s

Questions