MSIS549: ML and AI for Business Applications
Lesson 2

Foster School of Business

Course Outline

Lesson 1: Introduction to Deep Learning

Lesson 2: Neural Network Fundamentals

Lesson 3: Convolutional Neural Networks

Lesson 4: Recurrent Neural Networks

Lesson 5: Deep Learning Practice

Recap

Perceptual Problems are Challenging

- Traditional ML are not good at perceptual tasks that involving skills that seem natural and intuitive to humans.

- For example:
  - Image classification
  - Speech recognition
  - Learning a new language
  - Driving cars
  - Etc.

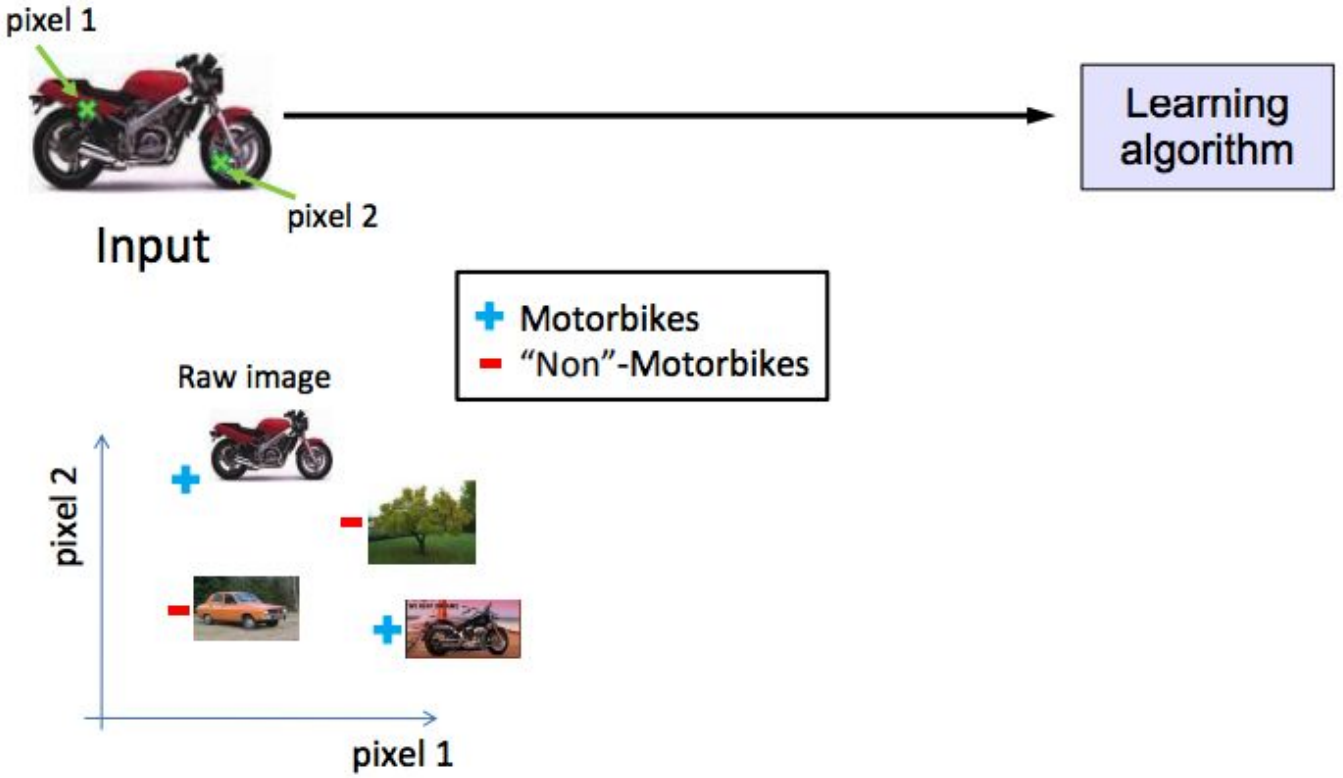
Why it's hard for ML at perceptual problems?

You see this:

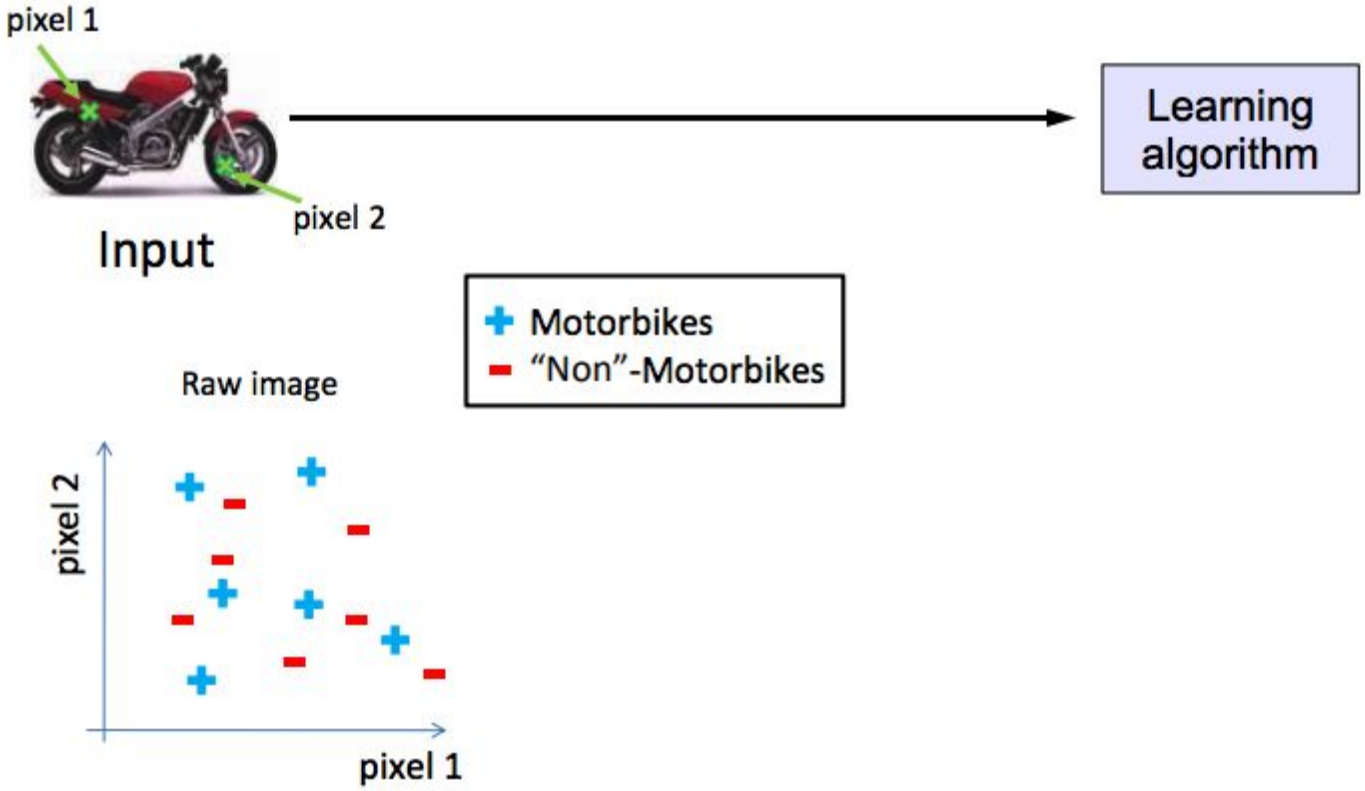
But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

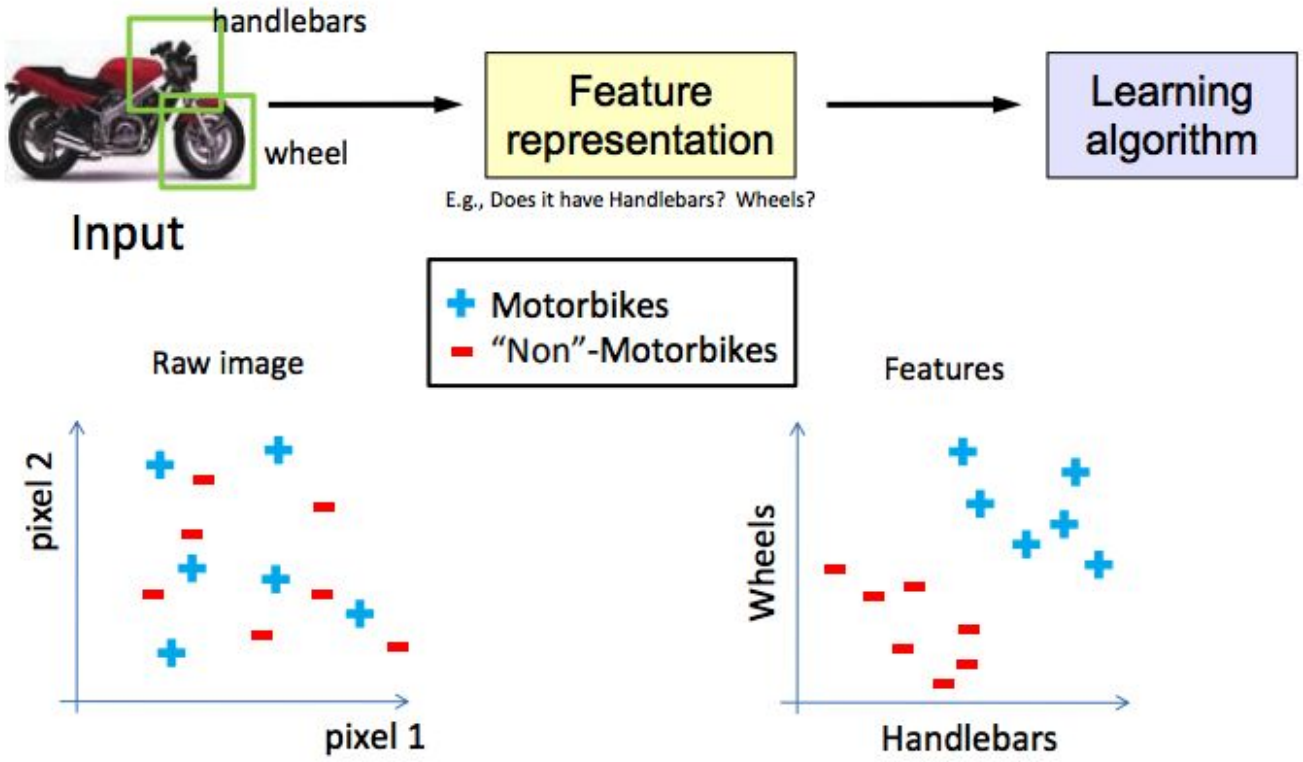
Why it's hard for ML at perceptual problems?



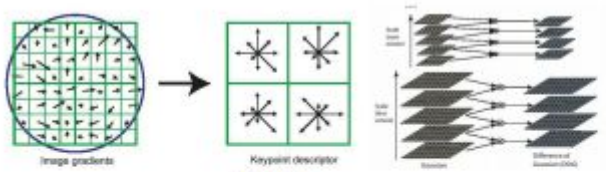
Why it's hard for ML at perceptual problems?



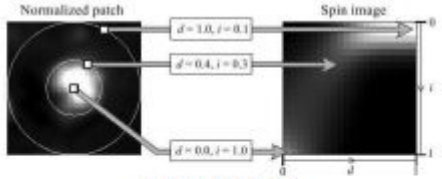
Why is ML bad at perceptual problems?



Some feature representations

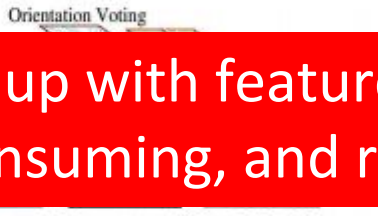


SIFT



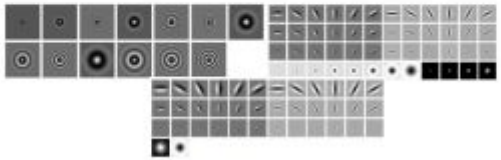
Spin image

Coming up with features is often difficult, time-consuming, and requires expert knowledge.

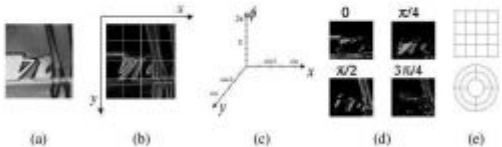


HoG

RIFT



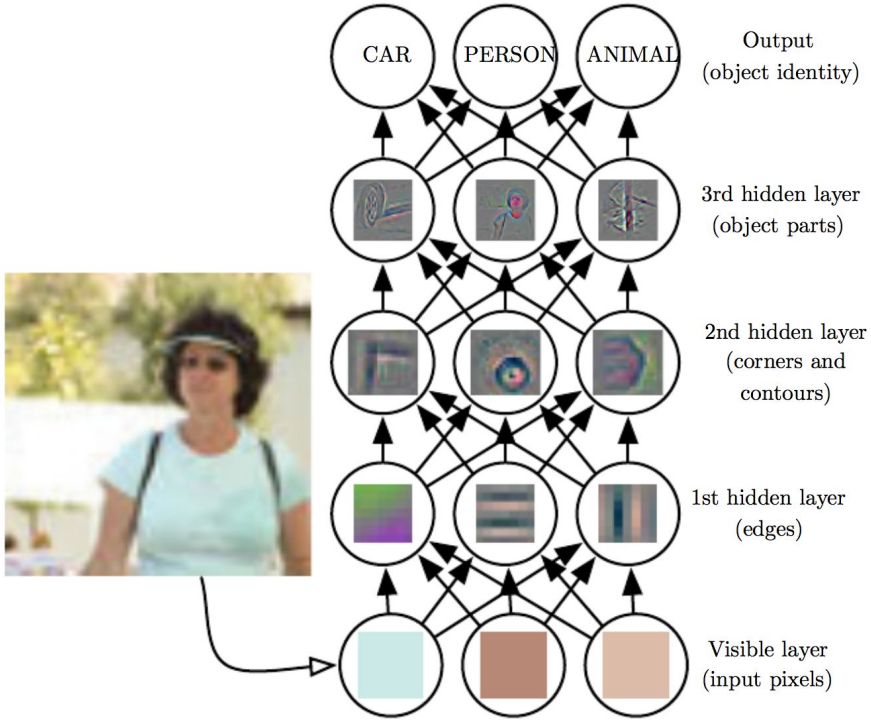
Textons



GLOH

Can we learn the representation?

- Hand engineered features are time consuming, brittle and not scalable in practice.
- Can we learn the underlying features directly from data?
- Build high-level features from low-level features.



End-to-end learning

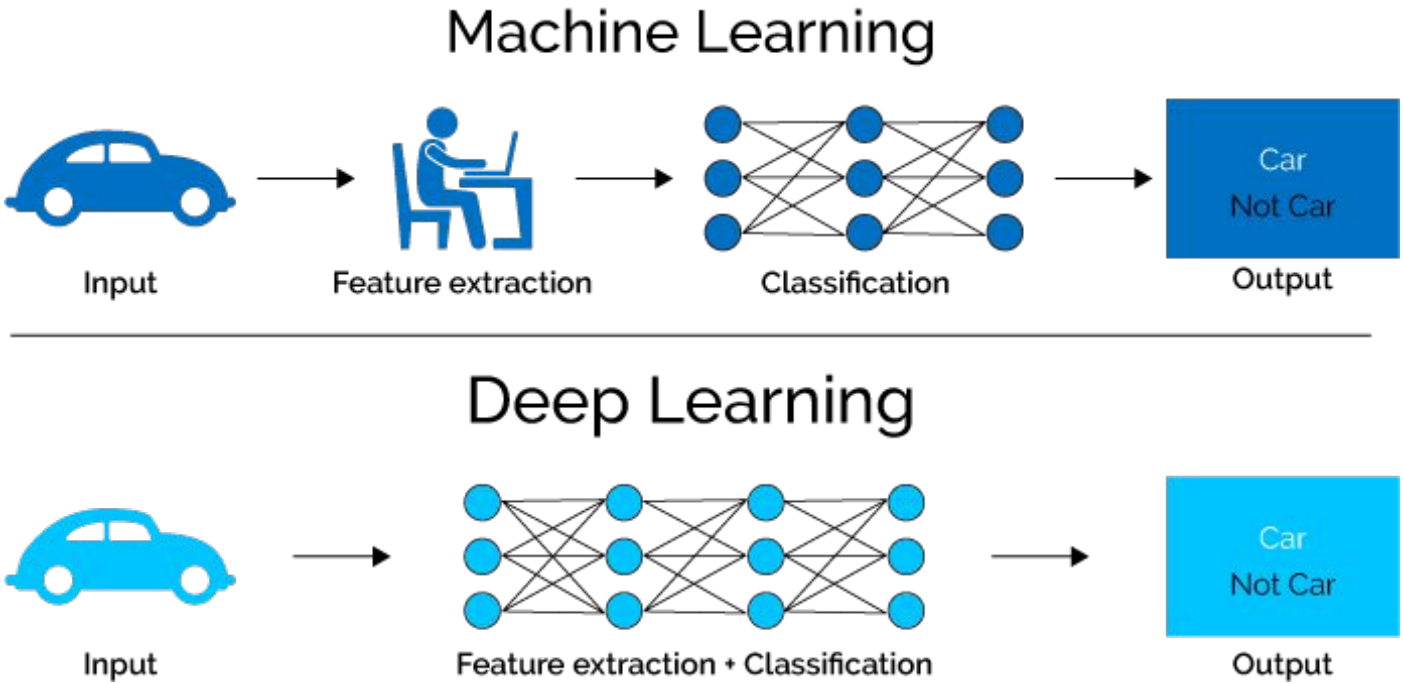
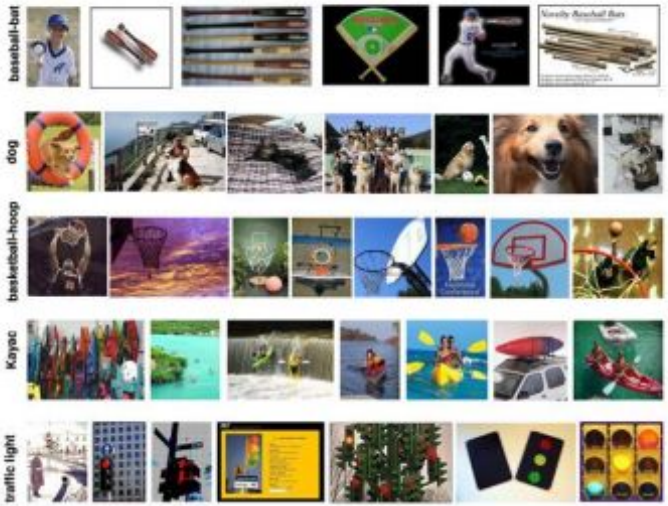


Image Classification: Caltech-101 Data

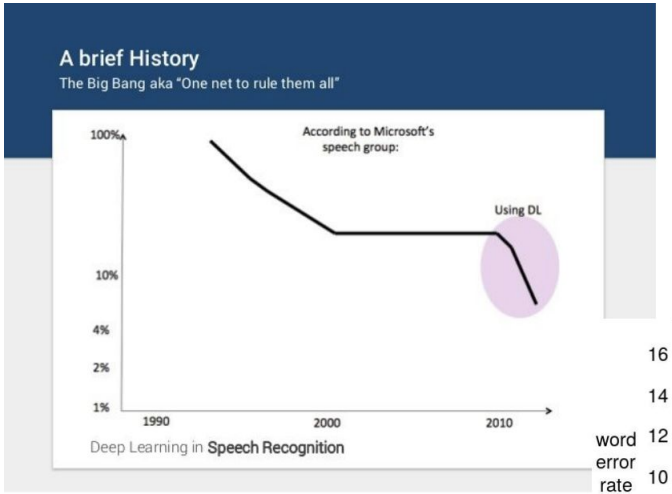
- Caltech-101 dataset
  - Around 10,000 images
  - Certainly not enough!



~80% is widely considered to be the limit on this dataset

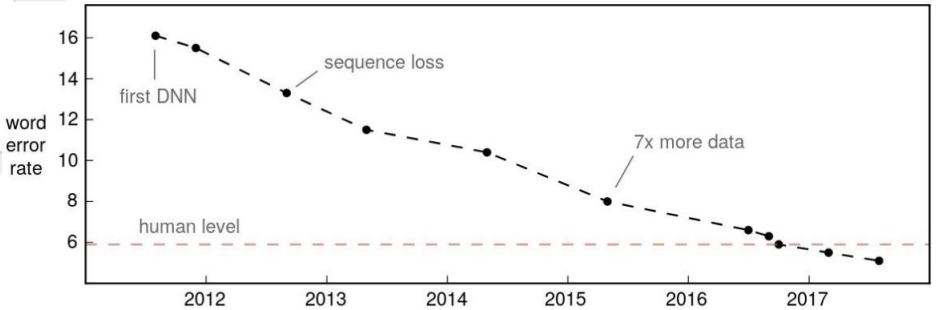
Algorithm	Accuracy (%)
SVM with Pyramid Matching Kernel (2005)	58.2%
Spatial Pyramid Matching (2006)	64.6%
SVM-KNN (2006)	66.2%
Sparse Coding + Pyramid Matching (2009)	73.2%
SVM Regression w object proposals (2010)	81.9%
Group-Sensitive MKL (2009)	84.3%
Deep Learning (pretrained on Imagenet) (2014)	91.4%

Speech recognition



Source: https://www.slideshare.net/LuMa921/deep-learning-the-past-present-and-future-of-artificial-intelligence

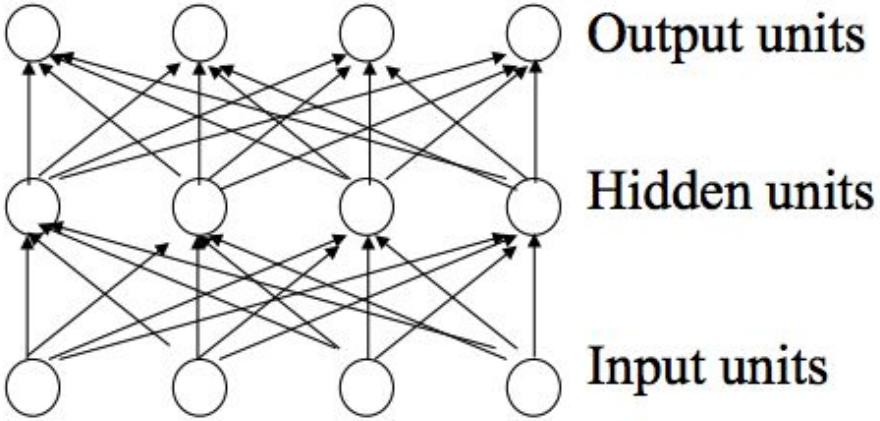
- Significant improvement in Word Error Rate (WER)
  - E.g., A WER of 5% roughly corresponds to 1 missed word for every 20
- Beating traditional approaches and even human?



Improvements in word error rate over time on the Switchboard conversational speech recognition benchmark. The test set was collected in 2000. It consists of 40 phone conversations between two random native English speakers.

Source: https://awni.github.io/speech-recognition/

Neural Network



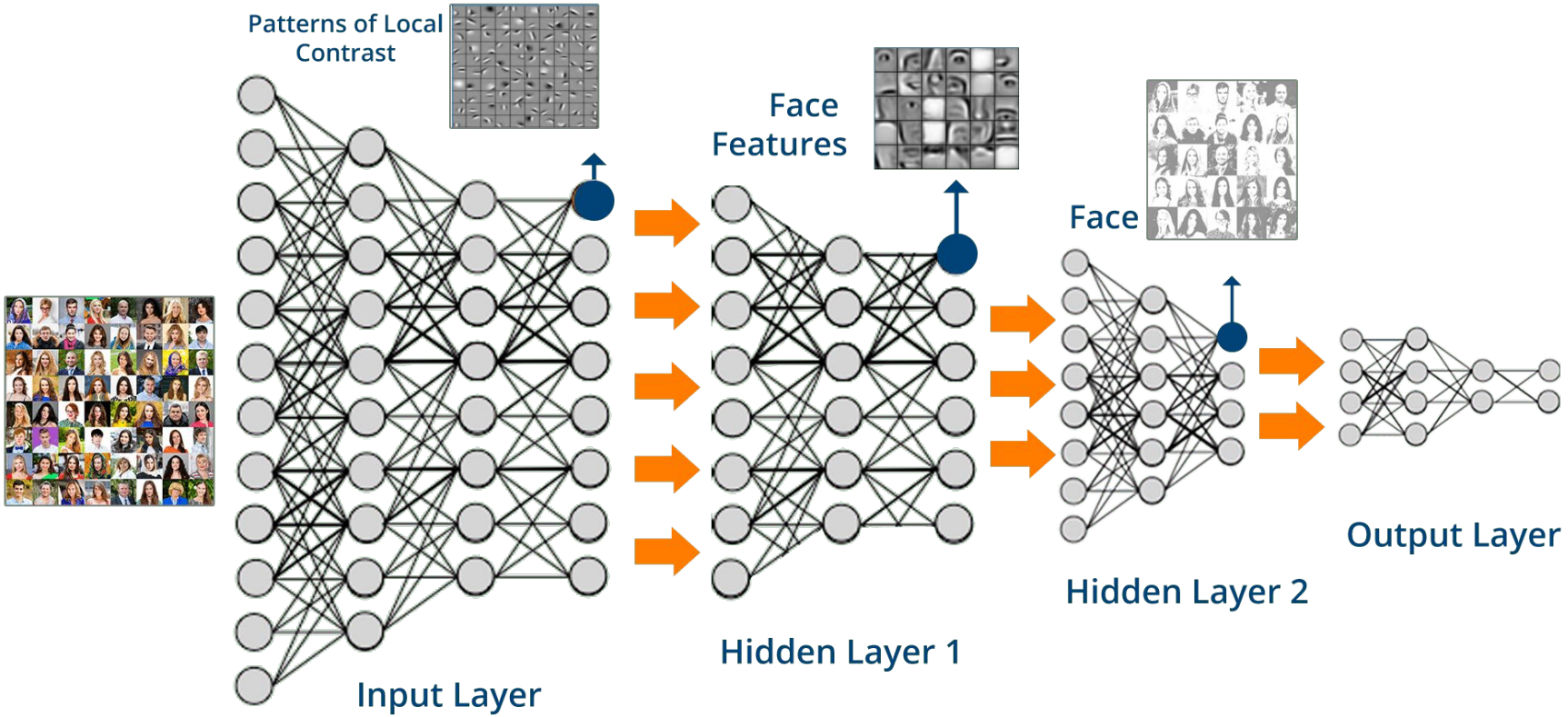
Layered feed-forward network

Neural networks are made up of **nodes** or **units**, connected by **links**

Each link has an associated **weight** and **activation level**

Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**

Deep Neural Network



Why now?

Neural Networks date back decades, so why the resurgence?

1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software

- Improved Techniques
- New Models
- Toolboxes

TensorFlow

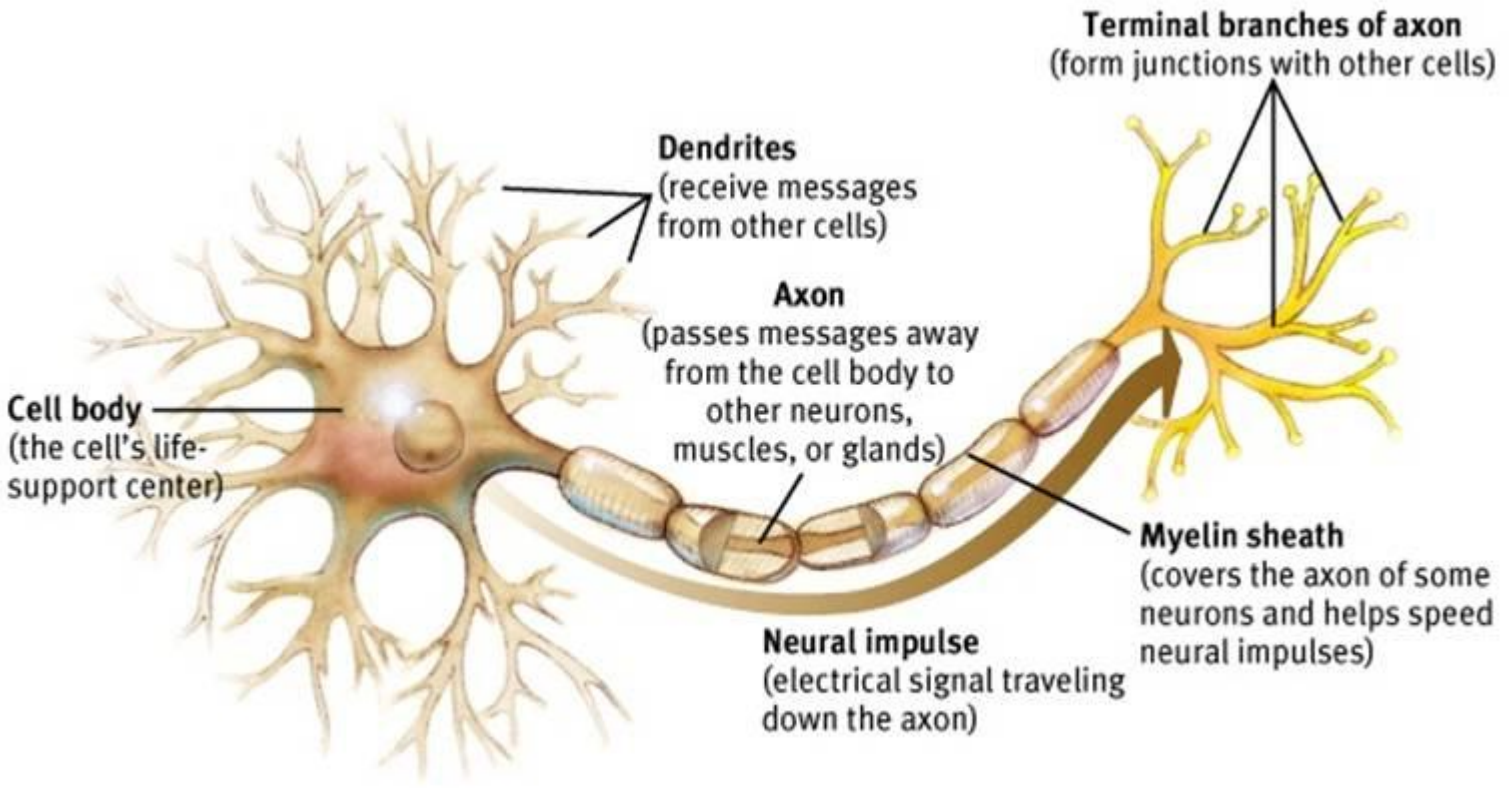
1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

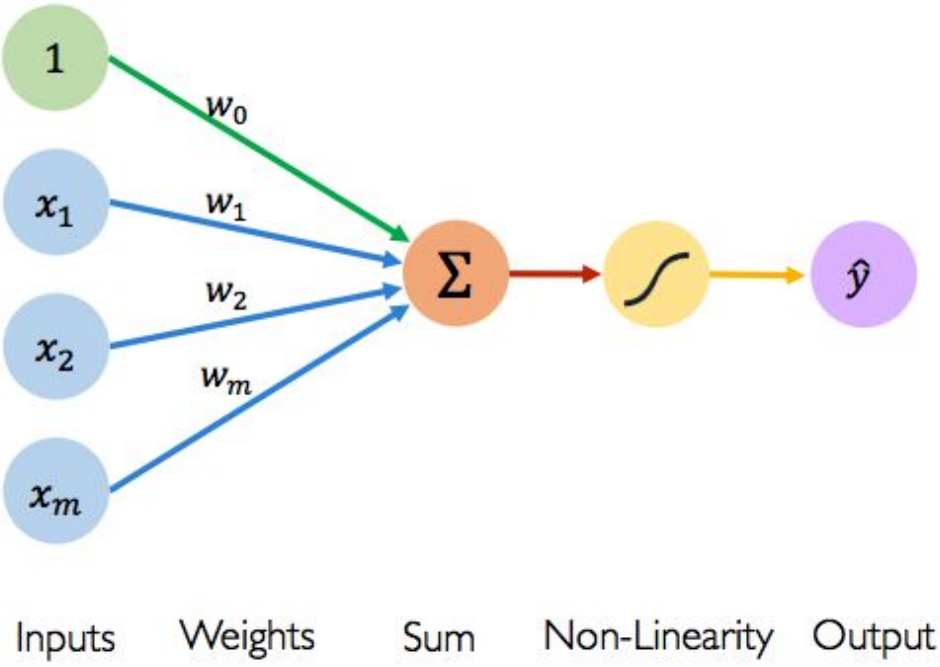
1986 Backpropagation
- Multi-Layer Perceptron

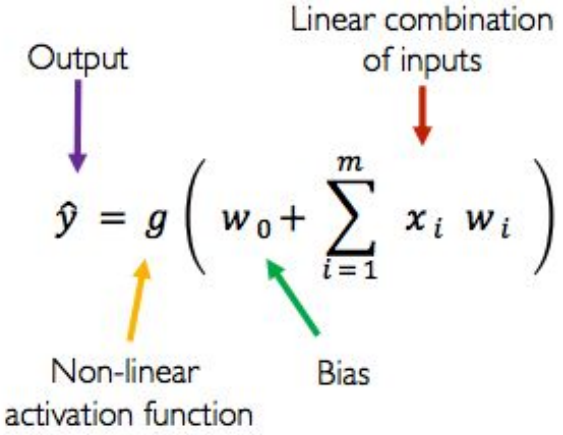
1995 Deep Convolutional NN
- Digit Recognition

Biology of a Neuron



The Perceptron: Forward Propagation

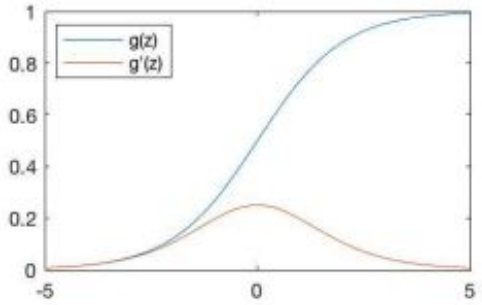




Foster School of Business

Common Activation Functions

Sigmoid Function

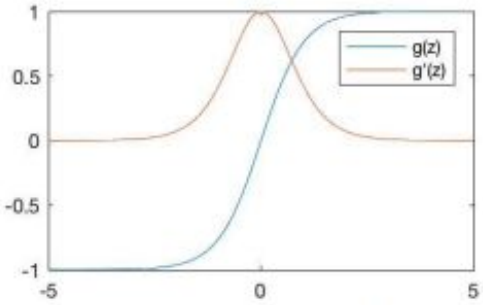


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

tf.nn.sigmoid(z)

Hyperbolic Tangent

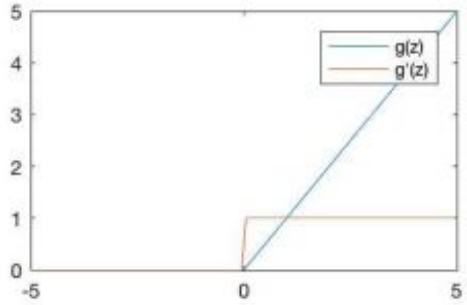


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)

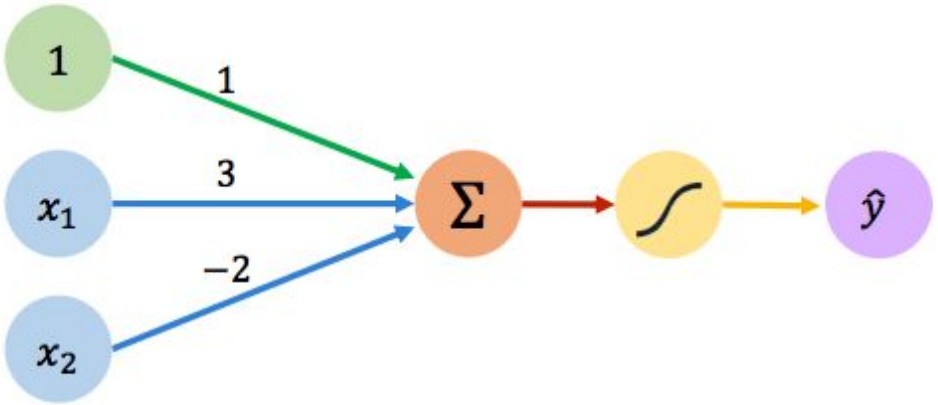


$$g(z) = \max(0, z)$$

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

tf.nn.relu(z)

Perceptron Binary Classification Example



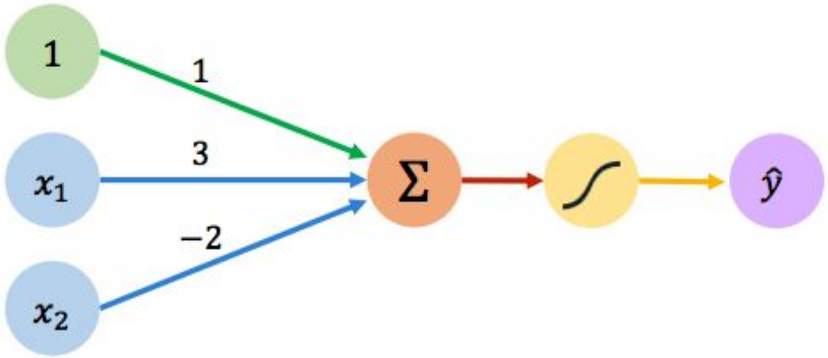
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

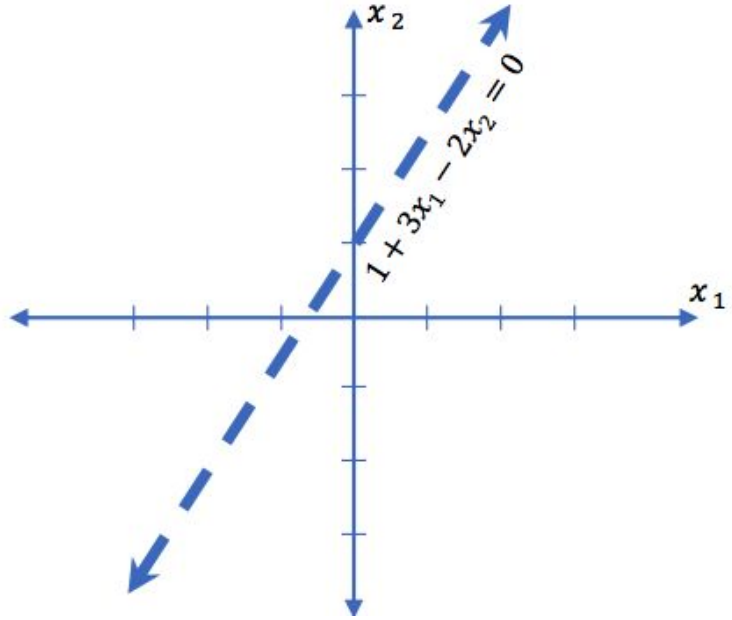
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

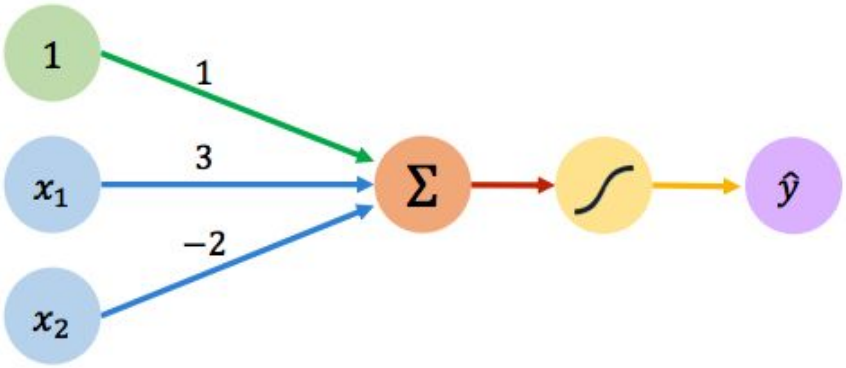
Perceptron Binary Classification Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



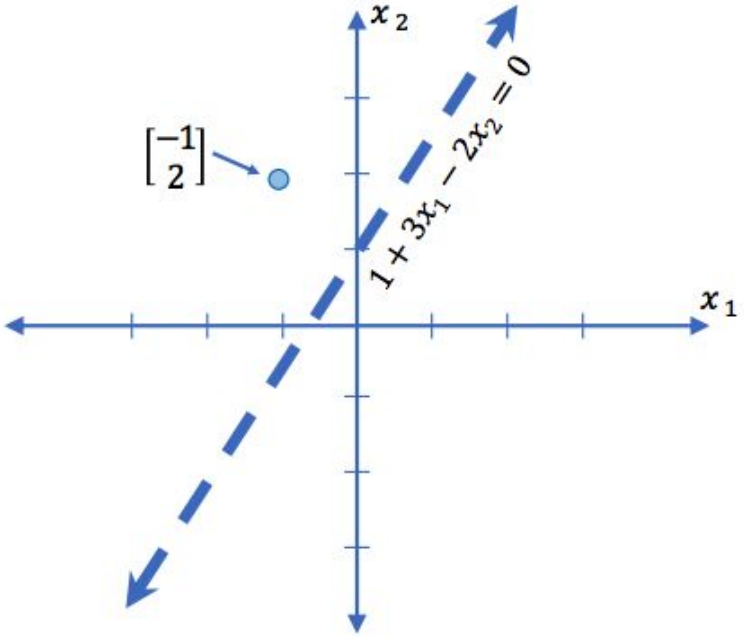
Perceptron Binary Classification Example



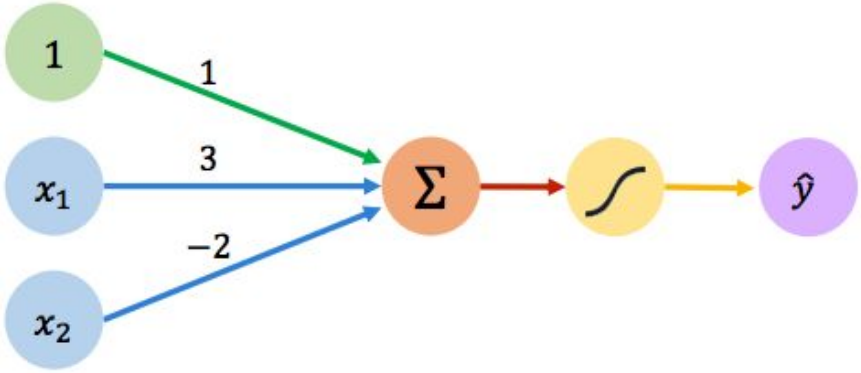
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

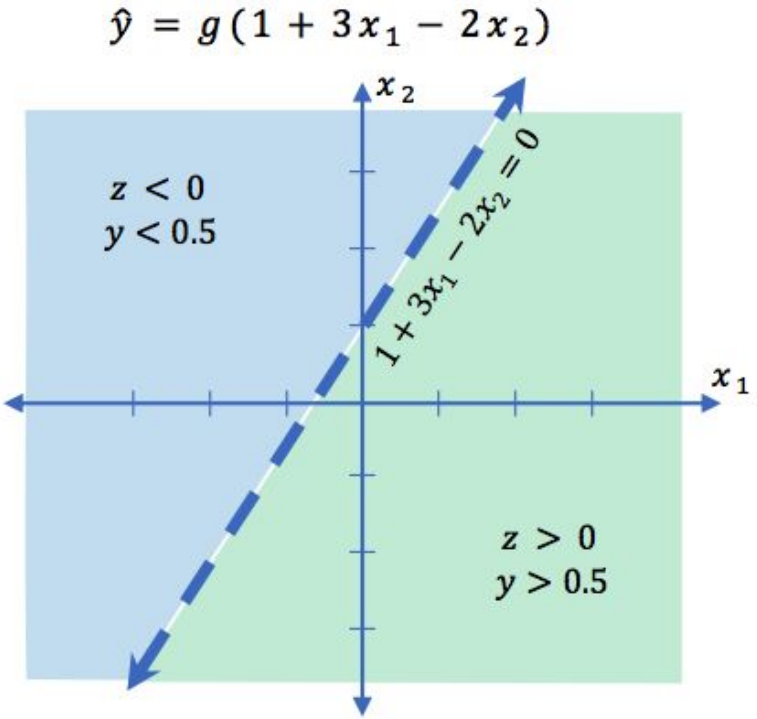
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

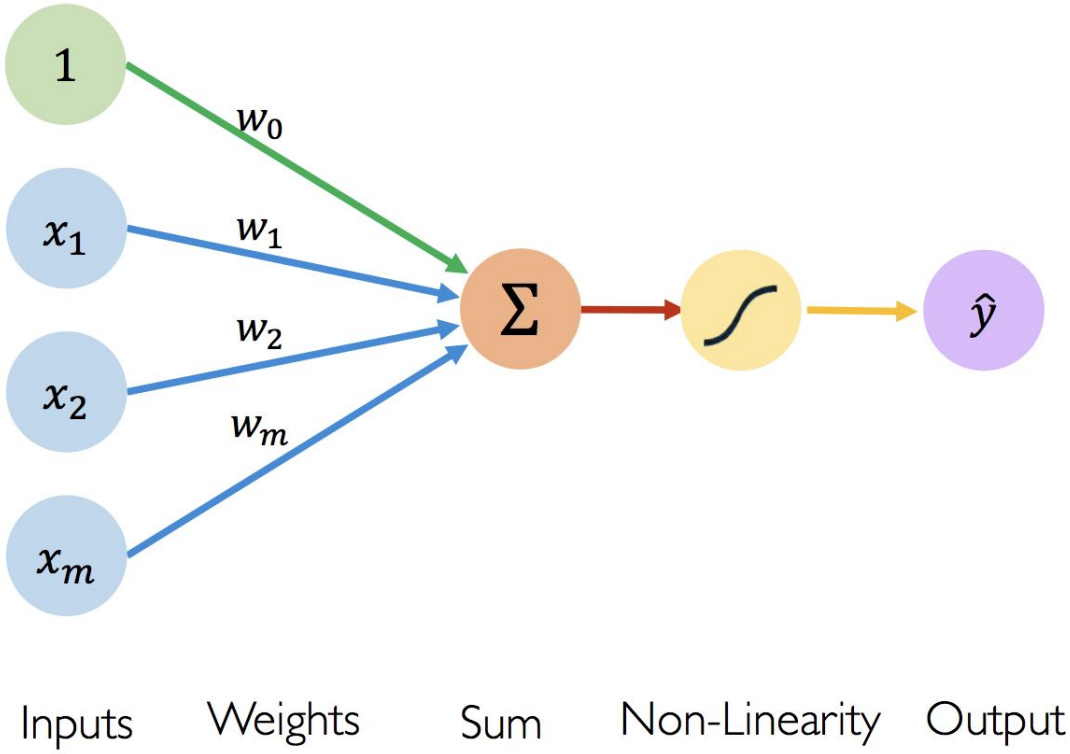


Perceptron Binary Classification Example

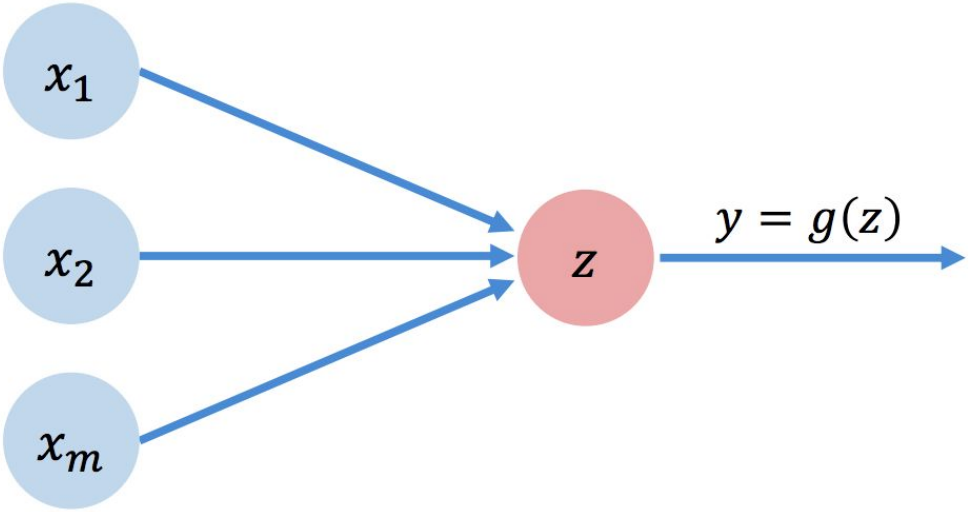




Perceptron Model

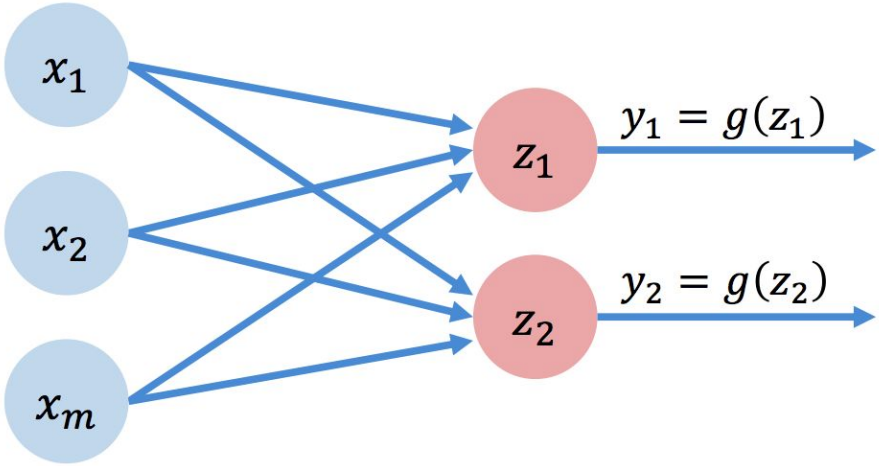


Single Output Perceptron



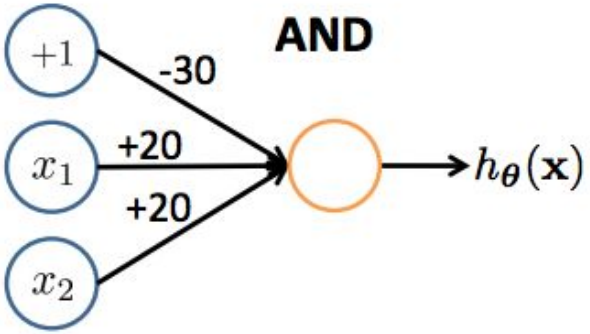
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

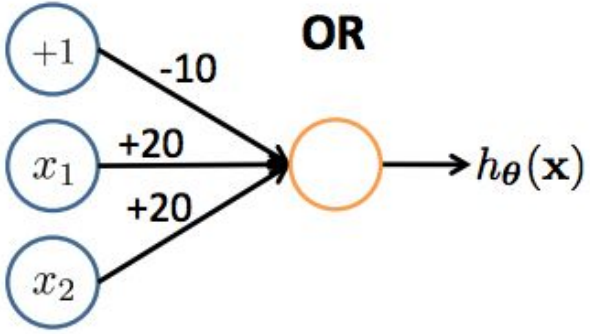
Multi Output Perceptron

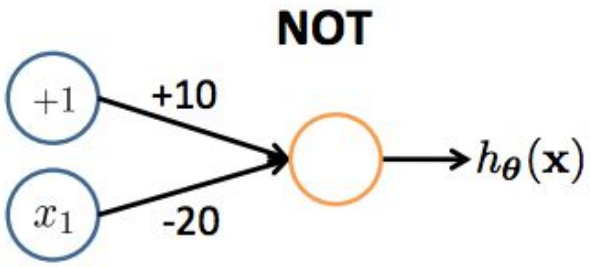


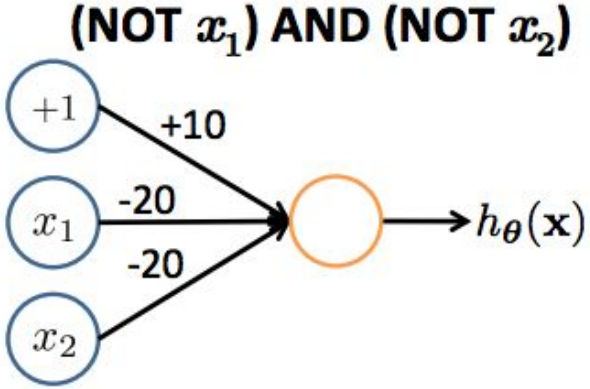
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Perceptrons Represent Boolean Functions









XOR Function

XOR truth table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

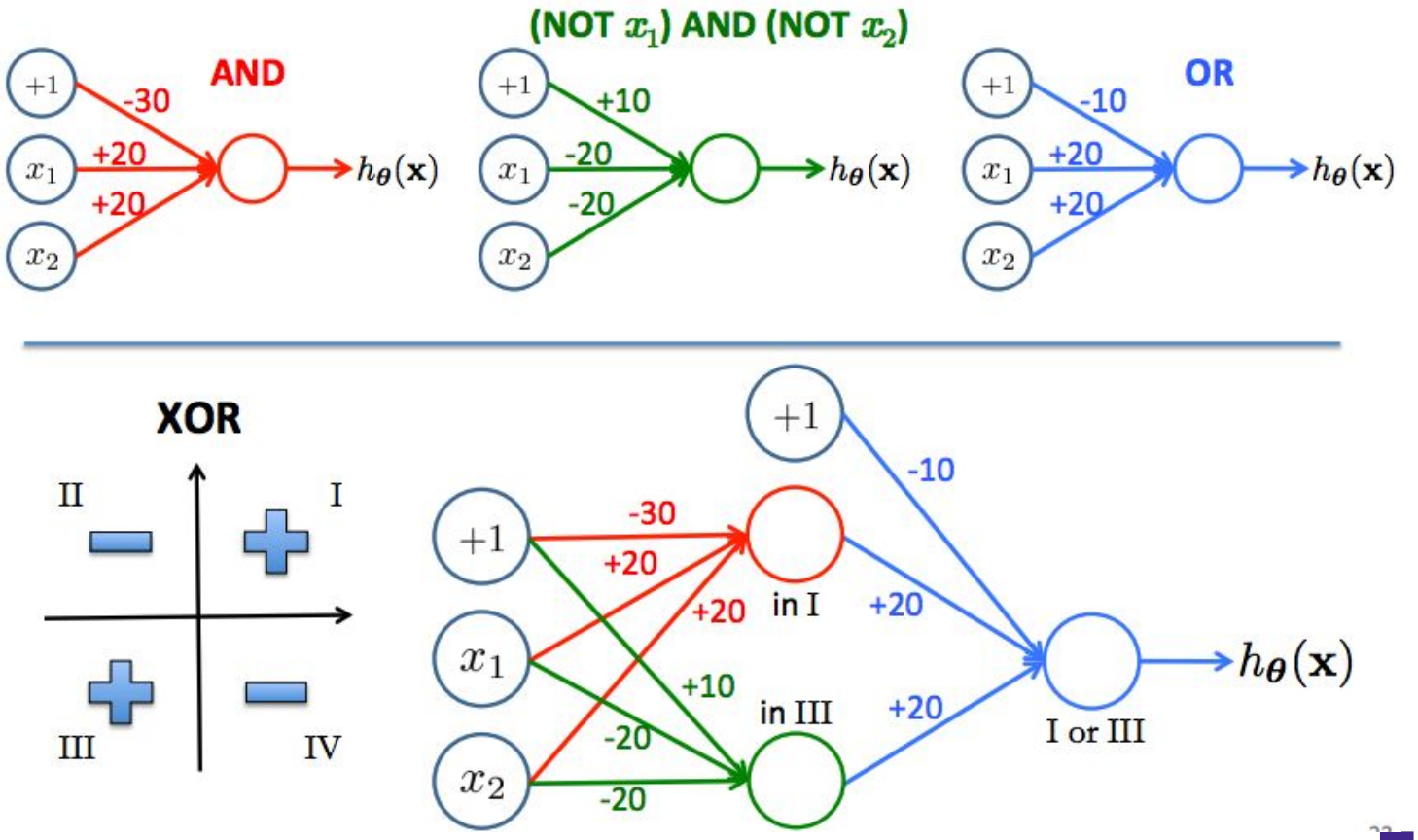
$$
\begin{aligned}
p \oplus q &= (p \wedge \neg q) \vee (\neg p \wedge q) \\
&= ((p \wedge \neg q) \vee \neg p) \wedge ((p \wedge \neg q) \vee q) \\
&= ((p \vee \neg p) \wedge (\neg q \vee \neg p)) \wedge ((p \vee q) \wedge (\neg q \vee q)) \\
&= (\neg p \vee \neg q) \wedge (p \vee q) \\
&= \neg(p \wedge q) \wedge (p \vee q)
\end{aligned}
$$

$$\neg(p \oplus q) = (p \wedge q) \vee (\neg p \wedge \neg q)$$

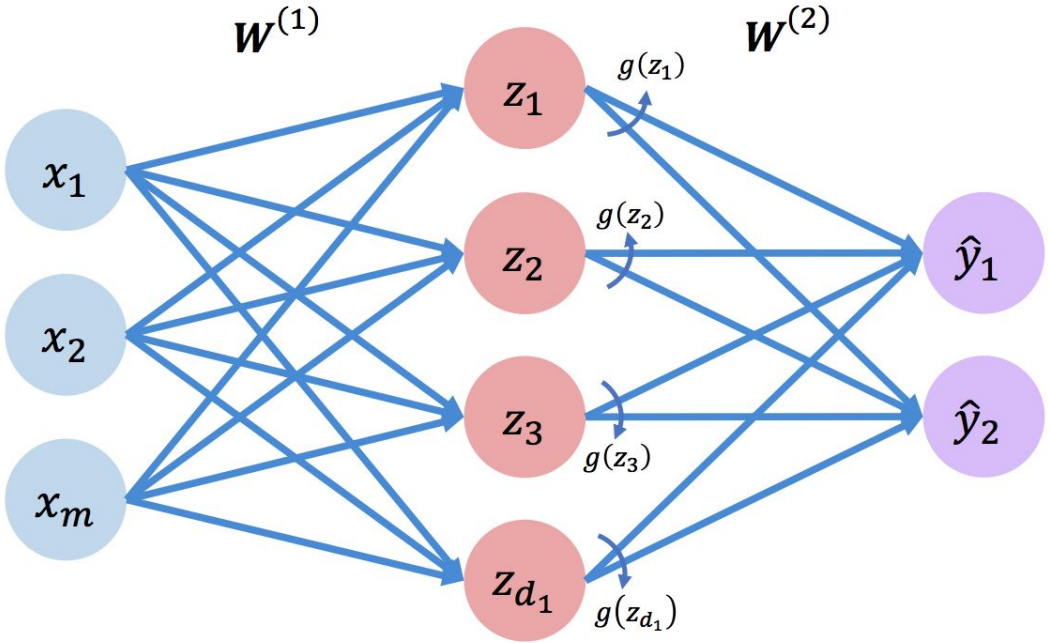
AND (NOT X1) AND (NOT X2)

OR

Create Non-Linear Functions from Linear Ones



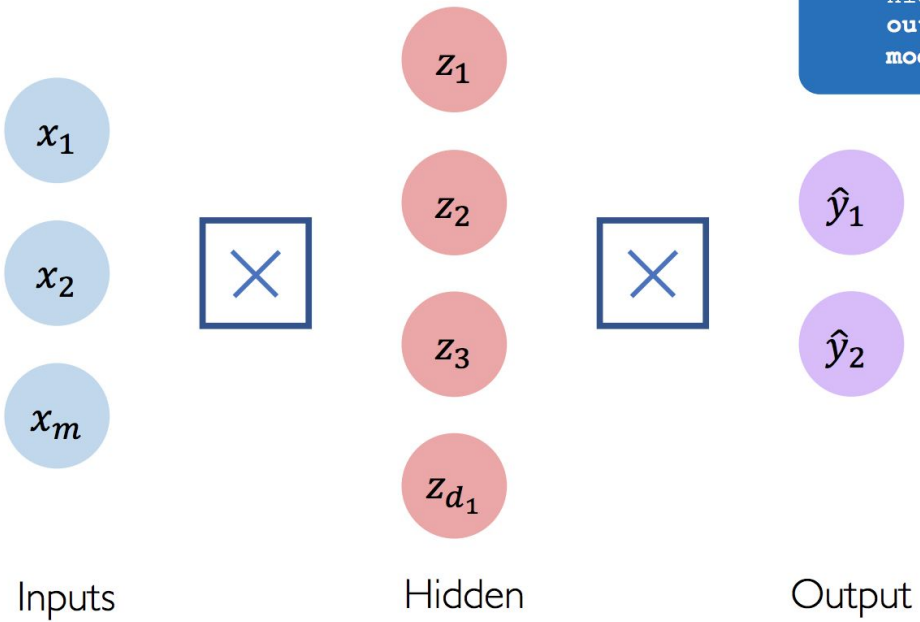
Single Layer Neural Network



Inputs Hidden Final Output

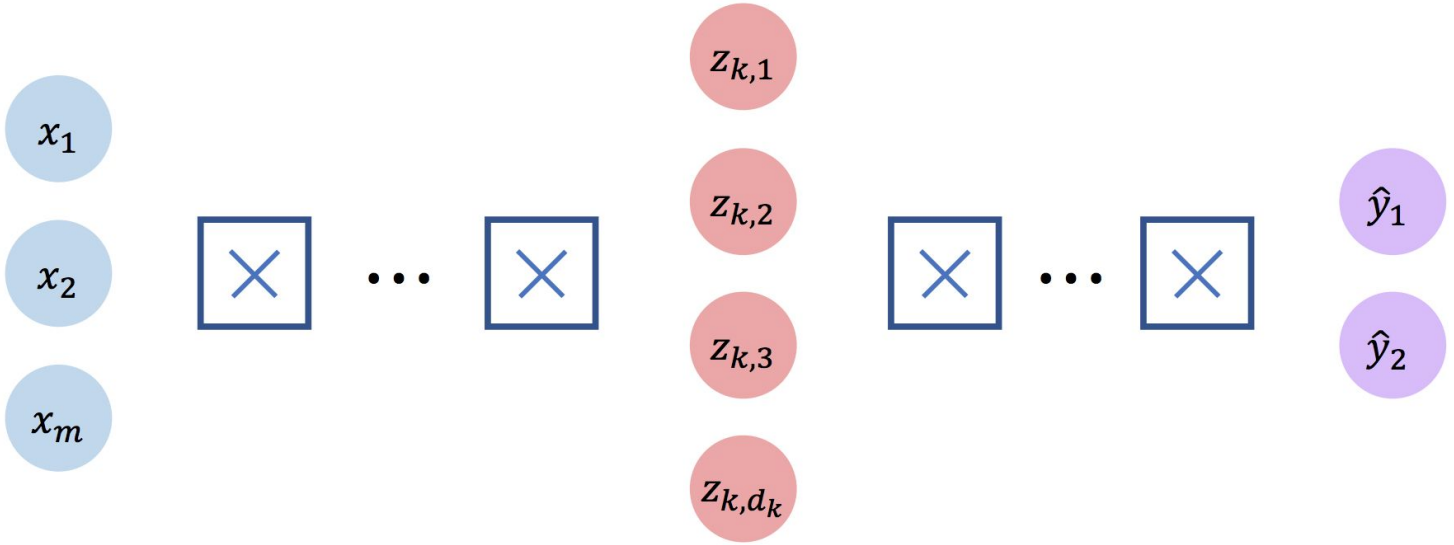
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)} \right)$$

Single Layer Neural Network



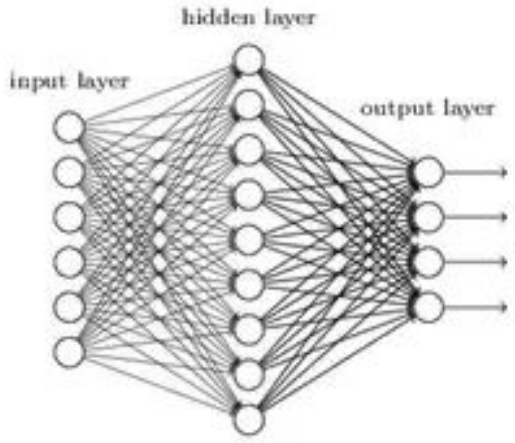

```
from tf.keras.layers import *

inputs = Inputs(m)
hidden = Dense(d1)(inputs)
outputs = Dense(2)(hidden)
model = Model(inputs, outputs)
```

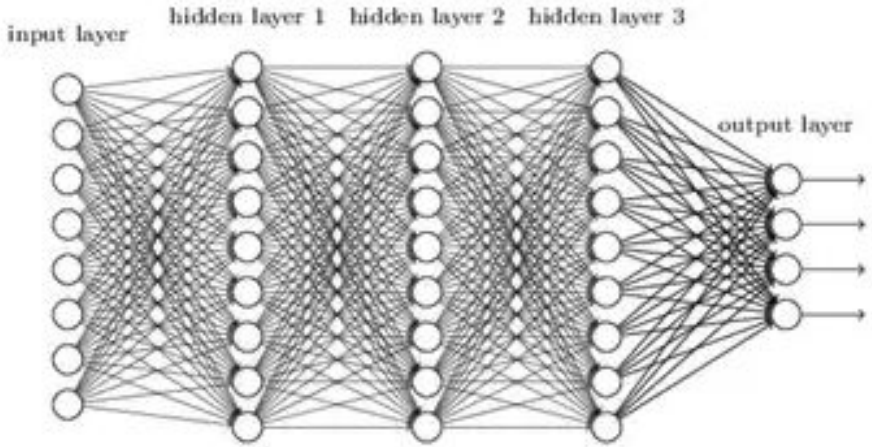
Multi-Layer (Deep) Neural Network



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

Shallow vs. Deep Networks





- Advantages of Deep Networks:
  - Deep representation allows for a hierarchy of representations.
  - Very wide and shallow networks are very good at memorization while deep networks are better at generalization.
  - Often outperform shallow ones with the same computational power.

Summary

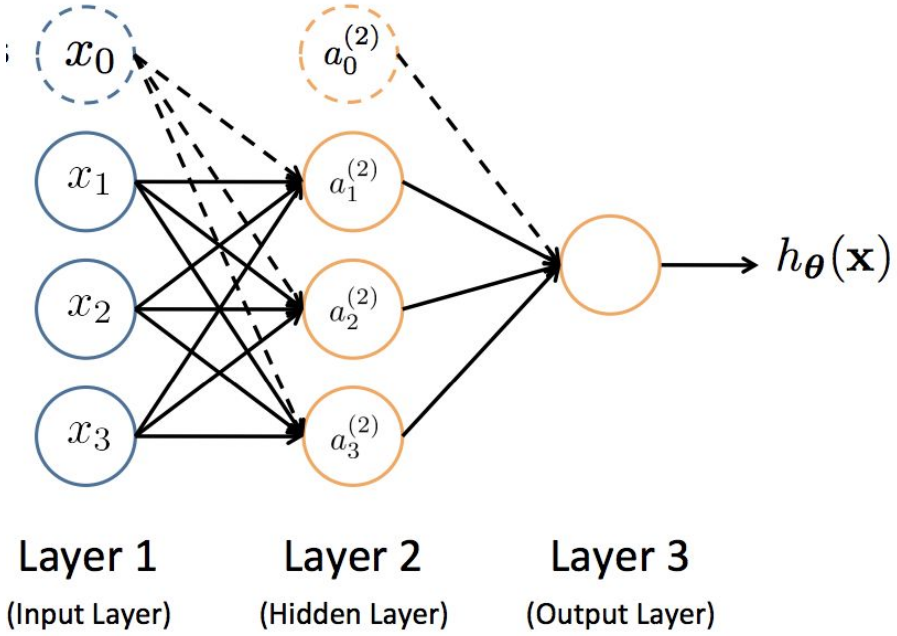
- Neural Networks try to mimic how human neurons work.
- Neural Networks gained huge success in perceptual problems, e.g. Image classification, speech recognition and etc.
- Resurgence of Neural Networks is due to big data, hardware and software.
- Perception is the building block for Neural Networks.
- Deep Neural Networks outperforms shallow ones.

Neural Network Fundamentals

Neural Networks

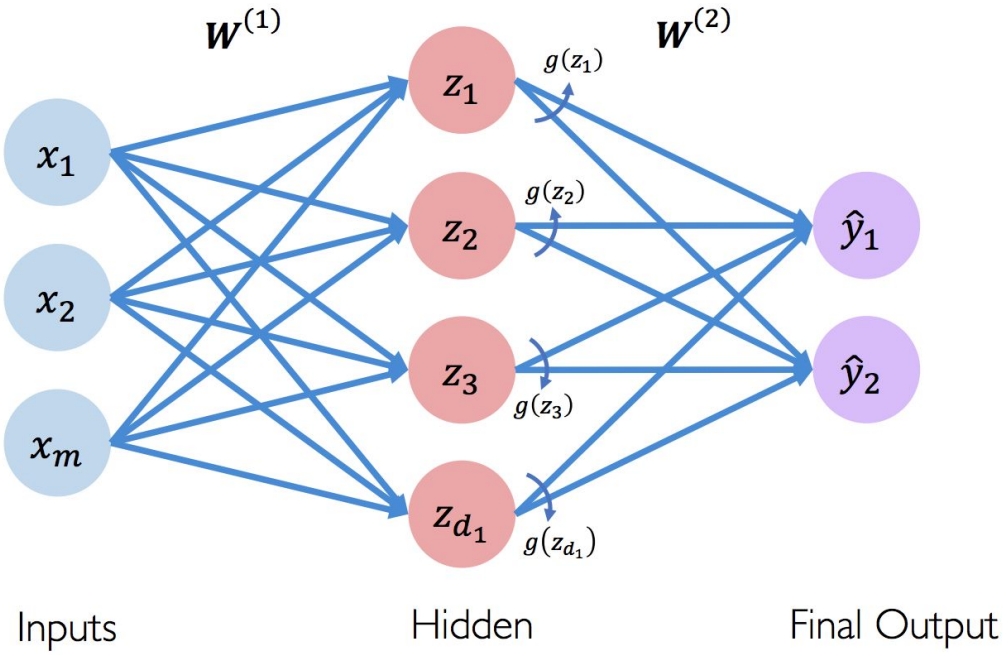
- Origins: Algorithms that try to mimic the brain.
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
- Artificial neural networks are not nearly as complex as the actual brain structure

Universal Approximation Theorem

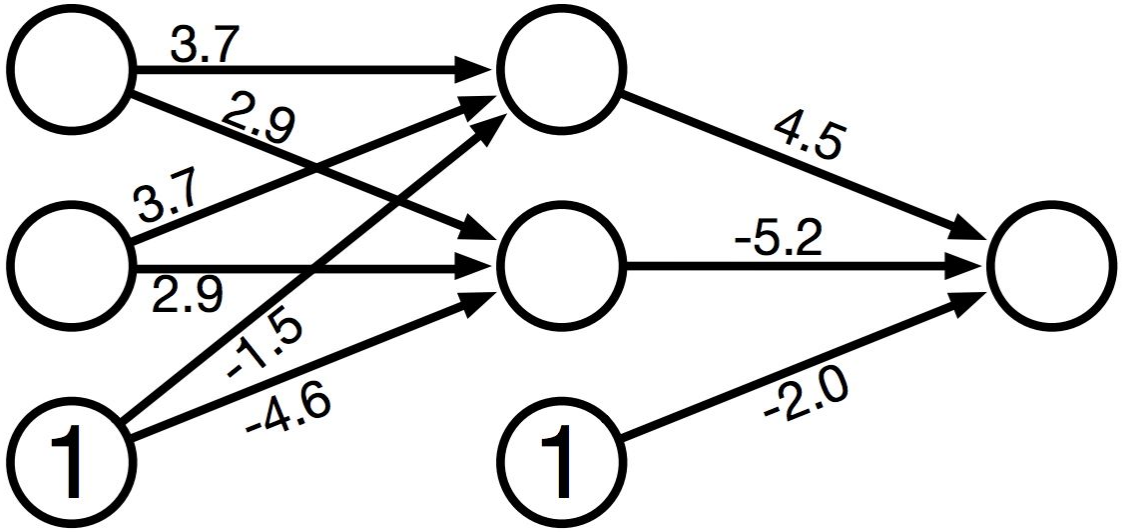


The universal approximation theorem found that a neural network with one hidden layer can approximate any continuous function (George Cybenko, 1989).

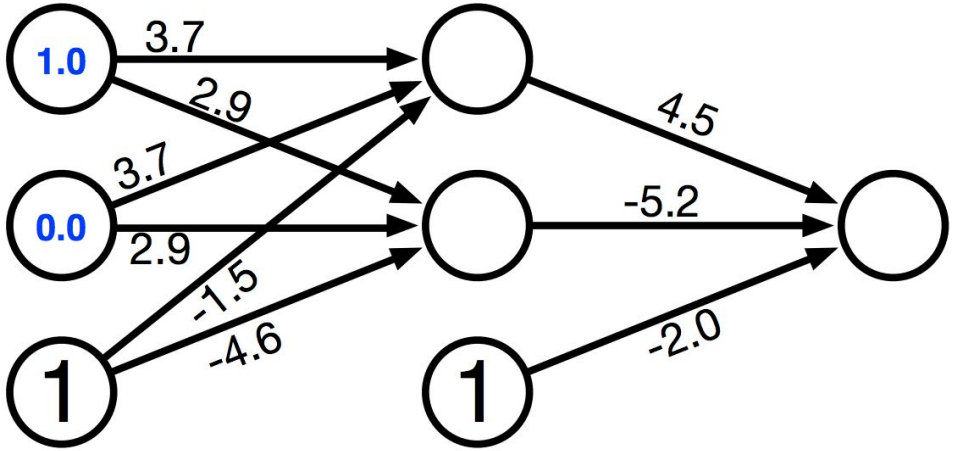
Neural Networks Forward Propagation



Neural Network Example



Neural Network Example

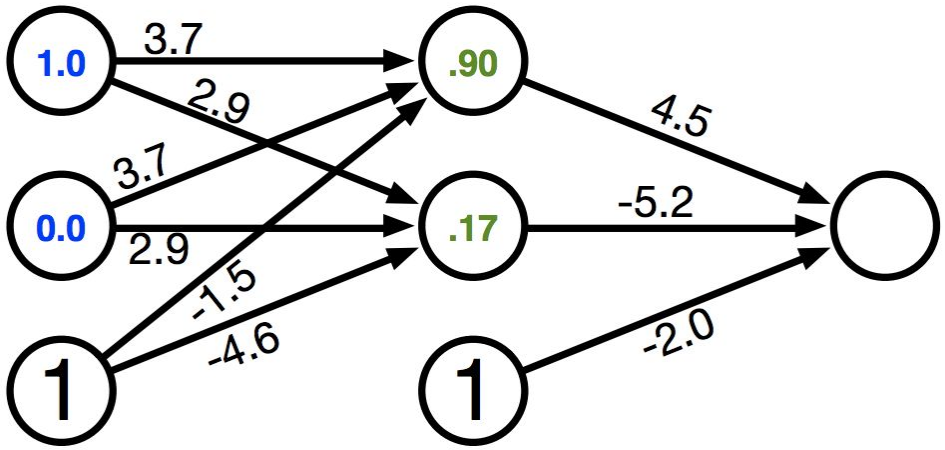


Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

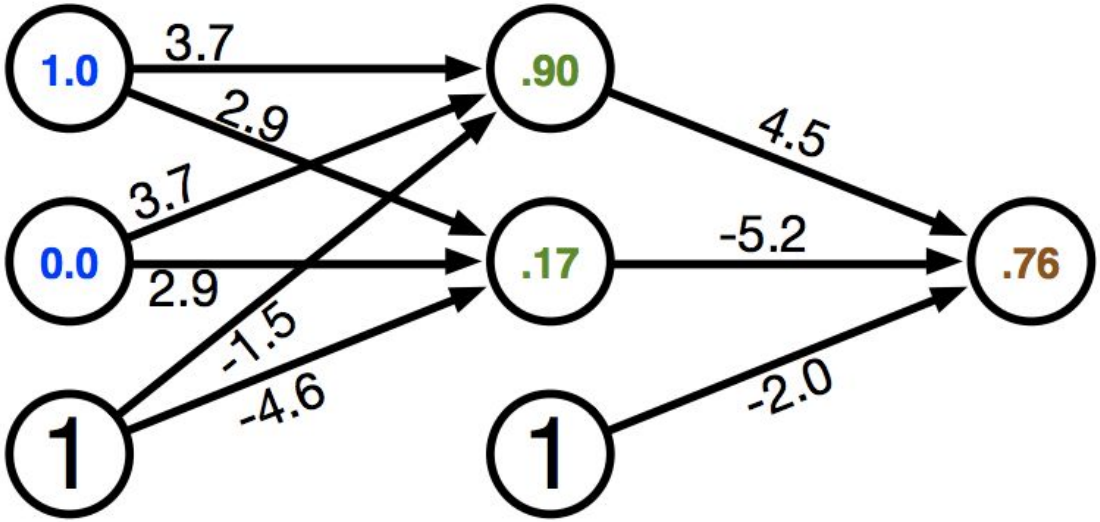
Neural Network Example



Output computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Neural Network Example



Neural Network Example

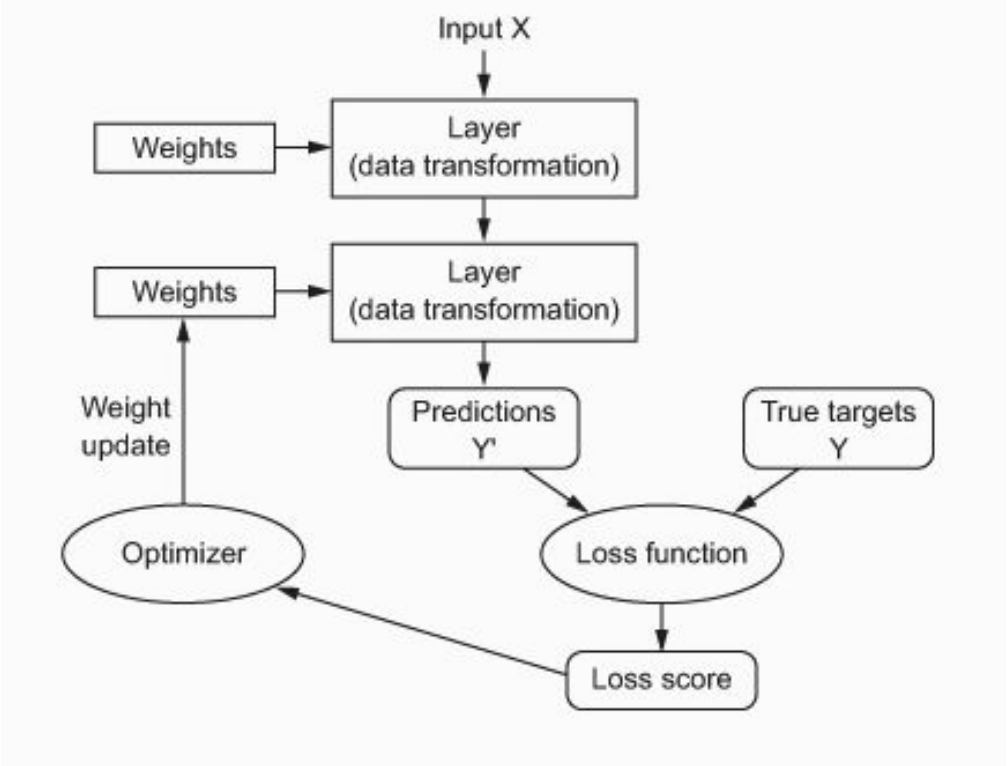
Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- Network is another implementation of XOR:
  - hidden node h_0 is OR
  - hidden node h_1 is AND
  - final layer operation is $h_0 - h_1$

How do we learn those weights on the edges in a Neural Network?

Training Neural Networks

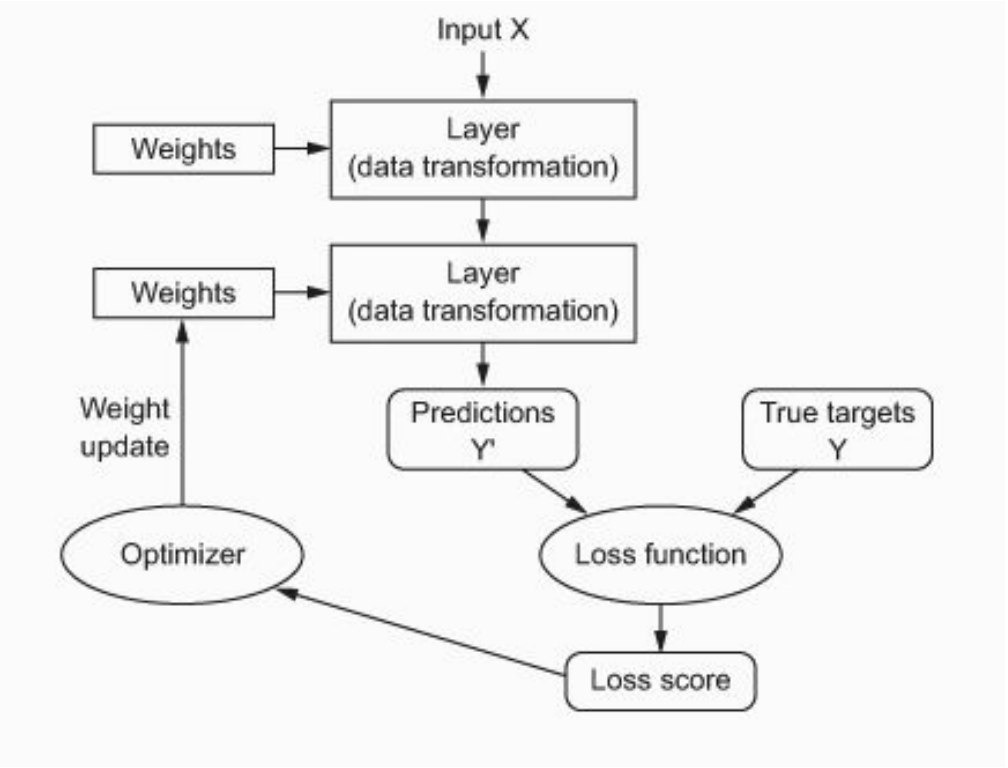
Learning means finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets.



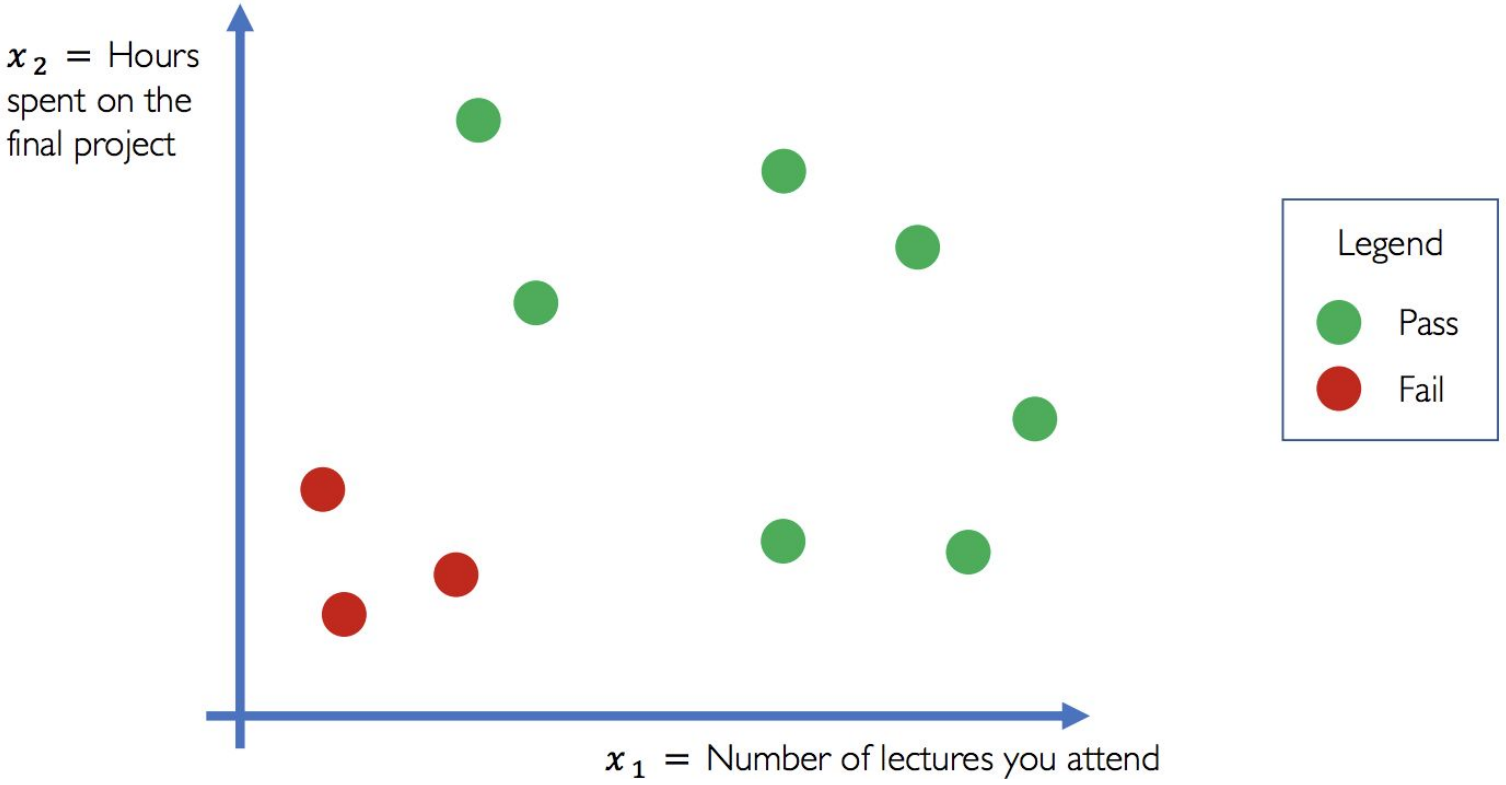
Neural Networks: Loss Function

Loss Function

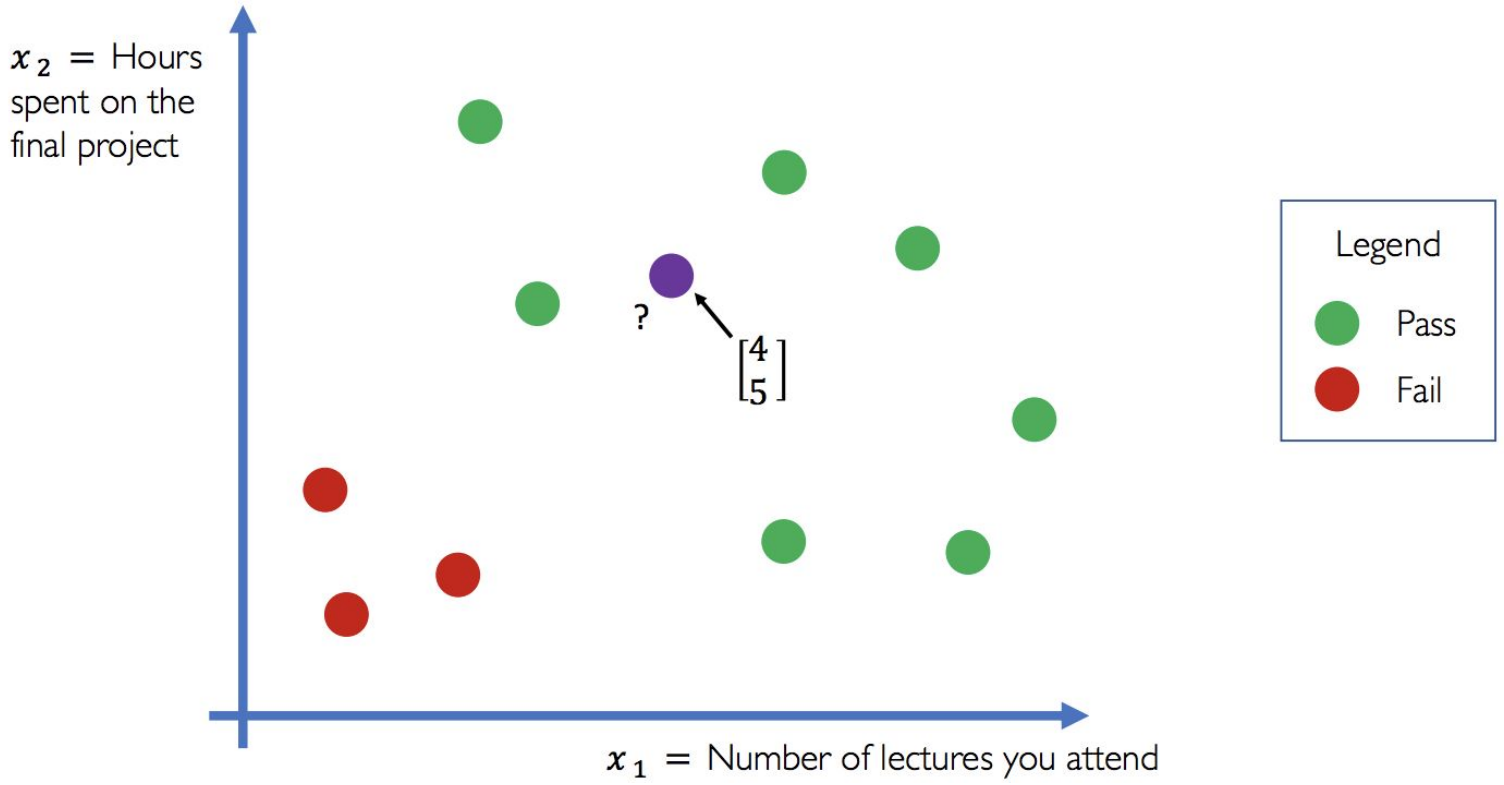
The **loss** is the quantity we attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.



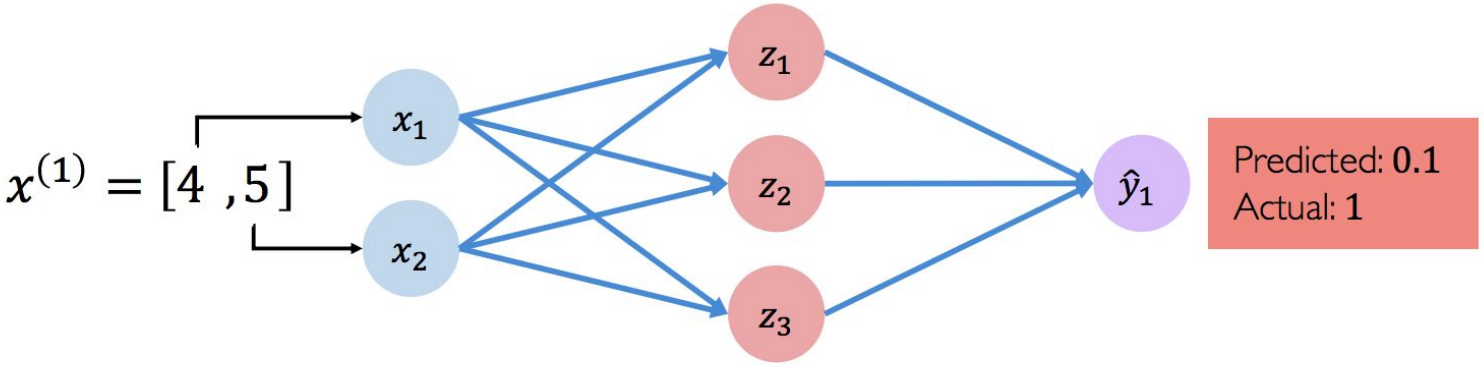
Will I pass this class?



Will I pass this class?

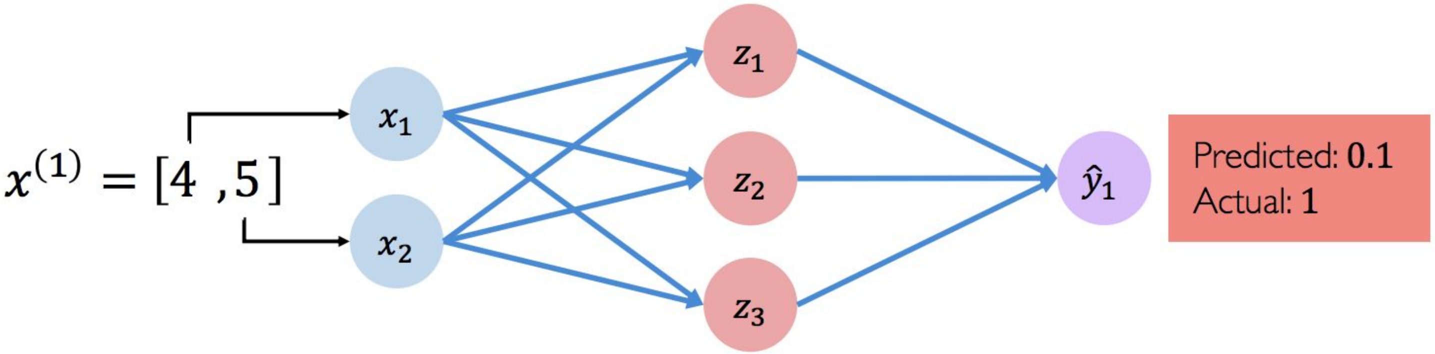


Will I pass this class?



Quantifying Loss

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

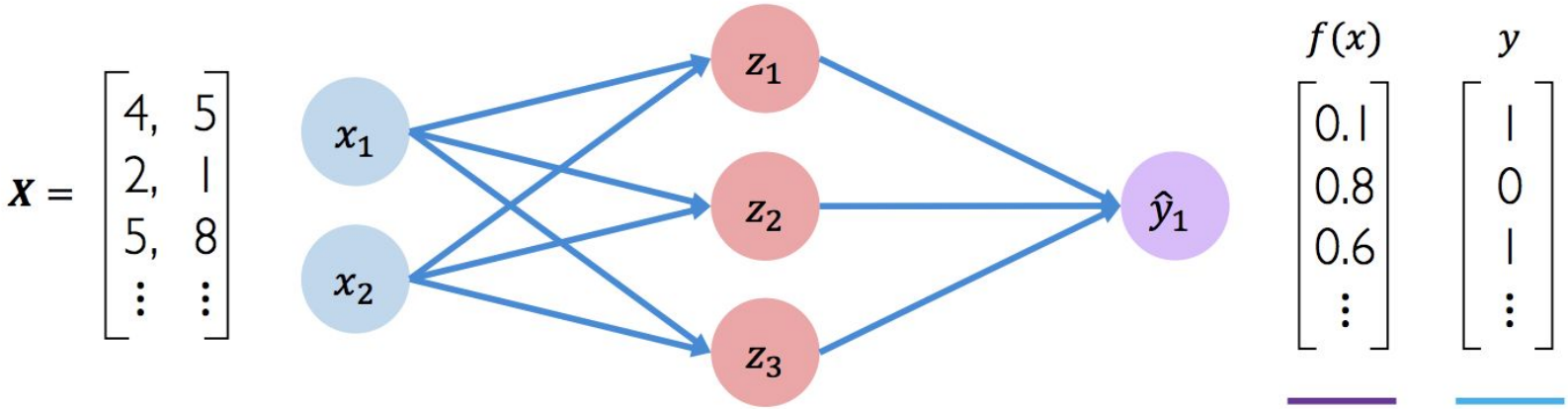


$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

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



Also known as:

- Objective function
- Cost function
- Empirical Risk

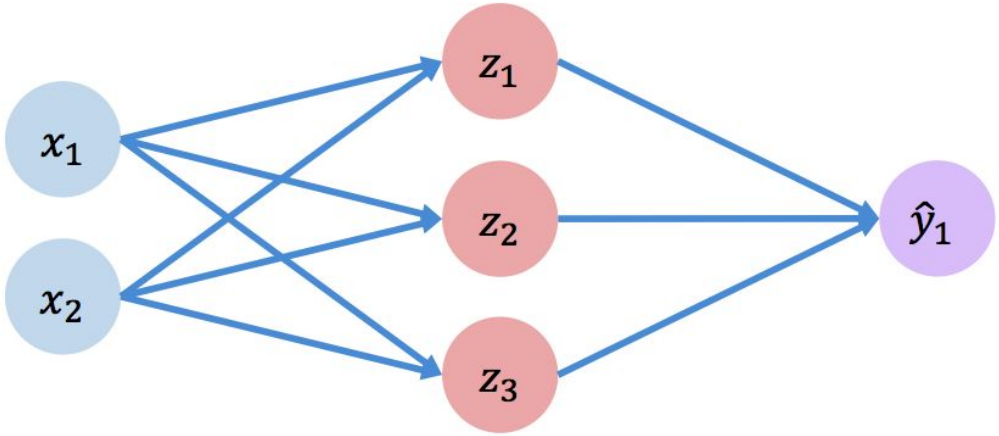
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Binary Cross Entropy Loss

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

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



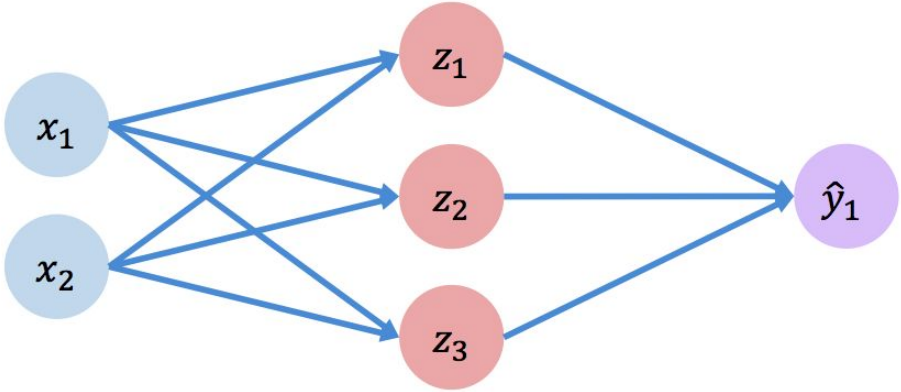
$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

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

Mean Squared Error Loss

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

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



$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}\right)^2$$

$f(x)$	y
30	90
80	20
85	95
⋮	⋮

Final Grades (percentage)

Learning Neural Networks

- Learning means finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets.

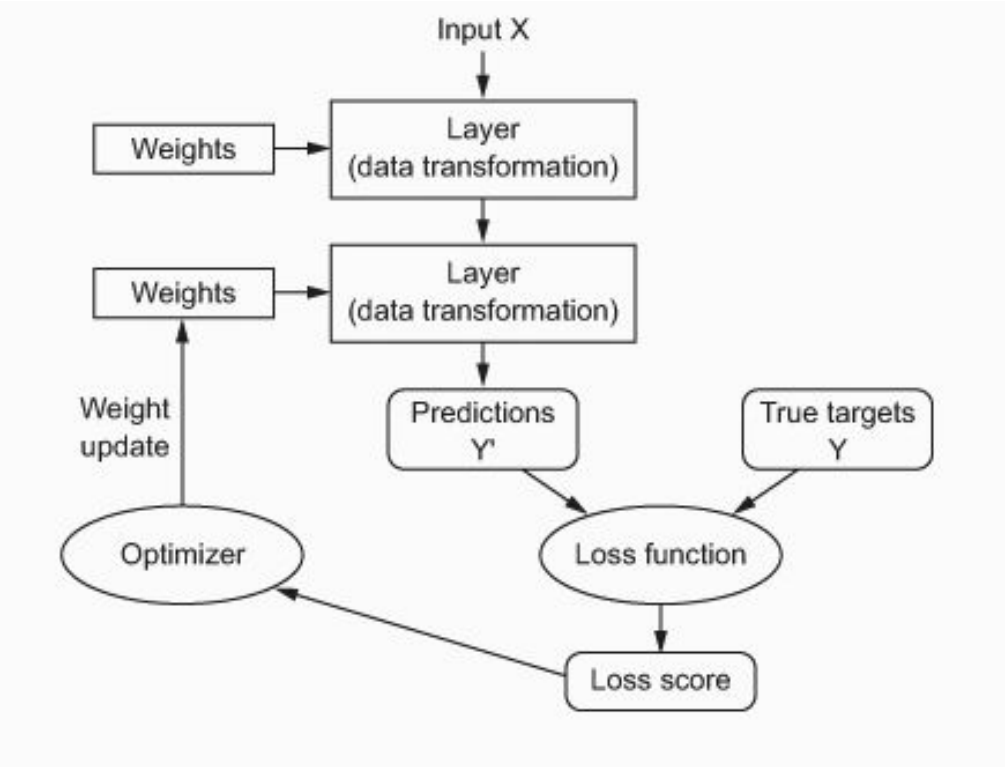
- The loss is the quantity you'll attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.

- The optimizer specifies the exact way in which the gradient of the loss will be used to update parameters.

Neural Networks: Optimization

Optimizer

The **optimizer** specifies the exact way in which the gradien of the loss will be used to update parameters.



Loss Optimization

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

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

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

Remember:

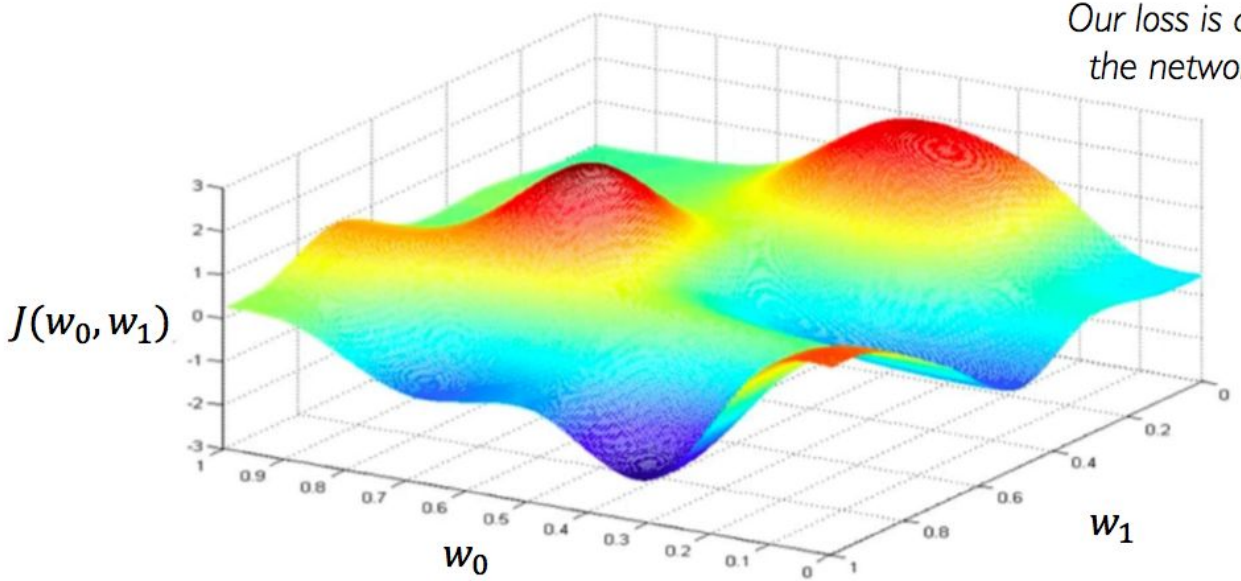
$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \ldots\right\}$$

Loss Optimization

Find the network weights that achieve the lowest loss.

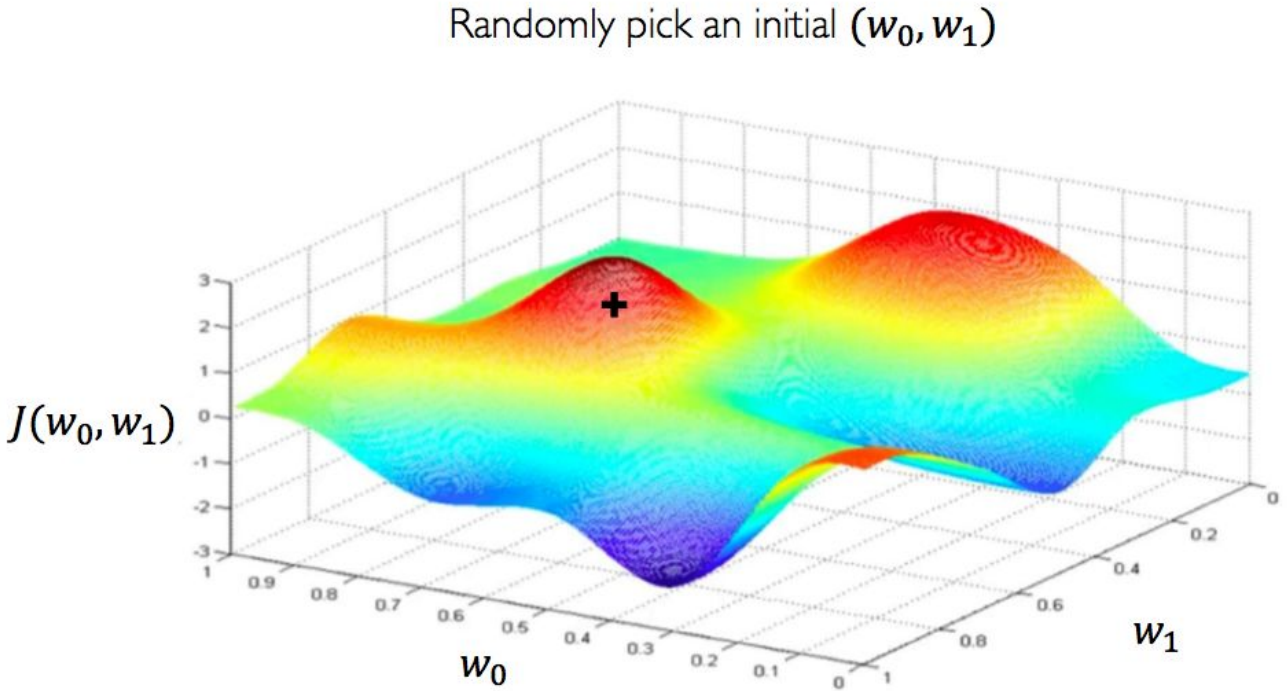
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

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



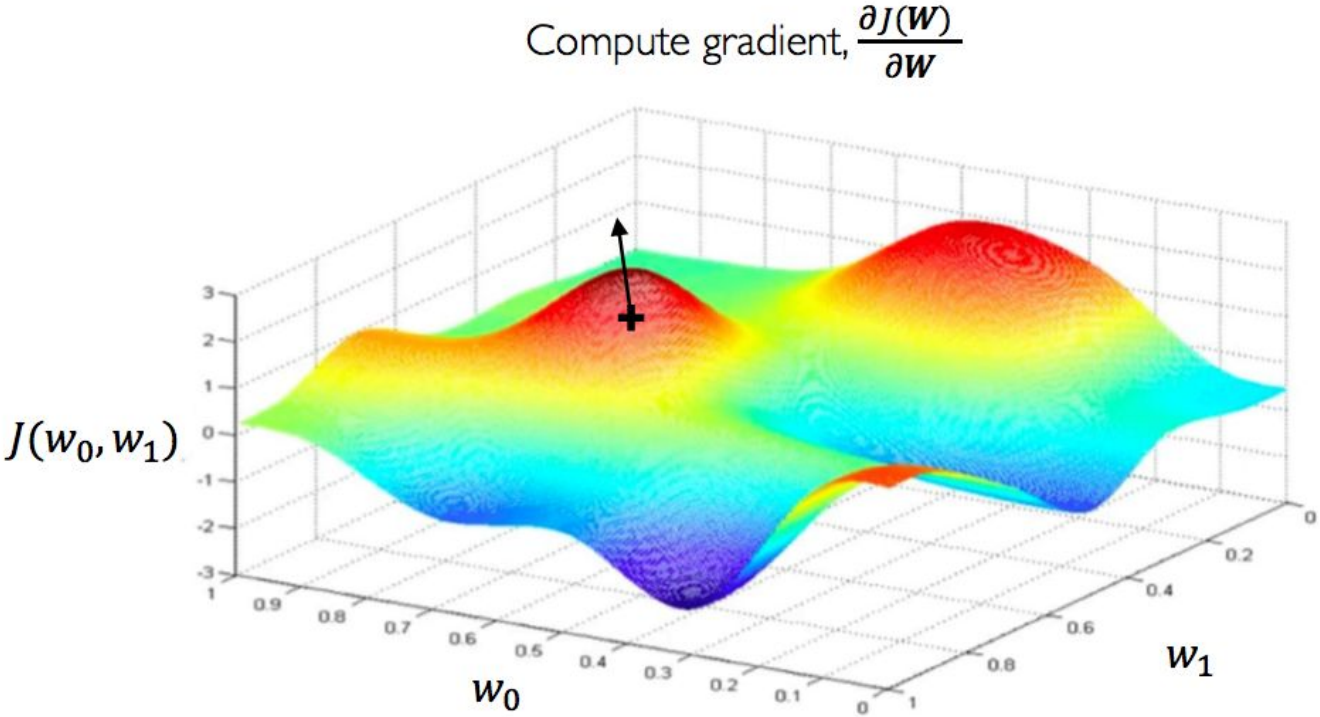
Loss Optimization

Find the network weights that achieve the lowest loss.



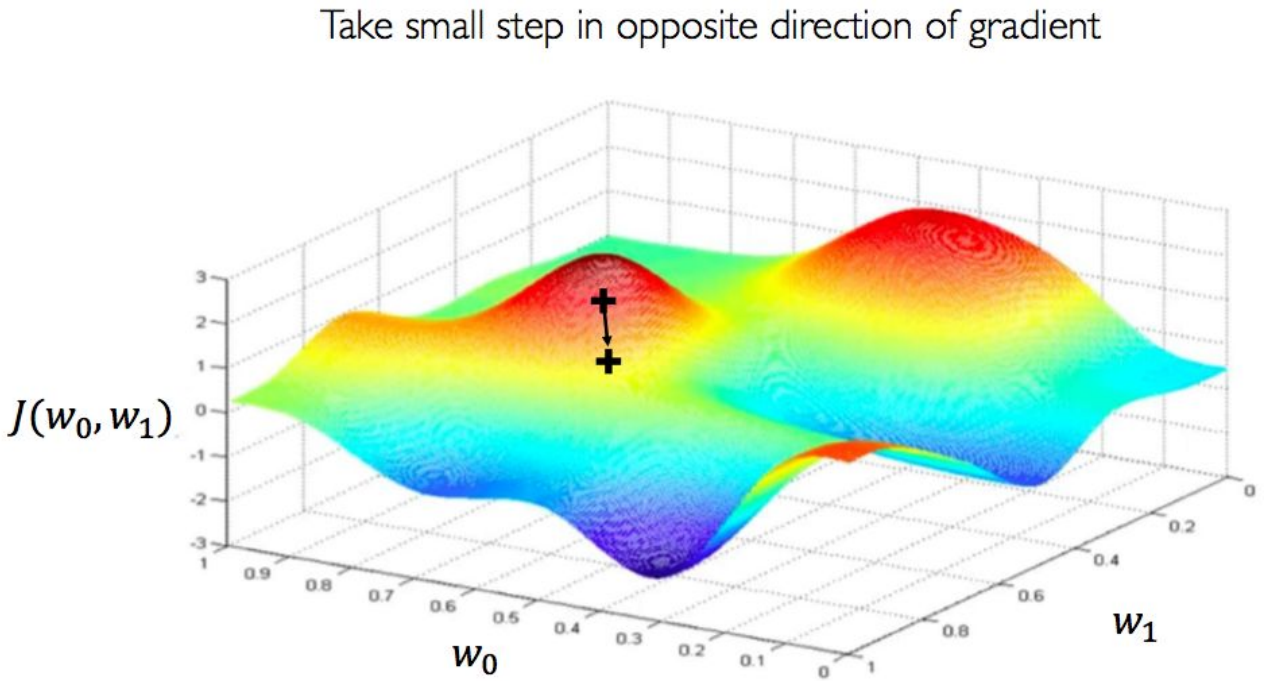
Loss Optimization

Find the network weights that achieve the lowest loss.



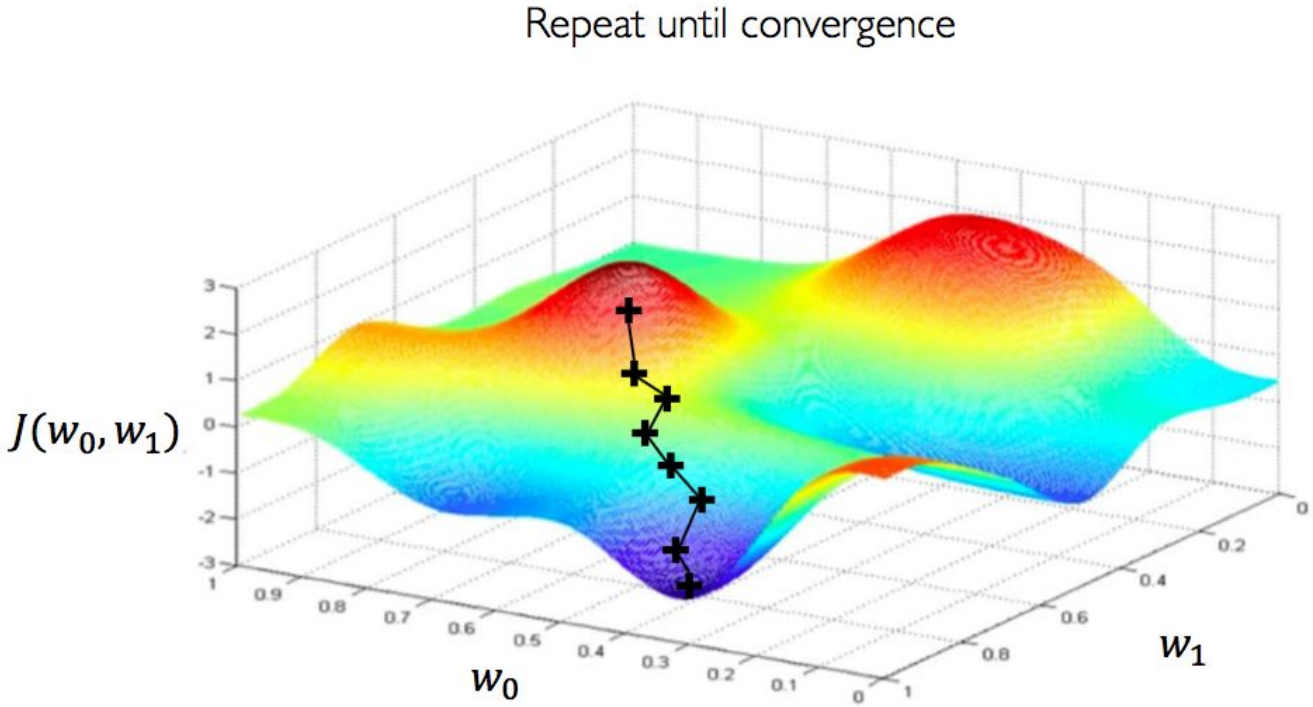
Loss Optimization

Find the network weights that achieve the lowest loss.



Loss Optimization

Find the network weights that achieve the lowest loss.



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, $\frac{\partial J(W)}{\partial W}$

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

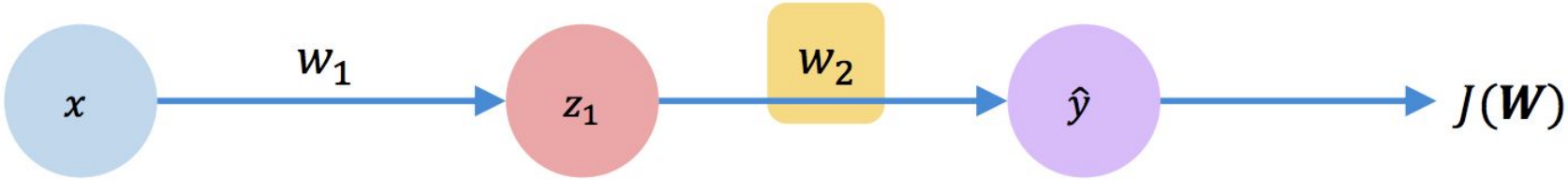
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

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

5. Return weights

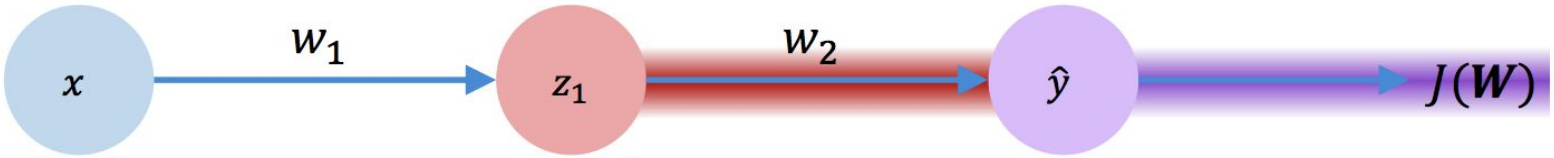
Training Neural Networks: Backpropagation

Computing Gradients: Backpropagation



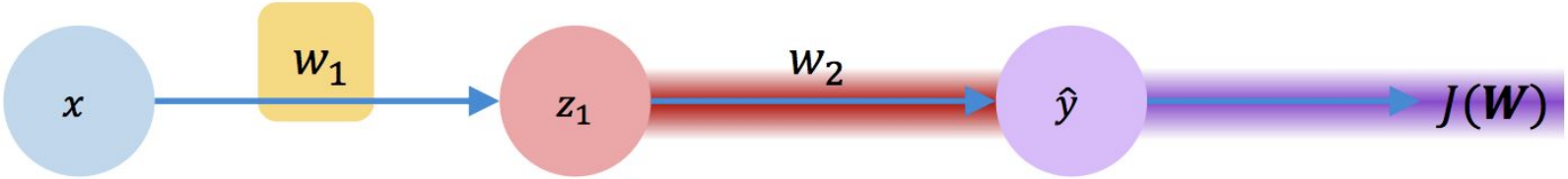
How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

Computing Gradients: Backpropagation

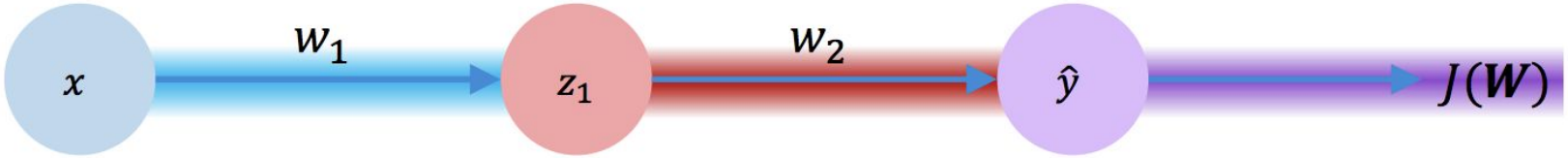


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradients: Backpropagation



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

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

Training Neural Networks: Adaptive learning

Loss Functions Can Be Difficult to Optimize

Algorithm

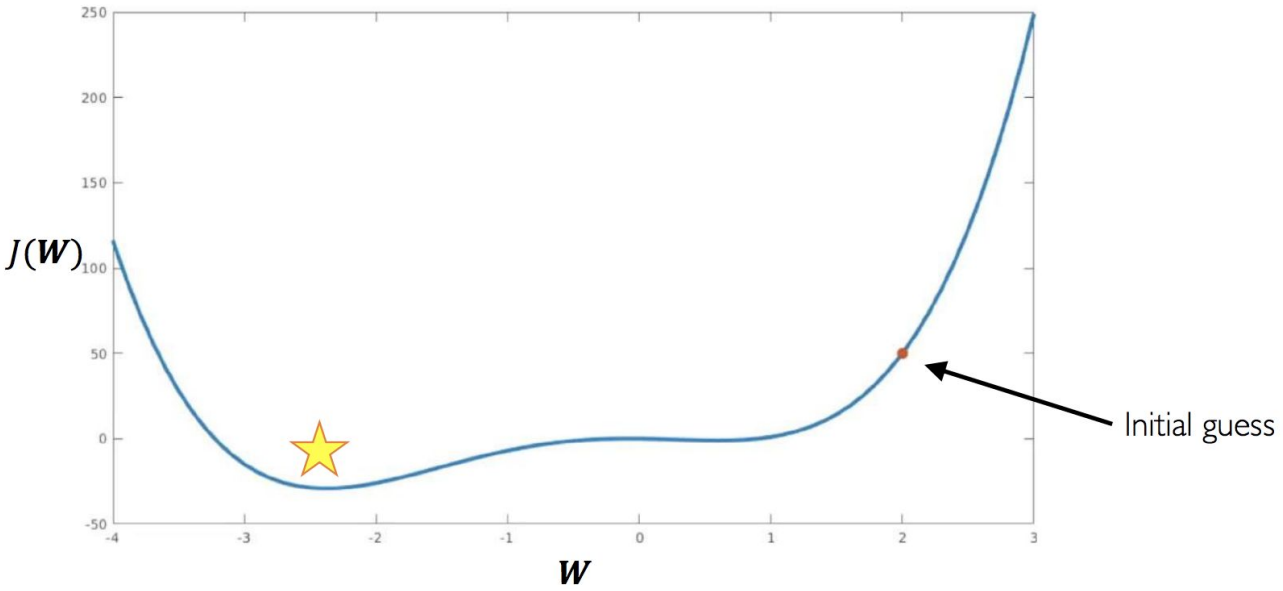
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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

Optimization through gradient descent

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

How can we set the learning rate?

Setting the Learning Rate



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

Large learning rates *overshoot, become unstable and diverge*

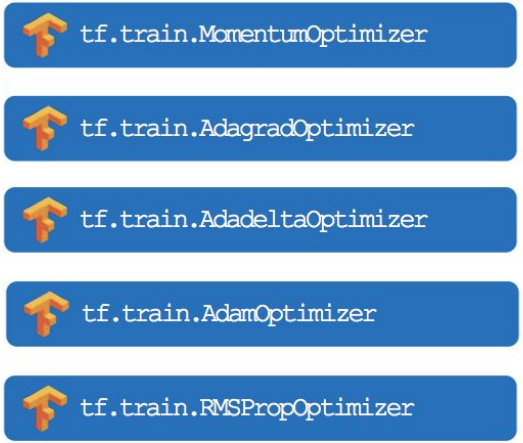
Stable learning rates *converge smoothly and avoid local minima*

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 gradient is
  - how fast learning is happening
  - size of particular weights
  - 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.

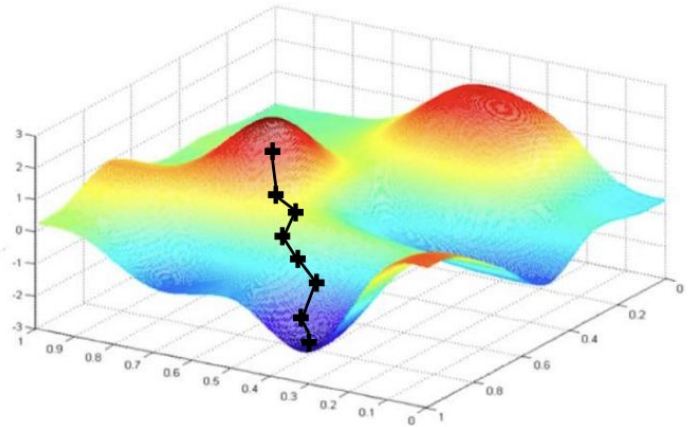
Additional details: http://ruder.io/optimizing-gradient-descent/

Training Neural Networks: Mini-batches

Batch 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

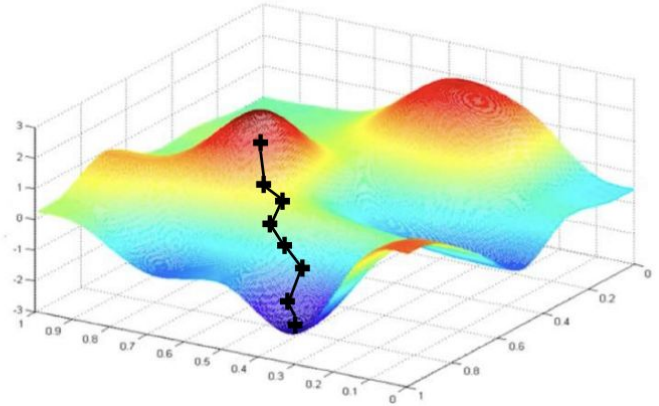


Batch Gradient Descent

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

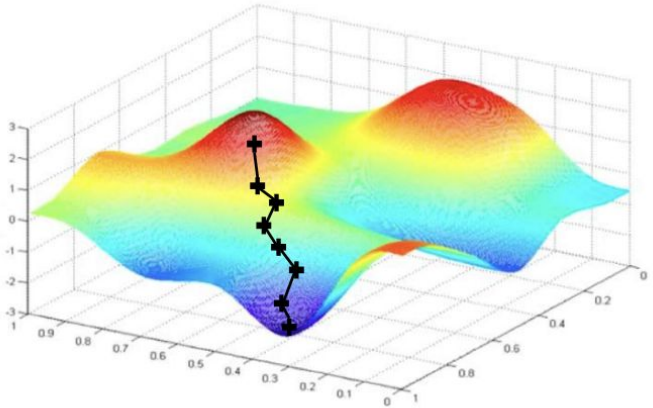
Can be very computational to compute!



Stochastic Gradient Descent

Algorithm

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

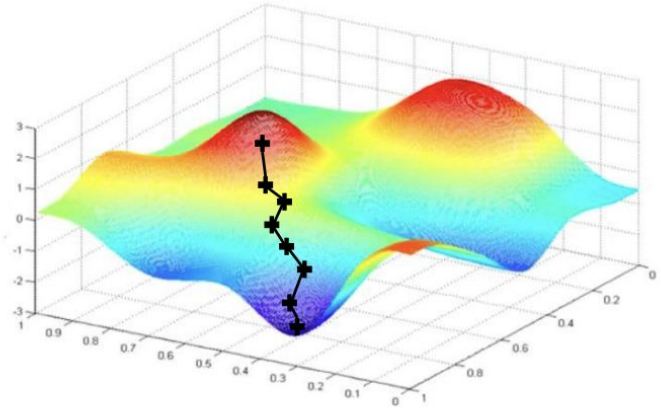


Stochastic Gradient Descent

Algorithm

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

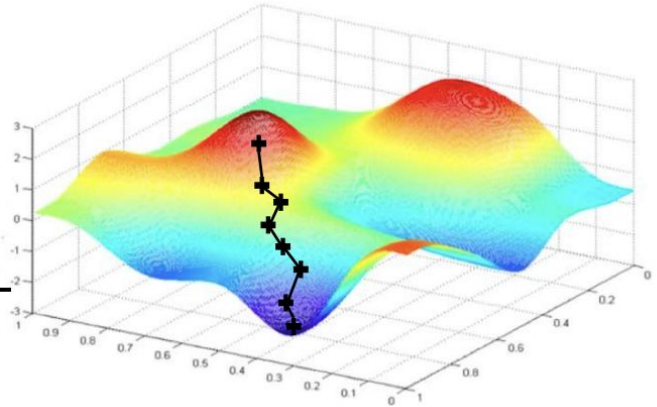
Easy to compute but **very noisy** (stochastic)!



Mini-Batch Gradient Descent

Algorithm

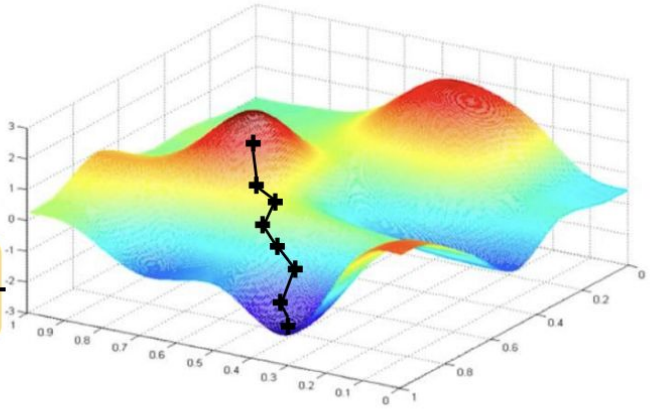
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial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



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!

Gradient Descent

Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

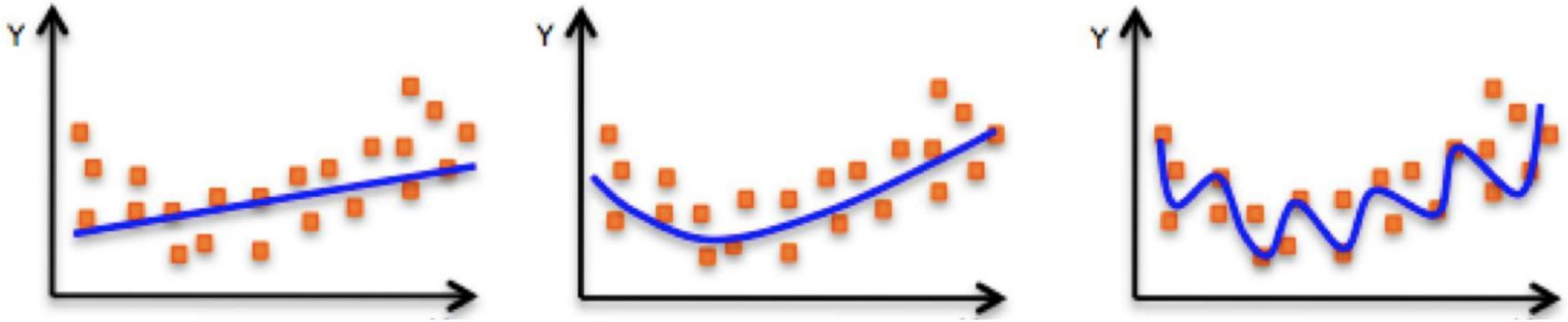
Mini-batches

- 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

Neural Networks: Overfitting

The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Regularization

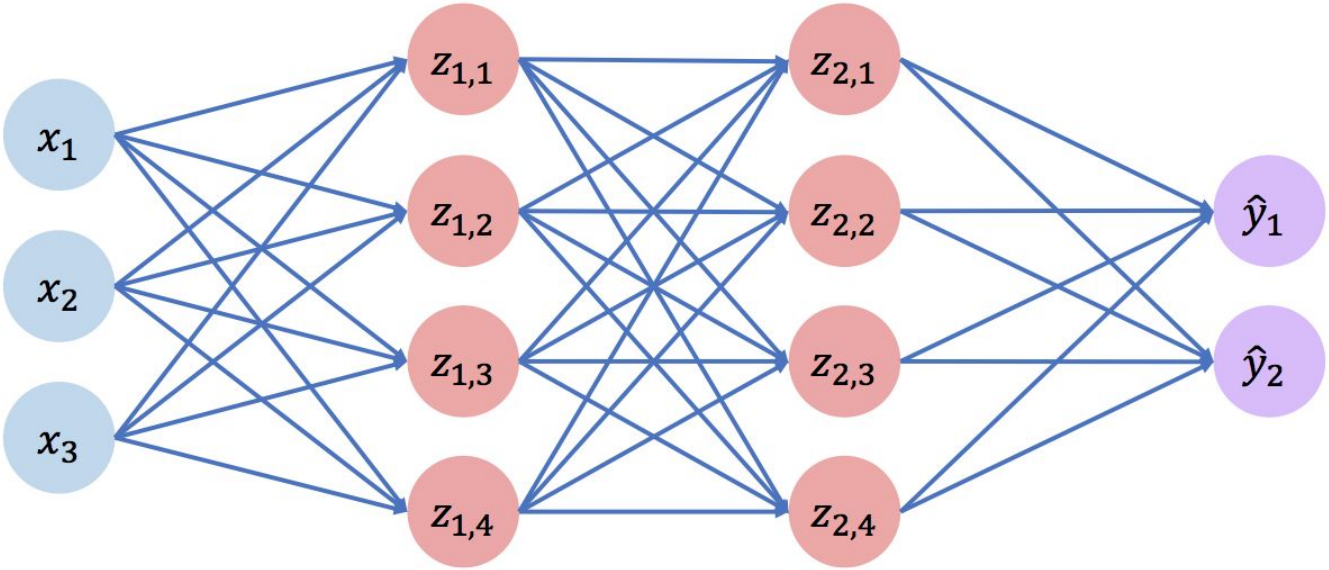
- Technique that constrains our optimization problem to discourage complex models.

- Improve generalization of our model on unseen data

- Common regularization techniques in NN
  - Dropout
  - Early stopping

Regularization: Dropout

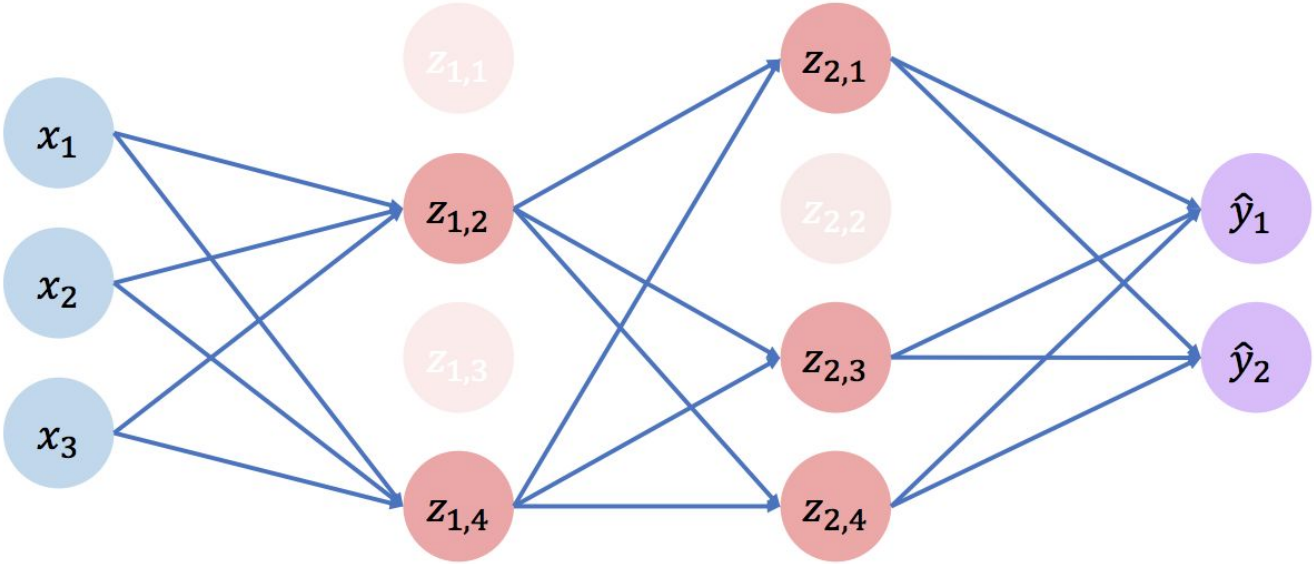
During training, randomly set some activations to 0



Regularization: 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

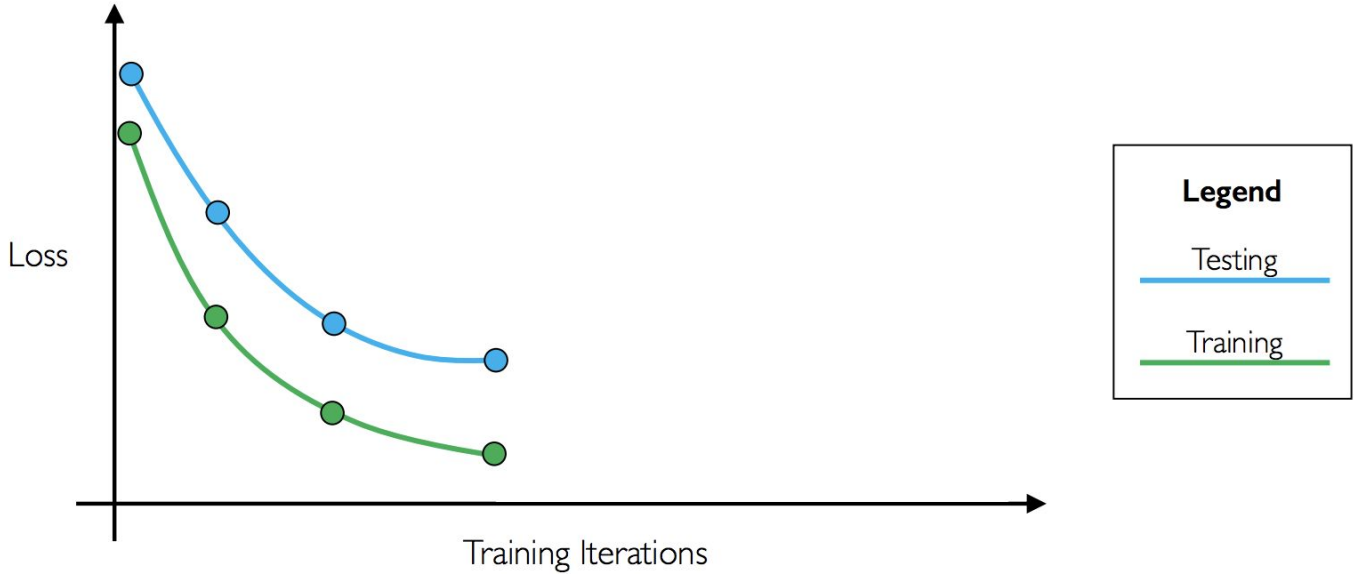


Regularization: Early Stopping

Stop training before we have a chance to overfit

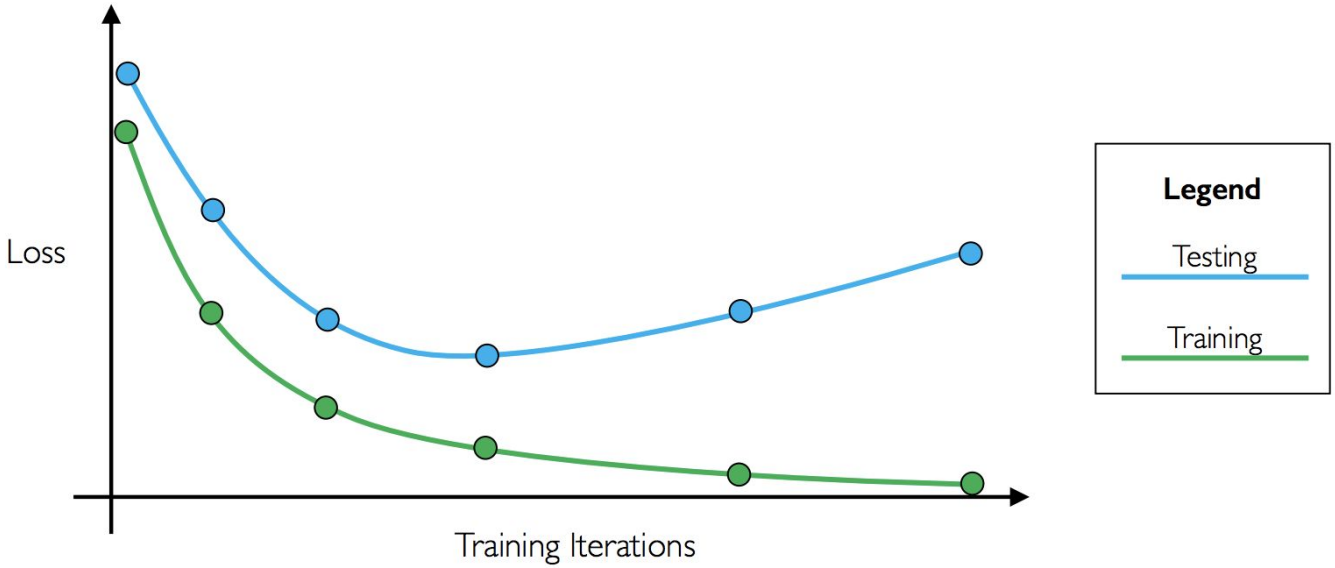
Regularization: Early Stopping

Stop training before we have a chance to overfit



Regularization: Early Stopping

Stop training before we have a chance to overfit



Regularization: Early Stopping

Stop training before we have a chance to overfit

Build NN: Last-layer Activation

- Last-layer Activation - This establishes useful constraints on the network's output.
  - The IMDB classification example used sigmoid in the last layer.
  - The regression example didn't use any last-layer activation.

Build NN: Loss Function

- Loss function - This should match the type of problem you're trying to solve.
  - The IMDB example used binary_crossentropy.
  - The regression example used MSE.

Build NN: Optimization Configuration

- Optimization configuration
  - What optimizer will you use?
  - What will its learning rate be?
  - In most cases, it’s safe to go with rmsprop and its default learning rate.

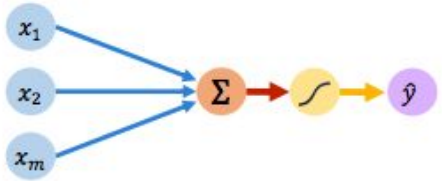
Last-layer Activation and Loss Function

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

Lecture Recap

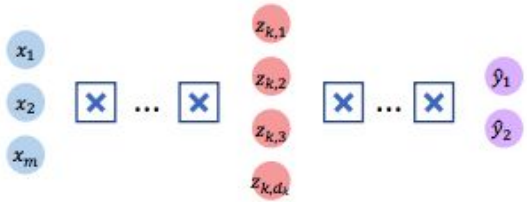
The Perceptron

- Structural building blocks
- Nonlinear activation functions



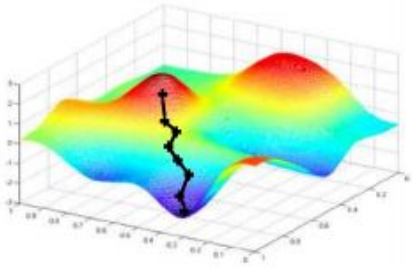
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Lab