11-11-2019

Artificial neural networks

Robert L. Peach

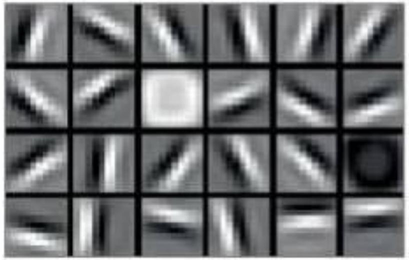
- Why deep learning?
- **Multi-layer perceptrons (MLP)**
  - The Perceptron
  - Activation functions
  - Forward propagation
  - Loss functions
  - Back propagation
  - Regularisation in ANNs
- **Convolutional neural networks (CNNs)**
  - Image classification / object identification
  - Convolutions
  - Pooling
- **Recurrent neural networks (RNNs)**
  - Forecasting
  - Increasing memory – LSTMs, Gated RNNs

Deep Learning

Hand engineered features are time consuming, brittle and not scalable in practice
Can we learn the **underlying features** directly from the data?

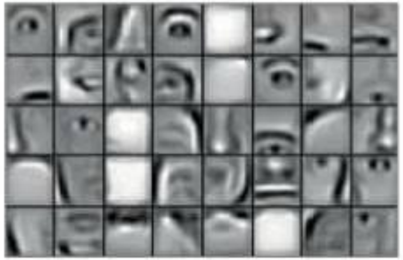
Millions of images

Low level features



Lines & Edges

Mid level features

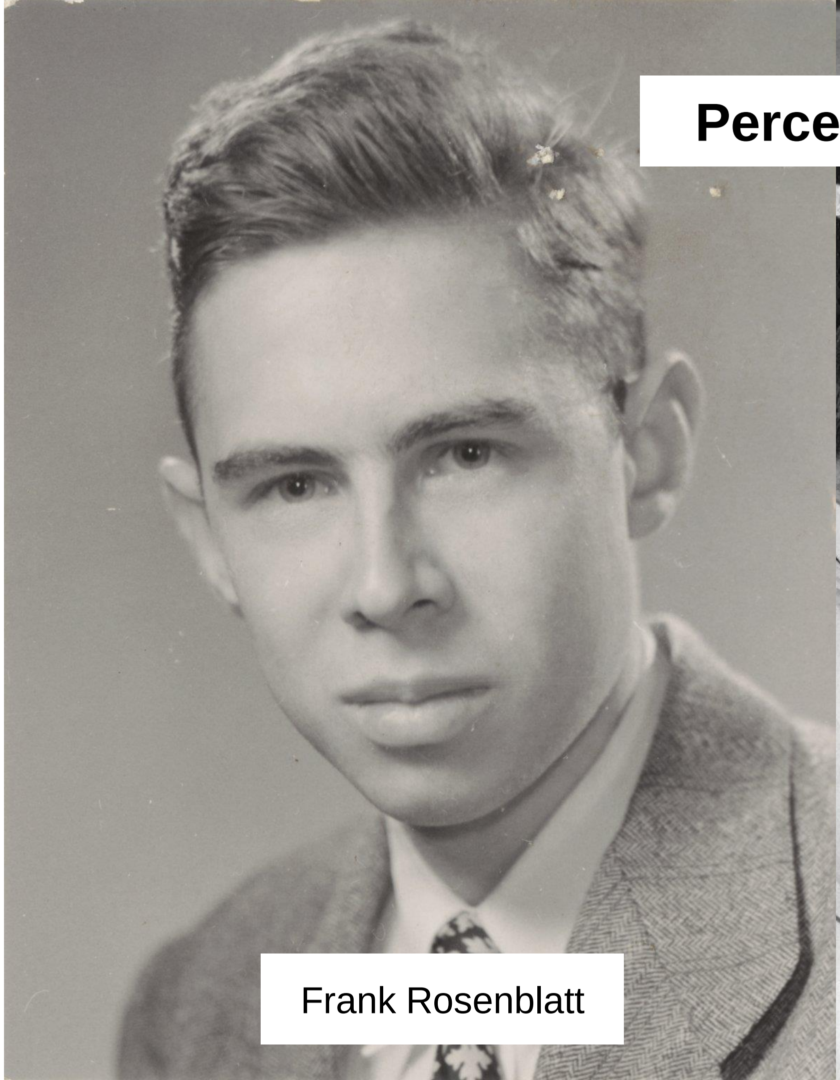


Eyes & Nose & Ears

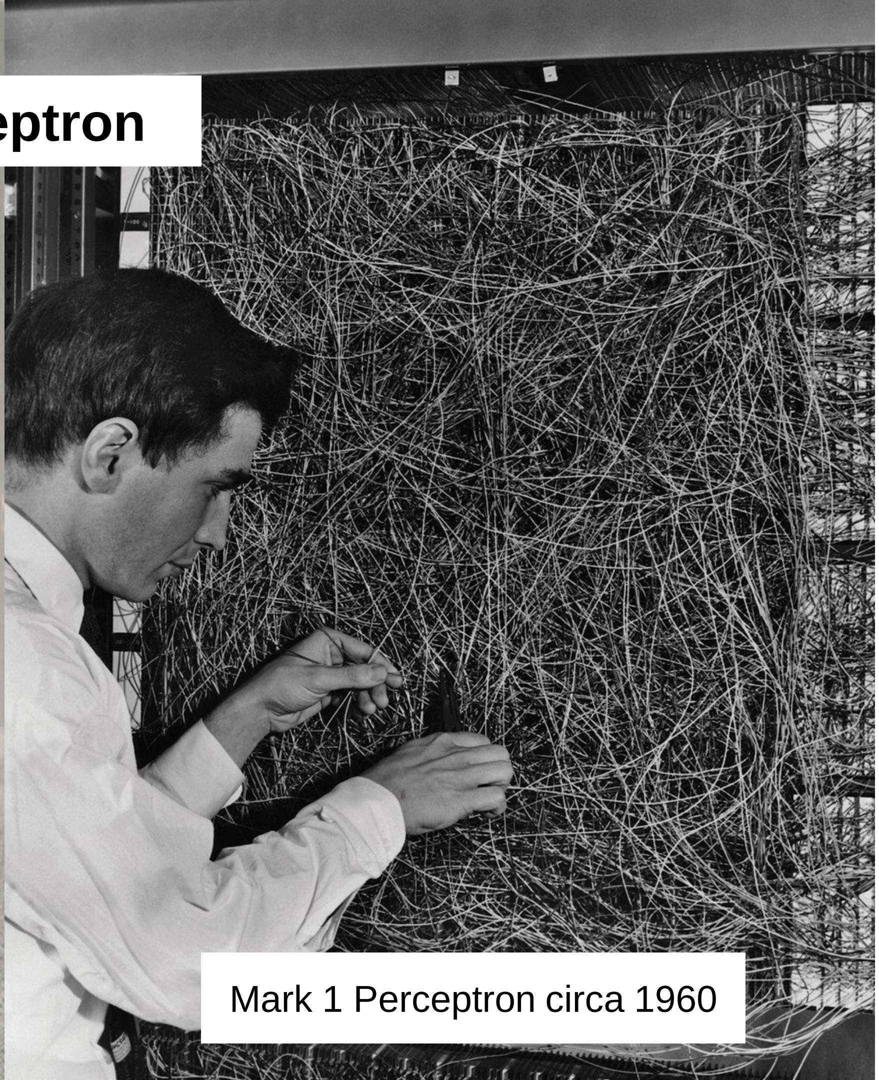
High level features

Facial Structure

Perceptron

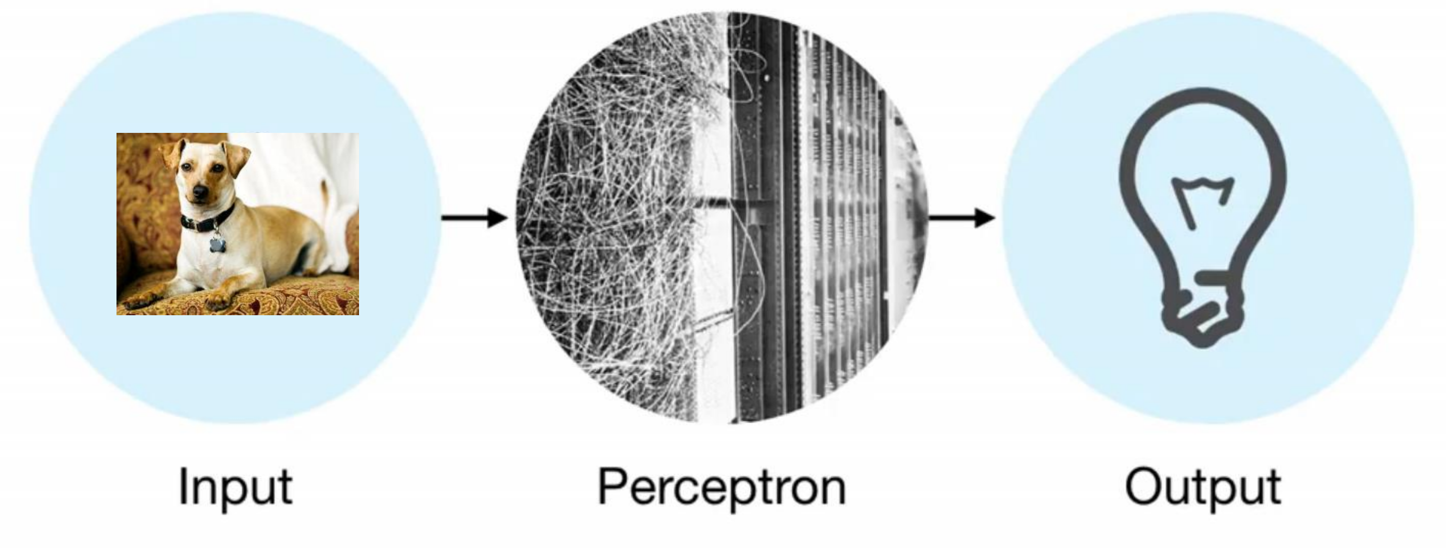


Frank Rosenblatt

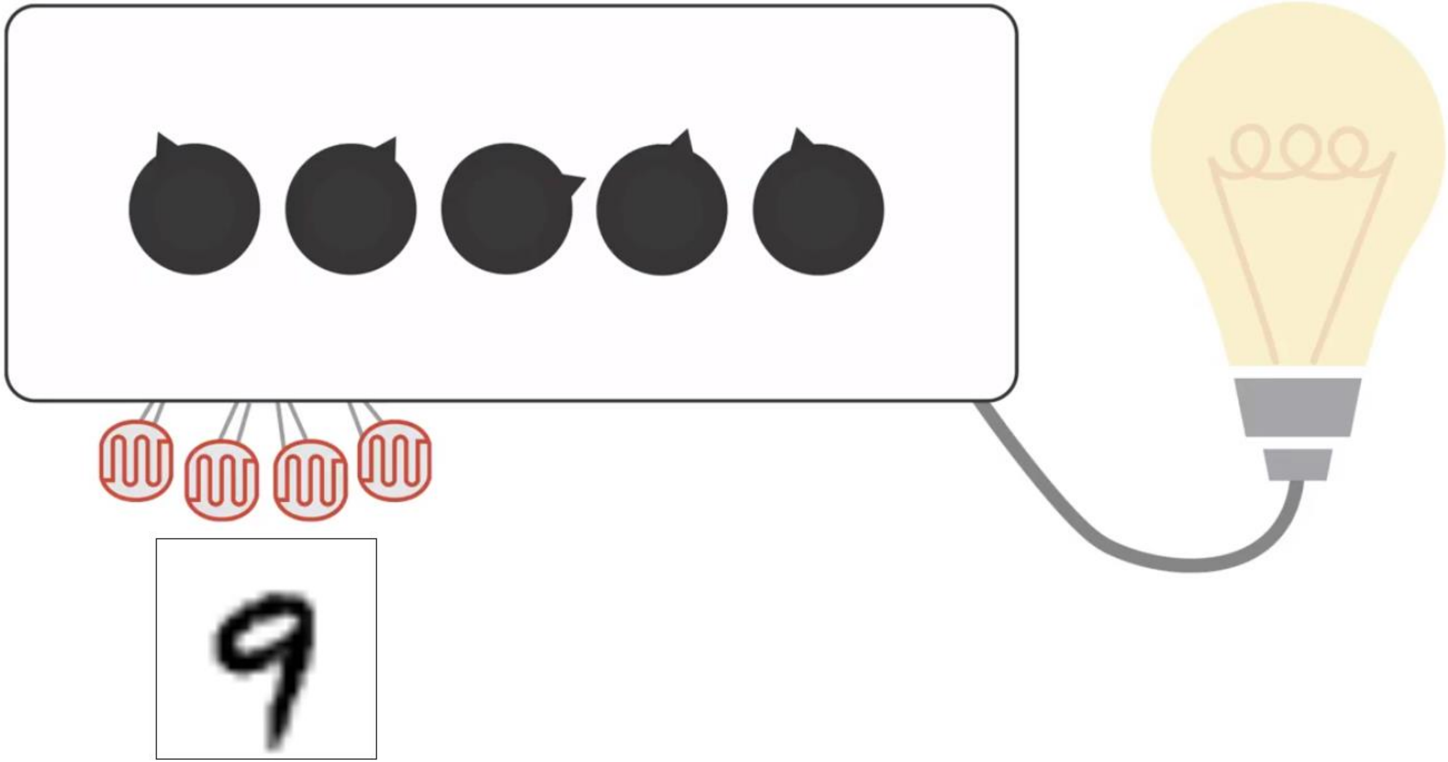


Mark 1 Perceptron circa 1960

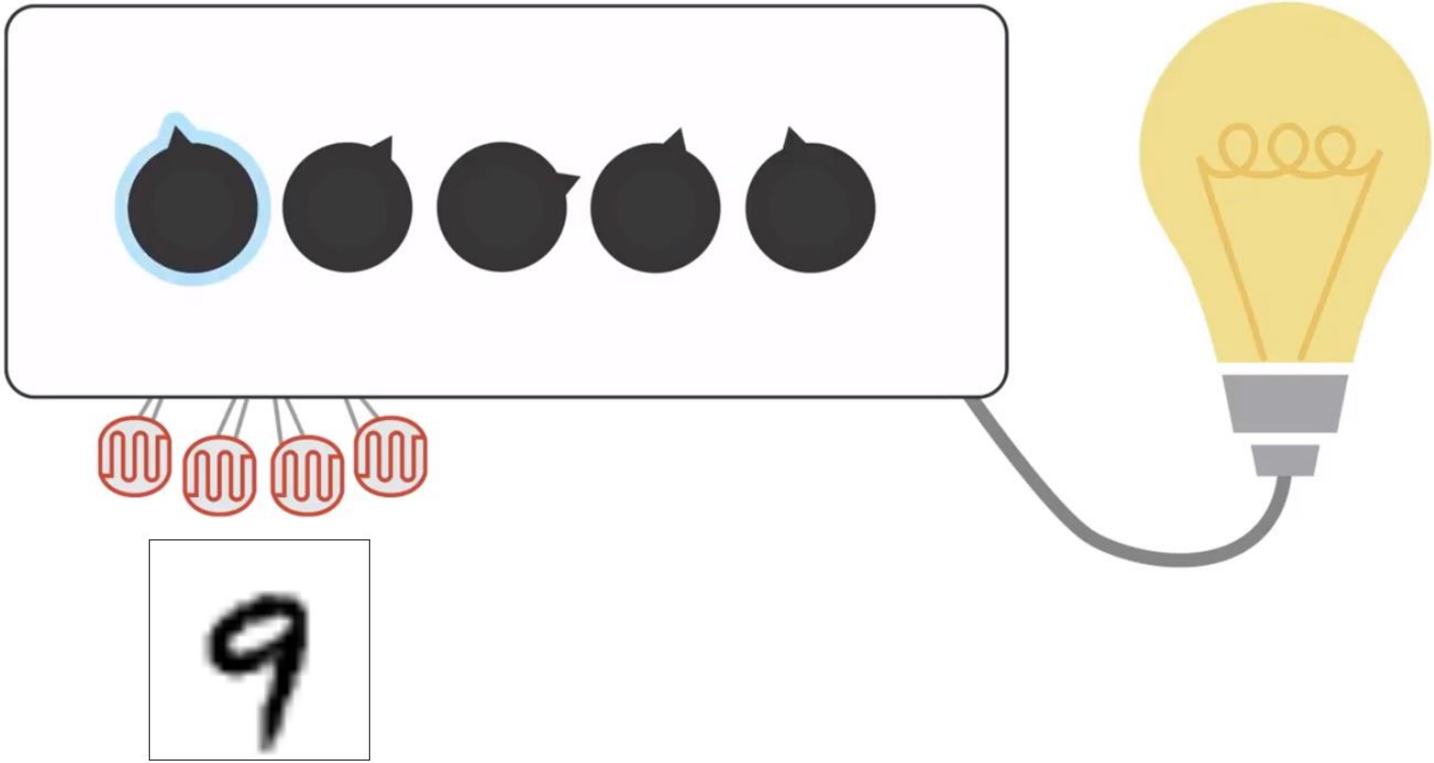
Rosenblatt’s Machine



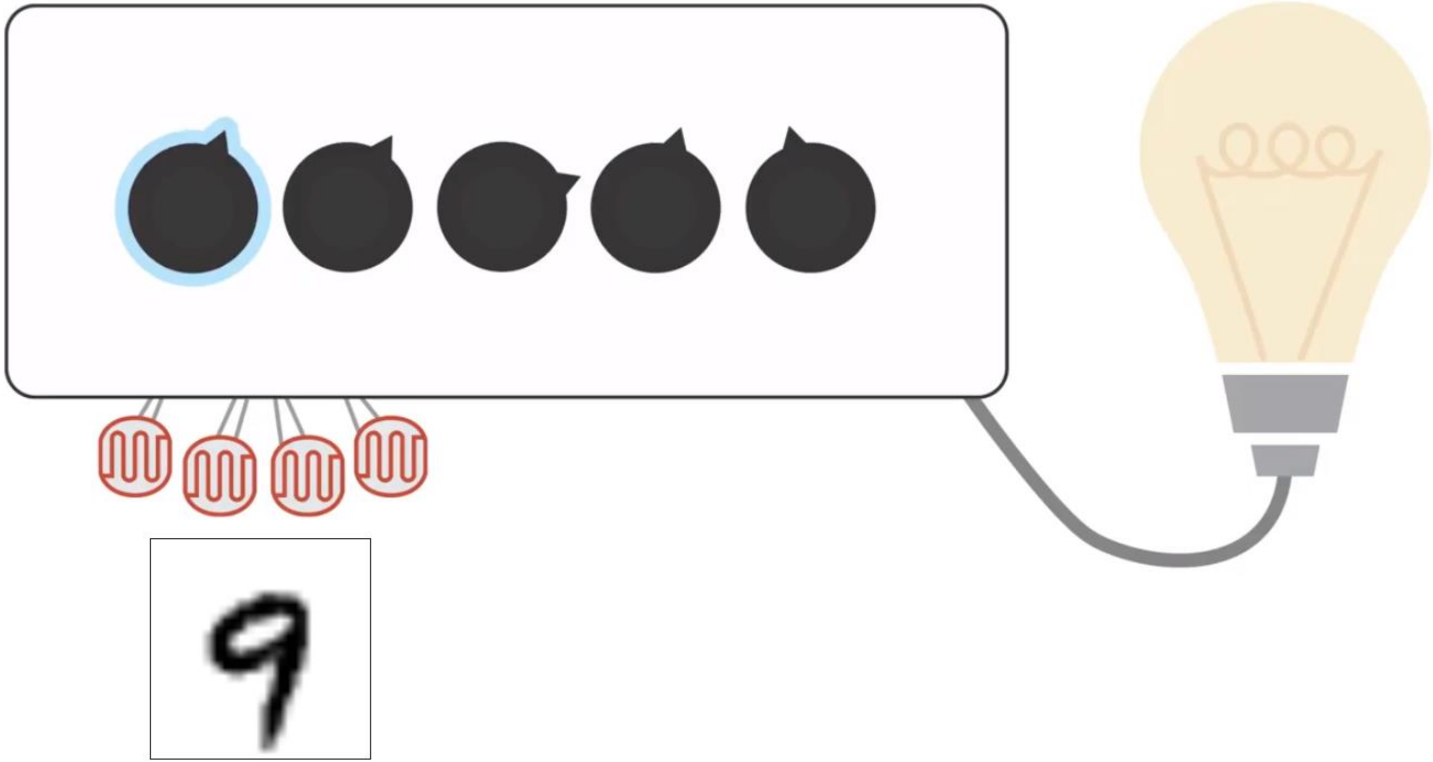
Rosenblatt’s Machine



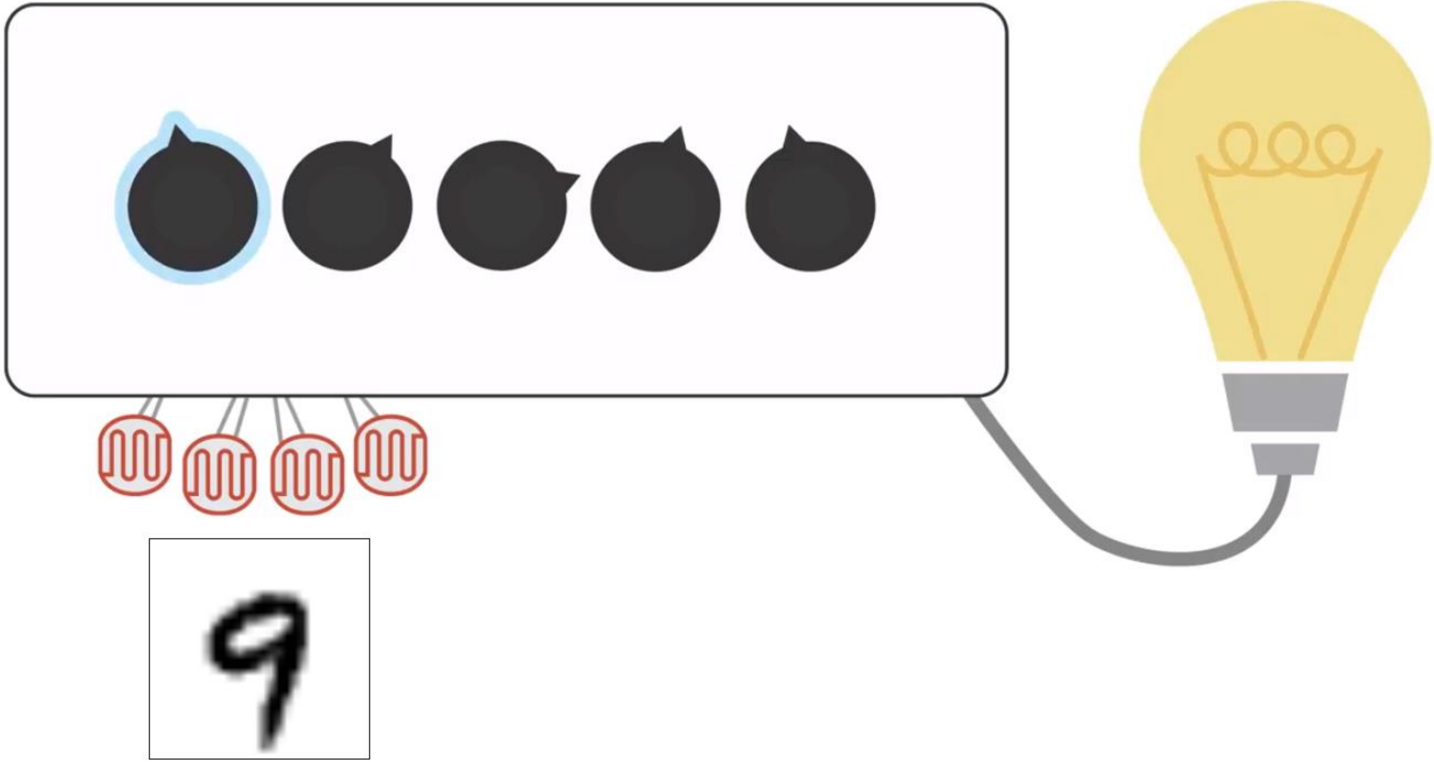
Rosenblatt's Machine



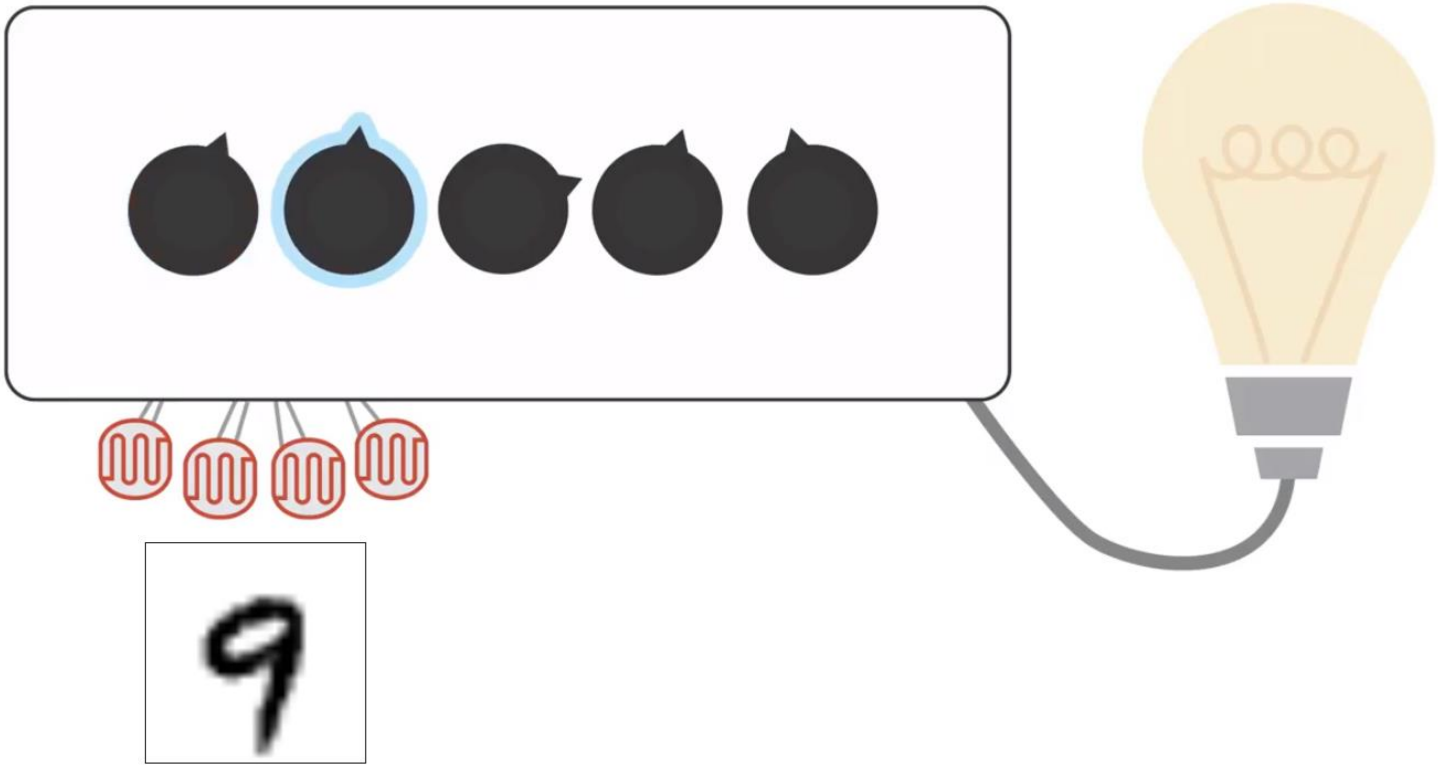
Rosenblatt’s Machine



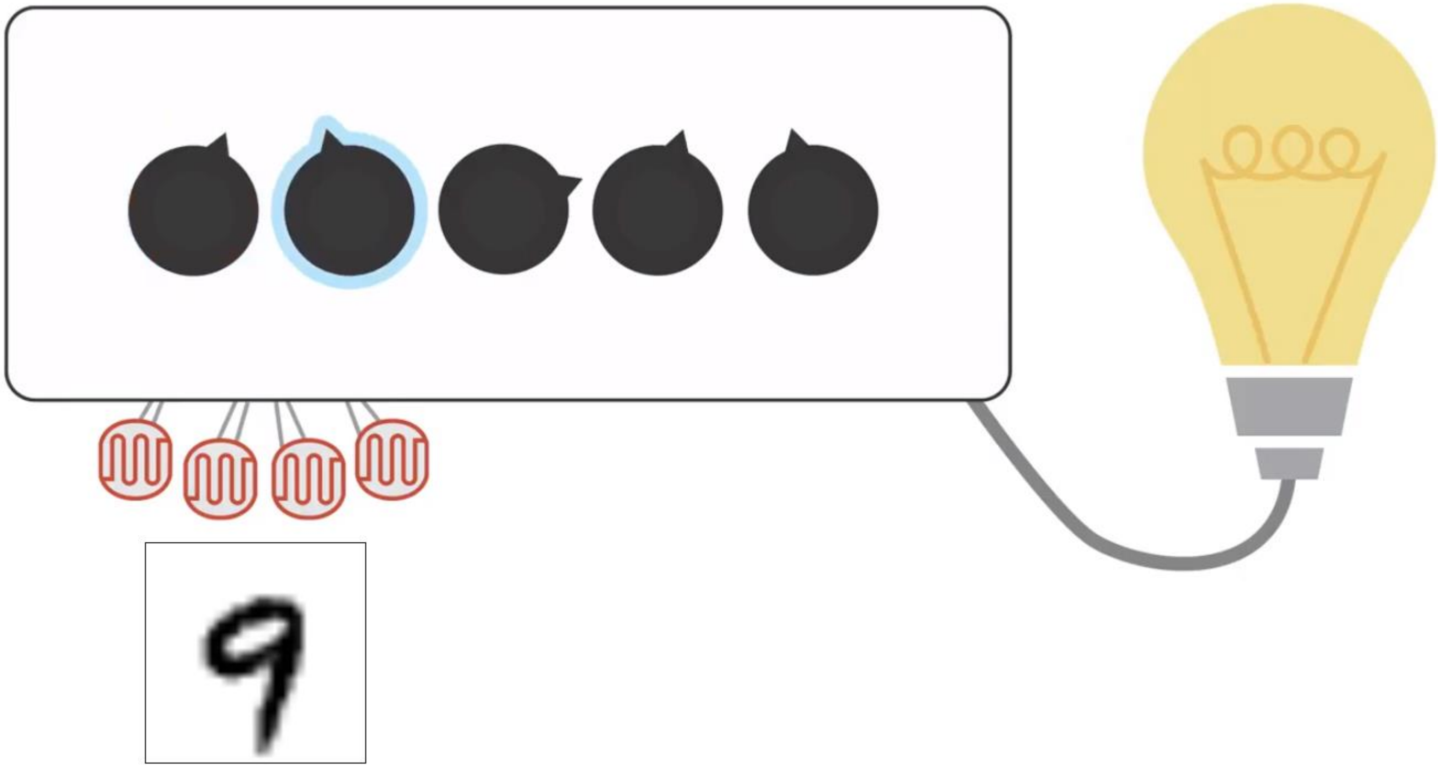
Rosenblatt’s Machine



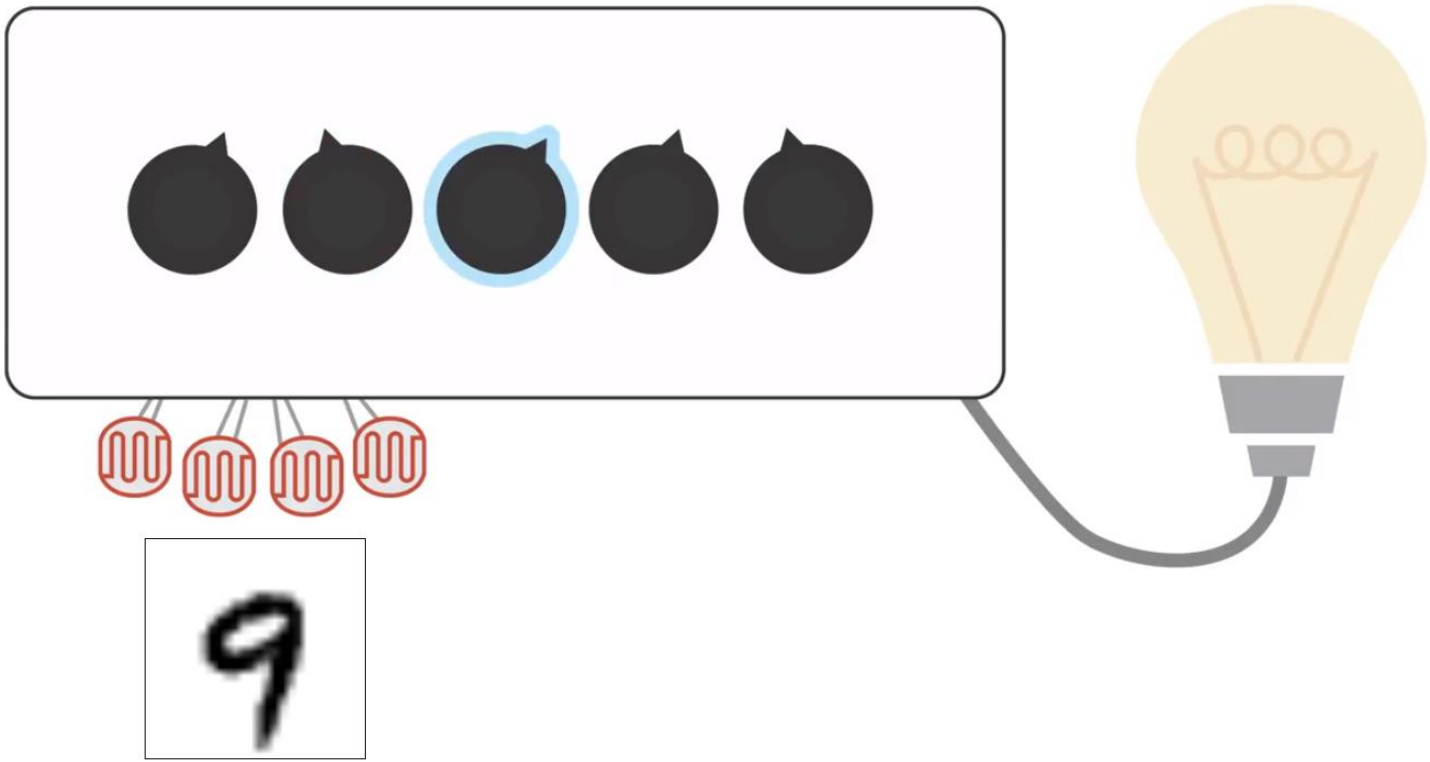
Rosenblatt’s Machine



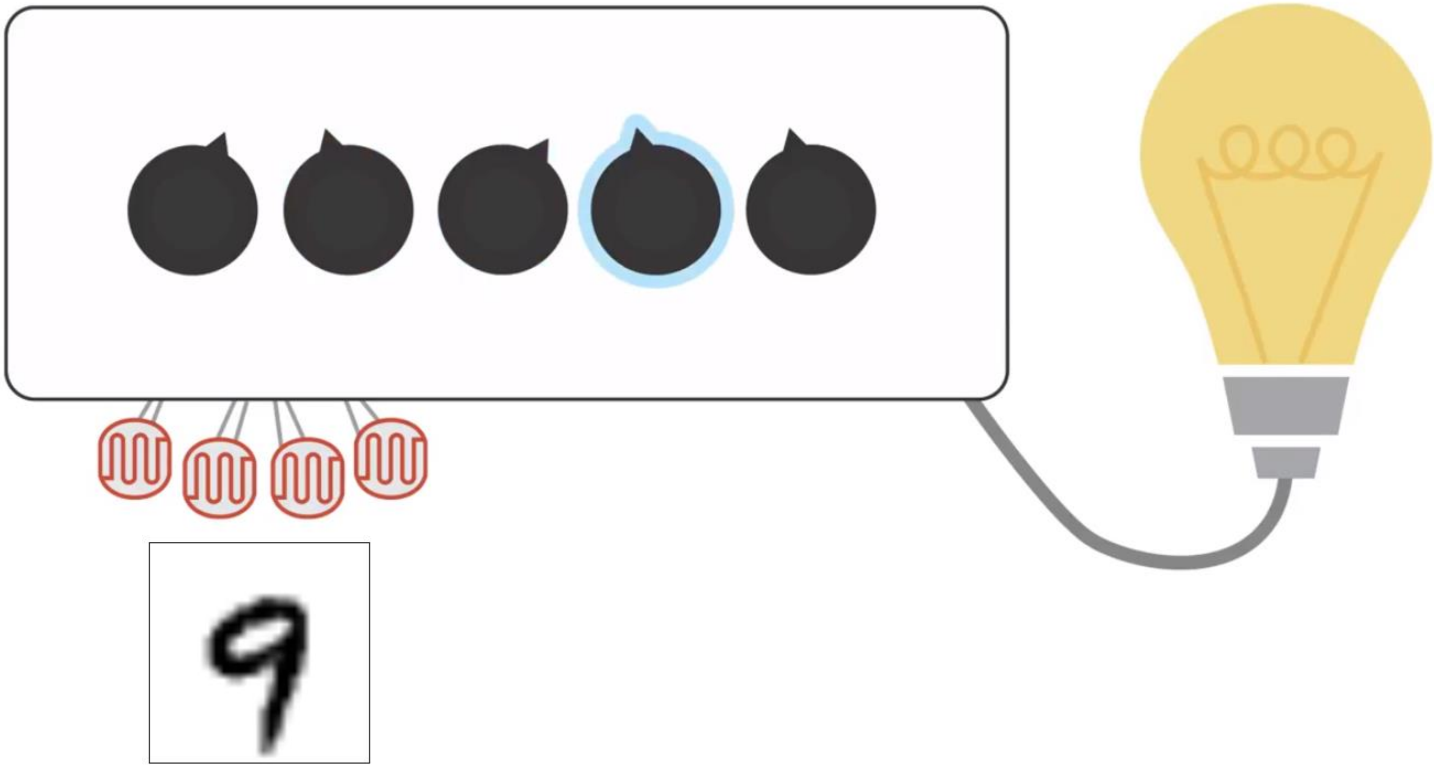
Rosenblatt's Machine



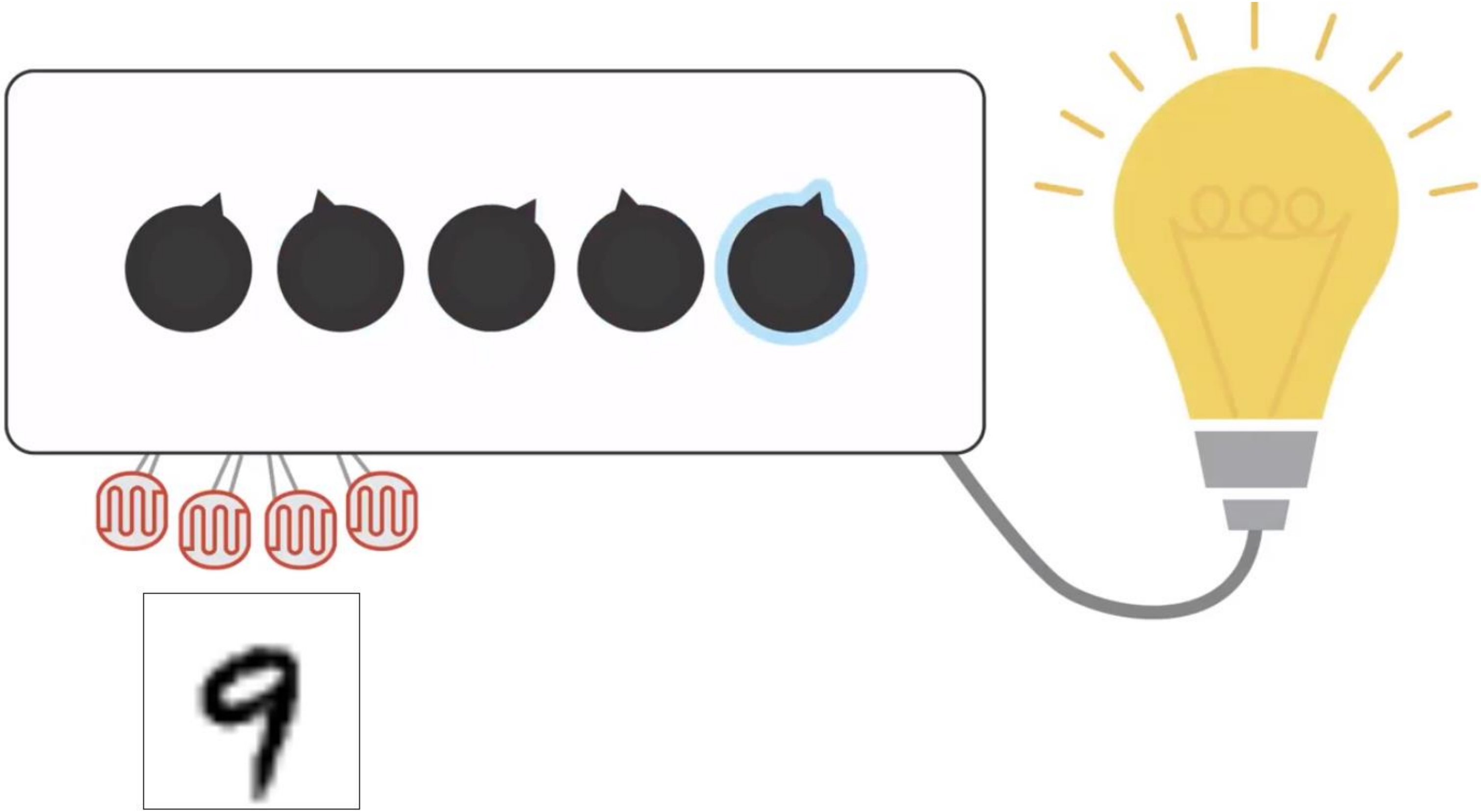
Rosenblatt’s Machine



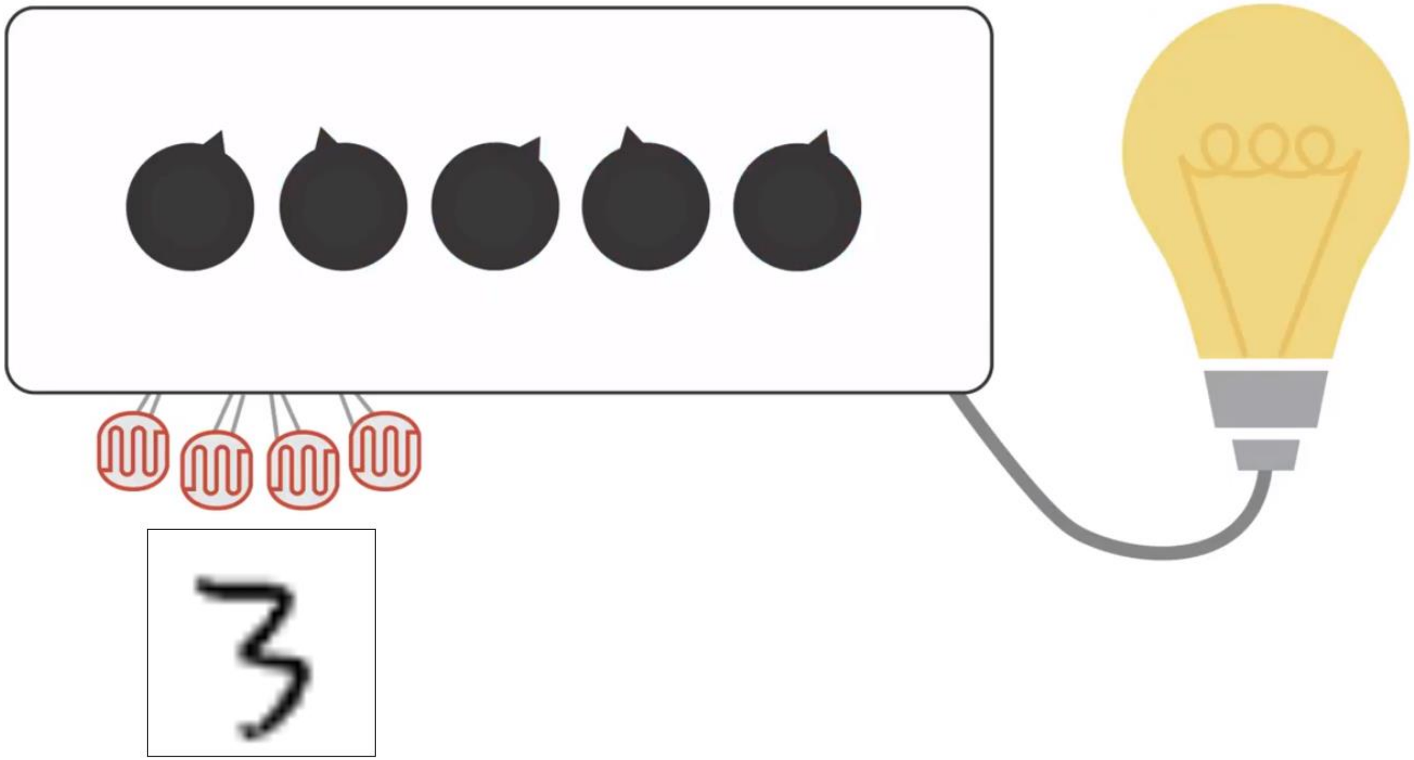
Rosenblatt’s Machine



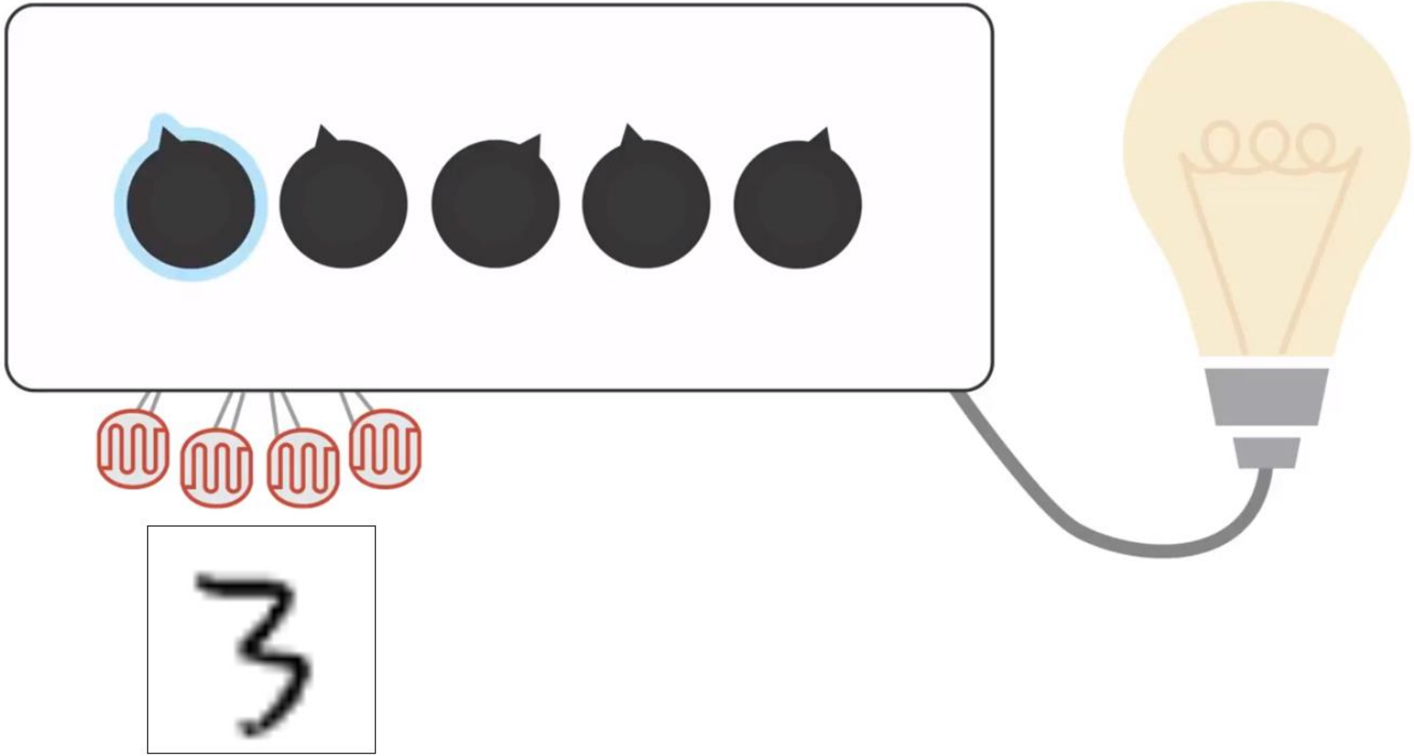
Rosenblatt’s Machine



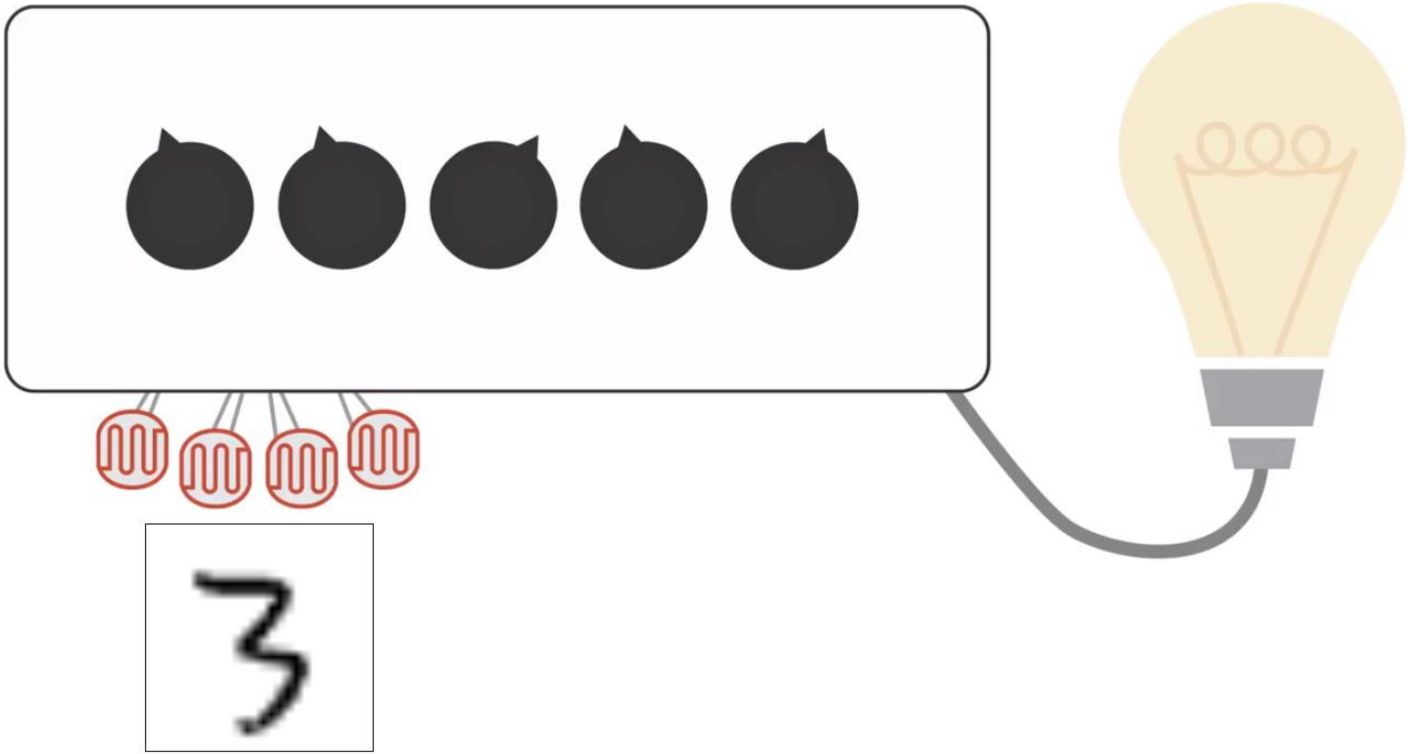
Rosenblatt's Machine



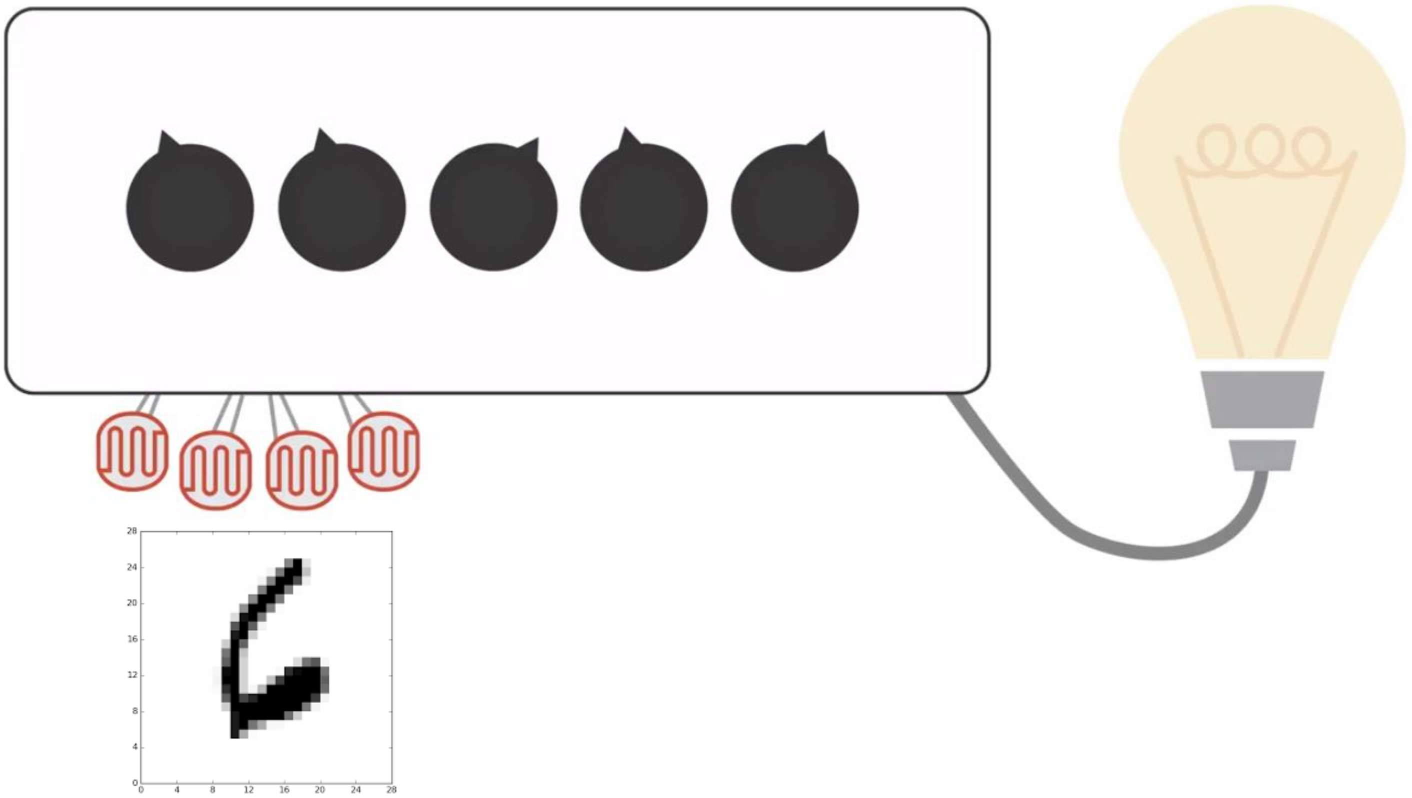
Rosenblatt’s Machine



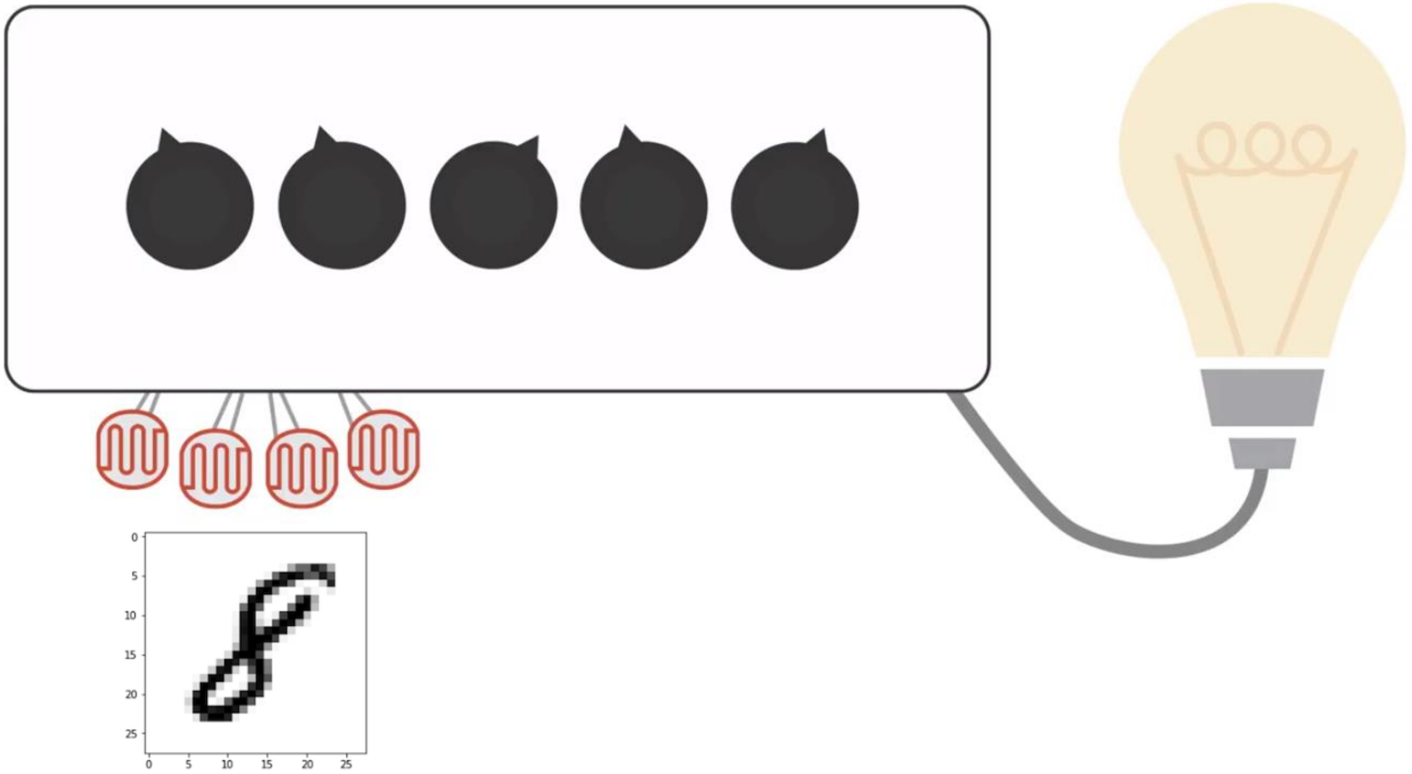
Rosenblatt's Machine



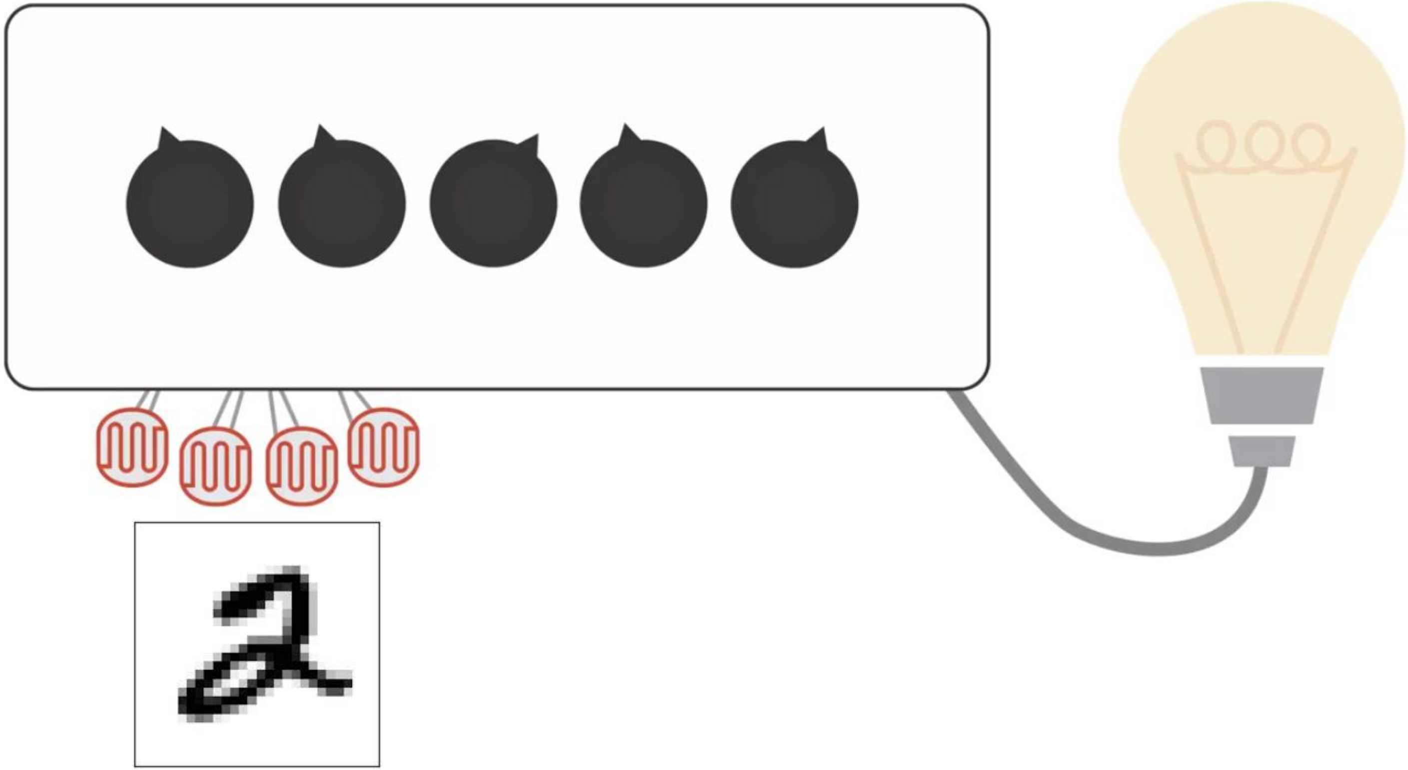
Rosenblatt’s Machine



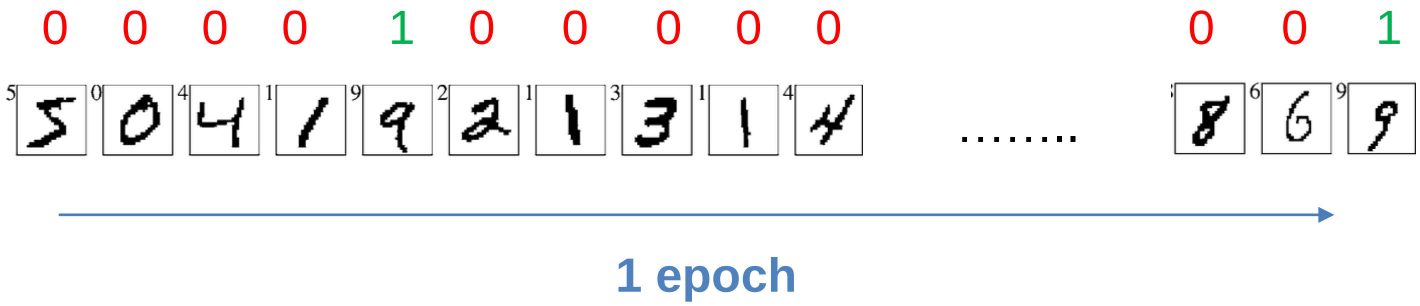
Rosenblatt’s Machine



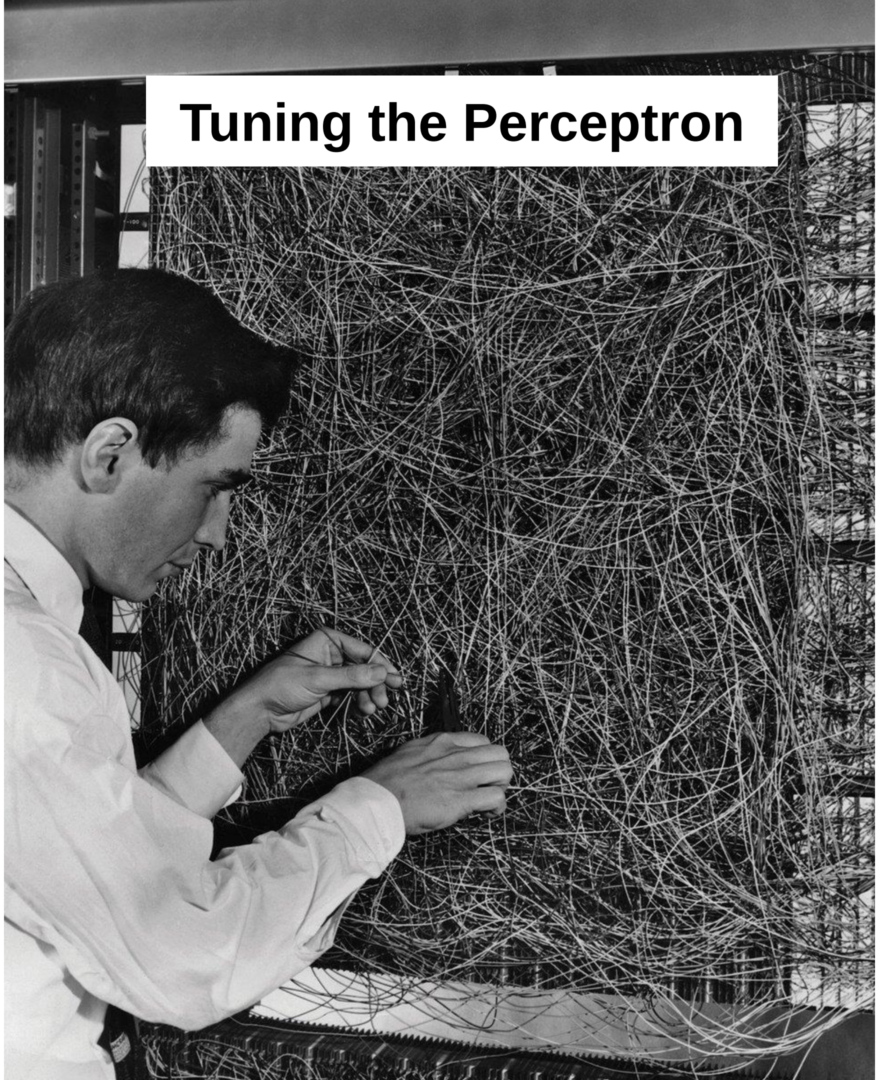
Rosenblatt’s Machine



Training Data



Tuning the Perceptron



We are thinking less about machines and more about algorithms…

The **perceptron** is the fundamental building block of deep learning.

Also known as a **neuron** in deep learning.

Why now?

Neural networks have existed for decades. Why do we care now?

Big data

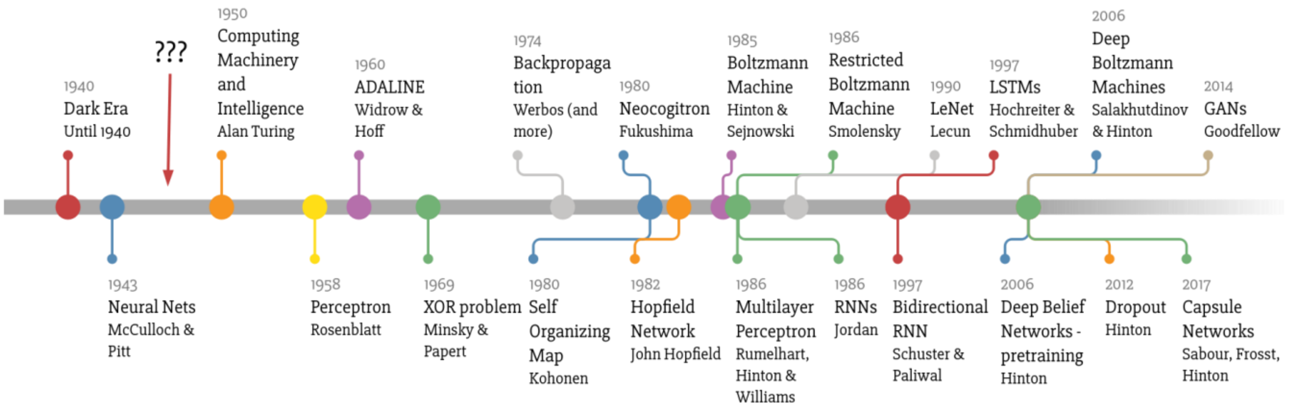
- Large datasets
- Easier collection and storage
- More platforms for collecting data

Hardware

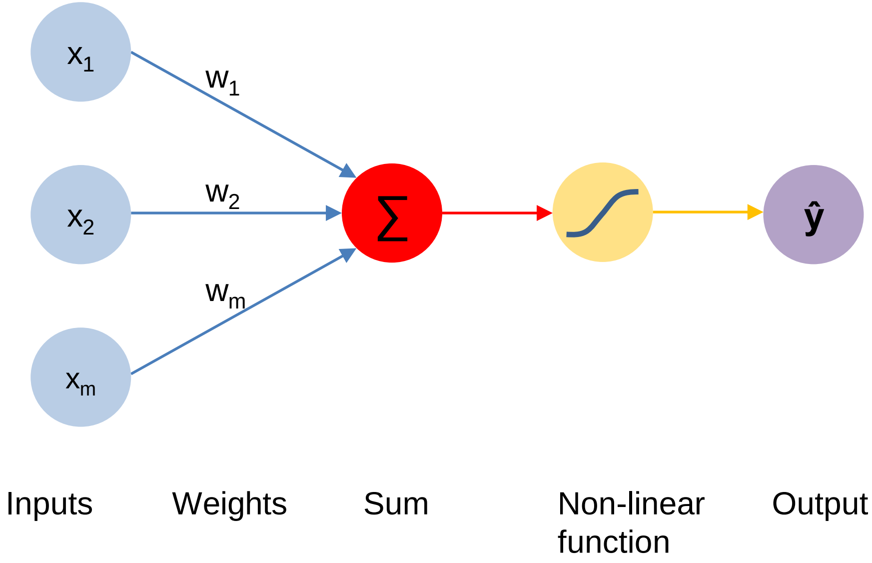
- Graphics processing units (GPUs)
- Massively parallelizable calculations

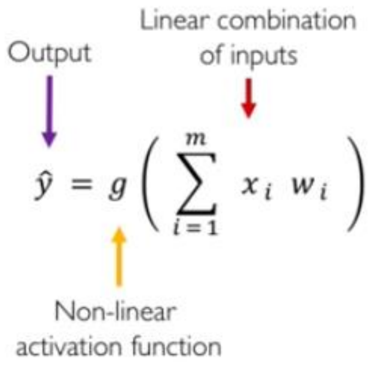
Software

- Improved mathematical architectures
- Efficient and open source toolboxes

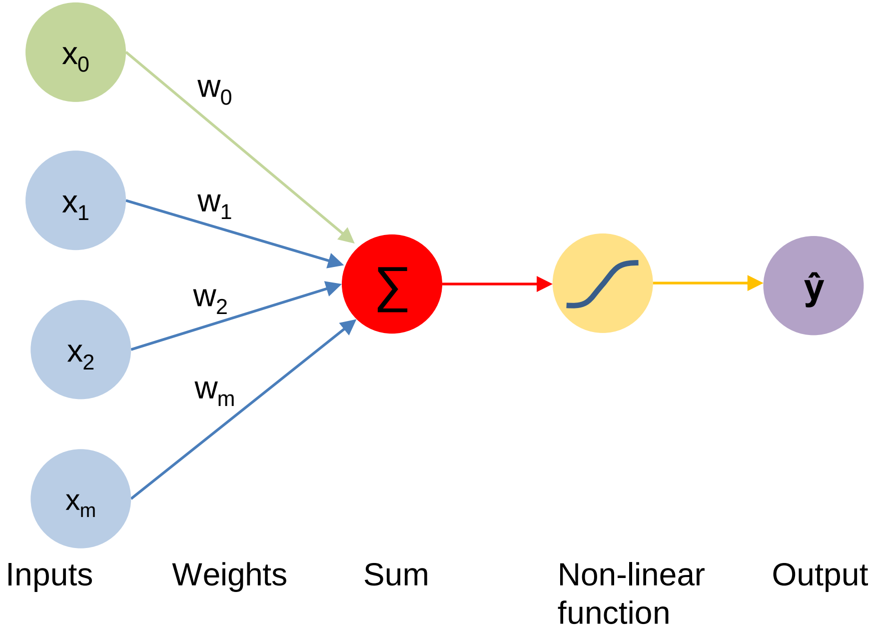


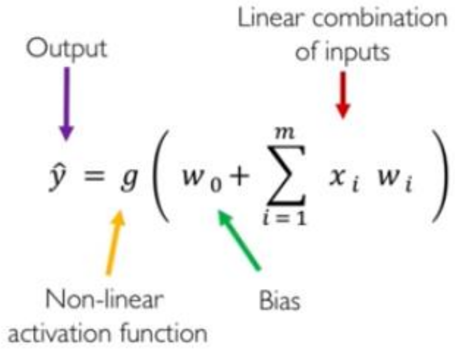
The perceptron: Forward propagation



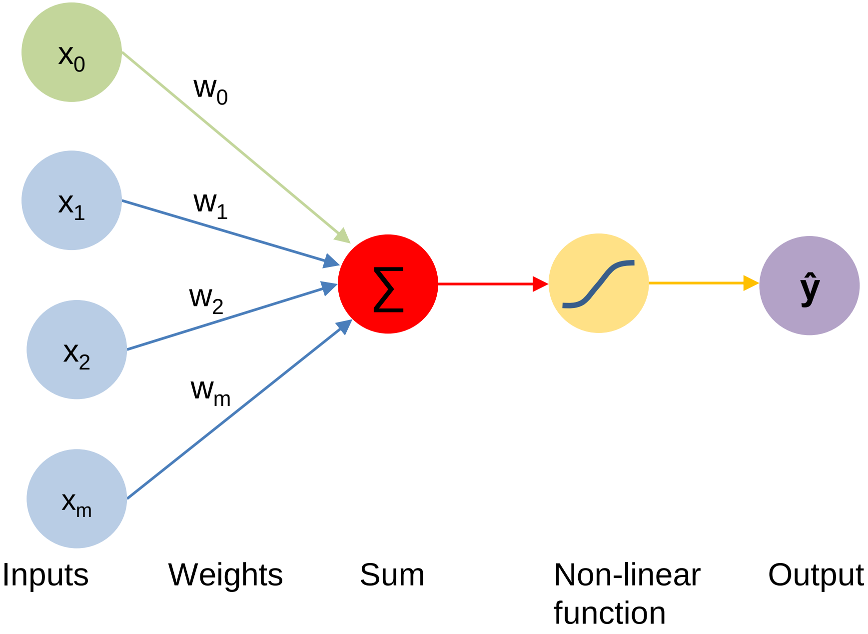


The perceptron: Forward propagation





The perceptron: Forward propagation

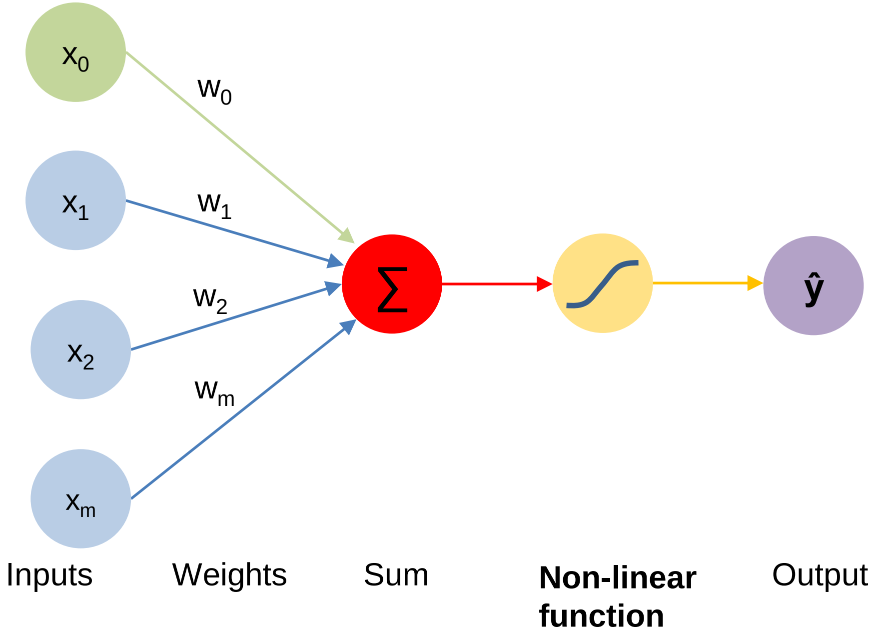


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

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

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

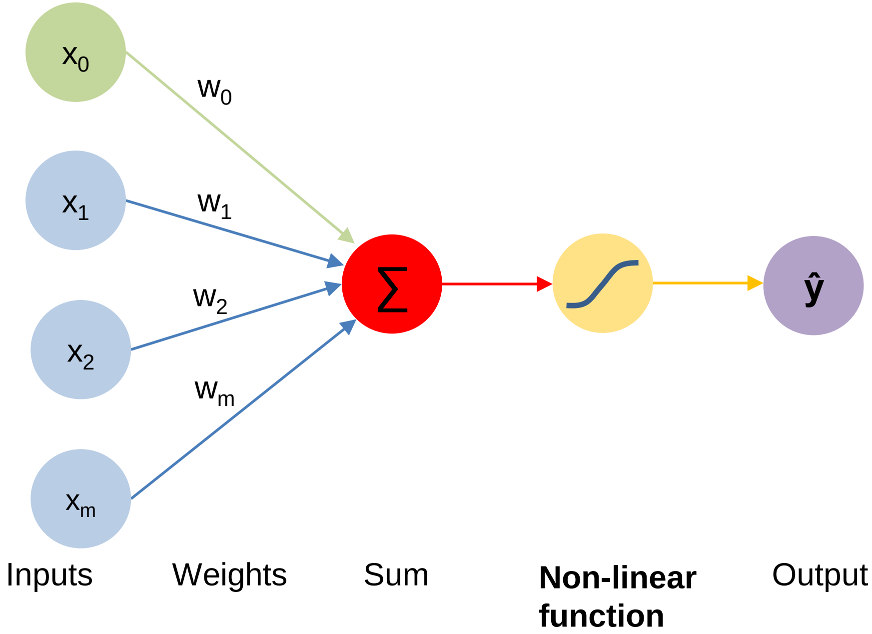
The perceptron: Forward propagation



Activation Functions

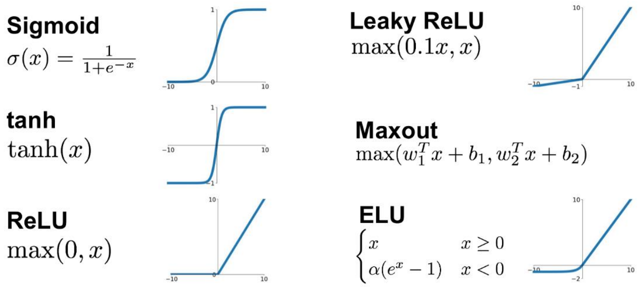
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

The perceptron: Forward propagation

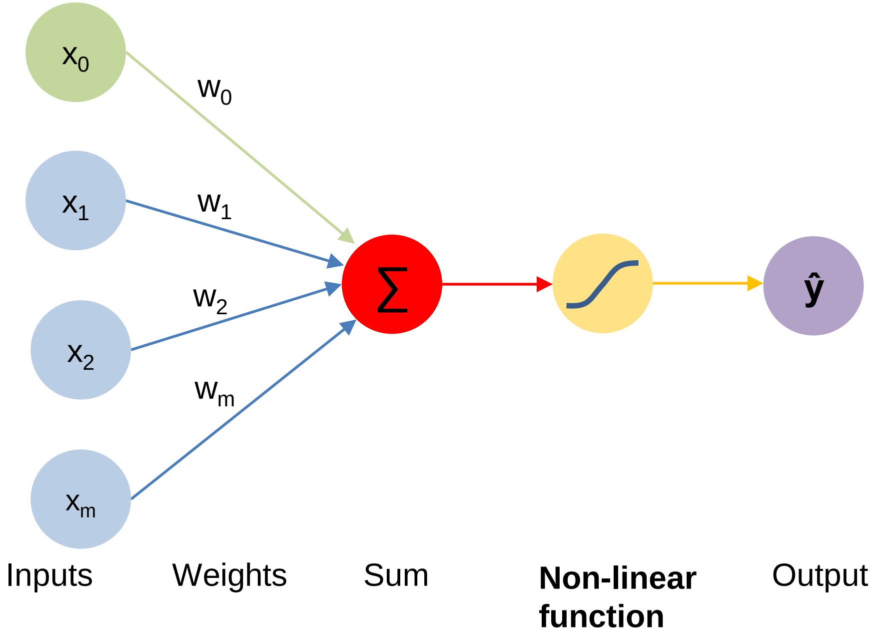


Activation Functions

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



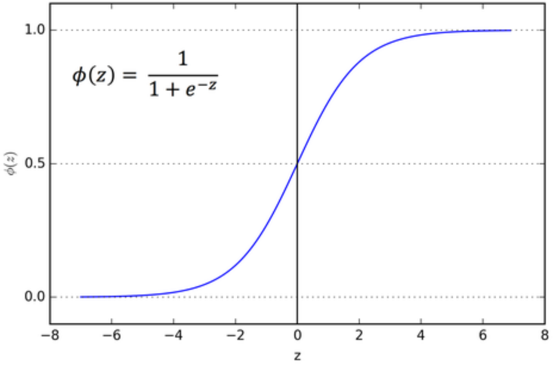
The perceptron: Forward propagation



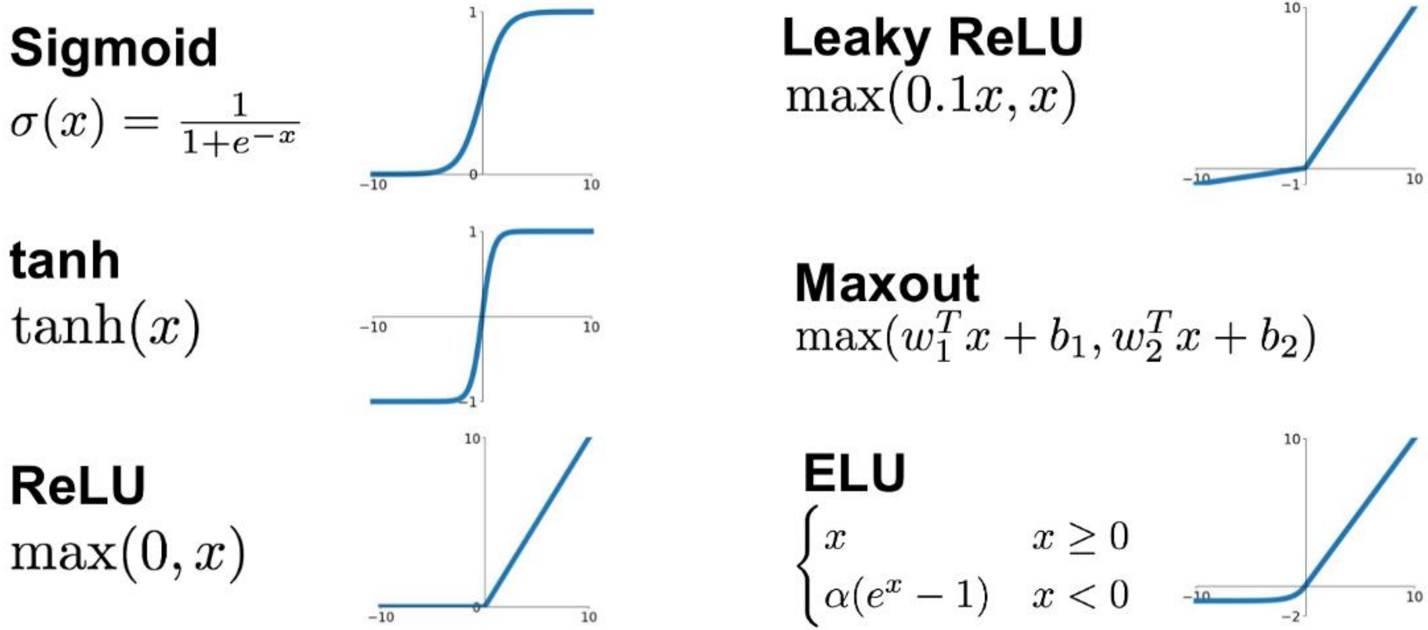
Activation Functions

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

Example: Logistic function



The perceptron: Forward propagation

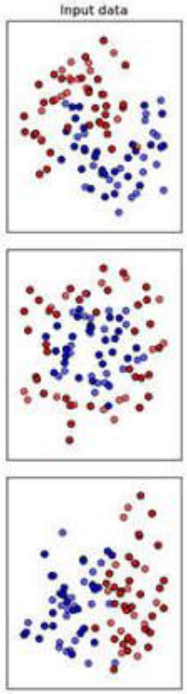


Why use activation functions?

Activation functions introduce ***non-linearities*** into the network

Linear activation functions produce linear decisions.

Non-linearities allow us to approximate arbitrarily complex functions

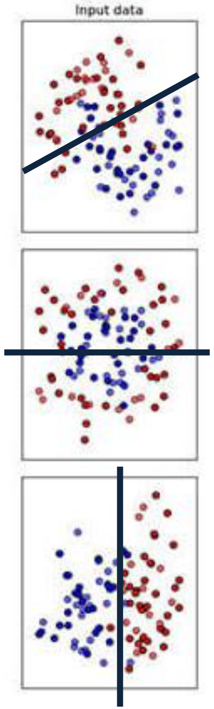


Why use activation functions?

Activation functions introduce ***non-linearities*** into the network

Linear activation functions produce linear decisions.

Non-linearities allow us to approximate arbitrarily complex functions

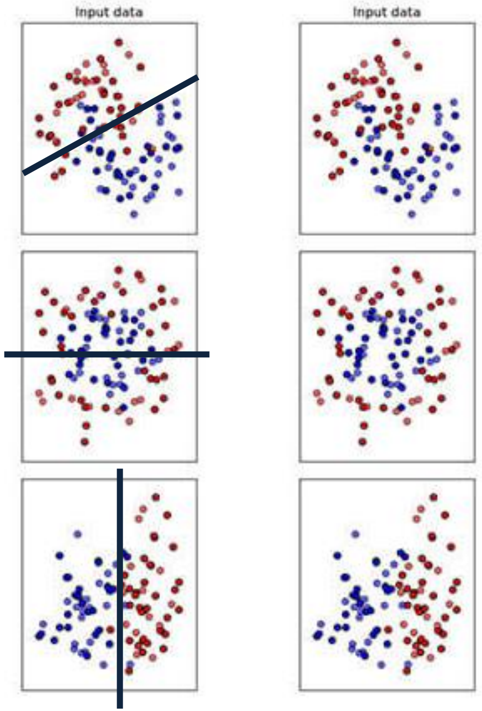


Why use activation functions?

Activation functions introduce ***non-linearities*** into the network

Linear activation functions produce linear decisions.

Non-linearities allow us to approximate arbitrarily complex functions

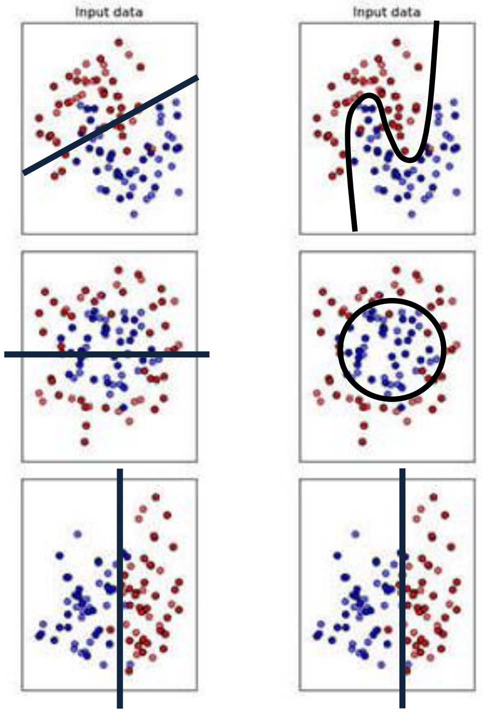


Why use activation functions?

Activation functions introduce ***non-linearities*** into the network

Linear activation functions produce linear decisions.

Non-linearities allow us to approximate arbitrarily complex functions

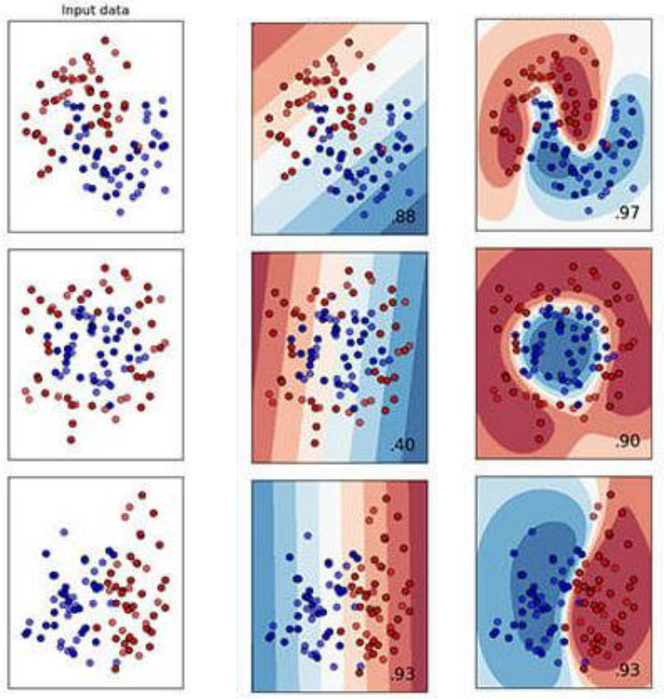


Why use activation functions?

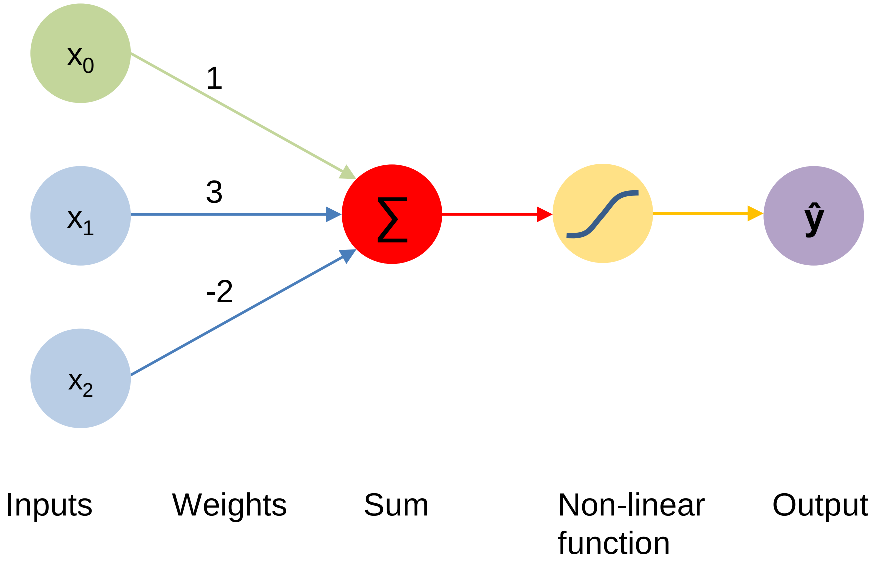
Activation functions introduce ***non-linearities*** into the network

Linear activation functions produce linear decisions.

Non-linearities allow us to approximate arbitrarily complex functions



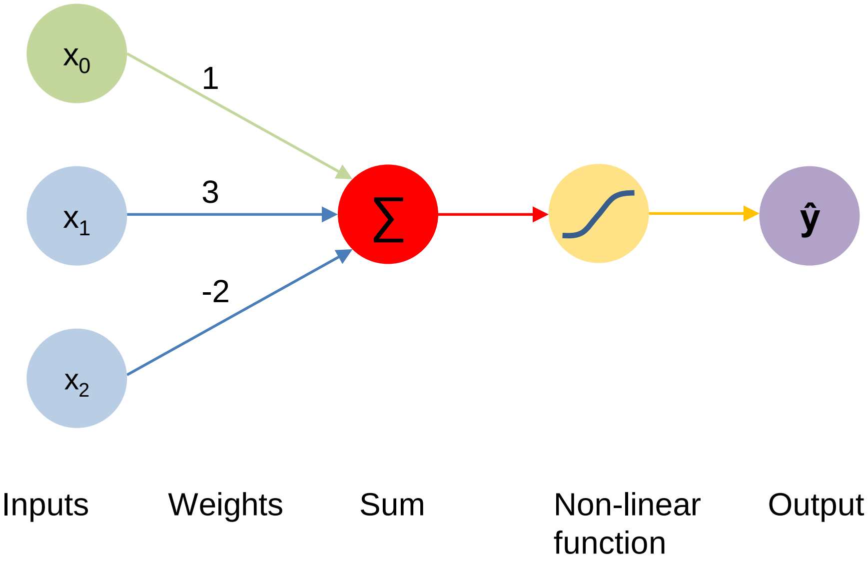
The perceptron: Forward propagation



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

$$
\begin{aligned}
\hat{y} &= g\,(\,w_0 + \boldsymbol{X}^T \boldsymbol{W}\,) \\
&= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\
\hat{y} &= g\,(1 + 3\,x_1 - 2\,x_2)
\end{aligned}
$$

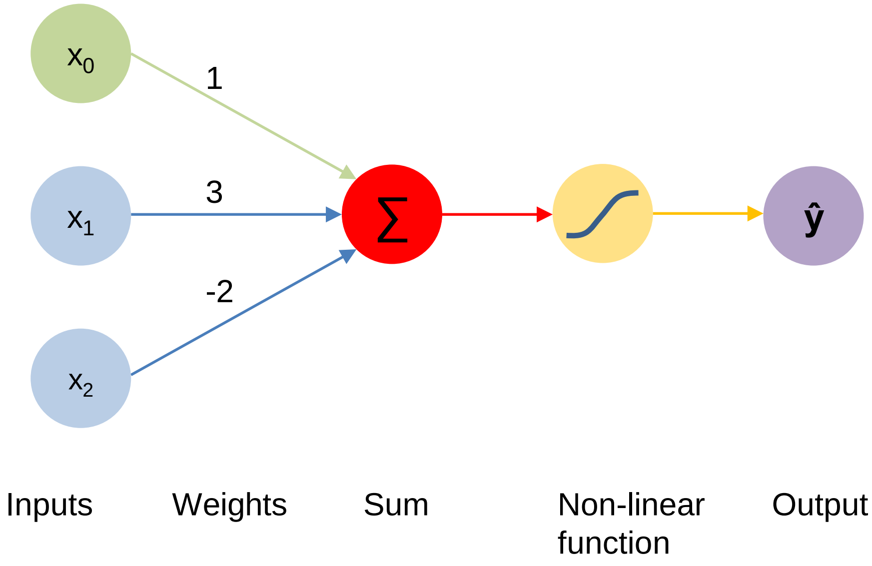
The perceptron: Forward propagation



We have: $\boldsymbol{w}_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

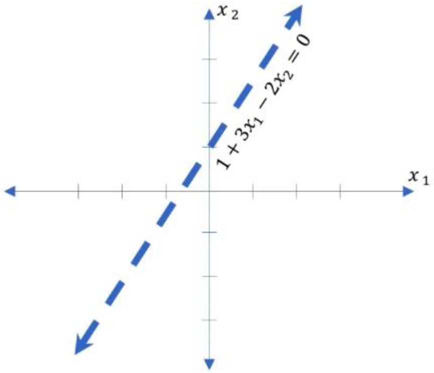
$$\begin{aligned} \hat{y} &= g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right) \\ \hat{y} &= g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}}) \end{aligned}$$

The perceptron: Forward propagation

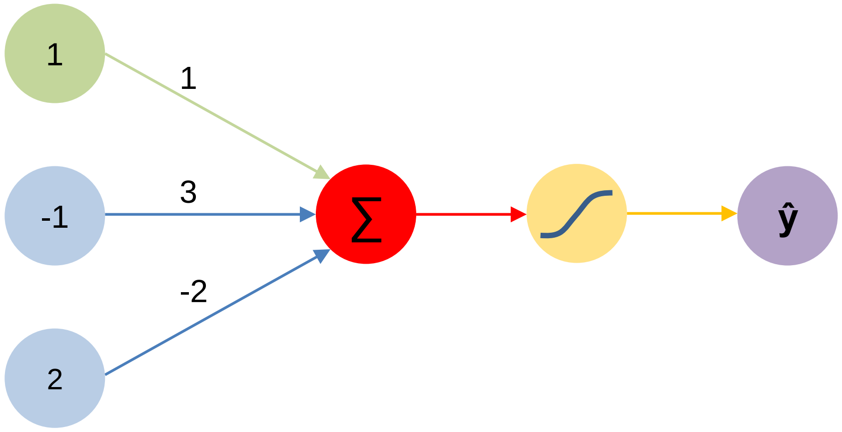


We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g\underbrace{(1 + 3x_1 - 2x_2)}_{\text{This is just a line in 2D!}}$$



The perceptron: Forward propagation



Inputs Weights Sum Non-linear function Output

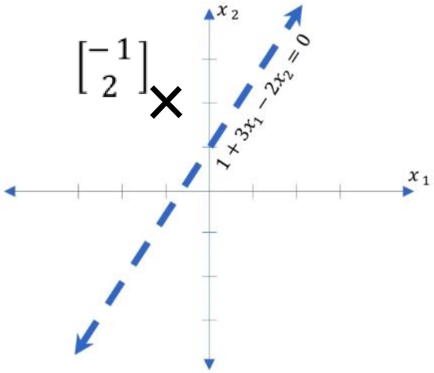
$$\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

$$\begin{aligned} \hat{y} &= g\big(1 + (3 * -1) - (2 * 2)\big) \\ &= g(-6) \approx 0.002 \end{aligned}$$

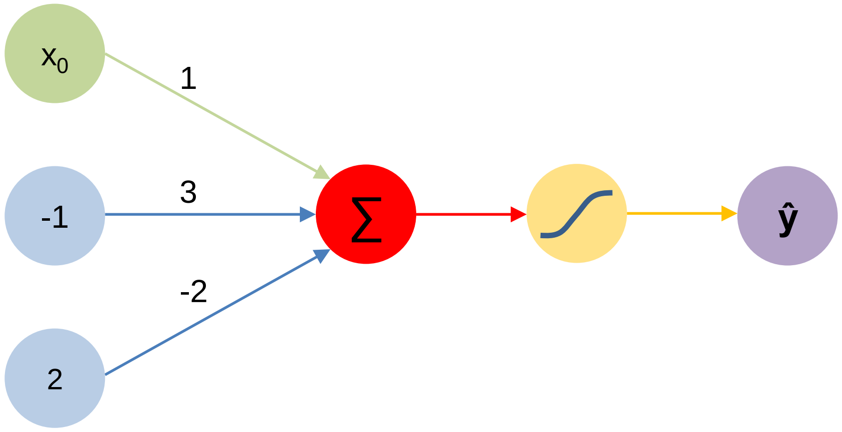
g = Logistic function

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

$$\begin{aligned} \hat{y} &= g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g\underbrace{(1 + 3x_1 - 2x_2)}_{\text{This is just a line in 2D!}} \end{aligned}$$



The perceptron: Forward propagation



Inputs Weights Sum Non-linear function Output

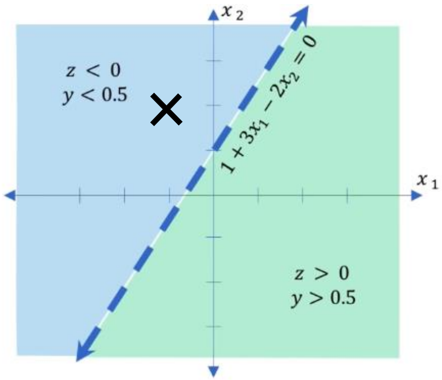
$$\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

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

g = Logistic function

We have: $\boldsymbol{w}_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$



The perceptron: 3 steps

Take a dot product → Add a Bias → Take a non-linearity

Output

Linear combination of inputs

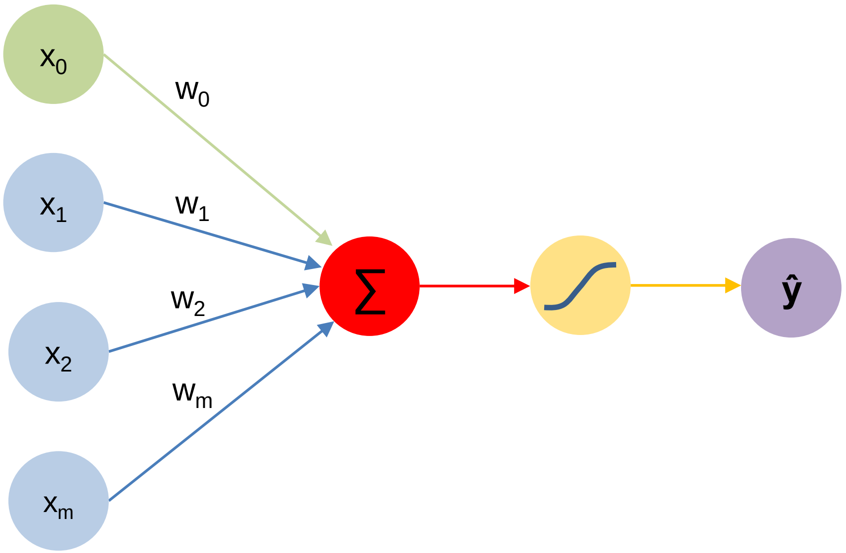
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \, w_i\right)$$

Non-linear activation function

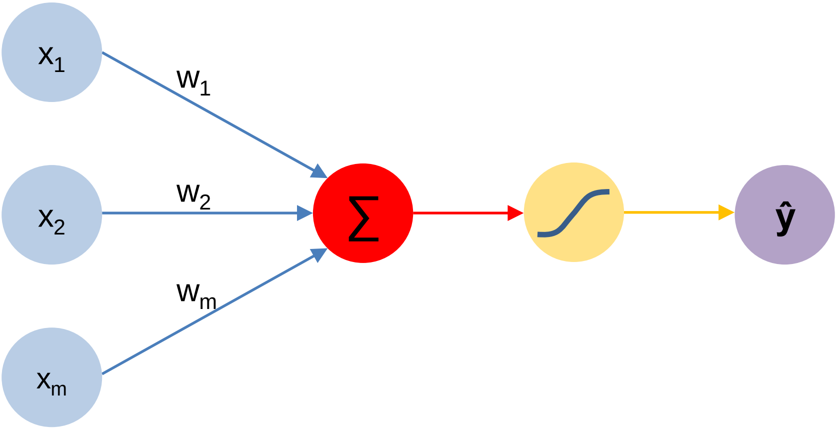
Bias

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

The perceptron

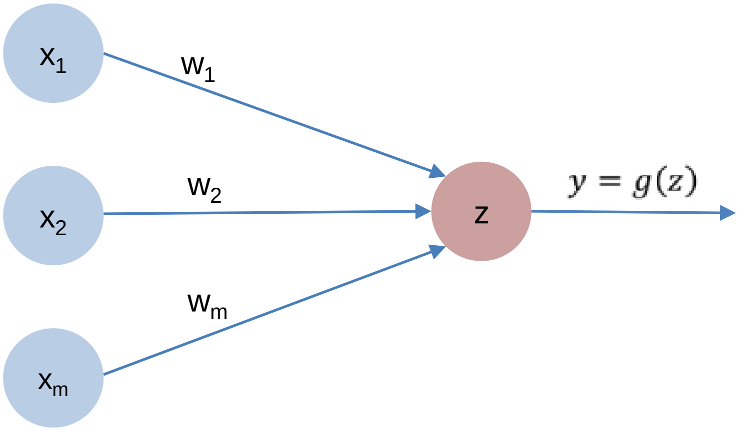


The perceptron



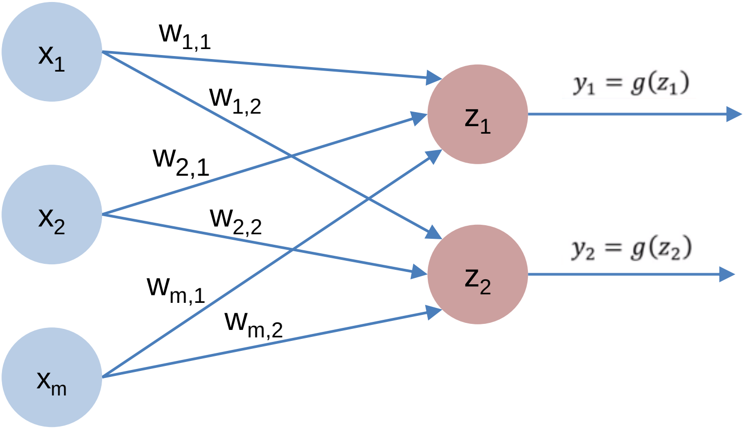
$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$

The perceptron



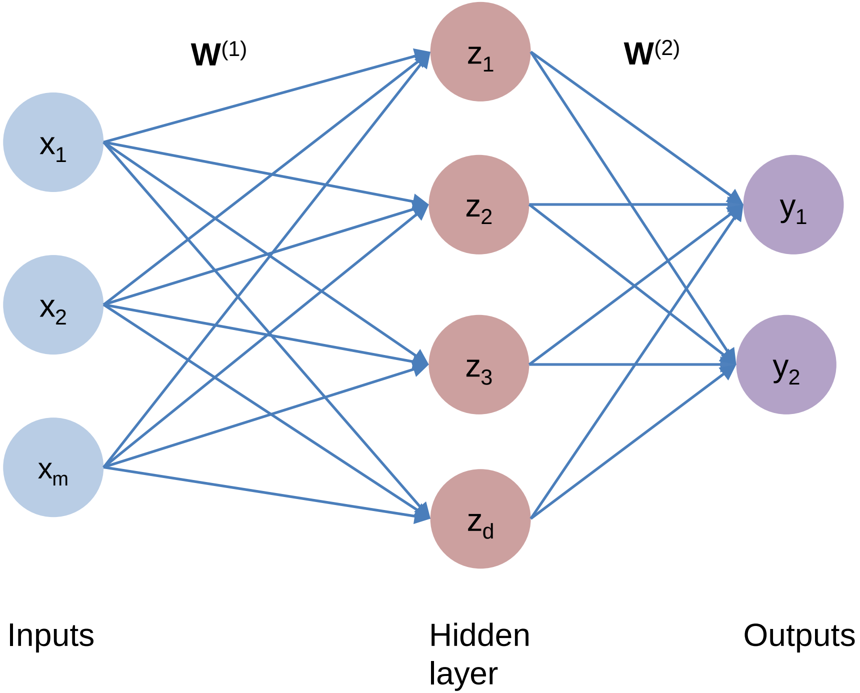
$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$

Multi-output perceptron



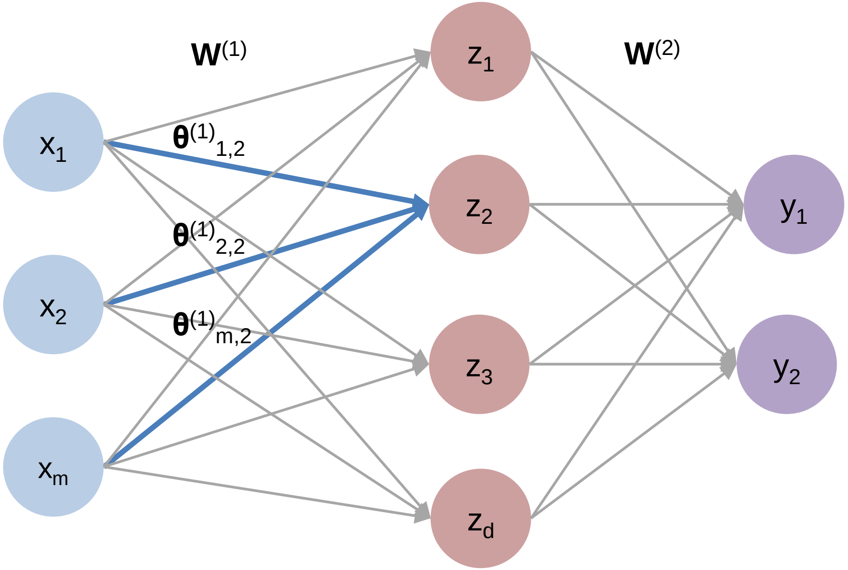
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Single layer Neural network



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

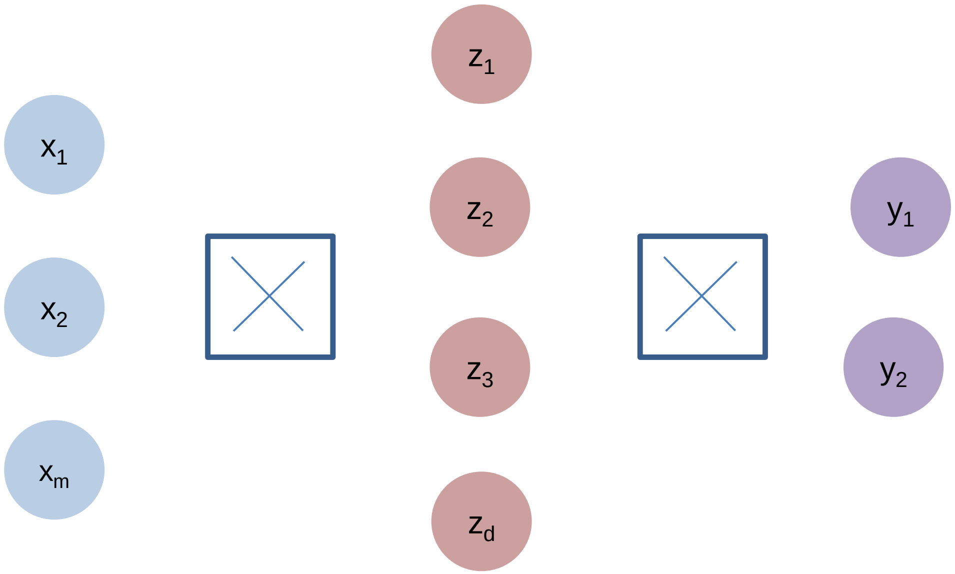
Multi-output perceptron



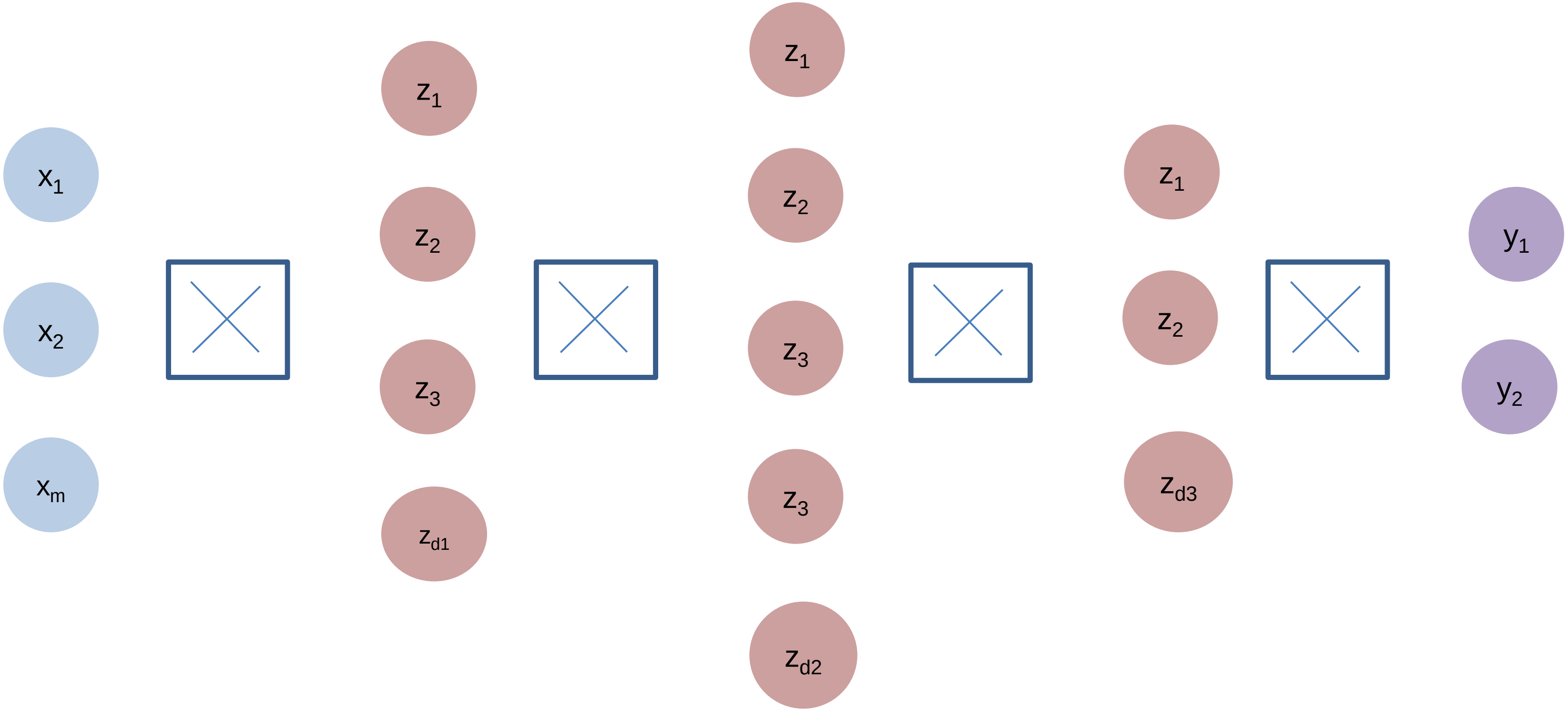
$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

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

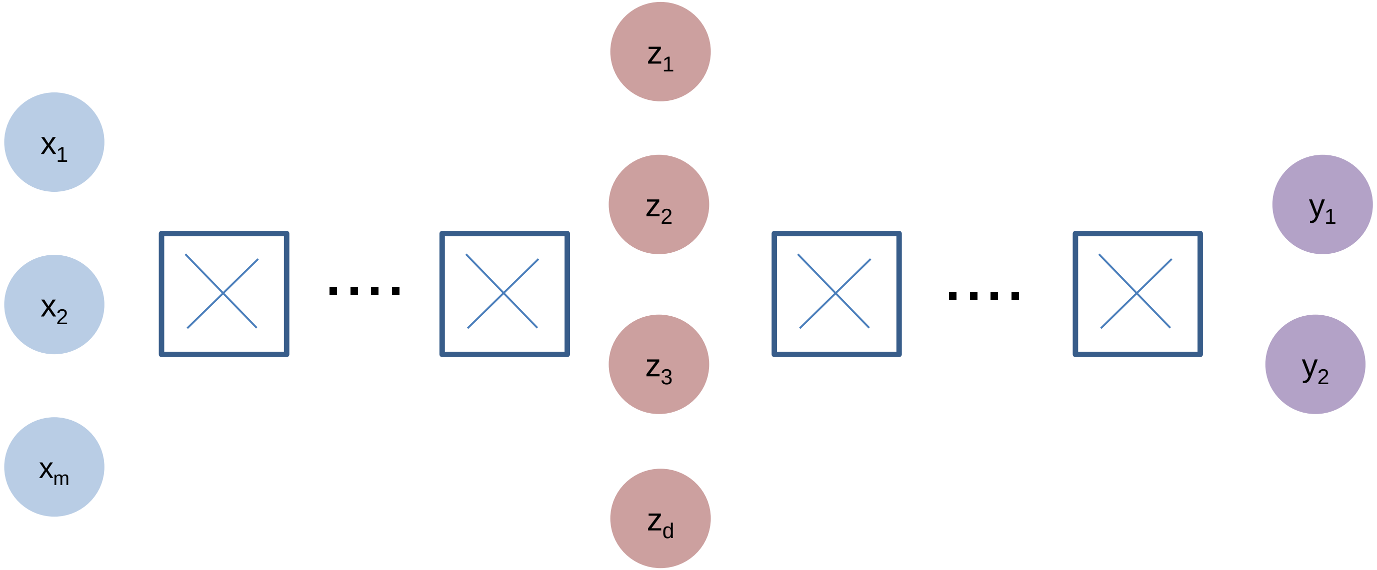
Single layer Neural Network



Deep Neural Network



Deep Neural Network



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

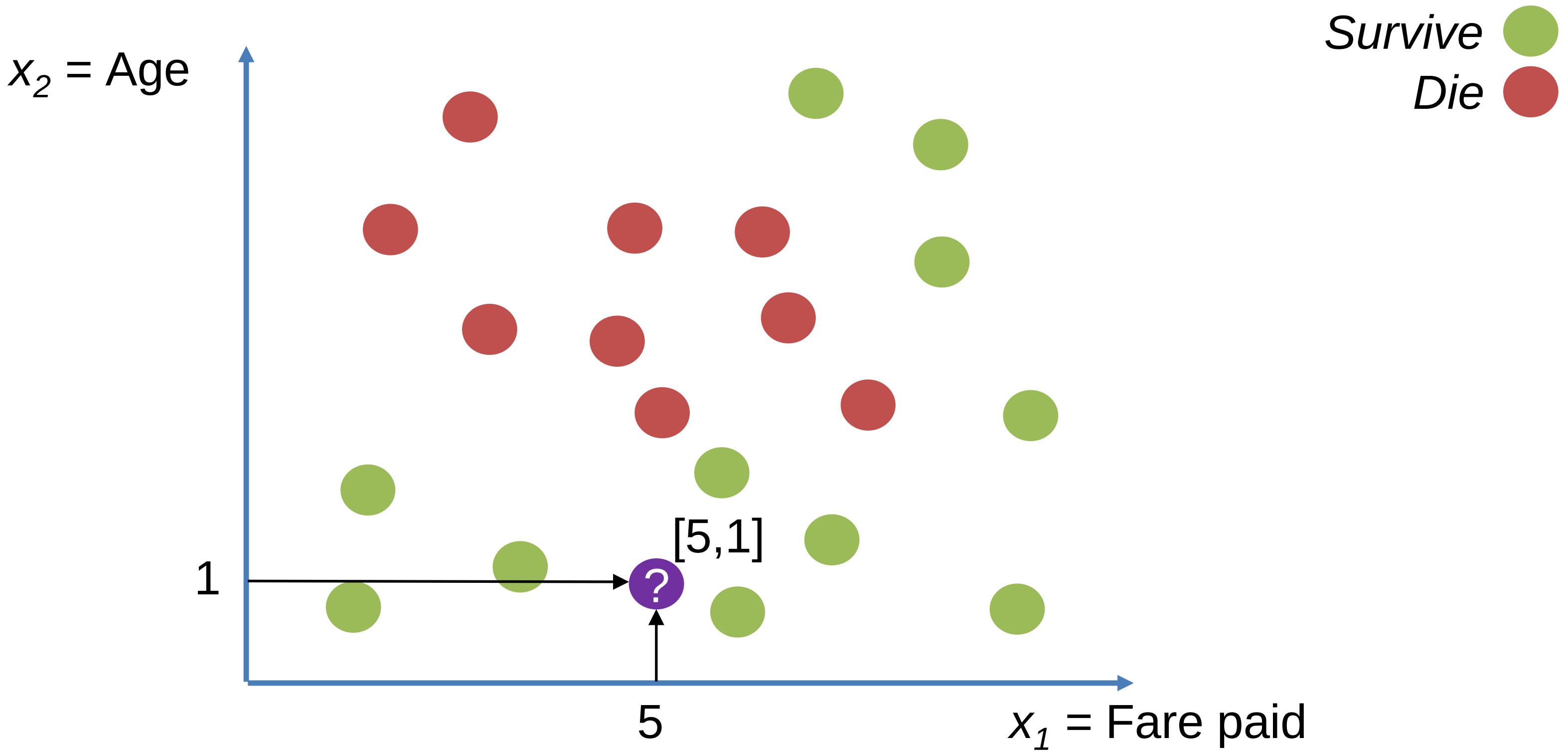
Lets apply a Neural Network

Lets play with the classic Titanic dataset: Predict who will survive!

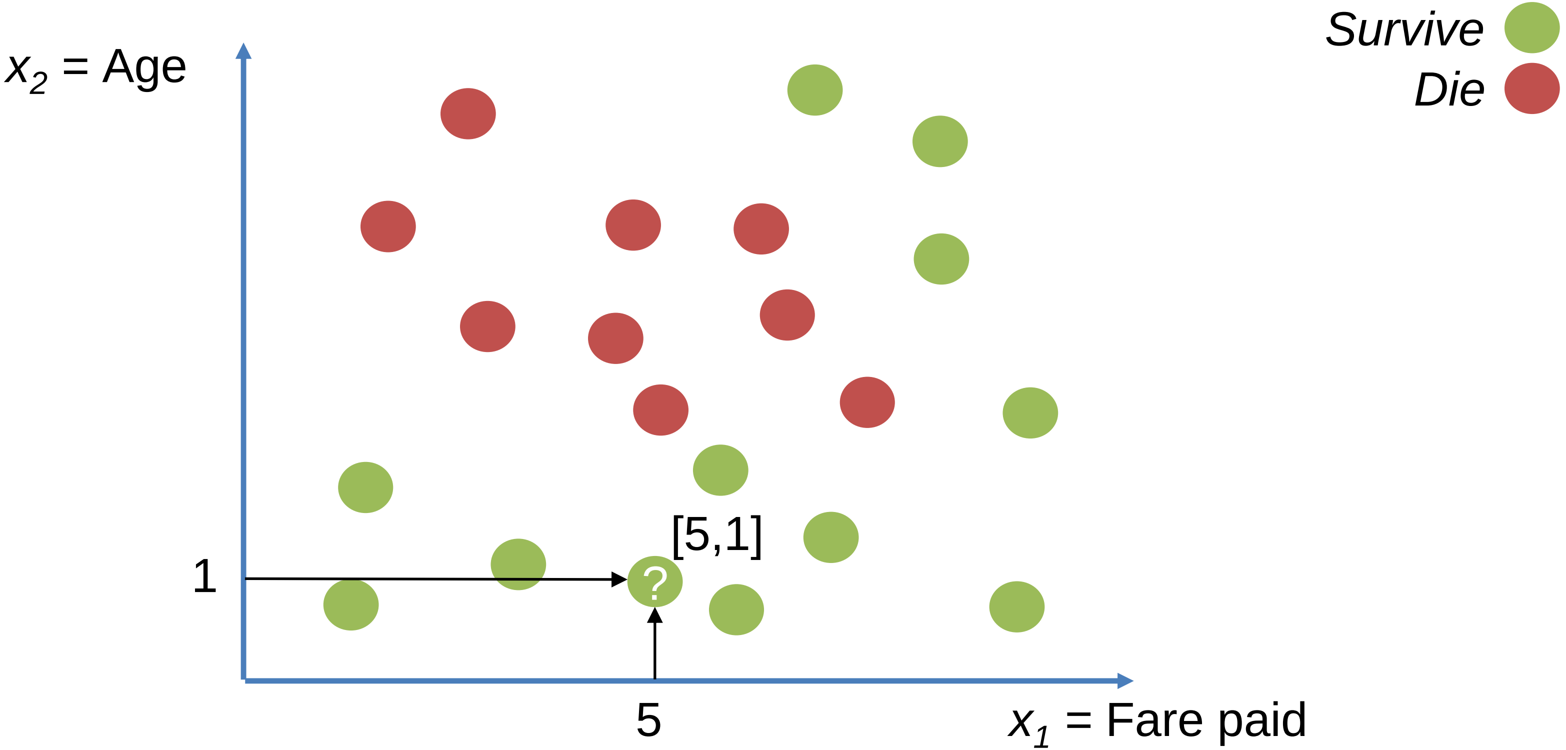
Lets only consider a two feature model:

x_1 = Fare paid
x_2 = Age

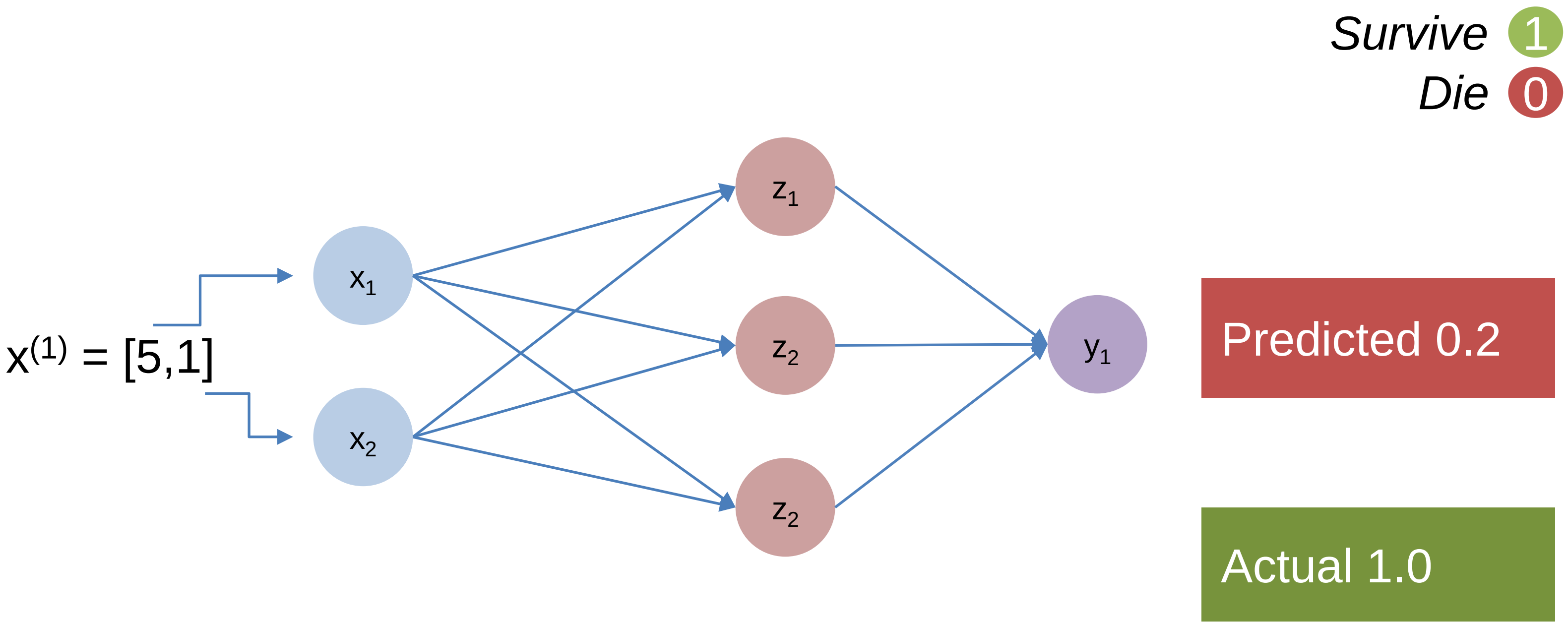
Lets apply a Neural Network



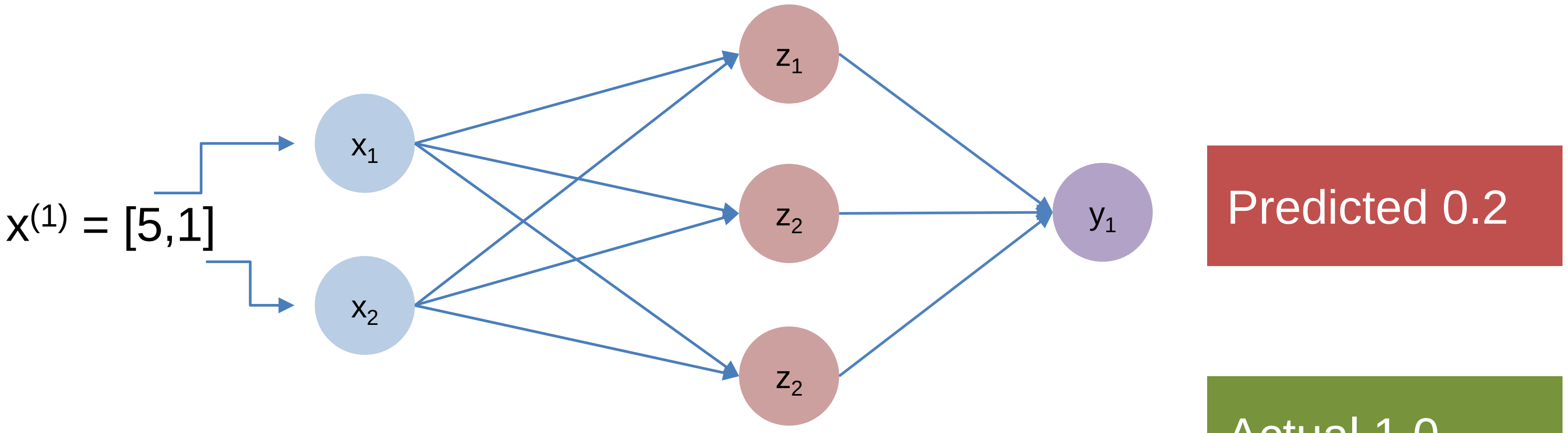
Lets apply a Neural Network



Example



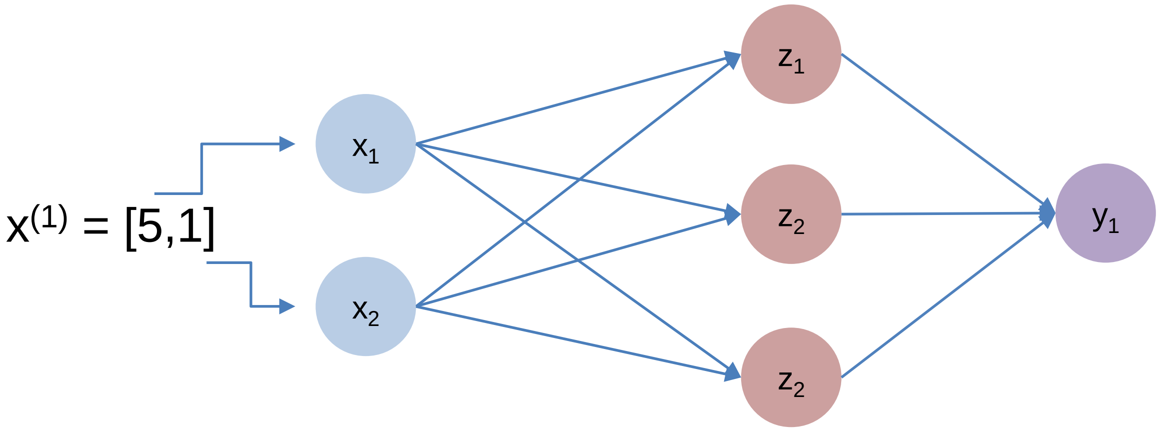
Example



The parameters are currently random!
We need to **train** our network.

Quantifying loss

We can use the errors from our predicted value relative to our actual value.
We use a loss function to define our loss.



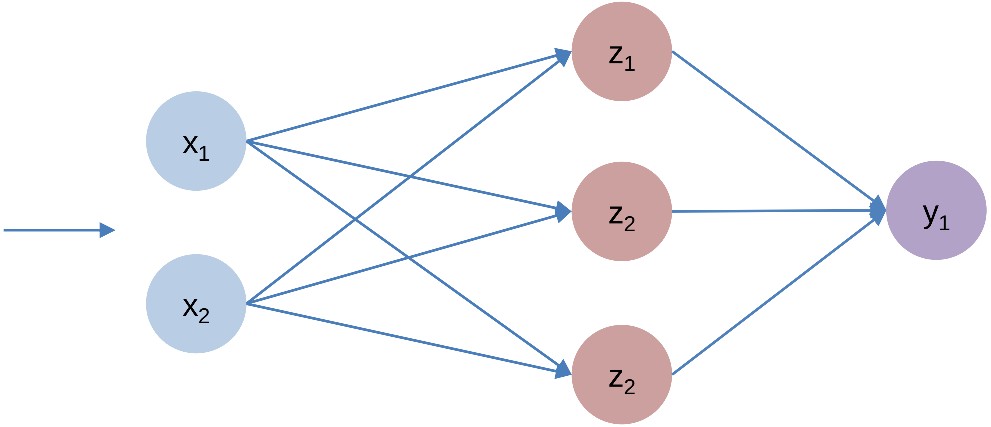
$$\mathcal{L}\left(\underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\right)$$

Empirical Loss

Loss functions are also known as: Objective functions, cost functions, empirical risk

Empirical loss: *The mean loss across all samples*

X = [[5,1]
[3,4]
[2,5]
…
[6,1]]



f(X) =[0.2
0.8
0.3
….
0.8]

Predicted

y = [1
0
0
….
1]

Actual

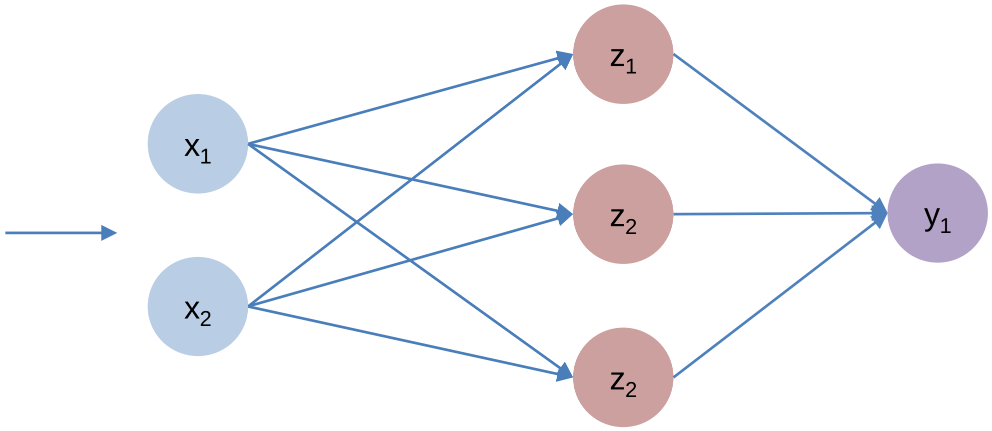
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\big)$$

Empirical Loss

Loss functions are also known as: Objective functions, cost functions, empirical risk

Empirical loss: *The mean loss across all samples*

$$X = \begin{matrix} [[5,1] \\ [3,4] \\ [2,5] \\ \dots \\ [6,1]] \end{matrix}$$



$$f(X) = \begin{matrix} [0.2 \\ 0.8 \\ 0.3 \\ \dots \\ 0.8] \end{matrix} \qquad y = \begin{matrix} [1 \\ 0 \\ 0 \\ \dots \\ 1] \end{matrix}$$

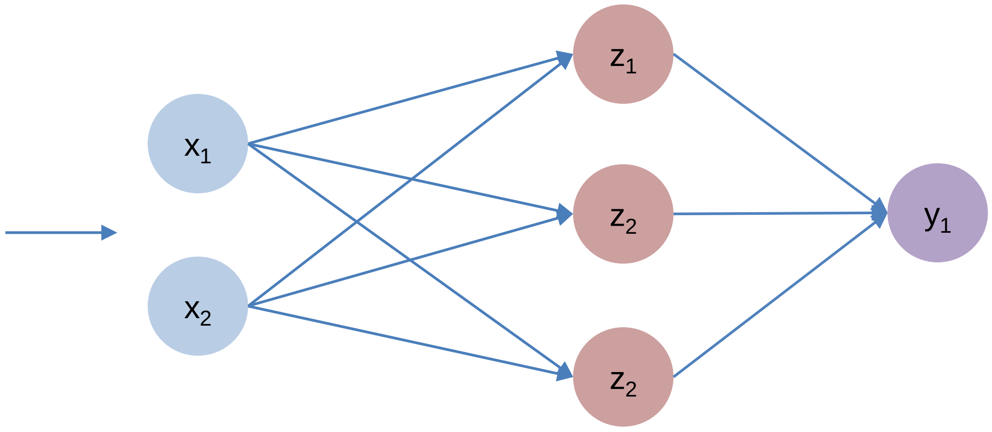
Predicted Actual

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

Binary Cross Entropy Loss

Comparing models that output a probability between 0 and 1

$$X = \begin{bmatrix} [5,1] \\ [3,4] \\ [2,5] \\ \dots \\ [6,1] \end{bmatrix}$$



$$f(X) = [0.2,\ 0.8,\ 0.3,\ \dots,\ 0.8] \qquad y = [1,\ 0,\ 0,\ \dots,\ 1]$$

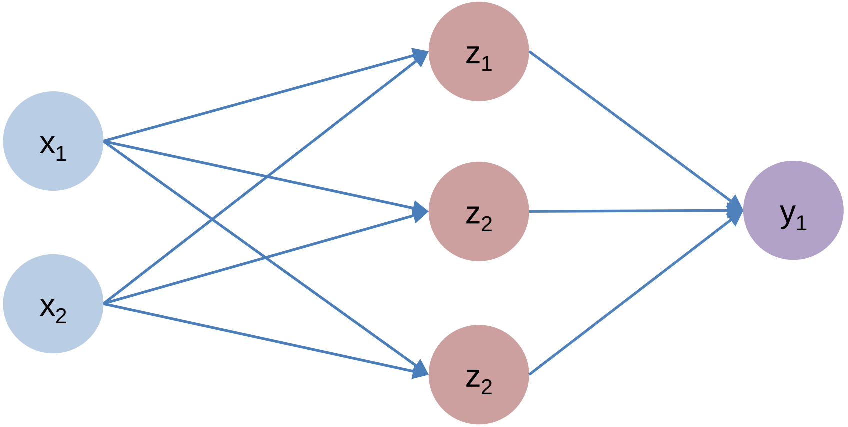
Predicted / Actual

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

Mean Square Error loss

Instead of 0 or 1, we might have a regression model for continuous output values

X = [[5,1]
[3,4]
[2,5]
…
[6,1]]



f(X) =[20
65
30
….
78]

Predicted

y = [17
22
28
….
54]

Actual

$$J(W) = \sum_{i=1}^{n} \frac{\left(w^T x(i) - y(i)\right)^2}{n}$$

Training Neural Network

Use the loss to train the network.

Can we find the weights that achieve the lowest loss?

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

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

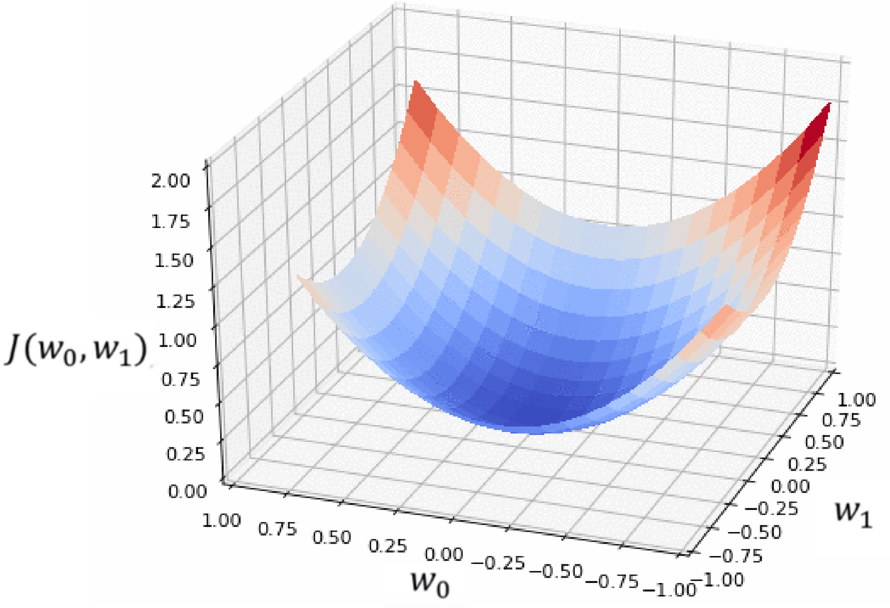
Remember:

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

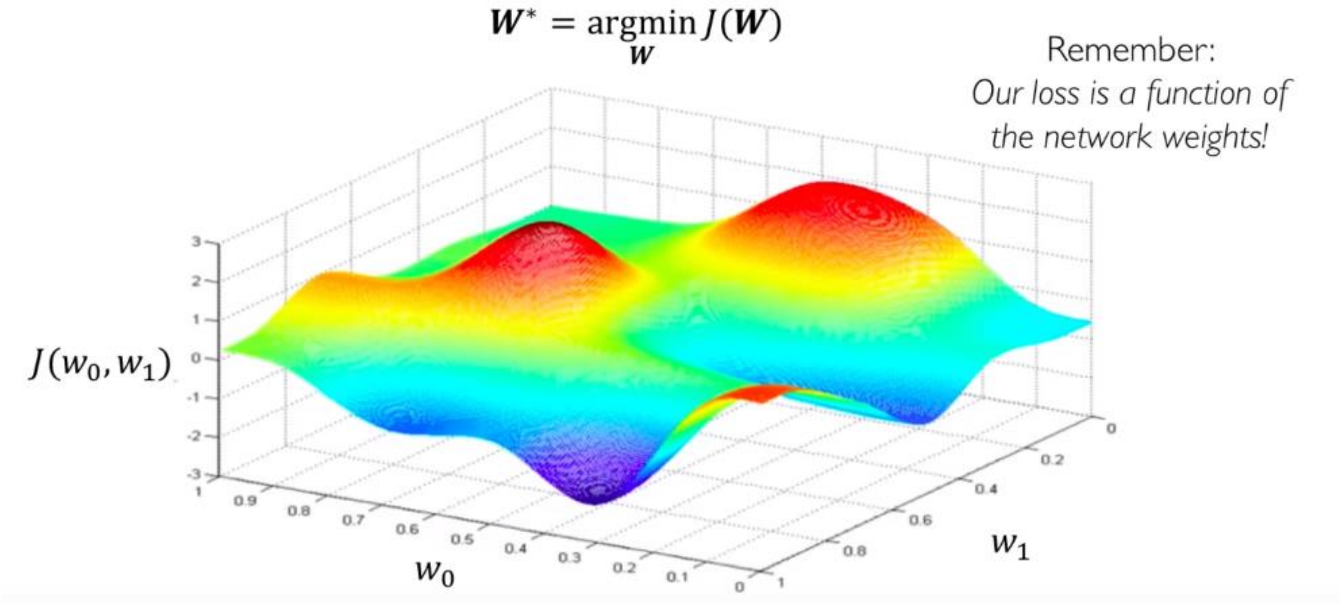
Training Neural Network

Use the loss to train the network.

Can we find the weights that achieve the lowest loss?

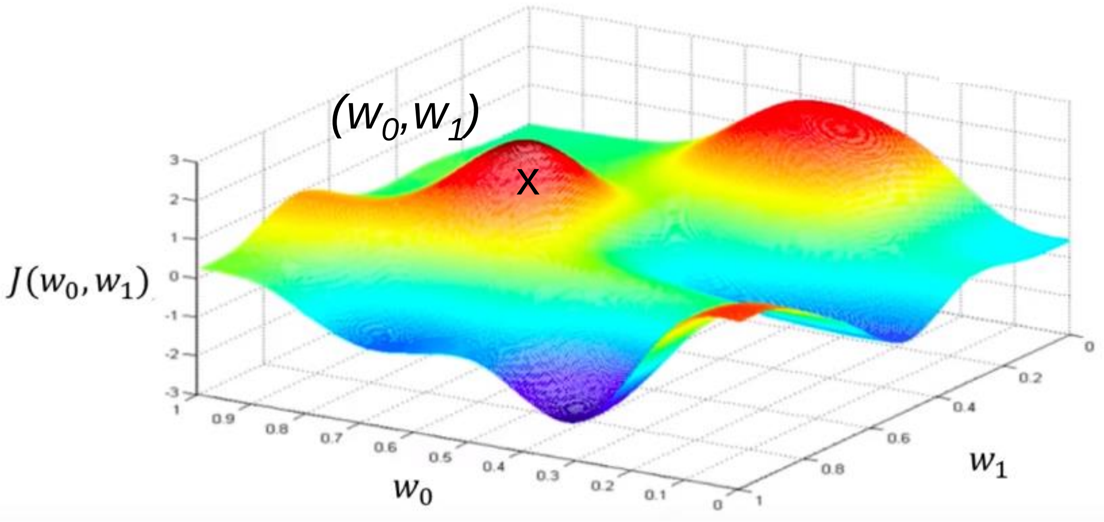


Gradient Descent



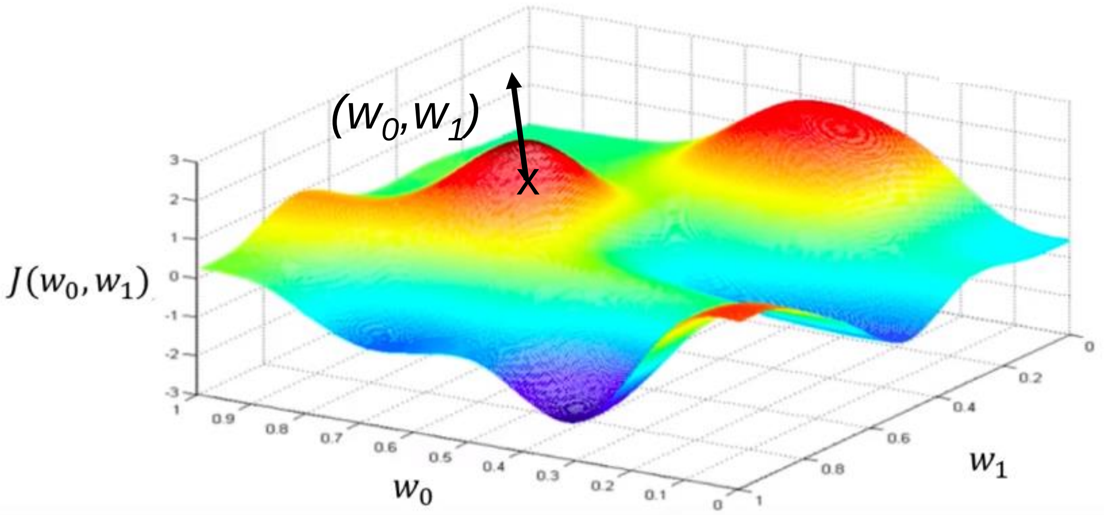
Gradient Descent

Compute the gradient at (w_0,w_1) $\frac{\partial J(W)}{\partial W}$



Gradient Descent

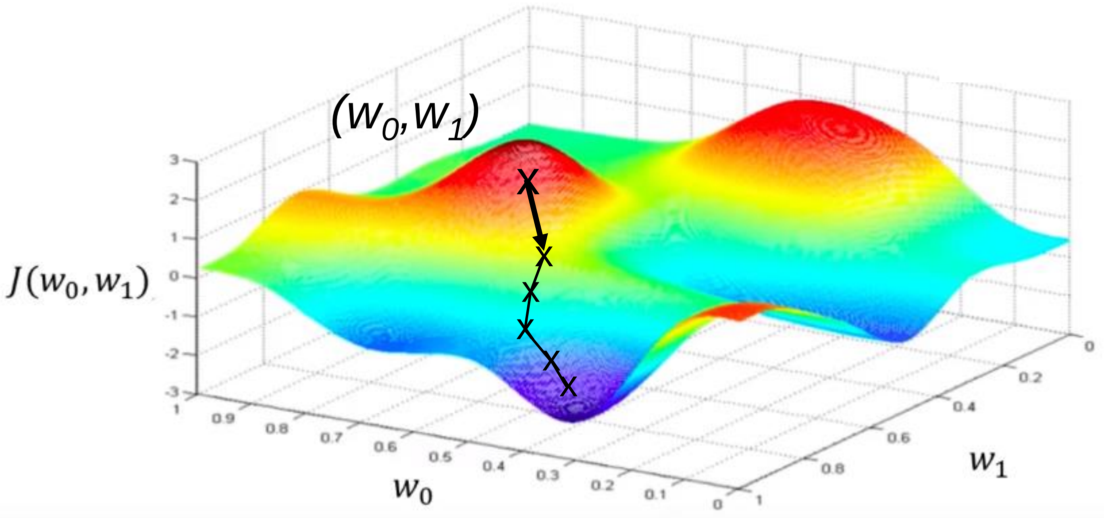
Compute the gradient at (w_0,w_1) $\frac{\partial J(W)}{\partial W}$



Notice this is the direction of maximum descent!

Gradient Descent

Take the opposite direction of maximum gradient



Summary of gradient descent

Initialize Weights → Compute Gradient → Update weights

Initialize Weights

- Random weights
- Draw weights from a Normal distribution

Compute Gradient

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

Update weights

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

Loop over until no large changes in W are seen.

How do we compute the gradient?

Initialize Weights

- Random weights
- Draw weights from a Normal distribution

Compute Gradient

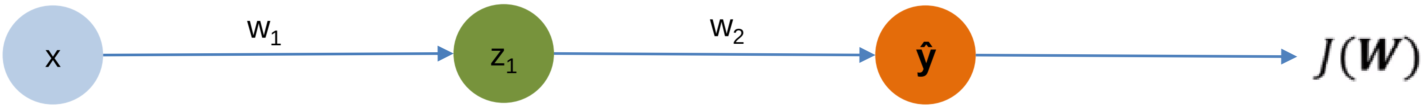
$$\frac{\partial J(W)}{\partial W}$$

Update weights

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

Loop over until no large changes in W are seen.

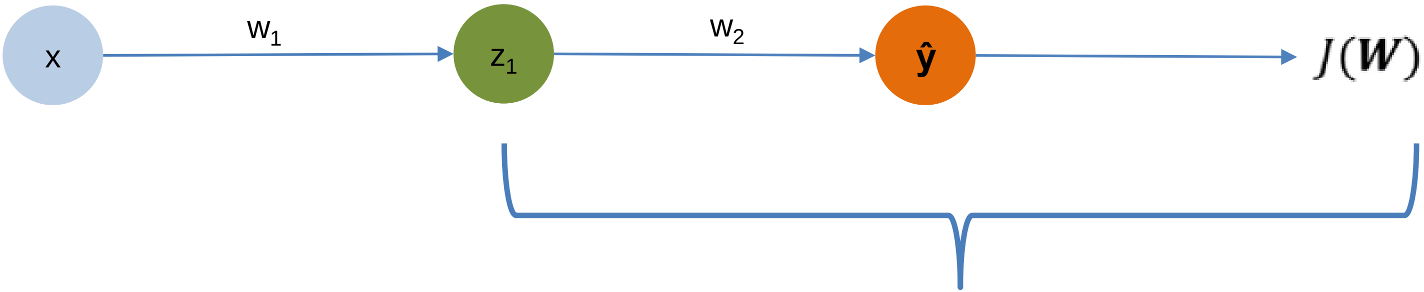
How do we compute the gradient?
Backpropagation



Compute how a small change in a weight, such as w_2, affects the final loss.

Lets calculate the gradient of the loss given w_2.

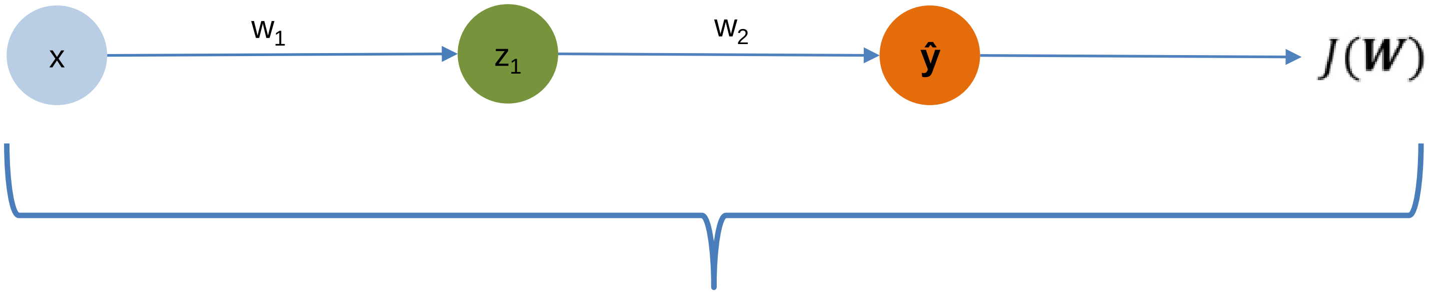
How do we compute the gradient?
Backpropagation



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

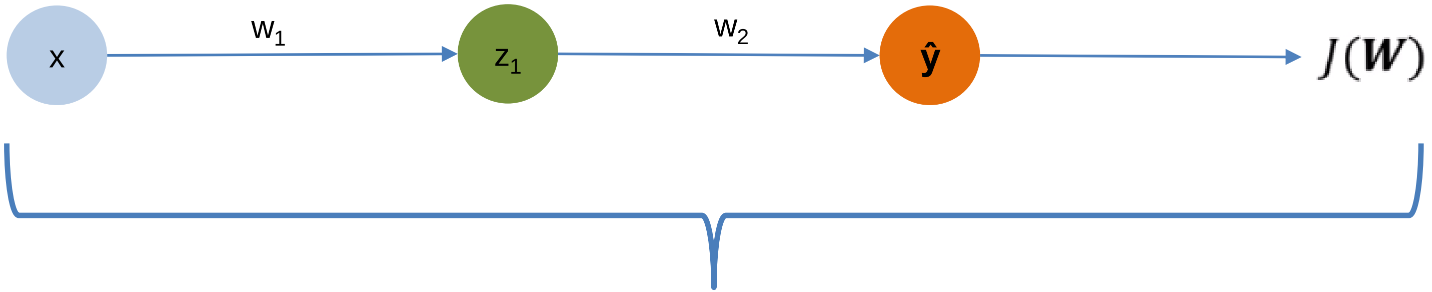
Compute the chain rule!

How do we compute the gradient?
Backpropagation



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

How do we compute the gradient?
Backpropagation



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

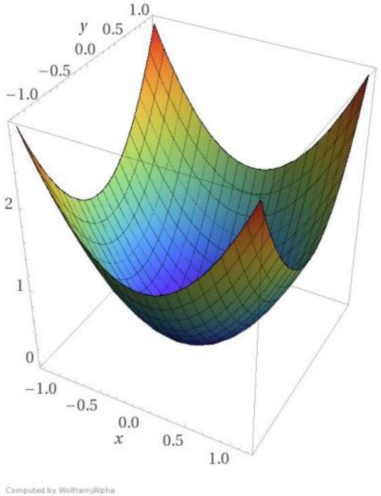
Back propagating the errors to the original input.

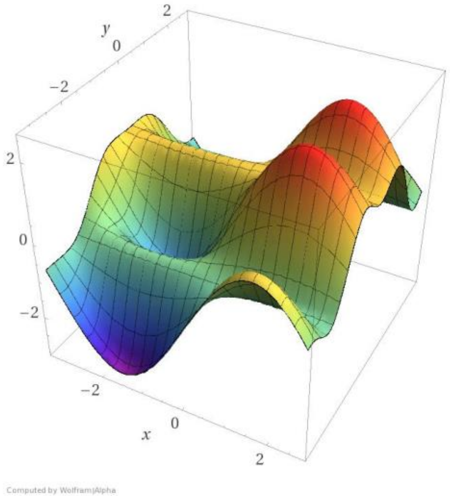
Repeat for every weight in the network!

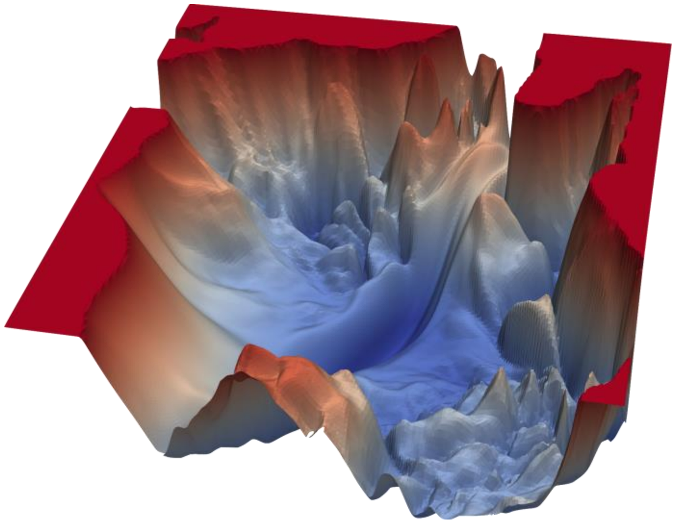
Loss Landscape

In practice training a real neural network is highly complex

Many local minima. Finding true minimum is difficult.







Summary of gradient descent

Initialize Weights → Compute Gradient → Update weights

- Random weights
- Draw weights from a Normal distribution

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

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

Loop over until no large changes in W are seen.

Summary of gradient descent

Initialize Weights

- Random weights
- Draw weights from a Normal distribution

Compute Gradient

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

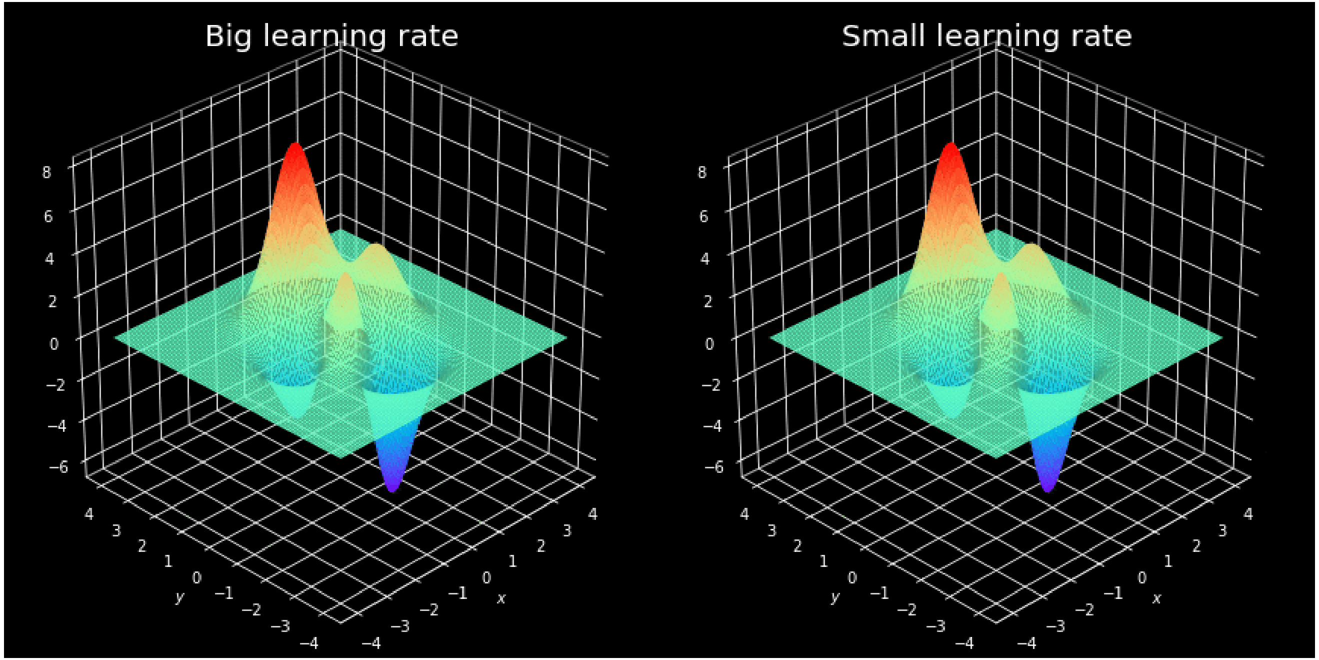
Update weights

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

$$\underbrace{\text{Compute Gradient} \rightarrow \text{Update weights}}_{\text{Loop over until no large changes in W are seen.}}$$

Loop over until no large changes in W are seen.

Learning rate



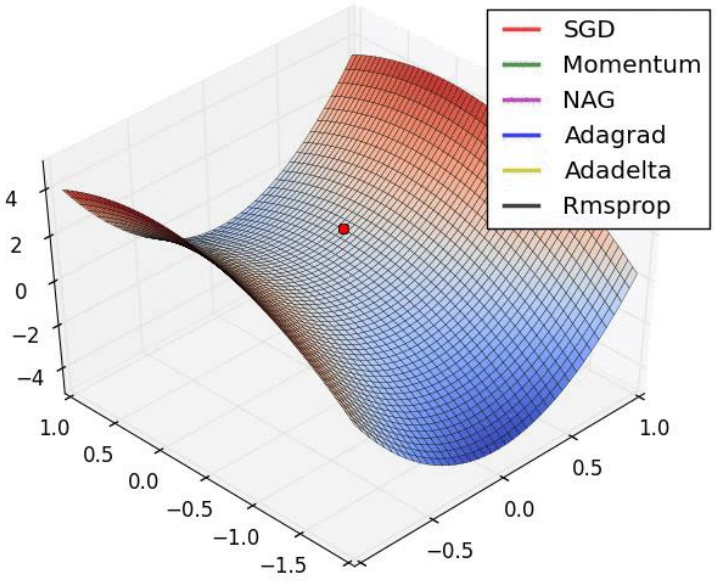
Learning rate

Optimising the learning rate?

Option 1: Fixed Learning Rates

Option 2: Adaptive Learning Rate algorithms

- Magnitude of gradient
- Size of weights
- Current learning rate
- Etc…



Tips for training: Computing gradients

Computing gradient across all data points is expensive to compute $\frac{\partial J(W)}{\partial W}$

Initialize Weights → Compute Gradient → Update weights

- Random weights
- Draw weights from a Normal distribution

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

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

Loop over until no large changes in W are seen.

Tips for training: Computing gradients

Computing gradient across all data points is expensive to compute $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$

Initialize Weights → Pick a single data point i → Compute Gradient → Update weights

Initialize Weights

- Random weights
- Draw weights from a Normal distribution

Pick a single data point i

Compute Gradient

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$$

Update weights

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

Loop over until no large changes in W are seen.

Stochastic gradient descent!

Easy to compute but noisy

Tips for training: Computing gradients

Computing gradient across all data points is expensive to compute $\frac{\partial J(W)}{\partial W}$

Initialize Weights → Pick a batch B data points → Compute Gradient → Update weights

Initialize Weights

- Random weights
- Draw weights from a Normal distribution

Pick a batch B data points

Compute Gradient

$$\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$$

Update weights

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

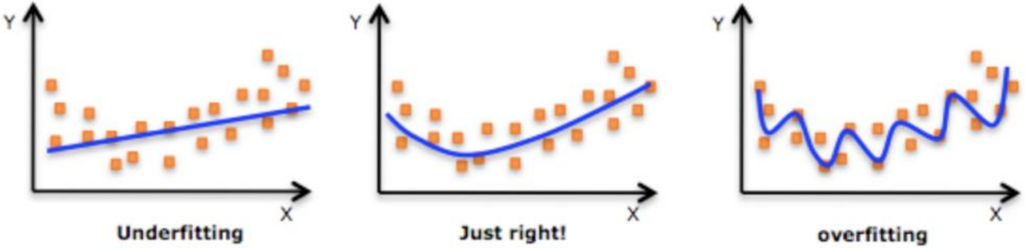
Loop over until no large changes in W are seen.

Batch gradient descent

Fast to compute and much better estimate than stochastic gradient descent

Tips for training: Overfitting

Regression



Classification

Tips for training: Overfitting

Similar to other algorithms (SVMs, Ridge Regression, etc.), we can implement regularization

What is it?

It constrains our optimization problem to discourage complex models

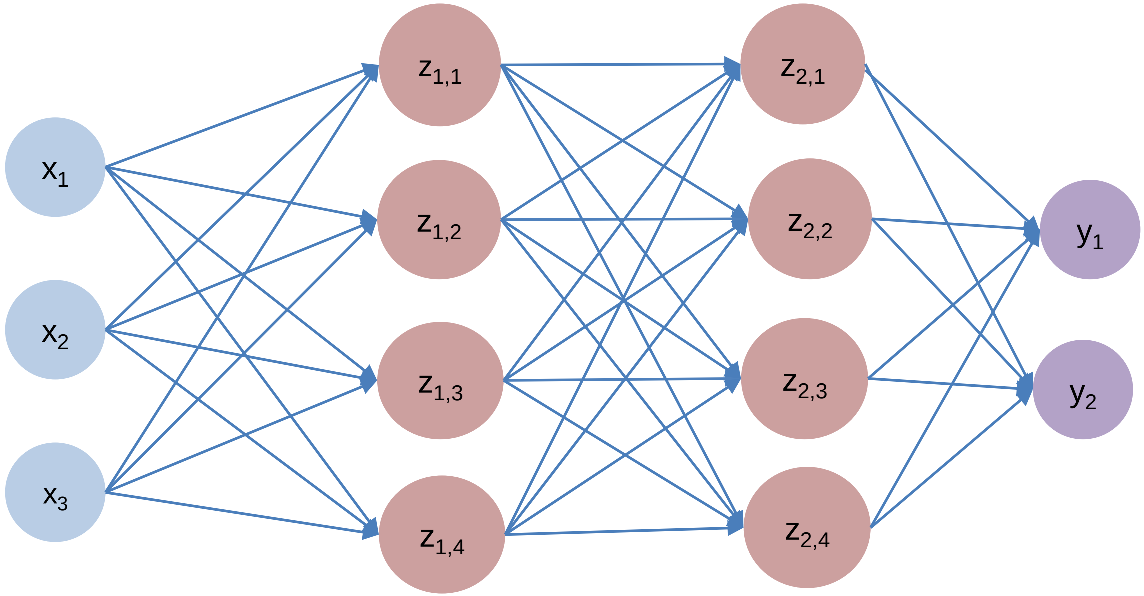
Why do we use it?

We need to make sure that we are producing a model that is as close to the generating function of the data.

We want our model to generalize to unseen data!

Dropout

Dropout randomly sets some activation neurons to 0

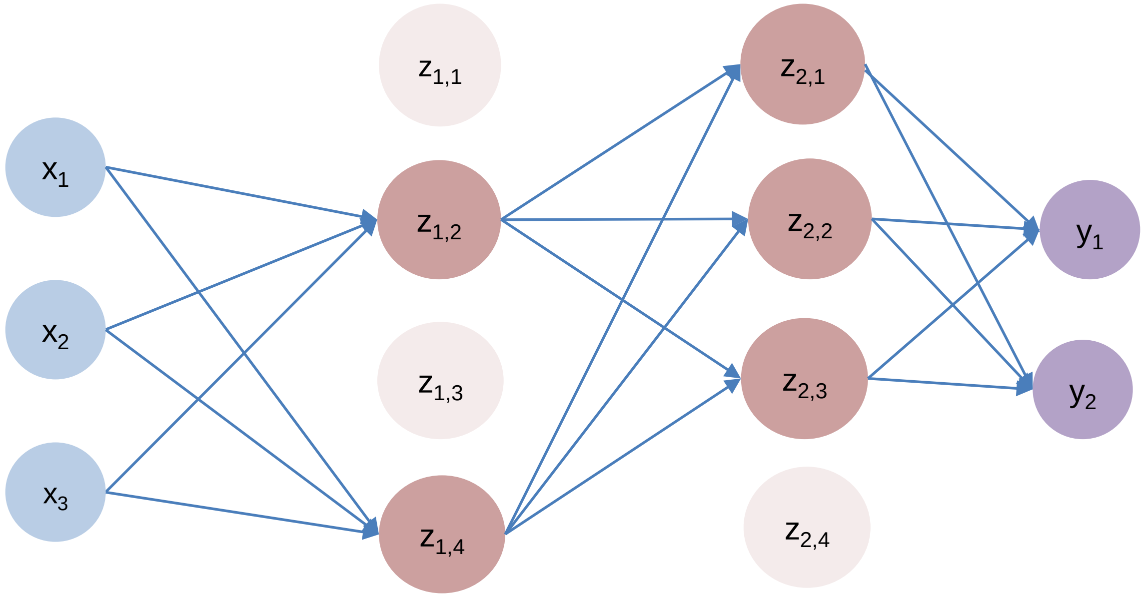


Dropout

Dropout randomly sets some activation neurons to 0

Typically 50% of neurons in each layer

Prevents reliance on single nodes

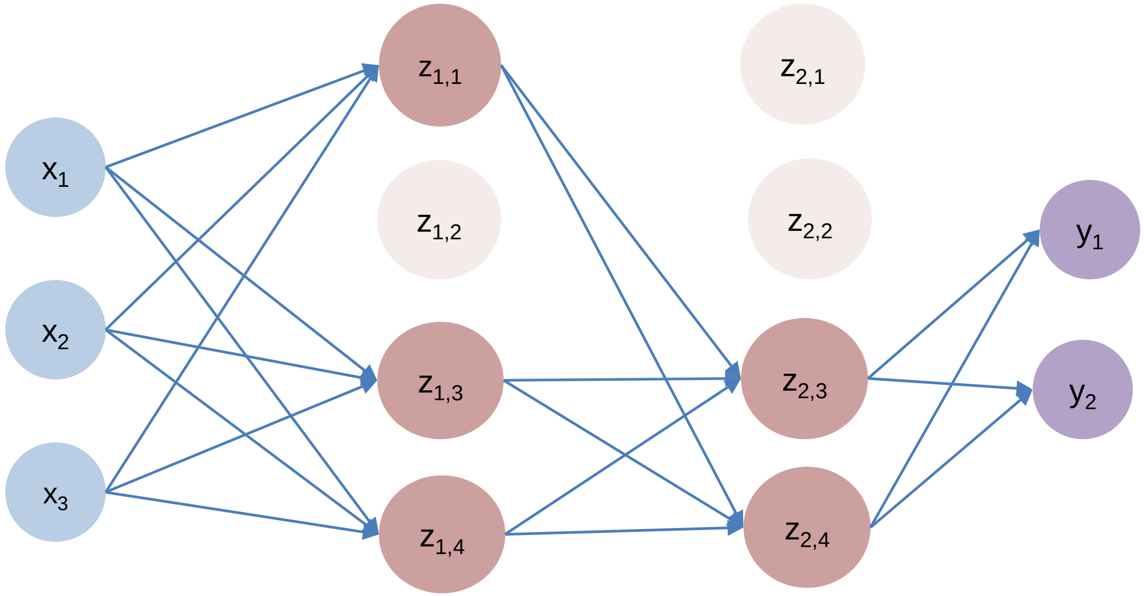


Dropout

Dropout randomly sets some activation neurons to 0

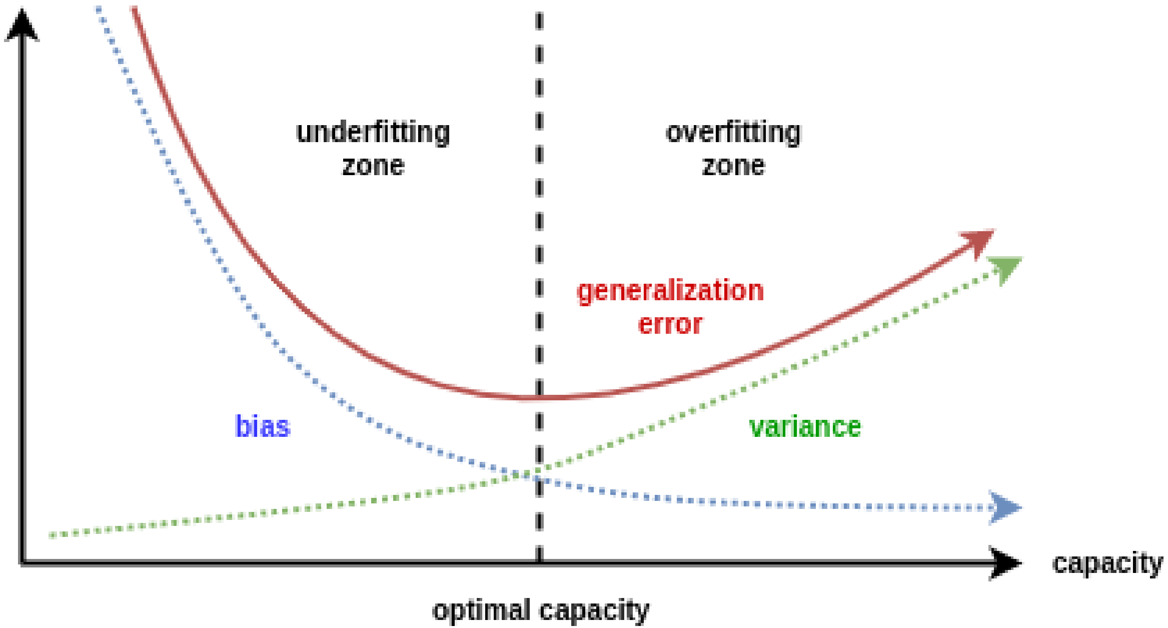
Typically 50% of neurons in each layer

Prevents reliance on single nodes



Early Stopping

If the model trains for long enough the a very complex and unbiased model can be learned but the variance or error increases as seen in the overfitting zone.



Quick Review

Perceptron

- A linear sum
- Non-linear activation function

Neural Network

- Stacking of perceptrons
- Optimisation through back propagation

Training

- Regularisation optimization
- Learning rate