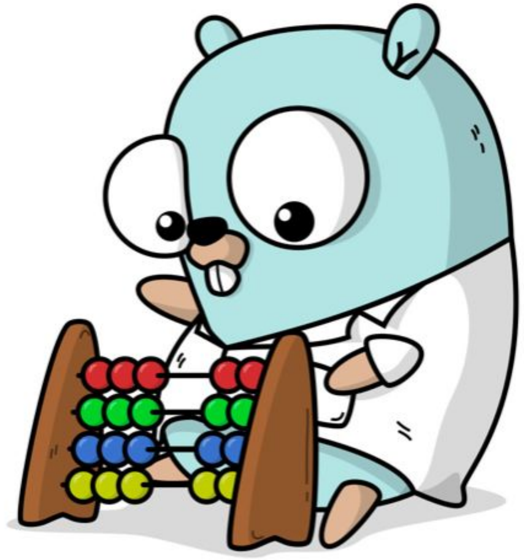
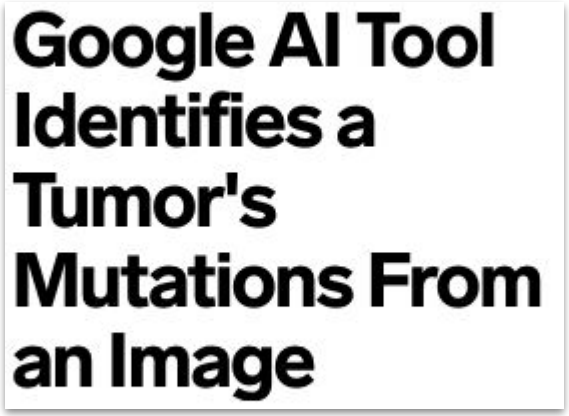
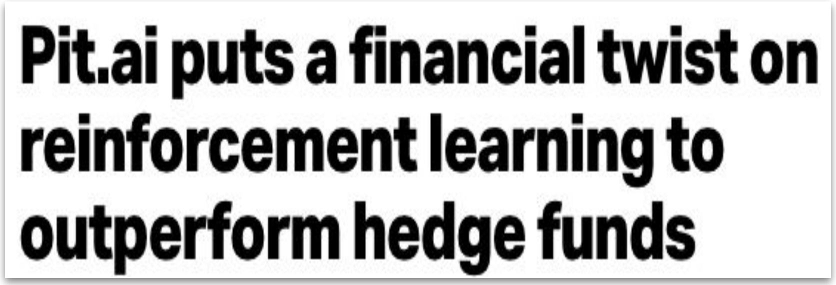
Deep Learning for Gophers

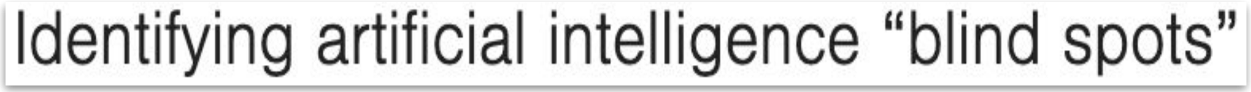
@iamrashminagpal

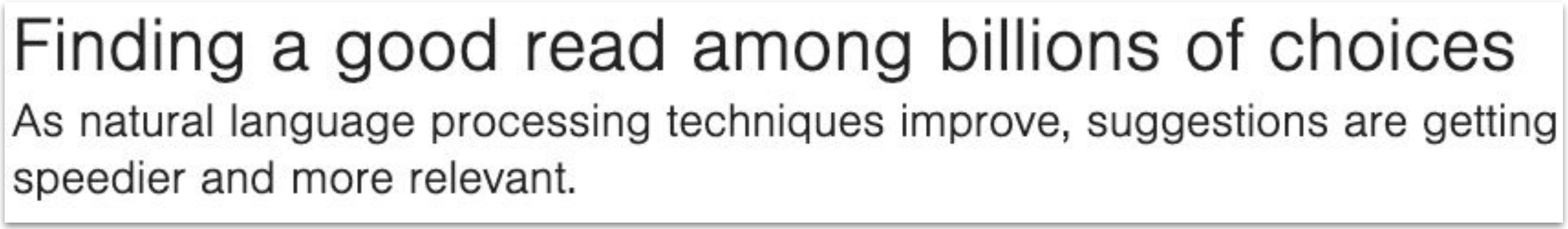


Rise of Deep Learning


Google AI Tool Identifies a Tumor's Mutations From an Image


Pit.ai puts a financial twist on reinforcement learning to outperform hedge funds


Identifying artificial intelligence “blind spots”

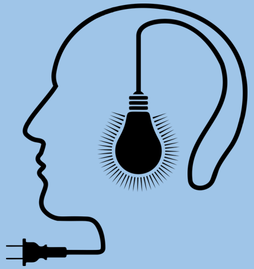

Finding a good read among billions of choices

As natural language processing techniques improve, suggestions are getting speedier and more relevant.

What is Deep Learning?

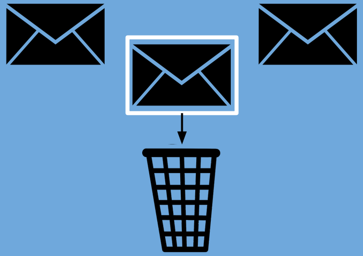
ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



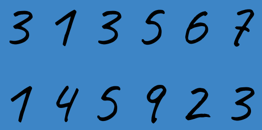
MACHINE LEARNING

Ability to learn without explicitly being programmed



DEEP LEARNING

Extract patterns from data using neural networks

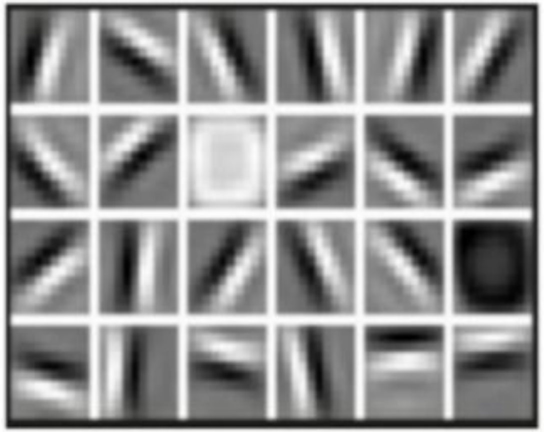


Why Deep Learning?

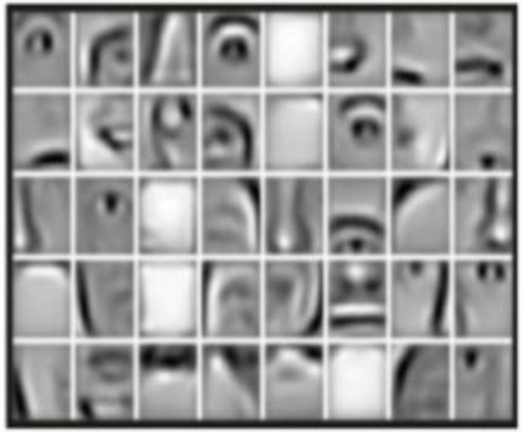
Hand engineered features are time consuming, brittle & not scalable in practise

Can we learn **underlying features** directly from data?

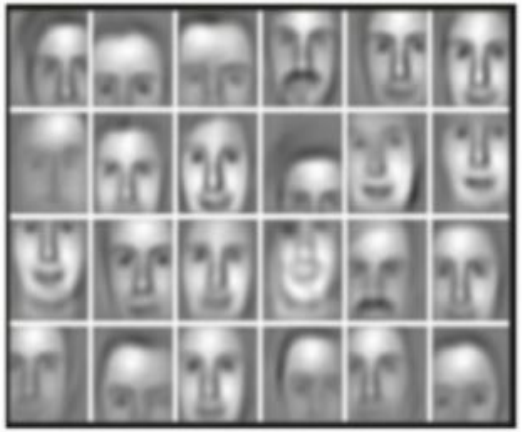
Low Level Features



Mid Level Features

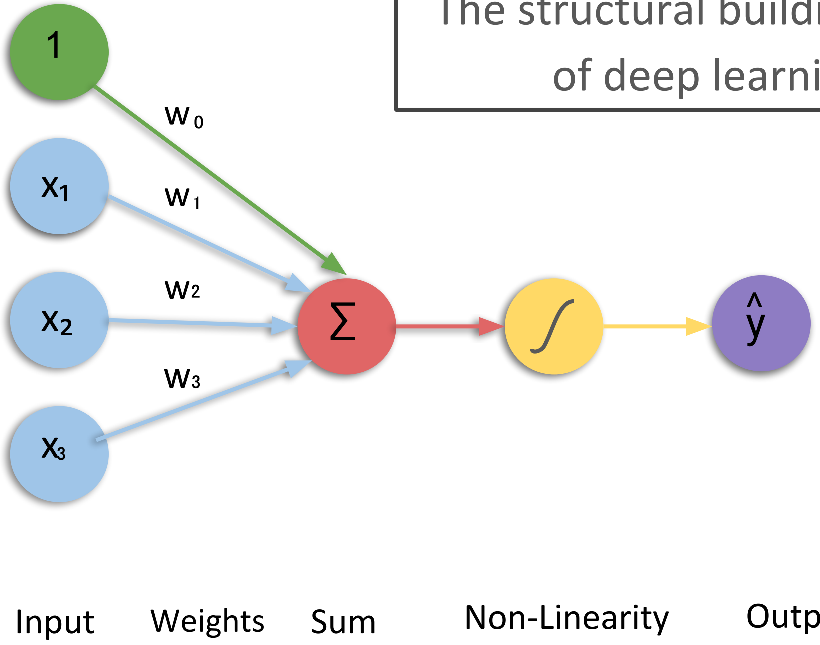


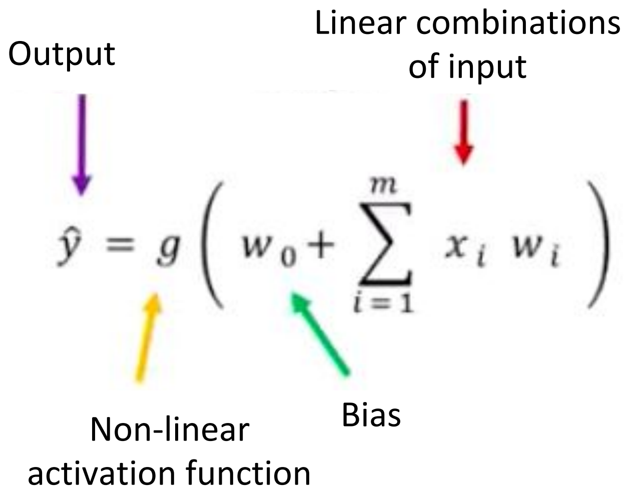
High Level Features



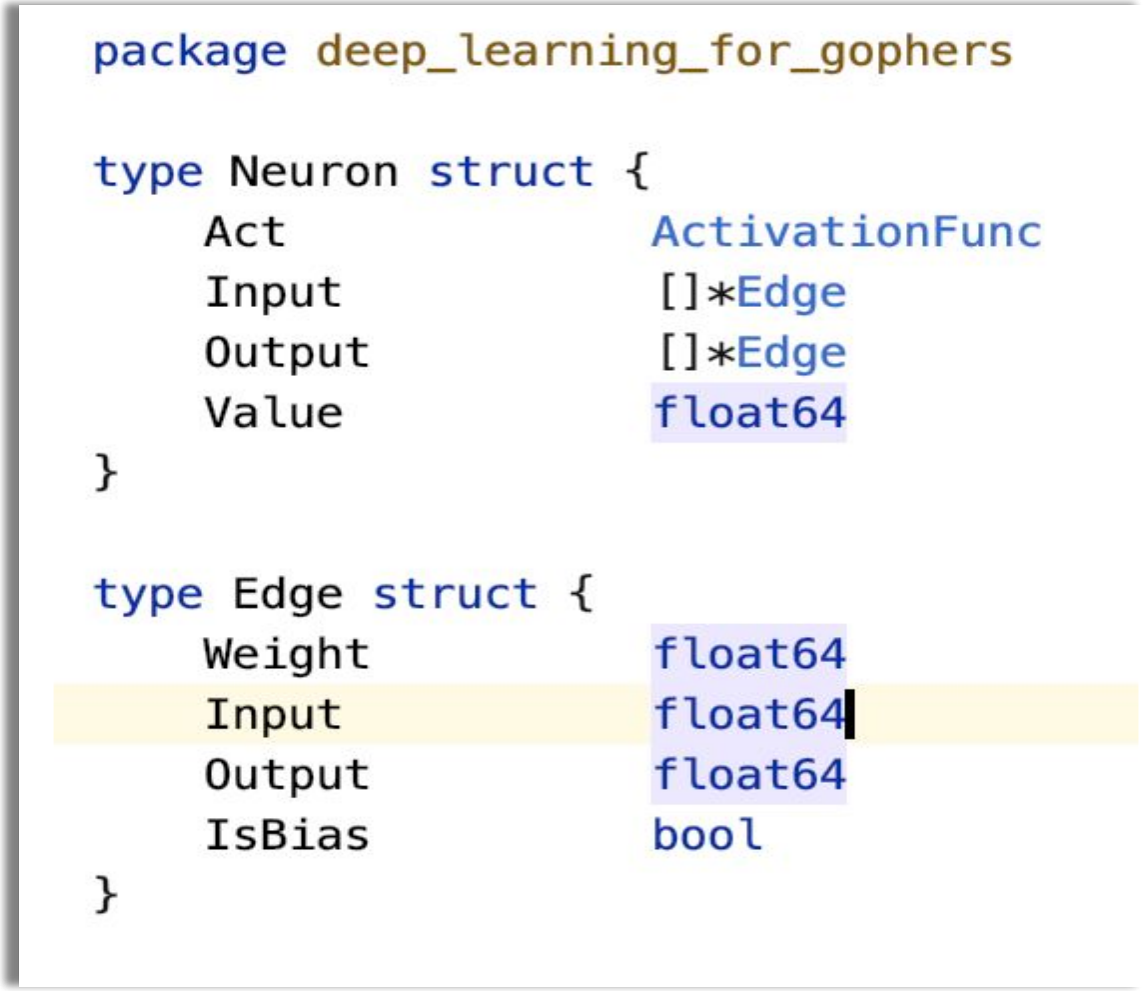
The Perceptron : Feedforward Propagation

The structural building block of deep learning



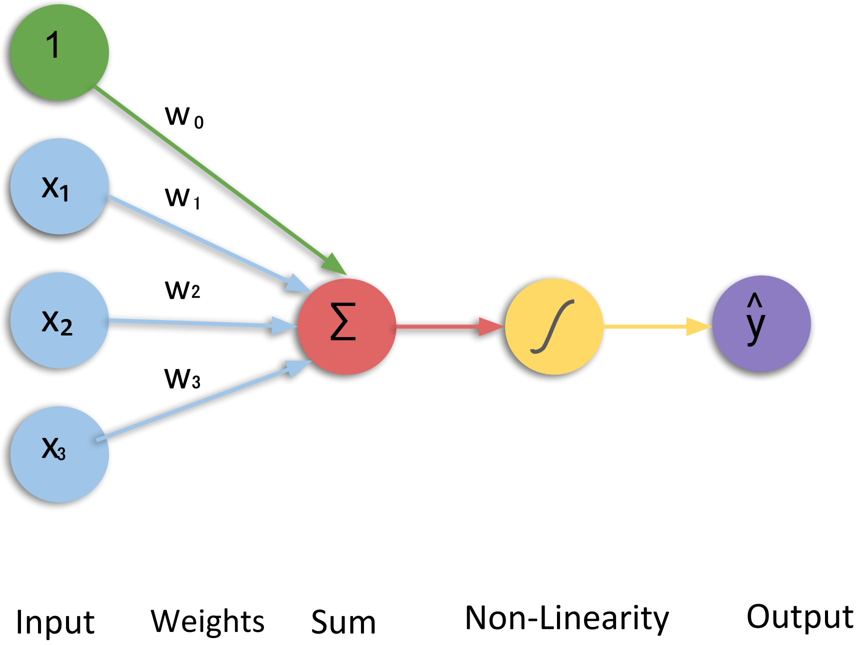


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$



```
package deep_learning_for_gophers

type Neuron struct {
    Act                 ActivationFunc
    Input               []*Edge
    Output              []*Edge
    Value               float64
}

type Edge struct {
    Weight              float64
    Input               float64
    Output              float64
    IsBias              bool
}
```

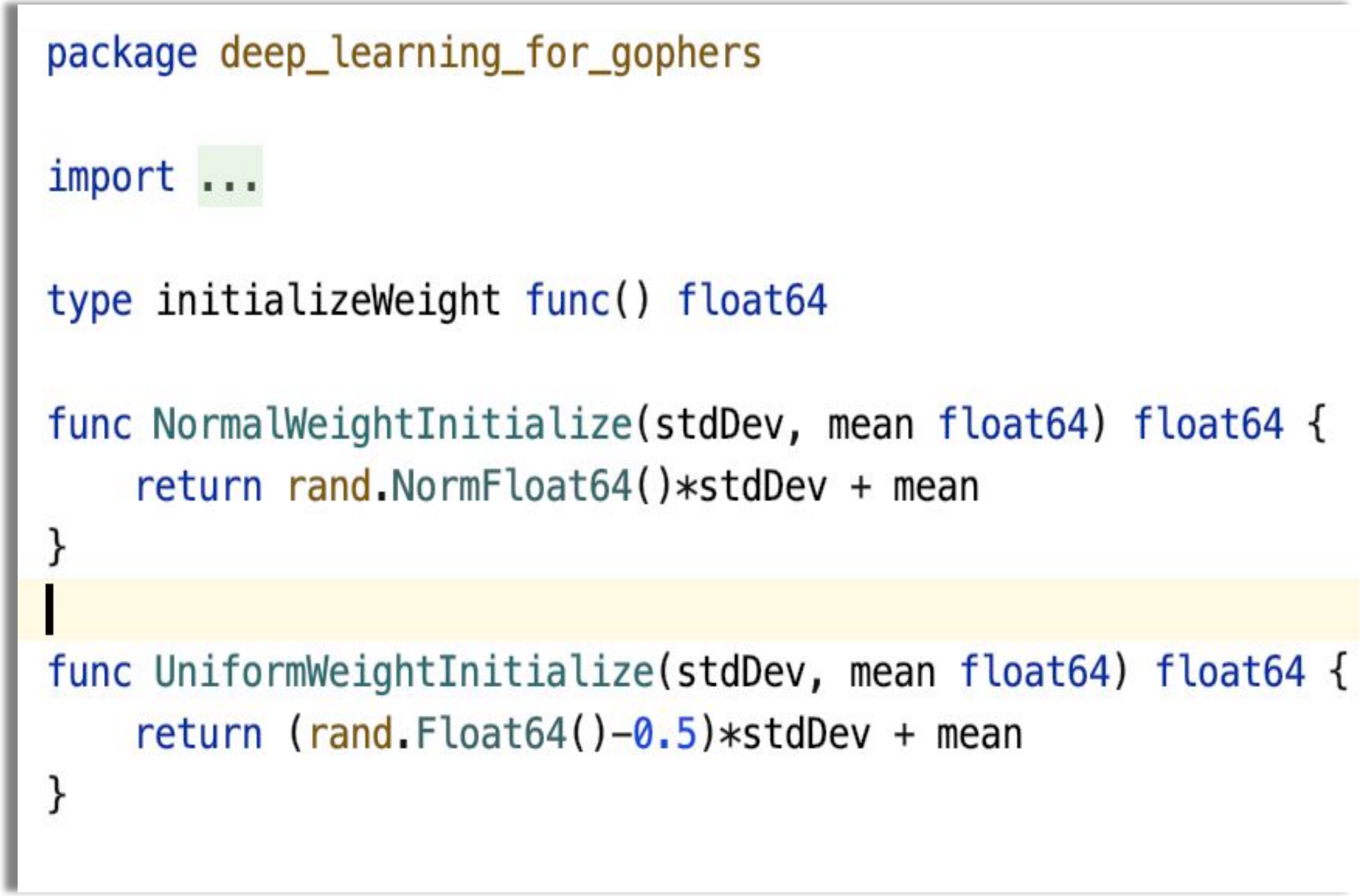
The Perceptron : Feedforward Propagation



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

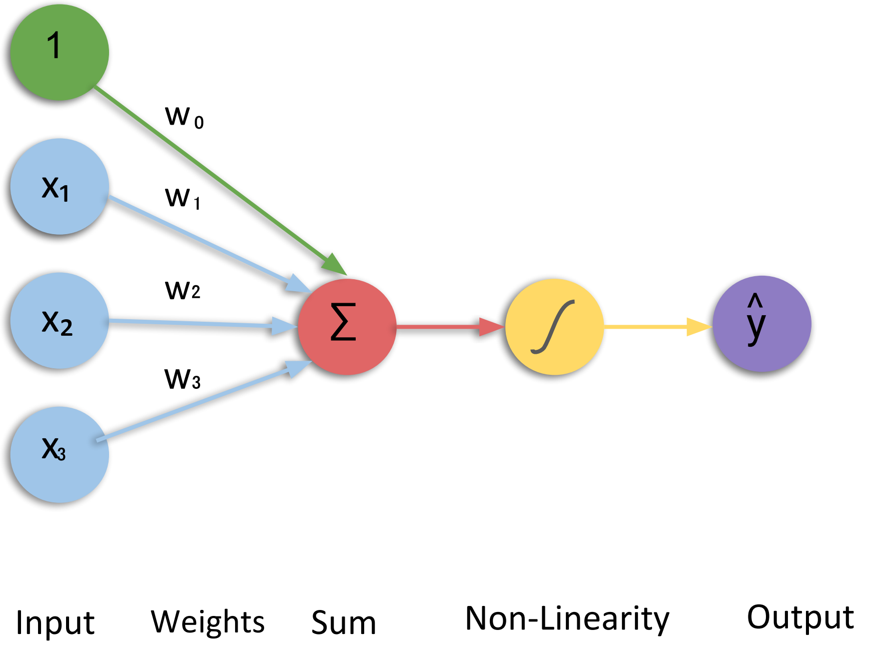
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

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



```go
package deep_learning_for_gophers

import ...

type initializeWeight func() float64

func NormalWeightInitialize(stdDev, mean float64) float64 {
	return rand.NormFloat64()*stdDev + mean
}

func UniformWeightInitialize(stdDev, mean float64) float64 {
	return (rand.Float64()-0.5)*stdDev + mean
}
```

The Perceptron : Feedforward Propagation

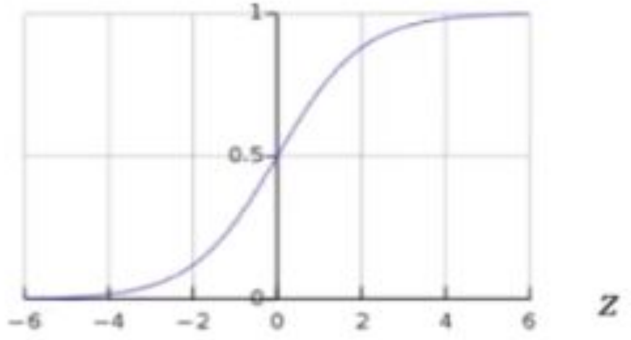


Activation Function

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

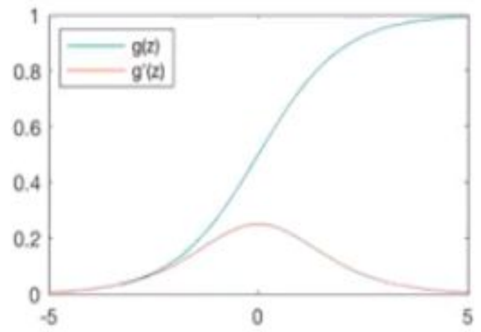
- Example : Sigmoid function

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



Common Activation Functions

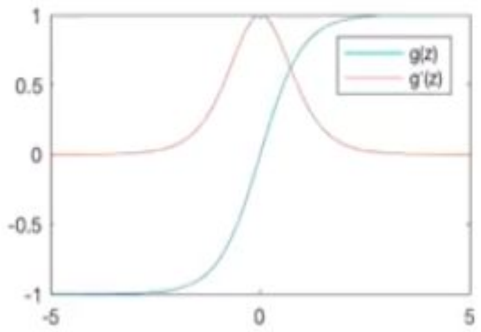
Sigmoid Function



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

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

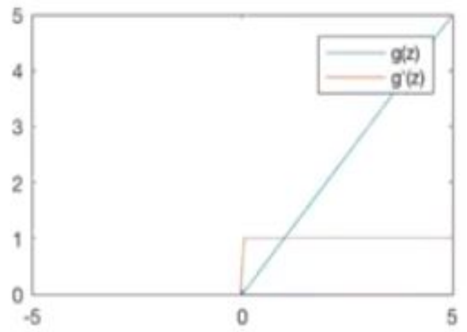
Hyperbolic Tangent



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

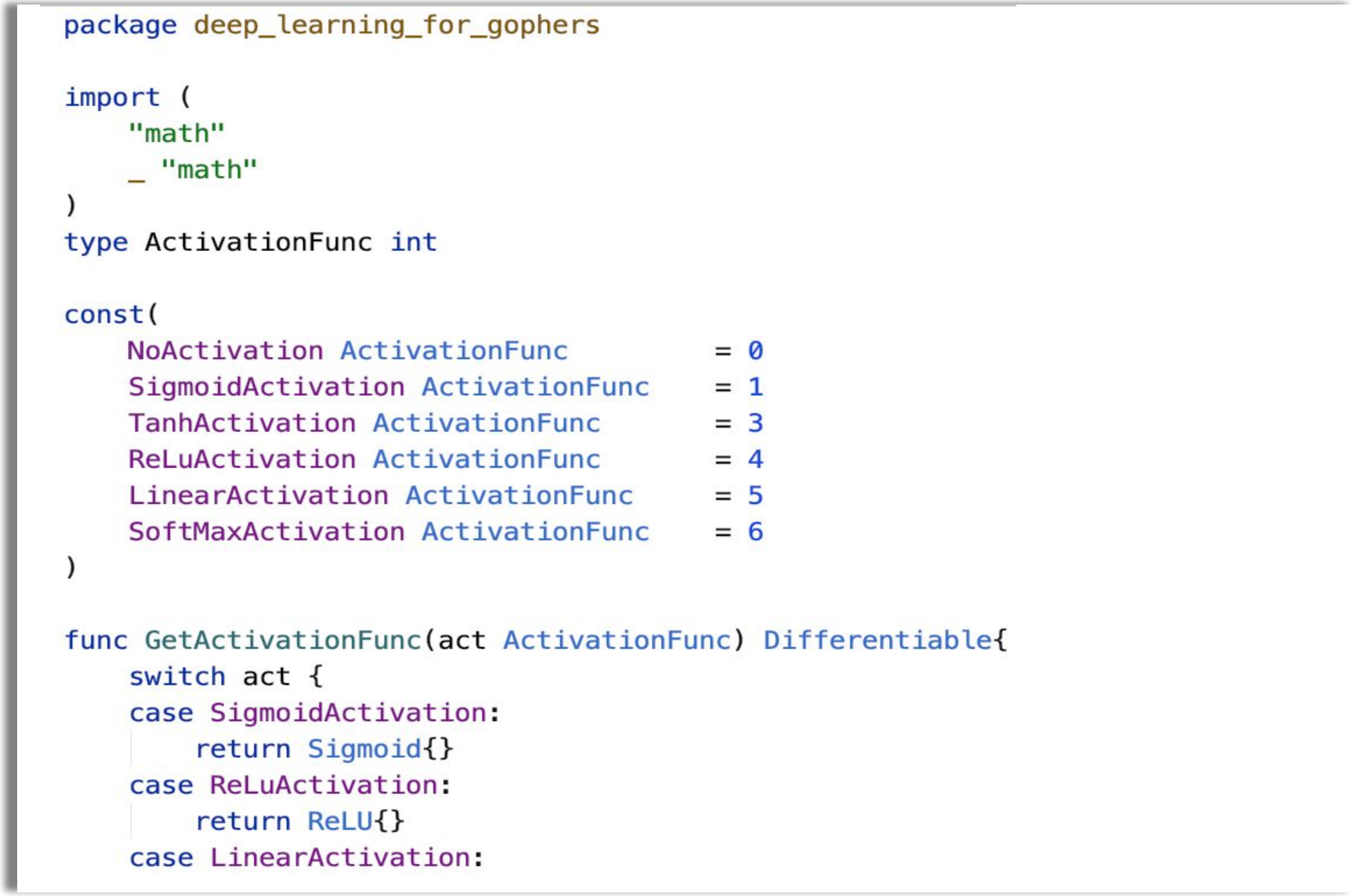
$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

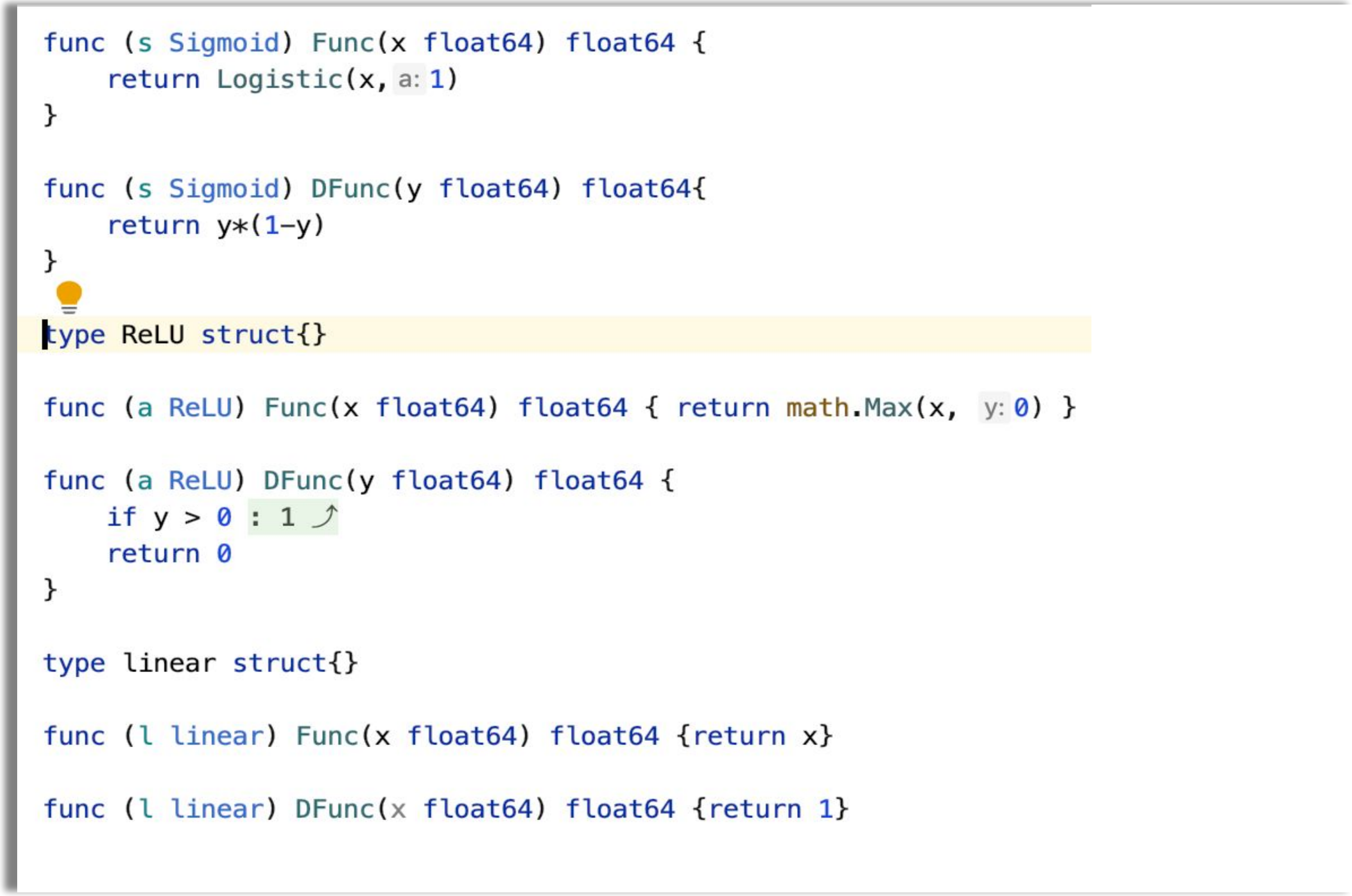


$$g(z) = \max(0, z)$$

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



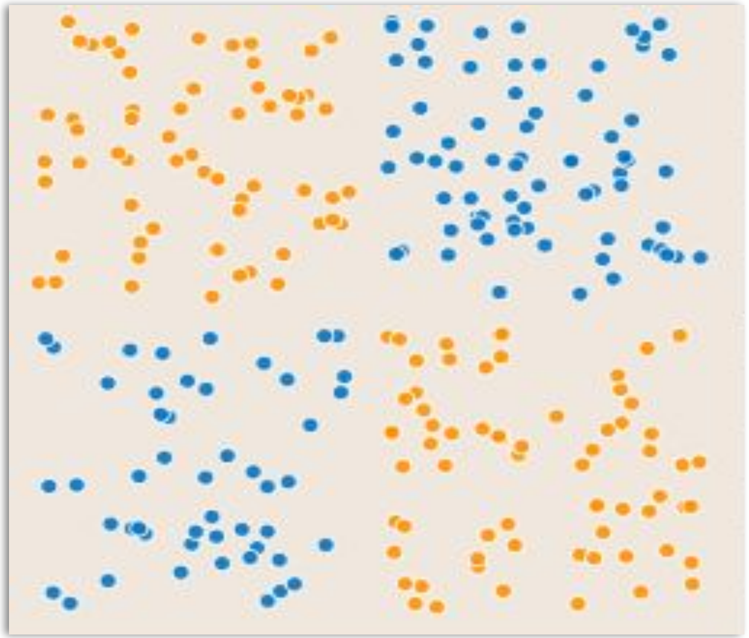
```go
package deep_learning_for_gophers

import (
    "math"
    _ "math"
)
type ActivationFunc int

const(
    NoActivation ActivationFunc         = 0
    SigmoidActivation ActivationFunc    = 1
    TanhActivation ActivationFunc       = 3
    ReLuActivation ActivationFunc       = 4
    LinearActivation ActivationFunc     = 5
    SoftMaxActivation ActivationFunc    = 6
)

func GetActivationFunc(act ActivationFunc) Differentiable{
    switch act {
    case SigmoidActivation:
        return Sigmoid{}
    case ReLuActivation:
        return ReLU{}
    case LinearActivation:
```



```go
func (s Sigmoid) Func(x float64) float64 {
    return Logistic(x, a: 1)
}

func (s Sigmoid) DFunc(y float64) float64{
    return y*(1-y)
}

type ReLU struct{}

func (a ReLU) Func(x float64) float64 { return math.Max(x,  y: 0) }

func (a ReLU) DFunc(y float64) float64 {
    if y > 0 : 1 ↗
    return 0
}

type linear struct{}

func (l linear) Func(x float64) float64 {return x}

func (l linear) DFunc(x float64) float64 {return 1}
```

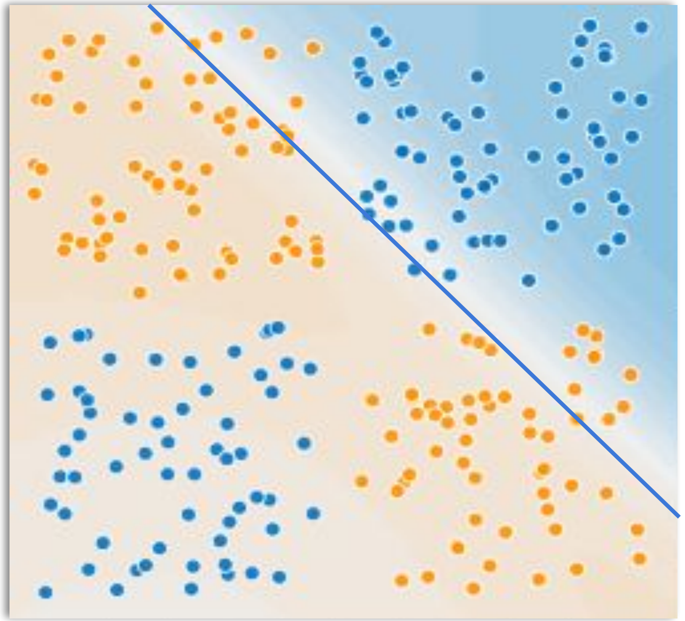
Importance of Activation Functions

The purpose of activation functions is to **introduce non-linearities** into the network

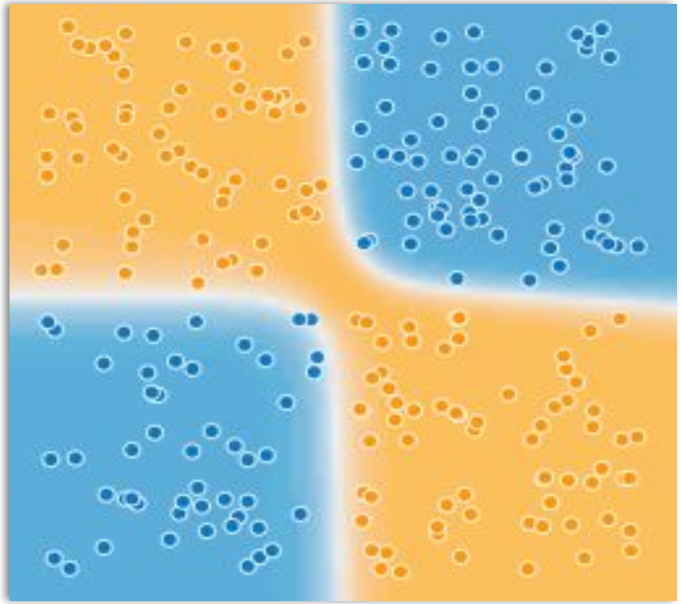


What is we want to build a Neural Network to distinguish between blue vs orange points?

Importance of Activation Functions

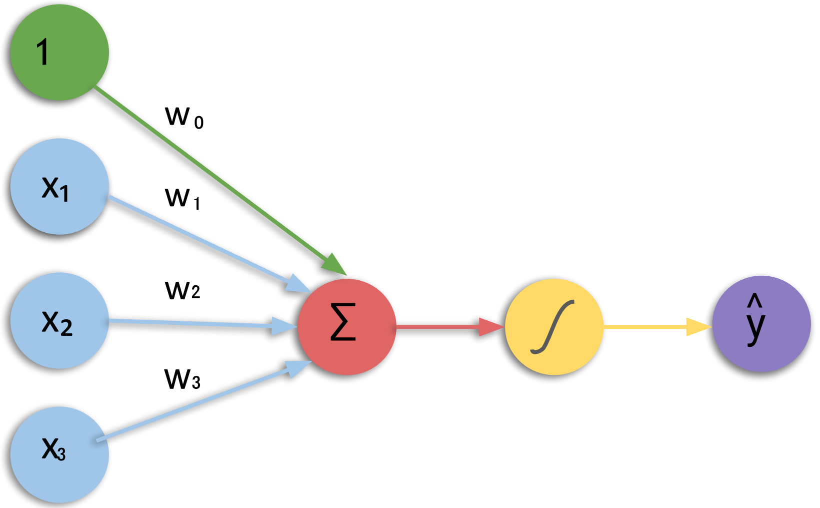


Linear activations produce linear decisions no matter what the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron : Example

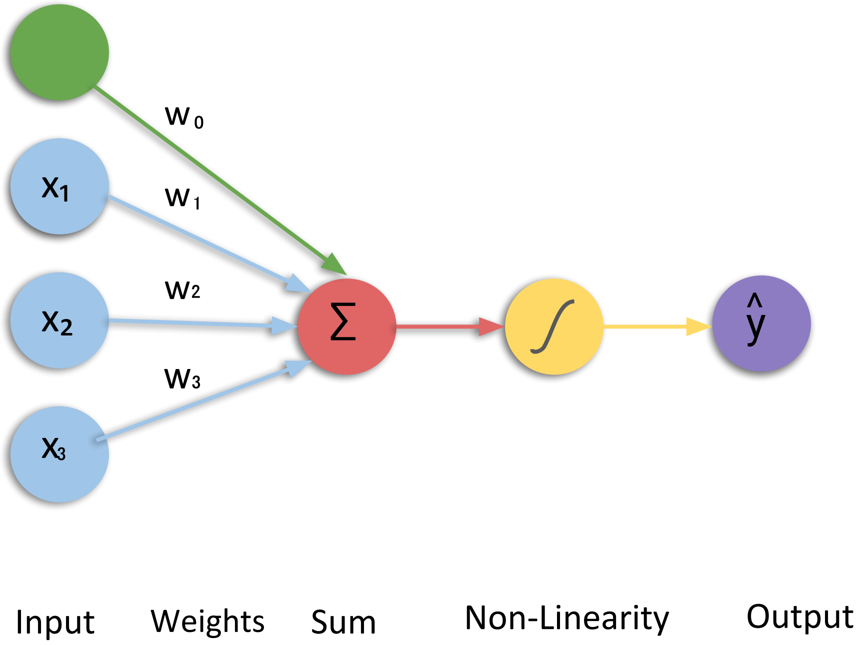


Input Weights Sum Non-Linearity Output

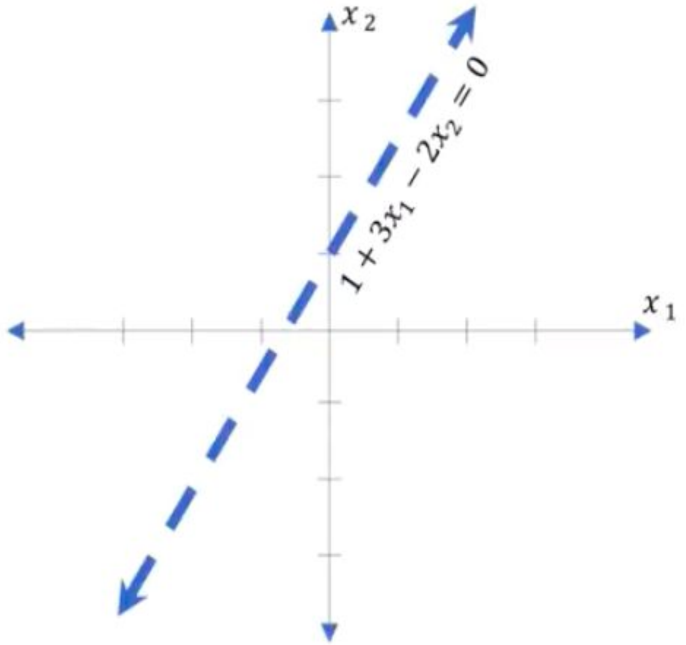
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

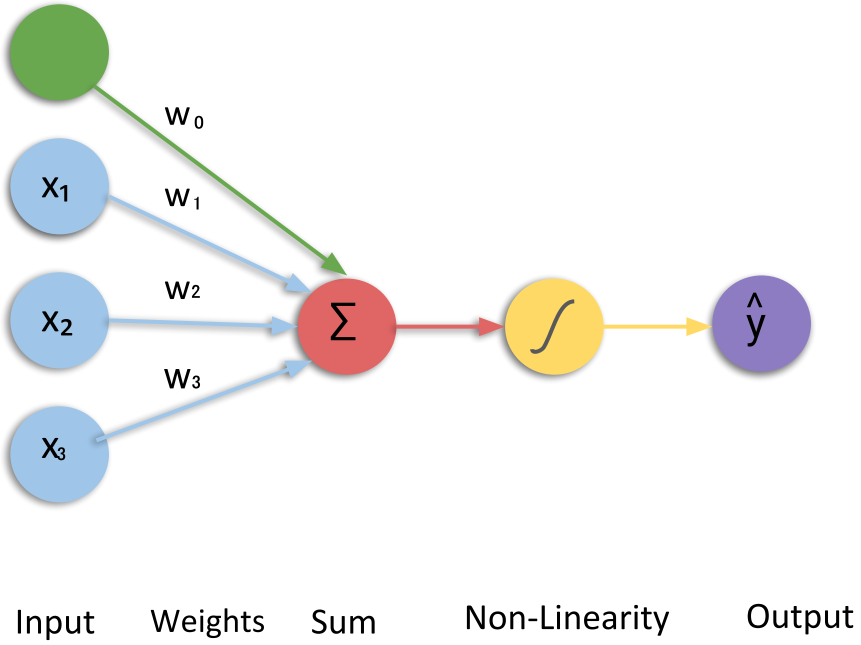
The Perceptron : Example



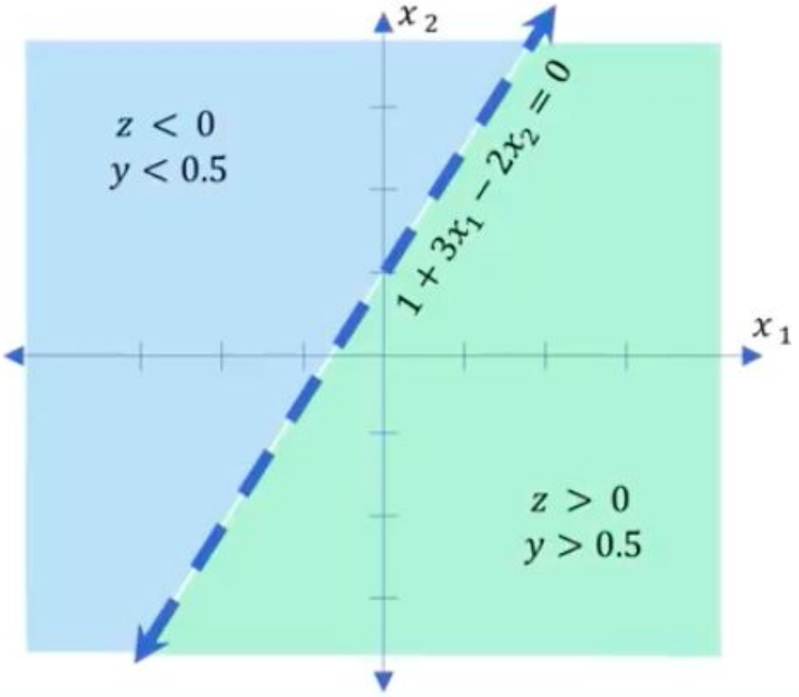
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron : Example

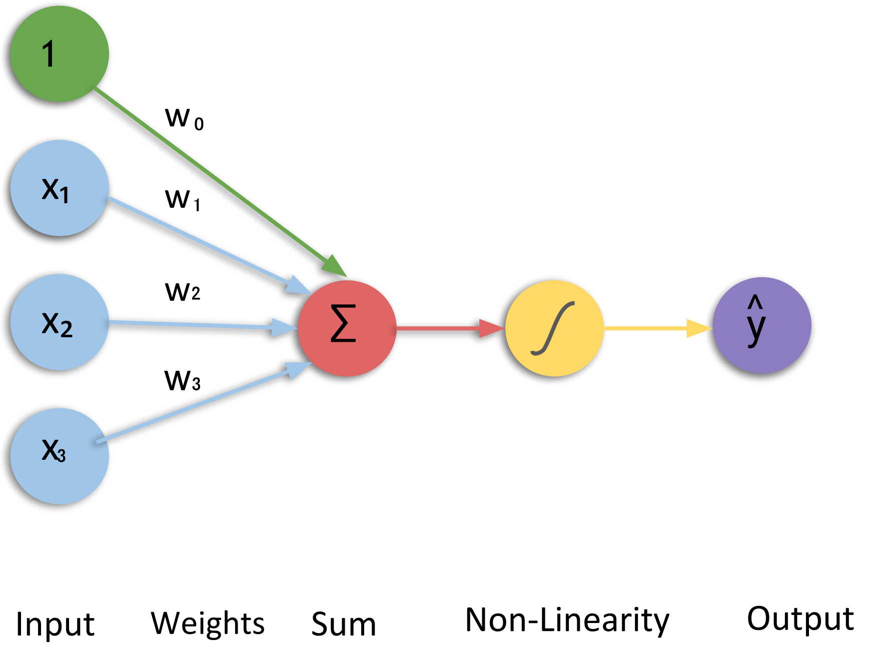


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

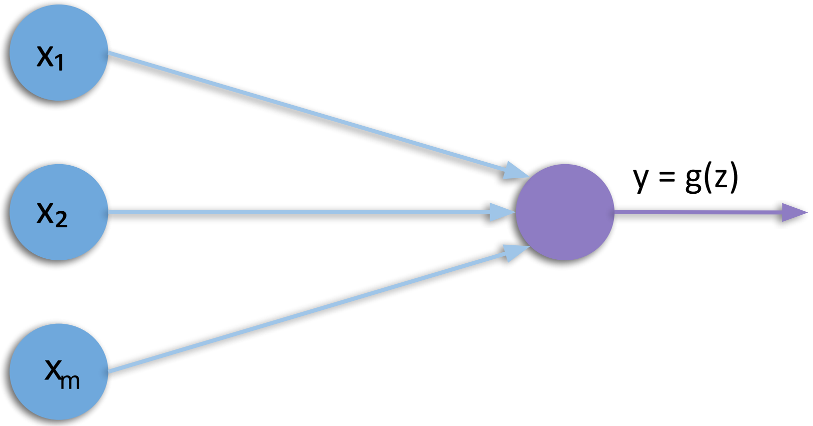


Building Neural Networks with Perceptrons

The Perceptron : Simplified

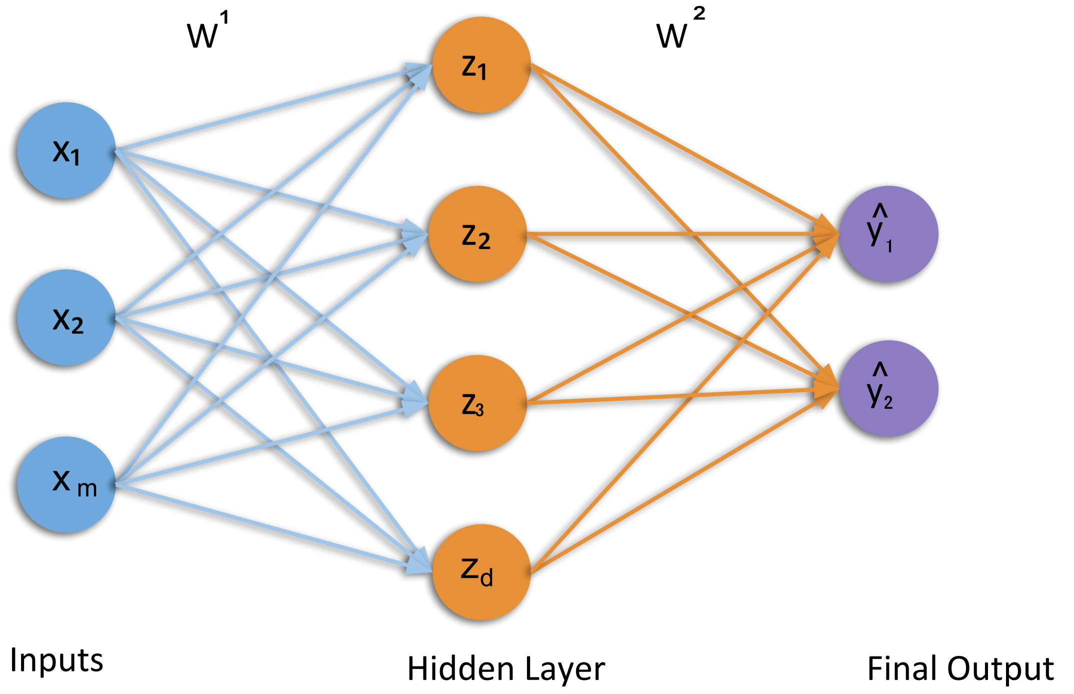


The Perceptron : Simplified



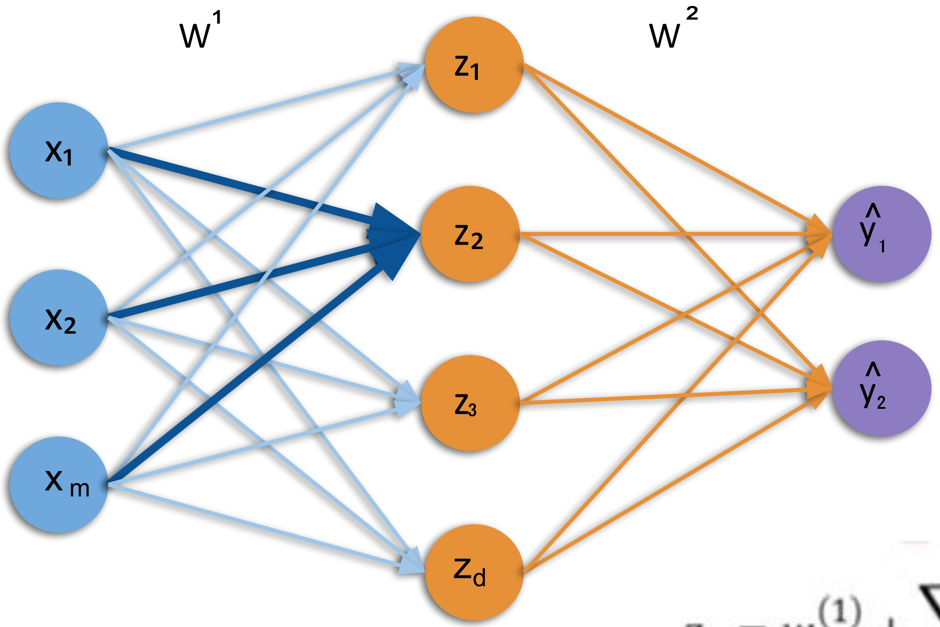
$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$

Single Layer Neural Network



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

Single Layer Neural Network



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

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

Applying Neural Networks

Example Problem

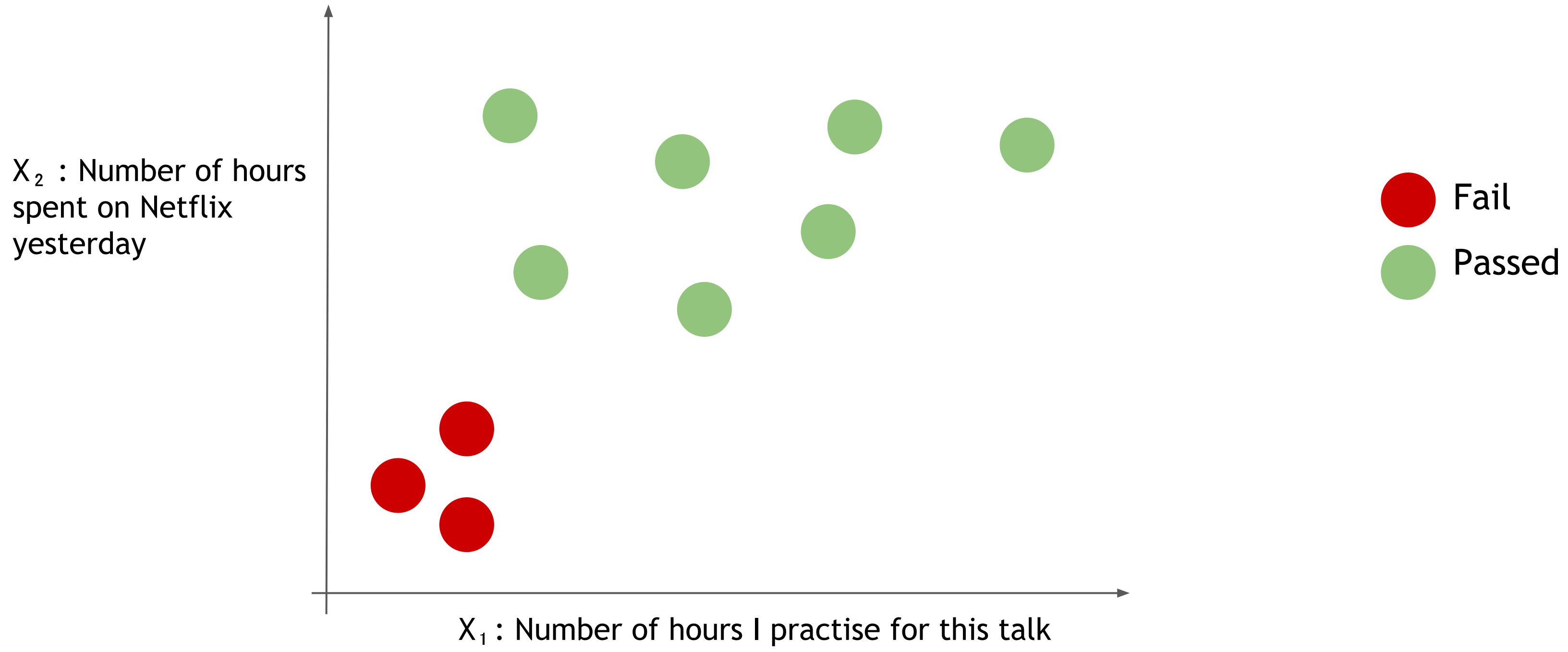
Will I be able to do justice to this topic?

Let’s start with simple two feature model

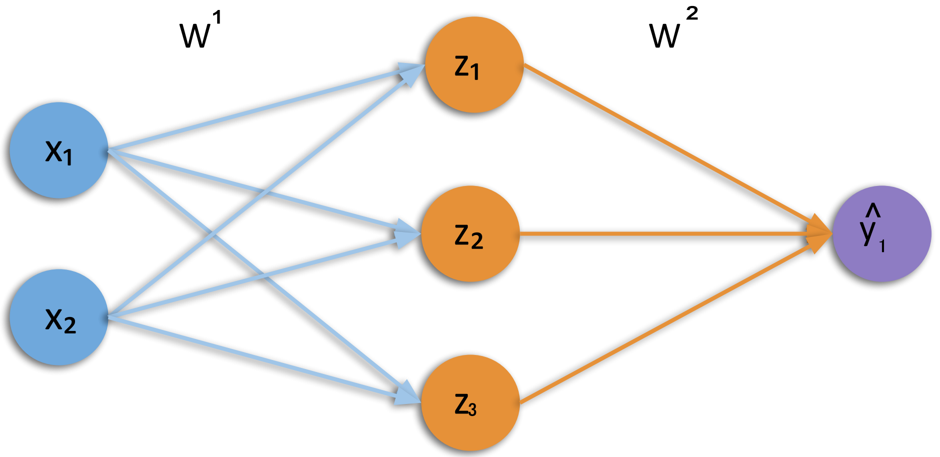
X1 : Number of hours I practise for this talk

X2 : Number of hours spent on Netflix yesterday

Will I be able to do justice to this topic?

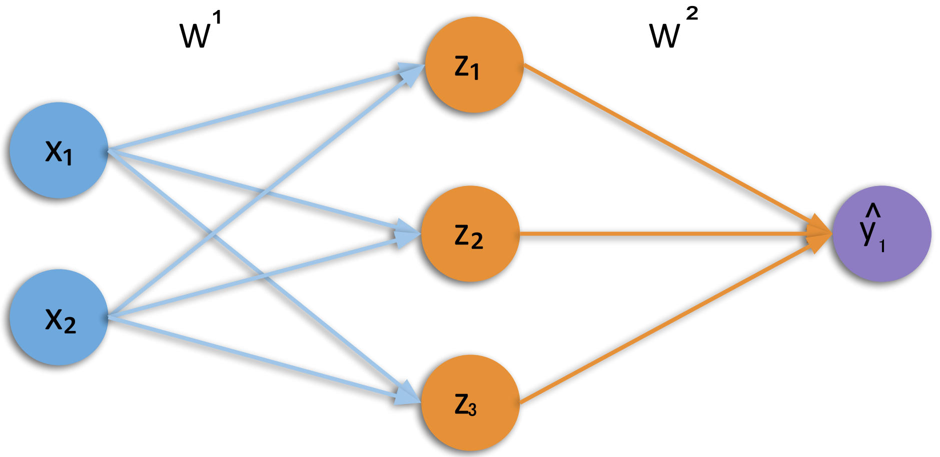


Single Layer Neural Network



Predicted : 0.1
Actual : 1

Quantifying Loss



Predicted : 0.1
Actual : 1

Training Neural Networks

Loss Optimization

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

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

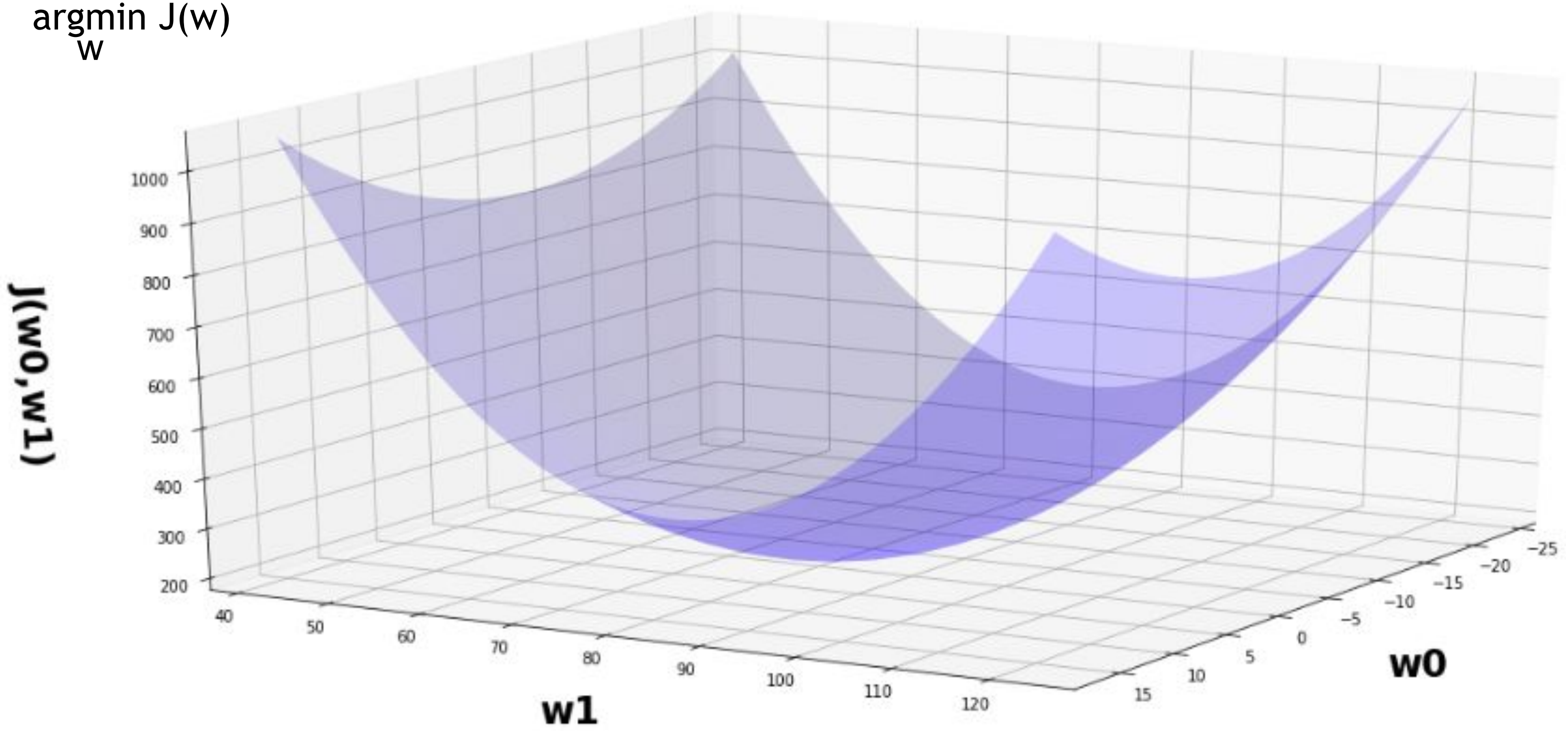
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

Remember:

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

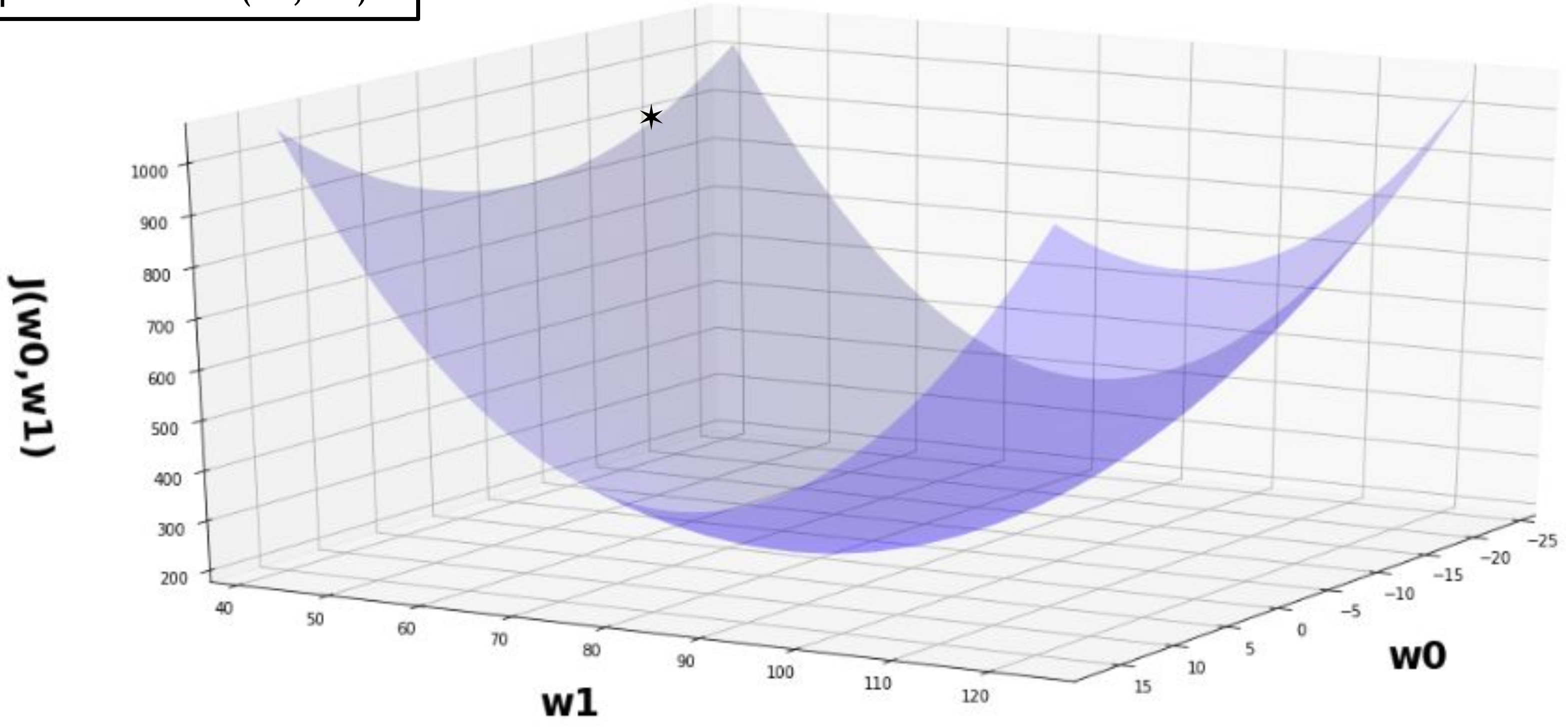
Loss Optimization

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



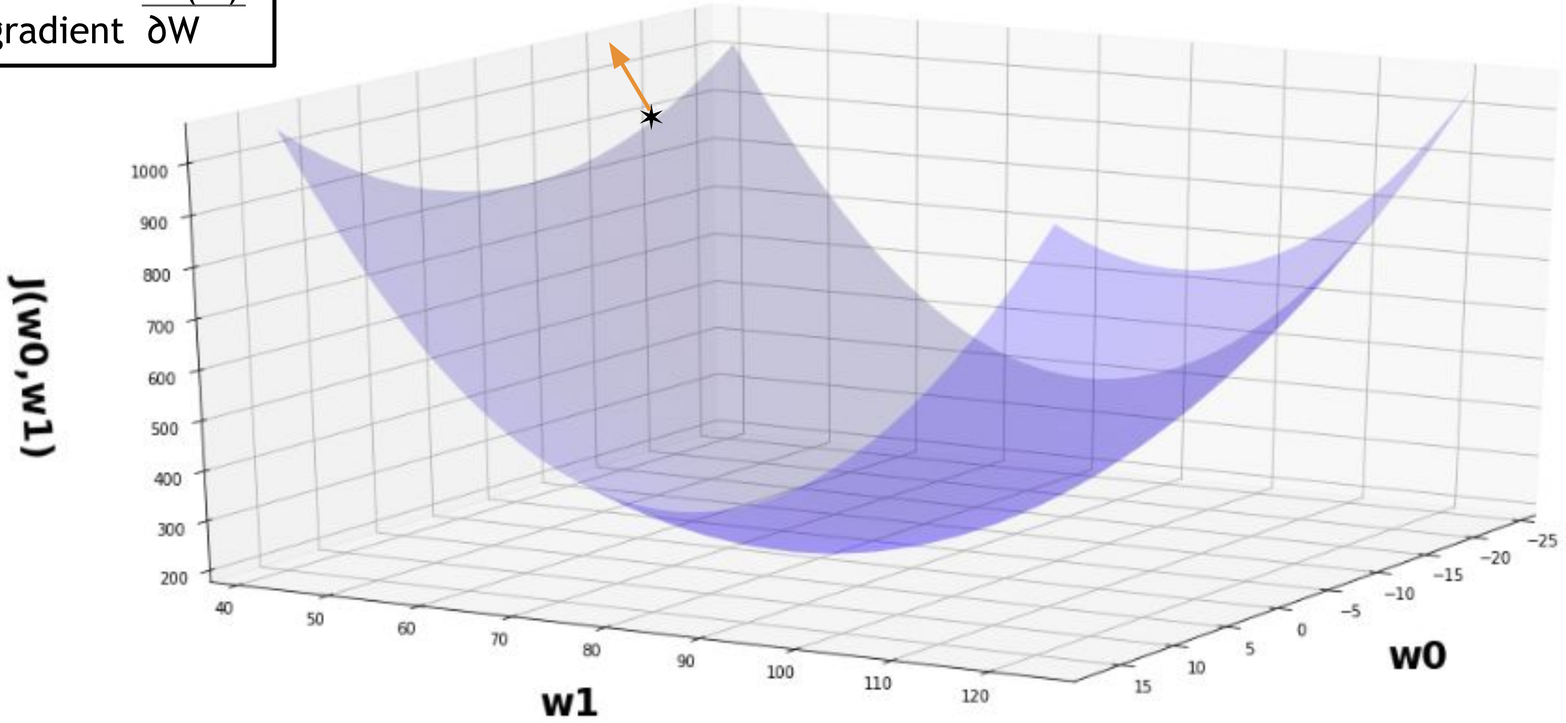
Loss Optimization

Randomly pick an initial (w_1, w_0)



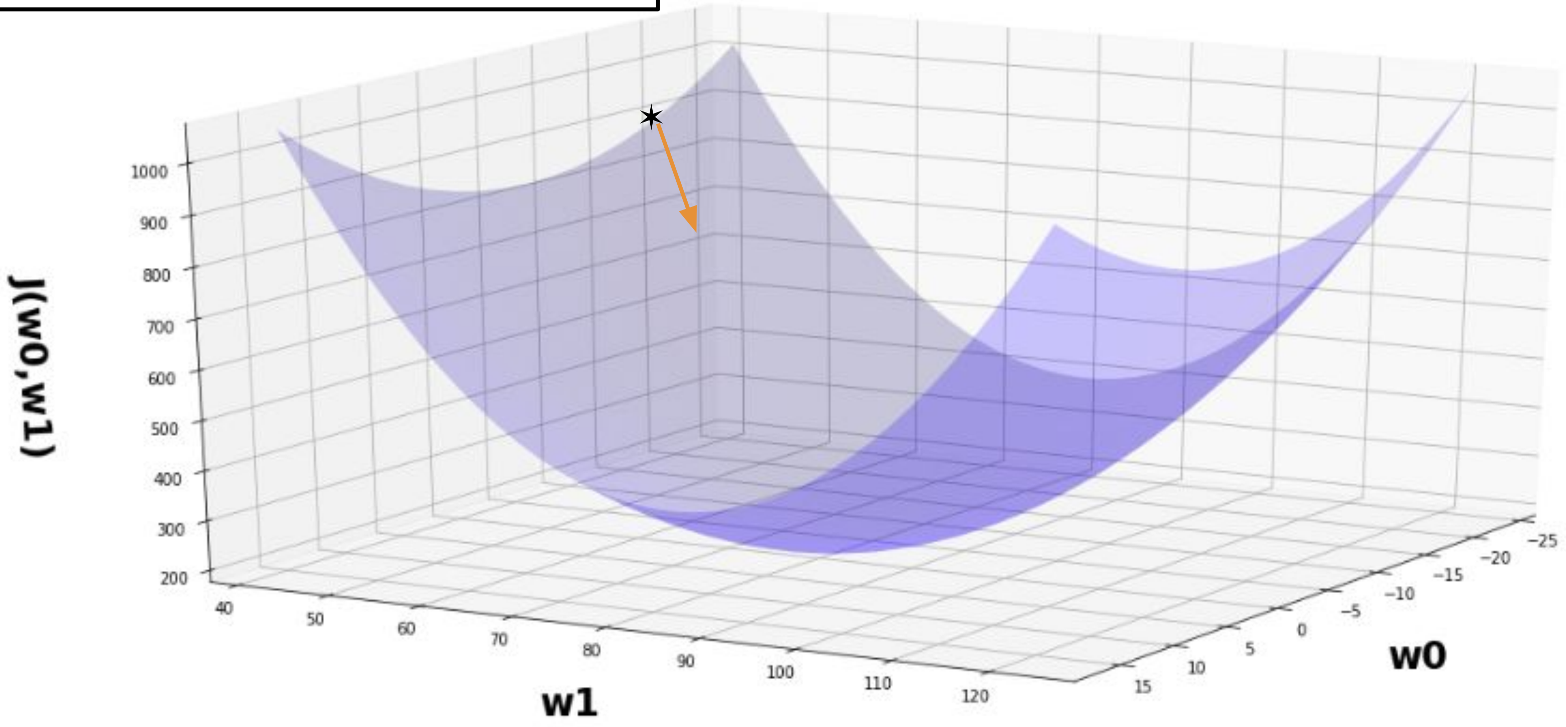
Loss Optimization

Compute gradient $\frac{\partial J(\mathbf{w})}{\partial \mathbf{W}}$



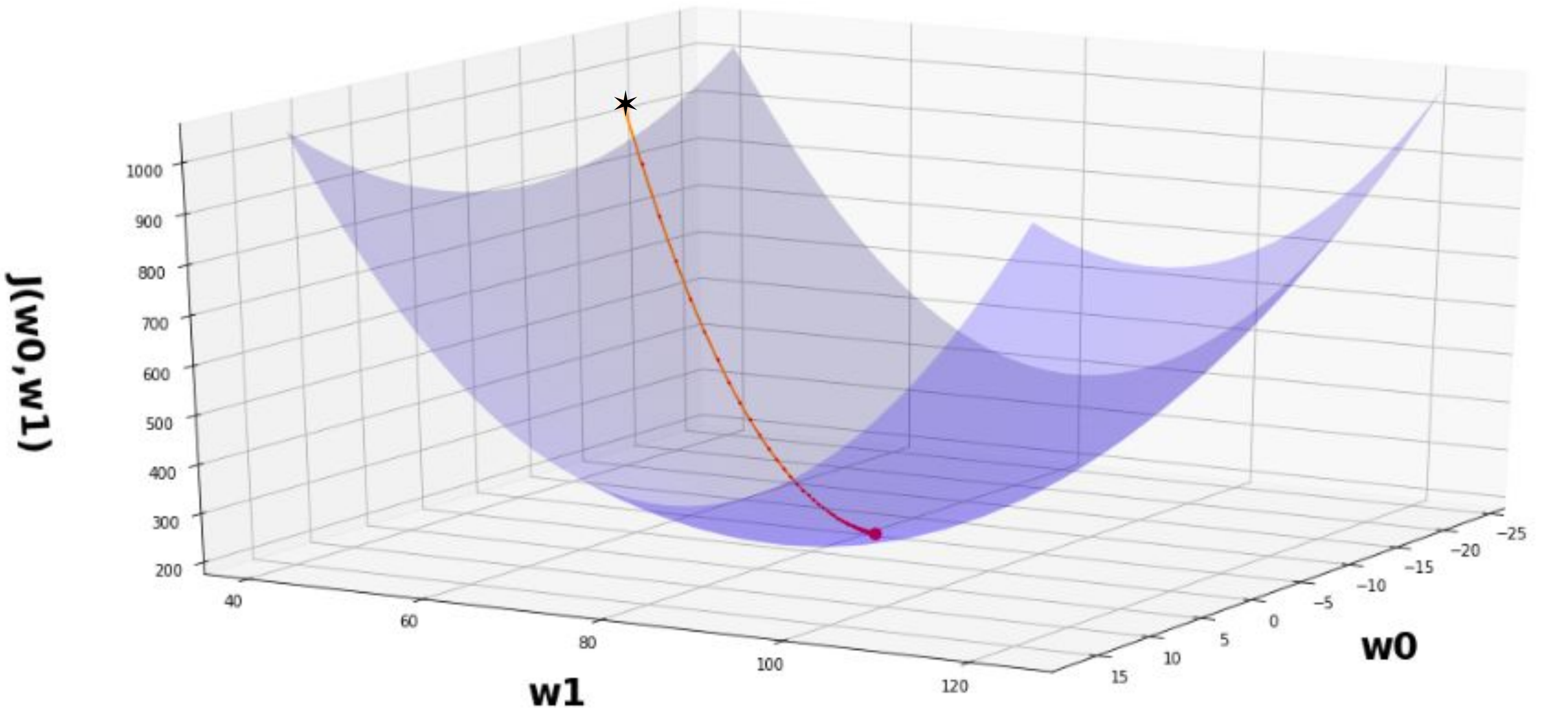
Loss Optimization

Take direction in opposite direction of gradient



Gradient Descent

Repeat until convergence

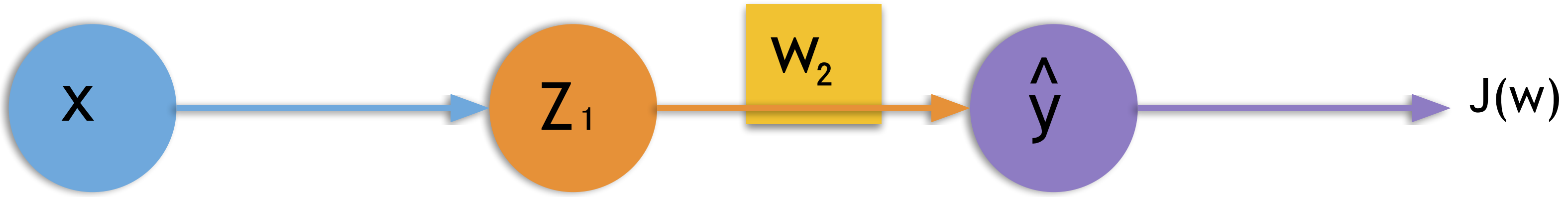


Gradient Descent

Algorithm

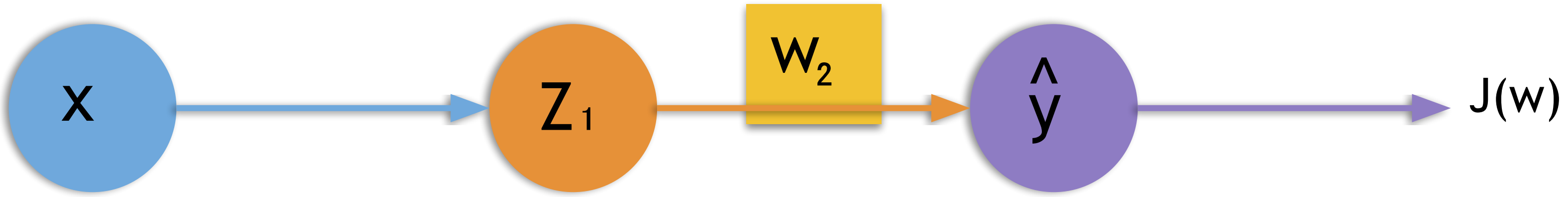
1. Initialize weights randomly
2. Loop until convergence
3. Compute gradient $\boxed{\frac{\partial J(w)}{\partial W}}$
4. Update weights $W \leftarrow W - \text{��} \frac{\partial J(w)}{\partial W}$
5. Return weights

Computing Gradients : Backpropagation



How does small change in one weight (eg W_2) affect the final loss **J(W)**?

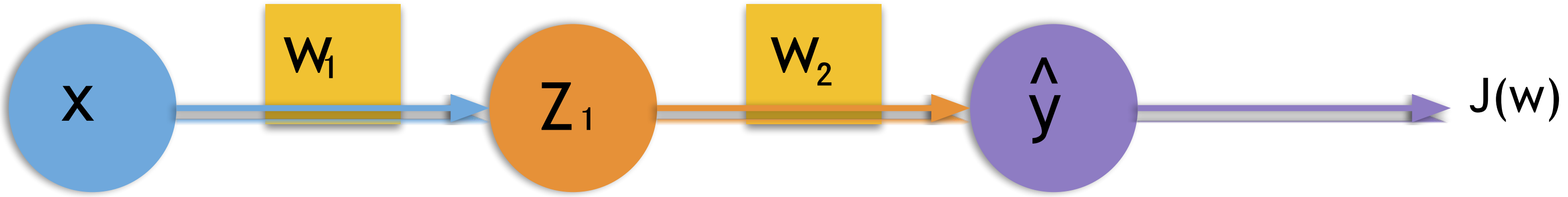
Computing Gradients : Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2}$$

Using chain rule

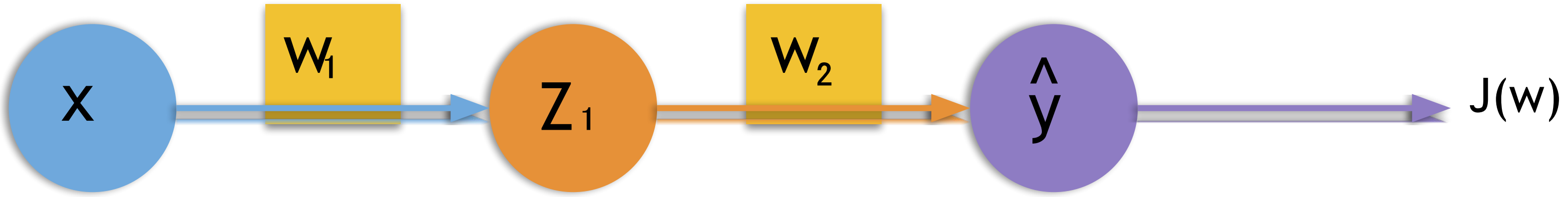
Computing Gradients : Backpropagation



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

Using chain rule

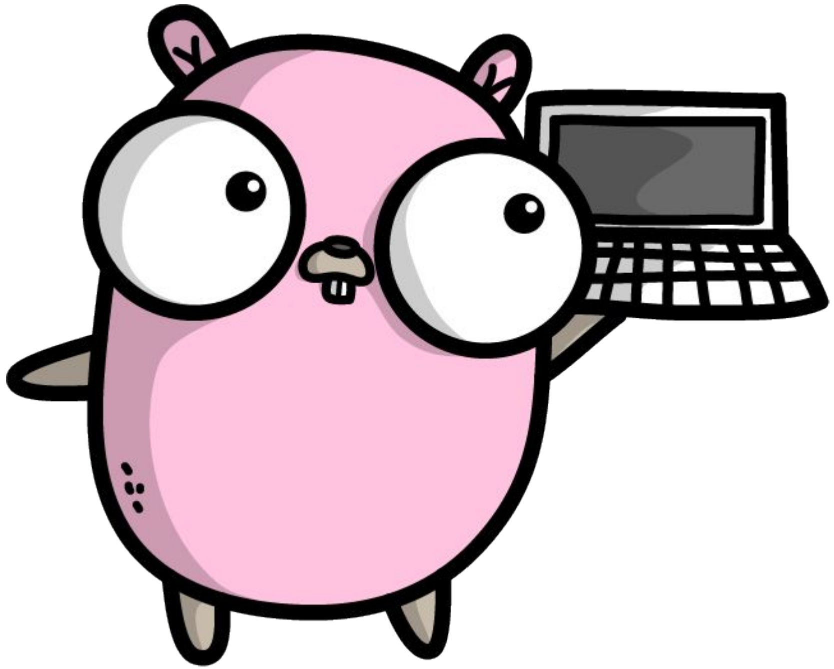
Computing Gradients : Backpropagation

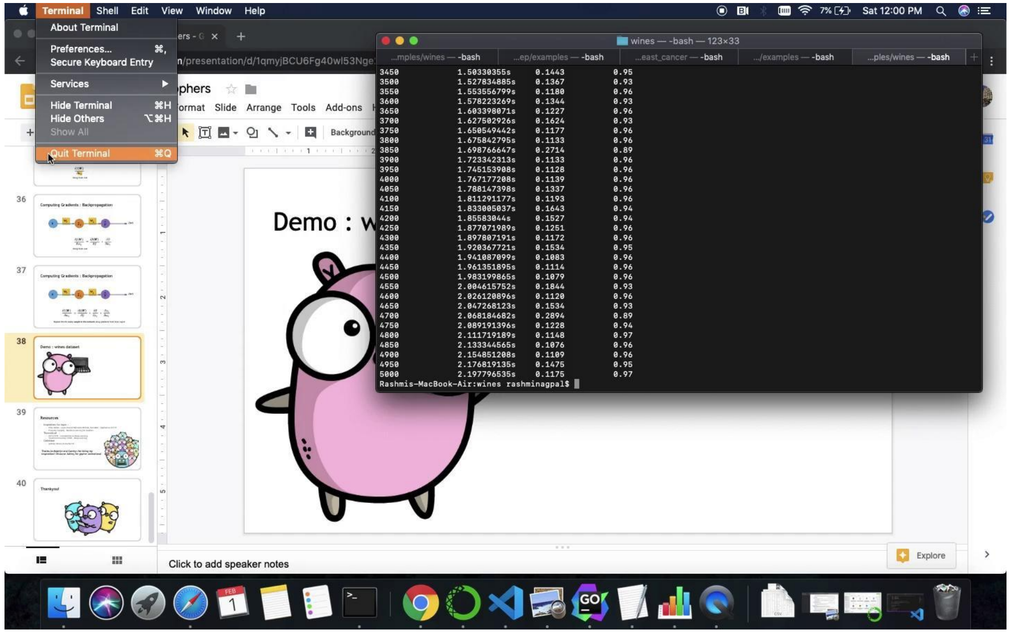


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

Demo



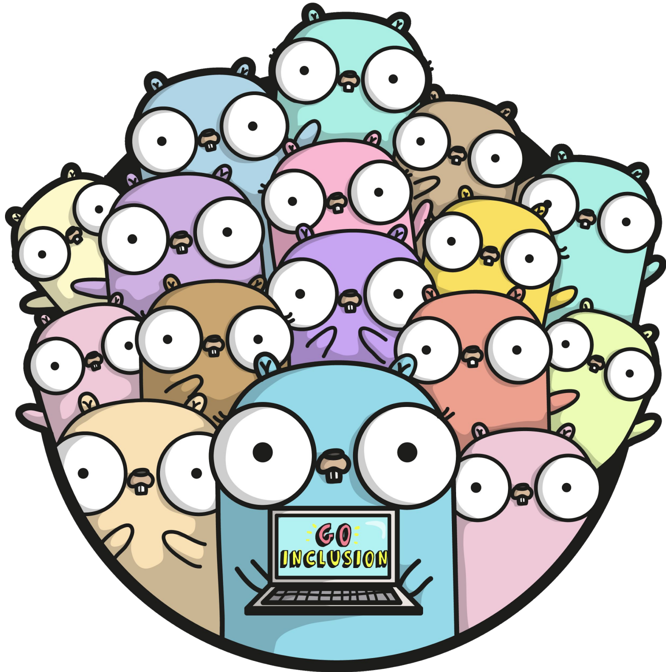


```
3450          1.50330355s    0.1443       0.95
3500          1.527834885s   0.1367       0.93
3550          1.553556799s   0.1180       0.96
3600          1.578223269s   0.1344       0.93
3650          1.603398071s   0.1227       0.96
3700          1.627502926s   0.1624       0.93
3750          1.650549442s   0.1177       0.96
3800          1.675842795s   0.1133       0.96
3850          1.698766647s   0.2714       0.89
3900          1.723342313s   0.1133       0.96
3950          1.745153908s   0.1128       0.96
4000          1.767177208s   0.1139       0.96
4050          1.788147398s   0.1337       0.96
4100          1.811291177s   0.1193       0.96
4150          1.833005037s   0.1643       0.94
4200          1.85583044s    0.1527       0.94
4250          1.877071989s   0.1251       0.96
4300          1.897807191s   0.1172       0.96
4350          1.920367721s   0.1534       0.95
4400          1.941087099s   0.1083       0.96
4450          1.961351895s   0.1114       0.96
4500          1.983199865s   0.1079       0.96
4550          2.004615752s   0.1844       0.93
4600          2.026120896s   0.1120       0.96
4650          2.047268123s   0.1534       0.93
4700          2.068184682s   0.2894       0.89
4750          2.089191396s   0.1228       0.94
4800          2.111719189s   0.1148       0.97
4850          2.133344565s   0.1076       0.96
4900          2.154851208s   0.1109       0.96
4950          2.176819135s   0.1475       0.95
5000          2.197796535s   0.1175       0.97
Rashmis-MacBook-Air:wines rashminagpal$
```

Resources

- Inspiration for topic :
  - Ellen Kobes - Learn Neural Networks WithGo, Not Math : GopherCon EU'19
  - Francesc Campoy - Machine Learning for Gophers
- Theoretical
  - MIT 6.S191 - Introduction to Deep Learning
  - Stanford University CS230 - Deep Learning
- Codebase
  - godeep library as starter kit

Thanks to Maartje and Carolyn for being my inspiration! Ofcourse Ashley,for gopher animations!



Thankyou!