



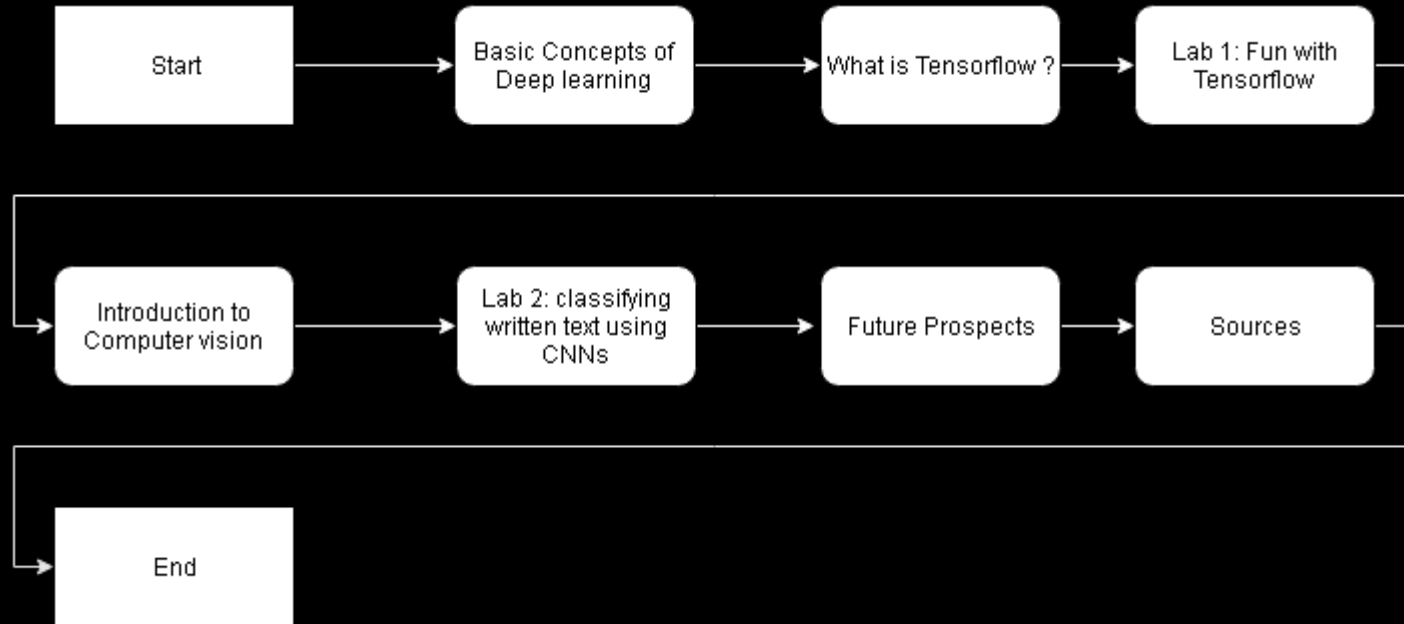
Introduction to Tensorflow

Falko Reinert

12.06.2019

Preparation

Map



Deep Learning fundamentals

The Rise of Deep Learning

“In the first decades of the 21st century, access to large amounts of data (known as big data), faster computers and advanced machine learning techniques were successfully applied to many problems throughout the economy. In fact, McKinsey Global Institute estimated in their famous paper "Big data: The next frontier for innovation, competition, and productivity" that "by 2009, nearly all sectors in the US economy had at least an average of 200 terabytes of stored data".

By 2016, the market for AI-related products, hardware, and software reached more than 8 billion dollars, and the New York Times reported that interest in AI had reached a "frenzy". The applications of big data began to reach into other fields as well, such as training models in ecology¹ and for various applications in economics. Advances in deep learning (particularly deep convolutional neural networks and recurrent neural networks) drove progress and research in image and video processing, text analysis, and even speech recognition.” - Wikipedia

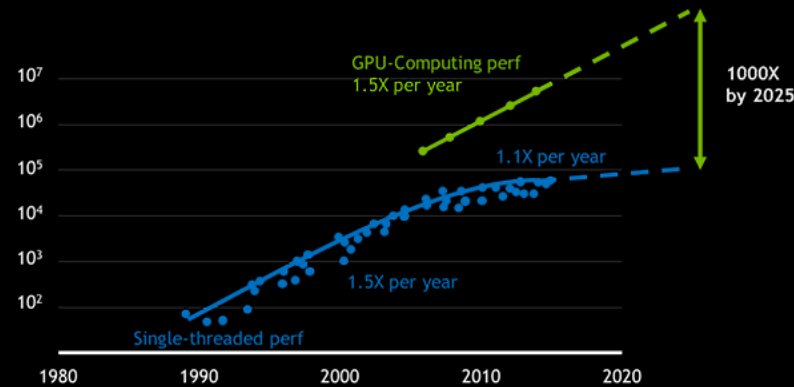
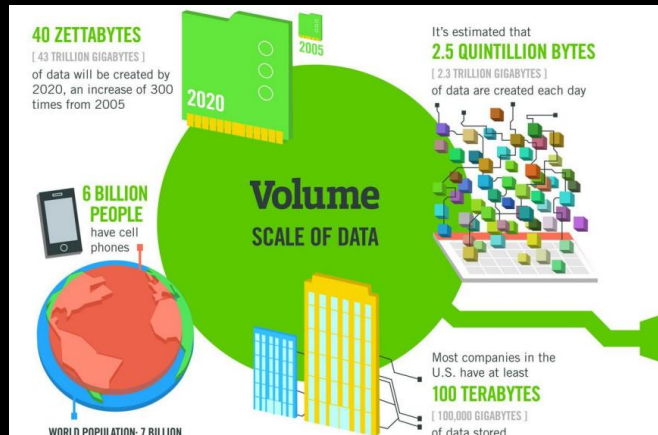
Summing it Up

Eventhough Neural Networks date back decades, they only started becoming relevant recently because of:

1. Big Data

2. Hardware

3. Software

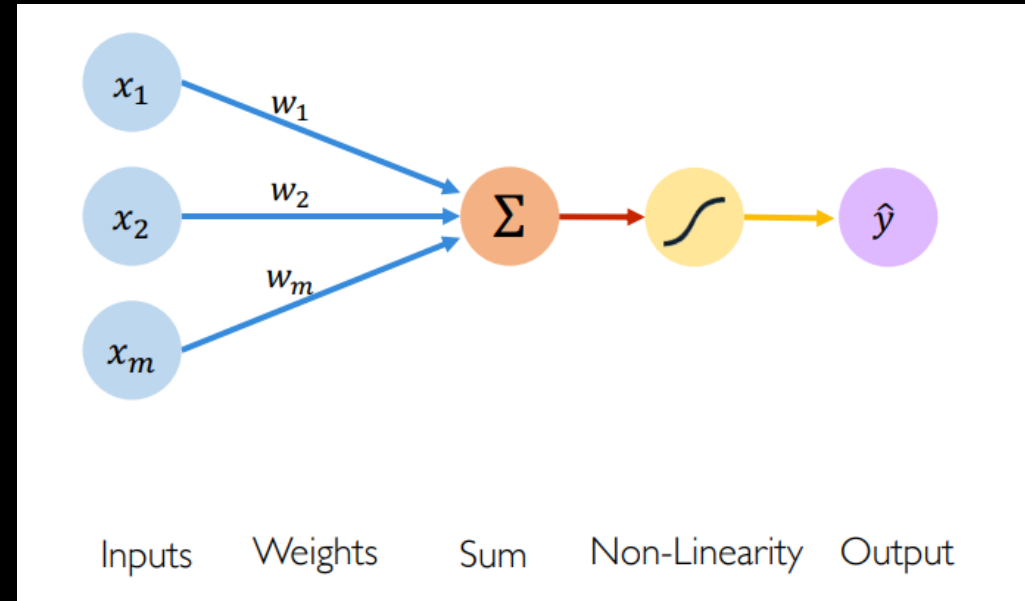


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Ghukatur, L. Hammond, and C. Batten. New plot and data collected for 2010-2015 by K. Rupp.

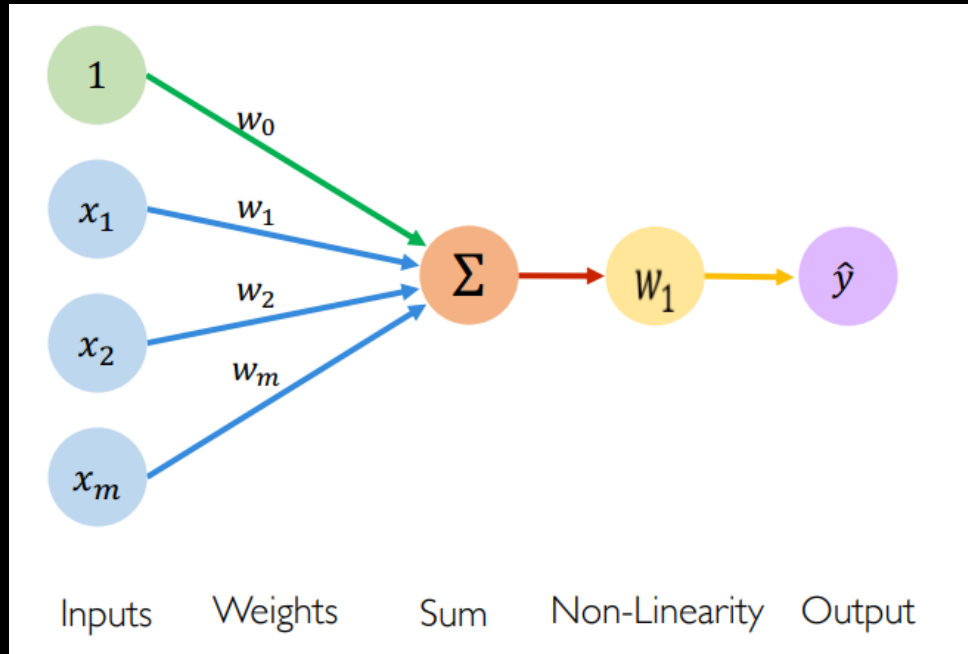


The Perceptron: the foundation of deep learning

- invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt
- First mentioned: “The perceptron: A probabilistic model for information storage and organization in the brain” by Rosenblatt, F



The Perceptron: Forward Propagation



Product of Inputs and weights

$$\tilde{y} = g(w_o + \sum_{i=1}^m x_i * w_i)$$

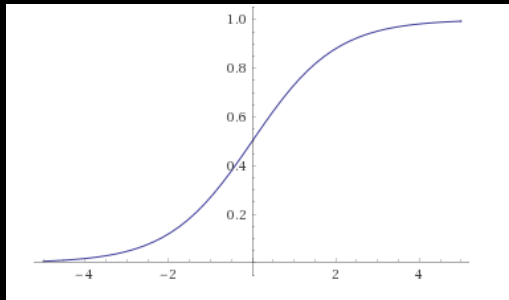
Output

Non-linear activation function

$$\tilde{y} = g(w_o + X^T * W)$$


Common Activation Functions

Sigmoid Function

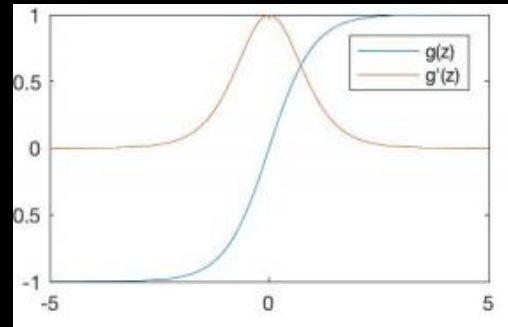


$$f(z) = \frac{L}{1 + e^{-k(x - x_0)}}$$

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

 `tf.nn.sigmoid(z)`

Hyperbolic Tangent

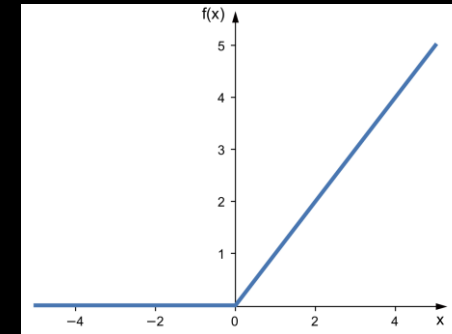


$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$F'(z) = 1 - f(z)^2$$


 `tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



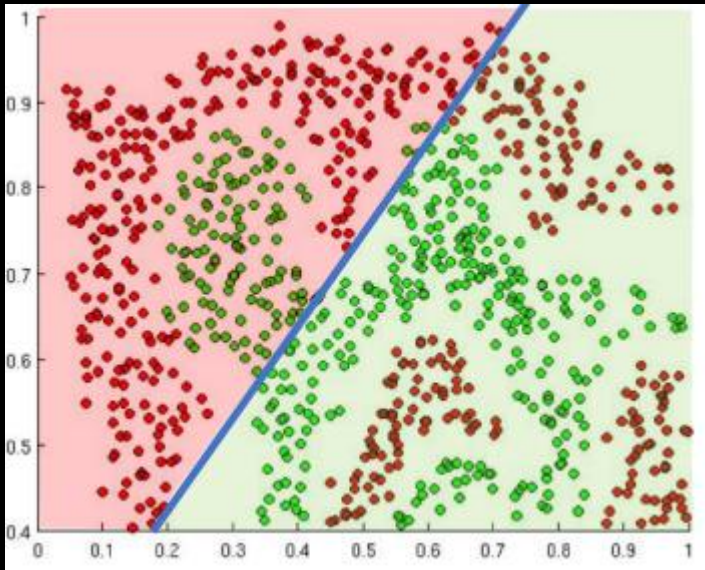
$$f(z) = \max(0, z)$$

$$f(z) = 1 \text{ if } z > 0 \text{ or } 0 \text{ if } z < 0$$

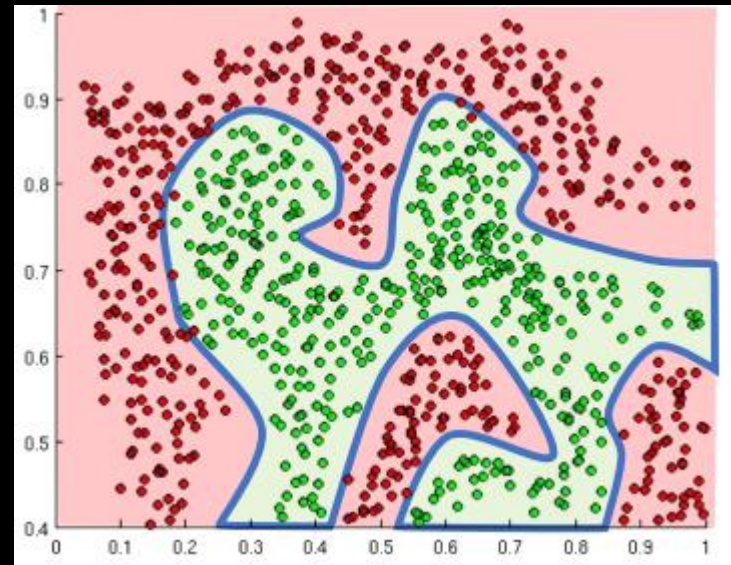
 `tf.nn.relu(z)`

Activation Functions: a consideration

Imagine we wanted to build a Neural Network to distinguish green vs red points. (Real Life Example: time vs. score plot for students)



linear approach



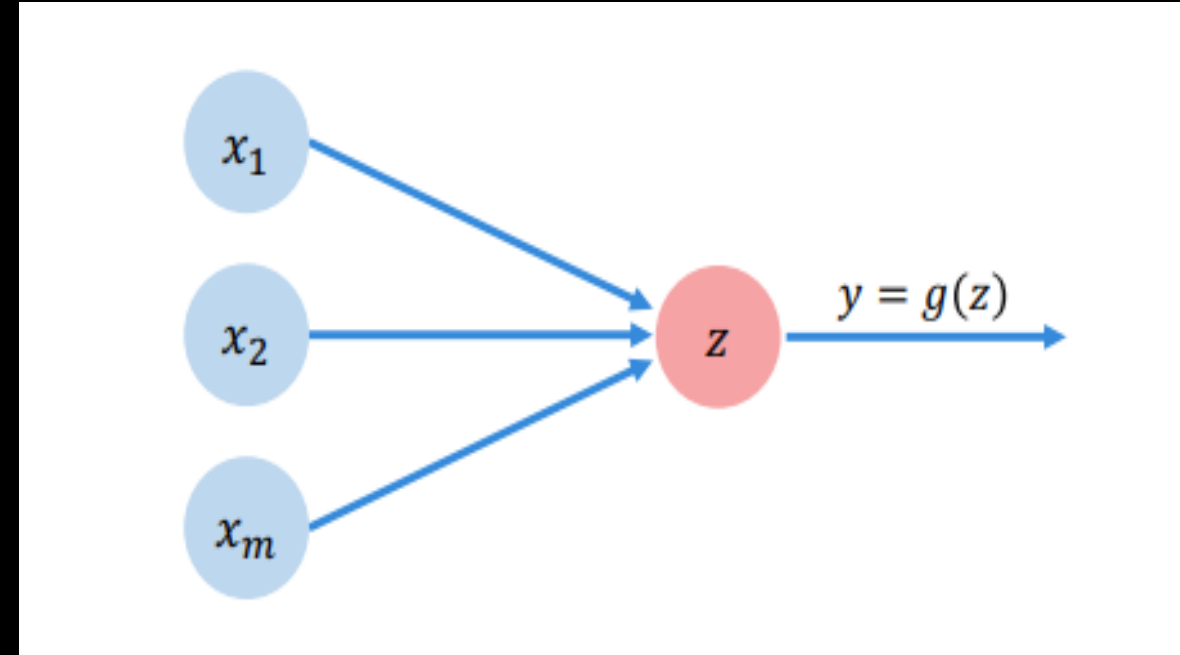
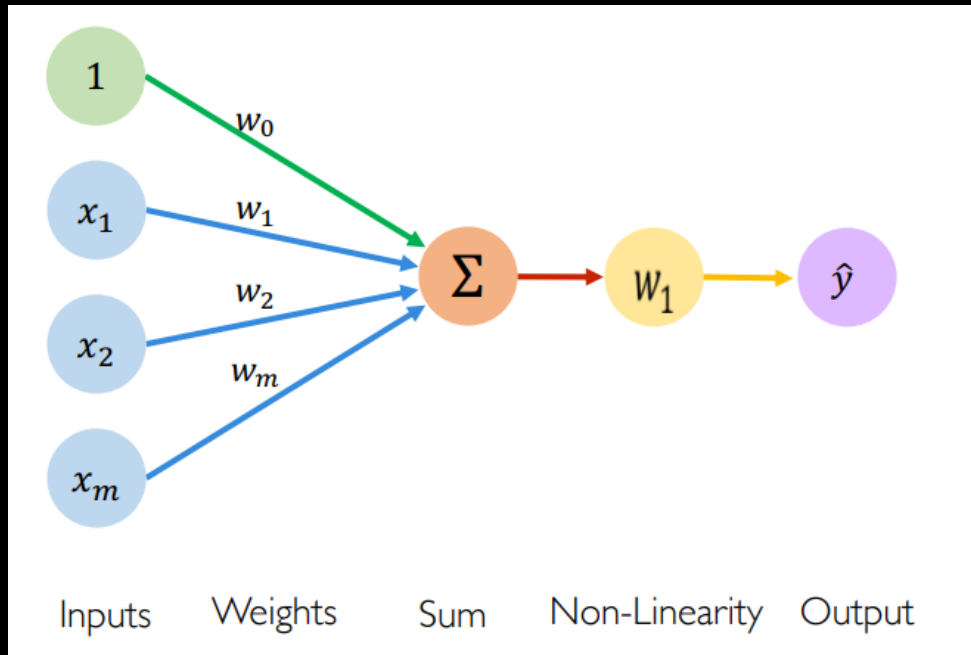
non linear approach

Conclusion

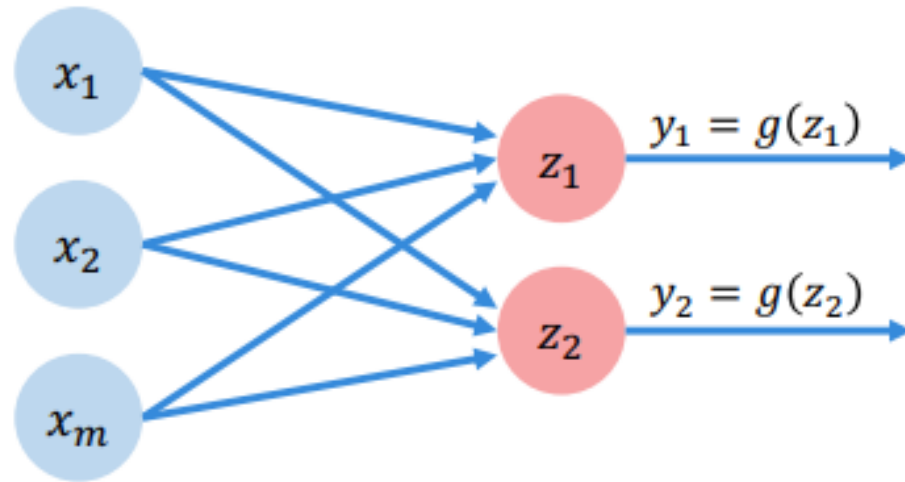
- Relationships in real world data are generally non – linear
- Linear activation functions can only describe linear relationships
- Non-linear activation function can describe arbitrary complex relationships
- Therefore Non-linear activation function are being used

Building Neural Networks

Simplifying the perceptron representation

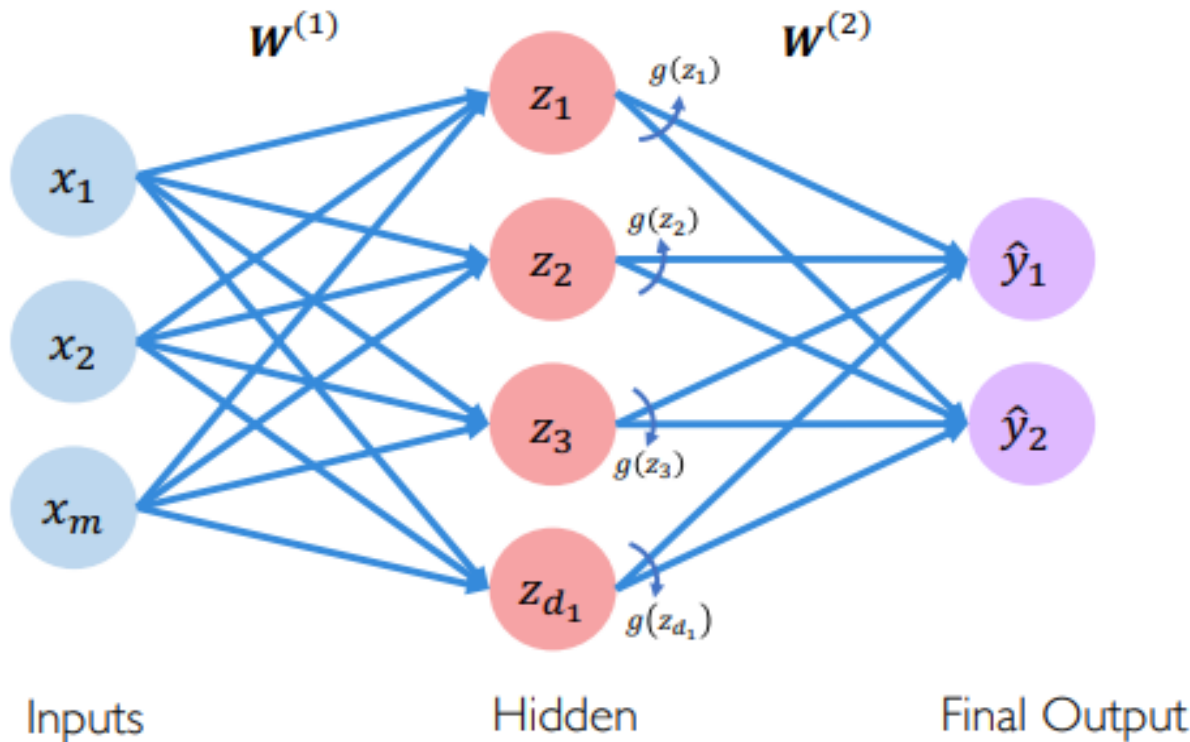


Multi output Perceptron



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

Single Layer Neural Network

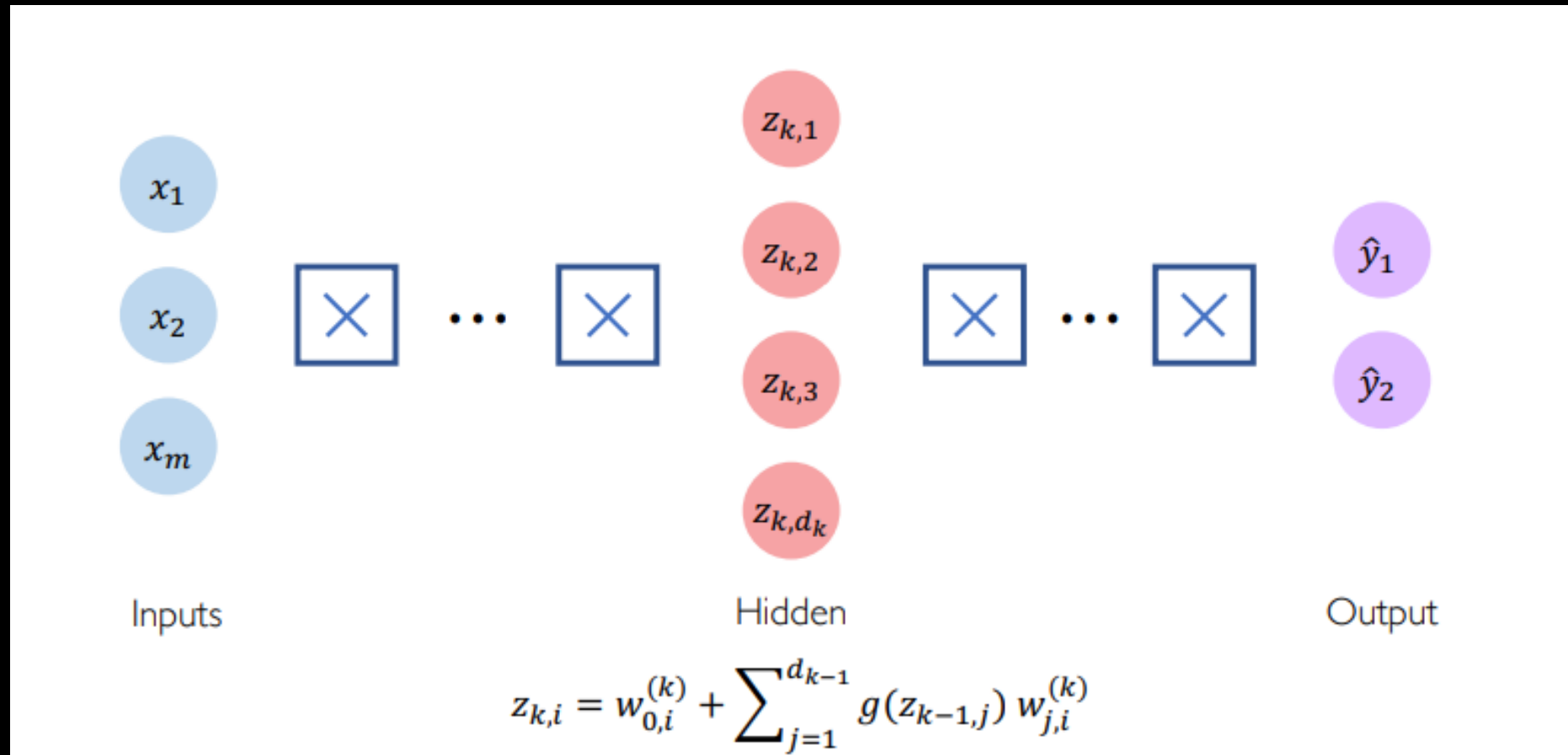


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \quad \hat{y}_i = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)} \right)$$



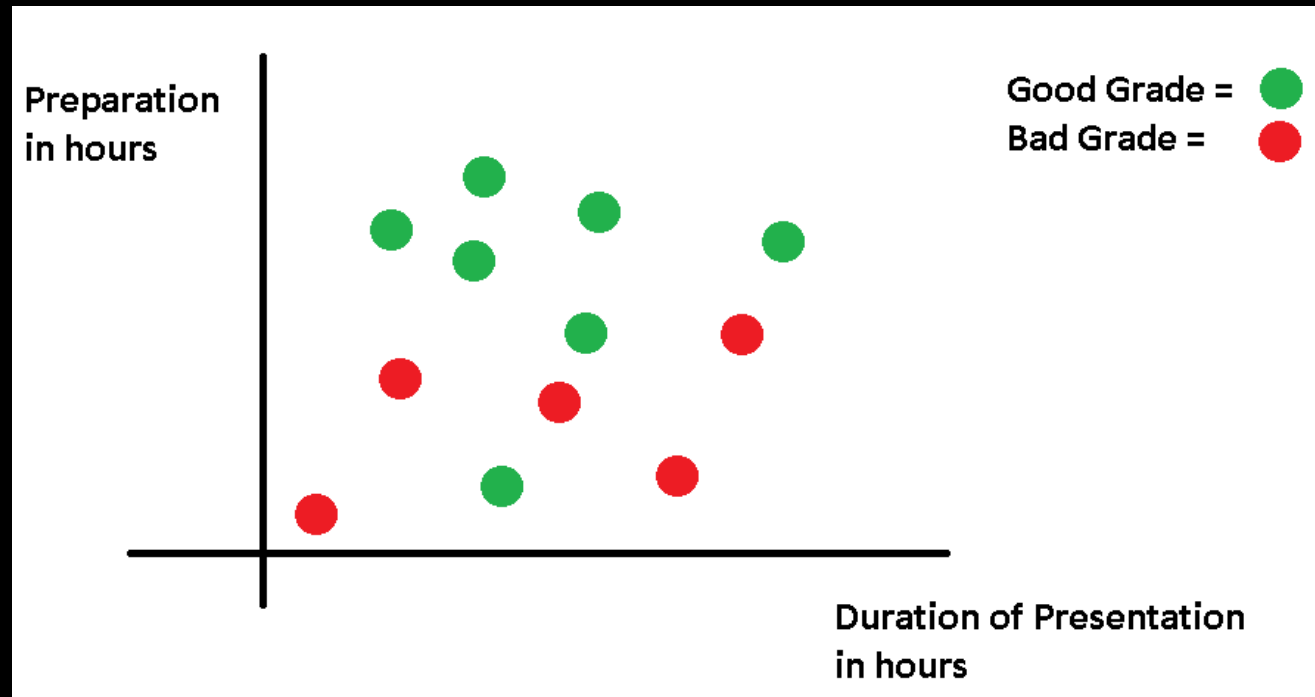
```
from tf.keras.layers import *  
  
inputs = Input(m)  
hidden = Dense(d1)(inputs)  
outputs = Dense(2)(hidden)  
model = Model(inputs, outputs)
```

Deep Neural Network

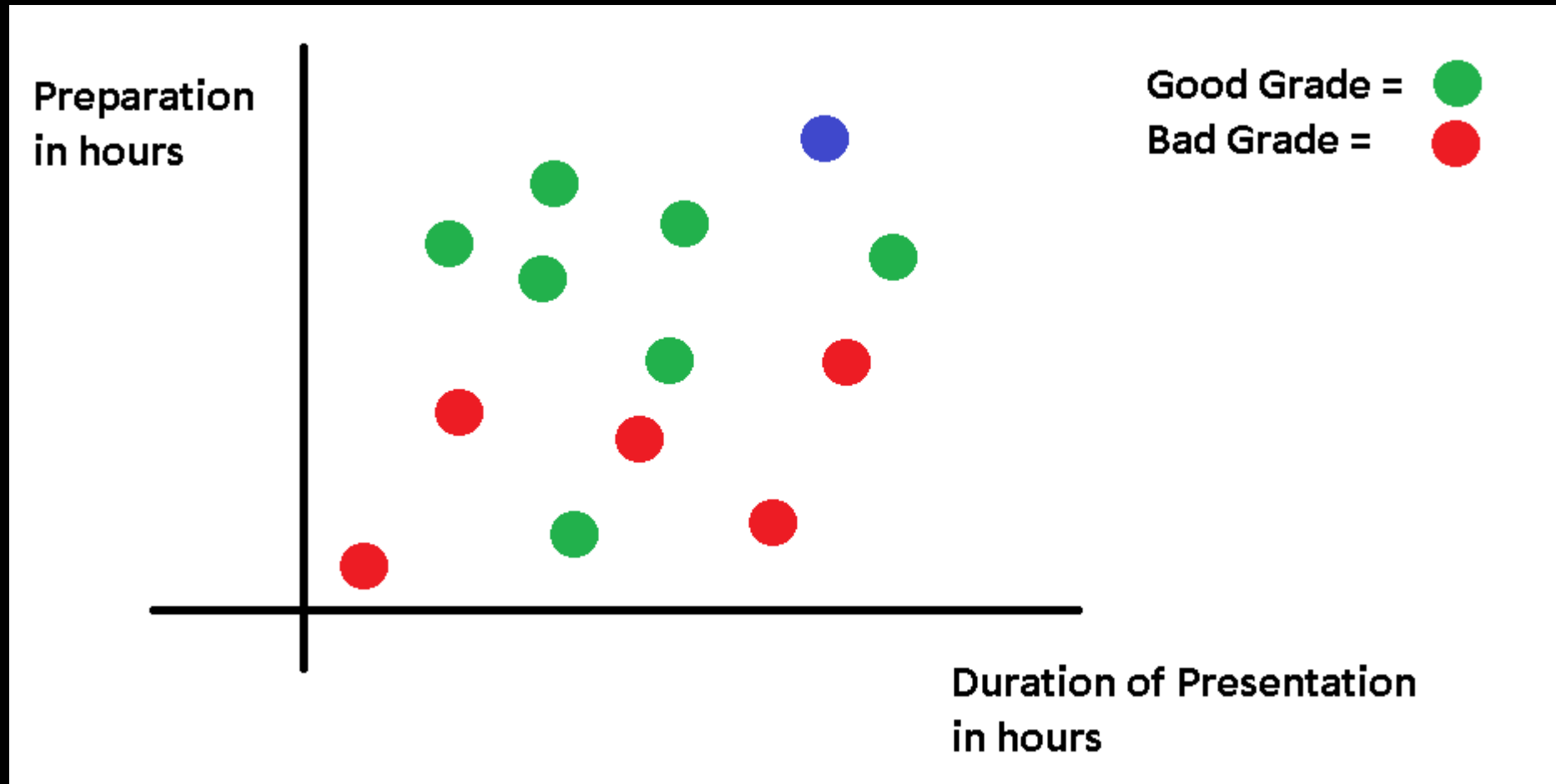


Neural Networks in Practise

Example Problem: How much time will I have to invest into preparing my e-portfolio to get a good grade?

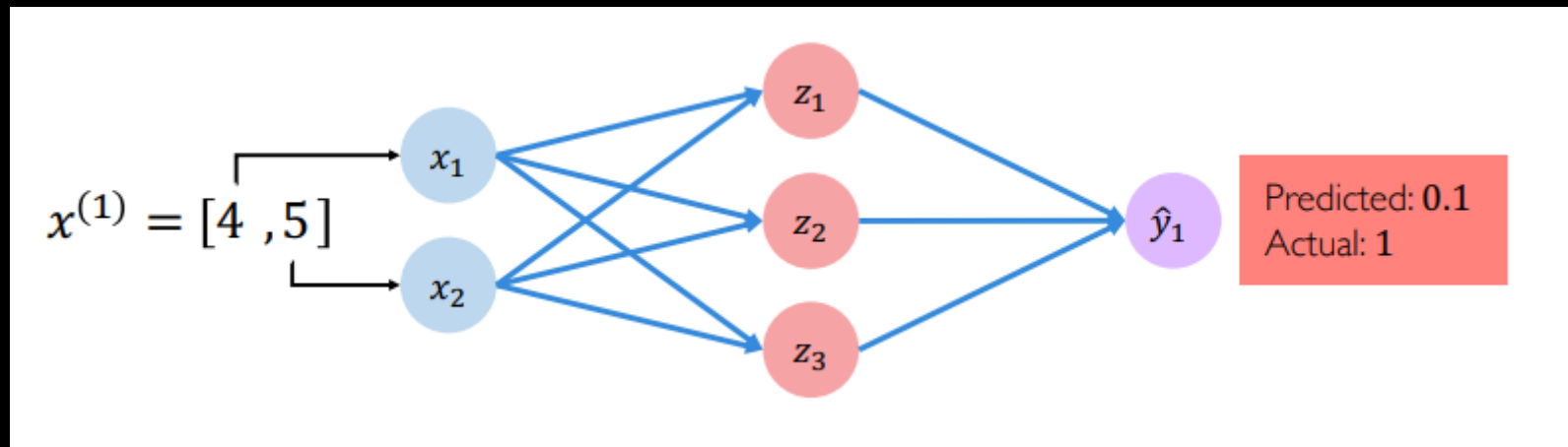


Will I get a good grade?



Predicting the outcome with a neural network

What is the problem here ?



Training Neural Networks

Defining Loss

The loss of a neural network is determined by the deviation of the prediction of the network from the actual value.

Loss Function:= $L(f(x^{(i)}; W), y^{(i)})$

Actual:= $y^{(i)}$

Predicted:= $f(x^{(i)}; W)$

Common Loss Functions

Empirical Loss: Mean loss from all the predicted inputs

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

Binary Cross Entropy Loss: Used for Models which output values between 0 and 1

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



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

Common Loss Functions

Mean Squared Error Loss:=

used with regression models that output continuous real numbers

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



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

How does defining a Loss Function help us to train Neural Network?

Loss Optimization

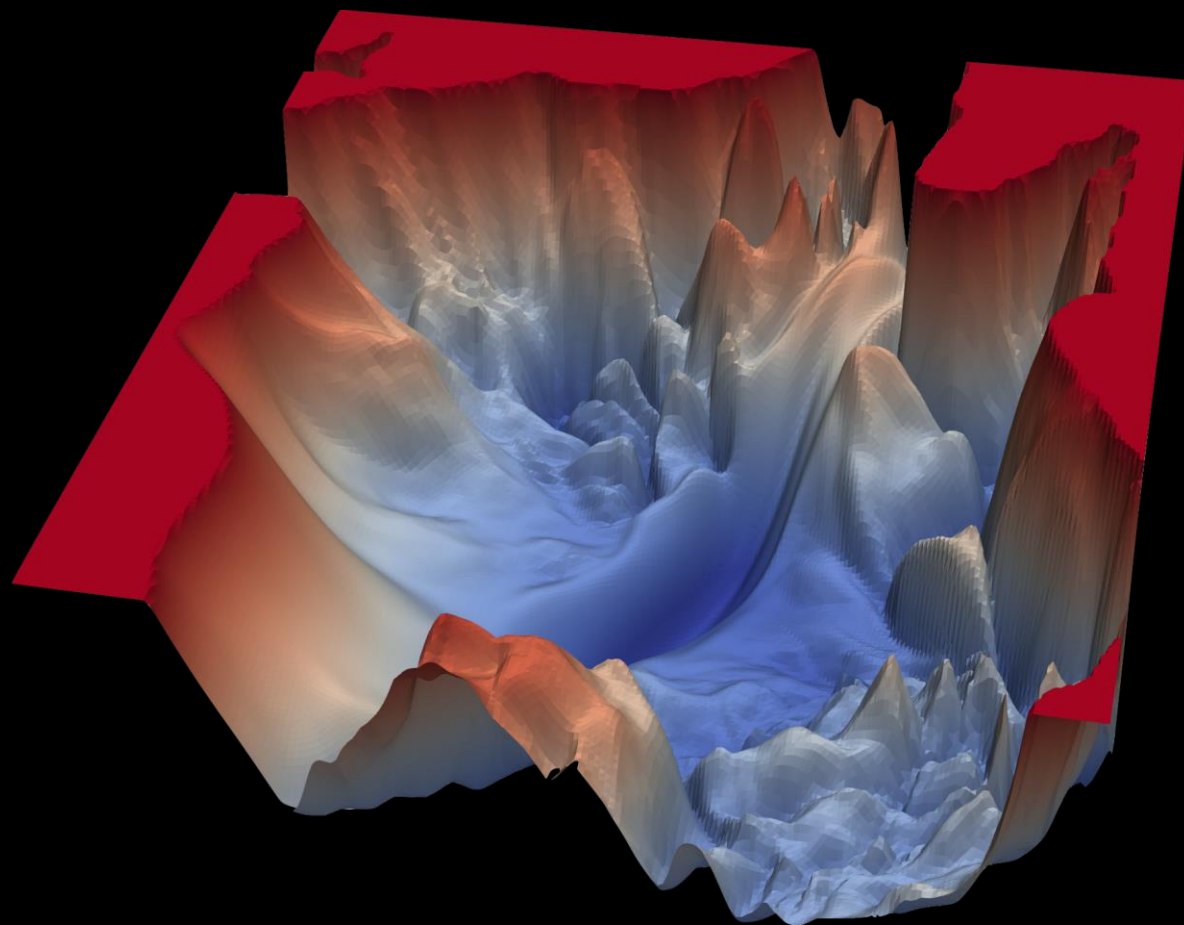
We want to find a network, which achieves the smallest loss

Since the weights are the only variable which can be modified. We need to find the weight which minimizes the loss.

Mathematical Definition:

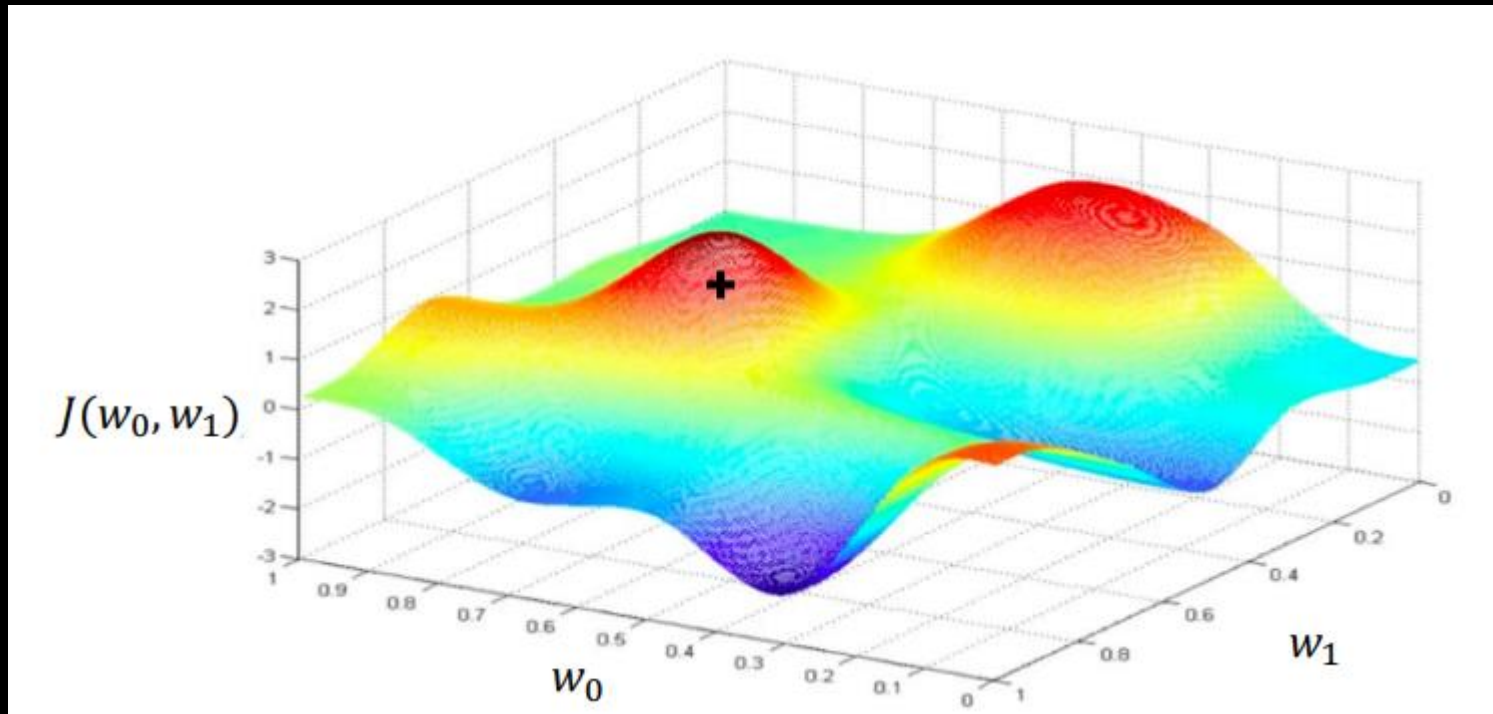
$$W^* = \operatorname{argmin} J(w)$$

Loss Landscape



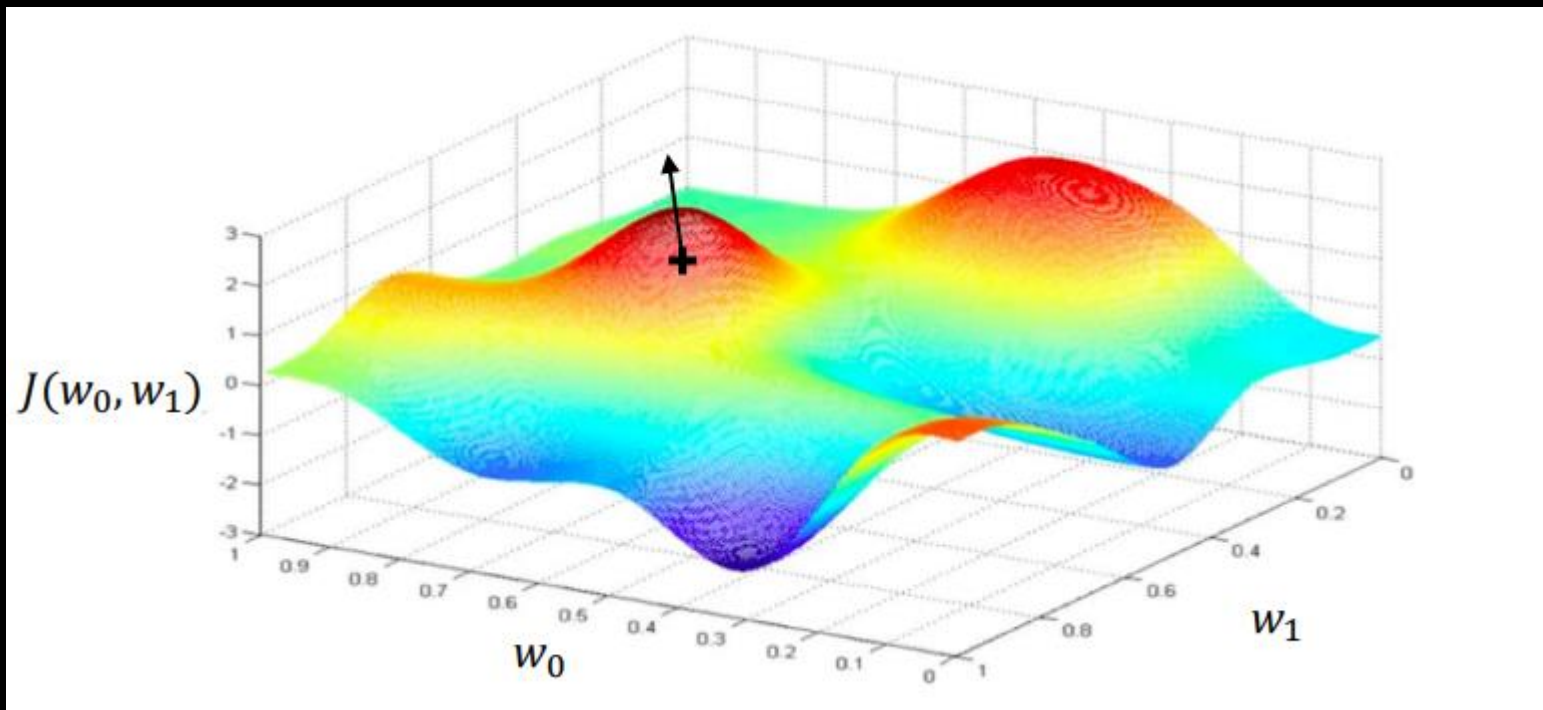
Loss Optimization with Gradient Descent

Step 1. Randomly pick initial weights (w_0, w_1)



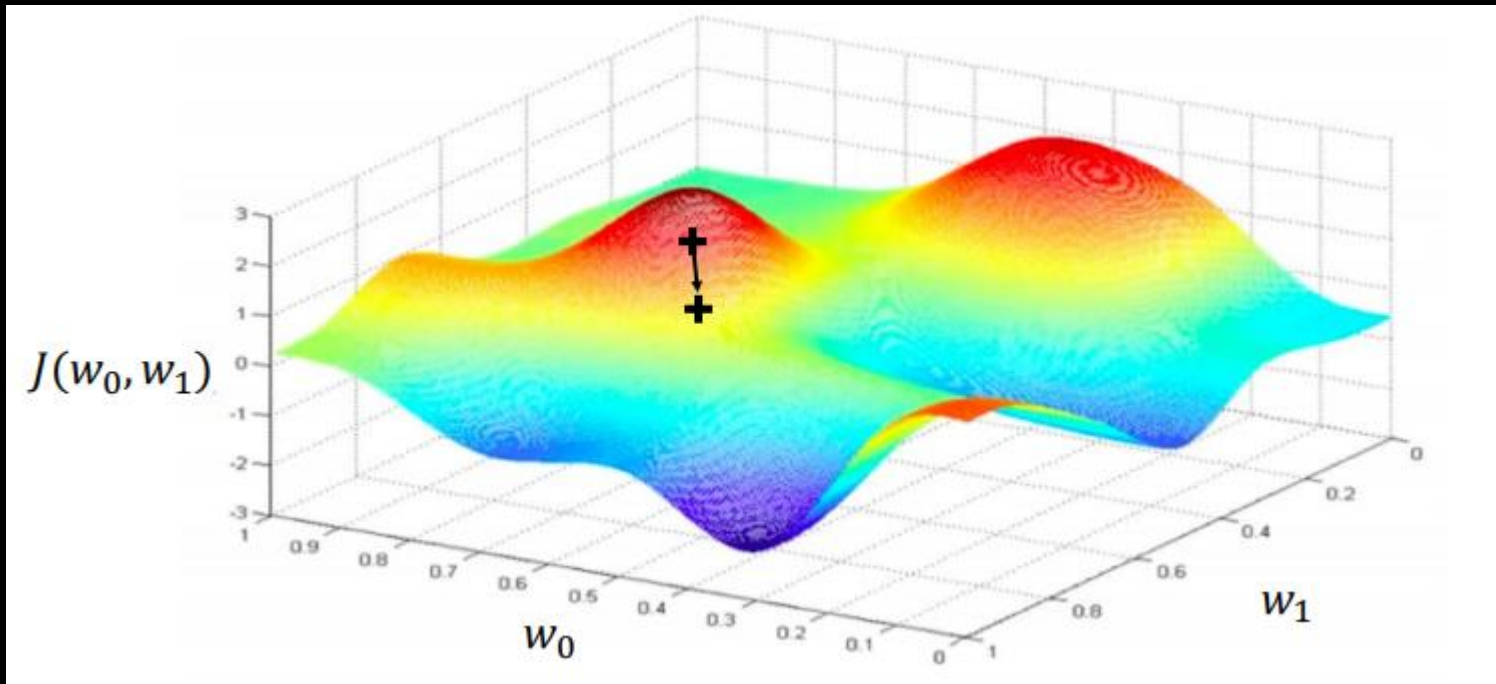
Gradient Descent

Step 2. Compute the gradient $\frac{\partial J(W)}{\partial W}$



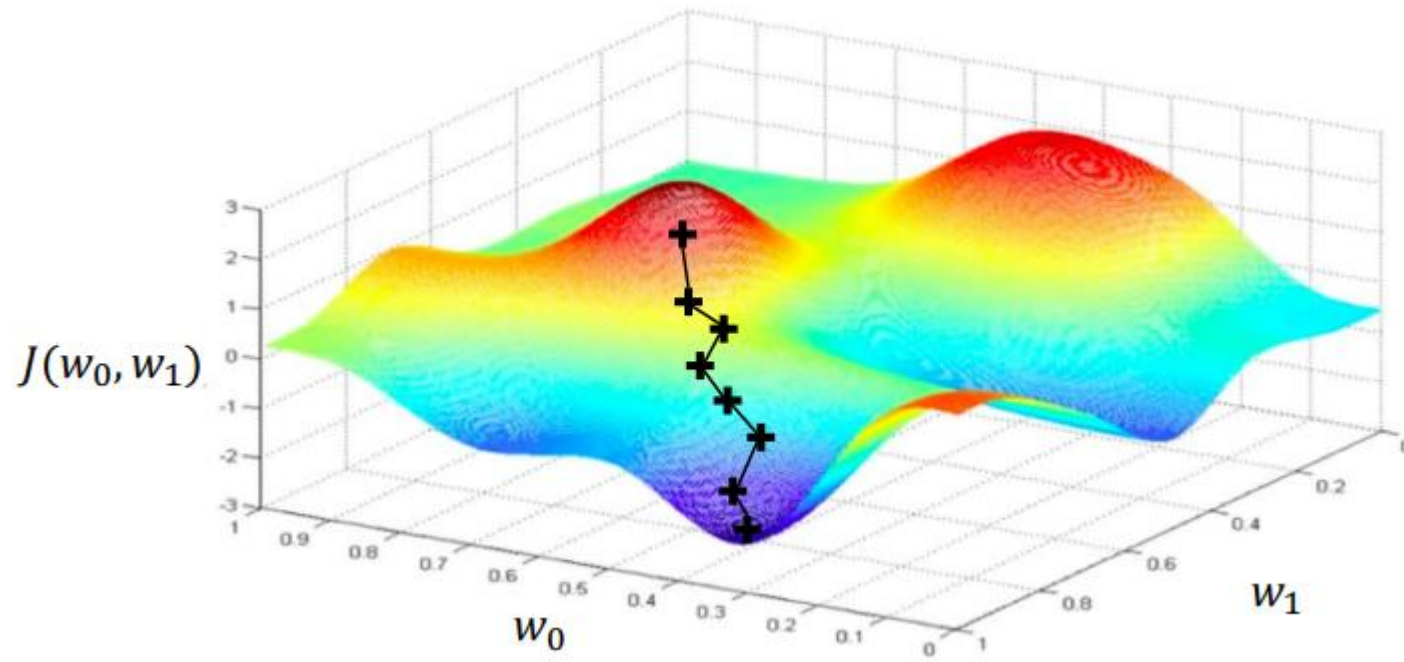
Gradient Descent

Step 3. Negate the gradient and take small step in the direction of the negated gradient



Gradient Descent


If $\frac{\partial J(W)}{\partial W} = z, |z| > 0$ return to step 1.




Loss Optimization by Gradient Descent

Algorithm:


Step 1. Randomly pick initial weights (w_0, w_1)

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

Step 2. Compute the gradient $\frac{\partial J(W)}{\partial W}$

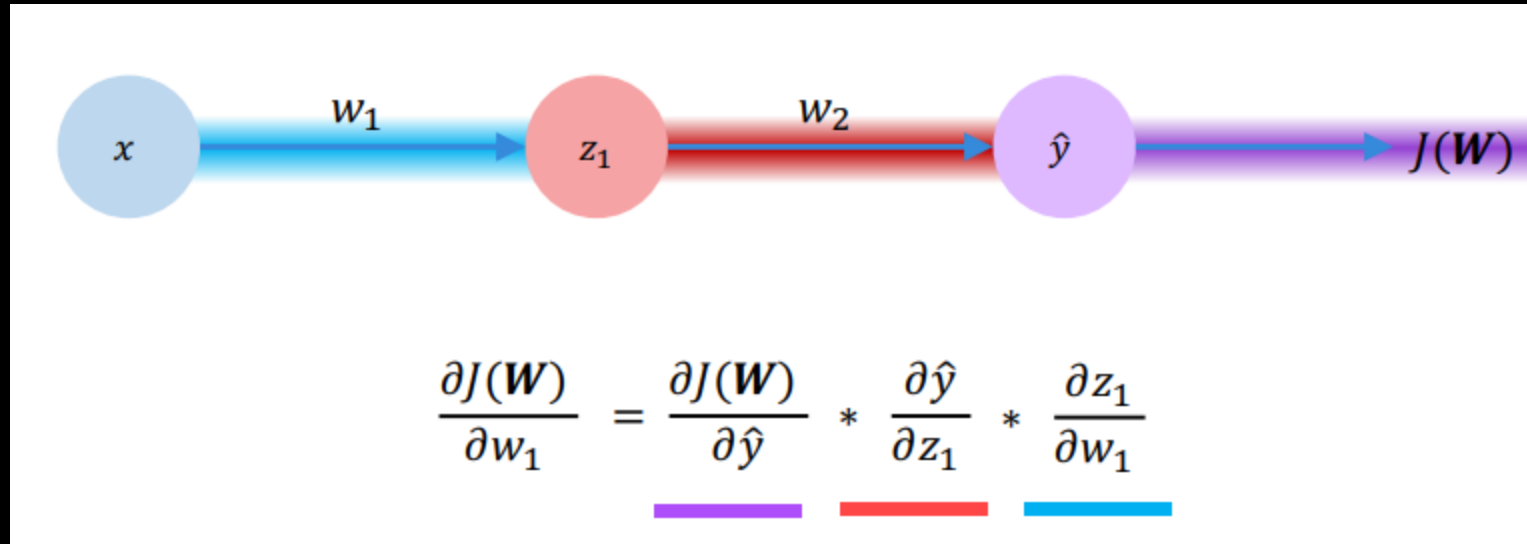
```
 grads = tf.gradients(ys=loss, xs=weights)
```

Step 3. Negate the gradient and take small step in the direction of the negated gradient

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

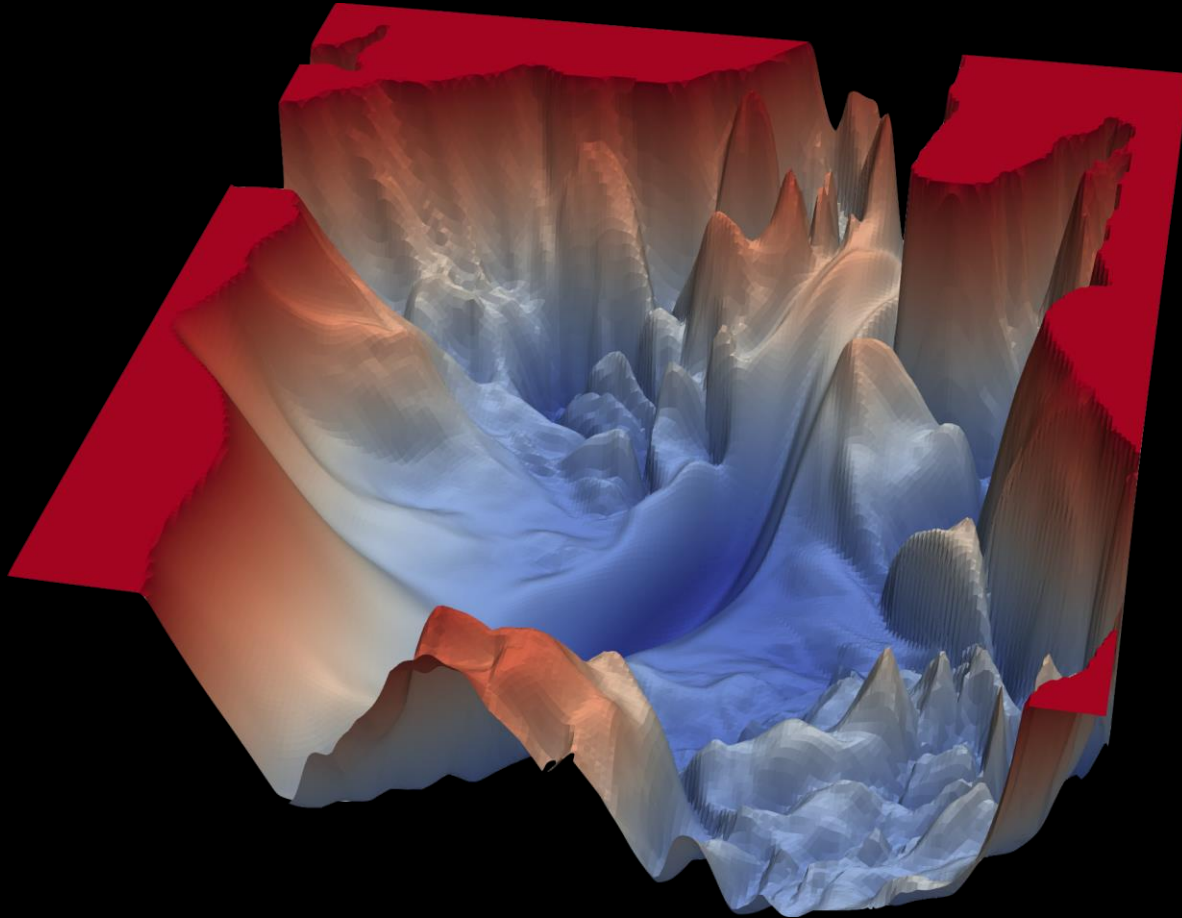
Step 4: If $\frac{\partial J(W)}{\partial W} = z, |z| > 0$ return to step 1.

Computing the Gradient through Backpropagation



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

Is there a problem with our current Loss optimization
Algorithm?

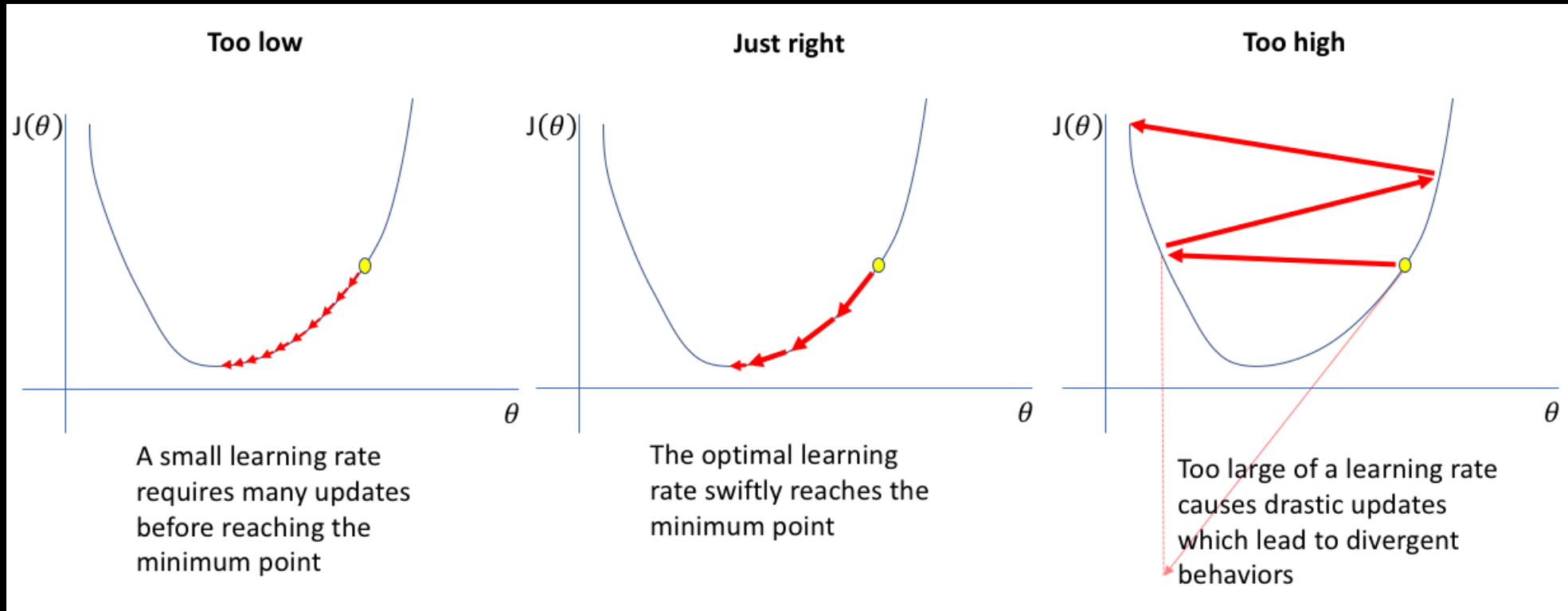


How can we optimize loss optimization?

Lets take a look at our Gradient again: $\eta \frac{\partial J(W)}{\partial W}$

- η = Learning Rate, the learning rate is variable
- 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

Learning Rates visualized



Finding the optimal learning rate

Solution 1: Try lots of different learning rates until satisfied with result

Solution 2: Design an adaptive learning rate that considers the landscape in which optimizing occurs, when choosing learning rate

Usual Suspects

- Adagrad
- Adadelta
- Adam
- RMSProp
- Momentum

 `tf.train.AdagradOptimizer`

 `tf.train.AdadeltaOptimizer`

 `tf.train.AdamOptimizer`

 `tf.train.RMSPropOptimizer`

 `tf.train.MomentumOptimizer`

Fixing the Gradient Computation Bottleneck

Problem: computing the gradient can be hard

- If the loss for a lot of predictions has to be considered during every gradient calculation. Gradient Computation becomes numerically expensive.
- Therefore compromises have to be made

Compromise 1: Stochastic Gradient Descent

- Stochastic gradient Calculation: only considers the loss of one prediction
- Benefit: easy to compute
- Negative: very noisy (stochastic)
- Mathematical Representation:

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

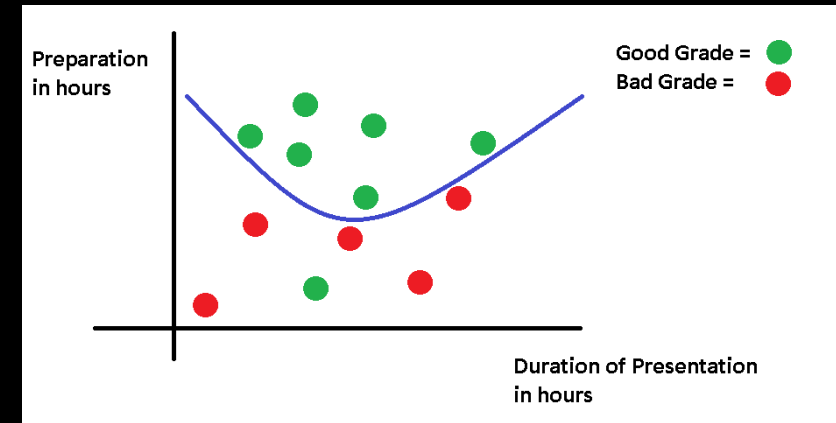
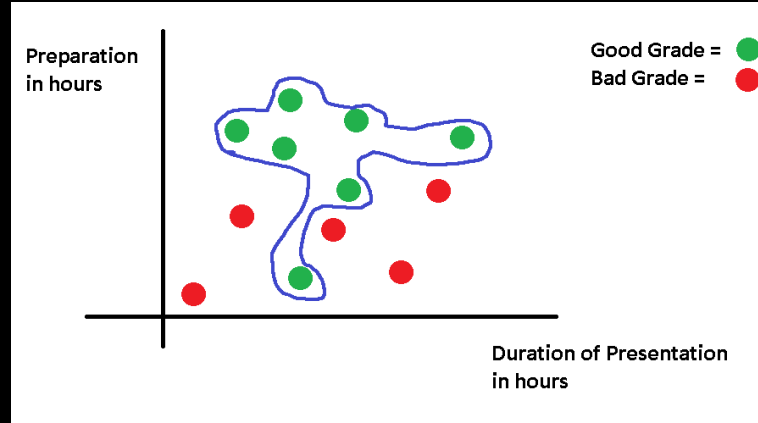
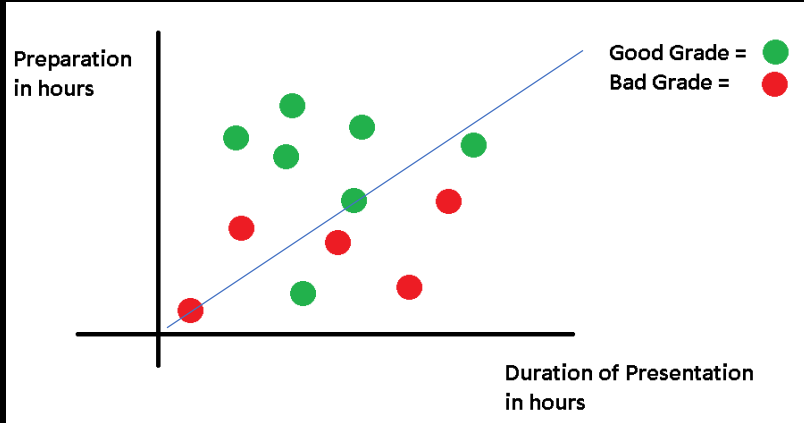
Compromise 2: Gradient descent using Batches

- Gradient calculation using Batches: considers the loss of multiple predictions but not for every prediction made
- Benefit: Still relatively easy compute, much higher accuracy than stochastic gradient descent
- Mathematical Representation:

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

The Last Frontier: „Overfitting“

Thought Experiment: Which of the following feature spaces shows the most ideally trained neural net

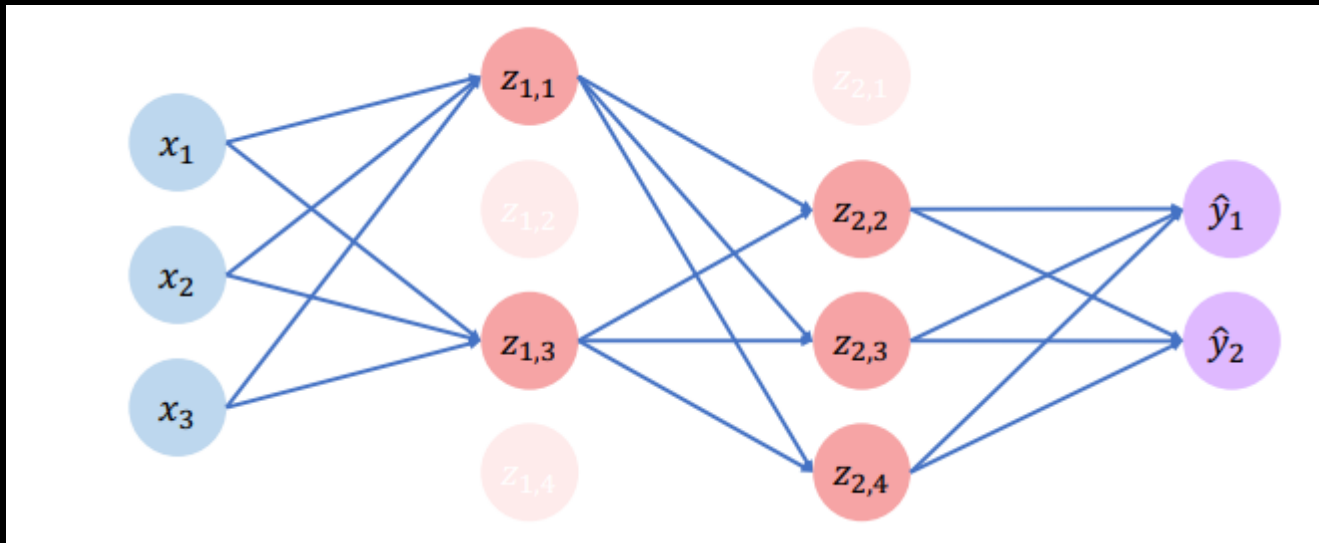


Preventing Overfitting through Regularization

- Technique that constrains our optimization problem to discourage complex models
- This keeps our model from memorizing the training data set

Reg. Technique 1: Dropout

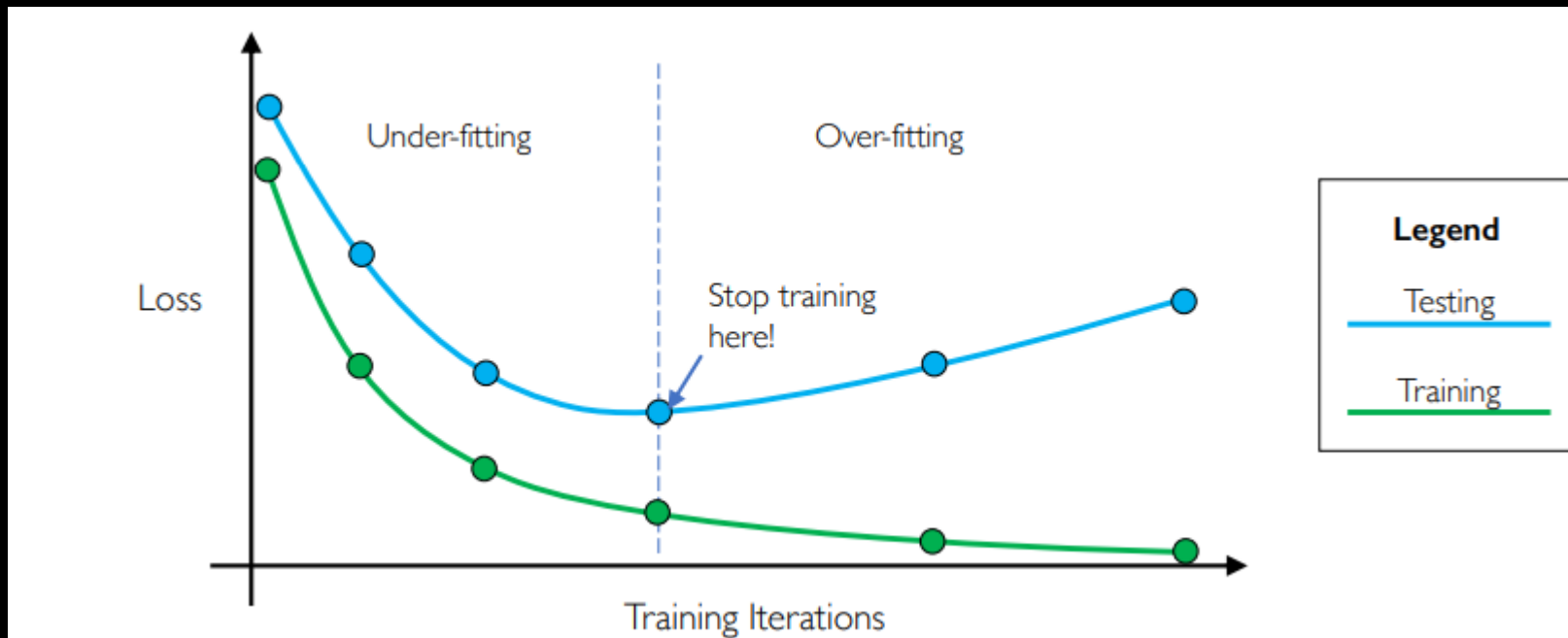
- Setting certain activation techniques to zero randomly
- Encourages the network to optimize more than one path through the network



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

Reg. Technique 2: Early Stopping

- Independently testing a neural network, while training using a different data set than the training data set



Summary Deep learning Basics

The Perceptron

- Foundation of deep learning
- Components of perceptrons

Neural Networks

- Using perceptron to create neural networks
- Understanding Loss
- Optimizing Loss

Training in Practise

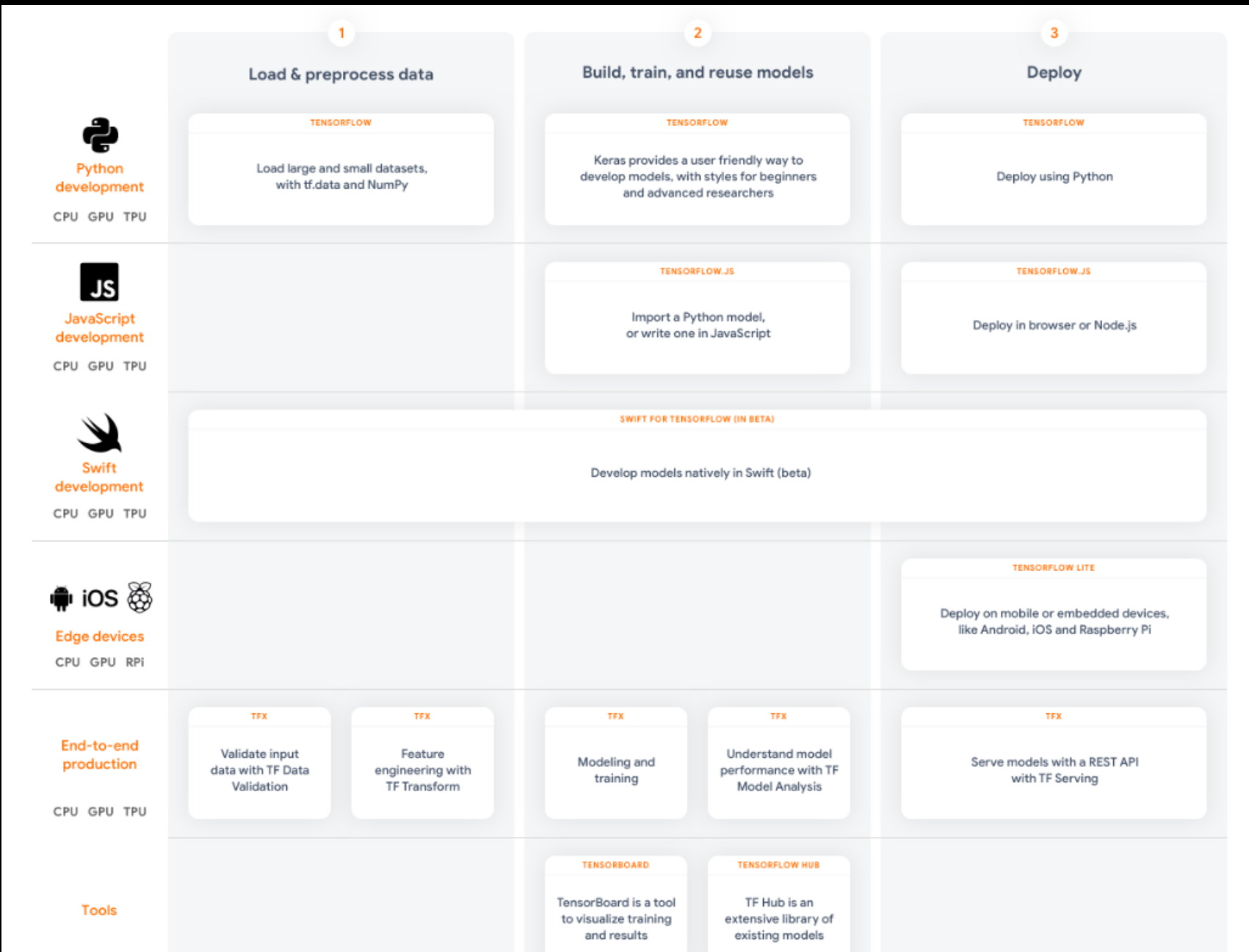
- Adaptive Learningrates
- Local minimum
- Gradient descent with batching
- Regularization

What is Tensorflow?

Basic Facts about Tensorflow

- „Tensor“ latin for Data
- Tensorflow = „Tensor“ + flow = Dataflow
- Deep Learning framework
- implemented in Python and C++
- developed by the Google Brain Team
- Tensorflow finds application in many google products like Gmail, Google Fotos and Google search





Tensorflow Components

- High Level APIs: Keras, Eager Execution, Estimators
- Low Level APIs: Tensors, variables, Graphs and Sessions, Ragged Tensors
- Tensorflow Debugger

The Keras API

- lets you build and configure models (`tf.keras.Sequential`)
- lets you train and evaluate models
- lets you Import pre-trained models
- lets you import datasets

Eager Execution

- imperative programming environment that evaluates operations immediately
- operations return concrete values instead of constructing a computational graph

Lab 1: Fun with Tensorflow

Lab 1

- Repository:

<https://github.com/OrangeFreitag/IntroductionToTensorflow>

Questions