Scientific Computation Project

Brain Tumor Classification (MRI)

MSCS Department
University of Tehran

Mohammad Zamani
610399135

What is Brain Tumor?

- A brain tumour is a growth of cells in the brain that multiplies in an abnormal, uncontrollable way.

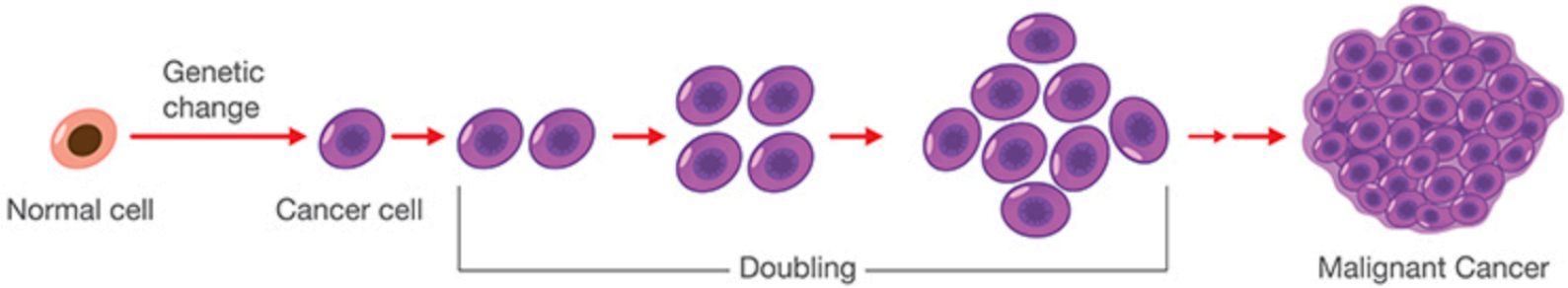


Figure: Process of cancer cell development

Brain Tumors

1. primary brain tumors
2. secondary brain tumors

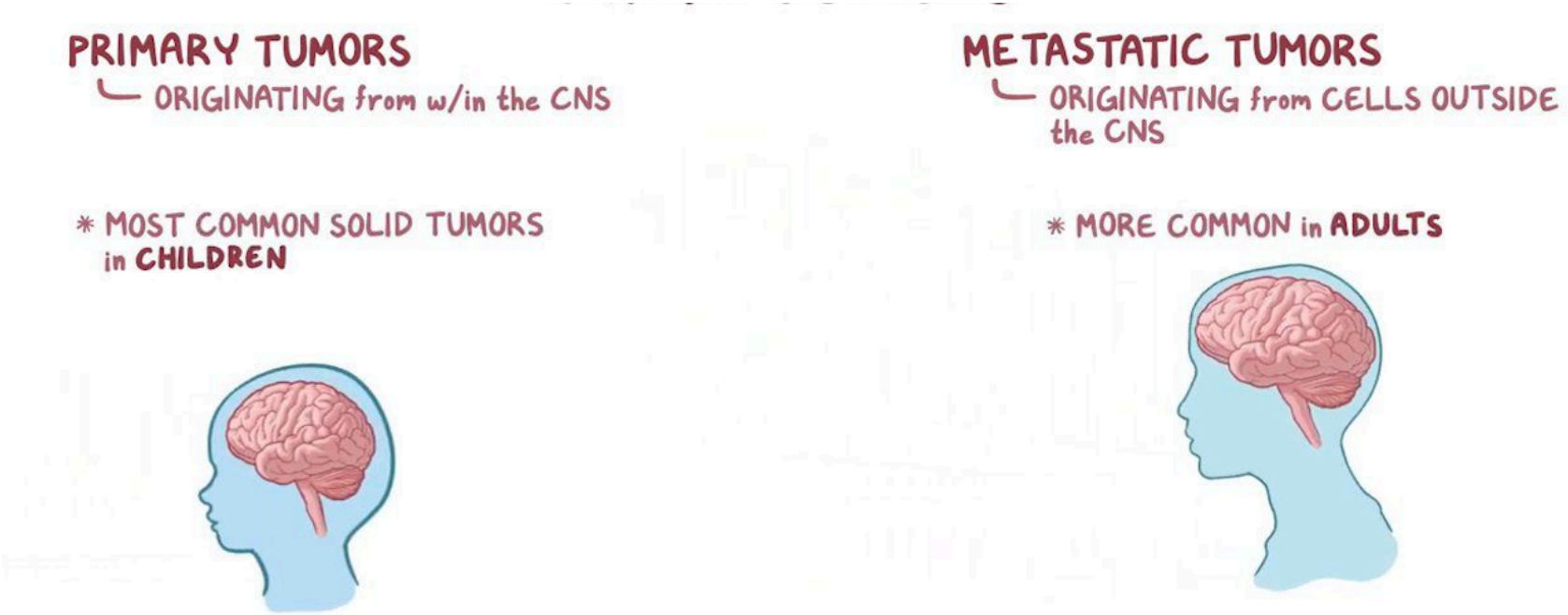


Figure: Primary vs Secondary brain tumor

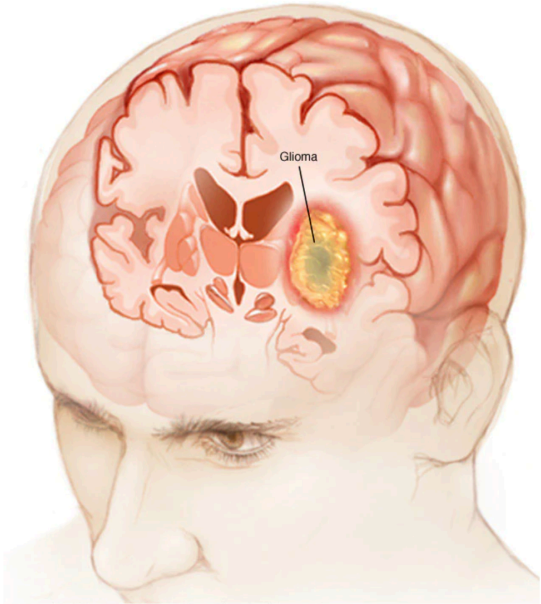
Types

- There are many types of brain tumors. The type of brain tumor is based on the kind of cells that make up the tumor.
  - Glioma
  - Meningioma
  - Pituitary

Glioma

- Glioma is a growth of cells that starts in the brain or spinal cord. The cells in a glioma look similar to healthy brain cells called glial cells. Glial cells surround nerve cells and help them function.

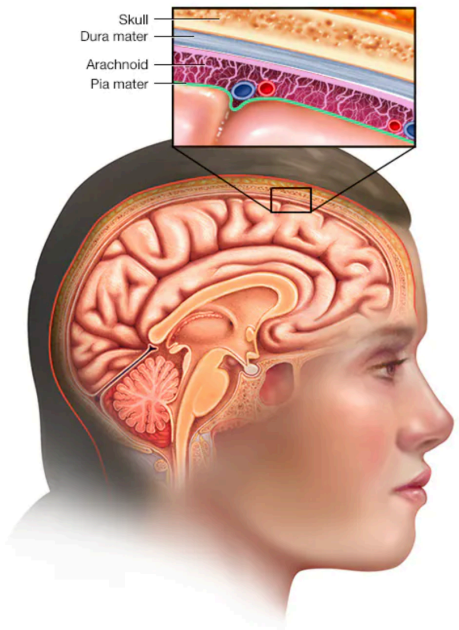
- Gliomas are most common in adults between ages 45 and 65 years old. But glioma can happen at any age.



Meningioma

- A meningioma is a tumor that grows from the membranes that surround the brain and spinal cord, called the meninges. A meningioma is not a brain tumor, but it may press on the nearby brain, nerves and vessels.

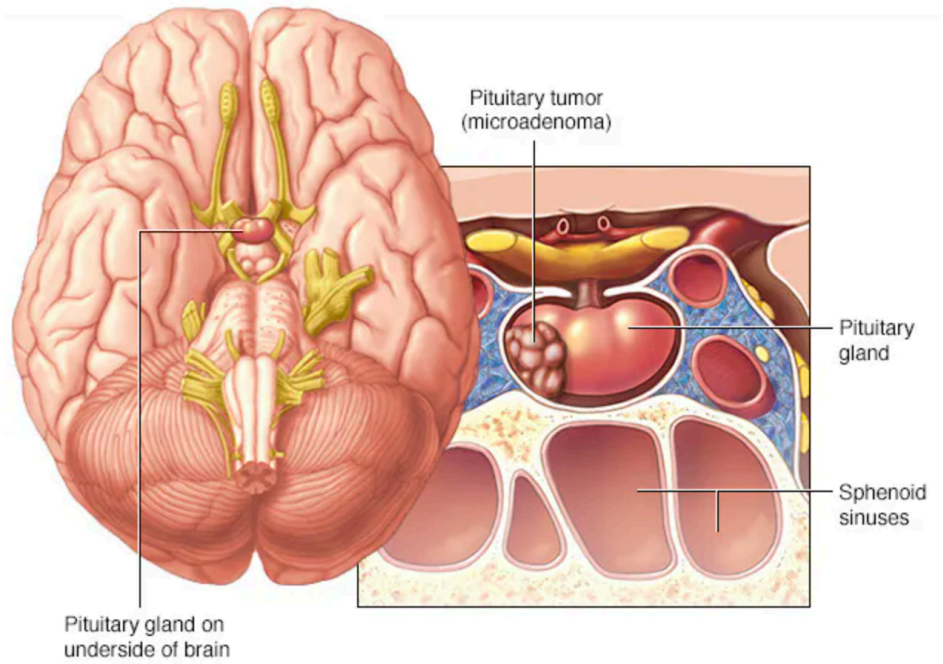
- Meningioma is the most common type of tumor that forms in the head.



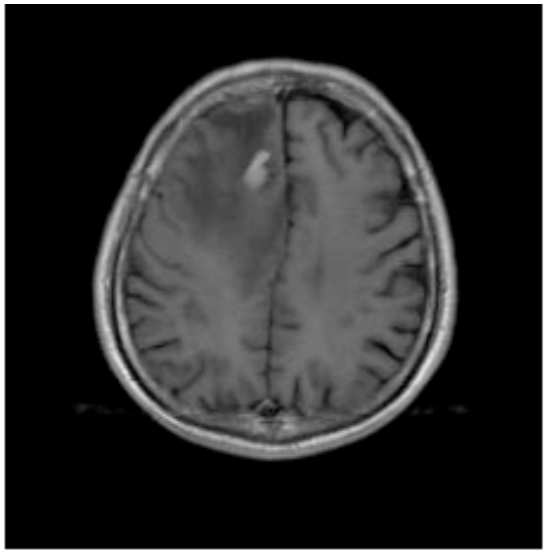
Pituitary

- Pituitary tumors are unusual growths that develop in the pituitary gland. This gland is an organ about the size of a pea. It's located behind the nose at the base of the brain.

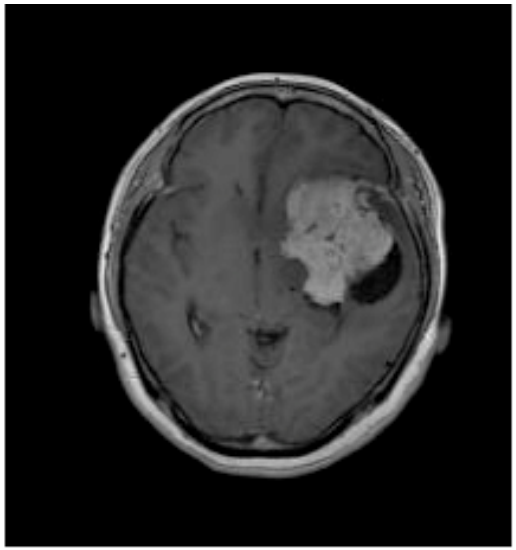
- Most pituitary tumors are benign. That means they are not cancer.



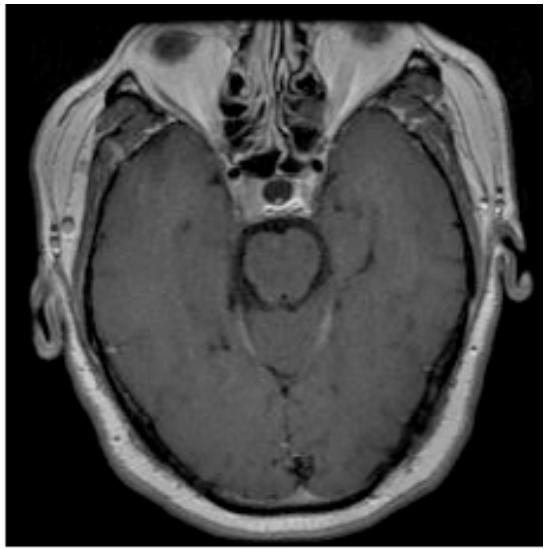
MRI



(a) glioma



(b) meningioma



(c) pituitary

Biological Analogy

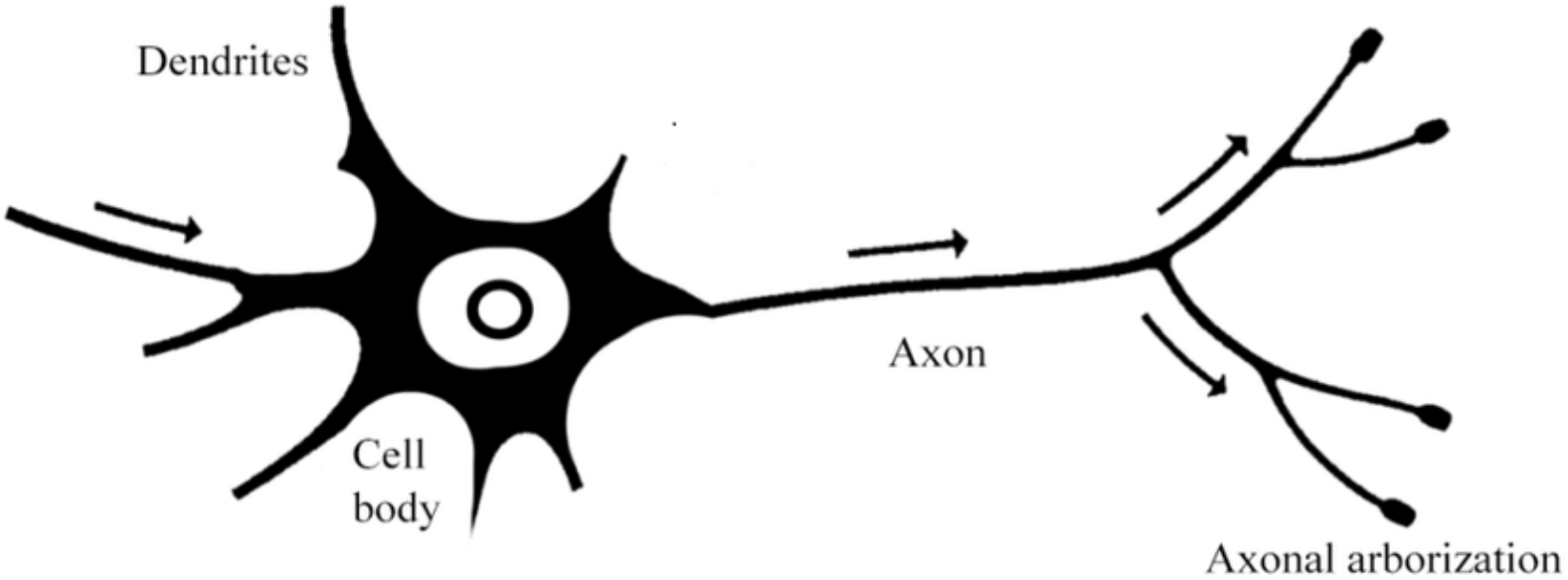


Figure: Anatomy of a biological neuron [**?**].

Activation Functions

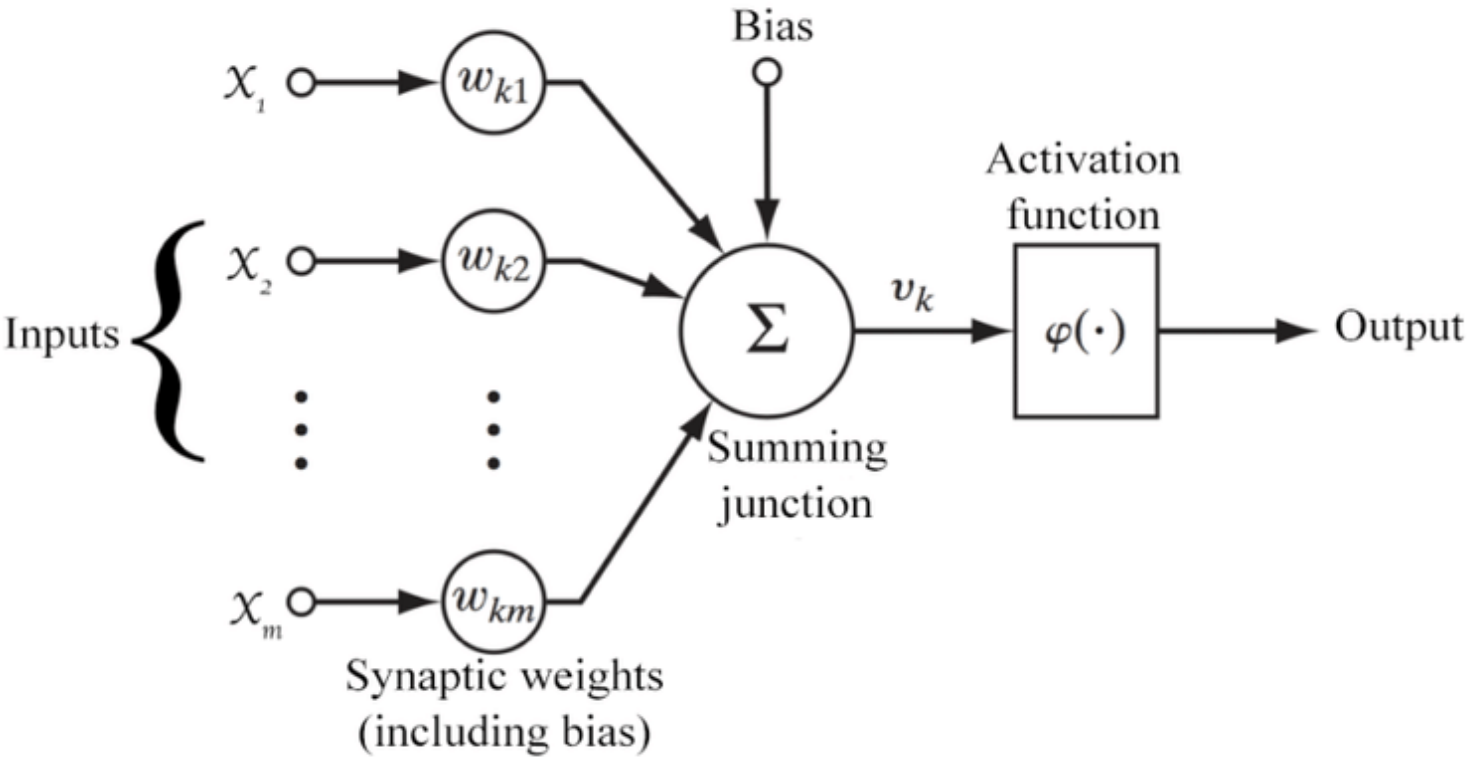


Figure: Neural network neuron [**?**].

Activation Functions

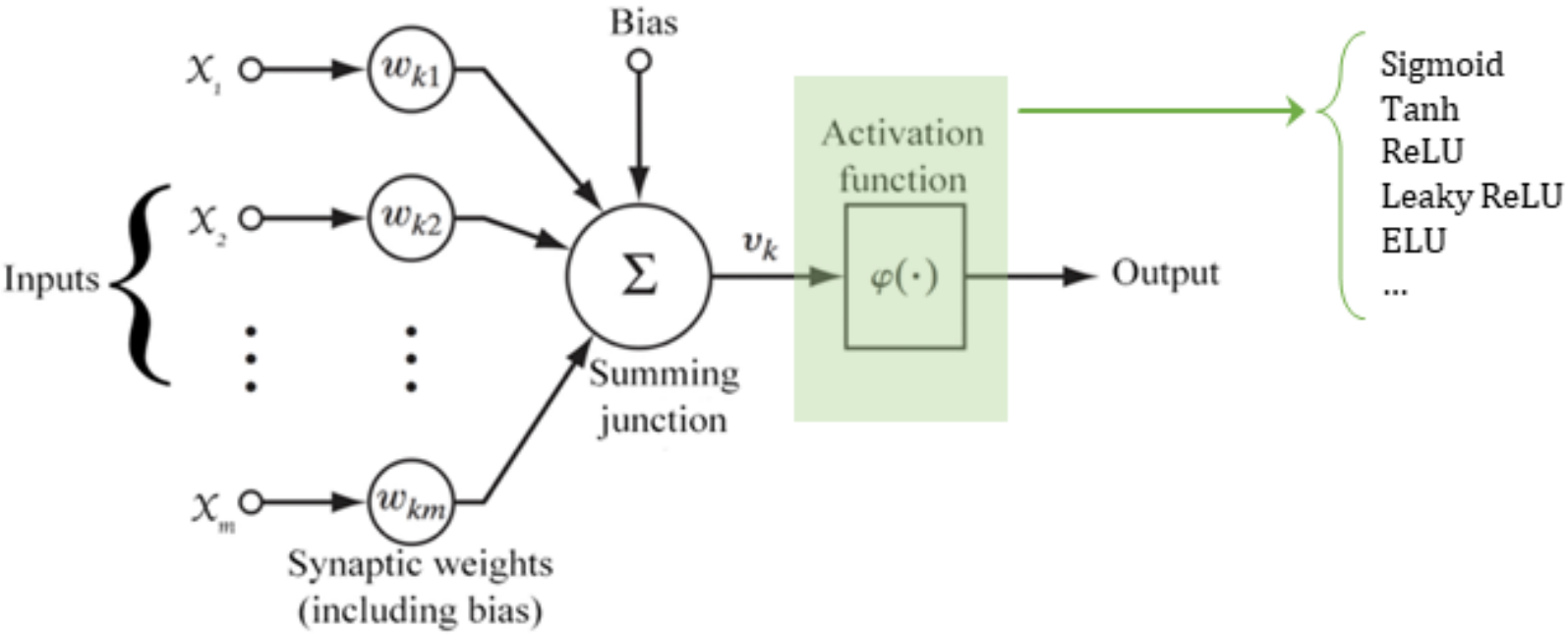
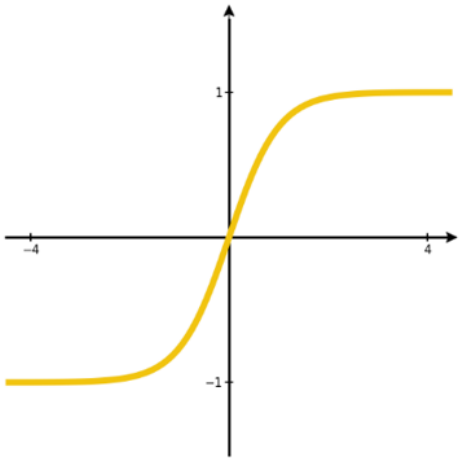
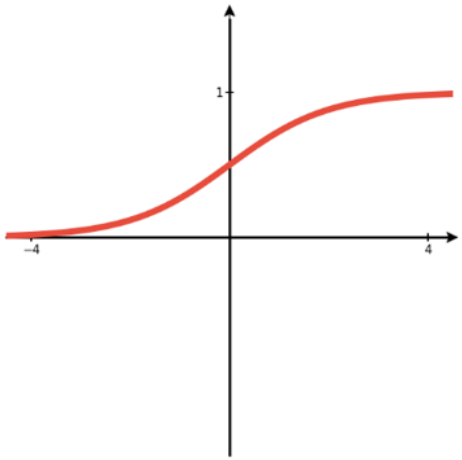
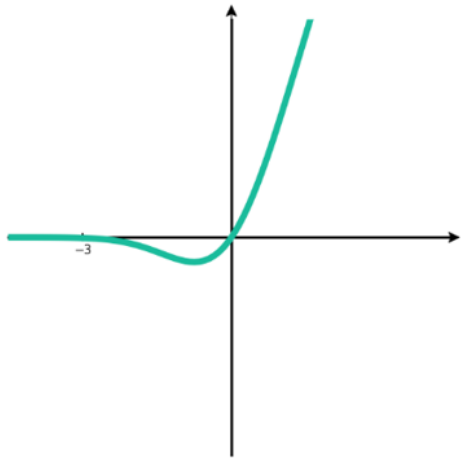


Figure: Activation function

Activation Functions

Tanh	Sigmoid	GELU
$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\sigma(x) = \frac{1}{1 + e^{-x}}$	$f(x) = \frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$
		

Softmax

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad i = 1, \cdots, J$$

Gradient Descent

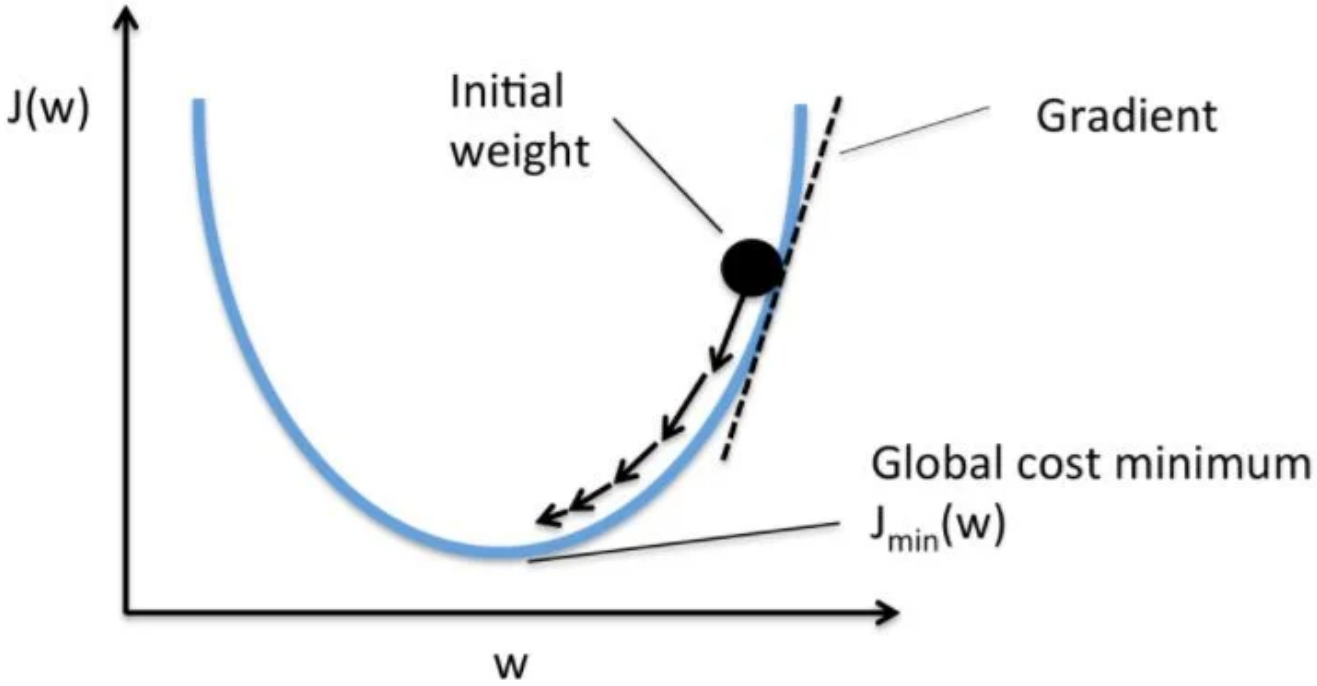


Figure: Gradient descent [**?**].

Gradient Descent

- Let's define our problem:
  - We have dataset $\mathcal{D} = \{x^i, y^{(i)}\}_{i=1}^n$.
  - f is a single layer perceptron.
  - Define $\hat{y}^{(i)} = f(x^{(i)})$.
- We want to minimize following cost function:

$$\mathcal{J}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

- We are going to use gradient descent algorithm. $\boldsymbol{w}$ will be updated as follows:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta\nabla_{\boldsymbol{w}}\mathcal{J}$$

Limitations of SLP

- What are the limitations of SLP?
- Can we learn all functions with SLP?

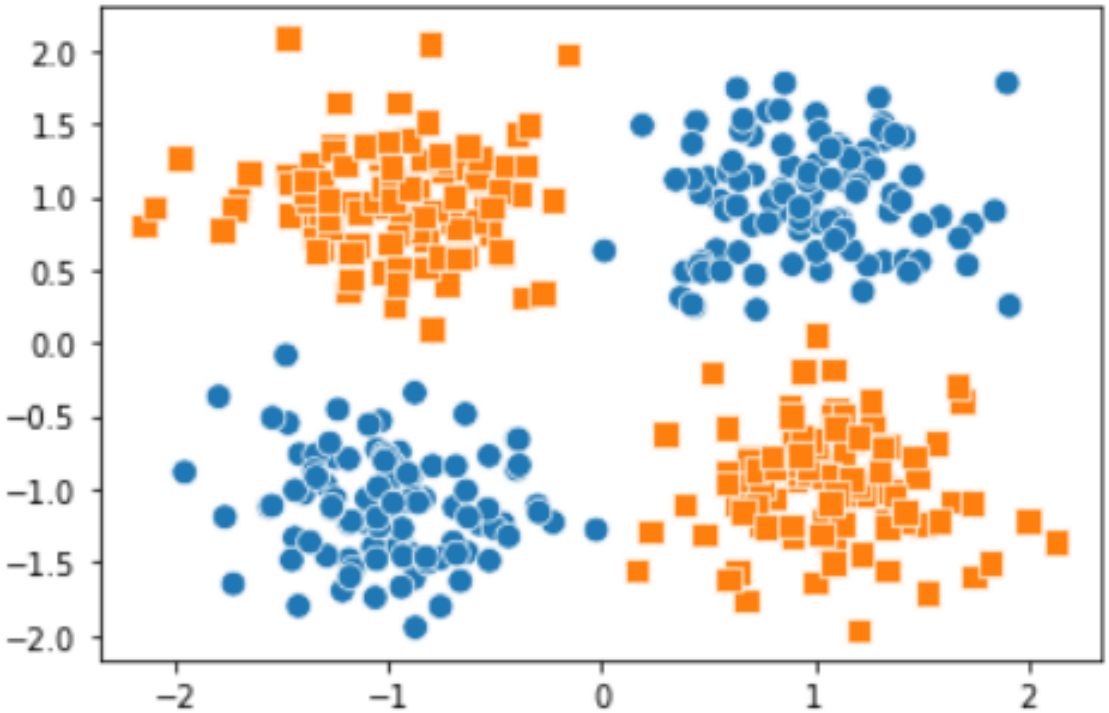


Figure: The XOR function is not linear separable.

Multi-Layer Perceptron

- So if we know some $f_1, \cdots, f_4$ we can use SLP to solve problem.

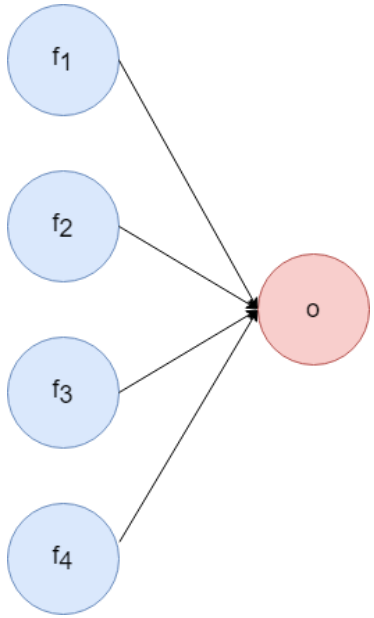


Figure: Using feature space f to solve problem.

- How to learn this f_is? Use SLP!

Multi-Layer Perceptron

- We can use inputs (x_i) to learn features (f_i)

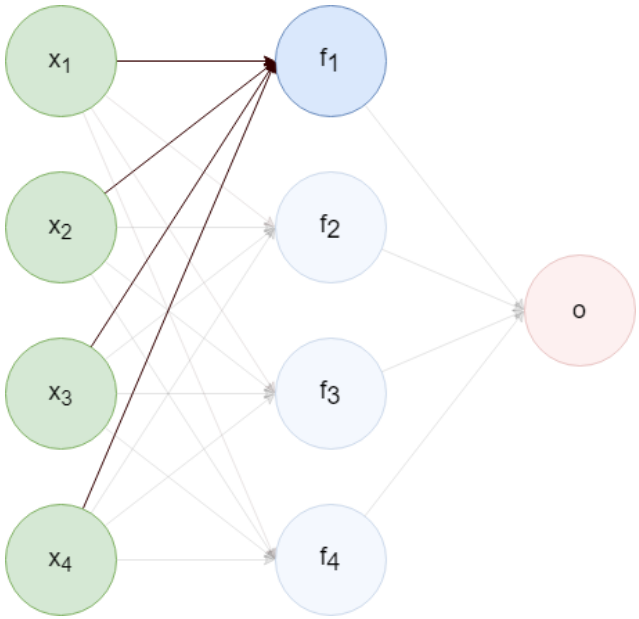


Figure: Using inputs to learn features.

- What if f_is are not sufficient? We can add more layers!

Multi-Layer Perceptron

- Adding more layers we will have a bigger network.

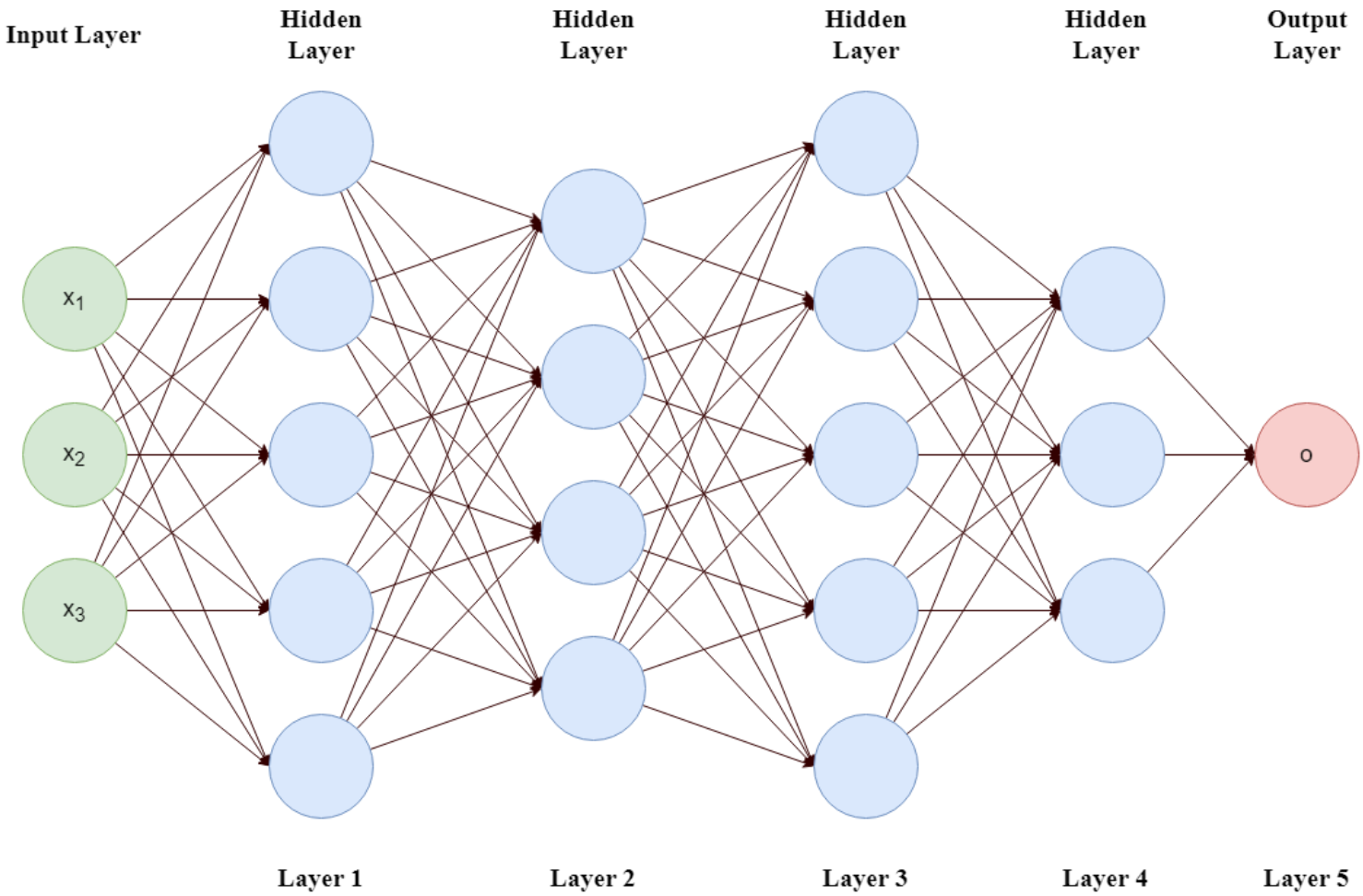


Figure: A 5 layer MLP.

Architecture of MLPs

- Important questions:
  - How many hidden layers should we have?
  - In each hidden layer, how many neurons should we have?

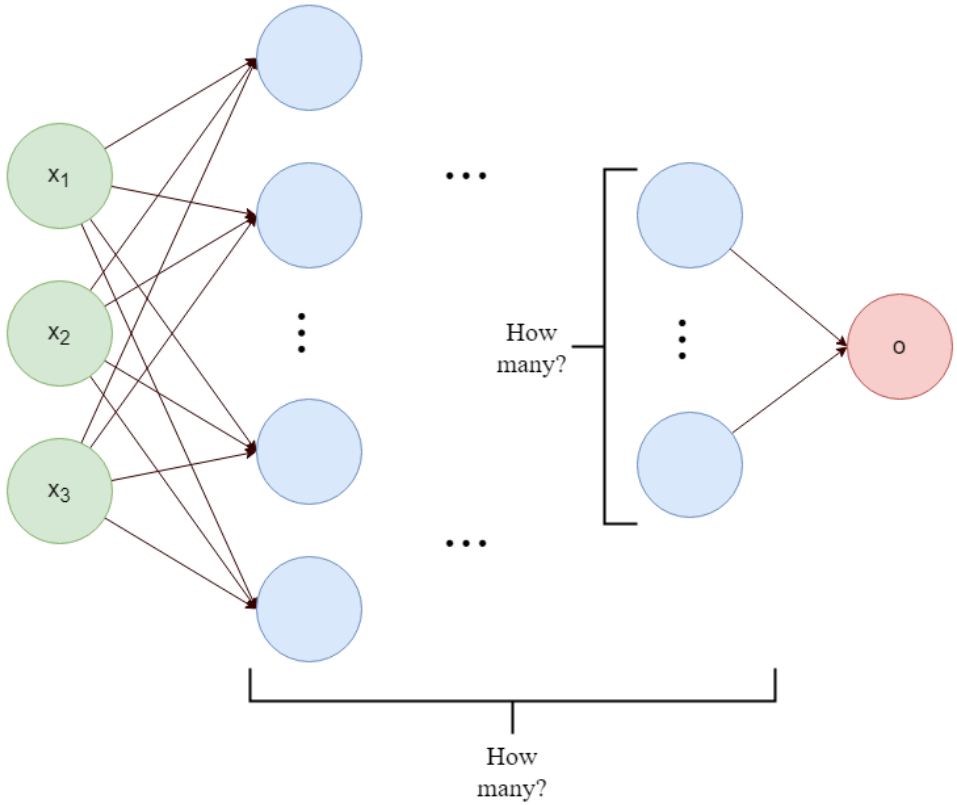


Figure: How many layers and neurons should we have?

Architecture of MLPs

- In practice:
  - You have limited resources
  - There is no universal rule to choose this hyperparameters
  - Need to experiment for different number of layers and neurons in each layer

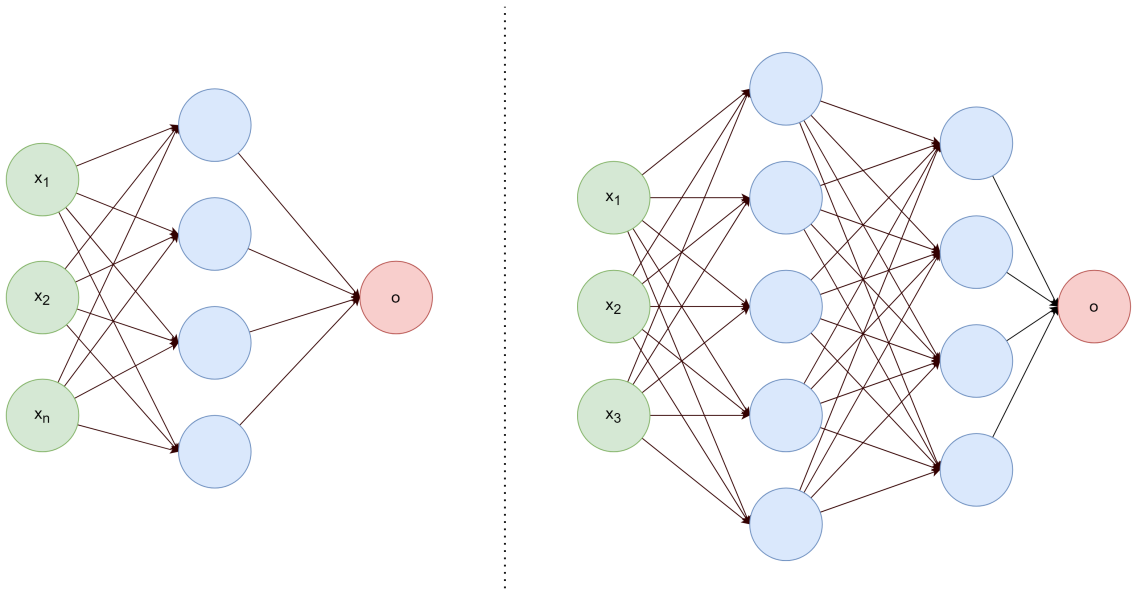


Figure: Experiment for different architecture and choose the best model.

XOR problem

- Now let's solve XOR problem with MLPs.
- We have two binary inputs, build an MLP to calculate their **XOR**.

- First let's build logical **AND** and **OR** functions.

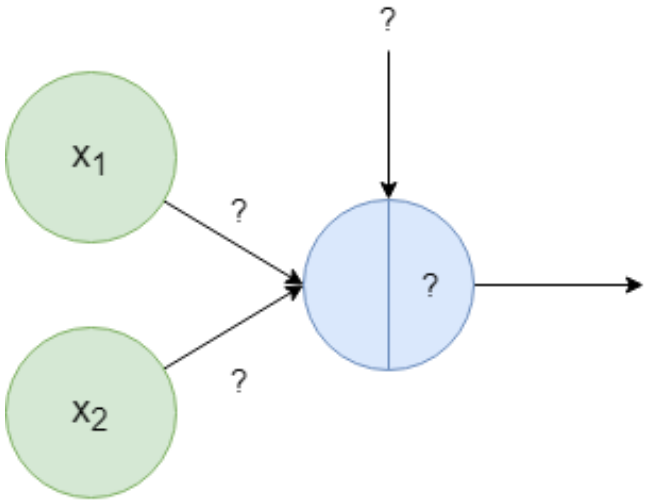
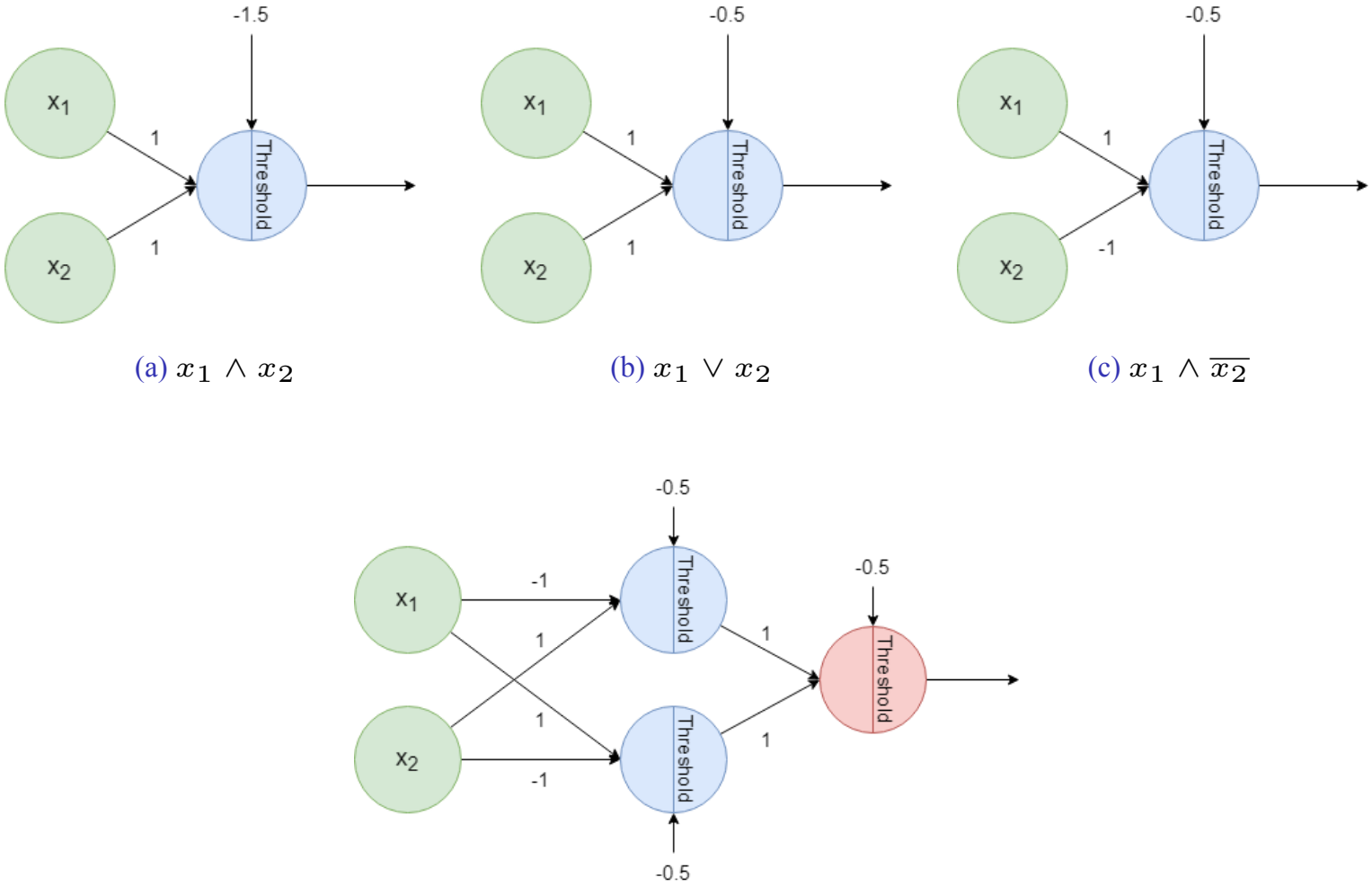


Figure: We need to find weights, biases and activation function.

XOR problem



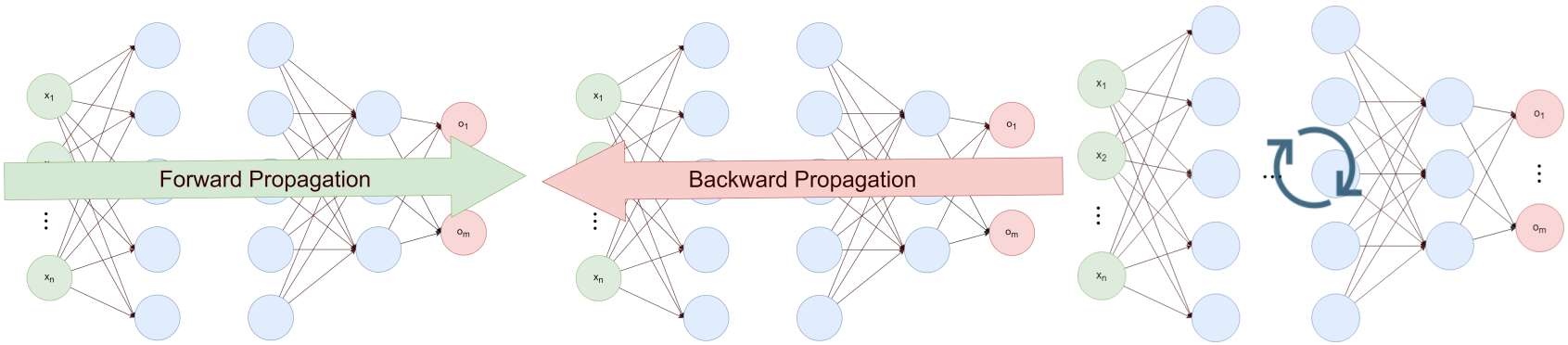
(a) $x_1 \wedge x_2$ (b) $x_1 \vee x_2$ (c) $x_1 \wedge \overline{x_2}$

Figure: MLP for XOR function.

Learning MLPs

- Till here we have used networks with predefined weights and biases.
- How to learn these parameters?

- The idea is to use gradient descent



(a) Forward pass (b) Backward pass (c) Update parameters

Learning MLPs

- Let's define the learning problem more formal:
  - $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$: dataset
  - f: network
  - W: all weights and biases of the network ($W^{[l]}$ and $\boldsymbol{b}^{[l]}$ for different l)
  - L: loss function

- We want to find W^* which minimizes following cost function:

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

- We are going to use gradient descent, so we need to find $\nabla_W \mathcal{J}$.

Forward Propagation

- First of all we need to find loss value.
- It only requires to know the inputs of each neuron.

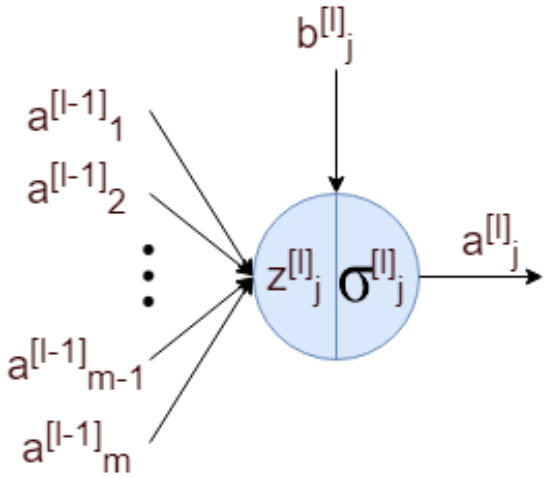


Figure: $a_j^{[l]} = \sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]}$

- So we can calculate these outputs layer by layer.

Forward Propagation

- After forward pass we will know:
  - Loss value
  - Network output
  - Middle values

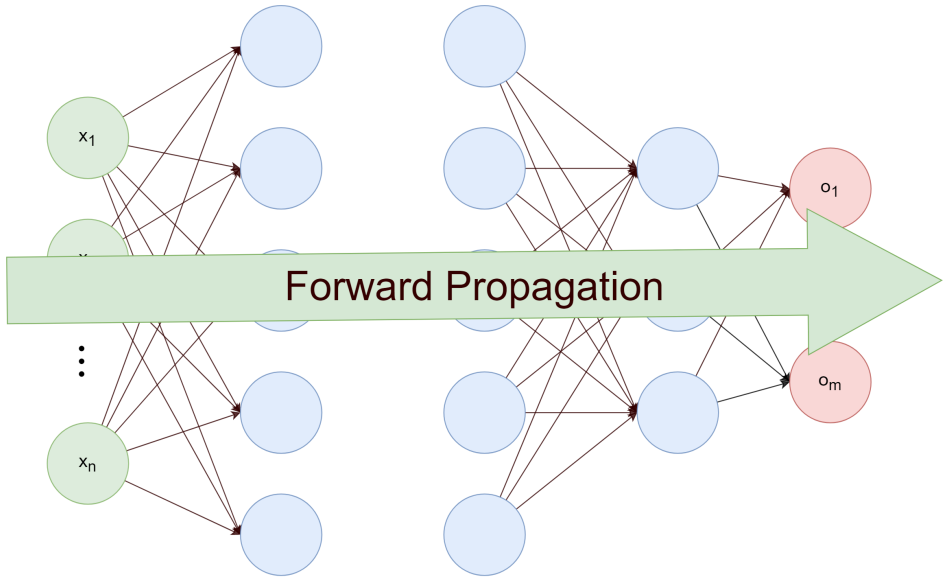


Figure: Forward pass

Backward Propagation

- Now we need to calculate $\nabla_W \mathcal{J}$.
- First idea:
  - Use analytical approach.
  - Write down derivatives on paper.
  - Find the close form of $\nabla_W \mathcal{J}$ (if it is possible to do so).
  - Implement this gradient as a function to work with.
  - Pros:
    - Fast
    - Exact
  - Cons:
    - Need to rewrite calculation for different architectures

Backward Propagation

- Second idea:
  - Using modular approach.
  - Computing the cost function consists of doing many operations.
  - We can build a computation graph for this calculation.
  - Each node will represent a single operation.

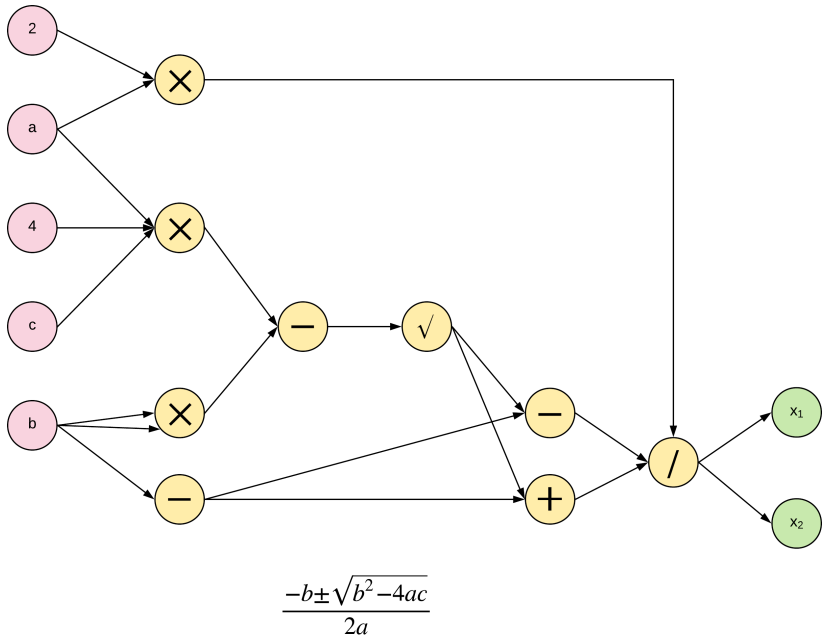
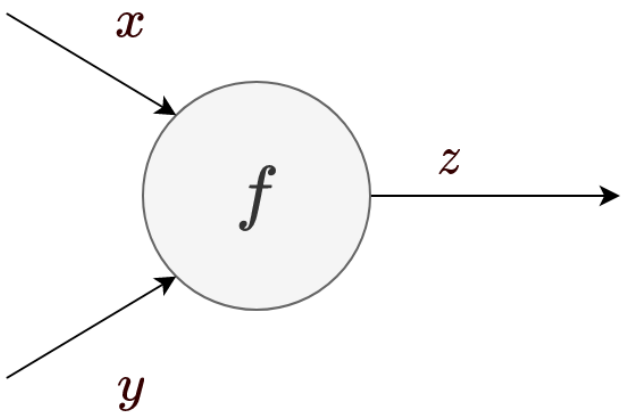


Figure: An example of computational graph, Source.

Backward Propagation

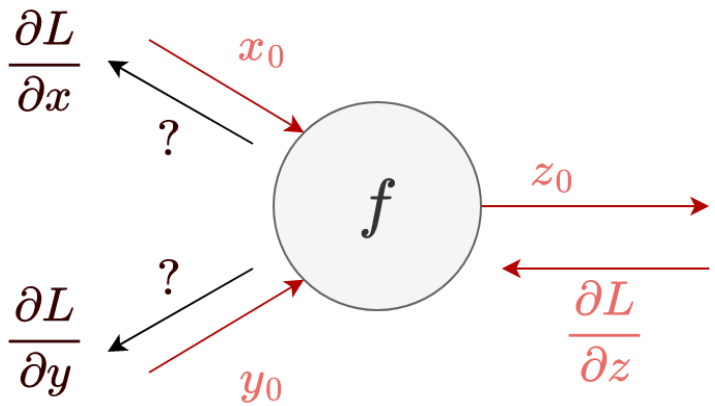
- In this approach if we know how to calculate gradient for single node or module, then we can find gradient with respect to each variables.
- Let's say we have a module as follow:



- It gets x and y as its input and returns $z = f(x, y)$ as its output.
- How to calculate derivative of loss with respect to module inputs?

Backward Propagation

- We know:
  - Module output for x_0 and y_0, let's call it z_0.
  - Gradient of loss with respect to module output at z_0, $\left(\frac{\partial L}{\partial z}\right)$.
- We want:
  - Gradient of loss with respect to module inputs at x_0 and y_0, $\left(\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}\right)$.

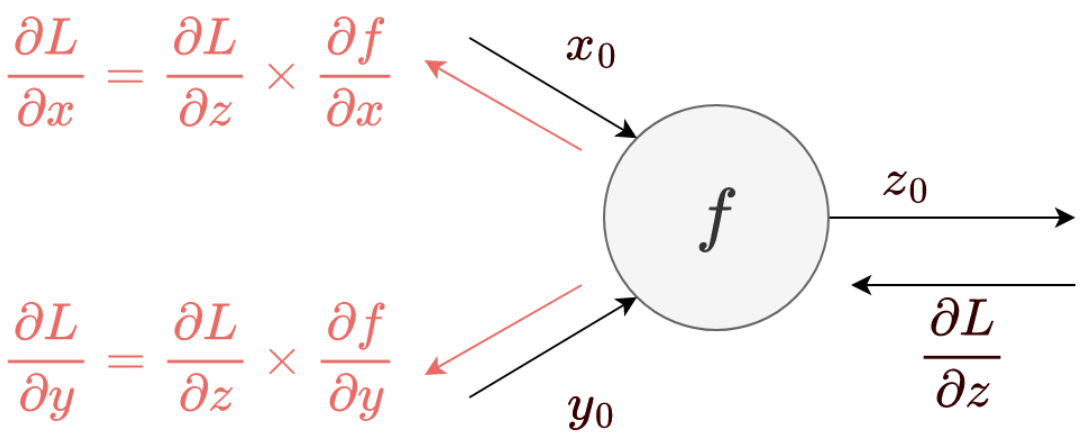


Backward Propagation

- We can use chain rule to do so.

Chain rule:

$$\left.\begin{aligned} z &= f(x,y) \\ L &= L(z) \end{aligned}\right\} \quad \Longrightarrow \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x}$$



Backward Propagation

- So after backward propagation we will have:
  - Gradient of loss with respect to each parameter.
  - We can apply gradient descent to update parameters.

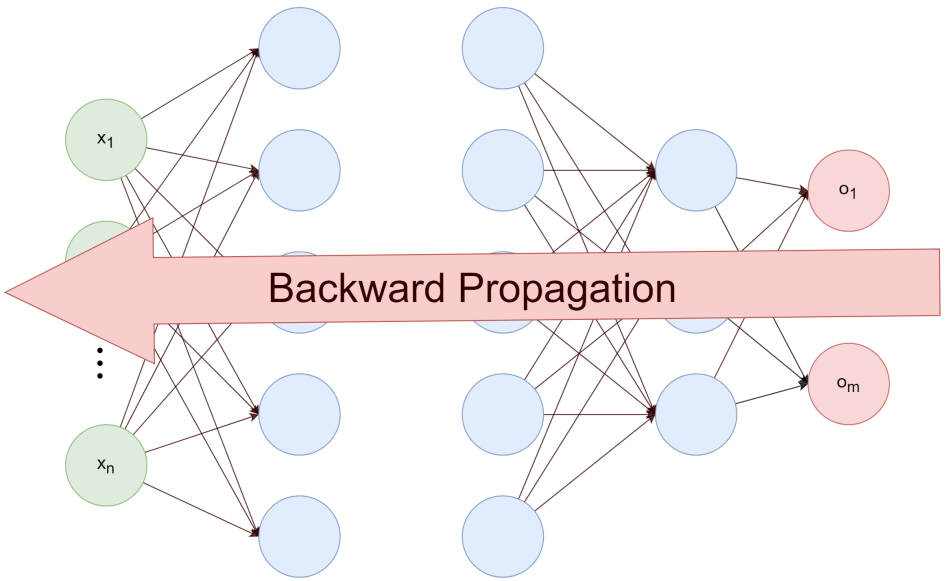


Figure: Backward pass

Loss functions

- So far you got familiar with gradient-based optimization.
- If $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{J}$, then we will update parameters with this simple rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g}$$

- But there is one question here, how to compute $\boldsymbol{g}$?
- Based on how we calculate $\boldsymbol{g}$ we will have different types of gradient descent:
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent

Various GD types

Recap:

Training cost function ($\mathcal{J}$) over a dataset usually is the average of loss function ($\mathcal{L}$) on entire training set, so for a dataset $\mathcal{D} = \{d_i\}_{i=1}^{n}$ we have:

$$\mathcal{J}(\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(d_i;\boldsymbol{\theta})$$

For example:

$$H(p,q) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)}\log(p(y_j^{(i)}))$$

$$\text{MSE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y_i-\hat{y}_i)^2$$

$$\text{MAE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}|(y_i-\hat{y}_i)|$$

Various GD types: Batch Gradient Descent

- In this type we use entire training set to calculate gradient.

Batch Gradient:

$$\boldsymbol{g} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- Using this method with very large training set:
  - Your data can be too large to process in your memory.
  - It requires a lot of processing to compute gradient for all samples.
- Using exact gradient may lead us to local minima.
- Moving noisy may help us get out of this local minimas.

Various GD types: Batch Gradient Descent

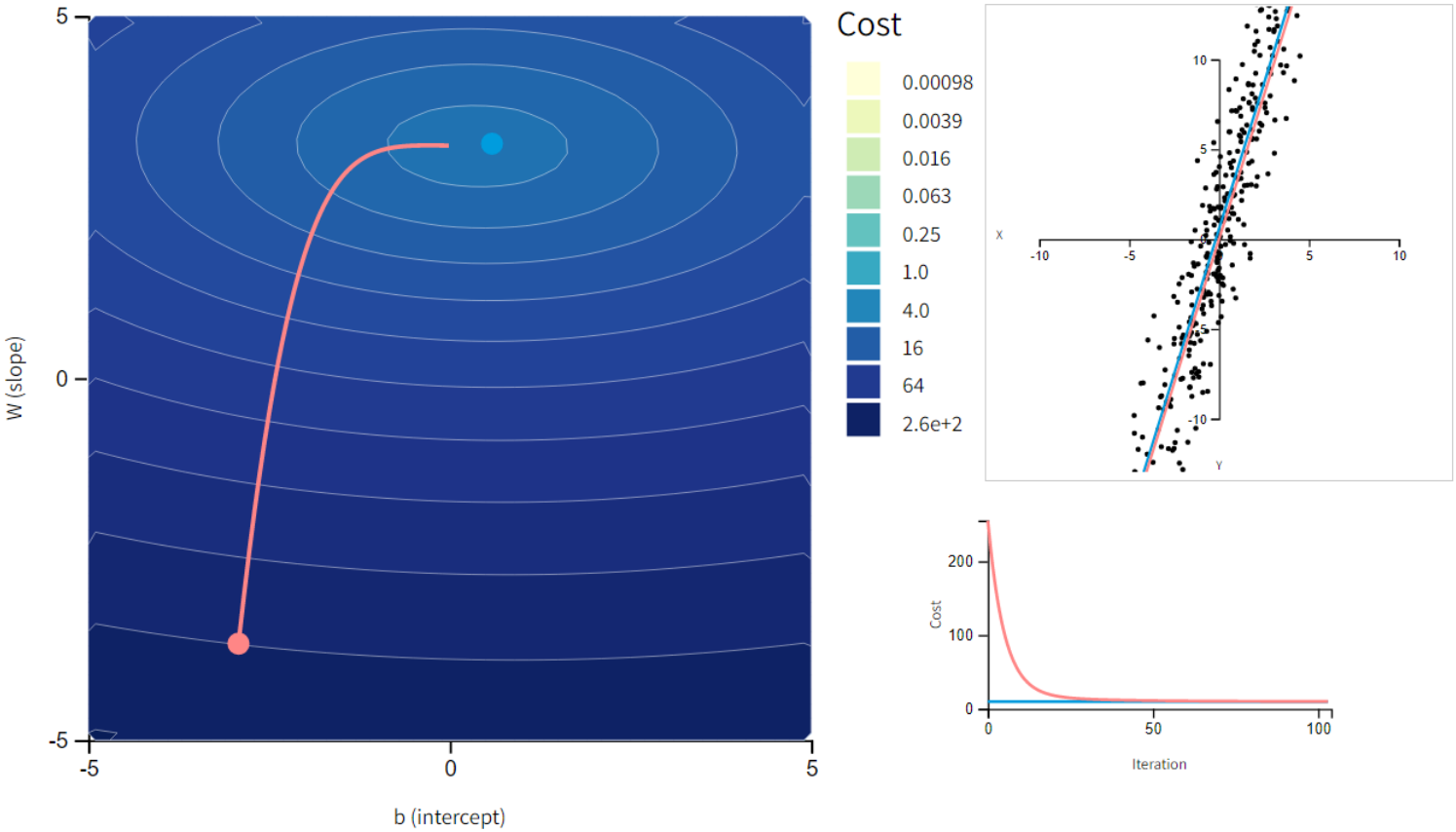


Figure: Optimization of parameters using BGD. Movement is very smooth [**?**].

Various GD types: Stochastic Gradient Descent

- Instead of calculating exact gradient, we can estimate it using our data.
- This is exactly what SGD does, it estimates gradient using only single data point.

Stochastic Gradient:

$$\hat{\boldsymbol{g}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- As we use an approximation of gradient, instead of gently decreasing, the cost function will bounce up and down and decrease only on average.
- This method is really computationally efficient cause we only need to calculate gradient for one point per iteration.

Various GD types: Stochastic Gradient Descent

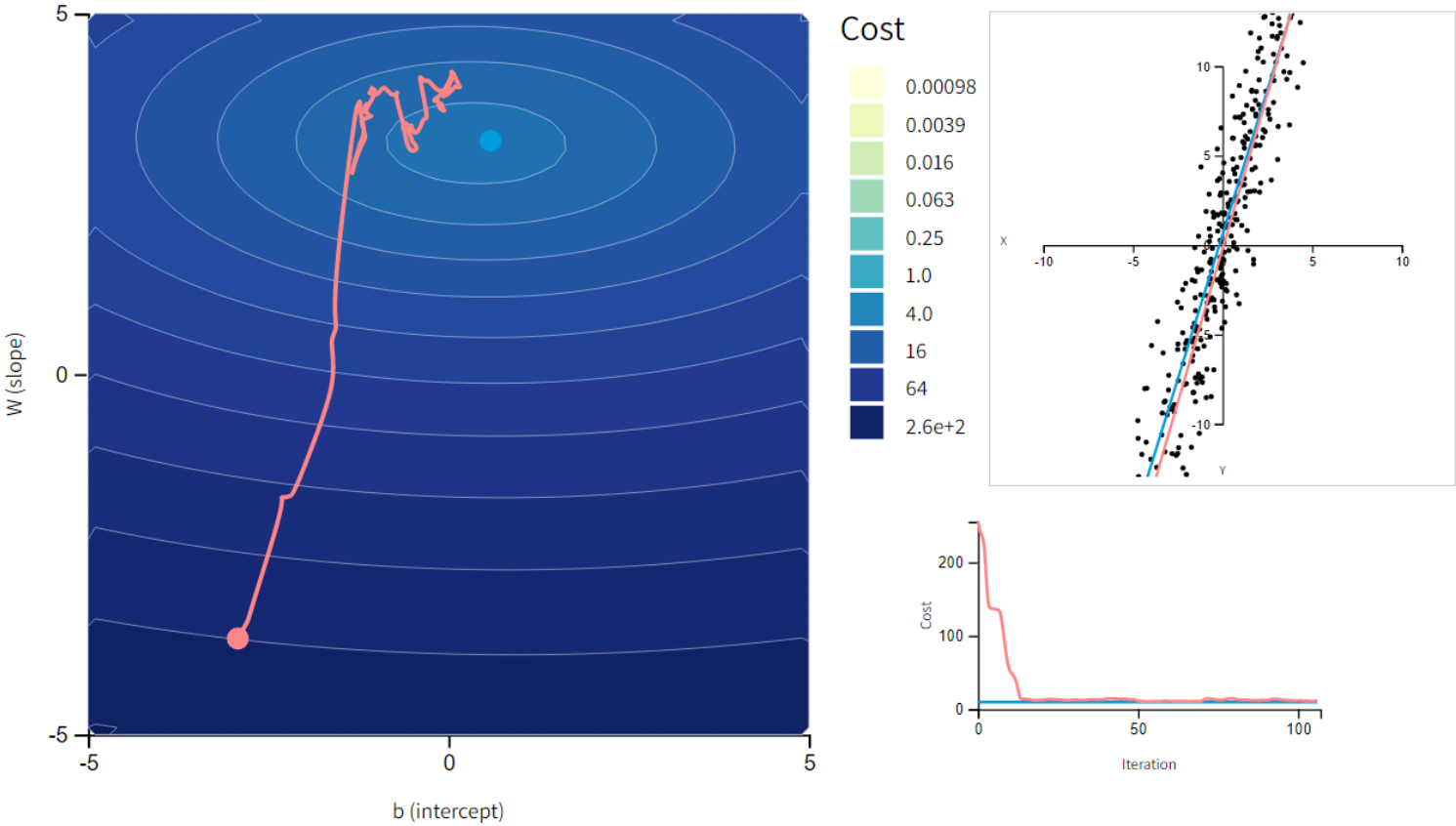


Figure: Optimization of parameters using SGD. As we expect, the movement is not that smooth [**?**].

Various GD types: Mini-Batch Gradient Descent

- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient:

$$\hat{\boldsymbol{g}} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d, \boldsymbol{\theta}), \quad \mathcal{B} \subset \mathcal{D}$$

- This is a better estimation than SGD.
- With this way we can get a performance boost from hardware optimization, especially when using GPUs.
- Batch size ($|\mathcal{B}|$) is a hyperparameter you need to tune.

Various GD types: Mini-Batch Gradient Descent

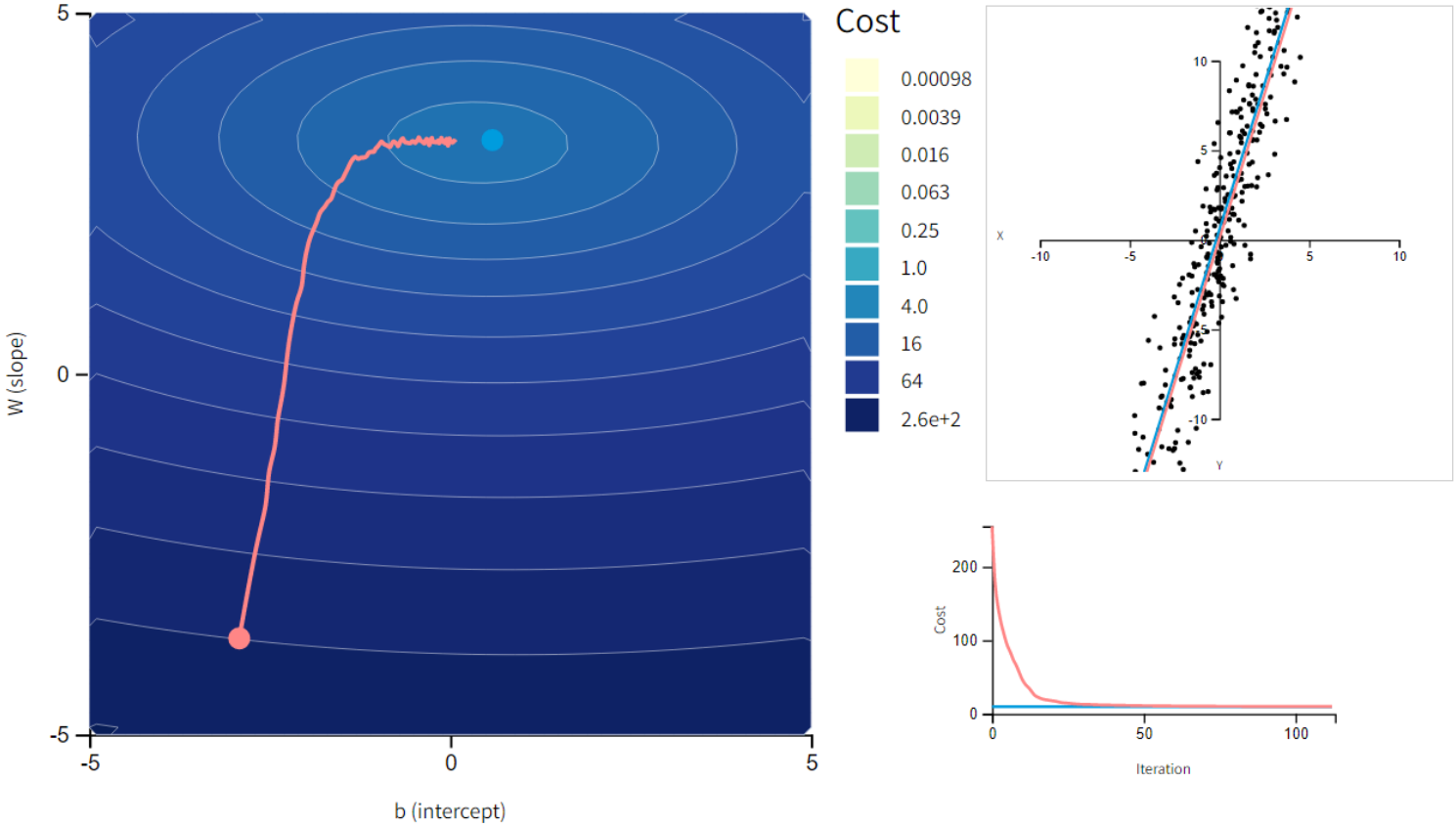


Figure: Optimization of parameters using MBGD. The movement is much smother than SGD and behaves like BGD [?].

Various GD types

- So we got familiar with different types of GD.
  - ✓ Batch Gradient Descent (BGD)
  - ✓ Stochastic Gradient Descent (SGD)
  - ✓ Mini-Batch Gradient Descent (MBGD)

- The most recommended one is MBGD, because it is computationally efficient.
- Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the generalization of your model.

Training MLPs

- So far we have learned about how MLPs work and how to update their parameters in order to perform better.
- But training MLPs is not that easy.
- You will face several different challenges in this procedure.
- In this section we will talk about this challenges and how to solve them.

Vanishing/Exploding Gradient

- The backpropagation algorithm propagates the error gradient while proceeding from the output layer to the input layer. The issue here is the magnitude of the cost function gradients through the layers...

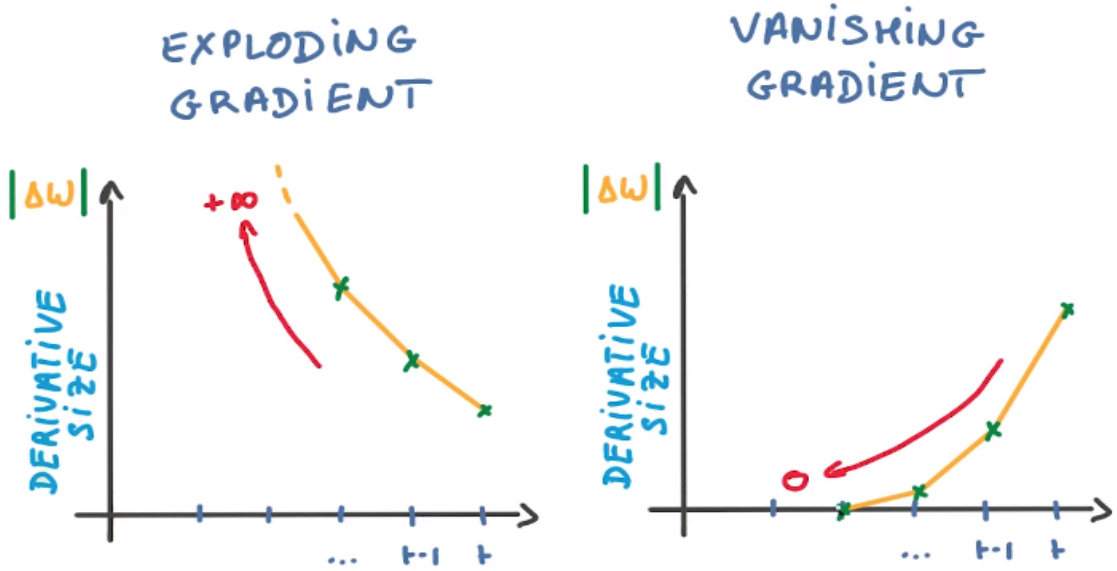


Figure: Vanishing/Exploding Gradient, Source

Vanishing/Exploding Gradient

- Vanishing
  - Gradients often get smaller as the algorithm progresses down. As a result, gradient descent updates do not effectively change the weights of the lower layer connections, and training never converges.
  - Make learning slow especially of front layers in the network.
- Exploding
  - Gradients can get bigger and bigger, so there are very large weight updates at many levels, causing the algorithm to diverge.
  - The model is not learning much on the training data therefore resulting in a poor loss.

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?
- A bad initialization may increase convergence time or even make optimization diverge.
- How to initialize?
  - Zero initialization
  - Random initialization

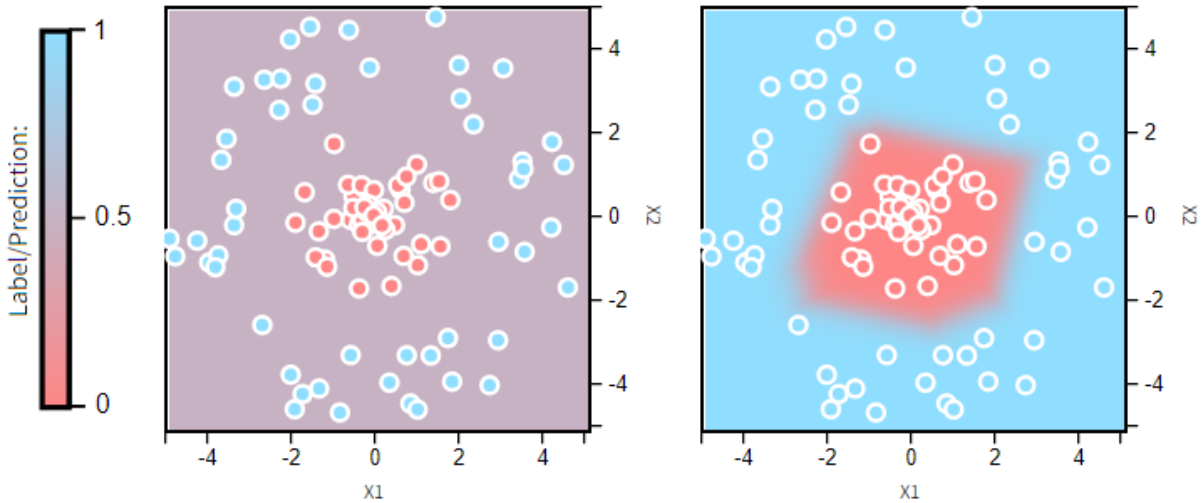


Figure: The output of a three layer network after about 600 epochs. (left) using a bad initialization method and (right) using an appropriate initialization [?].

Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = \mathbf{0}, \\ b^{[l]} = \mathbf{0} \end{cases}$$

- Simple but perform very poorly. (why?)
- Zero initialization will lead each neuron to learn the same feature
- This problem is known as network failing to break symmetry
- In fact any constant initialization suffers from this problem.

Weight Initialization: Random Initialization

- To use randomness in our initialization we can use uniform or normal distribution:

General Uniform Initialization:

$$\begin{cases} W^{[l]} \sim U(-r, +r), \\ b^{[l]} = 0 \end{cases}$$

General Normal Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2), \\ b^{[l]} = 0 \end{cases}$$

- But this is really crucial to choose r or σ properly.

More On Weight Initialization

- To read more about different weight initialization methods and deepen your understanding about how it can affect your network, visit here.

Problem: OverFitting in a Neural Network

- Why does overfitting happen in a neural network?
  - There are Too many free parameters.

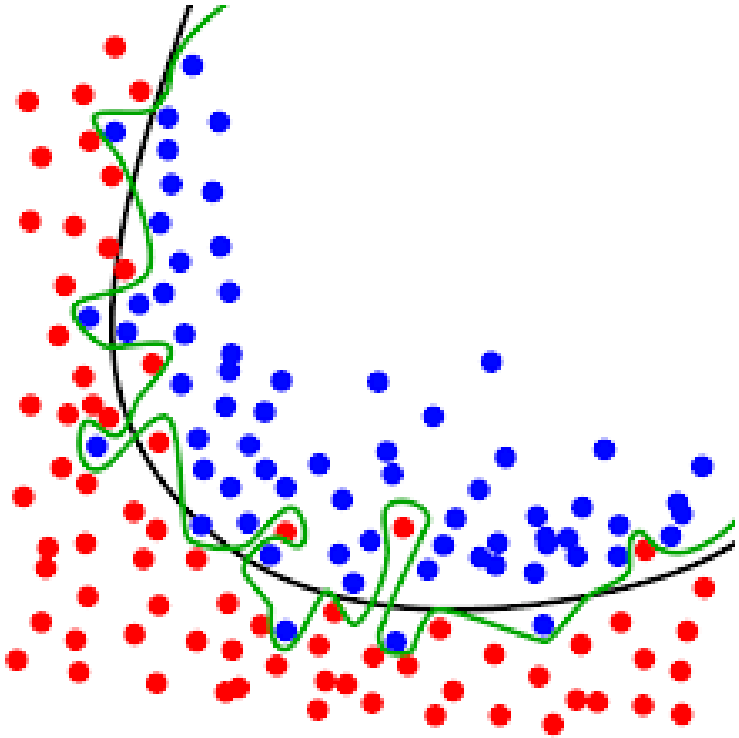


Figure: OverFitting in a neural network, Source

Early Stopping

- Stop the training procedure when the validation error is minimum.

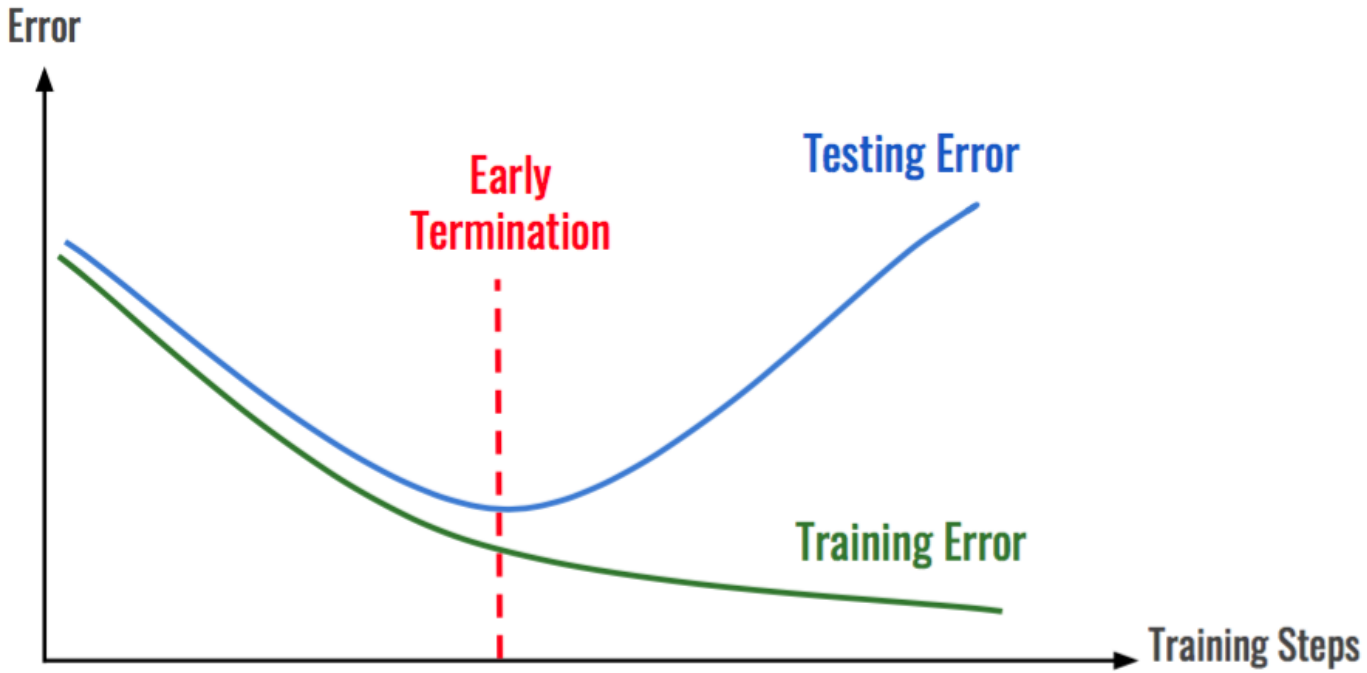


Figure: Early Stopping diagram, Source

Dropout: Training Time

- In each forward pass, randomly set some neurons to zero.
- The probability of dropping out for each neuron, which is called dropout rate, is a hyperparameter.
  - 0.5 is a common dropout rate.
- The probability of not dropping out is also called the keep probability.

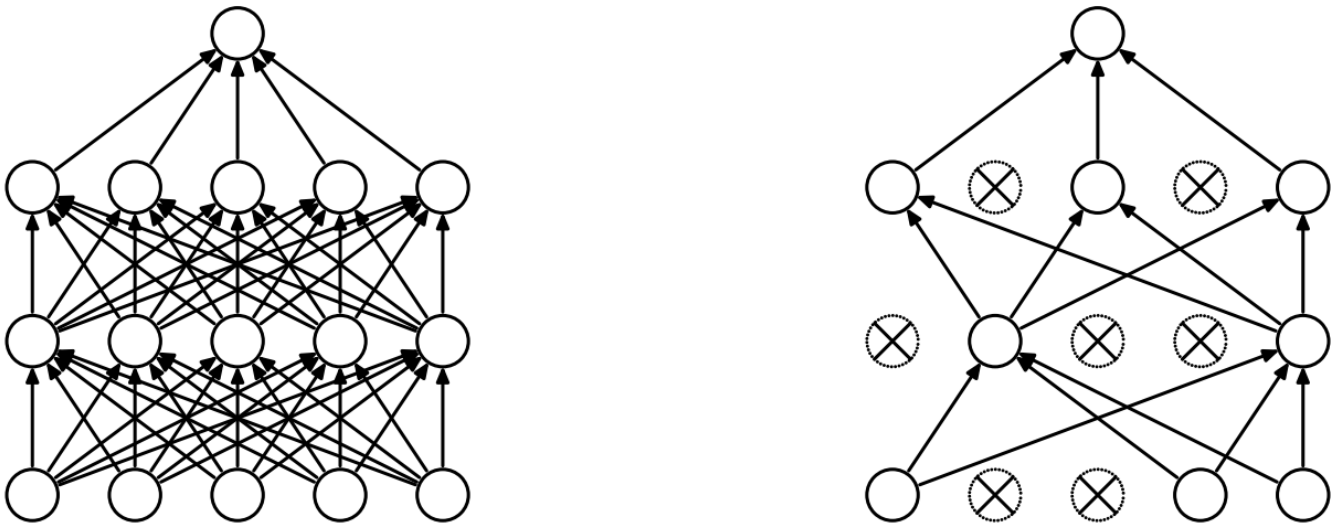


Figure: Behavior of dropout at training time, Source

Brief History of Computer Vision

- Before Deep Learning: Hand-Crafted Features

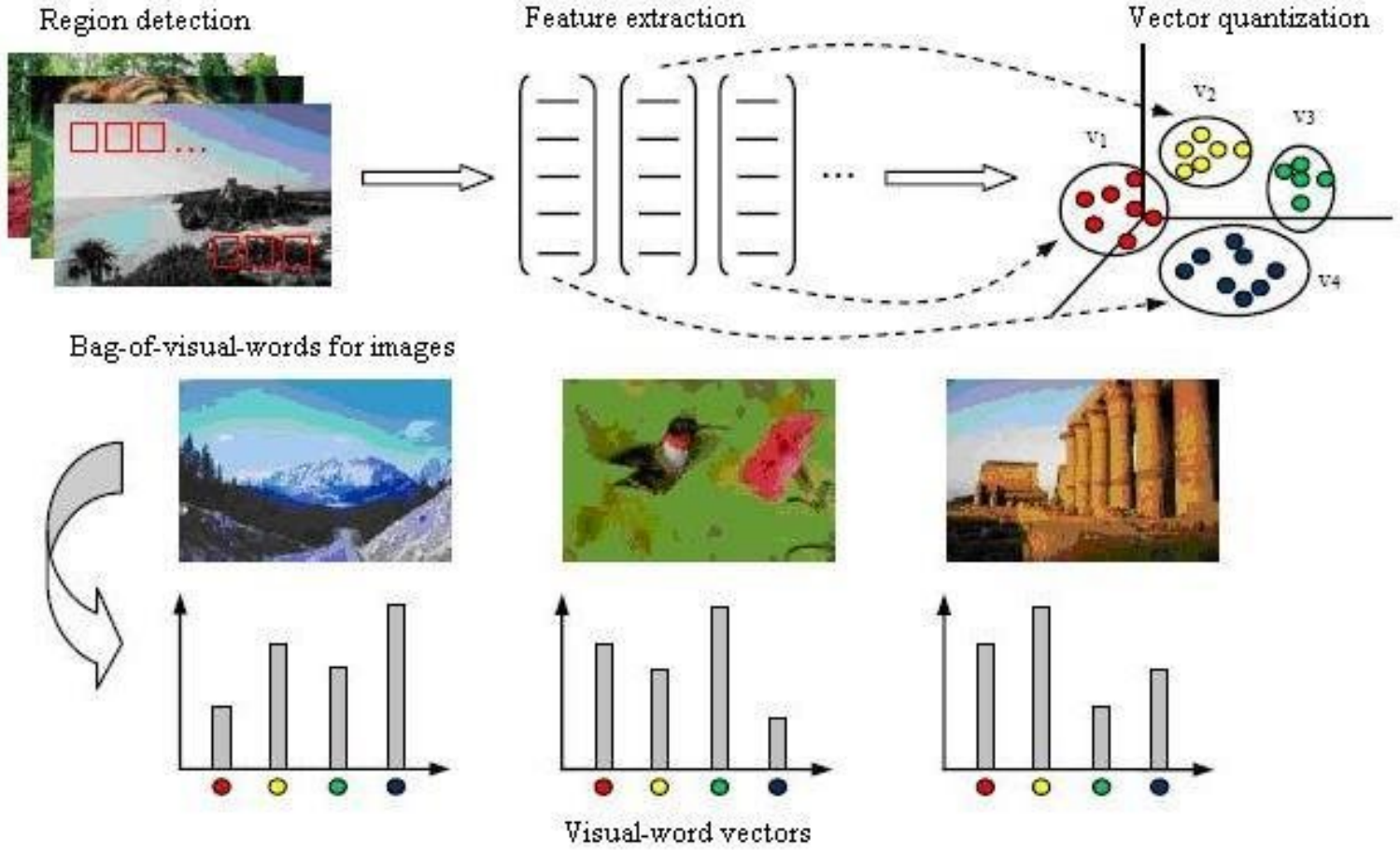
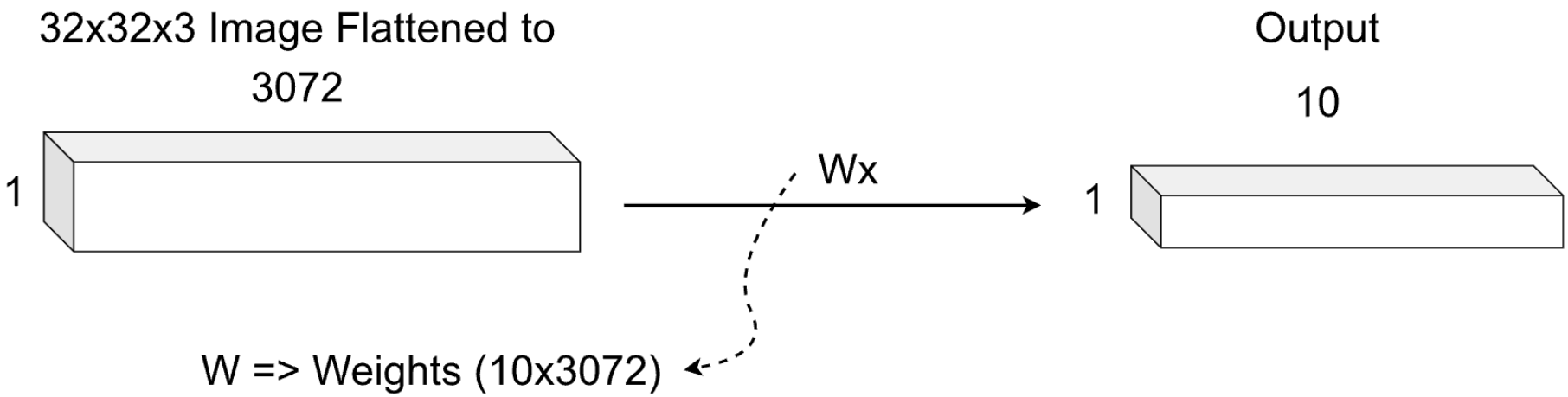


Figure: [?]

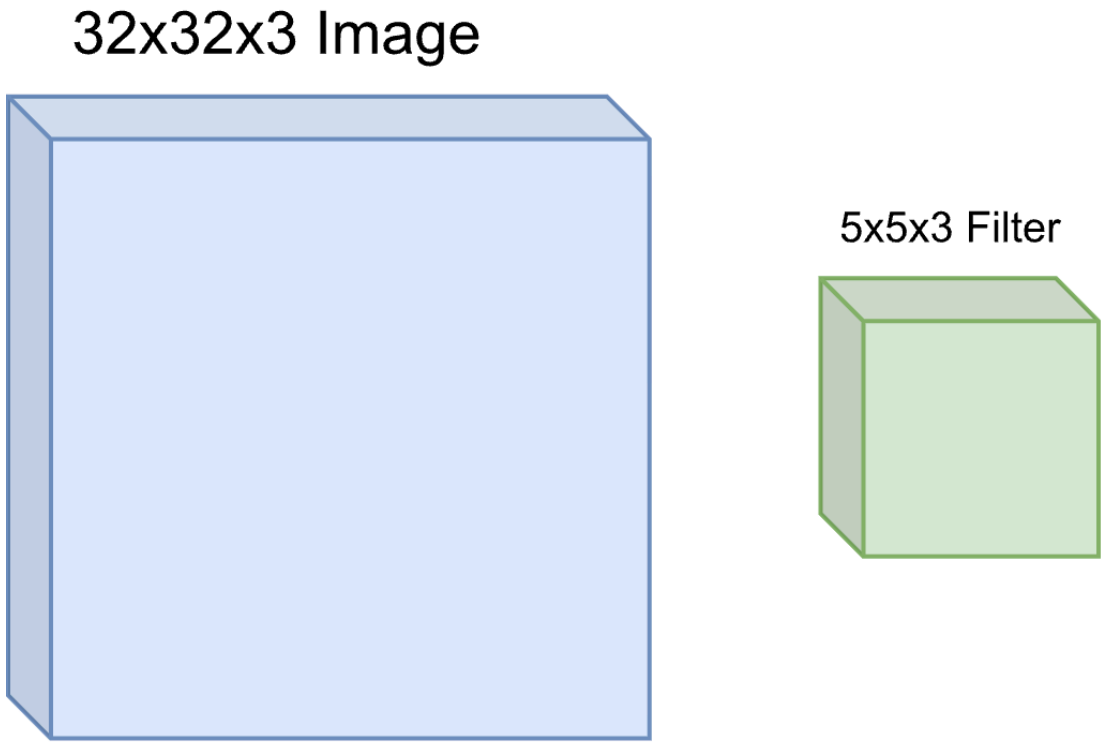
CNNs

- What we've been using:
  - Fully Connected Layers



CNNs

- What we're going to learn:
  - Convolutional Layer



What is Convolution and how does it work?

- This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel

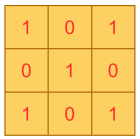
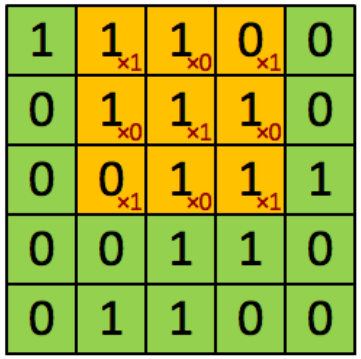


Figure: Convolving Kernel



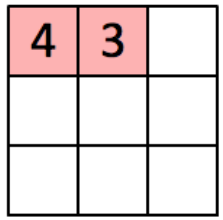


Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

Implementation

Finla Model

```
effnet = EfficientNetB0(weights='imagenet', include_top=False,
                        input_shape=(loadData.IMAGE_SIZE, loadData.IMAGE_SIZE, 3))
```

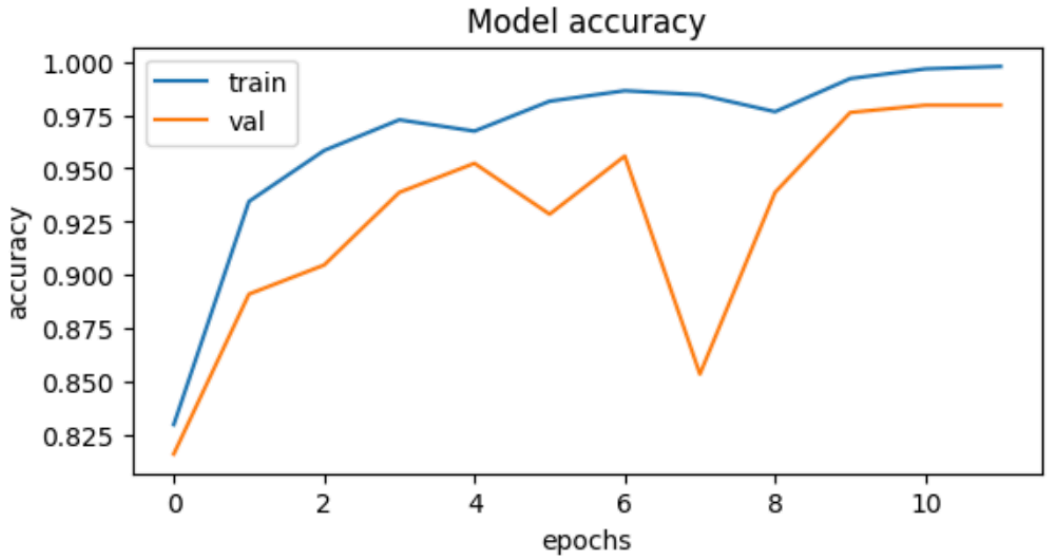
✓ 1.0s

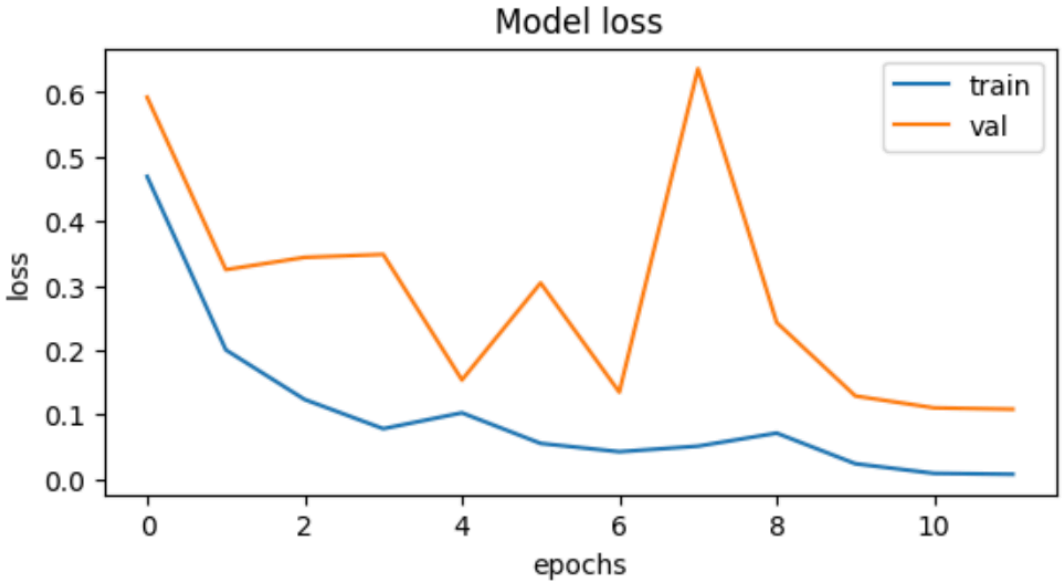
Designe Last Layer

```
model = effnet.output
model = tf.keras.layers.GlobalAveragePooling2D()(model)
model = tf.keras.layers.Dropout(rate=0.5)(model)
model = tf.keras.layers.Dense(4, activation='softmax')(model)
model = tf.keras.models.Model(inputs=effnet.input, outputs=model)
```

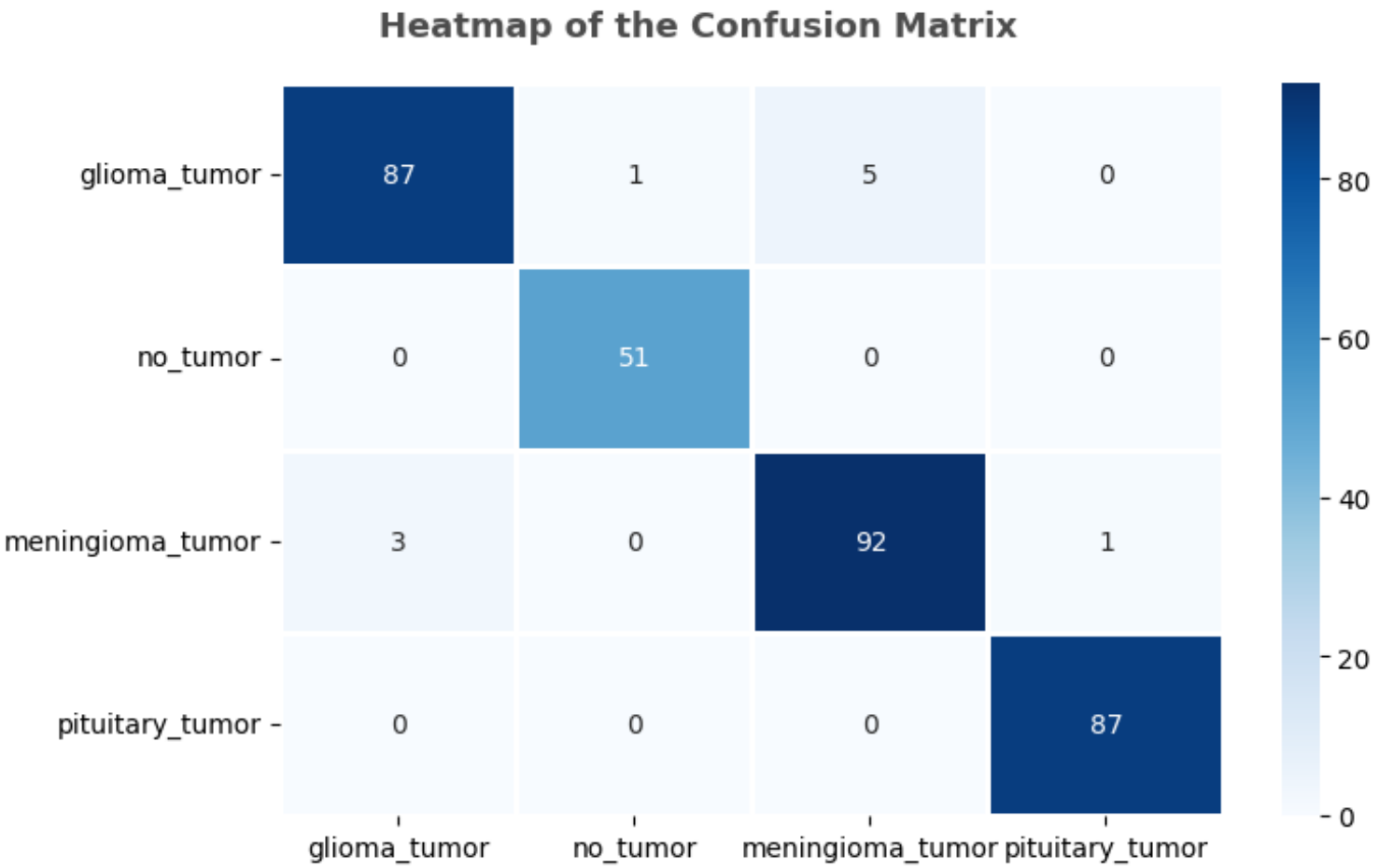
✓ 0.0s

Training


Model accuracy
train
val
accuracy
epochs
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0
2
4
6
8
10


Model loss
train
val
loss
epochs
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10

Confusion Matrix



Accuracy ≈ 97%!

```
test_loss, test_acc = model.evaluate(X_test,  y_test, verbose=2)
print("test accuracy : ", test_acc * 100, "%")
```

```
11/11 - 2s - loss: 0.0934 - accuracy: 0.9694 - 2s/epoch - 182ms/step
test accuracy :  96.94189429283142 %
```

Thank You!

Any Question?

Refrences

- Introduction To Machine Learning, Dr. SharifiZarChi Link
- Deep Learning for Vision Systems - Mohamed Elgendy - 2020 - Link
- CS231n: Convolutional Neural Networks for Visual Recognition Link
- Brain tumor intro - Mayo Clinic Link