Breast Cancer prediction using Neural Networks

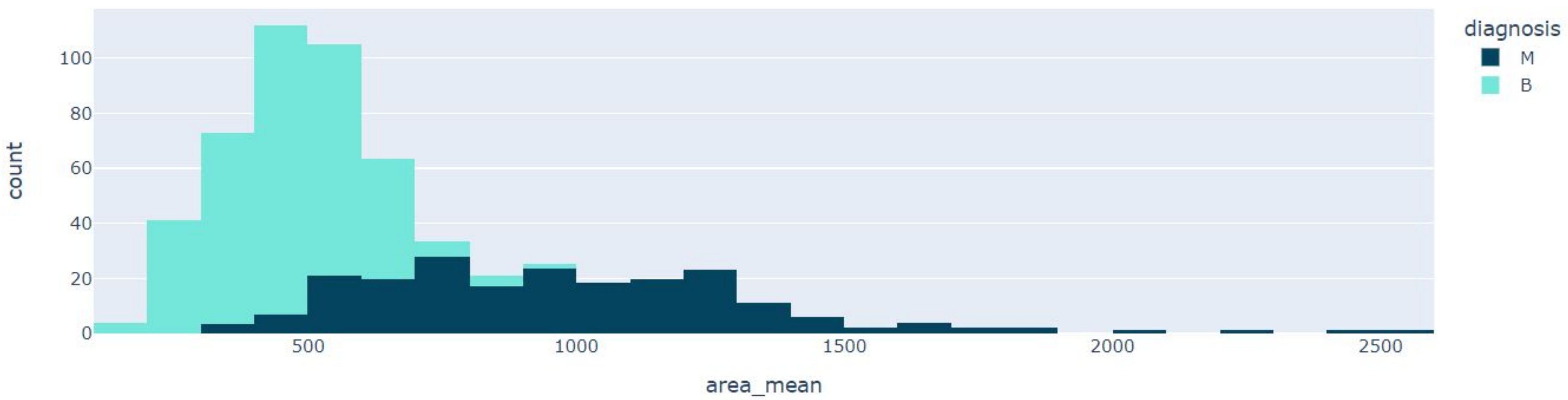
Ilija Doknić
Stefan Komarica

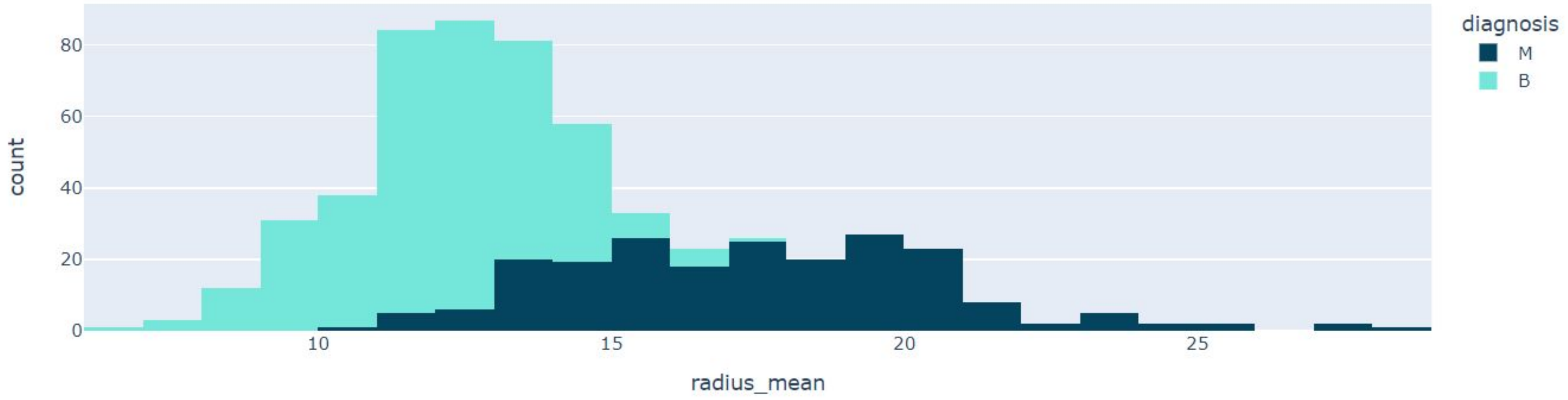
Dataset

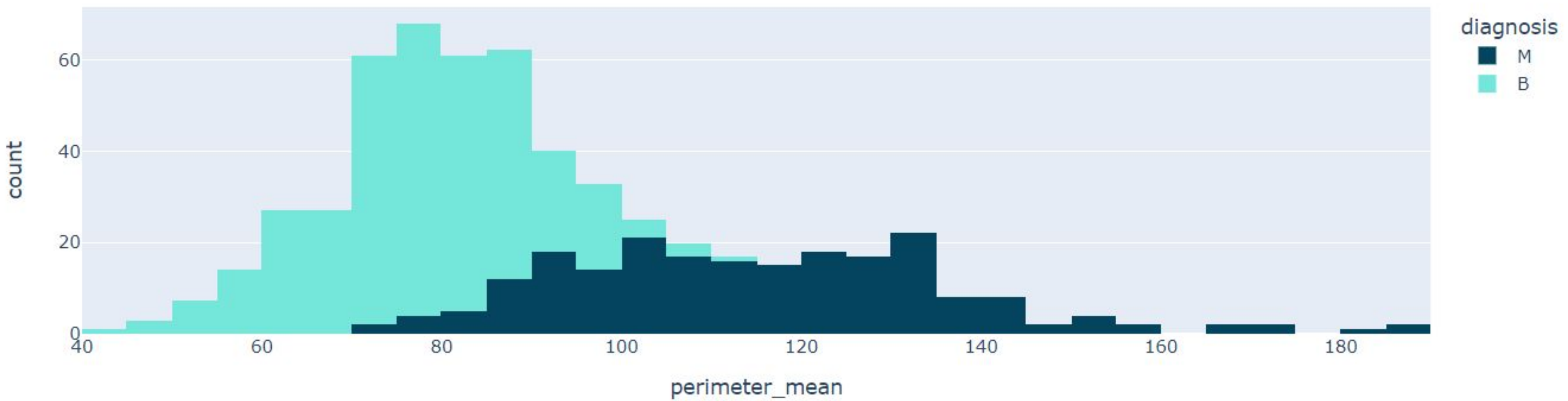
- Binary classification prediction for determining is breast cancer malignant or benign type
- Data taken from https://www.kaggle.com/datasets/yasserh/breast-cancer-dataset

Initial analysis

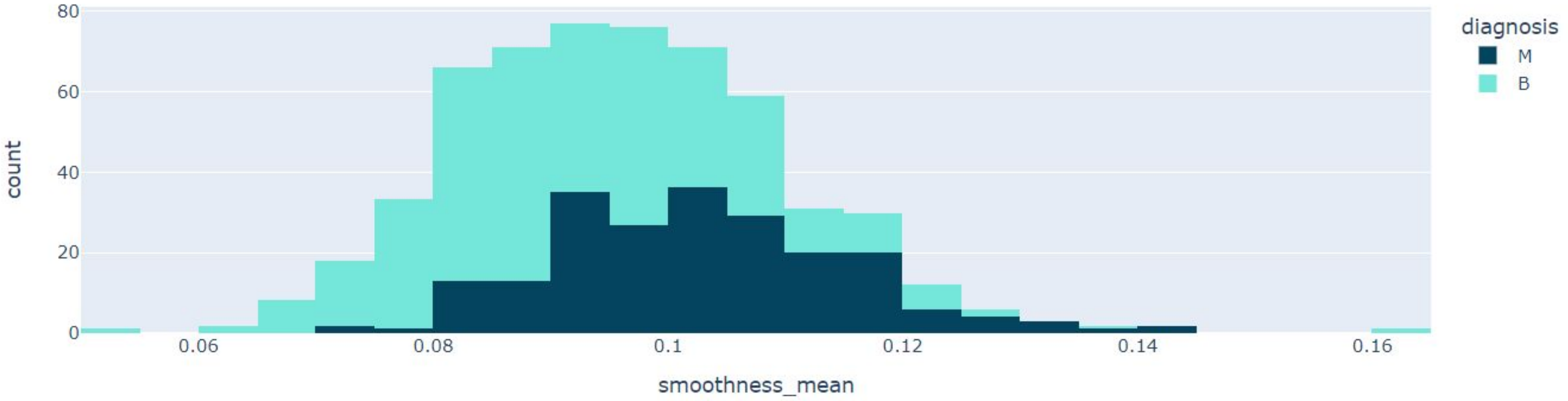
diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...
M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...
M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...
M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...
M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...
M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...


diagnosis
M
B
count
0
20
40
60
80
100
500
1000
1500
2000
2500
area_mean


diagnosis
M
B
count
0
20
40
60
80
10
15
20
25
radius_mean

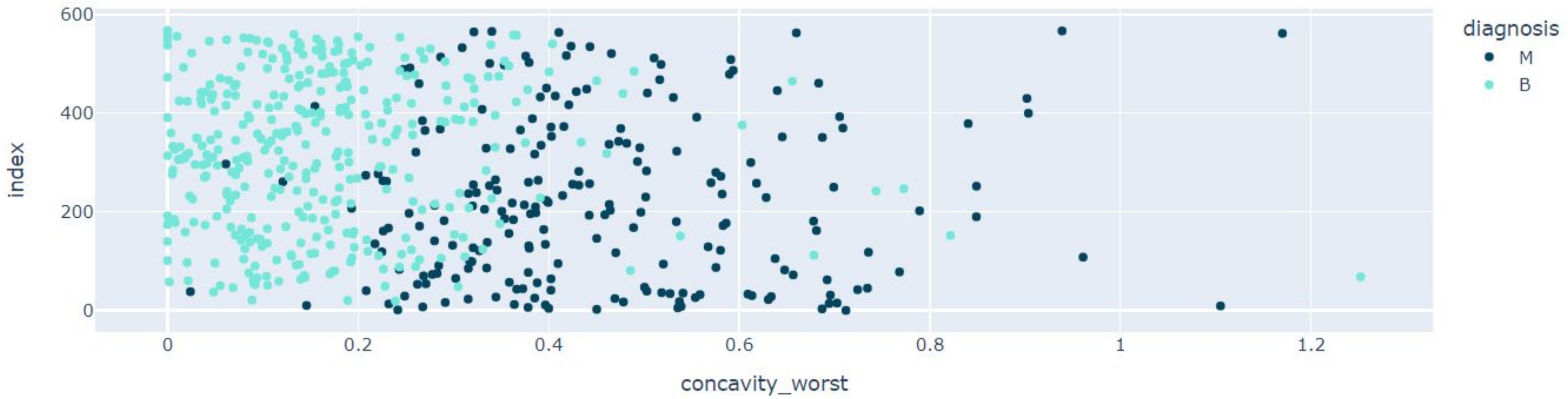


diagnosis
M
B
count
perimeter_mean
0
20
40
60
40
60
80
100
120
140
160
180

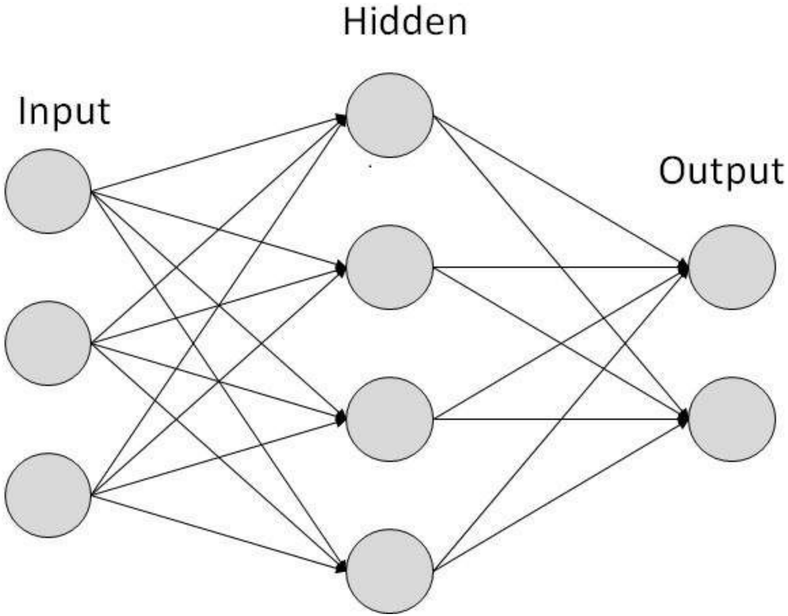


diagnosis
M
B
count
smoothness_mean
0
20
40
60
80
0.06
0.08
0.1
0.12
0.14
0.16

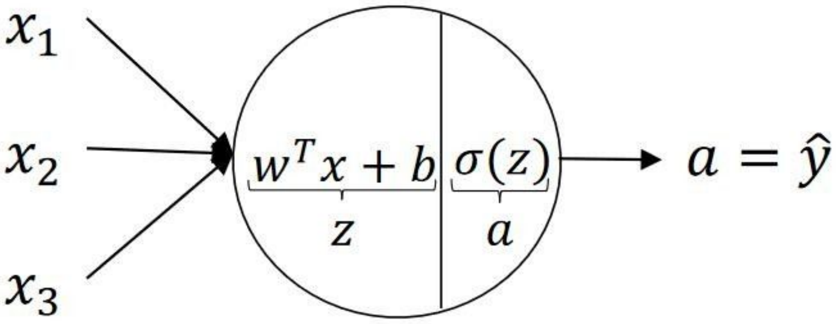

600
400
200
0
index
0.2
0.3
0.4
0.5
0.6
symmetry_worst
diagnosis
M
B


600
400
200
0
index
0
0.2
0.4
0.6
0.8
1
1.2
concavity_worst
diagnosis
M
B

Neural network



Single neuron



$$z = w^T x + b$$

$$a = \sigma(z)$$

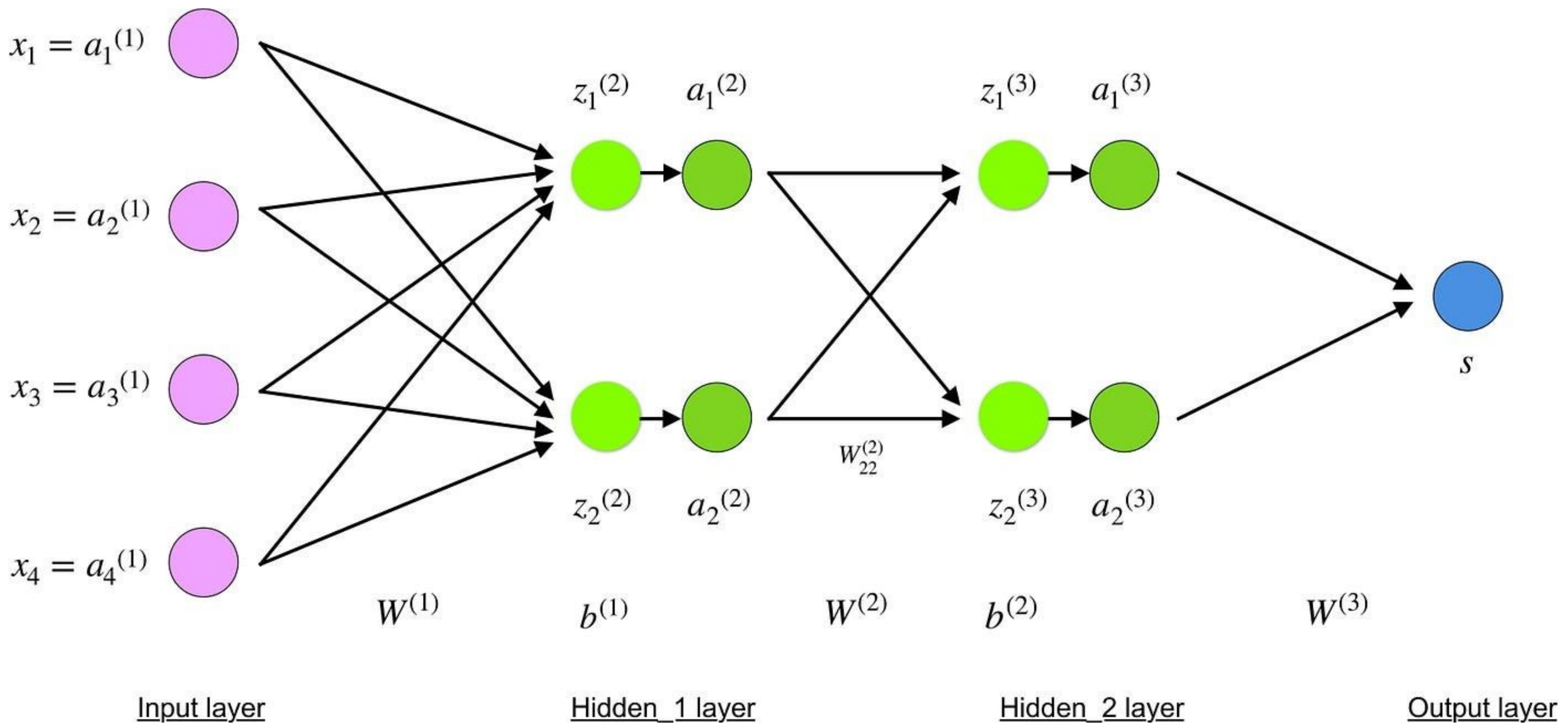
```python
class Dense:
  ...

  def forward(self,inputs):
    self.inputs = inputs
    self.output = np.dot(inputs,self.weights)+self.bias

class ReLU:
  def forward(self,inputs):
    self.inputs = inputs
    self.output = np.maximum(0, inputs)

class Sigmoid:
  def forward(self,inputs):
    self.output = 1/(1+np.exp(-np.array(inputs)))
```

LogLoss function

$$Log\ Loss(y,p) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1-y_i)\log(1-p_i))$$

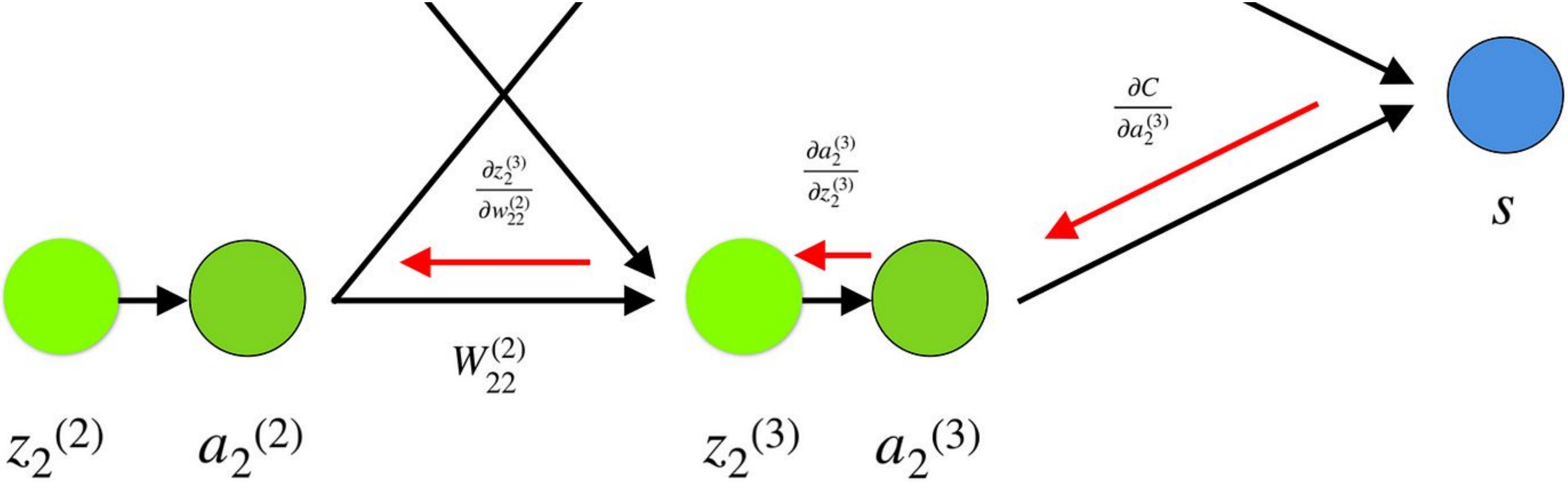
Implementation

```
class LogLoss(Loss):
  def forward(self,y_pred,y_true):
    y_pred = np.clip(y_pred, 1e-7, 1 - 1e-7)
    return (-1/len(y_true))*\
              (np.dot(y_true.T,np.log(y_pred))+\
               np.dot((1-y_true).T,np.log(1-y_pred)))
```



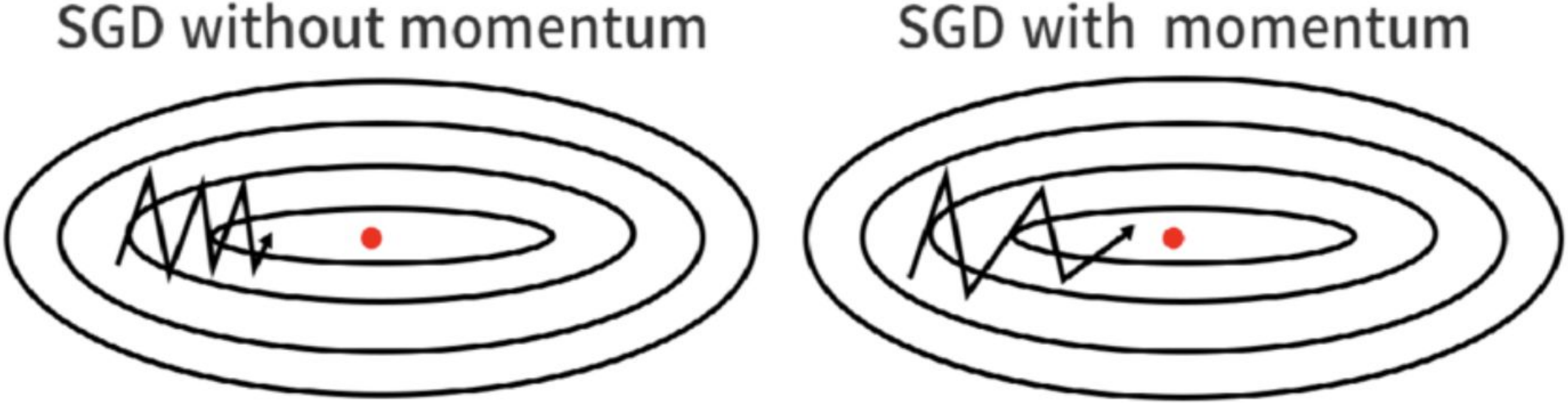
$x_1 = a_1^{(1)}$
$x_2 = a_2^{(1)}$
$x_3 = a_3^{(1)}$
$x_4 = a_4^{(1)}$
$z_1^{(2)}$
$a_1^{(2)}$
$z_2^{(2)}$
$a_2^{(2)}$
$z_1^{(3)}$
$a_1^{(3)}$
$z_2^{(3)}$
$a_2^{(3)}$
$W_{22}^{(2)}$
s
$W^{(1)}$
$b^{(1)}$
$W^{(2)}$
$b^{(2)}$
$W^{(3)}$
Input layer
Hidden_1 layer
Hidden_2 layer
Output layer

Back propagation



Implementation

```
class Dense:
    ...
  def backward(self,deriv_in):
    self.deriv_weights = np.dot(self.inputs.T, deriv_in)
    self.deriv_bias = np.sum(deriv_in,axis=0,keepdims=True)
    self.deriv_out = np.dot(deriv_in,self.weights.T

class Sigmoid:
    ...
  def backward(self,deriv_in):
    self.deriv_out = self.output*(1-self.output)*deriv_in
```


SGD without momentum
SGD with momentum

Adam optimization algorithm

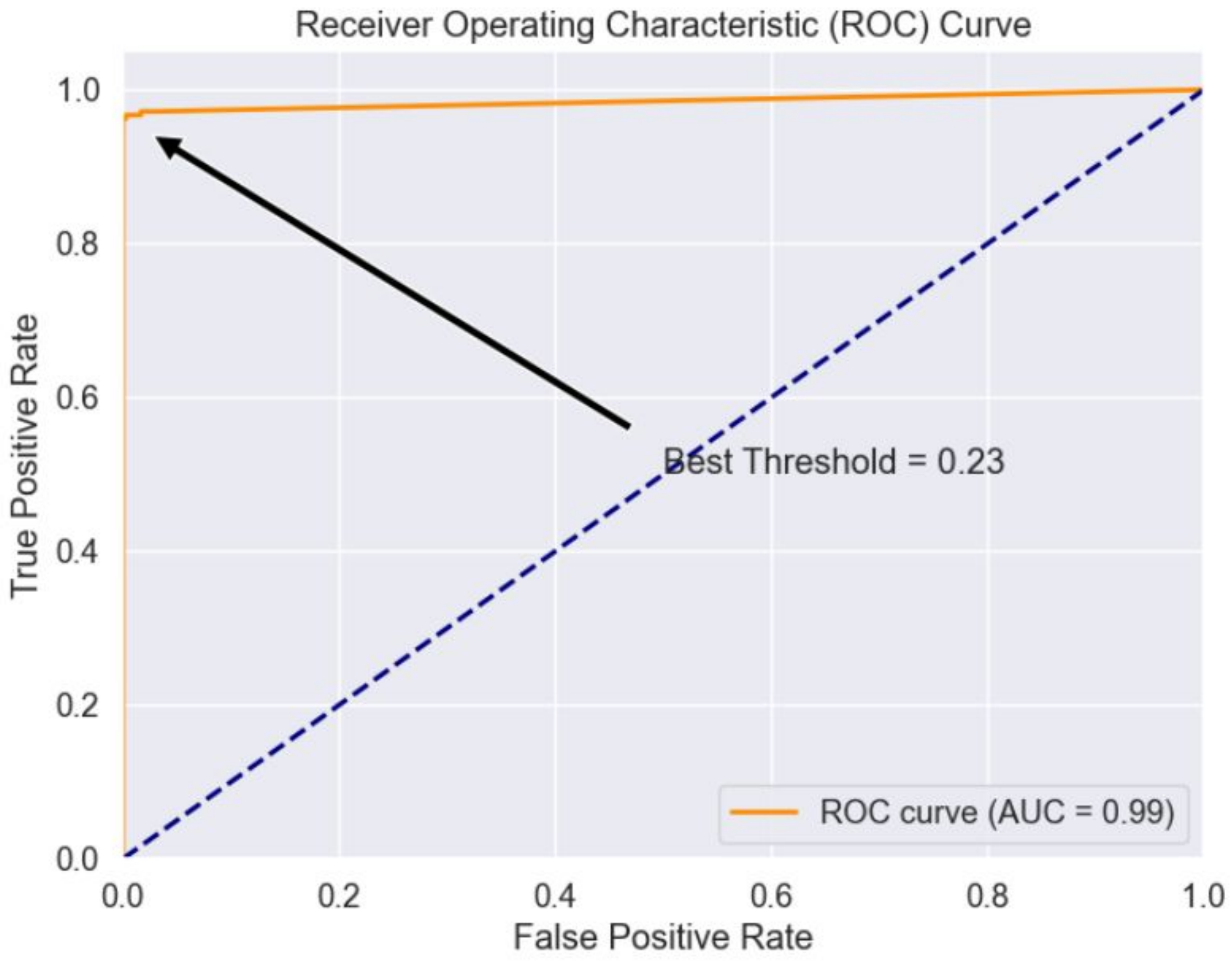
```
class Adam():
  def update_params(self,layer):

    if not hasattr(layer,'weight_cache'):
      layer.weight_cache = np.zeros_like(layer.weights)
      layer.weight_momentum = np.zeros_like(layer.weights)
      layer.bias_cache = np.zeros_like(layer.bias)
      layer.bias_momentum = np.zeros_like(layer.bias)
    # Momentum part of Adam
    layer.weight_momentum = self.beta1*layer.weight_momentum + (1-self.beta1)*layer.deriv_weights
    layer.bias_momentum = self.beta1*layer.bias_momentum + (1-self.beta1)*layer.deriv_bias

    weight_momentum_corrected = layer.weight_momentum/(1 - self.beta1**(self.iteration+1))
    bias_momentum_corrected = layer.bias_momentum/(1 - self.beta1**(self.iteration+1))

    layer.weight_cache = self.beta2*layer.weight_cache + (1-self.beta2)*layer.deriv_weights**2
    layer.bias_cache = self.beta2*layer.bias_cache + (1-self.beta2)*layer.deriv_bias**2

    weight_cache_correctted = layer.weight_cache/(1-self.beta2**(self.iteration+1))
    bias_cache_correctted = layer.bias_cache/(1-self.beta2**(self.iteration+1))

    layer.weights -= self.current_learning_rate*weight_momentum_corrected/(np.sqrt(weight_cache_correctted+self.epsilon))
    layer.weights -= self.current_learning_rate*bias_momentum_corrected/(np.sqrt(bias_cache_correctted+self.epsilon))

  def post_update_params(self):
    self.iteration +=1
```

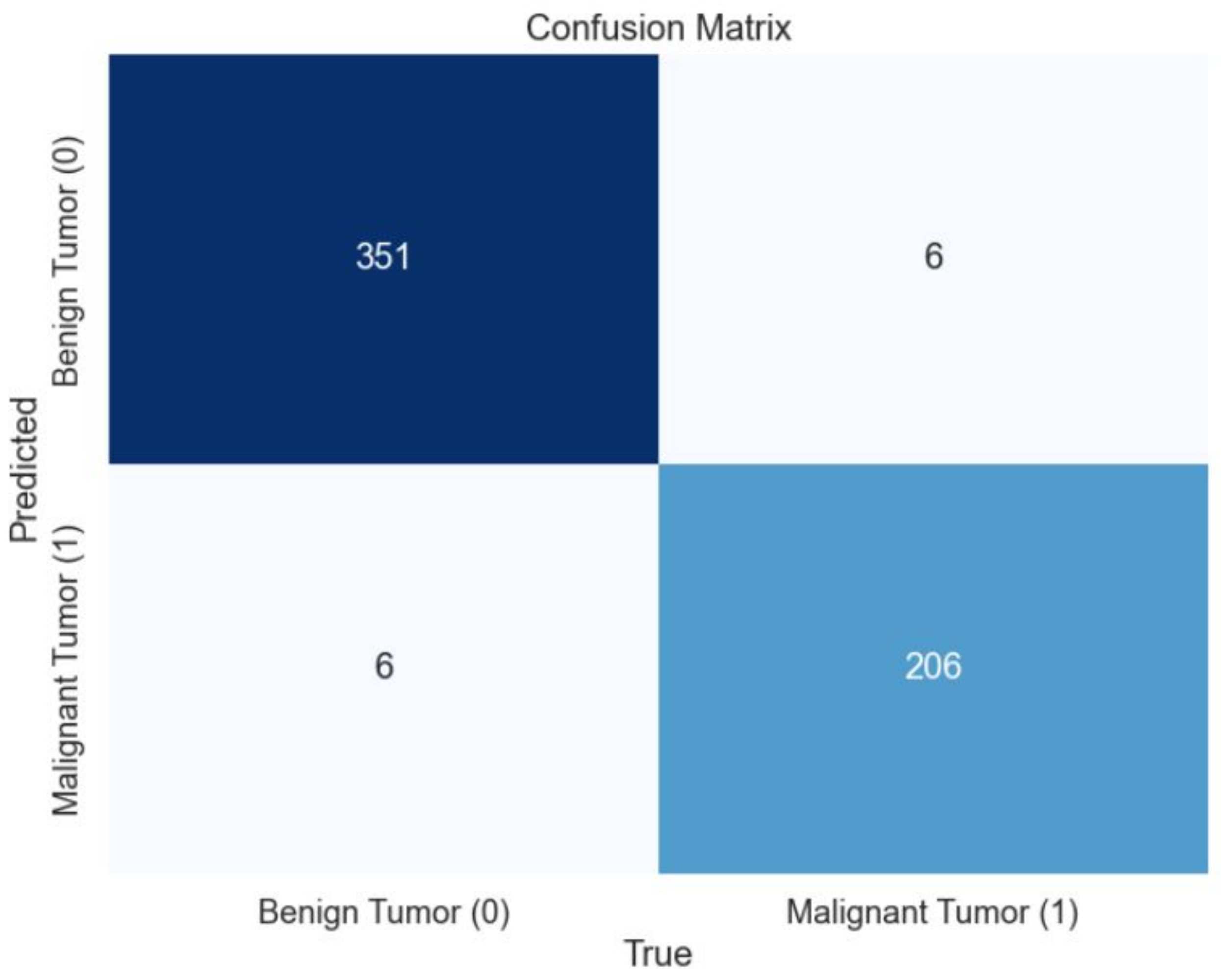
Repeat multiple times

```
for epoch in range(21):
    dense_1.forward(X_test)
    relu_1.forward(dense_1.output)
    dense_2.forward(relu_1.output)
    relu_2.forward(dense_2.output)
    dense_3.forward(relu_2.output)
    sigmoid.forward(dense_3.output)
    current_loss = loss.forward(sigmoid.output, y_test)
    y_pred_test = np.array(sigmoid.output > 0.5,dtype = int)

    dense_1.forward(X_train)
    relu_1.forward(dense_1.output)
    dense_2.forward(relu_1.output)
    relu_2.forward(dense_2.output)
    dense_3.forward(relu_2.output)
    sigmoid.forward(dense_3.output)
    current_loss = loss.forward(sigmoid.output, y_train)
    y_pred = np.array(sigmoid.output > 0.5,dtype = int)

    loss.backward(sigmoid.output, y_train)
    sigmoid.backward(loss.deriv_out)
    dense_3.backward(sigmoid.deriv_out)
    relu_2.backward(dense_3.deriv_out)
    dense_2.backward(relu_2.deriv_out)
    relu_1.backward(dense_2.deriv_out)
    dense_1.backward(dense_2.deriv_out)

    optimizer.update_params(dense_3)
    optimizer.update_params(dense_2)
    optimizer.update_params(dense_1)
```

Accuracy and loss results

```
Epoch: 1, Accuracy: 0.8451, Val-Accuracy: 0.8392, Loss: 0.6931
Epoch: 2, Accuracy: 0.6268, Val-Accuracy: 0.6294, Loss: 0.6606
Epoch: 3, Accuracy: 0.6268, Val-Accuracy: 0.6294, Loss: 0.6603
Epoch: 4, Accuracy: 0.6268, Val-Accuracy: 0.6294, Loss: 0.6598
Epoch: 5, Accuracy: 0.6268, Val-Accuracy: 0.6294, Loss: 0.6583
Epoch: 6, Accuracy: 0.6268, Val-Accuracy: 0.6294, Loss: 0.6507
Epoch: 7, Accuracy: 0.6948, Val-Accuracy: 0.6853, Loss: 0.5904
Epoch: 8, Accuracy: 0.9366, Val-Accuracy: 0.9371, Loss: 0.3593
Epoch: 9, Accuracy: 0.9413, Val-Accuracy: 0.951, Loss: 0.2293
Epoch: 10, Accuracy: 0.9742, Val-Accuracy: 0.972, Loss: 0.1778
Epoch: 11, Accuracy: 0.9554, Val-Accuracy: 0.958, Loss: 0.1539
Epoch: 12, Accuracy: 0.9718, Val-Accuracy: 0.979, Loss: 0.1253
Epoch: 13, Accuracy: 0.9765, Val-Accuracy: 0.979, Loss: 0.1161
Epoch: 14, Accuracy: 0.9765, Val-Accuracy: 0.979, Loss: 0.108
Epoch: 15, Accuracy: 0.9742, Val-Accuracy: 0.979, Loss: 0.1009
Epoch: 16, Accuracy: 0.9789, Val-Accuracy: 0.986, Loss: 0.095
Epoch: 17, Accuracy: 0.9789, Val-Accuracy: 0.986, Loss: 0.0867
Epoch: 18, Accuracy: 0.9836, Val-Accuracy: 0.986, Loss: 0.0813
Epoch: 19, Accuracy: 0.9836, Val-Accuracy: 0.986, Loss: 0.078
Epoch: 20, Accuracy: 0.9812, Val-Accuracy: 0.986, Loss: 0.0755
Epoch: 21, Accuracy: 0.9836, Val-Accuracy: 0.986, Loss: 0.0736
```


Receiver Operating Characteristic (ROC) Curve
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
Best Threshold = 0.23
ROC curve (AUC = 0.99)


Confusion Matrix
Predicted
Benign Tumor (0)
Malignant Tumor (1)
351
6
6
206
Benign Tumor (0)
Malignant Tumor (1)
True