Table of Contents

Introduction 3

Descriptive questions and more familiarity with the general and important concepts of neural network training 5

Implementation exercise 14

Step-by-step description of the functions used to implement the project 17

Conclusion and analysis of results 39

Introduction

The topic of training neural networks and machine learning systems has been researched by experts in the field of artificial intelligence since the beginning, and researchers who work in this field should update their information daily to be aware of new concepts and algorithms.

The objectives of this research activity:

1- Getting to know more about the general and important concepts of neural network training:

1. The concept of breaking the symmetry in neural networks

2. Theoretical basis of weight decay in L2 Regularization

3. Research on Huber's error function and its advantages and disadvantages

2- Implementation exercise:

1. Receiving the Fashion MNIST dataset and implementing the classification neural network

2. Dividing the data in the dataset into three parts: Training, validation, testing

3. Examining the output results of the neural network with different numbers of neurons in the hidden layer

4. Coding of different optimization and regularization methods

5. Comparison of accuracy and speed of different optimization methods

6. Implementing different methods and comparing their results and outputs

Different optimization methods for implementation:

- Stochastic Gradient Descent
- Momentum
- Adagrad
- RMSProp
- Adam

Different regularization methods for implementation:

- L1 Regularization
- L2 Regularization

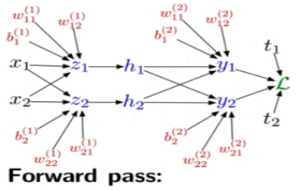
Descriptive questions and getting to know more about the general and important concepts of neural network training:

1. What does Breaking the Symmetry mean in neural networks? Explain fully.

In neural networks, the training process should not be started with initial weights of 0 (zero) or a fixed number for all neurons.

To better examine this concept, it is better to take a look at the neural network training process using the error backpropagation algorithm:

Multilayer Perceptron (multiple outputs):



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum (y_k - t_k)^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}} (y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k} h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

$$\overline{z_i} = \overline{h_i} \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i} x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

As you can see above, at the end of the decision of the neural network and at the beginning of the training process, the error that we receive at the end is a specific error in the form of a number, and this number is the same for neurons y1 and y2 in is considered But in the end, the thing that causes these two neurons to receive different derivatives is their weights. If the initial value of the two weights is defined in the same way, they will be trained in the same way, and in the end, their influence will be lost. The same process occurs in the entire network in the same way, and unfortunately, a general symmetry is created in the neural network, and this makes all the neurons act the same. For example, a neural network with 10 layers and 100 neurons in each layer actually becomes a neural network with 10 layers and 1 neuron in each layer, because all the neurons in each layer are trained in the same way. And all of them do not act differently and act exactly the same.

The solution is to set the initial weights completely randomly in order to destroy this symmetry (Breaking the Symmetry).

For example, we can use a special initializer, or after randomizing the weights, divide them all by the weight's standard deviation to get normal numbers for the weights.

In this case, in the training process, neurons receive different derivatives and are trained in different ways, and finally, each one will be more effective in the decision-making process.

2- Intuitively explain the basis of Weight decay in L2 Regularization. Mathematically, what happens that this method can cause better adjustment of parameters and better generalizability of the model?

One of the common problems in training neural networks is the possibility of overfitting. To date, many methods or techniques have been developed to prevent this from happening, one of which is to use regularization methods, which is explained in the answer to this question, the L2 Regularization method.

Intuitive explanation:

During the training process, after reaching a relatively favorable result on the training data, the model does not experience a better result by changing other parameters such as changing the slope of the obtained line. At this stage, if there is a limit on the value of the network weights is not applied, the model tries to make the weights of the network several times bigger in order to achieve a better result on the training data in order to make a favorable decision on the training data with higher confidence, because changing the slope of the line It doesn't get a better result, but the simultaneous multiplication of the weights of the network makes it obtain a much higher percentage of confidence and accuracy on the training data, and this is the most important reason for the phenomenon of overfitting.

This problem can be solved by adding a restriction on the weights of the network:

This restriction is applied to the neural network by adding (the sum of the network weights to the power of two) to the Cost function. As the weights of the network increase, more cost is added to the neural network, and this is not desirable for the learning process of the neural network. Therefore, it is tried to keep the sum of weights in the network as low as possible to prevent overfitting and make the model experience better generalizability.

This operation of limiting the increase of weights inside the neural network is called Weight decay.

Mathematical explanation:

$$Cost = Loss + \lambda \Sigma Wi^2 \quad (1)$$

$$\frac{\partial Cost}{\partial Wi} = \frac{\partial Loss}{\partial Wi} + 2*\lambda*Wi \quad (2)$$

$$\Delta Wi = -\frac{\partial Cost}{\partial Wi} \quad (3)$$

$$\Delta Wi = -\frac{\partial Loss}{\partial Wi} - 2*\lambda*Wi \quad (4)$$

Explanation of the above formulas:

1- As mentioned, the sum of all the weights inside the neural network is added to the power of two to the previous Loss Function formula to obtain the Cost Function.

2- We calculate the derivative of Cost Function with respect to Wi to interpret the process of updating network weights.

3- The weights of the network move against the direction of the derivative.

4- As you can see, a new member has been added to the weight update operation, and this member is responsible for controlling and limiting the increase in Wi weight.

For example:

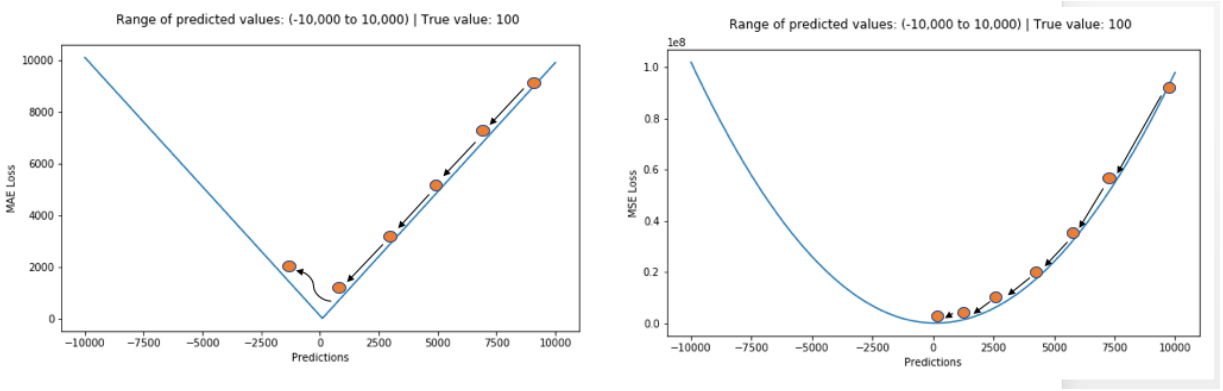
In the past, only Loss Function issued an order to increase a network weight by 3 units, but this new member of Regularization issues an order to decrease a network weight by 4 units. This process plays a fundamental role in limiting network weights.

This updating process continues until the Loss Function command and the Regularization command cancel together and reach a fixed location.

3- Research the Huber error function and write its relations. What is the use of this function? What are the advantages and disadvantages? Check the effect of changing the delta parameter (increase and decrease)

Huber's error function intelligently uses the combination of two error functions MSE and MAE in order to use the advantages of these two functions.

A summary of the advantages and disadvantages of the two error functions MSE and MAE:



MAE

This error function has constant derivatives at points close to the minimum point, and in other words, it returns constant derivatives at points far from or close to the minimum point, and the fact that I do not have a low derivative at points close to the minimum point may cause that Pass the minimum point.

MSE

In this error function, unlike the MAE error function, as we get closer to the minimum point, we experience fewer derivatives and take smaller steps to finally reach the minimum point.

And also one of the disadvantages of this error function is that this function, due to the presence of the power of two in the formula, has a much higher sensitivity to noisy data, and the presence

of noisy data causes us to have a much larger error in the model, and this causes that our model tends towards noisy data.

Huber's error function:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & |y - \hat{y}| > \delta \end{cases}$$

If the output error of the model was within a specified range (delta), Huber error function becomes the MSE error function, and on the other hand, if the output error of the model is outside the specified range, the Huber error function acts like the MAE error function.

The above formula is written in such a way that continuity and derivability are respected at the delta point.

Application and advantages and disadvantages of Huber's error function:

Application and advantages of Huber's error function:

1- This combination of the two mentioned error functions causes us to have different derivatives in terms of distance and proximity to the minimum point within the delta interval, unlike MAE, and as we get closer to the minimum point, we take slower steps in order to Finally, we reach the minimum point and like the MAE error function, the possibility of passing the minimum point disappears

2- If we go out of the delta range, we should not be sensitive to noise like the MSE error function and not deviate the model from the general path just because of noisy data.

Disadvantages:

1- The delta parameter cannot be easily determined in any problem.

2- If we start training from the initial point far from the minimum point, we would have much larger derivatives in the MSE error function and this would make us converge to the minimum point sooner. But because in this Huber error function, if we are far from the minimum point, exactly like the MAE error function, we will receive less and constant derivatives, which makes us converge to the minimum point much later.

The effects of changing the delta parameter:

Increase

The more we increase the value of delta, the closer we get to the MSE error function, and the effects of noisy data in model training increase, and we also have more derivatives in more points near the minimum point, and the speed of convergence to the minimum point increases.

Decrease

As we decrease the value of delta, we get closer to the MAE error function, and this makes us have more derivatives than MSE at the point near the minimum, and we face the risk of jumping or crossing the minimum point.

Implementation exercise:

The work report of the implementation of different regularization and optimization methods and the comparison of their accuracy and results with each other in a multi-layer neural network:

- Stochastic Gradient Descent
- Momentum
- Adagrad
- RMSProp
- Adam

Optimization algorithms reviewed:

Different regularization methods reviewed:

- L1 Regularization
- L2 Regularization

The dataset used to check the classification power of the neural network:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Fashion MNIST

Information from Fashion MNIST dataset:

The Fashion MNIST dataset is a large and free database of images from the world of fashion clothes that are commonly used to train and test various machine learning systems.

Some sample images from Fashion MNIST dataset:

Neural network architecture:

Hidden layer:

Number of neurons: variable (20-75-150)

Activation function: ReLU

Output layer:

Number of neurons: 10 (the number of classes in Fashion MNIST dataset)

Activation function: SoftMax

Step-by-step description of the functions used to implement the project:

Importing required libraries into the project:

```
import numpy as np
from scipy.special import logsumexp
import matplotlib.pyplot as plt
from progress.bar import IncrementalBar
import sys
```

We import the required libraries to the project environment so that we can use the functions inside them.

Numpy library:

To work with matrices and perform parallel processing for neural network training

Matplotlib library:

To plot accuracy and error graphs of the trained model on the training and validation data

Logsumexp from the Scipy library:

To perform mathematical calculations in order to calculate the activation functions. Like: Softmax

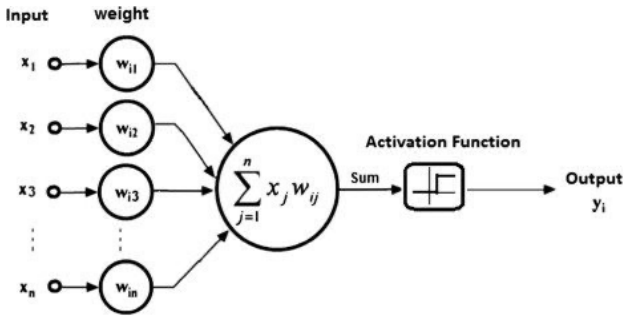
Progress.bar Library:

To display the Progressbar and view the progress of the Fit function

Sys library:

To use certain functions, for example better display of results on the terminal

Activation functions in neural networks:



As you can see in the picture above, in neural networks, each neuron transmits its inputs to the output with a specific activation function according to its task. These activation functions can be used in different layers according to the network task.

You will see examples of these activation functions below:

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Implementation of neural network activation functions:

```python
# derivative function, gets function output as input and computer derivative
class Linear:
    def __init__(self):
        self.__name__ = 'linear'

    def __call__(self, Z):
        return Z

    def derivative(self, Z, *args, **kwargs):
        return 1


class Relu:
    def __init__(self):
        self.__name__ = 'relu'

    def __call__(self, Z):

        return Z * (Z > 0)

    def derivative(self, Z, *args, **kwargs):
        return (Z > 0) * 1


class Sigmoid:
    def __init__(self):
        self.__name__ = 'sigmoid'

    def __call__(self, Z):
        return 1 / (1 + np.exp(-Z))

    def derivative(self, Z, *args, **kwargs):
        return Z * (1 - Z)



# stable version of softmax:
class Softmax:
    def __init__(self):
        self.__name__ = 'softmax'

    def __call__(self, Z):
        e = -np.max(Z, axis=1, keepdims=True)
        exp = np.exp(Z + e)
        return exp / np.sum(exp, axis=1, keepdims=True)

    def derivative(self, Z, *args, **kwargs):
        raise Exception('cannot get derivative')


class Tanh:
    def __init__(self):
        self.__name__ = 'tanh'

    def __call__(self, Z):
        return np.tanh(Z)

    def derivative(self, Z, *args, **kwargs):
        return 1 - (np.tanh(Z) ** 2)


sigmoid = Sigmoid
softmax = Softmax
relu = Relu
linear = Linear
tanh = Tanh
```

To train neural networks, we need a variety of activation functions:

- Linear
- ReLU
- Sigmoid
- Softmax
- Tanh

These functions have been implemented precisely based on their original formulas, along with the derivative of each of these activating functions, to perform neural network training steps, in their respective classes.

Functions related to the separation of training and test data:

```python
class Static_split:
    def __init__(self, test_split=0.2):
        self.test_split = test_split

    def set_data(self, X, y, shuffle=True):
        if shuffle == False:
            p = np.arange(X.shape[0])
        else:
            p = np.random.permutation(X.shape[0])

        split = int(X.shape[0] * self.test_split)
        self.X_test = X[p[:split]]
        self.y_test = y[p[:split]]
        self.X_train = X[p[split:]]
        self.y_train = y[p[split:]]

    def get_test_data(self):
        return self.X_test, self.y_test

    def get_train_data(self):
        return self.X_train, self.y_train


class kfold_split:
    def __init__(self, k=5):
        self.k = k

    def set_data(self, X, y, shuffle=True):
        if shuffle == False:
            p = np.arange(X.shape[0])
        else:
            p = np.random.permutation(X.shape[0])

        self.split = np.random.randint(0, self.k, size=X.shape[0])
        self.X = X
        self.y = y

    def __iter__(self):
        return self.__next__()

    def __next__(self):
        for i in range(self.k):
            yield self.X[self.split != i], self.y[self.split != i], self.X[self.split == i], self.y[self.split ==
i]
```

The two classes Static_split and kfold_split have the task of separating the data into two parts or k parts.

By enabling the Shuffle flag, the data can be shuffled before the sorting operation.

In the following, using these classes, we separate training and validation data.

Implementation of layers in the neural network:

```
class Dense:
    def __init__(self, unit, activation=linear, input_dim=0, trainable=True):
        self.unit = unit
        self.__name__ = 'Dense'
        self.name = "dense"
        self.trainable = trainable
        if type(activation) == str:
            if activation.lower() == 'linear':
                activation = Linear()
            elif activation.lower() == 'sigmoid':
                activation = Sigmoid()
            elif activation.lower() == 'relu':
                activation = Relu()
            elif activation.lower() == 'softmax':
                activation = Softmax()
            elif activation.lower() == 'tanh':
                activation = Tanh()
            else:
                raise Exception("Activation function " + activation + " not found!")

        self.activation = activation
        self.input_dim = input_dim

    def set_name(self, name):
        self.name = name

    def compile(self, input_dim=-1):
        if input_dim == -1:
            assert self.input_dim != 0
        else:
            self.input_dim = input_dim

        self.W = (np.random.random(size=(self.input_dim + 1, self.unit))) - 0.5
        self.W = self.W / (self.W.std() * (input_dim + self.unit) ** 0.5)

    def __call__(self, X, activation=True):
        if activation:
            return self.activation(np.dot(np.hstack((X, np.ones((X.shape[0], 1)))),
self.W))else:
            return np.dot(np.hstack((X, np.ones((X.shape[0], 1)))), self.W)

    def set_trainable(self, trainable):
        self.trainable = trainable

    #Update Weights
    def update(self, dw):
        if self.trainable:
            self.W = self.W + dw

    def reset_weights(self):
        self.W = (np.random.random(size=self.W.shape)) - 0.5
        self.W = self.W / (self.W.std() * (self.W.shape[0] + self.unit) ** 0.5)
```

The code piece of the Dense class is related to the implementation of the layers class in neural networks. In this class, the types of activation functions of the neurons of each layer are specified. Also, update and reset_weights functions can be used to update or reset the weights of neurons of the same layer.
The update function has been used in various Optimizer classes to train the neurons in each layer of the neural network.

Implementation of neural network output error function:

```
class Cross_entropy:
    def __init__(self, last_layer_activation):
        if last_layer_activation == 'sigmoid':
            self.get_loss = self.call_sigmoid
            self.derivative = self.derivative_sigmoid
        elif last_layer_activation == 'softmax':
            self.get_loss = self.call_softmax
            self.derivative = self.derivative_softmax

    def call_sigmoid(self, z, y):
        return np.mean(z * (1 - y) + np.logaddexp(0, -z))

    def derivative_sigmoid(self, h, y):
        return h - y

    def call_softmax(self, z, y):
        return -np.mean(np.sum(y * (z - logsumexp(z, axis=1, keepdims=True)),
axis=1))
    def derivative_softmax(self, h, y):
        return h - y

    def get_loss(self, h, y):
        return -np.mean(np.sum(y * np.log(h) + (1 - y) * np.log(1 - h), axis=1))

    def derivative(self, h, y):
        return (h - y) / (h * (1 - h))

    def __call__(self, h, y):
        self.get_loss(h, y)
```

The code piece of the Cross_entropy class is related to the implementation of neural network output error calculation formulas. In the following, this class has been used in the neural network training process.

Implementation of the model for training:

```
    def predict(self, X, batch_size=32, last_activation=True):
        return np.argmax(self.predict_proba(X, batch_size=batch_size, last_activation=last_activation), axis=1)

    def predict_proba(self, X, batch_size=32, last_activation=True):
        output = np.zeros((X.shape[0], self.layers[-1].unit))

        for i in range(0, X.shape[0], batch_size):
            t = X[i:min(i + batch_size, X.shape[0])].copy()
            for j, layer in enumerate(self.layers):
                t = layer(t, activation=(j != len(self.layers) - 1 or last_activation))

            output[i:min(i + batch_size, X.shape[0])] = t

        return output

    def evaluate(self, X, y, batch_size=32):
        if self.loss is not None:
            p = self.predict_proba(X, batch_size=batch_size, last_activation=not
self.remove_last_layer_activation)
            loss = self.loss(p, y)
            if self.regularization is not None:
                loss += self.regularization.cost(self.layers)

            if self.remove_last_layer_activation:
                p = self.layers[-1].activation(p)

            pred = np.argmax(p, axis=1)
            y_class = np.argmax(y, axis=1)

            acc = np.sum(pred == y_class) / p.shape[0]
            return {'loss': loss, 'acc': acc}
        else:
            p = self.predict_proba(X, batch_size=batch_size)

            pred = np.argmax(p, axis=1)
            y_class = np.argmax(y, axis=1)

            acc = np.sum(pred == y_class) / p.shape[0]
            return {'acc': acc}

    def summary(self):
        print("_" * 65)
        print('Layer (type)                 Output Shape              Param #')
        print("=" * 65)

        layer_type = 'Input'
        output_shape = '(None, {})'.format(str(self.layers[0].input_dim))
        params = '0'
        print('{:<29}'.format(layer_type) +
              '{:<26}'.format(output_shape) +
              '{}'.format(params))
        print("-" * 65)

        trainable_params = 0
        non_trainable_params = 0

        for l in self.layers:
            layer_type = l.name + ' ({})'.format(l.__name__)
            output_shape = '(None, {})'.format(str(l.unit))
            params = '{}'.format(l.W.shape[0] * l.W.shape[1])

            if l.trainable:
                trainable_params += l.W.shape[0] * l.W.shape[1]
            else:
                non_trainable_params += l.W.shape[0] * l.W.shape[1]

            print('{:<29}'.format(layer_type) +
                  '{:<26}'.format(output_shape) +
                  '{}'.format(params))
            print("-" * 65)

        print('Total params:', trainable_params + non_trainable_params)
        print('Trainable params:', trainable_params)
        print('Non-trainable params:', non_trainable_params)
        print("_" * 65)
        print('Optimizer : {}'.format(self.optimizer.__name__))
        if self.regularization is not None:
            print('Regularization : {}'.format(self.regularization.__name__))
        print("=" * 65)
        print()
        print()

    def save_weights(self, path):
        import os
        try:
            os.rmdir(path)
        except:
            pass

        os.mkdir(path)

        for i, layer in enumerate(self.layers):
            np.save(os.path.join(path, str(i) + '.npy'), layer.W)

    def load_weights(self, path):
        import os
        assert len(os.listdir(path)) == len(self.layers)

        for i, layer in enumerate(self.layers):
            self.layers[i].W = np.load(os.path.join(path, str(i) + '.npy'))

    def reset_weights(self):
        for layer in self.layers:
            layer.reset_weights()
```

```
class Sequential:
    def __init__(self):
        self.remove_last_layer_activation = False
        self.layers = []
        self.loss = None
        self.optimizer = None
        self.regularization = None
        self.layer_counter = 1

    def add(self, layer):
        layer.set_name('dense_' + str(self.layer_counter))
        self.layer_counter += 1
        self.layers.append(layer)

    def initialize_layers(self):
        input_dim = -1
        for layer in self.layers:
            layer.compile(input_dim)
            input_dim = layer.unit

    def compile(self, optimizer='SGD', loss='cross_entropy', regularization=None):
        if loss.lower() == 'cross_entropy':
            self.loss = Cross_entropy(self.layers[-1].activation.__name__)
            if self.layers[-1].activation.__name__ == "sigmoid" or self.layers[-1].activation.__name__ == "softmax":
                self.remove_last_layer_activation = True

        if type(optimizer) == str:
            if optimizer.lower() == 'sgd':
                optimizer = SGD()
            elif optimizer.lower() == 'momentum_sgd':
                optimizer = Momentum_SGD()
            elif optimizer.lower() == 'adagrad':
                optimizer = Adagrad()
            elif optimizer.lower() == 'rmsprop':
                optimizer = RMSprop()
            elif optimizer.lower() == 'adam':
                optimizer = Adam()
            else:
                raise Exception("Optimizer " + optimizer + " not found!")

        if regularization is not None and type(regularization) == str:
            if regularization.lower() == 'l1':
                regularization = L1()
            elif regularization.lower() == 'l2':
                regularization = L2()

        self.regularization = regularization
        self.optimizer = optimizer


        self.initialize_layers()

    def fit(self, X, y, batch_size=32, epoch=1, test_data=None):
        assert self.loss is not None

        history = {'acc': np.zeros(epoch), 'loss': np.zeros(epoch)}
        if test_data is not None:
            history['val_acc'] = np.zeros(epoch)
            history['val_loss'] = np.zeros(epoch)

        for epch in range(epoch):
            print('Epoch', epch + 1)
            sum_loss = 0
            sum_acc = 0
            bt_conter = 0

            required_batchs = int(X.shape[0] / batch_size + 0.99999999)
            bar = IncrementalBar(max=required_batchs,
                                 suffix='%(index)d/%(max)d - %(eta)ds')

            epoch_shufle = np.random.permutation(X.shape[0])

            for bt in range(0, X.shape[0], batch_size):
                bt_conter += 1
                # single batch:
                X_batch = X[epoch_shufle[bt:min(bt + batch_size, X.shape[0])]]
                y_batch = y[epoch_shufle[bt:min(bt + batch_size, X.shape[0])]]

                layer_output = [X_batch]
                for j, layer in enumerate(self.layers):
                    layer_output.append(layer(layer_output[-1]))

                delta_next = self.loss.derivative(layer_output[-1], y_batch)
                if not self.remove_last_layer_activation:
                    delta_next *= self.layers[-1].activation.derivative(layer_output[-1])

                grad = [0 for i in range(len(self.layers))]
                regularization_grad = [0 for i in range(len(self.layers))]
                if self.regularization is not None:
                    regularization_grad = self.regularization.derivative(self.layers)

                for i in range(len(self.layers) - 1, 0, -1):
                    delta_prev = np.dot(delta_next, self.layers[i].W.T[:, :-1]) * \
                                 self.layers[i - 1].activation.derivative(layer_output[i])

                    delta_wi = np.dot(np.hstack((layer_output[i], np.ones((layer_output[i].shape[0], 1)))).T,
                                      delta_next)
                    grad[i] = delta_wi / X_batch.shape[0] + regularization_grad[i]

                    delta_next = delta_prev

                delta_wi = np.dot(np.hstack((layer_output[0], np.ones((layer_output[0].shape[0], 1)))).T,
delta_next)
                grad[0] = delta_wi / X_batch.shape[0] + regularization_grad[0]

                self.optimizer.update(grad, self.layers)

                d = self.evaluate(X_batch, y_batch)
                loss, acc = d['loss'], d['acc']
                sum_acc += acc
                sum_loss += loss
                bar.next()

                 sys.stdout.write(" loss: %f acc: %f" % (sum_loss / (bt_conter + 1), sum_acc / (bt_conter + 1)))
                 sys.stdout.flush()

            bar.finish()
            d = self.evaluate(X, y)
            loss, acc = d['loss'], d['acc']
            if test_data is not None:
                d = self.evaluate(test_data[0], test_data[1])
                val_loss, val_acc = d['loss'], d['acc']
                history['val_loss'][epch] = val_loss
                history['val_acc'][epch] = val_acc
                history['loss'][epch] = loss
                history['acc'][epch] = acc
                print('loss: {} acc: {} val_loss: {} val_acc: {}'.format(loss, acc, val_loss, val_acc))
            else:
                history['loss'][epch] = loss
                history['acc'][epch] = acc
                print('loss: {} acc: {}'.format(loss, acc))

        return history
```

The Sequential class code piece is related to the implementation of the model for training and evaluation.
This piece of code is responsible for defining a model with the following features:

- Number of layers
- Type of error function
- Optimizer type
- Regularization type

Also, functions to display a summary of model information at the start of the training process or to save weights, etc. have been implemented in this class.

Implementation of regularization functions:
As you know, regularization is a method used in machine learning to prevent model overfitting. In the following two very practical methods L1 Regularization and L2 Regularization have been implemented for better neural network training.

- **Implementation of L1 Regularization function**

$$Cost = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

As you can see in the regularization expression above, the sum of the absolute value of the weights of the neurons of the neural network is added to the cost function. This operation makes the weights of the neural network tend to a low value and not increase. In fact, the model tries not to consider features that are not essential and their weight tends to zero or towards zero. The value of the Landa parameter also shows the importance of this regularization. So that the more we consider the value of Landa, the model tries to keep the weights closer to zero. This prevents overfitting.

This regularization function is implemented in the following code snippet:

```python
class L1:
    def __init__(self, lam=0.001):
        self.lam = lam
        self.__name__ = 'L1 Regularization'

    def cost(self, layers):

        # L1 Regularization Cost function
        # input: layers -> list of layers. you can access the weight matrix of i'th layer
        #                  using "layers[i].W".
        #                  layers[i].W[-1, :] is the bias term and layers[i].W[:-1, :]
        #                  will give you all weights
        #
        # output: one float number. this number is the total cost of the Regularization term.
        #         don't forget to effect "self.lam" in the return value. lam is the
        #         Regularization Coefficient.

        cost = 0

        for l in layers:
            cost += np.sum(np.abs(l.W[:-1, :]))

        return cost * self.lam


    def derivative(self, layers):

        # L1 Regularization Cost function Derivatives
        # input: layers -> list of layers. you can access weight matrix of i'th layer
        #                  using "layers[i].W".
        #                  layers[i].W[-1, :] is the bias term and layers[i].W[:-1, :]
        #                  will give you all weights
        #
        # output: a list of matrices. output[i] is a matrix with same shapes of layers[i].W
        #         matrix and indicates derivative of each weight respect to the Regularization term



        # initialize dw with the same shape as l.W.shape with bias

        dw = [np.zeros(l.W.shape) * self.lam for l in layers]

        for i in range(len(layers)):
            dw[i][:-1, :] = np.sign(layers[i].W[:-1, :]) * self.lam

        return dw
```

Code segment of class L1 has two functions Cost and Derivative. The task of the Cost function is to calculate the L1 Regularization part of the cost function, and the task of the Derivative function is to calculate the derivative of the L1 Regularization part of the cost function.
In the first function, according to the above formula, we add the sum of the absolute values of the weights of each layer to the cost function, and in the second function, the derivative of the same expression, which is equal to the step function, is implemented.
In the second function, we first create a matrix with the actual dimensions of the weights and then add the derivative of the weight of different layers to it. Because in the section of regularization functions, the dimension related to the term bias was removed. We return it to the dimensions of the derivative matrix, but the derivative value is not calculated for it.

- **Implementation of L1 Regularization function**

L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2}_{\text{Loss function}} + \underbrace{\lambda \sum_{j=0}^{M} W_j^2}_{\text{Regularization Term}}$$

As you can see in the regularization expression above, the sum of the weights of the neural network neurons to the power of two is added to the cost function. This operation, like the previous method, makes the weights of the neural network tend to a low value and not increase. In fact, the model tries not to consider features that are not essential and tends to weight them towards zero. The value of the Landa parameter also shows the importance of this regularization. So that the more we consider the value of Landa, the model tries to keep the weights closer to zero. This prevents overfitting.

This regularization function is implemented in the following code snippet:

```python
class L2:
    def __init__(self, lam=0.001):
        self.lam = lam
        self.__name__ = 'L2 Regularization'

    def cost(self, layers):

        # L2 Regularization Cost function
        # input: layers -> list of layers. you can access the weight matrix of i'th layer
        #                  using "layers[i].W".
        #                  layers[i].W[-1, :] is the bias term and layers[i].W[:-1, :]
        #                  will give you all weights
        #
        # output: one float number. this number is the total cost of the Regularization term.
        #         don't forget to effect "self.lam" in the return value. lam is the
        #         Regularization Coefficient.

        cost = 0

        for l in layers:
            cost += np.sum(np.square(l.W[:-1, :]))

        return cost * self.lam


    def derivative(self, layers):

        # input: layers -> list of layers. you can access weight matrix of i'th layer
        #                  using "layers[i].W".
        #                  layers[i].W[-1, :] is the bias term and layers[i].W[:-1, :]
        #                  will give you all weights
        #
        # output: a list of matrices. output[i] is a matrix with same shapes of layers[i].W
        #         matrix and indicates derivative of each weight respect to the Regularization term
        #

        # initialize dw with the same shape as l.W.shape with bias row

        dw = [np.zeros(l.W.shape) * self.lam for l in layers]

        for i in range(len(layers)):
            dw[i][:-1, :] = 2 * (layers[i].W[:-1, :]) * self.lam

        return dw
```

Like the previous method, the class L2 code piece has two functions, Cost and Derivative. The task of the Cost function is to calculate the L2 Regularization part of the cost function, and the task of the Derivative function is to calculate the derivative of the L2 Regularization part of the cost function.

In the first function, according to the above formula, we add the sum of the weights of each layer to the power of two to the cost function, and in the second function, the derivative of the same expression is implemented.

In the second function, we first create a matrix with the actual dimensions of the weights and then add the derivative of the weight of different layers to it. Because in the section of regularization functions, the dimension related to the term bias was removed. We return it to the dimensions of the derivative matrix, but the derivative value is not calculated for it.

Receiving the Fashion MNIST dataset – data pre-processing and preparation:

In this part, we first receive the data set introduced in the first part and enter it into the project. Then we start preprocessing and preparing the data to enter them into the neural network. This operation includes the following:

- Separation of training and test and validation data
- Normalize data
- One-hot encryption of data labels
- Changing the size of the images in the data set and linearizing them

```
# Load data, normalize and split into train, validation and test sets.

# Import fashion_mnist dataset

from keras.datasets import fashion_mnist
from tensorflow.keras.utils import to_categorical


def load_dataset():

    # load the fashion_mnist from keras.datasets
    # split train/test

    (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

    ## Normalizing data with Z-score normalization, mean=>0, std=>1

    train_mean = x_train.mean()
    train_std = x_train.std()

    x_train = (x_train - train_mean) / train_std
    x_test = (x_test - train_mean) / train_std

    # one-hot encoding of y

    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)

    # flatten Xs:

    x_train = x_train.reshape(x_train.shape[0], -1)
    x_test = x_test.reshape(x_test.shape[0], -1)

    ## split validation data

    split_data = Static_split(test_split=0.2)
    split_data.set_data(x_train, y_train)
    (X_train_new, y_train_new) = split_data.get_train_data()
    (X_val, y_val) = split_data.get_test_data()


    print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)

    return X_train_new, y_train_new, x_test, y_test, X_val, y_val
```

The mentioned operations in the load_dataset() function are implemented as follows:

1. Receiving the data set: Using the Keras library, we receive the Fashion-MNIST data set and divide it into 60,000 sets for training and 10,000 sets for testing.

2. Data normalization: To normalize the data using the Z-score method, we first calculate the mean and standard deviation of the data and then normalize the data using the relevant formulas. After the normalization method, the average of the data will be zero and their standard deviation will be equal to one.

3. Encoding of data labels: To enter data and their labels in order to train the neural network, the labels must be encoded as one-hot. We perform this operation using the Keras library and the to_categorical() function.

4. Changing the dimensions of images: 28 x 28 images in the data set should be entered into the neural network in a one-dimensional form and in a linear vector. We perform this operation with the reshape() function.

5. Splitting the training and validation data: As mentioned, the Static_split function has the task of separating the data into two parts with a certain coefficient. Using this function and the ratio of 0.2, we separate the training and validation data and divide them into two parts.

Implementation of optimizer functions:

• Implementation of the SGD optimizer function:

Optimization algorithms are used in machine learning to find the optimal parameters of a model. The simple SGD method for weight update calculates the gradient for a subset of the data as shown in the algorithm below and updates the weight by multiplying it by the learning rate. SGD is suitable for very large data sets as well as training data that contain noisy or sparse data.

You can see the pseudo code related to the SGD optimizer in the image below:

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

```python
# SGD Optimizer as a template to see how you can write Optimizers.
# Calling layer.update(delta) method, will simply update weight matrix
# by formula:
# w_new = w_old + delta
# All classes have a dummy output to prevent errors. the output is just SGD update rule

class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.__name__ = 'SGD'

    def update(self, grads, layers):
        for layer, delta in zip(layers, grads):
            layer.update(-delta * self.lr)

sgd = SGD
```

The SGD optimizer algorithm is completely and accurately implemented in the SGD class.
The update function updates the weights of each layer using the gradients calculated for each layer.

• **Implementation of the Momentum-SGD optimizer function:**

Momentum optimizer is presented to improve the convergence of gradient based optimization algorithms. In fact, it is an extension of the SGD optimizer and has made it converge faster.
The point in the Momentum optimizer is that this method for updating the weight is not dependent only on the current gradient, and the average gradient is maintained over time, preventing the optimizer from getting stuck in the local minimum. The Momentum optimizer is also suitable for data sets that have high noise and complex models with high dimensional parameters.

You can see the pseudocode related to the Momentum-SGD optimizer in the image below:

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

```
class Momentum_SGD(SGD):
    def __init__(self, lr=0.01, momentum=0.9):
        super().__init__(lr)
        self.momentum = momentum

        # Helping attribute
        self.velocity = None

        self.__name__ = 'Momentum_SGD'


    # eq 8.21 from page 300
    def update_velocity(self, grads):
        if self.velocity is None:
            self.velocity = [0 for i in range(len(grads))]

        for i in range(len(grads)):
            self.velocity[i] = self.momentum * self.velocity[i] - self.lr *
grads[i]

    def update(self, grads, layers):
        # inputs are grads and layers.
        #        grads is a list of gradient matrices.
        #        layers is a list of layers.

        self.update_velocity(grads)

        for layer, v in zip(layers, self.velocity):
            layer.update(v)

momentum_sgd = Momentum_SGD
```

update_velocity() function:
If it was the first time that the model is trained, it fills the velocity matrix completely with zero, and from the next times the value of the velocity matrix is calculated exactly according to the formula of the Momentum-SGD algorithm and is stored in the corresponding matrix.

update() function:
In this function, the velocity values which are vital in the member momentum optimization algorithm are calculated first, and then the weights of the different layers are updated using the calculated velocity values.

- **Implementation of the Adagrad optimizer function:**

The Adagrad optimizer operates on a “learning rate” component; In this way, it adapts the learning rate based on the parameters of the model. In this method, the learning rate is divided by

the root of the sum of the squares of the past and current gradients, and somehow the learning rate increases for the parameters that are updated less often. Note that in the SGD method, the gradient remains constant and unchanged, but in this method, AdaGrad automatically adjusts the learning rate according to the needs of each parameter.
AdaGrad is suitable for sparse data and problems with high dimensionality of input features.

You can see the pseudo code related to the Adagrad optimizer in the image below:

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

```
class Adagrad:
    def __init__(self, lr=0.01):
        self.global_lr = lr
        self.epsilon = 1e-7
        self.velocity = None
        self.accumulated_gradient = None

        self.__name__ = 'AdaGrad'

    def update(self, grads, layers):
        # inputs are grads and layers.
        #        grads is a list of gradient matrices.
        #        layers is a list of layers.

        if self.velocity is None:
            self.velocity = [0 for i in range(len(grads))]
            self.accumulated_gradient = [0 for i in range(len(grads))]

        for i in range(len(grads)):
            self.accumulated_gradient[i] = self.accumulated_gradient[i] + (grads[i] **
2)          lr = self.global_lr / (self.epsilon + self.accumulated_gradient[i] ** 0.5)

            self.velocity[i] = - lr * grads[i]

        for layer, v in zip(layers, self.velocity):
            layer.update(v)

adagrad = Adagrad
```

For each layer, accumulated_gradient is updated by adding the square of the gradient. Then, we calculate the root mean square of the gradient and the epsilon value. Finally, the learning rate is obtained by multiplying the inverse of the expression and finally multiplying by the gradient.

In fact, by using this method, as the size of the gradient or derivative increases, the learning rate decreases so that we do not pass the minimum point.

In this piece of code, velocity is actually the same as deltaW located in the pseudo-code of the above algorithm.

Then, using the update() function, the weights of each layer are updated using the velocity value.

- **Implementation of RMSprop optimizer function:**

RMSprop, like the adagrad optimizer, is an adaptive learning rate introduced to improve the AdaGrad method and overcome its limitations. In this method, instead of calculating the sum of the squares of the gradients, the exponential moving average of these gradients is used. Because the Adagrad method did not notice the decrease in the size of the gradient at the current moment due to summing up all the derivatives of the previous steps.
Using this optimizer is useful when the model has many weights or when the dataset is sparse.

You can see the pseudo code related to the RMSprop optimizer in the image below:

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

```python
class RMSprop:
    def __init__(self, lr=0.001, rho=0.9):
        self.global_lr = lr
        self.rho = rho
        self.epsilon = 1e-7

        # Helping attribute
        self.velocity = None
        self.accumulated_gradient = None

        self.__name__ = 'RMSprop'

    def update(self, grads, layers):
        # inputs are grads and layers.
        #        grads is a list of gradient matrices.
        #        layers is a list of layers.

        if self.velocity is None:
            self.velocity = [0 for i in range(len(grads))]
            self.accumulated_gradient = [0 for i in range(len(grads))]

        for i in range(len(grads)):
            self.accumulated_gradient[i] = self.rho * self.accumulated_gradient[i] +\
                                           (1 - self.rho) * (grads[i] ** 2)
            lr = self.global_lr / (self.epsilon + self.accumulated_gradient[i]) **
0.5
            self.velocity[i] = -lr * grads[i]

        for layer, v in zip(layers, self.velocity):
            layer.update(v)

rmsprop = RMSprop
```

The implementation of this optimizer is very similar to the Adagrad optimizer, with the difference that the existence of the rho hyperparameter makes it possible to reduce the accumulated_gradient, and this reduction can also solve the weakness of the Adagrad optimizer. Epsilon is also used as the initial learning rate in It will be considered and then updated.

• Implementation of the Adam optimizer function:

Adam's optimization algorithm is also a generalized version of SGD algorithm. Also, Adam's algorithm can be considered as a combination of RMSprop and Momentum. In fact, Adam is able to adaptively adjust the learning rate for each parameter based on the gradients of past iterations, like RMSProp. One of the advantages of this optimizer is its easy implementation and optimality in terms of computation. This optimizer is very useful for problems that are large in terms of data or parameters, or have gradients that are noisy.

You can see the pseudo-code for the Adam optimizer in the image below:

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

```
class Adam(Adagrad):
    def __init__(self, lr=0.001, rho_1=0.9, rho_2=0.999):
        self.global_lr = lr
        self.rho_1 = rho_1
        self.rho_2 = rho_2

        # Helping attribute
        self.first_moment_estimate = None
        self.second_moment_estimate = None
        self.t = 0
        self.__name__ = 'Adam'


    def update(self, grads, layers):
        # inputs are grads and layers.
        #        grads is a list of gradient matrices.
        #        layers is a list of layers.

        if self.first_moment_estimate is None:
            self.first_moment_estimate = [0 for i in range(len(grads))]
            self.second_moment_estimate = [0 for i in range(len(grads))]
            self.t = 0

        self.t += 1

        for i in range(len(layers)):
            self.first_moment_estimate[i] = self.rho_1 * self.first_moment_estimate[i] +\
                                            (1 - self.rho_1) * grads[i]
            s = self.first_moment_estimate[i] / (1 - self.rho_1 ** self.t)

            self.second_moment_estimate[i] = self.rho_2 * self.second_moment_estimate[i]
+\                                           (1 - self.rho_2) * (grads[i] ** 2)
            r = self.second_moment_estimate[i] / (1 - self.rho_2 ** self.t)

            deltaw = -self.global_lr * s / (1e-8 + r ** 0.5)

            layers[i].update(deltaw)

adam = Adam
```

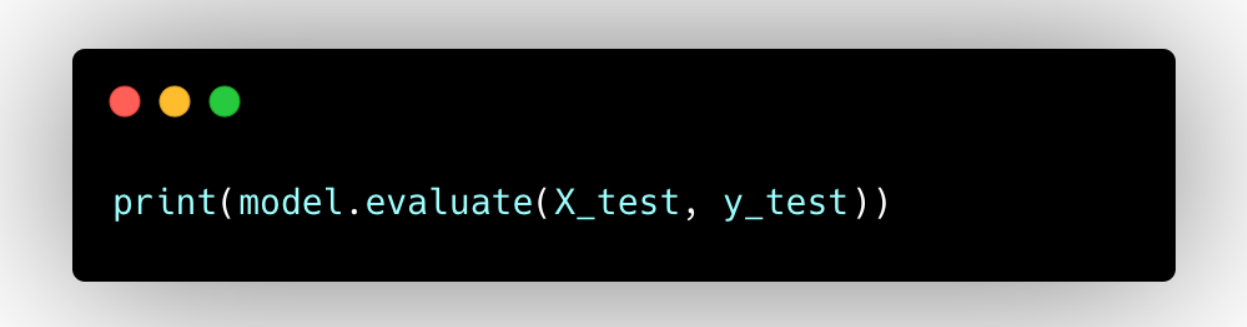
This optimizer is a combination of momentum and RMSprop optimizers and is implemented in the same way. Increasing the variable t will make the movement steps smaller, or the learning rate will decrease.
In the update function, it updates the estimate of the first and second moments of the gradient for each parameter and calculates the learning rate using these estimates.
rho_1 and rho_2 are parameters that control the exponential decay rate for the average mobility of the first and second moments, respectively.

Model training code piece with adjustable parameters:

```
X_train_new, y_train_new, X_test, y_test, X_val, y_val = load_dataset()

model = Sequential()
model.add(Dense(20, 'relu', input_dim=X_train_new.shape[1])) # hidden layer, 20 neurons with relu activation
function
model.add(Dense(10, 'softmax')) # output layer, 10 neurons with softmax activation function

model.compile(loss='cross_entropy', optimizer='sgd', regularization='l2')

model.summary()

history = model.fit(X_train_new, y_train_new, epoch=20,
                    batch_size=64, test_data=(X_val, y_val))

plt.figure()
plt.subplot(1, 2, 1)
plt.title('acc')
plt.plot(history['acc'], 'b-o', label='Train')
plt.plot(history['val_acc'], 'r--o', label='Test')
plt.grid()
plt.legend()

plt.subplot(1, 2, 2)
plt.title('loss')
plt.plot(history['loss'], 'b-o', label='Train')
plt.plot(history['val_loss'], 'r--o', label='Test')
plt.grid()
plt.legend()

plt.show()
```

The above code snippet is implemented to train different models with different goals, so that at first the architecture of the neural network is determined along with the number of layers and the number of neurons in each layer, and then only by placing the desired optimizer inside "optimizer" property and placing the desired regularization method in the "regularization" property and specifying the number of repetitions of the training stage in the "epoch" field, the desired neural network can be trained.
In the result review section, various models have been trained, and you can see their results review in the relevant section.

Code piece for the final evaluation of the accuracy of the model trained on the test data:



By using this piece of code, you can see the result and accuracy of the model on the test data, or in other words evaluate the model.

Conclusion and analysis of results

As in the introduction, you got acquainted with the objectives of this research and in the implementation part, with the method of doing the work, now it is necessary to use the implemented methods and train networks with various architectures to check that In which case, the best result is obtained.

The investigations are carried out in the following four stages:

1. Neural network training using SGD optimizer and L2 Regularization
Objective: to investigate and find the optimal number of neurons in the hidden or middle layer

2. Training the neural network with the desired number of neurons in the previous step using the L2 Regularization method
Objective: Finding the best optimizer

3. Neural network training with the desired number of neurons and optimizer in the previous step
Objective: To investigate the L1 Regularization method on the most optimal optimizer

4. Report the result on the test data, using the best trained neural network

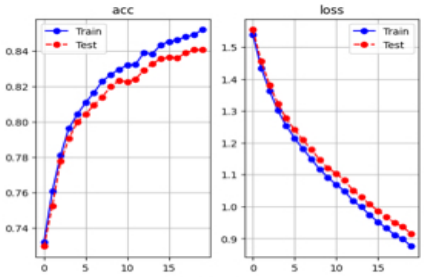
Neural network accuracy evaluation metrics:
Finally, we will test and check the results with the error and accuracy metrics of the model on the training and validation and test data:
loss,acc,val_loss,val_acc,test_loss,test_acc

First stage
In this step, we train the neural network with the SGD optimizer and the L2 Regularization method with different numbers of middle neurons in order to check how many neurons in the hidden layer will have a more favorable result in the output.

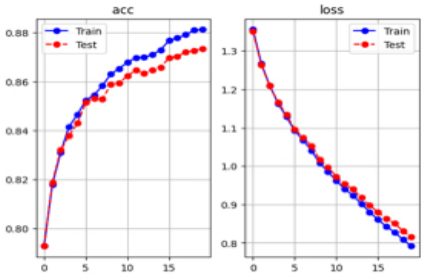
In this step, we train the neural network three times using 20, 75 and 150 neurons in the hidden layer.

The graphs and table of results can be seen in the following images:

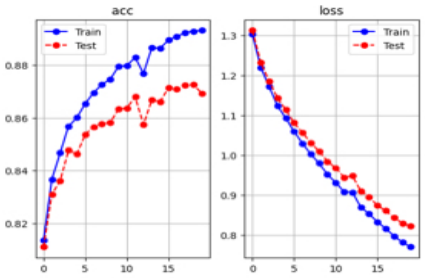
Optimizer: SGD
Regularization: L2



20 Neurons in the hidden layer



75 Neurons in the hidden layer



150 Neurons in the hidden layer

SGD/L2	loss	acc	val_loss	val_acc
20 Neurons in the hidden layer	0.8756	0.8518	0.9151	0.8408
75 Neurons in the hidden layer	**0.7914**	**0.8812**	**0.8151**	**0.8735**
150 Neurons in the hidden layer	0.7700	0.8931	0.8215	0.8694

As you can see in the results table above in 3 different modes, the accuracy of the model output on the validation data in the mode where we used 75 neurons in the hidden or middle layer was better than the other modes with 20 and 150 neurons.

Result: The result of this stage of the research was that by using the SGD optimizer and the L2 Regularization method, the number of 75 neurons in the hidden layer shows a more favorable result in the output on the validation data.

Second stage

At this stage, we train neural networks with different optimizers to check which one of the optimizers gives a better result in the same conditions, i.e. using 75 neurons in the hidden layer and the L2 Regularization method.

The graphs and table of results can be seen in the following images:

75 Neurons in the hidden layer/L2	**loss**	**acc**	**val_loss**	**val_acc**
SGD	0.7914	0.8812	0.8151	0.8735
Momentum SGD	0.3815	0.8939	0.4503	0.8753
Adagrad	**0.4053**	**0.9031**	**0.4525**	**0.8842**
RMSprop	0.3882	0.8930	0.4412	0.8746
Adam	0.3681	0.8958	0.4212	0.8765

As shown in the table above, in the same condition with the mentioned conditions, the Adagrad optimizer had a better result and accuracy on the validation data.

Result: In this problem or the mentioned project and mode (75 neurons in the hidden layer and using the L2 Regularization method), the Adagrad interpolator has a higher speed and accuracy than other optimizers, although with a small margin. Because in the number of repetitions of fixed training, it has achieved higher accuracy earlier than the other optimizers.

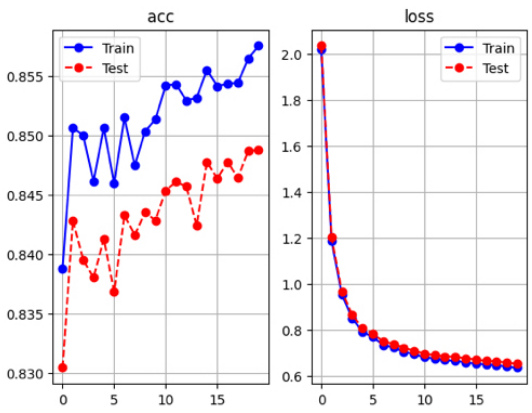
Third stage

According to the investigations carried out in the previous stages, we found that the neural network architecture with the number of 75 neurons in the hidden layer along with the Adagrad optimizer and the L2 Regularization method had the best results on the validation data.

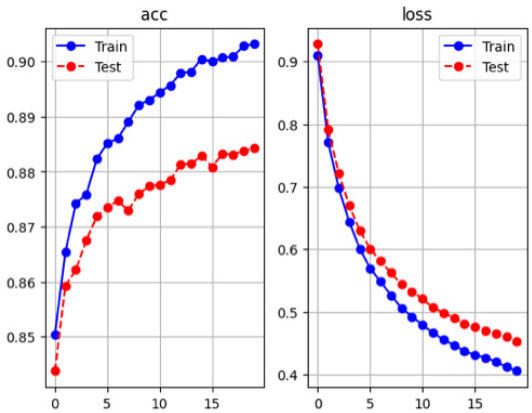
In this step, we train the same superior architecture this time with the L1 Regularization method and compare the results with the results of the previous architecture.

The graphs and table of results can be seen in the following images:

75 Neurons in the hidden layer
Optimizer: Adagrad



L1 Regularization



L2 Regularization

75 Neurons in the hidden layer/Adagard	**loss**	**acc**	**val_loss**	**val_acc**
L1 Regularization	0.6351	0.8575	0.6524	0.8487
L2 Regularization	**0.4053**	**0.9031**	**0.4525**	**0.8842**

As shown in the table above, the top architecture using the L2 Regularization method has recorded better results than the L1 Regularization method.

Result: After the investigations carried out in these three stages, the best neural network architecture information to solve this problem is as follows:

The number of neurons in the hidden layer: 75
Optimizer: Adagrad
Regularization method: L2 Regularization

Final stage

Finally, in this step, we evaluate the accuracy of the output of the mentioned superior architecture model using the command you saw in the implementation step.

You can see the table of results in the image below:

75 Neurons in the hidden layer/Adagard	**loss**	**acc**	**val_loss**	**val_acc**	**test_loss**	**test_acc**
L2 Regularization	0.4053	0.9031	0.4525	0.8842	**0.4789**	**0.8763**

Result: The accuracy of the trained model using the superior architecture evaluated in the above 3 steps on the test data is 87.63%.

Testing the superior architecture with 2 hidden layers! :

After the tests done, we added a layer to the top architecture layers to check the changes in accuracy on the test data.

You can see the table of results in the image below:

75 Neurons in 2 hidden layer/Adagard	**loss**	**acc**	**val_loss**	**val_acc**	**test_loss**	**test_acc**
L2 Regularization	0.3728	0.9167	0.4435	0.8915	**0.4661**	**0.8817**

Result: it is observed that using two hidden layers will lead to 88.17 % accuracy in test data.