Implementation exercise:

The purpose of this project is to code various methods of optimization and regularization

Then using them and checking and comparing their output and accuracy and results with each other in an MLP network. In this exercise, some parts of the code are already written and do not need to be changed and only need to be executed; But some other parts of the code need to be completed by you.

To perform this exercise, Fashion MNIST dataset is considered. You can get this dataset from GitHub through the following link:

https://github.com/zalandoresearch/fashion-mnist

Divide the data into three parts Train, Validation & Test according to the method in the code. To compare different models, you can use the final accuracy on the Validation data.

Optimization:

As introduced in the class, there are various optimization algorithms for neural network model training. You should implement these different methods and compare with each other in terms of accuracy and speed. For this purpose, you must implement the following methods in the corresponding blocks in the notebook; The SGD optimizer is implemented as an example:

➢Stochastic Gradient Descent ➢ Momentum

➢Adagrad

➢RMSProp

➢Adam

As you know, regularization methods are called a set of methods that can prevent overfitting. In this exercise, you should implement two main methods of regularization:

➢L1 Regularization ➢ L2 Regularization

To read more about each of these methods and observe the pseudo-code, refer to the Deep Learning reference book written by Goodfellow (Chapter 7 and 8.(

Architecture selection:

:Regularization

It is suggested to use a neural network with a hidden layer and an output layer. The number of output layer neurons is 10 (because we have 10 classes) and its activation function is Softmax. The hidden layer activation function will be ReLU. First train the model for three different values for the number of hidden layer neurons and then report the best architecture.

In the next step, train and test the best architecture for a fixed regularization with different optimizers. Also report the best option for this step.

In the final stage, train the best architecture with the best optimizer with another regularization that you have not tested in the previous stage and compare the two.

Finally, report the accuracy on the test data for your best model.

implementation

Introduction

The purpose of this project is to code various optimization and regularization methods and then use them, check and compare their output, accuracy and result with each other and obtain the best neural network model.

Optimization

Optimization in neural networks is the process of adjusting the weights of a neural network with the aim of minimizing the error between the predicted output and the actual output.

The goal of optimization is to find a set of weights that results in the most accurate predictions for new data, which involves finding a balance between overfitting and underfitting.

Various techniques are used for optimization in neural networks, such as gradient descent, Adam optimization, Adagrad optimization, etc. These techniques differ in how the weights are updated, but they all aim to find an optimal set of parameters that minimizes the error on new data.

In this exercise, SGD optimization, momentum, Adam, Adagrad and RMSProp have been implemented.

regularization

Regularization in neural networks is a technique used to avoid overfitting the model. Overfitting occurs when the model is too complex.

There are different types of regularization techniques in neural networks:

.1L1 regularization: This technique adds a term to the loss function that is equal to the absolute value of the weights. This regularization makes the model use fewer features.

.2L2 regularization: This technique adds a term to the loss function that is equal to the square of the weights.

.3Dropout: This technique randomly drops some neurons during training and forces the network to unlearn stronger features.

.4Early stopping: This technique stops training when the validation error starts to increase and avoids overfitting by avoiding further optimization of the model.

In this exercise, two L1 and L2 regularization techniques are implemented and analyzed.

Data collection

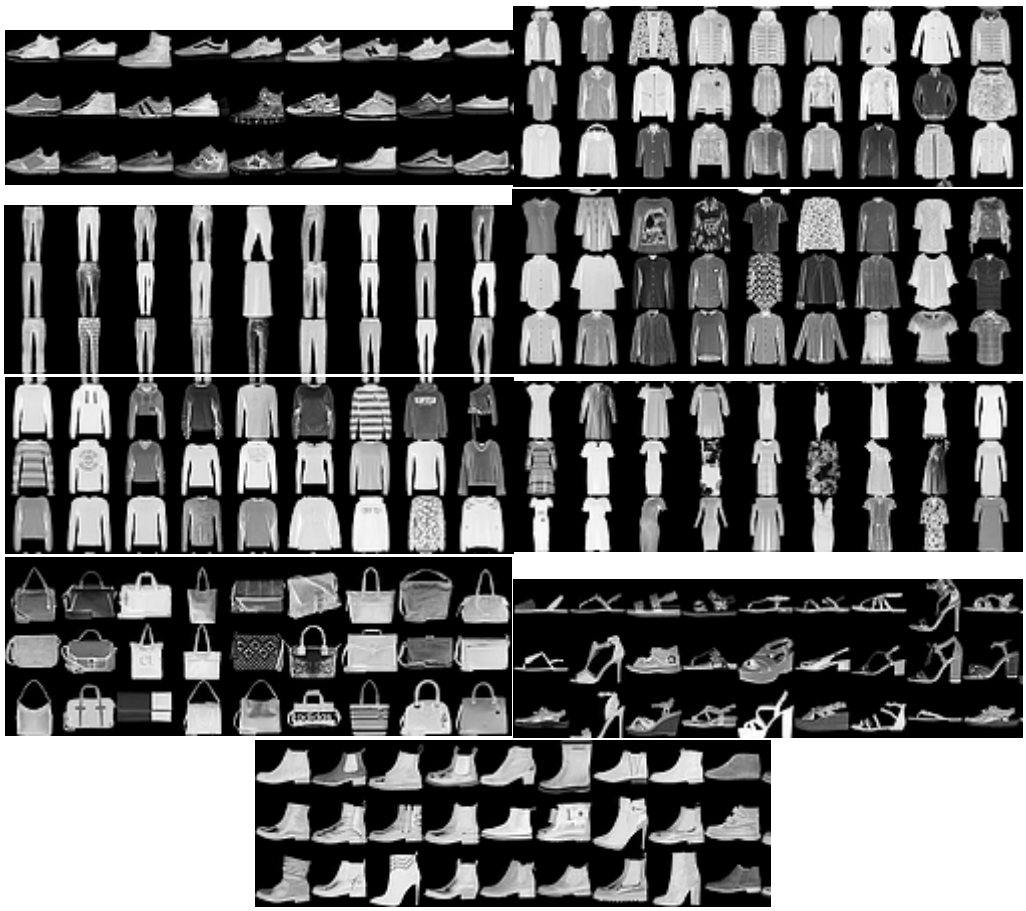
In this exercise, the Fashion MNIST dataset is examined.

Fashion MNIST is an image dataset that contains 70,000 gray scale images that include clothing such as T-shirts, dresses, shoes, and bags. These images are

divided into 10 categories, each of which represents a different type of clothing. (Table 1)

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

An example of the dataset can be seen in the images below.



Data normalization

```
def load_dataset():
    (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

    train_mean = x_train.mean()
    train_std = x_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)

    x_train = x_train / 255.0
    x_test = x_test / 255.0

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

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

    return x_train, y_train, x_test, y_test, X_val, y_val
```

The written code defines a function called load_dataset that loads the Fashion-MNIST dataset using Keras and then preprocesses and splits the data for training, validation, and testing. This function first loads the dataset into the training and test sets using the fashion_mnist.load_data function from the Keras library. The training set has 60,000 images and the test set has 10,000 images in the size of 28×28 pixels. In the next step, we normalize the data using the Z_score method. The function calculates the mean and standard deviation of the training data and uses them to normalize the pixel values of the dataset. Then, it encodes the data labels using the to_categorical function of Keras. Then the data, which is 28 x 28 images, must be converted into a one-dimensional form, and we do this with reshape. After preprocessing, the function divides the training data into training, validation and test sets using the Static_split method with a test split ratio of 0.2.

methods

Implementation of regularization functions

L1 Regularization

As mentioned, L1 Regularization or Lasso regularization is a method used in machine learning to prevent model overfitting.

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

As shown in the formula above, L1 Regularization assigns an expression to the loss function of the model that is equal to the sum of the absolute values of the weights. This expression causes the weights of the model to tend to a small value or set some weights to zero. This means that the model selects only the most important features and leaves the rest to zero. As the Landa model increases, it brings the weights closer to zero.

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

    def cost(self, layers):

        cost = 0
        for l in layers:
            cost += np.sum(np.abs(l.W[:-1, :]))

        return cost * self.lam

    def derivative(self, layers):

        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
```

The L1 Regularization class has two functions: "cost" and "derivative."

The "cost" function calculates the cost of L1 Regularization for a given set of layers. It takes the list of layers as input and calculates the sum of the absolute values of the weights. This sum is multiplied by self.lam to get the total cost of L1 Regularization.

The "derivative" function also calculates the derivative of the L1 Regularization cost according to the weight of the layers.

L2 Regularization

Like L1 Regularization, L2 Regularization is a common technique used in machine learning to avoid overfitting the model by adding a term to the loss function. It causes the model to encourage by shrinking the weights towards zero but not zero itself.

Mathematically, L2 Regularization adds the sum of squared weights term to the loss function.

λ is the regularization tuning parameter.

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

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

    def cost(self, layers):

        cost = 0
        for l in layers:
            cost += np.sum(np.square(l.W))

        return cost * self.lam / 2

    def derivative(self, layers):

        derivatives = []
        for l in layers:
            dW = np.zeros(l.W.shape)
            dW = l.W
            derivatives.append(dW * self.lam)

        return derivatives
```

Like L1 Regularization, this code also defines two functions: cost and derivative. The cost function calculates the cost of L2 regularization, while the derivative function calculates the derivative of the cost with respect to the weights. The lam parameter is also the regularization coefficient that controls the power of regularization. The cost function calculates the regularization cost for the layers and calculates the sum of the squared weights of each layer. It then multiplies the result by the regularization factor (lam) and divides it by 2 to get the total L2 regularization cost. The derivative function also calculates the derivative of the L2 Regularization cost according to the weight of each layer.

Implementation of Optmizer

SGD

SGD (Stochastic Gradient Descent) is an optimization algorithm used in machine learning to find the optimal parameters of a model. It is a type of gradient descent where instead of calculating the gradient of the loss function for the entire training set, the gradient is calculated for a small set of data.

By using only a subset of the data, SGD is faster and more computationally efficient, especially when working with large datasets.

SGD is suitable for large-scale datasets where the number of samples is very large, and the training data contains noise or sparse data ..

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
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 "update" function of the SGD class has two arguments: grads and layers.

The for loop in the "update" function iterates over the layers and their gradient matrices and applies the update rule. The update rule is also defined as follows:

w_new = w_old - delta * lr

where w_old is the current weight matrix, delta is the gradient matrix for that layer, and lr is the learning rate.

Momentum

Momentum is an optimization algorithm used to improve the convergence of gradient-based optimization algorithms.

In the Momentum optimizer, a running average of the gradients is maintained over time, which helps to smooth fluctuations in the gradients and prevent the optimizer from getting stuck in a local minimum. The idea is that we have a velocity vector that is in the direction of the minima and we update the weights as before.

Momentum is useful when dealing with highly noisy datasets or when gradients fluctuate frequently.

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

```python
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'

    def update(self, grads, layers):

        if self.velocity is None:
        # initialize velocity
            self.velocity = [np.zeros_like(grad) for grad in grads]

        for i, (layer, delta) in enumerate(zip(layers, grads)):
            self.velocity[i] = self.momentum * self.velocity[i] - self.lr * delta
            layer.update(self.velocity[i])

momentum_sgd = Momentum_SGD
```

The Momentum_SGD optimizer is an extension of the SGD optimizer that helps the optimizer move towards the minimum faster. This method is suitable for optimization problems with complex and high-dimensional parameter spaces.

The class has two properties: learning rate lr, and Momentum. Momentum controls the contribution of previous updates to the current update. It also has a speed covariate. This method sets the velocity to zero if it is not already initialized, then calculates the update and applies it to the layer's weight matrix.

.Adagrad

AdaGrad is an optimization algorithm that adapts the learning rate based on model parameters. The main idea behind AdaGrad is to decrease the learning rate for frequently updated parameters and increase the learning rate for infrequently updated parameters. This is done by scaling the learning rate of each parameter by the inverse of the square root of the sum of the gradients up to that point. In this way, 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.

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

```python
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):

        if self.accumulated_gradient is None:
            self.accumulated_gradient = [np.zeros_like(grad) for grad in grads]

        for i, (grad, layer) in enumerate(zip(grads, layers)):
            self.accumulated_gradient[i] += grad ** 2
            rms = np.sqrt(self.accumulated_gradient[i] + self.epsilon)
            delta = -self.global_lr * grad / rms
            layer.update(delta)

adagrad = Adagrad
```

This class takes a learning rate (lr) as input and initializes some variables including epsilon, velocity, and accumulated_gradient.

For each gradient and layer, updates accumulated_gradient by adding the square of the gradient. Then, it calculates the root mean square (rms) of the gradient and the epsilon value. Finally, to update the gradient layer weights, the root mean square of the gradients and the global learning rate are calculated.

RMSProp

RMSProp is an optimization algorithm similar to AdaGrad, but overcomes its limitation of rapidly decreasing learning rate by summing squared gradients instead of summing them.

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

RMSProp is suitable for datasets with sparse gradients, also where the gradients can become large and unstable and the learning rate must be adjusted dynamically.

RMSProp is useful when training deep neural networks that have large weights, especially when the problem is sparse data or noisy gradients

```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):

        if self.accumulated_gradient is None:
          self.accumulated_gradient = [np.zeros_like(grad) for grad in grads]

        for i, (grad, layer) in enumerate(zip(grads, layers)):
              self.accumulated_gradient[i] = self.rho*self.accumulated_gradient[i] + (1-self.rho)*(grad
** 2)
              rms_grad = np.sqrt(self.accumulated_gradient[i] + self.epsilon)
              delta = -self.global_lr * grad / rms_grad
              layer.update(delta)

rmsprop = RMSprop
```

Adam

Adam stands for Adaptive Moment Estimation, an optimization algorithm used for stochastic gradient descent in deep learning. It is a combination of two other optimization algorithms RMSProp and Momentum.

Adam can calculate the learning rate for each parameter based on past gradients.

Adam is suitable for a wide variety of datasets and models.

The Adam optimizer is suitable for training deep neural networks on a wide variety of datasets, especially for datasets with a large number of samples and parameters.

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 $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  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

        self.epsilon = 1e-8

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

    def update(self, grads, layers):

        if self.first_moment_estimate is None:
          self.first_moment_estimate = [np.zeros_like(grad) for grad in grads]
          self.second_moment_estimate = [np.zeros_like(grad) for grad in grads]

        self.t +=1
        for i, (grad, layer) in enumerate(zip(grads, layers)):

            self.first_moment_estimate[i] = self.rho_1*self.first_moment_estimate[i] + (1-
self.rho_1)*grad
            self.second_moment_estimate[i] = self.rho_2*self.second_moment_estimate[i] + (1-
self.rho_2)*(grad**2)

            tempS = self.first_moment_estimate[i]/(1-(np.power(self.rho_1, self.t)))
            tempR = self.second_moment_estimate[i]/(1-(np.power(self.rho_2, self.t)))
            temp1 = np.sqrt(tempR)
            temp = temp1+self.epsilon
            delta = -tempS/temp
            layer.update(self.global_lr *delta)

adam = Adam
```

The update method defines the main optimization step, where it updates the estimates of the first and second moments of the gradient for each parameter and calculates the learning rate using these estimates. The learning rate is then used to update the parameters in each layer of the neural network.

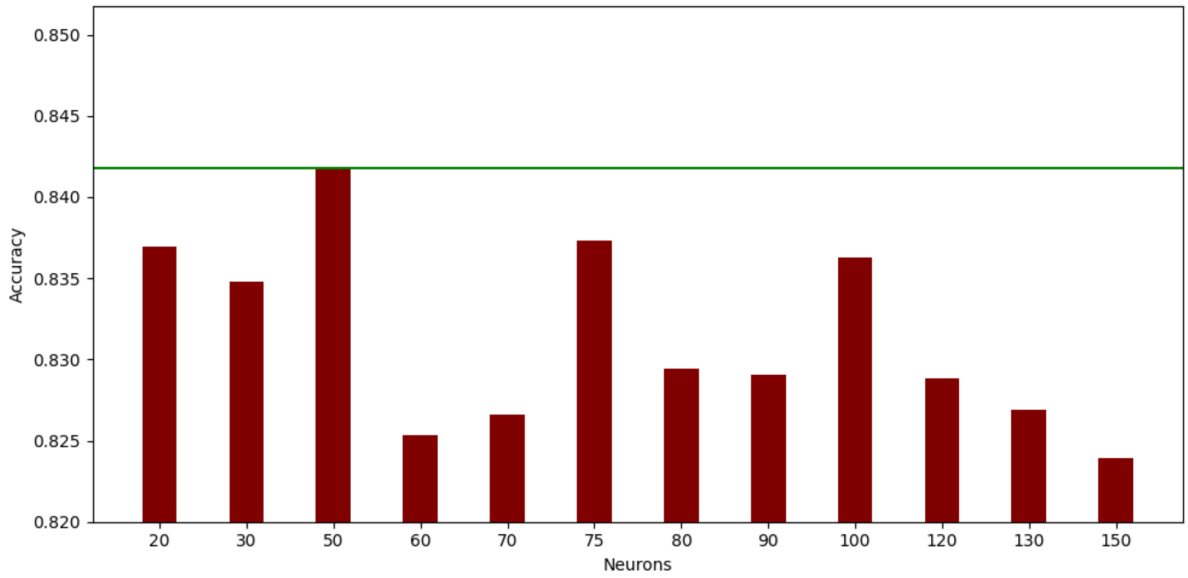
The parameters rho_1 and rho_2 control the exponential decay rate for the average mobility of the first and second moments, respectively. The epsilon parameter is added to prevent division by zero.

the experiment

Different neurons

To get the best model, first with the same optimizer and regularization, we test the number of different neurons to achieve higher accuracy. In the diagram below, the accuracy of the model is drawn on the validation data with the number of different neurons with the same parameters.

optimizer is equal to SGD, regularization is equal to L1, and the number of epochs is equal to 20.

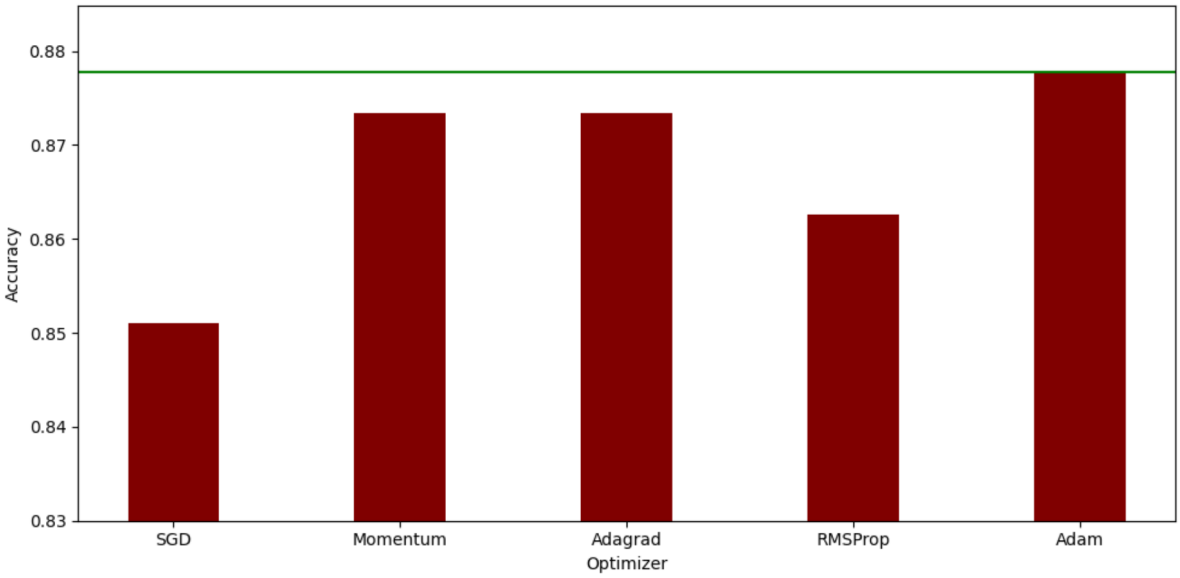


According to the obtained results, 50 neurons are suitable for the model and obtain the most accuracy.

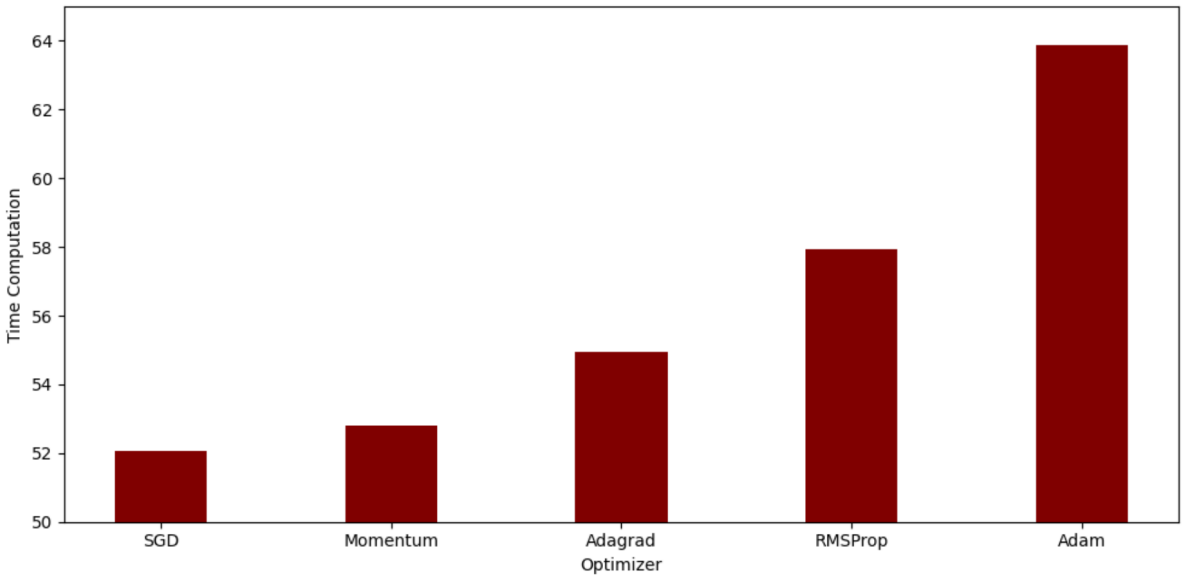
A different optimizer

In the next step, we test the model with 50 neurons with the same regularization and different optimizers, and the goal is to find the best optimizer.

The set parameters are equal to L2 regularization and epoch is also equal to 20. The results obtained for different optimizers are in the graph below.



We also compared the optimizers in terms of the training speed of the dataset.



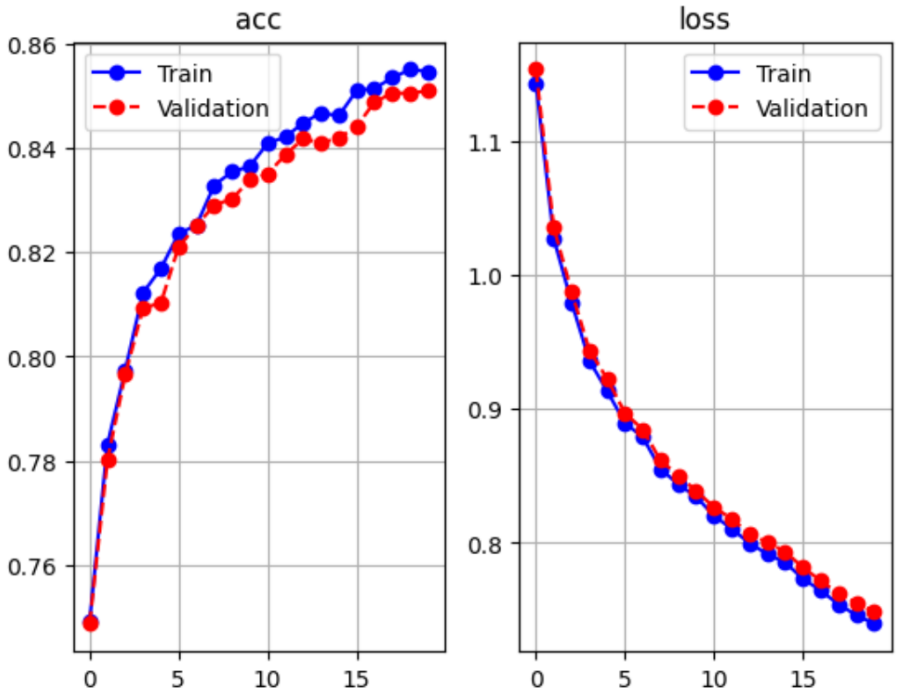
According to the above two graphs and their analysis, the following table can be concluded .

Optimizer	Train Accuracy	Train Loss	Val Accuracy	Val Loss	Time Computation
Adagrad	0.881	0.473	0.873	0.491	54.93
Adam	**0.893**	**0.381**	**0.877**	**0.423**	63.86
SGD	0.854	0.739	0.851	0.748	**52.06**
RMSProp	0.876	0.429	0.862	0.468	57.94
Momentum	0.887	0.409	0.873	0.449	52.79

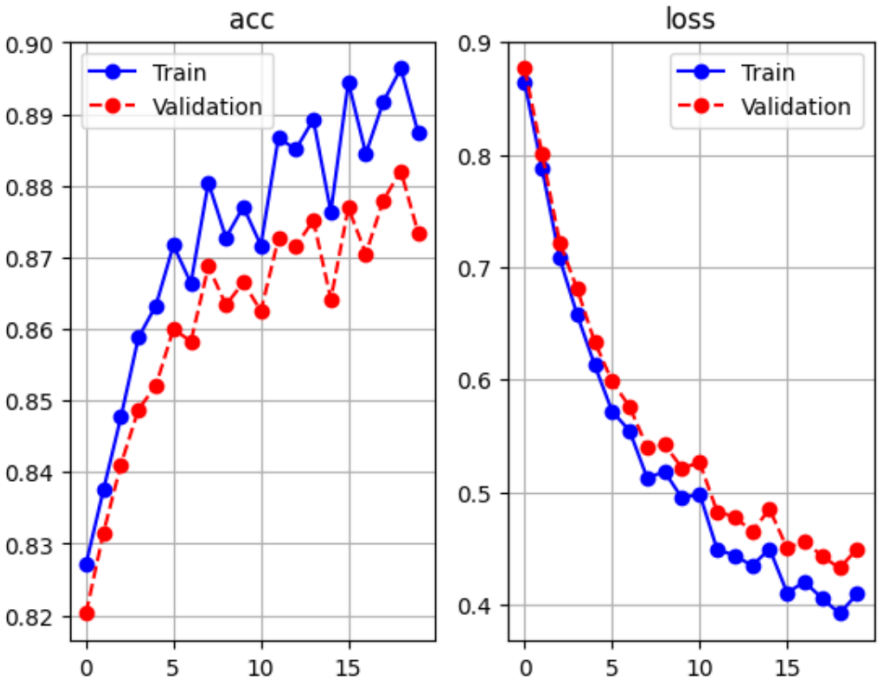
The highest accuracy is related to the adam optimizer, and then the highest accuracy is related to the Adagrad optimizer, but this difference is not significant; On the other hand, in the second graph, the training speed of the model with the Adagrad optimizer is higher than that of Adam, and we can ignore the insignificant difference in the

accuracy obtained in the two optimizers and consider the Adagrad optimizer as the best optimizer.

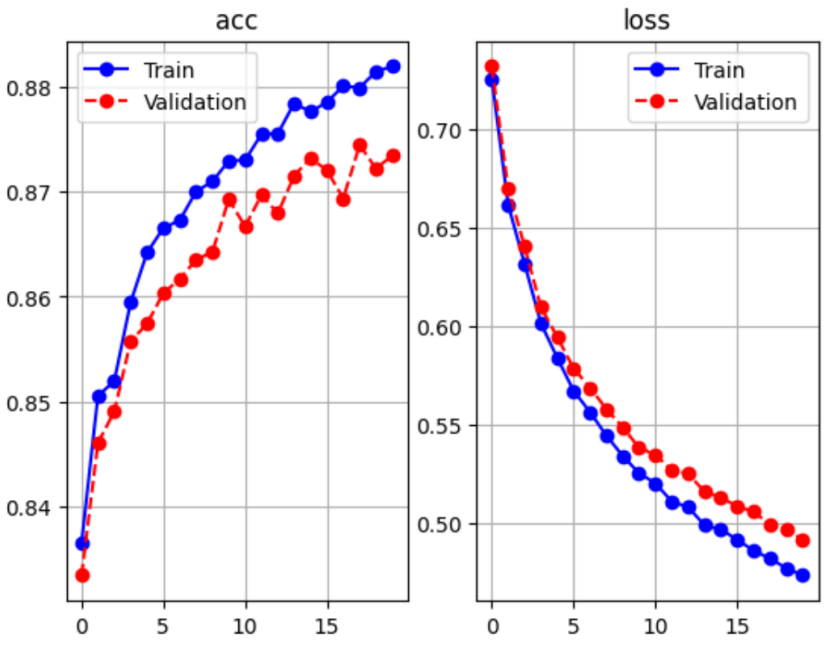
Accuracy and Loss graphs for training and validation data for optimizers are shown below.



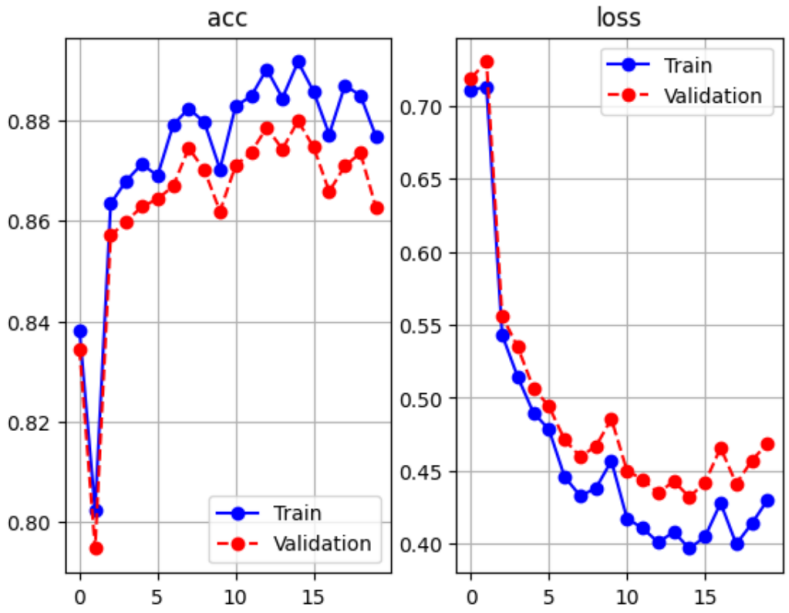
SGD



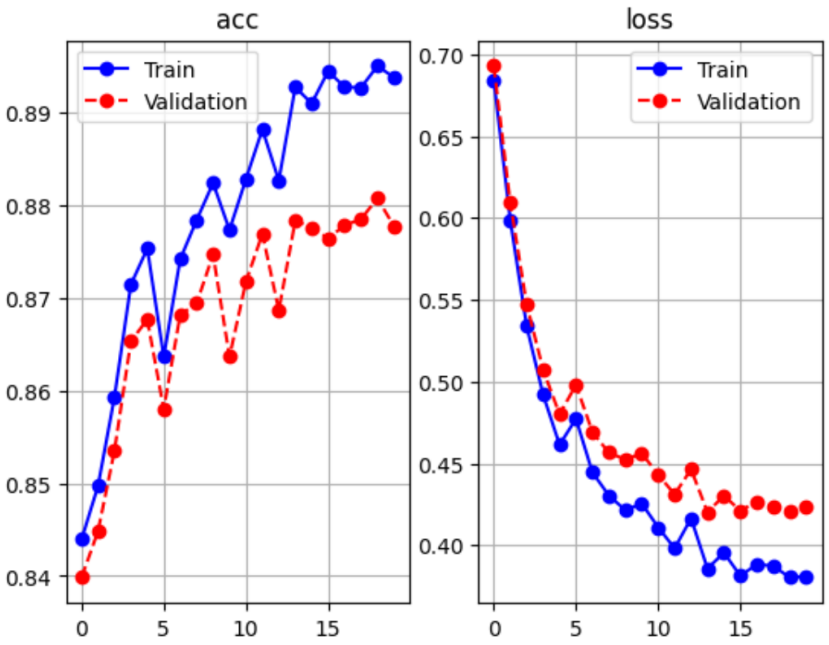
Momentum



Adagrad



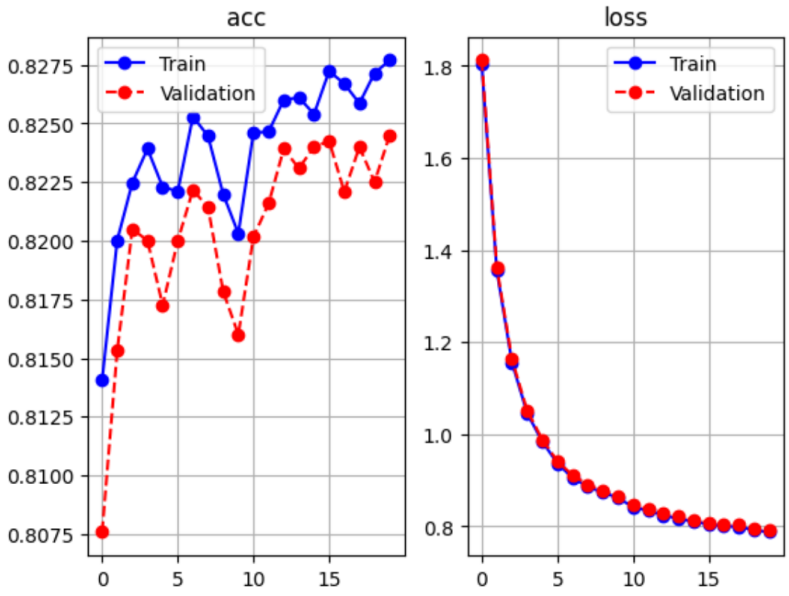
RMSProp



Adam

Regularization is different

The best model so far is a model with 50 neurons and the Adagrad optimizer tested with L2 regularization. Now here we test the model with different L1 regularization and examine the result. Its loss and accuracy chart is as follows**.**



Adagrad with L1/L2 Regularization	Train Accuracy	Train Loss	Val Accuracy	Val Loss	Time Computation
L1	0.827	0.787	0.824	0.792	77.16
L2	**0.881**	**0.479**	**0.869**	**0.5**	**58.52**

The comparison of the model with the Adagrad optimizer with two different regularizations is as follows.

According to the above table, the model with Adagrad optimizer with L2 regularization has higher accuracy and its training time is less than the model with L1 regularization.

Conclusion

According to the experiments, the best model is a model with 50 neurons and Adagrad optimizer and L2 regularization with epoch equal to 20.

By evaluating this model on test data, the accuracy of the model is equal to 85%.

```
model.evaluate(X_test, y_test)
```

```
{'loss': 0.5243740978476543, 'acc': 0.859}
```

Because in the previous section, the best accuracies were related to Adam and Adagrad, and we chose Adagrad due to its faster speed. To test the accuracy of the model, we also tested it with Adam on the test data, and the final accuracy was 86%.

```
model.evaluate(X_test, y_test)
```

```
{'loss': 0.4528363487553906 4, 'acc': 0.8651}
```

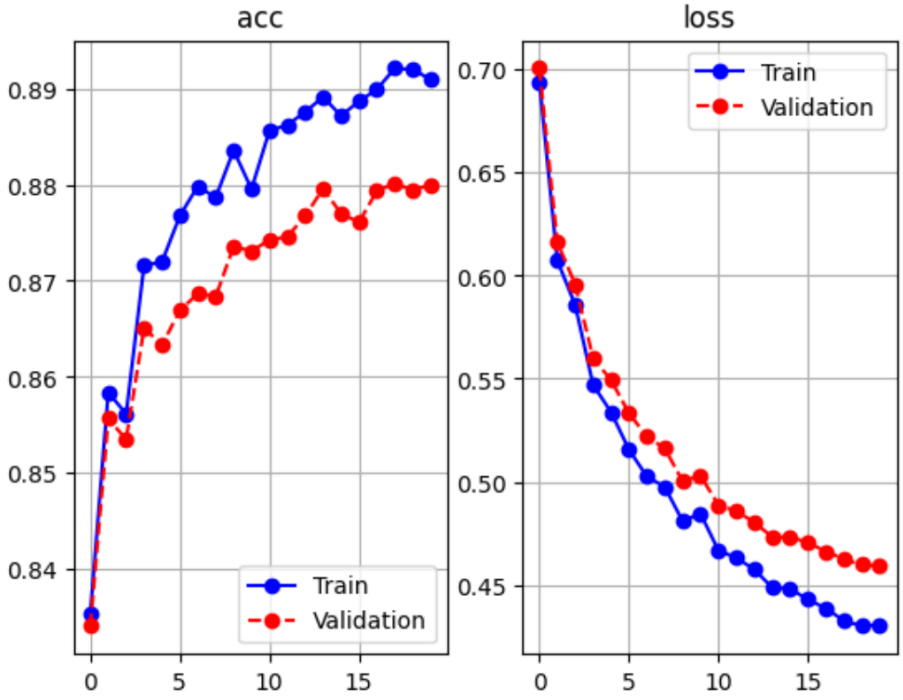
Architectural change

In the conducted experiments, the appropriate optimizer and regularization were chosen and to increase the accuracy, an attempt was made to change the architecture of the neural network. All architectures are trained with adagrad optimizer and L2 regularization and the results are reported.

First Architecture

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input                        (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 75)                58875
_________________________________________________________________
dense_2 (Dense)              (None, 50)                3800
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
_________________________________________________________________
```

```
Epoch 20
loss: 0.43067468993959246 acc: 0.8909583333333333 val_loss: 0.4590894608857973 val_acc: 0.88
Time:  87.52803468704224
```



The accuracy of this model on the test data is also equal to 86%.

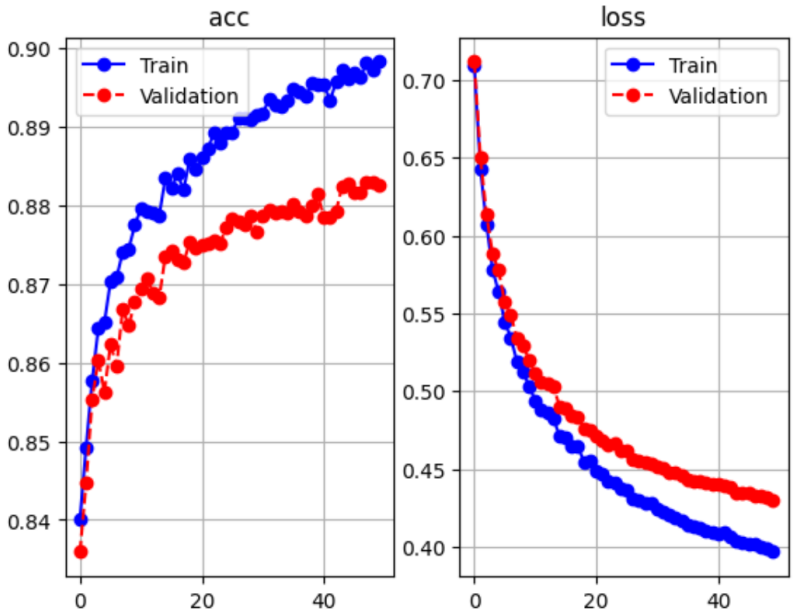
```
model.evaluate(X_test, y_test)
```

```
{'loss': 0.48961118201672693, 'acc': 0.8677}
```

The second architecture

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input                        (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 75)                58875
_________________________________________________________________
dense_2 (Dense)              (None, 10)                760
_________________________________________________________________
```

```
Epoch 50
loss: 0.3967624938373994 acc: 0.8982708333333334 val_loss: 0.42959262697411355 val_acc: 0.8825833333333334
Time:  176.71826481819153
```



The accuracy of this model on test data is also equal to 87%.

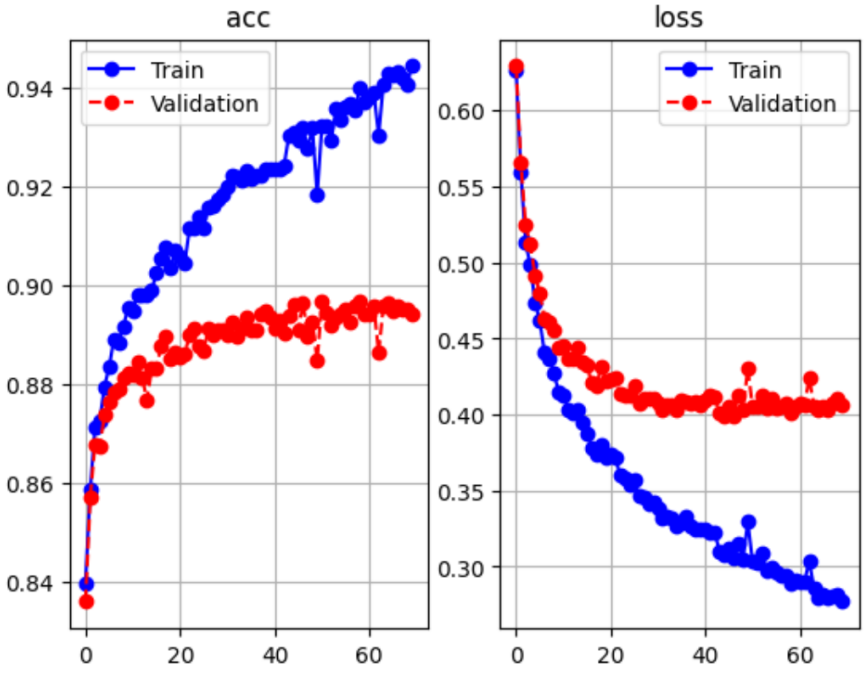
```
model.evaluate(X_test, y_test)
```

```
{'loss': 0.4594265105270959, 'acc': 0.8729}
```

The third architecture

Layer (type)	Output Shape	Param #
Input	(None, 784)	0
dense_1 (Dense)	(None, 175)	137375
dense_2 (Dense)	(None, 75)	13200
dense_3 (Dense)	(None, 50)	3800
dense_4 (Dense)	(None, 10)	510

```
Epoch 70
loss: 0.27694813390720907 acc: 0.94425 val_loss: 0.4067065922330842 val_acc: 0.8940833333333333
Time:  697.5085396766663
```



The accuracy of this model on the test data is also equal to 88%.

```
model.evaluate(X_test, y_test)
```

```
{'loss': 0.4342060006472418, 'acc': 0.8873}
```