Task 2: Feed Forward Neural Network for Multi-class Classification

Nghia Nim

Tasnim Ahmed

Helin Mazi

Ameena Zewail

New York University Abu Dhabi

29 February 2024

A. Introduction:

Feedforward neural networks are artificial neural networks in which nodes do not form loops. This type of neural network is also known as a multi-layer neural network as all information is only passed forward. During data flow, input nodes receive data, which travel through hidden layers, and exit output nodes.

a. Architecture of Feedforward Neural Networks

The architecture of a feedforward neural network consists of three types of layers: *the input layer, hidden layers, and the output layer*. Each layer is made up of units known as neurons, and the layers are interconnected by weights.

- Input Layer: This layer consists of neurons that receive inputs and pass them on to the next layer. The number of neurons in the input layer is determined by the dimensions of the input data.
- Hidden Layers: These layers are not exposed to the input or output and can be considered as the computational engine of the neural network. Each hidden layer's neurons take the weighted sum of the outputs from the previous layer, apply an activation function, and pass the result to the next layer. The network can have zero or more hidden layers.
- Output Layer: The final layer that produces the output for the given inputs. The number of neurons in the output layer depends on the number of possible outputs the network is designed to produce.

Each neuron in one layer is connected to every neuron in the next layer, making this a fully connected network. *The strength of the connection between neurons is represented by weights, and learning in a neural network involves updating these weights based on the error of the output.*

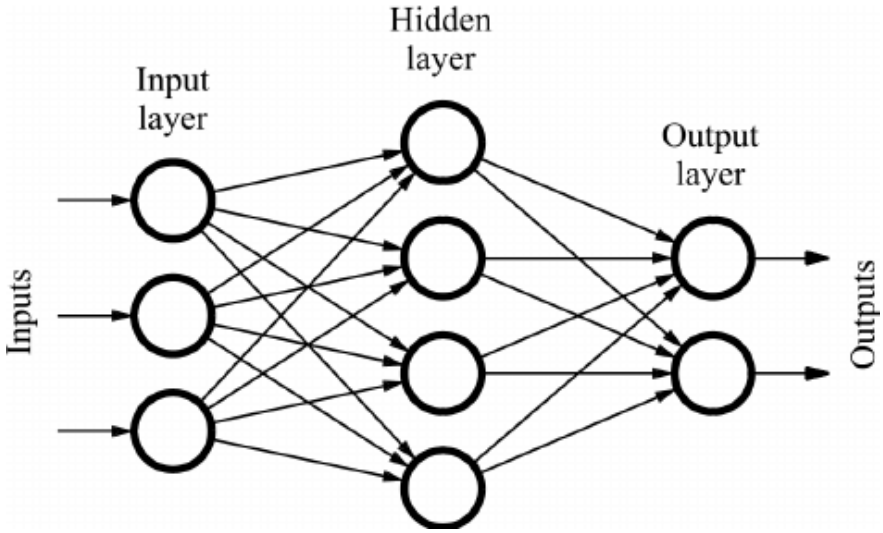

Figure 1: Basic neural network architecture

B. Objective:

The objective of this project is to implement and train a simplified neural network from scratch with linear connections between nodes for multi-classification tasks. Our implemented model should be capable of differentiating and classifying various clothing items present in the FashionMNIST dataset. Our goal is to find a suitable architecture, where the weights and biases present in input, hidden and output layers will be optimized through the minimization of the cross-entropy loss function. This optimization aims to produce a model that achieves the highest accuracy in the classification task.

C. Methodology

We chose to implement a simple neural network architecture with linear fully connected layers instead of implementing a convolutional neural network to avoid an increase in time complexity. Below we describe the dataset, our architecture and code implementation for designing and training the neural network.

Note: In the notebook, we have two different models implemented to showcase which one performs better. Below we describe the chosen model which is labelled as the "Final Model" in the notebook.

a. Dataset

The MNIST Fashion dataset is similar to the original MNIST dataset but consists of images of fashion items rather than handwritten digits. Each image in the dataset is 28x28 pixels, grayscale (meaning that each pixel value has a range from 0 to 255) and has 10 labels to classify.

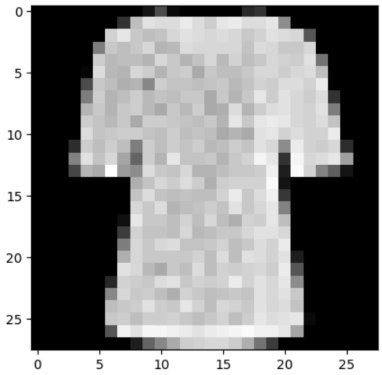

Figure 2: Each image is 28x28 pixels

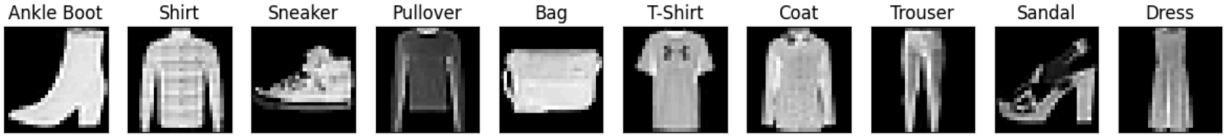

Figure 3: The unique classes of the dataset with their images

b. Architecture of the Network

As shown in Fig. 4, our model has an input layer, 3 hidden layers and an output layer. Given the fact we have 28x28 pixels for each image in our dataset, the input layer needs to have 28x28 = 784 neurons for each pixel in the image. The input layer forwards those pixel values to the first hidden layer with 1024 neurons which transforms the values and forwards them to the next hidden layer and so on:

```
FashionNN2(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=784, out_features=1024, bias=True)
  (dropout1): Dropout(p=0.2, inplace=False)
  (fc2): Linear(in_features=1024, out_features=256, bias=True)
  (dropout2): Dropout(p=0.2, inplace=False)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (dropout3): Dropout(p=0.2, inplace=False)
  (fc4): Linear(in_features=128, out_features=10, bias=True)
)
```

Figure 4: Architecture of our implemented model

Input Layer (784 neurons) → Hidden Layer 1(1024 Neurons) → Hidden Layer 2 (256 Neurons) → Hidden Layer 3 (128 Neurons) → Output Layer (10 Neurons)

After each hidden layer, we have incorporated dropout as a regularization technique to randomly deactivate 20% of the nodes within the layer. This strategy helps prevent overfitting by reducing co-dependency among units, thus enhancing the model's generalization capability. The neurons in each of the layers decrease progressively until we reach the output layer where we only have 10 neurons. We have 10 neurons in the output layers because each neuron corresponds to a class in the FashionMNIST dataset, outputting the probability for each case and the model predicts the class based on the neuron that has the highest probability.

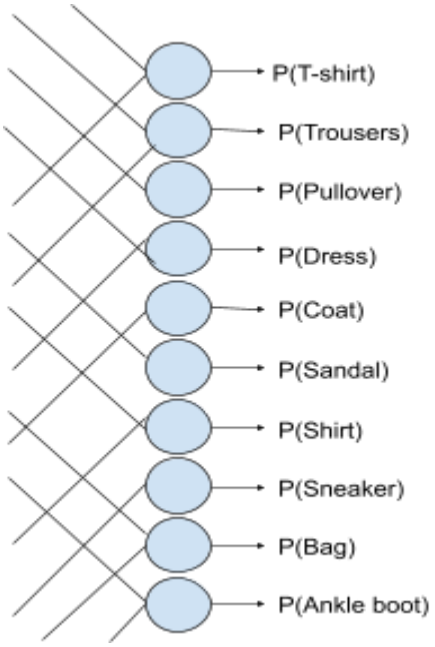

Figure 5. The output layer of our neural network

c. Training & Testing the Model

For training our model we implemented the basic strategy that is used in training Deep Learning models. As provided in the notebook from ATRC, we implemented the Cross-Entropy Loss function as the objective minimization function and the Stochastic Gradient Descent optimizer to minimize the loss function during the training of neural networks with different learning rates and momentums (explained in the following section). We train our model with a fixed number of 10 epochs.

```
net = FashionNN2()  # Re-initialize the network for each set of hyperparameters
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr, momentum=momentum)
num_epochs = 10
loss_history = []

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 200 == 199:  # log every 200 mini-batches
            current_loss = running_loss / 200
            loss_history.append(current_loss)
            running_loss = 0.0
```

Figure 6: Training loop - FashionNN2 is our model here

```
def evaluate_accuracy(model, dataloader):
    correct = 0
    total = 0
    predictions_list = []
    labels_list = []
    with torch.no_grad():
        for data in dataloader:
            images, labels = data
            labels_list.append(labels)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            predictions_list.append(predicted)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return (correct / total), labels_list, predictions_list
```

Figure 7: Model evaluation function - we send the trained model to this function for testing on the given test dataset. We implemented this as a function because of hyperparameter testing

d. Hyperparameter Testing

As mentioned earlier, we minimize the Cross-Entropy Loss function with the Stochastic Gradient Descent optimizer in the training process. Our hyperparameters in this case are the learning rate and momentum for the optimizer. To find the optimal hyperparameter value we train the models with the following values:

- Learning rates = [0.3, 0.03, 0.003]
- Momentums = [0.9, 0.8, 0.5]

D. Discussion

Given the number of values for learning rate and momentum, we have implemented a total of 9 different models based on different combinations of learning rate and momentum. As shown in the graph in Fig. 8, we can observe the varying performance of each model based on their learning rate and momentum. The choice of learning rate can significantly impact the performance of gradient descent. If the learning rate is too high, the algorithm may overshoot the minimum, and if it is too low, the algorithm may take too long to converge. We play around with the momentum value because it is a method used in optimization algorithms, specifically in gradient descent optimization, to improve convergence speed and avoid getting stuck in local minima.

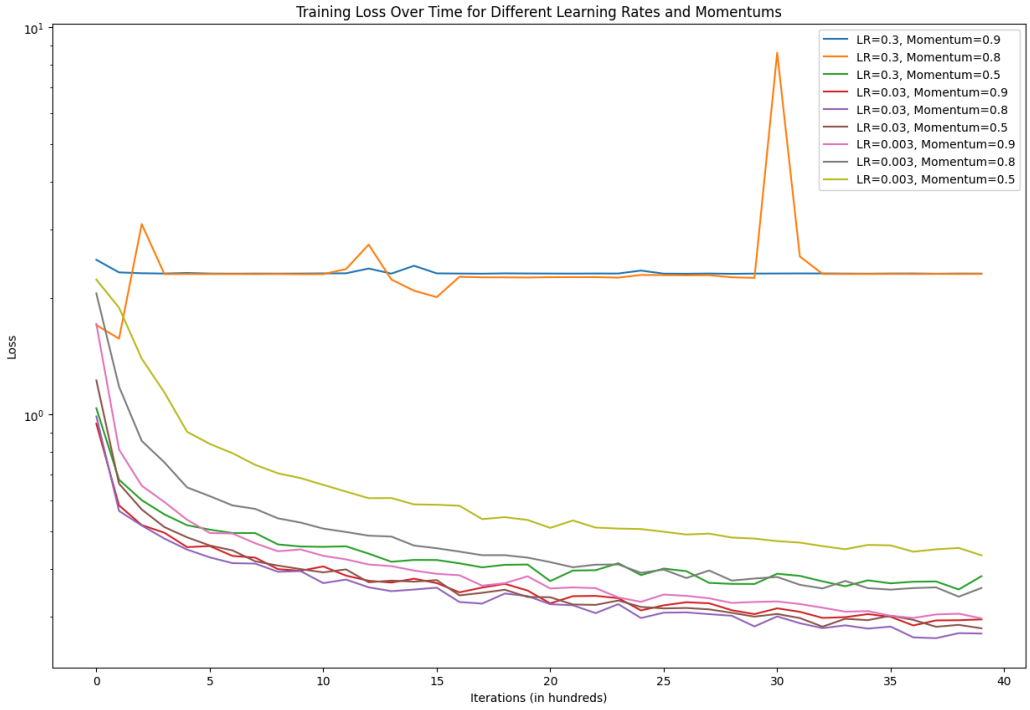

Figure 8: Loss vs Iteration graph for each of the 9 models

From the graph above we can see that as the learning rate increases so does the loss, which implies that a higher learning rate is not necessarily good for model optimization because it doesn't converge to the best minima of the loss function. As for momentum, it can't be generalized because it is dependent on the learning rate.

E. Conclusion

In Fig. 8 we can see that the model with a learning rate of 0.03 and momentum of 0.8 gives us the least loss value at the final iterations, which leads us to conclude that this model is the optimal model since it gives us the highest accuracy compared to all the other models implemented with different hyperparameters.

Final Parameters:

- *Epochs: 10*
- *Learning Rate: 0.03*
- *Momentum: 0.8*

To potentially achieve higher accuracy, we could have opted for a convolutional neural network (CNN) to leverage its ability to capture intricate patterns within the data. However, to avoid an increase in time complexity, we chose to focus on the construction of simple fully connected layers.

F. References

Turing, Understanding Feed Forward Neural Networks With Maths and Statistics, https://www.turing.com/kb/mathematical-formulation-of-feed-forward-neural-network