Deep Learning and Natural Language Processing Project

Moria Grohar, Shirel Zecharia

March 1, 2024

Contents

1 Abstract **3**

2 Introduction **3**

3 Related Work and Required Background **3**
3.1 Softmax Regression . . . 3
3.2 Activation Functions . . . 4
3.3 Simple Neural Networks . . . 5
3.3.1 Mathematical Representation . . . 5
3.3.2 Forward Propagation . . . 5
3.3.3 Backpropagation . . . 5
3.4 Convolutional Neural Networks . . . 6
3.4.1 Convolutional Layers . . . 6
3.4.2 Pooling Layers . . . 6
3.4.3 Fully Connected Layers . . . 6
3.5 Optimizers . . . 6
3.5.1 Stochastic Gradient Descent (SGD) . . . 6
3.5.2 Momentum SGD . . . 7
3.5.3 RMSprop (Root Mean Square Propagation) . . . 7
3.5.4 Adam (Adaptive Moment Estimation) . . . 7

4 Previous Attempts **8**
4.1 First Attempts with different data . . . 8
4.1.1 Dummy Model: . . . 8
4.1.2 Linear Regression: . . . 8
4.1.3 Ridge Regression: . . . 8
4.1.4 Random Forest: . . . 9
4.1.5 Linear Regression with TensorFlow: . . . 9
4.1.6 Neural Network Exploration: . . . 9
4.1.7 Comparison and Conclusion: . . . 9
4.2 Second Attempts with Fashion-MNIST data (our data) . . . 10
4.2.1 Soft max: . . . 10
4.2.2 CNN: . . . 12
4.2.3 Simple Neural Network: . . . 13
4.3 Comparing Between Models . . . 15

5 Project Description **16**

6 Simulation Results **17**
6.1 Softmax Regression Model . . . 17
6.2 Simple Neural Network Model . . . 17
6.3 Convolutional Neural Network Model . . . 18

7 Code **19**
7.1 Softmax Regression 19
7.2 Simple Neural Network Model 20
7.3 CNN Model 22

8 Conclusion **23**

1 Abstract

In this report, we present a study on implementing a neural network model for the classification of fashion items using the Fashion MNIST dataset.
We compare the performance of the neural network approach with traditional machine learning algorithms such as Logistic Regression, and Multi-Layer Perceptron (MLP).
The report discusses the project methodology, experimental results, and conclusions drawn from the comparison.

2 Introduction

This report explores the performance of various machine learning models for classifying fashion items from the well-known Fashion-MNIST dataset. This dataset consists of grayscale images of various clothing articles, each belonging to a specific category such as T-shirts, trousers, and shoes. The task of classifying these images is considered a classification problem.

The report delves into the implementation and evaluation of different models, including several models from sklearn, neural network from tensorflow and CNN.

The models were evaluated based on their loss function, with the aim of achieving the lowest possible loss to indicate better performance in classifying the fashion items.

The report compares and analyzes the results of each model, providing insights into their strengths and weaknesses for this specific classification task. It also discusses potential areas for further exploration and improvement.

3 Related Work and Required Background

3.1 Softmax Regression

Softmax regression, also known as multinomial logistic regression, is a generalization of logistic regression. While in logistic regression we assumed that the labels were binary: $y(i) \in 0, 1$, In Softmax regression we are able to handle multiple labels $y(i) \in 1, \ldots, K$ where k is the number of classes.

In softmax regression, the goal is to predict the probability that an input sample belongs to each possible class. This is achieved by applying the softmax function to the output of a linear model. The linear model computes a score for each class based on the input features, and the softmax function converts these scores into probabilities.

Mathematically, given an input vector x, softmax regression computes the probability $P(y = i|x)$ that the input belongs to class i as follows:

$$P(y = i|x) = \frac{e^{x^T \cdot W_i + b_i}}{\sum_{j=1}^{k} e^{x^T \cdot W_j + b_j}}$$

Where:
y is the target variable (class).
x is the vector of input features.
T denotes the transpose applied to the vector.
W_i: the weight vector associated with class i.
b_i: the bias term associated with class i.

Understanding The Process:

Linear Transformation: For each class, it computes a linear combination of the input features using class-specific weights. This generates a score for each class, indicating its "fit" for the data point.

Softmax Function: This function takes the vector of scores and transforms them into probabilities. It ensures:

- Non-negativity: All probabilities are positive (between 0 and 1).
- Normalization: The sum of all probabilities for a single data point is always 1, representing a valid probability distribution.

The softmax function is often used as the final activation layer in multi-class neural networks as we will discuss later in the article.

3.2 Activation Functions

Activation functions are mathematical functions used to determine the output of a neuron, influencing how information flows through the network and ultimately shaping its learning and decision-making capabilities.

Traditional neural networks without activation functions would simply perform linear transformations on the input data. This limitation restricts them from learning complex patterns and relationships in the data, hindering their ability to perform tasks like image recognition or natural language processing.

A neuron in a neural network receives multiple inputs, each carrying a specific value. These inputs are combined through weighted connections and then passed through an activation function. The function acts like a filter, transforming the combined input into a single output value. This output value then becomes the basis for the neuron's activation, influencing the signals it transmits to other neurons in the network.

The selection of an appropriate activation function depends on the specific task and network architecture. While ReLU and Leaky ReLU are popular choices due to their efficiency, other functions like sigmoid and tanh might be suitable for specific scenarios like output normalization or recurrent neural networks.

Popular Activation Functions:

- ❖ Rectified Linear Unit: ReLU (Rectified Linear Unit) activation functions serve as the non-linear activation function of choice in most CNN architectures. ReLU introduces non-linearity into the network, enabling it to learn complex mappings between the input and output. Mathematically, ReLU simply outputs the input if it is positive and zero otherwise, which simplifies computations and accelerates convergence during training.
- ❖ Leaky Rectified Linear Unit: Leaky ReLU is a variant of the ReLU activation function that addresses the "dying ReLU" problem, where neurons may become inactive and cease to update their weights during training. Unlike ReLU, which sets negative values to zero, Leaky ReLU allows a small, non-zero gradient for negative inputs. This modification ensures that neurons remain active and continue to contribute to the learning process, especially in deeper networks where vanishing gradients are more prevalent.
- ❖ Sigmoid: This function squishes the input values between 0 and 1, mimicking the firing behavior of biological neurons. However, its vanishing gradients can hinder learning in deep networks.
- ❖ Tanh (Hyperbolic tangent): Similar to sigmoid, tanh maps the input values to a range of -1 to 1. It offers slightly better gradients than sigmoid but suffers from similar limitations.

3.3 Simple Neural Networks

A simple neural network consists of interconnected neurons organized in layers. Each neuron calculates a weighted sum of its inputs, applies an activation function and passes the result on to the next layer.

3.3.1 Mathematical Representation

Let:

- x be the input vector,
- W be the weight matrix,
- b be the bias vector,
- f be the activation function,
- z be the weighted sum of inputs,
- a be the output after applying the activation function.

For a single neuron in a layer, the computation can be represented as:

$$z = Wx + b$$
$$a = f(z)$$

With multiple neurons in a layer, the calculations are performed in parallel, resulting in matrix multiplication for the weighted sum and the element-wise application of the activation function.

3.3.2 Forward Propagation

In forward propagation, the input data is fed into the network and the calculations are performed layer by layer until the final output is obtained.

1. **Input Layer**: The input data x is fed into the network.
2. **Hidden Layers**: Each hidden layer computes the weighted sum of inputs and applies the activation function to produce the output.
3. **Output Layer**: The final hidden layer's output is passed through the output layer, which produces the network's prediction.

Mathematically, forward propagation can be expressed as:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = f(z^{[l]})$$

where l denotes the layer index.

3.3.3 Backpropagation

Backpropagation is used to update the weights and biases of the network to minimize the loss function. Gradients are calculated and the parameters are adjusted using gradient descent.

The most important equations for backpropagation are the chain rule for calculating the gradients and the equations for updating the weights and biases:

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}}$$
$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial W^{[l]}}$$
$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial b^{[l]}}$$

where $\mathcal{L}$ represents the loss function.

3.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specific type of artificial neural network architecture particularly well-suited for analyzing image data. They excel at tasks like image classification, object detection, and image segmentation. Unlike traditional neural networks, CNNs incorporate an additional layer type called a convolutional layer to extract features from the input data. The essence of a CNN lies in its ability to automatically learn spatial hierarchies of features from raw data. These networks consist of several layers, typically including convolutional layers, pooling layers, and fully connected layers.

3.4.1 Convolutional Layers

Convolutional layers are the cornerstone of CNNs, responsible for learning and detecting patterns within the input data. They employ mathematical operations known as convolutions to extract features such as edges, textures, and other distinctive attributes from the input images. Mathematically, convolution involves sliding a filter or a kernel over the input data, performing element-wise multiplications, and summing the results to produce feature maps.

3.4.2 Pooling Layers

Pooling layers reduce the spatial dimensions of feature maps generated by convolutional layers while retaining important information. Common pooling techniques include max pooling and average pooling, where the former selects the maximum value from a region of the feature map, while the latter computes the average. Pooling helps in achieving translation invariance and reducing the computational burden by downsampling the feature maps.

3.4.3 Fully Connected Layers

These layers function similarly to traditional neural networks, connecting all neurons in one layer to all neurons in the next. They process the extracted features and learn complex relationships to form the final output, such as a class prediction or a set of bounding boxes for object detection.

3.5 Optimizers

Optimizers are used for updating the model's parameters to minimize the loss function during the training process, which ultimately leads to better performance of the model.

3.5.1 Stochastic Gradient Descent (SGD)

The SGD algorithm takes a single data point (or a small batch) at a time, calculates the gradient (the direction of steepest descent in the loss function), and updates the model's parameters in the opposite direction by a small learning rate. This technique is computationally inexpensive, making it suitable for large datasets. But on the other hand it may require many iterations to reach an optimal solution, especially for complex models and datasets.
The update rule for SGD can be represented as:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla L(\theta_t)$$

Where:
θ_t is the current parameter values.
η is the learning rate, a small positive scalar determining the step size.
$\nabla L(\theta_t)$ is the gradient of the function L at θ_t, which points in the direction of the steepest increase in L.

3.5.2 Momentum SGD

SGD with Momentum is used to improve the performance of the neural network. accelerates convergence, especially in the context of high curvature, small but consistent gradients, or noisy gradients. It does this by adding a fraction γ of the update vector of the past time step to the current update.
The update rule for Momentum SGD is:

$$v_t = \gamma \cdot v_{t-1} + \eta \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

Where:
v_t is the velocity at time step t, which is a combination of the current gradient and the velocity from the previous step v_{t-1}.
γ (often set between 0.9 and 0.99) is the momentum term, determining how much of the past velocity to add to the current step.

3.5.3 RMSprop (Root Mean Square Propagation)

RMSprop addresses a limitation of SGD, where frequent updates based on noisy gradients can lead to slow convergence. Its key Feature is that it incorporates an exponential moving average of the squared gradients for each parameter. This averages the recent updates and reduces the impact of large, infrequent gradients, leading to smoother convergence. This technique has an increased computational cost compared to SGD and RMSprop.This technique has an increased computational cost compared to SGD.

$$s_{t+1} = \beta \cdot s_t + (1 - \beta) \cdot (\nabla L(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_{t+1}} + \epsilon} \cdot \nabla L(\theta_t)$$

3.5.4 Adam (Adaptive Moment Estimation)

The Adam optimizer combines ideas from SGD, RMSprop, and momentum, aiming to adapt the learning rate for each parameter individually and accelerate convergence. It maintains exponential moving averages of gradients (similar to RMSprop) and squared gradients. It incorporates momentum to account for the history of past updates, addressing the issue of getting stuck in local minima. It adaptively adjusts the learning rate for each parameter based on these estimates, aiming for optimal updates. This technique has an increased computational cost compared to SGD and RMSprop.

$$m_{t+1} = \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla L(\theta_t)$$
$$v_{t+1} = \beta_2 \cdot v_t + (1 - \beta_2) \cdot (\nabla L(\theta_t))^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \cdot \hat{m}_{t+1}$$

4 Previous Attempts

4.1 First Attempts with different data

Model Selection and Evaluation:

The project explored the performance of various models for predicting flight ticket prices in a dataset containing information on bookings between major Indian cities. The target variable was the "Price" column, indicating a continuous value. Therefore, the problem was classified as a regression task.

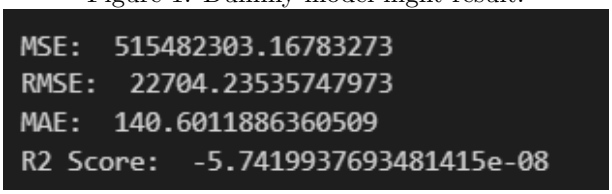
The data was split into train and test sets for model evaluation. Several models were implemented and compared based on their performance in reducing the mean squared error (MSE), which served as the loss function.

Model Performances:

4.1.1 Dummy Model:

This baseline model adopted a simple mean strategy, resulting in high loss values.

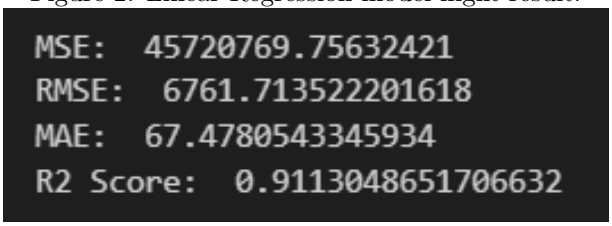
Figure 1: Dummy model flight result:



4.1.2 Linear Regression:

Compared to the dummy model, linear regression showed improvement but still yielded significant loss.

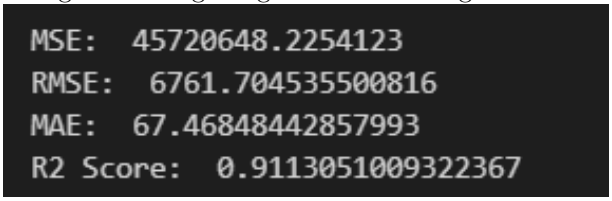
Figure 2: Linear Regression model flight result:



4.1.3 Ridge Regression:

A slight reduction in loss compared to linear regression was observed, but the values remained high.

Figure 3: Ridge Regression model flight result:



4.1.4 Random Forest:

This model achieved a more significant reduction in loss compared to prior models, although the value remained high.

Figure 4: Random Forest model flight result:

```
MSE:  7771155.464475996
RMSE:  2787.679225534386
MAE:  32.83148419706723
R2 Score:  0.9849244952485496
```

4.1.5 Linear Regression with TensorFlow:

Implementing linear regression using TensorFlow did not improve performance and even resulted in higher loss compared to the basic implementation.

Figure 5: Linear Regression with TensorFlow model flight result:

```
Epoch: 1000 Loss: 149983800.0
Test Loss: 149370112.0
```

4.1.6 Neural Network Exploration:

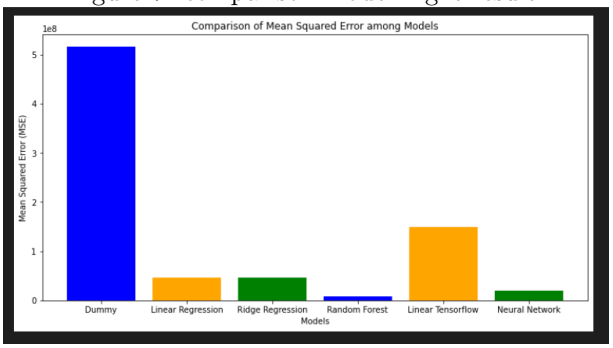
A neural network model with 2 hidden layers and 80 hidden units was constructed using TensorFlow. Hyperparameter tuning involved adjusting the learning rate and number of epochs, ultimately leading to the configuration with the lowest observed loss.

Figure 6: neural network model flight result:

```
Epoch: 10000 Loss: 17160926.0
Test Loss: 17839964.0
```

4.1.7 Comparison and Conclusion:

Figure 7: comparison model flight result:



While the neural network achieved competitive results compared to simpler models after hyperparameter adjustments, the random forest remained the best performing model based on the achieved loss values.

4.2 Second Attempts with Fashion-MNIST data (our data)

Model Selection and Evaluation:

The project explored the performance of various models for classification of a picture of cloth to a various category. Therefore, the problem was classified as a classification task.

The data was split into train and test sets for model evaluation, in some models also to validation. Several models were implemented and compared based on their performance in reducing the loss function.

Approach:

In this study, the waterfall approach is adopted, characterized by systematically running the model with predefined epochs and various parameters governing the model's architecture, such as the number of layers in the neural network and other relevant factors. This method aims to systematically explore the model's performance across different configurations and settings.

Model Performances:

4.2.1 Soft max:

A function that turns a vector of K real values into a vector of K real values that sum to 1.

In an effort to optimize the performance of the model's performance, various hyperparameters were explored. The number of epochs controlling the training iterations was adjusted. Different optimizers responsible for updating the model weights were investigated. The split of data for training, validation and testing was also investigated. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy.

The accuracy calculation process.

Figure 8: Running the model:

```
SoftMax model:
Epoch 1/20
1860/1860 [==============================] - 4s 2ms/step - loss: 0.6077 - accuracy: 0.7960 - val_loss: 0.5030 - val_accuracy: 0.8265
Epoch 2/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4668 - accuracy: 0.8398 - val_loss: 0.4755 - val_accuracy: 0.8334
Epoch 3/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4410 - accuracy: 0.8490 - val_loss: 0.4552 - val_accuracy: 0.8446
Epoch 4/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4260 - accuracy: 0.8533 - val_loss: 0.4338 - val_accuracy: 0.8512
Epoch 5/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4184 - accuracy: 0.8553 - val_loss: 0.4448 - val_accuracy: 0.8473
Epoch 6/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4125 - accuracy: 0.8579 - val_loss: 0.4258 - val_accuracy: 0.8551
Epoch 7/20
1860/1860 [==============================] - 3s 1ms/step - loss: 0.4065 - accuracy: 0.8593 - val_loss: 0.4235 - val_accuracy: 0.8552
Epoch 8/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.4034 - accuracy: 0.8597 - val_loss: 0.4265 - val_accuracy: 0.8540
Epoch 9/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.3994 - accuracy: 0.8609 - val_loss: 0.4192 - val_accuracy: 0.8584
Epoch 10/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.3961 - accuracy: 0.8609 - val_loss: 0.4146 - val_accuracy: 0.8600
Epoch 11/20
1860/1860 [==============================] - 2s 1ms/step - loss: 0.3944 - accuracy: 0.8620 - val_loss: 0.4287 - val_accuracy: 0.8516
Epoch 12/20
1860/1860 [==============================] - 4s 2ms/step - loss: 0.3923 - accuracy: 0.8625 - val_loss: 0.4281 - val_accuracy: 0.8550
Epoch 13/20
1860/1860 [==============================] - 4s 2ms/step - loss: 0.3897 - accuracy: 0.8627 - val_loss: 0.4286 - val_accuracy: 0.8526
Epoch 14/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3879 - accuracy: 0.8650 - val_loss: 0.4132 - val_accuracy: 0.8594
Epoch 15/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3856 - accuracy: 0.8649 - val_loss: 0.4165 - val_accuracy: 0.8576
Epoch 16/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3843 - accuracy: 0.8655 - val_loss: 0.4163 - val_accuracy: 0.8597
Epoch 17/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3841 - accuracy: 0.8639 - val_loss: 0.4230 - val_accuracy: 0.8564
Epoch 18/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3822 - accuracy: 0.8669 - val_loss: 0.4177 - val_accuracy: 0.8552
Epoch 19/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3803 - accuracy: 0.8656 - val_loss: 0.4291 - val_accuracy: 0.8536
Epoch 20/20
1860/1860 [==============================] - 3s 2ms/step - loss: 0.3810 - accuracy: 0.8663 - val_loss: 0.4251 - val_accuracy: 0.8557
329/329 [==============================] - 0s 930us/step - loss: 0.4251 - accuracy: 0.8557
```

The result of the run:

Figure 9: Result:

```
Test Loss: 0.4250730574131012
Test Accuracy: 0.8557142615318298
```

The classification report in concentrated form:

Figure 10: Classification Report:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.78      0.81      1078
           1       0.99      0.97      0.98      1036
           2       0.83      0.67      0.74      1084
           3       0.82      0.90      0.86      1026
           4       0.71      0.83      0.76      1029
           5       0.94      0.94      0.94      1058
           6       0.63      0.67      0.65      1065
           7       0.91      0.95      0.93      1088
           8       0.97      0.94      0.95      1018
           9       0.97      0.93      0.95      1018

    accuracy                           0.86     10500
   macro avg       0.86      0.86      0.86     10500
weighted avg       0.86      0.86      0.86     10500
```

All the data of the epochs can be shown in a graph that shows the loss function and the accuracy in every run.

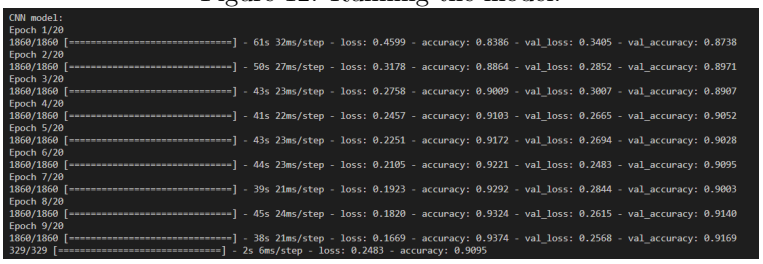
Figure 11: Soft max training history:

4.2.2 CNN:

A regularized type of feed-forward neural network that learns feature engineering by itself via filters (or kernel) optimization. In an effort to optimize the performance of the model's performance, various hyper-parameters were explored. The number of epochs controlling the training iterations was adjusted. Different optimizers responsible for updating the model weights were investigated. The split of data for training, validation and testing was also investigated. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy.

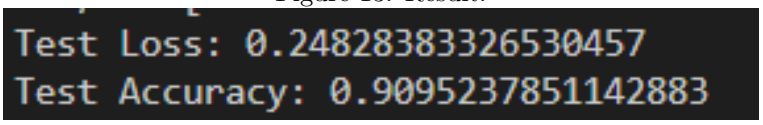
The accuracy calculation process:

Figure 12: Running the model:



The result of the run:

Figure 13: Result:



The classification report in concentrated form:

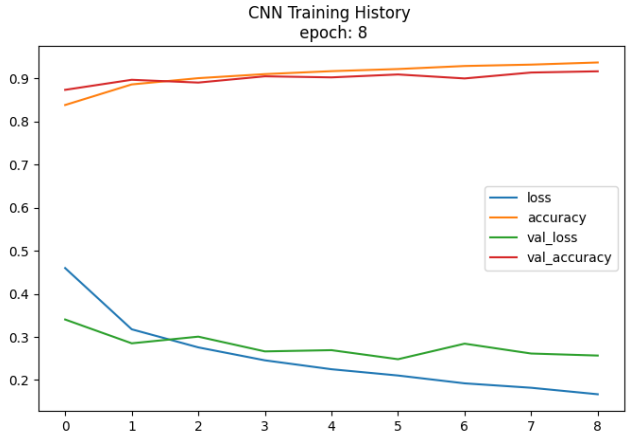
Figure 14: Classification Report:

Classification Report:

	precision	recall	f1-score	support
0	0.85	0.86	0.85	1078
1	0.99	0.99	0.99	1036
2	0.92	0.80	0.86	1084
3	0.90	0.92	0.91	1026
4	0.84	0.87	0.86	1029
5	0.98	0.98	0.98	1058
6	0.71	0.79	0.75	1065
7	0.97	0.96	0.97	1088
8	0.99	0.97	0.98	1018
9	0.97	0.97	0.97	1018
accuracy			0.91	10500
macro avg	0.91	0.91	0.91	10500
weighted avg	0.91	0.91	0.91	10500

All the data of the epochs can be shown in a graph that shows the loss function and the accuracy in every run.

Figure 15: CNN training history:



4.2.3 Simple Neural Network:

A group of interconnected units called neurons that send signals to one another. Neurons can be either biological cells or mathematical models. While individual neurons are simple, many of them together in a network can perform complex tasks.In an effort to optimize the performance of the model's performance, various hyperparameters were explored. The number of epochs controlling the training iterations was adjusted. Different optimizers responsible for updating the model weights were investigated. The split of data for training, validation and testing was also investigated. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy.

The accuracy calculation process:

Figure 16: Running the model:

```
Epoch 1/30
1772/1772 [==============================] - 5s 2ms/step - loss: 0.4817 - accuracy: 0.8259 - val_loss: 0.3929 - val_accuracy: 0.8583
Epoch 2/30
1772/1772 [==============================] - 6s 4ms/step - loss: 0.3661 - accuracy: 0.8665 - val_loss: 0.4036 - val_accuracy: 0.8535
Epoch 3/30
1772/1772 [==============================] - 5s 3ms/step - loss: 0.3297 - accuracy: 0.8787 - val_loss: 0.3265 - val_accuracy: 0.8841
Epoch 4/30
1772/1772 [==============================] - 6s 3ms/step - loss: 0.3059 - accuracy: 0.8860 - val_loss: 0.3282 - val_accuracy: 0.8856
Epoch 5/30
1772/1772 [==============================] - 5s 3ms/step - loss: 0.2880 - accuracy: 0.8916 - val_loss: 0.3180 - val_accuracy: 0.8867
Epoch 6/30
1772/1772 [==============================] - 5s 3ms/step - loss: 0.2731 - accuracy: 0.8974 - val_loss: 0.3237 - val_accuracy: 0.8854
Epoch 7/30
1772/1772 [==============================] - 6s 3ms/step - loss: 0.2602 - accuracy: 0.9026 - val_loss: 0.3279 - val_accuracy: 0.8792
Epoch 8/30
1772/1772 [==============================] - 6s 3ms/step - loss: 0.2485 - accuracy: 0.9054 - val_loss: 0.3142 - val_accuracy: 0.8876
Epoch 9/30
1772/1772 [==============================] - 5s 3ms/step - loss: 0.2357 - accuracy: 0.9107 - val_loss: 0.3217 - val_accuracy: 0.8898
Epoch 10/30
1772/1772 [==============================] - 6s 4ms/step - loss: 0.2264 - accuracy: 0.9136 - val_loss: 0.3156 - val_accuracy: 0.8913
Epoch 11/30
1772/1772 [==============================] - 7s 4ms/step - loss: 0.2189 - accuracy: 0.9173 - val_loss: 0.3118 - val_accuracy: 0.8921
Epoch 12/30
1772/1772 [==============================] - 7s 4ms/step - loss: 0.2116 - accuracy: 0.9195 - val_loss: 0.3384 - val_accuracy: 0.8884
Epoch 13/30
1772/1772 [==============================] - 5s 3ms/step - loss: 0.2049 - accuracy: 0.9222 - val_loss: 0.3130 - val_accuracy: 0.8989
Epoch 14/30
1772/1772 [==============================] - 4s 2ms/step - loss: 0.1963 - accuracy: 0.9254 - val_loss: 0.3874 - val_accuracy: 0.8817
219/219 [==============================] - 0s 1ms/step - loss: 0.3167 - accuracy: 0.8893
219/219 [==============================] - 0s 1ms/step - loss: 0.3167 - accuracy: 0.8893
```

The result of the run:

Figure 17: Result:

```
Test Loss: 0.31667688488960266
Test Accuracy: 0.8892857432365417
```

The classification report in concentrated form:

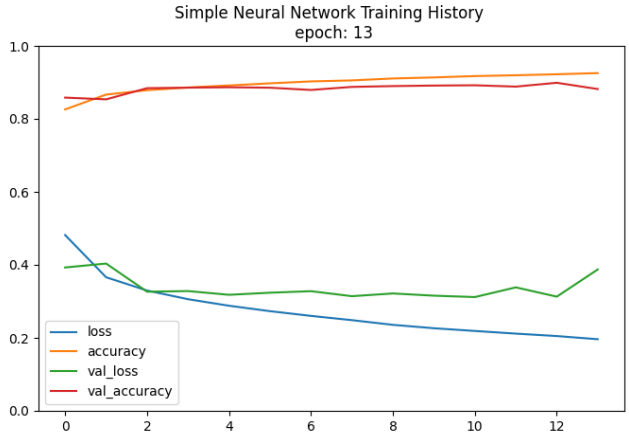
Figure 18: Classification Report:

Classification Report:

	precision	recall	f1-score	support
0	0.79	0.91	0.84	733
1	0.97	0.98	0.98	694
2	0.82	0.81	0.81	728
3	0.94	0.88	0.91	689
4	0.77	0.88	0.82	655
5	0.98	0.96	0.97	701
6	0.80	0.61	0.69	724
7	0.95	0.94	0.94	694
8	0.97	0.97	0.97	683
9	0.94	0.97	0.95	699
accuracy			0.89	7000
macro avg	0.89	0.89	0.89	7000
weighted avg	0.89	0.89	0.89	7000

All the data of the epochs can be shown in a graph that shows the loss function and the accuracy in every run.

Figure 19: Simple Neural Network training history:



4.3 Comparing Between Models

Figure 20: Comparing accuracy score between models:

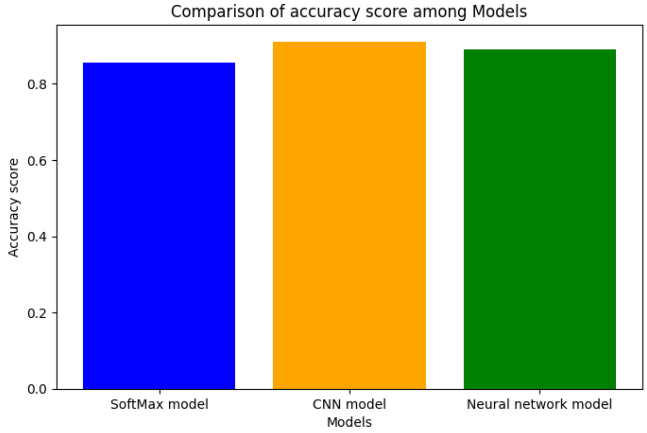
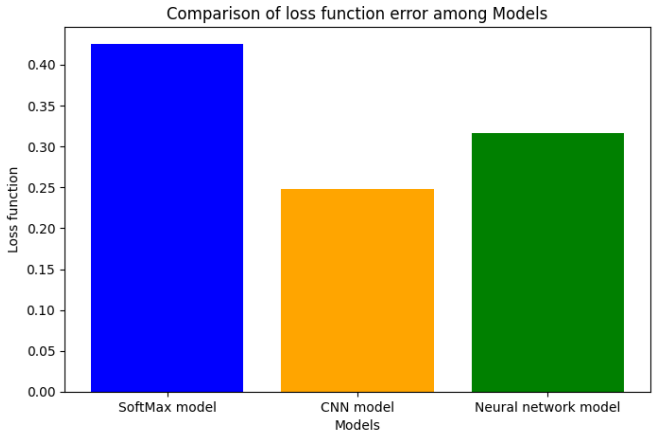


Figure 21: Comparing loss function between models:



The comparison shows that the CNN model has the lowest loss function and the highest accuracy among the evaluated models, indicating its superior performance.

5 Project Description

The optimal model architecture determined in this study is a Convolutional Neural Network (CNN) with the following specifications:

1. **input layer**:
   - Convolution layer with 32 filters of size 3x3.
   - Activation function: specified by the variable `activation`.
   - Input shape: determined by the variable `input_shape`.
2. **normalization**:
   - Batch normalization layer.
3. **Pooling Layers**:
   - MaxPooling2D layer with a pool size of 2x2.
4. **Convolutional Layers**:
   - Another convolutional layer with 64 filters of size 3x3.
   - Activation function: the same as specified in the variable `activation`.
5. **normalization**:
   - Batch normalization level.
6. **Pooling Layers**:
   - MaxPooling2D layer with a pool size of 2x2.
7. **Convolutional Layers**:
   - Third convolutional layer with 64 filters of size 3x3.
   - Activation function: the same as specified in the variable `activation`.
8. **Normalization**:
   - Batch normalization layer.
9. **Flattening Layer**:
   - Flattening layer for converting the 2D feature maps into a 1D vector.
10. **Dense Layers**:
    - Fully connected dense layer with 64 neurons.
    - Activation function: same as specified in the variable `activation`.
11. **Dropout**:
    - Dropout layer with a dropout rate specified by `dropout_rate`.
12. **Output Layer**:
    - Dense layer with the number of neurons equal to the number of classes: 10.
    - Activation function: softmax.

The result obtained by this CNN architecture is the lowest loss function and the highest accuracy among the models evaluated in the study. The results:

1. **Accuracy:** 91%
2. **Loss Function:** 0.25%

6 Simulation Results

6.1 Softmax Regression Model

Figure 22: Training history of the data

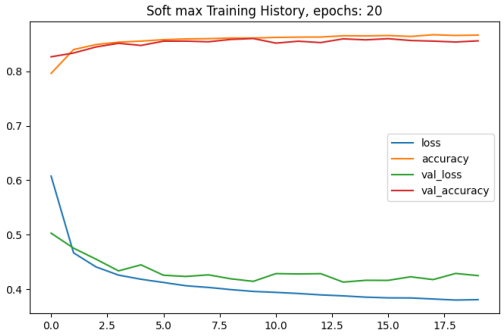


Figure 23: Output

```
Test Loss: 0.4250730574131012
Test Accuracy: 0.8557142615318298
```

6.2 Simple Neural Network Model

Figure 24: Training history of the data

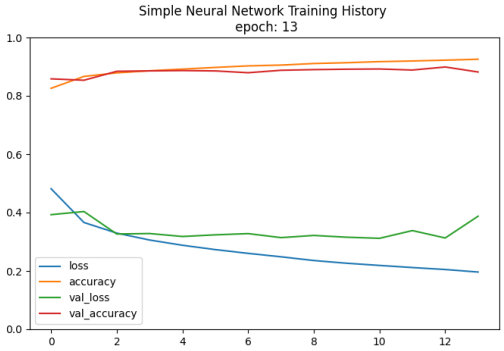


Figure 25: Output

```
Test Loss: 0.31667688488960266
Test Accuracy: 0.8892857432365417
```

6.3 Convolutional Neural Network Model

Figure 26: Training history of the data

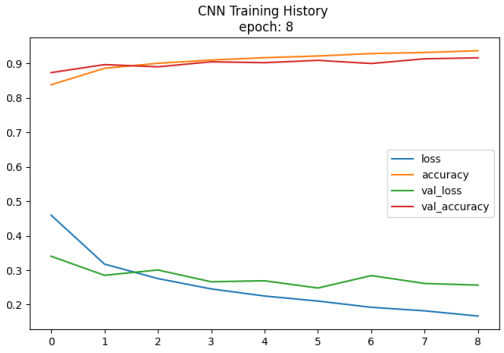


Figure 27: Output

```
Test Loss: 0.24828383326530457
Test Accuracy: 0.9095237851142883
```

7 Code

7.1 Softmax Regression

softMax.py implements softmax regression, a technique for multi-class classification tasks, using TensorFlow's Keras API. It comprises two functions: create-softmax-model and train-softmax-model. The former constructs a softmax regression model with a single Dense layer, which outputs class probabilities for input images. The latter function trains the model using the specified training and test data, employing the Adam optimizer and sparse categorical cross-entropy loss. The training process is visualized through a plot showcasing the model's training history (as we saw in figure 1), including loss and accuracy metrics over epochs.

Figure 28: Creation of the Softmax model

```
def create_softmax_model(input_shape, num_classes):
    """...

    model = models.Sequential([
        layers.Dense(num_classes, activation='softmax', input_shape=input_shape)
    ])

    return model
```

Figure 29: Training of the Softmax model

```
def train_softmax_model(X_train, y_train, X_test, y_test, num_classes):
    """...

    input_shape = X_train.shape[1:]

    # Create the model
    model = create_softmax_model(input_shape, num_classes)

    # Compile the model
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    epochs = 20
    # Train the model
    model_history = model.fit(X_train, y_train,
                              epochs= epochs,
                              batch_size=32,
                              validation_data=(X_test, y_test))

    # Display the training history
    pd.DataFrame(model_history.history).plot(figsize=(8, 5))
    plt.title('Soft max Training History, epochs: ' + str(epochs))
    plt.savefig("imgFolder/softMax_fig")
    plt.show()
    return model
```

7.2 Simple Neural Network Model

SimpleNeuralNetwork.py code implements a Simple Neural Network using TensorFlow's Keras API for image classification tasks. It consists of three functions: split-validation, create-neural-network and train-neural-network. The first split the data into, train, test and validation. The second, constructs a simple neural network model with multiple dense layers, along with batch normalization and dropout for regularization. The last function prepares the training and test datasets, compiles and trains the simple neural network model, and visualizes the training history, showcasing metrics evolution over epochs (as we saw in figure 10). Early stopping is used to prevent overfitting by halting training if validation loss does not improve for three consecutive epochs.

`split-validation:`

Splits the training data into training and validation sets. The function takes two parameters: X-train, which represents the features (input data) of the training set, and y-train, which represents the corresponding labels (output data) of the training set. It split the training data into training and validation value of size 10%. It return the x-train, y-train, x-validation, y-validation after splitting it.

`create-neural-network:`

Figure 30: Creation of simple neural network model

```
def create_neural_network(num_classes): # create_neural_network = 10
    model_input = layers.Input(shape=[28 ,28])
    x = layers.Flatten()(model_input)  # Flatten input
    x = layers.Dense(300, activation="relu")(x)  # 1st hidden layer
    x = layers.Dense(128, activation="relu")(x)   # 2nd hidden layer
    model_output = layers.Dense(num_classes, activation="softmax")(x)  # Output layer

    model = models.Model(inputs=model_input, outputs=model_output)
    return model
```

Flatten Layer: This layer flattens the 2D feature maps into a 1D vector, preparing the data for input into a fully connected neural network.

Dense Layers: Fully connected layers that perform classification based on the learned features. The first dense layer has 300 units with relu activation, enabling the network to learn complex patterns in the flattened feature vectors. The second dense layer has 128 units with relu activation, enabling the network to learn complex patterns in the flattened feature vectors. The final dense layer has units equal to the number of classes in the dataset, with softmax activation, which outputs probabilities for each class, indicating the likelihood of the input image belonging to each class.

`train-neural-network:`

Figure 31: Creation of simple neural network model

```python
def train_neural_network(model, X_train, y_train, X_test, y_test, x_validation, y_validation):
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    print(model.summary())
    X_train_shaped = X_train.values.reshape(-1, 28, 28)
    X_validation_shaped = x_validation.values.reshape(-1, 28, 28)
    X_test_shaped = X_test.values.reshape(-1, 28, 28)

    early_stopping = callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

    epochs=30
    model_history = model.fit(X_train_shaped,
                              y_train,
                              epochs=epochs,
                              validation_data=(X_validation_shaped, y_validation),
                              callbacks=[early_stopping])

    stop = early_stopping.stopped_epoch
    if stop == 0:
        stop = epochs
    pd.DataFrame(model_history.history).plot(figsize=(8, 5))
    plt.title('Simple Neural Network Training History \n epoch: ' + str(stop))
    plt.gca().set_ylim(0, 1)
    plt.savefig("imgFolder/simpleNeuralNetwork_fig")
    plt.show()

    model.evaluate(X_test_shaped, y_test)
```

This function prepares the training and test datasets by reshaping them into the format required for Simple neural network input, constructs the simple neural network model by calling the ‘create-neural-network‘ function, compiles the model and trains the model using the provided training data. We used early stopping to halt training if the validation loss does not improve for three consecutive epochs, trying to avoid overfitting. The training process’s history is plotted, showing metrics evolution over epochs.

7.3 CNN Model

CNN.py code implements a Convolutional Neural Network (CNN) using TensorFlow's Keras API for image classification tasks. It consists of two functions: create-cnn-model and train-cnn-model. The former constructs a CNN model with multiple convolutional and dense layers, along with batch normalization and dropout for regularization. The latter function prepares the training and test datasets, compiles and trains the CNN model, and visualizes the training history, showcasing metrics evolution over epochs (as we saw in figure 14). Early stopping is used to prevent overfitting by halting training if validation loss does not improve for three consecutive epochs.

`create-cnn-model:`

Figure 32: Creation of the CNN model

```
def create_cnn_model(input_shape, num_classes):

    """...
    activation = 'relu'
    dropout_rate = 0.3

    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation=activation, input_shape=input_shape),  # 32 filters of size 3x3
        BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=activation),
        BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=activation),
        BatchNormalization(),
        layers.Flatten(),
        layers.Dense(64, activation=activation),
        Dropout(dropout_rate),
        layers.Dense(num_classes, activation='softmax')
    ])

    return model
```

Convolutional Layers: These layers apply a convolution operation to the input image, extracting various features through filters. The activation function used is ReLU, which introduces non-linearity into the model, enabling it to learn complex patterns.

Batch Normalization: This technique is used to improve the training stability and speed by normalizing the inputs of each layer.

MaxPooling Layers: Max pooling reduces the spatial dimensions of the feature maps, effectively downsampling the input. It retains the most important features while reducing computational complexity and preventing overfitting.

Flatten Layer: This layer flattens the 2D feature maps into a 1D vector, preparing the data for input into a fully connected neural network.

Dense Layers: Fully connected layers that perform classification based on the learned features. The first dense layer has 64 units with ReLU activation, enabling the network to learn complex patterns in the flattened feature vectors. The final dense layer has units equal to the number of classes in the dataset, with softmax activation, which outputs probabilities for each class, indicating the likelihood of the input image belonging to each class.

Dropout: This layer was added to prevent overfitting. It works by randomly setting a fraction of input units to zero during training, which helps to prevent the network from relying too much on specific neurons and encourages it to learn more robust features. We've defined a Dropout layer with a dropout rate of 0.2. This means during training, 20% of the input units to the Dropout layer will be randomly set to zero.

`train-cnn-model:`

This function prepares the training and test datasets by reshaping them into the format required for CNN

input, constructs the CNN model by calling the ‘create-cnn-model‘ function, compiles the model and trains the model using the provided training data. We used early stopping to halt training if the validation loss does not improve for three consecutive epochs, trying to avoid overfitting. The training process’s history is plotted, showing metrics evolution over epochs.

Figure 33: Training of the CNN model

```python
def train_cnn_model(X_train, y_train, X_test, y_test, num_classes):
    """...
    # Reshape the input data for CNN
    X_train = np.array(X_train).reshape(-1, 28, 28, 1)
    X_test = np.array(X_test).reshape(-1, 28, 28, 1)

    # Create the model
    model = create_cnn_model((28, 28, 1), num_classes)  # grayscale image of size 28x28 pixels with one channel.

    # Compile the model
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # Apply early stopping
    early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

    # Train the model
    epochs = 20
    model_history = model.fit(X_train, y_train,
                              epochs=epochs,
                              batch_size=32,
                              validation_data=(X_test, y_test),
                              callbacks=[early_stopping])

    # Get the epoch where the fit stopped in
    stop = early_stopping.stopped_epoch
    if stop == 0:
        stop = epochs

    # Display the training history
    pd.DataFrame(model_history.history).plot(figsize=(8, 5))
    plt.title('CNN Training History \n epoch: ' + str(stop))
    plt.savefig("imgFolder/CNN_fig")
    plt.show()

    return model
```

8 Conclusion

In this study, various machine learning models were evaluated for the classification of fashion items using the Fashion-MNIST dataset. Comparison of traditional algorithms like Logistic Regression and Multi-Layer Perceptron with advanced techniques such as Softmax Regression, Convolutional Neural Networks (CNNs), and Simple Neural Networks revealed that the CNN architecture consistently outperformed others in accuracy and loss reduction. Through systematic exploration of hyperparameters, the optimized CNN model demonstrated superior performance, highlighting the efficacy of deep learning techniques for image classification tasks. These findings contribute valuable insights for applications in e-commerce, retail, and computer vision domains.