Deep Learning and Natural Language Processing Project

Moria Grohar - 323082024
Shirel Zecharia - 211551072

March 3, 2024

Contents

1 Abstract

The report presents a study on the implementation of a neural network model for the classification of fashion items using the Fashion MNIST dataset.
The performance of the neural network approach is compared with traditional machine learning algorithms such as softmax regression and multi-layer perceptron (MLP).
The report discusses the project methodology, the experimental results and the 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 serves as a common choice for the final activation layer in multi-class neural networks, as detailed 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

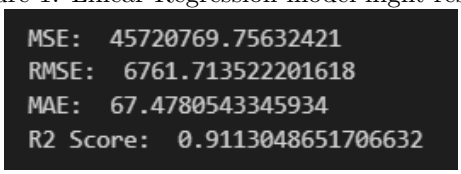
Model Selection and Evaluation:
The project explored the performance of various models for predicting flight ticket prices in a dataset containing information on bookings between major Indian cities. The target variable was the "Price" column, indicating a continuous value. Therefore, the problem was 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 Linear Regression:

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

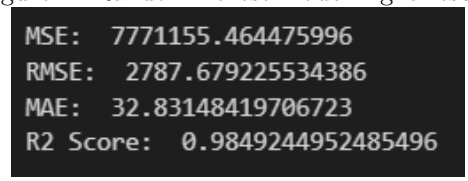
Figure 1: Linear Regression model flight result:



4.1.2 Random Forest:

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

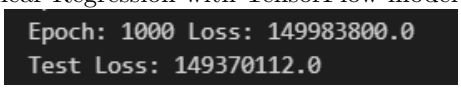
Figure 2: Random Forest model flight result:



4.1.3 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 3: Linear Regression with TensorFlow model flight result:



4.1.4 Neural Network Exploration:

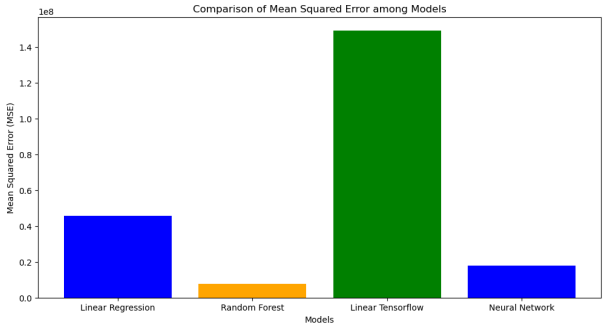
A neural network model with 2 hidden layers and 80 hidden units was constructed using TensorFlow. Hyperparameter tuning involved adjusting the learning rate and number of epochs, ultimately leading to the configuration with the lowest observed loss.

Figure 4: neural network model flight result:

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

4.1.5 Comparison and Conclusion:

Figure 5: 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.

The decision to change the dataset stemmed from its perceived lack of complexity, which prevented us from improving the results with advanced models such as RNN, CNN or attention mechanism to achieve optimal accuracy and loss function improvement:

4.2 Second Attempts with Fashion-MNIST 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:

In an attempt to optimize the performance of the model we tried different number of epochs, optimizers and loss functions. We were able to get better accuracy, from 79% we reached 85%).

4.2.2 Simple Neural Network:

In an effort to optimize the performance of the model, various hyperparameters were explored such as, different size of the kernel, numbers of layers, number of epochs, optimizers, loss functions, etc. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy. We were able to reach from accuracy of 87% to (89%) and loss function from 36% to 31%.

4.2.3 CNN:

In an effort to optimize the performance of the model, we explored hyperparameters such as different size of the kernel, number of epochs, optimizers, loss functions. Also we normalized the data between convolutional layers. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy. We were able to reach from 87% accuracy to (92%) accuracy, and from 33% loss to 24% loss.

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:** 92%
2. **Loss Function:** 0.24%

6 Simulation Results

6.1 Softmax Regression Model

Figure 6: Training history of the data

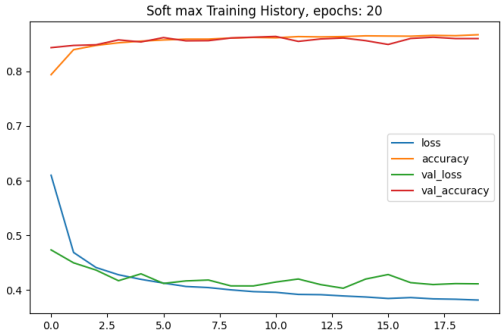


Figure 7: Output

```
Test Loss: 0.41161519289016724
Test Accuracy: 0.8596190214157104
```

6.2 Simple Neural Network Model

Figure 8: Training history of the data

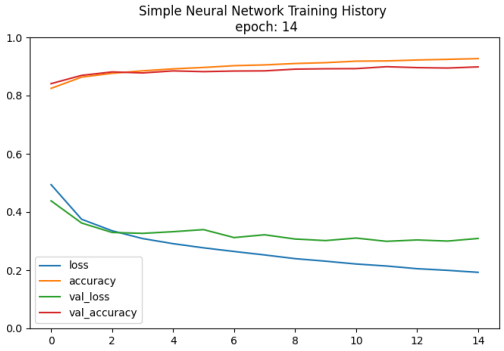


Figure 9: Output

```
Test Loss: 0.2876456379890442
Test Accuracy: 0.899571418762207
```

6.3 Convolutional Neural Network Model

Figure 10: Training history of the data

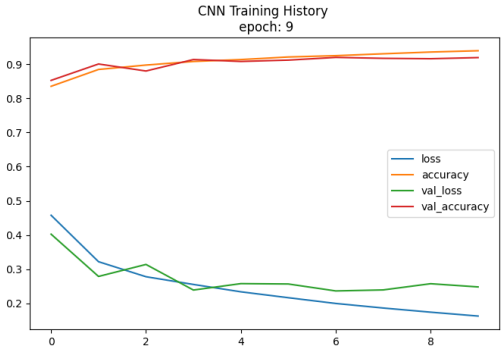


Figure 11: Output

```
Test Loss: 0.2364427149295807
Test Accuracy: 0.9198095202445984
```

6.4 Comparing Between Models

Figure 12: Comparing accuracy score between models:

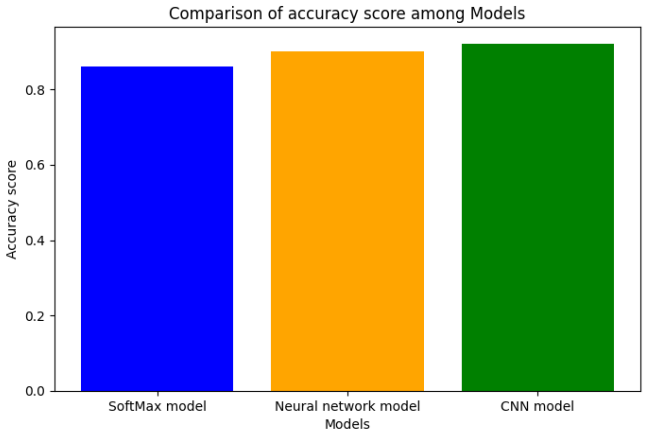
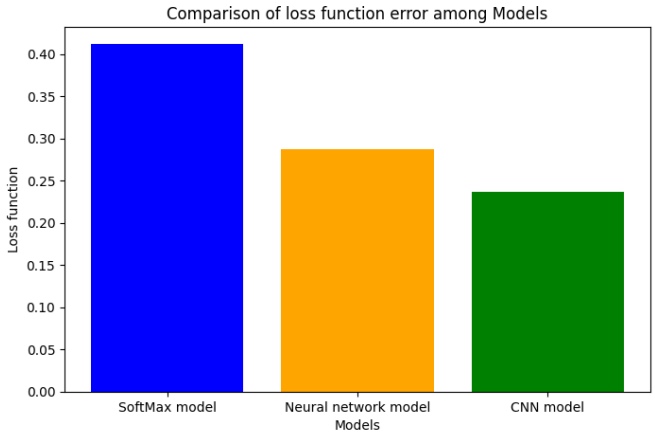


Figure 13: Comparing loss function between models:



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

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 (figure 7), including loss and accuracy metrics over epochs.

Figure 14: 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 15: 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 two functions: create-neural-network and train-neural-network. The former, constructs a simple neural network model with a dense layer. The latter 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 (figure 9). Early stopping is used to prevent overfitting by halting training if validation loss does not improve for three consecutive epochs.

`create-neural-network:`

Figure 16: 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
    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 Layer: Fully connected layers that perform classification based on the learned features. The first dense layer (hidden layer) has 300 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 17: Creation of simple neural network model

```
def train_neural_network(X_train, y_train, x_validation, y_validation, num_classes):
    model = create_neural_network(num_classes)  # grayscale image of size 28x28 pixels with one channel.

    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)

    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()

    return model
```

The function is responsible for preparing the training and test datasets, reshaping them into the required format for input into the Simple Neural Network. It then constructs the Simple Neural Network model by invoking the create-neural-network function, compiles the model, and proceeds to train it using the provided training data. Early stopping is employed during training to prevent overfitting, halting the process if the validation loss fails to improve for three consecutive epochs. The evolution of metrics over epochs during the training process is visualized through plotting the training history.

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 (figure 11). Early stopping is used to prevent overfitting by halting training if validation loss does not improve for three consecutive epochs.

`create-cnn-model:`

Figure 18: 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 have defined a dropout layer with a dropout rate of 0.3. This means during training, 30% 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 19: 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
```

7.4 Main

main.py includes the main function, which implements all the function that run the models an create comparison. `main`

Figure 20: Main function part 1

```python
from modelUtils import *
from fashionMNIST import *
from softMax import *
from SimpleNeuralNetwork import *
from CNN import *


def main():
    df_shuffled = get_data()

    # For SoftMax, CNN
    X_train1, y_train1, X_test1, y_test1 = split_data1(df_shuffled)

    # For RNN
    X_train0, y_train0, X_test0, y_test0, X_validation0, y_validation0 = split_data0(df_shuffled)

    # Number of unique classes
    num_classes = len(np.unique(y_train1))

    # For showcasing the results of all models
    loss_array = []
    model_array = []
    accuracy_array = []

    # Perform training for softmax model
    print("SoftMax model:")
    softmax_model = train_softmax_model(X_train1, y_train1, X_test1, y_test1, num_classes)
    (loss, accuracy) = evaluate_model(softmax_model, X_test1, y_test1)
    loss_array.append(loss)
    model_array.append("SoftMax model")
    accuracy_array.append(accuracy)
```

Figure 21: Main function part 2

```
    # Perform training for Neural network model
    print("Neural network model:")
    neural_network_model = train_neural_network(X_train0, y_train0, X_validation0, y_validation0, num_classes)
    (loss, accuracy) = evaluate_model(neural_network_model, X_test0.values.reshape(-1, 28, 28, 1), y_test0)
    loss_array.append(loss)
    model_array.append("Neural network model")
    accuracy_array.append(accuracy)

    # Perform training for CNN model
    print("CNN model:")
    cnn_model = train_cnn_model(X_train1, y_train1, X_test1, y_test1, num_classes)
    (loss, accuracy) = evaluate_model(cnn_model, X_test1.values.reshape(-1, 28, 28, 1), y_test1)
    loss_array.append(loss)
    model_array.append("CNN model")
    accuracy_array.append(accuracy)

    plt.figure(figsize=(8, 5))
    plt.bar(np.array(model_array), np.array(loss_array), color=['blue', 'orange', 'green'])
    plt.xlabel('Models')
    plt.ylabel('Loss function')
    plt.title('Comparison of loss function error among Models')
    plt.savefig("imgFolder/lossFuncComparison")
    plt.show()

    plt.figure(figsize=(8, 5))
    plt.bar(np.array(model_array), np.array(accuracy_array), color=['blue', 'orange', 'green'])
    plt.xlabel('Models')
    plt.ylabel('Accuracy score')
    plt.title('Comparison of accuracy score among Models')
    plt.savefig("imgFolder/accurateFuncComparison")
    plt.show()


main()
```

7.5 Fashion Mnist

fashionMNIST.py includes 3 functions that handles the data: get-data, split-data0, split-data1. `get-data`

Figure 22: Get data function

```
from __future__ import absolute_import, division, print_function, unicode_literals
import importlib
import pandas as pd
import numpy as np
import utils
from sklearn.model_selection import train_test_split

import tensorflow as tf
layers, models = tf.keras.layers, tf.keras.models

importlib.reload(utils)

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


def get_data():
    fashion_mnist = tf.keras.datasets.fashion_mnist

    (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
    # Combine train and test images and labels
    images = np.concatenate((train_images, test_images))
    labels = np.concatenate((train_labels, test_labels))
    # Flatten images to 1D arrays
    images = images.reshape(-1, 28 * 28)
    # Convert images and labels to DataFrame
    data = {'image_' + str(i): images[:, i] for i in range(images.shape[1])}
    data['label'] = labels
    df = pd.DataFrame(data)
    df_shuffled = df.sample(frac=1).reset_index(drop=True)
    return df_shuffled
```

`split-data0`

Splits the training data into training, testing and validation sets. The function takes one parameters: df-shuffled which is a dataframe. It split the df-shuffled into training and testing value of size 10%. Then it splits the training data into training and validation value of size 10%. It return the x-train, y-train, x-test, y-test, x-validation, y-validation after splitting it.

Figure 23: Split data with validation

```
def split_data0(df_shuffled):
    X = df_shuffled.drop('label', axis=1)
    y = df_shuffled['label']
    X_train_, X_test, y_train_, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
    X_train, X_validation, y_train, y_validation = train_test_split(X_train_, y_train_, test_size=0.1, random_state=42)

    # To transform the images to be in scale from 0-1
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    X_validation = X_validation / 255.0
    return X_train, y_train, X_test, y_test, X_validation, y_validation
```

`split-data1`

Splits the training data into training and testing sets. The function takes one parameters: df-shuffled which is a dataframe. It split the df-shuffled into training and test value of size 15%. It return the x-train, y-train, x-test, y-test after splitting it.

Figure 24: Split data without validation

```
def split_data1(df_shuffled):
    X = df_shuffled.drop('label', axis=1)
    y = df_shuffled['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

    # To transform the images to be in scale from 0-1
    X_train = X_train / 255.0
    X_test = X_test / 255.0

    return X_train, y_train, X_test, y_test
```

8 Conclusion

In this study, various machine learning models were evaluated for classifying 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.

9 Code Appendix

9.1 Figure 14: 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
```

9.2 Figure 15: 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('Softmax Training History, epochs: ' + str(epochs))
    plt.savefig("imgFolder/softMax_fig")
    plt.show()

    return model
```

9.3 Figure 16: 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
    model_output = layers.Dense(num_classes, activation="softmax")(x)  # Output layer

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

9.4 Figure 17: Creation of simple neural network model

```
    def train_neural_network(X_train, y_train, x_validation, y_validation, num_classes):
        # grayscale image of size 28x28 pixels with one channel.
        model = create_neural_network(num_classes)

        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)

        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()

        return model
```

9.5 Figure 18: 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
```

9.6 Figure 19: Training of the CNN model

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

    # 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
```

9.7 Figure 20+21: Main function part 1+2

```
def main():
    df_shuffled = get_data()

    # For SoftMax, CNN
    X_train1, y_train1, X_test1, y_test1 = split_data1(df_shuffled)

    # For RNN
    X_train0, y_train0, X_test0, y_test0, X_validation0, y_validation0 = \
        split_data0(df_shuffled)
    # Number of unique classes
    num_classes = len(np.unique(y_train1))

    # For showcasing the results of all models
    loss_array = []
    model_array = []
    accuracy_array = []

    # Perform training for softmax model
    print("SoftMax model:")
    softmax_model = train_softmax_model(X_train1, y_train1, X_test1, y_test1, num_classes)
    (loss, accuracy) = evaluate_model(softmax_model, X_test1, y_test1)
    loss_array.append(loss)
    model_array.append("SoftMax model")
    accuracy_array.append(accuracy)

    # Perform training for Neural network model
    print("Neural network model:")
    neural_network_model = train_neural_network(X_train0, \
        y_train0, X_validation0, y_validation0, num_classes)
    (loss, accuracy) = evaluate_model(neural_network_model, \
        X_test0.values.reshape(-1, 28, 28, 1), y_test0)
    loss_array.append(loss)
    model_array.append("Neural network model")
    accuracy_array.append(accuracy)

    # Perform training for CNN model
    print("CNN model:")
    cnn_model = train_cnn_model(X_train1, y_train1, \
        X_test1, y_test1, num_classes)
    (loss, accuracy) = evaluate_model(cnn_model, \
        X_test1.values.reshape(-1, 28, 28, 1), y_test1)
    loss_array.append(loss)
    model_array.append("CNN model")
    accuracy_array.append(accuracy)

    plt.figure(figsize=(8, 5))
    plt.bar(np.array(model_array), np.array(loss_array), \
        color=['blue', 'orange', 'green'])
    plt.xlabel('Models')
    plt.ylabel('Loss function')
    plt.title('Comparison of loss function error among Models')
    plt.savefig("imgFolder/lossFuncComparison")
    plt.show()
```

```
    plt.figure(figsize=(8, 5))
    plt.bar(np.array(model_array), np.array(accuracy_array), \
        color=['blue', 'orange', 'green'])
    plt.xlabel('Models')
    plt.ylabel('Accuracy score')
    plt.title('Comparison of accuracy score among Models')
    plt.savefig("imgFolder/accurateFuncComparison")
    plt.show()

main()
```

9.8 Figure 22: Get data function

```
from __future__ import absolute_import, division, print_function, unicode_literals
import importlib
import pandas as pd
import numpy as np
import utils
from sklearn.model_selection import train_test_split

import tensorflow as tf
layers, models = tf.keras.layers, tf.keras.models

importlib.reload(utils)

class_names = ['T-shirt/top', 'Trouser', 'Pullover',
'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag',
    'Ankle boot']


def get_data():
    fashion_mnist = tf.keras.datasets.fashion_mnist

    (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
    # Combine train and test images and labels
    images = np.concatenate((train_images, test_images))
    labels = np.concatenate((train_labels, test_labels))
    # Flatten images to 1D arrays
    images = images.reshape(-1, 28 * 28)
    # Convert images and labels to DataFrame
    data = {'image_' + str(i): images[:, i] for i in range(images.shape[1])}
    data['label'] = labels
    df = pd.DataFrame(data)
    df_shuffled = df.sample(frac=1).reset_index(drop=True)
    return df_shuffled
```

9.9 Figure 23: Split data with validation

```
def split_data0(df_shuffled):
    X = df_shuffled.drop('label', axis=1)
```

```
    y = df_shuffled['label']
    X_train_, X_test, y_train_, y_test = train_test_split(X, y, \
        test_size=0.1, random_state=42)
    X_train, X_validation, y_train, y_validation = train_test_split(X_train_, y_train_,\
        test_size=0.1, random_state=42)

    # To transform the images to be in scale from 0-1
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    X_validation = X_validation / 255.0
    return X_train, y_train, X_test, y_test, X_validation, y_validation
```

9.10 Figure 24: Split data without validation

```
def split_data1(df_shuffled):
    X = df_shuffled.drop('label', axis=1)
    y = df_shuffled['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, \
        random_state=42)

    # To transform the images to be in scale from 0-1
    X_train = X_train / 255.0
    X_test = X_test / 255.0

    return X_train, y_train, X_test, y_test
```