Implementing RNN and CNN to Classify Different Articles of Clothing

By: Ryan Dang

Introduction:

Task and Dataset: The dataset being used is the Fashion-MNIST dataset which is composed of 28x28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category. The categories are T-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, or angle boot. The products themselves are from Zalando (a German online retail company), and the training set has 60,000 images and the test set has 10,000 images. Using this dataset, the goal is to create two deep learning models that are able to classify the type of the article of clothing given an image of it.

Preprocessing: Initially, we want to reshape the data to include a single color channel (grayscale) and match the expected input shape of your neural network. Thus we convert each 28 x 28 image of the dataset into a matrix of size 28 x 28 x 1. Additionally, we normalize/scale each image by converting each pixel of the image to be of type float32, and then dividing the pixel value by 255, rescaling the pixel values to be in the range of [0,1]. Furthermore, we will convert all class labels into one-hot encoding vectors. The one-hot encoding vector will be a row vector, and for each image, it will have a dimension of 1 x 10. The vector consists of all zeros, except for the class it represents, and it would be represented by the number one. For instance, a boot image would have a class label of 9, meaning for all boot images the one hot encoding vector would be [0 0 0 0 0 0 0 0 1 0]. This is done in order to transform categorical data to numerical data, and will allow us to apply different algorithms and functions over it. Lastly, we will partition our dataset into a training set and a validation set (the test set was already given to us). Specifically, 80% of the data is for training, and the rest for validation.

Implementation/Architectures of the Deep Learning Systems:

Convolutional Neural Network:

Architecture: The architecture of a Convolutional Neural Network (CNN) is characterized by its hierarchical structure designed for processing and learning spatial patterns in data, particularly well-suited for image-related tasks. The CNN is composed of a multitude of layers chained together to systematically extract information from the input data, enabling the network to capture intricate patterns and features. The main types of layers used in a CNN are the input layer, convolutional layer, pooling layer, activation layer, and the fully connected layer The network begins with an input layer representing the raw data (such as 32 x 32 x 1 dimension image in this case). Next, we add the convolutional layers, the fundamental building blocks of the network. They apply filters to capture local patterns and features through convolution operations. After convolution, we get a feature map. Following this layer, is the activation layer. By using activation functions like Rectified Linear Units (ReLU) or Sigmoid, we are able to add nonlinearity to the network. Pooling layers follow, reducing spatial dimensions of the feature map by down-sampling through operations like max or average pooling. This process helps generalize features extracted by convolutional filters and helps the network recognize features independent of their location in the image. After the pooling layer, the CNN would then start with the convolutional layer again, and the cycle would restart. Once the cycle of layers is finished, the output is then flattened into a one-dimensional vector and passed through fully connected (dense) layers, which capture high-level representations and facilitate classification or regression tasks. The Fully Connected layer consists of the weights and biases along with the neurons where each neuron is connected to every neuron in the previous and the next layers. The final layer is the softmax activation layer, which normalizes the network's output to a probability distribution over predicted output classes.

Implementation:

We first load all the necessary packages and libraries to get the dataset and the functions needed to create the CNN (Tensorflow and Keras). Once the data is loaded in from the package, we preprocess the data as described in the introduction. Next, we initiate a Sequential model which holds all the necessary layers for the CNN. The first layer we add is the Input layer in order to instantiate a tensor representing the input of my model (the images). Then, we add the convolutional layer that contains 32 3x3 filters, does zero-padding to the input, and has strides of 1. Next we add the activation layer, and we use the Leaky ReLU activation function with an alpha/slope of 0.1. Following the activation layer, it is a pooling layer where we do Max Pooling with zero-padding, and the pool size is 2x2. After the pooling layer, we add a dropout layer with a drop rate of .25. The next eight layers will follow the same pattern as the first four layers (convolutional layer, Leaky ReLU layer, Dropout layer, and Max Pooling layer)**.** After the third Max Pooling layer, we have a Flatten layer that flattens the input into a one-dimensional array (i.e if the input is 32x32, it will flatten into an array with 1024 elements). Following the Flatten layer, we add a Dense (Fully Connected) layer which consolidates the input into 128 units (neurons) where each neuron in this layer is connected to every neuron in the previous layer. After the Dense layer, we have another Leaky ReLU layer with an alpha/slope of 0.1**,** and then we implement another Dropout layer with a drop rate of 0.25. The last layer is a Dense layer which consolidates the input into 10 units (neurons), and performs a softmax over the 10 values (converts the 10 values into values of a probability distribution).

After the model is created, we compile it using the categorical cross entropy as the loss function and using the Adam algorithm as the optimizer. For the metrics that the model is being evaluated on is the accuracy of the predicted labels with the actual labels. Once the model is compiled, we train the model using the training set and the validation set with a batch size of 128 and an epoch size of 20. This means that every 128 samples, the internal model parameters are updated, and the model will work through the entire training dataset 20 times. To enhance training, a learning rate scheduler is implemented to dynamically adjust the learning rate, and early stopping is employed to halt training when the validation loss ceases to improve. After the model is trained, we test the model with our test dataset.

Recurrent Neural Network:

Architecture:

The architecture of a Recurrent Neural Network (RNN) is designed to process sequential data by maintaining hidden states (or layers) that capture information from previous time steps. The key feature of RNNs is their ability to maintain a form of memory, allowing them to capture temporal dependencies in the input data. This is due to the hidden states maintaining information about previous inputs of the sequence. At each time step, the RNN receives an input and in conjunction with the hidden state at the previous time tape, the current output is produced and the hidden state is updated, serving as a summary of the information seen so far. The architecture is recurrent in nature, meaning it allows information to persist across different time steps and to be updated over time (the recurrent computation occurs in the hidden state). However, traditional RNNs suffer from the vanishing gradient problem, which limits their ability to capture long-term dependencies.

The type of RNN that is being used is the Gated Recurrent Unit (GRU), a variation of the Long Short Term Memory (LSTM). The GRU exhibits a specialized design for effectively modeling sequential data. Unlike traditional RNNs, GRUs incorporate gating mechanisms that help address the vanishing gradient problem, allowing them to capture long-range dependencies more efficiently. The core structure of a GRU involves candidate (hidden) states, update gates, and reset gates, helping to control the flow of information through the network. The reset gate decides what information to discard from the previous hidden state, aiding the network in adapting to changing patterns in sequential data. The candidate state is computed using the tanh activation function, and it is composed of the input and the previous hidden state that has been "reseted" by the reset gate (this controls how much influence the previous hidden state can have on the candidate state). The candidate state determines how much new information be presented in the new hidden state. The update gate regulates how much of the previous hidden state to retain and how much of the new candidate state to include in the new hidden state, offering a balance between memory retention and the integration of new information.

Implementation:

We first load all the necessary packages and libraries to get the dataset and the functions needed to create the RNN (Tensorflow and Keras). Once the data is loaded in from the package, we preprocess the data as described in the introduction. Next, we initiate a Sequential model which holds all the necessary layers for the RNN. The first layer we add is the Input layer in order to instantiate a tensor representing the input of my model (the images). Next, the RNN model is constructed with a Gated Recurrent Unit (GRU) layer where tanh is the activation function, and sigmoid is the activation function used for the recurrent step, consolidating the input into 128 units (neurons). After the GRU layer, we have a Flatten layer that flattens the input into a one-dimensional array. Next is the Dense (Fully Connected) layer with ReLU activation, consolidating the input into 128 units (neurons), followed by a Leaky ReLU layer with an alpha/slope of 0.1. It is then followed by a dropout layer with a drop rate of 25% which is used to prevent overfitting. Lastly, the final layer is a Dense layer which consolidates the input into 10 units (neurons), and performs a softmax activation over the 10 values for class predictions.

Similar to the CNN model, the RNN model is compiled using the Adam optimizer and sparse categorical cross entropy loss. Once the model is compiled, we start training the model using the training set and validation set with a batch size of 128 and an epoch size of 20. To enhance training, a learning rate scheduler is implemented to dynamically adjust the learning rate, and early stopping is employed to halt training when the validation loss ceases to improve. After the model is trained, we test the model with our test dataset.

Training Details:

Training for CNN and RNN:

The training for CNN and RNN were both the same. Once the model is created, we compile it using the Categorical Cross Entropy as the loss function and using the Adam algorithm as the optimizer, and the metrics that the model is being evaluated on is the accuracy of the predicted labels.

Categorical Cross Entropy measures the disparity between the predicted probability distribution of classes and the actual distribution of class labels. As the model iteratively adjusts its parameters during training, the goal is to minimize this cross-entropy loss. When the predicted probabilities align with the true distribution, the loss approaches zero. Conversely, if the predicted probabilities diverge from the true distribution, the loss increases.

The Adam algorithm (Adaptive Moment Estimation) is an adaptive learning rate method that has aspects of RMSprop and Momentum to dynamically adjust learning rates for the model parameters during training. The Adam algorithm maintains two moving averages (referred to as moments), one for gradients and one squared gradients. The first moment takes the average of past gradients, helping to capture the overall direction in which the parameters are changing. Moreover, it introduces a smoothing effect, mitigating the impact of frequent changes in the gradient's direction and smoothing them out. The second moment takes the root mean square value of past gradients, and it helps track the magnitude of the gradients. Additionally, it helps normalize the learning rates for each parameter, ensuring that parameters with steep gradients and those with shallow gradients are appropriately adjusted during the optimization process. Using these two moments, Adam is able to adaptively adjust learning rates for the model parameters based on both the direction and magnitude of historical gradients.

In the training, we have two different callbacks that would perform various actions at various stages of training. The two callbacks are a custom learning rate scheduler function and an EarlyStopping callback. The learning rate scheduler function makes the learning rate unchanged for the first 8 epochs and subsequently exponentially decays it. This is done to dynamically adust

the learning rate and to achieve better convergence. The EarlyStopping which halts training if the validation loss does not improve for two consecutive epochs in order to prevent overfitting and optimize training efficiency. Overall, these callbacks help enhance the efficiency and adaptability of the neural network optimization process.

Once the model is compiled, we train the model using the training set (80% of data) and the validation set (20% of the data) with a batch size of 128 and an epoch size of 20.This means that for every 128 samples of the training set, the model parameters are updated before the model continues through the training set. Furthermore, with an epoch size of 20, the model will process the entire training dataset 20 times (unless the EarlyStopping callback stops it early due to validation loss not improving). After the model is trained, we test the model with our test dataset.

Challenges/Obstacles:

Library Dependencies and Compatibility Issues:

The main package/library used to create the neural networks was the TensorFlow library. It had all the necessary functions and predefined objects that allowed creating a neural network to be easy. However, downloading TensorFlow became a more intricate process than initially envisioned, chiefly owing to the intricacies inherent in the realm of library dependencies and compatibility issues with my MacBook Pro.

It became evident that TensorFlow, being a sophisticated framework, is dependent upon an extensive array of external libraries for its seamless functionality. The challenge materialized as these dependencies, spanning from foundational numerical libraries to specialized tools, posed compatibility issues. Moreover, I had to pay meticulous attention to versioning of the different dependent libraries, as well as the versions of the certain libraries that are dependent on TensorFlow (some libraries had to be downgraded to older versions in order to work alongside TensorFlow). The solution to library dependency is relatively simple, but tedious. Using forums and documentation, I was able to ensure that I had the correct versions of the required libraries installed and which version of TensorFlow was needed to work alongside other packages. Moreover, I used a package manager or virtual environment to manage dependencies, making it easier to see which libraries I need to download.

After downloading TensorFlow and handling all the different dependencies, I had to solve the compatibility issue with my MacBook Pro. When trying to import anything from TensorFlow, an error message of “Illegal Hardware Instruction” appeared on the terminal, preventing the execution of the rest of the code. After research, the problem is only specific to MacBook Pro’s with a M1 chip, and it is supposed to be a bug. To fix this issue, I had to create a new environment using Minifroge, and use the environment when executing the code instead of the base environment VS Code defaults too.

Time-Efficiency:

To get the best accuracy possible, a multitude of testing and experimentation must be done on the different layers to see which layers (and the specifics of the layers) provide the best model possible for classifying the different articles of clothing. However, it is quite time consuming to test every different combination of layers or to tweak a layer in order to see if it improves the models. In order to solve this problem, I was able to shorten the training time, and I focused on the layers that impacted the accuracy the most. First, by training the dataset through five epochs instead of twenty epochs, I significantly reduced the training time, allowing me to test more models. Once the final model has been confirmed, I would then train the model through twenty epochs instead of five. Additionally, by focusing on the layers that impacted the accuracy the most, it would reduce the amount of variables that I would alter in the model, making the amount of different models to be tested to be reduced. By implementing these two solutions, I was able to be more efficient with my time, and I was able to find the final model in a less time-consuming manner.

Overfitting

A problem that I encountered when testing my models was that the loss value was considerably high (around .4). Additionally, when looking at the graphs measuring loss and accuracy, the training and validation curves for accuracy and loss were not in line with each other, showing signs of overfitting. To fix this issue, I implemented dropout layers throughout the model. The dropout layer randomly sets input units to 0 with a predefined frequency of rate at each step during training time to help prevent overfitting. When the dropout layers were added, this made loss go down significantly (around .2), and the training and validation curves for accuracy and loss were more in line with each other than previously, meaning the model is overfitting less, and the model's generalization capability became much better.

Results, Observations, and Conclusions:

Results and Observations for CNN:

From the CNN, we were able to get an accuracy of 0.919, and a loss of .243 for the test set. Specifically, the CNN incorrectly classified 814 articles of clothing, and correctly classified 9186 articles of clothing. The type of clothing that has been accurately predicted the least is a shirt, and conversely, the type of clothing that has been accurately predicted the most is trousers. Additionally, we see that trousers have the highest precision, recall, and F1-score, all with the score of 0.98, indicating the model's excellent ability to correctly identify instances of this class. Moreover, the bag also shows strong performance with a precision, recall and F1-score all being .99 as well. Shirts, however, have a lower recall of 0.70, suggesting that the model may struggle to identify instances of this class, and its overall F1-score is relatively lower at 0.78. Overall, the model appears to perform well, with high precision and recall for many classes. Additionally, based on the graphs measuring loss and accuracy respectively, we can see that the learning curves for loss and accuracy are not linear. The validation loss and validation accuracy are both in line with the training loss and training accuracy, with both the validation and training curves start to deviate from each other at the end (for both accuracy and loss).

Results and Observations for RNN:

From the RNN, we were able to get an accuracy of 0.897, and a loss of 0.296 for the test set. Specifically, the RNN incorrectly classified 1092 articles of clothing, and correctly classified 8908 articles of clothing. The type of clothing that has been accurately predicted the least is a shirt, and conversely, the type of clothing that has been accurately predicted the most is sneakers. Additionally, we see that sneakers have the highest precision, recall, and F1-score, with a score of 0.98, indicating the model's excellent ability to correctly identify instances of this clothing. Trousers also showed strong performance with a precision of 0.98, recall of 0.97, and an F1-score of 0.98. Shirts, however, have a lower recall of 0.71, suggesting that the model may struggle to identify instances of this class, and its overall F1-score is relatively lower at 0.74. Overall, the model appears to perform well, with high precision and recall for many classes. Additionally, based on the graph measuring loss and accuracy respectively, we can see that the

learning curves for loss and accuracy are not linear. The validation loss and validation accuracy are both in line with the training loss and training accuracy, and they are more in line than the validation and training curves produced by the CNN.

Conclusion:

Both the RNN and CNN had relatively high accuracy and low loss, meaning that both models are able to make correct predictions on the majority of the dataset, and even if our predictions are wrong, they are still close to the actual value. Additionally from both the graphs above, the models are not overfitting or underfitting from the training data, allowing both models to be used on more generalized data. From both the RNN and CNN, it seems that shirts have been misidentified the most, and sneakers were correctly identified the most. This means that sneakers are more distinguishable when compared to the other articles of clothing, and shirts have some features that are similar to other articles of clothing.

Even though the RNN and CNN both performed well, the CNN still performed slightly better than RNN. This is to be expected since the CNN's are more tailored to image classification problems. CNNs are designed to capture spatial hierarchies and local patterns in data, making them well-suited for image-related tasks. They use convolutional layers to learn hierarchical representations of visual features. On the other hand, RNN's are more effective in handling sequential and/or time-series data (i.e natural language processing). Images do not have a sequential nature like sentences, and using RNNs for image classification would not leverage the inherent spatial relationships in the data. Additionally, because the CNN is more geared towards the task, the time it took to train the CNN on the dataset took significantly less time than the time it took for the RNN to train the dataset. Overall, both the RNN and CNN models performed quite well on the classification task, and while RNN was not suited for this task, it still performed quite well.