**A person walking in a city

Description automatically generated< Fashion MNIST - Evaluating the Efficacy of CNN against Traditional ML Models>**

<CIND860 Capstone Project

By: Hasib Chowdhury (500439135)

Supervisor: Dr. Tamer Abdou >

**GitHub Link:** [**https://github.com/Hasib147/CIND860-Capstone-Project**](https://github.com/Hasib147/CIND860-Capstone-Project)

Table of Contents

[Revised abstract 3](#_Toc452457337)

[Introduction 4-5](#_Toc452457337)

[Literature Review 6-10](#_Toc452457337)

[Data Description 11-12](#_Toc452457337)

[Exploratory Data Analysis 13](#_Toc452457337)-14

CNN-simple [Modelling 15](#_Toc452457337)-18

Random Forest [Modelling 19-20](#_Toc452457337)

Effectiveness 21-29

Efficiency 30-32

Stability 33-36

Research Limitations……………………………………….…………………………………37

Conclusion……………………………………………………………………………………...38

References……………………………………….…………………………………………….39

# Revised Abstract (CIND860 Capstone Project)

For this project, I will use the Fashion MNIST dataset (<https://www.kaggle.com/datasets/zalando-research/fashionmnist>) from the Kaggle website to conduct the research to fulfill the “Advanced Data Analytics Project (CIND860)” requirements.

There are a total of 70,000 images for this dataset, which is split 60,000 for training and 10,000 for testing (comprising of 28x28 grayscale images). This exact dataset is also built-in in the keras library, which shows the exact images (shirts, pants, sneakers, etc.) in its pixel grayscale form as this is not available on Kaggle. The Kaggle dataset only has 2 .csv files for training and testing for the 784 pixels in the dataset (ranging from 1 to 255 in darkness of the image).

The theme that has been chosen for this project is the deep learning theme (specifically image classification on various Fashion attires) and the technique being used is Convolutional Neural Network (CNN) on the Fashion MNIST dataset.

Some of the research questions this project will go into detail is what models of the CNN is the most efficient to use. For example, Is CNN's performance practical when compared to traditional machine learning techniques such as Random Forest, SVM or XGBoost? As well as How the CNN's performance compared to other models like LeNet-5, VGGNet, ZFNet, or ResNet? *“Fashion businesses in general have used CNN on their e-commerce platforms to solve many problems such as clothes recognition, clothes search and recommendation. A core step for*

*all of these implementations is image classification. However, clothes classification is a challenge task as clothes have many properties, and the depth of clothes categorization is highly*

*complicated.” [1].*

One other thing I will investigate is if we are able to identify the significant features that can accurately predict the classification of the fashion attire dataset? Also the performance of different evaluation measures, such as Accuracy, Recall, and Precision vary? (within different layers of the dataset)

Additional research question: Just like the research paper being replicated, I will compare the simple CNN model with the 3 dropout CNN models that are used. I will see how they compare in terms of accuracy (with a smaller sample size of 70k rows, due to memory constraint) and if it matches with the table given in the article.

I will use Python as the main programming language. I will also look at which specific models of the CNN architecture are the most commonly used when evaluating the Fashion MNIST dataset in deep learning models. Throughout the fashion industry and also in Fashion e-commerce and in online retail such as Amazon and E-bay, the market has been growing in recent years and the “CNN model in particular has been shown greater efficiency in image c1assification” [3]. This is what this project will look into as the main technique that is going to be used throughout the project.

# Introduction

“The fashion market has changed dramatically over the last 30 years, resulting in an evolution in that industry. Understanding customer tastes and better-directing sales are the way to increase profit” [3]. “The rise of internet business lets people buy their clothes through websites, faster and easier. The introduction of methods to improve user’s experience when searching for items in these platforms is decisive” [3]. In platforms such as Amazon and E-bay, many consumers are buying their clothing and footwear online in recent years compared to the years prior where they were buying in-person at the store. The trend from retail clothes shopping in store to online shopping has drastically changed, mostly for the better good but there can be issues with online shopping such as fraud with credit cards, but most transactions are secure.

I have also used ChatGPT for this part of the project, when prompted “In 3-4 paragraphs describe Fashion MNIST specifically for Fashion Business industry if possible. And maybe include Amazon or E-bay as platforms for online shopping based on a literature review”. The ChatGPT-generated text indicated that Fashion MNIST dataset holds significant importance for the fashion business industry, serving as a cornerstone in the field of computer vision and machine learning. In addition, “Fashion MNIST is essentially a collection of grayscale images representing various fashion items, such as clothing, footwear, and accessories that we use in our daily lives. Each image is associated with a corresponding label, classifying the item it represents. This dataset plays a pivotal role in training and testing machine learning models, making it a valuable resource for fashion businesses seeking to leverage cutting-edge technologies.” [4]

In terms of how Amazon and E-bay play a role with Fashion MNIST in the business market. The AI states that “With the growth of e-commerce platforms like Amazon and eBay, the utilization of Fashion MNIST has gained particular relevance. These platforms, among others, employ machine learning and computer vision techniques to enhance the shopping experience for customers. By using Fashion MNIST, they can develop recommendation systems that suggest products based on a user's previous purchases and preferences, leading to increased customer engagement and sales. Additionally, they can implement image recognition algorithms to enable users to search for items by simply uploading a photo, making the shopping process more convenient and intuitive.” [4]

And lastly “In recent years, the fashion industry has witnessed a transformation in the way it operates, with the integration of technology and data-driven approaches. Fashion MNIST serves as a foundational tool for fashion businesses, allowing them to develop and refine machine learning models for image classification, object detection, and even style analysis. By observing the different types of capabilities this dataset can perform, companies can enhance their understanding of consumer trends, streamline inventory management, and ultimately drive growth in the highly competitive online marketplace” [4]. In other words, it plays a key role in the convergence of the fashion and technology market industry and it allows businesses of various sectors (including the clothing industry) to be more up to date when it comes to the innovation in the digital era (especially in online shopping such as Amazon, E-baby, Facebook marketplace, etc.)

# Literature Review:

The specific research paper that I will be replicating is the “Classifying Garments from Fashion-MNIST Dataset Through CNNs” taken from the *Advances in Science, Technology and Engineering Systems Journal* article Volume 6, published and made online in February 2021. This research paper is 6 pages long and consists of various CNN models used to interpret Fashion MNIST and its applications on deep learning.

“Convolutional Neural Network models have been shown efficiency in image c1assification. This paper presents four different Convolutional Neural Networks models that used Fashion-MNIST dataset. Fashion-MNIST is a dataset made to help researchers finding models to classify this kind of product such as clothes, and the paper that describes it presents a comparison between the main classification methods to find the one that better label this kind of data.[3]

In other words, this paper addresses the growing online fashion market's need for algorithms capable of identifying garments. Such algorithms can help companies in the clothing sales sector understand customer preferences, tailor marketing campaigns, and enhance the customer experience while shopping online or at the store. Convolutional Neural Networks (CNNs) are known for their efficiency in image classification, and this paper presents four different CNN models applied to the Fashion-MNIST dataset, I plan to use the same 4 CNN models and see how they are similar or different from one another when I try to calculate their efficiency as presented in this paper. The original research evaluated various machine learning models and achieved 89.7% accuracy using SVM. I will try to use the same technique and see if I end up with the same results (or at least close to it). Also in this paper, the authors propose the use of CNNs to label the Fashion-MNIST dataset, aiming to enhance accuracy. The results show that their new CNN model called "cnn-dropout-3" achieves an accuracy of 99.1%, which was the highest out of all the 15 different models that were tested however it maybe the more time consuming process, which I will investigate.

To summarize the introduction, this paper is an extension of work originally presented at the Iberian Conference on Information Systems and Technologies [3]. It emphasizes the changing dynamics in the fashion industry driven by internet business and the importance of understanding customer preferences and improving the user experience. Classifying clothing is part of the broader task of classifying scenes, and automating this process can assist deep learning researchers and provide insights into users' tastes, culture, and financial status. The original work used various AI models and achieved the best result using SVM with 89.7% accuracy. This paper proposes the use of CNNs for labeling the Fashion-MNIST dataset to improve classification accuracy. I will replicate the same method that they used but if I run into any further problems with the coding, I may switch to using precision or recall instead of accuracy if the results are off by a lot, I want to keep the results consistent as it’s the exact same dataset with the same number of labels as well as same training/test sets.

To summarize the background, this paper discusses the concepts of Machine Learning, Feature Learning, and Deep Learning. Feature Learning is essential for building models capable of pattern recognition, and Deep Learning methods, including Convolutional Neural Networks (CNNs), have shown promising results. CNNs are particularly effective for image classification tasks, and this paper highlights the key components of CNNs, such as convolutional layers, pooling layers, and dropout as a technique to mitigate overfitting. I will do this same technique like what they have done such as cnn-droput-1, cnn-dropout-2, etc. each with its own attributes and parameters such as number of epochs, batch sizes, optimizer used, etc. and see if the accuracy is consistent as to what they have given in the summary table of their paper.

In terms of related work, this specific paper has used a grand total of 28 different reference from various sources although each reference has its own aspects that were discussed but all deal with the Fashion MNIST and how it is used in deep learning and in the fashion industry. The paper references previous research in the field of clothing classification and recognition. It mentions works that used context-sensitive grammars, multi-class learners based on Random Forest, and Bidirectional Convolutional Neural Networks for clothing landmark localization and classification. I will try out Random Forest and bidirectional neural network and see how the results are if they differ a lot from the accuracy that they have provided.

In the dataset section of the paper, the Fashion-MNIST dataset is introduced as a drop-in alternative to the original MNIST dataset, containing grayscale images of fashion products. It has the same structure as MNIST but with fashion items instead of digits (valued from 0-9 in testing and training). The dataset is described as having two CSV files, one for training images and one for testing images, each with 785 columns, including a label column. The dataset's organization and structure are discussed for data access. Also the values in this csv are values from 0-255 depending on how light or dark the brightness is on that specific pixel (for example you can have very light top at 0 or very dark top at 255). This paper aims to explore and implement CNN models for clothing classification using the Fashion-MNIST dataset and compares the results with the original research, achieving a notable accuracy improvement. It is a valuable contribution to the field of machine learning and fashion recognition.

And in terms of CNN model usage, this paper presents four different Convolutional Neural Network (CNN) models (as discussed before) developed using Python with Keras and TensorFlow to label the Fashion-MNIST dataset. Training was conducted in a Jupyter notebook with GPU support, and Weights and Biases were used to monitor training and hardware usage. I will be using the same technique as this with Python but instead of Jupyter notebook, I will be using Google Collab instead and it will be done using GPU just like how they did it. The version of Tensorflow that I will be using is 2.13.0, this may differ from what they have used as an older version was used when this paper first got published online in February 2021, also I won’t be using weight/biases when doing my code for this project. Also the dataset will be pulled from Keras library directly just like how it’s done on this paper, I have used “from keras.datasets import fashion\_mnist” as the main command to get the Fashion MNIST, this is by default set at 60,000 samples for training and 10,000 samples for testing.

In terms of the results of the 4 different CNN models (that I will be replicating), from the paper the 4 different models had the following results:

1. **cnn-dropout-1 and cnn-dropout-3:**
   * These models employ two consecutive blocks consisting of convolution, max pooling, and dropout layers. Each block is connected to two fully connected layers, which, in turn, connect to an output layer with ten neurons, each representing a category.
   * The difference between the two models is that cnn-dropout-3 features considerably lower dropout values compared to cnn-dropout-1.
   * The topology of these models includes 44,426 trainable parameters.
2. **cnn-dropout-2:**
   * This model is similar to cnn-dropout-1 but with two convolution layers before each max pooling operation.
   * It contains around 32,340 trainable parameters.
3. **cnn-simple:**
   * cnn-simple is a simpler model with fewer layers, featuring only two convolution layers followed by a fully connected layer, along with dropout and max pooling layers.
   * It has 110,968 trainable parameters.
   * Due to its structure, the image reaches the dense layer with a size of 14x14 pixels (four times the size of other models), resulting in slower training in the dense layer.

I will try with my own 4 models of the same kind (CNN-simple is one type and CNN-dropout-1/2/3 are the 3 other types). All of these models are implemented using Keras Sequential models and use the Rectified Linear Unit (ReLU) activation functions for convolutional and dense layers. Softmax activation is used for the output layer. The optimizer chosen is Adadelta, with a batch size of 128, and the models are trained for 12 epochs. Additionally, image pixel luminosity values are normalized to float numbers between 0 and 1 to enhance results. These models aim to efficiently label the Fashion-MNIST dataset, making them suitable for real-time applications such as online stores and search websites. I am going to use the same sequential model as they did and the same activation function (ReLu for the dense layer and Softmax for ouput). I will also use the same epoch and batch sizes for all 4 models, but the optimizer I may use RMSProp or Adam instead of Adadelta, in terms of seeing if it make a difference but may use what they have done.

# Data Description:

In this dataset, there are a total of 60,000 images of the training dataset and 10,000 images of the test dataset. The 2 .csv files provided on Kaggle provide the details of all 784 pixels (28x28 in terms of height and width) and each pixel on certain labels range from 0-255 (with 0 being light and 255 being very dark). In terms of the labels of the fashion attire in this dataset, they are ranked according to the numbers they represent:

|  |  |
| --- | --- |
| **Label Number** | **Type of Fashion attire** |
| 0 | T-shirt/Top |
| 1 | Trouser |
| 2 | Pullover |
| 3 | Dress |
| 4 | Coat |
| 5 | Sandal |
| 6 | Shirt |
| 7 | Sneaker |
| 8 | Bag |
| 9 | Ankle Boot |

The above table correlates with the labels that were given on Kaggle. In addition, for the training dataset, there are 6,000 labels each for each attire for a grand total of 60,000 fashion images. For the testing however, its a smaller sample size and only 1,000 labels for each attire for a grand total of 10,000 fashion images. The .csv files do not specify the size of the certain clothing like whether certain shirts or trousers are small, medium, or large or whether any type of footwear such as sandals, sneakers, or ankle boots are size 1, 2, etc. and if its gender specific (for males or females).

# Exploratory Data Analysis:

An embedded technique known as the Random Forest Importance was used to take the top 20 pixels based on the ‘Importance’ value of the Fashion-MNIST train dataset (60,000 rows and 785 columns with column 1 being the dependent variable). The reason top 20 was used was because it is a decent value and the system would not crash (due to limited capabilities) or take a long time to generate a panda profiling report. The following 20 pixels on the train dataset had the highest importance based on the correlation of the dependent variable (label) and other independent variables (the remaining number of pixels). They do contain a high number of zeros, which were taken into account but of a lesser amount than the other pixels.

Rank Pixel # Importance

1 pixel547 0.009806

2 pixel603 0.009392

3 pixel491 0.008809

4 pixel575 0.007279

5 pixel263 0.006986

6 pixel407 0.006045

7 pixel435 0.005642

8 pixel687 0.005324

9 pixel519 0.005292

10 pixel631 0.005287

11 pixel401 0.005284

12 pixel180 0.005119

13 pixel234 0.005010

14 pixel236 0.004944

15 pixel68 0.004934

16 pixel611 0.004891

17 pixel96 0.004734

18 pixel555 0.004687

19 pixel594 0.004502

20 pixel427 0.004482

The number of trees used to generate this output was 100 trees based on the Random Forest Importance (RFI). I chose this value is because it fits in nicely with the 60,000 total rows of the train dataset. Also in practice, the number of trees can be tuned using techniques like cross-validation or grid search to find the optimal value for specific problems within deep learning. According to ChatGPT, “Using more trees can lead to a more robust model but might increase computational complexity but using fewer trees might lead to a less complex model but could lead to the model overfitting. The optimal number of trees can depend on factors like the dataset size, complexity, and available computational resources” [1]. I will experiment with the different number of trees and see which value is the most sufficient to use for this specific dataset of the top 20 pixels. In addition, this will be compared to other ML models such as SVM , XGBoost and the traditional simple-CNN model (with the basic parameters done on a simple scaled test set).

# Modelling (CNN-simple):

To answer one of the research questions from the modified abstract of this project. I compared the simple-CNN model with the basic parameters and compared it with different machine learning models such as Random Forest, SVM and XGBoost. And to compare their accuracies based off the exact same dataset, top 20 pixels from the combined dataset based on RFI 100 trees (with the label column and 70,000 rows).

The following is from the simple-CNN-model:

A screenshot of a computer

Description automatically generated

It gave 75.68% as the test accuracy, which is much lower than expected mainly cause of the small sample size (70,00 rows) and in the research paper there were over 110,000 trainable parameters listed on p. 992 of the article (they also received a test accuracy using 2 different TensorFlow versions) and in my case I only used the physical dataset with the numbers and one TensorFlow version (most recent version) 2.14.0 and Keras version 2.14.0. In addition, the research paper that I am replicating (on section 5.3 cnn-simple, page 992) indicates that “All those models were modeled based on Keras Sequential model. Convolutional and dense layers used Rectified Linear Unit (ReLU) activation functions, except by the last dense layer on each model (output layer), were Softmax was used. The optimizer used was Adadelta, Batch size was 128 and we trained the models for 12 epochs” [2]. I did this differently from what they did. The reason I chose ‘Adam’ was because I had used it in the past for this exact same dataset in the CIND850 course and in general, it is a popular and effective optimizer that works well generally for many deep learning tasks such as image classification (which is used in this case). The choice of optimizer is problem-specific and I used that but other optimizers like what they used (Adadelta) or other optimizers such as ‘SGD’ or ‘RMSprop’ may also work.

As for batch size and the number of epochs, as mentioned earlier I am using only 70,000 rows so I used lower values than the paper did. I used 10 for number of epochs is cause I’m working with a limited number of rows and columns and I didn’t want it to set it too low, which could cause underfitting or setting it too high which can lead to overfitting. I think 10 is a decent value for number of epochs in this particular situation, however if it was over 100,000 rows I may have done 100 epochs to speed up the process. For batch size, 64 is a decent value and I think it will work well for this particular dataset, reason being too high of a batch size can speed up the training but use too much computer memory (which I am limited to) and too low of a batch size such as 16 or 32 can be decent as well since a smaller batch size allows for more frequent weight updates and can help the model converge faster, but it may also require more epochs. That is why I chose 64 as the batch size value.

For the loss values/accuracy, steps 1-10 epochs gave 10 different loss & accuracy values along with the final test accuracy when the model was completed. Based on the observations, the numbers on the accuracy kept increasing in every single succeeding epoch (from 1 to 2, 2 to 3, etc.), whereas the loss value decreased in every single step (from 1-10 epochs). The loss value, in particular is obtained by the loss function that is defined when compiling the model. The value is calculated by assessing the difference between the predicted value and the actual value. A common loss function is 'categorical cross-entropy' (which I used) for multi-class classification tasks like Fashion-MNIST. As for the accuracy, it is reported in the training output, and it’s calculated by the model's evaluation on the provided data (70,000 rows, 21 columns) in each epoch. It represents the proportion of correct predictions the model makes. During training, the accuracy is calculated based on the model's predictions for the training dataset, comparing them with the true labels. There may have been trial & error involved in this due to many factors such as the sample size being used and only the top 20 pixels being used to train the whole dataset. This could have been higher if more epochs/batch size were used but due to memory constraints only the top 20 were considered. According to ChatGPT, “The loss and accuracy metrics provides with more details into the model's learning and generalization to new data. The goal is to reach an optimal point where the model isn't overfitting or underfitting (somewhere in between) and has a good generalization ability” [1].

Lastly, in terms of the parameters that were used when making the simple-CNN-model, there were 2 hidden layers (the ‘Conv2D’ and ‘MaxPooling2D’). In the code, the input layer is created using the Conv2D method: ‘layers.Conv2D(32, (3, 1), activation='relu', input\_shape=(20, 1, 1))’. This sets up the initial convolutional layer with 32 filters/kernels applied to the original data (70,000 rows) and each filter/kernel generates a different output each time, the (3, 1) is the filter size (a 3x1 matrix), using the ReLU activation function, and an input\_shape of (20, 1, 1), which indiactes the dimensions of the input data (20 x 1 x 1, in other words 20 rows, 1 column and black-and-white image) depicting the Fashion-MNIST dataset. As for the Max pooling, the ‘(2,1)’ represents the pooling size (2 rows 1 column pooling matrix). There were 2 dense layers, with 2 different activation functions, the values of ‘128’ and ‘10’ are used as the number of neurons in each respective dense layer. 10 is used on the 2nd dense layer because there are 10 classes of fashion images (labels 0-9, representing different fashion attire). In the first dense layer, it uses ‘ReLU’ as the activation function for the hidden layer with 128 units and it is effective in training deep neural networks. In the second dense layer, ‘Softmax’ is used as the output layer's activation function for multi-class classification (10 classes), generating class probabilities. The class with the highest probability is usually predicted.

The formula used for accuracy and loss (categorical cross entropy in this case) are the following:

**Accuracy = Number of Correct Predictions / Total Number of Predictions** [6]

**Categorical Crossentropy Loss= −∑i​yi​⋅log(pi​)**

Where:

* yi is the true label for class i.
* pi​ is the predicted probability of class i assigned by the model.

In the first particular epoch of this code, we are given the output:

Epoch 1/10

1094/1094 [==============================] - 14s 4ms/step - loss: 1.4770 - accuracy: 0.6320

In the context of the neural network training, during each epoch, the model makes predictions on the training data, compares these predictions to the actual labels, and calculates the accuracy based on the formula above. The loss, on the other hand, represents a measure of how well the model is performing, typically calculated using a loss function like categorical crossentropy (which I used in this case). As for the results of accuracy on each specific epoch:

* For each epoch during training, the accuracy is calculated based on the training data (accuracy: 0.6320 for the first epoch).
* After training is complete, the final accuracy is calculated on the test data (Test accuracy: 75.68%).

The loss value (1.4770 in the first epoch) represents the value of the loss function (categorical crossentropy) on the training data during that specific epoch. The goal during training is to minimize this loss, which indicates how well the model's predictions match the actual labels. The categorical crossentropy loss is commonly used for multi-class classification problems, in this case we are dealing with 10 different labels (labels 0-9, each number representing a different attire), so this specific loss function make the most sense to use.

# Random forest modelling:

The following code is from the Random Forest method:

A screenshot of a computer

Description automatically generated

I ran 3 different values for the ‘n\_estimators’ to try out 3 different values for the trees and see which gives the most significant results. They were very close with one another (less than 3 percent difference), 10 trees gave an accuracy of 73.51%, 50 trees gave an accuracy of 75.41%, whereas 100 trees gave a 76.11% test accuracy. These values can differ each time due to the specific trees being chosen to be evaluated and it is very high due to only the top 20 pixels being chosen from test set (784 total pixels in total). If 100-200 pixels were used instead of 20, the test accuracy may have been lower but can’t be used due to the system crashing. In the research paper, it was not indicated how many trees were used but it gave a 87% accuracy result on the table.

In addition, I also created a graph (full code on the .ipynb file) depicting the graph of the random forest trees based on it’s test accuracy on the 70,000 rows of data (top 20 pixels). I tested on a wide interval, from 0-500 trees:

A graph with a line going up

Description automatically generated

A grand total of 12 different values for trees were used in this interval (as shown on the graph above, 12 blue points) and from the looks of this, it stabilizes after 400 trees were compiled. The following was the output of the results:

Test accuracy (n\_estimators=10): 73.51%

Test accuracy (n\_estimators=20): 74.70%

Test accuracy (n\_estimators=50): 75.54%

Test accuracy (n\_estimators=100): 76.11%

Test accuracy (n\_estimators=150): 76.13%

Test accuracy (n\_estimators=200): 76.37%

Test accuracy (n\_estimators=250): 76.36%

Test accuracy (n\_estimators=300): 76.41%

Test accuracy (n\_estimators=350): 76.48%

Test accuracy (n\_estimators=400): 76.52%

Test accuracy (n\_estimators=450): 76.59%

Test accuracy (n\_estimators=500): 76.49%

When it compiles the ‘500’ tree value, it goes down but 0.10% from the ‘450’ tree value and this output was generated in intervals of 50 trees at a time. So most likely, 450 trees may have been the optimal value to use as this point in the graph is where the line flattens and the accuracy is more or less the same moving forward in the graph. In my situation however, due to the memory constraint I stuck with 100 trees but for better results 450 is more beneficial to use but it can take a long time to compile and run. There is less than one percent difference between 100 and 500 trees, so depending on how much memory one’s CPU has, any value between can work.

# Effectiveness:

For effectiveness, I used the following classification metrics: confusion matrix, accuracy, precision, recall & F-1 score to get the values. For the accuracy specifically on the combined dataset (70,000 rows) this is the following code:

A screenshot of a computer code

Description automatically generated

I made a couple of adjustments with the parameters as I was getting an error, due to compatibility issues. I had to reshape the ‘X\_test’ to the input data to match the CNN input shape because Convolutional layers in a CNN expects that the input data to have a certain shape, usually a 4D tensor. The shape of the input data for a 2D convolutional layer is (batch\_size, height, width, channels). But in my case, the CNN expects a 4D tensor with dimensions (batch\_size, 20, 1, 1). The 20 represents the number of pixels (top 20 pixels based on Importance in the combined dataset), and the first ‘1’ represents the height of the reshaped data and the second ‘1’represents the channel (grayscale, in this situation). So, reshaping is done to ensure that each image (rows) in the test set has the shape that the CNN model expects. In addition, the context of reshaping for CNN models, the number of pixels is the width of the dataset and the sample are treated as rows.

The other adjustment made was the ‘y\_test’ was one-hot encoded since the model used categorical crossentropy (due to multiple classes labeled 0-9 for different fashion attires). It converts these labels into binary vectors of length 10, where the index corresponding to the class is marked as 1, and others are 0. This is important for training and evaluating classification models, which we have in this case since there are 10 different fashion attires for 70,00 total rows and the grayscale images varying from 0-255 (depending on how dark or bright a certain pixel is). In summary, reshaping and one-hot encoding are necessary preprocessing steps to ensure that the input data matches the expected format of the CNN-simple-model and that the labels are appropriately represented for categorical classification.

The confusion matrix and classification report of **simple-CNN-model**:

Confusion Matrix (CNN-Simple Model):

[[1121 11 35 81 15 2 79 0 30 0]

[ 23 1301 12 47 4 1 7 0 4 0]

[ 38 2 1038 10 76 2 181 2 44 1]

[ 265 14 31 1003 26 6 30 1 18 12]

[ 129 6 459 56 495 1 264 0 33 0]

[ 1 0 0 2 0 1104 0 211 26 50]

[ 379 8 301 48 69 1 495 0 76 0]

[ 0 0 0 0 0 85 0 1223 4 71]

[ 11 4 61 12 13 22 38 9 1220 8]

[ 0 0 0 9 0 20 0 113 15 1275]]

Classification Report (CNN-Simple Model):

precision recall f1-score support

0 0.57 0.82 0.67 1374

1 0.97 0.93 0.95 1399

2 0.54 0.74 0.62 1394

3 0.79 0.71 0.75 1406

4 0.71 0.34 0.46 1443

5 0.89 0.79 0.84 1394

6 0.45 0.36 0.40 1377

7 0.78 0.88 0.83 1383

8 0.83 0.87 0.85 1398

9 0.90 0.89 0.90 1432

accuracy 0.73 14000

macro avg 0.74 0.73 0.73 14000

weighted avg 0.74 0.73 0.73 14000

I ended with a 73% accuracy, which is different from the above because the code may have used different values when it did the 80-20 train-test split from the original. It also shows the precision, recall and accuracy of each specific class one-by-one for all the 10 total fashion attires in the dataset. Due to the nature of this dataset (even when using random\_state=0) the values fluctuate each time the code is run and the numbers change for the confusion matrix and therefore the values for accuracy, precision and recall may also change (but still close to the 70-75% range)

The values vary each time the code is run so it may be inconsistent, it’s a 10x10 matrix since there are 10 different classes we are dealing with (10 different fashion attires label 0-9). Each row of the matrix represents the instances (observations or individual data point) in an actual class, whereas each column of the matrix represents the instances in a predicted class. There are also 4 different evaluation measures (True positive, False positive, True negative, and False negative) that we get from evaluating the elements (values) inside it. According to ChatGPT,

* **“True Positives (TP):** The diagonal elements represent the number of instances that were correctly predicted. For example, the element in the first row and first column (1002) is the number of instances of class 0 that were correctly predicted as class 0.
* **False Positives (FP):** These are the instances that were predicted as positive but were actually negative. For example, in the first row, the sum of values (4 + 31 + 180 + 45 + 2 + 87 + 0 + 23 + 0) excluding the diagonal represents the instances that were incorrectly predicted as class 0.
* **False Negatives (FN):** These are the instances that were predicted as negative but were actually positive. For example, in the first column, the sum of values (29 + 23 + 186 + 90 + 1 + 311 + 1 + 18 + 0) excluding the diagonal represents the instances of class 0 that were incorrectly predicted as other classes.
* **True Negatives (TN):** The sum of all values outside the first row and first column represents instances that were correctly predicted as not being in class 0.

In summary:

* Each entry (i, j) in the confusion matrix is the number of instances with true class i that were predicted as class j.
* The diagonal elements (from top-left to bottom-right) represent correct predictions.
* Off-diagonal elements represent incorrect predictions.” [4]

The confusion matrix and classification report of the **Random forest (400 trees) model**:

Random Forest Confusion Matrix:

[[1081 10 33 103 32 2 89 0 24 0]

[ 16 1325 7 42 4 1 4 0 0 0]

[ 25 3 943 14 228 2 141 0 38 0]

[ 179 12 28 1085 55 6 33 0 4 4]

[ 80 5 250 63 885 0 133 0 27 0]

[ 0 0 0 2 0 1147 0 177 20 48]

[ 310 7 256 66 206 2 476 0 54 0]

[ 0 0 0 0 0 76 0 1218 4 85]

[ 4 4 39 12 44 14 21 10 1246 4]

[ 0 0 0 4 0 25 0 85 11 1307]]

Random Forest Classification Report:

precision recall f1-score support

0 0.64 0.79 0.70 1374

1 0.97 0.95 0.96 1399

2 0.61 0.68 0.64 1394

3 0.78 0.77 0.78 1406

4 0.61 0.61 0.61 1443

5 0.90 0.82 0.86 1394

6 0.53 0.35 0.42 1377

7 0.82 0.88 0.85 1383

8 0.87 0.89 0.88 1398

9 0.90 0.91 0.91 1432

accuracy 0.77 14000

macro avg 0.76 0.76 0.76 14000

weighted avg 0.76 0.77 0.76 14000

In the previous stage, I used various values for random forest (ranging from 0-500) and when I made the graph of it, it seems to have flattened a bit after 400 trees so that is what I used as my specific value for the random forest.

The confusion matrix and classification report of the **SVM model**:

SVM Confusion Matrix:

[[1077 13 34 133 39 1 59 0 18 0]

[ 30 1306 5 45 5 2 4 0 1 1]

[ 21 8 802 16 331 4 164 0 47 1]

[ 259 18 29 1014 44 10 20 0 10 2]

[ 132 5 243 39 854 2 118 0 50 0]

[ 1 0 0 4 0 1091 0 234 17 47]

[ 354 6 322 64 235 8 320 0 68 0]

[ 0 0 0 0 0 87 0 1193 3 100]

[ 11 3 41 18 48 26 20 9 1214 8]

[ 0 0 0 12 0 16 0 100 10 1294]]

Classification Report (SVM):

precision recall f1-score support

0 0.57 0.78 0.66 1374

1 0.96 0.93 0.95 1399

2 0.54 0.58 0.56 1394

3 0.75 0.72 0.74 1406

4 0.55 0.59 0.57 1443

5 0.87 0.78 0.83 1394

6 0.45 0.23 0.31 1377

7 0.78 0.86 0.82 1383

8 0.84 0.87 0.86 1398

9 0.89 0.90 0.90 1432

accuracy 0.73 14000

macro avg 0.72 0.73 0.72 14000

weighted avg 0.72 0.73 0.72 14000

For this particular model, there were different SVM kernels to choose from, I specifically used radial basis function (rbf) since we are dealing with image data (70k rows) and this is a specific type of non-linear kernel which makes the most sense to use since we are working with pixel values ranging from 0-255 of various fashion attires. I used a C-value of 1.0 in the code to get this output as it seemed to be the one most consistent, this represents how well the regularization strength is in the model. As well as it results in a larger-margin hyperplane, whereas larger C-value results in a smaller-margin hyperplane.

The confusion matrix and classification report of the **XGBoost Model**:

XGBoost Confusion Matrix:

[[1075 7 32 101 35 2 101 0 20 1]

[ 9 1324 8 39 9 1 8 0 1 0]

[ 31 2 909 13 235 0 174 0 29 1]

[ 156 19 26 1079 48 2 61 0 10 5]

[ 67 3 247 65 864 1 173 0 23 0]

[ 0 0 1 4 0 1151 0 165 16 57]

[ 307 8 255 57 177 0 521 0 52 0]

[ 0 0 0 0 0 83 0 1223 3 74]

[ 6 3 44 13 25 23 31 4 1244 5]

[ 1 0 0 6 0 27 0 79 11 1308]]

Classification Report (XGBoost):

precision recall f1-score support

0 0.65 0.78 0.71 1374

1 0.97 0.95 0.96 1399

2 0.60 0.65 0.62 1394

3 0.78 0.77 0.78 1406

4 0.62 0.60 0.61 1443

5 0.89 0.83 0.86 1394

6 0.49 0.38 0.43 1377

7 0.83 0.88 0.86 1383

8 0.88 0.89 0.89 1398

9 0.90 0.91 0.91 1432

accuracy 0.76 14000

macro avg 0.76 0.76 0.76 14000

weighted avg 0.76 0.76 0.76 14000

In this model, when creating it, I used objective='multi:softmax' on the code, which specifies that the model should perform multiclass classification, and num\_class=10 as 10 different classes of the fashion attire were being evaluated.

**Summary Reports (for 4 ML models):**

Here is a table summarizing the outputs from above:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Accuracy** | **Precision (Weighted Avg)** | **Recall (Weighted Avg)** | **F1-Score (Weighted Avg)** |
| CNN-Simple | 0.73 | 0.74 | 0.73 | 0.73 |
| RandomForest  (400 trees) | 0.77 | 0.76 | 0.77 | 0.76 |
| SVM | 0.73 | 0.72 | 0.73 | 0.72 |
| XGBoost | 0.76 | 0.76 | 0.76 | 0.76 |

This is a similar result as to what I had for the test accuracy in the previous stage of this project. The CNN-simple was at 75.68 but decreased to 73%, the other 3 more or less stayed the same. However, in this situation the classification reports now show the precision, Recall, and F1-score which was not discussed earlier. I took the weighted average for all 10 classes to give one final average value for these 3 evaluation measures.

By overall comparison, randomforest had the highest accuracy among the 4 models at 77% and CNN-simple & XGBoost had moderate accuracies, whereas SVM had the lowest accuracy. The reason randomforest had the highest was because 400 trees were used and could have been less if smaller values were used.

**Formulas being used for evaluation measure (for 4 ML models):**

Each of the 4 models had all given a 10x10 confusion matrix followed by their respective classification report. Since there were 10 different classes, that is why a 10x10 matrix was produced and thus there were 100 values inside the confusion matrix. In addition, the precision and recall were evaluated 10 different times for each model (once for each class) and the accuracy was given 1 specific value (for the entire model) and in my specific case, 70,000 rows of the combined top 20 pixels ranked were used. The evaluation measures such as accuracy, precision, recall, and F-1 Score were done using specifically the scikit-learn package in Python, the library handles the computation done for each class (labels 0-9) and it provides a specific number for it (between 0-1). Also the scikit-learn (sklearn) library is a popular machine learning library in Python that provides tools for data preprocessing, modeling, and evaluation. I used this particular library, mainly cause I was dealing with a 10x10 matrix for each of the 4 models and doing computations one-by-one for each of the 10 total classes would be tedious especially with 70,000 rows of data which is what I have in my dataset.

For the formulas that were used by Python on my code, according to ChatGPT:

“The formulas for accuracy, precision, and recall are the same across the different classification models. Here are the formulas for each metric:

1. **Accuracy:**
   * Formula: Number of Correct Predictions / Total Number of Predictions
   * In other words, accuracy is the ratio of correctly predicted instances to the total number of instances.
2. **Precision (for a specific class i):**
   * Formula: [True Positives (TP) for class i] / [True Positives (TP) for class i + False Positives (FP) for class i]
   * Precision is the ability of a classifier not to label an instance as positive when it is negative.
3. **Recall (for a specific class i):**
   * Formula: [True Positives (TP) for class i] / [True Positives (TP) for class I + False Negatives (FN) for class i]
   * Recall (Sensitivity) is the ability of a classifier to find all positive instances.
4. **F1-Score (for a specific class i):** F1=2 × [ (Precision for class i × Recall for class i) / (Precision for class i + Recall for class i) ]

* The F1-Score is the harmonic mean of precision and recall, providing a single metric that combines both precision and recall. It ranges from 0 to 1, where a higher value indicates better performance. Like precision and recall, the F1-Score can be calculated for each class in a multi-class classification problem, and various averaging methods (micro, macro, weighted) can be used to obtain an overall F1-Score for the entire model.

These formulas apply to each class separately when dealing with multi-class classification problems. The overall metrics are usually calculated by averaging or aggregating these class-specific metrics.

The formulas are consistent across different classification models, including Random Forest, SVM, XGBoost, and the simple CNN model.” [4]

**Matthew’s Correlation Coefficient (for 4 ML models):**

I used one additional metric called matthew’s correlation coefficient (MCC) to test out 4 different values of it and see how it compares with one another. The MCC can be useful in this particular case since my data can be imbalanced due to it being trial & error in some situations such as when computing the Simple-CNN model (it gives different values each time it is run). The accuracy can be off in certain situations like this when the values change for the confusion matrix and that affects the accuracy, precision, and recall values. In addition, the MCC “takes into account true positives, true negatives, false positives, and false negatives, providing a single metric that ranges from -1 to 1, where 1 represents a perfect prediction, 0 indicates no better than random prediction, and -1 shows total disagreement between prediction and observation.” [4]. I used a particular package from the sklearn library to test it out, I used the “from sklearn.metrics import matthews\_corrcoef” command and did them 4 times for each model.

The following are the MCC for each model:

Simple CNN MCC: 0.7004829424477254

Random Forest MCC: 0.7399321629849797

SVM MCC: 0.6971706852726118

XGBoost MCC: 0.7383788683550587

In my case, for all the 4 models they range from 70-74% and since I have a multi-class classification problem with 10 classes, MCC is computed as the correlation between the observed and predicted classifications after converting them into binary values. The reported values for MCC are reasonably decent, which suggest that the models are making predictions that are better than random (although not quite perfect as some models are done via trial and error). According to ChatGPT (when prompted what each value means for each model), it says:

“Here's a brief interpretation of the MCC values:

* Simple CNN MCC: 0.70
  + The model's predictions have a strong positive correlation with the true labels, indicating good performance.
* Random Forest MCC: 0.74
  + The Random Forest model has a strong positive correlation with the true labels, suggesting good predictive performance.
* SVM MCC: 0.70
  + The SVM model's predictions have a positive correlation with the true labels, indicating reasonable predictive performance.
* XGBoost MCC: 0.74
  + The XGBoost model has a strong positive correlation with the true labels, suggesting good predictive performance.

In general, MCC values closer to 1 indicate better model performance. It's a useful metric when dealing with imbalanced datasets and provides a more comprehensive evaluation than accuracy alone.” [4] So overall, all 4 of them have good to reasonable performance meaning that they correlate closely with the actual accuracy values from the table given above (on page 30), although the actual values between the MCC and the accuracy values are slightly different.

The formular for Matthew’s Correlation is the following:

A math equation with a line and a plus and a line

Description automatically generated with medium confidence

This came from the site: <https://www.statisticshowto.com/matthews-correlation-coefficient/> [5] This formula was used 4 different times for each model, all had positive values.

# Efficiency:

**Training and Test times for 4 models:**

For the train/test times, I tested the amount of time it takes to train and test the 4 various models. For CNN-simple model, here is the training and testing time below:

Training Time - CNN-Simple: 25.40 seconds

Testing Time - CNN-Simple: 1.04 seconds

It took about 25.40 seconds to train the model and testing took a second. Running the same code (given on the .ipynb file) results in various different outputs ranging from 25-28 seconds for training and 1-3 seconds for testing, and this was done mainly via trial & error (even with random\_state=0 being used). There are several factors that can have this effect such as how the computer operates, how much memory it has, etc. so it is not 100% accurate.

For the random forest efficiency, in the previous stage different values for trees were experimented with and based off the graph from earlier it seems that it flattens at 400 trees, so I used 400 as the ‘n\_estimator’ value to determine the training and testing times. This was the output:

Training Time - Random Forest (400 trees): 33.90 seconds

Testing Time - Random Forest (400 trees): 0.76 seconds

The random forest took longer than the simple CNN-simple model with the same training and test split. Also, in addition I took the max\_depth of the random forest to be 10 in this particular situation because that is on the 10th level that it cuts off. I think this specific value is good to use because it keeps the results stable for using 400 total trees. If less trees were used, a different max depth value could be more adequate. The training and testing time do change based on the ‘max\_depth’ value (given on the code), if it was a different number the training could be slower or faster depending on what the user puts in.

The training and testing times for the CNN-Simple model provided on the output seem reasonable. Training time is the time it takes to train the model on the training data, and testing time is the time it takes to evaluate the model on the testing data.

The testing time is typically lower than the training time because during testing, the model is making predictions on the input data without updating its weights. The testing time of 1.04 seconds for the given CNN-Simple model on the combined dataset (70,000 rows) seems reasonable and is influenced by factors such as the complexity of the dataset and the size of the testing dataset. This could have changed significantly if the number of epochs and batch size used was larger than 10 & 64 respectively, but due to memory constraints it is kept at that amount.

For the SVM:

Training Time - SVM: 71.45 seconds

Testing Time - SVM: 33.54 seconds

The following is for XGBoost:

Training Time - XGBoost: 14.33 seconds

Testing Time - XGBoost: 0.26 seconds

So based on comparing these 4 different ML models, the XGBoost was the fastest in both the training and testing time. On the paper that I am replicating it doesn’t quite indicate the efficiency of training and test times on these 4 particular models, but it does give the accuracy of the results in a percentage in a table on the last page. They didn’t use XGBoost on their end, they used ‘Gradient boosting’ which is similar to it and ended up being the 6th best model in terms of accuracy. They also mention “When evaluating time, the two more accurate models, were also de faster in training.” [3] However in their particular case, it ended up being the 2 different CNN-dropout models which I didn’t experiment with in this situation.

**Memory Consumption for 4 models:**

For the memory consumption, I used a library called memory\_profiler and had to install it as it wasn’t on Collab already. The following code pip statement was used to install it:

pip install memory-profiler

After it was installed, I ran the code and for the Simple-CNN model (70,000 rows) I got:

peak memory: 5759.64 MiB, increment: 1.16 MiB

According to ChatGPT,

* **“Peak Memory:** This is the maximum amount of memory (in megabytes, MiB) that was used by the program during its execution. In this case, the peak memory usage was 5759.64 MiB.
* **Increment:** This represents the increase in memory usage from the start to the peak memory point. In this case, the memory increased by 1.16 MiB.” [4]

So, in summary, my chunk of code for the Simple-CNN model reached a peak memory usage of 5759.64 MiB, and the memory increased by 1.16 MiB during its execution. This information is helpful for understanding the memory requirements for my code. I also used a 16GB RAM laptop computer to run it and depending on the machine it could differ due to the system requirements on the device.

The research paper I am replicating does not give any details at all on the memory consumption for any of their models, so it is hard to tell if I am making a good estimate on the memory consumption.

For the Random forest (400 trees):

Peak memory: 6526.09 MiB

Memory increment: 735.34 MiB

I used a different method for randomforest than I did in CNN-simple because it is a different model and it may not work with certain type of code, including the fitting process of machine learning models that involve multiple steps internally (like fitting each tree in a random forest). I used ‘import resource’ in this scenario and got those results (given on the code).

The Random forest (400 trees) took longer than the simple-CNN model, it took about 3 minutes to run. The memory increment is a measure of how much extra memory the program required to train the Random Forest Classifier. This increase in memory is expected, as the Random Forest Classifier builds multiple decision trees during training, and each tree contributes to the overall memory usage. It also builds an ensemble of decision trees, and each tree is built independently, which can be computationally expensive. In my particular case, it was a high number of trees being used (400) since that is where it flattens, if less trees were used then it could have lower memory compared to simple-CNN.

For the XGBoost:

Peak memory: 6593.18 MiB

Memory increment: 0.00 MiB

In this case, I used a different library called ‘psutil’ using the command “!pip install psutil” because the memory-profiler and the other library was not working. It was the highest peak memory out of all the models and as expected due to the very fast train and test times. According to ChatGPT, “A memory increment of 0 MiB indicates that the memory usage did not increase during the training phase. This might happen in scenarios where the memory is allocated and deallocated quickly, and the peak memory is captured right after the allocation.

In some cases, the memory profiling tools might not be able to capture small increments accurately, leading to a reported memory increment of 0 MiB.” I think that unlike CNN-simple (where there is a lot of parameters) and random forest (dealing with several trees), the XGBoost is more precise to use even though it consumed the most memory it was the fastest and was computed within 20 seconds of running the code.

# Stability:

I used k-fold cross validation on the 4 models to compare the stability, I specifically used 10 as I think finding 10 different folds’ accuracy and putting them in an array to get the variance will tell me how stable the model is (if it’s close to 0). I also specifically used this particular library ‘from sklearn.model\_selection import KFold’ on Google collab to get my results.

For **simple-CNN-model** (training test – 60,000 rows):

Accuracy scores for each fold: [73.1166660785675, 72.76666760444641, 74.6500015258789, 73.7333357334137, 73.98333549499512, 73.416668176651, 74.11666512489319, 71.93333506584167, 74.56666827201843, 74.36666488647461]

Mean accuracy: 73.66500079631805

Variance: 0.6753579257509301

The values differ each time the code is run, even when using ‘random\_state=0’ as different executions of the code produces different 10-folds each time and there could be external factors or additional randomness in the Colab environment. I ran it 3 times and this was one of the outputs, on the other outputs the variance was 0.36 and 0.34 so this was pretty close. Overall, the mean accuracy was in the 73.4-73.7 range and the variance is relatively low (0.68 in this particular case). These values suggest that the model's performance is consistent across different folds, and the variance is not very high. A lower variance indicates more stability, and my results are within a reasonable range. So, based on these outputs, the model appears to be relatively stable across different folds in this particular cross-validation. One thing I will note is that the shuffle was turned on in this particular case, I tried it with off in one of the previous runs and it didn’t work too well. The data was originally shuffled before being split into folds but with random\_state=0 and the ‘tf.random.set\_seed(0)’, it should make the shuffling process reproducible unless there are other factors involved (which in my case, there was).

When I run the exact same code for the test set (10,000 rows) but with no k-fold cross validation (so no mean accuracy or variance), with just one test accuracy I get this output:

Epoch 1/10

938/938 [==============================] - 6s 5ms/step - loss: 1.5211 - accuracy: 0.6323

Epoch 2/10

938/938 [==============================] - 4s 4ms/step - loss: 0.8471 - accuracy: 0.6905

Epoch 3/10

938/938 [==============================] - 4s 4ms/step - loss: 0.7832 - accuracy: 0.7103

Epoch 4/10

938/938 [==============================] - 4s 4ms/step - loss: 0.7517 - accuracy: 0.7202

Epoch 5/10

938/938 [==============================] - 4s 4ms/step - loss: 0.7317 - accuracy: 0.7261

Epoch 6/10

938/938 [==============================] - 4s 4ms/step - loss: 0.7175 - accuracy: 0.7305

Epoch 7/10

938/938 [==============================] - 4s 4ms/step - loss: 0.7054 - accuracy: 0.7348

Epoch 8/10

938/938 [==============================] - 5s 5ms/step - loss: 0.6926 - accuracy: 0.7393

Epoch 9/10

938/938 [==============================] - 4s 4ms/step - loss: 0.6806 - accuracy: 0.7439

Epoch 10/10

938/938 [==============================] - 4s 4ms/step - loss: 0.6720 - accuracy: 0.7469

313/313 [==============================] - 1s 3ms/step - loss: 0.7081 - accuracy: 0.7352

Test accuracy on the test set: 73.52%

When comparing both this value with the accuracy for the train set (60k rows), they are both really close with one another (just off by 0.10-0.15) so this tells me that the model is stable. If it runs again, the values may change (as ‘random\_state’ does not work here) and the test accuracy in this particular case can be a bit higher (74 or 75) but it still is close so my model is neither under or overfitting. According to ChatGPT,

“Overfitting occurs when a model performs well on the training data but poorly on new data. Underfitting occurs when a model doesn't capture the underlying patterns in the training data and performs poorly on both the training and testing sets. Having similar accuracies on both sets suggests that my model has found a good balance and is performing well on unseen data. It's always a good practice to check other metrics and explore the model's behavior further, but based on the provided information, there is no clear evidence of overfitting or underfitting.” [4]

With that being said, I think its good overall however I do think it could change if different number of epochs or batch sizes or even the optimizer type was used, this could vary from model to model. I stuck with the basic parameters with 10 epochs, 64 batch size, and ‘adam’ optimizer to stay on the safe side as well as my computer memory was restricted and could not handle a huge amount of rows or it would crash.

For **Random forest (400 trees)** on the train set:

Accuracy scores for each fold: [76.3, 75.88333333333334, 77.38333333333334, 76.61666666666666, 76.16666666666667, 76.41666666666667, 76.28333333333333, 76.11666666666666, 76.2, 77.26666666666667]

Mean accuracy: 76.46333333333334

Variance: 0.21904444444444468

On the test set (10k rows):

Test accuracy on the test set: 76.29

For **SVM** on the train set:

Accuracy scores for each fold: [72.03333333333333, 72.18333333333334, 73.68333333333334, 72.98333333333333, 72.43333333333334, 72.11666666666666, 72.68333333333334, 72.66666666666667, 72.83333333333334, 73.3]

Mean accuracy: 72.69166666666666

Variance: 0.2548472222222234

On the test set (10k rows):

Test accuracy on the test set: 72.75

Just like with the simple-CNN-model there is no overfitting or underfitting in these 2 models, they are almost the same in terms of value. I specifically used 400 trees which is what I used overall in this project since the beginning (as this is where it flattened from the graph in the previous stage), using different values of trees might impact it. Also for SVM, it’s almost the same accuracy as the simple-CNN model but the variance is much lower (0.25 compared to 0.67), I used the same parameters as before for SVM, there are different kernels and ‘C’ values, I used rbf (radial basis function) and C=1.0 to get a decent result as I explained earlier why I used them.

For the last model (XGBoost), here are the results –

On the train (60k rows) set:

Accuracy scores for each fold (XGBoost): [0.77 0.75916667 0.76383333 0.76883333 0.759 0.76666667 0.76166667 0.76266667 0.76333333 0.76266667]

Mean accuracy (XGBoost): 0.7637833333333334

Variance (XGBoost): 1.2383611111111161e-05

On the test (10k rows) set:

Test accuracy on the test set (XGBoost): 76.40%

This was more or less the same as the previous 3 models where the accuracy on the train set matches (or at least close) to the one on the test set however the variance is much smaller compared to the other models. This may be due to the fact that it was originally getting a higher test accuracy value when computed by itself (99% test accuracy in the previous section). Also the XGBoost algorithm itself is known for being robust and having low variance. It tends to perform consistently across different subsets of the data.

# Research Limitations and Improvement:

The research I did overall had some key limitations from the original source paper that I was trying to replicate it with. As mentioned before, I only worked with the top 20 out of the 784 total pixels that were given in the original dataset from Kaggle based on Rank Importance. There were a total of 60,000 training data and 10,000 testing data that were combined to form 70,000 total rows of the best 20 pixels along with the label column (dependent variable). In the research paper however, in some cases they used over 100,000 trainable parameters to build their model like the CNN-simple model and I could not do this because of the memory constraint I had on my computer’s RAM. I only had 16GB to work with and the system would crash or take a very long time if I tried to use additional parameters and that may corrupt my system. One additional thing, this wasn’t mentioned on what specific data they used but in my case, I used the 2 .csv files that had numbers on them varying from 0-255 (70,000 rows) and not the actual image data. If they used the image data, it could result in different outcomes depending on what is evaluated. I also considered replicating 4 different ML models like they did, but in their case they used CNN-simple along with 3 dropouts (of different parameters) and they ended up with accuracy values in the high 90’s, in my situation my 4 ML models were 70-75% in accuracy due to the limited data being used. In addition, I also compared the precision, recall and F1-score of each model which they did not do.

In terms of improvements made to it, I think some stuff can easily be fixed to replicate the paper the way it is published. For example, in my case I used the 2 .csv files with numbers in them and used a sample of it (by taking the top 20 pixels) but the paper most likely used the actual image data. If a better computer was used with better proficiency with memory consumption and no crashes, this could have been more beneficial and gave more accurate results. Also in there report, they compared 15 different types of ML models and made graphs and interpretation using 4 of them and outlined what they got in a table at the very end of the report. I could not do all 15 of them due to the time constraint and I chose to do 4 ML models like they did but different ones (one of them being the same but with different parameters).

# Conclusion:

Going back to one of the original research questions: “Is CNN's performance practical when compared to traditional machine learning techniques such as Random Forest, SVM or XGBoost?” This may need to be rephrased as there are several factors as to how it outperforms the others and in some cases it does not. Originally when comparing accuracy it was tied for the lowest out of all 4 models in that category and did a bit better in precision and was lowest in recall and F1-score, also it was the 3rd best out of the 4 when comparing the Matthew’s correlation. It was also the 2nd faster in terms of training/testing time out of the 4, however basic parameters were used due to memory constraint and also took the least amount of bytes consumed when training it. I’d say it’s overall consistent with the other 3 models that were compared to throughout this project. One thing I did not do is compare it to the 3 dropout models that the research paper I was replicating did. The research paper had 3 dropout models of CNN labeled CNN-dropout-1, droput-2, etc. and all 4 had gained accuracy percentage of high 90’s (after performing various tasks) and also each dropout model had its own unique parameters and characteristics that were tested against one another. This could be a key factor in terms of how practical it is. In addition, a different environment may have been a factor. I used Google collab in my end with the 2 .csv files and keras and tensorflow as my main library packages (same as what they did) but they used JupyterNotebook with the actual image data and GPU built in, which I don’t have on my system.

Some things they did not do on their end was training and testing time for each model (which I did) as well as memory consumption, the amount of bytes it took to train a particular model. They also didn’t do k-fold cross validation for their top 4 models which I did and compared the accuracy on both the training and testing and all 4 of them had +/- 1 between one another so no overfitting and underfitting resulted. Overall, this project was fun to do replicating a paper and comparing 4 different ML models to various things within deep learning. I hope more research is done on this topic in the future to improve the outcome of the results that are already published.

# References:

[1] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

[2] Kayed, M., Anter, A., & Mohamed, H. (2020, February). Classification of garments from fashion MNIST dataset using CNN LeNet-5 architecture. In *2020 international conference on innovative trends in communication and computer engineering (ITCE)* (pp. 238-243). IEEE.

[3] LEITHARDT, V. (2021). Classifying garments from fashion-MNIST dataset through CNNs. *Advances in Science, Technology and Engineering Systems Journal*, *6*(1), 989-994.

[4] OpenAI. (2023). ChatGPT (GPT-3.5 model) [Large language model]. <https://chat.openai.com/chat>

[5] [**Stephanie Glen**](https://www.statisticshowto.com/contact/). "Matthews Correlation Coefficient" From [**StatisticsHowTo.com**](https://www.statisticshowto.com): Elementary Statistics for the rest of us! [**https://www.statisticshowto.com/matthews-correlation-coefficient/**](https://www.statisticshowto.com/matthews-correlation-coefficient/) **🡪 used for MCC formula**

[6] Bressler, N. [November 23, 2022]. *How to Check the Accuracy of Your Machine Learning Model.* Deepchecks. 🡪 **used for Accuracy formula**

https://deepchecks.com/how-to-check-the-accuracy-of-your-machine-learning-model/