Manraj Singh

IE 7860

Apr. 25, 2023

Dr. Ratna Chinnam

IE 7860 – Deep Learning & Attribution Modeling Report

Convolutional Neural Networks (CNNs) and Autoencoders are two types of deep learning models used for different tasks. Both models have shown remarkable performance on the Fashion-MNIST dataset, which contains grayscale images of clothing items. The dataset comprises 60,000 training images and 10,000 test images, each of size 28x28 pixels.

CNNs are primarily designed for image classification tasks. They consist of several convolutional layers, pooling layers, and fully connected layers. The convolutional layers detect local patterns in images, such as edges, textures, and shapes, by learning sets of filters. Pooling layers, usually employing max-pooling or average-pooling, down sample the feature maps, thus reducing the spatial dimensions and computational complexity of the model. Fully connected layers perform the final classification, mapping the high-level features extracted by the convolutional layers to the output classes. In the context of the Fashion-MNIST dataset, CNNs have achieved excellent classification performance, proving their capability to identify clothing items in a robust and efficient manner.

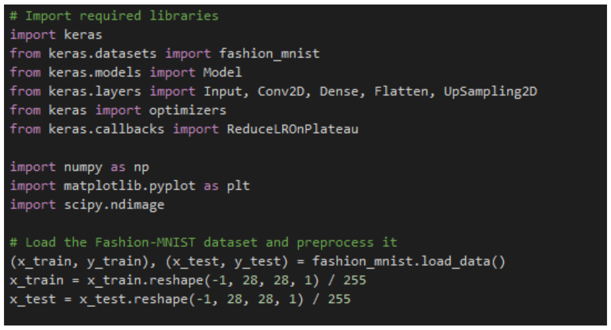
Autoencoders, on the other hand, are unsupervised learning models designed for tasks such as dimensionality reduction, feature extraction, and image denoising. They consist of two main components: an encoder and a decoder. The encoder compresses the input data, reducing its dimensions while retaining as much information as possible. The decoder then reconstructs the original data from the compressed representation. Autoencoders learn to perform this compression and reconstruction process by minimizing the reconstruction error, typically measured by the mean squared error (MSE) between the input and output images.

In the context of the Fashion-MNIST dataset, autoencoders can be used to extract low-dimensional latent representations of clothing items, which can then be used for various tasks, such as clustering, visualization, or as input to other models for classification. By using convolutional layers in both the encoder and decoder parts, autoencoders can leverage the same spatial pattern recognition capabilities that CNNs have, while learning to compress and reconstruct images efficiently.

In summary, CNNs and autoencoders serve different purposes, but both have shown impressive performance on the Fashion-MNIST dataset. CNNs excel at image classification tasks, leveraging their ability to learn spatial patterns and hierarchies of features to identify clothing items. Autoencoders, being unsupervised models, focus on learning low-dimensional latent representations of the input data, which can be used for a variety of tasks such as dimensionality reduction, feature extraction, and image denoising. By incorporating convolutional layers, both models can efficiently learn spatial patterns in images and effectively handle tasks related to the Fashion-MNIST dataset.

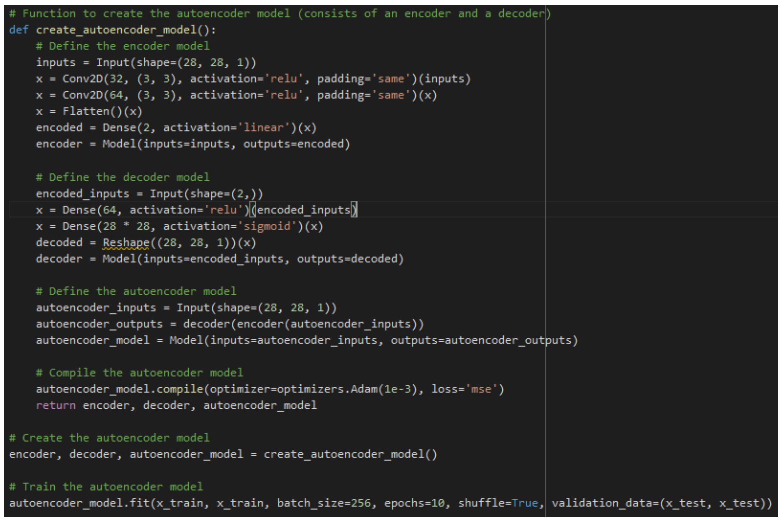
Model 1 (Autoencoder):

The dataset was imported with the commands below, and split for modeling:



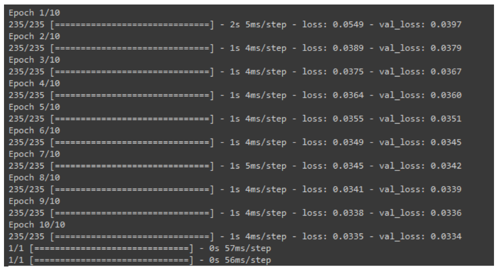
```
# Import required libraries
import keras
from keras.datasets import fashion_mnist
from keras.models import Model
from keras.layers import Input, Conv2D, Dense, Flatten, UpSampling2D
from keras import optimizers
from keras.callbacks import ReduceLROnPlateau

import numpy as np
import matplotlib.pyplot as plt
import scipy.ndimage

# Load the Fashion-MNIST dataset and preprocess it
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1) / 255
x_test = x_test.reshape(-1, 28, 28, 1) / 255
```

The model was built with a CNN with the following structure:



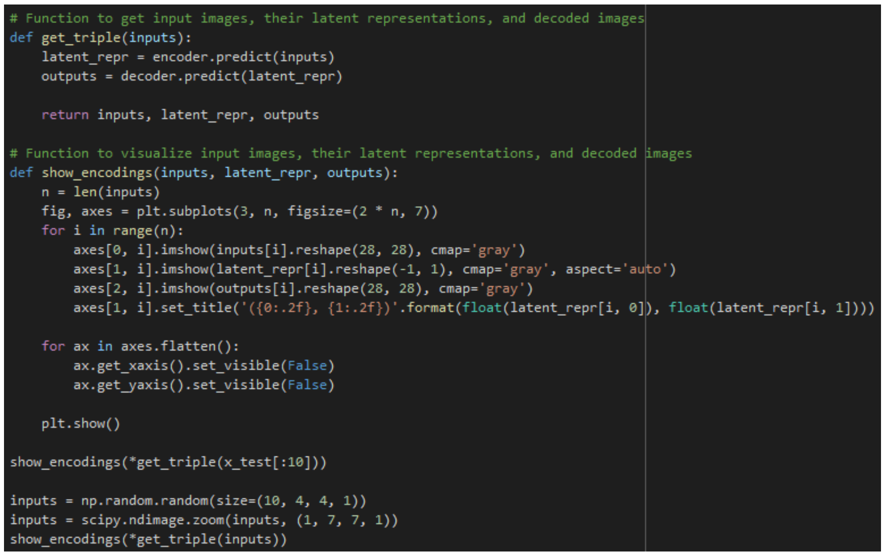
```
# Function to create the autoencoder model (consists of an encoder and a decoder)
def create_autoencoder_model():
    # Define the encoder model
    inputs = Input(shape=(28, 28, 1))
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = Flatten()(x)
    encoded = Dense(2, activation='linear')(x)
    encoder = Model(inputs=inputs, outputs=encoded)

    # Define the decoder model
    encoded_inputs = Input(shape=(2,))
    x = Dense(64, activation='relu')(encoded_inputs)
    x = Dense(28 * 28, activation='sigmoid')(x)
    decoded = Reshape((28, 28, 1))(x)
    decoder = Model(inputs=encoded_inputs, outputs=decoded)

    # Define the autoencoder model
    autoencoder_inputs = Input(shape=(28, 28, 1))
    autoencoder_outputs = decoder(encoder(autoencoder_inputs))
    autoencoder_model = Model(inputs=autoencoder_inputs, outputs=autoencoder_outputs)

    # Compile the autoencoder model
    autoencoder_model.compile(optimizer=optimizers.Adam(1e-3), loss='mse')
    return encoder, decoder, autoencoder_model

# Create the autoencoder model
encoder, decoder, autoencoder_model = create_autoencoder_model()

# Train the autoencoder model
autoencoder_model.fit(x_train, x_train, batch_size=256, epochs=10, shuffle=True, validation_data=(x_test, x_test))
```

The results of the model with the goal of a lower MSE is shown below, running for 10 epochs:



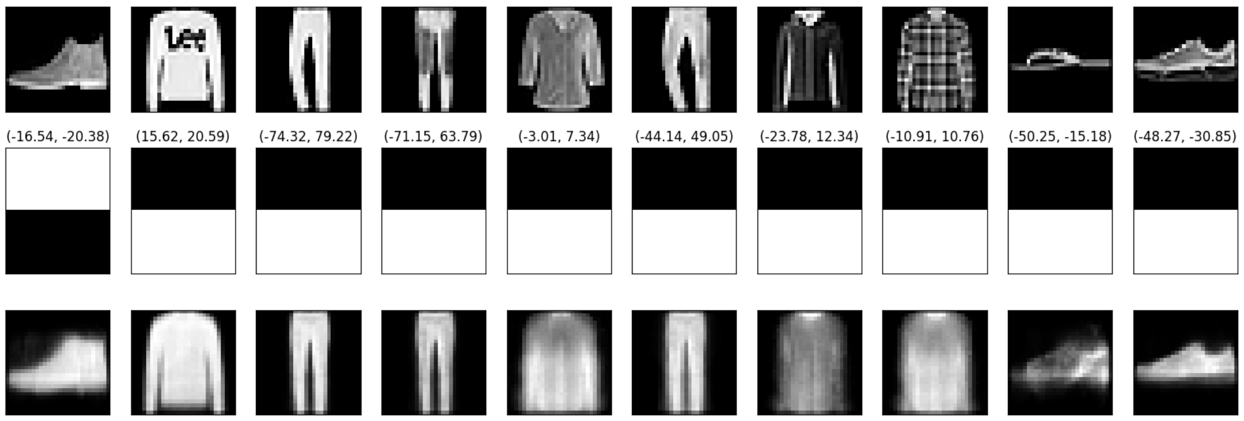
```
Epoch 1/10
235/235 [==============================] - 2s 5ms/step - loss: 0.0549 - val_loss: 0.0397
Epoch 2/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0389 - val_loss: 0.0379
Epoch 3/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0375 - val_loss: 0.0367
Epoch 4/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0364 - val_loss: 0.0360
Epoch 5/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0355 - val_loss: 0.0351
Epoch 6/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0349 - val_loss: 0.0345
Epoch 7/10
235/235 [==============================] - 1s 5ms/step - loss: 0.0345 - val_loss: 0.0342
Epoch 8/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0341 - val_loss: 0.0339
Epoch 9/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0338 - val_loss: 0.0336
Epoch 10/10
235/235 [==============================] - 1s 4ms/step - loss: 0.0335 - val_loss: 0.0334
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 56ms/step
```

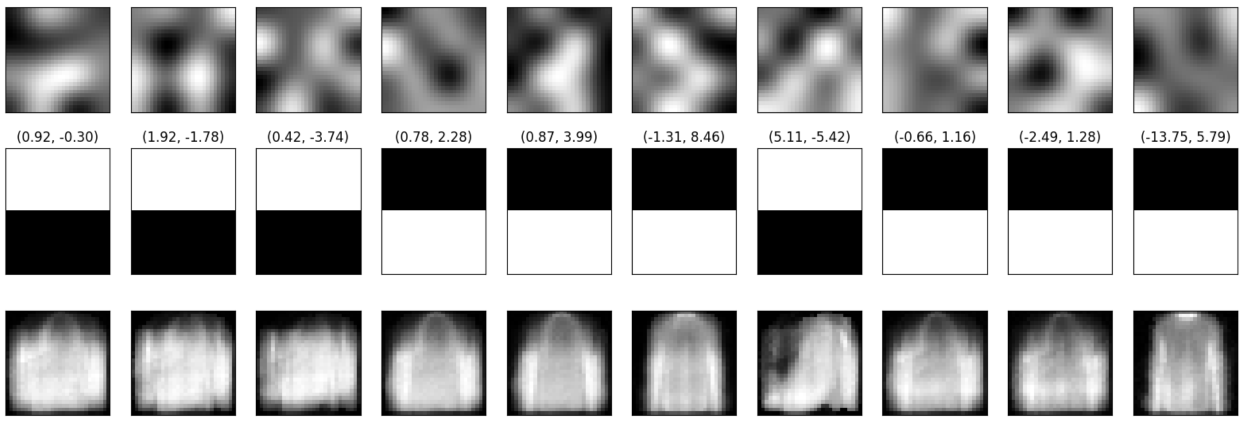
As we can see, the model was able to lower the validation loss to 0.0334, and we could then continue with the autoencoding and decoding aspect. We took the model results and built two functions, “get_triple” to get input images, their latent representations, and decoded images, and “show_encodings” to visualize the input images, their latent representations, and decoded images as shown below:



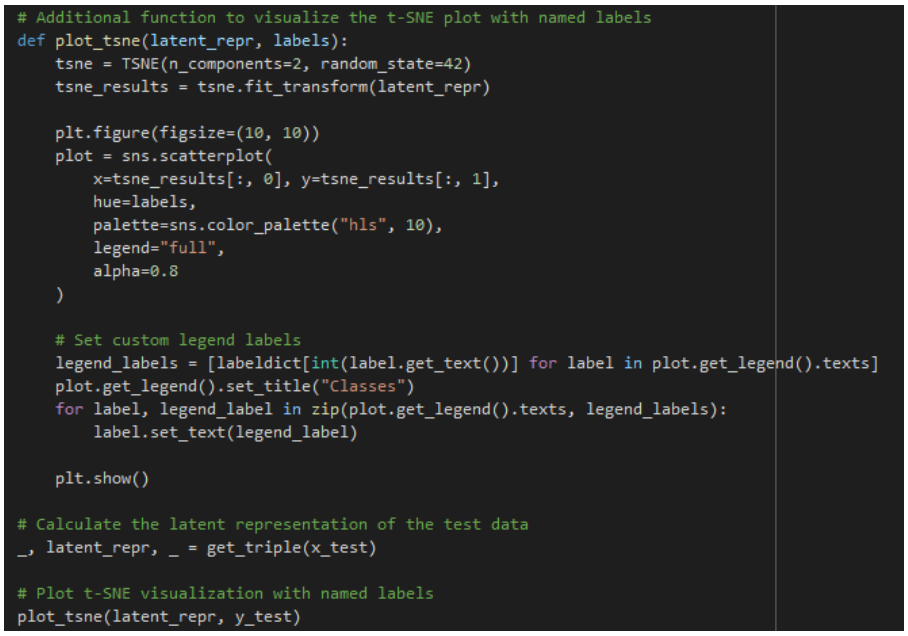
```python
# Function to get input images, their latent representations, and decoded images
def get_triple(inputs):
    latent_repr = encoder.predict(inputs)
    outputs = decoder.predict(latent_repr)

    return inputs, latent_repr, outputs

# Function to visualize input images, their latent representations, and decoded images
def show_encodings(inputs, latent_repr, outputs):
    n = len(inputs)
    fig, axes = plt.subplots(3, n, figsize=(2 * n, 7))
    for i in range(n):
        axes[0, i].imshow(inputs[i].reshape(28, 28), cmap='gray')
        axes[1, i].imshow(latent_repr[i].reshape(-1, 1), cmap='gray', aspect='auto')
        axes[2, i].imshow(outputs[i].reshape(28, 28), cmap='gray')
        axes[1, i].set_title('({0:.2f}, {1:.2f})'.format(float(latent_repr[i, 0]), float(latent_repr[i, 1])))

    for ax in axes.flatten():
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.show()

show_encodings(*get_triple(x_test[:10]))

inputs = np.random.random(size=(10, 4, 4, 1))
inputs = scipy.ndimage.zoom(inputs, (1, 7, 7, 1))
show_encodings(*get_triple(inputs))
```

The resultant image of the decoded images is pasted here:



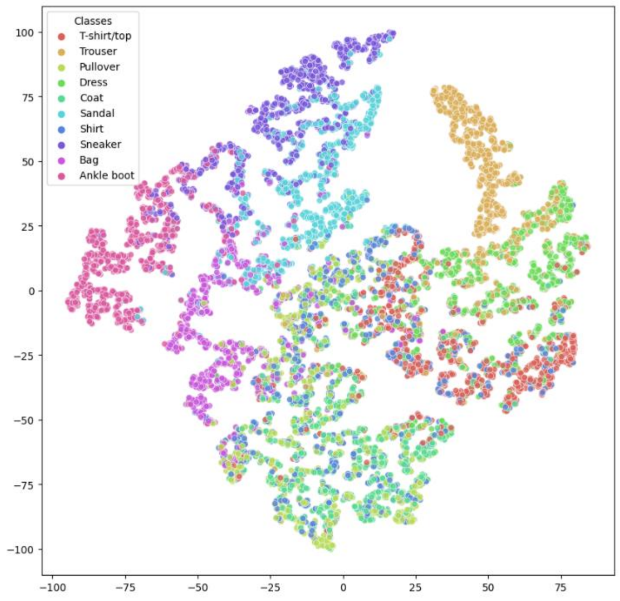


We took these functions and built a t-SNE visual representation of the clothing items:



```python
# Additional function to visualize the t-SNE plot with named labels
def plot_tsne(latent_repr, labels):
    tsne = TSNE(n_components=2, random_state=42)
    tsne_results = tsne.fit_transform(latent_repr)

    plt.figure(figsize=(10, 10))
    plot = sns.scatterplot(
        x=tsne_results[:, 0], y=tsne_results[:, 1],
        hue=labels,
        palette=sns.color_palette("hls", 10),
        legend="full",
        alpha=0.8
    )

    # Set custom legend labels
    legend_labels = [labeldict[int(label.get_text())] for label in plot.get_legend().texts]
    plot.get_legend().set_title("Classes")
    for label, legend_label in zip(plot.get_legend().texts, legend_labels):
        label.set_text(legend_label)

    plt.show()

# Calculate the latent representation of the test data
_, latent_repr, _ = get_triple(x_test)

# Plot t-SNE visualization with named labels
plot_tsne(latent_repr, y_test)
```

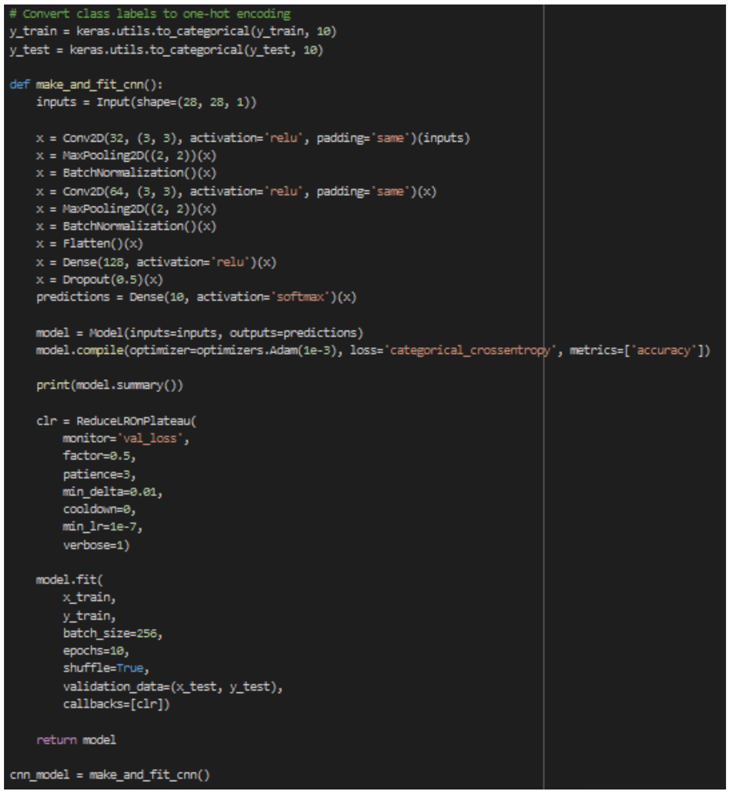
Following this, the resultant plot is shown below:



The autoencoder was useful to learn meaningful features about the input data, which can be used as input for other ML models. Thes features often capture underlying structure of the data, making it easier to allow further models to perform tasks such as classification on the fashion dataset.

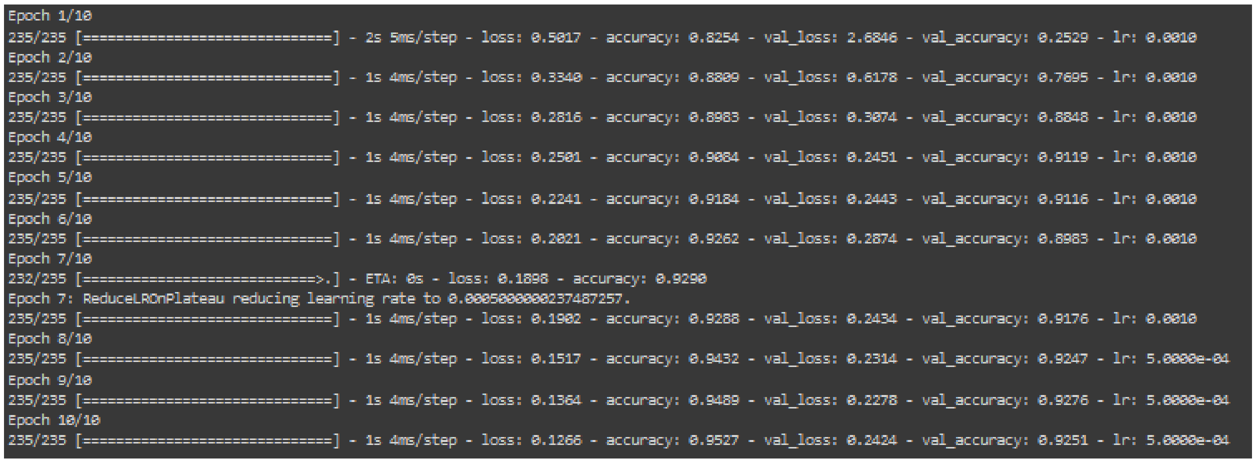
Model 2 (CNN):

The CNN model was initialized the same way as the autoencoder, importing the dataset and splitting the data simultaneously. The model was built with a focus on accuracy, and the structure of the model is shown here:



```python
# Convert class labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

def make_and_fit_cnn():
    inputs = Input(shape=(28, 28, 1))

    x = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.5)(x)
    predictions = Dense(10, activation='softmax')(x)

    model = Model(inputs=inputs, outputs=predictions)
    model.compile(optimizer=optimizers.Adam(1e-3), loss='categorical_crossentropy', metrics=['accuracy'])

    print(model.summary())

    clr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=3,
        min_delta=0.01,
        cooldown=0,
        min_lr=1e-7,
        verbose=1)

    model.fit(
        x_train,
        y_train,
        batch_size=256,
        epochs=10,
        shuffle=True,
        validation_data=(x_test, y_test),
        callbacks=[clr])

    return model

cnn_model = make_and_fit_cnn()
```

As we can see, convolutional layers, padding, batch normalization, flattening, dense layers, and dropout were all utilized to focus on improving the testing accuracy of the fashion model. The model was run for 10 epochs with an automatically adjusting learning rate, with the results shown below:

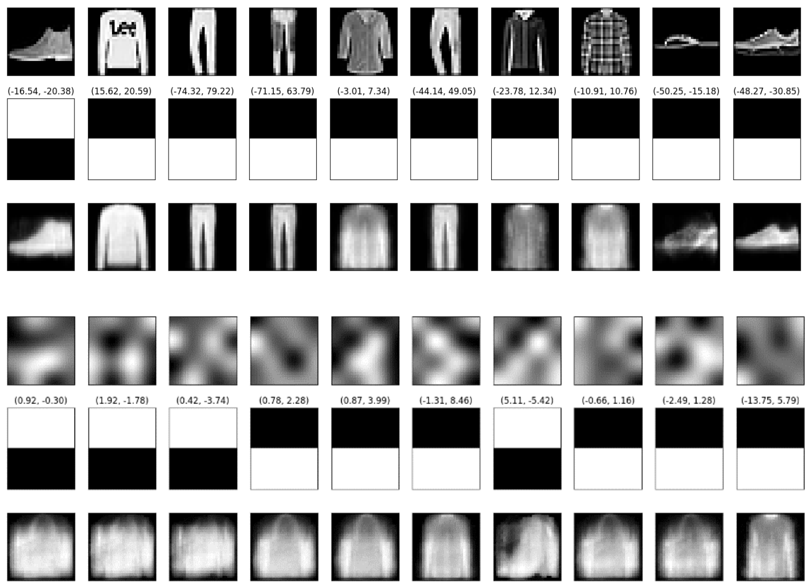


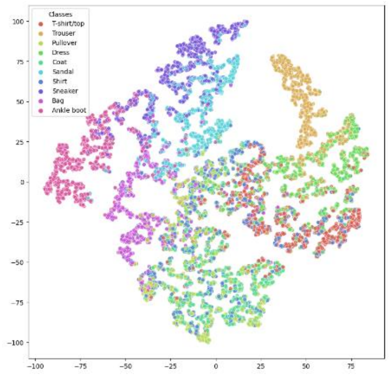
```
Epoch 1/10
235/235 [==============================] - 2s 5ms/step - loss: 0.5017 - accuracy: 0.8254 - val_loss: 2.6846 - val_accuracy: 0.2529 - lr: 0.0010
Epoch 2/10
235/235 [==============================] - 1s 4ms/step - loss: 0.3340 - accuracy: 0.8809 - val_loss: 0.6178 - val_accuracy: 0.7695 - lr: 0.0010
Epoch 3/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2816 - accuracy: 0.8983 - val_loss: 0.3074 - val_accuracy: 0.8848 - lr: 0.0010
Epoch 4/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2501 - accuracy: 0.9084 - val_loss: 0.2451 - val_accuracy: 0.9119 - lr: 0.0010
Epoch 5/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2241 - accuracy: 0.9184 - val_loss: 0.2443 - val_accuracy: 0.9116 - lr: 0.0010
Epoch 6/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2021 - accuracy: 0.9262 - val_loss: 0.2874 - val_accuracy: 0.8983 - lr: 0.0010
Epoch 7/10
232/235 [============================>.] - ETA: 0s - loss: 0.1898 - accuracy: 0.9290
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
235/235 [==============================] - 1s 4ms/step - loss: 0.1902 - accuracy: 0.9288 - val_loss: 0.2434 - val_accuracy: 0.9176 - lr: 0.0010
Epoch 8/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1517 - accuracy: 0.9432 - val_loss: 0.2314 - val_accuracy: 0.9247 - lr: 5.0000e-04
Epoch 9/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1364 - accuracy: 0.9489 - val_loss: 0.2278 - val_accuracy: 0.9276 - lr: 5.0000e-04
Epoch 10/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1266 - accuracy: 0.9527 - val_loss: 0.2424 - val_accuracy: 0.9251 - lr: 5.0000e-04
```

As we can see, the CNN model was very strong in learning quickly about the fashion dataset, with a validation accuracy of 92% in 10 epochs. CNNs are very strong with imaging data classification, as mentioned in the beginning summary.

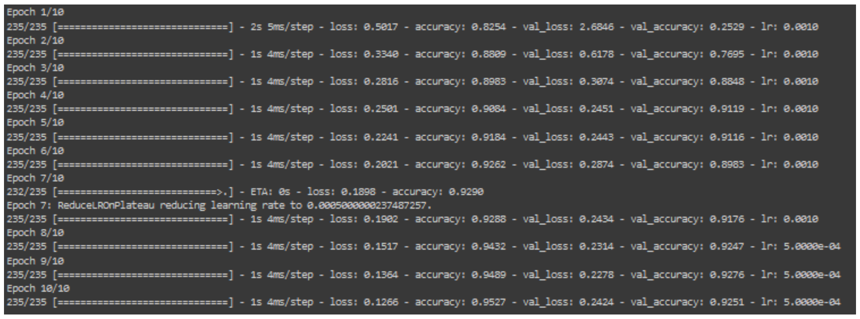
Code Outputs:

Model 1:





Model 2:


```
Epoch 1/10
235/235 [==============================] - 2s 5ms/step - loss: 0.5017 - accuracy: 0.8254 - val_loss: 2.6846 - val_accuracy: 0.2529 - lr: 0.0010
Epoch 2/10
235/235 [==============================] - 1s 4ms/step - loss: 0.3340 - accuracy: 0.8809 - val_loss: 0.6178 - val_accuracy: 0.7695 - lr: 0.0010
Epoch 3/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2816 - accuracy: 0.8983 - val_loss: 0.3074 - val_accuracy: 0.8848 - lr: 0.0010
Epoch 4/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2501 - accuracy: 0.9084 - val_loss: 0.2451 - val_accuracy: 0.9119 - lr: 0.0010
Epoch 5/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2241 - accuracy: 0.9184 - val_loss: 0.2443 - val_accuracy: 0.9116 - lr: 0.0010
Epoch 6/10
235/235 [==============================] - 1s 4ms/step - loss: 0.2021 - accuracy: 0.9262 - val_loss: 0.2874 - val_accuracy: 0.8983 - lr: 0.0010
Epoch 7/10
232/235 [============================>.] - ETA: 0s - loss: 0.1898 - accuracy: 0.9290
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
235/235 [==============================] - 1s 4ms/step - loss: 0.1902 - accuracy: 0.9288 - val_loss: 0.2434 - val_accuracy: 0.9176 - lr: 0.0010
Epoch 8/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1517 - accuracy: 0.9432 - val_loss: 0.2314 - val_accuracy: 0.9247 - lr: 5.0000e-04
Epoch 9/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1364 - accuracy: 0.9489 - val_loss: 0.2278 - val_accuracy: 0.9276 - lr: 5.0000e-04
Epoch 10/10
235/235 [==============================] - 1s 4ms/step - loss: 0.1266 - accuracy: 0.9527 - val_loss: 0.2424 - val_accuracy: 0.9251 - lr: 5.0000e-04
```