**Convolutional neural network (CNN) (Any One from the following):**

* Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

The initial block of import statements brings in necessary libraries. import tensorflow as tf imports the TensorFlow library, a cornerstone for numerical computation and machine learning, and assigns it the alias tf for brevity. from sklearn.model\_selection import train\_test\_split specifically imports the train\_test\_split function from scikit-learn, which is typically used to divide datasets into training and testing portions, although it's not directly used in this particular code snippet. from sklearn.metrics import classification\_report, confusion\_matrix imports two evaluation metrics from scikit-learn: classification\_report provides a detailed summary of classification performance, and confusion\_matrix generates a table showing the counts of true positives, true negatives, false positives, and false negatives. import pandas as pd imports the pandas library, essential for data manipulation and analysis, and gives it the alias pd. import numpy as np imports the NumPy library, fundamental for numerical operations in Python, especially for handling arrays, and is aliased as np. import seaborn as sns imports the seaborn library, built on matplotlib, offering high-level functions for creating informative statistical visualizations, aliased as sns. Lastly, import matplotlib.pyplot as plt imports the pyplot module from the matplotlib library, a comprehensive tool for creating static, interactive, and animated plots in Python.

The line (x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.fashion\_mnist.load\_data() utilizes TensorFlow Keras to load the Fashion MNIST dataset. This dataset contains grayscale images of 70,000 fashion articles, with 60,000 images for training and 10,000 for testing. The images are 28x28 pixels, and the labels are integers from 0 to 9, representing different clothing categories. The loaded data is unpacked into training images and their corresponding labels (x\_train, y\_train), and testing images and their labels (x\_test, y\_test).

(x\_train.shape, y\_train.shape), (x\_test.shape, y\_test.shape) displays the shapes of the loaded training and testing data. x\_train.shape and x\_test.shape will show the number of images and their dimensions (number of samples, height, width). y\_train.shape and y\_test.shape will show the number of labels in the respective sets.

class\_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'] defines a list named class\_names which contains the textual labels corresponding to the numerical labels (0-9) in the Fashion MNIST dataset. This list will be used later to interpret the predictions.

y\_train\_cat = tf.keras.utils.to\_categorical(y\_train) converts the training labels (y\_train), which are currently integer values, into a one-hot encoded format. One-hot encoding transforms each integer label into a binary vector where the index of the '1' corresponds to the class number. This is often necessary for training categorical classification models.

y\_test\_cat = tf.keras.utils.to\_categorical(y\_test) performs the same one-hot encoding for the testing labels (y\_test).

The subsequent block uses matplotlib to visualize the first image in the training set: plt.figure() creates a new figure for the plot. plt.imshow(x\_train[0]) displays the first image (x\_train[0]) as a 2D array of pixel intensities. By default, matplotlib will map these intensities to a color map. plt.colorbar() adds a colorbar to the plot, showing the mapping between pixel intensity values and colors. plt.grid(False) turns off the grid lines on the plot. plt.show() displays the generated image plot.

The next two lines normalize the pixel values of the training and testing images: train\_images = x\_train / 255.0 divides each pixel value in the x\_train array by 255.0. Since the pixel values in grayscale images typically range from 0 to 255, this operation scales the values to the range of 0 to 1, which can improve the training of neural networks. test\_images = x\_test / 255.0 performs the same normalization for the testing images in x\_test.

The following block visualizes the first 25 images from the normalized training set along with their true labels: plt.figure(figsize=(10,10)) creates a new figure with a specified size of 10x10 inches. for i in range(25): starts a loop that iterates 25 times to display 25 images. plt.subplot(5,5,i+1) creates a grid of 5x5 subplots and selects the (i+1)-th subplot for plotting. plt.xticks([]) and plt.yticks([]) turn off the x and y axis ticks for each subplot. plt.grid(False) turns off the grid lines for each subplot. plt.imshow(train\_images[i], cmap=plt.cm.binary) displays the i-th normalized training image in grayscale using the binary colormap. plt.xlabel(class\_names[y\_train[i]]) sets the label for each subplot to the corresponding class name obtained from the class\_names list using the original integer label y\_train[i]. plt.show() displays the grid of 25 images with their labels.

The subsequent block defines a convolutional neural network (CNN) model using TensorFlow Keras: model = tf.keras.Sequential([ ... ]) initializes a sequential model, where layers are added in a linear stack. tf.keras.layers.Input(shape=(28, 28, 1)) specifies the input shape for the first layer. It expects 28x28 grayscale images (hence the 1 for the number of color channels). tf.keras.layers.Conv2D(32, kernel\_size=(3, 3), padding='same', activation='relu') adds a 2D convolutional layer with 32 filters (output channels). The kernel\_size=(3, 3) specifies the size of the convolutional kernel, padding='same' ensures that the output has the same spatial dimensions as the input, and activation='relu' applies the Rectified Linear Unit activation function. tf.keras.layers.AvgPool2D(pool\_size=(2, 2)) adds an average pooling layer that downsamples the spatial dimensions of the input by a factor of 2 in both height and width. The next two Conv2D and AvgPool2D layers repeat a similar pattern, further extracting features and reducing dimensionality with 16 and then 8 filters, respectively. tf.keras.layers.BatchNormalization() adds a batch normalization layer, which helps to stabilize training by normalizing the activations of the previous layer. tf.keras.layers.Flatten() flattens the 2D feature maps from the previous layer into a 1D tensor, preparing the data for the dense layers. tf.keras.layers.Dense(128, activation='relu') adds a densely connected (fully connected) layer with 128 neurons and ReLU activation. tf.keras.layers.Dense(10, activation="softmax") adds the final dense layer with 10 neurons (corresponding to the 10 classes). The softmax activation function outputs a probability distribution over the 10 classes.

model.compile(loss="categorical\_crossentropy", optimizer="adam", metrics=["accuracy"]) configures the model for training. loss="categorical\_crossentropy" specifies the loss function to be minimized, which is appropriate for multi-class classification with one-hot encoded labels. optimizer="adam" selects the Adam optimization algorithm for updating the model's weights. metrics=["accuracy"] specifies that the accuracy should be calculated and reported during training and evaluation.

history = model.fit(x\_train, y\_train\_cat, epochs=10, validation\_data=(x\_test, y\_test\_cat)) trains the model. x\_train and y\_train\_cat are the training images and their one-hot encoded labels. epochs=10 specifies that the training process will iterate over the entire training dataset 10 times. validation\_data=(x\_test, y\_test\_cat) provides the test set as validation data to evaluate the model's performance on unseen data after each epoch. The training history (loss and accuracy at each epoch) is stored in the history object.

pd.DataFrame(history.history).plot(figsize=(10,7)) creates a pandas DataFrame from the history.history dictionary, which contains the training and validation metrics recorded during training, and then generates a line plot of these metrics over the epochs. figsize=(10,7) sets the size of the plot.

plt.title("Metrics Graph") sets the title of the plot to "Metrics Graph".

plt.show() displays the plot of the training and validation metrics.

model.evaluate(x\_test, y\_test\_cat) evaluates the trained model on the test data (x\_test, y\_test\_cat) and returns the loss and the specified metrics (accuracy).

predictions = model.predict(x\_test) uses the trained model to make predictions on the test images (x\_test). The output predictions will be a NumPy array of shape (number of test samples, 10), where each row contains the probability distribution over the 10 classes for a given test image.

predictions = tf.argmax(predictions, axis=1) converts the probability distributions in predictions into class labels by finding the index of the maximum probability along the axis 1 (the class axis).

y\_test = tf.argmax(y\_test\_cat, axis=1) converts the one-hot encoded test labels (y\_test\_cat) back into their original integer format by finding the index of the '1' along axis 1.

y\_test = tf.Variable(y\_test) converts the NumPy array y\_test into a TensorFlow Variable. This might be done for compatibility with subsequent TensorFlow operations, although in this context, it's not strictly necessary for classification\_report.

print(classification\_report(y\_test, predictions, zero\_division=0)) generates and prints a detailed classification report using the true labels (y\_test) and the model's predicted labels (predictions). zero\_division=0 handles cases where a class has no true or predicted samples, preventing division by zero errors.

cm = confusion\_matrix(y\_test, predictions) calculates the confusion matrix using the true labels and the predicted labels.

sns.heatmap(cm, cmap='crest', annot=True, fmt=".0f", xticklabels=class\_names, yticklabels=class\_names) creates a heatmap visualization of the confusion matrix using seaborn. cm is the confusion matrix data, cmap='crest' sets the color map, annot=True displays the values in each cell, fmt=".0f" formats the annotations as integers, and xticklabels and yticklabels provide the class names for the x and y axes.

plt.title("Confusion Matrix") sets the title of the heatmap plot.

plt.show() displays the confusion matrix heatmap.

The final block of code selects and visualizes 10 random test images along with their true and predicted labels: import random imports the random module for generating random numbers. images = [] and labels = [] initialize empty lists to store the selected images and their one-hot encoded labels. random\_indices = random.sample(range(len(x\_test)), 10) randomly selects 10 unique indices from the range of the number of test samples. The for loop iterates through the random\_indices, appending the corresponding test image (x\_test[idx]) and its one-hot encoded label (y\_test\_cat[idx]) to the images and labels lists. images = np.array(images) and labels = np.array(labels) convert the lists into NumPy arrays. fig = plt.figure(figsize=(20, 8)) creates a new figure with a specified size. rows = 2 and cols = 5 define the layout of the subplots as a 2x5 grid. x = 1 initializes a counter for the subplot index. The for loop iterates through the selected images and labels. fig.add\_subplot(rows, cols, x) adds a subplot at the x-th position in the grid. prediction = model.predict(tf.expand\_dims(image, axis=0)) makes a prediction for the current image. tf.expand\_dims(image, axis=0) adds a batch dimension to the single image, as the model expects input in batches. prediction = class\_names[tf.argmax(prediction.flatten())] gets the predicted class name by finding the index of the maximum probability in the flattened prediction array. label = class\_names[tf.argmax(label)] gets the true class name from the one-hot encoded label. plt.title(f"Label: {label}, Prediction: {prediction}") sets the title of the subplot to show both the true label and the model's prediction. plt.imshow(image/255.) displays the normalized image in the subplot. plt.axis("off") turns off the axis ticks and labels for the subplot. x += 1 increments the subplot counter. Finally, plt.show() displays the figure containing the 10 randomly selected test images with their true and predicted labels.