ChatGPT

Fashion MNIST CNN Code Walkthrough

Imports and Libraries

The script begins by importing necessary Python libraries. For example: - `pandas` **(**`pd`**)** – for data handling. Pandas can read CSV files into DataFrame objects, which makes it easy to inspect and manipulate tabular data.
- `numpy` **(**`np`**)** – for numerical array operations. Keras and many data operations use NumPy arrays under the hood.
- `matplotlib.pyplot` **(**`plt`**) and** `seaborn` **(**`sns`**)** – for plotting. Matplotlib can display images and charts; Seaborn provides nicer styles, often used here for the confusion matrix heatmap.
- `tensorflow.keras` **layers** (e.g. `Conv2D`, `MaxPooling2D`, `Flatten`, `Dense`, `Dropout`) – these are building blocks of a convolutional neural network. `Conv2D` adds convolutional layers, `MaxPooling2D` downsamples feature maps, `Flatten` reshapes data for fully-connected layers, `Dense` creates fully-connected layers, and `Dropout` is a regularization layer.
- `to_categorical` from Keras utilities – to convert integer labels into one-hot encoded vectors for multiclass classification.
- `sklearn.metrics` functions (`classification_report`, `confusion_matrix`) – to compute and display performance metrics after training.

Each import serves a clear purpose: data I/O and manipulation (pandas/NumPy), visualization (Matplotlib/Seaborn), model building (Keras layers), and model evaluation (scikit-learn metrics).

Loading and Inspecting Data

Next, the Fashion MNIST data (often in CSV format from sources like Kaggle) is loaded into pandas DataFrames:

```
train_df = pd.read_csv('fashion-mnist_train.csv')
test_df  = pd.read_csv('fashion-mnist_test.csv')
```

This creates `train_df` and `test_df` tables. The code often uses `train_df.head()` to display the first few rows of the DataFrame. The `.head()` method returns the first *n* rows (default 5), which is useful for checking that the data loaded correctly [1]. For example, `train_df.head()` will show the first 5 rows, including the `label` column and pixel columns.

Likewise, calling `train_df['label'].unique()` shows the distinct label values in the `label` column. The `unique()` function returns an array of the unique entries in that column [2]. For Fashion MNIST, you should see all integers 0–9, indicating 10 classes. This quick inspection ensures all expected classes are present and helps catch any loading issues.

Visualizing Sample Images

The Fashion MNIST dataset contains 28×28 pixel grayscale images of clothing items [3]. To verify the data looks correct, we often plot example images. The code might reshape the pixel values in a row into a 28×28 array and use Matplotlib to display it. For instance, one can loop over each class label, select a row with that label, reshape its 784 pixel values (`row[1:]`) into a 2D array, and show it with `plt.imshow(image, cmap='gray')`.

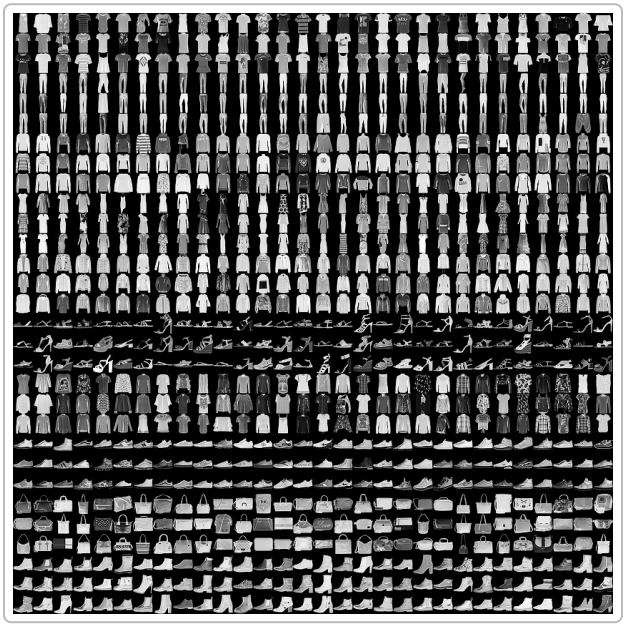

Figure: Sample images from the Fashion MNIST dataset (each small square is a 28×28 grayscale clothing item). These examples illustrate the 10 categories.

The figure above (a "sprite" of many Fashion-MNIST examples) shows how each row in our DataFrame corresponds to a small grayscale image. In code, `image = train_df.iloc[idx, 1:].values.reshape(28,28)` converts one row of 784 pixel values into a 28×28 image. Using `plt.imshow(image, cmap='gray')` then displays it. Plotting one example for each label (0–9) helps confirm that, e.g., label 0 indeed looks like a T-shirt/top, label 1 like a trouser, etc. This step is just for human verification of the data.

Mapping Numeric Labels to Class Names

Fashion MNIST labels are integers (0–9), each representing a clothing category. For clarity, the script defines a list of class names in the same order. According to the dataset documentation, the mapping is:
- 0 = T-shirt/top
- 1 = Trouser
- 2 = Pullover
- 3 = Dress
- 4 = Coat
- 5 = Sandal

- 6 = Shirt
- 7 = Sneaker
- 8 = Bag
- 9 = Ankle boot [4].

For example, the code might include:

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

This list ensures that when we have a numeric label `y`, `class_names[y]` gives the human-readable category. We use `class_names` to add titles to plots or labels in the final report. The list above is defined exactly as in the official TensorFlow example [5], ensuring consistency.

Preprocessing the Data

Before training, we prepare the data for the CNN. The main steps are:

- **Separate features and labels:** We split each DataFrame into input pixels (`X`) and target labels (`y`). For example:

  ```
  X_train = train_df.iloc[:, 1:].values  # all rows, columns 1 onward (pixel values)
  y_train = train_df['label'].values    # all labels
  ```

  Here, `X_train` becomes a 2D NumPy array of shape (num_samples, 784), and `y_train` is a 1D array of labels. The first column (index 0) was the label, so we take columns 1 onward for pixels.

- **Normalization (scaling pixel values):** Pixel values range from 0 to 255 (grayscale intensity). Neural networks train faster and more stably if input values are in a smaller range, usually 0–1. We scale by dividing by 255.0:

  ```
  X_train = X_train / 255.0
  X_test  = X_test / 255.0
  ```

  This converts all pixel values to floats between 0 and 1 [6]. It's important to apply the same scaling to both training and test sets.

- **Reshape data for the CNN:** Convolutional layers expect 4D input: `(batch_size, height, width, channels)`. Since our images are 28×28 with 1 color channel, we reshape:

```
X_train = X_train.reshape(-1, 28, 28, 1)
X_test  = X_test.reshape(-1, 28, 28, 1)
```

Here `-1` infers the number of samples automatically. After this, `X_train.shape` might be `(60000, 28, 28, 1)`, meaning 60,000 images of size 28×28 with 1 channel.

- **One-hot encode labels:** The labels `y_train` are integers 0–9, but for training with `categorical_crossentropy` loss we convert them to one-hot vectors. Using Keras:

```
y_train = to_categorical(y_train, num_classes=10)
y_test  = to_categorical(y_test, num_classes=10)
```

Now each label becomes a length-10 binary vector (e.g., label 3 → `[0,0,0,1,0,0,0,0,0,0]`). One-hot encoding matches our output layer's softmax, which predicts a probability for each class.

These preprocessing steps ensure the data is in the correct shape and scale for the CNN to learn effectively.

CNN Model Architecture

The core of the script is the convolutional neural network. A typical architecture might look like this:

- **Conv2D Layer:** The first layer could be `Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1))`. This applies 32 different 3×3 filters to the input image, producing 32 feature maps. Each filter "slides" over the image and learns to detect patterns like edges or textures [7]. The `activation='relu'` adds non-linearity (Rectified Linear Units), helping the network learn complex features. The parameter `filters=32` means we have 32 distinct kernels extracting different features (depth of output is 32) [7].

- **MaxPooling2D Layer:** Next, a pooling layer (e.g., `MaxPooling2D(pool_size=(2,2))`) downsamples each feature map. It takes each 2×2 block and outputs its maximum value. This halves the width and height of the feature maps [8], reducing computation and helping the network focus on the most prominent features. For instance, a 2×2 max-pool on a 26×26 map yields a 13×13 map.

- **Additional Conv/Pool Layers (optional):** Often we stack another `Conv2D` + `MaxPooling2D` pair with more filters (e.g., 64 filters of size 3×3), to learn higher-level features. Each added convolution + pooling makes the network deeper and its receptive field larger.

- **Flatten Layer:** After the convolutions, we use `Flatten()` to convert the final set of feature maps into a single 1D vector [9]. Flattening does not change the batch size; it just rearranges the data so we can feed it into Dense layers. For example, if the last pooled output was shape (None, 7, 7, 64), `Flatten()` turns it into (None, 7764).

- **Dense (Fully-Connected) Layers:** We then add one or more `Dense` layers. A typical choice is `Dense(128, activation='relu')`, meaning 128 neurons fully connected to the flattened inputs. This layer learns to combine the features extracted by the convolutions into class-relevant signals.

- **Dropout (Regularization):** To prevent overfitting, a `Dropout(0.5)` layer may be inserted after a Dense layer. During training, Dropout randomly sets 50% of the inputs to zero at each update [10]. This "dropping out" of nodes forces the network to not rely too heavily on any single neuron, which improves generalization [10].

- **Output Layer:** Finally, a `Dense(10, activation='softmax')` layer produces the class scores. There are 10 units (one per clothing category), and softmax turns the raw outputs (logits) into probabilities that sum to 1. Softmax is appropriate here because we have multiple classes and want a probability distribution over them [11]. For instance, if the network is 80% sure an image is class 7 (Sneaker) and 20% class 5 (Sandal), softmax will reflect that.

Each layer is added to the model in sequence, for example using a `Sequential` model in Keras:

```
model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D((2,2)))
# (add more Conv/Pool as needed)
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

In summary, convolutional layers detect patterns (using multiple filters) [7], pooling layers reduce spatial size (e.g. 2×2 max pooling shrinks images by half) [8], `Flatten` prepares data for Dense layers [9], and the final softmax layer outputs class probabilities [11]. Dropout layers improve robustness by randomly dropping nodes during training [10].

Compiling the Model

After defining the architecture, the model is compiled with a loss function, optimizer, and metrics. For example:

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- **Optimizer (Adam):** Adam is a popular stochastic gradient descent method that adapts the learning rate for each parameter. It generally works well out-of-the-box for many problems.
- **Loss function (`categorical_crossentropy`):** Since this is multiclass classification with one-hot labels,

categorical cross-entropy measures the difference between the predicted class probability distribution and the true distribution. Minimizing this loss trains the model to predict the correct class.
- **Metrics (`accuracy`):** We track accuracy (percentage of correct predictions) on the training and validation sets during training to monitor performance.

No additional citations are needed here, but it's important to choose these because they match a multi-class softmax model with one-hot labels.

Training the Model

The model is trained using the `fit()` method. For example:

```
history = model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

This runs training for 10 epochs (full passes through the training data), with a batch size of 64 samples per gradient update. `validation_split=0.2` means 20% of the training data is held out as a validation set to monitor performance. During each epoch, the model prints the training loss and accuracy, as well as the validation loss and accuracy. This helps us see if the model is learning (training accuracy goes up) and whether it might be overfitting (if validation accuracy lags or drops).

Training may take some time, depending on hardware. Often, one would look at the printed output or a plotted learning curve to ensure the model is converging (loss decreasing, accuracy increasing). Early stopping or more epochs can be used if needed, but in a simple script we might just run a fixed number of epochs.

Evaluating the Model

After training, we evaluate performance on the test set to get an unbiased estimate. This is done with:

```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
```

The `evaluate()` function returns the loss and any metrics (accuracy) on the test data. For example, it might print `loss: 0.30 - accuracy: 0.90`, indicating 90% accuracy on test data. This tells us how well the model generalizes to unseen examples.

Making Predictions

To see individual predictions, we use `model.predict()` on the test data. This returns an array of shape (num_samples, 10), where each row is the predicted probability distribution over the 10 classes for that sample. For example:

```
pred_probs = model.predict(X_test)       # shape (10000, 10)
y_pred = np.argmax(pred_probs, axis=1)  # shape (10000,)
y_true = np.argmax(y_test, axis=1)      # convert one-hot back to integer labels
```

Here, `np.argmax(pred_probs, axis=1)` picks the index of the highest probability in each row, giving the predicted class label (0–9) for each test image. We compare this `y_pred` array to the true labels `y_true`. (If `y_test` is still one-hot, we convert it back with `argmax`; otherwise if we kept original integer labels, we compare directly.)

Using `argmax` is standard because the softmax output gives probabilities, and the largest probability is the model's predicted class.

Reporting Performance

Finally, we assess and report detailed performance. Two common tools are:

- **Classification Report:** Using `sklearn.metrics.classification_report`, we can print precision, recall, and F1-score for each class. For example:

  ```
  from sklearn.metrics import classification_report
  print(classification_report(y_true, y_pred, target_names=class_names))
  ```

  This outputs a table showing for each class the precision (how many predicted that class were correct), recall (how many true instances of that class were found), and F1-score (harmonic mean of precision and recall), as well as support (the number of true instances). It also shows averages (macro, weighted) and overall accuracy. In scikit-learn's docs, a classification report is described as a "text summary of the precision, recall, F1 score for each class" [12]. For example, you might see that "Sneaker" has precision 0.92, recall 0.95, etc. This report helps identify if some classes are being predicted much worse than others.

- **Confusion Matrix Heatmap:** The confusion matrix shows how often each true class is predicted as each class. Compute it with `sklearn.metrics.confusion_matrix(y_true, y_pred)`, which returns a 10×10 array. To visualize it, one can use Seaborn:

  ```
  import seaborn as sns
  cm = confusion_matrix(y_true, y_pred)
  sns.heatmap(cm, annot=True, fmt='d',
              xticklabels=class_names, yticklabels=class_names, cmap='Blues')
  plt.xlabel('Predicted')
  plt.ylabel('True')
  plt.title('Confusion Matrix')
  plt.show()
  ```

This heatmap makes it easy to see, for example, if “Coat” images are often mistaken for “Pullover” (off-diagonal cells). Ideally, the highest values are along the diagonal (correct predictions). Normalizing the matrix or adding labels can also be done to improve readability.

Using these tools, the script reports overall accuracy as well as detailed per-class performance. The combination of the classification report and confusion matrix gives a comprehensive view: we see exact numbers (precision/recall/F1 [12]) and a visual map of where misclassifications occur. This completes the evaluation of the CNN on Fashion MNIST.

Key Concepts: *Convolutional Neural Network (CNN)* – a deep learning model especially effective for images, using convolution and pooling layers to automatically learn features [7]. *One-hot encoding* – turning categorical labels into binary vectors (needed for categorical cross-entropy loss). *Normalization* – scaling inputs (like pixels) to [0,1] so the model trains better [6]. *Softmax* – the final activation producing a probability distribution over classes [11]. *Dropout* – a regularization technique that “drops” nodes randomly during training to reduce overfitting [10]. All these steps together form a standard workflow for training an image classifier on the Fashion MNIST dataset.

---

1 **pandas.DataFrame.head — pandas 2.2.3 documentation**
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.head.html

2 **pandas.Series.unique — pandas 2.2.3 documentation**
https://pandas.pydata.org/docs/reference/api/pandas.Series.unique.html

3 4 5 6 **Basic classification: Classify images of clothing | TensorFlow Core**
https://www.tensorflow.org/tutorials/keras/classification

7 8 **ML Practicum: Image Classification | Machine Learning | Google for Developers**
https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks

9 **Flatten layer**
https://keras.io/api/layers/reshaping_layers/flatten/

10 **A Gentle Introduction to Dropout for Regularizing Deep Neural Networks - MachineLearningMastery.com**
https://www.machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

11 **Softmax function - Wikipedia**
https://en.wikipedia.org/wiki/Softmax_function

12 **classification_report — scikit-learn 1.6.1 documentation**
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html