Day 5 - Solve a Classification Problem using Multi-layer Perceptron

Tags

Building an Image Classifier using the Sequential API

Author: **Chandan Kumar**

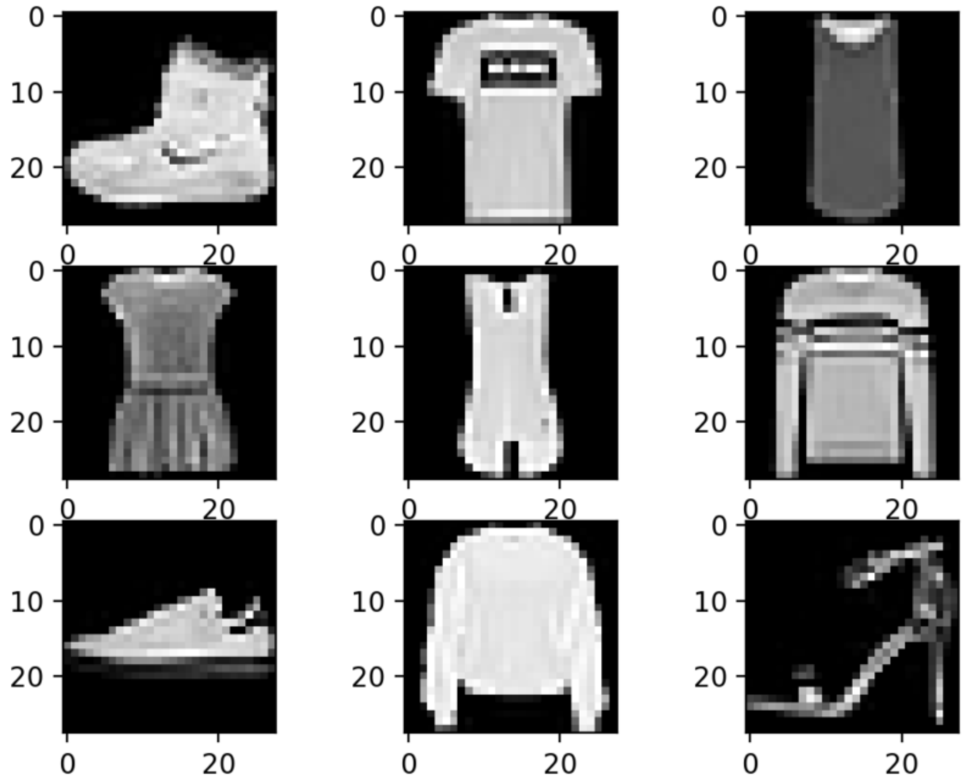
`enchandan.com`

Building an Image Classifier using the Sequential API
- Dataset
- Build the Network (Model)
- Some helper functions & attributes
- Compile the Model
- Train the Model
- Evaluate the Model
- Make Predictions

Dataset

```
'''
Docs:
https://www.tensorflow.org/api_docs/python/tf/keras/datasets/fashion_mnist/load_data

Source code:
https://github.com/tensorflow/tensorflow/blob/v2.3.1/tensorflow/python/keras/dataset
s/fashion_mnist.py#L30-L91
'''
def load_data():
  """Loads the Fashion-MNIST dataset.
  This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories,
  along with a test set of 10,000 images. This dataset can be used as
  a drop-in replacement for MNIST. The class labels are:
  | Label | Description |
  |:-----:|-------------|
  |   0   | T-shirt/top |
  |   1   | Trouser     |
  |   2   | Pullover    |
  |   3   | Dress       |
  |   4   | Coat        |
  |   5   | Sandal      |
  |   6   | Shirt       |
  |   7   | Sneaker     |
  |   8   | Bag         |
  |   9   | Ankle boot  |
  Returns:
      Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
      X: features
      y: labels
```



```
import tensorflow as tf
from tensorflow import keras

fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

print(X_train_full.shape)
print(y_train_full.shape)
print(X_test.shape)
print(y_test.shape)

print(X_train_full.dtype)

>>>
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
uint8
```

- Dataset has 28×28 array with integer values for pixels - ranging from 0-255
- We have Training and test datset. Need to also create validation dataset.
- Since we will train the neural network using Gradient descent, it is important to scale the values (of pixels).
- We'll scale the pixel-intensitites down to 0-1 range by dividing them by 255.0 (this also converts them to floats).

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

print(X_train.shape)
print(X_valid.shape)

print(y_train.shape)
print(y_valid.shape)

>>>
(55000, 28, 28)
(5000, 28, 28)
(55000,)
(5000,)
```

```
X_train[0][4] # values are scaled now

>>>
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.00784314, 0.        , 0.        , 0.0627451 , 0.82352941,
       0.88235294, 0.84313725, 0.68627451, 0.85098039, 0.84705882,
       0.75686275, 0.76862745, 0.88627451, 0.86666667, 0.81960784,
       0.19607843, 0.        , 0.        , 0.00784314, 0.        ,
       0.        , 0.        , 0.        ])
```

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

Build the Network (Model)

- Create the model using Sequential API

```
model = keras.models.Sequential()
'''
- Simplest kind of keras model for neural networks
- Composed of a single stack of layers - connected sequentially - This is called
  Sequential API
- https://github.com/tensorflow/tensorflow/blob/v2.3.1/tensorflow/python/keras/engin
e/sequential.py
'''

model.add(keras.layers.Flatten(input_shape=[28, 28]))
'''
- Built the first layer, add it to model
- It's role is to convert each input image to 1D array
  - If it receives input data X, it computes X.reshape(-1, 1)
- It is just there to do some simple preprocessing - takes shape of input instance
- Alternatively, we can use `keras.layers.InputLayer` as the first layer
'''

model.add(keras.layers.Dense(300, activation='relu'))
'''
- Next, add Dense hidden layer with 300 neurons.
- It uses ReLU activation fn.
- Each Dense layer manages it's own weight matrix
- contains all the connection weights between the neurons and their inputs
- also manages a vector of bias terms (one per neuron)
- when it receives some input data, it computes
'''
model.add(keras.layers.Dense(100, activation='relu'))
'''
- Then add another dense layer with 100 neurons and also using the RelU activation fn
'''

model.add(keras.layers.Dense(10, activation='softmax'))
'''
- Finally, add the dense layer with 10 neurons (one per class) with Softmax activatio
n fn (because classes are exclusive)

Specifying 'relu' is equal to specifying activation='keras.activations.relu'.
Other activation fns are availabe at keras.activations package - https://keras.io/act
ivations
'''
```

```
'''
Alternatively, network could have been build with the following code:
'''
```

```
model = keras.models.Sequential([
  keras.layers.Flatten(input_shape=[28, 28]),
  keras.layers.Dense(300, activation='relu'),
  keras.layers.Dense(100, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
```

```
model.summary()

>>>
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

- 28×28 = 784 input neurons required in Input Layer
- Later, number of layers and their number of neurons can be decided as per our architecture
- Dense layer is notorious for it's addition of a lot of parameters
- The first hidden layer has 784 x 300 connection weights plus 300 bias terms, which adds up to 2,35,500 parameters!
- This gives model flexibility to fit the training data, but also comes with risks of overfitting, esp. if we have less training data

Some helper functions & attributes

```
model.layers

>>>
[<tensorflow.python.keras.layers.core.Flatten at 0x7ff8369e04a8>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff8369e0d30>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff8369e1160>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff8369e14a8>]


hidden1 = model.layers[1]
print(hidden1.name)

>>>
dense


weights, biases = hidden1.get_weights()
print('Shape of weights: ', weights.shape)
print('Shape of biases:', biases.shape)

>>>
Shape of weights: (784, 300)
Shape of biases: (300,)
```

- Dense layer initialized the connection weights randomly (required to break the symmetry)
- biases were initialized to 0
- To have a different initialization method - use parameter `kernel_initializer` (kernel - another name for the matrix of connection weights) and `bias_initializer` for biases
  Refer → https://keras.io/initializers/
- `get_weights()`
  Returns the current weights of the layer.

  The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of Numpy arrays, which can in turn be used to load state into similarly parameterized layers.

Compile the Model

- After model is created, call `compile()` method to specify the loss function and the optimizer to use.
- Additionally, extra metrics to be computed during training and evaluation can be added

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd', metrics=['accu
racy'])
```

- `model.compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None, weighted_metrics=None, run_eagerly=None, **kwargs)`
  Configures the model for training.
- Equivalent options:
  - `sparse_categorical_crossentropy` ⇔ `keras.losses.sparse_categorical_crossentropy`
  - `sgd` ⇔ `keras.optimizers.SGD()`
  - `metrics['accuracy']` ⇔ `metrics=[keras.metrics.sparse_categorical_accuracy]`
  - Check keras.io/losses, keras.io/optimizers, keras.io/metrics
- `sparse_categorical_crossentropy` vs `categorical_crossentropy`
  https://stackoverflow.com/questions/58565394/what-is-the-difference-between-sparse-categorical-crossentropy-and-categorical-c

> 💡 In short, use `sparse_categorical_crossentropy` when your classes are mutually exclusive, i.e. you don't care at all about other close enough predictions.

- To convert sparse labels (i.e. class indices) to one-hot vector labels, use `keras.utils.to_categorical()` function.
  To go the other way round, use the `np.argmax()` method with axis=1.

Train the Model

```
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid))

>>>
Epoch 1/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.7231 - accuracy:
 0.7626 - val_loss: 0.5167 - val_accuracy: 0.8226
Epoch 2/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4893 - accuracy:
 0.8290 - val_loss: 0.4558 - val_accuracy: 0.8436
Epoch 3/30
1719/1719 [==============================] - 4s 3ms/step - loss: 0.4439 - accuracy:
 0.8449 - val_loss: 0.4233 - val_accuracy: 0.8534
Epoch 4/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4181 - accuracy:
 0.8525 - val_loss: 0.4149 - val_accuracy: 0.8584
.
.
.
Epoch 30/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.2275 - accuracy:
 0.9187 - val_loss: 0.3019 - val_accuracy: 0.8922
```

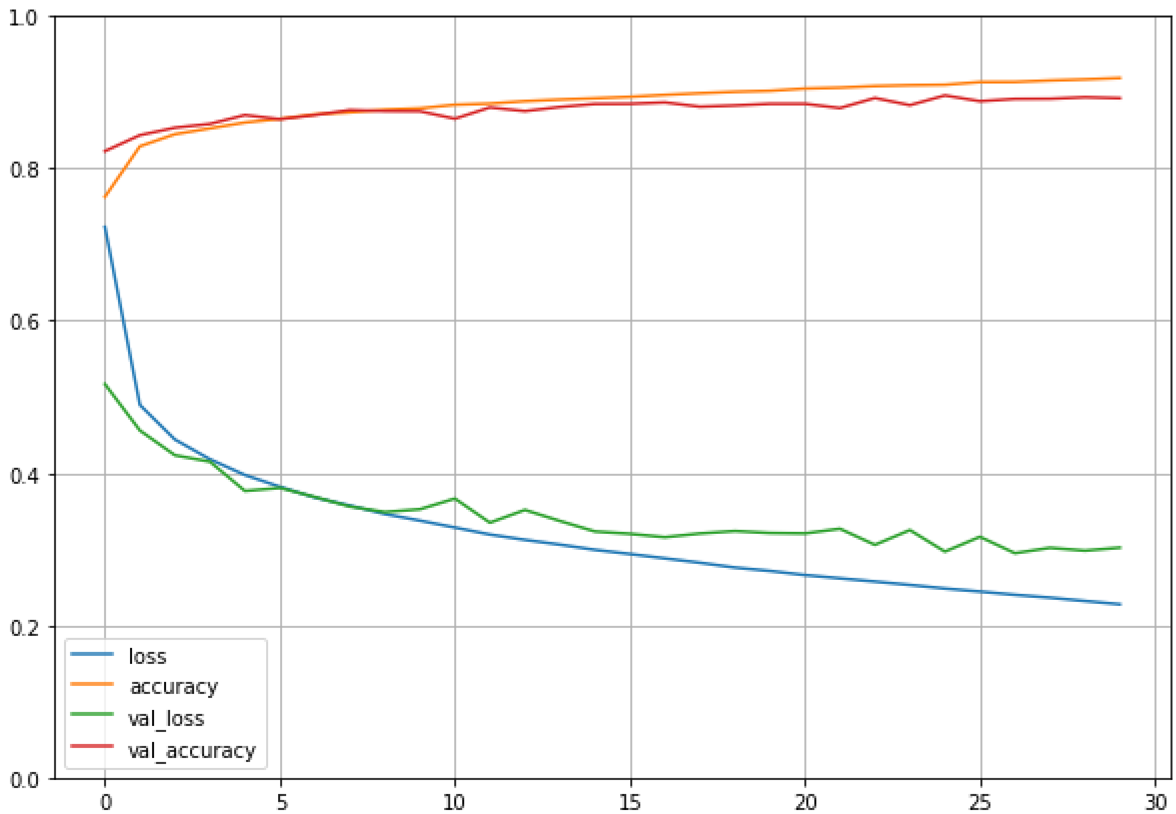
- `fit()` method sets the default value of `batch_size` to 32, that's why 55000 / 32 = 1719 above

```
history.history.keys()

>>>
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(10, 7))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```



- validation curves are close to training curves which means model is not overfitting
- x-axis represents 30 epochs
- the mean training loss and accuracy measured over each epoch
- the mean validation loss and accuracy measured at the end of each epoch
- If not satisfied with the model's performance. Following hyperparameter tuning can be done:
    - tweak Learning rate
    - try another optimizer (and always retune learning rate after changing any hyperparamter)
    - try changing number of layers
    - number of neurons per layer
    - batchsize while training & many more

Evaluate the Model

```
model.evaluate(X_test, y_test)

>>>
[66.51264953613281, 0.8410000205039978]
```

Make Predictions

```
X_new = X_test[9:12]
y_proba = model.predict(X_new)
y_proba.round(4)
```

- For each instance, model estimates one probability per class (from class 0-9)

```
y_pred = model.predict_classes(X_new)
y_pred

>>>
array([7, 4, 5])


import numpy as np
np.array(class_names)[y_pred]

>>>
array(['Sneaker', 'Coat', 'Sandal'], dtype='<U11')
```