C2_W2_Lab_2_training-categorical

February 10, 2021

1 Fashion MNIST using Custom Training Loop

In this ungraded lab, you will build a custom training loop including a validation loop so as to train a model on the Fashion MNIST dataset.

1.1 Imports

[1]:
```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import itertools
from tqdm import tqdm
import tensorflow_datasets as tfds
import matplotlib.ticker as mticker
```

1.2 Load and Preprocess Data

You will load the Fashion MNIST dataset using Tensorflow Datasets. This dataset has 28 x 28 grayscale images of articles of clothing belonging to 10 clases.

Here you are going to use the training and testing splits of the data. Testing split will be used for validation.

[2]:
```
train_data, info = tfds.load("fashion_mnist", split = "train", with_info =␣
↪True, data_dir='./data/', download=False)
test_data = tfds.load("fashion_mnist", split = "test", data_dir='./data/',␣
↪download=False)
```

[3]:
```
class_names = ["T-shirt/top", "Trouser/pants", "Pullover shirt", "Dress",␣
↪"Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Next, you normalize the images by dividing them by 255.0 so as to make the pixels fall in the range (0, 1). You also reshape the data so as to flatten the 28 x 28 pixel array into a flattened 784 pixel array.

[4]:
```
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]
```

[5]:
```
train_data = train_data.map(format_image)
test_data = test_data.map(format_image)
```

Now you shuffle and batch your training and test datasets before feeding them to the model.

[6]:
```
batch_size = 64
train = train_data.shuffle(buffer_size=1024).batch(batch_size)

test =  test_data.batch(batch_size=batch_size)
```

1.3 Define the Model

You are using a simple model in this example. You use Keras Functional API to connect two dense layers. The final layer is a softmax that outputs one of the 10 classes since this is a multi class classification problem.

[7]:
```
def base_model():
  inputs = tf.keras.Input(shape=(784,), name='digits')
  x = tf.keras.layers.Dense(64, activation='relu', name='dense_1')(inputs)
  x = tf.keras.layers.Dense(64, activation='relu', name='dense_2')(x)
  outputs = tf.keras.layers.Dense(10, activation='softmax',␣
↪name='predictions')(x)
  model = tf.keras.Model(inputs=inputs, outputs=outputs)
  return model
```

1.4 Define Optimizer and Loss Function

You have chosen `adam` optimizer and sparse categorical crossentropy loss for this example.

[8]:
```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
```

1.5 Define Metrics

You will also define metrics so that your training loop can update and display them. Here you are using `SparseCategoricalAccuracy`defined in `tf.keras.metrics` since the problem at hand is a multi class classification problem.

[9]:
```
train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
val_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
```

1.6 Building Training Loop

In this section you build your training loop consisting of training and validation sequences.

The core of training is using the model to calculate the logits on specific set of inputs and compute loss (in this case **sparse categorical crossentropy**) by comparing the predicted outputs to the true outputs. You then update the trainable weights using the optimizer algorithm chosen. Optimizer algorithm requires your computed loss and partial derivatives of loss with respect to each of the trainable weights to make updates to the same.

You use gradient tape to calculate the gradients and then update the model trainable weights using the optimizer.

[10]:
```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

This function performs training during one epoch. You run through all batches of training data in each epoch to make updates to trainable weights using your previous function. You can see that we also call update_state on your metrics to accumulate the value of your metrics. You are displaying a progress bar to indicate completion of training in each epoch. Here you use tqdm for displaying the progress bar.

[11]:
```
def train_data_for_one_epoch():
  losses = []
  pbar = tqdm(total=len(list(enumerate(train))), position=0, leave=True,␣
↪bar_format='{l_bar}{bar}| {n_fmt}/{total_fmt} ')
  for step, (x_batch_train, y_batch_train) in enumerate(train):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train,␣
↪y_batch_train)

      losses.append(loss_value)

      train_acc_metric(y_batch_train, logits)
      pbar.set_description("Training loss for step %s: %.4f" % (int(step),␣
↪float(loss_value)))
      pbar.update()
  return losses
```

At the end of each epoch you have to validate the model on the test dataset. The following function calculates the loss on test dataset and updates the states of the validation metrics.

[12]:
```
def perform_validation():
  losses = []
  for x_val, y_val in test:
      val_logits = model(x_val)
      val_loss = loss_object(y_true=y_val, y_pred=val_logits)
      losses.append(val_loss)
      val_acc_metric(y_val, val_logits)
  return losses
```

Next you define the training loop that runs through the training samples repeatedly over a fixed number of epochs. Here you combine the functions you built earlier to establish the following flow: 1. Perform training over all batches of training data. 2. Get values of metrics. 3. Perform validation to calculate loss and update validation metrics on test data. 4. Reset the metrics at the end of epoch. 5. Display statistics at the end of each epoch.

Note : You also calculate the training and validation losses for the whole epoch at the end of the epoch.

[13]:
```
model = base_model()

# Iterate over epochs.
epochs = 10
epochs_val_losses, epochs_train_losses = [], []
for epoch in range(epochs):
  print('Start of epoch %d' % (epoch,))

  losses_train = train_data_for_one_epoch()
  train_acc = train_acc_metric.result()

  losses_val = perform_validation()
  val_acc = val_acc_metric.result()
```

```
    losses_train_mean = np.mean(losses_train)
    losses_val_mean = np.mean(losses_val)
    epochs_val_losses.append(losses_val_mean)
    epochs_train_losses.append(losses_train_mean)

    print('\n Epoch %s: Train loss: %.4f  Validation Loss: %.4f, Train Accuracy:␣
↪%.4f, Validation Accuracy %.4f' % (epoch, float(losses_train_mean),␣
↪float(losses_val_mean), float(train_acc), float(val_acc)))

    train_acc_metric.reset_states()
    val_acc_metric.reset_states()
```

```
Start of epoch 0

Training loss for step 937: 0.1539: 100%|      | 937/938


 Epoch 0: Train loss: 0.5409  Validation Loss: 0.4577, Train Accuracy: 0.8108,
Validation Accuracy 0.8381
Start of epoch 1

Training loss for step 937: 0.3998: 100%|      | 937/938


 Epoch 1: Train loss: 0.3924  Validation Loss: 0.3949, Train Accuracy: 0.8584,
Validation Accuracy 0.8591
Start of epoch 2

Training loss for step 937: 0.3582: 100%|      | 937/938


 Epoch 2: Train loss: 0.3503  Validation Loss: 0.3750, Train Accuracy: 0.8723,
Validation Accuracy 0.8688
Start of epoch 3

Training loss for step 937: 0.2919: 100%|      | 937/938


 Epoch 3: Train loss: 0.3272  Validation Loss: 0.3692, Train Accuracy: 0.8792,
Validation Accuracy 0.8687
Start of epoch 4

Training loss for step 937: 0.1376: 100%|      | 937/938


 Epoch 4: Train loss: 0.3101  Validation Loss: 0.3745, Train Accuracy: 0.8853,
Validation Accuracy 0.8664
Start of epoch 5

Training loss for step 937: 0.2633: 100%|      | 937/938
```

```
 Epoch 5: Train loss: 0.2947  Validation Loss: 0.3540, Train Accuracy: 0.8913,
Validation Accuracy 0.8714
Start of epoch 6
```

```
Training loss for step 937: 0.1680: 100%|     | 937/938
```

```
 Epoch 6: Train loss: 0.2837  Validation Loss: 0.3527, Train Accuracy: 0.8954,
Validation Accuracy 0.8772
Start of epoch 7
```

```
Training loss for step 937: 0.3874: 100%|     | 937/938
```

```
 Epoch 7: Train loss: 0.2745  Validation Loss: 0.3701, Train Accuracy: 0.8972,
Validation Accuracy 0.8709
Start of epoch 8
```

```
Training loss for step 937: 0.1331: 100%|     | 937/938
```

```
 Epoch 8: Train loss: 0.2653  Validation Loss: 0.3519, Train Accuracy: 0.9011,
Validation Accuracy 0.8749
Start of epoch 9
```

```
Training loss for step 937: 0.1701: 100%|     | 937/938
```

```
 Epoch 9: Train loss: 0.2556  Validation Loss: 0.3495, Train Accuracy: 0.9043,
Validation Accuracy 0.8793
```

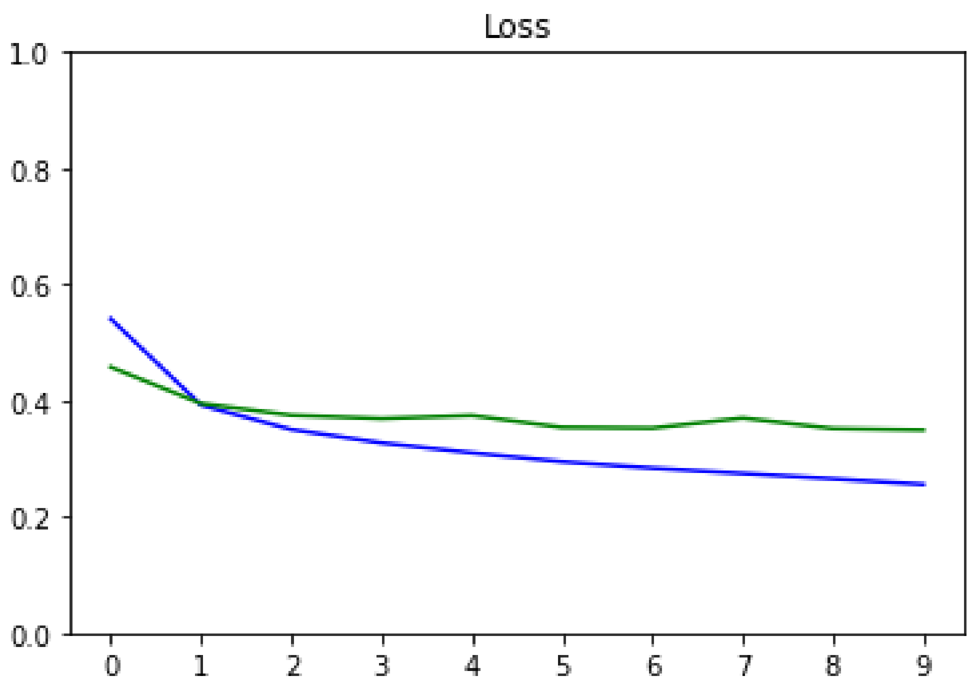
1.7 Evaluate Model

1.7.1 Plots for Evaluation

You plot the progress of loss as training proceeds over number of epochs.

[14]:
```
def plot_metrics(train_metric, val_metric, metric_name, title, ylim=5):
  plt.title(title)
  plt.ylim(0,ylim)
  plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
  plt.plot(train_metric,color='blue',label=metric_name)
  plt.plot(val_metric,color='green',label='val_' + metric_name)

plot_metrics(epochs_train_losses, epochs_val_losses, "Loss", "Loss", ylim=1.0)
```



This function displays a row of images with their predictions and true labels.

```
[15]: # utility to display a row of images with their predictions and true labels
      def display_images(image, predictions, labels, title, n):

        display_strings = [str(i) + "\n\n" + str(j) for i, j in zip(predictions,␣
       ↪labels)]

        plt.figure(figsize=(17,3))
        plt.title(title)
        plt.yticks([])
        plt.xticks([28*x+14 for x in range(n)], display_strings)
        plt.grid(None)
        image = np.reshape(image, [n, 28, 28])
        image = np.swapaxes(image, 0, 1)
        image = np.reshape(image, [28, 28*n])
        plt.imshow(image)
```

You make predictions on the test dataset and plot the images with their true and predicted values.

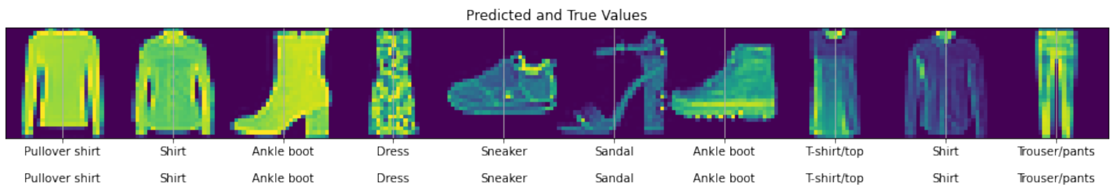
```
[16]: test_inputs = test_data.batch(batch_size=1000001)
      x_batches, y_pred_batches, y_true_batches = [], [], []

      for x, y in test_inputs:
        y_pred = model(x)
```

```
  y_pred_batches = y_pred.numpy()
  y_true_batches = y.numpy()
  x_batches = x.numpy()

indexes = np.random.choice(len(y_pred_batches), size=10)
images_to_plot = x_batches[indexes]
y_pred_to_plot = y_pred_batches[indexes]
y_true_to_plot = y_true_batches[indexes]

y_pred_labels = [class_names[np.argmax(sel_y_pred)] for sel_y_pred in␣
 ↪y_pred_to_plot]
y_true_labels = [class_names[sel_y_true] for sel_y_true in y_true_to_plot]
display_images(images_to_plot, y_pred_labels, y_true_labels, "Predicted and␣
 ↪True Values", 10)
```



```
Training loss for step 937: 0.1701: 100%|      | 938/938
```