MNIST handwritten digits classification with MLPs

In this notebook, we'll train a multi-layer perceptron model to classify MNIST digits using TensorFlow (version $\ge$ 2.0 required) with the Keras API.

First, the needed imports.

In [1]:
```
%matplotlib inline

import os
if not os.path.isfile('pml_utils.py'):
  !wget https://raw.githubusercontent.com/csc-training/intro-to-dl/master/day1/pml_
from pml_utils import show_failures
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model, to_categorical

from packaging.version import Version as LV

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

print('Using Tensorflow version: {}, and Keras version: {}.'.format(tf.__version__
assert(LV(tf.__version__) >= LV("2.0.0"))
```

```
Using Tensorflow version: 2.10.0, and Keras version: 2.10.0.
```

Let's check if we have GPU available.

In [2]:
```
gpus = tf.config.list_physical_devices('GPU')
if len(gpus) > 0:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError:
        pass
    from tensorflow.python.client import device_lib
    for d in device_lib.list_local_devices():
        if d.device_type == 'GPU':
            print('GPU', d.physical_device_desc)
else:
    print('No GPU, using CPU instead.')
```

```
No GPU, using CPU instead.
```

MNIST data set

Next we'll load the MNIST handwritten digits data set using TensorFlow's own tools. First time we may have to download the data, which can take a while.

In [3]:
```
#from tensorflow.keras.datasets import mnist, fashion_mnist
from tensorflow.keras.datasets import fashion_mnist

## MNIST:
#(X_train, y_train), (X_test, y_test) = mnist.load_data()
## Fashion-MNIST:
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

nb_classes = 10

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255.0
X_test /= 255.0

# one-hot encoding:
Y_train = to_categorical(y_train, nb_classes)
Y_test = to_categorical(y_test, nb_classes)

print()
#print('MNIST data loaded: train:',len(X_train),'test:',len(X_test))
print('Fashion-MNIST data loaded: train:', len(X_train), 'test:', len(X_test))
print('X_train:', X_train.shape)
print('y_train:', y_train.shape)
print('Y_train:', Y_train.shape)
```

```
Fashion-MNIST data loaded: train: 60000 test: 10000
X_train: (60000, 28, 28)
y_train: (60000,)
Y_train: (60000, 10)
```

The training data (`X_train`) is a 3rd-order tensor of size (60000, 28, 28), i.e. it consists of 60000 images of size 28x28 pixels. `y_train` is a 60000-dimensional vector containing the correct classes ("0", "1", ..., "9") for each training sample, and `Y_train` is a one-hot encoding of `y_train` .

Let's take a closer look. Here are the first 10 training digits (or fashion items for Fashion-MNIST):

In [4]:
```
class_labels = [
    'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
    'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
]
pltsize=1
plt.figure(figsize=(10*pltsize, pltsize))

for i in range(10):
    plt.subplot(1,10,i+1)
    plt.axis('off')
    plt.imshow(X_train[i,:,:], cmap="gray")
    #plt.title('Class: '+str(y_train[i]))
    plt.title('Class: ' + str(y_train[i]) + ' - ' + class_labels[y_train[i]])
    print('Training sample',i,': class:',y_train[i], ', one-hot encoded:', Y_train
```

```
Training sample 0 : class: 9 , one-hot encoded: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
Training sample 1 : class: 0 , one-hot encoded: [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Training sample 2 : class: 0 , one-hot encoded: [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Training sample 3 : class: 3 , one-hot encoded: [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
Training sample 4 : class: 0 , one-hot encoded: [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Training sample 5 : class: 2 , one-hot encoded: [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
Training sample 6 : class: 7 , one-hot encoded: [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
Training sample 7 : class: 2 , one-hot encoded: [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
Training sample 8 : class: 5 , one-hot encoded: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
Training sample 9 : class: 5 , one-hot encoded: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Multi-layer perceptron (MLP) network

Let's create an MLP model that has multiple layers, non-linear activation functions, and optionally dropout layers for regularization.

Initialization

We first create the `Input` of shape 28x28 to match the size of the input data. Then we use a `Flatten` layer to convert the 2D image data into vectors of size 784.

We add a `Dense` layer that has 20 output nodes. The `Dense` layer connects each input to each output with some weight parameter and then passes the result through a ReLU non-linear activation function.

The output of the last layer needs to be a softmaxed 10-dimensional vector to match the ground truth (`Y_train`). This means that it will output 10 values between 0 and 1 which sum to 1, hence, together they can be interpreted as a probability distribution over our 10 classes.

After all layers are created, we create the `Model` by specifying its inputs and outputs.

Finally, we select *categorical crossentropy* as the loss function, select *adam* as the optimizer, add *accuracy* to the list of metrics to be evaluated, and `compile()` the model. Adam is simply a an advanced version of stochastic gradient descent, note there are several different options for the optimizer in Keras that we could use instead of *adam*.

In [5]:
```
# Model initialization:
inputs = keras.Input(shape=(28, 28))
x = layers.Flatten()(inputs)

# A simple model:
x = layers.Dense(units=20, activation="relu")(x)

# The last layer needs to be like this:
outputs = layers.Dense(units=10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs,
                    name="mlp_model")
model.compile(loss='categorical_crossentropy',
```

```
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())
```

```
Model: "mlp_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28)]          0

 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 20)                15700

 dense_1 (Dense)             (None, 10)                210

=================================================================
Total params: 15,910
Trainable params: 15,910
Non-trainable params: 0
_________________________________________________________________
None
```

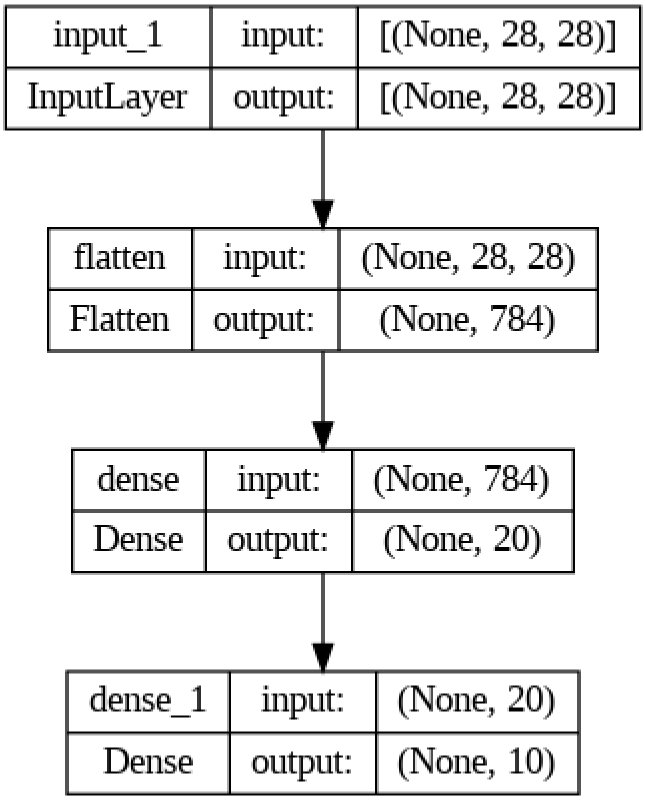
The summary shows that there are 15,910 parameters in total in our model.

For example for the first dense layer we have 785x20 = 15,700 parameters as the weight matrix is of size 785x20 (not 784, as there's an additional bias term).

We can also draw a fancier graph of our model.

In [6]:
```
plot_model(model, show_shapes=True)
```

Out[6]:



Learning

Next, we'll train our model. Notice how the interface is similar to scikit-learn: we still call the `fit()` method on our model object.

An *epoch* means one pass through the whole training data, we'll begin by running training for 10 epochs.

You can run code below multiple times and it will continue the training process from where it left off. If you want to start from scratch, re-initialize the model using the code a few cells ago.

We use a batch size of 32, so the actual input will be 32x784 for each batch of 32 images.

In [7]:
```
%%time
epochs = 10

history = model.fit(X_train, Y_train,
                    epochs=epochs,
                    batch_size=32,
                    verbose=2)
```

```
Epoch 1/10
1875/1875 - 7s - loss: 0.5871 - accuracy: 0.7978 - 7s/epoch - 3ms/step
Epoch 2/10
1875/1875 - 6s - loss: 0.4415 - accuracy: 0.8455 - 6s/epoch - 3ms/step
Epoch 3/10
1875/1875 - 5s - loss: 0.4098 - accuracy: 0.8567 - 5s/epoch - 3ms/step
Epoch 4/10
1875/1875 - 5s - loss: 0.3943 - accuracy: 0.8621 - 5s/epoch - 3ms/step
Epoch 5/10
1875/1875 - 6s - loss: 0.3785 - accuracy: 0.8666 - 6s/epoch - 3ms/step
Epoch 6/10
1875/1875 - 6s - loss: 0.3688 - accuracy: 0.8685 - 6s/epoch - 3ms/step
Epoch 7/10
1875/1875 - 6s - loss: 0.3578 - accuracy: 0.8722 - 6s/epoch - 3ms/step
Epoch 8/10
1875/1875 - 6s - loss: 0.3480 - accuracy: 0.8765 - 6s/epoch - 3ms/step
Epoch 9/10
1875/1875 - 6s - loss: 0.3379 - accuracy: 0.8802 - 6s/epoch - 3ms/step
Epoch 10/10
1875/1875 - 6s - loss: 0.3330 - accuracy: 0.8815 - 6s/epoch - 3ms/step
CPU times: user 1min 12s, sys: 14.1 s, total: 1min 26s
Wall time: 58.7 s
```

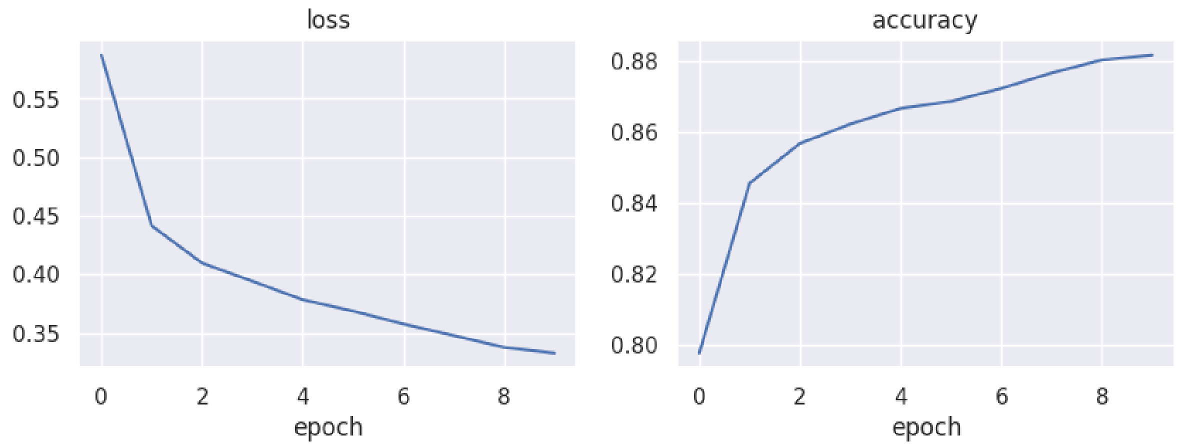
Let's now see how the training progressed.

- *Loss* is a function of the difference of the network output and the target values. We are minimizing the loss function during training so it should decrease over time.
- *Accuracy* is the classification accuracy for the training data. It gives some indication of the real accuracy of the model but cannot be fully trusted, as it may have overfitted and just memorizes the training data.

In [8]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,3))

ax1.plot(history.epoch,history.history['loss'])
ax1.set_title('loss')
ax1.set_xlabel('epoch')
```

```
ax2.plot(history.epoch,history.history['accuracy'])
ax2.set_title('accuracy')
ax2.set_xlabel('epoch');
```



Inference

For a better measure of the quality of the model, let's see the model accuracy for the test data.

In [9]:
```
%%time
scores = model.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
313/313 - 1s - loss: 0.3910 - accuracy: 0.8631 - 1s/epoch - 3ms/step
accuracy: 86.31%
CPU times: user 1.24 s, sys: 207 ms, total: 1.45 s
Wall time: 1.15 s
```

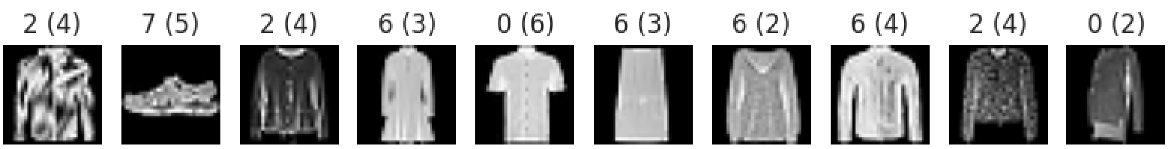
We can now take a closer look at the results using the `show_failures()` helper function.

Here are the first 10 test digits the MLP classified to a wrong class:

In [10]:
```
predictions = model.predict(X_test)

show_failures(predictions, y_test, X_test)
```

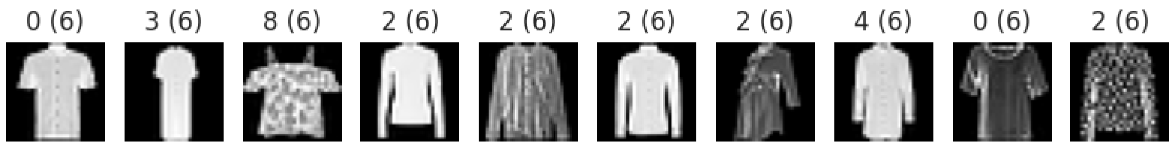
```
313/313 [==============================] - 1s 2ms/step
Showing max 10 first failures. The predicted class is shown first and the correct class in parenthesis.
```



We can use `show_failures()` to inspect failures in more detail. For example, here are failures in which the true class was "6":

In [11]:
```
show_failures(predictions, y_test, X_test, trueclass=6)
```

```
Showing max 10 first failures. The predicted class is shown first and the correct class in parenthesis.
```



We can also compute the confusion matrix to see which digits get mixed the most, and look at classification accuracies separately for each class:

In [12]:
```
from sklearn.metrics import confusion_matrix

print('Confusion matrix (rows: true classes; columns: predicted classes):'); print(
cm=confusion_matrix(y_test, np.argmax(predictions, axis=1), labels=list(range(10)))
print(cm); print()

print('Classification accuracy for each class:'); print()
for i,j in enumerate(cm.diagonal()/cm.sum(axis=1)): print("%d: %.4f" % (i,j))
```

```
Confusion matrix (rows: true classes; columns: predicted classes):

[[830   2  12  18   2   3 122   0  11   0]
 [  1 967   4  19   4   0   5   0   0   0]
 [ 16   3 810   7  75   0  81   0   8   0]
 [ 37  13  27 832  45   1  40   0   5   0]
 [  0   1 161  15 722   0  94   0   7   0]
 [  0   0   0   0   0 930   0  50   3  17]
 [133   2  95  25  52   0 671   0  22   0]
 [  0   0   0   0   0  15   0 946   1  38]
 [  4   0   0   6   5   1  10   4 970   0]
 [  0   0   0   1   0   4   1  41   0 953]]

Classification accuracy for each class:

0: 0.8300
1: 0.9670
2: 0.8100
3: 0.8320
4: 0.7220
5: 0.9300
6: 0.6710
7: 0.9460
8: 0.9700
9: 0.9530
```

Task 1: Model with two dense layers

Your task is to try the same problem as above, but with a more complex model. The new model should have **two dense layers**, each with:

- 50 units
- ReLU activation
- each followed by a dropout layer with a rate of 0.2

Dropout randomly sets a fraction of inputs to zero during training, which is one approach to regularization and can sometimes help to prevent overfitting.

You can consult the Keras documentation at https://keras.io/. For example, the Dense, Activation, and Dropout layers are described at https://keras.io/layers/core/.

The code below is missing the model definition. You can copy any suitable layers from the example above.

In [13]:
```
ex1_inputs = keras.Input(shape=(28, 28))
x = layers.Flatten()(ex1_inputs)

# First dense layer with dropout
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(0.2)(x)

# Second dense layer with dropout
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(0.2)(x)

# Output layer
ex1_outputs = layers.Dense(units=10, activation="softmax")(x)

# Create the model
model_ex1 = keras.Model(inputs=ex1_inputs, outputs=ex1_outputs, name="mlp_model_ex1
```

Execute cell to see the example answer. Note: in Google Colab you have to click and copy the answer manually.

In [14]:
```
# %load solutions/tf2-mnist-mlp-example-answer.py
ex1_inputs = keras.Input(shape=(28, 28))

x = layers.Flatten()(ex1_inputs)
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(rate=0.2)(x)
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(rate=0.2)(x)

ex1_outputs = layers.Dense(units=10, activation='softmax')(x)
```

In [15]:
```
assert ex1_outputs is not None, "You need to write the missing model definition"

ex1_model = keras.Model(inputs=ex1_inputs, outputs=ex1_outputs,
                        name="two_layer_mlp_model")
ex1_model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
print(ex1_model.summary())
```

```
Model: "two_layer_mlp_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_3 (InputLayer)        [(None, 28, 28)]          0         
                                                                 
 flatten_2 (Flatten)         (None, 784)               0         
                                                                 
 dense_5 (Dense)             (None, 50)                39250     
                                                                 
 dropout_2 (Dropout)         (None, 50)                0         
                                                                 
 dense_6 (Dense)             (None, 50)                2550      
                                                                 
 dropout_3 (Dropout)         (None, 50)                0         
                                                                 
 dense_7 (Dense)             (None, 10)                510       
                                                                 
=================================================================
Total params: 42,310
Trainable params: 42,310
Non-trainable params: 0
_________________________________________________________________
None
```

In [16]:

```
%%time
ex1_epochs = 10
ex1_history = ex1_model.fit(X_train, Y_train,
                            epochs=epochs,
                            batch_size=32,
                            verbose=2)
```

```
Epoch 1/10
1875/1875 - 7s - loss: 0.6416 - accuracy: 0.7732 - 7s/epoch - 4ms/step
Epoch 2/10
1875/1875 - 7s - loss: 0.4696 - accuracy: 0.8324 - 7s/epoch - 4ms/step
Epoch 3/10
1875/1875 - 7s - loss: 0.4326 - accuracy: 0.8448 - 7s/epoch - 3ms/step
Epoch 4/10
1875/1875 - 7s - loss: 0.4137 - accuracy: 0.8508 - 7s/epoch - 4ms/step
Epoch 5/10
1875/1875 - 7s - loss: 0.3981 - accuracy: 0.8559 - 7s/epoch - 4ms/step
Epoch 6/10
1875/1875 - 7s - loss: 0.3893 - accuracy: 0.8578 - 7s/epoch - 4ms/step
Epoch 7/10
1875/1875 - 7s - loss: 0.3812 - accuracy: 0.8622 - 7s/epoch - 4ms/step
Epoch 8/10
1875/1875 - 7s - loss: 0.3732 - accuracy: 0.8642 - 7s/epoch - 4ms/step
Epoch 9/10
1875/1875 - 7s - loss: 0.3618 - accuracy: 0.8666 - 7s/epoch - 4ms/step
Epoch 10/10
1875/1875 - 7s - loss: 0.3608 - accuracy: 0.8693 - 7s/epoch - 4ms/step
CPU times: user 1min 43s, sys: 23.9 s, total: 2min 6s
Wall time: 1min 11s
```

Let's plot the data to see how the training progressed.

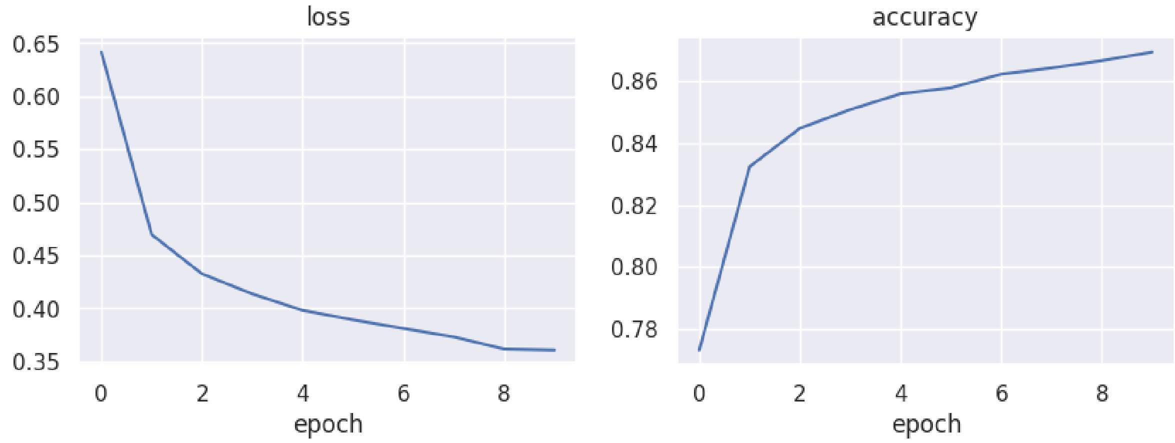
In [17]:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,3))

ax1.plot(ex1_history.epoch,ex1_history.history['loss'])
ax1.set_title('loss')
```

```
ax1.set_xlabel('epoch')

ax2.plot(ex1_history.epoch,ex1_history.history['accuracy'])
ax2.set_title('accuracy')
ax2.set_xlabel('epoch');
```



In [18]:
```
ex1_scores = ex1_model.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (ex1_model.metrics_names[1], ex1_scores[1]*100))
```

```
313/313 - 1s - loss: 0.3651 - accuracy: 0.8696 - 1s/epoch - 4ms/step
accuracy: 86.96%
```

Task 2: Model tuning

Modify the MLP model. Try to improve the classification accuracy, or experiment with the effects of different parameters. If you are interested in the state-of-the-art performance on permutation invariant MNIST, see e.g. this paper by Aalto University / The Curious AI Company researchers.

You can also consult the Keras documentation at https://keras.io/. For example, the Dense, Activation, and Dropout layers are described at https://keras.io/layers/core/.

In [19]:
```
ex2_inputs = keras.Input(shape=(28, 28))
x = layers.Flatten()(ex2_inputs)

# Two dense layers with ReLU activation and dropout
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense(units=50, activation="relu")(x)
x = layers.Dropout(0.2)(x)

# Output layer
ex2_outputs = layers.Dense(units=10, activation="softmax")(x)

# Create the model
model_ex2 = keras.Model(inputs=ex2_inputs, outputs=ex2_outputs, name="mlp_model_ex2

# Compile the model
model_ex2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accu

# Print the model summary
print(model_ex2.summary())

# Train the model
```

```
epochs = 10
history_ex2 = model_ex2.fit(X_train, Y_train, epochs=epochs, batch_size=32, verbose

# Evaluate the model on test data
scores_ex2 = model_ex2.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (model_ex2.metrics_names[1], scores_ex2[1] * 100))
```

```
Model: "mlp_model_ex2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 28, 28)]          0

 flatten_3 (Flatten)         (None, 784)               0

 dense_8 (Dense)             (None, 50)                39250

 dropout_4 (Dropout)         (None, 50)                0

 dense_9 (Dense)             (None, 50)                2550

 dropout_5 (Dropout)         (None, 50)                0

 dense_10 (Dense)            (None, 10)                510

=================================================================
Total params: 42,310
Trainable params: 42,310
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
1875/1875 - 7s - loss: 0.6528 - accuracy: 0.7651 - 7s/epoch - 4ms/step
Epoch 2/10
1875/1875 - 7s - loss: 0.4697 - accuracy: 0.8314 - 7s/epoch - 4ms/step
Epoch 3/10
1875/1875 - 7s - loss: 0.4357 - accuracy: 0.8433 - 7s/epoch - 4ms/step
Epoch 4/10
1875/1875 - 7s - loss: 0.4153 - accuracy: 0.8506 - 7s/epoch - 4ms/step
Epoch 5/10
1875/1875 - 7s - loss: 0.4002 - accuracy: 0.8547 - 7s/epoch - 4ms/step
Epoch 6/10
1875/1875 - 7s - loss: 0.3895 - accuracy: 0.8587 - 7s/epoch - 4ms/step
Epoch 7/10
1875/1875 - 7s - loss: 0.3829 - accuracy: 0.8611 - 7s/epoch - 4ms/step
Epoch 8/10
1875/1875 - 7s - loss: 0.3711 - accuracy: 0.8638 - 7s/epoch - 4ms/step
Epoch 9/10
1875/1875 - 7s - loss: 0.3664 - accuracy: 0.8680 - 7s/epoch - 4ms/step
Epoch 10/10
1875/1875 - 7s - loss: 0.3611 - accuracy: 0.8688 - 7s/epoch - 4ms/step
313/313 - 1s - loss: 0.3664 - accuracy: 0.8659 - 1s/epoch - 3ms/step
accuracy: 86.59%
```

Task 3: Fashion-MNIST

MNIST can be replaced with Fashion-MNIST, which can be used as drop-in replacement for MNIST. Fashion-MNIST contains images of 10 fashion categories:

Label	Description	Label	Description
0	T-shirt/top	5	Sandal
1	Trouser	6	Shirt
2	Pullover	7	Sneaker
3	Dress	8	Bag
4	Coat	9	Ankle boot

Replace the loading of MNIST data with Fashion-MNIST in the beginning of this notebook and re-run the experiments.

Run this notebook in Google Colaboratory using this link.