Question 3: image classification (40%)

In this question we will train a neural network to classify images of clothing. We will use Keras, a deep learning toolkit that is built on TensorFlow (developed at Google). You can install Keras and Tensorflow libraries on your local machine. However, you may find it more convenient to use Google's Colab environment for this question as all required libraries are pre-installed, and it may run faster than your computer. You can upload and download local notebooks to Colab but you should always save a recent version locally, for safety.

We will use the Fashion-MNIST dataset which is available in Keras. A tutorial that explains the dataset and the overall workflow of training an image classifier in Keras is available here:

https://www.tensorflow.org/tutorials/keras/classification

I highly recommend that you go through this first to get a good background understanding for this question.

In [3]:
```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```
```
2.5.0
```

In [4]:
```
# This makes figures that show how the training and testing accuracy and loss
# evolved against the number of epochs for the current training run

import matplotlib.pyplot as plt

def plot_train_history(h):

    plt.plot(h.history['accuracy'])
    plt.plot(h.history['val_accuracy'])
    plt.legend(("train accuracy","test accuracy"))
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.show()

    plt.plot(h.history['loss'])
    plt.plot(h.history['val_loss'])
    plt.legend(("train loss","test loss"))
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
```

Load and pre-process the images so all pixels are between 0 and 1

In [5]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Training the network

In the cell below a lot is happening.

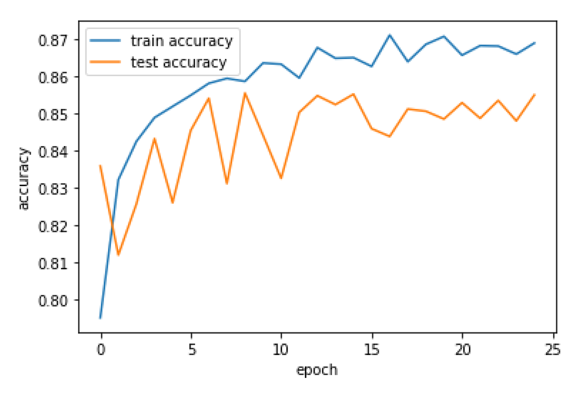
- learning_rate: This determines how quickly the network updates it weight in response to the incoming gradients. Change too slowly and the network may never reach the lowest loss value, change too fast and you run into the danger of oscillating. Typical values range between very small (1e-5) to 0.1
- max_epochs: One epoch is a pass over the whole training set. Setting this number tell the training algorithm to do this many passes over the whole data.

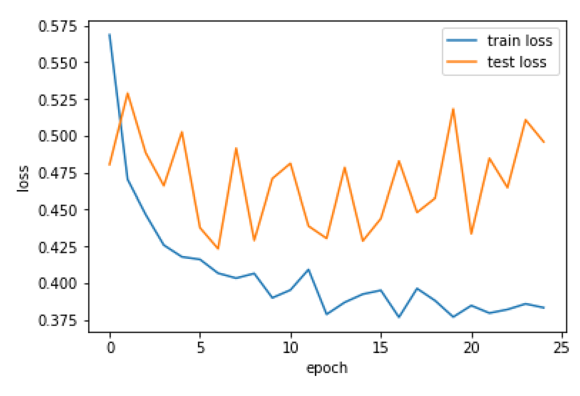
Network definition

Line 4-7 define the architecture of the network.

- Line 4: We tell keras that the model will be of the *Sequential* type, that is data is going to flow from the input to the output and we do not have any forks / loops.
- Line 6: In keras, Dense means a fully connected layer. To our model we add a Dense Layer, with 64 neurons. Assuming our input is x, the output after the fully connected layer will be of the form $y_1 = W_1 x$ Another important thing is the *activation* parameter, which we have set to sigmoid. This is the non-linearlity which will be applied to the **output** of this layer that is $y_{\sigma_1} = \sigma_1(W_1 x)$.
- Line 7: We additionally have another layer which maps the output y_{σ_1} to a single output, with another sigmoid as the activation function. The output of this sigmoid is used to classify if the class is 0 or 1. (-1 or 1 in case of Keras, but that conversion happens automatically and we do not need to worry about it.)
- Line 9: Just an architecture is not enough for learning. We need to specify a **loss function** as well as an optimizer. For this assignment, we start with Adam (an efficient version of Stochastic Gradient Descent) as the optimizier. However, we can choose different losses and see their effect on how we learn. All of this is brought together using **compile** in Keras.
- Line 13 is where the training happens. This done by calling the fit() method of the model with the training data (train_images and train_labels). We also pass the testing data to see how well we are doing along the way. This is just for evaluation and the network never uses this data to train. Once run, this will print a line every epoch to report loss and accuracy for both the training and testing sets.

In [40]:
```
learning_rate = 0.01
max_epochs = 25

model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(64, activation='ReLU'))
model.add(tf.keras.layers.Dense(64, activation='ReLU'))
model.add(tf.keras.layers.Dense(64, activation='ReLU'))
model.add(tf.keras.layers.Dense(10, activation='sigmoid'))

model.compile(optimizer=tf.keras.optimizers.Adam(lr=learning_rate),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

h = model.fit(train_images,
              train_labels,
              epochs=max_epochs,
              validation_data = (test_images, test_labels))

eval_results = model.evaluate(test_images, test_labels, verbose=0)
print("\nLoss, accuracy on test data: ")
print("%0.4f %0.2f%%" % (eval_results[0], eval_results[1]*100))

plot_train_history(h)
```
```
Epoch 1/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5687 - accuracy: 0.7950 - val_loss: 0.4805 - val_accuracy: 0.8358
Epoch 2/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4703 - accuracy: 0.8321 - val_loss: 0.5289 - val_accuracy: 0.8119
Epoch 3/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4466 - accuracy: 0.8425 - val_loss: 0.4885 - val_accuracy: 0.8257
Epoch 4/25
1875/1875 [==============================] - 5s 2ms/step - loss: 0.4259 - accuracy: 0.8489 - val_loss: 0.4661 - val_accuracy: 0.8432
Epoch 5/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4179 - accuracy: 0.8519 - val_loss: 0.5026 - val_accuracy: 0.8260
Epoch 6/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4162 - accuracy: 0.8549 - val_loss: 0.4377 - val_accuracy: 0.8454
Epoch 7/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4068 - accuracy: 0.8581 - val_loss: 0.4234 - val_accuracy: 0.8541
Epoch 8/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4034 - accuracy: 0.8595 - val_loss: 0.4915 - val_accuracy: 0.8311
Epoch 9/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4066 - accuracy: 0.8587 - val_loss: 0.4291 - val_accuracy: 0.8555
Epoch 10/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3900 - accuracy: 0.8636 - val_loss: 0.4709 - val_accuracy: 0.8441
Epoch 11/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3953 - accuracy: 0.8633 - val_loss: 0.4813 - val_accuracy: 0.8325
Epoch 12/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4093 - accuracy: 0.8596 - val_loss: 0.4388 - val_accuracy: 0.8503oss: 0.4
Epoch 13/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3788 - accuracy: 0.8678 - val_loss: 0.4305 - val_accuracy: 0.8548
Epoch 14/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3870 - accuracy: 0.8649 - val_loss: 0.4785 - val_accuracy: 0.8524
Epoch 15/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3925 - accuracy: 0.8651 - val_loss: 0.4287 - val_accuracy: 0.8552.3924 - accuracy: 0.86
Epoch 16/25
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3951 - accuracy: 0.8627 - val_loss: 0.4439 - val_accuracy: 0.8459
Epoch 17/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3768 - accuracy: 0.8712 - val_loss: 0.4829 - val_accuracy: 0.8438
Epoch 18/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3964 - accuracy: 0.8640 - val_loss: 0.4481 - val_accuracy: 0.8512
Epoch 19/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3881 - accuracy: 0.8687 - val_loss: 0.4578 - val_accuracy: 0.8506
Epoch 20/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3770 - accuracy: 0.8708 - val_loss: 0.5183 - val_accuracy: 0.8485
Epoch 21/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3848 - accuracy: 0.8657 - val_loss: 0.4336 - val_accuracy: 0.8529
Epoch 22/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3796 - accuracy: 0.8683 - val_loss: 0.4847 - val_accuracy: 0.8487
Epoch 23/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3821 - accuracy: 0.8682 - val_loss: 0.4648 - val_accuracy: 0.8535
Epoch 24/25
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3859 - accuracy: 0.8660 - val_loss: 0.5109 - val_accuracy: 0.8480
Epoch 25/25
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3833 - accuracy: 0.8690 - val_loss: 0.4959 - val_accuracy: 0.8550

Loss, accuracy on test data:
0.4959 85.50%
```





In [ ]:

Experimenting with the network

The following questions require you to train and evaluate a model with several different settings. You should modify the example code above so that it is convenient for you to run your tests repeatedly. For each question, start with the base case when performing comparisons.

1. The current architecture has a single hidden layer with 64 neurons in it. We can add more neuron in this hidden layer (try doubling it for example), this will make the layer "wider". Alternatively, we can add an additional layer. Compare and contrast the performance of these two approaches. How many parameters does the network contain for each setting you tried?

In [ ]:
```
#Just some notes on the testing above
# original values: dense layer with 64 and 10 neurons, with sigmoid activation function
#     result: 0.3426 87.63%
# double neuron in the hidden layer:
#     result: 0.3286 88.15% we can see from the sample that, the accuracy osciallte a lot
# adding one more layer:
#     result: 0.3419 87.82%
# using 2000 neurons:
#     result: 0.3156 89.15%
# Adding five more layers:
#     result: 0.4539 84.42%
```

In [ ]:
```
# Your answer here
# Doubling the neuron in the hidden layer did not do much in terms of improving our accuracy
#   and loss, but if we increase the neurons drastically(2000 neurons), we can see there is
#   huge improvement both on loss and accuracy.

# Adding more hidden layer actually reduce both the accuracy and loss.

# Adding more parameters is usually good for our model, but if we have too much,
#   such as having a wide model, it puts more pressure on our learning.

# If the model is too deep, it will increase our rate of learning and calculation speed,
#   it is also hard to train, because it often oscillate between boundary.
```

1. Change the **activation function** on the intermediate layers and measure the effect on accuracy and convergence rate. Discuss your results. In the lectures we saw that ReLU usually learns faster than sigmoids, does this hold true? How many epochs does the ReLU based training converge in?

In [ ]:
```
# Your answer here
#case 1: Sigmoid activation function with 64 neurons
#     result: 0.3426 87.63%

#case 2: ReLU activation function with 64 neurons
#     result: 0.3509 87.47%

#case 3: Leaky ReLU activation function with 64 neurons
#     result: 0.3837 86.10%

#case 4: tanh activation function with 64 neurons
#     result: 0.3330 88.26%

#Since using an epochs of 10 produce very similar result, we move onto using 50 epochs

#case 1: Sigmoid activation function with 64 neurons
#     result: 0.3847 88.42%

#case 2: ReLU activation function with 64 neurons
#     result: 0.4341 88.20%

#case 3: ReLU activation function with 3 layers of 64 neurons
#     result: 0.3912 89.02%

# From observing the patterns, ReLU and sigmoid both converged when epochs is around 20

# Is it not neccessary the case that ReLU will learn faster than Sigmoid, it will need to de
pend on the network
#   that it is trained with. In theory, ReLU does compute better than Sigmoid, since ReLU on
ly need to pick
#   max(0,x)and not peform expensive exponential operation as in sigmoid.
```

1. Vary the **learning rate** and show its effect on the accuracy and convergence speed of the network.

In [ ]:
```
# Your answer here
#case 1: ReLU activation function with 64 neurons, lr = 0.001, epochs = 25
#     result: 0.3303 88.34%

#case 2: ReLU activation function with 64 neurons, lr = 0.005, epochs = 25
#     result: 0.4000 86.57%

#case 3: ReLU activation function with 64 neurons, lr = 0.01, epochs = 25
#     result: 0.4361 84.93%

#case 4: ReLU activation function with 64 neurons, lr = 0.1, epochs = 25
#     result: 1.0204 66.76%

#case 5: ReLU activation function with 64 neurons, lr = 0.0001, epochs = 25
#     result: 0.3810 86.82%

#case 6: ReLU activation function with 64 neurons, lr = 0.00001, epochs = 25
#     result: 0.5609 81.47%

#case 7: ReLU activation function with 64 neurons, lr = 0.0005, epochs = 25
#     result: 0.3315 88.09%

#case: we set learning rate to be around 0.001 but its accuracy is not better than when it i
s set to 0.001


# We can see that when the learning rate start to increase from 0.001 , the accuracy actuall
y decrease.
```

1. How many epochs should the training run for? Justify your answer by making observations about convergence during your experiments.

In [ ]:
```
# Your answer here  ,
# we need to increase the number of epoch before it has a drastically increase of error(over
fitting)
# Therefore, we need to observe on where in specific does the accuracy starts to increase in
a slower rate and
#   lost start to increase.
# From the test case above, we see that when epochs reach between 19-25, it has the lowest l
oss, and slower
#   increase in accuracy, therefore, we set the maximum epochs to 25.
```

1. Finally, report your best combination of architecture, loss, activation function and learning rate, and evaluate your model with these settings. Discuss your result and the trade-off between classification accuracy and time/resources required to train your network.

In [ ]:
```
# Your answer here
#

# From the test cases,
#   Using deeper or wider layer will not make a huge improvement on the accuracy of the trai
ned data
#     (It does make improvement, but consider the cost of using wider and deeper network, ev
en wider network)
#     we still use the original structure, 64 neurons for the single hidden layer, since it
 is "good enough"

#   We can see that sigmoid and ReLU gives very similar outcome, therefore in the later test
#     cases, we mainly used ReLU since it gives the least computational cost.
#
#   For the learning rate, We can see that when the learning rate start to increase from 0.0
01 ,
#     the accuracy actually decrease. Therefore, the learning rate is set to be 0.001


# If we implement too many epochs, then its will take longer to compute, but if we can let i
t stop when converged
#   it will be a better option. For this example, since the training set always converged ar
ound 25, therefore we
#   set it to be 25.

# Adding more parameters(doubling neurons) is usually good for our model, but if we have too
much,  such as
#   having a wide model,it puts more pressure on our learning.

# If the model is too deep(too many layers), it will increase our rate of learning and calcu
lation speed,
#   it is also hard to train, because it often oscillate between boundary.

#Therefore, it is not only to build a model that has the best accuracy, but also runtime is
 very important.
```

Postgraduate students only (10%, other questions scaled to 90%)

Based on your previous experiments, implement another modification of your choice to improve the performance/accuracy tradeoff. Explain what you have done, why you chose to do it, and what effect it had on performance/accuracy. (Hint: even if it does not ultimately improve performance, the explanation and measurement is what is important for this question).

In [ ]:
```
# Your answer here
```