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 (http://colab.research.google.com/http://colab.research.google.com/) 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 (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 [ ]:

```
# TensorFlow and tf.keras
import tensorflow as tf

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

print(tf.__version__)
```

2.4.1

In [ ]:

```
# 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 [ ]:

```
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
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/tr
ain-labels-idx1-ubyte.gz (https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/train-labels-idx1-ubyte.gz)
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/tr
ain-images-idx3-ubyte.gz (https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/train-images-idx3-ubyte.gz)
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t1
0k-labels-idx1-ubyte.gz (https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/t10k-labels-idx1-ubyte.gz)
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t1
0k-images-idx3-ubyte.gz (https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/t10k-images-idx3-ubyte.gz)
4423680/4422102 [==============================] - 0s 0us/step
```

In [ ]:

```
print('Training data shape:',train_images[1].shape)
```

```
Training data shape: (28, 28)
```

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 $y_{\sigma 1}$ 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 [ ]:

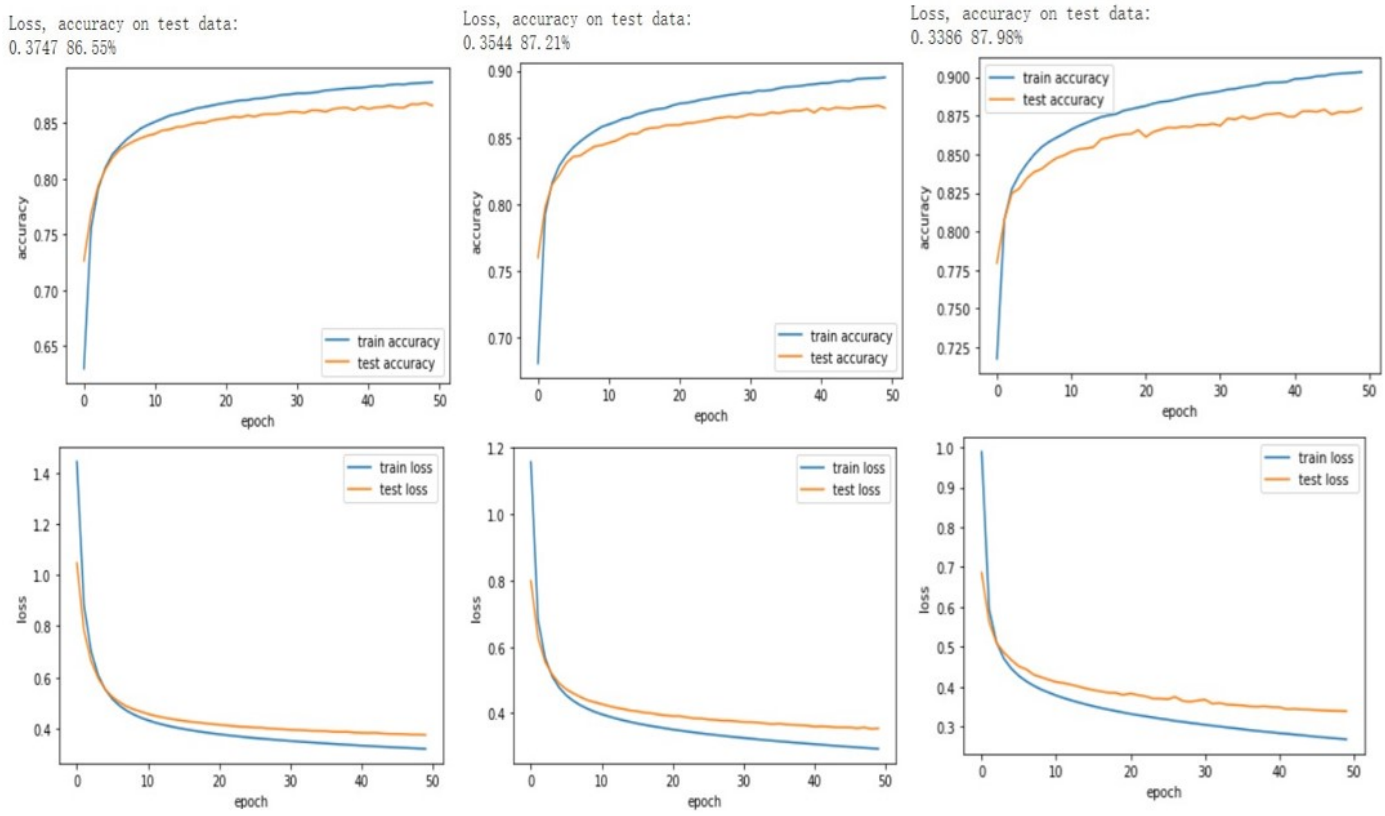
```
learning_rate = 0.0001
max_epochs = 50

model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
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)
```

Experimenting with the network

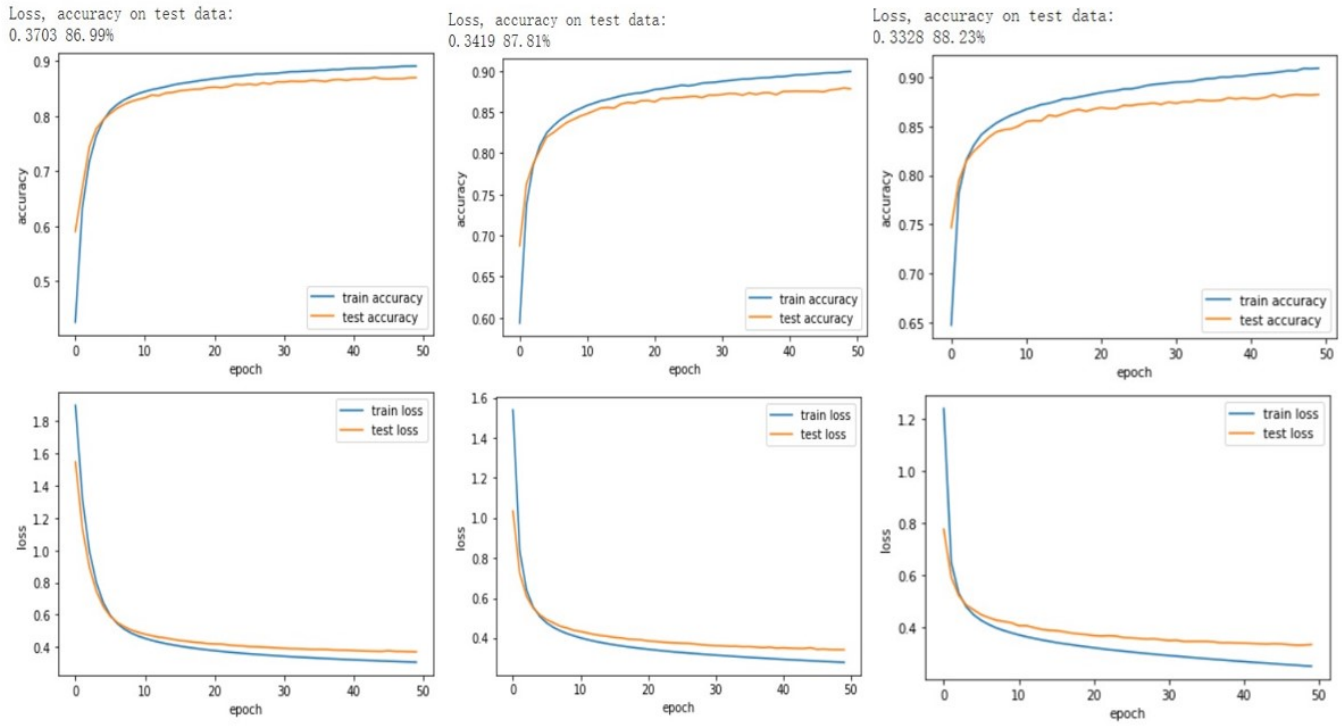
The following questions require you to train and evaluate a model with several different settings. You should modify the example code above so that it is convenient for you to run your tests repeatedly. For each question, start with the base case when performing comparisons.

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?

The Loss and Accuracy for Width of 32, 64 and 128 from left to right



The Loss and Accuracy (depth +1) with one added new hidden layer of 32, 64 and 128 from left to right



Width (no added layer)	Loss	Accuracy	Parameter
32	0.3747	86.55%	25418
64	0.3544	87.21%	50826
128	0.3386	87.98%	101642

Depth +1 (added 1 layer)	Loss	Accuracy	Parameter
32+32	0.3703	86.99%	26442
64+64	0.3419	87.81%	54922
128+128	0.3328	88.23%	118026

The tables above illustrate the losses, accuracies and the number of parameters required for corresponding setting. It is observed that the losses will decrease and accuracies will increase when we increase the neurons in the layer i.e., increase the width. However, with an increase of neurons, the number of corresponding w in weights vector will also increase. If the width is doubled, the number of parameters will also be doubled i.e., the number of parameters increase linearly 1:1 with the increase in width, and thus the computation will be a little more expensive.

Increasing the depth of a neural network will let the network approximate functions with increased non-linearity. However, in this task, the increase in depth did not improve the performance of the neuron network significantly. Meanwhile, it required a large number of parameters, thus it takes more memory for storing many new weights and the computation was expensive i.e., increased time and space complexity.

Therefore, in general, to improve the performance of the network, we should increase the width instead of the depth. According to Occam Razor, we shold keep the system simple and avoid the overfitting. However, this depends on the data we are using. If we require more non-linearity property, we could improve the network by increasing the depth.

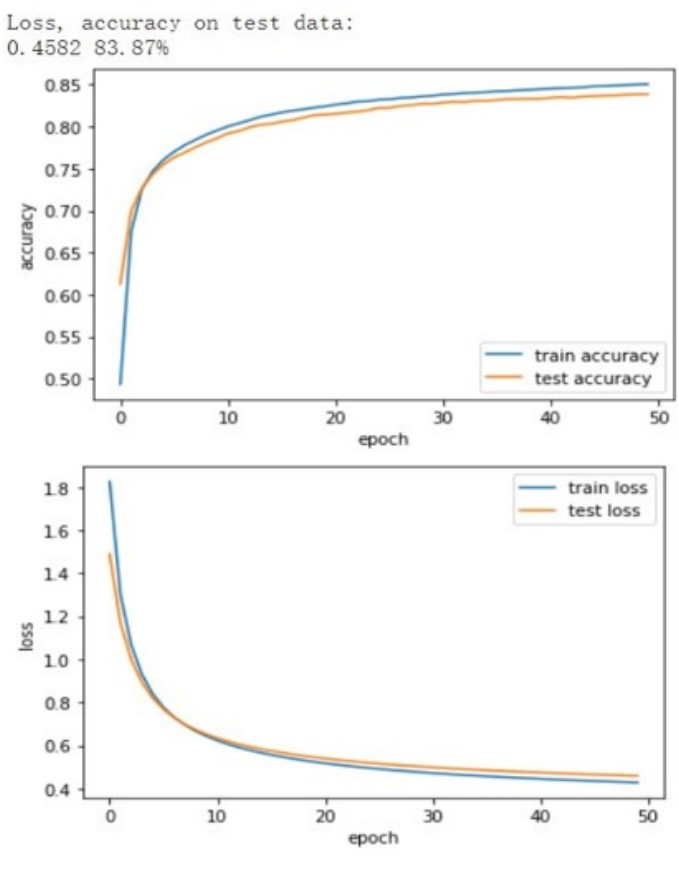
2. Change the **activation function** on the intermediate layers and measure the effect on accuracy and convergence rate. Discuss your results. In the lectures we saw that ReLU usually learns faster than sigmoids, does this hold true? How many epochs does the ReLU based training converge in?

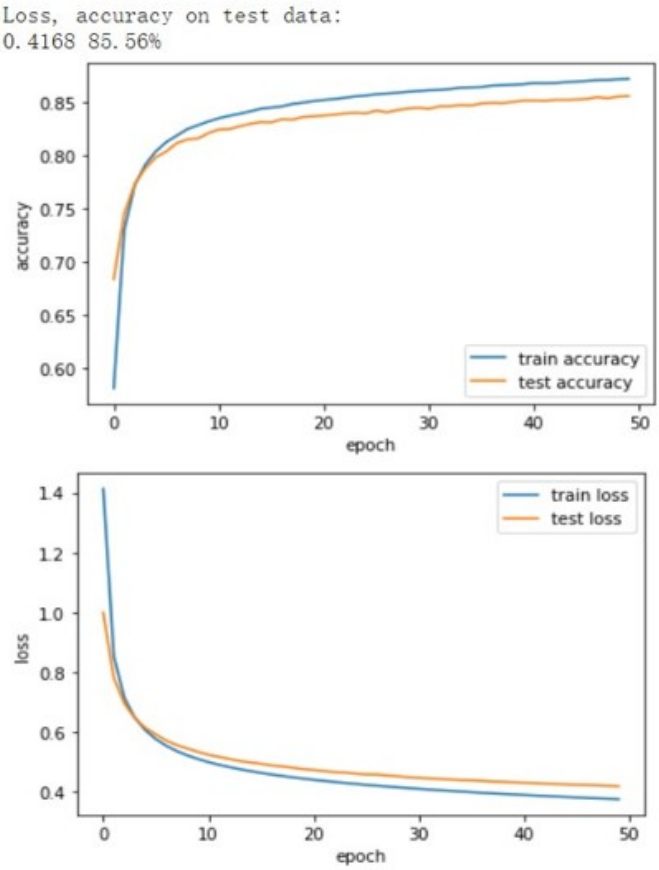
In [ ]:

```
learning_rate = 0.00001
max_epochs = 500

model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(68, 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)
```

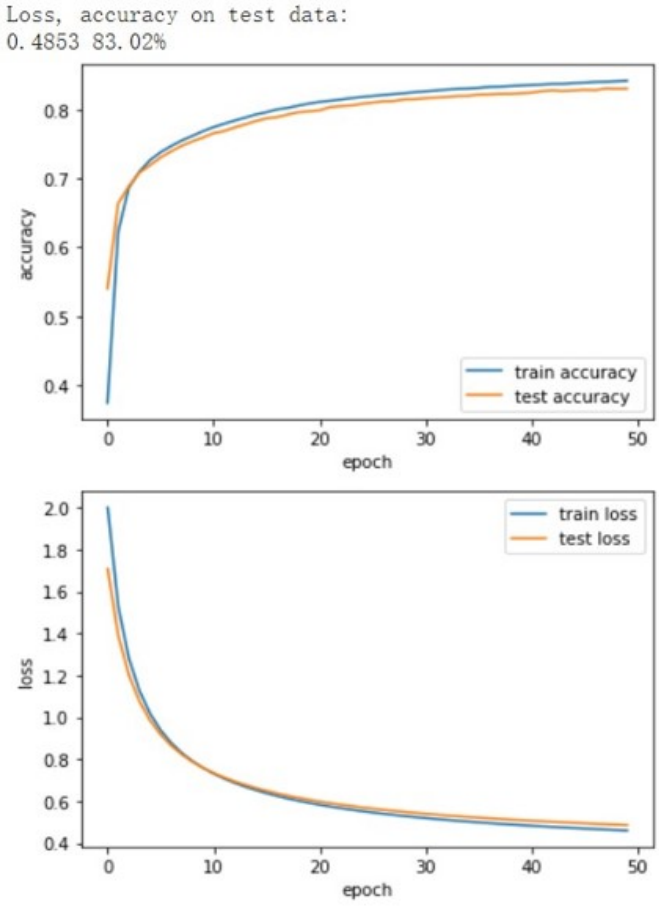
The Loss and Accuracy for different activation fuction of Sigmoid(left) and ReLU(right) for single hidden layer of

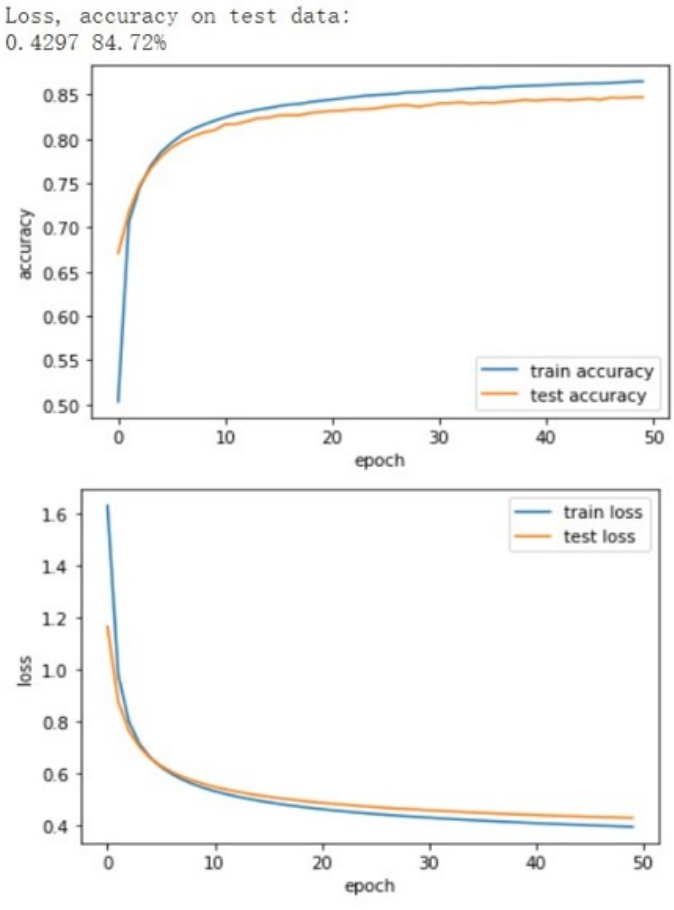
128 neurons





The Loss and Accuracy for different activation fuction of Sigmoid(left) and ReLU(right) for single hidden layer of 64 neurons





In this section, we will experiment on the effects of different activation functions: sigmoid and ReLU. The results above shows that the performance using activation of ReLU (rectified linear unit) instead of sigmoid improved from 83.87% to 85.56% for a single hidden layer of 128 neurons setting. And an improvement from 83.02% to 84.72% for a single hidden layer of 64 neurons setting. The difference showed in performance is because the different features of these two activation functions, which will be discussed in detail later.

The activation of sigmoid formula is σ(x) = 1/(1+e-x).

The activation of ReLU is F(x) = max (0, x)

The plot of them are illustrated below.

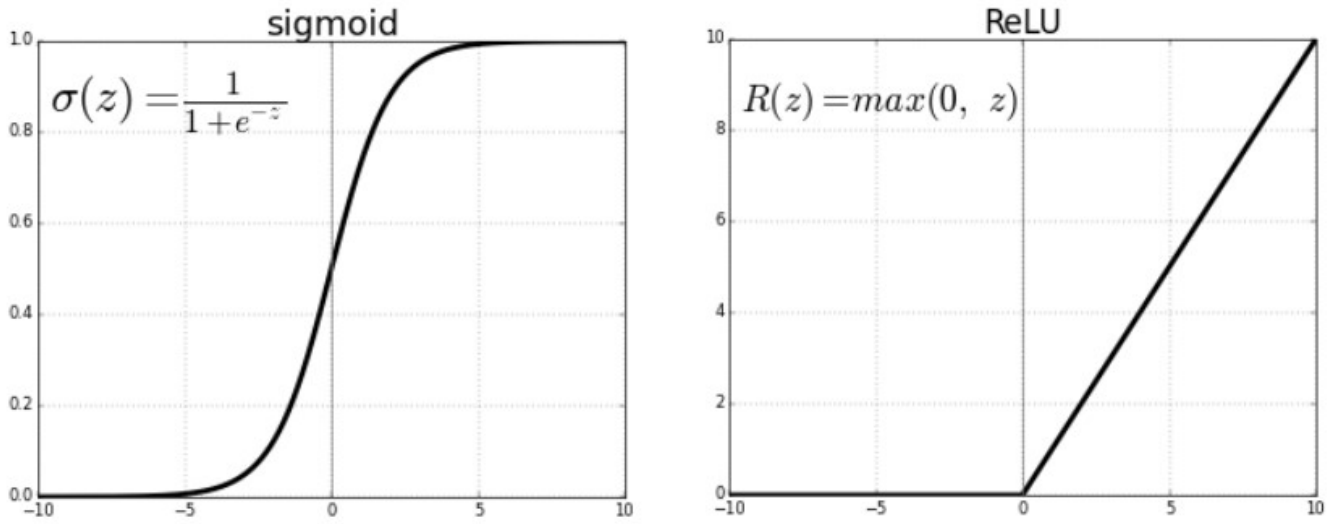
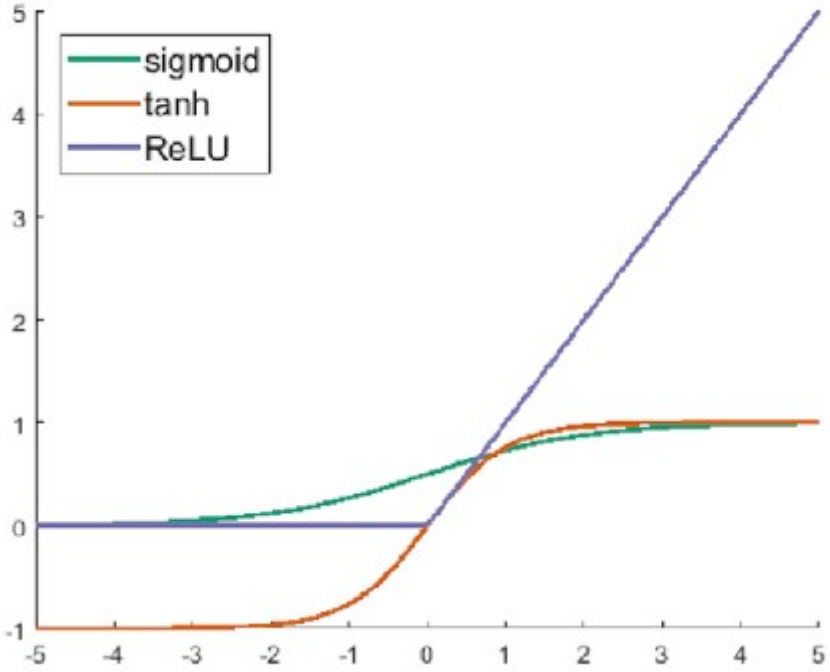


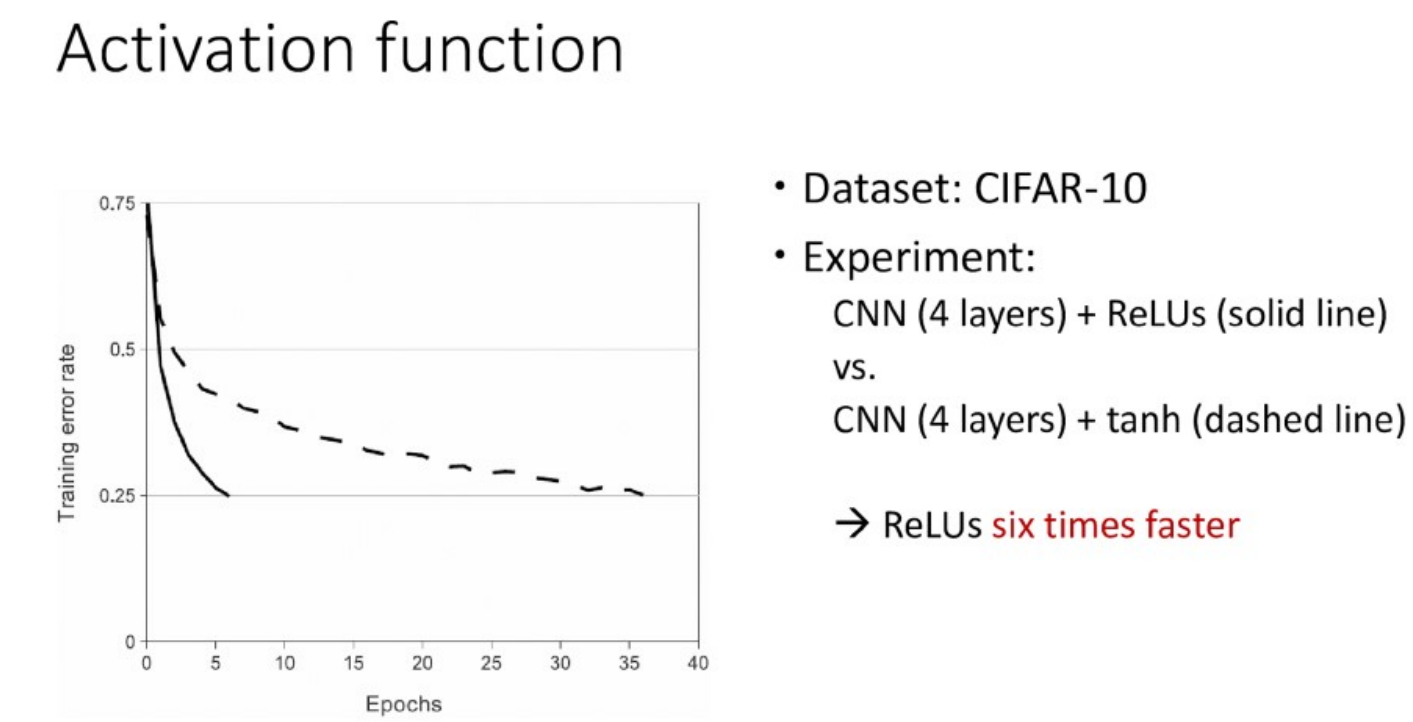
Fig: ReLU v/s Logistic Sigmoid



Historically, sigmoid was popular in deep learning since they have nice interpretation as a saturating “fringe of a neuron”. It can squashes number to range [0,1], but as shown in the graph, the output of sigmoid is not zero-

centred. This is not a good feature, because if the activation function is centred at zero, then it allows equally for grading in any direction. If weights are positive and negative, then the gradient will also be positive and negative, so we sort of ruled out all many possible directions for gradient. Another disadvantage of sigmoid is the saturation and thus slow to converge. As shown in its graph, the top and bottom are approximately flat, and thus the gradients are nearly zero, which is often called the “vanishing gradient”. The saturated neurons kills the gradients, because when we applying the gradient decend, it often stuck at that flat area of the curve and so the gradient will always be zero. So we basically waste a lot of time on gradient decend, trying to calculate the small values of gradient and end up converging very slowly. The last drawback of sigmoid is the exponential term in its formula, which is a bit expensive for computation.

However, ReLU function removed the exponential term and uses a simple equation of F(x) = max (0, x), which is very computationally efficient. ReLU function improved the saturation problem in positive region. However, there is still saturation in the negative region (The leaky ReLU does not have saturation in both x and y direction). This give rise to the most important advantage of ReLU than sigmoid that ReLU converges much faster than sigmoid. The figure below shows the derivatives of several activation functions. In positove region, reLU is a straight line without any flatness, unlike the other two. In the lecture, there is a data shows ReLU converges 6 times faster than the sigmoid, and the plot from the slide is shown below.



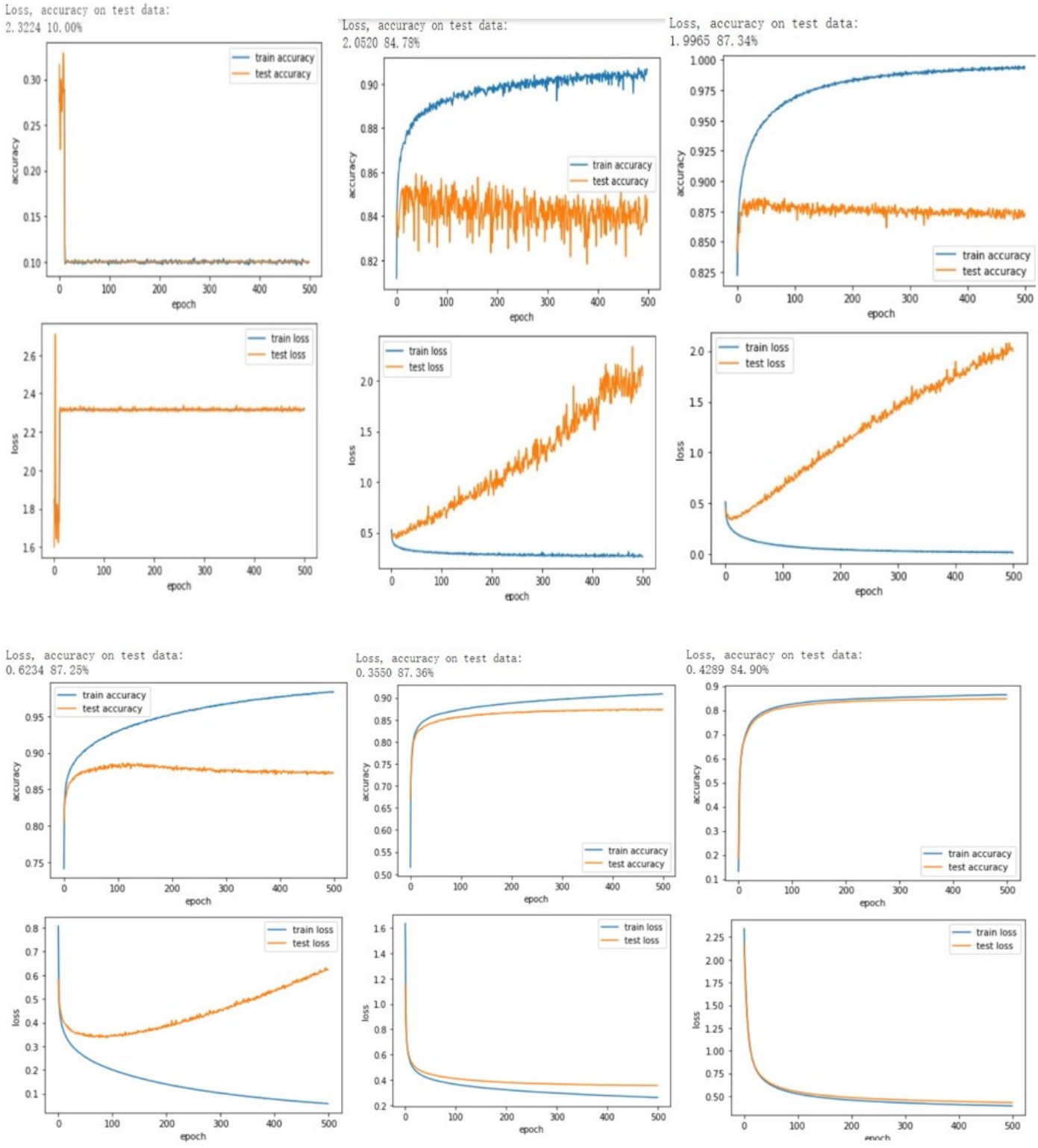
3. Vary the **learning rate** and show its effect on the accuracy and convergence speed of the network.

In [ ]:

```
learning_rate = 0.0001
max_epochs = 50

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(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)
```

In this section, we are going to experimenting on the learning rate. One of the key hyperparameters to set in order to train a neural network is the learning rate for gradient descent. This parameter scales the magnitude of our weight updates in order to minimize the network's loss function. This assignment uses Adam optimizer, which is short for Adaptive Moment Estimation. Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm. The amount that the weights are updated during training is referred to as the step size or the “learning rate.” The learning rate controls how quickly the model is adapted to the problem. Generally, smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.

The loss and accuracy plots for 500 epochs of learning rate of 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001 in order:



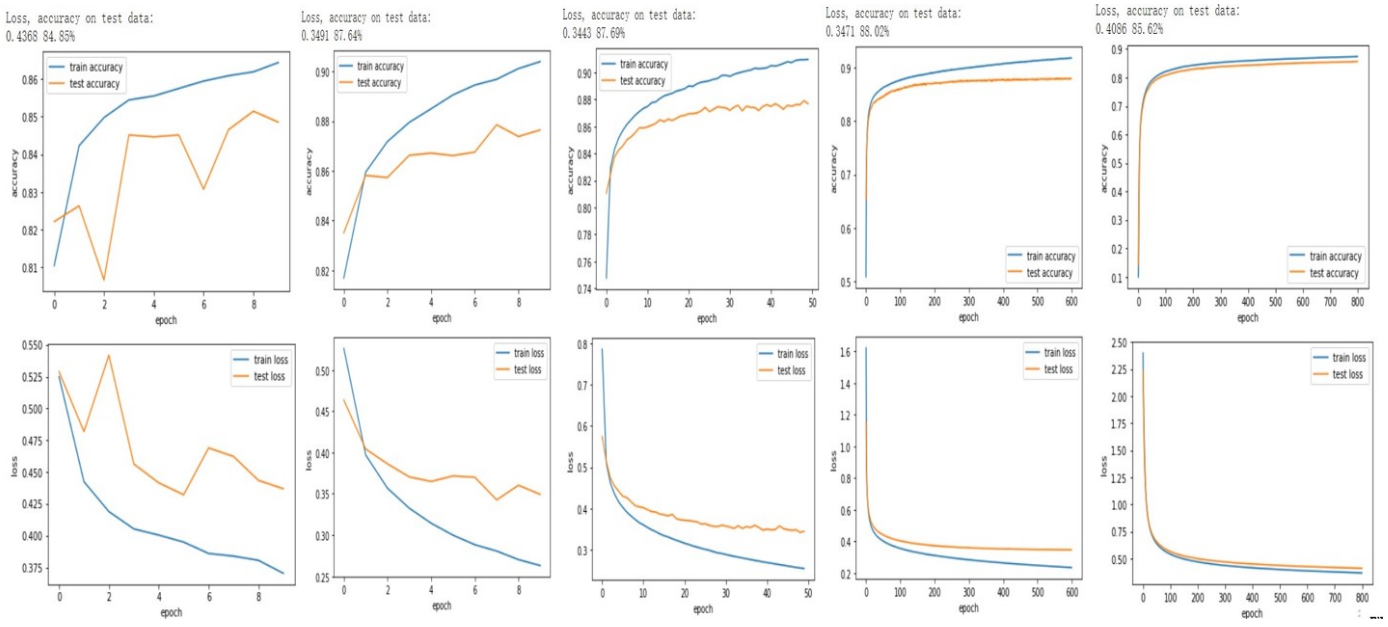
Learning rate	Epochs	Loss	Accuracy
0.1	500	2.3224	10%
0.01	500	2.0520	84.78%
0.001	500	1.9969	87.34%
0.0001	500	0.6234	87.25%
0.00001	500	0.3550	87.36%
0.000001	500	0.4289	84.90%

In the processing of testing the learning rate, the epochs were set relatively large of 500 iterations. And I was

experimenting learning rate from large learning rate to small learning rate, from 0.1 to 1e^-6. In the first case of learning rate of 0.1, it is sown that the learning rate is too large that the network can't learn and the accuracy keeps at 10%. There were also some significant signs of overfitting for large learning rates (0.01, 0.001, 0.0001), that the training error decreases while the testing error increases, which indicates the learning should be terminated earlier. Therefore, the max epochs was set to be smaller, that are 10, 10 and 50 respectively. After the adjustment, the plot of 0.01 and 0.001 below still shows significant fluctuations in loss and accuracy. This indicates these large learning rates are not suitable for this neural network setting. For the learning rates of 0.00001 and 0.000001, I increased the epochs to 600 and 800 respectively.It is observed that the accuracy and loss improved a little and there is no sign of overfitting. However, 800 epochs appears not enough for 0.000001 learning rate. Based on its behviour in the plot, we can increase the epochs and achieve higher accuracy without overfitting, but this is extremely time-inefficient. Furthermore, 0.00001 learning rate achieved the accuracy of 87.36% in 500 epochs, and 88.02% in 600 epochs, without overfitting. However, cosidering the time and accuracy trade-off, the combination of 0.00001 and 500 epochs are chosen to be the best of all configerations.

To prevent overfitting, I can decrease the number of neurons in the hidden layer, but this would impact the performance negatively as we discussed in section 1. Therefore, I kept the setting of one single hidden layer as 128 neurons and then experiment the learning rate on that, so that the best configuration can be chosen at the end of the report. Furthermore, for the overfitting behaviors, I chose the appropriate number of epochs to stop the learning before reaching there. The methodology of determining the epochs will be discussed in detail in the next section. But, generally, as mentioned above, for the same configuration, smaller learning rate will be learning through large epochs, and vice versa.

In summary, the learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. Large learning rate learns fast, but might lead to oscillation behaviour and hard for converging. Small learning rate learns slowly, it can converge to the exact solution, but with a large number of epochs, which is time-consuming. Therefore, there is a time and accuracy trade-off, and the learning rates should be chosen according to the settings, data and maybe through testing.



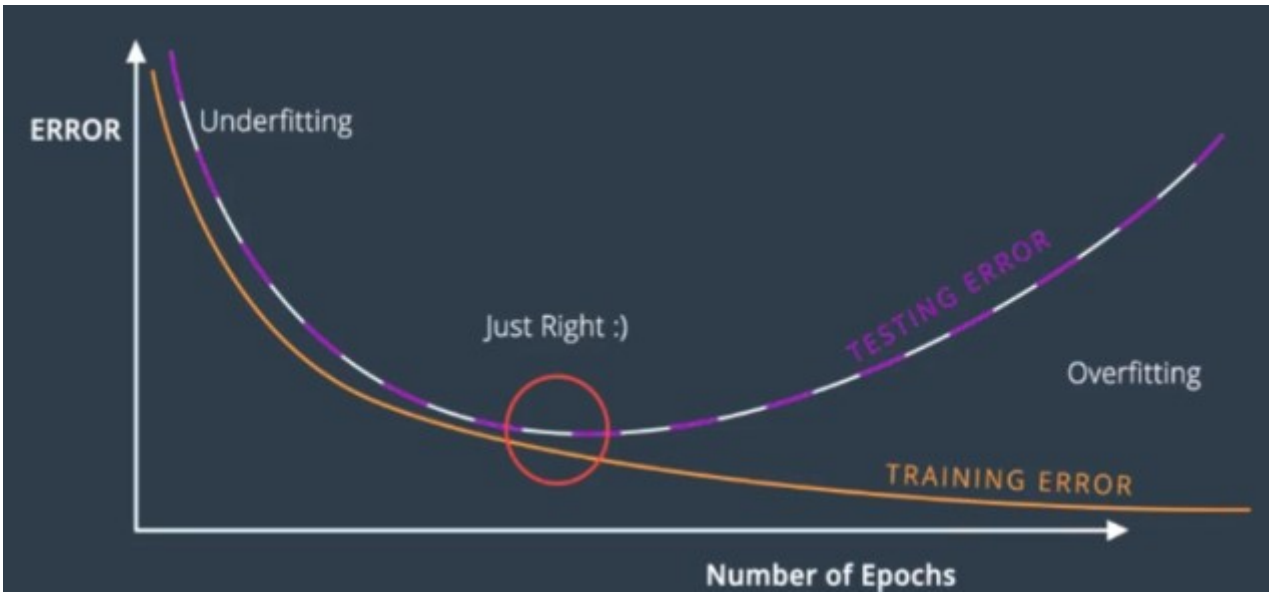
The plots above shows the loss and accuracy for the cases of : lr = 0.01,10 epochs; lr = 0.001, 10 epochs; lr = 0.0001, 50 epochs; lr = 0.00001, 600 epochs; and lr = 0.000001, 800 epochs from left to right.

Learning rate	Epochs	Loss	Accuracy
0.1	500	2.3224	10%
	10	2.3224	10%
0.01	500	2.0520	84.78%
	10	2.0520	84.78%
0.001	500	1.9969	87.34%
	10	1.9969	87.34%
0.0001	500	0.6234	87.25%
	50	0.3443	87.69%
0.00001	500	0.3550	87.36%
	600	0.3471	88.02%
0.000001	500	0.4289	84.90%
	800	0.4086	85.62%

The table above summarise all the losses and accuracies for different learning rates.

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

In this section, we will discuss how to determine the appropriate number of epochs. As mentioned above, a general rule is a large learning rate needs a small number of epochs, and a small learning rate needs large number of epochs. However, the number of epochs also varies with other parameter settings and our data. Usually, to find the appropriate number of epochs we need to observe the error vs epochs plot, such as the figure below.



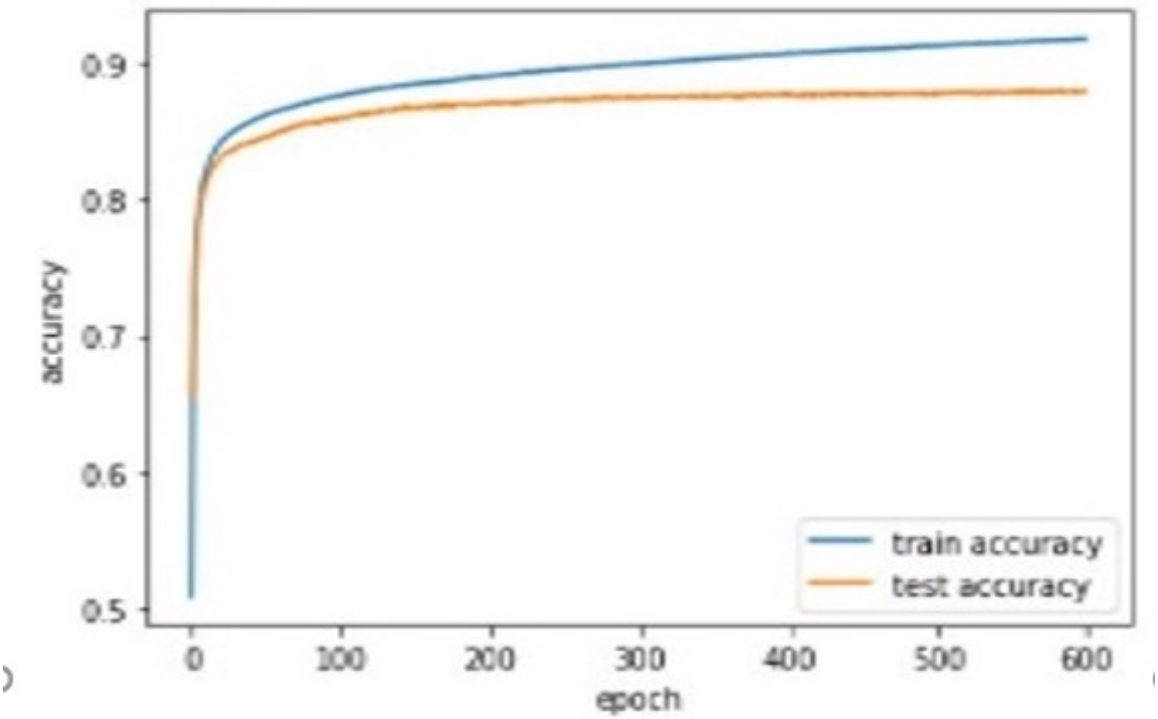
As long as the test error and the training error keeps decreasing, the training should continue. However, if the test error starts increasing that might be an indication of overfitting. Therefore, at beginning we should set the number of epochs as high as possible and terminate the training when validation error start increasing, and then pick the point that there is a divergence between the testing error and training error. And this is how I pick

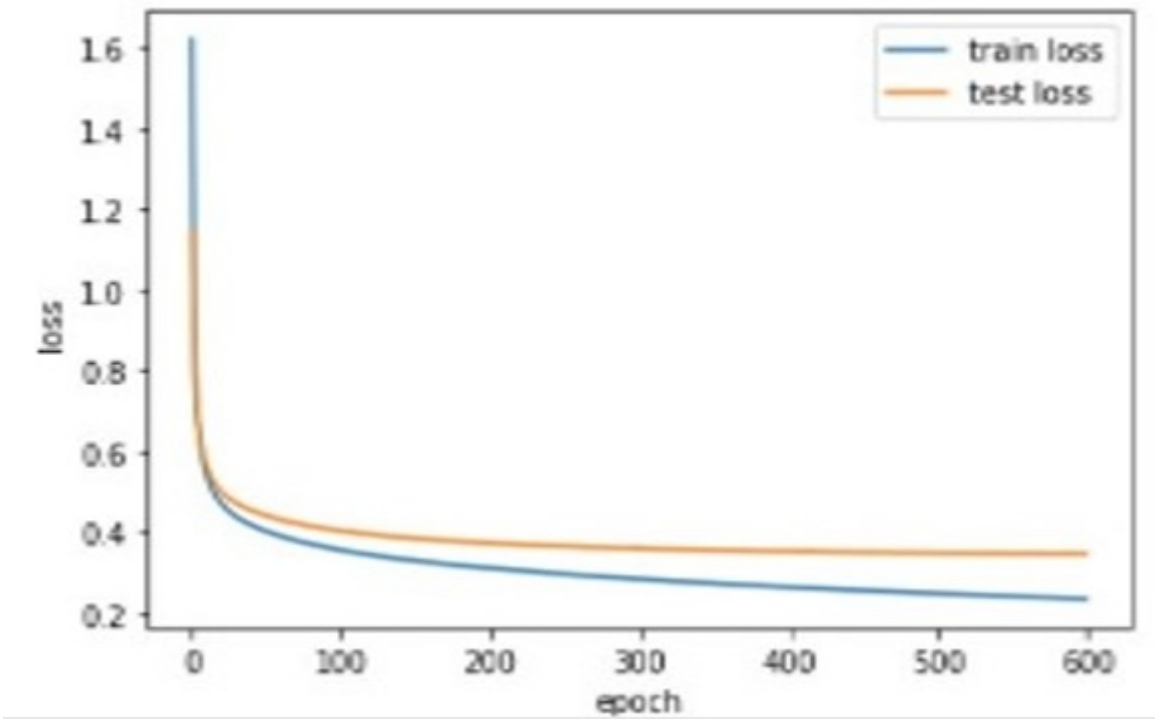
the epochs in previous section. For example, for learning rate of 0.01, 0.001 and 0.0001 with 500 epochs, we can observe from their plots that the appropriate max epochs should be set to be 10, 10 and 50 respectively. Thus, the neural network can stop training further and prevent overfitting.

5. 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 conclusion, after experimenting on these four attributes of the neural network, we found a better configuration of a single hidden layer of 128 units with ReLU activation and a learning rate of 0.0001 with 500 epochs of training. The loss and accuracy plots are shown below.

```
Loss, accuracy on test data:
0.3471 88.02%
```





The loss is 0.3471, and the accuracy is 88.02%.

There are two main trade-offs in this experiments. The first trade-off is the increase in accuracy due to the number of hidden layers units verses the space and time complexity. The increase in the units of the hidden layer provides higher accuracy, but as shown in the first table, the number of parameters calculated will increase significantly, thus takes more storage and time for computation. The second trade-off is the learning rate verses time complexity. Large learning rate can learn quickly, but in some cases it's hard to converge. Small learning rate can provide more precise solution, but it's needed to be trained through large number of epochs, which increases the time complexity. Therefore, in choosing the parameter of the neural networks, we need to consider the user needs with these trade-off to determine the suitable configuration.

References:

Vijay, U., 2021. What is epoch and How to choose the correct number of epoch. [online] Medium. Available at: https://medium.com/@upendravijay2/what-is-epoch-and-how-to-choose-the-correct-number-of-epoch-d170656adaaf (https://medium.com/@upendravijay2/what-is-epoch-and-how-to-choose-the-correct-number-of-epoch-d170656adaaf) [Accessed 25 May 2021].

Jordan, J., 2021. Setting the learning rate of your neural network.. [online] Jeremy Jordan. Available at: https://www.jeremyjordan.me/nn-learning-rate/ (https://www.jeremyjordan.me/nn-learning-rate/) [Accessed 25 May 2021].