```
from google.colab import drive
drive.mount('/content/drive')
import os
# start your code here
os.chdir("/content/drive/MyDrive/DL/Homework5") # change your working folder here
# end your code here

     Mounted at /content/drive
```

```
from google.colab import drive
drive.mount('/content/drive')

     Mounted at /content/drive
```

0. Batch size and epochs

When you train your model in TensorFlow, there is a parameter called batch-size, which is quite helpful in accelerating and stabilizing the learning process.

```
model.fit(trainX, trainY, epochs=200, batch_size=64)
```

Here, 'epochs=200' means that you will reuse your data set for 200 times to train the neural network.

'batch_size=64' means that in every epoch you are going to update your weights and bias once every 64 samples. If there is not enough samples left in the last batch, you will use the leftovers to update the weights and bias for the last time in this epoch.

1. Batch, Stochastic, and Minibatch Gradient Descent Method.

1.1 (Batch) Gradient Descent

At the very beginning of this course, we introduced the gradient descent (GD) method, which is for $l = 1, \dots, L$:

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]} \tag{1}$$
$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]} \tag{2}$$

where L is the number of layers and α is the learning rate. For the ith sample, we can compute a derivative $dW^{[l](i)}$ and $db^{[l](i)}$. The derivaties $dW^{[l]}$ and $db^{[l]}$ are the average of $dW^{[l](i)}$ and $db^{[l](i)}$ for all samples. This method is also called the batch gradient descent method. If you want to use (batch) gradient method, the batch_size can be set to the number of samples.

```
model.fit(trainX, trainY, epochs=200, batch_size=trainX.shape[0])
```

1.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) method still use equation (1) and (2) to update the weights and bias. The difference is that instead of using the average gradient of all samples to update the weights and bias once for every epoch, SGD updates the weights and bias once for every sample using the derivatives of that sample. Assuming there are m samples, SGD will update the weights and bias for m times. If SGD is used, the batch_size is set to be 1.

```
model.fit(trainX, trainY, epochs=200, batch_size=1)
```

1.3 Mini-Batch Gradient Descent

Mini-batch gradient descent (mini-batch GD) is something in between. In every epoch, GD updates too few times while SGD updates too many times. Mini-batch GD sets its batch_size somewhere between 1 and m, and generally provides faster training process.

```
model.fit(trainX, trainY, epochs=200, batch_size=64)
```

In Tensorflow, the default value for `batch_size` is 32. If you don't specify the batch_size, your batch_size is 32. The value of it is usually set to be the power of 2 to accommodate the binary property of CPU and GPU.

2. Testing batch_size on A Multi-Class Classification Problem

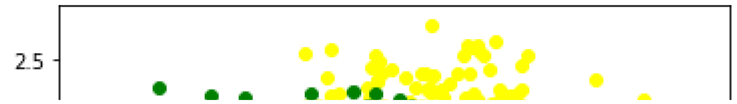
2.1 A Multi-Class Classification Problem

We will use a small multi-class classification problem as the basis to demonstrate the effect of batch size on learning.

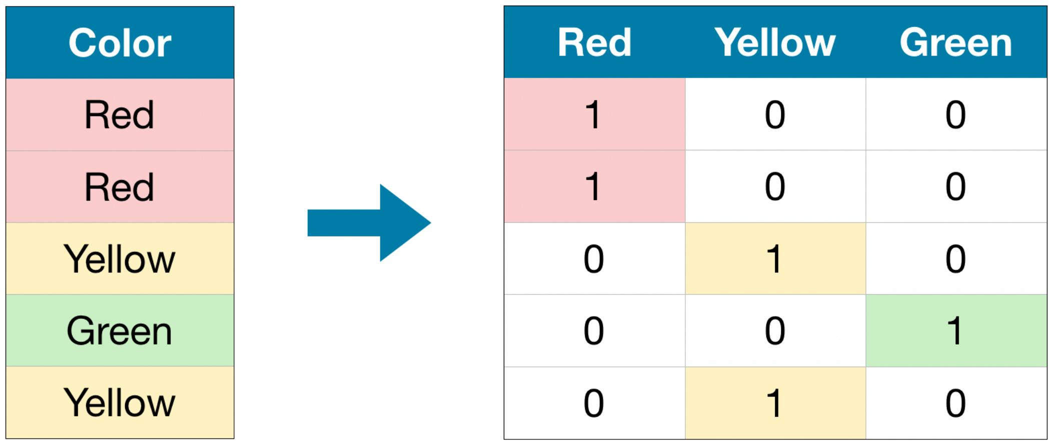
The scikit-learn class provides the `make_blobs()` function that can be used to create a multi-class classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class.

The problem can be configured to have two input variables (to represent the x and y coordinates of the points, 'n_features') and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
import numpy as np
import tensorflow as tf
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2) # 'n_samples: how many data points
# scatter plot for each class value
colors = ["r", "yellow", "g"]
for class_value in range(3):
    # select indices of points with the class label
    row_ix = np.where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1],color=colors[class_value])
# show plot
pyplot.show()
```



For multi-class classification problems, we usually transform the label of every sample (point) y to a one-hot vector. For example, 'red' (y=0) is [1 0 0], 'yellow' (y=1) is [0 1 0], and 'green' (y=2) is [0 0 1]. Because every vector has only 1 non-zero value, it is called a 'one-hot' vector.



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

In tensorflow, we can tranform the labels of samples to one hot vectors by calling

```
tf.keras.utils.to_categorical(y)
```

Please transform the lables (or the output of the dataset) to one hot vectors.

```
# transform multi-class lables to one hot vectors
# start your code here
y_one_hot = tf.keras.utils.to_categorical(y)  # please replace None by a meaningful command
# end your code here
print('The dimension of input data is', X.shape)
print('The dimension of transformed output is', y_one_hot.shape)
print('The 3rd sample is green:',y_one_hot[2,:])
```

```
    The dimension of input data is (1000, 2)
```

```
The dimension of transformed output is (1000, 3)
The 3rd sample is green: [0. 0. 1.]
```

The expected output:

```
The dimension of transformed lables is (1000, 3)
The 3rd sample is green: [0. 0. 1.]
```

2.2 Building neural network

We are going to build a shallow neural network to distinguish the 3 classes. Please use `functional API' to build a nerual network with

1. an input layer with appropriate shape
2. a hidden layer with 50 neurons and activation function 'relu'
3. an output layer with appropriate number of neurons and activation function 'softmax'.

Softmax function is usually used in the output layer in multi-class claasification problems.

```
# start your code here
inputs = tf.keras.Input(shape=2) # please replace None by a meaningful command

hidden = tf.keras.layers.Dense(50,activation=tf.nn.relu)(inputs)   # please replace None by a meaningful command

outputs = tf.keras.layers.Dense(3,activation=tf.nn.softmax)(hidden)   # please replace None by a meaningful command

#end your code here
model=tf.keras.models.Model(inputs=inputs,outputs=outputs,name='shallow_model')
model.summary()
model.save('initial_model')
```

```
Model: "shallow_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_16 (InputLayer)      [(None, 2)]               0

 dense_14 (Dense)           (None, 50)                150

 dense_15 (Dense)           (None, 3)                 153

=================================================================
Total params: 303
Trainable params: 303
Non-trainable params: 0
_________________________________________________________________
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` w
INFO:tensorflow:Assets written to: initial_model/assets
```

Expected output:

```
Model: "shallow_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 2)]               0

 hidden_layer (Dense)        (None, 50)                150

 output_layer (Dense)        (None, 3)                 153

=================================================================
Total params: 303
Trainable params: 303
Non-trainable params: 0
_________________________________________________________________
```

2.3 Compiling and fitting neural network using batch GD

We first use batch GD to train the neurual network. Please make sure that the batch_size is the number of samples in the training data. If you set batch_size greater than the number of samples in the training data, the batch_size will be automatically adjusted to this number.

The model will be trained for 200 epochs, and we split 20% of the data for validation.

```
import time
modelGD=tf.keras.models.load_model('initial_model')
modelGD.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.SGD(learning_rate=0.1), metrics=['accuracy'])
start_time = time.time()
# start your code here
historyGD = modelGD.fit(
    X,                     #please replace None by something meaningful
    y_one_hot,             #please replace None by something meaningful
    validation_split=0.2,
    epochs=200,
    batch_size=X.shape[0]) #please replace None by something meaningful
# end your code here
print("Time taken: %.2fs" % (time.time() - start_time))
```

```
Epoch 170/200
1/1 [==============================] - 0s 41ms/step - loss: 0.4645 - accuracy: 0.8150 - val_loss: 0.3970 - val_accur
Epoch 171/200
1/1 [==============================] - 0s 32ms/step - loss: 0.4681 - accuracy: 0.7950 - val_loss: 0.4275 - val_accur
Epoch 172/200
1/1 [==============================] - 0s 36ms/step - loss: 0.4634 - accuracy: 0.8150 - val_loss: 0.3961 - val_accur
Epoch 173/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4670 - accuracy: 0.7950 - val_loss: 0.4266 - val_accur
Epoch 174/200
1/1 [==============================] - 0s 41ms/step - loss: 0.4624 - accuracy: 0.8150 - val_loss: 0.3953 - val_accur
Epoch 175/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4658 - accuracy: 0.7975 - val_loss: 0.4255 - val_accur
Epoch 176/200
1/1 [==============================] - 0s 33ms/step - loss: 0.4614 - accuracy: 0.8175 - val_loss: 0.3944 - val_accur
Epoch 177/200

1/1 [==============================] - 0s 33ms/step - loss: 0.4646 - accuracy: 0.7975 - val_loss: 0.4245 - val_accur
Epoch 178/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4603 - accuracy: 0.8175 - val_loss: 0.3936 - val_accur
```

```
Epoch 179/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4634 - accuracy: 0.8000 - val_loss: 0.4236 - val_accur
Epoch 180/200
1/1 [==============================] - 0s 57ms/step - loss: 0.4594 - accuracy: 0.8175 - val_loss: 0.3928 - val_accur
Epoch 181/200
1/1 [==============================] - 0s 36ms/step - loss: 0.4624 - accuracy: 0.8012 - val_loss: 0.4227 - val_accur
Epoch 182/200
1/1 [==============================] - 0s 39ms/step - loss: 0.4584 - accuracy: 0.8188 - val_loss: 0.3920 - val_accur
Epoch 183/200
1/1 [==============================] - 0s 38ms/step - loss: 0.4613 - accuracy: 0.8025 - val_loss: 0.4217 - val_accur
Epoch 184/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4574 - accuracy: 0.8188 - val_loss: 0.3912 - val_accur
Epoch 185/200
1/1 [==============================] - 0s 40ms/step - loss: 0.4603 - accuracy: 0.8037 - val_loss: 0.4208 - val_accur
Epoch 186/200
1/1 [==============================] - 0s 32ms/step - loss: 0.4565 - accuracy: 0.8188 - val_loss: 0.3904 - val_accur
Epoch 187/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4593 - accuracy: 0.8050 - val_loss: 0.4199 - val_accur
Epoch 188/200
1/1 [==============================] - 0s 36ms/step - loss: 0.4556 - accuracy: 0.8188 - val_loss: 0.3897 - val_accur
Epoch 189/200
1/1 [==============================] - 0s 31ms/step - loss: 0.4583 - accuracy: 0.8062 - val_loss: 0.4193 - val_accur
Epoch 190/200
1/1 [==============================] - 0s 30ms/step - loss: 0.4548 - accuracy: 0.8188 - val_loss: 0.3890 - val_accur
Epoch 191/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4575 - accuracy: 0.8075 - val_loss: 0.4187 - val_accur
Epoch 192/200
1/1 [==============================] - 0s 33ms/step - loss: 0.4541 - accuracy: 0.8188 - val_loss: 0.3883 - val_accur
Epoch 193/200
1/1 [==============================] - 0s 34ms/step - loss: 0.4568 - accuracy: 0.8075 - val_loss: 0.4181 - val_accur
Epoch 194/200
1/1 [==============================] - 0s 31ms/step - loss: 0.4534 - accuracy: 0.8188 - val_loss: 0.3876 - val_accur
Epoch 195/200
1/1 [==============================] - 0s 45ms/step - loss: 0.4560 - accuracy: 0.8075 - val_loss: 0.4175 - val_accur
Epoch 196/200
1/1 [==============================] - 0s 35ms/step - loss: 0.4527 - accuracy: 0.8188 - val_loss: 0.3869 - val_accur
Epoch 197/200
1/1 [==============================] - 0s 33ms/step - loss: 0.4553 - accuracy: 0.8075 - val_loss: 0.4169 - val_accur
Epoch 198/200
```

Expected output for the 1st 3 epochs:

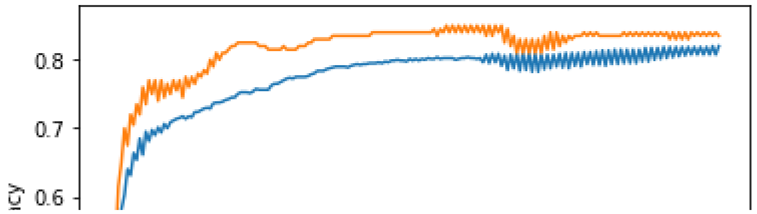
```
Epoch 1/200
1/1 [==============================] - 1s 540ms/step - loss: 2.2440 - accuracy: 0.3363 - val_loss: 1.0965 - val_accuracy: 0.3750
Epoch 2/200
1/1 [==============================] - 0s 37ms/step - loss: 1.1960 - accuracy: 0.3487 - val_loss: 0.8725 - val_accuracy: 0.5550
Epoch 3/200
1/1 [==============================] - 0s 31ms/step - loss: 0.9020 - accuracy: 0.5475 - val_loss: 0.8096 - val_accuracy: 0.5450
...
Time taken: 11.34s
```

Your loss, accuracy, val_loss, and val_accuracy may be different, but we shall see `1/1 [==============================]` appears for every epoch and the time taken is similar. It indicates that you have 1 batch for every epoch.

The final 'accuracy' and 'val_accuracy' should be greater than `0.8`.

Now, let's print out the training history to see how the training process looks like.

```
from matplotlib import pyplot
# plot training history
pyplot.plot(historyGD.history['accuracy'], label='train')
pyplot.plot(historyGD.history['val_accuracy'], label='validation')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.show()
```



2.4 Compiling and fitting neural network using stochastic gradient descent

We will reload the untrained model and train it using stochastic gradient descent method. In this case, we will update the weights and bias once for every sample in the training data. Please review section 1.2 to figure out the batch_size for stochastic gradient descent method.

Again, the model will be trained for 200 epochs, and we split 20% of the data for validation.

```
modelSGD=tf.keras.models.load_model('initial_model')
modelSGD.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.SGD(learning_rate=0.1), metrics=['accuracy']
start_time=time.time()
# start your code here
historySGD = modelSGD.fit(
    X,            #please replace None by something meaningful
    y_one_hot,    #please replace None by something meaningful
    validation_split=0.2,
    epochs=200,
    batch_size=1) #please replace None by something meaningful
# end your code here
print("Time taken: %.2fs" % (time.time() - start_time))
```

```
Epoch 170/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8605 - accuracy: 0.5838 - val_loss: 1.0214 - val_ac
Epoch 171/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8919 - accuracy: 0.6050 - val_loss: 0.9239 - val_ac
Epoch 172/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8679 - accuracy: 0.5800 - val_loss: 0.7153 - val_ac
Epoch 173/200

800/800 [==============================] - 1s 2ms/step - loss: 0.8672 - accuracy: 0.6162 - val_loss: 0.7879 - val_ac
Epoch 174/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9818 - accuracy: 0.5200 - val_loss: 1.0657 - val_ac
```

```
Epoch 175/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9133 - accuracy: 0.5587 - val_loss: 0.9664 - val_ac
Epoch 176/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9637 - accuracy: 0.4875 - val_loss: 1.0758 - val_ac
Epoch 177/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9968 - accuracy: 0.4950 - val_loss: 1.0765 - val_ac
Epoch 178/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9292 - accuracy: 0.5312 - val_loss: 0.9406 - val_ac
Epoch 179/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9097 - accuracy: 0.5600 - val_loss: 1.0426 - val_ac
Epoch 180/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9485 - accuracy: 0.5500 - val_loss: 0.8263 - val_ac
Epoch 181/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8946 - accuracy: 0.5763 - val_loss: 0.8768 - val_ac
Epoch 182/200
800/800 [==============================] - 1s 2ms/step - loss: 0.7914 - accuracy: 0.6500 - val_loss: 0.6954 - val_ac
Epoch 183/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8654 - accuracy: 0.6012 - val_loss: 1.2887 - val_ac
Epoch 184/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9629 - accuracy: 0.5725 - val_loss: 0.9791 - val_ac
Epoch 185/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9916 - accuracy: 0.5100 - val_loss: 0.9466 - val_ac
Epoch 186/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8867 - accuracy: 0.5800 - val_loss: 1.2734 - val_ac
Epoch 187/200
800/800 [==============================] - 1s 2ms/step - loss: 1.0469 - accuracy: 0.4712 - val_loss: 0.9630 - val_ac
Epoch 188/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9405 - accuracy: 0.5163 - val_loss: 1.0595 - val_ac
Epoch 189/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9800 - accuracy: 0.5038 - val_loss: 1.1398 - val_ac
Epoch 190/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9617 - accuracy: 0.5350 - val_loss: 1.0851 - val_ac
Epoch 191/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9446 - accuracy: 0.5400 - val_loss: 0.9825 - val_ac
Epoch 192/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9620 - accuracy: 0.5113 - val_loss: 0.8509 - val_ac
Epoch 193/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9934 - accuracy: 0.5325 - val_loss: 0.9715 - val_ac
Epoch 194/200

800/800 [==============================] - 1s 2ms/step - loss: 0.9483 - accuracy: 0.5250 - val_loss: 0.8260 - val_ac
Epoch 195/200
800/800 [==============================] - 1s 2ms/step - loss: 0.8958 - accuracy: 0.5675 - val_loss: 0.9636 - val_ac
```

```
Epoch 196/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9612 - accuracy: 0.4988 - val_loss: 0.9523 - val_ac
Epoch 197/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9814 - accuracy: 0.5375 - val_loss: 1.0717 - val_ac
```

Expected output for the first 3 epochs:

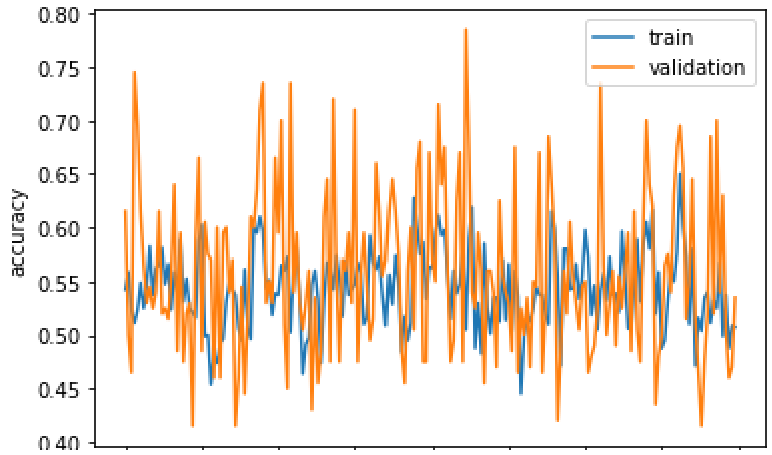
```
Epoch 1/200
800/800 [==============================] - 2s 2ms/step - loss: 0.8803 - accuracy: 0.5938 - val_loss: 1.3032 - val_accuracy: 0.5000
Epoch 2/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9080 - accuracy: 0.5562 - val_loss: 1.0528 - val_accuracy: 0.4600
Epoch 3/200
800/800 [==============================] - 1s 2ms/step - loss: 0.9684 - accuracy: 0.5688 - val_loss: 0.8736 - val_accuracy: 0.5950
...
Time taken: 382.39s
```

It's OK that you have different values in loss, accuracy, val_loss, and val_accuracy. But you should have `800/800 [==============================]` for every epoch. It indicates that you have `800` epochs for every epoch, which is the number of samples in the training data. Your 'Time taken' should be similar to ours without using any accerlerating device (GPU or TPU).

The final accuracy and val_accuracy should be greater than `0.4`.

Let's print the training history out.

```
# plot training history
pyplot.plot(historySGD.history['accuracy'], label='train')
pyplot.plot(historySGD.history['val_accuracy'], label='validation')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.show()
```



2.5 Compiling and fitting neural network using mini-batch gradient descent

We will reload the untrained model again and train it using mini-batch gradient descent method. Let batch_size=32, and see how training process looks like

Again, the model will be trained for 200 epochs, and we split 20% of the data for validation.

```
model32=tf.keras.models.load_model('initial_model')
model32.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.SGD(learning_rate=0.1), metrics=['accuracy'])
start_time=time.time()
# start your code here
history32 = model32.fit(
    X,              #please replace None by something meaningful
    y_one_hot,             #please replace None by something meaningful
    validation_split=0.2,
    epochs=200,
    batch_size=32) #please replace None by something meaningful
# end your code here
print("Time taken: %.2fs" % (time.time() - start_time))
```

```
Epoch 170/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4092 - accuracy: 0.8325 - val_loss: 0.3854 - val_accu

Epoch 171/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4074 - accuracy: 0.8288 - val_loss: 0.3770 - val_accu
Epoch 172/200
```

```
25/25 [==============================] - 0s 3ms/step - loss: 0.4035 - accuracy: 0.8300 - val_loss: 0.3768 - val_accu
Epoch 173/200
25/25 [==============================] - 0s 4ms/step - loss: 0.4086 - accuracy: 0.8275 - val_loss: 0.3491 - val_accu
Epoch 174/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4046 - accuracy: 0.8338 - val_loss: 0.3843 - val_accu
Epoch 175/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4065 - accuracy: 0.8238 - val_loss: 0.3505 - val_accu
Epoch 176/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4054 - accuracy: 0.8200 - val_loss: 0.3738 - val_accu
Epoch 177/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4056 - accuracy: 0.8250 - val_loss: 0.3489 - val_accu
Epoch 178/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4039 - accuracy: 0.8275 - val_loss: 0.3859 - val_accu
Epoch 179/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4023 - accuracy: 0.8350 - val_loss: 0.3791 - val_accu
Epoch 180/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4017 - accuracy: 0.8263 - val_loss: 0.3486 - val_accu
Epoch 181/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4039 - accuracy: 0.8213 - val_loss: 0.3623 - val_accu
Epoch 182/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4103 - accuracy: 0.8225 - val_loss: 0.3471 - val_accu
Epoch 183/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4068 - accuracy: 0.8250 - val_loss: 0.3530 - val_accu
Epoch 184/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4017 - accuracy: 0.8263 - val_loss: 0.3851 - val_accu
Epoch 185/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4038 - accuracy: 0.8300 - val_loss: 0.3403 - val_accu
Epoch 186/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4084 - accuracy: 0.8275 - val_loss: 0.3643 - val_accu
Epoch 187/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4068 - accuracy: 0.8300 - val_loss: 0.3576 - val_accu
Epoch 188/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4096 - accuracy: 0.8250 - val_loss: 0.3956 - val_accu
Epoch 189/200
25/25 [==============================] - 0s 3ms/step - loss: 0.3965 - accuracy: 0.8263 - val_loss: 0.3626 - val_accu
Epoch 190/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4017 - accuracy: 0.8263 - val_loss: 0.3605 - val_accu
Epoch 191/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4012 - accuracy: 0.8313 - val_loss: 0.3550 - val_accu

Epoch 192/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4040 - accuracy: 0.8275 - val_loss: 0.3732 - val_accu
Epoch 193/200
```

```
25/25 [==============================] - 0s 3ms/step - loss: 0.4114 - accuracy: 0.8163 - val_loss: 0.3690 - val_accu
Epoch 194/200
25/25 [==============================] - 0s 3ms/step - loss: 0.3990 - accuracy: 0.8325 - val_loss: 0.4167 - val_accu
Epoch 195/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4034 - accuracy: 0.8288 - val_loss: 0.3752 - val_accu
Epoch 196/200
25/25 [==============================] - 0s 4ms/step - loss: 0.4026 - accuracy: 0.8288 - val_loss: 0.3578 - val_accu
Epoch 197/200
25/25 [==============================] - 0s 3ms/step - loss: 0.4054 - accuracy: 0.8275 - val_loss: 0.3769 - val_accu
Epoch 198/200
```

Expected output:

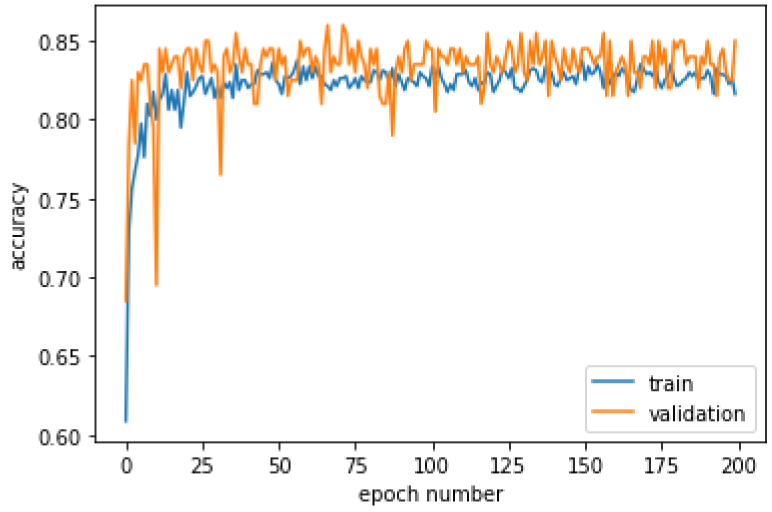
```
Epoch 1/200
25/25 [==============================] - 1s 10ms/step - loss: 0.8676 - accuracy: 0.6388 - val_loss: 0.6800 - val_accuracy: 0.7900
Epoch 2/200
25/25 [==============================] - 0s 3ms/step - loss: 0.6229 - accuracy: 0.7387 - val_loss: 0.5251 - val_accuracy: 0.8400
Epoch 3/200
25/25 [==============================] - 0s 4ms/step - loss: 0.5706 - accuracy: 0.7462 - val_loss: 0.5307 - val_accuracy: 0.8350
...
Time taken: 20.93s
```

You shall have `25/25 [==============================]` for every epoch, though the loss, accuracy, val_loss, val_accuracy might be different. Your 'Time taken' should be in the same level without using any accelerator (GPU or TPU).

The final output need to be greater than `0.8`.

Now, let's print the history out.

```
# plot training history
pyplot.plot(history32.history['accuracy'], label='train')
pyplot.plot(history32.history['val_accuracy'], label='validation')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.show()
```



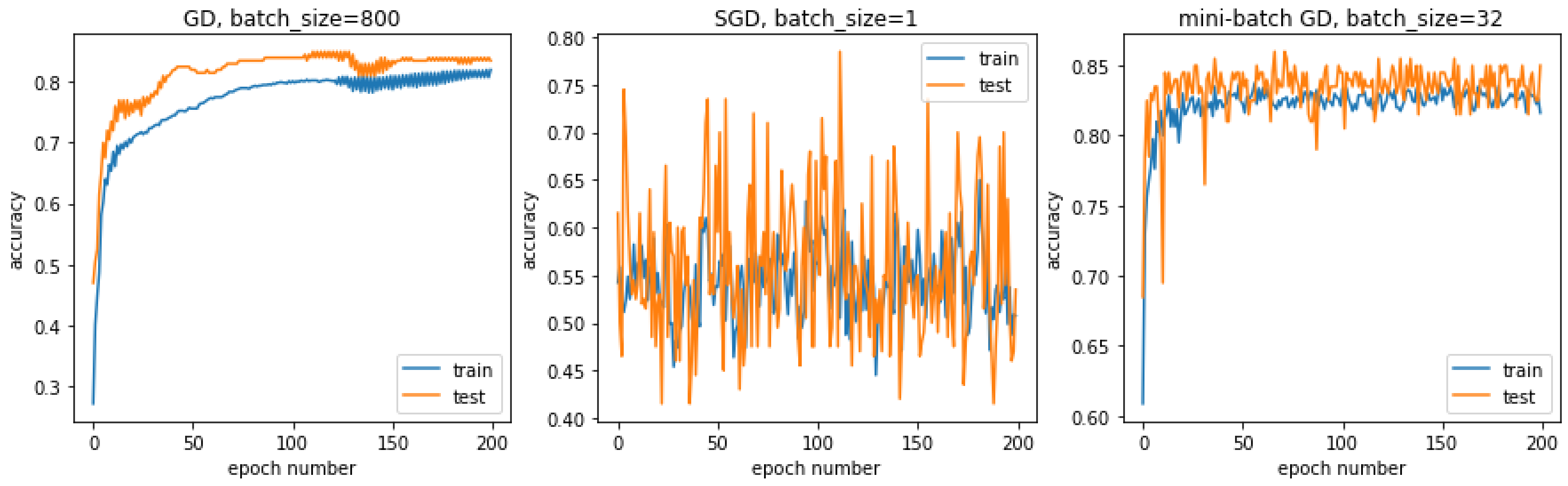
2.6 Conclusion

To give you a more straightforward comparison of the three methods, we print the training histories of three methods all together.

```
pyplot.figure(figsize=(15,4))

pyplot.subplot(1,3,1)
pyplot.plot(historyGD.history['accuracy'], label='train')
pyplot.plot(historyGD.history['val_accuracy'], label='test')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.title('GD, batch_size=800')

pyplot.subplot(1,3,2)
pyplot.plot(historySGD.history['accuracy'], label='train')
pyplot.plot(historySGD.history['val_accuracy'], label='test')
pyplot.title('SGD, batch_size=1')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.subplot(1,3,3)
pyplot.plot(history32.history['accuracy'], label='train')
pyplot.plot(history32.history['val_accuracy'], label='test')
pyplot.title('mini-batch GD, batch_size=32')
pyplot.xlabel('epoch number')
pyplot.ylabel('accuracy')
pyplot.legend()
pyplot.show()
```



Basically, large batch_size means more stable performance, and small (but not too small) batch_size means achieving the same level of performance within less epochs. Meanwhile, we also notice that with large batch_size, we can finish the same number of epochs in less time. In practice, you will need to adjust the batch_size to balance all the affect.

✓ 0s completed at 10:51 PM