DATA and SETTINGS

The following bit of code allows the connection to the google drive cloud storage.

```
# Setting up google drive
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
import sys
sys.path.append('/content/gdrive/MyDrive/Colab Notebooks')
```

Now, the main modules used for the code:

```
import my_utils as mu
import torch
from torch import nn
```

The train and test dataset are now loaded using the *load_data_fashion_mnist()* function in the *my_utils* module (renamed *mu*) and a batch size of 256 samples.

```
# Read training and test data from fashion mnist dataset
batch_size = 256
train_iter, test_iter = mu.load_data_fashion_mnist(batch_size)
```

THE MODEL

The model of the net has one backbone block with two MLPs. Each MLP is made of two linear functions and one activation function, and the layers are implemented in a sequence using *'nn.Sequential'.*

```
class Net(torch.nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
      super(Net, self).__init__()

      self.num_hidden = num_hidden
      self.num_inputs = num_inputs
      self.num_outputs = num_outputs

      self.backbone_mlp1 = nn.Sequential(
                          nn.Linear(num_hidden, num_hidden),
                          nn.ReLU(),
                          nn.Linear(num_hidden, num_hidden))

      self.backbone_mlp2 = nn.Sequential(
                          nn.Linear(num_inputs, num_inputs),
                          nn.ReLU(),
                          nn.Linear(num_inputs, num_inputs))

    def forward(self, x)
      x = Stem(x, 7)

      # Transpose image before first backbone mlp
      x = torch.transpose(x, 1, 2)
```

```
        x = self.backbone_mlp1(x)

        # Transpose image before second backbone mlp
        x = torch.transpose(x, 1, 2)
        x = self.backbone_mlp2(x)

        out = Classifier(x)

        return out
```

The Stem and Classifier functions are implemented in the forward method of the net.

Stem:

The Stem is defined as a class method and is used to split the input image tensor into N number of square patches, vectorise the patches and finally output a feature matrix $X \in R^{Nxd}$, with d = features.

```
    def Stem(self, x, PATCH_SIZE):
        '''
        A function to split the input image into square patches
        and then rearrange the pixels in each patch.
        The input image of size (h, w)
        is rearranged into (num_patches, patch_size^2).

        - PATCH_SIZE is the lenght of the square patch in pixels
        (i.e. 7x7 patch: PATCH_SIZE=7).

        - num_patches is the number of patches in the image,
        it is calculated by dividing the image size by the patch size
        '''
        PATCH_SIZE
        num_patches = (28//PATCH_SIZE)**2
        unfold = nn.Unfold(kernel_size=(7,7), stride=(7,7))

        x_divided = x.unfold(2, PATCH_SIZE, PATCH_SIZE)\
                    .unfold(3, PATCH_SIZE, PATCH_SIZE)

        x_divided = x_divided.reshape( -1, num_patches,\
                                      PATCH_SIZE, PATCH_SIZE)

        return x_divided.reshape(-1, num_patches, PATCH_SIZE**2)
```

Classifier:

The classifier is also declared as a class method to extract the mean feature vector $x \in R^d$ from the feature matrix. The mean features will be directly used for the classification and training process.

```
    def Classifier(self, x):
        '''
        A function to extract the mean feature vector
        from the feature matrix and that will be directly used for the
        classification and training process
        '''
```

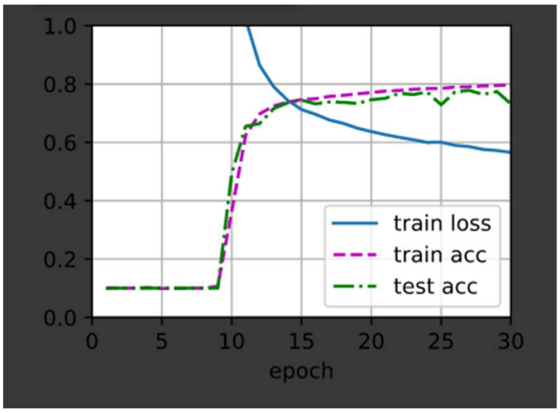
```
        x = x.mean(1, False)
        #print('after mean: ', x.size())

        return x
```

TRAINING SCRIPT

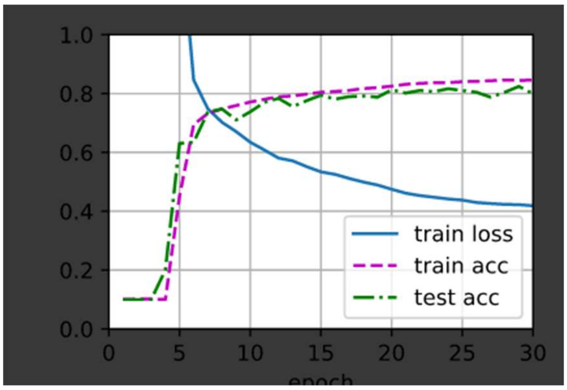
```
# Defined in file: ./chapter_linear-networks/softmax-regression-scratch.md
def train_epoch_ch3(net, train_iter, loss, updater):
    """The training loop defined in Chapter 3."""
    # Set the model to training mode
    if isinstance(net, torch.nn.Module):
        net.train()
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = mu.Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            updater.step()
            metric.add(float(l) * len(y), mu.accuracy(y_hat, y),
                       y.size().numel())
        else:
            l.sum().backward()
            updater(X.shape[0])
            metric.add(float(l.sum()), mu.accuracy(y_hat, y), y.numel())
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]

def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    """Train a model (defined in Chapter 3)."""
    animator = mu.Animator(xlabel='epoch',xlim=[0, num_epochs],ylim=[0, 1],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = mu.evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    print('train final accuracy: ', train_acc)
    print('test final loss: ', train_loss)
    print('test final accuracy: ', train_acc)
```

RESULTS

Several attempt where made adjusting the learning rate (between 0.6 and 0.2) and weight decay (between 0.0002 and 0.0007)



Learning rate = 0.2, weight decay = 0.0005



Learning rate = 0.6, weight decay = 0.0002

The following results were achieved using the following parameters:

- num_inputs, num_hidden, num_outputs = 49, 16, 10
- learning rate = 0.5
- weight decay = 0.0002
- epochs = 30
- loss = nn.CrossEntropyLoss()
- optimizer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)

```
[10] num_epochs = 30
     train_ch3(net, train_iter, test_iter, loss, num_epochs, optimizer)

     train final accuracy:  0.8388166666666667
     test final loss:  0.43833964443206785
     test final accuracy:  0.8137
```