Homework 4

1. The Effect of BatchNorm on a ConvNet

In this exercise, we will combine both the topics we covered in class this week. The goal of this exercise is to visualize the effective smoothness of a covolutional neural network with and without batch normalization.

Let $\varphi(\cdot; w) : \mathbb{R}^{28\times 28} \to \mathbb{R}^{10}$ denote a convolution neural network with parameters w which takes in an image of size 28×28 and returns a score for 10 output classes (All the MLPs and ConvNets we have considered so far fit this input-output description of φ, upto a reshaping of the images). Consider the objective function

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \varphi(x_i; w))$$

where ℓ is the multiclass logistic loss function. We define the **effective local smoothness** of this objective f at some w in a direction u as[1]

$$\widehat{L}(w; u) = \frac{\|\nabla f(w+u) - \nabla f(w)\|_2}{\|u\|_2}.$$

Observe that ∇f is the full gradient computed on the entire dataset. Here, we take u to be a minibatch stochastic gradient update at w, i.e.,

$$u = -\eta \frac{1}{B}\sum_{b=1}^{B} \nabla_w \ell(y_{i_b}, \varphi(x_{i_b}; w))$$

where $i_1, \cdots, i_B$ are random indices sampled from $\{1, \cdots, n\}$, B is the batch size and η is our learning rate.

Concretely, your task is as follows:

- Use the FashionMNIST dataset. Perform the same preprocessing as in previous homeworks.
- Code up a ConvNet module with two convolutional layers with the following structure (the input has 1 channel, so we write the image as 1 × 28 × 28):

$$\begin{array}{c}\text{input}\\ 1\times 28\times 28\end{array} \longrightarrow \begin{array}{c}\text{Conv}\\ k=5, \#\text{filters}=16, \text{padding=2}\end{array} \longrightarrow \text{ReLU} \longrightarrow \begin{array}{c}\text{MaxPool}\\ k=2\end{array} \longrightarrow$$

$$\longrightarrow \begin{array}{c}\text{Conv}\\ k=5, \#\text{filters}=32, \text{padding=2}\end{array} \longrightarrow \text{ReLU} \longrightarrow \begin{array}{c}\text{MaxPool}\\ k=2\end{array} \longrightarrow \begin{array}{c}\text{Linear}\\ \text{output}=10\end{array}$$

Here is how to interpret this specification:

- k denotes the kernel/filter size and “#filters” denotes the number of filters
- In PyTorch, the convolutions and pooling operations on images are called “Conv2d” and “Max- Pool2d” respectively.
- For the first conv layer,the specification asks you to use a kernel size of 5,and a padding of 2.The number of input channels is the same as the number of channels from the preceding layer (here, it is 1 since the preceding layer is just the image with

1 channel). Finally, the number of output channels is the same as the number of filters (here, 16). The second conv layer is constructed in a similarly; the number of input channels is the same as the number of outputs channels of the first conv layer (since ReLU and MaxPool do not change the number of channels). When not specified, we take the stride to be 1 for convolutions and equal to the kernel size for pooling (these are also PyTorch's defaults).
- The last "Linear" layer takes in the output of the second MaxPool and flattens it down to a vector of a certain size S. You are to figure out this size by running a dummy input through these layers and analyzing the output size, as we have done in the lab. The linear layer then maps this S-dimensional input to a 10-dimensional output, one for each class.

Hint: The ConvNet in the demo of week 3 matches this specification. You may use that code as a reference but you have to code up your own.

- Likewise, code up another ConvNet with a batch normalization layer inserted after each MaxPool. Note that use need to use "torch.nn.BatchNorm2d" for images, and this takes in the number of chan- nels as an input.
- Train each ConvNet, with and without batch norm, with a learning rate of 0.04 and a batch size of 32 for 10 passes through the data.
- At the end of each pass, compute the effective local smoothness as we have defined it above.
- Make three plots: the train loss, the test accuracy and the effective local smoothness on the y-axis and the number of passes on the x-axis. Plot both models with and without batch norm on the same plot.
- Comment how the local effective smoothness might explain why we can use larger learning rates with batch normalization. Recall for gradient descent on a L-smooth function, we can use a learning rate up to 1/L.

In [ ]:
```
import torch
import numpy as np
from torchvision.datasets import FashionMNIST
from torch.nn.functional import cross_entropy, relu
import time
import copy
from torch import nn
from torch import optim
import torch.nn.functional as F

import matplotlib.pyplot as plt
%matplotlib inline

torch.manual_seed(0)
np.random.seed(1)
```

Defining helper functions to train the model and loading training and test data

In [ ]:
```python
# download dataset (~117M in size)
train_dataset = FashionMNIST('../../data', train=True, download=False)
X_train = train_dataset.data # torch tensor of type uint8
y_train = train_dataset.targets # torch tensor of type Long
test_dataset = FashionMNIST('../../data', train=False, download=False)
X_test = test_dataset.data
y_test = test_dataset.targets

# choose a subsample of 10% of the data:
idxs_train = torch.from_numpy(
    np.random.choice(X_train.shape[0], replace=False, size=int(X_train.shape
X_train, y_train = X_train[idxs_train], y_train[idxs_train]
# idxs_test = torch.from_numpy(
#     np.random.choice(X_test.shape[0], replace=False, size=X_test.shape[0]/
# X_test, y_test = X_test[idxs_test], y_test[idxs_test]

print(f'X_train.shape = {X_train.shape}')
print(f'n_train: {X_train.shape[0]}, n_test: {X_test.shape[0]}')
print(f'Image size: {X_train.shape[1:]}')

# Normalize dataset: pixel values lie between 0 and 255
# Normalize them so the pixelwise mean is zero and standard deviation is 1

X_train = X_train.float()  # convert to float32
X_train = X_train.view(-1, 784)  # flatten into a (n, d) shape
mean, std = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - mean[None, :]) / (std[None, :] + 1e-6)  # avoid divide

X_test = X_test.float()
X_test = X_test.view(-1, 784)
X_test = (X_test - mean[None, :]) / (std[None, :] + 1e-6)

n_class = np.unique(y_train).shape[0]
```

```
X_train.shape = torch.Size([7200, 28, 28])
n_train: 7200, n_test: 10000
Image size: torch.Size([28, 28])
```

In [ ]:
```python
def compute_objective(model, X, y):
    """ Compute the multinomial logistic loss.
        model is a module
        X of shape (n, d) and y of shape (n,)
    """
    output = model(X)
    return cross_entropy(output, y, reduction='mean')

@torch.no_grad()
def compute_accuracy(model, X, y):
    """ Compute the classification accuracy
        ws is a list of tensors of consistent shapes
        X of shape (n, d) and y of shape (n,)
    """
    is_train = model.training  # if True, model is in training mode
    model.eval()  # use eval mode for accuracy
    score = model(X)
    predictions = torch.argmax(score, axis=1)  # class with highest score is
    if is_train:  # switch back to train mode if appropriate
        model.train()
    return (predictions == y).sum() * 1.0 / y.shape[0]

@torch.no_grad()
def compute_logs(model, verbose=False):
    is_train = model.training  # if True, model is in training mode
    model.eval()  # switch to eval mode
    train_loss = compute_objective(model, X_train, y_train)
    test_loss = compute_objective(model, X_test, y_test)
    train_accuracy = compute_accuracy(model, X_train, y_train)
    test_accuracy = compute_accuracy(model, X_test, y_test)
    if verbose:
        print(('Train Loss = {:.3f}, Train Accuracy = {:.3f}, ' +
               'Test Loss = {:.3f}, Test Accuracy = {:.3f}').format(
                train_loss.item(), train_accuracy.item(),
                test_loss.item(), test_accuracy.item())
    )
    if is_train:  # switch back to train mode if appropriate
        model.train()
    return (train_loss, train_accuracy, test_loss, test_accuracy)
```

In [ ]:
```
def minibatch_sgd_one_pass(model, X, y, learning_rate, batch_size, verbose=F
    model.train()
    num_examples = X.shape[0]
    average_loss = 0.0
    num_updates = int(round(num_examples / batch_size))

    for i in range(num_updates):
        idxs = np.random.choice(X.shape[0], size=(batch_size,)) # draw `batc
        model.train()  # make sure we are in train mode
        # compute the objective.
        objective = compute_objective(model, X[idxs], y[idxs])

        average_loss = 0.99 * average_loss + 0.01 * objective.item()
        if verbose and (i+1) % 100 == 0:
            print(average_loss)

        gradients = torch.autograd.grad(outputs=objective, inputs=model.para

        with torch.no_grad():
            for (w,g) in zip(model.parameters(), gradients):
                w -= learning_rate*g

    return model
```

Defining a CNN without BatchNorm

In [ ]:
```
class SimpleCNN(nn.Module):
    def __init__(self, nclasses: int, use_batchnorm: bool = False):
        super(SimpleCNN, self).__init__()
        self.conv_ensemble_1 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, p
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2))
        self.conv_ensemble_2 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5,
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2))
        if use_batchnorm:
            self.conv_ensemble_1.add_module('BN', nn.BatchNorm2d(16))
            self.conv_ensemble_2.add_module('BN', nn.BatchNorm2d(32))
        self.fully_connected_layer = torch.nn.Linear(1568, out_features=ncla
        # We know that the linear layer is nothing but the flattened output
        # Inspecting the flattened output gave us the answer 1568.

    def forward(self, x, **kwargs): #shape 1x28x28
        x = x.view(-1, 1, 28, 28)  # reshape input; convolutions need a char
        out = self.conv_ensemble_1(x)  # first convolution + relu + pooling
        out = self.conv_ensemble_2(out) # second convolution + relu + poolir
        out = out.view(out.shape[0], -1)  # flatten output
        out = self.fully_connected_layer(out)  # output layer
        return out
```

We used a quick method to identify the input dimensions of the fully connected layers as mentioned in the comments of the SimpleCNN module. We can also calculate the dimensions by going through eachof the layers and identifying the size of inputs and outputs.

For instance,

$$Input = 1 \times 28 \times 28$$

$$Conv1 = \frac{W-F+2P}{S} + 1 = (28 - 5 + 4)/1 + 1 = 28$$

where W = input size, F = filter size, P = Padding, S = Stride *(courtesy CS231n material on CNNs)*. This gives us the dimension of each colvolution filter.

$$MaxPool1 = (28 - 2 + 0)/2 + 1 = 14$$

Similarly, since the stride, padding and kernel sizes are the same for the convolution ensemble 2, we find that the $MaxPool2$ output will be 7*7 (for each filter). We know that there are 32 filters in the second ensemble, therefore the flattened output will be 7*7*32 = 1568

In [ ]:
```
def compute_flat_gradient(obj,model):
    """
    Compute gradient and flatten the output
    """
    grad = torch.autograd.grad(outputs=obj, inputs=model.parameters())
    return grad[0].flatten()

def compute_obj_eval(model):
    """
    Compute objective on full traininig set
    """
    model.eval()
    out = model(X_train)
    return cross_entropy(out, y_train, reduction='mean')

def flatten_params(params):

    weights = torch.FloatTensor()
    for w in params:
        w = w.flatten()
        weights = torch.cat([weights, w])

    return weights
```

In [ ]:

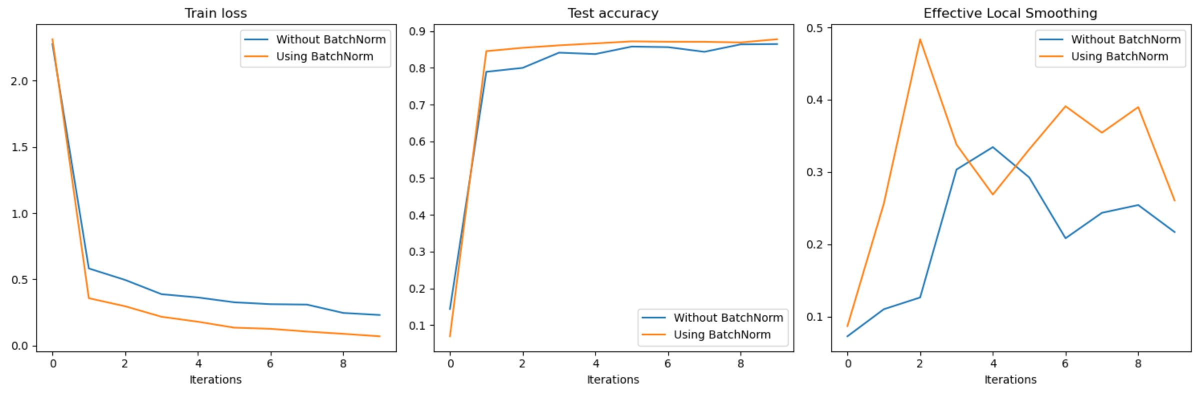
```
#Let us train this model

model = SimpleCNN(nclasses=10)

lr = 0.04
batch_size = 32
epochs = 10

logs = []
els = []

for _ in range(epochs):
    print(f"Iteration {_+1}", end=": ")
    logs.append(compute_logs(model, verbose=True))
    #Computing objective before update
    grad_before = compute_flat_gradient(obj=compute_obj_eval(model), model=m
    params_before = flatten_params(model.parameters())
    model = minibatch_sgd_one_pass(model, X_train, y_train, lr, batch_size,
    params_after = flatten_params(model.parameters())
    #Computing gradients after update
    grad_after = compute_flat_gradient(obj=compute_obj_eval(model), model=mo
    els.append(torch.norm(grad_after - grad_before)/torch.norm(-lr*(params_a
Iteration 1: Train Loss = 2.278, Train Accuracy = 0.145, Test Loss = 2.282,
Test Accuracy = 0.144
Iteration 2: Train Loss = 0.582, Train Accuracy = 0.796, Test Loss = 0.603,
Test Accuracy = 0.789
Iteration 3: Train Loss = 0.495, Train Accuracy = 0.816, Test Loss = 0.556,
Test Accuracy = 0.800
Iteration 4: Train Loss = 0.387, Train Accuracy = 0.858, Test Loss = 0.461,
Test Accuracy = 0.841
Iteration 5: Train Loss = 0.363, Train Accuracy = 0.868, Test Loss = 0.466,
Test Accuracy = 0.837
Iteration 6: Train Loss = 0.326, Train Accuracy = 0.884, Test Loss = 0.417,
Test Accuracy = 0.858
Iteration 7: Train Loss = 0.312, Train Accuracy = 0.887, Test Loss = 0.438,
Test Accuracy = 0.856
Iteration 8: Train Loss = 0.309, Train Accuracy = 0.885, Test Loss = 0.452,
Test Accuracy = 0.844
Iteration 9: Train Loss = 0.245, Train Accuracy = 0.914, Test Loss = 0.405,
Test Accuracy = 0.864
Iteration 10: Train Loss = 0.230, Train Accuracy = 0.919, Test Loss = 0.40
1, Test Accuracy = 0.865
```

Defining a CNN with BatchNorm

In [ ]:
```
#Let us train this model

model = SimpleCNN(nclasses=10, use_batchnorm=True)

lr = 0.04
batch_size = 32
epochs = 10

logs_bn = []
els_bn = []

for _ in range(epochs):
    print(f"Iteration {_+1}", end=": ")
    logs_bn.append(compute_logs(model, verbose=True))
    #Computing objective before update
    grad_before = compute_flat_gradient(obj=compute_obj_eval(model), model=m
    params_before = flatten_params(model.parameters())
    model = minibatch_sgd_one_pass(model, X_train, y_train, lr, batch_size,
    params_after = flatten_params(model.parameters())
    #Computing gradients after update
    grad_after = compute_flat_gradient(obj=compute_obj_eval(model), model=mo
    els_bn.append(torch.norm(grad_after - grad_before)/torch.norm(-lr*(param

    u1 = params_after - lr*grad_after
```

```
Iteration 1: Train Loss = 2.314, Train Accuracy = 0.073, Test Loss = 2.322,
Test Accuracy = 0.070
Iteration 2: Train Loss = 0.357, Train Accuracy = 0.881, Test Loss = 0.471,
Test Accuracy = 0.845
Iteration 3: Train Loss = 0.296, Train Accuracy = 0.904, Test Loss = 0.509,
Test Accuracy = 0.854
Iteration 4: Train Loss = 0.217, Train Accuracy = 0.930, Test Loss = 0.437,
Test Accuracy = 0.861
Iteration 5: Train Loss = 0.179, Train Accuracy = 0.940, Test Loss = 0.447,
Test Accuracy = 0.866
Iteration 6: Train Loss = 0.134, Train Accuracy = 0.957, Test Loss = 0.419,
Test Accuracy = 0.872
Iteration 7: Train Loss = 0.126, Train Accuracy = 0.961, Test Loss = 0.467,
Test Accuracy = 0.871
Iteration 8: Train Loss = 0.105, Train Accuracy = 0.972, Test Loss = 0.456,
Test Accuracy = 0.871
Iteration 9: Train Loss = 0.088, Train Accuracy = 0.973, Test Loss = 0.461,
Test Accuracy = 0.869
Iteration 10: Train Loss = 0.069, Train Accuracy = 0.981, Test Loss = 0.46
3, Test Accuracy = 0.878
```

In [ ]:
```python
logs = np.asarray(logs)
logs_bn = np.asarray(logs_bn)

fig,ax = plt.subplots(1,3, figsize=(15,5))
for a in ax: a.set_xlabel('Iterations');

ax[0].set_title('Train loss');
ax[0].plot(logs[:,0], label="Without BatchNorm")
ax[0].plot(logs_bn[:,0], label="Using BatchNorm")
ax[0].legend()

ax[1].set_title('Test accuracy');
ax[1].plot(logs[:,3], label="Without BatchNorm")
ax[1].plot(logs_bn[:,3], label="Using BatchNorm")
ax[1].legend()

ax[2].set_title('Effective Local Smoothing');
ax[2].plot([el.detach().numpy() for el in els], label="Without BatchNorm")
ax[2].plot([el.detach().numpy() for el in els_bn], label="Using BatchNorm")
ax[2].legend()

plt.tight_layout()
```



2. CNN for Text changes to Convulation Stride

In this exercise, you will train CNN's using different convulation strides and explore how accuracy in a text classification task changes. Concretely, your task is as follows:

- Use the Text REtrietest Conference (TREC) dataset from this weeks demo and lab. Perform the same preprocessing as in the demo and lab to tokenize the input.
- Run a CNN model for 50 epochs using batch size=32, learning rate=0.1, kernel size=3, and padding = 1. Train a six model using these parameters with a convulation stride = 1, 5, 10, 30, 100, and 500 each ten times and average the accuracies over the ten trials (you should train 60 model in total, resulting in 60 test accuracies, 10 per convulation stride).
- Create a bar graph with the convolution stride on the x axis and the accuracies on the y axis.

In [ ]:
```
# Download Text REtrietest Conference (TREC) Question Classifiction data (ht
import requests

# Training Data
URL = "https://cogcomp.seas.upenn.edu/Data/QA/QC/train_1000.label"
TREC_train_data = str(requests.get(URL).content).split("\\n")
train_label = []
train_input = []
for i, t in enumerate(TREC_train_data[:-1]):
    if i ==0:
        train_label.append(t.split(":")[0][2:])
    else:
      train_label.append(t.split(":")[0])
    train_input.append(" ".join(t.split(":")[1].split(" ")[1:]))

train={"input":train_input, "label":train_label}

# Test Data
URL = "https://cogcomp.seas.upenn.edu/Data/QA/QC/TREC_10.label"
TREC_test_data = str(requests.get(URL).content).split("\\n")
test_label = []
test_input = []
for i, t in enumerate(TREC_test_data[:-1]):
    if i ==0:
        test_label.append(t.split(":")[0][2:])
    else:
      test_label.append(t.split(":")[0])
    test_input.append(" ".join(t.split(":")[1].split(" ")[1:]))
test={"input":test_input, "label":test_label}

print("Example of Input/Label")
print("Input:", train_input[1])
print("Label:", train_label[1])

np.unique(list(train['label'])+list(test['label']))
```

```
Example of Input/Label
Input: What films featured the character Popeye Doyle ?
Label: ENTY
```

Out[ ]:
```
array(['ABBR', 'DESC', 'ENTY', 'HUM', 'LOC', 'NUM'], dtype='<U4')
```

In [ ]:
```
# Change string label to integer label
from sklearn import preprocessing
import numpy as np

le = preprocessing.LabelEncoder()
le.fit(train['label']+test['label'])
train['categorical_label'] = le.transform(train['label'])
test['categorical_label'] = le.transform(test['label'])
print("New Label:", train['categorical_label'][1])
print("Nclasses:", len(np.unique(list(train['categorical_label'])+list(test[
# Tokenize the data
from docopt import docopt
import spacy
spacy.load('en_core_web_sm')
nlp = spacy.load('en_core_web_sm')

#  extract features training_data
train_input_vec = [[nlp(v).vector] for v in train['input']]
test_input_vec = [[nlp(v).vector] for v in test['input']]
print("Length of One word Vector:", len(train_input_vec[0][0]))
print("Word Vector Example: ", train_input_vec[1])
```

```
New Label: 2
Nclasses: 6
Length of One word Vector: 96
Word Vector Example:  [array([ 0.0566946 ,  0.14431211, -0.09996037,  0.244
05442,  0.171442  ,
       -0.00698107,  0.5538774 , -0.04801344,  0.10472046, -0.07243   ,
        0.4615469 ,  0.61935836, -0.27421126, -0.36640364, -0.41472915,
       -0.3367687 , -0.80017847,  0.01405566,  0.3789693 , -0.26699215,
        0.06921673,  0.15967804,  0.48105857, -0.5937608 ,  0.54818934,
        0.47652608,  0.5596519 , -0.21928823,  0.08473988,  0.5288324 ,
       -0.47222987, -0.03617571, -0.03886651, -0.4117626 , -0.0573365 ,
        0.0047795 , -0.20796026,  0.08200786,  0.00801056,  0.2975297 ,
       -0.2289368 ,  0.33473474,  0.16553685, -0.12274171, -0.32280177,
       -0.28731257,  0.04599109,  0.05031594,  0.18770942,  0.06129282,
       -0.10349815, -0.06582403,  0.57545674,  0.15352702, -0.22472587,
       -0.04059407,  0.11420991,  0.4052118 ,  0.30882224, -0.26785436,
       -0.24282806,  0.06950301, -0.37544328, -0.21284622,  0.14027737,
       -0.06516401, -0.03447375, -0.0357369 ,  0.52523625, -0.5992119 ,
        0.11967656,  0.18832289,  0.23694097, -0.5269019 , -0.05410558,
       -0.36719215, -0.6547042 , -0.14447653, -0.3175742 , -0.3386358 ,
        0.10537597, -0.18030104, -0.62075025, -0.40350914, -0.35394078,
        0.5281108 ,  0.5338561 , -0.09701845, -0.49369   ,  0.4283861 ,
        0.02411923, -0.29138982,  0.8747733 ,  0.23499991,  0.1764298 ,
        0.06464942], dtype=float32)]
```

In [ ]:
```
from torch import optim
import torch.nn.functional as F

class TextCNN(nn.Module):
    def __init__(self,
                 nclasses:int,
                 linear_dimension:int,
                 kernel_size:int,
                 padding:int,
                 stride:int,
                 window_size: int = 16,
                 embedding_dim: int = 16,
                 filter_multiplier: int = 64):

        super(TextCNN, self).__init__()
        self.simpleconv = nn.Conv1d(in_channels=1, out_channels=filter_multi
                                    kernel_size=kernel_size, padding=padding
        self.maxpool = nn.MaxPool1d(kernel_size=kernel_size, stride=stride,
        self.linear = nn.Linear(in_features=linear_dimension, out_features=n
        self.softmax = nn.LogSoftmax(1)

    def forward(self, x, **kwargs):

        x = self.simpleconv(x)
        x = self.maxpool(x)
        F.relu(x)
        x = x.flatten(start_dim=1)
        x = self.linear(x)
        F.relu(x)
        x = self.softmax(x)

        return x
```

In [ ]:
```
from torch.utils.data import (TensorDataset, DataLoader, RandomSampler,
                              SequentialSampler)

def run_cnn_text(num_classes, batch_size, learning_rate, num_epochs, stride,
  model = TextCNN(num_classes, linear_dimension, kernel_size, padding, stric
  # Create Dataloaders
  train_inputs_tensor, test_inputs_tensor, train_label_tensor, test_label_te
  # Create DataLoader for training data
  train_data = TensorDataset(train_inputs_tensor, train_label_tensor)
  train_sampler = RandomSampler(train_data)
  train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_siz

  # Create DataLoader for testidation data
  test_data = TensorDataset(test_inputs_tensor, test_label_tensor)
  test_sampler = SequentialSampler(test_data)
  test_dataloader = DataLoader(test_data, sampler=test_sampler, batch_size=b


  optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.8)
  lossfn = nn.NLLLoss()
  for epoch in range(num_epochs):
    running_loss = 0.
    for i, batch in enumerate(train_dataloader): # create a new generator ev
      input, label  = b_input_ids, b_labels = tuple(t for t in batch)
      optimizer.zero_grad()
      outputs = model(input)
      loss = lossfn(outputs, label)
      loss.backward()
      optimizer.step()
      running_loss += loss.item()


  model = model.eval()
  num_correct = 0
  num_examples = 0
  # Check accuracy
  for i, batch in enumerate(test_dataloader):
    y, target = batch
    output = model(y)
    y_pred = []
    for o in output:
      y_pred.append(torch.max(o, -1).indices.item()) # Choose class with hig
    y_pred = torch.FloatTensor(y_pred)
    correct = torch.eq(torch.round(y_pred).type(target.type()), target).view
    num_correct += torch.sum(correct).item()
    num_examples += correct.shape[0]

  return num_correct / num_examples
```

We need to figure out the sizes of the linear dimensions for each stride length.

Using the formula $N = \frac{W-F+2P}{S} + 1$

where W = width of input vector of a layer, F = kernel size, P = padding, S = stride, gives dimensions of the output N.

1. **Stride = 1**

   `Input = (batch, 1, 96)`

   `Conv = (batch, 64, 96)`

   `MaxPool = (batch, 64, 96)`

   `Linear = (batch, 6144)`

2. **Stride = 5**

   `Input = (batch, 1, 96)`

   `Conv = (batch, 64, 20)`

   `MaxPool = (batch, 64, 4)`

   `Linear = (batch, 256)`

3. **Stride = 10**

   `Input = (batch, 1, 96)`

`Conv = (batch, 64, 10)`

`MaxPool = (batch, 64, 1)`

`Linear = (batch, 64)`

4. **Stride = 30**

   `Input = (batch, 1, 96)`

   `Conv = (batch, 64, 4)`

   `MaxPool = (batch, 64, 1)`

   `Linear = (batch, 64)`

5. **Stride = 100**

   `Input = (batch, 1, 96)`

   `Conv = (batch, 64, 1)`

   `MaxPool = (batch, 64, 1)`

   `Linear = (batch, 64)`

6. **Stride = 500**

   `Input = (batch, 1, 96)`

   `Conv = (batch, 64, 1)`

   `MaxPool = (batch, 64, 1)`

   `Linear = (batch, 64)`

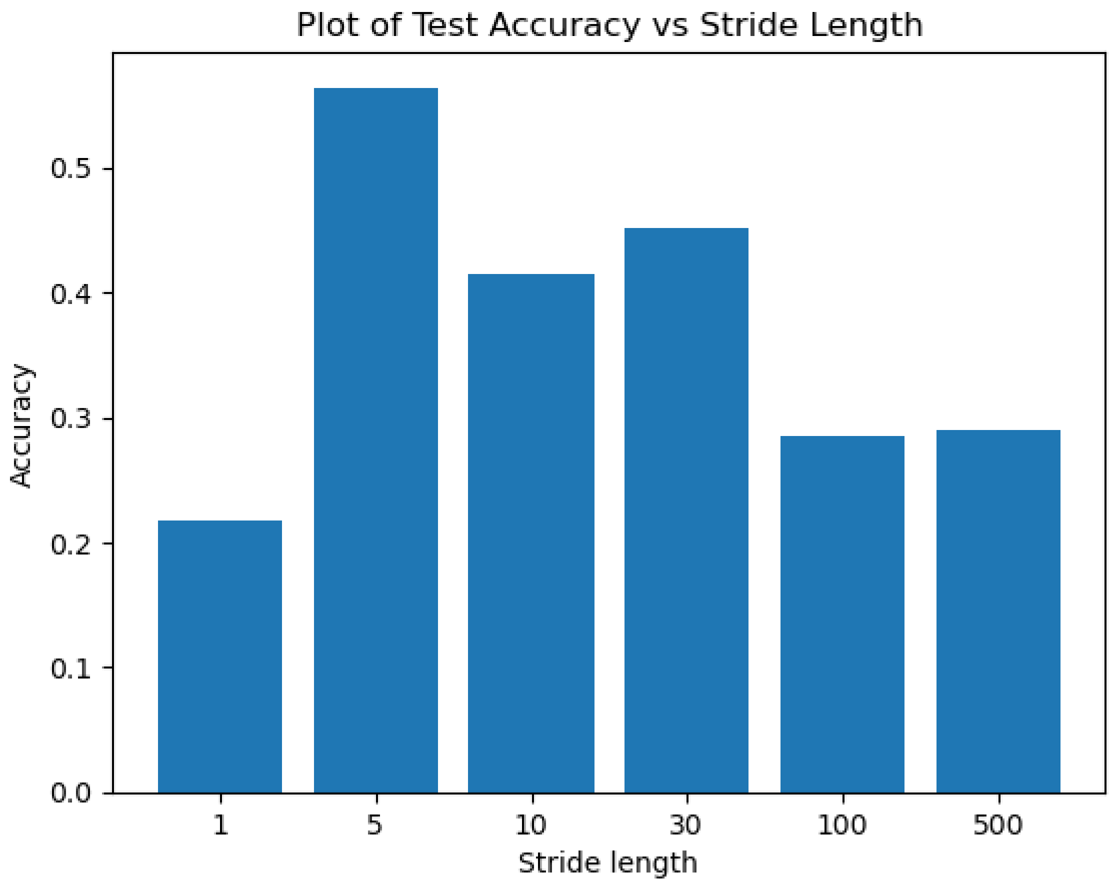
In [ ]:
```
strides = [1,5,10,30,100,500]
linear_dims = [6144,256,64,64,64,64]
```

In [ ]:
```
num_classes = 6
batch_size = 32
learning_rate = 0.1
num_epochs = 50
kernel_size = 3
padding = 1

accuracy = np.zeros((6,10))

for n, (stride, linear_dim) in enumerate(zip(strides,linear_dims)):
    for i in range(10):

        accuracy[n,i] = run_cnn_text(num_classes,
                                     batch_size,
                                        learning_rate,
                                        num_epochs,
                                        stride=stride,
                                        linear_dimension=linear_dim,
                                        kernel_size=kernel_size,
                                        padding=padding)


print(accuracy)
```

```
[[0.67  0.018 0.018 0.018 0.692 0.018 0.018 0.018 0.018 0.684]
 [0.558 0.562 0.564 0.55  0.596 0.55  0.614 0.54  0.546 0.562]
 [0.412 0.436 0.42  0.408 0.444 0.408 0.438 0.4   0.404 0.38 ]
 [0.454 0.468 0.466 0.422 0.488 0.35  0.46  0.48  0.45  0.472]
 [0.246 0.252 0.336 0.318 0.366 0.244 0.306 0.3   0.242 0.244]
 [0.336 0.32  0.232 0.356 0.298 0.37  0.25  0.24  0.254 0.242]]
```

In [ ]:

```
means = {str(s):a for s,a in zip(strides, accuracy.mean(axis=1))}
plt.bar(means.keys(), means.values());
plt.xlabel("Stride length");
plt.ylabel("Accuracy");
plt.title("Plot of Test Accuracy vs Stride Length");
```



3. (Bonus) Max pooling or convolution with stride?

In this exercise, we will compare two alternatives: convolution + max pooling, as considered in Ex- ercise 1, versus a convolution with a stride greater than 1. Throughtout this exercise, we do not use batch norm. Consider the ConvNet

$$\begin{matrix}\text{input} \\ 1\times 28\times 28\end{matrix} \longrightarrow \begin{matrix}\text{Conv} \\ k=5, \#\text{filters}=16, \text{padding=2}, \text{stride=???}\end{matrix} \longrightarrow \text{ReLU} \longrightarrow$$
$$\longrightarrow \begin{matrix}\text{Conv} \\ k=5, \#\text{filters}=32, \text{padding=2 stride=???}\end{matrix} \longrightarrow \text{ReLU} \longrightarrow \begin{matrix}\text{Linear} \\ \text{output}=10\end{matrix}$$

Your tasks are as follows:

- Figure out the right stride so that the input to the second conv layer and the final linear layer are identical in shape to those in the previous exercise.
- Train each ConvNet,the one from Exercise 1 with a learning rate of 0.04 and a batch size of 32 for 10 passes through the data. The rest of the setup, including dataset preprocessing, is identical to Exercise 1.
- Make four plots: the train/test loss/accuracy. Plot both models on the same plot.
- Is there any difference in performance?
- Also report the average time per pass for each of the models. You may use Python's "time" package to keep track of the wallclock time.

Based on these observations, could you speculate why one of the two might be better or worse than the other?

Solution

Based on the results from Exercise 1 we know that -

- Output dimensions of `Conv1 = (batch, 16, 28, 28)`
- Output simensions of `MaxPool1 = (batch, 16, 14, 14)`
- Output dimensions of `Conv2 = (batch, 32, 14, 14)`
- Output simensions of `MaxPool1 = (batch, 32, 7, 7)`

We need to find a stride length such that we get dimensions `(batch, 16, 14, 14)` after Conv1 and `(batch, 32, 7, 7)` after Conv2.

For Conv1,

$$S = \frac{W-F+2P}{N-1} = (28-5+4)/13 = 27/13 \approx 2$$

For Conv2,

$$S = \frac{W-F+2P}{N-1} = (14-5+4)/6 = 13/6 \approx 2$$

```
In [ ]: class MyCNNnoPooling(nn.Module):
            def __init__(self):
                super(MyCNNnoPooling, self).__init__()
```

```
        self.conv_ensemble_1 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, p
            torch.nn.ReLU())
        self.conv_ensemble_2 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5,
            torch.nn.ReLU())
        self.fully_connected_layer = torch.nn.Linear(1568, out_features=10)
        # We know that the linear layer is nothing but the flattened output
        # Inspecting the flattened output gave us the answer 1568.

    def forward(self, x, **kwargs): #shape 1x28x28
        x = x.view(-1, 1, 28, 28)  # reshape input; convolutions need a char
        out = self.conv_ensemble_1(x)  # first convolution + relu + pooling
        out = self.conv_ensemble_2(out) # second convolution + relu + poolir
        out = out.view(out.shape[0], -1)  # flatten output
        out = self.fully_connected_layer(out)  # output layer
        return out
```

In [ ]:
```
import time

model = MyCNNnoPooling()

lr = 0.04
batch_size = 32
epochs = 10

logs_np = []
start = time.time()
for _ in range(epochs):
    print(f"Iteration {_+1}", end=": ")
    logs_np.append(compute_logs(model, verbose=True))
    model = minibatch_sgd_one_pass(model, X_train, y_train, lr, batch_size,

print("Time taken:", round(time.time()-start,2))
```

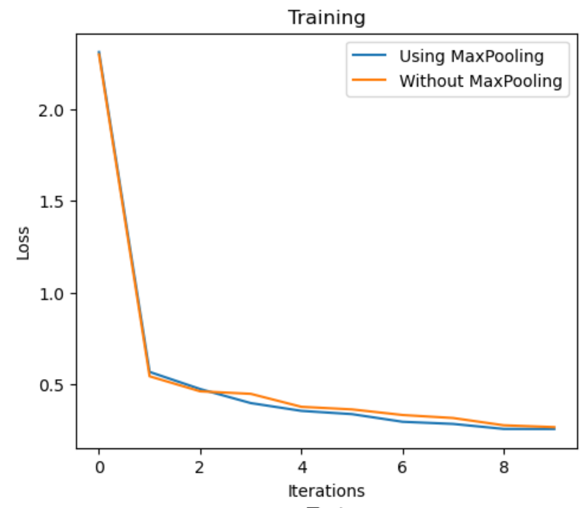
```
Iteration 1: Train Loss = 2.298, Train Accuracy = 0.146, Test Loss = 2.297,
Test Accuracy = 0.148
Iteration 2: Train Loss = 0.543, Train Accuracy = 0.804, Test Loss = 0.579,
Test Accuracy = 0.790
Iteration 3: Train Loss = 0.461, Train Accuracy = 0.837, Test Loss = 0.509,
Test Accuracy = 0.822
Iteration 4: Train Loss = 0.447, Train Accuracy = 0.833, Test Loss = 0.512,
Test Accuracy = 0.814
Iteration 5: Train Loss = 0.376, Train Accuracy = 0.869, Test Loss = 0.458,
Test Accuracy = 0.840
Iteration 6: Train Loss = 0.362, Train Accuracy = 0.870, Test Loss = 0.480,
Test Accuracy = 0.838
Iteration 7: Train Loss = 0.331, Train Accuracy = 0.880, Test Loss = 0.473,
Test Accuracy = 0.840
Iteration 8: Train Loss = 0.315, Train Accuracy = 0.886, Test Loss = 0.461,
Test Accuracy = 0.846
Iteration 9: Train Loss = 0.275, Train Accuracy = 0.902, Test Loss = 0.450,
Test Accuracy = 0.854
Iteration 10: Train Loss = 0.265, Train Accuracy = 0.904, Test Loss = 0.45
1, Test Accuracy = 0.856
Time taken: 73.04
```

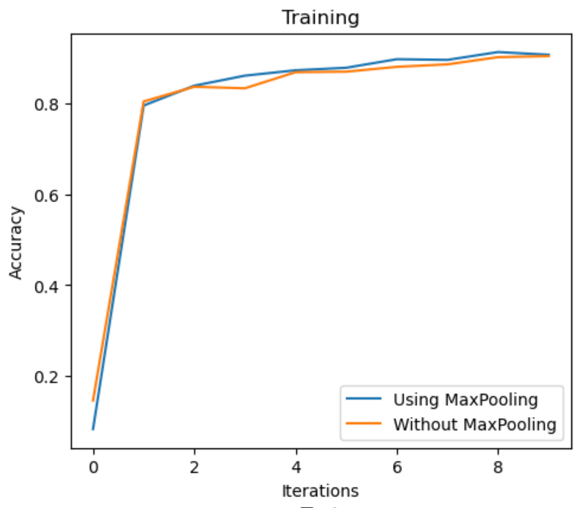
In [ ]:
```python
model = SimpleCNN(nclasses=10)

lr = 0.04
batch_size = 32
epochs = 10

logs = []
start = time.time()
for _ in range(epochs):
    print(f"Iteration {_+1}", end=": ")
    logs.append(compute_logs(model, verbose=True))
    model = minibatch_sgd_one_pass(model, X_train, y_train, lr, batch_size,
print("Time taken:", round(time.time()-start,2))
```

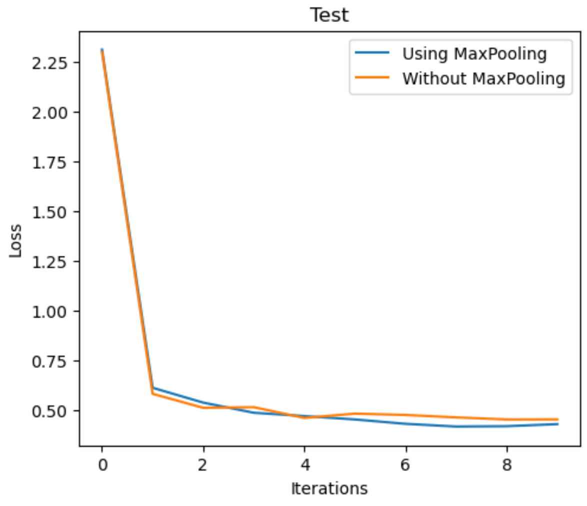
```
Iteration 1: Train Loss = 2.311, Train Accuracy = 0.083, Test Loss = 2.310,
Test Accuracy = 0.085
Iteration 2: Train Loss = 0.566, Train Accuracy = 0.795, Test Loss = 0.610,
Test Accuracy = 0.785
Iteration 3: Train Loss = 0.473, Train Accuracy = 0.839, Test Loss = 0.535,
Test Accuracy = 0.820
Iteration 4: Train Loss = 0.396, Train Accuracy = 0.861, Test Loss = 0.484,
Test Accuracy = 0.833
Iteration 5: Train Loss = 0.354, Train Accuracy = 0.873, Test Loss = 0.468,
Test Accuracy = 0.845
Iteration 6: Train Loss = 0.336, Train Accuracy = 0.878, Test Loss = 0.451,
Test Accuracy = 0.848
Iteration 7: Train Loss = 0.295, Train Accuracy = 0.897, Test Loss = 0.429,
Test Accuracy = 0.857
Iteration 8: Train Loss = 0.283, Train Accuracy = 0.896, Test Loss = 0.415,
Test Accuracy = 0.857
Iteration 9: Train Loss = 0.255, Train Accuracy = 0.913, Test Loss = 0.417,
Test Accuracy = 0.863
Iteration 10: Train Loss = 0.255, Train Accuracy = 0.907, Test Loss = 0.42
7, Test Accuracy = 0.855
Time taken: 139.78
```

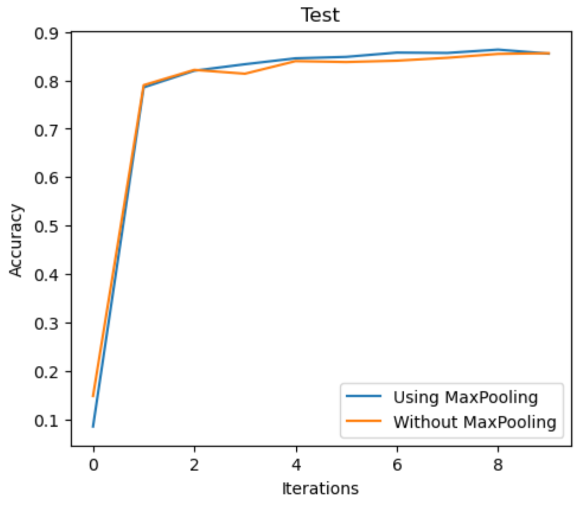
In [ ]:

```
fig, ax = plt.subplots(2,2)
fig.set_size_inches(12,10)
for a in ax.flatten(): a.set_xlabel("Iterations")
for a in ax: a[0].set_ylabel("Loss")
for a in ax: a[1].set_ylabel("Accuracy")
for a in ax[0,:]: a.set_title("Training")
for a in ax[1,:]: a.set_title("Test")

names = zip([logs, logs_np], ["Using MaxPooling", "Without MaxPooling"])
for log, label in names:
    log = np.asarray(log)
    ax[0,0].plot(log[:,0], label = label)
    ax[0,0].legend()
    ax[0,1].plot(log[:,1], label = label)
    ax[0,1].legend()
    ax[1,0].plot(log[:,2], label = label)
    ax[1,0].legend()
    ax[1,1].plot(log[:,3], label = label)
    ax[1,1].legend()
```









There is a marginal difference in performance between the two models with MaxPooling giving sligthly better results. The model with stride=2 and without MaxPooling trains much faster than the model with MaxPooling. The average execution time per training iteration is 7.3s for the model with stride=2 and without MaxPooling, and the average time per training iteration is 13.9s for the model using MaxPooling.

The model with MaxPooling takes longer to train because the convolotion layer has a smaller stride length, which means it captures more information from the image, which is also why the model performas slightly better. Whereas, the model with stride=2 performs convolution and pooling in the same step due to which it trains faster but retains lesser information about the input. Given the tradeoffs are minimal and the gain in performance is substantial, one might perfer a higher stride length instead of MaxPooling, especially when working with limited resources.