Homework 2: Introduction to PyTorch

PyTorch is a framework for creating and training neural networks. It's one of the most common neural network libraries, alongside TensorFlow, and is used extensively in both academia and industry. PyTorch was designed for simplicity -- The code you write is the code that is executed, unlike TensorFlow, and there's very little overhead in terms of reused code. In this homework, we'll explore the basic operations within PyTorch, and we'll design a neural network to classify images.

Let's start by importing the libraries that we'll need:

In [1]:
```
import torch
import torchvision
import torch.nn as nn

import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

If you can't import torch, go to www.pytorch.org (http://www.pytorch.org) and follow the instructions there for downloading PyTorch. You can select CUDA Version as None, as we won't be working with any GPUs on this homework.

PyTorch: Tensors

In PyTorch, data is stored as multidimensional arrays, called tensors. Tensors are very similar to numpy's ndarrays, and they support many of the same operations. We can define tensors by explicity setting the values, using a python list:

In [2]:
```
A = torch.tensor([[1, 2], [4, -3]])
B = torch.tensor([[3, 1], [-2, 3]])

print("A:")
print(A)

print('\n')

print("B:")
print(B)
```

```
A:
tensor([[ 1,  2],
        [ 4, -3]])


B:
tensor([[ 3,  1],
        [-2,  3]])
```

Just like numpy, PyTorch supports operations like addition, multiplication, transposition, dot products, and concatenation of tensors.

In [3]:
```
print("Sum of A and B:")
print(torch.add(A, B))

print('\n')

print("Elementwise product of A and B:")
print(torch.mul(A, B))

print('\n')

print("Matrix product of A and B:")
print(torch.matmul(A, B))

print('\n')

print("Transposition of A:")
print(torch.t(A))

print('\n')

print("Concatenation of A and B in the 0th dimension:")
print(torch.cat((A, B), dim=0))

print('\n')

print("Concatenation of A and B in the 1st dimension:")
print(torch.cat((A, B), dim=1))
```

```
Sum of A and B:
tensor([[4, 3],
        [2, 0]])


Elementwise product of A and B:
tensor([[ 3,  2],
        [-8, -9]])


Matrix product of A and B:
tensor([[-1,  7],
        [18, -5]])


Transposition of A:
tensor([[ 1,  4],
        [ 2, -3]])


Concatenation of A and B in the 0th dimension:
tensor([[ 1,  2],
        [ 4, -3],
        [ 3,  1],
        [-2,  3]])


Concatenation of A and B in the 1st dimension:
```

```
tensor([[ 1,  2,  3,  1],
        [ 4, -3, -2,  3]])
```

PyTorch also has tools for creating large tensors automatically, without explicity specifying the values:

In [4]:
```
print("3x4x5 Tensor of Zeros:")
print(torch.zeros(3, 4, 5))

print('\n')

print("5x5 Tensor with random elements sampled from a standard normal distrubtion:
print(torch.randn(5, 5))

print('\n')

print("Tensor created from a range:")
print(torch.arange(10))
```

```
3x4x5 Tensor of Zeros:
tensor([[[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]]])


5x5 Tensor with random elements sampled from a standard normal distrubtion:
tensor([[-0.1883, -1.3210,  1.4432,  1.0341,  0.2128],
        [-1.3903,  1.1469,  0.3570, -1.5392, -0.1707],
        [ 0.5931, -0.1459,  0.9271, -0.9253, -0.0581],
        [ 0.2566, -2.0578, -1.1957,  0.0302, -0.8827],
        [ 0.9766, -0.1762, -0.1773, -0.1481, -0.0236]])


Tensor created from a range:
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Now, use PyTorch tensors to complete the following computation:

Create a tensor of integers from the range 0 to 99, inclusive. Add 0.5 to each element in the tensor, and square each element of the result. Then, negate each element of the tensor, and apply the exponential to each element (i.e., change each element x into e^x). Now, sum all the elements of the tensor and print your result.

If you're right, you should get something very close to

$$\frac{1}{2} \cdot \sqrt{\pi} \approx 0.8862.$$

In [5]:
```
val = torch.arange(100).float()

### <YOUR CODE HERE> ####
print(val[:5])
val = val + 0.5;
print(val[:5])
val = val ** 2;
print(val[:5])
val = -val;
print(val[:5])
val = np.exp(val);
print(val[:5])
val = torch.sum(val);
### </YOUR CODE HERE> ###

print(val)
```

```
tensor([0., 1., 2., 3., 4.])
tensor([0.5000, 1.5000, 2.5000, 3.5000, 4.5000])
tensor([ 0.2500,  2.2500,  6.2500, 12.2500, 20.2500])
tensor([ -0.2500,  -2.2500,  -6.2500, -12.2500, -20.2500])
tensor([7.7880e-01, 1.0540e-01, 1.9305e-03, 4.7851e-06, 1.6052e-09])
tensor(0.8861)
```

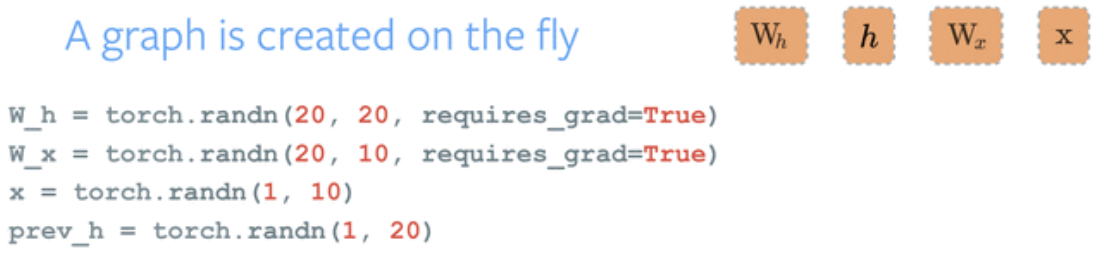
To do this, you'll need to use the PyTorch documentation at https://pytorch.org/docs/stable/torch.html (https://pytorch.org/docs/stable/torch.html). Luckily, PyTorch has very well-written docs.

PyTorch: Autograd

Autograd is PyTorch's automatic differentiation tool: It allows us to compute gradients by keeping track of all the operations that have happened to a tensor. In the context of neural networks, we'll interpret these gradient calculations as backpropagating a loss through a network.

To understand how autograd works, we first need to understand the idea of a **computation graph**. A computation graph is a directed, acyclic graph (DAG) that contains a blueprint of a sequence of operations. For a neural network, these computations consist of matrix multiplications, bias additions, ReLUs, softmaxes, etc. Nodes in this graph consist of the operations themselves, while the edges represent tensors that flow forward along this graph.

In PyTorch, the creation of this graph is **dynamic**. This means that tensors themselves keep track of their own computational history, and this history is build as the tensors flow through the network; this is unlike TensorFlow, where an external controller keeps track of the entire computation graph. This dynamic creation of the computation graph allows for lots of cool control-flows that are not possible (or at least very difficult) in TensorFlow.



Dynamic computation graphs are cool!

_ _

Let's take a look at a simple computation to see what autograd is doing. First, let's create two tensors and add them together. To signal to PyTorch that we want to build a computation graph, we must set the flag requires_grad to be True when creating a tensor.

In [6]:
```
a = torch.tensor([1, 2], dtype=torch.float, requires_grad=True)
b = torch.tensor([8, 3], dtype=torch.float, requires_grad=True)

c = a + b
```

Now, since a and b are both part of our computation graph, c will automatically be added:

In [7]:
```
c.requires_grad
```

Out[7]:
```
True
```

When we add a tensor to our computation graph in this way, our tensor now has a grad_fn attribute. This attribute tells autograd how this tensor was generated, and what tensor(s) this particular node was created from.

In the case of c, its grad_fn is of type AddBackward1, PyTorch's notation for a tensor that was created by adding two tensors together:

In [8]:
```
c.grad_fn
```

Out[8]:
```
<ThAddBackward at 0x20bd1b98c88>
```

Every grad_fn has an attribute called next_functions: This attribute lets the grad_fn pass on its gradient to the tensors that were used to compute it.

In [9]:
```
c.grad_fn.next_functions
```

Out[9]:
```
((<AccumulateGrad at 0x20bd1bf20f0>, 0),
 (<AccumulateGrad at 0x20bd1bf2b38>, 0))
```

If we extract the tensor values corresponding to each of these functions, we can see a and b!

In [10]:
```
print(c.grad_fn.next_functions[0][0].variable)
print(c.grad_fn.next_functions[1][0].variable)
```

```
tensor([1., 2.], requires_grad=True)
tensor([8., 3.], requires_grad=True)
```

In this way, autograd allows a tensor to record its entire computational history, implicitly creating a computational graph -- All dynamically and on-the-fly!

PyTorch: Modules and Parameters

In PyTorch, collections of operations are encapsulated as **modules**. One way to visualize a module is to take a section of a computational graph and collapse it into a single node. Not only are modules useful for encapsulation, they have the ability to keep track of tensors that are contained inside of them: To do this, simply wrap a tensor with the class torch.nn.Parameter.

To define a module, we must subclass the type torch.nn.Module. In addition, we must define a *forward* method that tells PyTorch how to traverse through a module.

For example, let's define a logistic regression module. This module will contain two parameters: The weight vector and the bias. Calling the *forward* method will output a probability between zero and one.

In [11]:
```
class LogisticRegression(nn.Module):

    def __init__(self):

        super().__init__()
        self.weight = nn.Parameter(torch.randn(10))
        self.bias = nn.Parameter(torch.randn(1))
        self.sigmoid = nn.Sigmoid()

    def forward(self, vector):
        return self.sigmoid(torch.dot(vector, self.weight) + self.bias)
```

Note that we have fixed the dimension of our weight to be 10, so our module will only accept 10-

dimensional data.

We can now create a random vector and pass it through the module:

In [12]:
```
module = LogisticRegression()
vector = torch.randn(10)
output = module(vector)
```

In [13]:
```
output
```

Out[13]:
```
tensor([0.0122], grad_fn=<SigmoidBackward>)
```

Now, say that our loss function is mean-squared-error and our target value is 1. We can then write our loss as:

In [14]:
```
loss = (output - 1) ** 2
```

In [15]:
```
loss
```

Out[15]:
```
tensor([0.9757], grad_fn=<PowBackward0>)
```

To minimize this loss, we just call loss.backward(), and all the gradients will be computed for us! Note that wrapping a tensor as a Parameter will automatically set requires_grad = True.

In [16]:
```
loss.backward()
```

In [17]:
```
print(module.weight.grad)
print(module.bias.grad)
```

```
tensor([-0.0103, -0.0066, -0.0586,  0.0296, -0.0111,  0.0222, -0.0075,  0.007
0,
         0.0192, -0.0358])
tensor([-0.0239])
```

Fully-connected Networks for Image Classification

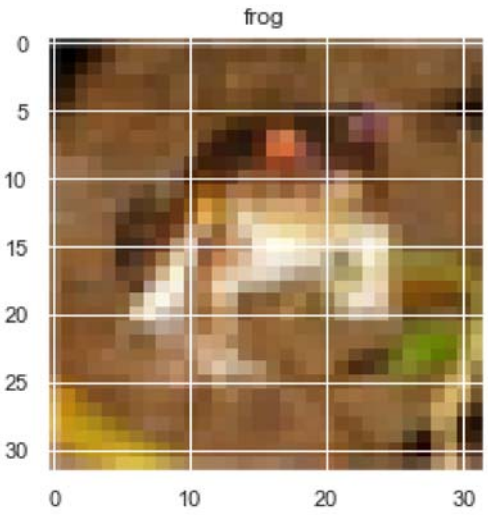
Using this knowledge, you will create a neural network in PyTorch for image classification on the CIFAR-10 dataset.

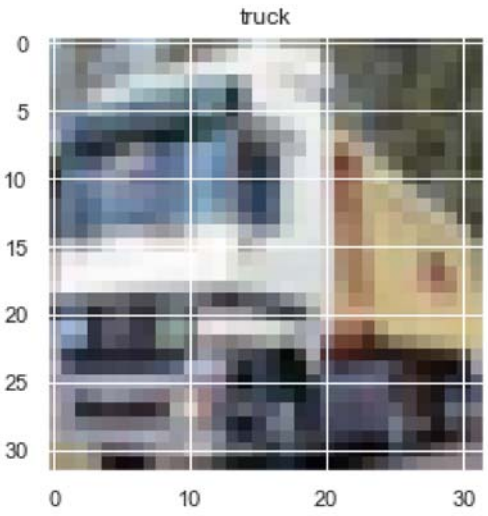
In [18]:
```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)
trainset = [(np.asarray(image) / 256, label) for image, label in trainset]
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4096, shuffle=True)

val_and_test_set = torchvision.datasets.CIFAR10(root='./data', train=False, downlo
val_and_test_set = [(np.asarray(image) / 256, label) for image, label in val_and_t

valset = val_and_test_set[:5000]
valset = (
    torch.stack([torch.tensor(pair[0]) for pair in valset]),
    torch.tensor([pair[1] for pair in valset])
)
testset = val_and_test_set[5000:]
testset = (
    torch.stack([torch.tensor(pair[0]) for pair in testset]),
    torch.tensor([pair[1] for pair in testset])
)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
```

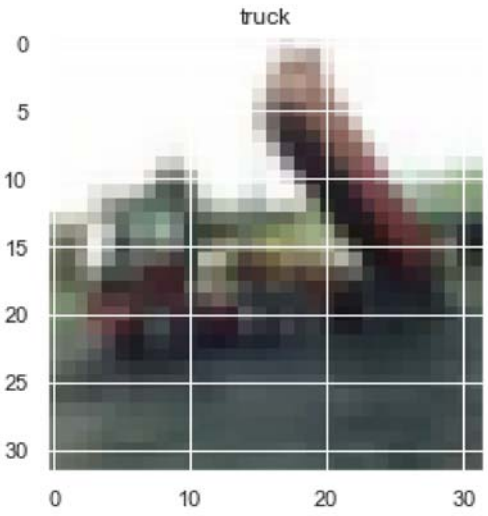
```
Files already downloaded and verified
Files already downloaded and verified
```

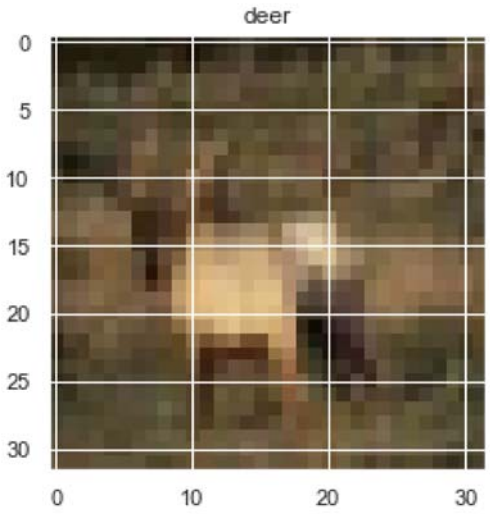
CIFAR-10 consists of 32 x 32 color images, each corresponding to a unique class indicating the object present within the image. Here are a few examples:

In [19]:
```
for image, label in trainset[:4]:
    plt.title(classes[label])
    plt.imshow(image)
    plt.show()
```









We've already split the dataset into training, validation, and test sets for you.

Your assignment is to create and train a neural network that properly classifies images in the CIFAR-10 dataset. You should achieve above 40% classification accuracy on the test set in order to receive full credit on this homework.

We've given you some starter code to achieve this task, but the rest is up to you. Google is your friend -- Looking things up on the PyTorch docs and on StackOverflow will be helpful.

To turn in the assignment, convert this notebook to a PDF (File -> Download As -> PDF via LaTeX) and submit to Gradescope.

In [23]:
```
class NeuralNet(nn.Module):

    def __init__(self, layer_sizes):

        super().__init__()

        ### <YOUR CODE HERE> ####
        self.fc1 = nn.Linear(layer_sizes[0], layer_sizes[1])
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(layer_sizes[1], layer_sizes[2])
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(layer_sizes[2], layer_sizes[3])
        ### </YOUR CODE HERE> ###

#         self.conv1 = nn.Conv2d(3, 6, 5)
#         self.relu1 = nn.ReLU()
#         self.pool1 = nn.MaxPool2d(2, 2)
#         self.conv2 = nn.Conv2d(6, 16, 5)
#         self.relu2 = nn.ReLU()
#         self.pool2 = nn.MaxPool2d(2, 2)
#         self.fc1 = nn.Linear(16 * 5 * 5, 120)
#         self.relu3 = nn.ReLU()
#         self.fc2 = nn.Linear(120, 84)
#         self.relu4 = nn.ReLU()
#         self.fc3 = nn.Linear(84, 10)

    def forward(self, images):

        ### <YOUR CODE HERE> ####
        x = self.fc1(images)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x
        ### </YOUR CODE HERE> ###

#         x = self.conv1(x)
#         x = self.relu1(x)
#         x = self.pool1(x)
#         x = self.conv2(x)
#         x = self.relu2(x)
#         x = self.pool2(x)
#         x = x.view(-1, 16 * 5 * 5)
#         x = self.fc1(x)
#         x = self.relu3(x)
#         x = self.fc2(x)
#         x = self.relu4(x)
#         x = self.fc3(x)
        return x
```

In [21]:
```
def reshape(images):
    '''
    Reshapes a set of images of the shape (batch_size, width, height, channels)
    into the proper shape (batch_size, width * height * channels) that the model c
    '''
    return images.reshape(images.shape[0], -1).float()
```

In [31]:
```
EPOCHS = 1000
LEARNING_RATE = 1e-5
HIDDEN_LAYER_SIZES = [32 * 32 * 3, 128, 32, 10]

net = NeuralNet(HIDDEN_LAYER_SIZES)
optimizer = torch.optim.Adam(net.parameters(), lr=LEARNING_RATE)
loss_fn = nn.CrossEntropyLoss()

print(net)

for epoch in range(EPOCHS):

    average_loss = 0

    for images, labels in trainloader:

        images = reshape(images)
        output = net(images)
        loss = loss_fn(output, labels)

        ### <YOUR CODE HERE> ####
        # Zero gradients, call .backward(), and step the optimizer.
        loss.backward()
        optimizer.step()
        ### </YOUR CODE HERE> ###

        average_loss += loss.item()

    average_loss /= len(trainloader)

    val_output = net(reshape(valset[0]))
    val_loss = loss_fn(val_output, valset[1]).item()

    print("(epoch, train_loss, val_loss) = ({0}, {1}, {2})".format(epoch, average_
```

```
6)
(epoch, train_loss, val_loss) = (990, 1.616645941367516, 1.610254168510437)
(epoch, train_loss, val_loss) = (991, 1.6141158434060903, 1.608218908309936
5)
(epoch, train_loss, val_loss) = (992, 1.613827842932481, 1.605301618576049
8)
(epoch, train_loss, val_loss) = (993, 1.60841350372021, 1.6017875671386719)
(epoch, train_loss, val_loss) = (994, 1.6019188532462487, 1.59797525405883
8)
(epoch, train_loss, val_loss) = (995, 1.6026384005179772, 1.594157576560974
1)
(epoch, train_loss, val_loss) = (996, 1.5965158205765944, 1.590595722198486
3)
(epoch, train_loss, val_loss) = (997, 1.590667697099539, 1.587579488754272
5)
(epoch, train_loss, val_loss) = (998, 1.5889038489415095, 1.585368514060974
1)
(epoch, train_loss, val_loss) = (999, 1.5834742417702308, 1.584109067916870
1)
```

In [32]:
```
### Here, we test the overall accuracy of our model. ###
test_output = net(reshape(testset[0]))
test_maxes = torch.argmax(test_output, dim=1)
print("Test accuracy:", torch.sum(test_maxes == testset[1]).item() / float(test_ma
```

```
Test accuracy: 0.4332
```

In [ ]:
```

```