CS 189 HW 6: Neural Networks

Note: before starting this notebook, please make a copy of it, otherwise your changes will not persist.

This part of the assignment is designed to get you familiar with how engineerings in the real world train neural network systems. It isn't designed to be difficult. In fact, everything you need to complete the assignment is available directly on the pytorch website here (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html). This note book will have the following components:

1. Understanding the basics of Pytorch (no deliverables)
2. Training a simple neural network on MNIST (Deliverable = training graphs)
3. Train a model on CIFAR-10 for Kaggle (Deliverable = kaggle submission and explanation of methods)

The last part of this notebook is left open for you to explore as many techniques as you want to do as well as possible on the dataset.

You will also get practice being an ML engineer by reading documentation and using it to implement models. The first section of this notebook will cover an outline of what you need to know -- we are confident that you can find the rest on your own.

Note that like all other assignments, you are free to use this notebook or not. You just need to complete the deliverables and turn in your code. If you want to run everything outside of the notebook, make sure to appropriately install pytorch to download the datasets and copy out the code for kaggle submission. If you don't want to use pytorch and instead want to use Tensorflow, feel free, but you may still need to install pytorch to download the datasets.

In [1]:
```
# Imports for pytorch
import numpy as np
import torch
import torchvision
from torch import nn
import matplotlib
from matplotlib import pyplot as plt
import tqdm
```

1. Understanding Pytorch

Pytorch is based on the "autograd" paradigm. Essentially, you perform operations on multi-dimensional arrays like in numpy, except pytorch will automatically handle gradient tracking. In this section you will understand how to use pytorch.

This section should help you understand the full pipeline of creating and training a model in pytorch. Feel free to re-use code from this section in the assigned tasks.

Content in this section closely follows this pytorch tutorial:
https://pytorch.org/tutorials/beginner/basics/intro.html
(https://pytorch.org/tutorials/beginner/basics/intro.html)

Tensors

Tensors can be created from numpy data or by using pytorch directly.

In [2]:
```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

np_array = np.array(data)
x_np = torch.from_numpy(np_array)

shape = (2,3,)
rand_tensor = torch.rand(shape)
np_rand_array = rand_tensor.numpy()

print(f"Tensor from np: \n {x_np} \n")
print(f"Rand Tensor: \n {rand_tensor} \n")
print(f"Rand Numpy Array: \n {np_rand_array} \n")
```

```
Tensor from np:
 tensor([[1, 2],
        [3, 4]])

Rand Tensor:
 tensor([[0.5726, 0.3143, 0.4400],
        [0.7205, 0.5531, 0.6039]])

Rand Numpy Array:
 [[0.5725672  0.31425166 0.43998164]
 [0.72045094 0.5531376  0.60389876]]
```

They also support slicing and math operations very similar to numpy. See the examples below:

In [3]:
```
# Slicing
tensor = torch.ones(4, 4)
print('First row: ',tensor[0])
print('First column: ', tensor[:, 0])

# Matrix Operations
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

# Getting a single item
scalar = torch.sum(y1) # sums all elements
item = scalar.item()
print("Sum as a tensor:", scalar, ", Sum as an item:", item)
```

```
First row:  tensor([1., 1., 1., 1.])
First column:  tensor([1., 1., 1., 1.])
Sum as a tensor: tensor(64.) , Sum as an item: 64.0
```

Autograd

This small section shows you how pytorch computes gradients. When we create tenors, we can set `requires_grad` to be true to indicate that we are using gradients. For most of the work that you actually do, you will use the `nn` package, which automatically sets all parameter tensors to have `requires_grad=True` .

In [4]:
```
# Below is an example of computing the gradient for a single data point in logist

x = torch.ones(5)  # input tensor
y = torch.zeros(1) # label
w = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
pred = torch.sigmoid(torch.matmul(x, w) + b)
loss = torch.nn.functional.binary_cross_entropy(pred, y)
loss.backward() # Computers gradients
print("W gradient:", w.grad)
print("b gradient:", b.grad)

# when we want to actually take an update step, we can use optimizers:
optimizer = torch.optim.SGD([w, b], lr=0.1)
print("Weight before", w)
optimizer.step() # use the computed gradients to update
# Print updated weights
print("Updated weight", w)

# Performing operations with gradients enabled is slow...
# You can disable gradient computation using the following enclosure:
with torch.no_grad():
    # Perform operations without gradients
    ...
```

```
W gradient: tensor([[0.3149],
        [0.3149],
        [0.3149],
        [0.3149],
        [0.3149]])
b gradient: tensor([0.3149])
Weight before tensor([[ 0.6470],
        [ 0.3009],
        [-0.2313],
        [ 1.3777],
        [-0.8343]], requires_grad=True)
Updated weight tensor([[ 0.6155],
        [ 0.2694],
        [-0.2628],
        [ 1.3462],
        [-0.8658]], requires_grad=True)
```

Devices

Pytorch supports accelerating computation using GPUs which are available on google colab. To use a GPU on google colab, go to runtime -> change runtime type -> select GPU.

Note that there is some level of strategy for knowing when to use which runtime type. Colab will kick users off of GPU for a certain period of time if you use it too much. Thus, its best to run simple models and prototype to get everything working on CPU, then switch the instance type over to GPU for training runs and parameter tuning.

Its best practice to make sure your code works on any device (GPU or CPU) for pytorch, but note that numpy operations can only run on the CPU. Here is a standard flow for using GPU acceleration:

In [5]:
```
# Determine the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)
# Next create your tensors
tensor = torch.zeros(4, 4, requires_grad=True)
# Move the tensor to the device you want to use
tensor = tensor.to(device)

# Perform whatever operations you want.... (often this will involve gradients)
# These operations will be accelerated by GPU.
tensor = 10*(tensor + 1)

# bring the tensor back to CPU, first detaching it from any gradient computations
tensor = tensor.detach().cpu()

tensor_np = tensor.numpy() # Convert to numpy if you want to perform numpy operat
```

```
Using device cpu
```

The NN Package

Pytorch implements composable blocks in `Module` classes. All layers and modules in pytorch inherit from `nn.Module`. When you make a module you need to implement two functions: `__init__(self, *args, **kwargs)` and `foward(self, *args, **kwargs)`. Modules also have some nice helper functions, namely `parameters` which will recursively return all of the parameters. Here is an example of a logistic regression model:

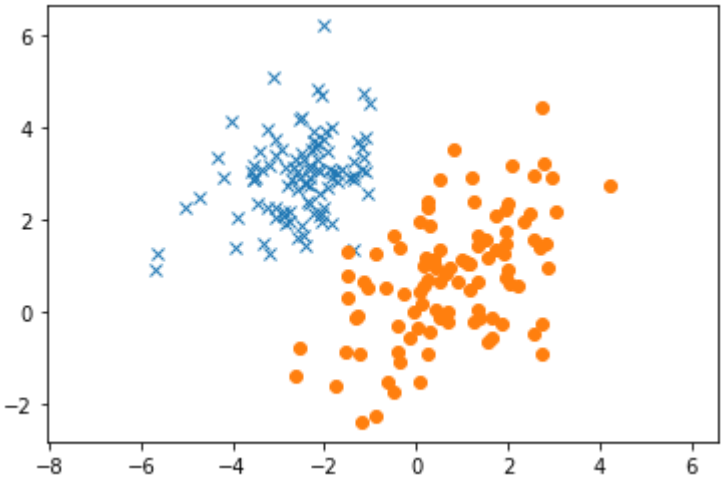
In [ ]:
```
class Perceptron(nn.Module):
  def __init__(self, in_dim):
    super().__init__()
    self.layer = nn.Linear(in_dim, 1) # This is a linear layer, it computes Xw +

  def forward(self, x):
    return torch.sigmoid(self.layer(x)).squeeze(-1)

perceptron = Perceptron(10)
perceptron = perceptron.to(device) # Move all the perceptron's tensors to the dev
print("Parameters", list(perceptron.parameters()))
```

```
Parameters [Parameter containing:
tensor([[-0.2404,  0.0982, -0.1174, -0.2526, -0.3120,  0.2454, -0.0026, -0.190
7,
         -0.2991,  0.2950]], requires_grad=True), Parameter containing:
tensor([0.0202], requires_grad=True)]
```

Datasets

Pytorch has nice interfaces for using datasets. Suppose we create a logistic regression dataset as follows:

In [ ]:
```
c1_x1, c1_x2 = np.random.multivariate_normal([-2.5,3], [[1, 0.3],[0.3, 1]], 100).
c2_x1, c2_x2 = np.random.multivariate_normal([1,1], [[2, 1],[1, 2]], 100).T
c1_X = np.vstack((c1_x1, c1_x2)).T
c2_X = np.vstack((c2_x1, c2_x2)).T
train_X = np.concatenate((c1_X, c2_X))
train_y = np.concatenate((np.zeros(100), np.ones(100)))
# Shuffle the data
permutation = np.random.permutation(train_X.shape[0])
train_X = train_X[permutation, :]
train_y = train_y[permutation]
# Plot the data
plt.plot(c1_x1, c1_x2, 'x')
plt.plot(c2_x1, c2_x2, 'o')
plt.axis('equal')
plt.show()
```



We can then create a pytorch dataset object as follows. Often times, the default pytorch datasets will create these objects for you. Then, we can apply dataloaders to iterate over the dataset in batches.

In [ ]:
```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(train_X), torch.from_nu
# We can create a dataloader that iterates over the dataset in batches.
dataloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True)
for x, y in dataloader:
    print("Batch x:", x)
    print("Batch y:", y)
    break

# Clean up the dataloader as we make a new one later
del dataloader
```

```
Batch x: tensor([[ 1.9860,  2.3458],
        [-2.3392,  2.3675],
        [-1.2057,  3.1054],
        [ 1.9862,  0.9026],
        [ 1.1984,  2.9069],
        [-3.3596,  3.1102],
        [ 2.2123,  0.5725],
        [-3.5328,  2.8846],
        [-0.2561,  0.4125],
        [-2.0352,  3.7572]], dtype=torch.float64)
Batch y: tensor([1., 0., 0., 1., 1., 0., 1., 0., 1., 0.], dtype=torch.float64)
```

Training Loop Example

Here is an example of training a full logistic regression model in pytorch. Note the extensive use of modules -- modules can be used for storing networks, computation steps etc.

In [ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

epochs = 10
batch_size = 10
learning_rate = 0.01

num_features = dataset[0][0].shape[0]
model = Perceptron(num_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
criterion = torch.nn.BCELoss()
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=

model.train() # Put model in training mode
for epoch in range(epochs):
    training_losses = []
    for x, y in tqdm.notebook.tqdm(dataloader, unit="batch"):
        x, y = x.float().to(device), y.float().to(device)
        optimizer.zero_grad() # Remove the gradients from the previous step
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        training_losses.append(loss.item())
    print("Finished Epoch", epoch + 1, ", training loss:", np.mean(training_losse

# We can run predictions on the data to determine the final accuracy.
with torch.no_grad():
    model.eval() # Put model in eval mode
    num_correct = 0
    for x, y in dataloader:
        x, y = x.float().to(device), y.float().to(device)
        pred = model(x)
        num_correct += torch.sum(torch.round(pred) == y).item()
    print("Final Accuracy:", num_correct / len(dataset))
    model.train() # Put model back in train mode
```

```
Using device cpu
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


Finished Epoch 1 , training loss: 0.8894232884049416
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


Finished Epoch 2 , training loss: 0.7322969049215317
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


Finished Epoch 3 , training loss: 0.6252537339925766
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))


Finished Epoch 4 , training loss: 0.5479712948203087
```

```
HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 5 , training loss: 0.48872431069612504

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 6 , training loss: 0.44216412454843523

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 7 , training loss: 0.4052919313311577

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 8 , training loss: 0.37566838040947914

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 9 , training loss: 0.35128752812743186

HBox(children=(FloatProgress(value=0.0, max=20.0), HTML(value='')))

Finished Epoch 10 , training loss: 0.3315702099353075
Final Accuracy: 0.895
```

Task 1: MLP For FashionMNIST

Earlier in this course you trained SVMs and GDA models on MNIST. Now you will train a multi-layer perceptron model on an MNIST-like dataset. Your deliverables are as follows:

1. Code for training an MLP on MNIST (can be in code appendix, tagged in your submission).
2. A plot of the training loss and validation loss for each epoch of training after trainnig for at least 8 epochs.
3. A plot of the training and validation accuracy, showing that it is at least 82% for validation by the end of training.

Below we will create the training and validation datasets for you, and provide a very basic skeleton of the code. Please leverage the example training loop from above.

Some pytorch components you should definetily use:

1. `nn.Linear`
2. Some activation: `nn.ReLU`, `nn.Tanh`, `nn.Sigmoid`, etc.
3. `nn.CrossEntropyLoss`

Here are challenges you will need to overcome:

1. The data is default configured in image form ie (28 x 28), versus one feature vector. You will need to reshape it somewhere to feed it in as vector to the MLP. There are many ways of doing this.

2. You need to write code for plotting.
3. You need to find appropriate hyper-parameters to achieve good accuracy.

Your underlying model must be fully connected or dense, and may not have convolutions etc., but you can use anything in torch.optim or any layers in torch.nn besides nn.Linear that do not have weights.

In [6]:
```
# Creating the datasets
#transform = torchvision.transforms.ToTensor() # feel free to modify this as you
import torchvision.transforms as transforms

training_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=transforms.Compose([transforms.ToTensor()]),
)

validation_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=transforms.Compose([transforms.ToTensor()]),
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz)
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz) to data/FashionMNIST/raw/train-images-idx3-ubyte.gz

HBox(children=(FloatProgress(value=0.0, max=26421880.0), HTML(value='')))


Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz)
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz) to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

HBox(children=(FloatProgress(value=0.0, max=29515.0), HTML(value='')))


Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz)
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz) to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

HBox(children=(FloatProgress(value=0.0, max=4422102.0), HTML(value='')))


Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-l
```

```
abels-idx1-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.c
om/t10k-labels-idx1-ubyte.gz)
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-l
abels-idx1-ubyte.gz (http://fashion-mnist.s3-website.eu-central-1.amazonaws.c
om/t10k-labels-idx1-ubyte.gz) to data/FashionMNIST/raw/t10k-labels-idx1-ubyt
e.gz
```

```
HBox(children=(FloatProgress(value=0.0, max=5148.0), HTML(value='')))
```

```
Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/FashionMNIS
T/raw

Processing...
Done!

/usr/local/lib/python3.7/dist-packages/torchvision/datasets/mnist.py:502: UserW
arning: The given NumPy array is not writeable, and PyTorch does not support no
n-writeable tensors. This means you can write to the underlying (supposedly non
-writeable) NumPy array using the tensor. You may want to copy the array to pro
tect its data or make it writeable before converting it to a tensor. This type
of warning will be suppressed for the rest of this program. (Triggered internal
ly at  /pytorch/torch/csrc/utils/tensor_numpy.cpp:143.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

In [7]:
```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
from torch.autograd import Variable

### YOUR CODE HERE ###
batch_size = 100

# Create data loaders.
train_loader = DataLoader(training_data, batch_size=batch_size)
test_loader = DataLoader(validation_data, batch_size=batch_size)
```

In [ ]:
```
len(validation_data)
```

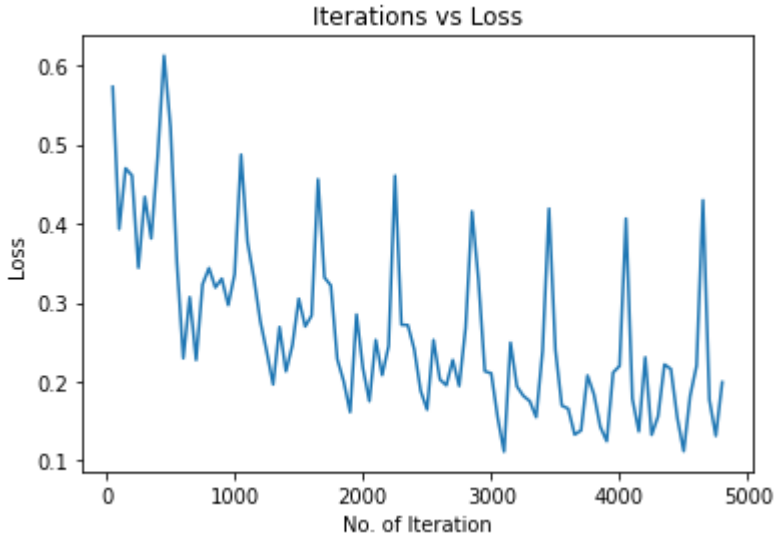
Out[5]:
```
10000
```

In [8]:
```
class FashionCNN(nn.Module):

    def __init__(self):
        super(FashionCNN, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.fc1 = nn.Linear(in_features=64*6*6, out_features=600)
        self.drop = nn.Dropout2d(0.25)
        self.fc2 = nn.Linear(in_features=600, out_features=120)
        self.fc3 = nn.Linear(in_features=120, out_features=10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.drop(out)
        out = self.fc2(out)
        out = self.fc3(out)

        return out
```

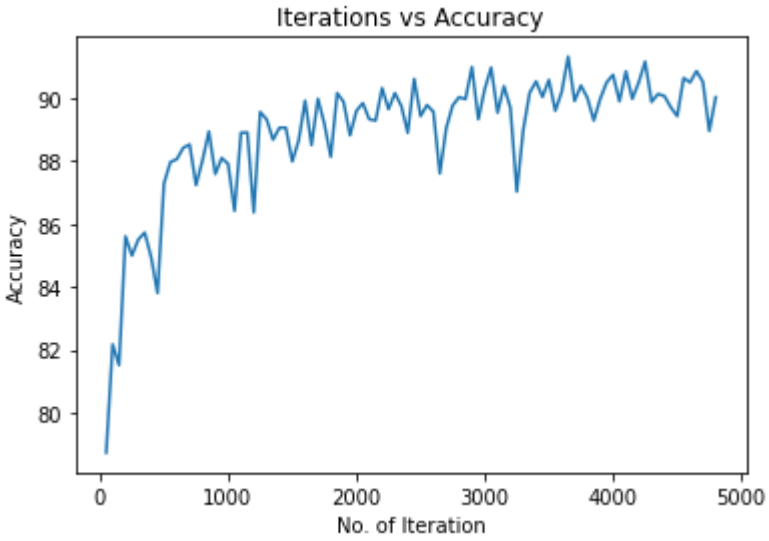
In [9]:
```
model = FashionCNN()
model.to(device)

error = nn.CrossEntropyLoss()

learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [10]:
```
num_epochs = 8
count = 0

loss_list = []
iteration_list = []
accuracy_list = []
predictions_list = []
labels_list = []

for epoch in range(num_epochs):
    for images, labels in train_loader:

        images, labels = images.to(device), labels.to(device)

        train = Variable(images.view(100,1,28,28))
        labels = Variable(labels)

        outputs = model(train)
        loss = error(outputs, labels)
        optimizer.zero_grad()
        loss.backward()

        optimizer.step()

        count += 1

        if count % 50 == 0:
            total = 0
            correct = 0

            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
                labels_list.append(labels)

                test = Variable(images.view(100, 1, 28, 28))

                outputs = model(test)

                predictions = torch.max(outputs, 1)[1].to(device)
                predictions_list.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
```

In [11]:
```
plt.plot(iteration_list, loss_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Loss")
plt.title("Iterations vs Loss")
plt.show()
```



In [12]:
```
plt.plot(iteration_list, accuracy_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Accuracy")
plt.title("Iterations vs Accuracy")
plt.show()
```



In [ ]:
```

```

Task 2: CNNs for CIFAR-10

In this section, you will create a CNN for the CIFAR dataset, and submit your predictions to Kaggle. It is recommended that you use GPU acceleration for this part.

Here are some of the components you should consider using:

1. `nn.Conv2d`
2. `nn.ReLU`
3. `nn.Linear`
4. `nn.CrossEntropyLoss`
5. `nn.MaxPooling2d` (though many implementations without it exist)

We encourage you to explore different ways of improving your model to get higher accuracy. Here are some suggestions for things to look into:

1. CNN architectures: AlexNet, VGG, ResNets, etc.
2. Different optimizers and their parameters (see torch.optim)
3. Image preprocessing / data augmentation (see torchvision.transforms)
4. Regularization or dropout (see torch.optim and torch.nn respectively)
5. Learning rate scheduling: https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate (https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate)
6. Weight initialization: https://pytorch.org/docs/stable/nn.init.html (https://pytorch.org/docs/stable/nn.init.html)

Though we encourage you to explore, there are some rules:

1. You are not allowed to install or use packages not included by default in the Colab Environment.
2. You are not allowed to use any pre-defined architectures or feature extractors in your network.
3. You are not allowed to use **any** pretrained weights, ie no transfer learning.
4. You cannot train on the test data.

Otherwise everything is fair game.

Your deliverables are as follows:

1. Submit to Kaggle and include your test accuracy in your report.
2. Provide at least (1) training curve for your model, depicting loss per epoch or step after training for at least 8 epochs.
3. Explain the components of your final model, and how you think your design choices contributed to it's performance.

After you write your code, we have included skeleton code that should be used to submit predictions to Kaggle. **You must follow the instructions below under the submission header**. Note that if you apply any processing or transformations to the data, you will need to do the same to the test data otherwise you will likely achieve very low accuracy.

It is expected that this task will take a while to train. Our simple solution achieves a training accuracy of 90.2% and a test accuracy of 74.8% after 10 epochs (be careful of overfitting!). This easily beats the best SVM based CIFAR10 model submitted to the HW 1 Kaggle! It is possible to achieve 95% or higher test accuracy on CIFAR 10 with good model design and tuning.

In [6]:
```
# Creating the datasets, feel free to change this as long as you do the same to t
# You can also modify this to split the data into training and validation.
# See https://pytorch.org/docs/stable/data.html#torch.utils.data.random_split

#transform = torchvision.transforms.ToTensor()
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
from torch.autograd import Variable

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

training_data = torchvision.datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=transform,
)

dataset_size = len(training_data)
val_size = 5000
train_size = len(training_data) - val_size

train_ds, val_ds = torch.utils.data.random_split(training_data, [train_size, val_

trainloader = DataLoader(training_data, batch_size=4, shuffle=True, num_workers=2
val_loader = DataLoader(val_ds, batch_size=8, num_workers=2)


# If you make a train-test partition it is up to you.
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://ww
w.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) to data/cifar-10-python.tar.gz

HBox(children=(FloatProgress(value=0.0, max=170498071.0), HTML(value='')))


Extracting data/cifar-10-python.tar.gz to data
```

In [12]:
```
AlexNet_Model = torch.hub.load('pytorch/vision:v0.6.0', 'alexnet', pretrained=Tru
AlexNet_Model.eval()
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.6.0
```

Out[12]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

In [13]:
```
import torch.nn as nn
#AlexNet_Model.classifier[1] = nn.Linear(9216,4096)
AlexNet_Model.classifier[4] = nn.Linear(4096,1024)
AlexNet_Model.classifier[6] = nn.Linear(1024,10)
```

In [14]:
```
AlexNet_Model.eval()
```

Out[14]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=1024, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1024, out_features=10, bias=True)
  )
)
```

In [16]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
AlexNet_Model.to(device)
```

Out[16]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=1024, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1024, out_features=10, bias=True)
  )
)
```

In [17]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(AlexNet_Model.parameters(), lr=0.001, momentum=0.9)
```

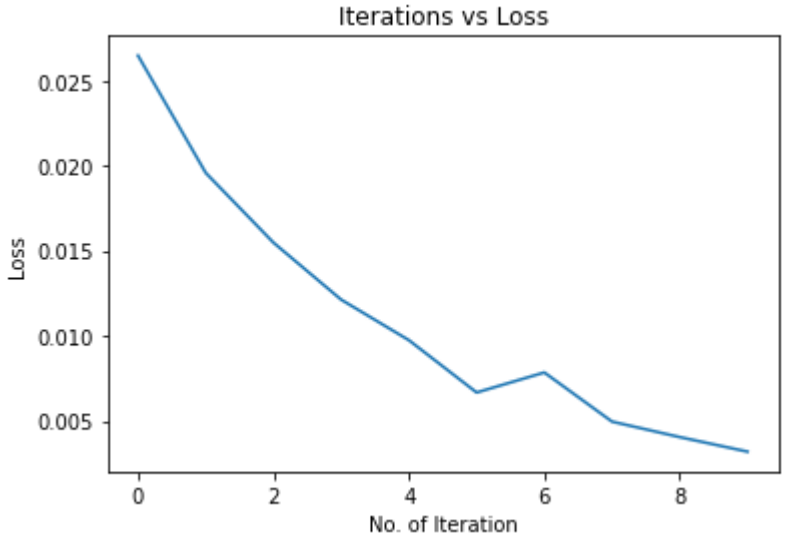
In [18]:
```
#device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

loss_list = []
for epoch in range(10):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):

        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        output = AlexNet_Model(inputs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

    loss_list.append(running_loss/i)
```

```
Using device cuda
[1,  2000] loss: 1.202
Time: 35.5198655128479
[1,  4000] loss: 0.906
Time: 70.96676397323608
[1,  6000] loss: 0.813
Time: 106.16651844978333
[1,  8000] loss: 0.749
Time: 141.55083227157593
[1, 10000] loss: 0.683
Time: 177.02404928207397
[1, 12000] loss: 0.665
Time: 212.3732464313507
[2,  2000] loss: 0.520
Time: 35.33420252799988
[2,  4000] loss: 0.500
Time: 70.77471399307251
[2,  6000] loss: 0.512
Time: 106.07242512702942
```

In [19]:
```
#Testing Accuracy
correct = 0
total = 0

with torch.no_grad():
    for data in val_loader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = AlexNet_Model(images)
        _, predicted = torch.max(outputs.data, 1)
        #preds.append(predicted)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print((100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 98.32 %
```

In [20]:
```
plt.plot(range(10), loss_list)
plt.xlabel("No. of Iteration")
plt.ylabel("Loss")
plt.title("Iterations vs Loss")

plt.show()
```



In [ ]:

In [ ]:

Kaggle Submission

The following code is for you to make your submission to kaggle. Here are the steps you must follow:

1. Upload `cifar_test_data.npy` to the colab notebook by going to files on the right hand pane, then hitting "upload".

2. Run the following cell to generate the dataset object for the test data. Feel free to modify the code to use the same transforms that you use for the training data. By default, this will re-use the `transform` variable.
3. In the second cell, write code to run predictions on the testing dataset and store them into an array called `predictions` .
4. Run the final cell which will convert your predictions array into a CSV for kaggle.
5. Go to the files pane again, and download the file called `submission.csv` by clicking the three dots and then download.

In [23]:
```
test_path = "data/cifar_test_data.npy"
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils, datasets, models


class CustomTensorDataset(Dataset):

    def __init__(self, tensors, transform=None, train=False):
        assert all(tensors[0].size(0) == tensor.size(0) for tensor in tensors)
        self.tensors = tensors
        self.transform = transform


    def __len__(self):
        return self.tensors[0].size(0)

    def __getitem__(self, index):
        x = self.tensors[0][index]

        if self.transform:
            x = self.transform(x)


        return x

def npy_loader(path):
    sample = torch.from_numpy(np.load(path))
    return sample

#dataset = datasets.ImageFolder(npy_loader(test_path), transform=transform)
#dataset = torch.utils.data.TensorDataset(npy_loader(test_path))

dataset = CustomTensorDataset(npy_loader(test_path), transform=transform)
# We can create a dataloader that iterates over the dataset in batches.
testloader = DataLoader(dataset, batch_size=8, num_workers=2)
```

In [24]:
```
from PIL import Image
import os
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
class CIFAR10Test(torchvision.datasets.VisionDataset):

    def __init__(self, transform=None, target_transform=None):
        super(CIFAR10Test, self).__init__(None, transform=transform,
                                          target_transform=target_transform)
        assert os.path.exists("cifar_test_data.npy"), "You must upload the test d
        self.data = [np.load("cifar_test_data.npy", allow_pickle=False)]

        self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
        self.data = self.data.transpose((0, 2, 3, 1))  # convert to HWC

    def __getitem__(self, index: int):
        img = self.data[index]
        img = Image.fromarray(img)
        if self.transform is not None:
            img = self.transform(img)
        return img

    def __len__(self) -> int:
        return len(self.data)

# Create the test dataset
testing_data = CIFAR10Test(
    transform=transform, # NOTE: Make sure transform is the same as used in the t
)
```

In [25]:
```
### YOUR CODE HERE ###
testloader = DataLoader(testing_data)
# Recommendation: create a `test_dataloader` from torch.utils.data.DataLoader wit


#Testing Accuracy

preds = []
with torch.no_grad():
    for data in testloader:
        image = data.to(device)
        AlexNet_Model.cuda()
        outputs = AlexNet_Model(image)
        _, predicted = torch.max(outputs.data, 1)
        preds.append(predicted)


# Store a numpy vector of the predictions for the test set in the variable `predi
```

In [26]:
```
predictions = [i.to(torch.device("cpu")).numpy() for i in preds]
#len(testing_data)
predictions = [i[0] for i in predictions]
```

In [27]:
```
# This code below will generate kaggle_predictions.csv file. Please download it d
import pandas as pd

if isinstance(predictions, np.ndarray):
    predictions = predictions.astype(int)
else:
    predictions = np.array(predictions, dtype=int)
assert predictions.shape == (len(testing_data),), "Predictions were not the corre
df = pd.DataFrame({'Category': predictions})
df.index += 1  # Ensures that the index starts at 1.
df.to_csv('submission.csv', index_label='Id')

# Now download the submission.csv file to submit.
```

Congrats! You made it to the end.