**University of Illinois** Fall 2018

# ECE 544NA: Pattern Recognition
## Lecture 5: September 11

Lecturer: Alexander Schwing Scribe: Tejas Hemant Thosani

# 1 Introduction to PyTorch: A Pattern Recognition Perspective

## 1.1 What is PyTorch?

Pytorch is a python-based package for speedy processing of large datasets, especially for applications in deep learning. It is a very flexible tool that allows users to leverage the power of Graphics Processing Units (GPUs) and use tensors (numpy-array like n-dimensional vectors) that are capable of computing efficiently in parallel and improving processing speeds.

## 1.2 Advantages of using Pytorch

Parallel programming is an integral technique in developing state-of-the-art deep learning algorithms. This is because it is fairly simple to use GPUs to create complex machine learning models by tracking history of variables (here tensors) and the operations performed on them. This history is transformed into an acyclic graph that interconnects tensors and functions. This is provided by the most important package that we shall learn to use - *autograd*.

# 2 Getting Started with PyTorch

## 2.1 Python IDE

It is assumed that the user has Python installed and an IDE setup to develop code in Python. If not, it is recommended to get Anaconda because it has a dedicated package manager which makes it east to install PyTorch. You can find the documentation/instructions here:

`https://www.anaconda.com/download`

You can also use other package managers like pip or chocolatey and use your IDE of choice for Python. Next we will look at installing the packages for PyTorch. (Note: Currently, PyTorch only supports Python 3.x; Python 2.x is not supported)

## 2.2 Installing PyTorch

As mentioned above, Anaconda is the recommended platform as it enables users to get all PyTorch dependencies in a single install [2]. We have two choices for using PyTorch - with or without CUDA (a parallel programming platform and API). For now, we can manage all our tasks in the tutorial without CUDA. Although, in the future you can always install CUDA and leverage the power of GPU for various applications.

In your Anaconda prompt, run the following commands:

```
conda install pytorch-cpu -c pytorch
pip3 install torchvision
```

To install PyTorch via pip (Linux platform), use the following commands:

```
pip3 install http://download.pytorch.org/whl/cpu/torch-0.4.1-cp35-cp35m-win_amd64.
    whl
pip3 install torchvision
```

## 2.3 Tensors

PyTorch provides a type of data structure to represent all different data, model parameters, gradients etc. for our purpose of machine learning. They are similar to Numpy arrays with the added capability of accelerated computing on GPUs. Basically, tensors are multi-dimensional vectors that can compute various operations and implement machine learning using parallel processing. You can get started with using tensors by simply importing *torch* as shown in the examples below.

### 2.3.1 Defining Tensors

```
import torch

x = torch.empty(5, 3) #creates an empty tensor of size 5x3
print(x)
x = torch.rand(5, 3)  #creates a tensor of size 5x3 with random values
print(x)
```

When you run the above two lines, x is first initialized as a 5x3 tensor of zeros and printed out. The second declaration of x initializes the tensor with random values.

An alternative way to initialize a tensor to zero or random values is similar to the numpy package. You can set the size of the tensor with the following technique and also set or override the data-type using the *dtype* parameter.

```
x = torch.zeros(5, 3, dtype=torch.long)
y = torch.randn_like(x,dtype=torch.float)
print("x: ", x, "\ny: ", y)
print("size of x is equal to size of y?: ", x.size()==y.size())
```

The above code prints out the two tensors x and y and then checks if they have the same size. Also, the PyTorch package provides the flexibility of defining tensors with data types without overriding the *dtype* parameter. We can define tensors with float, long and byte precision as follows:

$$w = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \quad x = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 8 & 13 \end{pmatrix} \quad y = \begin{pmatrix} 1 & 16 & 128 \\ 1 & 1 & 1 \\ 128 & 16 & 1 \end{pmatrix} \quad z = \begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 8 & 13 \end{pmatrix}$$

```
w = torch.FloatTensor(2,2)
x = torch.FloatTensor([[1,2,3],[5,8,13]])
y = torch.LongTensor([[1,16,128],[1,1,1],[128,16,1]])
z = x.view(3,-1)  #sets second dimension based on the other dimension indicated
print("w: ", w, "\nx: ", x, "\ny: ", y , "\nz: ", z)
```

Here we define w as an empty 2x2 tensor, whereas we define x and y by the values we want the tensors to hold. Matrix z has the same elements as x but reshaped as a 3x2 matrix. The -1 index automatically helps us set the second dimension based on the first dimension indicated by the user, whereas the *.view*() operator reshapes the vector.

### 2.3.2 Computation with Tensors

The basic operators and functions used on numpy arrays are also applicable for tensors. For example we can perform element-wise operations on two or more tensors, find the mean of the values held by a tensor or find their sum. Let's take a look at this by implementing an algorithm for the following exercise.

**Exercise 1.** Create a rank 2 tensor whose first row is [1,5,3,7] and whose second row is [2,4,1,0]. Find the sum of each row and take the mean of these two values. Then, divide by the standard deviation of the maximum value for each column.

**Solution:**
We will choose the precision of the tensor to be float since we need to find the mean and standard deviation which may be decimal numbers. We can use methods like *.sum*() and *.mean*() for tensors, which are prevalent in the numpy library as well. The *.max*(*dim*) call returns both the max value and the index at which it appears in the tensor. Further dividing by the standard deviation is straightforward and gives us the required result.

```
import torch

a = torch.FloatTensor([[1,5,3,7],[2,4,1,10]]) #Initializing tensor

a_sum = a.sum(dim=1)                #summing each row
a_mean = a_sum.mean()               #mean of summed rows
a_max, a_midx = a.max(dim=0)        #max value for each column
std_dev = a_max.std()               #standard deviation of max
result = a_mean/std_dev             #final element-wise division
print('Result: ',result.item())              #print as value
```

Output:

```
Result:  4.636099815368652
```

# 3 Automatic Differentiation and Optimization

In modeling and optimization, we seek to maximize a favorable quantity or minimize an unfavorable one. Examples of unfavorable objectives or cost functions are the drag in an airfoil for aircrafts, the number of transistors on an IC chip for power efficiency or the amount of weight in space shuttles for efficient flights subject to payload constraints. In IC design, maximizing the power transfer without affecting the bandwidth of operation is an example of a favorable objective.

Since these objectives are a function of the input/state variables, we can optimize it by finding the optimal point of operation. Analytically, this is done by equating the gradient of the objective function to zero and choosing the variables that satisfy the minimum condition.

We need some type of mechanism in PyTorch that helps us calculate the gradient after every operation performed on our input. This is provided by the *autograd* package. Given a tensor variable, we can do the following things:

- *requires_grad*() attribute: If we set it to **True**, the compiler starts to track all operations performed on the tensors
- *backward*() method: At the end of all the operations, we can call this method to compute the gradients automatically.
- detach() method: If we call this method, the compiler detaches the computation history from its memory and stops tracking operations in the future

Let us simplify this by taking a look at an example. Let's say we are given an objective function [4] as follows and we are required to find its gradient:

$$O(x) = \frac{1}{4}\sum_i 3(x_i + 2)^2$$

The weighted summation of the function is just an added step to build a longer acyclic graph of operations and track the variables. Here we sum over four values of $x_i$ and divide by four, which is basically averaging. Let us arbitrarily set the values of $x_{i=1}^4$ as 1.

If we partially differentiate the function $O(x)$ w.r.t each input $x_i$ at $x_i = 1$, we get the following result:

$$\frac{dO(x)}{dx_i}|_{x_i=1} = \frac{3}{2}(x_i + 2)|_{x_i=1} = 4.5$$

The *autograd* package provides this functionality without analytically solving for the equations ourselves. We can break this problem down into different parts to appreciate the flexibility of the *autograd* package.

1. Define a tensor that holds four values of $x_i$ at which the gradient is to be calculated.
2. Compute a second tensor $y_i$ that adds two to all values of $x_i$
3. Compute a third tensor $z_i$ that denotes the function to be averaged
4. Call the *backward*() method to compute gradients
5. Call the *grad* attribute to print out the gradient of the output

The Python code for this algorithm can be found below:

```
x = torch.ones(2,2, requires_grad=True)
y = x+2
print("x: ", x, "\ny: ",y)
print("\nSince x was created by user, its grad_fn value is: ",x.grad_fn)
print("\nSince y was created as a result of computation, its grad_fn value is: ",y
    .grad_fn, "\n")

z = y*y*3
out = z.mean()
print("z: ", z, "\nout:", out.item())
out.backward()        #computes derivative of output
print(x.grad)         #prints dout/dx
```

Output:

```
grad_fn value of x is:  None

grad_fn value of y is:  <AddBackward object at 0x00000024DDF84208>

grad_fn value of z is:  <MulBackward object at 0x00000024DDF841D0>

grad_fn value of out is:  <MeanBackward1 object at 0x00000024DDF84208>

dout/dx:  tensor([[4.5000, 4.5000],
                  [4.5000, 4.5000]])
```

We can see that the gradient computation took only two lines of code using the *autograd* package and the output of our algorithm matches our analytical output of 4.5.

To check how the compiler tracks the variables and builds the acyclic graph of operations, we have printed the $grad\_fn$ attribute of each variable. If the variable is user-defined, the $grad\_fn$ attribute is **None**, else it will be based on the operation performed.

### 3.1 Optimizers in PyTorch

Now that we know how to differentiate functions, we would like to use that for optimizing our objective and building neural networks. Although writing all the optimization algorithms would be counter-productive when building scalable applications with various functions. Pytorch provides torch.optim as a package that implements various optimization algorithms for the user.

To use an optimizer, we can do the following:

- Construct a *torch.optim* object to hold the current state of parameters and update them based on the computed gradients (i.e. using *backward*())
- Define per-parameter options like learning rates while constructing the object
- Call the *optimizer.step*() method to update the parameters after calculating gradients for current step
- Use *optimizer.zero_grad*() method to clear all the gradients after each step (helps refresh memory to ensure proper implementation of the optimizer)

There are times when optimization problems cannot be solved analytically and we require a more elegant solution to finding the minimum. What is also key to training neural networks is the gradient descent method. To construct an optimizer object performing stochastic gradient descent, for example, we use the following syntax:

```
params = torch.FloatTensor([1,1])      #parameter for a 2-d linear model
opt = torch.optim.SGD([params], lr = 0.01, momentum=0.9)
```

**Exercise 2.** Find the minimum of the following objective function using PyTorch:

$$f(x, y) = 3x^2 + 1.5y^2 - 2.3xy + 5.6x + 0.7y + 1.2$$

**Solution:**
For your convenience in visualizing, the objective function has been plotted out as a 3-d mesh plot in Figure 1. (You can find the code to plot it yourself in the Appendix section of this document)

As we can see, the function is a convex surface with multiple local minima. We would like to use the gradient descent method to iteratively step towards the global minimum and converge accordingly.

1. We begin by defining a function $fn$ that takes in the parameter values and return the output of the objective function. This function will be called at each step of the optimization process with updated values of the parameters.
2. Next, we define the start position of the parameter on the 2-d XY plane. This is purely arbitrary but can impact the number of iterations it takes to reach the optimal point.
3. We also define the optimizer object using the syntax discussed above and pass the start point as the model parameter as well as set the learning rate (higher learning rate indicates smaller step size)

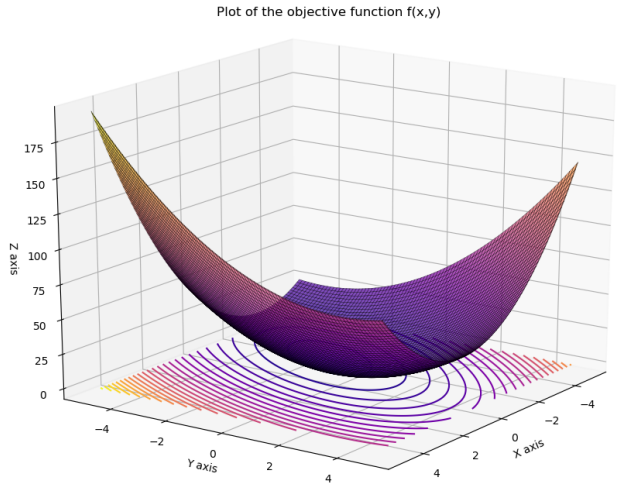


Figure 1: 3-D Mesh Plot of Objective Function

4. We run a loop to update the parameters by calculating the gradients and using that to step in the appropriate direction.

The following code shows the implementation of this algorithm:

```
import torch
def fn(inp):
    x = inp[0]
    y = inp[1]
    return 3*x**2 + 1.5*y**2 - 2.3*x*y + 5.6*x + 0.7*y + 1.2

val = torch.FloatTensor([5,5])
val.requires_grad = True
lr = 0.1
opt = torch.optim.SGD([val], lr=lr)

for i in range(100):
    opt.zero_grad()
    obj = fn(val)
    print("\t%d: %f"%(i, obj.item()))
    obj.backward()
    opt.step()

print("FINAL PREDICTION: ",val)
```

Output:

```
0: 87.699997
1: 43.215839
2: 27.155701
3: 17.311686
.
.
.
97: -3.326042
98: -3.326042
99: -3.326042
Final Prediction:  tensor([-1.4485, -1.3438], requires_grad=True)
```

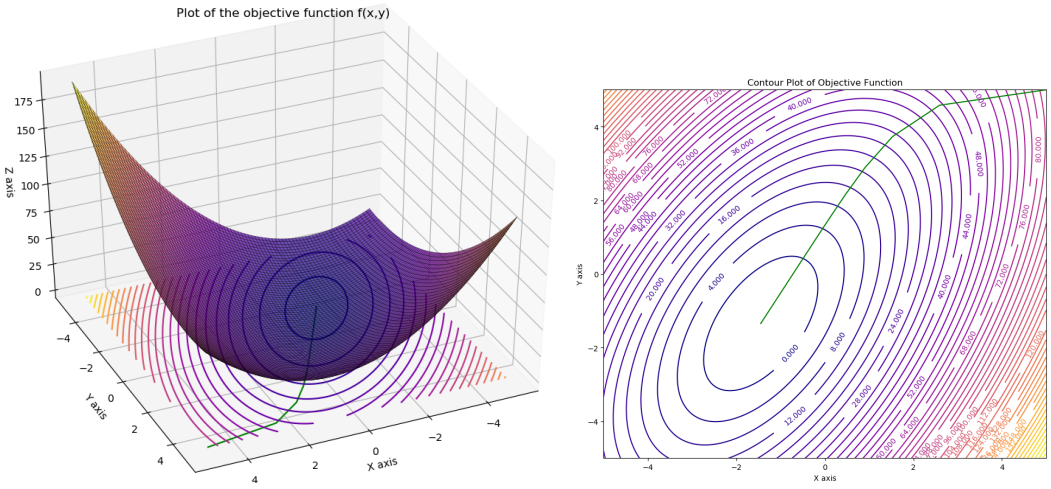


Figure 2: Surface and Contour Plots of Objective Function with Parameter Tracking

**Important**: Here an important thing to note is that we predefined the number of iterations it takes to achieve the minimum; however in practical applications we may not always be able to preempt the iterations and may need to set a tolerance of gradient below which the loop exits.

Figure 2 shows the trajectory of the parameters from the start point to the optimum point on both the contour plot and the 3-D mesh plot.

# 4 Modules and Neural Networks

We previously saw how the *autograd* package provided the ability to build computational graphs and find gradients. However, when looking at large neural networks, *autograd* alone cannot provide the required efficiency for optimizing parameters for multiple layers of a neural network since it is too low-level [3].

PyTorch simplifies this by providing the *nn* package for generating higher-level abstractions over raw computational graphs. Basically, it provides a black box to use as a module for developing artificial neural networks (ANN). Modules are defined by the *nn* package as layers of an ANN. It receives input Tensors and computes the output Tensors while also learning internal parameters of the model. The package also provides pre-written models that we can plug and play to build the network as well as loss functions to train and update the network parameters.

To initialize learnable parameters of the model, we must use $nn.Parameter$ subclass. The notable features of the *nn* package include:

- $forward()$ method: Defines the operations performed on the inputs in the forward direction to find the outputs (feed-forward mechanism)
- $\_\_init\_\_()$ method: Initializes the parameter weights and biases of the network for each layer

Let's look at our previous optimization example to structure it as a $nn.Module$ and achieve the same result.

**Exercise 3.** Implement x and y as part of a model and then find the minimum of the following objective function using PyTorch:

$$f(x, y) = 3x^2 + 1.5y^2 - 2.3xy + 5.6x + 0.7y + 1.2$$

**Solution:**
We can break down the solution into the following steps:

1. We begin by defining our modile as a class with the $\_\_init\_\_()$ constructor and $forward()$ definition. The $\_\_init\_\_()$ constructor defines parameters of the network as *torch.ones* and the $forward()$ function is derived from the objective function.
2. To construct a Module object, we can use our user-defined class name 'TestModule().'
3. We set the learning rate and ensure that the model parameters are initialized by $\_\_init\_\_()$.
4. We proceed to use gradient descent using the *torch.optim* library to optimize our function just like in Exercise 2.

The Python implementation is as follows:

```
import torch
from torch import nn

class TestModule(nn.Module):
    def __init__(self):
        super(TestModule, self).__init__()
        self.params = nn.Parameter(torch.ones(2))   #initializing parameters

    def forward(self):
        x = self.params[0]
        y = self.params[1]
        return (3*x**2 + 1.5*y**2 - 2.3*x*y + 5.6*x + 0.7*y + 1.2)

model = TestModule()
lr = 0.1
opt = torch.optim.SGD(model.parameters(),lr=lr)


for i in range(100):
  opt.zero_grad()
  obj = model()   #calls the forward() method of Module
  print("\t%d: %f"%(i,obj.item()))
  obj.backward()
  opt.step()

print("FINAL PREDICTION: ", list(model.parameters()))
```

**Note**: The only difference we find in our optimization loop is that instead of calling the function to find the output, we call $model()$ which references the $forward()$ method.

## 4.1 Handwriting Modeling with Neural Networks

The real power of PyTorch lies in leveraging the capabilities of modeling, training and implementing neural networks. Neural networks are architectures of machine learning which were inspired from the biological neural connections. Every time a neuron (node) is fired to learn something (train), the synapse (feed-forward mechanism) translates the neuron spike to other neurons in the network which update their core(memory/weights) and multiple layers of neurons undergo the same process.

Neural networks in computing also develop from the same notion of multiple feed-forward layers and large number of nodes/weights per layer. Since the number of nodes can grow manifold, it is necessary to process these nodes in parallel to improve the computational efficiency of the network.

Let's take a look at how we can construct a neural network using the predefined models like $nn.Conv2d$, $nn.ReLU$ and $nn.Linear$. We shall consider a handwriting modeling example as shown in Figure 3 below.

The input is a 32 by 32 image of a written letter. The input is followed by a series of convolutions and subsampling before mapping it into various sub-spaces using linear models. The final output is a 10-dimensional vector that indicates certain features about the letter to be recognized [4].

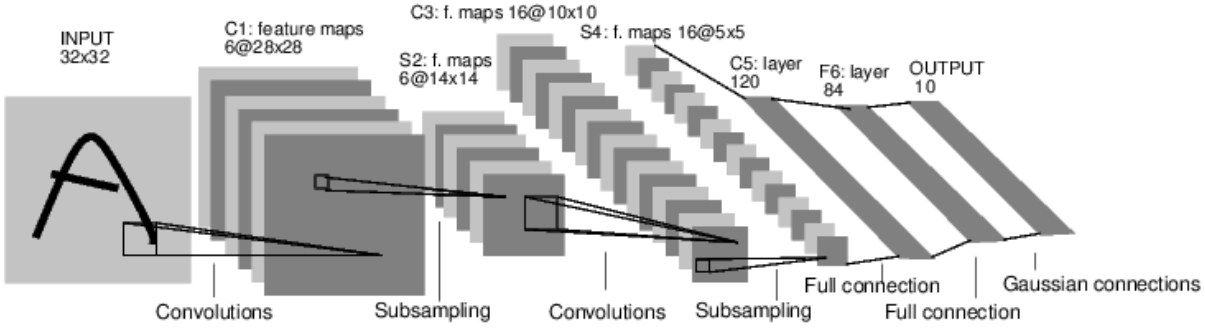


Figure 3: Neural Network Architecture [4] for Handwriting Modeling

The models and functions we want to use have been explained as follows:

- $nn.Conv2d(a, b, c)$: $a$ is number of input channels, $b$ is number of output channels and $c$ sets the size of the convolution window $(c * c)$
- $nn.Linear(a, b)$: Maps an input vector $a$ to an output vector $b$ using a linear model like $b = ma + c$
- $nn.functional.relu(x)$: Represents a widely used activation function which applies the function $max(0, x)$
- $nn.functional.max\_pool2d(data, (a, b))$: Down-samples input data by max pooling over an $a * b$ window

Now, since we know the structure of the ANN and the required functions and models to implement it, let's go ahead and write the code in Python. The basic steps to implement a Neural Network are as follows:

1. Define a Neural Network with tunable parameters
2. Process input using forward() and compute the loss
3. Propagate loss gradients through the network
4. Update parameters to train the model

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net,self).__init__()
        self.conv1 = nn.Conv2d(1,6,5)  # 1 input image channel, 6 output channels,
    5x5 square convolution
```

```
        self.conv2 = nn.Conv2d(6,16,5)

        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # Linear Model: y = Wx + b
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self,x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))   #corresponds to S2
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)      #corresponds to S4
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))               #corresponds to C5
        x = F.relu(self.fc2(x))               #corresponds to F6
        x = self.fc3(x)                   #corresponds to OUTPUT
        return x
    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()

#Output is computed using the Net object
input = torch.randn(1,1,32,32)
out = net(input)

#Loss is computed and propagated backwards to update weights using SGD
target = torch.randn(1,10)
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
criterion = nn.MSELoss()    #Here we use the Mean Squared Error Criterion

for i in range(500):
    optimizer.zero_grad()
    output = net(input)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
```

### Summary of Code

In our *__init__*() constructor, we define the basic transformations to compute the layers of our ANN. These include the convolutions and linear mapping that will be used later in our *forward* method. In *forward*, we call our helper functions for convolution and linear modeling along with subsampling using max pooling. After this we return the output. The 'num_flat_features' function basically stacks all our features together to reshape our samples.

Once the output is computed, we define the SGD optimzer and Mean Squared Error Loss. We run our loop to propagated the loss backwards and update the weights to train our ANN.

If we print out the loss before and after the 500 iterations of training, we have a non-zero loss between output and target initially which later diminishes to a negligible value:

```
BEFORE TRAINING:
Loss = 3.7799293994903564

AFTER TRAINING:
Loss = 1.0191847704872116e-14
```

This exercise shows how simple it is to implement neural networks using the abstractions provided by the nn package. Feel free to explore other machine learning problems from the PyTorch website here: `https://pytorch.org/tutorials/beginner/pytorch_with_examples.html`

# 5 Datasets and Dataloader

## 5.1 Data Managing and Processing Library in PyTorch

There are loads of other functionalities provided in PyTorch. One of them is developing and managing datasets [5]. The torch.utils.data library provides two main features:

- *torch.utils.data.Dataset*: Represents the actual dataset [1] and provides methods like input length $\_\_len\_\_()$ and getter function $\_\_getitem\_\_()$
- *torch.utils.data.DataLoader*: Samples the data set to access mini batches of your data - useful to load data in parallel using multiprocessing workers [5] without iterating through the data manually (also provides the option to shuffle the data)

The following exercise helps explain how we can generate our own dataset and also batch the data for further processing.

## 5.2 Creating Datasets and Batching the Data

**Exercise 4.** Create a dataset consisting of the following:

Inputs:4-d vectors sampled from the standard Gaussian distribution

Labels: scalar-valued tensors containing the index of the largest value of the input

Implement this so that the number of data-points is an arguments to the constructor for the dataset. Additionally, try using a dataloader to loop through the dataset.

**Solution:**

```
import torch
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):
    def __init__(self, num_data_points):
        self.inputs = torch.FloatTensor(num_data_points, 4).normal_()
        self.outputs = self.inputs.argmax(dim=1)

    def __len__(self):
        return self.inputs.size(0)

    def __getitem__(self, idx):
        return self.inputs[idx, :], self.outputs[idx]


sample_dataset = RandomDataset(10)
dataloader = DataLoader(sample_dataset, batch_size=2)

for inputs, outputs in dataloader:
    print(inputs, outputs)
```

Output:

```
tensor([[-0.8358, -0.3540,  0.1310, -0.2245],
        [-0.1878,  0.0898, -1.2088,  0.3630]]) tensor([2, 3])
tensor([[-1.0237, -0.9497, -0.2090,  0.2718],
        [ 1.0081, -1.6357, -1.1379, -0.6891]]) tensor([3, 0])
tensor([[ 0.4502,  1.0163, -0.4753, -1.7153],
        [-0.0926,  1.9447,  0.1237,  0.8800]]) tensor([1, 1])
tensor([[ 1.6357, -1.0838, -0.0973, -0.3301],
        [-1.7217,  1.7801,  1.6922, -1.4429]]) tensor([0, 1])
tensor([[ 0.7605, -0.7881,  0.9647,  0.8931],
        [-0.7072, -1.3178, -0.1263,  1.2763]]) tensor([2, 3])
```

**Summary of Code**

First, we define our own class called RandomDataset whose constructor requires the number of samples/data-points as a parameter. In the initilization method, we define the input to be a tensor with data type float and size equal to the number of data points by 4. This means that we have four features per sample. The outputs are defined as the index of the largest value in input using argmax.

Next we define a getter function to get the length (number of samples in the input as well as a getter function to return the inputs and corresponding labels.

We construct a Dataset object using the syntax $sample\_dataset = RandomDataset(10)$, where 10 denotes the number of samples. We create a DataLoader object to batch our data into 2 samples per batch. When we print the inputs and outputs, we see that the result is five batches of 2 samples each and that the corresponding labels are the index of the max in each sample. This is returned by our getitem function.

# 6 Linear Regression for Real Dataset

So far we have looked at automatic differentiation, optimizers and data batching for neural networks. Let us integrate all these different blocks to implement Linear Regression in PyTorch for a given dataset. Save the data file called '*linear_regression.data*' provided in this folder and proceed with the following exercise.

**Exercise 5.** Create a dataset consisting of the following:

Inputs: 10-d real-valued features

Output: 1-d real-valued tensor

Create a linear model and learn a model that minimizes squared loss between predictions and outputs. Implement this using a Module for your model and a Dataset wrapping the provided dataset.

**Solution:**
We begin by defining our two classes for the dataset and neural network module.

1. Dataset class: The data will be imported as input and labels; hence defining the dataset constructor is straightforward. The two getter functions for length and item remain the same as in our previous example.
2. Module class: Since we are dealing with a linear regression problem, our class constructor would only include a single layer which performs a linear mapping of our data. The linear transformation matrix will have size $(m * 1)$, where $m$ is the number of features. Hence it will take in $m$ values and return a scalar output for each sample.

The linear relationship [1] is illustrated below:

$$Y_n = (\Phi_{m,n})^T \ . \ W_m$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} \phi_{1,1} & \phi_{1,2} & \cdots & \phi_{1,m} \\ \phi_{2,1} & \phi_{2,2} & \cdots & \phi_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{n,1} & \phi_{n,2} & \cdots & \phi_{n,m} \end{pmatrix} . \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix}$$

Here $W_m$ represent the learnable parameters of the network, $\Phi_{m,n}$ is the input matrix with m features and n samples and $Y_n$ is the output vector.

After importing the data, we notice that there are 100 samples of 10 dimensions each as the input. It makes sense to batch the data into 10 batches of 10 samples each - hence we get a batched data matrix of 10 x 10. For this we will use the Dataloader class to cycle through the required batches.

After initializing our optimizer and model object, we proceed with gradient descent on each of our batches for 1000 iterations and then print our final trained parameters and final loss.

```
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader

class SampleData(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels
        self.labels.requires_grad = False

    def __len__(self):
        return self.data.size(0)

    def __getitem__(self,idx):
        return self.data[idx,:], self.labels[idx]

class OurModel(nn.Module):
    def __init__(self,feature_size):
        super(OurModel, self).__init__()
        self.model = nn.Linear(feature_size,1)

    def forward(self, inp):
        return self.model(inp)

data, labels = torch.load('linear_regression.data')
dataset = SampleData(data, labels)

model = OurModel(data.size(1))
lr = 1e-3
opt = torch.optim.SGD(model.parameters(),lr=lr)

dataloader = DataLoader(dataset, batch_size=10)


for i in range(1000):
    for inputs, labels in dataloader:
        opt.zero_grad()
        predictions = model(inputs).squeeze()
        loss = nn.MSELoss()(predictions, labels)
        print("%d: %f"%(i, loss.item()))
        loss.backward()
        opt.step()
```

```
print("\nFINAL MODEL PARAMETERS: ", list(model.parameters()))
print("\nFINAL LOSS: ", loss.item())
```

Output:

```
.
.
999: 0.000009
999: 0.000023
999: 0.000016

FINAL MODEL PARAMETERS:  [Parameter containing:
tensor([[ 0.1624, -0.1400, -0.0618,  0.1481, -0.2981,  0.1894, -0.0654,  0.1603,
          0.0434, -0.0392]], requires_grad=True), Parameter containing: tensor
             ([0.1103], requires_grad=True)]

FINAL LOSS:  1.6078476619441062e-05
```

# 7 Conclusion

In this tutorial, we learnt about how we can use tensors and the associated libraries to build different sub-blocks for machine learning. We started with the *autograd* package and learnt how to optimize a function. Next we explored how to build neural networks using pre-defined models and modules catered towards machine learning. We learnt how to manage datasets efficiently for initialization, processing and batching data. To summarize all the skills we learnt, we implemented linear regression on a dataset provided and trained our model by integrating all the functionalities from before. This tutorial emphasizes on exploring more examples online and learning many more libraries and packages available for PyTorch. Some extra examples like handwriting modeling with neural networks were also discussed.

# References


[1] Schwing, A. (2018). *ECE544NA: Pattern Recognition, L02 and L05 Slides*. Retrieved from https://courses.engr.illinois.edu/ece544na/fa2018/_site/index.html

[2] Paszke, A. (2018, September 18). *Get Started Locally*. https://pytorch.org/get-started/locally/

[3] Johnson, J. (2018, September 18). *Learning PyTorch With Examples*. Retrieved from https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

[4] Chintala, S. (2018, September 18). *Deep Learning With PyTorch: A 60 Minute Blitz*. Retrieved from https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

[5] Chilamkurthy, S. (2018, September 18). *Data Loading and Processing Tutorial*. Retrieved from https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

## Appendix

Code for Plotting the contour and 3-d mesh plots for optimization problem in Exercise 2:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

s = 1000
X = np.linspace(-5e0,5e0,s)
Y = X
Z = np.zeros((s,s))
for i,x in enumerate(X):
    for j,y in enumerate(Y):
        Z[i,j] = 3*x**2 + 1.5*y**2 - 2.3*x*y + 5.6*x + 0.7*y + 1

A,B = np.meshgrid(X,Y)
C = Z.transpose()
fig1 = plt.figure()
CM = plt.contour(A,B,C,50,cmap=cm.plasma)
plt.clabel(CM, CM.levels, inline=True, fontsize=10)
plt.title('Contour Plot of Objective Function')
plt.xlabel('X axis')
plt.ylabel('Y axis')

fig2 = plt.figure()
ax = fig2.gca(projection='3d')
surf = ax.plot_surface(A,B,C, rstride=12, cstride=12, alpha=0.7, cmap=cm.plasma,
    edgecolor='k',linewidth=0.3)

ax.contour(A,B,C, 25, zdir='z', cmap=cm.plasma, linestyles="solid", offset=-1)

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Plot of the objective function f(x,y)')
ax.view_init(15,30)
plt.show()
```