FIRST STEPS TOWARDS DEEP LEARNING

WITH PYTORCH

VIPUL VAIBHAW

Chapter 1

What are artificial neural networks? What are its components?

Inspiration

As we human beings evolved with time, curiosity to understand certain subjects increased exponentially. Universe, Singularity, Meaning of life, God, Infinity and Brain were the ones which made to the top of the list. With time our brain became more and more efficient, we starting thinking about subjects deeply and we started asking the right questions. I will list a few of the beautiful questions down here, and while you read them take a pause and applaud the centuries of human progress it shows -

- Are we alone in the universe?
- What is consciousness? Are all living beings conscious?
- What makes us human? - It can’t just be DNA because human genome is 99% identical to a chimpanzee
- What’s so weird about prime numbers?
- Do we have free will?
- P versus NP
- How does our brain work?

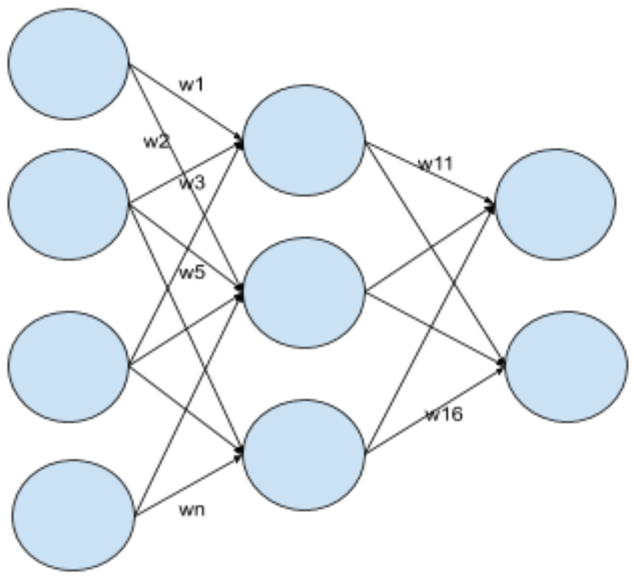
Did you notice? How beautifully and precise are these questions? Generations of people have spent time in thinking through these problems. We came up with questions like these, we thought! We think!

You see what I did there? Yes, Human Brain is the core of all the curiosity here. So what if we focus on developing a human brain? We will be able to put that brain to focus on trivial issues like driving a car,making sure that fraud doesn’t happen over credit card transactions etc. We can also put it to remote places where accessibility is an issue and solve complicated problems. We can put that brain to evaluate cardiovascular diseases via a retina scan. Now to do this, to carry out the tasks mentioned above we need intelligence. We need Artificial Intelligence!

Note - The Blue Brain project is a dedicated team of folks who are working to build a digital reconsruction and simulations of a rodent and eventually human brain.

How do we develop AI? Where do we go for inspiration? When we had to build a plane, we studied birds. For building helicopters, dragonflies came for rescue. So to build a system which can be put to solve trivial as well as complicated problems, we need to look for inspiration from the most intelligent species on earth. We need to look inside a living being which has the largest cerebral cortex relative to their size. We would need to look inside Human Brain!

What are neural networks?



Before this representation of neural network starts unsettling us let us take time to understand how this architecture was thought of and how to interpret the image.

As mentioned in the section above, the neural networks are inspired by the way our brain works. The picture above represents a graphical flow in which the circles, represents neurons and the edges represent axon. The numbers { w1, w2 … wn } are set of weights of the neural networks. The weights are the key part of neural networks because these are the parameters which gets tuned when neural network goes under training process.

Another thing to remember is that neural networks are not an exact representation of the brain. Brain is a very complicated organ and it is obvious that we don't learn by the method of backpropagation(to be discussed later). Having said that let us further examine the picture above, if you observe carefully you will see that not all neurons in layer 1 are connected to all the neurons in layer 2. However, all the neurons in layer 2 is connected to all the neurons in layer 3(the final layer). This type of layer is known as a **Fully Connected Layer**. If a neural network has all layers as fully connected layers then that neural network is called **Fully Connected Network**(FCN).

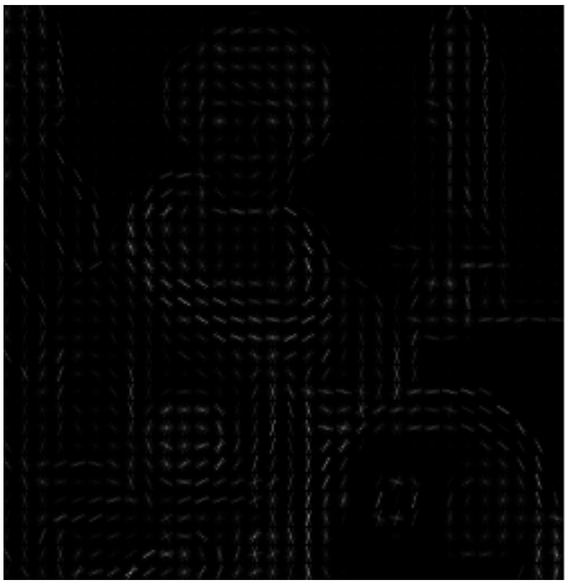
Before we digress to our next topic, let us spend some more time in understanding neural networks and their importance. Firstly, Neural networks are really powerful because of their ability to learn the relationships in a set of data on their own. Neural networks do not need to be told which features are important or not. Neural nets are capable of extracting the needed features and carry on the task efficiently.

Before neural nets became popular, people used to hardcode features by hand in the traditional machine learning algorithms. For example, in computer vision HOG(Histogram of Oriented Gradients) were used. Let us say that we want to detect a face then a hardcoded feature can be that there will always be a slope from cheeks to eyes.

Following is an opencv code which can generate a HOG from an image for us.

```
from skimage import exposure,feature, data
import cv2

image = data.astronaut()
_ , hogImage = feature.hog(image, orientations=8, 
pixels_per_cell=(16, 16),
    cells_per_block=(1, 1), transform_sqrt=True,
    block_norm="L1", visualize=True, multichannel=True)
hogImage = exposure.rescale_intensity(hogImage, out_range=(0, 
255))
hogImage = hogImage.astype("uint8")

while 1:
    cv2.imshow("HOG Image", hogImage)
    cv2.waitKey(1)
```

We won’t go through the code for now. Following will be the output.

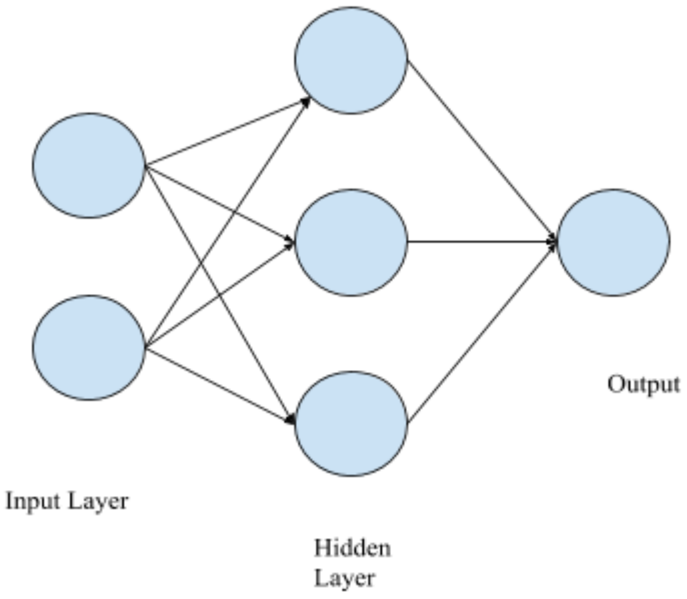


The challenge is that it is hard to write every possible feature in all the varying conditions by hand. That is why we give neural nets diverse data points so that it can learn these features by itself. Neural networks adapt to the new inputs they see. If designed well then it has an amazing capability to generalize across various scenarios and often across domains. A neural network which is trained to differentiate between cats and dogs can also be used to differentiate between elephants and giraffes via the process of *transfer learning*.

Fully Connected Neural Networks

History is witness that the field of Artificial Intelligence has gone through multiple phases of boom-and-bust cycles. It is like stock markets, after every bull run there is a correction and once in a while there is recession and depression.

I would say that the field of deep learning started with emergence of Multi-Layer Perceptrons(MLP). Simply put, it was a feed forward neural network. These MLPs had an input layer, a hidden layer and an output layer. MLPs could work well with data that is not linearly separable. The key thing about MLP is that all of its layers were fully connected. It means that each neuron of one layer was connected to all the neurons in the next layer.



In the year 1975, backpropagation became the practical way to train MLP. However deep learning was still not a mainstream way to do machine learning. That was because MLPs were usually trained on XOR datasets. More than the lack of computational power there was also a lack of real life datasets on which the practicality of MLP could be verified. Around 1995, Yann LeCun released MNIST dataset. This dataset had collection of handwritten digits. This for some time became benchmark for forthcoming neural networks.

In today's era we have a multitude of datasets available. Imagenets for benchmarking object classification neural networks. MOT17, PASCAL VOC etc for object tracking and detection. However, MNIST dataset is still important today. It helps us in quickly prototyping and testing our models on a light-weight dataset. If a model is performing well on MNIST dataset then it is safe to pursue the design of the model one has thought of and then train it with heavy datasets. There are some datasets which can be useful along with MNIST i.e Fashion MNIST, Omniglot dataset, etc.

We have got a neural network which is fully connected. It has got one or more than one hidden layers between input and output layers. We have datasets as well. Now, let us dive deeper into training process of fully connected neural networks.

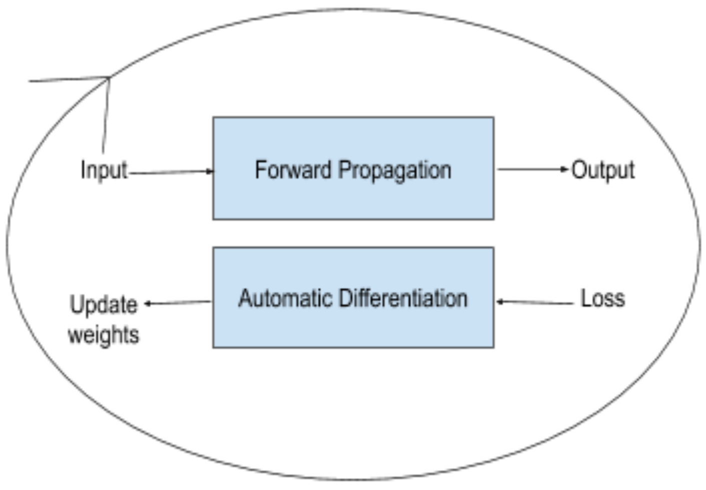
BackPropagation

In the sections above we have seen that neural networks have neurons in the layers and every neuron may or may not be connected to all the neurons in the next layer. The concept to which one should pay attention here is that every connection from one neuron to another carries weight.

Forward propagation is the process in which the input passes through the neural network to give an output(say probability). Backward propagation is the process in which we calculate how far we are from the ground truth and based on that we adjust the weights of the connections, so that next time we are closer to the ground truth. Intuitively, it tells every layer/connection/ neuron that given the current input how much they were responsible for the wrong output and how should they correct themselves. In this way when a new input comes, we get closer to the ground truth and the loss is minimized.

Broadly speaking, there are three steps in training -

1. Model Initialization (we will talk about it later)
2. Forward Propagation - This predict an output.
3. Back Propagation - On the basis of a defined loss function we calculate how far is the model from the ground truth and then we update the weights in the network. These updates happen via differentiation which the Optimizers do.



Training Loop

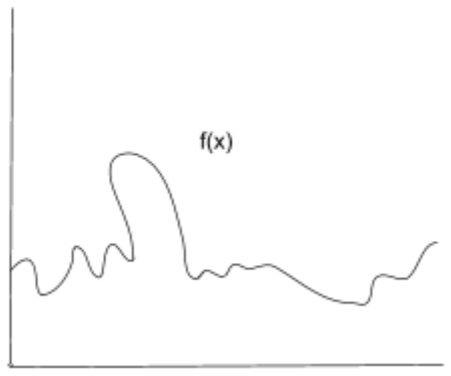
Backpropagation is mathematically hard to understand. Luckily we have got deep learning frameworks like pytorch, tensorflow, darknet, Mxnet etc which takes care of automatic differentiation for us.

Backpropagation is just another name for automatic differentiation. Our agenda here is to build basic intuition that can get you started in the field of deep learning. If you are interested in knowing more about backpropagation and you want to dive deep into the mathematics(beyond the scope of this book) of it, I would suggest you to do a literature survey and read classic papers by Hinton and Yan LeCun.

Note - Geoffery Hinton, father of current AI boom, is deeply suspicious about Backpropagation. He says “My view is throw it all away and start again, I don't think it's how the brain works. We clearly don't need all the labeled data". We humans try to find patterns in everything. We do it with the even scarcity and sparsity of data. There has to be a better way than backpropagation. We need to think more about unsupervised learning.

Universal Approximators

Deep fully connected neural networks are often called as universal approximators. This is because no matter how complicated a function may look like, neural networks can approximate it.



Universal approximations are quite a common phenomena in mathematics as well as in computer science.

A couple of things to keep in mind about universal approximation theorem is that neural networks can approximate a function and not exactly compute a function. Secondly, the function which is being approximated should be continuous. A function is said to be continuous for which sufficiently small changes in the input have arbitrarily small changes in the output. In simpler words, Neural networks can approximate sin(x) but not 1/|x| where x belongs to the set of real numbers. If you can formulate a problem well enough, neural networks can solve them for you. The part of problem formulation is difficult.

Universal Approximation theorem makes neural network truly remarkable. It allows us to work with any arbitrary functions. Imagine that all the data which you have collected is now plotted on a cartesian plane, there will be a wiggly line joining all those data points. Neural networks have this amazing capability to extract right features and predict based on those features or patterns.

If you are a mathematician or are interested in the proof of why Universal Approximation Theorem(beyond the scope of this book) holds for neural networks then you should read some original papers. One needs to be aware of fourier transformations, Hahn-Banach theorem etc to follow the proof.

The more layers the better the network?

This is an active area of research and AI practitioners often find it hard to take one side. While designing a neural network it is often advised that it is better to add more layers rather than adding more neurons in a layer.

Basically, as the hypothesis space of a learning algorithm grows, the algorithm can learn richer and better relationships. However, chances of becoming prone to overfitting and its generalization error increases. Hence, it is theoretically advisable to work with the minimal model that has enough capacity to learn the real structure of the data. But this is a very hand-wavy advice, because usually the core internal structure of the data is unknown, and often we don't understand the models which we train.

Shallow networks are very good at memorization, but not so good at *generalization*. The advantage of multiple layers is that they can learn features at various levels of abstraction. If one builds a very wide, very deep network, there are high chances of each layer just memorizing the output, and the neural network fails to generalize.

I know this is confusing! So one take on this can be to train a deep model and then try to minimize it. By following this approach one can quickly prototype and eventually follow *neural network compression and quantization*.

There is a dedicated research area for this topic called Neural Architecture Search (NAS), that focuses on creating algorithms or methods for finding the optimal architecture that fits certain data, by architecture meaning the number of layers, nodes, etc of a network. This is a subfield of AutoMachine Learning that it is growing in importance through this last years. More info and literature about it can be found here

---

Types of Learning

Before moving forward, It is important to understand the various ways in which a neural network learns.

1. Supervised Learning - In this way of training a model, we provide huge amount of labelled dataset to the neural network. The dataset are in pairs of input and the desired output.

2. Unsupervised Learning - It is inspired by one of the oldest learning algorithms, Hebbian learning. In this way of learning a model is given a dataset which is neither labelled nor classified.
3. Imitation Learning - This is popular in the field of reinforcement learning. In this mode of learning a policy is formed based on demonstrations.
4. Active Learning - In this mode of learning, the algorithm queries the source of information to get the desired output at new data points.

We will be mostly dealing with *supervised learning* in this book. However, I encourage the reader to explore active learning on its own. It has got a lot of applications in the field of *Natural Language Processing (NLP)*. Computer Vision world still relies heavily on supervised learning. One of the reasons being huge availability of labelled datasets. However, even in computer vision, one shot learning and zero shot learning have become active areas of research. Reader is advised to explore *ESZSL(Embarrassingly Simple Approach to Zero Shot Learning)* algorithm.

Weight Initialization

The sections above have acquainted us with the overall idea of training a neural network. One thing to understand here is how do we process the first input during training process? Recall that connections between various neurons have weights. How should we initialize those weights efficiently so that the learning can happen smoothly? Let us say that we initialized all the weights as zero. If we do that then we will lose the symmetry inside the neural networks. During backpropagation every layer will have similar weight updates and every layer will learn the same thing. This makes your model equivalent to a linear model.

Another way to solve the problem of weight initialization can be to randomly initialize the set of weights. This method often leads to two potential issues -

1. Vanishing Gradient Problem - Simply put, during backpropagation the weight updates will be so less that neural network will stop learning totally no matter how many epochs you run it for or how much more data you feed it.
2. Exploding gradients - This is opposite of vanishing gradient problem. It will stop model from reaching of global minima and during the optimization(of loss) the model will keep oscillating and will never learn.

There are a few methods in which we can overcome these problems like using relu activation function, dropouts or gradient clipping etc.

Note - Pytorch, by default, uses *Xavier weight Initialization*. *He Initialization*, *Fixup Initialization* are some other ways in which weights in the neural network can be initialized however, those are beyond the scope of the book.

Activation functions

Neural networks like MLP were inspired by brains. Apart from having neurons, weights and connections they also have something similar to action potential known as Activation Functions.

The idea is that we want an on/off mechanism for neurons. We want neurons to fire only when an input hits a certain threshold. Activation functions helps in introducing non-linearity into the output of a neuron. Non-linearity is an important property to have in neural networks because without non-linearity a neural network will be like a regression model.

There are a lot of activation functions available like sigmoid, tanh and relu (maxout and leaky relu are some variants of relu). Relu helps the neural networks architect to deal with vanishing gradient/gradient explosion problem.

It is a very simple function - `f(x) = max(0,x)` Basically, Relu outputs the maximum of zero or x.

Note - Whenever in doubt, use RELU(Rectified linear unit).

Understanding overfitting, underfitting and generalization

Keeping track of unending jargon is one of the toughest hurdles for a newbie entering in the field of Artificial Intelligence. It is important to understand Overfitting, Underfitting, and bias-variance trade-off because these are core terminologies of Machine Learning world.

One can say that the sole purpose of neural networks is to generalize well. Normally, we are used to algorithms like mergesort which are trained to do one particular task really well. In the machine learning world we want a model to detect object really well while also being able to do human pose estimation(Mask RCNN).

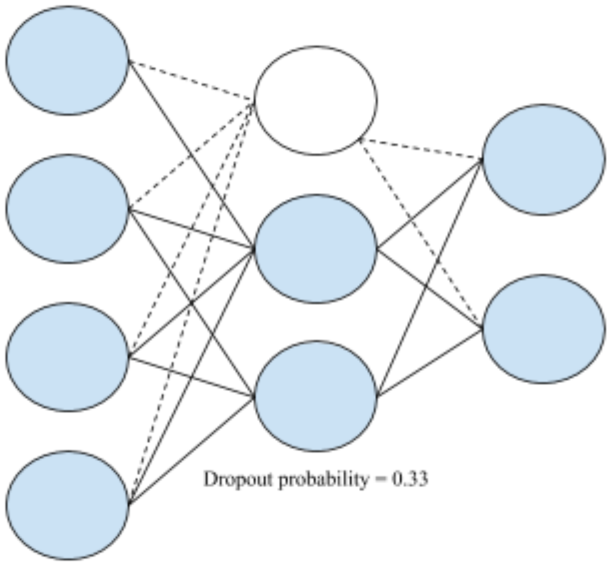
Overfitting happens when the model is performing really well on the training set, i.e the loss during training reduces but the model fails to perform well in test dataset/real world. While Underfitting happens when the model fails to learn from the training data and is unreliable. A generalized model is neither overfit nor underfit.

Note - An integral part of machine learning is bias-variance tradeoff. We can measure the generality of a model using this concept. Generalization is bound by the two undesirable outcomes — high bias and high variance.

Dropouts

Using dropouts can be an effective way to avoid overfitting of the model.

The idea here is to switch off some neurons in layers of neural networks randomly during each iteration, so that not all the neurons see all the training data. By using dropouts we train an ensemble of neural networks rather than training a single architecture.



Looking at the figure above, we can see that the dropout probability for the second layer was 0.33 . It means that each neuron has 0.33% chances of getting “dropped out” or switched off. Let us say that the first neuron of the second layer was switched off, now when a data will pass through this network and weights will be updated via back-propagation then the weights of the connections connecting that neuron won’t be updated. It is like that the neuron never saw that data point, in next iteration another set of neurons will be switched off.

Dropouts, however, can make us lose information. To counter this fact, Batch Normalization came into the picture. We can use less dropouts when we use Batch Normalization because it helps us to retain information while solving the problems of overfitting.

Dropouts are mostly used during training and deep learning frameworks make managing dropouts extremely easy.

Note - For serious readers, I’d highly recommend reading the paper on dropouts by G.Hinton et.al.

Chapter 2

Introduction to Pytorch

The barrier to entry is getting lower each day for the field of Deep Learning. There were times when people used to write stuff from scratch every time when they had to implement any idea. Today, we have got great frameworks like pytorch and keras(higher level api based on tensorflow) where we can easily build models and verify our ideas.

Note - If you don’t have a programming background then you can skip this chapter. This chapter assumes that the reader is versed with Python.

Why pyTorch?

Pytorch is a python based scientific computing framework which also helps us in designing deep neural networks. Being python friendly it makes it really easy beginners to start coding right away. Python has kind-of become lingua franca for the machine learning world.

```
“I've been using pytorch a few months now and I've never felt
better. I have more energy. my skin is clearer. my eyesight
has improved.”

— Andrej Karpathy
```

Some of the key things which I personally like is - - Simple and Intuitive APIs - Computational Graph

Dynamic computational graph of pytorch makes it really intuitive to code. Since the graph is not static as compared to tensorflow, it makes it easier for deep neural network architects to change the behaviour on fly.

```
“An additional benefit of Pytorch is that it allowed us to
give our students a much more in-depth understanding of what
was going on in each algorithm that we covered. With a static
computation graph library like TensorFlow, once you have
declaratively expressed your computation, you send it off to
the GPU where it gets handled like a black box. But with a
dynamic approach, you can fully dive into every level of the
computation, and see exactly what is going on.”

- Jeremy Howard
```

There are a lot of benchmarks showing that pytorch is faster than keras and sometimes comparable to tensorflow.

A confession here, I love cpp more than python. PyTorch is deeply integrated with the C++ code, and it shares some C++ backend with torch. One can further speed up things by using C++ because they will be closer to machine. Although one can write C++ code in tensorflow also.

Pytorch’s coding style is imperative rather than declarative(which tensorflow has). It makes things intuitive because a lot of people are used to the imperative coding style. If you are someone who wants to prototype ideas quickly then Pytorch will definitely increase developer productivity.

Debugging is often saviour of a developer. Pytorch brings ease in debugging as compare to other frameworks out there.

The important thing here to note is PyTorch uses different backends for each computation devices rather than using a single back-end. It uses tensor backend TH for CPU and THC for GPU. While neural network backends such as THNN and THCUNN for CPU and GPU respectively. Using separate backends makes it very easy to deploy PyTorch on constrained systems.

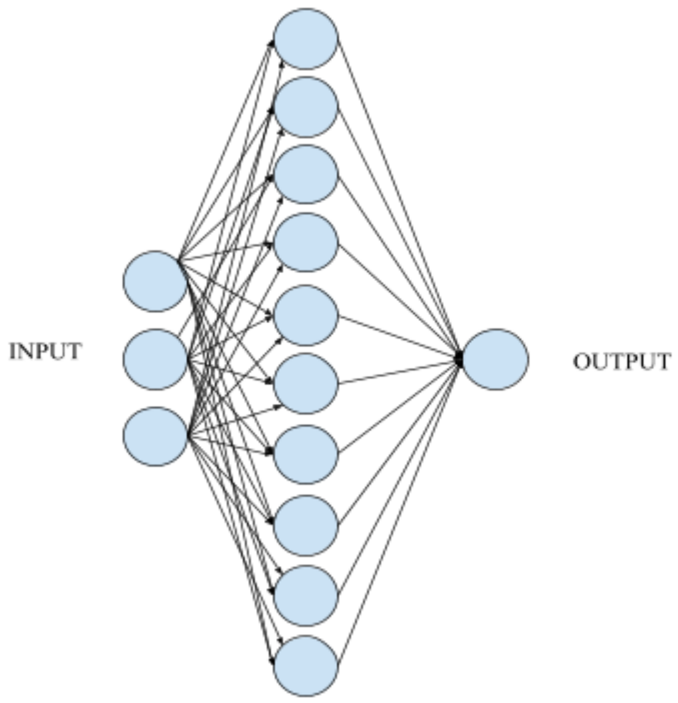
Writing first neural network in pytorch

```
“Talk is cheap. Show me the code”

- Linus Torvalds
```

For the sake of demonstration we will be designing a neural network in Pytorch with 1 10-neuron hidden layer, an input and an output layer.

The reader can take the code below and try to add more layers and experiment with changing the width(number of neurons) in the hidden layers to see how it affects the output or the training process.

Some stuff to know before proceeding to the following code - 1. Epoch - It is a measure of the number of times all of the training data are used once to update the weights. An epoch can have multiple iteration steps. Since whole data can be very big to load in memory, so we often load the data in batches and pass it through the neural network, such passes of mini-batches are known as iteration steps. 2. Iris dataset - Iris flower data set is a multivariate dataset which has 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor) and their petal length, sepal length, petal width and sepal width.



```
# Let us start with importing the libraries
# We will go in depth later

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
# Taking a few data points from iris dataset .
# Here X represents the list of [petal length, petal width and
sepal length].
# Y is the sepal width.

X = torch.tensor(([5.1, 3.5, 1.4], [6.7, 3.1, 4.4], [6.5, 3.2,
5.1]), dtype=torch.float) # 3 X 3 tensor
y = torch.tensor(([0.2], [1.4], [2]), dtype=torch.float) # 3 X
1 tensor
xTestInput = torch.tensor(([5.9, 3, 5.1]), dtype=torch.float)

# Given X we will try to predict sepal width using neural
networks

# We will now define a neural network in pytorch
# It will have two fully connected layers.
# In nn.Linear(3,10) , 3 specifies the input size and 10
specifies the output size.
# we will be giving 3 data points to the neural network [petal
length, petal width and sepal length] and it will have 10
neurons hidden layer which will pass the data to another layer
fc2.
# fc2 will give us 1 output i.e sepal width
```

```
class Neural_Network(nn.Module):
    def __init__(self):
        super(Neural_Network, self).__init__()
        self.fc1 = nn.Linear(3, 10)
        self.fc2 = nn.Linear(10, 1)

    def forward(self, x):
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        return x

# The following code is for training neural networks

model = Neural_Network()
model.train()

# we are using stochastic gradient descent optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01,
momentum=0.5)

# we train the neural networks for 1000 epochs.
# epochs basically means the number of time the whole set of
data will pass through the neural network.

for i in range(1000):
    optimizer.zero_grad()
    output = model(X)

    # We calculate the loss, i.e difference between ground
truth and predicted value.
    loss = F.binary_cross_entropy_with_logits(output, Y)
    print(loss)

    # Following code is to do backpropagation and for updating
weights.
    loss.backward()
    optimizer.step()

# Now our model is trained, so we pass a test input to see how
the model is performing.
print(model(xTestInput))
```

Essentials of pytorch

We have seen above an example which demonstrates how easy and intuitive it is to write. In this section we will try to dive deeper into Pytorch.

```
import torch
import torch.nn as nn
```

torch.nn contains all the necessary tools we would need while coding up a neural network like Linear Layers, RNNs etc. Reader should check this link out in order to understand more about torch.nn - here

This is how you initialize a tensor in pytorch. In pytorch everything is a tensor.

```
x = torch.tensor(([5.1,4.3,2.5],[5.1,4.3,2.5],[5.1,4.3,2.5]),
dtype=torch.float)
```

A Variable wraps a Tensor and supports nearly all the API’s defined by a Tensor. Variable also provides a backward method to perform backpropagation.

```
from torch.autograd import Variable
a = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
b = torch.sum(a**2)

# compute gradients of b with respect to a
b.backward()
print(a.grad())
```

One more thing which often comes handy is the knowledge of numpy to pytorch conversion and vice versa.

```
import numpy as np
a = np.array([1,2,3]) # numpy array
b = torch.from_numpy(a) # pytorch Tensor
```

We will keep using pyTorch in the book, especially in chapter 3 and 4. Readers are advised to refer Pytorch’s documentation, it is one of the best resources out there. Also fast ai’s module which is based on pytorch is extremely beginner friendly. It is worth a try.

Chapter 3

How to make a computer see?

Introduction

Computer Vision has been one of the most captivating areas of interest for researchers. One can say that a lot of progress which has been made in the field of deep learning was motivated by the challenge to solve some computer vision problems like object detection, classification, etc.

Honestly, we still don’t understand how we see! Our eyes in sync with our brain does amazingly complicated tasks(like completely ignoring our own nose while seeing) really well. Imagine yourself wearing a helmet while driving a motorbike, during the rainy season when droplets start to slide down on the visor our vision system is completely capable of ignoring that and gives us proper contextual information which helps us in driving.

The “Intelligent” vision systems which we design works really well with a lot of constraints. Anyways, the field is progressing rapidly and we are not only stuck in doing a single job really well like detection or classification but we are trying to get contextual information with which the machine can reason!

Convolutional Neural Networks

As promised, this book will help the reader to take their first steps in deep learning. So in this section we will be overlooking a lot of mathematics while focusing on intuition.

In deep learning world, Convolutional Neural Networks (or ConvNets or CNNs) have become a standard way to solve any problem related to images. The best part about CNNs are that they require much less preprocessing as compared to their predecessors. There are almost no hard coded features required.

Let us take a moment here and understand how a computer sees an image. To a computer an image is nothing but a multidimensional array(3-d if image is RGB). The values range from 0-255. To a computer, image is nothing but a collection of pixel values.

Can’t fully connected networks process images?

It is a common argument that why can’t we simply stretch the multidimensional array and flatten it out and pass it through a fully connected layer.

A pic would be nice here, working on it

Again not getting into mathematics but if we try to understand intuitively, if we reduce the dimension of any multidimensional array to a single dimension we will surely lose some information, right?

Note - There are some dimensionality reduction techniques which might not lose data always.

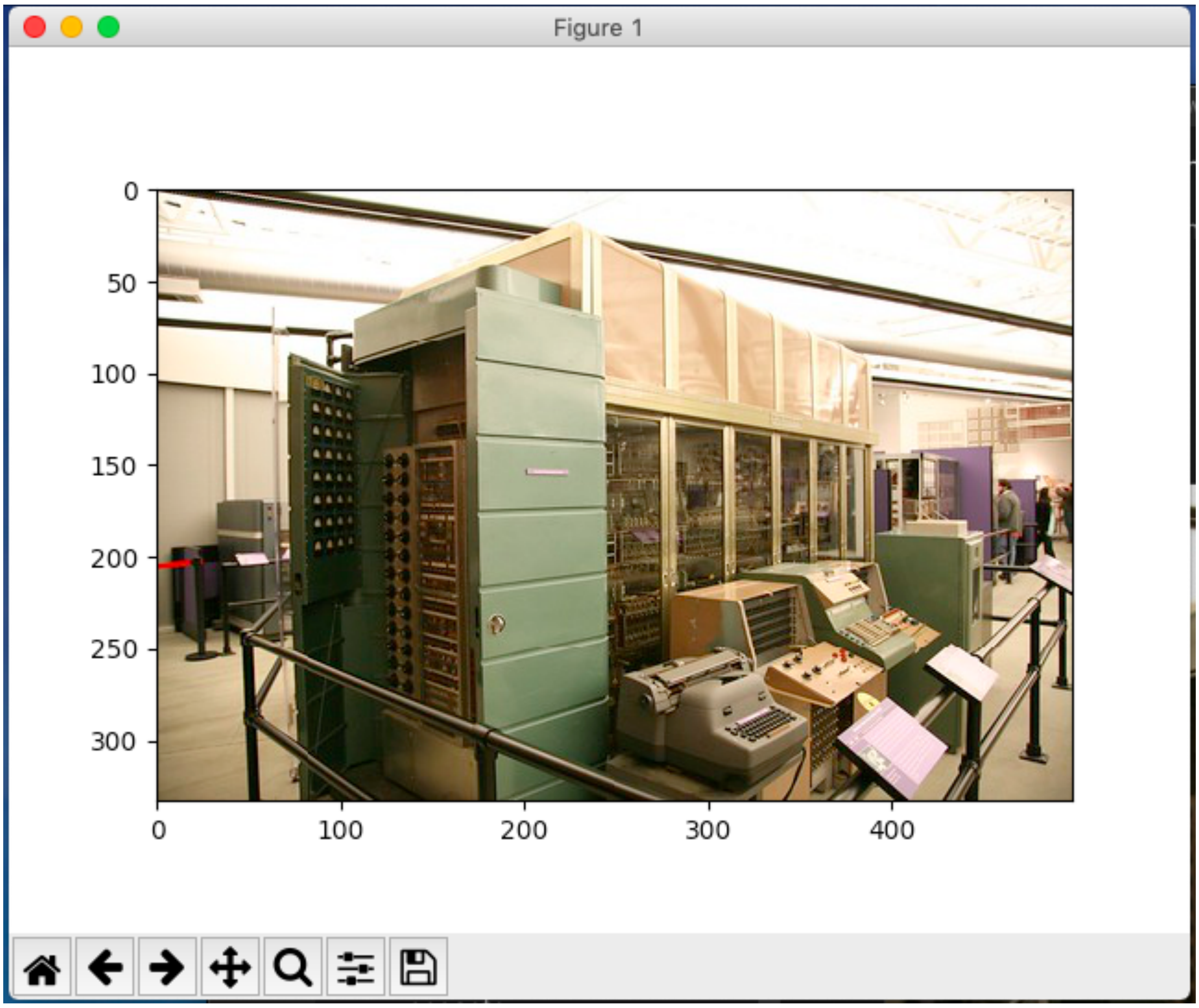
The following is the example image =

source - "Computer History Museum" by Scott Beale is licensed under CC BY-NC 2.0

Let us try to visualize a three dimensional array.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

image = "Chapter-3/assests/img_3d.jpg"

img=mpimg.imread(image)
print(img.shape) #shape will be (333, 499, 3) where 3
represents 3-dimension

imgplot = plt.imshow(img)
plt.show()
```

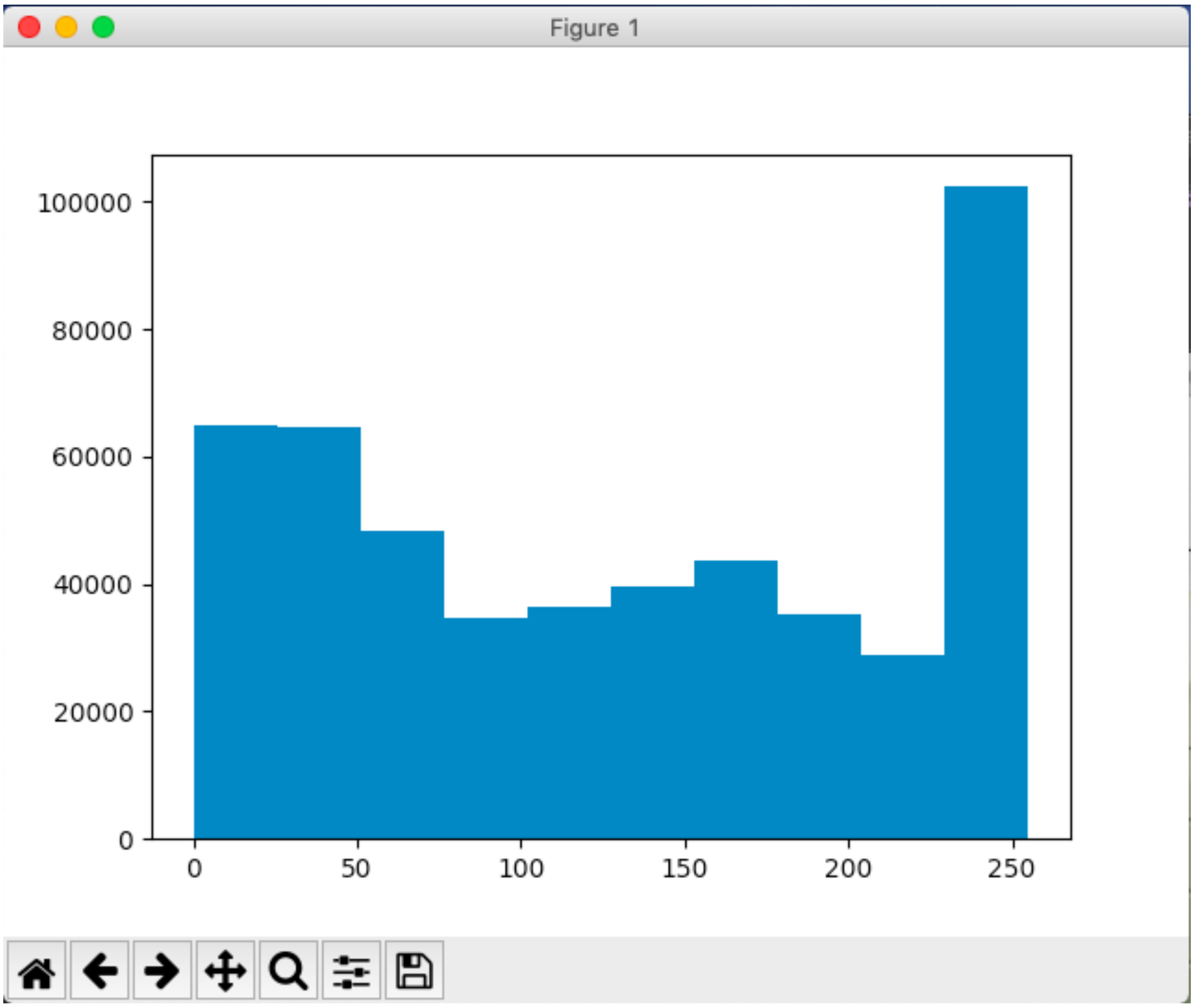
Following will be the output of the code -



If we try to flatten the image we can only visualize it via a histogram.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

image = "Chapter-3/assests/img_3d.jpg"

img=mpimg.imread(image)
print(img.shape)

img = img.flatten()
print(img.shape) # shape of the image will be (498501,) now.

plt.hist(img)
plt.show()
```

Following is the output -



we clearly cannot reason much by seeing th output of histogram.

In cases of extremely basic binary images like MNIST dataset, the multi-layer perceptron or a fully connected network can give decent results but in real world scenario we need a network which can take 3-d images as input and eventually extract relevant features out of it.

A ConvNet is able to successfully capture the Spatial features/context in an image by applying multiple filters or kernels.

A convolutional layer is built by using basically three components - - convolutional layer - pooling layer - fully connected layers

Usually a convNet takes an input image and gives score/class probabilities as an output.

Note - understanding convolutional layer will require some basic mathematics which we will ignore for now. Feel free to raise PRs related to this.

Writing a Convolutional Neural Network in pyTorch

```
#importing necessary libraries

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Note - There is a difference between “nn.Conv2d” and
“nn.functional.conv2d”. The “nn.Conv2d” is meant
# to be used as a convolutional layer directly. However
“nn.functional.conv2d” is meant to be used when
# you want your custom convolutional layer logic.

# we will use torchvision library to download and add
transformations to our data
import torchvision as tv
import torchvision.transforms as transforms

# our transformation pipeline
transform = transforms.Compose([tv.transforms.ToTensor(),
            tv.transforms.Normalize((0.4914, 0.4822, 0.4465),
(0.247, 0.243, 0.261))])

trainset = tv.datasets.CIFAR10(root="./
data",train=True,download=True,transform=transform)
dataloader =
torch.utils.data.DataLoader(trainset,batch_size=4,
shuffle=False, num_workers=4)

# Defining our model
class OurModel(nn.Module):
    def__init__(self):
        super(OurModel,self).__init__()
        self.conv1 = nn.Conv2d(3,6,5)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6,16,5)
        self.fc1 = nn.Linear(16*5*5,120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,10)

    def forward(self,x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1,16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
net = OurModel()
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(),lr=0.001,weight_decay=
1e-6, momentum = 0.9, nesterov = True)

# training
for epoch in range(2):
    running_loss= 0.0

    for i,data inenumerate(dataloader,0):
        inputs, labels = data
        optimizer.zero_grad()

        # forward prop
        outputs = net(inputs)
        loss = loss_func(outputs, labels)

        # backprop
        loss.backward() # compute gradients
        optimizer.step() # update parameters

        # print statistics
        running_loss += loss.item()
        if i %2000==1999: # print every 2000 mini-batches
            print('[epoch: %d, minibatch: %5d] loss: %.3f'%
(epoch +1, i +1, running_loss /2000))
            running_loss = 0.0

print("Training finished!")
```

That's is how you can write and quickly run a basic convolutional network in pytorch.

Now let's verify our model

```
testset = torchvision.datasets.CIFAR10(root='./data',
train=False,
                                       download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False,
num_workers=2)

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

dataiter = iter(testloader)
images, labels = dataiter.next()

outputs = net(images)
```

```
_, predicted = torch.max(outputs, 1)

print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for
j in range(4)))
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
for j in range(4)))
```

Chapter 4

How to make a computer remember stuff?

- Inspiration
- What are RNNs?
- LSTM and its variants
- Generating words with LSTMs in pytorch
- Advances in the field

Chapter 5

Where to go from here?

If you have come this far, it means that you are now familiar with basic terminologies of the deep learning world. I hope that this book has served you well in acquainting you with the deep learning world and a little bit with pyTorch.

Now, Following is how I will advise you to advance in this field -

1. Start with Fast.ai MOOCs.
2. Alongwith Fast ai's courses, keep reading lecture notes from Andrej Karpathy.
3. Once you are comfortable with Jeremy Howard's Fast Ai. Start CS 231n.
4. I'd also suggest you to dive in Machine Learning world now. CS 229.

The resources mentioned above should give you a great boost in your learning journey.

Once you find yourself ready to take challenges then you can read - - Neural Networks & Learning Machines - Simon Haykin - Deep Learning Book - Ian Goodfellow and Yoshua Bengio and Aaron Courville

Along the way, try to keep track on the popular papers which are released - CVPR - NeurIPS - ICML

Honestly, I have lost the track here. 250+ papers were released this year.

Also subscribe to the following newsletters - 1. OpenAI 2. For NLP

Feel free to add here by raising Pull Requests! :)