1. Overview of Deep Learning using PyTorch

Why Deep Learning has emerged recently?

Deep learning was in research for a long time. But the reason for its recent emergence is due to

- The availability of powerful computing machines, especially in the cloud
- The availability of huge amounts of data

Moore's Law states that the processing power of machines will double every year. Hence currently our machines are capable of training deep learning models with hundreds of layers. The data generation has also exploded due to the use of large amounts of digital devices leading to digital footprints.

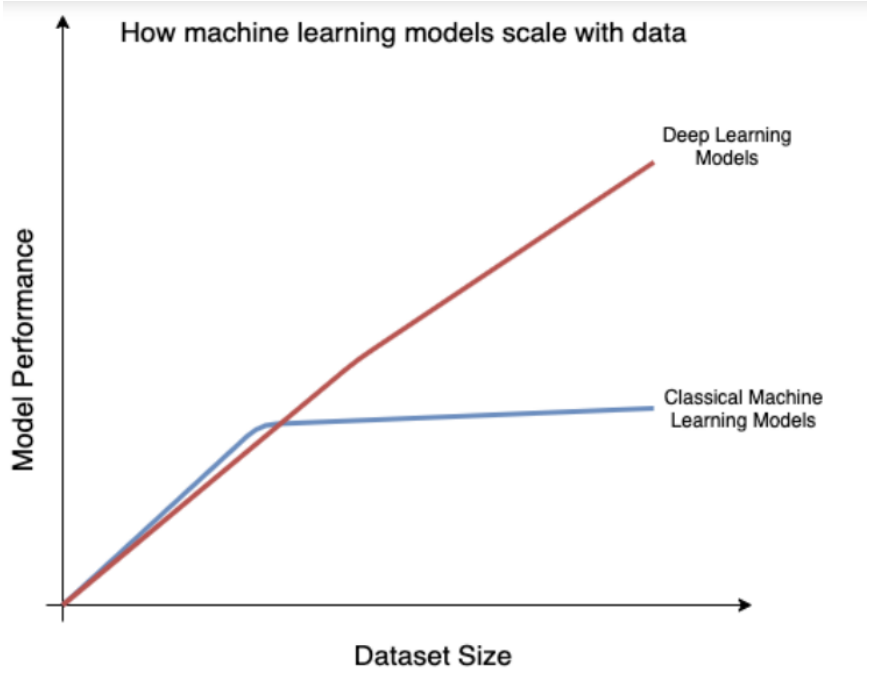


Figure 1.2 – Model performance versus dataset size

Traditional Machine Learning models require feature engineering on data and that affects the performance of the trained models. But with deep learning, we don't need to manually craft hand engineered features from dataset and they perform really well.

Types of Neural Network Layers

1. **Fully-Connected or Linear** - In this type of architecture each neuron of the preceding layer is connecting to each neuron of the succeeding layer.

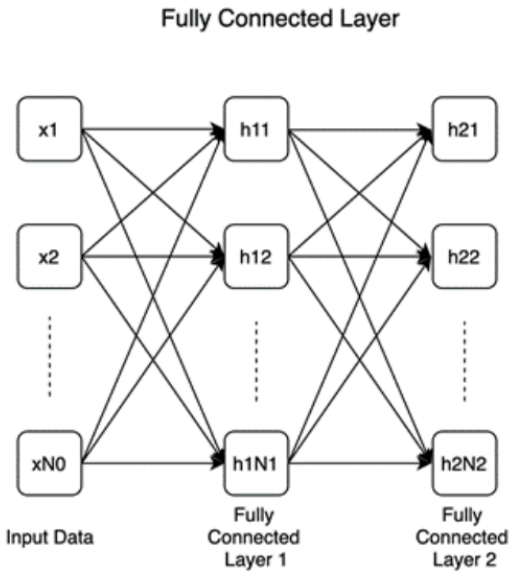


Figure 1.3 – Fully connected layer

2. **Convolution** - The input is passed through a convolution layer to get the output. This is called a Kernel or Filter.

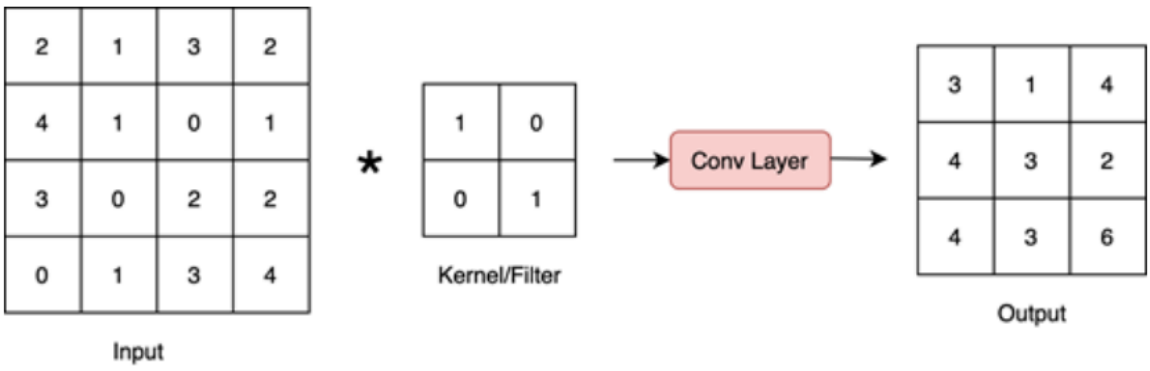


Figure 1.4 – Convolutional layer

3. **Recurrent** - Recurrent layers have an advantage over fully connected layers in that they exhibit memorizing capabilities, which comes in handy working with sequential data where one needs to remember past inputs along with the present inputs.

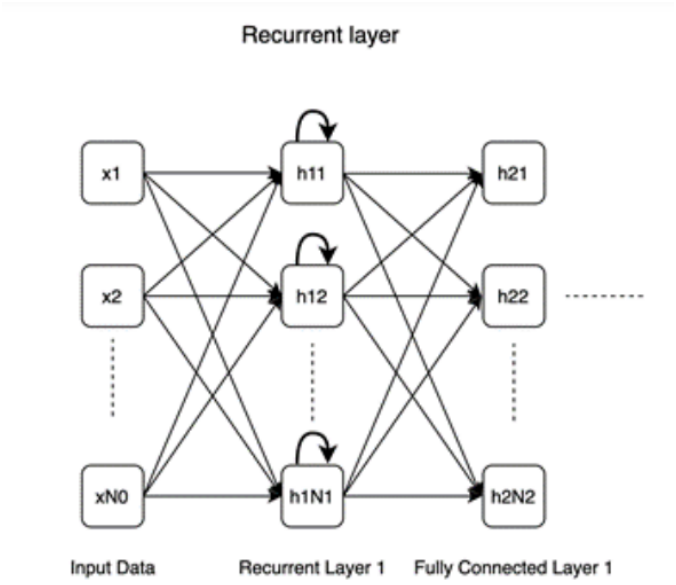


Figure 1.5 – Recurrent layer

4. **DeConv** - It is the reverse of Convolution layer. This layer works in expanding the input data spatially and is widely used in models that generate or reconstruct images.

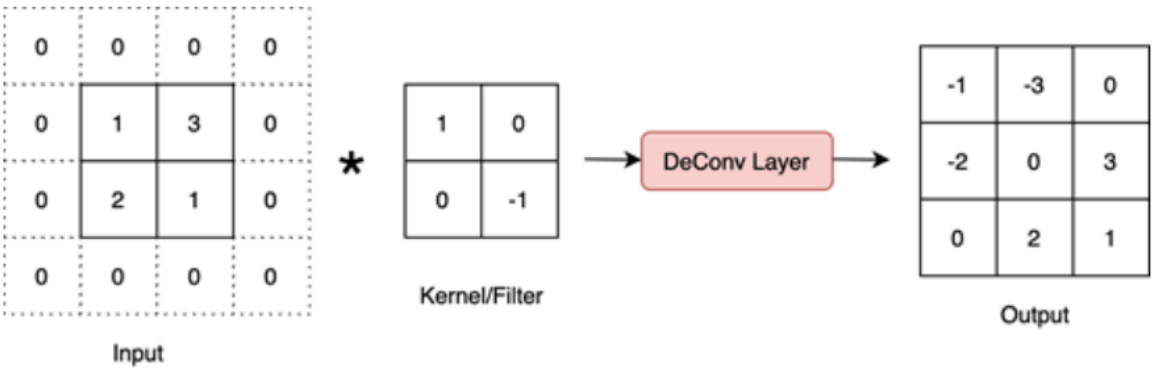


Figure 1.6 – Deconvolutional layer

5. Pooling - There are various types of Pooling like Max, Min or Mean. it is aimed to extract more information from a particular input data. The following example is a max-pooling layer that pools the highest number each from 2x2 sized subsections of the input.

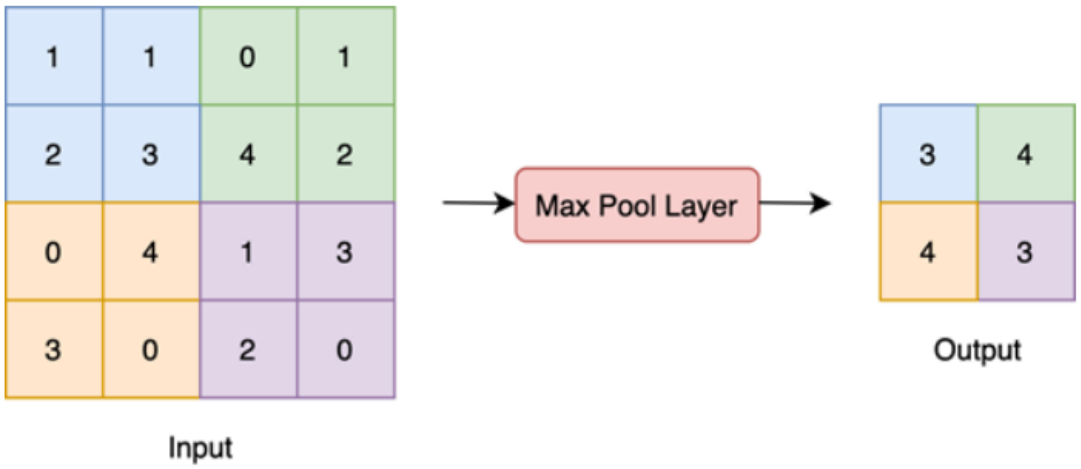


Figure 1.7 – Pooling layer

6. **Dropout** - Essentially, in a dropout layer, some neurons are temporarily switched off (marked with X in the diagram), that is, they are disconnected from the network. Dropout helps in model regularization as it forces the model to function well in sporadic absences of certain neurons, which forces the model to learn generalizable patterns instead of memorizing the entire training dataset.

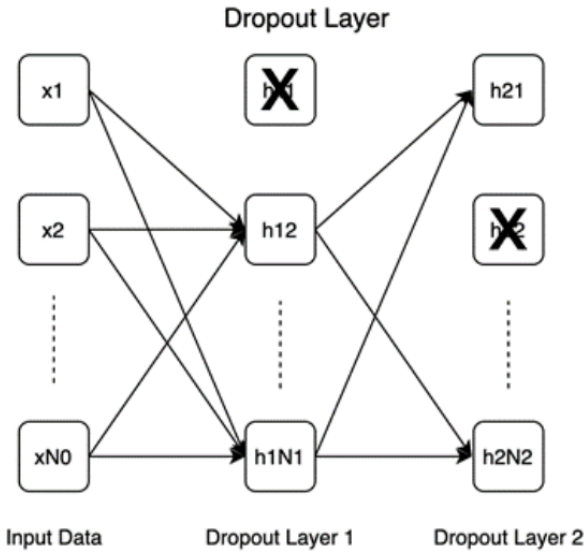


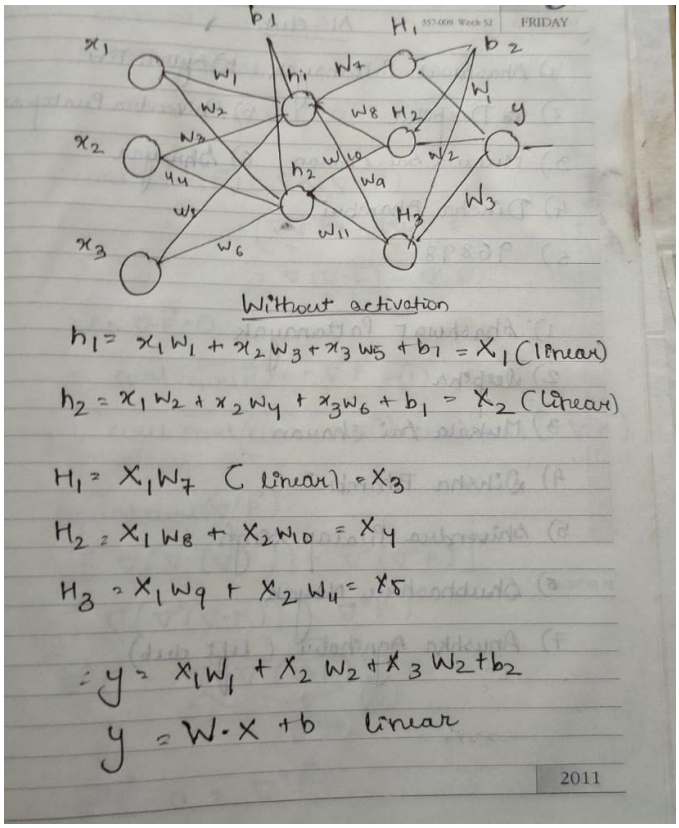
Figure 1.8 – Dropout layer

Activation Functions

They are mathematical functions that is applied to the input data when it is passed from one layer to another. But the question arise why do we even need it?

Why do we need Activation Functions?

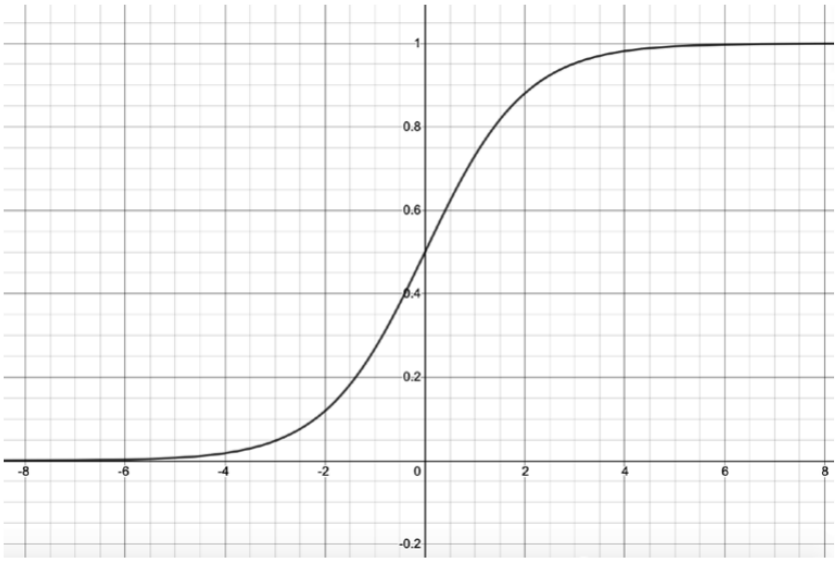
The main goal of activation functions is to induce the non-linearity in the neural network. If the activation functions are not present then our inputs and outputs are simply mapped by a linear mapping. Activation functions are necessary to prevent linearity. Without them, the data would pass through the nodes and layers of the network only going through linear functions (a*x+b). The composition of these linear functions is again a linear function and so no matter how many layers the data goes through, the output is always the result of a linear function. More info



Various Activation Functions

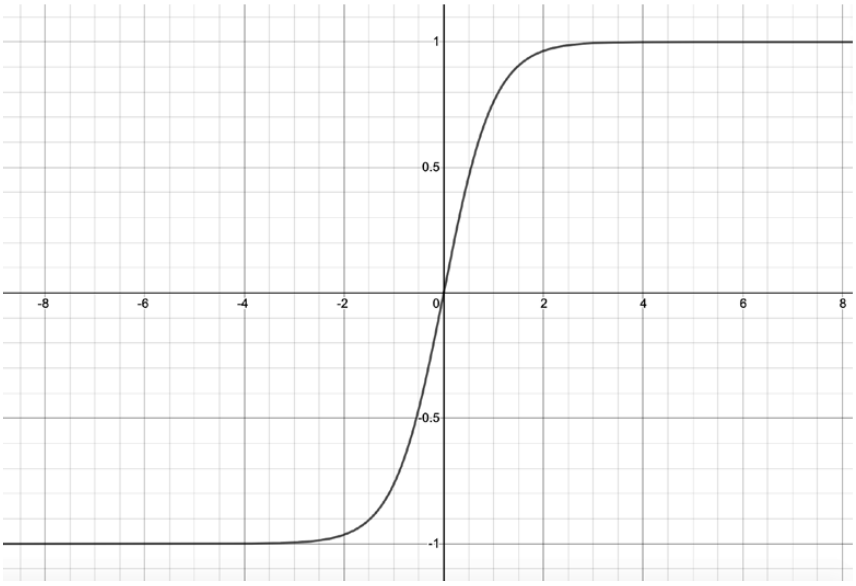
- **Sigmoid: T**he sigmoid function takes in a numerical value x as input and outputs a value y in the range (0, 1).

  $y = f(x) = \sigma(z) = \frac{1}{1+e^{-z}}$



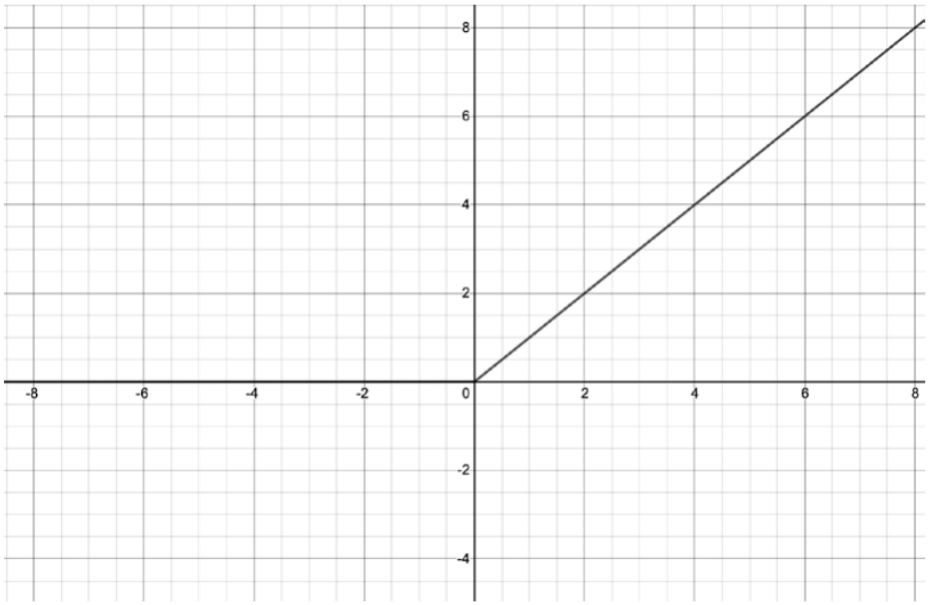
- **TanH -** Contrary to sigmoid, the output y varies from -1 to 1 in the case of the TanH activation function. Hence, this activation is useful in cases where we need both positive as well as negative outputs.

  $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$



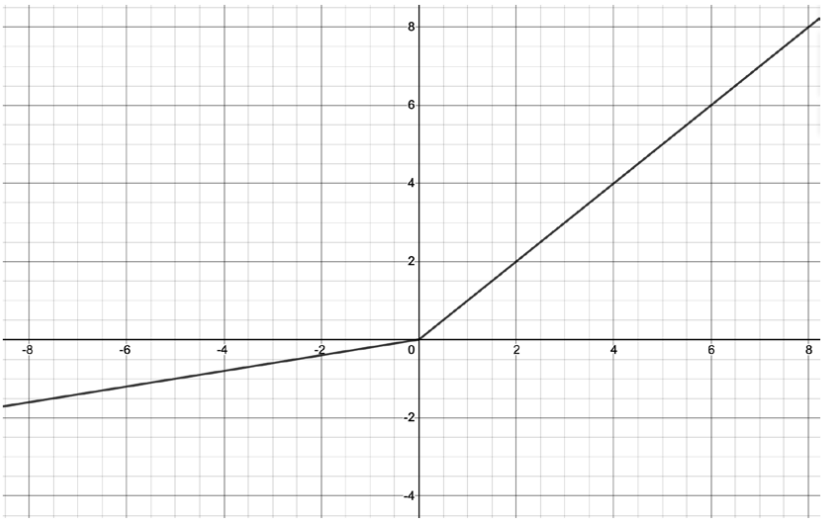
- **Rectified Linear Units (ReLUs) -** ReLU in comparison with the sigmoid and TanH activation functions is that the output keeps growing with the input whenever the input is greater than 0. This prevents the gradient of this function from diminishing to 0 as in the case of the previous two activation functions. Although, whenever the input is negative, both the output and the gradient will be 0.

  $y = f(x) = max(0, x)$



- **Leaky ReLU:** ReLUs entirely suppress any incoming negative input by outputting 0. We may, however, want to also process negative inputs for some cases. Leaky ReLUs offer the option of processing negative inputs by outputting a fraction k of the incoming negative input. This fraction k is a parameter of this activation function.

  $y = f(x) = max(kx, x)$



There are Many more activation functions like Swish, Softmax, ELU, PreReLU.

Optimization Methods

Optimization as the name suggests is a a method to update and tune our deep learning models. The process used to tune parameters is called Backpropagation. This is done on the basis of a Loss/Cost Function.

But what are these?

Loss Function: When we pass a set of inputs to the neural network we get an output. But in case of training we already have the actual output for that input set. So the difference between actual output and predicted output is called Loss function. Now if we calculate this for each input separately it is loss function. But the sum of losses for all inputs to get a total loss is called cost function.

This loss is then backpropagated to the previous layers using gradient descent and the ***chain rule of differentiation***.

The parameters or weights at each layer are accordingly modified in order to minimize the loss. The extent of modification is determined by a coefficient, which varies from 0 to 1, also known as the ***learning rate***.

- **Stochastic Gradient Descent (SGD)**

$$\beta = \beta - \alpha * \frac{\delta L(X, y, \beta)}{\delta \beta}$$

β is the parameter of the model and X and y are the input training data and the corresponding labels respectively. L is the loss function and α is the learning rate. SGD performs this update for every training example pair (X, y). A variant of this – mini-batch gradient descent – performs updates for every k examples, where k is the batch size. Gradients are calculated altogether for the whole mini-batch. Another variant, batch gradient descent, performs parameter updates by calculating the gradient across the entire dataset.

For more details read the Book on various optimization schedules.

PyTorch Library

The initial difference between these two was that PyTorch was based on eager execution whereas TensorFlow was built on graph-based deferred execution. Although, TensorFlow now also provides an eager execution mode.

- Eager execution is basically an imperative programming mode where mathematical operations are computed immediately.
- A deferred execution mode would have all the operations stored in a computational graph without immediate calculations and then the entire graph would be evaluated later.
- Eager execution is considered advantageous for reasons such as intuitive flow, easy debugging, and less scaffolding code

PyTorch Modules

The PyTorch library, besides offering the computational functions as NumPy does, also offers a set of modules that enable developers to quickly design, train, and test deep learning models.

torch.nn

When building a neural network architecture, the fundamental aspects that the network is built on are the number of layers, the number of neurons in each layer, and which of those are learnable, and so on. The PyTorch nn module enables users to quickly instantiate neural network architectures by defining some of these high-level aspects as opposed to having to specify all the details manually.

Without nn module the code is

```
import math
# we assume a 256-dimensional input and a 4-dimensional output
for this 1-layer neural network
# hence, we initialize a 256x4 dimensional matrix filled with
random values
weights = torch.randn(256, 4) / math.sqrt(256)

# we then ensure that the parameters of this neural network
# are trainable, that is, the numbers in the 256x4 matrix can be
# tuned with the help of backpropagation of gradients
weights.requires_grad_()
# finally we also add the bias weights for the 4-dimensional
output, and make these trainable too
bias = torch.zeros(4, requires_grad=True)
```

Instead using nn module to represent the same thing

```
nn.Linear(256, 4)
```

Within the torch.nn module, there is a submodule called torch.nn.functional.
This submodule consists of all the functions within the torch.nn module whereas all the other submodules are classes. These functions are loss functions, activating functions, and also neural functions that can be used to create neural networks in a functional

manner (that is, when each subsequent layer is expressed as a function of the previous layer) such as pooling, convolutional, and linear functions.

```
import torch.nn.functional as F
loss_func = F.cross_entropy
loss = loss_func(model(X), y)

#X is input and y is the target output
# model is the neural network model
```

torch.optim

This contains varuous tools and functionalities for running optimization schedules.

```
opt = optim.SGD(model.parameters(), lr=lr)
opt.step()
opt.zero_grad()
```

torch.utils.data

Under the utis.data module, torch provides its own dataset and DatasetLoader classes, which are extremely handy due to their abstract and flexible implementations. Basically, these classes provide intuitive and useful ways of iterating and performing other such operations on tensors. Using these, we can ensure high performance due to optimized tensor computations and also have fail-safe data I/O.

```
from torch.utils.data import (TensorDataset, DataLoader)
train_dataset = TensorDataset(x_train, y_train)
train_dataloader = DataLoader(train_dataset, batch_size=bs)
```

Doing Manually

```
for i in range((n-1)//bs + 1):
    x_batch = x_train[start_i:end_i]
    y_batch = y_train[start_i:end_i]
    pred = model(x_batch)
```

Using DataLoader

```
for x_batch,y_batch in train_dataloader:
    pred = model(x_batch)
```

Tensors in PyTorch

```
# Initiating tensors
points = torch.tensor([1.0, 4.0, 2.0, 1.0, 3.0, 5.0])

# Fetching first entry
float(points[0])
```

In PyTorch, tensors are implemented as views over a one-dimensional array of numerical data stored in contiguous chunks of memory. These arrays are called storage instances.

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points.storage()
```

Output

```
 1.0
 4.0
 2.0
 1.0
 3.0
 5.0
[torch.FloatStorage of size 6]
```

When we say a tensor is a view on the storage instance, the tensor uses the following information to implement the view:

• Size

- Storage
- Offset
- Stride

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points.size()

//Output
torch.size([3,2])
```

```
points.storage_offset()

//output
0
```

The offset here represents the index of the first element of the tensor in the storage array. Because the output is 0, it means that the first element of the tensor is the first element in the storage array

```
points[1].storage_offset()

output
2
```

Because points[1] is [2.0, 1.0] and the storage array is [1.0, 4.0, 2.0,1.0, 3.0, 5.0], we can see that the first element of the tensor [2.0, 1.0], that is, 2.0 is at index 2 of the storage array.

```
points.stride()

output
(2,1)
```

stride contains, for each dimension, the number of elements to be skipped in order to access the next element of the tensor. So, in this case, along the first dimension, in order to access the element after the first one, that is, 1.0 we need to skip 2 elements

(that is, 1.0 and 4.0) to access the next element, that is, 2.0. Similarly, along the second dimension, we need to skip 1 element to access the element after 1.0, that is, 4.0.

Data Types Allowed in PyTorch

```
• torch.float32 or torch.float—32-bit floating-point
• torch.float64 or torch.double—64-bit, double-precision floating-point
• torch.float16 or torch.half—16-bit, half-precision floating-point
• torch.int8—Signed 8-bit integers
• torch.uint8—Unsigned 8-bit integers
• torch.int16 or torch.short—Signed 16-bit integers
• torch.int32 or torch.int—Signed 32-bit integers
• torch.int64 or torch.long—Signed 64-bit integers
```

Training a Neural Network using PyTorch

Importing the Modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
```

The Architecture

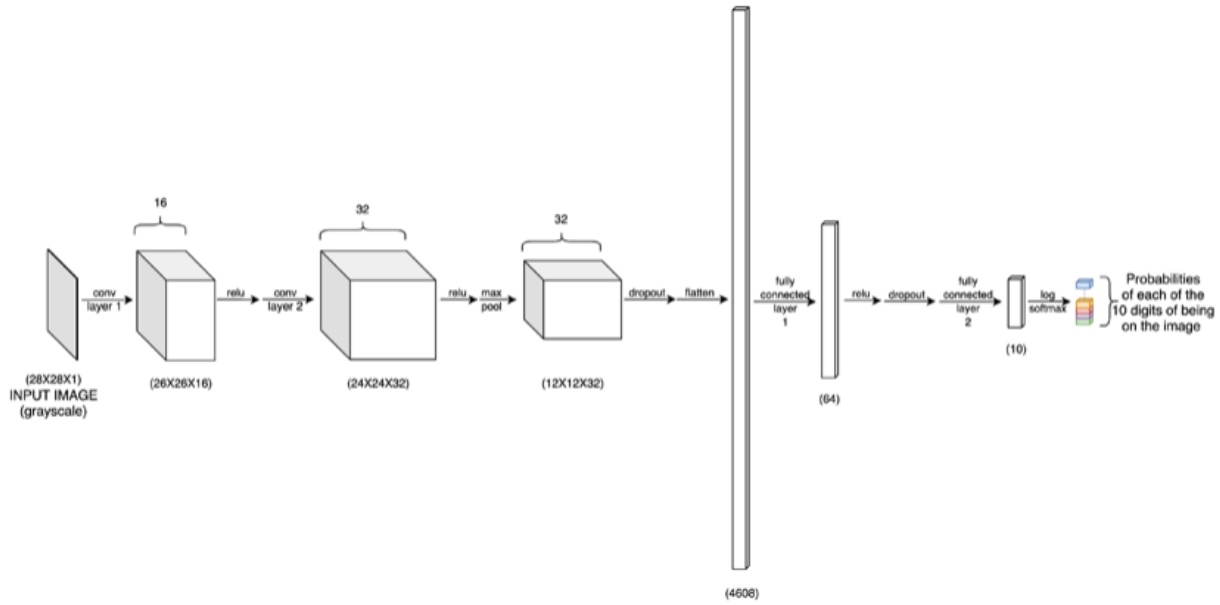


Figure 1.19 – Neural network architecture

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.cn1 = nn.Conv2d(1, 16, 3, 1)
        self.cn2 = nn.Conv2d(16, 32, 3, 1)
        self.dp1 = nn.Dropout2d(0.10)
        self.dp2 = nn.Dropout2d(0.25)
        self.fc1 = nn.Linear(4608, 64) # 4608 is basically 12 X 12 X 32
        self.fc2 = nn.Linear(64, 10)
    def forward(self, x):
        x = self.cn1(x)
        x = F.relu(x)
        x = self.cn2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dp1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dp2(x)
        x = self.fc2(x)
        op = F.log_softmax(x, dim=1)
        return op
```

The **init** function defines the core architecture of the model, that is, all the layers with the number of neurons at each layer. And the forward function, as

the name suggests, does a forward pass in the network. Hence it includes all the activation functions at each layer as well as any pooling or dropout used after any layer. This function shall return the final layer output, which we call the prediction of the model, which has the same dimensions as the target output (the ground truth).

Paramaters of the Convolution Layer

self.cn1 = nn.Conv2d(Param1, Param2, Param3, Param4)

Param1 → Input channel. 1 is for the grayscale images that we will feed to the model. if it is a coloured image then we will pass 3.

Param2 → The number of output channels. it is always higher than the input channels. In this case we have taken 16 output channels. So it essentially means our layer is trying to detect 16 different kinds of inofmration from the input image. Each of these channels are called a **feature map.** Ech of this feature maps has a dedicated kernels extracting the features for them.

As we move deeper we extend the number of output channels. We used 32 output channels in second Convolutional layer toextract more kinds of features.

Param3 → Kernel/Filter size. Here we have taken 3x3 kernel. It always should be odd to symmetrically distribute around the central pixel. But why 3x3?

Kernel of size 1 would be very small as it will cover very little area and extract very less information. On the other hand higher size such as 27x27 would be very close to the 28x28 image size and hence give us coarse-grained features. The visual features are local in the MNIST images, so we used smaller kernels. 3x3 is the most commonly used kernels in CNNs.

Param4 → It is the stride. Here we took stride as 1. As our kernel is small tride is also small to effectively pick up essential features continously. Taking strides such as 10 would result in skipping many pixels in the image. In case of 100 kernel size a stride of 10 was reasonable.

The larger the stride, the lower the number of convolution operations but the smaller the overall field of view for the kernel.

Optimization and Backpropagation

```
def train(model, device, train_dataloader, optim, epoch):
    model.train()
    for b_i, (X, y) in enumerate(train_dataloader):
        X, y = X.to(device), y.to(device)
        optim.zero_grad()
        pred_prob = model(X)
        loss = F.nll_loss(pred_prob, y) # nll is the
negative likelihood loss
        loss.backward()
        optim.step()
        if b_i % 10 == 0:
            print('epoch: {} [{}/{} ({:.0f}%)]\t training
loss: {:.6f}'.format(
                epoch, b_i * len(X), len(train_
 dataloader.dataset),
                100. * b_i / len(train_dataloader), loss.
 item()))
```

This iterates through the dataset in batches, makes a copy of the dataset on the given device, makes a forward pass with the retrieved data on the neural network model, computes the loss between the model prediction and the ground truth, uses the given optimizer to tune model weights, and prints training logs every 10 batches. The entire procedure done once qualifies as 1 epoch, that is, when the entire dataset has been read once.

Test routine Code

```
def test(model, device, test_dataloader):
    model.eval()
    loss = 0
    success = 0
    with torch.no_grad():
        for X, y in test_dataloader:
            X, y = X.to(device), y.to(device)
            pred_prob = model(X)
            loss += F.nll_loss(pred_prob, y,
reduction='sum').item()  # loss summed across the batch
            pred = pred_prob.argmax(dim=1,
 keepdim=True)  # us argmax to get the most
```

```
 likely prediction
            success += pred.eq(y.view_as(pred)).sum().
item()
    loss /= len(test_dataloader.dataset)
    print('\nTest dataset: Overall Loss: {:.4f}, Overall
Accuracy: {}/{} ({:.0f}%)\n'.format(
        loss, success, len(test_dataloader.dataset),
        100. * success / len(test_dataloader.dataset)))
```

Most of this function is similar to the preceding train function. The only difference is that the loss computed from the model predictions and the ground truth is not used to tune the model weights using an optimizer. Instead, the loss is used to compute the overall test error across the entire test batch.

Loading Data to setup Dataset

```
# The mean and standard deviation values are calculated
as the mean of all pixel values of all images in the
training dataset
train_dataloader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(), transforms.Normalize((0.1302,),
(0.3069,))])), # train_X.mean()/256. and train_X.
std()/256.
    batch_size=32, shuffle=True)
test_dataloader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1302,),
(0.3069,))
                   ])),
    batch_size=500, shuffle=False)
```

We set batch_size to 32, which is a fairly common choice. Usually, there is a trade-off in deciding the batch size. A very small batch size can lead to slow training due to frequent gradient calculations and can lead to extremely noisy gradients. Very large batch sizes can, on the other hand, also slow down raining due to a long waiting time to calculate gradients. It is mostly not worth waiting long before a single gradient update. It is rather advisable to make frequent, less precise gradients as it will eventually lead the model to a better set of learned parameters

Defining Optimizer and Device to use

```
torch.manual_seed(0)
device = torch.device("cpu")
model = ConvNet()
optimizer = optim.Adadelta(model.parameters(), lr=0.5)
```

We also set a seed to avoid unknown randomness and ensure repeatability.

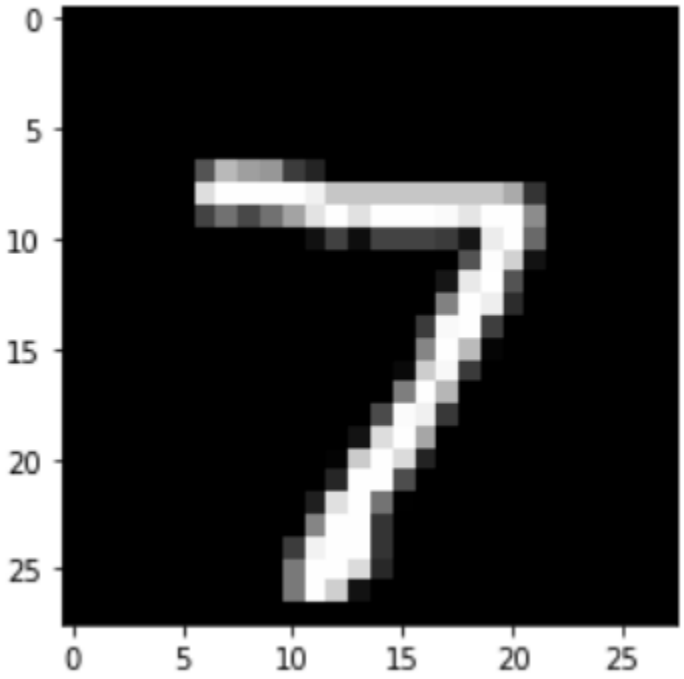
Running Model for Epochs

```
for epoch in range(1, 3):
    train(model, device, train_dataloader, optimizer,
epoch)
    test(model, device, test_dataloader)
```

Testing our Model

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)
plt.imshow(sample_data[0][0], cmap='gray',
interpolation='none')
```

Output



Prediction using Model

```
print(f"Model prediction is : {model(sample_data).data.
max(1)[1][0]}")
print(f"Ground truth is : {sample_targets[0]}")
```

We first calculate the class with maximum probability using the max function on axis=1. The max function outputs two lists – a list of probabilities of classes for every sample in sample_data and a list of class labels for each sample. Hence, we choose the second list using index [1]. We further select the first class label by using index [0] to look at only the first sample under sample_data.

The forward pass of the neural network done using model() produces probabilities. Hence, we use the max function to output the class with the maximum probability