# A refresher on deep learning

- The following graph indicates how deep learning models can leverage large amounts of data better than the classical machine models

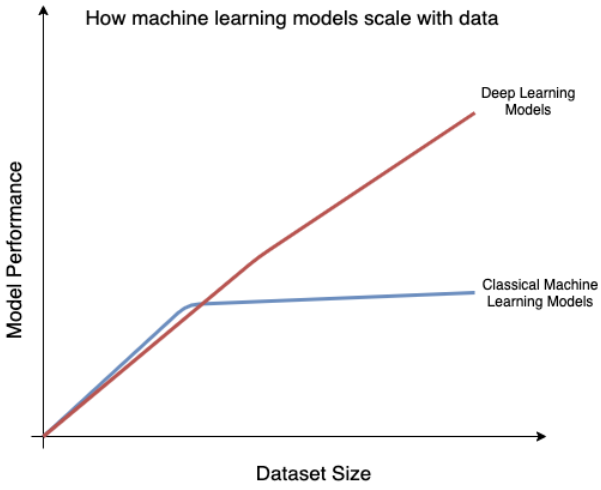



Figure 1.2 – Model performance versus dataset size

  - As can be seen in the graph, deep learning performance isn't necessarily distinguished up to a certain dataset size. However, as data size starts to further increase, deep neural networks begin outperforming the non-deep learning models

- A deep learning model can be built based on various types of neural network architectures that have been developed over the years. A prime distinguishing factor between the different architectures is the type and combination of layers that are used in the neural network. Some of the well-known layers are the following
  - Fully-connected or linear:

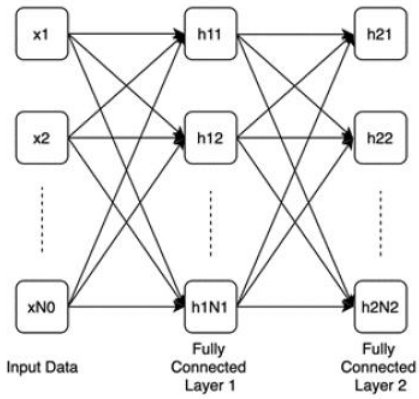
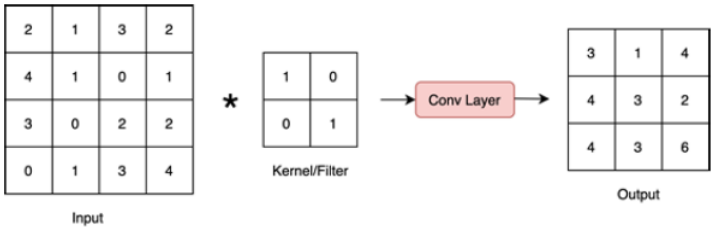



Figure 1.3 – Fully connected layer

  - Convolutional




- Recurrent

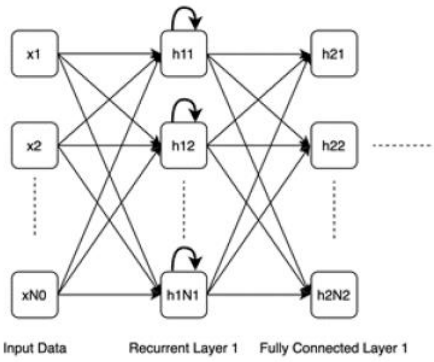



Figure 1.5 – Recurrent layer

  - 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

- DeConv (the reverse of a convolutional layer)

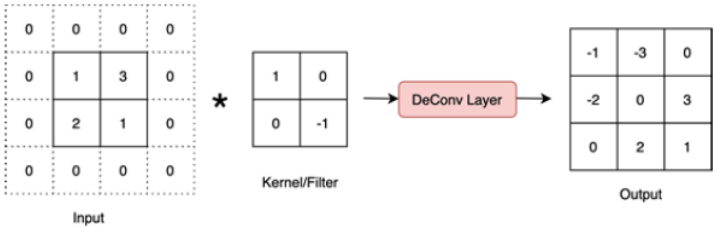



Figure 1.6 – Deconvolutional layer

  - The layer expands the input data spatially and hence is crucial in models that aim to generate or construct images, for example.

- Pooling

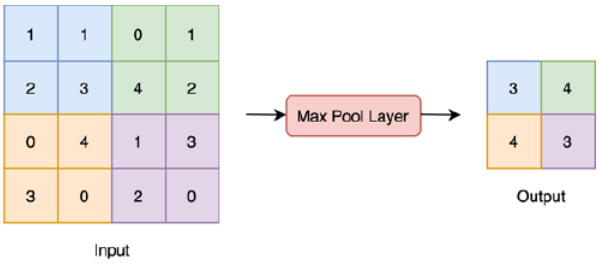



Figure 1.7 – Pooling layer

- Dropout

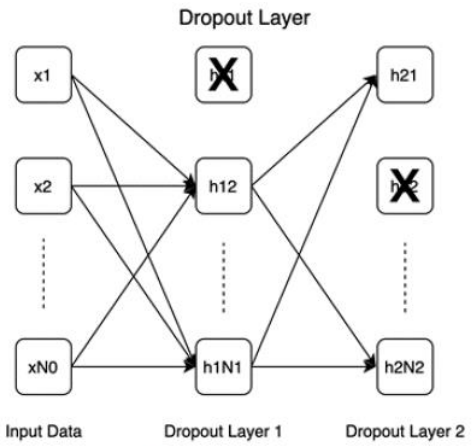



Figure 1.8 – Dropout layer

- ✧ 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

## *Activation functions*

- Sigmoid: a logistic function

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

The function is shown in graph form as follows:

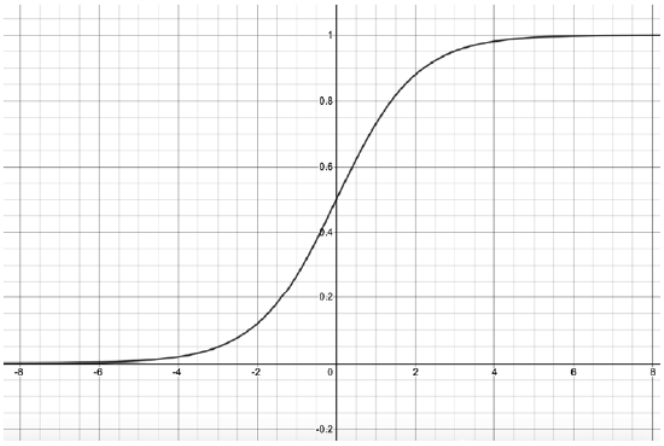



Figure 1.10 – Sigmoid function

- ➢ The function takes in a numerical x as input and outputs a value y in the range (0, 1)

- TanH

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

The function is shown in graph form as follows:

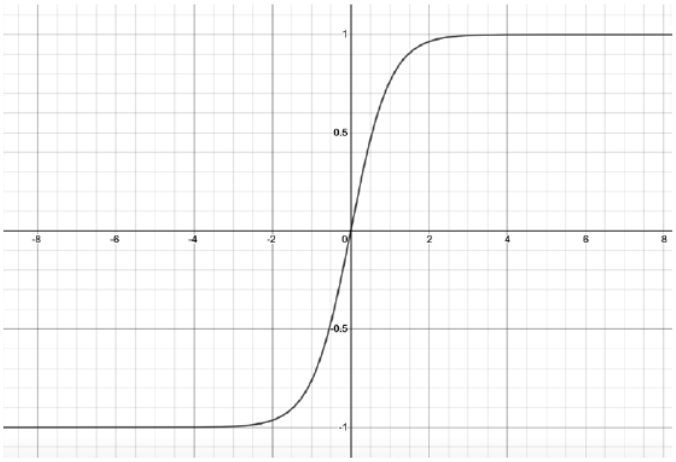



Figure 1.11 – TanH function

- The output y varies from -1 to 1, this activation is useful in cases where we need both positive as well as negative outputs

- Rectified linear units (ReLU)

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

The function is shown in graph form as follows:

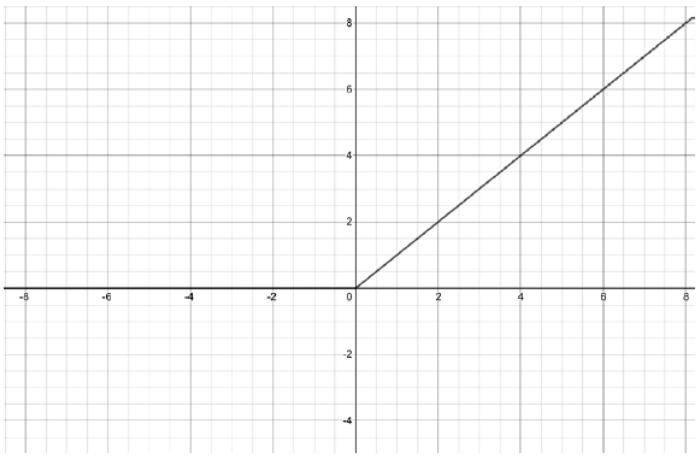



Figure 1.12 – ReLU function

- 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. Although, whenever the input is negative, both the output and gradient will be 0.

- Leaky ReLU
  - Leaky ReLU 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.

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

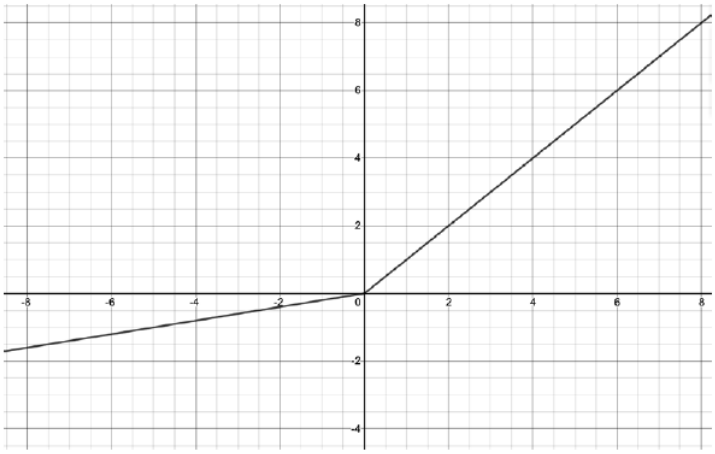


Figure 1.13 – Leaky ReLU function

## Optimization schedule

- Stochastic gradient descent (SGD)

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

  - 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
  - Another variant, batch gradient descent, performs parameter updates by calculating the gradient across the entire dataset

- Adagrad
  - We used a single learning rate for all the parameters of the model in the above example. But different parameters might need to be updated at different paces, especially in cases of sparse data, where some parameters are more actively involved in feature extraction than others.
  - Adagrad introduces the idea of pre-parameter updates,

$$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta \beta_i^t}$$

  - Here, we use the subscript i to denote th ith parameter and the superscript t as time step t of the gradient descent iterations. $SSG_i^t$ is the sum of squared gradients for the ith parameter starting from time step 0 to time step t. $\epsilon$ is used to denote a small value added to SSG to avoid division by zero.
  - Dividing the global learning rate $\alpha$ by the square root of SSG ensures that the learning rate for frequently changing parameters lowers faster than the learning rate for rarely updated parameters.

- Adadelta

- In Adagrad, the denominator of the learning rate is a term that keeps on rising in value due to added squared terms in every time step. This causes the learning rates to decay to vanishingly small values.
- To tackle this problem, Adadelta introduces the idea of computing the sum of squared gradients only up to previous time steps.
- In fact, we can express it as a running decaying average of the past gradients

$$SSG_i^t = \gamma * SSG_i^{t-1} + (1-\gamma) * (\frac{\delta L(X,y,\beta)}{\delta \beta_i^t})^2$$

- $\gamma$ here is the decaying factor we wish to choose for the previous sum of squared gradients.
- With this formulation, we ensure that the sum of squared gradients does not accumulate to a large value, thanks to the decaying average.
- Once $SSG_i^t$ is defined, we can use the Adagrad equation to define the update step for Adadelta.

- If we look closely at the Adagrad equation, the root mean squared gradient is not a dimensionless quantity and hence should ideally not be used as a coefficient for the learning rate.
  - To resolve this, we define another running average, this time for the squared parameter updates.
  - Let's first define the parameter update

$$\Delta\beta_i^t = \beta_i^{t+1} - \beta_i^t = -\frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X,y,\beta)}{\delta \beta_i^t}$$

  - And then, similar to the running decaying average of the past gradients equation, we can define the squared sum of parameter updates as

$$SSPU_i^t = \gamma * SSPU_i^{t-1} + (1-\gamma) * (\Delta\beta_i^t)^2$$

  - Here, SSPU is the sum of squared parameter updates. Once we have this, we can adjust for the dimensionality problem in the Adagrad equation with the final Adadelta equation

$$\beta_i^{t+1} = \beta_i^t - \frac{\sqrt{SSPU_i^t + \epsilon}}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X,y,\beta)}{\delta \beta_i^t}$$

  - Noticeably, the final Adadelta equation doesn't require any learning rate.
    - ✧ One can still provide a learning rate as a multiplier.
    - ✧ Hence, the only mandatory hyperparameter for this optimization schedule is the decaying factors.

- RMSprop
  - The internal workings of RMSprop are pretty similar to that of Adadelta.
  - The only difference is that RMSProp does not adjust for the dimensionality problem and hence the update equation stays the same as the equation in the Adagrad section.
  - This essentially means that we do need to specify both a base learning rate as well as a decaying factor in the case of RMSProp

- Adaptive Moment Estimation (Adam)

- This is another optimization schedule that calculates customized learning rates for each parameter. Just like Adadelta and RMSprop, Adam also uses the decaying average of the previous squared gradients as demonstrated in the first equation in the Adadelta section.
- However, it also uses the decaying average of previous gradient values

$$SG_i^t = \gamma' * SG_i^{t-1} + (1-\gamma') * \frac{\delta L(X,y,\beta)}{\delta \beta_i^t}$$

- SG and SSG are mathematically equivalent to estimating the first and second moments of the gradient respectively, hence the name of this method – adaptive moment estimation.
- Usually, $\gamma$ and $\gamma'$ are close to 1 and in that case, the initial values for both SG and SSG might be pushed towards zero. To counteract that, these two quantities are reformulated with the help of bias correction

$$SG_i^t = \frac{SG_i^t}{1-\gamma'} \quad \text{and} \quad SSG_i^t = \frac{SSG_i^t}{1-\gamma}$$

- Once they are defined, the parameter update is expressed as follows

$$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * SG_i^t$$

- Notice Adam optimization involves three hyperparameters – the base learning rate, and the two decaying rates for the gradients and squared gradients

- Adam is one of the most successful, if not the most successful, optimization schedule in recent times for training complex deep learning models.
- So which to use?
  - If dealing with sparse data, then the adaptive optimizers will be advantageous because of the per-parameter learning rate updates. SGD might also find a decent solution but will take much longer in terms of training time
  - Among the adaptive ones, Adagrad has the disadvantage of vanishing learning rates due to a monotonically increasing learning rate denominator
  - RMSProp, Adadelta, and Adam are quite close in terms of their performance on various deep learning tasks
    - ✧ RMSProp is largely similar to Adadelta, except for the use of the base learning rate in RMSprop vs. the use of decaying average of previous parameters updates in Adadelta.
    - ✧ Adam is slightly different in that it also includes the first-moment calculation for gradients and accounts for bias correction.
  - Overall, Adam could be the optimizer to go with, all else being equal.

# Exploring the PyTorch library

- PyTorch is a machine learning library for Python based on the Torch library.
  - PyTorch is competition for the other well-known deep library – TensorFlow, which is developed by Google.

- The initial difference between these two was that PyTorch was based on ***eager execution*** where TensorFlow was built on ***graph-based deferred execution***. Although, TensorFlow now also provides an eager execution mode
  - Eager execution is basically an imperative programming model 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 is more than just a deep learning library. With its NumPy-like syntax/interface it provides tensor computation capabilities with strong acceleration using GPUs.
  - But what is a tensor? Tensors are computational units, very similar to NumPy arrays, except that they can also be used on GPUs to accelerate computing

## 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. The following are some of the most useful modules

- **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 these are learnable, and so on.
  - The PyTorch nn model 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.
  - The following is a one-layer neural network initialization **without** using the nn module

```
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
ar 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)
```

    - We can instead use $nn.Linear(256,\ 4)$ to represent the same thing
- 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 function, activation 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.
- An example of a loss function using the torch.nn.functional module could be

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

  - Here, X is the input, y is the target output, and model is the neural network model

- **torch.optim**
  - as we train a neural network, we back-propagate errors to tune the weights or parameters of the network – the process that we call optimization.
  - The optim module includes all the tools and functionalities related to running various types of **optimization schedules** while training a deep learning model.
  - Let's say we define an optimizer during a training session using the torch.optim modules

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

  - Then, we don't need to manually write the optimization steps such as

```
with torch.no_grad():
    # applying the parameter updates using stochastic gradient
descent
    for param in model.parameters(): param -= param.grad * lr
    model.zero_grad()
```

  - We can simply write this

```
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.
  - For example, let's say we use torch.utils.data.DataLoader as follows

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

  - Then, we don't need to iterate through batches of data manually, like this

```
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)
```

  - We can simply write this instead

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

## Tensor modules

- A tensor is an n-dimensional array on which we can operate mathematical functions, accelerate computations via GPUs, and tensor can also be used to keep track of a computational graph and gradients, which prove vital for deep learning.
  - To run a tensor on a GPU, all we need is to cast the tensor into a certain data type

- Instantiate a tensor in PyTorch

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

  - To fetch the first entry

```
float(points[0])
```

  - To check the shape of tensor

```
points.shape
```

- In PyTorch, tensors are implemented as views over a one-dimensional array of numerical data stored in contiguous chunks of memory. There arrays are called storage instances. Every PyTorch tenso has a storage attribute that can be called to output the underlying storage instance for a tensor

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

    - The output is

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

  - The tensor uses the following information to implement the view
    - Size
    - Offset
    - Stride
    - Storage

- Size is similar to the *shape* attribute in NumPy, which tells us the number of elements across each dimension. The multiplication of these numbers equals the length of the underlying storage instance

```
points.size()
```

```
torch.Size([3, 2])
```

Figure 1.15 – PyTorch tensor size

- The offset 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.storage_offset()
```

```
0
```

Figure 1.16 – PyTorch tensor storage offset 1

```
points[1].storage_offset()
```

```
2
```

Figure 1.17 – PyTorch tensor storage offset 2

- Stride contains, for each dimension, the number of elements to be skipped in order to access the next element of the tensor.
  - 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 (1.0, 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.

```
points.stride()
```

```
(2, 1)
```

Figure 1.18 – PyTorch tensor stride

- The data contained within tensors is of numeric type. Specifically, PyTorch offers the following data types to contain within tensors
  - `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

- An example of how we specify a certain data type to be used for a tensor is as follows

```
points = torch.tensor([[1.0, 2.0], [3.0, 4.0]], dtype=torch.
float32)
```

  - Besides the data type, tensors in PyTorch also need a device specification where they will be stored. A device can be specified as instantiation

```
points = torch.tensor([[1.0, 2.0], [3.0, 4.0]], dtype=torch.
float32, device='cpu')
```

  - Or we can also create a copy of a tensor in the desired device

```
points_2 = points.to(device='cuda')
```

  - We can either allocate a tensor to a CPU (using $device = 'cpu'$), which happens by default if we do not specify a device, or we can allocate the tensor to a GPU (using $device = 'cuda'$)
    - When a tensor is placed on a GPU, the computations speed up and because the tensor APIs are largely uniform across CPU and GPU placed tensors in PyTorch, it is quite convenient to move the same tensor across devices, perform computations, and move it back.
  - If there are multiple devices of the same type, say more than one GPU, we can precisely locate the device we want to place the tensor in using the device index, such as the following

```
points_3 = points.to(device='cuda:0')
```

# Training a neural network using PyTorch

- We will use the MNIST dataset (available at http://yann.lecun.com/exdb/mnist/)
  - The MNIST dataset consists of 60,000 training samples and 10,000 test samples, where each sample is a grayscale image with 28*28 pixels.
  - PyTorch also provides the MNIST dataset under its Dataset module

- Step 1: import a few dependencies

```
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
```

- Step 2: define the model architecture

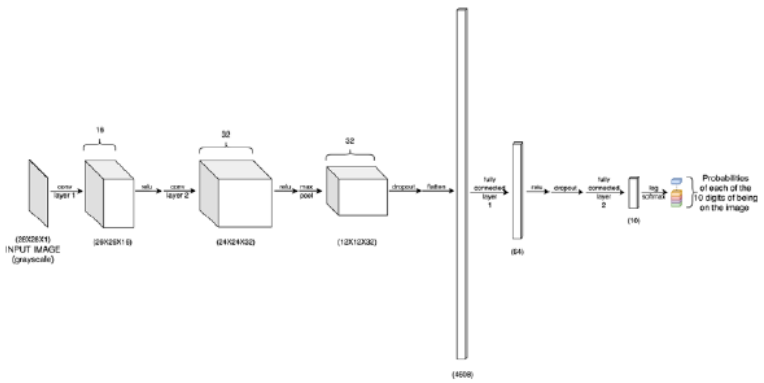



Figure 1.19 – Neural network architecture

  - The model consists of convolutional layers, dropout layers, as well as linear/fully connected layers, all available through the torch.nn module

```
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
```

Should be 1D

- The __init__ function defines the core architecture of the model, all the layers with the number of neurons.
- The forward function does a forward pass in the network. It indicates all the activation functions are each layer as well as any pooling or dropout used after any layer

- Notice that the first convolutional layer has a 1-channel input, a 16-channel output, a kernel size of 3, and a stride of 1. We decided on a kernel size of 3*3 for various reasons.
  - Firstly, kernel sizes are usually odd number so that the input image pixels are symmetrically distributed around a certain pixel. Why not go further to 5, 7, or, even 27?
    - Well, at the extreme end, a 27*27 kernel convolving over a 28*28 image would give us very coarse-grained features.
    - However, the most important visual features in the image are fairly local and hence it makes sense to use a small kernel that looks at a few neighboring pixels at a time, for visual patterns. 3*3 is one of the most common kernel sizes used in CNNs for solving computer vision problems
  - Note that we have two consecutive convolutional layers, both with 3*3 kernels.
    - This, in terms of spatial coverage, is equivalent to using one convolutional layer with a 5*5 kernel.
    - However, using multiple layers with a smaller kernel size is almost always preferred because it results in deeper networks, hence more complex learned features as well as fewer parameters due to smaller kernels.
  - The number of channels in the output of a convolutional layer is usually higher than or equal to the input number of channels.
    - Our first convolutional layer takes in one channel data and outputs 16 channels.
    - This is basically means that the layer is trying to detect 16 different kinds of information from the input image. Each of these channels is called a ***feature map*** and each of them has a dedicated kernel extracting features for them.

- We escalate the number of channels from 16 to 32 in the second convolutional layer, in an attempt to extract more kinds of features from the image.
  - This increment in the number of channels is common practice in CNNs
- Finally, the stride of 1 makes sense, as our kernel size is just 3
  - Keeping a larger stride value – say, 10 – would result in the kernel skipping many pixels in the image and we don't want to do that.
  - If, however, our kernel size was 100, we might have considered 10 as a reasonable stride value
  - The larger the stride, the lower the number of convolution operations but the smaller the overall field of view for the kernel

- Step 3: define the training routine, that is, the actual backpropagation step with ***torch.optim*** module

```
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 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 down once qualifies as 1 epoch, that is, when the entire dataset has been read once
  - Note the function
    - $loss.backward()$ – calculate the gradients
    - $optim.step()$ – update the parameters

- Step 4: similar to the preceding training routine, we write a test routine that can be used to evaluate the model performance on the test set

```
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 used to compute the overall test error across the entire test batch

- Step 5: next, we come to another critical component of this exercise, which is loading the dataset.
  - Thanks to PyTorch's DataLoader module, we can set up the dataset loading mechanism in a few lines of code

```
# 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)
```

  - As you can see, we set batch_size to 32, which is 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.
    - A very large batch size can, however, also low down training due to a long wait 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

➢ For both the training and test dataset, we specify the local storage location we want to save the dataset to, and the batch size, which determines the number of data instances that constitute one pass of training and test run.

- ✧ We also specify that we want to randomly shuffle training data instances to ensure a uniform distribution of data samples across baches.

➢ Finally, we also normalize the dataset to a normal distribution with a specified mean and standard deviation

- Step 6: we defined the training routine earlier. Now is the time to actually define which optimizer and device we will use to run the model training.

```
torch.manual_seed(0)
device = torch.device("cpu")

model = ConvNet()
optimizer = optim.Adadelta(model.parameters(), lr=0.5)
```

➢ We mentioned that Adadelta could be a good choice if we are dealing with sparse data.

- ✧ And this is a case of sparse data, because not all pixels in the image are informative.

➢ Note that if the device is "cuda", then after instantiating the model, most put model into GPU with $model.cuda()$

- Step 7: we start the actual process of training the model for k number of epochs, and we also keep testing the model at the end of each training epoch

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

For demonstration purposes, we will run the training for only two epochs. The output will be as follows:

```
epoch: 1 [0/60000 (0%)]  training loss: 2.306125
epoch: 1 [320/60000 (1%)]        training loss: 1.623073
epoch: 1 [640/60000 (1%)]        training loss: 0.998695
epoch: 1 [960/60000 (2%)]        training loss: 0.953389
epoch: 1 [1280/60000 (2%)]       training loss: 1.054391
epoch: 1 [1600/60000 (3%)]       training loss: 0.393427
epoch: 1 [1920/60000 (3%)]       training loss: 0.235708
epoch: 1 [2240/60000 (4%)]       training loss: 0.284237
epoch: 1 [2560/60000 (4%)]       training loss: 0.203838
epoch: 1 [2880/60000 (5%)]       training loss: 0.292076
epoch: 1 [3200/60000 (5%)]       training loss: 0.541438
epoch: 1 [3520/60000 (6%)]       training loss: 0.411091
epoch: 1 [3840/60000 (6%)]       training loss: 0.323946
epoch: 1 [4160/60000 (7%)]       training loss: 0.296546
                                 ⋮
epoch: 2 [56000/60000 (93%)]     training loss: 0.072877
epoch: 2 [56320/60000 (94%)]     training loss: 0.112689
epoch: 2 [56640/60000 (94%)]     training loss: 0.003503
epoch: 2 [56960/60000 (95%)]     training loss: 0.002715
epoch: 2 [57280/60000 (95%)]     training loss: 0.089225
epoch: 2 [57600/60000 (96%)]     training loss: 0.184287
epoch: 2 [57920/60000 (97%)]     training loss: 0.044174
epoch: 2 [58240/60000 (97%)]     training loss: 0.097794
epoch: 2 [58560/60000 (98%)]     training loss: 0.018629
epoch: 2 [58880/60000 (98%)]     training loss: 0.062386
epoch: 2 [59200/60000 (99%)]     training loss: 0.031968
epoch: 2 [59520/60000 (99%)]     training loss: 0.009200
epoch: 2 [59840/60000 (100%)]    training loss: 0.021790

Test dataset: Overall Loss: 0.0489, Overall Accuracy: 9850/10000 (98%)
```

Figure 1.20 – Training logs

- Step 8: now that we have trained a model with a reasonable test performance, we can also manually check whether the model inference on a sample image is correct

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)

plt.imshow(sample_data[0][0], cmap='gray',
interpolation='none')
```

The output will be as follows:

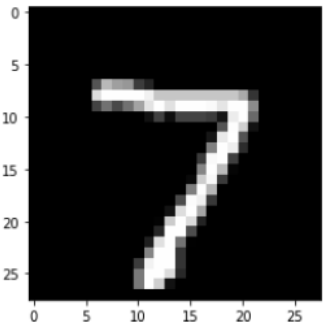



Figure 1.21 – Sample handwritten image

- And now we run the model inference for this image and compare it with the ground truth

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

  - Note that, for predictions, 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 the 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 using index [0] to look at only the first sample under the sample data. The output is

```
Model prediction is : 7
Ground truth is : 7
```

Figure 1.22 – PyTorch model prediction