Deep Learning 101

Gautham Ram Prabaharan

Bens Bernard

Nishant Kumar

Department of Computer Science

Rutgers University

OVERVIEW & PURPOSE

This course intended to introduce the deep learning algorithms by explaining the concepts, the math's associated with, and finally a simple implementation.

I. Preliminaries

   A. Introduction

   B. Data

   C. Model

   D. Optimization

   E. Types of learning

II. Linear Neural Network

   A. Linear Regression

   B. Softmax Regression

III. Multi-Layer Perceptron

   A. Underfitting, Overfitting

   B. Weight decay, Dropout

C. Forward Propagation

D. Backward Propagation

E. Initialization techniques and Numerical Stability

F. Implementation

IV. Convolution Neural Network

A. Convolution Operation

B. Padding and Stride

C. Pooling

D. Fully Connected Network

E. Implementation

V. Recurrent Neural Network

A. Motivation

B. Design

C. Perplexity Measure

D. Implementation

VI. Appendix

OBJECTIVES

1. Understanding of the literature.
2. Presentation of the maths used.
3. Connection between literature and maths through implementation examples.

MATERIALS NEEDED

1. Google Colab/ Jupyter Notebook
2. Pytorch

PRELIMINARIES

A. INTRODUCTION

Deep learning is a part of machine learning that focuses on artificial neural networks which is inspired by the working of the human brain to process through the input to predict the output. In simple terms, deep learning is a machine learning method that takes the input X to predict the output Y. The inputs can be numeric representation of texts, images, sounds or tabular data. The '*Deep*' part in the word 'Deep Learning' refers to the large number of layers in neural networks used to improve its ability to capture complex functions.

For any machine learning problem, the main components that we need to focus on are:

- Data used to train the model
- Model on how to transform the data
- Objective function to judge the correctness of the model
- Algorithm to adjust the model parameters to minimize loss

B. DATA

Data in Deep Learning can be seen as the collection of data points or samples or examples with which the model can be trained which is then used to predict for data that hasn't been encountered before. Each data point typically consists of a collection of either numerical or categorical attributes called *features* and a separate feature for which the prediction is to be made known as the *target*. In the case of a classification problem, the target variable is categorical whereas in the case of a regression problem, the target variable is numerical.

We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications.

The data then can be split into Training Data, Validation Data and Test Data.

Training Data: The data used to train the model.

Validation Data: The data used to tune the hyperparameters of the model.

Test Data: After the model is trained, the model is used on the Test Data to get the predictions and compare it with the actual values to help evaluate the model. This data is not accessed during the training and is used only to assess the final performance.

C. MODEL

Most deep learning algorithms involve transforming the data in some sense. Depending on the type of input data we are dealing with, it is important to select the type of modelling approach we take to tackle the problem. For example, image data can be handled well by the likes of CNNs, textual data by RNNs or depending on the data, statistical models can be the best bet. Selecting the right model could mean a huge difference in terms of efficiency and accuracy of the results.

D. OPTIMIZATION

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function ***f(x)*** by altering ***x***. We usually phrase most optimization problems in terms of minimizing ***f(x)***. Maximization may be accomplished via a minimization algorithm by minimizing ***f(x)***.

The function we want to minimize or maximize is called the objective function or criterion. When we are minimizing it, we may also call it the cost function, loss function or error function.

The choice of the objective function has an impact on the speed of learning and the overall accuracy. Each learning task requires an adapted objective function. In practice, one should benchmark results obtained from different objective functions and fine-tune the hyper-parameters that best suit the task at hand.

E. TYPES OF LEARNING

The types of learning algorithms differ in their approach, the type of data they input and output, and the type of task or problem that they are intended to solve. The following are different types of machine learning algorithms-

1) **Supervised learning**

   Supervised learning is when the model is getting trained on a labelled dataset. Labelled dataset is one which has both input and output parameters.

2) **Unsupervised learning**

   Unsupervised learning is used against data without any historical labels. The system is not given a predetermined set of outputs or correlations between inputs and outputs or a “correct answer.” The algorithm must figure out what it is seeing by itself, it has no storage of reference points. The goal is to explore the data and find some sort of patterns of structure.

3) **Semi-supervised learning**

   Semi-supervised learning falls somewhere in the middle of supervised and unsupervised learning. It is used because many problems that AI is used to solving require a balance of both approaches.

   In many cases the reference data needed for solving the problem is available, but it is either incomplete or somehow inaccurate. This is when semi-supervised learning is summoned for help since it is able to access the available reference data and then use unsupervised learning techniques to do its best to fill the gaps.

4) **Reinforcement learning**

   Reinforcement learning is a type of dynamic programming that trains algorithms using a system of reward and punishment.

   A reinforcement learning algorithm, or agent, learns by interacting with its environment. It receives rewards by performing correctly and penalties for doing so incorrectly. Therefore, it learns without having to be directly taught by a human – it learns by seeking the greatest reward and minimising penalty. This learning is tied to a context because what may lead to maximum reward in one situation may be directly associated with a penalty in another.

5) Self learning

Self-learning as a machine learning paradigm was introduced in 1982 along with a neural network capable of self-learning named Crossbar Adaptive Array (CAA). It is a learning with no external rewards and no external teacher advice. The CAA self-learning algorithm computes, in a crossbar fashion, both decisions about actions and emotions about consequence situations. The system is driven by the interaction between cognition and emotion.

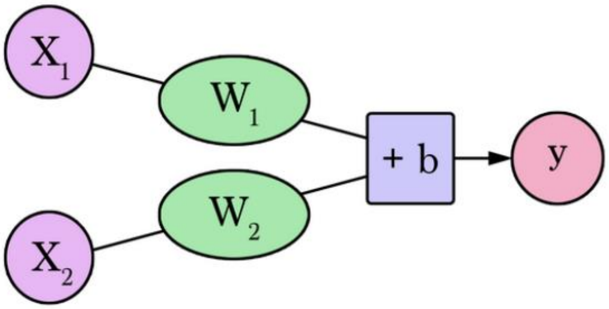
Linear Neural Network

A. Linear Regression

This is the task of predicting a *real valued target* y given a data point x. In linear regression, the simplest and still perhaps the most useful approach, we assume that prediction can be expressed as a *linear* combination of the input features.[1]

$\hat{y} = w_1 \cdot x_1 + \ldots + w_d \cdot x_d + b$ or

matrix-vector notation: $\hat{y} = X\mathbf{w}^T + b$

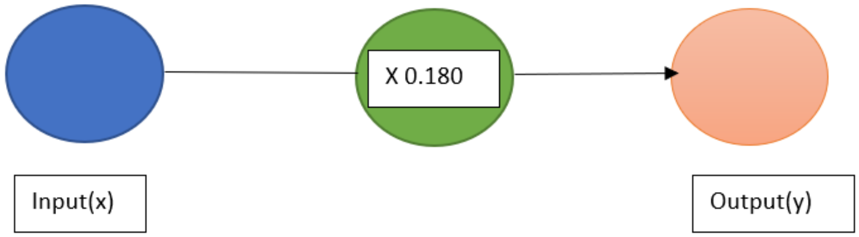


source: J. Alammar. https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/

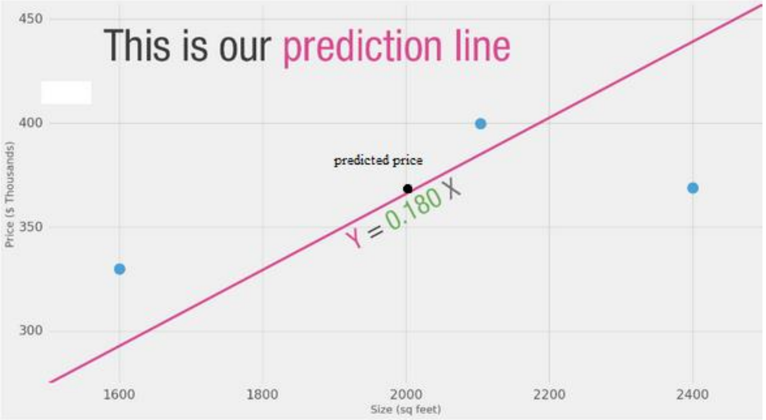
Example 1: Consider receiving a quote of $400,000 for a 2000 sq ft house (185 meters). Is this a good price considering our historical pricing?

Area (sq ft) (x)	Price(y)
2,104	399,900
1,600	329,900
2,400	369,900

$$y=Wx$$

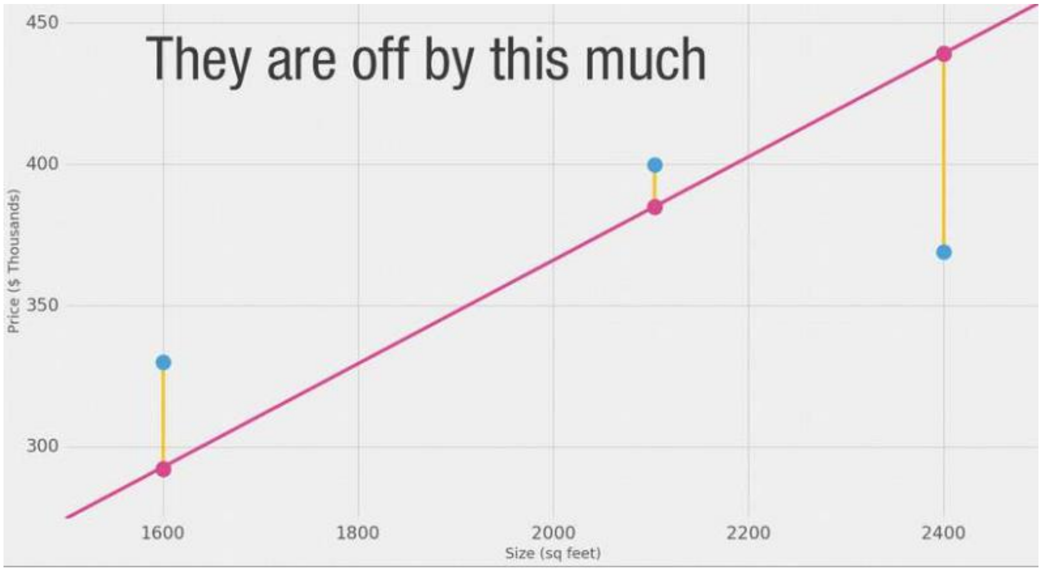


1. *Let's plot the data points. The black dots is the predicted price which is clearly less than $400,000 for a 2000 sq ft.*

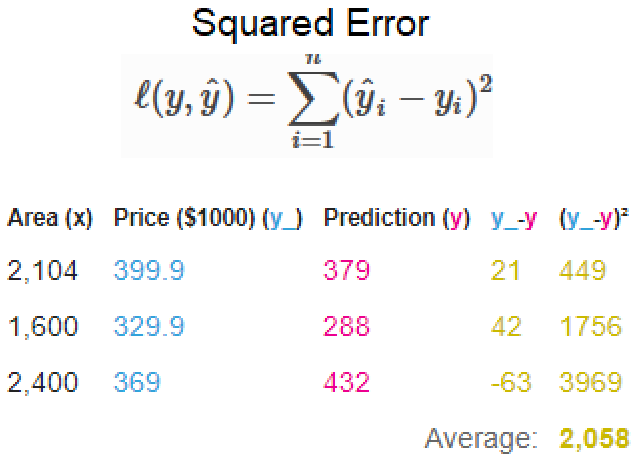


2. *Loss function.*

In order to say whether we've done a good job, we need some way to measure the quality of a model. Generally, we will define a loss function that says how far are our predictions from the correct answers.



source: J. Alammar. https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/



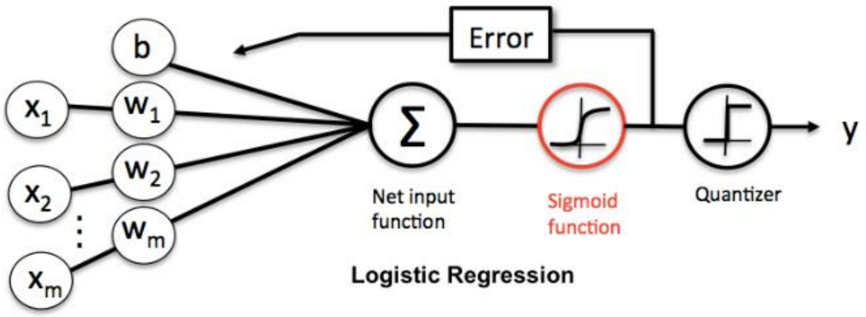
Squared Error

$$\ell(y,\hat{y}) = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

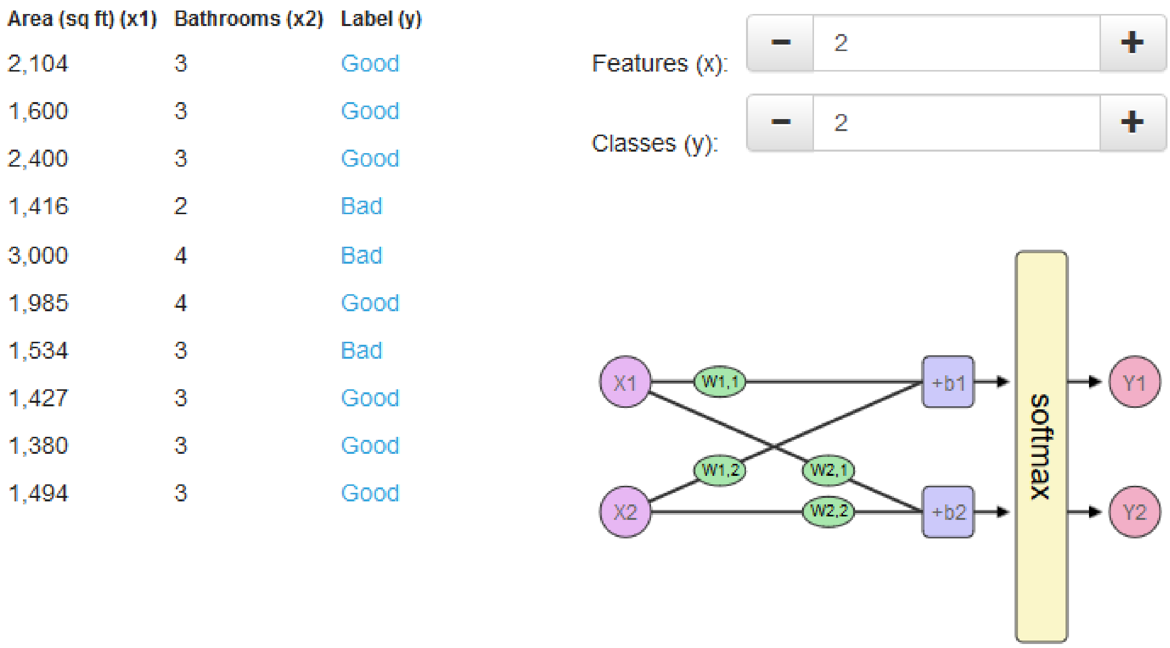
Area (x)	Price ($1000) (y_)	Prediction (y)	y_-y	(y_-y)²
2,104	399.9	379	21	449
1,600	329.9	288	42	1756
2,400	369	432	-63	3969
			Average:	**2,058**

B. **Softmax Regression**

In contrast to Linear Regression that predicts a real value y for a given data point x, Softmax Regression or Logistic Regression predicts the probability for an event or data point to belong to a category.



Example 2: Consider receiving a list of houses with features Area and number of bathrooms. How do we classify one house as Good or Bad based on our historical labeled data?



Area (sq ft) (x1)	Bathrooms (x2)	Label (y)
2,104	3	Good
1,600	3	Good
2,400	3	Good
1,416	2	Bad
3,000	4	Bad
1,985	4	Good
1,534	3	Bad
1,427	3	Good
1,380	3	Good
1,494	3	Good

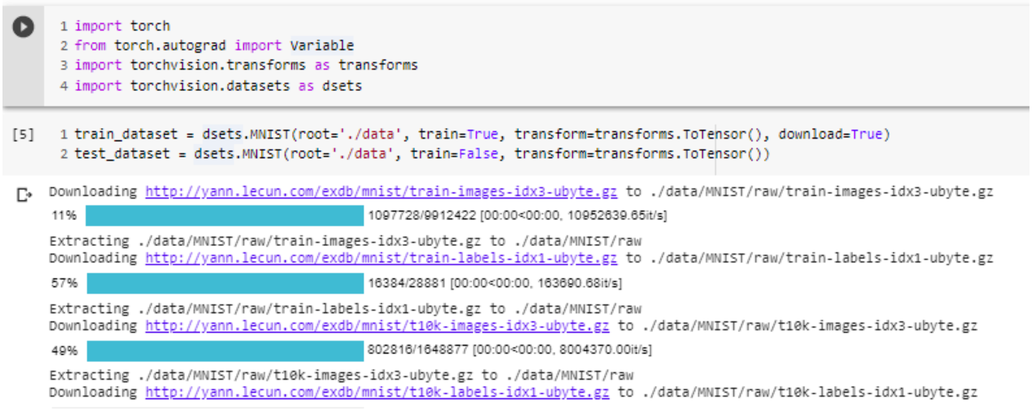
Features (x): − 2 +

Classes (y): − 2 +

C. Implementation

Logistic Regression on MNIST with PyTorch

1. Load Mnist Dataset

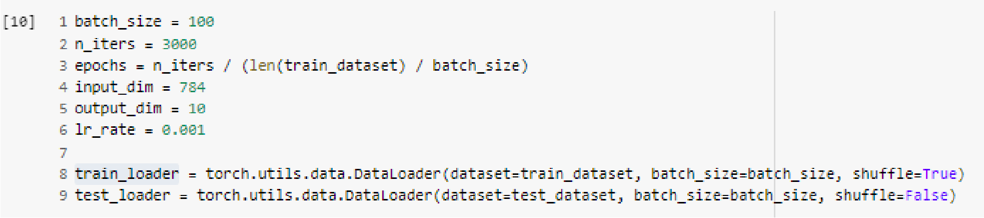


```
import torch
from torch.autograd import Variable
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

```
[5]  train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)
     test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.ToTensor())
```

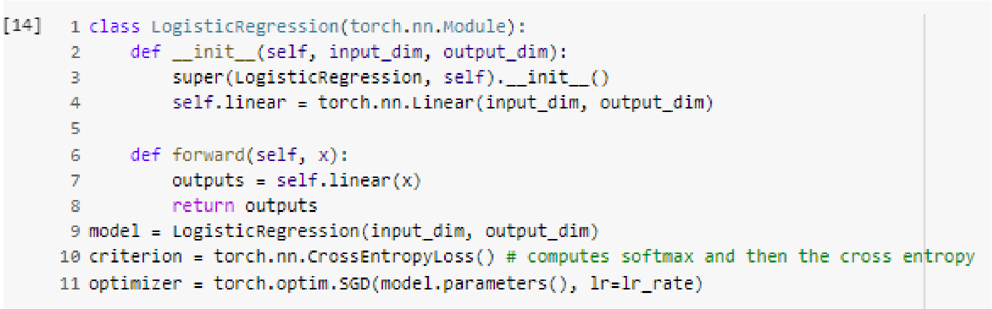
```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
11%  1097728/9912422 [00:00<00:00, 10952639.65it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
57%  16384/28881 [00:00<00:00, 163690.68it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
49%  802816/1648877 [00:00<00:00, 8004370.00it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

2. Data Split



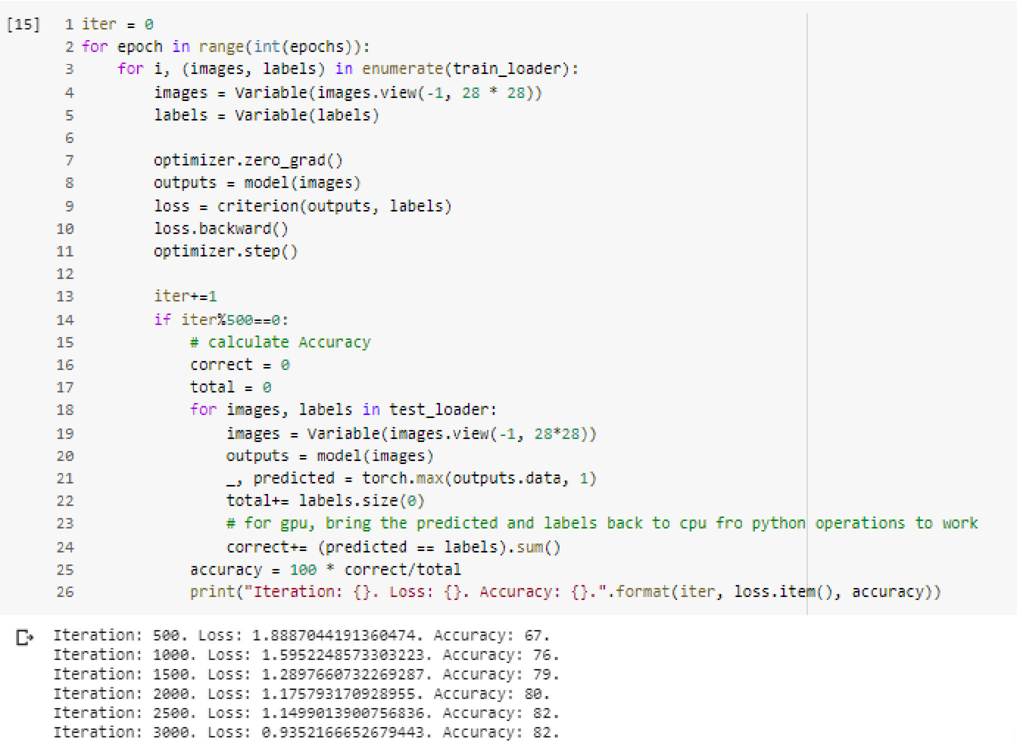
```python
batch_size = 100
n_iters = 3000
epochs = n_iters / (len(train_dataset) / batch_size)
input_dim = 784
output_dim = 10
lr_rate = 0.001

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

3. Model



```python
class LogisticRegression(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LogisticRegression, self).__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        outputs = self.linear(x)
        return outputs
model = LogisticRegression(input_dim, output_dim)
criterion = torch.nn.CrossEntropyLoss() # computes softmax and then the cross entropy
optimizer = torch.optim.SGD(model.parameters(), lr=lr_rate)
```

4. Training and Evaluation



```python
iter = 0
for epoch in range(int(epochs)):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.view(-1, 28 * 28))
        labels = Variable(labels)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        iter+=1
        if iter%500==0:
            # calculate Accuracy
            correct = 0
            total = 0
            for images, labels in test_loader:
                images = Variable(images.view(-1, 28*28))
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total+= labels.size(0)
                # for gpu, bring the predicted and labels back to cpu fro python operations to work
                correct+= (predicted == labels).sum()
            accuracy = 100 * correct/total
            print("Iteration: {}. Loss: {}. Accuracy: {}.".format(iter, loss.item(), accuracy))
```

```
Iteration: 500. Loss: 1.8887044191360474. Accuracy: 67.
Iteration: 1000. Loss: 1.5952248573303223. Accuracy: 76.
Iteration: 1500. Loss: 1.2897660732269287. Accuracy: 79.
Iteration: 2000. Loss: 1.175793170928955. Accuracy: 80.
Iteration: 2500. Loss: 1.1499013900756836. Accuracy: 82.
Iteration: 3000. Loss: 0.9352166652679443. Accuracy: 82.
```

Multi-Layer Perceptron

A. Underfitting, Overfitting

A model or algorithm is said to have **underfitting** when it cannot capture the underlying trend of the data. It occurs when the model or algorithm does not fit the data enough. Underfitting occurs if the model or algorithm shows low variance but high bias. It is often a result of an excessively simple model.

A model or algorithm is said to be **overfitted**, when we train it with a lot of data. It occurs when the model or algorithm contains more parameters than can be justified by the data. Overfitting occurs if the model or algorithm shows high variance but low bias

B. Weight decay, Dropout

Weight decay is used to prevent overfitting. Having fewer parameters is only one way of preventing our model from getting overly complex. But it is actually a very limiting strategy. More parameters mean more interactions between various parts of our neural network. Thus to use more parameters without making it overly complex we use weight decay. It is added to the loss function and acts as an additional term in the weight update rule that causes the weights to exponentially decay to zero.

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.

C. Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network in order from the input layer to the output layer. The input *X* provides the initial information that then propagates to the hidden units at each layer and finally produces the output *y^*. The architecture of the network entails determining its depth, width, and activation functions used on each layer. Depth is the number of hidden layers. Width is the number of units on each hidden layer since we don’t control neither input layer nor output layer dimensions.

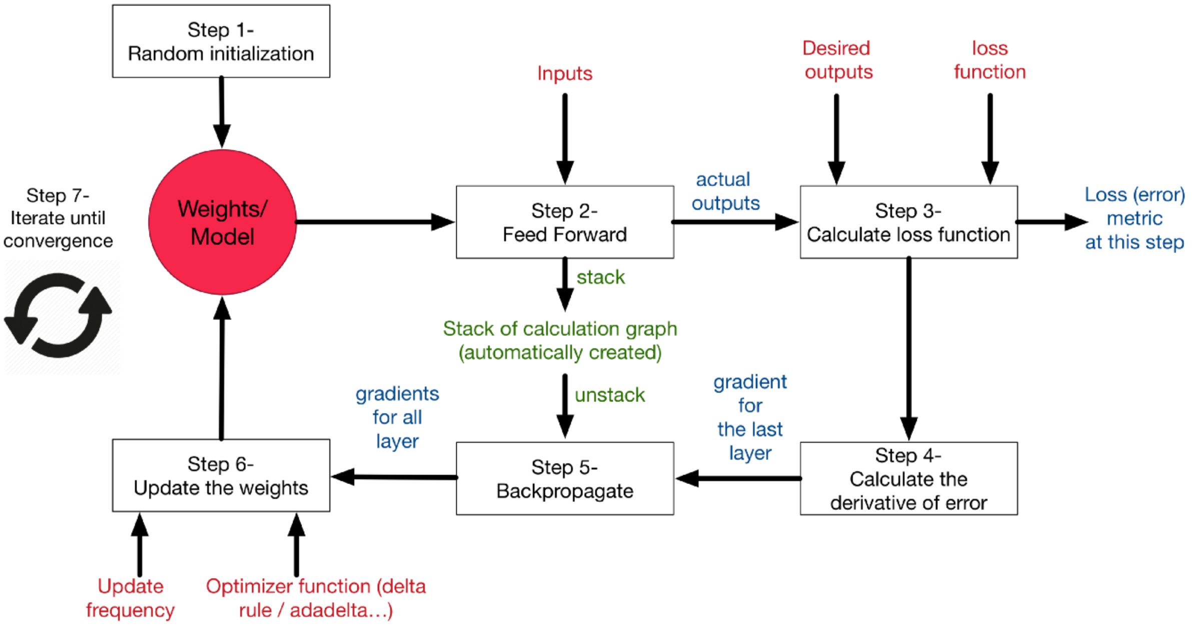
D. Backward Propagation

Backward propagation is the practice of fine tuning the weights of a neural net based on the error rate (i.e. loss) obtained in the previous epoch. This process is essential in making the model reliable by increasing its generalization. Once we calculate the loss after an epoch, based on it, we can fine tune the weights with the help of an optimization function such as Gradient Descent which helps us find the weights that will hopefully yield a smaller loss.

E. Initialization techniques and Numerical Stability

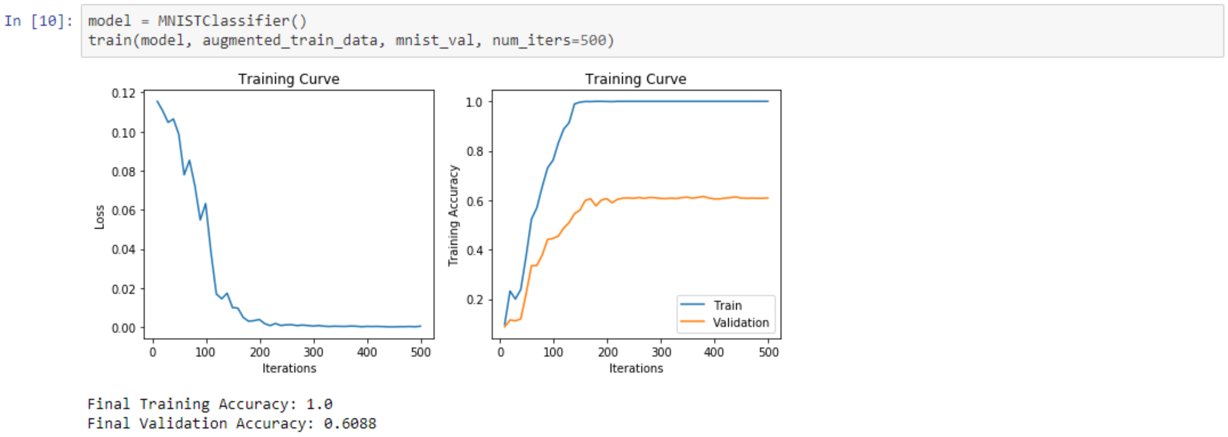
Performance of the model depends highly on the type of activation functions we use and the initialization strategy we undertake. Using activation functions like sigmoid function can lead to what's called **vanishing gradient problem** where the gradient is close to zero when the inputs are either too large or small. There's also a chance where the matrix product can yield very high values which is called the **exploding gradient problem**. These can be avoided by choosing the right activation functions (like RELU) and proper weight initialization by selecting the right distribution.

The working of a multi-layer perceptron can be encapsulated by the following flowchart.



F. Implementation (complete code in appendix)

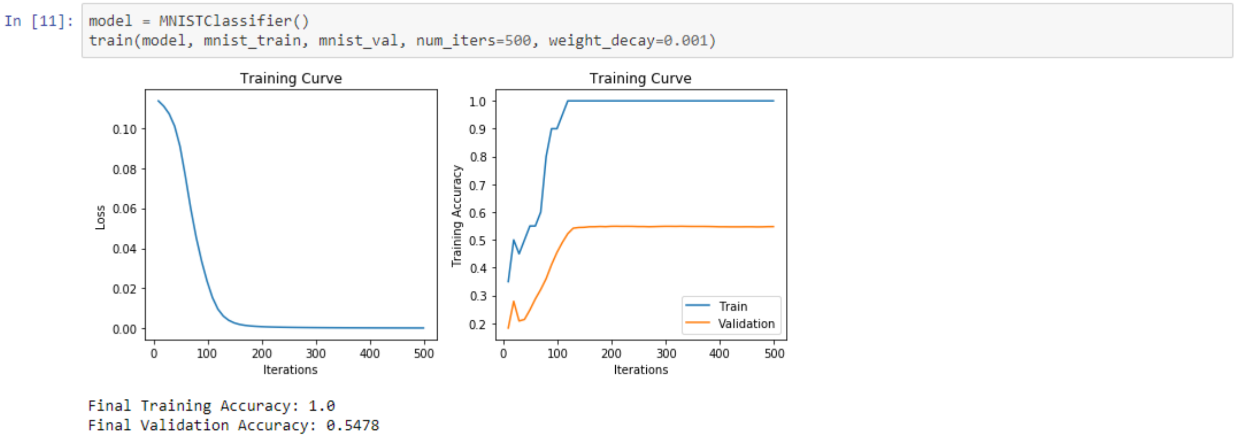
#Overfit model



```
In [10]: model = MNISTClassifier()
         train(model, augmented_train_data, mnist_val, num_iters=500)
```

```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.6088
```

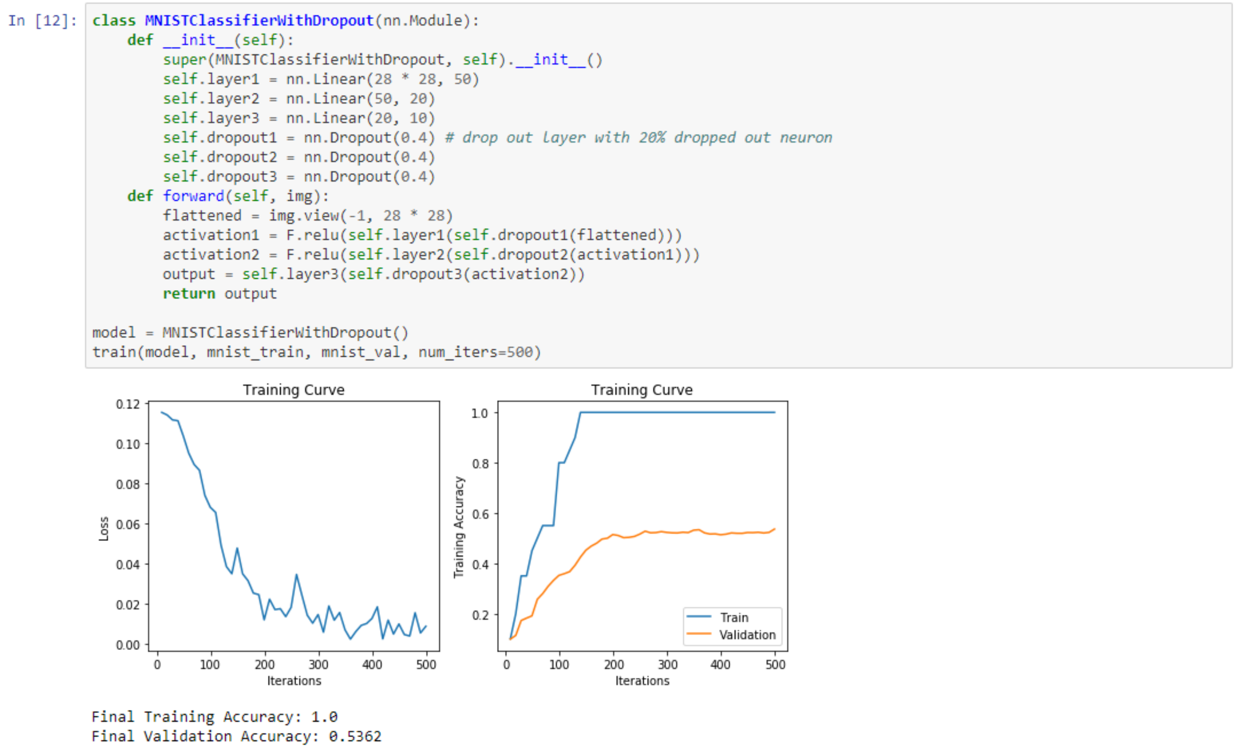
#Weight decay



```
In [11]: model = MNISTClassifier()
         train(model, mnist_train, mnist_val, num_iters=500, weight_decay=0.001)
```

```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.5478
```

#Dropout



```
In [12]: class MNISTClassifierWithDropout(nn.Module):
             def __init__(self):
                 super(MNISTClassifierWithDropout, self).__init__()
                 self.layer1 = nn.Linear(28 * 28, 50)
                 self.layer2 = nn.Linear(50, 20)
                 self.layer3 = nn.Linear(20, 10)
                 self.dropout1 = nn.Dropout(0.4) # drop out layer with 20% dropped out neuron
                 self.dropout2 = nn.Dropout(0.4)
                 self.dropout3 = nn.Dropout(0.4)
             def forward(self, img):
                 flattened = img.view(-1, 28 * 28)
                 activation1 = F.relu(self.layer1(self.dropout1(flattened)))
                 activation2 = F.relu(self.layer2(self.dropout2(activation1)))
                 output = self.layer3(self.dropout3(activation2))
                 return output

         model = MNISTClassifierWithDropout()
         train(model, mnist_train, mnist_val, num_iters=500)
```

```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.5362
```

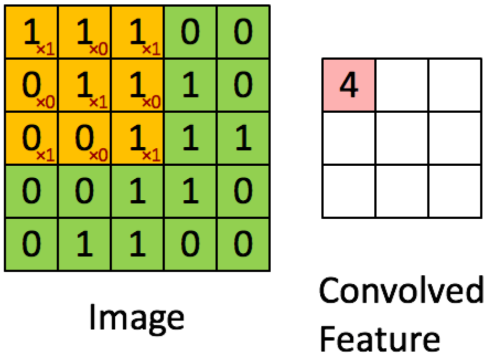
Convolutional Neural Networks

Regular neural networks scale poorly when the data has a two-dimensional structure such as an image or a speech signal. For example, for images of size 32 x 32 x 3, the first hidden layer alone will have 3072 weights. This can increase substantially for a fully connected structure and for larger images. Convolutional Neural Networks (CNNs) are designed in such a way to take advantage of the 2D structure of input data while using considerably fewer parameters than fully connected networks.

The general idea behind CNNs is to start looking for some low-level features such as edges and curves in the case of image data and then building up to more abstract concepts through a series of convolutional layers.

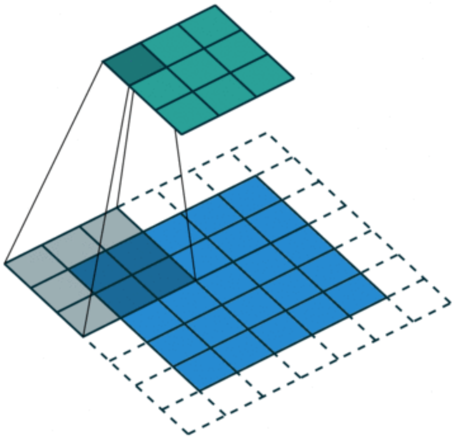
A. Convolutional layer

The convolutional layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. The feature map is obtained by running the filters over the image whose values are made non linear by using an activation function like ReLU.

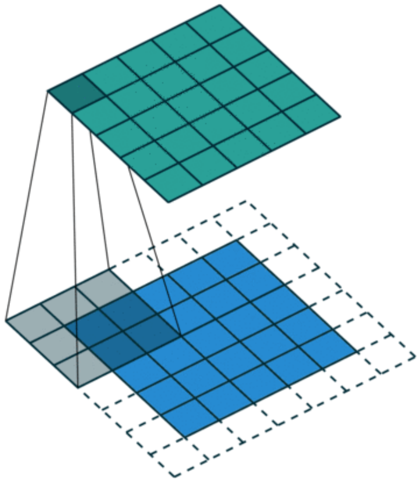


B. Padding and Stride

After convolution, the feature is reduced in dimensionality as compared to the input. Depending on the type of images we are dealing with, we may want to secure or increase the dimensionality of the image or considerably reduce it. We can preserve or increase the dimensionality of the image by padding or reduce drastically using stride. **Valid Padding** (left of the below image)is where no padding is added and the dimensionality is reduced after convolution whereas **Same Padding** (right of the below image) is where extra filler cells are added around the boundary of the feature matrix and depending on the size of the padding used, dimensionality is either increased or remains the same.



Valid Padding

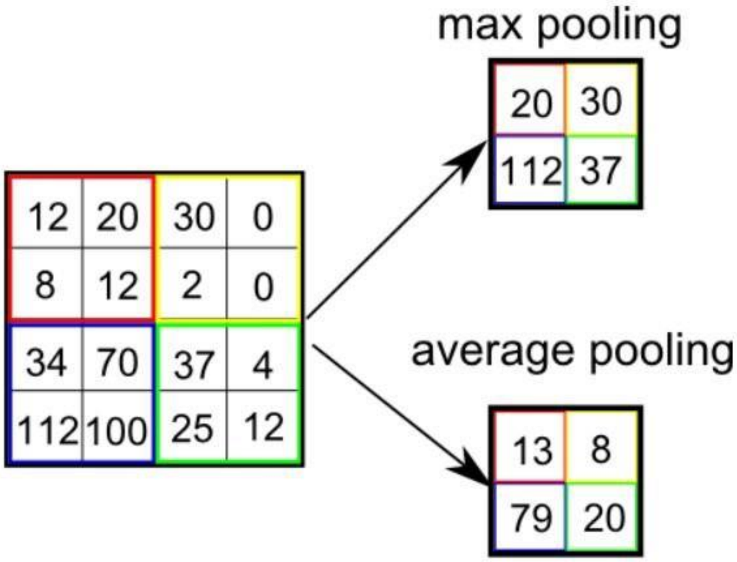


Same Padding

In cases where we need to reduce the dimensionality of the feature matrix either for computational efficiency or downsampling, we can use stride. By default the kernel is slid over the image one pixel at a time. By introducing a stride, which says by how many pixels to move at a time, we can go through the feature matrix quickly.

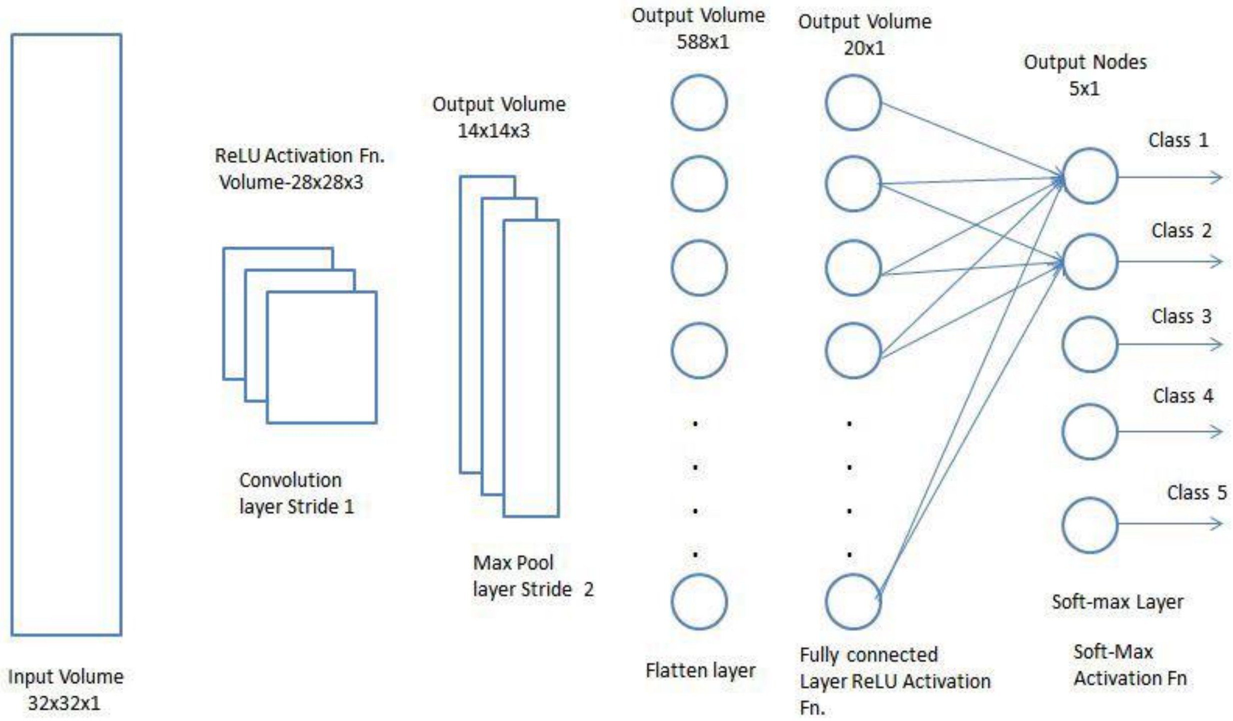
C. Pooling

Another way to decrease the computational power required to process the data through dimensionality reduction is pooling. It is useful for extracting dominant features which are rotational and positional invariant. There are two types of pooling: **Max Pooling** returns the maximum value of the portion of the image covered by the kernel and **Average Pooling** returns the average of all the values from the portion of the image covered by the kernel.



D. Fully connected layer

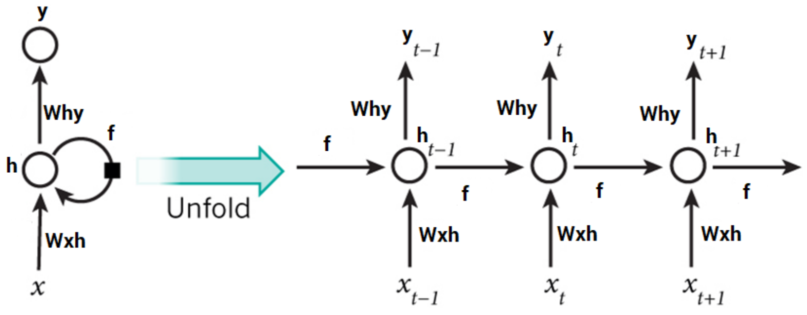
A fully connected layer is a useful way of learning non-linear combinations of high- level features as represented by the output of the convolutional layer. The output from the convolutional layers are flattened and fed to a multi-layer perceptron and helps in the classification process. Below is a sample architecture of a CNN network.



Recurrent Neural Network

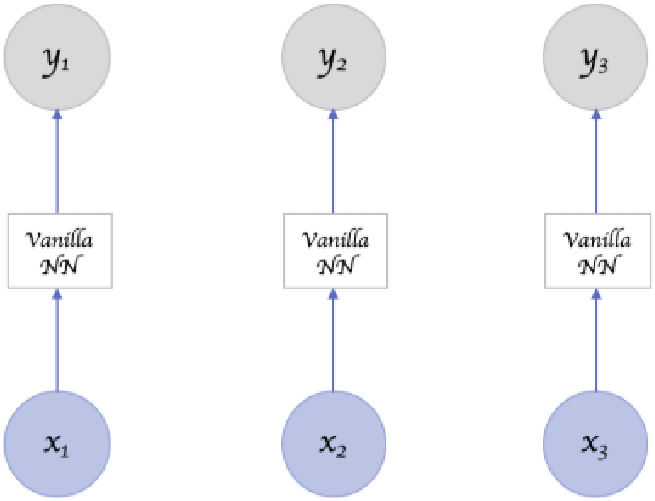
A. Motivation

Recurrent Neural Networks or RNNs are a special type of neural network designed for sequence problems. Given a standard feed-forward multilayer Perceptron network, a recurrent neural network can be thought of as the addition of loops to the architecture.

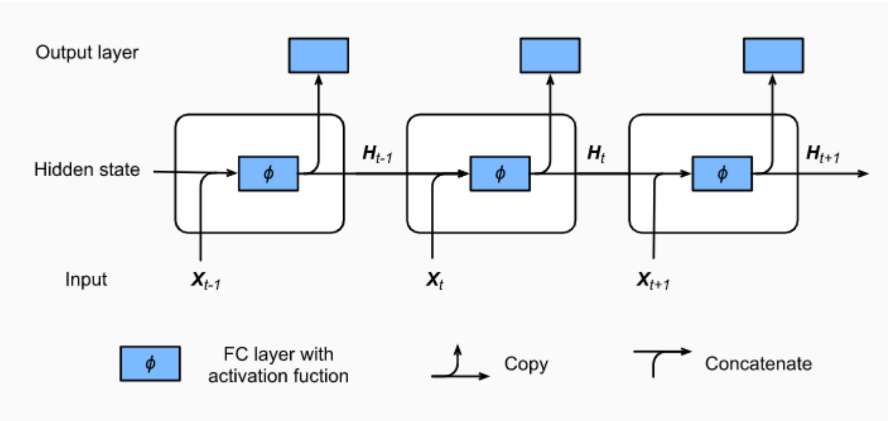


B. Design

1. Intuitively, someone can think of a series of Vanilla networks (NN) to deal with the sequential problem.But, the repeated Vanilla networks as you can see in fig below, does not capture relationships across inputs meaningfully. That's where the Recurrent neural network architecture has come to play a crucial role in the problem of sequential data.



2. The recurrent neural network architecture can be seen as a standard feed-forward multi-layer perceptron where we add loops in the architecture.Now, to control the amount of information to be remembered or forgotten,to learn broader abstractions from the input sequences, we add the concept of states.



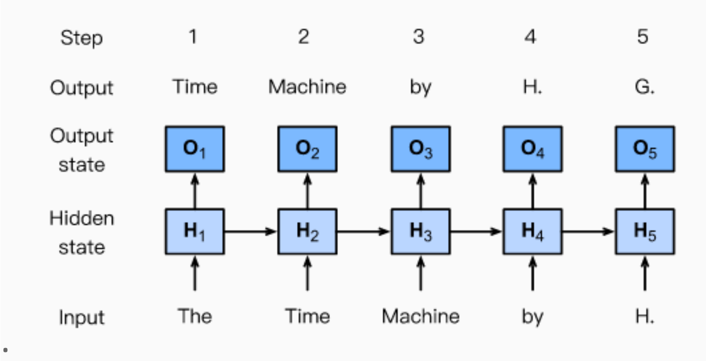
Input:$X_t \in R^{nxd} X_t \in R^{nxd}, t = 1, \dots, T$.X(t) is taken as the input to the network at time step t. For example, x1,could be a one-hot vector corresponding to a word of a sentence.

Hidden state: h(t) represents a hidden state at time t and acts as “memory” of the network. h(t) is calculated based on the current input and the previous time step’s hidden state:$\boldsymbol{H}_t = \boldsymbol{\phi}(\boldsymbol{X}_t \boldsymbol{W}_{xh} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hh} + \boldsymbol{b}_h)$. The function f is taken to be a non-linear transformation such as tanh, ReLU.

Weights: $W_{hh} \in R^{hxh}$.The RNN has input to hidden connections parameterized by a weight matrix U, hidden-to-hidden recurrent connections parameterized by a weight matrix W, and hidden-to-output connections parameterized by a weight matrix V and all these weights (U,V,W) are shared across time.

Output: O(t) The output of the fully connected layer is the hidden state of the current timestep Ht . Its model parameter is the concatenation of Wxh and Whh , with a bias of bh . The hidden state of the current timestep t , Ht , will participate in computing the hidden state Ht+1 of the next timestep t+1 . What is more, Ht will become the input for Ot , the fully connected output layer of the current timestep:$O_t = H_t W_{hq} + b_q$.

Word-level RNN language model. The input and label sequences are the time machine by H. and the time machine by H. G. respectively.



.

3. Train Recurrent Neural Networks

   a) The RNN forward pass can thus be represented by below set of equations:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

b) *Backpropagation through time (BPTT)* is actually a specific application of back propagation in recurrent neural networks.It requires us to expand the recurrent neural network one time step at a time to obtain the dependencies between model variables and parameters. Then, based on the chain rule, we apply backpropagation to compute and store gradients

In this simplified model above, we denote ht as the hidden state, xt as the input, and ot as the output at timestep t . In addition, wh and wo indicate the weights of hidden states and the output layer, respectively. As a result, the hidden states and outputs at each timesteps can be explained as:

$$h_t = f(x_t, h_{t-1}, w_h) \ and \ o_t = g(h_t, w_o)$$

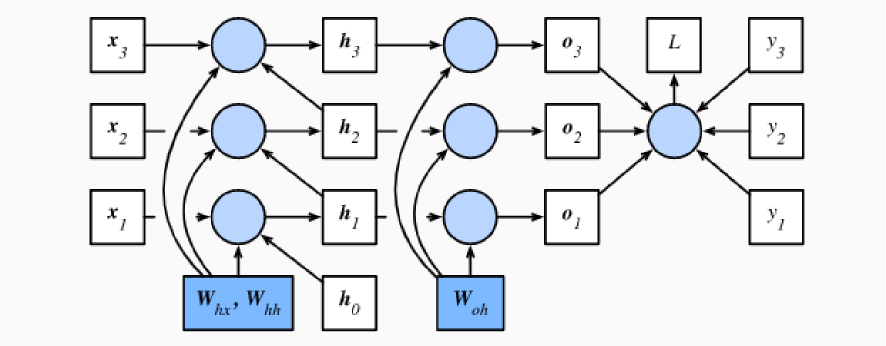
The discrepancy between outputs ot and the desired targets yt is then evaluated by an objective function as:

$$L(x, y, w_h, w_o) = \sum_{t=1}^{T} \quad l(y_t, o_t)$$

$$\partial \quad _{w_h} L = \sum_{t=1}^{T} \quad \partial \quad _{w_h} l(y_t, o_t)$$

$$= \sum_{t=1}^{T} \quad \partial \quad _{o_t} l(y_t, o_t) \partial \quad _{h_t} g(h_t, w_h)[\partial \quad _{w_h} h \quad _t].$$

The Computational Graph below illustrates the BPTT (L is the loss)



C. **Perplexity Measure**

1. In general, perplexity is a measurement of how well a probability model predicts a sample. In the context of Natural Language Processing, perplexity is one way to evaluate language models.

2. One way is to check how surprising the text is. A good language model is able to predict with high accuracy tokens which we will see next. Consider the following continuations of the phrase "It is raining", as proposed by different language models:

   a) "It is raining outside"

   b) "It is raining banana tree"

   c) "It is raining piouw;kcj pwepoiut"

   In terms of quality, example a) is clearly the best,example b) is considerably worse by producing a nonsensical extension, and last, example c) indicates a poorly trained model that does not fit data properly.

$$PPL = exp\,(-\frac{1}{n}\sum_{t=1}^{n} \quad logp(x_t|x_{t-1},\ldots,x_1 \quad)$$

3. Interpretation of the Perplexity value:

   (1) In general,higher is the Perplexity value and worse is the model.

   (2) In the best case scenario, the model always estimates the probability of the next symbol as 1 In this case the perplexity of the model is 1

(3) In the worst case scenario, the model always predicts the probability of the label category as 0. In this situation, the perplexity is infinite.

(4) At the baseline, the model predicts a uniform distribution over all tokens. In this case, the perplexity equals the size of the dictionary len(vocab)

CNN Code for Appendix

```
# Standard library
import random
import numpy as np

#importing MNIST dataset
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('/home/server/datasets/MNIST_data/')
mnist_data = list(mnist)
```

```
mnist_train = mnist_data[:20]    #  20 train images
mnist_val   = mnist_data[100:5100]     # 2000 validation images


class MNISTClassifier(object):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 10)
    #activation function ReLu with feed forward
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output
def train(model, train, valid, batch_size=20, num_iters=1, learn_rate=0.01, weight_decay=0):
    train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size,shuffle=True)
# shuffle after every epoch
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learn_rate, momentum=0.9,
weight_decay=weight_decay)


     iters, losses, train_acc, val_acc = [], [], [], []
```

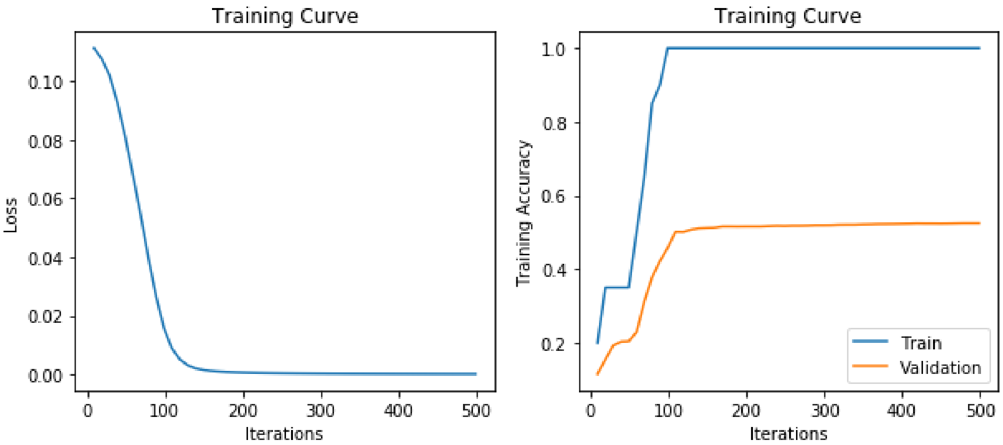
```
# training
n = 0  # the number of iterations
 while True:
     if n >= num_iters:
        break
    for imgs, labels in iter(train_loader):
        model.train()
        out = model(imgs)            # forward pass
        loss = criterion(out, labels) # computing the total loss
        loss.backward()       # backward pass
        optimizer.step()            # make the updates for each parameter
        optimizer.zero_grad()       # a clean up step for PyTorch
    # saving the current training information
         if n % 10 == 9:
            iters.append(n)
            losses.append(float(loss)/batch_size)
        # computing *average* loss
             train_acc.append(get_accuracy(model, train))
        # computing training accuracy
             val_acc.append(get_accuracy(model, valid))
        # computing validation accuracy
            n += 1
    # plotting
```

```
        plt.figure(figsize=(10,4))
        plt.subplot(1,2,1)
        plt.title("Training Curve")
        plt.plot(iters, losses, label="Train")
        plt.xlabel("Iterations")
        plt.ylabel("Loss")

        plt.subplot(1,2,2)
        plt.title("Training Curve")
        plt.plot(iters, train_acc, label="Train")
        plt.plot(iters, val_acc, label="Validation")
        plt.xlabel("Iterations")
        plt.ylabel("Training Accuracy")
        plt.legend(loc='best')
        plt.show()

        print("Final Training Accuracy: {}".format(train_acc[-1]))
        print("Final Validation Accuracy: {}".format(val_acc[-1]))
train_acc_loader = torch.utils.data.DataLoader(mnist_train, batch_size=100)
val_acc_loader = torch.utils.data.DataLoader(mnist_val, batch_size=1000)


def get_accuracy(model, data):
    correct = 0
```

```
total = 0

model.eval()

for imgs, labels in torch.utils.data.DataLoader(data, batch_size=64):

output = model(imgs)

pred = output.max(1, keepdim=True)[1] # get the index of the max logit

correct += pred.eq(labels.view_as(pred)).sum().item()

total += imgs.shape[0]

return correct / total
```

Running the code:

```
model = MNISTClassifier()

train(model, mnist_train, mnist_val, num_iters=500)
```



```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.5246
```

RNN Code for Appendix

```
#Getting data set from tensor flow

mnist = input_data.read_data_sets("MNIST_data/")
```

We reshape the data set from based on the parameters we set. we do so TensorFlow assumes that you are using one-hot encoding which we are not doing.

```
# hyperparameters

n_neurons = 128

learning_rate = 0.001

batch_size = 128

n_epochs = 10


# parameters

n_steps = 28 # 28 rows

n_inputs = 28 # 28 cols

n_outputs = 10 # 10 classes


# building RNN model

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

y = tf.placeholder(tf.int32, [None])


cell = tf.nn.rnn_cell.BasicRNNCell(num_units=n_neurons)

output, state = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

```
logits = tf.layers.dense(state, n_outputs)

cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)


loss = tf.reduce_mean(cross_entropy)

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)p


prediction = tf.nn.in_top_k(logits, y, 1)

accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32))
```

Reshaping dataset to [num_test, n_steps, n_inputs]

```
X_test = mnist.test.images # X_test shape: [num_test, 28*28]

X_test = X_test.reshape([-1, n_steps, n_inputs])

y_test = mnist.test.labels
```

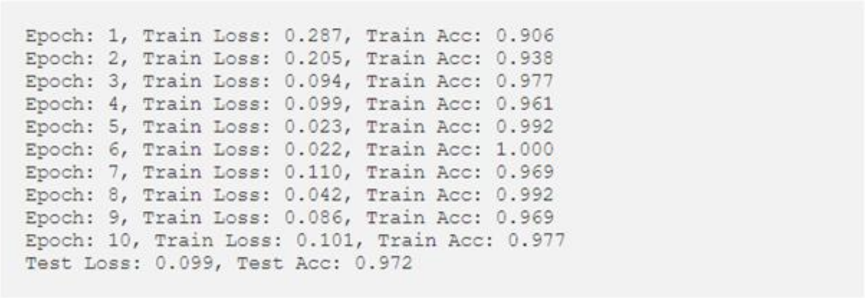
Training the model

```
# initialize the variables

init = tf.global_variables_initializer()
```

```
# training model

with tf.Session() as sess:

    sess.run(init)

    n_batches = mnist.train.num_examples // batch_size

    for epoch in range(n_epochs):

        for batch in range(n_batches):

            X_train, y_train = mnist.train.next_batch(batch_size)

            X_train = X_train.reshape([-1, n_steps, n_inputs])

            sess.run(optimizer, feed_dict={X: X_train, y: y_train})

    loss_train, acc_train = sess.run([loss, accuracy], feed_dict={X: X_train, y: y_train})

    print('Epoch: {}, Train Loss: {:.3f}, Train Acc: {:.3f}'.format(epoch + 1, loss_train, acc_train))

    loss_test, acc_test = sess.run([loss, accuracy], feed_dict={X: X_test, y: y_test})

    print('Test Loss: {:.3f}, Test Acc: {:.3f}'.format(loss_test, acc_test))
```

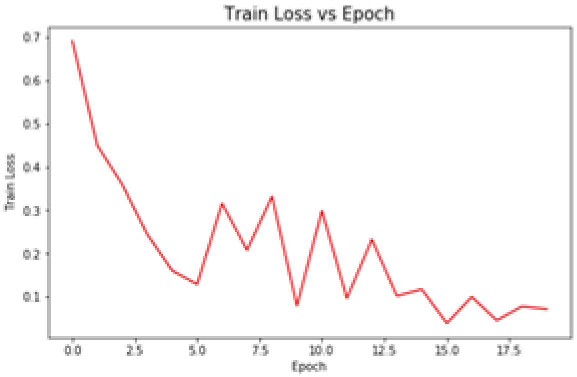
Output-

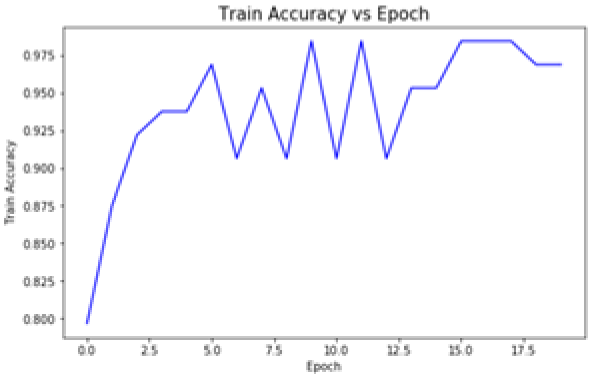
Test Loss and Test Accuracy



```
Epoch: 1, Train Loss: 0.287, Train Acc: 0.906
Epoch: 2, Train Loss: 0.205, Train Acc: 0.938
Epoch: 3, Train Loss: 0.094, Train Acc: 0.977
Epoch: 4, Train Loss: 0.099, Train Acc: 0.961
Epoch: 5, Train Loss: 0.023, Train Acc: 0.992
Epoch: 6, Train Loss: 0.022, Train Acc: 1.000
Epoch: 7, Train Loss: 0.110, Train Acc: 0.969
Epoch: 8, Train Loss: 0.042, Train Acc: 0.992
Epoch: 9, Train Loss: 0.086, Train Acc: 0.969
Epoch: 10, Train Loss: 0.101, Train Acc: 0.977
Test Loss: 0.099, Test Acc: 0.972
```

Train Loss Vs Epoch and Train Accuracy vs Epoch





Visualizing prediction made by RNN model

predict: 7

predict: 2

predict: 1

Prediction