**Pytorch**

* [Understanding PyTorch with an example: a step-by-step tutorial | by Daniel Godoy | Towards Data Science](https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e)

**Creating tensors from numpy arrays**

torch.from\_numpy(arr)

There is however one drawback with this. The array and the tensor are still linked and if we modify an element in the array it will be reflected in the tensor too. In order to overcome this:

torch.tensor(arr)

* **torch.tensor vs torch.Tensor :** The former retains the datatype of the array elements whereas the latter converts them to floats.

**Creating tensors from scratch**

* Empty tensors of desired shape

torch.empty(4,2)

* Tensors of zeroes or ones:

torch.zeros(4,3,dtype=torch.int32)

torch.ones(4,3)

* Values within a given range:

torch.arange(0,18,2).reshape(3,3)

torch.linspace(0,50,20).reshape(5,4)

* Random values
* Random values between 0 and 1

torch.rand(4,3)

* Values from normal distribution

torch.randn(4,3)

* Random integer values

torch.randint(low=0,high=10,size=(4,4))

* Using existing tensors as reference:

x= torch.zeros(2,5)

torch.rand\_like(x)

torch.randn\_like(x)

torch.randint\_like(x,low=0,high=25)

* Setting seed to obtain uniform values:

torch.manual\_seed(42)

torch.rand(2,3)

**Implementation of gradients in pytorch**

**Backpropagation in a single step**

1. Create a tensor with **requires\_grad = True**

x = torch.tensor(2.0,requires\_grad=True)

1. Define a function

y = 2\*x\*\*4 + x\*\*3 + 3\*x\*\*2 + 5\*x + 1

Since y was created as a result of an operation on x, y has an associated gradient function accesible as **y.grad\_fn** (first derivative of the function)

1. Backpropagation

y.backward()

1. Displaying the resulting gradient

x.grad

This is basically y’(x) , value of the first derivative of y at x.

**Backpropagation on multiple steps**

1. Create a tensor

x = torch.tensor([[1.,2,3],[3,2,1]], requires\_grad=True)

print(x)

1. Create a first layer( a function which directly operates upon this tensor)

y = 3\*x + 2

1. Create the second layer ( function that uses the intermediate result)

z = 2\*y\*\*2

1. Set the output to be the matrix mean( this value will be easier to propagate compared to the entire tensor)

out = z.mean()

1. Now perform back-propagation to find the gradient of z with respect to x:

out.backward()

Basically using chain rule first the gradient of z with respect to y is obtained and then wrt x.

The main idea is that x, y, z are part of a dynamic computational graph which facilitates this operation. to turn off this tracking we must set **requires\_grad = False.**

**Linear Regression using pytorch**

**Creating a column matrix of X values**

X = torch.linspace(1,50,50).reshape(-1,1)

reshape(-1,1) helps us to obtain a column vector

**Creating a random array of error values**

torch.manual\_seed(71)

e = torch.randint(-8,9,(50,1),dtype=torch.float)

**Creating a column matrix of y values**

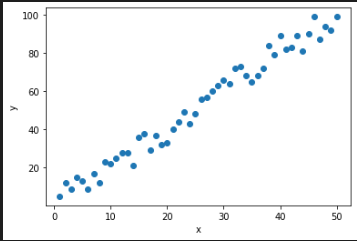
y = 2\*X + 1 + e

**Plotting the results**

plt.scatter(X.numpy(), y.numpy())

plt.ylabel('y')

plt.xlabel('x');

****

Note that we haven’t specified **requires\_grad = True**  hence backpropagation won’t work(y.backward())

**Simple linear model**

torch.manual\_seed(59)

model = nn.Linear(in\_features=1, out\_features=1)

print(model.weight)

print(model.bias)

The values that we obtain here are the ones that are preselected by the model at random. Obtaining predictions using these values will lead to wayward results.

**Setting up the model class**

class Model(nn.Module):

    def \_\_init\_\_(self,in\_features,out\_features):

        super().\_\_init\_\_()

        self.linear = nn.Linear(in\_features,out\_features)

    def forward(self,x):

        y\_pred = self.linear(x)

        return y\_pred

**nn.Linear –** fully connected dense layer

torch.manual\_seed(59)

model = Model(1, 1)

As models become more complex, it is better to iterate over all the model parameters:

for name, param in model.named\_parameters():

    print(name, '\t', param.item())

Passing a tensor to the instantiated model:

x = torch.tensor([2.0])

print(model.forward(x))

This is forward propagation. This is basically the value of the function y at the given x.

Next, we try to obtain predictions on an input vector from the model. At this point, no training has been carried out and thus the result obtained will be way off the actual result.

x1 = np.linspace(0.0,50.0,50)

w1 = 0.10597813129425049

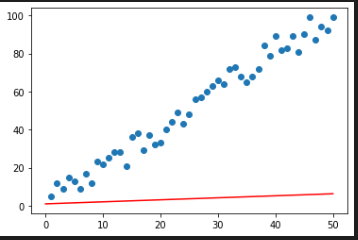
b1 = 0.9637961387634277

y1 = w1\*x1 + b1

The weight and bias are the random initilisations that were checked earlier.

plt.scatter(X.numpy(),y.numpy())

plt.plot(x1,y1,'r')



**Initializing the loss function and optimizer**

criterion = nn.MSELoss()

optimizer = torch.optim.SGD(model.parameters(),lr = 0.001)

**SGD –** stochastic gradient descent

**lr** is the learning rate. Decrease the value if the loss isn’t reducing(if it is stagnating).

**Training the model**

Following are the key steps involved in training the model:

1. Choose the number of epochs for which the training must be run
2. Initialise a list which will hold the loss value for every epoch
3. During every epoch, increment the value of the loop counter so that it can be reflected in the loss output
4. Obtain predictions from the model using the forward function( **model.forward()**)
5. Calculate the loss by comparing the predictions obtained with the actual expected values using the loss function defined earlier. (**criterion(y\_pred,y\_true)**
6. Append this loss value into the list and print out the loss for this epoch
7. Withe every backpropagation there is an accumulation of gradients . To prevent compounding of this we need clear the gradient before every new epoch **optimizer.zero\_grad()**
8. Backpropagate the loss through the network . **loss.backward()**
9. Update the hyperparameters of the network after every backprop. **optimizer.step()**

epochs = 50

losses = []

for i in range(epochs):

    i = i+1

    y\_pred = model.forward(X)

    loss = criterion(y\_pred,y)

    losses.append(loss.item())

    #Note : item() returns the value of the tensor as a python Integer

    print(f"epoch {i} loss:{loss.item()} weight:{model.linear.weight.item()} bias ={model.linear.bias.item()}")

    optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

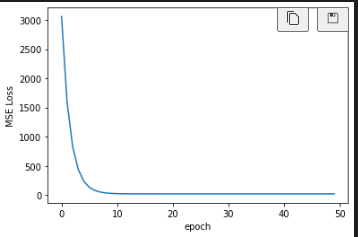
[python - Why do we need to call zero\_grad() in PyTorch? - Stack Overflow](https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch)

**Plotting epochs vs loss**

plt.plot(range(epochs),losses)

plt.ylabel('MSE Loss')

plt.xlabel('epoch')

****

**Datasets using Pytorch**

Basically dealing with the methodlogies that can be adopted while processing the inbuilt(as well as external datasets).

**Approach 1 : Using sklearn train\_test\_split**

Typically when we load our dataset, we use the train\_test\_split functionality from sklearn:

from sklearn.model\_selection import train\_test\_split

train\_X, test\_X, train\_y, test\_y = train\_test\_split(df.drop('target',axis=1).values,

                                                    df['target'].values, test\_size=0.2,

                                                    random\_state=33)

X\_train = torch.FloatTensor(train\_X)

X\_test = torch.FloatTensor(test\_X)

y\_train = torch.LongTensor(train\_y).reshape(-1, 1)

y\_test = torch.LongTensor(test\_y).reshape(-1, 1)

**Approach 2 : Using Pytorch’s Dataset and Dataloader classes**

from torch.utils.data import TensorDataset, DataLoader

data = df.drop('target',axis=1).values

labels = df['target'].values

iris = TensorDataset(torch.FloatTensor(data),torch.LongTensor(labels))

Once we have a dataset we can wrap it with a DataLoader. This gives us a powerful sampler that provides single- or multi-process iterators over the dataset**.**

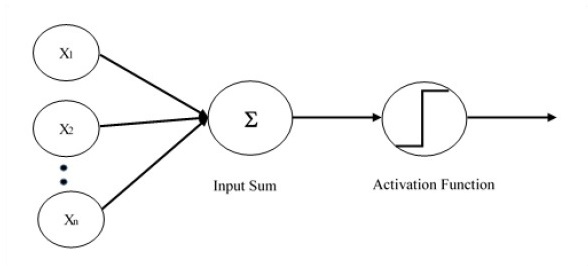
iris\_loader = DataLoader(iris, batch\_size=105, shuffle=True)

**Artificial Neural Networks**

* Thanks to the advent of deep learning, we are equipped with better methods to handle and process unstructured data

**The Neuron**

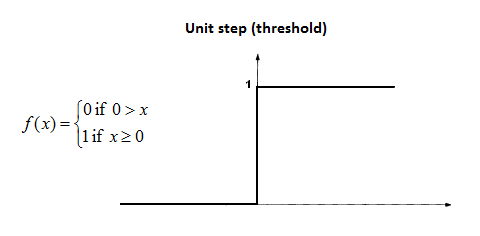
* Dendrites are receivers of signals, axons are the transmitters.



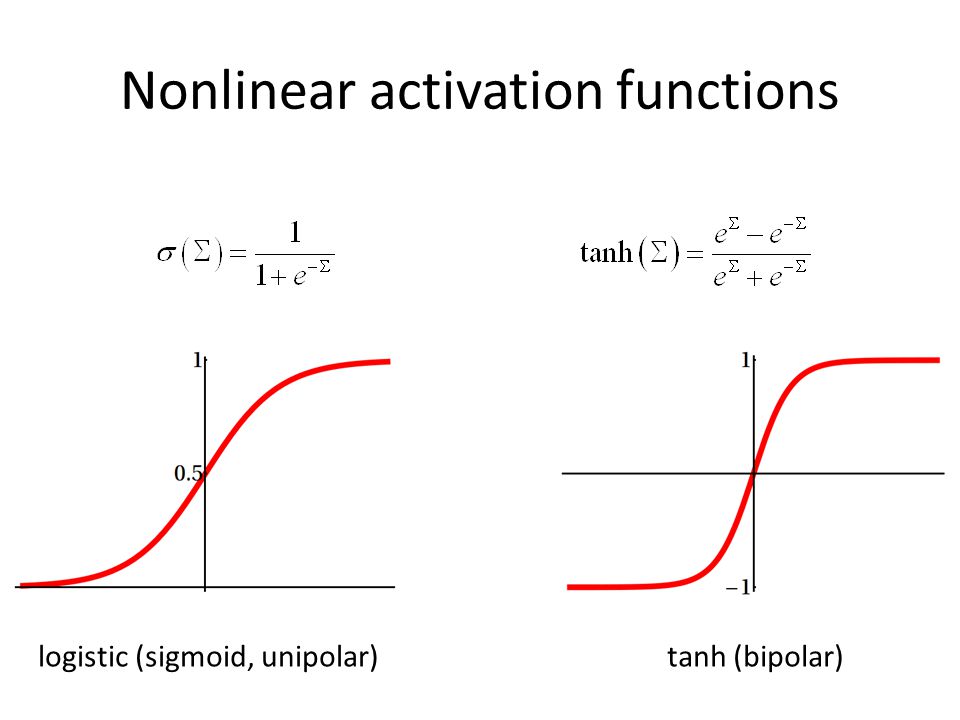
* The inputs are passed through to the neuron through synapses.
* The inputs are independent variables. They are all descriptors/observations referring to the same object/feature. They need to be standardized/normalised.
* Every input is assigned weights which are adjusted based on a variety of factors.

**Activation Function**

* Threshold function:



* Sigmoid: Advantage compared to threshold function is that this is a smooth curve
* Particularly useful in the output layer where we will be trying to make predictions



* Rectifier function: One of the most widely used activation function:

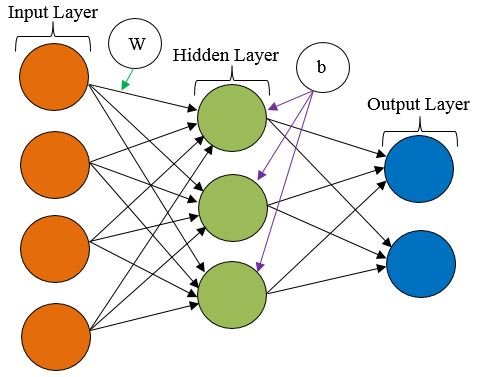


* Hyperbolic tangent function: Similar to sigmoid function but here the values go into negative too.



**How Neural Networks Work**

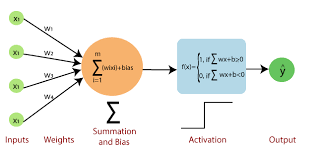
* The neurons in the hidden layer may vary in the way they assign weightages to inputs. Certain inputs can be totally disregarded while some maybe assigned higher priorities than the rest.



* The neurons functioning individually won’t be able to produce a meaningful result when they function in unison , we obtain the desired output.

**How Neural Networks Learn**

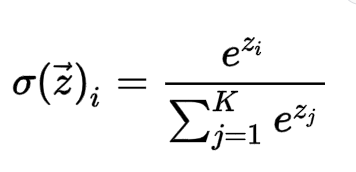
* There are two contrasting approaches. One, we can hard code the behavior that is desired or two, we code the architecture and allow the system to learn by itself.

****

* Once we obtain the output y’, we compare it with the actual value y and based on the error, we update the weights. **This process keeps repeating as long as the cost function can be further minimized.**
* **One epoch:** One complete training of the neural network over the dataset

**Multiclass classification**

* **Mutually Exclusive classes:** A datapoint can have only one class assigned to it. As a result during one hot encoding , only a single class can have the value **1.**
  + Here we use the **soft max activation** function.



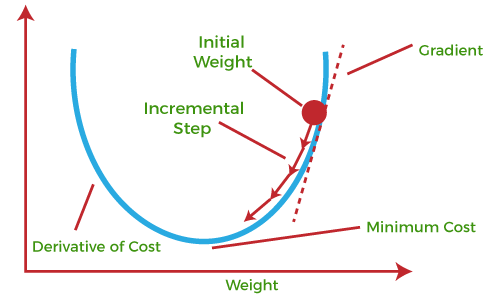
**K –** number of output classes.

The range will be between 0 and 1. This means that we sum up all the probabilities in the output layer add up to 1

* **Non-Exclusive classes:** A datapoint can have multiple classes assigned to it. Multiple classes can have the value **1** during one-hot-encoding.
  + In this case we use **sigmoid function** as the activation function. Usually, we set a threshold probability and if the output is above that we assign the data point to a particular class. In this case however we need to be careful since this threshold value could be breached for multiple classes.

**Gradient Descent**

* One of the approaches to arrive at the optimum weight values can be through brute force approach. But this isn’t feasible in many cases.
* In gradient descent we find the slope of cost function at each point. If slope is positive(rising) you move left, if it is negative(falling) we move to the right. We repeat this approach till we reach our optimal value
* The key idea is to move downhill since our ultimate idea is to obtain the minima for our cost function.
* **Adam is just another name for adaptive gradient descent.**
* **For classification problems we can opt for cross-entropy cost function.**

****

**Stochastic Gradient Descent**

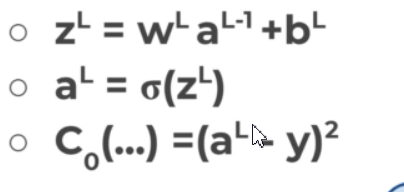
* Normal gradient requires the function to be convex (has a global minimum). We might arrive at a sub-par model when gradient descent is used for non-convex functions. (cases where local minima might mistakenly be considered as global minima)
* In the normal (batch gradient descent) we consider the entire dataset to train the model and then just the weights based on the error. In stochastic method however we train one data row at a time, adjusting weights after each row.
* This helps us overcome the shortcomings of normal batch gradient descent since the fluctuations are much higher which will help it identify the global minima rather than the local minima.
* It is also faster than batch gradient descent.
* The only pro for batch gradient descent algorithm is that it is deterministic. That is if we start with the same inputs and weights we will arrive at the same output every time.

**Read this :** [A Neural Network in 13 lines of Python (Part 2 - Gradient Descent) - i am trask](https://iamtrask.github.io/2015/07/27/python-network-part2/)

**Back-propogation**

* Due to the way the algorithm is designed, it is possible to determine which part of the neuron is responsible for the error being produced.

We have the following expressions for data within the neural network:



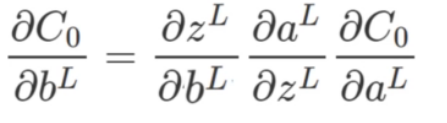
w – weights

a – activation layer output

b – bias

C- cost function

Next, we try to obtain the partial derivative of the cost function with respect to weights and split it using the chain rule:



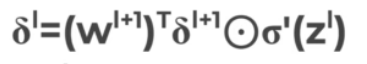
Using these, we obtain the error vector using the following formula:

 (Hadamard product: Element wise product of two vectors)



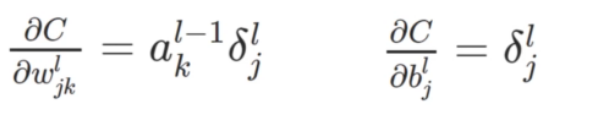
In this gradient we are expressing the rate of change of the cost function with respect to the output activations.

So the general expression for the error to be backpropagated can be expressed as :



By applying the transpose of the weight matrix (wl+1)T, we can think of moving the error backward through the network giving us an approximation of the error at the output of the lth layer.

The gradient of the cost function is given by :



In short , following are the steps involved in backpropagation:

* we use our weighted inputs , pass them through the activation function and obtain prediction through neurons
* Comparing these predictions with the actual values we obtain the cost function. We then obtain the partial derivative of this cost function with respect to the biases.
* **Significance of these gradients:** The main idea is that we can go back through the network using these gradients so that the weights and biases can be tweaked in a way such that the output of the error vector on the last output layer can be minimized.
* Using backpropagation, error terms are approximated for each layer and this continues until we obtain our minimum cost function value.

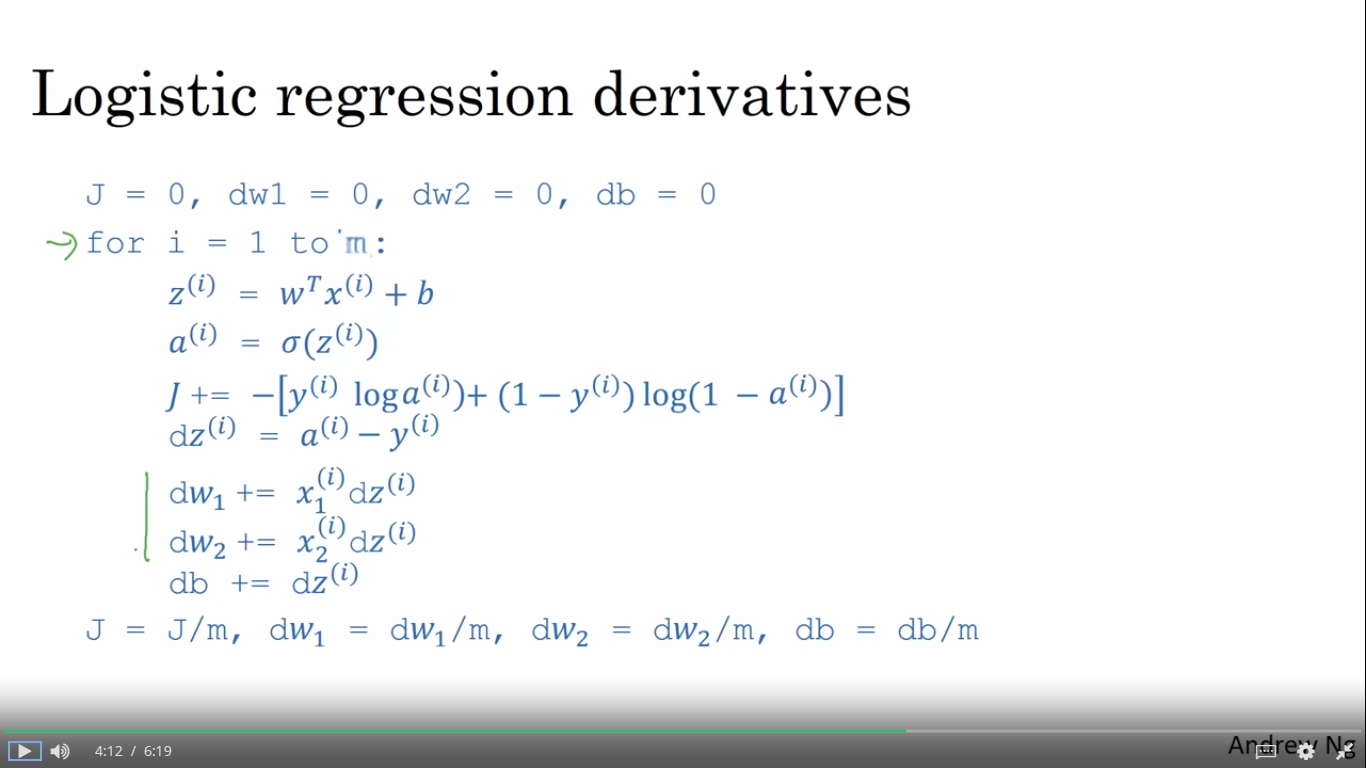
[A Step by Step Backpropagation Example – Matt Mazur](https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/)

[Neural networks and deep learning](http://neuralnetworksanddeeplearning.com/chap2.html)

**Training ANN with stochastic Gradient descent**

* Randomly initialize the weights to small numbers close to 0(but not 0)
* Input the first observation of your dataset in the input layer, each feature in one input node
* Forward propagation: Left to right: the neurons are activated in a way that the impact of each neuron’s activation is limited by the weights. Propagate the activations until getting the predicted result y.
* Compare the predicted result to the actual result. Measure the generated error.
* Backpropagation: Right to left: The error is back propagated. Update the weights according to how much they are responsible for the error. The learning rate decides by how much we update the weights
* Repeat the above steps and update the weights after each observation (Reinforcement learning) or: Repeat the steps but update the weights only after a batch of observations(batch learning)
* When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

**Algorithm for logistic regression**



**Implementation using Tensorflow and Keras**

**Importing the libraries**

import pandas as pd

import numpy as np

import tensorflow as tf

**Importing the dataset**

df = pd.read\_csv('Churn\_Modelling.csv')

X = df.iloc[:,3:-1].values

y = df.iloc[:,-1].values

**Encoding categorical Data**

Here we are obtaining the index of the gender column, since X is a numpy array and indexing through the column name is not possible.

**Label Encoding the 'Gender' column**

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

X[:,2] = le.fit\_transform(X[:,2])

**One-hot-encoding of 'Geography' column**

Here the only thing that changes is the index of the column upon which we one-hot-encode which is 1 in this case.

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder

ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthrough')

X = np.array(ct.fit\_transform(X))

**Split dataset into test set and training set**

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 0)

**Feature Scaling**

In deep learning, it is a prerequisite that feature scaling must be applied. Hence we scale everything including the dummy variables.

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

**Building the ANN**

* ANN implementation can either be sequential (with a defined input layer, hidden layers and an output) or a computational graph (where all the neurons are interconnected in a non sequential manner) like in Boltzmann machine.
* In our case we initialize a sequential object upon which our neural network is built.

ann = tf.keras.models.Sequential()

**Adding input layer and the first hidden layer**

The fully connected layer that we add will be built as an object of the dense class.

ann.add(tf.keras.layers.Dense(units=6,activation='relu'))

We must specify the number of neurons we need in our hidden layer(**units)**. There is no rule of thumb and possibly elbow methods can be used later to arrive at the optimum value.

The ‘**activation**’ parameter stands for the activation function that we desire to use. Currently we are using rectifier activation function(relu).

**Adding second hidden layer**

ann.add(tf.keras.layers.Dense(units=6,activation='relu'))

**Adding output layer**

ann.add(tf.keras.layers.Dense(units=1,activation='sigmoid'))

* The number of neurons in the output layer depend on the type of output. Currently we have a binary output and hence one neuron would suffice.
* If we had 3 possible categories, then would need 3 neurons.
* For the **output layer, we go for sigmoid activation function** since we get the probability along the predictions

**Training the ANN**

**Compiling the ANN**

ann.compile(optimizer ='adam',loss='binary\_crossentropy',metrics = ['accuracy'])

* For stochastic gradient descent, we opt for the **'adam' optimizer.**
* For non binary classification loss=**'catergorical\_crossentropy'** and the activation function in the output layer must be **'softmax'**.

**Training the ANN on the training set**

ann.fit(X\_train,y\_train,batch\_size = 32,epochs=100)

* While in theory we are using stochastic gradient descent, we specify the **'batch\_size'** for considering those many inputs at a time. It increases the efficiency of the training process.
* **'epochs'** is used to specify the number of times we need the neural network to train over the input data

**Predicting result of a single observation**

While considering the single variable take care of the following things:

* input must be passed as a 2D array
* be careful with the dummy variables. For example the observation might have the name of that particular category (country, gender etc) but what we have fed the model are the encoded values
* Scale the input before obtaining the predictions

print(ann.predict(sc.transform([[1,0,0,600,1,40,3,60000,2,1,1,50000]])))

**Predictions**

y\_pred = ann.predict(X\_test)

y\_pred = (y\_pred > 0.5)

output\_df = pd.DataFrame(data=y\_pred,columns=['predicted\_values'])

output\_df['actual\_values'] = y\_test

**Performance metrics**

from sklearn.metrics import confusion\_matrix,accuracy\_score,plot\_confusion\_matrix

con = confusion\_matrix(y\_pred,y\_test)

print(accuracy\_score(y\_test,y\_pred))

**Implementation using pytorch**

**Basic pytorch ANN(using IRIS dataset)**

**Importing libraries**

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

**Model Class**

class Model(nn.Module):

    def \_\_init\_\_(self, in\_features=4, h1=8, h2=9, out\_features=3):

        super().\_\_init\_\_()

        self.fc1 = nn.Linear(in\_features,h1)    # input layer

        self.fc2 = nn.Linear(h1, h2)            # hidden layer

        self.out = nn.Linear(h2, out\_features)  # output layer

    def forward(self, x):

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = self.out(x)

        return x

We are free to add more hidden layers if we need. The number of output features will be determined by the nature of the output.

**Instantiate the model**

torch.manual\_seed(32)

model = Model()

**Perform train-test split**

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features,labels,test\_size=0.2,random\_state=33)

X\_train = torch.FloatTensor(X\_train)

X\_test = torch.FloatTensor(X\_test)

y\_train = torch.LongTensor(y\_train)

y\_test = torch.LongTensor(y\_test)

**Defining the loss and optimizer**

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(),lr=0.01)

**Important : Usually for mutliclass classification problems, we need to do one-hot encoding during data preprocessing . But for pytorch if we consider the cross-entropy loss function this step isn't needed**

**Training the model**

epochs = 100

losses = []

for i in range(epochs):

    i = i+1

    y\_pred = model.forward(X\_train)

    #calculate loss and append it to the list

    loss = criterion(y\_pred,y\_train)

losses.append(loss.item())

    if i%10 ==0:

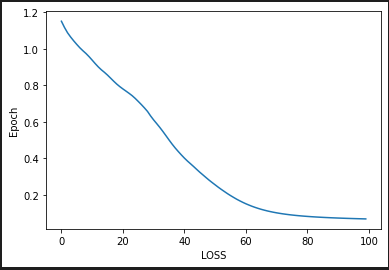
        print(f"epoch {i} loss:{loss.item()}")

optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

**Plotting the loss function**

****

**Evaluating the model performance**

We do this by obtaining the predictions of the model on the test set and then comparing the results with the actual value

with torch.no\_grad():

    y\_val = model.forward(X\_test)

    loss = criterion(y\_val, y\_test)

print(f'{loss:.8f}')

Gradient won't be calculated for code chunks under torch.no\_grad(). This helps in improving memory utilisation

**Obtaining the number of correct predictions**

correct = 0

with torch.no\_grad():

    for i,data in enumerate(X\_test):

        y\_val = model.forward(data)

        #y\_val is a tensor, cant print directly hence use str

        print(f'{i+1}.)  {str(y\_val)}   {y\_test[i]}')

        if y\_val.argmax().item() == y\_test[i]:

            correct += 1

print(f'We got {correct} correct')

The underlying principle is simple . The model prediction will hold values for each class and the highest value will be assigned to the class to which the dataset belongs. For example if the output is [0,2,-3] then the datapoint belongs to class 1.

Next we compare this with the actual class(**y\_test[i])**. If it matches then we classify it as a correct prediction.

**Saving the model**

torch.save(model.state\_dict(), 'IrisDatasetModel.pt')

This will save the learned parameters of the model but not the model class. To save the model class :

torch.save(model, PATH)

**Loading the model**

new\_model = Model()

new\_model.load\_state\_dict(torch.load('IrisDatasetModel.pt'))

new\_model.eval()

**Full ANN model (Using NYC Taxi Fare dataset)**

This project involves extensive feature engineering particularly while processing the timestamps as well as obtaining the distance between two points given their GPS coordinates. We will then separate out categorical and continuous columns to obtain a tabular model

**Need for embedding :** [(7) What does PyTorch Embedding do? - Quora](https://www.quora.com/What-does-PyTorch-Embedding-do)

**Importing the libraries**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

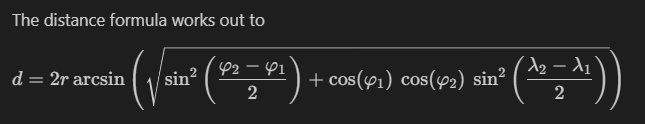
import torch

import torch.nn as nn

df = pd.read\_csv('NYCTaxiFares.csv')

**Calculating the distance travelled based on the GPS coordinates**

It is observed that the distance would be a better metric than just feeding the latitude and longitude to the model. We use the Haversine formula to measure the distance between two areas based on their GPS coordinates.



def haversine\_distance(df, lat1, long1, lat2, long2):

    """

    Calculates the haversine distance between 2 sets of GPS coordinates in df

    """

    r = 6371  # average radius of Earth in kilometers

    phi1 = np.radians(df[lat1])

    phi2 = np.radians(df[lat2])

    delta\_phi = np.radians(df[lat2]-df[lat1])

    delta\_lambda = np.radians(df[long2]-df[long1])

    a = np.sin(delta\_phi/2)\*\*2 + np.cos(phi1) \* np.cos(phi2) \* np.sin(delta\_lambda/2)\*\*2

    c = 2 \* np.arctan2(np.sqrt(a), np.sqrt(1-a))

    d = (r \* c) # in kilometers

    return d

df['dist\_km'] = haversine\_distance(df,'pickup\_latitude',

       'pickup\_longitude', 'dropoff\_latitude', 'dropoff\_longitude')

* The key takeaway is feature engineering. In this case the formula was obtained through wikipedia. Once i encounter the dataset, once i check for missing values and na values next i have to check if any such deductions can be made using the data.

**Processing the timestamp data to extract useful insights**

df['pickup\_datetime'] = pd.to\_datetime(df['pickup\_datetime'])

**pd.to\_datetime**  converts the timestamp in string format to pandas datetime format.

Accounting for the difference in datetime:

df['EDTDate'] = df['pickup\_datetime'] - pd.Timedelta(hours=4)

**pd.Timedelta** represents a duration of time. Can be used for **timestamp arithmetic**

**.dt** is the accessor object for datetime

df['Hour'] = df['EDTDate'].dt.hour

df['AMorPM'] = np.where(df['Hour']<12,'am','pm')

df['Weekday'] = df['EDTDate'].dt.strftime("%a")

**strftime** - used to convert dates in a given series object to the specified format. In this case we are extracting the days of the week

**Separating categorical columns from continuous columns**

cat\_cols = ['Hour','AMorPM','Weekday']

cont\_cols = ['pickup\_latitude', 'pickup\_longitude', 'dropoff\_latitude', 'dropoff\_longitude', 'passenger\_count', 'dist\_km']

Target label:

y\_col = ['fare\_amount']

**Converting the datatype of the categorical columns**

**Categorify :** An alternative to one-hot encoding. Pandas converts the corresponding column values to categorical codes.

# Convert our three categorical columns to category dtypes.

for cat in cat\_cols:

    df[cat] = df[cat].astype('category')

Accessing the categories:

df['columnname'].cat.categories

Accessing the category codes:

df['columnname'].cat.codes

**NaN values are assigned a code of -1. So this can be another way of cleaning our data**

**Creating a combined array comprising of categorical variables**

Next, we want to combine the three categorical columns into one input array using **numpy.stack**

hr = df['Hour'].cat.codes.values

ampm = df['AMorPM'].cat.codes.values

wkdy = df['Weekday'].cat.codes.values

cats = np.stack([hr, ampm, wkdy], 1)

Alternatively, this can be achieved using list comprehension:

cats = np.stack([df[cols].astype('category').cat.codes.values for cols in cat\_cols],axis=1)

**Converting numpy arrays to tensors**

cats = torch.tensor(cats, dtype=torch.int64)

We now stack up the continuous columns and convert them to tensors

conts = np.stack([df[cols].values for cols in cont\_cols],axis=1)

conts = torch.tensor(conts, dtype= torch.float)

The output labels:

y = torch.tensor(df[y\_col].values, dtype=torch.float)

**Setting an embedding size**

At this point, we have separated our categorical and continuous columns and also encoded our categorical columns. Encoding process is usually straightforward which involves assigning a number code to every distinct category. But a more efficient way of doing is through embeddings.

Embedding involves assigning a tensor value to each category and the size of these embedding tensors is determined by the rule :

***“The rule of thumb for determining the embedding size is to divide the number of unique entries in each column by 2, but not to exceed 50.”***

[(7) What does PyTorch Embedding do? - Quora](https://www.quora.com/What-does-PyTorch-Embedding-do)

# This will set embedding sizes for Hours, AMvsPM and Weekdays

cat\_szs = [len(df[col].cat.categories) for col in cat\_cols]

emb\_szs = [(size, min(50, (size+1)//2)) for size in cat\_szs]

emb\_szs

**Defining a tabular model**

class TabularModel(nn.Module):

    def \_\_init\_\_(self,emb\_szs,n\_cont,out\_sz,layers,p = 0.5):

        super().\_\_init\_\_()

        #This snippet creates embedding layers for the categorical variables

        self.embeds = nn.ModuleList([nn.Embedding(ni,nf) for ni,nf in emb\_szs])

        self.emb\_drop = nn.Dropout(p)

        self.bn\_cont = nn.BatchNorm1d(n\_cont) #Initialise the batch normalisation object for the continuous classes

        layerlist = []

        n\_emb = sum([nf for ni,nf in emb\_szs])

        n\_in = n\_emb + n\_cont #This is the total number of input features

        #layers is a list with dimensions of the hidden layer. So let's say in this case layers =[100,50]

        #The following for loop will churn out layers and will go on storing them in the layers list

        for i in layers:

            layerlist.append(nn.Linear(n\_in,i))

            layerlist.append(nn.ReLU(inplace=True))

            layerlist.append(nn.BatchNorm1d(i))

            layerlist.append(nn.Dropout(p))

            n\_in = i

        layerlist.append(nn.Linear(layers[-1],out\_sz)) #This will create the output layer

        #print(layerlist)

        #The layers that we need are present in the layerlist. They are executed sequentially

        self.layers = nn.Sequential(\*layerlist)

    def forward(self,x\_cat,x\_cont):

        embeddings = []

        #Assigning values to the one hot encodings

        for i,e in enumerate(self.embeds):

            embeddings.append(e(x\_cat[:,i]))

        #Concatinating the batch normalized continuous variables with the embedded categorical variables

        x = torch.cat(embeddings,1)

        x = self.emb\_drop(x)

        x\_cont = self.bn\_cont(x\_cont)

        x = torch.cat([x,x\_cont],1)

        x = self.layers(x)

        return x

**The steps involved:**

**\_\_init\_\_() function:**

* The inputs :

1. **emb\_szs :** This is a list fo tuples. Size of each categorical feature is extracted (number of distinct categories within that feature) and is passed with the embedding size
2. **n\_cont :** Number of continuous variables
3. **out\_sz :** output size (1 for regression and N for n-class classification problem)
4. **layers :** list of layer sizes. So if layers =[100,50], the first hidden layer will have 100 neurons while the second will have 50
5. **p :** Dropout probability for each layer (avoids overfitting)

* Set up the embedding layers for the categorical data with **torch.nn.ModuleList() and torch.nn.Embedding()**

self.embeds = nn.ModuleList([nn.Embedding(ni,nf) for ni,nf in emb\_szs])

* Setup a dropout function for the embeddings with the given probability **p.**
* Initialize a normalization function for the continuous variables with **torch.nn.BatchNorm1d()**

self.bn\_cont = nn.BatchNorm1d(n\_cont)

* Obtain the total number of input features by adding the number of continuous features(**n\_cont)** and the number of desired embedding layers(nf)
* Set up a sequence of neural network layers where each level includes a Linear function, an activation function(ReLU) , a normalization step and a dropout layer. The for loop will generate layers based on the inputs provided in **layers.** Upon generation of these layers, we pass the list to **nn.Sequential()**

self.layers = nn.Sequential(\*layerlist)

**forward function()**

* Preprocess the embeddings and normalize the continuous variables before passing them through the layers. In this step, tensor values are assigned to the embedded categorical data .
* We then use **torch.cat()**  to combine multiple tensors into one

**Initialize the model**

torch.manual\_seed(33)

model = TabularModel(emb\_szs,conts.shape[1],1,[200,100], p = 0.4)

**Loss function and optimizer**

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(),lr=0.001)

Perform train-test split and then head to model training:

import time

start\_time = time.time()

epochs = 100

losses = []

for i in range(epochs):

    i += 1

    y\_pred = model(cat\_train, cont\_train)

    loss = torch.sqrt(criterion(y\_pred,y\_train))

    losses.append(loss.item())

    if i%10 == 0:

        print(f'epoch {i} loss is {loss.item()}')

    optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

duration = time.time() - start\_time

print(f'Training duration is {duration/60} minutes')

**Evaluating the model loss**

with torch.no\_grad():

    y\_val = model(cat\_test,cont\_test)

    loss = torch.sqrt(criterion(y\_val,y\_test))

**Displaying the predictions**

for i in range(10):

    diff = np.abs(y\_val[i].item() - y\_test[i].item())

    print(f'{i}.) Predicted value:{y\_val[i]} Actual value:{y\_test[i]} Difference : {diff}')

**Full ANN : classification**

* The code remains almost the same except for these minor changes:
* the target y column will be **‘fare\_class’ (**categorical column)
* While instantiating the model choose N output neurons instead of 1. In this case there are 2 classes and therefore N=2

model = TabularModel(emb\_szs,conts.shape[1],2,[200,100], p = 0.4)

* Initialise the cross entropy loss function

**Obtaining prediction on new data:**

def test\_data(mdl): # pass in the name of the new model

    # INPUT NEW DATA

    plat = float(input('What is the pickup latitude?  '))

    plong = float(input('What is the pickup longitude? '))

    dlat = float(input('What is the dropoff latitude?  '))

    dlong = float(input('What is the dropoff longitude? '))

    psngr = int(input('How many passengers? '))

    dt = input('What is the pickup date and time?\nFormat as YYYY-MM-DD HH:MM:SS     ')

    # PREPROCESS THE DATA

    dfx\_dict = {'pickup\_latitude':plat,'pickup\_longitude':plong,'dropoff\_latitude':dlat,

         'dropoff\_longitude':dlong,'passenger\_count':psngr,'EDTdate':dt}

    dfx = pd.DataFrame(dfx\_dict, index=[0])

    dfx['dist\_km'] = haversine\_distance(dfx,'pickup\_latitude', 'pickup\_longitude',

                                        'dropoff\_latitude', 'dropoff\_longitude')

    dfx['EDTdate'] = pd.to\_datetime(dfx['EDTdate'])

    # We can skip the .astype(category) step since our fields are small,

    # and encode them right away

    dfx['Hour'] = dfx['EDTdate'].dt.hour

    dfx['AMorPM'] = np.where(dfx['Hour']<12,0,1)

    dfx['Weekday'] = dfx['EDTdate'].dt.strftime("%a")

    dfx['Weekday'] = dfx['Weekday'].replace(['Fri','Mon','Sat','Sun','Thu','Tue','Wed'],

                                            [0,1,2,3,4,5,6]).astype('int64')

    # CREATE CAT AND CONT TENSORS

    cat\_cols = ['Hour', 'AMorPM', 'Weekday']

    cont\_cols = ['pickup\_latitude', 'pickup\_longitude', 'dropoff\_latitude',

                 'dropoff\_longitude', 'passenger\_count', 'dist\_km']

    xcats = np.stack([dfx[col].values for col in cat\_cols], 1)

    xcats = torch.tensor(xcats, dtype=torch.int64)

    xconts = np.stack([dfx[col].values for col in cont\_cols], 1)

    xconts = torch.tensor(xconts, dtype=torch.float)

    # PASS NEW DATA THROUGH THE MODEL WITHOUT PERFORMING A BACKPROP

    with torch.no\_grad():

        z = mdl(xcats, xconts).argmax().item()

    print(f'\nThe predicted fare class is {z}')

**Tabular model summary:**

* Separate out the categorical and continuous columns
* Convert the datatype of categorical columns thus encoding their values in the process
* Create separate numpy arrays with categorical and continuous variables stacked in them respectively.
* Embed the categorical values using ModuleList() and Embedding() and obtain tensor values for the categorical codes. Once this done concatenate these values with the batch normalised continuous values.

**Full ANN with MNIST dataset**

* Each image in this a 28 x28 array. We flatten this out into a 784x1 array but will lose some information like the relationship between adjacent pixels while doing so.

**Loading the MNIST Dataset**

Defining Transforms:

transform = transforms.ToTensor()

Usually we can apply transformations like reshape, normalize etc using this. As of now , we cab just proceed with a basic operation.

**Loading the test and training set**

train\_data = datasets.MNIST(root='../Data',train=True, download =True,transform=transform)

test\_data = datasets.MNIST(root='../Data',train=False, download =True,transform=transform)

Upon examining the records from here , it is seen that data is present as a two item tuple. First item is our 28x28 tensor representing the image and second is a label.

image , label = train\_data[0]

image.shape

torch.Size([1, 28, 28])

The images are of the shape (**color\_channels,width,height)**

**Viewing the image**

plt.imshow(train\_data[0][0].reshape((28,28)), cmap="gray");

**Note :** That color\_channel dimension must not be fed to the imshow function. plt.imshow() expects images of the shape **(width,height)**

**Batch loading with dataloader**

torch.manual\_seed(101)

train\_loader = DataLoader(train\_data,batch\_size=100,shuffle=True)

test\_loader = DataLoader(test\_data,batch\_size=500,shuffle=False)

**Viewing a batch of images**

from torchvision.utils import make\_grid

np.set\_printoptions(formatter=dict(int=lambda x: f'{x:4}')) # to widen the printed array

# Grab the first batch of images

for images,labels in train\_loader:

    break

# Print the first 12 labels

print('Labels: ', labels[:12].numpy())

# Print the first 12 images

im = make\_grid(images[:12], nrow=12)  # the default nrow is 8

plt.figure(figsize=(10,4))

# We need to transpose the images from CWH to WHC

plt.imshow(np.transpose(im.numpy(), (1, 2, 0)));

images within trainloader have shape **[batch\_size, color\_channels,width,height]**

**Defining the model**

In this case since we have 28x28 image samples, input size =784

class MultiLayerPerceptron(nn.Module):

    def \_\_init\_\_(self,in\_sz = 784, out\_sz =10,layers=[120,84]):

        super().\_\_init\_\_()

        layer\_list =[]

        for i in layers:

            layer\_list.append(nn.Linear(in\_sz,i))

            layer\_list.append(nn.ReLU(inplace=True))

            in\_sz = i

        layer\_list.append(nn.Linear(layers[-1],out\_sz))

        self.layers = nn.Sequential(\*layer\_list)

    def forward(self,x):

        X = self.layers(x)

        return F.log\_softmax(X,dim=1) #Multi class classification

torch.manual\_seed(101)

model = MultilayerPerceptron()

Pretty much resembles the simple ANN except that we are ultimately using F.log\_softmax(X,dim=1)

**Counting the model parameters**

for param in model.parameters():

    print(param.numel())

**Loss function and optimizer**

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(),lr=0.001)

**Flattening the input images**

Before we move onto training,we need to flatten the input images.The batch tensors fed in by dataloader are of the shape [100,1,28,28] which needs to be converted to [100,784] ie **[batch\_size, c\*w\*h]**

images.view(100,-1)

This syntax means, the specified dimension stays intact(batch size in this case) while rest of the dimensions are squashed together

**Training and evaluation**

import time

start\_time = time.time()

#Training

epochs = 10

train\_losses = []

test\_losses = []

train\_correct = []

test\_correct = []

for i in range(epochs):

    trn\_corr = 0

    test\_corr =0

    for b, (X\_train,y\_train) in enumerate(train\_loader):

        b += 1

        y\_pred = model(X\_train.view(100,-1))

        loss = criterion(y\_pred,y\_train)

        predicted = torch.max(y\_pred.data,1)[1]

        batch\_corr = (predicted == y\_train).sum()

        trn\_corr += batch\_corr

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

        if b%200 == 0:

            acc = trn\_corr.item()\*100 /(100\*b)

            print(f'Epoch: {i} \t batch: {b} \t loss{loss.item()} accuracy:{acc}')

    train\_losses.append(loss.item())

    train\_correct.append(trn\_corr)

    with torch.no\_grad():

        for b,(X\_test,y\_test) in enumerate(test\_loader):

            y\_val = model(X\_test.view(500,-1))

            predicted = torch.max(y\_val.data,1)[1]

            test\_corr += (predicted==y\_test).sum()

    loss =  criterion(y\_val,y\_test)

    test\_losses.append(loss.item())

    test\_correct.append(test\_corr)

total\_time = time.time() - start\_time

print(f'Duration: {total\_time/60} minutes')

**Note :** test\_losses , train\_correct, test\_correct are only for visualizing the results and not compulsory for the training process

* The training procedure is the same as before. Only difference is the inner for loop which helps us process one batch at a time.
* **b** is the batch number which is incremented. The important part is the usage of **torch.max()** to obtain the prediction
* **torch.max()** returns a tensor of maximum values, and a tensor of the indices where the max values were found. In our code we're asking for the index positions of the maximum values along dimension 1. In this way we can match predictions up to image labels. We then compare these indices with the test set labels and calculate the total number of correct predictions.

predicted = torch.max(y\_val.data,1)[1]

* The code under **torch.no\_grad()**  is used to evaluate the performance of the model on the training set after every epoch of training

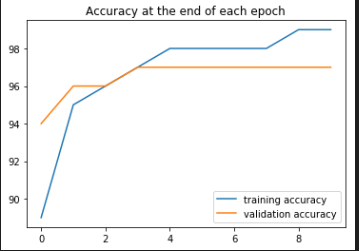
**Plotting the accuracy on test vs training set**

plt.plot([t/600 for t in train\_correct], label='training accuracy') #60000/100 = 600 batches

plt.plot([t/100 for t in test\_correct], label='validation accuracy')

plt.title('Accuracy at the end of each epoch')

plt.legend();

****

**Evaluating test data**

However, we'd like to compare the predicted values to the ground truth (the y\_test labels), so we'll run the test set through the trained model all at once.

test\_load\_all = DataLoader(test\_data,batch\_size=10000,shuffle=False)

with torch.no\_grad():

    correct = 0

    for X\_test, y\_test in test\_load\_all:

        y\_val = model(X\_test.view(len(X\_test), -1))  # pass in a flattened view of X\_test

        predicted = torch.max(y\_val,1)[1]

        correct += (predicted == y\_test).sum()

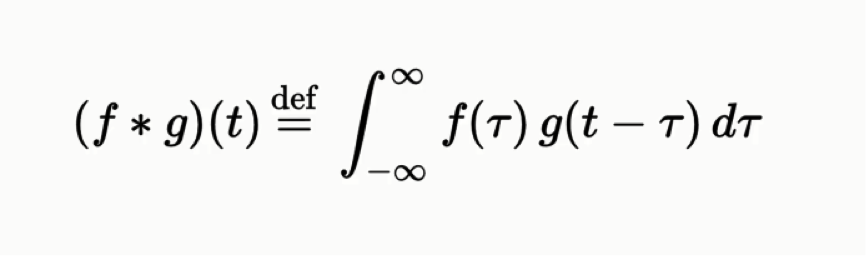
print(f'Test accuracy: {correct.item()}/{len(test\_data)} = {correct.item()\*100/(len(test\_data)):7.3f}%')

Test accuracy: 9725/10000 = 97.250%

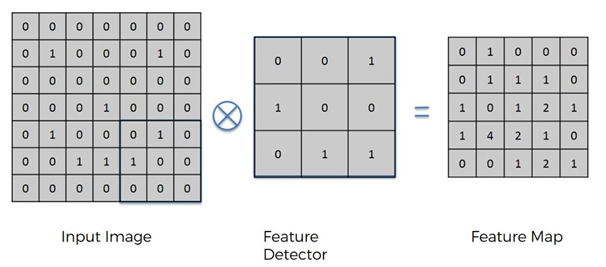
**Convolutional Neural Networks(CNN)**

* It is a deep learning algorithm which takes in an input image, assign importance to certain features within the image and be able to differentiate between them.
* Due to localized connections, there are comparatively lesser number of parameters.

**Step 1: Convolution: Extract features from within the image**

****

* We use a feature detector/kernel/filter (which is usually a 3x3 matrix) and convolute this with the input image. It is placed at the first pixel of the image and moves by **strides** which are specified in advance.
* This results in the creation of a feature map (result of convolution of input image with feature detector). It is typically smaller than the input image, which is one of the desired motives since it makes the processing faster.
* Although we may lose some information, the features that we are interested in are emphasised in the output. Typically, in the feature map, the pixels with the maximum values typically hold the features that we are interested in.

****

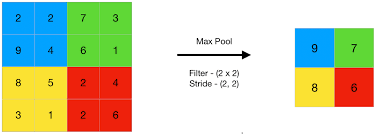
* We create many feature maps using different filter maps to obtain our first convolutional layer.
* Conclusion is that , although the size is reduced, the spatial relationships between the pixels are preserved.

**Step 1B: ReLU layer : Increase Non-linearity**

* Reason we use it is to **increase the non linearity** in our image. We do this because images by nature are non linear due to a variety of colours, edges and otehr factors. Performing mathematical operations like convolution might increase the linearity which isn’t ideal.

**Step 2 – Max Pooling (downsampling) : Reduces size, introduces spatial invariance and avoids overfitting**

* The neural network we train must have the feature of **‘spatial invariance’**: there must be flexibility with regards to the positioning, variability as well as the orientation of a feature. Remember how we augmented the baggage dataset on different backgrounds.
* Take a matrix/box of a particular size in the feature map and select the maximum value within that box. Consider only that value and then move the box to the next position. Ultimately we obtained a **pooled feature map.**

****

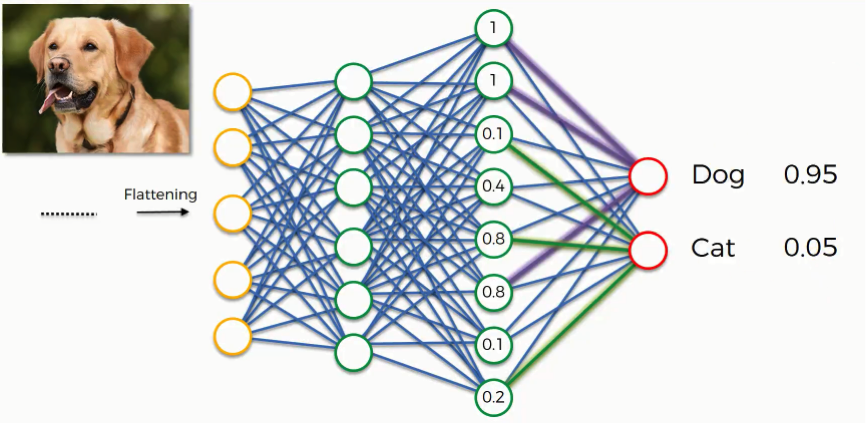
* By doing this, we are not just preserving the features, we also account for any distortion in the features. Example, the maximum pixel value of a feature can occur in any of the 4 pixel positions (in case a 2x2 box is chosen) and the result would still be the same. This is one of the key aspects towards establishing spatial invariance.
* Also, the size is reduced making the processing more efficient and avoids overfitting.
* It need noot just be max pooling, depending on the requirements we can go for average pooling, sum pooling or any other suitable pooling method.

**Step 3: Flattening**

* In this step, we flatten the pooling layers and all the pixel values are entered into a single vector which would be the input layer of the artificial neural network.

**Step 4: Full Connection : Voting is carried out by neurons in the hidden layer and predictions are ultimately made based on the weightage assigned to these votes**

* The flattened vector is fed into a neural network. The hidden layers have to be fully connected, which was not necessarily the case in artificial neural networks.
* We obtain prediction through this ANN, the cost function is calculated and the backpropogation step adjusts the weights for the next epoch.
* The output neurons vote/assign weightages to the inner layer neurons based on how reliable their outputs are.

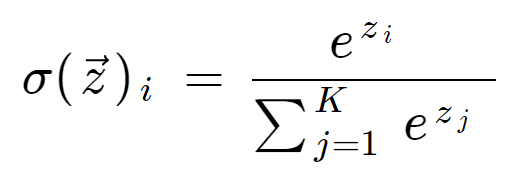
****

* For example, say we are classifying an image to be either a dog or a cat. Let’s say the image being fed currently is that of a dog. After certain epochs of training, the output neurons know that it is indeed a dog. Now, let’s say inner layer neurons A,B and C send strong probabilities to both classes that the image might belong to them.
* For dog class, this prediction holds up, because it is in fact the picture of a dog and hence high weightage is assigned to these neurons. For cat class however, these predictions are entirely the opposite and hence low weightages are assigned to these neurons .
* The cat class will assign high weightages that predict low probabilities of the image belonging to cat class. This kind of virtual voting is carried out by the neurons in the fully connected layer during every epoch and finally the **output neurons learn which fully connected neuron to listen to.** This effectively means the model differentiates the important/decisive features from the rest.

[The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3) – Adit Deshpande – Engineering at Forward | UCLA CS '19 (adeshpande3.github.io)](https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html)

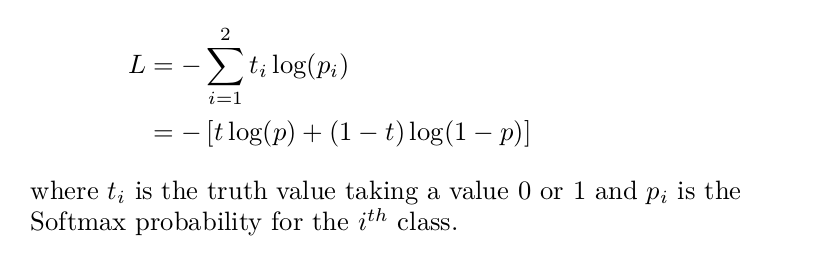
**Soft-Max and Cross Entropy**

* In the previous example we noticed that the neural network outputs probabilities for each neuron which ultimately adds up to 1. There might be a question asking why this is the case?
* Ideally the output neurons house real values which are normalized using something known as a softmax function.

****

**Cross entropy**

* Analogous to mean squared error in supervised learning. This is the loss function that we intend to minimize to obtain a better approximation.

****

* This can be a metric to compare the performance of two models. In the above dog-cat case, let us say two models both make the same predictions but their probabilities of predictions may differ(one might predict that the image is that of a dog with probability 0.8 while the other might predict that with a probability of 0.5).
* Cross entropy vs Mean squared error: When the output value has a small magnitude , improvement of model performance through mean squared error becomes a tedious process. We prefer cross entropy only for classification problems.

**References**

* Jeffry Hinton Youtube video(soft max output function)
* [A Friendly Introduction to Cross-Entropy Loss (rdipietro.github.io)](https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/)

**Implementation**

**Importing Libraries**

ImageDataGenerator is being used to preprocess the images

import pandas as pd

import tensorflow as tf

from keras.preprocessing.image import ImageDataGenerator

**Part 1- Data Preprocessing**

**Preprocessing the training set**

* We will apply some transformations on the images in the training set to avoid overfitting. This process of apply transformations is called image augmentation.
* Note – The images in the training set are both augmented and scaled. Whereas images in the test scaled are only scaled and no augmentation is carried out.

train\_datagen = ImageDataGenerator(

        rescale=1./255,  #this is for feature scaling. All the pixel values will be between 0 and 1

        shear\_range=0.2,

        zoom\_range=0.2,

        horizontal\_flip=True)

The ‘**rescale’** parameter is used to scaled every individual pixel of the input image.

Now that the image augmentation tool has been initialized , this has to be linked to the images in our training set.

training\_set = train\_datagen.flow\_from\_directory(

        'dataset/training\_set', #path of training set

        target\_size=(64, 64), # Size of inputs to be fed to neural network

        batch\_size=32,

        class\_mode='binary') #since ours is a binary classification problem

**‘target\_size’ :** Here we specify the size of the input images being fed to the neural network. Further down we will be using this size while initializing the convolution layer as well as obtaining predictions for single images.

**Preprocessing the test set**

test\_datagen = ImageDataGenerator(

        rescale=1./255)

test\_set = test\_datagen.flow\_from\_directory(

        'dataset/test\_set',

        target\_size=(64, 64),

        batch\_size=32,

        class\_mode='binary')

As mentioned previously, augmentation is not necessary for the test set images

**Building the CNN**

**Initializing the CNN**

cnn = tf.keras.models.Sequential()

**Step 1 – Convolution**

cnn.add(tf.keras.layers.Conv2D(filters=32,kernel\_size=3,activation='relu',input\_shape=[64,64,3]))

* The **‘filters’** argument is to specify how many feature detectors we would like to use. We can look into various commonly used filter architecture.
* **'input\_shape'** is the shape of the images that are being fed. Since we are feeding 64 by 64 color images this will be (64,64,3)
* **‘kernel\_size’** : Specify the height and width of the 2D convolution window

**Step 2 – Pooling**

cnn.add(tf.keras.layers.MaxPool2D(pool\_size=2,strides=2))

* **'pool\_size'**: size of the pooling window
* **'strides'** - by how many pixels should the pooling window be shifted at every step

**Adding a second convolutional layer**

cnn.add(tf.keras.layers.Conv2D(filters=32,kernel\_size=3,activation='relu'))

cnn.add(tf.keras.layers.MaxPool2D(pool\_size=2,strides=2))

The ‘**input\_shape**’ parameter is only added to the first layer to connect it to the input layer

**Step 3 – Flattening**

cnn.add(tf.keras.layers.Flatten())

**Step 4 – Full Connection**

We feed this to an artificial neural network.

cnn.add(tf.keras.layers.Dense(units=128,activation='relu'))

**Step 5 – Output layer**

cnn.add(tf.keras.layers.Dense(units=1,activation='sigmoid'))

**Training the CNN**

**Compiling the CNN**

cnn.compile(optimizer='adam',loss='binary\_crossentropy',metrics=['accuracy'])

**Training the CNN on training set and evaluating it on the test Set**

cnn.fit(training\_set,validation\_data=test\_set,epochs=25)

**Making Single predictions**

import numpy as np

from keras import utils

from keras\_preprocessing import image

test\_image = utils.load\_img('dataset/single\_prediction/cat\_or\_dog\_3.jpg',target\_size=(64,64))

* The first parameter is the path where our test image is stored
* The image for which we wish to obtain predictions must be of the same size as the input images. Therefore we use the **'target\_size'** parameter to specify the shape
* This test\_image will be in PIL format which needs to be converted to a 2D array

test\_image = image.img\_to\_array(test\_image)

**Converting single image into a batch of images(adding an extra dimension)**

Although the image is now in 2D array format, we had earlier fed the images in batches of 32. Hence ,the model can't be expected to take in a single image input. We need to put this single image into a batch by adding an extra dimension to it.

test\_image = np.expand\_dims(test\_image,axis = 0)

Usually the first dimension is the batch of images hence we mention **axis=0.**

result = cnn.predict(test\_image/255.0)

The result will hold 0’s and 1’s which can be difficult to interpret . Hence, we go for the following:

training\_set.class\_indices

if result[0][0] > 0.5:

    prediction = 'dog'

else:

    prediction='cat'

print(prediction)

**Using Pytorch**

1. **CNN on MNIST dataset**

**Defining Transforms:**

transform = transforms.ToTensor()

**Loading the test and training set**

train\_data = datasets.MNIST(root='../Data',train=True, download =True,transform=transform)

test\_data = datasets.MNIST(root='../Data',train=False, download =True,transform=transform)

**Loading data in batches using dataloader**

train\_loader = DataLoader(train\_data,batch\_size=10,shuffle=True)

test\_loader = DataLoader(test\_data,batch\_size=10,shuffle=False)

Before we move onto defining our Convolutional model class, we can break down the operations that will be carried out there.

To process an individual data point(image) add an additional dimension so that the model sees it as a batch:

x = X\_train.view(1,1,28,28)

x = F.relu(conv1(x))

x.shape

torch.Size([1, 6, 26, 26])

* It is noticed that the image obtained is 26x26 as against the original image which was 28 x 28. this is because the border information is being and we haven't applied padding.

x = F.max\_pool2d(x,2,2)

x.shape

torch.Size([1, 6, 13, 13])

* Because we are using a 2x2 kernel with a stride of 2 we are effectively cutting our image into half. That explains the reduction in size

x = F.relu(conv2(x))

x.shape

torch.Size([1, 16, 11, 11])

x = F.max\_pool2d(x,2,2)

x.shape

torch.Size([1, 16, 5, 5])

x.view(-1,16\*5\*5).shape

We feed this flattened tensor to the fully connected network.

**Model Class**

class ConvolutionalNetwork(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 6, 3, 1)

        self.conv2 = nn.Conv2d(6, 16, 3, 1)

        self.fc1 = nn.Linear(5\*5\*16, 120)

        self.fc2 = nn.Linear(120, 84)

        self.fc3 = nn.Linear(84,10)

    def forward(self, X):

        X = F.relu(self.conv1(X))

        X = F.max\_pool2d(X, 2, 2)

        X = F.relu(self.conv2(X))

        X = F.max\_pool2d(X, 2, 2)

        X = X.view(-1, 5\*5\*16)

        X = F.relu(self.fc1(X))

        X = F.relu(self.fc2(X))

        X = self.fc3(X)

        return F.log\_softmax(X, dim=1)

torch.manual\_seed(42)

model = ConvolutionalNetwork()

model

**Why it is 5x5x16 in fc1 . Images in MNIST have the shape (1,28,28) .**

\* after conv2d (1,6,26,26) where 6 - number of output filters in first conv layer

\* pooling (1,6,13,13)

\* conv2d (1,16,11,11) where 16 - number of output filters in second conv layer

\* pooling (1,16,5,5)

**Calculating the number of parameters**

total = []

for param in model.parameters():

    print(param.numel())

    total.append(param.numel())

print(f'Total parameters {sum(total)}')

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(),lr=0.001)

**Training the model**

import time

start\_time = time.time()

epochs = 5

train\_losses = []

test\_losses = []

train\_corr = []

test\_corr = []

for i in range(epochs):

    trn\_corr = 0

    tst\_corr = 0

    for b,(X\_train,y\_train) in enumerate(train\_loader):

        b+= 1

        y\_pred = model(X\_train)

        loss = criterion(y\_pred,y\_train)

        predictions = torch.max(y\_pred.data,1)[1]

        batch\_corr = (predictions == y\_train).sum()

        trn\_corr += batch\_corr

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

        if b%600 == 0:

            print(f'epoch: {i:2}  batch: {b:4} [{10\*b:6}/60000]  loss: {loss.item():10.8f}  \

accuracy: {trn\_corr.item()\*100/(10\*b):7.3f}%')

    train\_losses.append(loss.item())

    train\_corr.append(trn\_corr)

    with torch.no\_grad():

        for b, (X\_test, y\_test) in enumerate(test\_loader):

            # Apply the model

            y\_val = model(X\_test)

            # Tally the number of correct predictions

            predicted = torch.max(y\_val.data, 1)[1]

            tst\_corr += (predicted == y\_test).sum()

    loss = criterion(y\_val, y\_test)

    test\_losses.append(loss.item())

    test\_corr.append(tst\_corr)

print(f'\nDuration: {time.time() - start\_time:.0f} seconds')

**Evaluating all test data**

# Extract the data all at once, not in batches

test\_load\_all = DataLoader(test\_data, batch\_size=10000, shuffle=False)

with torch.no\_grad():

    correct = 0

    for X\_test, y\_test in test\_load\_all:

        y\_val = model(X\_test)  # we don't flatten the data this time

        predicted = torch.max(y\_val,1)[1]

        correct += (predicted == y\_test).sum()

print(f'Test accuracy: {correct.item()}/{len(test\_data)} = {correct.item()\*100/(len(test\_data)):7.3f}%')

**Obtaining prediction for a single image**

plt.imshow(test\_data[7585][0].reshape(28,28))

View what the image is .

with torch.no\_grad():

    new\_prediction = model(test\_data[7585][0].view(1,1,28,28)).argmax()

    print(f'The predicted number is {new\_prediction.item()}')

Important thing is to use **view()** to process the single image as a batch.

1. **CNN on CIFAR 10 dataset**

* 32x32 color images belonging to 10 classes
* The code is exactly the same except for two minor changes:

self.conv1 = nn.Conv2d(3,6,5,1)

Since we are dealing with color images, the number of color channels =3 and hence the number of input features = 3 instead of 1

self.fc1 = nn.Linear(6\*6\*16,120)

While flattening the images, the calculation slightly differs:

**Why it is 6x6x16 in fc1 . Images in cifar have the shape (3,32,32):**

* after conv2d (3,6,30,30) #6 - number of output filters in first conv layer
* pooling (3,6,15,15)
* conv2d (3,16,13,13) #16 - number of output filters in second conv layer
* pooling (3,16,6,6)

Apart from this everything else remains the same including the training process

1. **Loading real images and building CNN for that data**

Displaying images using matplotlib **imshow():**

plt.imshow(np.transpose(im.numpy(),(1,2,0)))

The imshow() function expects the image in (width, height, color\_channel) format and hence this transposing operation is necessary.

We use PIL library to read and open image files:

import os

from PIL import Image

from IPython.display import display

# Filter harmless warnings

import warnings

warnings.filterwarnings("ignore")

Opening a sample image

with Image.open('../Data/CATS\_DOGS/test/CAT/10107.jpg') as im:

    display(im)

**Creating a list of file imagenames:**

path = 'D:\\Courses\\Pytorch\\Convolutional Neural Networks\\Data\\CATS\_DOGS'

img\_names = []

for folder,subfolder,filenames in os.walk(path):

    for img in filenames:

        img\_names.append(folder+'\\'+img)

**Creating a dataframe of image sizes:**

Like we look out for NaN files while data cleaning csv files, we need to be vary of 0-by-0 images in the dataset. We check for these by checking the image sizes:

img\_size = []

rejected = []

for item in img\_names:

    try:

        with Image.open(item) as img:

            img\_size.append(img.size)

    except:

        rejected.append(item)

print(len(‘Image sizes: ‘,img\_size))

print(len(‘Rejected images: ‘,rejected))

# Convert the list to a DataFrame

df = pd.DataFrame(img\_sizes)

# Run summary statistics on image widths

df[0].describe()

# Run summary statistics on image heights

df[1].describe()

This is crucial because the statistics help us understand the general trend within the images being used. This helps us choose the general image size which we will be feeding in next.

**Image Preprocessing**

* Any model/ network that we define requires a consistent input image size across all samples.

**Transformations**

One of the basic transformations is to read the input image and convert it to a tensor. It must be noted that while doing so, the images are normalized ie, the pixel values are divided by 255.

transform = transforms.Compose([

    transforms.ToTensor()

])

im = transform(dog)

print(im.shape)

plt.imshow(np.transpose(im.numpy(), (1, 2, 0)));

So considering the pixel im[0,0]:

Before transformation:

90 95 98

After transformation:

tensor([0.3529, 0.3725, 0.3843])

* It must also be noted that the tensors are being transposed before being displayed via the imshow function.

**Resizing the image (transfoms.**

**resize(single\_value) :** the shorter edge of the image is matched to this value

**resize((val1,val2)) :** The image is reshaped to this dimension

transform = transforms.Compose([

    transforms.Resize(224),

    transforms.ToTensor()

])

im = transform(dog)

plt.imshow(np.transpose(im.numpy(),(1,2,0)))

**Center crop(transforms.CenterCrop(size))**

transform = transforms.Compose([

transforms.Resize(224),

    transforms.CenterCrop(224),

    transforms.ToTensor()

])

im = transform(dog) # this crops the original image

print(im.shape)

plt.imshow(np.transpose(im.numpy(), (1, 2, 0)));

A square of the specified size is cropped from the centre. Advisable to first use Resize followed by centre crop.

**Affine transformation –** An affinetransformation is one that preserves points and straight lines. Examples include rotation, reflection, and scaling. For instance, we can double the effective size of our training set simply by flipping the images.

**Random horizontal flip**

transform = transforms.Compose([

    transforms.RandomHorizontalFlip(p=1),  # normally we'd set p=0.5

    transforms.ToTensor()

])

im = transform(dog)

The input images are randomly rotated with a probability of p.

**Random rotation**

transform = transforms.Compose([

    transforms.RandomRotation(30),  # rotate randomly between +/- 30 degrees

    transforms.ToTensor()

])

If degrees is a number instead of sequence like (min, max), the range of degrees will be (-degrees, +degrees).

Putting them all together,

transform = transforms.Compose([

    transforms.RandomHorizontalFlip(p=1),  # normally we'd set p=0.5

    transforms.RandomRotation(30),

    transforms.Resize(224),

    transforms.CenterCrop(224),

    transforms.ToTensor()

])

im = transform(dog)

**Normalization** (transforms.Normalize(mean,standard deviation))

Once we have loaded the images into the tensor, next step is the normalization

transforms.Normalize([0.485, 0.456, 0.406],

                         [0.229, 0.224, 0.225])

* The values of mean = [0.485, 0.456, 0.406] and standard deviation = [0.229, 0.224, 0.225] have been found to work well with research
* The purpose of normalization is to make sure convergence happens faster during training

**Denormalization**

* In order to display these images back in their original form, we need to denormalize them

inv\_normalize = transforms.Normalize(

    mean=[-0.485/0.229, -0.456/0.224, -0.406/0.225],

    std=[1/0.229, 1/0.224, 1/0.225]

)

im\_inv = inv\_normalize(im)

plt.figure(figsize=(12,4))

plt.imshow(np.transpose(im\_inv.numpy(), (1, 2, 0)));

**Building the model**

**Define transforms**

train\_transform = transforms.Compose([

    transforms.RandomRotation(10),

    transforms.RandomHorizontalFlip(),

    transforms.Resize(224),

    transforms.CenterCrop(224),

    transforms.ToTensor(),

    transforms.Normalize(mean=[0.485,0.456,0.406],std=[0.229,0.224,0.225])

])

test\_transform = transforms.Compose([

    transforms.Resize(224),

    transforms.CenterCrop(224),

    transforms.ToTensor(),

    transforms.Normalize(mean=[0.485,0.456,0.406],std=[0.229,0.224,0.225])

])

**Preparing the train and test set using loaders**

root = '../Data/CATS\_DOGS'

train\_data = datasets.ImageFolder(os.path.join(root, 'train'), transform=train\_transform)

test\_data = datasets.ImageFolder(os.path.join(root, 'test'), transform=test\_transform)

torch.manual\_seed(42)

train\_loader = DataLoader(train\_data, batch\_size=10, shuffle=True)

test\_loader = DataLoader(test\_data, batch\_size=10, shuffle=True)

class\_names = train\_data.classes

print(class\_names)

print(f'Training images available: {len(train\_data)}')

print(f'Testing images available:  {len(test\_data)}')

**Important :** For inbuilt datasets we directly loaded them here. now we use **datasets.ImageFolder(path,transforms)**

**Displaying a batch of 10 images with their labels**

# Grab the first batch of 10 images

for images,labels in train\_loader:

    break

# Print the labels

print('Label:', labels.numpy())

print('Class:', \*np.array([class\_names[i] for i in labels]))

im = make\_grid(images, nrow=5)  # the default nrow is 8

# Inverse normalize the images

inv\_normalize = transforms.Normalize(

    mean=[-0.485/0.229, -0.456/0.224, -0.406/0.225],

    std=[1/0.229, 1/0.224, 1/0.225]

)

im\_inv = inv\_normalize(im)

# Print the images

plt.figure(figsize=(12,4))

plt.imshow(np.transpose(im\_inv.numpy(), (1, 2, 0)));

**Define the model**

class ConvolutionalNetwork(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 6, 3, 1)

        self.conv2 = nn.Conv2d(6, 16, 3, 1)

        self.fc1 = nn.Linear(54\*54\*16, 120)

        self.fc2 = nn.Linear(120, 84)

        self.fc3 = nn.Linear(84, 2)

    def forward(self, X):

        X = F.relu(self.conv1(X))

        X = F.max\_pool2d(X, 2, 2)

        X = F.relu(self.conv2(X))

        X = F.max\_pool2d(X, 2, 2)

        X = X.view(-1, 54\*54\*16)

        X = F.relu(self.fc1(X))

        X = F.relu(self.fc2(X))

        X = self.fc3(X)

        return F.log\_softmax(X, dim=1)

**Instantiating loss ,optimizer and model**

torch.manual\_seed(101)

CNNmodel = ConvolutionalNetwork()

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(CNNmodel.parameters(), lr=0.001)

CNNmodel

**Training the model**

import time

start\_time = time.time()

epochs = 2

#limits on number of batches

max\_trn\_batch = 800

max\_tst\_batch = 300

train\_losses = []

test\_losses = []

train\_correct = []

test\_correct = []

for i in range(epochs):

    train\_corr = 0

    test\_corr = 0

    for b,(X\_train,y\_train) in enumerate(train\_loader):

        #optional limit number of batches

        if b == max\_trn\_batch:

            break

        b += 1

        y\_pred = CNNmodel(X\_train)

        loss = criterion(y\_pred,y\_train)

        prediction = torch.max(y\_pred.data,1)[1]

        batch\_corr = (prediction == y\_train).sum()

        train\_corr += batch\_corr

        if b%200 == 0:

            print(f'epoch:{i} batch:{b} loss:{loss.item()} accuracy:{train\_corr.item()\*100/(10\*b)}')

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

    train\_losses.append(loss.item())

    train\_correct.append(train\_corr)

    with torch.no\_grad():

        for b,(X\_test,y\_test) in enumerate(train\_loader):

            #optional limit on number of batches

            if b == max\_tst\_batch:

                break

            b +=1

            y\_eval = CNNmodel(X\_test)

            prediction = torch.max(y\_eval.data,1)[1]

            test\_corr += (prediction == y\_test).sum()

    test\_loss = criterion(y\_eval,y\_test)

    test\_losses.append(loss.item())

    test\_correct.append(test\_corr)

print(f'\nDuration: {time.time() - start\_time:.0f} seconds')

**Using a pretrained model**

* In many cases we make use of inbuilt trained CNNs and only alter certain parameters to suit our needs.
* All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

In our case, let us use the AlexNet model:

AlexNetModel = models.alexnet(pretrained =True)

**It is very important that we do not change the pretrained weights and hence need to make sure there is no backpropagation:**

for param in AlexNetModel.parameters():

    param.requires\_grad = False

Within the alexnet model, we have a classifier whose parameters we will modify based on our requirements:

torch.manual\_seed(42)

AlexNetModel.classifier = nn.Sequential(nn.Linear(9216,1024),

                                        nn.ReLU(),

                                        nn.Dropout(),

                                        nn.Linear(1024,2),

                                        nn.LogSoftmax(dim=1)

                                        )

Setting the loss and optimizer:

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(AlexNetModel.classifier.parameters(),lr=0.001)

**Training the model**

import time

start\_time = time.time()

epochs = 1

#limits on number of batches

max\_trn\_batch = 800

max\_tst\_batch = 300

train\_losses = []

test\_losses = []

train\_correct = []

test\_correct = []

for i in range(epochs):

    train\_corr = 0

    test\_corr = 0

    for b,(X\_train,y\_train) in enumerate(train\_loader):

        #optional limit number of batches

        if b == max\_trn\_batch:

            break

        b += 1

        y\_pred = AlexNetModel(X\_train)

        loss = criterion(y\_pred,y\_train)

        prediction = torch.max(y\_pred.data,1)[1]

        batch\_corr = (prediction == y\_train).sum()

        train\_corr += batch\_corr

        if b%200 == 0:

            print(f'epoch:{i} batch:{b} loss:{loss.item()} accuracy:{train\_corr.item()\*100/(10\*b)}')

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

    train\_losses.append(loss.item())

    train\_correct.append(train\_corr)

    with torch.no\_grad():

        for b,(X\_test,y\_test) in enumerate(train\_loader):

            #optional limit on number of batches

            if b == max\_tst\_batch:

                break

            b +=1

            y\_eval = AlexNetModel(X\_test)

            prediction = torch.max(y\_eval.data,1)[1]

            test\_corr += (prediction == y\_test).sum()

    test\_loss = criterion(y\_eval,y\_test)

    test\_losses.append(loss.item())

    test\_correct.append(test\_corr)

print(f'\nDuration: {time.time() - start\_time:.0f} seconds')

**Note :** The backpropagation in this training will only update teh weights in the classifier part and not in the pretrained section

**Running an image through these models**

x = 2019

im = inv\_normalize(test\_data[x][0])

plt.imshow(np.transpose(im.numpy(), (1, 2, 0)));

# CNN Model Prediction:

CNNmodel.eval()

with torch.no\_grad():

    new\_pred = CNNmodel(test\_data[x][0].view(1,3,224,224)).argmax()

print(f'Predicted value: {new\_pred.item()} {class\_names[new\_pred.item()]}')

# AlexNet Model Prediction:

AlexNetModel.eval()

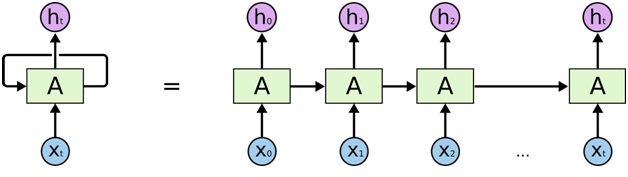
with torch.no\_grad():

    new\_pred = AlexNetModel(test\_data[x][0].view(1,3,224,224)).argmax()

print(f'Predicted value: {new\_pred.item()} {class\_names[new\_pred.item()]}')

**Recurrent Neural Networks**

* Like frontal lobe in the brain. It has short-term memory and uses the knowledge of past few observations to make predictions on the current input.



* The neural networks are connected to themselves through time, which helps them to pass on information and thus serves as a short-term memory.

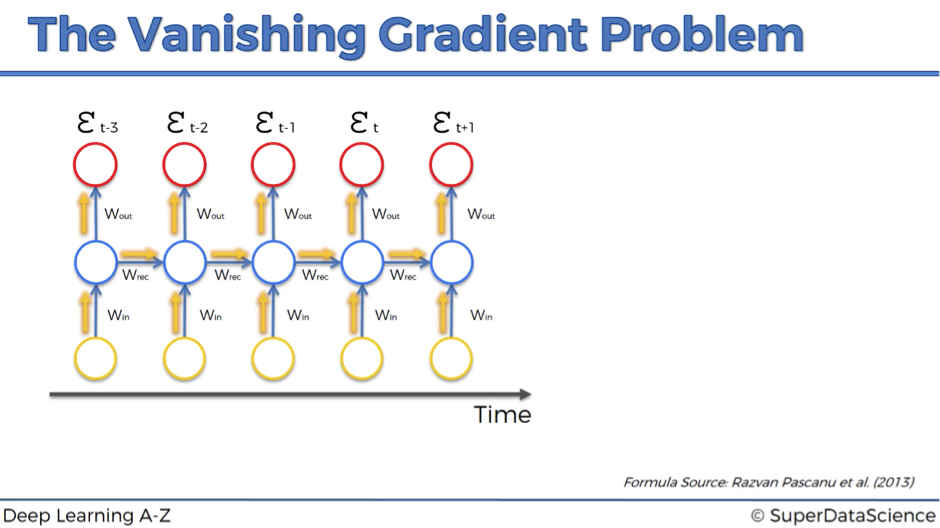
**Additional Material**

* **To check neural networks trained on large datasets:**

[karpathy (Andrej) · GitHub](https://github.com/karpathy)

* **Sunspring –** movie written entirely by an RNN

**Vanishing Gradient**

****

* Usually in backpropagation, the cost function is calculated and the input weights are adjusted based on the this cost function value.
* In RNNs however, it is not just about adjusting the input weights, but to also propagate this to all the neurons back in time that have contributed to this output.
* It can be observed in the diagram that for every propagation, the value is multiplied by Wrec. This is where the problem arises. When you multiply something by a small number, your value decreases very quickly.
* But, when you start with Wrec close to zero and multiply xt, xt-1, xt-2, xt-3, … by this value, your gradient becomes less and less with each multiplication.
* The lower the gradient is, the harder it is for the network to update the weights and the longer it takes to get to the result. As a result, some of the earlier neurons might not get trained properly. But that’s not it.
* The outputs of these earlier neurons/layers are being used as input for further layers. Thus, training at the time point t is happening all along based on inputs that are coming from untrained layers. As a result, the whole network is not being trained properly.

In summary,

*cost function calculated -> propagated back through time -> multiplied by Wrec at each step which diminishes the value -> some earlier layers might be untrained -> these earlier layers are used as inputs for layers ahead hence whole network is not trained properly*

**Solutions to the gradient Problems**

1. **Exploding Gradient (large values of Wrec)**
   * Truncated backpropagation
   * Penalties
   * Gradient Clipping
2. **Vanishing gradient (small values of Wrec)**
   * Weight Initialization
   * Echo state networks
   * Long short-term memory networks (LSTMs)

**Long Short-term Memory networks(LSTMs)**

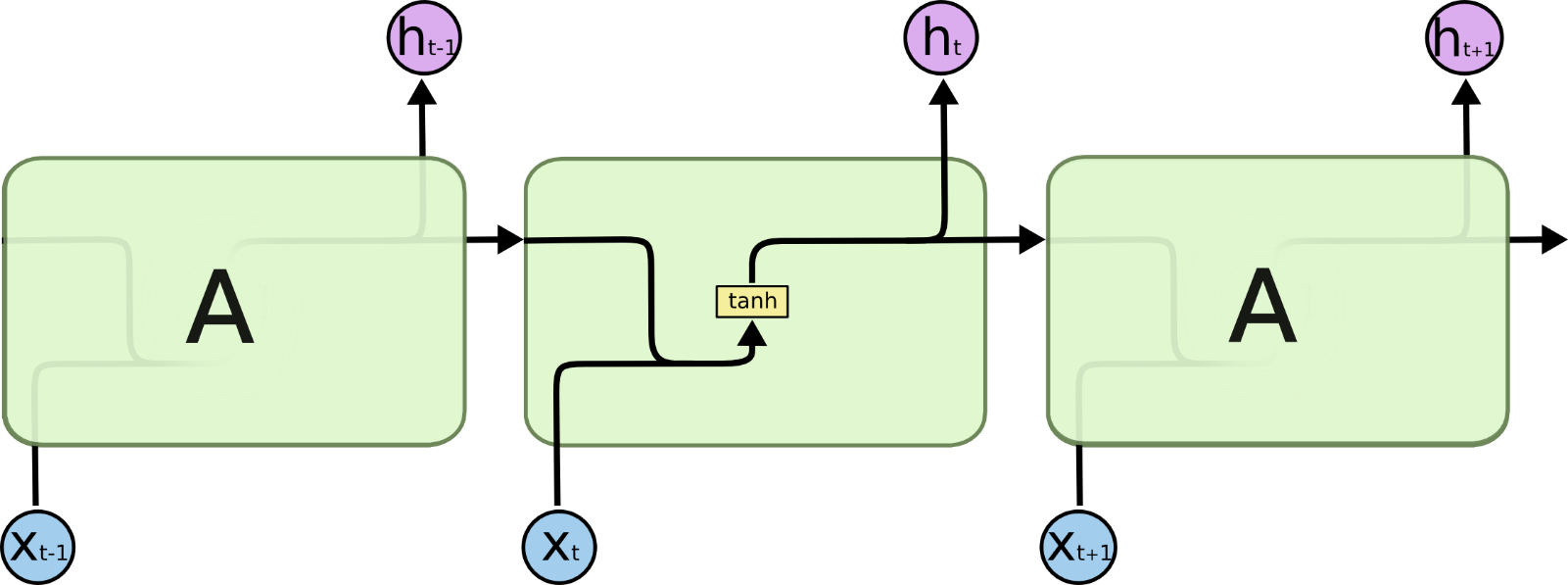
**Wrec <1 : vanishing gradient**

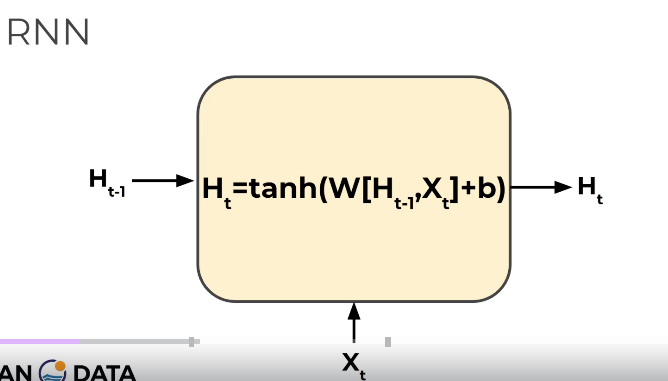
**Wrec >1 : exploding gradient**

* The underlying principle is to set **Wrec = 1**

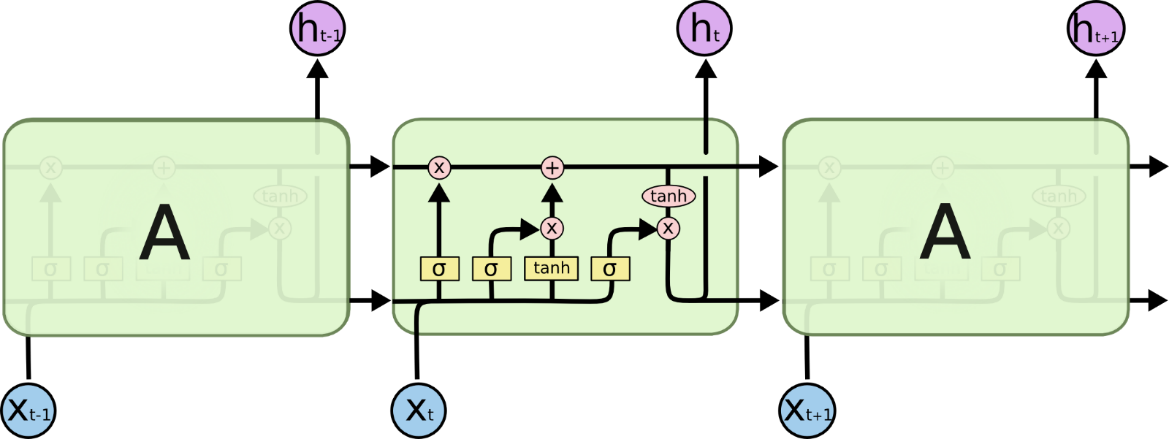
**Recommended reading :** [Understanding LSTM Networks -- colah's blog](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

**Standard RNN**

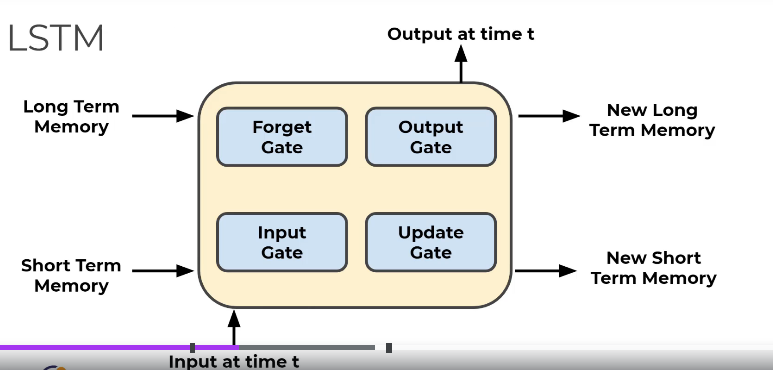
****

****

**LSTM**

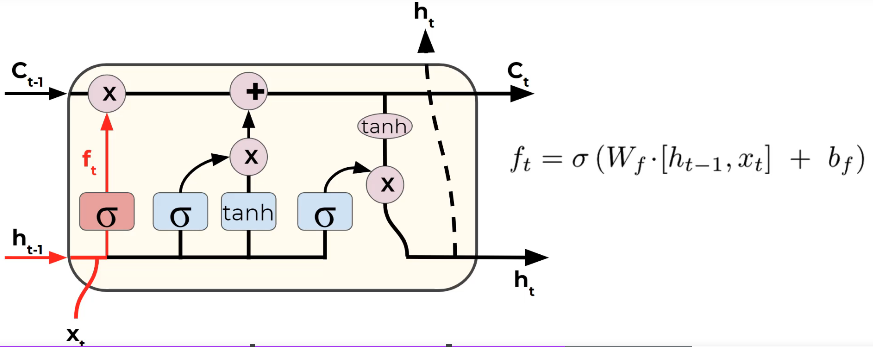
****

* There are 3 stages involved (from the left)
* **Forget gate :** The information that is no longer useful in the cell state is removed with the help of this.
* **Input Gate :** The addition of useful information to the cell state is done at this stage.
* **Output gate :** The task of extracting useful information from the current cell state to be presented as output is done at this stage.

****

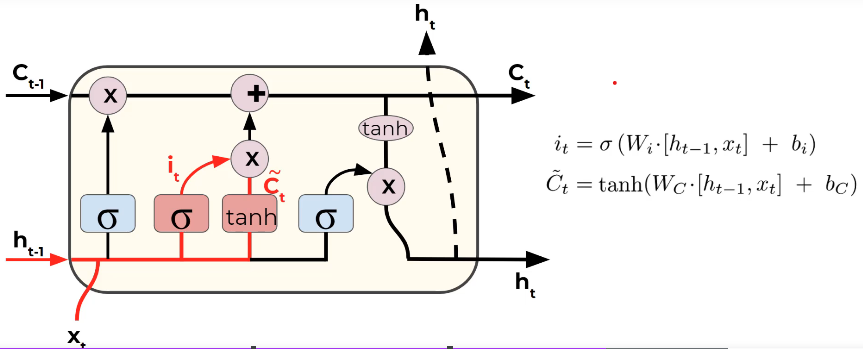
**Breaking down the stages**

**Forget gate**

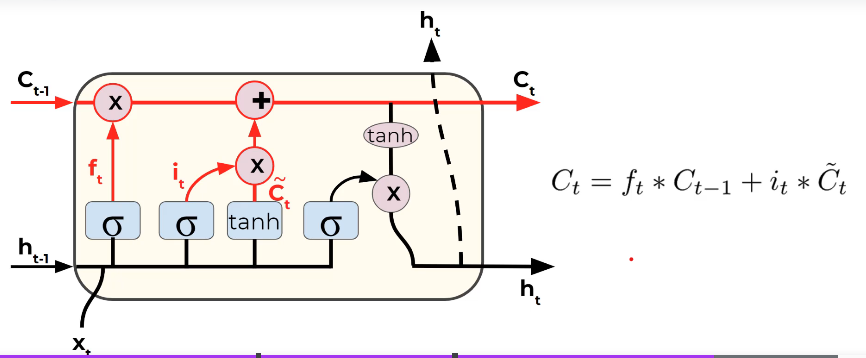


Here we are basically the previous layer inputs and the current inputs through a sigmoid function. the close it is to 1 the higher weightage it gets and the closer it is to 0, higher the probability of it being rejected.

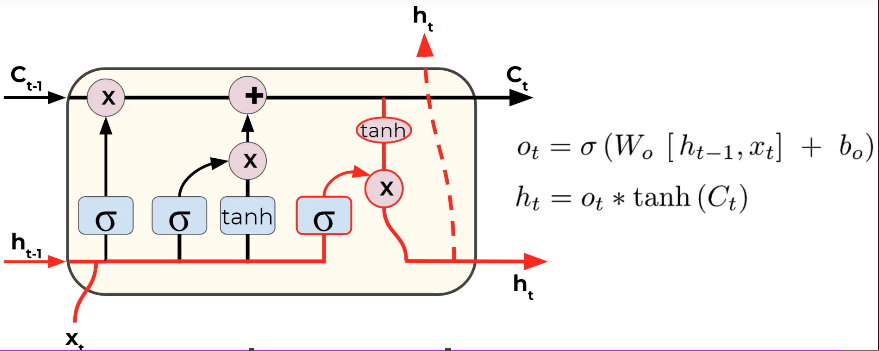
**Determining what information is to be stored**



**Updating the old cell state**

****

**Final output**

****

**GRU – Gated Recurrent Unit**

**Practical Applications**

[The Unreasonable Effectiveness of Recurrent Neural Networks (karpathy.github.io)](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

**Building a RNN**

**Importing the libraries**

import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

**Importing the training set**

df\_train = pd.read\_csv('Google\_Stock\_Price\_Train.csv')

**Note : df.iloc[:,a] vs df.iloc[:,a:a+1]**

**df.iloc[:,1:2]** yields a dataframe whereas **df.iloc[:,1]** yields a series. When we use the .values function, the dataframe is converted to a 2d array(which is the desired input format for neural network) whereas the series yields a 1d array

training\_set = df\_train.iloc[:,1:2].values

**Feature Scaling**

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature\_range=(0,1),copy=True)

scaled\_training\_set = scaler.fit\_transform(training\_set)

Whenever we build an RNN, particularly ones with sigmoid activation function in the output layer, use **normalisation for scaling.**

**Number of Timesteps**

* For a given time step 'N', the RNN will refer to the predictions of the past N instances.
* X\_train holds the previous 60 stock price values at every index, **y\_train** holds the current stock price

X\_train = []

y\_train = []

for i in range(60,len(training\_set)):

    #Getting the 60 previous stock prices

    X\_train.append(scaled\_training\_set[i-60:i,0])

    y\_train.append(scaled\_training\_set[i,0])

We obtain lists which need to be converted to numpy arrays before they can be fed into the rnn model.

X\_train,y\_train = np.array(X\_train),np.array(y\_train)

**Reshaping the data**

* Currently we have one indicator(prices over the last 60 days) to predict the stock prices. But by adding this new dimension, we will be able to add more indicators.
* We give 3 dimensions to the reshape function. (**number\_of\_data\_points**(rows),**number\_of\_timesteps**(columns),**number\_of\_indicators**)

X\_train = np.reshape(X\_train,(X\_train.shape[0],X\_train.shape[1],1))

**Building the RNN**

**Importing the Keras Libraries and packages**

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LSTM

from keras.layers import Dropout

**Initialising the RNN**

regressor = Sequential()

**Adding first LSTM layer and some Dropout regularization(to avoid overfitting)**

regressor.add(LSTM(units = 50, return\_sequences=True,input\_shape =(X\_train.shape[1],1)))

The LSTM function takes in 3 arguments:

* **units** : number of units we need in our LSTM layer. Increasing the number of units will result in high dimensionality which is desired in complex cases
* **return\_sequences** = True. By default this is set to false. We set this to true when we want to add more LSTM layers after the current one
* **input\_shape**: this needs the number of timesteps and the number of indicators as the inputs. This parameter is **needed only for the first LSTM layer.**

regressor.add(Dropout(rate=0.2))

**rate =** what percent of neurons in LSTM must be ignored during training(forward and backward propogation)

**Adding second LSTM layer and some Dropout regularization(to avoid overfitting)**

regressor.add(LSTM(units = 50, return\_sequences=True))

regressor.add(Dropout(rate=0.2))

**Adding third LSTM layer and some Dropout regularization(to avoid overfitting)**

regressor.add(LSTM(units = 50, return\_sequences=True))

regressor.add(Dropout(rate=0.2))

**Adding fourth LSTM layer and some Dropout regularization(to avoid overfitting)**

regressor.add(LSTM(units = 50))

regressor.add(Dropout(rate=0.2))

**Adding the output layer**

regressor.add(Dense(units=1))

**Compiling the RNN**

regressor.compile(optimizer='adam',loss='mean\_squared\_error')

For RNN it is recommended to use the 'RMSProp' optimizer.

**Fitting the RNN to the training set**

regressor.fit(X\_train,y\_train,epochs=100,batch\_size=32)

**Obtaining Predictions from the model**

* Our task here is to predict the stock prices in January 2017. Because of how we have trained the model, we will need the stock price info over the past 60 days to make this prediction.
* One thing is to be noted. We cannot alter the test data in any way. Hence we create this dataframe , obtain our desired inputs and then scale those values.

dataset\_total = pd.concat((df\_train['Open'],df\_test['Open']),axis=0)

**len(dataset\_total)-len(df\_test)-60** : Subtracting the lengths will give us the first stock value from 2017 and '-60' is to obtain the stock prices over the past 60 days.

inputs = dataset\_total[len(dataset\_total)-len(df\_test)-60:].values

inputs = inputs.reshape(-1,1)

inputs = scaler.transform(inputs)

We use reshape to get the right numpy shape We use reshape to get the right numpy shape.

**Next,we need to create the data structure that was fed as input to the network.(Each index has 60 previous values stored).**

X\_test = []

for i in range(60,80): #60+size\_of\_test(20)

    X\_test.append(inputs[i-60:i,0])

X\_test = np.array(X\_test)

**Converting this to a 3D structure:**

X\_test = np.reshape(X\_test,(X\_test.shape[0],X\_test.shape[1],1))

**Obtaining the predictions:**

predicted\_Google\_stock\_price = regressor.predict(X\_test)

We obtain the scaled values as the output . hence, the scaling must be inversed.

predicted\_Google\_stock\_price= scaler.inverse\_transform(predicted\_Google\_stock\_price)

**Visualizing the results**

plt.plot(real\_stock\_price,color='blue',label='Real Google Stock Price')

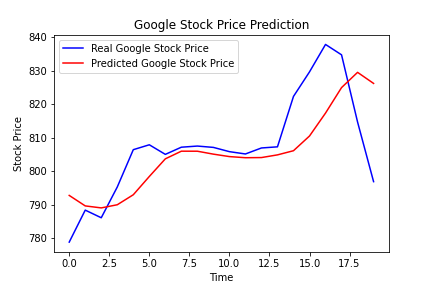
plt.plot(predicted\_Google\_stock\_price,color='red',label='Predicted Google Stock Price')

plt.legend()

plt.title('Google Stock Price Prediction')

plt.xlabel('Time')

plt.ylabel('Stock Price')



**Performance metrics**

import math

from sklearn.metrics import mean\_squared\_error

rmse = math.sqrt(mean\_squared\_error(real\_stock\_price, predicted\_Google\_stock\_price))

**Ways to improve the RNN model:**

* **Getting more training data:** we trained our model on the past 5 years of the Google Stock Price but it would be even better to train it on the past 10 years.
* **Increasing the number of timesteps:** the model remembered the stock prices from the 60 previous financial days to predict the stock price of the next day. That’s because we chose a number of 60 timesteps (3 months). You could try to increase the number of timesteps, by choosing for example 120 timesteps (6 months).
* **Adding some other indicators:** if you have the financial instinct that the stock price of some other companies might be correlated to the one of Google, you could add this other stock price as a new indicator in the training data.
* **Adding more LSTM layers:** we built a RNN with four LSTM layers but you could try with even more.
* **Adding more neurons in the LSTM layers:** we highlighted the fact that we needed a high number of neurons in the LSTM layers to respond better to the complexity of the problem and we chose to include 50 neurones in each of our 4 LSTM layers. You could try an architecture with even more neurones in each of the 4 (or more) LSTM layers.

**Using Pytorch**

**Self Organizing Maps(used for feature detection)**

* In SOMs , there are no activation functions. The weights are the characteristic of the nodes themselves. These weights are assigned at random in the beginning.
* For a given input data sample(one row), find the node that is closest to this data sample. This node is called the **BMU(best matching unit)**
* Once we obtain the BMU, we make it closer in value to the data sample(by updating the weights of that particular node); visually speaking, we pull the map closer to the data sample. Not just the BMU, the nodes within a given radius of the BMU are also pulled towards the data row(their weights are updated).
* This process is repeated for every row in the dataset.
* For every subsequent epoch, the radius of updating the weights of nodes in the neighbourhood of the BMU shrinks.

**Important features:**

* SOMs retain the topology of the input set
* SOMs reveal correlations that are not easily identified
* SOMs classify data without supervision
* They don’t need target vector -> Hence there is no backpropogation to update the weights
* No lateral connections between the output nodes(except for the weight update for nodes within a certain radius of the BMU)

[SOM tutorial part 1 (ai-junkie.com)](http://www.ai-junkie.com/ann/som/som1.html)

**Implementation**

**Importing dataset**

dataset = pd.read\_csv('Credit\_Card\_Applications.csv')

X = dataset.iloc[:,:-1].values

y = dataset.iloc[:,-1].values

Note, we aren't splitting this so that we can treat it as a supervised learning model. We do it so that it would be easier identify the fraudulent transactions in the end.

from sklearn.preprocessing import MinMaxScaler

sc = MinMaxScaler(feature\_range=(0,1))

X = sc.fit\_transform(X)

**Training the SOM**

Since SOM implementation isn’t avaialblke on sklearn we import this independent exceution of the self organizing map

from minisom import MiniSom

som = MiniSom(x=10,y=10,input\_len=15,sigma=1.0,learning\_rate=0.5)

**inputs :**

**x,y** - dimensions of SOM

**input\_len** - number of features in our dataset

**sigma** - radius of neighbourhood

**learning\_rate** - rate at which the weights are updated

* Before we train the model, we need to initialise the weights.

som.random\_weights\_init(X)

som.train\_random(data=X,num\_iteration=100)

**Visualizing and interpreting the output**

**MID(mean inter neuron distance**) is used to identify the outliers. Higher the MID, higher the probability that the winning node is an outlier

from pylab import bone,plot,show,colorbar,pcolor

pcolor(som.distance\_map().T)

colorbar()

First ,we need to add the MIDs for all the winning neurons. Note, the output of the distance\_map() function needs to be **transposed.**

**colorbar() –** Display legends for the colours

markers = ['o','s']

colors= ['r','g']

for i,x in enumerate(X):

    w = som.winner(x)

    plot(w[0]+0.5,

    w[1]+0.5,

    markers[y[i]],

    markeredgecolor=colors[y[i]],

    markerfacecolor='None',

    markersize=10,

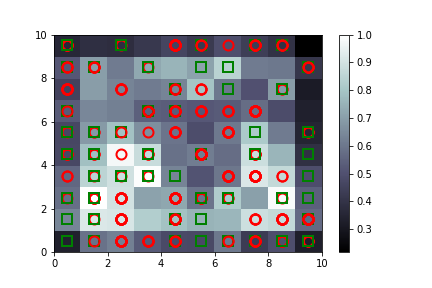
    markeredgewidth=2)

Explaining the code:

i - iterate through total number of customers

x - iterate through each row(each customer)

* We obtain the winning node for each data row . The coordinates of this are passed to the plot function
* The winning nodes are represented as squares.w[0] and w[1] will yield us the **left corner of the square.** Since we need our markers at the centre , we add 0.5 to each coordinate.
* **markers** and **colors** are defined within lists. The earlier separation of the target column will help us fill in the nodes based on the target feature values. (in this case green square – for approved and red circle for potential frauds)



**Interpreting the results(Finding the frauds in this case)**

mappings = som.win\_map(X)

This is a dictionary with the winning node as the key and the associated customers as the values. The coordinates of the winning nodes can also be obtained from the visualization above.

Since we are interested in potential frauds, we can get the coordinates of the outlier winning nodes and concatenate them**. In general, identify the coordinates of winning nodes that belong to our desired target value.**

frauds = np.concatenate((mappings[(4,2)],mappings[(8,2)]),axis=0)

Inverse scaling to obtain the fraudulent customers

frauds = sc.inverse\_transform(frauds)

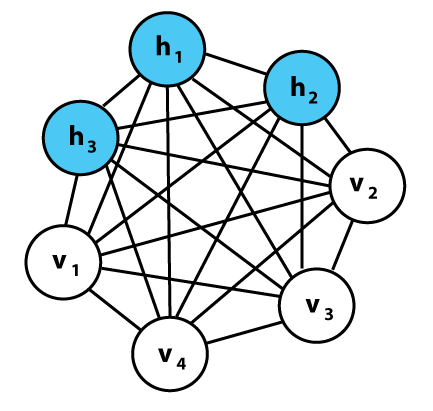
frauds\_df = pd.DataFrame(data=frauds)

frauds\_df.to\_csv('frauds.csv',index=False)

We thus obtain the list of customers whose transactions could potentially be fraudulent.

**Boltzmann Machines**

* The previous 4 models are **directed.** The flow of data/operations happens in a particular direction.
* There are visible nodes and hidden nodes(no output nodes). Important feature of the boltzmann machines is that irrespective of what the inputs are , the model generates parameters on its own.



**Energy-based models**

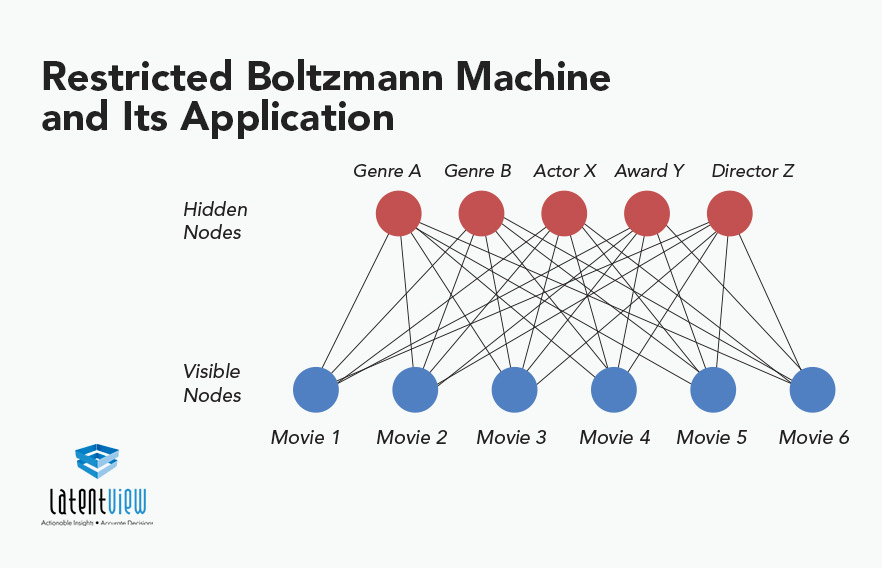
* We make use of the boltzmann’s distribution to obtain the probability of our system being in a particular state. It is noticed that the probability is inversely proportional to the energy in that system.

In simple words, systems are always inclined towards lowest energy states.

* In a boltzmann machine, the energy is described through the weight of the synapses. And the system will try to move towards the lowest energy states by modifying these weights.

**Restricted boltzmann Machines**

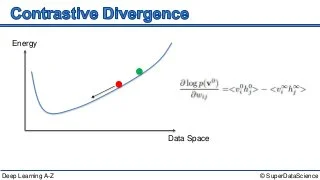
* As the number of nodes grow, it becomes increasingly difficult to implement the system. Hence we go for restricted boltzmann machines.



* Somewhat similar to CNNs, the model identifies important features from the data that is fed to it.
* Primary difference compared to earlier is that there are no intra layer connections(between the visible nodes or between the hidden nodes)
* Imagine we have a dataset for the 6 movies above where users are asked to rate whether they liked it or not. Based on their preferences, the boltzmann machine identifies shared features/hidden connections and assigns the hidden neurons to those features. This in our case could be the genre of the movie or some other category as shown in the image above.

**Contrastive Divergence**

* Because we have an undirected network, we can no longer use gradient descent to adjust the weights.
* The hidden nodes are updated based on the values fed to the input nodes. Next, the input nodes are updated by the hidden nodes(back and forth). Although the weights remain the same, **the original inputs and the reconstructed inputs aren’t the same.**
* This process is called as Gibbs Sampling. This process keeps on repeating until the original input matches the reconstructed input.

****

* As observed in the curve and discussed earlier, the system will move towards the lowest energy state.
* Unlike in gradient descent, we have complete control over the curve. Hence we redesign it in such a way that the system reaches lowest energy state as soon as the training data points are fed to it.
* A quicker way to do is to not way till the curve converges to the lowest value. We wait for 1or 2 passes(back and forth updation of weights) and then adjust the weights in a way that the minimum energy state is achieved.

**Deep belief networks**

* Stacking multiple RBMs/Boltzmann machines over one another.

**Code and Implementation**

**Importing Libraries**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.parallel

import torch.utils.data

from torch.autograd import Variable

**Importing datasets**

movies = pd.read\_csv('D:\\deep\_learning\\neural\_networks\\Boltzmann\_Machines\\ml-1m\\\movies.dat',sep='::',header=None,engine='python',encoding='latin-1')

**Hyperparameters**

* **sep:** since separator in this case isn't a comma
* **encoding :** in order to account for special characters
* **header :** since there is no header in the file
* **engine:** to specify the language in which the parsers are written. Can be either C or python

users = pd.read\_csv('D:\\deep\_learning\\neural\_networks\\Boltzmann\_Machines\\ml-100k\\users.dat',sep='::',header=None,engine='python',encoding='latin-1')

ratings = pd.read\_csv('D:\\deep\_learning\\neural\_networks\\Boltzmann\_Machines\\ml-1m\\ratings.dat',sep='::',header=None,engine='python',encoding='latin-1')

**Creating training and test sets**

We have multiple train-test splits to perform K fold train test splits.

training\_set = pd.read\_csv('D:\\deep\_learning\\neural\_networks\\Boltzmann\_Machines\\ml-100k\\u1.base',delimiter='\t')

Converting dataframe to numpy array:

training\_set = np.array(training\_set,dtype='int')

We are specifying the desired datatype(dtype) of the array elements. Repeating the same for the test set-

test\_set = pd.read\_csv('D:\\deep\_learning\\neural\_networks\\Boltzmann\_Machines\\ml-100k\\u1.test',delimiter='\t')

test\_set = np.array(test\_set,dtype='int')

**Getting the number of users and movies**

To obtain the number of users, we get the maximum value present in the test and the training set.

nb\_users = int(max(max(training\_set[:,0]),max(test\_set[:,0])))

nb\_movies = int(max(max(training\_set[:,1]),max(test\_set[:,1])))

**Creating a matrix of features**

Since we will be using pytorch, instead of creating 2d numpy array, we create a list of lists

* We will have one list per user

def convert(data):

    new\_data = []

    for id\_users in range(1,nb\_users+1):

        #Obtain the movies rated by the user

        id\_movies = data[:,1][data[:,0] == id\_users]

        #Obtain the user ratings

        id\_ratings = data[:,2][data[:,0] == id\_users]

        #For movies the user has not given ratings for, we need to enter a 0

        ratings = np.zeros(nb\_movies)

        ratings[id\_movies-1] = id\_ratings

        #Add this list of one user to the central list

        new\_data.append(ratings)

    return new\_data

training\_set = convert(training\_set)

test\_set =  convert(test\_set)

**Converting data into Pytorch sensors**

training\_set = torch.FloatTensor(training\_set)

**Converting ratings to binary ratings(like or not like)**

Using restricted Boltzmann machines, we will be building recommendation systems that have a binary output(like or not-like). To achieve this, we tune our inputs to a similar format.

* Replace the 0 ratings(movies which weren’t rated by the user but filled randomly) with -1. These are inputs for which we have no ratings from the user and were given 0 rating. But since that will meddle with the training process, we change it to -1.

training\_set[training\_set == 0] = -1

Now we replace the 1 and 2 star reviews with 0.(user didn't like those movies).The reviews with more than 3 stars must be replaced by 1.

training\_set[training\_set == 1] = 0

training\_set[training\_set == 2] = 0

training\_set[training\_set >= 3] = 1

test\_set[test\_set == 1] = 0

test\_set[test\_set == 2] = 0

test\_set[test\_set >= 3] = 1

**Creating the architecture of neural networks**

class RBM():

    def \_\_init\_\_(self,nv,nh):

        self.W = torch.randn(nh,nv)

        self.a = torch.randn(1,nh)

        self.b = torch.randn(1,nv)

    def sample\_h(self,x):

        wx = torch.mm(x,self.W.t()) #take transpose so that the dimensions match

        activation = wx + self.a.expand\_as(wx)

        p\_h\_given\_v = torch.sigmoid(activation)

        return p\_h\_given\_v,torch.bernoulli(p\_h\_given\_v)

    def sample\_v(self,y):

        wy = torch.mm(y,self.W) #no need to take the transpose(why???)

        activation = wy + self.b.expand\_as(wy)

        p\_v\_given\_h = torch.sigmoid(activation)

        return p\_v\_given\_h,torch.bernoulli(p\_v\_given\_h)

    def train(self,v0,vk,ph0,phk):

        self.W += (torch.mm(v0.t(),ph0) - torch.mm(vk.t(),phk)).t()

        self.b += torch.sum((v0-vk),0) #Done to maintain the dimension of b

        self.a += torch.sum((ph0-phk),0)

nv = len(training\_set[0])

nh = 100

batch\_size = 100

rbm = RBM(nv, nh)

**Init function**

* **nv**- number of visible nodes
* **nh** -number of hidden nodes
* **W** - we initiliaze weights in this step and it has to be random. By convention it is matrix of size nv\*nh
* Next , we initialize the bias. This has to be a two-dimensional vector with the first dimension being the batch and the second dimension being the bias
* **a:** bias for hidden nodes
* **b :** bias for visible nodes

**Probability of hidden nodes getting activated given visible nodes( P(h|v))**

* Second function we need involves sampling the nodes given the probability P(H/V). H- hidden node, V - visible node. This is analogous to the forward pass in neural networks.
* This is basically the sigmoid function applied to W\*X + b
* The function will activate hidden nodes based on probabilities it calculates based on the input
* This step is essential since we need to maximize our log likelihood function and gibbs sampling is an integral part of it. Gibbs sampling needs these probabilities
* **x**- visible neurons
* **torch.mm** - used to multiply two tensors
* **expand\_as(wx):** When adding the bias, we need to make sure that the bias gets added to each entry in the wx array. Hence we use expand\_as(wx)
* We pass this to the sigmoid function and obtain the probability of the hidden nodes getting activated given the visible nodes.
* **bernoulli(p\_h\_given\_v)**: What we have is an array/tensor of probability values and we are hoping to predict a binary output(like or didn't like) based on this. For this we need to use the bernoulli distribution**.torch.bernoulli(p\_h\_given\_v)**
* So basically probabilities below a certain threshold will be set to 0 while the rest will be set to 1

**Probability of visible nodes getting activated given hidden nodes (P(v|h))**

* **y** - hidden nodes
* Here, we won't need to take transpose while multiplying the weights(check dimensions)

**Contrastive Divergence**

* Aim is to minimize energy and maximize log likelihood
* We achieve this through Gibbs Sampling. We create a Gibbs Chain of k samples by sampling the hidden and visible nodes k times.
* **v0, vk**- visible nodes obtained after first and kth iteration
* **ph0,phk** - probabilities given v0 and vk

**Initializing the model**

The number of visible nodes will be the number of movies we have in this dataset. The number of hidden nodes however is upto us.

**Training the RBM**

nb\_epoch = 10

for epoch in range(1,nb\_epoch+1):

    train\_loss = 0

    s = 0.

    for id\_user in range(0, nb\_users - batch\_size,batch\_size):

        #The following are the weights that will be updated every epoch

        #We initially set it to the initial non-updated weights v0

        vk = training\_set[id\_user:id\_user+batch\_size]

        #This is the actual rating given by the users. Will be fixed.

        v0 = training\_set[id\_user:id\_user+batch\_size]

        #the sample\_h function returns the probability and the samples. We only need the former

        ph0,\_ = rbm.sample\_h(v0)

        for k in range(10):

            #First step is to sample the hidden nodes based on the visible node values

            \_,hk = rbm.sample\_h(vk)

            #Next, we obtain the updated visible nodes based on the first hidden nodes

            \_,vk = rbm.sample\_v(hk)

            #We have to make sure the entires with -1 ratings(the ones not rated by the users) aren't updated

            vk[v0<0]  = v0[v0<0]

        phk,\_ = rbm.sample\_h(vk)

        rbm.train(v0,vk,ph0,phk)

        #Measuring the loss

        #We make use of the mean absolute distance between the predicted value and the actual value to obtain our test loss

        train\_loss += torch.mean(torch.abs(v0[v0>=0]-vk[v0>=0]))

        #Incrementing the counter

        s += 1.

    #We divide the train\_loss because it will be cumulative(ex 0.2+0.2 over 2 cycles). We will need to average it out

    print('epoch: '+str(epoch)+' loss: '+str(train\_loss/s))

Steps involved in training:

**Measuring test loss**

test\_loss = 0

s = 0.

for id\_user in range(nb\_users):

    #We  don't change it to training set since we will be using these weights to activate the hidden neurons

    v = training\_set[id\_user:id\_user+1]

    vt = test\_set[id\_user:id\_user+1]

    # Now, we won't need K-fold contrastive divergence. Since the model has already been trained 10 times, it will just take 1 pass to update the parameters

    #If loop to check if there are valid ratings. Will be true in most cases

    if len(vt[v>0]) > 0:

        \_,h = rbm.sample\_h(v)

        \_,v = rbm.sample\_v(h)

        test\_loss += torch.mean(torch.abs(vt[vt>=0]-v[vt>=0]))

        s += 1.

print('test loss: '+str(test\_loss/s))

**Autoencoders**

* Aims for the outputs to be identical to the inputs.
* It can be used for feature detection and building recommender systems.
* Inputs are fed to the visible input nodes which are then encoded and fed to the hidden nodes. The output is finally obtained by decoding these values either directly or by using a soft-max function.

**Training an Autoencoder**

* We start with an array where the lines(the observations) correspond to the users and columns(the features) correspond to the movies. Each cell (u,i) contains the rating of the movie from the user u.
* The first user goes into the network. The input vector x contains all it’s ratings for all the movies. (say there are 10 movies. The first input vector will have the ratings given by the user to these two movies).
* The input vector is encoded into a vector z of lower dimensions by a mapping function(like sigmoid function):

z = f(Wx+b) where W is the vector of input weights and b the bias

* z is then decoded into the output vector y of same dimensions as x, aiming to replicate the input vector x.
* The reconstruction error d(x,y) = ||x-y|| is computed. The goal is to minimize it.
* Back-propagation: from right to left, the error is backpropagated. The weights are updated according to how much they are responsible for the error. The learning rate decides by how much we update the weights.
* Repeats the above steps after each observation(reinforcement learning) or only after a batch of observations(Batch learning).

[Building Autoencoders in Keras](https://blog.keras.io/building-autoencoders-in-keras.html)

**Overcomplete Hidden layers :** While we may be tempted to have a hidden layer of a higher dimension of the input layer, this might lead to sub optimal performance in some cases. We use the following solutions to overcome this issue:

1. **Sparse autoencoders:** In this method, a constraint is induced on the hidden layer that at any time, only **some of the hidden nodes** can be used for training **(not all of them)**. This helps prevent overfitting.

[Sparse, Stacked and Variational Autoencoder | by Venkata Krishna Jonnalagadda | Medium](https://medium.com/@venkatakrishna.jonnalagadda/sparse-stacked-and-variational-autoencoder-efe5bfe73b64)

K – sparse autoencoders – Code: [GitHub - snooky23/K-Sparse-AutoEncoder: Sparse Auto Encoder and regular MNIST classification with mini batch's](https://github.com/snooky23/K-Sparse-AutoEncoder)

1. **Denoising Autoencoders:** Here we feed a modified version of the inputs to the autoencoders. For example, some inputs will be replaced randomly by zeroes and the outputs are obtained. The outputs are compared with the original and not the modified ones.
2. **Contractive autoencoders:** A regularization penalty term is added to the backpropagated loss function.
3. **Stacked autoencoders:** An extra hidden layer is added during encoding.
4. **Deep autoencoders:** These are RBMs(restricted Boltzmann machines) stacked on top of each other.

**Code and Implementation**

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.parallel

from torch.autograd import Variable

**Creating test and training sets**

training\_set = pd.read\_csv("C:\\Workspace\\deep\_learning\\neural\_networks\\Autoencoders\\ml-100k\\u1.base",delimiter='\t')

training\_set = np.array(training\_set,dtype='int')

test\_set = pd.read\_csv("C:\\Workspace\\deep\_learning\\neural\_networks\\Autoencoders\\ml-100k\\u1.test",delimiter='\t')

test\_set = np.array(test\_set,dtype='int')

**Getting the total number of movies and users**

nb\_users = int(max(max(training\_set[:,0],),max(test\_set[:,0])))

nb\_movies = int(max(max(training\_set[:,1],),max(test\_set[:,1])))

**Creating the input matrix to the network**

def convert(data):

    new\_data = []

    for user in range(1,nb\_users+1):

        id\_movies = data[:,1][data[:,0] == user]

        id\_ratings = data[:,2][data[:,0] == user]

        ratings = np.zeros(nb\_movies)

        ratings[id\_movies-1] = id\_ratings

        new\_data.append(list(ratings))

    return new\_data

training\_set = convert(training\_set)

test\_set = convert(test\_set)

**Converting this data into torch tensors**

training\_set = torch.FloatTensor(training\_set)

test\_set = torch.FloatTensor(test\_set)

**Creating the architecture of neural network**

class SAE(nn.Module):

    def \_\_init\_\_(self,):

        super(SAE,self).\_\_init\_\_()

        #First encoding layer

        self.fc1 = nn.Linear(nb\_movies,20)

        #Second encoding layer

        self.fc2 = nn.Linear(20,10)

        #Third encoding layer

        self.fc3 = nn.Linear(10,20)

        #Fourth encoding layer

        self.fc4 = nn.Linear(20,nb\_movies)

        #Activation function

        self.activation = nn.Sigmoid()

    def forward(self,x):

        #Obtaining the first encoded vector

        x = self.activation(self.fc1(x))

        #Obtaining the second encoded vector

        x = self.activation(self.fc2(x))

        #Obtaining the first decoded vector

        x = self.activation(self.fc3(x))

        #Obtaining the reconstructed inputs

        x = self.fc4(x)

        return x

**Init function**

* **super(SAE,self)** - It is used so that the Stack auto encoder class(SAE) inherits all the function calls within the parent nn.Module class
* **nn.Linear(neurons in layer 1, neurons in layer 2)**
  + In our case(first encoding layer), neurons in layer 1 will be the number of input features which is number of movies in our case
  + The number of neurons in the hidden layer will be upto us and we can tweak it based on the performance metrics

**Forward function**

* For the reconstructed input, no need to use the activation function. Just the fully connected layer will suffice

**Defining the performance criterion**

sae = SAE()

criterion = nn.MSELoss()

optimizer = optim.RMSprop(sae.parameters(),lr=0.01,weight\_decay=0.5)

* **sae.parameters()** - Obtain all the parameters of autoencoders
* **lr** - learning rate
* **weight\_decay** - reduce learning rate after every few epochs to improve convergence

**Training the SAE**

nb\_epoch = 200

for epoch in range(1,nb\_epoch+1):

    train\_loss = 0

    s = 0.0 #Number of users who rated atleast one movie

    for user in range(nb\_users):

        input = Variable(training\_set[user]).unsqueeze(0)

        target = input.clone()

        #The following conditon checks if there are non-zero values in the row

        #So, the rows with all 0 ratings will be eliminated

        if torch.sum(target.data > 0) > 0:

            output = sae.forward(input)

            target.require\_grad = False

            output[target == 0] = 0

            loss = criterion(output,target)

            #The following factor is calculated so that the training loss is averaged out only over the non-zero rating values

            mean\_corrector = nb\_movies/float(torch.sum(target.data>0)+ 1e-10)

            #This will decide the direction in which the weights will be updated(backward or forward)

            loss.backward()

            #We are taking the square root because the output of the criterion function will be the squared error

            train\_loss += np.sqrt(loss.item() \* mean\_corrector)

            s += 1.0

            #This will decide the amount/intensity by which the weights will be updated

            optimizer.step()

    print('epoch: '+str(epoch)+' loss: '+str(train\_loss/s))

* The variable 's' is tracked so that the code is efficient and we only train with users who have entered valid ratings
* While feeding the input vector to the neural network, it can't be a 1D vector. Like in earlier case, we need to specify the batch as the first dimension
* **target.require\_grad** - When requires\_grad is set to True for a variable, pytorch tracks every operation on it and when you finally use the backward() method for a tensor which came from operation of some variables you can see the derivative of that final variable w.r.t the variables you set requires grad to True. Since we don't need this, we go for require\_grad = False
* **output[target == 0] = 0**
  + We have obtained our output by applying forward pass on our inputs
  + In the following step, we replace the ratings for those fields which were 0 in the original input
  + This can be done because they don't have an effect while calculating the errors.

**Important steps in training**

* Initialising the training loss and ‘s’. In order to make this code more efficient, we will be considering entries which have a non zero ratings(ones which have actually been rated by the users)
* Pytorch inputs cannot be 1D vectors, hence we convert our input vector to a 2D one by adding the batch number as the first dimension.
* We use an if-loop to consider only those data rows which have atleast one non-zero rating. This makes the code more efficient. Now we’ll have rows which have some zero ratings.
* Next we apply the forward pass. Upon its completion, we replace the fields which originally had a zero rating with 0.(example, let us say the movie A had a rating 0 and after the forward pass , it has been decoded as a 1 star rating. We change this back to 0 after the forward pass)
* Next, we use the criterion function to obtain the error. This outputs a squared error term.
* We then define a mean squared error term so that the training loss is averaged out only over the non-zero rating values.
* **loss.backward()**  call helps us determine the direction in which the weights must be updated(forward or backward pass)
* We then update the training loss and finally call the optimiser to decide the amount by which the weights must be updated.

**Measuring the test\_set performance**

s = 0.0

test\_loss = 0

for user in range(nb\_users):

    input = Variable(training\_set[user]).unsqueeze(0)

    target = Variable(test\_set[user]).unsqueeze(0)

    if torch.sum(target.data > 0) > 0:

        output = sae(input)

        target.require\_grad = False

        output[target == 0] == 0

        loss= criterion(output,target)

        mean\_corrector = nb\_movies/float(torch.sum(target.data>0)+1e-10)

        test\_loss += np.sqrt(loss.item()\*mean\_corrector)

        s += 1.0

print('s: ',s)

print('loss: '+str(test\_loss/s))

**Key points**

* While testing the model, the input is still the training set. We need our model to observe the user behaviour in the training set and use that to make predictions on the test set.
* The target however won’t be the clone of the input but the test set values instead.