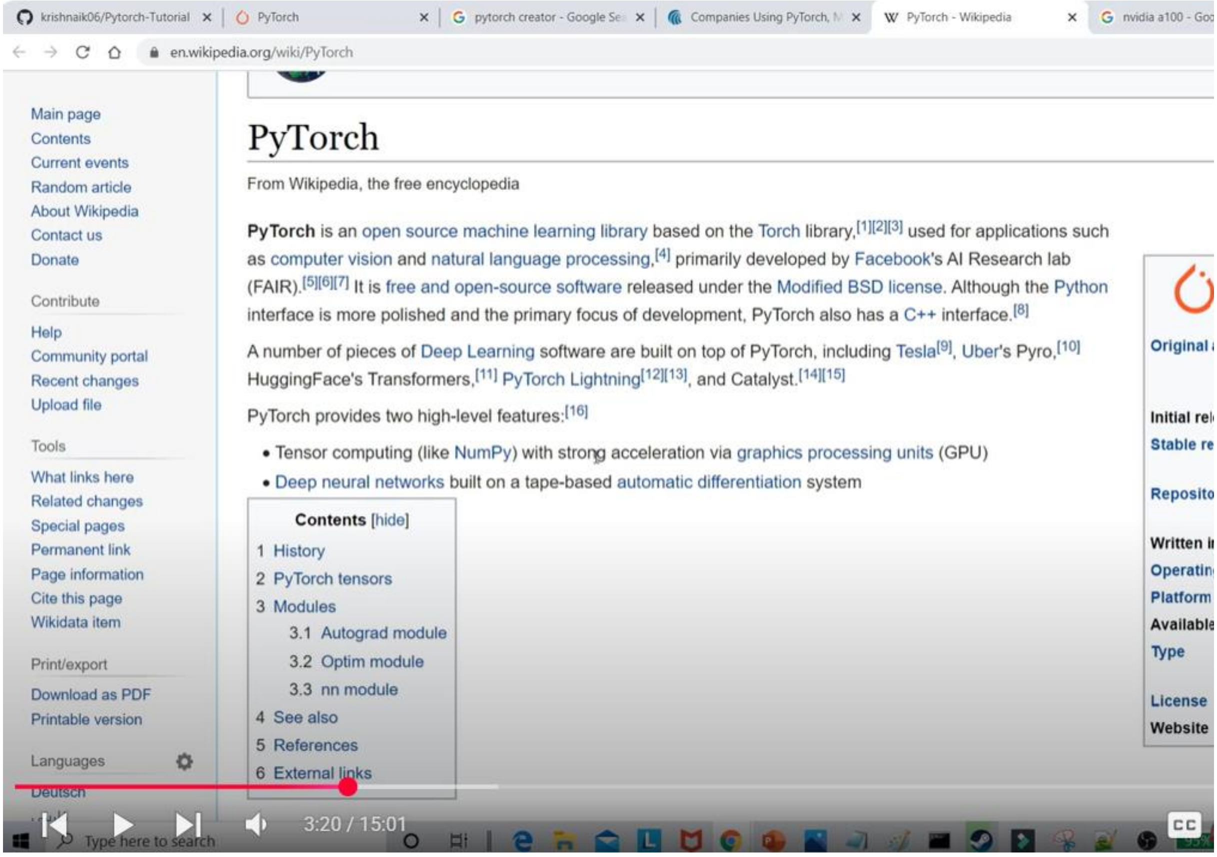
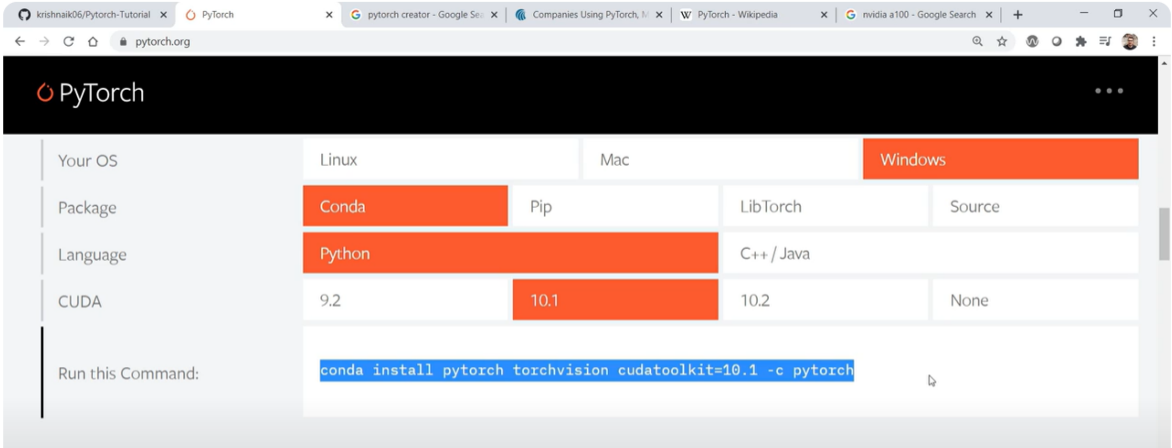

krishnaik06/Pytorch-Tutorial
PyTorch
pytorch creator - Google Se
Companies Using PyTorch, M
PyTorch - Wikipedia
nvidia a100 - Goo
en.wikipedia.org/wiki/PyTorch
Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate
Contribute
Help
Community portal
Recent changes
Upload file
Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item
Print/export
Download as PDF
Printable version
Languages
Deutsch
PyTorch
From Wikipedia, the free encyclopedia
PyTorch is an open source machine learning library based on the Torch library,[1][2][3] used for applications such as computer vision and natural language processing,[4] primarily developed by Facebook's AI Research lab (FAIR).[5][6][7] It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.[8]
A number of pieces of Deep Learning software are built on top of PyTorch, including Tesla[9], Uber's Pyro,[10] HuggingFace's Transformers,[11] PyTorch Lightning[12][13], and Catalyst.[14][15]
PyTorch provides two high-level features:[16]
• Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
• Deep neural networks built on a tape-based automatic differentiation system
Contents [hide]
1 History
2 PyTorch tensors
3 Modules
3.1 Autograd module
3.2 Optim module
3.3 nn module
4 See also
5 References
6 External links
Original
Initial rel
Stable re
Reposito
Written i
Operatin
Platform
Available
Type
License
Website
3:20 / 15:01
Type here to search




```yaml
name: envpytorch
channels:
  - defaults
  - pytorch
dependencies:
  - numpy=1.16.2
  - pandas=0.24.2
  - matplotlib=3.0.3
  - pillow=5.4.1
  - pip=19.0
  - plotly=3.7.0
  - scikit-learn=0.20.3
  - seaborn=0.9.0
  - python=3.7.3
  - jupyter=1.0.0
  - pytorch=1.5.1
  - torchvision=0.2.2
```

Anaconda Prompt (Anaconda3)

```
(base) D:\ML &DL projects\Youtube Videos\Live streaming\Pytorch>dir
 Volume in drive D is Data
 Volume Serial Number is 36B8-25F1

 Directory of D:\ML &DL projects\Youtube Videos\Live streaming\Pytorch

12-07-2020  17:14    <DIR>          .
12-07-2020  17:14    <DIR>          ..
12-07-2020  18:07               305 pytorch_env.yml
               1 File(s)            305 bytes
               2 Dir(s)  364,949,192,704 bytes free

(base) D:\ML &DL projects\Youtube Videos\Live streaming\Pytorch>
```

```
(base) D:\ML &DL projects\Youtube Videos\Live streaming\Pytorch>conda env create -f pytorch_env.yml
```

Conda activate envpytorch

```
(envpytorch) D:\ML &DL projects\Youtube Videos\Live streaming\Pytorch>python
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__version__
'1.5.1'
>>>
```

```
>>> torch.cuda.is_available()
True
```

```
>>> torch.cuda.current_device()
0
```

```
>>> torch.cuda.get_device_name(0)
'GeForce GTX 1650'
>>> _
```

```
>>> torch.cuda.memory_allocated()
0
>>> exit()
```

Pytorch Tutorial

Tensors Basics

A tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array.It is a term and set of techniques known in machine learning in the training and operation of deep learning models can be described in terms of tensors. In many cases tensors are used as a replacement for NumPy to use the power of GPUs.

Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors.

```
In [58]: import numpy as np

In [23]: lst=[3,4,5,6]
         arr=np.array(lst)

In [24]: arr.dtype
Out[24]: dtype('int32')
```

Convert Numpy To Pytorch Tensors

```
In [62]: tensors=torch.from_numpy(arr)
         tensors

Out[62]: tensor([3, 4, 5, 6], dtype=torch.int32)
```

```
In [26]: ### Indexing similar to numpy
         tensors[:2]

Out[26]: tensor([3, 4], dtype=torch.int32)
```

```
In [27]: tensors[1:4]

Out[27]: tensor([4, 5, 6], dtype=torch.int32)
```

Type here to search

14:06
14-07-2020

```
In [64]: ### Indexing similar to numpy
         tensors[2]

Out[64]: tensor(5, dtype=torch.int32)
```

```
In [27]: tensors[1:4]

Out[27]: tensor([4, 5, 6], dtype=torch.int32)
```

```
In [65]: #### Disadvantage of from_numpy. The array and tensor uses the same memory location
         tensors[3]=100
```

```
In [66]: tensors

Out[66]: tensor([  3,   4,   5, 100], dtype=torch.int32)
```

```
In [30]: arr

Out[30]: array([  3,   4,   5, 100])
```

```
In [68]: ### Prevent this by using torch.tensor
         tensor_arr=torch.tensor(arr)
         tensor_arr

Out[68]: tensor([  3,   4,   5, 100], dtype=torch.int32)
```

```
In [69]: tensor_arr[3]=120
         print(tensor_arr)
         print(arr)

         tensor([  3,   4,   5, 120], dtype=torch.int32)
         [  3   4   5 100]
```

```
In [70]: ##zeros and ones
         torch.zeros(2,3,dtype=torch.float64)

Out[70]: tensor([[0., 0., 0.],
                 [0., 0., 0.]], dtype=torch.float64)
```

```
In [37]: torch.ones(2,3,dtype=torch.float64)

Out[37]: tensor([[1., 1., 1.],
                 [1., 1., 1.]], dtype=torch.float64)
```

```
        [1., 1., 1.]], dtype=torch.float64)

n [73]: a=torch.tensor(np.arange(0,15).reshape(5,3))

n [76]: a[:,0:2]

ut[76]: tensor([[ 0,  1],
                [ 3,  4],
                [ 6,  7],
                [ 9, 10],
                [12, 13]], dtype=torch.int32)
```

Arithmetic Operation

```
In [38]: a = torch.tensor([3,4,5], dtype=torch.float)
         b = torch.tensor([4,5,6], dtype=torch.float)
         print(a + b)

         tensor([ 7.,  9., 11.])

In [39]: torch.add(a,b)

Out[39]: tensor([ 7.,  9., 11.])
```

We need to create same shape as in the output variable

```
Out[39]: tensor([ 7.,  9., 11.])

In [41]: c=torch.zeros(3)

In [42]: torch.add(a,b,out=c)

Out[42]: tensor([ 7.,  9., 11.])

In [43]: c

Out[43]: tensor([ 7.,  9., 11.])
```

To summing up all the numbers

```
In [44]: ##### Some more operations
         a = torch.tensor([3,4,5], dtype=torch.float)
         b = torch.tensor([4,5,6], dtype=torch.float)

In [45]: ### tensor[7,9,15]
         torch.add(a,b).sum()

Out[45]: tensor(27.)
```

Dot Products and Mult Operations

```
n [46]: x= torch.tensor([3,4,5], dtype=torch.float)
        y = torch.tensor([4,5,6], dtype=torch.float)

n [47]: x.mul(y)

ut[47]: tensor([12., 20., 30.])
```

Matrix multiplication

```
[51]: x = torch.tensor([[1,4,2],[1,5,5]], dtype=torch.float)
      y = torch.tensor([[5,7],[8,6],[9,11]], dtype=torch.float)

[52]: torch.matmul(x,y)
t[52]: tensor([[55., 53.],
               [90., 92.]])

[54]: torch.mm(x,y)
t[54]: tensor([[55., 53.],
               [90., 92.]])

[55]: x@y
t[55]: tensor([[55., 53.],
               [90., 92.]])
```

```
In [48]: x.dot(y)
Out[48]: tensor(62.)
```

Backpropogation meaning computing derivatives and slope

```
File  Edit  Format  View  Help
Back Propogation----compute derivatives

x^n----->derivative----n*x^n-1

y=x^2
dy/dx=2x
```

If we use requires grad then only we can perform backprogpogation right

```
[2]: x=torch.tensor(4.0,requires_grad=True)

[3]: x
t[3]: tensor(4., requires_grad=True)
```

```
In [4]: y=x**2
        y
Out[4]: tensor(16., grad_fn=<PowBackward0>)
```

Gradient means backpropogation

```
In [4]: y=x**2
        y

Out[4]: tensor(16., grad_fn=<PowBackward0>)

In [5]: #### Back propogation y=2*x
        y.backward()

In [6]: print(x.grad)

        tensor(8.)
```

```
[8]: lst=[[2.,3.,1.],[4.,5.,3.],[7.,6.,4.]]
     torch_input=torch.tensor(lst,requires_grad=True)

[9]: torch_input

t[9]: tensor([[2., 3., 1.],
              [4., 5., 3.],
              [7., 6., 4.]], requires_grad=True)
```

```
In [10]: ### y=x**3+x**2
         y=torch_input**3+torch_input**2

In [11]: y

Out[11]: tensor([[ 12.,  36.,   2.],
                 [ 80., 150.,  36.],
                 [392., 252.,  80.]], grad_fn=<AddBackward0>)

In [ ]:
```

```
In [12]: z=y.sum()

In [13]: z

Out[13]: tensor(1040., grad_fn=<SumBackward0>)
```

```
In [28]: z
Out[28]: tensor(1040., grad_fn=<SumBackward0>)

In [29]: z.backward()

In [30]: torch_input.grad
Out[30]: tensor([[ 16.,  33.,   5.],
                 [ 56.,  85.,  33.],
                 [161., 120.,  56.]])

In [ ]:
```

Building ann using pytorch for kaggle's pima diabetes

```
[1]: import pandas as pd
     df=pd.read_csv('diabetes.csv')
     df.head()
```

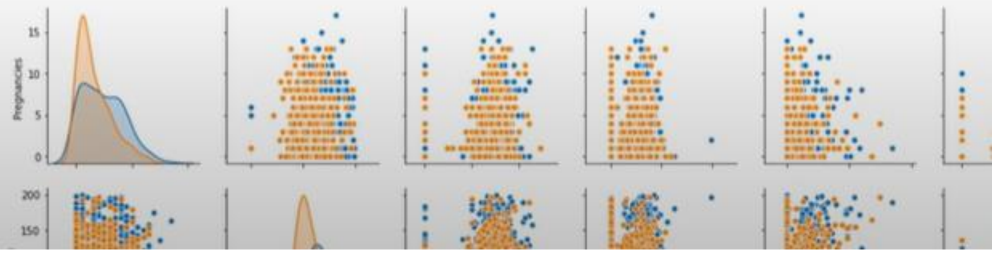
[1]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	

```
In [2]: df.isnull().sum()

Out[2]: Pregnancies                 0
        Glucose                     0
        BloodPressure               0
        SkinThickness               0
        Insulin                     0
        BMI                         0
        DiabetesPedigreeFunction    0
        Age                         0
        Outcome                     0
        dtype: int64
```

```
4]: import seaborn as sns

6]: import numpy as np
    df['Outcome']=np.where(df['Outcome']==1,"Diabetic","No Diabetic")

7]: df.head()
```

In [8]:
```
sns.pairplot(df,hue="Outcome")
```
Out[8]: <seaborn.axisgrid.PairGrid at 0x21f6cfcde80>



```
X=df.drop('Outcome',axis=1).values### independent features
y=df['Outcome'].values###dependent features
```

```
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)
```

In [12]:
```
#### Libraries From Pytorch
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Need to convert the independent feature as float tensors which is very much required

In [13]:
```
##### Creating Tensors
X_train=torch.FloatTensor(X_train)
X_test=torch.FloatTensor(X_test)
y_train=torch.LongTensor(y_train)
y_test=torch.LongTensor(y_test)
```

```
#### Creating Modelwith Pytorch

class ANN_Model(nn.Module):
    def __init__(self,input_features=8,hidden1=20,hidden2=20,out_features=2):
        super().__init__()
        self.f_connected1=nn.Linear(input_features,hidden1)
        self.f_connected2=nn.Linear(hidden1,hidden2)
        self.out=nn.Linear(hidden2,out_features)
    def forward(self,x):
        x=F.relu(self.f_connected1(x))
        x=F.relu(self.f_connected2(x))
        x=self.out(x)
        return x
```

```
In [18]: ####instantiate my ANN_model
         torch.manual_seed(20)
         model=ANN_Model()
```

```
In [19]: model.parameters

Out[19]: <bound method Module.parameters of ANN_Model(
           (f_connected1): Linear(in_features=8, out_features=20, bias=True)
           (f_connected2): Linear(in_features=20, out_features=20, bias=True)
           (out): Linear(in_features=20, out_features=2, bias=True)
         )>
```

To reduce the loss function we use optimizers

Crossentropyloss is used for multiclassification

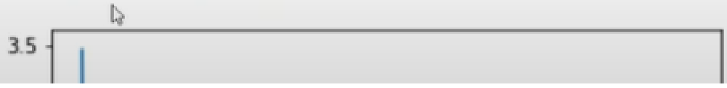
Lr learning rate should be less

```
In [22]: ###Backward Propogation-- Define the loss_function,define the optimizer
         loss_function=nn.CrossEntropyLoss()
         optimizer=torch.optim.Adam(model.parameters(),lr=0.01)
```

```
]: epochs=500
   final_losses=[]
   for i in range(epochs):
       i=i+1
       y_pred=model.forward(X_train)
       loss=loss_function(y_pred,y_train)
       final_losses.append(loss)
       if i%10==1:
           print("Epoch number: {} and the loss : {}".format(i,loss.item()))
       optimizer.zero_grad()
       loss.backward()
       optimizer.step()

   Epoch number: 1 and the loss : 3.457212209701538
```

```
26]: ### plot the loss function
     import matplotlib.pyplot as plt
     %matplotlib inline
```

```
27]: plt.plot(range(epochs),final_losses)
     plt.ylabel('Loss')
     plt.xlabel('Epoch')

27]: Text(0.5, 0, 'Epoch')
```



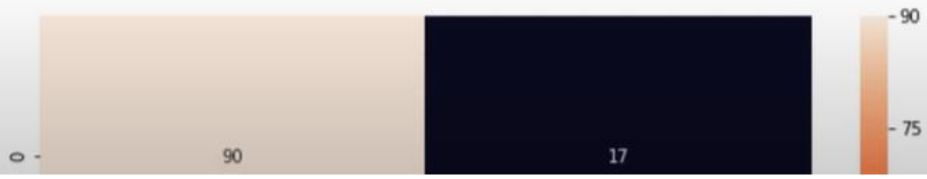
```
[28]: #### Prediction In X_test data
predictions=[]
for i,data in enumerate(X_test):
    print(model(data))
```

```
[29]: #### Prediction In X_test data
predictions=[]
with torch.no_grad():
    for i,data in enumerate(X_test):
        print(model(data))
```

```
[31]: #### Prediction In X_test data
predictions=[]
with torch.no_grad():
    for i,data in enumerate(X_test):
        y_pred=model(data)
        predictions.append(y_pred.argmax().item())
        print(y_pred.argmax().item())
```

```
In [33]: from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,predictions)
cm

Out[33]: array([[90, 17],
       [15, 32]], dtype=int64)
```

```
In [35]: plt.figure(figsize=(10,6))
sns.heatmap(cm,annot=True)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')

Out[35]: Text(69.0, 0.5, 'Predicted Values')
```




```
In [36]: from sklearn.metrics import accuracy_score
score=accuracy_score(y_test,predictions)
score

Out[36]: 0.7922077922077922
```

```
In [38]: #### Save the model
torch.save(model,'diabetes.pt')
```

```
In [39]: model=torch.load('diabetes.pt')

In [40]: model.eval()

Out[40]: ANN_Model(
           (f_connected1): Linear(in_features=8, out_features=20, bias=True)
           (f_connected2): Linear(in_features=20, out_features=20, bias=True)
           (out): Linear(in_features=20, out_features=2, bias=True)
         )

In [ ]:
```

```
In [41]: ### Predcition of new data point
         list(df.iloc[0,:-1])

Out[41]: [6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0]

In [43]: #### New Data
         lst1=[6.0, 130.0, 72.0, 40.0, 0.0, 25.6, 0.627, 45.0]

In [44]: new_data=torch.tensor(lst1)
```

```
46]: #### Predict new data using Pytorch
     with torch.no_grad():
         print(model(new_data))
         print(model(new_data).argmax().item())
```

House pricing prediction

Tutorial 5- Kaggle Advance House Price Prediction Using Pytorch- Tabular Dataset

https://docs.fast.ai/tabular.html https://www.fast.ai/2018/04/29/categorical-embeddings/
https://www.fast.ai/2018/04/29/categorical-embeddings/ https://yashuseth.blog/2018/07/22/pytorch-neural-network-for-tabular-data-with-categorical-embeddings/

1. Category Embedding

```
In [1]: import pandas as pd

In [2]: df=pd.read_csv('houseprice.csv',usecols=["SalePrice", "MSSubClass", "MSZoning", "LotFrontage", "LotArea
                                                  "Street", "YearBuilt", "LotShape", "1stFlrSF", "2ndFlrSF"]).dr

In [3]: df.shape

Out[3]: (1201, 10)
```

1. Theoretical knowledge of Deep Learning
2. ANN(Artificial Neural Network with Pytorch)
3. Feature Engineering {Categorical---Embedding Layer,Continous Variable}
4. pythonic Class to Create Feed Forward Neural Networks
5.

```
199]: df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 1201 entries, 0 to 1459
Data columns (total 10 columns):
MSSubClass      1201 non-null int64
MSZoning        1201 non-null object
LotFrontage     1201 non-null float64
LotArea         1201 non-null int64
Street          1201 non-null object
LotShape        1201 non-null object
YearBuilt       1201 non-null int64
1stFlrSF        1201 non-null int64
2ndFlrSF        1201 non-null int64
SalePrice       1201 non-null int64
dtypes: float64(1), int64(6), object(3)
memory usage: 103.2+ KB
```

Dataset---> Features{Categorical, Continous}

Pytorch ---Tabular Dataset

1. Categorical Features---Embedding Layers
2. Continuous Features

```
memory usage: 103.2+ KB

In [6]: for i in df.columns:
            print("Column name {} and unique values are {}".format(i,len(df[i].unique())))

Column name MSSubClass and unique values are 15
Column name MSZoning and unique values are 5
Column name LotFrontage and unique values are 110
Column name LotArea and unique values are 869
Column name Street and unique values are 2
Column name LotShape and unique values are 4
Column name YearBuilt and unique values are 112
Column name 1stFlrSF and unique values are 678
Column name 2ndFlrSF and unique values are 368
Column name SalePrice and unique values are 597

In [7]: import datetime
```

```
Column name SalePrice and unique values are 597
```

```
In [7]: import datetime
        datetime.datetime.now().year

Out[7]: 2020
```

```
In [8]: df['Total Years']=datetime.datetime.now().year-df['YearBuilt']
```

```
In [9]: df.drop("YearBuilt",axis=1,inplace=True)
```

```
[10]: df.columns

[10]: Index(['MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
             'LotShape', '1stFlrSF', '2ndFlrSF', 'SalePrice', 'Total Years'],
            dtype='object')
```

```
[11]: cat_features=["MSSubClass", "MSZoning", "Street", "LotShape"]
      out_feature="SalePrice"
```

```
In [12]: from sklearn.preprocessing import LabelEncoder
         lbl_encoders={}
         lbl_encoders["MSSubClass"]=LabelEncoder()
         lbl_encoders["MSSubClass"].fit_transform(df["MSSubClass"])

Out[12]: array([5, 0, 5, ..., 6, 0, 0], dtype=int64)
```

```
In [13]: lbl_encoders

Out[13]: {'MSSubClass': LabelEncoder()}
```

```
In [14]: from sklearn.preprocessing import LabelEncoder
         lbl_encoders={}
         for feature in cat_features:
             lbl_encoders[feature]=LabelEncoder()
             df[feature]=lbl_encoders[feature].fit_transform(df[feature])
```

```
1. Categorical Features----
a) Label Encoding
b) take all categorical features---{numpy,torch-->tensors}----> Embedding
Layers
```

Creating vectors and embed them as layers

```
In [31]: ### Stacking and Converting Into Tensors
         cat_features=np.stack([df['MSSubClass'],df['MSZoning'],df['Street'],df['LotShape']],1)
         cat_features

Out[31]: array([[5, 3, 1, 3],
                [0, 3, 1, 3],
                [5, 3, 1, 0],
                ...,
                [6, 3, 1, 3],
                [0, 3, 1, 3],
                [0, 3, 1, 3]], dtype=int64)
```

Don't convert categorical features to float

```
In [32]: ### Convert numpy to Tensors
         import torch
         cat_features=torch.tensor(cat_features,dtype=torch.int64)
         cat_features

Out[32]: tensor([[5, 3, 1, 3],
                 [0, 3, 1, 3],
                 [5, 3, 1, 0],
                 ...,
                 [6, 3, 1, 3],
                 [0, 3, 1, 3],
                 [0, 3, 1, 3]])
```

```
In [42]: #### create continuous variable
         cont_features=[]
         for i in df.columns:
             if i in ["MSSubClass", "MSZoning", "Street", "LotShape","SalePrice"]:
                 pass
             else:
                 cont_features.append(i)
```

```
[214]: cont_features

t[214]: ['LotFrontage', 'LotArea', '1stFlrSF', '2ndFlrSF', 'Total Years']
```

c) Lets take all the continuous values
d) Continuous--Numpy--Torch--->tensors

```
5]: ### Stacking continuous variable to a tensor
    cont_values=np.stack([df[i].values for i in cont_features],axis=1)
    cont_values=torch.tensor(cont_values,dtype=torch.float)
    cont_values

5]: tensor([[   65.,  8450.,   856.,   854.,    17.],
            [   80.,  9600.,  1262.,     0.,    44.],
            [   68., 11250.,   920.,   866.,    19.],
            ...,
            [   66.,  9042.,  1188.,  1152.,    79.],
            [   68.,  9717.,  1078.,     0.,    70.],
            [   75.,  9937.,  1256.,     0.,    55.]])

5]: cont_values.dtype

5]: torch.float32
```

Always do reshape to have two dimensional thing

```
Out[45]: torch.float32

In [47]: ### Dependent Feature
         y=torch.tensor(df['SalePrice'].values,dtype=torch.float).reshape(-1,1)
         y

Out[47]: tensor([[208500.],
                 [181500.],
```

```
In [218]: cat_features.shape,cont_values.shape,y.shape

Out[218]: (torch.Size([1201, 4]), torch.Size([1201, 5]), torch.Size([1201, 1]))

In [54]: len(df['MSSubClass'].unique())

Out[54]: 15

In [63]: #### Embedding Size For Categorical columns
         cat_dims=[len(df[col].unique()) for col in ["MSSubClass", "MSZoning", "Street", "LotShape"]]

In [64]: cat_dims

Out[64]: [15, 5, 2, 4]
```

```
1. Categorical Features----
a) Label Encoding---Done
b) take all categorical features---{numpy,torch-->tensors}---Done
c) Lets take all the continuous values---> Done
d) Continuous--Numpy--Torch--->tensors---> Done
e) Embedding Layers---Categorical Features
```

Embedding Size For Categorical columns

```
In [222]: len(df["Street"].unique())

Out[222]: 2

In [63]:
          cat_dims=[len(df[col].unique()) for col in ["MSSubClass", "MSZoning", "Street", "LotShape"]]

In [64]: cat_dims

Out[64]: [15, 5, 2, 4]
```

Why we need the dimensions – embedding will decide number of inputs and output based on number of length

```
65]: ### Thumbs Rule Output dimension should be setbased on the input dimension(min(50,feature dimension/2))
     bedding_dim= [(x, min(50, (x + 1) // 2)) for x in cat_dims]

66]: embedding_dim

66]: [(15, 8), (5, 3), (2, 1), (4, 2)]
```

In neural networks this is the first step

Why we are using Module list because it can have more embedding layers

```
1]: import torch
    import torch.nn as nn
    import torch.nn.functional as F
    embed_representation=nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
    embed_representation

1]: ModuleList(
```

```
n [138]: cat_featuresz=cat_features[:4]
         cat_featuresz

ut[138]: tensor([[5, 3, 1, 3],
                 [0, 3, 1, 3],
                 [5, 3, 1, 0],
                 [6, 3, 1, 0]])
```

```
n [141]: pd.set_option('display.max_rows', 500)
         embedding_val=[]
         for i,e in enumerate(embed_representation):
             embedding_val.append(e(cat_features[:,i]))
```

```
         ...,
         [-0.7072, -0.1270,  1.0319,  ..., -0.2547, -0.3440, -0.4935],
         [-1.9159,  0.6082,  1.0183,  ..., -0.4991, -0.5187, -0.9944],
         [-1.9159,  0.6082,  1.0183,  ..., -0.4991, -0.5187, -0.9944]],
        grad_fn=<EmbeddingBackward>),
 tensor([[ 0.4657,  1.8871, -0.2028],
         [ 0.4657,  1.8871, -0.2028],
         [ 0.4657,  1.8871, -0.2028],
         ...,
         [ 0.4657,  1.8871, -0.2028],
         [ 0.4657,  1.8871, -0.2028],
         [ 0.4657,  1.8871, -0.2028]], grad_fn=<EmbeddingBackward>),
 tensor([[-0.5360],
         [-0.5360],
         [-0.5360],
         ...,
         [-0.5360],
         [-0.5360],
         [-0.5360]], grad_fn=<EmbeddingBackward>),
 tensor([[ 0.8112,  0.1669],
         [ 0.8112,  0.1669],
         [-0.7741,  0.2652],
         ...,
```

```
n [144]: z = torch.cat(embedding_val, 1)
         z

ut[144]: tensor([[ 0.4979, -0.6349, -0.5640,  ...,  0.9383, -1.3483, -0.4345],
                 [-1.6998,  0.8508, -1.2298,  ...,  0.9383, -1.3483, -0.4345],
                 [ 0.4979, -0.6349, -0.5640,  ...,  0.9383,  0.9474, -1.1973],
                 ...,
                 [-0.9010,  0.7976,  0.3026,  ...,  0.9383, -1.3483, -0.4345],
                 [-1.6998,  0.8508, -1.2298,  ...,  0.9383, -1.3483, -0.4345],
                 [-1.6998,  0.8508, -1.2298,  ...,  0.9383, -1.3483, -0.4345]],
                grad_fn=<CatBackward>)

n [146]: #### Implement dropupout
```

Stack lot of all values used to numpy arrays and torch has functionalities concat

```
n [239]: #### Implement dropupout
         droput=nn.Dropout(.4)
```

Dropout layers will help us to prevent from overfitting

Dropout is one of the regularization methods

```
In [240]: #### Implement dropupout
          droput=nn.Dropout(.4)

In [241]: final_embed=droput(z)
          final_embed

Out[241]: tensor([[ 2.7822, -0.2741,  0.5509,  ..., -0.8933,  0.0000,  0.2781],
                  [-0.0000,  0.0000,  0.0000,  ..., -0.0000,  0.0000,  0.2781],
                  [ 0.0000, -0.2741,  0.0000,  ..., -0.0000, -1.2902,  0.4421],
                  ...,
                  [-0.0000, -0.2116,  1.7198,  ..., -0.8933,  0.0000,  0.2781],
                  [-0.0000,  1.0137,  1.6971,  ..., -0.0000,  0.0000,  0.2781],
                  [-3.1931,  0.0000,  1.6971,  ..., -0.8933,  1.3521,  0.2781]],
                 grad_fn=<MulBackward0>)
```

P is the dropout ratio

Here we need to add layers = [100,50]

There will be 100 neurons in hidden layer 1, 50 neurons in hidden layer 2

```python
class FeedForwardNN(nn.Module):

    def __init__(self, embedding_dim, n_cont, out_sz, layers, p=0.5):
        super().__init__()
        self.embeds = nn.ModuleList([nn.Embedding(inp,out) for inp,out in embedding_dim])
        self.emb_drop = nn.Dropout(p)
        self.bn_cont = nn.BatchNorm1d(n_cont)

        layerlist = []
        n_emb = sum((out for inp,out in embedding_dim))
        n_in = n_emb + n_cont

        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))

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

```python
    def forward(self, x_cat, x_cont):
        embeddings = []
        for i,e in enumerate(self.embeds):
            embeddings.append(e(x_cat[:,i]))
        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
```

```python
len(cont_features)
```

```
5
```

In [123]:
```python
torch.manual_seed(100)
model=FeedForwardNN(embedding_dim,len(cont_features),1,[100,50],p=0.4)
```

Define Loss And Optimizer

In [125]:
```python
loss_function=nn.MSELoss() ###Later convert to RMSE
optimizer=torch.optim.Adam(model.parameters(),lr=0.01)
```

In [126]:
```python
df.shape
```

Out[126]: (1201, 10)

```
249]: cont_values.shape
249]: torch.Size([1201, 5])
```

```
[249]: cont_values.shape
t[249]: torch.Size([1201, 5])
```

```
[250]: 1200*0.15
t[250]: 180.0
```

```
[128]: batch_size=1200
       test_size=int(batch_size*0.15)
       train_categorical=cat_features[:batch_size-test_size]
       test_categorical=cat_features[batch_size-test_size:batch_size]
       train_cont=cont_values[:batch_size-test_size]
       test_cont=cont_values[batch_size-test_size:batch_size]
       y_train=y[:batch_size-test_size]
       y_test=y[batch_size-test_size:batch_size]
```

```
244]: FeedForwardNN(
        (embeds): ModuleList(
          (0): Embedding(15, 8)
          (1): Embedding(5, 3)
          (2): Embedding(2, 1)
          (3): Embedding(4, 2)
        )
        (emb_drop): Dropout(p=0.4, inplace=False)
        (bn_cont): BatchNorm1d(5, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (layers): Sequential(
          (0): Linear(in_features=19, out_features=100, bias=True)
          (1): ReLU(inplace=True)
          (2): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (3): Dropout(p=0.4, inplace=False)
          (4): Linear(in_features=100, out_features=50, bias=True)
          (5): ReLU(inplace=True)
          (6): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (7): Dropout(p=0.4, inplace=False)
          (8): Linear(in_features=50, out_features=1, bias=True)
        )
```

Zero grad – resets the optimizers

```
epochs=5000
final_losses=[]
for i in range(epochs):
    i=i+1
    y_pred=model(train_categorical,train_cont)
    loss=torch.sqrt(loss_function(y_pred,y_train)) ### RMSE
    final_losses.append(loss)
    if i%10==1:
        print("Epoch number: {} and the loss : {}".format(i,loss.item()))
    optimizer.zero_grad()
    loss.backward()###backpropogation
    optimizer.step()
```

```
Epoch number: 3631 and the loss : 63506.36328125
```

[148]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(range(epochs), final_losses)
plt.ylabel('RMSE Loss')
plt.xlabel('epoch');
```

In [150]:
```
#### Validate the Test Data
y_pred=""
with torch.no_grad():
    y_pred=model(test_categorical,test_cont)
    loss=torch.sqrt(loss_function(y_pred,y_test))
print('RMSE: {}'.format(loss))
```

```
RMSE: 48271.9453125
```

In [178]:
```
data_verify=pd.DataFrame(y_test.tolist(),columns=["Test"])
```

In [179]:
```
data_predicted=pd.DataFrame(y_pred.tolist(),columns=["Prediction"])
```

In [180]:
```
data_predicted
```

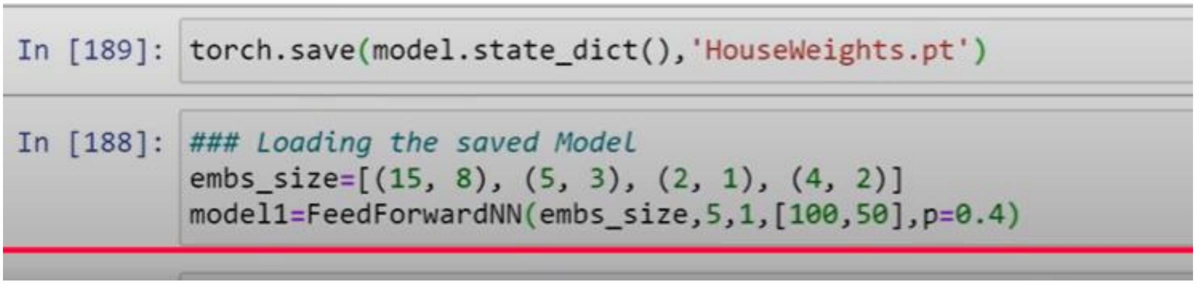
In [259]:
```
final_output=pd.concat([data_verify,data_predicted],axis=1)
final_output['Difference']=final_output['Test']-final_output['Prediction']
final_output.head()
```

Out[259]:

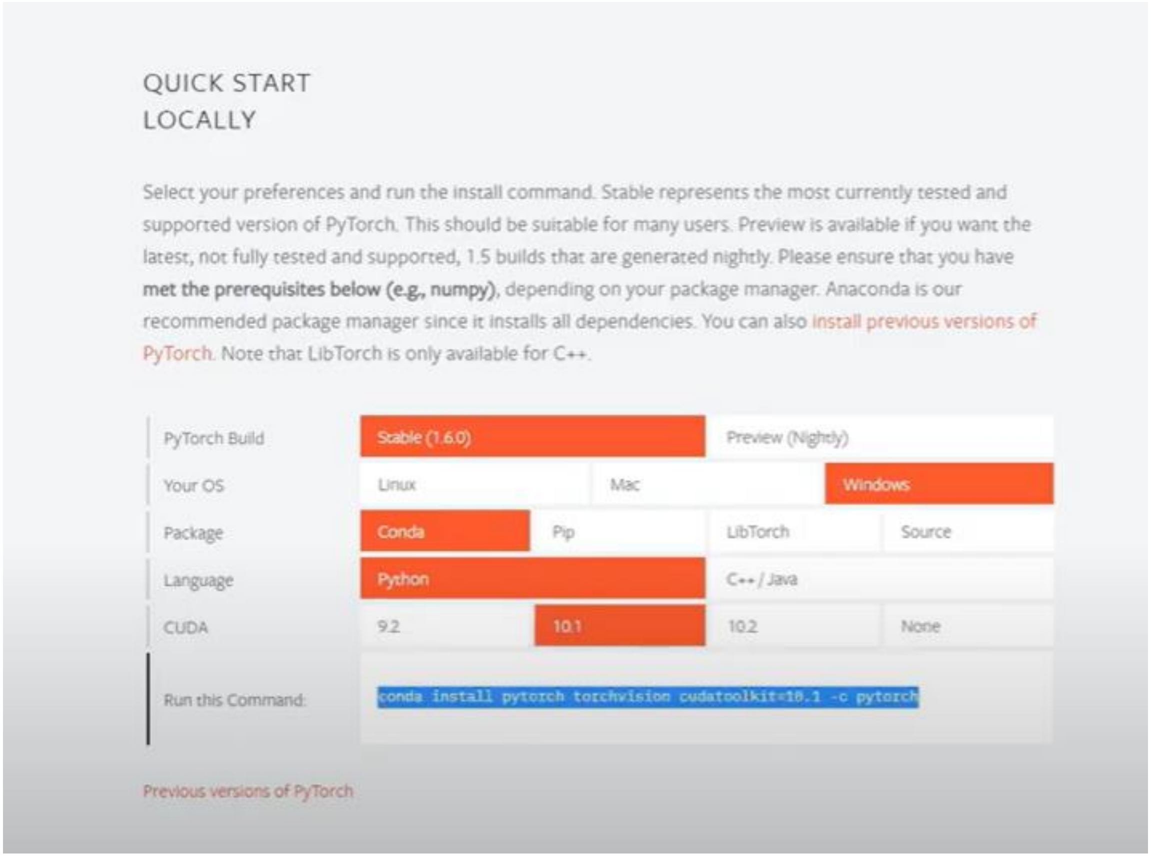
	Test	Prediction	Difference
0	130000.0	158548.109375	-28548.109375
1	138887.0	203009.031250	-64122.031250
2	175500.0	138875.734375	36624.265625
3	195000.0	226541.218750	-31541.218750
4	142500.0	208889.640625	-66389.640625

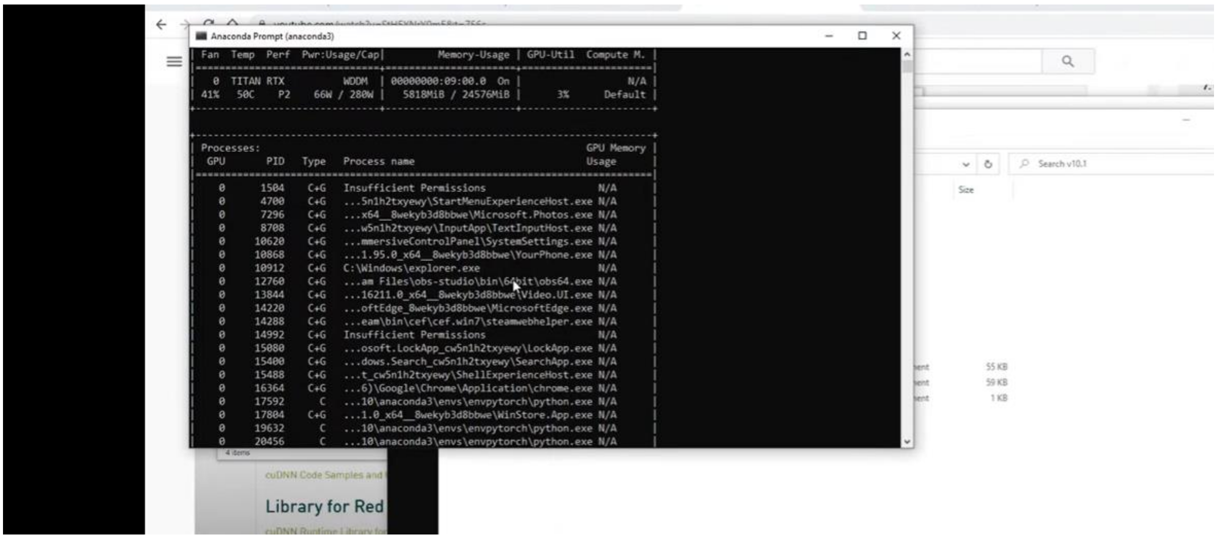
[262]:
```
#### Saving The Model
#### Save the model
torch.save(model,'HousePrice.pt')
```

State_dict – will save the weights



1. How To Run Pytorch Code In GPU Using CUDA Library





```
Anaconda Prompt (anaconda3)

(base) C:\Users\win10>activate envpytorch

(envpytorch) C:\Users\win10>conda install pytorch torchvision cudatoolkit=10.1 -c pytorch
```

Running ANN using GPU

```
In [87]: import torch

In [4]: torch.cuda.is_available()

Out[4]: True

In [5]: torch.cuda.current_device()

Out[5]: 0
```

```
In [7]: torch.cuda.get_device_name(0)

Out[7]: 'TITAN RTX'

In [16]: torch.cuda.memory_allocated()

Out[16]: 1024
```

```
In [6]: torch.cuda.memory_cached()

C:\Users\win10\anaconda3\envs\envpytorch\
y:346: FutureWarning: torch.cuda.memory_c
emory_reserved
  FutureWarning)

Out[6]: 0
```

By default memory will allocated to cpu

```
In [8]: var1=torch.FloatTensor([1.0,2.0,3.0])

In [9]: var1
Out[9]: tensor([1., 2., 3.])

In [15]: var1.device
Out[15]: device(type='cuda', index=0)
```

If we want to use gpu then use below commands

```
In [14]: var1=torch.FloatTensor([1.0,2.0,3.0]).cuda()

In [15]: var1
Out[15]: tensor([1., 2., 3.], device='cuda:0')
```

```
In [16]: var1.device
Out[16]: device(type='cuda', index=0)
```

Use below functions to run the models on GPU called CUDA

```
In [20]: #### Libraries From Pytorch
         import torch
         import torch.nn as nn
         import torch.nn.functional as F

In [68]: ##### Creating Tensors
         X_train=torch.FloatTensor(X_train).cuda()
         X_test=torch.FloatTensor(X_test).cuda()
         y_train=torch.LongTensor(y_train).cuda()
         y_test=torch.LongTensor(y_test).cuda()

In [14]: df.shape
```

```
In [28]: X_train.device

Out[28]: device(type='cpu')

In [14]: df.shape

Out[14]: (768, 9)
```

```
In [31]: model.parameters

Out[31]: <bound method Module.parameters of ANN_Model(
           (f_connected1): Linear(in_features=8, out_features=20, bias=True)
           (f_connected2): Linear(in_features=20, out_features=20, bias=True)
           (out): Linear(in_features=20, out_features=2, bias=True)
         )>

In [35]: for i in model.parameters():
             print(i.is_cuda)

         False
         False
         False
         False
         False
         False
```

To run the model in gpu use the following code

```
In [77]: model=model.cuda()
```