بسم الله الرحمن الرحیم

پروژه ی Bitcoin Price Prediction

درس هوش محاسباتی

نام و نام خانوادگی:هیلا حیدری

شماره دانشجویی:۹۵۰۱۲۲۶۸۰۰۸۱

گزارش فعالیت ها:

ابتدا با زبان پایتون و سپس با jupyter notebook آشنا شدیم.

و در مرحله بعد به جستجو در اینتنت و جمع آوری اطالعات پرداختیم.

Deep learning یکی از مباحث در هوش مصنوعی است در تعریف کلی یادگیری عمیق همان یاد گیری ماشین است بطوری که در سطوح مختلف نمایش یا انتزاع (abstraction) یادگیری را برای ماشین انجام می دهد که با این کار ماشین میتواند درک بهتری از واقعیت های موجود پیدا کرده و میتواند الگو ها ی مختلف را شناسایی کند.

برای شناسای یادگیری عمیق، ابتدا نیاز به دانستن شبکه های عصبی دارید. بر اساس تعریف مشهور، یادگیری عمیق در واقع همان یادگیری به وسیله شبکه های عصبی ای هستند که دارای لایه های پنهانی (Hidden Layers) زیادی میباشند. هر چقدر در لایه های یک شبکه های عصبی عمیق جلو تر میرویم به مدل های پیچیده تر و کامل تری میرسیم.

یادگیری عمیق در حوزه های مختلفی مورد استفاده قرار میگیرد از جمله پیش بینی قیمت بورس و بیت کوین و...

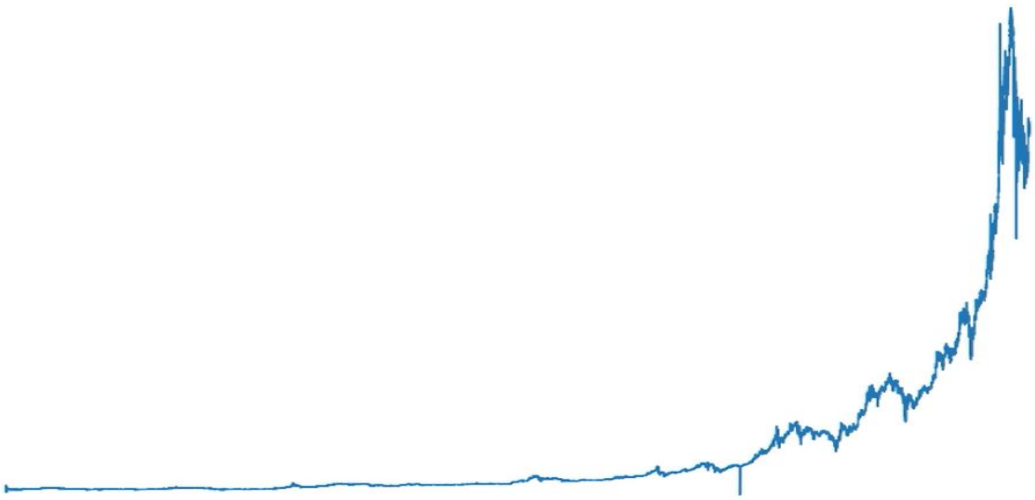
توضیحات پروژه

قرار است data را خودمان پیدا کنیم . از اکانت gitup پیدا کردم که مال خود لینک دانلودش هست و این خوش از یک وبسایتی میگیره که فایل ها را توی ۲ تا ۳ سال آپدیت میکنه تقریبا نزدیک به یک میلیون و دویست هزار بار قیمت هارو ذخیره کرده با ویژگی های مختلفشون پس dataرا اینجوری دانلود کردم

بحث libraryها:

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3   import pandas as pd
4
5   import os
6
7   from sklearn.preprocessing import StandardScaler, MinMaxScale
8   from sklearn.metrics import mean_squared_error, r2_score
9
10  import torch
11  import torch.nn as nn
12  import torch.nn.functional as F
```

Matplotlib.pyplot:اولی فقط برای کشیدن نقشه ها و نمودار بکار میره چون قراره یک نمودار بکشیم مثل این نمودار که کل قیمت هاست و ما قراره یک قسمتی از اون را پیش بینی کنیم



Numpy:برای محاسبات عددی پایتون است مثلا زمانی که data را میخوانیم بصورت text هستند باید آن را تبدبل کنیم به عدد و عدد ها را scale کنیم ببریم توی رنج ۰ و ۱ همچنین امکان ایجاد آرایه و ماتریس و... را به ما میدهد

Pandas:یک کتابخونه پایتون است که برای خواندن فایل های csv است خیلی متد ها دارد که کمک کننده هستند و به صورت یک جدول تمییز برای ما درست میکنه کل عدد هارو

برای انجام تجذیه تحلیل داده ها بکار گرفته میشود و این کتابخانه برای نگه داشتن فایل ها و داده ها بصورت سطر و ستون هستند کاربرد دارد

این ها مربوط به pandasهستند

```
[ ]  1  dataframe = pd.read_csv("BTC_price_history.csv",
     2                          names=["Timestamp", "Low", "High", "Open", "Close", "Volume", "WeightedPrice"])
     3  dataframe = dataframe.drop(columns=['Timestamp'])
     4  dataframe.head(5)
```

```
[ ]  1  plt.figure(figsize=(15, 10))
     2  dataframe.WeightedPrice.plot()
```

Os:برای خواندن فایل در فولدر استفاده میشود

Sklearn.preprocessing:برای scale کردن استفاده میشود در اینجا از standarscalerاستفاده نکردیم

Sklearn.metrics:در اینجا از r2 استفاده نکردیم چون جواب نمیداد

Torch:کل کاری که ما انجام میدهیم چونکه با deep learning بوده library های زیادی داریم ولی ما از torch استفاده میکنیم مثلا حتی توی average گرفتن هم از این استفاده میکنیم

Torch.nn:کل نیورال نتورک ها توی این مثلا linear که رابطه ی خطی میسازد از توی این ساخته میشود

Torch.nn.functional:این هم یکسری متد ها دارد .متد هایی هستند که محاسبات عادی دارند مثل max گرفتن ولی مخصوص شبکه های عصبی ساخته شده که سریع تر کار کنند

Generate_dataset:

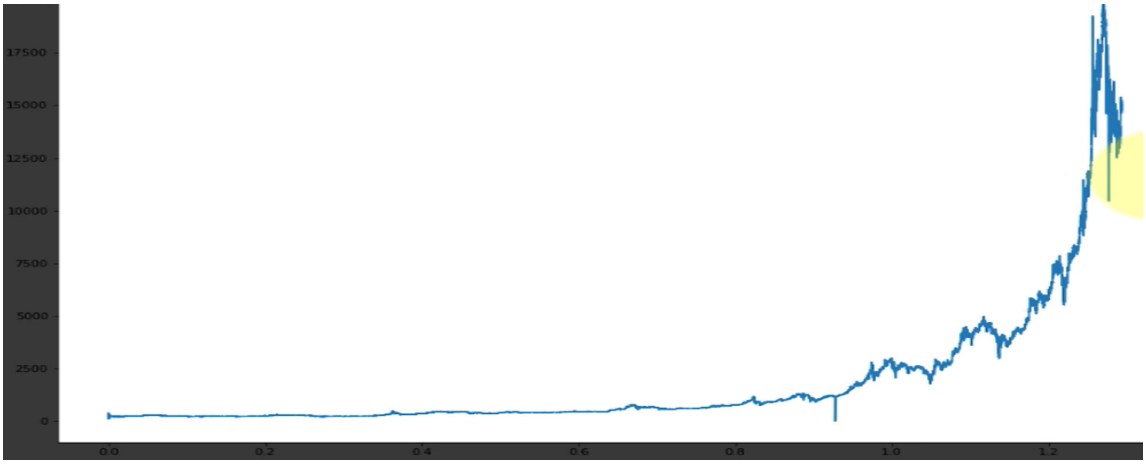
چون فایل های ما به شکل یک فایل csv است کاری که اول از همه میکنیم اینه که data مان را با pd میخوانیم

```
[ ]  1  dataframe = pd.read_csv("BTC_price_history.csv",
     2                          names=["Timestamp", "Low", "High", "Open", "Close", "Volume", "WeightedPrice"])
     3  dataframe = dataframe.drop(columns=['Timestamp'])
     4  dataframe.head(5)
```

این دقیقا شکل data ما است

	Low	High	Open	Close	Volume	WeightedPrice
0	300.0	300.0	300.0	300.0	0.010000	300.0
1	300.0	300.0	300.0	300.0	0.010000	300.0
2	370.0	370.0	370.0	370.0	0.010000	370.0
3	370.0	370.0	370.0	370.0	0.026556	370.0
4	377.0	377.0	377.0	377.0	0.010000	377.0

برای هر قیمتی که وجود دارد(weightedprice) اطلاعات مربوطه نوشته شده.برای هر کدام از این ها یک تایم زمانی داریم که من حذف کردم چون بدرد ما نمیخورد ما نیازی نداریم بدانیم مثلا توی ۲۰۰۷ چه قیمتی بوده فقط ترتیبشان برای ما مهم است . توی این جدول یکسری متد داریم که برامان مهم نیست و تغییری درشان ایجاد نمیکنیم چون هدف اصلی شبکه عصبی این است که متخصص توی کار نباشیم.



در حقیقت اینجا ۱/۲ معادل ۱۲۰۰۰۰۰ یعنی یک و دو دهم میلیون data است.ما فرض میکنیم اینجا را که تا سال ۲۰۱۹ است را میدانیم بعد میخواهیم بدانیم توی ۲۰۲۰ چقدر میشه هدفمان این است. هر کدام از این اعداد داخل جول بالا یک نقطه در داخل نمودار هستند که مثلا اوایل که بیت کوین مشهور نبود قیمتش پایین بود ولی بعد که مشهور شد قیمتش بالاتر رفت.

```
[ ]
def generate_dataset(data, test_size):
    """
    Generates dataset assuming only one variable for prediction
    Here ahead starts at 1
    :return:
    """

    scaler = MinMaxScaler(feature_range=(0, 1))
    scaler.fit(data)
    data = scaler.transform(data)

    train_size = len(data)-test_size

    bitcoin_train = data[:train_size, :]
    train_x, train_y = bitcoin_train[:, :-1], bitcoin_train[:, -1]

    bitcoin_test = data[train_size:, :]
    test_x, test_y = bitcoin_test[:, :-1], bitcoin_test[:,  -1]

    return train_x, train_y, test_x, test_y, scaler

```

اتفاق که می افتد اول از همه minmaxscaler استفاده میکنیم که این می آید min و max data(تمام data ها) را میگیرد مثلا این جا min data ۳۰۰ و max آن ۲۰۰۰۰ است پس می آید همه ی data هارا طوری میشکند که ۳۰۰ بشود ۰/۰۳و ۲۰۰۰۰ بشود ۹۹/. یعنی همه را میریزد بین ۰ و ۱ به صورتی که همون قدری که اول بودن همان وزن را حفظ کنند به این صورت عمل میکند مثلا فرض کنید ما ۱ و ۹ داریم اتفاق که می افتد می شود ۰/۱ و ۰/۹ اگر این را scale ۲برابر کنیم می شود ۲ و ۱۸ که این همان ۰/۱ و ۰/۹ است. پس کل هدفمان این است که همه را برگردانیم توی بازه ۰و۱

الان رنج volume همیشه جند صدم است در حالی که open یا price توی رنج چند صد هستند بعد به چند ۱۰۰۰۰ هم میرسند مشکل اینجاست که اگر ما scale نکنیم چون شبکه های عصبی فقط با عدد کار میکنند در واقع ما با عدد باید بهش بگوییم که روی price تمرکز کن به این صورت که بهش میگوییم ۲ از ۱ بیشتر است پس روی ۲ تمرکز کن و ۲ برات مهم تر است پس تمرکز ما اینجا روی open و close است و به volume به خاطر رنج کمش کاری نداریم.

برای scale کردن از minmaxscaler استفاده میکنیم.

ین کار هایی را که گفتیم را متد fit انجام میدهد.

Transform داده اولیه مان را میدهیم می آید data مان را می برد توی ویژگی که fit دارد که در اینجا منظور همان ۰و۱ کردن است.

این Generate_dataset یک testsize دارد برای این است که چقدر از این ۱/۲ میلیون data را ببریم به عنوان فشسف استفاده کنیم. اگر testsize را بدهیم ۵۰ درصد ۱/۲ میشود ۰/۶ که قطعا به ما جواب غلط میدهد چون اون قسمت پیک کردن های اصلی را اصلا ندیده ما testsize دستی دادیم ۸۰۰۰۰ دادیم که درواقع آخر های نمودارمان میشود که پیک کردن های اصلی را شامل شود.

```
[ ]  1  data_raw = dataframe.to_numpy()
     2  test_size = 80000
     3  # as data seems linear to 0.5, we omit that to focus on the challenges!
     4  train_x, train_y, test_x, test_y, scaler = generate_dataset(data_raw[500000:, :], int(test_size))
     5  print(train_x.shape)
     6  print(train_y.shape)
     7  print(test_x.shape)
     8  print(test_y.shape)
```

ما از data ۵۰۰۰۰ به بعد را دیدیم یعنی تا ۵۰۰۰۰ را ریختیم دور و از آن به بعد را دیدیم چون تا ۵۰۰۰۰ نوسانات خطی بود و فقط الگوریتم را کند میکرد و سود دیگری نداشت چالش اصلی از ۰/۶ به بعد است ولی اگر بخواهیم میتوانیم بیشتر ران کنیم.

Bitcoin_train این کل data منهای ۸۰۰۰۰ تا (چون مقدار train_size را ۸۰۰۰۰ دادیم) را میکند train و بقیه را میکند test

توی نمواری که داشتیم همه ی ستون ها به غیر از ستون آخر(که میشود weighted price) همه میشوند ویژگی های ورودی ما و ستون آخر چیزی است که ما میخواهیم پیش بینی کنیم پس ما ۵ تا وردی داریم.

این هم دقیقا همین کار را میکند و اون ۱- ها منظورش همان ستون آخر است که همه جا به عنوان test هستند

پ.ن: ۱-: یعنی از ستون ۰ تا ستون یکی مانده به آخر ولی ۱- یعنی ستون آخر

```
(713167, 5)
(713167,)
(80000, 5)
(80000,)

Constructing Tensor dataset
```

با اون شکانذنی که داشتیم ۷۱۳۱۶۷ تا برای train هستند و ۸۰۰۰۰ تا برای test ورودی یا همان trainsize مان ۵ تا ورودی دارد که منظورمان همان ۵ ستون است.

```
[ ]  1  train_x = torch.as_tensor(train_x.astype(np.float32)).cuda()
     2  train_y = torch.as_tensor(train_y.astype(np.float32)).cuda()
     3  test_x = torch.as_tensor(test_x.astype(np.float32)).cuda()
     4  test_y = torch.as_tensor(test_y.astype(np.float32)).cuda()
```

کاری که این جا میکنیم کار کردن با torch است. Torch همان np است که برای gpu ساخته شده چون اگر بخواهیم روی cpu ران کنیم خیلی طول میکشد روی gpu ران میکنیم.برای این کار از متدی به نام as_tensor که توی torch است کمک میگیریم که همان np array را میگیرد و میکند tensor.tensor همان vector بیشتر از ۲ بعد است که تا n هم میتواند برود(به vector ۲بعد ماتریس میگویند).

چون ما اینجا محدودیت رم داشتیم float32 گذاشتم.

Cuda هم متدی است که باعث میشود داده ها بروند توی gpu به جای cpu که همین باعث سرعت بیشتر میشود.

```
[ ]  1  train = torch.utils.data.TensorDataset(train_x, train_y)
     2  test = torch.utils.data.TensorDataset(test_x, test_y)
     3
     4  batch_size = len(train)
     5
     6  train_loader = torch.utils.data.DataLoader(dataset=train,
     7                                             batch_size=batch_size,
     8                                             shuffle=False)
     9
    10  test_loader = torch.utils.data.DataLoader(dataset=test,
    11                                            batch_size=batch_size,
    12                                            shuffle=False)
```

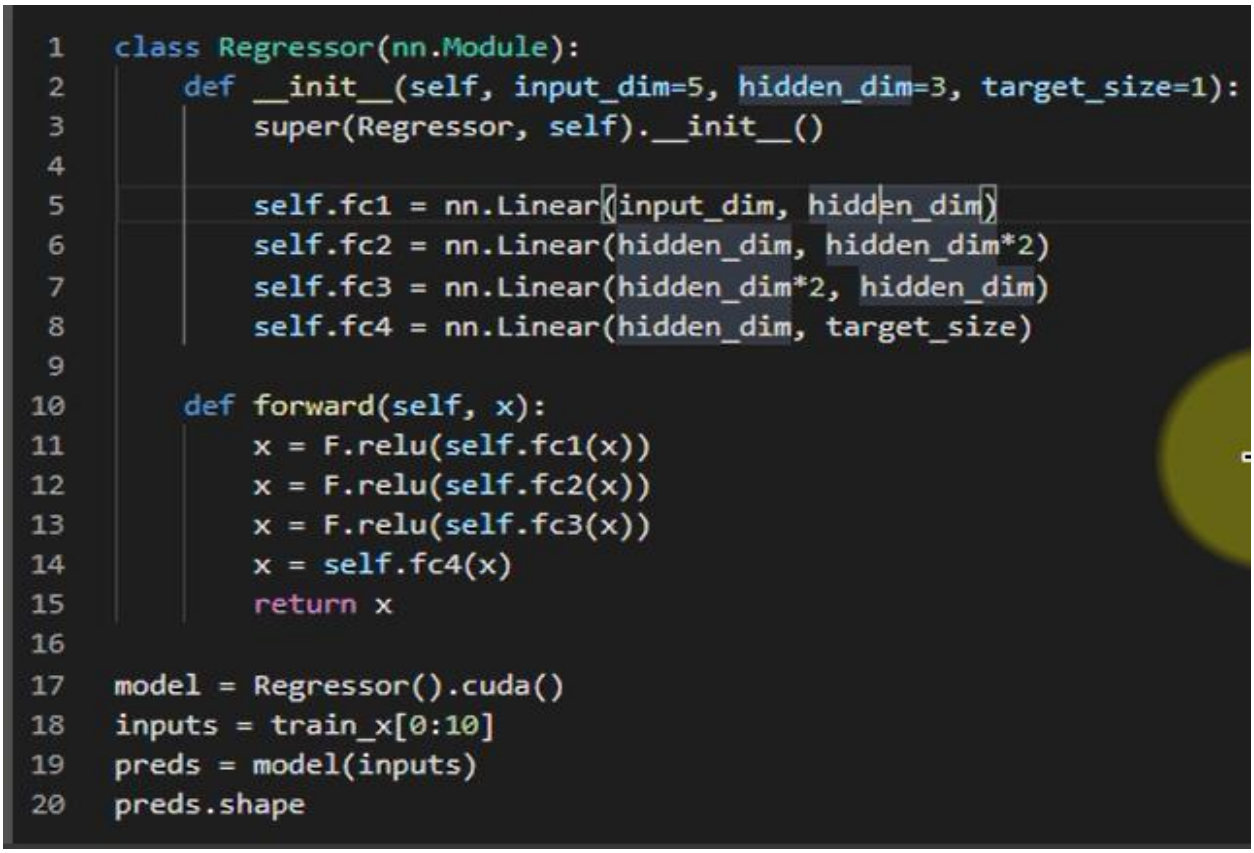
Tensordataset یک کلاس است.

ما یکسری از کار ها مثل بچ کردن data در همین کلاس است.

بچ کردن یعنی وقتی می خواهیم data را train کنیم (در اینجا نیازی نیست) مخصوصا برای تصویر که data خیلی بزرگ است نمیتوان همه را هم زمان داد چون خیلی رم میبرد همه data ها را همزمان نمیشه داد تکی هم نمیشه داد بخش بندی میکنیم به اصطلاح مثلا هر دفعه میگوییم برو ۲۰۰ تاش را بگیر . Tensordataset و dataloader این کار ها را برای ما میکنند یک کلاس خاص هستند.

میتوان batch_size داد ولی چون dataset ما کوچک است من batch_size را کل data دادم.

batch_size را دو تا درست میکنیم یکی برای train کردن یکی برای test کردن



```python
class Regressor(nn.Module):
    def __init__(self, input_dim=5, hidden_dim=3, target_size=1):
        super(Regressor, self).__init__()

        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, target_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x

model = Regressor().cuda()
inputs = train_x[0:10]
preds = model(inputs)
preds.shape
```

Class regressor این کل اون کلاسی است که داریم کل کار اون شبکه عصبی را برای ما انجام میدهد.

۵تا ستون داریم پس ما شبکه هایی که داریم ورودیش مجبور است که ۵تایی باشد فقط هم یک لایه ش (لایه ی اول) کافی است بقه را به دلخواه گذاشتم ۳ چون شبکه کوچکتر بهتر است سرعتش هم بیشتر است هرجه عدد بزرگتری بزاریم data بیشتر و زمان بیشتر میشود.

Target_size این است که نهایتا اون شبکه ای که داریم باید چه چیزی را پیش بینی کند که واسه ی ما ستون آخر یا weightedprice است پس برای Target_size یک value می خواهیم برای هر inputی که میدهیم.

از خط ۵ تا ۸ در این کلاس معماری شبکه است و خط های ۱۱تا ۱۵ هم همون کار را انجام میدهند برای flow کردن data توی شبکه.

چون همه به هم وصل هستند اسمش را fc گذاشتیم. این شبکه عصبی ماست.ماتریس های ما ۱*۱ ۱*۳ ۳*۳ ۶*۳ ۳*۶ ۳*۵ هستند.میتوان data ها را سطر سطر داد و میتوان کل data را با هم داد ما کل data را با هم دادیم.یکبار کل data را میدهیم همه یerror هاش را حساب میکند بعد متغییر ها را برای بهبود error ها آپدیت میکند ولی با یکبار انجام این کار یک عدد را خوب کرد و یک عدد دیگر را خراب کرد پس یکبار کافی نیست انقدر این کار را تکرار میکنیم تا یاد بگیرد مثلا ما ۱۰۰ بار این کار را تکرار کردیم که اگر error بار اول ۰/۱۵ بود توی بار آخر شد ۲ دهمزارم که این یعنی یادگرفت.

```
Epoch  71 MSE:  0.0020978715873775815
Epoch  72 MSE:  0.0019812476821243763
Epoch  73 MSE:  0.0018562690820544958
Epoch  74 MSE:  0.0017258316511288285
Epoch  75 MSE:  0.0015932382084429264
Epoch  76 MSE:  0.0014622723683714867
Epoch  77 MSE:  0.001337212510406971
Epoch  78 MSE:  0.0012212103465572
Epoch  79 MSE:  0.0011155520332977176
Epoch  80 MSE:  0.0010194546775892377
Epoch  81 MSE:  0.0009304831619374454
Epoch  82 MSE:  0.0008454677881672978
Epoch  83 MSE:  0.0007616473012603819
Epoch  84 MSE:  0.0006776262307539582
Epoch  85 MSE:  0.0005938134272582829
Epoch  86 MSE:  0.0005121907452121377
Epoch  87 MSE:  0.0004355291021056473
Epoch  88 MSE:  0.0003667924902401864 5
Epoch  89 MSE:  0.0003101707261521369
Epoch  90 MSE:  0.0002634206030052155
Epoch  91 MSE:  0.0002226865326520055 5
Epoch  92 MSE:  0.0001860187039710581 3
Epoch  93 MSE:  0.0001524171966593712 6
Epoch  94 MSE:  0.0001216597884194925 4
Epoch  95 MSE:  9.420995047548786e-05
Epoch  96 MSE:  7.084094977471977e-05
Epoch  97 MSE:  5.216824501985684e-05
Epoch  98 MSE:  3.829419074463658e-05
Epoch  99 MSE:  2.8684284188784 66e-05
Epoch 100 MSE:  2.2311078282655217e-05
```

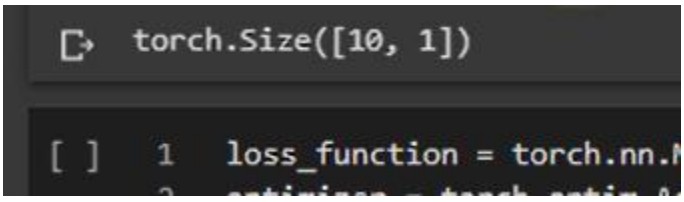
Relu تابعی است که خطی بودن را میشکند.

برای اینکه بتواند هر چیز غیر قطعی را پیش بینی کند ما به یک تابع نیاز داریم به نام activationfunction. این ها قطعا باید غیر خطی باشند.

Relu تابعی است که اگر بزرگتر از صفر بود خود data و اگر کوچکتر از صفر بود صفر میشود.به همین خاطر scale را بین ۰و۱ قرار دادیم میتوانستیم بین ۱-و۱ هم قرار دهیم ولی دیگر نمیتوانستیم از تابع relu استفاده کنیم چون هیچ data را منفی نمیکرد هر چیز منفی را صفر میکرد و باز همیشه بین ۰و۱ نشان میداد. این تابع میتواند خطی بودن را بشکند.

پس بعد از خروجی هر کدام از اون fc ها یک تابع relu قرار میدهیم ولی برای آخری نمیگذاریم چون ما به خود چیزی که میدهد نیاز داریم .

وقتی model را میسازیم بهش یک input میدهیم این input همان tensor است و به ما price میدهد.



اگر به torch ۱۰ تا ورودی بدهیم این ۱۰ تا value برای ما پیش بینی میکند.

```
1   loss_function = torch.nn.MSELoss().cuda()
2   optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
3
```

Lost_function همان error ی است دیدیم.

Optimizer را کپی کردم.

Lr یک ضریبی است که هر بار برای تغییر متغییر ها استفاده میشود تا error ها بهبود یابند چون تکرار را زیاد گذاشتیم(۱۰۰ بار میتوان بیشتر هم گذاشت فقط زمان بیشتر میشود) پرش زیاد لازم نیست .

```
# training
model.train()
for epoch in range(num_epochs):
    for idx, (inputs, labels) in enumerate(train_loader):
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()
```

این for یک لیست به ما میدهد .epoch همون کار را چند بار تکرار میکند. این ها کد های paytoch هستند که بهش zero_grad میگویند.

هر بار که data را میبریم تا انتها برمیگردیم و گرادیان را صفر میکنیم دوباره data را از اول میخوانیم که اگر error قبلی مان اشتباه بود دیگر error برنگردد.

```
# Step 2. Get our inputs ready for the network, that is, turn them into
# Tensors of word indices.
inputs_in = inputs
targets = labels.reshape(-1, 1)
```

Target همان ستون price است این recheck فقط برای این است که error ندهد.

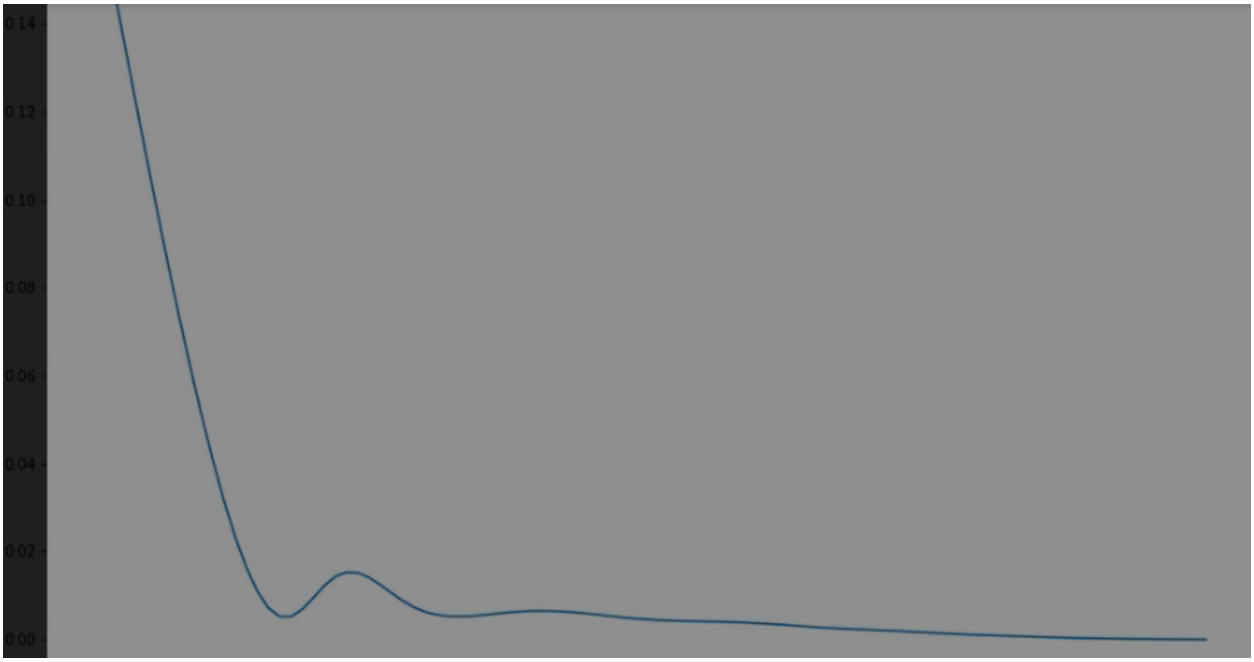
```
# training
model.train()
for epoch in range(num_epochs):
    for idx, (inputs, labels) in enumerate(train_loader):
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Tensors of word indices.
        inputs_in = inputs
        targets = labels.reshape(-1, 1)

        # Step 3. Run our forward pass.
        preds = model(inputs_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(preds, targets)

    if epoch % 1 == 0:
        print("Epoch ", epoch+1, "MSE: ", loss.item())
        hist[epoch] = loss.item()

        loss.backward()
        optimizer.step()
```

Predهمان model است که input را میدهیم و به ما priceرا میدهد.

Loss را حساب میکند. در این لحظه pred و target را داریم بعد error را برمیگردانیم در حقیقت ما بهش میگوییم که چه جوری جلو میرویم. hist عمل ذخیره کردن را انجام میدهد

اوایل چون خیلی نیاموخته شیبش زیاد است ولی بعدش منطقی تر میشود.

در این نمودار اگر ما bachsize را که کل data در نظر گرفته بودیم را بگیریم مثلا یک صدم data نمودارمان زیک زاکی میشد.



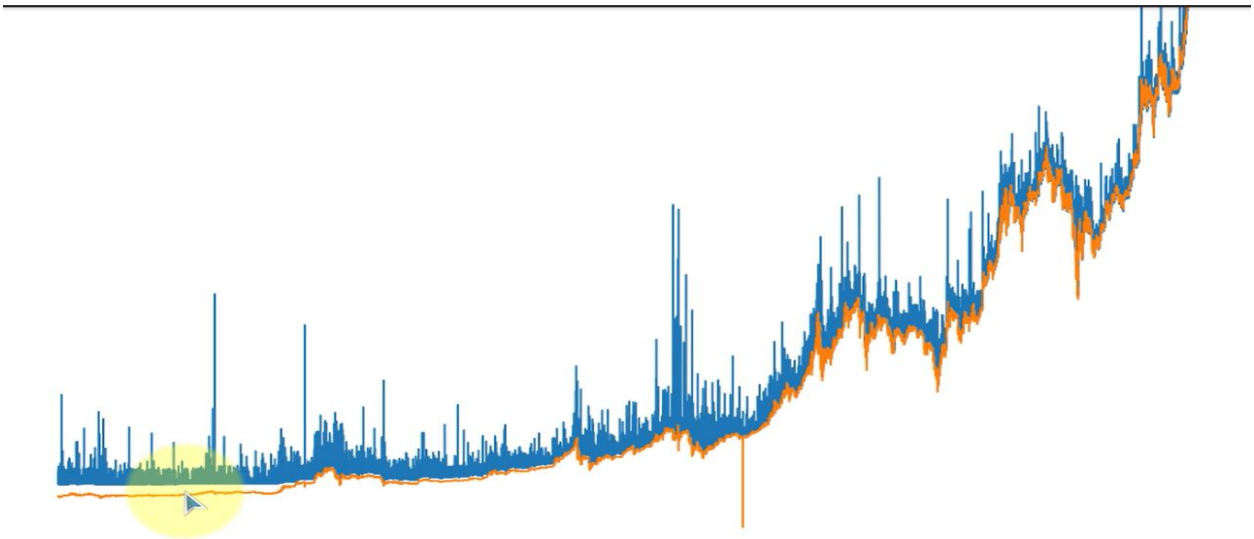

```
    # Step 4. Compute the loss, gradients, and update the parameters by
    #  calling optimizer.step()
    loss = loss_function(preds, targets)

if epoch % 1 == 0:
    print("Epoch ", epoch+1, "MSE: ", loss.item())
    hist[epoch] = loss.item()

    loss.backward()
    optimizer.step()
```

```
[ ]  1  model.eval()
     2  with torch.no_grad():
     3      preds = model(test_x)
     4      loss = loss_function(preds, test_y.reshape(-1, 1))
     5      print('loss', loss.item())
     6  test_rescaled = scaler.inverse_transform(np.concatenate((test_x.cpu().numpy(), test_y.cpu().reshape(-1, 1).numpy()), axis=1))[:,-1]
     7  preds_rescaled = scaler.inverse_transform(np.concatenate((test_x.cpu().numpy(), preds.cpu().numpy()), axis=1))[:,-1]
```

موقعی که میخواهیم model را تست کنیم برای اینکه ram و gpu کمتر مصرف شود میگوییم توی حالت eval هستیم و دیگر برای ما گرادیان حساب نکن فقط برو جلو بعد چون dataمان رویgpu بود میبریم توی cpu بعد numpy میکنیم که بتوان نمودار رسم کرد.

Scaler.inverse اعداد را بین min و max اصلی که در اینجا ۳۰۰ و ۲۰۰۰۰ بود قرار میدهد.

Pred یعنی همان چیز هایی که شبکه پیش بینی کرد .



میبینیم که train را دقیقا یاد گرفته چون نمودار پیش بینی شده دقیقا روی هم قرار گرفته اند.