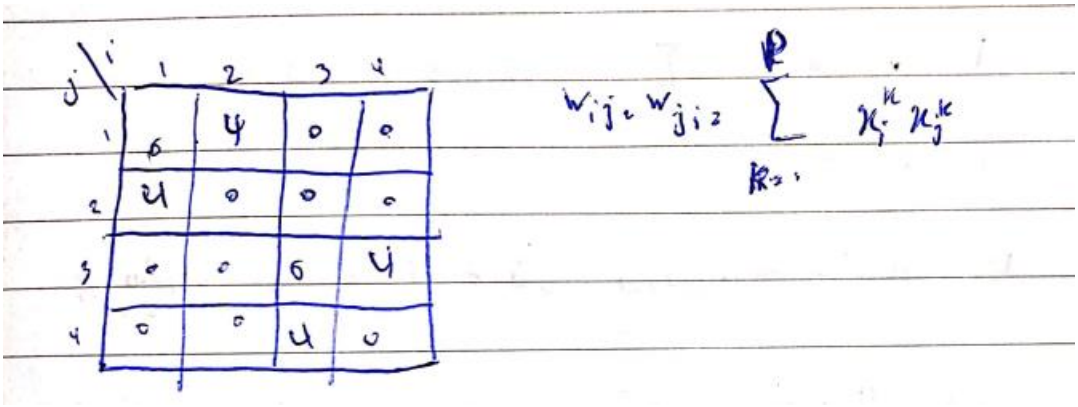
Q1)

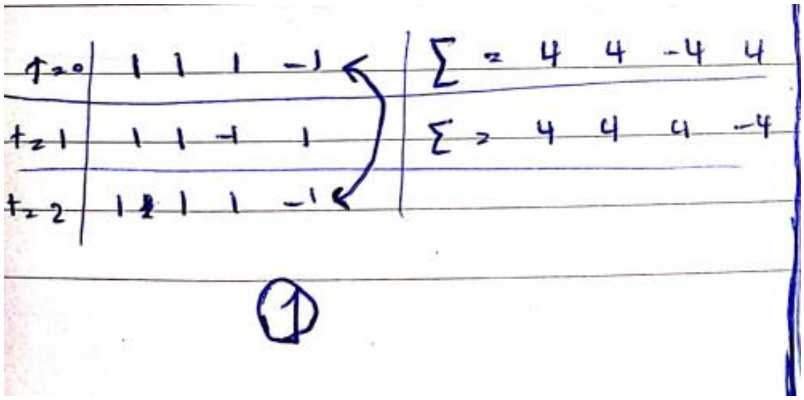
Pdf پاسخ در فایل زیپ موجود است.

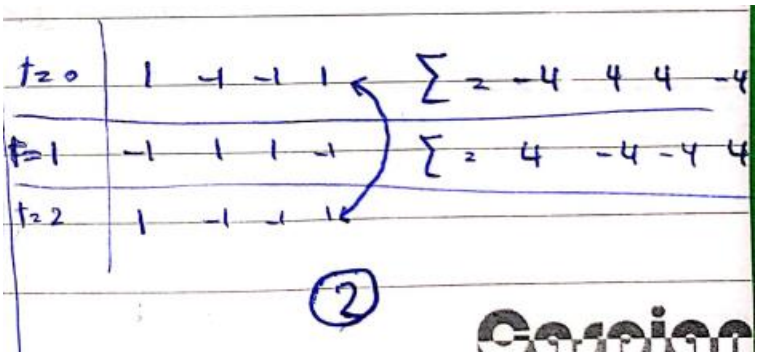
Q2)

برای بررسی این موضوع ابتدا ماتریس وزن ها را تشکیل دادم و پر کردم:



سپس چند نمونه مختلف را امتحان کردم تا ببینم آیا ورودی به حافظه ای همگرا میشود یا نه:





میبینیم که مدل به چیزی همگرا نشده و بین دو مقدار غیر حافظه گردش میکند. مقادیر دیگر هم همین وضعیت را دارند. و مدل هاپفیلد توانایی ذخیره سازی این حافظه ها را ندارد.

همچنین در مدل هاپفیلد لازم است که تعداد نورون ها از تعداد حافظه ها خیلی بیشتر باشد در حالی که در این مسئله، تعداد نورون ها و حافظه ها با هم برابر هستند. این مشکل باعث میشود که ذخیره سازی این حافظه ها با مشکل مواجه شود.

Q3)

این سوال هم خیلی جالب بود و به درک من از مدل و اکتیویشن فانکشن های مختلف خیلی کمک کرد با سپاس.

فایل پاسخ این سوال در فایل زیپ با نام CI-HW3-Q3-Numpy.ipynb موجود است.

برای حل این سوال ابتدا سعی کردم بدون نامپای و با استفاده از keras مدل های مختلف را تست کنم تا ببینم ساده ترین مدلی که میتواند این تابع را یادبگیرد چیست.

به طور کلی در لایه آخر یک نورون با اکتیویشن فانکشن خطی(بدون اکتیویشن فانکشن) قرار میدادم زیرا عملیات مورد نظر ما رگرشن است. در لایه های میانی هم توابع مختلف مثل ریلو و tanh را تست کردم و ریلو از همه بهتر جواب میداد.

در نهایت به مدلی با یک لایه مخفی با 20

نورون رسیدم:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                20

 dense_1 (Dense)             (None, 1)                 11

=================================================================
Total params: 31 (124.00 Byte)
Trainable params: 31 (124.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Input((1,)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1, activation='linear'),
])
```

وقتی از ساختار مدل مطمئن شدم به سراغ پیاده سازی مدل با numpy رفتم. کدهای نوشته شده برای پیاده سازی به کمک numpy در ادامه شرح داده شده اند:

کتابخانه ها مورد استفاده تنها numpy و matplotlib برای آموزش مدل و رسم نتایج هستند:

```
[167] import numpy as np
      import matplotlib.pyplot as plt
```

در این قسمت داده های آموزشی تعریف کردم که بازه داده ها از -5 تا 5 و توزیع داده ها نرمال میباشد:

```
[168] x_train = np.random.random((1000, 1, 1)) * 10 - 5
      y_train = (x_train ** 2).reshape(len(x_train),)
```

در این قسمت ماتریس وزن ها را به صورت رندوم با توزیع نرمال initialize کردم:

```
[170] W1 = np.random.rand(1, 20)
      W2 = np.random.rand(20, 1)
      b1 = np.random.rand(1,20)
      b2 = np.random.rand(1,1)
```

سپس توابع مورد نیازم که relu و تابع مشتق آن است را تعریف کردم:

```
[172] def ReLU(a):
        return np.maximum(a, 0)

      def ReLU_derivitive(a):
        return a > 0
```

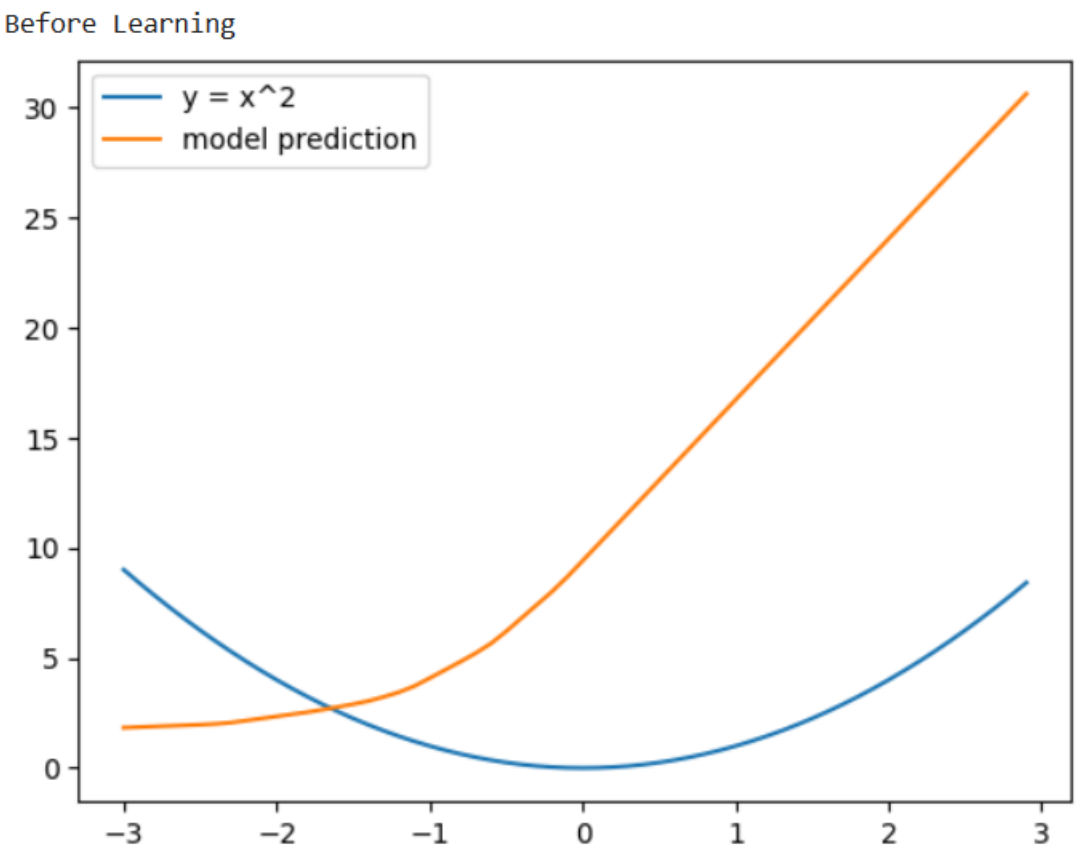
سپس عملیات فروارد را تعریف کردم:

```
[173] def forward(x, W1, W2, b1, b2):
        z1 = np.dot(x, W1) + b1
        a1 = ReLU(z1)
        z2 = np.dot(a1, W2) + b2
        a2 = z2
        yhat = a2
        return yhat
```

و پس از آن، عملیات backpropagation را طبق محاسباتی که کرده بودم ایجاد کردم:

```
[174] def backprop(x, W1, W2, b1, b2, y, yhat, alpha):
        z1 = np.dot(x, W1) + b1
        a1 = ReLU(z1)
        z2 = np.dot(a1, W2) + b2
        a2 = z2

        dl_dz2 = a2 - y
        dl_db2 = dl_dz2
        dl_dW2 = np.dot(a1.T, dl_dz2)

        dl_da1 = np.dot(W2, dl_dz2)
        dl_dz1 = dl_da1 * ReLU_derivitive(z1).T

        dl_db1 = dl_dz1
        dl_dW1 = np.dot(dl_dz1, x).T

        b1 = b1 - alpha * dl_db1.T
        W1 = W1 - alpha * dl_dW1
        b2 = b2 - alpha * dl_db2
        W2 = W2 - alpha * dl_dW2

        return W1, W2, b1, b2
```

حالا ابزارهای لازم برای آموزش مدل فراهم است. قبل از شروع آموزش پیشبینی مدل را برای اعداد -3 تا 3 با گام های 0.1 میبینیم:



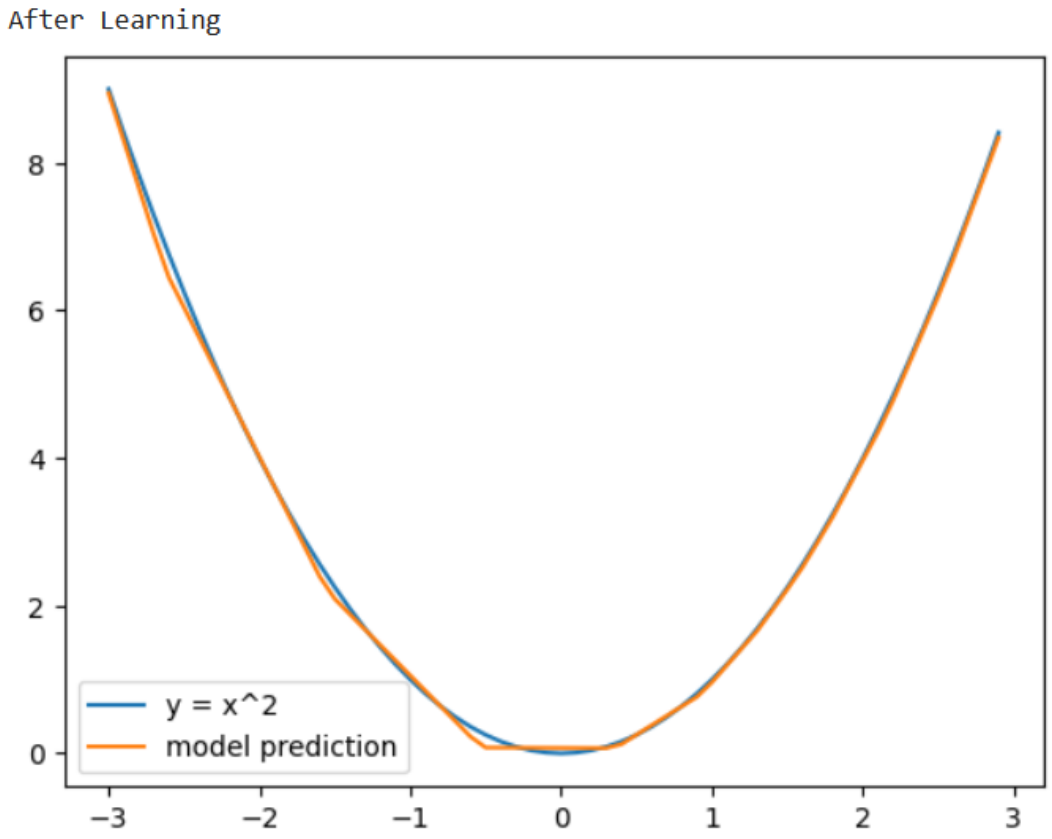
میبینیم که مدل نتیجه منطقی ای ندارد که طبیعی است.

سپس فرایند آموزش را شروع کردم. چون محاسبات سریع انجام میشد و محدودیتی در زمان نبود، 500 ایپاک برای آموزش تعیین کردم تا مدل کاملا train شود.

```
alpha = 0.01
epochs = 500

for epoch in range(epochs):
  for i in range(len(x_train)):
    yhat = forward(x_train[i], W1, W2, b1, b2)
    W1, W2, b1, b2 = backprop(x_train[i], W1, W2, b1, b2, y_train[i], yhat, alpha=alpha)
```

لازم به ذکر است که از روش SGD برای حل این سوال استفاده کردم.

پس از این که آموزش تکمیل شد مجددا پیشبینی مدل را برای بازه -3 تا 3 چاپ کردم.



میبینیم که مدل به خوبی train شده و پیشبینی ای نزدیک به y= x^2 دارد. برای دقیق تر شدن پیشبینی و کمتر شدن فاصله خط زرد با آبی میتوانستیم به جای 20 نورون از 50 نورون در لایه میانی استفاده کنیم اما چون زمانبرتر میشد من از همین 20 نورون استفاده کردم.

برای خوانایی بیشتر مدل، فایل پیاده سازی مدل با keras را هم در فایل زیپ قرار داده ام.

Q4)

Pdf پاسخ در فایل زیپ موجود است.

Q5)

SOM

اساسا مدل SOM برای انجام چنین کاری نیست. SOM برای کاهش ابعاد یا پخش یکسری نمونه روی نقشه دو بعدی است. با ورودی دادن نقشه شهر ها به SOM، این مدل تنها این شهر را را روی یک نقشه دو بعدی مپ میکند

MLP

برای مدل mlp میتوان گراف شهرها و فاصله ها را به صورت یک ماتریس n*n به مدل ورودی داد و در خروجی پاسخ TSP رابه عنوان لیبل مشخص کرد. اما مدل های mlp بر اساس گرادیان کار میکنند و w ها را به صورت پیوسته آپدیت میکنند. اما محاسبه TSP به صورت قطعی

انجام میشود و تنها یک پاسخ صحیح وجود خواهد داشت. بنابراین مدل mlp حتی اگر روی داده های آموزشی به دقت مناسبی برسد، نمیتواند generalization را به درستی انجام دهد. همچنین تعداد شهرها میتواند متفاوت باشد که کار را برای یادگیری mlp سخت تر میکند.

Hopfiled

همانطور که در اسلایدهای درسی اشاره شده از مدل Hopfield میتوان برای تسک های optimization استفاده کرد. یکی ازین تسک ها همین مسئله TSP میباشد. برای حل این مسئله به کمک Hopfield میتوان ابتدا هر شهر را یک نورون در مدل هاپفیلد در نظر گرفت سپس ماتریس شهر را به صورت one-hot در یک ماتریس n*n نشان داد:

$$M = \begin{bmatrix} A: & 1 & 0 & 0 & 0 \\ B: & 0 & 1 & 0 & 0 \\ C: & 0 & 0 & 1 & 0 \\ D: & 0 & 0 & 0 & 1 \end{bmatrix}$$

همچنین هزینه مسافرت بین شهر ها را میتوان در یک ماتریس n*n مثل C نشان داد که هر خانه ماتریس هزینه سفر از شهر سطر به شهر ستون را نشان میدهد.

در نهایت کافیست از این تابع انرژی برای مدل هاپفیلد استفاده کرد که در آن، M ماتریس شهرها و C ماتریس هزینه مسافرت است:

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{x}\sum_{y\neq x} C_{x,y}M_{x,i}(M_{y,i+1} + M_{y,i-1}) +$$

$$\left[\gamma_1\sum_{x}\left(1 - \sum_{i}M_{x,i}\right)^2 + \gamma_2\sum_{i}\left(1 - \sum_{x}M_{x,i}\right)^2\right]$$

این تابع همان تابعی است که در مدل هاپفیلد مینیمم میشود. با مینیمم کردن انرژی مدل هاپفیلد با این شرایط، به پاسخ TSP میرسیم.

منبع: Optimization Using Hopfield Network (tutorialspoint.com)