

گزارش تمرین اول شبکه عصبی

علیرضا آزادبخت ۹۹۴۲۲۰۱۹

تمرین ۱:

پیاده سازی از پایه شبکه های عصبی:

طبق خواسته صورت سوال کلاس های مورد نیاز را برای عملکرد درست شبکه های عصبی به صورت زیر آماده کردیم.

```
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    def forward(self, input_data):
        pass

    def backward(self, error, learning_rate):
        pass
```

```
class Fully_Connected_layer(Layer):
    def __init__(self, input_size, output_size):
        self.weights = np.random.uniform(low=-0.5, high=0.5, size=(input_size, output_size))
        self.bias = np.random.uniform(low=-0.5, high=0.5, size=(1, output_size))

    def forward(self, input_data):
        self.input = input_data
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output

    def backward(self, error, learning_rate):
        gradient = np.dot(self.input.T, error)
        self.weights -= learning_rate * gradient
        self.bias -= learning_rate * error
        return np.dot(error, self.weights.T)
```

```
class Activation_Layer(Layer):
    def __init__(self, function, d_function):
        self.activation = function
        self.d_activation = d_function

    def forward(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    def backward(self, error, learning_rate):
        return self.d_activation(self.input) * error
```

```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.d_loss = None

    def set_loss_function(self, function, d_function):
        self.loss = function
        self.d_loss = d_function

    def add_layer(self, layer):
        self.layers.append(layer)

    def fit(self, x, y, epoch, learning_rate):
```

```

errors = []
for i in range(epoch):
    error = 0
    for j in range(len(x)):
        output = x[j:j+1]
        for layer in self.layers:
            output = layer.forward(output)
        error += self.loss(y[j], output)
        d_error = self.d_loss(y[j], output)
        for layer in reversed(self.layers):
            d_error = layer.backward(d_error, learning_rate)
    if i%100==0:
        print("epoch {} Done: error = {}".format(i+1,f"{(error/len(x)):,}" ))
    errors.append(error/len(x))
return errors

def predict(self, x):
    result = []
    for i in range(len(x)):
        output = x[i:i+1]
        for layer in self.layers:
            output = layer.forward(output)
        result.append(output)
    return result

```

در مرحله کنار معماری چیده شده تابع های activation مانند sigmoid, softmax, tanh, sign, relu و یک تابع اندکینگ one hot و توابع loss مانند mse و cross entropy به همراه مشتق های آن ها پیاده سازی شدند.

در بخش شبکه اصلی از گرادیان دیسنتت معمولی استفاده شد که در مرحله اول ورودی های را به ترتیب لایه های شبکه که پشت سر هم در آرایه ای ذخیره سازی کرده است به جریان میاندازد و خروجی هر لایه را به ورودی بعدی میدهد و در مراحل باز پخش خطا و تغییرات وزن ها مسیر لایه ها را برعکس میاید و از تابع مشتق آن ها استفاده میکند و خطا را باز پخش میکند و وزن ها را با توجه به لرنینگ ریت تغییر میدهد.

مسئله رگرسیون:

برای بخش رگرسیون ابتدا داده های را با نسبت ۷۰ ۳۰ به تست و آموزش جدا کردیم و یک بار به یک مدل رگرسیون معمولی دادیم و از دقت این مدل به عنوان مدل بیس لاین استفاده کردیم.

در مرحله اول یک پرسپترون ساده به کمک فریم ورکی که نوشتیم پیاده سازی کردیم.

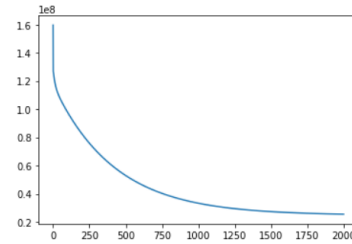
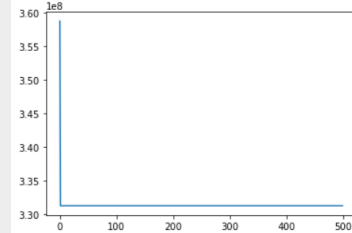
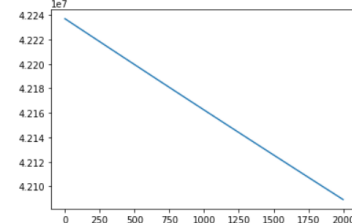
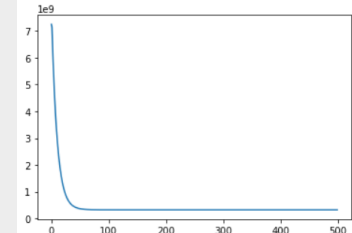
در مرحله دوم یک شبکه عصبی چند لایه با معماری زیر به کمک فریم ورکی که نوشتیم پیاده سازی کردیم.



در مرحله سوم یک پرسپترون به کمک پارتورچ پیاده سازی کردیم.

در مرحله چهارم همان معماری شبکه چند لایه را به کمک پایتورچ پیاده سازی کردیم.

و در زیر نتایج عملکرد مدل ها را مشاهده میکنیم:

model	MSE on test set	time	Train process
Linear Regression	10,081,303	-	-
My_Perceptron	41,863,718	2min 49s	
My_MLP	336,259,507	2min 3s	
Torch_Perceptron	44,078,244	832 ms	
Torch_MLP	335,471,872	1.86 s	

نتیجه گیری:

هیچ یک از مدل ها نتوانستند به دقت مدل رگرسیون خطی معمولی برسند که این به این معنی است که بر اساس گرادیان دیسنت مدل ها در بهینه محلی افتاده اند و چون ویژگی های دیگر مانند شتاب و اپتیمایز های بهینه تر استفاده نکردیم این اتفاق افتاده است.

اما بین شبکه ها مدل پرسپترون نوشته شده خودمان بهترین نتیجه را گرفته و مدل شبکه چند لایه با اختلاف کمی با هم نوع خودش به کمک پایتورچ فاصله دارد، اما نکته ای که وجود دارد شبکه های پای تورچ در سرعت اجرا به شدت از مدل های ما بهتر هستند که این نیز به دلیل بودن نسبی پایتون میباشد چون بخشی از کد

های ما در سطح پایتون اجرا میشدند و این باعث افزایش زمان مورد نیاز میشد. و مدل پرسپترون پایتورچ بر خلاف مدل خود با شیب معمولی و متوسطی بهینه میشد و خطایش کم میشد.

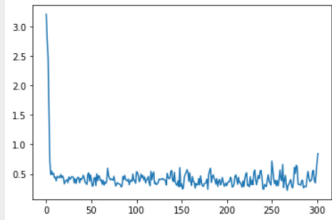
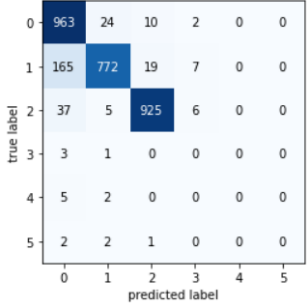
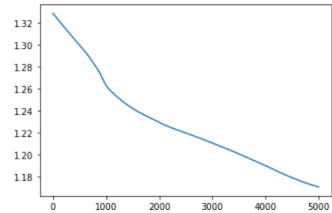
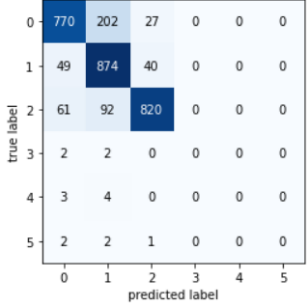
مسئله کلاسبندی:

برای بخش کلاسبندی ابتدا داده های را با نسبت ۷۰ ۳۰ به تست و آموزش جدا کردیم. در این بخش ۶ کلاس مختلط از داده ها داریم که قصد کلاسبندی آن ها را داریم و برای این کار از تابع cross entropy استفاده کردیم.

در مرحله اول یک شبکه عصبی چند لایه با معماری زیر به کمک فریم ورکی که نوشتیم پیاده سازی کردیم.



در مرحله دوم همان معماری شبکه چند لایه را به کمک پایتورچ پیاده سازی کردیم. و در زیر نتایج عملکرد مدل ها را مشاهده میکنیم:

model	Accuracy	F1-Macro avg	F1-Weighted avg	time	Train process	Confusion Matrix
My_MLP	0.90	0.45	0.90	2min 53s		
Torch_MLP	0.83	0.42	0.83	14 s		

نتیجه گیری:

مدل پایتورچ با وجود اینکه تقریباً ۱۷ برابر ایپاک بیشتری آموزش دید اما همچنان از نظر سرعت از مدل ما سریع تر بود اما همانطور که در ماترین کانفیوژن به صورت شهودی و از درصد دقت مدل ها متوجه قابل مشاهده است مدل پیاده سازی شده ما نتیجه خیلی بهتری گرفته و در عملیات کلاسبندی داده ها بسیار موفق تر عمل کرده است.

هیچ یک از مدل ها در پیشبینی کلاس های ۴ و ۵ و ۶ موفق نبودند و نتوانستند به هیچ عنوان آن ها را پیشبینی کنند، و به همین دلیل است که $f1\text{-micro avg}$ مقدار کمی محاسبه شده است. ممکن است کلاس های ۴ و ۵ و ۶ دارای نوز باشند یا نکته خاصی داشته باشند اما چون اطلاعات بیشتری از آن ها نداریم نمیتوانیم تصمیم درستی در این باره بگیریم و به به صلاح مدل ها پردازیم.

تمرین ۲:

مجموعه داده بانکی:

مقدمه:

در این تمرین دیتاستی شامل ۴۱۱۸۸ داده با بردار ویژگی ۲۱ از مشخصات کاربران بانک در اختیار ما قرار گرفته و وظیفه اصلی در این دیتاست پیشبینی این است که آیا کاربر با این ویژگی ها اشتراک خاصی را از بانک تهیه میکند یا خیر.

مشکلی که در داده ها وجود دارد نا متوازن بودن کلاس های آن میباشد به طوری که ۳۶۵۴۸ داده متعلق به کلاس صفر و تنها ۴۶۴۰ داده متعلق به کلاس یک میباشد. و باید شبکه به طوری تنظیم شود که بتواند این مشکل را هندل کند.

تمیز سازی داده ها:

حذف مقادیر null:

نکته ای که در برخورد اولیه با دیتاست به چشم میخورد مقادیر زیاد null در بعضی از ویژگی ها است به طور کلی:

default	8597
education	1731
housing	990
loan	990
job	330
marital	80

- برای ویژگی های کتگوریکال که مقادیر آن ها خالی بود مانند: 'education', 'housing', 'loan' به کمک مدل درختی CATBoost و فیچر های دیگه موجود در دیتاست بغیر از 'job', 'marital' می توان به پیشبینی مقادیر نال کردیم. مدل CATBoost یک نوع مدل درختی است که می تواند به راحتی فیچر های کتگوریکال را بدون انکود شدن به عنوان ورودی بگیرد و خروجی نیز به صورت کتگوریکال باز گرداند و از نظر قدر و سرعت از مدل های جنگل تصادفی بهتر عمل می کند. برای هر یک از فیچر های هدف این مدل را بر روی باقی بردار ویژگی که دارای مقدار در این ویژگی بودند آموزش دادیم و نتیجه پیشبینی را بر روی داده هایی که مقدار نال داشتند حساب کردیم و در آن ها قرار دادیم. (فیچر هدف را از این مجموعه حذف کردیم که باعث دیتا لیکج در کار های پیش رو نشویم)

- و فیچر default به دلیل زیاد بودن مقادیر نال آن به کلی دراپ شد

مهندسی ویژگی:

- در این بخش ویژگی های کتگوریکال ['housing', 'loan', 'contact', 'subscribed', ''] که دارای خاصیت ترتیب طبیعی هستند را به کمک labelEncoding کتابخانه sklearn انکد کردیم.
 - ویژگی های ['poutcome', 'education', 'marital', 'job'] که ترتیب خاصی نداشتند را به صورت one hot انکد کردیم.
- در پایان این بخش با ۳۹ ویژگی آماده مدل سازی دیتا و پیشبینی های هدف شدیم.

مدل سازی:

ابتدا مجموعه داده ها را با نسبت ۷۰ ۳۰ به به مجموعه داده های آموزش و تست تقسیم کردیم. در قسمت های بعدی در هر مرحله عصر تغییر یک مقدار و ویژگی شبکه را مورد بررسی قرار دادیم.

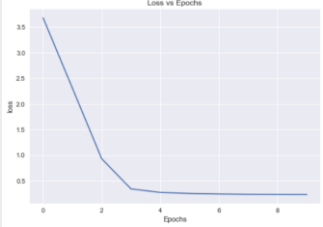
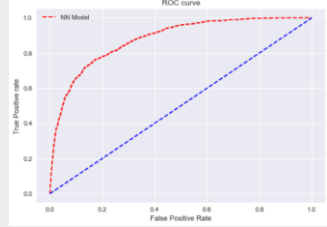
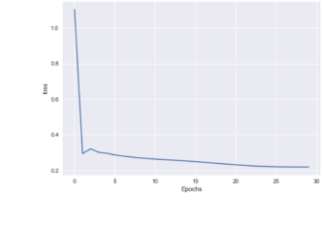
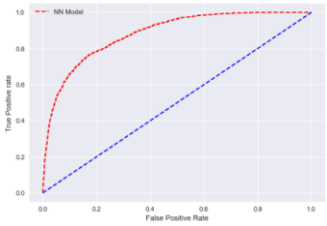
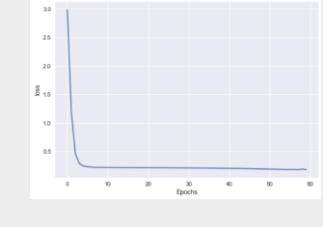
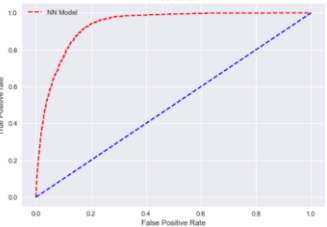
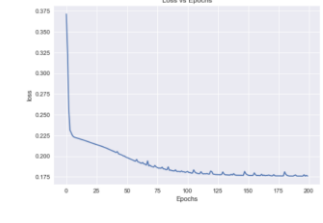
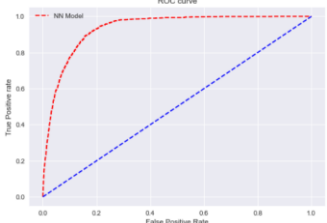
شبکه ای که طراحی کردیم به صورت احتمالاتی به ما احتمال اینکه هر داده عضو کدام کلاس است را به صورت احتمالاتی مشخص میکند در بخش نتایج مقادیر recall و precision با threshold پنجاه درصدی بر روی احتمالات در نظر گرفته شده و همانطور که میتوان از نمودار ROC دید با مقادیر دیگر میتوان نتایج بهتری نیز گرفت اما در این بخش با همان مقدار پنجاه درصد پیش میرویم.

معماری دیفالت تحلیل های انجام شده ۳ لایه نورون ۱۰ عضوی مخفی و یک نورون خروجی میباشد که در شکل هم معماری به صورت شهودی قابل مشاهده است.

: epoch

معماری:



Epoch	Learning rate	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
100	0.001	Adam	0.87	0.68	0.32		
300	0.001	Adam	0.88	0.69	0.31		
600	0.001	Adam	0.93	0.70	0.33		
2000	0.001	Adam	0.93	0.66	0.44		

بعد از ایپاک ۶۰۰ به بعد دیگر فرایند یادگیری جدیدی اتفاق نمی‌افتد و شبکه شروع به اور فیت شدن میکند

: Learning Rate

معماری:



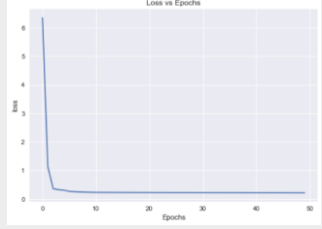
Epoch	Learning rate	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.1	Adam	0.90	0.67	0.40		
500	0.01	Adam	0.91	0.69	0.32		
500	0.001	Adam	0.88	0.69	0.32		

مقدار لرنینگ ریت ۰.۱ بقدری زیاد است که مدل از روی اپتیمم های خوب عبور میکند و مقدار ۰.۰۰۱ بقدری کم است که مدل نمیتواند از بهینه محلی خارج شود و مقدار بهینه ۰.۰۱ است.

: Layer Size

معماری:

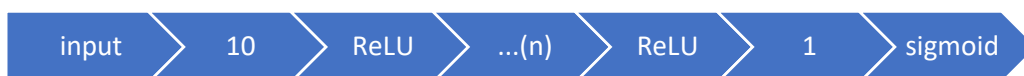


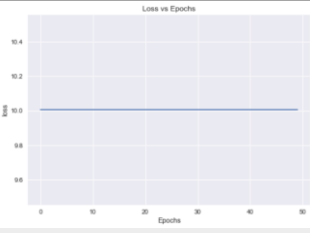
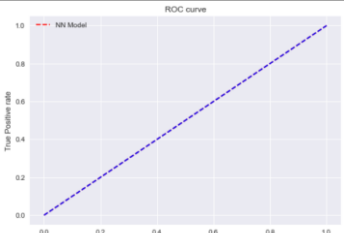
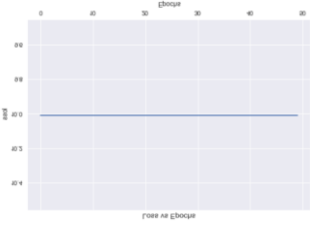
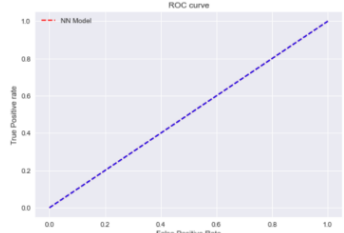
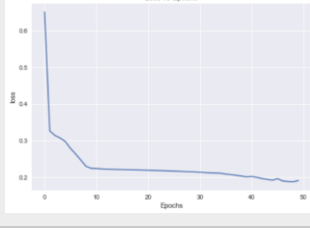
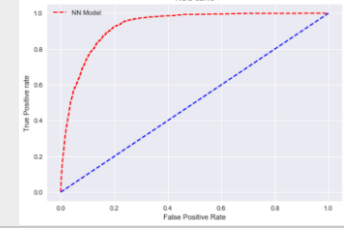
Epoch	Learning rate	M	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.01	5	Adam	0.71	0.67	0.36		
500	0.01	15	Adam	0.89	0.66	0.39		
500	0.01	25	Adam	0.88	0.66	0.39		

وقتی پیچیدگی شبکه از پیچیدگی مسئله بیشتر شود دقت مدل رفته رفته افت میکند، و هر اگر مدل به اندازه کافی پیچیده نباشد توانایی حل مسئله را ندارد.

: Number of Layer

معماری:



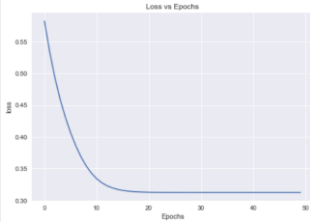
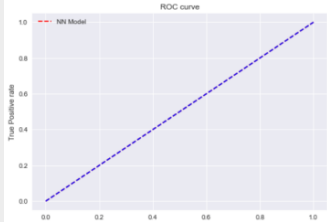
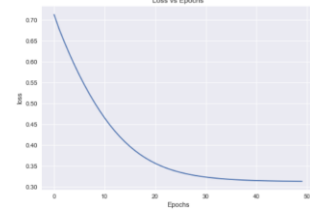
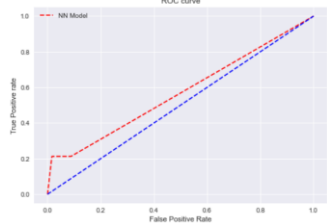
Epoch	Learning rate	n	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.01	2	Adam	0.5	0.0	0.0		
500	0.01	3	Adam	0.5	0.0	0.0		
500	0.01	6	Adam	0.92	0.66	0.42		

نکته ای که در بخش قبل گفتیم در این بخش هم به صورت تعداد نوروں ها خودش را نشان میدهد.

: Activation

معماری:



Epoch	Learning rate	optimizer	Func	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.01	Adam	Tanh	0.5	0.0	0.0		
500	0.01	Adam	sigmoid	0.56	0.0	0.0		

این دو تابع فعال سازی به دلیل مشکلاتی که دارند و محدودیتی که بر مقادیر خروجی خود اعمال میکنند مناسب این تسک نیستند و عملاً آموزش در یکی از آن ها به کلی اتفاق نمیافتد و در یکی از آن ها به مقدار خیلی کمی صورت میگیرد و این بخش میتوان نتیجه گرفت که ReLU نسبتاً تابع خیلی مناسبی محسوب میشود.

: optimizer

معماری:



Epoch	Learning rate	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.01	SGD	0.91	0.0	0.0		
500	0.01	Adadelta	0.68	0.11	1.00		
500	0.01	ASGD	0.62	0.0	0.0		
500	0.01	AdamW	0.90	0.66	0.36		

در این بخش مشاهده میکنیم که اکثر اپتیمایز ها نتایج خوبی نگرفته اند و اپتیمایز SGD هم به صورت تصادفی نتیجه نسبتا خوبی گرفته و یادگیری در آن صورت نگرفته ولی اپتیمایز های بر پایه Adam مانند AdamW نتایج خوبی کسب کرده اند.

: Normalize input

معماری:



Epoch	Learning rate	optimizer	AUC	Precision On Threshold =0.5	Recall On Threshold =0.5	Train Process	ROC
500	0.01	Adam	0.93	0.60	0.56		

روی دیتای نرمال نشده هم مدل 0.93 AUC گرفته بود پس نتیجه میگیریم با این معماری شبکه تاثر چندانی در نرمال بودن یا نبودن داده ها نمیتوانیم مشاهده کنیم.

نتیجه گیری:

شبکه ها طراحی شده را با انواع پارامتر ها بررسی کردیم و نتایج تمامی آن ها را گزارش دادیم و اگر بخواهیم کلسیفایر درستی طراحی کنیم ابتدا معاری و مقادیر بهینه را انتخاب میکنیم و از روی ROC حد مناسب را برای ترشهولد در نظر میگیریم و به جای آن ۰.۵ در کد ها قرار میدهم و خروجی مدل ها به صورت 0 و 1 میشود و نیاز مسئله برطرف میشود.