

حسابگری زیستی؛  
پیاده‌سازی شبکه عصبی؛  
کامیار میرزاوزیری؛ ۶۱۰۳۹۶۱۵۲

## فهرست مطالب

۱	مقدمات و ابزارها
۲	۱.۱ توابع کمکی
۲	۲.۱ ترسیم گر
۳	۳.۱ نورون
۳	۴.۱ شبکه عصبی
۴	۱.۴.۱ مقداردهی اولیه
۵	۲.۴.۱ محسابه
۶	۳.۴.۱ آموزش
۲	مشاهدات
۸	۱.۲ مدل ساده با سه نورون در لایه مخفی
۸	۲.۲ تغییر تعداد نورون‌ها در لایه مخفی
۹	۳.۲ استفاده از دسته‌های کوچک‌تر در آموزش شبکه
۱۰	۴.۲ تبرید درجه یادگیری به تدریج
۱۱	۵.۲ تغییر تابع فعال‌سازی از sigmoid به tanh
۱۲	۶.۲ داده‌هایی با سه کلاس به جای دو کلاس
۱۳	۷.۲ تغییر تعداد لایه‌های مخفی

# ۱ مقدمات و ابزارها

مواردی که در این بخش به آن‌ها می‌پردازیم در فایل utils.py پیاده‌سازی شده‌اند.  
همچنین تمام کدهای مربوط به این پروژه در مخزن گیت زیر موجود می‌باشد.

<https://github.com/kmirzavaziri/neural-network>

## ۱.۱ توابع کمکی

در پیاده‌سازی این پروژه به تعدادی تابع کمکی نیاز خواهیم داشت که همه آن‌ها را در کلاسی به اسم helpers پیاده‌سازی می‌کنیم.

در پیاده‌سازی محاسبات نورون‌ها به تابع sigmoid برای مشاهده شماره ۵، و softmax برای محاسبات لایه آخر نیاز خواهیم داشت. این تابع را به صورت زیر پیاده‌سازی می‌کنیم.

```
def sigmoid(x):  
    return 1 / (1 + np.exp(-np.array(x)))  
  
def softmax(x):  
    return np.exp(x) / np.sum(np.exp(x))
```

همچنین نیاز داریم تا خطای را بر اساس فرمولی که در صورت پروژه داده شده محاسبه کنیم که برای این کار نیز تابع کمکی زیر را در نظر می‌گیریم.

```
def loss(y, nno):  
    return -sum([np.log(nno[n][y[n]]) for n in range(len(y))]) / len(y)
```

این تابع خروجی شبکه عصبی که برداری از احتمالات است را به همراه پاسخ درست دریافت می‌کند و برای هر مورد، احتمالی که شبکه عصبی برای پاسخ درست خروجی داده را در نظر می‌گیرد و لگاریتم این مقادیر را جمع می‌زند که با توجه به علامت منفی باعث می‌شود هرچه مقدار loss کمتر باشد، پاسخ بهتری داشته باشیم.  
در نهایت دو تابع کمکی برای ایجاد داده‌ها از طریق کتابخانه sklearn ایجاد می‌کنیم. یکی از این تابع همان داده‌های موجود در صورت پروژه یعنی make\_moons و دیگری داده‌هایی با سه کلاس برای مشاهده شماره ۶ را خروجی می‌دهد. این تابع قادرند مجموعه داده‌های دریافت شده از کتابخانه sklearn را به دو بخش train و test بشکنند تا بتوانیم قدرت مدل شبکه عصبی را به درستی اندازه‌گیری کنیم.

```
def dataset(n, m):  
    np.random.seed(0)  
    x, y = datasets.make_moons(n + m, noise=0.20)  
    return x[:n], y[:n], x[n:], y[n:]  
  
def dataset_3(n, m):  
    np.random.seed(0)  
    x, y = datasets.make_classification(n + m, 2, n_classes=3, n_redundant  
=0, n_clusters_per_class=1)
```

```
return x[:n], y[:n], x[n:], y[n:]
```

## ۲۰۱ ترسیم گر

کلاسی به اسم Visualizer پیاده‌سازی می‌کنیم که وظیفه ترسیم داده‌ها و کلاس‌های آن‌ها را دارد. این کلاس از کتابخانه matplotlib برای ترسیم استفاده می‌کند و کد آن در فایل utils.py موجود است اما چون مربوط به موضوع پروژه نمی‌باشد به آن نمی‌پردازیم.

## ۳۰۱ نورون

می‌دانیم یک شبکه عصبی شامل تعدادی لایه است که در هر لایه تعدادی نورون وجود دارد. پس کلاسی به اسم Neuron برای پیاده‌سازی نورون‌ها که همان واحدهای محاسباتی شبکه عصبی می‌باشند تعریف می‌کنیم. این کلاس ساده تنها شامل دو متدهای `forward` و `backward` است که برای مقداردهی نورون به کار می‌روند تنها تعداد ورودی‌های نورون را به عنوان ورودی دریافت می‌کنند. دو بردار `w` و `b` را به طول تعداد ورودی‌های نورون تعریف می‌کنند.

```
def __init__(self, input_size):
    self.input_size = input_size
    self.w = list(np.random.randn(self.input_size))
    self.b = [0.0] * self.input_size
```

همانطور که می‌بینیم مقدار بردار `b` را در ابتدا صفر و مقدار بردار `w` را مقادیر تصادفی با توزیع نرمال در نظر می‌گیریم.

متدهایی موظف است سیگنال ورودی به نورون که یک بردار به طول تعداد ورودی‌های نورون است را بگیرد و با توجه به بردارهای `w` و `b` و به کمک فرمول زیر، حاصل جمع این مقادیر را خروجی بدهد.

$$\sum_{i=0}^{\text{input\_size}-1} x_i w_i + b_i.$$

```
def summation(self, signal):
    return sum([signal[i] * self.w[i] + self.b[i] for i in range(self.input_size)])
```

## ۴۰۱ شبکه عصبی

برای پیاده‌سازی شبکه عصبی، کلاسی به اسم NeuralNetwork تعریف می‌کنیم. از این کلاس انتظار داریم تا با دریافت یک سیگنال ورودی، سیگنالی را خروجی بدهد، و همچنین بتوانیم با داده‌هایی که پاسخ آن‌ها را از قبل می‌دانیم، آن را آموزش دهیم. پس دو متدهای `compute` و `train` باید روی این کلاس تعریف شده باشد. اما تعدادی توابع کمکی نیز برای سادگی کار روی این کلاس تعریف کردہ‌ایم که در ادامه هر یک از این توابع را بررسی می‌کنیم.

## ۱۰۴۰۱ مقداردهی اولیه

ابتدا لازم است پارامترهایی را در نظر بگیریم و با توجه به این پارامترها شبکه عصبی را راهاندازی کنیم. شبکه عصبی را طوری پیاده‌سازی می‌کنیم که موارد زیر همگی قابل تنظیم باشند.

- تعداد لایه‌ها و تعداد نورون‌ها در هر لایه
- درجه یادگیری  $\epsilon$
- قدرت تنظیم  $\lambda$
- سایز دسته‌ها در gradient descent
- درجه تبرید
- تابع فعال‌سازی

با این قابلیت انعطاف برای شبکه عصبی می‌توانیم تمام مشاهداتی که در صورت پروژه خواسته شده را به سادگی انجام دهیم. تعداد لایه‌ها و تعداد نورون‌های هر لایه را به صورت یک لیست از اعداد ورودی می‌گیریم که هر درایه آن تعداد نورون‌های آن لایه را نشان می‌دهد و بدیهتاً درایه اول نشان‌دهنده اندازه بردار ورودی شبکه عصبی و درایه آخر نشان‌دهنده اندازه بردار خروجی می‌باشد. متغیری به اسم layers روی شبکه عصبی تعریف می‌کنیم که در هر درایه آن به تعداد لازم نورون ایجاد می‌کنیم و در آن قرار می‌دهیم. اما توجه کنیم که برای لایه صفرم یعنی لایه ورودی نورونی ایجاد نمی‌کنیم چرا که محاسباتی در این لایه انجام نمی‌شود. پس کد تابع مقداردهی اولیه به صورت زیر خواهد شد.

```
def __init__(self, layer_sizes, epsilon, r_lambda, batch_size=None, annealing_degree=0, activation_function=ACTIVATION.TANH):  
    self.layer_sizes = layer_sizes  
    self.layers = []  
    for i in range(1, len(self.layer_sizes)):  
        self.layers.append([Neuron(self.layer_sizes[i - 1]) for j in range(self.layer_sizes[i])])  
  
    self.epsilon = epsilon  
    self.r_lambda = r_lambda  
    self.batch_size = batch_size  
    self.annealing_degree = annealing_degree  
    self.activation_function = activation_function
```

که در آن دو حالت ACTIVATION.SIGMOID و ACTIVATION.TANH به کمک کلاس زیر برای تابع فعال‌سازی تعریف شده‌اند.

```
class ACTIVATION:  
    TANH = 'tanh'  
    SIGMOID = 'sigmoid'
```

متد compute را برای محاسبه خروجی از روی ورودی داده شده تعریف می‌کنیم. اما در این متد نیاز به تابع فعالسازی در هر لایه داریم که به کمک یک متد کمکی آن را به دست می‌آوریم.

```
def activation_functions(self):
    if self.activation_function == ACTIVATION.TANH:
        af = np.tanh
    elif self.activation_function == ACTIVATION.SIGMOID:
        af = helpers.sigmoid

    return [af] * (len(self.layers) - 1) + [helpers.softmax]
```

این متد کمکی لیستی از توابع برمی‌گرداند که هر درایه آن تابع فعالسازی یک لایه است. (باز توجه کنیم که این لایه‌ها از لایه یکم شروع می‌شوند چرا که در لایه صفرم محاسباتی اتفاق نمی‌افتد) در این لیست تمام لایه‌های از تابع `tanh` یا `sigmoid` با توجه به تنظیمات شبکه پیروی می‌کنند اما لایه آخر از تابع `softmax`. با داشتن این متد می‌توانیم متد محاسبه را به راحتی پیاده‌سازی کنیم.

```
def compute(self, signal):
    self.signals_history = [signal]
    activation_functions = self.activation_functions()
    for i in range(len(self.layers)):
        signal = list(activation_functions[i])([neuron.summation(signal) for
                                                neuron in self.layers[i]]))
        self.signals_history.append(signal)
    return signal
```

در این متد سیگنال ورودی و خروجی هر لایه را در متغیری به اسم `signals_history` نگه می‌داریم تا بعداً در فرایند آموزش از آن استفاده کنیم. همچنین یک متد کمکی به اسم predict تعریف می‌کنیم که با دریافت ورودی `x` که لیستی از نقاط است، لیستی از کلاس پیش‌بینی شده برای هر نقطه را خروجی می‌دهد.

```
def predict(self, x):
    y_pred = []
    self.outputs_history = []
    for node in x:
        output = self.compute(node)
        self.outputs_history.append(output)
        y_pred.append(output.index(max(output)))
    return y_pred
```

این متد تنها در یک حلقه خروجی شبکه عصبی را برای هر نقطه محاسبه کرده و سپس شماره نورون خروجی با بیشترین سیگنال را به عنوان کلاس آن نقطه در نظر می‌گیرد. (چرا که تعبیر ما از خروجی‌ها، احتمال تعلق نقطه به آن دسته می‌باشد).

## ۳.۴.۱ آموزش

متد train را برای آموزش شبکه عصبی تعریف می‌کنیم.

این متد لیستی از بردارها (x) به همراه کلاسی که هر بردار متعلق به آن است (y) دریافت می‌کند.

سپس به تعداد محدودی خروجی شبکه را روی این نقاط به دست می‌آورد و هر بار با مقایسه خروجی شبکه با

پاسخ صحیح و به کمک الگوریتم backpropagation پارامترهای نورون‌ها را اصلاح می‌کند.

همچنین اگر یک شیء ترسیم‌کننده به این متد ورودی داده شده بود، از هر ۳۰۰ باری که حلقه اجرا می‌شود یک

بار کلاس‌های به دست آمده در این مرحله را برای استفاده در گزارش ترسیم می‌کند.

پس کد این متد به صورت زیر خواهد بود.

```
def train(self, x, y, max_iterations=1200, visualizer=None):
    if len(x) != len(y):
        raise Exception('x and y must have the same length.')
    batch_size = self.batch_size if self.batch_size is not None else len(x)
    for i in range(max_iterations + 1):
        outputs = []
        signals_histories = []
        batch_counter = 0
        for node in x:
            outputs.append(self.compute(node))
            signals_histories.append(self.signals_history)
            batch_counter += 1
            if batch_counter % batch_size == 0:
                self.backpropagate(signals_histories, y[batch_counter -
batch_size:batch_counter])
                signals_histories = []
                self.epsilon *= 1 - self.annealing_degree

        if visualizer and i % 300 == 0:
            y_pred = [output.index(max(output)) for output in outputs]
            visualizer.add(x, y_pred, title=f'Iteration {i}\nLoss {helpers.
loss(y, outputs)}')
```

توجه کنیم که با توجه به متغیر batch\_size ممکن است در هر دور که روی کل نقاط طی می‌شود، چند بار متد backpropagate فراخوانی شود.

همچنین هر بار تمام سیگنال‌های ردوبدل شده در لایه‌های میانی شبکه را برای تک‌تک نقاط در یک لیست ذخیره می‌کنیم تا متد backpropagate از این مقادیر استفاده کند.

از طرفی هر بار درجه یادگیری  $\epsilon$  را با توجه به درجه تبرید کم می‌کنیم. اگر درجه تبرید صفر باشد تغییری حاصل نمی‌شود اما در غیر این صورت این مقدار اندازی کم می‌شود.

اما متد backpropagate که از آن به صورت یک متد کمکی استفاده کردیم به صورت زیر تعریف شده.

```
def backpropagate(self, raw_sh, y):
```

```

w = []
b = []
for j in range(len(self.layers)):
    w.append(np.array([neuron.w for neuron in self.layers[j]]).T)
    b.append(np.array([neuron.b for neuron in self.layers[j]]).T)

signals_histories = [
    np.array([raw_sh[n][l] for n in range(len(raw_sh))])
    for l in range(len(self.layer_sizes))
]

delta = self.deltas(signals_histories, w, y)
for j in range(len(self.layers) - 1, -1, -1):
    w[j] += -self.epsilon * (signals_histories[j].T.dot(delta[j]) +
    self.r_lambda * w[j])
    b[j] += -self.epsilon * np.sum(delta[j], axis=0)
    for k in range(len(self.layers[j])):
        self.layers[j][k].w = w[j].T[k]
        self.layers[j][k].b = b[j].T[k]

```

این متده از متده کمکی دیگری برای محاسبه دلتا استفاده میکند و در آن تغییرات پارامترهای هر نورون به کمک کدی که در صورت پروژه موجود بود با کمی تغییر محاسبه میشود.  
اما متده `deltas` که از آن به صورت یک متده کمکی استفاده کردیم به صورت زیر تعریف شده.

```

def deltas(self, signals_histories, w, y):
    deltas = [0] * len(self.layers)

    deltas[-1] = np.array(signals_histories[-1])
    deltas[-1][range(len(y)), y] -= 1

    for i in range(len(self.layers) - 2, -1, -1):
        if self.activation_function == ACTIVATION.TANH:
            deltas[i] = deltas[i + 1].dot(w[i + 1].T) * (1 - np.power(
                signals_histories[i + 1], 2))
        elif self.activation_function == ACTIVATION.SIGMOID:
            deltas[i] = deltas[i + 1].dot(w[i + 1].T) * (signals_histories[
                i + 1] * (1 - signals_histories[i + 1]))

    return deltas

```

که در آن مقدار  $\delta$  در هر لایه به کمک مقدار دلتا در لایه بعدی محاسبه میشود و حلقه به صورت برعکس یعنی از لایه آخر شروع میکند. مقدار دلتا در لایه آخر به کمک فرمول زیر به دست میآید.

$$\delta_{-1} = \hat{y} - y.$$

و مقدار دلتا در لایه‌های قبلی به صورت زیر به دست می‌آید.

$$\delta_i = \text{afd}_i \cdot \delta_{i+1} \cdot w_{i+1}^T$$

اما با توجه به این که تابع sigmoid یا  $\tanh$  در تنظیمات شبکه انتخاب شده باشد، مقدار  $\text{afd}$  متفاوت است و به صورت زیر می‌باشد.

$$\text{afd}_i = \begin{cases} 1 - \text{sh}_i & \text{af} = \tanh \\ \text{sh}_i(1 - \text{sh}_i) & \text{af} = \text{sigmoid} \end{cases}$$

که در آن  $\text{sh}_i$  سیگنال خروجی لایه  $i$ -ام می‌باشد.

## ۲ مشاهدات

### ۱۰۲ مدل ساده با سه نورون در لایه مخفی

کد زیر که در فایل `py.1` موجود است برای راهاندازی و آموزش یک مدل ساده با سه نورون در یک لایه مخفی می‌نویسیم.

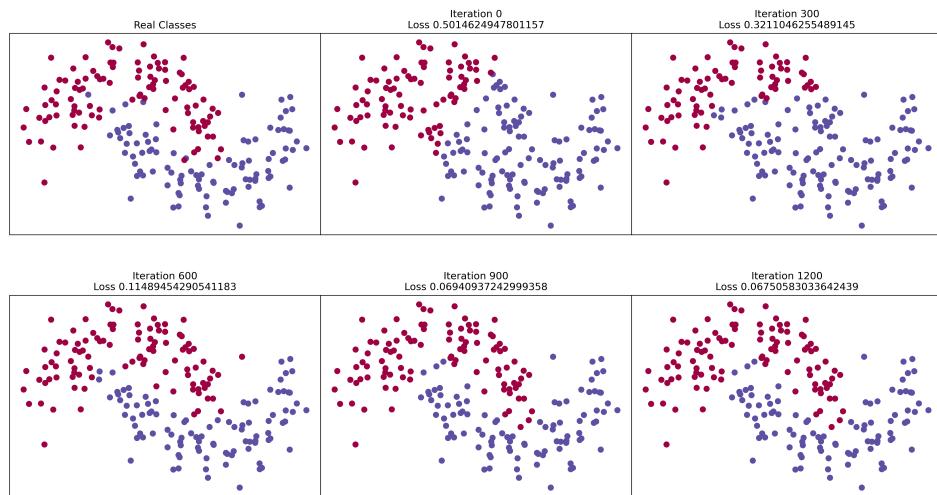
```
from utils import *
visualizer = Visualizer(2, 3)
x, y, _, _ = helpers.dataset(200, 0)
visualizer.add(x, y, title='Real Classes')

# Initiate
PARAMETERS_COUNT = 2
HIDDEN_LAYER_COUNT = 3
CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
nn = NeuralNetwork([PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT],
                    EPSILON, R_LAMBDA)

# Train
nn.train(x, y, visualizer=visualizer)

visualizer.show('1.png')
```

تعداد پارامترهای هر نقطه برابر دو است یعنی نقاط در یک فضای دو بعدی قرار دارند و تعداد نورون های ورودی نیز برابر دو است. همچنین دو کلاس داریم پس تعداد نورون های خروجی نیز برابر دو است. تعداد نورون های لایه مخفی را سه در نظر می گیریم و پارامترهای شبکه را نیز بر اساس صورت پروژه تعريف می کنیم. خروجی به صورت زیر خواهد شد.



## ۲.۰۲ تغییر تعداد نورون ها در لایه مخفی

کد زیر در فایل `py.2` موجود است. در این کد در یک حلقه، هر بار شبکه عصبی را با تعداد متفاوتی نورون در لایه مخفی تعريف می کنیم و برای سنجیدن مدل از داده های `test` به جای `train` استفاده می کنیم.

```
from utils import *
visualizer = Visualizer(2, 4)
x_train, y_train, x_test, y_test = helpers.dataset(200, 400)
visualizer.add(x_test, y_test, title='Real Classes')

# Iterate and predict over some different hidden layer sizes
PARAMETERS_COUNT = 2
CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
for HIDDEN_LAYER_COUNT in [1, 2, 3, 4, 5, 20, 40]:
    print(HIDDEN_LAYER_COUNT)
    nn = NeuralNetwork([PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT],
    EPSILON, R_LAMBDA)
    nn.train(x_train, y_train)
    y_pred = nn.predict(x_test)
    visualizer.add(
```

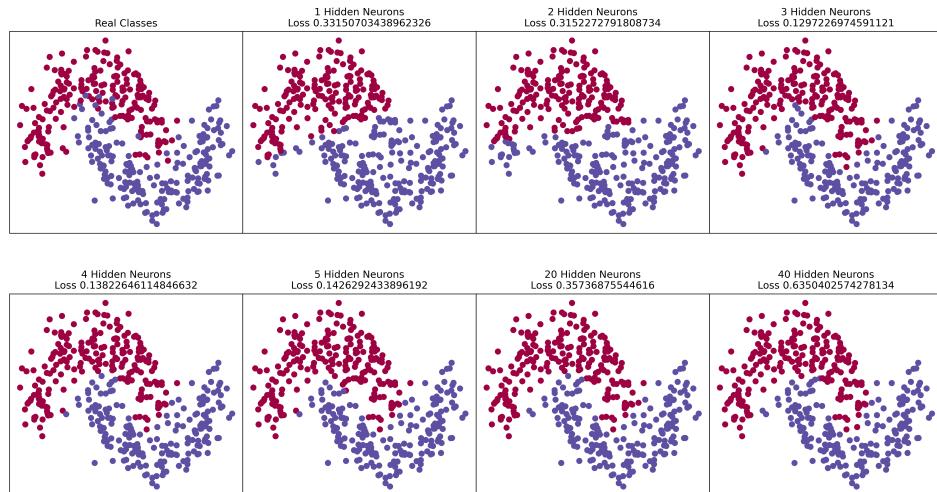
```

        x_test, y_pred,
        title=f'{HIDDEN_LAYER_COUNT} Hidden Neurons\nLoss {helpers.loss(
y_test, nn.outputs_history)}'
    )

visualizer.show('2.png')

```

خروجی به صورت زیر خواهد شد.



به نظر می‌رسد که همان سه نورون در لایه مخفی بهترین انتخاب باشد.

## ۳.۲ استفاده از دسته‌های کوچک‌تر در آموزش شبکه

کد زیر در فایل 3.py موجود است. در این کد در یک حلقه، هر بار تعداد دسته‌ها را متفاوت در نظر می‌گیریم. مانند مورد قبل برای سنجیدن مدل از داده‌های test به جای train استفاده می‌کنیم.

```

from utils import *
visualizer = Visualizer(2, 3)
x_train, y_train, x_test, y_test = helpers.dataset(200, 400)
visualizer.add(x_test, y_test, title='Real Classes')

# Iterate and predict over some different batch sizes
PARAMETERS_COUNT = 2
HIDDEN_LAYER_COUNT = 3
CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
for BATCH_SIZE in [10, 20, 50, 100, 200]:

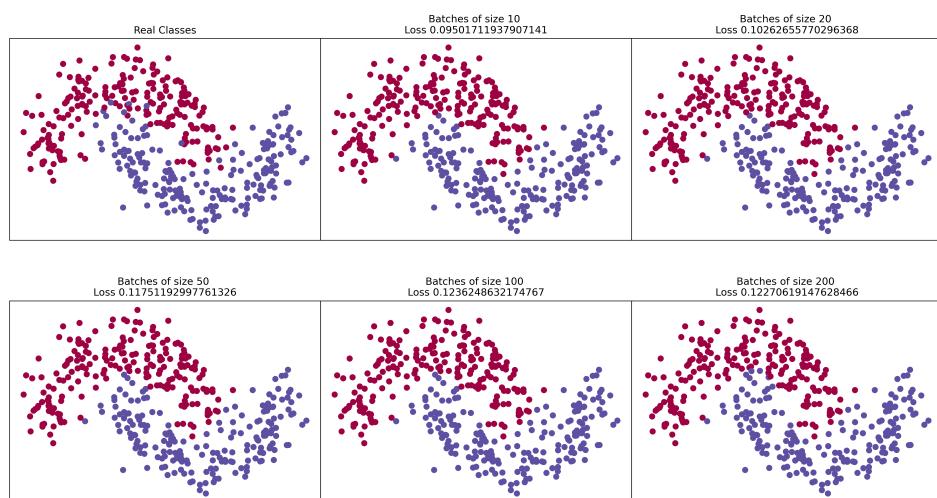
```

```

print(BATCH_SIZE)
nn = NeuralNetwork(
    [PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT], EPSILON,
    R_LAMBDA,
    batch_size=BATCH_SIZE
)
nn.train(x_train, y_train)
y_pred = nn.predict(x_test)
visualizer.add(
    x_test, y_pred,
    title=f'Batches of size {BATCH_SIZE} \nLoss {helpers.loss(y_test, nn.outputs_history)}'
)
visualizer.show('3.png')

```

خروجی به صورت زیر خواهد شد.



همانطور که در صورت پروژه آمده، به نظر می‌رسد استفاده از دسته‌های کوچک ۱۰-تایی خیلی بهتر از دسته کامل ۲۰۰-تایی عمل می‌کند.

## ۴۰۲ تبرید درجه یادگیری به تدریج

کد زیر در فایل `py.4` موجود است. در این کد در یک حلقه، هر بار درجه تبرید را متفاوت در نظر می‌گیریم. مانند موارد قبل برای سنجیدن مدل از داده‌های `test` به جای `train` استفاده می‌کنیم.

```

from utils import *
visualizer = Visualizer(2, 3)

```

```

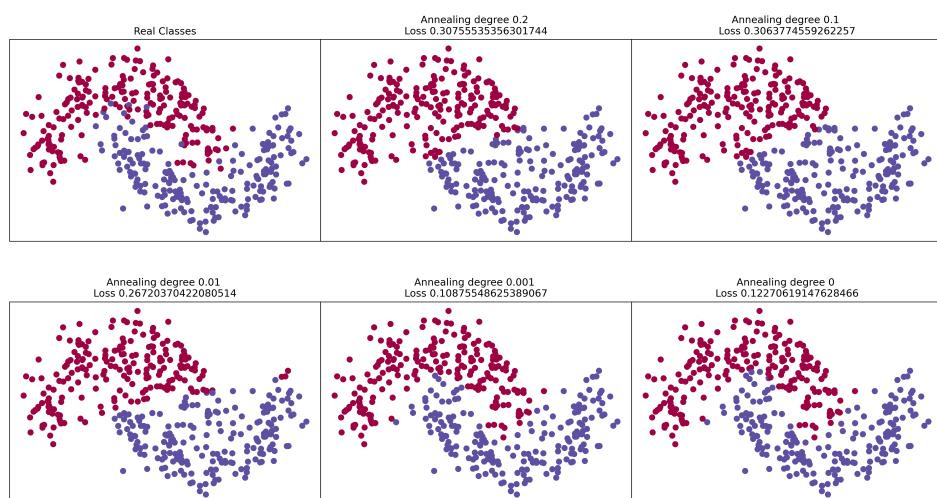
x_train, y_train, x_test, y_test = helpers.dataset(200, 400)
visualizer.add(x_test, y_test, title='Real Classes')

# Iterate and predict over some different annealing degrees
PARAMETERS_COUNT = 2
HIDDEN_LAYER_COUNT = 3
CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
for ANNEALING_DEGREE in [.2, .1, .01, .001, 0]:
    print(ANNEALING_DEGREE)
    nn = NeuralNetwork(
        [PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT], EPSILON,
        R_LAMBDA,
        annealing_degree=ANNEALING_DEGREE
    )
    nn.train(x_train, y_train)
    y_pred = nn.predict(x_test)
    visualizer.add(
        x_test, y_pred,
        title=f'Annealing degree {ANNEALING_DEGREE} \nLoss {helpers.loss(y_test, nn.outputs_history)}'
    )

visualizer.show('4.png')

```

خروجی به صورت زیر خواهد شد.



در حالاتی که درجه تبرید را بزرگ گرفتیم نتیجه از حالت معمولی با درجه صفر بدتر شده، اما می‌بینیم در حالتی که این درجه را خیلی کوچک و برابر ۰.۰۰۱ در نظر گرفتیم نتیجه بهتری حاصل شده.

## ۵.۲ تغییر تابع فعال‌سازی از $\tanh$ به sigmoid

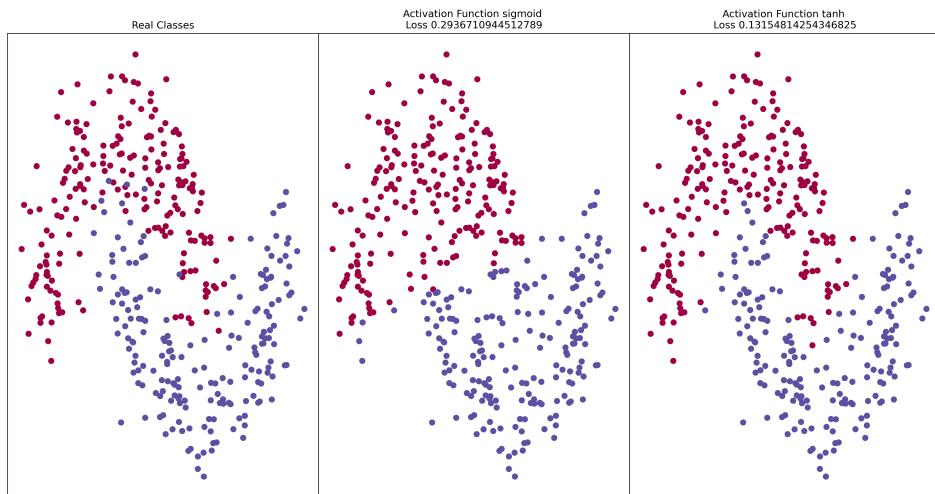
کد زیر در فایل ۵.py موجود است. در این کد یک بار با تابع فعال‌سازی  $\tanh$  و بار دیگر با تابع فعال‌سازی sigmoid شبکه عصبی را ایجاد می‌کنیم (تابع و فرمول‌های مربوط به هر یک در بخش معرفی شبکه عصبی آمده.) و مانند موارد قبل برای سنجیدن مدل از داده‌های test به جای train استفاده می‌کنیم.

```
from utils import *
visualizer = Visualizer(1, 3)
x_train, y_train, x_test, y_test = helpers.dataset(200, 400)
visualizer.add(x_test, y_test, title='Real Classes')

# Iterate and predict over different activation functions
PARAMETERS_COUNT = 2
HIDDEN_LAYER_COUNT = 3
CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
for ACTIVATION_FUNCTION in [ACTIVATION.SIGMOID, ACTIVATION.TANH]:
    print(ACTIVATION_FUNCTION)
    nn = NeuralNetwork(
        [PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT], EPSILON,
        R_LAMBDA,
        activation_function=ACTIVATION_FUNCTION
    )
    nn.train(x_train, y_train)
    y_pred = nn.predict(x_test)
    visualizer.add(
        x_test, y_pred,
        title=f'Activation Function {ACTIVATION_FUNCTION} \nLoss {helpers.loss(y_test, nn.outputs_history)}'
    )

visualizer.show('5.png')
```

خروجی به صورت زیر خواهد شد.



به نظر می‌رسد در این مسئله تابع  $\tanh$  بـهتر از  $\text{sigmoid}$  عمل می‌کند که با توجه به صورت پروژه همین انتظار می‌رود.

## ۶.۲ داده‌هایی با سه کلاس به جای دو کلاس

کد زیر در فایل `6.py` موجود است. در این کد داده‌هایی با سه کلاس تولید می‌کنیم که توضیح آن در بخش توابع کمکی آمده. مدل را آموزش می‌دهیم و مانند موارد قبل برای سنجیدن مدل از داده‌های `test` به جای `train` استفاده می‌کنیم.

```
from utils import *
visualizer = Visualizer(3, 3)
x_train, y_train, x_test, y_test = helpers.dataset_3(200, 400)
visualizer.add(x_train, y_train, title='Real Classes (Train)')

# Initiate
PARAMETERS_COUNT = 2
HIDDEN_LAYER_COUNT = 3
CLASSES_COUNT = 3
EPSILON = .01
R_LAMBDA = .01

nn = NeuralNetwork([PARAMETERS_COUNT, HIDDEN_LAYER_COUNT, CLASSES_COUNT],
                   EPSILON, R_LAMBDA)

# Train
nn.train(x_train, y_train, max_iterations=1500, visualizer=visualizer)
```

```

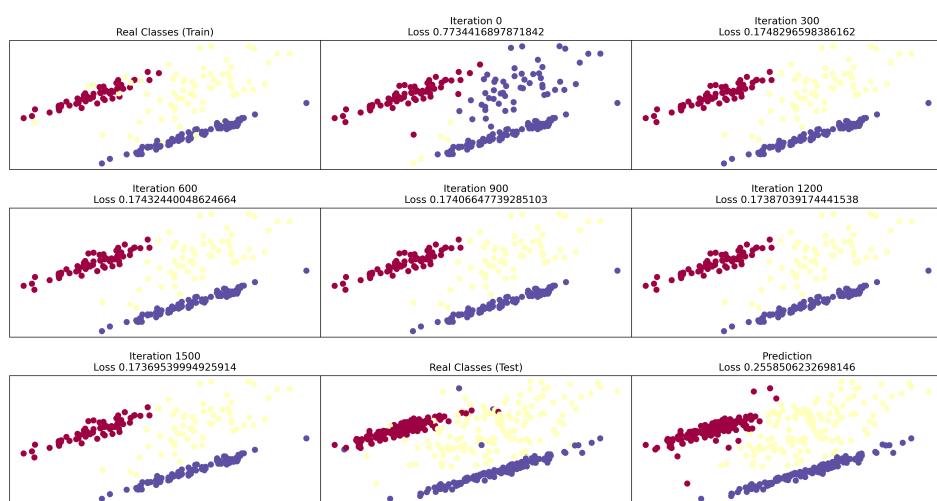
visualizer.add(x_test, y_test, title='Real Classes (Test)')

# Predict
y_pred = nn.predict(x_test)
visualizer.add(
    x_test, y_pred,
    title=f'Prediction\nLoss {helpers.loss(y_test, nn.outputs_history)}'
)

visualizer.show('6.png')

```

خروجی به صورت زیر خواهد شد.



در تصویر بالا ترسیم‌هایی از مراحل آموزش مدل، داده‌های اصلی test و train و همچنین کلاس‌های پیش‌بینی شده برای داده test قابل رؤیت است.

## ۷.۲ تغییر تعداد لایه‌های مخفی

کد زیر در فایل `7.py` موجود است. در این کد در یک حلقه، هر بار لایه‌های مخفی را متفاوت در نظر می‌گیریم. مانند موارد قبل برای سنجیدن مدل از داده‌های test به جای train استفاده می‌کنیم.

```

from utils import *
visualizer = Visualizer(2, 3)
x_train, y_train, x_test, y_test = helpers.dataset(200, 400)
visualizer.add(x_test, y_test, title='Real Classes')

# Iterate and predict over some different hidden layers
PARAMETERS_COUNT = 2

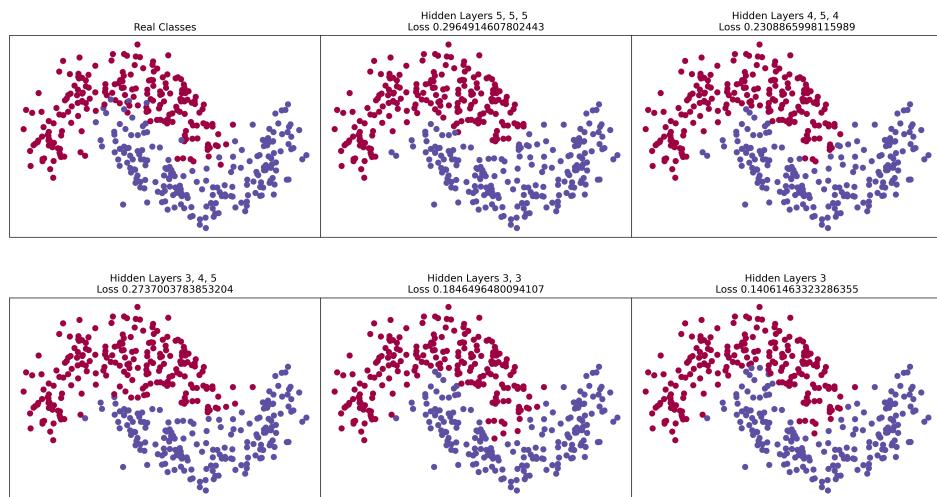
```

```

CLASSES_COUNT = 2
EPSILON = .01
R_LAMBDA = .01
for HIDDEN_LAYERS in [[5, 5, 5], [4, 5, 4], [3, 4, 5], [3, 3], [3]]:
    print(HIDDEN_LAYERS)
    nn = NeuralNetwork([PARAMETERS_COUNT, *HIDDEN_LAYERS, CLASSES_COUNT],
    EPSILON, R_LAMBDA)
    nn.train(x_train, y_train)
    y_pred = nn.predict(x_test)
    visualizer.add(
        x_test, y_pred,
        title=f'Hidden Layers {" ".join(map(str, HIDDEN_LAYERS))} \nLoss {helpers.loss(y_test, nn.outputs_history)}'
    )
visualizer.show('7.png')

```

خروجی به صورت زیر خواهد شد.



همانطور که در تصویر قابل مشاهده است به نظر می رسد همان یک لایه مخفی با سه نورون بهتر از سایر حالات عمل می کند.