دانشگاه تهران
دانشکده مهندسی برق و
کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق
تمرین اول

متین بذرافشان - 810100093
شهریار عطار - 810100186

فهرست

پرسش 1. Mcculloch Pitts

1-1. شبکه محاسبه مکمل 2

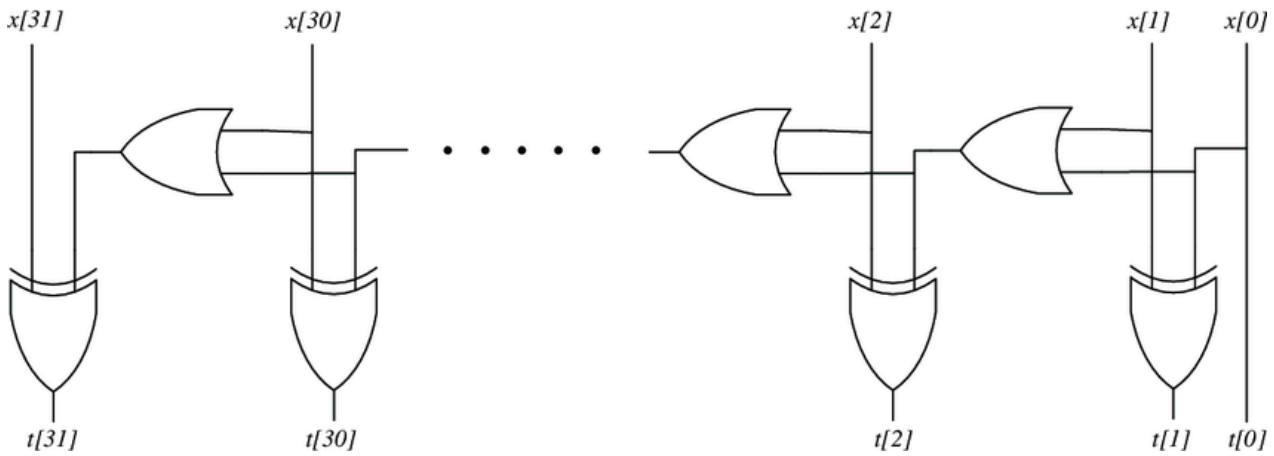
مدار مکمل دو

مدار مکمل دو یک سیستم عددی است که در آن، اعداد با استفاده از مکمل دو نمایش داده می‌شوند. این سیستم عددی برای نمایش اعداد منفی در کامپیوترها استفاده می‌شود.

برای محاسبه مکمل دو یک عدد، ابتدا تمام بیت‌های آن عدد را تغییر می‌دهیم (یعنی 1 به 0 و 0 به 1 تبدیل می‌شود) و سپس یک را به نتیجه اضافه می‌کنیم. برای مثال، مکمل دو عدد 5 در سیستم عددی 8 بیتی به شرح زیر است:

1. نمایش باینری عدد 5 در سیستم 8 بیتی: 00000101
2. تغییر بیت‌ها: 11111010
3. اضافه کردن یک: 11111011

پس مکمل دو عدد 5 در سیستم عددی 8 بیتی برابر است با 11111011. این روش برای جمع و تفریق اعداد در کامپیوترها بسیار مفید است و سیستم‌های کامپیوتری بر این مبنا کار می‌کنند. برای کسب بیشتر در مورد این مدار که نسخه اصلاح‌شده مدار مکمل 2 است میتوانید به این لینک مراجعه کنید.



شکل 1-1. مدار اصلاح‌شده مکمل 2

2-1. پیاده‌سازی تئوری شبکه

نورون McCulloch-Pitts:

این نورون که به نام نورون M-P نیز شناخته می‌شود، یک مدل ساده و اولیه از نورون‌های مغز است که در سال 1943 توسط وارن مک‌کالاک و والتر پیتس ارائه شد. این مدل برای شبیه‌سازی عملکرد نورون‌های بیولوژیکی و پیاده‌سازی شبکه‌های عصبی مصنوعی استفاده می‌شود.

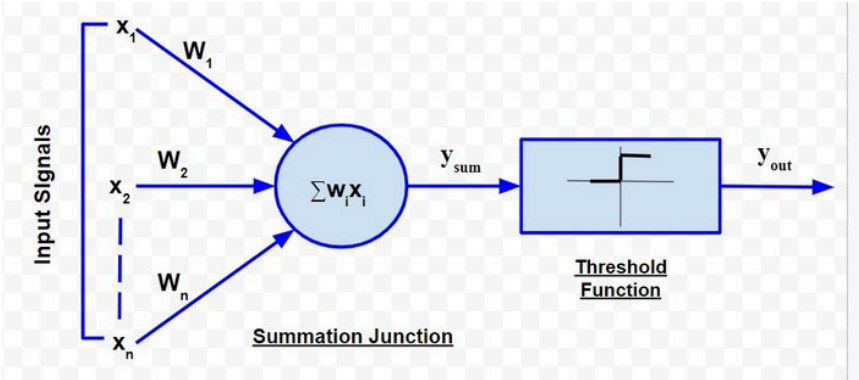
نورون M-P دارای ورودی‌های چندگانه است که هر کدام وزن خاص خود را دارند. این ورودی‌ها به یک تابع جمع‌کننده (Summation Function) منتقل می‌شوند که مجموع وزن‌دار ورودی‌ها را محاسبه می‌کند. اگر این مجموع از یک آستانه خاص (Threshold) بیشتر باشد، نورون خروجی 1 (یا فعال) را تولید می‌کند. در غیر این صورت، خروجی 0 (یا غیرفعال) خواهد بود.

این مدل ساده، با وجود محدودیت‌هایی که دارد، برای درک مفاهیم اولیه شبکه‌های عصبی و طراحی سیستم‌های بسیار ساده که قادر به انجام عملیات منطقی هستند، مفید است. برای کسب بیشتر در مورد این نورون میتوانید به این لینک مراجعه کنید.

نکته‌هایی که در رابطه با این نورون باید به آن دقت شود:

1. فعال‌سازی این نورون به صورت باینری است یعنی در هر لحظه خروجی یا 1 است یا 0، که با توجه به threshold مشخص می‌شود.
2. نورون‌های M-P با یال‌های وزن‌دار و جهت‌دار به هم متصل هستند.
3. به یال‌هایی که وزن مثبت دارند، تحریک کننده (excitatory) و به یال‌هایی با وزن منفی دارند، بازدارنده (inhibitory) می‌گویند. یال‌های تحریک کننده (excitatory) متصل به یک نورون وزن ثابتی دارند.

در شکل زیر فرمول محاسبه این نورون را مشاهده می‌کنید:



شکل 1-2. فرمول محاسبه خروجی نورون M-P

تعدادی از کاربردهای نورون M-P

AND

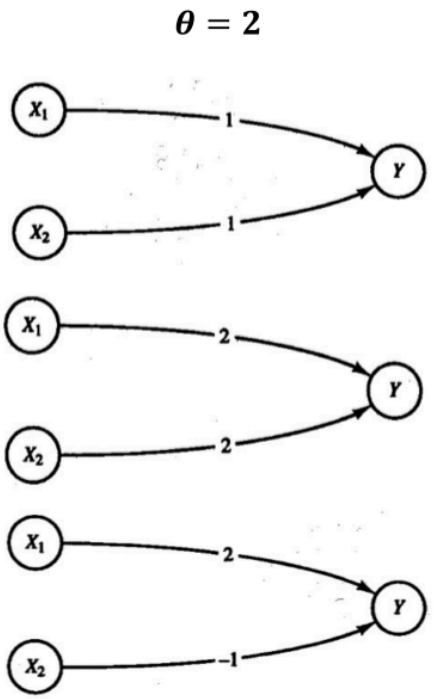
x_1	x_2	→	y
1	1		1
1	0		0
0	1		0
0	0		0

OR

x_1	x_2	→	y
1	1		1
1	0		1
0	1		1
0	0		0

AND Not

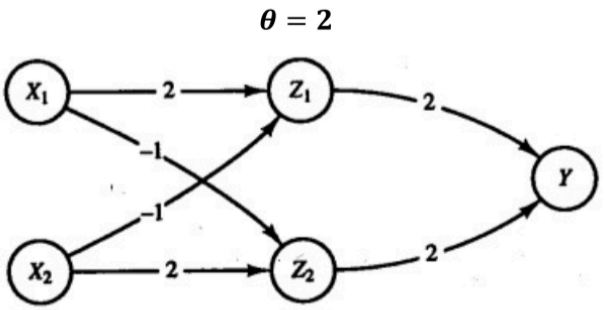
x_1	x_2	→	y
1	1		0
1	0		1
0	1		0
0	0		0



شکل 1-3. پیاده‌سازی گیت‌های منطقی AND و OR و NAND با کمک نورون M-P

- XOR

x_1	x_2	→	y
1	1		0
1	0		1
0	1		1
0	0		0

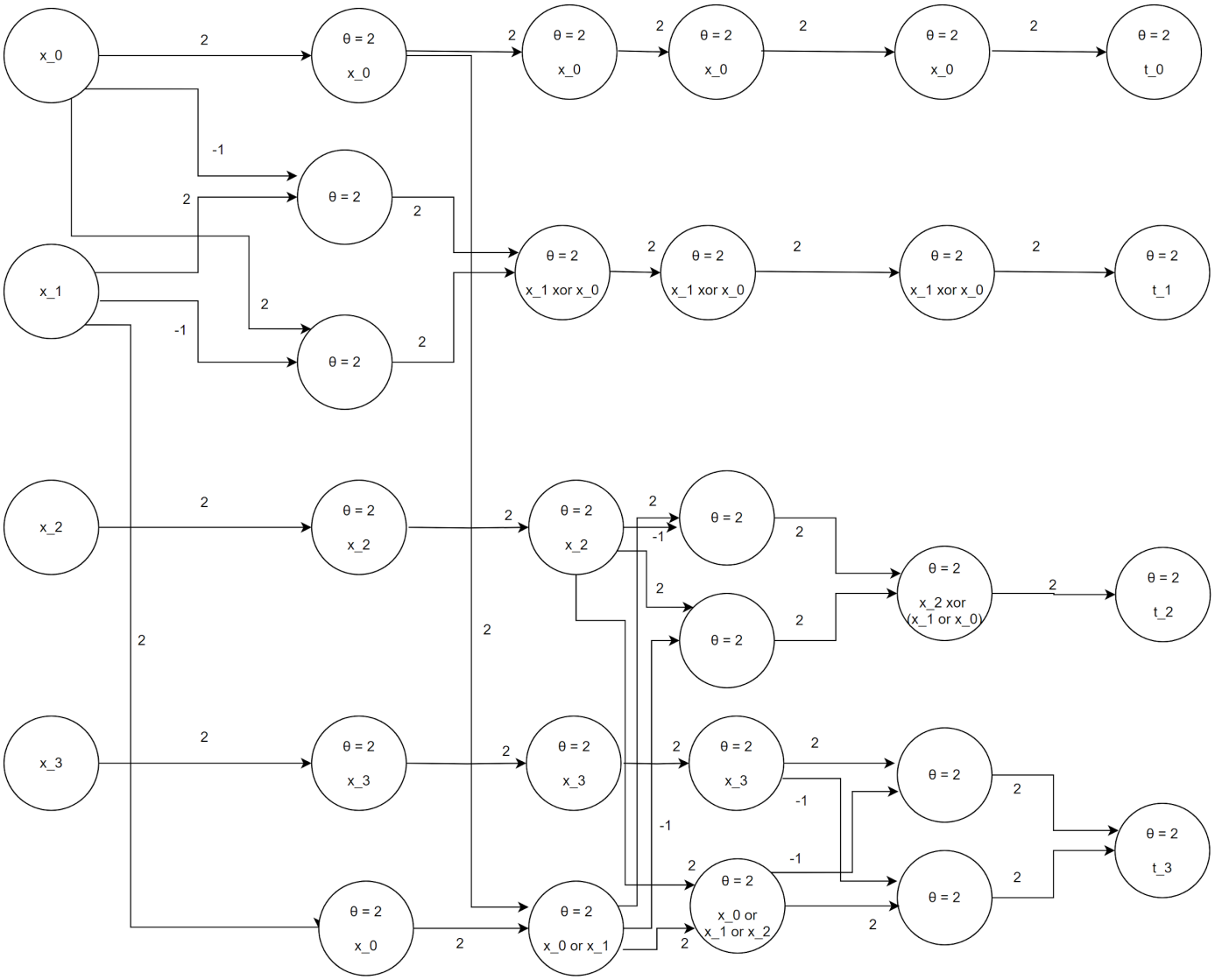


x_1 XOR $x_2 \leftrightarrow (x_1$ AND NOT $x_2)$ OR $(x_2$ AND NOT $x_1)$.

شکل 1-4. پیاده‌سازی گیت منطقی XOR با کمک نورون M-P

طراحی مدل شبکه عصبی

حال سعی میکنیم با کمک این نورون مدار شکل 1-1 را برای یک مدار با 4 بیت ورودی رسم کنیم. این کار را به می‌توان به صورت sequential انجام داد و اول t[0] و بعد به ترتیب t[1] و t[2] و t[3] را محاسبه کرد و یا می‌توان به صورت موازی محاسبات را انجام داد که ما در اینجا این کار را به صورت موازی انجام می‌دهیم.



شکل 1-5. پیاده‌سازی مدار مکمل 2 با کمک نورون M-P

3-1. پیاده‌سازی کد شبکه

پیاده‌سازی دیتاست

شبکه بالا از ساختار پایه‌ای نورون‌ها برای گیت‌های منطقی پایه‌ای (شکل‌های 1-2 و 1-3) استفاده می‌کند و مدار را می‌سازد. برای امتحان کردن شبکه بالا ابتدا تعدادی داده می‌سازیم، برای این کار به شکل زیر عمل می‌کنیم:

```
class TwosComplementDataset:
      def __init__(
      self,
      n_samples: int = 1000,
      n_bits: int = 4,
      transform: Callable = None
      ):
      self._n_samples = n_samples
      self._n_bits = n_bits
      self._transform = transform
      self._generate_data()

      def _generate_data(self):
      self._x = np.array([np.random.randint(0, 2, self._n_bits)
                          for _ in range(self._n_samples)])
      self._y = self._twos_complement(self._x)

      self._x = tensor(self._x, dtype=float32)
      self._y = tensor(self._y, dtype=float32)

      def _twos_complement(self, x: np.ndarray[np.ndarray[int]])
-> np.ndarray[int]:
      return np.array([self._twos_complement_single(x_i)
                       for x_i in x])

      def _twos_complement_single(self, x: np.ndarray[int]) ->
np.ndarray[int]:
      if np.all(x == 0):
            return x
      last_one_idx = np.where(x == 1)[0][-1]
      x = np.copy(x)
      x[:last_one_idx] = 1 - x[:last_one_idx]
      return x
```

```
    def __len__(self):
    return self._n_samples

    def __getitem__(self, idx):
    x = self._x[idx]
    y = self._y[idx]

    if self._transform:
        x, y = self._transform(x, y)

    return x, y

    def get_data(self):
    return self._x, self._y

    def get_dataloader(self, batch_size: int = 32):
    return DataLoader(
        TensorDataset(self._x, self._y),
        batch_size=self._n_samples,
        shuffle=True
    )

    def show_samples(self, n_samples: int = 5):
    x, y = self.get_data()
    for i in np.random.choice(range(len(x)), n_samples):
        print(f"x: {x[i].numpy()}, y: {y[i].numpy()}")
```

تعدادی از نمونه‌های تولید شده توسط این کد:

```
dataset = TwosComplementDataset(n_samples=1000, n_bits=4)
```

```
x: [0. 0. 1. 0.], y: [1. 1. 1. 0.]
x: [0. 1. 0. 0.], y: [1. 1. 0. 0.]
x: [1. 0. 0. 0.], y: [1. 0. 0. 0.]
x: [0. 0. 0. 1.], y: [1. 1. 1. 1.]
x: [0. 0. 0. 1.], y: [1. 1. 1. 1.]
```

پیاده‌سازی شبکه

برای پیاده‌سازی معماری شبکه عصبی از مدل `nn.Module` در کتابخانه `PyTorch` استفاده کردیم، دلیل استفاده از `threshold = 1.5` برای این است که در محاسبات اعداد اعشاری مقداری نویز داریم و در صورت استفاده از مقدار واقعی 2 دقت مدل به شدت پایین می‌آمد. در اینجا لایه‌ها را به صورت `nn.Linear` تعریف کردیم و پس از یک بار انجام ضرب ماتریکسی، threshold را روی اعداد به دست امده برای هر نورون اعمال می‌کنیم. برای نحوه پیاده‌سازی بدون استفاده از کتابخانه‌ها می‌توانید به این لینک مراجعه کنید.

```
class TwosComplementModel(nn.Module):
      def __init__(self, n_bits: int = 4, debug: bool = False):
      super(TwosComplementModel, self).__init__()
      self._n_bits = n_bits
      self._debug = debug
      self._threshold = 1.5

      # 4 -> 6 -> 5 -> 6 -> 5 -> 4

      # ------------- Layer 1 -------------
      self.layer_1 = nn.Linear(n_bits, 6)
      self.layer_1.weight = nn.Parameter(
            tensor(
                  [
                  [2, 0, 0, 0],
                  [-1, 2, 0, 0],
                  [2, -1, 0, 0],
                  [0, 0, 2, 0],
                  [0, 0, 0, 2],
                  [0, 2, 0, 0],
                  ], dtype=float32
            )
      )
      # ---------------------------------
```

```
    # ------------- Layer 2 -------------
    self.layer_2 = nn.Linear(6, 5)
    self.layer_2.weight = nn.Parameter(
        tensor(
            [
            [2, 0, 0, 0, 0, 0],
            [0, 2, 2, 0, 0, 0],
            [0, 0, 0, 2, 0, 0],
            [0, 0, 0, 0, 2, 0],
            [2, 0, 0, 0, 0, 2],
            ], dtype=float32
        )
    )
    # ---------------------------------
    # ------------- Layer 3 -------------
    self.layer_3 = nn.Linear(5, 6)
    self.layer_3.weight = nn.Parameter(
        tensor(
            [
            [2, 0, 0, 0, 0],
            [0, 2, 0, 0, 0],
            [0, 0, -1, 0, 2],
            [0, 0, 2, 0, -1],
            [0, 0, 0, 2, 0],
            [0, 0, 2, 0, 2],
            ], dtype=float32
        )
    )
    # ---------------------------------
    # ------------- Layer 4 -------------
    self.layer_4 = nn.Linear(6, 5)
    self.layer_4.weight = nn.Parameter(
        tensor(
            [
            [2, 0, 0, 0, 0, 0],
            [0, 2, 0, 0, 0, 0],
            [0, 0, 2, 2, 0, 0],
            [0, 0, 0, 0, 2, -1],
            [0, 0, 0, 0, -1, 2],
            ], dtype=float32
        )
    )
```

```
        # ------------- Layer 5 -------------
        self.layer_5 = nn.Linear(5, n_bits)
        self.layer_5.weight = nn.Parameter(
            tensor(
                [
                [2, 0, 0, 0, 0],
                [0, 2, 0, 0, 0],
                [0, 0, 2, 0, 0],
                [0, 0, 0, 2, 2],
                ], dtype=float32
            )
        )
        # ---------------------------------

        self._layers = [self.layer_1, self.layer_2, self.layer_3,
self.layer_4, self.layer_5]

        def _apply_threshold(self, x: tensor) -> tensor:
        return where(x >= self._threshold, tensor(1.0),
tensor(0.0))

        def forward(self, x: tensor) -> tensor:
        for i, layer in enumerate(self._layers):
            if self._debug:
                print(f"Layer {i+1}: {x}")
            x = F.relu(layer(x)) # or x = x @ layer.weight.T
            x = self._apply_threshold(x)
        return x
```

```
Accuracy: 100.00 percent
```

همانگونه که دیده می‌شود مدل با دقت خوبی دارد خروجی مورد نظر را تولید می‌کند.

پرسش 2 - حملات خصمانه در شبکه‌های عصبی

2-1. حملات خصمانه

حملات خصمانه در شبکه‌های عصبی مربوط به تلاش برای گمراه کردن یا ایجاد اختلال در عملکرد یک شبکه عصبی است. در این حملات، مهاجمان با ایجاد تغییرات ناچیز ولی دقیق (noise) در داده‌های ورودی، می‌توانند شبکه عصبی را به اشتباه بیاندازند. به عنوان مثال، ممکن است کسی بداند که قرار دادن نوع خاصی از استیکر در یک نقطه خاص روی علامت توقف می‌تواند به طور موثر علامت توقف را برای یک سیستم هوش مصنوعی نامرئی کند.

حملات خصمانه در شبکه‌های عصبی می‌توانند بر اساس میزان اطلاعاتی که حمله‌کننده در اختیار دارد، به سه دسته تقسیم شوند: حملات باکس سفید، باکس مشکی و باکس خاکستری.

1. **حملات باکس سفید (White-Box Hacking):** در این نوع حملات، حمله‌کننده اطلاعات کاملی از سیستم را در اختیار دارد. این اطلاعات می‌تواند شامل جزئیات دقیق معماری شبکه، پارامترها، وزن‌ها و بایاس‌ها باشد. این دسترسی کامل به اطلاعات سیستم به حمله‌کننده اجازه می‌دهد تا حملات بسیار دقیق و خاص را انجام دهد. در این روش معمولا از بازگشت گرادیان برای مهندسی‌سازی حمله استفاده می‌شود.
2. **حملات باکس مشکی (Black-Box Hacking):** در این نوع حملات، حمله‌کننده هیچگونه اطلاعاتی در مورد سیستم هدف ندارد. حمله‌کننده فقط می‌تواند خروجی سیستم را بر اساس ورودی‌هایی که ارائه می‌دهد مشاهده کند. این نوع حملات معمولاً شامل تلاش برای یادگیری خروجی‌های سیستم بر اساس ورودی‌های مختلف و سپس استفاده از این اطلاعات برای ایجاد حملات خصمانه است.
3. **حملات باکس خاکستری (Gray-Box Hacking):** این نوع حملات میانی بین دو نوع قبلی است. در این حملات، حمله‌کننده دارای اطلاعات محدودی در مورد سیستم است، اما نه به اندازه حملات باکس سفید. این می‌تواند شامل داشتن دسترسی به برخی از پارامترها یا بخش‌های خاص از معماری شبکه باشد.

همچنین این حمله‌ها معمولا با هدف‌های زیر انجام می‌شوند:

1. تغییر رفتار مدل به این صورت که به ازای یک داده خاص مدل رفتار مشخصی انجام دهد.
2. به منظور کاهش دقت مدل برای اینکه مدل لیبل‌ها را درست تشخیص ندهد.

در این تمرین با حمله باکس سفیدی را به منظور کاهش دقت مدل انجام می‌دهیم.

2-2. آشنایی با مجموعه دادگان

دیتاست MNIST

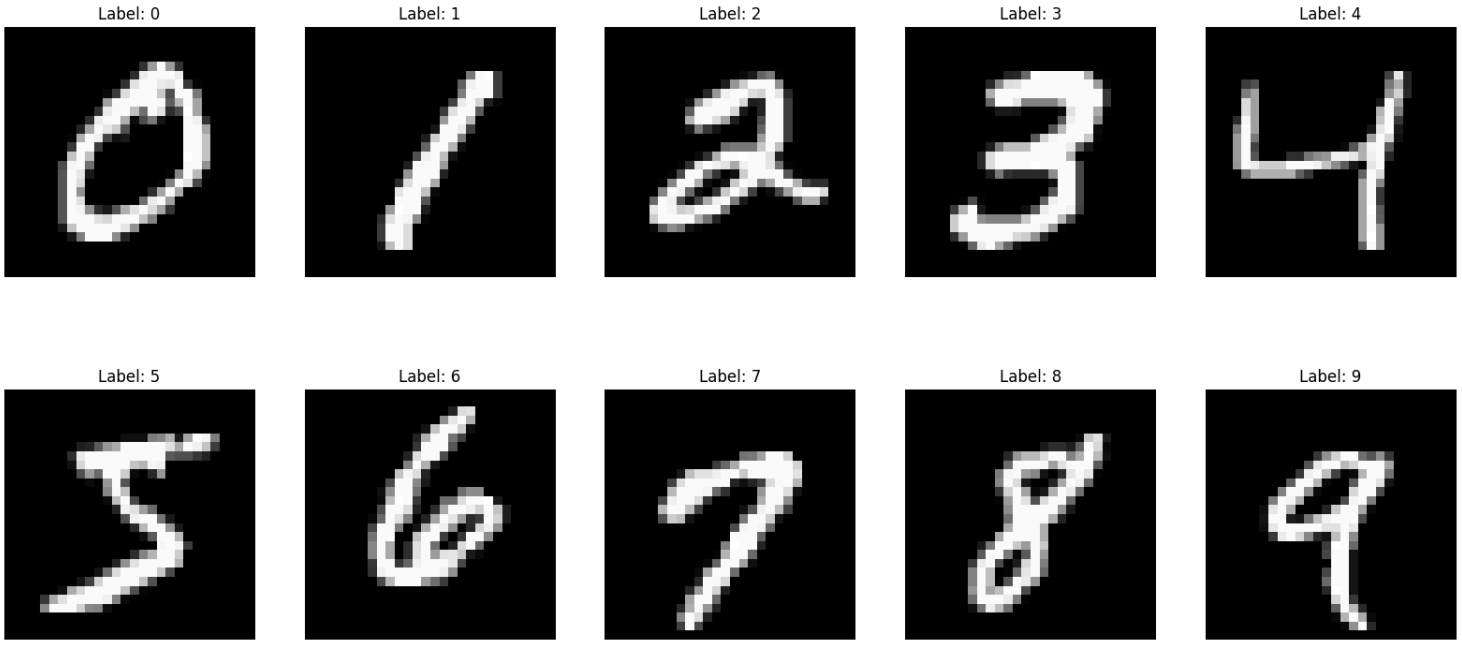
این دیتاست یکی از معروف‌ترین و پایه‌ای‌ترین دیتاست‌های بینایی ماشین است که برای تشخیص دست‌نوشته‌های رقمی (digits) به کار می‌رود. این دیتاست شامل 70,000 تصویر دست‌نوشته از ارقام 0 تا 9 است که در 2 دسته آموزش (60.000 عدد) و آزمون (10.000 عدد) قرار دارند. هر تصویر در این دیتاست، دارای ابعاد 28 × 28 پیکسل است.

برای خواندن این مدل به شکل زیر عمل میکنیم:

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
assert x_train.shape == (60000, 28, 28)
assert x_test.shape == (10000, 28, 28)
assert y_train.shape == (60000,)
assert y_test.shape == (10000,)
```

نمونه کلاس‌های موجود در دیتاست MNIST

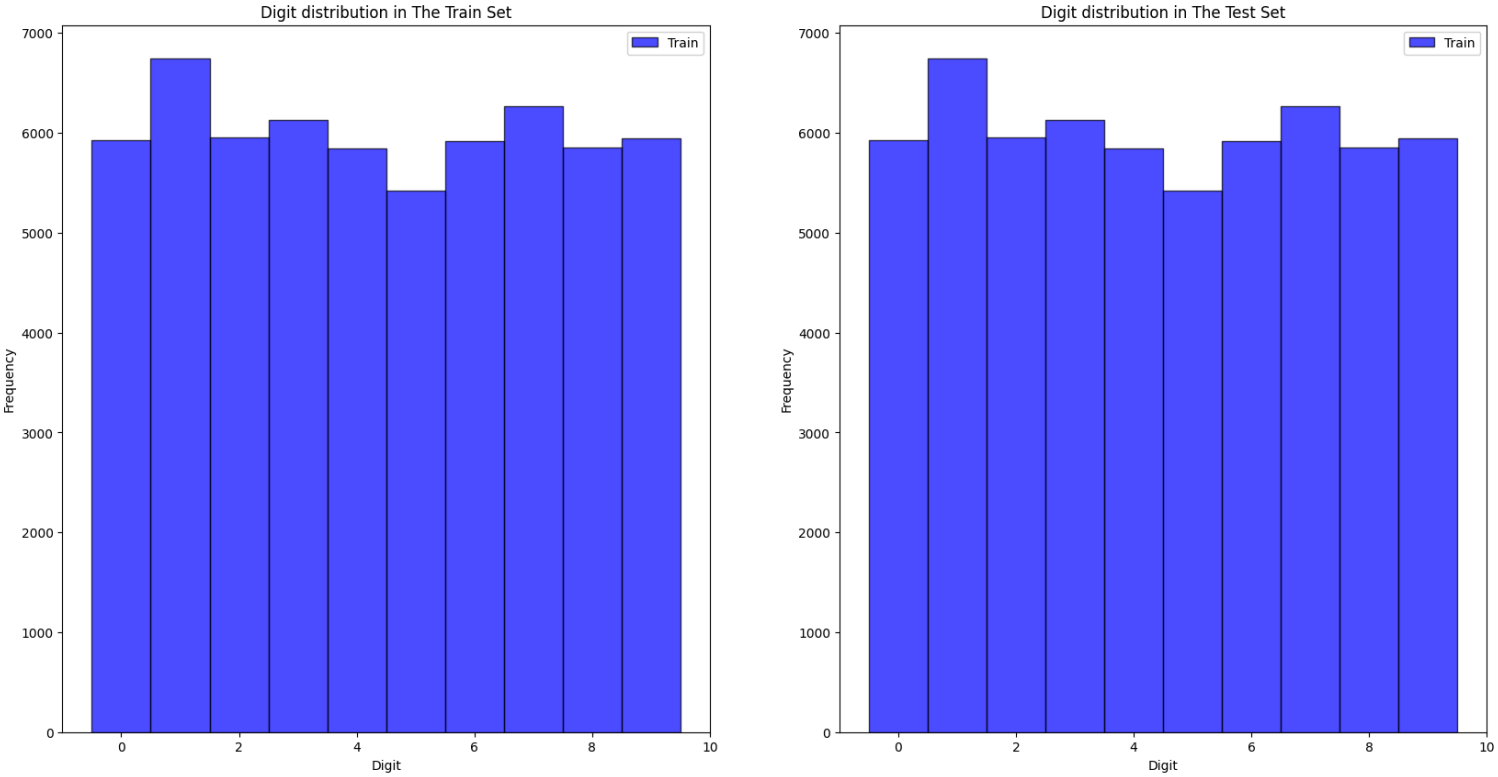
برای آشنایی بیشتر نمونه از هر کلاس را مشاهده می‌کنیم



شکل 2-1. نمونه‌ای از هر کلاس در دیتاست MNIST

توزیع کلاس‌ها

سپس به بررسی توزیع لیبل‌ها در دیتاست آموزش و آزمون می‌پردازیم:



شکل 2-2. نمودار histogram توزیع داده‌ها در دو دیتاست آموزش و آزمون

همانگونه که مشاهده می‌شود توزیع تقریبا یونیفرم است و نیازی به کاری برای یکسان کردن توزیع نداریم.

پیش‌پردازش (pre-process)

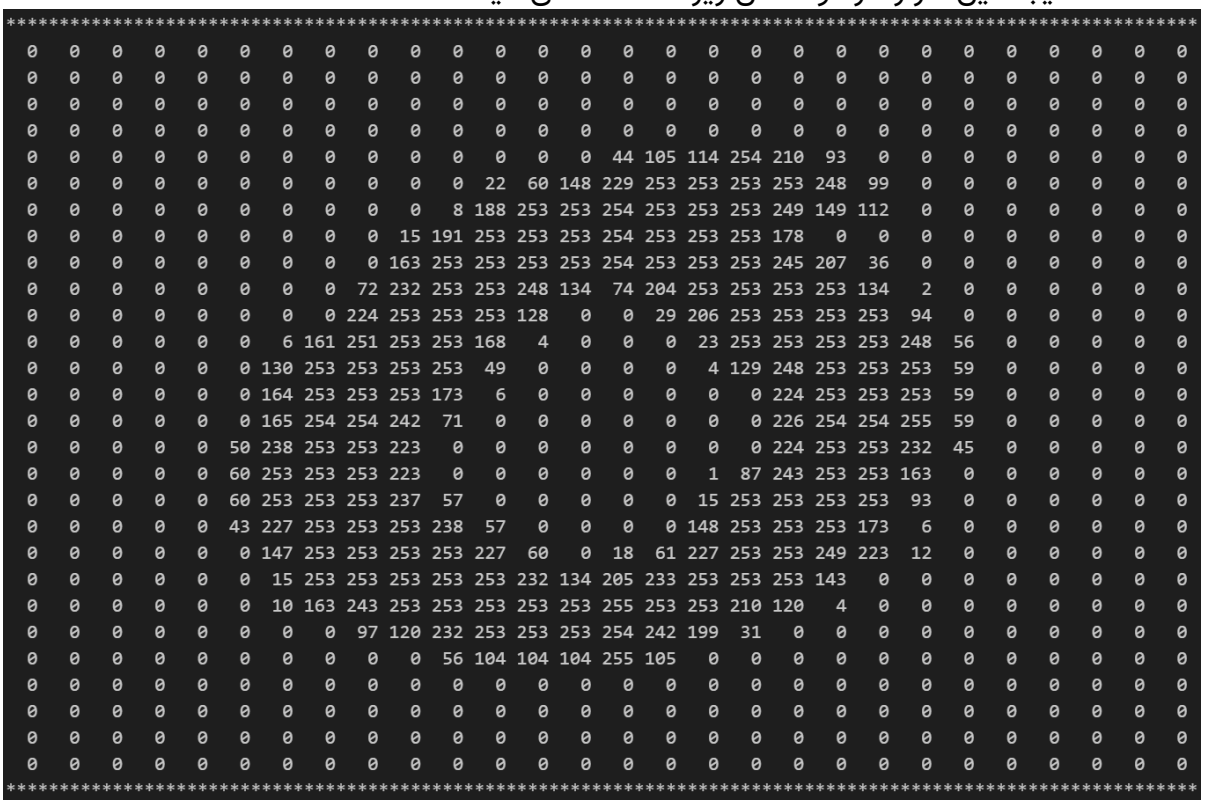
سپس به مرحله پیش‌پردازش (pre-process) می‌پردازیم. در اینجا از نرمال‌سازی بیشینه-کمینه (min-max normalization) استفاده می‌کنیم. دلیل این کار است که:

1. **یکنواختی مقیاس‌ها**: نرمال‌سازی باعث می‌شود تمام ویژگی‌ها به طور مساوی در عملکرد مدل مشارکت کنند، بدون توجه به مقیاس اصلی آن‌ها. این موضوع برای الگوریتم‌هایی که بر مبنای معیارهای فاصله استناد می‌کنند، مانند همسایه‌های نزدیک‌ترین (KNN) یا ماشین‌های بردار پشتیبان (Support Vector Machine)، بسیار مهم است.
2. **بهبود عملکرد مدل**: داده‌های نرمال‌سازی شده می‌توانند عملکرد مدل را بهبود بخشند و دقت یک مدل را افزایش دهند. این موضوع استقرار را در فرآیند بهینه‌سازی تقویت می‌کند، و باعث می‌شود آموزش مبتنی بر گرادیان به سرعت همگرا شود.
3. **جلوگیری از ناپایداری عددی**: نرمال‌سازی مشکلات مرتبط با گرادیان‌های ناپدید شونده یا منفجر شونده را کاهش می‌دهد، و به مدل‌ها اجازه می‌دهد به راحتی به راه‌حل‌های بهینه برسند.
4. **تفسیر و تصویرسازی آسان‌تر**: وقتی تمام ویژگی‌های یک مجموعه داده در یک مقیاس هستند، شناسایی و تصویرسازی روابط بین ویژگی‌های مختلف و انجام مقایسه‌های معنی‌دار آسان‌تر می‌شود.

برای این کار به صورت زیر عمل کردیم:

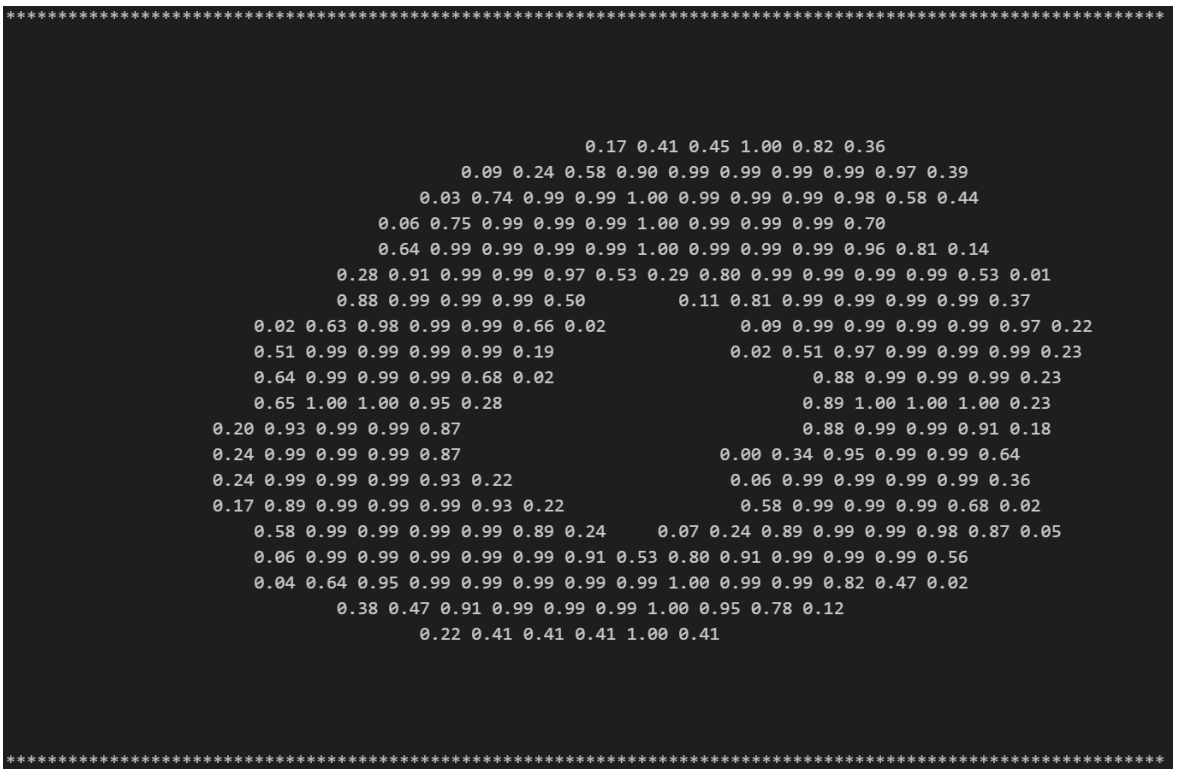
```
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)

x_train_scaled = minmax_scale(x_train)
x_test_scaled = minmax_scale(x_test)
```

نتیجه این کار را در دو عکس زیر مشاهده می‌کنید.



```
********************************************************************************************************************
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0  44 105 114 254 210  93   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0  22  60 148 229 253 253 253 253 248  99   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   8 188 253 253 254 253 253 253 249 149 112   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0  15 191 253 253 253 254 253 253 253 178   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0 163 253 253 253 253 254 253 253 253 245 207  36   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0  72 232 253 253 248 134  74 204 253 253 253 253 134   2   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0 224 253 253 253 128   0   0  29 206 253 253 253 253  94   0   0   0   0   0   0   0
  0   0   0   0   0   0   6 161 251 253 253 168   4   0   0   0  23 253 253 253 253 248  56   0   0   0   0   0   0
  0   0   0   0   0   0 130 253 253 253 253  49   0   0   0   0   4 129 248 253 253 253  59   0   0   0   0   0   0
  0   0   0   0   0   0 164 253 253 253 173   6   0   0   0   0   0   0 224 253 253 253  59   0   0   0   0   0   0
  0   0   0   0   0   0 165 254 254 242  71   0   0   0   0   0   0   0 226 254 254 255  59   0   0   0   0   0   0
  0   0   0   0   0  50 238 253 253 223   0   0   0   0   0   0   0   0 224 253 253 232  45   0   0   0   0   0   0
  0   0   0   0   0  60 253 253 253 223   0   0   0   0   0   0   1  87 243 253 253 163   0   0   0   0   0   0   0
  0   0   0   0   0  60 253 253 253 237  57   0   0   0   0   0  15 253 253 253 253  93   0   0   0   0   0   0   0
  0   0   0   0   0  43 227 253 253 253 238  57   0   0   0   0 148 253 253 253 173   6   0   0   0   0   0   0   0
  0   0   0   0   0   0 147 253 253 253 253 227  60   0  18  61 227 253 253 249 223  12   0   0   0   0   0   0   0
  0   0   0   0   0   0  15 253 253 253 253 253 232 134 205 233 253 253 253 143   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0  10 163 243 253 253 253 253 253 255 253 253 210 120   4   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0  97 120 232 253 253 253 254 242 199  31   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0  56 104 104 104 255 105   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
********************************************************************************************************************
```

شکل 2-3. مقادیر یک نمونه پیش از نرمال‌سازی



```
********************************************************************************************************************

                                                0.17 0.41 0.45 1.00 0.82 0.36
                                          0.09 0.24 0.58 0.90 0.99 0.99 0.99 0.99 0.97 0.39
                                     0.03 0.74 0.99 0.99 1.00 0.99 0.99 0.99 0.98 0.58 0.44
                                0.06 0.75 0.99 0.99 0.99 1.00 0.99 0.99 0.99 0.70
                                0.64 0.99 0.99 0.99 0.99 1.00 0.99 0.99 0.99 0.96 0.81 0.14
                           0.28 0.91 0.99 0.99 0.97 0.53 0.29 0.80 0.99 0.99 0.99 0.99 0.53 0.01
                           0.88 0.99 0.99 0.99 0.50         0.11 0.81 0.99 0.99 0.99 0.99 0.37
                     0.02 0.63 0.98 0.99 0.99 0.66 0.02             0.09 0.99 0.99 0.99 0.99 0.97 0.22
                     0.51 0.99 0.99 0.99 0.99 0.19                0.02 0.51 0.97 0.99 0.99 0.99 0.23
                     0.64 0.99 0.99 0.99 0.68 0.02                       0.88 0.99 0.99 0.99 0.23
                     0.65 1.00 1.00 0.95 0.28                          0.89 1.00 1.00 1.00 0.23
                0.20 0.93 0.99 0.99 0.87                               0.88 0.99 0.99 0.91 0.18
                0.24 0.99 0.99 0.99 0.87                          0.00 0.34 0.95 0.99 0.99 0.64
                0.24 0.99 0.99 0.99 0.93 0.22                      0.06 0.99 0.99 0.99 0.99 0.36
                0.17 0.89 0.99 0.99 0.99 0.93 0.22                  0.58 0.99 0.99 0.99 0.68 0.02
                     0.58 0.99 0.99 0.99 0.99 0.89 0.24     0.07 0.24 0.89 0.99 0.99 0.98 0.87 0.05
                     0.06 0.99 0.99 0.99 0.99 0.99 0.91 0.53 0.80 0.91 0.99 0.99 0.99 0.56
                     0.04 0.64 0.95 0.99 0.99 0.99 0.99 0.99 1.00 0.99 0.99 0.82 0.47 0.02
                           0.38 0.47 0.91 0.99 0.99 0.99 1.00 0.95 0.78 0.12
                                0.22 0.41 0.41 0.41 1.00 0.41

********************************************************************************************************************
```

شکل 2-4. مقادیر یک نمونه پس از نرمال‌سازی (برای خوانایی بیشتر 0 ها حذف شده)

3-2. ایجاد و آموزش مدل

حال قصد داریم از یک شبکه عصبی ساده با دو لایه پنهان برای طبقه‌بندی (classification) این دیتاست استفاده کنیم.

معماری (Architecture)

این شبکه عصبی دارای معماری زیر خواهد بود:

1. لایه ورودی: 784 نورون (پیکسل‌های 28x28) به 512 نورون با تابع فعال‌سازی ReLU. دلیل استفاده از 784 نورون ورودی برای این است که تعداد مشخصه‌های هر عکس (تعداد پیکسل‌های آن) 784 تا (28 * 28) می‌باشد.

2. لایه پنهان 1: 512 نورون به 128 نورون با تابع فعال‌سازی ReLU.

3. لایه پنهان 2: 128 نورون به 32 نورون با تابع فعال‌سازی ReLU.

4. لایه خروجی: 32 نورون به 10 نورون (متناظر با 10 کلاس) با تابع فعال‌سازی softmax.

در زمینه شبکه‌های عصبی، اصطلاح "logit" معمولاً به خروجی خام و غیرنرمال‌سازی شده یک نورون اشاره دارد. این logit-ها معمولاً از طریق یک تابع فعال‌سازی، مانند تابع سافت‌مکس (softmax)، تبدیل به احتمالاتی می‌شوند که مجموع آن‌ها برابر با یک است.

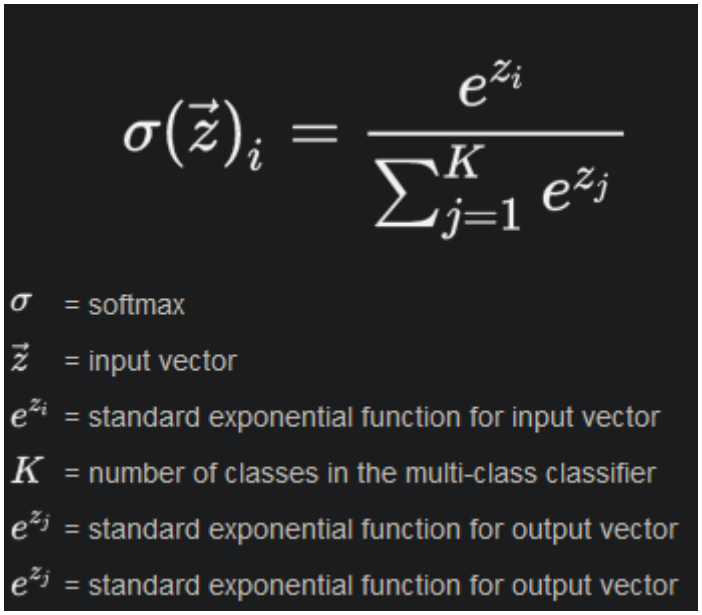
تابع لوژیت خود یک مفهوم حیاتی در آمار و یادگیری ماشین است، به خصوص در زمینه رگرسیون لجستیک (logistic regression). این تابع به عنوان یک تابع پیوند عمل می‌کند که احتمالاتی را که در بازه بین 0 و 1 قرار دارند، به اعداد حقیقی در کل خط عددی نگاشت می‌کند، که سپس می‌تواند برای بیان روابط خطی استفاده شود.

در یک شبکه عصبی، هر نورون مجموع وزن‌دار ورودی خود را محاسبه می‌کند، یک سوگیری (bias) اضافه می‌کند، و سپس یک تابع فعال‌سازی را اعمال می‌کند. نتیجه مجموع وزن‌دار به علاوه سوگیری (bias) برابر logit است. تابع فعال‌سازی سپس logit را به یک خروجی تبدیل می‌کند که بستگی به وظیفه مورد نظر دارد. به عنوان مثال، در یک مسئله طبقه‌بندی دودویی، ممکن است از یک تابع فعال‌سازی سیگموئید (sigmoid) استفاده شود تا خروجی را بین 0 و 1 فشرده کند، و آن را به عنوان یک احتمال تفسیر کند.

تابع فعال‌سازی آخر در معماری شبکه عصبی ما متفاوت است زیرا نسبت به دیگران وظیفه متفاوتی دارد. توابع فعال‌سازی در لایه‌های پنهان (ReLU) برای معرفی غیرخطی در شبکه استفاده می‌شوند، که این امکان را می‌دهد تا الگوهای پیچیده را یاد بگیرد.

از طرف دیگر، تابع فعال‌سازی سافت‌مکس در لایه خروجی برای تبدیل امتیازات خام و بی‌حد (logit-ها) به توزیع احتمال بر روی چندین کلاس استفاده می‌شود. این برای مسائل طبقه‌بندی چندکلاسه بسیار مفید است، جایی که یک ورودی باید به یکی از چندین کلاس اختصاص یابد.

تابع سافت‌مکس (softmax) یک بردار از اعداد حقیقی را به عنوان ورودی می‌گیرد و یک بردار دیگر با همان بعد را با مقادیری که بین 0 و 1 قرار دارند، برمی‌گرداند. از آنجا که این مقادیر به 1 می‌رسند، آن‌ها احتمالات معتبری را نشان می‌دهند. این تابع به کلاس‌هایی با لوژیت‌های بالاتر احتمالات بیشتری اختصاص می‌دهد، که به شما امکان انتخاب کلاس احتمالی‌تر را می‌دهد.



شکل 2-4. فرمول ریاضی برای تابع سافت‌مکس (softmax)

پارامترها (Parameters)

- نرخ یادگیری (Learning rate): 6e-5

نرخ یادگیری یک پارامتر در یادگیری عمیق است که تعیین می‌کند چقدر مدل در هر دوره یا مرحله آموزش تغییر کند. نرخ یادگیری بالا می‌تواند باعث شود یادگیری سریع‌تر اتفاق بیفتد، اما اگر خیلی بالا باشد، ممکن است مدل به حالت ناسازگار برسد و به جواب بهینه نرسد. برعکس، نرخ یادگیری پایین می‌تواند باعث شود یادگیری کندتر اتفاق بیفتد، اما اگر خیلی پایین باشد، ممکن است یادگیری بسیار کند شود یا حتی متوقف شود.

- تعداد دوره‌ها (Epochs): حدود 25

تعداد دوره‌ها تعداد باری است که الگوریتم یادگیری کل داده‌های آموزش را می‌بیند. بیشتر دوره‌ها می‌تواند باعث شود مدل بهتری را یاد بگیرد، اما اگر خیلی زیاد باشد، ممکن است مدل به حالت بیش‌برازش (overfitting) برسد و عملکرد خوبی روی داده‌های تست نداشته باشد.

- بهینه‌ساز (Optimizer): Adam

بهینه‌ساز یک الگوریتم یا روش است که برای تنظیم و بهینه‌سازی پارامترهای مدل استفاده می‌شود. Adam یکی از روش‌های محبوب برای بهینه‌سازی است که معمولاً عملکرد خوبی دارد.

بهینه ساز Adam، که مخفف Adaptive Moment Estimation است، یک الگوریتم بهینه سازی است که در یادگیری عمیق استفاده می شود. این الگوریتم در سال 2014 توسط دیدریک کینگما و جیمی با معرفی شد. بهینه ساز آدام ترکیبی از دو الگوریتم بهینه سازی دیگر است: Momentum و Root Mean Square Propagation ، که به شرح زیر عمل می کند:

1. Momentum: این الگوریتم با در نظر گرفتن "میانگین وزن دار نمایی" گرادیان ها، سرعت گرادیان را افزایش می دهد. استفاده از میانگین ها باعث می شود الگوریتم به سرعت بیشتری به سمت کمینه ها همگرا شود.
2. RMSProp: یک الگوریتم یادگیری تطبیقی است که سعی در بهبود AdaGrad دارد. به جای گرفتن مجموع تجمعی گرادیان های مربعی مانند AdaGrad، "میانگین حرکتی نمایی" را می گیرد.

آدام برای هر پارامتر مدل دو برآورد را حفظ می کند: میانگین حرکتی گرادیان و میانگین حرکتی گرادیان مربعی. این الگوریتم از این برآوردها برای تطبیق نرخ یادگیری برای هر وزن شبکه عصبی به طور جداگانه استفاده می کند، که می تواند منجر به یادگیری کارآمدتر شود. مزایای استفاده از آدام شامل کارایی محاسباتی، نیاز کم به حافظه، مناسب بودن برای مسائلی که از نظر داده و/یا پارامترها بزرگ هستند، و کارایی آن برای اهداف غیرثابت است.

- تابع خطا (Loss Function): Cross-Entropy Loss

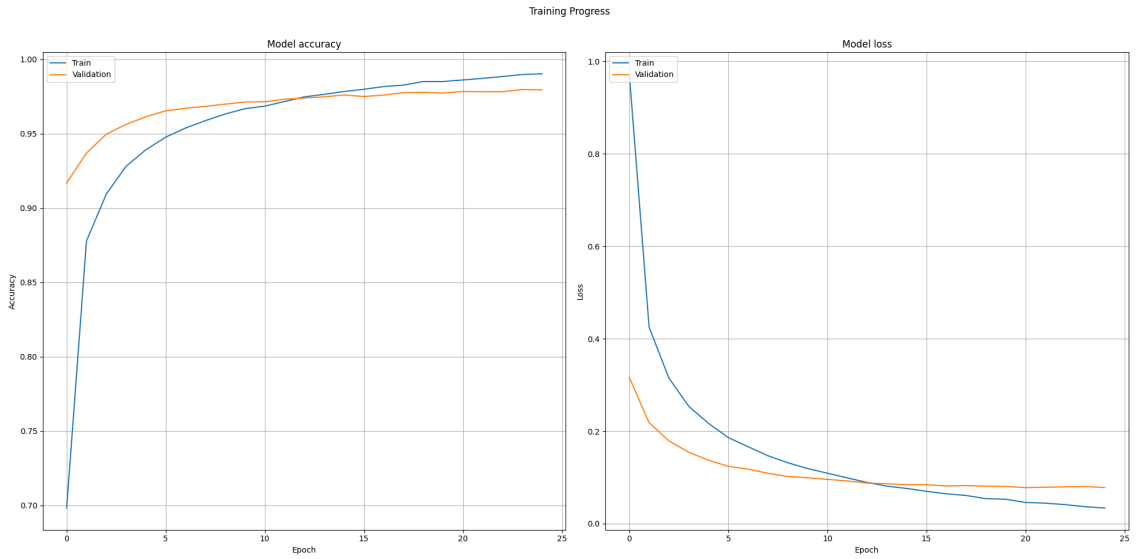
تابع خطا یا تابع هزینه، یک معیار از اختلاف بین پیش‌بینی مدل و داده‌های واقعی است. خطای آنتروپی متقاطع (Cross-Entropy Loss) یک تابع خطای رایج برای مسائل طبقه‌بندی است که میزان اختلاف بین پیش‌بینی مدل و برچسب واقعی را اندازه‌گیری می‌کند. تابع خطای آنتروپی متقاطع (Cross-Entropy Loss) برای اندازه گیری عملکرد یک مدل دسته بندی استفاده می شود که خروجی آن یک مقدار احتمال بین 0 و 1 است. خطای آنتروپی متقاطع (Cross-Entropy Loss) با افزایش اختلاف احتمال پیش بینی شده از برچسب واقعی افزایش می یابد. بنابراین پیش بینی یک احتمال 0.012 وقتی برچسب مشاهده واقعی 1 باشد، بد است و منجر به ارزش خطای بالا می شود. یک مدل کامل خطای لگاریتمی 0 خواهد داشت.

پیاده‌سازی

حال به پیاده‌سازی این شبکه می‌پردازیم، برای این کار ما از `tensorflow` استفاده کردیم.

```
model = Sequential(
      [
       Input(shape=(784,)),
       Dense(512, activation="relu"),
       Dropout(0.2),
       Dense(128, activation="relu"),
       Dropout(0.2),
       Dense(32, activation="relu"),
       Dropout(0.2),
       Dense(10, activation="softmax"),
      ]
)
```

```
EPOCHS = 25
optimizer = Adam(learning_rate=6e-5)
model.compile(optimizer=optimizer,
loss=sparse_categorical_crossentropy, metrics=["accuracy"])
history = model.fit(x_train_scaled, y_train, epochs=EPOCHS,
validation_split=0.2, callbacks=[tensorboard_callback],
batch_size=64)
```



شکل 2-5. نمودار تغییرات loss و accuracy برای در هنگام آموزش مدل

نتیجه آخرین دوره (epoch):

```
Epoch 25/25
750/750 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step -
accuracy: 0.9974 - loss: 0.0104 - val_accuracy: 0.9813 -
val_loss: 0.0934
```

سپس دقت مدل ساخته‌شده را بررسی می‌کنیم:

```
test_loss, test_acc = model.evaluate(x_test_scaled, y_test,
verbose=1)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 943us/step -
accuracy: 0.9785 - loss: 0.0809
```

همانگونه که مشاهده می‌شود پیش از حمله مدل با دقت خوبی در حال پیش‌بینی می‌باشد.

4-2. پیاده‌سازی حمله FGSM

حمله FGSM یا حمله نشانه گرادیان سریع (Fast Gradient Sign Method)، یکی از اولین و محبوب‌ترین حملات دشمنانه به شبکه‌های عصبی است. این حمله بر اساس چگونگی یادگیری شبکه‌های عصبی، یعنی گرادیان‌ها، طراحی شده است.

در این حمله، به جای کاهش خطا با تنظیم وزن‌ها بر اساس گرادیان‌های backpropagated، حمله داده ورودی را تنظیم می‌کند تا خطا را بر اساس همان گرادیان‌های backpropagated افزایش دهد. به عبارت دیگر، حمله گرادیان خطا را نسبت به داده ورودی استفاده می‌کند، سپس داده ورودی را تنظیم می‌کند تا خطا را بیشینه کند.

FGSM یک حمله white-box است که فرض می‌کند مهاجم دسترسی کامل و دانش کامل از مدل را دارد، از جمله معماری، ورودی‌ها، خروجی‌ها و وزن‌ها. هدف این حمله، اضافه کردن کمترین مقدار اغتشاش به داده‌های ورودی است تا باعث بروز خطای دلخواه شود.

مراحل پیاده‌سازی در حمله FGSM به شرح زیر است:

1. گرادیان های خطا را نسبت به تصویر ورودی محاسبه کنیم.
2. نشانه گرادیان‌ها را برای ایجاد اغتشاش بگیریم.
3. اغتشاش را به تصویر ورودی اضافه کنیم تا یک نمونه دشمنانه ایجاد کنیم.
4. مقادیر پیکسل نمونه دشمنانه را در بازه [0، 1] برش دهیم.
5. سپس نمونه دشمنانه به شبکه عصبی تغذیه می‌شود تا خروجی را بگیریم.

پیاده‌سازی

این بخش را می‌توان به راحتی با کتابخونه Foolbox انجام داد:

```
fmodel = fb.models.TensorFlowModel(model, bounds=(0, 1))

attack = fb.attacks.LinfFastGradientAttack()

adversarial_test_images, perturbations, success = attack(fmodel,
x_test_scaled_tensor, y_test_tensor, epsilons=0.2)
```

```
Attack Success Rate: 0.9748
```

یا می‌توانیم خودمان به صورت دستی پیاده‌سازی کنیم:

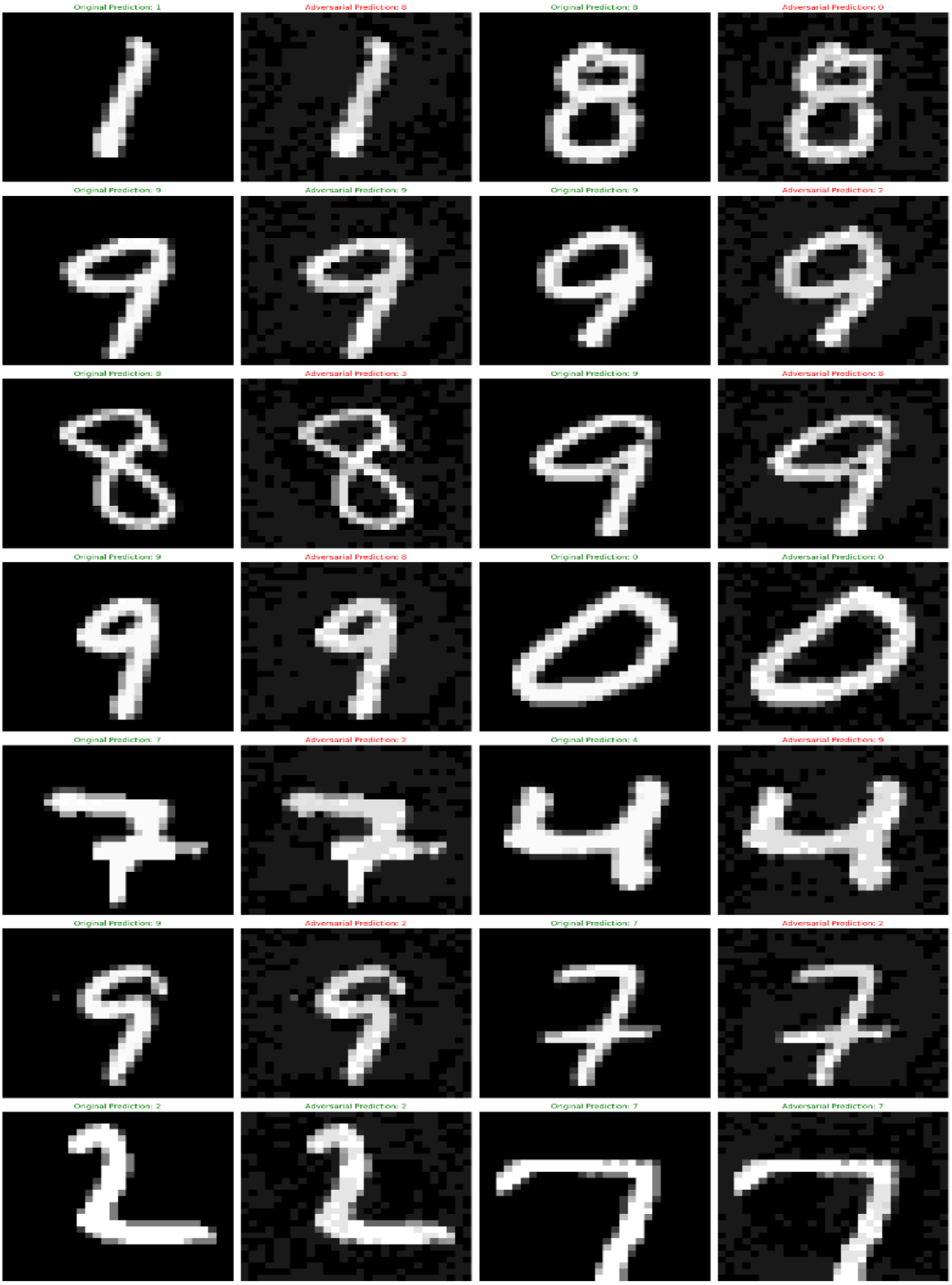
```
def create_adversarial_pattern(input_image, input_label, model):

      input_image = tf.convert_to_tensor(input_image,
dtype=tf.float32)

      with tf.GradientTape() as tape:
      tape.watch(input_image)
      prediction = model(input_image)
      loss = sparse_categorical_crossentropy(input_label,
prediction)

      gradient = tape.gradient(loss, input_image)

      return tf.sign(gradient)

def fgsm(input_image, input_label, model, epsilon=0.01):
      perturbation = create_adversarial_pattern(input_image,
input_label, model)
      adversarial_image = input_image + epsilon * perturbation
      adversarial_image = tf.clip_by_value(adversarial_image, 0,
1)
      return adversarial_image
```

```
advesarial_test_images = fgsm(x_test_scaled, y_test, model,
epsilon=0.1)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Original Accuracy: 0.9813
Adversarial Accuracy: 0.2102

Attack Efficiency: 0.7710999999999999
```

```
Attack Efficiency for Label 0: 0.10918367346938773
Attack Efficiency for Label 1: 0.4
Attack Efficiency for Label 2: 0.21317829457364335
Attack Efficiency for Label 3: 0.2504950495049505
Attack Efficiency for Label 4: 0.40631364562118133
Attack Efficiency for Label 5: 0.25
Attack Efficiency for Label 6: 0.13883089770354906
Attack Efficiency for Label 7: 0.28696498054474706
Attack Efficiency for Label 8: 0.29055441478439425
Attack Efficiency for Label 9: 0.40436075322101084
```

همانگونه که مشاهده می‌شود با انجام این کار دقت مدل تا حد خوبی کاهش می‌یابد و حمله با موفقیت انجام شده. نمونه‌هایی از عکس‌های ورودی در حال حمله را شکل زیر مشاهده می‌کنید:



شکل 2-6. نمونه‌ای از هر کلاس در حالت حمله خصمانه FGSM

5-2. پیاده‌سازی حمله PGD

حمله PGD یا حمله گرادیان نزولی پروژه‌ای (Projected Gradient Descent)، یکی از روش‌های پیشرفته برای ایجاد نمونه‌های دشمنانه در شبکه‌های عصبی است. این حمله بر اساس گرادیان‌های شبکه عمل می‌کند، مشابه حمله FGSM، اما با تفاوت‌های مهمی.

در حمله PGD، به جای ایجاد یک اغتشاش یک‌گامی (مانند حمله FGSM)، یک سری اغتشاش کوچک به صورت تکراری به تصویر اضافه می‌شود. در هر گام، یک اغتشاش کوچک (مشابه FGSM) به تصویر اضافه می‌شود و سپس تصویر حاصل به بازه مجاز برش زده می‌شود. این فرآیند برای تعدادی گام تکرار می‌شود.

حمله PGD نیز مانند FGSM یک حمله جعبه سفید (white-box) است که فرض می‌کند مهاجم دسترسی کامل و دانش کامل از مدل را دارد، از جمله معماری، ورودی‌ها، خروجی‌ها و وزن. هدف این حمله، اضافه کردن کمترین مقدار اغتشاش به داده‌های ورودی است تا باعث بروز خطای دلخواه شود.

حمله PGD نسبت به حمله FGSM موثرتر است چون از یک روش تکراری برای اعمال اغتشاش استفاده می‌کند. در حمله FGSM، اغتشاش فقط یک بار اعمال می‌شود. اما در حمله PGD، اغتشاش در چندین گام به صورت تکراری اعمال می‌شود.

این روش تکراری باعث می‌شود که حمله PGD بتواند به نقاطی از فضای ورودی دست یابد که حمله FGSM نمی‌تواند به آن‌ها برسد. به عبارت دیگر، حمله PGD می‌تواند اغتشاش‌هایی را ایجاد کند که باعث می‌شوند شبکه عصبی خطای بیشتری را تجربه کند.

پیاده‌سازی

برای پیاده‌سازی این بخش می‌توان باز هم از کتابخونه Foolbox کمک گرفت:

```
fmodel = fb.models.TensorFlowModel(model, bounds=(0, 1))

attack = fb.attacks.LinfPGD()

adversarial_test_images, _, success = attack(fmodel,
x_test_scaled_tensor, y_test_tensor, epsilons=0.2)
```

```
Attack Success Rate: 0.9997
```

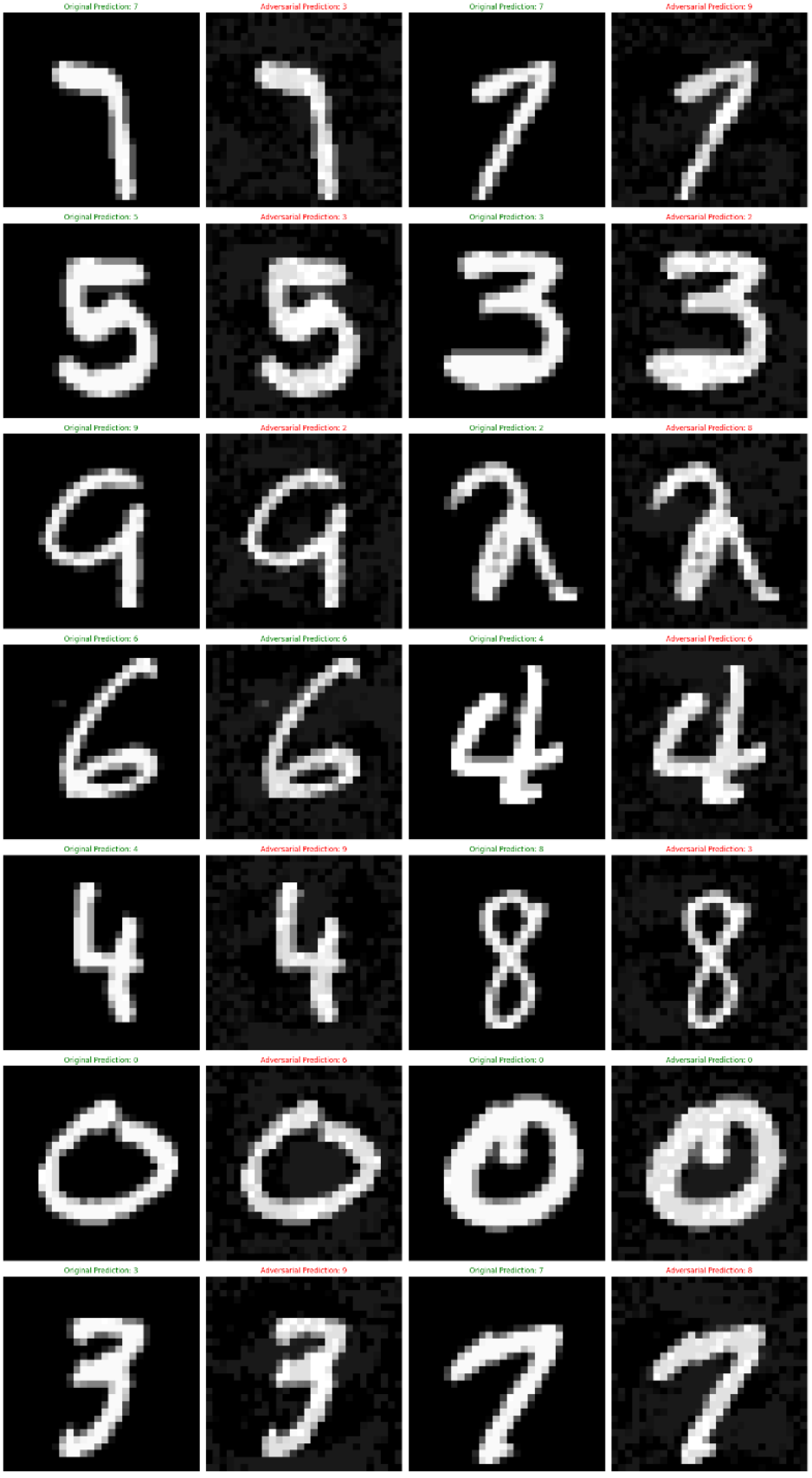
یا می‌توانیم خودمان به صورت دستی پیاده‌سازی کنیم:

```
def PGD_attack(input_image, input_label, model, epsilon=0.01,
epsilon_iter=0.01, num_iter=10):
      adversarial_image = input_image
      for i in range(num_iter):
      perturbation =
create_adversarial_pattern(adversarial_image, input_label,
model)
      adversarial_image = adversarial_image + epsilon_iter *
perturbation
      adversarial_image = tf.clip_by_value(adversarial_image,
input_image - epsilon, input_image + epsilon)
      adversarial_image = tf.clip_by_value(adversarial_image, 0,
1)
      return adversarial_image

adversarial_test_images = PGD_attack(x_test_scaled, y_test,
model, epsilon=0.1, epsilon_iter=0.01, num_iter=10)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 983us/step
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 960us/step
Original Accuracy: 0.9813
Adversarial Accuracy: 0.1165
Attack Efficiency: 0.76470003
```

```
Attack Efficiency for Label 0: 0.6734693877551021
Attack Efficiency for Label 1: 0.9911894273127754
Attack Efficiency for Label 2: 0.7877906976744186
Attack Efficiency for Label 3: 0.8752475247524752
Attack Efficiency for Label 4: 0.9643584521384929
Attack Efficiency for Label 5: 0.8576233183856503
Attack Efficiency for Label 6: 0.6732776617954072
Attack Efficiency for Label 7: 0.9416342412451362
Attack Efficiency for Label 8: 0.891170431211499
Attack Efficiency for Label 9: 0.9643211100099107
```

همانگونه که مشاهده می‌شود در روش PGD با پارامترهای یکسان مدل دقت کمتری داشته و حمله موثرتری رخ داده. در شکل زیر نیز می‌توانید تعدادی از نمونه‌ها را مشاهده کنید.



شکل 2-7. نمونه‌ای از هر کلاس در حالت حمله خصمانه PGD

پرسش 3 – Madaline و Adaline

Adaline .1-3

نورون Adeline

نورون Adeline (Adaptive Linear Neuron) یک نوع نورون در شبکه عصبی است که برای یادگیری و پردازش داده‌ها استفاده می‌شود. این نوع نورون از یک واحد خطی تک نورونی تشکیل شده است. در Adeline، فقط یک واحد خروجی وجود دارد و مقادیر خروجی دو قطبی (+1,-1) هستند. وزن‌ها بین واحد ورودی و واحد خروجی قابل تنظیم هستند. این شبکه از قاعده دلتا استفاده می‌کند (Delta Rule). قانون دلتا (Delta Rule) یک روش یادگیری در شبکه‌های عصبی است که برای بهینه‌سازی وزن‌ها استفاده می‌شود. در قانون دلتا، اصلاح در وزن یک گره برابر با ضرب خطا و ورودی است. این قانون بر اساس محاسبه خطا، که تفاوت بین خروجی مورد انتظار و خروجی واقعی است، عمل می‌کند. سپس، این خطا با ورودی ضرب می‌شود تا میزان تغییر وزن را مشخص کند. به این ترتیب، وزن‌ها به‌گونه‌ای تنظیم می‌شوند که خطا کمینه شود.

$$\text{net} = \sum_{i=1}^{n} w_i x_i + b$$

$$h = f(net) = \begin{cases} +1 & \text{net} \geq 0 \\ -1 & \text{net} < 0 \end{cases}$$

شکل 3-1. فرمول محاسبه خروجی نورون Adeline

$$w_i^+ = w_i^- + \alpha(t - net)x_i$$

$$\alpha > 0 : \alpha \text{ small enough learning rate}$$

$$b^+ = b^- + \alpha(t - net)$$

شکل 3-2. نحوه اصلاح وزن‌ها در نورون Adeline

دیتاست

حال سعی می‌کنیم با کمک این نورون به طبقه‌بندی (classification) یک دیتاست معروف به نام wines بپردازیم. مشخصات دیتاست به شکل زیر است:

```
.. _wine_dataset:

Wine recognition dataset
------------------------

**Data Set Characteristics:**

:Number of Instances: 178
:Number of Attributes: 13 numeric, predictive attributes and the
class
:Attribute Information:
      - Alcohol
      - Malic acid
      - Ash
      - Alcalinity of ash
      - Magnesium
      - Total phenols
      - Flavanoids
      - Nonflavanoid phenols
      - Proanthocyanins
      - Color intensity
      - Hue
      - OD280/OD315 of diluted wines
      - Proline
      - class:
      - class_0
      - class_1
      - class_2
```

```
:Summary Statistics:

============================== ==== ===== ======= =====
                                  Min   Max   Mean      SD
============================== ==== ===== ======= =====
Alcohol:                      11.0  14.8 13.0   0.8
Malic Acid:                   0.74  5.80 2.34  1.12
Ash:                          1.36  3.23 2.36  0.27
Alcalinity of Ash:            10.6  30.0 19.5   3.3
Magnesium:                    70.0 162.0 99.7  14.3
Total Phenols:                0.98  3.88 2.29  0.63
Flavanoids:                   0.34  5.08 2.03  1.00
Nonflavanoid Phenols:         0.13  0.66 0.36  0.12
Proanthocyanins:              0.41  3.58 1.59  0.57
Colour Intensity:             1.3  13.0  5.1   2.3
Hue:                          0.48  1.71 0.96  0.23
OD280/OD315 of diluted wines: 1.27  4.00    2.61  0.71
Proline:                      278  1680  746   315
============================== ==== ===== ======= =====

:Missing Attribute Values: None
:Class Distribution: class_0 (59), class_1 (71), class_2 (48)
:Creator: R.A. Fisher
:Donor: Michael Marshall
:Date: July, 1988

This is a copy of UCI ML Wine recognition datasets.
https://archive.ics.uci.edu/ml/machine-learning-databases/wine/w
ine.data

The data is the results of a chemical analysis of wines grown in
the same
region in Italy by three different cultivators. There are
thirteen different
measurements taken for different constituents found in the three
types of
wine.

Original Owners:

Forina, M. et al, PARVUS -
An Extendible Package for Data Exploration, Classification and
```

Correlation.
Institute of Pharmaceutical and Food Analysis and Technologies,
Via Brigata Salerno, 16147 Genoa, Italy.

Citation:

Lichman, M. (2013). UCI Machine Learning Repository
[https://archive.ics.uci.edu/ml]. Irvine, CA: University of
California,
School of Information and Computer Science.

|details-start|
References
|details-split|

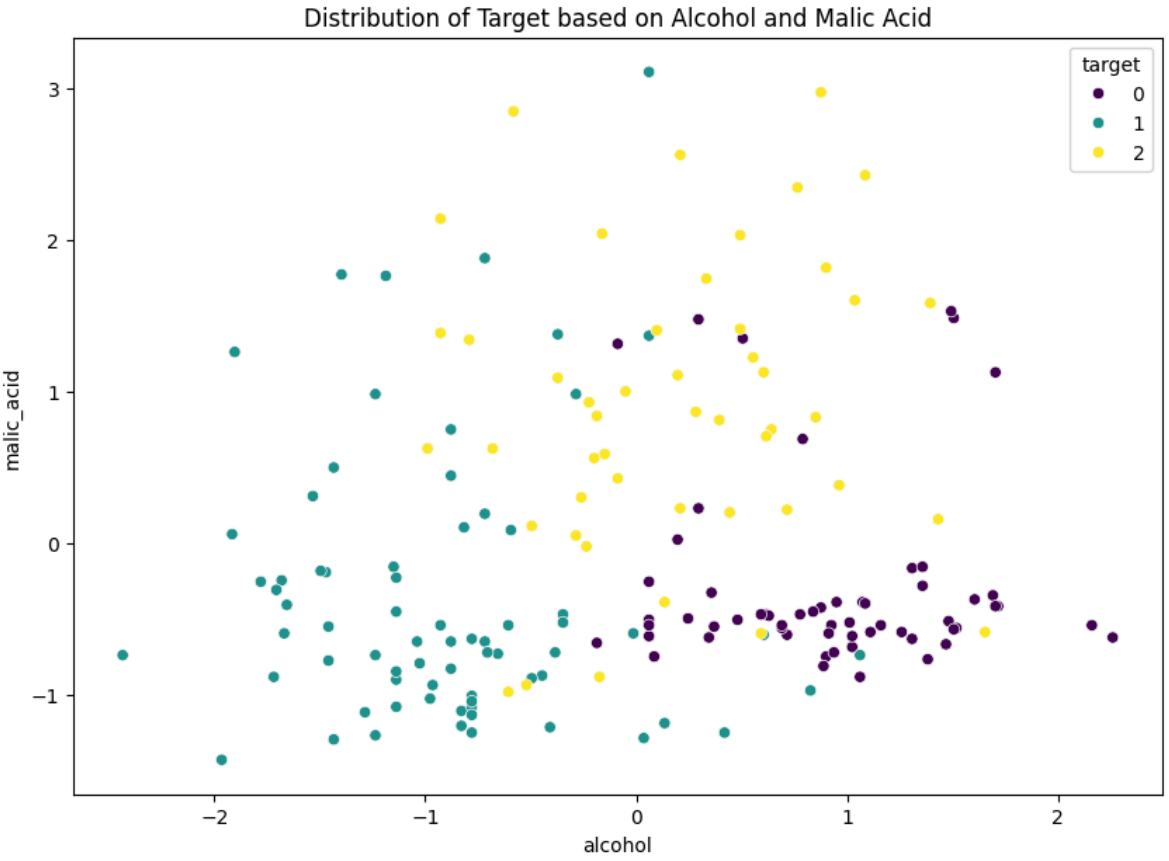
(1) S. Aeberhard, D. Coomans and O. de Vel,
Comparison of Classifiers in High Dimensional Settings,
Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and
Dept. of
Mathematics and Statistics, James Cook University of North
Queensland.
(Also submitted to Technometrics).

The data was used with many others for comparing various
classifiers. The classes are separable, though only RDA
has achieved 100 percent correct classification.
(RDA : 100 percent, QDA 99.4 percent, LDA 98.9 percent, 1NN 96.1
percent (z-transformed data))
(All results using the leave-one-out technique)

(2) S. Aeberhard, D. Coomans and O. de Vel,
"THE CLASSIFICATION PERFORMANCE OF RDA"
Tech. Rep. no. 92-01, (1992), Dept. of Computer Science and
Dept. of
Mathematics and Statistics, James Cook University of North
Queensland.
(Also submitted to Journal of Chemometrics).

|details-end|

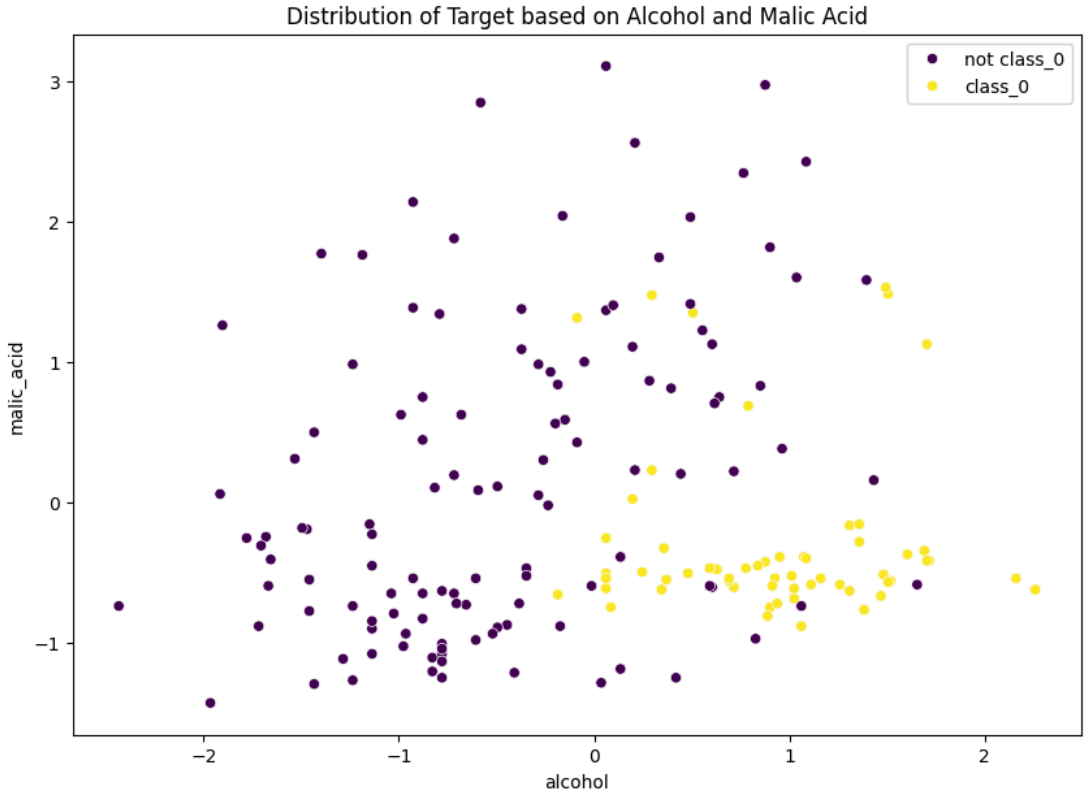
حال به بررسی پراکندگی داده‌ها بر اساس دو معیار `alcohol` و `malic_acid` می‌پردازیم.



شکل 3-3. نمودار پراکندگی داده‌ها بر اساس دو معیار `alcohol` و `malic_acid`

طبقه‌بندی (classification) برای کلاس `class_0`

چون ابتدا می‌خواهیم بر اساس `class_0` طبقه‌بندی (classification) کنیم نمودار را بر اساس داده‌هایی که عضو این کلاس هستند و نیستند تقسیم می‌کنیم:



شکل 3-4. نمودار پراکندگی داده‌ها بر اساس دو معیار `alcohol` و `malic_acid` و برای `class_0`

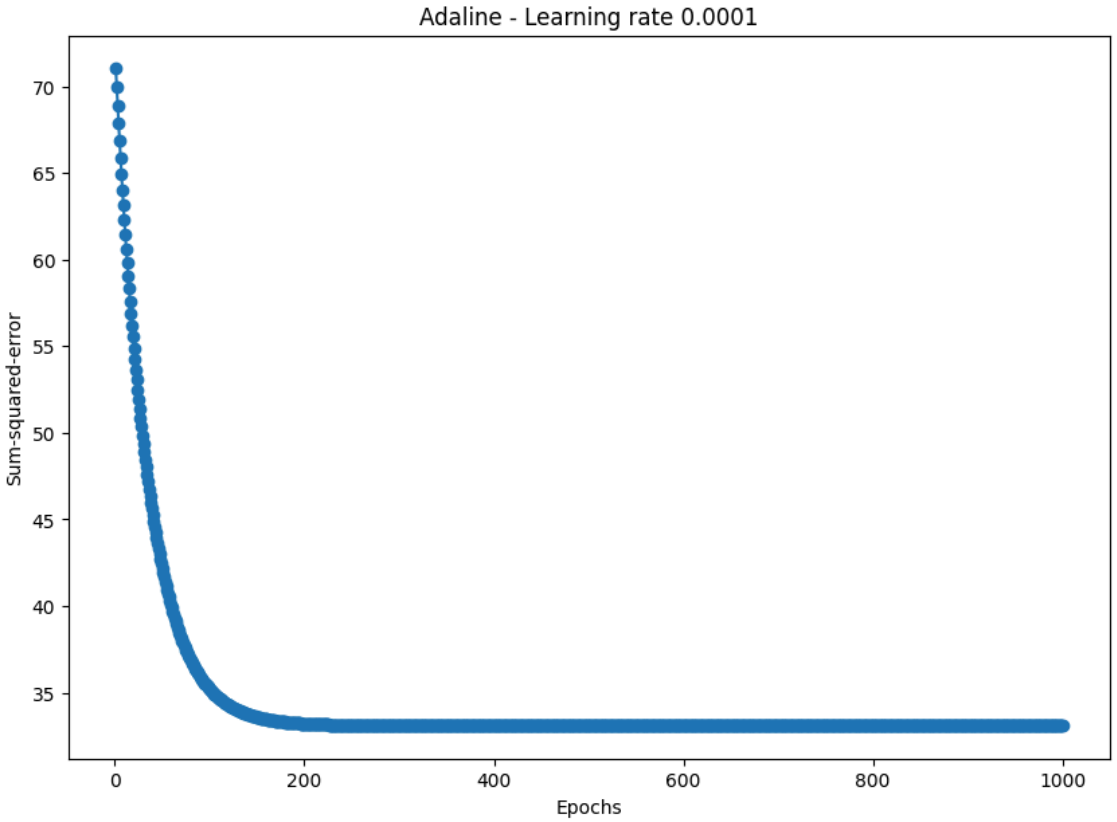
پیاده‌سازی

حال نورون Adeline را برای این منظور توسعه می‌دهیم:

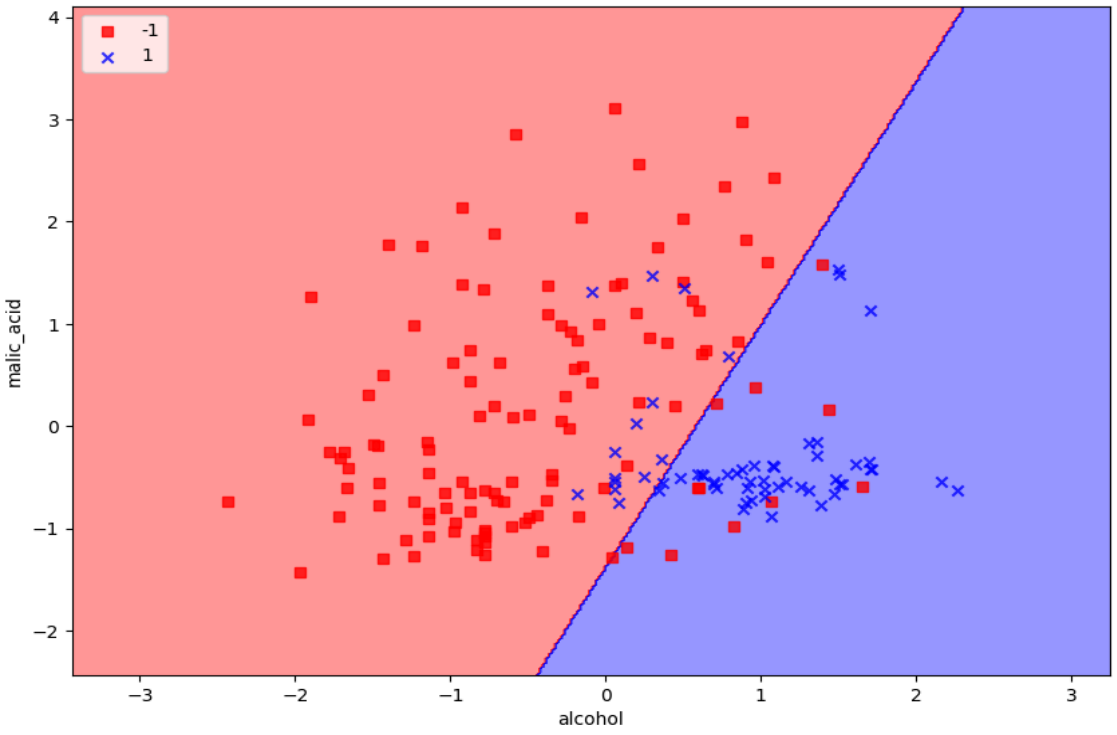
```
class Adaline:
      def __init__(self, learning_rate=0.01, n_iters=1000):
      self.learning_rate = learning_rate
      self.n_iters = n_iters

      def compile(self):
      pass

      def fit(self, x, y):
      self.weights = np.zeros(1 + x.shape[1]) + 0.01
      self.cost = []

      for _ in tqdm(range(self.n_iters)):
            output = self.net_input(x)
            errors = y - output
            self.weights[1:] += self.learning_rate *
x.T.dot(errors)
            self.weights[0] += self.learning_rate * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost.append(cost)

      def net_input(self, x):
      return np.dot(x, self.weights[1:]) + self.weights[0]

      def predict(self, x):
      self.predictions = np.where(self.net_input(x) >= 0.0, 1,
-1)
      return self.predictions
```

جدول 3-1. جدول دقت مدل برای class_0

```
+---------+--------------------+
| Dataset |      Accuracy      |
+---------+--------------------+
|  Train  | 0.852112676056338  |
|  Test   | 0.8888888888888888 |
+---------+--------------------+
```



شکل 3-5. نمودار تابع هزینه در گذر دوره‌ها (epochs) برای `class_0`



شکل 3-6. نتیجه مدل برای `class_0`

با توجه به اینکه تابع هزینه به راحتی کم شده و مدل دقت بالایی می‌توانیم نتیجه بگیریم که مدل به خوبی می‌تواند کلاس `class_0` از بقیه کلاس‌ها تشخیص دهد. این نتیجه را میتوان با بررسی شکل 3-6 که نشان می‌دهد نمونه‌ها تقریبا به صورت خطی جداپذیرند تایید کرد.

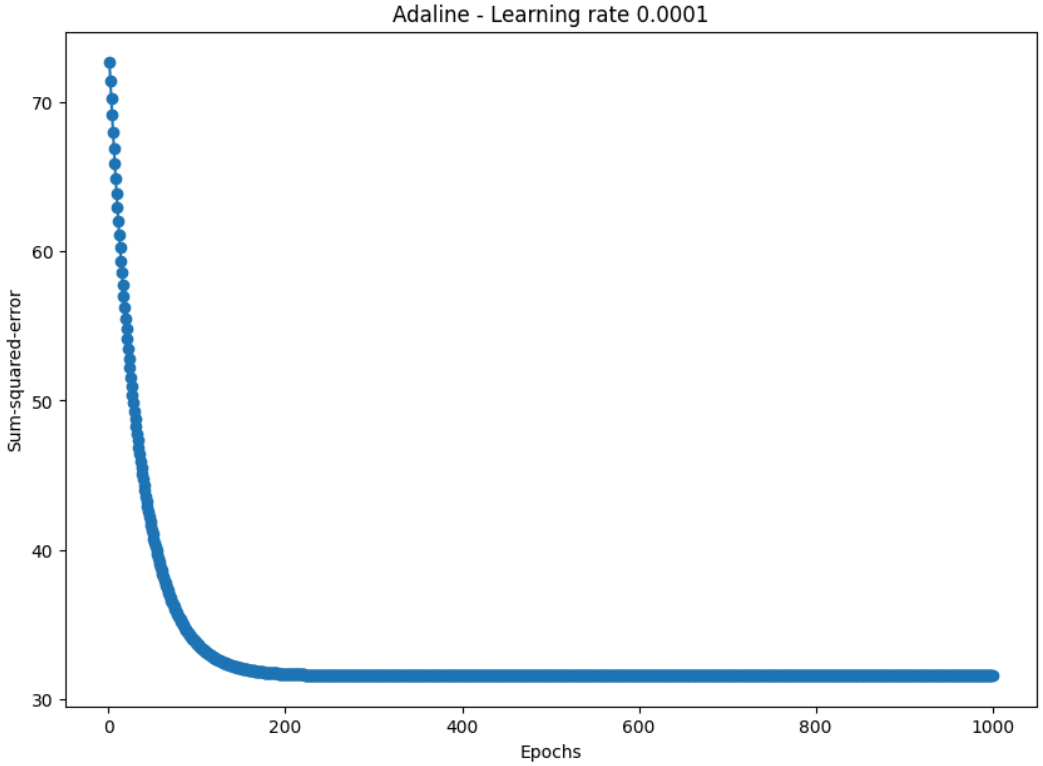
طبقه‌بندی (classification) برای کلاس `class_1`

سپس مراحل بالا را برای `class_1` نیز تکرار میکنیم:

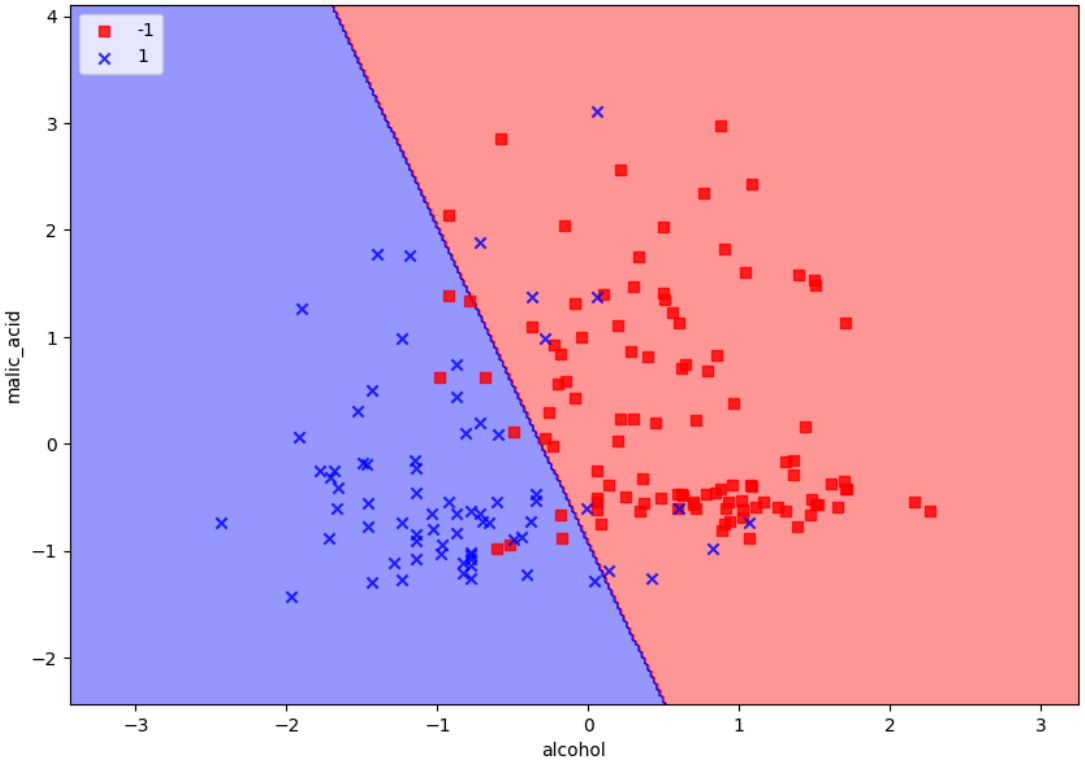
شکل 3-7. نمودار پراکندگی داده‌ها بر اساس دو معیار `alcohol` و `malic_acid` و برای `class_1`

جدول 3-2. جدول دقت مدل برای `class_1`

Dataset	Accuracy
Train	0.8732394366197183
Test	0.9444444444444444



شکل 3-8. نمودار تابع هزینه در گذر دوره‌ها (epochs) برای `class_1`



شکل 3-9. نتیجه مدل برای `class_1`

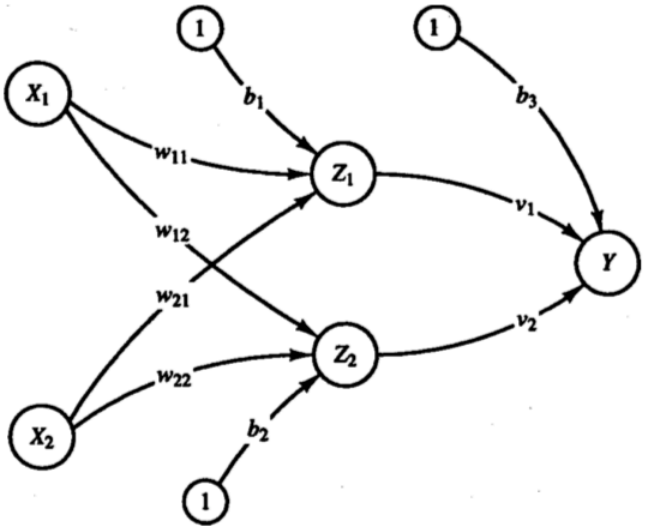
با توجه به اینکه تابع هزینه به راحتی کم شده و مدل دقت بالایی می‌توانیم نتیجه بگیریم که مدل به خوبی می‌تواند کلاس `class_1` از بقیه کلاس‌ها تشخیص دهد. این نتیجه را میتوان با بررسی شکل 3-9 که نشان می‌دهد نمونه‌ها تقریبا به صورت خطی جداپذیرند تایید کرد. اگر جدول 3-1 و 3-2 را با هم مقایسه کنیم به این نتیجه می‌رسیم که مدل در حالت دوم نتیجه بهتری داشته دلیل این است که در این حالت نمونه‌ها بیشتر به صورت خطی جداپذیرند و نیاز به مدل پیچیده‌ای برای جدا کردن آن‌ها نیست.

Madaline .2-3

نورون Madeline

Madaline یک نوع شبکه عصبی مصنوعی است. این نام مخفف Multiple Adaline است و به این معنی است که چندین نورون Adaline در یک شبکه با هم کار می‌کنند.

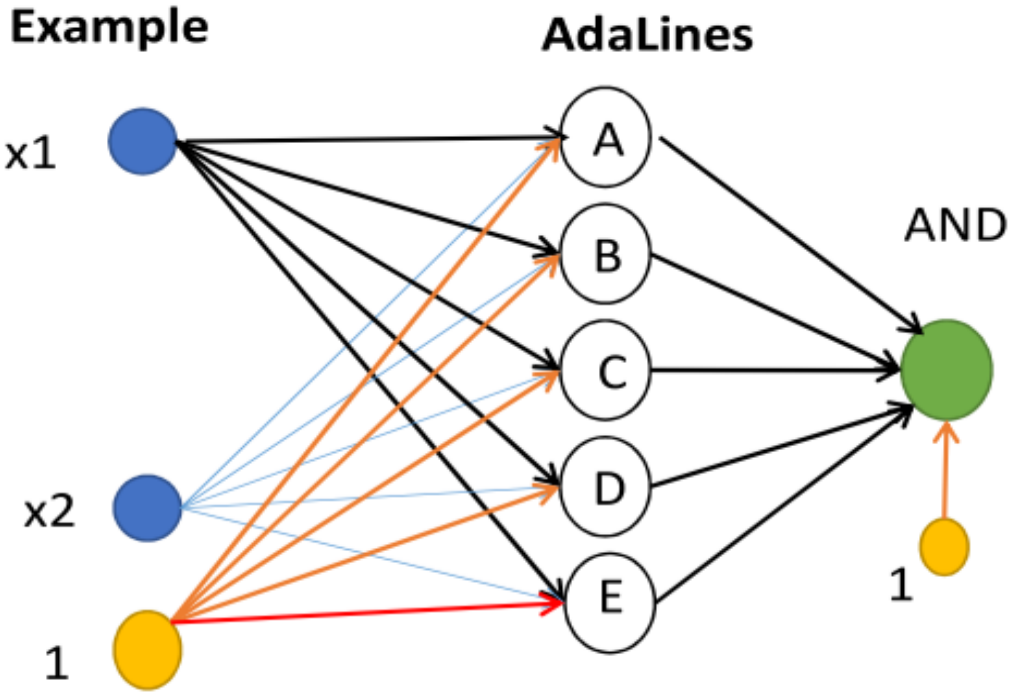
Madaline از یک لایه پنهان استفاده می‌کند. در این لایه، هر نورون Adaline به صورت مستقل از دیگران یاد می‌گیرد. سپس، خروجی‌های این نورون‌ها ترکیب می‌شوند تا خروجی نهایی شبکه را تولید کنند.



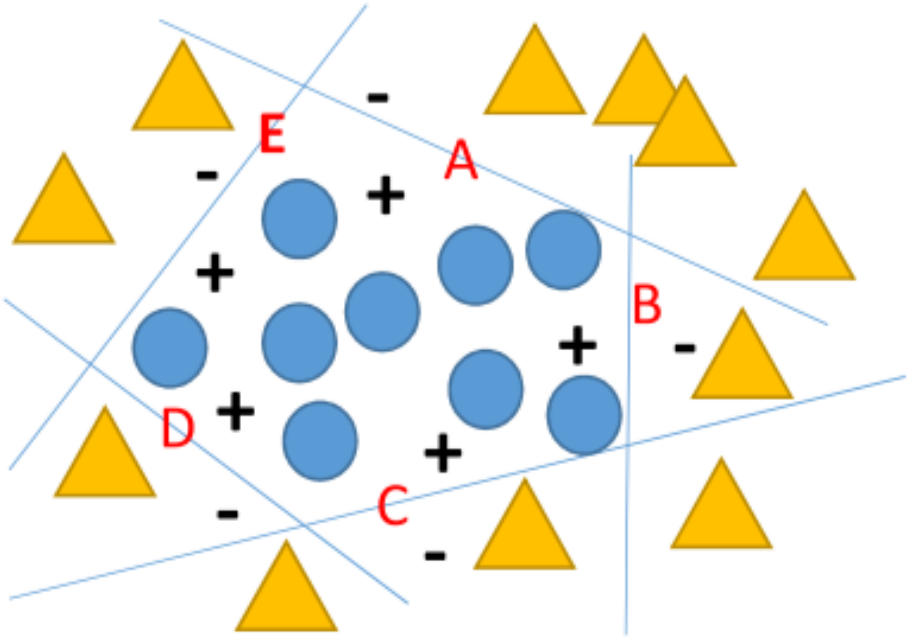
A Madaline with two hidden Adaline and one output Adaline.

شکل 3-10. شکل مدل Madline

کاربرد این مدل این است که زمان‌هایی که به صورت خطی جداپذیر نیستند و Adeline کار نمی‌کند می‌توان از این مدل کمک گرفت مثلا با مدل شکل 3-11 می‌توان داده‌های شکل 3-12 را طبقه‌بندی (classification) کرد.



شکل 3-11. مدل نمونه Madline



شکل 3-12. نتیجه طبقه‌بندی (classification) مدل شکل 3-11 روی یک نمونه

الگوریتم‌ها

برای اصلاح وزن‌ها در این نورون از دو الگوریتم رایج استفاده میشود به نام‌های MR-I و MR-II که توضیح هر کدام به شکل زیر است:

MR-I:

در این الگوریتم، برای ساخت شبکه Madaline از دو لایه استفاده می‌شود. لایه اول از تعدادی نورون Adaline استفاده می‌شود که هر کدام یک مرز تصمیم در داده پیدا می‌کند. لایه دوم برای نواحی ایجاد شده توسط مرزهای تصمیم ایجاد شده توسط لایه قبل تصمیم گیری می‌کند. معمولا لایه دوم عملکردی شبیه به یک عملگر منطقی همچون OR یا AND دارد.

در این الگوریتم، وزن‌های لایه اول متغیر و وزن‌های لایه دوم ثابت می‌باشند. پس از مشاهده هر نمونه، با توجه به درستی یا غلطی پیش‌بینی شبکه، وزن‌های نورون‌های لایه اول به‌روزرسانی می‌شوند.

این الگوریتم به این صورت است:

مرحله صفر: ابتدا با توجه به نیاز، وزن‌های لایه دوم را ثابت می‌کنیم. در اینجا برای پیاده سازی عملگر منطقی OR، وزن‌های شبکه لایه دوم +1 و بایاس آن (number of neurons - 1) در نظر گرفته می‌شود که تنها در صورتی خروجی -1 را ایجاد کند، که همه نورون‌ها خروجی -1 داشته باشند. وزن‌های لایه اول هم مانند الگوریتم Adaline می‌تواند مقادیر رندوم و ناچیزی باشد.

مرحله 1: شرط اتمام الگوریتم را چک می‌کنیم و درصورت خاتمه نیافتن به مرحله 2 می‌رویم.

مرحله 2: برای هر جفت ورودی(s) و لیبل(t) مراحل 3 تا 7 را انجام می‌دهیم.

مرحله 3: ورودی شبکه را مقداردهی می‌کنیم:

$$x_i = s_i$$

مرحله 4: مقدار ورودی هر نورون را با توجه به وزن‌های آن نورون محاسبه می‌کنیم:

$$z_{in} = b + \sum_{i=1}^{n} w_i x_i$$

مرحله 5: خروجی هر نورون را محاسبه می‌کنیم.(تابع فعال ساز Adaline)

$$z = f(z_{in})$$

مرحله 6: خروجی شبکه(لایه دوم) را محاسبه می‌کنیم.

$$y_{in} = b_{out} + \sum_{i=1}^{n} z_i v_i \; ; \; where\ b_{out} and\ v_i\ are\ bias\ and\ weights\ of\ second\ layer.$$

مرحله 7: خطا را محاسبه می‌کنیم. در 2 حالت خطا داریم:

- حالت اول: خروجی شبکه 1- باشد در حالی که مقدار درست +1 است: در این حالت باید نزدیک‌ترین وزن به صفر را به‌روزرسانی کنیم(زیرا تنها در صورتی خروجی شبکه -1 است که تمامی نورون‌های لایه قبل خروجی -1 داشته باشند.)

$$w_{iJ} = max(w_k)$$

$$w_{iJ} = w_{iJ} + \alpha(1 - z_{inJ}) \times x_i$$

$$b_J = b_J + \alpha(1 - z_{inJ})$$

- حالت دوم: خروجی شبکه 1+ باشد در حالی که مقدار درست -1 است: باید تمامی وزن‌های مثبت را کم کنیم تا شبکه به خروجی درست همگرا شود.

$$w_{iJ} = w_k(w_k > 0)$$

$$w_{iJ} = w_{iJ} + \alpha(-1 - z_{inJ}) \times x_i$$

$$b_J = b_J + \alpha(-1 - z_{inJ})$$

We consider first the MRI algorithm; the weights v_1 and v_2 and the bias b_3 that feed into the output unit Y are determined so that the response of unit Y is 1 if the signal it receives from either Z_1 or Z_2 (or both) is 1 and is -1 if both Z_1 and Z_2 send a signal of -1. In other words, the unit Y performs the logic function OR on the signals it receives from Z_1 and Z_2. The weights into Y are

$$v_1 = \frac{1}{2}$$

and

$$v_2 = \frac{1}{2},$$

with the bias

$$b_3 = \frac{1}{2}$$

(see Example 2.19). The weights on the first hidden Adaline (w_{11} and w_{21}) and the weights on the second hidden Adaline (w_{12} and w_{22}) are adjusted according to the algorithm.

شکل 3-13. توضیحات الگوریتم MR-I در کتاب مرجع (Fundementals on Neural Networks) بخش اول

Training Algorithm for Madaline (MRI). The activation function for units Z_1, Z_2, and Y is

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0; \\ -1 & \text{if } x < 0. \end{cases}$$

Step 0. Initialize weights:
Weights v_1 and v_2 and the bias b_3 are set as described;
small random values are usually used for **Adaline** weights.
Set the learning rate α as in the **Adaline** training algorithm (a small value).

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each bipolar training pair, s:t, do Steps 3–7.

Step 3. Set activations of input units:

$$x_i = s_i.$$

Step 4. Compute net input to each hidden **Adaline** unit:

$$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21},$$

$$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}.$$

Step 5. Determine output of each hidden **Adaline** unit:

$$z_1 = f(z\_in_1),$$

$$z_2 = f(z\_in_2).$$

Step 6. Determine output of net:

$$y\_in = b_3 + z_1 v_1 + z_2 v_2;$$

$$y = f(y\_in).$$

Step 7. Determine error and update weights:
If $t = y$, no weight updates are performed.
Otherwise:
If $t = 1$, then update weights on Z_J,
the unit whose net input is closest to **0**,

$$b_J(\text{new}) = b_J(\text{old}) + \alpha(1 - z\_in_J),$$

$$w_{iJ}(\text{new}) = w_{iJ}(\text{old}) + \alpha(1 - z\_in_J)x_i;$$

If $t = -1$, then update weights on all units Z_k that have positive net input,

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z\_in_k),$$

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z\_in_k)x_i.$$

شکل 3-14. توضیحات الگوریتم MR-I در کتاب مرجع (Fundementals on Neural Networks) بخش دوم

MR-II:

این الگوریتم حالت تعمیم یافته MR-I هست به طوری که وزن‌های لایه دوم آن قابل تغییر است، بدین شکل می‌تواند حالت های پیچیده تری را تشخیص دهد.

A more recent MADALINE training rule, called MRII [Widrow, Winter, & Baxter, 1987], allows training for weights in all layers of the net. As in earlier MADALINE training, the aim is to cause the least disturbance to the net at any step of the learning process, in order to cause as little "unlearning" of patterns for which the net had been trained previously. This is sometimes called the "don't rock the boat" principle. Several output units may be used; the total error for any input pattern (used in Step 7b) is the sum of the squares of the errors at each output unit.

Training Algorithm for Madaline (MRII).

Step 0. Initialize weights:
Set the learning rate α.

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each bipolar training pair, s:t, do Steps 3–7.

Step 3–6. Compute output of net as in the MRI algorithm.

Step 7. Determine error and update weights if necessary:

If $t \neq y$, do Steps 7a–b for each hidden unit whose net input is sufficiently close to 0 (say, between $-.25$ and .25). Start with the unit whose net input is closest to 0, then for the next closest, etc.

Step 7a. Change the unit's ouput
(from $+1$ to -1, or vice versa).

Step 7b. Recompute the response of the net.
If the error is reduced:
adjust the weights on this unit
(use its newly assigned output value
as target and apply the Delta Rule).

Step 8. Test stopping condition.
If weight changes have stopped (or reached an acceptable level), or if a specified maximum number of weight update iterations (Step 2) have been performed, then stop; otherwise continue.

A further modification is the possibility of attempting to modify pairs of units at the first layer after all of the individual modifications have been attempted. Similarly adaptation could then be attempted for triplets of units.

شکل 3-15. توضیحات الگوریتم MR-II در کتاب مرجع (Fundementals on Neural Networks) بخش اول

شکل 3-16. توضیحات الگوریتم MR-I و MR-II در کتاب مرجع (Fundementals on Neural Networks) بخش دوم

پیاده‌سازی

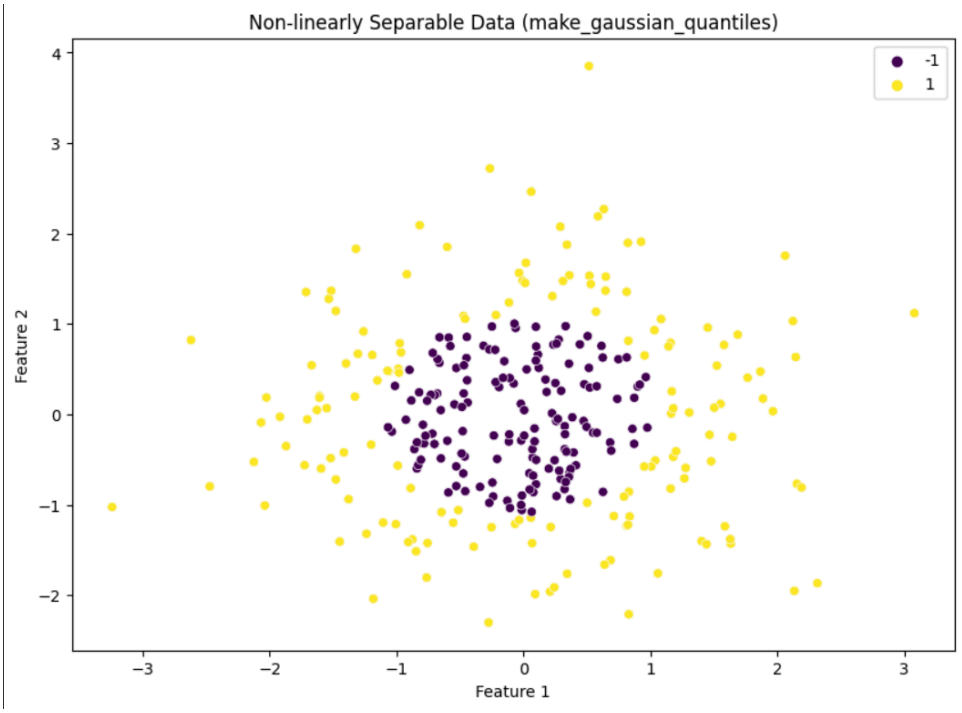
برای درک بهتر به پیاده سازی تابع fit شبکه Madaline بر اساس الگوریتم MR-I نگاه کنید:

```
def fit(self, X, Y):
        self.weights = np.zeros(shape = (self.n_neurons, X.shape[1])) + 0.01
        self.bias = np.zeros(self.n_neurons) + 0.01
        self.cost_per_epoch = []

        for _ in tqdm(range(self.n_iters)):
            cost = 0
            for x, label in zip(X, Y):
                z_in = np.sum(x * self.weights, axis = 1) + self.bias
                z = self._activation(z_in)

                y_in = np.dot(z, self.output_layer_weights)+self.output_layer_bias
                y = self._activation(y_in)

                cost += ((label - y) ** 2) / 2.0

                if(label == 1 and y == -1):
                    to_update = z_in.argmax()
                    self.weights[to_update, :] += self.learning_rate * (1 -
z_in[to_update]) * x
                    self.bias[to_update] += self.learning_rate * (1 -
z_in[to_update])

                elif(label == -1 and y == 1):
                    to_update = (z_in > 0)
                    n = to_update.sum()

                    x1 = np.reshape(-1 - z_in[to_update], (n, 1))
                    x2 = np.tile(x, (n, 1))

                    self.weights[to_update, :] += self.learning_rate * (x1 * x2)
                    self.bias[to_update] += self.learning_rate * (-1 -
z_in[to_update])


            self.cost_per_epoch.append(cost)
        return self
                y_in = np.dot(z, self.output_layer_weights) +
self.output_layer_bias
```

```
                y = self._activation(y_in)

                cost += ((label - y) ** 2) / 2.0

                if(label == 1 and y == -1):
                    to_update = z_in.argmax()
                    self.weights[to_update, :] += self.learning_rate * (1 -
z_in[to_update]) * x
                    self.bias[to_update] += self.learning_rate * (1 -
z_in[to_update])

                elif(label == -1 and y == 1):
                    to_update = (z_in > 0)
                    n = to_update.sum()

                    x1 = np.reshape(-1 - z_in[to_update], (n, 1))
                    x2 = np.tile(x, (n, 1))

                    self.weights[to_update, :] += self.learning_rate * (x1 * x2)
                    self.bias[to_update] += self.learning_rate * (-1 -
z_in[to_update])


            self.cost_per_epoch.append(cost
```

دیتاست

مجموعه داده زیر، به وسیله تصمیم گیرنده خطی قابل تشخیص نیست، اما از ترکیب چندین جدا کننده می‌توان تصمیم بهتری گرفت. این مجموعه داده `make_gaussian_quantile` نام دارد و در کتابخانه `scikit-learn` قرار دارد.

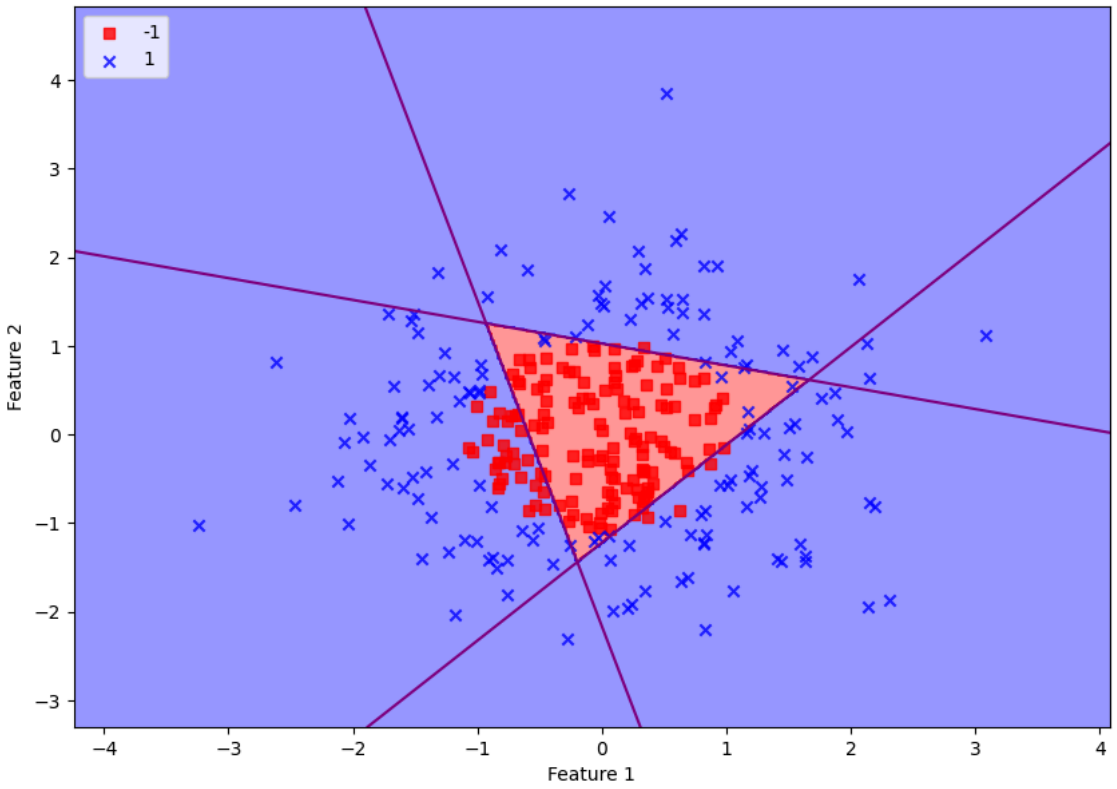


شکل 3-17. توزیع دیتاست مورد بررسی

بررسی مدل

برای نشان دادن بهتر نحوه کارکرد این شبکه، آن را با تعداد مختلفی نورون در لایه اول، بررسی می کنیم:

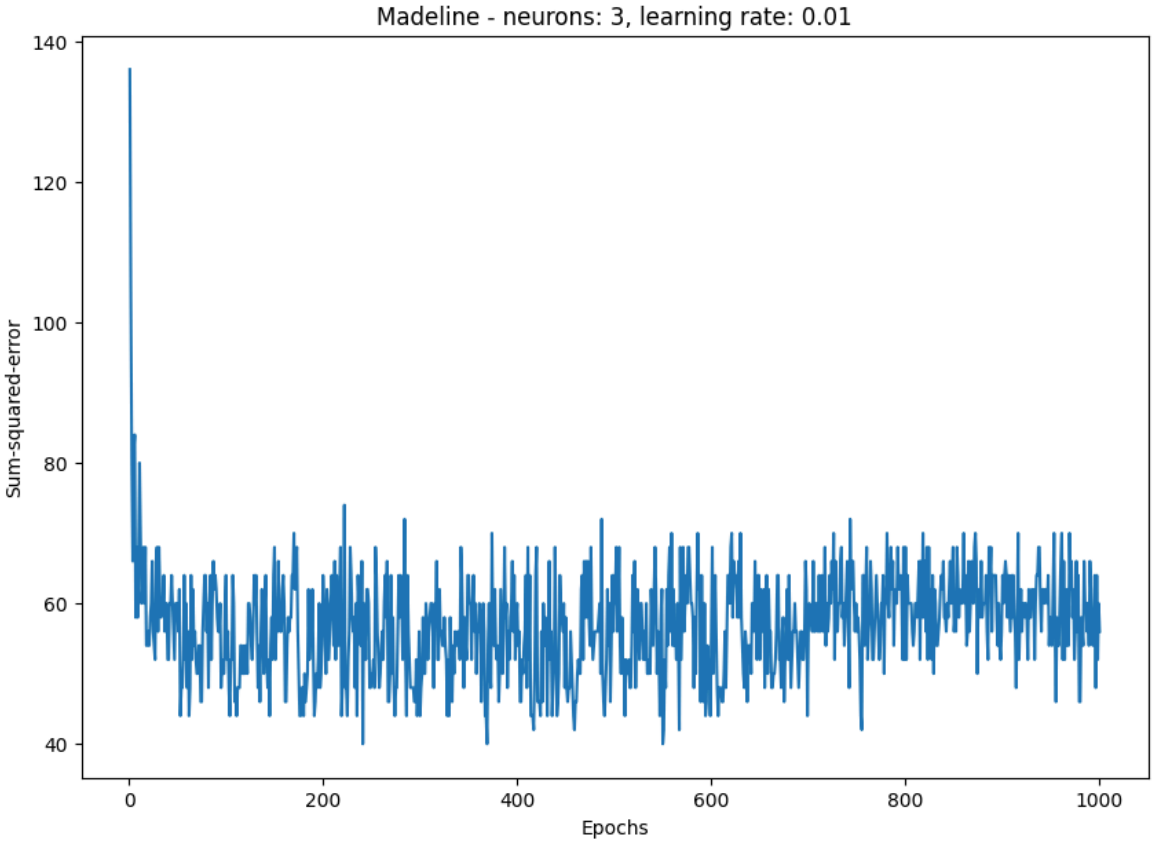
- **حالت اول: 3 نورون**



شکل 3-18. نتیجه طبقه‌بندی (classification) در حالت اول با 3 نورون

جدول 3-1. دقت مدل در حالت اول با 3 نورون

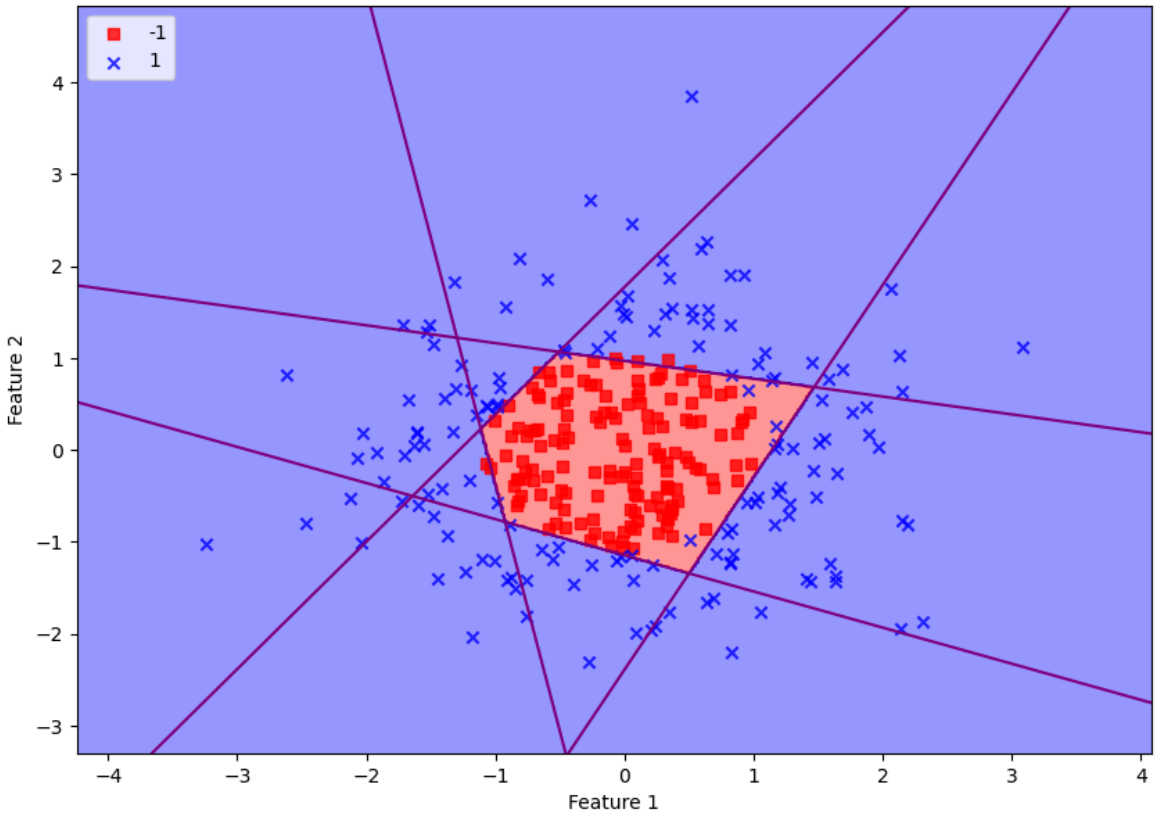
Dataset	Accuracy
Train	0.8625
Test	0.7833333333333333



شکل 3-19. نتیجه تغییر خطا مدل طبقه‌بندی (classification) در حالت اول با 3 نورون

در این حالت به دلیل اینکه نورون کافی برای طبقه‌بندی نداریم مدل دقت چندانی ندارد اما باز تا حد خوبی طبقه‌بندی را انجام می‌دهد.

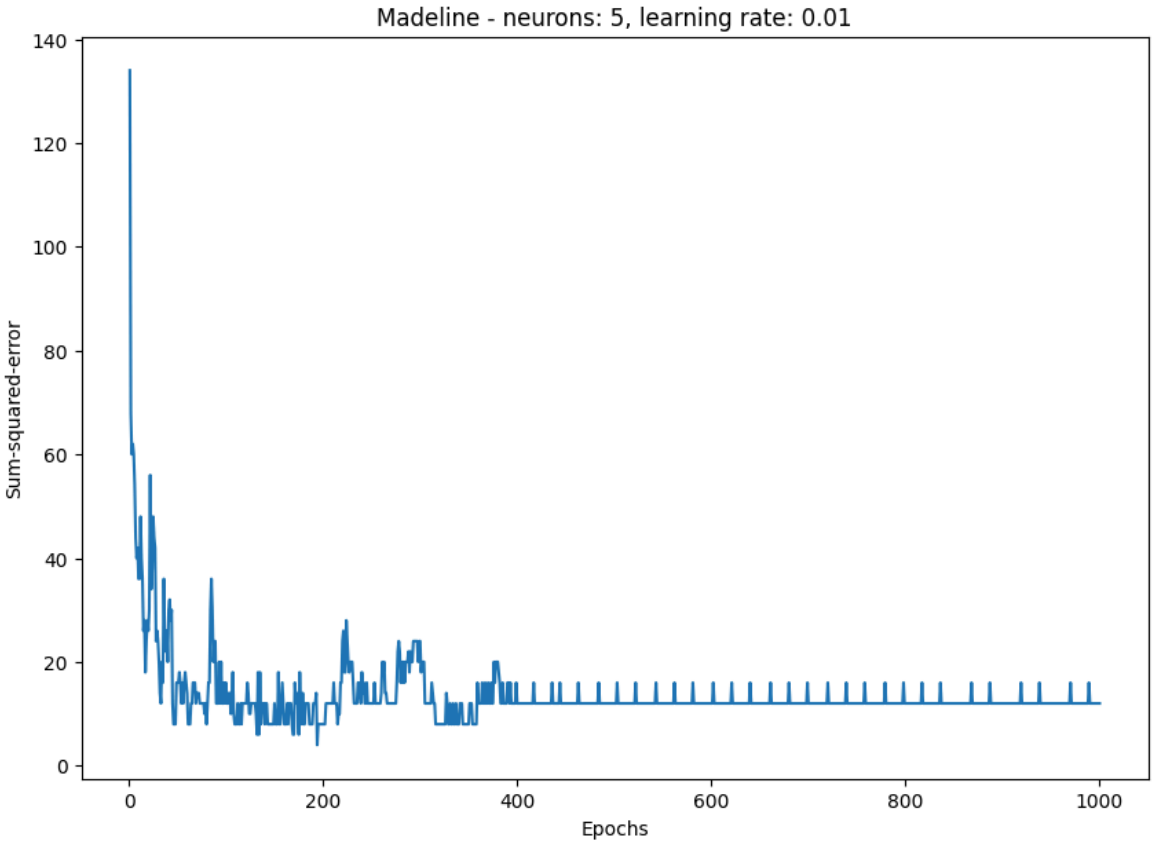
- حالت دوم: 5 نورون



شکل 3-20. نتیجه طبقه‌بندی (classification) در حالت دوم با 5 نورون

جدول 3-2. دقت مدل در حالت دوم با 5 نورون

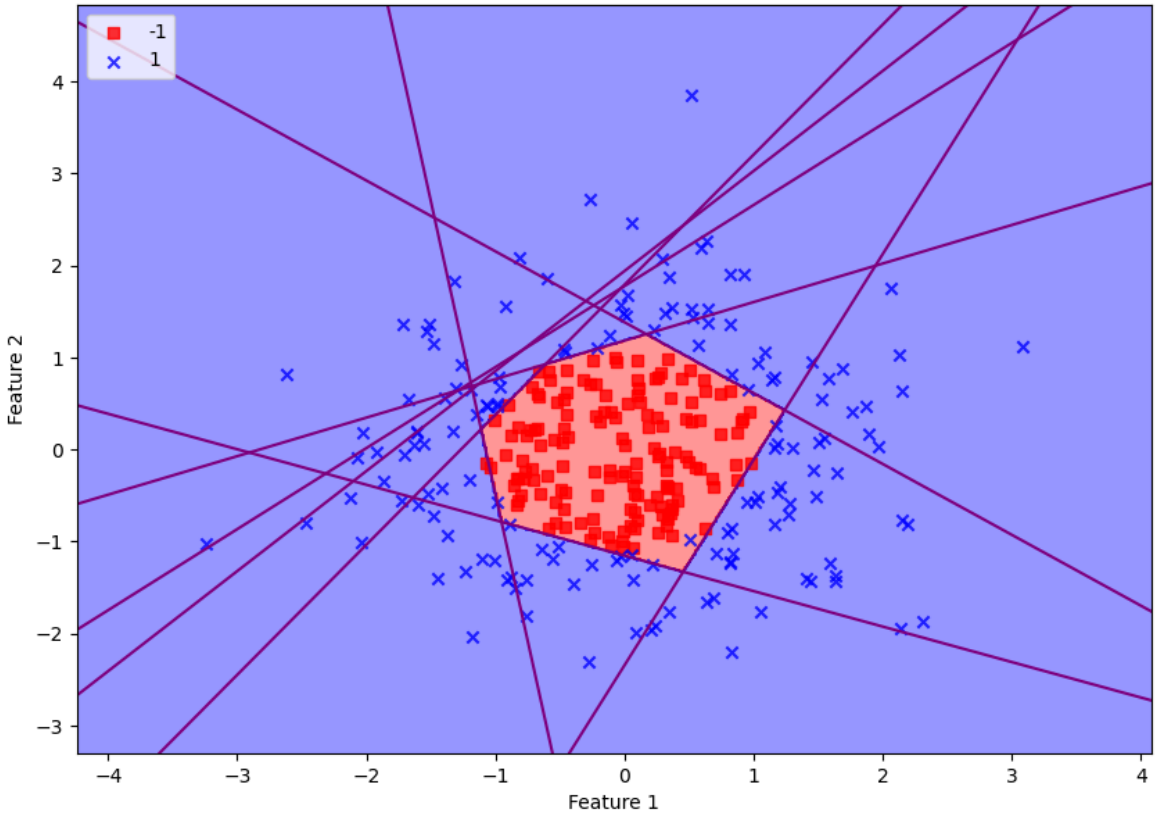
Dataset	Accuracy
Train	0.9791666666666666
Test	0.9166666666666666



شکل 3-21. نتیجه تغییر خطا مدل طبقه‌بندی (classification) در حالت دوم با 5 نورون

در این حالت نورون‌های بیشتری داریم و مدل با دقت بهتری می‌تواند طبقه‌بندی کند اما هنوز کافی نیست و دقت لازم را ندارد.

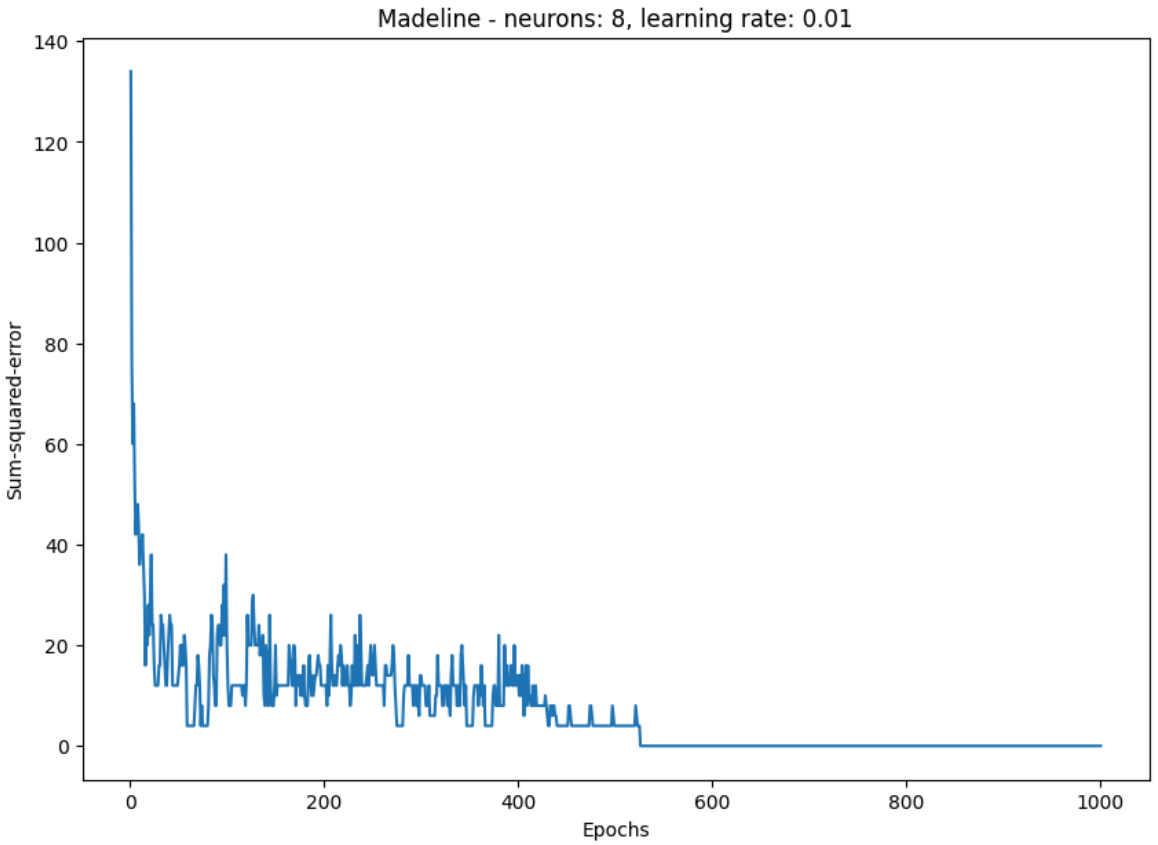
- حالت سوم: 8 نورون



شکل 3-22. نتیجه طبقه‌بندی (classification) در حالت سوم با 8 نورون

جدول 3-3. دقت مدل در حالت سوم با 8 نورون

Dataset	Accuracy
Train	1.0
Test	0.95



شکل 3-21. نتیجه تغییر خطا مدل طبقه‌بندی (classification) در حالت سوم با 8 نورون

اگر تغییر دقت مدل‌ها بر اساس تعداد را نورون را بررسی کنیم به این نتیجه می‌رسیم در حالتی که نورون‌های کافی نداریم (مانند 3 نورون) مدل تا حد امکان تلاش به طبقه‌بندی می‌کند اما به دلیل محدودیت مدل نمی‌تواند زیاد موفق باشد و برای اینکه مدل بتواند به درستی خط‌های مرزی جداکننده را رسم کند نیاز داریم که از تعداد بیشتری نورون استفاده کنیم.

پرسش 4 – شبکه عصبی بهینه

4-1. رگرشن (regression)

برازش بیش از حد (overfitting)

برازش بیش از حد (overfitting) در شبکه‌های عصبی، وقتی رخ می‌دهد که مدل به داده‌های آموزشی بیش از حد خوب سازگار شده و عملکرد خوبی روی داده‌های آموزشی دارد، اما روی داده‌های تست یا داده‌های جدید، عملکرد ضعیفی دارد. این مسئله نشان‌دهنده این است که مدل قادر به تعمیم داده‌ها به خوبی نیست و فقط قادر به بازتولید پاسخ‌هایی است که برای داده‌های آموزشی دیده است.

دلایل برازش بیش از حد (overfitting)

دلایل اصلی بروز برازش بیش از حد (overfitting) در شبکه‌های عصبی عبارتند از:

1. **پیچیدگی بیش از حد مدل**: وقتی یک مدل دارای تعداد زیادی پارامتر (مثل وزن‌ها و بایاس‌ها در شبکه عصبی) است، می‌تواند به داده‌های آموزشی بسیار خوب سازگار شود، اما این سازگاری ممکن است به خاطر یادگیری نویزهای موجود در داده‌های آموزشی باشد که باعث می‌شود مدل نتواند به خوبی به داده‌های جدید تعمیم یابد.
2. **کمبود داده‌های آموزشی**: وقتی داده‌های آموزشی کافی برای یادگیری الگوهای موجود در داده وجود ندارد، مدل ممکن است نتواند الگوهای کلی را یاد بگیرد و فقط به داده‌های آموزشی خاص سازگار شود.
3. **کمبود تنظیم (Regularization)**: تکنیک‌های تنظیم مانند L1 و L2 Regularization، Dropout و Early Stopping می‌توانند به کاهش برازش بیش از حد (overfitting) کمک کنند. این تکنیک‌ها با اعمال محدودیت بر روی وزن‌ها یا تعداد دوره‌های آموزش، می‌توانند جلوی سازگاری بیش از حد مدل با داده‌های آموزشی را بگیرند.
4. **کمبود داده‌های ورودی**: اگر تنوع کافی در داده‌های ورودی وجود نداشته باشد، مدل ممکن است نتواند الگوهای کلی را یاد بگیرد و فقط به داده‌های خاصی که در داده‌های آموزشی دیده است، سازگار شود.

مقابله با برازش بیش از حد (overfitting)

برای مقابله با برازش بیش از حد (overfitting) در شبکه‌های عصبی، می‌توان از روش‌های زیر استفاده کرد:

1. **تنظیم (Regularization)**: تکنیک‌های تنظیم مانند L1 و L2 Regularization به کاهش برازش بیش از حد (overfitting) کمک می‌کنند. این تکنیک‌ها با اعمال محدودیت بر روی وزن‌ها، جلوی سازگاری بیش از حد مدل با داده‌های آموزشی را می‌گیرند.
2. **افزایش داده (Data Augmentation)**: این روش با ایجاد تغییرات کوچک و غیرمعنی‌دار بر روی داده‌های آموزشی (مانند چرخش، بزرگنمایی، انتقال، و غیره)، تعداد داده‌های آموزشی را افزایش می‌دهد و به مدل کمک می‌کند تا الگوهای کلی‌تری را یاد بگیرد.
3. **استفاده از مدل‌های ساده‌تر**: مدل‌های ساده‌تر با تعداد کمتری پارامتر، کمتر مستعد برازش بیش از حد (overfitting) هستند. بنابراین، انتخاب یک مدل مناسب با تعداد مناسبی پارامتر می‌تواند به کاهش برازش بیش از حد (overfitting) کمک کند.
4. **Dropout**: این یک تکنیک تنظیم است که در طول فرآیند آموزش، برخی از نورون‌ها را به صورت تصادفی "خاموش" می‌کند. این باعث می‌شود شبکه عصبی بتواند بهتر تعمیم یابد و از برازش بیش از حد (overfitting) جلوگیری کند.
5. **Early Stopping**: در این روش، فرآیند آموزش زمانی متوقف می‌شود که عملکرد مدل روی داده‌های اعتبارسنجی دیگر بهبود نیابد. این روش جلوی برازش بیش از حد (overfitting) را می‌گیرد زیرا مانع از این می‌شود که مدل بیش از حد به داده‌های آموزشی سازگار شود.
6. **Batch Normalization**: این روش با نرمال‌سازی خروجی لایه‌ها، می‌تواند به کاهش برازش بیش از حد (overfitting) کمک کند.
7. **Ensemble Methods**: این روش‌ها با ترکیب چندین مدل مختلف، می‌توانند به کاهش برازش بیش از حد (overfitting) کمک کنند. مثلاً می‌توان چندین مدل با پیکربندی‌های مختلف آموزش داد و سپس خروجی نهایی را با میانگین گیری از خروجی‌های هر مدل به دست آورد.

هایپرپارامترها (hyperparameters)

هایپرپارامترها (hyperparameters) در شبکه‌های عصبی، پارامترهایی هستند که قبل از آموزش مدل تعیین می‌شوند و در فرآیند یادگیری تغییر نمی‌کنند. این پارامترها می‌توانند تأثیر زیادی بر عملکرد مدل داشته باشند. برخی از هایپرپارامترها (hyperparameters)ی متداول در شبکه‌های عصبی عبارتند از:

1. **نرخ یادگیری**: این پارامتر میزان تغییر وزن‌ها در هر بروزرسانی را کنترل می‌کند.
2. **تعداد لایه‌ها**: تعداد لایه‌های مخفی در شبکه عصبی.
3. **تعداد نورون‌ها**: تعداد نورون‌ها در هر لایه مخفی.
4. **تابع فعال‌سازی**: تابعی که بر روی خروجی هر نورون اعمال می‌شود.
5. **تعداد دوره‌ها (Epochs)**: تعداد دفعاتی که الگوریتم یادگیری کل داده‌های آموزشی را مرور می‌کند.
6. **اندازه دسته (Batch Size)**: تعداد نمونه‌های آموزشی که در هر بروزرسانی وزن‌ها در نظر گرفته می‌شود.

تعداد هایپرپارامترها (hyperparameters) به تعداد پارامترهایی اشاره دارد که قبل از آموزش باید تنظیم شوند. برای مثال، اگر شبکه عصبی ما دارای 2 لایه مخفی با 50 نورون در هر لایه و یک نرخ یادگیری ثابت است، ما دارای 3 هایپرپارامتر هستید: تعداد لایه‌ها، تعداد نورون‌ها در هر لایه، و نرخ یادگیری.

مقدار دهی هایپرپارامترها (hyperparameters) به معنی تنظیم و تعیین بهترین مقادیر برای این پارامترها است. این فرآیند می‌تواند به صورت دستی، شبکه گرید (Grid Search)، جستجوی تصادفی، یا با استفاده از الگوریتم‌های بهینه‌سازی مانند جستجوی بیزی انجام شود. هدف از این فرآیند بهبود عملکرد مدل بر روی داده‌های تست یا اعتبارسنجی است.

دیتاست

برای مشاهده تاثیر هایپرپارامترها (hyperparameters) در شبکه، ابتدا یک دیتاست سینوسی تولید می‌کنیم، به طوری که x هر نقطه ویژگی آن و y آن هدف ما می‌باشد.

بررسی مدل‌ها

افزایش نسبت داده‌های آموزش و آزمایش (train_test_ratio)

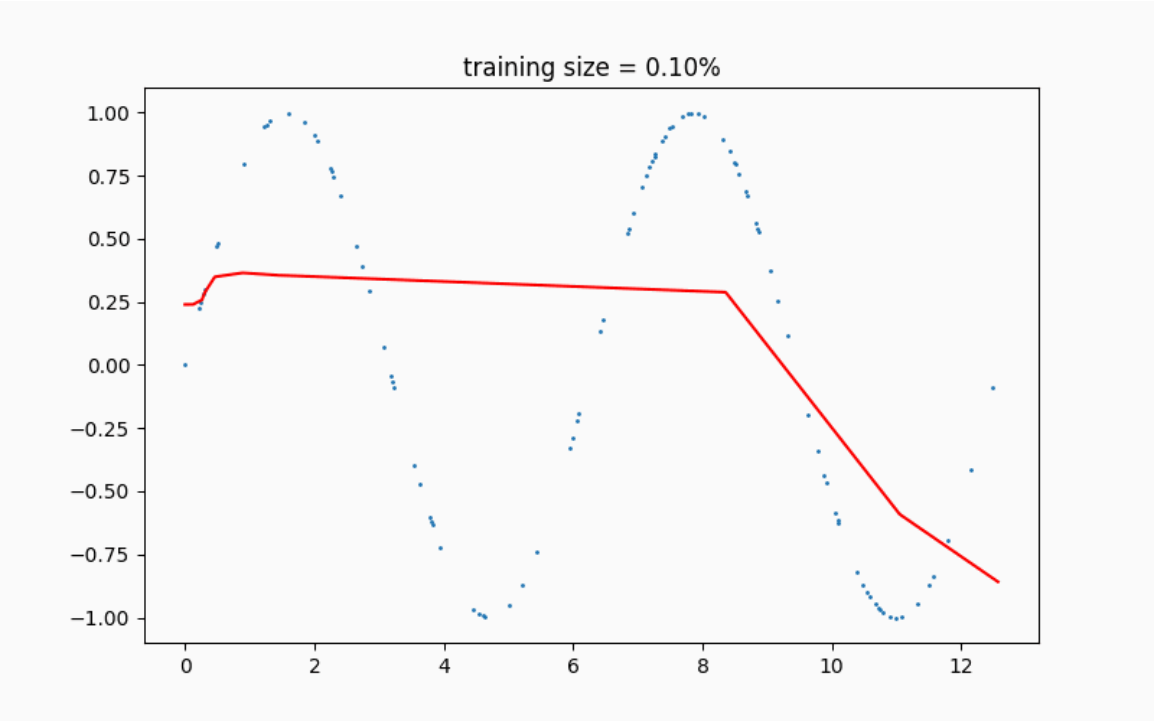
برای مشاهده تاثیر افزایش داده‌ها، از 10 درصد شروع کرده و هربار 10 درصد به داده‌های تست می‌افزاییم. برای این کار باید به 2 نکته توجه کنیم:

1. در هر افزایش، داده‌های قبلی را حفظ کنیم به طوری که همواره داده درصد های پایین‌تر زیرمجموعه داده‌هایی با درصد بالاتر باشند.
2. برای هر کدام یک مدل جدید درست می‌کنیم، زیرا مدل قبلی داده‌های قبلی را یکبار train کرده است.

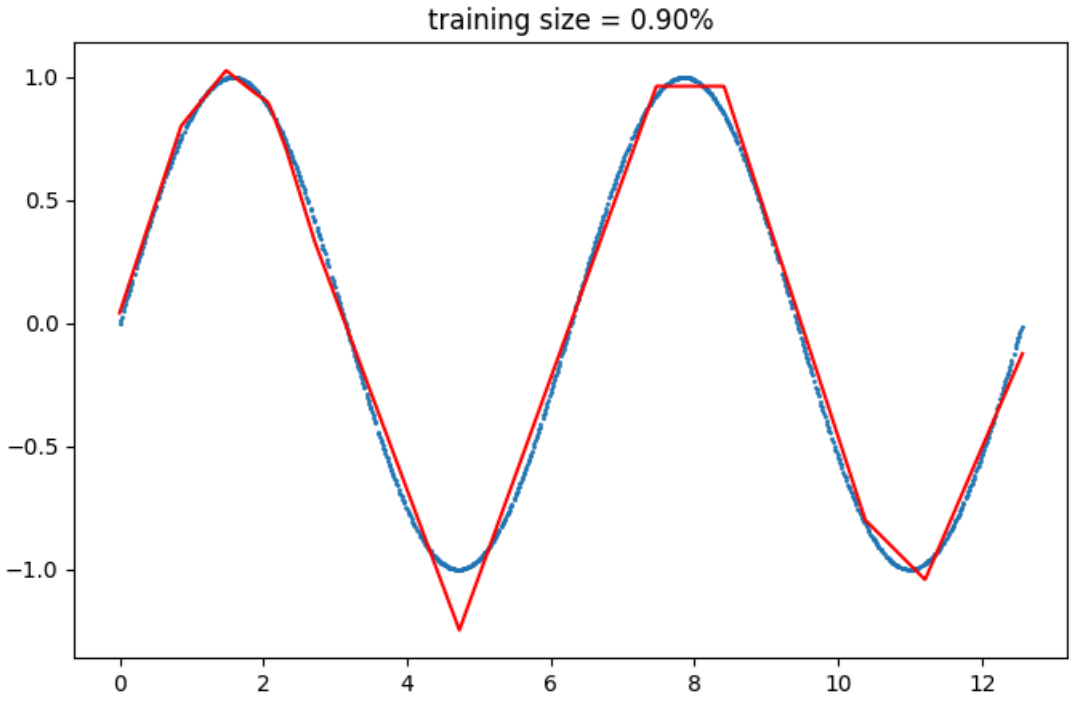
شبکه مورد استفاده:

```
def create_model(n_hidden_layers):
    model = Sequential()
    model.add(Dense(units=12, activation="relu", input_dim=1))
    for _ in range(n_hidden_layers):
        model.add(Dense(units=8, activation="relu"))
    model.add(Dense(units=1, activation="linear"))

    model.compile(
        optimizer="adam",
        loss="mean_squared_error",
    )

    return model

model = create_model(n_hidden_layers = 2)
model.fit(x_train, y_train, epochs=400, verbose=0,
batch_size=16)
```

برای یادگیری بهتر مدل‌مان، ما batch_size کم‌تر با تعداد دوره ی زیادی انتخاب کردیم تا شبکه تابع سینوسی را یاد بگیرد و نتیجه مطلوبی داشتیم.



شکل 4-1. نتیجه مدل با `test_size=0.9`



شکل 4-2. نتیجه مدل با `test_size=0.1`

برای ارزیابی مدل های Regression از MSE و R2-Score استفاده می‌شود.

- خطای میانگین مجموع مربعات(MSE):

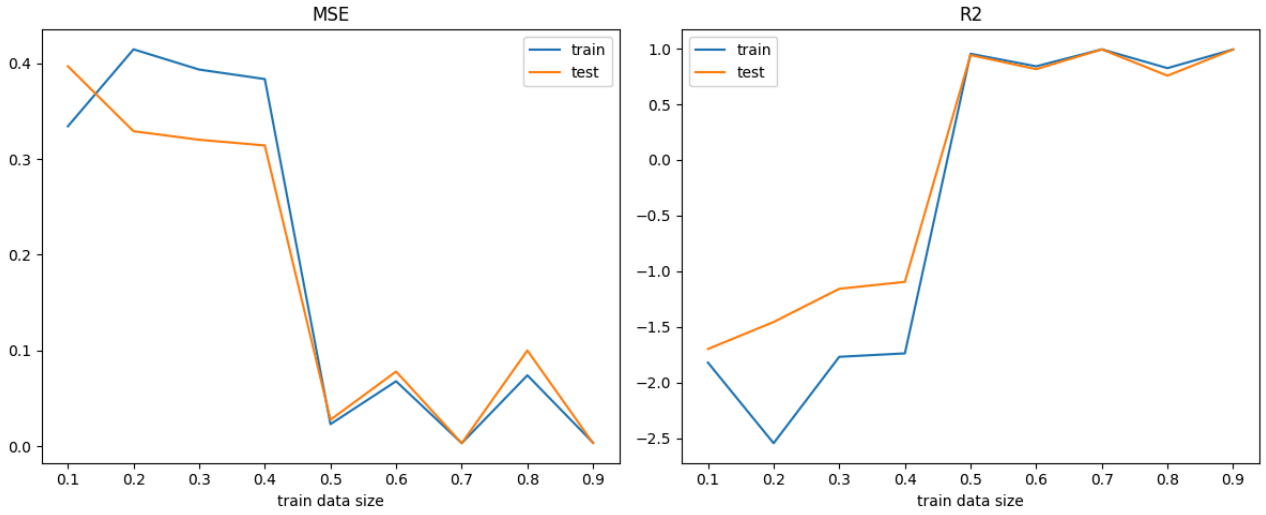
در این خطا اختلاف عدد پیش‌بینی شده و مقدار واقعی آن لحاظ می‌شود.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

- امتیاز R2:

معیاری برای متغیر های عددی که از 1 کوچک تر است. بیشتر بودن این عدد، برازی بهتری را نشان می‌دهد.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{\sum_{i=1}^{n}(y_i - \bar{y_i})^2}$$



شکل 4-3. تغییر خطای مدل با نسبت‌های مختلف داده برای آزمایش و آزمون

با افزایش داده‌های آموزش، دقت مدل بیشتر شده است، اما این افزایش از درصدی به بعد دیگر آنچنان محسوس نیست و صرفا پردازش سنگین‌تر شده.

همچنین، تعداد نورون‌ها و دیگر پارامترها طوری تنظیم شده‌اند تا از بیش برازش (overfitting) تا حد امکان جلوگیری شود. مدل های بیشتر (حدود 100 مدل برازش شده) و نمودارهای آنها در doc می‌توانید ببینید.

افزایش تعداد لایه‌ها

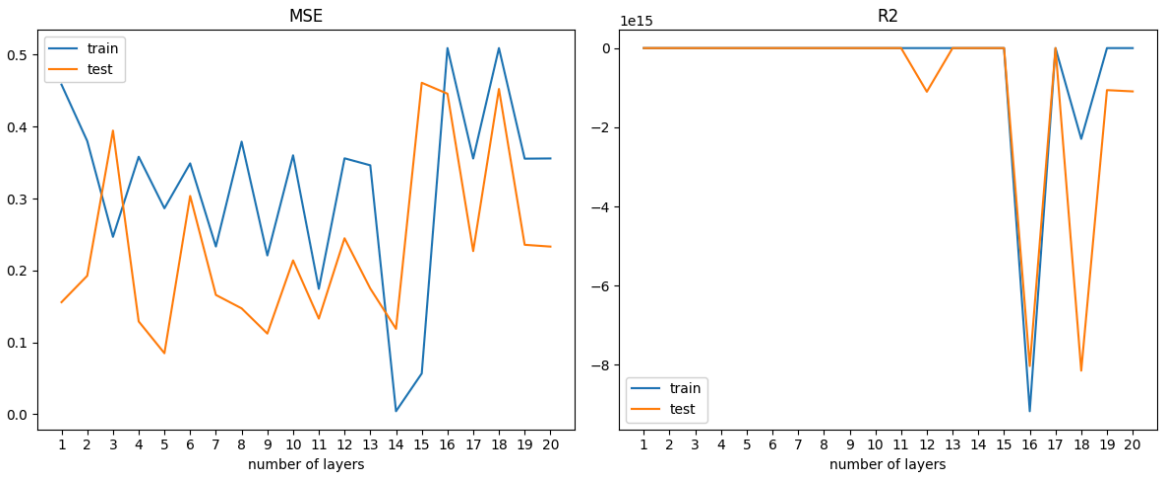
این حالت به شدت به هایپرپارامترها حساس بود، به طوری که تغییر هرکدام، باعث اختلاف نتیجه قابل ملاحظه‌ای می‌شد، اما عموما می‌توان نتایج را به صورت زیر خلاصه کرد:

1. افزایش لایه عموما باعث پیچیده‌تر شدن مدل میشود و امکان بیش برازش را برای داده‌های ساده‌تر فراهم می‌کند.
2. هنگامی که تعداد لایه بیش‌تر می‌شود باید تعداد داده‌ها یا تعداد دوره‌ها را نیز بیشتر کرد تا به شبکه یادگیری بهتری داشته باشد(در این‌باره در بخش 4.2 به طور کامل توضیح داده‌ایم.)
3. اگر `batch_size` زیاد باشد، به علت پیش‌بینی سخت تر مدل رگرشن (آن هم در دیتاست غیرخطی سینوسی)، شبکه نمی‌تواند به خوبی یادگیری را انجام دهد.

با توجه به تعابیر بالا ما با این مدل، برازش را انجام دادیم:

```
model.fit(x_train, y_train, epochs=10, verbose=False,
batch_size=10)
```

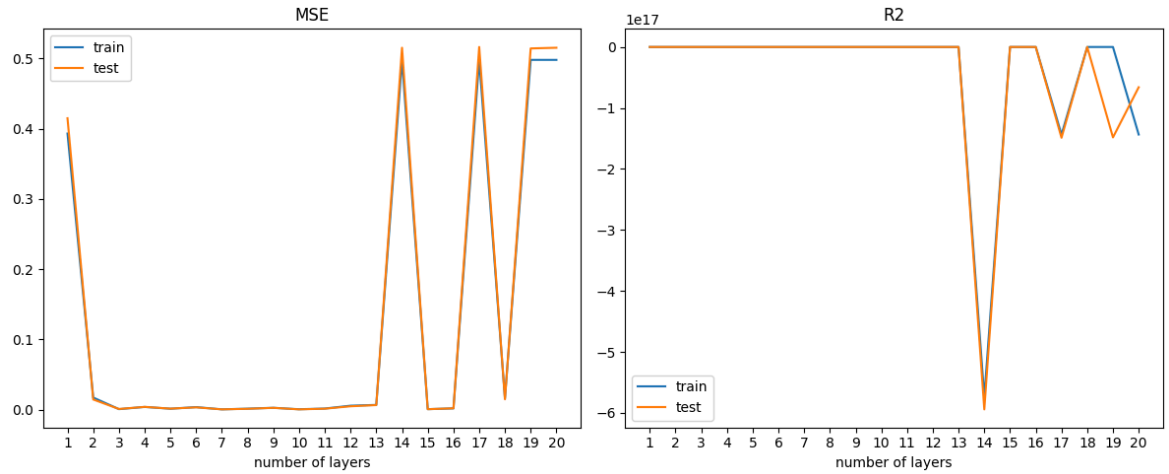
که می‌توان مشاهده کرد که تقریبا مدل چیزی نیاموخته است و همچنین بیش برازش (overfitting) اتفاق افتاده است.



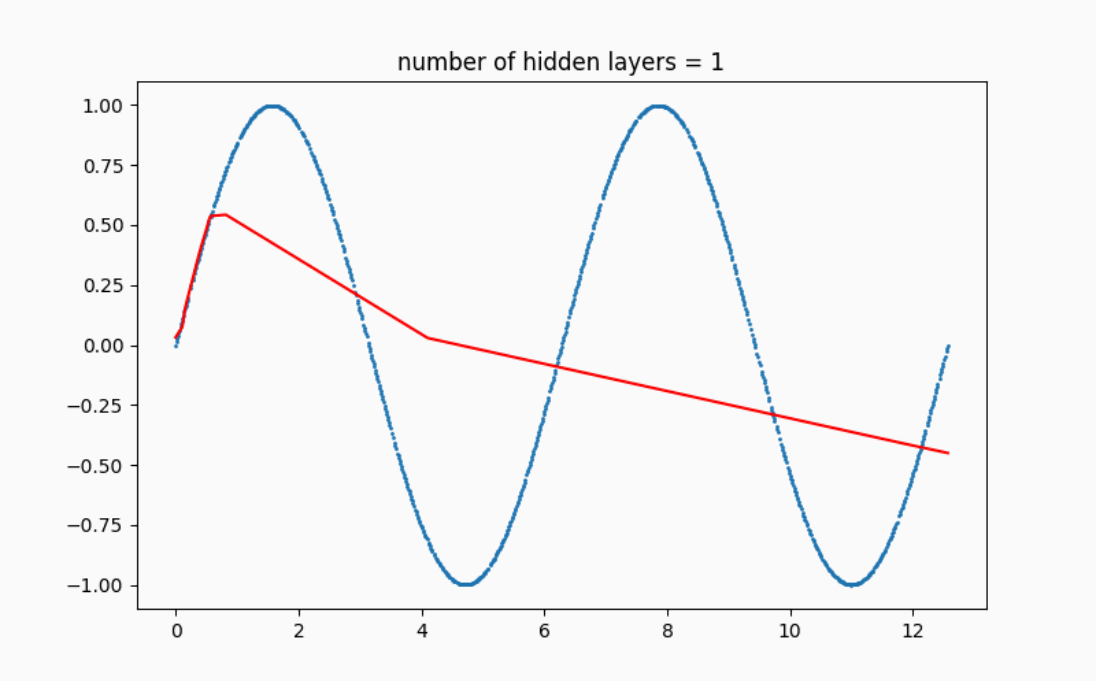
شکل 4-4. تغییر خطای مدل با تعداد مختلف لایه در حالت بیش برازش شده (overfitted)

در حالت بعدی تعداد دوره ها را بیشتر کردیم:

```
model.fit(x_train, y_train, epochs=200, verbose=False,
batch_size=16)
```

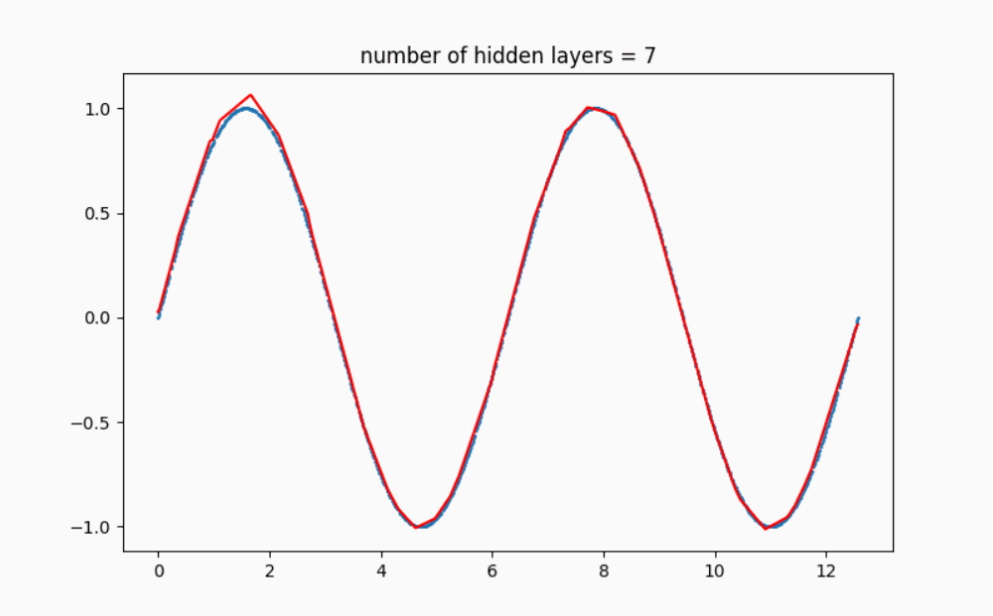


شکل 4-5. تغییر خطای مدل با تعداد مختلف لایه در حالت جلوگیری شده از بیش برازش (not overfitted)



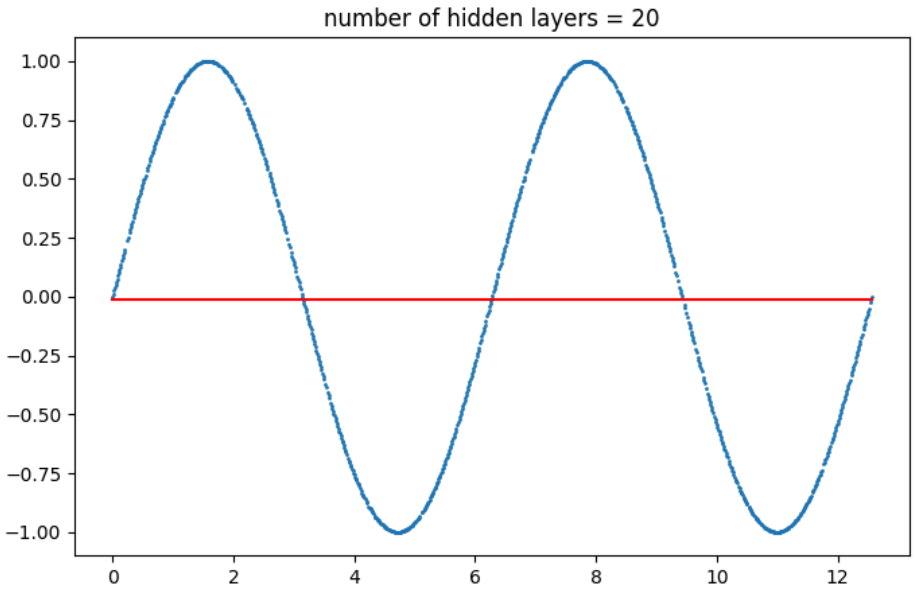
شکل 4-6. نتیجه مدل با داشتن یک لایه پنهان

در تعداد لایه پایین، یادگیری به خوبی رخ نمی‌دهد (مدل نمیتواند پیچیدگی سینوس را بیاموزد).



شکل 4-6. نتیجه مدل با داشتن 7 لایه پنهان

در تعداد لایه های مناسب (تقریبا 3 تا 12) مدل یادگیری مناسبی دارد.



شکل 4-7. نتیجه مدل با داشتن 20 لایه پنهان

در تعداد لایه بالا، به شدت واریانس زیادی در یادگیری شبکه می‌بینیم، به طوری که تعداد لایه 15 به بعد، مدل یادگیری مناسبی ندارد. دلیل این اتفاق Vanishing Gradient هست.

استفاده از Grid Search برای پیدا کردن تعداد لایه‌های بهینه

این جستجو، برای پیدا کردن هایپرپارامترهای بهینه برای شبکه عصبی استفاده می‌شود. به طوری که تمام حالت های داده شده را امتحان می‌کند و نتایج را نشان می‌دهد.

محدودیت‌های استفاده:

1. فضای جستجو بزرگ: اگر تعداد هایپرپارامترها زیاد باشد، چون تمامی حالات درنظر گرفته می‌شود(به طور ضربی افزایش پیدا می‌کند) فضای جستجو بسیار بزرگ شده و زمانبر خواهد بود.
2. فضای نمونه بزرگ: اگر شبکه بزرگ باشد یا تعداد نمونه‌ها زیاد باشد، برازش شبکه طول خواهد کشید.
3. هایپرپارامترهای پیوسته: این جستجو در فضای پیوسته نمی‌تواند کار کند، در نتیجه مجبور هستیم، متغیرهای پیوسته را گسسته کنیم و یا نمونه برداری کنیم که شاید باعث از بین رفتن مقدار بهینه شود.
4. وابستگی بین 2 پارامتر: اگر این حالت وجود داشته باشد، باید مطمئن شویم هر دوی آنها در فضای جستجو باشند و الا ممکن است به مقدار بهینه نرسیم. همچنین اگر متغیری مستقیما به متغیر دیگری وابسته باشد، ممکن نیست هردوی آن‌ها را در فضای جستجو قرار دهیم.

برای این سرچ از این مدل استفاده کرده‌ایم:

```
n_hidden_layers = range(1, 21)
model = KerasRegressor(model=create_model, verbose=0,
n_hidden_layers=n_hidden_layers, epochs=200, batch_size=16)
```

که نتیجه زیر حاصل شده است:

جدول 4-1. نتایج Grid Search برای تعداد لایه های مدل

	layers	mean_test_score
0	13	0.997435
1	14	0.991093
2	6	0.988971
3	9	0.987354
4	7	0.977423
5	3	0.968165
6	17	0.891060
7	8	0.889439
8	5	0.850572
9	11	0.847244
10	4	0.823633
11	10	0.797928
12	19	0.797416
13	2	0.740373
14	12	0.691777
15	16	0.658742
16	20	0.467164
17	1	0.308564
18	18	0.278578
19	15	0.174975

می‌بینیم که پایین‌ترین دقت‌ها برای تعداد لایه‌های زیاد(بیش برازش) و اندک(کم فرازش) می‌باشد. همچنین بهترین مدل، به دقت 99.9 درصدی برای داده تست رسیده است که در نوع خود جالب است!

2-4. طبقه‌بندی (classification)

برای طبقه‌بندی از داده‌های MNIST استفاده شده که پیش‌تر توضیح داده شده‌است.

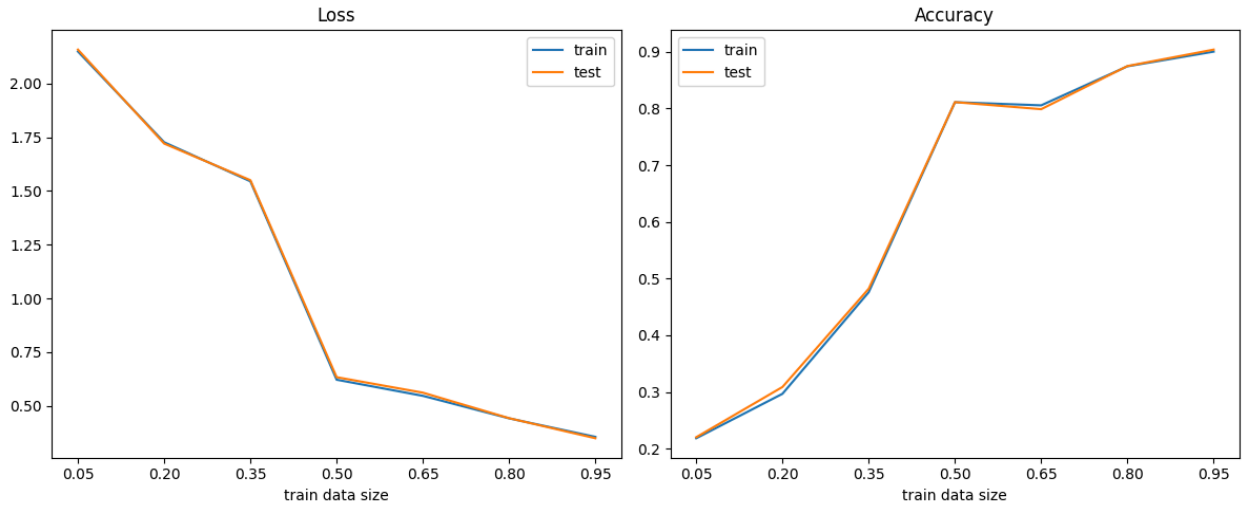
شبکه استفاده شده:

```
def create_model_classification(n_hidden_layers):
    model = Sequential()
    model.add(Dense(16, activation="relu", input_shape=(784,)))
    for _ in range(n_hidden_layers):
        model.add(Dense(10, activation="relu"))
    model.add(Dense(10, activation="softmax"))

    model.compile(
        optimizer="adam",
        loss="categorical_crossentropy",
        metrics = ["accuracy"]
    )
    return model
```

بررسی مدل‌ها

افزایش داده‌ها

همانطور که مشاهده می‌شود، افزایش داده ها باعث بهبود عملکرد شبکه می‌شود و با همچنین با انتخاب درست تعداد نورون‌ها و توابع فعالساز و همچنین `epoch` و `batch_size` مناسب از بیش برازش (overfitting) جلوگیری کرده‌ایم.



شکل 4-8. تغییر خطای مدل با نسبت‌های مختلف داده برای آزمایش و آزمون

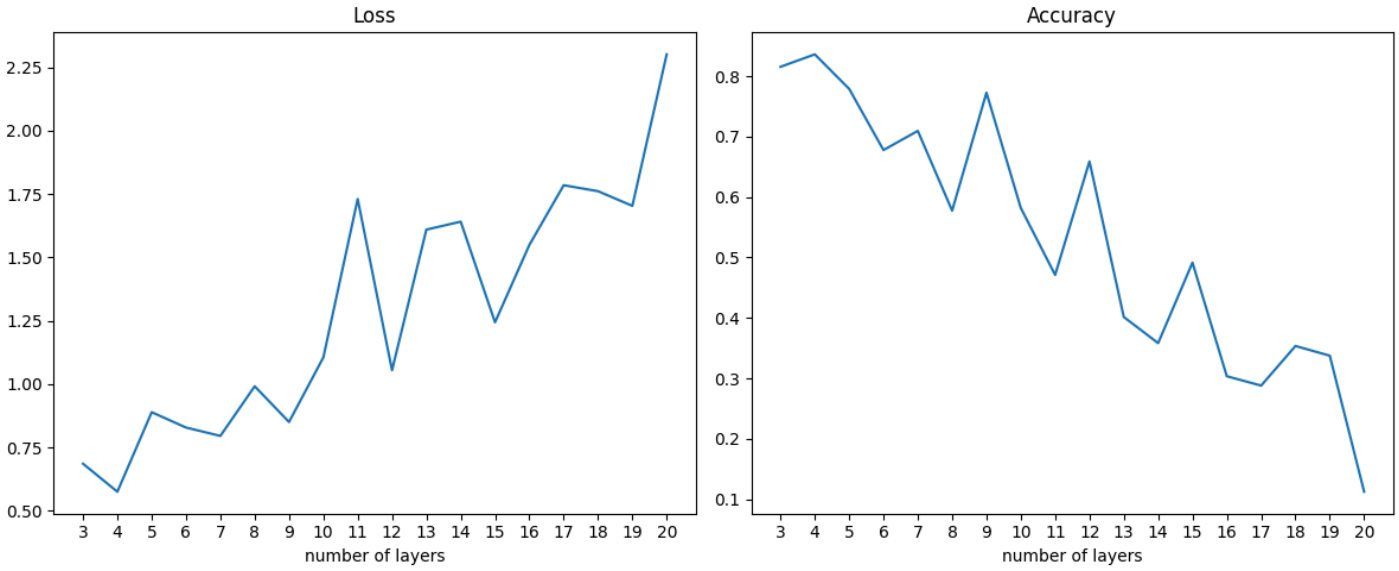
افزایش لایه‌ها

برای افزایش لایه‌ها، ما 2 حالت را بررسی کرده‌ایم. در حالت اول تعداد دوره‌ها ثابت ماندند و در حالت دوم با افزایش لایه‌ها تعداد دوره‌ها را نیز افزایش دادیم.

1. بدون افزایش تعداد دوره

```
model.fit(x, y, batch_size=4000, epochs=10, verbose = False)
```

چون که تعداد لایه‌ها در حال افزایش می‌باشد و تعداد دوره ها ثابت است، شبکه به علت لایه‌های زیاد پیچیده شده است و نورون‌ها فرصت مناسبی برای یادگیری داده‌ها ندارند. همچنین چون در شبکه تصحیح وزن‌ها با روش adam انجام می‌شود(که خود مبتنی بر Gradient descent می‌باشد) مشکل Vanishing Gradient رخ می‌دهد. به این معنی که مشتق هایی که با backpropagation به لایه‌های ابتدایی رسیده اند به صفر میل کرده‌اند و وزن‌های نورون‌های لایه های ابتدایی تغییر چندانی نمی‌کنند.



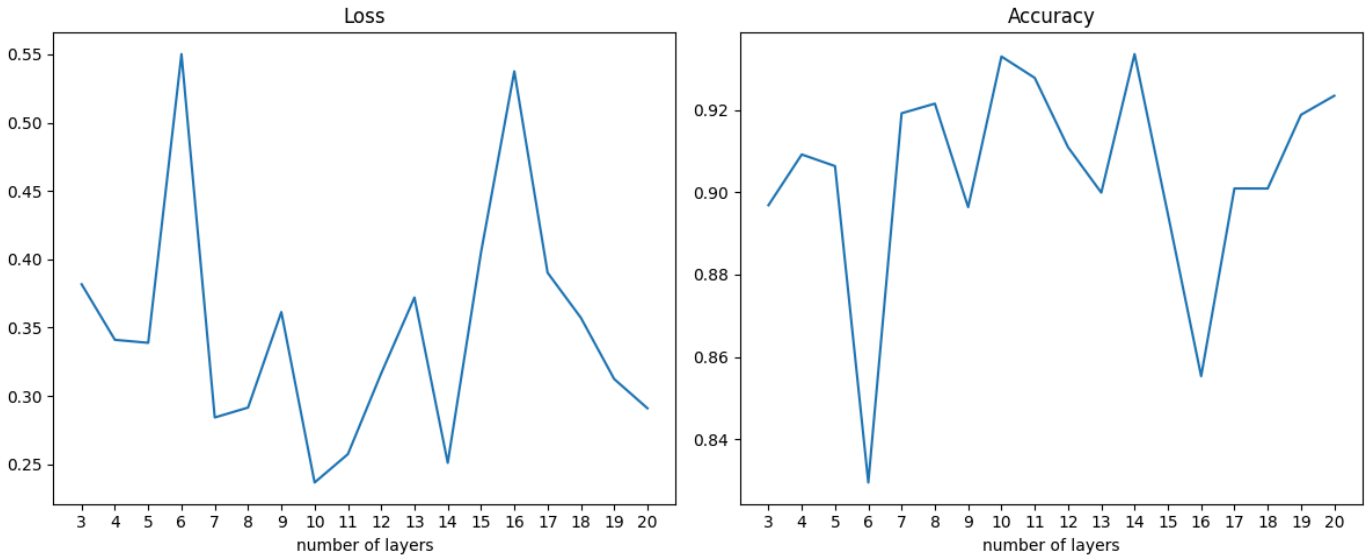
شکل 4-9. تاثیر تعداد لایه‌ها بر روی دقت مدل بدون افزایش دوره‌های تمرین

2. با افزایش تعداد دوره

```
model = create_model_classification(n_hidden_layers = i)
model.fit(x, y, batch_size=4000, epochs=5 * i, verbose = False)
```

با افزایش تعداد دوره‌ها (`epochs = 5 * i`)، شبکه پیچیده شده برای دفعات بیشتری داده‌ها را یاد می‌گیرد و با توجه به لایه‌های بیشتری که دارد، توانایی یادگیری مدل‌های پیچیده‌تری را کسب می‌کند. البته داده‌های مورد استفاده، یعنی دیتاست MNIST، داده‌های نسبتا ساده‌ای می‌باشند که پیچیده کردن مدل ممکن است منجر به بیش برازش شود.

همانطور که مشاهده می‌کنید، دقت مدل را نسبتا در بازه 84 تا 92 درصد نگه داشته‌ایم.



شکل 4-10. تاثیر افزایش دوره‌های تمرین همزمان با افزایش تعداد لایه‌ها بر روی دقت مدل

Grid Search

برای پیدا کردن شبکه بهینه، ابتدا به کمک معماری قبلی و استفاده از Grid Search سعی بر پیدا کردن مدل بهینه ای کردیم، که به دقت 97.3 درصدی دست یافتیم.

متغیر های مورد استفاده:

```
units = [32, 64, 128]
param_grid = {
    "units": units,
    "batch_size": [128, 256],
    "epochs": [10, 20]
}
```

جدول 4-2. نتایج Grid Search برای پیدا کردن پارامترهای مدل بهینه

	params	mean_test_score
0	{"batch_size": 128, "epochs": 20, "units": 128}	0.973107
1	{"batch_size": 256, "epochs": 20, "units": 128}	0.971696
2	{"batch_size": 128, "epochs": 10, "units": 128}	0.970446
3	{"batch_size": 128, "epochs": 20, "units": 64}	0.969304
4	{"batch_size": 256, "epochs": 20, "units": 64}	0.967304
5	{"batch_size": 256, "epochs": 10, "units": 128}	0.967071
6	{"batch_size": 128, "epochs": 10, "units": 64}	0.964214
7	{"batch_size": 128, "epochs": 20, "units": 32}	0.960607
8	{"batch_size": 256, "epochs": 10, "units": 64}	0.959089
9	{"batch_size": 128, "epochs": 10, "units": 32}	0.954518
10	{"batch_size": 256, "epochs": 20, "units": 32}	0.954107
11	{"batch_size": 256, "epochs": 10, "units": 32}	0.948750

توسعه شبکه بهینه چند لایه با کمک لایه Dropout

برای پیدا کردن مدل‌های بهینه، چند مدل با استفاده از لایه های متنوع تری(بدون استفاده از لایه‌های پیچشی) را آزمودیم. مدل پیش رو با بهره‌گیری از این لینک، توسعه داده شده است.

```
model = Sequential()
model.add(Dense(256, activation="relu", input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(64, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(32, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(10, activation="softmax"))
model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
```

در این مدل، از لایه‌های dropout استفاده شده است تا نورون‌ها با استقلال بیش‌تری به یادگیری به پردازند و فیچرهای بیشتری استخراج کنند، در نتیجه درصد دقت به بالای 98.1 می‌رسد.

```
test accuracy: 0.9816428422927856
```