



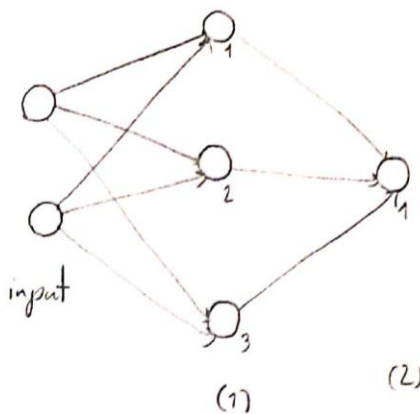
گزارش تمرین اول هوش محاسباتی

نام تهیه کننده: ملیکا نوبختیان

شماره دانشجویی: ۹۷۵۲۲۰۹۴

نسخه: ۱

۱- سوال اول



$$X = \begin{bmatrix} 1 & 2 \\ -1 & 1 \\ -2 & 1 \\ -4 & 0 \end{bmatrix} \begin{matrix} 0 \rightarrow (0) \\ \square \rightarrow (1) \\ \square \rightarrow (1) \\ 0 \rightarrow (0) \end{matrix}$$

(input points)

۴ × ۲
۴ عدد عددی

استاد وزن های اولیه ای ادل و ددم را به صورت random بدست می آوریم:

$$w_1^{(1)} = \begin{bmatrix} 0.1 & 0.8 \end{bmatrix}$$

$$w_2^{(1)} = \begin{bmatrix} 0.3 & 0.2 \end{bmatrix}$$

$$w_3^{(1)} = \begin{bmatrix} -0.5 & 0.5 \end{bmatrix}$$

$$w^{(1)} = \begin{bmatrix} 0.1 & 0.8 \\ 0.3 & 0.2 \\ -0.5 & 0.5 \end{bmatrix}$$

$$b^{(1)} = \begin{bmatrix} 0 & 0.9 & -0.4 \end{bmatrix}$$

۳ × ۲
۳ عدد عددی برای هر ورودی
۴ عدد عددی به صورت وزن

$$w_1^{(2)} = \begin{bmatrix} 0.6 & 0.7 & -0.3 \end{bmatrix}$$

۱ × ۳

$$b^{(2)} = \begin{bmatrix} 0.1 \end{bmatrix}$$

۱ × ۱

forward-pass

$$z^{(1)} = X \cdot w^{(1).T} + b^{(1)} = \begin{bmatrix} 1 & 2 \\ -1 & 1 \\ -2 & 1 \\ -4 & 0 \end{bmatrix} \begin{bmatrix} 0.1 & 0.3 & -0.5 \\ 0.8 & 0.2 & 0.5 \end{bmatrix} + \begin{bmatrix} 0 & 0.9 & -0.4 \end{bmatrix}$$

۴ × ۲ ۲ × ۳

$$= \begin{bmatrix} 1.7 & 0.7 & 0.5 \\ 0.7 & -0.1 & 1 \\ 0.6 & -0.4 & 1.5 \\ -0.4 & -1.2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0.9 & -0.4 \end{bmatrix} = \begin{bmatrix} 1.7 & 1.6 & 0.1 \\ 0.7 & 0.8 & 0.6 \\ 0.6 & 0.5 & 1.1 \\ -0.4 & -0.3 & 1.6 \end{bmatrix}$$

$$a^{(1)} = \text{ReLU}(z^{(1)}) = \begin{bmatrix} 1.7 & 1.6 & 0.1 \\ 0.7 & 0.8 & 0.6 \\ 0.6 & 0.5 & 1.1 \\ 0 & 0 & 1.6 \end{bmatrix}$$

$\hookrightarrow \max(0, z)$

$$z^{(2)} = a^{(1)} \cdot w^{(2).T} + b^{(2)} = \begin{bmatrix} 1.7 & 1.6 & 0.1 \\ 0.7 & 0.8 & 0.6 \\ 0.6 & 0.5 & 1.1 \\ 0 & 0 & 1.6 \end{bmatrix}_{4 \times 3} \begin{bmatrix} 0.6 \\ 0.7 \\ -0.3 \end{bmatrix}_{3 \times 1} + [0.1]$$

$$= \begin{bmatrix} 2.11 \\ 0.8 \\ 0.38 \\ -0.48 \end{bmatrix} + [0.1] = \begin{bmatrix} 3.11 \\ 0.9 \\ 0.48 \\ -0.38 \end{bmatrix}$$

$$a^{(2)} = \text{sigmoid}(z^{(2)}) = \begin{bmatrix} 0.95 \\ 0.71 \\ 0.61 \\ 0.40 \end{bmatrix}$$

\downarrow
 $\frac{1}{1 + e^{-x}}$

$$d^{(2)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

ماتریس هدف

backward-pass

$$\Delta w_{21} = \delta^{(2).T} \cdot a^{(1)}$$

$$\delta^{(2)} = (d^{(2)} - a^{(2)}) \cdot g'(z^{(2)})$$

سشن
 تابع فعال سازی

$$d^{(2)} - a^{(2)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.95 \\ 0.71 \\ 0.61 \\ 0.40 \end{bmatrix} = \begin{bmatrix} -0.95 \\ 0.29 \\ 0.39 \\ -0.4 \end{bmatrix}$$

در لایه output سشنی داریم:
 سشن تابع فعال سازی سلولها به این صورت است:
 $\text{sigmoid}(z) = \text{sigmoid}(z) (1 - \text{sigmoid}(z))$

$$g'(z^{(2)}) = a^{(2)} (1 - a^{(2)}) = \begin{bmatrix} 0.0475 \\ 0.2059 \\ 0.2379 \\ 0.24 \end{bmatrix}$$

$$\Delta w_{21} = \begin{bmatrix} -0.045 \\ 0.059 \\ 0.092 \\ -0.096 \end{bmatrix}_{4 \times 1}^T \begin{bmatrix} 1.7 & 1.6 & 0.1 \\ 0.7 & 0.8 & 0.6 \\ 0.6 & 0.5 & 1.1 \\ 0 & 0 & 1.6 \end{bmatrix}_{4 \times 3} =$$

$$\delta^{(2)} = \begin{bmatrix} -0.045 \\ 0.059 \\ 0.092 \\ -0.096 \end{bmatrix}$$

$$\begin{bmatrix} 0.02 & 0.0212 & -0.0215 \end{bmatrix}$$

$$\Delta w_{01} = \delta^{(1)} \cdot x \quad \delta^{(1)} = (\delta^{(2)} \cdot w^{(2)}) \cdot f'(z^{(1)})$$

$$\delta^{(2)} \cdot w^{(2)}: \begin{bmatrix} -0.045 \\ 0.059 \\ 0.092 \\ -0.096 \end{bmatrix}_{4 \times 1} \begin{bmatrix} 0.6 & 0.7 & -0.3 \end{bmatrix}_{1 \times 3} = \begin{bmatrix} -0.027 & -0.0315 & 0.0135 \\ 0.0354 & 0.0413 & -0.0177 \\ 0.0552 & 0.0644 & -0.0276 \\ -0.0576 & -0.0672 & 0.0288 \end{bmatrix}_{4 \times 3}$$

$$\text{Relu}'(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \quad f'(z^{(1)}) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\delta^{(1)} = \begin{bmatrix} -0.027 & -0.0315 & 0.0135 \\ 0.0354 & 0.0413 & -0.0177 \\ 0.0552 & 0.0644 & -0.0276 \\ 0 & 0 & 0.0288 \end{bmatrix}$$

$$\Delta w_{01} = \delta^{(1)} \cdot T = \begin{bmatrix} 1 & 2 \\ -1 & 1 \\ -2 & 1 \\ -4 & 0 \end{bmatrix}_{4 \times 2} = \begin{bmatrix} -0.172 & 0.036 \\ -0.183 & 0.078 \\ -0.028 & -0.018 \end{bmatrix}_{4 \times 2}$$

$$\mu = 1$$

$$w_{(2)} = w_{(2)} - \mu \Delta w_{12} = \begin{bmatrix} 0.6 & 0.7 & -0.3 \end{bmatrix} - \begin{bmatrix} 0.02 & 0.0212 & -0.0215 \end{bmatrix} \\ = \begin{bmatrix} 0.58 & 0.6788 & 0.2785 \end{bmatrix}$$

$$w_{(1)} = w_{(1)} - \mu \Delta w_{01} = \begin{bmatrix} 0.1 & 0.8 \\ 0.3 & 0.2 \\ -0.5 & 0.5 \end{bmatrix} - \Delta w_{01} = \begin{bmatrix} 0.272 & 0.764 \\ 0.483 & 0.122 \\ -0.472 & 0.518 \end{bmatrix}$$

$$b_{(2)} = b_{(2)} - \mu \text{sum}(\delta^{(2)}) = \begin{bmatrix} 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix}$$

$$b_{(1)} = b_{(1)} - \mu \text{sum}(\delta^{(1)}) = \begin{bmatrix} 0 & 0 & 0 & -0.4 \end{bmatrix} - \begin{bmatrix} 0.0636 & 0.0742 & -0.003 \end{bmatrix} \\ = \begin{bmatrix} -0.0636 & 0.8258 & -0.397 \end{bmatrix}$$

این سه مرنا باید Adaline تابعی می ست زیرا صرفاً یک خطای تان نقاط ددلاس را از هم جدا کرد
دیده است حراتی ه Adaline و Adaline داشته باشیم که باید خطای تان نقاط این ددلاس را
از هم جدا کنیم.

۲- سوال دوم

برای پیاده‌سازی این قسمت از لینک زیر کمک گرفتیم:

<https://zerowithdot.com/mlp-backpropagation/>

برای پیاده‌سازی یک MLP اولین گام این است که لایه‌های آن را به خوبی بسازیم. برای این کار من ابتدا کلاس Single_Layer را ساختم که در آن صرفاً یک لایه تنها را می‌سازیم. ابتدا به constructor این کلاس و مواردی باید برای ساختن یک لایه به آن بدهیم می‌پردازیم:

```
class Single_Layer:
    def __init__(
        self,
        number_of_units,
        activation_function,
        input_size = None
    ):
        self.input_size = input_size
        self.number_of_units = number_of_units
        self.W = None
        self.b = None
        self.A = None
        self.activation_func , self.dactivation = self.select_activation_function(activation_function)
```

برای ساختن یک لایه باید بدانیم چند نورون در آن وجود دارد، number_of_units این متغیر را مشخص می‌کند. هم چنین باید بدانیم چه تابعی برای فعال‌سازی استفاده می‌شود، activation_function نمایانگر این بخش است. هم چنین اگر لایه‌ای که می‌سازیم لایه ورودی باشد، لازم است ابعاد داده ورودی را هم نشان دهیم، input_size برای این منظور است. یک لایه W و b و A نیز خواهد داشت ولی مقادیر این قسمت وقتی شبکه MLP ساخته می‌شود مشخص می‌شوند. حالا به توضیح توابع مربوط به فعال‌سازی می‌رویم:

```
def select_activation_function(self, func):
    if func == 'sigmoid':
        act = self.sigmoid_activation
        dact = self.sigmoid_derivitive
    elif func == 'relu':
        act = self.ReLU_activation
        dact = self.ReLU_derivitive

    return act , dact

def sigmoid_activation(self, Z):
    return 1 / (1 + np.exp(-Z))

def ReLU_activation(self, Z):
    return np.maximum(Z, 0)

def sigmoid_derivitive(self, A):
    return A * (1 - A)

def ReLU_derivitive(self, A):
    A[A > 0] = 1
    A[A < 0] = 0
    return A
```

توابع فعال‌سازی موجود sigmoid و ReLU هستند. در این تابع ما خود تابع فعال‌سازی و مشتق آن را مشخص می‌کنیم. مشتق sigmoid به صورت $g(z)*(1 - g(z))$ است. در مورد ReLU مشتق مقادیر بزرگ‌تر از ۰ برابر یک و برای بقیه مقادیر صفر است. برای initialize کردن وزن‌های یک لایه به صورت زیر عمل می‌کنیم:

```
def initialize_weights(self):
    self.W = np.random.rand(self.number_of_units, self.input_size)
    self.b = np.random.rand(1, self.number_of_units)
```

W یک ماتریس با تعداد ردیف به اندازه تعداد نورون‌های موجود در آن لایه و تعداد ستون به اندازه ورودی آن است. b نیز یک ماتریس سطری با عناصر به اندازه تعداد نورون‌های لایه است. عملیات forward pass نیز با تابع زیر انجام می‌شود:

```
def forward_pass(self, X):
    Z = np.dot(X, self.W.T) + self.b
    A = self.activation_func(Z)
    self.A = A
    return A
```

ورودی این تابع ماتریس X است که می‌تواند ورودی اصلی شبکه یا مقادیر فعال‌سازی لایه قبل باشد. ماتریس X در ماتریس وزن‌ها ضرب داخلی می‌شود و سپس به آن مقادیر b اضافه می‌شود. برای این کار باید ترانهاده W در X ضرب شود در این صورت ابعاد W به صورت (input_size, units) می‌شود تا با ابعاد X که به صورت (example_count, input_size) است قابل ضرب شود. سپس تابع فعال‌سازی به Z اعمال می‌شود و مقدار A برای لایه ست می‌شود. در آخر به سراغ back propagation می‌رویم. می‌دانیم به طور کلی مقدار دلتا برای یک لایه به جز لایه آخر به صورت زیر است:

$$\phi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj}$$

که با توجه به شکل برابر با ضرب داخلی مقادیر دلتا لایه بعدی در مقادیر وزن موجود بین لایه موردنظر و لایه بعدی ضرب در مشتق تابع فعال‌سازی لایه فعلی است. در این صورت تابع back propagation برای یک لایه به صورت زیر خواهد بود:

```
def back_propagation(self, delta_next, w_next):
    dactivation = self.dactivation(self.A)
    delta = np.dot(delta_next, w_next) * dactivation
    return delta
```

ورودی‌های این تابع مقادیر دلتای لایه بعدی و مقادیر وزن آن است. پس از انجام عملیاتی که گفته شد ما به مقادیر دلتای لایه موردنظر خواهیم رسید.

پس از توضیح کلاس single_layer حالا به توضیح کلاس MLP می‌پردازیم:

```
class MLP:
    def __init__(
        self,
        layers,
        learning_rate
    ):
        input_size = layers[0].number_of_units
        layers[0].initialize_weights()
        for layer in layers[1:]:
            layer.input_size = input_size
            input_size = layer.number_of_units
            layer.initialize_weights()
        self.layers = layers
        self.lr = learning_rate
```

برای اینکه بتوانیم یک شبکه MLP بسازیم نیاز به لیستی از لایه‌ها که از نوع کلاس `single_layer` هستند و `learning_rate` نیازمندیم. در `constructor` شبکه مقادیر وزن‌ها `init` خواهند شد. در اینجا `input_size` تمام لایه‌ها به جز لایه اول مشخص خواهد شد که تعداد نوروهای لایه قبل از آن است. عملیات `forward pass` در شبکه با تابع زیر محاسبه می‌شود:

```
def forward_pass(self, x):
    output = self.layers[0].forward_pass(x)
    for layer in self.layers[1:]:
        output = layer.forward_pass(output)
    return output
```

برای این قسمت کافی است تابع `forward pass` هر لایه را صدا بزنیم و خروجی آن را به عنوان ورودی به لایه بعد بدهیم تا به خروجی برسیم. تابع `loss` ما در اینجا `MSE` خواهد بود که تابع زیر با گرفتن مقادیر پیش‌بینی شده و مقادیر واقعی آن را به دست می‌آورد:

```
def MSE_loss(self, y_pred, y_true):
    return np.sum(1/2*((y_true - y_pred)**2))
```

برای اینکه بتوانیم بفهمیم با توجه به پیش‌بینی ما هر نمونه متعلق به کدام کلاس است باید مقادیر پیش‌بینی شده را به صفر یا یک نگاشت کنیم. برای این کار یک `threshold` برابر 0.5 در نظر گرفتیم که در صورتی که پیش‌بینی بزرگ‌تر از این باشد متعلق به کلاس ۱ و در صورتی که کمتر باشد متعلق به کلاس ۰ است:

```
def predict_label(self, y_pred):
    y_pred[y_pred >= 0.5] = 1
    y_pred[y_pred < 0.5] = 0
    return y_pred
```


برای به دست آوردن مقدار accuracy باید بدانیم که نسبت به تعداد کل نمونه‌ها label چه تعداد از آن‌ها درست محاسبه شده‌است. تابع زیر مقدار accuracy را محاسبه می‌کند:

```
def Accuracy(self, y_pred, y_true):
    all_examples = y_true.shape[0]
    y_pred = self.predict_label(y_pred)
    true_predicted = np.sum(y_true == y_pred)

    return true_predicted / all_examples
```

در نهایت به توضیح فرآیند backpropagation در یک شبکه MLP می‌پردازیم:

```
def backward_pass(self, y_pred, y_true, X):
    delta_w = [None]*len(self.layers)
    delta_b = [None]*len(self.layers)
    last_layer = self.layers[-1]
    delta = ( y_true - y_pred ) * last_layer.dactivation(last_layer.A)
    for i in range(len(self.layers) - 1, 0, -1) :
        prev_A = self.layers[i - 1].A
        delta_w[i] = np.dot(delta.T, prev_A)
        delta_b[i] = np.sum(delta, axis=0, keepdims=True)

        delta = self.layers[i - 1].back_propagation(delta, self.layers[i].W)

    delta_w[0] = np.dot(delta.T, X)
    delta_b[0] = np.sum(delta, axis=0, keepdims=True)

    for i in range(len(self.layers)):
        self.layers[i].W += self.lr * delta_w[i]
        self.layers[i].b += self.lr * delta_b[i]
```

ورودی این تابع نمونه‌های ورودی X ، مقادیر پیش‌بینی شده y_{pred} و مقادیر اصلی label ها یعنی y_{true} هستند. ابتدا برای مقادیر تغییر وزن و b دو لیست می‌سازیم که طولی به اندازه تعداد لایه‌های شبکه دارند. حالا باید مقدار دلتا برای لایه آخر را حساب کنیم. برخلاف لایه‌های دیگر مقدار دلتا برای این لایه به صورت زیر محاسبه می‌شود:

$$\phi'(v_j)(d_j - y_j)$$

این مقدار برابر تفاوت مقادیر پیش‌بینی شده و اصلی ضرب در مشتق تابع فعال‌سازی لایه آخر است. پس از این کار فرآیند کلی کار را شروع می‌کنیم. برای به دست آوردن مقدار تغییر W لازم است مقدار دلتا برای آن لایه را در مقادیر فعال‌سازی لایه قبلی ضرب داخلی کنیم. برای به دست آوردن مقدار تغییر b هم فقط لازم است مقادیر موجود در دلتا را به صورت ستونی جمع کنیم (چون چند نمونه داریم). سپس با توجه به آنچه گفته شد مقدار دلتا برای لایه قبل را به دست می‌آوریم. در نهایت مقادیر تغییر در learning rate ضرب شده و به مقادیر W و b اضافه خواهند شد.

برای ساختن شبکه mlp از تابع زیر استفاده می‌کنیم. این تعداد با گرفتن اندازه ورودی، لیستی از تعداد unit های موجود در هر لایه، لیستی از توابع فعال‌سازی لایه‌ها و learning rate یک شبکه mlp می‌سازد. شبکه مورد نظر من یک لایه hidden با ۴ نورون دارد و learning rate آن نیز 0.01 است:

```
def build_MLP(
    input_size,
    units_in_each_layer,
    activation_for_each_layer,
    learning_rate
):
    layers = []
    first_layer = Single_Layer(units_in_each_layer[0], activation_for_each_layer[0], input_size)
    layers.append(first_layer)
    for units, activation in zip(units_in_each_layer[1:], activation_for_each_layer[1:]):
        layer = Single_Layer(units, activation)
        layers.append(layer)

    mlp = MLP(layers, learning_rate)
    return mlp
```

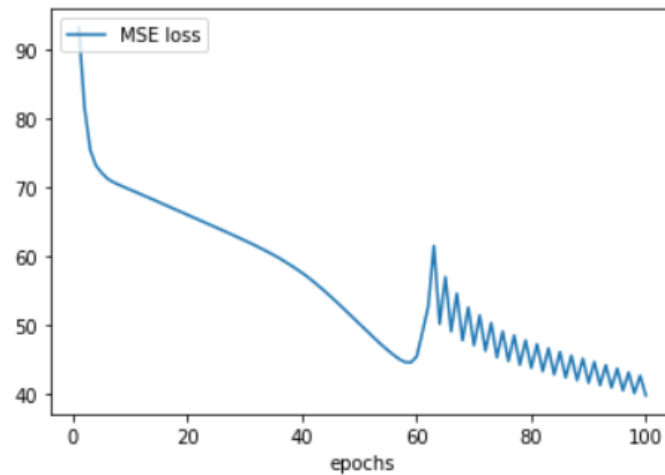
من برای آموزش شبکه خودم 100 epoch در نظر گرفتم و مقادیر loss و accuracy را در هر epoch به دست آورده و چاپ کردم. هم چنین این مقادیر برای کشیدن نمودار ذخیره می‌شوند:

```
loss = []
accuracy = []
epoch_idx = []
epochs = 100
for i in range(epochs):
    y_pred = model.forward_pass(X)
    model.backward_pass(y_pred, y_true, X)
    loss_epoch = model.MSE_loss(y_pred, y_true)
    accuracy_epoch = model.Accuracy(y_pred, y_true)
    print(f'epoch {i+1}/{epochs} ====> Loss:{loss_epoch}    Accuracy:{accuracy_epoch}')
    loss.append(loss_epoch)
    accuracy.append(accuracy_epoch)
    epoch_idx.append(i+1)
```

```
epoch 43/100 ====> Loss:45.391221537422204    Accuracy:0.8506944444444444
epoch 44/100 ====> Loss:45.320896587452864    Accuracy:0.84375
```

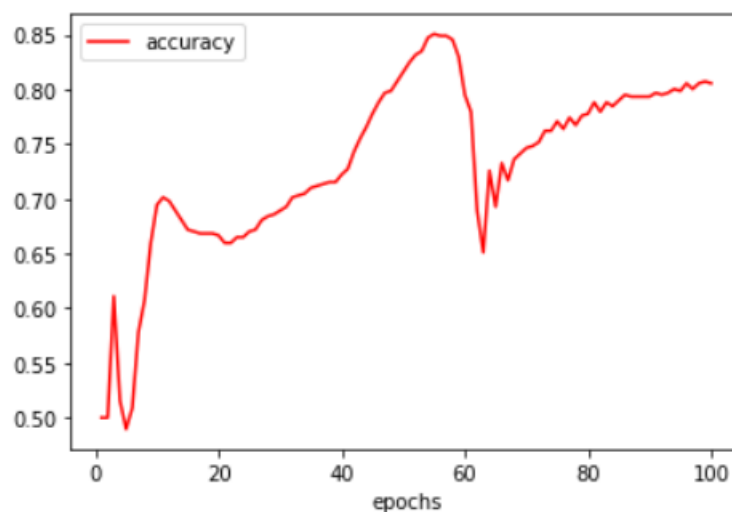
می‌توانیم نمودار loss را به صورت زیر رسم کنیم:

```
# plot loss
plt.plot(epoch_idx, loss, label='MSE loss')
plt.xlabel("epochs")
plt.legend(loc='upper left')
plt.show()
```



و accuracy نیز به صورت زیر رسم می‌شود:

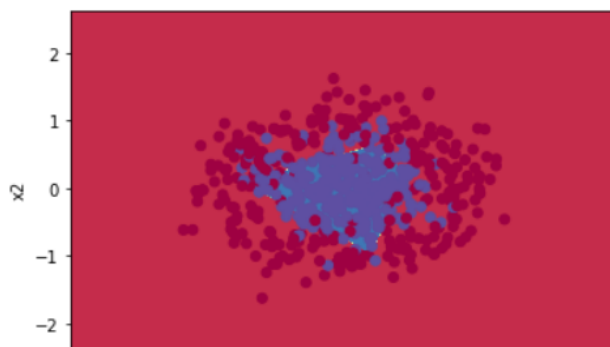
```
# Q2_graded
# plot accuracy
plt.plot(epoch_idx, accuracy, '-r', label='accuracy')
plt.xlabel("epochs")
plt.legend(loc='upper left')
plt.show()
```



برای رسم decision boundary از تابعی که در نوتبوک Decision_Boundary بود استفاده کردم و تنها کمی تغییرات در آن دادم:

```
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = model.forward_pass(np.c_[xx.ravel(), yy.ravel()])
    Z = model.predict_label(Z)
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=plt.cm.Spectral)
```

```
plot_decision_boundary(model, X.T, y_true.T)
```



برای اینکه تاثیر تعداد نورون‌های زیاد در لایه مخفی را بر روی فرآیند training ببینم ، یک شبکه با یک لایه hidden و ۳۰ نورون در نظر گرفتم و داده‌ها را روی آن آموزش دادم. قسمتی از نتایج این فرآیند در تصویر زیر قابل مشاهده است:

epoch 1/30 ==>	Loss:132.19308893422647	Accuracy:0.5
epoch 2/30 ==>	Loss:128.16643831480084	Accuracy:0.5
epoch 3/30 ==>	Loss:123.82014657607513	Accuracy:0.5
epoch 4/30 ==>	Loss:119.61480089462748	Accuracy:0.5
epoch 5/30 ==>	Loss:116.35083292219304	Accuracy:0.5
epoch 6/30 ==>	Loss:113.79520162584296	Accuracy:0.5833333333333334
epoch 7/30 ==>	Loss:111.83877493409277	Accuracy:0.6024305555555556
epoch 8/30 ==>	Loss:110.34974666474238	Accuracy:0.6041666666666666
epoch 9/30 ==>	Loss:109.20406150502492	Accuracy:0.6059027777777778
epoch 10/30 ==>	Loss:108.29074011041737	Accuracy:0.6041666666666666
epoch 11/30 ==>	Loss:107.5460416853461	Accuracy:0.5972222222222222
epoch 12/30 ==>	Loss:106.93161857001621	Accuracy:0.5989583333333334
epoch 13/30 ==>	Loss:106.40269265191051	Accuracy:0.59375
epoch 14/30 ==>	Loss:105.92985303316337	Accuracy:0.59375
epoch 15/30 ==>	Loss:105.48467764779684	Accuracy:0.5902777777777778
epoch 16/30 ==>	Loss:105.04455378423691	Accuracy:0.5920138888888888
epoch 17/30 ==>	Loss:104.61884929609955	Accuracy:0.5920138888888888
epoch 18/30 ==>	Loss:104.22293058179834	Accuracy:0.5920138888888888
epoch 19/30 ==>	Loss:103.82850688491358	Accuracy:0.5920138888888888
epoch 20/30 ==>	Loss:103.43622838239177	Accuracy:0.5920138888888888
epoch 21/30 ==>	Loss:103.02971039441869	Accuracy:0.5920138888888888

همان طور که در تصویر می‌بینیم مقادیر loss نسبتاً زیاد است و روند تدریجی بهبود accuracy بسیار کند است. این یکی از تاثیرات تعداد نورون زیاد در لایه مخفی است. تعداد نورون‌های زیاد در لایه مخفی تاثیراتی در شبکه می‌گذارد. تعداد نورون زیاد باعث پیچیده‌تر شدن شبکه می‌شود و همین کار آموزش را دشوارتر می‌کند. این تعداد نورون ممکن است از ظرفیت اصلی داده‌ها برای آموزش بیشتر باشد و چون نسبت اطلاعات به مقادیر قابل آموزش کم است overfitting به وجود می‌آورد. حتی اگر تعداد داده‌ها مناسب باشد، تعداد زیاد نورون‌ها روند آموزش را کند می‌کند و باعث می‌شود زمان بیشتری صرف آموزش شود.

برای امتحان کردن نتیجه تعداد زیاد لایه مخفی، یک شبکه با ۴ لایه در نظر گرفتیم که در لایه‌ها ۱۵ و ۶ و ۳ و ۱ در نظر گرفتیم و سعی کردم این شبکه را آموزش دهم:

```
input_size = X.shape[1]
examples_count = X.shape[0]
y_true = np.reshape(Y, (examples_count, 1))
units_in_each_layer = [15, 6, 3, 1]
activations = ['relu', 'relu', 'relu', 'sigmoid']
learning_rate = 0.01
model = build_MLP(input_size, units_in_each_layer, activations, learning_rate)

loss = []
accuracy = []
epoch_idx = []
epochs = 30
for i in range(epochs):
    y_pred = model.forward_pass(X)
    model.backward_pass(y_pred, y_true, X)
    loss_epoch = model.MSE_loss(y_pred, y_true)
    accuracy_epoch = model.Accuracy(y_pred, y_true)
    print(f'epoch {i+1}/{epochs} ====> Loss:{loss_epoch}      Accuracy:{accuracy_epoch}')
    loss.append(loss_epoch)
    accuracy.append(accuracy_epoch)
    epoch_idx.append(i+1)
```

```
epoch 1/30 ====> Loss:142.1914763824527      Accuracy:0.5
epoch 2/30 ====> Loss:140.28835191632345      Accuracy:0.5
epoch 3/30 ====> Loss:134.2475761469683      Accuracy:0.5
epoch 4/30 ====> Loss:74.70960778018608      Accuracy:0.4340277777777778
epoch 5/30 ====> Loss:143.9943464620662      Accuracy:0.5
epoch 6/30 ====> Loss:143.99429375277128      Accuracy:0.5
epoch 7/30 ====> Loss:143.99424008788782      Accuracy:0.5
epoch 8/30 ====> Loss:143.99418544150416      Accuracy:0.5
epoch 9/30 ====> Loss:143.99412978676858      Accuracy:0.5
```

همان طور که از تصویر مشخص است روند آموزش بسیار کند شده‌است به طوری که بهبود محسوسی در روند مشاهده نمی‌شود و accuracy هم تقریباً روند ثابتی در پیش دارد. هنگامی که تعداد لایه‌های مخفی زیاد می‌شود مقدار خطایی که در لایه‌های قبل propagate می‌شود آن قدر کم می‌شود که تاثیر چندانی ندارد و تقریباً بدون

نتیجه خاصی می ماند. به عبارت دیگر با مشکل vanishing gradient مواجه خواهیم شد. هم چنین به طور کلی complexity شبکه نیز افزایش می یابد و روند آموزش را سخت تر می کند. هم چنین ممکن است با overfitting نیز مواجه شویم.

۳- سوال سوم

دیتاست CIFAR-10 متشکل از ۵۰۰۰۰ تصویر برای training و ۱۰۰۰۰ تصویر برای تست است که تصاویری از هواپیما، اتومبیل، پرنده، گربه، گوزن، سگ، قورباغه، اسب، کشتی و کامیون را دارد که این موارد به ترتیب label های ۰ تا ۹ را به خود اختصاص می دهند. تصاویر در ابعاد (32, 32, 3) در این دیتاست موجود هستند. برای load کردن این دیتاست می توانیم به شکل زیر عمل کنیم:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

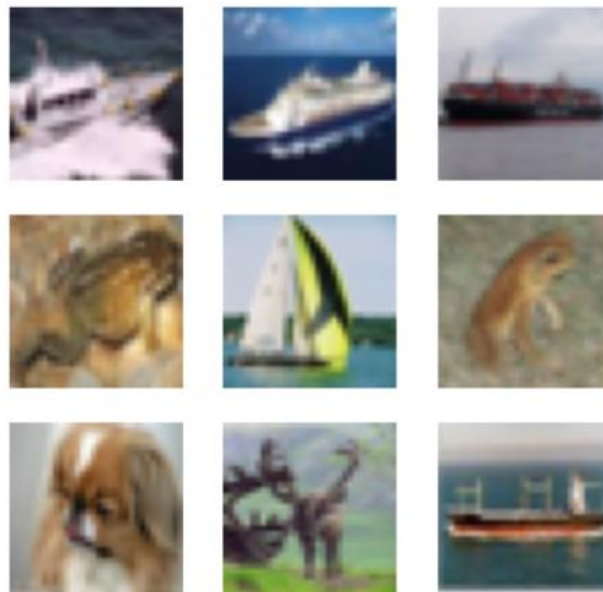
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.shape[2] * x_train.shape[3])
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2] * x_test.shape[3])

print("Train input shape : ", x_train.shape)
print("Train output shape : ", y_train.shape)
print("Test input shape : ", x_test.shape)
print("Test output shape : ", y_test.shape)
```

Downloading data from <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>
170500096/170498071 [=====] - 4s 0us/step
170508288/170498071 [=====] - 4s 0us/step
Train input shape : (50000, 3072)
Train output shape : (50000, 10)
Test input shape : (10000, 3072)
Test output shape : (10000, 10)

بدون استفاده از to_categorical برای label ها، صرفاً آن ها مقادیر ۰ تا ۹ خواهند داشت اما ما می خواهیم به صورت یک ماتریس با ۱۰ درایه باشد که به صورت one_hot بتوان label موردنظر را نشان داد. پس برای این کار از to_categorical استفاده می کنیم. چون ورودی MLP در نهایت ماتریس های سطری خواهند بود، پس ابعادی که پیش تر گفتیم تصاویر ورودی دارند برای آموزش مناسب نخواهند بود. برای حل این مشکل داده های ورودی reshape شده اند تا ابعاد (3072,) پیدا کنند. ابعاد دیتاست هم در تصویر مشخص است. در شکل صفحه بعد نمونه ای از چند تصویر این دیتاست را مشاهده می کنیم:

```
fig, axes1 = plt.subplots(3,3,figsize=(6,6))
for j in range(3):
    for k in range(3):
        i = np.random.choice(50000)
        axes1[j][k].set_axis_off()
        axes1[j][k].imshow(x_train[i,:])
```



برای ساختن یک MLP با کراس از تابع زیر استفاده کردم:

```
def make_Model(
    input_shape,
    layers,
    activations,
    output_classes,
    last_layer_activation = 'sigmoid'
):
    act_dict = {'relu': tf.nn.relu, 'sigmoid' : tf.nn.sigmoid, 'tanh': tf.nn.tanh}
    input = keras.layers.Input(shape=input_shape)
    x = input

    for i in range(len(layers)):
        x = keras.layers.Dense(layers[i], activation=act_dict[activations[i]])(x)

    output = keras.layers.Dense(output_classes, activation=act_dict[last_layer_activation])(x)
    model = Model(inputs=input, outputs=output)

    return model
```

این تابع شکل ورودی را می‌گیرد و هم چنین تعداد نورون‌های هر لایه و تابع فعال‌سازی آن را می‌گیرد. تعداد کلاس‌هایی که در نهایت برای دسته‌بندی نیاز داریم و هم چنین تابع فعال‌سازی لایه آخر دیگر ورودی‌های این تابع هستند. برای ورودی مدل کراس نیاز به لایه Input داریم. این لایه ساینز ورودی را می‌گیرد. برای لایه‌های

دیگر از Dense استفاده می‌کنیم. تعداد نورون‌های لایه آخر همون تعداد کلاس‌های خروجی خواهد بود. در آخر هم برای نهایی کردن مدل از Model استفاده می‌کنیم و input و output را مشخص می‌کنیم.

برای نتیجه بهتر روی دیتاست سعی کردم داده‌های آن را نرمال کنم. برای این کار تمام مقادیر موجود در train و test را بر ۲۵۵ تقسیم کردم تا همه مقادیر آن در این محدوده باشند. برای اینکه تاثیر این کار را متوجه شویم یک بار بدون نرمال‌سازی و یک بار با نرمال‌سازی آموزش را انجام می‌دهیم تا نتایج را مقایسه کنیم. من برای این کار یک mlp تنها با یک لایه مخفی در نظر گرفتم که ۱۰۲۴ نورون در خود دارد. نتایج بدون نرمال‌سازی به صورت زیر است و پارامترهای شبکه در آن مشخص هستند:

```
# First Model with 1 hidden layer - without normalization
input_shape = x_train.shape[1]
classes_count = y_train.shape[1]
layers_units_1 = [1024]
layers_activations_1 = ['relu']
last_layer_activation = 'sigmoid'

# build model
first_model = make_Model(input_shape, layers_units_1, layers_activations_1, classes_count, last_layer_activation)
opt = tf.keras.optimizers.SGD(learning_rate=0.01)
cce = tf.keras.losses.CategoricalCrossentropy()
first_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train,
    y=y_train,
    batch_size=128,
    epochs=10,
    validation_data=(x_test, y_test),
    shuffle=True,
)
```

```
Epoch 1/10
391/391 [=====] - 16s 40ms/step - loss: 7.5255 - accuracy: 0.1001 - val_loss: 7.5223 - val_accuracy: 0.1000
Epoch 2/10
391/391 [=====] - 22s 57ms/step - loss: 7.5220 - accuracy: 0.1000 - val_loss: 7.5223 - val_accuracy: 0.1000
Epoch 3/10
391/391 [=====] - 15s 40ms/step - loss: 7.5220 - accuracy: 0.1000 - val_loss: 7.5223 - val_accuracy: 0.1000
Epoch 4/10
391/391 [=====] - 19s 48ms/step - loss: 7.5220 - accuracy: 0.1000 - val_loss: 7.5223 - val_accuracy: 0.1000
```

همان طور که از تصویر مشخص است accuracy مدل پایین است و loss و accuracy در هر مرحله تغییر خاصی نمی‌کنند. حالا نتیجه را با normalization امتحان می‌کنیم:

```
# First Model with 1 hidden layer - with normalization
x_train_normal = np.divide(x_train, 255.0)
x_test_normal = np.divide(x_test, 255.0)
input_shape = x_train_normal.shape[1]
classes_count = y_train.shape[1]
layers_units_1 = [1024]
layers_activations_1 = ['relu']
last_layer_activation = 'sigmoid'

# build model
first_model = make_Model(input_shape, layers_units_1, layers_activations_1, classes_count, last_layer_activation)
opt = tf.keras.optimizers.SGD(learning_rate=0.01)
cce = tf.keras.losses.CategoricalCrossentropy()
first_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train_normal,
    y=y_train,
    batch_size=128,
    epochs=5,
    validation_data=(x_test_normal, y_test),
    shuffle=True,
)
```

```
Epoch 1/5
391/391 [=====] - 19s 48ms/step - loss: 2.0073 - accuracy: 0.2814 - val_loss: 1.9001 - val_accuracy: 0.3301
Epoch 2/5
391/391 [=====] - 14s 36ms/step - loss: 1.8300 - accuracy: 0.3562 - val_loss: 1.8281 - val_accuracy: 0.3648
Epoch 3/5
```


همان طور که در تصویر مشخص است مقدار loss و accuracy نسبت به حالت قبل تفاوت قابل توجهی دارد و این تفاوت حتی در 2 epoch اول هم محسوس است. مقدار loss به مقدار قابل توجهی کمتر است و مقدار accuracy به مقدار قابل توجهی بیشتر است. برای قسمت‌های بعد نیز از normalization استفاده خواهیم کرد.

مدل بعدی من دو لایه hidden دارد که تعداد نورون ۱۰۲۴ و ۵۱۲ خواهند داشت. با توجه به اینکه داده‌ها زیاد هستند و ویژگی‌های زیادی دارند به نظر می‌رسد این شبکه که کمی پیچیده‌تر است و دو لایه مخفی دارد و لایه دومی تعداد نزدیک‌تری نورون به لایه آخر دارد نتیجه بهتری داشته باشد. نتایج این مدل به صورت زیر خواهد بود:

```
# Second Model with 2 hidden layer - with normalization
layers_units_1 = [1024, 512]
layers_activations_1 = ['relu', 'relu']

# build model
second_model = make_model(input_shape, layers_units_1, layers_activations_1, classes_count, last_layer_activation)
opt = tf.keras.optimizers.SGD(learning_rate=0.01)
cce = tf.keras.losses.CategoricalCrossentropy()
second_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train_normal,
    y=y_train,
    batch_size=128,
    epochs=5,
    validation_data=(x_test_normal, y_test),
    shuffle=True,
)
```

```
Epoch 1/5
391/391 [=====] - 14s 36ms/step - loss: 1.6570 - accuracy: 0.4235 - val_loss: 1.6724 - val_accuracy: 0.4061
Epoch 2/5
391/391 [=====] - 14s 35ms/step - loss: 1.6320 - accuracy: 0.4322 - val_loss: 1.6321 - val_accuracy: 0.4339
Epoch 3/5
391/391 [=====] - 14s 36ms/step - loss: 1.6065 - accuracy: 0.4414 - val_loss: 1.6140 - val_accuracy: 0.4330
Epoch 4/5
391/391 [=====] - 15s 37ms/step - loss: 1.5883 - accuracy: 0.4471 - val_loss: 1.6067 - val_accuracy: 0.4345
Epoch 5/5
391/391 [=====] - 15s 38ms/step - loss: 1.5697 - accuracy: 0.4556 - val_loss: 1.5954 - val_accuracy: 0.4509
```

از تصویر بالا مشخص است که این مدل نتیجه بهتری نسبت به مدل قبل در همان epoch های اولیه داشته است. هر چند روند پیشرفت نسبت به مدل قبل کمی کاهش پیدا کرده‌است زیرا با پیچیده‌تر شدن شبکه فرآیند آموزش نیز پیچیده می‌شود و تغییرات وزن‌ها آرام‌تر انجام می‌شوند.

شبکه بعدی من ۶ لایه مخفی خواهد داشت که تعداد نورون‌ها در آن ۱۰۲۴ و ۵۱۲ و ۲۵۶ و ۱۲۸ و ۶۴ و ۳۲ خواهد بود. به نظر می‌رسد که هر چه شبکه پیچیده‌تر باشد عملکرد بهتری خواهد داشت اما سرعت کاهش خواهد یافت. هر چند در اینجا هم باید مراقب overfitting باشیم. این افزایش درنهایت مرزی خواهد داشت که یا دیگری بهبودی نخواهیم داشت یا مدل دچار overfitt خواهد شد.

نتایج این مدل را در شکل صفحه بعد می‌توانید مشاهده کنید. این مدل هم نسبت مدل قبلی نتایج بهتری دارد ولی باز هم در اینجا روند رشد آرام است و accuracy با سرعت کمتری بالا می‌رود و loss هم با سرعت آهسته‌تری کم می‌شود.

در مراحل بعد به سراغ تاثیر momentum و weight decay بر مدل و نتایج آن خواهیم پرداخت:

```
# Third Model with 6 hidden layer - with normalization
layers_units_1 = [1024, 512, 128, 64, 32, 16]
layers_activations_1 = ['relu', 'relu', 'relu', 'relu', 'relu', 'relu']

# build model
third_model = make_Model(input_shape, layers_units_1, layers_activations_1, classes_count, last_layer_activation)
opt = tf.keras.optimizers.SGD(learning_rate=0.01)
cce = tf.keras.losses.CategoricalCrossentropy()
third_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train_normal,
    y=y_train,
    batch_size=128,
    epochs=5,
    validation_data=(x_test_normal, y_test),
    shuffle=True,
)

Epoch 1/5
391/391 [=====] - 14s 36ms/step - loss: 1.5049 - accuracy: 0.4777 - val_loss: 1.5353 - val_accuracy: 0.4594
Epoch 2/5
391/391 [=====] - 14s 36ms/step - loss: 1.4923 - accuracy: 0.4804 - val_loss: 1.5351 - val_accuracy: 0.4584
Epoch 3/5
391/391 [=====] - 14s 36ms/step - loss: 1.4790 - accuracy: 0.4861 - val_loss: 1.4967 - val_accuracy: 0.4768
Epoch 4/5
391/391 [=====] - 14s 36ms/step - loss: 1.4684 - accuracy: 0.4893 - val_loss: 1.5306 - val_accuracy: 0.4605
Epoch 5/5
391/391 [=====] - 14s 36ms/step - loss: 1.4556 - accuracy: 0.4964 - val_loss: 1.4981 - val_accuracy: 0.4673
```

برای مشاهده عملکرد momentum روی یک مدل از مدل دوم استفاده کردم و momentum را برابر 0.9 قرار دادم. Momentum برای سرعت بخشیدن به فرآیند آموزش استفاده می‌شود. Learning rate به تنهایی نمی‌تواند در هر شرایطی بهترین نتیجه را داشته باشد برای همین momentum به کمک آن می‌آید که این فرآیند را چه در سمت کاهش چه در سمت افزایش سرعت دهد. در این صورت باید برای شبکه دوم نتایج بهتری را در نظر داشته باشیم. نتایج به این صورت است:

```
# Second Model with 2 hidden layer - with normalization - with momentum
layers_units_1 = [1024, 512]
layers_activations_1 = ['relu', 'relu']

# build model
second_model = make_Model(input_shape, layers_units_1, layers_activations_1, classes_count, last_layer_activation)
opt = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
cce = tf.keras.losses.CategoricalCrossentropy()
second_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train_normal,
    y=y_train,
    batch_size=128,
    epochs=5,
    validation_data=(x_test_normal, y_test),
    shuffle=True,
)

Epoch 1/5
391/391 [=====] - 14s 37ms/step - loss: 1.4471 - accuracy: 0.4974 - val_loss: 1.5394 - val_accuracy: 0.4479
Epoch 2/5
391/391 [=====] - 14s 37ms/step - loss: 1.4329 - accuracy: 0.5045 - val_loss: 1.5099 - val_accuracy: 0.4632
Epoch 3/5
391/391 [=====] - 14s 37ms/step - loss: 1.4235 - accuracy: 0.5073 - val_loss: 1.4779 - val_accuracy: 0.4754
Epoch 4/5
391/391 [=====] - 14s 37ms/step - loss: 1.4129 - accuracy: 0.5106 - val_loss: 1.4667 - val_accuracy: 0.4800
Epoch 5/5
391/391 [=====] - 14s 36ms/step - loss: 1.4038 - accuracy: 0.5157 - val_loss: 1.4641 - val_accuracy: 0.4822
```

همان طور که در تصویر مشخص است با توجه به نتایج مشخص است که momentum نتیجه بهتری برای ما فراهم کرده‌است و در کنار learning rate آپدیت وزن‌ها را بهبود بخشیده‌است. پس به نظر می‌رسد استفاده از آن به مدل ما کمک می‌کند.

حالا به سراغ weight decay می‌رویم. Weight decay به ما کمک می‌کند که از vanishing gradient جلوگیری کنیم. هنگامی که نورون‌های یک لایه چنان مقادیر را یاد گرفته‌اند که خروجی آن‌ها دقیقا ۱ یا ۰ است دیگر جایی برای propagate خطا به لایه‌های قبل نمی‌ماند و وزن‌های لایه‌های دیگر نمی‌توانند آپدیت شوند. در اینجا کاهش وزن به ما برای حل مشکل کمک می‌کند. در keras ۳ روش برای کاهش وزن پیشنهاد شده‌است؛ l1 regularization، l2 regularization و ترکیب هر دو روش قبلی. در l1 کاهش وزن روی مقادیر absolute وزن‌ها صورت می‌گیرد و این کار با ضرب l1 انجام می‌شود. در l2 کاهش وزن روی مربع مقادیر وزن‌ها انجام می‌شود و ضرب l2 دارد. ترکیب هر دو نیز استفاده می‌شود. هدف ما در اینجا کاهش وزن‌های لایه‌ها است ولی keras برای bias و مقادیر activation هم weight decay دارد. باز هم در اینجا weight decay را روش شبکه دوم اعمال می‌کنیم که لایه اول پنهان آن l1_l2 و لایه پنهان دوم آن l2 را دارد. انتظار داریم بهبود در کار شبکه حاصل شود زیرا با اعمال آن تغییر وزن واضح‌تری به لایه‌های پنهان اعمال خواهد شد. شکل زیر نتیجه این کار را نشان می‌دهد. پارامترهای regularization در تصویر مشخص هستند:

```
# Weight Decay
input = keras.layers.Input(shape=input_shape)
x = input
x = keras.layers.Dense(1024, activation=tf.nn.relu, kernel_regularizer=regularizers.l1_l2(l1=1e-3, l2=1e-2))(x)
x = keras.layers.Dense(512, activation=tf.nn.relu, kernel_regularizer=regularizers.l2(1e-3))(x)
output = keras.layers.Dense(classes_count, activation=tf.nn.sigmoid)(x)
weight_decay_model = Model(inputs=input, outputs=output)

opt = tf.keras.optimizers.SGD(learning_rate=0.01)
cce = tf.keras.losses.CategoricalCrossentropy()
weight_decay_model.compile(loss=cce, optimizer=opt, metrics=['accuracy'])
history = first_model.fit(
    x=x_train_normal,
    y=y_train,
    batch_size=128,
    epochs=5,
    validation_data=(x_test_normal, y_test),
    shuffle=True)
```

```
Epoch 1/5
391/391 [=====] - 15s 39ms/step - loss: 1.3952 - accuracy: 0.5164 - val_loss: 1.4694 - val_accuracy: 0.4813
Epoch 2/5
391/391 [=====] - 14s 36ms/step - loss: 1.3848 - accuracy: 0.5208 - val_loss: 1.4648 - val_accuracy: 0.4793
Epoch 3/5
391/391 [=====] - 14s 36ms/step - loss: 1.3736 - accuracy: 0.5241 - val_loss: 1.4973 - val_accuracy: 0.4707
Epoch 4/5
391/391 [=====] - 14s 36ms/step - loss: 1.3663 - accuracy: 0.5286 - val_loss: 1.4606 - val_accuracy: 0.4795
Epoch 5/5
391/391 [=====] - 14s 36ms/step - loss: 1.3573 - accuracy: 0.5315 - val_loss: 1.4518 - val_accuracy: 0.4839
```

در اینجا هم نتیجه بهتری را نسبت به حالت اولیه مدل مشاهده می‌کنیم که نشان از تاثیر weight decay در تغییرات propagate شده در مدل است که با قدرت بیشتری انتشار یافته‌اند.

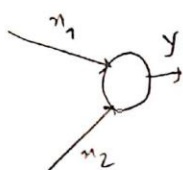
۴- سوال چهارم

- ۱) قابلیت تعمیم از بیشترین به کمترین در شبکه‌های عصبی به ترتیب است: MLP، MADALINE، ADALINE و Perceptron. Perceptron تنها می‌تواند یک خط ارائه دهد که دو کلاس را از یکدیگر جدا کند و قانون آپدیت وزن‌های آن با ۱ و -۱ اجرا می‌شود که دقیق نیست و چیز دقیق ارائه نمی‌دهد. هر چند ADALINE هم مانند Perceptron تنها یک خط جداکننده دارد که دو کلاس را از هم جدا می‌کند ولی تابع آپدیت وزن آن از مقدار خطای خروجی پیروی می‌کند که می‌تواند تخمین دقیق‌تری به ما ارائه بدهد. MADALINE می‌تواند چندین خط برای جداکردن محدوده‌ها به ما بدهد که تخمین دقیق‌تری در این صورت خواهیم داشت ولی باز هم مشکل است که چندین محدوده را برای ما تعیین کند. در نهایت MLP بیشترین تعمیم را خواهد داشت زیرا می‌تواند اشکال مختلفی برای جداسازی کلاس‌ها ارائه دهد و با BP وزن‌ها با یک روش اصولی و به تدریج اصلاح خواهند شد تا بتوانند تقریب بهتری به ما بدهند.
- ۲) معمولاً مشکل overfitting در شبکه‌هایی رخ می‌دهد که تعداد داده‌های آموزشی کم است و نسبت به آن ساختار شبکه بسیار پیچیده است. در این حالت شبکه دچار کمبود داده برای یادگیری وزن‌هایش می‌شود که باعث می‌شود به طور زیادی نسبت به داده‌های ورودی fit شود و در این صورت نسبت به داده‌های دیگر عملکرد خوبی نخواهد داشت. هم چنین هنگامی که یک مدل variance زیادی داشته باشد یا داده‌های نویزی زیادی در بین داده‌ها باشد باز هم overfitting رخ خواهد داد.
- ۳) برای حل مشکل overfitting راه‌های متفاوتی وجود دارد. یک راه این است که ابتدا با تعداد نوروں کمی در لایه hidden کار را شروع کنیم و سپس در صورت لزوم نوروں اضافه کنیم تا شبکه پیچیده نشود. استفاده از test set نیز در کنار داده‌های آموزشی به ما کمک می‌کند که بتوانیم بهتر وضعیت مدل خود را بسنجیم و از مشکل آگاه شویم. Weight Decay هم راهکار مناسبی است که وزن‌ها را به تدریج کم می‌کند و با این کار مانع از overfitting می‌شود. نگه داشتن بهترین وزن‌ها هنگامی که مدل دیگر پیشرفتی نمی‌کند هم موثر است. هم چنین می‌توانیم داده‌های تست را به k دسته تقسیم کنیم و از مقدار خطای کلی آن‌ها میانگین بگیریم.
- ۴) Underfitting زمانی رخ می‌دهد که مدل ما نتواند به رابطه خوبی میان ورودی‌ها و خروجی‌ها دست پیدا کند. در این حالت Bias مدل زیاد است. اگر مدل ما ساده باشد یا داده‌های ما خیلی ساده باشند و تعداد کمی ویژگی داشته باشند ممکن است این اتفاق رخ دهد. برای حل آن می‌توانیم پیچیدگی مدل خود را افزایش دهیم یا از ویژگی‌های بیشتری برای داده‌های ورودی استفاده کنیم.

۵- سوال پنجم

⑤

$$L_v = 0.01 \quad \rho = 0.9$$



$$y = a n_1^2 + b n_2^2 + c n_1 n_2 + d \quad \text{MSE} = \frac{1}{2} (d - y)^2$$

$$a = -1, b = 1, c = -1, d = 2$$

Momentum $\Delta w_{ji}^{(n)} = \alpha \Delta w_{ji}^{(n-1)} + \eta \delta_j^{(n)} x_i^{(n)}$
برای بار اول برابر صفر است

$$\boxed{\text{Epoch 1}}$$

$$\boxed{(1, -1) \rightarrow 10} \quad y = -1 \times 1 - 1 \times 1 + 1 + 2 = \boxed{1}$$

$$\Delta a_1 = \frac{dE_i}{da} = \frac{dE}{dy} \frac{dy}{da} = (d - y) \times n_1^2 = (10 - 1) \times 1 = \boxed{9}$$

$$a = a + \mu \times \Delta a_1 = -1 + 0.01 \times 9 = \boxed{0.91}$$

$$\Delta b_1 = \frac{dE}{db} = \frac{dE}{dy} \frac{dy}{db} = (d - y) \times n_2^2 = 9 \times 1 = \boxed{9}$$

$$b = b + \mu \times \Delta b_1 = 1 + 0.01 \times 9 = \boxed{1.09}$$

$$\Delta c_1 = \frac{dE}{dc} = \frac{dE}{dy} \frac{dy}{dc} = (d - y) \times n_1 n_2 = 9 \times 1 \times -1 = \boxed{-9}$$

$$c = c + \mu \times \Delta c_1 = -1 - 9 \times 0.01 = \boxed{-1.09}$$

$$\Delta d_1 = \frac{dE}{dd} = \frac{dE}{dy} \frac{dy}{dd} = (d - y) \times 1 = \boxed{9}$$

$$d = d + \mu \times \Delta d_1 = 2 + 0.09 = \boxed{2.09}$$

$$\boxed{(2, 0) \rightarrow 13} \quad y = 0.91 \times 4 + 2.09 = \boxed{5.73}$$

$$\Delta a_2 = (d - y) \times n_1^2 = 7.27 \times 4 = 29.08$$

$$\Delta a_2' = 0.9 \times 0.01 \times 9 + 0.01 \times 29.08 = \boxed{0.3718}$$

momentum

$$a = a + \Delta a_2' = 0.91 + 0.3718 = \boxed{1.2818}$$

$$\Delta b_2 = (d-y) \times n_2^2 = 0 \quad \Delta b_2' = 0.9 \times 9 \times 0.01 + 0 = 0.081$$

$$b = b + \Delta b_2' = 1.09 + 0.081 = \boxed{1.171}$$

$$\Delta c_2 = 0 \quad \Delta c_2' = 0.9 \times 0.01 \times -9 = \boxed{-0.081}$$

$$c = c + \Delta c_2' = -1.09 - 0.081 = \boxed{-1.171}$$

$$\Delta d_2 = 1 \times (d-y) \times 7.63 \quad \Delta d_2' = 0.9 \times 9 \times 0.01 + 0.01 \times 7.63 = \boxed{0.1573}$$

$$d = d + \Delta d_2' = 2.09 + 0.1573 = \boxed{2.2473}$$

$$(-1, 2) \rightarrow 15 \quad y = 1.2818 \times 1 + 1.171 \times 4 - 2 \times -1.171 + 2.2473$$

$$= 16.5209$$

$$\Delta a_3 = (15 - 16.5209) \times (-1)^2 = -1.5209$$

$$\Delta a_3' = \frac{0.9}{0.3718} \times \Delta a_2' + \mu \times \Delta a_3 = \boxed{0.18253}$$

$$a = a + \Delta a_3' = \boxed{1.46433}$$

$$\Delta b_3 = -1.5209 \times (2)^2 = \boxed{-6.0836}$$

$$\Delta b_3' = 0.081 \times 0.9 - 6.0836 \times 0.01 = \boxed{0.0720}$$

$$b = b + \Delta b_3' = 1.171 + 0.0720 = \boxed{1.183}$$

$$\Delta c_3 = -1.5209 \times -1 \times 2 = 3.0418$$

$$\Delta c_3' = 0.9 \times -0.081 + 0.01 \times 3.0418 = \boxed{-0.103318}$$

$$c = c + \Delta c_3' = -1.171 + -0.103318 = \boxed{-1.2743}$$

$$\Delta d_3 = -1.5209$$

$$\Delta d_3' = 0.9 \times 0.1573 - 1.5209 \times 0.01 = 0.1263$$

$$d = d + \Delta d_3' = 2.2473 + 0.1263 = \boxed{2.3736}$$