دانشکده مهندسی کامپیوتر

گزارش پایانی درس: مبانی هوش محاسباتی

عنوان پژوهش: تمرین سوم بخش دوم (پیاده سازی عصبی مصنوعی)

ارائه دهنده:

نرگس سادات موسوی جد

شادی شاهی محمدی

استاد درس:

دکتر کارشناس

بهار ۱۴۰۴

فهرست:

۱–تشریح کد:

این برنامه شامل سه قسمت اصلی می‌باشد. بخش اول مربوط به آماده‌سازی داده است. بخش دوم کلاس نورون است که محاسبات یک نورون در آن انجام می‌شود؛ و بخش سوم کلاس شبکه عصبی است که در آن یک شبکه عصبی با امکاناتی نظیر تعیین تعداد لایه‌ها، تعیین تابع فعال‌ساز برای هر لایه و تعیین روش بهینه‌سازی می‌باشد.

۱–۱–توابع آماده‌سازی داده‌ها:

تابع import_data وظیفه‌ی بارگذاری و پیش‌پردازش داده‌های مجموعه‌ی CIFAR-100 را از پنج فایل دسته‌ای (batch) بر عهده دارد. ابتدا مسیرهای فایل‌های داده به‌صورت لیستی در متغیر paths تعریف می‌شوند. سپس با پیمایش این مسیرها، هر فایل به‌صورت باینری باز شده و با استفاده از pickle بارگذاری می‌شود. داده‌های تصویری و برچسب‌ها از هر فایل استخراج و به‌ترتیب به لیست‌های images و labels افزوده می‌شوند. در پایان، داده‌های تصویری به آرایه‌ی NumPy تبدیل شده و برای نرمال‌سازی به محدوده [۰,۱] مقیاس‌بندی می‌شوند. تابع در نهایت آرایه‌ی تصاویر و لیست برچسب‌ها را بازمی‌گرداند.

```
     def import_data():
    paths = ['.\\cifar-100 dataset\\data_batch_1', '.\\cifar-100
dataset\\data_batch_2',
             '.\\cifar-100 dataset\\data_batch_3', '.\\cifar-100
dataset\\data_batch_4',
             '.\\cifar-100 dataset\\data_batch_5']
    images = []
    labels = []
    for path in paths:
        with open(path, 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
            images.extend(dict[b'data'])
            labels.extend(dict[b'labels'])
    # scaling
    images = np.array(images) / 255.0
    return images, labels
```

مشابه همین تابع، در تابع impoer_test نیز داده‌های تست بارگذاری می‌شود.

تابع two_class_encode برای تبدیل برچسب‌های چندکلاسه به برچسب‌های دودویی طراحی شده است. این تابع لیستی از برچسب‌ها (labels) را دریافت کرده و برای هر برچسب بررسی می‌کند که آیا مقدار آن برابر با ۰ است یا خیر. اگر برچسب برابر با ۰ باشد، مقدار ۱ در لیست جدید (binary_labels) قرار می‌گیرد و در غیر این صورت مقدار ۰ افزوده می‌شود.

```
def two_class_encode(labels):
    binary_labels = []
    for label in labels:
        if label == 0:
            binary_labels.append(1)
        else:
            binary_labels.append(0)
    return binary_labels
```

تابع one_hot_encode رچسب‌های عددی را به قالب کدگذاری one-hot تبدیل می‌کند. این تابع یک لیست از برچسب‌ها (labels) و تعداد کلاس‌ها را دریافت می‌کند. برای هر برچسب، یک آرایه‌ی صفر با طول برابر تعداد کلاس‌ها ایجاد می‌شود و تنها عنصری که اندیس آن با مقدار برچسب برابر است، به ۱ تغییر می‌یابد.

```
def one_hot_encode(labels, num_class=10):
    one_hot_labels = []
    for label in labels:
        one_hot_label = np.zeros(num_class)
        one_hot_label[label]=1
        one_hot_labels.append(one_hot_label)
    return one_hot_labels
```

۱-۲-کلاس نورون:

این کلاس شامل توابع زیر است:

۱-سازنده:

ورودی‌های آن شامل num_inputs (تعداد ویژگی‌های ورودی)، activation_func (تابع فعال‌سازی)، differential_activation (مشتق تابع فعال‌سازی) و ha_weight (برای مشخص کردن استفاده از مقداردهی اولیه He) هستند.

```
def __init__(self, num_inputs, activation_func, differential_activation,
ha_weight=False):
    if ha_weight:
        self.weights = np.random.randn(num_inputs+1) * np.sqrt(2. /
num_inputs)
    else:
        self.weights = np.random.randn(num_inputs+1) * 0.01
    self.activation_func = activation_func
    self.differential_activation = differential_activation
```

۲-Setter وgetter

این بخش از کلاس برای تعیین و محافظت از تابع فعال‌سازی و مشتق آن استفاده می‌شود.

```
@property
def activation_func(self):
    return self._activation_func

@activation_func.setter
def activation_func(self, f):
    if not callable(f):
        raise ValueError("Error: this is not a function")
    self._activation_func = f

@property
def differential_activation(self):
    return self._differential_activation

@differential_activation.setter
def differential_activation(self, f):
    if not callable(f):
        raise ValueError("Error: this is not a function")
    self._differential_activation = f
```

۳-get_predicted_label(x)

این تابع برای پیش‌بینی خروجی نورون بر اساس ورودی داده‌شده xاستفاده می‌شود. ابتدا با استفاده از تابع get_z، مجموع وزن‌دار ورودی‌ها (به همراه بایاس) محاسبه می‌شود، سپس نتیجه‌ی حاصل از آن به تابع فعال‌سازی اعمال می‌گردد. این خروجی، مقدار نهایی پیش‌بینی‌شده‌ی نورون برای آن ورودی خاص خواهد بود .

```
def get_predicted_label(self, x):
    z = self.get_z(x)
    return self.activation_func(z)
```

۴-get_z(x)

این تابع مسئول محاسبه‌ی حاصل‌ضرب داخلی بین وزن‌ها و ورودی‌هاست.

```
def get_z(self, x):
    x=np.append(x, 1)
    return np.dot(self.weights, x)
```

۱-۳- کلاس شبکه عصبی:

این کلاس یک شبکه‌ی عصبی را توصیف می‌کند. توابع آن به شرح زیر است:

۱-سازنده کلاس:

شبکه عصبی، از نورون‌های تعریف‌شده با کلاس Neuron ساخته می‌شود. این شبکه از چند لایه‌ی پنهان (hidden layers) و یک لایه‌ی خروجی تشکیل شده و می‌تواند برای مسائل دسته‌بندی دودویی یا چندکلاسه استفاده شود. سازنده‌ی این کلاس از چند بخش تشکیل شده است:

مقداردهی اولیه پارامترها و متغیرهای پایه:

در ابتدای تابع، متغیرهایی برای تنظیم ساختار و رفتار شبکه عصبی مقداردهی می‌شوند. این متغیرها شامل تعداد ویژگی‌های ورودی، اطلاعات مربوط به لایه‌های پنهان (تعداد نورون‌ها و نوع تابع فعال‌سازی)، تعداد کلاس‌های مسئله، اندازه دسته‌ها در آموزش، نرخ یادگیری، مقدار اپسیلون برای جلوگیری از تقسیم بر صفر، تعداد دورهای آموزشی (ایپاک) ، ضریب مومنتوم، و گزینه‌ای برای استفاده از مقداردهی اولیه HE هستند. همچنین لیست‌هایی برای ذخیره نورون‌های لایه‌های پنهان، نورون‌های خروجی، سرعت تغییر وزن‌ها در الگوریتم مومنتوم، و تاریخچه خطا و دقت در طول آموزش تعریف می‌شوند.

```
def __init__(self, num_input, hidden_layers_info, num_class, batch_size=100,
learning_rate=0.1, epsilon=1e-7,
             epoch=20, momentum_gamma=0.9, ha_wight=False):
    self.hidden_layers = []
    self.output_neurons = []
    self.learning_rate = learning_rate
    self.epsilon = epsilon
    self.epoch = epoch
    self.num_class = num_class
    self.momentum_gamma = momentum_gamma
    self.velocities_hidden = []
    self.velocities_output = []
    self.batch_size = batch_size
    self.loss_history = []
    self.accuracy_history = []
```

ایجاد لایه‌های پنهان شبکه:

در این بخش، به ازای هر لایه پنهان که در اطلاعات ورودی مشخص شده، مجموعه‌ای از نورون‌ها ساخته می‌شود. اگر لایه اول باشد، تعداد ورودی‌های نورون‌ها برابر با تعداد ویژگی‌ها خواهد بود. در لایه‌های بعدی، ورودی هر نورون برابر با تعداد نورون‌های لایه قبلی خواهد بود. برای هر نورون، تابع فعال‌سازی و مشتق آن از روی ورودی تنظیم می‌شود و نورون ساخته‌شده به لیست همان لایه اضافه می‌شود.

```
for layer_num in range(len(hidden_layers_info)):
    layer = []
    input_size = num_input if layer_num == 0 else
hidden_layers_info[layer_num - 1][0]
    for _ in range(hidden_layers_info[layer_num][0]):
        layer.append(Neuron(input_size, hidden_layers_info[layer_num][1],
hidden_layers_info[layer_num][2], ha_wight))
    self.hidden_layers.append(layer)
    self.velocities_hidden.append([np.zeros_like(neuron.weights) for neuron
in layer])

output_input_size = num_input if len(self.hidden_layers) == 0 else
hidden_layers_info[-1][0]
```

ساخت لایه خروجی:

در مرحله پایانی، نورون‌های لایه خروجی ساخته می‌شوند. اگر مسئله دسته‌بندی دودویی باشد، تنها یک نورون خروجی با تابع فعال‌سازی "سیگموید" و مشتق آن تعریف می‌شود. اما اگر تعداد کلاس‌ها بیشتر از یک باشد، به ازای هر کلاس، یک نورون خروجی با تابع همانی و مشتق ثابت یک ساخته می‌شود.

```
if num_class == 1:
    self.output_neurons.append(
        Neuron(output_input_size, self.sigmoid, self.sigmoid_derivative,
ha_wight))
    self.velocities_output = [np.zeros_like(self.output_neurons[0].weights)]
else:
    for _ in range(num_class):
        self.output_neurons.append(
            Neuron(output_input_size, lambda x: x, lambda x: 1, ha_wight))
    self.velocities_output = [np.zeros_like(neuron.weights) for neuron in
self.output_neurons]
```

۲-توابع فعال‌ساز و softmax

در ادامه توابع فعال‌ساز مورد نیاز و مشتق آن‌ها تعریف شده است:

```
@staticmethod
def sigmoid(x):
    x = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x))

@staticmethod
def sigmoid_derivative(x):
    s = NeuralNetwork.sigmoid(x)
    return s * (1 - s)

@staticmethod
def relu(x):
    return np.maximum(0, x)
```

```
@staticmethod
def relu_derivative(x):
    return (x > 0).astype(float)
```

در زیر تابع softmax که در طبقه‌بندی چند کلاسه، برای بدست آوردن توزیع احتمال تعلق به هر کلاس استفاده می‌شود، پیاده‌سازی شده است:

```
    def softmax(self, z_i):
    z_i = np.clip(z_i, -500, 500)
    exp_z = np.exp(z_i - np.max(z_i))
    sum_exp_z = np.sum(exp_z)
    return exp_z / sum_exp_z
```

۳-تابع get_output

تابع get_output وظیفه دارد که یک نمونه ورودی را از ابتدا تا انتهای شبکه عصبی عبور دهد و خروجی نهایی شبکه را محاسبه کند. ابتدا ورودی اولیه در لیستی ذخیره می‌شود که قرار است خروجی هر لایه را نیز به ترتیب در خود نگه دارد. سپس در یک حلقه، هر لایه پنهان شبکه طی می‌شود و برای هر نورون آن لایه، خروجی (یعنی مقدار پس از تابع فعال‌سازی) و مقدار z محاسبه شده و به لیست‌های مربوطه اضافه می‌گردد.

پس از عبور از همه لایه‌های پنهان، ورودی به لایه خروجی داده می‌شود. اگر شبکه فقط یک کلاس خروجی داشته باشد (یعنی مسئله دودویی باشد)، خروجی نهایی با تابع سیگموید محاسبه می‌شود، در غیر این صورت از تابع سافت‌مکس برای محاسبه احتمال دسته‌ها استفاده می‌گردد. در نهایت، این تابع سه مقدار برمی‌گرداند: خروجی نهایی شبکه، خروجی‌های لایه‌ها، و مقادیر تحریک (z) هر لایه که برای استفاده در مرحله پس‌انتشار ضروری هستند.

```
def get_output(self, x):
    input_for_layers = [x]
    z_layers = []

    for layer in self.hidden_layers:
        hidden_output = [neuron.get_predicted_label(input_for_layers[-1]) for
neuron in layer]
        z_layer = [neuron.get_z(input_for_layers[-1]) for neuron in layer]
        input_for_layers.append(hidden_output)
        z_layers.append(z_layer)

    output_input = input_for_layers[-1]
    output_z = [neuron.get_z(output_input) for neuron in self.output_neurons]
    z_layers.append(output_z)
```

```
    if self.num_class == 1:
        output = self.sigmoid(output_z[0])
    else:
        output = self.softmax(output_z)

    return output, input_for_layers, z_layers
```

۴- تابع multi_class_cross_entropy_loss

تابع multi_class_cross_entropy_loss وظیفه محاسبه‌ی خطای مدل (تابع هزینه) را دارد که بر اساس معیار "کراس انتروپی (Cross-Entropy) "تعریف شده است. اگر شبکه برای دسته‌بندی دودویی آموزش دیده باشد (یعنی تعداد کلاس‌ها برابر با یک باشد)، از فرمول کراس انتروپی مخصوص دسته‌بندی دودویی استفاده می‌شود.

برای جلوگیری از بروز خطای عددی در محاسبه لگاریتم صفر، مقدار کوچکی به نام اپسیلون به مقادیر اضافه می‌گردد. اگر شبکه برای دسته‌بندی چندکلاسه باشد، از فرمول کلی‌تر کراس انتروپی استفاده می‌شود که در آن ضرب عنصر به عنصر بین برچسب‌های واقعی و لگاریتم خروجی پیش‌بینی‌شده محاسبه و در نهایت مجموع گرفته می‌شود. نتیجه این تابع یک عدد مثبت است که بیانگر میزان خطای مدل برای یک نمونه است.

```
def multi_class_cross_entropy_loss(self, y_true, y_pred):
    if self.num_class == 1:
        y_true = y_true[0] if isinstance(y_true, (list, np.ndarray)) else
y_true
        y_pred = y_pred[0] if isinstance(y_pred, (list, np.ndarray)) else
y_pred
        return -(y_true * math.log(y_pred + self.epsilon) +
                 (1 - y_true) * math.log(1 - y_pred + self.epsilon))
    else:
        return -np.sum(y_true * np.log(y_pred + self.epsilon))
```

۵- تابع backpropagation

تابع backpropagation وظیفه محاسبه‌ی مقادیر دلتا (خطاهای محلی) برای تمام نورون‌های شبکه عصبی را دارد که یکی از مراحل اصلی در یادگیری به روش پس‌انتشار خطا (Backpropagation) است. این دلتاها بعداً برای به‌روزرسانی وزن نورون‌ها استفاده می‌شوند.

ابتدا با استفاده از تابع get_output، مقدار خروجی پیش‌بینی‌شده‌ی شبکه و مقادیر z برای هر لایه محاسبه می‌شود. سپس آرایه‌ای به نام deltas برای ذخیره خطای هر لایه ایجاد می‌گردد.

اگر شبکه برای دسته‌بندی دودویی باشد، مقدار دلتا در خروجی به‌صورت تفاضل بین مقدار پیش‌بینی‌شده و مقدار درست محاسبه می‌شود. در غیر این صورت (در مسائل چندکلاسه)، دلتا خروجی به‌صورت تفاضل برداری بین خروجی شبکه و بردار برچسب‌های صحیح محاسبه می‌شود.

در ادامه، برای هر لایه پنهان به‌ترتیب از انتهای شبکه به سمت ابتدای آن، مقدار دلتا با استفاده از فرمول استاندارد مشتق ترکیبی محاسبه می‌شود. در این فرمول، ابتدا وزن‌های نورون‌های لایه‌ی بعدی و دلتاهای آن لایه در نظر گرفته می‌شوند و سپس با استفاده از مشتق تابع فعال‌سازی در مقدار z مربوط به هر نورون، مقدار دلتا برای آن نورون به‌دست می‌آید. این مقدار خطا در نهایت برای تمام لایه‌ها محاسبه و به صورت لیستی از آرایه‌ها بازگردانده می‌شود.

```
def backpropagation(self, x, y_true):
    y_pred, _, z_layers = self.get_output(x)
    deltas = [None] * (len(self.hidden_layers) + 1)

    if self.num_class == 1:
        y_val = y_true[0] if isinstance(y_true, (list, np.ndarray)) else
y_true
        deltas[-1] = np.array([y_pred - y_val])
    else:
        deltas[-1] = y_pred - y_true

    for l in reversed(range(len(self.hidden_layers))):
        delta_next = deltas[l + 1]
        weights_next = np.array([n.weights for n in (
            self.output_neurons if l == len(self.hidden_layers) - 1 else
self.hidden_layers[l + 1])])
        delta_current = []

        for j, neuron in enumerate(self.hidden_layers[l]):
            z = z_layers[l][j]
            g_prime = neuron.differential_activation(z)
            error = np.dot(weights_next[:, j], delta_next)
            delta_current.append(error * g_prime)

        deltas[l] = np.array(delta_current)

    return deltas
```

۶- تابع make_batch

این تابع برای تقسیم کردد مجموعه داده‌ی آموزشی به دسته‌های کوچک‌تر است که برای یادگیری دسته‌ای استفاده می‌شود:

```
def make_batches(self, x_train, y_train):
    return [(x_train[i:i + self.batch_size], y_train[i:i + self.batch_size])
            for i in range(0, len(x_train), self.batch_size)]
```

۷-تابع tarin

تابع trainمسئول آموزش شبکه عصبی با استفاده از داده‌های آموزشی (ویژگی‌ها و برچسب‌ها) و بر مبنای الگوریتم پس‌انتشار خطا (Backpropagation) است. در ابتدا، تاریخچه‌ی خطا و دقت برای هر دوره‌ی آموزشی (epoch)ریست می‌شود. اگر هیچ بهینه‌سازی مشخص نشده باشد، از gradient_descentبه‌صورت پیش‌فرض استفاده می‌شود.

در هر دوره‌ی آموزشی، داده‌ها به دسته‌های کوچک (batch) تقسیم می‌شوند و برای هر دسته، گرادیان وزن‌ها در لایه‌های پنهان و خروجی با مقدار صفر مقداردهی اولیه می‌گردد. سپس برای هر نمونه در دسته، خروجی شبکه محاسبه شده، خطا با استفاده از تابع multi_class_cross_entropy_lossبه‌دست می‌آید و دلتاها با تابع backpropagationاستخراج می‌شوند. سپس با استفاده از دلتاها، گرادیان نسبت به وزن برای هر نورون در لایه‌های خروجی و پنهان محاسبه و ذخیره می‌شوند.

پس از اتمام هر دسته، وزن‌های نورون‌ها با استفاده از گرادیان‌های محاسبه‌شده و تابع بهینه‌سازی مشخص‌شده، به‌روزرسانی می‌شوند. سپس میانگین خطای کل دوره محاسبه شده و در تاریخچه ذخیره می‌گردد. در ادامه، دقت شبکه روی کل داده‌های آموزش محاسبه می‌شود و اگر در چند دوره‌ی متوالی، کاهش خطا کمتر از مقدار آستانه min_deltaباشد، با استفاده از تابع check_early_stoppingآموزش به‌صورت زودهنگام متوقف می‌گردد. این کار از بیش‌ازحد آموزش دادن (overfitting) جلوگیری می‌کند.

```
def train(self, x_train, y_train, optimizer=None, patience=5, min_delta=1e-
4):
    self.loss_history = []
    self.accuracy_history = []

    if optimizer is None:
        optimizer = self.gradient_descent

    best_loss = float('inf')
    epochs_without_improvement = 0

    for epoch in range(self.epoch):
        total_loss = 0
        batches = self.make_batches(x_train, y_train)

        for x_batch, y_batch in batches:
```

```
            grad_output = [np.zeros_like(neuron.weights) for neuron in
self.output_neurons]
            grad_hidden = [[np.zeros_like(neuron.weights) for neuron in
layer] for layer in self.hidden_layers]

            batch_loss = 0
            for x, y_true in zip(x_batch, y_batch):
                y_pred, input_for_layers, _ = self.get_output(x)
                batch_loss += self.multi_class_cross_entropy_loss(y_true,
y_pred)
                deltas = self.backpropagation(x, y_true)

                output_input = np.append(input_for_layers[-1], 1)
                for i, neuron in enumerate(self.output_neurons):
                    grad_output[i] += deltas[-1][i] * output_input

                for l, layer in enumerate(self.hidden_layers):
                    hidden_input = np.append(input_for_layers[l], 1)
                    for j, neuron in enumerate(layer):
                        grad_hidden[l][j] += deltas[l][j] * hidden_input

            for i, neuron in enumerate(self.output_neurons):
                neuron.weights -= optimizer(grad_output[i] / len(x_batch),
layer='output', idx=i)

            for l, layer in enumerate(self.hidden_layers):
                for j, neuron in enumerate(layer):
                    neuron.weights -= optimizer(grad_hidden[l][j] /
len(x_batch),
                                                layer='hidden', layer_idx=l,
idx=j)

            total_loss += batch_loss

        avg_loss = total_loss / len(x_train)
        self.loss_history.append(avg_loss)

        correct = 0
        for x, y in zip(x_train, y_train):
            output, _, _ = self.get_output(x)
            if self.num_class == 1:
                pred = 1 if output.item() >= 0.5 else 0
                true = int(y)
            else:
                pred = np.argmax(output)
                true = np.argmax(y)
            correct += (pred == true)

        accuracy = correct / len(x_train)
        self.accuracy_history.append(accuracy)
        print(f"Epoch {epoch + 1}/{self.epoch} - Loss: {avg_loss:.4f} -
Accuracy: {accuracy:.4f}")

        should_stop, best_loss, epochs_without_improvement =
self.check_early_stopping(
            avg_loss, best_loss, epochs_without_improvement, min_delta,
patience)
```

```
        if should_stop:
            print(f"Early stopping triggered at epoch {epoch + 1}")
            break
```

۸- تابع check_early_stopping

تابع check_early_stopping برای پیاده‌سازی توقف زودهنگام (Early Stopping) در فرآیند آموزش استفاده می‌شود تا از بیش‌ازحد آموزش دادن مدل جلوگیری شود. این تابع بررسی می‌کند که آیا در مقدار میانگین خطا (avg_loss) نسبت به بهترین خطای ثبت‌شده تا این لحظه (best_loss) بهبود قابل‌توجهی حاصل شده یا نه. اگر اختلاف بین بهترین خطا و خطای فعلی بیشتر از آستانه تعیین‌شده (min_delta) باشد، یعنی مدل بهبود داشته، در نتیجه متغیر شمارنده‌ی عدم بهبود ریست می‌شود (به صفر برمی‌گردد) و آموزش ادامه می‌یابد. اما اگر این بهبود حاصل نشده باشد، شمارنده‌ی epochs_without_improvement یک واحد افزایش می‌یابد. اگر این شمارنده به مقدار تعیین‌شده توسط patience برسد، آموزش متوقف خواهد شد.

```
def check_early_stopping(self, avg_loss, best_loss,
epochs_without_improvement, min_delta, patience):
    if best_loss - avg_loss > min_delta:
        return False, avg_loss, 0
    else:
        return epochs_without_improvement + 1 >= patience, best_loss,
epochs_without_improvement + 1
```

۹- توابع بهینه‌ساز

تابع gradient_descent و تابع momentum دو روش متفاوت برای به‌روزرسانی وزن‌ها در فرآیند آموزش شبکه عصبی هستند. تابع gradient_descent ساده‌ترین شکل الگوریتم نزول گرادیان است که صرفاً گرادیان را در نرخ یادگیری ضرب کرده و برای به‌روزرسانی وزن‌ها بازمی‌گرداند؛ این روش بدون در نظر گرفتن وضعیت قبلی وزن‌ها عمل می‌کند.

اما تابع momentum نسخه‌ای پیشرفته‌تری است. در این روش، علاوه بر گرادیان فعلی، جهت حرکت قبلی (ذخیره‌شده در متغیر سرعت v) نیز در نظر گرفته می‌شود. این باعث می‌شود به‌روزرسانی وزن‌ها نرم‌تر و پایدارتر باشد و از گیرافتادن در مینیمم‌های محلی جلوگیری شود. تابع با استفاده از پارامترهای layer (لایه مورد نظر)، layer_idx (شماره لایه) و idx (شماره نورون) مشخص می‌کند که گرادیان مربوط به کدام نورون است، سپس مقدار جدید سرعت را محاسبه و ذخیره می‌کند و در نهایت آن را به عنوان گام به‌روزرسانی وزن برمی‌گرداند.

```
def gradient_descent(self, gradient, **kwargs):
    return self.learning_rate * gradient

def momentum(self, gradient, **kwargs):
    layer = kwargs.get('layer', None)
    layer_idx = kwargs.get('layer_idx', None)
    idx = kwargs.get('idx', None)

    if layer == 'output':
        v = self.velocities_output[idx]
        v_new = self.momentum_gamma * v + self.learning_rate * gradient
        self.velocities_output[idx] = v_new
        return v_new
    elif layer == 'hidden':
        v = self.velocities_hidden[layer_idx][idx]
        v_new = self.momentum_gamma * v + self.learning_rate * gradient
        self.velocities_hidden[layer_idx][idx] = v_new
        return v_new
    else:
        return self.learning_rate * gradient
```

۱۰- تابع test_network

تابع test_network برای ارزیابی عملکرد مدل آموزش‌دیده روی داده‌های تست به کار می‌رود. ابتدا برای هر نمونه‌ی ورودی در x_test، خروجی شبکه با استفاده از تابع get_output محاسبه می‌شود. پس از آن، معیارهایی نظیر ماتریس آشفتگی(confusion matrix) ، نمره F1 و گزارش F1 برای هر کلاس محاسبه می‌شود. همچنین، دو نمودار برای تاریخچه‌ی خطا (Loss) و دقت (Accuracy) ترسیم می‌شود.

```
def test_network(self, x_test, y_test):
    y_pred = []
    for x in x_test:
        output, _, _ = self.get_output(x)
        if self.num_class == 1:
            pred = 1 if output.item() >= 0.5 else 0
        else:
            pred = np.argmax(output)
        y_pred.append(pred)

    y_true_labels = [int(y) if self.num_class == 1 else np.argmax(y) for y in
y_test]
    cm = confusion_matrix(y_true_labels, y_pred)
    f1 = f1_score(y_true_labels, y_pred, average='macro')
    c = classification_report(y_true_labels, y_pred)  # ← fixed here
    self.plot_loss()
    self.plot_accuracy()
    return cm, f1, c
```

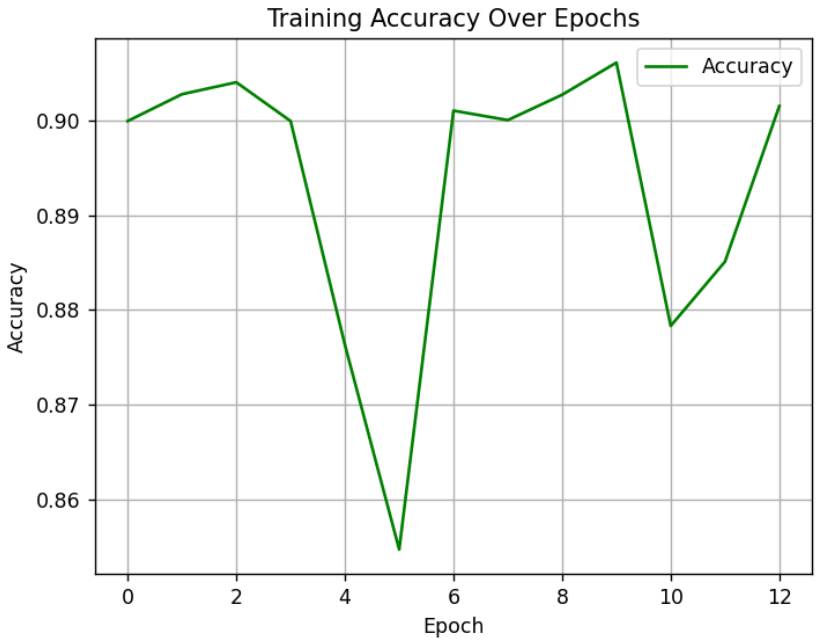
۲–نتایج:

نتایج طبقه‌بندی تک کلاسه با یک نورون:

```
[[8991    9]
 [ 981   19]]
0.49239140867844566
              precision    recall  f1-score   support

           0       0.90      1.00      0.95      9000
           1       0.68      0.02      0.04      1000

    accuracy                           0.90     10000
   macro avg       0.79      0.51      0.49     10000
weighted avg       0.88      0.90      0.86     10000
```



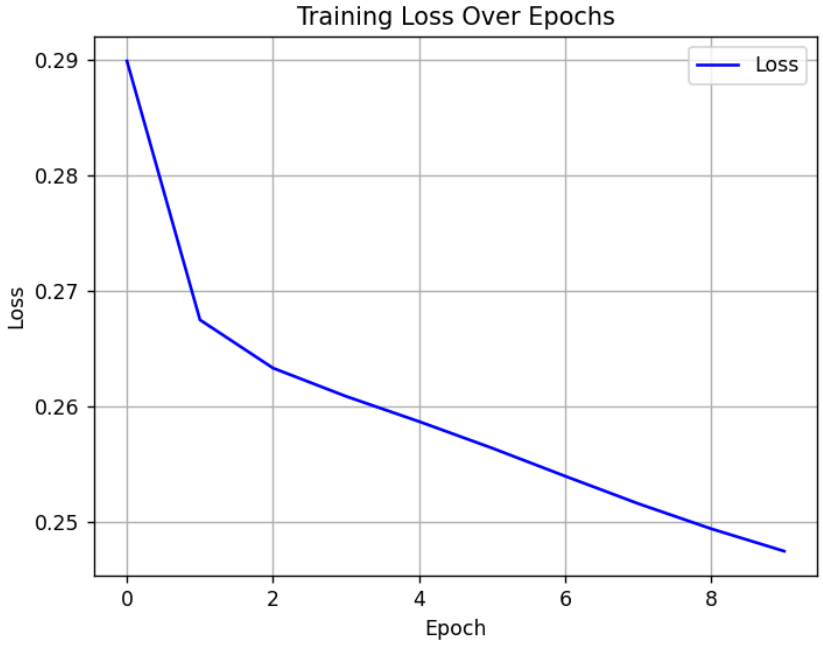
بدلیل آموزش دسته‌ای (mini batch)، نوساناتی در نمودار دقت و خطا مشاهده می‌شود.

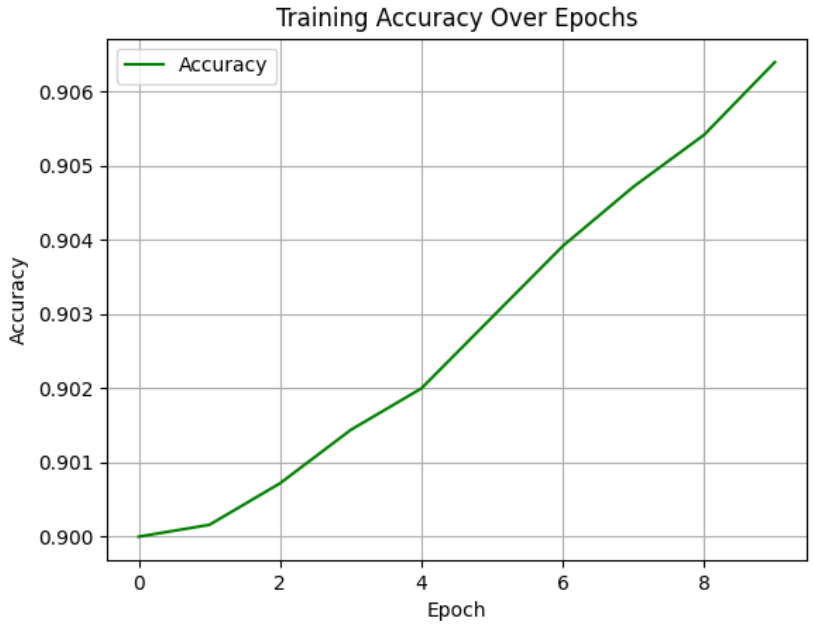
نتایج شبکه تک کلاسه، با لایه پنهان ۶۴ نورونی: همانطور که مشاهده می‌شود اضافه کردن لایه پنهان، باعث بهبود نتایج می‌شود:

```
[[8978   22]
 [ 920   80]]
0.5476720090027676
              precision    recall  f1-score   support

           0       0.91      1.00      0.95      9000
           1       0.78      0.08      0.15      1000

    accuracy                           0.91     10000
   macro avg       0.85      0.54      0.55     10000
weighted avg       0.89      0.91      0.87     10000
```



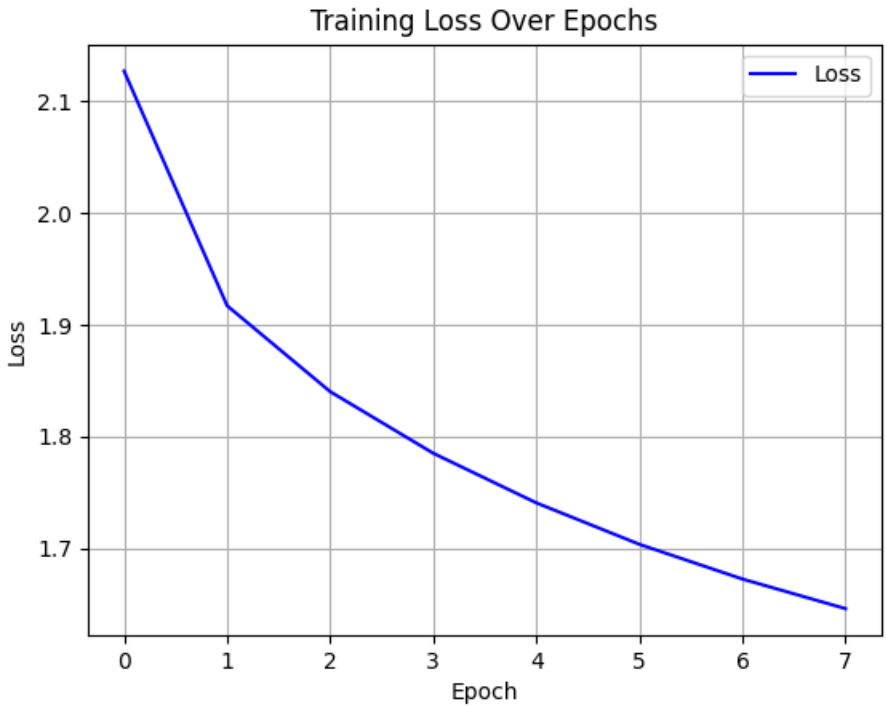


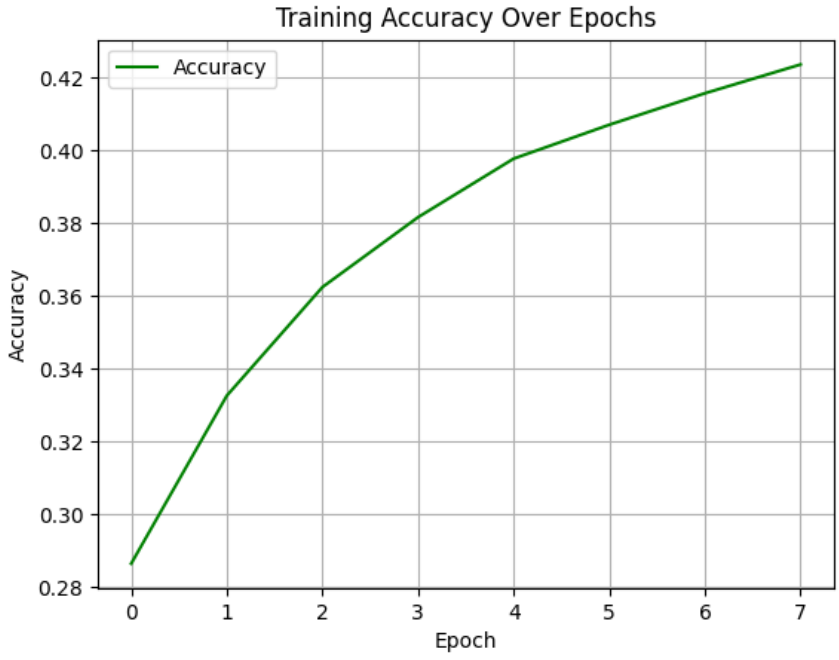
نتایج طبقه‌بندی چند کلاسه با لایه‌ی پنهان ۶۴ نورونی: بدلیل اینکه تعداد کلاس‌ها افزایش یافته و مدل دقیقا باید شکل داخل تصویر را به یکی از ۱۰ کلاس ممکن نسبت دهد، دقت مقداری افت کرده است.

```
[[454 110  22  25  14  21  19  75 207  53]
 [ 29 668   6  24  10  16  23  41  75 108]
 [116  83 156  89 157  78 121 129  52  19]
 [ 37 140  43 275  44 166 100 105  42  48]
 [ 65  72  96  58 309  53 124 174  33  16]
 [ 31  99  57 181  51 300  86 135  41  19]
 [  8  93  48 130 105  64 454  55  17  26]
 [ 38  84  22  63  61  44  33 554  29  72]
 [ 96 160   0  27   7  23   8  20 594  65]
 [ 36 333   3  25   6  10  33  54 102 398]]
0.4043759419234433
              precision    recall  f1-score   support

           0       0.50      0.45      0.48      1000
           1       0.36      0.67      0.47      1000
           2       0.34      0.16      0.21      1000
           3       0.31      0.28      0.29      1000
           4       0.40      0.31      0.35      1000
           5       0.39      0.30      0.34      1000
           6       0.45      0.45      0.45      1000
           7       0.41      0.55      0.47      1000
           8       0.50      0.59      0.54      1000
           9       0.48      0.40      0.44      1000

    accuracy                           0.42     10000
   macro avg       0.42      0.42      0.40     10000
weighted avg       0.42      0.42      0.40     10000
```



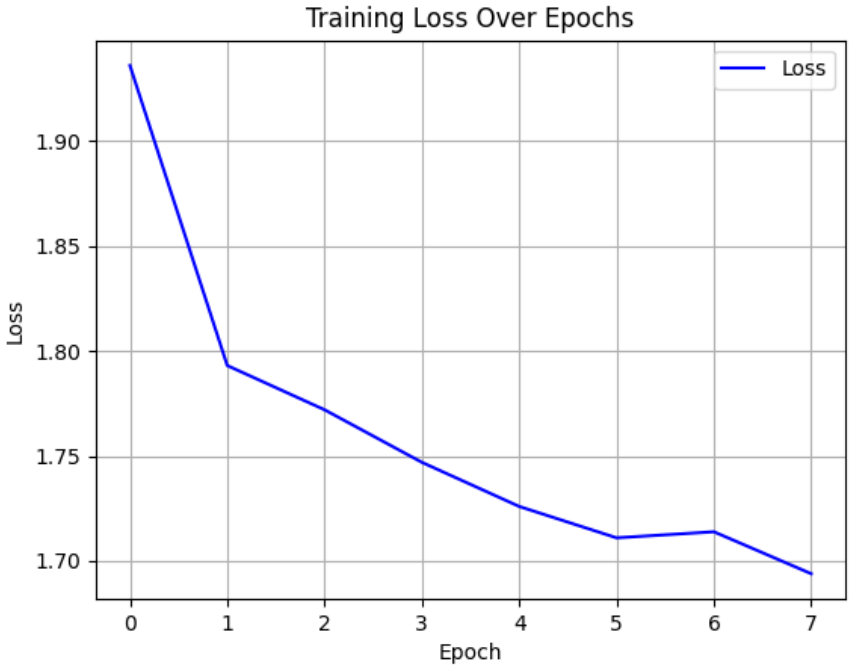


نتایج طبقه‌بندی چند کلاسه با لایه‌ی پنهان ۶۴ نورونی و بهینه‌ساز مومنتوم: نتایج نشان داد که بهینه‌ساز مومنتوم سرعت همگرایی بیشتری دارد.

```
[[509  38  71  16  11  38  16  32 236  33]
 [ 44 510  17  25   7  52  22  30 142 151]
 [180  30 251  53  64 201 107  61  34  19]
 [ 68  43  68 156  33 409  79  50  44  50]
 [108  21 168  33 204 195 128  89  36  18]
 [ 53  35  80 104  32 496  57  66  48  29]
 [ 30  24  94  77  73 222 402  32  28  18]
 [ 89  51  46  45  68 164  33 403  39  62]
 [145  75  15  13   7  46  10  15 623  51]
 [ 52 180  13  30   6  44  36  44 161 434]]
0.3902912542871542
              precision    recall  f1-score   support

           0       0.40      0.51      0.45      1000
           1       0.51      0.51      0.51      1000
           2       0.30      0.25      0.28      1000
           3       0.28      0.16      0.20      1000
           4       0.40      0.20      0.27      1000
           5       0.27      0.50      0.35      1000
           6       0.45      0.40      0.43      1000
           7       0.49      0.40      0.44      1000
           8       0.45      0.62      0.52      1000
           9       0.50      0.43      0.47      1000

    accuracy                           0.40     10000
   macro avg       0.41      0.40      0.39     10000
weighted avg       0.41      0.40      0.39     10000
```


Training Loss Over Epochs
Loss
1.90
1.85
1.80
1.75
1.70
Loss
0
1
2
3
4
5
6
7
Epoch

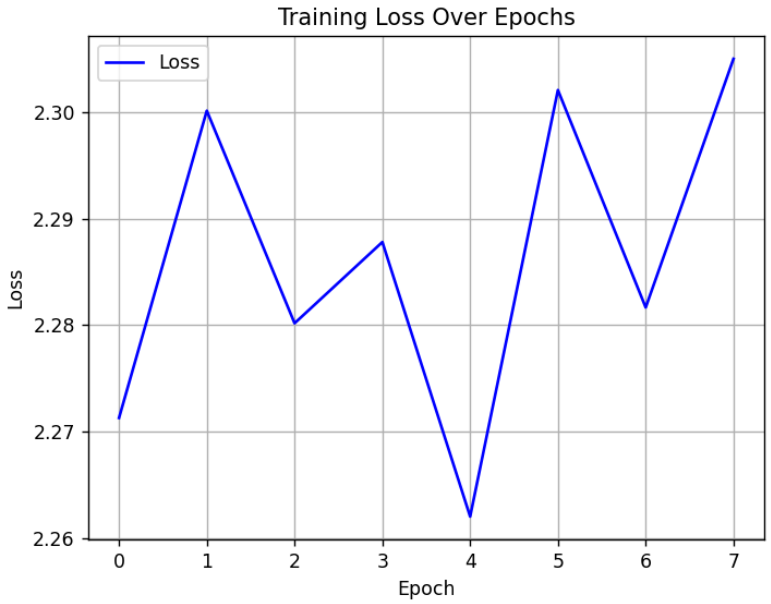

Training Accuracy Over Epochs
Accuracy
0.41
0.40
0.39
0.38
0.37
Accuracy
0
1
2
3
4
5
6
7
Epoch

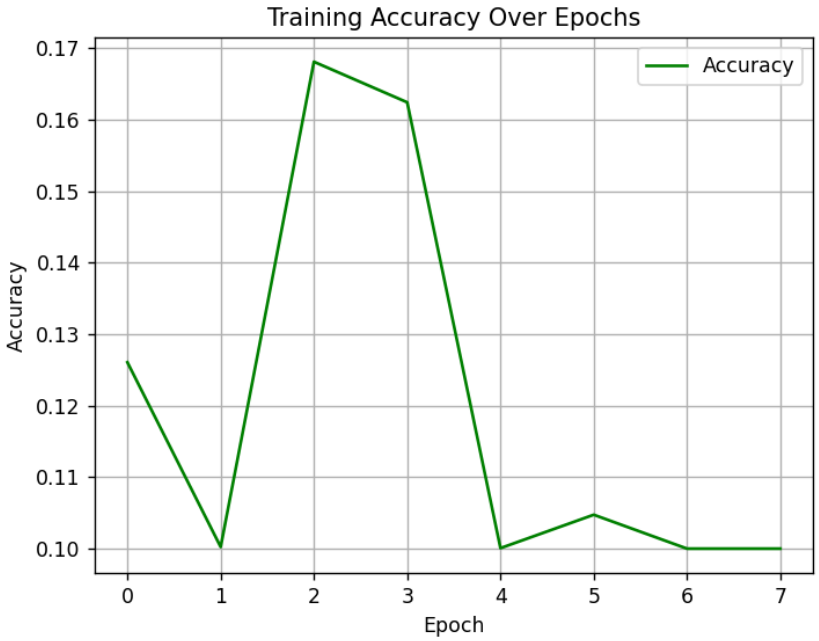
نتایج طبقه‌بندی چند کلاسه با لایه‌ی پنهان ۶۴ نورونی و تابع فعال‌ساز relu: در این حالت مدل ناپایدار شده است و نتیجه مطلوبی نداشت.

```
0.01818181818181818
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      1000
           1       0.00      0.00      0.00      1000
           2       0.00      0.00      0.00      1000
           3       0.00      0.00      0.00      1000
           4       0.10      1.00      0.18      1000
           5       0.00      0.00      0.00      1000
           6       0.00      0.00      0.00      1000
           7       0.00      0.00      0.00      1000
           8       0.00      0.00      0.00      1000
           9       0.00      0.00      0.00      1000

    accuracy                           0.10     10000
   macro avg       0.01      0.10      0.02     10000
weighted avg       0.01      0.10      0.02     10000
```





لینک گیت هاب: https://github.com/Rshshad/neural_network_homework3-