گزارش پروژه شبکه های عصبی

بخش اول – لایه ها

۱-۱ لایه Fully Connected :

تکمیل تابع وزن دهی:

- **اگر متد وزن دهی رندم باشد:**

به صورت زیر فراخوانی تابع رندم نامپای انجام شده و وزن های رندوم انتخاب میشوند، سپس با ضرب کردن آنها در ۰.۰۱ آنها را اسکیل میکنیم.

```
-   If self.initialize_method == "random":
        # Initialize weights with random values using np.random.randn
        return np.random.randn(self.output_size, self.input_size) * ۰,۰۱
```

- **اگر متد وزن دهی xavier باشد:**

مقداردهی اولیه xavier یک روش رایج برای مقداردهی اولیه وزن‌ها در یک لایه شبکه عصبی است که می‌تواند به شبکه کمک کند در طی فرآیند آموزش به سرعت به جواب مسئله برسد. این روش از یک توزیع گاوسی برای مقداردهی اولیه وزن‌ها استفاده می‌کند که واریانس آن به نسبت جذر معکوس تعداد ورودی‌ها به لایه جاری به اضافه تعداد خروجی‌های آن لایه است.

در این قطعه کد، متغیر self.input_size تعداد نورون‌های لایه قبلی را نشان می‌دهد، در حالی که self.output_size تعداد نورون‌های لایه جاری را نمایش می‌دهد.کد، مقدار واریانس مورد نیاز برای مقداردهی اولیه وزن‌ها را با استفاده از فرمول مربوط به روش مقداردهی اولیه xavierمحاسبه می‌کند و سپس با استفاده از تابع np.random.randn در NumPy، یک ماتریس تصادفی با ابعاد (self.output_size, self.input_size) تولید کرده و آن را در ضریب محاسبه شده برای واریانس، ضرب می‌کند. در نهایت، این ماتریس به عنوان وزن‌های مقداردهی اولیه لایه برگردانده می‌شود

```
elif self.initialize_method == "xavier":
    # Initialize weights using Xavier initialization
    xavier_stddev = np.sqrt(۲ / (self.input_size + self.output_size))
    return np.random.randn(self.output_size, self.input_size) * xavier_stddev
```

- **اگر متد وزن دهی he باشد:**

مقداردهی اولیه he نیز یک روش رایج برای مقداردهی اولیه وزن‌های یک لایه شبکه عصبی است که به شبکه کمک می‌کند در طی فرآیند آموزش به سرعت به جواب مسئله برسد. در این روش، مقدار واریانس اولیه وزن‌ها برابر با دو برابر نسبت جذر معکوس تعداد ورودی‌ها به لایه جاری است. در این قطعه کد، متغیر self.input_size تعداد نورون‌های لایه قبلی را نشان می‌دهد. ابتدا با استفاده از فرمول مربوط به روش مقداردهی اولیه هی، واریانس اولیه وزن‌ها محاسبه می‌شود و سپس با استفاده از تابع np.random.randn در NumPy، یک ماتریس تصادفی با ابعاد (self.output_size, self.input_size) تولید می‌شود و در ضریب محاسبه شده برای واریانس ضرب می‌شود. در نهایت، این ماتریس به عنوان وزن‌های مقداردهی اولیه لایه برگردانده می‌شود.

```
-   elif self.initialize_method == "he":
        # Initialize weights using He initialization
        he_stddev = np.sqrt(۲ / self.input_size)
        return np.random.randn(self.output_size, self.input_size) *
    he_stddev
```

تکمیل تابع تعیین بایاس:

```
def initialize_bias(self):
    # Initialize bias with zeros
    return np.zeros((self.output_size, ۱))
```

در این تابع، با استفاده از تابع np.zeros در پکیج NumPy، برداری به اندازه تعداد نورون‌های لایه جاری که با self.output_size مشخص شده است، تولید می‌شود. اندازه دوم این بردار برابر یک است، زیرا هر بایاس یک مقدار تکی دارد که برای هر نورون اعمال می‌شود. سپس این بردار به عنوان بایاس مقداردهی اولیه لایه برگردانده می‌شود

تکمیل تابع Forward :

```
# NOTICE: BATCH_SIZE is the first dimension of A_prev
self.input_shape = A_prev.shape
A_prev_tmp = np.copy(A_prev)
```

در این کد ماتریسی مشابه لایه نورون های ورودی ساخته میشود

```
# Check if A_prev is output of convolutional layer
if len(A_prev_tmp.shape) > ۲:
    batch_size = A_prev_tmp.shape[۰]
    A_prev_tmp = A_prev_tmp.reshape(batch_size, -۱).T
else:
    batch_size = A_prev_tmp.shape[۰]

self.reshaped_shape = A_prev_tmp.shape
```

سپس چک میکنیم که این ورودی ها از یک لایه کانولوشنی آمده اند یا خیر. اگر ورودی بیش از دو بعد داشته باشد خروجی یک لایه کانولوشنی است. علت اهمیت بررسی این موضوع این است که در شبکه‌های عصبی، لایه‌های کانولوشنی و لایه‌های پرسپترون کامل (fully connected) به طور معمول به صورت متناوب در کنار هم قرار می‌گیرند. ورودی لایه‌های پرسپترون کامل (fully connected) باید به صورت یک بردار یک بعدی باشد، در حالی که ورودی لایه‌های کانولوشنی به صورت یک آرایه چند بعدی است. بنابراین، اگر ورودی لایه فعلی یک آرایه چند بعدی باشد (مانند خروجی لایه کانولوشنی)، لازم است آن را به یک بردار یک بعدی تبدیل کرد تا به لایه پرسپترون کامل (fully connected) داده شود. به همین دلیل، در این قسمت از کد، با بررسی تعداد بعدهای ورودی، از اینکه ورودی لایه کانولوشنی باشد یا خیر، اطمینان حاصل شده و اگر ورودی یک آرایه چند بعدی باشد، آن را به یک بردار یک بعدی تبدیل می‌کند.

```
# Forward part
W, b = self.weights, self.biases
Z = np.dot(W, A_prev_tmp) + b
return Z
```

در این مرحله مرحله فوروارد شبکه fully connected پیاده سازی شده است، همانطور که مشاهده میشود Z حاصل دات شدن وزن ها و نورون های ورودی و اعمال بایاس میباشد.

تکمیل تابع backward :

```
def backward(self, dZ, A_prev):
    """
    Backward pass for fully connected layer.
        args:
            dZ: derivative of the cost with respect to the output of the
current layer
            A_prev: activations from previous layer (or input data)
        returns:
            dA_prev: derivative of the cost with respect to the activation of
the previous layer
            grads: list of gradients for the weights and bias
    """
    A_prev_tmp = np.copy(A_prev)

    # Check if A_prev is output of convolutional layer
    if len(A_prev_tmp.shape) > ۲:
        batch_size = A_prev_tmp.shape[۰]
        A_prev_tmp = A_prev_tmp.reshape(batch_size, -۱).T
    else:
        batch_size = A_prev_tmp.shape[۰]

    # Backward part
    W, b = self.weights, self.biases
    dW = np.dot(dZ, A_prev_tmp.T) / batch_size
    db = np.sum(dZ, axis=۱, keepdims=True) / batch_size
    dA_prev = np.dot(W.T, dZ)
    grads = [dW, db]

    # Reshape dA_prev to the shape of A_prev
    if len(A_prev.shape) > ۲:
        dA_prev = dA_prev.T.reshape(self.input_shape)

    return dA_prev, grads
```

در این قسمت تابع backward برای لایه fully connected پیاده‌سازی شده است. نیاز داریم که از مشتقات جزئی استفاده کنیم تا بتوانیم به دنبال وزن‌ها و بایاس‌های بهینه برای کاهش خطا باشیم.

برای این کار، ابتدا باید بررسی کنیم که آیا خروجی لایه قبل از نوع convolutional بوده است یا خیر. برای این منظور، از یک شرط if استفاده شده است. اگر خروجی قبلی کانولوشنی بود، باید ابعاد آن را به شکلی تغییر دهیم تا بتوانیم در محاسبات بعدی از آن استفاده کنیم.

سپس در بخشbackward ، باید با استفاده از مشتقات جزئی، مقدار gradient را برای وزن‌ها و بایاس‌ها محاسبه کنیم. پس از محاسبه این مقادیر، باید مشتق نسبت به تابع فعالسازی قبلی نیز محاسبه شود.

در نهایت مقدارهای محاسبه شده برای gradient و مشتق نسبت به فعالساز قبلی در قالب لیستی به صورت [dW, db] و dA_prev بازگردانده می‌شوند

۲-۱ لایه convolution دو بعدی:

تکمیل تابع وزن دهی:

مشابه وزن دهی در لایه fully connected عمل میکنیم:

```
def initialize_weights(self):
    """
    Initialize weights.
    returns:
        weights: initialized kernel with shape: (kernel_size[0],
kernel_size[1], in_channels, out_channels)
    """
    if self.initialize_method == "random":
        return np.random.randn(self.kernel_size[۰], self.kernel_size[۱],
self.in_channels, self.out_channels) * ۰,۰۱
    elif self.initialize_method == "xavier":
        xavier_stddev = np.sqrt(۲,۰ / (self.in_channels + self.out_channels))
        return np.random.randn(self.kernel_size[۰], self.kernel_size[۱],
self.in_channels,
                               self.out_channels) * xavier_stddev
    elif self.initialize_method == "he":
        he_stddev = np.sqrt(۲,۰ / self.in_channels)
        return np.random.randn(self.kernel_size[۰], self.kernel_size[۱],
self.in_channels,
                               self.out_channels) * he_stddev
    else:
        raise ValueError("Invalid initialization method")
```

تکمیل تابع تعیین بایاس:

```
def initialize_bias(self):
    """
    Initialize bias.
    returns:
        bias: initialized bias with shape: (1, 1, 1, out_channels)
    """
    if self.initialize_method == "random":
        return np.zeros((١, ١, ١, self.out_channels)) * ٠,٠١
    if self.initialize_method == "xavier":
        return np.zeros((١, ١, ١, self.out_channels)) * np.sqrt(١ /
self.out_channels)
    if self.initialize_method == "he":
        return np.zeros((١, ١, ١, self.out_channels)) * np.sqrt(٢ /
self.out_channels)
    else:
        raise ValueError("Invalid initialization method")
```

این تابع با در نظر گرفتن روش مقدار دهی اولیه مشخص شده در متغیر initialize_method، bias را از طریق ضرب ماتریس صفر و یا یک به شکل تعریف شده در shape آن یعنی ۱و ۱، ۱، (out_channels) و مقدار مشخص شده در هر یک از روش‌های initialize_method (یعنی random، xavier و he) محاسبه می‌کند و آن‌ها را بازگردانده و در متغیر bias در instance لایه Convolution قرار می‌دهد. در صورتی که متغیر initialize_method با هیچکدام از مقادیر ""random، xavier"" و "he" برابر نباشد، یک خطا برگردانده می‌شود. علت وابستگی بایاس به متد وزن دهی این است که در لایه های Fully Connected، ورودی ها به صورت بردار های یک بعدی و نهایتاً به یک نورون خروجی متصل می شوند. بنابراین هر نورون به یک بایاس نیاز دارد که به عنوان عامل تفاوت در سطح نورون ها عمل کند. بنابراین، بایاس در لایه های Fully Connected برای هر نورون باید ثابت باشد. اما در لایه های Convolutional، وزن ها به عنوان یک فیلتر برای انجام عملیات کانولوشن در سطح ورودی استفاده می شوند. در واقع، برای هر فیلتر یک بایاس وجود دارد که به صورت همزمان با کانولوشن محاسبه می شود. بنابراین، در لایه های Convolutional، بایاس برای هر فیلتر متفاوت است.

تکمیل تابع Target Shape :

```python
def target_shape(self, input_shape):
    """
    Calculate the shape of the output of the convolutional layer.
    args:
        input_shape: shape of the input to the convolutional layer
    returns:
        target_shape: shape of the output of the convolutional layer
    """
    # Assuming 'same' padding and stride of 1
    H = input_shape[0]
    W = input_shape[1]
    num_filters = self.num_filters
    kernel_size = self.kernel_size
    target_shape = (H, W, num_filters)
    return target_shape
```

این کد یک تابع به نام `target_shape` را پیاده‌سازی می‌کند که با دریافت شکل ورودی به لایه کانولوشن، شکل خروجی این لایه را محاسبه می‌کند. شکل خروجی لایه کانولوشن به چند عامل بستگی دارد، از جمله شکل ورودی، اندازه کرنل کانولوشن، تعداد فیلترها و پارامترهای پدینگ و استراید.

در این کد، محاسبه شکل خروجی با فرض پدینگ `same` و استراید ۱ انجام می‌شود. همچنین، فرض شده است که لایه کانولوشن شامل `num_filters` فیلتر و کرنل مربعی با اندازه `kernel_size` است. با گرفتن شکل ورودی `(H، W، C)`، جایی که `H` و `W` ارتفاع و عرض ورودی را نشان می‌دهند و `C` تعداد کانال‌های ورودی است، می‌توان شکل خروجی را محاسبه کرد.

با توجه به فرض پدینگ `same` و استراید ۱، ارتفاع و عرض خروجی با ارتفاع و عرض ورودی برابر هستند و تعداد کانال‌های خروجی با تعداد فیلترها برابر است. بنابراین، شکل خروجی به صورت یک تاپل `(H، W، num_filters)` نمایش داده می‌شود.

تکمیل تابعsingle_step_convolve :

```
def single_step_convolve(self, a_slic_prev, W, b):
    """
    Convolve a slice of the input with the kernel.
    args:
        a_slic_prev: slice of the input data
        W: kernel
        b: bias
    returns:
        Z: convolved value
    """
    # Element-wise multiplication
    s = np.multiply(a_slic_prev, W)

    # Sum over all elements
    Z = np.sum(s)

    # Add bias as type float using np.float()
    Z = np.float۳۲(Z + b)

    return Z
```

این تابع برای اعمال یک گام کانولوشن بر روی یک برش از داده ورودی وزن شده با استفاده از یک ناحیه از نواحی کرنل و بایاس است. مقدار ضرب وزن شده در داده ورودی با استفاده از عملگر ضرب انجام شده و سپس این مقدارها با هم جمع شده و به آن بایاس اضافه می‌شود تا خروجی نهایی بدست آید

تکمیل تابع Forward :

```
# Get the kernel and bias parameters
W, b = self.get_params()
# Get the shape of the previous layer activations
(batch_size, H_prev, W_prev, C_prev) = A_prev.shape
# Get the shape of the kernel
(kernel_size_h, kernel_size_w, C_prev, C) = W.shape
# Get the stride
stride_h, stride_w = self.stride
# Get the padding
padding_h, padding_w = self.padding
# Calculate the output shape
H = int((H_prev + ۲ * padding_h - kernel_size_h) / stride_h) + ۱
W = int((W_prev + ۲ * padding_w - kernel_size_w) / stride_w) + ۱
# Initialize the output activations
Z = np.zeros((batch_size, H, W, C))
# Pad the input activations
A_prev_pad = self.pad(A_prev, self.padding)
# Perform convolution
for i in range(batch_size):
    for h in range(H):
        h_start = h * stride_h
        h_end = h_start + kernel_size_h
        for w in range(W):
            w_start = w * stride_w
            w_end = w_start + kernel_size_w
            for c in range(C):
                a_slice_prev = A_prev_pad[i, h_start:h_end, w_start:w_end, :]
                Z[i, h, w, c] = self.single_step_convolve(a_slice_prev,
W[..., c], b[..., c])
return Z
```

در این کد، یک لایه کانولوشن پیاده‌سازی شده است که برای فعالسازی از تابع سیگموید استفاده می‌کند. در تابع `forward`، با دریافت `A_prev` که ورودی یا خروجی لایه قبلی است، پیاده‌سازی پردازش پیشروی برای لایه کانولوشن انجام می‌شود.

در ابتدا، مقادیر `W` و `b` به صورت `None` تعریف شده‌اند. همچنین مقدار ``batch_size`، `H_prev`، `W_prev`` و `C_prev` با استفاده از شکل `A_prev` محاسبه شده‌اند. همچنین شکل ``W`، `kernel_size_h`، `kernel_size_w`، `C_prev`` و `C` نیز مشخص شده‌اند. سپس مقادیر ``stride_h`، `stride_w`، `padding_h`` و `padding_w` تعیین شده و اندازه ماتریس خروجی با توجه به مقادیر ورودی و هایپرپارامترهای تعیین شده محاسبه شده و در `H` و `W` ذخیره شده است. در ادامه، با استفاده از تابع `pad`، ماتریس ورودی `A_prev` پدینگ شده و در `A_prev_pad` ذخیره شده است.

سپس با استفاده از چهار حلقه `for`، کرنل بر روی ورودی اعمال شده و مقادیر ماتریس خروجی `Z` محاسبه شده است. در هر مرحله، یک سلایس از ورودی با `a_slice_prev` مشخص شده و با استفاده از تابع

`single_step_convolve`، محاسبات لایه انجام شده است. در نهایت، ماتریس `Z` به عنوان خروجی لایه بازگردانده شده است.

تکمیل تابع Backward :

```
def backward(self, dZ, A_prev):
    """
    Backward pass for convolutional layer.
    args:
        dZ: gradient of the cost with respect to the output of the
convolutional layer
        A_prev: activations from previous layer (or input data)
        A_prev.shape = (batch_size, H_prev, W_prev, C_prev)
    returns:
        dA_prev: gradient of the cost with respect to the input of the
convolutional layer
        gradients: list of gradients with respect to the weights and bias
    """
    # Extract parameters
    W, b = self.params
    (batch_size, H_prev, W_prev, C_prev) = A_prev.shape
    (kernel_size_h, kernel_size_w, C_prev, C) = W.shape
    stride_h, stride_w = self.stride
    padding_h, padding_w = self.padding

    # Initialize gradients
    dA_prev = np.zeros_like(A_prev)
    dW = np.zeros_like(W)
    db = np.zeros_like(b)

    # Pad A_prev
    A_prev_pad = self.pad(A_prev, padding_h, padding_w)

    # Pad dA_prev
    dA_prev_pad = self.pad(dA_prev, padding_h, padding_w)

    # Loop over the batch
    for i in range(batch_size):
        a_prev_pad = A_prev_pad[i]
        da_prev_pad = dA_prev_pad[i]

        # Loop over vertical axis of the output volume
        for h in range(self.output_h):
            # Vertical start and end of the current slice
            h_start = h * stride_h
            h_end = h_start + kernel_size_h

            # Loop over horizontal axis of the output volume
            for w in range(self.output_w):
                # Horizontal start and end of the current slice
                w_start = w * stride_w
                w_end = w_start + kernel_size_w
```

```
                # Loop over the channels
                for c in range(C):
                    # Slice A_prev_pad
                    a_slice = a_prev_pad[h_start:h_end, w_start:w_end, :]

                    # Update gradients
                    da_prev_pad[h_start:h_end, w_start:w_end, :] +=
np.multiply(dZ[i, h, w, c], W[..., c])
                    dW[..., c] += np.multiply(dZ[i, h, w, c], a_slice)
                    db[..., c] += dZ[i, h, w, c]

        # Set the ith example's dA_prev to the unpadded da_prev_pad
        dA_prev[i, :, :, :] = da_prev_pad[padding_h:-padding_h, padding_w:-
padding_w, :]

    # Package gradients
    grads = [dW, db]

    return dA_prev, grads

٤
```

در این تابع گرادیان هزینه نسبت به خروجی لایه کانولوشنی و ورودی قبلی (یا داده ورودی) محاسبه می‌شود.

ورودی‌های تابع عبارتند از:

- dZ: گرادیان هزینه نسبت به خروجی لایه کانولوشنی
- A_prev: فعال‌سازی‌ها از لایه قبلی (یا داده ورودی)

خروجی‌های تابع شامل:

- dA_prev: گرادیان هزینه نسبت به ورودی لایه کانولوشنی
- gradients: لیستی از گرادیان‌ها نسبت به وزن‌ها و بایاس

در این تابع ابتدا پارامترهای لایه مانند وزن‌ها و بایاس استخراج می‌شوند. سپس ابعاد ورودی قبلی و وزن‌ها محاسبه می‌شوند. در ادامه پارامترهایی مانند قدم، پدینگ و گرادیان‌ها مقداردهی اولیه می‌شوند.

پس از آن، ورودی قبلی با استفاده از پدینگ، گسترش داده می‌شود. همچنین گرادیان نسبت به ورودی قبلی نیز با استفاده از پدینگ گسترش داده می‌شود. سپس برای هر داده در دسته، ورودی قبلی گسترده و گرادیان نسبت به ورودی قبلی گسترده بازیابی می‌شوند.

سپس برای هر بخشی از ورودی قبلی گسترده، برشی از آن با اندازه مشخص برای کرنل لایه کانولوشنی انجام می‌شود. سپس با استفاده از برش‌ها و وزن‌ها، گرادیان‌ها محاسبه می‌شوند. در نهایت، گرادیان‌ها برای هر داده در دسته جمع‌آوری می‌شوند و به عنوان خروجی تابع بازگردانده می‌شوند.

۳-۱ لایه Pooling دو بعدی:

پیاده سازی تابع Target Shape:

```
def target_shape(self, input_shape):
    """
    Calculate the shape of the output of the layer.
        args:
            input_shape: shape of the input
        returns:
            output_shape: shape of the output
    """
    H = (input_shape[۰] - self.kernel_size[۰]) // self.stride[۰] + ۱
    W = (input_shape[۱] - self.kernel_size[۱]) // self.stride[۱] + ۱
    return H, W
```

این تابع برای محاسبه‌ی ابعاد خروجی از لایه‌ی Max Pooling استفاده می‌شود. در ورودی، ابعاد ورودی به لایه‌ی Max Pooling به صورت یک tuple از شکل (batch_size, H, W, C) قرار می‌گیرد که بیانگر ابعاد داده‌های ورودی به شبکه هستند. سپس ابعاد خروجی به صورت یک tuple شامل ارتفاع و عرض ویژگی‌های خروجی محاسبه می‌شود.

برای این کار، ابتدا ابعاد ورودی به لایه‌ی Max Pooling و سپس ابعاد کرنل Max Pooling و Stride و Mode (Max یا Average) دریافت می‌شود. سپس ابعاد خروجی با توجه به فرمول زیر محاسبه می‌شود:

H = (H_prev - pool_height) / stride + ۱

W = (W_prev - pool_width) / stride + ۱

که در آن H_prev و W_prev ارتفاع و عرض ویژگی‌های ورودی به لایه‌ی Max Pooling هستند، pool_height و pool_width ابعاد کرنل Max Pooling هستند و stride اندازه گام موقع پرش کرنل روی ورودی است. سپس ابعاد خروجی به صورت یک tuple شامل ارتفاع و عرض ویژگی‌های خروجی بازگردانده می‌شود.

پیاده سازی تابع Forward :

```
def forward(self, A_prev):
    """
    Forward pass for max pooling layer.
        args:
            A_prev: activations from previous layer (or input data)
        returns:
            A: output of the max pooling layer
    """
    # Get dimensions of input
    (batch_size, H_prev, W_prev, C_prev) = A_prev.shape
    # Get dimensions of filter
    (f_h, f_w) = self.kernel_size
    # Get stride values
    strideh, stridew = self.stride
    # Compute output dimensions
    H = int((H_prev - f_h) / strideh) + 1
    W = int((W_prev - f_w) / stridew) + 1
    # Initialize output with zeros
    A = np.zeros((batch_size, H, W, C_prev))
    # Loop over each training example
    for i in range(batch_size):
        # Loop over vertical axis of output volume
        for h in range(H):
            h_start = h * strideh
            h_end = h_start + f_h
            # Loop over horizontal axis of output volume
            for w in range(W):
                w_start = w * stridew
                w_end = w_start + f_w
                # Loop over channels of output volume
                for c in range(C_prev):
                    # Slice input for current filter
                    a_prev_slice = A_prev[i, h_start:h_end, w_start:w_end, c]
                    if self.mode == "max":
                        # Compute max value for filter
                        A[i, h, w, c] = np.max(a_prev_slice)
                    elif self.mode == "average":
                        # Compute average value for filter
                        A[i, h, w, c] = np.mean(a_prev_slice)
                    else:
                        raise ValueError("Invalid mode")

    return A
```

این تابع برای پیاده‌سازی فرآیند forward pass در لایه Max Pooling به‌کار می‌رود. در این تابع، با دریافت ورودی A_prev که برابر با فعال‌سازی لایه قبلی (یا داده‌های ورودی) است، عملیات max pooling را انجام می‌دهیم. برای این کار، ابتدا اندازه و شکل ورودی را بررسی می‌کنیم و سپس با استفاده از اندازه کرنل و اندازه ورودی، شکل و خروجی لایه را محاسبه می‌کنیم. سپس با استفاده از حلقه‌های تو در تو، برای هر سطر و ستون از ورودی، بخشی از ورودی که به اندازه کرنل با آن هم‌پوشانی دارد را انتخاب می‌کنیم و سپس با استفاده از

حالت max یا average مقدار خروجی را محاسبه می‌کنیم و در آرایه خروجی A قرار می‌دهیم. در نهایت، آرایه خروجی را به عنوان خروجی لایه برمی‌گردانیم.

پیاده سازی تابع create_mask_from_window :

```
def create_mask_from_window(self, x):
    """
    Create a mask from an input matrix x, to identify the max entry of x.
        args:
            x: numpy array
        returns:
            mask: numpy array of the same shape as window, contains a True at
the position corresponding to the max entry of x.
    """
    mask = x == np.max(x)
    return mask
```

این تابع به ازای یک ورودی `x`، یک ماسک با همان ابعاد ورودی تولید می‌کند که در آن فقط در محلی که بیشترین مقدار در ورودی `x` وجود دارد، مقدار `True` قرار داده می‌شود و در سایر جاهای ماتریس مقدار `False` قرار می‌گیرد. در واقع مقدار `True` در جایی که بیشترین مقدار در `x` وجود دارد، به عنوان ماسک برای تشخیص آن استفاده می‌شود.

پیاده سازی تابع distribute_value:

```
def distribute_value(self, dz, shape):
    """
    Distribute the input value in the matrix of dimension shape.
        args:
            dz: input scalar
            shape: the shape (n_H, n_W) of the output matrix for which we
want to distribute the value of dz
        returns:
            a: distributed value
    """
    # Implement distribute_value
    (n_H, n_W) = shape
    average = dz / (n_H * n_W)
    a = np.ones(shape) * average
    return a
```

این تابع برای توزیع یک مقدار ورودی به صورت یکنواخت بر روی یک ماتریس به شکل دلخواه استفاده می‌شود. ورودی دو پارامتر dz و shape دارد که dz یک مقدار عددی است و shape یک tuple از دو مقدار n_H و n_W است. با استفاده از این دو مقدار، ماتریسی به ابعاد n_H در n_W ایجاد می‌شود و مقدار dz به صورت یکنواخت بر روی تمامی عناصر ماتریس توزیع می‌شود. سپس ماتریس نهایی به عنوان خروجی تابع بازگردانده می‌شود.

پیاده سازی تابع Backward :

```
def backward(self, dZ, A_prev):
    """
    Backward pass for max pooling layer.
        args:
            dA: gradient of cost with respect to the output of the max
pooling layer
            A_prev: activations from previous layer (or input data)
        returns:
            dA_prev: gradient of cost with respect to the input of the max
pooling layer
    """
    # Implement backward pass for max pooling layer
    (f_h, f_w) = self.kernel_size
    strideh, stridew = self.stride
    batch_size, H_prev, W_prev, C_prev = A_prev.shape
    batch_size, H, W, C = dZ.shape
    dA_prev = np.zeros((batch_size, H_prev, W_prev, C_prev))
    for i in range(batch_size):
        for h in range(H):
            for w in range(W):
                for c in range(C):
                    h_start = h * strideh
```

```
                    h_end = h_start + f_h
                    w_start = w * stridew
                    w_end = w_start + f_w
                    if self.mode == "max":
                        a_prev_slice = A_prev[i, h_start:h_end,
w_start:w_end, c]
                        mask = self.create_mask_from_window(a_prev_slice)
                        dA_prev[i, h_start:h_end, w_start:w_end, c] +=
np.multiply(mask, dZ[i, h, w, c])
                    elif self.mode == "average":
                        dz = dZ[i, h, w, c]
                        dA_prev[i, h_start:h_end, w_start:w_end, c] +=
self.distribute_value(dz, (f_h, f_w))
                    else:
                        raise ValueError("Invalid mode")
    # Don't change the return
    return dA_prev, None
```

این کد، تابع backward برای لایه max pooling را پیاده سازی می‌کند. این تابع با استفاده از گرادیان هزینه نسبت به خروجی لایه max pooling و فعال‌سازی‌های لایه قبلی (یا داده ورودی)، گرادیان هزینه نسبت به ورودی لایه max pooling را محاسبه می‌کند. در این تابع، ابتدا مقادیر مورد نیاز برای استفاده در عملیات backward از جمله اندازه هسته، اندازه گام، اندازه دسته، اندازه فضا و حالت max یا average استخراج می‌شوند.

سپس، یک آرایه صفر برای مقدار dA_prev ایجاد می‌شود و در یک حلقه چهارگانه از اندیس‌های بچ، ارتفاع، پهنا و عمق لایه max pooling استفاده می‌شود. در هر مرحله، اندیس‌های مربوط به فضای ورودی max pooling با استفاده از اندازه هسته و گام محاسبه می‌شوند. سپس، در حالت max، شرایط محاسبه‌ی برش و ماسک از ماتریس فعال‌سازی‌های لایه قبلی با استفاده از برش هسته تعیین می‌شود و مقدار dA_prev با استفاده از ضرب ماتریسی با ماسک و مقدار dZ محاسبه می‌شود. در حالت average، مقدار dz محاسبه می‌شود و با استفاده از توزیع این مقدار در محدوده‌ی برش، مقدار dA_prev محاسبه می‌شود.

در نهایت، گرادیان هزینه نسبت به ورودی لایه max pooling (dA_prev) به همراه None (به خاطر استفاده در شبکه‌های پیچشی چند لایه‌ای) برگشت داده می‌شود.

بخش دوم – توابع فعالسازی

- **تابع زیگموید**

```
-   class Sigmoid(Activation):
        def forward(self, Z: np.ndarray) -> np.ndarray:
            """
            Sigmoid activation function.
                args:
                x: input to the activation function
                returns:
                    sigmoid(x)
            """
            A = 1 / (1 + np.exp(-Z))
            return A

        def backward(self, dA: np.ndarray, Z: np.ndarray) -> np.ndarray:
            """
            Backward pass for sigmoid activation function.
                args:
                    dA: derivative of the cost with respect to the
    activation
                    Z: input to the activation function
                returns:
                    derivative of the cost with respect to Z
            """
            A = self.forward(Z)
            dZ = dA * A * (1 - A)
            return dZ
```

کد شامل دو تابع forward و backward است که به ترتیب برای انجام فرایند فعال‌سازی و محاسبه گرادیان در لایه زیگموید استفاده می‌شوند. در تابع forward، با گرفتن ورودی Z از کاربر، ابتدا این تابع مقدار خروجی را با استفاده از تابع زیگموید محاسبه کرده و سپس آن را برمی‌گرداند. در تابع backward نیز، با گرفتن گرادیان هزینه نسبت به خروجی لایه (dA) و ورودی لایه (Z)، ابتدا خروجی لایه زیگموید (A) را با استفاده از تابع forward محاسبه می‌کند، سپس گرادیان هزینه نسبت به ورودی لایه (dZ) را با استفاده از زنجیره‌ای‌سازی مشتقات محاسبه می‌کند و آن را برمی‌گرداند.

- تابع رلو

```
-   class ReLU(Activation):
        def forward(self, Z: np.ndarray) -> np.ndarray:
            """
            ReLU activation function.
                args:
                    x: input to the activation function
                returns:
                    relu(x)
            """
            A = np.maximum(0, Z)
            return A

        def backward(self, dA: np.ndarray, Z: np.ndarray) -> np.ndarray:
            """
            Backward pass for ReLU activation function.
                args:
                    dA: derivative of the cost with respect to the
    activation
                    Z: input to the activation function
                returns:
                    derivative of the cost with respect to Z
            """
            dZ = np.array(dA, copy=True)
            dZ[Z <= 0] = 0
            return dZ
```

در متد forward، با گرفتن ورودی Z، تابع فعال‌سازی ReLU را اجرا می‌کند و خروجی آن را A برمی‌گرداند.

در متد backward، با گرفتن مشتق کراسیون (dA) و ورودی Z، گرادیان تابع فعال‌سازی ReLU را نسبت به Z محاسبه می‌کند و خروجی آن را به صورت dZ برمی‌گرداند. این کد ابتدا dZ را برابر مقدار dA قرار می‌دهد و سپس برای مقادیری که در Z کمتر از صفر هستند، مقدار آنها را به صفر تغییر می‌دهد.

- تابع tanh

```
-   class Tanh(Activation):
        def forward(self, Z: np.ndarray) -> np.ndarray:
            """
            Tanh activation function.
                args:
                    x: input to the activation function
                returns:
                    tanh(x)
            """
            A = np.tanh(Z)
            return A

        def backward(self, dA: np.ndarray, Z: np.ndarray) -> np.ndarray:
            """
            Backward pass for tanh activation function.
                args:
                    dA: derivative of the cost with respect to the
    activation
                    Z: input to the activation function
                returns:
                    derivative of the cost with respect to Z
            """
            A = self.forward(Z)
            dZ = dA * (\ - np.square(A))
```

در متد forward، تابع تانژانت هایپربولیک (tanh) به عنوان تابع فعال‌سازی پیاده‌سازی شده است. ورودی این تابع یک ماتریس است که نشان دهنده‌ی خروجی لایه‌ی قبلی یا داده‌های ورودی است. خروجی این تابع یک ماتریس با همان ابعاد ورودی است.

در متد backward، ابتدا با استفاده از متد forward، خروجی لایه‌ی قبلی محاسبه می‌شود. سپس با استفاده از زنجیره‌قاعده، گرادیان لایه‌ی فعال‌سازی با توجه به گرادیان خروجی شبکه محاسبه می‌شود. خروجی این متد نیز یک ماتریس با همان ابعاد ورودی است.

- تابع LinearActivation:

```
class LinearActivation(Activation):
    def linear(Z: np.ndarray) -> np.ndarray:
        """
        Linear activation function.
            args:
                x: input to the activation function
            returns:
                x
        """
        A = Z
        return A

    def backward(dA: np.ndarray, Z: np.ndarray) -> np.ndarray:
        """
        Backward pass for linear activation function.
            args:
                dA: derivative of the cost with respect to the
activation
                Z: input to the activation function
            returns:
                derivative of the cost with respect to Z
        """
        dZ = dA
        return dZ
```

تابع فعال‌سازی خطی به شکل زیر تعریف می‌شود:

$$f(x) = x$$

تابع forward برای این تابع فعال‌سازی، ماتریس ورودی را دریافت کرده و بدون تغییر برروی آن اعمال می‌کند.

تابع backward نیز مشتقات نسبت به ورودی خود را دریافت کرده و با توجه به قاعده زنجیره‌ای، مشتقات نسبت به ورودی را به دست می‌آورد. با توجه به تابع فعال‌سازی خطی، مشتق آن برابر یک است. بنابراین مشتق نسبت به ورودی برابر خواهد بود با مشتق نسبت به خروجی که از آن به عنوان ورودی استفاده شده است.

بخش سوم – بهینه سازها:

- **گرادیان کاهشی:**

```
class GD:
    def __init__(self, layers_list: dict, learning_rate: float):
        """
        Gradient Descent optimizer.
            args:
                layers_list: dictionary of layers name and layer object
                learning_rate: learning rate
        """
        self.learning_rate = learning_rate
        self.layers = layers_list

    def update(self, grads, name):
        """
        Update the parameters of the layer.
            args:
                grads: list of gradients for the weights and bias
                name: name of the layer
            returns:
                params: list of updated parameters
        """
        layer = self.layers[name]
        params = []
        for i in range(len(grads)):
            params.append(layer.parameters[i] - self.learning_rate *
grads[i])
        return params
```

متد `update` در کلاس `GD` برای به‌روزرسانی پارامترهای لایه‌ها با استفاده از روش Gradient Descent استفاده می‌شود. در این متد، ورودی‌های `grads` و `name` دریافت می‌شوند `grads`. یک لیست از گرادیان‌ها است که شامل گرادیان برای وزن‌ها و بایاس مربوط به لایه مورد نظر است `name`. نام لایه‌ای است که می‌خواهیم پارامترهای آن را به‌روزرسانی کنیم.

ابتدا، با استفاده از نام لایهname آن لایه را از دیکشنری لایه‌ها (`layers`) بازیابی می‌کنیم و در متغیر `layer`قرار می‌دهیم. سپس، یک لیست خالی به نام `params` تعریف می‌کنیم که در آن پارامترهای به‌روزرسانی شده را ذخیره خواهیم کرد. در حلقه`for` ، برای هر یک از اجزای لیست`grads` ، که شامل گرادیان وزن‌ها و بایاس است، مقدار به‌روزرسانی شده آن را محاسبه می‌کنیم. برای هر گرادیان، از مقدار قبلی مربوط به آن پارامتر(`layer.parameters[i]`) ، مقدار نرخ یادگیری (`self.learning_rate`) را ضرب می‌کنیم و از آن مقدار کم می‌کنیم. سپس مقدار به‌روزرسانی شده را به لیست `params` اضافه می‌کنیم.

در نهایت، لیست `params` حاوی پارامترهای به‌روزرسانی شده را به عنوان خروجی از متد `update` برمی‌گردانیم.

- آدام

```
- import numpy as np

  class Adam:
      def __init__(self, layers_list, learning_rate=۰,۰۰۱, beta۱=۰,۹,
  beta۲=۰,۹۹۹, epsilon=۱e-۸):
          self.layers = layers_list
          self.learning_rate = learning_rate
          self.beta۱ = beta۱
          self.beta۲ = beta۲
          self.epsilon = epsilon
          self.V = {}
          self.S = {}
          for name in layers_list:
              v = [np.zeros_like(p) for p in
  layers_list[name].parameters]
              s = [np.zeros_like(p) for p in
  layers_list[name].parameters]
              self.V[name] = v
              self.S[name] = s

      def update(self, grads, name, epoch):
          layer = self.layers[name]
          params = []
          for i in range(len(grads)):
              self.V[name][i] = self.beta۱ * self.V[name][i] + (۱ -
  self.beta۱) * grads[i]
              self.S[name][i] = self.beta۲ * self.S[name][i] + (۱ -
  self.beta۲) * np.square(grads[i])
              V_corrected = self.V[name][i] / (۱ - np.power(self.beta۱,
  epoch))
              S_corrected = self.S[name][i] / (۱ - np.power(self.beta۲,
  epoch))
              params.append(
                  layer.parameters[i] - self.learning_rate * V_corrected
  / (np.sqrt(S_corrected) + self.epsilon))
          return params
```

الگوریتم Adam یک روش بهینه‌سازی است که بر پایه ترکیبی از میانگین متحرک اول (Momentum) و مربع میانگین متحرک دوم (RMSprop) استوار است. الگوریتم Adam معمولاً برای بهینه‌سازی مدل‌های یادگیری عمیق استفاده می‌شود.

- در تابع `__init__`، متغیرهای مربوط به الگوریتم Adam و لایه‌های شبکه را مقداردهی اولیه می‌کنیم. برای هر لایه، متغیرهای `V` و `S` را با ابعاد مشابه پارامترهای لایه ایجاد می‌کنیم و آنها را با مقدار صفر مقداردهی اولیه می‌کنیم.

- تابع `update` برای به‌روزرسانی پارامترهای لایه با استفاده از الگوریتم Adam استفاده می‌شود. در این تابع، مقادیر `V` و `S` را با توجه به گرادیان‌های ورودی به روز می‌کنیم. سپس مقادیر اصلاح شده `V_corrected` و `S_corrected` را محاسبه می‌کنیم با توجه به عدد شماره تکرار (epoch) و پارامترهای بتا (beta). در نهایت، پارامترهای لایه را با استفاده از این مقادیر اصلاح شده به‌روزرسانی می‌کنیم و آنها را در لیست `params` ذخیره می‌کنیم.

الگوریتم Adam با استفاده از این روش بهینه‌سازی، میزان یادگیری (learning rate) را برای هر پارامتر به‌صورت جداگانه تطبیق می‌دهد و می‌تواند به سرعت و کارایی بیشتری در بهینه‌سازی مدل‌های یادگیری عمیق منجر شود.

بخش چهارم – توابع هزینه:

- کراس آنتروپی

```python
import numpy as np

class BinaryCrossEntropy:
    def __init__(self) -> None:
        pass

    def compute(self, y_hat: np.ndarray, y: np.ndarray) -> float:
        """
        Computes the binary cross entropy loss.
            args:
                y: true labels (n_classes, batch_size)
                y_hat: predicted labels (n_classes, batch_size)
            returns:
                binary cross entropy loss
        """
        batch_size = y.shape[١]
        cost = -np.sum(y * np.log(y_hat + ١e-١٠) + (١ - y) * np.log(١ -
y_hat + ١e-١٠)) / batch_size
        return np.squeeze(cost)

    def backward(self, y_hat: np.ndarray, y: np.ndarray) -> np.ndarray:
        """
        Computes the derivative of the binary cross entropy loss.
            args:
                y: true labels (n_classes, batch_size)
                y_hat: predicted labels (n_classes, batch_size)
            returns:
                derivative of the binary cross entropy loss
        """
        return np.divide(y_hat - y, (y_hat * (١ - y_hat)) + ١e-١٠)
```

این کد تابع `compute` را پیاده‌سازی می‌کند تا تابع خطا کراس انتروپی دوتایی را محاسبه کند. در این تابع، ابتدا اندازه بچ‌سایز (تعداد نمونه‌ها در هر بچ) را به صورت `batch_size` مشخص می‌کند. سپس تابع خطا را با استفاده از فرمول کراس انتروپی دوتایی محاسبه می‌کند. نتیجه به صورت یک عدد اسکالر خروجی داده می‌شود.

- **کمترین مربعات خطا**

```python
import numpy as np


class MeanSquaredError:
    def __init__(self):
        pass

    def compute(self, y_pred, y_true):
        """
        computes the mean squared error loss
            args:
                y_pred: predicted labels (n_classes, batch_size)
                y_true: true labels (n_classes, batch_size)
            returns:
                mean squared error loss
        """
        batch_size = y_pred.shape[1]
        cost = np.sum(np.square(y_pred - y_true)) / (2 * batch_size)
        return np.squeeze(cost)

    def backward(self, y_pred, y_true):
        """
        computes the derivative of the mean squared error loss
            args:
                y_pred: predicted labels (n_classes, batch_size)
                y_true: true labels (n_classes, batch_size)
            returns:
                derivative of the mean squared error loss
        """
        return y_pred - y_true
```

این کلاس شامل دو تابع compute و backward است. تابع compute برای محاسبه خطای میانگین مربعات استفاده می‌شود. این تابع ابتدا اندازه بچ‌سایز (تعداد نمونه‌ها در هر بچ) را به صورت batch_size محاسبه کرده و سپس خطای میانگین مربعات را با استفاده از فرمول معادله مربعات محاسبه می‌کند.

تابع backward نیز مشتق خطای میانگین مربعات را نسبت به پیش‌بینی‌ها (y_pred) و برچسب‌های واقعی (y_true) محاسبه می‌کند و مقدار مشتق را برمی‌گرداند.

بخش پنجم – مدل:

```
class Model:
    def __init__(self, arch, criterion, optimizer, name=None):
        """
        Initialize the model.
        args:
            arch: dictionary containing the architecture of the model
            criterion: loss
            optimizer: optimizer
            name: name of the model
        """
        if name is None:
            self.model = arch
            self.criterion = criterion
            self.optimizer = optimizer
            self.layers_names = list(arch.keys())
        else:
            self.model, self.criterion, self.optimizer, self.layers_names =
self.load_model(name)

    def is_layer(self, layer):
        """
        Check if the layer is a layer.
        args:
            layer: layer to be checked
        returns:
            True if the layer is a layer, False otherwise
        """
        return isinstance(layer, (Conv۲D, MaxPool۲D, FC))

    def is_activation(self, layer):
        """
        Check if the layer is an activation function.
        args:
            layer: layer to be checked
        returns:
            True if the layer is an activation function, False otherwise
        """
        return isinstance(layer, Activation)

    def forward(self, x):
        """
        Forward pass through the model.
        args:
            x: input to the model
        returns:
            output of the model
        """
        tmp = []
        A = x
        for l in range(len(self.layers_names)):
            Z = self.model[self.layers_names[l]].forward(A)
            tmp.append(Z.copy())
            A = self.model[self.layers_names[l]].activation.forward(Z)
            tmp.append(A.copy())
```

```
        return tmp

    def backward(self, dAL, tmp, x):
        """
        Backward pass through the model.
        args:
            dAL: derivative of the cost with respect to the output of the
model
            tmp: list containing the intermediate values of Z and A
            x: input to the model
        returns:
            gradients of the model
        """
        dA = dAL
        grads = {}
        for l in reversed(range(len(self.layers_names))):
            if l > ١:
                Z, A = tmp[l * ٢ - ١], tmp[l * ٢ - ٢]
            else:
                Z, A = tmp[l * ٢ - ١], x
            dZ = dA * self.model[self.layers_names[l]].activation.backward(Z)
            dA, grad = self.model[self.layers_names[l]].backward(dZ, A)
            grads[self.layers_names[l]] = grad
        return grads

    def update(self, grads):
        """
        Update the model.
        args:
            grads: gradients of the model
        """
        for layer_name in self.layers_names:
            if self.is_layer(self.model[layer_name]) and not
isinstance(self.model[layer_name], MaxPool٢D):
                self.model[layer_name].update(grads[layer_name])

    def one_epoch(self, x, y):
        """
        One epoch of training.
        args:
            x: input to the model
            y: labels
            batch_size: batch size
        returns:
            loss
        """
        tmp = self.forward(x)
        AL = tmp[-١]
        loss = self.criterion.compute(AL, y)
        dAL = self.criterion.backward(AL, y)
        grads = self.backward(dAL, tmp, x)
        self.update(grads)
        return loss

    def save(self, name):
        """
        Save the model.
```

```
        args:
            name: name of the model
        """
        with open(name, 'wb') as f:
            pickle.dump((self.model, self.criterion, self.optimizer,
self.layers_names), f)

    def load_model(self, name):
        """
        Load the model.
        args:
            name: name of the model
        returns:
            model, criterion, optimizer, layers_names
        """
        with open(name, 'rb') as f:
            return pickle.load(f)

    def shuffle(self, m, shuffling):
        order = list(range(m))
        if shuffling:
            np.random.shuffle(order)
        return order

    def batch(self, X, y, batch_size, index, order):
        """
        Get a batch of data.
        args:
            X: input to the model
            y: labels
            batch_size: batch size
            index: index of the batch
                e.g: if batch_size = 3 and index = 1 then the batch will be
from index [3, 4, 5]
            order: order of the data
        returns:
            bx, by: batch of data
        """
        last_index = min(index + batch_size, len(order))
        batch = order[index:last_index]
        if X.ndim == ٤:
            bx = X[batch]
            by = y[batch]
            return bx, by
        else:
            bx = np.expand_dims(X[:, batch], axis=٠)
            by = np.expand_dims(y[:, batch], axis=٠)
            return bx, by

    def compute_loss(self, X, y, batch_size):
        """
        Compute the loss.
        args:
            X: input to the model
            y: labels
            Batch_Size: batch size
        returns:
```

```
            loss
        """
        m = X.shape[0] if X.ndim == 2 else X.shape[1]
        order = self.shuffle(m, False)
        cost = 0
        for b in range(m // batch_size):
            bx, by = self.batch(X, y, batch_size, b * batch_size, order)
            tmp = self.forward(bx)
            AL = tmp[-1]
            cost += self.criterion.compute(AL, by)
        return cost

    def train(self, X, y, epochs, val=None, batch_size=32, shuffling=False,
verbose=1, save_after=None):
        """
        Train the model.
        args:
            X: input to the model
            y: labels
            epochs: number of epochs
            val: validation data
            batch_size: batch size
            shuffling: if True shuffle the data
            verbose: if 1 print the loss after each epoch
            save_after: save the model after training
        """
        train_cost = []
        val_cost = []
        m = X.shape[0] if X.ndim == 2 else X.shape[1]
        for e in tqdm(range(1, epochs + 1)):
            order = self.shuffle(m, shuffling)
            cost = 0
            for b in range(m // batch_size):
                bx, by = self.batch(X, y, batch_size, b * batch_size, order)
                cost += self.one_epoch(bx, by)
            train_cost.append(cost)
            if val is not None:
                val_cost.append(self.compute_loss(val[0], val[1],
batch_size))
            if verbose != 0:
                if e % verbose == 0:
                    print("Epoch {}: train cost = {}".format(e, cost))
                if val is not None:
                    print("Epoch {}: val cost = {}".format(e, val_cost[-1]))
        if save_after is not None:
            self.save(save_after)
        return train_cost, val_cost

    def predict(self, X):
        """
        Predict the output of the model.
        args:
            X: input to the model
        returns:
            predictions
        """
        return self.forward(X)[-1]
```

در این کلاس، مدل با استفاده از یک معماری (arch)، تابع هزینه (criterion) و الگوریتم بهینه‌سازی (optimizer) ساخته می‌شود. این اطلاعات در کانستراکتور کلاس دریافت می‌شوند.

در این کد، توابع و متدهای زیر پیاده‌سازی شده‌اند:

۱. `is_layer(layer)` وظیفه آن بررسی این است که آیا ورودی یک لایه است یا نه.

۲. `is_activation(layer)` وظیفه آن بررسی این است که آیا لایه‌ای که به عنوان ورودی دریافت می‌کند، یک تابع فعال‌سازی است یا نه.

۳. `forward(x)` وظیفه آن انجام مراحل پیش‌روی شبکه بر روی ورودی x است و خروجی را برمی‌گرداند.

۴. `backward(dAL, tmp, x)` وظیفه آن انجام مراحل پس‌روی شبکه بر اساس مشتق هزینه نسبت به خروجی شبکه است و گرادیان‌های مدل را برمی‌گرداند.

۵. `update(grads)` وظیفه آن به‌روزرسانی پارامترهای مدل بر اساس گرادیان‌های محاسبه شده است.

۶. `one_epoch(x, y)` وظیفه آن اجرای یک دوره از آموزش است و هزینه را محاسبه می‌کند.

۷. `save(name)` وظیفه آن ذخیره کردن مدل به عنوان یک فایل است.

۸. `load_model(name)` وظیفه آن بارگیری مدل از یک فایل است.

۹. `shuffle(m, shuffling)` وظیفه آن ترتیب دادن داده‌ها را بررسی می‌کند و در صورت نیاز، ترتیب داده‌ها را تصادفی می‌کند.

۱۰. `batch(X, y, batch_size, index, order)` وظیفه آن انتخاب یک دسته از داده‌ها است که برای آموزش استفاده می‌شود.

۱۱. `compute_loss(X, y, batch_size)` وظیفه آن محاسبه هزینه بر روی داده‌ها است.

۱۲. `train(X, y, epochs, val=None, batch_size=۳, verbose=False, shuffling=۱, save_after=None)` وظیفه آن آموزش مدل بر روی داده‌ها است.

۱۳. `predict(X)` وظیفه آن پیش‌بینی خروجی مدل بر روی داده‌ها است.

در نهایت برای دو تسک داده شده دو مدل تعریف شد که به صورت مجزا در ژوپیتر نوت بوک توضیح داده شده اند.