گزارش پروژه شبکه های عصبی

بخش اول – لایه ها

* 1. 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 0.01 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) \* 0.01
* 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 xavier محاسبه 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(2 / (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(2 / 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, 1))

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) > 2:  
 batch\_size = A\_prev\_tmp.shape[0]  
 A\_prev\_tmp = A\_prev\_tmp.reshape(batch\_size, -1).T  
else:  
 batch\_size = A\_prev\_tmp.shape[0]  
  
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) > 2:  
 batch\_size = A\_prev\_tmp.shape[0]  
 A\_prev\_tmp = A\_prev\_tmp.reshape(batch\_size, -1).T  
 else:  
 batch\_size = A\_prev\_tmp.shape[0]  
  
 # Backward part  
 W, b = self.weights, self.biases  
 dW = np.dot(dZ, A\_prev\_tmp.T) / batch\_size  
 db = np.sum(dZ, axis=1, 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) > 2:  
 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 بخش backward، 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

* 1. لایه convolution دو بعدی:

تکمیل تابع وزن دهی:

مشابه وزن دهی در لایه fully connected عمل میکنیم:

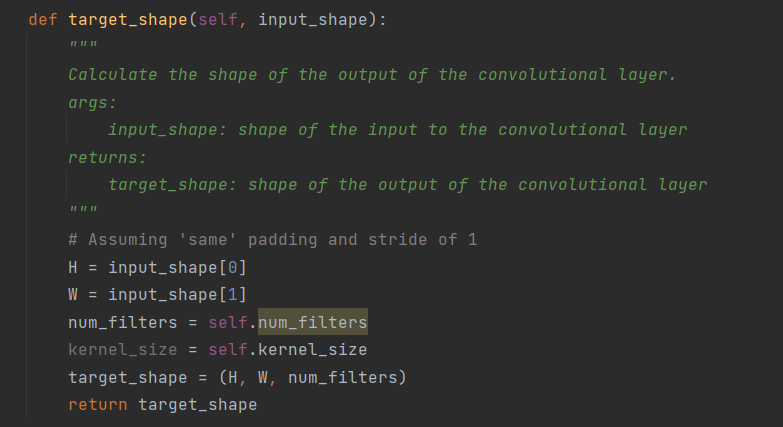
def initialize\_weights(self):  
 *"""  
 Initialize weights.  
 returns:  
 weights: initialized kernel with shape: (kernel\_size[0], kernel\_size[1], in\_channels, out\_channels)  
 """* if self.initialize\_method == "random":  
 return np.random.randn(self.kernel\_size[0], self.kernel\_size[1], self.in\_channels, self.out\_channels) \* 0.01  
 elif self.initialize\_method == "xavier":  
 xavier\_stddev = np.sqrt(2.0 / (self.in\_channels + self.out\_channels))  
 return np.random.randn(self.kernel\_size[0], self.kernel\_size[1], self.in\_channels,  
 self.out\_channels) \* xavier\_stddev  
 elif self.initialize\_method == "he":  
 he\_stddev = np.sqrt(2.0 / self.in\_channels)  
 return np.random.randn(self.kernel\_size[0], self.kernel\_size[1], 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((1, 1, 1, self.out\_channels)) \* 0.01  
 if self.initialize\_method == "xavier":  
 return np.zeros((1, 1, 1, self.out\_channels)) \* np.sqrt(1 / self.out\_channels)  
 if self.initialize\_method == "he":  
 return np.zeros((1, 1, 1, self.out\_channels)) \* np.sqrt(2 / 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 1و 1، 1، 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 :



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 تابع single\_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.float32(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 + 2 \* padding\_h - kernel\_size\_h) / stride\_h) + 1  
W = int((W\_prev + 2 \* padding\_w - kernel\_size\_w) / stride\_w) + 1  
# 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

4

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.

* 1. 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[0] - self.kernel\_size[0]) // self.stride[0] + 1  
 W = (input\_shape[1] - self.kernel\_size[1]) // self.stride[1] + 1  
 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 + 1

W = (W\_prev - pool\_width) / stride + 1

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 \* (1 - 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 لایه name 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=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):  
   self.layers = layers\_list  
   self.learning\_rate = learning\_rate  
   self.beta1 = beta1  
   self.beta2 = beta2  
   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.beta1 \* self.V[name][i] + (1 - self.beta1) \* grads[i]  
   self.S[name][i] = self.beta2 \* self.S[name][i] + (1 - self.beta2) \* np.square(grads[i])  
   V\_corrected = self.V[name][i] / (1 - np.power(self.beta1, epoch))  
   S\_corrected = self.S[name][i] / (1 - np.power(self.beta2, 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
* 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[1]  
   cost = -np.sum(y \* np.log(y\_hat + 1e-10) + (1 - y) \* np.log(1 - y\_hat + 1e-10)) / 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 \* (1 - y\_hat)) + 1e-10)

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
* 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: