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 زاویه‌گیری 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 فعالساز 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:

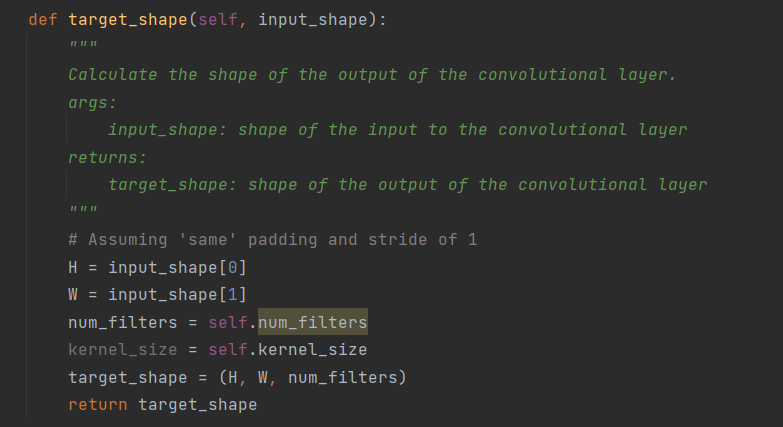
def initialize\_weights(self):  
 *"""  
 Initialize weights.  
 returns:  
 weights: initialized kernel with shape: (kernel\_size[0], kernel\_size[1], in\_channels, out\_channels)  
 """* if self.initialize\_method == "random":  
 return np.random.randn(self.kernel\_size[0], self.kernel\_size[1], self.in\_channels, 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 (1، 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 نرون 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. لازم به ذکر است که این پیاده‌سازی برای فرض پدینگ و استراید `same` و با توجه به تعداد فیلترها و اندازه کرنل مشخص شده، طراحی شده است و می‌توان آن را برای پدینگ و استراید‌های متفاوت، تغییر داد.

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

تابع backward در لایه کانولوشنی اجرا می‌شود. 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.