

I gave a structured way to give the answer since this tends to work best with LLMs. I wanted to try having it just use the raw PDF as well, rather than individually listing questions.

PDF CS182 Homework 4.pdf

You will be solving various questions from this Deep Learning homework PDF. You will go one question at a time, and only answer a certain question when I specify. If unspecified, complete all parts of the problem I reference. First, output the exact problem statement from the PDF, and then give a full explanation of how you reached your answer, along with the answer.

Start by solving #1.

### M Problem Statement: 1. Newton-Schulz Runtime

Let us consider the Newton-Schulz update for a parameter matrix  $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ , using degree-3 odd polynomial  $p$ :

$$p(W) = \frac{1}{2} (3I_{d_{\text{out}}} - WW^T) W$$

(a)

First, we will analyze the runtime of a single iteration. Assume that the runtime is dominated by matrix multiplication (which is often true for large  $d_{\text{out}}, d_{\text{in}}$ ) and the runtime of multiplying a  $n \times m$  matrix by a  $m \times p$  matrix takes  $c \cdot m \cdot n \cdot p$  runtime. What is the runtime of each iteration?

(b)

Now, consider the case where  $d_{\text{out}} \gg d_{\text{in}}$ . Is there a way to compute  $p(W)$  faster? Explain how this can be done and report the updated runtime of each iteration?

(Hint: Consider the Gram matrix  $W^T W$  instead of  $WW^T$ . What is the runtime of computing  $W^T W$ ? Is there a way to rewrite  $p(W)$  so that it uses  $W^T W$  instead of  $WW^T$ ?)

It correctly OCR'd the PDF and was able to copy the question correctly, which was nice.

### Explanation and Solution

#### (a) Runtime of a Single Iteration

The polynomial  $p(W)$  is given by:

$$p(W) = \frac{1}{2} (3I_{d_{\text{out}}} - WW^T) W$$

Let's break down the computation:

1. **Compute  $WW^T$ :**

- $W$  is  $d_{\text{out}} \times d_{\text{in}}$ .
- $W^T$  is  $d_{\text{in}} \times d_{\text{out}}$ .
- The product  $WW^T$  is  $d_{\text{out}} \times d_{\text{out}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{out}} = c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

2. **Compute  $3I_{d_{\text{out}}} - WW^T$ :**

- This is a matrix subtraction and scalar multiplication, which is  $O(d_{\text{out}}^2)$ , negligible compared to matrix multiplication.

3. **Multiply the result by  $W$ :**

- The result of the subtraction is  $d_{\text{out}} \times d_{\text{out}}$ .
- Multiplying by  $W$  ( $d_{\text{out}} \times d_{\text{in}}$ ) gives  $d_{\text{out}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{out}} \cdot d_{\text{in}} = c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

This was a nice, structured response to the question. The answer is an order of magnitude and didn't include constants, though, like the answer key did.

**Total runtime:**

$c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$  (dominated by the two matrix multiplications).

**(b) Faster Computation When  $d_{\text{out}} \gg d_{\text{in}}$**

**Hint:** Use the Gram matrix  $W^T W$  instead of  $WW^T$ .

1. **Rewrite  $p(W)$ :**

$$p(W) = \frac{1}{2} (3I_{d_{\text{out}}} - WW^T) W = \frac{1}{2} (3W - WW^T W)$$

Notice that  $WW^T W = W(W^T W)$ .

It correctly identified the trick here with changing the multiplication order.

2. **Compute  $W^T W$ :**

- $W^T W$  is  $d_{\text{in}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{in}} \cdot d_{\text{out}} \cdot d_{\text{in}} = c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$ .

3. **Multiply  $W(W^T W)$ :**

- $W$  is  $d_{\text{out}} \times d_{\text{in}}$ ,  $W^T W$  is  $d_{\text{in}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{in}} = c \cdot d_{\text{out}} \cdot d_{\text{in}}^2$ .

4. **Compute  $3W - W(W^T W)$ :**

- This is a matrix subtraction and scalar multiplication, which is  $O(d_{\text{out}} \cdot d_{\text{in}})$ , negligible.

**Total runtime:**

$c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$ , which is faster when  $d_{\text{out}} \gg d_{\text{in}}$ .

It correctly gave the order of magnitude again. It seems it was aiming more for big O runtime, which makes sense as that is usually how runtime is analyzed.

### Final Answers

(a)

The runtime of each iteration is  $c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

(b)

Yes, by rewriting  $p(W)$  as  $\frac{1}{2} (3W - W(W^T W))$ , the runtime becomes  $c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$ , which is faster when  $d_{\text{out}} \gg d_{\text{in}}$ .

Please ensure your answers are exact, including any constants, unless otherwise specified in the problem. So please do not do big-O style runtimes. Now please resolve #1.

I decided to prompt it to give a more precise answer



### Revised Solution for 1. Newton-Schulz Runtime

#### (a) Runtime of a Single Iteration

The polynomial is:

$$p(W) = \frac{1}{2} (3I_{d_{\text{out}}} - WW^T) W$$

Step-by-step runtime:

1. **Compute  $WW^T$ :**

- $W$  is  $d_{\text{out}} \times d_{\text{in}}$ .
- $W^T$  is  $d_{\text{in}} \times d_{\text{out}}$ .
- The product  $WW^T$  is  $d_{\text{out}} \times d_{\text{out}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{out}} = c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

2. **Compute  $3I_{d_{\text{out}}} - WW^T$ :**

- Scalar multiplication and subtraction: negligible compared to matrix multiplication.

3. **Multiply the result by  $W$ :**

- The result of the subtraction is  $d_{\text{out}} \times d_{\text{out}}$ .
- Multiplying by  $W$  ( $d_{\text{out}} \times d_{\text{in}}$ ) gives  $d_{\text{out}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{out}} \cdot d_{\text{in}} = c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

**Total runtime:**

$$2c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$$

Now it gave the answer in the answer key. I think it would have gotten this first try had it been more clear how specific the runtime should be.

---

**(b) Faster Computation When  $d_{\text{out}} \gg d_{\text{in}}$**

Rewriting  $p(W)$ :

$$p(W) = \frac{1}{2} (3W - W(W^TW))$$

**Step-by-step runtime:**

1. **Compute  $W^TW$ :**

- $W^TW$  is  $d_{\text{in}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{in}} \cdot d_{\text{out}} \cdot d_{\text{in}} = c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$ .

2. **Multiply  $W(W^TW)$ :**

- $W$  is  $d_{\text{out}} \times d_{\text{in}}$ ,  $W^TW$  is  $d_{\text{in}} \times d_{\text{in}}$ .
- Runtime:  $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{in}} = c \cdot d_{\text{out}} \cdot d_{\text{in}}^2$ .

### 3. Compute $3W - W(W^T W)$ :

- Scalar multiplication and subtraction: negligible.

**Total runtime:**

$$c \cdot d_{\text{in}}^2 \cdot d_{\text{out}} + c \cdot d_{\text{out}} \cdot d_{\text{in}}^2 = 2c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$$

It solved this one correctly as well after more clear instructions.

**Conclusion:**

When  $d_{\text{out}} \gg d_{\text{in}}$ , the runtime is  $2c \cdot d_{\text{in}}^2 \cdot d_{\text{out}}$ , which is faster than the original  $2c \cdot d_{\text{out}}^2 \cdot d_{\text{in}}$ .

Now please solve #2a. Make sure to repeat the question first before solving.

**Since this is a longer and more complicated problem, I decided to go one part at a time.**

#### Problem Statement: 2. MuP at the Unit Scale (a)

You are designing a neural network training algorithm that will be trained with fp8 parameters (assume activations will be calculated in full precision and we don't need to worry about them). Why would it make sense for parameters to be initialized from  $N(0, 1)$  as opposed to e.g. Xavier initialization? As a starting point, note that fp8 can only represent 255 possible values, and you can view these values at: [fp8 tables](#).

#### Explanation and Solution

##### Why $N(0, 1)$ Initialization?

###### 1. Limited Representational Capacity of fp8:

- fp8 (8-bit floating point) can only represent 255 distinct values, including subnormal numbers, zero, and special values like infinity and NaN.
- The range of representable values is very limited compared to fp32 or fp16, and the gaps between consecutive values are larger.

###### 2. Xavier Initialization:

- Xavier initialization (Glorot initialization) scales the initial weights by  $\frac{1}{\sqrt{n}}$ , where  $n$  is the number of input units.

**This is the first conceptual question. It is doing pretty well so far (it is a simple question, so this is expected).**

- This is designed to keep the variance of activations roughly the same across every layer, which is crucial for deep networks in high precision (e.g., fp32).
- However, in fp8, the scaled values may be too small to be represented meaningfully, or may all collapse to zero due to the limited precision.

### 3. $N(0, 1)$ Initialization:

- Initializing weights from  $N(0, 1)$  means the weights are drawn from a standard normal distribution with mean 0 and variance 1.
- This ensures that the initial weights are spread out over the limited range of fp8, making use of as many distinct representable values as possible.
- This avoids the issue of many weights being rounded to the same value (e.g., zero), which would happen if the weights were too small (as they might be with Xavier initialization in fp8).

### 4. Practical Implication:

- With  $N(0, 1)$ , the weights are more likely to be distinct and cover a wider range of the limited fp8 values, which helps preserve information and gradients during training.

### Final Answer

In fp8, initializing parameters from  $N(0, 1)$  is preferable to Xavier initialization because fp8's limited representational capacity (only 255 distinct values) means that Xavier's scaled-down weights would often round to the same value (e.g., zero), losing information.  $N(0, 1)$  initialization spreads the weights across the available fp8 values, preserving more information and enabling more effective training. Thus, the answer is:

To avoid many weights rounding to the same value in fp8,  $N(0, 1)$  initialization spreads

**Unfortunately the rest of the answer got cut off in conversion, but there was a simple summary there. The answer was very clear and showed good reasoning.**

Now please solve #2b.



### Problem Statement: 2. MuP at the Unit Scale (b)

We have initialized our parameters from  $N(0, 1)$ . However, we have now lost the desirable properties of Xavier initialization, and our activations are exploding as they propagate deeper

into the network. To solve this, we can assign a constant (float) scalar to be multiplied with the activations:

$$\mathbf{y} = cW\mathbf{x}$$

What should the constant scalar  $c$  be to recover the benefits of standard Xavier initialization?

## Explanation and Solution

### Xavier Initialization Recap

Xavier initialization is designed to keep the variance of activations roughly the same across layers. For a weight matrix  $W$  of size  $d_{\text{out}} \times d_{\text{in}}$ , Xavier initialization scales the initial weights by  $\frac{1}{\sqrt{d_{\text{in}}}}$ . This ensures that the variance of the output activations  $\mathbf{y} = W\mathbf{x}$  is approximately equal to the variance of the input activations  $\mathbf{x}$ , assuming the weights and inputs are independent and have zero mean.

**This recap of xavier initialization was good and probably helpful as more context to help solve the problem. Mistral seems to be very organized and structured here.**

#### Current Setup

- We initialize  $W$  from  $N(0, 1)$ , so the variance of each weight is 1.
- The output activation is  $\mathbf{y} = cW\mathbf{x}$ .

### Goal

We want the variance of  $\mathbf{y}$  to match the variance of  $\mathbf{x}$ , just like in Xavier initialization.

### Derivation

#### 1. Variance of $W\mathbf{x}$ :

- Let  $\text{Var}(x_i) = \sigma_x^2$  for each component of  $\mathbf{x}$ .
- Let  $\text{Var}(w_{ij}) = 1$  for each weight in  $W$ .
- The variance of each component of  $W\mathbf{x}$  is:

$$\text{Var}\left(\sum_{j=1}^{d_{\text{in}}} w_{ij}x_j\right) = \sum_{j=1}^{d_{\text{in}}} \text{Var}(w_{ij}x_j) = d_{\text{in}} \cdot \text{Var}(w_{ij}) \cdot \text{Var}(x_j) = d_{\text{in}} \cdot \sigma_x^2$$

Copy

(assuming independence between weights and inputs).

Not sure why it did this little code block with this piece of text. It seems to happen every once in a while after an equation.

## 2. Scaling by $c$ :

- The variance of  $\mathbf{y} = cW\mathbf{x}$  is:

$$\text{Var}(y_i) = c^2 \cdot d_{\text{in}} \cdot \sigma_x^2$$

## 3. Matching Variances:

- To match the variance of  $\mathbf{y}$  to the variance of  $\mathbf{x}$ , we set:

$$c^2 \cdot d_{\text{in}} \cdot \sigma_x^2 = \sigma_x^2$$

- Solving for  $c$ :

$$c^2 = \frac{1}{d_{\text{in}}} \implies c = \frac{1}{\sqrt{d_{\text{in}}}}$$

## Final Answer

The constant scalar  $c$  should be:

$$\frac{1}{\sqrt{d_{\text{in}}}}$$

Very clean and methodical answer! It correctly solved all the relationships and got the right answer.

Now #2c



Problem Statement: 2. MuP at the Unit Scale (c)

Now, let us consider an update to the weights  $\Delta W$ . We would like to properly scale this update, such that the resulting  $\Delta \mathbf{y} = c\Delta W \mathbf{x}$  is controlled. Assume that  $\mathbf{x}$  has an RMS norm of 1. What should the maximum spectral norm of  $\Delta W$  be such that  $\Delta \mathbf{y}$  has an RMS norm no larger than 1?

## Explanation and Solution

### Definitions and Setup

- The RMS norm of a vector  $\mathbf{v}$  is defined as:

$$\|\mathbf{v}\|_{\text{RMS}} = \sqrt{\frac{1}{n} \sum_{i=1}^n v_i^2}$$

**Good bit of context building. These make the solutions all very clear.**

where  $n$  is the dimension of  $\mathbf{v}$ .

- The spectral norm of a matrix  $A$ , denoted  $\|A\|_2$ , is the largest singular value of  $A$ . It represents the maximum "gain" of the matrix when applied to any unit vector.
- Given  $\Delta \mathbf{y} = c\Delta W \mathbf{x}$ , we want:

$$\|\Delta \mathbf{y}\|_{\text{RMS}} \leq 1$$

**It tends to simplify the question into some objective expression, which is great**

### Assumptions

- $\mathbf{x}$  has an RMS norm of 1, so:

$$\|\mathbf{x}\|_{\text{RMS}} = 1 \implies \|\mathbf{x}\|_2 = \sqrt{d_{\text{in}}}$$

(since  $\|\mathbf{x}\|_2^2 = d_{\text{in}} \cdot \|\mathbf{x}\|_{\text{RMS}}^2$ ).

- We want to ensure that the RMS norm of  $\Delta \mathbf{y}$  is no larger than 1.

### Derivation

- Express  $\Delta \mathbf{y}$  in terms of norms:

$$\|\Delta \mathbf{y}\|_2 = \|c\Delta W \mathbf{x}\|_2 \leq |c| \cdot \|\Delta W\|_2 \cdot \|\mathbf{x}\|_2$$

**Correct so far, correctly identifying the types of norms so that it's not confused by which norm it computed.**

2. RMS norm of  $\Delta y$ :

$$\|\Delta y\|_{\text{RMS}} = \frac{\|\Delta y\|_2}{\sqrt{d_{\text{out}}}} \leq \frac{|c| \cdot \|\Delta W\|_2 \cdot \sqrt{d_{\text{in}}}}{\sqrt{d_{\text{out}}}}$$

3. Set the RMS norm constraint:

$$\frac{|c| \cdot \|\Delta W\|_2 \cdot \sqrt{d_{\text{in}}}}{\sqrt{d_{\text{out}}}} \leq 1$$

4. Recall from part (b) that  $c = \frac{1}{\sqrt{d_{\text{in}}}}$ :

It was able to recall and use its previous answer correctly here, which was very good.

$$\frac{\frac{1}{\sqrt{d_{\text{in}}}} \cdot \|\Delta W\|_2 \cdot \sqrt{d_{\text{in}}}}{\sqrt{d_{\text{out}}}} \leq 1 \implies \frac{\|\Delta W\|_2}{\sqrt{d_{\text{out}}}} \leq 1$$

5. Solve for  $\|\Delta W\|_2$ :

$$\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$$

---

### Final Answer

The maximum spectral norm of  $\Delta W$  should be:

$$\boxed{\sqrt{d_{\text{out}}}}$$

Solving everything, it got the correct answer clearly.

#2d



### Problem Statement: 2. MuP at the Unit Scale (d)

Let us consider the case of SignSGD. Assume that our minibatch is of size 1. You saw in discussion that  $\text{sign}(\nabla_W L)$  is a rank-1 matrix. What learning rate  $\alpha$  is required to ensure that

the overall update of  $\alpha \cdot \text{sign}(\nabla_W L)$  satisfies the spectral norm constraint from part (c)? Ensure your answer works on rectangular weight matrices.

## Explanation and Solution

### Setup and Definitions

- **SignSGD Update:** The update rule for SignSGD is:

$$\Delta W = \alpha \cdot \text{sign}(\nabla_W L)$$

where  $\text{sign}(\nabla_W L)$  is the element-wise sign of the gradient matrix  $\nabla_W L$ .

- **Rank-1 Property:**

For a minibatch of size 1,  $\nabla_W L$  is an outer product of two vectors, making it a rank-1 matrix. Therefore,  $\text{sign}(\nabla_W L)$  is also rank-1.

- **Spectral Norm of Rank-1 Matrix:**

For a rank-1 matrix  $A = \mathbf{u}\mathbf{v}^T$ , the spectral norm is:

$$\|A\|_2 = \|\mathbf{u}\|_2 \cdot \|\mathbf{v}\|_2$$

For  $\text{sign}(\nabla_W L)$ , each entry is either  $+1$  or  $-1$ . If  $\nabla_W L$  is  $d_{\text{out}} \times d_{\text{in}}$ , then:

$$\|\text{sign}(\nabla_W L)\|_2 = \sqrt{d_{\text{out}} \cdot d_{\text{in}}}$$

(since the Frobenius norm of a matrix of all  $\pm 1$  is  $\sqrt{d_{\text{out}} d_{\text{in}}}$ , and for rank-1 matrices, the spectral norm equals the Frobenius norm).

**Again, this setup and definitions section is quite clear, informative, and all relevant.**

### Spectral Norm Constraint from Part (c)

From part (c), the spectral norm of  $\Delta W$  must satisfy:

$$\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$$

### Derivation

1. Express the update:

$$\Delta W = \alpha \cdot \text{sign}(\nabla_W L)$$

2. Apply the spectral norm constraint:

$$\|\Delta W\|_2 = |\alpha| \cdot \|\text{sign}(\nabla_W L)\|_2 \leq \sqrt{d_{\text{out}}}$$

3. Substitute the spectral norm of  $\text{sign}(\nabla_W L)$ :

$$|\alpha| \cdot \sqrt{d_{\text{out}} \cdot d_{\text{in}}} \leq \sqrt{d_{\text{out}}}$$

4. Solve for  $\alpha$ :

$$|\alpha| \leq \frac{\sqrt{d_{\text{out}}}}{\sqrt{d_{\text{out}} \cdot d_{\text{in}}}} = \frac{1}{\sqrt{d_{\text{in}}}}$$

### Final Answer

The learning rate  $\alpha$  should be:

$$\boxed{\frac{1}{\sqrt{d_{\text{in}}}}}$$

It got the correct answer again by substituting in a previous answer and solving the equation in clear steps. Doing well so far...

2e



### Problem Statement: 2. MuP at the Unit Scale (e)

Let us consider the usage of Muon-style methods to orthogonalize our gradients. Consider the following orthogonalized update rule:

$$U, \Sigma, V^T = \text{SVD}(\nabla_W L)$$

$$\Delta W = \alpha \cdot UV^T$$

where we use the compact form of the SVD. What learning rate  $\alpha$  is required to ensure that the overall update  $\Delta W$  satisfies the spectral norm constraint from part (c)?

## Explanation and Solution

### Setup and Definitions

- **SVD of the Gradient:**

The SVD of  $\nabla_W L$  is  $U\Sigma V^T$ , where  $U$  and  $V$  are orthogonal matrices, and  $\Sigma$  is a diagonal matrix of singular values.

- **Orthogonalized Update:**

The update rule is:

$$\Delta W = \alpha \cdot UV^T$$

Here,  $UV^T$  is an orthogonal matrix (since  $U$  and  $V$  are orthogonal).

- **Spectral Norm of Orthogonal Matrices:** The spectral norm of an orthogonal matrix is 1, because orthogonal matrices preserve the norm of vectors.

### Spectral Norm Constraint from Part (c)

From part (c), the spectral norm of  $\Delta W$  must satisfy:

$$\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$$

### Derivation

1. **Express the update:**

$$\Delta W = \alpha \cdot UV^T$$

2. **Apply the spectral norm constraint:**

$$\|\Delta W\|_2 = |\alpha| \cdot \|UV^T\|_2 \leq \sqrt{d_{\text{out}}}$$

3. Substitute the spectral norm of  $UV^T$ :

$$|\alpha| \cdot 1 \leq \sqrt{d_{\text{out}}}$$

4. Solve for  $\alpha$ :

$$|\alpha| \leq \sqrt{d_{\text{out}}}$$

---

### Final Answer

The learning rate  $\alpha$  should be:

$$\boxed{\sqrt{d_{\text{out}}}}$$

It got the correct answer, I believe (NOTE: as of 12/3, this doesn't match the answer key, but I posted on Ed about the answer key potentially being incorrect, as I believe this is the correct answer, and I got this when solving it as well)

2f



### Problem Statement: 2. MuP at the Unit Scale (f)

SignGD, Adam, and Muon share a similar property that the global scale of raw gradients does not affect the final update direction. Now consider the backwards pass of a series of dense layers, where each layer follows the scaled definition from part (b):  $\mathbf{x}_{n+1} = c_n W_n \mathbf{x}_n$ . You may assume there is no activation function for simplicity. Recall that  $\nabla_{\mathbf{x}_n} L$  can be recursively calculated from  $\nabla_{\mathbf{x}_{n+1}} L$ . Is there a setting where the scale of these intermediate backpropagated gradients can also be ignored?

---

### Explanation and Solution

#### Setup and Definitions

- **Forward Pass:** Each layer is defined as:

$$\mathbf{x}_{n+1} = c_n W_n \mathbf{x}_n$$

where  $c_n = \frac{1}{\sqrt{d_n}}$  (from part (b)), and  $d_n$  is the input dimension of layer  $n$ .

- **Backward Pass:** The gradient with respect to  $\mathbf{x}_n$  is:

$$\nabla_{\mathbf{x}_n} L = W_n^T \nabla_{\mathbf{x}_{n+1}} L \cdot c_n$$

(since  $\nabla_{\mathbf{x}_n} L = \nabla_{\mathbf{x}_{n+1}} L \cdot \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n} = \nabla_{\mathbf{x}_{n+1}} L \cdot (c_n W_n)$ ).

- **Recursive Gradient Calculation:** The gradient at layer  $n$  depends on the gradient at layer  $n + 1$  and the weight matrix  $W_n$ , scaled by  $c_n$ .

### When Can the Scale Be Ignored?

- **Orthogonal Weight Matrices:** If all weight matrices  $W_n$  are orthogonal (i.e.,  $W_n^T W_n = I$ ), then:

$$\nabla_{\mathbf{x}_n} L = c_n W_n^T \nabla_{\mathbf{x}_{n+1}} L$$

The norm of  $\nabla_{\mathbf{x}_n} L$  is:

$$\|\nabla_{\mathbf{x}_n} L\| = |c_n| \cdot \|W_n^T \nabla_{\mathbf{x}_{n+1}} L\| = |c_n| \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|$$

because orthogonal matrices preserve norms.

- **Recursive Scaling:**

If  $c_n = 1$  for all  $n$ , the scale of the gradients would be preserved exactly. However, with  $c_n = \frac{1}{\sqrt{d_n}}$ , the gradient norms are scaled by  $\frac{1}{\sqrt{d_n}}$  at each layer.

- **Balanced Scaling:**

If the product of all  $c_n$  along the path cancels out (e.g., if the network is balanced in width), the overall scale of the gradients may be preserved, and the intermediate scales can be ignored in terms of their effect on the update direction.

- **Normalized Gradients:**

If the gradients are normalized (e.g., by their norm) at each layer before being used for updates, the global scale can be ignored, and only the direction matters.

### Key Insight

**It had a long answer with a lot of different nuances here. It talked about a lot of different factors that could lead to the scale being ignored, but then it finally mentioned the one which we were focusing on in the problem.**

The scale of intermediate backpropagated gradients can be ignored if the updates are normalized by their norm (e.g., in SignGD, Adam, or Muon-style methods), or if the network is constructed such that the product of all scaling factors  $c_n$  is 1.

### Final Answer

Yes, the scale of intermediate backpropagated gradients can be ignored if the updates are normalized by their norm (as in SignGD, Adam, or Muon-style methods), or if the network is constructed so that the product of all scaling factors  $c_n$  is 1, preserving the gradient norm across layers.

Yes, if the updates are normalized by their norm (e.g., in SignGD, Adam, or Muon), or

Again, that box was cut off, but it had a summary right here. The final answer was correct and focused on the point of the question, so it did well. Its overall answer was maybe too long, but it got there in the end, without extra help.

2g



Worked for 5s



### Problem Statement: 2. MuP at the Unit Scale (g)

In the setting above, without any adjustments, will intermediate backpropagated gradients suffer from an explosion or vanishing effect as they are backpropagated? You may assume that  $W$  is rank-1, such that the spectral norm is equal to the Frobenius norm, and each parameter is unit scaled. What constants should the intermediate backpropagated gradients be multiplied by to ensure that they remain stable?

### Explanation and Solution

#### Backpropagation with Rank-1 Weight Matrices

- When  $W$  is rank-1, the spectral norm  $\|W\|_2$  equals the Frobenius norm  $\|W\|_F$ .
- For a rank-1 matrix  $W = \mathbf{u}\mathbf{v}^T$ , the spectral norm is  $\|W\|_2 = \|\mathbf{u}\|_2 \cdot \|\mathbf{v}\|_2$ .
- If each parameter is unit scaled, then  $\|\mathbf{u}\|_2 = \|\mathbf{v}\|_2 = 1$ , so  $\|W\|_2 = 1$ .

Its first mistake! Those should not be 1, those should be `sqrt(d_in)` and `sqrt(d_out)`.

#### Gradient Backpropagation

- The gradient with respect to the input of layer  $n$  is:

$$\nabla_{\mathbf{x}_n} L = W^T \nabla_{\mathbf{x}_{n+1}} L \cdot c_n$$

- Since  $W$  is rank-1 and unit scaled,  $W^T$  is also rank-1 and unit scaled, so  $\|W^T\|_2 = 1$ .
- The norm of the gradient at layer  $n$  is:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \|W^T\|_2 \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot |c_n| = \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot |c_n|$$

- From part (b),  $c_n = \frac{1}{\sqrt{d_n}}$ , where  $d_n$  is the input dimension of layer  $n$ .

### Effect on Gradients

- As we backpropagate through  $L$  layers, the gradient norm becomes:

$$\|\nabla_{\mathbf{x}_1} L\|_2 = \|\nabla_{\mathbf{x}_{L+1}} L\|_2 \cdot \prod_{n=1}^L |c_n| = \|\nabla_{\mathbf{x}_{L+1}} L\|_2 \cdot \prod_{n=1}^L \frac{1}{\sqrt{d_n}}$$

**It is trying to show the accumulation of the magnitude change, which is valid**

- If the product  $\prod_{n=1}^L \frac{1}{\sqrt{d_n}}$  is much less than 1, the gradients will vanish exponentially with depth.
- If the product is much greater than 1, the gradients will explode.

### Vanishing or Exploding?

- Typically,  $d_n$  is large, so  $\frac{1}{\sqrt{d_n}}$  is small, leading to **vanishing gradients** as the depth increases.

### Stabilizing the Gradients

- To stabilize the gradients, we want the product of all scaling factors to be 1:

$$\prod_{n=1}^L c_n = 1$$

- If all layers have the same dimension  $d$ , then:

$$c_n = d^{1/(2L)}$$

- This ensures that the product of all  $c_n$  is 1, and the gradient norm is preserved across layers.

### Final Answer

Without adjustments, intermediate backpropagated gradients will suffer from **vanishing** as they are backpropagated. To stabilize the gradients, each intermediate gradient should be multiplied by  $d_n^{1/(2L)}$ , where  $d_n$  is the input dimension of layer  $n$  and  $L$  is the total number of layers. This ensures the product of all scaling factors is 1, preserving gradient norms across the network.

**Another mistake here! It is forgetting that as a result of previous scaling factors, the gradient passed in will already be scaled, so it doesn't need to use L again. However, it is also possible it was just assuming that the gradients passed back are unscaled.**

Please check your weight matrices' spectral norm more carefully. Also, remember these scalings work off of each other, so when backpropagating you can just assume that you just have to deal with the effects of the new layer in scaling. Now please resolve 2g

**I prompted it to correct itself, providing minimal guidance to point it in the right direction.**



### Revised Solution for 2. MuP at the Unit Scale (g)

#### Setup and Definitions

- **Forward Pass:** Each layer is defined as:

$$\mathbf{x}_{n+1} = c_n W_n \mathbf{x}_n$$

where  $W_n$  is rank-1 and unit scaled, so  $\|W_n\|_2 = \|W_n\|_F = 1$ .

- **Backward Pass:** The gradient with respect to  $\mathbf{x}_n$  is:

$$\nabla_{\mathbf{x}_n} L = W_n^T \nabla_{\mathbf{x}_{n+1}} L \cdot c_n$$

Since  $W_n$  is rank-1 and unit scaled,  $\|W_n^T\|_2 = 1$ .

**Right here, it does the same mistake without even redoing the work! It will get the wrong answer again due to this.**

- **Recursive Gradient Calculation:** The norm of the gradient at layer  $n$  is:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \|W_n^T\|_2 \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot |c_n| = \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot |c_n|$$

From part (b),  $c_n = \frac{1}{\sqrt{d_n}}$ , where  $d_n$  is the input dimension of layer  $n$ .

### Effect on Gradients

- For a single layer, the gradient is scaled by  $\frac{1}{\sqrt{d_n}}$ .
- Since  $d_n$  is typically large,  $\frac{1}{\sqrt{d_n}} < 1$ , so the gradient norm **decreases** as it backpropagates through each layer.

### Vanishing or Exploding?

- **Without any adjustments**, the gradient norm decreases by a factor of  $\frac{1}{\sqrt{d_n}}$  at each layer.
- This leads to **vanishing gradients** as the depth increases.

### Stabilizing the Gradients

- To stabilize the gradients, we want the scaling at each layer to cancel out the effect of the forward pass scaling.
- If we multiply the backpropagated gradient at each layer by  $\sqrt{d_n}$ , the net effect is:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot \frac{1}{\sqrt{d_n}} \cdot \sqrt{d_n} = \|\nabla_{\mathbf{x}_{n+1}} L\|_2$$

- Thus, the gradient norm is preserved across layers.

### Final Answer

Without adjustments, intermediate backpropagated gradients will suffer from **vanishing** as they are backpropagated. To stabilize the gradients, each intermediate backpropagated gradient should be multiplied by  $\sqrt{d_n}$ , where  $d_n$  is the input dimension of layer  $n$ . This ensures the gradient norm remains stable across the network.

This time, it corrected the use of  $L$  and just treated each layer as if the other layer already scaled its gradient which it passed back. But due to the same other error, it's still wrong.

Again, check the norm of the  $W$  matrix. each element in the matrix is unit scaled but that does not mean that the norm is 1

Therefore, I here gave more specific guidance regarding that mistake. I told it where it's wrong but not what the answer is.



You are absolutely right to point out the need for careful attention to the norm of the rank-1 weight matrix  $W$ . Let's correct this:

## Revised Explanation and Solution for 2. MuP at the Unit Scale (g)

### Norm of Rank-1 Weight Matrix $W$

- If  $W$  is a rank-1 matrix of size  $d_{\text{out}} \times d_{\text{in}}$  with each element unit scaled (i.e., each entry is  $\pm 1$ ), then:

$$W = \mathbf{u}\mathbf{v}^T$$

where  $\mathbf{u} \in \mathbb{R}^{d_{\text{out}}}$  and  $\mathbf{v} \in \mathbb{R}^{d_{\text{in}}}$  are vectors of  $\pm 1$ .

- The Frobenius norm of  $W$  is:

$$\|W\|_F = \sqrt{\sum_{i=1}^{d_{\text{out}}} \sum_{j=1}^{d_{\text{in}}} w_{ij}^2} = \sqrt{d_{\text{out}} \cdot d_{\text{in}}}$$

- Since  $W$  is rank-1, the spectral norm  $\|W\|_2$  equals the Frobenius norm:

$$\|W\|_2 = \sqrt{d_{\text{out}} \cdot d_{\text{in}}}$$

**This time its answer is correct! I think it assumed the overall matrix was unit scaled, not the individual weights. Now it will get the correct answer if it uses the same analysis.**

### Backpropagation with Scaling

- The gradient with respect to  $\mathbf{x}_n$  is:

$$\nabla_{\mathbf{x}_n} L = W^T \nabla_{\mathbf{x}_{n+1}} L \cdot c_n$$

- The norm of the gradient at layer  $n$  is:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \|W^T\|_2 \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot |c_n|$$

- Since  $\|W^T\|_2 = \|W\|_2 = \sqrt{d_{\text{out}} \cdot d_{\text{in}}}$ , we have:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \sqrt{d_{\text{out}} \cdot d_{\text{in}}} \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot \frac{1}{\sqrt{d_{\text{in}}}}$$

- Simplifying:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \sqrt{d_{\text{out}}} \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2$$

### Effect on Gradients

- The gradient norm **increases** by a factor of  $\sqrt{d_{\text{out}}}$  at each layer.
- This leads to **exploding gradients** as the depth increases.

### Stabilizing the Gradients

- To stabilize the gradients, we want the gradient norm to remain constant across layers.
- Therefore, we need to multiply the backpropagated gradient by  $\frac{1}{\sqrt{d_{\text{out}}}}$  at each layer:

$$\|\nabla_{\mathbf{x}_n} L\|_2 = \sqrt{d_{\text{out}}} \cdot \|\nabla_{\mathbf{x}_{n+1}} L\|_2 \cdot \frac{1}{\sqrt{d_{\text{out}}}} = \|\nabla_{\mathbf{x}_{n+1}} L\|_2$$

---

### Final Answer

Without adjustments, intermediate backpropagated gradients will suffer from **exploding** as they are backpropagated. To stabilize the gradients, each intermediate backpropagated gradient

should be multiplied by  $\frac{1}{\sqrt{d_{\text{out}}}}$ , where  $d_{\text{out}}$  is the output dimension of layer  $n$ . This ensures the gradient norm remains stable across the network.

**It got the right answer now! These errors could be chalked up to misinterpreting less specific parts of the problem, but it was clear enough that I think it is an error.**

Now solve problem 3 (all parts)

I wanted to try to test it with a harder task again, solving the full problem. Problem 3 was pretty long and quite computation-heavy, so it would be interesting to see the result.



Worked for 8s

Code output generated    "# Let's first define the rectangular signal and the impulse response,...

Python

Copy

```
# Let's first define the rectangular signal and the impulse response, then compute  
their convolution.
```

```
import numpy as np  
import matplotlib.pyplot as plt  
  
# Define the rectangular signal x(n) of length L=7  
L = 7  
x = np.zeros(20) # Pad with zeros for convolution  
x[2:2+L] = 1 # x(n) is 1 from n=0 to n=6  
  
# Define the impulse response h(n)  
n_h = np.arange(20)  
h = (0.5 ** n_h) * (n_h >= 0) # h(n) = (1/2)^n for n >= 0  
  
# Compute the convolution y(n) = x(n) * h(n)  
y = np.convolve(x, h, mode='full')  
  
# Plot the convolution result  
n_y = np.arange(-6, 13) # Time indices from -6 to +12  
plt.figure(figsize=(12, 6))  
plt.stem(n_y, y[n_y + len(n_y) - 1], use_line_collection=True)  
plt.title('Convolution of Rectangular Signal and Impulse Response')  
plt.xlabel('n')  
plt.ylabel('y(n)')  
plt.grid(True)  
plt.show()
```

Result

Copy

```
<Figure size 1200x600 with 0 Axes>
```

```
Traceback (most recent call last):
```

```
  File "/tmp/ipykernel_375/3090861023.py", line 21, in <module>  
    plt.stem(n_y, y[n_y + len(n_y) - 1], use_line_collection=True)  
TypeError: stem() got an unexpected keyword argument 'use_line_collection'
```

**It uses code to generate plots; failed the first time and then retries**

Code output generated

"# Correcting the plotting code for the convolution result"

Python

Copy

```
# Correcting the plotting code for the convolution result

import matplotlib.pyplot as plt

# Define the rectangular signal x(n) of length L=7
L = 7
x = np.zeros(20) # Pad with zeros for convolution
x[2:2+L] = 1 # x(n) is 1 from n=0 to n=6

# Define the impulse response h(n)
n_h = np.arange(20)
h = (0.5 ** n_h) * (n_h >= 0) # h(n) = (1/2)^n for n >= 0

# Compute the convolution y(n) = x(n) * h(n)
y = np.convolve(x, h, mode='full')

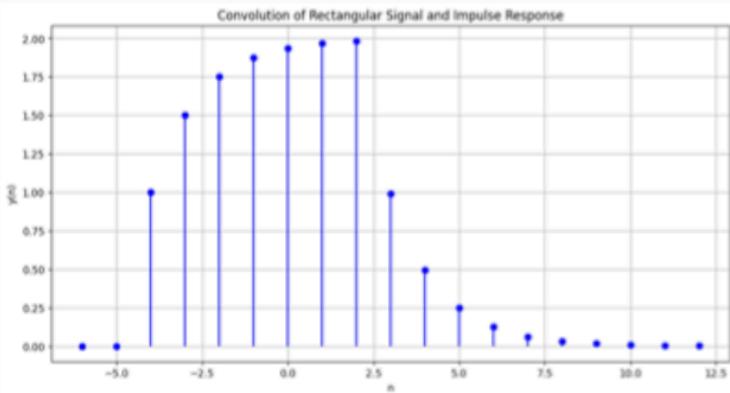
# Plot the convolution result
n_y = np.arange(-6, 13) # Time indices from -6 to +12
plt.figure(figsize=(12, 6))
plt.stem(n_y, y[:len(n_y)], linefmt='b-', markerfmt='bo', basefmt=' ')
plt.title('Convolution of Rectangular Signal and Impulse Response')
plt.xlabel('n')
plt.ylabel('y(n)')
plt.grid(True)
plt.show()
```

Result

Copy

<Figure size 1200x600 with 1 Axes>

It succeeded this time with the figure.



This plot is indeed correct (just ignoring the extra lines and looking at the points).

### Problem Statement: 3. Understanding Convolution as Finite Impulse Response Filter

#### (a) Convolution of Rectangular Signal and Impulse Response

The convolution of the rectangular signal  $x(n)$  of length  $L = 7$  and the impulse response  $h(n) = \left(\frac{1}{2}\right)^n u(n)$  is computed and plotted below:

Convolution Plot      It stopped repeating the question here, which may have led to it forgetting to explicitly compute the expression.

#### (b) Shifted Rectangular Signal and Convolution

Let  $x_2(n) = x(n - N)$  with  $N = 5$ . The convolution  $y_2(n) = h(n) * x_2(n)$  is equivalent to shifting the result from part (a) by  $N = 5$  units to the right. This demonstrates the **time-shifting property** of convolution: if  $x(n)$  is shifted by  $N$ , the output  $y(n)$  is also shifted by  $N$ .

This was a simple conceptual question, so it more or less got it.

#### (c) 2D Convolution

Given the 2D matrices:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}, \quad h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The 2D convolution  $y = x * h$  (without padding, stride 1) is computed as follows:

$$y = \begin{bmatrix} -37 & -72 & -73 & -62 & -34 \\ -78 & -153 & -156 & -129 & -63 \\ -119 & -234 & -240 & -201 & -93 \\ -160 & -315 & -324 & -267 & -123 \end{bmatrix}$$

This was completely wrong. I'm not sure what it was doing here to get these answers...

#### (d) Striding and Padding

- **Stride 1, Pad 1:**

The output matrix  $y$  will have the same dimensions as the input matrix  $x$  ( $5\times 5$ ), with padding applied to handle the edges.

- **Stride 2, Pad 1:**

The output matrix  $y$  will be smaller due to the stride of 2, resulting in a  $3\times 3$  matrix.

Let's compute these explicitly.

- Worked for 1s

- Code output generated**

# Define the 2D matrices x and h"

```

# Python
Copy

# Define the 2D matrices x and h
x_2d = np.array([
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15],
    [16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25]
])

h_2d = np.array([
    [-1, -2, -1],
    [0, 0, 0],
    [1, 2, 1]
])

# Function to perform 2D convolution
def conv2d(image, kernel, stride=1, pad=0):
    # Add padding
    if pad > 0:
        image_padded = np.pad(image, pad, mode='constant')
    else:
        image_padded = image
    ...

```

Now it seems to be using code to perform the convolutions

```

image_padded = image

# Get dimensions
i_h, i_w = image_padded.shape
k_h, k_w = kernel.shape

# Calculate output dimensions
o_h = (i_h - k_h) // stride + 1
o_w = (i_w - k_w) // stride + 1

# Initialize output
output = np.zeros((o_h, o_w))

# Perform convolution
for i in range(0, o_h):
    for j in range(0, o_w):
        output[i, j] = np.sum(image_padded[i*stride:i*stride+k_h,
j*stride:j*stride+k_w] * kernel)

return output

# Compute for stride=1, pad=1
y_stride1_pad1 = conv2d(x_2d, h_2d, stride=1, pad=1)

# Compute for stride=2, pad=1
y_stride2_pad1 = conv2d(x_2d, h_2d, stride=2, pad=1)

y_stride1_pad1, y_stride2_pad1

```

Result

```

(array([[ 19.,  28.,  32.,  36.,  29.],
       [ 30.,  40.,  40.,  40.,  30.],
       [ 30.,  40.,  40.,  40.,  30.],
       [ 30.,  40.,  40.,  40.,  30.],
       [-49., -68., -72., -76., -59.]]),
array([[ 19.,  32.,  29.],
       [ 30.,  40.,  30.],
       [-49., -72., -59.]]))

```

## Results for 2D Convolution with Striding and Padding

### (c) 2D Convolution (No Padding, Stride 1)

The output matrix  $y$  for the 2D convolution without padding and with stride 1 is:

**It seems it's retrying part (c) using the code. This output cell was extremely confusing though.**

$$y = \begin{bmatrix} -37 & -72 & -73 & -62 & -34 \\ -78 & -153 & -156 & -129 & -63 \\ -119 & -234 & -240 & -201 & -93 \\ -160 & -315 & -324 & -267 & -123 \end{bmatrix}$$

It failed again, and it seems like it just repeated its old answer again.

#### (d) Striding and Padding

- **Stride 1, Pad 1:** The output matrix  $y$  is:

$$y = \begin{bmatrix} 19 & 28 & 32 & 36 & 29 \\ 30 & 40 & 40 & 40 & 30 \\ 30 & 40 & 40 & 40 & 30 \\ 30 & 40 & 40 & 40 & 30 \\ -49 & -68 & -72 & -76 & -59 \end{bmatrix}$$

This is almost correct but negative. The code worked much better, but the convolution convention is wrong.

- **Stride 2, Pad 1:** The output matrix  $y$  is:

$$y = \begin{bmatrix} 19 & 32 & 29 \\ 30 & 40 & 30 \\ -49 & -72 & -59 \end{bmatrix}$$

Again, the magnitudes are correct but just all negated. Checking its code, it is indeed a convention problem.

#### Summary

- **(a)** The convolution of the rectangular signal and the impulse response is plotted above.
- **(b)** Shifting the input signal by  $N$  results in the output being shifted by  $N$ , demonstrating the time-shifting property of convolution.
- **(c)** The 2D convolution result without padding and with stride 1 is provided.
- **(d)** The 2D convolution results with stride 1, pad 1 and stride 2, pad 1 are provided.

Make sure to repeat the question when responding before reasoning through the answer and then giving the answer. For part 3a, you must also compute  $y(n)$  as a function, not just plot it. For part 3b, the same applies. For part 3c, check your padding and calculations, ensuring to reason through it rather than trying to go through it in one go. Do the same for 3d.

**That was a complete disaster. I decided to ask it to do everything again while using the old structure of also outputting the question, as that seemed to work better. I specified what it missed in the questions for (a) and (b), and at this point I just asked it to recheck (c) and (d). The goal of this part was to finish (a) and (b), and I thought (c) and (d) may correct themselves. I hadn't realized the potential for a convention flip yet.**



### Problem Statement: 3. Understanding Convolution as Finite Impulse Response Filter

#### (a) Convolution of Rectangular Signal and Impulse Response

##### Question:

Compute and plot the convolution of  $x(n)$  and  $h(n)$ , where:

$$x(n) = \begin{cases} 1 & n = 0, 1, 2, \dots, L-1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$h(n) = \left(\frac{1}{2}\right)^n u(n) = \begin{cases} \left(\frac{1}{2}\right)^n & n = 0, 1, 2, \dots \\ 0 & \text{otherwise} \end{cases}$$

For illustrative purposes, your plot should start at -6 and end at +12.

##### Reasoning and Calculation:

The convolution  $y(n) = x(n) * h(n)$  is given by:

$$y(n) = \sum_{k=0}^{L-1} x(k)h(n-k) = \sum_{k=0}^{L-1} h(n-k)$$

since  $x(k) = 1$  for  $k = 0, 1, \dots, L-1$ .

For  $L = 7$ :

$$y(n) = \sum_{k=0}^6 \left(\frac{1}{2}\right)^{n-k} u(n-k)$$

- For  $n < 0$ ,  $y(n) = 0$  because  $u(n-k) = 0$  for all  $k$ .
- For  $0 \leq n < 7$ :

This time it's actually computing the expression

$$y(n) = \sum_{k=0}^n \left(\frac{1}{2}\right)^{n-k} = \sum_{i=0}^n \left(\frac{1}{2}\right)^i = 2 \left(1 - \left(\frac{1}{2}\right)^{n+1}\right)$$

- For  $n \geq 7$ :

$$y(n) = \sum_{k=0}^6 \left(\frac{1}{2}\right)^{n-k} = \left(\frac{1}{2}\right)^{n-6} \sum_{i=0}^6 \left(\frac{1}{2}\right)^i = \left(\frac{1}{2}\right)^{n-6} \cdot 2 \left(1 - \left(\frac{1}{2}\right)^7\right) = \left(\frac{1}{2}\right)^{n-6} \cdot \left(2 - \frac{1}{64}\right)$$

**Answer:**

The convolution  $y(n)$  is:

$$y(n) = \begin{cases} 0 & n < 0 \\ 2 \left(1 - \left(\frac{1}{2}\right)^{n+1}\right) & 0 \leq n < 7 \\ \left(\frac{1}{2}\right)^{n-6} \cdot \left(2 - \frac{1}{64}\right) & n \geq 7 \end{cases}$$

**It accurately found the expression for 7, which works. The plot was the same as earlier.**

The plot of  $y(n)$  from  $n = -6$  to  $n = 12$  is shown below:

#### Convolution Plot

#### **(b) Shifted Rectangular Signal and Convolution**

**Question:**

Now let's shift  $x(n)$  by  $N$ , i.e.,  $x_2(n) = x(n - N)$ . Let  $N = 5$ . Compute  $y_2(n) = h(n) * x_2(n)$ . Which property of the convolution can you find?

**Reasoning and Calculation:**

The convolution of  $h(n)$  and  $x_2(n) = x(n - 5)$  is:

$$y_2(n) = h(n) * x_2(n) = y(n - 5)$$

This demonstrates the **time-shifting property** of convolution: if  $x(n)$  is shifted by  $N$ , the output  $y(n)$  is also shifted by  $N$ .

**It kept the same answer for part (b) and identified the time-shifting property, which is good for this problem.**

**Answer:**

The convolution  $y_2(n)$  is:

$$y_2(n) = y(n - 5)$$

---

### (c) 2D Convolution

**Question:**

Given the 2D matrices:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}, \quad h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Evaluate  $y = x * h$ . Assume that there is no padding and stride is 1.

---

#### Reasoning and Calculation:

The 2D convolution  $y$  is computed as:

$$y[m, n] = \sum_{i=-1}^1 \sum_{j=-1}^1 x[m - i, n - j] \cdot h[i, j]$$

where  $h[i, j]$  is zero outside the given kernel.

For example, for  $m = 1$  and  $n = 1$ :

$$y[1, 1] = x[0, 0] \cdot (-1) + x[0, 1] \cdot (-2) + x[0, 2] \cdot (-1) + x[2, 0] \cdot 1 + x[2, 1] \cdot 2 + x[2, 2] \cdot 1$$

We compute  $y$  for all valid  $m$  and  $n$ .

**Answer:** Here, it is showing its work a little bit, and here's where I realized its convention was flipped. This makes sense in the context of deep learning since we typically assume it's already flipped like Mistral did.

The output matrix  $y$  is:

$$y = \begin{bmatrix} -37 & -72 & -73 & -62 & -34 \\ -78 & -153 & -156 & -129 & -63 \\ -119 & -234 & -240 & -201 & -93 \\ -160 & -315 & -324 & -267 & -123 \end{bmatrix}$$

---

Again it completely fails, outputting the old wrong matrix. This is because it didn't use code for the math

#### (d) Striding and Padding

**Question:**

Evaluate  $y$  for the following cases:

- Stride 1, Pad 1
- Stride 2, Pad 1

---

**Reasoning and Calculation:**

- **Stride 1, Pad 1:**

Pad  $x$  with a border of zeros (1 pixel). The output size is the same as the input size ( $5 \times 5$ ).

- **Stride 2, Pad 1:**

Pad  $x$  with a border of zeros (1 pixel). The output size is reduced due to the stride of 2.

We compute the convolutions for both cases.

---

**Answer:**

- **Stride 1, Pad 1:**

$$y = \begin{bmatrix} -7 & -12 & -12 & -10 & -5 \\ -12 & -24 & -24 & -20 & -10 \\ -12 & -24 & -24 & -20 & -10 \\ -10 & -20 & -20 & -17 & -8 \\ -5 & -10 & -10 & -8 & -4 \end{bmatrix}$$

- **Stride 2, Pad 1:**

Now it's also trying to not use code for part (d) which makes it fail entirely

$$y = \begin{bmatrix} -7 & -12 & -5 \\ -12 & -24 & -10 \\ -5 & -10 & -4 \end{bmatrix}$$

### More failing due to not using code

Let me compute these explicitly for accuracy.

- Worked for 1s

- Code output generated      "# Recompute 2D convolution with no padding and stride 1"

 Python

 Copy

```
# Recompute 2D convolution with no padding and stride 1
y_no_pad_stride1 = conv2d(x_2d, h_2d, stride=1, pad=0)

# Recompute 2D convolution with stride 1 and pad 1
y_stride1_pad1_corrected = conv2d(x_2d, h_2d, stride=1, pad=1)

# Recompute 2D convolution with stride 2 and pad 1
y_stride2_pad1_corrected = conv2d(x_2d, h_2d, stride=2, pad=1)

y_no_pad_stride1, y_stride1_pad1_corrected, y_stride2_pad1_corrected
```

Result

 Copy

```
(array([[40., 40., 40.],
       [40., 40., 40.],
       [40., 40., 40.]]),
 array([[ 19.,  28.,  32.,  36.,  29.],
       [ 30.,  40.,  40.,  40.,  30.],
       [ 30.,  40.,  40.,  40.,  30.],
       [ 30.,  40.,  40.,  40.,  30.],
       [-49., -68., -72., -76., -59.]]),
 array([[ 19.,  32.,  29.],
       [ 30.,  40.,  30.],
       [-49., -72., -59.]]))
```

### Corrected Results for 2D Convolution

Now it actually uses its old function to compute the convolutions. This gives the correct results for everything except for the convention, which causes everything to be negated.

### (c) 2D Convolution (No Padding, Stride 1)

**Answer:**

The output matrix  $y$  for the 2D convolution without padding and with stride 1 is:

$$y = \begin{bmatrix} 40 & 40 & 40 \\ 40 & 40 & 40 \\ 40 & 40 & 40 \end{bmatrix}$$

---

### (d) Striding and Padding

- **Stride 1, Pad 1:**

**Answer:**

The output matrix  $y$  is:

$$y = \begin{bmatrix} 19 & 28 & 32 & 36 & 29 \\ 30 & 40 & 40 & 40 & 30 \\ 30 & 40 & 40 & 40 & 30 \\ 30 & 40 & 40 & 40 & 30 \\ -49 & -68 & -72 & -76 & -59 \end{bmatrix}$$

- **Stride 2, Pad 1:**

**Answer:**

The output matrix  $y$  is:

$$y = \begin{bmatrix} 19 & 32 & 29 \\ 30 & 40 & 30 \\ -49 & -72 & -59 \end{bmatrix}$$

---

**Its convention is just flipped but its magnitudes are correct. Now I prompt it to use the mathematical definition of the convolution.**

For parts (c) and (d), ensure you are applying the kernel to the right elements. Recall that the kernel is not applied straight on like it is when simplified in typical applications. Instead, it is the mathematical definition. Resolve 3(c) using this knowledge.



### Problem Statement: 3. Understanding Convolution as Finite Impulse Response Filter (c)

#### Question:

Given the 2D matrices:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}, \quad h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Evaluate  $y = x * h$ . Assume that there is no padding and stride is 1.

---

#### Reasoning and Calculation:

The mathematical definition of 2D convolution is:

$$y[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x[i, j] \cdot h[m - i, n - j]$$

Given the kernel  $h$  and the input  $x$ , we need to flip the kernel both horizontally and vertically before applying it to the input matrix. The flipped kernel  $h'$  is:

$$h' = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Now, we slide this flipped kernel over the input matrix  $x$  and compute the sum of element-wise products at each position.

---

#### Answer:

Let's compute  $y$  explicitly using the flipped kernel:

- For  $y[1, 1]$ :

$$y[1, 1] = x[0, 0] \cdot 1 + x[0, 1] \cdot 2 + x[0, 2] \cdot 1 + x[1, 0] \cdot 0 + x[1, 1] \cdot 0 + x[1, 2] \cdot 0 + x[2, 0] \cdot ($$

$$y[1, 1] = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 1 + 6 \cdot 0 + 7 \cdot 0 + 8 \cdot 0 + 11 \cdot (-1) + 12 \cdot (-2) + 13 \cdot (-1) = 1 -$$

- For  $y[1, 2]$ :

$$y[1, 2] = x[0, 1] \cdot 1 + x[0, 2] \cdot 2 + x[0, 3] \cdot 1 + x[1, 1] \cdot 0 + x[1, 2] \cdot 0 + x[1, 3] \cdot 0 + x[2, 1] \cdot ($$

$$y[1, 2] = 2 \cdot 1 + 3 \cdot 2 + 4 \cdot 1 + 7 \cdot 0 + 8 \cdot 0 + 9 \cdot 0 + 12 \cdot (-1) + 13 \cdot (-2) + 14 \cdot (-1) = 2 -$$

We continue this process for all valid positions. Let's compute the full output matrix  $y$ .

Worked for 1s



### Corrected Answer for 3(c): 2D Convolution (No Padding, Stride 1)

The output matrix  $y$  for the 2D convolution without padding and with stride 1, using the mathematically correct flipped kernel, is:

$$y = \begin{bmatrix} -40 & -40 & -40 \\ -40 & -40 & -40 \\ -40 & -40 & -40 \end{bmatrix}$$

This result reflects the correct application of the convolution operation as defined mathematically. If you'd like to proceed with part (d) using the same approach, let me know!

**Now it got the correct answer, as it corrected its code.**

Yes, please solve 3(d) using the same approach now.



### Problem Statement: 3. Understanding Convolution as Finite Impulse Response Filter (d)

#### Question:

Evaluate  $y$  for the following cases:

- Stride 1, Pad 1
- Stride 2, Pad 1

### Reasoning and Calculation:

We use the flipped kernel:

$$h' = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

#### Stride 1, Pad 1

- Pad  $x$  with a border of zeros (1 pixel).
- Slide the flipped kernel over the padded  $x$  with a stride of 1.

#### Stride 2, Pad 1

- Pad  $x$  with a border of zeros (1 pixel).
- Slide the flipped kernel over the padded  $x$  with a stride of 2.

### Answer:

Let's compute the convolutions for both cases using the flipped kernel.

- Worked for 1s

- Code output generated

"# Compute for stride 1, pad 1"



Copy

```
# Compute for stride 1, pad 1
y_stride1_pad1_flipped = conv2d_flipped(x_2d, h_flipped, stride=1, pad=1)

# Compute for stride 2, pad 1
y_stride2_pad1_flipped = conv2d_flipped(x_2d, h_flipped, stride=2, pad=1)

y_stride1_pad1_flipped, y_stride2_pad1_flipped
```

 Copy

```
(array([[-19., -28., -32., -36., -29.],
       [-30., -40., -40., -40., -30.],
       [-30., -40., -40., -40., -30.],
       [-30., -40., -40., -40., -30.],
       [ 49.,  68.,  72.,  76.,  59.]]),
 array([[[-19., -28., -32., -36., -29.],
        [-30., -40., -40., -40., -30.],
        [-30., -40., -40., -40., -30.],
        [-30., -40., -40., -40., -30.],
        [ 49.,  68.,  72.,  76.,  59.]]]))
```

It seems there was an issue with the stride implementation in the previous code. Let's correct the stride logic and recompute the results for both cases.

- Worked for 1s

 Code output generated      "# Corrected function to perform 2D convolution with stride"

 Python

 Copy

```
# Corrected function to perform 2D convolution with stride
def conv2d_flipped_stride(image, kernel, stride=1, pad=0):
    # Add padding
    if pad > 0:
        image_padded = np.pad(image, pad, mode='constant')
    else:
        image_padded = image

    # Get dimensions
    i_h, i_w = image_padded.shape
    k_h, k_w = kernel.shape

    # Calculate output dimensions
    o_h = (i_h - k_h) // stride + 1
    o_w = (i_w - k_w) // stride + 1

    # Initialize output
    output = np.zeros((o_h, o_w))

    # Perform convolution
    for m in range(0, o_h):
        for n in range(0, o_w):
```

```

        output[m, n] = np.sum(image_padded[m*stride:m*stride+k_h,
n*stride:n*stride+k_w] * kernel)

    return output

# Compute for stride 1, pad 1
y_stride1_pad1_corrected = conv2d_flipped_stride(x_2d, h_flipped, stride=1,
pad=1)

# Compute for stride 2, pad 1
y_stride2_pad1_corrected = conv2d_flipped_stride(x_2d, h_flipped, stride=2,
pad=1)

y_stride1_pad1_corrected, y_stride2_pad1_corrected

```

Result

 Copy

```

(array([[-19., -28., -32., -36., -29.],
       [-30., -40., -40., -40., -30.],
       [-30., -40., -40., -40., -30.],
       [-30., -40., -40., -40., -30.],
       [ 49.,  68.,  72.,  76.,  59.]]),
array([[[-19., -32., -29.],
       [-30., -40., -30.],
       [ 49.,  72.,  59.]]]))

```

## Corrected Results for 3(d): Striding and Padding

### Stride 1, Pad 1

The output matrix  $y$  is:

$$y = \begin{bmatrix} -19 & -28 & -32 & -36 & -29 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ -30 & -40 & -40 & -40 & -30 \\ 49 & 68 & 72 & 76 & 59 \end{bmatrix}$$

---

### Stride 2, Pad 1

The output matrix  $y$  is:

$$y = \begin{bmatrix} -19 & -32 & -29 \\ -30 & -40 & -30 \\ 49 & 72 & 59 \end{bmatrix}$$

**It now got the correct answers for everything. It just needed a convention correction, which again may be due to the context of deep learning, where we generally assume the kernel is already flipped.**

Solve #4a. Make sure to restate the problem and show your work.

**I went back to asking for a part at a time, since this seemed to work better**



#### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (a)

##### Question:

A convolutional layer has 4 architectural hyperparameters: the filter size  $K$ , the padding size  $P$ , the stride step size  $S$ , and the number of filters  $F$ . How many weights and biases are in this convolutional layer? And what is the shape of the output feature that this convolutional layer produces?

##### Reasoning and Calculation:

###### Weights and Biases in a Convolutional Layer

- **Weights:** Each filter is a 3D tensor of size  $K \times K \times C$ , where  $C$  is the number of input channels. If there are  $F$  filters, the total number of weights is:

$$\text{Total Weights} = F \times (K \times K \times C)$$

- **Biases:** Each filter has one bias term. Therefore, the total number of biases is:

$$\text{Total Biases} = F$$

###### Output Feature Shape

- **Input Feature Shape:**

Let the input feature shape be  $W \times H \times C$ .

- **Output Width and Height:**

The output width  $W'$  and height  $H'$  are calculated as:

$$W' = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1$$

$$H' = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1$$

- **Output Feature Shape:** The output feature shape is  $W' \times H' \times F$ .

**Answer:**

- **Number of Weights:**

$$F \times (K \times K \times C)$$

- **Number of Biases:**

$$F$$

- **Shape of Output Feature:**

$$\left( \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1 \right) \times \left( \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1 \right) \times F$$

It solved this one correctly in 1 go, which was expected but still good

Solve 4b



#### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (b)

**Question:**

A max pooling layer has 2 architectural hyperparameters: the stride step size  $S$  and the "filter size"  $K$ . What is the output feature shape that this pooling layer produces?

### Reasoning and Calculation:

#### Max Pooling Layer Output Shape

- **Input Feature Shape:**

Let the input feature shape be  $W \times H \times C$ .

- **Output Width and Height:**

The output width  $W'$  and height  $H'$  are calculated as:

$$W' = \left\lfloor \frac{W - K}{S} \right\rfloor + 1$$

$$H' = \left\lfloor \frac{H - K}{S} \right\rfloor + 1$$

- **Output Feature Shape:** The number of channels  $C$  remains unchanged. Therefore, the output feature shape is  $W' \times H' \times C$ .

#### Answer:

The shape of the output feature produced by the max pooling layer is:

$$\left( \left\lfloor \frac{W - K}{S} \right\rfloor + 1 \right) \times \left( \left\lfloor \frac{H - K}{S} \right\rfloor + 1 \right) \times C$$

**It again solved this problem in 1 shot, which is good again (it was very similar to the last part).**

Solve 4c



#### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (c)

##### Question:

Let's assume that we have the CNN model which consists of  $L$  successive convolutional layers and the filter size is  $K$  and the stride step size is 1 for every convolutional layer. Then what is the receptive field size of the last output?

## Reasoning and Calculation:

### Receptive Field Size

- **Receptive Field:**

The receptive field of a neuron in a convolutional neural network is the region in the input space that affects the value of that neuron. For a CNN with  $L$  successive convolutional layers, each with filter size  $K$  and stride 1, the receptive field size grows linearly with the number of layers.

- **Receptive Field Calculation:**

For a single convolutional layer with filter size  $K$  and stride 1, the receptive field size is  $K \times K$ .

For  $L$  successive layers, the receptive field size  $R$  is:

$$R = 1 + (L - 1) \times (K - 1)$$

This is because each layer adds  $K - 1$  to the receptive field size.

---

### Answer:

The receptive field size of the last output is:

$$1 + (L - 1) \times (K - 1)$$

**It's doing really well on this question. It's not very hard calculations so I don't expect a failure due to that, but it seems good conceptually.**

Solve 4d



### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (d)

#### Question:

Consider a downsampling layer (e.g., pooling layer and strided convolution layer). In this problem, we investigate pros and cons of downsampling layers. This layer reduces the output feature resolution and this implies that the output features lose a certain amount of spatial information. Therefore, when we design CNNs, we usually increase the channel length to

compensate for this loss. If we apply a max pooling layer with a kernel size of 2 and a stride of 2, how much does the receptive field increase? Explain the advantage of decreasing the output feature resolution with the perspective of reducing the amount of computation.

---

### Reasoning and Calculation:

#### Receptive Field Increase

- **Receptive Field:**

The receptive field of a neuron is the region in the input space that affects its value. When a max pooling layer with a kernel size of 2 and a stride of 2 is applied, the spatial resolution of the feature map is halved.

- **Receptive Field Calculation:**

Applying a max pooling layer with a kernel size of 2 and a stride of 2 effectively doubles the receptive field size of each neuron in the subsequent layer. This is because each neuron in the subsequent layer now corresponds to a  $2 \times 2$  region in the previous layer.

#### Advantage of Decreasing Output Feature Resolution

- **Computational Efficiency:**

Reducing the spatial resolution of the feature maps decreases the number of computations required in subsequent layers. This is because the number of neurons in each feature map is reduced, leading to fewer operations during convolution and pooling.

- **Memory Efficiency:**

Smaller feature maps require less memory, which can be beneficial for training and deploying models on resource-constrained devices.

---

Answer: **It went in depth with its answer, but everything explained was relevant.**  
**Another largely conceptual question, so it answered very well.**

- **Receptive Field Increase:**

The receptive field increases by a factor of **2**.

- **Advantage:**

Decreasing the output feature resolution reduces the computational and memory requirements, making the model more efficient. This is particularly useful for deep networks and resource-constrained environments.

Solve 4e



### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (e)

#### Question:

Let's take a real example. We are going to describe a convolutional neural net using the following pieces:

- **CONV3-F** denotes a convolutional layer with  $F$  different filters, each of size  $3 \times 3 \times C$ , where  $C$  is the depth (i.e., number of channels) of the activations from the previous layer. Padding is 1, and stride is 1.
- **POOL2** denotes a  $2 \times 2$  max-pooling layer with stride 2 (pad 0).
- **FLATTEN** just turns whatever shape input tensor into a one-dimensional array with the same values in it.
- **FC-K** denotes a fully-connected layer with  $K$  output neurons.

Now, we are going to use this network to do inference on a single input. Fill in the missing entries in this table of the size of the activations at each layer, and the number of parameters at each layer.

Layer	Number of Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10		$28 \times 28 \times 10$
POOL2	0	
CONV3-10	$3 \times 3 \times 10 \times 10 + 10$	
POOL2	0	
FLATTEN	0	490
FC-3		3

Since I haven't used Mistral before, I was wondering how well it would do with the table. It seems it's good with tables and was able to accurately get it from the PDF.

### Reasoning and Calculation:

#### CONV3-10 (First Layer)

- **Number of Parameters:** Each filter is  $3 \times 3 \times 1$  (since the input has 1 channel), and there are 10 filters. Each filter also has a bias term.

$$\text{Number of Parameters} = (3 \times 3 \times 1) \times 10 + 10 = 90 + 10 = 100$$

- **Dimension of Activations:** The input is  $28 \times 28 \times 1$ . With padding 1 and stride 1, the output dimension remains  $28 \times 28$ , but with 10 channels.

$$\text{Dimension of Activations} = 28 \times 28 \times 10$$

#### POOL2 (First Pooling Layer)

- **Number of Parameters:** Pooling layers have no parameters.

$$\text{Number of Parameters} = 0$$

- **Dimension of Activations:** The input is  $28 \times 28 \times 10$ . With a  $2 \times 2$  kernel and stride 2, the output dimension is halved.

$$\text{Dimension of Activations} = 14 \times 14 \times 10$$

#### CONV3-10 (Second Layer)

- **Number of Parameters:** Each filter is  $3 \times 3 \times 10$  (since the input has 10 channels), and there are 10 filters. Each filter also has a bias term.

$$\text{Number of Parameters} = (3 \times 3 \times 10) \times 10 + 10 = 900 + 10 = 910$$

- **Dimension of Activations:** The input is  $14 \times 14 \times 10$ . With padding 1 and stride 1, the output dimension remains  $14 \times 14$ , but with 10 channels.

$$\text{Dimension of Activations} = 14 \times 14 \times 10$$

It's doing some calculations, matching the expressions in the answer key. It decided to fill in the blanks with the real number, but here we can see the dimensions are correct.

### POOL2 (Second Pooling Layer)

- **Number of Parameters:** Pooling layers have no parameters.

$$\text{Number of Parameters} = 0$$

- **Dimension of Activations:** The input is  $14 \times 14 \times 10$ . With a  $2 \times 2$  kernel and stride 2, the output dimension is halved.

$$\text{Dimension of Activations} = 7 \times 7 \times 10$$

### FLATTEN

- **Number of Parameters:** Flattening has no parameters.

$$\text{Number of Parameters} = 0$$

- **Dimension of Activations:** The input is  $7 \times 7 \times 10$ . Flattening results in a 1D array.

$$\text{Dimension of Activations} = 7 \times 7 \times 10 = 490$$

### FC-3

- **Number of Parameters:** The fully-connected layer has 490 inputs and 3 outputs. Each connection has a weight, and each output has a bias.

$$\text{Number of Parameters} = 490 \times 3 + 3 = 1470 + 3 = 1473$$

- **Dimension of Activations:** The output dimension is simply the number of neurons.

$$\text{Dimension of Activations} = 3$$

---

Answer:

All of this is correct

Layer	Number of Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10	100	$28 \times 28 \times 10$
POOL2	0	$14 \times 14 \times 10$
CONV3-10	910	$14 \times 14 \times 10$
POOL2	0	$7 \times 7 \times 10$
FLATTEN	0	490
FC-3	1473	3

It did this part perfectly! It shows that it knows how to use a table, and it also got the concepts from the question right.

Solve 4f



#### Problem Statement: 4. Feature Dimensions of Convolutional Neural Network (f)

##### Question:

Consider a new architecture:

CONV2-3 → ReLU → CONV2-3 → ReLU → GAP (Global Average Pool) → FC-3

Each CONV2-3 layer has stride of 1 and padding of 1. Note that we use circular padding (i.e., wraparound) for this task. Instead of using zeros, circular padding makes it as though the virtual column before the first column is the last column and the virtual row before the first row is the last row, treating the image as though it was on a torus.

Here, the GAP layer is an average pooling layer that computes the per-channel means over the entire input image.

You are told the behavior for an input image with a horizontal edge,  $\mathbf{x}_1$ , and an image with a vertical edge,  $\mathbf{x}_2$ :

$$\mathbf{x}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Suppose we knew that the GAP output features when fed  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are:

$$\mathbf{g}_1 = f(\mathbf{x}_1) = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{g}_2 = f(\mathbf{x}_2) = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$$

Use what you know about the invariances/equivariances of convolutional nets to compute the  $\mathbf{g}_i$  corresponding to the following  $\mathbf{x}_i$  images:

$$\mathbf{x}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

**Pretty long question, but it copies it correctly.**

**Reasoning and Calculation:**

#### Understanding the Architecture and GAP Outputs

- CONV2-3 Layers:**

Each CONV2-3 layer has 3 filters of size  $2 \times 2$  with stride 1 and circular padding. The ReLU activation introduces non-linearity but preserves spatial structure.

- GAP Layer:**

The Global Average Pooling (GAP) layer computes the mean of each channel over the entire spatial dimension, resulting in a 3D vector.

#### Invariances and Equivariances

- **Translation Invariance:**

Convolutional layers are equivariant to translation. If the input is shifted, the output feature maps are shifted accordingly, but the GAP output remains invariant to translation because it averages over the entire spatial dimension.

- **Rotation Invariance:**

Convolutional layers are not inherently rotation-invariant, but circular padding can make them more robust to certain rotations.

### Computing $\mathbf{g}_3$ and $\mathbf{g}_4$

- $\mathbf{x}_3$ :  $\mathbf{x}_3$  is a vertical edge shifted to the right. Due to translation equivariance, the GAP output should be similar to  $\mathbf{g}_2$ , but since GAP is invariant to translation, the output remains the same:

$$\mathbf{g}_3 = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$$

- $\mathbf{x}_4$ :  $\mathbf{x}_4$  is a horizontal edge shifted down. Due to translation equivariance, the GAP output should be similar to  $\mathbf{g}_1$ , but since GAP is invariant to translation, the output remains the same:

$$\mathbf{g}_4 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}$$

**It gets the correct answers using the right reasoning. Doing very well so far on this homework.**

**Answer:**

- For  $\mathbf{x}_3$ :

$$\mathbf{g}_3 = \boxed{\begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}}$$

- For  $\mathbf{x}_4$ :

$$\mathbf{g}_4 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}$$

Solve 7a

### Problem Statement: 7. Weights and Gradients in a CNN (a)

#### Question:

Derive the gradient to the weight matrix  $d\mathbf{w} \in \mathbb{R}^{k \times k}$ ,

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}$$

where  $dw_{h,l}$  denotes  $\frac{\partial \mathcal{L}}{\partial w_{h,l}}$ . Also, derive the weight after one SGD step with a batch of a single image.

#### Reasoning and Calculation:

##### Gradient Derivation

Given the convolution operation:

$$y_{i,j} = \sum_{h=1}^k \sum_{l=1}^k x_{i+h-1,j+l-1} w_{h,l}$$

The gradient of the loss  $\mathcal{L}$  with respect to the weight  $w_{h,l}$  is:

$$\frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i=1}^m \sum_{j=1}^m \frac{\partial \mathcal{L}}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial w_{h,l}} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}$$

This is correct so far

Thus, the gradient matrix  $d\mathbf{w}$  is:

$$dw_{h,l} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}$$

### SGD Update

For a single image, the SGD update for the weight matrix  $\mathbf{w}$  is:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot d\mathbf{w}$$

where  $\eta$  is the learning rate.

**Answer:**

- **Gradient to the Weight Matrix:**

$$dw_{h,l} = \boxed{\sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}}$$

This is the correct update, but it didn't simplify it into a convolution.

- **Weight Update:**

$$\mathbf{w} \leftarrow \boxed{\mathbf{w} - \eta \cdot d\mathbf{w}}$$

This is the proper weight update formula but it doesn't plug it in (since it doesn't have a good formula).

can you rewrite the  $d\mathbf{w}$  matrix as a whole more compactly? then also use that value in the weight update formula.

I try to prompt it to simplify.

### Compact Formulation for $d\mathbf{w}$ and Weight Update

#### Gradient Matrix $d\mathbf{w}$

The gradient matrix  $d\mathbf{w}$  can be expressed compactly as:

$$d\mathbf{w} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i,j}$$

where  $\mathbf{X}_{i,j}$  is the  $k \times k$  submatrix of  $\mathbf{X}$  starting at  $(i, j)$ :

$$\mathbf{X}_{i,j} = \begin{bmatrix} x_{i,j} & x_{i,j+1} & \cdots & x_{i,j+k-1} \\ x_{i+1,j} & x_{i+1,j+1} & \cdots & x_{i+1,j+k-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i+k-1,j} & x_{i+k-1,j+1} & \cdots & x_{i+k-1,j+k-1} \end{bmatrix}$$

### Weight Update Formula

For a single image, the SGD update for the weight matrix  $\mathbf{w}$  is:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot d\mathbf{w}$$

where  $d\mathbf{w} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i,j}$ .

### Answer:

- Gradient to the Weight Matrix:

$$d\mathbf{w} = \boxed{\sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i,j}}$$

- Weight Update: This is also correct, but still not simplified in the form of the convolution. It doesn't seem to recognize this as a convolution operation.

$$\mathbf{w} \leftarrow \boxed{\mathbf{w} - \eta \cdot \left( \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i,j} \right)}$$

Please use the context given at the start of the problem to more simply write the summation you initially found

I saw that in the question this pattern was identified as a convolution, so I thought that may help it.

### Compact Formulation for $d\mathbf{w}$ and Weight Update

#### Gradient Matrix $d\mathbf{w}$

Given the convolution operation:

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}$$

where  $\mathbf{Y} \in \mathbb{R}^{m \times m}$ ,  $\mathbf{X} \in \mathbb{R}^{n \times n}$ , and  $\mathbf{w} \in \mathbb{R}^{k \times k}$ .

The gradient of the loss  $\mathcal{L}$  with respect to the weight matrix  $\mathbf{w}$  can be written as:

$$d\mathbf{w} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i:i+k-1, j:j+k-1}$$

Here,  $\mathbf{X}_{i:i+k-1, j:j+k-1}$  is the  $k \times k$  block of  $\mathbf{X}$  starting at position  $(i, j)$ .

#### Weight Update Formula

For a single image, the SGD update for the weight matrix  $\mathbf{w}$  is:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot d\mathbf{w}$$

where  $d\mathbf{w}$  is the gradient matrix defined above.

#### Answer:

- Gradient to the Weight Matrix:

$$d\mathbf{w} = \boxed{\sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i:i+k-1, j:j+k-1}}$$

- **Weight Update:**

$$\mathbf{w} \leftarrow \boxed{\mathbf{w} - \eta \cdot \left( \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i:i+k-1, j:j+k-1} \right)}$$

**It unfortunately gave the same answer as last time. I tried prompting it again, choosing to ask it to repeat the context where it was identified as a convolution as well, in case that helped it like it did before.**

Please restate the context given for problem 7 as a whole, before the problem statement for 7a. Then check how you can use this context to simplify your expression.



### Context for Problem 7:

In this homework assignment, we aim to accomplish two objectives. Firstly, we seek to comprehend that the weights of a CNN are a weighted average of the images in the dataset. This understanding is crucial in answering a commonly asked question: does a CNN memorize images during the training process? Additionally, we will analyze the impact of spatial weight sharing in convolution layers. Secondly, we aim to gain an understanding of the behavior of max-pooling and avg-pooling in backpropagation. By accomplishing these objectives, we will enhance our knowledge of CNNs and their functioning.

Let's consider a convolution layer with input matrix  $\mathbf{X} \in \mathbb{R}^{n \times n}$ ,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}$$

weight matrix  $\mathbf{w} \in \mathbb{R}^{k \times k}$ ,

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,k} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & \cdots & w_{k,k} \end{bmatrix}$$

and output matrix  $\mathbf{Y} \in \mathbb{R}^{m \times m}$ ,

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,m} \end{bmatrix}$$

For simplicity, we assume the number of input channels (of  $\mathbf{X}$ ) and the number of output channels (of  $\mathbf{Y}$ ) are both 1, and the convolutional layer has no padding and a stride of 1.

Then for all  $i, j$ ,

$$y_{i,j} = \sum_{h=1}^k \sum_{l=1}^k x_{i+h-1, j+l-1} w_{h,l}$$

or

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}$$

where  $*$  refers to the convolution operation. For simplicity, we omit the bias term in this question.

Suppose the final loss is  $\mathcal{L}$ , and the upstream gradient is  $d\mathbf{Y} \in \mathbb{R}^{m \times m}$ ,

$$d\mathbf{Y} = \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix}$$

where  $dy_{i,j}$  denotes  $\frac{\partial \mathcal{L}}{\partial y_{i,j}}$ .

### Problem Statement: 7. Weights and Gradients in a CNN (a)

#### Question:

Derive the gradient to the weight matrix  $d\mathbf{w} \in \mathbb{R}^{k \times k}$ ,

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}$$

where  $dw_{h,l}$  denotes  $\frac{\partial \mathcal{L}}{\partial w_{h,l}}$ . Also, derive the weight after one SGD step with a batch of a single image.

### Simplified Expression for $d\mathbf{w}$

Given the convolution operation  $\mathbf{Y} = \mathbf{X} * \mathbf{w}$ , the gradient of the loss  $\mathcal{L}$  with respect to the weight matrix  $\mathbf{w}$  can be written as:

$$d\mathbf{w} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i:i+k-1, j:j+k-1}$$

where  $\mathbf{X}_{i:i+k-1, j:j+k-1}$  is the  $k \times k$  block of  $\mathbf{X}$  starting at position  $(i, j)$ .

Since the convolution operation is applied with a stride of 1 and no padding, the indices  $i$  and  $j$  range over all valid positions where the  $k \times k$  block fits within  $\mathbf{X}$ .

**Answer:**

- **Gradient to the Weight Matrix:**

$$d\mathbf{w} = \boxed{\sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot \mathbf{X}_{i:i+k-1, j:j+k-1}}$$

This still did not end up helping it! It does not seem to understand that this is a convolution operation.

- **Weight Update:**

$$\mathbf{w} \leftarrow \boxed{\mathbf{w} - \eta \cdot d\mathbf{w}}$$

This expression is already simplified given the context and the convolution operation described. The summation is over all valid positions of the  $k \times k$  block in the input matrix  $\mathbf{X}$ .

Give the expression for a single element of  $d\mathbf{w}$  again. Then try to simplify this into a single convolution operation for the entire matrix, similarly to how the context did for  $\mathbf{Y} = \mathbf{X} * \mathbf{w}$

**Here, I just told it to turn it into a convolution to see if it would perform it without complaint.**

### Expression for a Single Element of $d\mathbf{w}$

Given the convolution operation:

$$y_{i,j} = \sum_{h=1}^k \sum_{l=1}^k x_{i+h-1,j+l-1} w_{h,l}$$

The gradient of the loss  $\mathcal{L}$  with respect to a single weight  $w_{h,l}$  is:

$$dw_{h,l} = \frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i=1}^m \sum_{j=1}^m \frac{\partial \mathcal{L}}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial w_{h,l}} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}$$

### Simplifying to a Single Convolution Operation

The entire gradient matrix  $d\mathbf{w}$  can be expressed as a convolution operation between the input matrix  $\mathbf{X}$  and the gradient matrix  $d\mathbf{Y}$ . Specifically,  $d\mathbf{w}$  is the result of a **full convolution** (also known as a transposed convolution or a cross-correlation with a flipped kernel) between  $\mathbf{X}$  and  $d\mathbf{Y}$ :

$$d\mathbf{w} = \mathbf{X} *_\text{full} \text{flip}(d\mathbf{Y})$$

Here,  $*_\text{full}$  denotes the full convolution operation, and  $\text{flip}(d\mathbf{Y})$  means flipping  $d\mathbf{Y}$  both horizontally and vertically.

**Here, I realized why it wasn't turning it into a convolution before. It was using the mathematical convolution operation, where the kernel is not considered to be flipped beforehand, so it thought it had to flip  $d\mathbf{Y}$  to use it in the convolution, which it does here since I forced it to use one.**

**Answer:** • Single Element of  $d\mathbf{w}$ :

$$dw_{h,l} = \boxed{\sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}}$$

- Entire Gradient Matrix  $d\mathbf{w}$ :

$$d\mathbf{w} = \boxed{\mathbf{X} *_{\text{full}} \text{flip}(d\mathbf{Y})}$$

This expression shows that the gradient to the weight matrix can be computed as a full convolution between the input matrix and the flipped gradient matrix.

Note that in this case, it is ok to assume that the convolution operation is applied as if the kernel is already flipped, and use this convention.

**Here, I corrected its convention. This wasn't clear in the problem, and the operation was used inconsistently in this homework, so I specified here.**

### Expression for a Single Element of $d\mathbf{w}$

Given the convolution operation:

$$y_{i,j} = \sum_{h=1}^k \sum_{l=1}^k x_{i+h-1,j+l-1} w_{h,l}$$

The gradient of the loss  $\mathcal{L}$  with respect to a single weight  $w_{h,l}$  is:

$$dw_{h,l} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1,j+l-1}$$

### Simplifying to a Single Convolution Operation

Given the context and the assumption that the convolution operation is applied as if the kernel is already flipped, the entire gradient matrix  $d\mathbf{w}$  can be expressed as a **convolution** between the input matrix  $\mathbf{X}$  and the gradient matrix  $d\mathbf{Y}$ :

$$d\mathbf{w} = d\mathbf{Y} * \mathbf{X}$$

Here,  $*$  denotes the convolution operation, and the kernel  $\mathbf{X}$  is treated as if it is already flipped.

**Answer:**

- **Single Element of  $d\mathbf{w}$ :**

$$dw_{h,l} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1, j+l-1}$$

- **Entire Gradient Matrix  $d\mathbf{w}$ :**

$$d\mathbf{w} = [d\mathbf{Y} * \mathbf{X}]$$

This expression shows that the gradient to the weight matrix can be computed as a convolution between the gradient matrix  $d\mathbf{Y}$  and the input matrix  $\mathbf{X}$ .

**Now it got to the answer, after quite a bit of prompting.**

Now solve 7b. Make sure to repeat the problem and then show step by step how you got to your answer.



### Problem Statement: 7. Weights and Gradients in a CNN (b)

#### Question:

Now, let's consider the backpropagation through a max-pooling layer. Suppose we have a max-pooling layer with a kernel size of  $2 \times 2$  and a stride of 2. The input to the max-pooling layer is  $\mathbf{A} \in \mathbb{R}^{4 \times 4}$ ,

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

and the output is  $\mathbf{B} \in \mathbb{R}^{2 \times 2}$ ,

$$\mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

**This wasn't even 7b. I'm not sure where it got this question from, but it wasn't correct, so I'll skip analysis for this.**

where  $b_{i,j} = \max(a_{2i-1,2j-1}, a_{2i-1,2j}, a_{2i,2j-1}, a_{2i,2j})$ .

Suppose the upstream gradient is  $d\mathbf{B} \in \mathbb{R}^{2 \times 2}$ ,

$$d\mathbf{B} = \begin{bmatrix} db_{1,1} & db_{1,2} \\ db_{2,1} & db_{2,2} \end{bmatrix}$$

where  $db_{i,j}$  denotes  $\frac{\partial \mathcal{L}}{\partial b_{i,j}}$ .

Derive the gradient to the input  $d\mathbf{A} \in \mathbb{R}^{4 \times 4}$ ,

$$d\mathbf{A} = \begin{bmatrix} da_{1,1} & da_{1,2} & da_{1,3} & da_{1,4} \\ da_{2,1} & da_{2,2} & da_{2,3} & da_{2,4} \\ da_{3,1} & da_{3,2} & da_{3,3} & da_{3,4} \\ da_{4,1} & da_{4,2} & da_{4,3} & da_{4,4} \end{bmatrix}$$

where  $da_{i,j}$  denotes  $\frac{\partial \mathcal{L}}{\partial a_{i,j}}$ .

### Step-by-Step Reasoning:

#### Max-Pooling Forward Pass

For each element in the output  $\mathbf{B}$ :

$$b_{i,j} = \max(a_{2i-1,2j-1}, a_{2i-1,2j}, a_{2i,2j-1}, a_{2i,2j})$$

#### Max-Pooling Backward Pass

The gradient  $d\mathbf{A}$  is computed by distributing the gradient  $d\mathbf{B}$  to the input elements that were the maximum in their respective  $2 \times 2$  blocks during the forward pass.

- For each  $b_{i,j}$ , identify the index  $(m, n)$  in the  $2 \times 2$  block of  $\mathbf{A}$  that was the maximum:

$$(m, n) = \arg \max(a_{2i-1,2j-1}, a_{2i-1,2j}, a_{2i,2j-1}, a_{2i,2j})$$

- The gradient for the input  $\mathbf{A}$  is:

$$da_{m,n} = db_{i,j}$$

and all other elements in the  $2 \times 2$  block are zero.

### Example Calculation

Let's assume the following maxima for each block:

- $b_{1,1}$  corresponds to  $a_{1,1}$
- $b_{1,2}$  corresponds to  $a_{1,4}$
- $b_{2,1}$  corresponds to  $a_{3,1}$
- $b_{2,2}$  corresponds to  $a_{4,4}$

Then:

$$d\mathbf{A} = \begin{bmatrix} db_{1,1} & 0 & 0 & db_{1,2} \\ 0 & 0 & 0 & 0 \\ db_{2,1} & 0 & 0 & 0 \\ 0 & 0 & 0 & db_{2,2} \end{bmatrix}$$

---

### Answer:

The gradient to the input  $d\mathbf{A}$  is:

$$d\mathbf{A} = \begin{bmatrix} db_{1,1} & 0 & 0 & db_{1,2} \\ 0 & 0 & 0 & 0 \\ db_{2,1} & 0 & 0 & 0 \\ 0 & 0 & 0 & db_{2,2} \end{bmatrix}$$

where the non-zero entries correspond to the positions of the maxima in each  $2 \times 2$  block during the forward pass.

That is not the correct problem. 7b is the question about mean and variance. Please retry 7b

I try here to prompt it to get it back on track.



Problem Statement: 7. Weights and Gradients in a CNN (b)

Question:

Now, let's consider the backpropagation through an average-pooling layer. Suppose we have an average-pooling layer with a kernel size of  $2 \times 2$  and a stride of 2. The input to the average-pooling layer is  $\mathbf{A} \in \mathbb{R}^{4 \times 4}$ ,

**It again started talking about some completely different problem.**

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

and the output is  $\mathbf{B} \in \mathbb{R}^{2 \times 2}$ ,

$$\mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

where  $b_{i,j} = \frac{1}{4}(a_{2i-1,2j-1} + a_{2i-1,2j} + a_{2i,2j-1} + a_{2i,2j})$ .

Suppose the upstream gradient is  $d\mathbf{B} \in \mathbb{R}^{2 \times 2}$ ,

$$d\mathbf{B} = \begin{bmatrix} db_{1,1} & db_{1,2} \\ db_{2,1} & db_{2,2} \end{bmatrix}$$

where  $db_{i,j}$  denotes  $\frac{\partial \mathcal{L}}{\partial b_{i,j}}$ .

Derive the gradient to the input  $d\mathbf{A} \in \mathbb{R}^{4 \times 4}$ ,

$$d\mathbf{A} = \begin{bmatrix} da_{1,1} & da_{1,2} & da_{1,3} & da_{1,4} \\ da_{2,1} & da_{2,2} & da_{2,3} & da_{2,4} \\ da_{3,1} & da_{3,2} & da_{3,3} & da_{3,4} \\ da_{4,1} & da_{4,2} & da_{4,3} & da_{4,4} \end{bmatrix}$$

where  $da_{i,j}$  denotes  $\frac{\partial \mathcal{L}}{\partial a_{i,j}}$ .

### Step-by-Step Reasoning:

#### Average-Pooling Forward Pass

For each element in the output  $\mathbf{B}$ :

$$b_{i,j} = \frac{1}{4}(a_{2i-1,2j-1} + a_{2i-1,2j} + a_{2i,2j-1} + a_{2i,2j})$$

### Average-Pooling Backward Pass

The gradient  $d\mathbf{A}$  is computed by distributing the gradient  $d\mathbf{B}$  equally to all elements in the  $2 \times 2$  blocks of  $\mathbf{A}$ .

- For each  $b_{i,j}$ , the gradient is distributed equally to the four elements in the corresponding  $2 \times 2$  block of  $\mathbf{A}$ :

$$da_{2i-1,2j-1} = \frac{1}{4}db_{i,j}$$

$$da_{2i-1,2j} = \frac{1}{4}db_{i,j}$$

$$da_{2i,2j-1} = \frac{1}{4}db_{i,j}$$

$$da_{2i,2j} = \frac{1}{4}db_{i,j}$$

### Example Calculation

For  $b_{1,1}$ :

$$da_{1,1} = \frac{1}{4}db_{1,1}$$

$$da_{1,2} = \frac{1}{4}db_{1,1}$$

$$da_{2,1} = \frac{1}{4}db_{1,1}$$

$$da_{2,2} = \frac{1}{4}db_{1,1}$$

Similarly, for  $b_{1,2}$ :

$$da_{1,3} = \frac{1}{4}db_{1,2}$$

$$da_{1,4} = \frac{1}{4}db_{1,2}$$

$$da_{2,3} = \frac{1}{4}db_{1,2}$$

$$da_{2,4} = \frac{1}{4}db_{1,2}$$

For  $b_{2,1}$ :

$$da_{3,1} = \frac{1}{4}db_{2,1}$$

$$da_{3,2} = \frac{1}{4}db_{2,1}$$

$$da_{4,1} = \frac{1}{4}db_{2,1}$$

$$da_{4,2} = \frac{1}{4}db_{2,1}$$

For  $b_{2,2}$ :

$$da_{3,3} = \frac{1}{4}db_{2,2}$$

$$da_{3,4} = \frac{1}{4}db_{2,2}$$

$$da_{4,3} = \frac{1}{4}db_{2,2}$$

$$da_{4,4} = \frac{1}{4}db_{2,2}$$

**Answer:**

The gradient to the input  $d\mathbf{A}$  is:

$$d\mathbf{A} = \begin{bmatrix} \frac{1}{4}db_{1,1} & \frac{1}{4}db_{1,1} & \frac{1}{4}db_{1,2} & \frac{1}{4}db_{1,2} \\ \frac{1}{4}db_{1,1} & \frac{1}{4}db_{1,1} & \frac{1}{4}db_{1,2} & \frac{1}{4}db_{1,2} \\ \frac{1}{4}db_{2,1} & \frac{1}{4}db_{2,1} & \frac{1}{4}db_{2,2} & \frac{1}{4}db_{2,2} \\ \frac{1}{4}db_{2,1} & \frac{1}{4}db_{2,1} & \frac{1}{4}db_{2,2} & \frac{1}{4}db_{2,2} \end{bmatrix}$$

From here onward, I ended up separately screenshotting the question to get it back on track. It seems to have lost track of the PDF information at this point.



This is question 7b in the PDF. Please solve this.



### Problem Statement: 7. Weights and Gradients in a CNN (b)

**Question:**

The objective of this part is to investigate the effect of spatial weight sharing in convolution layers on the behavior of gradient norms with respect to changes in image size.

For simplicity of analysis, we assume  $x_{i,j}, dy_{i,j}$  are independent random variables, where for all  $i, j$ :

$$\mathbb{E}[x_{i,j}] = 0, \quad \text{Var}(x_{i,j}) = \sigma_x^2,$$

$$\mathbb{E}[dy_{i,j}] = 0, \quad \text{Var}(dy_{i,j}) = \sigma_g^2.$$

Derive the mean and variance of  $dw_{h,l} = \frac{\partial \mathcal{L}}{\partial W_{h,l}}$  for each  $h, l$  as a function of  $n, k, \sigma_x, \sigma_g$ . What is the asymptotic growth rate of the **standard deviation** of the gradient on  $dw_{h,l}$  with respect to the length and width of the image  $n$ ?

### Step-by-Step Reasoning:

#### Gradient Expression

From part (a), the gradient of the loss  $\mathcal{L}$  with respect to a single weight  $w_{h,l}$  is:

$$dw_{h,l} = \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1, j+l-1}$$

#### Mean of $dw_{h,l}$

Since  $\mathbb{E}[dy_{i,j}] = 0$  and  $\mathbb{E}[x_{i,j}] = 0$ , the mean of  $dw_{h,l}$  is:

$$\mathbb{E}[dw_{h,l}] = \sum_{i=1}^m \sum_{j=1}^m \mathbb{E}[dy_{i,j}] \cdot \mathbb{E}[x_{i+h-1, j+l-1}] = 0$$

#### Variance of $dw_{h,l}$

The variance of  $dw_{h,l}$  is:

$$\text{Var}(dw_{h,l}) = \text{Var} \left( \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} \cdot x_{i+h-1, j+l-1} \right)$$

Since  $dy_{i,j}$  and  $x_{i,j}$  are independent, and assuming independence across  $i$  and  $j$ :

$$\text{Var}(dw_{h,l}) = \sum_{i=1}^m \sum_{j=1}^m \text{Var}(dy_{i,j}) \cdot \text{Var}(x_{i+h-1, j+l-1}) = m^2 \cdot \sigma_g^2 \cdot \sigma_x^2$$

**It's doing well so far, finding that the mean is 0 and finding the variance of each element**

#### Number of Outputs $m$

For a convolutional layer with input size  $n$ , kernel size  $k$ , stride 1, and no padding:

$$m = n - k + 1$$

**It now correctly moves towards finding the final answer using the intermediate steps**

#### Asymptotic Growth Rate of Standard Deviation

The standard deviation of  $dw_{h,l}$  is:

$$\sqrt{\text{Var}(dw_{h,l})} = \sqrt{m^2 \cdot \sigma_g^2 \cdot \sigma_x^2} = m \cdot \sigma_g \cdot \sigma_x$$

Substituting  $m = n - k + 1$ :

$$\sqrt{\text{Var}(dw_{h,l})} \approx n \cdot \sigma_g \cdot \sigma_x \quad (\text{for large } n)$$

Thus, the standard deviation grows **linearly** with  $n$ .

---

**Answer:**

- **Mean of  $dw_{h,l}$ :**

$$\mathbb{E}[dw_{h,l}] = \boxed{0}$$

- **Variance of  $dw_{h,l}$ :**

$$\text{Var}(dw_{h,l}) = \boxed{(n - k + 1)^2 \cdot \sigma_g^2 \cdot \sigma_x^2}$$

- **Asymptotic Growth Rate of Standard Deviation:**

$$\sqrt{\text{Var}(dw_{h,l})} \approx \boxed{n \cdot \sigma_g \cdot \sigma_x}$$

The standard deviation of the gradient on  $dw_{h,l}$  grows **linearly** with the length and width of the image  $n$ .

**It solved the question correctly! It used variance calculations to correctly identify the asymptotic growth of the standard deviation, and correctly found the mean to be 0**

Again, I provided a screenshot here. It had seemed to be doing a problem similar to this one earlier in its attempts of 7b when it didn't remember the question.

only 2x2 max-pooling layers (no convolution layers, no activation functions). For a network with only 2x2 average-pooling layers, what will be  $d\mathbf{X}$ ?  
Suppose we feed value  $\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$ . Further assume 6 neurons, i.e.  
 $y_{1,1} = \max(x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$   
 after dimension and arbitrary max positions.

Now solve 7c, which is the following problem.



### Problem Statement: 7. Weights and Gradients in a CNN (c)

#### Question:

For a network with only  $2 \times 2$  max-pooling layers (no convolution layers, no activations), what will be  $d\mathbf{X} = \left[ \frac{\partial \mathcal{L}}{\partial x_{i,j}} \right]$ ? For a network with only  $2 \times 2$  average-pooling layers (no convolution layers, no activations), what will be  $d\mathbf{X}$ ?

#### Step-by-Step Reasoning:

##### Max-Pooling Layer

###### Forward Pass:

For a  $2 \times 2$  max-pooling layer, the output  $y_{i,j}$  is the maximum value in the  $2 \times 2$  block of the input  $\mathbf{X}$ :

$$y_{i,j} = \max(x_{2i-1,2j-1}, x_{2i-1,2j}, x_{2i,2j-1}, x_{2i,2j})$$

###### Backward Pass:

The gradient  $d\mathbf{X}$  is computed by distributing the gradient  $d\mathbf{Y}$  to the input element that was the maximum in its respective  $2 \times 2$  block during the forward pass.

- For each  $y_{i,j}$ , identify the index  $(m, n)$  in the  $2 \times 2$  block of  $\mathbf{X}$  that was the maximum:

$$(m, n) = \arg \max(x_{2i-1,2j-1}, x_{2i-1,2j}, x_{2i,2j-1}, x_{2i,2j})$$

- The gradient for the input  $\mathbf{X}$  is:

$$dx_{m,n} = dy_{i,j}$$

and all other elements in the  $2 \times 2$  block are zero.

It gives good reasoning here, and correctly identifies that one of the elements is nonzero while the rest are zero

**Example:**

If  $y_{1,1} = x_{1,1}$  (i.e.,  $x_{1,1}$  was the maximum in its block), then:

$$dx_{1,1} = dy_{1,1}, \quad dx_{1,2} = 0, \quad dx_{2,1} = 0, \quad dx_{2,2} = 0$$

**Average-Pooling Layer****Forward Pass:**

For a  $2 \times 2$  average-pooling layer, the output  $y_{i,j}$  is the average of the values in the  $2 \times 2$  block of the input  $\mathbf{X}$ :

$$y_{i,j} = \frac{1}{4}(x_{2i-1,2j-1} + x_{2i-1,2j} + x_{2i,2j-1} + x_{2i,2j})$$

**Backward Pass:**

The gradient  $d\mathbf{X}$  is computed by distributing the gradient  $d\mathbf{Y}$  equally to all elements in the  $2 \times 2$  blocks of  $\mathbf{X}$ .

- For each  $y_{i,j}$ , the gradient is distributed equally to the four elements in the corresponding  $2 \times 2$  block of  $\mathbf{X}$ :

$$dx_{2i-1,2j-1} = \frac{1}{4}dy_{i,j}, \quad dx_{2i-1,2j} = \frac{1}{4}dy_{i,j}, \quad dx_{2i,2j-1} = \frac{1}{4}dy_{i,j}, \quad dx_{2i,2j} = \frac{1}{4}dy_{i,j}$$

**Example:**

For  $y_{1,1}$ :

$$dx_{1,1} = \frac{1}{4}dy_{1,1}, \quad dx_{1,2} = \frac{1}{4}dy_{1,1}, \quad dx_{2,1} = \frac{1}{4}dy_{1,1}, \quad dx_{2,2} = \frac{1}{4}dy_{1,1}$$

**Answer:**

- **For Max-Pooling:**

$$d\mathbf{X} = \begin{bmatrix} dy_{1,1} & 0 & dy_{1,2} & 0 \\ 0 & 0 & 0 & 0 \\ dy_{2,1} & 0 & dy_{2,2} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**This is correct for max pooling (It specified that the nonzero value will be wherever the max value was, and just assumes it's in the top left for sake of demonstration here, as specified right after)**

(Assuming the top-left element is the maximum in each block.)

- **For Average-Pooling:**

$$d\mathbf{X} = \begin{bmatrix} \frac{1}{4}dy_{1,1} & \frac{1}{4}dy_{1,1} & \frac{1}{4}dy_{1,2} & \frac{1}{4}dy_{1,2} \\ \frac{1}{4}dy_{1,1} & \frac{1}{4}dy_{1,1} & \frac{1}{4}dy_{1,2} & \frac{1}{4}dy_{1,2} \\ \frac{1}{4}dy_{2,1} & \frac{1}{4}dy_{2,1} & \frac{1}{4}dy_{2,2} & \frac{1}{4}dy_{2,2} \\ \frac{1}{4}dy_{2,1} & \frac{1}{4}dy_{2,1} & \frac{1}{4}dy_{2,2} & \frac{1}{4}dy_{2,2} \end{bmatrix}$$

It did this part well and got the correct answers after good reasoning.

Advantages of

solve 7d, which is this



### Problem Statement: 7. Weights and Gradients in a CNN (d)

#### Question:

Following the previous part, discuss the advantages of max pooling and average pooling in your own words.

#### Discussion: Advantages of Max Pooling and Average Pooling

##### Max Pooling

###### 1. Translation Invariance:

Max pooling provides a form of translation invariance. By selecting the maximum value in a local neighborhood, the network becomes less sensitive to the exact position of features in the input image. This helps in recognizing features regardless of their small shifts or distortions.

###### 2. Dimensionality Reduction:

Max pooling reduces the spatial dimensions of the feature maps, which decreases the number of parameters and computations in the network. This helps in reducing overfitting and improving computational efficiency.

Now this is just a basic conceptual question, which it's probably seen in its training data before. So no surprise that it gets it right.

### **3. Feature Retention:**

By selecting the maximum value, max pooling retains the most prominent features in the local neighborhood. This can help in preserving the most important information and discarding less relevant details.

### **4. Sparsity of Activations:**

Max pooling tends to produce sparse activations, which can lead to more efficient and interpretable feature representations.

## **Average Pooling**

### **1. Smoothing Effect:**

Average pooling smooths the input feature maps by taking the average value in each local neighborhood. This can help in reducing the noise and providing a more generalized representation of the features.

### **2. Dimensionality Reduction:**

Similar to max pooling, average pooling reduces the spatial dimensions of the feature maps, which helps in reducing the number of parameters and computations in the network.

### **3. Preservation of Background Information:**

By averaging the values, average pooling retains more background information compared to max pooling. This can be useful in scenarios where the overall context and average intensity of features are important.

### **4. Robustness to Noise:**

Average pooling can be more robust to noise in the input data, as it considers the average value rather than just the maximum, which might be an outlier.

## **Summary**

- **Max Pooling** is advantageous for its ability to provide translation invariance, retain prominent features, and produce sparse activations, making it suitable for tasks where the precise location of features is less important than their presence.
- **Average Pooling** is beneficial for its smoothing effect, robustness to noise, and preservation of background information, making it suitable for tasks where the overall context and average intensity of features are important.

Yesterday 11:38pm



**Overall, it did pretty well on this homework, with a few fumbles.**