

Question 1

user [File]: file-3mgGXU5BQCf5MroKsUPZsb-Screenshot 2025-09-26 at 11.06.49 PM.png

[Uploading the whole question as an image for GPT to have more context.](#)

ChatGPT Here $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and

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

(a) Cost of one iteration Matrix-matrix multiply ($n \times m$ by $m \times p$) costs $cnmp$.

- Compute $A = WW^{\top}$: $(d_{\text{out}} \times d_{\text{in}}) \cdot (d_{\text{in}} \times d_{\text{out}}) \Rightarrow c d_{\text{out}}^2 d_{\text{in}}$. - Form $B = 3I - A$: negligible vs multiplies. - Compute BW : $(d_{\text{out}} \times d_{\text{out}}) \cdot (d_{\text{out}} \times d_{\text{in}}) \Rightarrow c d_{\text{out}}^2 d_{\text{in}}$.

Total per iteration $\approx 2c d_{\text{out}}^2 d_{\text{in}} = \Theta(d_{\text{out}}^2 d_{\text{in}})$.

—
(b) When $d_{\text{out}} \gg d_{\text{in}}$: a faster way Use associativity:

$$(WW^{\top})W = W(W^{\top}W).$$

So

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

This replaces the large WW^{\top} (size $d_{\text{out}} \times d_{\text{out}}$) by the Gram matrix $G = W^{\top}W$ (size $d_{\text{in}} \times d_{\text{in}}$).

Costs: - Compute $G = W^{\top}W$: $(d_{\text{in}} \times d_{\text{out}}) \cdot (d_{\text{out}} \times d_{\text{in}}) \Rightarrow c d_{\text{out}} d_{\text{in}}^2$. - Compute WG : $(d_{\text{out}} \times d_{\text{in}}) \cdot (d_{\text{in}} \times d_{\text{in}}) \Rightarrow c d_{\text{out}} d_{\text{in}}^2$. - Subtraction from $3W$: negligible.

Total per iteration $\approx 2c d_{\text{out}} d_{\text{in}}^2 = \Theta(d_{\text{out}} d_{\text{in}}^2)$.

Speedup: factor $\sim \frac{d_{\text{out}}}{d_{\text{in}}}$ when $d_{\text{out}} \gg d_{\text{in}}$.

user What is the runtime for $W^{\top}W$?

ChatGPT $W^{\top}W$ multiplies a $(d_{\text{in}} \times d_{\text{out}})$ matrix by a $(d_{\text{out}} \times d_{\text{in}})$ matrix, so using the $c nmp$ rule:

$$\text{time}(W^{\top}W) = c d_{\text{in}} d_{\text{out}} d_{\text{in}} = c d_{\text{out}} d_{\text{in}}^2 = \Theta(d_{\text{out}} d_{\text{in}}^2).$$

(The result is a $d_{\text{in}} \times d_{\text{in}}$ Gram matrix.)

[This is interesting since this is not how I would approach the problem.](#)

user And why would this be faster given that $d_{\text{out}} \gg d_{\text{in}}$?

ChatGPT Great question!

Let's compare the two runtimes:

- **Original approach (using WW^{\top}):** Computing WW^{\top} costs $\Theta(d_{\text{out}}^2 d_{\text{in}})$. This scales *quadratically* in d_{out} .

- **Rewritten approach (using $W^{\top}W$):** Computing $W^{\top}W$ costs $\Theta(d_{\text{out}} d_{\text{in}}^2)$. This scales *linearly* in d_{out} and quadratically in d_{in} .

—
Why it's faster when $d_{\text{out}} \gg d_{\text{in}}$: If d_{out} is much larger than d_{in} , then: - $d_{\text{out}}^2 d_{\text{in}} \gg d_{\text{out}} d_{\text{in}}^2$. - In fact, the speedup factor is roughly

$$\frac{d_{\text{out}}^2 d_{\text{in}}}{d_{\text{out}} d_{\text{in}}^2} = \frac{d_{\text{out}}}{d_{\text{in}}}.$$

So by switching from WW^{\top} (a huge $d_{\text{out}} \times d_{\text{out}}$ matrix) to the smaller Gram matrix $W^{\top}W$ (only $d_{\text{in}} \times d_{\text{in}}$), you reduce the dominant cost by a factor of $\frac{d_{\text{out}}}{d_{\text{in}}}$.

That's why the $W^{\top}W$ -based rewrite is **much more efficient** in the regime $d_{\text{out}} \gg d_{\text{in}}$.

Do you want me to also **show the full runtime expression for $p(W)$ ** in both cases, side by side, so you can directly see where the speedup comes from?

GPT-5 seems to be able to get the answer correct in one shot for part (a). For part (b), it looks correct to me too, but not how I would approach the problem. This is why I prompted it subsequently to tell me the runtime when using $W^T W$, and why that would be faster given that $d_{out} \gg d_{in}$.

Question 2

FP8 weight initialization

user [File]: file-WYiF3fCMQgTg12BXSCGnsf-Screenshot 2025-09-28 at 11.59.36.png

ChatGPT Short answer: because with fp8 you don't want the *stored* numbers to be tiny.

- Xavier/He give $O(1/fan)$. For big layers (e.g., $\text{fan_in} = 4096 \cdot (2/8192) \cdot 0.0156$; *forevenlargerlayersit's smaller.* – FP8 (e.g., E4M3) has very few representable values near zero and a minimum normal magnitude around 1.610. So a Xavier scale Gaussian will quantize so coarsely that many weights collapse to 0 or to one or two bins. The huge variance loss and effectively

If instead you initialize weights from $N(0,1)$ (unit scale), you spread the values across the fp8 codebook, preserving sign diversity and relative differences after quantization. Then, because activations are computed in full precision, you can apply the intended scaling in the forward graph (or as a learned/layerwise scale), e.g.

$$\text{effective_weight} = (\text{fp8_weight})(\text{scale1}/\text{fan_in})$$

This gives you the *functional* effect of Xavier while keeping the *stored* fp8 values well-resolved and away from underflow/zero. Hence $N(0,1)$ makes sense for fp8 parameters; it avoids quantization collapse and keeps training numerically stable.

user [File]: file-RZHy7KyiwA1jj95qFrc7ak-Screenshot 2025-09-28 at 12.11.25.png

ChatGPT Set the scalar so that the **effective** weight variance matches Xavier's.

If $W \sim \mathcal{N}(0, 1)$ and you use $y = cWx$, then the effective weights are $W' = cW$ with $\text{Var}(W') = c^2$.

Xavier (Glorot) initialization wants

$$\text{Var}(W') = \frac{2}{\text{fan_in} + \text{fan_out}}.$$

So choose

$$c = \sqrt{\frac{2}{\text{fan_in} + \text{fan_out}}}.$$

(If you only wanted forward variance preservation, you'd use $c \approx 1/\sqrt{\text{fan_in}}$; but for "standard Xavier," the formula above is the match.)

user I thought Xavier initialization makes the variance $1/d$, where d is fan_in . So should be $1/d$ or $1/\sqrt{d}$?

When using LLMs to solve problems, we have to keep an eye out for inconsistencies with lectures.

In this case, the convention for Xavier initialization in lecture is different to that used by ChatGPT.

ChatGPT Good question — there are actually **two slightly different conventions** floating around, which is why it feels confusing.

1. Xavier/Glorot initialization Glorot Bengio (2010) define the variance of the weights as

$$\text{Var}(W) = \frac{2}{\text{fan_in} + \text{fan_out}}.$$

- If you look at the **forward-only case** (just trying to keep activations from exploding), then the variance that preserves signal is

$$\text{Var}(W) = \frac{1}{\text{fan_in}}.$$

- If you look at the **backward-only case** (trying to keep gradients stable), then

$$\text{Var}(W) = \frac{1}{\text{fan_out}}.$$

Xavier's trick is to take the *average* of the two, i.e.

$$\frac{1}{2} \left(\frac{1}{\text{fan_in}} + \frac{1}{\text{fan_out}} \right) = \frac{2}{\text{fan_in} + \text{fan_out}}.$$

-
2. What does that mean for your scalar c ? If your weights come from $\mathcal{N}(0, 1)$, they have variance
1. Multiplying by c makes the variance c^2 .

- To mimic **forward-variance preservation only**: set $c = \frac{1}{\sqrt{\text{fan_in}}}$.

- To mimic **full Xavier initialization (Glorot's average)**: set $c = \sqrt{\frac{2}{\text{fan_in} + \text{fan_out}}}$.

So the safe answer to your homework's wording ("recover the benefits of *standard Xavier*") is:

$$c = \sqrt{\frac{2}{\text{fan_in} + \text{fan_out}}}$$

But if your instructor or notes were emphasizing *just* the forward-propagation case, then the answer would be

$$c = \frac{1}{\sqrt{\text{fan_in}}}.$$

Do you want me to show you a quick derivation of why forward propagation stability leads specifically to the $1/\sqrt{d}$ (fan_in) scaling?

user [File]: file-TVtfboQFw4scBGCsf3HvBL-Screenshot 2025-09-28 at 12.17.44.png

ChatGPT Let $x \in \mathbb{R}^{d_{\text{in}}}$, $y \in \mathbb{R}^{d_{\text{out}}}$. $\text{RMS}(v) = \|v\|_2 / \sqrt{\dim(v)}$.

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

using the spectral norm $\|\Delta W\|_2$.

Requiring $\|\Delta y\|_{\text{RMS}} \leq 1$ with $\|x\|_{\text{RMS}} = 1$ (so $\|x\|_2 = \sqrt{d_{\text{in}}}$) gives

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

If you plug typical choices for c :

- Forward-preserving ($c = 1/\sqrt{d_{\text{in}}}$): $\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$.

- Standard Xavier ($c = \sqrt{2/(d_{\text{in}} + d_{\text{out}})}$): $\|\Delta W\|_2 \leq \sqrt{\frac{d_{\text{out}}(d_{\text{in}} + d_{\text{out}})}{2d_{\text{in}}}}$.

But the general bound above is the clean answer.

user [File]: file-FQkb7m6v5z96wy1KMxC9ui-Screenshot 2025-09-28 at 12.36.35.png

ChatGPT Let the update be $\Delta W = -\alpha \text{sign}(\nabla_W L)$. From (c) we need

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

For SignSGD with batch size 1, $\text{sign}(\nabla_W L)$ is rank-1 and can be written as

$$\text{sign}(\nabla_W L) = u v^\top, \quad u \in \{\pm 1\}^{d_{\text{out}}}, \quad v \in \{\pm 1\}^{d_{\text{in}}}.$$

The spectral norm of a rank-1 outer product is the product of the vector norms:

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

Thus

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

(which works for rectangular $d_{\text{out}} \times d_{\text{in}}$ matrices).

Examples: - If $c = \frac{1}{\sqrt{d_{\text{in}}}}$ (forward-preserving), then $\alpha_{\max} = 1/\sqrt{d_{\text{in}}}$. - If $c = \sqrt{2/(d_{\text{in}} + d_{\text{out}})}$ (standard Xavier), then $\alpha_{\max} = \sqrt{(d_{\text{in}} + d_{\text{out}})/2}/d_{\text{in}}$.

user [File]: file-4cz5QK8FwW6m7MKmTsqB2L-Screenshot 2025-09-28 at 12.44.02.png

ChatGPT Because U and V are column-orthonormal (compact SVD), the matrix UV^\top is a partial isometry with

$$\|UV^\top\|_2 = 1$$

(for any rectangular shapes).

Your update is $\Delta W = \alpha UV^\top$, hence

$$\|\Delta W\|_2 = \alpha.$$

From part (c) the constraint is

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

Therefore the required learning rate is

$$\boxed{\alpha \leq \frac{1}{|c|} \sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}}}$$

(works for rectangular $d_{\text{out}} \times d_{\text{in}}$).

Examples: - If $c = 1/\sqrt{d_{\text{in}}}$: $\alpha_{\max} = \sqrt{d_{\text{out}}}$. - If $c = \sqrt{2/(d_{\text{in}} + d_{\text{out}})}$: $\alpha_{\max} = \sqrt{\frac{d_{\text{out}}(d_{\text{in}} + d_{\text{out}})}{2d_{\text{in}}}}$.

user [File]: file-Xdu1JBt8UqBdKibVQoMGjc-Screenshot 2025-09-28 at 12.47.57.png

ChatGPT Yes.

For a linear stack $x_{n+1} = c_n W_n x_n$,

$$\nabla_{W_n} L = c_n (\nabla_{x_{n+1}} L) x_n^\top.$$

Any scalar rescaling of the *intermediate* backprop signal, $\nabla_{x_{n+1}} L \leftarrow s \nabla_{x_{n+1}} L$, just multiplies $\nabla_{W_n} L$ by the same scalar s .

With scale-invariant update rules—e.g. - **SignSGD:** $\Delta W_n \propto \text{sign}(\nabla_{W_n} L)$, - **Adam (RMS-normalized):** $\Delta W_n \propto \nabla_{W_n} L / \sqrt{\text{RMS}(\nabla_{W_n} L^2)}$, - **Muon/orthogonalized:** $\Delta W_n \propto UV^\top$ from the SVD of $\nabla_{W_n} L$,

the update is homogeneous of degree 0 in the gradient ($f(sG) = f(G)$ for $s > 0$), so the magnitude s cancels. Thus the scale of the intermediate gradients $\nabla_{x_n} L$ can be ignored.

A convenient setting is to choose the forward scalars so that activations have unit RMS at each layer (your c_n from part (b)), and use one of the above scale-invariant optimizers. Then only the **directions** of x_n and $\nabla_{x_{n+1}} L$ matter; their absolute scales do not. The per-layer step size still must satisfy the spectral-norm bounds you derived earlier.

user [File]: file-T8km7vDzxnebEmD7vgcgcb-Screenshot 2025-09-28 at 12.54.04.png

ChatGPT Yes—without any extra scaling the back-propagated gradients **explode**.

For layer $x_{n+1} = c_n W_n x_n$,

$$\nabla_{x_n} L = c_n W_n^\top \nabla_{x_{n+1}} L \Rightarrow \|\nabla_{x_n} L\|_2 \leq c_n \|W_n\|_2 \|\nabla_{x_{n+1}} L\|_2.$$

With the assumptions: - W_n is rank-1 $\|W_n\|_2 = \|W_n\|_F$. - Each parameter has unit scale $\|W_n\|_F = \sqrt{d_{\text{out}} d_{\text{in}}}$.

So each layer multiplies the gradient magnitude by

$$\gamma_n = c_n \sqrt{d_{\text{out}} d_{\text{in}}}.$$

This is > 1 for typical c_n choices (e.g., $c_n = 1/\sqrt{d_{\text{in}}}$ gives $\gamma_n = \sqrt{d_{\text{out}}}$), hence explosion.

To keep intermediate gradients stable, multiply the backprop signal at layer $n + 1$ by

$$s_n = \frac{1}{c_n \|W_n\|_2} = \frac{1}{c_n \sqrt{d_{\text{out}} d_{\text{in}}}} \quad \text{each layer.}$$

Special cases: - If $c_n = \frac{1}{\sqrt{d_{\text{in}}}}$ (forward-preserving), then $s_n = \frac{1}{\sqrt{d_{\text{out}}}}$. - If $c_n = \sqrt{\frac{2}{d_{\text{in}} + d_{\text{out}}}}$ (standard Xavier), then $s_n = \sqrt{\frac{d_{\text{in}} + d_{\text{out}}}{2d_{\text{in}} d_{\text{out}}}}$.

Applying these per layer makes the intermediate backpropagated gradients stay at a constant scale.

Generally the answers are correct, though one must remember that the Xavier initialization in lecture is $N(0, \frac{1}{d})$ since it suggests another version of it.

Question 3

Discrete-time convolution

user [File]: file-DNBVKG8EDb3Kydrog7fXcZ-Screenshot 2025-09-26 at 11.38.53 PM.png [File]: file-EjDNVwVeH6fqToEpenurfm-Screenshot 2025-09-28 at 13.38.25.png

ChatGPT Compute and plot $y[n] = x[n] * h[n]$ where $x[n] = 1$ for $n=0..6$ (rect of length $L=7$), 0 otherwise $h[n] = (1/2)^n u[n] = (1/2)^n$ for $n >= 0$, 0 otherwise Plot from $n = -6$ to 12

```
import numpy as np import matplotlib.pyplot as plt
def y_of_n(n) : if n < 0 : return 0.0 else if 0 <= n <= 6 : sum_k=0..n (1/2)^k = 2 - (1/2)^n return 2.0 - (0.5)**n else : n >= 7 sum_k=0..6 (1/2)^n-k = (1/2)^n-6 * sum_k=0..6 (1/2)^k = (1/2)^n-6 * 2 * (1 - (1/2)^7) = (127/64) * (1/2)^n-6 return (127.0/64.0) * (0.5) ** (n - 6)
ns = np.arange(-6, 13) ys = np.array([y_of_n(int(n)) for n in ns])
plt.figure() markerline, stemlines, baseline = plt.stem(ns, ys, use_line_collection = True) plt.title("Convolution y[n] = x[n] * h[n] (L = 7, h[n] = (1/2)^n u[n])") plt.xlabel("n") plt.ylabel("y[n]") plt.grid(True) plt.show()
```

ChatGPT Here's the discrete-time convolution $y[n] = x[n] * h[n]$ for - $x[n] = 1$ for $n = 0, \dots, 6$ (rect of length $L = 7$), else 0 - $h[n] = (\frac{1}{2})^n u[n]$.

Closed-form (derived from overlap limits in the sum $y[n] = \sum_i x[i]h[n-i]$):

$$y[n] = \begin{cases} 0, & n < 0, \\ \sum_{i=0}^n \left(\frac{1}{2}\right)^{n-i} = 2 - \left(\frac{1}{2}\right)^n, & 0 \leq n \leq 6, \\ \sum_{i=0}^6 \left(\frac{1}{2}\right)^{n-i} = \frac{127}{64} \left(\frac{1}{2}\right)^{n-6}, & n \geq 7. \end{cases}$$

I computed and plotted $y[n]$ from $n = -6$ to 12 (stem plot) so you can see the ramp-up during overlap with the rect and the exponential tail afterward.

user Generalize this to any L

ChatGPT Let - $x[n] = 1$ for $n = 0, \dots, L-1$ (0 otherwise), - $h[n] = (\frac{1}{2})^n u[n]$.

Then $y[n] = (x * h)[n] = \sum_{i=0}^{L-1} h[n-i]$ over the overlap.

$$y[n] = \begin{cases} 0, & n < 0, \\ \sum_{k=0}^n \left(\frac{1}{2}\right)^k = \frac{1 - (1/2)^{n+1}}{1 - 1/2} = 2 - 2^{-n}, & 0 \leq n \leq L-1, \\ \sum_{k=0}^{L-1} \left(\frac{1}{2}\right)^{n-k} = 2^{-n} \sum_{k=0}^{L-1} 2^k = 2^{-n} (2^L - 1), & n \geq L. \end{cases}$$

So the rising part is $2 - 2^{-n}$ until $n = L-1$, and for $n \geq L$ the tail is an exponential 2^{-n} scaled by $2^L - 1$.

(Extra generalization: if $h[n] = a^n u[n]$ with $|a| < 1$, replace 2 by $\frac{1}{1-a}$ and 2^{-n} by a^{-n} : $y[n] = \frac{1-a^{n+1}}{1-a}$ for $0 \leq n \leq L-1$, and $y[n] = a^n \frac{1-a^L}{1-a}$ for $n \geq L$.)

user Can you check your answer again to be sure!

ChatGPT Good catch—one detail in my “extra generalization” needed fixing. I re-derived everything carefully. Here’s the clean, correct result.

Let - $x[n] = 1$ for $n = 0, \dots, L-1$ (0 otherwise), - $h[n] = a^n u[n]$ with $|a| < 1$.

Then $y[n] = (x * h)[n] = \sum_{i=0}^{L-1} a^{n-i} u[n-i]$, which gives

$$y[n] = \begin{cases} 0, & n < 0, \\ \sum_{k=0}^n a^k = \frac{1-a^{n+1}}{1-a}, & 0 \leq n \leq L-1, \\ \sum_{i=0}^{L-1} a^{n-i} = a^{n-(L-1)} \sum_{k=0}^{L-1} a^k = a^{n-(L-1)} \frac{1-a^L}{1-a}, & n \geq L. \end{cases}$$

Specializing to $a = \frac{1}{2}$:

$$y[n] = \begin{cases} 0, & n < 0, \\ 2 - 2^{-n}, & 0 \leq n \leq L-1, \\ 2^{-n}(2^L - 1) \text{ (equivalently } \frac{1-2^{-L}}{1-1/2} 2^{-(n-(L-1))}), & n \geq L. \end{cases}$$

So the only correction is that for the general a -case tail you need the factor $a^{n-(L-1)}$, not a^n . Everything else checks out.

user [File]: file-Jm5ysSkXgRjSF6a6BYAFpB-Screenshot 2025-09-28 at 14.30.53.png

ChatGPT Shift the input, shift the output by the same amount. If $x_2[n] = x[n-N]$ and $y[n] = (x * h)[n]$, then

$$y_2[n] = (h * x_2)[n] = (x_2 * h)[n] = y[n-N].$$

This is the **time-shift (shift-equivariance) property** of convolution (and also shows **commutativity**: $h * x_2 = x_2 * h$).

For your case with $h[n] = (1/2)^n u[n]$ and a length- L rectangle $x[n]$:

$$y[n] = \begin{cases} 0, & n < 0, \\ 2 - 2^{-n}, & 0 \leq n \leq L-1, \\ (2^L - 1) 2^{-n}, & n \geq L, \end{cases}$$

so with $N = 5$,

$$y_2[n] = y[n-5] = \begin{cases} 0, & n < 5, \\ 2 - 2^{-(n-5)}, & 5 \leq n \leq L+4, \\ (2^L - 1) 2^{-(n-5)}, & n \geq L+5. \end{cases}$$

(General $h[n] = a^n u[n]$, $|a| < 1$: $y_2[n] = 0$ for $n < N$; $y_2[n] = \frac{1-a^{n-N+1}}{1-a}$ for $N \leq n \leq N+L-1$;
 $y_2[n] = a^{n-(N+L-1)} \frac{1-a^L}{1-a}$ for $n \geq N+L$.)

user [File]: file-9KS7HP7xgca5x5BhaTr7FX-Screenshot 2025-09-28 at 14.33.02.png

ChatGPT We’re doing **true 2-D convolution** (as in the formula shown), i.e. flip the kernel and then slide it with stride 1 and no padding.

Given

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

the flipped kernel for convolution is

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

With valid convolution (no pad, stride 1), the output size is 3×3 . At the top-left location:

$$\begin{aligned} y[0,0] &= \sum_{i,j=0}^2 x[i,j]\tilde{h}[i,j] \\ &= \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \\ 11 & 12 & 13 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \\ &= (1+4+3) - (11+24+13) = 8 - 48 = -40. \end{aligned}$$

Because the kernel computes a vertical gradient and x increases linearly, every valid 3×3 patch gives the same value. Thus

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

Did not expect it to be able to compute the convolutions accurately. This is pretty cool. In the past, LLMs tended to do rounding errors when handling multiple numbers.

(If you instead use **correlation**—i.e., slide h without flipping—you'd get the same matrix with +40 everywhere.)

user Why do we need to flip the kernel?

I was curious about this since I didn't do this the first time. Learned something :)

ChatGPT Great question!

Whether we flip the kernel or not depends on whether we are doing **true convolution** (as defined in signal processing) or **correlation** (as often used in computer vision and neural networks).

1. **Mathematical definition of convolution (signals systems)** The 1D convolution is defined as:

$$(y = x * h)[n] = \sum_{i=-\infty}^{\infty} x[i] h[n-i].$$

Notice that the filter h is evaluated at $n-i$, which means we **reverse (flip)** it in time before sliding. - In 2D, the same principle applies: $h[m-i, n-j] \rightarrow$ the kernel is flipped in both directions.

This flipping is essential to preserve theoretical properties like **commutativity** ($x * h = h * x$) and to match the idea of an **impulse response** of an LTI system.

2. **Correlation (used in CNNs)** In deep learning frameworks, what they call “convolution” is usually:

$$(y = x \star h)[n] = \sum_{i=-\infty}^{\infty} x[i] h[n+i],$$

which is **correlation**, not convolution—no flipping.

Why is this acceptable in CNNs? Because the kernel weights are learned. If flipping mattered, the network would just learn the flipped version during training. So correlation and convolution are equivalent in practice for learning systems.

3. **Summary** - **Signal processing / LTI systems** → **Flip + Slide** (true convolution).
- **CNNs / computer vision** → **No flip** (correlation, but called convolution).

So, we flip in classical convolution to stay consistent with the definition and theoretical properties of LTI systems.

Do you want me to show explicitly what would happen in your 5×5 example if we *don't* flip the kernel (correlation instead), so you can compare the two outputs side by side?

user [File]: file-1XQd3hD5Mwvpsd4KroqpB-Screenshot 2025-09-28 at 14.37.14.png

ChatGPT Using **true 2-D convolution** (flip h first), input $x \in \mathbb{R}^{5 \times 5}$ and kernel $h \in \mathbb{R}^{3 \times 3}$ from (c).

Output size formula:

$$H_{\text{out}} = \left\lceil \frac{H + 2P - K}{S} \right\rceil + 1, \quad W_{\text{out}} = \left\lceil \frac{W + 2P - K}{S} \right\rceil + 1$$

with $H = W = 5$, $K = 3$.

We flip h for convolution:

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

(i) stride $S = 1$, pad $P = 1$ Size: 5×5 . The result is

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

(ii) stride $S = 2$, pad $P = 1$ Size: 3×3 . The result is

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

(If you instead used correlation—no flip—the signs of the constant interior values would be $+40$ rather than -40 , and border values would change accordingly.)

The answer might be correct – I approached the computation differently. The conclusion on the property of convolutions in (b) agrees with my own. I was expecting it to fail on computing 2D convolutions, but it correctly found the answers for both (c) and (d).

Question 4

Weights and biases calculation

user [File]: file-3shry8DapRYenshDDs9VbA-Screenshot 2025-09-28 at 21.09.16.png

ChatGPT **Weights biases**

- Each filter (kernel) has size $K \times K$ and depth $C \rightarrow **K^2C$ weights per filter**. - There is $**1$ bias per filter**. - With F filters: - **Total weights:** $F \cdot K^2C$ - **Total biases:** F - **Total parameters:** $F(K^2C + 1)$

Output feature map shape

For input $W \times H \times C$ with padding P and stride S :

$$W_{\text{out}} = \left\lceil \frac{W - K + 2P}{S} \right\rceil + 1, \quad H_{\text{out}} = \left\lceil \frac{H - K + 2P}{S} \right\rceil + 1.$$

Number of channels equals the number of filters, F .

So the output tensor shape is $**W_{\text{out}} \times H_{\text{out}} \times F**$.

user is $C == F$?

ChatGPT Good question — **no, $C \neq F$ ** in general.

- C = number of **input channels** (e.g., 3 for RGB images). - F = number of **filters** (also called kernels) in the convolutional layer.

Each filter spans **all input channels** (C), so its depth is fixed by the input. But the number of filters F is a design choice: it determines how many output channels (feature maps) you produce.

So the relationship is:

$$\text{Input shape: } W \times H \times C \longrightarrow \text{Output shape: } W_{\text{out}} \times H_{\text{out}} \times F$$

Special case: If you deliberately set $F = C$, the number of input and output channels match, but that's by design — not a requirement.

Do you want me to also explain why filters must have depth C (and not arbitrary depth)?

user [File]: file-PkDqyS3T7kGcABB7e7eHUu-Screenshot 2025-09-28 at 21.13.47.png

ChatGPT For a **max pooling layer**:

- Input shape: $W \times H \times C$ - Hyperparameters: filter size K , stride S

Pooling is applied **independently on each channel**, so the number of channels does **not** change.

The output width and height are:

$$W_{\text{out}} = \left\lfloor \frac{W - K}{S} \right\rfloor + 1, \quad H_{\text{out}} = \left\lfloor \frac{H - K}{S} \right\rfloor + 1$$

Thus the **output shape** is:

$$W_{\text{out}} \times H_{\text{out}} \times C$$

Key difference from convolution: pooling does **not** introduce new filters, so the channel dimension C is preserved.

Do you want me to also show the formula if padding is added (some frameworks allow it for pooling)?

user why do you need the floor operator?

ChatGPT Great catch — the **floor operator** is needed because the pooling (and convolution) kernel can't usually cover the input perfectly unless $W - K$ is divisible by S .

Here's why:

- The formula for the output width before flooring is:

$$\frac{W - K}{S} + 1$$

- But if $(W - K)$ is **not** divisible by S , you'd get a non-integer number of steps. For example:
- Suppose $W = 7$, $K = 2$, $S = 3$. - Then $\frac{7-2}{3} + 1 = \frac{5}{3} + 1 = 2.\overline{6}$. - You can't produce $2.\overline{6}$ windows in reality — you can only fit **2 full windows** without spilling outside the boundary.

So we take the **floor** to indicate: - Only the integer number of valid windows fits. - Any remainder pixels at the edge are dropped (unless padding is applied).

In practice: frameworks like **PyTorch** and **TensorFlow** also let you specify “same” vs “valid” padding modes to control whether those extra edge pixels are handled by padding or discarded.

Would you like me to show a small **visual example with numbers** so you can see how the kernel slides and why flooring is necessary?

user [File]: file-XmqLCEuy3R5mLsotFFNmYx-Screenshot 2025-09-28 at 21.15.22.png

ChatGPT We're asked about the **receptive field size** after L successive convolutional layers, each with filter size K and stride $S = 1$.

Step 1: Receptive field for a single layer - With filter size K , stride 1, each output neuron sees K input units. - So receptive field = K .

Step 2: Add a second layer - Now each output unit of the second layer sees K outputs of the first layer. - Each of those already covered a window of size K in the input. - With stride = 1, these overlap, so the new receptive field grows as:

$$RF(2) = K + (K - 1) = 2K - 1$$

Step 3: Generalize to L layers - Every additional layer of stride 1 expands the receptive field by $(K - 1)$. - So:

$$RF(L) = K + (L - 1)(K - 1)$$

Final answer:

$$\boxed{\text{Receptive field size after } L \text{ layers} = K + (L - 1)(K - 1)}$$

Do you want me to also extend this formula for the case when stride $S > 1$?

user [File]: file-BgLQrqN19PzdMf5pcG1Dh2-Screenshot 2025-09-28 at 21.18.27.png
ChatGPT Let's unpack this carefully.

Receptive field increase - We apply **max pooling with kernel size $K = 2$, stride $S = 2$ **. - Each pooled output corresponds to a 2×2 block in the input. - So, the **receptive field doubles in both width and height** compared to what stride-1 convolution would give.

In other words, the receptive field size increases by a **factor of 2**.

Why this matters - **Decreasing feature resolution (downsampling):** - Reduces the **spatial resolution** (fewer activations per feature map). - Each neuron now represents a **larger portion of the input**. - **Advantages:** 1. **Reduced computation and memory** - If width and height are halved, the number of activations drops by $1/4$. - This reduces both forward and backward pass costs. 2. **Larger receptive field per unit** - Helps the network capture more **global context**. 3. **Dimensionality reduction** - Helps prevent overfitting and reduces redundancy.
- **Trade-off:** Loss of fine-grained spatial detail (can harm localization tasks like segmentation).

Answer (concise):

- The receptive field increases by a factor of 2^2 when applying 2×2 pooling with stride 2.
- Advantage: although spatial resolution decreases, computation is reduced (since the feature map shrinks by $4\times$), while each unit sees a larger input area, improving efficiency and global context capture.

Do you want me to also show a **numerical example** (say input 8×8) so you can see exactly how the resolution and receptive field change?

user So the receptive field becomes $2^2(K + (K-1)(L-1))$?

ChatGPT Almost — let's go step by step and be precise.

Before pooling From part (c), the receptive field after L convolutional layers (stride = 1) is:

$$RF_{\text{conv}}(L) = K + (L - 1)(K - 1).$$

After adding **one 2×2 pooling layer with stride 2** - Each pooling output corresponds to a 2×2 region of its input. - Since stride = 2, the effective “jump” doubles: each output unit now covers 2×2 patches of whatever the receptive field was in the previous layer.

So the receptive field **doubles**, but not by simply multiplying the whole formula by 2. Instead, the general update rule for receptive field when adding a layer is:

$$RF_{\text{new}} = RF_{\text{old}} + (K_{\text{new}} - 1) \cdot \text{jump}_{\text{old}},$$

where jump_{old} is the spacing in input pixels between adjacent features of the old layer. - For stride-1 convolutions, $\text{jump} = 1$. - After pooling with stride $S = 2$, jump doubles: $\text{jump}_{\text{new}} = 2$.

So with $K_{\text{pool}} = 2$, $S = 2$:

$$RF_{\text{pool}} = RF_{\text{conv}}(L) + (2 - 1) \cdot 1 = RF_{\text{conv}}(L) + 1,$$

and the ****jump**** becomes 2, meaning each step in the pooled feature map spans 2 input pixels. Thus, effectively each unit of the pooled map corresponds to ****roughly 2×** the previous receptive field size ****** in each dimension.

So the correct receptive field after pooling is:

$$RF_{\text{after}} \approx 2 \cdot RF_{\text{conv}}(L).$$

Your formula $2 \cdot (K + (K - 1)(L - 1))$ is indeed correct under the stride-2 pooling assumption, because the pooling doubles the scale of the receptive field.

Would you like me to ****generalize** the formula ****** for any pooling kernel size K_p and stride S_p , not just 2×2 with stride 2?

user [File]: file-VMamNxAJGHQWN2QuWFhV8m-Screenshot 2025-09-28 at 21.35.22.png

ChatGPT Here are the missing entries:

- ****CONV3-10** (from $28 \times 28 \times 1 \rightarrow 28 \times 28 \times 10$) ****** Number of parameters: ****** $3 \times 3 \times 1 \times 10 + 10$ ******
- ****POOL2** (from $28 \times 28 \times 10 \rightarrow 14 \times 14 \times 10$) ****** Number of parameters: ****0**** (already shown)
- ****CONV3-10** (from $14 \times 14 \times 10 \rightarrow 14 \times 14 \times 10$) ****** Dimension of activations: ****** $14 \times 14 \times 10$ ******
- ****POOL2** (from $14 \times 14 \times 10 \rightarrow 7 \times 7 \times 10$) ****** Number of parameters: ****0**** Dimension of activations: ****** $7 \times 7 \times 10$ ******
- ****FLATTEN**** (already shown) ****** $490 = 7 \times 7 \times 10$ ******
- ****FC-3** (from $490 \rightarrow 3$) ****** Number of parameters: ****** $490 \times 3 + 3$ ****** Dimension of activations: ****3**** (already shown)

If you want the numeric totals too: - CONV3-10 1: $90 + 10 = 100$ - CONV3-10 2: $3 \times 3 \times 10 \times 10 + 10 = 900 + 10 = 910$ - FC-3: $1470 + 3 = 1473$ Overall params: $100 + 910 + 1473 = 2,483$.

user [File]: file-BwL1WzcihxNHqWusdEJZBq-Screenshot 2025-09-28 at 21.39.26.png

ChatGPT Nice — this question is testing your understanding of ****equivariance and invariance**** of CNNs, especially with ****circular padding**** and ****global average pooling (GAP)****.

Recap of what's given - ****Architecture:****

CONV2-3 → ReLU → CONV2-3 → ReLU → GAP → FC-3

- Both convolutions: stride = 1, circular padding = 1. - GAP: averages over the spatial extent, so output is ****translation-invariant****. - Known outputs: - Horizontal edge x_1 :

$$g_1 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}$$

- Vertical edge x_2 :

$$g_2 = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$$

Now consider x_3 - x_3 is a ****diagonal edge****. - By symmetry, convolutional nets (with circular padding) are ****equivariant to translations**** and (to some extent) rotations/reflections, but after

GAP the representation is **invariant to position**. - That means: - A horizontal edge activates the **“horizontal” filter** → channel 1. - A vertical edge activates the **“vertical” filter** → channel 2. - A diagonal edge is essentially a **superposition** of horizontal + vertical.

So, GAP output for x_3 should be a **linear combination**:

$$g_3 \approx \begin{bmatrix} 0.8 \\ 0.8 \\ 0 \end{bmatrix}.$$

Answer:

$$g_3 = f(x_3) = \begin{bmatrix} 0.8 \\ 0.8 \\ 0 \end{bmatrix}.$$

Do you want me to also walk through the **next part of the question** (where they ask for other transformed inputs, like rotations of x_1 or x_2)?

user Why is x_3 a diagonal edge?

The edge in the image is actually a vertical edge; the vision capabilities of GPT-5 failed this time interestingly.

ChatGPT Good catch — let's slow down and look carefully at x_3 .

Here is the matrix they gave:

$$x_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Observations - The **ones form a vertical stripe** in column 6. - Every row has a “1” in the same column → that's a **vertical edge**, not a diagonal.

Why I mistakenly called it diagonal earlier I assumed the “shifted ones” pattern would form a diagonal (like stepping across the matrix), but actually here the “1” stays in the same column across all rows. So the image is essentially the same as x_2 , just shifted to the right because of circular padding.

Important point Because we're using **circular padding** + **global average pooling**, the network is **translation-invariant**: - Shifting the vertical edge left or right does **not** change the GAP output.

So for x_3 :

$$g_3 = g_2 = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}$$

Corrected answer: x_3 represents a **vertical edge**, so its GAP output is the same as x_2 .

Do you want me to also compute g_4 if they give you a true **diagonal edge image** (like 1s along the main diagonal)?

user [File]: file-NtV6QPwdRhRrN2Ypm3AvPy-Screenshot 2025-09-28 at 21.42.24.png

ChatGPT x_4 is a **horizontal edge** (a whole row of 1s). It's just a **vertical translation** of x_1 .

With stride-1 convolutions and **circular padding**, the conv stack is translation-equivariant, and the **GAP** layer removes location information (translation-invariant). So shifting the horizontal edge doesn't change the GAP features.

$$g_4 = f(x_4) = g_1 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}.$$

Generally its answers are correct, though for (f) it mistakenly thought x_3 represents an image with a diagonal edge even though it is a vertical edge.

Question 5

Coding Portion

Question 6

Coding Portion

Question 7

Gradient computation steps

user [File]: file-FyJzHnnQsV4Ehs7SANDy1o-Screenshot 2025-09-26 at 11.58.21 PM.png [File]: file-R61fuDRZfNDX3zyyxHrdng-Screenshot 2025-09-28 at 23.01.11.png

ChatGPT Great, let's carefully derive part (a).

We're asked to compute the gradient of the loss \mathcal{L} w.r.t. the convolution weights $w_{h,l}$, given:

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

Step 1: Chain rule setup The loss depends on $w_{h,l}$ only through the outputs $y_{i,j}$. So:

$$\frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i=1}^m \sum_{j=1}^m \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{h,l}}.$$

We already know that:

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

Step 2: Derivative of $y_{i,j}$ w.r.t. $w_{h,l}$ From the definition of convolution:

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

So:

$$\frac{\partial y_{i,j}}{\partial w_{h,l}} = x_{i+h-1, j+l-1}.$$

Step 3: Put it together Therefore:

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

This is exactly the **convolution of the input X with the upstream gradient $dY**.$

Step 4: Gradient matrix form So the gradient matrix $dw \in \mathbb{R}^{k \times k}$ is:

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

Step 5: SGD update rule If the learning rate is η , one SGD step updates the weights as:

$$w_{h,l}^{\text{new}} = w_{h,l} - \eta dw_{h,l}.$$

Or, in matrix form:

$$\mathbf{w}^{\text{new}} = \mathbf{w} - \eta dw.$$

Final result:

- Gradient:

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

- Weight update after one SGD step:

$$w_{h,l}^{\text{new}} = w_{h,l} - \eta \sum_{i=1}^m \sum_{j=1}^m dy_{i,j} x_{i+h-1, j+l-1}.$$

Do you want me to also show how this can be seen as a **cross-correlation** (like convolution without flipping) between X and dY , which is how frameworks like PyTorch actually implement it?

user [File]: file-KWjtUGmb8WrX2A3Lkh7HwV-Screenshot 2025-09-28 at 23.19.15.png
ChatGPT Let

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

(from stride 1, no padding).

Assume all $x_{p,q}$ and $dy_{i,j}$ are mutually independent with $\mathbb{E}[x_{p,q}] = 0$, $\text{Var}(x_{p,q}) = \sigma_x^2$ and $\mathbb{E}[dy_{i,j}] = 0$, $\text{Var}(dy_{i,j}) = \sigma_g^2$.

For fixed h, l , the pairs $\{(dy_{i,j}, x_{i+h-1, j+l-1})\}_{i,j=1}^m$ are i.i.d. and all the x -indices are distinct, so

- **Mean**

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

- **Variance** For one term,

$$\text{Var}(dy_{i,j} x_{i+h-1, j+l-1}) = \mathbb{E}[dy^2] \mathbb{E}[x^2] - (\mathbb{E}[dy] \mathbb{E}[x])^2 = \sigma_g^2 \sigma_x^2.$$

By independence across (i, j) ,

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

- **Standard deviation and asymptotics**

$$\text{sd}(dw_{h,l}) = (n - k + 1) \sigma_g \sigma_x = \Theta(n) \quad (\text{for fixed } k).$$

So the gradient's mean is 0, its variance scales with the **area** of the valid output $(n - k + 1)^2$, and its standard deviation grows **linearly** with image size n .

user [File]: file-Cmk5TWuX3jiQbuL9yCqLZh-Screenshot 2025-09-28 at 23.27.15.png

ChatGPT Here's the back-prop rule for ** 2×2 pooling (stride 2, no overlap)**.

1) Max-pooling (2×2) For each pooled output $y_{u,v} = \max\{x_{2u-1,2v-1}, x_{2u-1,2v}, x_{2u,2v-1}, x_{2u,2v}\}$ with upstream grad $dy_{u,v}$:

- Create a **mask** that is 1 at the arg-max location inside that 2×2 block and 0 elsewhere. - Send the entire gradient to that location; others get 0.

Example (hint case: top-left is the max):

$$dX = \begin{bmatrix} dy_{1,1} & 0 \\ 0 & 0 \end{bmatrix}.$$

In general, for block $B_{u,v}$,

$$(dX)_{i,j} = \begin{cases} dy_{u,v}, & (i,j) = \arg \max_{(p,q) \in B_{u,v}} x_{p,q}, \\ 0, & \text{otherwise.} \end{cases}$$

(If there are ties, any subgradient is valid; most libraries pick the first max. Splitting the grad evenly among maxima is also a valid choice.)

If you have ** L stacked max-pool layers**, the gradient routes back along the chain of arg-max positions at each layer (no averaging/scaling), i.e., it's a scatter via the composed masks.

2) Average-pooling (2×2) Each pooled output is

$$y_{u,v} = \frac{1}{4} \sum_{(p,q) \in B_{u,v}} x_{p,q}.$$

So the upstream gradient is divided **equally** among the four inputs in the block:

$$(dX)_{p,q} = \begin{cases} \frac{1}{4} dy_{u,v}, & (p,q) \in B_{u,v}, \\ 0, & \text{otherwise.} \end{cases}$$

For ** L stacked avg-pool layers**, each back-step divides by 4, so the grad to each element in the $2^L \times 2^L$ receptive field is

$$\frac{1}{4^L} dy \quad (\text{from the corresponding top-layer element}).$$

user [File]: file-9S5y8LAGNUPhDdax2qm5Cp-Screenshot 2025-09-28 at 23.35.39.png

Should be accurate since LLMs are pretty good at qualitative questions.

ChatGPT Here's a clear comparison of the two pooling strategies, based on the gradient behavior we just derived:

Max Pooling - **Forward pass:** Selects the strongest activation in each region. This emphasizes the most discriminative local features (edges, corners, textures). - **Backward pass:** The gradient is routed only to the location of the maximum. This creates **sparse gradients**, which encourage the network to focus on the most important features. - **Advantages:** - Provides **translation invariance** by ensuring that the presence of a feature (not its exact position) is preserved. - Helps with noise robustness, since small fluctuations that aren't maxima get ignored. - Encourages the network to learn sharp, localized detectors.

Average Pooling - **Forward pass:** Averages activations in each region. This captures the overall “presence” of a feature rather than just the strongest response. - **Backward pass:** The gradient is evenly distributed across all inputs in the region, leading to **smoother gradient flow**. - **Advantages:** - Retains more contextual information than max pooling. - Produces less sparse signals, which can help training stability. - Can be useful in tasks where overall texture or background matters (e.g., scene classification).

Summary: - Max pooling highlights the **most salient feature**, making it good for object recognition tasks. - Average pooling preserves **global information**, making it better for capturing overall patterns or when fine-grained detail is less important.

Would you like me to also discuss why **modern CNNs often replace pooling entirely with strided convolutions**, and how that compares?

Correct answers generated in one shot. This is not very surprising since this question involves some relatively basic computation.