Question 1 calculations

hw4.pdf
PDF

Do questions 1,2,3,4, and 7 please. Make sure all calculations and explanations are accurate and detailed. Explain all math. If you are unsure, do not guess, just ask for clarification or help.

Reasoned for 19m 52s

I'll go question by question and part by part. I'll flag any assumptions I have to make (e.g. about RMS definition or network depth) so you can adjust to your course's exact conventions.

---

1. Newton–Schulz Runtime

We update a matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ with

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

We're told: multiplying an $n \times m$ matrix by an $m \times p$ matrix takes time $c\,nmp$.

(a) Runtime of one iteration (naïve implementation)

We need to compute:

1. $A = WW^\top$ (size $d_{\text{out}} \times d_{\text{out}}$)
2. $B = 3I_{d_{\text{out}}} - A$
3. $C = BW$
4. $p(W) = \frac{1}{2}C$

Only 1 and 3 involve matrix–matrix multiplies (dominant cost).

1. **Compute** $A = WW^\top$

- Shape: W is $d_{\text{out}} \times d_{\text{in}}$, $W^\top$ is $d_{\text{in}} \times d_{\text{out}}$.
- Using the given rule, runtime:

$$c\,(d_{\text{out}})(d_{\text{in}})(d_{\text{out}}) = c\,d_{\text{out}}^2 d_{\text{in}}.$$

2. **Compute $C = (3I_{d_{\text{out}}} - A)W$**

- $B = 3I_{d_{\text{out}}} - A$ is matrix add/subtract and scaling, which is $O(d_{\text{out}}^2)$, but we're told to ignore non-matmul costs.
- Shape: B is $d_{\text{out}} \times d_{\text{out}}$, W is $d_{\text{out}} \times d_{\text{in}}$.
- Runtime:

$$c\,(d_{\text{out}})(d_{\text{out}})(d_{\text{in}}) = c\,d_{\text{out}}^2 d_{\text{in}}.$$

3. The final scaling by $1/2$ is negligible.

Total runtime per iteration (naïve):

$$T_{\text{naive}} = c\,d_{\text{out}}^2 d_{\text{in}} + c\,d_{\text{out}}^2 d_{\text{in}} = 2c\,d_{\text{out}}^2 d_{\text{in}}.$$

(b) Faster when $d_{\text{out}} \gg d_{\text{in}}$ using $W^\top W$

We're told to consider the Gram matrix $W^\top W$.

Key algebra trick:

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

So

$$\begin{aligned} p(W) &= \frac{1}{2}\left(3I - WW^\top\right)W \\ &= \frac{1}{2}\left(3W - (WW^\top)W\right) \\ &= \frac{1}{2}\left(3W - W(W^\top W)\right). \end{aligned}$$

Let $G = W^\top W$ (Gram matrix, size $d_{\text{in}} \times d_{\text{in}}$). Then we can compute:

1. $G = W^\top W$
2. $H = WG$
3. $p(W) = \frac{1}{2}(3W - H)$ (linear-time ops)

Now compute the runtimes:

1. **Compute $G = W^\top W$**

- $W^\top$ is $d_{\text{in}} \times d_{\text{out}}$, W is $d_{\text{out}} \times d_{\text{in}}$.
- Runtime:

$$c\,(d_{\text{in}})(d_{\text{out}})(d_{\text{in}}) = c\,d_{\text{out}} d_{\text{in}}^2.$$

2. **Compute $H = WG$**

- W is $d_{\text{out}} \times d_{\text{in}}$, G is $d_{\text{in}} \times d_{\text{in}}$.
- Runtime:

$$c\,(d_{\text{out}})(d_{\text{in}})(d_{\text{in}}) = c\,d_{\text{out}}d_{\text{in}}^2.$$

3. Scalar multiplies/adds again negligible.

Total runtime per iteration (Gram-trick):

$$T_{\text{Gram}} = 2c\,d_{\text{out}}d_{\text{in}}^2.$$

Speedup factor:

$$\frac{T_{\text{naive}}}{T_{\text{Gram}}} = \frac{2c\,d_{\text{out}}^2 d_{\text{in}}}{2c\,d_{\text{out}}d_{\text{in}}^2} = \frac{d_{\text{out}}}{d_{\text{in}}}.$$

So when $d_{\text{out}} \gg d_{\text{in}}$, the Gram formulation is much faster:

- Naïve: $O(d_{\text{out}}^2 d_{\text{in}})$
- Gram: $O(d_{\text{out}}d_{\text{in}}^2)$.

---

2. MuP at the Unit Scale

We consider a dense layer

$$y = cWx$$

with parameters W initialized i.i.d. $\mathcal{N}(0, 1)$, and an external scalar c.

Let d_{in} be the input dimension (fan-in).

(a) Why initialize with $\mathcal{N}(0, 1)$ instead of Xavier in fp8?

In fp8 we have only 255 representable values.

- **Xavier/He** initializations scale weights as $\mathcal{N}(0, \sigma^2)$ with $\sigma \sim 1/\sqrt{d_{\text{in}}}$ (or similar), so for wide layers σ becomes very small.
- In fp8, when the true weight distribution is very narrow around 0, **most values underflow to 0 or collapse to a tiny subset of the 255 representable values**. That wastes dynamic range and effectively reduces many different weights to the same quantized value.
- If we instead sample $w_{ij} \sim \mathcal{N}(0, 1)$, the weights are spread across the full fp8 grid. This:

- uses the available representable values more uniformly,
- keeps each parameter at roughly **unit scale**, independent of layer width, which is exactly what MuP / maximal-update parameterization likes: the scaling is handled separately rather than baked into the raw weights.

We then recover Xavier-like behavior by scaling activations with the **float** scalar c, which is stored in higher precision, rather than by shrinking the underlying fp8 weights.

(b) Choosing c to recover Xavier-like forward variance

Assume:

- $w_{ij} \sim \mathcal{N}(0,1)$,
- x_i are i.i.d., mean 0, variance $\mathrm{Var}(x_i) = v_x$,
- $y = cWx$, and a single output coordinate is:

$$y_j = c \sum_{i=1}^{d_{\text{in}}} w_{ji} x_i.$$

Because w_{ji} and x_i are independent and zero-mean:

$$\mathbb{E}[y_j] = 0.$$

For the variance:

$$\begin{aligned}
\mathrm{Var}(y_j) &= c^2 \,\mathrm{Var}\Big(\sum_{i=1}^{d_{\text{in}}} w_{ji} x_i\Big) \\
&= c^2 \sum_{i=1}^{d_{\text{in}}} \mathrm{Var}(w_{ji} x_i) \quad \text{(cross-terms vanish by independence and zero mean)} \\
&= c^2 \sum_{i=1}^{d_{\text{in}}} \mathbb{E}[w_{ji}^2 x_i^2] \\
&= c^2 \sum_{i=1}^{d_{\text{in}}} \mathbb{E}[w_{ji}^2]\,\mathbb{E}[x_i^2] \quad \text{(independent)} \\
&= c^2 \sum_{i=1}^{d_{\text{in}}} (1)\, v_x = c^2 d_{\text{in}} v_x.
\end{aligned}$$

Xavier's goal is roughly to keep $\mathrm{Var}(y_j) \approx \mathrm{Var}(x_i) = v_x$. Thus we require:

$$c^2 d_{\text{in}} v_x = v_x \quad \Rightarrow \quad c^2 d_{\text{in}} = 1 \quad \Rightarrow \quad c = \frac{1}{\sqrt{d_{\text{in}}}}$$

(choosing the positive root).

So to mimic a Xavier-style initialization with unit-variance weights, we take

$$\boxed{c = \frac{1}{\sqrt{d_{\text{in}}}}.}$$

(c) Spectral norm bound on ΔW so Δy has RMS ≤ 1

We want to control the update

$$\Delta y = c \Delta W x$$

so that for any x with RMS norm 1, the RMS norm of Δy is at most 1.

Let:

- Input dimension be d_{in}, output dimension d_{out}.
- For a vector $v \in \mathbb{R}^d$, **RMS norm**: $\|v\|_{\text{RMS}} := \sqrt{\frac{1}{d} \sum_{i=1}^{d} v_i^2} = \frac{\|v\|_2}{\sqrt{d}}$.

Given $\|x\|_{\text{RMS}} = 1$, we have $\|x\|_2 = \sqrt{d_{\text{in}}}$.

For $\Delta y = c \Delta W x$:

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

Use the spectral norm bound $\|\Delta W x\|_2 \le \|\Delta W\|_2 \|x\|_2$:

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

To ensure $\|\Delta y\|_{\text{RMS}} \le 1$ for any such x, we need:

$$|c| \|\Delta W\|_2 \sqrt{\frac{d_{\text{in}}}{d_{\text{out}}}} \le 1 \quad \Rightarrow \quad \boxed{\|\Delta W\|_2 \le \frac{1}{|c|} \sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}}.}$$

That's the spectral norm constraint.

(d) SignSGD learning rate α under that constraint

Mini-batch size is 1. For a fully-connected layer, the true gradient has the form

$$\nabla_W L = g x^\top,$$

where $g \in \mathbb{R}^{d_\text{out}}$ is the upstream gradient and $x \in \mathbb{R}^{d_\text{in}}$ is the input.

For SignSGD, we use

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

Each element of $\text{sign}(\nabla_W L)$ is

$$\text{sign}(g_i x_j) = \text{sign}(g_i) \, \text{sign}(x_j),$$

so

$$\text{sign}(\nabla_W L) = a b^\top$$

where $a_i = \text{sign}(g_i) \in \{-1, 0, 1\}$ and $b_j = \text{sign}(x_j) \in \{-1, 0, 1\}$.

This is a rank-1 matrix. The spectral norm of a rank-1 outer product is the product of the vector ℓ_2 norms:

$$\|ab^\top\|_2 = \|a\|_2 \|b\|_2.$$

In the "worst case" where no entry is exactly zero, we have:

- $\|a\|_2 = \sqrt{\sum_{i=1}^{d_\text{out}} a_i^2} = \sqrt{d_\text{out}}$.
- $\|b\|_2 = \sqrt{\sum_{j=1}^{d_\text{in}} b_j^2} = \sqrt{d_\text{in}}$.

So

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

For the update:

$$\|\Delta W\|_2 = |\alpha| \, \|\text{sign}(\nabla_W L)\|_2 = |\alpha| \sqrt{d_\text{out} d_\text{in}}.$$

To satisfy the constraint from (c),

$$|\alpha| \sqrt{d_\text{out} d_\text{in}} \le \frac{1}{|c|} \sqrt{\frac{d_\text{out}}{d_\text{in}}}.$$

Solve for α:

1. Divide both sides by $\sqrt{d_{\text{out}} d_{\text{in}}}$ (positive):

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

2. Combine square roots:

$$\sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}} \cdot \frac{1}{\sqrt{d_{\text{out}} d_{\text{in}}}} = \sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}} \cdot \frac{1}{\sqrt{d_{\text{out}}}\sqrt{d_{\text{in}}}} = \frac{1}{d_{\text{in}}}.$$

So

$$|\alpha| \leq \frac{1}{|c| d_{\text{in}}}.$$

A maximal safe choice is

$$\boxed{\alpha = \frac{1}{c\, d_{\text{in}}}}$$

(assuming $c > 0$). This works for rectangular $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$.

(e) Muon-style orthogonalized update: learning rate α

We perform SVD:

$$\nabla_W L = U \Sigma V^\top, \quad \Delta W = \alpha U V^\top.$$

Here $U \in \mathbb{R}^{d_{\text{out}} \times r}$ and $V \in \mathbb{R}^{d_{\text{in}} \times r}$ have orthonormal columns, and r is the rank.

- Consider $M := UV^\top$. Its spectral norm is 1:
  - $MM^\top = UV^\top V U^\top = U I_r U^\top = UU^\top$.
  - The eigenvalues of $UU^\top$ are 1 (with multiplicity r) and 0 otherwise, so the largest singular value of M is 1.

Thus:

$$\|\Delta W\|_2 = \|\alpha U V^\top\|_2 = |\alpha| \|UV^\top\|_2 = |\alpha|.$$

To satisfy the same bound as in (c):

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

So a maximal choice is

$$\boxed{\alpha = \frac{1}{c}\sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}}.}$$

(f) When can we ignore the scale of intermediate backpropagated gradients?

We have a stack of linear layers (no activations):

$$x_{n+1} = c_n W_n x_n.$$

Backprop recursion:

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

The parameter gradient is

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

Now suppose we **rescale** the intermediate gradient at layer $n+1$ by some scalar $\lambda_n > 0$:

$$\tilde{g}_{n+1} := \lambda_n \nabla_{x_{n+1}} L.$$

Then

$$\nabla_{W_n} L \propto \tilde{g}_{n+1} x_n^\top$$

is simply scaled by λ_n relative to the original gradient.

Algorithms like **SignGD**, **Adam (with normalized steps)**, and **Muon** are (approximately) **invariant to global gradient scale**: multiplying the whole gradient by a positive constant doesn't change the update *direction* and often doesn't change the actual update (after normalization/clipping/etc.).

So in a setting where **every layer is updated by a scale-invariant optimizer** (SignGD / Adam-with-normalization / Muon), we can:

- arbitrarily scale $\nabla_{x_n} L$ at each layer,
- as long as we use that scaled version consistently in $\nabla_{W_n} L$.

The *direction* of the parameter update remains unchanged. That means the **absolute scale of intermediate backpropagated gradients can be ignored** (or normalized for numerical convenience) in such a setup; we only need their direction (relative shape across coordinates).

(g) Explosion / vanishing of backprop gradients, and stabilizing constants

We again have $x_{n+1} = c_n W_n x_n$, and the backprop recursion

$$g_n := \nabla_{x_n} L = c_n W_n^\top g_{n+1}.$$

We are told to assume:

- W_n is rank-1,
- each parameter is **unit-scaled** (entries of order 1),
- for rank-1 matrices, the spectral norm equals the Frobenius norm.

For a rank-1 matrix with all entries of order 1 and shape $d_{\text{out}} \times d_{\text{in}}$:

- There are $d_{\text{out}} d_{\text{in}}$ entries.
- Frobenius norm squared:

$$\|W_n\|_F^2 \approx d_{\text{out}} d_{\text{in}} \quad \Rightarrow \quad \|W_n\|_F \approx \sqrt{d_{\text{out}} d_{\text{in}}}.$$

- Spectral norm $\|W_n\|_2 = \|W_n\|_F \approx \sqrt{d_{\text{out}} d_{\text{in}}}$.

Thus each backprop step roughly multiplies gradient magnitude by

$$\text{factor}_n \approx c_n \|W_n\|_2 \approx c_n \sqrt{d_{\text{out}} d_{\text{in}}}.$$

If we choose c_n as in (b) for forward stability:

$$c_n = \frac{1}{\sqrt{d_{\text{in}}}},$$

then

$$\text{factor}_n \approx \frac{1}{\sqrt{d_{\text{in}}}} \sqrt{d_{\text{out}} d_{\text{in}}} = \sqrt{d_{\text{out}}}.$$

- If $d_{\text{out}} > 1$, then $\sqrt{d_{\text{out}}} > 1$ and **gradients explode** exponentially with depth.
- If $d_{\text{out}} < 1$ (not really meaningful in this discrete setting), they would vanish.

So **yes**, without adjustments, intermediate backpropagated gradients will typically **explode** (in the usual wide-layer case).

To stabilize them, we can **renormalize each gradient** after every layer by the inverse of this factor. That is, multiply g_n by:

$$\boxed{\beta_n = \frac{1}{c_n \|W_n\|_2} \approx \frac{1}{c_n \sqrt{d_{\text{out}} d_{\text{in}}}}.}$$

- With $c_n = 1/\sqrt{d_{\text{in}}}$, this simplifies to

$$\beta_n \approx \frac{1}{(1/\sqrt{d_{\text{in}}})\sqrt{d_{\text{out}} d_{\text{in}}}} = \frac{1}{\sqrt{d_{\text{out}}}}.$$

So in a square $d \times d$ layer, you'd multiply the gradient after backpropagating through that layer by roughly $1/\sqrt{d}$. That keeps the expected gradient scale roughly constant across many layers.

---

3. Understanding Convolution as an FIR Filter

We have a rectangular (boxcar) signal of length L:

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

In the figure they show $L = 7$, so I'll use $L = 7$ explicitly.

The impulse response is

h(n) = \left(\tfrac12\right)^n u(n) = \begin{cases} \left(\tfrac12\right)^n, & n = 0,1,2,\dots,\\ 0, & \text{otherwise}. \end{cases} \] :contentReference[oaicite:12]{index=12} --- ### (a) Compute \(y(n) = (x*h)(n)\), plot from \(n=-6\) to \(12\) By definition of discrete convolution: \[y(n) = (x*h)(n) = \sum_{k=-\infty}^{\infty} x(k)\,h(n-k). \] :contentReference[oaicite:13]{index=13} Because \(x(k)\) is nonzero only for \(k=0,\dots,6\), we can write: \[y(n) = \sum_{k=0}^{6} h(n-k).

Recall $h(m) = (1/2)^m$ for $m \geq 0$ and 0 otherwise, so the summation effectively only includes terms where $n - k \geq 0$, i.e. $k \leq n$.

So for each n:

- If $n < 0$: then for all $k \geq 0$, $n - k < 0$, so $h(n-k) = 0$.
  Hence $y(n) = 0$.
- If $0 \leq n \leq 6$: allowed k values are $k = 0, 1, \dots, n$ (cannot exceed 6).

$$y(n) = \sum_{k=0}^{n} \left(\tfrac{1}{2}\right)^{n-k} = \sum_{r=0}^{n} \left(\tfrac{1}{2}\right)^{r} \quad (\text{change variable } r = n-k)$$
$$= \frac{1-(1/2)^{n+1}}{1-1/2} = 2\left(1-\left(\tfrac{1}{2}\right)^{n+1}\right).$$

- If $n \geq 7$: we can use all $k = 0, \dots, 6$, since $n-k \geq n-6 \geq 1$.

$$y(n) = \sum_{k=0}^{6} \left(\tfrac{1}{2}\right)^{n-k} = \left(\tfrac{1}{2}\right)^{n-6} \sum_{j=0}^{6} \left(\tfrac{1}{2}\right)^{j}$$
$$= \left(\tfrac{1}{2}\right)^{n-6} \cdot \frac{1-(1/2)^7}{1-1/2}$$
$$= 2\big(1-(1/2)^7\big)\left(\tfrac{1}{2}\right)^{n-6}.$$

Since $(1/2)^7 = 1/128$, we have $1-(1/2)^7 = 127/128$, so the tail is:

$$y(n) = 2 \cdot \frac{127}{128}\left(\tfrac{1}{2}\right)^{n-6} = \frac{254}{128} \cdot 2^{-(n-6)} \quad \text{for } n \geq 7.$$

Piecewise formula:

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

For plotting from $n = -6$ to 12, we can tabulate values (approximate decimals):

- $n = -6, -5, -4, -3, -2, -1$: $y(n) = 0$
- $n = 0$: $2(1-(1/2)^1) = 2(1-1/2) = 1$
- $n = 1$: $2(1-(1/2)^2) = 2(1-1/4) = 2 \cdot \frac{3}{4} = 1.5$
- $n = 2$: $2(1-(1/2)^3) = 2(1-1/8) = 2 \cdot \frac{7}{8} = 1.75$
- $n = 3$: $2(1-1/16) = 2 \cdot \frac{15}{16} = 1.875$
- $n = 4$: $2(1-1/32) = 2 \cdot \frac{31}{32} = 1.9375$
- $n = 5$: $2(1-1/64) = 2 \cdot \frac{63}{64} = 1.96875$
- $n = 6$: $2(1-1/128) = 2 \cdot \frac{127}{128} \approx 1.984375$
- Tail uses the exponential formula:
  - $n = 7$: $2(1-1/128) \cdot (1/2)^1 = \frac{127}{64} \cdot \frac{1}{2} = \frac{127}{128} \approx 0.9921875$
  - $n = 8$: multiply by $\frac{1}{2}$ again ≈ 0.49609375
  - $n = 9$: ≈0.248046875
  - $n = 10$: ≈0.1240234375
  - $n = 11$: ≈0.06201171875

- $n = 12$: ≈0.031005859375

If you sketch those, you get a rising ramp from 0 to just under 2 over $n = 0$ to 6, then an exponentially decaying tail.

(b) Convolution with shifted input $x_2(n) = x(n-5)$ and property

We define

$$x_2(n) = x(n-5), \quad N = 5. : contentReference[oaicite:14]index = 14$$

Let

$$y_2(n) = (h * x_2)(n).$$

Use the **time-shift property of convolution**:

For any signals x and h, and integer shift N,

$$x_2(n) = x(n-N) \quad \Rightarrow \quad (h * x_2)(n) = (h * x)(n-N).$$

So

$$\boxed{y_2(n) = y(n-5).}$$

Thus the output is exactly the same shape as in (a), but shifted right by 5.

Property identified: time-shift invariance (or shift equivariance) of convolution.

(c) 2D convolution with given x and Sobel-like filter h

We're given:

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

We are to compute $y = x * h$ with **no padding** and stride 1.

The 2D convolution definition (as in the statement) is:

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

i.e. we flip the kernel in both dimensions ("flip and drag").

Flipping h horizontally and vertically yields:

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

So convolution with h is equivalent to **correlation with $\tilde{h}$**.

The input is 5×5, kernel 3×3, no padding, stride 1 ⇒ output is 3×3. For each output position, we take the corresponding 3×3 patch of x, multiply elementwise by $\tilde{h}$, and sum.

Let's compute one position explicitly, say the top-left output $y_{1,1}$:

- Patch of x (rows 1–3, cols 1–3):

$$P = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \\ 11 & 12 & 13 \end{bmatrix}.$$

- Elementwise multiply $P \odot \tilde{h}$:

$$\begin{bmatrix} 1 \cdot 1 & 2 \cdot 2 & 3 \cdot 1 \\ 6 \cdot 0 & 7 \cdot 0 & 8 \cdot 0 \\ 11 \cdot (-1) & 12 \cdot (-2) & 13 \cdot (-1) \end{bmatrix} = \begin{bmatrix} 1 & 4 & 3 \\ 0 & 0 & 0 \\ -11 & -24 & -13 \end{bmatrix}.$$

- Sum all entries:

$$1 + 4 + 3 - 11 - 24 - 13 = 8 - 48 = -40.$$

You can check (systematically or by symmetry) that every 3×3 patch of this numerically linear matrix produces the same sum with $\tilde{h}$, so **all entries of y are -40.**

Thus:

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

(d) With padding and stride

Now we consider the same convolution with:

1. stride $S = 1$, pad $P = 1$,
2. stride $S = 2$, pad $P = 1$.

We still use true convolution (with flipped kernel $\tilde{h}$).

Case (i): stride = 1, pad = 1

- Pad the 5×5 input with a 1-pixel border of zeros ⇒ padded size 7×7.
- Kernel 3×3, stride 1 ⇒ output size

$$7 - 3 + 1 = 5$$

  in each dimension ⇒ 5×5.

Compute top-left output $y_{1,1}$ as an example:

- Padded x (schematically):

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

- Patch at top-left (rows 1–3, cols 1–3 of the padded input):

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 6 & 7 \end{bmatrix}.$$

- Multiply with $\tilde{h}$ and sum:

$$\begin{aligned} &0 \cdot 1 + 0 \cdot 2 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 + 2 \cdot 0 \\ &\quad + 0 \cdot (-1) + 6 \cdot (-2) + 7 \cdot (-1) = -12 - 7 = -19. \end{aligned}$$

Carrying out all positions (as I checked programmatically) gives:

$$\boxed{y_{\text{pad}=1,S=1} = \begin{bmatrix} -19 & -29 & -39 & -49 & -31 \\ -34 & -40 & -40 & -40 & -46 \\ -59 & -60 & -60 & -60 & -61 \\ -84 & -80 & -80 & -80 & -74 \\ -49 & -51 & -53 & -55 & -41 \end{bmatrix}}.$$

(You'd typically only need the mechanism; I'm giving the full numeric matrix in case your homework wants explicit values.)

Case (ii): stride = 2, pad = 1

Same padded 7×7 input, same kernel, but stride $S = 2$:

- Output size:

$$\text{out\_dim} = \left\lfloor \frac{7-3}{2} \right\rfloor + 1 = \left\lfloor \frac{4}{2} \right\rfloor + 1 = 2 + 1 = 3,$$

  so output is 3×3.

Using stride-2 "jumps" between patches and the same conv rule, the resulting output is:

$$\boxed{y_{\text{pad}=1,S=2} = \begin{bmatrix} -19 & -39 & -31 \\ -59 & -60 & -61 \\ -49 & -53 & -41 \end{bmatrix}.}$$

---

4. Feature Dimensions in a CNN

Assume input feature shape is $W \times H \times C$ (width, height, channels).

(a) Conv layer: parameter count and output shape

Hyperparameters:

- filter size K (assume square $K \times K$),
- padding P,
- stride S,
- number of filters F.

Each filter has shape $K \times K \times C$:

- weights per filter: K^2C,
- for F filters: K^2CF weights,
- plus 1 bias per filter ⇒ F biases.

Total parameters:

$$\boxed{K^2CF + F.}$$

Output shape:

Width dimension:

$$W_{\text{out}} = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1.$$

Height:

$$H_{\text{out}} = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1.$$

Channels = number of filters F.

So output feature shape:

$$\boxed{W_{\text{out}} \times H_{\text{out}} \times F.}$$

(b) Max pooling layer output shape

Given:

- pooling filter size K,
- stride S,
- no padding (usual pooling).

Then for input $W \times H \times C$:

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

channels remain the same C.

So:

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

(c) Receptive field of last output after L conv layers (stride 1, kernel size K)

Each conv layer (stride 1, kernel K) increases the receptive field by $K - 1$ in each dimension.

Let R_ℓ be the receptive field size (e.g. width) after layer ℓ.

- Before any conv: $R_0 = 1$ (each input pixel only "sees" itself).

- After first layer: $R_1 = K$.
- Each additional layer adds $K - 1$:
  $R_{\ell+1} = R_\ell + (K - 1)$.

Solve this recurrence:

$$\begin{aligned} R_L &= R_1 + (L-1)(K-1) \\ &= K + (L-1)(K-1) \\ &= (K-1)L + 1. \end{aligned}$$

So the receptive field of the last output (in 1D per dimension) is:

$$\boxed{R_L = (K-1)L + 1.}$$

(Interpreted per spatial dimension; in 2D it's $R_L \times R_L$.)

(d) Effect of a 2×2, stride-2 downsampling layer on receptive field & computation

A max-pooling layer with kernel $K = 2$ and stride $S = 2$ takes **blocks of** 2×2 in the input and emits one output per block.

- Immediately after such a pooling layer, each output unit's receptive field (relative to that layer's input) is exactly 2×2.
- If the input to the pooling layer had a receptive field R (in pixels of the original image), then the new receptive field is:

  $$R_{\text{new}} = R + (K-1)J,$$

  where J is the effective stride so far. For the very first layer, $J = 1$, so the receptive field grows from 1 to 2 – doubling in linear size and quadrupling in area. For deeper layers, this additive formula generalizes, but intuitively each pooling layer makes each feature depend on a larger chunk of the original image.

So **each pooling layer meaningfully increases receptive field**, in particular for 2×2 pooling it expands the region each subsequent feature "summarizes".

Computation advantage:

- The layer **halves the spatial resolution** in each dimension:
  - $W \to W/2$,

  - $H \to H/2$.
- A subsequent conv layer cost scales roughly like

$$\text{cost} \propto W_{\text{out}} H_{\text{out}} K^2 C_{\text{in}} C_{\text{out}}.$$

- Halving both W and H reduces the spatial term $W_{\text{out}} H_{\text{out}}$ by a factor of **4**.
- We often compensate by **doubling the channel count** (e.g. from C to $2C$) to maintain representational power. That multiplies cost by factor 2, but the spatial reduction by 4 dominates, so the net cost often drops by about a factor of 2.

So decreasing the spatial resolution **dramatically reduces the number of multiply–adds** in later layers, allowing deeper or wider networks for the same compute budget.

(e) Concrete CNN example: fill the table

Network pieces:

- **CONV3-F:** 3×3 conv with F filters, padding 1, stride 1.
- **POOL2:** 2×2 max pool, stride 2, pad 0.
- **FLATTEN**
- **FC-K:** fully-connected with K outputs. Biases included.

Given table:

Layer	# Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10	?	$28 \times 28 \times 10$
POOL2	0	$14 \times 14 \times 10$
CONV3-10	$3 \times 3 \times 10 \times 10 + 10$	?
POOL2	?	?
FLATTEN	0	490
FC-3	?	3

Let's fill in:

1. **First CONV3-10** (input channels $C = 1$):

- Weights: $3 \times 3 \times 1 \times 10 = 9 \times 10 = 90$.
- Biases: 10.
- Total: $90 + 10 = 100$.

Activations: $28 \times 28 \times 10$ (given).

2. **POOL2**:
   - No parameters: 0.
   - Spatial dims halved: $28 \to 14$, $28 \to 14$, channels unchanged ⇒ $14 \times 14 \times 10$ (given).
3. **Second CONV3-10** (input channels $C = 10$):
   - Parameters already written: $3 \times 3 \times 10 \times 10 + 10$.
     - If you expand: $3 \times 3 = 9$; $9 \times 10 \times 10 = 900$; $900 + 10 = 910$.
   - Output activations keep same spatial size (padding 1, stride 1) and filter count 10 ⇒ $\boxed{14 \times 14 \times 10}$.
4. **Second POOL2**:
   - #params: 0.
   - Spatial dims halved: $14 \to 7$, $14 \to 7$; channels 10 ⇒ $\boxed{7 \times 7 \times 10}$.
5. **FLATTEN**:
   - No parameters: 0.
   - Flatten $7 \times 7 \times 10 = 49 \times 10 = 490$ ⇒ activation size 490 (given).
6. **FC-3**:
   - Input dimension: 490.
   - Weights: 490×3.
     - $490 \times 3 = 490 + 490 + 490 = 1470$.
   - Biases: 3.
   - Total params: $1470 + 3 = 1473$.

So the completed table:

Layer	# Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10	$3 \times 3 \times 1 \times 10 + 10 = 100$	$28 \times 28 \times 10$
POOL2	0	$14 \times 14 \times 10$

Layer	# Parameters	Dimension of Activations
CONV3-10	$3 \times 3 \times 10 \times 10 + 10 = 910$	$14 \times 14 \times 10$
POOL2	0	$7 \times 7 \times 10$
FLATTEN	0	490
FC-3	$490 \times 3 + 3 = 1473$	3

(f) New architecture and equivariances

Architecture:

$$\text{CONV2-3} \to \text{ReLU} \to \text{CONV2-3} \to \text{ReLU} \to \text{GAP} \to \text{FC-3}.$$

- CONV2-3: 2×2 conv, 3 filters, stride 1, padding 1 (circular padding, i.e. wrap-around on an 8×8 grid).
- GAP: Global average pool (per-channel mean over entire spatial map).
- Input images x_1, x_2, x_3, x_4 are 8×8 binary “edge” patterns; outputs (before softmax) for some are given as:

$$g_1 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}, \quad g_2 = \begin{bmatrix} 0 \\ 0.8 \\ 0 \end{bmatrix}. : contentReference[oaicite:26]index = 26$$

We’re asked to compute g_3 and g_4 for new inputs x_3 and x_4 which are **translated versions** (shifted horizontally/vertically) of x_2 and x_1 respectively.

Key facts:

- Convolution with **circular padding** is **translation-equivariant** on the 2D torus: shifting the input simply shifts the feature maps.
- GAP then averages each channel over all spatial positions, which makes the result **translation-invariant**: a translation of the input does **not** change the per-channel mean.

So:

- x_3 is a horizontal shift of x_2 ⇒ its feature maps after the last conv layer are a shifted version of those for x_2. GAP wipes out the shift, so the per-channel averages are **identical**. Therefore,

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

- Similarly, x_4 is a vertical shift of x_1. Same argument ⇒ per-channel means unchanged, so

$$\boxed{g_4 = g_1 = \begin{bmatrix} 0.8 \\ 0 \\ 0 \end{bmatrix}.}$$

---

7. Weights and Gradients in a CNN

We consider a single-channel 2D conv layer:

- Input $X \in \mathbb{R}^{n \times n}$,
- Kernel $w \in \mathbb{R}^{k \times k}$,
- Output $Y \in \mathbb{R}^{m \times m}$ with $m = n - k + 1$ (no padding, stride 1).

The forward pass for each output:

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,\, j+l-1}\, w_{h,l}. :contentReference[oaicite:30]index=30$$

We denote upstream gradient $dY \in \mathbb{R}^{m \times m}$ with entries $dy_{i,j} = \frac{\partial L}{\partial y_{i,j}}$.

(a) Gradient to the weight matrix dw, and one-step SGD update

We want $dw_{h,l} = \frac{\partial L}{\partial w_{h,l}}$.

Use the chain rule:

$$\frac{\partial L}{\partial w_{h,l}} = \sum_{i,j} \frac{\partial L}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{h,l}} = \sum_{i,j} dy_{i,j} \frac{\partial y_{i,j}}{\partial w_{h,l}}.$$

From the forward expression:

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

So

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

Thus:

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

Collect these into a matrix $dw \in \mathbb{R}^{k\times k}$.

One-step SGD update (batch size 1, learning rate η):

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

In matrix form:

$$\boxed{w^{(\text{new})} = w - \eta\, dw.}$$

Interpretation: each kernel weight becomes a **weighted sum of overlapping patches of the input**, where weights are given by the upstream gradients. Over many steps and many images, w becomes a kind of **weighted average of training image patches**.

(b) Mean/variance of $dw_{h,l}$ vs image size n

Assumptions (for all i, j):

- $x_{i,j}$ are independent with
  - $\mathbb{E}[x_{i,j}] = 0$,
  - $\mathrm{Var}(x_{i,j}) = \sigma_x^2$.
- $dy_{i,j}$ are independent with
  - $\mathbb{E}[dy_{i,j}] = 0$,
  - $\mathrm{Var}(dy_{i,j}) = \sigma_g^2$.
- All $x_{i,j}$ are independent of all $dy_{p,q}$.

Recall:

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

Let's define

$$Z_{i,j} := dy_{i,j}\ x_{i+h-1,j+l-1}.$$

So

$$dw_{h,l} = \sum_{i=1}^{m}\sum_{j=1}^{m} Z_{i,j}.$$

Mean of $Z_{i,j}$:

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

Variance of $Z_{i,j}$:

$$\begin{aligned}\mathrm{Var}(Z_{i,j}) &= \mathbb{E}[Z_{i,j}^2] - (\mathbb{E}[Z_{i,j}])^2 \\ &= \mathbb{E}[dy_{i,j}^2 x_{i+h-1,j+l-1}^2] \\ &= \mathbb{E}[dy_{i,j}^2]\,\mathbb{E}[x_{i+h-1,j+l-1}^2] \quad \text{(independent)} \\ &= \sigma_g^2\sigma_x^2.\end{aligned}$$

Independence across (i, j):

- Different (i, j) use different pairs $(dy_{i,j}, x_{i+h-1,j+l-1})$.
- By independence assumptions, $Z_{i,j}$ are independent across (i, j).

So we can sum variances:

$$\mathrm{Var}(dw_{h,l}) = \sum_{i,j} \mathrm{Var}(Z_{i,j}) = m^2\sigma_g^2\sigma_x^2.$$

The mean is

$$\mathbb{E}[dw_{h,l}] = \sum_{i,j} \mathbb{E}[Z_{i,j}] = 0.$$

Recall $m = n - k + 1$ (no padding, stride 1). So

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

Standard deviation:

$$\mathrm{Std}(dw_{h,l}) = \sqrt{\mathrm{Var}(dw_{h,l})} = (n-k+1)\sigma_x\sigma_g.$$

For large n, $n-k+1 \approx n$. So the **asymptotic growth rate**:

$$\boxed{\mathbb{E}[dw_{h,l}] = 0, \quad \mathrm{Var}(dw_{h,l}) = (n-k+1)^2\sigma_x^2\sigma_g^2, \quad \mathrm{Std}(dw_{h,l}) \sim O(n).}$$

The standard deviation of the gradient **grows linearly in the image size n** (side length).

(c) Network with only 2×2 pooling layers: what is dX?

We now consider a network built only from **2×2 pooling layers** (no convs, no nonlinearities).

We'll separate the two cases.

2×2 Max-pooling only

Consider one 2×2 max-pooling operation with stride 2 on

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix},$$

output

$$y_{11} = \max(x_{11}, x_{12}, x_{21}, x_{22}).$$

Suppose the top-left value wins:

$$y_{11} = x_{11}. :contentReference[oaicite:37]index=37$$

Then:

- $\frac{\partial y_{11}}{\partial x_{11}} = 1$,
- $\frac{\partial y_{11}}{\partial x_{12}} = 0$,
- $\frac{\partial y_{11}}{\partial x_{21}} = 0$,
- $\frac{\partial y_{11}}{\partial x_{22}} = 0$.

If the upstream gradient is $dy_{11} = \frac{\partial L}{\partial y_{11}}$, then by chain rule:

$$\begin{aligned}
dx_{11} &= \frac{\partial L}{\partial x_{11}} = dy_{11} \cdot \frac{\partial y_{11}}{\partial x_{11}} = dy_{11}, \\
dx_{12} &= dy_{11} \cdot 0 = 0, \\
dx_{21} &= 0, \\
dx_{22} &= 0.
\end{aligned}$$

So within each 2×2 pooling region, **all of the gradient flows to the max location**, others get 0.

In a deeper network with only 2×2 max-pools stacked until a single scalar output:

- Each pooling layer picks a single "winner" in each 2×2 block.
- Ultimately there is exactly **one input pixel** that wins the max at each scale; the final scalar output is the global max.
- The gradient from the scalar output flows **only to that global-max pixel**; all other pixels have gradient 0.

So for such a network,

$$\boxed{dX_{i,j} = \begin{cases} \dfrac{\partial L}{\partial y}, & \text{if } x_{i,j} \text{ is the global max,} \\ 0, & \text{otherwise,} \end{cases}}$$

where y is the final scalar output (assuming a single scalar).

2×2 Average-pooling only
For a 2×2 average pool:

$$y_{11} = \frac{1}{4}(x_{11} + x_{12} + x_{21} + x_{22}).$$

Then:

$$\frac{\partial y_{11}}{\partial x_{pq}} = \frac{1}{4} \text{ for each } (p, q) \in \{1, 2\}^2.$$

So with upstream gradient dy_{11},

$$dx_{pq} = \frac{\partial L}{\partial x_{pq}} = dy_{11} \cdot \frac{1}{4}.$$

Stacking multiple average pools to get a single scalar output essentially computes the global average (up to a constant factor depending on exact architecture):

- Each pooling splits its gradient among 4 inputs by a factor $\frac{1}{4}$.
- Repeated across layers, every input pixel receives **the same gradient**, up to a global factor.

For a typical complete average-pool-only network that reduces to a single scalar mean of all N pixels $y = \frac{1}{N}\sum_{i,j} x_{i,j}$:

$$\frac{\partial y}{\partial x_{i,j}} = \frac{1}{N}.$$

So

$$\boxed{dX_{i,j} = \frac{1}{N}\frac{\partial L}{\partial y} \quad \text{for all } i, j.}$$

Gradients are **dense and uniformly spread** in the average-pooling-only case.

(d) Advantages of max vs average pooling (qualitative)

From the gradient behavior and forward behavior:

Max pooling:

- **Forward:** preserves the strongest activation in each local region, promoting *invariance to small translations* — if a feature shifts slightly within the pool window, the max remains similar.
- **Backward:** gradient flows only through the max locations, creating a **sparse** gradient:
  - Pros: highlights the most “important” features, can help focus learning on salient structures like edges.
  - Cons: other pixels in the region receive no gradient, so learning can be less smooth, potentially slower to refine subtle patterns.

Average pooling:

- **Forward:** computes local averages, producing smoother, more blurred representations; preserves information about all activations in a region.
- **Backward:** gradient is spread **uniformly** across all inputs in a region:
  - Pros: every location in the region gets some gradient signal, making optimization smoother and less brittle; often better when you care about overall presence of a pattern rather than the single strongest pixel.

   - Cons: can blur out sharp edges or peaks; less selective, weaker inductive bias for “pick the strongest feature”.

In practice:

- Max pooling is often good for detecting **sparse, localized features** such as edges and corners, giving strong translation robustness.
- Average/global average pooling is often used near the end of CNNs to summarize features across the whole image and reduce parameters, especially when you want to aggregate evidence rather than pick a single strongest activation.

If you’d like, I can also turn some of these into small numeric examples (e.g., explicitly enumerating all pooling gradients for a slightly bigger image) so you can sanity-check your intuition.

GPT 5- Pro did quite well on this homework. It got everything correct except question 3d, on which it made a calculation error/programatic error - it did not show full work trace, just said it expanded the work done before.

It was kind of surprising to me how just feeding ChatGPT the pdf would allow it to run properly, the only drawback being the reasoning time - it took 20+ minutes to get a response.

I verified all the other work - it seemed as though GPT 5-Pro was quite proficient at algebra/derivations, only the numbers were either calculated incorrectly or the python program was written wrong for q3d.