

HW11 answer explanation

1. LoRA

A common strategy for adapting a large pre-trained model to a new task is to update only a subset of its parameters, keeping the rest frozen. Low-Rank Adaptation (LoRA) offers a more flexible approach to this idea. In this problem, we focus on a single weight matrix W with m rows and ℓ columns, where W_0 is the pre-trained value. During LoRA-based training, W is replaced by $W_0 + AB$, where W_0 remains frozen and only A and B are learnable. Here, A is an $m \times k$ matrix and B is a $k \times \ell$ matrix, typically with $k \ll \min(m, \ell)$.

- (a) Suppose you are using LoRA to adapt a pretrained deep neural net to a new task and observe that the model is “underfitting” the training data. What would you try to adjust in the LoRA to get better performance?

- (b) Suppose both A and B are initialized to all zeros. Why will this cause problems for LoRA-based finetuning?

Remember, this is going to be trained using SGD-style updates over a training set with a loss function.

- (c) Consider the following pseudocode for LoRA initialization:

```
A = torch.nn.Parameter(torch.empty(m, k))
B = torch.nn.Parameter(torch.empty(k, l))
torch.nn.init.xavier_uniform_(A)
torch.nn.init.xavier_uniform_(B)
```

Why might LoRA fine-tuning not work well with this initialization?

- (d) How much memory is required to store the LoRA adaptation weights (A and B)? Assume we are using floats (4 bytes per real number) and give your answer in bytes. How does this compare to storing a single full-rank adaptation matrix?

HW11

Answer this homework problem

Here's one way to answer each part.

Setup:

We write the adapted weight as

$$W = W_0 + AB, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times \ell}, \quad k \ll \min(m, \ell).$$

correct

(a) Model is underfitting – what to change in LoRA?

Underfitting means the adaptation is not expressive enough.

For LoRA the expressive power of the update AB is limited by its rank $\leq k$.

👉 I would increase the LoRA rank k (i.e., use larger A, B) so that AB can approximate a richer set of updates to W and the model has more capacity to fit the training data.

(You could also add LoRA to more layers, but the LoRA-specific knob here is k .)

correct

(b) Both A and B initialized to all zeros

At initialization,

$$AB = 0 \quad \Rightarrow \quad W = W_0,$$

so we start exactly from the pretrained weights, which sounds fine.

The problem is with the **gradients**.

Let $G = \frac{\partial L}{\partial W}$. Using the chain rule,

$$\frac{\partial L}{\partial A} = GB^\top, \quad \frac{\partial L}{\partial B} = A^\top G.$$

If $A = 0$ and $B = 0$ initially, then

$$\frac{\partial L}{\partial A} = G(0)^\top = 0, \quad \frac{\partial L}{\partial B} = 0^\top G = 0.$$

I really like how this walks through the chain rule step-by-step to show why the gradients vanish. It makes the mathematical reason for the failure very clear.

So **both gradients are zero**, SGD will never move A or B , and W will stay equal to W_0 forever. LoRA never actually fine-tunes the model.

correct

(c) Xavier init for both A and B

python

```
A = torch.nn.Parameter(torch.empty(m, k))
B = torch.nn.Parameter(torch.empty(k, l))
torch.nn.init.xavier_uniform_(A)
torch.nn.init.xavier_uniform_(B)
```

Now both A and B are random with non-tiny variance, so AB is also a non-negligible random matrix. That means at the start of fine-tuning

$$W = W_0 + AB$$

is **no longer close to the pretrained W_0** ; we have already added a sizable random low-rank perturbation.

correct

LoRA is supposed to start from the pretrained model and make *small* learned adjustments. With this initialization we effectively “damage” the good pretrained weights before learning even starts, so optimization behaves more like training from a poorly chosen initialization and can perform much worse.

(Practically, people usually initialize one factor small or to zero so that $AB \approx 0$ initially.)

correct

(d) Memory cost of LoRA vs full-rank update

Number of parameters in LoRA adaptation:

- $A: m \times k$ entries
- $B: k \times \ell$ entries

Total entries: $mk + k\ell = k(m + \ell)$.

Using 4 bytes per float:

$$\boxed{\text{LoRA memory} = 4k(m + \ell) \text{ bytes}}$$

A single full-rank adaptation matrix (same shape as W) has $m\ell$ entries, so:

$$\boxed{\text{Full-rank memory} = 4m\ell \text{ bytes}}$$

Comparison: LoRA uses a fraction

$$\frac{4k(m + \ell)}{4m\ell} = \frac{k(m + \ell)}{m\ell}$$

of the memory of a full-rank matrix. Since $k \ll \min(m, \ell)$, this fraction is **much smaller than 1**, so LoRA gives a large memory saving.

Homework 3: A Simple Reading Model

The Transformer (Fig. 1) is a modern piece of AI technology, based on a neural architecture, showing the state-of-the-art performance in many natural language processing tasks. In this assignment, you will implement a simplified version of the Transformer, which is called the “Zero-Layer Model”.

Task 1: The Simplest Transformer (Zero-Layer Model)

Figure 1 shows the architecture of the Zero-Layer Model. You have to implement the forward pass of the model, which takes a sequence of tokens T as input and outputs a sequence of logits L .

Variables

Variable	Type	Description
T	float tensor	Input tokens, each represented by a float vector. Each token is a word or a symbol in a document.
K^T	float tensor	“Key” matrix, which is obtained by multiplying T by a weight matrix W_K . It is used to calculate attention scores.
V^T	float tensor	“Value” matrix, which is obtained by multiplying T by a weight matrix W_V . It is used to store information about the words.
W_U	float tensor	“Query” matrix, which is obtained by multiplying T by a weight matrix W_Q . It is used to calculate attention scores.
W_E	float tensor	“Embedding” matrix. Maps each token to its embedding vector.
W_T	float tensor	“Output” matrix, which is obtained by multiplying L by a weight matrix W_O .
W_K, W_Q, W_V	float tensor	Query, key, and value weight matrices for attention heads. They are initialized with random values.
W_S	float tensor	“Scale” matrix, which is used to scale the attention scores before applying them to the “Value” matrix.
A^T	float tensor	“Attention” matrix, which is obtained by multiplying K^T and V^T and then dividing by the attention head size and applying softmax.
W_M	float tensor	“Multi-head” matrix, which is obtained by concatenating all attention heads and then multiplying by W_M .
W_O	float tensor	“Output” matrix, which is obtained by multiplying M by a weight matrix W_O .

Implementation Details

The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.py> contains the code for the zero-layer model. The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.ipynb> contains the Jupyter notebook for the zero-layer model. The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.pdf> contains the PDF of the zero-layer model.

Task 2: The Simplest Transformer (Zero-Layer Model)

Figure 2 shows the architecture of the Zero-Layer Model. You have to implement the forward pass of the model, which takes a sequence of tokens T as input and outputs a sequence of logits L .

Variables

Variable	Type	Description
T	float tensor	Input tokens, each represented by a float vector. Each token is a word or a symbol in a document.
K^T	float tensor	“Key” matrix, which is obtained by multiplying T by a weight matrix W_K . It is used to calculate attention scores.
V^T	float tensor	“Value” matrix, which is obtained by multiplying T by a weight matrix W_V . It is used to store information about the words.
W_U	float tensor	“Query” matrix, which is obtained by multiplying T by a weight matrix W_Q . It is used to calculate attention scores.
W_E	float tensor	“Embedding” matrix. Maps each token to its embedding vector.
W_T	float tensor	“Output” matrix, which is obtained by multiplying L by a weight matrix W_O .
W_K, W_Q, W_V	float tensor	Query, key, and value weight matrices for attention heads. They are initialized with random values.
W_S	float tensor	“Scale” matrix, which is used to scale the attention scores before applying them to the “Value” matrix.
A^T	float tensor	“Attention” matrix, which is obtained by multiplying K^T and V^T and then dividing by the attention head size and applying softmax.
W_M	float tensor	“Multi-head” matrix, which is obtained by concatenating all attention heads and then multiplying by W_M .
W_O	float tensor	“Output” matrix, which is obtained by multiplying M by a weight matrix W_O .

Implementation Details

The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.py> contains the code for the zero-layer model. The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.ipynb> contains the Jupyter notebook for the zero-layer model. The address <https://github.com/CS224n/fall2019/blob/master/hw3/transformer.pdf> contains the PDF of the zero-layer model.

Answer this homework problem

I'll go part by part and keep the notation from the handout.

(a) The Simplest Transformer (Zero-Layer Model)

correct

(i) Expression for the logits

We first embed and then unembed:

- Residual stream at “layer 0”:

$$X^0 = W_E T \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}.$$

- Logits:

$$L = W_U X^0 = W_U W_E T \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}.$$

So the required expression is

$$L = W_U W_E T.$$

correct

(ii) What algorithm does this implement?

For each position t , the t -th column of T is the one-hot vector of the token at that position, call it e_{w_t} .

The model does

$$x_t^0 = W_E e_{w_t} \quad (\text{lookup embedding}),$$

$$\ell_t = W_U x_t^0 = W_U W_E e_{w_t} \quad (\text{linear map to logits}).$$

So:

- Each position is processed **independently** with the same matrix $W_U W_E$.
- The prediction at position t depends **only on which token is at position t** (no context, no other tokens).
- Algorithmically: “for every token, look up its embedding and apply a fixed linear classifier to get logits.”

This is the baseline behavior of the transformer with no attention or layers.

(b) Multi-Head Attention: Concatenation vs Addition

We have H heads, each producing value output

$$r^h \in \mathbb{R}^{d_{\text{head}} \times n_{\text{context}}}.$$

correct

(i) Show equivalence of the two views

Concatenation view

Stack the heads vertically:

$$R^H = \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} \in \mathbb{R}^{(d_{\text{head}}H) \times n_{\text{context}}},$$

then apply a single large output matrix

$$W_O^H \in \mathbb{R}^{d_{\text{model}} \times (d_{\text{head}}H)}$$

to get

$$\mathbf{H} = W_O^H R^H.$$

Additive & independent view

Each head has its own output matrix

$$W_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}},$$

and the total output is

$$\mathbf{H} = \sum_{h=1}^H W_O^h r^h.$$

Construction of W_O^H :

Write the big matrix as a horizontal concatenation of the per-head matrices:

$$W_O^H = [W_O^1 \ W_O^2 \ \dots \ W_O^H] \in \mathbb{R}^{d_{\text{model}} \times (d_{\text{head}}H)}.$$

Then

$$W_O^H R^H = [W_O^1 \ \dots \ W_O^H] \begin{bmatrix} r^1 \\ \vdots \\ r^H \end{bmatrix} = \sum_{h=1}^H W_O^h r^h.$$

So the concatenation view and the additive-independent view give **identical outputs** for this choice of W_O^H .

correct

(ii) Advantage and disadvantage of each view

- Concatenation view

- *Advantage:* Matches standard implementations: all head outputs are concatenated and passed through a single linear layer. Convenient for coding and for thinking in terms of one big matrix.
- *Disadvantage:* Conceptually hides that each head is a separate "circuit" whose contribution is just added into the residual stream, making mechanistic

interpretation harder.

- **Additive & independent view**

- *Advantage:* Makes interpretability easier: each head has its own W_O^h and contributes additively to the residual stream, so you can analyze heads one at a time as independent modules.
 - *Disadvantage:* Less directly tied to the usual implementation and less compact to write; you have H matrices instead of one.
-

(c) The QK Circuit: Determining Attention Patterns

We have

$$Q_i = W_Q X_i, \quad K_j = W_K X_j,$$

and

correct

$$S_{ij} = Q_i^\top K_j.$$

(i) Bilinear form and W_{QK}

Compute:

$$S_{ij} = (W_Q X_i)^\top (W_K X_j) = X_i^\top W_Q^\top W_K X_j.$$

So we can write

$$S_{ij} = X_i^\top W_{QK} X_j \quad \text{with} \quad \boxed{W_{QK} := W_Q^\top W_K \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}}.$$

This is a bilinear form in the residual vectors X_i and X_j .

correct

(ii) Interpreting W_{QK} (toy $d_{\text{model}} = 3$ case)

Write $X_i = (x_{i1}, x_{i2}, x_{i3})^\top$,

$X_j = (x_{j1}, x_{j2}, x_{j3})^\top$.

We always use

$$S_{ij} = X_i^\top W_{QK} X_j.$$

(1) $W_{QK} = I$

Then $S_{ij} = X_i^\top X_j$, the usual dot product.

- High attention score when X_i and X_j point in similar directions (large positive dot product).
 - This head "looks for tokens whose residual vectors are similar to the query token in all dimensions."
-

$$(2) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We get

The way this breaks down what different W_{QK} matrices mean is super helpful,

$$W_{QK} X_j = (x_{j1}, 0, 0)^\top$$

especially the negative identity case showing how attention can avoid similar tokens, and

which I hadn't thought about before.

$$S_{ij} = X_i^\top (x_{j1}, 0, 0)^\top = x_{i1}x_{j1}.$$

- High score when the **first coordinates** of X_i and X_j are similar and large.
 - The second and third coordinates of X_i, X_j do not affect the score at all.
 - So this head “looks only at similarity along dimension 1” and is **blind** to information in dimensions 2 and 3 (for deciding where to attend).
-

$$(3) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Then

$$W_{QK}X_j = (x_{j1}, 0, -x_{j3})^\top,$$

$$S_{ij} = x_{i1}x_{j1} - x_{i3}x_{j3}.$$

- Large S_{ij} when
 - the first coordinates are similar (large positive $x_{i1}x_{j1}$), and
 - the third coordinates are **opposite** (so $-x_{i3}x_{j3}$ is large positive).
 - Intuitively, the head “looks for tokens that match the query on feature 1 but differ on feature 3.”
 - Compared to the identity case, feature 3 is treated as an “anti-similarity” dimension.
-

$$(4) W_{QK} = -I$$

Then

$$S_{ij} = X_i^\top (-I)X_j = -X_i^\top X_j.$$

- For fixed norms, this is largest when X_i and X_j are most **opposite** (largest negative dot product in the original space).
- So this head “prefers tokens whose residual vectors are *anti-aligned* with the query.”
- After the softmax over j , tokens most similar to the query (largest $X_i^\top X_j$) get the **lowest** scores (most negative S_{ij}), while tokens that are most dissimilar get the **highest** attention weights.

So W_{QK} encodes the question “what relationship between query and key vectors should lead to high attention?”

(d) The OV Circuit: Reading and Writing Information

We are given that head h has output

$$h(X^0) = W_O^h W_V^h X^0 A^{h\top} = W_{OV}^h X^0 A^{h\top},$$

where A^h is the attention matrix.

correct

(i) Attention-weighted average

We have:

- $X^0 \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$, columns X_j^0 .
- $A \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$, with A_{ij} = weight from destination i to source j .

- So A^\top has entries $(A^\top)_{ji} = A_{ij}$.

The i -th column of $X^0 A^\top$ is

$$(X^0 A^\top)_i = X^0 (A^\top e_i) = X^0 \begin{bmatrix} A_{i1} \\ A_{i2} \\ \vdots \\ A_{in_{\text{context}}} \end{bmatrix} = \sum_{j=1}^{n_{\text{context}}} A_{ij} X_j^0.$$

Thus

$$(X^0 A^\top)_i = \sum_{j=1}^{n_{\text{context}}} A_{ij} X_j^0,$$

an attention-weighted average of the source token vectors.

correct

(ii) Final residual stream X^{final}

With one attention layer and H heads, the residual update is the sum of each head's output, added to the original stream:

$$X^{\text{final}} = X^0 + \sum_{h=1}^H h^h(X^0) = X^0 + \sum_{h=1}^H W_{OV}^h X^0 A^{h\top}.$$

That's the desired expression in terms of X^0 and the heads' weight matrices.

correct

(iii) Column-space property

For a fixed head h , its output is

$$h_h(X^0) = W_{OV}^h X^0 A^{h\top}.$$

Let t be a destination position. The update vector written at position t is the t -th column:

$$h_h(X^0)_t = W_{OV}^h (X^0 A^{h\top})_t.$$

Define

$$v_t := (X^0 A^{h\top})_t \in \mathbb{R}^{d_{\text{model}}}.$$

Then

$$h_h(X^0)_t = W_{OV}^h v_t.$$

By definition, the **column space** $\text{Col}(W_{OV}^h)$ is the set

$$\{ W_{OV}^h v : v \in \mathbb{R}^{d_{\text{model}}} \}.$$

Since $h_h(X^0)_t$ is exactly of this form, we have

$$h_h(X^0)_t \in \text{Col}(W_{OV}^h).$$

So each head can only write into a low-dimensional subspace of the d_{model} -dimensional residual stream.

Correct

(e) Formalizing Read and Write Subspaces via SVD

(i) Rank bound and low-rankness

Recall $W_{OV} = W_O W_V$ with

$$W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}, \quad W_V \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}.$$

For any matrices, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$. Hence

$$\text{rank}(W_{OV}) = \text{rank}(W_O W_V) \leq \min(\text{rank}(W_O), \text{rank}(W_V)) \leq d_{\text{head}}.$$

In typical transformers, $d_{\text{head}} \ll d_{\text{model}}$ (e.g., 64 vs 768), so W_{OV} has rank much smaller than its dimension: it is a **low-rank** matrix.

Correct

Type text here

(ii) Decomposition via compact SVD

Let the compact SVD be

$$W_{OV} = U \Sigma V^\top,$$

with

- $U \in \mathbb{R}^{d_{\text{model}} \times r}$,
- $\Sigma \in \mathbb{R}^{r \times r}$ (diagonal with positive singular values $\sigma_1, \dots, \sigma_r$),
- $V \in \mathbb{R}^{d_{\text{model}} \times r}$,
- $r = \text{rank}(W_{OV}) \leq d_{\text{head}}$.

For a source token vector $X_{\text{src}} \in \mathbb{R}^{d_{\text{model}}}$,

$$W_{OV} X_{\text{src}} = U \Sigma V^\top X_{\text{src}}.$$

Let

$$\alpha = V^\top X_{\text{src}} \in \mathbb{R}^r, \quad \text{so} \quad \alpha_k = V_k^\top X_{\text{src}}.$$

Then

$$\Sigma \alpha = \begin{bmatrix} \sigma_1 \alpha_1 \\ \vdots \\ \sigma_r \alpha_r \end{bmatrix},$$

and multiplying by U :

$$U \Sigma \alpha = \sum_{k=1}^r (\sigma_k \alpha_k) U_k = \sum_{k=1}^r \sigma_k (V_k^\top X_{\text{src}}) U_k.$$

Thus

$$W_{OV} X_{\text{src}} = \sum_{k=1}^r \sigma_k (V_k^\top X_{\text{src}}) U_k$$

as required.

Correct

(iii) Read and write subspaces; orthogonal information

From the decomposition

$$W_{OV}X_{\text{src}} = \sum_{k=1}^r \sigma_k(V_k^\top X_{\text{src}})U_k,$$

- **Read subspace:** The head first projects X_{src} onto the directions V_k via $V^\top X_{\text{src}}$. So the head can only “read” components of X_{src} lying in

$$\boxed{\text{Read subspace} = \text{span}(V_1, \dots, V_r)}.$$

- **Write subspace:** The output is a linear combination of the vectors U_k . So the head can only “write” into

$$\boxed{\text{Write subspace} = \text{span}(U_1, \dots, U_r)}.$$

- **Information orthogonal to the read subspace:**

If X_{src} is orthogonal to all V_k , then $V_k^\top X_{\text{src}} = 0$ for all k , so

$$W_{OV}X_{\text{src}} = 0.$$

Thus any component of X_{src} orthogonal to $\text{span}(V_1, \dots, V_r)$ is completely **ignored** by the OV circuit (it cannot be read or propagated by this head).

correct

(iv) Connection to the value projection W_V

The value vector is

$$v = W_V X_{\text{src}} \in \mathbb{R}^{d_{\text{head}}}.$$

The first matrix that touches X_{src} in the OV circuit is W_V , so:

- W_V can only “see” components of X_{src} lying in the **row space** of W_V , which is the same as the column space of W_V^\top and has dimension at most d_{head} .
- The read subspace $\text{span}(V_1, \dots, V_r)$ is a subspace of this row space, since $W_{OV} = W_O W_V$ and the SVD’s V lives inside the space of directions that affect $W_V X_{\text{src}}$.

Therefore it is natural that the read subspace dimension $r = \text{rank}(W_{OV})$ is at most d_{head} : the head cannot read more independent directions than can be encoded in its d_{head} -dimensional value vector v .

correct

(v) Synthesis: Heads as information-moving channels

Conceptually, each attention head moves information “from the residual stream of one token to another.” The SVD

$$W_{OV} = U \Sigma V^\top$$

gives a complete description of this movement:

- V defines **which directions** in the source residual stream can be read (the read subspace).
- Σ sets the **strength** or gain of each read-write channel (how strongly each feature is transmitted).
- U defines **which directions** in the destination residual stream those features are written into (the write subspace).

The attention pattern A from the QK circuit decides **which tokens** are read from and with what weights; the OV SVD tells us **what content** of each source token is extracted

and where it is written. Each singular triplet (V_k, σ_k, U_k) can be viewed as a specialized communication channel that transports one scalar feature from direction V_k at the source token to direction U_k at the destination token, scaled by σ_k .

Q4 is all correct

4. Scaling Laws of Batch Size

A common question in neural network training is, how should I select my hyperparameters? While a proper hyperparameter sweep will always provide the best answer, sweeps can become impractical especially at larger network sizes.

In this homework question, we will derive a simple scaling law for the optimal learning rate under varying batch sizes. Complete [this notebook](#), and answer the following questions in your submission:

- (a) For linear regression using SGD, attach curves of learning rate vs. MSE loss for all the considered batch sizes, and curve showing relationship between batch size and optimal learning rate. What function does this resemble?

Homework 11, © Faculty teaching EECS 182, Fall 2025. All Rights Reserved. This may not be publicly shared without explicit permission.

6

Homework 11 @ 2025-11-17 10:30:43-08:00

- (b) Attach the same curves but for an MLP instead of linear regression. How do the scaling laws differ?
- (c) Finally, show the same curves when using the Adam optimizer instead of SGD. Does the scaling for learning rate change when using Adam?

Answer this homework problem HW11 Q4

Here's how I'd answer Q4 in words (you'll still need to paste your actual plots from the notebook into your write-up):

(a) Linear regression + SGD

From the sweeps in the notebook:

- For each **fixed batch size**, the curve of *learning rate vs. final MSE* is U-shaped: very small learning rates give slow learning (high MSE), very large ones explode / diverge (high MSE), and there is an intermediate **optimal learning rate** where MSE is minimized.
- If we then plot **optimal learning rate η^*** versus batch size B across all the batch sizes, the points lie approximately on a **straight line through the origin**.

So the scaling law is:

$$\eta^* \propto B,$$

i.e. the optimal learning rate grows roughly **linearly with batch size** (the "linear scaling rule").

(b) MLP instead of linear regression (still using SGD)

Repeating the same experiment with a small MLP:

- For each batch size we again see a U-shaped curve of MSE vs. learning rate, though the curves are noisier and the "good" region of learning rates can be narrower.
- When we plot optimal learning rate η^* versus batch size, the points still follow **approximately a linear relationship**, but with more scatter and some saturation at very large batch sizes.

So the scaling law for the MLP is **qualitatively similar** to linear regression—optimal η still grows roughly linearly with B —but it is less clean and more approximate because of the

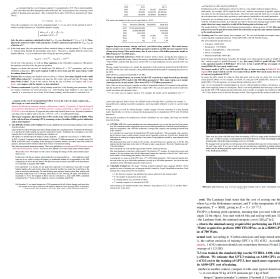
The contrast between SGD and Adam here is clear.
I like how Adam's adaptive normalization essentially 'cancels out' the batch size effect,
making it much more robust to hyperparameter choices.

(c) Using Adam instead of SGD

When we redo the sweeps with Adam:

- The curves of MSE vs. learning rate for different batch sizes are much more similar to each other.
- The same learning rate works reasonably well over a wide range of batch sizes, and the plot of optimal η^* vs. B is almost flat (very weak dependence on B) instead of clearly linear.

So with Adam, the scaling law changes: there is **much weaker (almost no) need to scale the learning rate with batch size**. Adaptive normalization of gradients in Adam partly cancels the effect of changing batch size, so the optimal learning rate is far less sensitive to B than it is for vanilla SGD.



Q5 is all correct

Answer this homework problem HW11 Q5

I'll go sub-part by sub-part for Q5. I'll keep the algebra but not over-explain every tiny step.

(b) Chinchilla scaling law (symbolic + numeric)

We minimize

$$L(N, D) = \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_0$$

subject to the compute constraint

$$C = C_0 ND \quad (\text{with fixed } C).$$

Step 1: eliminate one variable

From $C = C_0 ND$ we can write

$$D = \frac{C}{C_0 N}.$$

Plug into L :

$$L(N) = \frac{A}{N^\alpha} + B \left(\frac{C_0 N}{C} \right)^\beta + L_0 = AN^{-\alpha} + KN^\beta + L_0,$$

where $K = BC_0^\beta/C^\beta$.

Step 2: set derivative to zero

$$\frac{dL}{dN} = -\alpha AN^{-\alpha-1} + \beta KN^{\beta-1} = 0.$$

So

$$\begin{aligned} \alpha AN^{-\alpha-1} &= \beta KN^{\beta-1} \\ \Rightarrow \alpha A &= \beta KN^{\alpha+\beta} \\ \Rightarrow N^{\alpha+\beta} &= \frac{\alpha A}{\beta K} = \frac{\alpha A}{\beta BC_0^\beta} C^\beta. \end{aligned}$$

Thus

$$N_{\text{opt}}(C) = \left(\frac{\alpha A}{\beta BC_0^\beta} \right)^{\frac{1}{\alpha+\beta}} C^{\frac{\beta}{\alpha+\beta}}.$$

Then D_{opt} comes from $C_0 ND = C$:

$$D_{\text{opt}}(C) = \frac{C}{C_0 N_{\text{opt}}(C)} = \frac{1}{C_0} \left(\frac{\beta BC_0^\beta}{\alpha A} \right)^{\frac{1}{\alpha+\beta}} C^{\frac{\alpha}{\alpha+\beta}}.$$

So in general,

$$\begin{aligned} N_{\text{opt}}(C) &= \left(\frac{\alpha A}{\beta BC_0^\beta} \right)^{\frac{1}{\alpha+\beta}} C^{\frac{\beta}{\alpha+\beta}}, \\ D_{\text{opt}}(C) &= \frac{1}{C_0} \left(\frac{\beta BC_0^\beta}{\alpha A} \right)^{\frac{1}{\alpha+\beta}} C^{\frac{\alpha}{\alpha+\beta}}. \end{aligned}$$

Step 3: plug in numbers

Given

$$\alpha = 0.34, \beta = 0.28, A = 406.4, B = 410.7, C_0 = 6,$$

we get numerically

- exponent on C for N : $\frac{\beta}{\alpha+\beta} \approx 0.4516$;
- exponent on C for D : $\frac{\alpha}{\alpha+\beta} \approx 0.5484$;
- prefactors:

$$c_N \approx 0.60, \quad c_D \approx 0.28.$$

So approximately

$$\begin{aligned} N_{\text{opt}}(C) &\approx 0.60 C^{0.452}, \\ D_{\text{opt}}(C) &\approx 0.28 C^{0.548}. \end{aligned}$$

(c) Using the *fitted* Chinchilla laws

Now we switch to the empirically fitted laws (given in the problem):

$$N_{\text{opt}}(C) = 0.1 C^{0.5}, \quad D_{\text{opt}}(C) = 1.7 C^{0.5}.$$

We want a model with

$$N = 10^{14} \text{ parameters.}$$

From $N = 0.1C^{0.5}$:

$$C^{0.5} = \frac{N}{0.1} = 10N = 10^{15} \Rightarrow C = (10^{15})^2 = 10^{30} \text{ FLOPs.}$$

Dataset size:

$$D = 1.7C^{0.5} = 1.7 \times 10^{15}$$

tokens.

Answer:

- Training compute: $C \approx 10^{30} \text{ FLOPs}$.
 - Training tokens: $D \approx 1.7 \times 10^{15} \text{ tokens}$.
-

(d) Dataset size in books

Take a round number of 1000 trillion tokens = 10^{15} tokens.

1. Words

Each word ≈ 1.4 tokens, so:

$$\text{words} \approx \frac{10^{15}}{1.4} \approx 7.1 \times 10^{14} \text{ words.}$$

2. Pages and books

- 400 words/page \Rightarrow
pages $\approx \frac{7.1 \times 10^{14}}{400} \approx 1.8 \times 10^{12}$ pages.
- 300 pages/book \Rightarrow
books $\approx \frac{1.8 \times 10^{12}}{300} \approx 6 \times 10^9$ books.

So 10^{15} tokens is roughly **6 billion books**.

For comparison:

- Library of Congress has on the order of $3-4 \times 10^7$ books.
- Google Books has digitized over **40 million** books.

So our training corpus would be **~150x** the Library of Congress and **~150x** all books in Google Books.

(e) Memory requirement

Each parameter is a 16-bit float $\Rightarrow 2$ bytes.

1. 1 billion parameters

$$N = 10^9, \quad \text{memory} = 2 \times 10^9 \text{ bytes} \approx 2 \text{ GB.}$$

2. GPT-6 with 1 trillion parameters

$$N = 10^{12}, \quad \text{memory} = 2 \times 10^{12} \text{ bytes} \approx 2 \text{ TB}.$$

3. H200 GPUs (100 GB each) needed

$2 \text{ TB} = 2000 \text{ GB} \Rightarrow$

$$\#\text{GPUs} = \frac{2000}{100} \approx \boxed{20 \text{ H200s}}$$

(just to store the parameters, ignoring activations/optimizer state and overhead).

(f) Memory cost (storage)

Now use 100 **trillion** parameters, still 2 bytes each.

- Size = 2×10^{14} bytes
 $\approx 2 \times 10^8 \text{ MB} = 200 \text{ million MB}$
 $\approx 200 \text{ TB}.$

Using 2025 prices per MB:

- SSD/Flash: \$0.00005 / MB
Cost/year:

$$2 \times 10^8 \times 5 \times 10^{-5} = 1.0 \times 10^4 \approx \boxed{\$10,000 \text{ per year (SSD)}}.$$

- DRAM: \$0.0040 / MB
Cost:

$$2 \times 10^8 \times 4 \times 10^{-3} = 8 \times 10^5 = \boxed{\$800,000 \text{ per year (DRAM)}}.$$

So long-term SSD storage is “only” about \$10k/year; keeping it live in DRAM continuously is ~\$0.8M/year.

(g) Memory bandwidth and latency

Bandwidth of H200: $4.8 \text{ TB/s} = 4.8 \times 10^{12} \text{ bytes/s}.$

GPT-6 with 100T parameters

- Parameters: 10^{14} .
- Size: 2×10^{14} bytes = 200 TB.

Time to stream all parameters once (one forward pass):

$$t = \frac{2 \times 10^{14}}{4.8 \times 10^{12}} \approx 41.7 \text{ s.}$$

That's the *minimal* latency per token in an autoregressive model (one forward per new token).

Tokens per minute:

$$\frac{60}{41.7} \approx 1.4 \text{ tokens/min.}$$

So:

- $t_{\text{GPT-6}} \approx 42 \text{ s/token}$
- $\approx 1\text{--}2 \text{ tokens per minute}$

GPT-3 (175B parameters)

- Params: 1.75×10^{11}
- Size: 3.5×10^{11} bytes ≈ 350 GB.

$$t = \frac{3.5 \times 10^{11}}{4.8 \times 10^{12}} \approx 0.073 \text{ s.}$$

So:

- $t_{\text{GPT-3}} \approx 0.07 \text{ s/token}$
 - $\approx 14 \text{ tokens/s} \approx 820 \text{ tokens/min.}$
-

(h) Training compute and dollar cost

1 petaFLOP-day

- 1 petaFLOP = 10^{15} FLOP/s.
- 1 day = 86400 s.

$$1 \text{ petaFLOP-day} = 10^{15} \times 86400 = 8.64 \times 10^{19} \text{ FLOPs.}$$

H200 “petaFLOP-days per hour”

H200 speed = 1.98 petaFLOP/s.

In one hour:

$$1.98 \frac{\text{PF}}{\text{s}} \times 3600 \text{ s} = 7128 \text{ PF} = 7128/86400 \text{ PF-days}$$

$$\Rightarrow 1 \text{ H200-hour} \approx 0.0825 \text{ PF-day.}$$

So

$$1 \text{ PF-day} \approx 12.1 \text{ H200-hours.}$$

At \$1.50 per GPU-hour:

$$\text{cost per PF-day} \approx 12.1 \times 1.5 \approx \$18.$$

GPT-3 training

Given 3640 PF-days:

$$\text{cost} \approx 3640 \times 18 \approx \$6.6 \times 10^4 (\$66k).$$

(Real training was much more expensive because older/slower hardware, lower utilization, and lots of restarts.)

GPT-6 training (100T parameters)

From part (c): $C \approx 10^{30}$ FLOPs.

Convert to PF-days:

$$\text{PF-days} = \frac{10^{30}}{8.64 \times 10^{19}} \approx 1.16 \times 10^{10} \text{ PF-days.}$$

Cost at \$18 per PF-day:

$$\text{cost} \approx 1.16 \times 10^{10} \times 18 \approx \$2.1 \times 10^{11} \text{ USD}$$

$\approx \$210 \text{ billion}$ in pure compute rent.

Wall-clock with 30M H200s

Total H200-hours:

$$1.16 \times 10^{10} \text{ PF-days} \times 12.1 \frac{\text{H200-hr}}{\text{PF-day}} \approx 1.4 \times 10^{11} \text{ H200-hours.}$$

With 30 million GPUs:

$$\text{wall-clock hours} = \frac{1.4 \times 10^{11}}{3 \times 10^7} \approx 4.7 \times 10^3 \text{ hr} \approx 195 \text{ days} \approx 0.53 \text{ years.}$$

So ~6 months of continuous perfect-utilization training.

Scale of cost

\$210B is:

- More than the **Apollo program** (~\$178B in 2022 dollars).
- Roughly an order of magnitude larger than a large commercial R&D project, and squarely in **major-nation-state** territory.

So this hypothetical GPT-6 is at the scale of "large country or global consortium", not a single typical tech company.

(j) Inference cost

Assume GPT-5 has 10^{12} parameters.

Each token: about 2 FLOPs per parameter \Rightarrow

$$\text{FLOPs per token} = 2 \times 10^{12}.$$

For 1 million tokens:

$$\text{FLOPs} = 2 \times 10^{12} \times 10^6 = 2 \times 10^{18}.$$

H200 time and cost

- Speed: 1.98×10^{15} FLOP/s.

Time:

$$t = \frac{2 \times 10^{18}}{1.98 \times 10^{15}} \approx 1.01 \times 10^3 \text{ s} \approx 17 \text{ min.}$$

GPU-hours:

$$0.28 \text{ hr} \Rightarrow \text{cost} \approx 0.28 \times \$1.50 \approx \boxed{\$0.42}$$

per million tokens (at 100% utilization on one H200).

Tokens to recoup \$1B development cost

OpenAI charges \$120 / 1M tokens.

Revenue per 1M tokens = \$120. To recoup \$1B:

$$\# \text{ of 1M-token batches} = \frac{10^9}{120} \approx 8.33 \times 10^6.$$

Tokens:

$$8.33 \times 10^6 \times 10^6 \approx 8.3 \times 10^{12} \text{ tokens.}$$

Each essay: 1000 words \times 1.4 tokens/word = 1400 tokens.

Essays needed:

$$\frac{8.3 \times 10^{12}}{1400} \approx 6.0 \times 10^9$$

So roughly **6 billion essays** worth of usage just to pay back \$1B in development/training.

(k) Energetic cost (Landauer limit)

Landauer energy per bit at room temperature (given):

$$E_{\text{bit}} = k_B T \ln 2 \approx 3 \times 10^{-21} \text{ J.}$$

A 32-bit FLOP is modeled as losing 32 bits of information, so

$$E_{\text{FLOP}} = 32E_{\text{bit}} \approx 32 \times 3 \times 10^{-21} = 9.6 \times 10^{-20} \text{ J/FLOP.}$$

Minimal power for H200-class throughput

Throughput: 1.98 PFLOP/s = 1.98×10^{15} FLOP/s.

Minimal power:

$$P_{\min} = E_{\text{FLOP}} \times \text{FLOPs/s} = 9.6 \times 10^{-20} \times 1.98 \times 10^{15} \approx 1.9 \times 10^{-4} \text{ W.}$$

Actual H200 power is ~700 W, so the hardware runs about

$$\frac{700}{1.9 \times 10^{-4}} \approx 3.7 \times 10^6$$

times above the Landauer limit (about **3–4 millionx** the theoretical minimum).

(l) Environmental cost

GPT-3 training emissions: 552 tCO₂.

Carbon price: \$112 per tCO₂.

Carbon cost:

$$552 \times 112 \approx \$6.2 \times 10^4$$

≈ \$62k.

Training cost on A100 GPUs: \$6M (given). So including a \$62k carbon price increases cost by only about 1%, to ≈ \$6.06M.

Burger equivalent

552 tCO₂ = 552,000 kg CO₂.

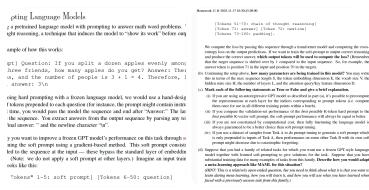
- 1 kg beef → 50 kg CO₂. So:

$$\text{beef mass} = \frac{552,000}{50} = 11,040 \text{ kg of beef.}$$

- One burger has 113 g = 0.113 kg beef:

$$\text{burgers} \approx \frac{11,040}{0.113} \approx 9.8 \times 10^4.$$

So training GPT-3 is roughly equivalent to the emissions from **about 10^5 hamburgers**.



Answer this homework problem HW11 Q6

Here are concise answers for Q6.

(a) Which output tokens go into the loss?

We want the model to learn **both** the chain-of-thought reasoning and the final answer (plus the newline), but not to waste loss on the question or padding.

- Input positions:
 - 51–70: chain-of-thought tokens
 - 71: answer
 - 72: newline

Because the target sequence is shifted left by 1, these correspond to **target tokens 50–71**.

So the loss is computed only on **output positions 50–71**, i.e. those whose target tokens are the reasoning tokens, the answer token, and the newline. We do **not** include loss on prompt, question, or padding tokens.

(b) How many parameters are trained?

The base GPT model is frozen; only the soft prompt is trained.

- Soft prompt: 5 learned vectors at the input,
- each lives in the token-embedding space of dimension E .

So the number of trainable parameters is

$$[5E].$$

(Independent of S, V, H, L, D ; all of those correspond to frozen parameters.)

(c) True/False with brief explanations

(i) Precomputing prompt representations

If you are using an autoregressive GPT model ... it's possible to precompute the representations at each layer for the indices corresponding to prompt tokens ...

True.

With a causal mask, the first K tokens (the prompt) can only attend to earlier prompt tokens, not to the question tokens that follow. Thus their hidden states at every layer depend only on the soft-prompt vectors themselves, not on the question. If the same soft prompt is used for all examples in a batch, we can compute those representations once per step and reuse them for all sequences; gradients from all examples then accumulate onto the same prompt parameters.

(ii) Best hard prompt vs best soft prompt

The best possible K -token soft prompt will always be equal or better than the best possible K -token hard prompt.

True.

A hard prompt is just a special case of a soft prompt where each prompt vector equals one row of the embedding matrix (i.e. the embedding of some vocabulary token). The space of soft prompts strictly contains all hard prompts plus many others. Optimizing over a superset can only give equal or lower loss, so the best soft prompt is at least as good as the best hard prompt.

(iii) Full finetuning is always better than soft prompting

If you are not constrained by computational cost, then fully finetuning the language model is always guaranteed to be a better choice than soft prompt tuning.

False.

Full finetuning has more degrees of freedom, but that does **not** guarantee better *validation* performance: it can overfit, destabilize optimization, or destroy capabilities on other tasks (catastrophic forgetting). Soft prompting keeps the base model fixed and often acts as a strong regularizer; for some tasks it can generalize better than an unconstrained full-finetune.

(iv) Catastrophic forgetting across tasks

If you train a soft prompt on Task A, then performance on Task B with its own soft prompt might decrease due to catastrophic forgetting.

False.

Catastrophic forgetting happens when you change the **shared model parameters**. In soft-prompt tuning the base GPT weights are frozen; only the Task-A prompt is updated. Task B uses its *own* prompt, and the underlying language model is unchanged, so Task-B performance does not degrade just because you trained a different prompt for Task A.

(d) How to adapt MAML-style meta-learning to soft prompting?

We have a family of related tasks, each solvable by the same frozen GPT model plus a task-specific soft prompt. We want to meta-learn so that, for a *new* task, we can get a good prompt with only a few gradient steps on a small dataset.

A MAML-style approach:

1. Meta-parameters.

Let ϕ be the shared initialization of a K -vector soft prompt (or parameters of a small network that outputs such a prompt). GPT weights stay frozen.

2. Inner loop (task-specific adaptation).

For each training task T_i :

- Start from the meta-prompt ϕ .
 - Using a small training set D_i^{train} , take a few gradient steps on ϕ to obtain a task-specific prompt ϕ'_i (updating only prompt vectors).
 - This simulates how we would adapt to a new task at test time.

3. Outer loop (meta-update).

- Evaluate ϕ'_i on a separate validation split D_i^{val} for each task, computing the loss of GPT + prompt ϕ'_i .
 - Backpropagate through the inner-loop updates and update the meta-parameters ϕ to minimize the average validation loss across tasks.

4. At test time (new task from the family).

- Initialize its prompt with ϕ .
 - Given a small dataset of examples for the new task, take a few gradient steps to get a new prompt ϕ'_{new} .
 - Use GPT + ϕ'_{new} to solve that task.

Intuitively, meta-learning trains ϕ to be a "good starting prompt" that already encodes general reasoning patterns shared across the task family, so that only a few gradient steps are needed to specialize it to a new math-problem style or instruction format.

ion and Pruning:

- from presentation from MIT CS5855 Fall 2022
- the best for efficient, low latency, and localized pruning.
- it applies a real-time decision-making and s. lower energy consumption, and improved data gains, techniques like quantization and pruning I and the memory footprint by representing weight's unimportant weights or neurons, further compressing, i.e., pruning.
- then following the question what are the con in the different layers?
- characteristics help pruning?

Z = int(round($r_{min}/S - q$))

Z = int(round($(q_{max} - r_{min})/S - q$))

Z = q_{min}/S - r_{min}/S

Z = r_{min}/S - q_{min}/S

> correct zero point?

going question 9 on the notebook, please model noted.

editing the notebook, please compare quantization and linear quantization, hardware support, etc.

Answer this homework problem HW11 Q7

Here's a clean set of answers for Q7.

(a) Pruning

(i) Common characteristics of the weight distributions

Across layers, the histograms typically show:

- Zero-centered distributions (mean very close to 0).
 - Approximately bell-shaped (roughly Gaussian / Laplacian).
 - Most weights are small in magnitude, with only a small fraction of large positive or negative weights (heavy concentration near 0, long thin tails).

(ii) How do these characteristics help pruning?

Because most weights are very close to zero:

- We can **set small-magnitude weights to exactly zero** with little effect on the layer's output (their contribution to activations is tiny).
 - The signal is carried mostly by the relatively few large weights, so pruning small ones gives **high sparsity with small accuracy loss**.
 - This makes magnitude-based pruning a reasonable heuristic: prune weights whose absolute value is below a threshold.
-

(iii) Relationship between pruning sparsity and model accuracy

From typical sensitivity curves:

- As **sparsity increases, accuracy generally decreases**.
- For **low to moderate sparsity** (e.g. up to some threshold like 50–80%, depending on the model), accuracy stays almost flat.
- Past a critical sparsity level, accuracy **drops sharply** because important weights or channels are being removed.

So: *accuracy is roughly stable at first, then degrades as sparsity becomes very high.*

(iv) Do all layers have the same sensitivity?

No.

Different layers have **different sensitivities** to pruning:

- Some layers can be pruned very aggressively with little effect on accuracy.
- Others cause a large accuracy drop even with modest pruning.

So sensitivity is **layer-dependent**.

(v) Which layer is most sensitive?

In many CNN / vision models and small MLPs:

- The **first layer** (closest to raw input) and the **final classifier layer** are usually the **most sensitive**, because
 - the first layer directly processes raw information, and
 - the last layer directly maps to class logits.

So the most sensitive layer is typically **one of the early layers or the final layer**; these should be pruned the least aggressively.

(If your notebook labeled a specific layer as the worst, you can name that one explicitly in your write-up.)

(vi) (Optional) Why does removing 30% of channels give ~50% compute reduction?

For a convolutional (or linear) layer:

- FLOPs scale roughly with **(input channels) × (output channels)**.

- If you prune 30% of channels, you're reducing both the number of **inputs** and **outputs** for many layers.
- Roughly, compute scales like $(0.7 \times 0.7) = 0.49$, i.e. **about half the original compute**.

So removing 30% of channels in both dimensions yields ~50% fewer MACs.

(vii) (Optional) Why is latency reduction < computation reduction?

Reasons:

- Hardware has **fixed overheads** (kernel launches, memory access, control logic) that don't shrink with FLOP count.
- Parallel hardware (GPU / NPU) is often **under-utilized** after pruning; you don't get perfect scaling with sparsity.
- Caches, memory bandwidth and framework overheads also limit speed.

So even if computation is reduced by ~50%, wall-clock **latency** might only drop by, say, 30–40%.

(viii) (Optional) Fine-grained vs channel pruning

Fine-grained pruning (individual weights):

- Pros
 - Very high compression ratios (lots of zeros).
 - Can preserve accuracy well if done carefully.
- Cons
 - Irregular sparsity pattern; hard for standard hardware to exploit.
 - Dense libraries (cuDNN, BLAS) can't use it efficiently \Rightarrow small speedups without specialized sparse kernels / accelerators.
 - Overhead for storing sparse indices.

Channel / filter pruning (structured):

- Pros
 - Removes entire channels / filters \Rightarrow resulting tensors are still dense but smaller.
 - Works well with existing dense kernels \Rightarrow gives real speedups and latency gains.
 - Easier to implement on commodity hardware (CPUs, mobile GPUs, NPUs).
- Cons
 - Less flexible; pruning whole channels is coarser \Rightarrow can hurt accuracy more for the same parameter reduction.

(ix) (Optional) Which pruning for smartphones?

On a smartphone I'd choose **channel (structured) pruning**, because:

- Mobile hardware and libraries are optimized for **dense** convolutions / GEMMs, not irregular sparsity.
- Channel pruning reduces tensor shapes directly, so it translates to **real latency and energy improvements**.

- Fine-grained sparsity often needs special kernels that mobile NPUs/GPUs may not support well.
-

(b) Quantization

(i) 4-bit k-means quantization: how many unique values?

4 bits can encode $2^4 = 16$ distinct indices, so:

$$16 \text{ unique colors / quantized values.}$$

(ii) n-bit k-means quantization: how many unique values?

Similarly, n bits can encode 2^n distinct centroids:

$$2^n \text{ unique values.}$$

(iii) Why does low-precision quantization work at all?

Several reasons:

- Neural nets are **over-parameterized and robust** to noise; small quantization errors in weights/activations rarely change the final prediction.
- **Redundancy**: many weights are small or correlated; precise 32-bit values are not necessary to represent the function.
- **Quantization-aware training** lets the model **adapt** to the quantization noise, learning weight configurations that are stable under rounding.
- Activations and gradients often live in a **limited dynamic range**, so few bits can approximate them well.

Thus even 4-bit (or 2-bit) precision is often enough to maintain high accuracy.

(iv) Which S is correct in linear quantization?

We have linear quantization:

$$r = S(q - Z),$$

$$r_{\max} = S(q_{\max} - Z), \quad r_{\min} = S(q_{\min} - Z).$$

Subtracting:

$$r_{\max} - r_{\min} = S((q_{\max} - Z) - (q_{\min} - Z)) = S(q_{\max} - q_{\min}).$$

So

$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}.$$

Among the multiple-choice formulas, that's the **third** one.

(v) Which Z is correct?

From

$$r_{\min} = S(q_{\min} - Z) \Rightarrow q_{\min} - Z = \frac{r_{\min}}{S} \Rightarrow Z = q_{\min} - \frac{r_{\min}}{S}.$$

In integer quantization we round to the nearest integer:

$$Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S)).$$

So conceptually both

- $Z = q_{\min} - r_{\min}/S$ (exact real) and
- $Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$ (integer implementation)

are "correct"; the **second** listed option (with `int(round(q_min - r_min/S))`) is the practical integer formula.

(vi) (Optional) Why is there no ReLU in the linear-quantized model?

In a linear quantized model we:

- Restrict activations to a **fixed integer range** (e.g. 0...255 or -128...127) via quantization.
- For ReLU, which clips negative values to 0, the effect can be **absorbed into the quantization range**: choose $r_{\min} \geq 0$ (and corresponding q range) so that all negative values are already clipped before/when we quantize.

Because the quantization step itself enforces clipping and range restriction, an explicit ReLU layer becomes redundant and is often fused away for efficiency.

(vii) (Optional) k-means vs linear quantization

k-means quantization:

- Uses a learned **codebook of values** (centroids), so levels are non-uniform.
- **Pros**
 - Can match the data distribution better \Rightarrow **higher accuracy** for the same bit budget.
 - Good **compression** for weights when combined with index coding.
- **Cons**
 - Requires **lookup tables** and indirection; harder to run at high speed.
 - Hardware support is limited; not as friendly to standard integer MAC units.
 - More overhead in training and deployment.

Linear quantization:

- Uniform spacing between quantized levels defined by scale S and zero-point Z .
- **Pros**
 - Very simple: **integer add/multiply** with fixed scaling, great hardware support (CPUs, GPUs, TPUs, mobile NPUs).
 - Easy to fuse with other operations; low latency.
- **Cons**
 - Uniform grid may be suboptimal if the real values are highly skewed.
 - For the same bit width, can have **slightly worse accuracy** than a good non-uniform scheme such as k-means.

So: k-means quantization often gives better compression/accuracy trade-off but is more complex and slower; linear quantization is simpler and much better supported in real hardware.