



+ New Chat

⌘ K

OK Computer

Researcher

Slides

Chat History

Newton-Schulz Runtime ...

幻灯片照片排版

All Chats

Mobile App

Michael ... Upgrade



Overall, Kimi k2 almost one-shotted all the problems correctly in this HW set, except for a few subparts which were dependent on image recognition or misread hallucinations

1. Newton-Schulz Runtime

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

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

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

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

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

Please solve this problem thoroughly



Optimizing Polynomial Matrix Multiplication by Exploiting Smaller Dimension >

Here is a thorough solution to the Newton-Schulz runtime problem:

(a) Runtime of a Single Iteration

The degree-3 odd polynomial for the Newton-Schulz update is:

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

The naïve implementation computes this as:

1. **Compute WW^T** : Multiply $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ by $W^T \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$

- Runtime: $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{out}} = cd_{\text{out}}^2 d_{\text{in}}$

2. **Compute $(WW^T)W$** : Multiply the result $(WW^T) \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$ by $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$

- Runtime: $c \cdot d_{\text{out}} \cdot d_{\text{out}} \cdot d_{\text{in}} = cd_{\text{out}}^2 d_{\text{in}}$

3. **Scaling and subtraction**: Compute $\frac{3}{2}W - \frac{1}{2}(WW^TW)$ (element-wise operations, negligible cost)

Total runtime per iteration: $2cd_{\text{out}}^2 d_{\text{in}}$

(b) Accelerated Computation when $d_{\text{out}} > d_{\text{in}}$

When $d_{\text{out}} > d_{\text{in}}$, we can compute the **Gram matrix** $W^T W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ instead, which is much

smaller.

Method:

Rewrite the polynomial by factoring W on the right:

$$p(W) = W \left(\frac{3}{2}I - \frac{1}{2}W^T W \right)$$

Steps:

1. **Compute $W^T W$** : Multiply $W^T \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ by $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$

- Runtime: $c \cdot d_{\text{in}} \cdot d_{\text{out}} \cdot d_{\text{in}} = cd_{\text{in}}^2 d_{\text{out}}$

2. **Compute $\frac{3}{2}I - \frac{1}{2}W^T W$** : Element-wise scaling and subtraction (negligible cost)

3. **Compute $W \times \text{result}$** : Multiply $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ by the matrix from step 2 $\in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$

- Runtime: $c \cdot d_{\text{out}} \cdot d_{\text{in}} \cdot d_{\text{in}} = cd_{\text{out}} d_{\text{in}}^2$

Total updated runtime per iteration: $cd_{\text{in}}^2 d_{\text{out}} + cd_{\text{out}} d_{\text{in}}^2 = 2cd_{\text{out}} d_{\text{in}}^2$

Correctly one-shotted
the first two parts of
question 1.

Speedup Analysis:

Since $d_{\text{out}} > d_{\text{in}}$, we have $d_{\text{in}}^2 d_{\text{out}} < d_{\text{out}}^2 d_{\text{in}}$. The new approach yields a **speedup factor of**

$\frac{d_{\text{out}}}{d_{\text{in}}}$ over the naïve implementation. This is significant when the output dimensionality is much larger than the input dimensionality, as is common in many neural network layers.

This model only extracts text from images, switch to K1.5 for full visual understanding.



2. MuP at the Unit Scale

By now, we have seen how the maximal-update parameterization allows us to properly scale updates to weights, based on the shapes of the dense layers. In last week's homework, we hinted at how this procedure can apply either as a layer-wise learning rate, or as a direct adjustment of the forward pass graph.

This time, we will consider how these same principles can be used in the context of low-precision training. GPUs that support low-precision training tend to exhibit a linear speedup in computation time as precision is lowered (i.e. doing a matrix multiplication in fp16 is twice as fast as in fp32). For this reason, we would like to design training algorithms that can remain numerically stable even at precisions as low as 8 bits.

- You are designing a neural network training algorithm that will train with fp8 parameters (assume activations will be calculated in full precision and we don't need to worry about them). **Why would it make sense for parameters to be initialized from $N(0, 1)$ as opposed to e.g. Xavier initialization?** As a starting point, note that fp8 can only represent 255 possible values, and you can view these values at: https://microsoft.github.io/fp8_tables.php.

- (b) We have initialized our parameters from $N(0, 1)$. However, we have now lost the desirable properties of Xavier initialization, and our activations are exploding as they propagate deeper into the network. To solve this, we can assign a constant (float) scalar to be multiplied with the activations:

$$y = cWx.$$

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

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

Please solve this problem thoroughly



Here is a thorough solution to the MuP at the Unit Scale problem:

(a) Why $N(0,1)$ Initialization for fp8?

With fp8's extremely limited representation (only 255 distinct values), **Xavier initialization would waste most of the representable range**. Xavier typically initializes with variance $\approx 1/d_{in}$, producing values densely clustered near zero. In fp8, this causes:

- **Severe quantization:** Most values would quantize to zero or the same few subnormal values
- **Lost precision:** The dynamic range of fp8 (± 448 for E4M3, ± 57344 for E5M2) would be underutilized
- **Vanishing gradients:** Quantized near-zero weights cannot learn effectively

$N(0,1)$ initialization uses the full fp8 range, ensuring:

- Each representable value carries meaningful information
- Gradient updates have sufficient magnitude to escape quantization bins
- The network can exploit fp8's dynamic range maximally

The tradeoff is uncontrolled variance growth, which we address via explicit scaling.

(b) The Scaling Constant c

We want $\text{Var}(y) = \text{Var}(x)$ to recover Xavier's variance-preserving property.

Given:

- Forward pass: $y = cWx$
- $W \in \mathbb{R}^{(d_{out} \times d_{in})}$ with $W_{ij} \sim N(0,1)$, so $\text{Var}(W_{ij}) = 1$
- $x \in \mathbb{R}^{d_{in}}$ with $\text{Var}(x_j) = 1$ (unit variance)

Variance propagation rule:

$$\text{Var}(y_i) = \sum_j \text{Var}(c \cdot W_{ij} \cdot x_j) = c^2 \cdot d_{in} \cdot \text{Var}(W_{ij}) \cdot \text{Var}(x_j) = c^2 \cdot d_{in}$$

To ensure $\text{Var}(y_i) = \text{Var}(x_j) = 1$:

$$c^2 \cdot d_{in} = 1$$

Therefore: $c = 1/\sqrt{d_{in}}$

(c) Maximum Spectral Norm of the Update

We want to bound the **update to weights ΔW** such that the output perturbation Δy remains controlled.

Given:

- Perturbed forward pass: $y + \Delta y = c(W + \Delta W)x$
- Therefore: $\Delta y = c\Delta Wx$
- Constraint: $\|x\|_{\text{RMS}} = 1$
- Goal: $\|\Delta y\|_{\text{RMS}} \leq 1$

Analysis:

1. RMS-L2 relationship: $\|v\|_{\text{RMS}} = \|v\|_2 / \sqrt{\dim(v)}$
2. Spectral norm property: $\|Av\|_2 \leq \|A\|_2 \cdot \|v\|_2$

Combining these:

$$\begin{aligned}\|\Delta y\|_{\text{RMS}} &= \|c\Delta Wx\|_2 / \sqrt{d_{\text{out}}} \\ &\leq c \cdot \|\Delta W\|_2 \cdot \|x\|_2 / \sqrt{d_{\text{out}}}\end{aligned}$$

With $\|x\|_{\text{RMS}} = 1$, we have $\|x\|_2 = \sqrt{d_{\text{in}}}$. From part (b), $c = 1/\sqrt{d_{\text{in}}}$:

$$\begin{aligned}\|\Delta y\|_{\text{RMS}} &\leq (1/\sqrt{d_{\text{in}}}) \cdot \|\Delta W\|_2 \cdot \sqrt{d_{\text{in}}} / \sqrt{d_{\text{out}}} \\ &= \|\Delta W\|_2 / \sqrt{d_{\text{out}}}\end{aligned}$$

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

$$\|\Delta W\|_2 / \sqrt{d_{\text{out}}} \leq 1 \Rightarrow \|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$$

Answer: The maximum spectral norm of the weight update should be $\sqrt{d_{\text{out}}}$.

This ensures that when the update is applied, the output change remains at unit scale, maintaining stable training in the low-precision regime.

(d) Let us consider the case of SignSGD. Assume that our minibatch is of size 1. You saw in discussion that $\text{sign}(\nabla_W L)$ is a rank-1 matrix. What learning rate α is required to ensure that the overall update of $\alpha \cdot \text{sign}(\nabla_W L)$ satisfies the spectral norm constraint from part (c)? Ensure your answer works on rectangular weight matrices.

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

$$U, \Sigma, V^T = \text{SVD}(\nabla_W L) \quad (1)$$

$$\Delta W = \alpha \cdot U V^T. \quad (2)$$

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

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

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

please continue it thoroughly

💡 Unit-scaled N(0,1) initialization stabilizes rank-1 weight matrices >

Here is the thorough continuation for parts (d)-(g) of the MuP at the Unit Scale problem:

(d) SignSGD with Spectral Norm Constraint

For SignSGD, the weight update is $\Delta W = \alpha \cdot \text{sign}(\nabla_W L)$. From part (c), we require $\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$.

Given that $\text{sign}(\nabla_W L)$ is rank-1, its spectral norm equals its Frobenius norm. In the worst case where all entries are non-zero, each element is ± 1 , so:

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

To satisfy the constraint:

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

$$\alpha \leq 1/\sqrt{d_{\text{in}}}$$

This learning rate ensures the sign-based update respects the spectral norm bound for rectangular matrices.

(e) Muon-Style Orthogonalized Update

The orthogonalized update is $\Delta W = \alpha \cdot U V^T$ where $\nabla_W L = U \Sigma V^T$ (compact SVD).

Key property: U and V have orthonormal columns, so UV^T has spectral norm 1:

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

Therefore:

Correctly one-shot the second question and arguably for part e) provides a less tight constraint as possible. Perhaps it did not coincide with the same logic as part d)

$$\|\Delta W\|_2 = \alpha \cdot \|UV^T\|_2 = \alpha$$

From part (c), we need $\|\Delta W\|_2 \leq \sqrt{d_{\text{out}}}$:

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

Notably, this learning rate is independent of the input dimension d_{in} , as Muon automatically normalizes the gradient direction.

(f) Ignoring Scale of Intermediate Gradients

During backpropagation: $\nabla_{\{x_n\}} L = c_n W_n^T \nabla_{\{x_{n+1}\}} L$ where $c_n = 1/\sqrt{d_{\text{in},n}}$.

Yes, the scale can be ignored if we apply scale-invariant operations to the backpropagated gradients:

1. **Sign operation**: Taking $\text{sign}(\nabla_{\{x_{n+1}\}} L)$ removes magnitude information
2. **Orthogonalization**: Applying Muon-style SVD to gradient matrices before backpropagation
3. **Normalization**: RMS/Layer norm on gradients before multiplication

These methods only depend on gradient direction, making the absolute scale of $\nabla_{\{x_{n+1}\}} L$ irrelevant to computing $\nabla_{\{x_n\}} L$, analogous to how SignGD/Adam/Muon treat parameter updates.

(g) Gradient Explosion/Vanishing and Stabilization

Effect Without Adjustments:

With unit variance parameters ($W_{ij} \sim N(0,1)$) and rank-1 structure:

- $\|W_n\|_2 = \sqrt{(d_{\text{out},n} \cdot d_{\text{in},n})}$
- Backward scale factor: $c_n \|W_n\|_2 = (1/\sqrt{d_{\text{in},n}}) \cdot \sqrt{(d_{\text{out},n} \cdot d_{\text{in},n})} = \sqrt{d_{\text{out},n}}$

This causes **gradient explosion** —each layer magnifies gradients by $\sqrt{d_{\text{out},n}}$. Over multiple layers, gradients grow exponentially.

Stabilization Constants:

Multiply each layer's backpropagated gradient by:

$$\gamma_n = 1/\sqrt{d_{\text{out},n}}$$

This creates symmetric scaling with the forward pass ($1/\sqrt{d_{\text{in},n}}$ forward, $1/\sqrt{d_{\text{out},n}}$ backward), ensuring stable gradient flow in both directions. The product of scaling factors across layers remains $O(1)$, preventing exponential explosion/vanishing.

Note: If "unit scaled" instead meant $\|W_n\|_F = 1$, then $\|W_n\|_2 = 1$ and gradients would vanish by $1/\sqrt{d_{in,n}}$, requiring stabilization constant $\sqrt{d_{in,n}}$. However, given the problem's context of $N(0,1)$ initialization, the first interpretation (explosion with $1/\sqrt{d_{out,n}}$ correction) is more consistent.

This model only extracts text from images, switch to K1.5 for full visual understanding.



Using Convolution as Finite Impulse Response Filter
In time signal, the output of linear time invariant system is defined as:

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

part signal, h , is impulse response (also referred to as the filter). Please note this is to 'flip and drag'. But for neural networks, we simply implement the filtering operation. Interestingly, in CNN there we have filter weights are initialized and updated. Even though you implement ended up with getting the flipped kernel. In this question, we will follow the filter rectangular signal with the length of L (sometimes also called the "rect" or "boxcar" signals). This signal is defined as:

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

is plot for $L = 7$, with time indices shown from -2 to 8 (so some implicit zeros)

Figure 1: The rectangular signal with the length of L

(a) The impulse response is defined as

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

Compute and plot the convolution of $x(n)$ and $h(n)$. For illustrative purposes, you plot should not be a 4D surface (e.g., $x(n) \otimes h(n)$).
(b) Now, consider the case by $M = 1$, i.e., $x(n) = x(n - M)$. Early for $M = 2$. Then, compute $y[n] = x(n) * h(n)$. What property of the convolution can you find?
Now, let's consider 2D case. The example of 2D signal is given in the image. The operation of 2D convolution is defined as

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

where x is input signal, h is 2D filter and y is the output signal.
(c) 2D matrices, x and h are given like below:

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

$$h = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \quad (2)$$

Then, calculate y . Assume that there is no padding and stride is 1.
(d) Now let's consider stride and padding. Estimate y for following cases.

please solve this next! make sure to pay attention to the figure



Let's solve the problem step by step, following the instructions and using the provided figures and definitions.

Part (a): Compute and Plot the Convolution of $x(n)$ and $h(n)$

Given:

- $x(n)$ is a rectangular signal of length $L = 7$, defined as:

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

- $h(n)$ is defined as:

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

Convolution Calculation:

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

Since $x[n]$ and $h[n]$ are non-zero only for $n = 0, 1, 2, \dots, 6$ and $n = 0, 1, 2, \dots$

respectively, the sum is effectively:

$$y[n] = \sum_{i=0}^6 x[i] \cdot h[n-i]$$

Calculating $y[n]$ for $n = -6$ to $n = 12$ (as suggested for illustration purposes):

- For $n < 0$ and $n > 6$, $y[n] = 0$ because $h[n-i]$ will be zero.
- For $n = 0$ to $n = 6$, calculate each $y[n]$ by summing the products of $x[i]$ and $h[n-i]$ for $i = 0$ to 6 .

Example Calculations:

- $y[0] = h[0] = (\frac{1}{2})^0 = 1$
- $y[1] = h[0] + h[1] = 1 + \frac{1}{2} = 1.5$
- $y[2] = h[0] + h[1] + h[2] = 1 + \frac{1}{2} + \frac{1}{4} = 1.75$
- Continue this pattern up to $y[6]$.

Plotting:

The plot should show $y[n]$ starting from 1, increasing as more terms are added, and then decreasing as terms start to become zero again.

Part (b): Shift $x(n)$ by N and Compute $y_2(n) = h(n) * x_2(n)$

Given $N = 5$, $x_2(n) = x(n - N)$.

Convolution Calculation:

$$y_2[n] = h[n] * x_2[n] = \sum_{i=-\infty}^{\infty} h[i] \cdot x_2[n-i]$$

$$y_2[n] = \sum_{i=0}^6 h[i] \cdot x_2[n-i]$$

Since $x_2[n] = x[n - 5]$, the non-zero values of $x_2[n]$ are for $n = 5, 6, 7$.

Example Calculations:

- $y_2[5] = h[0] = 1$
- $y_2[6] = h[0] + h[1] = 1.5$

- $y_2[7] = h[0] + h[1] + h[2] = 1.75$

Property of Convolution:

The property that can be observed is the **commutative property** of convolution, i.e., $h[n] * x[n] = x[n] * h[n]$. This is because convolution is commutative, and the shift operation does not affect this property.

Kimi failed here for part c, it could not extract the matrix numbers from the image provided in the prompt. This feels more like an ocr/image recognition failure rather than a reasoning capability.

Part (c): 2D Convolution

Given matrices x and h , compute the 2D convolution $y[m, n]$ without padding and stride 1.

Convolution Calculation:

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

Since x and h are non-zero only within their defined ranges, the sum is effectively over the overlapping regions.

Example Calculations:

- For each m, n within the bounds of x and h , compute the sum of products of corresponding elements.

Plotting:

The resulting matrix y should be plotted, showing the effect of the convolution operation.

Part (d): Striding and Padding

i. Stride, pad = 1, 1

- **Padding:** Add a border of zeros around x to increase its size by 2 units on each side.
- **Stride:** Move h across x with a step size of 1.

ii. Stride, pad = 2, 1

- **Padding:** Add a border of zeros around x to increase its size by 1 unit on each side.
- **Stride:** Move h across x with a step size of 2.

Convolution Calculation:

For each case, compute $y[m, n]$ using the same formula, adjusting the bounds of the sum

Correct Logic. Kimi correctly described the effects (output size reduction) and the arithmetic, though it could not produce the specific numerical answer for the matrices it missed in 3(c)

based on the stride and padding.

Plotting:

Plot the resulting matrices y for each case to visualize the effect of different strides and paddings.

This step-by-step solution should help you understand and compute the convolution operations as described in the problem.



Homework 4 - 0 2025-09-27 00:19:17Z

Building blocks of CNNs: Let's assume that input feature shape is $B' \times H \times W \times C'$, where B' is the batch size, H is the height and C' is the number of channels of input feature.

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

(b) A max pooling layer has 2 architectural hyperparameters: the stride step $K=3$ and the "filter size" $C=2$. What is the output shape of this max pooling layer?

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

(d) Consider a downsampling layer (e.g. pooling layer and added convolution layer). In this problem, we investigate prior and prior of downsampling layers. This layer reduces the output feature resolution, and the output feature resolution becomes a multiple of the input feature resolution. Therefore, when we design CNNs, we usually increase the batch size to compensate this loss. For example, if we apply a max pooling layer with a kernel size of 2 and a stride of 2, we increase the output feature size by a factor of 2. If we apply this max pooling layer, how much does the receptive field increase? Explain the idea of downsampling the output feature resolution with the perspective of reducing the size of computation.

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

- CONV3-10: denotes a convolutional layer with $K=3$ different filters, each of size $3 \times 3 \times C$, where C is the depth (i.e. number of channels) of the activation from the previous layer. Padding is 1, and stride is 1.
- POOL2: denotes a 2×2 max-pooling layer with stride 2 (pad 0).
- FLATTEN: just takes whatever shape input tensor into a one-dimensional array with the same values in it.
- FC-3: denotes a fully connected layer with 3 output neurons.

Note: All CONV3-10 and FC-3 layers have biases as well as weights. Do not forget the biases when calculating the output.

Note: We are going to use this network to do inference on a single input. W is the missing entries in this table or the size of the activations at each layer, and the number of parameters at each layer. You can choose your answer as a computation (e.g. $128 \times 128 \times 3$) in the style of the already filled-in entries of the table.

please solve this as well!



Let's solve the problem step by step, filling in the missing entries in the table for the size of the activations at each layer and the number of parameters at each layer.

Given Information:

- Input: $28 \times 28 \times 1$
- CONV3-10: $3 \times 3 \times 1 \times 10$, stride 1, padding 0
- POOL2: $3 \times 3 \times 10 \times 10$, stride 2, padding 0
- FC-3: fully connected layer with 3 output neurons

Calculations:

1. CONV3-10:

- **Number of Parameters:** $3 \times 3 \times 1 \times 10 + 10$ (10 filters, each with 9 parameters plus one bias)

$$= 90 + 10 = 100$$

- **Dimension of Activations:** $(28 - 3 + 0) \times (28 - 3 + 0) \times 10 = 26 \times 26 \times 10$

2. POOL2:

- **Number of Parameters:** 0 (no parameters in pooling layer)
- **Dimension of Activations:** $\frac{26}{2} \times \frac{26}{2} \times 10 = 13 \times 13 \times 10$

3. POOL2 (second occurrence):

- **Number of Parameters:** 0 (no parameters in pooling layer)
- **Dimension of Activations:** $\frac{13}{2} \times \frac{13}{2} \times 10 = 6.5 \times 6.5 \times 10$ (This should be rounded or adjusted based on the specific implementation, typically to $6 \times 6 \times 10$)

4. FLATTEN:

- **Number of Parameters:** 0 (no parameters in flattening layer)
- **Dimension of Activations:** $6 \times 6 \times 10 = 360$ (flattening to a 1D vector)

5. FC-3:

- **Number of Parameters:** $360 \times 3 + 3$ (3 outputs, each connected to 360 inputs plus one bias)

$$= 1080 + 3 = 1083$$

- **Dimension of Activations:** 3

Final Table:

Table		<input type="checkbox"/> Copy
Layer	Number of Parameters	Dimension of Activations
Input	0	$28 \times 28 \times 1$
CONV3-10	100	$26 \times 26 \times 10$

POOL2	0	$13 \times 13 \times 10$
POOL2	0	$6 \times 6 \times 10$
FLATTEN	0	360
FC-3	1083	3

This table now includes all the necessary calculations for the size of the activations at each layer and the number of parameters at each layer.



Diagram showing the architecture: CONV2-3 → ReLU → CONV2-3 → ReLU → GAP → FC-3

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

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

You are told the behavior for an input image with a horizontal edge, w_1 , and an image with a vertical edge, w_2 :

$$w_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Suppose we know that the GAP output features when the w_1 and w_2 are:

$$g_1 = f(w_1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad g_2 = f(w_2) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Use what you know about the invariance/independence of convolutional nets to complete the g_3 , corresponding to the following w_3 image:

$$w_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$g_3 = f(w_3) = \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix}$$

solve this next part!



To solve this part of the problem, we need to understand the behavior of convolutional layers with circular padding and how they respond to different input images. We'll also need to consider the properties of the Global Average Pooling (GAP) layer.

Understanding Circular Padding

Circular padding treats the image as if it were on a torus, where the edges wrap around to the opposite side. This means that when the convolution kernel moves off one edge of the image, it re-enters from the opposite side.

Behavior of the Network

Given the architecture:

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

- Each CONV2-3 layer has a kernel size of 3×3 , stride of 1, and padding of 1 (circular).

- The ReLU activation function is applied after each convolution.
- The GAP layer computes the average of each channel across the entire feature map.
- The FC-3 layer is a fully connected layer with 3 outputs.

Analyzing the Input Images

We have three input images:

1. \mathbf{x}_1 : Horizontal edge

2. \mathbf{x}_2 : Vertical edge

3. \mathbf{x}_3 : Diagonal edge

Given the GAP outputs for \mathbf{x}_1 and \mathbf{x}_2 :

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

Invariance and Equivariance

Convolutional networks are equivariant to translations, meaning that a shift in the input image results in a corresponding shift in the output feature map. With circular padding, the network exhibits a form of rotational equivariance because the wrapping around the edges simulates rotation.

Calculating \mathbf{g}_3

Since \mathbf{x}_3 is a diagonal edge, it can be seen as a rotation of \mathbf{x}_1 or \mathbf{x}_2 by 45 degrees. Due to the rotational equivariance induced by circular padding, the output should be similar to \mathbf{g}_1 or \mathbf{g}_2 , but the exact channel that is activated might differ depending on the specific weights of the convolutional filters.

Given the symmetry and the fact that \mathbf{x}_3 is a diagonal, it is likely that the network will respond similarly to how it responds to horizontal or vertical edges, but the activation might be spread across different channels due to the diagonal nature of the edge.

Assuming the network has learned to respond similarly to all orientations:

- \mathbf{g}_3 might be a combination of \mathbf{g}_1 and \mathbf{g}_2 , or it might activate a different channel if the network has learned to distinguish diagonal edges.

Without specific weights, we can hypothesize:

$$\mathbf{g}_3 = \begin{bmatrix} 0.4 \\ 0.4 \\ 0 \end{bmatrix}$$

This assumes that the diagonal edge activates both the horizontal and vertical channels equally due to the symmetry of the problem.

Conclusion

The exact values of \mathbf{g}_3 depend on the specific weights of the network, but due to the rotational equivariance and the symmetry of the problem, it is reasonable to expect a similar pattern to \mathbf{g}_1 and \mathbf{g}_2 .

✓

The weight of a CNN is a weighted average of the images in the dataset. This understanding is crucial in answering a commonly asked question: does a CNN memorize images during the training process? Additionally, we will analyze the impact of spatial receptive fields in convolutional layers. Secondly, we are interested in the effect of spatial receptive fields on the gradient flow during backpropagation. By accomplishing these objectives, we will enhance our knowledge of CNNs and their functioning.

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

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

weight matrix $\mathbf{w} \in \mathbb{R}^{m \times n}$:

$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{bmatrix}$ (2)

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

$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \dots & y_{1,m} \\ y_{2,1} & y_{2,2} & \dots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \dots & y_{m,n} \end{bmatrix}$ (3)

For simplicity, we assume the number of the input channel of \mathbf{X} is 1 and the number of the output channel of simple \mathbf{Y} is 1 and the convolutional layer has no padding and a stride of 1. Thus for all i, j :

$y_{i,j} = \sum_{k=1}^n \sum_{l=1}^m w_{i,k} x_{k,l}$ (4)

$\mathbf{N} = \mathbf{X} * \mathbf{w}$ (5)

where \ast refers to the convolution operation. For simplicity, we consider the bias term in this operation.

Suppose the first row is \mathbf{C} , and the variance product is $\mathbf{P} \in \mathbb{R}^{m \times m}$:

$\mathbf{C} = \begin{bmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,m} \\ c_{2,1} & c_{2,2} & \dots & c_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \dots & c_{m,n} \end{bmatrix}$ (6)

where $c_{i,j}$ denotes $\frac{\partial y_{i,j}}{\partial x_{1,1}}$. Note, denotes the height after one MAX step with a batch of a single image.

(a) Define the product in the weight matrix $\mathbf{w} \in \mathbb{R}^{m \times n}$:

$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{bmatrix}$ (7)

where $w_{i,j}$ denotes $\frac{\partial y_{i,j}}{\partial w_{1,1}}$. Note, denotes the height after one MAX step with a batch of a single image.

(b) The relevance of the part is to investigate the effect of spatial weight during convolution layers in the behavior of gradient vector with respect to gradient vector in input image.

For simplicity of analysis, we assume $x_{1,1}, \dots, x_{1,n}$ are the same column variables, where $x_{1,1}, \dots, x_{1,n}$ are the same and variance of $x_{1,1} = \frac{n}{n-1}$ for each $i > 1$. Definition of $x_{1,1}, x_{1,2}, \dots, x_{1,n}$. What is the maximum product rate of the gradient vector with respect to the gradient on $x_{1,1}$ with respect to the length and width of the image?

Now, we can prove that $w_{1,1}$ is the most relevant because we can do the following:

(i) If you cannot assume that $x_{1,1}, \dots, x_{1,n}$ follow normal distribution in your derivation or proof, then you cannot assume that $x_{1,1}, \dots, x_{1,n}$ follow normal distribution in your derivation or proof.

(ii) If a network with only 2d max pooling layers (no convolution layers), an architecture, what will

Simple case first, where $\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix}$, \mathbf{F} is the filter size and arbitrary max positions. part, discuss the advantages of max pooling to finish the question "Inductive Bias of Max Pooling".

now solve these last problems!

✓

Compute Variance of Convolution Gradients and Backprop Through Pooling >

Here is the complete solution to the remaining problems (3, 4, and 5):

Problem 3: Understanding Convolution as Finite Impulse Response Filter

Part (a): Convolution $y[n] = x[n] * h[n]$

Given:

- Rectangular signal: $x[n] = 1$ for $n = 0..6$, 0 otherwise
- Impulse response: $h[n] = (1/2)^n$ for $n \geq 0$, 0 otherwise

• Impulse response: $h[n] = (1/2)^n$ for $n \geq 0$, 0 otherwise

Computation:

The convolution $y[n] = \sum_{i=0}^6 h[n-i]$ is piecewise:

- $n < 0$: $y[n] = 0$ (no overlap)
- $0 \leq n \leq 6$: $y[n] = \sum_{i=0}^n (1/2)^{n-i} = 2(1 - (1/2)^{n+1})$
- $n > 6$: $y[n] = \sum_{i=0}^6 (1/2)^{n-i} = 127 \cdot (1/2)^n$

Values for $n = -6$ to 12 :

n:	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
8	9	10	11	12										
$y[n]:$	0	0	0	0	0	0	1.000	1.500	1.750	1.875	1.938	1.969	1.984	
	0.992	0.496	0.248	0.124	0.062	0.031								

Plot: (sketch)

- Rises exponentially from $n=0$ to $n=6$, approaching 2
- Decays exponentially from $n=7$ onward, following $127 \cdot (1/2)^n$

Part (b): Shift Property

With $x_2[n] = x[n-5]$ ($N=5$):

$$y_2[n] = h[n] * x_2[n] = y[n-5]$$

This demonstrates **time-invariance**: shifting the input shifts the output identically. The system response shape remains unchanged, only translated along the time axis.

Part (c): 2D Convolution

Given matrices (from figure):

Copy
<pre>X = [[6, 16, 22, 23], [?, ?, ?, ?], [?, ?, ?, ?], [?, ?, ?, ?]]</pre>
<pre>h = [[2, ?], [?, ?]]</pre>

Note: Specific values for X and h are incomplete in the provided image. Assuming a 4×4 input and 2×2 kernel:

For no padding, stride 1, the output y will be 3×3 :

$$y[m,n] = \sum_{i=0}^1 \sum_{j=0}^1 X[m+i, n+j] \cdot h[i,j]$$

Example with assumed values $X = [[6,16,22,23],[10,20,26,27],[14,24,30,31],[18,28,34,35]]$ and $h = [[2,3],[4,5]]$:

Copy

$$\begin{aligned}y[0,0] &= 6 \cdot 2 + 16 \cdot 3 + 10 \cdot 4 + 20 \cdot 5 = 200 \\y[0,1] &= 16 \cdot 2 + 22 \cdot 3 + 20 \cdot 4 + 26 \cdot 5 = 308 \\y[0,2] &= 22 \cdot 2 + 23 \cdot 3 + 26 \cdot 4 + 27 \cdot 5 = 362 \\... &\quad (\text{compute remaining 6 values similarly})\end{aligned}$$

Part (d): Striding and Padding

i. stride=1, pad=1 (circular):

- Output size = input size (e.g., 4×4 for 4×4 input)
- Each position uses wrapped-around values from edges

ii. stride=2, pad=1 (circular):

- Output size = $\text{ceil}(W/2) \times \text{ceil}(H/2)$ (e.g., 2×2 for 4×4 input)
- Skips every other position, reducing resolution by half

Problem 4: Feature Dimensions of CNN

Part (a): Conv Layer Parameters

Given: Input $W \times H \times C$, filter $K \times K$, padding P , stride S , F filters

Parameters:

- Weights: $K^2 \times C \times F$
- Biases: F
- Total: $K^2CF + F$

Output shape:

- Width: $\lfloor (W - K + 2P)/S \rfloor + 1$

- Height: $\lfloor (H - K + 2P)/S \rfloor + 1$
- Channels: F

Part (b): Pooling Layer Output Shape

Given: Input $W \times H \times C$, filter $K \times K$, stride S

Output shape:

- Width: $\lfloor (W - K)/S \rfloor + 1$
- Height: $\lfloor (H - K)/S \rfloor + 1$
- Channels: C (unchanged)
- Parameters: 0

Part (c): Receptive Field Size

For L successive conv layers with filter size K, stride 1:

Receptive field: $(L(K - 1)) + 1$

Each layer adds $K-1$ to the effective field since the first layer already covers K.

Part (d): Downsampling Effects

For 2×2 pooling with stride 2:

- **Receptive field increases by factor of 2** (doubled spatial extent)
- **Computational advantage:** Subsequent layers operate on $1/4$ the pixels, reducing computation by $4 \times$ per layer. This allows either deeper networks or faster training within the same budget.

Part (e): Network Table

Completed table:

Table			<input type="checkbox"/> Copy
Layer	Number of 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	$14 \times 14 \times 10$	

For parts A-D, kimi derived the correct answer, but for part e) being dependent on a the image, Kimi read the prompt as padding 0,(likely a hallucination or misreading), leading to an output size of 26×26 instead of 28×28 . It also skipped the second conv3-10 layer entirely.

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

For part f) it was partially correct: identified that the shifted image x_4 results in the same GAP output g_1 . However, for the diagonal image x_3 , Kimi guessed the activation would split between the first two channels. The official solution implies the network has a specific "diagonal" filter (Channel 3), which Kimi missed.

Part (f): Circular Padding Invariances

Given X_1 (horizontal edge) $\rightarrow g_1 = [0.8, 0, 0]^T$ and X_2 (vertical edge) $\rightarrow g_2 = [0, 0.8, 0]^T$.

With **circular padding**, the network exhibits **rotation-equivariance** on a torus:

- **X_3 (diagonal edge)**: This is a 45° rotation of X_1/X_2 . The network likely activates channel 3: $g_3 = [0, 0, 0.8]^T$
- **X_4 (position-shifted edge)**: Due to translation-equivariance of conv + circular padding, $g_4 = g_1$ (same as original horizontal edge)

The GAP layer is **invariant** to translations and **equivariant** to rotations (with circular padding).

Problem 5: Weights and Gradients in a CNN

Part (a): Weight Gradient Derivation

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

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

This is the **cross-correlation** between dY and X .

SGD update (learning rate η):

$$w_{\{h,l\}} \leftarrow w_{\{h,l\}} - \eta \cdot dw_{\{h,l\}}$$

Part (b): Gradient Statistics

Given $E[x] = 0$, $\text{Var}(x) = \sigma_x^2$, $E[dy] = 0$, $\text{Var}(dy) = \sigma_y^2$, and independence:

$$\text{Mean: } E[dw_{\{h,l\}}] = \sum \sum E[x]E[dy] = 0$$

$$\text{Variance: } \text{Var}(dw_{\{h,l\}}) = \sum \sum \text{Var}(x \cdot dy) = m^2 \sigma_x^2 \sigma_y^2$$

Since $m = n - k + 1$:

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

$$\sigma(dw_{\{h,l\}}) = (n - k + 1) \sigma_x \sigma_y$$

Asymptotic growth: $O(n)$ (linear in image size)

Part (c): Pooling Backpropagation

2x2 Max Pooling:

- Forward: $y_{i,j} = \max(x_{2i-1,2j-1}, x_{2i-1,2j}, x_{2i,2j-1}, x_{2i,2j})$
- Backward: $dX_{p,q} = dy_{\lceil p/2 \rceil, \lceil q/2 \rceil}$ if $x_{p,q}$ was the max, else 0

2x2 Average Pooling:

- Forward: $y_{i,j} = (1/4) \sum_{p,q \in \text{block}} x_{p,q}$
- Backward: $dX_{p,q} = dy_{\lceil p/2 \rceil, \lceil q/2 \rceil} / 4$ (equal distribution to all 4 positions)

Part (d): Pooling Advantages

Max Pooling:

- **Feature selection:** Preserves only most salient features (edges, textures)
- **Sparsity:** Creates sparse representations beneficial for deep networks
- **Translation invariance:** Small shifts don't change max output
- **Better for classification:** Focuses on discriminative patterns

Average Pooling:

- **Smoothness:** Blurs features, reduces noise sensitivity
- **Gradient flow:** Gradients reach all positions, better for optimization
- **Magnitude preservation:** Maintains average signal strength
- **Better for localization:** Preserves spatial information more evenly

Trade-off: Max pooling excels at **discrimination** (classification), while average pooling excels at **preservation** (reconstruction, segmentation).

This model only extracts text from images, switch to K1.5 for full visual understanding.

