

Special Participation A

Exploration of Different Input Forms on HW5 (ChatGPT 5.1 Auto)

When we use large language models to solve knowledge-based problems, we may encounter the input of images/formulas. Taking hw5 as an example, I tried:

- 1) text input only (without any formulas),
- 2) text input + image input (there might be some formula garbled),
- 3) inputting the entire question as an image,
- 4) directly inputting the pdf.

Then, I observed the problem-solving capabilities of the large language model under different inputs.

Q1:

I am taking Deep Neural Networks class this semester. Now I have some problems when doing the practice about Pooling/Downsampling, Data Augmentation, Normalization Layers, Dropout. I hope you can assist me in solving the problem and provide me with a detailed solution approach.

A1:

Got you 😊 That's a nice cluster of topics—pooling, data augmentation, normalization, dropout all tend to show up together in assignments.

To really help, I'll need to see the **exact questions / code / error messages** you're struggling with (you can just paste them). But instead of leaving you hanging, let me first set up a **compact but detailed cheat-sheet + solution patterns** for these topics so when you send the questions, we can plug into this logic directly. 1. Pooling / Downsampling

1.1 What pooling is doing (intuition)

- **Max pooling:** keeps the strongest activation (e.g., edge presence).
- **Average pooling:** averages all activations (smoother, more “blurred”).
- Purpose:
 - Reduce spatial size → fewer parameters, less compute
 - Provide **translation invariance** (small shifts in input don't change pooled output much)
 - Control overfitting

1.2 Typical questions & how to solve them

(a) Manually compute output of pooling layer

Given:

- Input feature map shape: \$(H, W, C)\$
- Pool size: \$k \times k\$
- Stride: \$s\$

- Padding: p

Output height/width:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p - k}{s} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1 \quad (1)$$

Steps to answer:

1. Compute $H_{\text{out}}, W_{\text{out}}$ with formula.
 2. For each window, explicitly take max/average.
 3. If there are multiple channels, pooling is applied **per channel independently**.
-

(b) Difference between pooling vs strided convolution vs global average pooling

- **Strided conv:** Learnable filters; downsamples *and* transforms.
- **Pooling:** No learnable parameters; just aggregates.
- **Global average pooling:** Average over entire spatial map → output shape becomes $(C,)$. Often used before final FC layer / classifier.

When an exam asks “*Why use pooling instead of large stride convolutions?*”, you can say:

- Pooling provides a fixed, simple inductive bias: invariance and robustness, fewer parameters, less risk of overfitting.
 - Strided conv is more flexible but can overfit and is more computationally expensive.
-

2. Data Augmentation

2.1 Core idea

- **Goal:** Simulate plausible variations of data (flips, crops, noise, color jitter) to improve generalization.
- Used **only on training data**, never on validation/test.

2.2 Typical questions

(a) Conceptual: Why does data augmentation help generalization?

Key points:

- It effectively increases dataset size.
- Encourages **invariance** to transformations (e.g., rotation, flipping, small translation).
- Acts as a form of **regularization**: the model must perform well on many variations, not memorize specific examples.

(b) Implementing augmentation (e.g., in PyTorch)

Pattern:

```
import torchvision.transforms as T

train_transforms = T.Compose([
    T.RandomResizedCrop(224),
```

```

T.RandomHorizontalFlip(p=0.5),
T.ColorJitter(brightness=0.2, contrast=0.2),
T.ToTensor(),
T.Normalize(mean, std),
])

test_transforms = T.Compose([
    T.Resize(256),
    T.CenterCrop(224),
    T.ToTensor(),
    T.Normalize(mean, std),
])

```

Common pitfalls they like to test:

- Don't put random augmentations in **test/validation**.
- Normalization (`Normalize`) usually goes after converting to tensor.

3. Normalization Layers (BatchNorm / LayerNorm / etc.)

3.1 Batch Normalization (most common)

For a mini-batch x of shape (N, C, H, W) for CNNs or (N, D) for MLPs:

1. Compute mean/variance **per feature/channel** over the batch (and sometimes spatial dims):

$$\mu = \frac{1}{m} \sum_{i=1}^m x_i, \quad \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2 \quad (2)$$

2. Normalize:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (3)$$

3. Apply learnable affine transform:

$$y_i = \gamma \hat{x}_i + \beta \quad (4)$$

where γ, β are parameters.

3.2 Why BN helps (answers for conceptual questions)

- **Reduces internal covariate shift** (original explanation; conceptually: stabilizes the distribution of activations).
- Allows higher learning rates.
- Acts like a regularizer (mini-batch noise).
- Speeds up convergence.

3.3 Train vs Eval mode (very common exam bug)

- **During training:**
 - Use **batch statistics** (current mini-batch mean and variance).

- Update running estimates (moving average) of mean/variance.
- **During evaluation:**
 - Use **running mean and variance**, not batch stats.

Typical PyTorch pattern:

```
model.train()  # BN & Dropout in training mode
...
model.eval()   # BN uses running stats, Dropout off
```

If they show you an accuracy drop because someone forgot to call `model.eval()`, you know why.

4. Dropout

4.1 Forward pass behavior

During **training**:

- For each neuron activation h , sample a Bernoulli mask:

$$m \sim \text{Bernoulli}(p_{\text{keep}}) \quad (5)$$

$$h' = \frac{m}{p_{\text{keep}}} \cdot h \quad (6)$$

- So with probability $1 - p_{\text{keep}}$, the activation is zeroed.
- The scaling by $1/p_{\text{keep}}$ ensures the expected value stays the same.

During **test**:

- Use full network, no dropping; no scaling in “inverted dropout” (which is how frameworks usually do it).

4.2 Why dropout works (conceptual answer)

- Forces network to not rely on any single neuron.
- Approximate ensemble of many thinned networks.
- Reduces co-adaptation and overfitting.

4.3 Typical pitfalls they ask about

- Using very large dropout rates → underfitting.
 - Using dropout **and** BatchNorm together: usually apply dropout **after** activations (and often after BN + nonlinearity in MLPs; for CNNs, sometimes only in classifier head).
 - Important to turn dropout off with `model.eval()` at test time.
-

5. How to Approach Your Specific Problems

When you send your questions, we'll likely need to:

1. Identify the type of question

- Math/shape reasoning (e.g., "What is output size after pooling + conv?").
- Implementation bug (code snippet).
- Conceptual comparison (e.g., "BatchNorm vs LayerNorm vs InstanceNorm").
- Training behavior (e.g., "Why does training accuracy drop when I enable dropout?").

2. Apply a structured checklist

- For **pooling**:
 - Input shape, kernel, stride, padding → compute output size.
 - Manual window operations if needed.
- For **augmentation**:
 - Only in train? Are operations appropriate (don't distort labels)?
- For **normalization**:
 - Are you using `train()` / `eval()` correctly?
 - Are you normalizing over correct dimension?
- For **dropout**:
 - Are you using dropout rate reasonably? (0.1–0.5 typical)
 - Is dropout disabled at test time?

3. Write a step-by-step solution

- For numeric questions: compute shapes and values explicitly.
- For theory: give 3–4 bullet points that answer "what / why / when".

What I need from you next

Send me **one of your actual practice problems** (or several), for example:

- A screenshot / text of the question, or
- A piece of code that's not behaving as expected, plus what you expected vs what you observe.

I'll then:

- Walk through it step by step.
- Tie it to the formulas/ideas above.
- Show you the reasoning so you can reuse the pattern on similar questions.

Whenever you're ready, paste the first question and we'll dissect it together.

Q2:

1. Convolutional Networks

Note: Throughout this problem, we will use the convention of NOT flipping the filter before dragging it over the signal. This is the standard notation with neural networks (ie, we assume the filter given to us is already

flipped)

(a) List two reasons we typically prefer convolutional layers instead of fully connected layers when working with image data.

(b) Consider the following 1D signal: [1, 4, 0, -2, 3]. After convolution with a length-3 filter, no padding, stride=1, we get the following sequence: [-2, 2, 11]. What was the filter?

(Hint: Just to help you check your work, the first entry in the filter that you should find is 2. However, if you try to use this hint directly to solve for the answer, you will not get credit since this hint only exists to help you check your work.)

(c) Transpose convolution is an operation to help us upsample a signal (increase the resolution). For example, if our original signal were [a, b, c] and we perform transpose convolution with pad=0 and stride=2, with the filter [x, y, z], the output would be [ax, ay, az +bx, by, bz +cx, cy, cz]. Notice that the entries of the input are multiplied by each of the entries of the filter. Overlaps are summed. Also notice how for a fixed filtersize and stride, the dimensions of the input and output are swapped compared to standard convolution. (For example, if we did standard convolution on a length-7 sequence with filtersize of 3 and stride=2, we would output a length-3 sequence).

If our 2D input is [-1 2; 3,1] and the 2D filter is " [+1 -1;0 +1]

What is the output of transpose convolution with pad=0 and stride=1?

A2:

Great, this is a classic CNN fundamentals problem. I'll go **part by part** and show the **reasoning explicitly**, the way graders expect to see it.

1(a) Why convolutional layers instead of fully connected layers for images?

Two standard reasons (either of these pairs would be full credit):

Reason 1: Parameter sharing → far fewer parameters

- A convolutional filter is reused across all spatial locations.
- This drastically reduces the number of parameters compared to fully connected layers.
- Fewer parameters → less overfitting, more efficient learning.

Reason 2: Exploiting spatial structure (locality & translation invariance)

- Images have strong **local correlations** (nearby pixels are related).
- Convolutions capture local patterns (edges, textures).
- The same feature can be detected anywhere in the image → **translation equivariance**.

(Alternative acceptable reasons: better generalization, computational efficiency, inductive bias for vision.)

1(b) Recovering the 1D convolution filter

Given

- Input signal:

$$x = [1, 4, 0, -2, 3] \quad (7)$$

- Filter length = 3
- No padding, stride = 1
- Output:

$$y = [-2, 2, 11] \quad (8)$$

- **Filter is NOT flipped**
- Hint: first filter entry is **2**

Let the filter be:

$$w = [w_1, w_2, w_3] \quad (9)$$

Step 1: Write convolution equations

Each output is a dot product:

First output

$$1w_1 + 4w_2 + 0w_3 = -2 \quad (10)$$

Second output

$$4w_1 + 0w_2 + (-2)w_3 = 2 \quad (11)$$

Third output

$$0w_1 + (-2)w_2 + 3w_3 = 11 \quad (12)$$

Step 2: Use the hint $w_1 = 2$

From output 1:

$$1(2) + 4w_2 = -2 \Rightarrow 4w_2 = -4 \Rightarrow w_2 = -1 \quad (13)$$

From output 2:

$$4(2) - 2w_3 = 2 \Rightarrow 8 - 2w_3 = 2 \Rightarrow w_3 = 3 \quad (14)$$

✓ Final filter

$$\boxed{[2, -1, 3]} \quad (15)$$

Quick check (recommended in exams)

$$0(2) + (-2)(-1) + 3(3) = 2 + 9 = 11 \checkmark \quad (16)$$

1(c) 2D Transpose Convolution

Given

- Input:

$$X = \begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix} \quad (17)$$

- Filter:

$$K = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \quad (18)$$

- Pad = 0, Stride = 1
 - **Transpose convolution** (no flipping)
-

Step 1: Output size

For transpose convolution with stride 1 and pad 0:

$$(H + K_H - 1) \times (W + K_W - 1) = 3 \times 3 \quad (19)$$

Initialize output:

$$Y = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (20)$$

Step 2: Place weighted filters and sum overlaps

Input (0,0) = -1

Add \$-1 \cdot K\$ at (0,0):

$$\begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix} \quad (21)$$

Input (0,1) = 2

Add \$2 \cdot K\$ at (0,1):

$$\begin{bmatrix} 2 & -2 \\ 0 & 2 \end{bmatrix} \quad (22)$$

Input (1,0) = 3

Add \$3 \cdot K\$ at (1,0):

$$\begin{bmatrix} 3 & -3 \\ 0 & 3 \end{bmatrix} \quad (23)$$

Input (1,1) = 1

Add \$1 \cdot K\$ at (1,1):

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \quad (24)$$

Step 3: Sum everything

Final output:

$$\boxed{\begin{bmatrix} -1 & 3 & -2 \\ 3 & -3 & 1 \\ 0 & 3 & 1 \end{bmatrix}} \quad (25)$$

✓ Final Answers Summary

(a)

- Parameter sharing → fewer parameters
- Exploits spatial locality + translation invariance

(b)

$$\boxed{[2, -1, 3]} \quad (26)$$

(c)

$$\boxed{\begin{bmatrix} -1 & 3 & -2 \\ 3 & -3 & 1 \\ 0 & 3 & 1 \end{bmatrix}} \quad (27)$$

If you want, next we can:

- Compare **transpose convolution vs upsampling + convolution**
- Do a **similar question with padding/stride ≠ 1**
- Or move on to **pooling / normalization / dropout** problems from the same homework set

Correct! And GPT provided a detailed process as I requested.

Q3:

2. Batch Normalization for CNN

(a) Consider the following diagram where the shaded blocks are the entries participating in one normalization step for a CNN-type architecture. N represents the mini-batch, H, W represent the different pixels of the “image” at this layer, and C represents different channels.

- Which one denotes the process of batch normalization? Please use ■ for your selections.

A B C

- Which one denotes layer normalization? Please use ■ for your selections.

A B C

(b) Consider a simplified BN where we do not divide by the standard deviation of the data batch. Instead, we just de-mean our data batch before applying the scaling factor γ and shifting factor β . For simplicity, consider scalar data in an n-sized batch: $[x_1, x_2, \dots, x_n]$. Specifically, we let $\hat{x}_i = x_i - \mu$ where

μ is the average n

$\frac{1}{n} \sum x_j$

$j=1$ across the batch and output $[y_1, y_2, \dots, y_n]$ where $y_i = \gamma \hat{x}_i + \beta$ to the next layer. Assume we have a final loss L somewhere downstream. Calculate $\frac{\partial L}{\partial x_i}$

$\frac{\partial L}{\partial x_i}$

i

in terms of $\frac{\partial y}{\partial x_i}$

$\frac{\partial L}{\partial y_j}$

j

for $j = 1, \dots, n$ as well as γ and β as needed.

Numerically, what is $\frac{\partial L}{\partial x_1}$

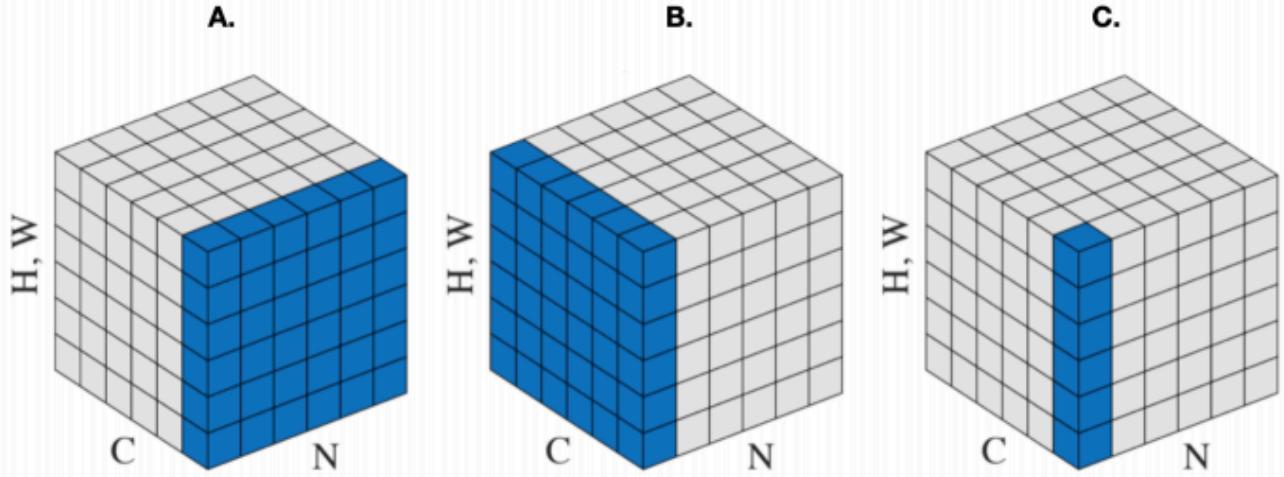
$\frac{\partial L}{\partial x_1}$

1

when $n = 1$ and our input batch just consists of $[x_1]$ with an output

batch of $[y_1]$? (Your answer should be a real number. No need to justify.)

What happens when $n \rightarrow \infty$? (Feel free to assume here that all relevant quantities are bounded.)



A3:

Thought 1m 51s >

Nice one – this is a very “standard” BN/LN picture + gradient question.

I'll go part by part.

2(a) Which is BatchNorm, which is LayerNorm?

Recall:

- **BatchNorm for CNNs:** for each channel c , we normalize over **all samples in the batch and all spatial locations**: axes = (N, H, W) , at fixed C .
- **LayerNorm:** for each sample n , we normalize over **all features of that sample**, i.e. over (C, H, W) , at fixed N .

Look at the cubes:

- **A:** the blue region is one whole “sample” (one N) across all C, H, W .
→ that matches **LayerNorm**.
- **B:** the blue region is one whole “channel” across all samples N and all spatial positions H, W .
→ that matches **BatchNorm**.
- **C:** only a thin column (one spatial position only) – neither BN nor standard LN.

So:

- **Batch normalization:** ■ B
- **Layer normalization:** ■ A

2(b) Gradient for simplified BN without variance

We have a batch of scalars:

$$x_1, x_2, \dots, x_n \quad (28)$$

Batch mean:

$$\mu = \frac{1}{n} \sum_{j=1}^n x_j \quad (29)$$

Centered values:

$$\hat{x}_i = x_i - \mu \quad (30)$$

Output:

$$y_i = \gamma \hat{x}_i + \beta = \gamma(x_i - \mu) + \beta \quad (31)$$

Flight $\frac{\partial L}{\partial y_j}$.

We want $\frac{\partial L}{\partial x_i}$ in terms of the y_j and γ .

Step 1: Derivative of y_j w.r.t. x_i

$$y_j = \gamma(x_j - \mu) + \beta \quad (32)$$

- $\frac{\partial y_j}{\partial x_i} = \delta_{ij}$ (Kronecker delta)
- $\mu = \frac{1}{n} \sum_{k=1}^n x_k \Rightarrow \frac{\partial \mu}{\partial x_i} = \frac{1}{n}$

So:

$$\frac{\partial y_j}{\partial x_i} = \gamma \left(\frac{\partial x_j}{\partial x_i} - \frac{\partial \mu}{\partial x_i} \right) = \gamma \left(\delta_{ij} - \frac{1}{n} \right) \quad (33)$$

Step 2: Chain rule

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^n \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^n g_j \gamma \left(\delta_{ij} - \frac{1}{n} \right) \quad (34)$$

Flight $S = \sum_{j=1}^n g_j$. Then:

$$\frac{\partial L}{\partial x_i} = \gamma \left(g_i - \frac{1}{n} S \right) = \gamma \left(\frac{\partial L}{\partial y_i} - \frac{1}{n} \sum_{j=1}^n \frac{\partial L}{\partial y_j} \right) \quad (35)$$

That's the general expression.

Special case: $n = 1$

Batch is just $[x_1]$.

Then:

$$\frac{\partial L}{\partial x_1} = \gamma \left(g_1 - \frac{1}{1} g_1 \right) = 0 \quad (36)$$

So **numerically**:

$$\boxed{\frac{\partial L}{\partial x_1} = 0} \quad (37)$$

(Intuitively: with a 1-element batch, $\mu = x_1$, so $\hat{x}_1 = 0$ and $y_1 = \beta$ does not depend on x_1 at all.)

What happens as $n \rightarrow \infty$?

We have:

$$\frac{\partial L}{\partial x_i} = \gamma \left(g_i - \frac{1}{n} \sum_{j=1}^n g_j \right) \quad (38)$$

If we assume all relevant quantities are bounded (in particular, that $\sum_j g_j$ does not grow faster than $O(1)$), then:

$$\frac{1}{n} \sum_{j=1}^n g_j \rightarrow 0 \quad (39)$$

and thus

$$\boxed{\frac{\partial L}{\partial x_i} \rightarrow \gamma g_i = \gamma \frac{\partial L}{\partial y_i}} \quad (40)$$

So in the infinite-batch limit, the “mean-subtraction” term has vanishing influence on the gradient; it behaves like a simple affine transform $y_i \approx \gamma x_i + \beta$.

For the second question, I attempted to directly copy the text and upload the screenshot as an image. This resulted in some characters appearing as garbled text. For the third question, I tried to take a screenshot of the entire question and send it to the large model.

Result:

- a) is not correct. This indicates a deficiency in multi-modal capabilities.
- b) is correct. However, the derivation process is overly complicated and involves variables such as g .

Q4:

3. Depthwise Separable Convolutions

Depthwise separable convolutions are a type of convolutional operation used in deep learning for image processing tasks. Unlike traditional convolutional operations, which perform both spatial and channel-wise convolutions simultaneously, depthwise separable convolutions decompose the convolution operation into two separate operations: Depthwise convolution and Pointwise convolution.

This can be viewed as a low-rank approximation to a traditional convolution. For simplicity, throughout this problem, we will ignore biases while counting learnable parameters.

- (a) Suppose the input is a three-channel 224×224 -resolution image, the kernel size of the convolutional layer is 3×3 , and the number of output channels is 4.

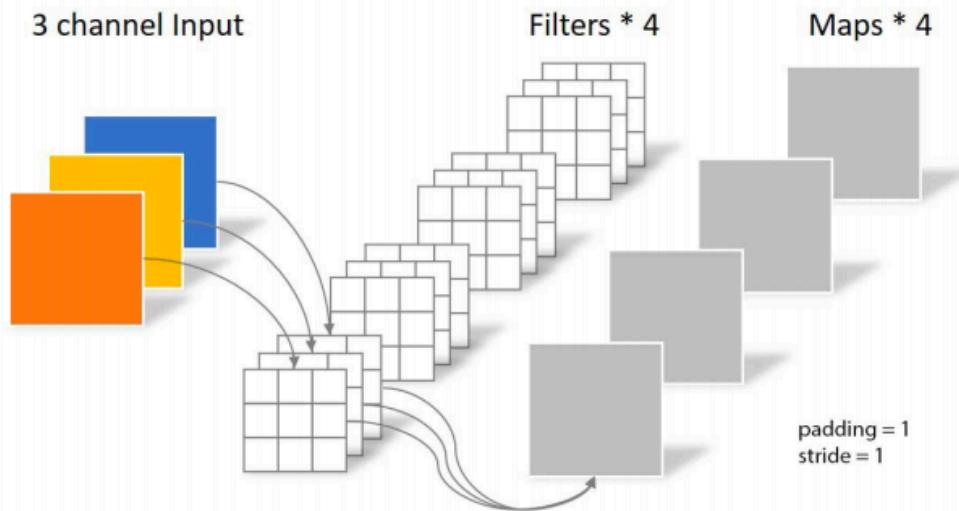


Figure 1: Traditional convolution.

What is the number of learnable parameters in the traditional convolution layer?

- (b) Depthwise separable convolution consists of two parts: depthwise convolutions (Fig.2) followed by pointwise convolutions (Fig.3). Suppose the input is still a three-channel 224×224 -resolution image. The input first goes through depthwise convolutions, where the number of output channels is the same as the number of input channels, and there is no “cross talk” between different channels. Then, this intermediate output goes through pointwise convolutions, which is basically a traditional convolution with the filter size being 1×1 . Assume that we have 4 output channels.

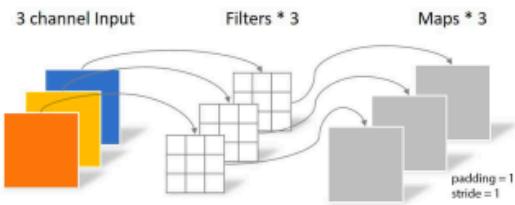


Figure 2: Depthwise convolution.

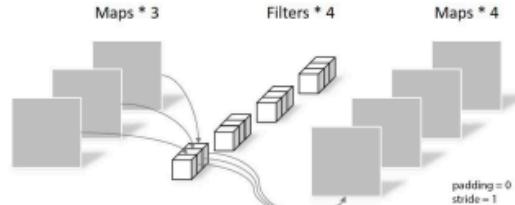


Figure 3: Pointwise convolution

What is the total number of learnable parameters of the depthwise separable convolution layer which consists of both depthwise and pointwise convolutions?

A4:

Thought 18s >

Nice, this is a classic parameter-counting question.

(a) Traditional convolution

- Input: 3 channels
- Kernel size: 3×3
- Output channels: 4
- No bias terms.

For a standard conv, **each output channel** has one kernel of size $(3 \times 3 \times \text{in_channels}) = 3 \times 3 \times 3 = 27$ weights.

Total parameters:

$$\# \text{params} = 4 \times 3 \times 3 \times 3 = 4 \times 27 = \boxed{108} \quad (41)$$

(b) Depthwise separable convolution

Two parts: **depthwise** then **pointwise**.

1. Depthwise convolution

- One 3×3 filter per input channel (no cross-channel mixing).
- 3 input channels \Rightarrow 3 filters.
- Each filter has $3 \times 3 = 9$ weights.

$$\# \text{depthwise params} = 3 \times 9 = 27 \quad (42)$$

(Equivalently: $3 \times 3 \times 3$ with depth multiplier 1.)

2. Pointwise convolution (1×1)

- Acts like a fully connected layer across channels at each spatial location.
- Kernel size: 1×1
- In-channels = 3, out-channels = 4.

Each output channel has a $1 \times 1 \times 3$ kernel \rightarrow 3 weights.

So:

$$\# \text{pointwise params} = 4 \times 3 = 12 \quad (43)$$

Total depthwise separable params

$$\# \text{total} = 27 + 12 = \boxed{39} \quad (44)$$

So: traditional conv has **108** learnable parameters, while the depthwise separable version has **39**.

For the third question, I tried to take a screenshot of the entire question and send it to the large model.
Result: Correct!

Q5:

Please answer the question '4. Regularization and Dropout' in the PDF I provided.

[hw5.pdf]

A5:

Thought 1m 41s >

Here we go — full solution to **Q4. Regularization and Dropout**.

Setup recap

We have linear regression with dropout on the **input features**:

$$L(\check{w}) = \mathbb{E}_{R \sim \text{Bern}(p)} \left[\|y - (R \odot X)\check{w}\|_2^2 \right] \quad (45)$$

where $R_{ij} \sim \text{i.i.d. Bernoulli}(p)$, and $X \in \mathbb{R}^{n \times d}$ is fixed.

Let Γ_j^\vee (their $\check{\Gamma}_j$) be diagonal with

$$\Gamma_{jj}^\vee = \|X_{:,j}\|_2 \quad (46)$$

(the norm of column j of X). hw5

4(a) Show that the expected loss becomes ridge with a data-dependent penalty

Write $X' = R \odot X$. Then

$$\|y - X'\check{w}\|^2 = (y - X'\check{w})^\top (y - X'\check{w}) = y^\top y - 2y^\top X'\check{w} + \check{w}^\top X'^\top X'\check{w}. \quad (47)$$

Take expectation over R .

1. **First term:** $\mathbb{E}[y^\top y] = y^\top y$ (constant).

2. **Second term:**

Since $\mathbb{E}[R_{ij}] = p$ and X is fixed,

$$\mathbb{E}[X'] = \mathbb{E}[R \odot X] = pX \quad (48)$$

so

$$\mathbb{E}[y^\top X'\check{w}] = y^\top \mathbb{E}[X']\check{w} = y^\top pX\check{w}. \quad (49)$$

Thus the expected cross-term is $-2y^\top pX\check{w}$.

1. **Third term:** $\mathbb{E}[X'^\top X']$.

The (j,k) -entry of $X'^\top X'$ is

$$(X'^\top X')_{jk} = \sum_{i=1}^n X'_{ij} X'_{ik} = \sum_i R_{ij} R_{ik} X_{ij} X_{ik}. \quad (50)$$

- If $j \neq k$: R_{ij} and R_{ik} are independent Bernoulli(p), so $\mathbb{E}[R_{ij}R_{ik}] = p^2$. Hence

$$\mathbb{E}[(X'^\top X')_{jk}] = p^2 \sum_i X_{ij} X_{ik} = p^2 (X^\top X)_{jk}. \quad (51)$$

- If $j=k$: $R_{ij}^2 = R_{ij}$, so $\mathbb{E}[R_{ij}^2] = p$. Then

$$\mathbb{E}[(X'^\top X')_{jj}] = p \sum_i X_{ij}^2 = p (X^\top X)_{jj}. \quad (52)$$

We can write this compactly as

$$\mathbb{E}[X'^\top X'] = p^2 X^\top X + p(1-p) \text{diag}(X^\top X), \quad (53)$$

because on the diagonal we want p , but p^2 already appears in the first term, so we add $(p-p^2)\text{diag}(X^\top X) = p(1-p)\text{diag}(X^\top X)$.

Flight Γ be diagonal with $\Gamma_{jj}^2 = (X^\top X)_{jj}$, i.e. $\Gamma = \text{diag}(X^\top X)^{1/2}$. Then

$$\check{w}^\top \mathbb{E}[X'^\top X'] \check{w} = p^2 \check{w}^\top X^\top X \check{w} + p(1-p) \check{w}^\top (\Gamma^\top)^2 \check{w} = p^2 \|X \check{w}\|^2 + p(1-p) \|\Gamma^\top \check{w}\|^2. \quad (54)$$

Putting everything together:

$$\mathbb{E}\|y - X' \check{w}\|^2 = y^\top y - 2py^\top X \check{w} + p^2 \check{w}^\top X^\top X \check{w} + p(1-p) \|\Gamma^\top \check{w}\|^2 = \|y - pX \check{w}\|^2 + p(1-p) \|\Gamma^\top \check{w}\|^2. \quad (55)$$

So indeed

$$L(\check{w}) = \|y - pX \check{w}\|_2^2 + p(1-p) \|\Gamma^\top \check{w}\|_2^2 \quad (56)$$

with Γ as defined.

4(b) How to transform \check{w} to look like solution to (3)?

The “classical” Tikhonov problem (3) uses the original data matrix X :

$$L(w) = \|y - Xw\|^2 + \|\Gamma w\|^2. \quad (57)$$

From (5), our dropout-trained objective is

$$L(\check{w}) = \|y - pX \check{w}\|^2 + p(1-p) \|\Gamma^\top \check{w}\|^2. \quad (58)$$

Define a rescaled weight vector

$$w := p\check{w}. \quad (59)$$

Then $\check{w} = \frac{1}{p} w$, and

- Data term:

$$\|y - pX \check{w}\|^2 = \|y - Xw\|^2. \quad (60)$$

- Regularizer:

$$p(1-p)\|\Gamma^\vee \check{w}\|^2 = p(1-p)\left\|\Gamma^\vee \frac{1}{p}w\right\|^2 = \frac{1-p}{p}\|\Gamma^\vee w\|^2. \quad (61)$$

So in terms of w , the objective is

$$L(w) = \|y - Xw\|^2 + \frac{1-p}{p}\|\Gamma^\vee w\|^2, \quad (62)$$

which is exactly of the form (3) with

- $\Gamma = \sqrt{\frac{1-p}{p}}\Gamma^\vee$

Answer: multiply the dropout-trained weights by p :

$$w = p\check{w} \quad (63)$$

Then w is the minimizer of a standard Tikhonov problem with suitably scaled Γ .

(This is analogous to “inverted dropout,” where we rescale weights/activations when going to test-time.)

4(c) Change of variables to classical ridge form

Original generalized Tikhonov problem:

$$L(w) = \|y - Xw\|^2 + \|\Gamma w\|^2, \quad (64)$$

with Γ invertible.

Flight

$$w = \Gamma^{-1}w_e. \quad (65)$$

Then

- Regularizer:

$$\|\Gamma w\|^2 = \|\Gamma\Gamma^{-1}w_e\|^2 = \|w_e\|^2. \quad (66)$$

- Data term:

$$\|y - Xw\|^2 = \|y - X\Gamma^{-1}w_e\|^2 = \|y - X_e w_e\|^2, \quad (67)$$

where we define

$$X_e := X\Gamma^{-1}. \quad (68)$$

Thus the problem becomes

$$L(w_e) = \|y - X_e w_e\|^2 + \|w_e\|^2, \quad (69)$$

which is classical ridge regression (with $\lambda=1$; more generally one can absorb any λ into Γ).

4(d) Column norms of X_e and relation to batchnorm

Now assume Γ is **diagonal** and invertible, with

$$\Gamma_{jj} \propto \|X_{:,j}\|_2. \quad (70)$$

From part (c), $X_e = X\Gamma^{-1}$.

The j -th column of X_e is

$$(X_e)_{:,j} = \frac{1}{\Gamma_{jj}} X_{:,j}. \quad (71)$$

So its norm is

$$\|(X_e)_{:,j}\| = \frac{1}{|\Gamma_{jj}|} \|X_{:,j}\| \propto \frac{\|X_{:,j}\|}{\|X_{:,j}\|} = \text{constant.} \quad (72)$$

If we choose $\Gamma_{jj} = |X_{:,j}|$ exactly, **all columns of X_e have unit norm**.

Conclusion: after the change of variables, the *effective* training matrix X_e has (approximately) equal-norm columns —i.e., its features are normalized.

Speculation on dropout vs batchnorm

- Dropout on inputs is equivalent (in expectation) to **data-dependent L2 regularization** whose strength depends on the column norms of X ; via the above change of variables, this is equivalent to doing ridge regression on a feature-normalized matrix X_e .
- Batch normalization, on the other hand, explicitly **normalizes activations** (features) to have similar mean and variance during training.

So:

- Both dropout (through its induced Tikhonov penalty) and batchnorm favor representations where features have similar scale.
- This suggests a connection: dropout behaves a bit like training on a normalized feature space (plus noise), while batchnorm directly normalizes intermediate features, which in turn changes how strong dropout's effective regularization is on each unit.

For the fourth question, I tried to upload the whole pdf and ask ChatGPT to answer the specific question.

Result: Totally Correct!