

Practice Questions on NN, CNNs, and Activation Functions


Questions

? In a feedforward neural network, what ensures information flows only from input to output without loops?

Options:

- a) Recurrent connections
 - b) Weight sharing
 - c) Directed acyclic structure
 - d) Convolution
-

Answer Analysis

- **a) Recurrent connections → Incorrect**
 - Recurrent connections *introduce loops* in the network (like in RNNs), where outputs are fed back into earlier layers. This is the opposite of what we want in a feedforward NN.
- **b) Weight sharing → Incorrect**
 - Weight sharing is a property mainly seen in CNNs (convolutional layers) where the same kernel/weights are applied across different spatial positions. It does not determine the flow of information or prevent loops.
- **c) Directed acyclic structure →  Correct**
 - Feedforward neural networks are *Directed Acyclic Graphs (DAGs)*.
 - "Directed" → information flows forward, from input to hidden to output.

- "Acyclic" → no cycles/loops exist, ensuring data doesn't get stuck in feedback paths.
- This is exactly what ensures input → output flow without loops.
- **d) Convolution → Incorrect**
 - Convolution is an operation used in CNNs for feature extraction. It does not inherently ensure absence of loops—it's just one type of layer.




What is the main role of bias in a neuron?

Options:

- a) To scale the input
- b) To shift the activation threshold
- c) To reduce overfitting
- d) To normalize input

Answer Analysis

- **a) To scale the input → Incorrect**
 - Scaling of inputs is controlled by *weights*, not the bias term.
 - Weights determine how strongly each input influences the neuron.
- **b) To shift the activation threshold →  Correct**
 - Bias allows the activation function to be shifted left or right.
 - Without bias, the neuron's output would always be forced through the origin (0,0).
 - With bias, the model can better fit data by adjusting where the "threshold" for activation lies.
- **c) To reduce overfitting → Incorrect**
 - Overfitting is controlled with methods like dropout, weight decay (regularization), or early stopping.


- Bias has no direct role in reducing overfitting.
- **d) To normalize input → Incorrect**
 - Normalization (e.g., batch normalization, min-max scaling) is a preprocessing or architectural step.
 - Bias doesn't normalize—it just shifts the activation.

? The vanishing gradient problem is most severe when using:

Options:

- a) ReLU
- b) Tanh
- c) Sigmoid
- d) Leaky ReLU

Answer Analysis

- **a) ReLU → Incorrect**
 - ReLU avoids vanishing gradients in the positive region (derivative = 1).
 - But it can suffer from the *dying ReLU problem* (gradient = 0 for negative inputs).
 - Not the main culprit for vanishing gradients.
- **b) Tanh → Partially correct, but not the worst**
 - Tanh squashes outputs to $[-1, 1]$.
 - For large positive/negative inputs, derivatives approach 0 → gradient vanishes.
 - However, it's still *less severe* than sigmoid since outputs are centered around 0.
- **c) Sigmoid →  Correct**

- Sigmoid squashes inputs to $[0, 1]$.
- For large $|x|$, derivative is very small (< 0.01).
- When stacked across many layers, this leads to severe *vanishing gradients*.
- Training deep networks with sigmoid is very difficult.
- **d) Leaky ReLU → Incorrect**
 - Leaky ReLU allows a small slope (like $0.01x$) for negative inputs.
 - This prevents gradients from completely vanishing (fixes “dying ReLU” issue).
 - Not prone to vanishing gradients like sigmoid/tanh.

👉 Quick memory tip:

- **Sigmoid = worst for vanishing gradient**
- **Tanh = also shrinks gradients, but less bad**
- **ReLU/Leaky ReLU = better for deep networks**

? What is the primary benefit of parameter sharing in CNNs?


Options:

- a) Faster backpropagation
- b) Reduced number of learnable parameters
- c) Increased receptive field
- d) Avoids overfitting completely

Answer Analysis


- **a) Faster backpropagation → Incorrect**
 - While fewer parameters *indirectly* make training faster, the *primary* benefit of parameter sharing is not speed of backprop—it’s about reducing

parameter count.

- **b) Reduced number of learnable parameters →  Correct**
 - In CNNs, the same filter/kernel weights are used across the entire image.
 - This means instead of learning unique weights for every pixel connection (like in a fully connected layer), the network reuses weights, massively reducing parameters.
 - Example: a 3×3 filter has only 9 weights, no matter how large the image is.
- **c) Increased receptive field → Incorrect**
 - The receptive field grows with *stacking more layers* or using larger kernels/pooling, not because of parameter sharing itself.
- **d) Avoids overfitting completely → Incorrect**
 - While fewer parameters can *reduce risk* of overfitting, it doesn't eliminate it. Regularization methods (dropout, data augmentation, etc.) are still needed.

 Quick intuition:


- Without parameter sharing → "memorization machine" (too many weights).
- With parameter sharing → "pattern detector" (fewer weights, reusable features).

 Which layer in a CNN usually helps in translation invariance?

Options:


- a) Fully connected
- b) Convolution
- c) Pooling
- d) BatchNorm

Answer Analysis

- **a) Fully connected → Incorrect**
 - Fully connected layers treat every input separately.
 - They don't naturally capture translation invariance—shifting the image even slightly can drastically change activations.
- **b) Convolution → Partially correct, but not the main one**
 - Convolution provides **translation equivariance**: if the input shifts, the feature map shifts in the same way.
 - But equivariance \neq invariance. It detects the shift but doesn't ignore it.
- **c) Pooling →  Correct**
 - Pooling (e.g., max pooling, average pooling) summarizes features within a region.
 - This reduces sensitivity to small shifts or translations in the input.
 - Example: if an edge shifts by 1 pixel, max pooling still detects it, making the network more **translation invariant**.
- **d) BatchNorm → Incorrect**
 - Batch Normalization stabilizes training by normalizing activations.
 - It has nothing to do with translation invariance.

👉 Key distinction:

- **Convolution → translation *equivariance*** (output shifts when input shifts).
 - **Pooling → translation *invariance*** (ignores small shifts).
-


 If a CNN uses stride > 1 in convolution, the effect is:


Options:

- a) Increased number of filters
- b) Spatial downsampling


- c) Reduced receptive field
 - d) More overlapping patches
-

Answer Analysis

- **a) Increased number of filters → Incorrect**
 - The number of filters is set by the model designer (hyperparameter).
 - Stride doesn't change how many filters are used—it changes *how* they slide over the input.
- **b) Spatial downsampling →  Correct**
 - With stride > 1 , the filter moves in bigger steps.
 - This means fewer positions are covered, so the output feature map is smaller.
 - Effectively, stride > 1 = **downsampling** (similar to pooling).
- **c) Reduced receptive field → Incorrect**
 - Stride doesn't shrink the receptive field. In fact, stacking strided convolutions often *increases* the receptive field because each feature covers more of the input.
- **d) More overlapping patches → Incorrect**
 - Stride > 1 → *less overlap* between receptive fields, since the filter skips positions.
 - Stride = 1 → maximum overlap.

 Quick intuition:


- **Stride = 1 → detailed scan, overlapping patches**
 - **Stride > 1 → coarse scan, fewer outputs → smaller feature map**
-

 Dilated convolutions are used primarily for:

Options:

- a) Increasing receptive field without increasing parameters
 - b) Avoiding vanishing gradients
 - c) Reducing FLOPs
 - d) Increasing overfitting
-

Answer Analysis

- **a) Increasing receptive field without increasing parameters →  Correct**
 - Dilated (or atrous) convolutions insert gaps between kernel elements.
 - This lets the filter “see” a larger context (bigger receptive field) without adding more weights.
 - Example: a 3×3 kernel with dilation=2 covers a 5×5 effective region, but still has only 9 parameters.
 - **b) Avoiding vanishing gradients → Incorrect**
 - Vanishing gradients are about activation functions (sigmoid/tanh) and deep backpropagation.
 - Dilated convolutions don’t address this issue.
 - **c) Reducing FLOPs → Incorrect**
 - FLOPs depend mostly on kernel size, input size, and number of filters.
 - Dilated convolutions may even *increase* FLOPs since receptive fields are larger, though the parameter count stays the same.
 - **d) Increasing overfitting → Incorrect**
 - Overfitting is about how well a model generalizes.
 - Dilated convolutions don’t inherently increase overfitting—they just change spatial coverage.
-

⚡ Quick intuition:

- Standard convolution = local vision 


- Dilated convolution = "zoomed-out" vision (larger context, same parameter budget).

? Which activation function is most prone to the "dying neuron" problem?

Options:

- a) ReLU
- b) Leaky ReLU
- c) ELU
- d) Softmax

Answer Analysis

- **a) ReLU →  Correct**
 - In ReLU, if the input is negative, output = 0.
 - If weights update in such a way that the neuron keeps producing negative inputs, it gets stuck always outputting 0.
 - Gradient also becomes 0 → neuron is "dead" (won't learn anymore).
 - This is the classic *dying ReLU problem*.
- **b) Leaky ReLU → Incorrect**
 - Leaky ReLU fixes dying neurons by allowing a small slope (e.g., 0.01x) for negative inputs.
 - Neurons don't completely die because gradient is never exactly 0.
- **c) ELU → Incorrect**
 - Exponential Linear Units also keep a small non-zero gradient in the negative region.
 - They reduce the dying neuron problem.
- **d) Softmax → Incorrect**

- Softmax is used at the output layer for classification.
- It doesn't suffer from "dying neuron" since it's not a hidden layer activation function in the same sense.

👉 Quick memory hook:


- **ReLU** = risk of dying neurons
- **Leaky ReLU / ELU** = fixes dying neurons
- **Sigmoid / Tanh** = vanishing gradient issue

? Why is Softmax often used in the output layer of classification networks?

Options:

- a) It introduces non-linearity
- b) It outputs probabilities summing to 1
- c) It avoids vanishing gradients
- d) It reduces dimensionality

Answer Analysis

- **a) It introduces non-linearity → Incorrect**
 - True, softmax is nonlinear, but that's not the main reason we use it at the output.
 - Many other activations are nonlinear too, but they don't serve the probability interpretation.
- **b) It outputs probabilities summing to 1 →  Correct**
 - Softmax converts raw logits (any real values) into a probability distribution.
 - Each output is in the range $[0, 1]$, and all outputs sum to 1.
 - This makes it perfect for multi-class classification.
- **c) It avoids vanishing gradients → Incorrect**

- Vanishing gradients is mostly an issue with sigmoid/tanh in hidden layers.
- Softmax doesn't solve this problem—it's used for interpretability of outputs.
- **d) It reduces dimensionality → Incorrect**
 - Softmax keeps the same number of outputs as the number of classes.
 - It doesn't reduce dimensions, it just transforms logits into probabilities.

⚡ Quick intuition:


- **Hidden layers** → extract features
- **Output layer (softmax)** → turn features into class probabilities

? Swish activation function is defined as:

Options:

- a) $x \cdot \tanh(x)$
- b) $x \cdot \sigma(x)$
- c) $\max(0, x)$
- d) $\sigma(x)(1 - \sigma(x))$

Answer Analysis

- **a) $x \cdot \tanh(x)$ → Incorrect**
 - This looks like a custom activation, but it's not Swish.
 - Tanh squashes values to $[-1,1]$, so this is different from Swish.
- **b) $x \cdot \sigma(x)$ →  Correct**
 - Swish is defined as:

$$f(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$$

- It's smooth, non-monotonic, and often outperforms ReLU in deep networks.
- **c) $\max(0, x) \rightarrow$ Incorrect**
 - That's ReLU, not Swish.
- **d) $\sigma(x)(1 - \sigma(x)) \rightarrow$ Incorrect**
 - That's the derivative of the sigmoid function, not an activation function itself.

👉 Quick memory tip:


- **ReLU = $\max(0, x)$**
- **Swish = $x \cdot \text{sigmoid}(x)$**
- **Mish = $x \cdot \tanh(\text{softplus}(x))$**

? Batch Normalization primarily helps by:

Options:

- a) Making gradients vanish slower
- b) Reducing internal covariate shift
- c) Increasing dataset size artificially
- d) Eliminating need for activation functions

Answer Analysis

- **a) Making gradients vanish slower \rightarrow Incorrect**
 - BatchNorm *can* help stabilize gradients indirectly, but that's not its *primary purpose*.
 - The main benefit is controlling the distribution of activations, not directly fixing vanishing gradients.
- **b) Reducing internal covariate shift \rightarrow  Correct**

- Internal covariate shift = change in distribution of layer inputs during training as parameters update.
- BatchNorm normalizes activations (mean ~ 0 , variance ~ 1) within a batch, keeping distributions stable.
- This speeds up training, allows higher learning rates, and adds a bit of regularization.
- **c) Increasing dataset size artificially → Incorrect**
 - That's **data augmentation**, not BatchNorm.
- **d) Eliminating need for activation functions → Incorrect**
 - BatchNorm works *with* activations (e.g., ReLU, Swish).
 - It doesn't replace them.

⚡ Extra note:

- BatchNorm also smooths optimization, reduces sensitivity to initialization, and sometimes acts as a mild regularizer.




Dropout works by:

Options:

- a) Randomly removing weights
- b) Randomly setting some neuron outputs to zero
- c) Reducing learning rate dynamically
- d) Clipping gradient values

Answer Analysis

- **a) Randomly removing weights → Incorrect**
 - Dropout doesn't delete weights from the network.

- It only ignores certain neurons *temporarily* during training, not permanently removing weights.
- **b) Randomly setting some neuron outputs to zero →  Correct**
 - During training, Dropout randomly “drops” some neuron activations (sets them to 0).
 - This prevents neurons from co-adapting too much, forcing the network to learn more robust features.
 - At inference time, all neurons are used but their outputs are scaled (or equivalently, dropout is turned off).
- **c) Reducing learning rate dynamically → Incorrect**
 - That’s **learning rate scheduling**, not dropout.
- **d) Clipping gradient values → Incorrect**
 - That’s **gradient clipping**, used to prevent exploding gradients.
 - Not related to dropout.

👉 Quick mental model:

- Dropout = training an **ensemble of thinned networks** inside one big network.


 Which optimizer adapts learning rate for each parameter individually?

Options:

- a) SGD
- b) Momentum
- c) Adam
- d) NAG


Answer Analysis

- **a) SGD → Incorrect**

- Standard Stochastic Gradient Descent uses a fixed global learning rate for all parameters.
- It does not adapt per-parameter learning rates.
- **b) Momentum → Incorrect**
 - Momentum adds an exponentially decaying average of past gradients to accelerate learning in the right direction.
 - But the learning rate is still global (same for all parameters).
- **c) Adam →  Correct**
 - Adam = Adaptive Moment Estimation.
 - It combines ideas from **Momentum** (moving average of gradients) and **RMSProp** (adaptive learning rates per parameter).
 - Each parameter gets its own learning rate, adjusted based on its gradient history.
- **d) NAG (Nesterov Accelerated Gradient) → Incorrect**
 - NAG is a refinement of Momentum that looks ahead before updating.
 - Still uses a global learning rate, not per-parameter adaptation.

 Quick memory hook:


- **SGD & Momentum & NAG → one global LR**
- **Adam, RMSProp, Adagrad → per-parameter adaptive LR**

 Depthwise separable convolutions, as used in MobileNet, reduce:

Options:

- a) FLOPs and parameters
- b) Training dataset requirements
- c) Receptive field size
- d) Gradient vanishing

Answer Analysis

- **a) FLOPs and parameters →  Correct**
 - Depthwise separable convolution splits a standard convolution into:
 1. **Depthwise convolution** → applies one filter per channel (no mixing across channels).
 2. **Pointwise convolution (1×1)** → combines information across channels.
 - This dramatically reduces **computational cost (FLOPs)** and **number of parameters** while keeping performance competitive.
- **b) Training dataset requirements → Incorrect**
 - Dataset requirements don't change due to convolution type.
 - Small models might need less data to avoid overfitting, but that's not the main design motivation.
- **c) Receptive field size → Incorrect**
 - Receptive field depends on kernel size, stride, and depth of layers.
 - Depthwise separable convolution does not shrink the receptive field—it keeps it the same as standard convolution.
- **d) Gradient vanishing → Incorrect**
 - Gradient vanishing is about deep activations (sigmoid/tanh), not convolution design.
 - Depthwise separable convs don't fix vanishing gradients.

 Quick intuition:

- **Standard conv ($k \times k$, $M \rightarrow N$ channels)** = expensive ($k^2 \times M \times N$ params).
 - **Depthwise separable conv** = depthwise ($k^2 \times M$) + pointwise ($M \times N$).
 - Huge savings in parameters & FLOPs → why MobileNet runs efficiently on mobile devices.
-




Residual connections in ResNet help primarily with:

Options:

- a) Reducing training data need
- b) Avoiding vanishing gradients
- c) Increasing FLOPs
- d) Enforcing weight sharing

Answer Analysis

- **a) Reducing training data need → Incorrect**
 - Residual connections don't reduce dataset requirements.
 - They just make training very deep networks possible and stable.
- **b) Avoiding vanishing gradients →  Correct**
 - The skip (identity) connections allow gradients to flow directly back through earlier layers.
 - This bypasses the problem where gradients shrink as they pass through many nonlinear layers (vanishing gradient).
 - This is the *primary reason* why ResNet enables training of 50+, 100+, even 1000+ layer networks.
- **c) Increasing FLOPs → Incorrect**
 - Residual connections actually add *negligible* computation (just element-wise addition).
 - They don't increase FLOPs significantly.
- **d) Enforcing weight sharing → Incorrect**
 - Weight sharing happens in CNN filters (e.g., convolution kernels), not because of residual connections.



Quick recap:

- **Main problem in deep nets before ResNet** → vanishing gradients.
- **ResNet solution** → identity skip connections → easy gradient flow.



Consider the following Code Snippet. What will be the output of the program?

```
import torch
import numpy as np

a = torch.randn(size=(2,2,2,3,4), dtype=torch.float64)
b = torch.randn(size=(2,2,3,4,3), dtype=torch.float64)
c = torch.matmul(a, b)
print(c.shape)
```

Step 1: Shape of inputs

- **a** has shape **(2, 2, 2, 3, 4)**
- **b** has shape **(2, 2, 3, 4, 3)**

Think of `torch.matmul` as batched matrix multiplication:

- The last two dimensions are treated as matrix dimensions.
- All earlier dimensions are batch dimensions and must broadcast.

Step 2: Last two dimensions (matrix multiply)

- For **a** → last two dims: **(3, 4)**
- For **b** → last two dims: **(4, 3)**

👉 These are compatible because inner dimensions match ($4 = 4$).

- Resulting matrix shape = **(3, 3)**

Step 3: Batch dimensions

- `a` has batch dims: **(2, 2, 2)**
- `b` has batch dims: **(2, 2, 3)**

Now, PyTorch applies **broadcasting rules**:

- Compare batch dims from right to left:
 - `a`: (2, 2, 2)
 - `b`: (2, 2, 3)
 - Last dim: 2 (from `a`) vs 3 (from `b`) → **✗** mismatch, not broadcastable.

Step 4: Runtime check

This means the code will actually raise a **RuntimeError** instead of producing an output shape.

The error will be something like:

```
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 2
```

✅ Final Answer

The program will **not print a shape**.

It will throw a **RuntimeError due to incompatible batch dimensions**.

? You instantiate the SimpleCNN model from the tutorial with depth = 4. What is the value of in_channels for the last convolutional layer that is created inside the loop?

```
class SimpleCNN(nn.Module):
    def __init__(self, activation='relu', depth=2):
        super(SimpleCNN, self).__init__()
        # ... (rest of the init method) ...
        layers = []
        in_channels = 1
        for i in range(depth):
            out_channels = 7 * (3**i)
            layers.append(nn.Conv2d(in_channels, out_channels, ...))
            # ...
            in_channels = out_channels
```

Code in focus

```
in_channels = 1
for i in range(depth):
    out_channels = 7 * (3**i)
    layers.append(nn.Conv2d(in_channels, out_channels, ...))
    in_channels = out_channels
```

You set `depth = 4`.

Step-by-step

- **Before loop:**

```
in_channels = 1
```

Iteration 0 (`i=0`):

- `out_channels = 7 * (3**0) = 7 * 1 = 7`

- Conv2d created with `(in=1, out=7)`
 - Update: `in_channels = 7`
-

Iteration 1 (`i=1`):

- `out_channels = 7 * (3**1) = 21`
 - Conv2d created with `(in=7, out=21)`
 - Update: `in_channels = 21`
-

Iteration 2 (`i=2`):

- `out_channels = 7 * (3**2) = 63`
 - Conv2d created with `(in=21, out=63)`
 - Update: `in_channels = 63`
-

Iteration 3 (`i=3`):

- `out_channels = 7 * (3**3) = 189`
 - Conv2d created with `(in=63, out=189)`
 - Update: `in_channels = 189`
-

✅ Final Answer

For `depth = 4`, the **last convolutional layer** is created with:

- `in_channels = 63`
 - `out_channels = 189`
-

⚡ So the value of `in_channels` for the last conv layer = 63



You train a CNN on an image dataset with the following code:

```
# training
model.train()
for epoch in range(n):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
# evaluation
model.eval()
@torch.no_grad()
def get_accuracy(correct = 0, total = 0):
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = outputs.max(1)
        correct += (predicted == labels).sum().item()
#... Rest of the code.
```

Even for a large n , the model gives very low accuracy. Identify the reason and suggest changes.



The Problem

Your **training loop** is:

```
model.train()
for epoch in range(n):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
```

Notice what's missing? 📌

- You compute `loss`
- You reset the gradients with `optimizer.zero_grad()`
- ❌ But you never call `loss.backward()`
- ❌ And you never call `optimizer.step()`

That means:

- **No gradient is ever calculated**
- **Weights are never updated**
- The model just stays at its initial random parameters → hence accuracy stays very low no matter how long you train.

✅ Corrected Training Loop

```
model.train()
for epoch in range(n):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad() # 1. Reset gradients
        loss.backward()       # 2. Backpropagate loss
        optimizer.step()      # 3. Update weights
```

🔍 Evaluation Loop

Your evaluation part is mostly fine:

```
model.eval()
@torch.no_grad()
def get_accuracy(correct=0, total=0):
    for images, labels in test_loader:
        outputs = model(images)
```

```
_, predicted = outputs.max(1)
correct += (predicted == labels).sum().item()
total += labels.size(0)
return correct / total
```

✓ The only small thing missing is updating `total` → otherwise accuracy is `correct / 0`.

✨ Final Answer

- **Reason for low accuracy:** The model never updates because `loss.backward()` and `optimizer.step()` are missing in the training loop.
- **Fix:** Add them inside the loop:

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

? You are given the CIFAR-10 dataset and a batch size of 128 images. What will be the output of the following snippet?

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
print(images.shape)
```

🧩 Given

- Dataset: **CIFAR-10**
- Each image: `32 × 32 × 3` (RGB, 3 channels)
- Batch size = `128`

What happens in `DataLoader`

When you do:

```
images, labels = dataiter.next()
```

- `images` → A batch of 128 images stacked together
- `labels` → The corresponding 128 labels

But careful: PyTorch stores images in channel-first format

- Not `(32, 32, 3)` like NumPy/TensorFlow
- But `(3, 32, 32)` → `[channels, height, width]`

So final shape

- Batch dimension: `128`
- Channels: `3`
- Height: `32`
- Width: `32`

```
images.shape = torch.Size([128, 3, 32, 32])
```

Answer

```
torch.Size([128, 3, 32, 32])
```

? Consider the following code snippet:

```
import torch
import torch.nn as nn
conv = nn.Conv3d(
    in_channels = 3,
    out_channels=64,
    kernel_size = (1, 3, 3),
    stride = (1, 1, 1),
    bias = False
)
```

Conceptually, along the temporal axis, this layer is equivalent to:

- (a) A 2D conv run on each frame separately (same weights for all frames)
- (b) A 1D conv over time
- (c) A 2D conv on each frame with different weights at each time step
- (d) A per-channel (depthwise) conv only, with no channel mixing

Step 1. What does `Conv3d` expect?

Input shape: `(N, C_in, D, H, W)`

- `N`: batch
- `C_in`: input channels (e.g. RGB = 3)
- `D`: depth = time (frames in a video)
- `H, W`: height, width

Step 2. Kernel shape





`kernel_size = (1, 3, 3)`

- Temporal size = **1**
- Spatial size = **3×3**

So the filter:

- Looks at **1 frame at a time** (no temporal extent)
 - Applies a **2D convolution over $H \times W$**
 - Same weights are applied to every time slice because temporal kernel = 1.
-

Step 3. Check the options

- **(a) A 2D conv run on each frame separately (same weights for all frames) →**
 Correct
 - Kernel size `1` in time means it does not mix across frames.
 - Weights are shared across frames → same conv applied to each.
 - **(b) A 1D conv over time →  Wrong**
 - Would require `kernel_size > 1` along time to aggregate across frames.
 - **(c) A 2D conv on each frame with different weights at each time step →  Wrong**
 - That would imply *non-shared weights per time slice*, which is not how conv layers work.
 - **(d) A per-channel (depthwise) conv only, with no channel mixing →  Wrong**
 - Here `in_channels=3 → out_channels=64`, meaning weights *do* mix RGB channels. Depthwise conv would require `groups=in_channels`.
-

Final Answer

(a) A 2D conv run on each frame separately (same weights for all frames)
