# Practice questions

❓ Why is weight initialization important in neural networks?

Weight initialization is important because it breaks symmetry and ensures activations/gradients stay in stable ranges. Poor initialization can cause vanishing or exploding gradients, while good schemes like Xavier or He initialization allow faster and more reliable training.

## Interview POV

**Difficulty:** Medium (fundamentals check).

- If you say *"initialization prevents vanishing/exploding gradients and speeds up learning"* → that's the safe short answer.
- If you can also mention *Xavier vs He initialization* → that's a strong, practical answer.

## Why Weight Initialization Matters

When we train a neural network, we start with **initial weights** before backpropagation updates them.

- If initialization is poor, training may fail or become very slow.
- Good initialization ensures stable gradients and faster convergence.

## Problems with Bad Initialization

1. **All weights = 0**
    - Every neuron computes the same thing → symmetry.
    - No diversity → network can't learn.

2. **Weights too small**
    - Activations shrink → gradients vanish.
    - Deeper layers barely update.
3. **Weights too large**
    - Activations explode → gradients blow up.
    - Leads to unstable training (loss = NaN).

---

## Benefits of Proper Initialization

- **Breaks symmetry** → neurons learn different features.
- **Controls variance** → keeps activations and gradients in reasonable ranges across layers.
- **Faster convergence** → optimizer reaches good solutions quickly.

---

## Common Initialization Schemes

- **Xavier/Glorot Initialization** (for Sigmoid/Tanh)
    - Keeps the variance of activations the same across layers.
- **He Initialization (Kaiming)** (for ReLU)
    - Accounts for ReLU dropping negative activations.
- **Orthogonal Initialization**
    - Ensures weight matrices are well-conditioned (useful in RNNs).

---

❓ In backpropagation, why do we compute gradients with respect to weights instead of directly adjusting outputs?

We compute gradients with respect to weights because weights are the learnable parameters that define the function mapping inputs to outputs. Outputs aren't independent variables — they're determined by the weights and inputs. If we

adjusted outputs directly, it would only fix the current batch without changing the underlying model. By updating weights using gradients, we ensure the network learns a better mapping that generalizes to new data.

---

❓ Explain the role of padding in convolutional neural networks. What happens to the feature map size if we use ”valid” vs. ”same” padding?

Padding lets us control whether the network shrinks feature maps (‘valid’) or preserves size (‘same’). In practice, ‘same’ is often used to build deep CNNs, while ‘valid’ is used when we deliberately want dimension reduction.

## Valid vs. Same Padding

- **Valid Padding ("no padding")**
  - No extra pixels added.
  - Output feature map shrinks:

    $$\text{Output size} = \frac{(W-K)}{\text{stride}} + 1$$

    (where W = input size, K = kernel size).
  - **Edges lose coverage.**
- **Same Padding ("zero padding")**
  - Enough padding is added so that the output spatial size ≈ input size (when stride=1).
  - Keeps dimensions constant across layers.
  - Makes it easier to stack many conv layers without shrinking.

❓ Why are 1 × 1 convolutions used in CNN architectures (e.g., GoogleNet)? Mention two benefits.

1×1 convolutions are mainly used for reducing channel dimensions to make deeper networks computationally efficient, and for enabling non-linear combinations of

feature maps across channels. That's why they were a key building block in GoogleNet's Inception modules.

❓ BatchNorm introduces two learnable parameters (γ,β). What is their purpose?

γ and β in BatchNorm let the model undo the strict normalization if needed: γ allows the network to restore any necessary variance (e.g., undo the unit variance constraint if the task needs higher/lower variance), and β shifts them, i.e., allows the network to restore non-zero mean activations if needed. This keeps the normalization benefits while preserving the network's flexibility.

❓ Explain how dropout acts as a form of ensemble learning.

Dropout acts like an ensemble because each mini-batch sees a different subnetwork (due to the random dropping of neurons). Effectively, it trains many subnetworks in parallel and, at test time, combines them by scaling weights. This approximates the effect of averaging an ensemble of models, improving generalization.

❓ Give one example of a real-world problem where an RNN is preferred over a feed-forward network, and explain why.

A good real-world example is next-word prediction in text. Here, the current output depends on a sequence of previous words. An RNN is preferred over a feedforward network because it maintains hidden states that capture temporal dependencies, letting the model 'remember' past context, which a plain feedforward network cannot do.

❓ What is the function of the input gate in an LSTM? How does it differ from the forget gate?

The forget gate decides how much of the previous long-term memory to keep or discard, while the input gate controls how much of the new information should be added to the long-term memory. Together, they regulate the updates to the cell state at each time step.

❓ Why is training GANs unstable? Mention two challenges faced during optimization.

GAN training is unstable because the generator and discriminator are in a dynamic min-max game, rather than simple supervised learning. Two common challenges are: 1) **mode collapse**, where the generator produces low-diversity outputs, and 2) **vanishing gradients**, where a strong discriminator gives almost no learning signal to the generator.

❓ What potential issue might occur with the gradients during training?
(a)The ReLU activation function may cause vanishing gradients in deeper layers
(b)The network may encounter exploding gradients due to high weight initialization
(c)The code will work fine with no gradient issues
(d)None of the above

```
#Consider the following snippet:
import torchimport torch.nn as nn
import torch.optim as optim

class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(10, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 1)
    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    model = SimpleNet()
    optimizer = optim.SGD(model.parameters(), lr=0.01)
    input_data = torch.randn(1, 10)
    output = model(input_data)
    optimizer.zero_grad()
    output.backward()
    optimizer.step()
```

(c) Code will work fine with no gradient issues

❓ During inference, for a particular input, calling sample(start_token, max_len=20), produces the output sentence: It is raining cats and dogs today.
What will be the output if you call sample(start_token)?

```
#Consider the following inference code for a trained RNN model:
def sample(self, start_token, max_len=5):
    sampled_ids = []
    inputs = torch.tensor([start_token]).unsqueeze(0)
    inputs = self.embed(inputs)
    h = torch.zeros(self.num_layers, 1, self.hidden_size).to(inputs.device)
    for _ in range(max_len):
        outputs, h = self.rnn(inputs, h)
        outputs = self.linear(outputs.squeeze(1))
        _, predicted = outputs.max(1)
        sampled_ids.append(predicted.item())
        # Stop if <end> token is predicted (id=2)
        if predicted.item() == 2:
            break
```

```
        inputs = self.embed(predicted.unsqueeze(0)).unsqueeze(1)
    return sampled_ids
```

When `max_len` is not provided, it defaults to 5. The RNN will generate at most 5 tokens starting from `start_token`. So the output sentence will be truncated to the first 5 tokens:
`"It is raining cats and"`

❓ The model below doesn't train well. Highlight the potential issue with respect to gradients and suggest potential fixes.

```
#Consider the following code snippet:
import torch
import torch.nn as nn
import torch.optim as optim

class DeepNet(nn.Module):
    def __init__(self):
        super(DeepNet, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(10, 10) for _ in range(50)])
        self.fc_out = nn.Linear(10, 1)

    def forward(self, x):
        for layer in self.layers:
            x = torch.relu(layer(x))
        x = self.fc_out(x)
        return x

model = DeepNet()
input_data = torch.randn(1, 10)
output = model(input_data)
output.backward()
```

The network has two main issues: (1) it's very deep, so gradients may vanish, preventing early layers from learning — this can be fixed with residual connections, BatchNorm, or proper initialization; (2) no optimizer is used, so even though gradients are computed, weights are never updated — adding an optimizer like Adam or SGD and calling `optimizer.step()` solves this.

❓ What makes a GAN "conditional," and how are labels injected? Write a code snippet for the same.

A GAN is conditional when both the generator and discriminator receive **extra information like class labels**. Labels are injected by concatenating a one-hot or embedded vector with the noise vector in the generator and with the input in the discriminator.

```
import torch
import torch.nn as nn

class ConditionalGenerator(nn.Module):
    def __init__(self, noise_dim=100, label_dim=10, output_dim=28*28):
        super().__init__()
        input_dim = noise_dim + label_dim
        self.net = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim),
            nn.Tanh()
        )

    def forward(self, noise, labels):
        # Assume labels are one-hot encoded
        x = torch.cat([noise, labels], dim=1)
        return self.net(x)
```

```
class ConditionalDiscriminator(nn.Module):
    def __init__(self, input_dim=28*28, label_dim=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim + label_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img, labels):
        x = torch.cat([img, labels], dim=1)
        return self.net(x)

# Example usage
batch_size = 16
noise = torch.randn(batch_size, 100)
labels = torch.zeros(batch_size, 10)
labels[range(batch_size), torch.randint(0, 10, (batch_size,))] = 1  # one-hot
G = ConditionalGenerator()
D = ConditionalDiscriminator()
fake_images = G(noise, labels)
pred = D(fake_images, labels)
```

```
# Initialize Generator G and Discriminator D
G = Generator()
D = Discriminator()

# Training loop
for each training step:

    # ----------------------
    # 1. Train Discriminator
```

```
# ----------------------
real_images, labels = get_real_batch()       # labels: one-hot or integer
noise = random_noise(batch_size)

# Generate fake images conditioned on labels
fake_images = G(noise, labels)

# Compute discriminator loss
D_loss = -[log(D(real_images, labels)) + log(1 - D(fake_images, labels))]

# Backprop & update D
D.zero_grad()
D_loss.backward()
optimizer_D.step()

# ----------------------
# 2. Train Generator
# ----------------------
noise = random_noise(batch_size)
fake_images = G(noise, labels)              # conditioned on same labels

# Generator loss: try to fool D
G_loss = -log(D(fake_images, labels))

# Backprop & update G
G.zero_grad()
G_loss.backward()
optimizer_G.step()
```