Question 1: You're implementing a Random Forest classifier and notice that increasing n_estimators from 100 to 1000 improves training accuracy from 99.2% to 99.8% but validation accuracy remains at 87.3%. The training time increased 10×. What does this indicate?

- The model is underfitting - need more complex base estimators (deeper trees)
- ***Diminishing returns from ensemble size - the 100-tree model already captures ensemble diversity; additional trees only fit training noise without improving generalisation***
- Random Forest cannot overfit regardless of n_estimators due to bagging's variance reduction
- The validation set is too small to show improvement - collect more validation data

==========

Question 2

Implementing gradient boosting from scratch, you write:

```python
def gradient_boosting_regressor(X, y, n_estimators=100, learning_rate=0.1):
    F = np.zeros(len(y))  # Initial prediction
    models = []

    for i in range(n_estimators):
        residuals = y - F  # Line A
        tree = DecisionTreeRegressor(max_depth=3)
        tree.fit(X, residuals)  # Line B
        models.append(tree)
        F += learning_rate * tree.predict(X)  # Line C

    return models
```

For L2 loss (MSE), this implementation is correct. If you switch to L1 loss (MAE), which line(s) must change?

- ***Line A only - residuals should be sign(y - F) for L1 loss, representing the negative gradient of absolute error***
- Lines A and B - residuals become sign(y - F), and tree should fit median of residuals in each leaf instead of mean
- Line C only - learning rate should adapt based on line search to find optimal step size for L1 loss
- All lines must change - L1 loss requires completely different algorithm structure

==========

Question 3

You implement k-fold cross-validation with feature selection:

```python
# Approach A
selector = SelectKBest(f_classif, k=10)
X_selected = selector.fit_transform(X, y)  # Select on ALL data
scores = cross_val_score(model, X_selected, y, cv=5)
# Approach B
def cv_with_selection(X, y, model, k_features=10):
    kfold = KFold(n_splits=5, shuffle=True, random_state=42)
    scores = []
    for train_idx, val_idx in kfold.split(X):
        selector = SelectKBest(f_classif, k=k_features)
        X_train_selected = selector.fit_transform(X[train_idx], y[train_idx])
        X_val_selected = selector.transform(X[val_idx])
        model.fit(X_train_selected, y[train_idx])
        scores.append(model.score(X_val_selected, y[val_idx]))
```

```
    return np.mean(scores)
```

Approach A reports 92% CV accuracy. Approach B reports 78% CV accuracy. Which is the reliable estimate?

- Approach A - fitting selector on all data provides more stable feature selection; Approach B has higher variance due to different features selected per fold
- ***Approach B - feature selection must be inside CV loop to prevent data leakage; Approach A's 92% is optimistically biased because features were selected using validation fold information***
- Both are valid but measure different things - A measures performance with optimal features, B measures realistic deployment performance
- Neither - feature selection should use a separate held-out set before any cross-validation

==========

Question 4

Implementing logistic regression with L2 regularisation, you compute the gradient:

```python
def compute_gradient(X, y, weights, lambda_reg):
    m = len(y)
    predictions = sigmoid(X @ weights)

    # Option A
    gradient = (1/m) * X.T @ (predictions - y) + lambda_reg * weights

    # Option B
    gradient = (1/m) * X.T @ (predictions - y) + (lambda_reg/m) * weights

    # Option C
    gradient = (1/m) * X.T @ (predictions - y)
```

```
    gradient[1:] += lambda_reg * weights[1:]  # Don't regularise bias

    # Option D

    gradient = (1/m) * X.T @ (predictions - y)

    gradient[1:] += (lambda_reg/m) * weights[1:]  # Don't regularise bias


    return gradient
```

Which option follows scikit-learn's LogisticRegression(penalty='l2') convention?

- Option A - standard L2 regularisation adds λw to gradient
- Option B - regularisation term should be scaled by 1/m for consistency with loss averaging
- Option C - bias term (weights[0]) should not be regularised, but λ is not scaled by m
- ***Option D - bias not regularised AND λ scaled by m, matching sklearn's C parameter (C = 1/λm)***

==========

Question 5

You're debugging a custom implementation of AdaBoost:

```python
def adaboost(X, y, n_estimators=50):
    n_samples = len(y)
    weights = np.ones(n_samples) / n_samples
    models, alphas = [], []


    for _ in range(n_estimators):
        tree = DecisionTreeClassifier(max_depth=1)
        tree.fit(X, y, sample_weight=weights)
```

```
        predictions = tree.predict(X)


        incorrect = (predictions != y)
        error = np.sum(weights * incorrect) / np.sum(weights)  # Line A


        if error > 0.5:  # Line B
            break


        alpha = 0.5 * np.log((1 - error) / (error + 1e-10))  # Line C
        weights *= np.exp(alpha * incorrect)  # Line D
        weights /= np.sum(weights)


        models.append(tree)
        alphas.append(alpha)


    return models, alphas
```

The model performs poorly. What's wrong?

- Error threshold in Line B should be >= 0.5, not > 0.5, to handle the edge case of random classifier
- The alpha calculation is wrong - should be np.log((1 - error) / error) without the 0.5 factor
- ***Weight update should be: weights *= np.exp(alpha * (2*incorrect - 1)), because correctly classified samples should have weights DECREASED (multiplied by exp(-alpha))***
- Weight update should use: weights *= np.exp(-alpha * incorrect), because incorrect samples should have decreased weights

==========

Question 6

You implement a custom loss function for multi-label classification:

```python
class MultiLabelLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, logits, targets):
        # logits: (batch, num_classes), raw scores
        # targets: (batch, num_classes), binary labels {0, 1}

        # Option A
        probs = torch.sigmoid(logits)
        loss = -targets * torch.log(probs) - (1-targets) * torch.log(1-probs)
        return loss.mean()

        # Option B
        loss = F.binary_cross_entropy_with_logits(logits, targets, reduction='mean')
        return loss

        # Option C
        probs = torch.softmax(logits, dim=1)
        loss = -targets * torch.log(probs)
```

```
        return loss.sum(dim=1).mean()
```

Which implementation is correct for multi-label classification?

- Option A - manual BCE implementation with sigmoid, mathematically correct but numerically unstable for extreme logits
- Option B - PyTorch's built-in function handles numerical stability via log-sum-exp trick
- Option C - softmax ensures probabilities sum to 1, which is required for classification
- ***Options A and B are equivalent and both correct; Option C is wrong because softmax assumes mutually exclusive classes***

==========

Question 7

Training a deep network, you observe gradient norms across layers:

Epoch 1:

Layer 1 (input): grad_norm = 0.0003 Layer 5 (middle): grad_norm = 0.0012 Layer 10 (output): grad_norm = 0.8500

Epoch 50: Layer 1 (input): grad_norm = 0.0001 Layer 5 (middle): grad_norm = 0.0008 Layer 10 (output): grad_norm = 1.2300

What problem does this indicate and which solution is most effective?

- Learning rate too high for early layers - use layer-wise learning rate decay (higher LR for early layers)
- Batch normalisation statistics unstable - increase batch size or use group normalisation
- ***Vanishing gradients in early layers - add residual/skip connections to provide gradient highways bypassing intermediate layers***
- Exploding gradients in output layer - apply gradient clipping with max_norm=1.0 to stabilise training

==========

Question 8

You implement custom batch normalisation:

```python

class BatchNorm1d(nn.Module):
```

```
  def __init__(self, num_features, eps=1e-5, momentum=0.1):
      super().__init__()
      self.gamma = nn.Parameter(torch.ones(num_features))
      self.beta = nn.Parameter(torch.zeros(num_features))
      self.register_buffer('running_mean', torch.zeros(num_features))
      self.register_buffer('running_var', torch.ones(num_features))
      self.eps = eps
      self.momentum = momentum

  def forward(self, x):
      if self.training:
          mean = x.mean(dim=0)
          var = x.var(dim=0, unbiased=False)

          # Update running statistics
          self.running_mean = (1 - self.momentum) * self.running_mean + self.momentum
* mean
          self.running_var = (1 - self.momentum) * self.running_var + self.momentum * var

          x_norm = (x - mean) / torch.sqrt(var + self.eps)
      else:
          x_norm = (x - self.running_mean) / torch.sqrt(self.running_var + self.eps)

      return self.gamma * x_norm + self.beta
```

```

During training, loss converges well. During evaluation (model.eval()), predictions are poor. What's the bug?

- Running statistics update should use .detach() to prevent gradient flow: `self.momentum * mean.detach()`
- Running statistics should be updated with unbiased variance (unbiased=True) to match evaluation behavior
- ***The running statistics update breaks autograd - should use in-place operations: `self.running_mean.mul_(1-self.momentum).add_(self.momentum * mean)`***
- Momentum interpretation is inverted - PyTorch uses running_mean = momentum * running_mean + (1-momentum) * batch_mean

==========
Question 9

Implementing learning rate warmup with cosine decay:

```python

def get_lr(step, total_steps, warmup_steps, base_lr, min_lr=1e-6):

    if step < warmup_steps:

        # Linear warmup

        return base_lr * step / warmup_steps

    else:

        # Cosine decay

        progress = (step - warmup_steps) / (total_steps - warmup_steps)

        return min_lr + 0.5 * (base_lr - min_lr) * (1 + math.cos(math.pi * progress))

```

Training a transformer with base_lr=1e-3, warmup_steps=1000, total_steps=100000. At step 50000, what is the learning rate?

- ~7.5e-4 (cosine has decayed about 25% from peak)
- ***~5.0e-4 (cosine at midpoint, halfway between base_lr and min_lr)***
- ~2.5e-4 (cosine has decayed about 75% from peak)
```

- ~1e-3 (still at peak learning rate)

==========

Question 10

You implement gradient accumulation for training with limited GPU memory:

```python
accumulation_steps = 4

optimiser.zero_grad()

for i, (inputs, targets) in enumerate(dataloader):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss = loss / accumulation_steps  # Line A
    loss.backward()


    if (i + 1) % accumulation_steps == 0:
        optimiser.step()
        optimiser.zero_grad()
```

Why is Line A necessary?

- ***Dividing loss scales gradients to match effective batch size - without it, accumulated gradients would be 4× larger than training with actual batch size 4×***
- Dividing loss prevents numerical overflow when accumulating gradients across multiple backward passes
- Dividing loss is optional - it only affects the scale of reported loss values for logging purposes
- Dividing loss compensates for the reduced batch statistics in batch normalisation layers

==========

Question 11

Implementing data augmentation for medical image classification (chest X-rays):

```python
train_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=30),
    transforms.ColorJitter(brightness=0.3, contrast=0.3),
    transforms.ToTensor(),
    transforms.Normalise(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

A radiologist reviews the augmentation pipeline and identifies a critical issue. What is it?

- RandomRotation(30°) can rotate anatomical structures to unrealistic orientations - lungs appearing diagonal makes images clinically invalid for training
- CenterCrop may remove diagnostically important regions at image edges - use RandomResizedCrop or full image with padding
- ImageNet normalisation values are wrong for X-rays - should compute dataset-specific mean/std from chest X-ray training data
- ***ColorJitter is inappropriate - X-rays are grayscale, and brightness/contrast changes could mask or create false pathological findings***

==========

Question 12

You implement a U-Net decoder with skip connections:

```python
class UNetDecoder(nn.Module):
    def __init__(self):
```

```
    super().__init__()
    # Encoder outputs: [64, 128, 256, 512] channels at decreasing resolutions

    self.up1 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
    self.conv1 = nn.Conv2d(512, 256, kernel_size=3, padding=1)  # 256 + 256 = 512

    self.up2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
    self.conv2 = nn.Conv2d(256, 128, kernel_size=3, padding=1)  # 128 + 128 = 256

    self.up3 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
    self.conv3 = nn.Conv2d(128, 64, kernel_size=3, padding=1)   # 64 + 64 = 128

def forward(self, x, skip_connections):
    # skip_connections = [s1, s2, s3] from encoder (64, 128, 256 channels)
    s1, s2, s3 = skip_connections

    x = self.up1(x)          # 512 -> 256 channels, 2× resolution
    x = torch.cat([x, s3], dim=1)  # Line A: concatenate with skip
    x = self.conv1(x)

    x = self.up2(x)
    x = torch.cat([x, s2], dim=1)
    x = self.conv2(x)
```

```
        x = self.up3(x)
        x = torch.cat([x, s1], dim=1)
        x = self.conv3(x)

        return x
```

The model crashes at Line A with "RuntimeError: Sizes of tensors must match". The encoder produces feature maps where each downsampling halves spatial dimensions. What causes this?

- ***ConvTranspose2d with stride=2 doesn't exactly double resolution when input dimensions are odd - use nn.Upsample(scale_factor=2) followed by Conv2d for precise size matching***
- Skip connection channels don't match - s3 has 256 channels but x after up1 also has 256, creating 512 input to conv1 which is correct
- The concatenation dimension is wrong - should use dim=0 for batch concatenation instead of dim=1 for channel concatenation
- Encoder and decoder must use same padding mode - if encoder uses padding='valid', decoder spatial sizes won't align with skip connections

==========
Question 13

Implementing Intersection over Union (IoU) for semantic segmentation evaluation:

```python
def compute_iou(pred, target, num_classes):
    """
    pred: (H, W) tensor with class indices
    target: (H, W) tensor with class indices
    """
```

```
    ious = []
    for cls in range(num_classes):
        pred_mask = (pred == cls)
        target_mask = (target == cls)

        intersection = (pred_mask & target_mask).sum().float()
        union = (pred_mask | target_mask).sum().float()

        if union == 0:
            # Option A: Skip this class
            continue
            # Option B: IoU = 1.0 (both pred and target have no pixels of this class)
            # ious.append(1.0)
            # Option C: IoU = 0.0 (penalise missing class)
            # ious.append(0.0)
        else:
            ious.append((intersection / union).item())

    return np.mean(ious)  # mIoU
```

For a dataset where class "motorcycle" rarely appears (only in 5% of images), which handling of union==0 gives the most meaningful mIoU?

- ***Option A (skip) - if class doesn't appear in both pred and target, it's not relevant for this image; including it would artificially inflate or deflate mIoU***

- Option B (IoU=1.0) - perfect agreement that class is absent; model correctly predicts no motorcycles when there are none
- Option C (IoU=0.0) - penalises model for not predicting rare classes; encourages better rare class detection
- Use frequency-weighted IoU instead - multiply each class IoU by its pixel frequency to reduce impact of rare classes

==========

Question 14

Debugging a transfer learning pipeline for image classification:

```python
# Load pretrained ResNet
model = torchvision.models.resnet50(pretrained=True)
# Freeze all layers
for param in model.parameters():
    param.requires_grad = False
# Replace classifier
model.fc = nn.Linear(2048, num_classes)
# Train
optimiser = torch.optim.Adam(model.parameters(), lr=0.001)
for epoch in range(num_epochs):
    for inputs, targets in train_loader:
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()
```

```

After training, validation accuracy is only 52% (random baseline for 10-class problem is 10%). Inspecting the model, fc.weight values are identical to initialisation. What's wrong?

- Optimiser should only receive fc.parameters() - currently it receives all parameters including frozen ones, but since frozen params have requires_grad=False, gradients aren't computed for them anyway
- ***The fc layer was added AFTER freezing parameters, so fc.weight.requires_grad is True, but optimiser was created before fc was modified - need to recreate optimiser***
- Requires_grad=False on backbone prevents gradient flow entirely - gradients don't propagate through frozen layers to reach fc layer
- Learning rate is too low for training from scratch - fc layer with random init needs higher LR (e.g., 0.01) than fine-tuning would

==========
Question 15

Implementing mixed precision training:

```python

scaler = torch.cuda.amp.GradScaler()

for inputs, targets in train_loader:

    optimiser.zero_grad()



    with torch.cuda.amp.autocast():

        outputs = model(inputs)

        loss = criterion(outputs, targets)



    # Option A

    loss.backward()

    optimiser.step()
```
```

```
# Option B
scaler.scale(loss).backward()
scaler.step(optimiser)
scaler.update()

# Option C
scaler.scale(loss).backward()
scaler.unscale_(optimiser)
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
scaler.step(optimiser)
scaler.update()
```

Training diverges with Option A but works with Option B. When would Option C be necessary instead of B?

- Option C is needed when training very deep networks to prevent gradient explosion
- Option C is needed when batch size is very small, causing high gradient variance
- ***Option C is needed when using gradient clipping - unscale_ converts gradients back to FP32 scale before clipping, otherwise clipping threshold is meaningless***
- Option C is only needed for debugging - it allows inspecting true gradient magnitudes

==========

Question 16

Implementing self-attention from scratch:

```python
class SelfAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
```

```
    super().__init__()
    self.num_heads = num_heads
    self.head_dim = embed_dim // num_heads

    self.q_proj = nn.Linear(embed_dim, embed_dim)
    self.k_proj = nn.Linear(embed_dim, embed_dim)
    self.v_proj = nn.Linear(embed_dim, embed_dim)
    self.out_proj = nn.Linear(embed_dim, embed_dim)

def forward(self, x, mask=None):
    B, T, C = x.shape

    Q = self.q_proj(x).view(B, T, self.num_heads, self.head_dim).transpose(1, 2)
    K = self.k_proj(x).view(B, T, self.num_heads, self.head_dim).transpose(1, 2)
    V = self.v_proj(x).view(B, T, self.num_heads, self.head_dim).transpose(1, 2)

    # Attention scores
    scores = Q @ K.transpose(-2, -1)  # Line A

    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))

    attn = F.softmax(scores, dim=-1)
```

```
        out = attn @ V



        out = out.transpose(1, 2).contiguous().view(B, T, C)

        return self.out_proj(out)
```

The attention weights are very peaked (one position gets ~99% weight). What scaling is missing at Line A?

- scores = (Q @ K.transpose(-2, -1)) / self.head_dim - scaling by d_k normalises for the number of dimensions summed
- ***scores = (Q @ K.transpose(-2, -1)) / math.sqrt(self.head_dim) - scaling by √d_k prevents softmax saturation when dot products grow with dimension***
- scores = (Q @ K.transpose(-2, -1)) / math.sqrt(self.num_heads) - scaling by √num_heads balances contribution across heads
- scores = F.normalise(Q, dim=-1) @ F.normalise(K, dim=-1).transpose(-2, -1) - cosine similarity instead of dot product

==========
Question 17

Implementing causal masking for autoregressive language modeling:

```python
def create_causal_mask(seq_len):
    # Option A
    mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
    return ~mask  # Invert: True where attention allowed


    # Option B
    mask = torch.tril(torch.ones(seq_len, seq_len)).bool()
    return mask
```

```
    # Option C
    mask = torch.ones(seq_len, seq_len).bool()
    for i in range(seq_len):
        mask[i, i+1:] = False
    return mask
```

For sequence "The cat sat", position 2 ("sat") should attend to positions 0,1,2 but not future positions. Which implementation is correct?

- All three produce equivalent masks - they all create lower triangular matrix where mask[i,j]=True iff j<=i
- Option A is wrong - triu with diagonal=1 creates upper triangular, and negation makes it lower triangular INCLUDING diagonal, but the logic is inverted for masked_fill
- ***Option B and C are correct; Option A depends on how the mask is used (True=attend vs True=mask out)***
- Only Option C is correct - explicit loop is the only way to ensure correct diagonal handling

==========

Question 18

Implementing a simple tokeniser for transformer training:

```python
class SimpleTokeniser:
    def __init__(self, vocab_size=10000):
        self.vocab_size = vocab_size
        self.word2idx = {'<PAD>': 0, '<UNK>': 1, '<BOS>': 2, '<EOS>': 3}
        self.idx2word = {v: k for k, v in self.word2idx.items()}

    def fit(self, texts):
        word_freq = Counter()
```

```
        for text in texts:
            word_freq.update(text.lower().split())

        # Add most frequent words to vocabulary
        for word, _ in word_freq.most_common(self.vocab_size - 4):
            idx = len(self.word2idx)
            self.word2idx[word] = idx
            self.idx2word[idx] = word

    def encode(self, text, max_len=512):
        tokens = [self.word2idx.get(w, self.word2idx['<UNK>'])
                  for w in text.lower().split()]
        tokens = [self.word2idx['<BOS>']] + tokens + [self.word2idx['<EOS>']]

        # Padding
        if len(tokens) < max_len:
            tokens += [self.word2idx['<PAD>']] * (max_len - len(tokens))
        else:
            tokens = tokens[:max_len]

        return tokens
```

When max_len=10 and text="Hello world how are you today my friend", the encoding has a critical flaw. What is it?

- <BOS> and <EOS> count toward max_len - effective content length is max_len-2, which may truncate more content than intended
- Padding tokens should be -100, not 0 - using 0 causes loss computation to include padding positions
- Word-level tokenisation creates massive vocabulary - should use subword tokenisation (BPE) for open-vocabulary handling
- ***Truncation removes <EOS> token - after truncating to 10 tokens, the sequence lacks end marker, confusing the model about sequence boundaries***

==========

Question 19

Implementing BERT-style masked language modeling loss:

```python
def compute_mlm_loss(model, input_ids, mask_prob=0.15):
    batch_size, seq_len = input_ids.shape

    # Create masking
    rand = torch.rand(input_ids.shape)
    mask_positions = (rand < mask_prob) & (input_ids != PAD_TOKEN)

    # Create labels (-100 for non-masked positions)
    labels = input_ids.clone()
    labels[~mask_positions] = -100

    # Replace masked tokens
    masked_input = input_ids.clone()
    masked_input[mask_positions] = MASK_TOKEN  # Line A
```

```
    # Forward pass

    logits = model(masked_input)



    # Compute loss only on masked positions

    loss = F.cross_entropy(logits.view(-1, vocab_size), labels.view(-1),
ignore_index=-100)



    return loss
```

BERT's original paper uses a more complex masking strategy at Line A. What is it and why?

- ***80% MASK token, 10% random token, 10% unchanged - prevents model from learning that MASK always means "predict here", improving fine-tuning where MASK doesn't exist***
- 100% MASK token for training efficiency - the 80/10/10 split was only for ablation studies and not used in final model
- Replace with learned [MASK] embedding instead of token - allows gradient flow through the mask position
- Apply masking only to content words (nouns, verbs) not function words (the, is, a) - forces model to learn semantic rather than syntactic patterns

==========
Question 20

Debugging a text classification fine-tuning pipeline:

```python
# Load pretrained BERT

tokeniser = BertTokeniser.from_pretrained('bert-base-uncased')

model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=5)

# Prepare data
```

```
def tokenise(text):
    return tokeniser(text, padding='max_length', truncation=True, max_length=128, return_tensors='pt')

# Training loop
model.train()
for epoch in range(3):
    for batch in train_loader:
        inputs = tokenise(batch['text'])
        inputs = {k: v.squeeze(0).to(device) for k, v in inputs.items()}
        labels = batch['label'].to(device)

        outputs = model(**inputs, labels=labels)
        loss = outputs.loss

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

Validation loss decreases for epoch 1, then increases sharply. Training loss keeps decreasing. What's the most likely cause?

- Missing gradient accumulation - batch size too small for stable training
- The squeeze(0) operation corrupts batch dimension - should process entire batch together
- Missing warmup scheduler - BERT requires linear warmup to stabilise early training
- ***Learning rate too high - BERT fine-tuning requires very small LR (2e-5 to 5e-5); larger LR causes catastrophic forgetting of pretrained knowledge***