Week 4 Sample Oral Assessment Questions

CSE151B SS1 2026

Below are questions that will be similar to what you'll see on this week's oral assessment. As mentioned in class, you'll be asked two questions–first an "understand" question, and then an "experiment" question. For questions where we imagine there are a few possible phrasings, there are multiple sample acceptable answers listed. Overall, if you completed the implementations for A1 and understood how and why they worked (either by thinking through or experimenting with different components), you should be well prepared.

Note that all of these contain excerpts from your assignment solution code for your reference, but you don't need to do a detailed read of every line of code to answer them–it's just there in case it helps as you answer the question!

Remember that we will only count your answer after you say **"My answer is..."** and we will only give hints after you answer "yes" to "Would you like a hint?" or answer incorrectly.

Understand

1. [Topic: why a contrastive model still needs classification logits at eval time]

```
# writeup: "...your contrastive model must still be able to produce classification logits
# at evaluation time---for example, by training the provided classifier head jointly with
# the contrastive loss."

class SupConModel(ScenarioModel):
    def forward(self, inputs, targets):
        # returns an L2-normalized projection embedding, used for the contrastive loss
        ...
        return embedding

    def encode(self, inputs):
        """Return classifier logits (used during evaluation)."""
        outputs = self.encoder(**inputs)
        cls_output = outputs.last_hidden_state[:, 0, :]
        logits = self.classify(self.dropout(cls_output))
        return logits
```

   Q: Above is an implementation of a Supervised (SupCon) Contrastive Learning model. Typically, a model's predicted output is computed by taking argmax over class logits. Why can't that be computed directly on the contrastive embedding that SupConModel.forward produces? Think about why this model needs a separate encode() method at all.

   Possible answer: My answer is: The contrastive embedding is just a point in a

representation space — it's not a score per class, so there's no argmax over classes to take on it directly. To get an actual predicted label you need something that maps the representation to one score per scenario class (a classifier head with target_size outputs). The contrastive loss alone never learns that mapping, so a separate path — here, encode(), which reuses the classifier head — is needed to produce real logits at evaluation time.

Hints
Tier 1: "Does the contrastive projection embedding have one number per scenario class, or just a fixed-size vector in some representation space?"
Tier 2: "What would you need to add on top of the embedding to turn it into one score per class?"
Tier 3: "The embedding alone has no per-class structure — you need a trained classifier head to turn it into logits argmax can be taken over."

2. [Topic: BERT [CLS] token pooling for classification]

```
def forward(self, inputs, targets):
  outputs = self.encoder(**inputs)
  cls = outputs.last_hidden_state[:, 0, :]   # <CLS> token
  dropped = self.dropout(cls)
  logits = self.classify(dropped)
  return logits
```

Q: Above is the forward pass of the baseline BERT classifier. Why does the code index last_hidden_state[:, 0, :] specifically, instead of e.g. averaging over all tokens?

Possible answer: My answer is: BERT prepends a special [CLS] token to every input, and during pretraining that token's final hidden state is trained to aggregate information about the whole sequence. Indexing position 0 pulls out that token's 768-dim representation and uses it as a fixed-size summary of the sentence for classification.

Hints
Tier 1: "What special token does a BERT tokenizer add at the start of every sequence?"
Tier 2: "last_hidden_state has shape (batch, seq_len, hidden). What does slicing index 0 on the seq_len axis pick out?"
Tier 3: "It's the [CLS] token — BERT is pretrained so that token's final hidden state summarizes the whole sequence, which is why classification heads read from it."

Experiment

1. [Topic: Hidden state dimensions]

```
class Classifier(nn.Module):
    def __init__(self, args, target_size):
        super().__init__()
        input_dim = args.embed_dim
        self.top = nn.Linear(input_dim, args.hidden_dim)
        self.relu = nn.ReLU()
        self.bottom = nn.Linear(args.hidden_dim, target_size)

    def forward(self, hidden):
        middle = self.relu(self.top(hidden))
        logit = self.bottom(middle)
        return logit
```

Q: Above is code that takes BERT's [CLS] hidden state (768 dimensional given by args.embed_dim) and turns it into an array of logits for classification predictions. An entrepreneur wants to create a smaller, lightweight version of BERT that duplicates the overall architecture of the transformer but decreases the hidden state dimensionality to 32. Why could this possibly be a bad idea for model performance?

Possible answers: My answer is: The amount of information that can be stored in the hidden representations dramatically decreases when reducing from 768 to 32. This causes the model to lose important information on larger or more complex sequences.
My answer is: Reducing the hidden state size also reduces the model's overall architectural size, leading to less capacity to learn complex relationships.

Hints:
Tier 1: The hidden state contains information about the meaning of tokens and the context that links them together.
Tier 2: Reducing the size of the hidden state leads to compounding effects where each vector at every step of the transformer also has reduced capacity.
Tier 3: Reducing the hidden size from 768 to 32 greatly decreases the model's representational capacity. Since every attention layer and feed-forward layer uses

these 32-dimensional hidden states, the model can encode much less information about each token, making it more likely to underfit and produce lower prediction accuracy.

2. [Topic: contrastive loss temperature]

```
# loss.py, SupConLoss.__init__
def __init__(self, temperature=0.07, contrast_mode='all', base_temperature=0.07):
    ...
    self.temperature = temperature

# loss.py, SupConLoss.forward
anchor_dot_contrast = torch.div(
    torch.matmul(anchor_feature, contrast_feature.T),
    self.temperature)
...
loss = - (self.temperature / self.base_temperature) * mean_log_prob_pos
```

Q: The similarity scores between every pair of embeddings get divided by temperature before a softmax is taken over them. Suppose an experimenter running the supcon task raises the temperature way up, from the default 0.07 to 10. What would you expect to happen to how well the model learns to separate different scenario classes in embedding space?

Possible answer: My answer is: Dividing by a much larger temperature squashes all the similarity scores much closer together before the softmax, so the resulting distribution over positives/negatives becomes nearly uniform instead of sharply peaked — the loss essentially can no longer tell the difference between a very similar pair and a very dissimilar one. That destroys most of the gradient signal that would normally pull same-class embeddings together and push different-class ones apart, so I'd expect training to learn a much less useful (less separated) embedding space, and downstream classification accuracy to suffer as a result.

Hints

Tier 1: "Temperature divides the similarity scores right before a softmax is applied. What does dividing a set of numbers by a much larger value do to the differences between them?"

Tier 2: "If the softmax input values are all nearly the same size, is the resulting softmax distribution sharply peaked on the true positives, or close to uniform over everything?"

Tier 3: "A high temperature flattens the softmax toward uniform, so the loss barely distinguishes positives from negatives anymore — the gradient signal that pulls

same-class embeddings together and pushes different-class ones apart largely disappears, hurting the learned embedding space."

3. <We may ask you which of the experiments in the assignment you chose to try, and then ask you a question based on it. For this sample question, the student chose the stochastic weight averaging (SWA) technique in their assignment. >
[Topic: Custom technique — stochastic weight averaging (SWA)]

```
model_SWA = AveragedModel(model)                                    # wrap model for SWA
scheduler_SWA = SWALR(model.optimizer, swa_lr=args.learning_rate)  # SWA scheduler
start_SWA = 5   # start SWA at epoch 5
...
if use_SWA and epoch_count >= start_SWA:
    scheduler_SWA.step()
    model_SWA.update_parameters(model)
...
if use_SWA and epoch_count > start_SWA:
    update_bn(train_dataloader, model)
    model = model_SWA
```

Q: An experimenter changes start_SWA from 5 down to 0, so weight-averaging begins on epoch 0 — averaging in the model's weights from the very first epoch, when it's barely trained and still close to its random classifier initialization. How would you expect the final averaged model to compare to one that starts averaging only after several epochs of normal training?

Possible answer: My answer is: Averaging in those very early, barely-trained snapshots pulls the final averaged weights back toward that poor early state, diluting the benefit of the later, better-converged snapshots. SWA is meant to average weights once training has already settled into a good region of the loss landscape — starting immediately mixes in noisy, mostly-untrained weights and would likely produce a worse final model than waiting until the model has partially converged first.

Hints
Tier 1: "What do the model's weights look like at epoch 0, versus after several epochs of real training?"
Tier 2: "If you average a well-trained snapshot with a barely-trained one, does the result lean toward 'well-trained,' or get dragged toward 'barely-trained'?"
Tier 3: "Averaging in very early, low-quality weights drags the final averaged model backward — SWA is meant to average only once training has already reached a good region, not from the start."