Practice Problems for PyTorch. Advanced Machine Learning

1. You are working on a classification problem with 10 input variables and 3 output classes.

(a) **Write a model in PyTorch using `torch.nn.Sequential` that includes a Softmax at the end. Why might this cause issues with standard cross-entropy?**

```
import torch
import torch.nn as nn

# Here we explicitly add a final Softmax
model = nn.Sequential(
nn.Linear(10, 16),
nn.ReLU(),
nn.Linear(16, 3),
nn.Softmax(dim=1) # final activation
)


#you also could just do multiclass logistic reg instead of a NN
model = nn.Sequential(
nn.Linear(10, 3),
nn.Softmax(dim=1) # final activation
)
```

(b) **Write a training loop for this model in PyTorch.**

```
def train_model(model, optimizer, train_dl, epochs=10):
            # Instead of CrossEntropyLoss (which expects raw logits),
            # we'll use NLLLoss, which expects log probabilities.
            criterion = nn.NLLLoss()

            for epoch in range(epochs):
                  model.train() # put model in training mode
                  total_samples = 0
                  sum_loss = 0.0

                  for x_batch, y_batch in train_dl:
                        optimizer.zero_grad()

                        # Forward pass: model(...) now returns probabilities
                        # because our model ends with Softmax(dim=1).
                        probs = model(x_batch)

                        # "Undo" the softmax by taking log(probabilities).
                        # Add a small epsilon to avoid log(0.0).
                        log_probs = torch.log(probs + 1e-8)

                        # NLLLoss expects log probabilities.
                        loss = criterion(log_probs, y_batch)

                        loss.backward()
                        optimizer.step()

                        # Bookkeeping
                        batch_size = y_batch.size(0)
                        total_samples += batch_size
                        sum_loss += loss.item() * batch_size

                  train_loss = sum_loss / total_samples
                  print(f"Epoch{epoch+1}/{epochs},Loss:{train_loss:.4f}")

            return model
```

Why is this needed? If we used `CrossEntropyLoss` directly on the model's output, it would run *another* log-softmax internally. This is generally not what we want when our model already outputs a softmax. Instead, we "undo" the softmax by taking the log ourselves, and feed that into `NLLLoss`, which expects log probabilities.

2. Write a function that, given a model and a data loader, computes balanced accuracy. Assume you have a binary classification problem.

```
import numpy as np
from sklearn.metrics import balanced_accuracy_score

def compute_balanced_accuracy(model, data_loader):
      model.eval()
      all_preds = []
      all_labels = []

      with torch.no_grad():
            for x, y in data_loader:
                  probs = model(x)
                  # Since we use Softmax, 'probs' are probs.
                  # For multi-class, we pick class argmax either way:
                  preds = torch.argmax(logits_or_probs, dim=1)
                  # .cpu() moves the tensor from GPU memory to CPU memory
                  # so we can convert it to a NumPy array with .numpy()
                  all_preds.append(preds.cpu().numpy())
                  all_labels.append(y.cpu().numpy())

      all_preds = np.concatenate(all_preds)
      all_labels = np.concatenate(all_labels)

      # Compute balanced accuracy
      return balanced_accuracy_score(all_labels, all_preds)
```

3. What is the shape of the tensor `out` in the following code?

```
embed = nn.Embedding(5, 7)
x = torch.LongTensor([[1,0,1,4,2,1]])
out = embed(x)
```

The embedding dimension is 7, and the input `x` has shape $(1, 6)$. Thus:

$$(1, 6, 7).$$

4. What is the value of the tensor `x.grad` after running the next few lines?

```
x = torch.tensor([1, 3, 2], requires_grad=True)
L = (3*x + 7).sum()
L.backward()
```

$$L = \sum_i (3x_i + 7).$$

Thus the gradient of L w.r.t. each x_i is 3, so:

$$\texttt{x.grad} = [3, 3, 3].$$

5. Create the following tensors with shape `(2,2,2)`: all zeros, all ones, and random with normal distribution.

```
# All zeros
t1 = torch.zeros((2,2,2))

# All ones
t2 = torch.ones((2,2,2))

# Random with normal distribution
t3 = torch.randn((2,2,2))
```

6. Describe what each of the following lines of code are doing during a training loop:

- `loss.backward()`:
  - Calculates gradients of `loss` with respect to x.
  - The gradients are stored in `param.grad` for each parameter.
- `optimizer.step()`:
  - Updates the parameters based on the gradients computed by `loss.backward()`.