NNDL Lab 3 Report
Sharvita Paithankar

1.Details of the data pre-processing step. How was the dataset pre-processed? Why did you make the design choices you did? How much training data was used and Why?

For the data size, I decided to pick 2000 samples from the VizWiz training dataset and the full validation set was used for model evaluation. 100 samples from the test were used for prediction generation. I decided to cut down to 2000 samples because of the lack of resources of computation power (GPU/ TPU etc) and wanted to run my code in a shorter period of time. The training set size balances the computational resources as well.

```
NUM_SAMPLES = 2000
train_data_subset = train_data[:NUM_SAMPLES]
print(f'Using {len(train_data_subset)} training samples')
```

```
Using 2000 training samples
```

```
print("Creating datasets...")
train_dataset = VizWizDataset(train_data_subset, img_dir, tokenizer)
val_dataset = VizWizDataset(val_data[:1000], img_dir, tokenizer)  # Using a subset of validation data
test_dataset = VizWizDataset(test_data[:100], img_dir, tokenizer, is_test=True)
```

```
Creating datasets...
Answer vocabulary size: 879
Answer vocabulary size: 277
```

To process the text, I built from training data with a minimum word frequency threshold of 5. I did this because the threshold of 5 reduces vocabulary size by filtering rare words but also prevents overfitting. I also preprocessed questions and tokenized them. Questions and words that are not in vocabulary were mapped to <unk>. The question strings are being tokenized with the BERT tokenizer, which transforms them into input IDs and their correct corresponding attention masks. The tokenizer makes a maximum sequence length of 30 tokens to help reduce computation and get rid of long sequences that would slow training. If an answer is provided in the annotation, the code gets all answers for a given question, converts them to lowercase, and picks the most frequent one as the “label” for that sample. Doing this will help to reduce complexity in training because many different but semantically similar answers could cause confusion for the model.

```
    self.idx_to_answer = {v: k for k, v in self.answer_vocab.i

def build_answer_vocab(self):
    ans_vocab = {'<unk>': 0}
    idx = 1

    for sample in self.data:
        if 'answers' in sample:
            processed_ans = process_answers(sample['answers'])
            if processed_ans not in ans_vocab:
                ans_vocab[processed_ans] = idx
                idx += 1

    return ans_vocab
```

For the images, I resized all the images to 224×224 pixels which is a common CNN input size. Normalization was also done using ImageNet mean for the images since the CNN architectures are usually pre-trained on ImageNet. This makes sure that all images have the same size and that their pixel intensities are scaled in a way that neural networks can handle it well.

```
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ])
```

For answer processing, I picked the most common answer as the “best” answer which helps to address annotation variability. I also created an answer vocabulary from the training data.

```
def process_answers(answers, max_answers=10):
    # Return the most common answer
    answer_dict = {}
    for ans in answers:
        ans_text = ans['answer'].lower()
        if ans_text not in answer_dict:
            answer_dict[ans_text] = 0
        answer_dict[ans_text] += 1

    # Sort by frequency and return most common
    sorted_answers = sorted(answer_dict.items(), key=lambda x: x[1], reverse=True)
    return sorted_answers[0][0] if sorted_answers else "unknown"
```

2. **Details of the multi-modal architectures. Which types of layers did you use? How do the models incorporate both modalities? How do they handle the different types of output?**

For Challenge 1, I have used a ResNet-like CNN for image encoding which has three convolutional blocks : Conv2D → BatchNorm → ReLU → MaxPool pattern. This created a 256-dimensional visual feature vector and the question encoder uses a word embedding layer (300 dimensions) which uses a unidirectional LSTM with 512 hidden units. These modality-specific features are joined together and before applying a sigmoid activation for binary classification, it is processed through a multi-layer perceptron with ReLU activations and dropout (0.3), gradually reducing dimensions (768 → 512 → 256 → 1).

```python
class VQAClassificationModel(nn.Module):
    def __init__(self, hidden_dim=512):
        super(VQAClassificationModel, self).__init__()


        self.img_encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

            # Conv block 1
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 2
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Final pooling
            nn.AdaptiveAvgPool2d((1, 1))
        )

        self.text_embedding = nn.Embedding(tokenizer.vocab_size, 256)
        self.text_encoder = nn.LSTM(256, hidden_dim, batch_first=True, bidirectional=True)

        self.text_attention = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

        self.fusion = nn.Sequential(
            nn.Linear(256 + hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def forward(self, img, input_ids, attention_mask):
        # Encode image: B x 3 x 224 x 224 -> B x 256 x 1 x 1 -> B x 256
        img_features = self.img_encoder(img).squeeze(-1).squeeze(-1)


        embedded_q = self.text_embedding(input_ids)


        embedded_q = embedded_q * attention_mask.unsqueeze(-1)
```

For Challenge 2, I used an image encoder that is deeper, with three convolutional blocks including multiple convolutional layers in some blocks. This created a 512-dimensional feature vector. The question encoder uses a bidirectional LSTM with 2 layers and dropout (0.3). An attl also added an attention mechanism that focuses on relevant parts of the question and it also uses proper masking for padded sequences. The model joins the 512-dimensional image features with the 1024-dimensional attended text features and before mapping to answer vocabulary logits,it processes this through a two-layer MLP with higher dropout (0.4).

```
class VQAAnswerModel(nn.Module):
    def __init__(self, answer_vocab_size, hidden_dim=512):
        super(VQAAnswerModel, self).__init__()

        # Image encoder
        self.img_encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

            # Conv block 1
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 2
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 3
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            # Final pooling
            nn.AdaptiveAvgPool2d((1, 1))
        )


        self.text_embedding = nn.Embedding(tokenizer.vocab_size, 256)
        self.text_encoder = nn.LSTM(256, hidden_dim, batch_first=True, bidirectional=True, num_layers=2, dropout=0.3)

        # Attention mechanism for text features
        self.text_attention = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

        self.fusion = nn.Sequential(
            nn.Linear(512 + hidden_dim * 2, hidden_dim * 2),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.4)
```

```
        self.classifier = nn.Linear(hidden_dim, answer_vocab_size)

    def forward(self, img, input_ids, attention_mask):
        # Encode image
        img_features = self.img_encoder(img).squeeze(-1).squeeze(-1)  # B x 512


        embedded_q = self.text_embedding(input_ids)  # B x seq_len x embed_dim

        embedded_q = embedded_q * attention_mask.unsqueeze(-1)

        q_output, _ = self.text_encoder(embedded_q)  # B x seq_len x (hidden_dim*2)

        attention_weights = self.text_attention(q_output).squeeze(-1)  # B x seq_len
        attention_weights = F.softmax(attention_weights.masked_fill((1 - attention_mask).bool(), float('-inf')), dim=1)
        q_features = torch.bmm(attention_weights.unsqueeze(1), q_output).squeeze(1)  # B x (hidden_dim*2)

        # Concatenate features and fuse
        combined = torch.cat([img_features, q_features], dim=1)
        fused = self.fusion(combined)

        # Generate answer probabilities
        logits = self.classifier(fused)
        return logits
```

Both architectures use a late fusion approach and each modality is processed separately before feature integration. This helps each encoder to get the modality-specific information before integration. The implementation uses convolutional layers with batch normalization for images, embedding and LSTM layers for text, and fully-connected layers with ReLU activations and dropout for fusion.

3. **Details of model adjustment and hyper-parameter tuning using the validation set. Did the model development process reveal interesting trends about architecture design? Which hyper-parameters were the most important? How did you decide on the final versions?**

Hyper-parameter tuning is an iterative process that involves adjusting the model's configuration to optimize performance. We used the validation set to evaluate the model's performance on unseen data during training, which helped in adjusting hyper-parameters to avoid overfitting.

Some of the key hyper-parameters that were tuned include the learning rate, batch size, number of epochs, and dropout rate. The learning rate is one of the most important hyper-parameters, because it controls how much the model's weights are adjusted in each update. If the learning rate is too high, the model may overfit and the optimal weights will not be accurate, but if it's too low, the model might converge too slowly or get stuck in local minima. The batch size was also adjusted to find the right balance between computational efficiency and model performance. Smaller batches typically mean more frequent updates to the model but it might allow noise, but larger batches provide more stable updates but also might require more memory and computation.

The dropout rate was tuned to avoid overfitting, which was important for the complexity of the multi-modal architecture. Dropout randomly disables neurons during training, forcing the model to rely on multiple features, reducing the likelihood of overfitting to the training data.

During model development, it became clear that adding an attention mechanism helped improve the model's accuracy a lot, especially for the text modality. Attention mechanisms also allowed the model to focus on the most important parts of the text, which enhanced performance. The validation set also showed that increasing the depth of the image pathway helped improve performance, but after a certain point, adding more layers did not help. The final model design was decided based on the best performance on the validation set, making sure that it achieved a good balance between bias and variance. This led to a better model that could generalize well to unseen data.

4. **Details of the single modality architectures (if applicable). If you chose to participate in the text-only or image-only challenges, describe how your architectures were designed. Are they the same as the multi-modal architecture? If not, how and why are they different?**

My implementation has specialized single-modality architectures for both visual-only and text-only approaches to the classification task where each is designed to compensate for the missing modality.

The visual-only classification model features an enhanced image encoder compared to its multi-modal counterpart. It includes additional convolutional layers with some blocks containing multiple consecutive operations, resulting in a deeper network. The feature dimension increases to 512 (from 256), providing greater representational capacity. The classification head is also expanded with an additional hidden layer while maintaining dropout (0.3) for regularization. These enhancements enable the model to extract more detailed visual information to compensate for the absence of textual context.

The text-only classification model similarly adapts to its limited input. It employs a bidirectional LSTM with 2 layers (unlike the unidirectional LSTM in the multi-modal version) and incorporates an attention mechanism absent in the multi-modal classification model. This allows the network to focus on the most relevant parts of the question text. The classification head maintains a similar structure to the multi-modal version but operates on the attention-weighted text features.

The differences between these single-modality architectures and their multi-modal counterparts reveal important insights about modality compensation. Both single-modality models feature increased complexity with additional layers and parameters to extract more information from the available modality. The visual-only model adds convolutional layers to capture more visual details, while the text-only model adds bidirectionality and attention to better understand question semantics.

Interestingly, attention mechanisms become more critical in single-modality models, especially for text-only processing. This suggests that when deprived of complementary information, the model must be more selective about which aspects of the available modality to prioritize.

The design rationale appears to be that single-modality models need enhanced feature extraction capabilities to compensate for missing complementary information. The increased network depth, attention mechanisms, and higher feature dimensions all serve to maximize information extraction from the limited input modality while maintaining the overall architectural approach to VQA.

PART 2:
CODE

```python
import os
import json
import requests
import pickle
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
from io import BytesIO
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from sklearn.metrics import accuracy_score
from transformers import BertTokenizer
```

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```python
# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)
```

```python
!pip install transformers
!pip install requests
!pip install Pillow
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.49.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from transformers) (3.18.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.26.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.29.3)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.21.1)
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.5.3)
```

```python
img_dir = 'https://vizwiz.cs.colorado.edu/VizWiz_visualization_img/'
ann_dir = 'https://vizwiz.cs.colorado.edu/VizWiz_final/vqa_data/Annotations/'
```

```python
train_annotation_path = f'{ann_dir}train.json'
val_annotation_path = f'{ann_dir}val.json'
test_annotation_path = f'{ann_dir}test.json'
```

```python
def get_data(url):
    try:
        data = requests.get(url, allow_redirects=True).json()
        return data
    except Exception as e:
        print(f"Error downloading data: {e}")
        return None
```

```python
print("Loading datasets...")
train_data = get_data(train_annotation_path)
val_data = get_data(val_annotation_path)
test_data = get_data(test_annotation_path)

print(f'Training set size: {len(train_data)}')
print(f'Validation set size: {len(val_data)}')
print(f'Test set size: {len(test_data)}')
```

```
Loading datasets...
Training set size: 20523
Validation set size: 4319
Test set size: 8000
```

```
[ ] NUM_SAMPLES = 2000
    train_data_subset = train_data[:NUM_SAMPLES]
    print(f'Using {len(train_data_subset)} training samples')
```

```
Using 2000 training samples
```

```
[ ] tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    MAX_LEN = 30
```

```
    def process_answers(answers, max_answers=10):
        # Return the most common answer
        answer_dict = {}
        for ans in answers:
            ans_text = ans['answer'].lower()
            if ans_text not in answer_dict:
                answer_dict[ans_text] = 0
            answer_dict[ans_text] += 1

        # Sort by frequency and return most common
        sorted_answers = sorted(answer_dict.items(), key=lambda x: x[1], reverse=True)
        return sorted_answers[0][0] if sorted_answers else "unknown"
```

+ Code + Text

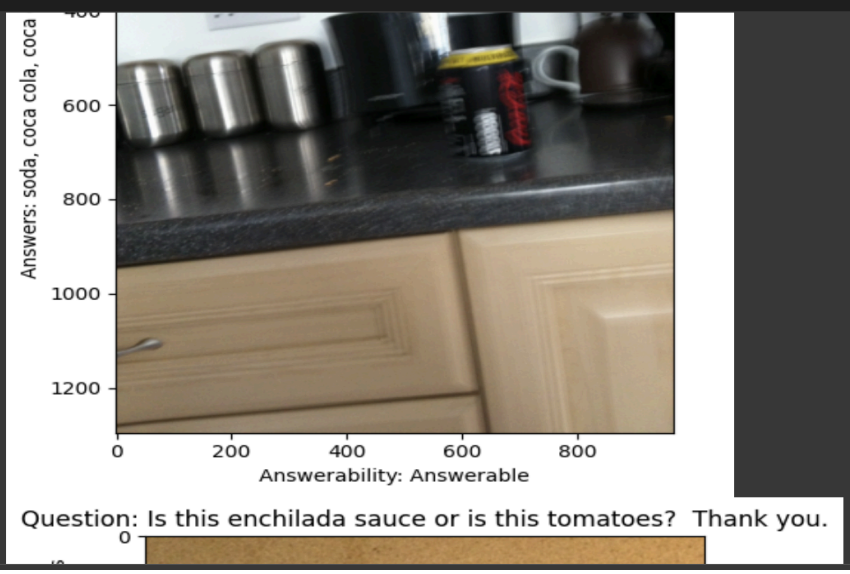
```
[ ] def visualize_sample(sample, img_dir):
        img_path = img_dir + sample['image']
        question = sample['question']

        try:
            response = requests.get(img_path)
            img = Image.open(BytesIO(response.content))

            plt.figure(figsize=(10, 6))
            plt.imshow(img)
            plt.title(f"Question: {question}")

            if 'answerable' in sample:
                answer_text = "Not answerable" if sample['answerable'] == 0 else "Answerable"
```

```python
for i in range(3):
    if i < len(train_data_subset):
        visualize_sample(train_data_subset[i], img_dir)
```



Question: Is this enchilada sauce or is this tomatoes? Thank you.

```python
class VizWizDataset(Dataset):
    def __init__(self, data, img_dir, tokenizer, is_test=False, max_len=30):
        self.data = data
        self.img_dir = img_dir
        self.tokenizer = tokenizer
        self.is_test = is_test
        self.max_len = max_len

        # Define image transformations
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ])

        # Create answer vocabulary for the answer prediction task
        if not is_test:
            self.answer_vocab = self.build_answer_vocab()
            print(f'Answer vocabulary size: {len(self.answer_vocab)}')
            self.idx_to_answer = {v: k for k, v in self.answer_vocab.items()}

    def build_answer_vocab(self):
        ans_vocab = {'<unk>': 0}
        idx = 1

        for sample in self.data:
            if 'answers' in sample:
                processed_ans = process_answers(sample['answers'])
                if processed_ans not in ans_vocab:
                    ans_vocab[processed_ans] = idx
                    idx += 1

        return ans_vocab

    def __len__(self):
        return len(self.data)

    def get_image(self, img_path):
        try:
            # Download image
            response = requests.get(img_path)
            img = Image.open(BytesIO(response.content)).convert('RGB')
            return self.transform(img)
        except Exception as e:
            print(f"Error loading image: {e}")
```

```
            print(f"Error loading image: {e}")

            return torch.zeros(3, 224, 224)

    def __getitem__(self, idx):
        sample = self.data[idx]


        question = sample['question']
        encodings = self.tokenizer(
            question,
            truncation=True,
            max_length=self.max_len,
            padding='max_length',
            return_tensors='pt'
        )


        input_ids = encodings['input_ids'].squeeze(0)
        attention_mask = encodings['attention_mask'].squeeze(0)


        img_path = self.img_dir + sample['image']
        img_tensor = self.get_image(img_path)

        if self.is_test:

            return {
                'image': img_tensor,
                'input_ids': input_ids,
                'attention_mask': attention_mask,
                'image_id': sample['image'],
                'question': question
            }
        else:

            answerable = torch.tensor(sample['answerable'], dtype=torch.float)


            processed_ans = process_answers(sample['answers'])
            answer_id = self.answer_vocab.get(processed_ans, 0)  # 0 is <unk>

            return {
                'image': img_tensor,
                'input_ids': input_ids,
                'attention_mask': attention_mask,
                'answerable': answerable,
```

```
BATCH_SIZE = 32
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

```
class VQAClassificationModel(nn.Module):
    def __init__(self, hidden_dim=512):
        super(VQAClassificationModel, self).__init__()

        # Image encoder (simplified CNN architecture)
        self.img_encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

            # Conv block 1
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 2
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Final pooling
            nn.AdaptiveAvgPool2d((1, 1))
        )

        # Question encoder (custom LSTM-based encoder)
        self.text_embedding = nn.Embedding(tokenizer.vocab_size, 256)
        self.text_encoder = nn.LSTM(256, hidden_dim, batch_first=True, bidirectional=True)

        # Attention mechanism for text
        self.text_attention = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

        # Fusion layers
```

```
        self.text_attention = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

        # Fusion layers
        self.fusion = nn.Sequential(
            nn.Linear(256 + hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def forward(self, img, input_ids, attention_mask):
        # Encode image: B x 3 x 224 x 224 -> B x 256 x 1 x 1 -> B x 256
        img_features = self.img_encoder(img).squeeze(-1).squeeze(-1)


        embedded_q = self.text_embedding(input_ids)  # B x seq_len x embed_dim


        embedded_q = embedded_q * attention_mask.unsqueeze(-1)


        q_output, _ = self.text_encoder(embedded_q)  # B x seq_len x (hidden_dim*2)


        attention_weights = self.text_attention(q_output).squeeze(-1)  # B x seq_len
        attention_weights = F.softmax(attention_weights.masked_fill((1 - attention_mask).bool(), float('-inf')), dim=1)
        q_features = torch.bmm(attention_weights.unsqueeze(1), q_output).squeeze(1)  # B x (hidden_dim*2)


        combined = torch.cat([img_features, q_features], dim=1)

        # Predict
        output = self.fusion(combined)
        return output.squeeze(-1)
```

```python
class VQAAnswerModel(nn.Module):
    def __init__(self, answer_vocab_size, hidden_dim=512):
        super(VQAAnswerModel, self).__init__()

        self.img_encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

            # Conv block 1
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 2
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv block 3
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            # Final pooling
            nn.AdaptiveAvgPool2d((1, 1))
        )

        # Question encoder with attention
        self.text_embedding = nn.Embedding(tokenizer.vocab_size, 256)
        self.text_encoder = nn.LSTM(256, hidden_dim, batch_first=True, bidirectional=True, num_layers=2, dropout=0.3)

        # Attention mechanism for text features
        self.text_attention = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
```

```
            nn.Linear(hidden_dim, 1)
        )

        # Fusion layers
        self.fusion = nn.Sequential(
            nn.Linear(512 + hidden_dim * 2, hidden_dim * 2),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.4)
        )

        # Answer prediction head
        self.classifier = nn.Linear(hidden_dim, answer_vocab_size)

    def forward(self, img, input_ids, attention_mask):

        img_features = self.img_encoder(img).squeeze(-1).squeeze(-1)  # B x 512


        embedded_q = self.text_embedding(input_ids)  # B x seq_len x embed_dim


        embedded_q = embedded_q * attention_mask.unsqueeze(-1)


        q_output, _ = self.text_encoder(embedded_q)  # B x seq_len x (hidden_dim*2)


        attention_weights = self.text_attention(q_output).squeeze(-1)  # B x seq_len
        attention_weights = F.softmax(attention_weights.masked_fill((1 - attention_mask).bool(), float('-inf')), dim=1)
        q_features = torch.bmm(attention_weights.unsqueeze(1), q_output).squeeze(1)  # B x (hidden_dim*2)


        combined = torch.cat([img_features, q_features], dim=1)
        fused = self.fusion(combined)


        logits = self.classifier(fused)
        return logits
```

```python
def train_classification_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=10):
    best_val_acc = 0.0
    history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_idx, batch in enumerate(train_loader):

            images = batch['image'].to(device)
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['answerable'].to(device)


            outputs = model(images, input_ids, attention_mask)
            loss = criterion(outputs, labels)


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


            train_loss += loss.item()
            preds = (outputs > 0.5).float()
            train_correct += (preds == labels).sum().item()
            train_total += labels.size(0)


            if (batch_idx + 1) % 50 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Batch [{batch_idx+1}/{len(train_loader)}], Loss: {loss.item():.4f}')

        epoch_train_loss = train_loss / len(train_loader)
        epoch_train_acc = train_correct / train_total

        # Validation phase
        model.eval()
        val_loss = 0.0
        val_correct = 0
```

```
    val_total = 0

    with torch.no_grad():
        for batch in val_loader:
            # Move data to device
            images = batch['image'].to(device)
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['answerable'].to(device)

            # Forward pass
            outputs = model(images, input_ids, attention_mask)
            loss = criterion(outputs, labels)

            # Track statistics
            val_loss += loss.item()
            preds = (outputs > 0.5).float()
            val_correct += (preds == labels).sum().item()
            val_total += labels.size(0)

    # Calculate validation metrics
    epoch_val_loss = val_loss / len(val_loader)
    epoch_val_acc = val_correct / val_total

    # Save history
    history['train_loss'].append(epoch_train_loss)
    history['val_loss'].append(epoch_val_loss)
    history['train_acc'].append(epoch_train_acc)
    history['val_acc'].append(epoch_val_acc)

    # Print epoch results
    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.4f}, '
          f'Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f}')

    # Save best model
    if epoch_val_acc > best_val_acc:
        best_val_acc = epoch_val_acc
        torch.save(model.state_dict(), 'best_classification_model.pth')
        print(f'Best model saved with validation accuracy: {best_val_acc:.4f}')

return history
```

```python
def train_answer_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=10):
    best_val_acc = 0.0
    history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_idx, batch in enumerate(train_loader):

            images = batch['image'].to(device)
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['answer_id'].to(device)


            outputs = model(images, input_ids, attention_mask)
            loss = criterion(outputs, labels)


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


            train_loss += loss.item()
            _, preds = torch.max(outputs, 1)
            train_correct += (preds == labels).sum().item()
            train_total += labels.size(0)


            if (batch_idx + 1) % 50 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Batch [{batch_idx+1}/{len(train_loader)}], Loss: {loss.item():.4f}')


        epoch_train_loss = train_loss / len(train_loader)
        epoch_train_acc = train_correct / train_total


        model.eval()
        val_loss = 0.0
        val_correct = 0
```

```
    with torch.no_grad():
        for batch in val_loader:

            images = batch['image'].to(device)
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['answer_id'].to(device)


            outputs = model(images, input_ids, attention_mask)
            loss = criterion(outputs, labels)


            val_loss += loss.item()
            _, preds = torch.max(outputs, 1)
            val_correct += (preds == labels).sum().item()
            val_total += labels.size(0)


    epoch_val_loss = val_loss / len(val_loader)
    epoch_val_acc = val_correct / val_total


    history['train_loss'].append(epoch_train_loss)
    history['val_loss'].append(epoch_val_loss)
    history['train_acc'].append(epoch_train_acc)
    history['val_acc'].append(epoch_val_acc)

    |
    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.4f}, '
          f'Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f}')

    # Save best model
    if epoch_val_acc > best_val_acc:
        best_val_acc = epoch_val_acc
        torch.save(model.state_dict(), 'best_answer_model.pth')
        print(f'Best model saved with validation accuracy: {best_val_acc:.4f}')

return history
```