UMBC CMSC 475/675 Neural Networks

Homework 3. 100 points

Due: May 15, 2025. 23:59 Eastern

Answer all problems according to the instructions provided.
Submit one `.zip` file named `LastName_FirstName_hw3.zip` containing a single `pdf` named `hw3.pdf` with the answers to both the written and coding sections. The pdf **must** contain typeset or handwritten answers for the written problems and relevant code snippets, outputs and results, and other requested information for the programming tasks. Put all other files (code, data, outputs, etc.) in a single directory named "`code`"

1 Build Your Own Transformer 50 points

In this section, we will build the Transformer architecture as introduced in the paper *Attention is All You Need* (https://arxiv.org/pdf/1706.03762). You will construct the model using PyTorch, including its encoder and decoder stacks, multi-head attention, and positional encoding. Complete the code provided in each following part. **Training and evaluation are not required**, but the code must be correct both logically and in terms of syntax.

Part 1: Multi-head Attention In this part, you will implement the core attention mechanism used throughout the Transformer. The multi-head attention module projects the input into multiple query, key, and value subspaces, applies scaled dot-product attention in parallel across heads, and concatenates the results. Your implementation should support multiple heads and include appropriate linear projections and dropout.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads,proj_drop=0.1, atten_drop=0.1):
        super(MultiHeadAttention,self).__init__()
        assert d_model % num_heads == 0, "d_model divisible by num_heads."
        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        # Linear layers to project input to q, k, v
        < your-code >

         # Output linear layer
        < your-code >

        # Dropout for attention weights
        < your-code >

    def forward(self, query, key, value, mask=None):
        # Project inputs to q, k, v
        < your-code >

        # Calculate scaled dot-product attention scores
        # Apply mask (if provided)
        # (Hint: use torch.matmul and scale by sqrt(d_k))
        < your-code >
```

```
        # Attention dropout
        < your-code >

        # Concatenate heads and apply final linear projection
        < your-code >
        return output
```

Part 2: Positional Encoding Positional Encoding is used to inject the position information of each token in the input sequence. It uses sine and cosine functions of different frequencies to generate the positional encoding.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding,self).__init__()
        # Define your position embedding function
        < your-code >

    def forward(self,x):
        # Add positional encoding to input x
        return < your-code >
```

Part 3: EncoderLayer In this part, you will implement a single layer of the Transformer encoder, as shown in Figure 1. Each encoder layer consists of a multi-head self-attention block followed by a position-wise feed-forward network. Residual connections, layer normalization, and dropout are applied after each sub-layer. You should use your `MultiHeadAttention` module from Part 1 as a building block.

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super(EncoderLayer, self).__init__()

        #  Use your Multi-Head Attention module
        < your-code >

        #  Define position-wise feed-forward network
        < your-code >

        # Define two layer normalization layers
        < your-code >

        # Define dropout layers if needed
        < your-code >

    def forward(self, x):
        < your-code >
        return output
```

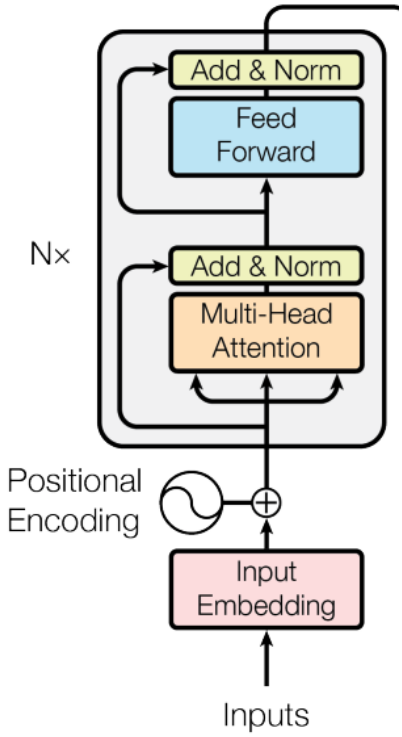

Figure 1: Transformer Encoder

Note: In the Transformer encoder architecture (see Figure 1), there are N stacked identical layers. The `EncoderLayer` class below defines *one single layer* of the encoder, not the entire encoder stack.

Components of the Transformer Encoder. The Transformer encoder consists of Nstacked identical layers, each composed of a *multi-head self-attention mechanism* and *a feed-forward network*.

- *Layer normalization* is applied before each sub-layer to stabilize and accelerate training. It operates independently on each position and feature dimension, unlike batch normalization which depends on batch statistics.
- *Residual connections* are applied around each sub-layer, followed by layer normalization and dropout. This allows gradients to flow better during backpropagation and prevents vanishing or exploding gradients in deep stacks.
- By stacking several such encoder blocks, the model learns deep contextual representations, essential for capturing long-range dependencies in input sequences.

Part 4: DecoderLayer Based on Figure 2, implement the `DecoderLayer` class, with two Multi-Head Attention layers, a Position-wise Feed-Forward layer, and three Layer Normalization layers.

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, proj_drop, atten_drop):
        super(DecoderLayer, self).__init__()
        # Define projection layers for masked self-attention
        < your-code >
        # Define projection layers for encoder-decoder attention
        < your-code >

        # Define position-wise feed-forward network
        < your-code >
        # Define three layer normalization layers
        < your-code >

        # Optional: Define dropout layers if needed
        < your-code >

    def forward(self, x, enc_out, mask):
        # Masked multi-head self-attention on decoder input
        < your-code >
        # Encoder-decoder attention (attend over encoder outputs)
        < your-code >

        # Position-wise feed-forward network
        < your-code >
        return output
```

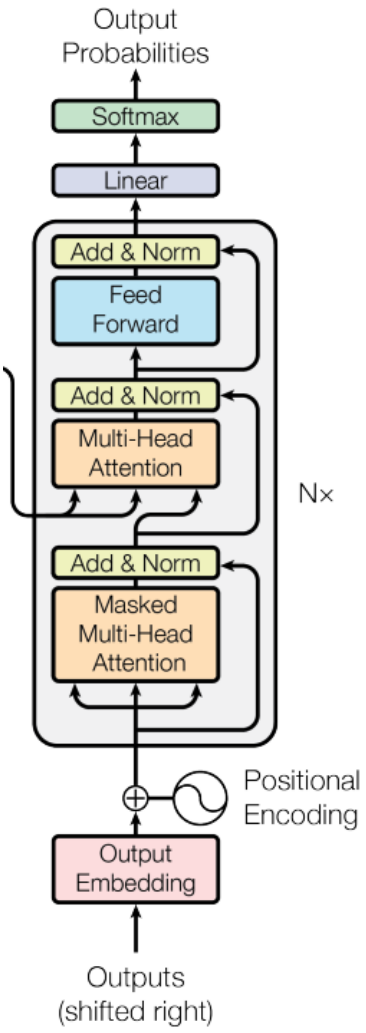

Figure 2: Transformer Decoder Architecture

Components of the Transformer Decoder The decoder consists of a stack of N identical layers, each designed to generate outputs conditioned on both previous outputs and the encoder's final representations. Each decoder layer is composed of three main sub-components:

- a masked multi-head self-attention mechanism
- an encoder-decoder attention mechanism
- a position-wise feed-forward network.

The masked self-attention ensures that predictions for a position can only depend on earlier positions in the output sequence, preserving the autoregressive property needed for generation tasks. The encoder-decoder attention allows the decoder to attend to all positions in the encoder's output, effectively capturing the source sequence information.

As in the encoder, each sub-layer is preceded by layer normalization and surrounded by a residual connection, followed by dropout for regularization. By stacking multiple decoder layers, the model can generate complex, contextually rich sequences conditioned on the source input.

Part 5: Implement Transformer Class In this final part, you will assemble the full Transformer model using the components you have implemented in previous parts. Your model should include token embeddings, positional encodings, stacked encoder and decoder layers, and an output projection layer. The encoder processes the input sequence, and the decoder generates the output sequence conditioned on both the previous outputs and the encoder's representations.

Note: In this question, **you only need to handle attention masks in the decoder**. Specifically, the masked self-attention sub-layer should apply a causal mask to prevent attending to future tokens, and the encoder-decoder attention may optionally take a mask indicating which encoder positions to attend to. The encoder layers do not use any masks.

```
class Transformer(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, num_layers, dropout
    =0.1, max_seq_length=512):
        super(Transformer, self).__init__()
        # Token embedding and Positional encoding
        < your-code >
        # Encoder stack (N layers)
        < your-code >
        # Decoder stack (N layers)
        < your-code >
        # Output projection layer
        <your-code>

    def forward(self, src, tgt, mask):
        < your-code >
        return output
```

2 Adversarial Attacks — 30 points

In this part, we will use an existing GitHub repository to implement and compare different adversarial attack approaches on standard image classification models. You can run this code in Google Colab or on any computer where you can clone GitHub repositories and install packages using `pip`.

Step 1: Load three different pretrained models from PyTorch. There are several pretrained models available. One example is shown below:

```
import torch
import torchvision.models as models

# Load the pretrained ResNet50 model from torchvision
model = models.resnet50(pretrained=True)
```

Step 2: In class we discussed a few adversarial attack techniques such as PGD, FGSM, etc. `torchattacks` is a framework which lets you attack any model with many such adversarial attack techniques. Install `torchattacks` and then use it to attack your loaded model with 3 different attack techniques. An example is shown here: https://github.com/Harry24k/adversarial-attacks-pytorch/blob/master/demo/White-box%20Attack%20on%20ImageNet.ipynb

Step 3: Report accuracy in a 3×3 table (3 rows for models and 3 columns for attack techniques).

Step 4: Choose any image of your choice and show the corresponding adversarial images generated by each attack technique from Step 2. For each adversarial image, report the average mean-squared error between the original image and the adversarial image.

3 Spatial Reasoning with Diffusion Models — 20 points

For this part, you can use any online diffusion model to generate images (this will not count as an academic integrity violation). Some easily available online demos include: Stable Diffusion 2.1: https://huggingface.co/spaces/stabilityai/stable-diffusion, Flux: https://huggingface.co/spaces/black-forest-labs/FLUX.1-schnell, etc.

Choose 3 different diffusion models. Tell us which ones you are using and provide websites/links.

With each model, generate 4 images for each of the following prompts:

1. An elephant above a motorcycle.
2. A teddy bear below a bed.
3. A chair above a knife.
4. A fork below a plate.
5. A bus above a banana.

Show your generated images for each model and each prompt.
(3 models $\times$ 5 prompts $\times$ 4 images $=$ 60 images total)

How many of these images have the correct spatial relationship as mentioned in the prompt (eg. for the prompt *"A teddy bear below a bed"* is the teddy bear actually generated below the bed?)

3.1. (EXTRA CREDIT) Quantitative Evaluation: VISOR scores *10 points*

In 2022, Gokhale et al [1] https://arxiv.org/abs/2212.10015 wrote a paper that proposed an evaluation metric for quantifying spatial reasoning abilities of text-to-image generation models. They defined an evaluation metric named "VISOR" which is explained in the paper. For each model in your above experiment, what are the VISOR unconditional and conditional scores?

References

[1] Tejas Gokhale, Hamid Palangi, Besmira Nushi, Vibhav Vineet, Eric Horvitz, Ece Kamar, Chitta Baral, and Yezhou Yang. Benchmarking spatial relationships in text-to-image generation. *arXiv preprint arXiv:2212.10015*, 2022. 6