COMP691 Assingment 2

Conccordia University
Zirui Qiu
40050008
leoqiuzirui@gmail.com

March 31, 2023

Question 1

a.

The formula to calculate receptive field:

$$RF = RF_{prev} + (kernel\_size - 1) \cdot jump$$

The center of the input is(114,114)
For layer 1 kernel: 11x11
stride: 4

$$RF = 1 + (11 - 1) * 1 = 1 + 10 * 1 = 11$$
$$Jump = 1(initial) * 4(stride) = 4$$
$$HalfRF = \frac{11}{2} = 5$$

Therefore, the receptive field with respect to input is:
Height1 = 114 - 5 = 109
Width1 = 114 - 5 = 109
Height2 = 114 + 5 = 119
Width2 = 114 + 5 = 119

For layer 2 kernel: 3x3
stride: 2

$$RF = 11 + (3 - 1) * 4 = 11 + 2 * 4 = 19$$
$$Jump = Jump_{prev} * stride = 4 * 2 = 8$$
$$HalfRF = \frac{19}{2} = 9$$

Therefore, the receptive field with respect to input is:
Height1 = 114 - 9 = 105
Width1 = 114 - 9 = 105
Height2 = 114 + 9 = 123
Width2 = 114 + 9 = 123

For layer 3 kernel: 5x5
stride: 1

$$RF = 19 + (5 - 1) * 8 = 11 + 4 * 8 = 51$$
$$HalfRF = \frac{51}{2} = 25$$

Therefore, the receptive field with respect to input is:
Height1 = 114 - 25 = 89
Width1 = 114 - 25 = 89
Height2 = 114 + 25 = 139
Width2 = 114 + 25 = 139

b.

[1]:
```python
import torch
model = torch.hub.load('pytorch/vision:v0.10.0', 'alexnet', pretrained=True)
model.eval()
```

```
/home/ziruiqiu/anaconda3/envs/DL2/lib/python3.9/site-packages/tqdm/auto.py:22:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
Downloading: "https://github.com/pytorch/vision/zipball/v0.10.0" to
/home/ziruiqiu/.cache/torch/hub/v0.10.0.zip
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to
/home/ziruiqiu/.cache/torch/hub/checkpoints/alexnet-owt-7be5be79.pth
100%|| 233M/233M [00:23<00:00, 10.3MB/s]
```

[1]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
```

```
)
```

[28]:
```
from PIL import Image
import matplotlib.pyplot as plt

def display_image(file_path):
    img = Image.open(file_path)
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```

[112]:
```
import torchvision.models as models
import torchvision.transforms as transforms




# Define transformation for input image
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

denormalize = transforms.Normalize(mean=[-0.485/0.229, -0.456/0.224, -0.406/0.
 ↪225], std=[1/0.229, 1/0.224, 1/0.225])

# Load class labels
with open('imagenet1000_clsidx_to_labels.txt') as f:
    class_labels = [line.strip() for line in f.readlines()]

def predict(image_path):
    # Load input image
    image = Image.open(image_path).convert('RGB')

    # Apply transformation to input image
    input_tensor = transform(image)
    input_batch = input_tensor.unsqueeze(0)
    #print(input_tensor.shape)
    # Make prediction on input image
    with torch.no_grad():
        output = model(input_batch)

    # Get index of top prediction
    _, index = torch.max(output, 1)
```

```
    # Get class label for top prediction
    label = class_labels[index]

    # Print top prediction label
    print("Top prediction: ", label)

predict('dog.jpg')
display_image('dog.jpg')
predict('jellyfish.jpg')
display_image('jellyfish.jpg')
predict('frog.jpg')
display_image('frog.jpg')
```

```
Top prediction:  218: 'Welsh springer spaniel',
```

```
Top prediction:  107: 'jellyfish',
```

```
Top prediction:  31: 'tree frog, tree-frog',
```

c.

[101]:
```
import numpy as np
def generate_adversarial_example(model, image_path, target_classes, alpha=0.
↪00001, num_steps=1000):
    # Load input image
    image = Image.open(image_path).convert('RGB')

    # Apply transformation to input image
    input_tensor = transform(image)
    input_batch = input_tensor.unsqueeze(0)

    # Get predicted class label for input image
    with torch.no_grad():
        output = model(input_batch)
    _, index = torch.max(output, 1)
    true_class = index.item()

    # Select target class to optimize for
    target_class = target_classes[np.random.randint(0, len(target_classes))]

    # Define optimizer and loss function
    optimizer = torch.optim.Adam([input_batch.requires_grad_()], lr=0.01)
    loss_fn = torch.nn.CrossEntropyLoss()
    # Run optimization
    for i in range(num_steps):
        # Compute loss
        output = model(input_batch)
        loss = alpha * torch.norm(input_batch.view(-1), p=1) + loss_fn(output,␣
↪torch.tensor([target_class]))
        #print(torch.norm(input_batch.view(-1), p=1))
        # Update input tensor
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Check if target class has been reached
        _, index = torch.max(output, 1)
        predicted_class = index.item()
        if predicted_class == target_class:
            break

    # Get predicted class label for adversarial example
    with torch.no_grad():
        output = model(input_batch)
```

```
    _, index = torch.max(output, 1)
    adversarial_class = index.item()
    # Print true class, target class, and adversarial class
    print("True class: ", class_labels[true_class])
    print("Target class: ", class_labels[target_class])
    print("Adversarial class: ", class_labels[adversarial_class])

    # Convert input tensor back to PIL image
    #adversarial_image = transforms.ToPILImage()(input_batch.squeeze())

    return input_batch, class_labels[adversarial_class]
```

[102]:
```
for i in range(3):
    adversarial_image, adversarial_class = generate_adversarial_example(model,␣
 ↪'dog.jpg', [i])
    adversarial_image = denormalize((adversarial_image)).clamp(0, 1)
    # Convert input tensor back to PIL image
    adversarial_image = transforms.ToPILImage()(adversarial_image.squeeze(0))
    plt.imshow(adversarial_image)
    plt.title(adversarial_class)
    plt.axis('off')
    plt.show()
```

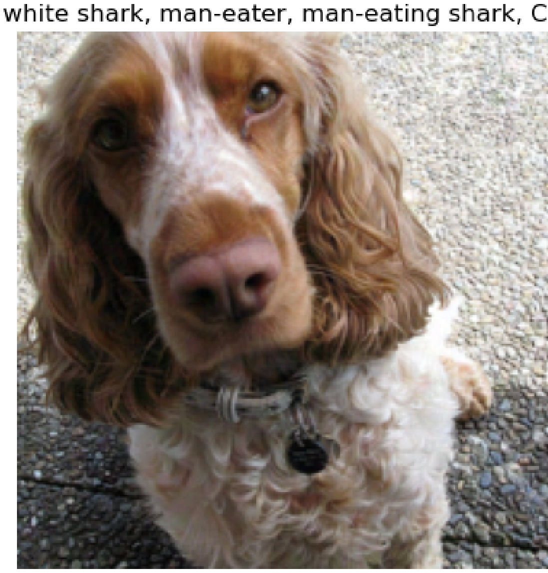
```
True class:  218: 'Welsh springer spaniel',
Target class:  {0: 'tench, Tinca tinca',
Adversarial class:  {0: 'tench, Tinca tinca',
```


```
True class:  218: 'Welsh springer spaniel',
Target class:  1: 'goldfish, Carassius auratus',
Adversarial class:  1: 'goldfish, Carassius auratus',
```

1: 'goldfish, Carassius auratus',

True class: 218: 'Welsh springer spaniel',
Target class: 2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias',
Adversarial class: 2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias',

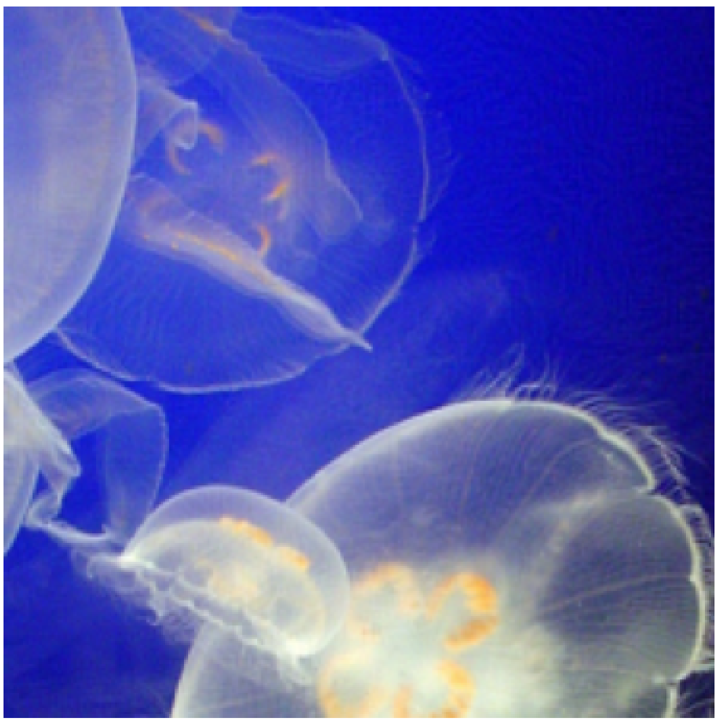
2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias',



[103]:
```
for i in range(3, 6):
    adversarial_image, adversarial_class = generate_adversarial_example(model,␣
 ↪'jellyfish.jpg', [i])
    adversarial_image = denormalize((adversarial_image)).clamp(0, 1)
    # Convert input tensor back to PIL image
    adversarial_image = transforms.ToPILImage()(adversarial_image.squeeze(0))
    plt.imshow(adversarial_image)
    plt.title(adversarial_class)
    plt.axis('off')
    plt.show()
```

```
True class:  107: 'jellyfish',
Target class:  3: 'tiger shark, Galeocerdo cuvieri',
Adversarial class:  3: 'tiger shark, Galeocerdo cuvieri',
```

3: 'tiger shark, Galeocerdo cuvieri',



```
True class:  107: 'jellyfish',
Target class:  4: 'hammerhead, hammerhead shark',
Adversarial class:  4: 'hammerhead, hammerhead shark',
```

4: 'hammerhead, hammerhead shark',

```
True class:  107: 'jellyfish',
Target class:  5: 'electric ray, crampfish, numbfish, torpedo',
Adversarial class:  5: 'electric ray, crampfish, numbfish, torpedo',
```

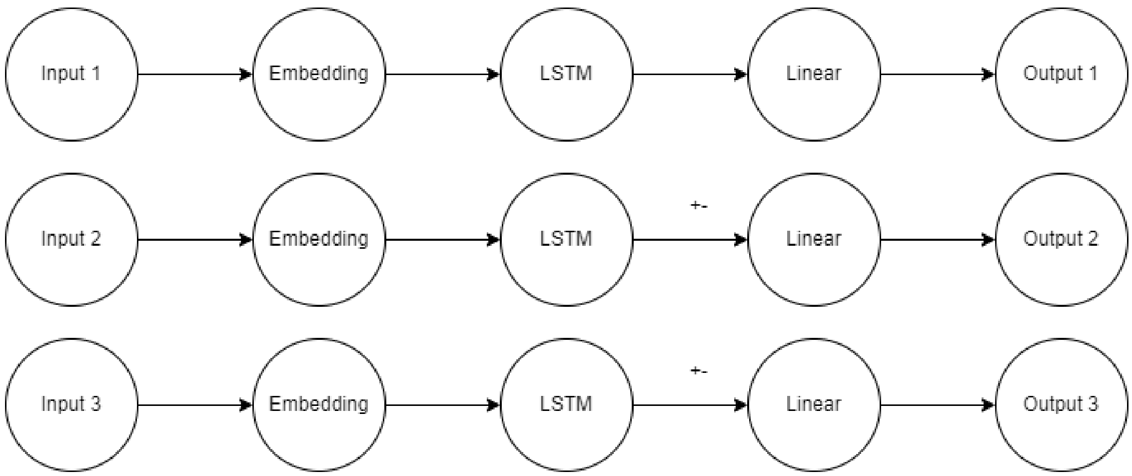
5: 'electric ray, crampfish, numbfish, torpedo',

Questions 2

Q2.1 Detaching or not?

1. When not using truncated backprop through time, the model would backpropagate through the entire sequence, making it computationally expensive and prone to vanishing or exploding gradients. By implementing TBPTT, the model backpropagates through a fixed number of steps (controlled by seq_length), which makes the training more efficient.

2. Assume the model has one layer and a sequence length of 3. We can represent the graph as follows:



-
- In this graph, the LSTM layer receives inputs from the Embedding layer and is connected to the Linear layer. The outputs from the Linear layer are the predictions at each time step. The arrows represent the flow of information and gradients during forward and backward passes.
- When we use truncated backpropagation through time, we limit the number of time steps that gradients are backpropagated through. In this example, the gradients would only flow through the LSTM connections up to a certain number of time steps (e.g., 3 steps in this case).
- The detach() function is used to stop gradients from flowing further back in time than the specified number of time steps. In this example, if we apply detach() after 3 time steps, the gradients will not be backpropagated beyond the third time step. This reduces the amount of computation and memory required during training and can help prevent the vanishing gradient problem in long sequences.

3. Detaching the hidden states breaks the computational graph, and the memory associated with the previous states is freed up. When the computational graph is not detached, it keeps track of all the historical states and their gradients, which increases the memory consumption. By using TBPTT and detaching the hidden states, we can prevent excessive memory usage and make the training process more efficient.

Model Prediction

Below we will use our model to generate text sequence!

[38]:
```
# Sample from the model
with torch.no_grad():
    with open('sample.txt', 'w') as f:
        # Set intial hidden ane cell states
        state = (torch.zeros(num_layers, 1, hidden_size).to(device),
                 torch.zeros(num_layers, 1, hidden_size).to(device))

        # Select one word id randomly
        prob = torch.ones(vocab_size)
        input = torch.multinomial(prob, num_samples=1).unsqueeze(1).to(device)

        for i in range(num_samples):
            # Forward propagate RNN
            output, state = model(input, state)

            # Sample a word id
            prob = output.exp()
            word_id = torch.multinomial(prob, num_samples=1).item()

            # Fill input with sampled word id for the next time step
            input.fill_(word_id)

            # File write
            word = corpus.dictionary.idx2word[word_id]
            word = '\n' if word == '<eos>' else word + ' '
            f.write(word)

            if (i+1) % 100 == 0:
                print('Sampled [{}/{}] words and save to {}'.format(i+1,␣
↪num_samples, 'sample.txt'))
! cat sample.txt
```

```
to <unk> up footing the union leader says <unk> <unk> senior vice president at
royal university
professor space roger student strongly matter was a crucial disappointing in los
angeles that led to the <unk> and <unk> of ronald reagan 's journal
confidence players <unk> and great together a
```

Q2.2 Sampling strategy

[15]:
```
# Sample greedily from the model
with torch.no_grad():
    with open('sample_greedy.txt', 'w') as f:
        # Set intial hidden ane cell states
```

```
        state = (torch.zeros(num_layers, 1, hidden_size).to(device),
                 torch.zeros(num_layers, 1, hidden_size).to(device))

        # Select one word id randomly
        prob = torch.ones(vocab_size)
        input = torch.multinomial(prob, num_samples=1).unsqueeze(1).to(device)

        for i in range(num_samples):
            # Forward propagate RNN
            output, state = model(input, state)

            # Sample a word id
            word_id = output.argmax(1).item()

            # Fill input with sampled word id for the next time step
            input.fill_(word_id)

            # File write
            word = corpus.dictionary.idx2word[word_id]
            word = '\n' if word == '<eos>' else word + ' '
            f.write(word)

            if (i+1) % 100 == 0:
                print('Sampled [{}/{}] words and save to {}'.format(i+1,␣
↪num_samples, 'sample_greedy.txt'))

! cat sample_greedy.txt
```

```
in the u.s. august
the company said the acquisition is subject to a definitive agreement by <unk>
its <unk> subsidiary of <unk> <unk> inc. a <unk> calif. <unk> subsidiary
the <unk> <unk> company said it is seeking to sell its <unk> unit and <unk>
coors co. of america
```

Q2.3 Embedding Distance

[18]:
```
# Embedding distance
import torch.nn.functional as F

def cosine_distance(word1, word2, model):
    # Get the word indices
    idx1 = corpus.dictionary.word2idx[word1]
    idx2 = corpus.dictionary.word2idx[word2]

    # Get the word embeddings from the model
    embed1 = model.embed(torch.tensor(idx1).to(device)).detach().cpu()
    embed2 = model.embed(torch.tensor(idx2).to(device)).detach().cpu()
```

```
    # Calculate the cosine similarity between the two embeddings
    cos_sim = F.cosine_similarity(embed1, embed2, dim=0)

    # Calculate the cosine distance as 1 - cosine similarity
    cos_dist = 1 - cos_sim

    return cos_dist

word1 = "army"
word2 = "taxpayer"
cos_dist = cosine_distance(word1, word2, model)
print("Cosine distance between '{}' and '{}': {:.4f}".format(word1, word2,␣
 ↪cos_dist))
```

```
Cosine distance between 'army' and 'taxpayer': 0.9321
```

Q2.4 Teacher Forcing (Extra Credit 2 points)

[13]:
```
# Training code with Teacher Forcing
# Modified RNNLM model for step-by-step training without teacher forcing
class RNNLM_no_teacher_forcing(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
        super(RNNLM_no_teacher_forcing, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, h):
        # Embed word ids to vectors
        x = self.embed(x)

        # Forward propagate LSTM
        out, (h, c) = self.lstm(x, h)

        # Decode hidden states of all time steps
        out = self.linear(out.squeeze(1))
        return out, (h, c)

model_no_tf = RNNLM_no_teacher_forcing(vocab_size, embed_size, hidden_size,␣
 ↪num_layers).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_no_tf.parameters(), lr=learning_rate)
```

```
# Train the model without teacher forcing
for epoch in range(num_epochs):
    # Set initial hidden and cell states
    states = (torch.zeros(num_layers, batch_size, hidden_size).to(device),
              torch.zeros(num_layers, batch_size, hidden_size).to(device))

    for i in range(0, ids.size(1) - seq_length, seq_length):
        # Get mini-batch inputs and targets
        inputs = ids[:, i:i+1].to(device)  # Only take the first input of the␣
↪sequence
        targets = ids[:, (i+1):(i+1)+seq_length].to(device)

        # Initialize the cumulative loss for this sequence
        cumulative_loss = 0

        # Loop through the sequence
        for j in range(seq_length):
            # Forward pass
            states = detach(states)
            outputs, states = model_no_tf(inputs, states)

            # Calculate the loss
            loss = criterion(outputs, targets[:, j])
            cumulative_loss += loss.item()

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            clip_grad_norm_(model_no_tf.parameters(), 0.5)
            optimizer.step()

            # Use the predicted output as input for the next time step
            inputs = outputs.argmax(dim=1).unsqueeze(1).to(device)

        step = (i+1) // seq_length
        if step % 100 == 0:
            avg_loss = cumulative_loss / seq_length
            print('Epoch [{}/{}], Step[{}/{}], Loss: {:.4f}, Perplexity: {:5.2f}'
                .format(epoch+1, num_epochs, step, num_batches, avg_loss, np.
↪exp(avg_loss)))
```

```
Epoch [1/5], Step[0/1549], Loss: 8.4087, Perplexity: 4486.13
Epoch [1/5], Step[100/1549], Loss: 6.5210, Perplexity: 679.25
Epoch [1/5], Step[200/1549], Loss: 6.7618, Perplexity: 864.21
Epoch [1/5], Step[300/1549], Loss: 6.7580, Perplexity: 860.89
Epoch [1/5], Step[400/1549], Loss: 6.6612, Perplexity: 781.50
Epoch [1/5], Step[500/1549], Loss: 6.6264, Perplexity: 754.74
Epoch [1/5], Step[600/1549], Loss: 6.4988, Perplexity: 664.34
```

```
Epoch [1/5], Step[700/1549], Loss: 6.8312, Perplexity: 926.29
Epoch [1/5], Step[800/1549], Loss: 6.5867, Perplexity: 725.40
Epoch [1/5], Step[900/1549], Loss: 6.9411, Perplexity: 1033.87
Epoch [1/5], Step[1000/1549], Loss: 6.7880, Perplexity: 887.12
Epoch [1/5], Step[1100/1549], Loss: 6.9334, Perplexity: 1025.93
Epoch [1/5], Step[1200/1549], Loss: 6.6696, Perplexity: 788.05
Epoch [1/5], Step[1300/1549], Loss: 6.9659, Perplexity: 1059.84
Epoch [1/5], Step[1400/1549], Loss: 6.7953, Perplexity: 893.66
Epoch [1/5], Step[1500/1549], Loss: 6.8082, Perplexity: 905.27
Epoch [2/5], Step[0/1549], Loss: 7.3388, Perplexity: 1538.88
Epoch [2/5], Step[100/1549], Loss: 6.4142, Perplexity: 610.44
Epoch [2/5], Step[200/1549], Loss: 6.5203, Perplexity: 678.78
Epoch [2/5], Step[300/1549], Loss: 6.6017, Perplexity: 736.34
Epoch [2/5], Step[400/1549], Loss: 6.4344, Perplexity: 622.91
Epoch [2/5], Step[500/1549], Loss: 6.2837, Perplexity: 535.78
Epoch [2/5], Step[600/1549], Loss: 6.3646, Perplexity: 580.92
Epoch [2/5], Step[700/1549], Loss: 6.7754, Perplexity: 876.06
Epoch [2/5], Step[800/1549], Loss: 6.5647, Perplexity: 709.62
Epoch [2/5], Step[900/1549], Loss: 6.6884, Perplexity: 803.07
Epoch [2/5], Step[1000/1549], Loss: 6.5059, Perplexity: 669.07
Epoch [2/5], Step[1100/1549], Loss: 6.7770, Perplexity: 877.39
Epoch [2/5], Step[1200/1549], Loss: 6.4880, Perplexity: 657.21
Epoch [2/5], Step[1300/1549], Loss: 6.7233, Perplexity: 831.53
Epoch [2/5], Step[1400/1549], Loss: 6.6928, Perplexity: 806.58
Epoch [2/5], Step[1500/1549], Loss: 6.6925, Perplexity: 806.37
Epoch [3/5], Step[0/1549], Loss: 7.0970, Perplexity: 1208.36
Epoch [3/5], Step[100/1549], Loss: 6.2202, Perplexity: 502.83
Epoch [3/5], Step[200/1549], Loss: 6.2394, Perplexity: 512.55
Epoch [3/5], Step[300/1549], Loss: 6.4985, Perplexity: 664.13
Epoch [3/5], Step[400/1549], Loss: 6.3901, Perplexity: 595.89
Epoch [3/5], Step[500/1549], Loss: 6.1349, Perplexity: 461.70
Epoch [3/5], Step[600/1549], Loss: 6.2869, Perplexity: 537.47
Epoch [3/5], Step[700/1549], Loss: 6.6364, Perplexity: 762.38
Epoch [3/5], Step[800/1549], Loss: 6.4524, Perplexity: 634.23
Epoch [3/5], Step[900/1549], Loss: 6.5398, Perplexity: 692.13
Epoch [3/5], Step[1000/1549], Loss: 6.4948, Perplexity: 661.67
Epoch [3/5], Step[1100/1549], Loss: 6.7849, Perplexity: 884.36
Epoch [3/5], Step[1200/1549], Loss: 6.3687, Perplexity: 583.30
Epoch [3/5], Step[1300/1549], Loss: 6.7048, Perplexity: 816.32
Epoch [3/5], Step[1400/1549], Loss: 6.6204, Perplexity: 750.25
Epoch [3/5], Step[1500/1549], Loss: 6.7342, Perplexity: 840.67
Epoch [4/5], Step[0/1549], Loss: 7.1596, Perplexity: 1286.36
Epoch [4/5], Step[100/1549], Loss: 6.1672, Perplexity: 476.83
Epoch [4/5], Step[200/1549], Loss: 6.3235, Perplexity: 557.54
Epoch [4/5], Step[300/1549], Loss: 6.5372, Perplexity: 690.33
Epoch [4/5], Step[400/1549], Loss: 6.2792, Perplexity: 533.37
Epoch [4/5], Step[500/1549], Loss: 6.2083, Perplexity: 496.85
Epoch [4/5], Step[600/1549], Loss: 6.2896, Perplexity: 538.92
```

```
Epoch [4/5], Step[700/1549], Loss: 6.5763, Perplexity: 717.91
Epoch [4/5], Step[800/1549], Loss: 6.3302, Perplexity: 561.28
Epoch [4/5], Step[900/1549], Loss: 6.4013, Perplexity: 602.60
Epoch [4/5], Step[1000/1549], Loss: 6.4701, Perplexity: 645.55
Epoch [4/5], Step[1100/1549], Loss: 6.5227, Perplexity: 680.39
Epoch [4/5], Step[1200/1549], Loss: 6.2732, Perplexity: 530.20
Epoch [4/5], Step[1300/1549], Loss: 6.7449, Perplexity: 849.71
Epoch [4/5], Step[1400/1549], Loss: 6.5712, Perplexity: 714.26
Epoch [4/5], Step[1500/1549], Loss: 6.5846, Perplexity: 723.89
Epoch [5/5], Step[0/1549], Loss: 7.2840, Perplexity: 1456.75
Epoch [5/5], Step[100/1549], Loss: 6.1739, Perplexity: 480.06
Epoch [5/5], Step[200/1549], Loss: 6.2192, Perplexity: 502.28
Epoch [5/5], Step[300/1549], Loss: 6.7737, Perplexity: 874.59
Epoch [5/5], Step[400/1549], Loss: 6.3300, Perplexity: 561.16
Epoch [5/5], Step[500/1549], Loss: 6.1504, Perplexity: 468.92
Epoch [5/5], Step[600/1549], Loss: 6.3650, Perplexity: 581.14
Epoch [5/5], Step[700/1549], Loss: 6.4883, Perplexity: 657.38
Epoch [5/5], Step[800/1549], Loss: 6.2693, Perplexity: 528.10
Epoch [5/5], Step[900/1549], Loss: 6.3427, Perplexity: 568.33
Epoch [5/5], Step[1000/1549], Loss: 6.4073, Perplexity: 606.24
Epoch [5/5], Step[1100/1549], Loss: 6.4607, Perplexity: 639.48
Epoch [5/5], Step[1200/1549], Loss: 6.2769, Perplexity: 532.16
Epoch [5/5], Step[1300/1549], Loss: 6.7547, Perplexity: 858.12
Epoch [5/5], Step[1400/1549], Loss: 6.6242, Perplexity: 753.08
Epoch [5/5], Step[1500/1549], Loss: 6.5663, Perplexity: 710.75
```

Q2.5 Distance Comparison

[14]:
```
word1 = "army"
word2 = "taxpayer"

# Calculate cosine distances for both models
cos_dist_tf = cosine_distance(word1, word2, model)  # With teacher forcing
cos_dist_no_tf = cosine_distance(word1, word2, model_no_tf)  # Without teacher forcing

# Print the results
print("Cosine distance with teacher forcing: {:.4f}".format(cos_dist_tf))
print("Cosine distance without teacher forcing: {:.4f}".format(cos_dist_no_tf))
```

```
Cosine distance with teacher forcing: 0.9321
Cosine distance without teacher forcing: 1.1265
```

Discussion:

The model with teacher forcing has a smaller cosine distance (0.9321) compared to the model without teacher forcing (1.1265).

This suggests that the learned representations of words in the embedding space are influenced by the training method. In this case, the model trained with teacher forcing seems to learn word embeddings that place “army” and “taxpayer” closer together in the embedding space than the model trained without teacher forcing.

It’s important to note that this observation is based on a single pair of words, and further analysis would be needed to draw more general conclusions about the effect of teacher forcing on word embeddings. However, this example demonstrates that the choice of training method can have an impact on the learned representations of words.

Question 3

a.

a is question assumption.

b.

We assume A is the attention score matrix computed by the self-attention operation S(X) for the input sequence X, and let A' be the attention score matrix computed by the self-attention operation S(PX) for the input sequence PX. Then we have:

$$A = \frac{1}{\alpha} X W_q W_k^T X^T \tag{1}$$

$$A' = \frac{1}{\alpha} P X W_q W_k^T (PX)^T \tag{2}$$

To show that P S(X) = S(PX), we need to show that P A = A' P. Let us consider the product P A:

$$PA = P(\frac{1}{\alpha} X W_q W_k^T X^T) = (\frac{1}{\alpha} P X W_q W_k^T (PX)^T) P = A'P \tag{3}$$

Therefore, for any permutation matrix P, P S(X) = S(PX), which means that the self-attention operation is permutation-invariant.

c.

From previous question, we have:

$$PS(X) = S(PX)$$

and

$$G(PX) = w^T S(PX) = w^T PS(X)$$

For a specific example, we can define $w^T = [1, 1, 1]$ and we have:

$$G(PX) = w^T PS(X) = [1, 1, 1]PS(X)$$

The operation here is taking the summation of all rows. In this situation, order of rows is not important, so the permutation can be ignored. To conclude, we have:

$$G(PX) = w^T PS(X) = w^T S(X) = G(X)$$

d. Using nn.Linear

[117]:
```
import torch.nn as nn
import torch.nn.functional as F

class PositionwiseFeedforward(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(PositionwiseFeedforward, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```python
        self.fc2 = nn.Linear(hidden_size, input_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

import torch

B, T, D = 5, 10, 20
Z = torch.randn(B, T, D)

ffn = PositionwiseFeedforward(D, D*4)
output = ffn(Z)

assert output.shape == (B, T, D), "Output shape is incorrect"
print("Output shape is correct")
```

```
Output shape is correct
```

d. Using nn.Conv1d

[118]:
```python
class PositionwiseFeedforwardConv(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(PositionwiseFeedforwardConv, self).__init__()
        self.conv1 = nn.Conv1d(input_size, hidden_size, kernel_size=1)
        self.conv2 = nn.Conv1d(hidden_size, input_size, kernel_size=1)

    def forward(self, x):
        x = x.permute(0, 2, 1)  # transpose from (B, T, D) to (B, D, T)
        x = F.relu(self.conv1(x))
        x = self.conv2(x)
        x = x.permute(0, 2, 1)  # transpose back to (B, T, D)
        return x
ffn_conv = PositionwiseFeedforwardConv(D, D*4)
output_conv = ffn_conv(Z)

assert output_conv.shape == (B, T, D), "output_conv shape is incorrect"
print("output_conv shape is correct")
```

```
output_conv shape is correct
```