Question 1

1. For layer 1, the receptive field of the center position w.r.t. the input image is the kernel size of layer 1 : $RF_{1\to img} = K_1 = 11$, where RF is the side length of the receptive field, and K is kernel size. Thus, the receptive field rectangle on the input image can be expressed as $(114 - \frac{11-1}{2}, 114 - \frac{11-1}{2})$ (lower left corner), $(114 + \frac{11-1}{2}, 114 + \frac{11-1}{2})$ (upper right corner), that is $(109, 109), (119, 119)$.

   For layer 2, the receptive field of the center position w.r.t. the layer 1 is : $RF_{2\to 1} = KS_2 = 3$. Then, the receptive field w.r.t. the input image is the receptive field of the 3×3 pixels on layer 1 w.r.t. input image : $RF_{2\to img} = KS_1 + (RF_{2\to 1} - 1) \times S_1 = 19$, where S is stride. Thus, the receptive field rectangle can be expressed as : $(114 - \frac{19-1}{2}, 114 - \frac{19-1}{2})$, that is $(105, 105), (123, 123)$.

   For layer 3, similarly, $RF_{3\to 2} = KS_3 = 5$, $RF_{3\to 1} = KS_2 + (RF_{3\to 2} - 1) \times S_2 = 3 + (5-1) \times 2 = 11$, $RF_{3\to img} = KS_1 + (RF_{3\to 1} - 1) \times S_1 = 11 + (11-1) \times 4 = 51$. Thus, the receptive field rectangle can be expressed as : $(114 - \frac{51-1}{2}, 114 - \frac{51-1}{2})$, that is $(89, 89), (139, 139)$.

The code part starts on the next page.

```
%reset
import torch
import urllib.request
from PIL import Image
from torchvision import models, transforms
import requests
from io import BytesIO
import matplotlib.pyplot as plt


# Load the class labels file
response =
requests.get("https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3a/raw/238f720ff059c1f82f368259d1ca4ffa5dd8f9f5/imagenet1000_clsidx_to_labels.txt")
classes = []
for label in response.text.splitlines():
    classes.append(label)
#     classes.append(label)

# Load the model
model = models.alexnet(pretrained=True)
model.eval()

# Define the preprocessing steps for the input image
preprocess = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Define a function to classify a single image
def classify_image(image, preprocess_=True):
    if preprocess_:
        image_tensor = preprocess(image)
        image_tensor = image_tensor.unsqueeze(0)  # Add a batch dimension
    else:
        image_tensor = image
    # Pass the image through the model
    with torch.no_grad():
        output = model(image_tensor)

    # Get the top prediction and its score
    _, index = torch.max(output, 1)
    score = torch.nn.functional.softmax(output, dim=1)[0, index]

    # Return the class label and score as strings
```

```
    class_label = classes[index]
    score = score.item()
    return class_label, score

# Test the function on some sample images
#
https://github.com/ajschumacher/imagen/blob/master/imagen/n00007846_147031_person.jpg
image_paths = [

 "https://github.com/ajschumacher/imagen/raw/master/imagen/n01776313_1303_tick.jpg",

 "https://github.com/ajschumacher/imagen/raw/master/imagen/n01443537_2625_goldfish.jpg",

 "https://github.com/ajschumacher/imagen/raw/master/imagen/n01944390_3990_snail.jpg"
]

for image_path in image_paths:
    print("Image:", image_path)
    # Load the image and apply preprocessing
    response = requests.get(image_path)
    image = Image.open(BytesIO(response.content))
    plt.imshow(image)
    plt.axis("off")

    class_label, score = classify_image(image)

    # Print the top prediction and its score
    print("Prediction:", class_label)
    print("Score:", score)
    plt.show()
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

```
/home/wa_ziqia/miniconda3/envs/deepl2023/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/wa_ziqia/miniconda3/envs/deepl2023/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=AlexNet_Weights.IMAGENET1K_V1`. You can also use `weights=AlexNet_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
Image:
https://github.com/ajschumacher/imagen/raw/master/imagen/n01776313_1303_tick.jpg
Prediction:  78: 'tick',
Score: 0.8674294948577881
```

```
Image:
https://github.com/ajschumacher/imagen/raw/master/imagen/n01443537_2625_goldfish.jpg
Prediction:  1: 'goldfish, Carassius auratus',
Score: 0.8977013826370239
```

```
Image:
https://github.com/ajschumacher/imagen/raw/master/imagen/n01944390_3990_snail.jpg
Prediction:  113: 'snail',
Score: 0.5390090346336365
```

```
import torch.optim as optim
import numpy as np
import random
from torchvision.transforms.functional import to_pil_image

def unnormalize(tensor):
    mean = torch.tensor([0.485, 0.456, 0.406]).view(-1, 1, 1)
    std = torch.tensor([0.229, 0.224, 0.225]).view(-1, 1, 1)
    unnormalized_tensor = tensor * std + mean
    return unnormalized_tensor

def tensor_to_image(tensor):
    unnormalized_tensor = unnormalize(tensor)
    return to_pil_image(unnormalized_tensor.clamp(0, 1))

# Define the objective function
def objective(r, alpha, model_output, target):
    l1_norm = torch.norm(r, 1)
    loss = torch.nn.functional.cross_entropy(model_output, target)
    return alpha * l1_norm + loss

# Function to create an adversarial example
def adversarial_example(image_tensor, model, true_class, target_class, alpha,
learning_rate=0.01, max_iter=300):
    r = torch.zeros_like(image_tensor, requires_grad=True, device='cuda')
    optimizer = optim.Adam([r], lr=learning_rate)
```

```
    for i in range(max_iter):
        model_output = model(image_tensor + r)
        _, predicted_class = torch.max(model_output, 1)
        if predicted_class.item() == target_class:
            break

        optimizer.zero_grad()
        loss = objective(r, alpha, model_output, torch.tensor([target_class],
device='cuda'))
        loss.backward()
        optimizer.step()

    return image_tensor + r

# Select two random images and three alternative classes for each
random_image_paths = random.sample(image_paths, k=2)
target_classes = [[500, 550, 700], [600, 650, 750]]

# Create adversarial examples for each image
model.to('cuda')
adversarial_examples = []
for i, image_path in enumerate(random_image_paths):
    image_tensor =
preprocess(Image.open(BytesIO(requests.get(image_path).content))).unsqueeze(0).
to('cuda')
    true_class = torch.argmax(model(image_tensor)).item()
    for target_class in target_classes[i]:
        adv_example = adversarial_example(image_tensor, model, true_class,
target_class, alpha=0.001)
        adversarial_examples.append((image_path, adv_example))

# Visualize original and adversarial examples and labels found by AlexNet
for image_path, adv_example in adversarial_examples:
    # Show original image
    original_image = Image.open(BytesIO(requests.get(image_path).content))
    plt.imshow(original_image)
    plt.axis("off")
    plt.title("Original Image")
    plt.show()

    # Show adversarial example
    adv_image = tensor_to_image(adv_example.squeeze(0).detach().cpu())
    plt.imshow(adv_image)
    plt.axis("off")
    plt.title("Adversarial Example")
    plt.show()

    # Print labels found by AlexNet for original and adversarial images
```

```
model.to('cpu')
true_label, _ = classify_image(original_image)
print("True label:", true_label)
adv_label, _ = classify_image(adv_example.cpu(), preprocess_=False)
print("Adversarial label:", adv_label)
```

```
True label:  113: 'snail',
Adversarial label:  500: 'cliff dwelling',
```

Original Image

Adversarial Example

```
True label:  113: 'snail',
Adversarial label:  550: 'espresso maker',
```

```
True label:  113: 'snail',
Adversarial label:  700: 'paper towel',
```


```
True label:  78: 'tick',
Adversarial label:  600: 'hook, claw',
```

Original Image

Adversarial Example

```
True label:  78: 'tick',
Adversarial label:  650: 'microphone, mike',
```


```
True label:  78: 'tick',
Adversarial label:  750: 'quilt, comforter, comfort, puff',
```

Question 2

1. (1) Truncated backpropagation through time (TBPTT) is a technique used to train recurrent neural networks (RNNs) by limiting the number of timesteps over which gradients are backpropagated. Compared with regular BPTT, TBPTT reduce the computational complexity and memory requirements during training, as well as to mitigate the vanishing and exploding gradient problems.

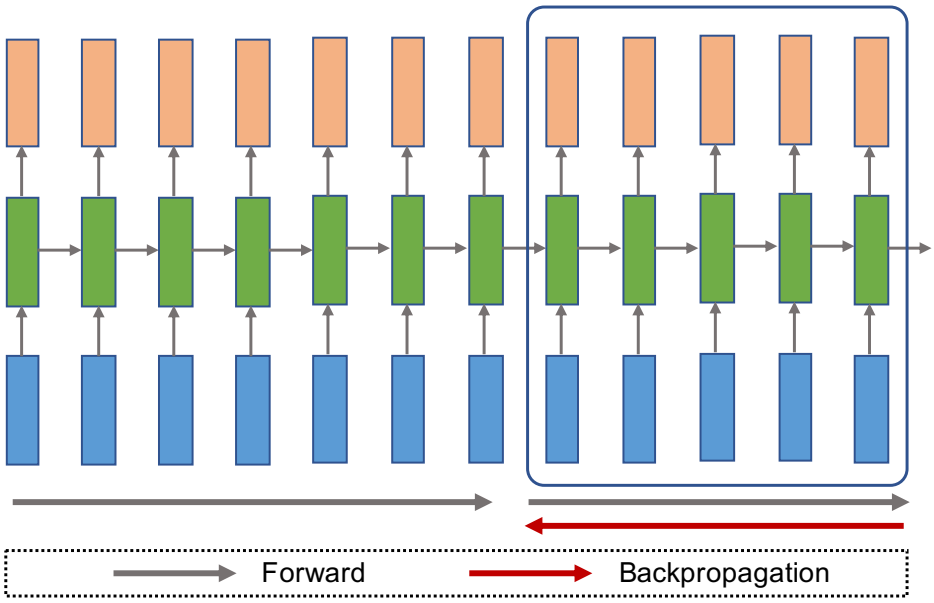

FIGURE 1 – Truncated backpropagation through time.

(2) As shown in Figure 1 , the purpose of this function is to detach the hidden states from their computational graph, preventing gradients from being backpropagated beyond the current sequence length. This results in a shorter computational graph and less memory usage.

(3) By using the detach() function and limiting the backpropagation to a fixed sequence length, the implementation consumes less GPU memory during training. This is because the computational graph is shorter, and fewer gradients need to be stored and updated. This not only reduces the memory requirements but also helps improve training efficiency.

The code part starts on the next page.

🦜 NN-Based Language Model

In this excercise we will run a basic RNN based language model and answer some questions about the code. It is advised to use GPU to run the code. First run the code then answer the questions below that require modifying it.

```
#@title 🧱 Imports & Hyperparameter Setup
#@markdown Feel free to experiment with the following hyperparameters at your
#@markdown leasure. For the purpose of this assignment, leave the default
values
#@markdown and run the code with these suggested values.
# Some part of the code was referenced from below.
# https://github.com/pytorch/examples/tree/master/word_language_model
# https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/02-
intermediate/language_model

! git clone https://github.com/yunjey/pytorch-tutorial/
%cd pytorch-tutorial/tutorials/02-intermediate/language_model/

import torch
import torch.nn as nn
import numpy as np
from torch.nn.utils import clip_grad_norm_

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
embed_size = 128 #@param {type:"number"}
hidden_size = 1024 #@param {type:"number"}
num_layers = 1 #@param {type:"number"}
num_epochs = 5 #@param {type:"slider", min:1, max:10, step:1}
batch_size = 20 #@param {type:"number"}
seq_length = 30 #@param {type:"number"}
learning_rate = 0.002 #@param {type:"number"}
#@markdown Number of words to be sampled ⬇️
num_samples = 50 #@param {type:"number"}

print(f"--> Device selected: {device}")
```

```
Cloning into 'pytorch-tutorial'...
remote: Enumerating objects: 917, done.•[K
remote: Total 917 (delta 0), reused 0 (delta 0), pack-reused 917•[K
Receiving objects: 100% (917/917), 12.80 MiB | 953.00 KiB/s, done.
Resolving deltas: 100% (490/490), done.
/home/wa_ziqia/Documents/assignments/COMP691/assignment2/pytorch-
tutorial/tutorials/02-intermediate/language_model
--> Device selected: cuda
```

```
from data_utils import Dictionary, Corpus

# Load "Penn Treebank" dataset
corpus = Corpus()
ids = corpus.get_data('data/train.txt', batch_size)
vocab_size = len(corpus.dictionary)
num_batches = ids.size(1) // seq_length

print(f"Vcoabulary size: {vocab_size}")
print(f"Number of batches: {num_batches}")
```

```
Vcoabulary size: 10000
Number of batches: 1549
```

🤖 Model Definition

As you can see below, this model stacks `num_layers` many LSTM units vertically to construct our basic RNN-based language model. The diagram below shows a pictorial representation of the model in its simplest form (i.e `num_layers`=1).

```
# RNN based language model
class RNNLM(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
        super(RNNLM, 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)
```

```
        # Reshape output to (batch_size*sequence_length, hidden_size)
        out = out.reshape(out.size(0)*out.size(1), out.size(2))

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

🏓 Training

In this section we will train our model, this should take a couple of minutes! Be patient 😊

```
model = RNNLM(vocab_size, embed_size, hidden_size, num_layers).to(device)

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

# Truncated backpropagation
def detach(states):
    return [state.detach() for state in states]


# Train the model
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+seq_length].to(device)
        targets = ids[:, (i+1):(i+1)+seq_length].to(device)

        # Forward pass
        states = detach(states)
        outputs, states = model(inputs, states)
        loss = criterion(outputs, targets.reshape(-1))

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

        step = (i+1) // seq_length
        if step % 100 == 0:
```

```
            print ('Epoch [{}/{}], Step[{}/{}], Loss: {:.4f}, Perplexity:
{:5.2f}'
                   .format(epoch+1, num_epochs, step, num_batches, loss.item(),
np.exp(loss.item())))
```

```
Epoch [1/5], Step[0/1549], Loss: 9.2125, Perplexity: 10021.20
Epoch [1/5], Step[100/1549], Loss: 5.9930, Perplexity: 400.62
Epoch [1/5], Step[200/1549], Loss: 5.9335, Perplexity: 377.48
Epoch [1/5], Step[300/1549], Loss: 5.7679, Perplexity: 319.86
Epoch [1/5], Step[400/1549], Loss: 5.6859, Perplexity: 294.68
Epoch [1/5], Step[500/1549], Loss: 5.1088, Perplexity: 165.47
Epoch [1/5], Step[600/1549], Loss: 5.1871, Perplexity: 178.95
Epoch [1/5], Step[700/1549], Loss: 5.3405, Perplexity: 208.62
Epoch [1/5], Step[800/1549], Loss: 5.1862, Perplexity: 178.80
Epoch [1/5], Step[900/1549], Loss: 5.0756, Perplexity: 160.07
Epoch [1/5], Step[1000/1549], Loss: 5.1047, Perplexity: 164.79
Epoch [1/5], Step[1100/1549], Loss: 5.3735, Perplexity: 215.62
Epoch [1/5], Step[1200/1549], Loss: 5.1934, Perplexity: 180.07
Epoch [1/5], Step[1300/1549], Loss: 5.0448, Perplexity: 155.22
Epoch [1/5], Step[1400/1549], Loss: 4.8532, Perplexity: 128.15
Epoch [1/5], Step[1500/1549], Loss: 5.1126, Perplexity: 166.11
Epoch [2/5], Step[0/1549], Loss: 5.4353, Perplexity: 229.37
Epoch [2/5], Step[100/1549], Loss: 4.5311, Perplexity: 92.86
Epoch [2/5], Step[200/1549], Loss: 4.6928, Perplexity: 109.16
Epoch [2/5], Step[300/1549], Loss: 4.6528, Perplexity: 104.87
Epoch [2/5], Step[400/1549], Loss: 4.5334, Perplexity: 93.07
Epoch [2/5], Step[500/1549], Loss: 4.1462, Perplexity: 63.19
Epoch [2/5], Step[600/1549], Loss: 4.4488, Perplexity: 85.53
Epoch [2/5], Step[700/1549], Loss: 4.3758, Perplexity: 79.51
Epoch [2/5], Step[800/1549], Loss: 4.4419, Perplexity: 84.94
Epoch [2/5], Step[900/1549], Loss: 4.1842, Perplexity: 65.64
Epoch [2/5], Step[1000/1549], Loss: 4.2938, Perplexity: 73.24
Epoch [2/5], Step[1100/1549], Loss: 4.5950, Perplexity: 98.99
Epoch [2/5], Step[1200/1549], Loss: 4.4834, Perplexity: 88.54
Epoch [2/5], Step[1300/1549], Loss: 4.2096, Perplexity: 67.33
Epoch [2/5], Step[1400/1549], Loss: 3.9488, Perplexity: 51.88
Epoch [2/5], Step[1500/1549], Loss: 4.2778, Perplexity: 72.08
Epoch [3/5], Step[0/1549], Loss: 6.4186, Perplexity: 613.12
Epoch [3/5], Step[100/1549], Loss: 3.8480, Perplexity: 46.90
Epoch [3/5], Step[200/1549], Loss: 3.9550, Perplexity: 52.20
Epoch [3/5], Step[300/1549], Loss: 3.9774, Perplexity: 53.38
Epoch [3/5], Step[400/1549], Loss: 3.8468, Perplexity: 46.84
Epoch [3/5], Step[500/1549], Loss: 3.3904, Perplexity: 29.68
Epoch [3/5], Step[600/1549], Loss: 3.8905, Perplexity: 48.93
Epoch [3/5], Step[700/1549], Loss: 3.6955, Perplexity: 40.26
Epoch [3/5], Step[800/1549], Loss: 3.7929, Perplexity: 44.39
Epoch [3/5], Step[900/1549], Loss: 3.4134, Perplexity: 30.37
Epoch [3/5], Step[1000/1549], Loss: 3.6280, Perplexity: 37.64
```

```
Epoch [3/5], Step[1100/1549], Loss: 3.8570, Perplexity: 47.32
Epoch [3/5], Step[1200/1549], Loss: 3.7760, Perplexity: 43.64
Epoch [3/5], Step[1300/1549], Loss: 3.4525, Perplexity: 31.58
Epoch [3/5], Step[1400/1549], Loss: 3.1765, Perplexity: 23.96
Epoch [3/5], Step[1500/1549], Loss: 3.5267, Perplexity: 34.01
Epoch [4/5], Step[0/1549], Loss: 4.5772, Perplexity: 97.24
Epoch [4/5], Step[100/1549], Loss: 3.2972, Perplexity: 27.04
Epoch [4/5], Step[200/1549], Loss: 3.4941, Perplexity: 32.92
Epoch [4/5], Step[300/1549], Loss: 3.4213, Perplexity: 30.61
Epoch [4/5], Step[400/1549], Loss: 3.3724, Perplexity: 29.15
Epoch [4/5], Step[500/1549], Loss: 2.8634, Perplexity: 17.52
Epoch [4/5], Step[600/1549], Loss: 3.4233, Perplexity: 30.67
Epoch [4/5], Step[700/1549], Loss: 3.1417, Perplexity: 23.14
Epoch [4/5], Step[800/1549], Loss: 3.3011, Perplexity: 27.14
Epoch [4/5], Step[900/1549], Loss: 2.9525, Perplexity: 19.15
Epoch [4/5], Step[1000/1549], Loss: 3.1870, Perplexity: 24.22
Epoch [4/5], Step[1100/1549], Loss: 3.3395, Perplexity: 28.20
Epoch [4/5], Step[1200/1549], Loss: 3.3653, Perplexity: 28.94
Epoch [4/5], Step[1300/1549], Loss: 2.9624, Perplexity: 19.34
Epoch [4/5], Step[1400/1549], Loss: 2.7995, Perplexity: 16.44
Epoch [4/5], Step[1500/1549], Loss: 3.0662, Perplexity: 21.46
Epoch [5/5], Step[0/1549], Loss: 3.6857, Perplexity: 39.87
Epoch [5/5], Step[100/1549], Loss: 3.0083, Perplexity: 20.25
Epoch [5/5], Step[200/1549], Loss: 3.1410, Perplexity: 23.13
Epoch [5/5], Step[300/1549], Loss: 3.0676, Perplexity: 21.49
Epoch [5/5], Step[400/1549], Loss: 3.0267, Perplexity: 20.63
Epoch [5/5], Step[500/1549], Loss: 2.5589, Perplexity: 12.92
Epoch [5/5], Step[600/1549], Loss: 3.1300, Perplexity: 22.87
Epoch [5/5], Step[700/1549], Loss: 2.8441, Perplexity: 17.19
Epoch [5/5], Step[800/1549], Loss: 2.9708, Perplexity: 19.51
Epoch [5/5], Step[900/1549], Loss: 2.6096, Perplexity: 13.59
Epoch [5/5], Step[1000/1549], Loss: 2.8992, Perplexity: 18.16
Epoch [5/5], Step[1100/1549], Loss: 3.0622, Perplexity: 21.37
Epoch [5/5], Step[1200/1549], Loss: 3.0491, Perplexity: 21.10
Epoch [5/5], Step[1300/1549], Loss: 2.6631, Perplexity: 14.34
Epoch [5/5], Step[1400/1549], Loss: 2.4941, Perplexity: 12.11
Epoch [5/5], Step[1500/1549], Loss: 2.7091, Perplexity: 15.02
```

🤔 Questions

1️⃣ Q2.1 Detaching or not? (10 points)

The above code implements a version of truncated backpropagation through time. The implementation only requires the `detach()` function (lines 7-9 of the cell) defined above the loop and used once inside the training loop.

- Explain the implementation (compared to not using truncated backprop through time).
- What does the `detach()` call here achieve? Draw a computational graph. You may choose to

answer this question outside the notebook.

- When using using line 7-9 we will typically observe less GPU memory being used during training, explain why in your answer.

🔮 Model Prediction

Below we will use our model to generate text sequence!

```
# 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
```

```
outright <unk> <unk> <unk> so a western spokesman in new york
rogers got N aliens and instructions have suffered their office
and it has big early terms with momentum
we know that he will his next two games needs and credible while our political
perspective
the
```

2️⃣ Q2.2 Sampling strategy (7 points)

Consider the sampling procedure above. The current code samples a word:

```
word_id = torch.multinomial(prob, num_samples=1).item()
```

in order to feed the model at each output step and feeding those to the next timestep. Copy below the above cell and modify this sampling startegy to use a greedy sampling which selects the highest probability word at each time step to feed as the next input.

```
# Sample greedily from the model
# 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.argmax(prob, dim=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
```

```
counter that the leadership change reflects the high of all this
the <unk> of the <unk> mr. jones has turned <unk> to the <unk>
mr. roman also raised pinkerton 's equity investment in the u.s.
he also told reporters that he would n't elaborate
he would
```

3️⃣ Q2.3 Embedding Distance (8 points)

Our model has learned a specific set of word embeddings.

- Write a function that takes in 2 words and prints the cosine distance between their embeddings using the word embeddings from the above models.
- Use it to print the cosine distance of the word "army" and the word "taxpayer".

Refer to the sampling code for how to output the words corresponding to each index. To get the index you can use the function `corpus.dictionary.word2idx`.

```python
# Embedding distance
def cosine_distance(model, word1, word2):
    idx1 = corpus.dictionary.word2idx[word1]
    idx2 = corpus.dictionary.word2idx[word2]

    # Retrieve the embeddings
    embed1 = model.embed(torch.tensor([idx1]).to(device)).squeeze()
    embed2 = model.embed(torch.tensor([idx2]).to(device)).squeeze()

    # Normalize the embeddings
    norm1 = embed1 / torch.norm(embed1)
    norm2 = embed2 / torch.norm(embed2)

    # Compute the cosine distance (dot product of normalized embeddings)
    distance = torch.dot(norm1, norm2).item()

    return 1 - distance

word1 = "army"
word2 = "taxpayer"
distance = cosine_distance(model, word1, word2)
print(f"The cosine distance between '{word1}' and '{word2}' is {distance:.4f}")
```

```
The cosine distance between 'army' and 'taxpayer' is 0.9354
```

4️⃣ Q2.4 Teacher Forcing (Extra Credit 2 points)

What is teacher forcing?

> Teacher forcing works by using the actual or expected output from the training dataset at the current time step $y(t)$ as input in the next time step $X(t+1)$, rather than the output generated by the network.

In the 🏓 `Training` code this is achieved, implicitly, when we pass the entire input sequence (`inputs = ids[:, i:i+seq_length].to(device)`) to the model at once.

Copy below the 🏓 `Training` code and modify it to disable teacher forcing training. Compare the performance of this model, to original model, what can you conclude? (compare perplexity and convergence rate)

```
# Training code 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 the first word
        targets = ids[:, (i+1):(i+1)+seq_length].to(device)

        # Initialize loss
        loss = 0

        for j in range(seq_length):
            # Forward pass
            states = detach(states)
            outputs, states = model(inputs, states)

            # Calculate loss
            loss += criterion(outputs, targets[:, j])

            # Update input for the next step
            _, predicted = outputs.max(1)
            inputs = predicted.unsqueeze(1).detach()

        # Normalize loss by sequence length
        loss /= seq_length

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

        step = (i+1) // seq_length
        if step % 100 == 0:
```

```
            print ('Epoch [{}/{}], Step[{}/{}], Loss: {:.4f}, Perplexity:
{:5.2f}'
                   .format(epoch+1, num_epochs, step, num_batches, loss.item(),
np.exp(loss.item())))
```

```
Epoch [1/5], Step[0/1549], Loss: 6.6925, Perplexity: 806.36
Epoch [1/5], Step[100/1549], Loss: 6.5009, Perplexity: 665.77
Epoch [1/5], Step[200/1549], Loss: 6.5244, Perplexity: 681.54
Epoch [1/5], Step[300/1549], Loss: 6.6948, Perplexity: 808.16
Epoch [1/5], Step[400/1549], Loss: 6.5107, Perplexity: 672.32
Epoch [1/5], Step[500/1549], Loss: 6.3665, Perplexity: 582.00
Epoch [1/5], Step[600/1549], Loss: 6.3464, Perplexity: 570.43
Epoch [1/5], Step[700/1549], Loss: 6.5332, Perplexity: 687.59
Epoch [1/5], Step[800/1549], Loss: 6.3586, Perplexity: 577.42
Epoch [1/5], Step[900/1549], Loss: 6.4413, Perplexity: 627.23
Epoch [1/5], Step[1000/1549], Loss: 6.4862, Perplexity: 656.02
Epoch [1/5], Step[1100/1549], Loss: 6.5744, Perplexity: 716.54
Epoch [1/5], Step[1200/1549], Loss: 6.4850, Perplexity: 655.22
Epoch [1/5], Step[1300/1549], Loss: 6.6614, Perplexity: 781.61
Epoch [1/5], Step[1400/1549], Loss: 6.4868, Perplexity: 656.39
Epoch [1/5], Step[1500/1549], Loss: 6.4544, Perplexity: 635.51
Epoch [2/5], Step[0/1549], Loss: 6.4039, Perplexity: 604.21
Epoch [2/5], Step[100/1549], Loss: 6.1875, Perplexity: 486.63
Epoch [2/5], Step[200/1549], Loss: 6.3588, Perplexity: 577.56
Epoch [2/5], Step[300/1549], Loss: 6.5496, Perplexity: 698.93
Epoch [2/5], Step[400/1549], Loss: 6.4144, Perplexity: 610.59
Epoch [2/5], Step[500/1549], Loss: 6.2568, Perplexity: 521.53
Epoch [2/5], Step[600/1549], Loss: 6.2490, Perplexity: 517.51
Epoch [2/5], Step[700/1549], Loss: 6.4943, Perplexity: 661.36
Epoch [2/5], Step[800/1549], Loss: 6.2322, Perplexity: 508.90
Epoch [2/5], Step[900/1549], Loss: 6.4288, Perplexity: 619.42
Epoch [2/5], Step[1000/1549], Loss: 6.4199, Perplexity: 613.95
Epoch [2/5], Step[1100/1549], Loss: 6.4798, Perplexity: 651.84
Epoch [2/5], Step[1200/1549], Loss: 6.3701, Perplexity: 584.11
Epoch [2/5], Step[1300/1549], Loss: 6.5265, Perplexity: 683.02
Epoch [2/5], Step[1400/1549], Loss: 6.3400, Perplexity: 566.80
Epoch [2/5], Step[1500/1549], Loss: 6.2940, Perplexity: 541.32
Epoch [3/5], Step[0/1549], Loss: 6.3361, Perplexity: 564.60
Epoch [3/5], Step[100/1549], Loss: 6.0814, Perplexity: 437.65
Epoch [3/5], Step[200/1549], Loss: 6.3158, Perplexity: 553.24
Epoch [3/5], Step[300/1549], Loss: 6.4307, Perplexity: 620.60
Epoch [3/5], Step[400/1549], Loss: 6.3100, Perplexity: 550.07
Epoch [3/5], Step[500/1549], Loss: 6.1807, Perplexity: 483.35
Epoch [3/5], Step[600/1549], Loss: 6.1920, Perplexity: 488.82
Epoch [3/5], Step[700/1549], Loss: 6.3868, Perplexity: 593.96
Epoch [3/5], Step[800/1549], Loss: 6.1156, Perplexity: 452.86
Epoch [3/5], Step[900/1549], Loss: 6.3059, Perplexity: 547.80
```

```
Epoch [3/5], Step[1000/1549], Loss: 6.3361, Perplexity: 564.56
Epoch [3/5], Step[1100/1549], Loss: 6.3867, Perplexity: 593.90
Epoch [3/5], Step[1200/1549], Loss: 6.2883, Perplexity: 538.25
Epoch [3/5], Step[1300/1549], Loss: 6.4212, Perplexity: 614.77
Epoch [3/5], Step[1400/1549], Loss: 6.3092, Perplexity: 549.61
Epoch [3/5], Step[1500/1549], Loss: 6.1814, Perplexity: 483.65
Epoch [4/5], Step[0/1549], Loss: 6.2447, Perplexity: 515.28
Epoch [4/5], Step[100/1549], Loss: 6.0225, Perplexity: 412.61
Epoch [4/5], Step[200/1549], Loss: 6.2967, Perplexity: 542.80
Epoch [4/5], Step[300/1549], Loss: 6.3786, Perplexity: 589.09
Epoch [4/5], Step[400/1549], Loss: 6.2813, Perplexity: 534.49
Epoch [4/5], Step[500/1549], Loss: 6.1486, Perplexity: 468.06
Epoch [4/5], Step[600/1549], Loss: 6.1711, Perplexity: 478.69
Epoch [4/5], Step[700/1549], Loss: 6.3145, Perplexity: 552.55
Epoch [4/5], Step[800/1549], Loss: 6.0381, Perplexity: 419.08
Epoch [4/5], Step[900/1549], Loss: 6.2771, Perplexity: 532.27
Epoch [4/5], Step[1000/1549], Loss: 6.2572, Perplexity: 521.77
Epoch [4/5], Step[1100/1549], Loss: 6.2865, Perplexity: 537.25
Epoch [4/5], Step[1200/1549], Loss: 6.2059, Perplexity: 495.64
Epoch [4/5], Step[1300/1549], Loss: 6.3128, Perplexity: 551.57
Epoch [4/5], Step[1400/1549], Loss: 6.2084, Perplexity: 496.92
Epoch [4/5], Step[1500/1549], Loss: 6.1116, Perplexity: 451.08
Epoch [5/5], Step[0/1549], Loss: 6.2378, Perplexity: 511.73
Epoch [5/5], Step[100/1549], Loss: 5.9888, Perplexity: 398.93
Epoch [5/5], Step[200/1549], Loss: 6.2179, Perplexity: 501.66
Epoch [5/5], Step[300/1549], Loss: 6.3315, Perplexity: 562.02
Epoch [5/5], Step[400/1549], Loss: 6.2712, Perplexity: 529.10
Epoch [5/5], Step[500/1549], Loss: 6.0086, Perplexity: 406.90
Epoch [5/5], Step[600/1549], Loss: 6.1346, Perplexity: 461.54
Epoch [5/5], Step[700/1549], Loss: 6.2731, Perplexity: 530.11
Epoch [5/5], Step[800/1549], Loss: 5.9667, Perplexity: 390.24
Epoch [5/5], Step[900/1549], Loss: 6.2845, Perplexity: 536.17
Epoch [5/5], Step[1000/1549], Loss: 6.2624, Perplexity: 524.50
Epoch [5/5], Step[1100/1549], Loss: 6.1798, Perplexity: 482.90
Epoch [5/5], Step[1200/1549], Loss: 6.2152, Perplexity: 500.32
Epoch [5/5], Step[1300/1549], Loss: 6.2706, Perplexity: 528.79
Epoch [5/5], Step[1400/1549], Loss: 6.1707, Perplexity: 478.54
Epoch [5/5], Step[1500/1549], Loss: 6.1911, Perplexity: 488.38
```

Conclusion for Q2.4

Compared with the original model, the model trained without teacher forcing presents a slower convergence rate and higher perplexity values. This is because, without teacher forcing, the model relies on its own generated outputs to learn, which might not be accurate, especially in the early stages of training. As a result, the model might take more time to learn the correct patterns and dependencies in the data. In some cases, it could lead to a more robust model that can better handle unexpected inputs during inference, but it generally takes longer to train and may not achieve the same performance as the model trained with teacher forcing.

5️⃣ Q2.5 Distance Comparison (+1 point)

Repeat the work you did for `3️⃣ Q2.3 Embedding Distance` for the model in `4️⃣ Q2.4 Teacher Forcing` and compare the distances produced by these two models (i.e. with and without the teacher forcing), what can you conclude?

```
word1 = "army"
word2 = "taxpayer"
distance_without_tf = cosine_distance(model, word1, word2)
print(f"The cosine distance between '{word1}' and '{word2}' without teacher forcing is {distance_without_tf:.4f}")
```

```
The cosine distance between 'army' and 'taxpayer' without teacher forcing is 0.9480
```

Conclusion for Q2.5

The cosine distances produced by the models with and without teacher forcing are different due to the differences in training strategies. They have different learned representations. However, different final embedding layers may not be sufficient to draw conclusions about the overall quality or performance of the models. We still need to consider evaluating both models on a validation dataset and comparing their perplexities.

```

```

Question 3

1. First, computing $\boldsymbol{A}$ for $\boldsymbol{PX}$:

$$\begin{aligned}\boldsymbol{A_{PM}} &= \frac{1}{\alpha}\boldsymbol{PXW}_q\boldsymbol{W}_k^T(\boldsymbol{PX})^T \\ &= \frac{1}{\alpha}\boldsymbol{PXW}_q\boldsymbol{W}_k^T\boldsymbol{X}^T\boldsymbol{P}^T \\ &= \boldsymbol{PAP}^T\end{aligned}$$

Then for $S(\boldsymbol{PX})$, we get :

$$S(\boldsymbol{PX}) = softmax(\boldsymbol{A_{PM}})\boldsymbol{PXW}_v = softmax(\boldsymbol{PAP}^T)\boldsymbol{PXW}_v$$

Softmax function is invariant under permutation of its input elements, specifically :

$$\begin{aligned}(\boldsymbol{P}softmax(\boldsymbol{A})\boldsymbol{P}^T)_{p(i)p(j)} &= softmax(\boldsymbol{A})_{ij} \\ &= \frac{e^{\boldsymbol{A}_{ij}}}{\sum e^{\boldsymbol{A}_{ij}}} \\ &= \frac{(e^{\boldsymbol{PAP}^T})_{p(i)p(j)}}{\sum(e^{\boldsymbol{PAP}^T})_{p(i)p(j)}} \\ &= softmax(\boldsymbol{PAP}^T)_{p(i)p(j)} \\ \Rightarrow softmax(\boldsymbol{PAP}^T) &= \boldsymbol{P}softmax(\boldsymbol{A})\boldsymbol{P}^T\end{aligned}$$

So we get :

$$S(\boldsymbol{PX}) = \boldsymbol{P}softmax(\boldsymbol{A})\boldsymbol{P}^T\boldsymbol{PXW}_v$$

For $\boldsymbol{P}$ is a permutation matrix, $\boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{I}$, we get :

$$S(\boldsymbol{PX}) = \boldsymbol{P}softmax(\boldsymbol{A})\boldsymbol{XW}_v = \boldsymbol{P}S(\boldsymbol{X})$$

2. Based on the question (a), we can get :

$$G(\boldsymbol{PX}) = \boldsymbol{w}^T S(\boldsymbol{PX}) = \boldsymbol{w}^T\boldsymbol{P}S(\boldsymbol{X})$$

$\boldsymbol{P}S(\boldsymbol{X})$ permutes the rows of $S(\boldsymbol{X})$, accordingly, if the linear operation by $\boldsymbol{w}^T$ is taking the sum along the rows, the permutation can be ignored. Thus, we can define $\boldsymbol{w} = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^T \in R^{T\times 1}$ to achieve the operation :

$$G(\boldsymbol{PX}) = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}\boldsymbol{P}S(\boldsymbol{X}) = \sum_{t=1}^{T} S(\boldsymbol{X})_{t,:} = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}S(\boldsymbol{X}) = G(\boldsymbol{X})$$

The code part starts on the next page.

```
%reset
import torch
import torch.nn as nn

# Position-wise Feedforward Network using nn.Linear
class PositionwiseFeedforwardLinear(nn.Module):
    def __init__(self, d_model):
        super(PositionwiseFeedforwardLinear, self).__init__()
        self.fc1 = nn.Linear(d_model, d_model)

    def forward(self, x):
        return torch.relu(self.fc1(x))


# Position-wise Feedforward Network using nn.Conv1d
class PositionwiseFeedforwardConv1d(nn.Module):
    def __init__(self, d_model):
        super(PositionwiseFeedforwardConv1d, self).__init__()
        self.conv1 = nn.Conv1d(d_model, d_model, kernel_size=1)

    def forward(self, x):
        x = x.permute(0, 2, 1)
        x = torch.relu(self.conv1(x))
        return x.permute(0, 2, 1)

# Create random input tensor Z with size (B, T, D)
B, T, D = 5, 10, 64
Z = torch.rand(B, T, D)

# Instantiate the Position-wise Feedforward Network layers
pwff_linear = PositionwiseFeedforwardLinear(D)
pwff_conv1d = PositionwiseFeedforwardConv1d(D)


def copy_parameters(src_model, dest_model):
    for src_param, dest_param in zip(src_model.parameters(),
dest_model.parameters()):
        dest_param.squeeze().data.copy_(src_param.squeeze().data)

copy_parameters(pwff_conv1d, pwff_linear)


# Forward pass
out_linear = pwff_linear(Z)
out_conv1d = pwff_conv1d(Z)

# Validate the implementations match
```

```
assert torch.allclose(out_linear, out_conv1d, atol=1e-5), "Implementations do
not match"
print("Match!")
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
Match!
```

```
```