HW4: Autoencoders & RNNs

The homework is broken up as follows:

How to Submit This Homework

This homework has been provided to you as an `.ipynb` file. I expect you to complete the notebook and submit the following on Canvas:

1. The completed notebook itself with your answers filled in (including any source code used to answer the question)
2. A `.pdf` file of the notebook that shows all your answers. This will make grading easier and I will appreciate that.

If you submit only one of these, you will lose points. No late homework will be graded. No submissions via email or other media will be graded.

Finally, each question specifies exactly what should be provided in your answer -- for example, if no code is requested, do not provide it. Your submitted notebook (and PDF) should not contain any additional content or cells beyond what is requested.

Setup

If you do not have access to a CUDA-compatible NVIDIA GPU, it is recommended that you run this notebook in Google Colab. There, you will have the option to enable GPU acceleration with `Runtime >> Change runtime type >> Hardware accelerator >> GPU >> Save`. Note that you may have to re-connect or restart the notebook after changing runtime types.

```
%%capture
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('hw4.ipynb')
```

```
# helper code from the course repository
%%capture
!git clone https://github.com/interactiveaudiolab/course-deep-learning.git
```

```
# install common pacakges used for deep learning
!cd course-deep-learning/ && pip install -r requirements.txt


%matplotlib inline
%cd course-deep-learning/

import datetime
import math
import torch
import matplotlib.pyplot as plt
import numpy as np
import IPython.display as ipd
from matplotlib.animation import FuncAnimation
from pathlib import Path
from torch.utils.tensorboard import SummaryWriter
from torchsummary import summary
from tqdm import tqdm
import torchvision

from utils.gan import *
```

```
/content/course-deep-learning
```

Question 1: Masked Autoencoder, Revisited (three points total)

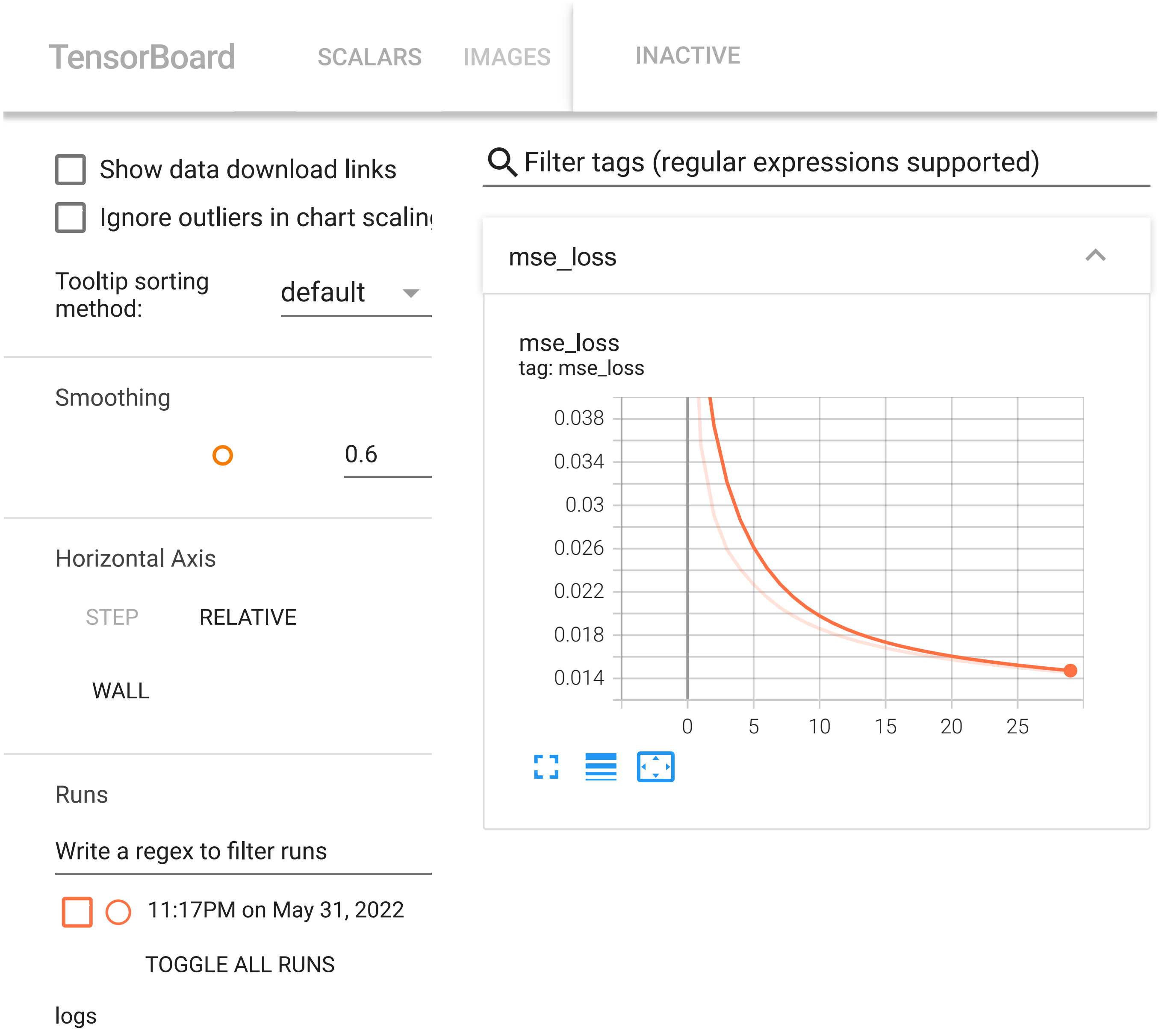
1.1 A Bigger Mask (one point)

Train the MLP-based inpainting autoencoder from notebook 7 but this time, re-write `apply_mask()` (and your training code) to **deterministically zero-out the *left half* of every input image. Train for 30 epochs, and place a plot of your reconstruction loss vs. epochs below.**

Hint: make sure that you apply the mask to a copy of each input rather than destructively editing the input itself, as the original input still has to serve as a target!

```
class MLPEncoder(torch.nn.Module):

    def __init__(self,
                 number_of_hidden_layers: int,
                 input_size: int,
                 hidden_size: int,
                 latent_size: int,
                 activation: torch.nn.Module):
        """Construct a simple MLP encoder"""

        super().__init__()

        assert number_of_hidden_layers >= 0, "Encoder number_of_hidden_layers must be
        dims_in = [input_size] + [hidden_size] * number_of_hidden_layers
        dims_out = [hidden_size] * number_of_hidden_layers + [latent_size]  # final ou
        layers = []
        for i in range(number_of_hidden_layers + 1):
            layers.append(torch.nn.Linear(dims_in[i], dims_out[i]))

            if i < number_of_hidden_layers:
                layers.append(activation)

        self.net = torch.nn.Sequential(*layers)

    def forward(self, x: torch.Tensor):
        return self.net(x)


class MLPDecoder(torch.nn.Module):

    def __init__(self,
                 number_of_hidden_layers: int,
                 latent_size: int,
                 hidden_size: int,
                 input_size: int,
                 activation: torch.nn.Module):
        """Construct a simple MLP decoder"""

        super().__init__()

        assert number_of_hidden_layers >= 0, "Decoder number_of_hidden_layers must be

        dims_in = [latent_size] + [hidden_size] * number_of_hidden_layers
        dims_out = [hidden_size] * number_of_hidden_layers + [input_size]  # final out

        layers = []
        for i in range(number_of_hidden_layers + 1):
            layers.append(torch.nn.Linear(dims_in[i], dims_out[i]))

            if i < number_of_hidden_layers:
                layers.append(activation)

        # apply Tanh after final layer to bound pixels to range [-1, 1]
        layers.append(torch.nn.Sigmoid())

        self.net = torch.nn.Sequential(*layers)

    def forward(self, x: torch.Tensor):
        return self.net(x)


class MLPAutoencoder(torch.nn.Module):
    def __init__(self,
                 number_of_hidden_layers: int,
                 input_size: int,
                 hidden_size: int,
                 latent_size: int,
                 activation_encoder: torch.nn.Module = torch.nn.ReLU(),
                 activation_decoder: torch.nn.Module = torch.nn.LeakyReLU(0.2)
                ):
        """Construct a simple MLP autoencoder

        number_of_hidden_layers: An int. Must be >=0. Defines the number of
                hidden layers for both the encoder E and decoder D
        latent_size:  An int. Defines the size of the latent representation produced b
                      the encoder.
        hidden_size: An int. The size of each hidden layer for the encoder E and
                     the decoder D.
        input_size: An int. Determines the size of the input and output images
        activation_encoder: A torch.nn.Module defining the activation function in ever
                            hidden layer of the encoder.
        activation_decoder: A torch.nn.Module defining the activation function in ever
                            hidden layer of the decoder.
        """
        super().__init__()

        self.encoder = MLPEncoder(
            number_of_hidden_layers=number_of_hidden_layers,
            input_size=input_size,
            hidden_size=hidden_size,
            latent_size=latent_size,
            activation=activation_encoder
        )

        self.decoder = MLPDecoder(
            number_of_hidden_layers=number_of_hidden_layers,
            input_size=input_size,
            hidden_size=hidden_size,
            latent_size=latent_size,
            activation=activation_decoder
        )

    def encode(self, x: torch.Tensor):
        return self.encoder(x)

    def decode(self, x: torch.Tensor):
        return self.decoder(x)

    def forward(self, x: torch.Tensor):
        return(self.decode(self.encode(x)))
```

```
%load_ext tensorboard
%tensorboard --logdir logs --port 6006
```



Show data download links
Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis
STEP [✓ RELATIVE] WALL

Runs
Write a regex to filter runs
11:17PM on May 31, 2022

logs

mse_loss

mse_loss
tag: mse_loss

```
%%capture
# autoencoder training hyperparameters
image_size = 28
batch_size = 64
```

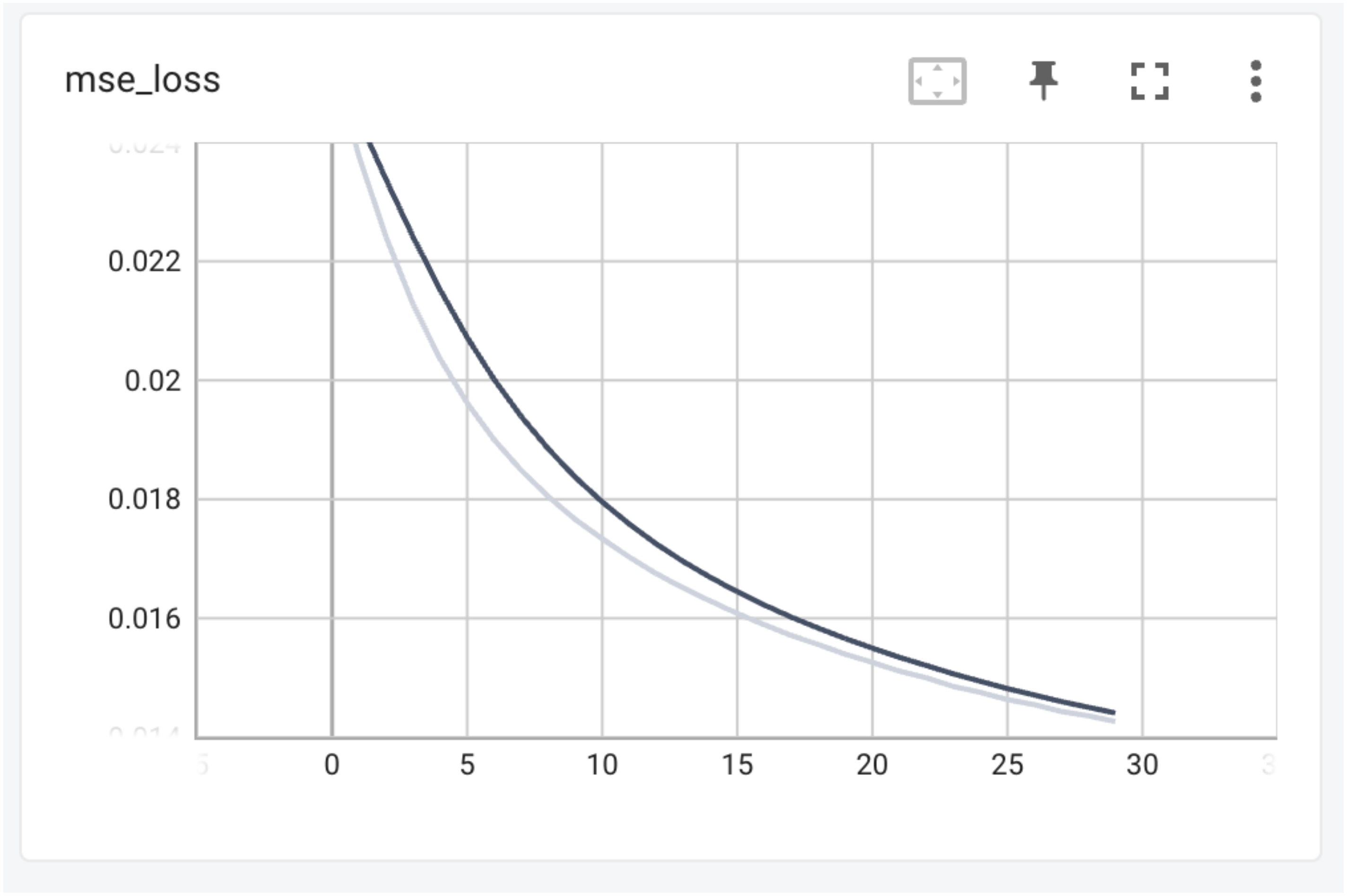
```
latent_size = 64
hidden_size = 256
number_of_hidden_layers = 2
lr = 0.0002
epochs = 30

# fix random seed
torch.manual_seed(0)

# select device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# load MNIST dataset
mnist = load_mnist(batch_size=batch_size)

# initialize the model
model = MLPAutoencoder(
    number_of_hidden_layers=number_of_hidden_layers,
    latent_size=latent_size,
    hidden_size=hidden_size,
    input_size=image_size*image_size,
).to(device)

# use an optimizer to handle parameter updates
opt = torch.optim.Adam(model.parameters(), lr=lr)

# loss function: mean-squared error between original and reconstructed images
loss = torch.nn.MSELoss()

# save all log data to a local directory
run_dir = "logs"

# to clear out TensorBoard and start totally fresh, we'll need to
# remove old logs by deleting them from the directory
!rm -rf ./logs/

# timestamp the logs for each run so we can sort through them
run_time = datetime.datetime.now().strftime("%I:%M%p on %B %d, %Y")

# initialize a SummaryWriter object to handle all logging actions
logger = SummaryWriter(log_dir=Path(run_dir) / run_time, flush_secs=20)

# log reconstructions of a fixed set of images during training
example_batch, _ = next(iter(mnist))
example_batch = example_batch.to(device)
logger.add_image("training_images", make_grid(example_batch, math.floor(math.sqrt(bat
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dat

                                    9913344/? [00:00<00:00, 51852898.86it/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./dat

                                    29696/? [00:00<00:00, 1145187.72it/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data

                                    1649664/? [00:00<00:00, 5388314.61it/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data
```

```
def apply_mask(x: torch.Tensor, image_size: int):
    """Apply mask that zero-out the left-half of each input in batch"""

    n_batch = x.shape[0]

    x_masked = x.clone()

    # apply masks to each image in batch
    for i in range(n_batch):
        for j in range(int(image_size / 2)):
            x_masked[
            i, :, :, j
            ] = 0
    return x_masked


for epoch in range(epochs):

    # weight batch losses/scores proportional to batch size
    iter_count = 0
    loss_epoch = 0

    for batch_idx, batch_data in enumerate(mnist):

        # we only care about inputs, not labels
        x_real, _ = batch_data

        # obtain batch of inputs and move to device
        x_real = x_real.to(device)
        n_batch = x_real.shape[0]

        # apply random masks
```

```
            x_masked = apply_mask(x_real, image_size)
            x_masked = x_masked.reshape(n_batch, -1)

            # flatten input images
            x_real = x_real.reshape(n_batch, -1)

            model.zero_grad()

            # train on a batch of inputs
            x_reconstructed = model(x_masked)
            loss_batch = loss(x_real, x_reconstructed)
            loss_batch.backward()
            opt.step()

            # log loss
            loss_epoch += loss_batch.detach().item() * n_batch
            iter_count += n_batch

        # plot loss
        loss_epoch /= iter_count
        logger.add_scalar("mse_loss", loss_epoch, epoch)

        # plot example generated images
        with torch.no_grad():
            masked_example = apply_mask(example_batch, image_size)
            logger.add_image("masked_images", make_grid(masked_example, math.floor(math.sq
            reconstructed_batch = model(masked_example.reshape(batch_size, -1)).reshape(ba
            logger.add_image("reconstructed_images", make_grid(reconstructed_batch, math.f

        if not epoch % 10:
            print(f"Epoch: {epoch + 1}\tMSE Loss: {loss_epoch :0.4f}")
```

```
Epoch: 1        MSE Loss: 0.0634
Epoch: 11       MSE Loss: 0.0186
Epoch: 21       MSE Loss: 0.0157
```

(YOUR LOSS PLOT HERE)



1.2 Reconstruction (one point)

The code below provides a hard-coded example image of a "3." **Pass this image through your new inpainting autoencoder and include the reconstructed image below. Does your autoencoder faithfully reconstruct the input (i.e. produce a similar-looking image of the same class)? If not, briefly explain why you think the reconstruction looks the way it does.**

[ ] ↳ *5 cells hidden*

1.3 Augmentation (one point)

In [notebook 4](), we learned how to apply random augmentations to image data. **Create a `RandomAffine` transform that applies:**

- **Rotations between -10 and 10 degrees**
- **Horizontal and vertical translations of at most 0.2**
- **Shear between -5 and 5 degrees**

Using this randomized transform, create three augmented versions of the hard-coded image above. For each augmented version obtain a reconstruction by passing it to your inpainting

autoencoder. Include the three augmented images and three corresponding reconstructions below.

Finally, answer the following question: does your autoencoder reconstruct the augmented images correctly and consistently? If not, explain why you think this may be the case.

[ ] ↳ *7 cells hidden*

Question 2: A Simple Sequence Task (three points total)

In this question, we'll explore the differences between standard recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) using a simple addition task akin to the one used in the original LSTM paper.

[ ] ↳ *14 cells hidden*

Question 3: Sequence-to-Sequence Translation (three points total)

Sequence-to-sequence tasks such as language translation require a model capable of converting variable-length inputs to variable-length outputs, with no guarantee that each position in the input will map directly to a position in the output. This can be thought of as a "many-to-many" task, as illustrated by the second figure from the right in Andrej Karpathy's diagram:

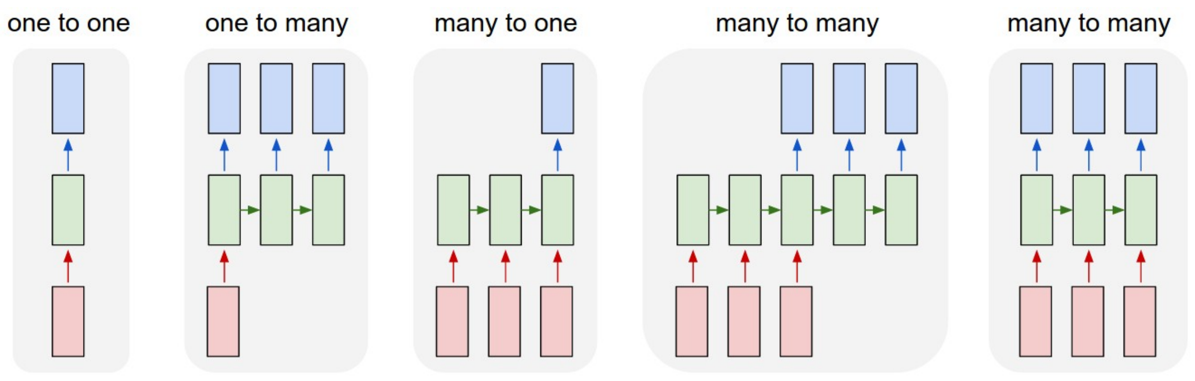


Image source: "The Unreasonable Effectiveness of Recurrent Neural Networks" (Karpathy)

For example, consider the following English-to-Spanish translation:

```
I like chocolate --> Me gusta el chocolate
```

The input sequence contains three words, while the output sequence contains four. Moreover, the individual words and grammatical structure of the sentences do not cleanly correspond, as `chocolate` is preceded by the definite article `el` and the subject switches from the speaker to the chocolate. It's easy to see why sequence-to-sequence translation can be a challenging task!

To overcome these difficulties, Sutskever et al. proposed to use recurrent **encoder** and **decoder** networks. First, special **start and stop tokens** are added to the ends of all sentences. Then, the encoder RNN loops over an input sequence, and its final hidden state is taken as a representation of the entire input "context." This context is passed to the decoder RNN, which generates one word at a time **autoregressively** (taking its own past predictions as inputs) until it produces a stop token. This allows for arbitrary input and output lengths, as (a) the only representation of the input that the decoder sees is a single context vector, and (b) the decoder is free to generate for as long as it wants to before emitting a stop token.

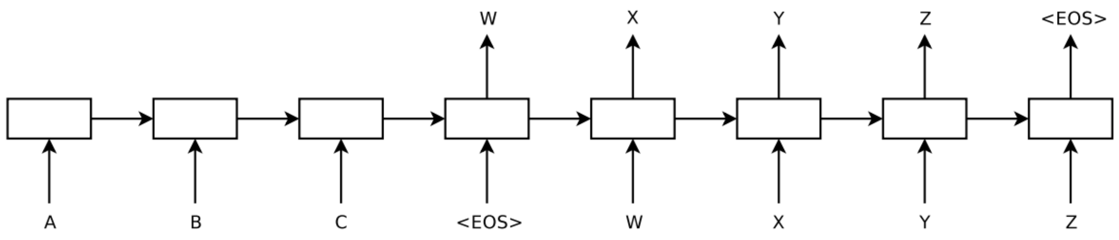


Image source: "Sequence to Sequence Learning with Neural Networks" (Sutskever et al. 2014). "EOS" is the end-of-sentence stop token.

In this question, we'll explore how to use the encoder-decoder architecture to perform German-to-English translation. This will require a bit of setup, which is documented in the next section. To simplify the process and avoid messing with your local Python environment, it's recommended that you use Google Colab.

As a final note, elements of this question are adapted from Ben Trevett's tutorials on sequence modeling in PyTorch. You are welcome to look at these tutorials for additional information.

3.1 Installation & Data Preparation (zero points)

Run these cells to prepare your environment and load data. You may be prompted to approve the installation by typing `y`, as well as to restart your notebook environment once the installation is complete.

[ ] ↳ *12 cells hidden*

3.2 RNN Translation (one point)

Finally, we can train an encoder-decoder model to perform sequence-to-sequence translation. We'll operate the encoder similar to the "layer" format we discussed in notebook 8, while the decoder will predict for one step at a time -- more akin to the "cell" format.

Modify the definitions in the cell below as instructed by the accompanying comments. Then, run the following cell to train and evaluate your encoder-decoder network for German-to-English translation. Finally, include a plot of your training and validation lossses by epoch where instructed below.

If your implementation is "correct," you should not see loss values far above `5.0` as your model trains. You can also check your implementation by running the example translation provided after the training code. Don't worry if your model isn't perfrect -- as long as a few predicted words roughly correspond to the actual translation, it indicates that your implementation is mostly correct:

Actual Translation	Predicted Translation
`a dog walked by the park`	`a a animal a walked EOS EOS`

[ ] ↳ *9 cells hidden*

3.3 LSTM Translation (one point)

Next, we'll substitute `nn.RNN` for `nn.LSTM` in our encoder and decoder networks. This means we'll also have to deal with both hidden *and* cell states, as covered in notebook 8.

Modify the definitions in the cell below as instructed by the accompanying comments. Then, run the following cell to train and evaluate your encoder-decoder network for German-to-English translation. Finally, include a plot of your training and validation lossses by epoch where instructed below.

If your implementation is "correct," you should not see loss values far above `5.0` as your model trains. You can also check your implementation by running the example translation provided after the training code. Don't worry if your model isn't perfrect -- as long as a few predicted words roughly correspond to the actual translation, it indicates that your implementation is mostly correct:

Actual Translation	Predicted Translation
`a dog walked by the park`	`a a animal a walked EOS EOS`

[ ] ↳ *7 cells hidden*

3.4 Reversing Sequences (one point)

Sutskever et al. also performed experiments in which the source sequence was reversed before being passed to the encoder network. We

Modify the data-loading code above so that all references to `tokenize_de` are replaced with `tokenize_de_reverse`. Re-generate the data and re-train your LSTM model from question 3.3. Include a plot of your training and validation lossses by epoch where instructed below. Finally, give a brief explanation of how (if at all) your network's performance changed, and why (if a change occurs).

[ ] ↳ *8 cells hidden*

Question 4: Short Response (two points total)

[ ] ↳ *14 cells hidden*

✓ 24s completed at 6:30 PM