In [ ]:
```
#Imports used throughout the notebook
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

import os
import pandas as pd
import numpy as np

from torch.utils.data import Dataset, DataLoader
from torchvision.io import read_image

import seaborn as sns
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

#Moving computations to the gpu if available
device = (
    torch.device("cuda")
    if torch.cuda.is_available()
    else torch.device("mps")
    if torch.backends.mps.is_available()
    else torch.device("cpu")
)
print(f"Using {device} device")
```

```
Using cuda device
```

Problem 1:

a: NLP

Please discuss the recent trend of rapidly increasing sizes of NLP architectures.

Answer:

NLP architectures, especially in combination with transformers, have been found to scale exceptionally well with size. To my knowledge, this trend that larger models perform better is basically ubiqutous across all topics in NLP. The reason we see these models explode due to the sheer amount of available AND accesible data, combined with the massive advances in GPUs and TPUs in recent years. Thus, it has been seen that models with more parameters can learn more complex patterns and are better at generalizing from training data.

While these improvements are generally considered to a step towards technological advancements, some people are concerned with the ethical and ecological dilemmas of such models. Firstly, training these humongous models are very costly with some articles claiming the training of GPT-4 emitted upwards of 15 metric tons of CO_2, and others

claiming estimates of 43,200kg CO2 emitted daily while running chatGPT - more than the average emissions from 30 people taking a transatlantic flight.

Secondly, some people are scared that such intelligent models might eventually turn sentient if the trends continue. How does a model determine what is right and wrong, could this be abused? What if these models fall into the hands of the wrong people? These are questions shared by many individuals in todays world, and questions even people like Sam Altman, CEO of OpenAI, have talked about in podcasts and even hearings.

I think there are a lot of opinions on this topic and its a very interesting ongoing debate. For this assignment though, I hope this was "enough" of a discussion, without deep diving into the topic completely.

b: Transfer Learning

Classify the following example of transfer learning. More exactly, what are the domains and tasks, and which are being changed?

Source: Using a step counter to monitor exercise in healthy people.

Target: Using a step counter to indicate recovery progression in a patient

Answer:

The domain D is a combination of the feature space and the marginal distribution.
In this case, the feature space

$$\chi_s = \text{steps counted in healthy people}$$

$$\chi_t = \text{steps counted in patients}$$

and the marginal distribution

$$P(\chi_s) = \mathbb{N}_0$$

$$P(\chi_t) = \mathbb{N}_0$$

While their distributions might be of the same form, its important to realise that recovering patients on avg. probably wont be having as many steps. This does not mean we can't use the information from the source domain to learn something about the target domain.

The task T is then a combination of possible labels (y), and an unknown conditional probability function.
Here, the possible labels arent explicit, but in this context, and with available information, I went with {exercising, not-exercising}. Other possible sets of labels could be:{fat, fit}, {rich, poor}, {old, young}, {male, female} etc. Its just about what information is available on the people providing step information, and I think this part of the exercise could have been worded better.

$$y_s = \{\text{exercising, not-exercising}\}$$

$$y_t = \{\text{recovered, not-recovered}\}$$

Lastly, for the conditional probability function

$$P(y|\chi_s) = \text{predicting exercise levels in people}$$

$$P(y|\chi_t) = \text{predicting recovery progression in people}$$

Thus, following https://commons.wikimedia.org/w/index.php?curid=58812416 by By Emilie Morvant - Own work, CC BY-SA 4.0, we can see that we have the same source and target marginal distributions on χ, but the tasks are different between the source and target domains.

Meaning, we land in Inductive Transfer Learning.

c: Attention

Assume sdotproduct attention, and that the hidden states of the encoder layer are [0,1,4],[-1,1,2],[1,1,1],[2,1,1]. If the activation for the previous decoder is [0.1,1,-2], what is the attention-context vector?

Answer:

sdot prodcut attention is given as

$$a_{ij} = s_{i-1}^T h_j$$

We have the hidden states (h) of the encoder given
$h_1 = [0, 1, 4]$
$h_2 = [-1, 1, 2]$
$h_3 = [1, 1, 1]$
$h_4 = [2, 1, 1]$
And the previous decoder activation s_{i-1}
$s_{i-1} = [0.1, 1, -2]$
Thus, we can calculate the sdot product $a_{ij} < br >$ a_{i1} = [0.1, 1, -2]^T \sdot [0, 1, 4] = -7$< br >$ a_{i2} = [0.1, 1, -2]^T \sdot [-1, 1, 2] = -3.1$< br >$ a_{i3} = [0.1, 1, -2]^T \sdot [1, 1, 1] = -0.9$< br >$ a_{i4} = [0.1, 1, -2]^T \sdot [2, 1, 1] = -0.8$

Using these, we can calculate teh attention weights α_{ij}.
α_{ij} is given as

$$\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_k e^{a_{ik}}}$$

since we have calculated our a_{ij}'s, we start of with calculating $\sum_k e^{a_{ik}}$
$\sum_k e^{a_{ik}} = e^{-7} + e^{-3.1} + e^{-0.9} + e^{-0.8} = 0.90185970821$
Thus, we get

$$\alpha_{i1} = \frac{e^{-7}}{e^{0.902}} = 0.0003700547$$

$$\alpha_{i2} = \frac{e^{-3.1}}{e^{0.902}} = 0.01828160879$$

$$\alpha_{i3} = \frac{e^{-0.9}}{e^{0.902}} = 0.16499176618$$

$$\alpha_{i4} = \frac{e^{-0.8}}{e^{0.902}} = 0.18234410171$$

Now we can calculate the context vector c_i given as

$$c_i = \sum_j \alpha_{ij} h_j$$

As

$c_i = 0.0003700547\sdot[0, 1, 4] + 0.01828160879\sdot[-1, 1, 2] + 0.16499176618\sdo$

A bit prettier:

$$c_i = [0.511398, 0.365988, 0.385379]$$

d: Transformers

Explain the 'positional encoding' step for transformers. Why is it done, how is it done?

Answer:

Unlike RNNs or LSTMs, which inherently process data sequentially, transformers process input data in parallel. Thus, transformers need to incorporate the information given by the sequence order. This is why positional encoding is used. An example of two sentences that would be identical without positional encoding in transformers could be "I did very well in Deep Learning" and "very well in Deep Learning I did".

One way to do it, as in the original paper, a positional encoding tensor were added to the inputs, with the same dimension, such that they can be summed:

$$\Chi \rightarrow \Chi + PE$$

Where PE is defined as

$$PE(pos, 2i) = \sin(\frac{pos}{10000\sdot\frac{2i}{d_{model}}})$$

$$PE(pos, 2i+1) = \cos(\frac{pos}{10000\sdot\frac{2i}{d_{model}}})$$

Where pos is the position and i is the dimension.
This is a "relative encoding" approach, as the values of the encoding does not scale with

input length.
The discussion on how to approach encoding is not yet over, so while this implementation showcased additive encoding, it might not be the best approach.

e: Bounding box detection:

Given a dataset with two classes; cats and dogs, and the following detections:

TP = True positive FP = False positive

cat_det = [TP, FP, TP, FP, TP]
pred_scores_cat = [0.7, 0.3, 0.5, 0.6, 0.55]

dog_det = [FP, TP, TP, FP, TP, TP]
pred_scores_dog = [0.4, 0.35, 0.95, 0.5, 0.6, 0.7]

There are in total 3 cats and 4 dogs in the images.

Calculate the mean average precision (mAP)

Answer:

mAP is defined as

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$

Where AP is the average precision for each class.
if we start off calculating Precision and Recall for each threshold:

For cats:

We have 3 cats, I've sorted the table by pred_scores_cat high to low.

Threshold	TP	FP	Precision	Recall
>0.7	1	0	1/1=1	1/3
>0.6	1	1	1/2=0.5	1/3
>0.55	2	1	2/3=0.67	2/3
>0.5	3	1	3/4=0.75	3/3
>0.3	3	2	3/5=0.6	3/3

For dogs:

Same procedure:

Threshold	TP	FP	Precision	Recall
>0.95	1	0	1/1=1	1/4=0.25
>0.7	2	0	2/2=1	2/4=0.5

Threshold	TP	FP	Precision	Recall
>0.6	3	0	3/3=1	3/4=0.75
>0.5	3	1	3/4=0.75	3/4=0.75
>0.4	3	2	3/5=0.6	3/4=0.75
>0.35	4	2	4/6=0.67	4/4=1

Now, we can calculate the Average Precision AP for both cats and dogs by taking the precision values whenever we get a new true positive and dividing with total number of cats or dogs:

$$AP_c = \frac{1.0 + \frac{2}{3} + \frac{3}{4}}{3} = 0.8056$$

$$AP_d = \frac{1.0 + 1.0 + 1.0 + \frac{4}{6}}{4} = 0.9167$$

Results have been rounded to 4 decimals.

Finally, we calculate mAP as

$$mAP = \frac{0.8056 + 0.9167}{2} = 0.8612$$

f: Semantic segmentation - FCN 1:

Given an image sized 1024x768x3 (width x height x channels), with 7 classes. What is the size of the target image if targets are one-hot encoded?

Answer:

If we onehot encode the target image, we need a channel for each class, so we end up with the target image size of 1024x768x7

g: Semantic segmentation - FCN 2:

What is a fully-convolutional network? When is it useful?

Answer:

A fully-convolutional network is a network consisting only of convolotional layers. This means there are no fully connected layers at the end, which is what we typically see in traditional convolutional networks. The fully connected layers are instead replaced by convolutional layers with a 1x1 kernel. A key feature of fully-convolutional networks, is its flexibility in terms of input image sizes as they dont rely on the fully connected layers that normally prohibits such flexibility.

FCNs are really usefull when we need a spatial understanding and our output is a map of labels corresponding to the input image. Some areas, we've had exercise sessions about, include Object detection, style transfer, and I believe there was also exercises on semantic segmentation. There are also other areas FCNs are useful in, but these were the memorable ones.

h:Residual Networks:

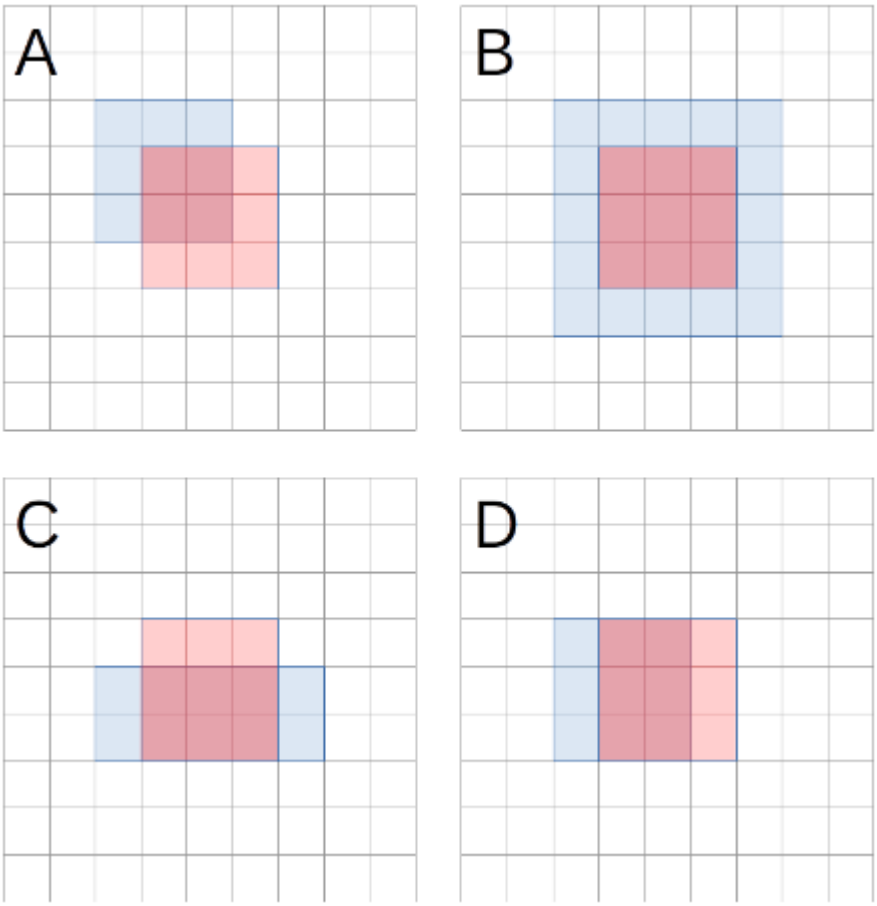
Explain residual layers and their advantage.

Answer:

Residual layers are like normal layers, but they have a "shortcut" made by 1x1 convolutions, that can be used to skip a block of layers and instead just add the original input of the block to the output of the final convolutional layer before passing through the activation function. Thus, the residual layers makes it easier and faster to train very deep networks, as the residual layers help alleviate vanishing gradients. Furthermore, this allows the network to learn if the added layers are needed or not, and if not, the network can push the weights of the block towards zero to allow the input to move almost freely through the layer.

i: Intersection-Over-Union

Calculate the intersection over union in for these four bounding-boxes and target bounding boxes:



Answer:

As the name imply, we calculate IOU as

$$IOU = \frac{\text{Area of Intersection of two boxes}}{\text{Area of Union of two boxes}}$$

$IOU_A = \frac{4}{14} = 0.29$
$IOU_B = \frac{9}{25} = 0.36$
$IOU_C = \frac{6}{13} = 0.46$
$IOU_D = \frac{6}{12} = 0.50$

j: Variational autoencoders:

What are the strengths of a variational autoencoder (VAE) compared to an autoencoder (AE)?

Answer:

VAEs regularize th encoding's distribution during training, which, in turn, ensures its latent space has good properties to allow new data to be generated. So, VAEs are basically way better at generating new data compared to traditional AEs.
Rather than representing latent attributes with discrete values, VAEs encode laten attributes as a probability distribution over the laten space for a given input.

There are, however, some drawbacks to this. Mostly at the cost of model complexity and potentially more challenging model training.

Problem 2:

Below is attached a script for generating a data set to learn translation of dates between multiple human readable and a machine readable format (ISO 8601).

Task: Using an encoder-decoder setup, perform translation from human readable to machine readable formats. Please express the performance of your trained network in terms of average accuracy of the translated output (so, accuracy on a per-character basis).

Restriction: we specifically demand that your presented solution does not include an attention layer.

Despite this restriction, the task can be solved in numerous different ways. Here are some examples of solutions of similar problems, for inspiration:

https://jscriptcoder.github.io/date-translator/Machine%20Translation%20with%20Attention%20model.html

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Script for generating data set below:

In [ ]:
```
#Copied from dateTrans_student1.py file from brightspace
"""
Created on Mon Oct 18 17:47:38 2021

@author: au207178
"""

#https://www.kaggle.com/eswarchandt/neural-machine-translation-with-attention-da

from faker import Faker
import random
from tqdm import tqdm
from babel.dates import format_date

from faker import Faker
fake = Faker()

Faker.seed(101)
random.seed(101)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



#%% pytorch dataset

class datesDataset(torch.utils.data.Dataset):
```

```
    def __init__(self,locale='da',inputLength=40,outputLength=12, dataSetSize=10

        self.inputLength=inputLength
        self.outputLength=outputLength
        self.length=dataSetSize
        self.lan=locale

        self.FORMATS= ['short', # d/M/YY
           'medium', # MMM d, YYY
           'long', # MMMM dd, YYY
           'full', # EEEE, MMM dd, YYY
           'd MMM YYY',
           'd MMMM YYY',
           'dd/MM/YYY',
           'EE d, MMM YYY',
           'EEEE d, MMMM YYY']


        #generate vocabularies:
        alphabet=sorted(tuple('abcdefghijklmnopqrstuvwxyzæøå'))
        numbers=sorted(tuple('0123456789'))
        symbols=['<SOS>','<EOS>',' ',',','.','/','-','<unk>', '<pad>'];
        self.humanVocab=dict(zip(symbols+numbers+alphabet,
                           list(range(len(symbols)+len(numbers)+len(alphabet))))
        self.machineVocab =dict(zip(symbols+numbers,list(range(len(symbols)+len(
        self.invMachineVocab= {v: k for k, v in self.machineVocab.items()}

    def string_to_int(self,string, length, vocab):
        string = string.lower()


        if not len(string)+2<=length: #+2 to make room for SOS and EOS
            print(len(string),string)
            print('Length:',length)

            raise AssertionError()


        rep = list(map(lambda x: vocab.get(x, '<unk>'),string))
        rep.insert(0,vocab['<SOS>']); rep.append(vocab['<EOS>']) #add start and

        if len(string) < length:
            rep += [vocab['<pad>']] * (length - len(rep))

        return rep

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        dt = fake.date_object()

        date = format_date(dt, format=random.choice(self.FORMATS), locale=self.l
        human_readable = date.lower().replace(',', '')
        machine_readable = dt.isoformat()

        humanEncoded=torch.LongTensor(self.string_to_int(human_readable,self.inp
        machineEncoded=torch.LongTensor(self.string_to_int(machine_readable,self
```

```
        return human_readable, machine_readable, humanEncoded,machineEncoded

e=datesDataset()
human_readable, machine_readable, humanEncoded,machineEncoded=e[0]
```

In [ ]:
```
#Doing some basic testing to get a feel for the dataset

#e[x][0] are the human readable dates, e[x][1] are the machine readable dates. [
e[0]
#e.machineVocab
#e.humanVocab
```

Out[ ]:
```
('20 nov. 1989',
 '1989-11-20',
 tensor([ 0, 11,  9,  2, 32, 33, 40,  4,  2, 10, 18, 17, 18,  1,  8,  8,  8,
8,
         8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,
8,
         8,  8,  8,  8]),
 tensor([ 0, 10, 18, 17, 18,  6, 10, 10,  6, 11,  9,  1]))
```

In [ ]:
```
#define network
#I spent a couple of hours trying with an LSTM implementation, but in the end I
#The good thing about using GRU as the architecture, is its faster convergence c
#I used the tutorial from https://pytorch.org/tutorials/intermediate/seq2seq_tra

#https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_p=0.1):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.dropout = nn.Dropout(dropout_p)

    def forward(self, input):
        embedded = self.dropout(self.embedding(input))
        output, hidden = self.gru(embedded)
        return output, hidden

#https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, encoder_outputs, encoder_hidden, target_tensor=None):
        batch_size = encoder_outputs.size(0)
        decoder_input = torch.empty(batch_size, 1, dtype=torch.long, device=devi
        decoder_hidden = encoder_hidden
        decoder_outputs = []

        for i in range(40): #Max length of input sequence is 40
            decoder_output, decoder_hidden  = self.forward_step(decoder_input, d
            decoder_outputs.append(decoder_output)
```

```
            if target_tensor is not None:
                # Teacher forcing: Feed the target as the next input
                decoder_input = target_tensor[:, i].unsqueeze(1) # Teacher forci
            else:
                # Without teacher forcing: use its own predictions as the next i
                _, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze(-1).detach()  # detach from history

        decoder_outputs = torch.cat(decoder_outputs, dim=1)
        decoder_outputs = F.log_softmax(decoder_outputs, dim=-1)
        return decoder_outputs, decoder_hidden, None # We return `None` for cons

    def forward_step(self, input, hidden):
        output = self.embedding(input)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.out(output)
        return output, hidden



class Net(nn.Module):
    def __init__(self, encoder, decoder, device):
        super(Net, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio=0.4):
        # src: source sequence, trg: target sequence
         #Teacher forcing ratio is the probability of using teacher forcing, I t
         #but 0.4 seemed fine, while also not making the model rely on it too mu
        batch_size = src.size(0)
        max_len = trg.size(1)  #12 (len of longest output sequence)
        trg_vocab_size = self.decoder.embedding.num_embeddings #19 (len of machi


        # tensor to store decoder outputs
        outputs = torch.zeros(batch_size, max_len, trg_vocab_size).to(self.devic

        # encode the source sentence
        encoder_output, hidden = self.encoder(src) #As we're not doing attention

        # first input to the decoder is the <sos> tokens
        input = torch.tensor([0]*batch_size).to(self.device)  # assuming <sos> t
        input = input.unsqueeze(1)  # add batch dimension


        for t in range(1, max_len):
            output, hidden = self.decoder.forward_step(input, hidden) #output is
            outputs[:, t] = output.squeeze(1) #Save the logits in the outputs te
            teacher_force = random.random() < teacher_forcing_ratio

            top1 = output.squeeze(1).max(1)[1] #Get the index of the highest log
            input = (trg[:, t] if teacher_force else top1).unsqueeze(1) #If teac

        return outputs
```

In [ ]:
```
#data processing
X, y, X_enc, y_enc = zip(*[e[i] for i in range(e.length)])

#Making the encoded data into tensors
X_enc = torch.stack(X_enc)
y_enc = torch.stack(y_enc)
```

In [ ]:
```
#Splitting the data into train and test sets
#Defining the inputs and targets, creating the dataset
inputs = torch.tensor(X_enc, dtype=torch.long).to(device)
targets = torch.tensor(y_enc, dtype=torch.long).to(device)
dataset = torch.utils.data.TensorDataset(inputs, targets)
indices = list(range(len(dataset)))

#Batch size
batch_size = 16 #Tested with bs 4, 16, 24, 32, 64. 16 seemed to be the best, but

#Splitting the dataset into training, validation and test sets
train_indices, temp_indices = train_test_split(indices, test_size=0.2, random_st
valid_indices, test_inddices = train_test_split(temp_indices, test_size=0.5, ran


#Create subsets of the data using the split indices
train_data = torch.utils.data.Subset(dataset, train_indices)
valid_data = torch.utils.data.Subset(dataset, valid_indices)
test_data = torch.utils.data.Subset(dataset, test_inddices)

# Create the dataloaders
train_dl = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_worke
valid_dl = DataLoader(valid_data, batch_size=batch_size, shuffle=True, num_worke
test_dl = DataLoader(test_data, batch_size=batch_size, shuffle=True, num_workers
```

```
C:\Users\Jens\AppData\Local\Temp\ipykernel_6728\3526470852.py:3: UserWarning: To
copy construct from a tensor, it is recommended to use sourceTensor.clone().detac
h() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tens
or(sourceTensor).
  inputs = torch.tensor(X_enc, dtype=torch.long).to(device)
C:\Users\Jens\AppData\Local\Temp\ipykernel_6728\3526470852.py:4: UserWarning: To
copy construct from a tensor, it is recommended to use sourceTensor.clone().detac
h() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tens
or(sourceTensor).
  targets = torch.tensor(y_enc, dtype=torch.long).to(device)
```

In [ ]:
```
encoder = Encoder(input_size=48, hidden_size=512).to(device) #tested hidden size
decoder = Decoder(hidden_size=512, output_size=19).to(device)
net = Net(encoder, decoder, device).to(device)
#Optimizer and loss function
optimizer = optim.Adam(net.parameters(), lr=0.001) #tested with lr 0.005, 0.002,
loss = nn.CrossEntropyLoss()
#Training
nEpochs = 2001
best_loss = 1000
best_acc = 0
best_valid_loss = 1000
best_valid_acc = 0

mod = 50 #Modulus for printing the best results

for iEpoch in range(nEpochs):
```

```
    net.train()
    totLoss=0
    cor_pred = 0
    tot_pred = 0
    cor_pred_val = 0
    tot_pred_val = 0
    for xbatch,ybatch in train_dl:
        xbatch=xbatch.to(device=device)
        ybatch=ybatch.to(device=device)

        y_pred = net(src=xbatch, trg=ybatch)
        y_pred = y_pred.permute(0, 2, 1) ## permute to match CrossEntropyLoss in

        #Next section is to calculate the performance of average accuracy of the
        #Convert the logits to predictions
        softmax = nn.Softmax(dim=1)
        predictions = torch.argmax(softmax(y_pred), dim=1)

        # Mask to filter out padding tokens in the targets
        mask = ybatch != 8
        # Use the mask to filter out padding in both predictions and targets
        predictions_masked = torch.masked_select(predictions, mask)
        targets_masked = torch.masked_select(ybatch, mask)

        # Correct predictions
        cor_pred += (predictions_masked == targets_masked).sum().item()

        # Total predictions in batch
        tot_pred += targets_masked.size(0)

        #Back to Calculating loss like normal
        loss_val = loss(y_pred, ybatch.long())
        totLoss+=loss_val

        # Zero the gradients before running the backward pass.
        net.zero_grad()

        loss_val.backward()

        #gradient clipping to deal with exploding gradients.
        torch.nn.utils.clip_grad_norm_(net.parameters(),10, norm_type=2.0)
        optimizer.step()
    accuracy_train = cor_pred / tot_pred
    if (accuracy_train > best_acc) & (iEpoch % mod == 0):
        best_acc = accuracy_train
        print('Train acc', iEpoch,":", best_acc)

    if (totLoss < best_loss) & (iEpoch % mod == 0):
        best_loss = totLoss
        print('Train loss',iEpoch,":", totLoss)

    net.eval()  # Set the model to evaluation mode
    valid_loss_sum = 0.0
    correct_predictions = 0
    #For this i just reused the code from training, some naming might be weird
    for inputs, labels in valid_dl:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs, labels)
        outputs = outputs.permute(0, 2, 1)
        batch_loss = loss(outputs, labels.long())
```

```
        valid_loss_sum += batch_loss.item()

        #Convert the logits to predictions
        softmax = nn.Softmax(dim=1)
        predictions = torch.argmax(softmax(y_pred), dim=1)

        # Mask for non-padding tokens in the targets
        mask = ybatch != 8  # Create a mask for non-padding tokens
        # Use the mask to filter out padding in both predictions and targets
        predictions_masked = torch.masked_select(predictions, mask)
        targets_masked = torch.masked_select(ybatch, mask)

        # Correct predictions
        cor_pred_val += (predictions_masked == targets_masked).sum().item()

        # Total predictions in batch
        tot_pred_val += targets_masked.size(0)

    # Calculate accuracy
    valid_acc = cor_pred_val / tot_pred_val

    valid_loss = valid_loss_sum / len(valid_dl)

    if (valid_loss < best_valid_loss) & (iEpoch % mod == 0):
        best_valid_loss = valid_loss
        netImage = net.state_dict()
        print('Val loss', iEpoch,":", best_valid_loss)
        bestPred = outputs  # Be cautious about overwriting bestPred every epoch
    if (valid_acc > best_valid_acc) & (iEpoch % mod == 0):
        best_valid_acc = valid_acc
        netImage = net.state_dict()
        print('Val acc', iEpoch,":", best_valid_acc)
        bestPred = outputs  # Be cautious about overwriting bestPred every epoch
```

```
Train acc 0 : 0.11875
Train loss 0 : tensor(14.7818, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 0 : 2.901568651199341
Val acc 0 : 0.14583333333333334
Train acc 50 : 0.43645833333333334
Train loss 50 : tensor(10.1085, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 50 : 1.9745641946792603
Val acc 50 : 0.4479166666666667
Train acc 100 : 0.4479166666666667
Train loss 100 : tensor(9.0248, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 100 : 1.756047010421753
Train acc 150 : 0.4875
Train loss 150 : tensor(8.5627, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 150 : 1.7199493646621704
Val acc 150 : 0.53125
Train acc 200 : 0.559375
Train loss 200 : tensor(7.9359, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 200 : 1.4749699831008911
Train acc 250 : 0.6114583333333333
Train loss 250 : tensor(7.1089, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 250 : 0.625
Train acc 350 : 0.6864583333333333
Train loss 350 : tensor(5.9997, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 350 : 1.1982444524765015
Val acc 350 : 0.671875
Train acc 400 : 0.6895833333333333
Train loss 400 : tensor(5.6454, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 400 : 0.6927083333333334
Train acc 450 : 0.7010416666666667
Train loss 450 : tensor(5.3352, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 450 : 1.093652367591858
Train acc 500 : 0.7114583333333333
Train loss 500 : tensor(5.3036, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 500 : 1.0750452280044556
Val acc 500 : 0.7395833333333334
Train acc 550 : 0.7375
Train loss 550 : tensor(5.0830, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 550 : 1.0521665811538696
Train acc 600 : 0.75625
Train loss 600 : tensor(4.8340, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 600 : 1.0252037048339844
Val acc 600 : 0.78125
Train acc 650 : 0.7739583333333333
Train loss 650 : tensor(4.6653, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 650 : 1.0037099123001099
Train acc 700 : 0.7854166666666667
Train loss 700 : tensor(4.5094, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 700 : 0.9986371397972107
Val acc 700 : 0.8177083333333334
Train acc 750 : 0.7885416666666667
Train loss 750 : tensor(4.3457, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 750 : 0.9896564483642578
Train acc 800 : 0.8114583333333333
Train loss 800 : tensor(4.1921, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 800 : 0.9822566509246826
Train acc 850 : 0.8260416666666667
Train loss 850 : tensor(4.0083, device='cuda:0', grad_fn=<AddBackward0>)
Val loss 850 : 0.9776142835617065
Train acc 900 : 0.8385416666666666
Train loss 900 : tensor(3.8529, device='cuda:0', grad_fn=<AddBackward0>)
```

```
Val acc 900 : 0.8541666666666666
Train acc 950 : 0.8552083333333333
Train loss 950 : tensor(3.6764, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 950 : 0.8645833333333334
Train acc 1000 : 0.8666666666666667
Train loss 1000 : tensor(3.4964, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1050 : 0.8864583333333333
Train loss 1050 : tensor(3.3152, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1050 : 0.8854166666666666
Train acc 1100 : 0.8947916666666667
Train loss 1100 : tensor(3.1408, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1100 : 0.8958333333333334
Train acc 1150 : 0.9072916666666667
Train loss 1150 : tensor(2.9969, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1150 : 0.9270833333333334
Train acc 1200 : 0.91875
Train loss 1200 : tensor(2.8574, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1250 : 0.9302083333333333
Train loss 1250 : tensor(2.6904, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1250 : 0.9375
Train acc 1300 : 0.9375
Train loss 1300 : tensor(2.5597, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1350 : 0.9416666666666667
Train loss 1350 : tensor(2.4314, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1350 : 0.953125
Train acc 1400 : 0.9552083333333333
Train loss 1400 : tensor(2.3210, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1450 : 0.9604166666666667
Train loss 1450 : tensor(2.1846, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1500 : 0.9729166666666667
Train loss 1500 : tensor(2.0555, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1500 : 0.9583333333333334
Train acc 1550 : 0.975
Train loss 1550 : tensor(1.9942, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1550 : 0.984375
Train acc 1600 : 0.978125
Train loss 1600 : tensor(1.9046, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1650 : 0.990625
Train loss 1650 : tensor(1.7974, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1700 : 0.9927083333333333
Train loss 1700 : tensor(1.7097, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1700 : 0.9895833333333334
Train acc 1750 : 0.99375
Train loss 1750 : tensor(1.6500, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1800 : 0.9958333333333333
Train loss 1800 : tensor(1.5887, device='cuda:0', grad_fn=<AddBackward0>)
Val acc 1800 : 1.0
Train loss 1850 : tensor(1.5418, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1900 : 0.9979166666666667
Train loss 1900 : tensor(1.4939, device='cuda:0', grad_fn=<AddBackward0>)
Train acc 1950 : 1.0
Train loss 1950 : tensor(1.4556, device='cuda:0', grad_fn=<AddBackward0>)
Train loss 2000 : tensor(1.4223, device='cuda:0', grad_fn=<AddBackward0>)
```

In [ ]:
```
#Some testing
best_test_loss = 1000
net.eval()  # Set the model to evaluation mode
test_loss_sum = 0.0
best_test_acc = 0
cor_pred_test = 0
```

```
tot_pred_test = 0
for inputs, labels in test_dl:
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = net(inputs, labels)
    outputs = outputs.permute(0, 2, 1)
    batch_loss = loss(outputs, labels.long())
    test_loss_sum += batch_loss.item()

    #Convert the logits to predictions
    softmax = nn.Softmax(dim=1)
    predictions = torch.argmax(softmax(outputs), dim=1)

    # Mask for non-padding tokens in the targets
    mask = labels != 8  # Create a mask for non-padding tokens
    # Use the mask to filter out padding in both predictions and targets
    predictions_masked = torch.masked_select(predictions, mask)
    targets_masked = torch.masked_select(labels, mask)

    # Correct predictions
    cor_pred_test += (predictions_masked == targets_masked).sum().item()

    # Total predictions in batch
    tot_pred_test += targets_masked.size(0)

    # Calculate accuracy
test_acc = cor_pred_test / tot_pred_test

test_loss = test_loss_sum / len(test_dl)

if test_loss < best_test_loss:
    best_loss = test_loss
    netImage = net.state_dict()
    print('updated best loss')
    bestPred = outputs  # Be cautious about overwriting bestPred every epoch

print('Test loss:', test_loss)
print('Test acc:', test_acc)
```

```
updated best loss
Test loss: 1.410487413406372
Test acc: 0.775
```

Training and validation

So, we converge towards 1 in validation accuracy pretty fast, especially if we tune the hyperparameters a bit towards more hidden layers. The validation loss seems to converge around 1.1 and the training loss around 1.2x. With validation and training accuracy reaching 1.0 before 150 epochs.

Test results

So a test accuracy of 80% was around the best i achieved (the highest i saw was 80,3). This was tested with different hyperparameters. It can be seen in the comments which values i tested with, but i tested on hidden_size, lr in the optimizer, and different batch sizes. If I'd been able to implement the logic using LSTM, I suspect i would've gotten

better accuracy, however, time was running from me. Furthermore, with the use of attention we also would expect significant improvement to the accuracy. Maybe it could also have been interesting to look at the Date-level Accuracy, instead of only the Character accuracy. Overall, I think we would like higher accuracy, as the task, on an intuition level, does not seem that hard. I am not entirely sure why I was only able to achieve 80% accuracy. Maybe I'm overfitting, however, when I was running fewer epochs, the results were worse overall. Finally, I tried running 2000 epochs a few times, without any improvements. Also running with learning rates of down to 0.000001 without improvements.

In [ ]:

In [ ]: