A Practical Guide to Building a Seq2Seq Transformer from scratch with PyTorch

## Introduction

This jupyter notebook details how to build a **sequence-to-sequence** (Seq2Seq) model using **PyTorch**, focusing on the **Transformer** architecture. Our goal is to develop a transpiler that translates and transpiles PL/I code into Kotlin. Throughout the notebook, we’ll cover everything from setting up the necessary libraries to training the model and evaluating its performance

## Libraries and setup

We start by installing the required libraries:

!pip install torch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2  
!pip install torchtext==0.8.1  
!pip install antlr4-python3-runtime==4.9.2

We then import all the necessary libraries. **json** will be used to import our data, **antlr4** provides the backbone for our transpiler, enabling us to parse PL/I code and manipulate its structure with ease, **torch** PyTorch empowers us to build and train neural networks seamlessly, **torchtext** complements PyTorch by providing utilities for text processing and dataset handling, **jinja2** simplifies code generation with its template engine ensuring a smooth transition between languages and to finish **matplotlib** will come in handy during training for visualising our progress

import json  
  
from antlr4 import \*  
from pli.PLILexer import PLILexer  
from pli.PLIParser import PLIParser  
from pli.PLIVisitor import PLIVisitor  
  
import torch  
import torch.nn as nn  
import torch.optim as optim  
from torchtext.data import Field, TabularDataset, BucketIterator  
  
from jinja2 import Template  
  
import matplotlib.pyplot as plt

## The Transformer Model

The Transformer class encapsulates a custom implementation of the Transformer model, a powerful architecture for **sequence-to-sequence** tasks. It comprises various components, including **embedding layers** for source and target sequences, **positional embeddings** to capture sequence order, and a multi-layer Transformer module. The model utilizes **dropout** for regularization and employs **masks** to handle padding and prevent information leakage during training. With these components, the Transformer class can efficiently process source and target sequences, facilitating tasks like language translation or code generation

class Transformer(nn.Module):  
  
 # Initialises the Transformer model  
 def \_\_init\_\_(  
 self,  
 embedding\_size,  
 src\_vocab\_size,  
 trg\_vocab\_size,  
 src\_pad\_idx,  
 num\_heads,  
 num\_encoder\_layers,  
 num\_decoder\_layers,  
 forward\_expansion,  
 dropout,  
 max\_len,  
 device,  
 ):  
 super(Transformer, self).\_\_init\_\_()  
 self.src\_word\_embedding = nn.Embedding(src\_vocab\_size, embedding\_size)  
 self.src\_position\_embedding = nn.Embedding(max\_len, embedding\_size)  
 self.trg\_word\_embedding = nn.Embedding(trg\_vocab\_size, embedding\_size)  
 self.trg\_position\_embedding = nn.Embedding(max\_len, embedding\_size)  
  
 self.device = device  
 self.transformer = nn.Transformer(  
 embedding\_size,  
 num\_heads,  
 num\_encoder\_layers,  
 num\_decoder\_layers,  
 forward\_expansion,  
 dropout,  
 )  
 self.fc\_out = nn.Linear(embedding\_size, trg\_vocab\_size)  
 self.dropout = nn.Dropout(dropout)  
 self.src\_pad\_idx = src\_pad\_idx  
  
 # Generates a mask for the source sequence to handle padding  
 def make\_src\_mask(self, src):  
 src\_mask = src.transpose(0, 1) == self.src\_pad\_idx  
  
 # (N, src\_len)  
 return src\_mask.to(self.device)  
  
 # Forward pass of the Transformer model  
 def forward(self, src, trg):  
 src\_seq\_length, N = src.shape  
 trg\_seq\_length, N = trg.shape  
  
 src\_positions = (  
 torch.arange(0, src\_seq\_length)  
 .unsqueeze(1)  
 .expand(src\_seq\_length, N)  
 .to(self.device)  
 )  
  
 trg\_positions = (  
 torch.arange(0, trg\_seq\_length)  
 .unsqueeze(1)  
 .expand(trg\_seq\_length, N)  
 .to(self.device)  
 )  
  
 embed\_src = self.dropout(  
 (self.src\_word\_embedding(src) + self.src\_position\_embedding(src\_positions))  
 )  
 embed\_trg = self.dropout(  
 (self.trg\_word\_embedding(trg) + self.trg\_position\_embedding(trg\_positions))  
 )  
  
 src\_padding\_mask = self.make\_src\_mask(src)  
 trg\_mask = self.transformer.generate\_square\_subsequent\_mask(trg\_seq\_length).to(  
 self.device  
 )  
  
 out = self.transformer(  
 embed\_src,  
 embed\_trg,  
 src\_key\_padding\_mask=src\_padding\_mask,  
 tgt\_mask=trg\_mask,  
 )  
 out = self.fc\_out(out)  
 return out

## Data preparation and tokenization

For our model to understand and generate text, we need to tokenize our input data. We also set up our datasets to allow us to manage training and testing data efficiently

### Tokenize

We define a simple **tokenizer** that splits a string into tokens based on whitespace, it then creates two **Field** objects with some specifications and finally we create a dictionary that maps field names to tuples to be used later to specify how to load and process the data

import torchtext.data as data  
  
tokenizer = lambda x: x.split()  
  
pli = data.Field(sequential=True, use\_vocab=True, tokenize=tokenizer, lower=True, init\_token="<sos>", eos\_token="<eos>")  
ktl = data.Field(sequential=True, use\_vocab=True, tokenize=tokenizer, lower=True, init\_token="<sos>", eos\_token="<eos>")  
  
fields = {'pli': ('p', pli), 'ktl': ('k', ktl)}

### Splits

We’re using the TabularDataset module from the torchtext.data package to create separate datasets for **training** and **testing**. The data is loaded from JSON files located in the ‘data’ directory. We specify the format of the data as JSON and define the fields to be extracted from the JSON files using the fields parameter

train, test = data.TabularDataset.splits(  
 path='data',  
 train='train.json',  
 test='test.json',  
 format='json',  
 fields=fields,  
)

### Vocab

Building a vocabulary involves creating a dictionary that maps each unique word in the dataset to a unique index. This process is crucial for natural language processing tasks as it allows machine learning models to represent words as numerical values, which they can process and understand. In this case, we’re building vocabularies for the PL/I (pli) and Kotlin (ktl) datasets, ensuring that the model has a predefined set of words it can understand and process during training and inference

pli.build\_vocab(train, max\_size=10000, min\_freq=1)  
ktl.build\_vocab(train, max\_size=10000, min\_freq=1)

## Translate and transpile

### Translate

Here we define a **translate\_sequence** function. It tokenizes the input sentence, adds < sos > and < eos > tokens at the beginning and end respectively, converts the tokens to indices using the vocabulary of the pli field, converts the indices to a PyTorch tensor, and iteratively predicts the next token in the translated sequence using the trained model until either the < eos > token is predicted or the maximum length is reached. Finally, it converts the predicted indices back to tokens using the vocabulary of the Kotlin field and returns the translated sentence, removing the start token < sos > as it is only used for signaling the model to start generation and is not part of the actual translated text

def translate\_sequence(sentence, pli, ktl, device, max\_length=50):  
 if type(sentence) == str:  
 tokens = [token.lower() for token in sentence.split()]  
 else:  
 tokens = [token.lower() for token in sentence]  
 # Add <SOS> and <EOS> in beginning and end respectively  
  
 tokens.insert(0, pli.init\_token)  
 tokens.append(pli.eos\_token)  
  
 # Iterate each languae token and convert to an index  
 text\_to\_indices = [pli.vocab.stoi[token] for token in tokens]  
  
 # Convert to Tensor  
 sentence\_tensor = torch.LongTensor(text\_to\_indices).unsqueeze(1).to(device)  
  
 outputs = [ktl.vocab.stoi["<sos>"]]  
 for i in range(max\_length):  
 trg\_tensor = torch.LongTensor(outputs).unsqueeze(1).to(device)  
  
 with torch.no\_grad():  
 output = model(sentence\_tensor, trg\_tensor)  
  
 best\_guess = output.argmax(2)[-1, :].item()  
 outputs.append(best\_guess)  
  
 if best\_guess == ktl.vocab.stoi["<eos>"]:  
 break  
  
 translated\_sentence = [ktl.vocab.itos[idx] for idx in outputs]  
 # remove start token  
 return translated\_sentence[1:]

### Transpile

We also define a **transpile\_sequence** function. It retrieves the code tokens and context data from the input dictionary. Then, it initializes an empty list to store the transpiled code with proper indentation. The function then iterates through the tokens, adjusting the indentation level based on curly braces {} encountered in the code. It ignores the token. And after iterating through all the tokens, the function joins the transpiled code with proper spacing and renders it using the contextual data. Finally, it returns the transpiled code and the updated indentation level

def transpile\_sequence(translated, level):  
 tokens = translated["code"]  
 data = translated["context"]  
 lint = []  
  
 for t in tokens:  
 spacer = "".rjust(level \* 4)  
 if t == "{":  
 level += 1  
 elif t == "}" and level > 0:  
 level -= 1  
 spacer = "".rjust(level \* 4)  
  
 if t != "<eos>":  
 lint.append(spacer + t)  
  
 code = " ".join(lint)  
 t = Template(code)  
  
 return t.render(data), level

## Specifics

Before defining the final parameters and training our model we have a few useful functions to define

This **remove\_eos** function removes all < eos > tokens and then concatenate the remaining tokens into a single string

def remove\_eos(witheos):  
 noeos = []  
 for w in witheos:  
 if w != '<eos>':  
 noeos.append(w)  
 return " ".join(noeos)

The following functions, **save\_checkpoint** and **load\_checkpoint**, are essential for saving and loading the state of our model during training or inference

def save\_checkpoint(state, filename="checkpoint.pth.tar"):  
 print("=> Saving checkpoint")  
 torch.save(state, filename)  
  
  
def load\_checkpoint(checkpoint, model, optimizer):  
 print("=> Loading checkpoint")  
 model.load\_state\_dict(checkpoint["state\_dict"])  
 optimizer.load\_state\_dict(checkpoint["optimizer"])

## Training

### Hyperparameters

We start by defining some **model hyperparameters** that help define how our model processes and transforms our input data. We also define some **training hyperparameters** that regulate the training process of our model

# ready to define everything we need for training our Seq2Seq model  
device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
  
load\_model = True  
save\_model = True  
  
# Model hyperparameters  
src\_vocab\_size = len(pli.vocab)  
trg\_vocab\_size = len(ktl.vocab)  
embedding\_size = 512  
num\_heads = 8  
num\_encoder\_layers = 6  
num\_decoder\_layers = 6  
dropout = 0.10  
max\_len = 100  
forward\_expansion = 4  
src\_pad\_idx = ktl.vocab.stoi["<pad>"]  
  
global level  
# Training hyperparameters  
num\_epochs = 300  
learning\_rate = 3e-4  
batch\_size = 32  
training\_losses = []

### Model Initialization and Training Setup

We then **initialize** the **model**, **optimizer**, **scheduler**, and **criterion** for training a Transformer model. we also create **iterators** for the training and test datasets that facillitate efficient data management, improve computational efficiency, and enhance the overall training process’s effectiveness

train\_iterator, test\_iterator = BucketIterator.splits(  
 (train, test),  
 batch\_size=batch\_size,  
 sort\_within\_batch=True,  
 sort\_key=lambda x: len(x.p),  
 device=device,  
)  
model = Transformer(  
 embedding\_size,  
 src\_vocab\_size,  
 trg\_vocab\_size,  
 src\_pad\_idx,  
 num\_heads,  
 num\_encoder\_layers,  
 num\_decoder\_layers,  
 forward\_expansion,  
 dropout,  
 max\_len,  
 device,  
).to(device)  
  
optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  
  
scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(  
 optimizer, factor=0.1, patience=10, verbose=True  
)  
  
pad\_idx = ktl.vocab.stoi["<pad>"]  
criterion = nn.CrossEntropyLoss(ignore\_index=pad\_idx)

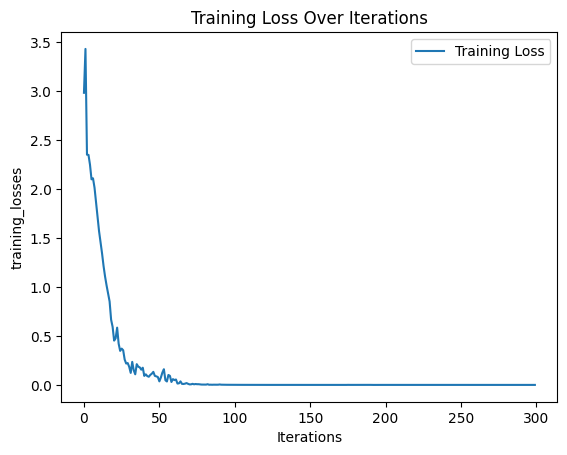
### Training loop

Finaly we define our custom training loop. Within the training loop, batches of data are fetched using the **training\_iterator**. Input and target sequences are transferred to the appropriate device (CPU or GPU), and the model is then trained using **forward** and **backward** passes. **Gradient clipping** is applied to prevent exploding gradients, and the **optimizer** is used to update the model parameters based on the computed gradients. After training, we evaluate the model’s performance on example pli sequences, translating them to Kotlin using the trained model. Finally we utilise matplotlib to visualize the training loss over iterations and optionally save the trained model checkpoint for future use

if \_\_name\_\_ == '\_\_main\_\_':  
 for epoch in range(num\_epochs):  
 model.eval()  
 # little bit of output to check the progress  
 if epoch % 50 == 0:  
 print(f"[Epoch {epoch} / {num\_epochs}]")  
 # Iterate over the dataset and extract PLI sequences  
 sentences = [  
 {'pli': ['PROCEDURE', 'MAIN', '{{type0}}', '{{type1}}'],  
 'context': {'type0': 'Array', 'type1': 'String'}},  
 {'pli': ['DO'], 'context': {}},  
 {'pli': ['END'], 'context': {}}  
 ]  
 print(f"Translated example sentence:")  
 level = 0  
 for s in sentences:  
 translated = translate\_sequence(  
 s['pli'], pli, ktl, device, max\_length=50  
 )  
 transpiled, level = transpile\_sequence({  
 'code': translated,  
 'context': s['context']  
 }, level)  
 print(f"{transpiled}")  
 last\_5\_losses = training\_losses[-5:]  
 print("Last 5 training losses:", last\_5\_losses)  
 model.train()  
 losses = []  
 for batch\_idx, batch in enumerate(train\_iterator):  
 # Get input and targets and get to cuda  
 inp\_data = batch.p.to(device)  
 target = batch.k.to(device)  
 # Forward  
 output = model(inp\_data, target[:-1, :])  
 output = output.reshape(-1, output.shape[2])  
 target = target[1:].reshape(-1)  
 optimizer.zero\_grad()  
 loss = criterion(output, target)  
 losses.append(loss.item())  
 # Back prop  
 loss.backward()  
 # Clip to avoid exploding gradient issues, makes sure grads are  
 # within a healthy range  
 torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1)  
 # Gradient descent step  
 optimizer.step()  
  
 training\_losses.append(loss.item())  
 mean\_loss = sum(losses) / len(losses)  
 scheduler.step(mean\_loss)  
 # Print the final epoch  
 print(f"[Epoch {num\_epochs} / {num\_epochs}]")  
 # Iterate over the dataset and extract PLI sequences  
 for s in sentences:  
 translated = translate\_sequence(  
 s['pli'], pli, ktl, device, max\_length=50  
 )  
 transpiled, level = transpile\_sequence({  
 'code': translated,  
 'context': s['context']  
 }, level)  
 print(f"{transpiled}")  
 # Translate and transpile here if needed  
 last\_5\_losses = training\_losses[-5:]  
 print("Last 5 training losses:", last\_5\_losses)  
  
 plt.plot(training\_losses, label='Training Loss')  
 plt.xlabel('Iterations')  
 plt.ylabel('training\_losses')  
 plt.title('Training Loss Over Iterations')  
 plt.legend()  
 plt.show()  
if save\_model:  
 checkpoint = {  
 "state\_dict": model.state\_dict(),  
 "optimizer": optimizer.state\_dict(),  
 }  
 save\_checkpoint(checkpoint)

[Epoch 0 / 300]  
Translated example sentence:  
<pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>  
<pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> () () <pad> <pad> <pad> <pad>  
<pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>  
Last 5 training losses: []  
[Epoch 50 / 300]  
Translated example sentence:  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.11536741256713867, 0.13401655852794647, 0.09288042783737183, 0.0880018100142479, 0.0810730904340744]  
[Epoch 100 / 300]  
Translated example sentence:  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.002143696416169405, 0.0021836746018379927, 0.002142140408977866, 0.0020195324905216694, 0.0023125687148422003]  
Epoch 125: reducing learning rate of group 0 to 3.0000e-05.  
Epoch 145: reducing learning rate of group 0 to 3.0000e-06.  
[Epoch 150 / 300]  
Translated example sentence:  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.0009272661409340799, 0.0009802684653550386, 0.0009581581107340753, 0.0009119620081037283, 0.0009031783556565642]  
Epoch 156: reducing learning rate of group 0 to 3.0000e-07.  
Epoch 167: reducing learning rate of group 0 to 3.0000e-08.  
Epoch 178: reducing learning rate of group 0 to 3.0000e-09.  
[Epoch 200 / 300]  
Translated example sentence:  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.0009122670162469149, 0.0009905740153044462, 0.0010367712238803506, 0.0011590280337259173, 0.0010337315034121275]  
[Epoch 250 / 300]  
Translated example sentence:  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.001014587003737688, 0.0008746228413656354, 0.001056693377904594, 0.0011161871952936053, 0.001087463111616671]  
[Epoch 300 / 300]  
fun main (args: Array<String>)  
{  
}  
Last 5 training losses: [0.0009633959271013737, 0.0010461953934282064, 0.0010198841337114573, 0.0018401391571387649, 0.0009006673935800791]  
=> Saving checkpoint

/usr/local/lib/python3.9/dist-packages/torchtext/data/batch.py:23: UserWarning: Batch class will be retired soon and moved to torchtext.legacy. Please see the most recent release notes for further information.  
 warnings.warn('{} class will be retired soon and moved to torchtext.legacy. Please see the most recent release notes for further information.'.format(self.\_\_class\_\_.\_\_name\_\_), UserWarning)



## Running the Transpiler

And as little bonus we can show you an example of how our transpiller works. We define a **run\_model** function where we’re processing a PL1 file, which contains original PL1 code. We then load our pre-trained model and its optimizer using the checkpoint file. We set up a lexer and parser for the PL1 code using ANTLR4. We generate a dataset from the parsed code using a visitor pattern, which iterates through the AST (Abstract Syntax Tree) generated by the parser. For each statement in the PL1 code, we translate it into Kotlin using our sequence-to-sequence model and transpile it into Kotlin code. The transpiled Kotlin code is accumulated and printed as the output. This function essentially automates the process of translating our PL1 code to Kotlin

def run\_model(filename):  
  
 with open(filename, 'r') as file:  
 original\_code = file.read()  
 print("PL1:")  
 print(original\_code)  
 print()  
  
 load\_checkpoint(torch.load("checkpoint.pth.tar"), model, optimizer)  
 # Lexer setup  
 input\_stream = FileStream(filename)  
 lexer = PLILexer(input\_stream)  
 stream = CommonTokenStream(lexer)  
  
 # Parser setup  
 parser = PLIParser(stream)  
 tree = parser.program()  
  
 # Dataset generation  
 visitor = PLIVisitor()  
 statements = visitor.visit(tree)  
  
 # Accumulate transpiled sequences  
 transpiled\_code = ""  
 level = 0  
 for s in statements:  
 translated = translate\_sequence(  
 s["pli"], pli, ktl, device, max\_length=50  
 )  
 transpiled, level = transpile\_sequence({  
 'code': translated,  
 'context': s['context']  
 }, level)  
 transpiled\_code += transpiled+ "\n"  
  
 # Print the entire block of transpiled code  
 print("KTL:")  
 print("\n" + transpiled\_code)  
  
# Example usage:  
filename = "FIB.PLI" # Replace with the actual filename  
run\_model(filename)

PL1:  
  
 Factorial: proc options (main);  
 dcl (n,result) fixed bin(31);  
 n = 5;  
 result = Compute\_factorial(n);  
  
 end Factorial;  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 /\* Subroutine \*/  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 Compute\_factorial: proc (n) returns (fixed bin(31));  
 dcl n fixed bin(15);  
 if n <= 1 then  
 return(1);  
  
 return( n\*Compute\_factorial(n-1) );  
  
 end Compute\_factorial;  
  
  
=> Loading checkpoint  
KTL:  
  
fun main (args: Array<String>)  
{  
 var n : Int  
 var result : Int  
 n = 5  
 result = compute\_factorial(n)  
}  
fun compute\_factorial(n : Int) : Int  
{  
 if(n<=1)  
 {  
 return 1  
 }  
 return n\*compute\_factorial(n-1)  
}

# Conclusion

And there you have it ! This notebook provides a comprehensive look at building and training a Seq2Seq Transformer model in PyTorch, from scratch, for in this case the specific task of transpiling PL/I code to Kotlin. The model not only demonstrates the capabilities of neural networks in handling language translation tasks but also highlights the flexibility of the Transformer architecture in processing sequence data.

For those interested in further exploring machine translation models, another article is available where I utilize the Hugging Face API to achieve similar tasks. This API simplifies many of the steps covered here and provides powerful tools to be able to deploy machine translation models in production environments.