Sentence Classification with Transformers

In this exercise you will implement a Transformer (https://arxiv.org/pdf/1706.03762.pdf) and use it to judge the grammaticality of English sentences.

A quick note: if you receive the following TypeError "super(type, obj): obj must be an instance or subtype of type", try restarting your kernel and re-running all cells. Once you have finished making changes to the model constuctor, you can avoid this issue by commenting out all of the model instantiations after the first (e.g. lines starting with "model = ClassificationTransformer").

In [10]:
```
import numpy as np
import csv
import torch

from gt_7643.transformer import ClassificationTransformer

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

The Corpus of Linguistic Acceptability (CoLA)

The Corpus of Linguistic Acceptability (CoLA (https://nyu-mll.github.io/CoLA/)) in its full form consists of 10657 sentences from 23 linguistics publications, expertly annotated for acceptability (grammaticality) by their original authors. Native English speakers consistently report a sharp contrast in acceptability between pairs of sentences. Some examples include:

`What did Betsy paint a picture of?` (Correct)

`What was a picture of painted by Betsy?` (Incorrect)

You can read more info about the dataset here (https://arxiv.org/pdf/1805.12471.pdf). This is a binary classification task (predict 1 for correct grammar and 0 otherwise).

Can we train a neural network to accurately predict these human acceptability judgements? In this assignment, we will implement the forward pass of the Transformer architecture discussed in class. The general intuitive notion is that we will *encode* the sequence of tokens in the sentence, and then predict a binary output based on the final state that is the output of the model.

Load the preprocessed data

We've appended a "CLS" token to the beginning of each sequence, which can be used to make predictions. The benefit of appending this token to the beginning of the sequence (rather than the end) is that we can extract it quite easily (we don't need to remove paddings and figure out the length of each individual sequence in the batch). We'll come back to this.

We've additionally already constructed a vocabulary and converted all of the strings of tokens into integers which can be used for vocabulary lookup for you. Feel free to explore the data here.

In [11]:
```
train_inxs = np.load('./gt_7643/datasets/train_inxs.npy')
val_inxs = np.load('./gt_7643/datasets/val_inxs.npy')
train_labels = np.load('./gt_7643/datasets/train_labels.npy')
val_labels = np.load('./gt_7643/datasets/val_labels.npy')

# load dictionary
word_to_ix = {}
with open("./gt_7643/datasets/word_to_ix.csv", "r") as f:
    reader = csv.reader(f)
    for line in reader:
        word_to_ix[line[0]] = line[1]
print("Vocabulary Size:", len(word_to_ix))

print(train_inxs.shape) # 7000 training instances, of (maximum/padded) length 43 words.
print(val_inxs.shape) # 1551 validation instances, of (maximum/padded) length 43 words.
print(train_labels.shape)
print(val_labels.shape)

# load checkers
d1 = torch.load('./gt_7643/datasets/d1.pt')
d2 = torch.load('./gt_7643/datasets/d2.pt')
d3 = torch.load('./gt_7643/datasets/d3.pt')
d4 = torch.load('./gt_7643/datasets/d4.pt')
```

```
Vocabulary Size: 1542
(7000, 43)
(1551, 43)
(7000,)
(1551,)
```

Transformers

We will be implementing a one-layer Transformer **encoder** which, similar to an RNN, can encode a sequence of inputs and produce a final output state for classification. This is the architecture:

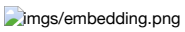
You can refer to the original paper (https://arxiv.org/pdf/1706.03762.pdf) for more details.

Instead of using numpy for this model, we will be using Pytorch to implement the forward pass. You will not need to implement the backward pass for the various layers in this assigment.

The file `gt_7643/transformer.py` contains the model class and methods for each layer. This is where you will write your implementations.

Deliverable 1: Embeddings

We will format our input embeddings similarly to how they are constructed in BERT (source of figure) (https://arxiv.org/pdf/1810.04805.pdf). Recall from lecture that unlike a RNN, a Transformer does not include any positional information about the order in which the words in the sentence occur. Because of this, we need to append a positional encoding token at each position. (We will ignore the segment embeddings and [SEP] token here, since we are only encoding one sentence at a time). We have already appended the [CLS] token for you in the previous step.



Your first task is to implement the embedding lookup, including the addition of positional encodings. Open the file `gt_7643/transformer.py` and complete all code parts for `Deliverable 1`.

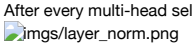
In [3]:
```
inputs = train_inxs[0:2]
inputs = torch.LongTensor(inputs)

model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedforward=2048, dim_k=96,
                                  dim_v=96, dim_q=96, max_length=train_inxs.shape[1])

embeds = model.embed(inputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(embeds, d1)).item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017988268518820405
```

Deliverable 2: Multi-head Self-Attention

Attention can be computed in matrix-form using the following formula:


We want to have multiple self-attention operations, computed in parallel. Each of these is called a *head*. We concatenate the heads and multiply them with the matrix `attention_head_projection` to produce the output of this layer.

After every multi-head self-attention and feedforward layer, there is a residual connection + layer normalization. Make sure to implement this, using the following formula:


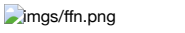
Open the file `gt_7643/transformer.py` and implement the `multihead_attention` function. We have already initialized all of the layers you will need in the constructor.

In [4]:
```
hidden_states = model.multi_head_attention(embeds)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(hidden_states, d2)).item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017100314144044115
```

Deliverable 3: Element-Wise Feed-forward Layer

Open the file `gt_7643/transformer.py` and complete code for `Deliverable 3` : the element-wise feed-forward layer consisting of two linear transformers with a ReLU layer in between.



In [5]:
```
model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedforward=2048, dim_k=96,
                                  dim_v=96, dim_q=96, max_length=train_inxs.shape[1])

outputs = model.feedforward_layer(hidden_states)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(outputs, d3)).item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017168164486065507
```

Deliverable 4: Final Layer

Open the file `gt_7643/transformer.py` and complete code for `Deliverable 4` , to produce binary classification scores for the inputs based on the output of the Transformer.

In [6]:
```
model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedforward=2048, dim_k=96,
                                  dim_v=96, dim_q=96, max_length=train_inxs.shape[1])

scores = model.final_layer(outputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(scores, d4)).item()) # should be very small (<1e-5)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 1.8956918665935518e-06
```

Deliverable 5: Putting it all together

Open the file `gt_7643/transformer.py` and complete the method `forward` , by putting together all of the methods you have developed in the right order to perform a full forward pass.

In [12]:
```
inputs = train_inxs[0:2]
inputs = torch.LongTensor(inputs)

outputs = model.forward(inputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(outputs, scores)).item()) # should be very small (<1e-5)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 1.9999999949504854e-06
```

Great! We've just implemented a Transformer forward pass for text classification. One of the big perks of using PyTorch is that with a simple training loop, we can rely on automatic differentation (autograd (https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html)) to do the work of the backward pass for us. This is not required for this assignment, but you can explore this on your own.

Make sure when you submit your PDF for this assignment to also include a copy of `transformer.py` converted to PDF as well.