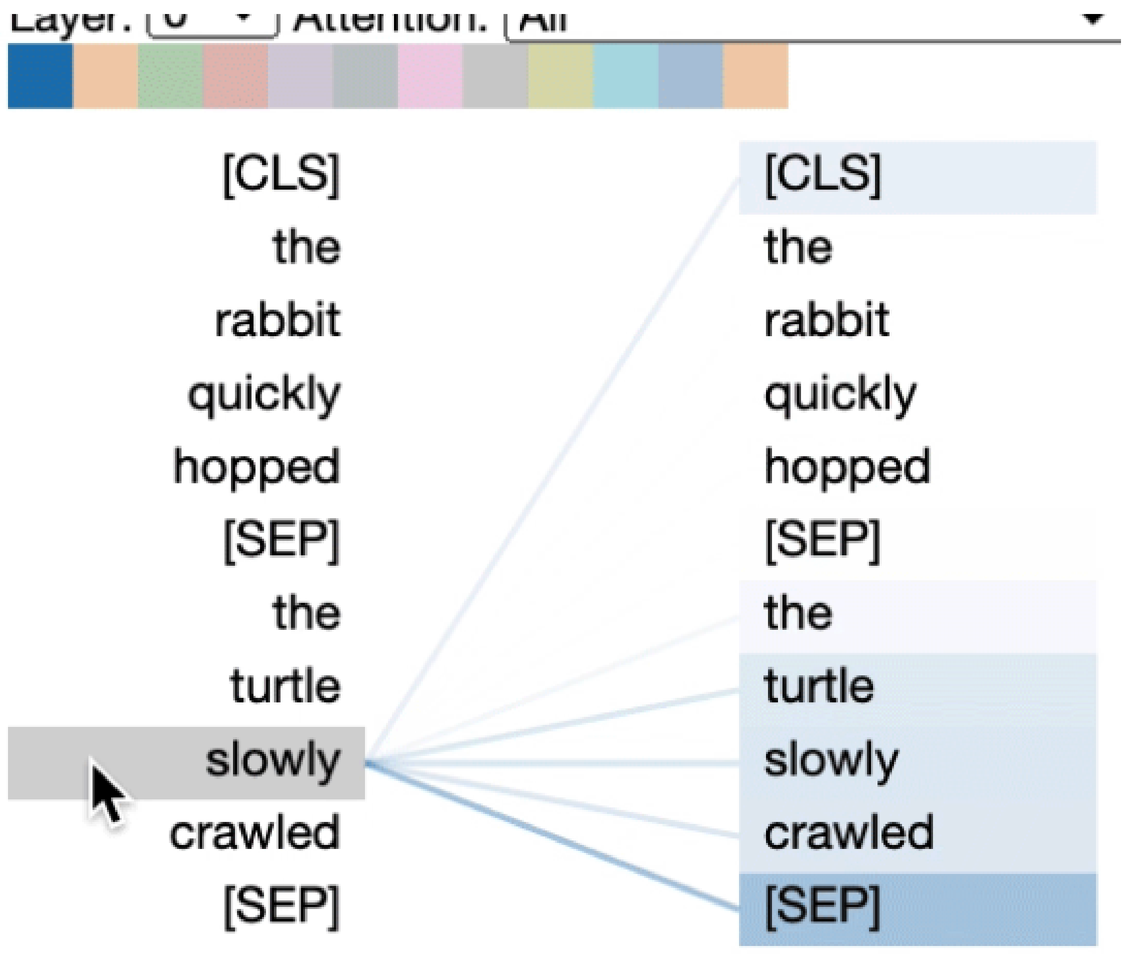
CW1_Report

February 9, 2024

0.0.1 CW1 Report (30% Report)

Part A) Understanding Self-Attention Mechanism **Questions** 1) **What is the concept of single-head self-attention?**

This refers to a mechanism which is also known as the scaled dot product attention, this mechanism allows the model to weigh the different tokens within a input sequence capturing the different dependencies between tokens via a representation. This single-head self attention model will produce attention matrix, which can be used to determine meaning through representation, or be used in the synthesis of the output of a decoder.



Showing the attention one word has to others. We are in essence learning the structure of language, or image pixel values by looking at how related via trained weights.

2) **What is the concept of multi=head self-attention?**

Multi-headed self attention models, use the concept of single headed self attention modules in parallel, concatenating them, this is because one single head can generate one representation for the input tokens, via the weights for queries, key and values, however multiple allow the module to learn multiple different representations which add to the models robustness in understanding inputs.

3) **What is the difference between single-head and multi-head self-attention?**

The main difference between these two mechanisms is that single head is run once, and such multihead runs multiple dot scaled products, which generates multiple attention matrices which is then concatenated, compared to single, where one attention matrix is used. Also in pratice, when using multi-headed self-attention the dimension of the dimension of keys, values, queries is dependant on the number of heads (based on youtube video).

Excercises

Note: For all excercises that mention dimensions, batches have been ignored, as these ask theoretical questions, however, this would only mean initial dimension B is preappended to all dimensions.

1) Calculate the dimensions of W_q, W_v, W_k in "Single-head Attention" section?

- W_q dimension is (256, 64), in which (Embedding Dimension, Head Embedding Dimension)
- W_v dimension is (256, 64), in which (Embedding Dimension, Head Embedding Dimension)
- W_k dimension is (256, 256) in which (Embedding Dimension, Embedding Dimension)

[ ]:
```
embdim = 256  # D
headdim = 64  # Internal D
tokens = torch.randn(1, 5, embdim) # batch, tokens, embedding
Wq = torch.randn(embdim, headdim) / math.sqrt(embdim)
Wk = torch.randn(embdim, headdim) / math.sqrt(embdim)
Wv = torch.randn(embdim, embdim) / math.sqrt(embdim)

# Dimensions of these weights are:

print(f"Wq dimensions: {Wq.shape}")
print(f"Wk dimensions: {Wk.shape}")
print(f"Wv dimensions: {Wv.shape}")
```

2) **Calculate the dimensions of W_q, W_v, W_k in "Multi-head Attention" section?**

- For all three weights, we need to account for the number of heads also in the architecture, the weight dimensions are **embdim, headcnt * headdim**:
- W_dim = (Embedding Dimension, (Head Count * Head Embedding))
- Which is (768, 768)

```
[ ]: embdim = 768
     headcnt = 12
     headdim = embdim // headcnt
     # print(headdim)
     assert headdim * headcnt == embdim
     tokens = torch.randn(1, 5, embdim) # batch, tokens, embedding

     # We use all the 256, ( 768)  ~ which is (256), (64 * 12 (heads))
     Wq = torch.randn(embdim, headcnt * headdim) / math.sqrt(embdim) # heads packed␣
      ↪in a single dim
     Wk = torch.randn(embdim, headcnt * headdim) / math.sqrt(embdim) # heads packed␣
      ↪in a single dim
     Wv = torch.randn(embdim, headcnt * headdim) / math.sqrt(embdim) # heads packed␣
      ↪in a single dim


     # Dimensions of these weights are:

     print(f"Wq dimensions: {Wq.shape}")
     print(f"Wk dimensions: {Wk.shape}")
     print(f"Wv dimensions: {Wv.shape}")
```

3) **Show the weight of the causal attention mask in "Causal attention mask" section.**

- The shape of such casual mask is that of all zeros, where a upper traingle is a large negative number like -inf, but in this case -10000
- It has dimension (N, N) which is the token numbers x token numbers, below is an example.

```
[12]: token_num = 5
      attn_mask = torch.ones(token_num,token_num,)
      attn_mask = -1E4 * torch.triu(attn_mask,1)
```

4) **Show the figure of the causal attention mask in "Causal attention mask" section.**

```
[14]: # Answering Excercise 4):
      attn_mask
```

```
[14]: tensor([[    -0., -10000., -10000., -10000., -10000.],
              [    -0.,     -0., -10000., -10000., -10000.],
              [    -0.,     -0.,     -0., -10000., -10000.],
              [    -0.,     -0.,     -0.,     -0., -10000.],
              [    -0.,     -0.,     -0.,     -0.,     -0.]])
```

Part B) Understanding the structure of Transformer Questions 1) **What are the main compositional layers of a transformer?**

The transformer consists of transformer blocks, in which each of these blocks, take as input inputs that have been embedded or the output of another transformer block, within these transformer blocks, there are usually 2-3 sublayers, depending on whether it is a encoder or decoder.

These sublayers include Multi-head attention module and a Feedforward module, these are connected by layer normalisation and residual connections (skip connection). These move forward through the model to the output of the transformer which is the different representations of the different tokens, these can be obtained and futher used in another MLP head, or pooled for use in decoding.

2) **What is the purpose of layer normalization in a transformer?**

Normalisation in neural networks is important in order to stablise learning, via preventing extremely large numbers that may cause exploding gradients and numeric instabilities. Two main ways for normalisation which include layer and batch normalisation, the reasoning for layer normalisation compared to batch normalisation is that no reliance on batches is needed to perform normalisation, such that we can still learn in parallel without needing to deal with normalisation in batches.

3) **What is the difference between the training and testing stage of a transformer?**

In our coursework, training occurs each epoch, and then we use model.evaluate to test this model. The basic idea, is that in the training, we are updating the weights through backpropagation of these keys, values and queries weight sets whilst in inference/testing stage, we use such weights to output the thoughtout representations of each of these inputs. In the context of using a decoder as part of the transformer model, the output of the decoder will be compared to the target output in order to calculate the loss.

4) **What is the skip connection in a transformer?**

Skip connections, were also used in RNNs, and image classifiers with really large number of hidden layers, this is because when we have long structures, like those with a number of encoder blocks, it becomes harder to train due to gradients becoming increasingly smaller through each backpropogation of the layers, also known as vanishing gradient problem, with this residual connection, we can make sure there are larger gradients during backpropogations by creating connections that flow past these differentiable operations. So thier use in transformers makes sense.

5) **Why does a transformer usually use multi-head attention instead of single head one?**

We usually want to learn multiple representations of the data, it is like having a second opinion on a decision or a third or a fourth, you get a really informed decision, in relation to transformers, we get a really good representation that is robust because we combine all of the knowledge learned from these multiple attention blocks making them perform better up to a certain point (mentioned in lecture).

Excercises

1) **To show the projection weight (WK) of K in the multi-head attention layer (Tip: check the PyTorch build-in class nn.MultiheadAttention).**

[ ]:
```
# In essense, the data for WK is within the function of mha layer,
print(mha.in_proj_weight.shape) # 3 * embdim x embdim
mha.in_proj_weight.data = torch.cat([Wq, Wk, Wv], dim=1).T
```

```
# If we want to obtain weights specifically for query.
Wk = mha.in_proj_weight.data[1]
```

2) **To show the projection weight (WQ) of Q in the multi-head attention layer**

[ ]:
```
Wq = mha.in_proj_weight.data[0]
```

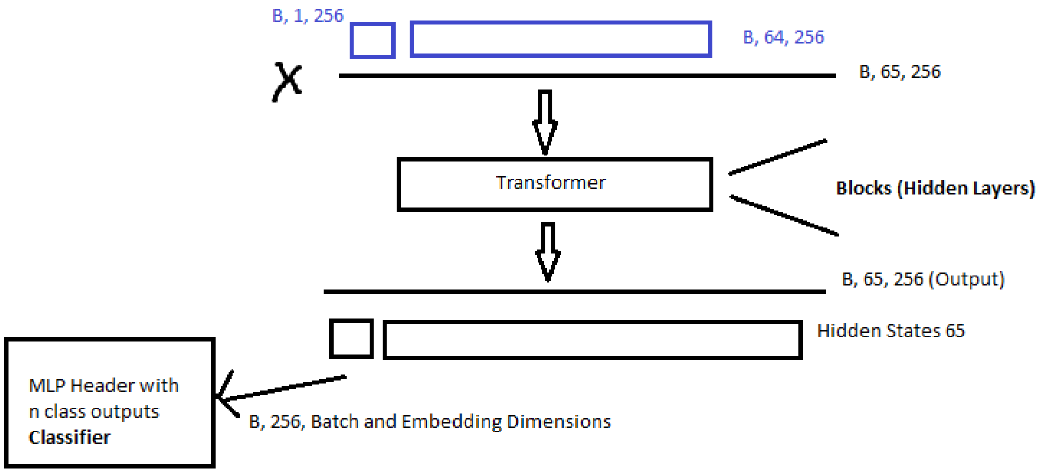
3) **To show the projection weight (WV) of V in the multi-head attention layer**

[ ]:
```
Wv = mha.in_proj_weight.data[2]
```

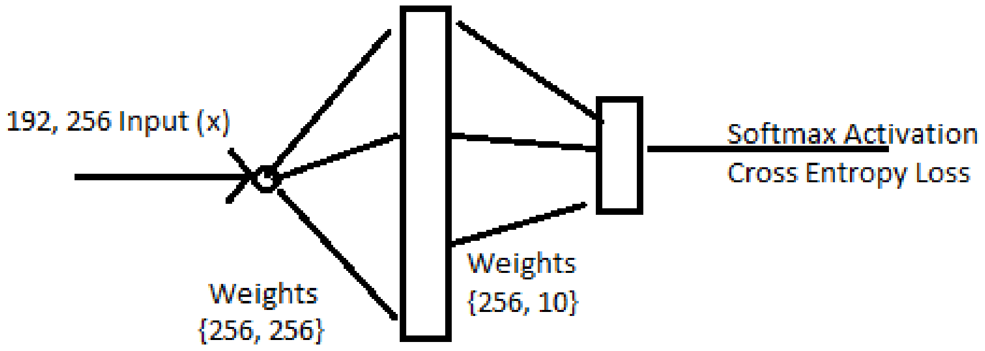
Part C) Image classification with Transformer Questions 1) **How to use a trained Transformer to perform image classification (testing stage)?**

The model is trained by using the class tokens encoded representation, which is then passed onto the MLP classifier, which is optimised based on CrossEntropyLoss. This is accessed through the `last_hidden_state` attribute of the model as the first index.

I drew some diagrams to show the input embeddings are inputted to transformer, to output.



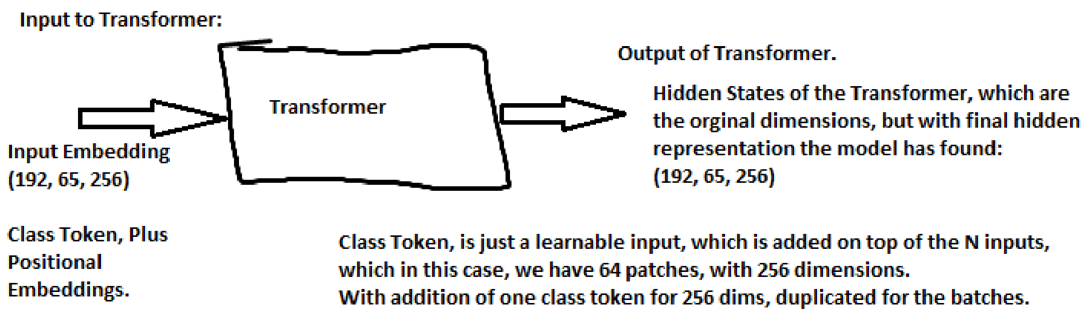
Output of class token hidden state to MLP head diagram.



2) What are the input and the output of the Transformer?

The input and output of the transformer is the same, in the case of this coursework, I have obtained dimension (192, 65, 256)

These dimensions represents: Batch Size, Class + Input Tokens , Model Embedding Size.


Input to Transformer:
Transformer
Output of Transformer.
Hidden States of the Transformer, which are the orginal dimensions, but with final hidden representation the model has found: (192, 65, 256)
Input Embedding (192, 65, 256)
Class Token, Plus Positional Embeddings.
Class Token, is just a learnable input, which is added on top of the N inputs, which in this case, we have 64 patches, with 256 dimensions.
With addition of one class token for 256 dims, duplicated for the batches.

3) What augmentations are used for training data?

These are shown in the code below, however there are operations such as random crop, random horizontal flip , normalisation and different transformation.

```
[ ]: transforms.RandomHorizontalFlip(),
         transforms.RandomCrop(32, padding=4),
         transforms.ToTensor(),
         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
     ]))
     # augmentations are super important for CNN trainings, or it will overfit very␣
      ↪fast without achieving good generalization accuracy
```

4) What's the difference between the preprocess of the train set and the validation set?

In the validation set of data, augmentation include converting to tensor in order to normalise, however does not include random cropping of the images, as we are simply testing the models performance, and not training, this transformation allows the model to learn, but we are just evaluating.

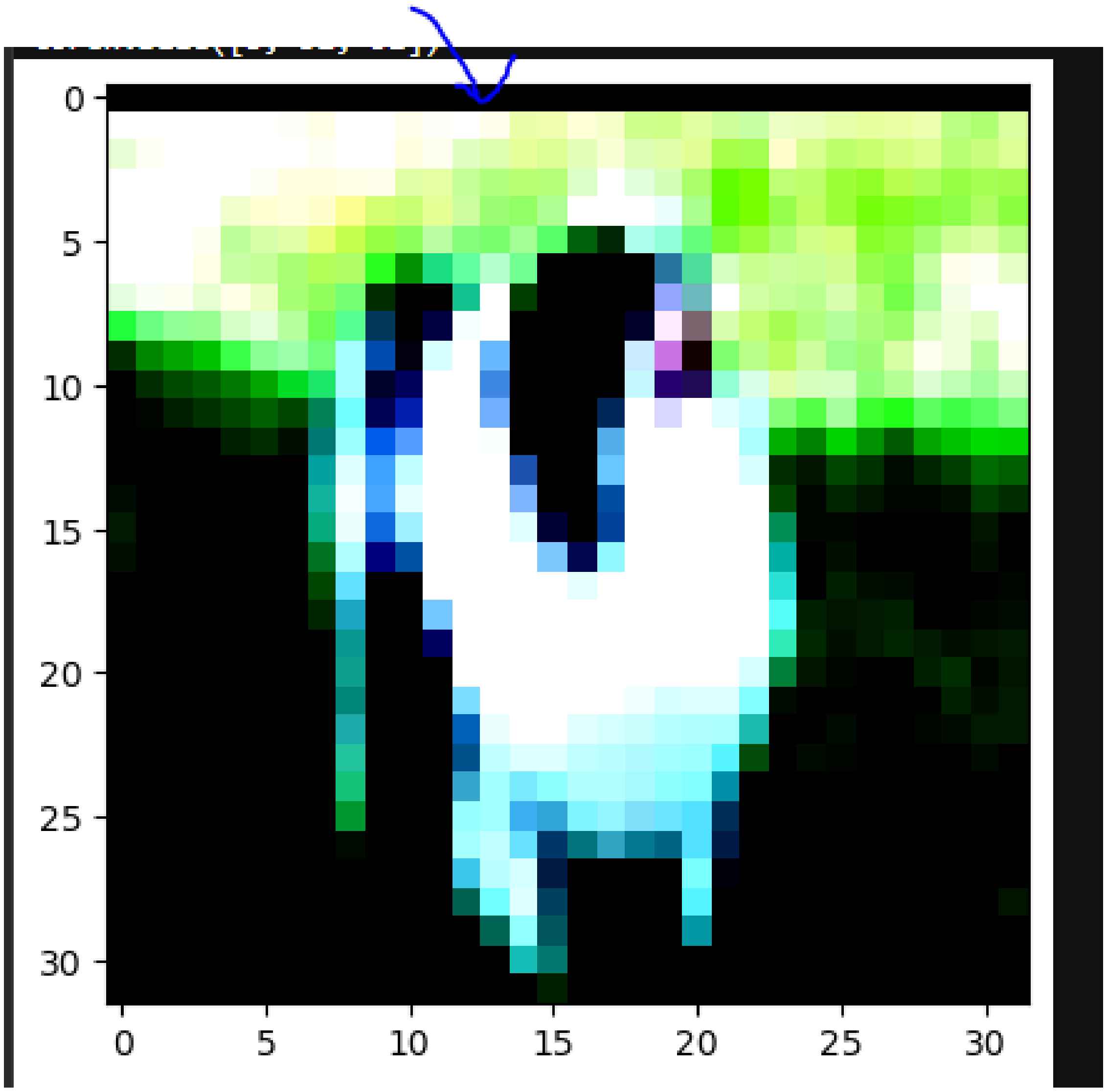


Image of random cropping shown, illustrating no need for this to be in the testing dataset, as we simply want to determine how well the model is, not training it in evaluation.

5) **Why augmentations for training data is necessary?**

With these transforms, this creates more variances between dataset, in the attempt to make the system more robust to rotation and translation variance, and better understand the images, this allows the model to try and learn structure within the image such as inductive bias CNNs have, such as rotation invariance and locality understanding.

Excercises

1) **To load and preprocess the train set and validation set from the CIFAR10 dataset.**

```
[ ]: !mkdir data
dataset = CIFAR10(root='./data/', train=True, download=True, transform=
transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
]))
# augmentations are super important for CNN trainings, or it will overfit very␣
 ↪fast without achieving good generalization accuracy
val_dataset = CIFAR10(root='./data/', train=False, download=True,␣
 ↪transform=transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.
 ↪2010)),]))
#%%
batch_size = 192 # 96
train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

2) **To transform training images to patch embedding.**

[ ]:
```
patch_embed = nn.Conv2d(3, config.hidden_size, kernel_size=4, stride=4).cuda()

pbar = tqdm(train_loader, leave=False)
for i, (imgs, labels) in enumerate(pbar):
    patch_embs = patch_embed(imgs.cuda()) # Transforms images to patch␣
 ↪embeddings.
```

3) **To add a class token to the patch embedding.**

[ ]:
```
# Define Class Token.
CLS_token = nn.Parameter(torch.randn(1, 1, config.hidden_size, device="cuda") /␣
 ↪math.sqrt(config.hidden_size))

# Concat class token to patch embeddings.
input_embs = torch.cat([CLS_token.expand(imgs.shape[0], 1, -1), patch_embs],␣
 ↪dim=1)
```

4) **To feed the input image embedding into the Transformer.**

[ ]:
```
# Model should be defined, and also shown in the coursework notebook,
output = model(inputs_embeds=input_embs)
```

5) **To plot the training loss curve to show its variation with the epoch.**

[ ]:
```
EPOCH_RANGES = [10, 20, 30]

for epoch_range in EPOCH_RANGES:
    plt.title(f"Loss/Epoch ~ {epoch_range} Epochs")
    rev_loss = loss_list.copy()
    # print(rev_loss)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.plot(rev_loss[:epoch_range])
```

6) **To plot the accuracy score curve to show its variation with the epoch.**

[ ]:
```
EPOCH_RANGES = [10, 20, 30]
for epoch_range in EPOCH_RANGES:
    plt.title(f"Accuracy/Epoch ~ {epoch_range} Epochs")
    rev_acc = acc_list.copy()
    rev_acc.reverse()
    # print(rev_loss)
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.plot(rev_acc[:epoch_range])
```