ECE 60146 Deep Learning Homework 9

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

April 30, 2023

1 Introduction

So far, in previous homeworks, we dealt with two major architectural elements in the neural networks meant for deep learning (DL): convolutional layers and recurrence layers. Recently, in addition to these, a new element was proposed, which is attention layers. The objectives of this homework are as follows: **i)** to gain insights into the multi-headed self-attention mechanism and to understand the transformer architecture in deeper level. **ii)** To understand how the original transformer, which was designed for language translation, can be adopted to process in Vision Transformer (ViT). **iii)** To implement our own ViT for image classification. Before embarking on the homework solution, I want to explain the attention mechanism of ViT with the equations so that I can refer them later in my code.

1.1 Self-Attention Mechanism in ViT

I will first explain the attention mechanism by assuming number of attention head equals 1, then I will generalize it. To begin with, instead of images, if we have words, then we would represent each word by its pre-trained embedding, i.e. `word2vec`, or we may initialie weights randomly and train them during the training of the classifier. In our example, instead of words in a sentence, we have images. One idea of using an image as if it is a sentence is dividing it into patches and each patch can be regarded as a word or elements in a sequence. For example, if image size is 64×64, and patch

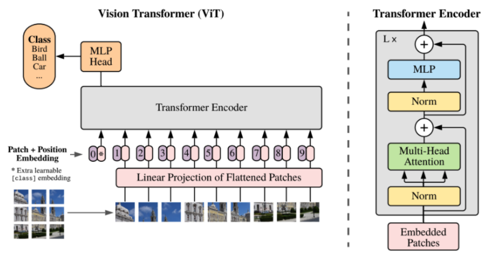


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

Figure 1: Illustration of the ViT architecture

size is 16×16, then there will be 16 patches. Similar to word embedding, we project each patch to a lower dimensional space by a linear layer. These low-dimensional vectors are the embeddings of the patches. So, ignoring the batch size, after the linear layer, we will have $X \in \mathbb{R}^{N_w \times M}$, where

N_w and M represent the maximum words in a sequence and the size of embeddings. Then, for each embedding, we obtain three type of tensors which are query, key and value tensors. They are learned by the following equations:

$$\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \end{aligned} \tag{1}$$

where $Q, K, V \in \mathbb{R}^{N_w \times M}$, and $W_Q, W_K, W_V \times \mathbb{R}^{M \times M}$. First triplet are queue, key and value matrices whose rows correspond to each words' queue, key and value vectors respectively. Second triplet consists of the corresponding layers for queue, key, and value vectors. What do these vectors represent? At which part of this there is an attention? The idea of attention is we train the network by considering the relation between patches. For instance, to check the relevance of patch 1 to patch 2, we take the dot product of the query vector of the first path and key vector of the second patch. After calculating the relevance of patch 1 to all other patches, we calculate its final value by doing weighted summation of the value vectors, which represent the patches, of other patches in the image. Doing so, we obtain **attention-enriched** embeddings. These calculations are given below:

$$Z = \frac{Softmax(QK^T)}{\sqrt{M}} V \tag{2}$$

As we do weighted summation, we use Softmax layer so that all weights sum up to 1. Furthermore, as the embedding size increases, the variance of the embedding vectors increase, so we normalize it with $\sqrt{M}$. Z matrix is attention-enriched embeddings and it is passed to a fully-connected neural network inside the encoder.

Experimentally, it has been observed that using sole embeddings in the attention layer is not sufficient for good generalization performance so they divided the embeddings into partitions, which are called **attention heads**. Thus, procedure that was explained above is exactly applied to each attention head or partition. Different than that, if embedding size is 100 and there are 5 attention heads, then M becomes $100/5 = 20$ instead of 100. As you may notice, representation power of that method is more powerful because we define 3 more linear layer for each extra head so number of parameters in the model increases.

These attention layers are used in cascade of encoders, which constitutes the master encoder of the transformer. One example of that can be seen in Figure 2. Inside the encoder of ViT, that type of network was implemented. I used the code written by Prof. Kak and also I wrote the multi-headed self-attention layer with `torch.einsum()` function as I will explain in the bonus section.

Rest of the homework is structured as follows. In Section 2, I explain the dataset that has been used. In Section 3, I explain the implementation of the initial layer, positional embeddings and class token in ViT, Figure 1. Moreover, in that section, I explain the design of self-attention with `torch.einsum` function, which is the bonus section. In Section 4, I present the results and discuss them. In Section 5, I talk about lessons learned and lastly, I explain how to run the code properly.

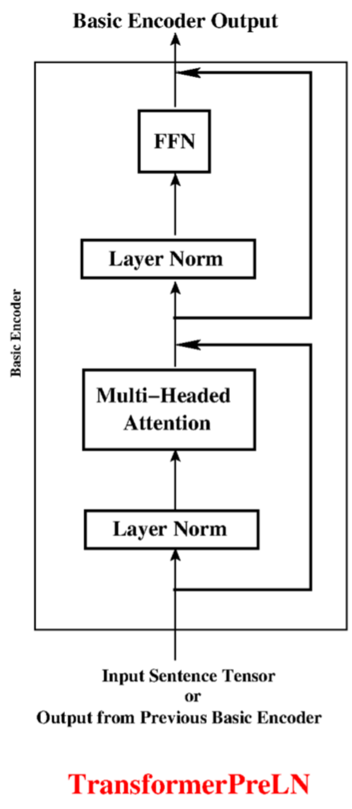


Figure 2: Transformer Encoder

2 Dataset

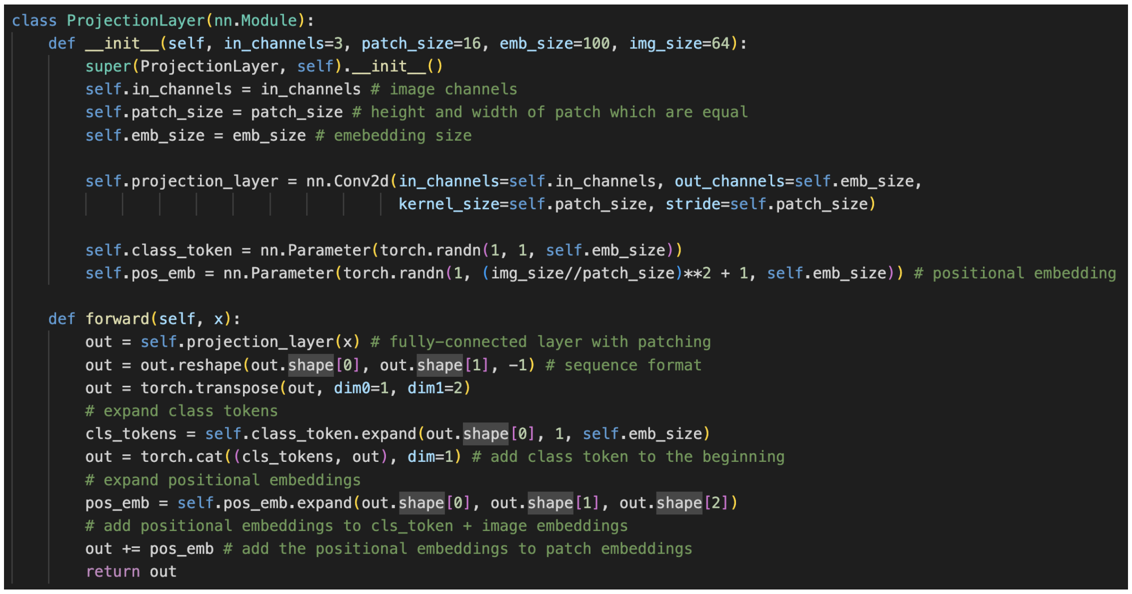
I used the same dataset in HW4 as instructed in the homework 9 pdf. Thus, I used my previous code to generate the dataset from COCO 2014 dataset. Dataset consists of train and validation sets, which include 7.5k train images and 2.5k validation images. It includes five different classes which are bus, cat, dog, pizza, and airplane. Dataset is balanced, that is, each 1.5k image in train set belongs to a particular class and similarly for test set, each 500 image belongs to one class. I am not going to explain the code and COCO API as it is the same as in HW4 and I explained it there.

3 Method

In this section, I will explain my ViT implementation. I used `ViTHelper.py` module to implement the self-attention and encoder part of the transformer, which were explained in Section 1.1. I also implemented the self-attention layer using `torch.einsum()` but I will explain that later. First, I implemented the projection layer, which positional embedings, class tokens, and projection of patches to their embeddings via linear layer.

3.1 ProjectionLayer

I implemented the projection layer in my `model.py` module. It can be found in Figure 3. At initialization, it takes four arguments: number of channels in images, patch size, embedding size, and image size. In `__init__()` method, there are one projection layer and two learnable parameters. Projection layer performs both patching and forward pass through a linear layer operations



```python
class ProjectionLayer(nn.Module):
    def __init__(self, in_channels=3, patch_size=16, emb_size=100, img_size=64):
        super(ProjectionLayer, self).__init__()
        self.in_channels = in_channels # image channels
        self.patch_size = patch_size # height and width of patch which are equal
        self.emb_size = emb_size # emebedding size

        self.projection_layer = nn.Conv2d(in_channels=self.in_channels, out_channels=self.emb_size,
                                          kernel_size=self.patch_size, stride=self.patch_size)

        self.class_token = nn.Parameter(torch.randn(1, 1, self.emb_size))
        self.pos_emb = nn.Parameter(torch.randn(1, (img_size//patch_size)**2 + 1, self.emb_size)) # positional embedding

    def forward(self, x):
        out = self.projection_layer(x) # fully-connected layer with patching
        out = out.reshape(out.shape[0], out.shape[1], -1) # sequence format
        out = torch.transpose(out, dim0=1, dim1=2)
        # expand class tokens
        cls_tokens = self.class_token.expand(out.shape[0], 1, self.emb_size)
        out = torch.cat((cls_tokens, out), dim=1) # add class token to the beginning
        # expand positional embeddings
        pos_emb = self.pos_emb.expand(out.shape[0], out.shape[1], out.shape[2])
        # add positional embeddings to cls_token + image embeddings
        out += pos_emb # add the positional embeddings to patch embeddings
        return out
```

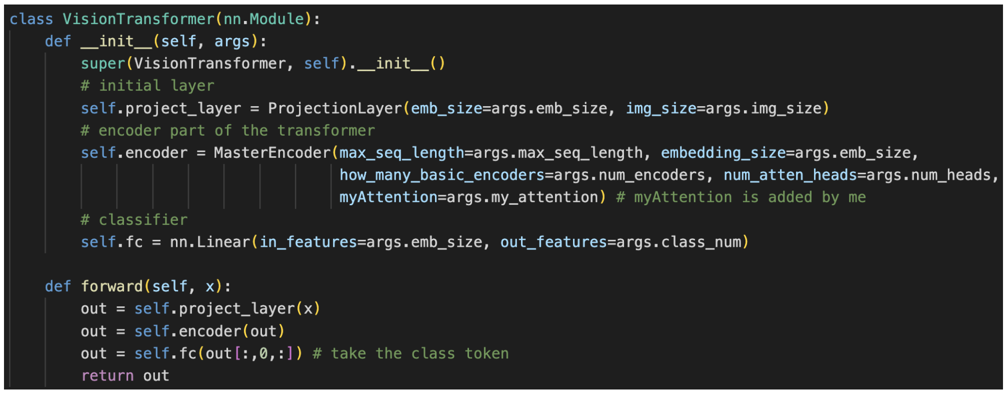
Figure 3: Implementation of `ProjectionLayer`

simultaneously. I did the following trick: let's say image size is 64×64 and patch size is 16×16. I set the *stride* $= 16$ so that each kernel exactly covers a patch without any overlap with other patches. In that scenario, one element-wise multiplication of the kernel and its corresponding patch is equivalent to flattening that patch and connecting all the entries to a single neuron. As we want to have an embedding of size different than 1, let's say 100, instead of a single neuron, we can use 100 different kernels and get 100 features/output neurons at the end! So, I set the `in_channels` to 3 and `out_channels` to 100 and perform the patching and projection at the same time. Moreover, class token is a vector which keeps the classification information during the flow of the embeddings through encoders of the transformer. This was shown to be effective in BERT so they used in ViT too. Lastly, to inform the network about the position of the patches, it is good idea to use positional embeddings. Similar to words or patches, we want network to learn the embeddings for corresponding positions of patches. There are various ways of doing that, I used the idea of adding positional embeddings to patch embeddings. I could have used fixed embeddings and concatenate them to patch embeddings but this would increase the computational cost, which is already high due to self-attention layers. I initialized the class token and positional embeddings as learnable parameters by using `nn.Parameter()`.

In the forward function, input is in image format, that is, its shape is `(batch size, number of channels, height, width)`. Then, it is divided into patches and the patches are projected to embeddings by `projection_layer`. Then, its output is put into sequential data format, which is `(batch size, sequence length, embedding size)`. And sequence length equals to number of patches. After that, class token embedding is added to the beginning of the sequence to if sequence length is 16, it becomes 17. Lastly, positional embeddings are expanded (copied) for each element in batch and they are added to the class token + patch embeddings. Its result is returned to be used in the encoder of the transformer. In the encoder part, I utilized Prof. Kak's code and I implemented the whole ViT in `VisionTransformer` class, which will be explained next.

3.2 VisionTransformer

I implemented the whole vision transformer architecture under my `VisionTransformer` class in `model.py` module. Its implementation is simple and given in Figure 4. ViT has three main com-



```python
class VisionTransformer(nn.Module):
    def __init__(self, args):
        super(VisionTransformer, self).__init__()
        # initial layer
        self.project_layer = ProjectionLayer(emb_size=args.emb_size, img_size=args.img_size)
        # encoder part of the transformer
        self.encoder = MasterEncoder(max_seq_length=args.max_seq_length, embedding_size=args.emb_size,
                                     how_many_basic_encoders=args.num_encoders, num_atten_heads=args.num_heads,
                                     myAttention=args.my_attention) # myAttention is added by me
        # classifier
        self.fc = nn.Linear(in_features=args.emb_size, out_features=args.class_num)

    def forward(self, x):
        out = self.project_layer(x)
        out = self.encoder(out)
        out = self.fc(out[:,0,:]) # take the class token
        return out
```

Figure 4: Implementation of `VisionTransformer`

ponenets which are projection layer, encoder, and fully-connected classifier layer. I explained the first one. About the second one, I used `MasterEncoder` because it consists of encoder blocks as in Figure 2. Number of such encoder blocks is hyper-parameter just like number of attention heads. As a final layer, I used fully-connected layer to predict the class of the image. Its input channel equals to the embedding size and output size is number of classes, which is 5. In `forward()` function, input in the shape of `(batch size, number of channels, height, width)`. It is passed to projection layer and its output is in the shape of `(batch size, sequence length, embedding size)` as explained above. This tensor consists of patch sequences for each image in the batch. They are passed to encoder. One key point here is that as opposed to encoder-decoder networks, encoder's job is to create **attention-enriched** embeddings rather than obtaining a low-dimensional representation of images. Thus, output shape is `(batch size, sequence length, embedding size)`. As explained previously, only class token embedding is used for classification so the first element of the sequence is chose for each image and they are passed to the fully-connected layer. These embeddings are mapped to 5-dimensional vectors for classification, whose entries represent the confidence for the corresponding class. One important point in Figure 4, `MasterEncoder` has `myAttention` argument. If it is `True`, it performs my implementation of multi-headed self-attention layers with `torch.einsum()`, else it runs Prof. Kak's implementation. Next, I will explain how I implemented that.

3.3 MySelfAttention (**BONUS**)

In this section, I present my `MySelfAttention` class, which is a substitute for `SelfAttention` class. Instead of using transpose, matrix multiplication and view operation in `SelfAttention` and `AttentionHead` classes, I use `torch.einsum()` function to perform all necessary operations once! Before explaining my implementation of self-attention, I want to explain how to perform matrix operations in `torch.einsum()` so that my implementation of `MySelfAttention` is clear to the reader. `torch.einsum()` uses Einstein's tensor notation to perform tensor multiplications in arbitrary axis. These tensor operations include, tensor multiplication, Hadamard product, transpose, summing columns or rows, trace, diagonal, etc. Some of the examples on how to perform these operatios via `torch.einsum()` are given below:

Matrix Multiplication

Definition of matrix multiplication is as follows let $A \in \mathbb{R}^{N_1 \times N_2}$ and $B \in \mathbb{R}^{N_2 \times N_3}$. Then,

$$a_{ik}b_{kj} := (AB)_{ij} = \sum_{k=1}^{N_2} a_{ik}b_{kj} \tag{3}$$

which is Einstein's notation. We do not write the summation term. `torch.einsum()` follows the same logic. To perform this multiplication, one needs to simply write `torch.einsum("ik,kj->ij", A, B)`. Common letters that do not appear at the output are the axis that we multiply the corresponding elements and sum them up.

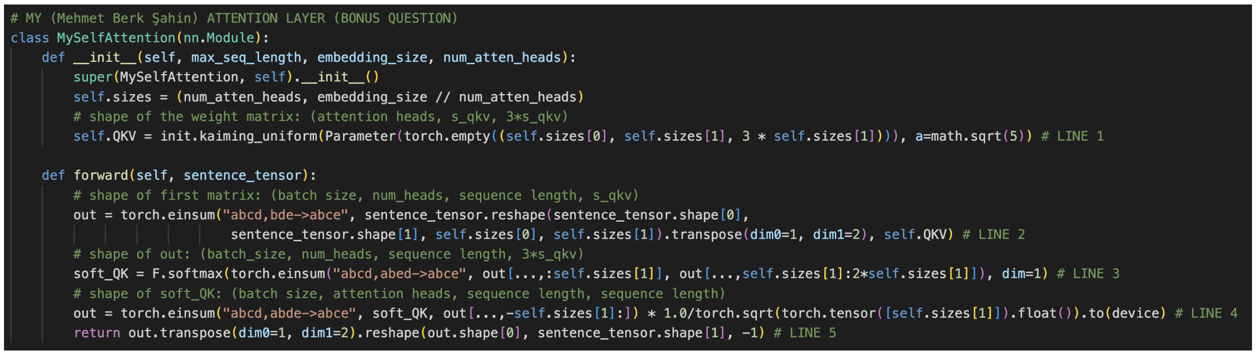
Transposed Matrix Multiplication

I also want to show the matrix multiplication that includes transpose of a matrix. Normally, we need to write two commands to transpose a matrix and put it into matrix multiplication. Rather than that, we can just write it as a single expression of `torch.einsum()`. Let $A, B \in \mathbb{R}^{N_1 \times N_2}$, then

$$a_{ik}b_{jk} := (AB^T)_{ij} = \sum_{k=1}^{N_2} a_{ik}b_{jk} \tag{4}$$

can be performed by using `torch.einsum("ik,jk->ij", A, B)`. I think these operations are enough for our purposes because although we have tensors, we just want to use 2 axes of the tensors for matrix multiplication and keep the others fixed as I will elaborate on further next.

I implemented `MySelfAttention` class in `ViT.py` module. It is given in Figure 5. As you can see, it consists of 5 lines. **Note that I only count the logic lines. To keep the code simple, I replaced `SelfAttention` class with my class. I could have put the same logic lines into the line where `SelfAttention` is called without any class implementation**. In **LINE 1**, instead of initializing W_Q, W_K, and W_V matrices seperately, I



```python
# MY (Mehmet Berk Şahin) ATTENTION LAYER (BONUS QUESTION)
class MySelfAttention(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super(MySelfAttention, self).__init__()
        self.sizes = (num_atten_heads, embedding_size // num_atten_heads)
        # shape of the weight matrix: (attention heads, s_qkv, 3*s_qkv)
        self.QKV = init.kaiming_uniform(Parameter(torch.empty((self.sizes[0], self.sizes[1], 3 * self.sizes[1]))), a=math.sqrt(5)) # LINE 1

    def forward(self, sentence_tensor):
        # shape of first matrix: (batch size, num_heads, sequence length, s_qkv)
        out = torch.einsum("abcd,bde->abce", sentence_tensor.reshape(sentence_tensor.shape[0],
                                    sentence_tensor.shape[1], self.sizes[0], self.sizes[1]).transpose(dim0=1, dim1=2), self.QKV) # LINE 2
        # shape of out: (batch_size, num_heads, sequence length, 3*s_qkv)
        soft_QK = F.softmax(torch.einsum("abcd,abed->abce", out[...,:self.sizes[1]], out[...,self.sizes[1]:2*self.sizes[1]]), dim=1) # LINE 3
        # shape of soft_QK: (batch size, attention heads, sequence length, sequence length)
        out = torch.einsum("abcd,abde->abce", soft_QK, out[...,-self.sizes[1]:]) * 1.0/torch.sqrt(torch.tensor([self.sizes[1]]).float()).to(device) # LINE 4
        return out.transpose(dim0=1, dim1=2).reshape(out.shape[0], sentence_tensor.shape[1], -1) # LINE 5
```

Figure 5: Implementation of `MySelfAttention`

initialized them in a single weight matrix. As they are learnable, I used `Parameter()`. In the forward method, input's shape is `(batch size, sequence length, embedding size)`. We need to perform calculations in equations (1) for each attention head seperately. To do so, I first reshaped the `sentence_tensor` to `(batch size, sequence length, number of attention heads, s_qkv)` where $s_{qkv} = \mathit{embedding\_size}/\mathit{attention\_heads}$. In other words, `s_qkv` is the embedding size at each attention head. Then, although I don't need to do that, as it is more intuitive in that way, I changed the order of axis 1 and 2. So, first input of `torch.einsum()` has the following shape `(batch size, number of attention heads, sequence length, s_qkv)`. As its fourth axis is `s_qkv` dimensional, `QKV` weight tensor is `(attention heads, s_qkv, 3*s_qkv)` because it consists of combination of W_Q, W_K, and W_V weights for each attention head. I entered this to `torch.einsum()` in **LINE 2** as inputs. Both tensors' last two axes are matrix multiplied and other axes remain fixed. So, its output shape is `(batch size, attention heads, sequence length, 3*s_qkv)`. If we partition this tensor in third axis direction, we get Q, K, and V tensors for each attention head, each patch in a image in the batch. Therefore, in **LINE 3**, I took the first two partitions, which are Q and K, and calculate the ratio between softmax and $\sqrt{M}$ in equation (2). To do so, I used the logic explained in equation (4). I performed transposed matrix multiplication, QK^T, between the last two axes of the inputs, other axes remained fixed. In **LINE 4**, I performed matrix multiplication between the last two axes of the output of LINE 3 and the third partition of LINE 2, which corresponds to exactly equation (2). Then, I obtained **attention-enriched** weights and returned them by correcting its shape as `(batch size, sequence length, embedding size)`. Note that input and output shapes of the self-attention layer are exactly the same as they should be. I will not explain the train and inference codes for the vision transformer because I copied them from my HW4 code.

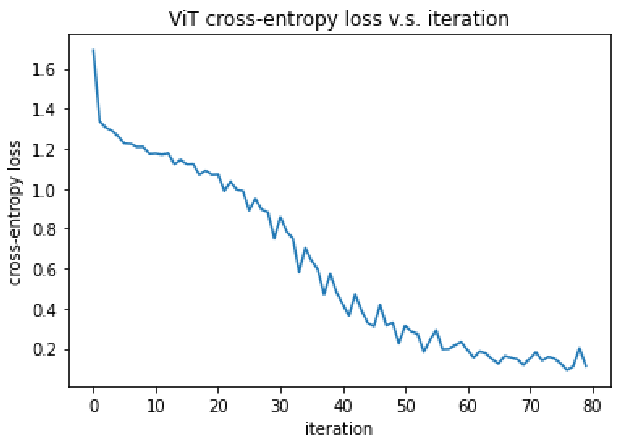
4 Experiments & Results

In this section, I present the hyperparameters of ViT training with my self-attention implementation and Prof.Kak's self-attention implementation. For a fair comparison, I used the same hyperparameters in both experiments. They are presented in Table 1.

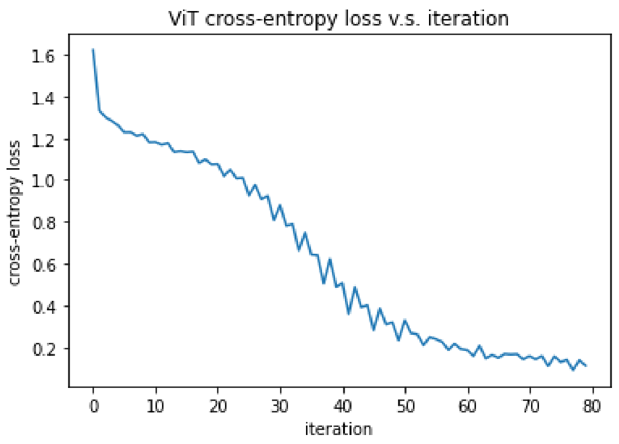
Hyperparameters	**Values**
Number of Attention Heads	5
Number of Encoders	3
Embedding Size	100
Image Size	64
Sequence Length	17
Learning Rate	2e-4
β_1	0.9
β_2	0.999
Batch Size	16
Patch Size	16
Epochs	20

Table 1: Hyperparameters for ViT training

I used Adam optimizer in the training so there are β_1 and β_2 values in the table. I experimented with various parameters to tune the performance of ViT on the test set and I will discuss them in the upcoming sections. Training of both models can be seen in Figure 6. First plot is the cross-entropy loss history of ViT with Prof. Kak's self attention layer and second plot is the cross-entropy loss history of ViT with `MySelfAttention` layer, which was implemented by me using `torch.einsum()`. Both plots look very similar to each other, which verifies that I implemented the self-attention layer correctly. This similarity also exist in the confusion matrices, which will be explained later. In Figure 6, it can be seen that both models get very close to 0 and they converged in 20 epochs. That implies ViT is powerful enough to learn the train dataset. To test its generalization performance,

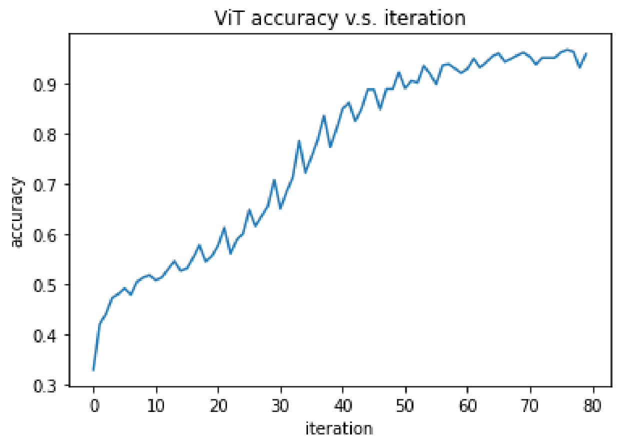


(a) cross-entropy loss v.s. iteration

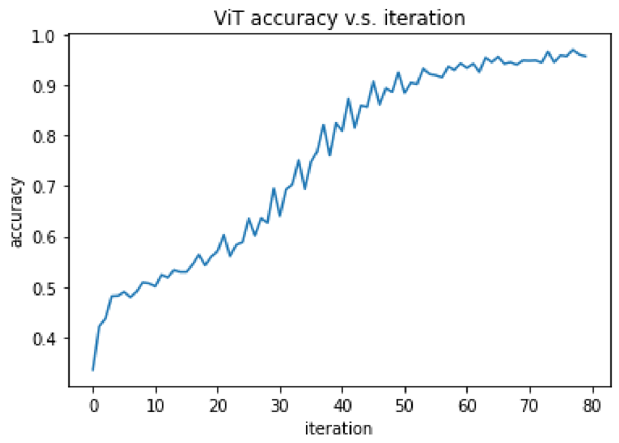


(b) cross-entropy loss v.s. iteration (MySelfAttention)

Figure 6: Comparison of train cross-entropy loss of ViT and ViT with MySelfAttention



(a) classification accuracy v.s. iteration

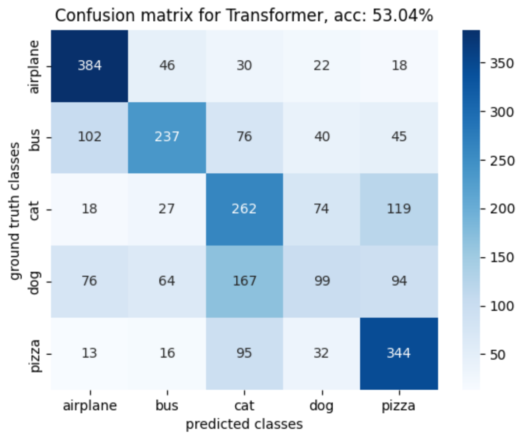


(b) classification accuracy v.s. iteration (MySelfAttention)

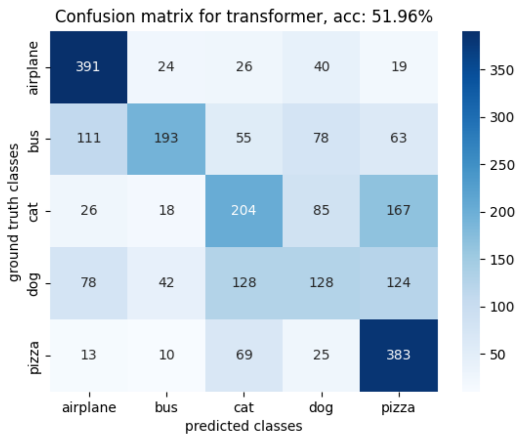
Figure 7: Comparison of train accuracy plots of ViT and ViT with MySelfAttention

one needs to check confusion matrices. But before that, let's discuss the accuracy plots, which can be found in Figure 7. Values at y-axis represents the mean classification accuracies over 100 iterations. They are in the range of 0 and 1, i.e., 0.9 represents 90% mean classification accuracy over all classes. In Figure 7, classification accuracy of both models approach to 1, which is an another indicator of model is powerful enough to learn the dataset.

To test model's generalization performance, I run forward pass on the test set, which consists of 1.5k images and each class has 500 samples. To see the results and make quantitative assessment, I plotted the confusion matrices, which can be seen in Figure 8. First confusion matrix consists of the predictions of ViT with Prof. Kak's self-attention implementation. And the second confusion matrix consists of the predictions of ViT with my implementation of self-attention. Since I used the same random seed for both experiments, results are very similar to each other. That clearly implies that my self-attention implementation with `torch.einsum` is correct. Furthermore, prediction patterns in both figures consistent with each other. For instance, classification of dog and cat is hard for both models whereas classification of airplane and pizza is easy. Although number



(a) ViT confusion matrix for test set with Prof. Kak's Self-Attention



(b) ViT confusion matrix for test set with `MySelfAttention`

Figure 8: Comparison of confusion matrices of Prof. Kak's self-attention and my implementation with `torch.einsum()` (*random seed*= 1)

of correct and wrong predictions at each cell are close to each other in both confusion matrices, there are 1% mean classification accuracy difference between them. The reason for this may be in `nn.Linear`, by default there are biases. So, each neural network for query, key and value vectors include biases. However, in my implementation, I did not initialize any bias so that may result in 1% difference on the classification accuracy. Lastly, I did the same initialization in the source code of `nn.Linear` to get a good performance because bad initialization degrades performance or slows down the training significantly.

Train and test results clearly show that ViT is powerful enough to learn the data but it overfits it because there is a significant performance decrease between in test classification accuracy compared to train classification accuracy. This is expected because transformers perform well in large datasets. As mentioned in the ViT paper that is given in the homework assignment, Vision Transformer overfits more than ResNet architectures with comparable computational cost on small datasets. In the paper, authors discussed a comparison between Resnet and ViT and they say ViT performs much worse than ResNet in 9M image dataset. ViT perform better on datasets with size +90M images. In this homework, our dataset size is 7.5k images, which is very very small. So, it is expected result to see ViT overfitting the data. In fact, ViT performs worse than my shallow CNNs in HW4. My best model at that homework gives 61.8% whereas ViT performs 53.04%. However, ViT's train accuracy beats all three models in HW4, which may be an indication of the fact that ViT is more powerful than those shallow CNNs. Furthermore, I believe 53% test accuracy is not bad because if we just randomly predicted a class, its accuracy would be approximately 20%. Considering that fact, 53% is good enough for the purpose of this homework. Lastly, I want to remind the reader that although I run the training for 20 epochs, I took the best model by looking at the validation results at each epoch, which is an another way of doing early stopping.

5 Lessons Learned

In this homework, I gained insights into the multi-headed self-attention mechanism and I understood the transformer architecture in a deeper level. I changed self-attention code and implemented the same logic with `torch.einsum()`. Thanks to `torch.einsum()`, I was able to do arbitrary tensor multiplications at once and get rid of redundant concetanations, transpose, reshaping, and slicing. I first understood how transformers are used for language tasks and then I learned how to implement that kind of architecture to image domain by reviewing the ViT paper and implementing its projection (initial layer). I combined that layer with master encoder and final classifier (fully-connected layer) to build my own ViT. I trained ViT on 7.5k image dataset and since model is too complex and powerful for such a small-sized dataset, it overfitted the data and exhibited poor generalization. This aligns with the discussion in the ViT paper.

6 How to run the code?

In the last section, I want to explain how to run the code for obtaining the same results. Firstly, we need to generate the train and test datasets in the current directory. To do so, running following command is enough:

```
python dataset.py --data_dir <coco directory>
```

where `data_dir` is the path of "coco" directory. It consists of 2014 train and 2014 test datasets with their annotations which should be downloaded from COCO website. If you encounter any error, you should check that folder names are consistent with the folder names in the code. After that, to run the trainings with the hyperparameters in 1, following commands are sufficient:

ViT Train

```
python hw9_MehmetBerkSahin.py
```

ViT Train with `MySelfAttention`

```
python hw9_MehmetBerkSahin.py --my_attention True
```

If you want to obtain the best results, you need to train the model for 7 epochs by using "`--epochs 7`". After training, cross-entropy and classification histories will be saved to the current folder in pickle format. You can load them and plot them. To test the models and display the confusion matrices, following commands are sufficient:

ViT Test

```
python hw9_MehmetBerkSahin.py --train False
```

ViT Test with `MySelfAttention`

```
python hw9_MehmetBerkSahin.py --my_attention True --train False
```

After these commands, mean test scores for cross-entropy and classification accuracies will be displayed on terminal and confusion matrices will be saved to the current directory in ".png" format. For the full details of the implementation, you can check the source code. I believe it is well-commented and clear. Note that my `MySelfAttention` layer for the bonus question was added to `ViTHelper.py` module.

7 Source Code

7.1 hw9_MehmetBerkSahin.py

```
 # my code
from dataset import MyCOCODataset
from model import VisionTransformer, train, inference
# torch
import torch
from torch.utils.data import DataLoader
# other
import argparse
import pickle
import random
import numpy as np



if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    # attention parameters
    parser.add_argument("--num_heads", default=5, type=int, help="number of attention heads")
    parser.add_argument("--num_encoders", default=3, type=int, help="number of encoders in transformer")
    parser.add_argument("--my_attention", default=False, type=eval, help="switch for using my attention")
    # model parameters
    parser.add_argument("--emb_size", default=100, type=int, help="number of embedding dimension")
    parser.add_argument("--img_size", default=64, type=int, help="heigh=width of an image")
    parser.add_argument("--max_seq_length", default=16+1, type=int, help="length of the patch sequences")
    parser.add_argument("--patch_size", default=16, type=int, help="patch size")
    parser.add_argument("--class_num", default=5, type=int, help="number of classes to classify")
    parser.add_argument("--model_name", default="transformer", type=str, help="name of the model")
    # train parameters
    parser.add_argument("--lr", default=2e-4, type=float, help="learning rate")
    parser.add_argument("--epochs", default=20, type=int, help="number of epochs")
    parser.add_argument("--beta1", default=0.9, type=float, help="beta1 parameter in Adam")
    parser.add_argument("--beta2", default=0.999, type=float, help="beta2 parameter in Adam")
    parser.add_argument("--device", default="cuda:0", type=str, help="select a device")
    parser.add_argument("--batch_size", default=16, type=int, help="batch size for SGD")
    # train & test
    parser.add_argument("--train", default=True, type=eval, help="run code for training or inference")
    args = parser.parse_args()

    # for reproducible results
    seed = 1
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False
    torch.autograd.set_detect_anomaly(True)

    # initialize the dataset
    train_data = MyCOCODataset(folder_name='train_data')
    val_data = MyCOCODataset(folder_name='val_data', train=False)
    train_loader = DataLoader(dataset=train_data, batch_size=16, num_workers=2)
    val_loader = DataLoader(dataset=val_data, batch_size=16, num_workers=2)
    # initialize model
    model = VisionTransformer(args)
    if args.train:
        results = train(model, train_loader, val_loader, args)
        pickle.dump(results, open("train_results.pkl", "wb"))
    else:
        inference(model, val_loader, args)
```

7.2 model.py

```
from ViTHelper import MasterEncoder
import argparse
from PIL import Image
from sklearn.metrics import confusion_matrix
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

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

class ProjectionLayer(nn.Module):
    def __init__(self, in_channels=3, patch_size=16, emb_size=100, img_size=64):
        super(ProjectionLayer, self).__init__()
        self.in_channels = in_channels # image channels
        self.patch_size = patch_size # height and width of patch which are equal
        self.emb_size = emb_size # emebedding size
        # projection layer = patching + linear layer
        self.projection_layer = nn.Conv2d(in_channels=self.in_channels, out_channels=self.emb_size,
                                          kernel_size=self.patch_size, stride=self.patch_size)

        self.class_token = nn.Parameter(torch.randn(1, 1, self.emb_size)) # for classification
        self.pos_emb = nn.Parameter(torch.randn(1, (img_size//patch_size)**2 + 1, self.emb_size)) # positional embedding

    def forward(self, x):
        out = self.projection_layer(x) # fully-connected layer with patching
        out = out.reshape(out.shape[0], out.shape[1], -1) # sequence format
        out = torch.transpose(out, dim0=1, dim1=2)
        # expand class tokens
        cls_tokens = self.class_token.expand(out.shape[0], 1, self.emb_size)
        out = torch.cat((cls_tokens, out), dim=1) # add class token to the beginning
        # expand positional embeddings
        pos_emb = self.pos_emb.expand(out.shape[0], out.shape[1], out.shape[2])
        # add positional embeddings to cls_token + image embeddings
        out += pos_emb # add the positional embeddings to patch embeddings
        return out

class VisionTransformer(nn.Module):
    def __init__(self, args):
        super(VisionTransformer, self).__init__()
        # initial layer
        self.project_layer = ProjectionLayer(emb_size=args.emb_size, img_size=args.img_size)
        # encoder part of the transformer
        self.encoder = MasterEncoder(max_seq_length=args.max_seq_length, embedding_size=args.emb_size,
                                     how_many_basic_encoders=args.num_encoders, num_atten_heads=args.num_heads,
                                     myAttention=args.my_attention) # myAttention is added by me
        # classifier
        self.fc = nn.Linear(in_features=args.emb_size, out_features=args.class_num)

    def forward(self, x):
        out = self.project_layer(x)
        out = self.encoder(out)
        out = self.fc(out[:,0,:]) # take the class token for class prediction
        return out

def train(model, train_loader, val_loader, args):
    device = args.device
    print(f"training is running on {device}...")
    # evaluation metrics for each 100 steps
    train_cross_hist, train_class_hist = [], []
    # set device, loss and optimization method16
    net = model.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=args.lr, betas=(args.beta1, args.beta2))
    for epoch in range(args.epochs):
        # running evaluations for each 100 steps
        cross_running_loss = 0.0
        class_running_acc = 0.0
        for i, data in enumerate(train_loader):
            # put model and data to gpu if available
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad() # set gradients 0
            outputs = net(inputs) # forward pass
            # calculate cross-entropy loss
            loss = criterion(outputs, labels)
            loss.backward() # backpropagation
            optimizer.step() # update weights
            cross_running_loss += loss.item()
            # classification accuracy
            preds = torch.argmax(outputs, dim=1) # choose the highest prob. class
            results = torch.eq(labels, preds) # classification accuracy (ca)
            # save the accuracy
            acc = torch.sum(results).item() / results.size(dim=0) # mean ca
            class_running_acc += acc

            if (i+1) % 100 == 0:
                print(f"TRAIN: [epoch {epoch + 1}, batch: {i + 1}] cross-entropy" +
                      f" loss: {round(cross_running_loss / 100, 3)}")
                print(f"TRAIN: [epoch {epoch + 1}, batch: {i + 1}] classification" +
                      f" accuracy: {round(class_running_acc / 100, 3)}")
                print("*"*40)
                # save the results for each 100 steps
                train_cross_hist.append(cross_running_loss / 100)
                train_class_hist.append(class_running_acc / 100)
                cross_running_loss = 0.0
                class_running_acc = 0.0

        #inference(net, val_loader, args)
    # save the last model
    print("training is completed!")
    torch.save(net.state_dict(), args.model_name)
    print("model is saved!")
    # return the loss and accuracy
    return {"cross_entropy" : train_cross_hist,
            "class_acc" : train_class_hist}

def inference(model, val_loader, args):
    model.load_state_dict(torch.load(args.model_name)) # load the saved model
    device = args.device
    print(f"inference is running on {device}...")
    y_preds, y_trues = [], []
    # set device, loss and optimization method16
    net = model.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    net.eval() # inference mode
    class_acc, cross_loss = 0, 0
    for i, data in enumerate(val_loader):
        # put model and data to gpu if available
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = net(inputs) # forward pass
        # calculate cross-entropy loss
        loss = criterion(outputs, labels)
        cross_loss += loss.item()
        # classification accuracy
        preds = torch.argmax(outputs, dim=1) # choose the highest prob. class
        results = torch.eq(labels, preds) # classification accuracy (ca)
        y_trues.extend(labels.cpu())
        y_preds.extend(preds.cpu())
        # save the accuracy
        acc = torch.sum(results).item() / results.size(dim=0) # mean ca
        class_acc += acc

    class_acc /= len(val_loader)
    cross_loss /= len(val_loader)
    print("inference is completed!")
    print("*"*45)
    print(f"validation cross-entropy loss: {cross_loss:.4f}")
    print(f"validation classification accuracy: {class_acc:.4f}")
    # plot confusion matrix
    classes = ["airplane", "bus", "cat", "dog", "pizza"]
    cm = confusion_matrix(y_trues, y_preds)
    df_cm = pd.DataFrame(cm, index = [i for i in classes], columns = [i for i in classes])
    acc = (np.trace(cm) / np.sum(cm)) * 100
    sns.heatmap(df_cm, annot=True, fmt='g', cmap="Blues")
    plt.title(f"Confusion matrix for {args.model_name}, acc: {acc:.2f}%")
    plt.xlabel("predicted classes")
    plt.ylabel("ground truth classes")
    plt.savefig(f'{args.model_name}_confusion_matrix.png')



# test code
if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    # attention parameters
    parser.add_argument("--num_heads", default=10, type=int, help="number of attention heads")
    parser.add_argument("--num_encoders", default=5, type=int, help="number of encoders in transformer")
    # model parameters
    parser.add_argument("--emb_size", default=100, type=int, help="number of embedding dimension")
    parser.add_argument("--img_size", default=64, type=int, help="heigh=width of an image")
    parser.add_argument("--max_seq_length", default=16+1, type=int, help="length of the patch sequences")
    parser.add_argument("--patch_size", default=16, type=int, help="patch size")
    parser.add_argument("--class_num", default=5, type=int, help="number of classes to classify")


    args = parser.parse_args()
    # test projection layer
    model = ProjectionLayer()
    # (B, C, H, W) = (4, 3, 64, 64)
    x = torch.randn(4, 3, 64, 64)
    y = model(x)
    print("ProjectionLayer works successfully!")
    print("output shape:", y.shape)

    # test transformer encoder
    model = VisionTransformer(args)
    x = torch.randn(4, 3, 64, 64)
    y = model(x)
    print("VisionTransformer works successfully!")
    print("output shape:", y.shape)
```

7.3 dataset.py

```
import os
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
import random
from PIL import Image

class MyCOCODataset(Dataset):
    def __init__(self, folder_name, aug=False, train=True):
        super().__init__()
        self.folder_name = folder_name # dataset path
        self.files = os.listdir(folder_name) # file names in list
        random.shuffle(self.files) # shuffle the list
        # Later one can add augmentations to this pipeline easily
        if train:
            self.transforms = tvt.Compose([
                #tvt.RandomHorizontalFlip(p=0.5),
                #tvt.RandomVerticalFlip(p=0.5),
                #tvt.RandomRotation(degrees=15),
                #tvt.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
                tvt.ToTensor(),
            ])
        else:
            self.transforms = tvt.Compose([
                tvt.ToTensor()
            ])
        # class names
        self.classes = ["airplane", "bus", "cat", "dog", "pizza"]

    def __len__(self):
        return len(self.files)

    def __getitem__(self, item):
        file_name = self.files[item]
        img = Image.open(os.path.join(self.folder_name, file_name)) # load image
        img = self.transforms(img) # convert it to tensor with normalization
        class_name = file_name.split("_")[0] # get the index of label
        label = self.classes.index(class_name) # get label
        return img, label

def plot_images(train_data):
    # class names
    classes = ["airplane", "bus", "cat", "dog", "pizza"]
    # plot random 5 images for each class
    fig, axs = plt.subplots(5, 5)
    fig.tight_layout()
    for i, cat in enumerate(classes):
        files = train_data[cat]
        chosen = random.sample(files, 5) # sample 5 random images
        for j, file_name in enumerate(chosen):
            img = Image.open(os.path.join('train_data', file_name)) # load image
            axs[i, j].imshow(img) # plot image
            axs[i, j].axis('off')
            axs[i, j].set_title(f"{cat} {j + 1}", size=7)
    plt.show()

def set_datasets(data_dir="/Users/berksahin/Desktop"):
    # create folders
    if not os.path.exists("train_data"):
        os.mkdir("train_data")
    if not os.path.exists("val_data"):
        os.mkdir("val_data")

    annFile = os.path.join(data_dir, "coco/annotations/instances_train2014.json")
    coco = COCO(annFile)
    # classes for the problem in hw4
    classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
    catIds = coco.getCatIds(catNms=classes)  # get class indices
    # keys: class names, values: list of files names
    train_data = dict(zip(classes, [[] for i in range(len(classes))]))
    val_data = dict(zip(classes, [[] for i in range(len(classes))]))
    print("dataset generation has started...")
    for i, idx in enumerate(catIds):
        imgIds = coco.getImgIds(catIds=idx)
        imgIds = np.random.choice(imgIds, size=2000, replace=False)

        for counter, img_idx in enumerate(imgIds):
            # get the file name of the image
            file_name = coco.loadImgs(int(img_idx))[0]['file_name']
            img_path = os.path.join(data_dir, f"coco/images/{file_name}")
            # open the image as PIL image in RGB format
            img = Image.open(img_path).convert("RGB")
            img = img.resize((64, 64)) # resize

            class_name = classes[i]
            save_name = class_name + "_" + file_name
            if counter < 1500:
                save_dir = "train_data"
                train_data[class_name].append(save_name)
            else:
                save_dir = "val_data"
                val_data[class_name].append(save_name)
            # save the image in new dataset folder
            img.save(os.path.join(save_dir, save_name))

    print("train and validation datasets are ready!")

    return {"train_data" : train_data, "val_data" : val_data}

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("--data_dir", default="/Users/berksahin/Desktop", help="Current directory path of the coco folder")
    args = parser.parse_args()
    np.random.seed(5)
    random.seed(5)
    # create the dataset
    data = set_datasets(args.data_dir)
    plot_images(data["train_data"])

    # construct datasets
    train_dataset = MyCOCODataset(folder_name='train_data')
    val_dataset = MyCOCODataset(folder_name='val_data')
```

7.4 ViTHelper.py

```
##  This code is from the Transformers co-class of DLStudio:

##            https://engineering.purdue.edu/kak/distDLS/

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.parameter import Parameter

from torch.nn import init
import math

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

class MasterEncoder(nn.Module):
    def __init__(self, max_seq_length, embedding_size, how_many_basic_encoders, num_atten_heads, myAttention=False):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.basic_encoder_arr = nn.ModuleList([BasicEncoder(
            max_seq_length, embedding_size, num_atten_heads, myAttention) for _ in range(how_many_basic_encoders)])  # (A)

    def forward(self, sentence_tensor):
        out_tensor = sentence_tensor
        for i in range(len(self.basic_encoder_arr)):  # (B)
            out_tensor = self.basic_encoder_arr[i](out_tensor)
        return out_tensor


class BasicEncoder(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads, myAttention=False):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.embedding_size = embedding_size
        self.qkv_size = self.embedding_size // num_atten_heads
        self.num_atten_heads = num_atten_heads
        if myAttention:
            self.self_attention_layer = MySelfAttention(
                max_seq_length, embedding_size, num_atten_heads)  # (A)
        else:
            self.self_attention_layer = SelfAttention(
                max_seq_length, embedding_size, num_atten_heads)  # (A)
        self.norm1 = nn.LayerNorm(self.embedding_size)  # (C)
        self.W1 = nn.Linear(self.max_seq_length * self.embedding_size,
                            self.max_seq_length * 2 * self.embedding_size)
        self.W2 = nn.Linear(self.max_seq_length * 2 * self.embedding_size,
                            self.max_seq_length * self.embedding_size)
        self.norm2 = nn.LayerNorm(self.embedding_size)  # (E)

    def forward(self, sentence_tensor):
        input_for_self_atten = sentence_tensor.float()
        normed_input_self_atten = self.norm1(input_for_self_atten)
        output_self_atten = self.self_attention_layer(
            normed_input_self_atten).to(device)  # (F)
        input_for_FFN = output_self_atten + input_for_self_atten
        normed_input_FFN = self.norm2(input_for_FFN)  # (I)
        basic_encoder_out = nn.ReLU()(
            self.W1(normed_input_FFN.view(sentence_tensor.shape[0], -1)))  # (K)
        basic_encoder_out = self.W2(basic_encoder_out)  # (L)
        basic_encoder_out = basic_encoder_out.view(
            sentence_tensor.shape[0], self.max_seq_length, self.embedding_size)
        basic_encoder_out = basic_encoder_out + input_for_FFN
        return basic_encoder_out

####################################  Self Attention Code TransformerPreLN ##########################################

class SelfAttention(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.attention_heads_arr = nn.ModuleList([AttentionHead(self.max_seq_length,
                                                                self.qkv_size) for _ in range(num_atten_heads)])  # (A)

    def forward(self, sentence_tensor):  # (B)
        concat_out_from_atten_heads = torch.zeros(sentence_tensor.shape[0], self.max_seq_length,
                                                  self.num_atten_heads * self.qkv_size).float()
        for i in range(self.num_atten_heads):  # (C)
```

```python
            sentence_tensor_portion = sentence_tensor[:,
                                                      :, i * self.qkv_size: (i+1) * self.qkv_size]
            concat_out_from_atten_heads[:, :, i * self.qkv_size: (i+1) * self.qkv_size] =          \
                self.attention_heads_arr[i](sentence_tensor_portion)  # (D)
        return concat_out_from_atten_heads


class AttentionHead(nn.Module):
    def __init__(self, max_seq_length, qkv_size):
        super().__init__()
        self.qkv_size = qkv_size
        self.max_seq_length = max_seq_length
        self.WQ = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (B)
        self.WK = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (C)
        self.WV = nn.Linear(max_seq_length * self.qkv_size,
                            max_seq_length * self.qkv_size)  # (D)
        self.softmax = nn.Softmax(dim=1)  # (E)

    def forward(self, sentence_portion):  # (F)
        Q = self.WQ(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (G)
        K = self.WK(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (H)
        V = self.WV(sentence_portion.reshape(
            sentence_portion.shape[0], -1).float()).to(device)  # (I)
        Q = Q.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (J)
        K = K.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (K)
        V = V.view(sentence_portion.shape[0],
                   self.max_seq_length, self.qkv_size)  # (L)
        A = K.transpose(2, 1)  # (M)
        print("KT:", A.shape)
        QK_dot_prod = Q @ A  # (N)
        print("QK_dot:", QK_dot_prod.shape)
        rowwise_softmax_normalizations = self.softmax(QK_dot_prod)  # (O)
        print("After softmax:", rowwise_softmax_normalizations.shape)
        Z = rowwise_softmax_normalizations @ V
        coeff = 1.0/torch.sqrt(torch.tensor([self.qkv_size]).float()).to(device)  # (S)
        Z = coeff * Z  # (T)
        print("final:", Z.shape)
        return Z

# MY (Mehmet Berk Şahin) ATTENTION LAYER (BONUS QUESTION)
class MySelfAttention(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super(MySelfAttention, self).__init__()
        self.sizes = (num_atten_heads, embedding_size // num_atten_heads)
        # shape of the weight matrix: (attention heads, s_qkv, 3*s_qkv)
        self.QKV = init.kaiming_uniform(Parameter(torch.empty((self.sizes[0], self.sizes[1], 3 * self.sizes[1]))), a=math.sqrt(5)) # LINE 1

    def forward(self, sentence_tensor):
        # shape of first matrix: (batch size, num_heads, sequence length, s_qkv)
        out = torch.einsum("abcd,bde->abce", sentence_tensor.reshape(sentence_tensor.shape[0],
                            sentence_tensor.shape[1], self.sizes[0], self.sizes[1]).transpose(dim0=1, dim1=2), self.QKV) # LINE 2
        # shape of out: (batch_size, num_heads, sequence length, 3*s_qkv)
        soft_QK = F.softmax(torch.einsum("abcd,abed->abce", out[...,:self.sizes[1]], out[...,self.sizes[1]:2*self.sizes[1]]), dim=1) # LINE 3
        # shape of soft_QK: (batch size, attention heads, sequence length, sequence length)
        out = torch.einsum("abcd,abde->abce", soft_QK, out[...,-self.sizes[1]:]) * 1.0/torch.sqrt(torch.tensor([self.sizes[1]]).float()).to(device) # LINE 4
        return out.transpose(dim0=1, dim1=2).reshape(out.shape[0], sentence_tensor.shape[1], -1) # LINE 5
```