final-ml4sci-task-3-1

March 8, 2023

1 ML4SCI TASK 3.1

2 ELECTRON VS PHOTONS

3 ViT- Vision Transformer….

- (**model inspired from ”An image is worth 16*16 word”**),
- (**“Attention is all you need”**)

Since the data set of this and the Task1 is the same I have used the same starting code to import and create data loaders

```
[1]: # importing basic modules
     import pandas as pd
     import numpy as np
```

To read the file of type hdf5 we need to import h5py

```
[2]: import h5py
```

This code below separates the X and y for the training from both the directories inside the root/input directory

```
[3]: vishak = h5py.File('/kaggle/input/electron-vs-photons-ml4sci/
      ↪SingleElectronPt50_IMGCROPS_n249k_RHv1.hdf5')
```

```
[4]: vishak.keys()
```

```
[4]: <KeysViewHDF5 ['X', 'y']>
```

WE SEE THAT IT CONTAINS X AND y INSINDE IT SO THAT WE CAN SPLIT IT LATER ON

```
[5]: vishak['/X']  # code to open the X values inside the File

     #Same way can be used for y
```

```
[5]: <HDF5 dataset "X": shape (249000, 32, 32, 2), type "<f4">
```

****LETS CREATE A FUNCTION THAT CONCATENATES THE IMAGES OF TWO DIRECTORIES ie ELECTRONS AND PHOTONS****

[6]:
```
img_rows, img_cols, nb_channels=32, 32, 2 #   channels=2(hit energy and time) ␣
 ↪image dimension is 32*32
input_dir = '/kaggle/input/electron-vs-photons-ml4sci'

decays=['SinglePhotonPt50_IMGCROPS_n249k_RHv1','SingleElectronPt50_IMGCROPS_n249k_RHv1']

def load_data(decays):
    global input_dir

    dsets = [h5py.File(input_dir+'/'+decay+'.hdf5') for decay in decays]

    X = np.concatenate([dset['/X'][0:300000] for dset in dsets])
    y = np.concatenate([dset['/y'][0:300000] for dset in dsets])
    assert len(X) == len(y)

    return X, y
```

[7]:
```
X,y=load_data(decays)
```

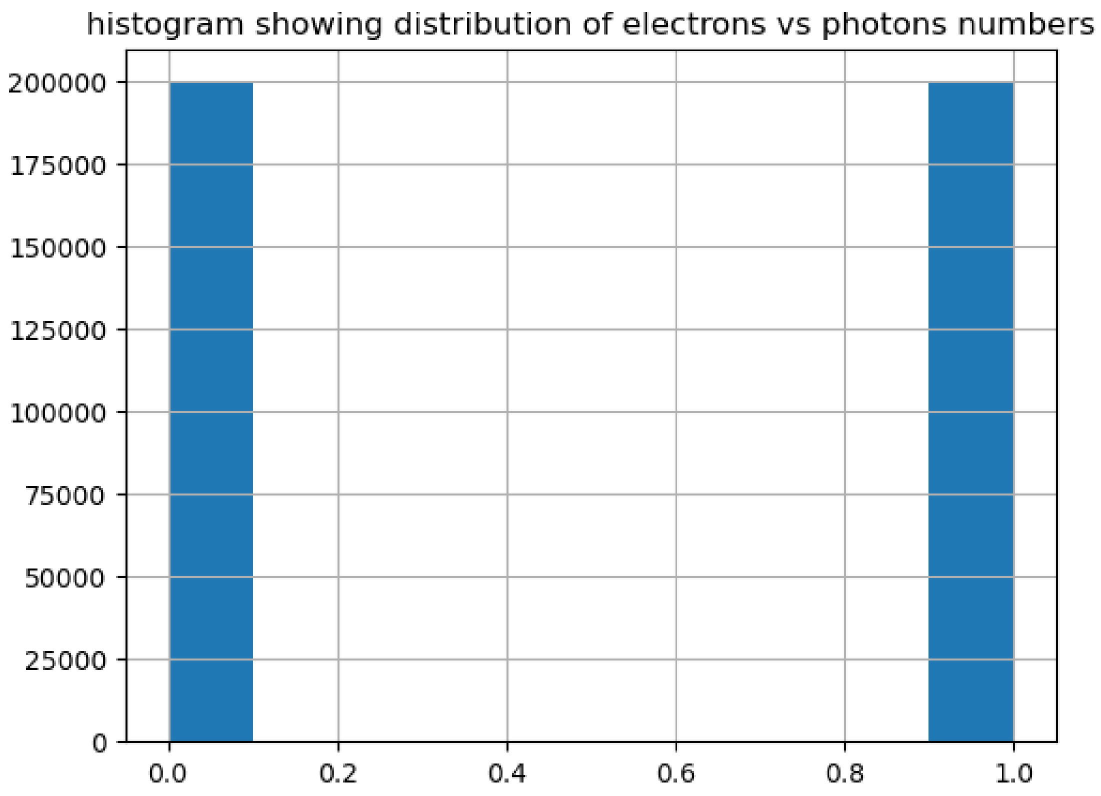
[8]:
```
y=pd.DataFrame(y)
```

[9]:
```
y.value_counts(normalize=False)
```

[9]:
```
0.0    200000
1.0    200000
dtype: int64
```

[10]:
```
import matplotlib.pyplot as plt
```

[11]:
```
y.hist()
plt.title("histogram showing distribution of electrons vs photons numbers");
```



WE SEE THAT BOTH THE ELECTRONS AND PHOTONS HAVE EQUAL NUMBER OF IMAGES

LETS TRY PRINTING THE IMAGE

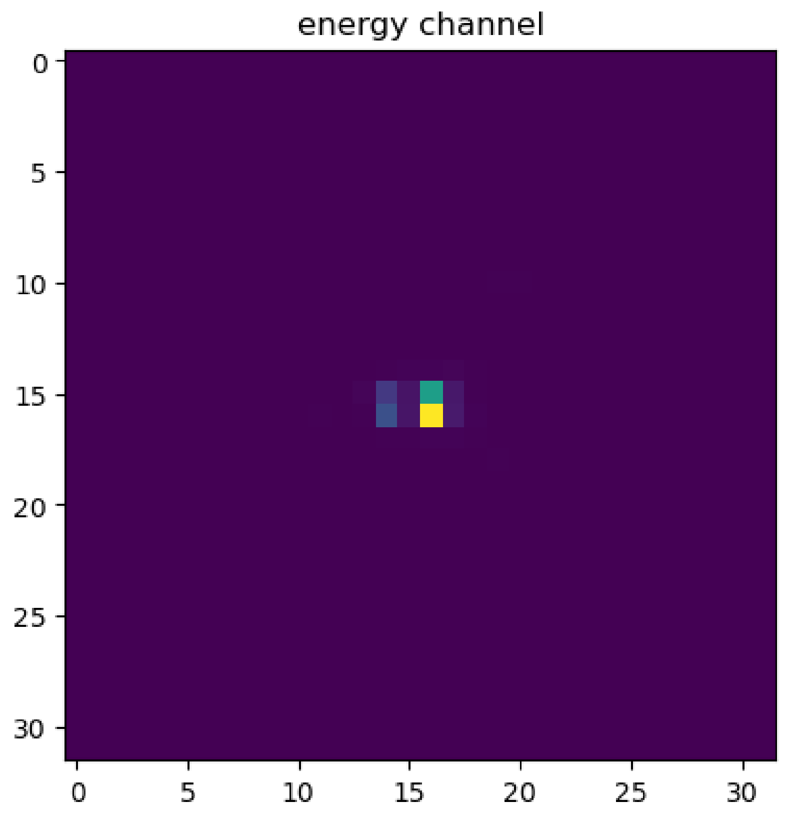
[12]:
```
X.shape
```

[12]:
```
(400000, 32, 32, 2)
```

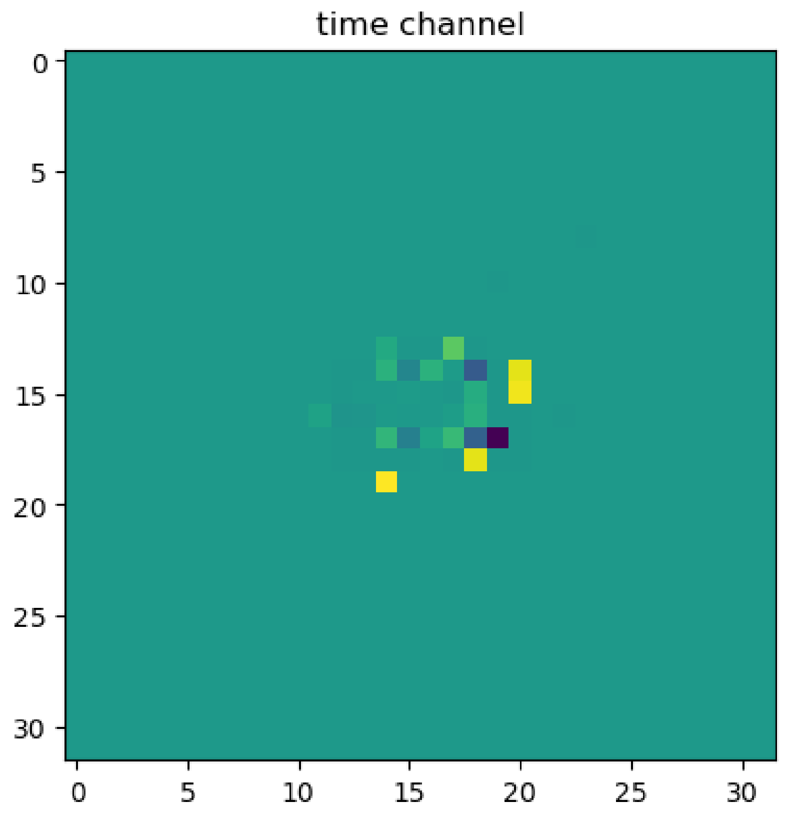
[13]:
```
X[0].shape
```

[13]:
```
(32, 32, 2)
```

[14]:
```
plt.imshow(X[0,:,:,0])    #shows us the first channel out of the two channels
plt.title("energy channel");
```



```
[15]: plt.imshow(X[0,:,:,1])    #shows us the second channel out of the two channels
      plt.title("time channel");
```



REFERENCE

- DATASETS TAKEN FROM ML4SCI – https://cernbox.cern.ch/index.php/s/AtBT8y4MiQYFcgc (photons) https://cernbox.cern.ch/index.php/s/FbXw3V4XNyYB3oA (electrons)

4 PIPELINE OF THE PROJECT

- LETS SPLIT THE DATA INTO TRAIN AND TEST
- LETS CREATE THE DATASET AND DATA LOADER FIRST
- CREATE THE ARCHITECTURE OF THE MODEL

[16]:
```
X=X/(X.max())
```

[17]:
```
from sklearn.model_selection import train_test_split
```

[18]:
```
X=X.transpose(0,3,1,2)
```

splitting the dataset into train and test sets

[19]:
```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.
↪1,shuffle=True,random_state=17)
```

[20]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset,DataLoader
import torch.nn.functional as F
```

Setting the device to cuda if its available

[21]:
```
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[22]:
```
device
```

[22]:
```
device(type='cuda')
```

[23]:
```
y_train=np.array(y_train)
y_test=np.array(y_test)
```

pushing the training data into the device so that it can be processed quickly

[24]:
```
X_train=(torch.tensor(X_train,dtype=torch.float32)).to(device)
X_test=torch.tensor(X_test,dtype=torch.float32)
y_train=torch.tensor(y_train,dtype=torch.float32).to(device)
y_test=torch.tensor(y_test,dtype=torch.float32)
```

This is the dataset which returns the image pixel values and the target

[25]:
```
class train_dset(Dataset):

    def __init__(self,X,y):

        self.X=X
        self.y=y

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

    def __getitem__(self,idx):

        return self.X[idx],self.y[idx]
```

[26]:
```
traindset= train_dset(X_train,y_train)
```

[27]:
```
train_loader= DataLoader(traindset,batch_size=32,shuffle=True)
```

[28]:
```
next(iter(train_loader))[0].shape
```

[28]:
```
torch.Size([32, 2, 32, 32])
```

[29]:
```
import torch
import torch.nn as nn
```

5 1. PatchEmbed

- **input:** Tensor of the shape **(n,in_chan,widht,height)**
- This class splits the image pixel and then embed them into the embedding dimension
- Basically the approach here is that the image which had 2 channels that is energy channel and time channel this is passed through a conv2d layer of embed dim number of filters. This returns a matrix that has embed_dim of channels
- Since the stride and the filter size of the conv3d was kept to be the same as the patch size, the formula

$ finaldim = ((initialdim+2*padding-filtersize)/stride)+1$ —–A

- when this formula is applied, the initaildim is taken such that patch size fits it. Therefore it will be some thing like $sqrt(numberOfPatches) * patchDim = imageDim$
- Applying the above equation on equation A we get that final dim = $sqrt(numberFilters)$
- So finally after passing through the conv2d layer the image matrix which was $(n, 2, 32, 32)$ is now changed into **$(n$,$embedDim$,$sqrt(numberFilters)$,$sqrt(numberFilters)$)**
- **output:** At last i have flattened the tensor using .flatten(2) which merges the dim 2 and 3 and then .transpose(1,2) which finally results in the dimesnsion **(n,n_patches,embed_dim)**

[30]:
```
class PatchEmbed(nn.Module):
    """Split image into patches and then embed them.
    Parameters
    ----------
    img_size : int
        Size of the image (it is a square).
    patch_size : int
        Size of the patch (it is a square).
    in_chans : int
        Number of input channels.
    embed_dim : int
        The emmbedding dimension.
    Attributes
    ----------
    n_patches : int
        Number of patches inside of our image.
    proj : nn.Conv2d
        Convolutional layer that does both the splitting into patches
        and their embedding.
    """
    def __init__(self, img_size, patch_size, in_chans=2, embed_dim=768):
        super().__init__()
        self.img_size = img_size
        self.patch_size = patch_size
        self.n_patches = (img_size // patch_size) ** 2


        self.proj = nn.Conv2d(
                in_chans,
                embed_dim,
                kernel_size=patch_size,
                stride=patch_size,
        )

    def forward(self, x):
        """Run forward pass.
        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, in_chans, img_size, img_size)`.
        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches, embed_dim)`.
        """
        x = self.proj(
                x
            )  # (n_samples, embed_dim, n_patches ** 0.5, n_patches ** 0.5)
        x = x.flatten(2)  # (n_samples, embed_dim, n_patches)
        x = x.transpose(1, 2)  # (n_samples, n_patches, embed_dim)

        return x
```

6 2. Attention

- **input:** Tensor of dimension **(n,n__patches+1,embed__dim)**
- NOTE: Here the dim1 is n__patches +1 becuse the first patch always is the [cls] token which is used to predict the class at the end
- What this class does is that it finds the relation between all the patches/tokens using the concept of attention
- firstly it is passed through a linear layer that outputs 3*dim each corresponding to the qkv that is query, key and value
- after that i split the 3*dim into a (3,n__heads,head__dim)
- Then we matrix multiply the query with key and use the scaling factor as seen in the reseach paper

- after this we use the softmax to the last dimension so that we get the probablity distribution after mutilplying the the value tensor
- that tensor is called as the weighted_avg ,this is then projected by the projection layer and finally we get the output
- **output:** Tensor of the shape (n,n_patches+1,embed_dim)

[31]:
```
class Attention(nn.Module):
    """Attention mechanism.
    Parameters
    ----------
    dim : int
        The input and out dimension of per token features.
    n_heads : int
        Number of attention heads.
    qkv_bias : bool
        If True then we include bias to the query, key and value projections.
    attn_p : float
        Dropout probability applied to the query, key and value tensors.
    proj_p : float
        Dropout probability applied to the output tensor.
    Attributes
    ----------
    scale : float
        Normalizing consant for the dot product.
    qkv : nn.Linear
        Linear projection for the query, key and value.
    proj : nn.Linear
        Linear mapping that takes in the concatenated output of all attention
        heads and maps it into a new space.
    attn_drop, proj_drop : nn.Dropout
        Dropout layers.
    """
    def __init__(self, dim, n_heads=12, qkv_bias=True, attn_p=0., proj_p=0.):
        super().__init__()
        self.n_heads = n_heads
        self.dim = dim
        self.head_dim = dim // n_heads
        self.scale = self.head_dim ** -0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_p)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_p)

    def forward(self, x):
        """Run forward pass.
        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, n_patches + 1, dim)`.
        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches + 1, dim)`.
        """
        n_samples, n_tokens, dim = x.shape

        if dim != self.dim:
            raise ValueError

        qkv = self.qkv(x)  # (n_samples, n_patches + 1, 3 * dim)
        qkv = qkv.reshape(
                n_samples, n_tokens, 3, self.n_heads, self.head_dim
        )  # (n_smaples, n_patches + 1, 3, n_heads, head_dim)
        qkv = qkv.permute(
                2, 0, 3, 1, 4
        )  # (3, n_samples, n_heads, n_patches + 1, head_dim)

        q, k, v = qkv[0], qkv[1], qkv[2]
        k_t = k.transpose(-2, -1)  # (n_samples, n_heads, head_dim, n_patches +␣
↪1)
        dp = (
           q @ k_t
        ) * self.scale # (n_samples, n_heads, n_patches + 1, n_patches + 1)
        attn = dp.softmax(dim=-1)  # (n_samples, n_heads, n_patches + 1,␣
↪n_patches + 1)
        attn = self.attn_drop(attn)

        weighted_avg = attn @ v  # (n_samples, n_heads, n_patches +1, head_dim)
        weighted_avg = weighted_avg.transpose(
                1, 2
        )  # (n_samples, n_patches + 1, n_heads, head_dim)
        weighted_avg = weighted_avg.flatten(2)  # (n_samples, n_patches + 1,␣
↪dim)

        x = self.proj(weighted_avg)  # (n_samples, n_patches + 1, dim)
        x = self.proj_drop(x)  # (n_samples, n_patches + 1, dim)

        return x
```

7 3. Multilayer Perceptron

- **input:** Tensor of the shape (n_samples, n_patches + 1, in_features)‘
- the forward function of this class contains 2 fully connected layers in which there is one hidden layer and the neurons in the hidden layer is given by the mlp_ratio
- the second fc layer again converts the tensor back to the input shape and it has 2 dropouts in between inorder to prevent overfitting
- **output:** Tensor of the shape (n_samples, n_patches +1, out_features)

[32]:
```python
class MLP(nn.Module):
    """Multilayer perceptron.
    Parameters
    ----------
    in_features : int
        Number of input features.
    hidden_features : int
        Number of nodes in the hidden layer.
    out_features : int
        Number of output features.
    p : float
        Dropout probability.
    Attributes
    ----------
    fc : nn.Linear
        The First linear layer.
    act : nn.GELU
        GELU activation function.
    fc2 : nn.Linear
        The second linear layer.
    drop : nn.Dropout
        Dropout layer.
    """
    def __init__(self, in_features, hidden_features, out_features, p=0.):
        super().__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = nn.GELU()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(p)

    def forward(self, x):
        """Run forward pass.
        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, n_patches + 1, in_features)`.
        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches +1, out_features)`
        """
        x = self.fc1(
                x
        ) # (n_samples, n_patches + 1, hidden_features)
        x = self.act(x)  # (n_samples, n_patches + 1, hidden_features)
        x = self.drop(x)  # (n_samples, n_patches + 1, hidden_features)
        x = self.fc2(x)  # (n_samples, n_patches + 1, out_features)
        x = self.drop(x)  # (n_samples, n_patches + 1, out_features)

        return x
```

8 4. Block

- **input:** Tensor of the shape (n_samples, n_patches + 1, in_features)‘
- This class contains the residual block that adds itself to the output of the Attention and the MLP classes forward functions.
- First the input is layer normalized and then fed into the Attention and the MLP classes
- **output:** Tensor of the shape (n_samples, n_patches + 1, in_features)‘

[32]:
```
class Block(nn.Module):
    """Transformer block.
    Parameters
    ----------
    dim : int
        Embeddinig dimension.
    n_heads : int
        Number of attention heads.
    mlp_ratio : float
        Determines the hidden dimension size of the `MLP` module with respect
        to `dim`.
    qkv_bias : bool
        If True then we include bias to the query, key and value projections.
    p, attn_p : float
        Dropout probability.
    Attributes
    ----------
    norm1, norm2 : LayerNorm
        Layer normalization.
    attn : Attention
        Attention module.
    mlp : MLP
        MLP module.
    """
    def __init__(self, dim, n_heads, mlp_ratio=4.0, qkv_bias=True, p=0.,␣
↪attn_p=0.):
        super().__init__()
        self.norm1 = nn.LayerNorm(dim, eps=1e-6)
        self.attn = Attention(
                dim,
                n_heads=n_heads,
                qkv_bias=qkv_bias,
                attn_p=attn_p,
                proj_p=p
        )
        self.norm2 = nn.LayerNorm(dim, eps=1e-6)
        hidden_features = int(dim * mlp_ratio)
        self.mlp = MLP(
                in_features=dim,
                hidden_features=hidden_features,
                out_features=dim,
        )

    def forward(self, x):
        """Run forward pass.
        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, n_patches + 1, dim)`.
        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches + 1, dim)`.
        """
        x = x + self.attn(self.norm1(x))
        x = x + self.mlp(self.norm2(x))

        return x
```

9 5. Vision Transformer

- **input:** Tensor of the shape **(n_samples, in_chans, img_size, img_size)**
- This is the final class that contains all the classes which were defined until now.
- Initially I randomly initialised the cls token to the same size of that of the embed_dim **(1,1,embed_dim)** , and also intialised the position embedding randomly to the shape **(n,n_patches+1,embed_dim)**. Here the +1 is there for the cls token
- After this I expanded the the cls token into the number of trianing examples along the dimension 0

- Then concatenate this into the output that came after passing through the patch_embeds forward function
- Did the residual adding and added the positional embedding and here the point to be noted is that python broadcasts the positional embedding tensor to the required dimension along the dim0
- Now that the patches/tokens are ready we can proceed to implement the blocks which contains the attnetion and the MLP classes
- In between we have normalisation so that the mean of that dimension becomes 0 and the std deviation becomes 1
- This is observed that it imporoves the trainig
- Now the starting token among the number of patches is taken because that is the cls token and that is used to classify furher.
- the cls token is passed through the linear layer that converts it into the number of classes here i have kept it as 1 and then appplied sigmoid so that i get the probablity of that training example belonging to the class 1
- **output:** Tensor of the shape **(n_samples, 1)**‘

[33]:
```python
class VisionTransformer(nn.Module):
    """Simplified implementation of the Vision transformer.
    Parameters
    ----------
    img_size : int
        Both height and the width of the image (it is a square).
    patch_size : int
        Both height and the width of the patch (it is a square).
    in_chans : int
        Number of input channels.
    n_classes : int
        Number of classes.
    embed_dim : int
        Dimensionality of the token/patch embeddings.
    depth : int
        Number of blocks.
    n_heads : int
        Number of attention heads.
    mlp_ratio : float
        Determines the hidden dimension of the `MLP` module.
    qkv_bias : bool
        If True then we include bias to the query, key and value projections.
    p, attn_p : float
        Dropout probability.
    Attributes
    ----------
    patch_embed : PatchEmbed
```

```
        Instance of `PatchEmbed` layer.
    cls_token : nn.Parameter
        Learnable parameter that will represent the first token in the sequence.
        It has `embed_dim` elements.
    pos_emb : nn.Parameter
        Positional embedding of the cls token + all the patches.
        It has `(n_patches + 1) * embed_dim` elements.
    pos_drop : nn.Dropout
        Dropout layer.
    blocks : nn.ModuleList
        List of `Block` modules.
    norm : nn.LayerNorm
        Layer normalization.
    """
    def __init__(
            self,
            img_size=32,
            patch_size=16,
            in_chans=2,
            n_classes=1,
            embed_dim=768,
            depth=12,
            n_heads=12,
            mlp_ratio=4.,
            qkv_bias=True,
            p=0.,
            attn_p=0.,
    ):
        super().__init__()

        self.patch_embed = PatchEmbed(
                img_size=img_size,
                patch_size=patch_size,
                in_chans=in_chans,
                embed_dim=embed_dim,
        )
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(
                torch.zeros(1, 1 + self.patch_embed.n_patches, embed_dim)
        )
        self.pos_drop = nn.Dropout(p=p)

        self.blocks = nn.ModuleList(
            [
                Block(
                    dim=embed_dim,
                    n_heads=n_heads,
                        mlp_ratio=mlp_ratio,
                        qkv_bias=qkv_bias,
                        p=p,
                        attn_p=attn_p,
                    )
                    for _ in range(depth)
                ]
            )

            self.norm = nn.LayerNorm(embed_dim, eps=1e-6)
            self.head = nn.Linear(embed_dim, n_classes)


        def forward(self, x):
            """Run the forward pass.
            Parameters
            ----------
            x : torch.Tensor
                Shape `(n_samples, in_chans, img_size, img_size)`.
            Returns
            -------
            logits : torch.Tensor
                Logits over all the classes - `(n_samples, n_classes)`.
            """
            n_samples = x.shape[0]
            x = self.patch_embed(x)

            cls_token = self.cls_token.expand(
                    n_samples, -1, -1
            )  # (n_samples, 1, embed_dim)
            x = torch.cat((cls_token, x), dim=1)  # (n_samples, 1 + n_patches,␣
↪embed_dim)
            x = x + self.pos_embed  # (n_samples, 1 + n_patches, embed_dim)
            x = self.pos_drop(x)

            for block in self.blocks:
                x = block(x)

            x = self.norm(x)

            cls_token_final = x[:, 0]  # just the CLS token
            x = self.head(cls_token_final)
            x=torch.sigmoid(x)


            return x
```

[34]:
```
model=VisionTransformer()
```

[35]:
```
from tqdm import tqdm
```

10 Training Loop

number of epochs = 15

criterion = nn.BCELoss()

optimizer = optim.Adam()

batch_size = 32

[ ]:
```
import torch
```

[ ]:
```
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# define hyperparameters
batch_size = 32
learning_rate = 1e-5
num_epochs = 15

model=model.to(device)


# define loss function and optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# initialize variables for storing loss and epoch count
train_loss = []
epoch_count = []

# train model
for epoch in (range(num_epochs)):
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs and labels from the dataloader
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()

    # store the epoch loss and epoch count
    train_loss.append(running_loss / len(train_loader))
    epoch_count.append(epoch + 1)

    # print epoch statistics
    print('Epoch %d loss: %.3f' % (epoch+1, running_loss / len(train_loader)))

# plot the loss graph
plt.plot(epoch_count, train_loss)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

print('Finished training')
```

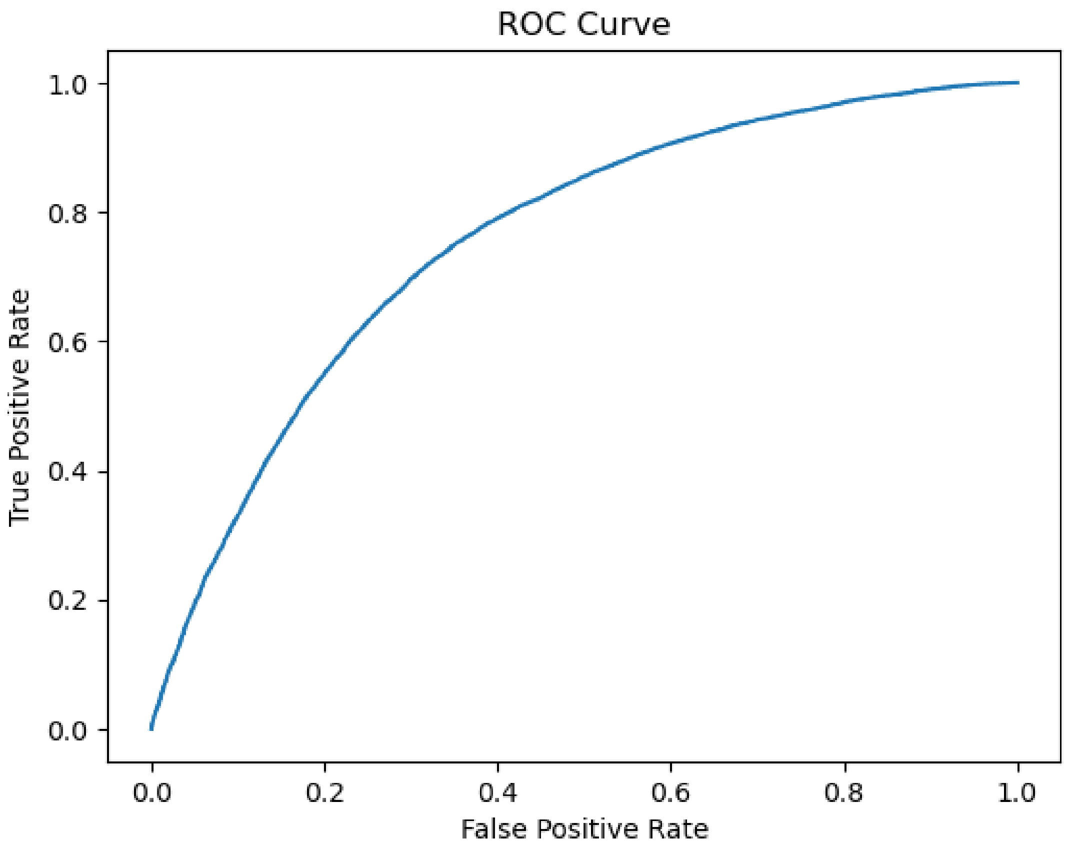
[37]:
```
from sklearn.metrics import roc_auc_score, roc_curve
# set model to evaluation mode
model.eval()
mdoel=model.to('cpu')

# evaluate test data
with torch.no_grad():
    y_pred = (model(X_test))
#     test_loss = nn.BCELoss()(y_pred, y_test.float())
    test_roc_auc = roc_auc_score(y_test, y_pred)

# print('Test Loss: %.3f' % test_loss)
print('Test ROC AUC: %.3f' % test_roc_auc)

# plot ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

```
Test ROC AUC: 0.757
```



This is the ROC AUC curve after training the model....

Due to resource shortage I couldnt manage to run more epochs and one thing i noticed is that

- when the epochs were increased little more after 15 the loss fell sharply and the model overfitted the data because the ROC AUC fell as the epochs were incerased in that case
- I had saved the models state_dict and then transfered laerned it due to again resource shortage my model used to get chrashed again and again.

11 comparision between CNN and ViT

1. ROC AUC - the value of this almost close by. Talking about the partiality to CNN that it had more epochs trained and more time bevause it was trained very quickly (linear complexity). Other hand for Vision Transformer model very less number of epochs were possible to be trained and got less time to tune the hyper parameters as the training time was big (quadractic compelxity)
2. difference in the training part:

   –> CNN had more epochs

   –> learning rate was 100 times smaller in case of ViT

–> The models are different

3. **RESULTS**

–> the ROC AUC OF CNN = 80.2

–> the ROC AUC OF ViT = 75.7 (76.2)

12 Further plans in the ViT Model:

1. As the project is based on Vision Transformer for binary classification, one of the future plans is to further optimize the model's performance. We can explore ways to fine-tune the hyperparameters and increase the dataset size.

- Have thought of the different modifications which I would perform. → As stated in the research paper "Attention is all you need" , The time complexity of the model is O(m^2) . That is the time complexity of the model is quadratic and while I trained the model I also felt the same, It took lots of time even after using a GPU. By using filters with stride I am expecting better results because in that case the number of filters is reduced and the time is saved

  ```
  Time plays a crucial role so that we can experiment with the model and tune the hyperpar
  ```

  ```
  → Use a hybrid between the CNN and the Vision Transformer: What I meant by a hybrid is th
  ```

- Ensemble: To increase the performance of the model I have thought to implement the ensemble between multiple models.
- As I read the research paper I came to know that ViT models can outcast the CNN models only with a huge amount of data. When there is less data CNN works better than ViT and vice versa when there is more data. So I want to extend the model to more training examples. One more idea that I have is that to use Data Augmentation technique.
- Talking about the validation of the model I have just split the data into a train and test set. I want to extend this to K-fold cross validation so that I don't overfit the model and can fine tune the hyperparameters.
- Additionally, we could investigate ways to apply the model to other similar problems or tasks. We can extend the vision transformer to the particle classification which is used in the Compact Muon Solenoid (CMS) experiments in the Large Hadron Collider (LHC).

```
[ ]:
```