DeepLense: Exploring Transformers

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
!tar zxf gdrive/MyDrive/lenses.tar.gz
```

```
!pip install pytorch_lightning
```

```
Requirement already satisfied: pytorch_lightning in /usr/local/lib/python3.7
Requirement already satisfied: torchmetrics>=0.4.1 in /usr/local/lib/python3
Requirement already satisfied: PyYAML>=5.4 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: typing-extensions>=4.0.0 in /usr/local/lib/py
Requirement already satisfied: pyDeprecate<0.4.0,>=0.3.1 in /usr/local/lib/p
Requirement already satisfied: numpy>=1.17.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: torch>=1.8.* in /usr/local/lib/python3.7/dist
Requirement already satisfied: packaging>=17.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: tensorboard>=2.2.0 in /usr/local/lib/python3.
Requirement already satisfied: tqdm>=4.41.0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: fsspec[http]!=2021.06.0,>=2021.05.0 in /usr/l
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: aiohttp in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/py
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr
Requirement already satisfied: protobuf>=3.6.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/loca
Requirement already satisfied: absl-py>=0.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: grpcio>=1.24.3 in /usr/local/lib/python3.7/di
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/pytho
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/l
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/d
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dis
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/pyth
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/pytho
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/py
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/pyt
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /u
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.7
```

✓ 1m 7s completed at 7:24 PM

```
Requirement already satisfied: charset-normalizer<3.0,>=2.0 in /usr/local/li
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3
Requirement already satisfied: asynctest==0.13.0 in /usr/local/lib/python3.7
```

```
# Standard libraries
import os
import shutil
import numpy as np
import random
import math
import json
from functools import partial
from PIL import Image

# Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline

# PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim

# Torchvision
import torchvision
from torchvision import transforms
from torchvision.utils import make_grid
from torchvision.datasets import DatasetFolder, ImageFolder
from torch.utils.data import DataLoader

# Imports for ROC AUC
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, roc_auc_score, auc
from itertools import cycle

# Imports for PyTorch Lightning
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
pl.seed_everything(42)

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("c
print("Device:", device)
```

```
Global seed set to 42
Device: cuda:0
```

```
# Kaggle kernel
LENSES_DATASET_PATH = "/content/lenses" # Should point to the root of the dataset
BATCH_SIZE = 16
TEST_SPLIT=0.1
```

```
VAL_SPLIT=0.01
INPUT_HEIGHT = 150
INPUT_WIDTH = 150
TRAIN='train'
TEST='test'
VAL='val'


lenses_files = []
for folder in {'sub', 'no_sub'}:
    for file in os.listdir(os.path.join(LENSES_DATASET_PATH, folder)):
        if file.endswith(".jpg"):
            lenses_files.append(os.path.join(LENSES_DATASET_PATH, folder, file))


def copy_images(imagePaths, folder):
    if not os.path.exists(folder):
        os.makedirs(folder)
    for path in imagePaths:
        imageName = path.split(os.path.sep)[-1]
        label = path.split(os.path.sep)[-2]
        labelFolder = os.path.join(folder, label)
        if not os.path.exists(labelFolder):
            os.makedirs(labelFolder)
        destination = os.path.join(labelFolder, imageName)
        shutil.copy(path, destination)


np.random.shuffle(lenses_files)
valPathsLen = int(len(lenses_files) * VAL_SPLIT)
testPathsLen = int(len(lenses_files) * TEST_SPLIT)
trainPathsLen = len(lenses_files) - valPathsLen - testPathsLen
print(f"Train : {trainPathsLen}, Test: {testPathsLen}, Val:{valPathsLen}")
```

```
Train : 8900, Test: 1000, Val:100
```

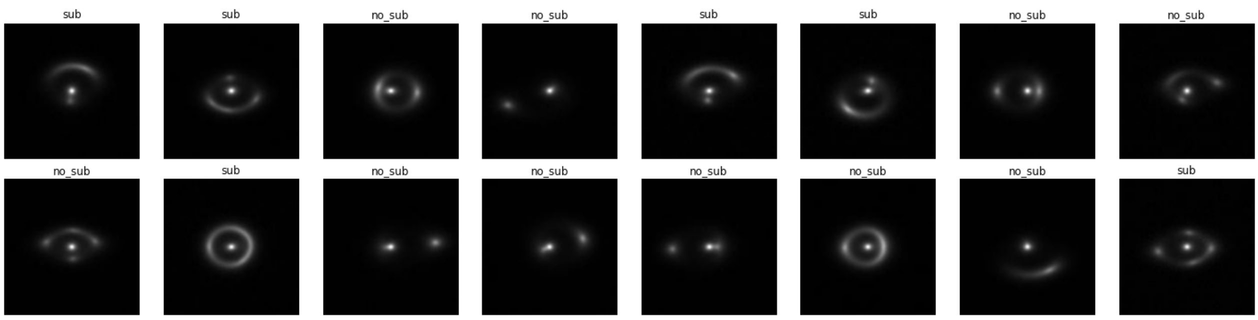
```
testInd = trainPathsLen + testPathsLen


trainPaths = lenses_files[:trainPathsLen]
testPaths = lenses_files[trainPathsLen:testInd]
valPaths = lenses_files[testInd:]


copy_images(trainPaths, TRAIN)
copy_images(testPaths, TEST)
copy_images(valPaths, VAL)


trainTransforms = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.ToTensor()
])
testTransforms = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.ToTensor()
```

```
    transforms.ToTensor()
])


trainDataset = ImageFolder(root=TRAIN, transform=trainTransforms)
testDataset = ImageFolder(root=TEST, transform=testTransforms)
valDataset = ImageFolder(root=VAL, transform=testTransforms)

print(f"[INFO] Training dataset contains {len(trainDataset)} samples.")
print(f"[INFO] Test dataset contains {len(testDataset)} samples.")
print(f"[INFO] Validation dataset contains {len(valDataset)} samples.")
```

```
[INFO] Training dataset contains 8900 samples.
[INFO] Test dataset contains 1000 samples.
[INFO] Validation dataset contains 100 samples.
```

```
trainDataLoader = DataLoader(trainDataset,
        batch_size=BATCH_SIZE, shuffle=True, num_workers=2)
testDataLoader = DataLoader(testDataset, shuffle=False, drop_last=False, batch_siz
valDataLoader = DataLoader(valDataset, shuffle=False, drop_last=False, batch_size=


def visualize_batch(batch, classes, dataset_type):
    fig = plt.figure("{} batch".format(dataset_type), figsize=(20, 5))
    for i in range(0, BATCH_SIZE):
        ax = plt.subplot(2, 8, i+1)
        image = batch[0][i].cpu().numpy()
        image = image.transpose((1, 2, 0))
        image = (image * 255.0).astype("uint8")
        idx = batch[1][i]
        label = classes[idx]
        plt.imshow(image)
        plt.title(label)
        plt.axis("off")
    plt.tight_layout()
    plt.show()


trainBatch = next(iter(trainDataLoader))
print(trainBatch[0].shape)
visualize_batch(trainBatch, trainDataset.classes, "train")
```

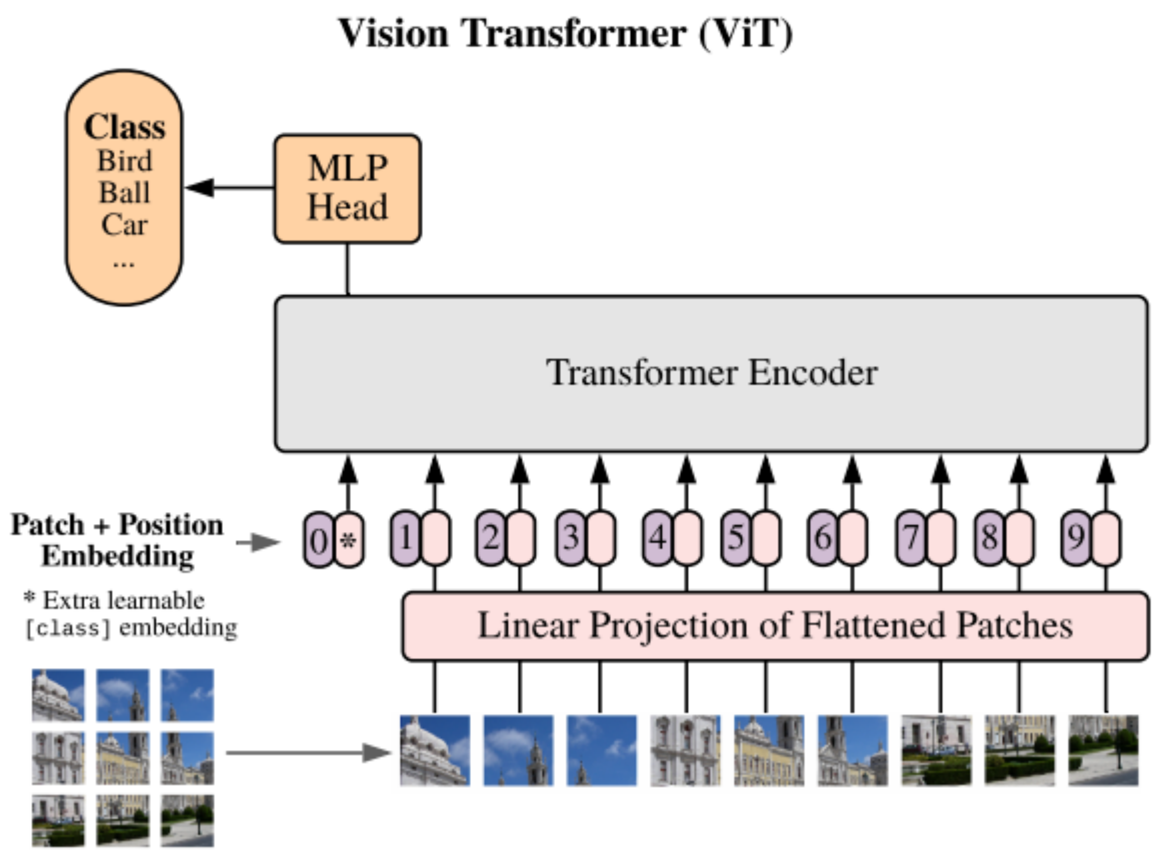
```
torch.Size([16, 3, 224, 224])
```



Breaking image into multiple patches and the flattening them.m

```
def img_to_patch(x, patch_size, flatten_channels=True):
    B, C, H, W = x.shape
    #x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size, patch_size)
    x = x.reshape(B, C, torch.div(H, patch_size, rounding_mode='trunc'), patch_s

    x = x.permute(0, 2, 4, 1, 3, 5)
    x = x.flatten(1,2)
    if flatten_channels:
        x = x.flatten(2,4)
    return x
```

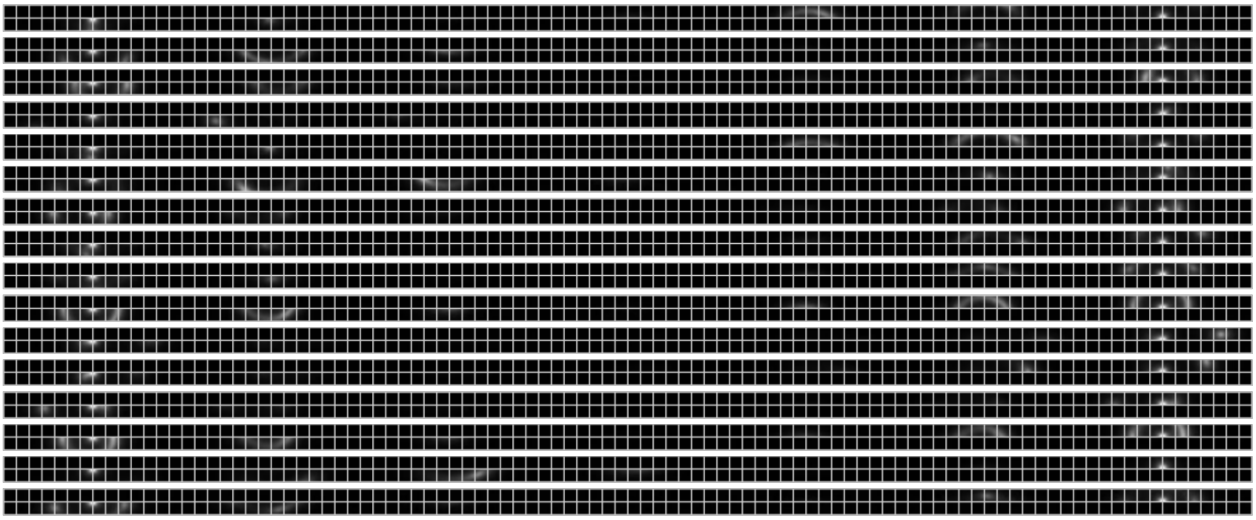
Vision Transformer



Patch Embeddings

```
img_patches = img_to_patch(trainBatch[0], patch_size=16, flatten_channels=False)

fig, ax = plt.subplots(trainBatch[0].shape[0], 1, figsize=(40,10))

print('Display patch embedding result: ')
# 224x224 images, split as 16x16 patches so 196 patches in total, displaying 98 p

for i in range(trainBatch[0].shape[0]):
    img_grid = make_grid(img_patches[i], nrow=98, normalize=True, pad_value=0.8)
    img_grid = img_grid.permute(1, 2, 0)
    ax[i].imshow(img_grid)
    ax[i].axis('off')
plt.show()
plt.close()
```

```
Display patch embedding result:
```



The Vision Transformer model

```
class AttentionBlock(nn.Module):
    """
        embed_dim - dims of the input and attn of feature vectors
        hidden_dim - dims of hidden layer in FFN i.e 2 x embed_dim
        num_heads - number of heads in MHA block
        dropout - amount of dropout to apply in FFN
    """
    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout)
        )


    def forward(self, x):
        inp_x = self.layer_norm_1(x)
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x


class VisionTransformer(nn.Module):
    """
        embed_dim - dims of the input feature vectors
        hidden_dim - dims of the hidden layer in the FFN within transformer
        num_channels - num of channels of the input i.e. 3 in our case of RGB ima
        num_heads - num of heads to use in the MHA block
        num_layers - num of layers in transformer
        num_classes - num of classes to predict
        patch_size - num of pixels that the patches have per dim
        num_patches - max number of patches of an image
        dropout - amount of dropout to apply in the FFN and on the input encoding
    """
    def __init__(self, embed_dim, hidden_dim, num_channels, num_heads, num_layers
        super().__init__()

        self.patch_size = patch_size

        self.input_layer = nn.Linear(num_channels*(patch_size**2), embed_dim)
        self.transformer = nn.Sequential(*[AttentionBlock(embed_dim, hidden_dim,
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(embed_dim),
            nn.Linear(embed_dim, num_classes)
        )
        self.dropout = nn.Dropout(dropout)
```

```
        self.cls_token = nn.Parameter(torch.randn(1,1,embed_dim))
        self.pos_embedding = nn.Parameter(torch.randn(1,1+num_patches,embed_dim)


    def forward(self, x):
        x = img_to_patch(x, self.patch_size)
        B, T, _ = x.shape
        x = self.input_layer(x)

        cls_token = self.cls_token.repeat(B, 1, 1)
        x = torch.cat([cls_token, x], dim=1)
        x = x + self.pos_embedding[:,:T+1]

        x = self.dropout(x)
        x = x.transpose(0, 1)
        x = self.transformer(x)

        cls = x[0]
        out = self.mlp_head(cls)
        return out


# We will be using PyTorch's Lightning module to organize our model code.

class ViT(pl.LightningModule):
    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)
        self.example_input_array = next(iter(trainDataLoader))[0]
        self.predictions = [] # Actual predictions for ROC AUC

    def forward(self, x):
        return self.model(x)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100,
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        if mode == 'test':
            self.predictions.append(preds.argmax(dim=-1))

        self.log(f'{mode}_loss', loss, )
        self.log(f'{mode}_acc', acc)
        return loss

    def training_step(self, batch, batch_idx):
```

```
    def training_step(self, batch, batch_idx):
        loss = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="test")


CHECKPOINT_PATH = "./saved_models/"


def train_model(**kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ViT"),
                         gpus=1 if str(device)=="cuda:0" else 0,
                         max_epochs=180,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode=
                                    LearningRateMonitor("epoch")],
                         enable_progress_bar=False,
                         log_every_n_steps=4)

    #trainer.logger._log_graph = True         # If True, we plot the computation
    #trainer.logger._default_hp_metric = None # Optional logging argument that we

    pl.seed_everything(42, workers=True) # To be reproducable
    model = ViT(**kwargs)
    trainer.fit(model=model, train_dataloaders=trainDataLoader, val_dataloaders=va

    # Using PyTorch Lightning we can load a revised model and use it as the best
    val_result = trainer.validate(model, dataloaders=valDataLoader, verbose=False
    test_result = trainer.test(model, dataloaders=testDataLoader, verbose=False)

    result = {"test": test_result[0]["test_acc"], "val": val_result[0]["val_acc"]

    return model, result


EMBED_DIM=256
HIDDEN_DIM=512
RESIZE_IMG=224
NUM_HEADS=8
NUM_LAYERS=6
PATCH_SIZE=16
NUM_CHANNELS=3
NUM_PATCHES=int((RESIZE_IMG*RESIZE_IMG) / (PATCH_SIZE*PATCH_SIZE))
NUM_CLASSES=2
DROPOUT=0.2
```

Training the model

```
model, results = train_model(model_kwargs={
```

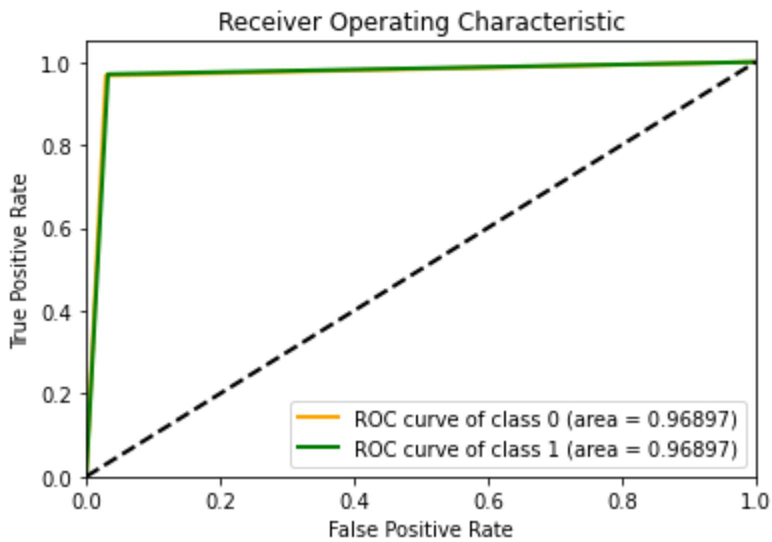
```
                                  'embed_dim': EMBED_DIM,
                                  'hidden_dim': HIDDEN_DIM,
                                  'num_heads': NUM_HEADS,
                                  'num_layers': NUM_LAYERS,
                                  'patch_size': PATCH_SIZE,
                                  'num_channels': NUM_CHANNELS,
                                  'num_patches': NUM_PATCHES,
                                  'num_classes': NUM_CLASSES,
                                  'dropout': DROPOUT
                              }, lr=3e-4)
print("ViT results", results)
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
Global seed set to 42
Missing logger folder: saved_models/ViT/lightning_logs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name  | Type              | Params | In sizes          | Out sizes
----------------------------------------------------------------------------
0 | model | VisionTransformer | 3.4 M  | [16, 3, 224, 224] | [16, 2]
----------------------------------------------------------------------------
3.4 M     Trainable params
0         Non-trainable params
3.4 M     Total params
13.645    Total estimated model params size (MB)
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
ViT results {'test': 0.968999981880188, 'val': 0.9800000190734863}
```

ROC AUC Score

```
class LB(LabelBinarizer):
    def transform(self, y):
        Y = super().transform(y)
        if self.y_type_ == 'binary':
            return np.hstack((Y, 1-Y))
        else:
            return Y

    def inverse_transform(self, Y, threshold=None):
        if self.y_type_ == 'binary':
            return super().inverse_transform(Y[:, 0], threshold)
        else:
            return super().inverse_transform(Y, threshold)


y_score = torch.cat(model.predictions).cpu().detach().numpy()

y_test = []
for _, labels in testDataLoader:
```

```
    y_test.append(labels.cpu().detach().numpy())

y_test = np.concatenate(y_test)


lb = LB()
y_test = lb.fit_transform(y_test)
y_score = lb.fit_transform(y_score)


n_classes = y_test.shape[1]
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
colors = ['orange', 'green']

for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2, label='ROC curve of class {0} (a:

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Save Model

```
torch.save(model.state_dict(), 'st5_model.pth')
```