CMPE 597 Sp. Tp. Deep Learning - Term Project

MIM Architecture - CIFAR100 Dataset

Anıl Turgut - 2022702072

Selahattin Seha Cirit - 2023705006

In this Jupyter Notebook, a study was carried out by finetuning the classification model with the image embeddings obtained using the pre-trained self-supervised learning model that we proposed in the project. Image embeddings in this notebook were created using the **MIM_MAE_Refined_l16** model developed by *Institute for Machine Learning, Johannes Kepler University Linz*. As the output of this model, there is an embedding list output with **201728** dimensions (after reshaping 3D to 2D) for each image. The results of these embeddings resulting from the pretext task were analyzed using *Three Layer MLP* and *SupportVectorClassifier (SVC)* models.

Moreover, this notebook includes the analysis of *self-supervised learning* (spesifically MIM) with **CIFAR100** dataset. In the following tasks, we are introduce our work in detail. Let's move on.

Task 1: Importing Libraries

In [26]:

```
import copy
import cv2
import glob
import json
import keras
import matplotlib.pyplot as plt
import numpy as np
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import torchvision.transforms as T
import torchvision.datasets as datasets
import tqdm
from tqdm.notebook import tqdm as tqdm_note
import zipfile
from copy import deepcopy
from PIL import Image
import pandas as pd
from sklearn.metrics import accuracy_score, classification_report, f1_score, precision_s
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import socket
from tensorflow.keras import datasets as tfdatasets, layers, models
```

The libraries to be used have been imported as in the cell above. *TensorFlow Keras*, *Torch* and *ScikitLearn* libraries were used when establishing classification models. Other libraries are also used for different purposes.

Task 2.1: Extracting CIFAR100 Dataset

In [27]:
```
class_names = pd.read_csv("cifar100_classnames.csv")

if 'Unnamed: 0' in class_names.columns:
    class_names.drop(columns=['Unnamed: 0'], inplace=True)

(train_images, train_labels), (test_images, test_labels) = tfdatasets.cifar100.load_data

class_names.head()
```

Out[27]:

	ClassName
0	apple
1	aquarium_fish
2	baby
3	bear
4	beaver

In [28]:
```
train_size = int(len(train_images) * 0.05)
train_images, train_labels = train_images[:train_size], train_labels[:train_size]
test_size = int(len(test_images) * 0.05)
test_images, test_labels = test_images[:test_size], test_labels[:test_size]
print(train_images.shape, train_labels.shape)
print(test_images.shape, test_labels.shape)
```

```
(2500, 32, 32, 3) (2500, 1)
(500, 32, 32, 3) (500, 1)
```

Dataset size is decreased intentionally due to the fact that MIM architecture has almost 200k dimension for each image embeddings and even the embedding size of 2500 images are near 13gb in disk. Our local computers are not enable to handle pretext task of 3000 CIFAR100 images. Therefore, we will make our analysis based on this assumption.

The *CIFAR-100* dataset consists of 60000 32x32 colour images in 100 subclasses, with 600 images per class. There are 50000 training images and 10000 test images.

This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Reference: https://www.cs.toronto.edu/~kriz/cifar.html

class_names list is defined accordingly to the referenced source to test the results properly

Task 2.2: Pretext Task - Preparing MIM model

In [29]:
```
torch.manual_seed(88)
mim_mael16 = torch.hub.load("ml-jku/MIM-Refiner", "mae_refined_l16")
```

```
Using cache found in C:\Users\anil.turgut/.cache\torch\hub\ml-jku_MIM-Refiner_main
```

MIM (Masked Image Modeling)-Refiner, a contrastive learning boost for pre-trained MIM models. The motivation behind MIM-Refiner is rooted in the insight that optimal representations within MIM models generally reside in intermediate layers. Accordingly, MIM-Refiner leverages multiple contrastive heads that are connected to diverse intermediate layers. In each head, a modified nearest neighbor objective helps to construct respective semantic clusters.

We have used the **MIM MAE Refiner l16** model with almost 1.1 gb size.

Using **Torch.hub**, we have loaded the model to our working environment to compute image embeddings as pretext task.

References:

- https://github.com/ml-jku/MIM-Refiner
- https://arxiv.org/abs/2402.10093
- https://paperswithcode.com/sota/self-supervised-image-classification-on

In [30]:
```
device = torch.device('cuda' if torch.cuda.is_available() else "cpu") # not enough gpu m
#device = "cpu"
device
```

Out[30]:
```
device(type='cuda')
```

Device is mainly selected as Cuda due to its performance. However, our local machines have not powerful gpu (*NVIDIA GeForce MX330 2GB*), sometimes *CPU* is selected intentionally.

In [31]:
```
mim_mael16.to(device)
```

Out[31]:
```
PrenormVit(
  (patch_embed): VitPatchEmbed(
    (proj): Conv2d(3, 1024, kernel_size=(16, 16), stride=(16, 16))
  )
  (pos_embed): VitPosEmbed2d()
  (cls_tokens): VitClassTokens()
  (blocks): ModuleList(
    (0-23): 24 x PrenormBlock(
      (norm1): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
      (attn): DotProductAttention1d(
        (qkv): Linear(in_features=1024, out_features=3072, bias=True)
        (proj): Linear(in_features=1024, out_features=1024, bias=True)
      )
      (drop_path1): DropPath(drop_prob=0.000)
      (norm2): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=1024, out_features=4096, bias=True)
        (act): GELU(approximate='none')
        (fc2): Linear(in_features=4096, out_features=1024, bias=True)
      )
      (drop_path2): DropPath(drop_prob=0.000)
    )
  )
  (norm): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
)
```

We moved our MIM model to the device to ensure that all computations involving the model parameters and inputs will be performed on the specified device. Also, all the computed images' embeddings will have **201728** dimensions (after reshaping 3D to 2D) with this *MIM* model. In other words, a CIFAR image has the shape of 32x32x3. MIM model generates embeddings from the image with a shape of **(1,197,1024)** and when we reshape it to use it our network, it results with **(1, 201728)**.

In [32]:
```
transform_image = T.Compose([T.ToTensor(), T.Resize(244), T.CenterCrop(224), T.Normalize

def load_image(img: str) -> torch.Tensor:

    img = Image.open(img)

    transformed_img = transform_image(img)[:3].unsqueeze(0)

    return transformed_img

def compute_embeddings(images: list) -> list:

    all_embeddings = []

    with torch.no_grad():

      for image in images:

        image = transform_image(image)[:3].unsqueeze(0)
        embeddings = mim_mael16(image.to(device))
        all_embeddings.append(np.array(embeddings[0].cpu().numpy()).reshape(1, -1).tolis

    return all_embeddings
def plot_image(tensor_image):
    # Convert tensor to NumPy array and transpose dimensions
    numpy_img = tensor_image.squeeze().permute(1, 2, 0).cpu().numpy()

    # Plot the image
    plt.imshow(numpy_img)
    plt.axis('off')
    plt.show()
```

Cell above have 3 functions to help while transforming image to the shape that *MIM* can understand. *load_image* and *plot_image* functions basically loads the *.jpg* or *.png* format images, transformes to the tensor and plot the loaded image respectively.

compute_embeddings function is defined to compute image embeddings from the given image list using *MIM* model. In our project, CIFAR training and test image datasets will be executed by this model and output embeddings will be an input for our downstream task -*Classification*-.

2.3 Computing/Loading Embeddings

Using *compute_embeddings* function above, we will compute the each image embeddings in train/test dataset. Then, we are going to store this embeddings as JSON file not to recalculate again and again.

In [ ]:
```
embeddings = compute_embeddings(train_images)

with open("CIFAR100Embeddings/_mimcifar100_all_embeddings.json", "w") as f:
        f.write(json.dumps(embeddings))

test_embeddings = compute_embeddings(test_images)

with open("CIFAR100Embeddings/_mimcifar100_all_embeddings_test.json", "w") as f:
        f.write(json.dumps(test_embeddings))
```

In [33]:
```
with open('CIFAR100Embeddings/_mimcifar100_all_embeddings.json') as f:
    embeddings = json.load(f)
```

```
with open('CIFAR100Embeddings/_mimcifar100_all_embeddings_test.json') as f:
    test_embeddings = json.load(f)
```

embeddings are the computed embeddings for the CIFAR10 training images (2500 records) and **test_embeddings** are the computed embeddings for the CIFAR10 training images (500 records). We do not need to compute the embeddings for label (y values) since the pretext task is unlabeled.

In [34]:
```
copied_training_embeddings = embeddings.copy()
copied_training_embeddings = np.array(copied_training_embeddings).reshape(-1, 201728)
copied_test_embeddings = test_embeddings.copy()
copied_test_embeddings = np.array(copied_test_embeddings).reshape(-1, 201728)
```

Copied embeddings will be used in the SVC model in below.

2.4 Preparing Dataset for Training

In [35]:
```
X_train = embeddings
X_train = np.array(X_train).reshape(-1, 201728)
y_train = train_labels
ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y_train)
y_train = ohe.transform(y_train)
X_test = test_embeddings
X_test = np.array(X_test).reshape(-1, 201728)
y_test = test_labels
ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y_test)
y_test = ohe.transform(y_test)
```

In [36]:
```
# convert pandas DataFrame (X) and numpy array (y) into PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
```

In [37]:
```
print("Shape of X_train:",X_train.shape)
print("Shape of y_train:",y_train.shape)
print("Shape of X_test:",X_test.shape)
print("Shape of y_test:",y_test.shape)
```

```
Shape of X_train: torch.Size([2500, 201728])
Shape of y_train: torch.Size([2500, 100])
Shape of X_test: torch.Size([500, 201728])
Shape of y_test: torch.Size([500, 100])
```

In preparing dataset section, we reshaped the computed embeddings of images as (-1,201728). Also labels are redefined as one-hot-encoded list. Thus, each label record consists of 100 dimensions and including one *1* rest is *0*.

All the dataset items are converted to *tensor* to be used in perceptron model.

3.1 Downstream Task - Three Layer Classification Model

In [44]:
```
class MIMVisionTransformerClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(201728, 256)
        self.act = nn.ReLU()
        self.hidden2 = nn.Linear(256, 128)
        self.act = nn.ReLU()
```

```
        self.hidden3 = nn.Linear(128, 128)
        self.act = nn.ReLU()
        self.output = nn.Linear(128, 100)

    def forward(self, x):
        x = self.act(self.hidden(x))
        x = self.act(self.hidden2(x))
        x = self.act(self.hidden3(x))
        x = self.output(x)
        return x

# loss metric and optimizer
model = MIMVisionTransformerClassifier()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# prepare model and training parameters
n_epochs = 20
batch_size = 32
batches_per_epoch = len(X_train) // batch_size
```

Classifier is designed to classify image labels using embeddings using Three Hidden Layer perceptron model. 201728 image embeddings for each image will be corresponded by 3 hidden layers where first one with 256 hidden units and rest two have 128 hidden units. Output will be 100 class values through FC layer.

In this example, we have increased the number of layer since both number of training records (compulsorily) and class labels have been increased.

Loss is selected as CrossEntropyLoss since the problem is multi-class classification and optimizer is selected as *Adam* optimizer to reduce the effects of hyperparameters and including momentum and SGD mechanism.

After analyzing the model training process, epoch numbers and batch sizes selected accordingly.

In [45]:
```
best_acc = - np.inf   # init to negative infinity
best_weights = None
train_loss_hist = []
train_acc_hist = []
test_loss_hist = []
test_acc_hist = []
```

In [46]:
```
# training loop
for epoch in range(n_epochs):
    epoch_loss = []
    epoch_acc = []
    # set model in training mode and run through each batch
    model.train()
    with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for i in bar:
            # take a batch
            start = i * batch_size
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # forward pass
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            # backward pass
            optimizer.zero_grad()
            loss.backward()
            # update weights
            optimizer.step()
```

```python
            # compute and store metrics
            acc = (torch.argmax(y_pred, 1) == torch.argmax(y_batch, 1)).float().mean()
            epoch_loss.append(float(loss))
            epoch_acc.append(float(acc))
            bar.set_postfix(
                loss=float(loss),
                acc=float(acc)
            )
    # set model in evaluation mode and run through the test set
    model.eval()
    y_pred = model(X_test)
    ce = loss_fn(y_pred, y_test)
    acc = (torch.argmax(y_pred, 1) == torch.argmax(y_test, 1)).float().mean()
    ce = float(ce)
    acc = float(acc)
    train_loss_hist.append(np.mean(epoch_loss))
    train_acc_hist.append(np.mean(epoch_acc))
    test_loss_hist.append(ce)
    test_acc_hist.append(acc)
    if acc > best_acc:
        best_acc = acc
        best_weights = copy.deepcopy(model.state_dict())
    print(f"Epoch {epoch} validation: Cross-entropy={ce:.2f}, Accuracy={acc*100:.1f}%")

# Restore best model
model.load_state_dict(best_weights)
```

```
Epoch 0: 100%|███████████████████████████████████████████████| 78/78 [00:26<00:00,  2.90
batch/s, acc=0.0312, loss=4.51]
Epoch 0 validation: Cross-entropy=4.55, Accuracy=2.6%
Epoch 1: 100%|████████████████████████████████████████████████████| 78/78 [00:26<00:00,
2.96batch/s, acc=0, loss=3.99]
Epoch 1 validation: Cross-entropy=3.98, Accuracy=7.8%
Epoch 2: 100%|███████████████████████████████████████████████| 78/78 [00:28<00:00,  2.76
batch/s, acc=0.0312, loss=3.58]
Epoch 2 validation: Cross-entropy=3.87, Accuracy=10.2%
Epoch 3: 100%|███████████████████████████████████████████████| 78/78 [00:26<00:00,  2.91
batch/s, acc=0.0625, loss=3.45]
Epoch 3 validation: Cross-entropy=3.54, Accuracy=12.0%
Epoch 4: 100%|████████████████████████████████████████████████| 78/78 [00:28<00:00,  2.7
9batch/s, acc=0.219, loss=2.79]
Epoch 4 validation: Cross-entropy=3.38, Accuracy=17.4%
Epoch 5: 100%|████████████████████████████████████████████████| 78/78 [00:27<00:00,  2.8
3batch/s, acc=0.344, loss=2.24]
Epoch 5 validation: Cross-entropy=3.41, Accuracy=17.6%
Epoch 6: 100%|████████████████████████████████████████████████| 78/78 [00:27<00:00,  2.8
0batch/s, acc=0.406, loss=1.97]
Epoch 6 validation: Cross-entropy=3.68, Accuracy=16.0%
Epoch 7: 100%|████████████████████████████████████████████████| 78/78 [00:28<00:00,  2.7
0batch/s, acc=0.438, loss=1.84]
Epoch 7 validation: Cross-entropy=3.47, Accuracy=17.8%
Epoch 8: 100%|████████████████████████████████████████████████| 78/78 [00:39<00:00,  2.0
0batch/s, acc=0.188, loss=2.49]
Epoch 8 validation: Cross-entropy=4.29, Accuracy=13.2%
Epoch 9: 100%|█████████████████████████████████████████████████| 78/78 [00:58<00:00,  1.
32batch/s, acc=0.25, loss=2.55]
Epoch 9 validation: Cross-entropy=4.34, Accuracy=13.6%
Epoch 10: 100%|███████████████████████████████████████████████| 78/78 [01:00<00:00,  1.3
0batch/s, acc=0.281, loss=2.48]
Epoch 10 validation: Cross-entropy=5.70, Accuracy=10.4%
Epoch 11: 100%|███████████████████████████████████████████████| 78/78 [00:56<00:00,  1.3
7batch/s, acc=0.438, loss=2.12]
```

```
Epoch 11 validation: Cross-entropy=4.07, Accuracy=18.6%
Epoch 12: 100%|████████████████████████████████████████████████| 78/78 [00:54<00:00,  1.42batch/s, acc=0.25, loss=2.86]
Epoch 12 validation: Cross-entropy=4.85, Accuracy=15.0%
Epoch 13: 100%|███████████████████████████████████████████████| 78/78 [00:56<00:00,  1.38batch/s, acc=0.469, loss=1.57]
Epoch 13 validation: Cross-entropy=3.86, Accuracy=19.6%
Epoch 14: 100%|███████████████████████████████████████████████| 78/78 [00:54<00:00,  1.44batch/s, acc=0.219, loss=2.29]
Epoch 14 validation: Cross-entropy=5.21, Accuracy=11.8%
Epoch 15: 100%|█████████████████████████████████████████████████| 78/78 [00:57<00:00,  1.37batch/s, acc=0.5, loss=1.41]
Epoch 15 validation: Cross-entropy=4.84, Accuracy=15.6%
Epoch 16: 100%|███████████████████████████████████████████████| 78/78 [00:56<00:00,  1.38batch/s, acc=0.281, loss=2.25]
Epoch 16 validation: Cross-entropy=5.12, Accuracy=11.8%
Epoch 17: 100%|█████████████████████████████████████████████████| 78/78 [00:56<00:00,  1.38batch/s, acc=0.5, loss=1.48]
Epoch 17 validation: Cross-entropy=6.49, Accuracy=9.8%
Epoch 18: 100%|██████████████████████████████████████████████| 78/78 [00:56<00:00,  1.38batch/s, acc=0.844, loss=0.611]
Epoch 18 validation: Cross-entropy=5.06, Accuracy=17.4%
Epoch 19: 100%|███████████████████████████████████████████████| 78/78 [00:57<00:00,  1.35batch/s, acc=0.469, loss=1.78]
Epoch 19 validation: Cross-entropy=4.57, Accuracy=21.8%
```

Out[46]:

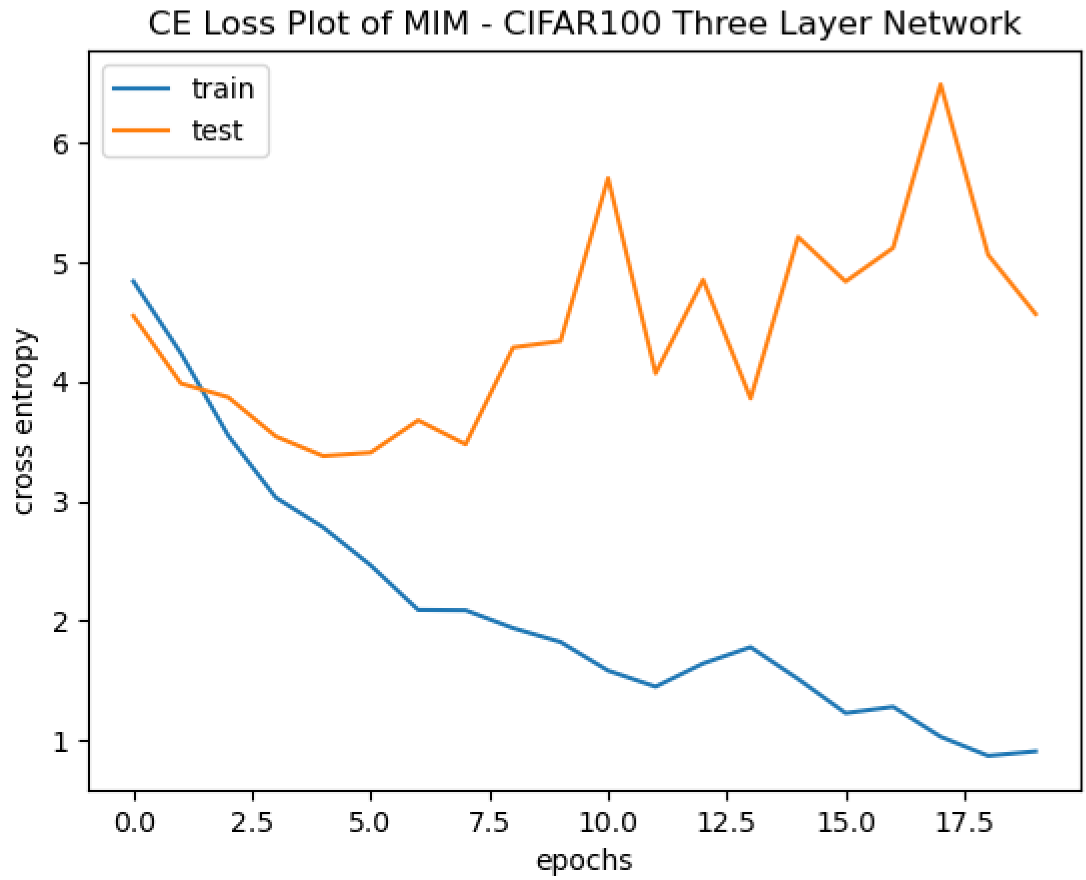
```
<All keys matched successfully>
```

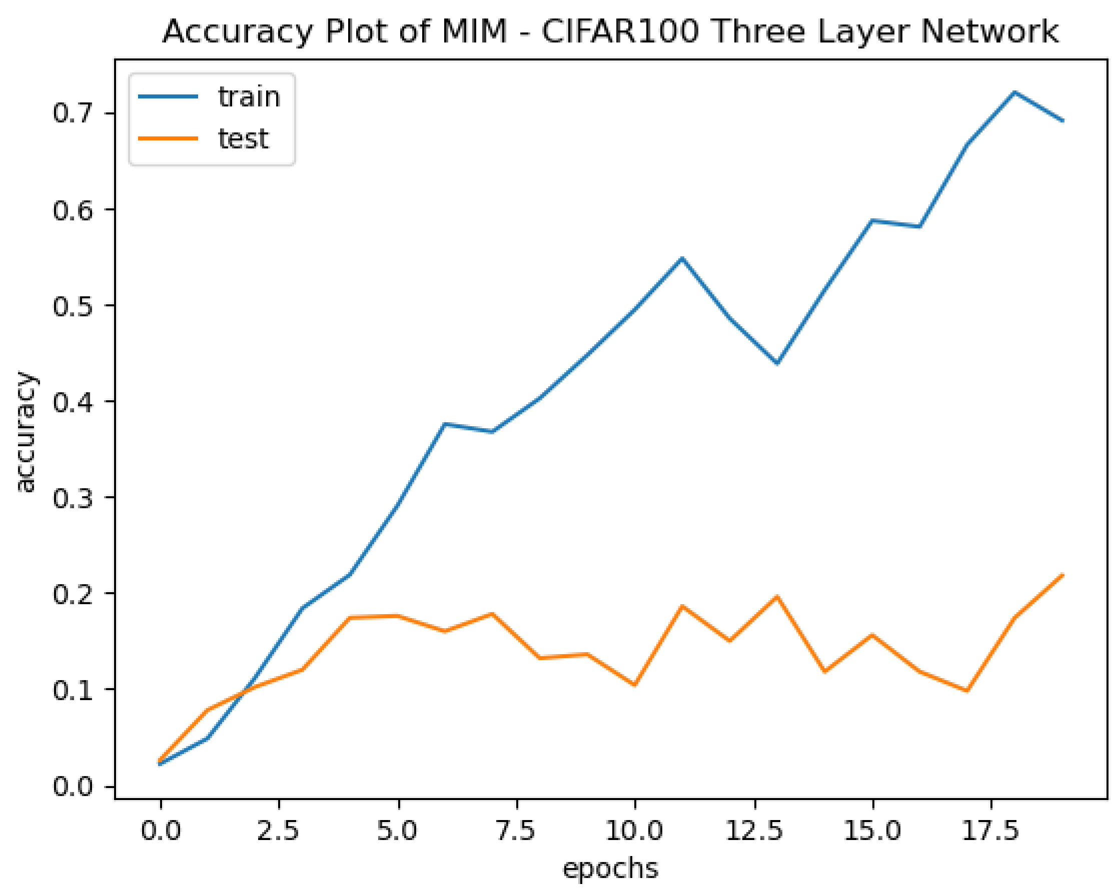
Model is trained 10 epochs and resulted with almost 70% percent training accuracy and 22% test accuracy.

In [83]:

```python
# Plot the loss and accuracy
plt.plot(train_loss_hist, label="train")
plt.plot(test_loss_hist, label="test")
plt.xlabel("epochs")
plt.ylabel("cross entropy")
plt.title("CE Loss Plot of MIM - CIFAR100 Three Layer Network")
plt.legend()
plt.show()

plt.plot(train_acc_hist, label="train")
plt.plot(test_acc_hist, label="test")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("Accuracy Plot of MIM - CIFAR100 Three Layer Network")
plt.legend()
plt.show()
```





Accuracy in training is continuosly improving as expected, whereas test accuracy kind of oscillates but improved. But CE Loss graph shows us that improving training performance might result with decrease in test performance as well.

3.2 Downstream Task - TensorFlow Trainer Classification

In this part, we have used the *TensorFlow Trainer* module to train classification network. Architecture is similar to the previous model.

In [57]:
```
from tensorflow.keras import models, layers

tf_model = models.Sequential()

tf_model.add(layers.Dense(256, activation='relu', input_shape=(201728,)))

tf_model.add(layers.Dense(128, activation='leaky_relu', input_shape=(256,)))

tf_model.add(layers.Dense(128, activation='relu', input_shape=(128,)))

tf_model.add(layers.Dense(100, activation='softmax'))

tf_model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 256)	51,642,624
dense_13 (Dense)	(None, 128)	32,896
dense_14 (Dense)	(None, 128)	16,512
dense_15 (Dense)	(None, 100)	12,900

Total params: 51,704,932 (197.24 MB)

Trainable params: 51,704,932 (197.24 MB)

Non-trainable params: 0 (0.00 B)

In [58]:
```
tf_model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Optimizer and loss function is selected as *adam* and *SparseCategoricalCrossEntropy* to see whether there is any improvement in changing loss function.

In [59]:
```
history = tf_model.fit(copied_training_embeddings, train_labels, epochs=10,
                       validation_data = (copied_test_embeddings, test_labels))
```

```
Epoch 1/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 39s 458ms/step - accuracy: 0.0093 - loss: 19.86
35 - val_accuracy: 0.0260 - val_loss: 4.5147
Epoch 2/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 28s 352ms/step - accuracy: 0.0351 - loss: 4.411
1 - val_accuracy: 0.0400 - val_loss: 4.4000
Epoch 3/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 28s 355ms/step - accuracy: 0.1211 - loss: 3.883
4 - val_accuracy: 0.0640 - val_loss: 4.5077
Epoch 4/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 28s 351ms/step - accuracy: 0.2581 - loss: 3.093
3 - val_accuracy: 0.1820 - val_loss: 3.7984
Epoch 5/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 27s 346ms/step - accuracy: 0.4978 - loss: 1.902
5 - val_accuracy: 0.2640 - val_loss: 3.3742
Epoch 6/10
79/79 ━━━━━━━━━━━━━━━━━━━━ 27s 346ms/step - accuracy: 0.7478 - loss: 0.942
```

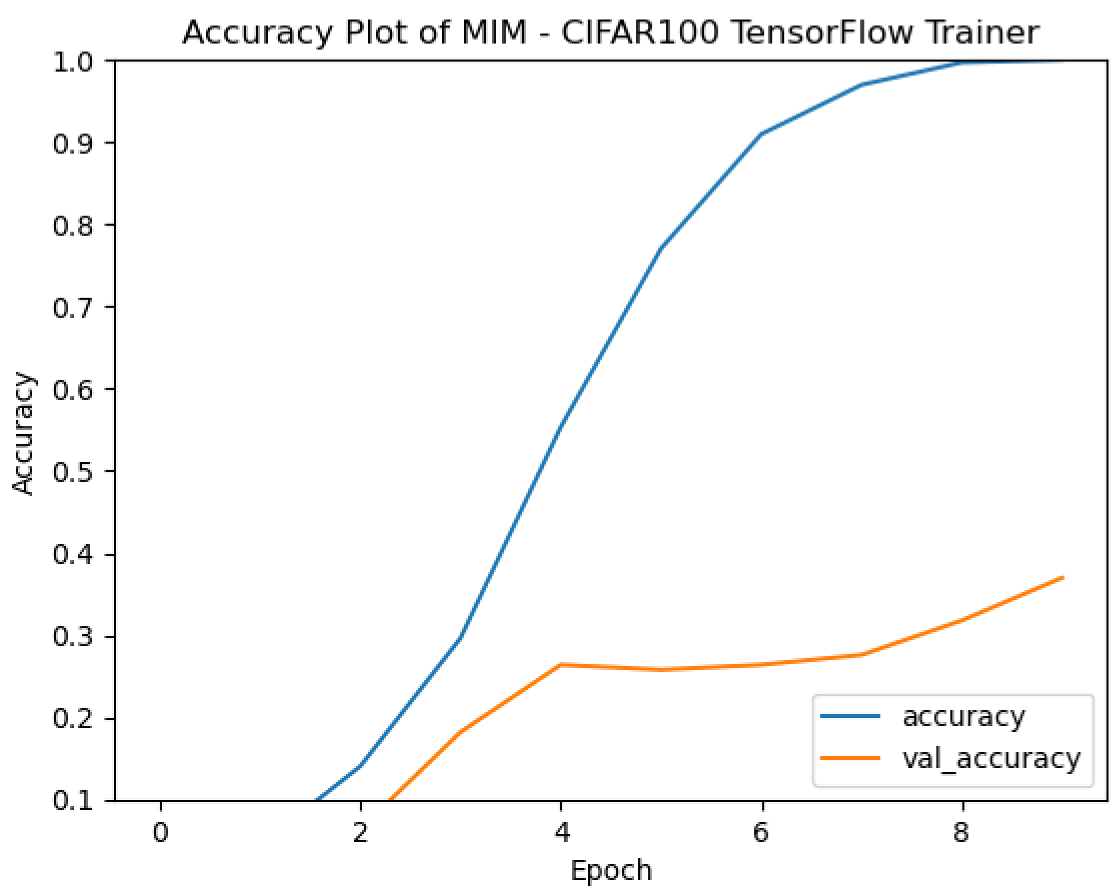
```
1 - val_accuracy: 0.2580 - val_loss: 3.6018
Epoch 7/10
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28s 351ms/step - accuracy: 0.9045 - loss: 0.412
2 - val_accuracy: 0.2640 - val_loss: 3.7134
Epoch 8/10
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28s 353ms/step - accuracy: 0.9608 - loss: 0.198
5 - val_accuracy: 0.2760 - val_loss: 3.6780
Epoch 9/10
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28s 356ms/step - accuracy: 0.9951 - loss: 0.064
4 - val_accuracy: 0.3180 - val_loss: 3.3912
Epoch 10/10
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28s 354ms/step - accuracy: 1.0000 - loss: 0.024
0 - val_accuracy: 0.3700 - val_loss: 3.1902
```

In [82]:

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title("Accuracy Plot of MIM - CIFAR100 TensorFlow Trainer")
plt.ylim([0.1, 1])
plt.legend(loc='lower right')
```

Out[82]:

```
<matplotlib.legend.Legend at 0x2643200fd90>
```



Accuracy plot shows that the results of *TF Trainer* and *Three Layer Perceptron* models are not similar in this dataset. Train and test performance of the network has been increased in *TF Trainer* 99% and 38% respectively.

Let's finally analyze the *Support Vector Classifier (SVC)* model to analyze the results.

3.3 Downstream Task - Classification Using Support Vector Classifier (SVC)

In [61]:
```python
from sklearn import svm

clf = svm.SVC(gamma='scale')

print(len(embeddings))

clf.fit(np.array(embeddings).reshape(-1, 201728), train_labels)
```

```
2500
C:\Users\anil.turgut\AppData\Local\anaconda3\Lib\site-packages\sklearn\utils\validation.py:1184: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[61]:
```
▾ SVC
SVC()
```

In [62]:
```python
from socket import socket
# train metrics
y_pred_train_list = []

for embed in tqdm_note(embeddings):

    y_pred = clf.predict(np.array(embed).reshape(1, -1))
    y_pred_train_list.append(y_pred[0])
```

```
  0%|          | 0/2500 [00:00<?, ?it/s]
```

In [63]:
```python
y_pred_test_list = []

for embed in tqdm_note(test_embeddings):

    y_pred = clf.predict(np.array(embed).reshape(1, -1))
    y_pred_test_list.append(y_pred[0])
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
```

In [65]:
```python
with open("CIFAR100Predictions/mimcifar100trainingpreds.json", "w") as f:
        f.write(json.dumps(np.array(y_pred_train_list).tolist()))

with open("CIFAR100Predictions/mimcifar100testpreds.json", "w") as f:
        f.write(json.dumps(np.array(y_pred_test_list).tolist()))
```

In [ ]:
```python
y_pred_train_list
with open('CIFAR100Predictions/mimcifar10trainingpreds.json') as f:
    y_pred_train_list = json.load(f)


with open('CIFAR100Predictions/mimcifar10testpreds.json') as f:
    y_pred_test_list = json.load(f)
```

In [66]:
```python
# Calculate evaluation metrics on the training data
accuracy_train = accuracy_score(train_labels, y_pred_train_list)
precision_train = precision_score(train_labels, y_pred_train_list, average='weighted')
recall_train = recall_score(train_labels, y_pred_train_list, average='weighted')
f1_train = f1_score(train_labels, y_pred_train_list, average='weighted')

print("Training Accuracy:", round(accuracy_train,6))
print("Training Precision:", round(precision_train,6))
print("Training Recall:", round(recall_train,6))
print("Training F1-score:", round(f1_train,6))
```

```
Training Accuracy: 0.0156
Training Precision: 0.000243
```

```
Training Recall: 0.0156
Training F1-score: 0.000479
```

```
C:\Users\anil.turgut\AppData\Local\anaconda3\Lib\site-packages\sklearn\metrics\_classifi
cation.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in
labels with no predicted samples. Use `zero_division` parameter to control this behavio
r.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [67]:

```
# Calculate evaluation metrics
accuracy = accuracy_score(test_labels, y_pred_test_list)
precision = precision_score(test_labels, y_pred_test_list, average='weighted')
recall = recall_score(test_labels, y_pred_test_list, average='weighted')
f1 = f1_score(test_labels, y_pred_test_list, average='weighted')

print("Test Accuracy:", round(accuracy,4))
print("Test Precision:", round(precision,4))
print("Test Recall:", round(recall,4))
print("Test F1-score:", round(f1,4))
```

```
Test Accuracy: 0.006
Test Precision: 0.0
Test Recall: 0.006
Test F1-score: 0.0001
```

```
C:\Users\anil.turgut\AppData\Local\anaconda3\Lib\site-packages\sklearn\metrics\_classifi
cation.py:1469: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in
labels with no predicted samples. Use `zero_division` parameter to control this behavio
r.
  _warn_prf(average, modifier, msg_start, len(result))
```

SVC performed very poorly on this embeddings, it might due to the dimension of the embeddings.

4.1 Testing Classifier Model Performance on Image from different Dataset

In [68]:

```
input_file = "C:\\Users\\anil.turgut\\Desktop\\CMPE597\\Project\\Code\\ExampleTestImages

new_image = load_image(input_file)

plot_image(new_image)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
```

In [65]:
```
mim_mael16.to(device)
```

Out[65]:
```
PrenormVit(
  (patch_embed): VitPatchEmbed(
    (proj): Conv2d(3, 1024, kernel_size=(16, 16), stride=(16, 16))
  )
  (pos_embed): VitPosEmbed2d()
  (cls_tokens): VitClassTokens()
  (blocks): ModuleList(
    (0-23): 24 x PrenormBlock(
      (norm1): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
      (attn): DotProductAttention1d(
        (qkv): Linear(in_features=1024, out_features=3072, bias=True)
        (proj): Linear(in_features=1024, out_features=1024, bias=True)
      )
      (drop_path1): DropPath(drop_prob=0.000)
      (norm2): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=1024, out_features=4096, bias=True)
        (act): GELU(approximate='none')
        (fc2): Linear(in_features=4096, out_features=1024, bias=True)
      )
      (drop_path2): DropPath(drop_prob=0.000)
    )
  )
  (norm): LayerNorm((1024,), eps=1e-06, elementwise_affine=True)
)
```

4.1.1 - SVC

In [74]:
```
with torch.no_grad():
    embedding = mim_mael16(new_image.to(device))

    embedding = np.array(embedding[0].detach().cpu().numpy()).reshape(1, -1).tolist()

    embedding = torch.tensor(embedding, dtype=torch.float32)

    embedding.to(device)
```

```
    prediction = clf.predict(np.array(embedding).reshape(1, -1))

    print("Predicted class: " + class_names['ClassName'].iloc[prediction[0]])
```

```
Predicted class: possum
```

4.1.2 - Three Layer Torch Network

In [80]:
```
with torch.no_grad():

    model.eval()
    prediction = model(embedding)
    prediction =  prediction.cpu().detach().numpy()
    max_indices = np.argsort(prediction, axis=1)[:, -2:]
    index = np.argmax(prediction, axis = 1)

    print("Predicted class: " + class_names['ClassName'].iloc[index[0]] +
          " with value " + str(prediction[0][max_indices[0][1]]))
    print("Second Predicted class: " + class_names['ClassName'].iloc[max_indices[0][0]]
          + " with value " + str(prediction[0][max_indices[0][0]]))
```

```
Predicted class: rocket with value 8.268451
Second Predicted class: castle with value 4.3918552
```

4.1.3 - TF Trainer

In [81]:
```
prediction = tf_model.predict(embedding)

print("Predicted class: " +class_names['ClassName'].iloc[np.argmax(prediction,axis = 1)[
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 342ms/step
Predicted class: beetle
```

Unfortunately, three of the classifiers are unable to correctly classify the image labels. However neural network classifiers have closer prediction than the SVC.

5. Results/Analysis

MIM-Refiner leverages multiple contrastive heads that are connected to diverse intermediate layers. In each head, a modified nearest neighbor objective helps to construct respective semantic clusters. In this notebook, we have analyzed ImageNet1k fed pretrained *MIM* model with the *CIFAR100* dataset. Image embeddings obtained from *MIM* model are used as an input to train 3 different classification model/networks as following: *Three Layer Network*, *TensorFlow Trainer* and *Support Vector Classifier*.

As mentioned before, within the scope of this study, we could not work with the entire CIFAR100 dataset because at this point, our local computers cannot store the relevant embeddings and even if they could, we do not have the hardware to train the models. Based on this assumption, we tried to analyze the work we've done and mention what can also be done as future work ?

Neural network models (*TF Trainer*) perform not sufficient performance on CIFAR100 *MIM* embeddings with almost *40%* test accuracy. However, SVC performed exteremely poorly on both training and test datasets. It might be the reason that since dimension of each embedding is significantly large and having not sufficient amount of training data.

In terms of performance/memory of the algorithm analysis, *neural network classifiers* are more efficient than the *SVC* in terms of both speed and the memory usage.

References

- https://www.cs.toronto.edu/~kriz/cifar.html
- https://github.com/ml-jku/MIM-Refiner
- https://arxiv.org/abs/2402.10093
- https://paperswithcode.com/sota/self-supervised-image-classification-on
- https://towardsdatascience.com/dino-emerging-properties-in-self-supervised-vision-transformers-summary-ab91df82cc3c
- https://arxiv.org/abs/2010.11929v2
- https://arxiv.org/abs/1706.03762
- https://paperswithcode.com/sota/self-supervised-image-classification-on