Configuration Manual

MSc Research Project
Data Analytics

Taher Abbas Poonawala
Student ID: x21118043

School of Computing
National College of Ireland

Supervisor: Professor Jorge Basilio

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Taher Abbas Poonawala
Student ID:	x21118043
Programme:	Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Professor Jorge Basilio
Submission Due Date:	15/12/2022
Project Title:	Configuration Manual
Word Count:	878
Page Count:	10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Taher Abbas Poonawala
Date:	30th January 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Taher Abbas Poonawala
x21118043

1 Introduction

This document will walk you through the procedure to execute the code, the steps include setting up the AWS cloud infrastructure, creating an EC2 instance that will enable the GPUs with high-end configuration, visual studio code as the IDE, WinSCP to copy the files to and fro the systems. once the IDE is set up the next steps are placing the dataset in the required directory followed by code execution to train and test the models.

2 Amazon Web Services and Amazon Elastic Compute Cloud Setup

Because the code comprises a large volume dataset and conducts a sophisticated computation, it need not only a large number of CPUs but also a high memory connected GPU-enabled hardware. As a result, Amazon Web Services (AWS) and Amazon Elastic Compute Cloud (EC2) are suitable tools to consider. Below are the steps to configure AWS and EC2:

- Navigate to `https://cloud.ncirl.ie/` and click on the highlighted banner to login into the AWS account, as shown in 1.

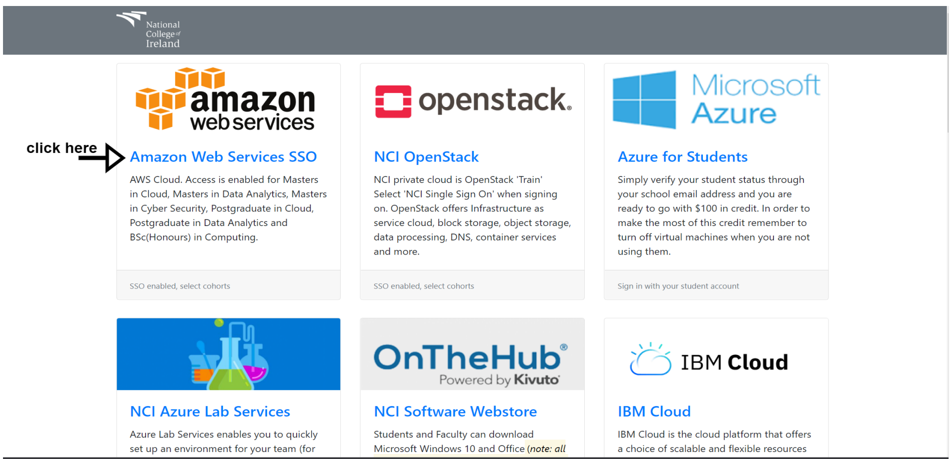


Figure 1: AWS Login

- Once the login is successful navigate to the search bar and type EC2, an EC2 dashboard will open refer to figure 2

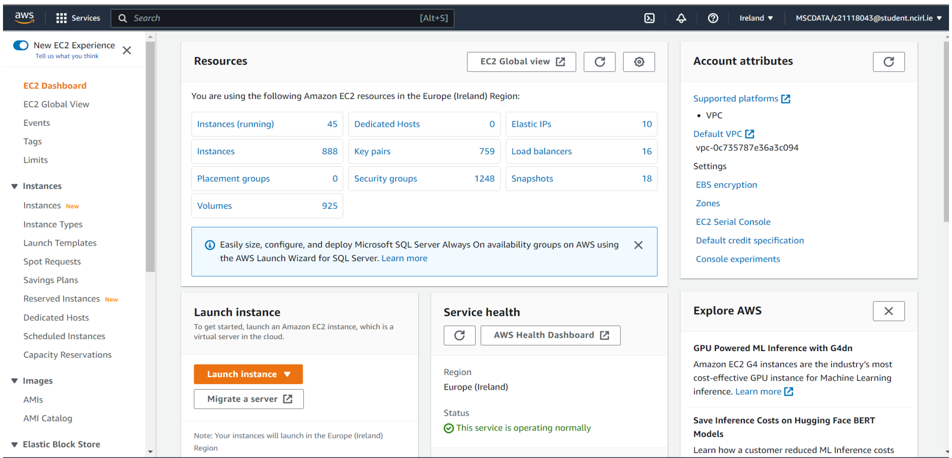


Figure 2: EC2 Dashboard

- Click on "Launch instances" to create an instance, once the launch an instance page opens up enter the required details such as instance name, select Ubuntu as OS image just below that select "Deep Learning AMI" with PyTorch version 1.13.0, select the instance type as shown in figure 3.

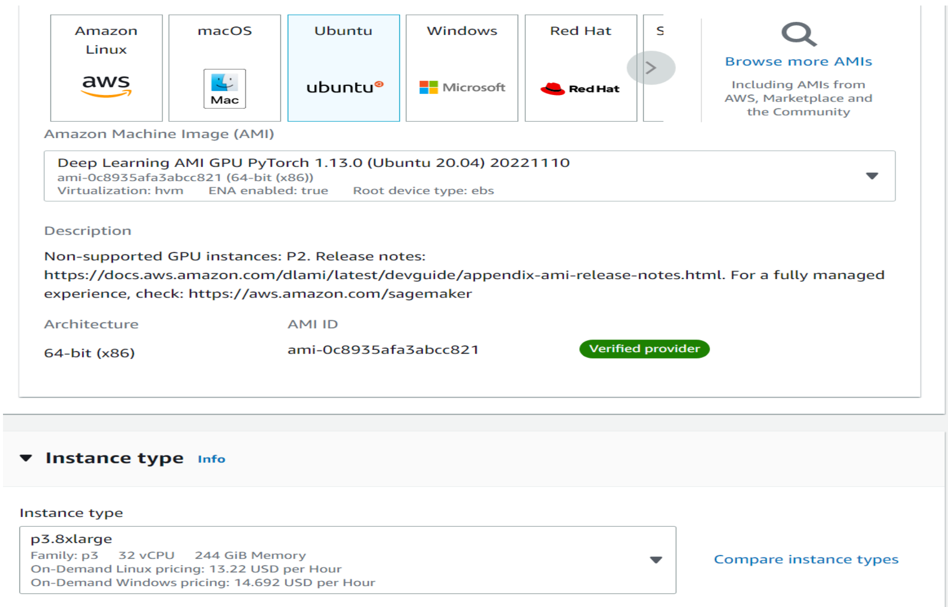


Figure 3: Launch an EC2 Instance

- Create a key pair login by clicking on create new key pair.
- Enter the network details.
- Choose the size of storage as 75 GB, since OS image takes around one-third space, the dataset is 7.7GB in size, and additional libraries will take up some space as well.
- After clicking on launch instance, refer figure 4 page will open up and it will have the details of IPs addresses which are important since it will be login into the instance.

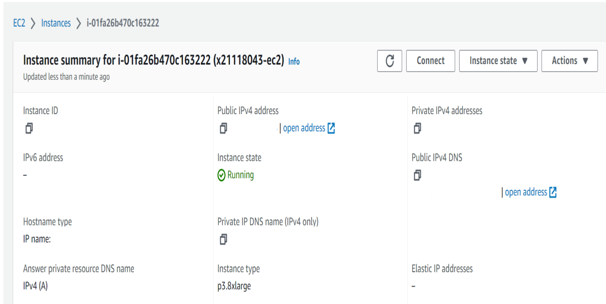


Figure 4: EC2 Instance Dashboard

- Download the vscode from the official vscode page[1] according to the type of OS.
- After installing the vscode click on the extensions, as highlighted in the figure 5 install all those extensions in vscode.

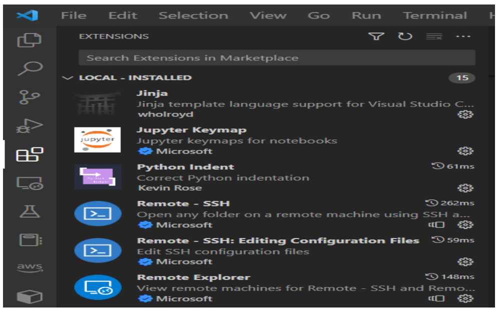


Figure 5: Required Extension in VSCODE

- Once the SSH extension is installed, click on open a remote window from the bottom left corner, click on edit the configuration file and enter details as mentioned in figure 6, save and close it, again click on open a remote window it. Click on to connect to host enter ubuntu and click next. The hostname is Public IPv4 DNS and the identity file is the file downloaded during the keypair generation.

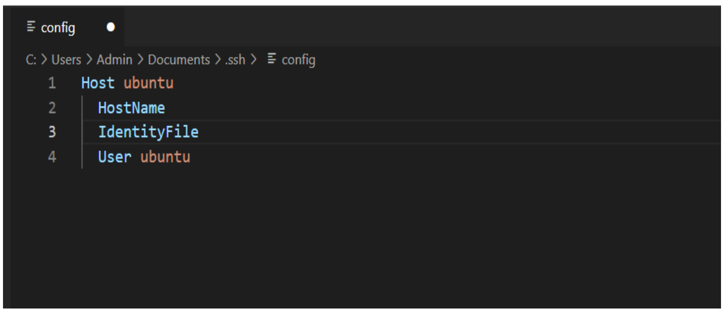


Figure 6: Editing the Configuration File

- Download WinSCP from the official website and install it as per the type of OS. Create a new session by providing the Public IPv4 address from the EC2 instance dashboard page, refer to figure 4. This will help to move the dataset files from the local system to remote location.

3 Python Libraries Requirement

Before execution of the code below libraries are needed to be imported into the system. Also, special care needs to take while installing the torchvision and torch library, to install the foresaid library execute the command- *pip install torch==1.12.1+cu116 torchvision==0.13.1+cu116 torchaudio==0.12.1 –extra-index-url* ***`https://download.pytorch.org/whl/cu116`***. Figure 7 shows the list of libraries used for the execution. Additionally, after installing the torchvision library navigate to the `/home/ubuntu/`

```
## Standard libraries
import os
from copy import deepcopy

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.set()
import h5py as h5
usplit = 'train'
tesplit= 'test'
trsplit= 'val'
import random

## tqdm for loading bars
from tqdm.notebook import tqdm
import numpy

## 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 datasets
from torchvision import transforms

# PyTorch Lightning
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
#confusion matrix
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn import metrics

# Import tensorboard
▶ Launch TensorBoard Session
%load_ext tensorboard

DATASET_PATH = "/home/ubuntu/dataset/."
# Path to the folder where the pretrained models will saved
CHECKPOINT_PATH = "/home/ubuntu/models/."
# In this notebook, we use data loaders with heavier computational processing. It is recommended to use as many
# workers as possible in a data loader, which corresponds to the number of CPU cores
NUM_WORKERS = os.cpu_count()

# Setting the seed
pl.seed_everything(10)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:3") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", device)
print("Number of workers:", NUM_WORKERS)
```

Figure 7: List of Libraries

`.local/lib/python3.8/site-packages/torchvision/datasets/pcam.py` and swap the values test and validate md5 checksum values, this is a known issue exists within the library. After swapping the values the finally the value will look like this, refer to figure 8.

```
_FILES = {
    "train": {
        "images": (
            "camelyonpatch_level_2_split_train_x.h5",  # Data file name
            "1Ka0XfEMiwgCYPdTI-vv6eUElOBnKFKQ2",  # Google Drive ID
            "1571f514728f59376b705fc836ff4b63",  # md5 hash
        ),
        "targets": (
            "camelyonpatch_level_2_split_train_y.h5",
            "1269yhu3pZDP8UYFQs-NYs3FPwuK-nGSG",
            "35c2d7259d906cfc8143347bb8e05be7",
        ),
    },
    "test": {
        "images": (
            "camelyonpatch_level_2_split_test_x.h5",
            "1qV65ZqZvWzuIVthK8eVDhIwrbnsJdbg_",
            "d8c2d60d490dbd479f8199bdfa0cf6ec",
        ),
        "targets": (
            "camelyonpatch_level_2_split_test_y.h5",
            "17BHrSrwWKjYsOgTMmoqrIjDy6Fa2o_gP",
            "60a7035772fbdb7f34eb86d4420cf66a",
        ),
    },
    "val": {
        "images": (
            "camelyonpatch_level_2_split_valid_x.h5",
            "1hgshYGWK8V-eGRy8LToWJJgDU_rXWVJ3",
            "d5b63470df7cfa627aeec8b9dc0c066e",
        ),
        "targets": (
            "camelyonpatch_level_2_split_valid_y.h5",
            "1bH8ZRbhSVAhScTS0p9-ZzGnX91cHT3uO",
            "2b85f58b927af9964a4c15b8f7e8f179",
        ),
    },
}
```

Figure 8: TorchVision library

[1]`https://code.visualstudio.com/download`

4 SimCLR Modeling

4.1 Augmentation Pipelines

The augmentation pipeline is set up into two cells according to the experiments. As per the SimCLR architecture, two different augmented images need to be generated that will be fed into the SimCLR model, this has been implemented in class *TransformationConstrastive* refer figure 9. The list of transformations applied is mentioned in figure 10.

```
class ContrastiveTransformations(object):

    def __init__(self, base_transforms, n_views=2):
        self.base_transforms = base_transforms
        self.n_views = n_views

    def __call__(self, x):
        return [self.base_transforms(x) for i in range(self.n_views)]
✓ 0.9s
```

Figure 9: contrastive Transformation

```
Expirement No-1 without using augmentation pipeline

#TransformationConstrastive = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,), (0.5,))])
✓ 1.6s

Expirement No- 2 Using augmentation pipeline

TransformationConstrastive = transforms.Compose([transforms.RandomHorizontalFlip(),
                                                transforms.RandomResizedCrop(size=90),
                                                transforms.RandomApply([
                                                    transforms.ColorJitter(brightness=0.5,
                                                                           contrast=0.5,
                                                                           saturation=0.5,
                                                                           hue=0.1)
                                                ], p=0.8),
                                                transforms.RandomGrayscale(p=0.2),
                                                transforms.GaussianBlur(kernel_size=9),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.5,), (0.5,))
                                               ])
✓ 0.1s
```

Figure 10: Augmentation pipeline

Figure 11 shows the dataset[2] import and visualization few examples of applying the transformation. Copy the dataset from the local machine to the EC2 instance using WinSCP[3] to the location `/home/ubuntu/dataset/`.

```
# Visualize some examples
pl.seed_everything(10)
NUM_IMAGES = 6
imgs = torch.stack([img for idx in range(NUM_IMAGES) for img in UnlabeledData[idx][0]], dim=0)
img_grid = torchvision.utils.make_grid(imgs, nrow=6, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(10,5))
plt.title('Augmented image examples from the dataset')
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
✓ 0.8s
```

Figure 11: Dataset Visualization

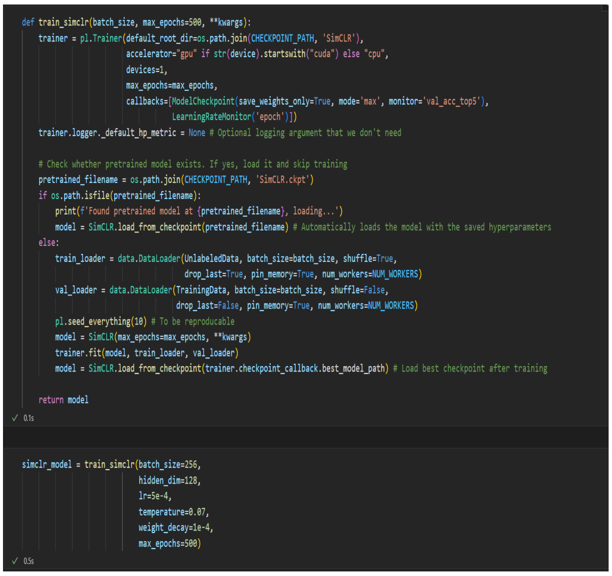
4.2 SimCLR Training

Once the dataset and augmentation pipeline is set up, the next step is to start with the SimCLR implementation. To enhance the readability of the code the modeling is divided into two cells. Figure 12 shows the architecture of the SimCLR model.

```python
class SimCLR(pl.LightningModule):

    def __init__(self, hidden_dim, lr, temperature, weight_decay, max_epochs=500):
        super().__init__()
        self.save_hyperparameters()
        assert self.hparams.temperature > 0.0, 'The temperature must be a positive float!'
        # Base model f(.)
        self.convnet = torchvision.models.resnet18(num_classes=4*hidden_dim)  # Output of last linear layer
        # The MLP for g(.) consists of Linear->ReLU->Linear
        self.convnet.fc = nn.Sequential(
            self.convnet.fc,  # Linear(ResNet output, 4*hidden_dim)
            nn.ReLU(inplace=True),
            nn.Linear(4*hidden_dim, hidden_dim)
        )

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                                lr=self.hparams.lr,
                                weight_decay=self.hparams.weight_decay)
        lr_scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
                                                            T_max=self.hparams.max_epochs,
                                                            eta_min=self.hparams.lr/50)
        return [optimizer], [lr_scheduler]

    def info_nce_loss(self, batch, mode='train'):
        imgs, _ = batch
        imgs = torch.cat(imgs, dim=0)

        # Encode all images
        feats = self.convnet(imgs)
        # Calculate cosine similarity
        cos_sim = F.cosine_similarity(feats[:,None,:], feats[None,:,:], dim=-1)
        # Mask out cosine similarity to itself
        self_mask = torch.eye(cos_sim.shape[0], dtype=torch.bool, device=cos_sim.device)
        cos_sim.masked_fill_(self_mask, -9e15)
        # Find positive example -> batch_size//2 away from the original example
        pos_mask = self_mask.roll(shifts=cos_sim.shape[0]//2, dims=0)
        # InfoNCE loss
        cos_sim = cos_sim / self.hparams.temperature
        nll = -cos_sim[pos_mask] + torch.logsumexp(cos_sim, dim=-1)
        nll = nll.mean()
        # Logging loss
        self.log(mode+'_loss', nll)
        # Get ranking position of positive example
        comb_sim = torch.cat([cos_sim[pos_mask][:,None],  # First position positive example
                              cos_sim.masked_fill(pos_mask, -9e15)],
                             dim=-1)
        sim_argsort = comb_sim.argsort(dim=-1, descending=True).argmin(dim=-1)
        # Logging ranking metrics
        self.log(mode+'_acc_top1', (sim_argsort == 0).float().mean())
        self.log(mode+'_acc_top5', (sim_argsort < 5).float().mean())
        self.log(mode+'_acc_mean_pos', 1+sim_argsort.float().mean())

        return nll

    def training_step(self, batch, batch_idx):
        return self.info_nce_loss(batch, mode='train')

    def validation_step(self, batch, batch_idx):
        self.info_nce_loss(batch, mode='val')
```

Figure 12: SimCLR Architecture

Finally figure 13 shows the SimCLR training function definition and calls the SimCLR training function with batch size as 256, learning rate as 5e^{-4}, the temperature is set as 0.07, max_epoch is 500.



```python
def train_simclr(batch_size, max_epochs=500, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, 'SimCLR'),
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=max_epochs,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode='max', monitor='val_acc_top5'),
                                    LearningRateMonitor('epoch')])
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, 'SimCLR.ckpt')
    if os.path.isfile(pretrained_filename):
        print(f'Found pretrained model at {pretrained_filename}, loading...')
        model = SimCLR.load_from_checkpoint(pretrained_filename) # Automatically loads the model with the saved hyperparameters
    else:
        train_loader = data.DataLoader(UnlabeledData, batch_size=batch_size, shuffle=True,
                                       drop_last=True, pin_memory=True, num_workers=NUM_WORKERS)
        val_loader = data.DataLoader(TrainingData, batch_size=batch_size, shuffle=False,
                                     drop_last=False, pin_memory=True, num_workers=NUM_WORKERS)
        pl.seed_everything(10) # To be reproducable
        model = SimCLR(max_epochs=max_epochs, **kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = SimCLR.load_from_checkpoint(trainer.checkpoint_callback.best_model_path) # Load best checkpoint after training

    return model
```

```python
simclr_model = train_simclr(batch_size=256,
                            hidden_dim=128,
                            lr=5e-4,
                            temperature=0.07,
                            weight_decay=1e-4,
                            max_epochs=500)
```

Figure 13: SimCLR Training

[3]`https://zenodo.org/record/2546921#.Y5nN2XbP07E`
[3]`https://winscp.net/eng/download.php`

4.3 Launching the TensorBoard to View SimCLR Accuracy and Loss

To understand the SimCLR model behavior TensorBoard functionality is implemented to view the accuracy and loss over the datasets, refer to figure 14.

```
▶ Launch TensorBoard Session
%tensorboard --logdir /home/ubuntu/saved_models/.
✓ 9.1s
```

Figure 14: Tensor Board

5 Classification: Logistic Regression

After training the SimCLR model and saving the checkpoint also known as model weight. The next step is to implement a single-layer classifier, for this task Logistic Regression(LR) is been implemented in the class *LogisticRegression*, refer to figure 15.

```python
class LogisticRegression(pl.LightningModule):

    def __init__(self, feature_dim, num_classes, lr, weight_decay, max_epochs=100):
        super().__init__()
        self.save_hyperparameters()
        # Mapping from representation h to classes
        self.model = nn.Linear(feature_dim, num_classes)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                                lr=self.hparams.lr,
                                weight_decay=self.hparams.weight_decay)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                      milestones=[int(self.hparams.max_epochs*0.6),
                                                                  int(self.hparams.max_epochs*0.8)],
                                                      gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode='train'):
        feats, labels = batch
        preds = self.model(feats)
        true_y, pred_y = [], []
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()
        self.log(mode + '_loss', loss)
        self.log(mode + '_acc', acc)
        return loss

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

    def training_step(self, batch, batch_idx):
        return self._calculate_loss(batch, mode='train')

    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')

    def evaluate(data_loader, module):
        with torch.no_grad():
            progress = ["/", "-", "\\", "|", "/", "-", "\\", "|"]
            module.eval().cuda()
            true_y, pred_y = [], []
            for i, batch_ in enumerate(data_loader):
                X, y = batch_
                print(progress[i % len(progress)], end="\r")
                y_pred = torch.argmax(module(X.cuda()), dim=1)
                true_y.extend(y.cpu())
                pred_y.extend(y_pred.cpu())
            confusion_matrix = metrics.confusion_matrix(true_y, pred_y)
            cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
            cm_display.plot()
            plt.show()
            print(classification_report(true_y, pred_y, digits=3))
            return true_y, pred_y
✓ 5.5s
```

Figure 15: Class Logistic Regression

5.1 Dataset Encoding

After implementing the logistic regression class, the next task is to encode the images from the train and test images which will be the input classifier, refer figure 16 and 17.

```python
img_transforms = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize((0.5,), (0.5,))])

train_img_data = datasets.PCAM(root=DATASET_PATH, split=trsplit, download=True,
                               transform=img_transforms)
test_img_data = datasets.PCAM(root=DATASET_PATH, split=tesplit, download=True,
                              transform=img_transforms)

print("Number of val examples:", len(train_img_data))
print("Number of test examples:", len(test_img_data))
✓ 1.9s
```

Figure 16: Training and Test Data Encoding

```python
@torch.no_grad()
def prepare_data_features(model, dataset):
    # Prepare model
    network = deepcopy(model.convnet)
    network.fc = nn.Identity()  # Removing projection head g(.)
    network.eval()
    network.to(device)

    # Encode all images
    data_loader = data.DataLoader(dataset, batch_size=64, num_workers=NUM_WORKERS, shuffle=False, drop_last=False)
    feats, labels = [], []
    for batch_imgs, batch_labels in tqdm(data_loader):
        batch_imgs = batch_imgs.to(device)
        batch_feats = network(batch_imgs)
        feats.append(batch_feats.detach().cpu())
        labels.append(batch_labels)

    feats = torch.cat(feats, dim=0)
    labels = torch.cat(labels, dim=0)

    # Sort images by labels
    labels, idxs = labels.sort()
    feats = feats[idxs]

    return data.TensorDataset(feats, labels)
```
✓ 0.1s

```python
train_feats_simclr = prepare_data_features(simclr_model, train_img_data)
test_feats_simclr = prepare_data_features(simclr_model, test_img_data)
```
✓ 22.1s

Figure 17: Encoding Function

5.2 Training Logistic Regression

The next step is training the logistic regression, refer to figure 18 and evaluating the results over a different number of images per label, here 10, 20, 50, 100, 200, 500 images per label are used also the batch_size is set as 64, max_epoch is 100 and learning_rate is 1e^{-3}.

```python
def train_logreg(batch_size, train_feats_data, test_feats_data, model_suffix, max_epochs=100, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "LogisticRegression"),
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=max_epochs,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode='max', monitor='val_acc'),
                                    LearningRateMonitor("epoch")],
                         enable_progress_bar=False,
                         check_val_every_n_epoch=10)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_feats_data, batch_size=batch_size, shuffle=True,
                                   drop_last=False, pin_memory=True, num_workers=0)
    test_loader = data.DataLoader(test_feats_data, batch_size=batch_size, shuffle=False,
                                  drop_last=False, pin_memory=True, num_workers=0)

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"LogisticRegression_{model_suffix}.ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = LogisticRegression.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)  # To be reproducable
        model = LogisticRegression(**kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = LogisticRegression.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    # Test best model on train and validation set
    train_result = trainer.test(model, train_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    _ = LogisticRegression.evaluate(test_loader, model)
    result = {"train": train_result[0]["test_acc"], "test": test_result[0]["test_acc"], "test_mape":test_result[0]["test_loss"]}
    return model, result
```
✓ 0.1s

Figure 18: Training Logistic Regression

5.3 Logistic Regression Results

Figure 19 cell shows the graph of accuracy versus the number of images and MAPE values, additionally, the confusion matrix with the model classification report is also implemented. Also, the model is evaluated after every 10 epochs so that the model does not overfit the dataset refer to figure 20.

```python
def get_smaller_dataset(original_dataset, num_imgs_per_label):
    new_dataset = data.TensorDataset(
        *[t.unflatten(0, (2, -1))[:,:num_imgs_per_label].flatten(0, 1) for t in original_dataset.tensors]
    )
    return new_dataset
✓ 0.1s
```

```python
results = {}
for num_imgs_per_label in [10, 20, 50, 100, 200, 500]:
    sub_train_set = get_smaller_dataset(train_feats_simclr, num_imgs_per_label)
    _, small_set_results = train_logreg(batch_size=64,
                                        train_feats_data=sub_train_set,
                                        test_feats_data=test_feats_simclr,
                                        model_suffix=num_imgs_per_label,
                                        feature_dim=train_feats_simclr.tensors[0].shape[1],
                                        num_classes=2,
                                        lr=1e-3,
                                        weight_decay=1e-3)
    results[num_imgs_per_label] = small_set_results
✓ 1m 31.1s
```

Figure 19: Calling the LR Train function

```python
dataset_sizes = sorted([k for k in results])
test_scores = [results[k]["test"] for k in dataset_sizes]
test_mape = [results[k]["test_mape"] for k in dataset_sizes]

fig = plt.figure(figsize=(6,4))
plt.plot(dataset_sizes, test_scores, '--', color="#000", marker="*", markeredgecolor="#000", markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(dataset_sizes, labels=dataset_sizes)
plt.title("Image classification over dataset size", fontsize=14)
plt.xlabel("Number of images per class")
plt.ylabel("Test accuracy")
plt.minorticks_off()
plt.show()

fig = plt.figure(figsize=(6,4))
plt.plot(dataset_sizes, test_mape, '-', color="#000", marker="*", markeredgecolor="#000", markerfacecolor="y", markersize=16)
plt.xscale("log")
plt.xticks(dataset_sizes, labels=dataset_sizes)
plt.title("Image classification over dataset size", fontsize=14)
plt.xlabel("Number of images per class")
plt.ylabel("MAPE")
plt.minorticks_off()
plt.show()

for k, score in zip(dataset_sizes, test_scores):
    print(f'Test accuracy for {k:3d} images per label: {100*score:4.2f}%')

for k,mape in zip(dataset_sizes,test_mape):
    print(f'MAPE for {k:3d} images per label: {mape:4.2f}%')
✓ 0.9s
```

Figure 20: Results

6 Baseline: ResNet18

Once the SimCLR based on self-supervised contrastive learning and logistic regression are implemented, the next step is to implement the baseline model, hence ResNet18 as a fully supervised learning is used. Figure 21 shows the implementation of the ResNet18 under class *ResNet*.

```python
class ResNet(pl.LightningModule):

    def __init__(self, num_classes, lr, weight_decay, max_epochs=100):
        super().__init__()
        self.save_hyperparameters()
        self.model = torchvision.models.resnet18(num_classes=num_classes)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
                                lr=self.hparams.lr,
                                weight_decay=self.hparams.weight_decay)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
                                                      milestones=[int(self.hparams.max_epochs*0.7),
                                                                  int(self.hparams.max_epochs*0.9)],
                                                      gamma=0.1)
        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()

        self.log(mode + '_loss', loss)
        self.log(mode + '_acc', acc)
        return loss

    def training_step(self, batch, batch_idx):
        return self._calculate_loss(batch, mode='train')

    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')

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

    def evaluate(data_loader, module):
        with torch.no_grad():
            progress = ["/", "-", "\\", "|", "/", "-", "\\", "|"]
            module.eval().cuda()
            true_y, pred_y = [], []
            for i, batch_ in enumerate(data_loader):
                X, y = batch_
                print(progress[i % len(progress)], end="\r")
                y_pred = torch.argmax(module(X.cuda()), dim=1)
                true_y.extend(y.cpu())
                pred_y.extend(y_pred.cpu())
            confusion_matrix = metrics.confusion_matrix(true_y, pred_y)
            cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])
            cm_display.plot()
            plt.show()
            print(classification_report(true_y, pred_y, digits=3))
            return true_y, pred_y
✓ 0.1s
```

Figure 21: ResNet18 Class

6.1 Augmentation Pipeline

To evaluate the result on a fair basis the augmentation pipeline is implemented for the baseline model as well, and it has been sectioned as per the experiments, refer figure 22.

Expirement No-1 without using augmentation pipeline

```
#train_transforms = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

#train_img_aug_data = datasets.PCAM(root=DATASET_PATH, split=trsplit, download=True,transform=train_transforms)
```

Expirement No-2 using augmentation pipeline

```
train_transforms = transforms.Compose([transforms.RandomHorizontalFlip(),
                                       transforms.RandomResizedCrop(size=96, scale=(0.8, 1.0)),
                                       transforms.RandomGrayscale(p=0.2),
                                       transforms.GaussianBlur(kernel_size=9, sigma=(0.1, 0.5)),
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5,), (0.5,))
                                       ])

train_img_aug_data = datasets.PCAM(root=DATASET_PATH, split=trsplit, download=True,
                                   transform=train_transforms)
```

Figure 22: Image Augmentations

6.2 Training ResNet18

The training function is similar to the logistic regression and validation is performed after every 2 epochs so that the model does not overfit during the training step refer figure 23.

```
def train_resnet(batch_size, max_epochs=100, **kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ResNet"),
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=max_epochs,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_acc"),
                                    LearningRateMonitor("epoch")],
                         check_val_every_n_epoch=2)
    trainer.logger._default_hp_metric = None

    # Data loaders
    train_loader = data.DataLoader(train_img_aug_data, batch_size=batch_size, shuffle=True,
                                   drop_last=True, pin_memory=True, num_workers=NUM_WORKERS)
    test_loader = data.DataLoader(test_img_data, batch_size=batch_size, shuffle=False,
                                  drop_last=False, pin_memory=True, num_workers=NUM_WORKERS)

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH,'ResNet.ckpt')
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model at %s, loading..." % pretrained_filename)
        model = ResNet.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(10) # To be reproducable
        model = ResNet(**kwargs)
        trainer.fit(model, train_loader, test_loader)
        model = ResNet.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    # Test best model on validation set
    train_result = trainer.test(model, train_loader, verbose=False)
    val_result = trainer.test(model, test_loader, verbose=False)
    _ = ResNet.evaluate(test_loader, model)
    result = {"train": train_result[0]["test_acc"], "test": val_result[0]["test_acc"],"resnet_test_mape":val_result[0]["test_loss"]}

    return model, result
✓ 0.9s
```

Figure 23: Training Function

6.3 Model Training and Result

The last cell calls the training function and the confusion matrix, classification report, and accuracy of the model are printed, refer figure 24.

```
resnet_model, resnet_result = train_resnet(batch_size=64,
                                           num_classes=2,
                                           lr=1e-3,
                                           weight_decay=2e-4,
                                           max_epochs=100)
print(f"Accuracy on training set: {100*resnet_result['train']:4.2f}%")
print(f"Accuracy on test set: {100*resnet_result['test']:4.2f}%")
print(f"MAPE of Baseline model:{resnet_result['resnet_test_mape']:4.2f}%")
✓ 34.2s
```

Figure 24: Baseline Results