AlexNet_Implementation

Summary

This is an implementation of AlexNet utilizing the CIFAR-100 dataset. AlexNet was originally trained on ImageNet, which has millions of images, so training it on CIFAR-100s dataset, which contains 50,000 training images) is going to yield substandard results. However, the goal is to gain a deeper understanding of CNNs, not to win an image-recognition contest.

Implementation Overview

Imports

Import all the relevant PyTorch and matplotlib/numpy/sklearn libraries in addition to make_grid which will be used to visualize some of the data, and tqdm for tracking progress.

```
# Imports
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms
from torchvision.utils import make_grid
import tqdm
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import f1_score
```

Data Preparation

Download the datasets for training and testing. Various transforms will also be applied. In the original AlexNet paper, it seems they did not normalize or do any pre-processing of the images other than

cropping them down:
"Given a rectangular image, we first rescaled the image such that the shorter side was of length 256, and then cropped out the central 256×256 patch from the resulting image. We did not pre-process the images in any other way".

This code has the final transforms tested. See the outputs below for variations on transforms and the resulting outputs.

Various tested things:

- Randomly get a crop after oversizing. i.e resize image to 230 and then grab a 227 crop
- Random flips. This could also be done with rotations.
- Variations on the values for normalizing.
- Various batch sizes (32, 64, 128). The original authors used a batch size of 128.

```
# Download the data

# Will need to resize 28x28 LeNet and 227x227 for AlexNet
transform_train = transforms.Compose([
    transforms.Resize(230), # Get a random crop
    transforms.RandomHorizontalFlip(),
    transforms.Resize(227)
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.Resize(230),
    transforms.Resize(227),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

training_data = datasets.CIFAR100(
    root = 'data', #directory to store the dataset in
    # True for training dataset, False for testing
    train = True,
    download = True,
    transform = transform_train
)

testing_data = datasets.CIFAR100(
    root = 'data',
```

```
    train = False,
    download = True,
    transform = transform_test
)
classes = training_data.classes

# DataLoader pytorch.org/docs/stable/data.html
batch_size = 128
training_dataloader = DataLoader(training_data, batch_size=batch_size)
testing_dataloader = DataLoader(testing_data, batch_size=batch_size)

# For CIFAR-100 we expect 60k total images
# 100 classes with 600 images each
# that's 50k training and 10k testing (500 training/100 testing per class)
print(f'training: {batch_size*len(training_dataloader)}, testing:
{batch_size*len(testing_dataloader)}')
```

Visualize Data

Print some images and the associated labels to manually verify veracity. `img = img / 2 + 0.5` helps counter some of the normalization, making it easier for human eyes to discriminate.

```
# Peek at the data
# manually verify labels are correct:
# https://huggingface.co/datasets/cifar100

def imshow(img):
  # img = img / 2 + 0.5
  npimg = img.numpy()
  plt.figure(figsize=(15, 15))
  plt.imshow(np.transpose(npimg, (1, 2, 0)))
  plt.show()

dataiter = iter(training_dataloader)
images, labels = next(dataiter)
images = images[:5]
labels = labels[:5]

imshow(make_grid(images))
print('Ordered Labels:')
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(5)))
```

Output:

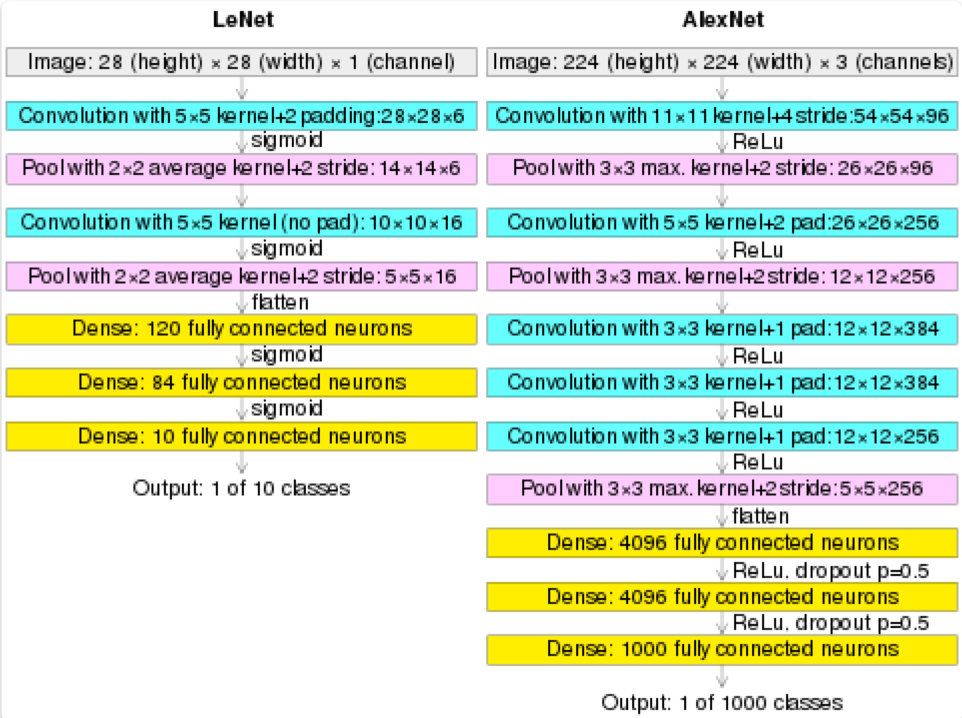
```

```
Ordered Labels:
cattle dinosaur apple boy   aquarium_fish
```

# Training and Testing

This code encapsulates essential functionalities for training and evaluating the model. The train function is responsible for iteratively updating the model parameters based on the provided training dataset. It operates in a loop over mini-batches, computing predictions, calculating the loss between predictions and ground truth labels, backpropagating the loss, and updating the model's parameters accordingly. Additionally, it includes an optional progress check for monitoring the training process. On the other hand, the test function evaluates the trained model's performance on a separate testing dataset. It calculates the average loss and accuracy over all batches and computes the F1 score to assess the model's overall performance.

```
Define Training/Testing

# train
def train(dataloader, model, loss_fn, optimizer, device):
  model.train()
  # step being a training step
  for step, (X, y) in enumerate(dataloader):
    # send to CPU/GPU
    X = X.to(device)
    y = y.to(device)
    # model's prediction
    pred = model(X)
    # get loss
    loss = loss_fn(pred, y)
    # backprop
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```
```

```
    # progress check
    if step % 100 == 0:
      print(f'step: {step}, loss: {loss.item()}')

# validation
def test(dataloader, model, loss_fn, device):
  num_steps = len(dataloader)
  model.eval()
  test_loss = 0
  correct = 0
  y_true = []
  y_predicted = []

  with torch.no_grad():
    for X, y in dataloader:
      X = X.to(device)
      y = y.to(device)
      pred = model(X)
      loss = loss_fn(pred, y)
      test_loss += loss.item()
      y_true.extend(y.cpu().numpy())
      y_hat = pred.argmax(1)
      y_predicted.extend(y_hat.cpu().numpy())
      correct_step = (y_hat == y).type(torch.float).sum().item()
      correct += correct_step

  test_loss /= num_steps # average loss
  correct = correct / (num_steps * batch_size)
  f1 = f1_score(y_true, y_predicted, average='macro')

  print()
  print(f'F1 Score: {f1}')
  print(f'Test Accuracy: {correct}')
```

Define Model

AlexNet is implemented using the following image as a guide:



AlexNet consists of five convolutional layers followed by three fully connected layers. Each convolutional layer is followed by a rectified linear unit (ReLU) activation function to introduce non-linearity. The network architecture comprises alternating convolutional and max-pooling layers to extract hierarchical features from input images. Batch normalization is applied after the first convolutional layer to accelerate training. Dropout regularization is also utilized in the fully connected layers to prevent overfitting. The network's output is the logits representing the predicted class probabilities for the input image. The `forward` method defines the flow of data through the network layers, sequentially passing input through convolutional, activation, pooling, and fully connected layers, ultimately producing the final logits.

GPU compute is limited without paying, so BatchNorm2d() was added in the hope it would speed up training when running on CPU. There are also print statements for debugging.

```
# AlexNet

class AlexNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = nn.Sequential(
      nn.Conv2d(in_channels=3,
                out_channels=96,
                kernel_size=11,
                stride=4,
                padding=0),
      nn.BatchNorm2d(96), # Was hoping this might speed up training
```

```
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3,
                     stride=2)
    )
      self.conv2 = nn.Sequential(
        nn.Conv2d(in_channels=96,
                  out_channels=256,
                  kernel_size=5,
                  stride=1,
                  padding=2),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3,
                     stride=2)
    )
      self.conv3 = nn.Sequential(
        nn.Conv2d(in_channels=256,
                  out_channels=384,
                  kernel_size=3,
                  stride=1,
                  padding=1),
        nn.ReLU()
    )
      self.conv4 = nn.Sequential(
        nn.Conv2d(in_channels=384,
                  out_channels=384,
                  kernel_size=3,
                  stride=1,
                  padding=1),
        nn.ReLU()
    )
      self.conv5 = nn.Sequential(
        nn.Conv2d(in_channels=384,
                  out_channels=256,
                  kernel_size=3,
                  stride=1,
                  padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3,
                     stride=2)
    )
      self.flatten = nn.Flatten()
      self.fc1 = nn.Sequential(
          nn.Linear(9216, 4096),
```

```
            nn.ReLU(),
            nn.Dropout(0.5)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5)
        )
        self.fc3 = nn.Sequential(
            nn.Linear(4096, 100), # 100 classes in CIFAR-100
            nn.ReLU()
        )

    def forward(self, x):
        x = self.conv1(x)
        # print(f'size after conv1: {x.size()}')
        x = self.conv2(x)
        # print(f'size after conv2: {x.size()}')
        x = self.conv3(x)
        # print(f'size after conv3: {x.size()}')
        x = self.conv4(x)
        # print(f'size after conv4: {x.size()}')
        x = self.conv5(x)
        # print(f'size after conv5: {x.size()}')
        x = x.reshape(x.size(0), -1)
        # print(f'size after reshape: {x.size()}')
        x = self.fc1(x)
        # print(f'size after fc1: {x.size()}')
        x = self.fc2(x)
        # print(f'size after fc2: {x.size()}')
        logits = self.fc3(x)
        return logits
```

Running the Model

Simply prints some information about the configuration and then runs training/testing.

```
# get device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
# create model
model = AlexNet().to(device)
epochs = 10
```

```
lr = 5e-3
# optimize
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

print(f'Using {device}')
print(f'Optimizer: {optimizer}')
print(f'Epochs {epochs}')
print(f'Batch Size: {batch_size}')
print(f'Model Info: {AlexNet}')

for t in tqdm.tqdm(range(epochs)):
  print(f'Epoch\n\n{t}')
  train(training_dataloader, model, loss_fn, optimizer, device)
  test(testing_dataloader, model, loss_fn, device)
print('Done!')
```

Outputs

I will not be able to efficiently test the hyperparameters due to time and GPU constraints. Instead of programmatically iterating over each possible combination I am simply going to randomly test different things I feel like playing with.

- AlexNet 1
  - No additional transforms/normalization
  - Batch size 32
  - Learn Rate 0.005
  - Optimizer SGD
  - Epochs 10
  - *F1* 0.245
  - *Accuracy* 0.255
- AlexNet 2
  - Random center crop (image to 230x230 to 227x227)
  - Batch size 64
  - Learn Rate 0.005
  - Optimizer SGD
  - Epochs 10
  - *F1* 0.149
  - *Accuracy* 0.181
- AlexNet 3
  - Random center crop (image to 230x230 to 227x227)

    - Normalization
    - Batch size 64
    - Learn Rate 0.001
    - Optimizer SGD
    - Epochs 10
    - *F1* 0.003
    - *Accuracy* 0.021
- AlexNet 4
    - Random center crop (image to 230x230 to 227x227)
    - Normalization
    - Random Horizontal Flip
    - Batch size 128
    - Learn Rate 0.005
    - Optimizer SGD
    - Epochs 10
    - *F1* 0.017
    - *Accuracy* 0.048
- AlexNet 5
  - Random Crop
  - Random Horizontal Flip
  - Batch size 64
  - Learn Rate 0.005
  - Optimizer SGD
  - Epochs 10
  - *F1* 0.128
  - *Accuracy* 0.159
  AlexNet 6
  - Random Crop
  - Random Horizontal Flip
  - Batch size 32
  - Learn Rate 0.005
  - Optimizer SGD
  - Epochs 10
  - *F1* 0.276
  - *Accuracy* 0.288
  AlexNet 7
  - Random Crop
  - Random Horizontal Flip
  - Normalization
  - Batch size 32
  - Learn Rate 0.005

- Optimizer SGD
- Epochs 20
- *F1* 0.473
- *Accuracy* 0.482

AlexNet 8

- Random Crop
- Random Horizontal Flip
- Normalization
- Batch size 32
- Learn Rate 0.005
- Optimizer ADAM
- Epochs 2
- Stopped early. See output

AlexNet 1

No normalization or other augmentation at this stage.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.005
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Epochs 10
Batch Size: 32
Model Info: <class '__main__.AlexNet'>
```

In the first epoch it seems to be overshooting the minima:

```
step: 0, loss: 4.604136943817139
step: 100, loss: 4.605429172515869
step: 200, loss: 4.603725910186768
step: 300, loss: 4.602735996246338
step: 400, loss: 4.606757164001465
step: 500, loss: 4.603265762329102
step: 600, loss: 4.605870723724365
```

```
step: 700, loss: 4.605766296386719
step: 800, loss: 4.606579780578613
step: 900, loss: 4.606809139251709
step: 1000, loss: 4.604169845581055
step: 1100, loss: 4.605472564697266
step: 1200, loss: 4.604623317718506
step: 1300, loss: 4.605377674102783
step: 1400, loss: 4.606267929077148
step: 1500, loss: 4.605053901672363
  5%|▌         | 1/20 [01:39<31:32, 99.60s/it]
F1 Score: 0.0036451901262634172
Test Accuracy: 0.017671725239616614
```

Though it improves over time there is still the issue of overshooting the minima. With dropout there is some confidence the model is learning rather than simply overfitting.

```
Epoch 4
step: 0, loss: 4.142165184020996
step: 100, loss: 3.9160821437835693
step: 200, loss: 3.973121166229248
step: 300, loss: 3.8272202014923096
step: 400, loss: 3.9439640045166016
step: 500, loss: 4.291998863220215
step: 600, loss: 3.522207260131836
step: 700, loss: 4.111637115478516
step: 800, loss: 3.718421459197998
step: 900, loss: 3.9666943550109863
step: 1000, loss: 3.9658987522125244
step: 1100, loss: 3.9765686988830566
step: 1200, loss: 4.051502227783203
step: 1300, loss: 3.8093719482421875
step: 1400, loss: 3.599269390106201
step: 1500, loss: 3.7137014865875244
 25%|██▌       | 5/20 [08:07<24:20, 97.34s/it]
F1 Score: 0.09901280992137369
Test Accuracy: 0.1391773162939297
```

AlexNet 2

With random cropping and increased batch size of 64. Ideally random cropping will help prevent overfitting and increase the feature recognition.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.005
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Epochs 10
Batch Size: 64
Model Info: <class '__main__.AlexNet'>
```

Out:

```
Epoch 9
step: 0, loss: 3.8393609523773193
step: 100, loss: 3.5322840213775635
step: 200, loss: 3.3416495323181152
step: 300, loss: 3.2470955848693848
step: 400, loss: 3.444653034210205
step: 500, loss: 3.4551656246185303
step: 600, loss: 3.584714412689209
step: 700, loss: 3.433584213256836
100%|██████████| 10/10 [29:09<00:00, 174.96s/it]
F1 Score: 0.14870911190840477
Test Accuracy: 0.18132961783439 49
Done!
```

AlexNet 3

With random cropping, increased batch size of 64, and normalization. Also dropped the learning rate to 1e-3 from 5e-3 to help with overshooting minima.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
```

```
    foreach: None
    lr: 0.001
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Epochs 10
Batch Size: 64
Model Info: <class '__main__.AlexNet'>
```

Out:

```
Epoch 9
step: 0, loss: 4.604341506958008
step: 100, loss: 4.604414463043213
step: 200, loss: 4.602235794067383
step: 300, loss: 4.60159969329834
step: 400, loss: 4.603744983673096
step: 500, loss: 4.605149745941162
step: 600, loss: 4.6016364097595215
step: 700, loss: 4.603060245513916
100%|██████████| 10/10 [31:41<00:00, 190.19s/it]
F1 Score: 0.003407055583136615
Test Accuracy: 0.02119824840764331
Done!
```

Clearly this one is bad. The learning rate will be adjusted back to 0.005.

AlexNet 4

With random cropping, increased batch size of 128, and normalization. Returned the learning rate to 0.005. Added RandomHorizontalFlip() to the training data.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.005
    maximize: False
    momentum: 0
```

```
    nesterov: False
    weight_decay: 0
)
Epochs 10
Batch Size: 128
Model Info: <class '__main__.AlexNet'>
```

Out:

```
Epoch 9
step: 0, loss: 4.485115051269531
step: 100, loss: 4.381525039672852
step: 200, loss: 4.291108131408691
step: 300, loss: 4.329562664031982
100%|██████████| 10/10 [31:53<00:00, 191.37s/it]
F1 Score: 0.01732490333417422
Test Accuracy: 0.0479628164556962
Done!
```

Still bad, I will get rid of random cropping and normalization but leave in RandomHorizontalFlip() as the original authors used a similar augmentation during training.

AlexNet 5

Back to a simpler set of transforms.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.005
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Epochs 10
Batch Size: 64
Model Info: <class '__main__.AlexNet'>
```

Out:

```
Epoch 9
step: 0, loss: 3.8122899532318115
step: 100, loss: 3.6361472606658936
step: 200, loss: 3.4220173358917236
step: 300, loss: 3.3954293727874756
step: 400, loss: 3.498685121536255
step: 500, loss: 3.3461503982543945
step: 600, loss: 3.706328868865967
step: 700, loss: 3.482452869415283
100%|██████████| 10/10 [27:13<00:00, 163.40s/it]
F1 Score: 0.128266877455268
Test Accuracy: 0.15943471337579618
Done!
```

It seems having `batch_size = 32` is the best option overall. I want to try the AlexNet 1 parameters but with random center crop and random horizontal flip.

AlexNet 6

With `batch_size = 32`, random cropping and random horizontal flip.

```
Epoch 9
step: 0, loss: 3.5397918224334717
step: 100, loss: 3.267432928085327
step: 200, loss: 3.1370394229888916
step: 300, loss: 3.0607476234436035
step: 400, loss: 2.7535886764526367
step: 500, loss: 3.1194634437561035
step: 600, loss: 2.7183964252471924
step: 700, loss: 3.4080727100372314
step: 800, loss: 3.0442261695861816
step: 900, loss: 3.068878412246704
step: 1000, loss: 2.8053441047668457
step: 1100, loss: 2.815408945083618
step: 1200, loss: 2.8268918991088867
step: 1300, loss: 2.900090217590332
step: 1400, loss: 2.6602871417999268
step: 1500, loss: 2.9289915561676025
100%|██████████| 10/10 [24:40<00:00, 148.09s/it]
F1 Score: 0.2757321955664497
```

```
Test Accuracy: 0.2878394568690096
Done!
```

AlexNet 7

Lastly, I ran one with the AlexNet 6 parameters but added normalization and increased the epochs to 20.

```
Using cuda
Optimizer: SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.005
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Epochs 20
Batch Size: 32
Model Info: <class '__main__.AlexNet'>
```

Similar results as AlexNet 6 after 10 epochs:

```
Epoch 9
step: 0, loss: 3.680598258972168
step: 100, loss: 3.2466609477996826
step: 200, loss: 3.193988561630249
step: 300, loss: 3.1848771572113037
step: 400, loss: 2.725031614303589
step: 500, loss: 3.1518399715423584
step: 600, loss: 2.868248462677002
step: 700, loss: 3.350416421890259
step: 800, loss: 2.896730899810791
step: 900, loss: 3.1338818073272705
step: 1000, loss: 2.8393263816833496
step: 1100, loss: 2.858152151107788
step: 1200, loss: 2.933701992034912
step: 1300, loss: 2.994683027267456
step: 1400, loss: 2.605807065963745
```

```
step: 1500, loss: 2.8658103942871094

 50%|█████     | 10/20 [28:00<27:43, 166.30s/it]

F1 Score: 0.27519449784215344
Test Accuracy: 0.2918330670926518
```

At 20 epochs:

```
Epoch

19
step: 0, loss: 2.309366226196289
step: 100, loss: 2.0782630443573
step: 200, loss: 1.9713388681411743
step: 300, loss: 1.9979872703552246
step: 400, loss: 1.617879033088684
step: 500, loss: 1.871593952178955
step: 600, loss: 1.7235015630722046
step: 700, loss: 1.9068670272827148
step: 800, loss: 2.2559616565704346
step: 900, loss: 1.5877236127853394
step: 1000, loss: 1.4503012895584106
step: 1100, loss: 2.006448268890381
step: 1200, loss: 1.7350914478302002
step: 1300, loss: 1.9358537197113037
step: 1400, loss: 1.423730492591858
step: 1500, loss: 1.907639980316162
100%|██████████| 20/20 [55:38<00:00, 166.94s/it]
F1 Score: 0.47266912575064696
Test Accuracy: 0.48232827476038337
Done!
```

AlexNet 8

Same as AlexNet 7 but with Adam optimizer instead of SGD.

```
Using cuda
Optimizer: Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
```

```
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.005
    maximize: False
    weight_decay: 0
)
Epochs 20
Batch Size: 32
Model Info: <class '__main__.AlexNet'>
```

I stopped this before it finished because it is obviously broken:

```
Epoch


1
step: 0, loss: 4.605169773101807
step: 100, loss: 4.605169773101807
step: 200, loss: 4.605169773101807
step: 300, loss: 4.605169773101807
step: 400, loss: 4.605169773101807
step: 500, loss: 4.605169773101807
step: 600, loss: 4.605169773101807
step: 700, loss: 4.605169773101807
step: 800, loss: 4.605169773101807
step: 900, loss: 4.605169773101807
step: 1000, loss: 4.605169773101807
step: 1100, loss: 4.605169773101807
step: 1200, loss: 4.605169773101807
step: 1300, loss: 4.605169773101807
step: 1400, loss: 4.605169773101807
step: 1500, loss: 4.605169773101807
 10%|█         | 2/20 [05:34<50:09, 167.20s/it]
F1 Score: 0.00019801980198019803
Test Accuracy: 0.009984025559105431
Epoch


2
```

```
step: 0, loss: 4.605169773101807
step: 100, loss: 4.605169773101807
```