Homework 9

Yuxin Sun

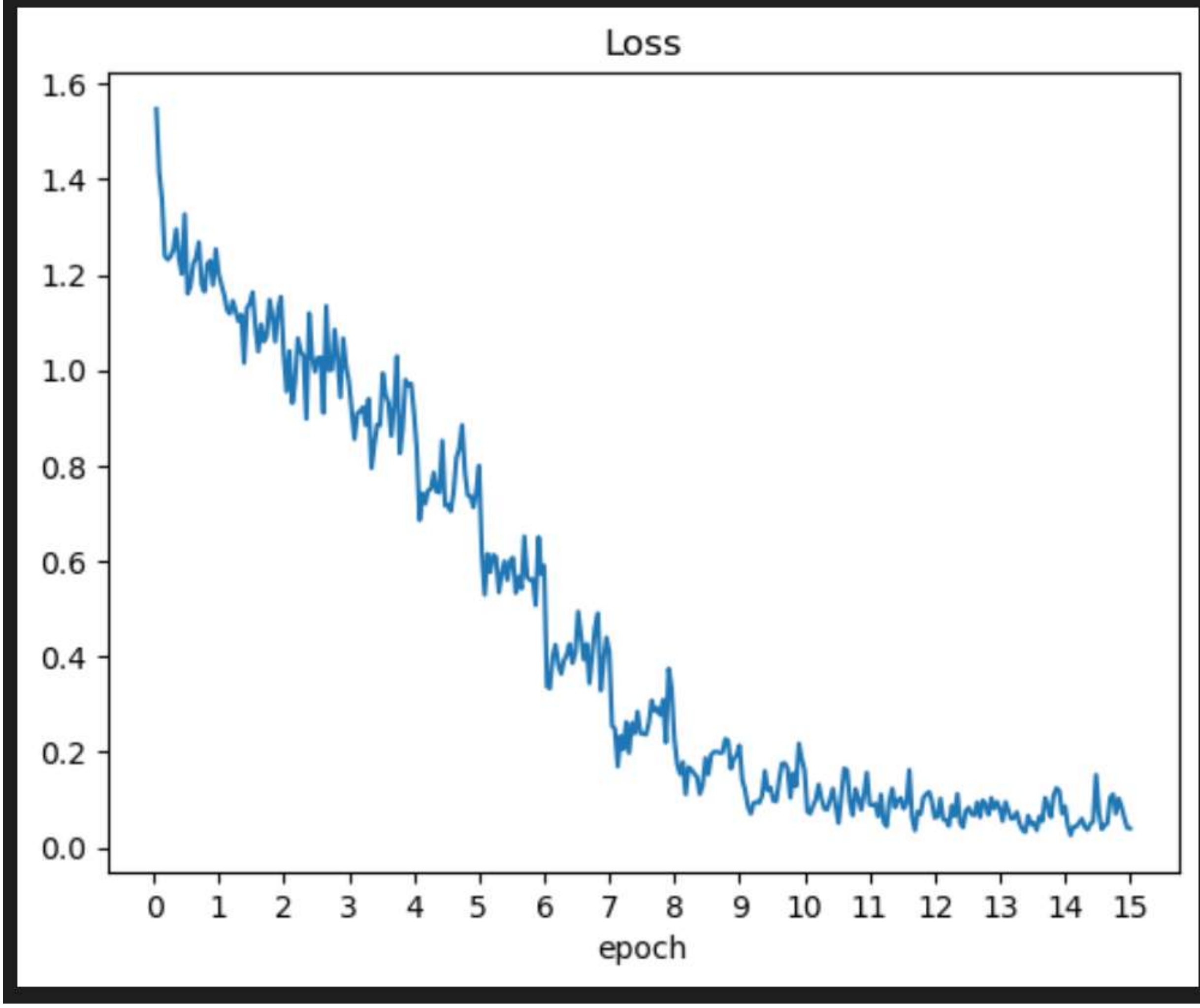
Results

With

```
batch_size=16,
epoch_size=15,
lr=1e-4,
betas=(0.9, 0.99),
measure_rate=20
```

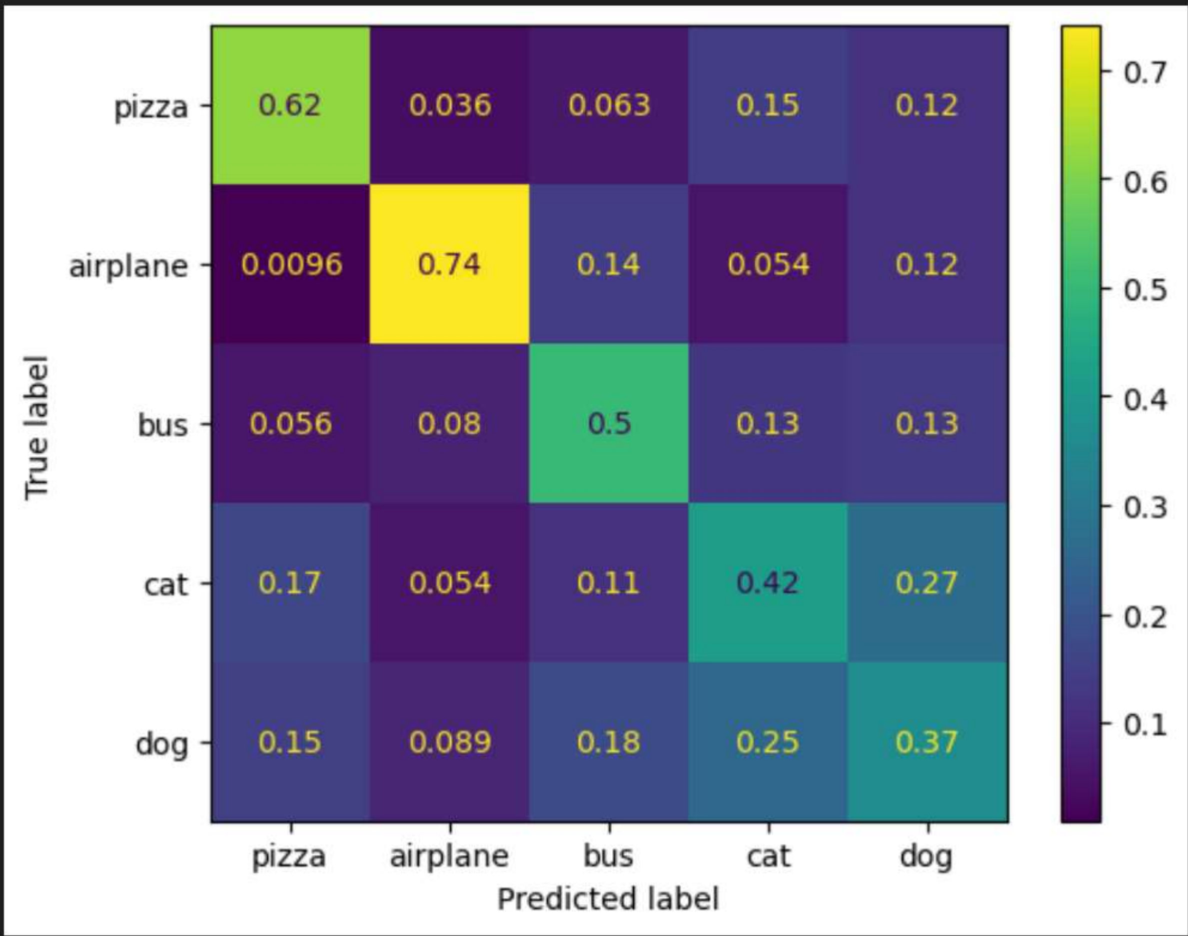
I get

```
Accuracy: 53.20%
```



(Figure 1: Loss)

```
tensor([[324.,  16.,  40.,  75.,  45.],
        [  5., 332.,  90.,  28.,  45.],
        [ 29.,  36., 318.,  66.,  51.],
        [ 86.,  24.,  72., 215., 103.],
        [ 77.,  40., 111., 131., 141.]])
```

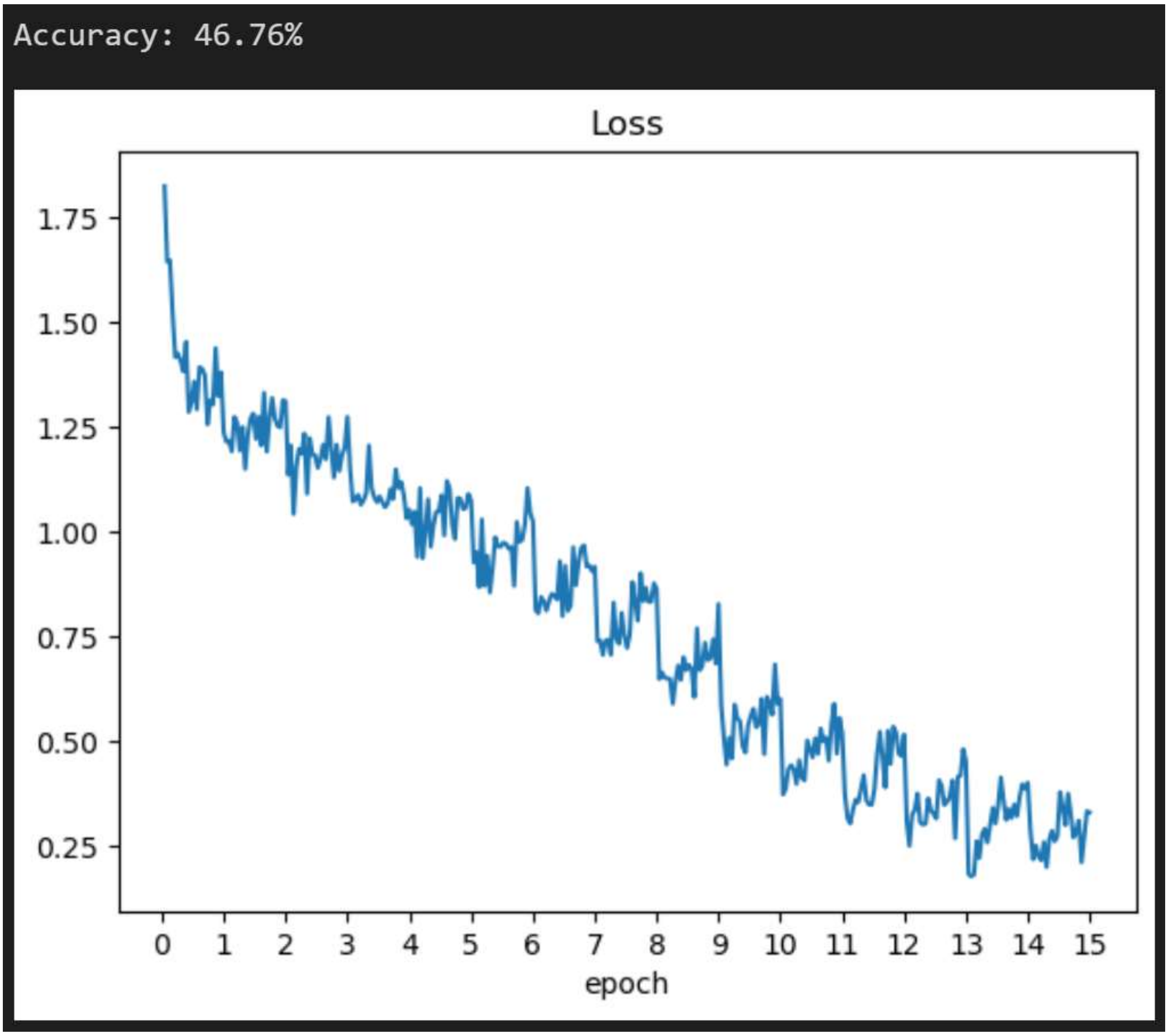


(Figure 2: Confusion Matrix)

Using `torch.einsum()`

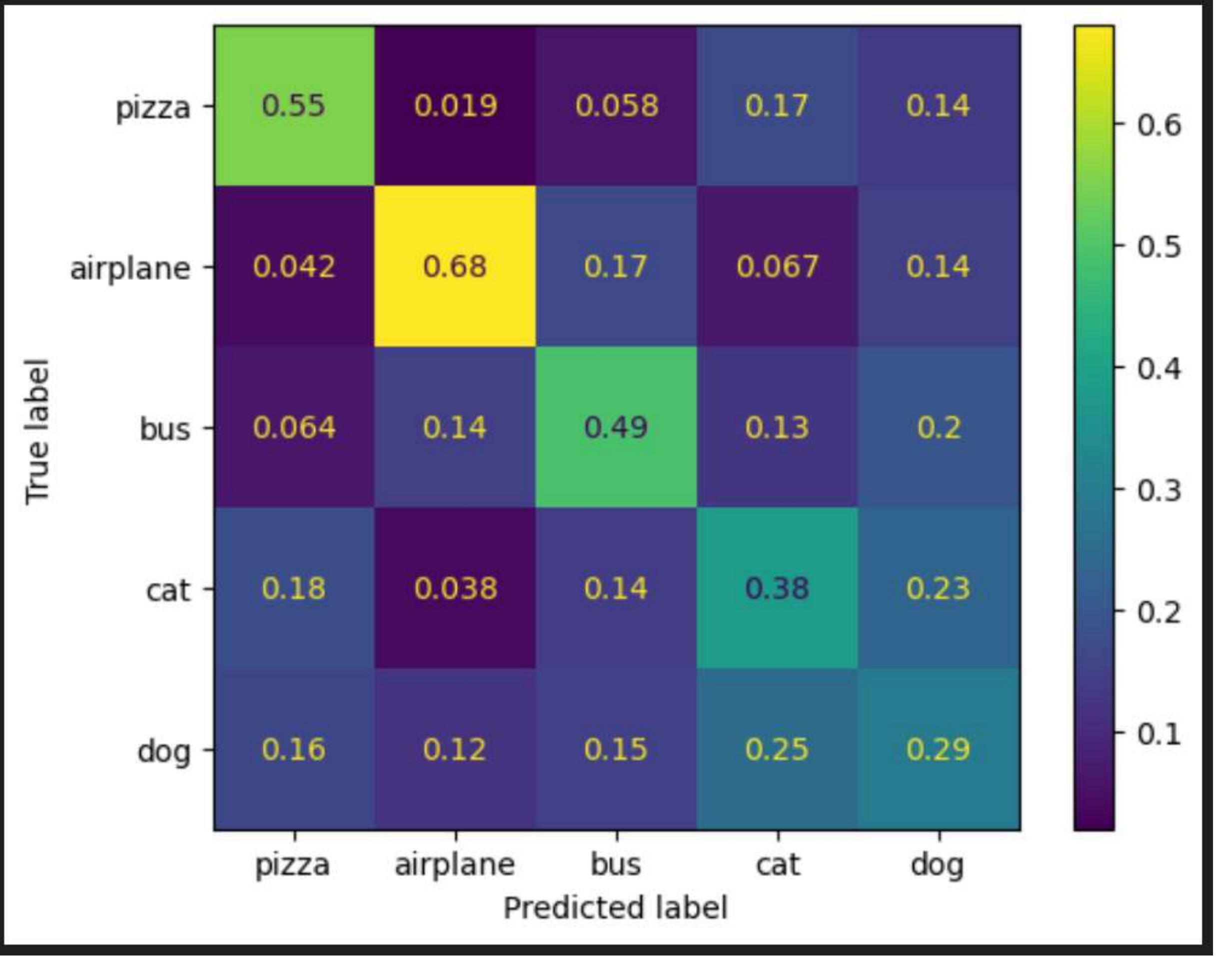
The accuracy should be the same as before.

The smaller accuracy is probally due to no `bias` used in this implementation.



(Figure 3: Loss)

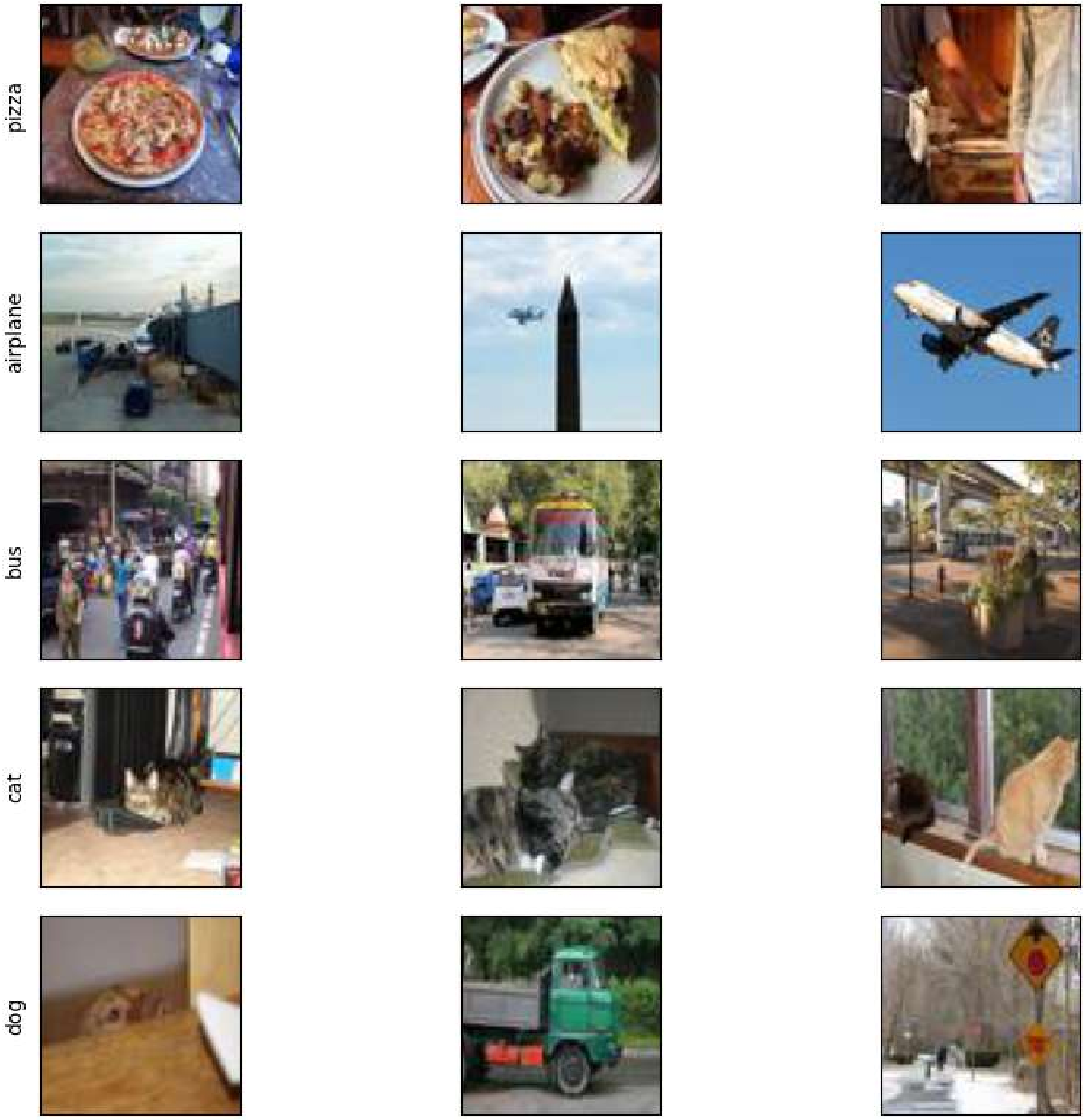
```
tensor([[302.,   8.,  27.,  83.,  80.],
        [ 23., 283.,  77.,  32.,  85.],
        [ 35.,  60., 228.,  61., 116.],
        [ 98.,  16.,  65., 183., 138.],
        [ 89.,  49.,  68., 121., 173.]])
```



(Figure 4: Confusion Matrix)

Datasets

Same as hw4...



Codes

Multi-head Attention Using `torch.einsum()`:

```
class BasicEncoder(nn.Module):
    def __init__():
        ...
        # replace SelfAttention with SelfAttention2
        self.self_attention_layer = SelfAttention2(
                    max_seq_length, embedding_size, num_atten_heads)  # (A)
        ...

class SelfAttention2(nn.Module):
    def __init__(self, max_seq_length, embedding_size, num_atten_heads):
        super().__init__()
        self.max_seq_length = max_seq_length
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
```

```
        self.qkv_size = self.embedding_size // num_atten_heads
        # we just need one tensor:
        self.wqkv_tensor = nn.Parameter(
            torch.randn(num_atten_heads, 3, max_seq_length, self.qkv_size,
max_seq_length, self.qkv_size),
            requires_grad=True
        ).to(device)
        self.coeff =
1.0/torch.sqrt(torch.tensor(self.qkv_size).float()).to(device)

    def forward(self, sentence_tensor):
        b, s, d = sentence_tensor.shape
        x = sentence_tensor.view(b, s, self.num_atten_heads, self.qkv_size)
        # b: batch index,
        # s: sequence index, si sj --> i j
        # h: attention-head index,
        # q: S_{qkv}
        q, k, v = tuple(torch.einsum('bshq,hksqSQ->kbhSQ', x,
self.wqkv_tensor))
        QK_dot_prod = torch.einsum('bhiq,bhjq->bhij', q, k)
        rowwise_softmax_normalizations = torch.softmax(QK_dot_prod, dim=-1)
        z = torch.einsum('bhsj,bhjq->bshq', rowwise_softmax_normalizations, v)
        z = self.coeff * z.reshape(b,s,d)
        return z
```

My `ViT` Net

```
class ViT(nn.Module):
    def __init__(self, *, max_seq_length=17, embedding_size=64, encoders=2,
num_atten_heads=4,
                 device="cpu"):
        super().__init__()
        self.encorder = MasterEncoder(max_seq_length, embedding_size, encoders,
num_atten_heads)
        # tarnsform sentences to embeddings:
        self.embedding = nn.Sequential(
            nn.Conv2d(3, embedding_size, 16, stride=16, padding=0),
            nn.Flatten(start_dim=2, end_dim=-1)
        )
        self.class_head = nn.Parameter(torch.randn(embedding_size,
device=device), requires_grad=True)
        self.position = nn.Parameter(2*torch.rand(max_seq_length,
embedding_size, device=device)-1,requires_grad=True)
        # MLP with only one hidden layer is enough
        self.mlp = nn.Sequential(
            nn.Flatten(),
            nn.Linear(max_seq_length*embedding_size, 128),
            nn.ReLU(),
            nn.Linear(128, 5)
        )

    def forward(self, x):
        # x_out is of (B, 16, D)
        x = self.embedding(x).transpose(1,2)
        # class_head: (B,  1, D)
```

```
        class_head = self.class_head.repeat(x.shape[0],1,1)

        x = torch.cat((x, class_head), dim=1).add(self.position)
        x = self.encorder(x)
        out = self.mlp(x)
        return out

    def train(self, p):
        # p contains all the runtime parameters
        self.device = p.device
        net = self.to(p.device)
        start_time = time.perf_counter()
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(net.parameters(), lr=p.lr, betas=p.betas)
        loss_record = []
        for epoch in range(1, p.epoch_size+1):
            running_loss = 0.
            epoch_start_time = time.perf_counter()
            for i, data in enumerate(p.train_dataloader):
                img_tensor, label_tensor = data
                img_tensor = img_tensor.to(p.device)
                label_tensor = label_tensor.to(p.device)
                optimizer.zero_grad()
                output = net(img_tensor)
                loss = criterion(output, label_tensor)
                running_loss += loss.item()
                loss.backward()
                optimizer.step()

                j = i+1
                if j % p.measure_rate == 0:
                    avg_loss = running_loss/p.measure_rate
                    running_loss = 0.
                    loss_record.append(avg_loss)
                    current_time = time.perf_counter()
                    time_elapsed = current_time - epoch_start_time
                    print("[epoch:%d/%d] [iter:%4d] elapsed time:%4d secs 
loss: %.5f"\
                        %(epoch, p.epoch_size, j, time_elapsed, avg_loss) )

        time_stamp = '_'.join(time.ctime().split(" ")[1:4])
        saved_model = p.save_dir + "saved_model_" + time_stamp
        torch.save(net.state_dict(), saved_model)
        print("model.state_dict() saved to ", saved_model)
        saved_file = p.save_dir + "saved_file_" + time_stamp
        with open(saved_file, 'w') as f:
            for loss in loss_record:
                f.write("%.5f\n" % loss)
            f.flush()
        total_time = time.perf_counter() - start_time
        print("Total training time: %5d secs"%total_time)
        return loss_record, time_stamp


    def test(self, p, time_stamp):
```

```
        self.load_state_dict(torch.load(p.save_dir + "saved_model_" +
time_stamp))
        self.device = p.device
        net = self.to(p.device)
        accuracy = 0
        cf_matrix = torch.zeros(5,5)
        with torch.no_grad():
            for i, data in enumerate(p.test_dataloader):
                img_tensor, label_tensor = data
                img_tensor = img_tensor.to(p.device)
                label_tensor = label_tensor.to(p.device)
                output = net(img_tensor)
                predicted_idx = torch.argmax(output, dim=1)
                for label,prediction in zip(label_tensor,predicted_idx):
                    cf_matrix[label][prediction] += 1
                    if label == prediction:
                        accuracy += 1
        accuracy = float(accuracy) / sum(cf_matrix.view(-1))
        print("Accuracy: %0.2f%%"%(accuracy*100))
        return cf_matrix
```

Discussion

Is the network performance better than CNN version?

- My ViT can recognize pizza better than CNN.
- Its performance for the other 4 classes is always worse than CNN.