EE-449 HW-I

M. Bahadır Dönmezoğlu
2304426

1)

1.1

Neural Networks are trained using an optimization process. And that optimization process requires a loss function to calculate model error. In other words, in that way we know how good our model is. And also loss function generates the gradients that drive the training of the Network.

cross entropy $H(P, Q) = -\sum_{x \text{ in } X} P(x) * \log(Q(x))$
For all

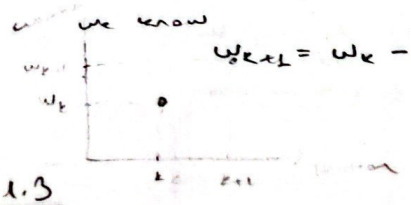
nn.CrossEntropyLoss

$P(x)$ → expected probability (known)
$Q(x)$ → predicted probability (by the model)

Loss functions express how far off the mark our computed output is. Optimizers shape and mold the model into its most accurate possible form by adjusting with the weights which is found by the weighted cross-entropy loss function.

1.2

Stochastic gradient descent (SGD)



1.3

① Batch is the number of samples that we take in one train generation cycle, or number of samples processed before the model is updated. On the other hand an epoch is the number of complete passes through the training dataset.

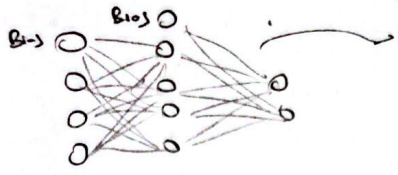
② Samples → N
Batch size → B
number of batches per epoch = $\frac{N}{B}$

③ For every batch I am updating the model the number of batches per epoch was $\frac{N}{B}$ → if I multiply it with E then $\frac{N}{B} E$ = number of SGD iterations

1.4

MLP classifier K hidden units $\rightarrow$ H_k = size of each hidden unit

assume 1 hidden layer $H_1 \times D_{out}$ ← output dimensions for $k = 1, 2 \dots K$



4×2 → # of connections btw second and third layer

2 → # of connections btw bias and third layer

D_{out}

input layer 3 units | hidden layer 4 units | output layer 2 units

↓

there is $D_{in} \times H_1$ ← input dimension

3×4 → number of connections btw first and second layer

$H_1 = 4$ → number of connections btw bias and second layer

When we have K hidden units with H_k for each then

$$D_{in} \times H_1 + \sum_{k=1}^{n-1} H_k \times H_{k+1} + H_K \cdot D_{out} + \sum_{k=1}^{\wedge} H_k + 0$$

- $D_{in} \times H_1$: between input and first layer
- $\sum_{k=1}^{n-1} H_k \times H_{k+1}$: btw hidden layers
- $H_K \cdot D_{out}$: btw last layer and out
- $\sum_{k=1}^{\wedge} H_k$: for every hidden btw bias and output
- 0: btw bias and output

Question 2

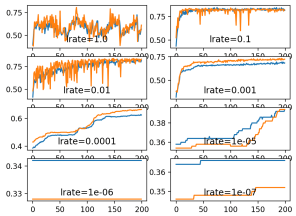
1) Our error functions refer to the generalization performance.
2) The loss curve informative about that.
3) If I would have trained the cnn's, they would have been more informative and accurate than that of mlp's since we have more layers to identify correct class. (cnn5>cnn4>cnn3>mlp2>mlp1)
4) If we have more parameters the accuracy of classification could be increased and the error could be decreased.
5) The depth of the architecture means the layers of architecture. It also increases the accuracy of classification and decreases the error.
6) -
7) As much as I understood the concept, I believe there should be some units specialized for specific classes. Some units more active when we give some class. Otherwise, when we give some other class, they are not that much active which means in a way the unit is a specific unit for that class.
8) Probably mlp_2 ' s weights more interpretable because it is linear in the first step.
9) Mlp_1 and mlp_2 is similar because they are ann structures. And cnn's are similar because they are cnn structures.
10) I would have picked cnn_5 because we have much more layers and it makes the system to decide more easily.

Question 3

1) If I could obtain the graphs, It would be seen with relu activation function we would have better accuracy and smaller loss with respect to sigmoid activation function. When the depth increased again we would have better accuracies and decreased loss function.
2) For sigmoid activation which is a non-linear activation function the values transformed into a value between 0.0 and 1.0. Which means inputs much larger than 1 is transformed 1 and values much smaller than zero snapped to 0. Because of limited saturation and sensitivity we have some challenges to continue to adapt the weights to improve our accuracy and performance.
3) As in two if we would not scaled the inputs to the interval -1 to 1, the values at -1 would snapped to 0 and 1 to 1. In that case we would have some problem because in the scale 0 and -1 would have the same value.

Question 4

1) If we have very small learning rate we would have some problem like the training time. Because for math function of leaning rate we are updating the regression line with that function. In that case with very small steps and it causes the program to learn very slow. On the other hand, for large learning rate, the model is allowed to learn faster but we could have some other problem like sub-optimal final set of weights. We would not have the globally optimized weights as in small learning rate.



2) As stated in 1, we would have globally optimized weights in the case of small learning rate and sub-optimal final set of weights in the case of large learning rate.
3) If we don't have so weird weights from the bigger learning rate steps, we would have better graphs, I think. I tried couple of times with 0.1 learning rate and I believe it was not that bad. And probably with other 0.01 and 0.001 leaning rates the weights would become better and globally optimized.
4) For Adam optimizer we have 0.001 default learning rate. It is better than this case I think because we were using Adam all the time and 0.001 learning rate at the same time. In scheduled learning rate case we are using some 0.1, 0.01 learning rate SGD optimizer for intervals. I believe with Adam optimizer it should be more accurate. But at the same time with 0.1 learning rate we could reach the optimization much faster than as in Adam's. And with the other smaller leaning rates the optimization could be made better.

Code

```
#!/usr/bin/env python3
# -- coding: utf-8 --
"""
Created on Sat Apr 23 11:03:40 2022

@author: bahadir
"""
#%%
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
import os
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms


input_size = 28*28  #28*28 pixels
hidden_size = 64
```

```
num_classes = 10
num_epochs = 15
batch_size = 50
learning_rate = 0.01

# OBTAINING THE DATA
transform_normalize = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5,),(0.5,))])
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True,
download = True,transform = transform_normalize)
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train =
False,transform = transform_normalize)

_, val_dataset = torch.utils.data.random_split(train_data, [54000, 6000])

val_loader =
torch.utils.data.DataLoader(dataset=val_dataset,batch_size=batch_size,shuff
le= False)

train_generator = torch.utils.data.DataLoader(train_data, batch_size =
batch_size, shuffle = True)
test_generator = torch.utils.data.DataLoader(test_data, batch_size =
batch_size , shuffle = False)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Using {device} device")

#%%
class mlp_1(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
      super(mlp_1, self).__init__()
      self.input_size = input_size
      self.fc1 = nn.Linear(input_size , hidden_size)
      self.relu1 = nn.ReLU()
      self.pred = nn.Linear(hidden_size,num_classes)
  def forward(self, x):
      x = x.view(-1, self.input_size)
      output = self.fc1(x)
      output = self.relu1(output)
      output = self.pred(output)
      return output

model_mlp_1 = mlp_1 (input_size, hidden_size, num_classes)
model_mlp_1.to(device)

error = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_mlp_1.parameters(), lr = learning_rate)

runs = 10
num_epochs = 15
count = 0
all_accuracy_list = []
all_lost_list = []

for run in range(runs):
    # Lists for visualization of loss and accuracy
    loss_list = []
    iteration_list = []
```

```
    accuracy_list = []

    # Lists for knowing classwise accuracy
    predictions_list = []
    labels_list = []

    number_correct_train_prediction=0
    #train the model
    for epoch in range(num_epochs):
        for images, labels in train_generator:

            train = torch.autograd.Variable(images.view(-1, 28*28))
            labels = torch.autograd.Variable(labels)

            # Forward pass
            outputs = model_mlp_1(train)
            train_prediction = torch.max(outputs, 1)[1].to(device)
            loss = error(outputs, labels)

            # Initializing a gradient as 0 so there is no mixing of
gradient among the batches
            optimizer.zero_grad()

            #Propagating the error backward
            loss.backward()

            # Optimizing the parameters
            optimizer.step()
            number_correct_train_prediction += ( train_prediction ==
labels).sum()

            count += 1

        # Validating the model

            if count%10 == 0 :
                model_mlp_1.eval()
                total = 0
                correct = 0

                for images, labels in test_generator:
                    images, labels = images.to(device), labels.to(device)
                    labels_list.append(labels)

                    test = torch.autograd.Variable(images.view(-1, 28*28))

                    outputs = model_mlp_1(test)
                    #prediction gets index of predicted label
                    predictions = torch.max(outputs, 1)[1].to(device)
                    predictions_list.append(predictions)
                    correct += (predictions == labels).sum()

                    total += len(labels)

                accuracy_train =
number_correct_train_prediction/batch_size*10
                accuracy_validation = correct * 100 / total
                loss_list.append(loss.data)
                iteration_list.append(count)
                accuracy_list.append(accuracy_validation)
                model_mlp_1.train()
```

```
            if count%500 == 0 :
                print("Iteration: {}, Loss: {}, Accuracy:
{}%".format(count, loss.data, accuracy_validation))
    all_accuracy_list.append(accuracy_list)

best_test_accuracy = max(accuracy_list)
best_test_accuracy_ind = accuracy_list.index(best_test_accuracy)
# ortalama hesapları
loss_curve = []
loss_curve_c = []
average = 0
for i in range(len(loss_list)):
    average += loss_list[runs][i]
loss_curve.append(average / len(loss_curve_c))

train_accuracy = []
avg_train_accuracy = number_correct_train_prediction/count
average = 0
for i in range(len(accuracy_train)):

    average += avg_train_accuracy[runs][i]
    train_accuracy.append(average)/len(accuracy_train)


avg_val_accuracy = []
for i in range(len(accuracy_validation)):
    average = 0
    average += all_accuracy_list[runs][i]
    avg_val_accuracy.append(average/len(all_accuracy_list))
#en son model testi


# dictionary
results_dict = {
     #name of the architecture,
     'name': mlp_1,
     #average of the training loss curves from all runs
     'loss curve': loss_curve,
     #average of the training accuracy curves from all runs
     'train acc curve': train_accuracy,
     #average of the validation accuracy curves from all runs
     'val acc curve': avg_val_accuracy,
     #the best test accuracy value from all runs
     'test acc': best_test_accuracy,
     #the weights of the first layer of the trained ANN with the best test
performance
     'weights': weights
     }
#%%
class mlp_2(nn.Module):
  def _init_(self, input_size, hidden_size, num_classes):
      super(mlp_2, self)._init_()
      self.input_size = input_size
      self.fc1 = nn.Linear(input_size , 16)
      self.relu1 = nn.ReLU()
      self.fc2 = nn.Linear(16 , hidden_size, bias = "False")
      self.pred = nn.Linear(hidden_size,num_classes)
  def forward(self, x):
      x = x.view(-1, self.input_size)
      output = self.fc1(x)
      output = self.relu1(output)
```

```
        output = self.fc2(output)
        output = self.pred(output)
        return output

model_mlp_2 = mlp_2 (input_size, hidden_size, num_classes)
model_mlp_2.to(device)

error = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_mlp_2.parameters(), lr = learning_rate)


num_epochs = 15
count = 0
# Lists for visualization of loss and accuracy
loss_list = []
iteration_list = []
accuracy_list = []

# Lists for knowing classwise accuracy
predictions_list = []
labels_list = []

#train the model
for epoch in range(num_epochs):
    for images, labels in train_generator:

        train = torch.autograd.Variable(images.view(-1, 28*28))
        labels = torch.autograd.Variable(labels)

        # Forward pass
        outputs = model_mlp_2(train)
        loss = error(outputs, labels)

        # Initializing a gradient as 0 so there is no mixing of gradient
among the batches
        optimizer.zero_grad()

        #Propagating the error backward
        loss.backward()

        # Optimizing the parameters
        optimizer.step()

        count += 1

    # Testing the model

        if count%10 == 0 :
            model_mlp_2.eval()
            total = 0
            correct = 0

            for images, labels in test_generator:
                images, labels = images.to(device), labels.to(device)
                labels_list.append(labels)

                test = torch.autograd.Variable(images.view(-1, 28*28))

                outputs = model_mlp_2(test)
                #prediction gets index of predicted label
                predictions = torch.max(outputs, 1)[1].to(device)
```

```
                predictions_list.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
            model_mlp_2.train()
        if count%500 == 0 :
            print("Iteration: {}, Loss: {}, Accuracy: {}%".format(count,
loss.data, accuracy))

#%%

class cnn_3(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
      super(cnn_3, self).__init__()
      self.input_size = input_size
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,
stride=1, padding="valid")
      self.relu1 = nn.ReLU()
      self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=7,
stride=1, padding="valid")
      self.relu2 = nn.ReLU()
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
      self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5,
stride=1, padding="valid")
      self.pool2 = nn.MaxPool2d(2,2)
      self.pred = nn.Linear(16*3*3,num_classes)
  def forward(self, x):
      output = self.conv1(x)
      output = self.relu1(output)
      output = self.conv2(output)
      output = self.relu2(output)
      output = self.pool1(output)
      output = self.conv3(output)
      output = self.pool2(output)
      output = output.view(-1, 16*3*3)
      output = self.pred(output)
      return output

model_cnn_3 = cnn_3 (input_size, hidden_size, num_classes)
model_cnn_3.to(device)

error = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_cnn_3.parameters(), lr = learning_rate)


num_epochs = 15
count = 0
# Lists for visualization of loss and accuracy
loss_list = []
iteration_list = []
accuracy_list = []

# Lists for knowing classwise accuracy
predictions_list = []
labels_list = []
```

```
# transfer your model to train mode
model_cnn_3.train()


#train the model
for epoch in range(num_epochs):
    for images, labels in train_generator:

        train = torch.autograd.Variable(images.view(batch_size,1, 28,28))
        labels = torch.autograd.Variable(labels)

        # Forward pass
        outputs = model_cnn_3(train)
        loss = error(outputs, labels)

        # Initializing a gradient as 0 so there is no mixing of gradient
among the batches
        optimizer.zero_grad()

        #Propagating the error backward
        loss.backward()

        # Optimizing the parameters
        optimizer.step()

        count += 1

    # Testing the model

        if count%10 == 0 :
            model_cnn_3.eval()
            total = 0
            correct = 0
            print(count)
            for images, labels in test_generator:
                images, labels = images.to(device), labels.to(device)
                labels_list.append(labels)

                test = torch.autograd.Variable(images.view(batch_size,1,
28,28))

                outputs = model_cnn_3(test)
                #prediction gets index of predicted label
                predictions = torch.max(outputs, 1)[1].to(device)
                predictions_list.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
            model_cnn_3.train()
        if count%500 == 0 :
            print("Iteration: {}, Loss: {}, Accuracy: {}%".format(count,
loss.data, accuracy))


#%%
```

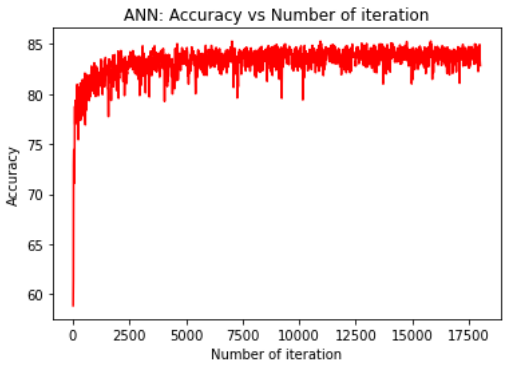
```
class cnn_4(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
      super(cnn_4, self).__init__()
      self.input_size = input_size
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,
stride=1, padding="valid")
      self.relu1 = nn.ReLU()
      self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=5,
stride=1, padding="valid")
      self.relu2 = nn.ReLU()
      self.conv3 = nn.Conv2d(in_channels=8, out_channels=8, kernel_size=3,
stride=1, padding="valid")
      self.relu3 = nn.ReLU()
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2,padding="valid")
      self.conv4 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5,
stride=1, padding="valid")
      self.relu4 = nn.ReLU()
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2,padding="valid")
      # self.pred = nn.Linear(16*3*3,num_classes)
  def forward(self, x):
      output = self.conv1(x)
      output = self.relu1(output)
      output = self.conv2(output)
      output = self.relu2(output)
      output = self.conv3(output)
      output = self.relu3(output)
      output = self.pool1(output)
      output = self.conv4(output)
      output = self.relu4(output)
      output = self.pool2(output)
      output = output.view(-1, 16*3*3)
      output = self.pred(output)
      return output

model_cnn_4 = cnn_4 (input_size, hidden_size, num_classes)
model_cnn_4.to(device)

error = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_cnn_4.parameters(), lr = learning_rate)


num_epochs = 15
count = 0
# Lists for visualization of loss and accuracy
loss_list = []
iteration_list = []
accuracy_list = []

# Lists for knowing classwise accuracy
predictions_list = []
labels_list = []

# transfer your model to train mode
model_cnn_4.train()


#train the model
for epoch in range(num_epochs):
    for images, labels in train_generator:

        train = torch.autograd.Variable(images.view(batch_size,1, 28,28))
```

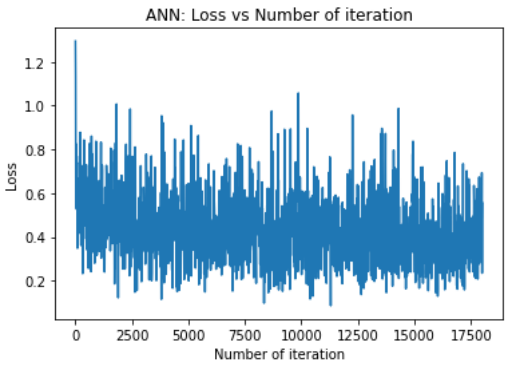
```
        labels = torch.autograd.Variable(labels)

        # Forward pass
        outputs = model_cnn_4(train)
        loss = error(outputs, labels)

        # Initializing a gradient as 0 so there is no mixing of gradient
among the batches
        optimizer.zero_grad()

        #Propagating the error backward
        loss.backward()

        # Optimizing the parameters
        optimizer.step()

        count += 1

    # Testing the model

        if count%10 == 0 :
            model_cnn_4.eval()
            total = 0
            correct = 0
            print(count)
            for images, labels in test_generator:
                images, labels = images.to(device), labels.to(device)
                labels_list.append(labels)

                test = torch.autograd.Variable(images.view(batch_size,1,
28,28))

                outputs = model_cnn_4(test)
                #prediction gets index of predicted label
                predictions = torch.max(outputs, 1)[1].to(device)
                predictions_list.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
            model_cnn_4.train()
        if count%500 == 0 :
            print("Iteration: {}, Loss: {}, Accuracy: {}%".format(count,
loss.data, accuracy))
#%%

class cnn_5(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
      super(cnn_5, self).__init__()
      self.input_size = input_size
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,
stride=1, padding="valid")
      self.relu1 = nn.ReLU()
      self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3,
stride=1, padding="valid")
      self.relu2 = nn.ReLU()
```

```
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=8, kernel_size=3,
stride=1, padding="valid")
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=8, out_channels=8, kernel_size=3,
stride=1, padding="valid")
        self.relu4 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2,padding="valid")
        self.conv5 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3,
stride=1, padding="valid")
        self.relu5 = nn.ReLU()
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=16,
kernel_size=3, stride=1, padding="valid")
        self.relu6 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2,padding="valid")
        self.pred = nn.Linear(16*3*3,num_classes)
    def forward(self, x):
        output = self.conv1(x)
        output = self.relu1(output)
        output = self.conv2(output)
        output = self.relu2(output)
        output = self.conv3(output)
        output = self.relu3(output)
        output = self.conv4(output)
        output = self.relu4(output)
        output = self.pool1(output)
        output = self.conv5(output)
        output = self.relu5(output)
        output = self.conv6(output)
        output = self.relu6(output)
        output = self.pool2(output)
        output = output.view(-1, 16*3*3)
        output = self.pred(output)
        return output

model_cnn_5 = cnn_5 (input_size, hidden_size, num_classes)
model_cnn_5.to(device)

error = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_cnn_5.parameters(), lr = learning_rate)


num_epochs = 15
count = 0
# Lists for visualization of loss and accuracy
loss_list = []
iteration_list = []
accuracy_list = []

# Lists for knowing classwise accuracy
predictions_list = []
labels_list = []

# transfer your model to train mode
model_cnn_5.train()


#train the model
for epoch in range(num_epochs):
    for images, labels in train_generator:

        train = torch.autograd.Variable(images.view(batch_size,1, 28,28))
```

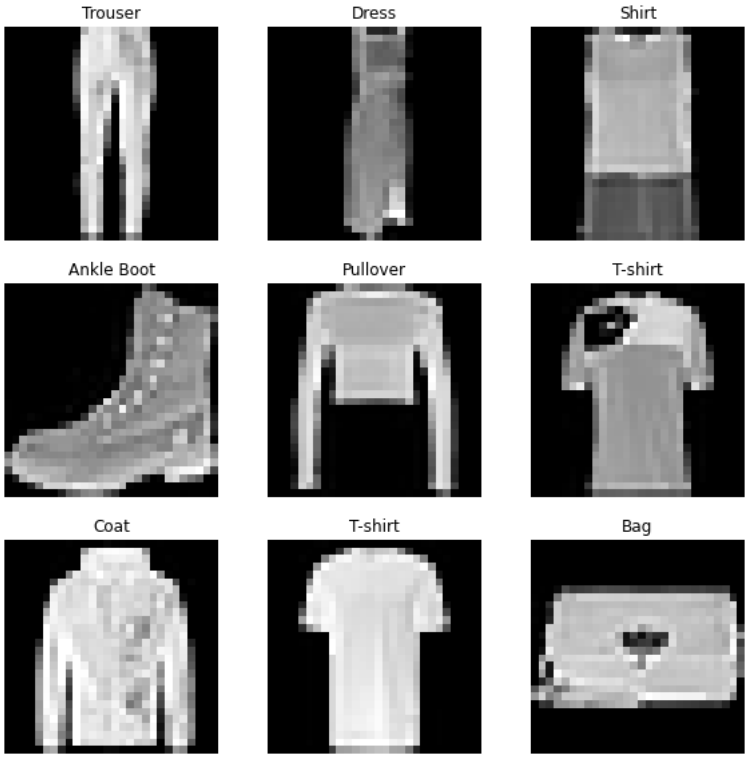
```
        labels = torch.autograd.Variable(labels)

        # Forward pass
        outputs = model_cnn_5(train)
        loss = error(outputs, labels)

        # Initializing a gradient as 0 so there is no mixing of gradient
among the batches
        optimizer.zero_grad()

        #Propagating the error backward
        loss.backward()

        # Optimizing the parameters
        optimizer.step()

        count += 1

    # Testing the model

        if count%10 == 0 :
            model_cnn_5.eval()
            total = 0
            correct = 0
            print(count)
            for images, labels in test_generator:
                images, labels = images.to(device), labels.to(device)
                labels_list.append(labels)

                test = torch.autograd.Variable(images.view(batch_size,1,
28,28))

                outputs = model_cnn_5(test)
                #prediction gets index of predicted label
                predictions = torch.max(outputs, 1)[1].to(device)
                predictions_list.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)

            accuracy = correct * 100 / total
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
            model_cnn_5.train()
        if count%500 == 0 :
            print("Iteration: {}, Loss: {}, Accuracy: {}%".format(count,
loss.data, accuracy))
```

The accuracy and error graph with respect to number of iterations. (Because I did not have enough time I could not have 10 runs but the graphs for 1 run are the followings)





Iterating and Visualizing the Dataset



Output of model mlp_1

```
Python 3.9.7 (default, Sep 16 2021, 08:50:36)
Type "copyright", "credits" or "license" for more information.

IPython 7.29.0 -- An enhanced Interactive Python.

runcell(1, '/Users/bahadir/.spyder-py3/untitled0.py')
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
  0%|          | 0/26421880 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
  0%|          | 0/29515 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
  0%|          | 0/4422102 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
  0%|          | 0/5148 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Using cpu device

runcell(2, '/Users/bahadir/.spyder-py3/untitled0.py')
Iteration: 500, Loss: 0.30281394720077515, Accuracy: 80.58999633789062%
Iteration: 1000, Loss: 0.21061915159225464, Accuracy: 79.33999633789062%
Iteration: 1500, Loss: 0.25873738527297974, Accuracy: 80.76000213623047%
Iteration: 2000, Loss: 0.7406589984893799, Accuracy: 82.63999938964844%
Iteration: 2500, Loss: 0.6610649824142456, Accuracy: 83.56999969482422%
Iteration: 3000, Loss: 0.4102923274040222, Accuracy: 83.12000274658203%
Iteration: 3500, Loss: 0.4878459870815277, Accuracy: 83.19999694824219%
Iteration: 4000, Loss: 0.26858386397361755, Accuracy: 81.2699966430664%
```

Iteration: 4500, Loss: 0.2061496526002884, Accuracy: 83.38999938964844%
Iteration: 5000, Loss: 0.38322025537490845, Accuracy: 81.95999908447266%
Iteration: 5500, Loss: 0.4410715401172638, Accuracy: 83.0999984741211%
Iteration: 6000, Loss: 0.28156420588493347, Accuracy: 83.87000274658203%
Iteration: 6500, Loss: 0.4532167315483093, Accuracy: 83.69000244140625%
Iteration: 7000, Loss: 0.31280383467674255, Accuracy: 83.91999816894531%
Iteration: 7500, Loss: 0.37113675475120544, Accuracy: 81.26000213623047%
Iteration: 8000, Loss: 0.6681090593338013, Accuracy: 83.2300033569336%
Iteration: 8500, Loss: 0.45726126432418823, Accuracy: 84.55999755859375%
Iteration: 9000, Loss: 0.13477914035320282, Accuracy: 84.13999938964844%
Iteration: 9500, Loss: 0.5429471731185913, Accuracy: 82.66999816894531%
Iteration: 10000, Loss: 0.3629310131072998, Accuracy: 82.95999908447266%
Iteration: 10500, Loss: 0.2201155722141266, Accuracy: 82.66999816894531%
Iteration: 11000, Loss: 0.195046067237854, Accuracy: 83.61000061035156%
Iteration: 11500, Loss: 0.3919444680213928, Accuracy: 83.41999816894531%
Iteration: 12000, Loss: 0.32264891266822815, Accuracy: 83.87999725341797%
Iteration: 12500, Loss: 0.38747042417526245, Accuracy: 81.75%
Iteration: 13000, Loss: 0.23510989546775818, Accuracy: 82.12000274658203%
Iteration: 13500, Loss: 0.3643873333930969, Accuracy: 83.88999938964844%
Iteration: 14000, Loss: 0.3985835313796997, Accuracy: 83.16000366210938%
Iteration: 14500, Loss: 0.43487799167633057, Accuracy: 82.19999694824219%
Iteration: 15000, Loss: 0.2986260950565338, Accuracy: 84.11000061035156%
Iteration: 15500, Loss: 0.5013670921325684, Accuracy: 84.91000366210938%
Iteration: 16000, Loss: 0.5255785584449768, Accuracy: 83.54000091552734%
Iteration: 16500, Loss: 0.4205489754676819, Accuracy: 83.5199966430664%
Iteration: 17000, Loss: 0.3422054052352905, Accuracy: 83.72000122070312%
Iteration: 17500, Loss: 0.5949613451957703, Accuracy: 83.63999938964844%
Iteration: 18000, Loss: 0.3004949986934662, Accuracy: 84.0199966430664%
Iteration: 18500, Loss: 0.4404122829437256, Accuracy: 84.20999908447266%
Iteration: 19000, Loss: 0.32943251729011536, Accuracy: 83.31999969482422%
Iteration: 19500, Loss: 0.4472751319408417, Accuracy: 83.81999969482422%
Iteration: 20000, Loss: 0.5838795304298401, Accuracy: 82.87000274658203%
Iteration: 20500, Loss: 0.4288015067577362, Accuracy: 83.69000244140625%
Iteration: 21000, Loss: 0.49116089940071106, Accuracy: 82.79000091552734%
Iteration: 21500, Loss: 0.28854048252105713, Accuracy: 83.16000366210938%
Iteration: 22000, Loss: 0.6280359625816345, Accuracy: 82.2300033569336%
Iteration: 22500, Loss: 0.277372807264328, Accuracy: 83.41999816894531%
Iteration: 23000, Loss: 0.40415894985198975, Accuracy: 83.94000244140625%
Iteration: 23500, Loss: 0.32747092843055725, Accuracy: 83.33000183105469%
Iteration: 24000, Loss: 0.17109818756580353, Accuracy: 82.68000030517578%
Iteration: 24500, Loss: 0.17607517540454865, Accuracy: 83.63999938964844%
Iteration: 25000, Loss: 0.4437640905380249, Accuracy: 82.62999725341797%
Iteration: 25500, Loss: 0.32746464014053345, Accuracy: 83.5199966430664%
Iteration: 26000, Loss: 0.3665664792060852, Accuracy: 83.41999816894531%
Iteration: 26500, Loss: 0.4107753038406372, Accuracy: 84.51000213623047%
Iteration: 27000, Loss: 0.2935720980167389, Accuracy: 83.9800033569336%
Iteration: 27500, Loss: 0.26401621103286743, Accuracy: 83.20999908447266%

Iteration: 28000, Loss: 0.45233094692230225, Accuracy: 81.77999877929688%
Iteration: 28500, Loss: 0.6597248315811157, Accuracy: 84.33999633789062%
Iteration: 29000, Loss: 0.15411332249641418, Accuracy: 83.1500015258789%
Iteration: 29500, Loss: 0.33012256026268005, Accuracy: 83.55999755859375%
Iteration: 30000, Loss: 0.4475177824497223, Accuracy: 82.3499984741211%
Iteration: 30500, Loss: 0.48495978116989136, Accuracy: 83.83999633789062%
Iteration: 31000, Loss: 0.2543870806694031, Accuracy: 84.02999877929688%
Iteration: 31500, Loss: 0.5947397351264954, Accuracy: 81.80999755859375%
Iteration: 32000, Loss: 0.7134096026420593, Accuracy: 83.16000366210938%
Iteration: 32500, Loss: 0.4818018078804016, Accuracy: 82.41999816894531%
Iteration: 33000, Loss: 0.2352738380432129, Accuracy: 83.05999755859375%
Iteration: 33500, Loss: 0.3248215913772583, Accuracy: 84.02999877929688%
Iteration: 34000, Loss: 0.6254218816757202, Accuracy: 81.80000305175781%
Iteration: 34500, Loss: 0.4107733964920044, Accuracy: 81.66999816894531%
Iteration: 35000, Loss: 0.16429512202739716, Accuracy: 84.44000244140625%
Iteration: 35500, Loss: 0.652666449546814, Accuracy: 83.6500015258789%
Iteration: 36000, Loss: 0.6114413738250732, Accuracy: 84.19999694824219%
Iteration: 36500, Loss: 0.4186820089817047, Accuracy: 82.62000274658203%
Iteration: 37000, Loss: 0.18643659353256226, Accuracy: 82.68000030517578%
Iteration: 37500, Loss: 0.5062575340270996, Accuracy: 83.91000366210938%
Iteration: 38000, Loss: 0.27177637815475464, Accuracy: 83.5%
Iteration: 38500, Loss: 0.34579676389694214, Accuracy: 84.47000122070312%
Iteration: 39000, Loss: 0.5221287608146667, Accuracy: 83.05000305175781%
Iteration: 39500, Loss: 0.5015085339546204, Accuracy: 82.36000061035156%
Iteration: 40000, Loss: 0.3973713219165802, Accuracy: 84.41999816894531%
Iteration: 40500, Loss: 0.6183730959892273, Accuracy: 82.19999694824219%
Iteration: 41000, Loss: 0.2259405255317688, Accuracy: 83.58000183105469%
Iteration: 41500, Loss: 0.3109637200832367, Accuracy: 82.52999877929688%
Iteration: 42000, Loss: 0.35020583868026733, Accuracy: 83.93000030517578%
Iteration: 42500, Loss: 0.3647209405899048, Accuracy: 83.55000305175781%
Iteration: 43000, Loss: 0.20796659588813782, Accuracy: 84.0%
Iteration: 43500, Loss: 0.4239516854286194, Accuracy: 83.98999786376953%
Iteration: 44000, Loss: 0.4327169358730316, Accuracy: 83.1500015258789%
Iteration: 44500, Loss: 0.23013907670974731, Accuracy: 83.88999938964844%
Iteration: 45000, Loss: 0.45363253355026245, Accuracy: 83.87000274658203%
Iteration: 45500, Loss: 0.3878730535507202, Accuracy: 83.69000244140625%
Iteration: 46000, Loss: 0.3042542338371277, Accuracy: 83.27999877929688%
Iteration: 46500, Loss: 0.4374173879623413, Accuracy: 83.52999877929688%
Iteration: 47000, Loss: 0.6995729804039001, Accuracy: 84.3499984741211%
Iteration: 47500, Loss: 0.47563475370407104, Accuracy: 84.19999694824219%
Iteration: 48000, Loss: 0.18715380132198334, Accuracy: 83.80000305175781%
Iteration: 48500, Loss: 0.2865796983242035, Accuracy: 84.36000061035156%
Iteration: 49000, Loss: 0.2543128430843353, Accuracy: 81.97000122070312%
Iteration: 49500, Loss: 0.3243083655834198, Accuracy: 83.55999755859375%
Iteration: 50000, Loss: 0.254349946975708, Accuracy: 83.66000366210938%
Iteration: 50500, Loss: 0.19897890090942383, Accuracy: 84.1500015258789%
Iteration: 51000, Loss: 0.6560769081115723, Accuracy: 82.98999786376953%

Iteration: 51500, Loss: 0.26254352927207947, Accuracy: 83.83000183105469%
Iteration: 52000, Loss: 0.36631956696510315, Accuracy: 83.0%
Iteration: 52500, Loss: 0.44154155254364014, Accuracy: 83.55000305175781%
Iteration: 53000, Loss: 0.35836535692214966, Accuracy: 83.76000213623047%
Iteration: 53500, Loss: 0.3914065659046173, Accuracy: 83.44999694824219%
Iteration: 54000, Loss: 0.33277738094329834, Accuracy: 83.44000244140625%
Iteration: 54500, Loss: 0.17659877240657806, Accuracy: 81.91000366210938%
Iteration: 55000, Loss: 0.3023258149623871, Accuracy: 83.0999984741211%
Iteration: 55500, Loss: 0.4416563808917999, Accuracy: 84.68000030517578%
Iteration: 56000, Loss: 0.25262391567230225, Accuracy: 83.72000122070312%
Iteration: 56500, Loss: 0.33610793948173523, Accuracy: 83.2300033569336%
Iteration: 57000, Loss: 0.3640691041946411, Accuracy: 83.58000183105469%
Iteration: 57500, Loss: 0.24220745265483856, Accuracy: 84.19000244140625%
Iteration: 58000, Loss: 0.3605872392654419, Accuracy: 82.81999969482422%
Iteration: 58500, Loss: 0.4062664806842804, Accuracy: 82.93000030517578%
Iteration: 59000, Loss: 0.4225532114505768, Accuracy: 84.73999786376953%
Iteration: 59500, Loss: 0.21230916678905487, Accuracy: 82.87999725341797%
Iteration: 60000, Loss: 0.39100274443626404, Accuracy: 83.68000030517578%
Iteration: 60500, Loss: 0.33584052324295044, Accuracy: 83.36000061035156%
Iteration: 61000, Loss: 0.30361419916152954, Accuracy: 83.02999877929688%
Iteration: 61500, Loss: 0.4462762475013733, Accuracy: 82.25%
Iteration: 62000, Loss: 0.23850220441818237, Accuracy: 84.55999755859375%
Iteration: 62500, Loss: 0.2812877595424652, Accuracy: 83.58000183105469%
Iteration: 63000, Loss: 0.19811183214187622, Accuracy: 83.61000061035156%
Iteration: 63500, Loss: 0.2716415822505951, Accuracy: 82.30000305175781%
Iteration: 64000, Loss: 0.3163633942604065, Accuracy: 84.08000183105469%
Iteration: 64500, Loss: 0.44884583353996277, Accuracy: 84.22000122070312%
Iteration: 65000, Loss: 0.3843729794025421, Accuracy: 82.80999755859375%
Iteration: 65500, Loss: 0.41382843255996704, Accuracy: 83.52999877929688%
Iteration: 66000, Loss: 0.273374080657959, Accuracy: 83.16999816894531%
Iteration: 66500, Loss: 0.37973418831825256, Accuracy: 83.91999816894531%
Iteration: 67000, Loss: 0.34746283292770386, Accuracy: 83.87999725341797%
Iteration: 67500, Loss: 0.2731022238731384, Accuracy: 83.13999938964844%
Iteration: 68000, Loss: 0.720797061920166, Accuracy: 83.31999969482422%
Iteration: 68500, Loss: 0.42492496967315674, Accuracy: 83.5999984741211%
Iteration: 69000, Loss: 0.8572826981544495, Accuracy: 83.08999633789062%
Iteration: 69500, Loss: 0.7402145862579346, Accuracy: 82.36000061035156%
Iteration: 70000, Loss: 0.3226686716079712, Accuracy: 82.02999877929688%
Iteration: 70500, Loss: 0.39504000544548035, Accuracy: 83.48999786376953%
Iteration: 71000, Loss: 0.4160623252391815, Accuracy: 84.26000213623047%
Iteration: 71500, Loss: 0.4345466196537018, Accuracy: 83.47000122070312%
Iteration: 72000, Loss: 0.3333628475666046, Accuracy: 82.44000244140625%
Iteration: 72500, Loss: 0.26853904128074646, Accuracy: 83.8499984741211%
Iteration: 73000, Loss: 0.25342822074890137, Accuracy: 83.70999908447266%
Iteration: 73500, Loss: 0.13076511025428772, Accuracy: 84.0199966430664%
Iteration: 74000, Loss: 0.2478911578655243, Accuracy: 83.54000091552734%
Iteration: 74500, Loss: 0.26679158210754395, Accuracy: 84.29000091552734%

Iteration: 75000, Loss: 0.2873034179210663, Accuracy: 83.54000091552734%
Iteration: 75500, Loss: 0.5308369994163513, Accuracy: 82.87999725341797%
Iteration: 76000, Loss: 0.3455490171909332, Accuracy: 82.2300033569336%
Iteration: 76500, Loss: 0.4261252284049988, Accuracy: 84.02999877929688%
Iteration: 77000, Loss: 0.38856834173202515, Accuracy: 84.22000122070312%
Iteration: 77500, Loss: 0.22032853960990906, Accuracy: 84.19000244140625%
Iteration: 78000, Loss: 0.48993387818336487, Accuracy: 83.33000183105469%
Iteration: 78500, Loss: 0.3756147027015686, Accuracy: 84.22000122070312%
Iteration: 79000, Loss: 0.28483447432518005, Accuracy: 83.95999908447266%
Iteration: 79500, Loss: 0.39620548486709595, Accuracy: 82.75%
Iteration: 80000, Loss: 0.3458147943019867, Accuracy: 82.13999938964844%
Iteration: 80500, Loss: 0.501900315284729, Accuracy: 83.88999938964844%
Iteration: 81000, Loss: 0.30823373794555664, Accuracy: 83.94999694824219%
Iteration: 81500, Loss: 0.36390233039855957, Accuracy: 84.4800033569336%
Iteration: 82000, Loss: 0.535457968711853, Accuracy: 83.86000061035156%
Iteration: 82500, Loss: 0.37262198328971863, Accuracy: 83.95999908447266%
Iteration: 83000, Loss: 0.24250398576259613, Accuracy: 83.8499984741211%
Iteration: 83500, Loss: 0.44313758611679077, Accuracy: 84.43000030517578%
Iteration: 84000, Loss: 0.38506463170051575, Accuracy: 83.79000091552734%
Iteration: 84500, Loss: 0.3214307427406311, Accuracy: 83.12999725341797%
Iteration: 85000, Loss: 0.25141793489456177, Accuracy: 83.55000305175781%
Iteration: 85500, Loss: 0.5035153031349182, Accuracy: 83.66999816894531%
Iteration: 86000, Loss: 0.31425562500953674, Accuracy: 83.43000030517578%
Iteration: 86500, Loss: 0.2417401820421219, Accuracy: 83.93000030517578%
Iteration: 87000, Loss: 0.5597366094589233, Accuracy: 82.93000030517578%
Iteration: 87500, Loss: 0.17427173256874084, Accuracy: 83.66999816894531%
Iteration: 88000, Loss: 0.272776335477829, Accuracy: 83.86000061035156%
Iteration: 88500, Loss: 0.3144603669643402, Accuracy: 84.3499984741211%
Iteration: 89000, Loss: 0.2684382498264313, Accuracy: 84.18000030517578%
Iteration: 89500, Loss: 0.3395521640777588, Accuracy: 83.69999694824219%
Iteration: 90000, Loss: 0.38788169622421265, Accuracy: 84.1500015258789%
Iteration: 90500, Loss: 0.38354283571243286, Accuracy: 83.83000183105469%
Iteration: 91000, Loss: 0.32705390453338623, Accuracy: 84.23999786376953%
Iteration: 91500, Loss: 0.48719486594200134, Accuracy: 84.61000061035156%
Iteration: 92000, Loss: 0.3821240961551666, Accuracy: 83.0%
Iteration: 92500, Loss: 0.4958348572254181, Accuracy: 83.77999877929688%
Iteration: 93000, Loss: 0.3400924801826477, Accuracy: 82.91999816894531%
Iteration: 93500, Loss: 0.5626175403594971, Accuracy: 83.08999633789062%
Iteration: 94000, Loss: 0.21134696900844574, Accuracy: 84.37999725341797%
Iteration: 94500, Loss: 0.17371642589569092, Accuracy: 83.27999877929688%
Iteration: 95000, Loss: 0.46369290351867676, Accuracy: 84.16000366210938%
Iteration: 95500, Loss: 0.4145772159099579, Accuracy: 82.6500015258789%
Iteration: 96000, Loss: 0.7021473050117493, Accuracy: 83.37000274658203%
Iteration: 96500, Loss: 0.15428540110588074, Accuracy: 83.73999786376953%
Iteration: 97000, Loss: 0.4518117904663086, Accuracy: 83.75%
Iteration: 97500, Loss: 0.6609390377998352, Accuracy: 83.79000091552734%
Iteration: 98000, Loss: 0.32353127002716064, Accuracy: 84.33999633789062%

Iteration: 98500, Loss: 0.3781585991382599, Accuracy: 84.0199966430664%
Iteration: 99000, Loss: 0.26480385661125183, Accuracy: 82.97000122070312%
Iteration: 99500, Loss: 0.17064589262008667, Accuracy: 82.91000366210938%
Iteration: 100000, Loss: 0.3470631539821625, Accuracy: 83.41000366210938%
Iteration: 100500, Loss: 0.32398369908332825, Accuracy: 83.5999984741211%
Iteration: 101000, Loss: 0.2756451666355133, Accuracy: 82.61000061035156%
Iteration: 101500, Loss: 0.5528647303581238, Accuracy: 82.36000061035156%
Iteration: 102000, Loss: 0.20035777986049652, Accuracy: 84.26000213623047%
Iteration: 102500, Loss: 0.32144874334335327, Accuracy: 84.29000091552734%
Iteration: 103000, Loss: 0.310200959444046, Accuracy: 83.26000213623047%
Iteration: 103500, Loss: 0.2894570827484131, Accuracy: 82.02999877929688%
Iteration: 104000, Loss: 0.43208399415016174, Accuracy: 83.45999908447266%
Iteration: 104500, Loss: 0.266829252243042, Accuracy: 83.27999877929688%
Iteration: 105000, Loss: 0.281006395816803, Accuracy: 83.4800033569336%
Iteration: 105500, Loss: 0.40699881315231323, Accuracy: 82.9000015258789%
Iteration: 106000, Loss: 0.09518630057573318, Accuracy: 84.29000091552734%
Iteration: 106500, Loss: 0.5113990902900696, Accuracy: 83.80999755859375%
Iteration: 107000, Loss: 0.34743449091911316, Accuracy: 81.88999938964844%
Iteration: 107500, Loss: 0.3559413552284241, Accuracy: 82.38999938964844%
Iteration: 108000, Loss: 0.2803821861743927, Accuracy: 83.79000091552734%
Iteration: 108500, Loss: 0.35234755277633667, Accuracy: 84.27999877929688%
Iteration: 109000, Loss: 0.1956750899553299, Accuracy: 84.44000244140625%
Iteration: 109500, Loss: 0.4831933081150055, Accuracy: 82.12999725341797%
Iteration: 110000, Loss: 0.33321288228034973, Accuracy: 83.22000122070312%
Iteration: 110500, Loss: 0.20174503326416016, Accuracy: 83.76000213623047%
Iteration: 111000, Loss: 0.21332469582557678, Accuracy: 83.26000213623047%
Iteration: 111500, Loss: 0.33487269282341003, Accuracy: 84.26000213623047%
Iteration: 112000, Loss: 0.3886587917804718, Accuracy: 82.01000213623047%
Iteration: 112500, Loss: 0.4633243978023529, Accuracy: 83.1500015258789%
Iteration: 113000, Loss: 0.49445632100105286, Accuracy: 83.22000122070312%
Iteration: 113500, Loss: 0.24930933117866516, Accuracy: 83.11000061035156%
Iteration: 114000, Loss: 0.33140599727630615, Accuracy: 83.8499984741211%
Iteration: 114500, Loss: 0.47656071186065674, Accuracy: 84.22000122070312%
Iteration: 115000, Loss: 0.26355352997779846, Accuracy: 83.54000091552734%
Iteration: 115500, Loss: 0.3040688633918762, Accuracy: 83.18000030517578%
Iteration: 116000, Loss: 0.20085611939430237, Accuracy: 83.0999984741211%
Iteration: 116500, Loss: 0.22258232533931732, Accuracy: 82.94999694824219%
Iteration: 117000, Loss: 0.4282199442386627, Accuracy: 81.18000030517578%
Iteration: 117500, Loss: 0.41322657465934753, Accuracy: 84.16999816894531%
Iteration: 118000, Loss: 0.2263931781053543, Accuracy: 83.63999938964844%
Iteration: 118500, Loss: 0.39206868410110474, Accuracy: 83.08000183105469%
Iteration: 119000, Loss: 0.4031245708465576, Accuracy: 83.04000091552734%
Iteration: 119500, Loss: 0.20501123368740082, Accuracy: 83.29000091552734%
Iteration: 120000, Loss: 0.5641018748283386, Accuracy: 83.3499984741211%
Iteration: 120500, Loss: 0.6886283159255981, Accuracy: 82.51000213623047%
Iteration: 121000, Loss: 0.40759408473968506, Accuracy: 83.18000030517578%
Iteration: 121500, Loss: 0.35319554805755615, Accuracy: 84.05000305175781%

Iteration: 122000, Loss: 0.21558105945587158, Accuracy: 83.6500015258789%
Iteration: 122500, Loss: 0.40932953357696533, Accuracy: 83.69999694824219%
Iteration: 123000, Loss: 0.3327326476573944, Accuracy: 81.58999633789062%
Iteration: 123500, Loss: 0.22456085681915283, Accuracy: 83.25%
Iteration: 124000, Loss: 0.18864290416240692, Accuracy: 83.56999969482422%
Iteration: 124500, Loss: 0.6091114282608032, Accuracy: 83.12000274658203%
Iteration: 125000, Loss: 0.1262509673833847, Accuracy: 82.30999755859375%
Iteration: 125500, Loss: 0.08094341307878494, Accuracy: 83.87999725341797%
Iteration: 126000, Loss: 0.6233301758766174, Accuracy: 82.55999755859375%
Iteration: 126500, Loss: 0.6446961760520935, Accuracy: 82.79000091552734%
Iteration: 127000, Loss: 0.3820938467979431, Accuracy: 82.16999816894531%
Iteration: 127500, Loss: 0.3577289581298828, Accuracy: 83.41000366210938%
Iteration: 128000, Loss: 0.375624418258667, Accuracy: 83.73999786376953%
Iteration: 128500, Loss: 0.25023114681243896, Accuracy: 83.45999908447266%
Iteration: 129000, Loss: 0.39272844791412354, Accuracy: 83.11000061035156%
Iteration: 129500, Loss: 0.08804668486118317, Accuracy: 82.33999633789062%
Iteration: 130000, Loss: 0.469747930765152, Accuracy: 80.91999816894531%
Iteration: 130500, Loss: 0.2750425636768341, Accuracy: 83.55000305175781%
Iteration: 131000, Loss: 0.5213786363601685, Accuracy: 83.11000061035156%
Iteration: 131500, Loss: 0.5690513253211975, Accuracy: 83.70999908447266%
Iteration: 132000, Loss: 0.616614580154419, Accuracy: 83.20999908447266%
Iteration: 132500, Loss: 0.3532946705818176, Accuracy: 83.43000030517578%
Iteration: 133000, Loss: 0.4209326207637787, Accuracy: 83.70999908447266%
Iteration: 133500, Loss: 0.45869219303131104, Accuracy: 84.33999633789062%
Iteration: 134000, Loss: 0.3076147437095642, Accuracy: 83.19000244140625%
Iteration: 134500, Loss: 0.19371548295021057, Accuracy: 83.56999969482422%
Iteration: 135000, Loss: 0.2911071181297302, Accuracy: 84.38999938964844%
Iteration: 135500, Loss: 0.3322262465953827, Accuracy: 82.19999694824219%
Iteration: 136000, Loss: 0.4935471713542938, Accuracy: 84.05000305175781%
Iteration: 136500, Loss: 0.2867492735385895, Accuracy: 84.08000183105469%
Iteration: 137000, Loss: 0.2083195298910141, Accuracy: 84.18000030517578%
Iteration: 137500, Loss: 0.40460994839668274, Accuracy: 83.76000213623047%
Iteration: 138000, Loss: 0.5483617782592773, Accuracy: 81.87000274658203%
Iteration: 138500, Loss: 0.4425499737262726, Accuracy: 83.23999786376953%
Iteration: 139000, Loss: 0.2834149897098541, Accuracy: 83.54000091552734%
Iteration: 139500, Loss: 0.38974037766456604, Accuracy: 84.45999908447266%
Iteration: 140000, Loss: 0.26345789432525635, Accuracy: 83.45999908447266%
Iteration: 140500, Loss: 0.44058680534362793, Accuracy: 84.38999938964844%
Iteration: 141000, Loss: 0.40848538279533386, Accuracy: 83.4800033569336%
Iteration: 141500, Loss: 0.3014581501483917, Accuracy: 84.12000274658203%
Iteration: 142000, Loss: 0.3034467399120331, Accuracy: 83.5999984741211%
Iteration: 142500, Loss: 0.5528811812400818, Accuracy: 83.80999755859375%
Iteration: 143000, Loss: 0.21474123001098633, Accuracy: 83.36000061035156%
Iteration: 143500, Loss: 0.24837471544742584, Accuracy: 82.81999969482422%
Iteration: 144000, Loss: 0.567377507686615, Accuracy: 83.05000305175781%
Iteration: 144500, Loss: 0.505003809928894, Accuracy: 83.58999633789062%
Iteration: 145000, Loss: 0.374724417924881, Accuracy: 83.3499984741211%

Iteration: 145500, Loss: 0.25919148325920105, Accuracy: 83.75%
Iteration: 146000, Loss: 0.2885187864303589, Accuracy: 82.68000030517578%
Iteration: 146500, Loss: 0.36414337158203125, Accuracy: 83.87999725341797%
Iteration: 147000, Loss: 0.43444305658340454, Accuracy: 82.16999816894531%
Iteration: 147500, Loss: 0.4750231802463531 5, Accuracy: 83.63999938964844%
Iteration: 148000, Loss: 0.40188896656036377, Accuracy: 83.54000091552734%
Iteration: 148500, Loss: 0.17747002840042114, Accuracy: 83.72000122070312%
Iteration: 149000, Loss: 0.43634548783302307, Accuracy: 83.0999984741211%
Iteration: 149500, Loss: 0.26762819290161133, Accuracy: 83.05000305175781%
Iteration: 150000, Loss: 0.6319757103919983, Accuracy: 83.61000061035156%
Iteration: 150500, Loss: 0.8159933686256409, Accuracy: 84.2699966430664%
Iteration: 151000, Loss: 0.25713998079299927, Accuracy: 82.79000091552734%
Iteration: 151500, Loss: 0.2505124807357788, Accuracy: 83.25%
Iteration: 152000, Loss: 0.47852763533592224, Accuracy: 83.12999725341797%
Iteration: 152500, Loss: 0.21219272911548615, Accuracy: 83.62999725341797%
Iteration: 153000, Loss: 0.33991196751594543, Accuracy: 82.81999969482422%
Iteration: 153500, Loss: 0.24736586213111877, Accuracy: 83.33000183105469%
Iteration: 154000, Loss: 0.396263986825943, Accuracy: 83.51000213623047%
Iteration: 154500, Loss: 0.392807275056839, Accuracy: 83.9000015258789%
Iteration: 155000, Loss: 0.2613295316696167, Accuracy: 80.98999786376953%
Iteration: 155500, Loss: 0.1512991338968277, Accuracy: 84.37999725341797%
Iteration: 156000, Loss: 0.38262712955474854, Accuracy: 82.91000366210938%
Iteration: 156500, Loss: 0.2102205604314804, Accuracy: 83.43000030517578%
Iteration: 157000, Loss: 0.4081282913684845, Accuracy: 82.66000366210938%
Iteration: 157500, Loss: 0.6372280716896057, Accuracy: 83.5999984741211%
Iteration: 158000, Loss: 0.16470997035503387, Accuracy: 83.33000183105469%
Iteration: 158500, Loss: 0.2516077756881714, Accuracy: 81.54000091552734%
Iteration: 159000, Loss: 0.496998131275177, Accuracy: 83.19999694824219%
Iteration: 159500, Loss: 0.24644863605499268, Accuracy: 83.98999786376953%
Iteration: 160000, Loss: 0.28458914160728455, Accuracy: 82.94000244140625%
Iteration: 160500, Loss: 0.26771780848503113, Accuracy: 83.80000305175781%
Iteration: 161000, Loss: 0.6110700368881226, Accuracy: 83.45999908447266%
Iteration: 161500, Loss: 0.17711366713047028, Accuracy: 83.44999694824219%
Iteration: 162000, Loss: 0.22262363135814667, Accuracy: 84.2300033569336%
Iteration: 162500, Loss: 0.30724066495895386, Accuracy: 83.6500015258789%
Iteration: 163000, Loss: 0.04916658252477646, Accuracy: 83.44000244140625%
Iteration: 163500, Loss: 0.3108535706996918, Accuracy: 83.4000015258789%
Iteration: 164000, Loss: 0.23842169344425201, Accuracy: 83.22000122070312%
Iteration: 164500, Loss: 0.6651309132575989, Accuracy: 83.58999633789062%
Iteration: 165000, Loss: 0.15058039128780365, Accuracy: 82.52999877929688%
Iteration: 165500, Loss: 0.36667102575302124, Accuracy: 83.62999725341797%
Iteration: 166000, Loss: 0.3420976400375366, Accuracy: 83.62999725341797%
Iteration: 166500, Loss: 0.17737054824829102, Accuracy: 83.56999969482422%
Iteration: 167000, Loss: 0.20291493833065033, Accuracy: 83.69999694824219%
Iteration: 167500, Loss: 0.5418334007263184, Accuracy: 83.7699966430664%
Iteration: 168000, Loss: 0.335311621427536, Accuracy: 81.55999755859375%
Iteration: 168500, Loss: 0.23483005166053772, Accuracy: 82.70999908447266%

Iteration: 169000, Loss: 0.785353422164917, Accuracy: 84.06999969482422%
Iteration: 169500, Loss: 0.219070166349411, Accuracy: 83.94999694824219%
Iteration: 170000, Loss: 0.22469772398471832, Accuracy: 83.5199966430664%
Iteration: 170500, Loss: 0.409658819437027, Accuracy: 84.11000061035156%
Iteration: 171000, Loss: 0.5425499677658081, Accuracy: 82.19000244140625%
Iteration: 171500, Loss: 0.29179221391677856, Accuracy: 83.08999633789062%
Iteration: 172000, Loss: 0.17382650077342987, Accuracy: 83.16000366210938%
Iteration: 172500, Loss: 0.6815604567527771, Accuracy: 83.62000274658203%
Iteration: 173000, Loss: 0.23285533487796783, Accuracy: 81.83999633789062%
Iteration: 173500, Loss: 0.2947159707546234, Accuracy: 80.88999938964844%
Iteration: 174000, Loss: 0.40750041604042053, Accuracy: 83.25%
Iteration: 174500, Loss: 0.33002281188964844, Accuracy: 83.55000305175781%
Iteration: 175000, Loss: 0.2694067060947418, Accuracy: 83.87000274658203%
Iteration: 175500, Loss: 0.1962227076292038, Accuracy: 84.18000030517578%
Iteration: 176000, Loss: 0.2044074833393097, Accuracy: 83.69000244140625%
Iteration: 176500, Loss: 0.2063823938369751, Accuracy: 82.83999633789062%
Iteration: 177000, Loss: 0.2574367821216583, Accuracy: 81.75%
Iteration: 177500, Loss: 0.22588175535202026, Accuracy: 83.58000183105469%
Iteration: 178000, Loss: 0.584827184677124, Accuracy: 81.63999938964844%
Iteration: 178500, Loss: 0.2749483287334442, Accuracy: 82.91999816894531%
Iteration: 179000, Loss: 0.5337590575218201, Accuracy: 82.23999786376953%
Iteration: 179500, Loss: 0.5469070672988892, Accuracy: 83.80999755859375%
Iteration: 180000, Loss: 0.29110756516456604, Accuracy: 83.9000015258789%