Pytorch for Deep Learning

Content:

Basics of Pytorch

Matrices

- In pytorch, matrix(array) is called tensors.
- 3*3 matrix koy. This is 3x3 tensor.
- Lets look at array example with numpy that we already know.
    - We create numpy array with np.numpy() method
    - Type(): type of the array. In this example it is numpy
    - np.shape(): shape of the array. Row x Column

In [2]:
```
# import numpy library
import numpy as np

# numpy array
array = [[1,2,3],[4,5,6]]
first_array = np.array(array) # 2x3 array
print("Array Type: {}".format(type(first_array))) # type
print("Array Shape: {}".format(np.shape(first_array))) # shape
print(first_array)
```

```
Array Type: <class 'numpy.ndarray'>
Array Shape: (2, 3)
[[1 2 3]
 [4 5 6]]
```

In [3]:
```
# import pytorch library
import torch

# pytorch array
tensor = torch.Tensor(array)
print("Array Type: {}".format(tensor.type)) # type
print("Array Shape: {}".format(tensor.shape)) # shape
print(tensor)
```

```
Array Type: <built-in method type of Tensor object at 0x7896ff66d630>
Array Shape: torch.Size([2, 3])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

In [4]:
```
# numpy ones
print("Numpy {}\n".format(np.ones((2,3))))

# pytorch ones
print(torch.ones((2,3)))
```

```
Numpy [[1. 1. 1.]
 [1. 1. 1.]]

tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

In [5]:
```
# numpy random
print("Numpy {}\n".format(np.random.rand(2,3)))

# pytorch random
print(torch.rand(2,3))
```

```
Numpy [[0.15976649 0.56928831 0.70437698]
 [0.528048   0.86208399 0.11570357]]

tensor([[0.9693, 0.6065, 0.2636],
        [0.1159, 0.9310, 0.1479]])
```

- Even if when I use pytorch for neural networks, I feel better if I use numpy. Therefore, usually convert result of neural network that is tensor to numpy array to visualize or examine.
- Lets look at conversion between tensor and numpy arrays.
    - torch.from_numpy(): from numpy to tensor
    - numpy(): from tensor to numpy

In [6]:
```
# random numpy array
array = np.random.rand(2,2)
print("{} {}\n".format(type(array),array))

# from numpy to tensor
from_numpy_to_tensor = torch.from_numpy(array)
print("{}\n".format(from_numpy_to_tensor))

# from tensor to numpy
tensor = from_numpy_to_tensor
from_tensor_to_numpy = tensor.numpy()
print("{} {}\n".format(type(from_tensor_to_numpy),from_tensor_to_numpy))
```

```
<class 'numpy.ndarray'> [[0.7089765  0.97180698]
 [0.00673483 0.33032512]]

tensor([[0.7090, 0.9718],
        [0.0067, 0.3303]], dtype=torch.float64)

<class 'numpy.ndarray'> [[0.7089765  0.97180698]
 [0.00673483 0.33032512]]
```

Basic Math with Pytorch

- Resize: view()
- a and b are tensor.
- Addition: torch.add(a,b) = a + b
- Subtraction: a.sub(b) = a - b
- Element wise multiplication: torch.mul(a,b) = a * b
- Element wise division: torch.div(a,b) = a / b
- Mean: a.mean()
- Standart Deviation (std): a.std()

In [7]:
```
# create tensor
tensor = torch.ones(3,3)
print("\n",tensor)

# Resize
print("{}{}\n".format(tensor.view(9).shape,tensor.view(9)))

# Addition
print("Addition: {}\n".format(torch.add(tensor,tensor)))

# Subtraction
print("Subtraction: {}\n".format(tensor.sub(tensor)))

# Element wise multiplication
print("Element wise multiplication: {}\n".format(torch.mul(tensor,tensor)))

# Element wise division
print("Element wise division: {}\n".format(torch.div(tensor,tensor)))

# Mean
tensor = torch.Tensor([1,2,3,4,5])
print("Mean: {}".format(tensor.mean()))

# Standart deviation (std)
print("std: {}".format(tensor.std()))
```

```
 tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
torch.Size([9])tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])

Addition: tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])

Subtraction: tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])

Element wise multiplication: tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])

Element wise division: tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])

Mean: 3.0
std: 1.5811388492584229
```

Variables

- It accumulates gradients.
- We will use pytorch in neural network. And as you know, in neural network we have backpropagation where gradients are calculated. Therefore we need to handle gradients.

- Difference between variables and tensor is variable accumulates gradients.
- We can make math operations with variables, too.
- In order to make backward propagation we need variables

In [8]:
```
# import variable from pytorch library
from torch.autograd import Variable

# define variable
var = Variable(torch.ones(3), requires_grad = True)
var
```

Out[8]:
```
tensor([1., 1., 1.], requires_grad=True)
```

- Assume we have equation y = x^2
- Define x = [2,4] variable
- After calculation we find that y = [4,16] (y = x^2)
- Recap o equation is that o = (1/2)*sum(y)* = *(1/2)*sum(x^2)
- deriavative of o = x
- Result is equal to x so gradients are [2,4]
- Lets implement

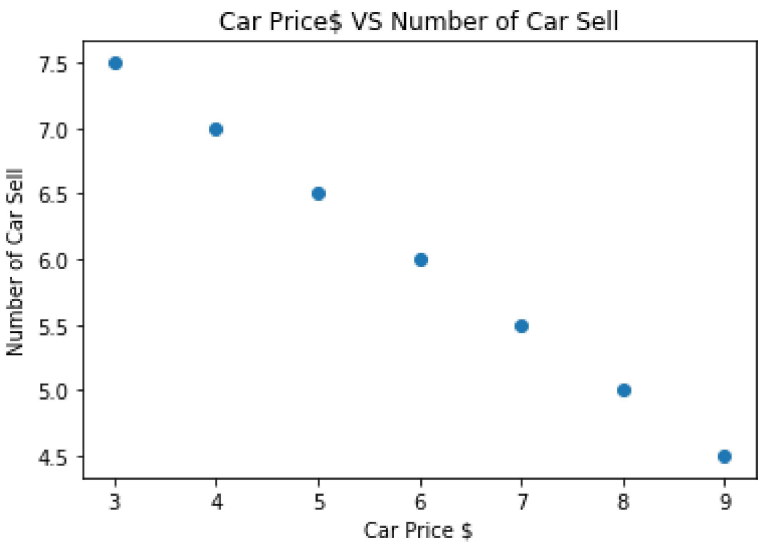
In [9]:
```
# lets make basic backward propagation
# we have an equation that is y = x^2
array = [2,4]
tensor = torch.Tensor(array)
x = Variable(tensor, requires_grad = True)
y = x**2
print(" y =  ",y)

# recap o equation o = 1/2*sum(y)
o = (1/2)*sum(y)
print(" o =  ",o)

# backward
o.backward() # calculates gradients

# As I defined, variables accumulates gradients. In this part there is only one variable x.
# Therefore variable x should be have gradients
# Lets look at gradients with x.grad
print("gradients: ",x.grad)
```

```
 y =   tensor([ 4., 16.], grad_fn=<PowBackward0>)
 o =   tensor(10., grad_fn=<MulBackward0>)
gradients:  tensor([2., 4.])
```

Linear Regression

- y = Ax + B.
  - A = slope of curve
  - B = bias (point that intersect y-axis)
- For example, we have car company. If the car price is low, we sell more car. If the car price is high, we sell less car. This is the fact that we know and we have data set about this fact.
- The question is that what will be number of car sell if the car price is 100.

In [10]:
```
# As a car company we collect this data from previous selling
# lets define car prices
car_prices_array = [3,4,5,6,7,8,9]
car_price_np = np.array(car_prices_array,dtype=np.float32)
car_price_np = car_price_np.reshape(-1,1)
car_price_tensor = Variable(torch.from_numpy(car_price_np))

# lets define number of car sell
number_of_car_sell_array = [ 7.5, 7, 6.5, 6.0, 5.5, 5.0, 4.5]
number_of_car_sell_np = np.array(number_of_car_sell_array,dtype=np.float32)
number_of_car_sell_np = number_of_car_sell_np.reshape(-1,1)
number_of_car_sell_tensor = Variable(torch.from_numpy(number_of_car_sell_np))

# lets visualize our data
import matplotlib.pyplot as plt
plt.scatter(car_prices_array,number_of_car_sell_array)
plt.xlabel("Car Price $")
plt.ylabel("Number of Car Sell")
plt.title("Car Price$ VS Number of Car Sell")
plt.show()
```

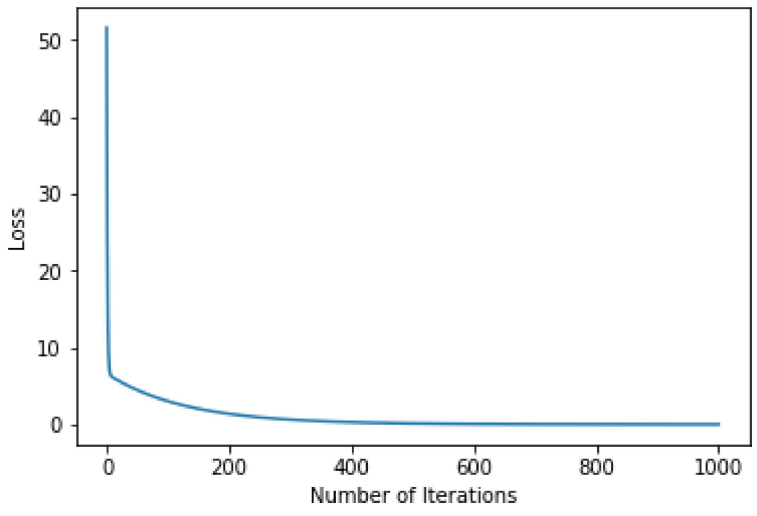


- Now this plot is our collected data
- We have a question that is what will be number of car sell if the car price is 100$
- In order to solve this question we need to use linear regression.
- We need to line fit into this data. Aim is fitting line with minimum error.
- **Steps of Linear Regression**
    1. create LinearRegression class
    2. define model from this LinearRegression class
    3. MSE: Mean squared error
    4. Optimization (SGD:stochastic gradient descent)
    5. Backpropagation
    6. Prediction
- Lets implement it with Pytorch

In [11]:
```
# Linear Regression with Pytorch

# libraries
import torch
from torch.autograd import Variable
import torch.nn as nn
import warnings
warnings.filterwarnings("ignore")

# create class
class LinearRegression(nn.Module):
    def __init__(self,input_size,output_size):
        # super function. It inherits from nn.Module and we can access everythink in nn.Module
        super(LinearRegression,self).__init__()
        # Linear function.
        self.linear = nn.Linear(input_dim,output_dim)

    def forward(self,x):
        return self.linear(x)

# define model
input_dim = 1
output_dim = 1
model = LinearRegression(input_dim,output_dim) # input and output size are 1

# MSE
mse = nn.MSELoss()

# Optimization (find parameters that minimize error)
learning_rate = 0.02   # how fast we reach best parameters
optimizer = torch.optim.SGD(model.parameters(),lr = learning_rate)

# train model
loss_list = []
iteration_number = 1001
for iteration in range(iteration_number):

    # optimization
    optimizer.zero_grad()

    # Forward to get output
    results = model(car_price_tensor)

    # Calculate Loss
    loss = mse(results, number_of_car_sell_tensor)

    # backward propagation
    loss.backward()

    # Updating parameters
    optimizer.step()

    # store loss
    loss_list.append(loss.data)

    # print loss
    if(iteration % 50 == 0):
        print('epoch {}, loss {}'.format(iteration, loss.data))
```

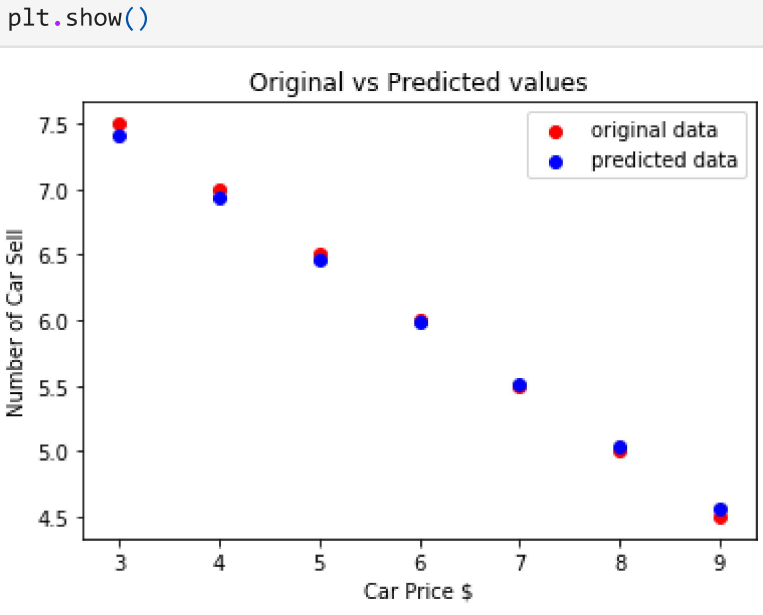
```
plt.plot(range(iteration_number),loss_list)
plt.xlabel("Number of Iterations")
plt.ylabel("Loss")
plt.show()
```

```
epoch 0, loss 51.623226165771484
epoch 50, loss 4.478211402893066
epoch 100, loss 3.0261149406433105
epoch 150, loss 2.044872283935547
epoch 200, loss 1.3818049430847168
epoch 250, loss 0.9337430000305176
epoch 300, loss 0.6309698820114136
epoch 350, loss 0.42637336254119873
epoch 400, loss 0.28811806440353394
epoch 450, loss 0.1946932077407837
epoch 500, loss 0.13156220316886902
epoch 550, loss 0.08890160918235779
epoch 600, loss 0.060074903070926666
epoch 650, loss 0.040595151484012604
epoch 700, loss 0.027431808412075043
epoch 750, loss 0.018536604940891266
epoch 800, loss 0.012525811791419983
epoch 850, loss 0.008464040234684944
epoch 900, loss 0.005719634238630533
epoch 950, loss 0.003864944468319416
epoch 1000, loss 0.002611754694953561
```



- Number of iteration is 1001.
- Loss is almost zero that you can see from plot or loss in epoch number 1000.
- Now we have a trained model.
- While usign trained model, lets predict car prices.

In [12]:
```
# predict our car price
predicted = model(car_price_tensor).data.numpy()
plt.scatter(car_prices_array,number_of_car_sell_array,label = "original data",color ="red")
plt.scatter(car_prices_array,predicted,label = "predicted data",color ="blue")

# predict if car price is 10$, what will be the number of car sell
#predicted_10 = model(torch.from_numpy(np.array([10]))).data.numpy()
#plt.scatter(10,predicted_10.data,label = "car price 10$",color ="green")
plt.legend()
plt.xlabel("Car Price $")
plt.ylabel("Number of Car Sell")
plt.title("Original vs Predicted values")
plt.show()
```



Logistic Regression

- Linear regression is not good at classification.
- We use logistic regression for classification.
- linear regression + logistic function(softmax) = logistic regression

- **Steps of Logistic Regression**

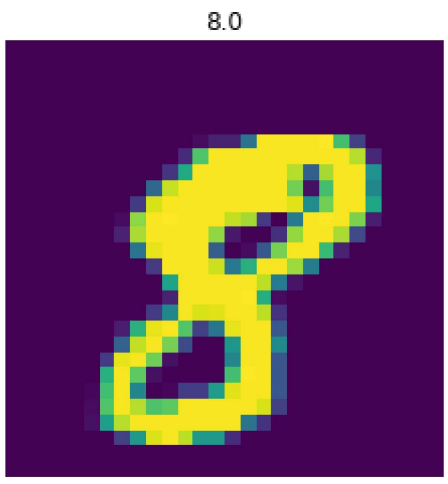
1. Import Libraries
2. Prepare Dataset
    - We use MNIST dataset.
    - There are 28*28 images and 10 labels from 0 to 9
    - Data is not normalized so we divide each image to 255 that is basic normalization for images.
    - In order to split data, we use train_test_split method from sklearn library
    - Size of train data is 80% and size of test data is 20%.
    - Create feature and target tensors. At the next parts we create variable from these tensors. As you remember we need to define variable for accumulation of gradients.
    - batch_size = batch size means is that for example we have data and it includes 1000 sample. We can train 1000 sample in a same time or we can divide it 10 groups which include 100 sample and train 10 groups in order. Batch size is the group size. For example, I choose batch_size = 100, that means in order to train all data only once we have 336 groups. We train each groups(336) that have batch_size(quota) 100. Finally we train 33600 sample one time.
    - epoch: 1 epoch means training all samples one time.
    - In our example: we have 33600 sample to train and we decide our batch_size is 100. Also we decide epoch is 29(accuracy achieves almost highest value when epoch is 29). Data is trained 29 times. Question is that how many iteration do I need? Lets calculate:
        - training data 1 times = training 33600 sample (because data includes 33600 sample)
        - But we split our data 336 groups(group_size = batch_size = 100) our data
        - Therefore, 1 epoch(training data only once) takes 336 iteration
        - We have 29 epoch, so total iterarion is 9744(that is almost 10000 which I used)
    - TensorDataset(): Data set wrapping tensors. Each sample is retrieved by indexing tensors along the first dimension.
    - DataLoader(): It combines dataset and sample. It also provides multi process iterators over the dataset.
    - Visualize one of the images in dataset
3. Create Logistic Regression Model
    - Same with linear regression.
    - However as you expect, there should be logistic function in model right?
    - In pytorch, logistic function is in the loss function where we will use at next parts.
4. Instantiate Model
    - input_dim = 28*28 # size of image px*px
    - output_dim = 10 # labels 0,1,2,3,4,5,6,7,8,9
    - create model
5. Instantiate Loss
    - Cross entropy loss
    - It calculates loss that is not surprise :)
    - It also has softmax(logistic function) in it.
6. Instantiate Optimizer
    - SGD Optimizer
7. Traning the Model
8. Prediction

- As a result, as you can see from plot, while loss decreasing, accuracy(almost 85%) is increasing and our model is learning(training).

In [13]:
```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import DataLoader
import pandas as pd
from sklearn.model_selection import train_test_split
```

In [14]:
```
# Prepare Dataset
# load data
train = pd.read_csv(r"../input/train.csv",dtype = np.float32)

# split data into features(pixels) and labels(numbers from 0 to 9)
targets_numpy = train.label.values
features_numpy = train.loc[:,train.columns != "label"].values/255 # normalization

# train test split. Size of train data is 80% and size of test data is 20%.
features_train, features_test, targets_train, targets_test = train_test_split(features_numpy,
                                                                               targets_numpy,
                                                                               test_size = 0.2,
                                                                               random_state = 42)

# create feature and targets tensor for train set. As you remember we need variable to accumulate gradients. Therefore j
featuresTrain = torch.from_numpy(features_train)
targetsTrain = torch.from_numpy(targets_train).type(torch.LongTensor) # data type is long

# create feature and targets tensor for test set.
featuresTest = torch.from_numpy(features_test)
targetsTest = torch.from_numpy(targets_test).type(torch.LongTensor) # data type is long

# batch_size, epoch and iteration
batch_size = 100
n_iters = 10000
num_epochs = n_iters / (len(features_train) / batch_size)
num_epochs = int(num_epochs)

# Pytorch train and test sets
```

```
train = torch.utils.data.TensorDataset(featuresTrain,targetsTrain)
test = torch.utils.data.TensorDataset(featuresTest,targetsTest)

# data loader
train_loader = DataLoader(train, batch_size = batch_size, shuffle = False)
test_loader = DataLoader(test, batch_size = batch_size, shuffle = False)

# visualize one of the images in data set
plt.imshow(features_numpy[10].reshape(28,28))
plt.axis("off")
plt.title(str(targets_numpy[10]))
plt.savefig('graph.png')
plt.show()
```



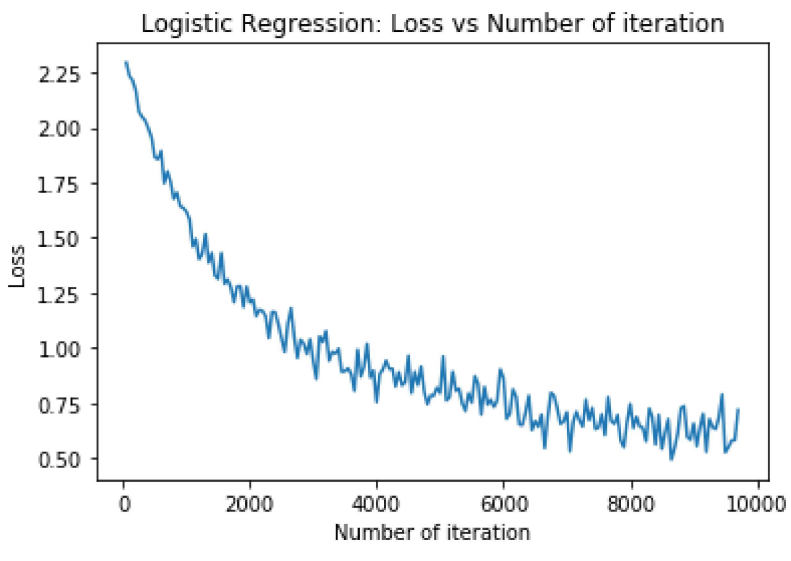
In [15]:
```
# Create Logistic Regression Model
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LogisticRegressionModel, self).__init__()
        # Linear part
        self.linear = nn.Linear(input_dim, output_dim)
        # There should be logistic function right?
        # However logistic function in pytorch is in loss function
        # So actually we do not forget to put it, it is only at next parts

    def forward(self, x):
        out = self.linear(x)
        return out

# Instantiate Model Class
input_dim = 28*28 # size of image px*px
output_dim = 10  # labels 0,1,2,3,4,5,6,7,8,9

# create logistic regression model
model = LogisticRegressionModel(input_dim, output_dim)

# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# SGD Optimizer
learning_rate = 0.001
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

In [16]:
```
# Traning the Model
count = 0
loss_list = []
iteration_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        # Define variables
        train = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # Clear gradients
        optimizer.zero_grad()

        # Forward propagation
        outputs = model(train)

        # Calculate softmax and cross entropy loss
        loss = error(outputs, labels)

        # Calculate gradients
        loss.backward()

        # Update parameters
        optimizer.step()

        count += 1

        # Prediction
        if count % 50 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Predict test dataset
            for images, labels in test_loader:
                test = Variable(images.view(-1, 28*28))
```

```
            # Forward propagation
            outputs = model(test)

            # Get predictions from the maximum value
            predicted = torch.max(outputs.data, 1)[1]

            # Total number of labels
            total += len(labels)

            # Total correct predictions
            correct += (predicted == labels).sum()

        accuracy = 100 * correct / float(total)

        # store loss and iteration
        loss_list.append(loss.data)
        iteration_list.append(count)
    if count % 500 == 0:
        # Print Loss
        print('Iteration: {}  Loss: {}  Accuracy: {}%'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 1.86466646194458  Accuracy: 68%
Iteration: 1000  Loss: 1.618392825126648  Accuracy: 76%
Iteration: 1500  Loss: 1.3139762878417969  Accuracy: 78%
Iteration: 2000  Loss: 1.2069573402404785  Accuracy: 80%
Iteration: 2500  Loss: 1.0424375534057617  Accuracy: 81%
Iteration: 3000  Loss: 0.9390164017677307  Accuracy: 82%
Iteration: 3500  Loss: 0.8956833481788635  Accuracy: 82%
Iteration: 4000  Loss: 0.7555015683174133  Accuracy: 83%
Iteration: 4500  Loss: 0.9651548266410828  Accuracy: 83%
Iteration: 5000  Loss: 0.7957883477210999  Accuracy: 84%
Iteration: 5500  Loss: 0.7530776858329773  Accuracy: 84%
Iteration: 6000  Loss: 0.8636155724525452  Accuracy: 84%
Iteration: 6500  Loss: 0.6697360277175903  Accuracy: 84%
Iteration: 7000  Loss: 0.7095925807952881  Accuracy: 85%
Iteration: 7500  Loss: 0.6400054097175598  Accuracy: 85%
Iteration: 8000  Loss: 0.7464028000831604  Accuracy: 85%
Iteration: 8500  Loss: 0.5430745482444763  Accuracy: 85%
Iteration: 9000  Loss: 0.658008873462677  Accuracy: 85%
Iteration: 9500  Loss: 0.5279005169868469  Accuracy: 85%
```

In [17]:
```
# visualization
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("Logistic Regression: Loss vs Number of iteration")
plt.show()
```



Artificial Neural Network (ANN)

- Logistic regression is good at classification but when complexity(non linearity) increases, the accuracy of model decreases.
- Therefore, we need to increase complexity of model.
- In order to increase complexity of model, we need to add more non linear functions as hidden layer.

- What we expect from artificial neural network is that when complexity increases, we use more hidden layers and our model can adapt better. As a result accuracy increase.

- **Steps of ANN:**
  1. Import Libraries
     - In order to show you, I import again but we actually imported them at previous parts.
  2. Prepare Dataset
     - Totally same with previous part(logistic regression).
     - We use same dataset so we only need train_loader and test_loader.
     - We use same batch size, epoch and iteration numbers.
  3. Create ANN Model
     - We add 3 hidden layers.
     - We use ReLU, Tanh and ELU activation functions for diversity.
  4. Instantiate Model Class

        - input_dim = 28*28 # *size of image px*px
        - output_dim = 10 # labels 0,1,2,3,4,5,6,7,8,9
        - Hidden layer dimension is 150. I only choose it as 150 there is no reason. Actually hidden layer dimension is hyperparameter and it should be chosen and tuned. You can try different values for hidden layer dimension and observe the results.
        - create model
    5. Instantiate Loss
        - Cross entropy loss
        - It also has softmax(logistic function) in it.
    6. Instantiate Optimizer
        - SGD Optimizer
    7. Traning the Model
    8. Prediction
- As a result, as you can see from plot, while loss decreasing, accuracy is increasing and our model is learning(training).
- Thanks to hidden layers model learnt better and accuracy(almost 95%) is better than accuracy of logistic regression model.

In [18]:
```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
```

In [19]:
```
# Create ANN Model
class ANNModel(nn.Module):

    def __init__(self, input_dim, hidden_dim, output_dim):
        super(ANNModel, self).__init__()

        # Linear function 1: 784 --> 150
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity 1
        self.relu1 = nn.ReLU()

        # Linear function 2: 150 --> 150
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        # Non-linearity 2
        self.tanh2 = nn.Tanh()

        # Linear function 3: 150 --> 150
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        # Non-linearity 3
        self.elu3 = nn.ELU()

        # Linear function 4 (readout): 150 --> 10
        self.fc4 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Linear function 1
        out = self.fc1(x)
        # Non-linearity 1
        out = self.relu1(out)

        # Linear function 2
        out = self.fc2(out)
        # Non-linearity 2
        out = self.tanh2(out)

        # Linear function 2
        out = self.fc3(out)
        # Non-linearity 2
        out = self.elu3(out)

        # Linear function 4 (readout)
        out = self.fc4(out)
        return out

# instantiate ANN
input_dim = 28*28
hidden_dim = 150 #hidden layer dim is one of the hyper parameter and it should be chosen and tuned. For now I only say i
output_dim = 10

# Create ANN
model = ANNModel(input_dim, hidden_dim, output_dim)

# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# SGD Optimizer
learning_rate = 0.02
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

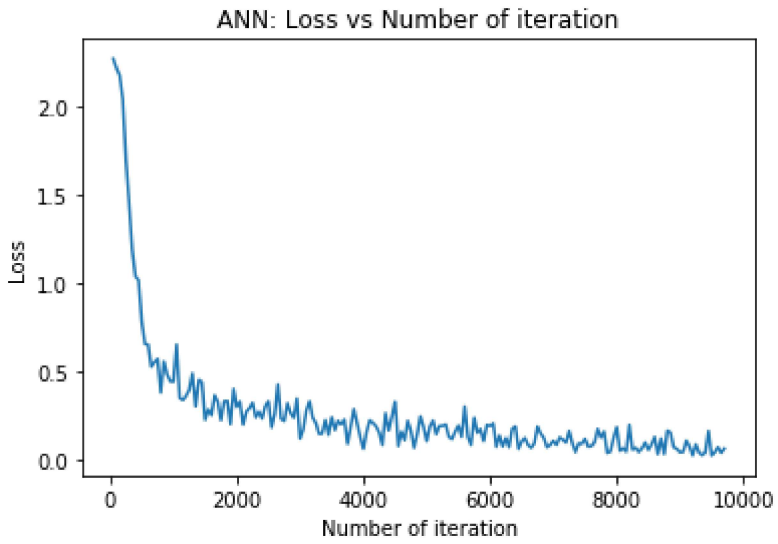
In [20]:
```
# ANN model training
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
```

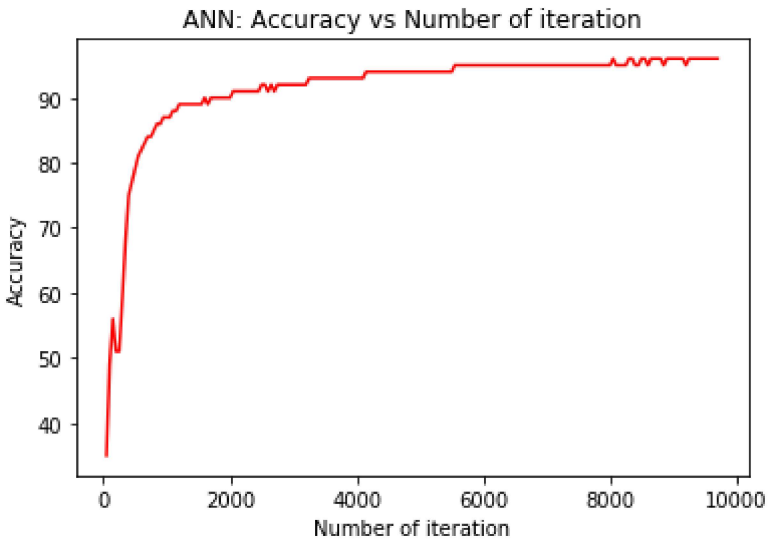
```
        train = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # Clear gradients
        optimizer.zero_grad()

        # Forward propagation
        outputs = model(train)

        # Calculate softmax and ross entropy loss
        loss = error(outputs, labels)

        # Calculating gradients
        loss.backward()

        # Update parameters
        optimizer.step()

        count += 1

        if count % 50 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Predict test dataset
            for images, labels in test_loader:

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

                # Forward propagation
                outputs = model(test)

                # Get predictions from the maximum value
                predicted = torch.max(outputs.data, 1)[1]

                # Total number of labels
                total += len(labels)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 0.7841647863388062  Accuracy: 79 %
Iteration: 1000  Loss: 0.44362664222717285  Accuracy: 87 %
Iteration: 1500  Loss: 0.2270955592393875  Accuracy: 89 %
Iteration: 2000  Loss: 0.2995416224002838  Accuracy: 90 %
Iteration: 2500  Loss: 0.3309236466884613  Accuracy: 92 %
Iteration: 3000  Loss: 0.12039276957511902  Accuracy: 92 %
Iteration: 3500  Loss: 0.24051378667354584  Accuracy: 93 %
Iteration: 4000  Loss: 0.061754897236824036  Accuracy: 93 %
Iteration: 4500  Loss: 0.32856684923171997  Accuracy: 94 %
Iteration: 5000  Loss: 0.10621517896652222  Accuracy: 94 %
Iteration: 5500  Loss: 0.19252893328666687  Accuracy: 94 %
Iteration: 6000  Loss: 0.19084887206554413  Accuracy: 95 %
Iteration: 6500  Loss: 0.10210537910461426  Accuracy: 95 %
Iteration: 7000  Loss: 0.10722008347511292  Accuracy: 95 %
Iteration: 7500  Loss: 0.11853181570768356  Accuracy: 95 %
Iteration: 8000  Loss: 0.18384100496768951  Accuracy: 95 %
Iteration: 8500  Loss: 0.05575636774301529  Accuracy: 96 %
Iteration: 9000  Loss: 0.04267650097608566  Accuracy: 96 %
Iteration: 9500  Loss: 0.02393564023077488  Accuracy: 96 %
```

In [21]:

```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("ANN: Loss vs Number of iteration")
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("ANN: Accuracy vs Number of iteration")
plt.show()
```





Convolutional Neural Network (CNN)

- CNN is well adapted to classify images.
- **Steps of CNN:**
    1. Import Libraries
    2. Prepare Dataset
        - Totally same with previous parts.
        - We use same dataset so we only need train_loader and test_loader.
    3. Convolutional layer:
        - Create feature maps with filters(kernels).
        - Padding: After applying filter, dimensions of original image decreases. However, we want to preserve as much as information about the original image. We can apply padding to increase dimension of feature map after convolutional layer.
        - We use 2 convolutional layer.
        - Number of feature map is out_channels = 16
        - Filter(kernel) size is 5*5
    4. Pooling layer:
        - Prepares a condensed feature map from output of convolutional layer(feature map)
        - 2 pooling layer that we will use max pooling.
        - Pooling size is 2*2
    5. Flattening: Flats the features map
    6. Fully Connected Layer:
        - Artificial Neural Network that we learnt at previous part.
        - Or it can be only linear like logistic regression but at the end there is always softmax function.
        - We will not use activation function in fully connected layer.
        - You can think that our fully connected layer is logistic regression.
        - We combine convolutional part and logistic regression to create our CNN model.
    7. Instantiate Model Class
        - create model
    8. Instantiate Loss
        - Cross entropy loss
        - It also has softmax(logistic function) in it.
    9. Instantiate Optimizer
        - SGD Optimizer
    10. Traning the Model
    11. Prediction
- As a result, as you can see from plot, while loss decreasing, accuracy is increasing and our model is learning(training).
- Thanks to convolutional layer, model learnt better and accuracy(almost 98%) is better than accuracy of ANN. Actually while tuning hyperparameters, increase in iteration and expanding convolutional neural network can increase accuracy but it takes too much running time that we do not want.

In [22]:
```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
```

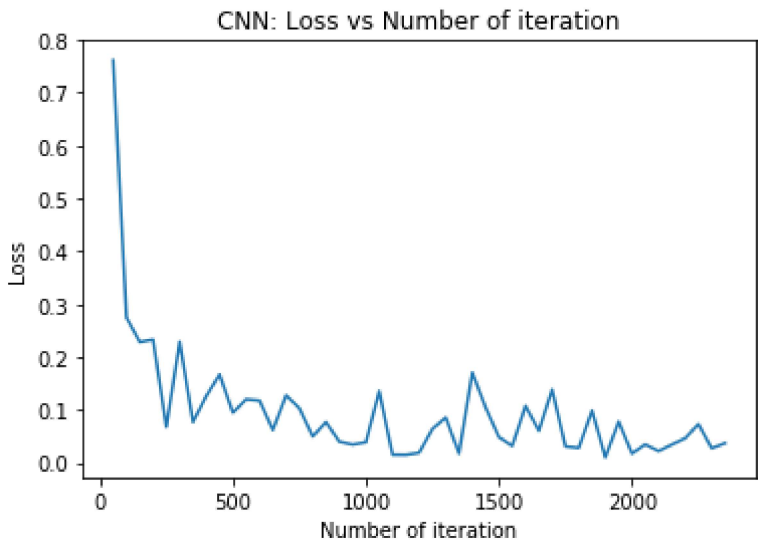
In [23]:
```
# Create CNN Model
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolution 1
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # Fully connected 1
        self.fc1 = nn.Linear(32 * 4 * 4, 10)

    def forward(self, x):
        # Convolution 1
        out = self.cnn1(x)
        out = self.relu1(out)

        # Max pool 1
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)

        # flatten
        out = out.view(out.size(0), -1)

        # Linear function (readout)
        out = self.fc1(out)

        return out

# batch_size, epoch and iteration
batch_size = 100
n_iters = 2500
num_epochs = n_iters / (len(features_train) / batch_size)
num_epochs = int(num_epochs)

# Pytorch train and test sets
train = torch.utils.data.TensorDataset(featuresTrain,targetsTrain)
test = torch.utils.data.TensorDataset(featuresTest,targetsTest)

# data loader
train_loader = torch.utils.data.DataLoader(train, batch_size = batch_size, shuffle = False)
test_loader = torch.utils.data.DataLoader(test, batch_size = batch_size, shuffle = False)

# Create CNN
model = CNNModel()

# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# SGD Optimizer
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

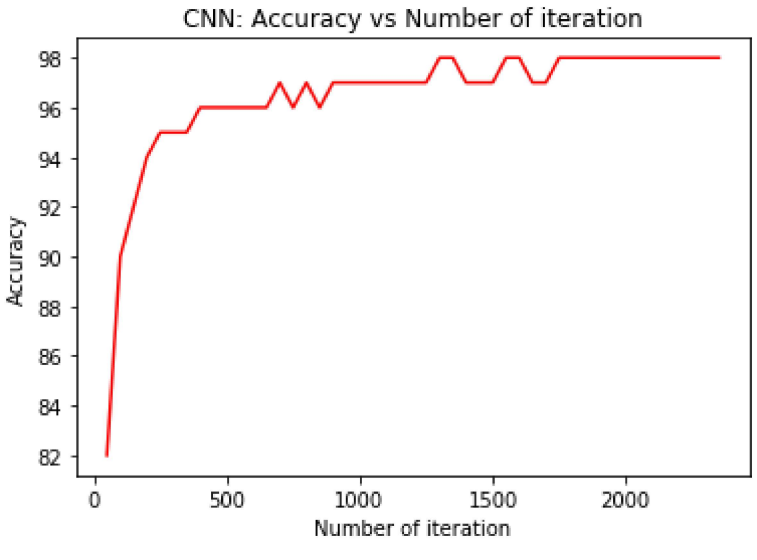
In [24]:
```
# CNN model training
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        train = Variable(images.view(100,1,28,28))
        labels = Variable(labels)

        # Clear gradients
        optimizer.zero_grad()

        # Forward propagation
        outputs = model(train)

        # Calculate softmax and ross entropy loss
        loss = error(outputs, labels)
```

```
        # Calculating gradients
        loss.backward()

        # Update parameters
        optimizer.step()

        count += 1

        if count % 50 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                test = Variable(images.view(100,1,28,28))

                # Forward propagation
                outputs = model(test)

                # Get predictions from the maximum value
                predicted = torch.max(outputs.data, 1)[1]

                # Total number of labels
                total += len(labels)

                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 0.09469277411699295  Accuracy: 96 %
Iteration: 1000  Loss: 0.03887723386287689  Accuracy: 97 %
Iteration: 1500  Loss: 0.04873870685696602  Accuracy: 97 %
Iteration: 2000  Loss: 0.01742550916969776  Accuracy: 98 %
```

In [25]:
```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.show()
```





Conclusion

we learn:

1. Basics of pytorch
2. Linear regression with pytorch
3. Logistic regression with pytorch
4. Artificial neural network with with pytorch
5. Convolutional neural network with pytorch