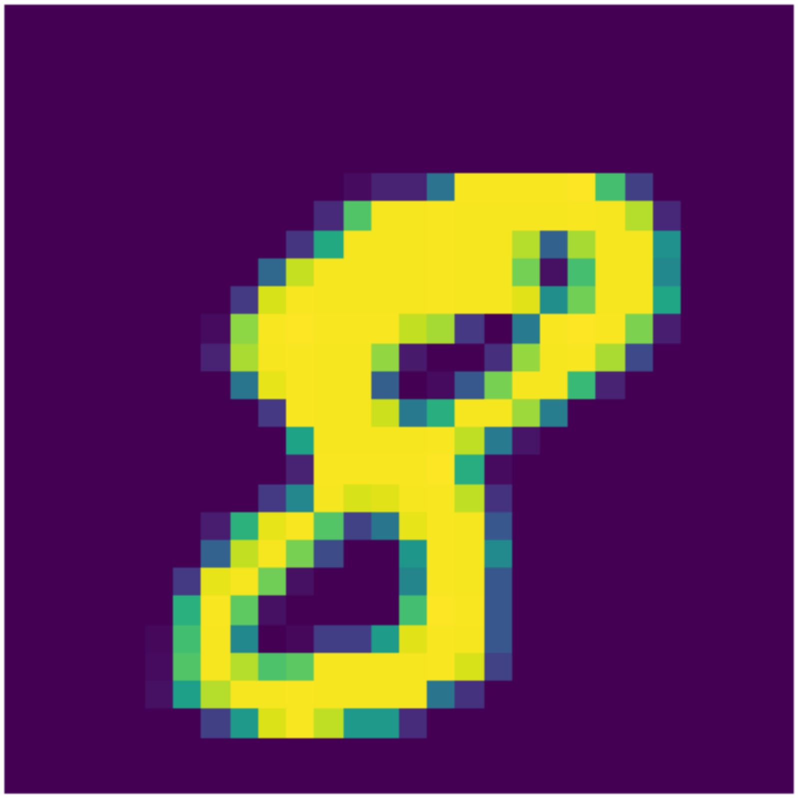
Recurrent Neural Network with Pytorch

In [5]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will l

import os
print(os.listdir("/Users/jagtarsinghmatharu/Desktop/Deep learning/assignment

# Any results you write to the current directory are saved as output.
```

```
['test.csv', 'train.csv', 'sample_submission.csv']
```

In [6]:
```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
```

In [8]:
```
# Import Libraries
train = pd.read_csv(r"/Users/jagtarsinghmatharu/Desktop/Deep learning/assign

# 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 # normaliz

# 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_spli




# create feature and targets tensor for train set. As you remember we need v
featuresTrain = torch.from_numpy(features_train)
targetsTrain = torch.from_numpy(targets_train).type(torch.LongTensor) # data

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

# 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 = TensorDataset(featuresTrain,targetsTrain)
test = 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 [9]:

```
# Create RNN Model
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        # RNN
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):

        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

```
# batch_size, epoch and iteration
batch_size = 100
n_iters = 8000
num_epochs = n_iters / (len(features_train) / batch_size)
num_epochs = int(num_epochs)

# Pytorch train and test sets
train = TensorDataset(featuresTrain,targetsTrain)
test = 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)

# Create RNN
input_dim = 28    # input dimension
hidden_dim = 100  # hidden layer dimension
layer_dim = 1     # number of hidden layers
output_dim = 10   # output dimension

model = RNNModel(input_dim, hidden_dim, layer_dim, output_dim)

# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# SGD Optimizer
learning_rate = 0.05
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

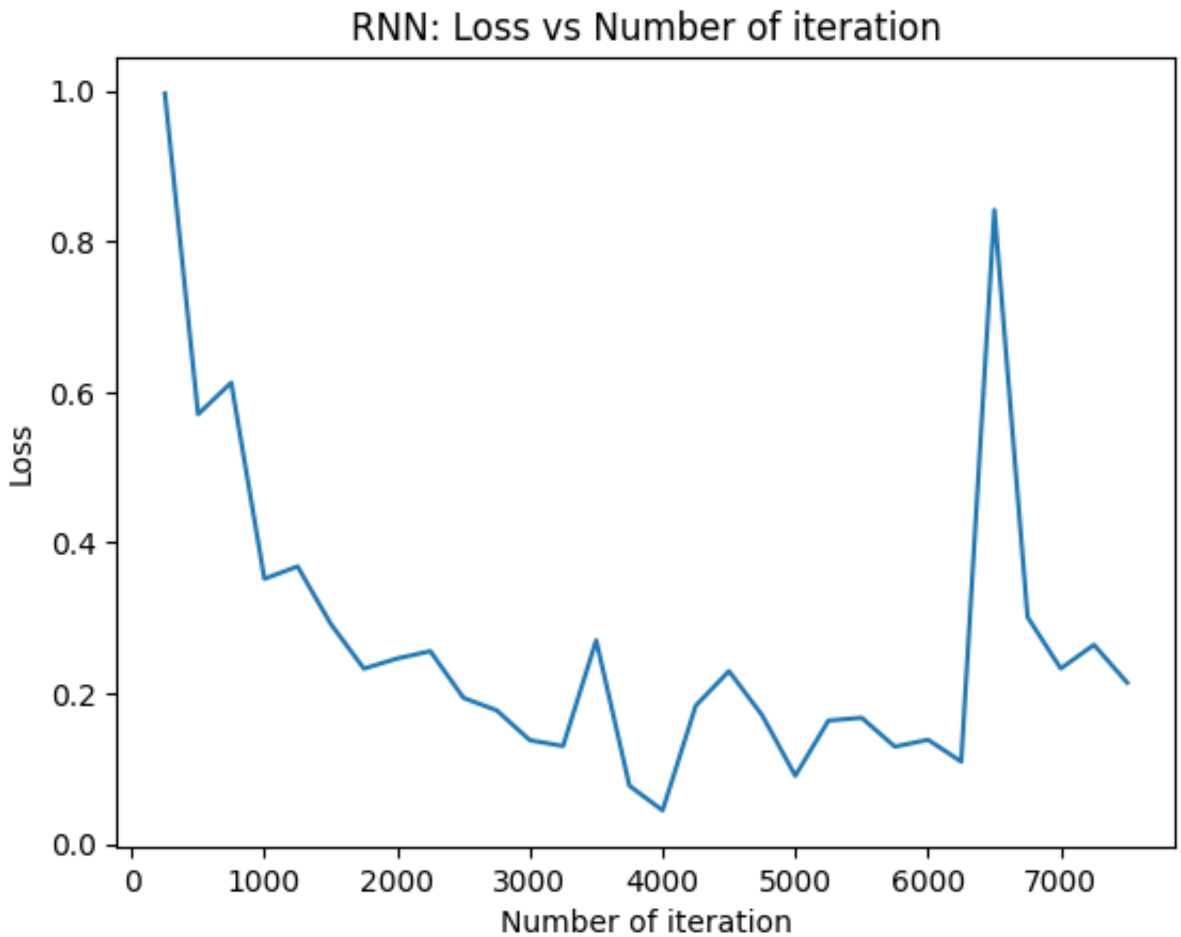
In [11]:

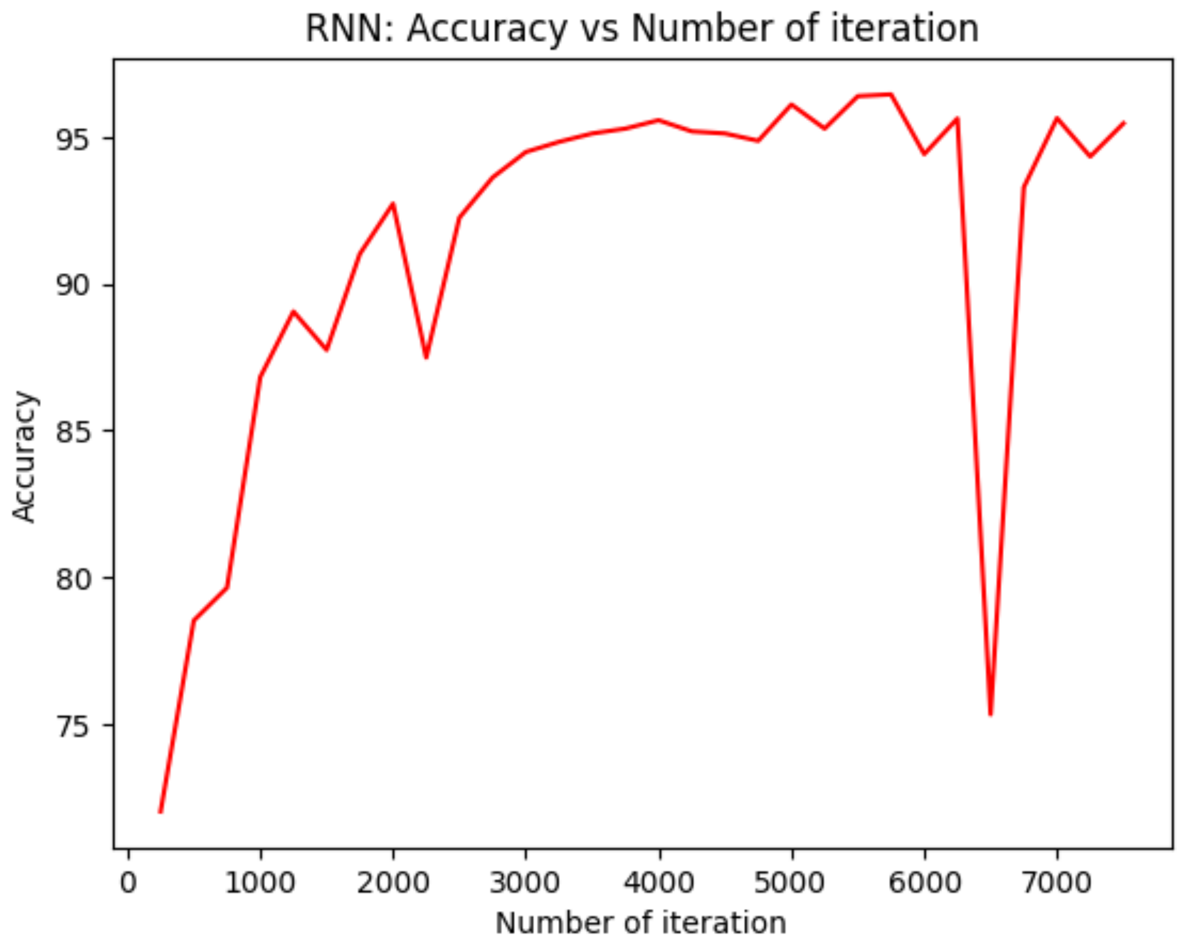
```
seq_dim = 28
loss_list = []
iteration_list = []
accuracy_list = []
count = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        train  = Variable(images.view(-1, seq_dim, input_dim))
        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 % 250 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                images = Variable(images.view(-1, seq_dim, input_dim))

                # Forward propagation
                outputs = model(images)

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

                # Total number of labels
                total += labels.size(0)

                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
```

```
Iteration: 500  Loss: 0.5704835653305054  Accuracy: 78.52381134033203 %
Iteration: 1000  Loss: 0.3518897294998169  Accuracy: 86.82142639160156 %
Iteration: 1500  Loss: 0.29200878739356995  Accuracy: 87.75 %
Iteration: 2000  Loss: 0.24608106911182404  Accuracy: 92.73809814453125 %
Iteration: 2500  Loss: 0.1938478648662567  Accuracy: 92.26190185546875 %
Iteration: 3000  Loss: 0.1376369446516037  Accuracy: 94.5 %
Iteration: 3500  Loss: 0.27033406496047974  Accuracy: 95.13095092773438 %
Iteration: 4000  Loss: 0.04438907653093338  Accuracy: 95.58333587646484 %
Iteration: 4500  Loss: 0.22941091656684875  Accuracy: 95.13095092773438 %
Iteration: 5000  Loss: 0.09047890454530716  Accuracy: 96.11904907226562 %
Iteration: 5500  Loss: 0.16744785010814667  Accuracy: 96.4047622680664 %
Iteration: 6000  Loss: 0.13822081685066223  Accuracy: 94.42857360839844 %
Iteration: 6500  Loss: 0.8418068885803223  Accuracy: 75.32142639160156 %
Iteration: 7000  Loss: 0.2327769249677658  Accuracy: 95.6547622680664 %
Iteration: 7500  Loss: 0.21402066946029663  Accuracy: 95.47618865966797 %
```

In [12]:
```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("RNN: 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("RNN: Accuracy vs Number of iteration")
plt.savefig('graph.png')
plt.show()
```





In conclusion

the model's performance during training has been somewhat mixed. There is a steady decrease in loss from 0.57 to 0.21, indicating that the model is improving its predictions.

The accuracy of the model has improved from 78.5% to 95.5%, which is a significant improvement.

However, the accuracy dips to 75.3% at iteration 6500, which could be due to overfitting or a noisy dataset.

Overall, the model seems to have learned the patterns and features of the dataset well.