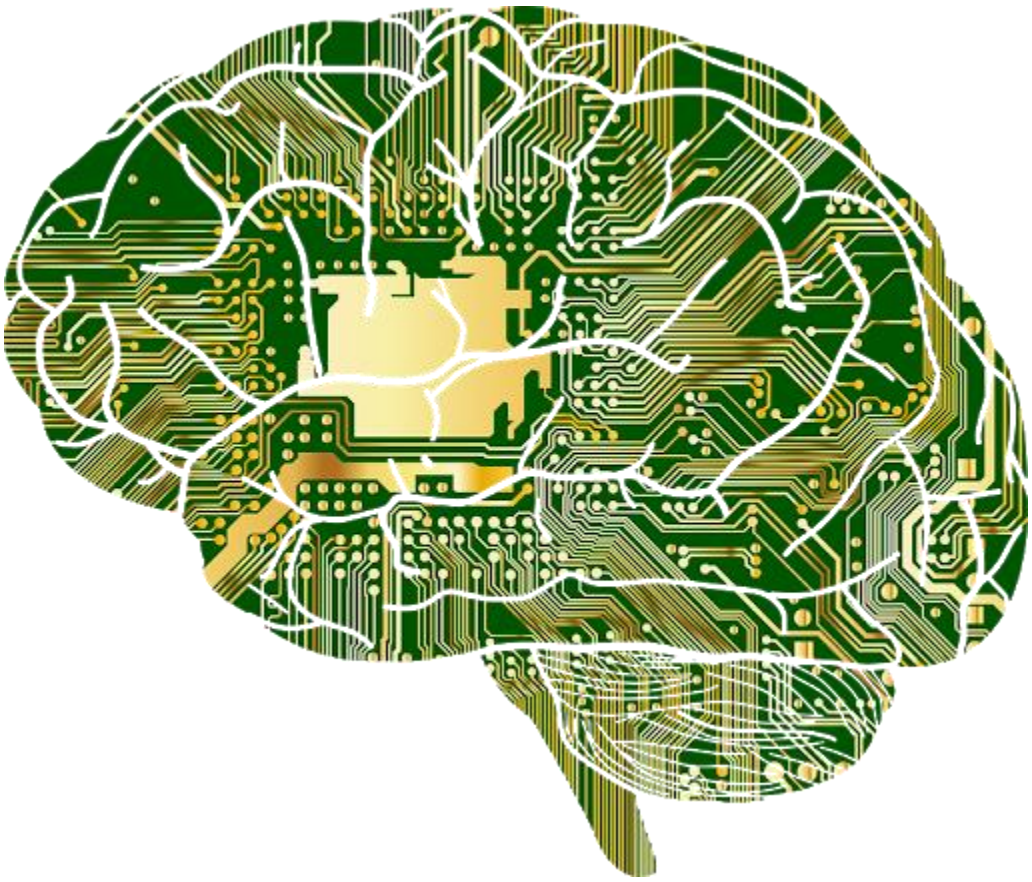


Using Recurrent Neural Networks for recognizing digits

پروژه درس هوش محاسباتی

هستی کوچکی – ۹۶۰۱۲۲۶۸۰۰۰۷

فردوس حاجی زاده- ۹۶۰۱۲۲۶۸۱۰۰۵

دانشگاه گیلان - دی ماه ۹۹

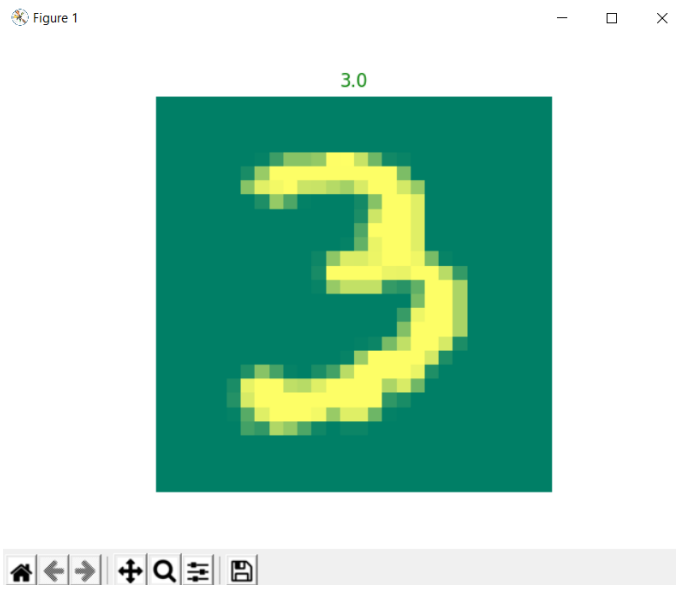
هدف از پیاده سازی این پروژه تشخیص صحیح اعداد از هزاران تصویر است. این تصاویر اعدادی هستند که به صورت دست نویس نوشته شده اند و ما به وسیله پیاده سازی شبکه عصبی بازگشتی (RNN) قصد داریم این اعداد را به درستی تشخیص دهیم.

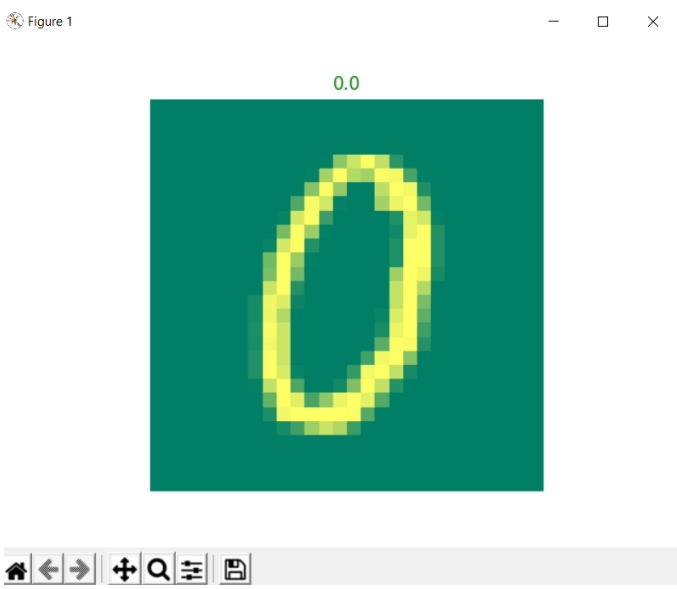
کتابخانه استفاده شده در این پروژه Pytorch است و دیتاست استفاده شده در آن دیتاست معروف MNIST است که در یک فایل به نام train.csv در فولدر input همراه با پروژه قرار گرفته است.

```
# Input data files are available in the "/input/" directory.

import os
print(os.listdir("input"))
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
# Prepare Dataset
# load data
train = pd.read_csv(r"input/train.csv",dtype = np.float32)
```

فایل train تصاویر gray-scale ای را توصیف می کند که ارقام بین 0 تا 9 دست نویس هستند. این فایل train شامل 758 ستون است که ستون اول (lable) عددی است که توسط کاربر ترسیم شده و بقیه ستون ها حاوی مقادیر پیکسل تصاویر هستند.

دو نمونه از تصاویر و lable آن ها که هنگام اجرای کد هم نمایش داده می شوند:





این فایل شامل حدودا 42000 عدد است که 80 درصد آن برای train کردن شبکه و از 20 درصد بقیه برای test کردن استفاده می شود.

برای شروع ستون lable به عنوان target و ما بقیه ستون ها به عنوان features (ورودی شبکه) هم برای test هم برای train استفاده می شود:

```
# 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.
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
```

سپس تصویر دو نمونه از اعداد موجود در دیتاست به عنوان نمونه همراه با lable آن ها به کاربر نشان داده می شود و این تصاویر در directory پروژه با نام photo1 و photo2 ذخیره می شوند:

```
# visualize two of the images in data set
plt.imshow(features_numpy[5].reshape(28,28),cmap='summer')
plt.axis("off")
plt.title(str(targets_numpy[5]),color="green")
plt.savefig('photo1.png')
plt.show()
plt.imshow(features_numpy[9].reshape(28,28),cmap='summer')
plt.axis("off")
plt.title(str(targets_numpy[9]),color="green")
plt.savefig('photo2.png')
plt.show()
```

سپس مدل ابتدایی RNN تعریف می شود که بعد بتوان توسط آن شبکه عصبی مورد نظر با تعداد لایه های دلخواه را ایجاد کنیم :

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

        #  hidden layer 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, nonlinearity='relu')

        # 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
```

بعد از تعریف RNN ، سایزbatch برابر با 100 ، و تعداد iteration برابر با 5000 در نظر گرفته می شود و با توجه به تعداد دیتاهای موجود برای آموزش شبکه تعداد کل epoch ها محاسبه خواهد شد (12 epochs) . سپس با توجه به سایز batch ، دیتاهایی که باید برای ورود به شبکه جهت train و test مورد استفاده قرار بگیرند، لود می شوند و در انتها یک شبکه RNN شامل یک لایه hidden به سایز 100 ایجاد شده و کار آموزش شبکه آغاز می شود.

```
# batch_size, epoch and iteration
batch_size = 100
n_iters = 5000
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)
```

برای آموزش شبکه در هر epoch داده های موجود در دیتاست train به همراه lable های آن ها به شبکه داده می شود (forward propagation) سپس مقدار loss به ازای داده های وارد شده محاسبه می شود و با استفاده از الگوریتم گرادیان کاهشی (Gradient Descent) سعی می کنیم مقدار loss را در هر iteration minimize کنیم .(Backward propagation)

```
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
```

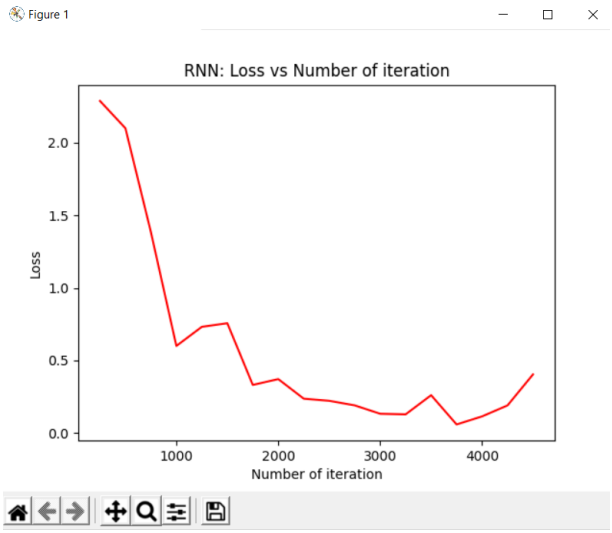
پس از هر 250 iteration دیتاست مربوط به test را به شبکه وارد می کنیم این باربدون وارد کردن lable ها. سپس مقادیر تخمینی شبکه را با lable آن مقایسه کرده و درصد Accuracy شبکه محاسبه می شود. پس از هر 500 iteration نیز مقادیر loss و accuracy در console چاپ می شوند.

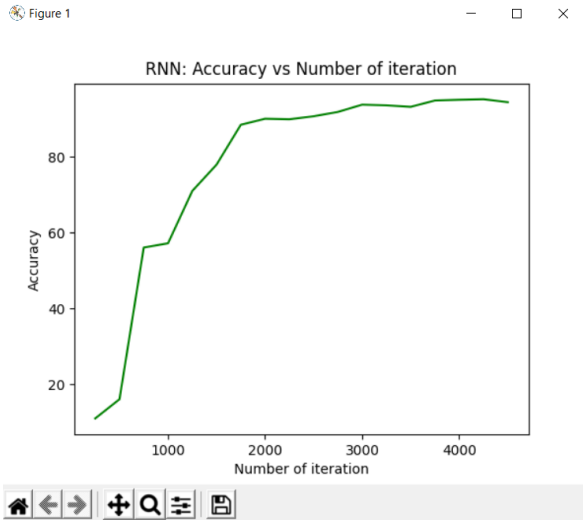
```
if count % 250 == 0:
    # Calculate Accuracy
    correct = 0
    total = 0
    # Iterate through test dataset
    for images, labels in test_loader:
        images = Variable(images.view(-1, seq_dim, input_dim))

        # 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, loss.data.item(), accuracy))
```

در نهایت پس از اتمام iteration 5000 دو نمودار به عنوان خروجی نمایش داده می شود که نمودار اول نشان دهنده تعداد loss نسبت به تعداد iteration و نمودار دوم نشان دهنده accuracy نسبت به تعداد iteration است. این نمودار ها با نام های graph1 و graph2 در directory پروژه ذخیره می شوند.

```
plt.plot(iteration_list,loss_list,color="red")
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("RNN: Loss vs Number of iteration")
plt.savefig('graph1.png')
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "green")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("RNN: Accuracy vs Number of iteration")
plt.savefig('graph2.png')
plt.show()
```

تصاویر نمودار ها و console پس از اتمام iteration 5000:






```
Iteration: 500  Loss: 2.100726366043091  Accuracy: 16.10714340209961 %
Iteration: 1000  Loss: 0.5998964309692383  Accuracy: 57.21428680419922 %
Iteration: 1500  Loss: 0.7565293312072754  Accuracy: 77.95237731933594 %
Iteration: 2000  Loss: 0.37141913175582886  Accuracy: 90.04762268066406 %
Iteration: 2500  Loss: 0.2216823697090149  Accuracy: 90.70237731933594 %
Iteration: 3000  Loss: 0.1324249655008316  Accuracy: 93.76190185546875 %
Iteration: 3500  Loss: 0.26045382022857666  Accuracy: 93.19047546386719 %
Iteration: 4000  Loss: 0.11421789228916168  Accuracy: 95.04762268066406 %
Iteration: 4500  Loss: 0.40396127104759216  Accuracy: 94.41666412353516 %
```