In [1]:

```
import torch
import torch.nn as nn
from torch.optim import SGD
import numpy as np
```

Упражнение, для реализации "Ванильной" RNN

- Попробуем обучить сеть восстанавливать слово hello по первой букве. т.е. построим charecter-level модель

In [2]:

```
a = torch.ones((3,3))*3
b = torch.ones((3,3))*5
```

In [3]:

```
a
```

Out[3]:

```
tensor([[3., 3., 3.],
        [3., 3., 3.],
        [3., 3., 3.]])
```

In [4]:

```
b
```

Out[4]:

```
tensor([[5., 5., 5.],
        [5., 5., 5.],
        [5., 5., 5.]])
```

In [5]:

```
a @ b
```

Out[5]:

```
tensor([[45., 45., 45.],
        [45., 45., 45.],
        [45., 45., 45.]])
```

In [6]:

```
a * b
```

Out[6]:

```
tensor([[15., 15., 15.],
        [15., 15., 15.],
        [15., 15., 15.]])
```

In [7]:

```
word = 'ololoasdasdddqweqw123456789'
# word = 'hello'
```

Датасет.

Позволяет:

- Закодировать символ при помощи one-hot
- Делать итератор по слову, которыей возвращает текущий символ и следующий как таргет

In [8]:

```
class WordDataSet:

    def __init__(self, word):
        self.chars2idx = {}
        self.indexs  = []
        for c in word:
            if c not in self.chars2idx:
                self.chars2idx[c] = len(self.chars2idx)

            self.indexs.append(self.chars2idx[c])

        self.vec_size = len(self.chars2idx)
        self.seq_len  = len(word)

    def get_one_hot(self, idx):
        x = torch.zeros(self.vec_size)
        x[idx] = 1
        return x

    def __iter__(self):
        return zip(self.indexs[:-1], self.indexs[1:])

    def __len__(self):
        return self.seq_len

    def get_char_by_id(self, id):
        for c, i in self.chars2idx.items():
            if id == i: return c
        return None
```

Реализация базовой RNN

Скрытый элемент

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Выход сети

$$y_t = W_{hy}h_t$$

In [9]:

```
class VanillaRNN(nn.Module):

    def __init__(self, in_size=5, hidden_size=3, out_size=5):
        super(VanillaRNN, self).__init__()
        self.x2hidden    = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.hidden      = nn.Linear(in_features=hidden_size, out_features=hidden_size)
        self.activation  = nn.Tanh()
        self.outweight   = nn.Linear(in_features=hidden_size, out_features=out_size)

    def forward(self, x, prev_hidden):
        hidden = self.activation(self.x2hidden(x) + self.hidden(prev_hidden))
#         Версия без активации - может происходить gradient exploding
#         hidden = self.x2hidden(x) + self.hidden(prev_hidden)
        output = self.outweight(hidden)
        return output, hidden
```

Инициализация переменных

In [10]:

```
ds = WordDataSet(word=word)
rnn = VanillaRNN(in_size=ds.vec_size, hidden_size=8, out_size=ds.vec_size)
criterion = nn.CrossEntropyLoss()
e_cnt     = 200
optim     = SGD(rnn.parameters(), lr = 0.1, momentum=0.9)
```

Обучение

In [11]:

```
CLIP_GRAD = True

for epoch in range(e_cnt):
    hh = torch.zeros(rnn.hidden.in_features)
    loss = 0
    optim.zero_grad()
    for sample, next_sample in ds:
        x = ds.get_one_hot(sample).unsqueeze(0)
        target =  torch.LongTensor([next_sample])

        y, hh = rnn(x, hh)

        loss += criterion(y, target)


    loss.backward()

    if epoch % 10 == 0:
        print (loss.data.item())
        if CLIP_GRAD: print("Clip gradient : ", torch.nn.utils.clip_grad_norm_(rnn.para
meters(), max_norm=5))
    else:
        if CLIP_GRAD: torch.nn.utils.clip_grad_norm_(rnn.parameters(), max_norm=1)

#     print("Params : ")
#     num_params = 0
#     for item in rnn.parameters():
#         num_params += 1
#         print(item.grad)
#     print("NumParams :", num_params)
#     print("Optimize")

    optim.step()
```

```
71.36385345458984
Clip gradient :  6.86864607027851
52.808048248291016
Clip gradient :  18.76307334055694
35.8555793762207
Clip gradient :  10.531217057028805
18.26323699951172
Clip gradient :  6.579962303244384
5.721493244171143
Clip gradient :  7.209886694543383
3.5520081520080566
Clip gradient :  4.494360920223783
2.1401476860046387
Clip gradient :  3.101429260895991
4.411831855773926
Clip gradient :  10.557477043117581
6.175124645233154
Clip gradient :  22.936552644124166
2.303819179534912
Clip gradient :  1.3037154058705873
1.2081799507141113
Clip gradient :  7.820986150586707
1.9899497032165527
Clip gradient :  1.0529065038256773
1.7811946868896484
Clip gradient :  1.485835834794035
1.6265811920166016
Clip gradient :  1.1076671744673605
1.5290412902832031
Clip gradient :  0.7525557152875515
1.4064817428588867
Clip gradient :  0.6585222868922677
0.26937103271484375
Clip gradient :  0.33546065151328947
0.13422203063964844
Clip gradient :  0.1849632183589549
0.09629058837890625
Clip gradient :  0.07863002441880117
0.07278156280517578
Clip gradient :  0.042655780828987494
```

Тестирование

In [12]:

```
rnn.eval()
hh = torch.zeros(rnn.hidden.in_features)
id = 0
softmax  = nn.Softmax(dim=1)
predword = ds.get_char_by_id(id)
for c in enumerate(word[:-1]):
    x = ds.get_one_hot(id).unsqueeze(0)
    y, hh = rnn(x, hh)
    y = softmax(y)
    m, id = torch.max(y, 1)
    id = id.data[0]
    predword += ds.get_char_by_id(id)
print ('Prediction:\t' , predword)
print("Original:\t", word)
assert(predword == word)
```

```
Prediction:     ololoasdasddqweqw123456789
Original:       ololoasdasddqweqw123456789
```

ДЗ

Реализовать LSTM и GRU модули, обучить их предсказывать тестовое слово Сохранить ноутбук с предсказанием и пройденным assert и прислать на почту a.murashev@corp.mail.ru с темой:

[МФТИ_2019_1] ДЗ №8 ФИО

In [13]:

```
#тестовое слово
word = 'ololoasdasddqweqw123456789'
```

Реализовать LSTM

In [14]:

```
#Написать реализацию LSTM и обучить предсказывать слово

class LSTM(nn.Module):

    def __init__(self, in_size=5, hidden_size=3, out_size=5):
        super(LSTM, self).__init__()

        self.first_x       = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.first_hidden  = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.first_sigm    = nn.Sigmoid()

        self.second_x      = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.second_hidden = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.second_sigm   = nn.Sigmoid()

        self.third_x       = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.third_hidden  = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.third_tanh    = nn.Tanh()

        self.fourth_x      = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.fourth_hidden = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.fourth_sigm   = nn.Sigmoid()

        self.tanh          = nn.Tanh()
        self.outweight     = nn.Linear(in_features=hidden_size, out_features=out_size)

    def forward(self, x, prev_hidden, prev_C):
        first              = self.first_sigm(self.first_x(x) + self.first_hidden(prev_h
idden))
        second             = self.second_sigm(self.second_x(x) + self.second_hidden(pre
v_hidden))
        third              = self.third_tanh(self.third_x(x) + self.third_hidden(prev_h
idden))
        fourth             = self.fourth_sigm(self.fourth_x(x) + self.fourth_hidden(pre
v_hidden))

        C                  = first * prev_C + second * third
        hh                 = fourth * self.tanh(C)
        output             = self.outweight(hh)

        return output, hh, C
```

In [15]:

```
def train_lstm(net, is_CLIP_GRAD=True, e_cnt=100, hid_s=3):
    criterion = nn.CrossEntropyLoss()
    optim     = SGD(net.parameters(), lr = 0.1, momentum=0.9)

    CLIP_GRAD = is_CLIP_GRAD

    for epoch in range(e_cnt):
        hh = torch.zeros(hid_s)
        C = torch.zeros(hid_s)
        loss = 0
        optim.zero_grad()
        for sample, next_sample in ds:
            x = ds.get_one_hot(sample).unsqueeze(0)
            target =  torch.LongTensor([next_sample])

            y, hh, C = net(x, hh, C)

            loss += criterion(y, target)

        loss.backward()

        if epoch % 50 == 0:
            print (loss.data.item())
            if CLIP_GRAD: print("Clip gradient : ", torch.nn.utils.clip_grad_norm_(net.
parameters(), max_norm=5))
        else:
            if CLIP_GRAD: torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1)

        optim.step()
```

In [16]:

```
def test_lstm(net, hid_s=3):
    net.eval()
    hh = torch.zeros(hid_s)
    C = torch.zeros(hid_s)
    id = 0
    softmax  = nn.Softmax(dim=1)
    predword = ds.get_char_by_id(id)
    for c in enumerate(word[:-1]):
        x = ds.get_one_hot(id).unsqueeze(0)
        y, hh, C = net(x, hh, C)
        y = softmax(y)
        m, id = torch.max(y, 1)
        id = id.data[0]
        predword += ds.get_char_by_id(id)
    print ('Prediction:\t' , predword)
    print("Original:\t", word)
    assert(predword == word)
```

In [17]:

```
ds = WordDataSet(word=word)
hid_s = 5
lstm = LSTM(in_size=ds.vec_size, hidden_size=hid_s, out_size=ds.vec_size)
```

In [18]:

```
train_lstm(net=lstm, is_CLIP_GRAD=True, e_cnt=1000, hid_s=hid_s)
```

```
72.03343963623047
Clip gradient :  4.185040428649836
26.815519332885742
Clip gradient :  12.007424742641563
13.877161026000977
Clip gradient :  2.108402869076971
3.4952526092529297
Clip gradient :  4.598913211104315
1.6169724464416504
Clip gradient :  1.7350150708633005
0.5218076705932617
Clip gradient :  0.3565007929432089
0.25933170318603516
Clip gradient :  0.05049733697658246
0.16981220245361328
Clip gradient :  0.03405291813887915
0.12395763397216797
Clip gradient :  0.025201981814414767
0.0975809097290039
Clip gradient :  0.019990398241705398
0.08038997650146484
Clip gradient :  0.01653678048972525
0.06836795806884766
Clip gradient :  0.014091730677699846
0.059477806091308594
Clip gradient :  0.012279191169092148
0.05263805389404297
Clip gradient :  0.010879666357407626
0.047209739685058594
Clip gradient :  0.009766171411975829
0.042799949645996094
Clip gradient :  0.008860466554641292
0.03914451599121094
Clip gradient :  0.008108552160995104
0.036060333251953125
Clip gradient :  0.00747308009018271
0.033432960510253906
Clip gradient :  0.006931976957205789
0.031165122985839844
Clip gradient :  0.006465429413582028
```

In [19]:

```
test_lstm(net=lstm, hid_s=hid_s)
```

```
Prediction:      ololoasdasddqweqw123456789
Original:        ololoasdasddqweqw123456789
```

Реализовать GRU

In [20]:

```
#Написать реализацию GRU и обучить предсказывать слово

class GRU(nn.Module):

    def __init__(self, in_size=5, hidden_size=3, out_size=5):
        super(GRU, self).__init__()

        self.first_x       = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.first_hidden  = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.first_sigm    = nn.Sigmoid()

        self.second_x      = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.second_hidden = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.second_sigm   = nn.Sigmoid()

        self.third_x       = nn.Linear(in_features=in_size, out_features=hidden_size)
        self.third_hidden  = nn.Linear(in_features=hidden_size, out_features=hidden_siz
e)
        self.third_tanh    = nn.Tanh()

        self.outweight     = nn.Linear(in_features=hidden_size, out_features=out_size)

    def forward(self, x, prev_hidden):
        first = self.first_sigm(self.first_x(x) + self.first_hidden(prev_hidden))
        second = self.second_sigm(self.second_x(x) + self.second_hidden(prev_hidden))

        hh = prev_hidden - first * prev_hidden + first *self.third_tanh(self.third_x(x)
+
                                                                        self.third_hidd
en(second * prev_hidden))
        output = self.outweight(hh)

        return output, hh
```

In [21]:

```
ds = WordDataSet(word=word)
hid_s = 5
gru = GRU(in_size=ds.vec_size, hidden_size=hid_s, out_size=ds.vec_size)
```

In [22]:

```
def train_gru(net, is_CLIP_GRAD=True, e_cnt=100, hid_s=3):
    CLIP_GRAD = is_CLIP_GRAD

    criterion = nn.CrossEntropyLoss()
    optim     = SGD(net.parameters(), lr = 0.1, momentum=0.9)

    for epoch in range(e_cnt):
        hh = torch.zeros(hid_s)
        loss = 0
        optim.zero_grad()
        for sample, next_sample in ds:
            x = ds.get_one_hot(sample).unsqueeze(0)
            target =  torch.LongTensor([next_sample])

            y, hh = net(x, hh)

            loss += criterion(y, target)


        loss.backward()

        if epoch % 50 == 0:
            print (loss.data.item())
            if CLIP_GRAD: print("Clip gradient : ", torch.nn.utils.clip_grad_norm_(net.
parameters(), max_norm=5))
        else:
            if CLIP_GRAD: torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1)

        optim.step()
```

In [23]:

```
def test_gru(net, hid_s=3):
    net.eval()
    hh = torch.zeros(hid_s)
    id = 0
    softmax  = nn.Softmax(dim=1)
    predword = ds.get_char_by_id(id)
    for c in enumerate(word[:-1]):
        x = ds.get_one_hot(id).unsqueeze(0)
        y, hh = net(x, hh)
        y = softmax(y)
        m, id = torch.max(y, 1)
        id = id.data[0]
        predword += ds.get_char_by_id(id)
    print ('Prediction:\t' , predword)
    print("Original:\t", word)
    assert(predword == word)
```

In [24]:

```
train_gru(net=gru, is_CLIP_GRAD=True, e_cnt=500, hid_s=hid_s)
```

```
71.65504455566406
Clip gradient :  4.756080706807321
5.6750030517578125
Clip gradient :  1.1757942029989996
1.221165657043457
Clip gradient :  8.48200200945489
1.4299821853637695
Clip gradient :  12.019203308928248
0.2422170639038086
Clip gradient :  0.2695188530743263
0.09823322296142578
Clip gradient :  0.0316749506522398
0.06674957275390625
Clip gradient :  0.019709762150494043
0.05107879638671875
Clip gradient :  0.015059797172100507
0.04145336151123047
Clip gradient :  0.012270645496640354
0.03486061096191406
Clip gradient :  0.010402715270363683
```

In [25]:

```
test_gru(net=gru, hid_s=hid_s)
```

```
Prediction:      ololoasdasddqweqw123456789
Original:        ololoasdasddqweqw123456789
```