بسم الله الرحمن الرحیم

گزارش تمرین سری 7 پایتورچ نوروساینس

امیرحسین رستمی

پوریا ملاحسینی

دکتر کربلایی آقاجان

دانشگاه صنعتی شریف

تابستان 98

پاسخ خواسته ی شماره اول :

نکته ی مهم: در تمارین قبلی جهت تمیز کردن loss function داشتیم که از ابزار قوی optimizer ـ که به کمک بهره گیری از هاب های زیر پیاده سازی شده بودـ استفاده کردیم اما در این تمرین به دلیل پرهیز از پیچیدگی ماژول کلاس های شبکه ی عصبی از استفاده کردن از این ابزار خود داری کرده ام.

- پیش بخش :آنچه گذشت

مروری بر کد شبکه ی عصبی ساده ی تمرین قبل هم چنین منابع زیر جهت پیاده سازی ابزار های از قبیل adamOptimizer و ... استفاده شده است.

- https://github.com/jcjohnson/pytorch-examples
- https://github.com/synxlin/nn-compression

در ادامه ابتدا کد شبکه ی تک لایه ی تمرین شماره 5 را می آوریم و سپس جهت ساده سازی شبکه و پیاده سازی ساختار فیدبک دار(بازگشتی و زیست محور) یک نمونه کد دیگر از شبکه ی عصبی پیاده سازی می کنیم که ضریب پیچیدگی به نسبت کمتری در بردارد.

کد شبکه ی تمرین سری قبل :

```
1
2   # one hidden layer neural networks...
3
4   import torch
5   import numpy as np
6   import torch.nn as nn
7   import torch.nn.functional as F
8
9   class Net(nn.Module):
10
11      def __init__(self, input_size, output_size, hidden_size, learning_rate):
12
13          super(Net, self).__init__()  # one hidden layer
14          self.inputSize = input_size
15          self.outputSize = output_size
16          self.hiddenSize = hidden_size
17
18          self.learning_rate = learning_rate
19          self.current_error = 1
20          self.previous_error = 0
21          # saving log for drawing the plots...
22          self.netLog = []
23
24          self.last_step = 1
25          self.dynamic_learning_rate = 1
26          # using previous datas... for backward & error propagation ...
27          self.learning_gradient = 1
28          self.previous_learning_gradient = 1
29
30          # weights initialize
31          self.neuronLog = []
32          self.W1 = torch.randn(self.inputSize, self.hiddenSize)  # first Weights ...
33          self.W2 = torch.randn(self.hiddenSize, self.outputSize)  # second Weights ...
34          return
35
36
```

```
    def forward(self, X):

        # input * weights == firstLayer ...
        # firtsLayer * Function(sigmond funtion instead u(t)) --> input for hiddenLayer
        # HiddenLayer * secondWeights
        # afterHiddenLayer * Function(Sigmond function instead u(t)) --> final output

        self.firstL = torch.matmul(X, self.W1)
        self.afterFirstL = self.sigmoid(self.firstL)  # activation function
        self.updateNeuronLog(self.afterFirstL)
        self.HiddenL = torch.matmul(self.afterFirstL, self.W2)
        afterHiddenL = self.sigmoid(self.HiddenL)
        return afterHiddenL


    #Activation Function
    def sigmoid(self, s):
        # sigmoid function
        return (1 / (1 + torch.exp(-s)))

    # ramp function
    def relu(self, s):
        if(s<=0):
            return 0
        else:
            return s

    def reluPrime(self, s):
        # derivative of ramp function
        if (s<= 0 ):
            return 0
        else:
            return 1

    def sigmoidPrime(self, s):
        # derivative of sigmoid function
        # it converges and has a lot of flactuation ... return self.sigmoid(s) * (1 - self.sigmoid(s))
        return s*(1-s)

    def backward(self,inp,realOutput,gottenOutput):

        # for firstLayer
        self.outEr = realOutput - gottenOutput
        self.outDelta = self.outEr * self.sigmoidPrime(gottenOutput)

        # for secondLayer
        self.afterFirstL_Error = torch.matmul(self.outDelta,torch.t(self.W2))
        self.afterFirstL_Delta = self.afterFirstL_Error * self.sigmoidPrime(self.afterFirstL)

        # modify Weights
        # firstLayer Ws
        self.W1 = self.W1 + torch.matmul(torch.t(inp),self.afterFirstL_Delta) * self.learning_rate
        self.W2 = self.W2 + torch.matmul(torch.t(self.afterFirstL_Delta),self.outDelta) * self.learning_rate
        self.current_error = torch.mean(torch.abs(self.outEr))
        # add to log ...
        self.updateLog()

        self.learning_gradient = (self.previous_error - self.current_error) / self.previous_error
        if self.dynamic_learning_rate == 1 and self.previous_error != 0:

            if self.learning_gradient > self.previous_learning_gradient:
                self.learning_rate = self.learning_rate / 1.1
                # self.last_step = self.last_step / 2
            else:
                self.learning_rate = self.learning_rate * 1.1
                # self.last_step = 1

        self.previous_learning_gradient = self.learning_gradient
        self.previous_error = self.current_error
        return
```

```
    def train(self, inp, realOut):
        # forward + backward pass for training
        calOut = self.forward(inp)
        self.backward(inp, realOut, calOut)
        # print(self.W1, '\n', self.W2)
        return

    def saveWeights(self, repository, address):
        # we will use the PyTorch internal storage functions
        torch.save(repository, address)
        # torch.load(address)
        return

    def updateLog(self):
        self.netLog.append([self.current_error * 10, self.learning_rate, self.learning_gradient])
        return

    def getLog(self):
        outlog = np.array(self.netLog)
        return outlog

    def updateNeuronLog(self, hidden_neurons_log):
        self.neuronLog.append([np.array(hidden_neurons_log)])
        return

    def getNeuronLog(self):
        return self.neuronLog

    def setLearningRate(self, learning_rate):
        self.learning_rate = learning_rate
        return

    def setDynamicLearningRate(self, dynamic_learning_rate):
        self.dynamic_learning_rate = dynamic_learning_rate
        return

    def setTrain(self, training_sample_input, training_sample_output, batch_size, epoches_number, learning_rate,
                 dynamic_learning_rate):

        self.setLearningRate(learning_rate)
        self.setDynamicLearningRate(dynamic_learning_rate)
        X1 = []
        Y1 = []

        # saving data in X1,Y1 --> need for plot ...
        for i in range(0, batch_size):
            X1.append(training_sample_input)
            Y1.append(training_sample_output)

        X1 = torch.cat(X1, 0)
        Y1 = torch.cat(Y1, 0)
        for i in range(0, epoches_number):
            self.train(X1, Y1)
        return
```

- بخش اول : تکمیل و توانمند سازی مدل به کمک پیاده سازی ساختار فیدبک دار(بازگشتی) جهت هر چه نزدیک تر شدن به حالت زیستی موجود به کمک مقاله ی ارایه شده

ابتدا توجه کنید که رابطه ی حیاتی ایجاد گر فیدبک رابطه ی کلیدی زیر می باشد:

$$\mathbf{x}_t = (1-\alpha)\mathbf{x}_{t-1} + \alpha(W^{\text{rec}}\mathbf{r}_{t-1} + W^{\text{in}}\mathbf{u}_t) + \sqrt{2\alpha\sigma^2_{\text{rec}}}\ \mathbf{N}(0,1),$$

حال به پیاده سازی این رابطه در شبکه ی عصبی می پردازیم : اما پیش از پیاده سازی طبق گفته ی صورت تمرین داریم که از ترم آخر که به نوعی نویز گاوسی سفیدی! اعمالی در خروجی است صرف نظر می کنیم.

حال برویم سراغ پیاده سازی شبکه... پیش از بررسی ذکر این نکته ضروری است که در کد پیاده شده ما یک لایه از ورودی به پنهان اولیه داریم،یک لایه ی دیگر از پنهان اول به پنهان دوم و لایه ی دیگر از پنهان دوم به خروجی وجود دارد و مانور رابطه ی فوق طبق متن مقاله در لایه های پنهان به پنهان و ورودی به پنهان می باشد.

شبکه ی پیاده سازی شده :

```python
import torch
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # weights initialize
        self.inpHid1 = nn.Linear(input_size, hidden_size)
        self.hid1Hid2 = nn.Linear(hidden_size, hidden_size)
        self.hid2Out = nn.Linear(hidden_size, output_size)
        self.activationFunction = nn.Tanh()

    def forward(self, input, hidden):

        # someHow as
        a = 0.5

        # input * weights == firstLayer ...
        # firtsLayer * Function(sigmond funtion instead u(t)) --> input for hiddenLayer
        # HiddenLayer * secondWeights
        # afterHiddenLayer * Function(Sigmond function instead u(t)) --> final output

        # according to the formula.
        hidden = self.activationFunction((1-a)*hidden + a*self.inpHid1(input) + a*self.hid1Hid2(nn.functional.relu(hidden)))
        output = self.activationFunction(self.hid2Out(hidden))
        return output, hidden

    def backProp(self):
        # creating  backpropagation matrix
        return Variable(torch.zeros(1, self.hidden_size))
```

نکات پیش شبیه سازی و اجرای دیتاست ها

- می دانیم خروجی برخی دیتاست ها باینری هست و این ناپیوستگی برای شبکه ی ما ناخوشآیند است لذا در چنین مواقع یک تابع اکتیواسیون خوب قرار داده ایم از قبیل tanh و sigmoid و relu که در این مرحله ما tanh را درنظر گرفتیم.
- طبق گفته ی صورت پروژه از نویز داخلی صرف نظر کرده ایم و صرفا نویز ورودی را فرض بر وجود کرده ایم.

نکته ی چالشی : چگونه رفتار Excitatory-Inhibitory را پیاده سازی کنیم؟

پاسخ : به این شکل که می دانیم(طبق اسلاید ها و مقاله) اثر تحریکی همانند خانه های با علامت مثبت_صفر و اثر Inhibitory همانند خانه های با علامت منفی_صفر می باشد و داریم که برای پیاده سازی نورون های چند گانه باید این فیتلر علامت را روی ماتریس وزن خود اعمال کنیم.

حال سوال بعد،چگونه این پیاده سازی را انجام بدهیم؟

پاسخ : در هر مرحله به روز رسانی ماتریس اوزان این شرط را اعمال می کنیم به این طریق که :

- اوزان روی قطر اصلی صفر می شوند.
- به میزان درصد نورون های تحریکی،خانه های ماتریس وزن دارای مقدار مثبت_صفر و به میزان درصد نورون های مهارکننده نیز خانه های ماتریس وزن دارای مقدار منفی_صفر می گردد و با توجه به بحث اسلاید های درس و مقاله از آنجا که شدت میزان نورون های تحریکی خیلی بیشتر از نورون های مهاری است بنده در نظر گرفتم که مثلا 80 درصد نورون ها تحریک کننده و 20 درصد نورون های مهاری است.

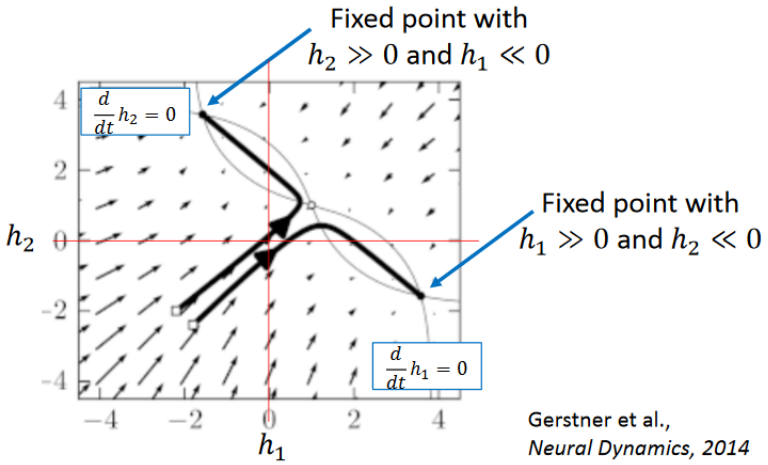
که موارد بالا به صورت اجرای جاروبی for در هر مرحله به روز رسانی شبکه انجام می گیرد.

- **پیاده سازی تسک اول: Perceptual Decision Making**

1- **توضیحات و معرفی** : این تسک طبق مقاله به دو روش پیاده سازی شده است که در اینجا به بررسی هر دو روش می پردازیم

می دانیم که یادگیری من جمله مسایل پر چالش و بررسی نوروساینس است.غالب اوقات نورون ها در اثر مجموعه ای از سیگنال های ورودی شروع به تصمیم گیری می کنند و این سیگنال های ورودی می توانند دارای نویز نسبی ای باشند.در فرآیند تصمیم گیری همانند مطالب ذکر شده در سری اسلاید های درس داریم که تصمیم گیری بدین طریق انجام می گیرد که در اثر ورود تحریک به شبکه ی عصبی برخی دسته نورون ها excite شده و برخی نورون های inhibit شده است و به یاد داریم که دست و پنجه نرم کردن این شدت تحریک/مهار نقش اصلی در تصمیم گیری شبکه ی عصبی دارد.

برای شفاف تر شدن این قضیه داریم که به اسلاید ها رجوع می کنیم:



Gerstner et al.,
Neural Dynamics, 2014

همانطورکه می دانید اگر شدت تحریک نورون های تحریکی به اندازه خوبی از شدت مهار نورون های مهار کننده بهتر باشد تصمیم گیری مطابق میل نورون های تحریکی خواهد بود اما اگر شدت تحریک نورون های مهاری بیشتر باشد در نهایت خواهیم داشت که تصمیم گیری به سمت میل مهارکنندگی خواهد بود.

2- **تولید دیتا ست**:

آزمایش مرجع مربوط به این بخش،آزمایش حرکت تصادفی است.برای مدل سازی این آزمایش،نیاز داریم درصد حرکت نقاط را در دو جهت مختلف به گونه ای در ورودی مدل کنیم.می دانیم که هرچقدر درصد نقاط که در یک جهت حرکت کنند بیشتر باشد،تعیین جهت حرکت عمومی راحت تر است.برای مدل کردن این رفتار،از دو مقدار ثابت استفاده می کنیم که نماد درصد حرکت نقاط هستند.

تسک اول : Perceptual Decision Making

- **تصمیم گیری پس از مشاهده ی کل ورودی**

در این پیاده سازی،شبکه پس از مشاهده ی تمام ورودی اعمال شده تصمیم خود را در یک جهت می گیرد.برای سادگی پیاده سازی بدین ترتیب عمل می کنیم:

پس از اتمام هر ورودی درطول زمان، تنها آخرین خروجی را مد نظر قرار می دهیم و با مقایسه ی آن با خروجی مطلوب وزن ها را آپدیت می کنیم توجه کنید که همانطورکه اشاره کردم خروجی ها دو بیتی هستند که یکی صفرو ودیگری یک می باشد(جهت یابی و طبیعتا یک جهت انتخاب می شود).

- **تصمیم گیری بلافاصله پس از رسیدن به تصمیم اولیه در اثر ورود داده ها**

نمونه ای از خروجی را در ادامه خواهیم دید و اثر تغییر مقدار نورون های تحریکی و مهاری را خواهیم دید.

شکل ضمیمه شده(دو شکل اول) درصد پاسخ گویی شبکه روی داده ی تست را به ازای تحریک هایی با طول مختلف و به ازای تفاوت در اندازه ی ورودی نشان می دهد.مشاهده می کنیم که با افزایش مقدار درصد پاسخ گویی زیاد می شود و افزایش طول تحریک نیز به طور تقریبی اثری در افزایش پاسخ گویی دارد.

هم چنین در ادامه نیز نمودار های مربوط به حالت Loss , convergance to truth را هم قرار داده ام.

#یادآوری :

به جدول زیر که از مقاله اقتباس شده است توجه کنید و با نتایج خروجی کد مقایسه کنید

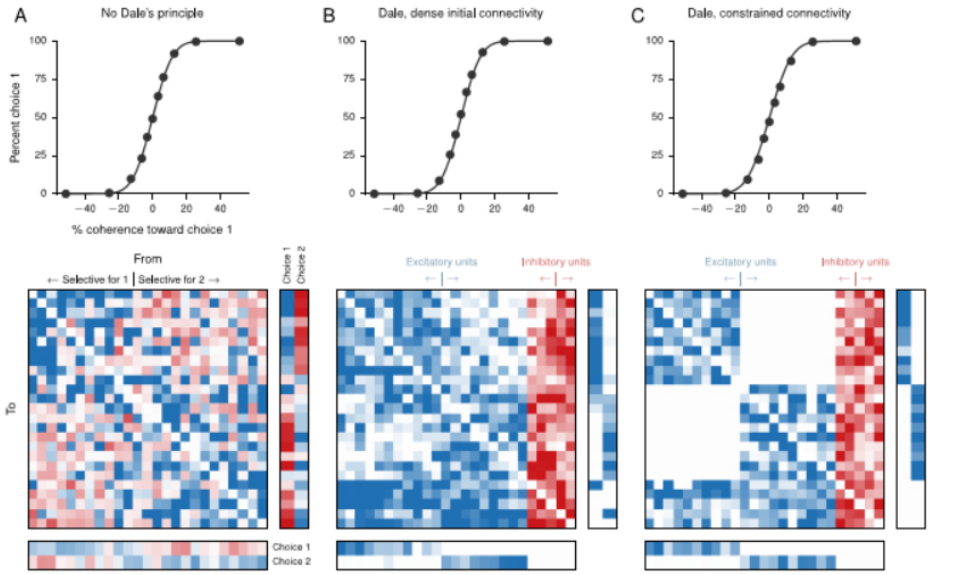
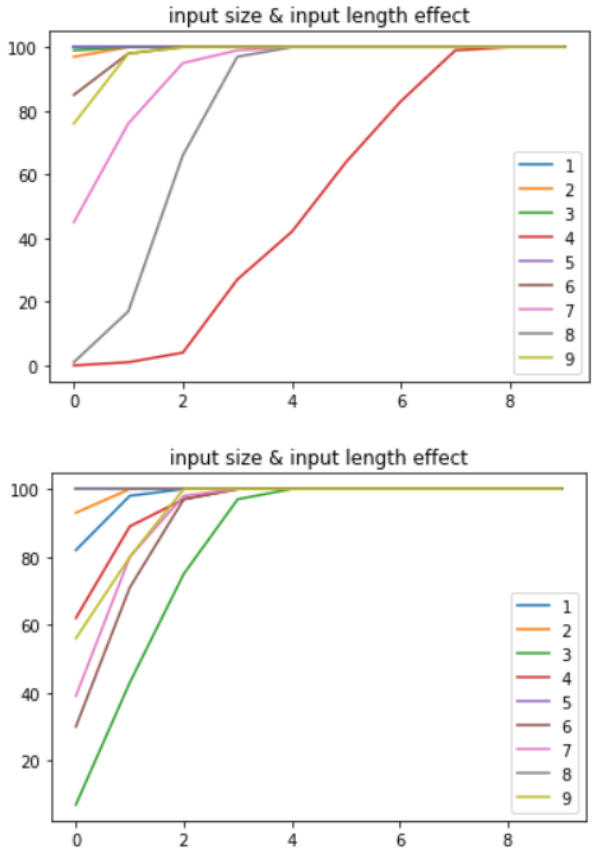
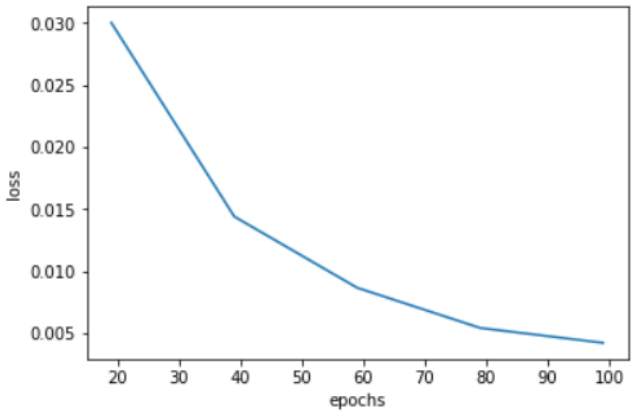


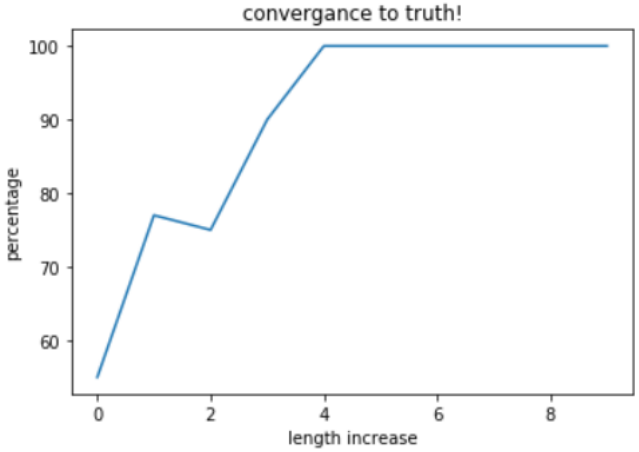
Fig 3. Perceptual decision-making networks with different constraints. (A) Psychometric function (percent choice 1 as a function of signed coherence) and connection weights (input, upper-right; recurrent, upper-left; and output, lower) for a network in which all weights may be positive or negative, trained for a perceptual decision-making task. Connections go from columns ("pre-synaptic") to rows ("post-synaptic"), with blue representing positive weights and red negative weights. Different color scales (arbitrary units) were used for the input, recurrent, and output matrices but are consistent across the three networks shown. In the psychometric function, solid lines are fits to a cumulative Gaussian distribution. In this and the networks in *B* and *C*, self-connections were not allowed. In each case 100 units were trained, but only the 25 units with the largest absolute selectivity index (Eq 30) are shown, ordered from most selective for choice 1 (large positive) to most selective for choice 2 (large negative). (B) A network trained for the same task as in *A* but with the constraint that excitatory units may only project positive weights and inhibitory units may only project negative weights. All input weights were constrained to be excitatory, and the readout weights, considered to be "long-range," were nonzero only for excitatory units. All connections except self-connections were allowed, but training resulted in a strongly clustered pattern of connectivity. Units are again sorted by selectivity but separately for excitatory and inhibitory units (20 excitatory, 5 inhibitory). (C) Same as *B* but with the additional constraint that excitatory recurrent units receiving input for choice 1 and excitatory recurrent units receiving input for choice 2 do not project to one another, and each group sends output to the corresponding choice.

- **نمودار بخش پیاده سازی اول تسک اول:**
  - نمودار اول : اثرگذاری سایز دنباله ورودی و اندازه ی ورودی در پاسخ (به ازای 2 بار ران مختلف!)

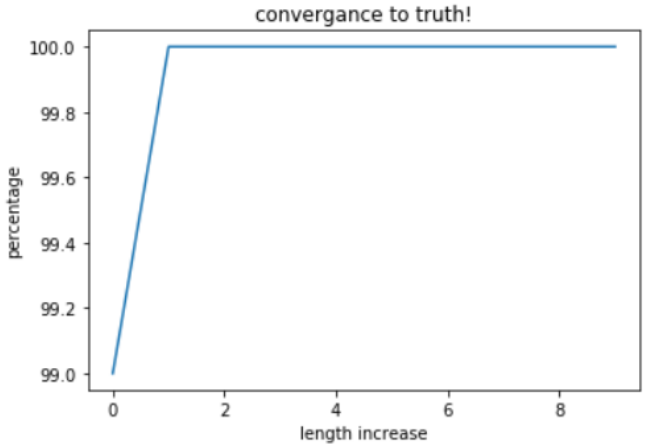


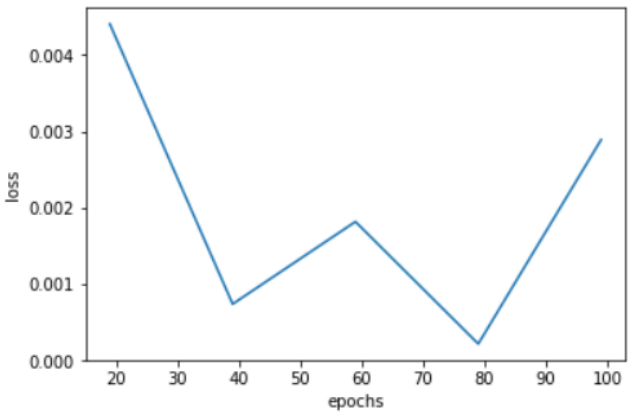
  - نمودار دوم : در ادامه نمودار های loss و میزان صحت را برای چند مرتبه سایز ورودی بررسی می کنیم.





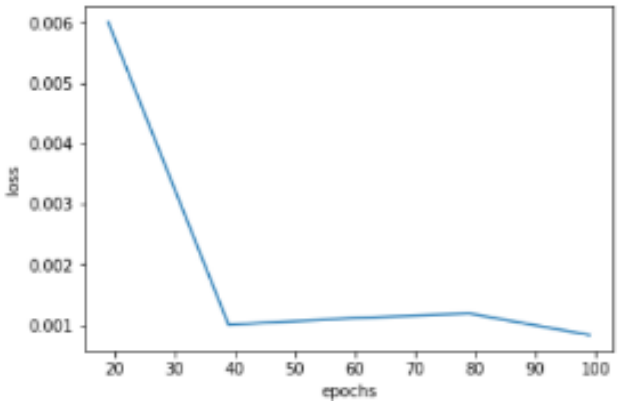
تغییر سایز ورودی : (در کد اصلی جاروب 10 مرحله ای زده ام و در هر مرحله نمودار loss و convergance to truth را ترسیم کرده ام.



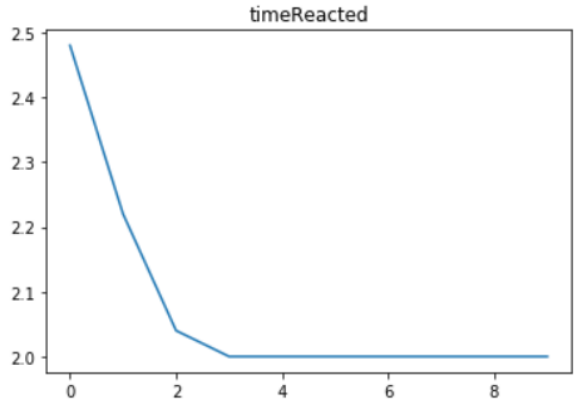


و هم چنین برای حالت های دیگر که به تدریج سایز ورودی افزایش پیدا می کند خواهیم داشت که درصد رخداد صحت خروجی به شدت زودتر به 100 درصد همگرا می گردد.

- **نمودار بخش پیاده سازی دوم تسک اول:**
  - نمودار loss function و convergence to truth



نمودار های های همگرایی به پاسخ درست و نمودار زمان واکنش دهی



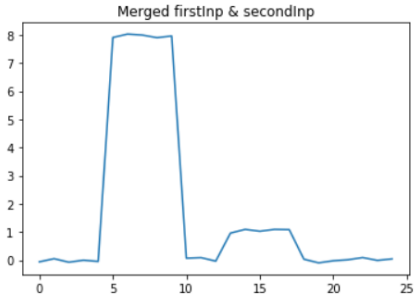
تسک دوم: Parametric Working Memory

تفاوت این تسک با تسک قبلی در این است که در این تسک به بررسی فرکانسی دو ورودی می پردازیم و البته منظورم از بررسی فرکانسی این است که ببینیم فرکانس کدام یک بیشتر وکدام یک کمتر است اما توجه کنید که دو ورودی به صورت پارالل وارد نمی شوند بلکه به صورت متوالی و با مدت زمانی مکث بین دو ورودی انجام می گیرد.

- **پیاده سازی دیتاست :**

برای پیاده سازی دیتاست ابتدا می دانیم که شبکه ی یک ورودی دارد لذا برای اینکه بتوانیم هر دو ورودی را به مدار بدهیم لازم است تا با یک فاصله ی زمانی معقول دو ورودی را با یکدیگر(به همراه مقدار درفاصله ی زمانی وسط صفر!) متصل می کنیم و به شبکه می دهیم و در عوض دو خروجی می گیریم که به ترتیب متناظر با فرکانس اول و فرکانس دوم می باشد لذا هر کدام که توسط شبکه بزرگتر تعیین گردد مقدار یک خروجی را از آن خود می کند.

نمونه ورودی اعمالی :



نکته : در ادامه خواهیم دید که هرچه مقدار تفاوت دو تحریک زیاد تر باشد تصمیم گیری برای شبکه راحت تر بوده و درصد پاسخ صحیح بیشتر خواهد شد.

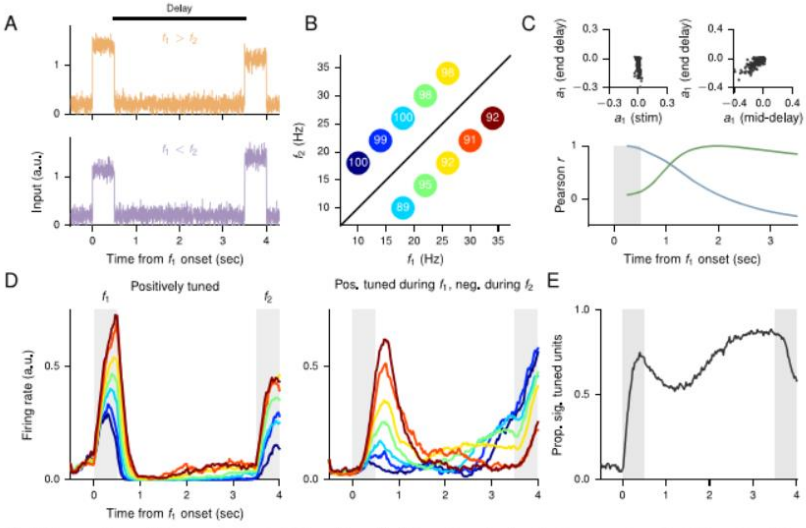
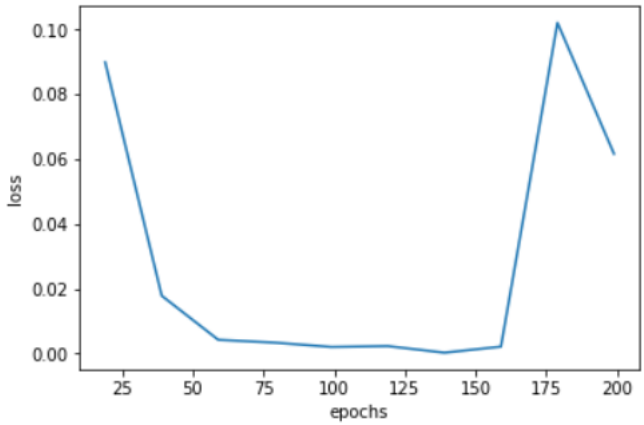


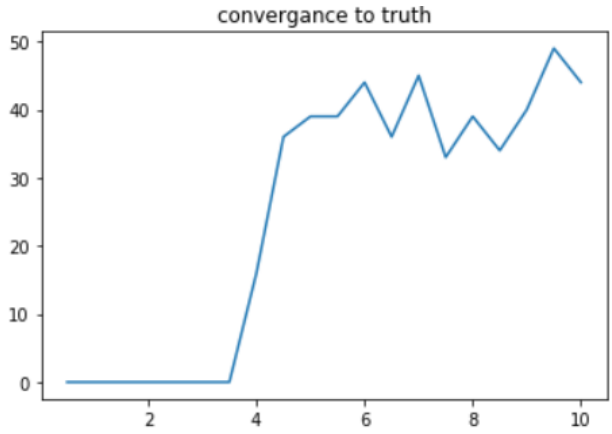
Fig 7. Parametric working memory task. (A) Sample positively tuned inputs, showing the case where $f_1 > f_2$ (upper) and $f_1 < f_2$ (lower). Recurrent units also receive corresponding negatively tuned inputs. (B) Percentage of correct responses for different combinations of f_1 and f_2. This plot also defines the colors used for each condition, labeled by f_1, in the remainder of the figure. Due to the overlap in the values of f_1, there are 7 distinct colors representing 10 trial conditions. (C) Lower: Correlation of the tuning a_1 (see text) at different time points to the tuning in the middle of the first stimulus period (blue) and middle of the delay period (green). Upper: The tuning at the end of delay vs. middle of the first stimulus (left) and the end of delay vs. middle of the delay (right). (D) Single-unit activity for a unit that is positively tuned for f_1 during both stimulus periods (left), and for a unit that is positively tuned during the first stimulus period but negatively tuned during the second stimulus period (right). (E) Proportion of significantly tuned units based on a simple linear regression of the firing rates as a function of f_1 at each time point.

نمودار ها:

- **loss Function**



- **convergence to truth**



تسک سوم : Sequence Execution

در این تسک به دنبال مدل کردن حرکت یک نقطه توسط چشم هستیم و خروجی شبکه ی عصبی ما محل چشم است و دارای دو مختصه ی طول و عرض است.

پیاده سازی تسک: **پیش از پیاده سازی تسک ابتدا مروری بر مطالب ذکر شده در مقاله می پردازیم.**

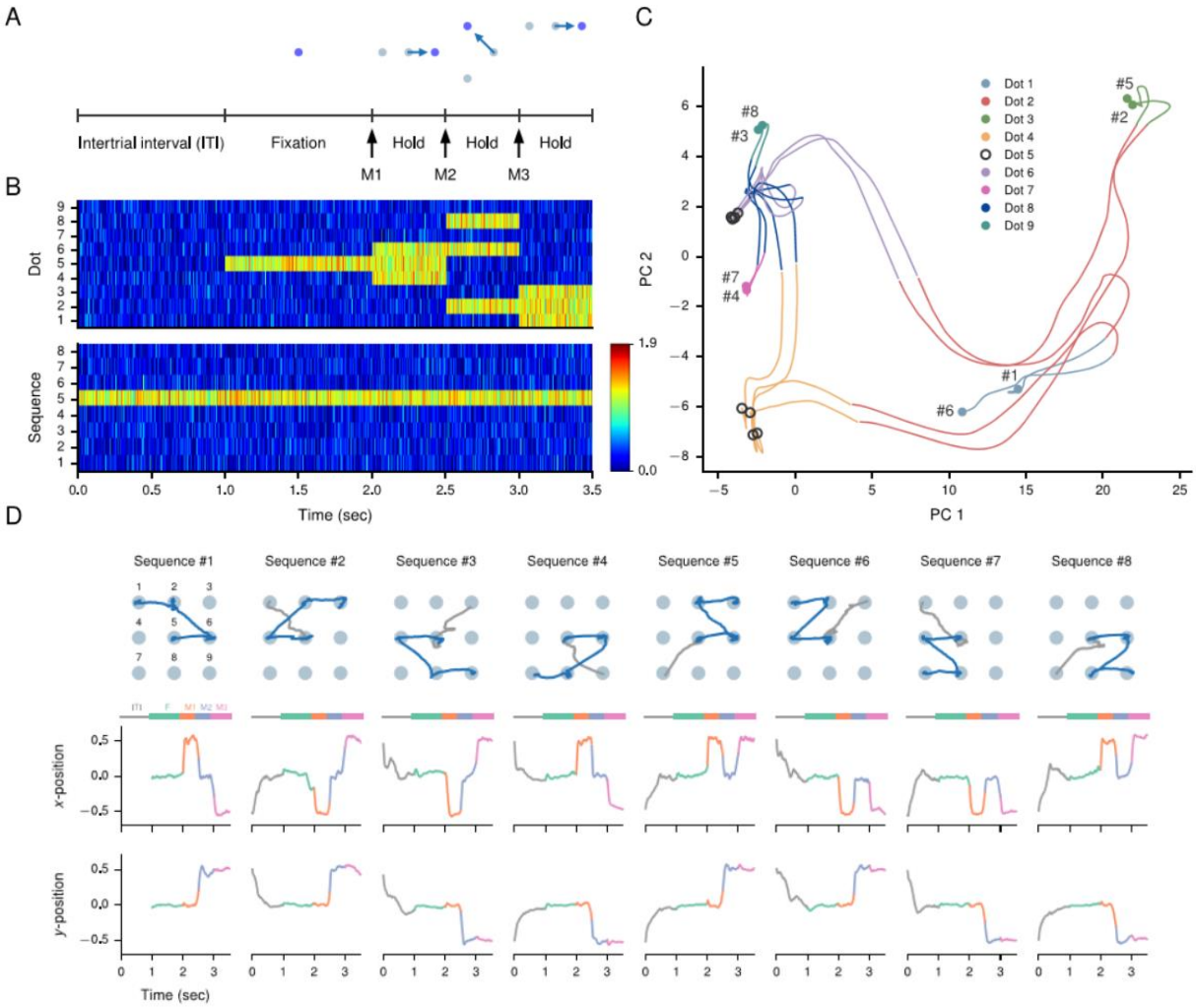


Fig 8. Eye-movement sequence execution task. (A) Task structure (for Sequence 5) and (B) sample inputs to the network. During the intertrial interval (ITI) the network receives only the input indicating the current sequence to be executed. Fixation is indicated by the presence of a fixation input, which is (the central) one of 9 possible dot positions on the screen. During each movement, the current dot plus two possible target dots appear. (C) State-space trajectories during the three movements M1, M2, and M3 for each sequence, projected on the first two principal components (PCs) (71% variance explained, note the different axis scales). The network was run with zero noise to obtain the plotted trajectories. The hierarchical organization of the sequence of movements is reflected in the splitting off of state-space trajectories. Note that all sequences start at fixation, or dot 5 (black), and are clustered here into two groups depending on the first move in the sequence. (D) Example run in which the network continuously executes each of the 8 sequences once in a particular order; the network can execute the sequences in any order. Each sequence is separated by a 1-second ITI during which the eye position returns from the final dot in the previous trial to the central fixation dot. Upper: Eye position in "screen" coordinates. Lower: *x* and *y*-positions of the network's outputs indicating a point on the screen. Note the continuity of dynamics across trials.

تولید دیتاست :

همانطور که متن مقاله به صورت دقیق به نحوه تولید دیتاست اشاره کرد داریم که دیتاست دارای دو بخش است

1- بخشی با 9 ورودی که متناظر با 9 محل موجود در صفحه است
2- بخشی با 8 ورودی که متناظر با یکی از 8 دنباله ی حرکتی موجود است

لذا در برآیند داریم که 17 = 8 + 9 ورودی به شبکه اعمال می شود. در ابتدا 9 ورودی اول همگی خاموش اند و سپس شروع اولیه ی حرکت (5 امین ورودی) که در وسط صفحه قرار دارد برای مدتی روشن می شود.در ادامه داریم که برای سه بازه ی متوالی در هربازه سه ورودی روشن هستند که متناظر با محل فعلی نقطه،و دو محل احتمالی بعدی آن هستند. هم چنین ورودی های دیگر که نگارنده ی نوع دنباله ی حرکتی اند(مورد دوم بالا) در تمام طول زمان اجرا ثابت اند و خروجی ها هم x,y متناظر با محل چشم است.

- **پیاده سازی تسک:**

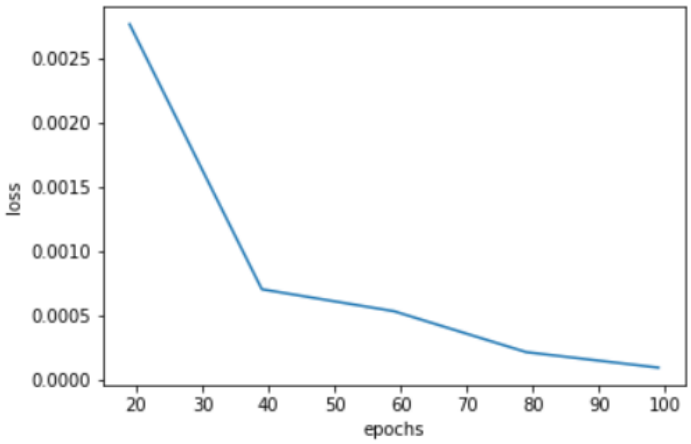
در این بخش ما به دو طریق training را انجام داده ایم.

1- تمام 17 ورودی موجود را در train شبکه به آن می دهیم و لذا طبیعی است که loss function با نرخ بیشتری میرا می گردد و در ادامه نمودار ها هم چنین چیزی را نشان می دهند.
2- در این حالت صرفا ورودی های مربوط به محل به شبکه برای انجام train داده می شود و کاری به دنباله ی حرکتی نداریم و خب منطقی است که در این حالت نسبت به حالت قبل با نرخ کمتری loss function میرا می گردد.(در ادامه نمودار ها را ضمیمه کرده ام).

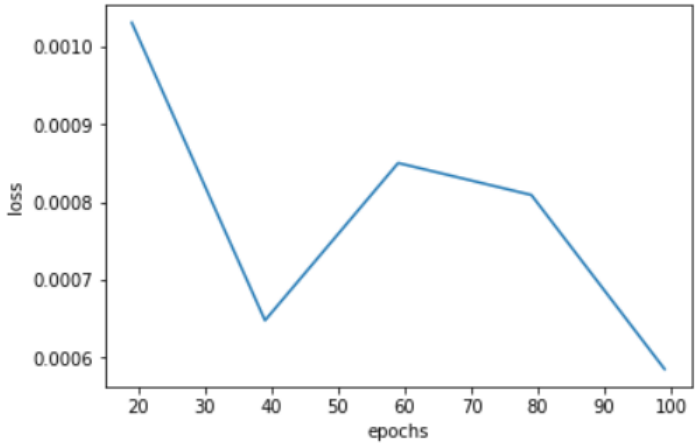
نمودار ها:

نمودار lossFunction :

- پیاده سازی حالت اول:



- پیاده سازی حالت دوم:



نتیجه گیری و استدلال :

همانطور که ذکر کردم از آنجا که در Train شماره اول از همه ی 17 تا ورودی استفاده کردیم لذا انتظار معقولی بود که طبق نکته ای که اشاره کردم loss function با rate نبستا خوبی کاهش پیدا کند اما در حالت دوم که از ورودی های حرکتی صرف نظر کردم طبیعتا نسبت به حالت اول باید با نرخ دیرتری loss function کاهش می یافت که همین نتیجه در نمودار های بالا مشاهده شده است.

سایر نمودار ها: چون تعداد نمودار ها خیلی زیاد می شد صرفا یک جهت حرکتی را مورد بررسی قرار می دهم.

همانطورکه انتظار می رفت در حالت Train کردن شبکه با مقدار محدود از ورودی ها خواهیم داشت که شباهت بین output و target نسبت به حالتی که شبکه به کمک دیتاست قوی تری و تعداد ورودی های بیشتری ترین شده است کمتر خواهد بود که همانطورکه در نمودار های موجود در شکل زیر می بینید همین اتفاق رخ می دهد و دو نمودار اول مربوط به ترین کردن 1 و دو نمودار دوم مربوط به ترین کردن 2 است.