RNN Tutorial

Robert Kim & Nuttida Rungratsameetaweemana

robert.f.kim@gmail.com & nr2869@columbia.edu

May 31, 2024

Contents

1 Basic Setup

1.1 Installing Anaconda

Anaconda allows you to create project-specific virtual environments. Refer to "Setting up conda and python.pdf" in the `server` channel on Slack to download and install Anaconda.

1.2 Setting up a new Anaconda environment

To create a new environment, run the following command in the terminal:

```
conda create --name rnn_test python=3.6
```

The above command will create a new virtual environment named `rnn_test`. The environment will be based on **Python 3.6**. The command will install some basic python packages.

Once the environment is created, we need to "activate" it in your current terminal:

```
conda activate rnn_test
```

Your current terminal should now show (`rnn_test`) before your username indicating that you are now in the `rnn_test` environment.

NOTE: If you are running conda for the first time, you might have to run the following commands before proceeding further:

```
conda init bash
source ~/.bashrc
```

To "deactivate", simply type

```
conda deactivate
```

To list all the existing virtual environments, type

```
conda env list
```

1.3 Installing other python packages

The environment we created above contains only basic packages. Here, we are going to install a few more important packages for plotting, array-processing, and machine learning.

Make sure you are in the `rnn_test` environment. Then run the following commands one at a time:

```
conda install -c anaconda numpy
```

```
conda install -c conda-forge matplotlib
```

```
conda install -c anaconda scipy
```

NOTE: You might have to upgrade numpy to 1.15 after installing everything for `matplotlib` to work.

```
conda install numpy=1.15
```

1.4 Installing Tensorflow

Tensorflow is a machine learning platform developed by Google. We will be using this package for constructing and training RNNs using backpropagation.

There are many versions, but my code is tested for Tensorflow 1.10.0. Run the following command to install:

```
pip install tensorflow==1.10.0
```

To enable Tensorflow to run on GPUs, we would also need to install an additional package (`tensorflow-gpu`). We will skip this for now, since the package will not run on machines without gpus (CUDA).

To make sure everything's installed correctly, first start python in your current terminal by typing "python" (without the quotation marks). Next, run the following commands:

```
import tensorflow as tf
print(tf.__version__)
```

If the output from the above command is "1.10.0", then everything is set up correctly!

2 Setting up Rate RNN Code

2.1 Overview

Continuous-variable rate RNNs, where recurrently connected units communicate via continuous signals (instead of discrete action potentials), have been widely utilized to uncover circuit mechanisms critical for performing various cognitive tasks [1–3]. Units in a continuous rate RNN are usually governed by the following set of equations:

$$\boldsymbol{\tau}\frac{d\boldsymbol{x}}{dt} = -\boldsymbol{x}(t) + \boldsymbol{w}\boldsymbol{r}(t) + \boldsymbol{w}_{in}\boldsymbol{u}(t) + \boldsymbol{\xi} \tag{1a}$$

$$\boldsymbol{r}(t) = \sigma(\boldsymbol{x}(t)) = \frac{1}{1+\exp(-\boldsymbol{x}(t))} \tag{1b}$$

$$\boldsymbol{o}(t) = \boldsymbol{w}_{out}\boldsymbol{r}(t) + b \tag{1c}$$

where $\boldsymbol{\tau} \in \mathbb{R}^{1\times N}$ refers to the synaptic time constants, $\boldsymbol{x} \in \mathbb{R}^{N\times T}$ represents the synaptic current variable from N units across T time-points, $\boldsymbol{r} \in \mathbb{R}^{N\times T}$ is the firing rates estimated by passing the synaptic current values ($\boldsymbol{x}$) through a nonlinear activation function (sigmoid in this case), $\boldsymbol{w}_{in} \in \mathbb{R}^{N\times N_{in}}$ defines connection weights from the time-varying inputs ($\boldsymbol{u} \in \mathbb{R}^{1\times T}$) to the network, and $\boldsymbol{w} \in \mathbb{R}^{N\times N}$ contains connection weights between N units. The output of the network ($\boldsymbol{o} \in \mathbb{R}^{1\times T}$) is a linear combination of all the firing rates specified by the output connection weight matrix, $\boldsymbol{w}_{out} \in \mathbb{R}^{1\times N}$, and the bias term, b. A schematic diagram illustrating a network producing a positive output signal upon receiving an input pulse is shown in fig. 1.

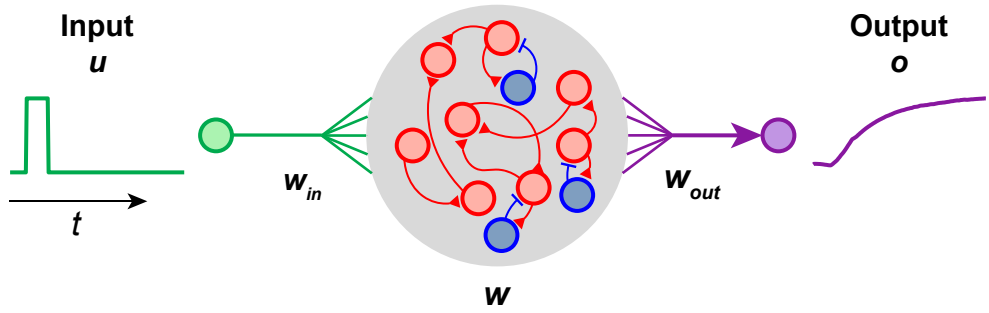

Figure 1: Schematic diagram illustrating a continuous RNN receiving a brief input pulse (green). The network consists of excitatory (red) and inhibitory (blue) units connected to one another (connection patterns specified by $\boldsymbol{w}$). The network output (purple) is a linear combination of the unit activities.

Eq. (1a) is discretized using the first-order Euler approximation method:

$$\begin{aligned} \boldsymbol{x}_t &= \left(1 - \frac{\Delta t}{\boldsymbol{\tau}}\right)\boldsymbol{x}_{t-1} + \frac{\Delta t}{\boldsymbol{\tau}}(\boldsymbol{w}\boldsymbol{r}_{t-1} + \boldsymbol{w}_{in}\boldsymbol{u}_{t-1}) \\ &+ \boldsymbol{\mathcal{N}}(0, 0.01) \end{aligned} \tag{2}$$

where $\Delta t = 5$ ms is the discretization time step size.

2.2 Cloning the RNN code

The code for training rate and spiking RNNs is available here: `https://github.com/rkim35/spikeRNN`.

If you are using git, you can clone the repository:

```
git clone https://github.com/rkim35/spikeRNN.git
```

The GitHub page lists and explains all the input parameters required for training rate RNNs.

2.3 Training rate RNNs

To train a rate RNN on a simple task, first activate the "rnn_test" environment if you haven't already:

```
conda activate rnn_test
```

Next, run the following command to train a rate RNN to perform the Go-NoGo task:

```
python main.py --gpu 0 --gpu_frac 0.20 --n_trials 5000 --mode train \
--N 200 --P_inh 0.20 --som_N 0 --apply_dale True --gain 1.5 --task go-nogo \
--act sigmoid --loss_fn l2 --decay_taus 4 20 --output_dir ../
```

The above command will initiate training which will take about 5 minutes to train on a decent GPU. Note that "\" indicates a line break. The trained network will contain 200 units (20% of the units are inhibitory). The training will stop if the termination criteria are met within the first 5000 trials (`n_trials`). No additional connectivity constraints are used (i.e. `som_N` is set to 0). The trained model will be saved as a MATLAB-formatted file (.mat) in the output directory (`../models/go-nogo/P_rec_0.20_Taus_4.0_20.0`).

NOTE: You can ignore the first input argument (`--gpu 0`). It is only relevant if you are training on a machine with GPUs.

2.4 Analyzing trained rate RNNs

The above command will save everything to a .mat file. The mat file contains several important parameters:

1. `w` ($N \times N$): connectivity matrix (not masked; more on this below)
2. `w_in` ($N \times 1$): input weight matrix
3. `w_out` ($1 \times N$): readout weights
4. `taus_gaus` ($N \times 1$): synaptic decay time constants (not normalized; more on this below)
5. `taus` (2×1): minimum and maximum synaptic decay time constants [min, max]

6. `inh` ($N \times 1$): binary vector indicating which units are inhibitory (1's for inhibitory units)
7. `exc` ($N \times 1$): binary vector indicating which units are excitatory (1's for excitatory units)

To compile the final connectivity matrix that satisfies Dale's principle, you need to multiply `w` by the mask matrix (`m`):

```
final_w = np.matmul(w, m)
```

To normalize the synaptic decay time constants (`taus_gaus`), the trained variable (`taus_gaus`) is passed through the sigmoid function:

```
taus_sig = (1/(1+np.exp(-taus_gaus))*(taus[1] - taus[0])) + taus[0]
```

References

1. Mante, V., Sussillo, D., Shenoy, K. V. & Newsome, W. T. Context-dependent computation by recurrent dynamics in prefrontal cortex. *Nature* **503,** 78–84 (2013).
2. Song, H. F., Yang, G. R. & Wang, X.-J. Training Excitatory-Inhibitory Recurrent Neural Networks for Cognitive Tasks: A Simple and Flexible Framework. *PLOS Computational Biology* **12,** e1004792 (2016).
3. Miconi, T. Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks. *eLife* **6,** e20899 (2017).
4. Kim, R., Li, Y. & Sejnowski, T. J. Simple framework for constructing functional spiking recurrent neural networks. *Proceedings of the National Academy of Sciences* **116,** 22811–22820. eprint: `https://www.pnas.org/content/116/45/22811.full.pdf` (2019).
5. Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* **78,** 1550–1560 (1990).
6. Yang, G. R., Joglekar, M. R., Song, H. F., Newsome, W. T. & Wang, X.-J. Task representations in neural networks trained to perform many cognitive tasks. *Nature Neuroscience* **22,** 297–306 (2019).

3 Python Rate RNN Class

3.1 Introduction to Python

To introduce basic python commands, here we will load the trained model from the previous section and plot some of the variables.

First import packages:

```
import os, scipy.io
import numpy as np
import matplotlib.pyplot as plt
```

`scipy.io` will be used to load the trained .mat file. `numpy` will be used for matrix operations. `matplotlib.pyplot` will be used for plotting (in MATLAB style).

Next, we will load the trained model:

```
model_dir = './spikeRNN/models/go-nogo/P_rec_0.2_Taus_4.0_20.0'
model_fname = 'Task_go-nogo_N_200_Taus_4.0_20.0_Act_sigmoid.mat'
mat_data = scipy.io.loadmat(os.path.join(model_dir, model_fname))
```

Make sure to change `model_dir` and `model_fname`!

All the variables saved in the mat file will be loaded into a dictionary variable (i.e., `mat_data`). The variable names you saw in MATLAB in the previous section are the keys for the dictionary. You can output the keys of the dictionary:

```
mat_data.keys()
```

You can access the value stored under each key by using either the square brackets or `get()` method. For example, you can run the following commands to access the connectivity weight matrix (`w`):

```
w = mat_data['w']
w = mat_data.get('w')
```

You can confirm the dimensions of the matrix by using the `shape` function in `numpy`:

```
print(w.shape)
```

Lastly, we can plot sample output signals from the trained model (`eval_os` variable in the mat file):

```
plt.figure();
plt.plot(np.transpose(mat_data['eval_os'][:,]));
plt.show()
```

Running the above commands should produce a figure that looks like fig. 3.

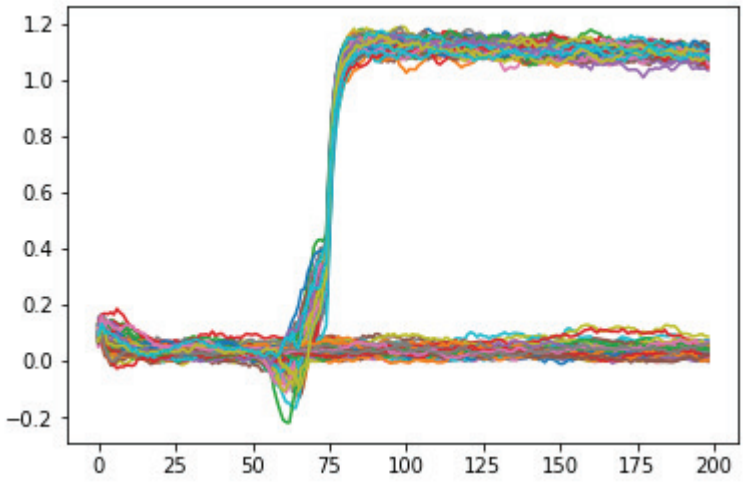

Figure 2: Example output signals from a rate model trained to perform the Go-NoGo task.

3.2 Repository overview

```
spikeRNN/
├── rate/ ..... contains Python files for constructing and training rate RNNs
│   ├── main.py ..... main script for training rate RNNs
│   ├── model.py ..... contains the implementation of the rate model. Also contains
│   │   implementations of several task paradigms
│   └── util.py ..... contains several general-purpose utility functions
└── spiking/ ..... contains MATLAB files for simulating spiking (LIF) RNNs
    ├── LIF_network_fnc.m ..... main script that implements the LIF model
    ├── lambda_grid_search.m . script for identifying an optimal scaling factor for converting a
    │   rate RNN to a spiking RNN
    └── eval_go_nogo.m  script to assess the task performance of a spiking RNN converted from
        a trained rate RNN on the Go-NoGo task
```

3.3 Rate RNN class

The continuous rate RNN model is implemented in `rate/model.py`. The model is defined as a Python class (`RN_RNN_dale`) and contains several attributes which can be found in the initialization method:

FR_RNN_dale initialization method

```
def __init__(self, N, P_inh, P_rec, w_in, som_N, w_dist, gain, apply_dale,
↪  w_out):
    """
    Network initialization method
    N: number of units (neurons)
    P_inh: probability of a neuron being inhibitory
    P_rec: recurrent connection probability
    w_in: NxN weight matrix for the input stimuli
    som_N: number of SOM neurons (set to 0 for no SOM neurons)
    w_dist: recurrent weight distribution ('gaus' or 'gamma')
    apply_dale: apply Dale's principle ('True' or 'False')
    w_out: Nx1 readout weights

    Based on the probability (P_inh) provided above,
    the units in the network are classified into
    either excitatory or inhibitory. Next, the
    weight matrix is initialized based on the connectivity
    probability (P_rec) provided above.
    """
    self.N = N
    self.P_inh = P_inh
    self.P_rec = P_rec
    self.w_in = w_in
    self.som_N = som_N
    self.w_dist = w_dist
    self.gain = gain
    self.apply_dale = apply_dale
    self.w_out = w_out

    # Assign each unit as excitatory or inhibitory
    inh, exc, NI, NE, som_inh = self.assign_exc_inh()
    self.inh = inh
    self.som_inh = som_inh
    self.exc = exc
    self.NI = NI
    self.NE = NE
    # Initialize the weight matrix
    self.W, self.mask, self.som_mask = self.initialize_W()
```

The model has several attributes, most of which should be provided as input arguments in `main.py` (covered in the next section):

- `N`: number of units in the model
- `P_inh`: proportion of the units that will be inhibitory
- `P_rec`: connectivity sparsity (i.e., connection probability)

- `w_in`: input weight matrix (see fig. 1)
- `som_N`: number of units that will be "somatostatin-expressing" inhibitory neurons (see [4])
- `w_dist`: probability density function for initializing the connectivity weight matrix (w)
- `gain`: constant gain factor that determines the standard deviation of the initial connectivity weight matrix (i.e., larger gain factor $\rightarrow$ larger deviation). This attribute is not super important (as the connectivity weights get updated via backpropagation).
- `apply_dale`: boolean attribute for applying Dale's principle
- `w_out`: readout weight matrix (see fig. 1)
- `inh`: $N \times 1$ binary vector with 1's indicating inhibitory units
- `som_inh`: $N \times 1$ binary vector with 1's indicating somatostatin inhibitory units
- `exc`: $N \times 1$ binary vector with 1's indicating excitatory units
- `NI`: total number of inhibitory units
- `NE`: total number of excitatory units

The initialization method above calls two helper functions (`assign_exc_inh` and `initialize_W`). The first helper function (`assign_exc_inh`) is used to randomly determine which units are excitatory vs. inhibitory during initialization. The method returns `inh`, `exc`, `NI`, `NE`, and `som_inh` (see above).

References

1. Mante, V., Sussillo, D., Shenoy, K. V. & Newsome, W. T. Context-dependent computation by recurrent dynamics in prefrontal cortex. *Nature* **503,** 78–84 (2013).
2. Song, H. F., Yang, G. R. & Wang, X.-J. Training Excitatory-Inhibitory Recurrent Neural Networks for Cognitive Tasks: A Simple and Flexible Framework. *PLOS Computational Biology* **12,** e1004792 (2016).
3. Miconi, T. Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks. *eLife* **6,** e20899 (2017).
4. Kim, R., Li, Y. & Sejnowski, T. J. Simple framework for constructing functional spiking recurrent neural networks. *Proceedings of the National Academy of Sciences* **116,** 22811–22820. eprint: `https://www.pnas.org/content/116/45/22811.full.pdf` (2019).
5. Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* **78,** 1550–1560 (1990).
6. Yang, G. R., Joglekar, M. R., Song, H. F., Newsome, W. T. & Wang, X.-J. Task representations in neural networks trained to perform many cognitive tasks. *Nature Neuroscience* **22,** 297–306 (2019).

4 Task Paradigms

If you want to add a new task paradigm or edit an existing task paradigm, the script that you want to edit is `model.py` under `rate/`. The script implements three task paradigms: Go-NoGo, delayed match-to-sample (aka XOR), and context-dependent input integration (aka "Mante" task). These tasks are described in detail in my papers. Each task paradigm has **two** functions associated with it: one for generating task-specific input signals and the other for generating target output signals. For example, the Go-NoGo task has these two functions:

- `generate_input_stim_go_nogo(settings)`
- `generate_target_continuous_go_nogo(settings, label)`

The function for generating task-specific input signals has one input argument (`settings`). The input argument is a dictionary variable that defines parameters important for the task. For example, we have the following setting for the Go-NoGo task (defined in `main.py`):

Go-NoGo task setting

```
# GO-NoGo task
settings = {
    'T': 200, # trial duration (in steps)
    'stim_on': 50, # input stim onset (in steps)
    'stim_dur': 25, # input stim duration (in steps)
    'DeltaT': 1, # sampling rate
    'taus': args.decay_taus, # decay time-constants (in steps)
    'task': args.task.lower(), # task name
}
```

In the above setting, the Go-NoGo task duration is 200 time steps (`T`). For the Go trials, the input pulse will be given at time step 50 (`stim_on`). The duration of the pulse is 25 time steps (`stim_dur`).

Based on these settings, the function (`generate_input_stim_go_nogo(settings)`) will generate **one** trial for the Go-NoGo task. The trial will be either NoGo (50% chance) or Go (50% chance) trial. The function will output the input signal (`u`) and the label for the generated trial (`label`; 0 for NoGo and 1 for Go).

The following code will generate 100 Go-NoGo trials (make sure to run this in `spikeRNN/rate`):

Generating 100 Go-NoGo trials

```
import os, scipy.io
import numpy as np
import matplotlib.pyplot as plt
from model import generate_input_stim_go_nogo

# GO-NoGo task
settings = {
    'T': 200, # trial duration (in steps)
    'stim_on': 50, # input stim onset (in steps)
    'stim_dur': 25, # input stim duration (in steps)
    'DeltaT': 1, # sampling rate
    'taus': 20, # decay time-constants (in steps)
    'task': 'go-nogo', # task name
}

inputs = np.zeros((100, settings['T']))
for i in range(100):
    u, label = generate_input_stim_go_nogo(settings)
    inputs[i, :] = u

# Plotting the generated trials
plt.figure()
plt.imshow(inputs)
plt.show()
```

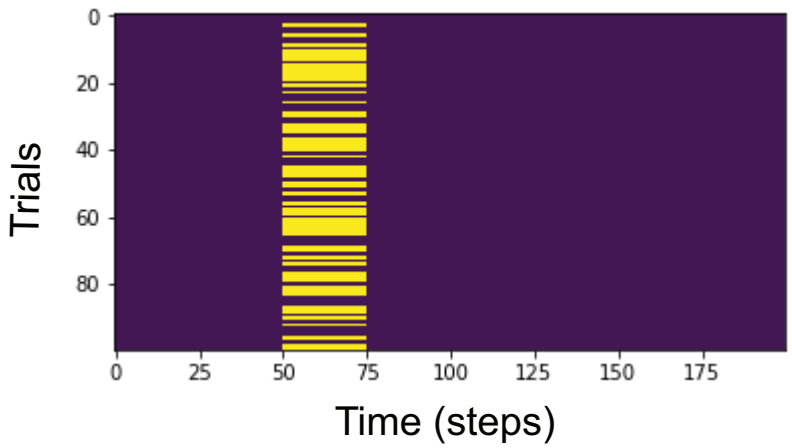

Figure 3: Example Go-NoGo trials (100 trials) generated from `generate_input_stim_go_nogo` section 4

Exercise

Change the above code to generate Go-NoGo trials with a longer pulse duration for the Go trials.

5 Evaluating trained RNNs

`eval_tf` function in `model.py` can be used to evaluate trained RNNs. The function takes three inputs: `model_dir`, `settings`, `u`.

Exercise

First, train an RNN to perform the Go-NoGo task. Once the training is done, use `eval_tf` to evaluate the task performance of the RNN on 100 randomly generated Go-NoGo trials.

6 BPTT and `TensorFlow` Overview

6.1 Backpropagation Through Time (BPTT) overview

Rate RNNs can be trained to produce output signals ($\boldsymbol{o}$) closely resembling target signals ($\boldsymbol{z} \in \mathbb{R}^{1\times T}$) associated with a specific task. A loss function ($\mathcal{L}$), which measures how close the RNN output signals are to the target signals, is employed to train and assess if the RNNs successfully learned the task. For example, the mean squared error (MSE) can be used to define the loss function:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}(z(t) - o(t))^2 = \frac{1}{T}\sum_{t=1}^{T}(z(t) - (\boldsymbol{w}_{out}\boldsymbol{r}(t) + b))^2 \tag{3}$$

where T is the total number of time points in a single trial.

Since the set of equations that govern the units (eq. (1)) are continuous and differentiable, a gradient-descent supervised method, known as backpropagation through time (BPTT; [5]), is often used to train rate RNNs to perform cognitive tasks [2, 6]. Given a set of n model parameters ($\boldsymbol{\theta} \in \{\theta_1, \theta_2, \ldots, \theta_n\}$), the gradient-descent algorithm tunes and optimizes the parameters to minimize the loss function ($\mathcal{L}$) in an iterative manner:

$$\theta_j^{(i+1)} = \theta_j^{(i)} - \eta\left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j^{(i)}}\right) \tag{4}$$

In eq. (4), $\theta_j^{(i)}$ is the j-th model parameter at iteration i, and η is the learning rate, which controls the rate of the convergence of the gradient descent. The model parameters include the synaptic time constants ($\boldsymbol{\tau}$), recurrent connectivity structure ($\boldsymbol{w}$), readout weights ($\boldsymbol{w}_{out}$), and bias (b). Therefore, we have $\boldsymbol{\theta} \in \{\boldsymbol{\tau}, \boldsymbol{w}, \boldsymbol{w}_{out}, b\}$.

For each model parameter, BPTT needs to compute the gradient (partial differential in eq. (4)). For the readout weights ($\boldsymbol{w}_{out}$), the gradient can be computed by simply differentiating the loss function (eq. (3)) with respect to $\boldsymbol{w}_{out}$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_{out}} = \frac{1}{T}\sum_{t}(-2)\cdot(z(t) - \boldsymbol{w}_{out}\boldsymbol{r}(t) - b)\cdot\boldsymbol{r}(t) \tag{5a}$$

$$= \frac{2}{T}\sum_{t}\boldsymbol{r}(t)\cdot(\boldsymbol{w}_{out}\boldsymbol{r}(t) + b - z(t)) \tag{5b}$$

$$= \frac{2}{T}\sum_{t}\boldsymbol{r}(t)\cdot(o(t) - z(t)) \tag{5c}$$

Similarly, differentiating the loss function with respect to the bias term (b) leads to the same gradient as eq. (5c).

For the recurrent connections ($\boldsymbol{w}$), computing the gradient is more involved. First, the gradient of the loss function at time t ($\mathcal{L}_t$) is defined as

$$\frac{\partial \mathcal{L}_t}{\partial w_{ik}} = \frac{\partial \mathcal{L}_t}{\partial r_t^{(i)}}\frac{\partial r_t^{(i)}}{\partial x_t^{(i)}}\frac{\partial x_t^{(i)}}{\partial w_{ik}} \tag{6}$$

where $r_t^{(i)}$ is the rate activity of unit i at time t, $x_t^{(i)}$ refers to the synaptic activity of unit i at time t, and $w_{ik} \in \boldsymbol{w}$ is the synaptic weight from unit k to unit i. For the first gradient on the righthand side (i.e., $\partial \mathcal{L}_t / \partial r_t^{(i)}$), the loss function needs to be first rewritten as:

$$\mathcal{L}_t = \frac{1}{T}\left(z_t - \left(\sum_{j=1}^{N} w_{out}^{(j)} r_t^{(j)} + b\right)\right)^2 \tag{7a}$$

$$= \frac{1}{T}\left(z_t - \left(w_{out}^{(i)} r_t^{(i)} + \sum_{\substack{j=1 \\ j\neq i}}^{N} w_{out}^{(j)} r_t^{(j)} + b\right)\right)^2 \tag{7b}$$

Differentiating eq. (7b) with respect to $r_t^{(i)}$ results in:

$$\frac{\partial \mathcal{L}_t}{\partial r_t^{(i)}} = -\frac{2}{T} \cdot w_{out}^{(i)} \cdot \left(z_t - \left(w_{out}^{(i)} r_t^{(i)} + \sum_{\substack{j=1 \\ j\neq i}}^{N} w_{out}^{(j)} r_t^{(j)} + b\right)\right) \tag{8a}$$

$$= -\frac{2}{T} \cdot w_{out}^{(i)} \cdot (z_t - o_t) \tag{8b}$$

The second gradient on the righthand side of eq. (6) (i.e., $\partial r_t^{(i)} / \partial x_t^{(i)}$) can be computed using eq. (1b):

$$\frac{\partial r_t^{(i)}}{\partial x_t^{(i)}} = \sigma(x_t^{(i)}) \cdot (1 - \sigma(x_t^{(i)})) \tag{9}$$

Lastly, the third gradient on the righthand side of eq. (6) (i.e., $\partial x_t^{(i)} / \partial w_{ik}$) can be computed by first substituting $\alpha = \Delta t / \boldsymbol{\tau}$ into the discretized version of eq. (1a) (using Euler approximation method):

$$\boldsymbol{x}_t = (1 - \alpha)\, \boldsymbol{x}_{t-1} + \alpha(\boldsymbol{w}\boldsymbol{r}_{t-1} + \boldsymbol{w}_{in}\boldsymbol{u}_{t-1}) + \boldsymbol{\xi} \tag{10}$$

Focusing only on unit i leads to:

$$x_t^{(i)} = (1 - \alpha) x_{t-1}^{(i)} + \alpha \left(\sum_{j=1}^{N} w_{ij} \cdot r_{t-1}^{(j)} + \boldsymbol{w}_{in}^{(i)} \boldsymbol{u}_{t-1}\right) + \boldsymbol{\xi} \tag{11}$$

Applying the chain rule results in:

$$\frac{\partial x_t^{(i)}}{\partial w_{ik}} = \sum_{t'=1}^{t} \left(\frac{\partial x_t^{(i)}}{\partial x_{t'}^{(i)}} \frac{\partial x_{t'}^{(i)}}{\partial w_{ik}}\right) \tag{12}$$

Due to the recurrent nature, $\partial x_t^{(i)} / \partial x_{t'}^{(i)}$ can be expressed as

$$\frac{\partial x_t^{(i)}}{\partial x_{t'}^{(i)}} = \prod_{q=t'+1}^{t} \frac{\partial x_q^{(i)}}{\partial x_{q-1}^{(i)}} \tag{13}$$

Substituting eq. (13) into eq. (12) leads to

$$\frac{\partial x_t^{(i)}}{\partial w_{ik}} = \sum_{t'=1}^{t} \left\{ \left(\prod_{q=t'+1}^{t} \frac{\partial x_q^{(i)}}{\partial x_{q-1}^{(i)}} \right) \frac{\partial x_{t'}^{(i)}}{\partial w_{ik}} \right\} \tag{14}$$

The gradient of the loss function with respect to w_{ik} can be calculated by summing over all the time points:

$$\frac{\partial \mathcal{L}}{\partial w_{ik}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial w_{ik}} = \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}_t}{\partial r_t^{(i)}} \frac{\partial r_t^{(i)}}{\partial x_t^{(i)}} \frac{\partial x_t^{(i)}}{\partial w_{ik}} \right) \tag{15}$$

6.2 TensorFlow overview

Tensorflow is an open source deep learning library from Google. If you followed the instructions from the beginning (section 1.4), you should have it properly installed.

In general, writing a TensorFlow script involves:

1. Constructing a computational graph
2. Executing the computational graph within a TensorFlow session

A computational graph in TensorFlow is composed of nodes representing variables or operations. We will use the following example to illustrate this.

TensorFlow example

```
import tensorflow as tf

# Construct a computational graph
input_1 = tf.constant(3, name='input_1')
input_2 = tf.constant(10, name='input_2')
prod = tf.multiply(input_1, input_2, name='Mult_op')

# Run the above graph in a TF session
with tf.Session() as sess:
    writer = tf.summary.FileWriter("tf_example", sess.graph)
    print(sess.run(prod))
    writer.close()
```

The above code creates a very simple computational graph composed of two "input" nodes which are set to constant values (3 and 10). The graph also involves a third node that multiples the two input nodes (`prod`).

The second part of the code then activates a TensorFlow session (`tf.Session()`) and executes the graph. It also writes the graph into a file (`FileWriter`). After running the code, you should have a folder named `tf_example` created in the same directory as the code. That folder will contain files that are named somethint like `events.out.tfevents...`. These files can then be loaded using TensorBoard to visualize the constructed graph. Open another terminal window and run the following command in the same directory as the code:

```
tensorboard --log_dir tf_example
```

While that command is running, open a web browser (chrome, firefox, etc...) and go to `http://localhost:6006`. If everything ran without any errors, you should see the graph that looks like this:

Consistent with our code, our graph contains two input nodes (`input_1` and `input_2`) along with the multiplication node (`Mult_op`).

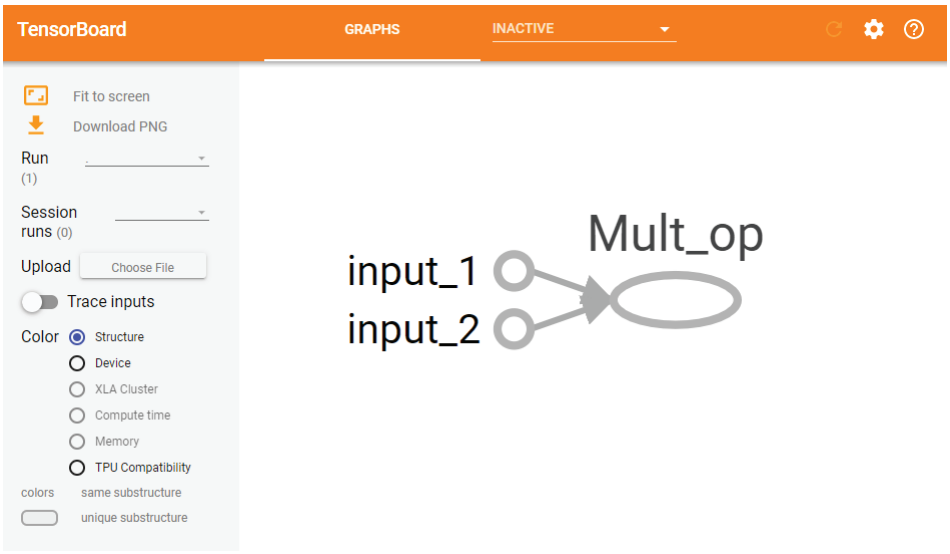

Figure 4: Using TensorBoard to visualize computational graphs

Exercise

Modify section 6.2 to add another input node and a node that sums all three input nodes.

If you are running your TensorFlow code and TensorBoard on a remote machine, then you can forward everything to your local machine by first ssh'ing to the remote machine using the following terminal command:

```
ssh -L 16006:127.0.0.1:6006 username@remote_machine_IP
```

The above command will forward everything on port 6006 of the remote machine to port 16006 on your local machine. Once you logged into your remote machine using the command, then run the TensorBoard command.

6.3 XOR gate implementation

In this section, we will implement a TensorFlow model that can solve the classical XOR (exclusive OR) problem to introduce a few additional TensorFlow features. The XOR problem involves comparing two input values (A and B). If both A and B are turned off (i.e., set to 0) or on (set to 1), then the output should be 0. If only one of the input variables is turned on, then the output should be 1.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR logic

We will construct the following graph (fig. 5) to solve the XOR problem:

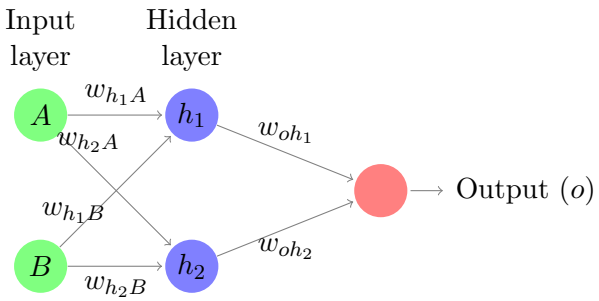

Figure 5: XOR gate computational graph

Based on the graph above (fig. 5), we have

$$\begin{aligned} h_1 &= \sigma(w_{h_1A}A + w_{h_1B}B + b_1) \\ h_2 &= \sigma(w_{h_2A}A + w_{h_2B}B + b_2) \\ o &= \sigma(w_{oh_1}h1 + w_{oh_2}h2 + b_3) \end{aligned} \tag{16}$$

Now, we derive the cost function for this task. Given the input samples (x), we want to **maximize** the following likelihood:

$$L = P(y|x) = \prod_{i=1}^{m} o_i^{y_i}(1 - o_i)^{1-y_i}$$

where $y_i \in \{0, 1\}$ for the i-th case. We apply log on both sides to get:

$$\log(L) = \sum_{i=1}^{m} y_i \log(o_i) + (1 - y_i) \log(1 - o_i) \tag{17}$$

Traditionally, we want to **minimize** $-\log(L)$:

$$\hat{\theta} = \arg\min_{\theta}(-\log(L))$$

Now that we have defined everything, we can start implementing in TensorFlow:

```
import tensorflow as tf

x = tf.placeholder(tf.float32, shape=[4, 2], name = 'input')
y = tf.placeholder(tf.float32, shape=[4, 1], name = 'prediction')
```

As shown above, we are using `tf.placeholder` to reserve the input and output variables (x and y, respectively). The input dimension is $[4 \times 2]$ as there are four possible cases for the two input variables (A and B in table 1). The dimension for the output variable is $[4 \times 1]$ (i.e., binary vector for the four cases).

Next, we define and initialize the connectivity weights (w_{h_1A}, w_{h_2A}, w_{h_1B}, w_{h_2B}) from the input to the hidden layer:

```
w_HI = tf.Variable(tf.random_normal((2, 2)), name = 'input_to_hidden')
```

We initialized the connectivity weight matrix ($[2 \times 2]$) using `tf.random_normal` (random numbers generated from a normal Gaussian distribution).

Similarly, we define and initialize the connections from the hidden layer to the output node:

```
w_OH = tf.Variable(tf.random_normal((2, 1)), name = 'hidden_to_output')
```

Note the dimension of the matrix ($[2 \times 1]$).

We now define and initialize the bias terms (b_1, b_2, and b_3 in eq. (16)):

```
b_12 = tf.Variable(tf.zeros((2, )), name = 'biases1_2')
b_3 = tf.Variable(tf.zeros((1, )), name = 'bias3')
```

Now that we have finished defining and initializing all the necessary variables, we can construct the forward pass:

```
o_ = tf.sigmoid(tf.matmul(x, w_HI) + b_12)
o = tf.sigmoid(tf.matmul(o_, w_OH) + b_3)
```

Using the predictions (o) obtained from the forward pass, we can then define the cost function by using eq. (17):

```
L = y*tf.log(o) + (1 - y)*tf.log(1 - o)
loss = tf.reduce_mean(-L)
```

Since we want to minimize the loss function, we define an optimizer and instruct it to minimize the loss value:

```
lr = 0.01 # learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate = lr)
training_op = optimizer.minimize(loss)
```

This concludes the graph construction part for the XOR task. The following snippet summarizes everything we have done so far:

```
"""
Part 1. Graph construction
"""
import tensorflow as tf

# Input and output nodes
x = tf.placeholder(tf.float32, shape=[4, 2], name = 'input')
y = tf.placeholder(tf.float32, shape=[4, 1], name = 'prediction')

# Connectivity weight matrices
w_HI = tf.Variable(tf.random_normal((2, 2)), name = 'input_to_hidden')
w_OH = tf.Variable(tf.random_normal((2, 1)), name = 'hidden_to_output')

# Constant bias terms
b_12 = tf.Variable(tf.zeros((2, )), name = 'biases1_2')
b_3 = tf.Variable(tf.zeros((1, )), name = 'bias3')

# Forward pass
o_ = tf.sigmoid(tf.matmul(x, w_HI) + b_12)
o = tf.sigmoid(tf.matmul(o_, w_OH) + b_3)

# Loss
L = y*tf.log(o) + (1 - y)*tf.log(1 - o)
loss = tf.reduce_mean(-L)

# Optimizer and training operation
lr = 0.01 # learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate = lr)
training_op = optimizer.minimize(loss)
```

The second part of the script involves training the graph that we just constructed. We need to first define the four input cases along with the ground-truth output values:

```
import numpy as np

# Four possible input cases
X = np.zeros((4, 2))
X[1, 1] = 1
X[2, 0] = 1
X[3, :] = 1

# Output labels
Y = np.zeros((4, 1))
Y[1:3] = 1
\end{lstlisting}

Next, we need to activate a TensorFlow session:
\begin{lstlisting}[numbers=none, language=python]
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

Finally, we will train our network over multiple trials (1000 trials):

```
trained = False
counter = 0
while trained == False:
    _, out, lo = sess.run([training_op, o, loss], feed_dict = {x: X, y: Y})
    if counter%5000 == 0:
        print(lo)
        if lo < 0.10:
            trained = True
            break
    counter += 1
print(out)
```

The above code trains the network and checks every 5000 trials if the loss value (`lo`) becomes sufficiently small (0.10). Running the code should print out the loss value every 5000 trials and the final prediction values:

```
0.72176915
0.68415034
0.6614804
0.58664054
0.38935828
0.19854748
0.115661845
0.07815207
[[0.0761022 ]
 [0.9261108 ]
 [0.9121122 ]
 [0.06265213]]
```

As you can see, the final prediction values are close to the actual ground-truth values (`Y`).

Exercise

What happens if you remove the nonlinear activation function (σ) in the hidden layer output (i.e., `o_`)?

The full version of the script is shown below:

XOR gate implementation in TensorFlow

```
"""
Part 1. Graph construction
"""
import tensorflow as tf

# Input and output nodes
x = tf.placeholder(tf.float32, shape=[4, 2], name = 'input')
y = tf.placeholder(tf.float32, shape=[4, 1], name = 'prediction')

# Connectivity weight matrices
w_HI = tf.Variable(tf.random_normal((2, 2)), name = 'input_to_hidden')
w_OH = tf.Variable(tf.random_normal((2, 1)), name = 'hidden_to_output')

# Constant bias terms
b_12 = tf.Variable(tf.zeros((2, )), name = 'biases1_2')
b_3 = tf.Variable(tf.zeros((1, )), name = 'bias3')

# Forward pass
o_ = tf.sigmoid(tf.matmul(x, w_HI) + b_12)
o = tf.sigmoid(tf.matmul(o_, w_OH) + b_3)

# Loss
L = y*tf.log(o) + (1 - y)*tf.log(1 - o)
loss = tf.reduce_mean(-1*L)

# Optimizer and training operation
lr = 0.01 # learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate = lr)
training_op = optimizer.minimize(loss)
```

XOR gate implementation in TensorFlow cont.

```
"""
Part 2. Execution
"""
import numpy as np

# Four possible input cases
X = np.zeros((4, 2))
X[1, 1] = 1
X[2, 0] = 1
X[3, :] = 1

# Output labels
Y = np.zeros((4, 1))
Y[1:3] = 1

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

trained = False
counter = 0
while trained == False:
    _, out, lo = sess.run([training_op, o, loss], feed_dict = {x: X, y: Y})
    if counter%5000 == 0:
        print(lo)
        if lo < 0.10:
            trained = True
            break
    counter += 1
print(out)
```