Monte Carlo Methods
Lecture 18: Monte Carlo Techniques in Artificial Intelligence

Dieter W. Heermann

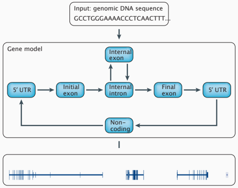
January 23, 2021

Heidelberg University

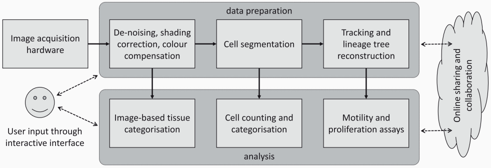
Table of Contents

Introduction

Introduction I



Neural network application to gene finding. Image taken from [1].


data preparation
Image acquisition hardware
De-noising, shading correction, colour compensation
Cell segmentation
Tracking and lineage tree reconstruction
User input through interactive interface
Image-based tissue categorisation
Cell counting and categorisation
Motility and proliferation assays
analysis
Online sharing and collaboration

Cell imaging analysis with neural networks. Image taken from [2].

Introduction II

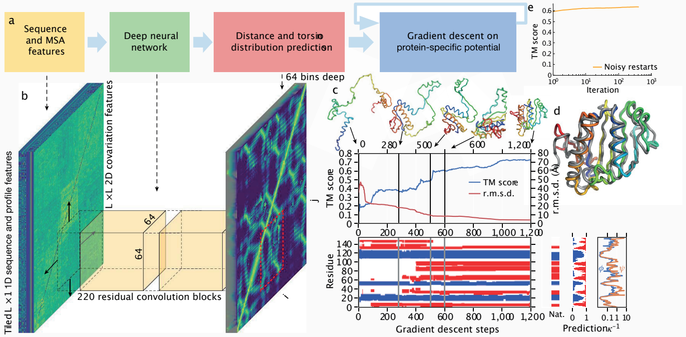

a
Sequence and MSA features
Deep neural network
Distance and torsion distribution prediction
Gradient descent on protein-specific potential
b
Tiled L × 1 1D sequence and profile features
L × L 2D covariation features
64
64
220 residual convolution blocks
64 bins deep
i
j
c
0
280
500
600
1,200
TM score
r.m.s.d. (Å)
TM score
r.m.s.d.
Residue
Gradient descent steps
Nat.
Prediction
κ⁻¹
φ
ψ
d
e
Noisy restarts
Iteration

Fig. 2 | The folding process illustrated for CASP13 target T0986s2. CASP target T0986s2, L = 155, PDB: 6N9V. a, Steps of structure prediction. b, The neural network predicts the entire L × L distogram based on MSA features, accumulating separate predictions for 64 × 64-residue regions. c, One iteration of gradient descent (1,200 steps) is shown, with the TM score and root mean square deviation (r.m.s.d.) plotted against step number with five snapshots of the structure. The secondary structure (from SST[33]) is also shown (helix in blue, strand in red) along with the native secondary structure (Nat.), the secondary structure prediction probabilities of the network and the uncertainty in torsion angle predictions (as κ⁻¹ of the von Mises distributions fitted to the predictions for φ and ψ). While each step of gradient descent greedily lowers the potential, large global conformation changes are effected, resulting in a well-packed chain. d, The final first submission overlaid on the native structure (in grey). e, The average (across the test set, n = 377) TM score of the lowest-potential structure against the number of repeats of gradient descent per target (log scale).

Figure 1: Image taken from: Improved protein structure prediction using potentials from deep learning Nature 2020 [3].

Introduction III

(Machine) learning can be roughly categorized into supervised and unsupervised. Typical techniques include:

- supervised methods:
  - Artificial Neural Network,
  - Support Vector Machines and linear classifiers
  - Bayesian Statistics,
  - k-Nearest Neighbors,
  - Hidden Markov Model
  - Decision Trees
- un-supervised methods
  - Autoencoders,
  - Expectation Maximization,
  - Self-Organizing Maps,
  - k-Means
  - Fuzzy clustering
  - Density-based clustering.

Methods developed and applications of machine learning in biophysical problems [1] range from finding genes, as featured in the introduction to the analysis of images such as computer tomography spanning the entire variety of bio-biological problems.

Markov Decision Process

Introduction I

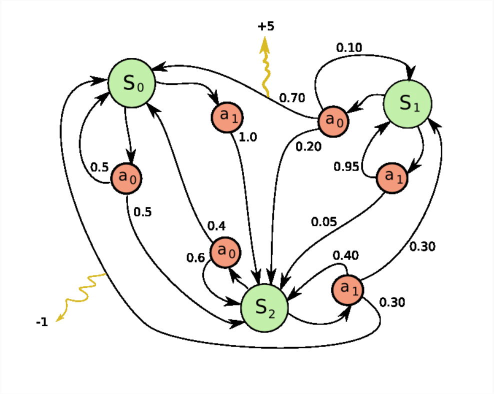


Figure 2: Markov decision process. Image taken from: https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690.

Introduction II

Let S be a state space (a countable non-empty set) , A be the action space (countable non-empty set of actions) and O the observation space. Let P_0 be a transition probability kernel that assigns to $(S = s, A = a) \in S \times A$ a probability measure over $S \times \mathbb{R} : P_0(\cdot \mid s, a)$.

A countable Markov Decision Process (MDP) is defined as a triple $M = (S, A, P_0)$. We further define a reward function

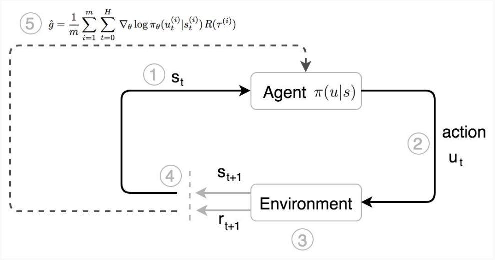
$$R(s,a) = \mathbb{E}[R \mid S = s, A = a] = \int_{\mathbb{R}} \sum_{s' \in S} R \cdot P_0(s', R \mid s, a) dR \,. \tag{1}$$

Furthermore let

$$R(s,a,s') = \mathbb{E}[R \mid S_t = s, A = a, S_{t+1} = s'] = \int_{\mathbb{R}} R \cdot P_0(s', R \mid s, a) dR \,. \tag{2}$$

A Markov Reward Process (MRP) is a Markov process with a reward function. Hence a tuple (S, P, R, γ). γ is a discount factor, where $\gamma \in [0, 1]$.

Introduction III



5 $\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\,R(\tau^{(i)})$
1 s_t
Agent $\pi(u|s)$
2 action u_t
4 s_{t+1}
Environment
r_{t+1}
3

Figure 3: Markov decision process, Figure taken from: https://medium.com/@jonathan-hui/rl-policy-gradients-explained-9b13b688b146.

From the definition it is clear that for a sequence $S_1, A_1 \ldots, S_{t-1}, A_{t-1}, S_t, A_t, S_{t+1}, A_{t+1}$ we have

$$\mathbb{P}[S_{t+1}, R_{t+1} \mid S_t; A_t] = \mathbb{P}[S_{t+1}, R_{t+1} \mid S_1, A_1 \ldots, S_{t-1}, A_{t-1}, S_t; A_t] \,. \tag{3}$$

Thus the sequence is Markovian

At each step t an agent:

Introduction IV

- Receives observation o_t
- Receives (immediate) scalar reward R_t
- Executes action at a_t

The environment:

- Receives action a_t
- Emits observation O_{t+1}
- Emits scalar reward R_{t+1}

If $s_t = o_t$ the environment is fully observable. Let G_t be the total discounted rewards from time step t

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \ . \tag{4}$$

Deep Learning

Recall that the **gradient descent** method computes the gradient of the cost function J w.r.t. to a parameter θ

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \ . \tag{5}$$

with the rate α. Choosing a proper learning rate can be difficult. Usually a learning rate schedule is used where α progressively decreases with the number of iterations. The challenge is that mostly the function that we want to minimize is highly non-convex so that there is a high probability to get trapped in numerous suboptimal local minima.

Stochastic gradient descent (SGD) performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \ . \tag{6}$$

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily.

Algorithm 1 Stochastic gradient descent (SGD)

```
1: for number of epochs do
2:     randomly shuffle data $(x, y)$
3:     for number of data do
4:         $\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$
5:     end for
6: end for
```

Mini-batch gradient descent: update for every mini-batch of n training examples

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \; . \tag{7}$$

Algorithm 2 Mini-batch gradient descent

```
1: for number of epochs do
2:     randomly shuffle data $(x, y)$
3:     for batch in take out a batch from data of size $m$ do
4:         $\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
5:     end for
6: end for
```

Gradient descent optimization algorithms

SGD with momentum [4]. Let $\gamma < 1$ be the resistance, then

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \ . \end{aligned} \tag{8}$$

The **Adaptive Moment Estimation (Adam)** [5] is often used in packages like Tensorflow [6] for the gradient descent:

Deep Learning IV

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \ . \end{aligned} \tag{9}$$

Adam stores an exponentially decaying average of past squared gradients v_t and keeps an exponentially decaying average of past gradients m_t. They are estimates of the first moment and the second moment of the gradients respectively producing a bias ($\beta_1 \approx 1$ and $\beta_2 \approx 1$). This being opposed by

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{10}$$

leading to the actual gradient descent

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \ . \tag{11}$$

Neural Networks: Introduction I

The general goal is to create **artificial neural networks** (graphs) (**ANN**) that imitate to some extend the capabilities of the human brain:

- learning
- generalization
- adaptivity
- fault tolerance
- ...

We want this for example for

- pattern classification
- function approximation
- ...

Neural Networks: Introduction II

Pioneering work was done by McCulloch and Pitts with the Perceptron [7, 8]. This was extended by Minsky and Papert [9].

McCulloch and Pitts proposed a binary threshold model as a computational model for an artificial neuron. Let $x_1, ..., x_n$ be the input values and $y = 0, 1$ be the output. The perceptron is defined by

$$y = \begin{cases} 0, & \sum_i x_i w_i \leq b \\ 1, & \sum_i x_i w_i > b \end{cases} . \tag{12}$$

where $w_1, ..., w_n$ are the synaptical weights that Rosenblat [8] introduced (see Figure 4). This can be reformulated as

$$y = \Theta(\sum_{j=1}^{n} w_j x_j - b) . \tag{13}$$

This generates an output of 1 if the sum is above a certain threshold. Sometimes we include b in the sum and set $w_0 = -b$ and x_0 to a constant input $x_0 = 1$.

Neural Networks: Introduction III

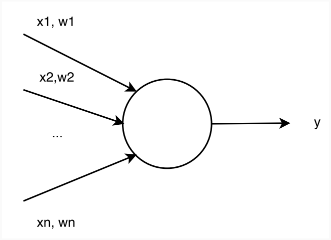


Figure 4: Perceptron.

In this setting

- positive weights correspond to **excitatory synapses**
- negative weights correspond to **inhibitory synapses**.

Clearly one can also use other activation function like

- piecewise linear
- sigmoid neuron

Neural Networks: Introduction IV

- gaussian.

Most often used is the sigmoid function (here the **logistic function**)

$$g(x) = \frac{1}{1 + e^{-\beta x}} \ . \tag{14}$$

The above constructed node is then the basic unit in a network (graph) of nodes. Thus the ANN's are weighted directed graphs where

$$\begin{aligned} \text{neuron} &\cong \text{node} \qquad (15) \\ \text{connection between neuron} &\cong \text{directed edge with weights} \qquad (16) \end{aligned}$$

Example: XOR

Neural Networks: Introduction V

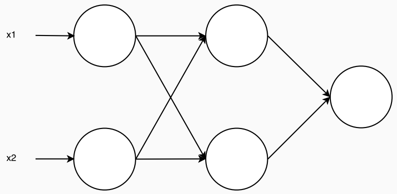


Figure 5: Perceptron XOR.

Connectionist models for gene regulation in the form of recurrent Hopfield [10] networks have been proposed by Mjolsness and others [11–13] to describe regulatory networks as directed graphs or matrices of interactions without restrictions on connectivity. These continuous time networks model interphase expression of a cell based on interaction weights that are free to take positive and negative real values.

Neural Networks: Introduction VI

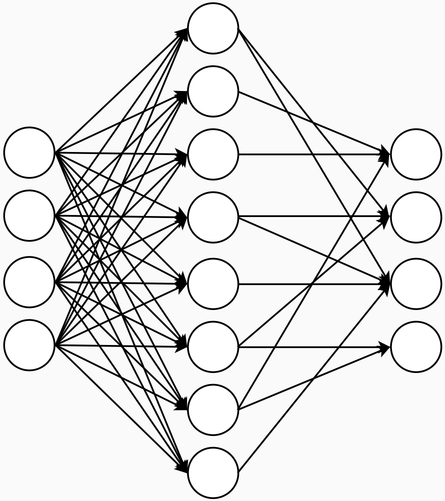


Figure 6: Example of a neural network topology with input and output layers and one hidden layer.

Neural Networks: Introduction VII

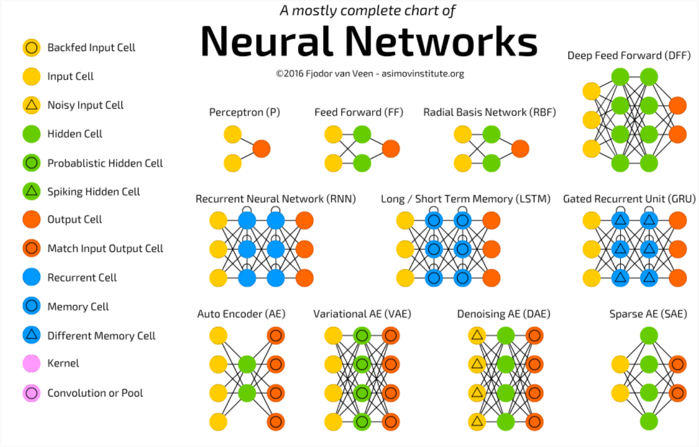


Figure 7: Overview of possible architectural designs of neural networks. Image taken from **https://medium.com/@carynmccarthy15/a-beginners-guide-to-recurrent-neural-networks-bfacb27bddb6**.

Backpropagation I

Prerequisite: Let $s \odot t$ ($(s \odot t)_j = s_j t_j$) denote the Hadamard product (Schur product), i.e., the elementwise product of the two vectors

Let $C(w, b)$ be a cost function where w is a weight and b is a bias with two assumptions about the form of the cost function:

- a
- b

quadratic cost function

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2 , \tag{26}$$

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2} \sum_j (y_j - a_j^L)^2 . \tag{27}$$

The goal of backpropagation is to compute the partial derivatives $\partial C / \partial w$ and $\partial C / \partial b$

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) , \tag{17}$$

Backpropagation II

$$a^l = \sigma(w^l a^{l-1} + b^l)\ . \tag{18}$$

to compute a^l we compute $z^l \equiv w^l a^{l-1} + b^l$ weighted input to the neurons in layer l

$a^l = \sigma(z^l)$

z^l

$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$

z^l_j is just the weighted input to the activation function for neuron j in layer l.

Let δ^l_j be the error in the jth neuron in the lth layer:

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}\ . \tag{29}$$

Backpropagation provides a procedure to compute the error δ^l_j. δ^l denotes the vector of errors associated with layer l.

For the error in the output layer L we have

Backpropagation III

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{19}$$

Proof: We have

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{20}$$

and with the chain rule we obtain

$$\begin{aligned} \delta_j^L &= \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \qquad (21)\\ &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \; . \qquad (22) \end{aligned}$$

because the sum is over all neurons k in the output layer and the output activation a_k^L of the kth neuron depends only on the weighted input z_j^L for the jth neuron when $k = j$. If $k \neq j$ $\partial a_k^L / \partial z_j^L$ is zero.

Backpropagation IV

Since $a_j^L = \sigma(z_j^L)$ we write $\sigma'(z_j^L)$ we have Equation 22

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{23}$$

proving our assumption.

$$\delta^L = \nabla_a C \odot \sigma'(z^L)\ . \tag{24}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L)\ . \tag{25}$$

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)\ , \tag{26}$$

Proof: We rewrite $\delta_j^l = \partial C / \partial z_j^l$ in terms of $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$

Backpropagation V

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{27}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{28}$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \; . \tag{29}$$

Note that

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \tag{30}$$

and taking the derivative

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \; . \tag{31}$$

we get

Backpropagation VI

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \; . \tag{32}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{33}$$

$$\frac{\partial C}{\partial b} = \delta, \tag{34}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{35}$$

$$\frac{\partial C}{\partial w} = a_{\mathrm{in}} \delta_{\mathrm{out}}, \tag{36}$$

Backpropagation VII

Algorithm 3 Backpropagation Algorithm (single input)

```
1:  repeat
2:      Set the corresponding activation $a^1$ for the input layer
3:      for $l = 2, 3, \dots, L$ do
4:          $z^l = w^l a^{l-1} + b^l$
5:          $a^l = \sigma(z^l)$
6:      end for
7:      $\delta^L = \nabla_a C \odot \sigma'(z^L)$
8:      for $l = L-1, L-2, \dots, 2$ do
9:          $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
10:     end for
11:     $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$
12:     $\frac{\partial C}{\partial b^l_j} = \delta^l_j$
13: until convergence is reached
```

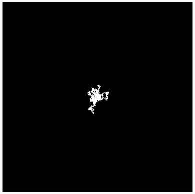
Backpropagation VIII

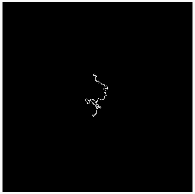
Algorithm 4 Backpropagation Algorithm (batch input)

```
1: repeat
2:    for each sample $x$ do
3:       Set the corresponding activation $a^{x,1}$ for the input layer
4:       for $l = 2, 3, \ldots, L$ do
5:          $z^l = w^l a^{l-1} + b^l$
6:          $a^l = \sigma(z^l)$
7:       end for
8:       $\delta^L = \nabla_a C \odot \sigma'(z^L)$
9:       for $l = L-1, L-2, \ldots, 2$ do
10:         $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
11:      end for
12:   end for
13:   for $l = L, L-1, \ldots, 2$ do
14:      $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
15:      $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$
16:   end for
17:   $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$
18:   $\frac{\partial C}{\partial b^l_j} = \delta^l_j$
19: until convergence is reached
```

Classification I

Let us look at the problem of classifying walks into random walk and self-avoiding walk. Below are two images from a set of generated images.


random walk


Self avoiding walk

Classification: Example

```python
#from __future__ import print_function, division

import os
import numpy as np
import tensorflow as tf
#import matplotlib.pyplot as plt

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from skimage import data as dt
from skimage import transform
from skimage.color import rgb2gray
from scipy import misc
import random


# Larger sample for RW and SAW
# Size dependence of the classification and recognition
# What has the network learned
```

./progs/Walkclassifier.py

Classification: Example

```python

def plot_set(images, set):

    for i in range(len(set)):
        plt.subplot(1, len(set), i+1)
        plt.axis('off')
        plt.imshow(images[set[i]],cmap="gray")
        plt.subplots_adjust(wspace=0.5)

    plt.show()

    pass


def load_data(data_directory,labels,images,L):
    directories = [d for d in os.listdir(data_directory)
                   if os.path.isdir(os.path.join(data_directory,
    d))]

    for d in directories:
```

./progs/Walkclassifier.py

Classification: Example

```
        label_directory = os.path.join(data_directory, d)
        file_names = [os.path.join(label_directory, f)
                      for f in os.listdir(label_directory)
                      if f.endswith(".png")]
        for f in file_names:
            img = dt.imread(f,as_gray=True)
            cropped = img[L/4:3*L/4,L/4:3*L/4]
            images.append(cropped)
            labels.append(int(d))

    print(len(images))
    print(len(labels))
    pass


def show_sample_prediction(sample_images,sample_labels):

  fig = plt.figure(figsize=(10, 10))
  for i in range(len(sample_images)):
    truth = sample_labels[i]
```

./progs/Walkclassifier.py

Classification: Example

```
    prediction = predicted[i]
    plt.subplot(5, 2,1+i)
    plt.axis('off')
    color='green' if truth == prediction else 'red'
    plt.text(40, 10, "Truth:        {0}\nPrediction: {1}".format(
    truth, prediction),
         fontsize=12, color=color)
    plt.imshow(sample_images[i],  cmap="gray")

  plt.show()



tf.set_random_seed(4711)

L = 500
crop_L = L / 2

ROOT_PATH = "/Users/heermann/tensorflow/Prog/Walk/data/"

train_data_directory = os.path.join(ROOT_PATH, "Training")
```

./progs/Walkclassifier.py

Classification: Example

```
test_data_directory = os.path.join(ROOT_PATH, "Testing")

labels = []
images = []

load_data(train_data_directory,labels,images,L)
images = np.array(images)

set = [1, 2,3,4]
plot_set(images,set)


g = tf.Graph()

tf.image.per_image_standardization(images)



# Initialize placeholders
with g.as_default():
```

./progs/Walkclassifier.py

Classification: Example

```
x = tf.placeholder(dtype = tf.float32, shape = [None, crop_L,
  crop_L])
y = tf.placeholder(dtype = tf.int32, shape = [None])

# Flatten the input data
images_flat = tf.contrib.layers.flatten(x)

# Fully connected layer
fully_connected1 = tf.contrib.layers.fully_connected(
 images_flat, 12, tf.nn.relu)
fully_connected2 = tf.contrib.layers.fully_connected(
 fully_connected1, 6, tf.nn.relu)
logits = tf.contrib.layers.fully_connected(fully_connected2,
 12, tf.nn.relu)

# Define a loss function
loss = tf.reduce_mean(tf.nn.
 sparse_softmax_cross_entropy_with_logits(labels = y,

     logits = logits))
# Define an optimizer
train_op = tf.train.AdagradOptimizer(learning_rate=0.001,
 name="Optimizer").minimize(loss)

# Convert logits to label indexes
```

Classification: Example

```

# Define an accuracy metric
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

print("images_flat: ", images_flat)
print("logits: ", logits)
print("loss: ", loss)
print("predicted_labels: ", correct_pred)


# Add ops to save and restore all the variables.
saver = tf.train.Saver()
sess = tf.Session()

sess.run(tf.global_variables_initializer())

for i in range(201):
        print('EPOCH', i)
        _, accuracy_val = sess.run([train_op, accuracy],
 feed_dict={x: images, y: labels})
        if i % 10 == 0:
```

./progs/Walkclassifier.py

Classification: Example

```
                print("Loss: ", loss)
            print('DONE WITH EPOCH')

    save_path = saver.save(sess, "./my-model.ckpt")
    saver.save(sess, './my-model')
    # Display layers
    layers = {v.op.name: v for v in tf.trainable_variables()}
    print(layers)
    #


# Pick 10 random images
sample_indexes = random.sample(range(len(images)), 10)
sample_images = [images[i] for i in sample_indexes]
sample_labels = [labels[i] for i in sample_indexes]

# Run the "correct_pred" operation
predicted = sess.run([correct_pred], feed_dict={x: sample_images
    })[0]

# Print the real and predicted labels
```

./progs/Walkclassifier.py

Classification: Example

```python
print(sample_labels)
print(predicted)

# Display the predictions and the ground truth visually.

show_sample_prediction(sample_images,sample_labels)


# ################################################################################

# Load the test data
test_labels = []
test_images = []

load_data(test_data_directory,test_labels,test_images,L)
test_images = np.array(test_images)
test_set = [4, 5, 6,7]
plot_set(test_images,test_set)

tf.image.per_image_standardization(test_images)
```

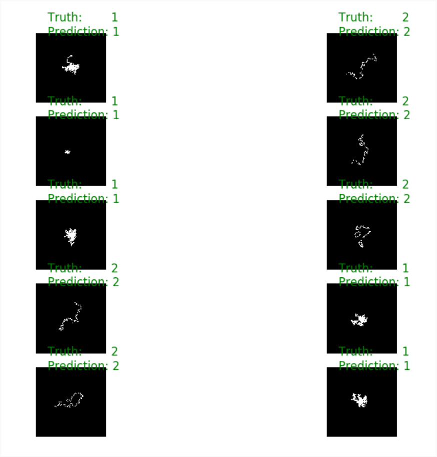
./progs/Walkclassifier.py

Classification: Example

```

# Run predictions against the full test set.
predicted = sess.run([correct_pred], feed_dict={x: test_images})
    [0]

# Calculate correct matches
match_count = sum([int(y == y_) for y, y_ in zip(test_labels,
    predicted)])
print('Match Count = ' + str(match_count) + 'out of =' + str(len(
    test_images)))


sess.close()
```

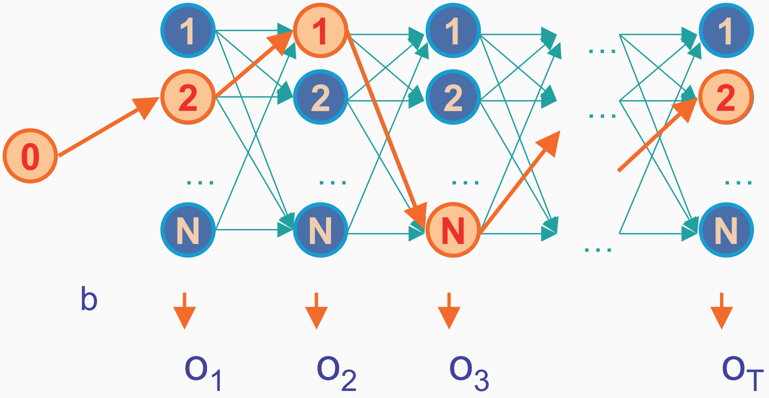
./progs/Walkclassifier.py

Classification: Example I



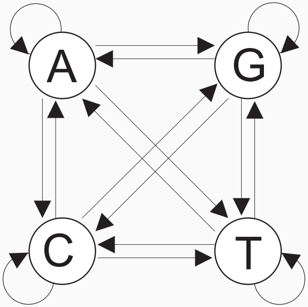
The result of the classifcation after minimal training is already very impressive.

Hidden Markov Model I



Example I

- We would like to identify stretches of sequences that are actually functional (code for proteins or have regulatory functions) from non-coding or junk sequences.
- In prokaryotic DNA we have only two kinds of regions, ignore regulatory sequences which are coding (+) and non-coding (-) and the four letters A,C,G,T.



Hidden Markov Model I

- This simulates a very common phenomenon [14]:

  There is some underlying dynamic system running along according to simple and uncertain dynamics, but we cannot see it.

- All we can see are some noisy signals arising from the underlying system. From those noisy observations we want to do things like predict the most likely underlying system state, or the time history of states, or the likelihood of the next observation

What are Hidden Markov Models good for?

- useful for modeling protein/DNA sequence patterns
- probabilistic state-transition diagrams
- Markov processes - independence from history
- hidden states

Where does one use HMM's I

- protein families
- DNA patterns
- secondary structure (helix, strand, coil (each has 20x20 table with transition frequencies between neighbors $a_i \rightarrow a_{i+1}$))
- protein fold recognition
- fold classification
- gene silencing
- ...

Example I

- **Example: CpG - Islands**

  The CpG sites or CG sites are regions of DNA where a cytosine nucleotide is followed by a guanine nucleotide in the linear sequence of bases along its $5' \rightarrow 3'$ direction. (definition from Wikipedia)

  - Regions labeled as CpG - islands $\longrightarrow +$ model
  - Regions labeled as non-CpG - islands $\longrightarrow$ - model
  - Maximum likelihood estimators for the transition probabilities for each model

$$a_{st} = \frac{c_{st}}{\sum_{t'} c_{st'}}$$

  and analogously for the - model. c_{st} is the number of times letter t followed letter s in the labeled region.

Definition I

- A Hidden Markov Model is a two random variable process, in which one of the random variables is hidden, and the other random variable is observable.
- It has a finite set of states, each of which is associated with a probability distribution.
- Transitions among the states are governed by transition probabilities.
- In a particular state an observation can be generated, according to the associated probability distribution.
- It is only the observation, not the state visible to an external observer, and therefore states are “hidden” from the observer.

Example I

-
    - For DNA, let + denote coding and - non-coding. Then a possible observed sequence could be
    $$O = AACCTTCCGCGCAATATAGGTAACCCCGG$$
    and
    $$Q = - - + + + + + + + + + + + + + + + + + - - - - - - - -$$
- Question: How can one find CpG - islands in a long chain of nucleotides?
- Merge both models into one model with small transition probabilities between the chains.

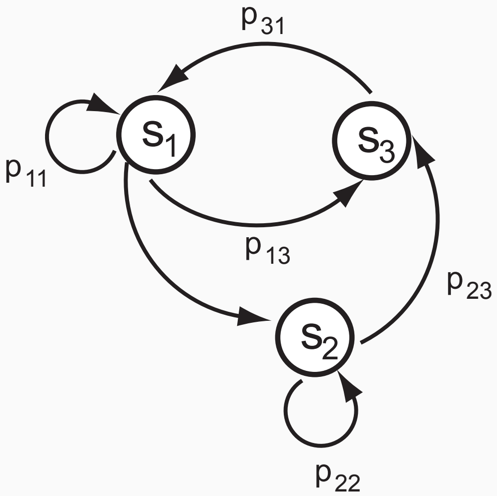
Definition (formal) I

A Hidden Markov Model (HMM) $\lambda =< Y, S, A, B >$ consists of:

- an output alphabet $Y = \{1, ..., b\}$
- a state space $S = \{1, ..., c\}$ with a unique initial state s_0
- a transition probability distribution $A(s'|s)$
- an emission probability distribution $B(y|s, s')$

- HMMs are equivalent to (weighted) finite state automata with outputs on transitions.
- Unlike MMs, constructing HMMs, estimating their parameters and computing probabilities are not so straightforward.

Definition (formal) II



Probabilities I

Given a HMM λ and a state sequence $S = (s_1, ..., s_{t+1})$, the probability of an output sequence $O = (o_1, ..., o_t)$ is

$$P(O|S,\lambda) = \prod_{i=1}^{t} P(o_i|s_i, s_{i+1}, \lambda) = \prod_{i=1}^{t} B(o_i|s_i, s_{i+1}) \; . \tag{37}$$

Given λ, the probability of a state sequence $S = (s_1, ..., s_{t+1})$ is

$$P(S|\lambda) = \prod_{i=1}^{t} P(s_{i+1}|s_i) = \prod_{i=1}^{t} A(s_{i+1}|s_i) \; . \tag{38}$$

Of importance is the probability of an output sequence $O = (o_1, ..., o_t)$ under a given λ. It is easy to show that the straightforward computation yields

$$P(O|\lambda) = \sum_{S} \prod_{i=1}^{t} A(s_{i+1}|s_i) B(o_i|s_i, s_{i+1}) \tag{39}$$

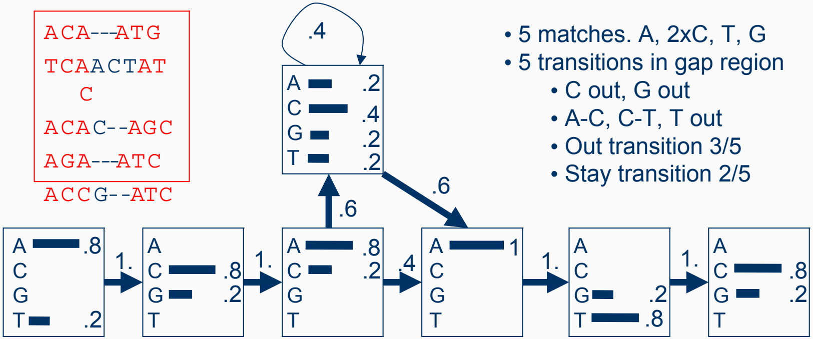
with a computational complexity of $(2c+1) * c^{t+1}$ multiplications.

Example I

Example: Multiple Sequence Alignments

- In theory, making an optimal alignment between two sequences is computationally straightforward (Smith-Waterman algorithm), but aligning a large number of sequences using the same method is almost impossible (e.g. $O(t^N)$).
- The problem increases exponentially with the number of sequences involved (the product of the sequence lengths).
- Statistical Methods:
  - Expectation Maximization Algorithm (deterministic).
  - Gibbs Sampler (stochastic).
  - Hidden Markov Models (stochastic).
- Advantages for HMM: theoretical explanation, no sequence ordering, no insertion and deletion penalties, using prior information.
- Disadvantage for HMM: large number of sequences for training.

Example II



- 5 matches. A, 2xC, T, G
- 5 transitions in gap region
  - C out, G out
  - A-C, C-T, T out
  - Out transition 3/5
  - Stay transition 2/5

ACA---ATG 0.8x1x0.8x1x0.8x0.4x1x0.8x1x0.2 = $3.3x10^{-2}$

Basic Problem I

- There are three basic problems:

  1. Given a model, how likely is a specific sequence of observed values (**evaluation problem**).
  2. Given a model and a sequence of observations, what is the most likely state sequence in the model that produces the observations (**decoding problem**).
  3. Given a model and a set of observations, how should the model's parameters be updated so that it has a high probability of generating the observations (**learning problem)**.

Forward algorithm I

- We define $\alpha_s(i)$ as the probability being in state s at position i:

$$\alpha_s(i) = P(o_1, ..., o_i, s_i = s|\lambda) \ . \tag{40}$$

- Base case: $\alpha_s(1)$ if $s = s_0$ and $\alpha_s(0) = 0$ otherwise
- Induction:

$$\alpha_s(i+1) = \max_{s \in S} A(s|s')B(o_i|s', s)\alpha_s(i) \ . \tag{41}$$

- Finally, at the end:

$$P(o_1, ..., o_k|\lambda) = \sum_{s \in S} \alpha_s(k) \ . \tag{42}$$

- Partial sums could as well be computed right to left (backward algorithm), or from the middle out
- In general, for any position i:

$$P(O|\lambda) = \sum_{s \in S} \alpha_s(i)\beta_s(i) \ . \tag{43}$$

- This algorithm could be used, e.g. to identify which λ is most likely to have produced an output sequence O.

Forward algorithm II

What is the most probable path given observations (decoding problem)?

- Given $o_1, ..., o_t$ what is

$$\text{argmax}_S P(s, o_1, ...o_t|\lambda) \; ? \tag{44}$$

- Slow and stupid answer:

$$\text{argmax}_S \frac{P(o_1, ..., o_t|s)P(s)}{P(o_1, ..., o_t)} \; . \tag{45}$$

Viterbi algorithm I

- We define $\delta_s(i)$ as the probability of the most likely path leading to state s at position i:

$$\delta_s(i) = \max_{s_1,\ldots,s_{i-1}} P(s_1, \ldots, s_{i-1}, o_1, \ldots, o_{i-1}, s_i = s|M)\ . \tag{46}$$

- Base case: $\delta_s(1)$ if $s = s_0$ and $\delta_s(0) = 0$ otherwise

- Again we proceed recursively:

$$\delta_s(i+1) = \max_{s \in S} A(s|s')B(o_i|s', s)\delta_s(i) \tag{47}$$

  and since we want to know the identity of the best state sequence and not just its probability, we also need

$$\Psi(i+1) = \text{argmax}_{s \in S} A(s|s')B(o_i|s', s)\delta_s(i)\ . \tag{48}$$

- Finally, we can follow Ψ backwards from the most likely final state.

- The Viterbi algorithm efficiently searches through $|S|^T$ paths for the one with the highest probability in $O(T|S|^2)$ time.

Viterbi algorithm II

- In practical applications, use log probabilities to avoid underflow errors.
- Can be easily modified to produce the n best paths.
- A beam search can be used to prune the search space further when $|S|$ is very large (n-gram models).

n-gram Models I

- Predicting the next state s_n depending on $s_1, ..., s_{n-1}$ results in

$$P(s_n | s_1, ..., s_{n-1}) \ . \tag{49}$$

- Markov Assumption $(n-1)$th order : last $n-1$ states are in the same equivalence class.

Parameter estimation I

- Given an HMM with a fixed architecture, how do we estimate the probability distributions A and B?
- If we have labeled training data, this is not any harder than estimating non-Hidden Markov Models (supervised training):

$$A(s'|s) = \frac{C(s \to s')}{\sum_{s''} C(s \to s'')} \tag{50}$$

$$B(o|s,s') = \frac{C(s \to s', o)}{C(s \to s')} \tag{51}$$

Forward-Backward Algorithm I

- Also known as the **Baum-Welch algorithm**.
- Instance of the **Expectation Maximization (EM) algorithm**:

  1. Choose a model at random.
  2. E: Find the distribution of state sequences given the model.
  3. M: Find the most likely model given those state sequences.
  4. Go back to 2.

- Our estimate of A is:

$$A(s'|s) = \frac{E[C(s \rightarrow s')]}{E[C(s \rightarrow ?)]} \tag{52}$$

Forward-Backward Algorithm II

- We estimate $E[C(s \to s')]$ via $\tau_t(s, s')$, the probability of moving from state s to state s' at position t given the output sequence O:

$$\begin{aligned}
\tau_t(s, s') &= P(s_t = s, s_{t+1} = s' | O, \lambda) && (53)\\
&= \frac{P(s_t = s, s_{t+1} = s', O | \lambda)}{P(O|\lambda)} && (54)\\
&= \frac{\alpha_s(t) A(s|s') B(o_{t+1}|s, s') \beta_{s'}(t+1)}{\sum_{s''} \alpha_{s''}} . && (55)
\end{aligned}$$

- This lets us estimate A:

$$A(s'|s) = \frac{\sum_t \tau_t(s, s')}{\sum_t \sum_{s''} \tau_t(s, s'')} . \quad (56)$$

- We can estimate B along the same lines, using $\sigma_t(o, s, s')$, the probability of emitting o while moving from state s to state s' at position t given the output sequence O.
- Alternate re-estimating A from τ, then τ from A, until estimates stop changing.
- If the initial guess is close to the right solution, this will converge to an optimal solution.

Reinforcement Learning I

The fundamental idea of reinforcement learning is to interaction with the environment and learn from this interaction. Let $\mathcal{S}$ denote the states that the environment can be in and $\mathcal{A}$ the actions that an **agent** interacting with the environment can take [15–17]. For each interaction the agent gets a **return** or **reward** $r \in \mathcal{R}$. The agent is trained maximizing the cumulative reward. The actions are chosen according to a policy π.

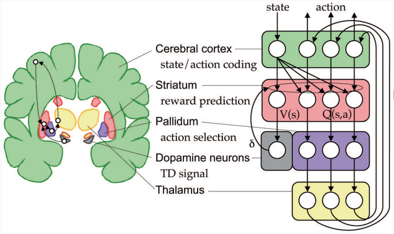
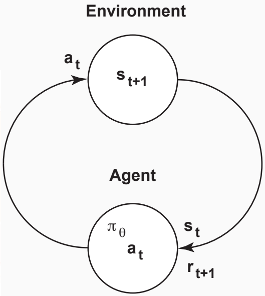


Image taken from: Kenji Doya (DOI: 10.2976/1.2732246)



State cycle for the reinforcement learning

Reinforcement Learning II

Figure 8

Let π be a **policy** mapping a state to an action

$$\pi : \mathcal{S} \to \mathcal{A} \tag{57}$$

$$s \mapsto a \tag{58}$$

The state cycle (c.f. Figure 8) is

$$t : s_t \to a_t \to s_{t+1} \tag{59}$$

$$(s_t, a_t, s_{t+1}) \to r_{t+1} \tag{60}$$

Let τ be a sequence or trajectory under π

$$\tau : (s_1, a_1, r_1, ..., s_T, a_T, r_T) \sim \pi \tag{61}$$

Reinforcement Learning III

where T is the **horizon**. We have $a_i \sim \pi_\theta(a_i|s_i)$ and $s_i \sim P(s_i|s_{i-1}, a_{i-1})$. Since the next state in the trajectory depends only on the immediate predecessor we have

$$P_\theta(s_1, a_1, r_1, ..., s_T, a_T, r_T) = \mu(s_1) \prod_{i=2}^{T} \pi(a_i|s_i) P(s_i|s_{i-1}, a_{i-1}) \tag{62}$$

that the probability P to get the trajectory τ is split into individual transitions and represents the dynamics (Markov chain). μ is the starting state distribution. Such a sequence can be obtained using Monte Carlo methods.

The policy function is usually parameterized with a parameter θ:

$$\pi_\theta(s) = \pi(a \mid s, \theta) = p(a \mid s; \theta) \ . \tag{63}$$

Our objective is to maximize the return

$$\max_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \max_\theta \int \pi_\theta(\tau) R(\tau) d\tau \tag{64}$$

with the return function R that usually is a function of (s_i, a_i, s_{i+1}). Hence we want to find

Reinforcement Learning IV

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta}[\textstyle\sum_t^T r(s_t, a_t)] \quad (65)$$

$$\theta^* = \arg\max_\theta \mathbb{E}_{(s.a)\sim\pi_\theta(s,a)}[r(s, a)] \quad (66)$$

for finite horizon and infinite horizon respectively. Maximizing could be done by taking the derivative with respect to the return. However, the return may not be differentiable. This could be rectified by using a neural network (see later). We will take the approach of the **policy gradient**, i.e., taking the derivative with respect to the policy (parameter θ).

We have assumed that the policy is stochastic, i.e., mapping state s under the condition of parameter value θ to a with probability p. We distinguish between deterministic and stochastic policies:

- deterministic policy: $\pi(a \mid s, \theta) = 1$,
- Stochastic policy: $\pi(a \mid s, \theta) = p(a \mid s; \theta)$.

We have the choice to optimize values or actions:

Reinforcement Learning V

- **Values policy**: Learn the interaction between states, actions and subsequent rewards.
- **Action policy**: Determine which is the best action to choose given the above.

Let $V^\pi(s)$ be the value of state s following policy π (**value-state function**):

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[G_t | S_t = s] \tag{67}$$

where

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{68}$$

is the **cumulative discounted return** with **discount parameter** γ $(0 < \gamma \leq 1)$.

Further, let $Q^\pi(s, a)$ be the **action-value function**

$$Q^\pi(s, a) = \mathbb{E}_{a\sim\pi}[G_t | S_t = s, A_t = a] \tag{69}$$

and

Reinforcement Learning VI

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{70}$$

the **advantage** telling us how much better or worse the action a is. Note that in complex formulae, for clarity, we are dropping the subscript θ.

We can define the **reward function** in terms of the state-value or action-state function as

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s)\sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a) \tag{71}$$

where $d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta)$, i.e., the stationary distribution of the Markov chain of the policy π.

We want to take the gradient of the reward function to maximize the return with respect to the policy parameterized by θ. Before we do this, consider the following:

Reinforcement Learning VII

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{X\sim p(X\mid\theta)}[f(X)] &= \nabla_\theta \left(\int_{\mathcal{X}} f(X)p(X \mid \theta) dX \right) && (72)\\
&= \int_{\mathcal{X}} f(X) \nabla_\theta \left(p(X \mid \theta)\right) dX && (73)\\
&= \int_{\mathcal{X}} f(X)p(X \mid \theta) \frac{\nabla_\theta \left(p(X \mid \theta)\right)}{p(X \mid \theta)} dX && (74)\\
&= \int_{\mathcal{X}} f(X)p(X \mid \theta) \nabla_\theta \left(\log p(X \mid \theta)\right) dX && (75)\\
&= \mathbb{E}_{X\sim p(X\mid\theta)} \left[f(X)\nabla_\theta \left(\log p(X \mid \theta)\right)\right] \,. && (76)
\end{aligned}$$

Hence, for the policy this implies

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \,. \quad (77)$$

We define the score function to be

$$\nabla_\theta \log \pi_\theta(s,a) \,. \quad (78)$$

Reinforcement Learning VIII

It describes how sensitive the stochastic policy π to is to changes in θ, i.e. how likely the trajectory is under the current policy.

Example: Policy function

Let us look at the following example of the policy function for a linear model for the unnormalized log-probability: $\phi(s,a)^T\theta$ i.e. weighting of the actions using a linear combination of features $\phi(s,a)$

The score function for a softmax policy is:

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^T\theta}}{\sum_{a'\in\mathcal{A}} e^{\phi(s,a')^T\theta}} \tag{79}$$

$$\nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)] \tag{80}$$

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}\ . \tag{81}$$

Reinforcement Learning IX

Stochastic Gradient Policy Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right] . \tag{82}$$

Hence, the computation of the policy gradient reduces to a simple expectation. Thus, we are looking for sampling algorithms were trajectories are generated, the action value or value state function being evaluated along the trajectory (sometimes called **play out** or **episode**) and the gradient of the log of the policy computed.

To show the above statement, we take the following steps (see Lilian Weng https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html):

Reinforcement Learning X

$$\nabla_\theta V^\pi(s) = \nabla_\theta \Big[\sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \Big] \tag{83}$$

$$= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta Q^\pi(s,a) \Big] \tag{84}$$

$$= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} P(s',r|s,a)(r + V^\pi(s')) \Big] \tag{85}$$

$$= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a) \nabla_\theta V^\pi(s') \Big] \tag{86}$$

$$= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \Big] . \tag{87}$$

Reinforcement Learning XI

$$\nabla_\theta V^\pi(s) = \phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a)\nabla_\theta V^\pi(s') \tag{88}$$

$$= \phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a)\nabla_\theta V^\pi(s') \tag{89}$$

$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\nabla_\theta V^\pi(s') \tag{90}$$

$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\nabla_\theta V^\pi(s') \tag{91}$$

$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)[\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1)\nabla_\theta V^\pi(s'')] \tag{92}$$

$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2)\nabla_\theta V^\pi(s'') \tag{93}$$

$$= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2)\phi(s'') + \ldots \tag{94}$$

$$= \ldots; \text{ Repeatedly unrolling the part of } \nabla_\theta V^\pi(.) \tag{95}$$

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to x, k)\phi(x) \; . \tag{96}$$

Reinforcement Learning XII

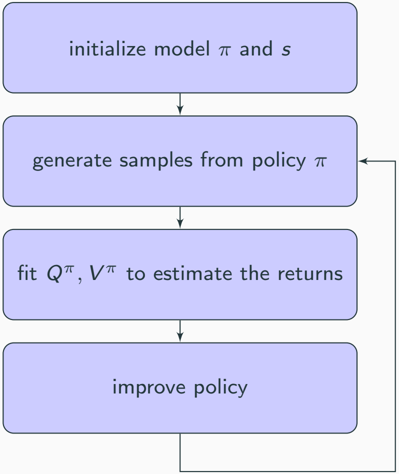
$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^\pi(s_0) && (97)\\
&= \sum_s \sum_{k=0}^{\infty} \rho^\pi(s_0 \to s, k)\phi(s) && (98)\\
&= \sum_s \eta(s)\phi(s) && (99)\\
&= \left(\sum_s \eta(s)\right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && (100)\\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && (101)\\
&= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)\ . && (102)
\end{aligned}$$

$\sum_s \eta(s)$ is the average length of the episode in the continuous case. And further

Reinforcement Learning XIII

$$
\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) && (103)\\
&= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} && (104)\\
&= \mathbb{E}_\pi\left[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)\right] . && (105)
\end{aligned}
$$

Reinforcement Learning XIV


initialize model π and s
generate samples from policy π
fit Q^π, V^π to estimate the returns
improve policy

Actually, the question is how to compute the score function

$$\nabla_\theta \log \pi_\theta(s, a) \tag{106}$$

Reinforcement Learning XV

specifically under light that gradients can be very noisy. They suffer from high variance and low convergence. We have

$$\nabla_\theta \log P_\theta(\tau) = \nabla \log \left(P(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t) \right) \tag{107}$$

$$= \nabla_\theta \left[\log \mu(s_1) + \sum_{t=1}^{T} (\log \pi_\theta(a_t|s_t) + \log P(s_{t+1}|s_t, a_t)) \right] \tag{108}$$

$$= \nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \; . \tag{109}$$

Using gradient ascend

$$\theta \leftarrow \theta + \alpha \nabla f(x) \tag{110}$$

we can write the generic algorithm is as follows:

Reinforcement Learning XVI

Algorithm 5 Gradient Policy

```
1: repeat
2:     $\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(s_{i,t}, a_{i,t}))(\sum_{t=1}^{T} R(s_{i,t}, a_{i,t}))$
3:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
4: until finished
```

α is the **learning rate** determining the rate of convergence.

Monte-Carlo Policy Evaluation I

The **Monte Carlo policy gradient** or **REINFORCE** estimates (learns) the value state function V^π from episodes under policy π. Hence, we generate episodes

$$s_1, a_1, \dots s_T, a_T \sim \pi \; . \tag{111}$$

The Monte-Carlo policy evaluation uses empirical mean return instead of expected return. Note that for the cumulated discounted return we have

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi[G_t | a_t, a_t] \tag{112}$$

and hence we can write

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^\pi(s, a) \nabla_\theta \ln \pi_\theta(a|s)] \qquad (113) \\ &= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(a_t|s_t)] \qquad (114) \end{aligned}$$

and sample the return. Our goal is find the policy, i.e. the value of θ maximizing the return

Monte-Carlo Policy Evaluation II

$$\theta^* = \arg\max_{\theta} \ \mathbb{E}_\pi \left[\sum_{t=1}^{T} \gamma^t r_t \right] \ . \tag{115}$$

Monte-Carlo Policy Evaluation: Algorithm I

A generic version is listed in Algorithm 6.

Algorithm 6 Generic Monte-Carlo Policy Evaluation: REINFORCE

```
1: Initialize the policy parameter $\theta$
2: repeat
3:     Generate episode using $\pi_\theta \sim (s_1, a_1, ..., a_T, s_T)$
4:     for $t$ in range $(1, T)$ do
5:         Evaluate $G_t$
6:         $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(a_t|s_t)$
7:     end for
8: until false
```

Monte Carlo Policy Gradient: Example I

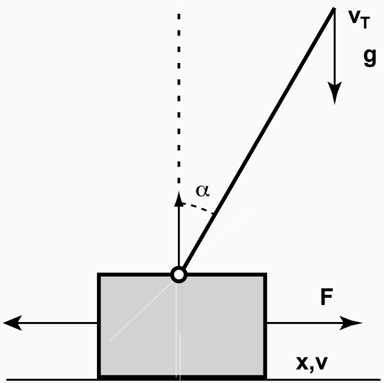


Figure 9: Balancing of a stick

Monte Carlo Policy Gradient: Example II

- The task at hand is to balance a stick in a one-dimensional setting. The stick is mounted frictionless on a rail and can be moved to left and to the right. The stick can rotate and is subject to gravitation (for info on the gym environment implementing the balance stick see https://github.com/openai/gym/wiki/CartPole-v0).
- Neglecting friction, the equations of motion are [18]:

$$\ddot{x} = .\frac{F + m_p l \left(\alpha^2 \sin\alpha - \ddot{\alpha}\cos\alpha\right)}{m_c + m_p} \tag{116}$$

$$\ddot{\alpha} = \frac{g\sin\alpha + \cos\alpha\left(.\frac{-F - m p_l \dot{\alpha}^2 \sin\alpha}{m_c + m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p \cos\alpha^2}{m_c + m_p}\right)}\ . \tag{117}$$

- In this example we assume that there are two action $a = 0, 1$ or $a = -1, +1$ corresponding to **left** and **right**. The state of the system is given by a state vector with the components $s =$(position (x), velocity (v), stick angle (α), velocity at tip (v_T)).

Monte Carlo Policy Gradient: Example III

Table 1: openai CartPole v0 states (https://openai.com/resources/)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	$\sim -41.8°$	$\sim 41.8°$
3	Pole Velocity At Tip	-Inf	Inf

- Since our action is binary, we can choose the logistic function as part of the policy π

$$\mathrm{L}(x) = \frac{1}{1+e^{-x}} \ . \tag{118}$$

- We can define the policy π as

$$\pi_\theta(s, a=0) = 1 - \mathrm{L}(s^T\theta) \tag{119}$$

$$\pi_\theta(s, a=1) = \mathrm{L}(s^T\theta) \ . \tag{120}$$

Monte Carlo Policy Gradient: Example IV

- Our task is to estimate the state action function

$$Q^{\pi}(s, a) \tag{121}$$

from the discounted return function

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t} r_T \tag{122}$$

where the reward is 1 for every step taken, including the termination step.

$$J(\theta) \approx \sum_{t=1}^{T} \pi(a_t \mid s_t, \theta) A_t \ . \tag{123}$$

- For one episode we have

$$\nabla_\theta J(\theta) \approx \sum_{t=1}^{T} G_t \nabla_\theta \log \pi_\theta(s, a) \tag{124}$$

$$\frac{d}{dx}\mathrm{sigmoid}(x) = \mathrm{sigmoid}(x)(1 - \mathrm{sigmoid}(x)) \ . \tag{125}$$

- The problem is considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.

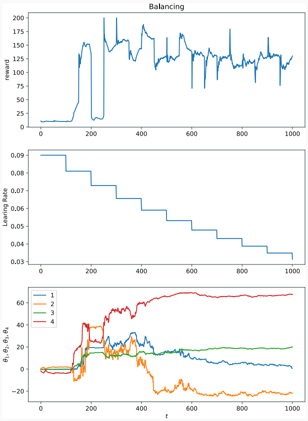
Monte Carlo Policy Gradient: Example

```
def episode(theta, max_episode_length=1000):

    observation = env.reset()

    actions     = []
    states      = []
    rewards     = []
    done        = False

    i = 0

    while not done:

        i+=1

        action = get_action(theta, observation)
        states.append(observation)
        actions.append(action)
        observation, reward, done, info = env.step(action)
        rewards.append(reward)

        if i > max_episode_length:
           break

    return np.array(rewards), np.array(states), np.array(actions)
```

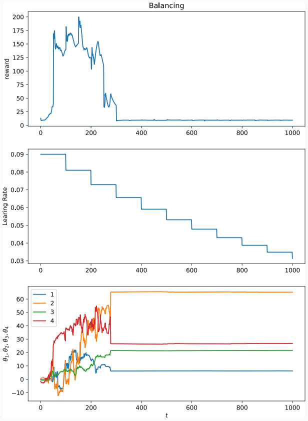
Monte Carlo Policy Gradient: Example

```python

def discounted_sum_of_rewards(rewards, gamma):

    cum = np.zeros_like(rewards)
    c     = 0.0
    for i, r in enumerate(rewards[::-1]):
        c         = r + gamma * c
        cum[i] = c
    return cum[::-1]
```

Monte Carlo Policy Gradient: Example I



(a) Example 1



(b) Example 2

Figure 10: Two examples of balancing of a stick using Monte Carlo policy gradient reinforcement learning. Max play out length was 1000 and 1000 episodes were calculated.

Monte-Carlo Policy Evaluation: Algorithm I

Algorithm 7 Generic Monte-Carlo Policy Evaluation

```
1: Given $\pi$ the policy to be evaluated
2: Initialize $V$ randomly
3: Returns(s) $\leftarrow$ empty list for all $s \in S$
4: repeat
5:     Generate episode using $\pi$
6:     for $s$ in trial do
7:         $R \leftarrow$ return following the first occurrence of $s$
8:         Append $R$ to Returns(s)
9:         $V(s) \leftarrow$ average(Returns(s))
10:    end for
11: until false
```

Bellman Equation:

$$V^{\pi}(s) = \sum_{a} \pi(a \mid s) \left(R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a V^{\pi}(s') \right) . \tag{126}$$

Monte-Carlo Policy Evaluation: Neural Network Policy I

- This works well because the output is a probability over available actions.
- If we feed it with a neural network, we will get higher values and thus we will be more likely to choose the actions that we learned produce a better reward.
- In the long-run, this will trend towards a deterministic policy, $\pi(a \mid s, \theta) = 1$, but it will continue to explore as long as one of the probabilities does not dominate the others (which will likely take some time).

For the algorithm we are going to assume

- a differentiable policy parameterization $\pi(a \mid s, \theta)$
- and define the step-size $\alpha > 0$.

Monte-Carlo Policy Evaluation: Neural Network Policy II

Algorithm 8 Generic Monte-Carlo Policy Evaluation Neural Network

```
1:  Initialize policy parameters $\theta$
2:  repeat
3:      Generate episode using $\pi$
4:      for $N$ batches do
5:          Generate an episode $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$, following $\pi(a \mid s, \theta)$
6:          for $t = 0, ..., T-1$ do
7:              $G_t \leftarrow$ from step $t$
8:          end for
9:          Calculate the loss $L(\theta) = -\frac{1}{N} \sum_t^T ln(\gamma^t G_t \pi(a_t \mid s_t, \theta))$
10:         Update policy parameters through backpropagation: $\theta := \theta + \alpha \nabla_\theta L(\theta)$
11:     end for
12: until $n$ episodes
```

We are going to apply the neural network approach to the balancing of a stick problem defined above. We will be using a fully connected neural network as shown in Figure 11. The layer size is halved from one layer to the next. The last layer essentially represents a binary decision to move left or right.

Monte-Carlo Policy Evaluation: Neural Network Policy III

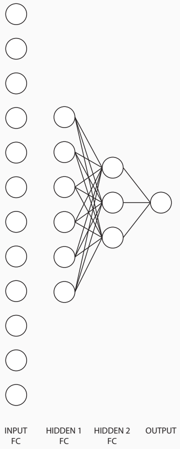


Figure 11: Principle design of the neural network to be used in the learning of the policy gradient in our example. All layers are fully connected. Only the last layers in this graph show the actual connectivity.

Monte Carlo Policy Gradient: Neural Network

```python
import gym
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected
import warnings

class policy_estimator(object):

    def __init__(self, sess, env):
        # Pass TensorFlow session object
        self.sess = sess
        # Get number of inputs and outputs from environment
        self.n_inputs = env.observation_space.shape[0]
        self.n_outputs = env.action_space.n
        self.learning_rate = 0.01

        # Define number of hidden nodes
        self.n_hidden_nodes = 256

        # Set graph scope name
        self.scope = "policy_estimator"
```

Monte Carlo Policy Gradient: Neural Network

```
        # Create network
        with tf.variable_scope(self.scope):
            initializer = tf.contrib.layers.xavier_initializer()

            # Define placholder tensors for state, actions,
            # and rewards
            self.state   = tf.placeholder(tf.float32,
                    [None, self.n_inputs], name='state')
            self.rewards = tf.placeholder(tf.float32,
                    [None], name='rewards')
            self.actions = tf.placeholder(tf.int32,
                    [None], name='actions')

            layer_1 = fully_connected(self.state,
                    self.n_hidden_nodes,
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            layer_2 = fully_connected(layer_1,
                    int(self.n_hidden_nodes/2),
```

Monte Carlo Policy Gradient: Neural Network

```
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            layer_3 = fully_connected(layer_2,
                    int(self.n_hidden_nodes/4),
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            layer_4 = fully_connected(layer_3,
                    int(self.n_hidden_nodes/8),
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            layer_5 = fully_connected(layer_4,
                    int(self.n_hidden_nodes/16),
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            layer_6 = fully_connected(layer_5,
                    int(self.n_hidden_nodes/32),
                    activation_fn=tf.nn.swish,
                    weights_initializer=initializer)
            output_layer = fully_connected(layer_6,
                    self.n_outputs,
```

Monte Carlo Policy Gradient: Neural Network

```
                activation_fn=None,
                weights_initializer=initializer)

        # Get probability of each action
        self.action_probs = tf.squeeze(
            tf.nn.softmax(output_layer -
                tf.reduce_max(output_layer)))

        # Get indices of actions
        indices = tf.range(0, tf.shape(output_layer)[0]) \
            * tf.shape(output_layer)[1] + self.actions

        selected_action_prob = tf.gather(
            tf.reshape(self.action_probs, [-1]),indices)

        # Define loss function
        self.loss = -tf.reduce_mean(
          tf.log(selected_action_prob) * self.rewards)

        # Get gradients and variables
```

Monte Carlo Policy Gradient: Neural Network

```
            self.tvars = tf.trainable_variables(self.scope)
            self.gradient_holder = []
            for j, var in enumerate(self.tvars):
                self.gradient_holder.append(
                    tf.placeholder(tf.float32,
                        name='grads' + str(j)))

            self.gradients = tf.gradients(self.loss,
                                self.tvars)

            # Minimize training error
            self.optimizer = tf.train.AdamOptimizer(
                    self.learning_rate)
            self.train_op  = self.optimizer.apply_gradients(
                    zip(self.gradient_holder, self.tvars))


    def predict(self, state):
        probs = self.sess.run([self.action_probs],
                    feed_dict={self.state: state})[0]
```

Monte Carlo Policy Gradient: Neural Network

```
        return probs


    def update(self, gradient_buffer):
        feed = dict(zip(self.gradient_holder, gradient_buffer))
        self.sess.run([self.train_op], feed_dict=feed)


    def get_vars(self):
        net_vars = self.sess.run(
                    tf.trainable_variables(self.scope))
        return net_vars


    def get_grads(self, states, actions, rewards):
        grads = self.sess.run([self.gradients],
            feed_dict={
            self.state: states,
            self.actions: actions,
            self.rewards: rewards
```

Monte Carlo Policy Gradient: Neural Network

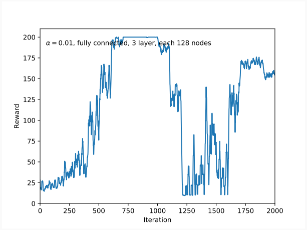
```
            })[0]
        return grads
```

Stability I

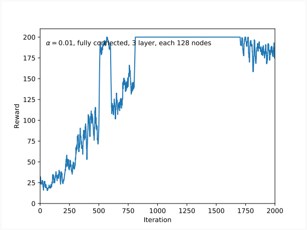
Stability:

> *Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values and the target values [19].*

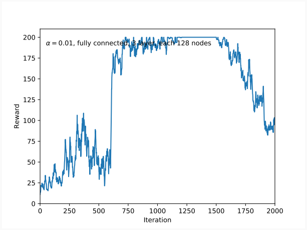
Stability II



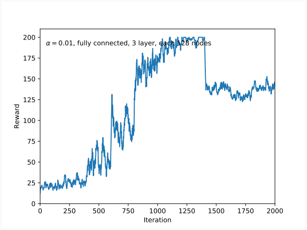
(a) first



(b) second



(c) third



(d) forth

Stability III

Figure 12: Demonstration of the variance involved in the REINFORCE algorithm. Here results from the application of neural network learning of the policy is shown. Shown are results for a neural network where the first layer consists of 128 fully connected nodes as shown schematically in Figure 11. The episode length was a maximum of 100.

Stability IV

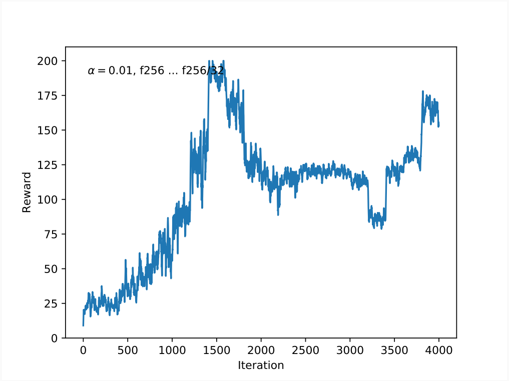


Figure 13: Shown are results for a neural network where the first layer consists of 256 fully connected nodes as shown schematically in Figure 11. The sample used an episode length of a maximum of 200.

REINFORCE Algorithm with Baseline I

To reduce the variance, the standard is to introduce a function $b(s_t)$ inside the expectation on which we are computing the gradient. b is supposed to be an expected return. Let $R(\tau) = \sum_{t=0}^{T-1} r_t$ where we have set the discount parameter equal to one. We can write

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\Big[R(\tau)\Big] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\left(\sum_{t=0}^{T-1} r_t\right)\cdot\nabla_\theta\left(\sum_{t=0}^{T-1}\log\pi_\theta(a_t|s_t)\right)\right] \tag{127}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{T-1} r_{t'}\sum_{t=0}^{t'}\nabla_\theta\log\pi_\theta(a_t|s_t)\right] \tag{128}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r_{t'}\right)\right]. \tag{129}$$

With this we can introduce the baseline function b

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)\right]. \tag{130}$$

REINFORCE Algorithm with Baseline II

If γ is not one than

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)\right] \tag{131}$$

$$\approx \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} - b(s_t)\right)\right] \tag{132}$$

with the baseline

$$b(s_t) \approx \mathbb{E}[r_t + \gamma r_{t+1} + \cdots + \gamma^{T-1-t} r_{T-1}]\,. \tag{133}$$

The REINFORCE Algorithm with baseline is shown in Algorithm 9. Let

$$\theta_p := \theta_p + \alpha_p \gamma^t \delta \nabla_{\theta p} ln(\pi(a_t \mid s_t, \theta_p) \tag{134}$$

where δ is the difference between the actual value and the predicted value at that given state:

REINFORCE Algorithm with Baseline III

$$\delta = G_t - v(S_t, \theta_v) \ . \tag{135}$$

Note that the subscripts p and v to differentiate between the policy estimation function and the value estimation function. Thus, we assume a differentiable policy parameterization $\pi(a \mid s, \theta_p)$ and a differentiable policy parameterization $v(s, \theta_v)$

REINFORCE Algorithm with Baseline IV

Algorithm 9 REINFORCE with baseline: Monte-Carlo policy gradient

```
1:  Define step-size $\alpha_p > 0$, $\alpha_v > 0$
2:  Initialize policy parameters $\theta_p$, $\theta_v$
3:  repeat
4:      for $N$ batches do
5:          Generate an episode $s_0, a_0, r_1, ..., , s_{T-1}, a_{T-1}, r_T$, following $\pi(a \mid s, \theta_p)$
6:          for  $t = 0, ..., T-1$ do
7:              $G_t \leftarrow$ from step $t$
8:          end for
9:          $\delta \leftarrow G_t - v(s, \theta_v)$
10:         Calculate the loss $L(\theta_v) = \frac{1}{N}\sum_t^T(\gamma^t G_t - v(s_t, \theta_v))^2$
11:         Calculate the loss $L(\theta_p) = -\frac{1}{N}\sum_t^T ln(\gamma^t \delta \pi(a_t \mid S_t, \theta_p))$
12:         Update policy parameters through backpropagation: $\theta_p := \theta_p + \alpha_p \nabla_\theta^p L(\theta_p)$
13:         Update policy parameters through backpropagation: $\theta_v := \theta_v + \alpha_v \nabla_\theta^v L(\theta_v)$
14:     end for
15: until $n$ episodes
```

Advantage Function I

Recall the definition of the action state function

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} r_t \,\middle|\, s_0 = s, a_0 = a\right] \tag{136}$$

and the value-state function

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} r_t \,\middle|\, s_0 = s\right] \tag{137}$$

and the advantage function

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)\,. \tag{138}$$

We have [20]

Advantage Function II

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)\right] \tag{139}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \left(Q^\pi(s_t, a_t) - V^\pi(s_t)\right)\right] \tag{140}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A^\pi(s_t, a_t)\right] \tag{141}$$

$$\approx \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A^{\pi,\gamma}(s_t, a_t)\right] . \tag{142}$$

Q-Learning I

- Deep Q network and the epsilon-greedy policy.
- Q learning is a value based method of supplying information to inform which action an agent should take.
- In tabular Q-learning, for example, you are selecting the action that gives the highest expected reward ($max_a' Q(s', a')$), possibly also in an ϵ-greedy fashion) which means if the values change slightly, the actions and trajectories may change radically.

The Q learning rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s,a)] \tag{143}$$
$$0 \leq \gamma \geq 1 \tag{144}$$
$$\alpha \tag{145}$$

with α being the learning rate.

Both α and the $Q(s, a)$ subtraction are not required to be explicitly defined in deep Q learning, as the neural network will take care of that during its optimized learning

Q-Learning II

process, i.e., deep Q-learning applies the Q-learning updating rule during the training process. A neural network is created which takes the state s as its input, and then the network is trained to output appropriate $Q(s, a)$ values for each action in state s.

Actor-Critic policy gradient algorithm I

Actor-critic methods consist of two models, which may optionally share parameters:

- Critic updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- Actor updates the policy parameters θ for $\pi_\theta(a|s)$ in the direction suggested by the critic.

Let α_θ and α_w be two learning rates. predefined for policy and value function parameter updates respectively. The actor-critic Monte-Carlo policy gradient algorithm is shown in Algorithm 10.

Actor-Critic policy gradient algorithm II

Algorithm 10 Actor-Critic: Monte-Carlo policy gradient

```
1: Initialize $s, \theta, w$ at random; sample $a \sim \pi_\theta(a|s)$
2: for $t$ (1,...,T) do
3:     Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
4:     Then sample the next action $a' \sim \pi_\theta(a'|s')$
5:     Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$
6:     Compute the correction (TD error) for action-value at time t:
           - $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
           - $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
7:     Update $a \leftarrow a'$ and $a \leftarrow s'$
8: end for
```

ϵ Greedy Strategy I

Let r be a uniform random number. ϵ-greedy strategy

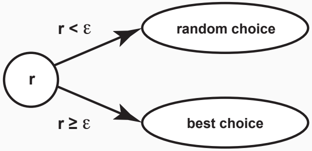


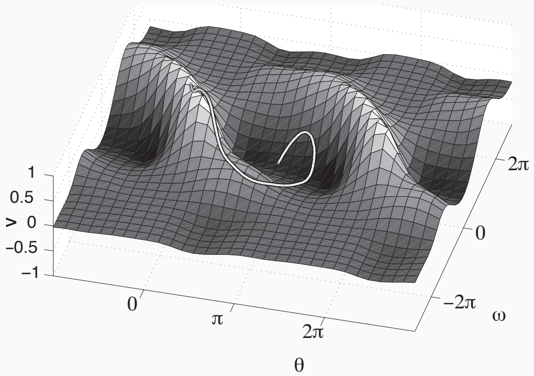
Figure 14: ϵ-greedy strategy. r is a uniform random number.

ϵ Greedy Strategy II

Algorithm 11 ϵ-greedy strategy

```
1: for $i$ (1,...,samples) do
2:     $r \sim$ uniform(0,1)
3:     if $r < \epsilon$ then
4:         choose random action
5:     else
6:         choose best action
7:     end if
8: end for
```

Value Function Fitting



Double pendulum

Value Function Fitting: Example

```python
'''

Source:
https://adventuresinmachinelearning.com/reinforcement-learning-
    tensorflow/

'''

import gym

import tensorflow as tf
import numpy as np
import time
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.stats import norm
import random as random
import math
```

Value Function Fitting: Example

```python
BATCH_SIZE = 100
MAX_EPSILON = 1.0
MIN_EPSILON = 0.0
LAMBDA = 0.001
GAMMA = 0.99


class Model:


    def __init__(self, num_states, num_actions, batch_size):

        self._num_states = num_states
        self._num_actions = num_actions
        self._batch_size = batch_size

        # define the placeholders
        self._states = None
        self._actions = None
```

Value Function Fitting: Example

```python
        # the output operations
        self._logits = None
        self._optimizer = None
        self._var_init = None

        # now setup the model
        self._define_model()


    def _define_model(self):

        self._states = tf.placeholder(shape=[None, self._num_states], dtype=tf.float32)
        self._q_s_a  = tf.placeholder(shape=[None, self._num_actions], dtype=tf.float32)

        # create a couple of fully connected hidden layers
        fc1          = tf.layers.dense(self._states, 50, activation=tf.nn.relu)
        fc2          = tf.layers.dense(fc1, 50, activation=tf.nn.relu)
        self._logits = tf.layers.dense(fc2, self._num_actions)

        loss            = tf.losses.mean_squared_error(self._q_s_a, self._logits)
```

Value Function Fitting: Example

```python
        self._optimizer = tf.train.AdamOptimizer().minimize(loss)
        self._var_init  = tf.global_variables_initializer()


    def predict_one(self, state, sess):
        return sess.run(self._logits, feed_dict={self._states:
                                                     state.reshape
    (1, self._num_states)})


    def predict_batch(self, states, sess):
        return sess.run(self._logits, feed_dict={self._states:
    states})


    def train_batch(self, sess, x_batch, y_batch):
        sess.run(self._optimizer, feed_dict={self._states: x_batch,
     self._q_s_a: y_batch})


class Memory:

    def __init__(self, max_memory):
```

Value Function Fitting: Example

```python
        self._max_memory = max_memory
        self._samples = []

    def add_sample(self, sample):

        self._samples.append(sample)
        if len(self._samples) > self._max_memory:
            self._samples.pop(0)

    def sample(self, no_samples):

        if no_samples > len(self._samples):
            return random.sample(self._samples, len(self._samples))
        else:
            return random.sample(self._samples, no_samples)


class GameRunner:
```

Value Function Fitting: Example

```
    def __init__(self, sess, model, env, memory, max_eps, min_eps,
     decay, render=True):

        self._sess = sess
        self._env = env
        self._model = model
        self._memory = memory
        self._render = render
        self._max_eps = max_eps
        self._min_eps = min_eps
        self._decay = decay
        self._eps = self._max_eps
        self._steps = 0
        self._reward_store = []
        self._max_x_store = []


    def run(self):

        state = self._env.reset()
        tot_reward = 0
```

Value Function Fitting: Example

```
        max_x = -100

        while True:
            if self._render:
                self._env.render()

            action = self._choose_action(state)
            next_state, reward, done, info = self._env.step(action)
            if next_state[0] >= 0.1:
                reward += 10
            elif next_state[0] >= 0.25:
                reward += 20
            elif next_state[0] >= 0.5:
                reward += 100

            if next_state[0] > max_x:
                max_x = next_state[0]

            # is the game complete? If so, set the next state to
            # None for storage sake
```

Value Function Fitting: Example

```python
        if done:
            next_state = None

        self._memory.add_sample((state, action, reward, next_state))
        self._replay()

        # exponentially decay the eps value
        self._steps += 1
        self._eps = MIN_EPSILON + (MAX_EPSILON - MIN_EPSILON) \
                                  * math.exp(-LAMBDA * self._steps)

        # move the agent to the next state and accumulate the reward
        state = next_state
        tot_reward += reward

        # if the game is done, break the loop
        if done:
            self._reward_store.append(tot_reward)
            self._max_x_store.append(max_x)
            break
```

Value Function Fitting: Example

```
        print("Step {}, Total reward: {}, Eps: {}".format(self.
    _steps, tot_reward, self._eps))

    def _choose_action(self, state):

        if random.random() < self._eps:
            return random.randint(0, self._model._num_actions - 1)
        else:
            return np.argmax(self._model.predict_one(state, self.
    _sess))

    def _replay(self):

        batch       = self._memory.sample(self._model._batch_size)
        states      = np.array([val[0] for val in batch])
        next_states = np.array([(np.zeros(self._model._num_states)
                                 if val[3] is None else val[3]) for
     val in batch])

        # predict Q(s,a) given the batch of states
```

Value Function Fitting: Example

```
        q_s_a = self._model.predict_batch(states, self._sess)

        # predict Q(s',a') - so that we can do gamma * max(Q(s'a'))
 below
        q_s_a_d = self._model.predict_batch(next_states, self._sess
)

        # setup training arrays
        x = np.zeros((len(batch), self._model._num_states))
        y = np.zeros((len(batch), self._model._num_actions))
        for i, b in enumerate(batch):

            state, action, reward, next_state = b[0], b[1], b[2], b
[3]

            # get the current q values for all actions in state
            current_q = q_s_a[i]

            # update the q value for action
            if next_state is None:
                # in this case, the game completed after action, so
 there is no max Q(s',a')
                # prediction possible
                current_q[action] = reward
```

Value Function Fitting: Example

```python
            else:
                current_q[action] = reward + GAMMA * np.amax(q_s_a_d[i])
            x[i] = state
            y[i] = current_q

        self._model.train_batch(self._sess, x, y)



if __name__ == "__main__":

    env_name = 'MountainCar-v0'
    env_name = 'Acrobot-v1'
    env = gym.make(env_name)

    num_states = env.env.observation_space.shape[0]
    num_actions = env.env.action_space.n

    model = Model(num_states, num_actions, BATCH_SIZE)
    mem = Memory(50000)
```

Value Function Fitting: Example

```
    with tf.Session() as sess:
        sess.run(model._var_init)
        gr = GameRunner(sess, model, env, mem, MAX_EPSILON,
 MIN_EPSILON, LAMBDA)
        num_episodes = 300
        cnt = 0
        while cnt < num_episodes:
            if cnt % 10 == 0:
                print('Episode {} of {}'.format(cnt+1, num_episodes
 ))
            gr.run()
            cnt += 1

        plt.plot(gr._reward_store)
        plt.show()
        plt.close("all")
        plt.plot(gr._max_x_store)
        plt.show()
```

Continuous State and Action Space I

We will rely on the Stochastic Policy Gradient Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \,. \tag{146}$$

Hence, the computation of the policy gradient reduces to a simple expectation.

Policy modeling: parameterized by a function θ: $\pi_\theta(a|s)$

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \tag{147}$$

where $d^\pi(s)$ is the stationary distribution of Markov chain for π_θ for which

$$d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta) \tag{148}$$

and this is the probability that $s_t = s$ when starting from s_0 and following policy π_θ for t steps.

Problems:

Continuous State and Action Space II

- in generalized policy iteration, the policy improvement step $\arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a)$ requires a full scan of the action space, suffering from the **curse of dimensionality**

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \pi_\theta(a|s) && (149) \\ &\propto \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \nabla_\theta \pi_\theta(a|s) && (150) \end{aligned}$$

Continuous State and Action Space III

$$
\begin{aligned}
\nabla_\theta V^\pi(s) &= \phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) [\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1) \nabla_\theta V^\pi(s'')] \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \nabla_\theta V^\pi(s'') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \phi(s'') + \sum_{s'''} \rho^\pi(s \to s''', 3 \\
&= \ldots; \text{ Repeatedly unrolling the part of } \nabla_\theta V^\pi(.) \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to x, k) \phi(x) \, .
\end{aligned}
$$

Continuous State and Action Space IV

The nice rewriting above allows us to exclude the derivative of Q-value function $\nabla_\theta Q^\pi(s,a)$.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^\pi(s_0) && (160)\\
&= \sum_s \sum_{k=0}^{\infty} \rho^\pi(s_0 \to s, k)\phi(s) && (161)\\
&= \sum_s \eta(s)\phi(s) && (162)\\
&= \left(\sum_s \eta(s)\right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && (163)\\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && (164)\\
&= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)\ . && (165)
\end{aligned}
$$

In the episodic case, the constant of proportionality $(\sum_s \eta(s))$ is the average length of an episode.

Continuous State and Action Space V

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \textstyle\sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) && (166)\\
&= \textstyle\sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} && (167)\\
&= \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] && (168)
\end{aligned}$$

where $\mathbb{E}_\pi$ refrers to $\mathbb{E}_{s\sim d_\pi, a\sim\pi_\theta}$ when both state and action distributions follow the policy π_θ (on policy).

The policy gradient theorem lays the theoretical foundation for various policy gradient algorithms. This vanilla policy gradient update has no bias but high variance. Many following algorithms were proposed to reduce the variance while keeping the bias unchanged.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \;. \tag{169}$$

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Genetic Algorithms

Genetic Algorithms I

Genetic algorithms apply the principles derived from Darwin's principles (natural selection):

- Individuals in population compete for resources.
- Fittest individuals mate to create more offsprings than others.
- Fittest parent propagates genes through generation; parents may produce offsprings better than either parent.
- Generation are coupled to the environment.

The objective is to maintains the population of n individuals along with their fitness scores. Hence, central to genetic algorithms are the notions of population and the fitness function. Each iteration generates from an initial population a new one. There are three operators operating on the individuals from the population:

- **Selection**
  Individuals with better fitness scores pass genes on to successive generations.
- **Crossover**
  The selection operator is applied to select two individuals, and randomly choose crossover sites to exchange the genes at these sites.

Genetic Algorithms II

- **Mutation**
  Insert random genes in offsprings to maintain diversity.

The operation is assumed to be applied to **genes**, often represented by a sequence from an alphabet Σ:

$$ADEAGEF$$

Algorithm 12 Generic Genetic Algorithm

```
1: Generate the initial population
2: Compute fitness
3: repeat
4:     Selection
5:     Crossover
6:     Mutation
7:     Compute fitness
8: until population has converged
```

Genetic Algorithms III

Let us look at the survival probability of individual i wit fitness f_i. One possibility is

$$P_i = \frac{f_i}{\sum_i f_i} \ . \tag{170}$$

Genetic algorithms have the following advantages:

- No gradients are required
- Can be parallelized
- Can optimize continuous as well as discrete functions
- Can be applied to multi-objective problems

Genetic Algorithm: Example

```python
# Source: https://www.geeksforgeeks.org/genetic-algorithms/
# Python3 program to create target string, starting from
# random string using Genetic Algorithm

import random

# Number of individuals in each generation
POPULATION_SIZE = 100

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

# Target string to be generated
TARGET = "I love GeeksforGeeks"

class Individual(object):
  '''
  Class representing individual in population
  '''
```

Genetic Algorithm: Example

```python
def __init__(self, chromosome):
  self.chromosome = chromosome
  self.fitness = self.cal_fitness()

@classmethod
def mutated_genes(self):
  '''
  create random genes for mutation
  '''
  global GENES
  gene = random.choice(GENES)
  return gene

@classmethod
def create_gnome(self):
  '''
  create chromosome or string of genes
  '''
  global TARGET
  gnome_len = len(TARGET)
```

Genetic Algorithm: Example

```
    return [self.mutated_genes() for _ in range(gnome_len)]

  def mate(self, par2):
    '''
    Perform mating and produce new offspring
    '''

    # chromosome for offspring
    child_chromosome = []
    for gp1, gp2 in zip(self.chromosome, par2.chromosome):

      # random probability
      prob = random.random()

      # if prob is less than 0.45, insert gene
      # from parent 1
      if prob < 0.45:
        child_chromosome.append(gp1)

      # if prob is between 0.45 and 0.90, insert
```

Genetic Algorithm: Example

```
      # gene from parent 2
      elif prob < 0.90:
        child_chromosome.append(gp2)

      # otherwise insert random gene(mutate),
      # for maintaining diversity
      else:
        child_chromosome.append(self.mutated_genes())

    # create new Individual(offspring) using
    # generated chromosome for offspring
    return Individual(child_chromosome)

  def cal_fitness(self):
    '''
    Calculate fittness score, it is the number of
    characters in string which differ from target
    string.
    '''
    global TARGET
```

Genetic Algorithm: Example

```python
    fitness = 0
    for gs, gt in zip(self.chromosome, TARGET):
      if gs != gt: fitness+= 1
    return fitness

# Driver code
def main():
  global POPULATION_SIZE

  #current generation
  generation = 1

  found = False
  population = []

  # create initial population
  for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))
```

Genetic Algorithm: Example

```python
while not found:

  # sort the population in increasing order of fitness score
  population = sorted(population, key = lambda x:x.fitness)

  # if the individual having lowest fitness score ie.
  # 0 then we know that we have reached to the target
  # and break the loop
  if population[0].fitness <= 0:
    found = True
    break

  # Otherwise generate new offsprings for new generation
  new_generation = []

  # Perform Elitism, that mean 10% of fittest population
  # goes to the next generation
  s = int((10*POPULATION_SIZE)/100)
  new_generation.extend(population[:s])
```

Genetic Algorithm: Example

```
    # From 50% of fittest population, Individuals
    # will mate to produce offspring
    s = int((90*POPULATION_SIZE)/100)
    for _ in range(s):
      parent1 = random.choice(population[:50])
      parent2 = random.choice(population[:50])
      child = parent1.mate(parent2)
      new_generation.append(child)

    population = new_generation

    print("Generation: {}\tString: {}\tFitness: {}".\
      format(generation,
      "".join(population[0].chromosome),
      population[0].fitness))

    generation += 1


  print("Generation: {}\tString: {}\tFitness: {}".\
```

Genetic Algorithm: Example

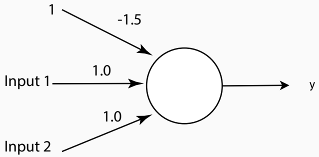
```
    format(generation,
    "".join(population[0].chromosome),
    population[0].fitness))

if __name__ == '__main__':
  main()
```

Excercises

Excercises I

Exercise 1: **Single Layer Perceptron**

Consider a simple perceptron (see Figure): what will the output be when the input is (0, 0)? What about inputs (0, 1), (1, 1) and (1, 0)? What if we change the bias weight to -0.5?



Excercises II

Exercise 2: **Basis Functions**

Given a test vector x_i, the output of a neural network is defined as

$$f(x_i) = \sum_{j=0}^{M} w_j \phi_j(x_i, v_j) \ . \tag{171}$$

The weights of the neurons can be learned by employing the back-propagation rule with sample-based gradient descent. In the lecture neural networks with sigmoid neurons have been introduced, but it is possible to employ different basis functions:

- Which properties do these basis functions have to fulfill?
- Is the number of parameters for $\phi(x_i, v_j)$ limited? Could several different basis functions be used for the same neural network?

Exercise 3: **Error Convergence**

Given 2-1 network trained with one single pattern by means of back-propagation of error and learning rate $\eta = 0.1$. Let the pattern (p, t) be defined by $p = (p1, p2) = (0.3, 0.7)$. Verify whether the error

$$E = \frac{1}{2}(t - y)^2 \tag{172}$$

converges and if so, at what value?

Bibliography

Bibliography I

References

[1] Maxwell W. Libbrecht and William Stafford Noble. Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321–332, 2015.

[2] Andrey Kan. Machine learning applications in cell image analysis. *Immunology & Cell Biology*, 95(6):525–530, 2020/02/24 2017.

[3] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[4] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv e-prints*, page arXiv:1412.6980, 12 2014.

Bibliography II

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv e-prints*, page arXiv:1603.04467, 03 2016.

[7] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[8] Frank Rosenblat. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[9] Marvin Minsky and Seymour A. Papert. *Perceptrons: An introduction to Computational Geometry*. MIT Press, 1969.

[10] Proc. Nat. Acad. Sci. 79 2554-2558 J. J. Hopfield. 1982. 1982.

[11] J. Reinitz E. Mjolsness, D. H. Sharp. *J. Theor. Biol.*, 152:429–453, 1991.

[12] Mechanisms of Development 49 133-158 J. Reinitz, D. H. Sharp. 1995. 1995.

Bibliography III

[13] M. Kaufman R. Thomas, D. Thieffry. *Bul. Math. Biol.*, 57(2):247–276, 1995.

[14] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, and David Haussler. Hidden markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235(5):1501–1531, 1994.

[15] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[16] R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[17] Schulman J. Wolski F. Dhariwal P. Radford A. and Klimov O. Proximal policy optimization algorithms. In *International Conference on Learning Representations*, 2017.

[18] Razvan V. Florian. Correct equations for the dynamics of the cart-pole system. Technical report, 2020.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[20] *https://arxiv.org/abs/1506.02438*, 2018.

Index

Index I

Index II

Index III

Index IV

Index V

Index VI