Microsoft

Deep Learning World Workshop

James McCaffrey & Ricky Loynd
Microsoft Research

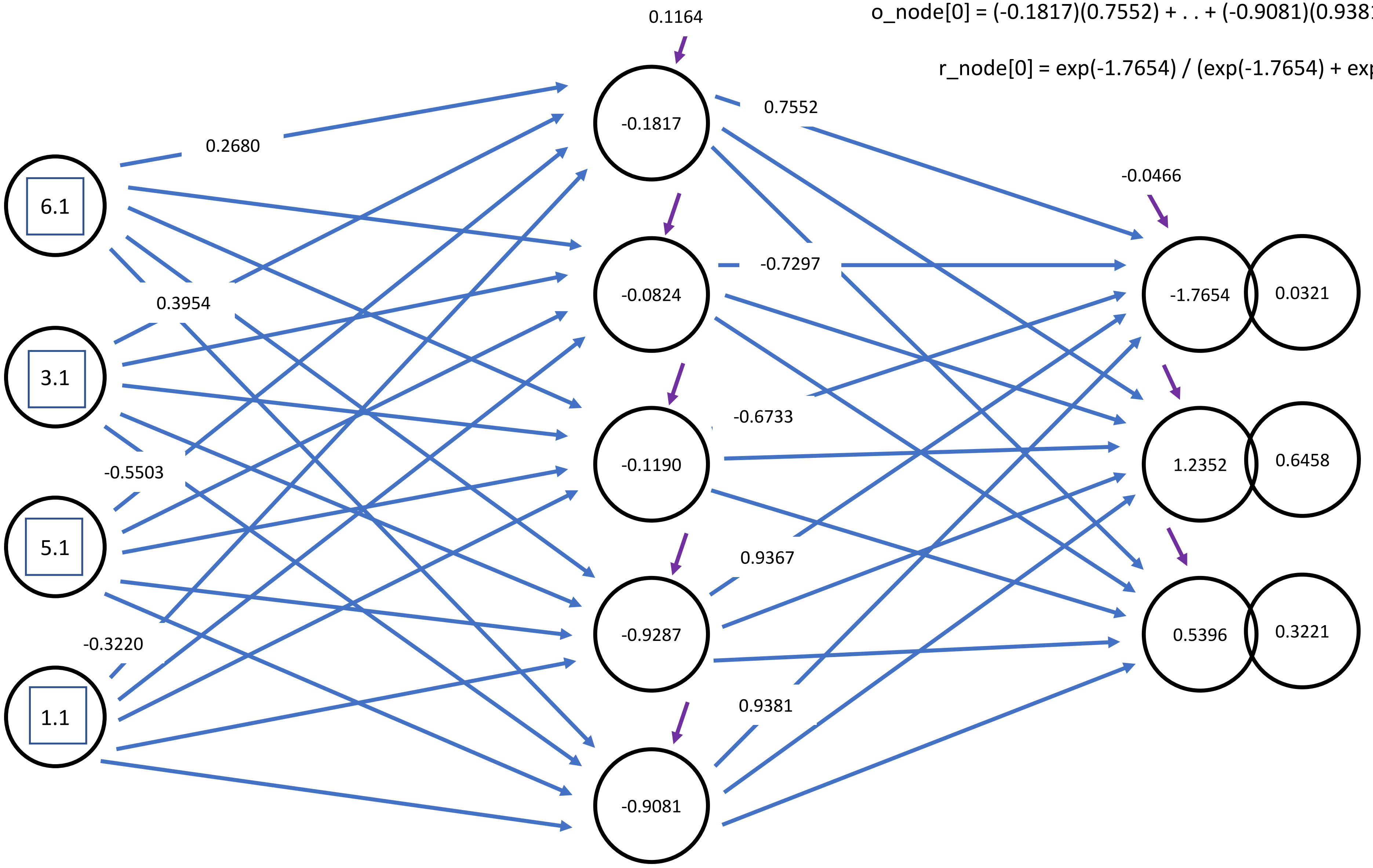
Monday, June 4, 2018

1. Neural Network Input-Output Mechanism

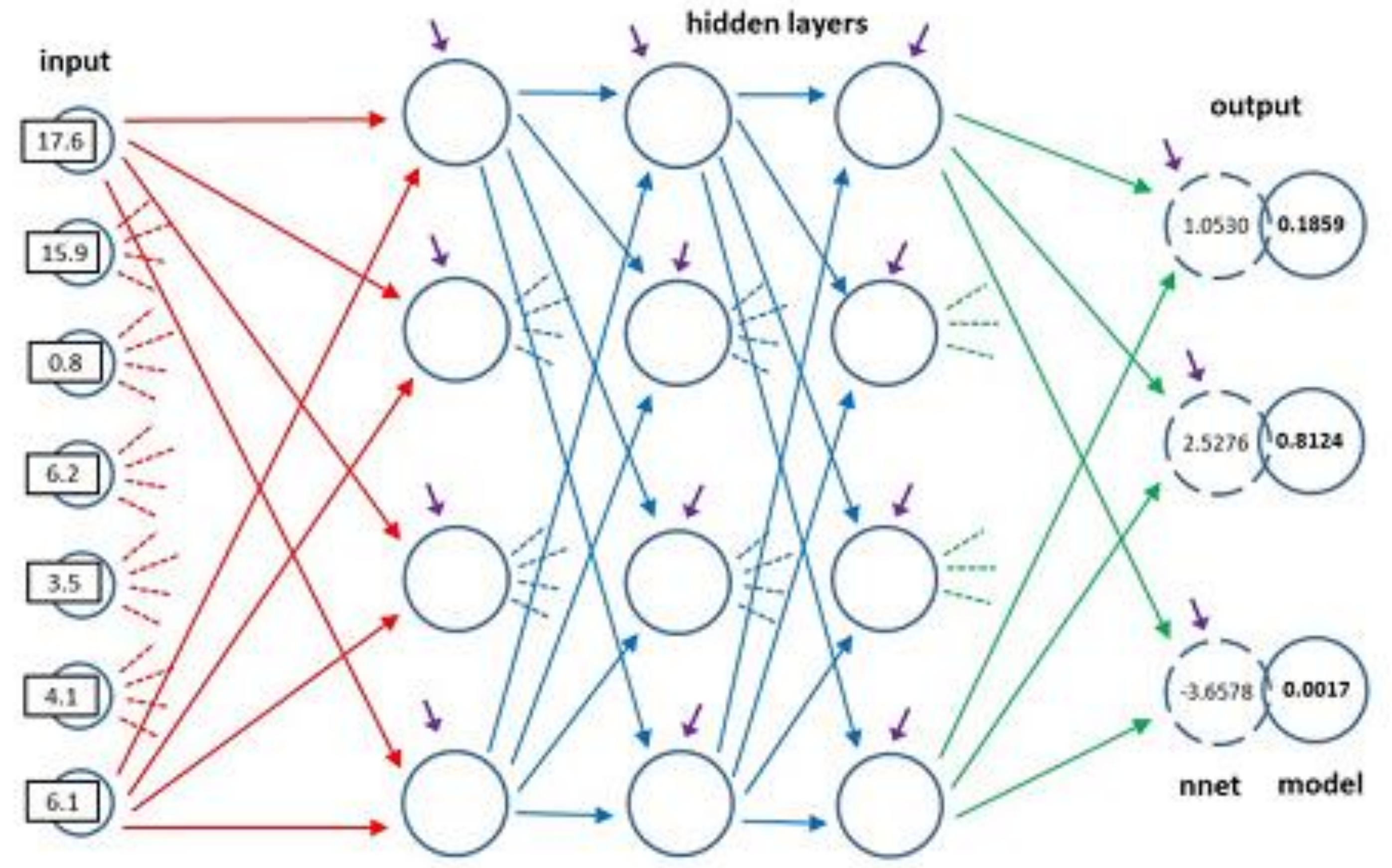
h_node[0] = tanh((6.1)(0.2680) + . . + (1.1)(-0.3220) + 0.1164) = -0.1817

o_node[0] = (-0.1817)(0.7552) + . . + (-0.9081)(0.9381) + -0.0466 = -1.7654

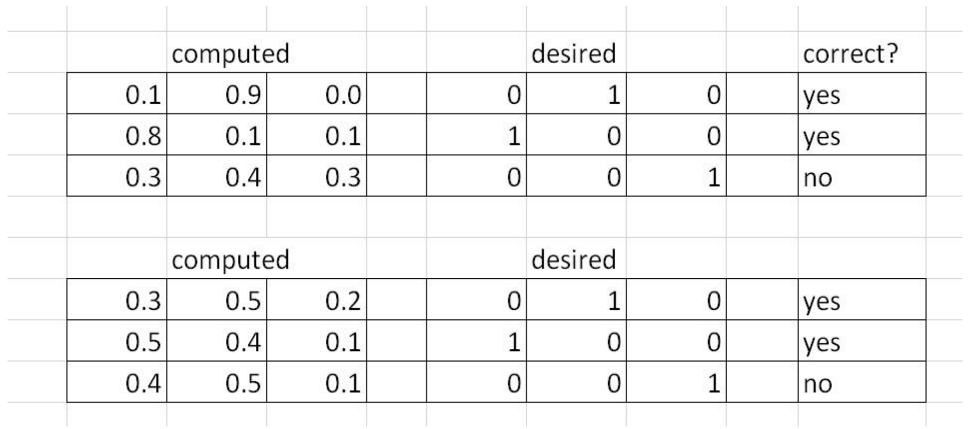
r_node[0] = exp(-1.7654) / (exp(-1.7654) + exp(1.2352) + exp(0.5396)) = 0.0321



1. Neural Network Input-Output Mechanism



2. Error and Accuracy



Model A

computed				desired			correct?
0.1	0.9	0.0		0	1	0	yes
0.8	0.1	0.1		1	0	0	yes
0.3	0.4	0.3		0	0	1	no

Model B

computed				desired			
0.3	0.5	0.2		0	1	0	yes
0.5	0.4	0.1		1	0	0	yes
0.4	0.5	0.1		0	0	1	no

Both models 0.67 classification accuracy

Mean squared error:

A: $(0.1 - 0)^2 + (0.9 - 1)^2 + (0.0 - 0)^2 +$
$(0.8 - 1)^2 + (0.1 - 0)^2 + (0.1 - 0)^2 +$
$(0.3 - 0)^2 + (0.4 - 0)^2 + (0.3 - 1)^2 \;/\; 3 = 0.273$

B: $(0.3 - 0)^2 + (0.5 - 1)^2 + (0.2 - 0)^2 +$
$(0.5 - 1)^2 + (0.4 - 0)^2 + (0.1 - 0)^2 +$
$(0.4 - 0)^2 + (0.5 - 0)^2 + (0.1 - 1)^2 \;/\; 3 = 0.733$

Mean cross entropy error:

A: $-(\ln(0.1) * 0 + \ln(0.9) * 1 + \ln(0.0) * 0) + \ldots \;/\; 3 = 0.511$

B: $-(\ln(0.3) * 0 + \ln(0.5) * 1 + \ln(0.2) * 0) + \ldots \;/\; 3 = 1.230$

3. Encoding and Normalization

Predictor and Label Encoding

1-of-(N-1) Coding

```
red    = ( 1,  0,  0)
yellow = ( 0,  1,  0)
blue   = ( 0,  0,  1)
green  = (-1, -1, -1)

binary: (-1, +1)
```

1-of-N / "one-hot" Coding

```
red    = ( 1,  0,  0,  0)
yellow = ( 0,  1,  0,  0)
blue   = ( 0,  0,  1,  0)
green  = ( 0,  0,  0,  1)

binary: (0, 1)
```

Normalization

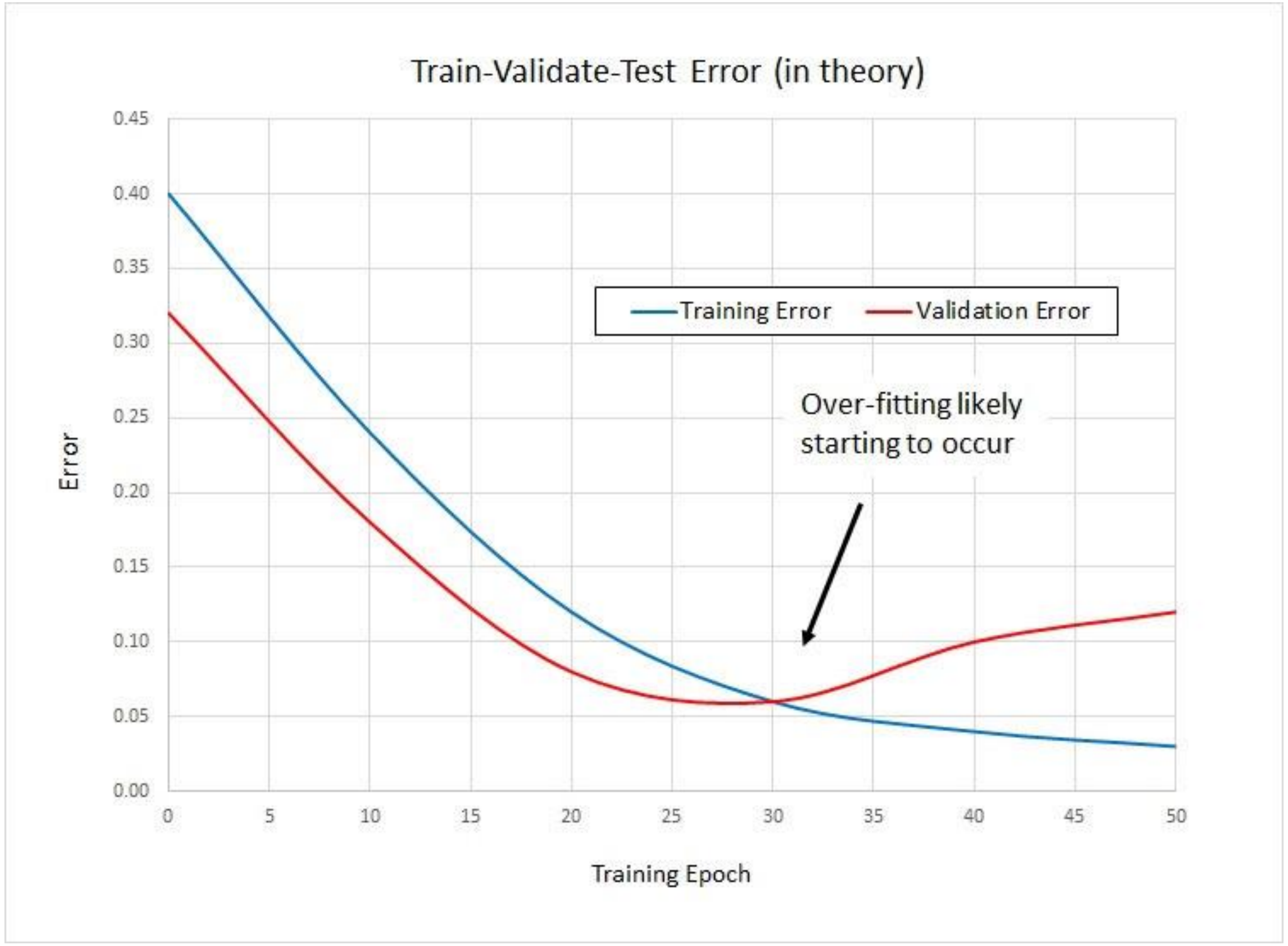
						z-score		min-max		order magnitude	
	Age	**Income**	**Education**	**Sex**	**Political**	**Age**	**Income**	**Age**	**Income**	**Age**	**Income**
	24	$24,000.00	Low	F	Democrat	-1.20	-1.30	0.00	0.00	2.40	2.40
	62	$82,000.00	Medium	M	Republican	1.68	1.56	1.00	1.00	6.20	8.20
	38	$64,000.00	High	M	Other	-0.14	0.67	0.37	0.69	3.80	6.40
	30	$40,000.00	Medium	F	Democrat	-0.74	-0.51	0.16	0.28	3.00	4.00
	45	$42,000.00	High	F	Reublican	0.39	-0.41	0.55	0.31	4.50	4.20
mean	39.80	$50,400.00	NA	NA	NA						
sd	13.18	$20,293.84									
max	62	$82,000.00									
min	24	$24,000.00									
max-min	38	$58,000.00									

$v' = (v - mean) / sd$

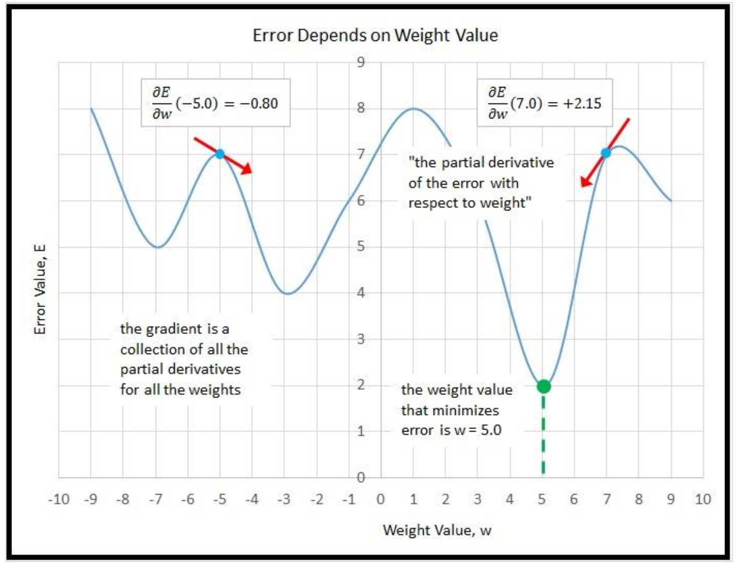
$v' = (v - min) / (max - min)$

$v' = v / K$

4. Train-Test and Train-Validate-Test and k-Fold Cross Validation



5. Back-Propagation, Stochastic Gradient Descent, Momentum, Batch, Online, Mini-Batch



$$E = \frac{1}{2} * \sum (t_k - o_k)^2$$ (with log-sig or softmax)

$$\Delta w_{jk} = \underset{\substack{\uparrow \\ \text{learning rate}}}{\eta} * \overbrace{[\, x_j * \underbrace{\underbrace{(o_k - t_k)}_{\text{error } e_k} * \underbrace{o_k * (1 - o_k)}_{\text{derivative of output activation } \varphi_k{}'}}_{\text{signal } \delta_k} \,]}^{\frac{\partial E}{\partial w_{jk}}}$$

note: the math equation for delta often leaves out the explicit -1 factor

note: if you use cross entropy instead of squared error, the derivative of the output activation cancels out to 1 by some rather remarkable algebra

$$\Delta w_{ij} = \underset{\substack{\uparrow \\ \text{learning rate}}}{\eta} * \overbrace{[\, x_i * \underbrace{\underbrace{\sum \delta_k w_{jk}}_{\text{error } e_j} * \underbrace{(1 + h_j) * (1 - h_j)}_{\text{derivative of hidden activation } \varphi_j{}'}}_{\text{signal}} \,]}^{\frac{\partial E}{\partial w_{ij}}}$$

note: assumes hidden activation is tanh()

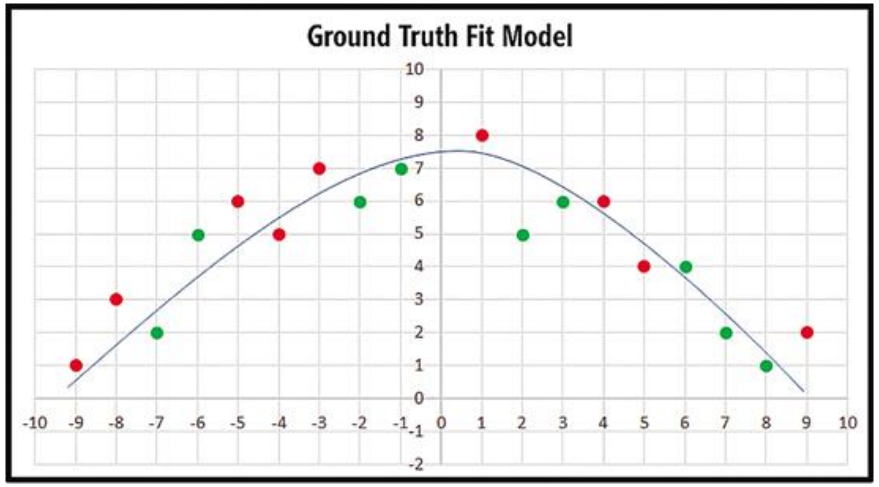
Standard Momentum

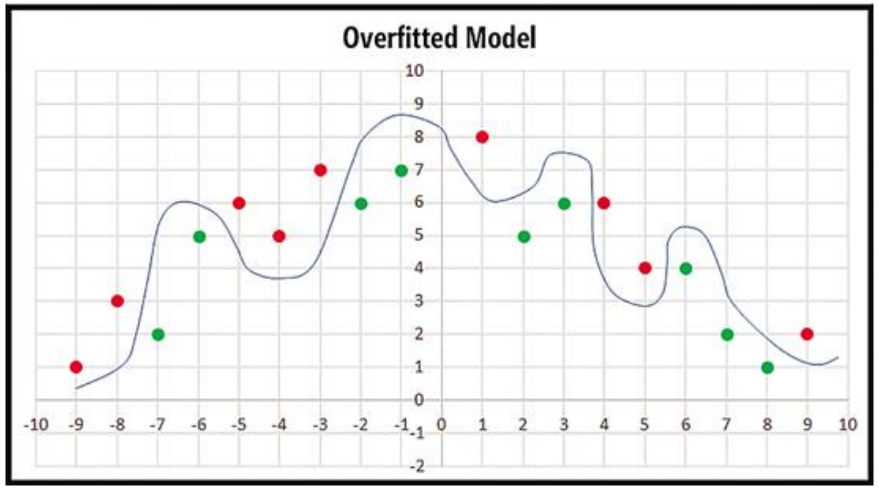
$$\Delta w_{ij} = -1 * \left(\eta * \frac{\partial E}{\partial w_{ij}}\right) + (\alpha * prev(\Delta w_{ij}))$$

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

```
# 1. update input-to-hidden weights
for i in range(self.ni):
  for j in range(self.nh):
    delta = learnRate * ihGrads[i,j]
    self.ihWeights[i,j] += delta
    self.ihWeights[i,j] += momentum * ih_prev_weights_delta[i,j]
    ih_prev_weights_delta[i,j] = delta  # save
```

6. Overfitting and Regularization





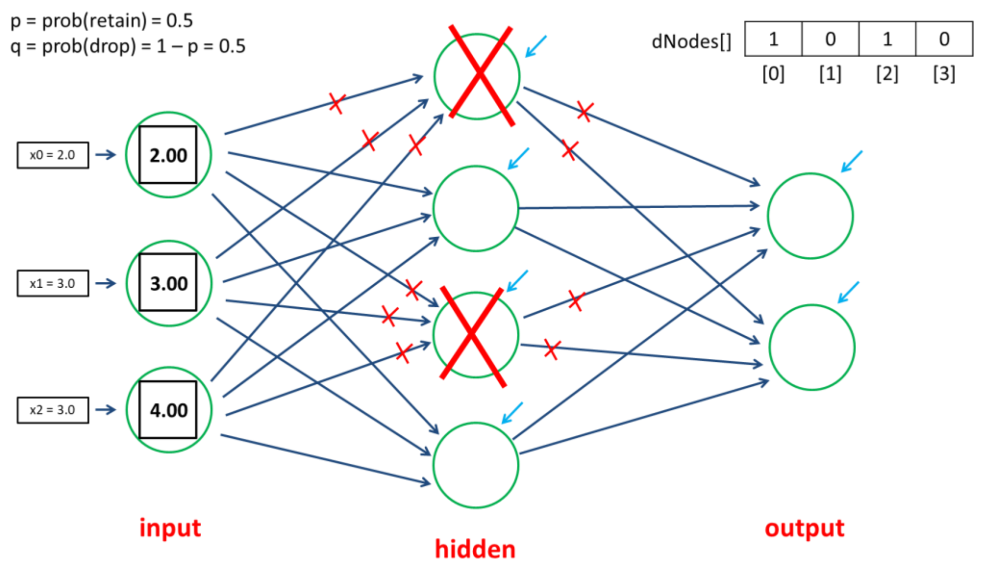
L2 Regularization

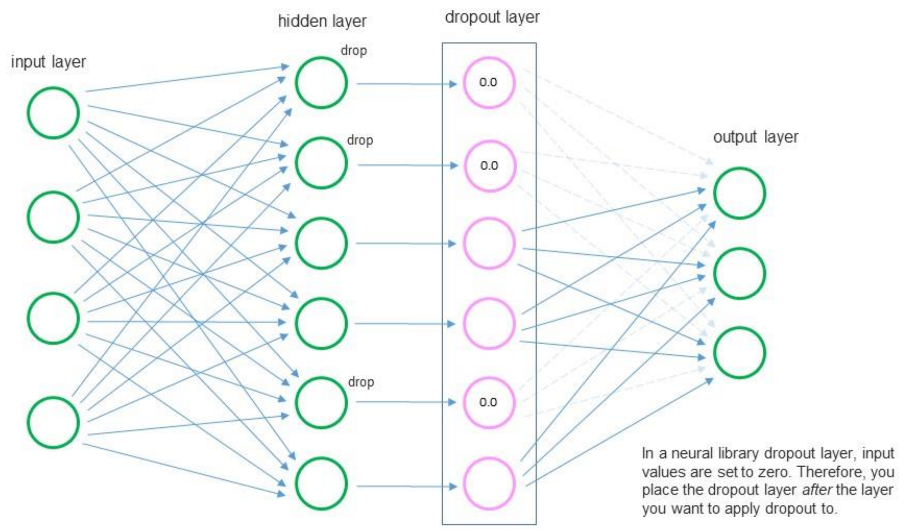
$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{plain error}} + \underbrace{\frac{\lambda}{2} * \sum {w_i}^2}_{\text{weight penalty}}$$

elegant math $\Downarrow$ $\Downarrow$ simple math

$$\Delta w_{jk} = \underset{\substack{\uparrow \\ \text{learning rate}}}{\eta} * \overbrace{[\, x_j * \underbrace{(o_k - t_k) * o_k * (1 - o_k) \,] + [\, \lambda * w_{jk} \,]}_{\text{signal}}}^{\frac{\partial E}{\partial w_{jk}}}$$

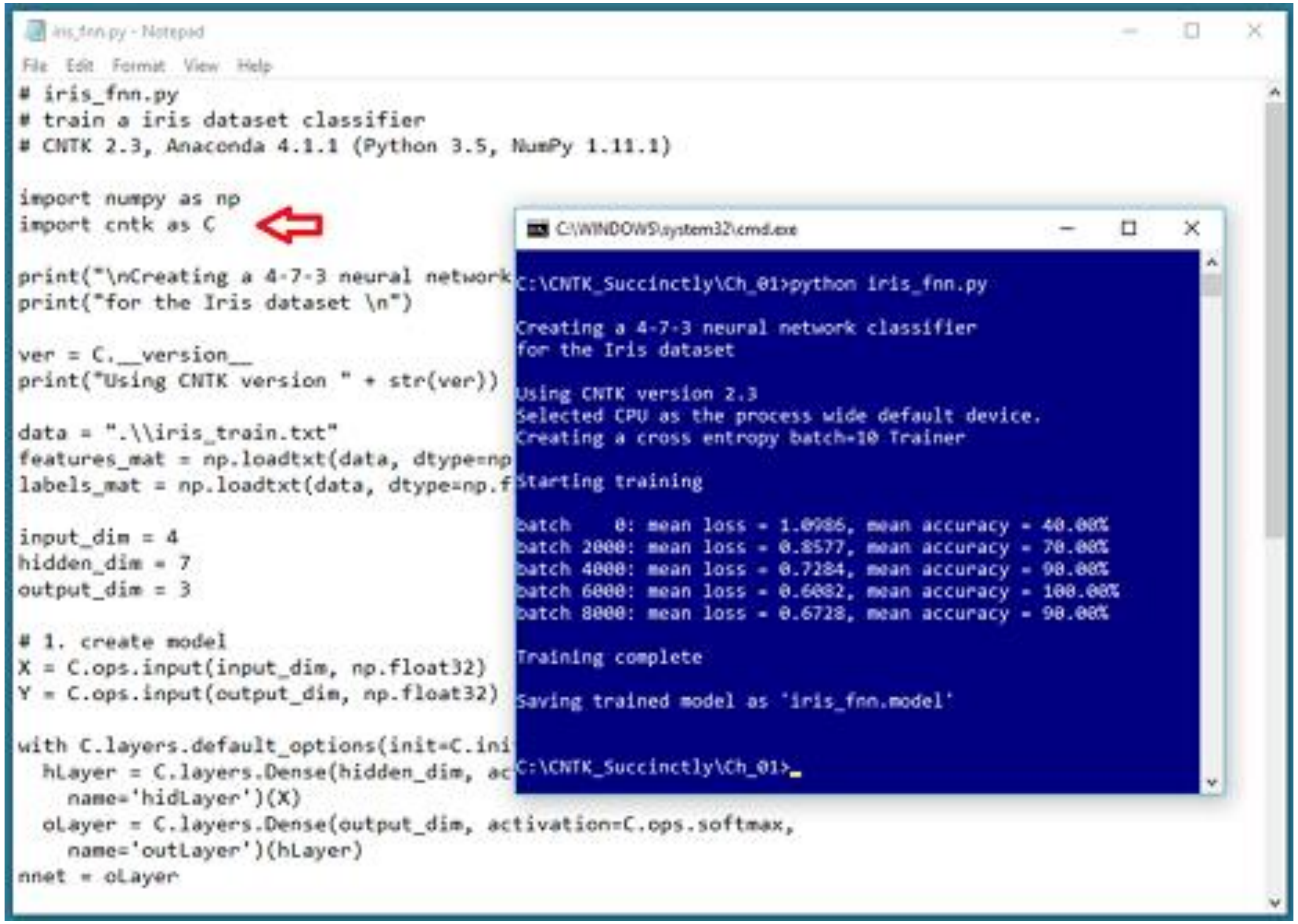
7. Overfitting and Dropout





In a neural library dropout layer, input values are set to zero. Therefore, you place the dropout layer *after* the layer you want to apply dropout to.

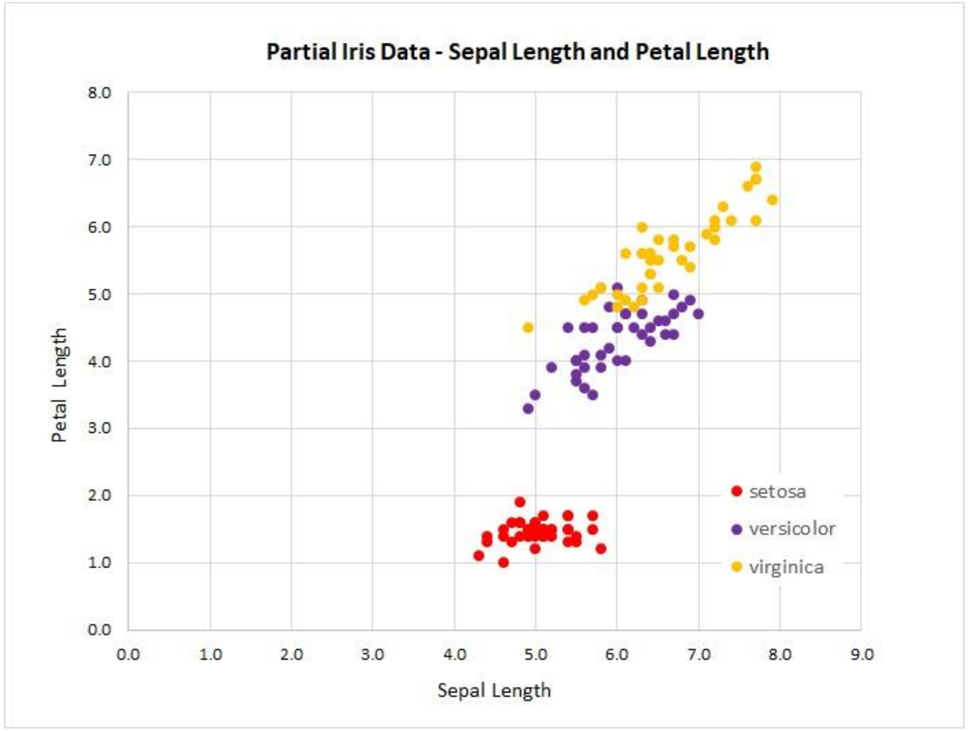
8. Python Ecosystem plus CNTK or TensorFlow with Keras

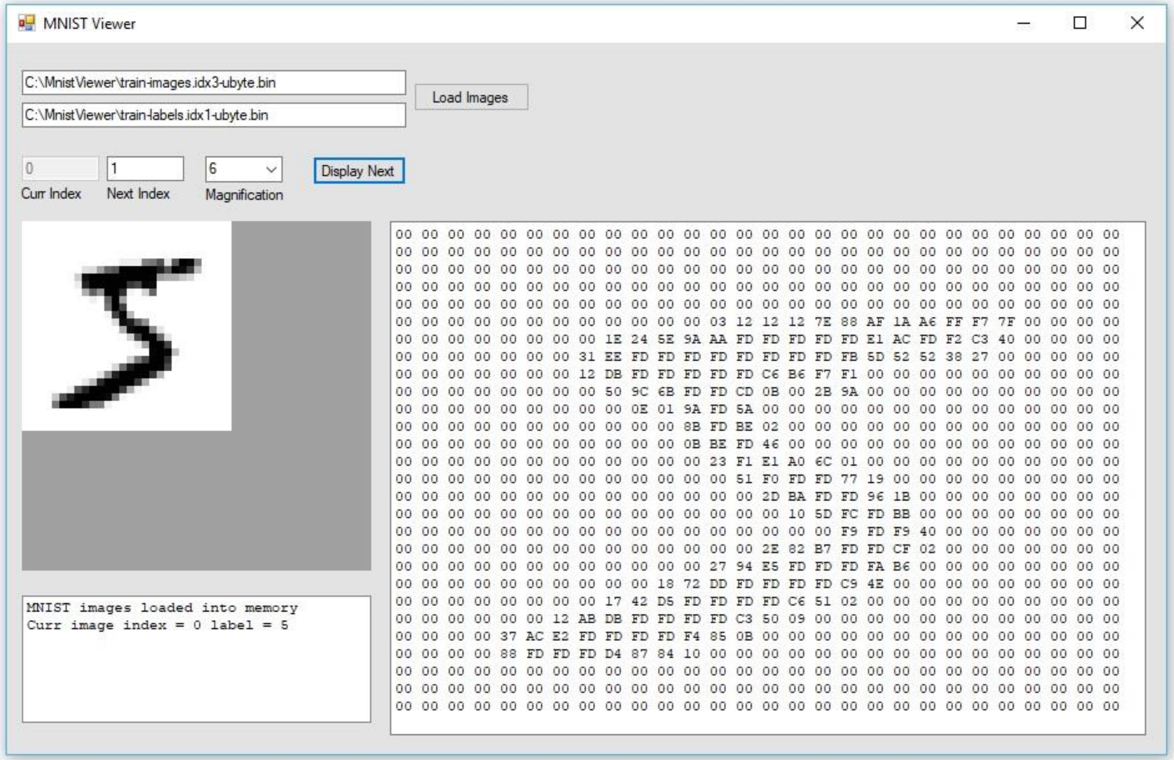


```
# iris_fnn.py
# train a iris dataset classifier
# CNTK 2.3, Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

import numpy as np
import cntk as C

print("\nCreating a 4-7-3 neural network
print("for the Iris dataset \n")

ver = C.__version__
print("Using CNTK version " + str(ver))

data = ".\\iris_train.txt"
features_mat = np.loadtxt(data, dtype=np
labels_mat = np.loadtxt(data, dtype=np.f

input_dim = 4
hidden_dim = 7
output_dim = 3

# 1. create model
X = C.ops.input(input_dim, np.float32)
Y = C.ops.input(output_dim, np.float32)

with C.layers.default_options(init=C.ini
  hLayer = C.layers.Dense(hidden_dim, ac
    name='hidLayer')(X)
  oLayer = C.layers.Dense(output_dim, activation=C.ops.softmax,
    name='outLayer')(hLayer)
nnet = oLayer
```

```
C:\CNTK_Succinctly\Ch_01>python iris_fnn.py

Creating a 4-7-3 neural network classifier
for the Iris dataset

Using CNTK version 2.3
Selected CPU as the process wide default device.
Creating a cross entropy batch=10 Trainer

Starting training

batch    0: mean loss = 1.0986, mean accuracy = 40.00%
batch 2000: mean loss = 0.8577, mean accuracy = 70.00%
batch 4000: mean loss = 0.7284, mean accuracy = 90.00%
batch 6000: mean loss = 0.6082, mean accuracy = 100.00%
batch 8000: mean loss = 0.6728, mean accuracy = 90.00%

Training complete

Saving trained model as 'iris_fnn.model'


C:\CNTK_Succinctly\Ch_01>_
```

9. The Iris and MNIST Datasets





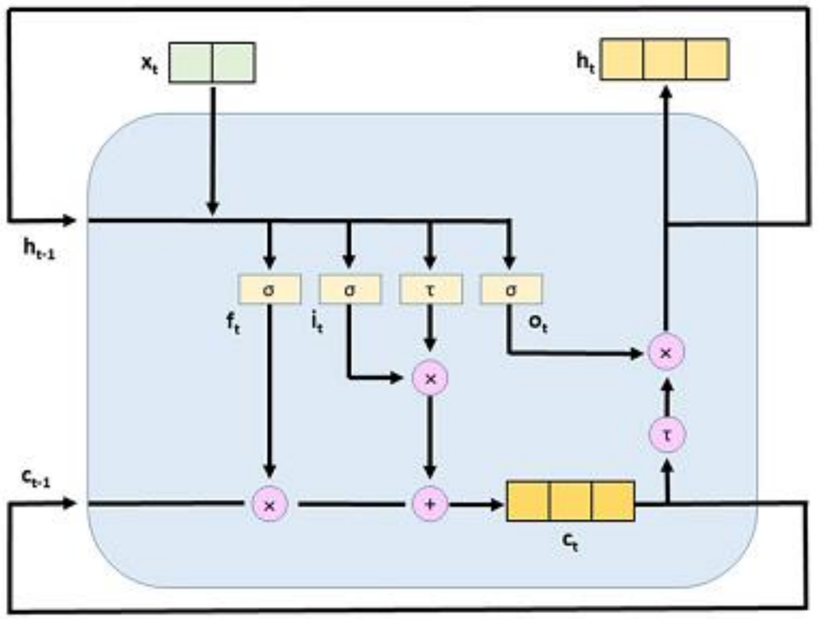
C:\MnistViewer\train-images.idx3-ubyte.bin
C:\MnistViewer\train-labels.idx1-ubyte.bin

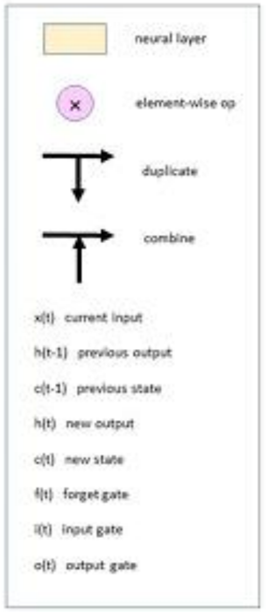
Curr Index: 0
Next Index: 1
Magnification: 6

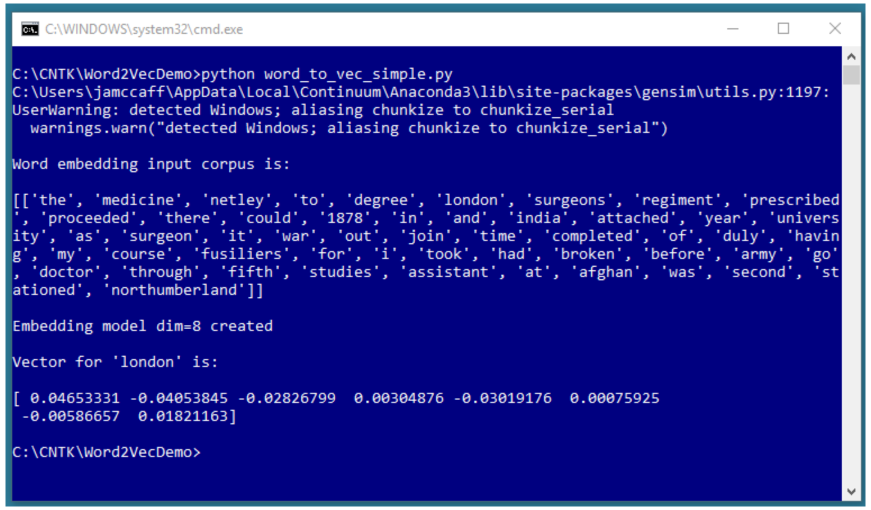
```
MNIST images loaded into memory
Curr image index = 0 label = 5
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 12 12 12 7E 88 AF 1A A6 FF F7 7F 00 00 00 00
00 00 00 00 00 00 00 00 1E 24 5E 9A AA FD FD FD FD FD E1 AC FD F2 C3 40 00 00 00 00
00 00 00 00 00 00 00 31 EE FD FD FD FD FD FD FD FD FB 5D 52 52 38 27 00 00 00 00 00
00 00 00 00 00 00 00 12 DB FD FD FD FD FD C6 B6 F7 F1 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 50 9C 6B FD FD CD 0B 00 2B 9A 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 0E 01 9A FD 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 8B FD BE 02 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 0B BE FD 46 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 23 F1 E1 A0 6C 01 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 51 F0 FD FD 77 19 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 2D BA FD FD 96 1B 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 5D FC FD BB 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F9 FD F9 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 82 B7 FD FD CF 02 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 27 94 E5 FD FD FD FA B6 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 18 72 DD FD FD FD FD C9 4E 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 17 42 D5 FD FD FD FD C6 51 02 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 12 AB DB FD FD FD FD C3 50 09 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 37 AC E2 FD FD FD FD F4 85 0B 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 88 FD FD FD D4 87 84 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

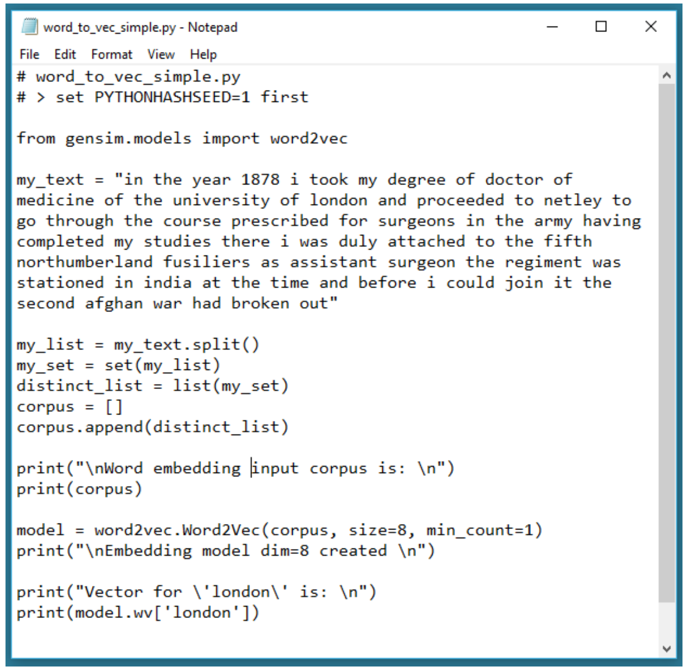
10. Word Embeddings and LSTM Recurrent Networks







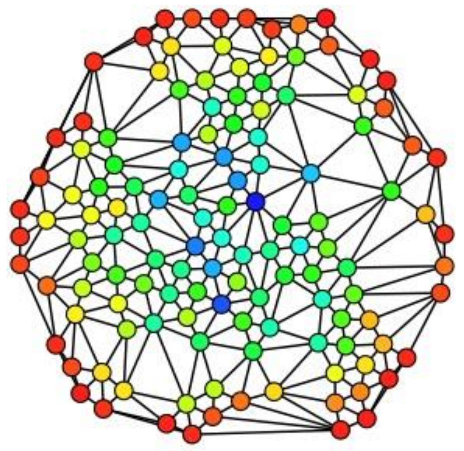
```
C:\CNTK\Word2VecDemo>python word_to_vec_simple.py
C:\Users\jamccaff\AppData\Local\Continuum\Anaconda3\lib\site-packages\gensim\utils.py:1197:
UserWarning: detected Windows; aliasing chunkize to chunkize_serial
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")

Word embedding input corpus is:

[['the', 'medicine', 'netley', 'to', 'degree', 'london', 'surgeons', 'regiment', 'prescribed
', 'proceeded', 'there', 'could', '1878', 'in', 'and', 'india', 'attached', 'year', 'univers
ity', 'as', 'surgeon', 'it', 'war', 'out', 'join', 'time', 'completed', 'of', 'duly', 'havin
g', 'my', 'course', 'fusiliers', 'for', 'i', 'took', 'had', 'broken', 'before', 'army', 'go'
, 'doctor', 'through', 'fifth', 'studies', 'assistant', 'at', 'afghan', 'was', 'second', 'st
ationed', 'northumberland']]

Embedding model dim=8 created

Vector for 'london' is:

[ 0.04653331 -0.04053845 -0.02826799  0.00304876 -0.03019176  0.00075925
 -0.00586657  0.01821163]

C:\CNTK\Word2VecDemo>
```



```
# word_to_vec_simple.py
# > set PYTHONHASHSEED=1 first

from gensim.models import word2vec

my_text = "in the year 1878 i took my degree of doctor of
medicine of the university of london and proceeded to netley to
go through the course prescribed for surgeons in the army having
completed my studies there i was duly attached to the fifth
northumberland fusiliers as assistant surgeon the regiment was
stationed in india at the time and before i could join it the
second afghan war had broken out"

my_list = my_text.split()
my_set = set(my_list)
distinct_list = list(my_set)
corpus = []
corpus.append(distinct_list)

print("\nWord embedding input corpus is: \n")
print(corpus)

model = word2vec.Word2Vec(corpus, size=8, min_count=1)
print("\nEmbedding model dim=8 created \n")

print("Vector for \'london\' is: \n")
print(model.wv['london'])
```

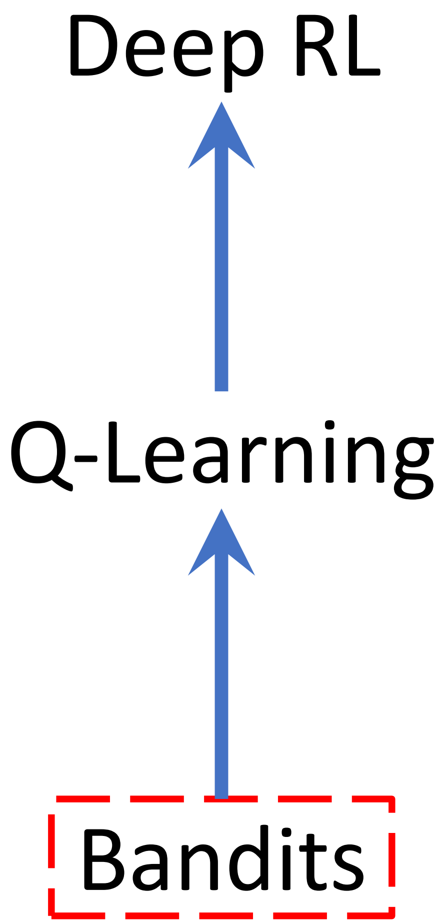
Recap

1. Neural network input-output (weights, biases, activation, softmax)
2. Error and accuracy (squared error, cross entropy error aka log loss)
3. Encoding and normalization (one-hot, 1-of-(n-1), min-max, z-score, order-n)
4. Train-validate-test
5. Back-propagation (stochastic gradient descent, learning rate, gradients, batch and mini-batch, momentum)
6. Regularization (overfitting, L1 vs. L2)
7. Dropout
8. Python and libraries (NumPy, CNTK, TensorFlow, Keras)
9. Standard datasets (Fisher's Iris data, MNSIT image classification, IMDB sentiment analysis)
10. LSTM networks (recurrent networks, Elman and Jordan networks, IMDB sentiment analysis)

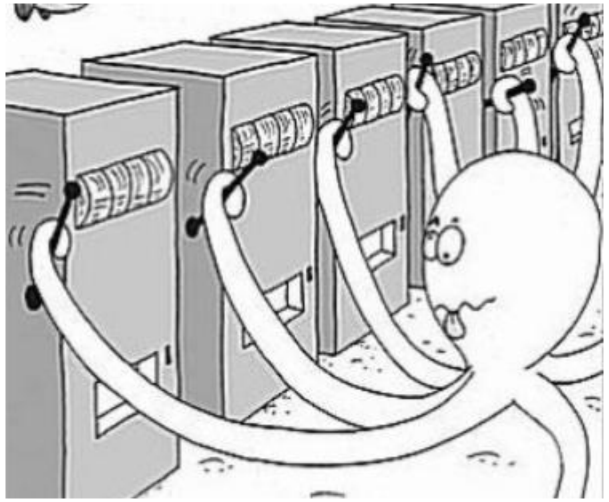


"betweenness centrality"

Reinforcement Learning: Bandits


Deep RL
Q-Learning
Bandits

Multi-armed Bandit



Different arms give reward with different probabilities

- **Supervised learning** requires correct (labeled) outputs/actions.
- But **rewards** are relative, so a single reward reveals nothing.
- Model-based RL would reveal the bandit's percentages of payout.
- But **model-free RL** agents can only try things out to discover which actions give the most reward.
- The main problem is balancing **exploration vs. exploitation**.
  - Always going to your **favorite restaurant** means you can't discover a better one.
  - Always trying **new restaurants** means you spend little time enjoying the best ones.

1. Create a 3-armed bandit environment. Let the user enter each action manually. Output each reward received. Set each reward to either 1.0 or 0.0. Fix the probabilities of non-zero reward to be [0.2, 0.5, 0.7] for the three possible actions.

- Open a cmd window.
- Navigate to PAW2018\Bandits\code
- Run the program
  - python bandits.py
  - Type an action (integer between 0 and 2):
- Try the action 2 enough times to double-check its probability of reward (0.7).
- Exit the program by hitting Enter.

2. Connect a uniform random agent to the environment. Output the mean reward received over 1000 steps.

- Edit the code
  - notepad bandits.py
  - Find where USE_RANDOM_AGENT is defined.
  - Set USE_RANDOM_AGENT (only) to True.
  - Find where USE_RANDOM_AGENT is used.
    - Lines 30 & 31
    - action = np.random.randint(0, N)
  - Save the file.

- Run the program again.
  - Mean reward received = 0.467

3. Modify the agent to record the reward received for each action, and use the epsilon-greedy algorithm to balance exploration with exploitation. Find the value of epsilon that earns the most reward over 1000 total steps.

- Epsilon-greedy algorithm
  - Flip a (biased) coin to choose whether to explore or exploit.
  - If exploring, take a random action.
  - If exploiting, take the best action, based on past rewards.
- See how USE_EPSILON_GREEDY is used in the code.
- Set USE_EPSILON_GREEDY (only) to True.
- Run the program again.
  - Mean reward received = 0.681   (What would a perfect score be?)
- Tune the epsilon hyperparameter for maximum reward.
- Definition: Policy = An agent’s rules for choosing actions.
  - In this case, the core of the policy is the “model” stored in the probs array.

Can we do better than epsilon-greedy?

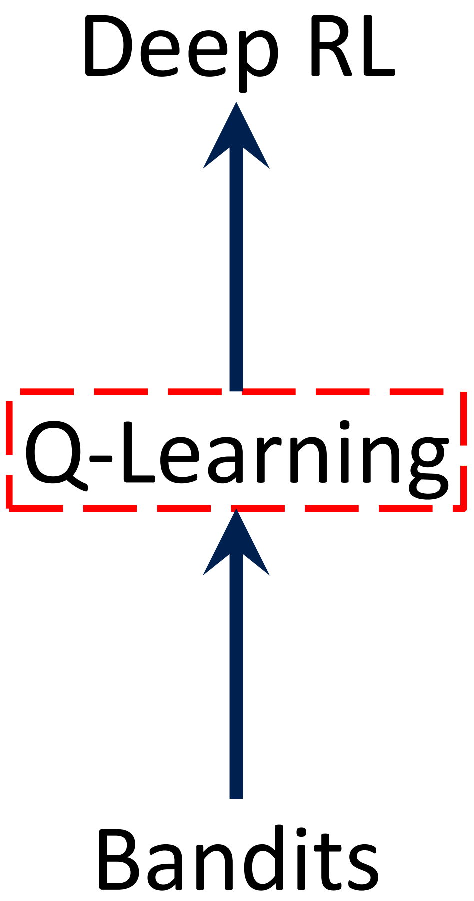
- Weaknesses of epsilon-greedy:
  - Exploration continues forever, even after we are sure of the best action.
    - One solution to this problem is explained in an extra exercise.
  - It seems brittle to use max to take the best action between two almost-equal actions.

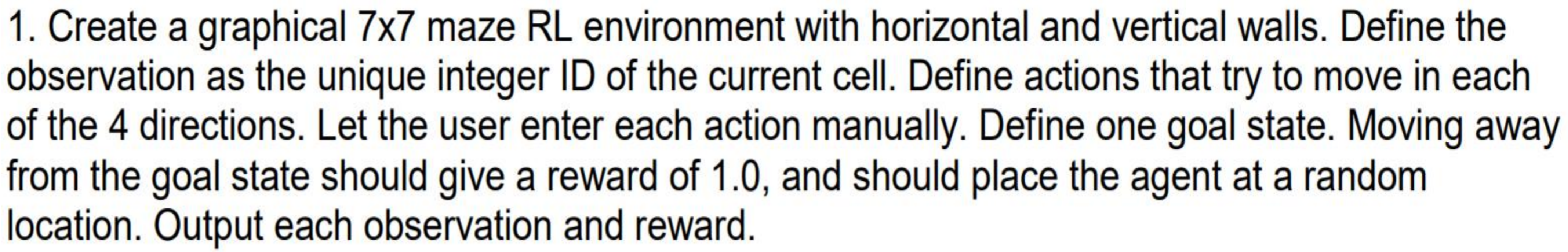
- One solution: Thompson Sampling
  - Combine exploration and exploitation in the same smooth operation.
  - Use the history of rewards to generate a random Beta distribution, different each step.
  - Use the highest part of the Beta distribution to define the action.

4. Modify the agent to use Thompson Sampling instead of epsilon-greedy. Compare the new mean reward received to epsilon-greedy's performance.

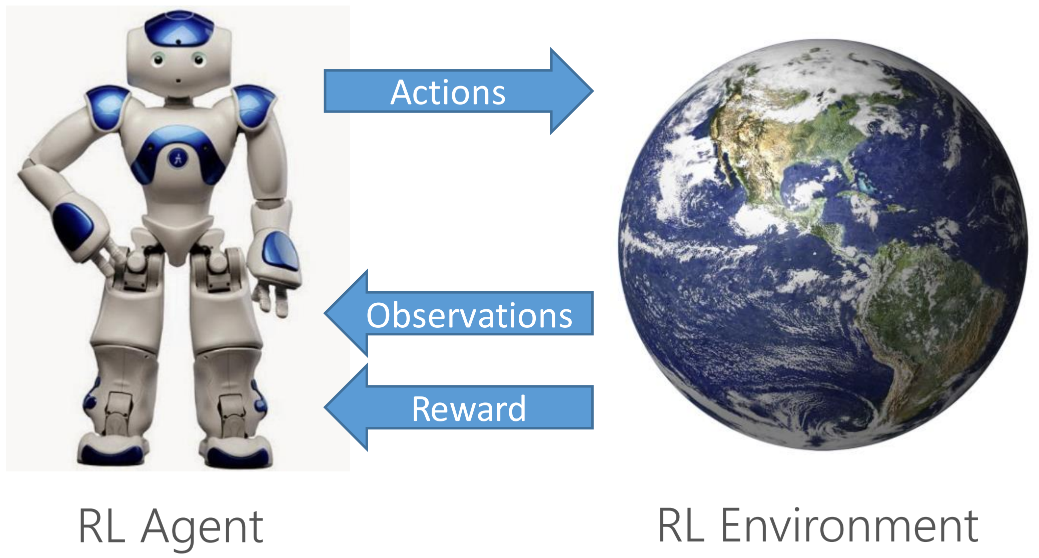
- See how USE_THOMPSON_SAMPLING is used in the code.
- Set USE_THOMPSON_SAMPLING (only) to True.
- Run the program again.
  - Mean reward received = 0.652
- Raise the number of trials to 10000
- Run the program again for Thompson Sampling, and then for epsilon-greedy.
- Which method is better now? Why?

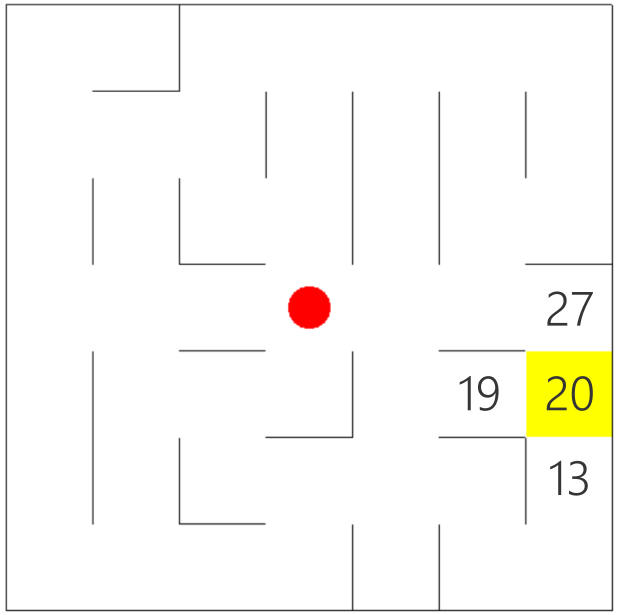
Reinforcement Learning: Q-Learning


Deep RL
Q-Learning
Bandits

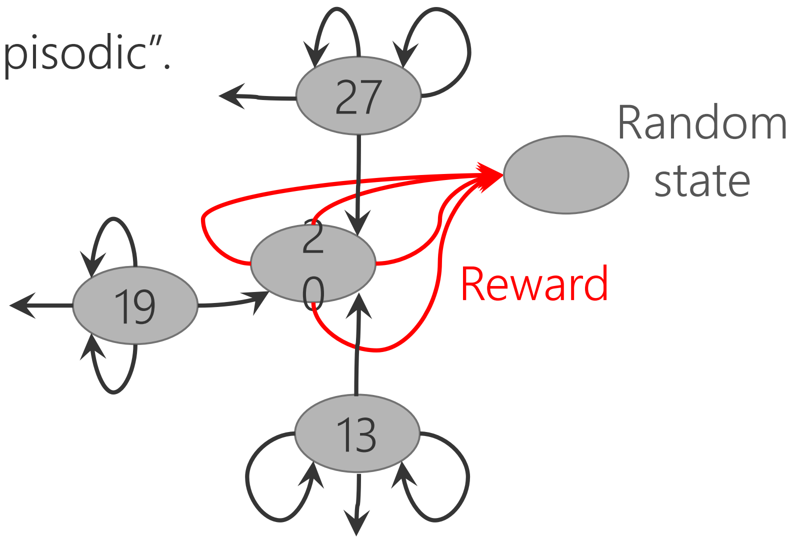

1. Create a graphical 7x7 maze RL environment with horizontal and vertical walls. Define the observation as the unique integer ID of the current cell. Define actions that try to move in each of the 4 directions. Let the user enter each action manually. Define one goal state. Moving away from the goal state should give a reward of 1.0, and should place the agent at a random location. Output each observation and reward.

- In a cmd window, navigate to PAW2018\Q_Learning\code
- Run the program
  - python demo.py --display
  - Try it out. How do you earn reward?
- How is this maze environment different than a bandit problem?
  - The right action depends on where you are, which depends on your past actions.
- What is the meaning of the Observation number at the top?
- Exit the program by entering Alt-F4, or by clicking the X.





- This maze is a Markov Decision Process (MDP).
- The agent's current cell defines the environment's Markov state.
- Actions can lead to other states, and can receive reward.
- Future events depend only on the current state and action.
- Each agent observation is simply the ID of the current state.
- The agent doesn't (directly) observe the walls of the maze.
- The agent doesn't (directly) observe the structure of the MDP.
- This maze is a "continuing" task, but could be made "episodic".



Part of this maze's MDP representation

2. Connect a random agent to the environment. Output the mean reward received over 10,000 steps. Disable the graphical display for fast evaluation.

- The random agent is already connected.
- To run the random agent for 10,000 steps, without the display:
  - python demo.py --train
  - Mean reward per step = 0.0041

How can an agent learn to solve a maze like this?

- By supervised learning?
  - Would require the correct action label for every possible state.

- Could a bandit-solving agent solve this maze?
  - If reward was given after every action, and each state had a different agent.
  - But the reward can be greatly delayed. (Reward is sparse.)

- Can we treat every possible series of actions (trajectory) like a single action?
  - That would create too many (infinite) actions.

- We have to consider whole trajectories as series of actions, with sparse rewards.
  - This requires some algorithm for **combining** rewards over trajectories.

Running (exponential) average

Learning rate   Target data point

$$A \leftarrow A + \alpha(T - A)$$

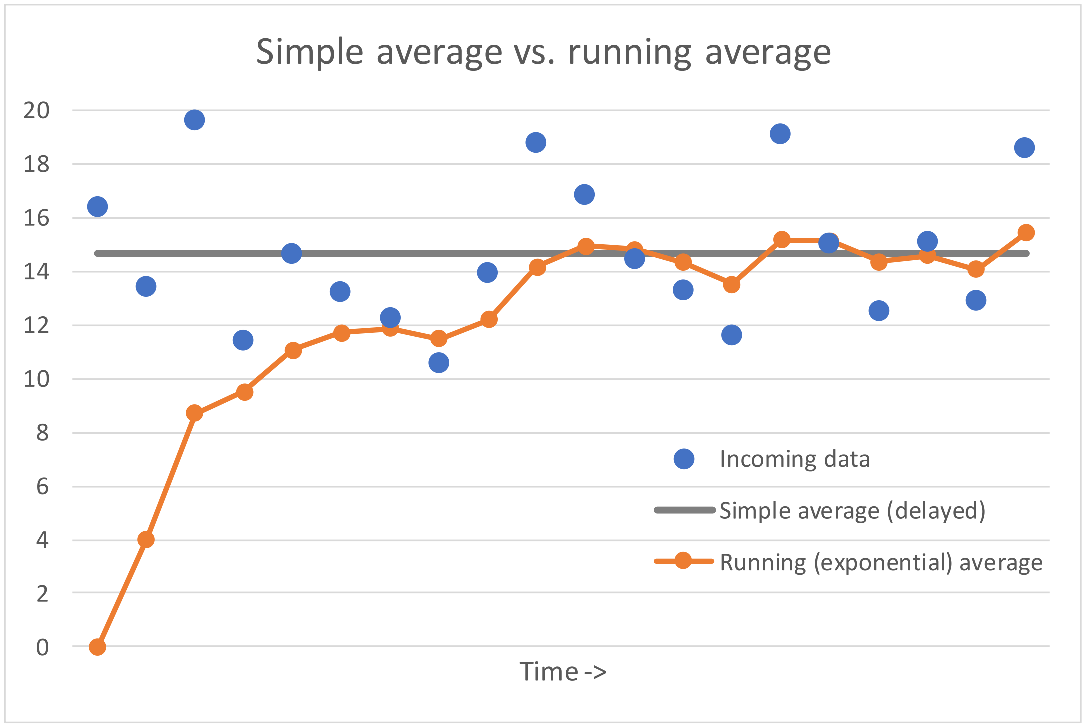
A is the running average

Exponentially weighted (discounted) sum of past data points

$$\frac{A}{\alpha} = T_{t-1} + \gamma T_{t-2} + \gamma^2 T_{t-3} + \gamma^3 T_{t-4} \dots$$

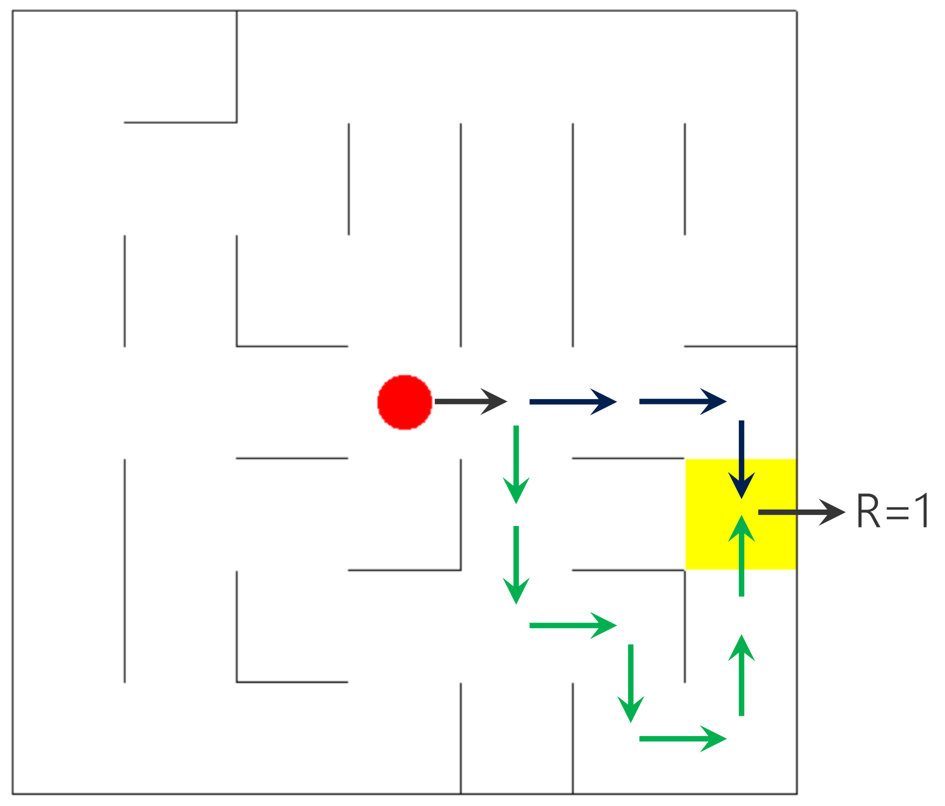
$$\gamma = 1 - \alpha$$

Discount factor gamma, less than 1.0



- Simple rule: On each step, move a small fraction of the distance toward the target. (Interpolation)
- Formula can be derived as minimization of MSE, same as linear regression for a single parameter.

How to rate the goodness of sequences of actions?

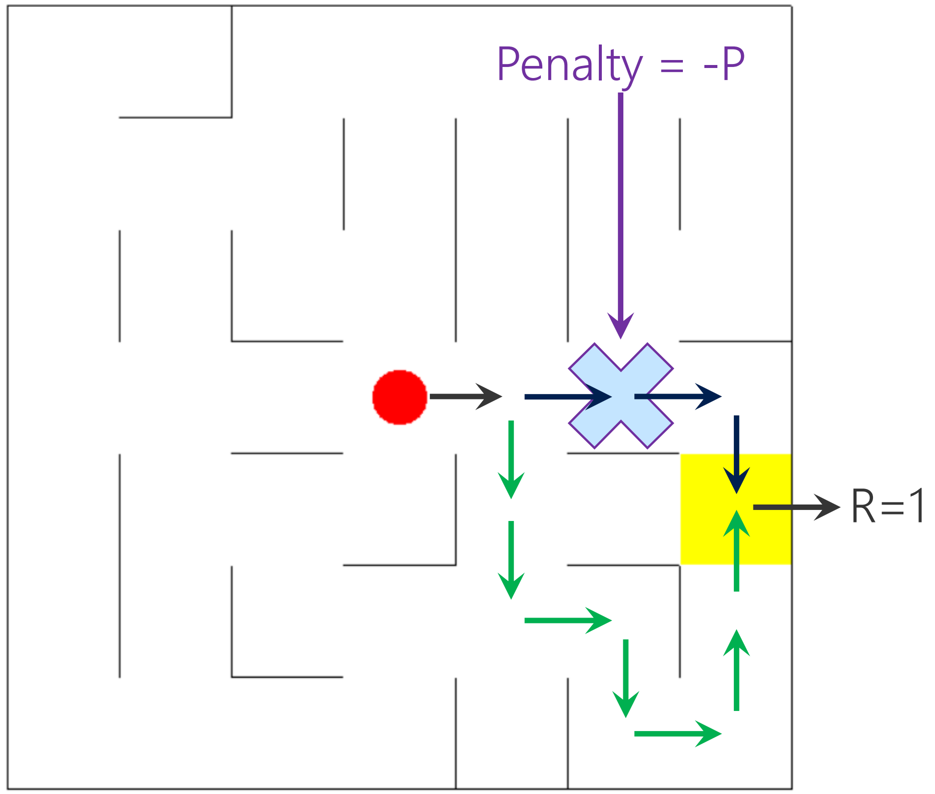


Which path is better? Why?

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
0	0	0	0	1				
0	0	0	0	0	0	0	0	1

We expect near-term rewards to count more than long-term rewards.

How to rate the goodness of sequences of actions?

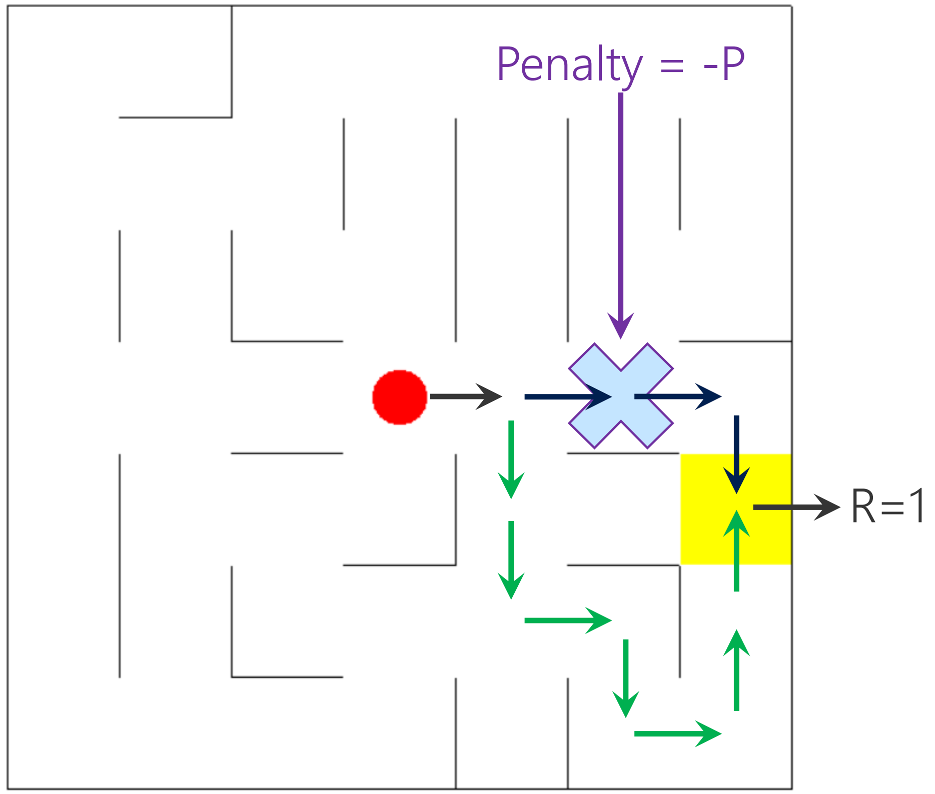


Now which path is better?

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
0	0	-P	0	1				
0	0	0	0	0	0	0	0	1

All the rewards received, positive or negative, need to be considered.

How to rate the goodness of sequences of actions?



R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
0	0	-P	0	1				
0	0	0	0	0	0	0	0	1

Cumulative Discounted Reward

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \ldots$$

Discount factor gamma, like 0.9

The **return** of the trajectory.

G stands for Gain, Goal, or Goodness

Cumulative discounted reward, from *T=t* $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \ldots$$

Cumulative discounted reward, from *T=t+1* $$G_{t+1} = R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \gamma^3 R_{t+5} \ldots$$

G defined recursively $$G_t = R_{t+1} + \gamma G_{t+1} \longleftarrow \text{The neatest trick in RL !!}$$

The goodness of a sequence of actions starting at **one point** in time, is equal to the **next reward** plus the **discounted** goodness of the sequence of actions starting at the **next point** in time.

V is an estimate of measurement *G*

The **Value Function** $\longrightarrow V(S_t) = \mathbb{E}[G_t]$

Starting in state *S* at time *t*

1-step look-ahead

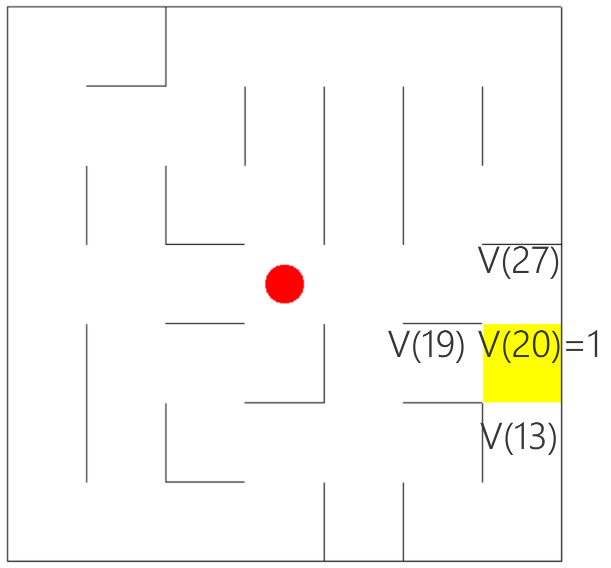
$$\underbrace{V(S_t)}_{\text{Old estimate}} \approx \underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{Newer, better estimate. } \textit{Use this as the target.}}$$

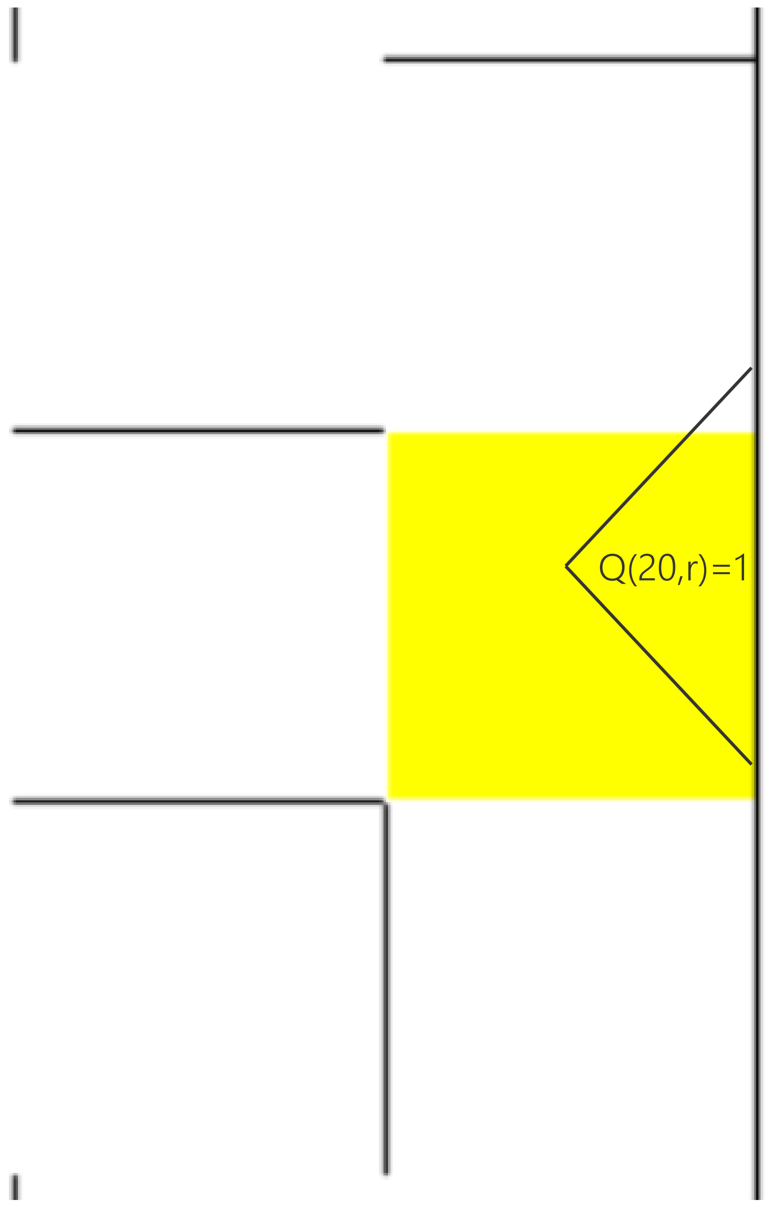
Formula for running average

$$A \leftarrow A + \alpha(T - A)$$
$$A = V(S_t)$$
$$T = R_{t+1} + \gamma V(S_{t+1})$$

$$V(S_t) \leftarrow V(S_t) + \alpha\Big(\overbrace{R_{t+1} + \gamma V(S_{t+1})} - V(S_t)\Big)$$

Temporal difference (TD) learning of the state-value function





The state value function *V(S)* doesn't provide enough information for the agent to select the best action.

But the action-value function *Q(S,A)* does!

The Q-value estimates the expected cumulative discounted reward after taking action *A* in state *S*.
Q stands for Quality

Temporal difference (TD) learning of the state-value function *V* *(One example of bootstrapping)*

$$V(S_t) \leftarrow V(S_t) + \alpha\big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big)$$

Temporal difference (TD) learning of the action-value function *Q* *(Also called Q-learning)*

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \underbrace{\max_a Q(S_{t+1}, a)}_{} - Q(S_t, A_t)\right)$$

Our policy (when not exploring) is to take the action with the highest Q-value at each step.

Pseudo-code for tabular Q-learning in a continuing RL task:

- Initialize table Q(s,a) arbitrarily. (But zeros work well.)
- Repeat for each step:
    - Choose A_t from S_t using a policy derived from Q (like epsilon-greedy)
    - Take action A_t, then observe R_{t+1} and S_{t+1}
    - Update element $Q(S_t,A_t)$ in the table according to the formula above. (The backup operation)
    - $S \leftarrow S_{t+1}$

3. Modify the agent to use Q-learning with epsilon-greedy exploration, according to the equation below. Set epsilon=0.1, learning rate alpha = 0.1, and discount factor gamma = 0.9. Reevaluate.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right)$$

- Edit Q_Learning\code\worker.py
  - Find the Worker.create_agent method.
  - Comment out the 2 RandomAgent lines.
  - Un-comment the 2 TabQAgent lines.
  - Save worker.py

- Open tab_q_agent.py
  - Compare TabQAgent.step to the pseudo-code for Q-learning.

- To train the tab_q_agent for 10,000 steps:
  - python demo.py --train
  - Mean reward per step = 0.0422
  - Peak reward/step of 0.14 is reached by 9000 steps

- View the final Q table.
- We aren't using a held-out test set. Is that a problem?

4. Tune these three hyper-parameters to maximize total reward over 10,000 steps.

- Find the hyper-parameters in the code.
  - epsilon
  - learning rate alpha
  - discount factor gamma

- Spend a few minutes adjusting these to improve total reward.

5. Evaluate your best set of hyper-parameters on the random seed provided by the instructor.

- Seed = 257257

- Fabulous prize for the best-performing agent.

Microsoft

Thank You!

Deep Learning World Workshop

James McCaffrey & Ricky Loynd
Microsoft Research