COMP9444 Neural Networks and Deep Learning Term 3, 2020

Solutions to Exercises 1: Perceptrons

This page was last updated: 09/21/2020 14:02:32

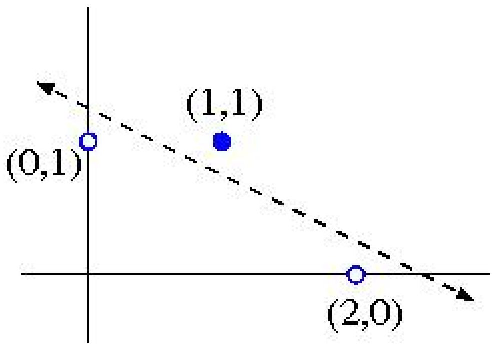
---

1. **Perceptron Learning**

   a. Construct by hand a Perceptron which correctly classifies the following data; use your knowledge of plane geometry to choose appropriate values for the weights w_0, w_1 and w_2.

Training Example	x_1	x_2	Class
a.	0	1	-1
b.	2	0	-1
c.	1	1	+1

The first step is to plot the data on a 2-D graph, and draw a line which separates the positive from the negative data points:



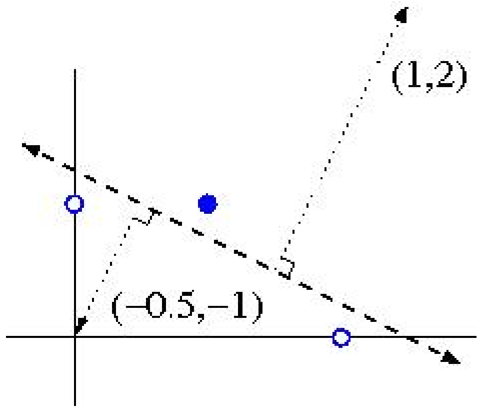
This line has slope -1/2 and x_2-intersect 5/4, so its equation is:

$x_2 = 5/4 - x_1/2$,
i.e. $2x_1 + 4x_2 - 5 = 0$.

Taking account of which side is positive, this corresponds to these weights:

$w_0 = -5$
$w_1 = 2$
$w_2 = 4$

Alternatively, we can derive weights $w_1=1$ and $w_2=2$ by drawing a vector normal to the separating line, in the direction pointing towards the positive data points:



The bias weight w_0 can then be found by computing the dot product of the normal vector with a perpendicular vector from the separating line to the origin. In this case $w_0 = 1(-0.5) + 2(-1) = -2.5$

(Note: these weights differ from the previous ones by a normalizing constant, which is fine for a Perceptron)

b. Demonstrate the Perceptron Learning Algorithm on the above data, using a learning rate of 1.0 and initial weight values of

$w_0 = -1.5$
$w_1 = \ \ 0$
$w_2 = \ \ 2$

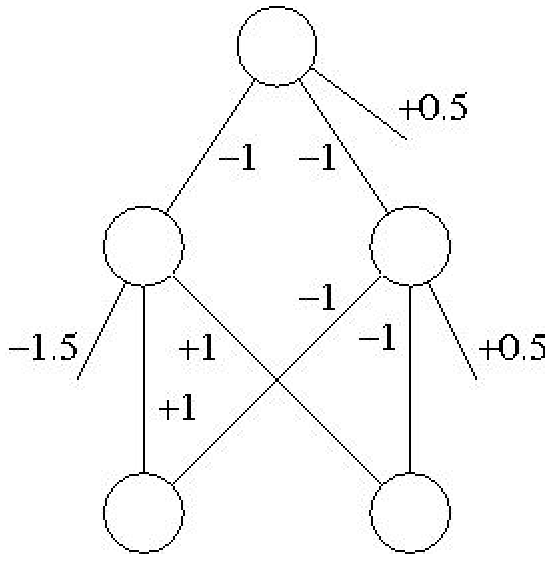
In your answer, you should clearly indicate the new weight values at the end of each training step.

Iteration	w_0	w_1	w_2	Training Example	x_1	x_2	Class	$s=w_0+w_1x_1+w_2x_2$	Action
1	-1.5	0	2	a.	0	1	-	+0.5	Subtract
2	-2.5	0	1	b.	2	0	-	-2.5	None
3	-2.5	0	1	c.	1	1	+	-1.5	Add
4	-1.5	1	2	a.	0	1	-	+0.5	Subtract
5	-2.5	1	1	b.	2	0	-	-0.5	None
6	-2.5	1	1	c.	1	1	+	-0.5	Add
7	-1.5	2	2	a.	0	1	-	+0.5	Subtract
8	-2.5	2	1	b.	2	0	-	+1.5	Subtract
9	-3.5	0	1	c.	1	1	+	-2.5	Add
10	-2.5	1	2	a.	0	1	-	-0.5	None
11	-2.5	1	2	b.	2	0	-	-0.5	None
12	-2.5	1	2	c.	1	1	+	+0.5	None

2. **XOR Network**

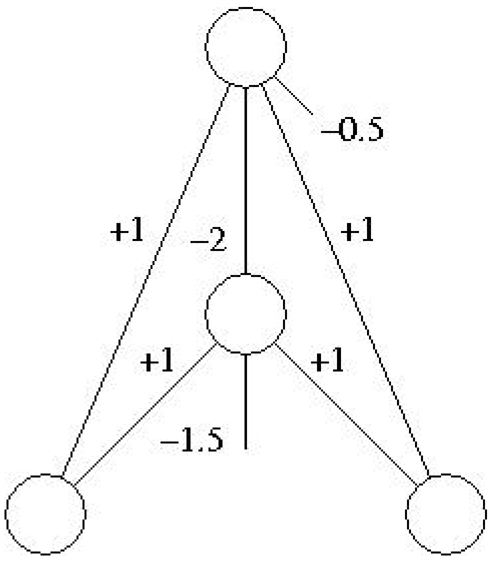
Construct by hand a Neural Network (or Multi-Layer Perceptron) that computes the XOR function of two inputs. Make sure the connections, weights and biases of your network are clearly visible.

There are a number of ways to express XOR as a combination of simpler functions that are linearly separable. For example, using NOR as an abbreviation for "NOT OR", (x_1 XOR x_2) can be written as (x_1 AND x_2) NOR (x_1 NOR x_2). This decomposition allows us to compute XOR with a network like this:

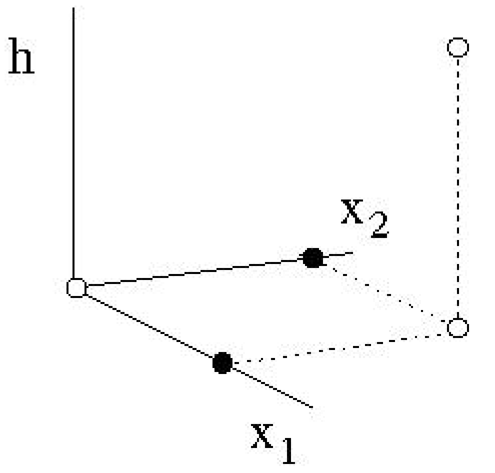


Challenge: Can you construct a Neural Network to compute XOR which has only one hidden unit, but also includes shortcut connections from the two inputs directly to the (one) output.
Hint: start with a network that computes the inclusive OR, and then try to think of how it could be modified.

Exclusive OR (XOR) is very similar to normal (inclusive) OR, except for the case where both inputs are True, i.e. where (x_1 AND x_2) is True. We therefore introduce a single hidden unit which computes (x_1 AND x_2). This hidden unit is connected to the output with a negative weight, thus forcing that case to be classified negatively.



The addition of this hidden "feature" creates a 3-dimensional space in which the points can be linearly separated by a plane. The weights for the output unit (+1,+1,-2) specify a vector perpendicular to the separating plane, and its distance from the origin is determined by the output bias divided by the length of this vector.



3. **Computing any Logical Function with a 2-layer Network**

   Assuming False=0 and True=1, explain how each of the following could be constructed:

   a. Perceptron to compute the OR function of *m* inputs

      Set the bias weight to -½, all other weights to 1.

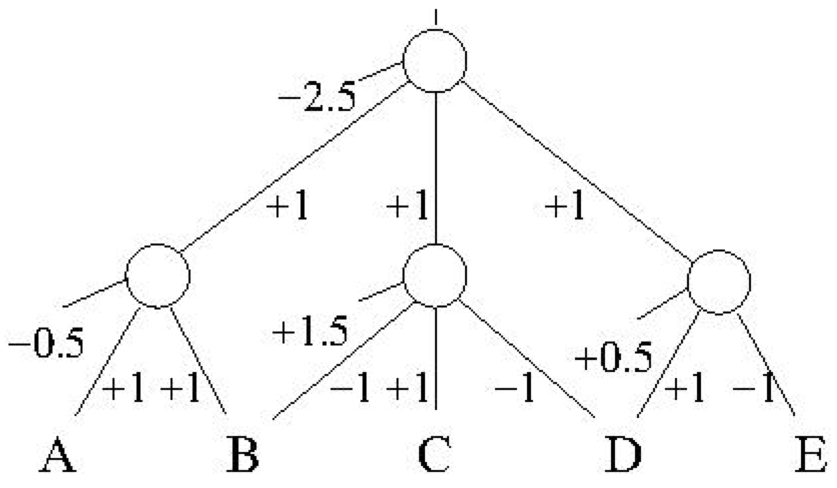
      It makes sense for the input-to-output weights to be 1, because any of the inputs being True makes it more likely for the output to be True. In fact, the ONLY way the output can be False is if ALL the inputs are False. By setting the bias to -½, we insure that the linear combination is slightly negative when all of the inputs are False, but becomes positive when any of the inputs is True.

   b. Perceptron to compute the AND function of *n* inputs

      Set the bias weight to (½ - *n*), all other weights to 1.

      The ONLY way the conjunction can be True is if ALL the inputs are True. By setting the bias to (½ - *n*), we insure that the linear combination is slightly positive when all of the inputs are True, but becomes negative when any of the inputs is False.

   c. 2-layer Neural Network to compute the function (A ∨ B) ∧ (¬ B ∨ C ∨ ¬ D) ∧ (D ∨ ¬ E)



      Each hidden node should compute one disjunctive term in the expression. The input-to-hidden weights are -1 for items that are negated, +1 for the others. The output node then computes the conjunction of all the hidden nodes, as in part 2.

   d. 2-Layer Neural Network to compute any (given) logical expression, assuming it is written in Conjunctive Normal Form.

As in the example above, each hidden node should compute one disjunctive term in the expression; the output node then computes the conjunction of all these hidden nodes. The input-to-hidden weights should be -1 for items that are negated, +1 for the others. The bias for each hidden node should be (k - ½) where k is the number of items that are negated in the disjunctive term corresponding to that node.

COMP9444 Neural Networks and Deep Learning Term 3, 2020

Solutions to Exercises 2: Backprop

This page was last updated: 09/22/2020 09:14:00

1. **Identical Inputs**

   Consider a degenerate case where the training set consists of just a single input, repeated 100 times. In 80 of the 100 cases, the target output value is 1; in the other 20, it is 0. What will a back-propagation neural network predict for this example, assuming that it has been trained and reaches a global optimum? (Hint: to find the global optimum, differentiate the error function and set to zero.)

   When sum-squared-error is minimized, we have

   $$\begin{aligned} E &= 80*(z-1)^2/2 + 20*(z-0)^2/2 \\ dE/dz &= 80*(z-1) + 20*(z-0) \\ &= 100*z - 80 \\ &= 0 \text{ when } z = 0.8 \end{aligned}$$

   When cross entropy is minimized, we have

   $$\begin{aligned} E &= -80*\log(z) - 20*\log(1-z) \\ dE/dz &= -80/z + 20/(1-z) \\ &= (-80*(1-z) + 20*z)/(z*(1-z)) \\ &= (100*z - 80)/(z*(1-z)) \\ &= 0 \text{ when } z = 0.8\text{, as before.} \end{aligned}$$

2. **Linear Transfer Functions**

   Suppose you had a neural network with linear transfer functions. That is, for each unit the activation is some constant *c* times the weighted sum of the inputs.

   a. Assume that the network has one hidden layer. We can write the weights from the input to the hidden layer as a matrix $\mathbf{W}^{HI}$, the weights from the hidden to output layer as $\mathbf{W}^{OH}$, and the bias at the hidden and output layer as vectors $\mathbf{b}^H$ and $\mathbf{b}^O$. Using matrix notation, write down equations for the value **O** of the units in the output layer as a function of these weights and biases, and the input **I**. Show that, for any given assignment of values to these weights and biases, there is a simpler network with no hidden layer that computes the same function.

      Using vector and matrix multiplication, the hidden activations can be written as

      $$\mathbf{H} = c * (\mathbf{b}^H + \mathbf{W}^{HI} * \mathbf{I})$$

      The output activations can be written as

      $$\begin{aligned} \mathbf{O} &= c * [\mathbf{b}^O + \mathbf{W}^{OH} * \mathbf{H}] \\ &= c * [\mathbf{b}^O + \mathbf{W}^{OH} * c * (\mathbf{b}^H + \mathbf{W}^{HI} * \mathbf{I})] \\ &= c * [(\mathbf{b}^O + \mathbf{W}^{OH} * c * \mathbf{b}^H) + (\mathbf{W}^{OH} * c * \mathbf{W}^{HI}) * \mathbf{I}] \end{aligned}$$

      Because of the associativity of matrix multiplication, this can be written as

$\mathbf{O} = c * (\mathbf{b}^{OI} + \mathbf{W}^{OI} * \mathbf{I})$

where

$$\mathbf{b}^{OI} = \mathbf{b}^{O} + \mathbf{W}^{OH} * c * \mathbf{b}^{H}$$
$$\mathbf{W}^{OI} = \mathbf{W}^{OH} * c * \mathbf{W}^{HI}$$

Therefore, the same function can be computed with a simpler network, with no hidden layer, using the weights $\mathbf{W}^{OI}$ and bias $\mathbf{b}^{OI}$.

b. Repeat the calculation in part (a), this time for a network with any number of hidden layers. What can you say about the usefulness of linear transfer functions?

By removing the layers one at a time as above, a simpler network with no hidden layer can be constructed which computes exactly the same function as the original multi-layer network. In other words, with linear activation functions, you don't get any benefit from having more than one layer.

COMP9444 Neural Networks and Deep Learning Term 3, 2020

Solutions to Exercises 3: Probability

This page was last updated: 09/30/2020 07:13:27

1. **Bayes' Rule**

   One bag contains 2 red balls and 3 white balls. Another bag contains 3 red balls and 2 green balls. One of these bags is chosen at random, and two balls are drawn randomly from that bag, without replacement. Both of the balls turn out to be red. What is the probability that the first bag is the one that was chosen?

   Let B = first bag is chosen, R = both balls are red. Then

   $P(R \mid B) = (2/5)*(1/4) = 1/10$

   $P(R \mid \neg B) = (3/5)*(2/4) = 3/10$

   $P(R) = (1/2)*(1/10) + (1/2)*(3/10) = 1/5$

   $P(B \mid R) = P(R|B)*P(B) / P(R) = (1/10)*(1/2)/(1/5) = 1/4$

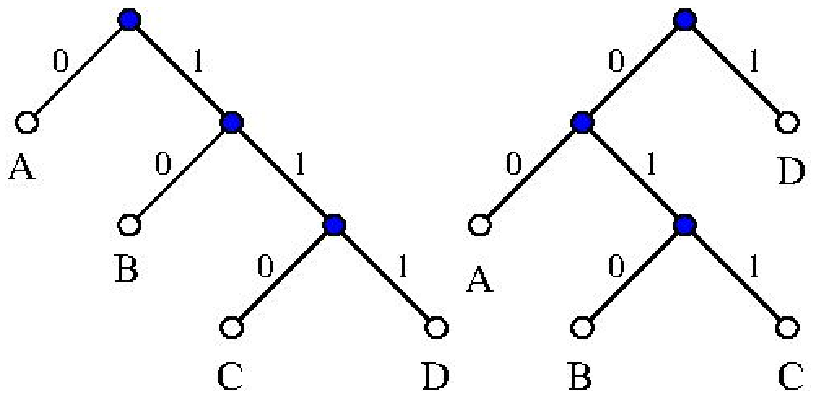
2. **Entropy and Kullback-Leibler Divergence**

   Consider these two probability distributions on the same space Ω = {A, B, C, D}

   $p = \langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8} \rangle$

   $q = \langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2} \rangle$

   a. Construct a Huffmann tree for each distribution *p* and *q*



   b. Compute the entropy H(*p*)

   $$H(p) = H(q) = \tfrac{1}{2}(-\log \tfrac{1}{2}) + \tfrac{1}{4}(-\log \tfrac{1}{4}) + \tfrac{1}{8}(-\log \tfrac{1}{8}) + \tfrac{1}{8}(-\log \tfrac{1}{8})$$
   $$= \tfrac{1}{2}(1) + \tfrac{1}{4}(2) + \tfrac{1}{8}(3) + \tfrac{1}{8}(3) = 1.75$$

   c. Compute the KL-Divergence in each direction $D_{KL}(p \,||\, q)$ and $D_{KL}(q \,||\, p)$

   $$D_{KL}(p \,||\, q) = \tfrac{1}{2}(2-1) + \tfrac{1}{4}(3-2) + \tfrac{1}{8}(3-3) + \tfrac{1}{8}(1-3) = 0.5$$

   $$D_{KL}(q \,||\, p) = \tfrac{1}{4}(1-2) + \tfrac{1}{8}(2-3) + \tfrac{1}{8}(3-3) + \tfrac{1}{2}(3-1) = 0.625$$

   Which one is larger? Why?

$D_{KL}(q \parallel p)$ is larger, mainly because the frequency of D has increased from ⅛ to ½, so it incurs a cost of 3-1=2 additional bits every time it occurs (which is often).

COMP9444 Neural Networks and Deep Learning
Term 3, 2020

Solutions to Exercises 4: PyTorch

This page was last updated: 09/30/2020 07:14:53

Download the zip file `Ex4code.zip` and unzip it. This will create a directory `Ex4code` with two very simple PyTorch programs, `slope.py` and `xor.py`.

---

1. **Adjusting the Learning Rate and Momentum**

   The program `slope.py` solves the simplest possible machine learning task:

   solve `F(x) = A*x` such that `F(1)=1`

   a. Run the program by typing

   ```
   python3 slope.py --lr 0.1
   ```

   Try running the code using each of the following values for learning rate:

   ```
   0.01, 0.1, 0.5, 1.0, 1.5, 1.9, 2.0, 2.1
   ```

   Describe what happens in each case, in terms of the success and speed of the algorithm.

   ```
   0.01  the task is learned in 998 epochs
   0.1   the task is learned in  97 epochs
   0.5   the task is learned in  16 epochs
   1.0   the task is learned in   2 epoch
   1.5   the task is learned in  16 epochs
   1.9   the task is learned in  97 epochs
   2.0   the parameter oscillates between 0.0 and 2.0
   2.1   the parameter explodes, first to inf and then to nan
   ```

   b. Now add momentum, by typing:

   ```
   python3 slope.py --mom 0.1
   ```

   Try running the code with learning rate kept at the default value of `1.9` but with momentum equal to `0.1, 0.2`, etc. up to `0.9`. For which value of momentum is the task solved in the fewest epochs?

   The fewest is `13` epochs, when momentum is `0.3`.

   What happens when the momentum is `1.0`? What happens when it is `1.1`?

   When momentum is `1.0`, the Parameter cycles around and never converges. When momentum is `1.1`, the Parameter blows up to Infinity.

2. **Learning the XOR Task**

   The program `xor.py` trains a 2-layer neural network on the XOR task.

   a. Run the code ten times by typing

   ```
   python3 xor.py
   ```

   For how many runs does it reach the global minimum? For how many runs does it reach a local minimum?

It reaches the global minimum in approximately 50% of runs; it gets stuck in a local minimum for the remaining 50%.

b. Keeping the learning rate fixed at `0.1`, can you find values of momentum (`--mom`) and initial weight size (`--init`) for which the code converges relatively quickly to the global minimum on virtually every run?

We have found that these hyperparameters successfully reach the global minimum in 99% of runs:

```
python3 xor.py --mom 0.9 --init 0.01
```

COMP9444 Neural Networks and Deep Learning
Term 2, 2020

Solutions to Exercise 5: Hidden Units and Convolution

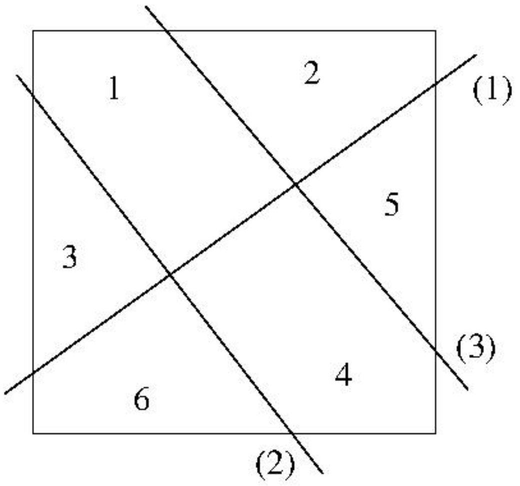
This page was last updated: 10/09/2020 11:24:04

1. **Hidden Unit Geometry**

   Consider a fully connected feedforward neural network with 6 inputs, 2 hidden units and 3 outputs, using tanh activation at the hidden units and sigmoid at the outputs. Suppose this network is trained on the following data, and that the training is successful.

```
Item  Inputs  Outputs
----  ------  -------
      123456    123
----  ------  -------
 1.   100000    000
 2.   010000    001
 3.   001000    010
 4.   000100    100
 5.   000010    101
 6.   000001    110
```

   Draw a diagram showing:
   a. for each input, a point in hidden unit space corresponding to that input, and
   b. for each output, a line dividing the hidden unit space into regions for which the value of that output is greater/less than one half.



2. **Softmax**

   Recall that the formula for Softmax is

   Prob(i) = $\exp(z_i) / \Sigma_j \exp(z_j)$

   Consider a classification task with three classes 1, 2, 3. Suppose a particular input is presented, producing outputs

   z_1=1.0, z_2=2.0, z_3=3.0

   and that the correct class for this input is Class 2. Compute the following, to two decimal places:
   a. Prob(i), for i = 1, 2, 3

Prob(1) = $e^1/(e^1 + e^2 + e^3)$ = 2.718/30.193 = 0.09
Prob(2) = $e^2/(e^1 + e^2 + e^3)$ = 7.389/30.193 = 0.24
Prob(3) = $e^3/(e^1 + e^2 + e^3)$ = 20.086/30.193 = 0.67

b. d(log Prob(2))/dz_j, for j = 1, 2, 3

d(log Prob(2))/dz_1 = d(z_2 - log Σ_j exp(z_j))/dz_1 = -exp(z_1)/Σ_j exp(z_j) = -0.09
d(log Prob(2))/dz_2 = d(z_2 - log Σ_j exp(z_j))/dz_2 = 1 - exp(z_2)/Σ_j exp(z_j) = 1 - 0.24 = 0.76
d(log Prob(2))/dz_3 = d(z_2 - log Σ_j exp(z_j))/dz_3 = -exp(z_3)/Σ_j exp(z_j) = -0.67

Note how the correct class (2) is pushed up, while the incorrect class with the highest activation (3) is pushed down the most.

3. **Convolutional Network Architecture**

One of the early papers on Deep Q-Learning for Atari games (Mnih et al, 2013) contains this description of its Convolutional Neural Network:

> "The input to the neural network consists of an 84 × 84 × 4 image. The first hidden layer convolves 16 8 × 8 filters with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 32 4 × 4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered."

For each layer in this network, compute the number of

a. weights per neuron in this layer (including bias)
b. neurons in this layer
c. connections into the neurons in this layer
d. independent parameters in this layer

You should assume the input images are gray-scale, there is no padding, and there are 18 valid actions (outputs).

First Convolutional Layer:

J = K = 84, L = 4, M = N = 8, P = 0, s = 4

weights per neuron:	1 + M × N × L = 1 + 8 × 8 × 4 = 257
width and height of layer:	1+(J-M)/s = 1+(84-8)/4 = 20
neurons in layer:	20 × 20 × 16 = 6400
connections:	20 × 20 × 16 × 257 = 1644800
independent parameters:	16 × 257 = 4112

Second Convolutional Layer:

J = K = 20, L = 16, M = N = 4, P = 0, s = 2

weights per neuron:	1 + M × N × L = 1 + 4 × 4 × 16 = 257
width and height of layer:	1+(J-M)/s = 1+(20-4)/2 = 9
neurons in layer:	9 × 9 × 32 = 2592
connections:	9 × 9 × 32 × 257 = 666144
independent parameters:	32 × 257 = 8224

Fully Connected Layer:

weights per neuron:	1 + 2592 = 2593
neurons in layer:	256
connections:	256 × 2593 = 663808
independent parameters:	663808

Output Layer:

weights per neuron:	1 + 256 = 257
neurons in layer:	18
connections:	18 × 257 = 4626
independent parameters:	4626

COMP9444 Neural Networks and Deep Learning Term 3, 2020

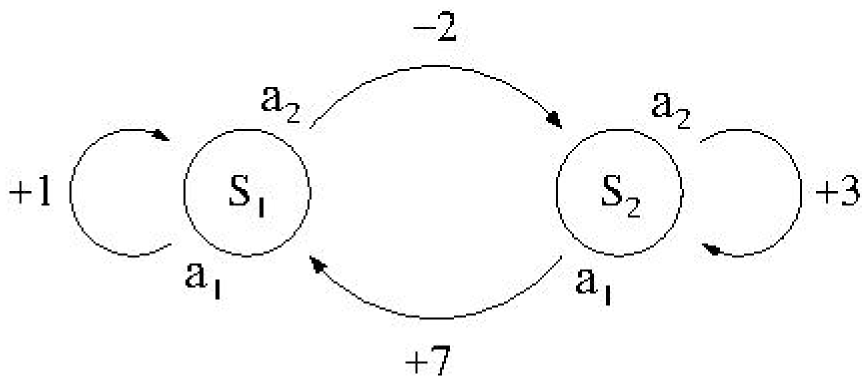
Solutions to Exercise 7: Reinforcement Learning

This page was last updated: 11/08/2020 06:07:09

Consider an environment with two states S = $\{S_1, S_2\}$ and two actions A = $\{a_1, a_2\}$, where the (deterministic) transitions δ and reward *R* for each state and action are as follows:

$\delta(S_1, a_1) = S_1, R(S_1, a_1) = +1$
$\delta(S_1, a_2) = S_2, R(S_1, a_2) = -2$
$\delta(S_2, a_1) = S_1, R(S_2, a_1) = +7$
$\delta(S_2, a_2) = S_2, R(S_2, a_2) = +3$

1. Draw a picture of this environment, using circles for the states and arrows for the transitions.



2. Assuming a discount factor of $\gamma = 0.7$, determine:
   a. the optimal policy $\pi^* : S \to A$

      $\pi^*(S_1) = a_2$
      $\pi^*(S_2) = a_1$

   b. the value function $V : S \to R$

      $V(S_1) = -2 + \gamma V(S_2)$
      $V(S_2) = +7 + \gamma V(S_1)$
      So $V(S_1) = -2 + 7\gamma + \gamma^2 V(S_1)$
      i.e. $V(S_1) = (-2 + 7\gamma)/(1 - \gamma^2) = (-2 + 7 \times 0.7)/(1 - 0.49) = 5.69$
      Then $V(S_2) = 7 + 0.7 \times 5.69 = 10.98$

   c. the "Q" function $Q : S \times A \to R$

      $Q(S_1, a_1) = 1 + \gamma V(S_1) = 4.98$
      $Q(S_1, a_2) = V(S_1) = 5.69$
      $Q(S_2, a_1) = V(S_2) = 10.98$
      $Q(S_2, a_2) = 3 + \gamma V(S_2) = 10.69$

   Writing the Q values in a matrix, we have:

Q	a_1	a_2
S_1	4.98	5.69
S_2	10.98	10.69

Trace through the first few steps of the Q-learning algorithm, with a learning rate of 1 and with all Q values initially set to zero. Explain why it is necessary to force exploration through probabilistic choice of actions, in order to ensure convergence to the true Q values.

With a deterministic environment and a learning rate of 1, the Q-Learning update rule is

$Q(S, a) \leftarrow r(S, a) + \gamma \max_b Q(\delta(S, a), b)$

Let's assume the agent starts in state S_1. Since the initial Q values are all zero, the first action must be chosen randomly. If action a_1 is chosen, the agent will get a reward of +1 and update
$Q(S_1,a_1) \leftarrow 1 + \gamma \times 0 = 1$

If we do not force exploration, the agent will always prefer action a_1 in state S_1, and will never explore action a_2. This means that $Q(S_1, a_2)$ will remain zero forever, instead of converging to the true value of 5.69 . If we do force exploration, the next steps may look like this:

current state	chosen action	new Q value
S_1	a_2	$-2 + \gamma*0 = -2$
S_2	a_2	$+3 + \gamma*0 = +3$

At this point, the table looks like this:

Q	a_1	a_2
S_1	1	-2
S_2	0	3

Again, we need to force exploration, in order to get the agent to choose a_1 from S_2, and to again choose a_2 from S_1

current state	chosen action	new Q value
S_2	a_1	$+7 + \gamma*1 = 7.7$
S_1	a_2	$-2 + \gamma*7.7 = 3.39$

Q	a_1	a_2
S_1	1	3.39
S_2	7.7	3

Further steps will refine the Q value estimates, and, in the limit, they will converge to their true values.

3. Now let's consider how the Value function changes as the discount factor γ varies between 0 and 1. There are four deterministic policies for this environment, which can be written as π_{11}, π_{12}, π_{21} and π_{22}, where $\pi_{ij}(S_1) = a_i$, $\pi_{ij}(S_2) = a_j$

a. Calculate the value function $V^{\pi}_{(\gamma)}: S \rightarrow R$ for each of these four policies (keeping γ as a variable)

$V^{\pi}_{11}(S_1) = +1 + \gamma V^{\pi}_{11}(S_1)$, so $V^{\pi}_{11}(S_1) = 1/(1 - \gamma)$
$V^{\pi}_{11}(S_2) = +7 + \gamma V^{\pi}_{11}(S_1) = 7 + \gamma/(1 - \gamma)$

$V^{\pi}_{12}(S_1) = V^{\pi}_{11}(S_1) = 1/(1 - \gamma)$
$V^{\pi}_{12}(S_2) = 3/(1 - \gamma)$

$V^{\pi}_{21}(S_1) = -2 + 7\gamma + \gamma^2 V^{\pi}_{21}(S_1)$, so $V^{\pi}_{21}(S_1) = (-2 + 7\gamma)/(1 - \gamma^2)$
$V^{\pi}_{21}(S_2) = +7 - 2\gamma + \gamma^2 V^{\pi}_{21}(S_2)$, so $V^{\pi}_{21}(S_2) = (7 - 2\gamma)/(1 - \gamma^2)$

$V^{\pi}_{22}(S_1) = -2 + 3\gamma/(1 - \gamma)$
$V^{\pi}_{22}(S_2) = 3/(1 - \gamma)$

b. Determine for which range of values of γ each of the policies $\pi_{11}, \pi_{12}, \pi_{21}, \pi_{22}$ is optimal

π_{11} is optimal when
$0 < V^{\pi}_{11}(S_1) - V^{\pi}_{21}(S_1) = ((1 + \gamma) - (-2 + 7\gamma))/(1 - \gamma^2) = (3 - 6\gamma)/(1 - \gamma^2)$, i.e. $0 \leq \gamma \leq 0.5$

π_{22} is optimal when
$0 < V^{\pi}_{22}(S_2) - V^{\pi}_{21}(S_2) = (3(1 + \gamma) - (7 - 2\gamma))/(1 - \gamma^2) = (-4 + 5\gamma)/(1 - \gamma^2)$, i.e. $0.8 \leq \gamma < 1.0$

π_{21} is optimal for $0.5 \leq \gamma \leq 0.8$

π_{12} is never optimal because it is dominated by π_{11} when $\gamma < 2/3$ and by π_{22} when $\gamma > 0.6$

COMP9444 Neural Networks and Deep Learning
Term 3, 2020

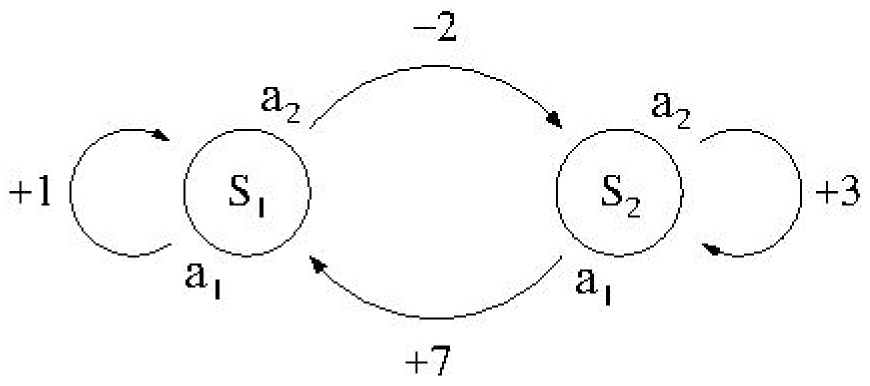
Solutions to Exercise 7: Reinforcement Learning

This page was last updated: 11/08/2020 06:07:09

Consider an environment with two states $S = \{S_1, S_2\}$ and two actions $A = \{a_1, a_2\}$, where the (deterministic) transitions δ and reward R for each state and action are as follows:

$$\delta(S_1, a_1) = S_1, R(S_1, a_1) = +1$$
$$\delta(S_1, a_2) = S_2, R(S_1, a_2) = -2$$
$$\delta(S_2, a_1) = S_1, R(S_2, a_1) = +7$$
$$\delta(S_2, a_2) = S_2, R(S_2, a_2) = +3$$

1. Draw a picture of this environment, using circles for the states and arrows for the transitions.



2. Assuming a discount factor of $\gamma = 0.7$, determine:
   a. the optimal policy $\pi^* : S \rightarrow A$

   $\pi^*(S_1) = a_2$
   $\pi^*(S_2) = a_1$

   b. the value function $V : S \rightarrow R$

   $V(S_1) = -2 + \gamma V(S_2)$
   $V(S_2) = +7 + \gamma V(S_1)$
   So $V(S_1) = -2 + 7\gamma + \gamma^2 V(S_1)$
   i.e. $V(S_1) = (-2 + 7\gamma)/(1 - \gamma^2) = (-2 + 7 \times 0.7)/(1 - 0.49) = 5.69$
   Then $V(S_2) = 7 + 0.7 \times 5.69 = 10.98$

   c. the "Q" function $Q : S \times A \rightarrow R$

   $Q(S_1, a_1) = 1 + \gamma V(S_1) = 4.98$
   $Q(S_1, a_2) = V(S_1) = 5.69$
   $Q(S_2, a_1) = V(S_2) = 10.98$
   $Q(S_2, a_2) = 3 + \gamma V(S_2) = 10.69$

Writing the Q values in a matrix, we have:

Q	a_1	a_2
S_1	4.98	5.69
S_2	10.98	10.69

Trace through the first few steps of the Q-learning algorithm, with a learning rate of 1 and with all Q values initially set to zero. Explain why it is necessary to force exploration through probabilistic choice of actions, in order to ensure convergence to the true Q values.

With a deterministic environment and a learning rate of 1, the Q-Learning update rule is

$Q(S, a) \leftarrow r(S, a) + \gamma \max_b Q(\delta(S, a), b)$

Let's assume the agent starts in state S_1. Since the initial Q values are all zero, the first action must be chosen randomly. If action a_1 is chosen, the agent will get a reward of +1 and update
$Q(S_1,a_1) \leftarrow 1 + \gamma \times 0 = 1$

If we do not force exploration, the agent will always prefer action a_1 in state S_1, and will never explore action a_2. This means that $Q(S_1, a_2)$ will remain zero forever, instead of converging to the true value of 5.69 . If we do force exploration, the next steps may look like this:

current state	chosen action	new Q value
S_1	a_2	$-2 + \gamma*0 = -2$
S_2	a_2	$+3 + \gamma*0 = +3$

At this point, the table looks like this:

Q	a_1	a_2
S_1	1	-2
S_2	0	3

Again, we need to force exploration, in order to get the agent to choose a_1 from S_2, and to again choose a_2 from S_1

current state	chosen action	new Q value
S_2	a_1	$+7 + \gamma*1 = 7.7$
S_1	a_2	$-2 + \gamma*7.7 = 3.39$

Q	a_1	a_2
S_1	1	3.39
S_2	7.7	3

Further steps will refine the Q value estimates, and, in the limit, they will converge to their true values.

3. Now let's consider how the Value function changes as the discount factor γ varies between 0 and 1. There are four deterministic policies for this environment, which can be written as π_{11}, π_{12}, π_{21} and π_{22}, where $\pi_{ij}(S_1) = a_i$, $\pi_{ij}(S_2) = a_j$

a. Calculate the value function $V^{\pi}_{(\gamma)}: S \rightarrow R$ for each of these four policies (keeping γ as a variable)

$V^{\pi}{}_{11}(S_1) = +1 + \gamma\, V^{\pi}{}_{11}(S_1)$, so $V^{\pi}{}_{11}(S_1) = 1/(1 - \gamma)$
$V^{\pi}{}_{11}(S_2) = +7 + \gamma\, V^{\pi}{}_{11}(S_1) = 7 + \gamma/(1 - \gamma)$

$V^{\pi}{}_{12}(S_1) = V^{\pi}{}_{11}(S_1) = 1/(1 - \gamma)$
$V^{\pi}{}_{12}(S_2) = 3/(1 - \gamma)$

$V^{\pi}{}_{21}(S_1) = -2 + 7\gamma + \gamma^2 V^{\pi}{}_{21}(S_1)$, so $V^{\pi}{}_{21}(S_1) = (-2 + 7\gamma)/(1 - \gamma^2)$
$V^{\pi}{}_{21}(S_2) = +7 - 2\gamma + \gamma^2 V^{\pi}{}_{21}(S_2)$, so $V^{\pi}{}_{21}(S_2) = (7 - 2\gamma)/(1 - \gamma^2)$

$V^{\pi}{}_{22}(S_1) = -2 + 3\gamma/(1 - \gamma)$
$V^{\pi}{}_{22}(S_2) = 3/(1 - \gamma)$

b. Determine for which range of values of γ each of the policies $\pi_{11}, \pi_{12}, \pi_{21}, \pi_{22}$ is optimal

π_{11} is optimal when
$0 < V^{\pi}{}_{11}(S_1) - V^{\pi}{}_{21}(S_1) = ((1 + \gamma) - (-2 + 7\gamma))/(1 - \gamma^2) = (3 - 6\gamma)/(1 - \gamma^2)$, i.e. $0 \le \gamma \le 0.5$

π_{22} is optimal when
$0 < V^{\pi}{}_{22}(S_2) - V^{\pi}{}_{21}(S_2) = (3(1 + \gamma) - (7 - 2\gamma))/(1 - \gamma^2) = (-4 + 5\gamma)/(1 - \gamma^2)$, i.e. $0.8 \le \gamma < 1.0$

π_{21} is optimal for $0.5 \le \gamma \le 0.8$

π_{12} is never optimal because it is dominated by π_{11} when $\gamma < 2/3$ and by π_{22} when $\gamma > 0.6$

COMP9444 Neural Networks and Deep Learning
Term 3, 2020

Solutions to Exercise 8: Hopfield Networks

This page was last updated: 11/10/2020 11:44:44

1.
   a. Compute the weight matrix for a Hopfield network with the two memory vectors $[1, -1, 1, -1, 1, 1]$ and $[1, 1, 1, -1, -1, -1]$ stored in it.

   The outer product W_1 of $[1, -1, 1, -1, 1, 1]$ with itself (but setting the diagonal entries to zero) is

$$\begin{matrix} 0 & -1 & 1 & -1 & 1 & 1 \\ -1 & 0 & -1 & 1 & -1 & -1 \\ 1 & -1 & 0 & -1 & 1 & 1 \\ -1 & 1 & -1 & 0 & -1 & -1 \\ 1 & -1 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & -1 & 1 & 0 \end{matrix}$$

   The outer product W_2 of $[1, 1, 1, -1, -1, -1]$ with itself (but setting the diagonal entries to zero) is

$$\begin{matrix} 0 & 1 & 1 & -1 & -1 & -1 \\ 1 & 0 & 1 & -1 & -1 & -1 \\ 1 & 1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & 0 & 1 & 1 \\ -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 & 0 \end{matrix}$$

   The weight matrix W is $(1/6)\times(W_1 + W_2) = (1/3)\times$

$$\begin{matrix} 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 & 0 \end{matrix}$$

   b. Confirm that both these vectors are stable states of this network.

$$\begin{aligned} \mathrm{sgn}(W.[1, -1, 1, -1, 1, 1]) &= \mathrm{sgn}((2/3)\times[1, -1, 1, -1, 1, 1]) \\ &= [1, -1, 1, -1, 1, 1] \end{aligned}$$

   so this one is stable. Similarly,

$$\begin{aligned} \mathrm{sgn}(W.[1, 1, 1, -1, -1, -1]) &= \mathrm{sgn}((2/3)\times[1, 1, 1, -1, -1, -1]) \\ &= [1, 1, 1, -1, -1, -1] \end{aligned}$$

   so this one is stable too.

2. Consider the following weight matrix W:

$$\begin{array}{rrrrr} 0.0 & -0.2 & 0.2 & -0.2 & -0.2 \\ -0.2 & 0.0 & -0.2 & 0.2 & 0.2 \\ 0.2 & -0.2 & 0.0 & -0.2 & -0.2 \\ -0.2 & 0.2 & -0.2 & 0.0 & 0.2 \\ -0.2 & 0.2 & -0.2 & 0.2 & 0.0 \end{array}$$

a. Starting in the state [1, 1, 1, 1, –1], compute the state flow to the stable state using asynchronous updates.

   W.[1, 1, 1, 1, –1] = [0, -0.4, 0, -0.4, 0]. Hence:
   If neuron 1, 3, or 5 updates first, its total net input is 0, so it does not change state;
   If neuron 2 updates first, its total net input is -0.4, and it's current value is +1, so it changes state to –1, and the new state is [1, –1, 1, 1, –1]. Call this Case A.
   If neuron 4 updates first, its total net input is -0.4, and it's current value is one, so it changes state to –1, and the new state is [1, 1, 1, –1, –1]. Call this Case B.

   Case A: W.[1, –1, 1, 1, –1] = [0.4, –0.4, 0.4, –0.8, –0.4]. Hence:
   If neurons 1, 2, 3, or 5 update first, there is no state change.
   If neuron 4 updates first, it flips, and the new state is [1, –1, 1, –1, –1].
   W.[1, –1, 1, –1, –1] = [0.8, –0.8, 0.8, –0.8, –0.8]. So no matter which neuron updates, there is no change. This is a stable state.

   Case B: W.[1, 1, 1, –1, –1] = [0.4, –0.8, 0.4, –0.4, –0.4]. Hence:
   If neurons 1, 3, 4 or 5 update first, there is no state change.
   If neuron 2 updates first, it flips, and the new state is [1, –1, 1, –1, –1].
   This is the same state as that reached in case A, and as seen in case A, it is a stable state.

b. Starting in the (same) state [1, 1, 1, 1, –1], compute the next state using synchronous updates.

   W.[1, 1, 1, 1, –1] = [0, -0.4, 0, -0.4, 0], so neurons 2 and 4 flip, resulting in a state of [1, –1, 1, –1, –1]. (We know from the previous part that this is a stable state.)