Computational Principles of Mobile Robotics

Deep Learning for Robots

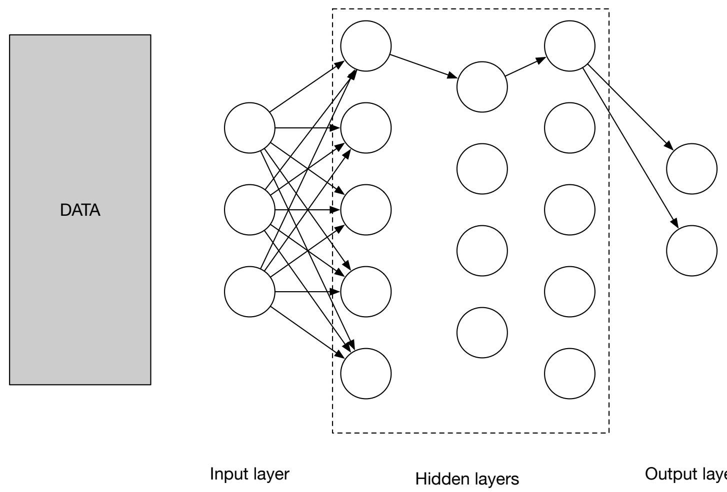
6.1 Learning-based methods

- Supervised learning. We have samples of the form (x,y). We learn a function that models y=h(x).
- Unsupervised learning. We are given sample answers y and we seek to infer some regularity over the set of y.
- Reinforcement learning. We wish to maximize the net benefit from taking a series of action that depend on one another using sequential data.
- Large language models. These models leverage the availability of extremely large text-based models.

6.2 Deep learning networks

- Prior to the 1980's there were very few applications of neural networks for autonomous systems.
  - Lack of large datasets for supervised learning approaches.
  - Lack of inexpensive massively parallelized hardware for training and operation.
- From 2000 on, these constraints have been overcome and now various deep networks are commonplace in autonomous systems.

6.3 Basic neural network structure

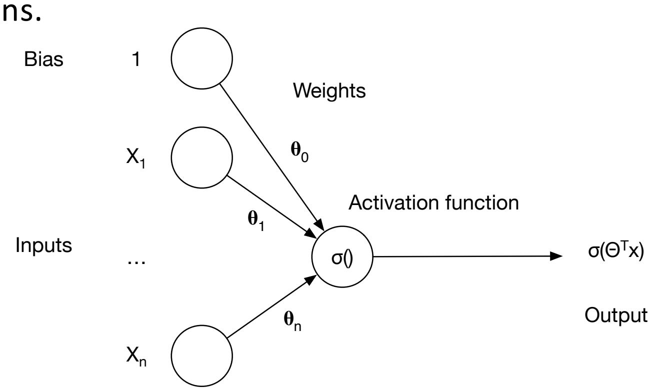


Many different architectures, some common features

- Directed collection of computational units (neurons) that perform a computation on their inputs and pass the information on.
- Organized in layers.
- For supervised networks
  - Networks are trained by providing a significant amount of data with corresponding results.
- Fundamental problems include
  - How to encode input and output
  - How to organize the computational units
  - How to train them

6.3.1 The Perceptron

- Basic (most simple) is a feed forward deep neural network comprised of perceptrons.



Deep learning structures

- Forward propagation
  - Can compute the output y of the neural network
    - $f(x; \Theta) = \sigma(\Theta_k^T\, \sigma(\Theta_{k-1}^T\, \sigma(\Theta_{k-2}^T\, \sigma(\, \dots \sigma(\Theta_1^T\, x) \dots)))$
  - Here $\sigma()$ is the activation function an k is the number of layers
- Goal is to find a set of weights that minimizes the error between the output of the network and the training data
- Typically choose a loss function like mean squared error or cross entropy
  - $\min \frac{1}{n} \sum_{i=1}^{n} l(f(x_i; \Theta), y_i)$

Solving for this

- Do gradient descent to adjust the weights in the net
  - $\Theta^{(t+1)} = \Theta^{(t)} - \gamma \nabla L(\Theta^{(t)})$
- Backpropagation
  - First adjust the last layer weights
  - Propagate error back to each previous layers
  - Adjust previous layer weights
- Note the constraints
  - Good data, differentiable loss function -> differentiable activation function

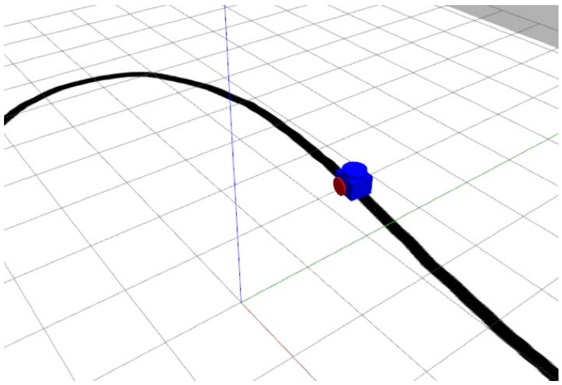
Rumelhart, Hinton & Williams (1986)

DNN (D is often omitted) or ANN (same for A)

- Extremely general approach.
- Has proven to be an extremely effective technique when you have a large amount of good data.
- Huge design space for any given problem.

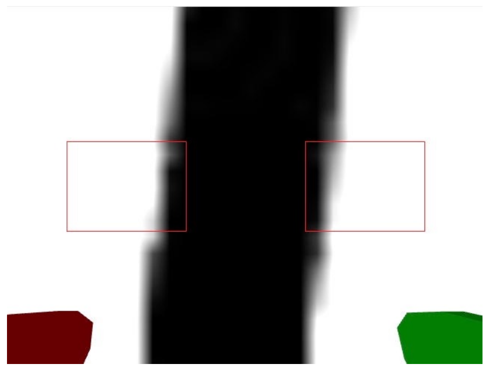
6.3.1 Following a line with a DNN

- Suppose you want to have a robot follow a line.



Line following

- We have a downward facing sensor that obtains average intensity over a window

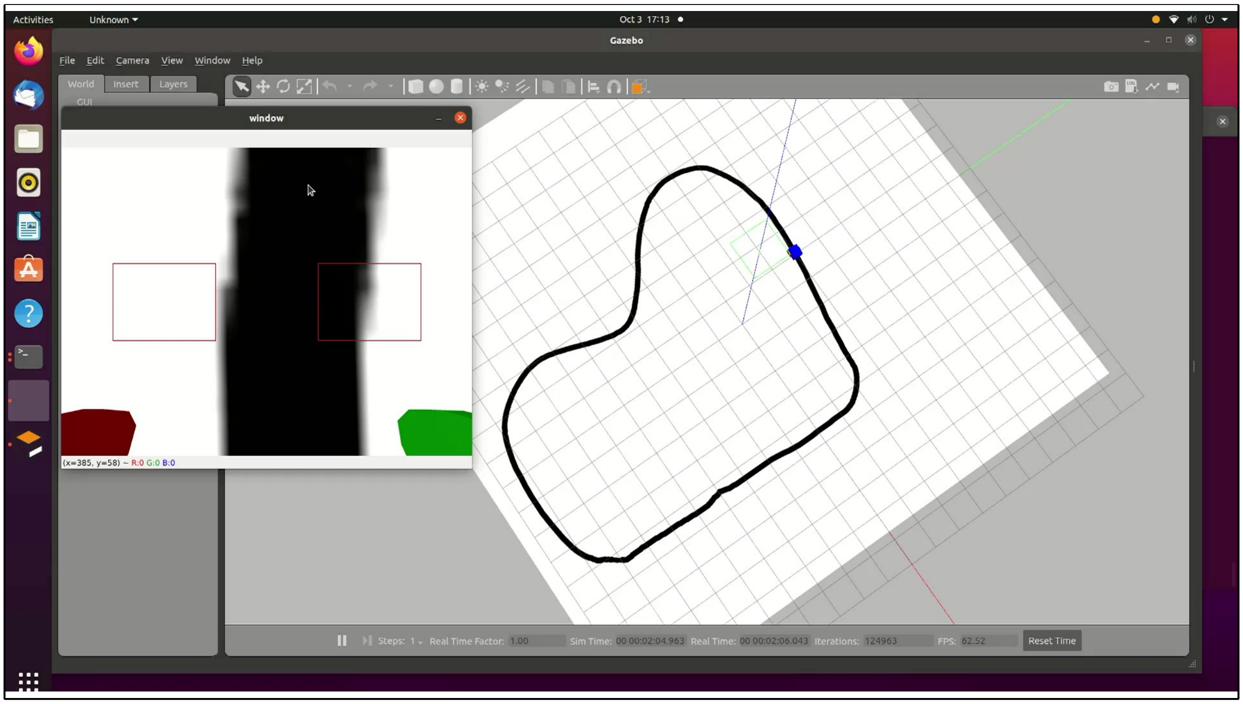


CPMR 3rd Edition - Chapter 6

11

Architecture

- Inputs (2) are fed into a stage using ReLU activation function
  - ReLU(x) is max(0, x)
- Followed by a levels of ReLU with 8 neurons
- Followed by a Softmax output stage
  - Softmax is $\sigma(y) = \frac{e_j^y}{\sum_{i=1}^k e_i^y}$
  - Softmax makes outputs sum to 1 and tends to choose one winner



Activities
Unknown
Oct 3 17:13
Gazebo
File
Edit
Camera
View
Window
Help
World
Insert
Layers
window
Steps:
Real Time Factor:
Sim Time:
Real Time:
Iterations:
FPS:
Reset Time

Classification

- Cross entropy function used as the loss function
- This was trained on about 8,000 inputs
- Took about 10 minutes to train on my machine at home.

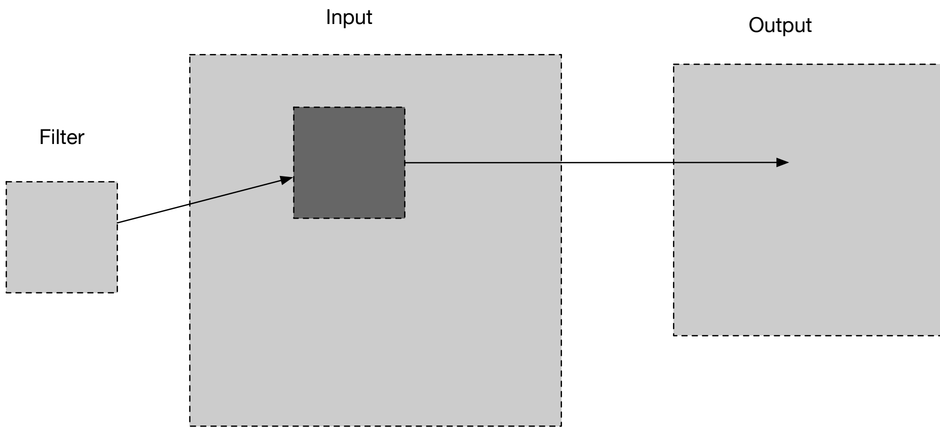
6.4 Convolutional neural networks (CNNs)

- Observe that NN’s treat each neuron completely separately.
- No ‘spatial relationship’ between neurons.
- Many data types have some relationship between adjacent points
  - Think images
- Convolutional neural nets are NN’s that are designed to be shift or space invariant.

CNN architectures

- Feed forward
- Layers
  - Input layer
  - Hidden layers
  - Output layers
- Layers can be of many types
  - Convolutional (perform convolutions)
  - Pooling (collapse data)
  - Fully connected

CNN Architecture: Convolutional layer



CNN architecture: Pooling

- Typically not learned
- Decrease resolution of the image by pooling (averaging, max, min) to reduce the dimensionality of the image.
- Output is the (average, max, min) under the window being used.

6.4.1 Road following with a CNN

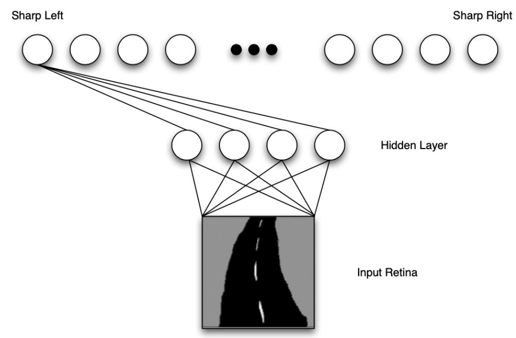
- CNN with about 27 million connections and 250K parameters
- Takes as input (rgb) three images (left, right, center)
- 9 layers
  - Normalization (not learned)
  - 5 convolutional layers
  - 3 fully connected layers
- Capable of autonomous driving about 98% of the time.

Bojarski et al. (2016)

Road following

- Much of the original work looked at tracking road features (lane markers).
- Almost all competitive algorithms now utilize CNN's.
- There exist large datasets to make training easier and to enable comparisons of solutions.
- Rather than looking at state of the art solutions, lets look at two points in the solution space
  - ALVINN
  - One we (you) can code yourself

Road following: ALVINN (early 1990's)



30x32 retina
4 hidden units
30 output units

960x4 + 4x32 weights

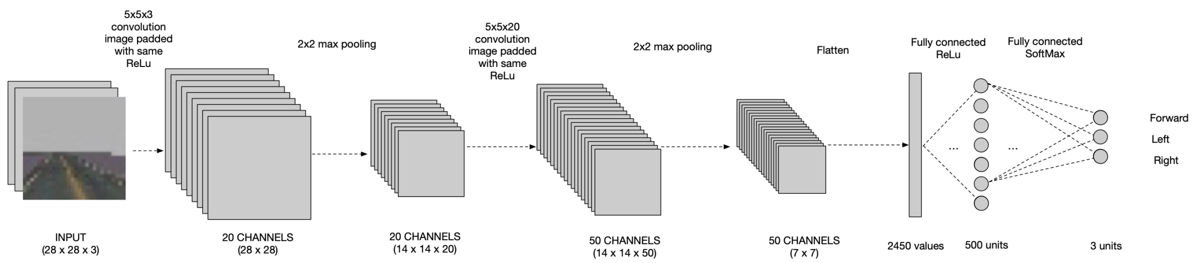
Pomerleau

Road following: now (toy example)

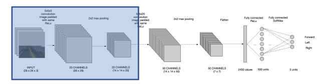
- We can use standard tools (e.g., tensorflow) to build simple CNN models that map inputs (images, typically downsampled) into steering commands.
- So lets do this
  - Collect data. Set of images (camera view, your turn angle)
    - L, R, F
  - Split the data into training data and testing data
    - We will drive on the robot later
- Now we need an architecture to describe the function
  - F(input-image)->{L, R, F}

Nawaz Ahmad tutorial for Raspberry Pi

Road following now: (toy example)

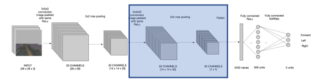


Road following now (Toy example)



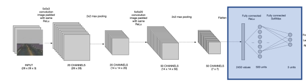
- Downsample the image (800,600,3) -> (28,28,3)
- Architecture level 1(CONV->RELU->POOL)
  - First a 2D convolution layer (5,5,3) output goes through reLu
    - Produces a number for each pixel in the image.
    - Each convolution as 5x5x3+1 (bias) = 76 parameters
    - And in parallel we will have 20 separate convolution channels
    - 76x20=1520 parameters
  - Now we downsample this by max pooling over a 2x2 window (take the max value over the outputs) so each output channel is now only 14x14.
  - Total is 20 ‘images’ each of size (14,14) or one image (14,14,20)

Road following now (Toy example)



- Architecture level 2 (CONV->RELU->POOL)
  - Input is 14x14x20 image
  - We convolve each image by a (5,5,20) filter with one bias value through reLu (501 parameters)
  - We will have 50 of them -> 25,050 parameters
  - Now again do a maxpooling using a (2,2) pool
    - Output image is 7x7x50

Road following now (Toy example)



- Architecture level 3 (Flatten->Dense->Dense->Softmax)
  - Input is 7x7x50 = 2450 values
  - Flatten into an array (2450)
  - Have a dense layer of 500 nodes
    - 2450x500+500=1,225,500 parameters
  - Have a dense layer of 3 nodes
    - 500x3+3 parameters = 1503
  - Softmax activation - no parameters
- Total architecture parameters
  - 1502+25050+1225500+1503=1,253,573 parameters

Road follow now (Toy example)

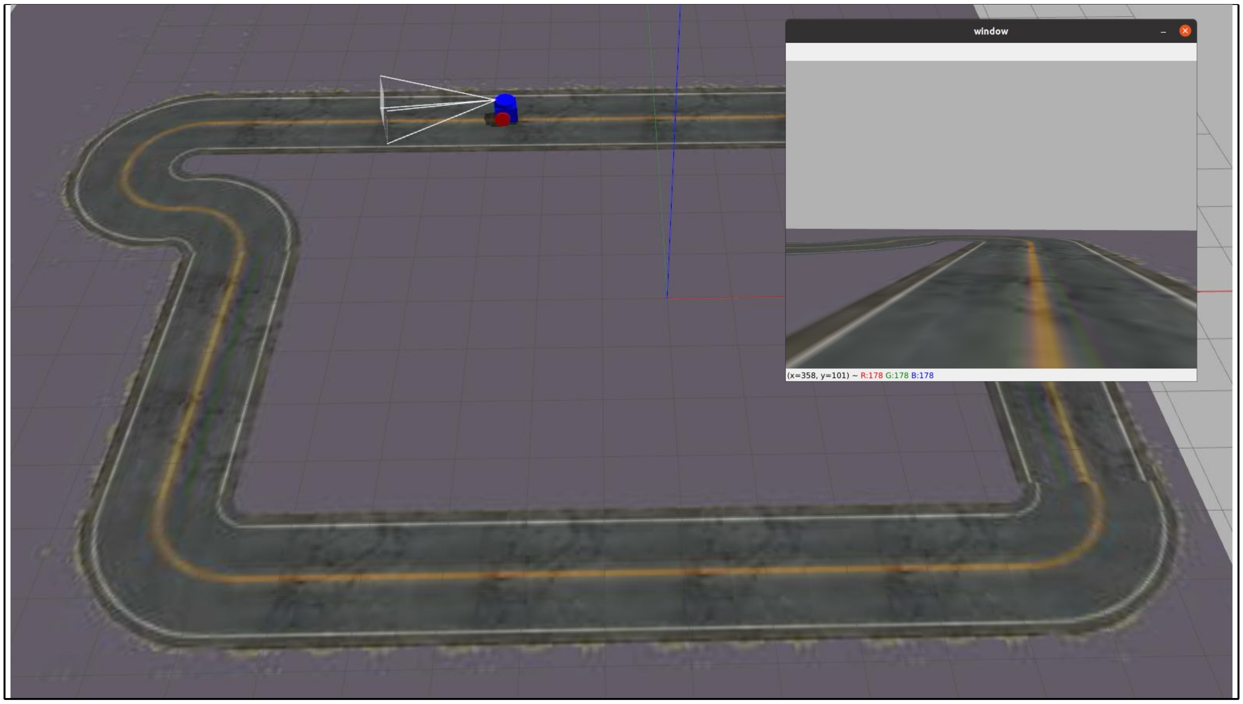
```
class LeNet:
  @staticmethod
  def build(width, height, depth, classes):
    # initialize the model
    model = Sequential()
    inputShape = (height, width, depth)
# first set of CONV => RELU => POOL layers
    model.add(Conv2D(20, (5, 5), padding="same",
      input_shape=inputShape))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# second set of CONV => RELU => POOL layers
    model.add(Conv2D(50, (5, 5), padding="same"))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# first (and only) set of FC => RELU layers
    model.add(Flatten())
    model.add(Dense(500))
    model.add(Activation("relu"))
# softmax classifier
    model.add(Dense(classes))
    model.add(Activation("softmax"))
# return the constructed network architecture
    return model
```

Road following now (Toy example)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 20)        1520
_________________________________________________________________
activation (Activation)      (None, 28, 28, 20)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 20)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 50)        25050
_________________________________________________________________
activation_1 (Activation)    (None, 14, 14, 50)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 50)          0
_________________________________________________________________
flatten (Flatten)            (None, 2450)              0
_________________________________________________________________
dense (Dense)                (None, 500)               1225500
_________________________________________________________________
activation_2 (Activation)    (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 1503
_________________________________________________________________
activation_3 (Activation)    (None, 3)                 0
=================================================================
Total params: 1,253,573
Trainable params: 1,253,573
```

Road following now (Toy example)

- We need data (lots of data).
- Have a simulator (Gazebo), with a simulated camera. We just need a road that we can drive along and capture
  - Raw image
  - What the driver (you) did when you saw the input
- Making more data
  - Data augmentation
- Dealing with anomalous situations
  - Collecting 'unusual' data.


window
(x=358, y=101) ~ R:178 G:178 B:178

6.5 Other topical architectures

- There exist a large number of different NN architectures that have been applied to NN, as well as other advanced AI approaches (e.g, Reinforcement Learning).
- Presence of standard implementations in libraries such as TF makes these architectures readily available.

6.6 Learning control

- We have built a very simple control system using CNN
- Obvious control is to replace the human input by machine output.
  - There exist other approaches. Should output be fed into a traditional control algorithm? Or be used directly?
- Key problem in control is that teaching by ‘normal’ behaviour is unlikely to experience unusual conditions.
  - How to provide such examples in training data?

6.7 Training regimes

- Sim2Real?
  - Lots of data, no problem with dangerous situations, but realism is an issue.
- Real world?
  - Where to collect the data? Can we build ‘good’ rather than ‘average’ systems?

6.8 Representing output features

- Just as encoding the data is important, so is an appropriate choice of output feature representation.
- One-Hot feature encoding is popular, but the use of a population to encode the distribution of the output may be more appropriate for some tasks.

6.9 Reinforcement learning

- Agent aims to learn a control policy π that optimizes the agent's long-term accumulated reward through interactions with the environment.
- Problem is modelled as a Markov Decision Process (MDP) characterized by
  - State space S
  - Action space A
  - Reward function r
  - State transition probability function P
- At each state s, agent selects and action a, receives a reward r(s,a) and transitions to a new state s'

Reinforcement learning

- Goal of the agent is to maximize the cumulative reward.
- In robotics, we are typically (but not always) interested in infinite interaction with the environment
  - Seek to maximize the discounted reward $\sum_{k=1}^{\infty} \gamma^k r_{t+k}$
- Agent executes a policy, we seek to find a policy that maximizes this reward through interactions with the environment.

Reinforcement learning

- A given policy is better than another if the expected return of the policy is at least as good as the other.
- That is q(s,a) >= q'(s,a)
- We can identify the optimal policy

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Reinforcement learning

- Bellman optimality equation

$$q_\pi(s, a) = E[R_{t+1} + \gamma \max_{a'}(s', a')].$$

- For any state-action pair (s,a) at time the expected return is the immediate reward plus a discounted version of the maximum reward we will get by taking an action from the new state s'.

6.9.1 Q-learning

- This suggests a very straightforward approach – represent this mapping as a table

$$Q'(s,a) = (1-\alpha)Q(s,a) + \alpha(R + \gamma \max_{a'} q(s',a'))$$

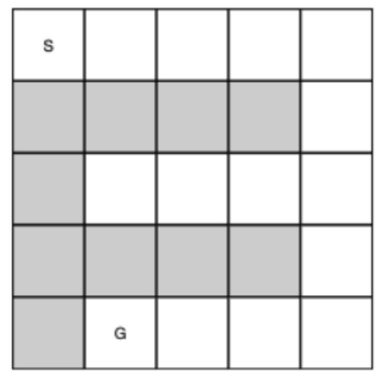
- When the agent takes an action a from state s going to state s', you update the Q value of this using the reward you get, plus the best q value for any action you could take from the new state.

Q-learning

```
Initialize Q(s,a)
for each episode do:
   Initialize state s
   for each step in the current episode do:
      a = select_action(Q, s)
      Take action a and observe reward R and next state s'
      Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
      s = s'
```

Q-learning

- Lets solve a simple maze by exploration
- Actions
  - Up, down, left right
- States
  - 25 possible locations
- Rewards
  - -100 if the agent hits a wall or tries to leave
  - -1 for each motion
  - +500 if it gets to the goal



Q-learning

Set all values to 0 q(25, 4)

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Q-learning

We will explore many times (100’s?)

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Q-learning

Each time we start at the same location s

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Q-learning

We explore until we get to the goal, or take more than N steps.

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Q-learning

We explore until we get to the goal, or take more than N steps.

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Selecting an action

- Two extremes
  - Choose the best action from the Q table (exploitation)
  - Choose a random action (exploration)
- Neither extreme is ideal
  - Common approach is known as epsilon greedy
    - Epsilon percentage of the time, explore randomly
    - 1-epsilon, explore in a greedy fashion

Q-learning

Take the action, get the reward and the next state

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Q-learning

Update Q

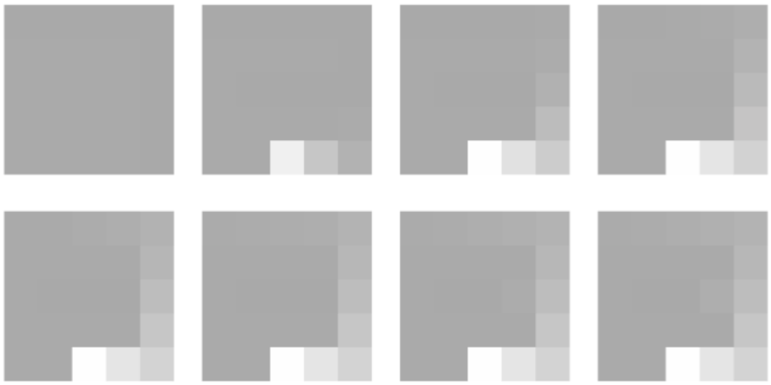
```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

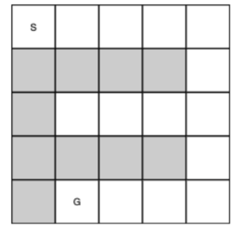
Q-learning

```
Initialize Q(s,a)
for each episode do:
    Initialize state s
    for each step in the current episode do:
        a = select_action(Q, s)
        Take action a and observe reward R and next state s'
        Q(s,a)=(1 - alpha) * Q(s,a) + alpha * (R + gamma * max(Q(s',a'))
        s = s'
```

Update state

Q-learning





Q-learning

- Some properties
  - Model free (no model of what is being solved)
  - Temporal difference (updates take place after each step)
  - Off-policy (estimates rewards base on greedy policy)
  - Exploitation versus exploration

6.9.2 Deep Q-learning

- Critical limitation of Q learning is the use of a table indexed by state and action
  - Explosion of table size with increasing complexity of both
  - Generalization to continuous values of state or action is difficult
- Deep Q network (and deep RL generally) addresses this by replacing the Q table with a deep neural net (DQN).
  - NN takes state (however encoded) -> Q values for every action
  - Update the application of the Bellman equation to take advantage of this network.

Deep Q-learning

- Critical issue
  - In DNN typically take a batch of data to train. Here we have individual interactions with the environment.
  - Stability over training runs
- Solutions
  - Replay buffer, batch/mini-batch updating
  - Separate target network

Deep RL has shown spectacular performance

- Atari Games
- Robot locomotion (walking)
- Network optimization
- Others

Large number of DRL algorithms now (Actor Critic, PPO, DQL, etc.)

6.10 Using large language models in robotics

- Underlying concept is to train a machine to predict the next word in a sequence.
- The large training set, complex model structure, and sophisticated training regime enables these machines to develop a deep representation of common sense knowledge.
- Can use this repeatedly to predict responses in a chatbot but other applications exist.
  - For example, which way to turn in a maze or how to move if an unexpected obstacle is encountered.
  - Large set of visual models (e.g., CLIP) which have started to replace traditional vision-based approaches to image understanding.