Introduction to Reinforcement Learning

A mini course @ HCMUS, Vietnam

Lectures 4-6

Long Tran-Thanh

Long.tran-thanh@warwick.ac.uk

Function approximation

From tabular to large-scale RL

So far:

- We have access to/can observe every single V(s) and Q(s,a)
- This is called tabular RL (we can store these values in, e.g., lookup tables)

Large-scale RL:

- State/action space is very large
- Backgammon: 10^20 states
- Computer Go: 10^170 states
- Helicopter flying: continuous state space

Issues of naïve application of tabular RL to large-scale RL:

- Lack of memory storage
- Cannot visit every state (or state-action pair)

Function approximation in RL

Idea:

- Think about V(s) and Q(s,a) as functions on s and (s,a), respectively
- Use some **function approximation** techniques to approximate these functions

$$\hat{v}(s, \mathbf{w}) \approx V^{\pi}(s)$$

$$\hat{q}(s, a, \mathbf{w}) \approx Q^{\pi}(s, a)$$

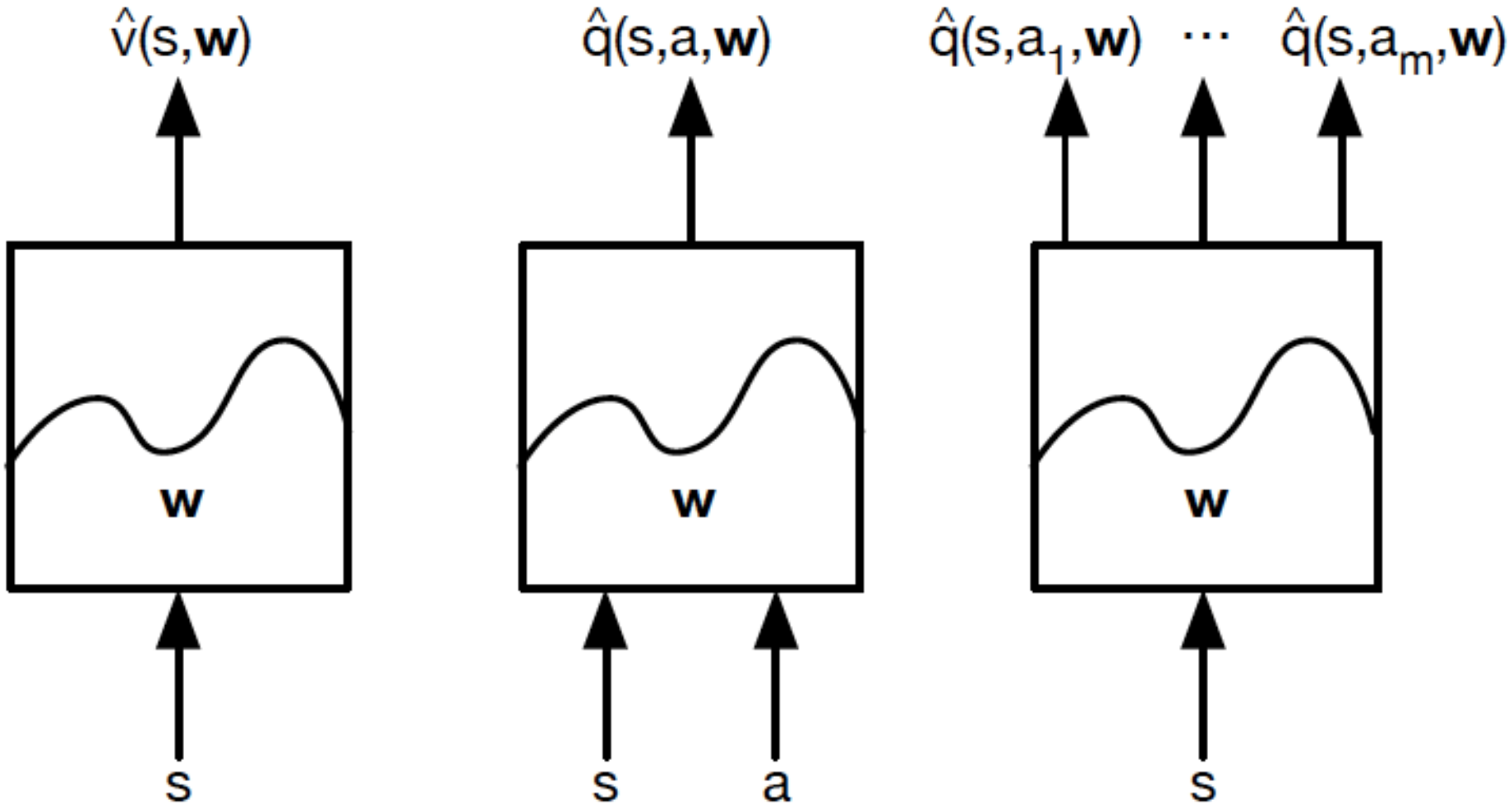
Advantages:

- Can approximate unseen states/state-action pairs
- Concise representations

Disadvantages:

- Decreased accuracy
- Need to learn the best representation (extra layer of complexity)

Function approximation in RL



Taken from David Silver's slides

Function approximation in RL

Types of function approximators:

- Linear combination of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- …

Function approximation in RL

Types of function approximators:

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- …

Function approximation in RL

Types of function approximators:

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- …

These are differentiable function approximators

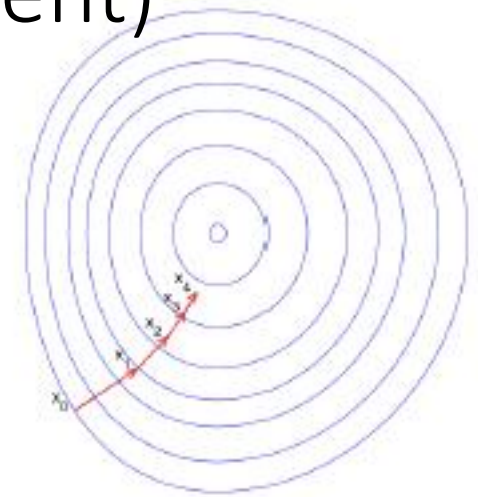
A bit of continuous optimisation (gradient descent)

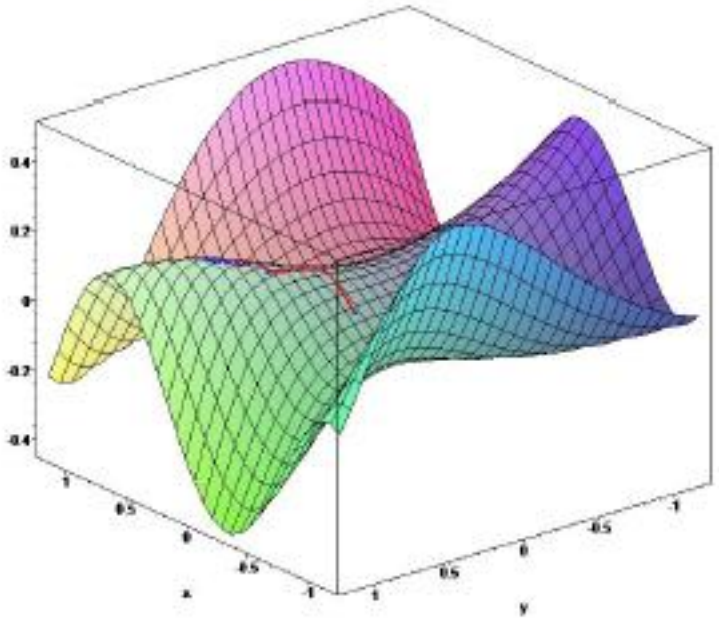
*J(**w**)*: a differentiable function of parameter vector ***w***
Goal: find local (global) minimum of *J(**w**)*

- Gradient of *J(**w**)*: $\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix}$

- Adjust **w** in the direction of the negative gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$





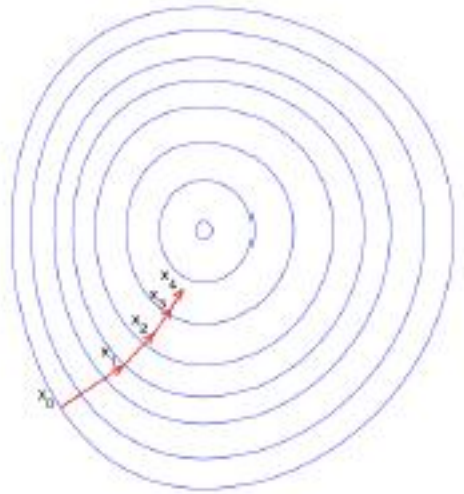
Taken from David Silver's slides

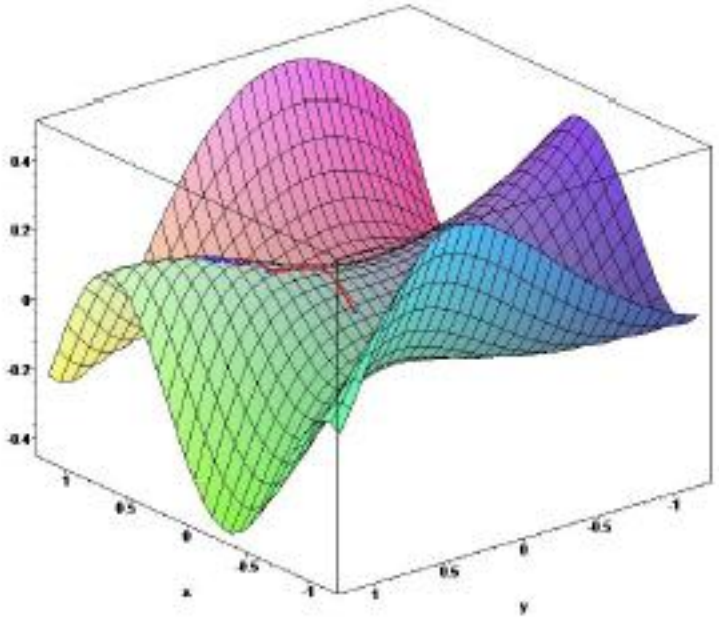
Value function approximation with SGD

Problem def: Given policy π and true value function $V^{\pi}(s)$

Goal: Find parameter vector ***w*** that minimises the following Mean-Squared Error (loss) function:

$$J(\mathbf{w}) = \mathbb{E}_{\pi}\left[(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))^2\right]$$





Taken from David Silver's slides

Value function approximation with SGD

Problem def: Given policy π and true value function $V^{\pi}(s)$

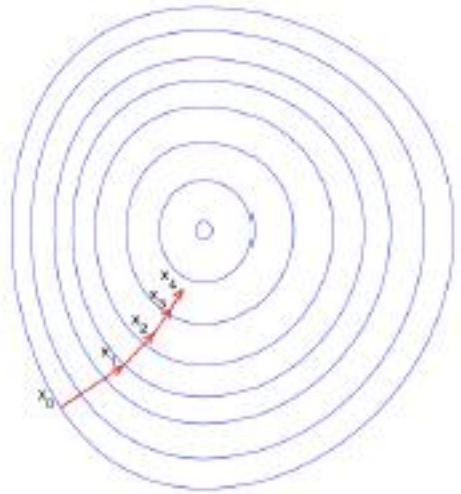
Goal: Find parameter vector ***w*** that minimises the following Mean-Squared Error (loss) function:

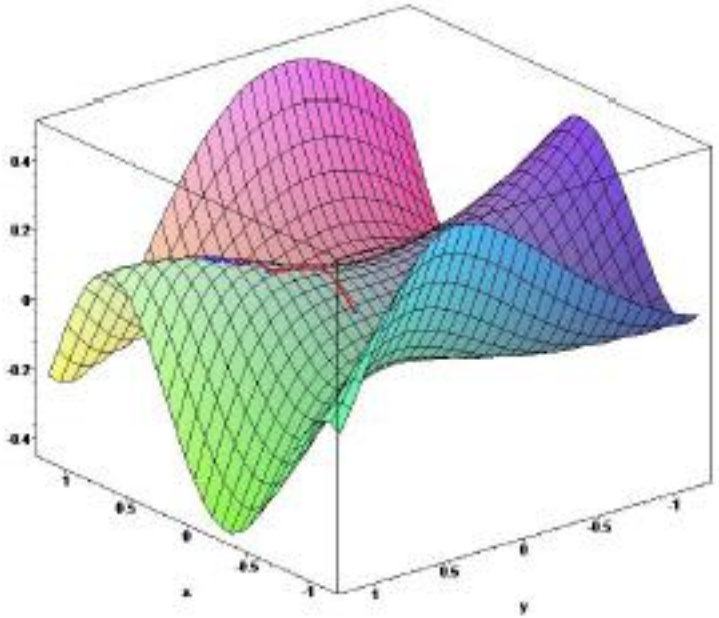
$$J(\mathbf{w}) = \mathbb{E}_{\pi}\left[(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))^2\right]$$

- Gradient descent algorithm:

$$\Delta \mathbf{w} = -\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) J(\mathbf{w}) = \alpha \mathbb{E}_{\pi}\left[(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))^2\right]$$

- Issue: we cannot have value of all *s* -> we sample some





Taken from David Silver's slides

Value function approximation with SGD

Problem def: Given policy π and true value function $V^{\pi}(s)$

Goal: Find parameter vector ***w*** that minimises the following Mean-Squared Error (loss) function:

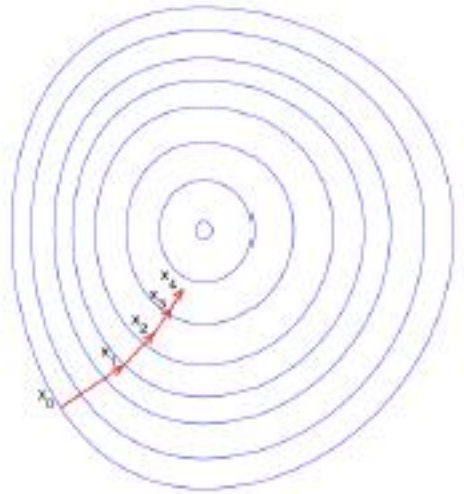
$$J(\mathbf{w}) = \mathbb{E}_{\pi}\left[(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))^2\right]$$

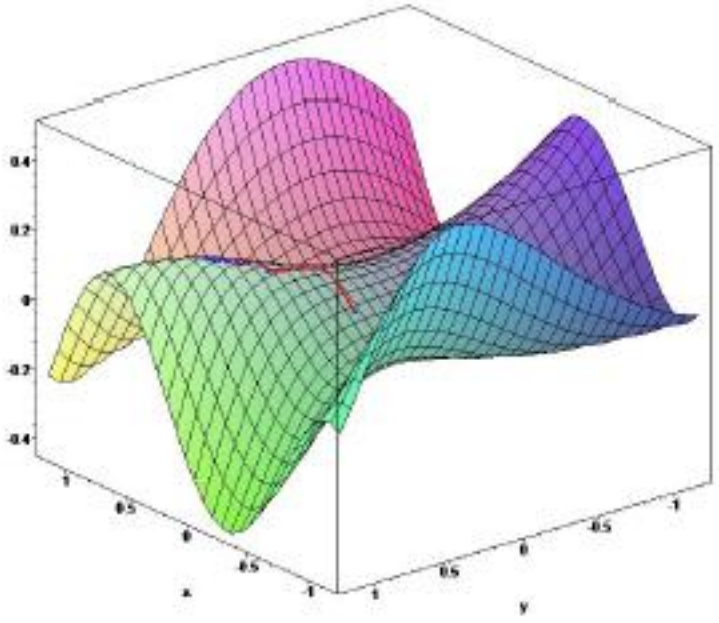
- Gradient descent algorithm:

$$\Delta \mathbf{w} = -\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha \mathbb{E}_{\pi}\left[(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})\right]$$

- Issue: we cannot have value of all *s* -> we sample some *s*
- Stochastic gradient descent (SGD):

$$\Delta \mathbf{w} = \alpha(V^{\pi}(s) - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})$$





Taken from David Silver's slides

Linear state value function approximation

Feature vector of each state *s*: $\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$

Linear function approximator: $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w}$

Linear state value function approximation

Feature vector of each state *s*: $\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$

Linear function approximator: $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w}$

Loss function: $J(\mathbf{w}) = \mathbb{E}\left[(V^\pi(s) - \mathbf{x}(s)^T \mathbf{w})^2\right]$

- Quadratic in **w**
- Stochastic gradient descent will converge to the global minimum
- SGD's update rule becomes very simple:

$$\Delta \mathbf{w} = \alpha \left(V^\pi(s) - \mathbf{x}(s)^T \mathbf{w}\right) \mathbf{x}(s)$$

step size | prediction error | feature vector

Linear state value function approximation

In an ideal world:

Loss function: $J(\mathbf{w}) = \mathbb{E}\left[(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w})^2\right]$

SGD's update rule : $\Delta\mathbf{w} = \alpha\left(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w}\right)\mathbf{x}(s)$

Linear state value function approximation

In an ideal world:
Loss function: $J(\mathbf{w}) = \mathbb{E}\left[(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w})^2\right]$
SGD's update rule: $\Delta\mathbf{w} = \alpha\left(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w}\right)\mathbf{x}(s)$

In real life, we don't know $V^{\pi}(s)$

Idea: use model-free approaches (MC, TD) to evaluate $V^{\pi}(s)$

Linear state value function approximation

In an ideal world:
Loss function: $J(\mathbf{w}) = \mathbb{E}\left[(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w})^2\right]$
SGD's update rule: $\Delta\mathbf{w} = \alpha\left(V^{\pi}(s) - \mathbf{x}(s)^T\mathbf{w}\right)\mathbf{x}(s)$

In real life, we don't know $V^{\pi}(s)$

Idea: use model-free approaches (MC, TD) to evaluate $V^{\pi}(s)$

Implementation: in the formulae above, replace $V^{\pi}(s)$ with:

- Total return G_t^{π} in case of MC
- TD target in case of TD(0) or TD(lambda)

Linear state value function approximation

MC method:

$$\Delta \mathbf{w} = \alpha \left(G_t^{\pi} - \mathbf{x}(s_t)^T \mathbf{w} \right) \mathbf{x}(s_t)$$

TD(0) learning:

$$\begin{aligned} \Delta \mathbf{w} &= \alpha \left(r_t + \gamma \mathbf{x}(s_{t+1})^T \mathbf{w} - \mathbf{x}(s_t)^T \mathbf{w} \right) \mathbf{x}(s_t) \\ &= \alpha \delta_t \mathbf{x}(s_t) \end{aligned}$$

TD(lambda):

- Forward-view:

$$\Delta \mathbf{w} = \alpha \left(G_t^{\lambda} - \mathbf{x}(s_t)^T \mathbf{w} \right) \mathbf{x}(s_t)$$

- Backward-view:

$$\begin{aligned} \delta_t &= r_t + \gamma \mathbf{x}(s_{t+1})^T \mathbf{w} - \mathbf{x}(s_t)^T \mathbf{w} \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(s_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

State-action value function approximation

So far we focused on approximating the value function of a given policy π

Now let's investigate how to find the optimal policy under the function approximation framework

State-action value function approximation

So far we focused on approximating the value function of a given policy π

Now let's investigate how to find the optimal policy under the function approximation framework

Recall that if we want to learn the optimal policy, we need to evaluate the value function of state-action pairs -> we need to approximate this function

$$\hat{q}(s, a, \mathbf{w}) \approx Q^{\pi}(s, a)$$

State-action value function approximation

So far we focused on approximating the value function of a given policy π

Now let's investigate how to find the optimal policy under the function approximation framework

Recall that if we want to learn the optimal policy, we need to evaluate the value function of state-action pairs -> we need to approximate this function

$$\hat{q}(s, a, \mathbf{w}) \approx Q^{\pi}(s, a)$$

How to do it?

- Use SGD to optimise **w** to minimse the MSE loss function
- Combine this with model-free approaches

Linear state-action value function approximation

We go straight to the linear function approximation case:

$$\mathbf{x}(s,a) = \begin{pmatrix} x_1(s,a) \\ \vdots \\ x_n(s,a) \end{pmatrix}$$

Linear state-action value function: $\hat{q}(s,a,\mathbf{w}) = \mathbf{x}(s,a)^T \mathbf{w}$

Stochastic gradient descent update (if we know the true q value):

$$\Delta \mathbf{w} = \alpha \left(Q^{\pi}(s,a) - \mathbf{x}(s,a)^T \mathbf{w} \right) \mathbf{x}(s,a)$$

Linear state-action value function approximation

In case we don’t know the true Q values:

- Use the model-free approaches to estimate these Q values

Linear state-action value function approximation

In case we don’t know the true Q values:

- Use the model-free approaches to estimate these Q values

MC-based: $\Delta \mathbf{w} = \alpha \left(G_t^{\pi} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$

Linear state-action value function approximation

In case we don't know the true Q values:

- Use the model-free approaches to estimate these Q values

MC-based: $\Delta \mathbf{w} = \alpha \left(G_t^{\pi} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$

TD-based (SARSA, Q-learning):

- TD(0): $\Delta \mathbf{w} = \alpha \left(r_t + \gamma \mathbf{x}(s_{t+1}, a_{t+1})^T \mathbf{w} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$

Linear state-action value function approximation

In case we don't know the true Q values:

- Use the model-free approaches to estimate these Q values

MC-based: $\Delta \mathbf{w} = \alpha \left(G_t^{\pi} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$

TD-based (SARSA, Q-learning):

- TD(0): $\Delta \mathbf{w} = \alpha \left(r_t + \gamma \mathbf{x}(s_{t+1}, a_{t+1})^T \mathbf{w} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$
- Forward-view TD(lambda): $\Delta \mathbf{w} = \alpha \left(G_t^{\lambda} - \mathbf{x}(s_t, a_t)^T \mathbf{w} \right) \mathbf{x}(s_t, a_t)$
- Backward-view TD(lambda):

$$\delta_t = r_t + \gamma \mathbf{x}(s_{t+1}, a_{t+1})^T \mathbf{w} - \mathbf{x}(s_t, a_t)^T \mathbf{w}$$
$$E_t = \gamma \lambda E_{t_1} + \mathbf{x}(s_t, a_t)$$
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Non-linear value function approximation

Linear function approximation:

- Simple model space
- Nice theoretical properties
- Efficient provably converging solutions (SGD)

Non-linear value function approximation

Linear function approximation:

- Simple model space
- Nice theoretical properties
- Efficient provably converging solutions (SGD)

But:

- Model misspecification (most applications are non-linear)

Can we do better with non-linear function approximation? How?

Non-linear value function approximation

Linear function approximation:

- Simple model space
- Nice theoretical properties
- Efficient provably converging solutions (SGD)

But:

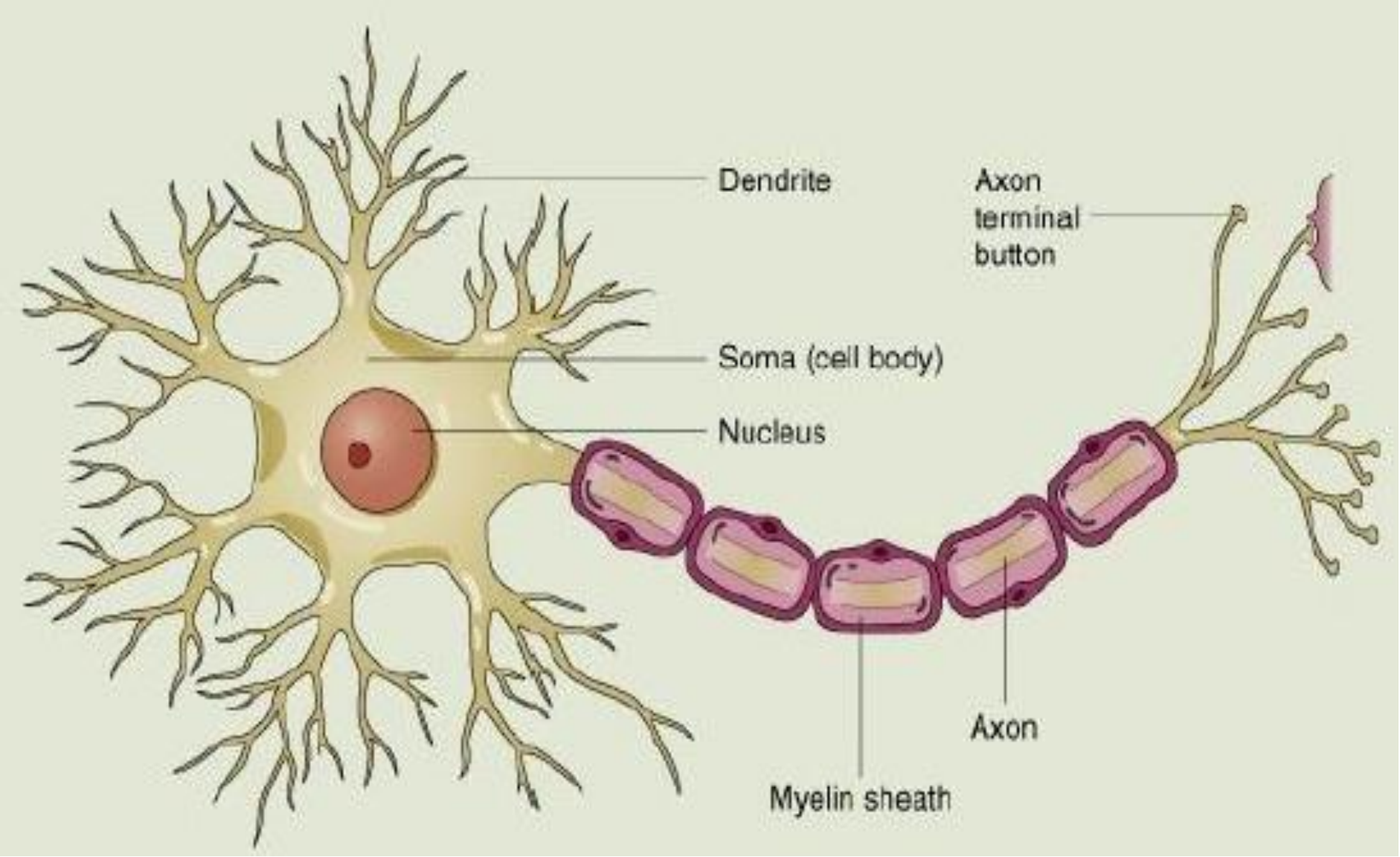
- Model misspecification (most applications are non-linear)

Can we do better with non-linear function approximation? How?

Idea: use deep neural nets to approximate the value functions

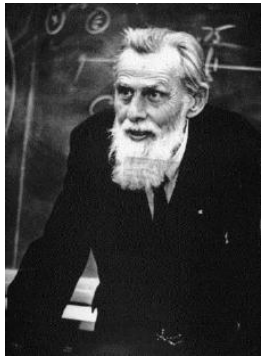
Deep Neural Nets

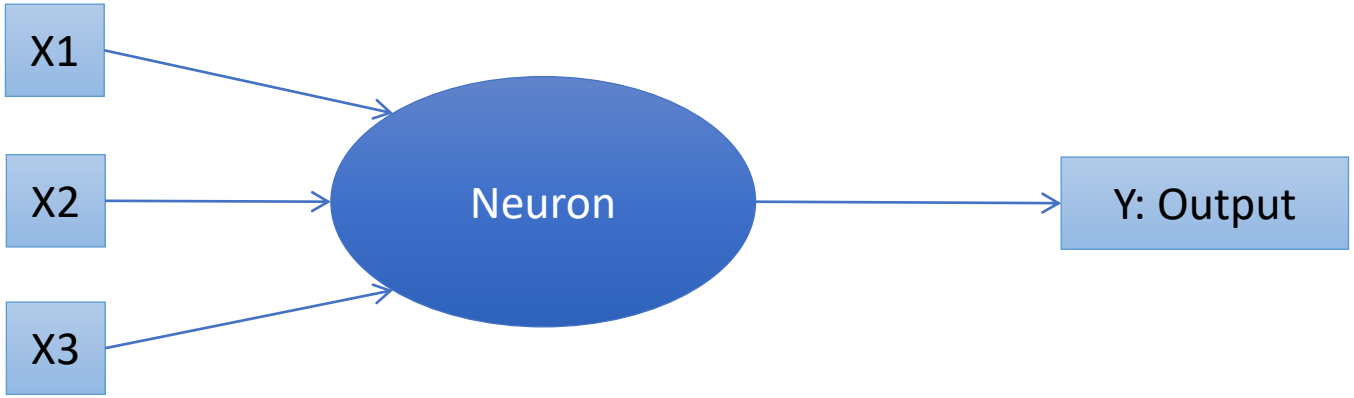
Biological Neurons



Inspiration from the brain

Warren McCulloch and Walter Pitts (1943) "A Logical Calculus of the Ideas Immanent in Nervous Activity".

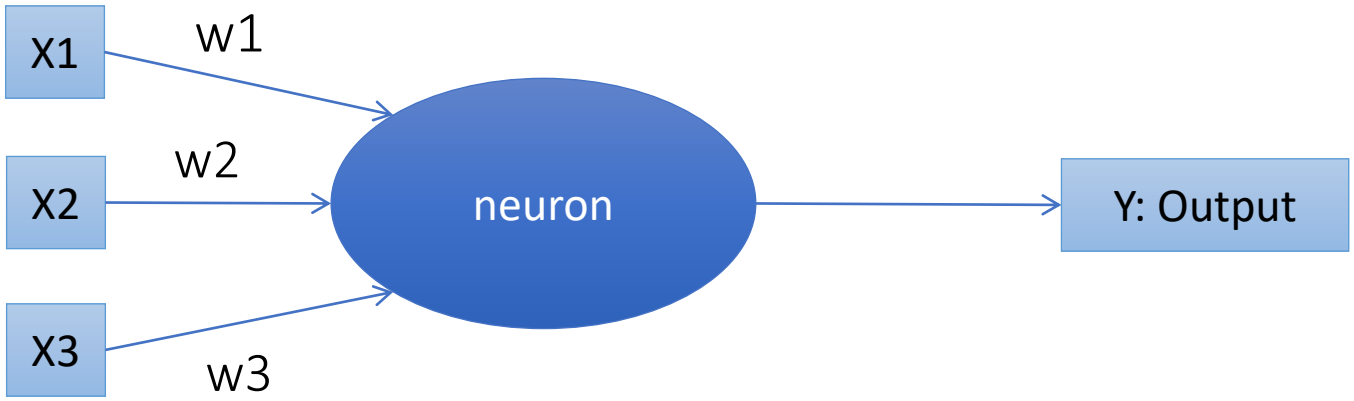




Contains key properties of **real** neurons:

- Synaptic weights
- Cumulative affect
- Threshold for activation "all or nothing”
  (neuron fires an output signal if the sum of inputs is above threshold)

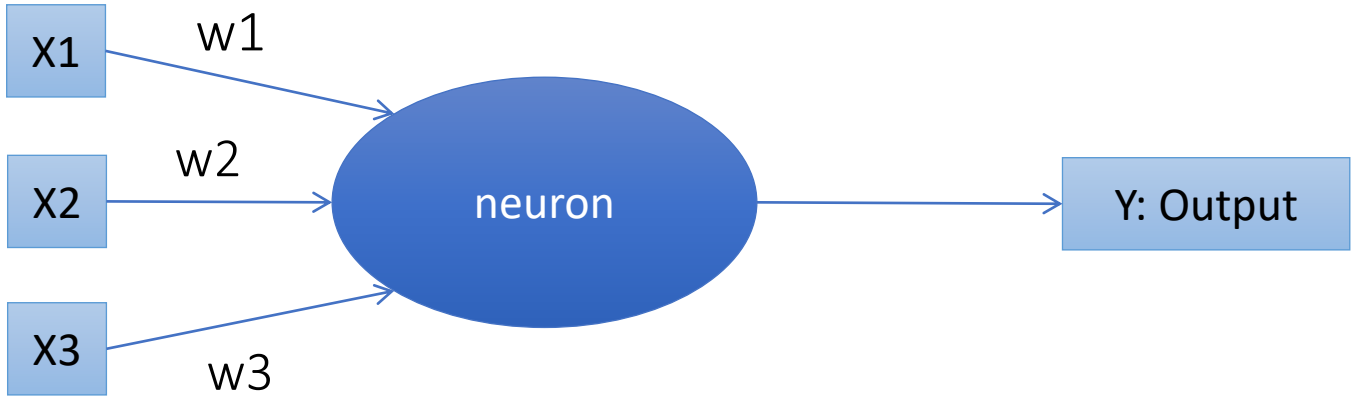
The perceptron model (Rosenblatt, 1957)



$$\sum_i x_i w_i \longrightarrow f\left(\sum_i x_i w_i\right) = y = \begin{cases} 1 & \text{if } \sum_i x_i w_i > T \\ 0 & \text{otherwise} \end{cases}$$

- Synaptic weights
- Cumulative affect
- Threshold for activation "all or nothing”
- Self-training the weights

Perceptron as linear separator/regressor



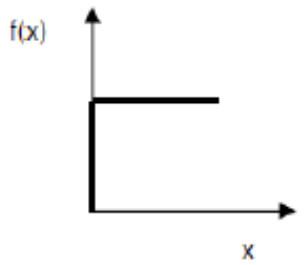
$$\sum_i x_i w_i + b \longrightarrow f\left(\sum_i x_i w_i\right)$$

f: activation function

Types of activation functions

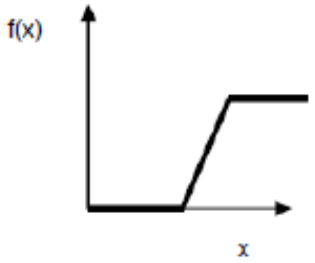
Threshold Function

$$f(x) = \begin{cases} 1 & if \ x \geq 0 \\ 0 & if \ x < 0 \end{cases}$$



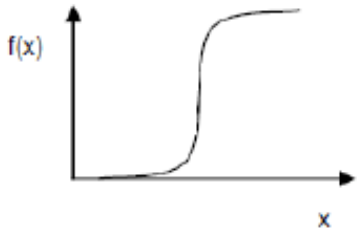
Piecewise-Linear Function

$$f(x) = \begin{cases} 1 & if \ x \geq 0.5 \\ x + 0.5 & if \ -0.5 \leq x \leq 0.5 \\ 0 & if \ x \leq 0.5 \end{cases}$$



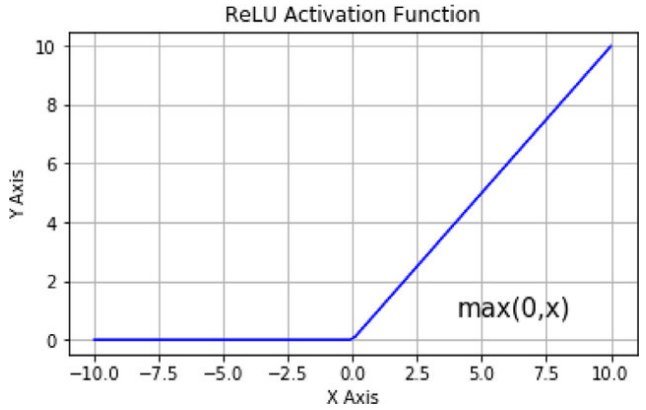
Sigmoid Function

$$f(x) = \frac{1}{1 + e^{-x}}$$

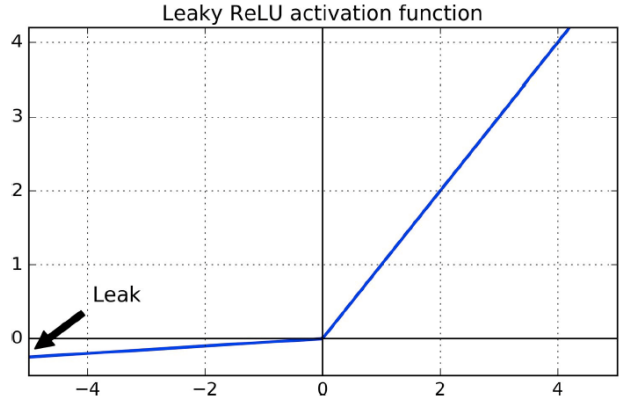


Types of activation functions

ReLU (rectified linear unit)

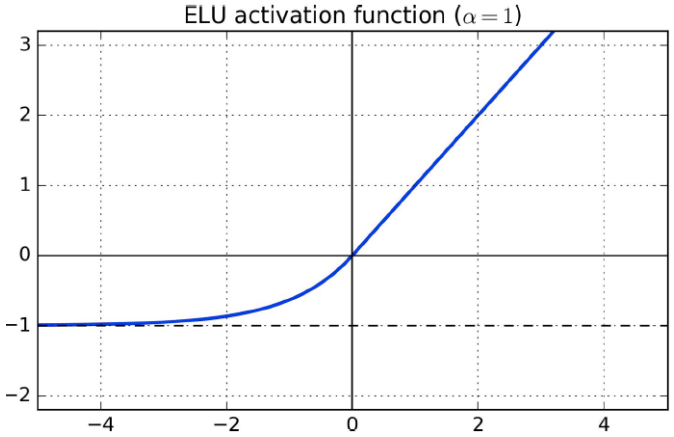


Leaky ReLU



$$\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z,\ z)$$

ELU (exponential linear unit)



$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$

Multi-layered neural networks

How can we achieve non-linear separators with the perceptron model?

Possible solution: multi-layer neural nets

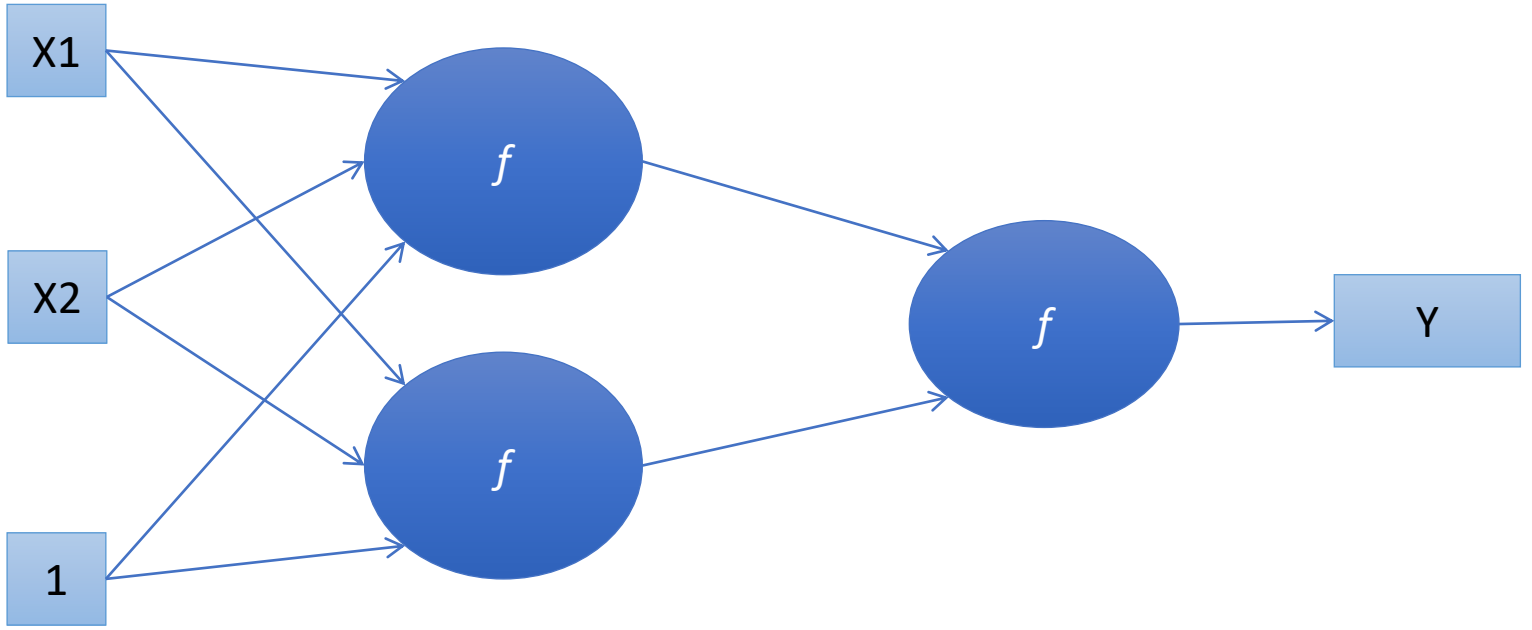
- Instead of having inputs feeding directly into output neurons, let’s add some intervening "hidden" neurons in between? The brain is certainly like that.

- Intuition: If we think of single neuron as dividing a space into low vs high output with a single line...

- … then multiple neurons = multiple dividing lines

- Non-linear separation can be approximated by a set of linear lines

Multi-layered neural networks (cont'd)



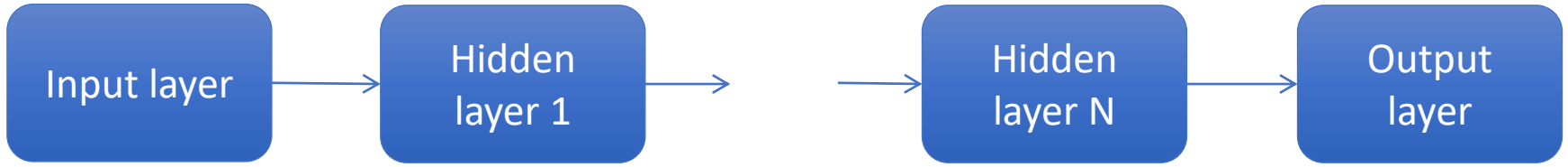
Input layer Hidden layers Output layer

- Neurons feeding into other neurons...
- Our black box is quite complicated now; can approximate arbitrary functions given enough hidden neurons.

Training multi-layered neural networks

This sounds cool! But bow can we train this complex back box?

Idea: We could use the usual gradient descent approach to train the weights between the last hidden layer and the output layer.



Issue: what about the weights of the other hidden layers?

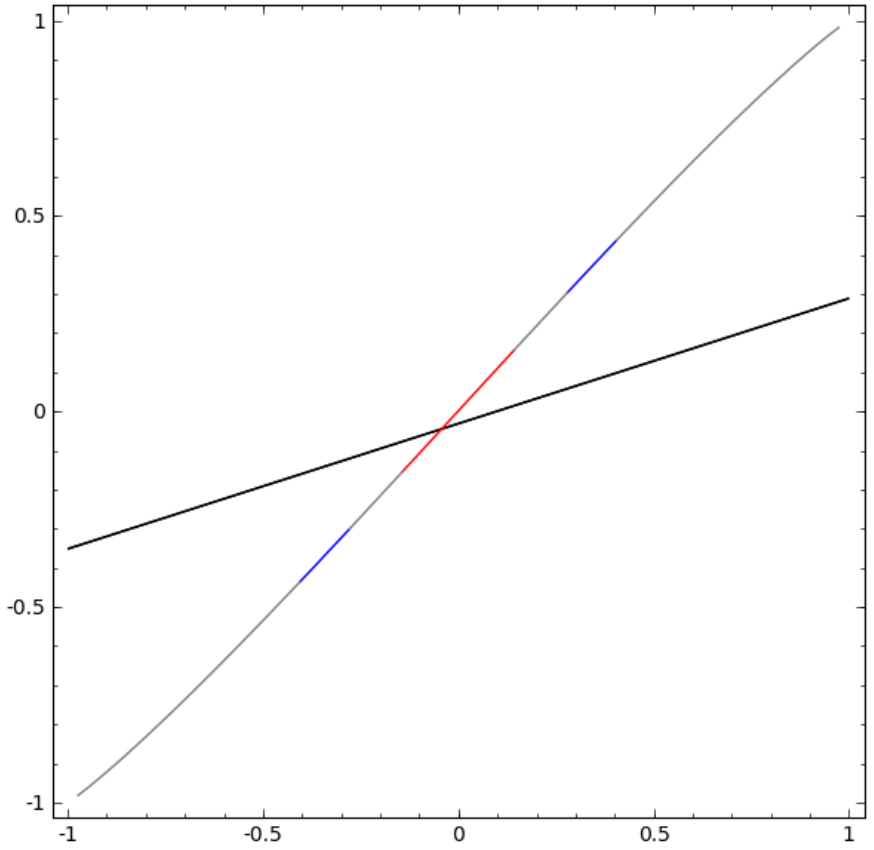
Solution: backpropagation of errors (Rumelhart, Hinton, and Williams, 1986)

Deep neural networks

Main idea of deep neural networks: transform the input space into higher level abstractions with lower dimensions

- Multi-layer architecture (typically with many hidden layers) – hence the name deep learning
- Each layer is responsible for a space transformation step
- By doing so, the complexity of non-linearity is decreased
- This is, however, is very expensive. Needs to rely on new computational solutions: GPUs, grid computing

Deep neural networks



Source: Chris Olah's blog (http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/)

DNNs as universal approximators

Goal: We want to approximate the decision boundary function with neural networks

Approximable function: f is approximable by neural networks if $\forall \varepsilon > 0$, there exist a neural network N such that:

$$\forall \mathbf{x}: \quad |f(\mathbf{x}) - N(\mathbf{x})| < \varepsilon$$

DNNs as universal approximators

Goal: We want to approximate the decision boundary function with neural networks

Approximable function: *f* is approximable by neural networks if $\forall \varepsilon > 0$, there exist a neural network N such that:

$$\forall \mathbf{x}: \quad |f(\mathbf{x}) - N(\mathbf{x})| < \varepsilon$$

Cybenko (1989): For any continuous function *f*, there exists a neural network N with sigmoid activation functions that can approximate *f* with ε error (this result was for shallow networks).

DNNs as universal approximators

Goal: We want to approximate the decision boundary function with neural networks

Approximable function: *f* is approximable by neural networks if $\forall \varepsilon > 0$, there exist a neural network N such that:

$$\forall \mathbf{x}: \quad |f(\mathbf{x}) - N(\mathbf{x})| < \varepsilon$$

Cybenko (1989): For any continuous function *f*, there exists a neural network N with sigmoid activation functions that can approximate *f* with ε error (this result was for shallow networks).

Deep networks: Rolnick&Tegmark (2018) + Kidger&Lyons (2020): deep neural nets with ReLU can use exponentially smaller number of neurons to achieve same unversal approximability

DNNs as universal approximators

Goal: We want to approximate the decision boundary function with neural networks

Approximable function: *f* is approximable by neural networks if $\forall \varepsilon > 0$, there exist a neural network N such that:

$$\forall \mathbf{x}: \quad |f(\mathbf{x}) - N(\mathbf{x})| < \varepsilon$$

Cybenko (1989): For any continuous function *f*, there exists a neural network N with sigmoid activation functions that can approximate *f* with ε error (this result was for shallow networks).

Deep networks: Rolnick&Tegmark (2018) + Kidger&Lyons (2020): deep neural nets with ReLU can use exponentially smaller number of neurons to achieve same universal approximability.

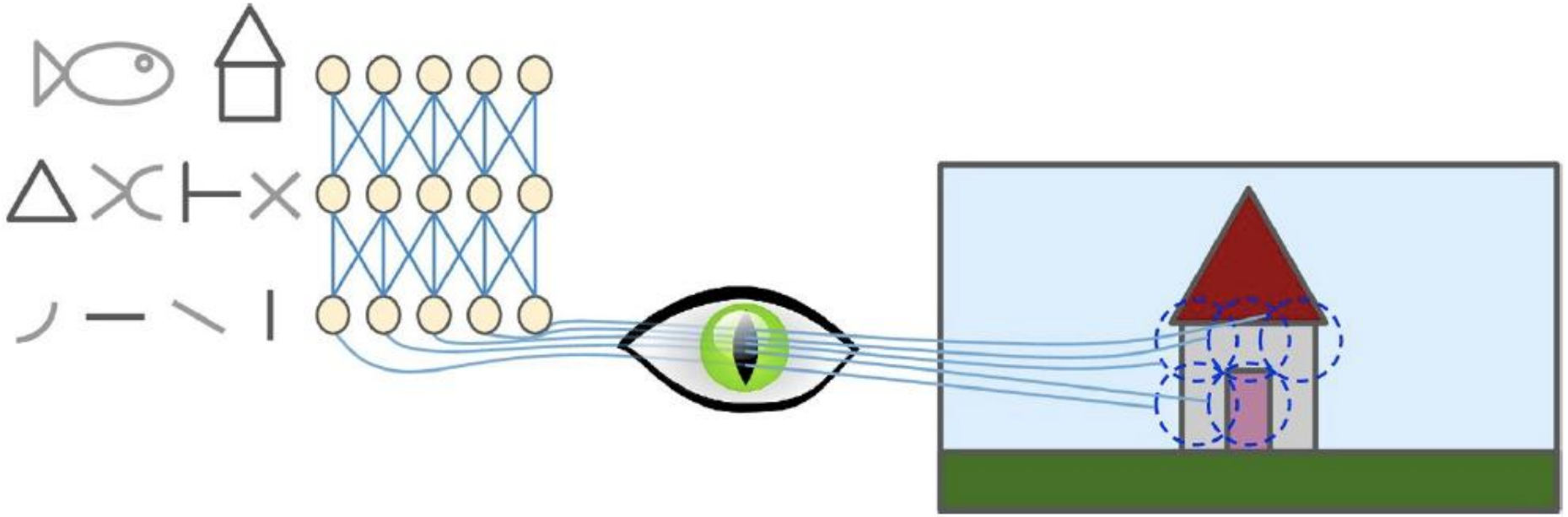
Width of the network: Johnson (2019): width <= dimension of inputs -> not universal approximators (doesn't matter how deep the network is). Hanin&Sellke (2018): width = #(input_dim) + 1 suffices.

Convolutional neural nets

Images taken from book of Aurélien Géron: Hands-On Machine Learning with Scikit-Learn & TensorFlow

Neural nets in computer vision:

- One of the core domains of CS and AI
  - Identify (classify) visual objects
- NNs are the state-of-the-art solution techniques
  - Natural representation of human vision
  - Intuitive model for convolutional networks (CNNs)



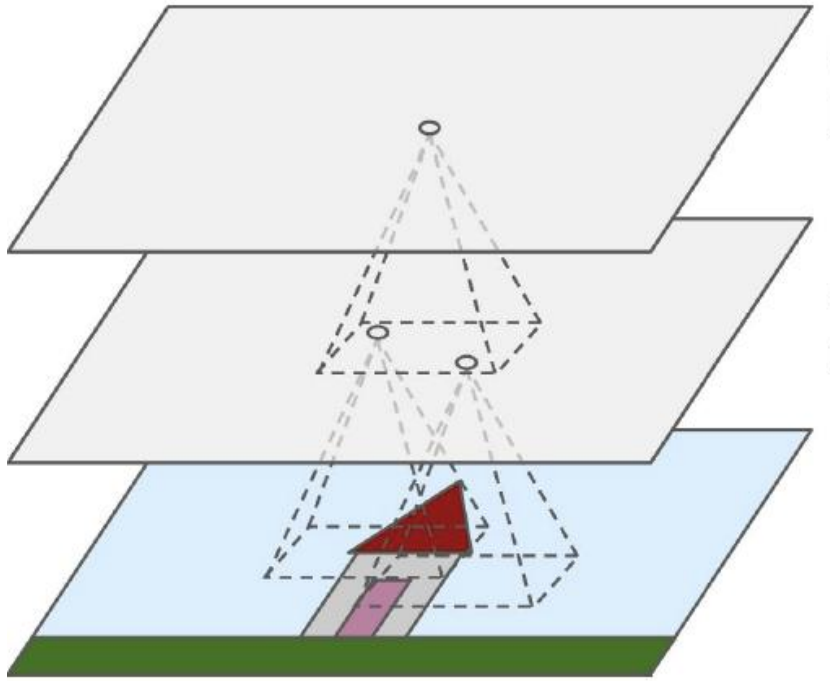
Ideally (before discussing CNNs):

- DNN with fully connected layers (each node is connected to every single node in the next layer):
  - Can learn (almost) any data model
  - E.g., if we don't need a connection -> set its weight close to 0
  - So it should be able to classify the visual object

- Issues:
  - Too many parameters
  - E.g., 100x100 picture (in pixels) – 1st hidden layer with 1000 neurons -> 10 million connections

Main idea

- Convolutional layer:
  - Neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields

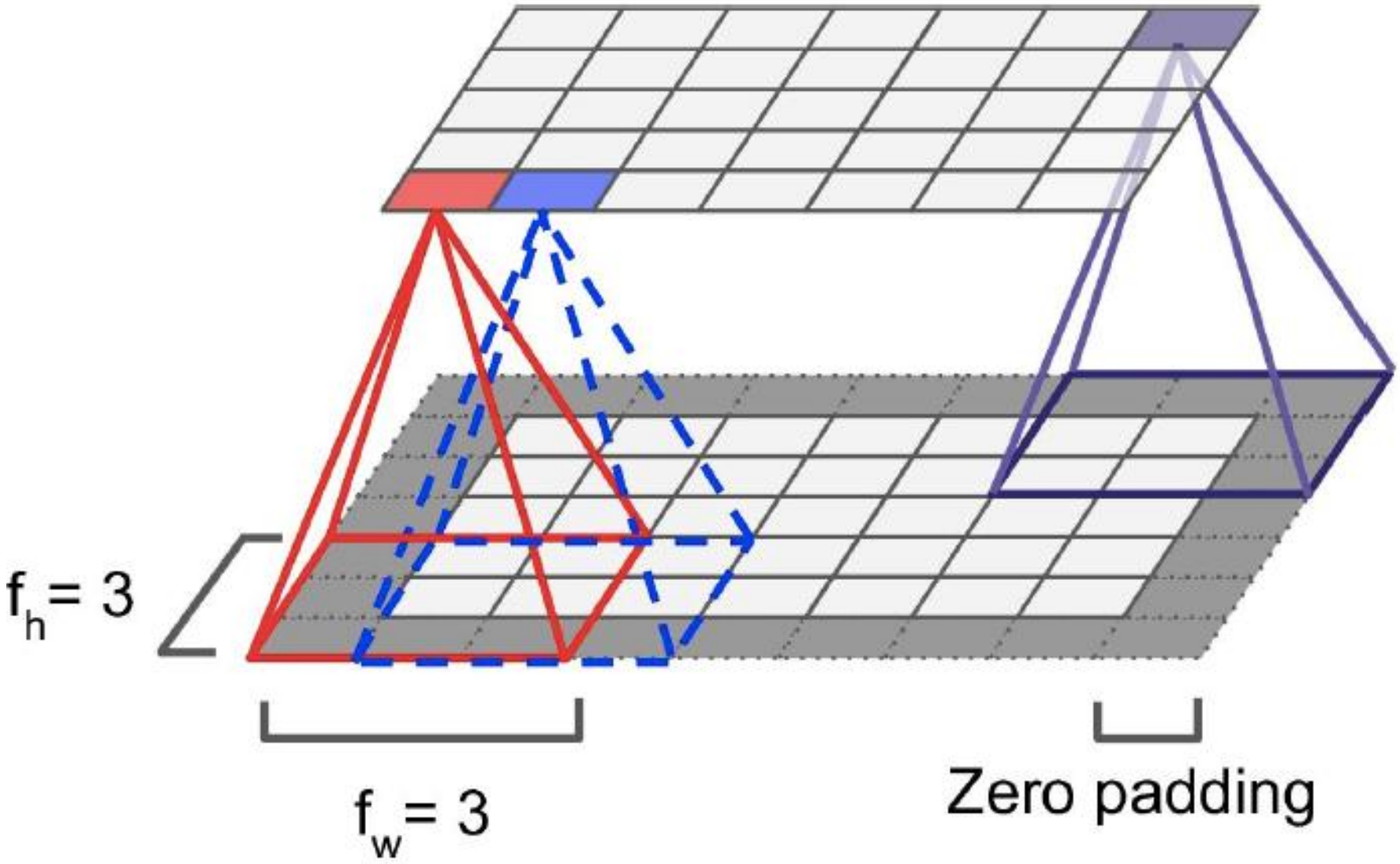


Convolutional neural nets

- Convolutional layer:
  - Motivated by nature
  - neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields
  - Next convolutional layer: each neuron is connected only to neurons within a small rectangle of first layer
  - Etc.

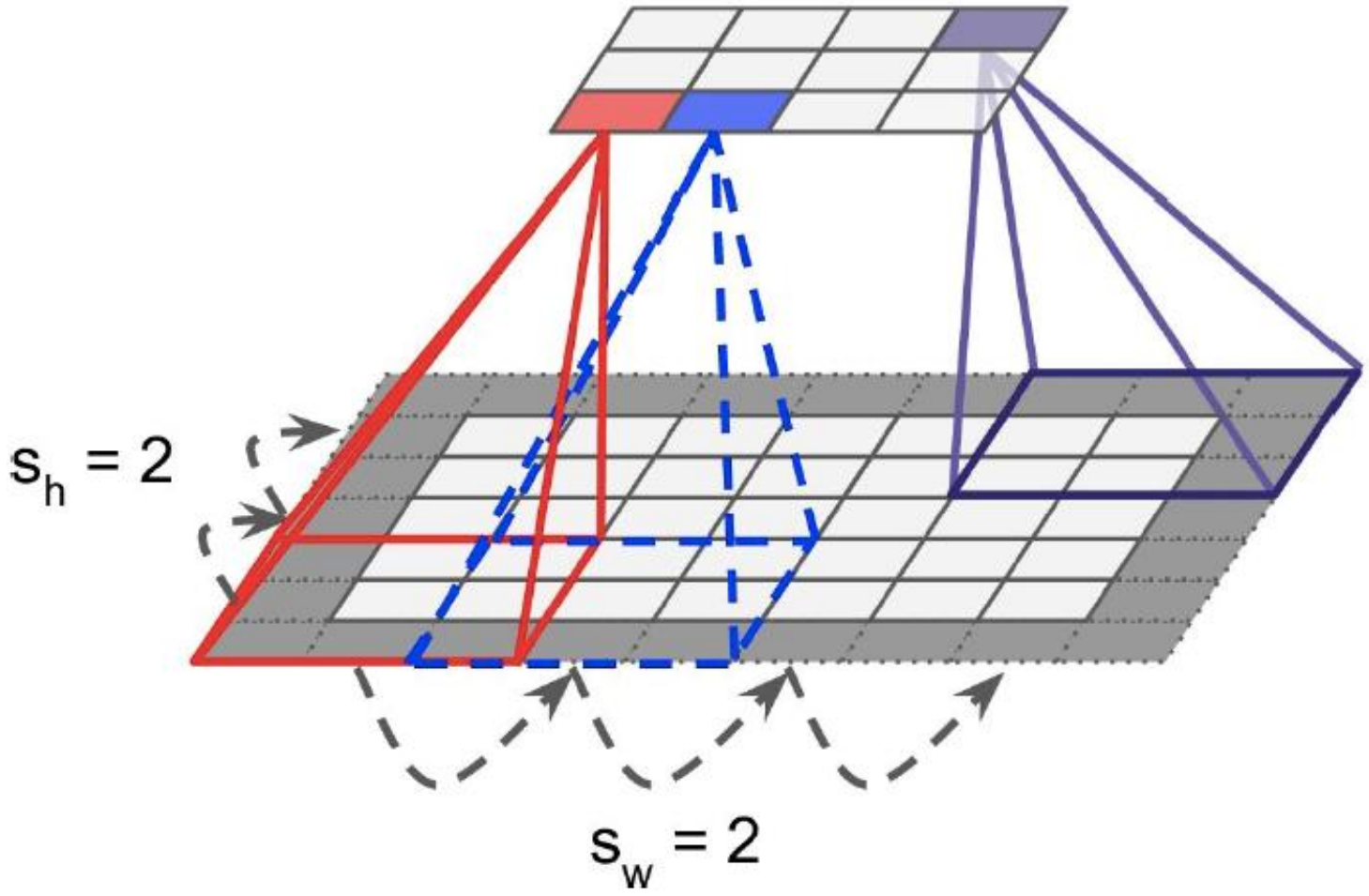
- Main benefits:
  - Can learn a lower level non-linear representation of data
  - Differentiable -> can use SGD to train

Zero padding



Reducing size of convolutional layer:

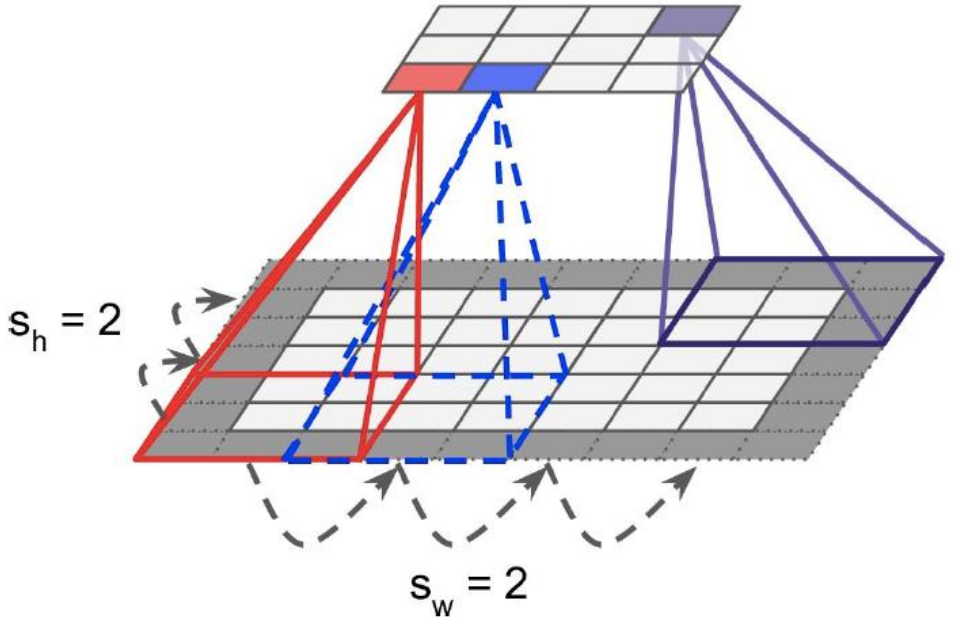
- Stride: size of jump between neighbouring rectangles
- Using large strides will significantly reduce the layer's size, but the cost is information loss



Filters

- Weight of edges that connect the rectangle with the neuron together form a matrix
- Matrix = filter

- E.g.: $\begin{pmatrix} 010 \\ 010 \\ 010 \end{pmatrix}$ = only consider the central line and ignore the rest



Pooling layers

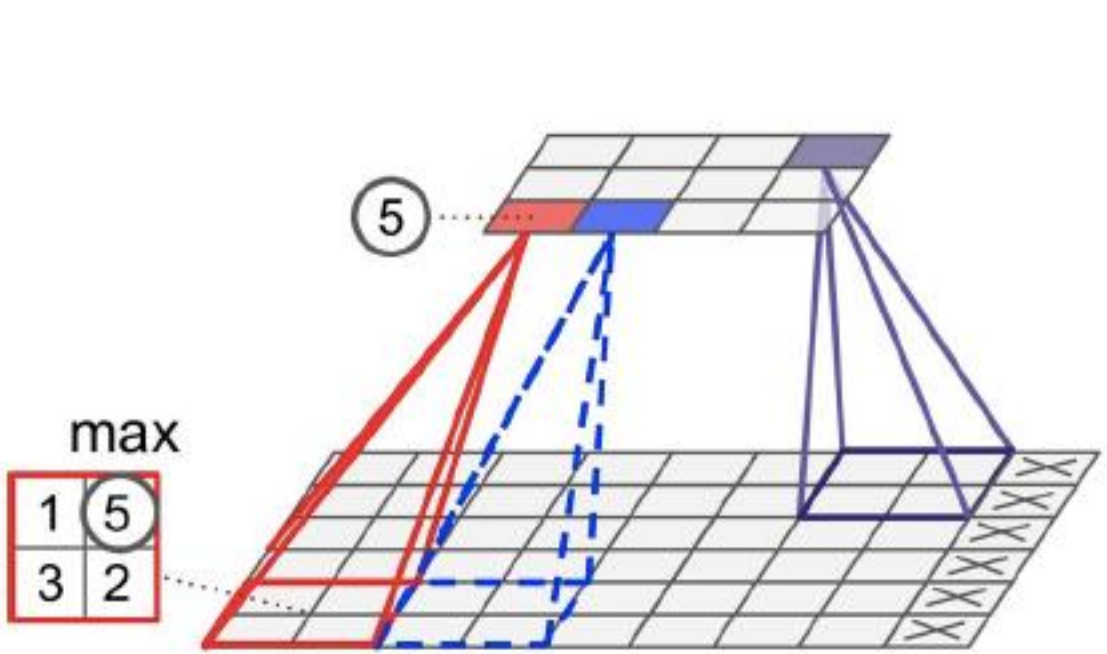
Goal: to reduce input image size (to reduce computational load, memory, etc)

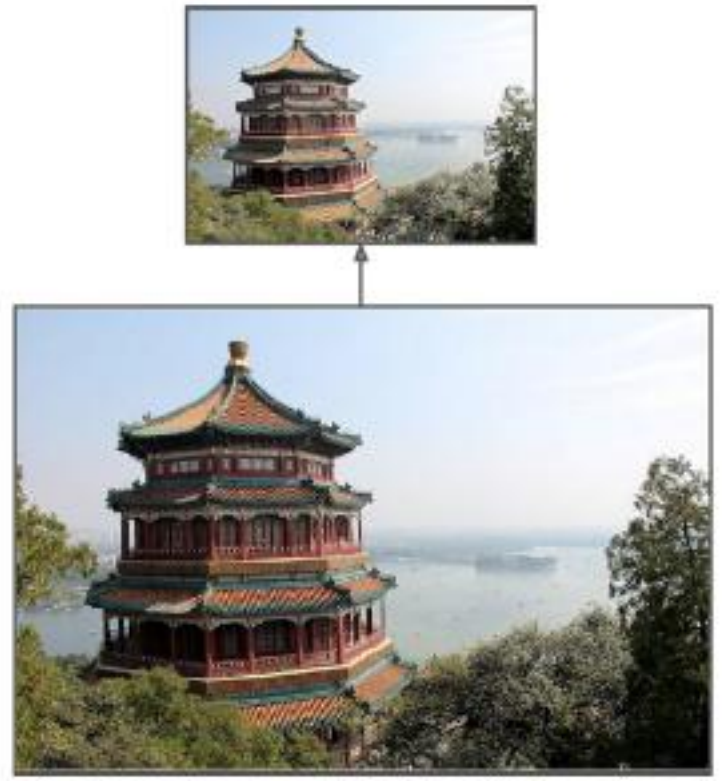
- Additional consequence: location invariance – the CNN can tolerate a small image shift

Pooling layer: similar to convolutional layer – each neuron is connected to a rectangle of input layer

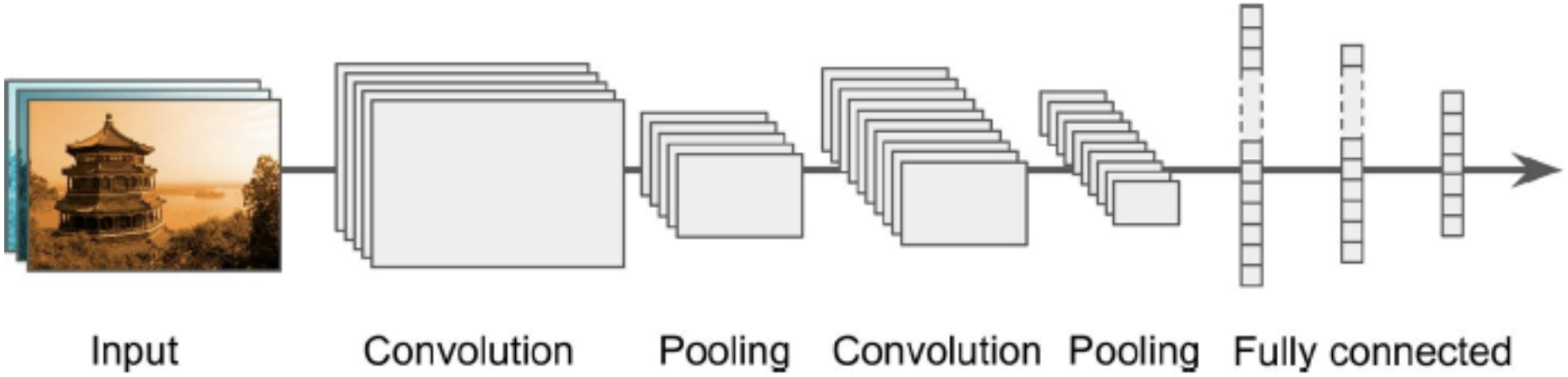
- Rectangle size, stride, padding, etc are the same
- But there are NO WEIGHTS!!!
- Aggregation: max or mean
- Max pooling: aggregation = max (other inputs are dropped)
- Average pooling: aggregation = mean

Max pooling example





CNN architectures



Universal approximability of CNNs

Yarotsky (2018) + Zhou (2018): CNNs with arbitrary depth are universal approximators

Open problems: Minimum depth? Minimum width?

Deep reinforcement learning

Deep reinforcement learning

- Use DNNs to represent
  - Value, Q function
  - Policy
  - Model

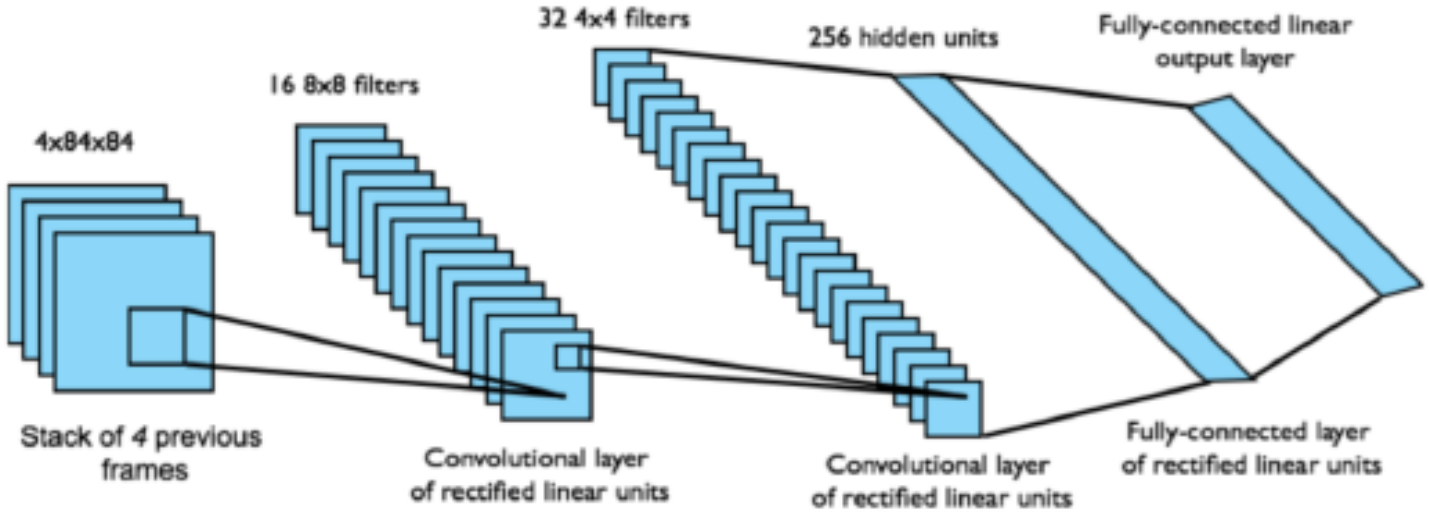
- Optimise loss function by stochastic gradient descent (SGD)

- Notable example: DQN (deep Q-network)

Architecture of DQNs

DQNs for Atari games:

- End-to-end learning of values Q(s, a) from pixels s
- Input: state s is stack of raw pixels from last 4 frames
- Output: Q(s, a) for 18 joystick/button positions
- Reward: change in score for that step
- Network architecture and hyperparameters fixed across all games



Taken from Emma Brunskill's slides

Architecture of DQNs

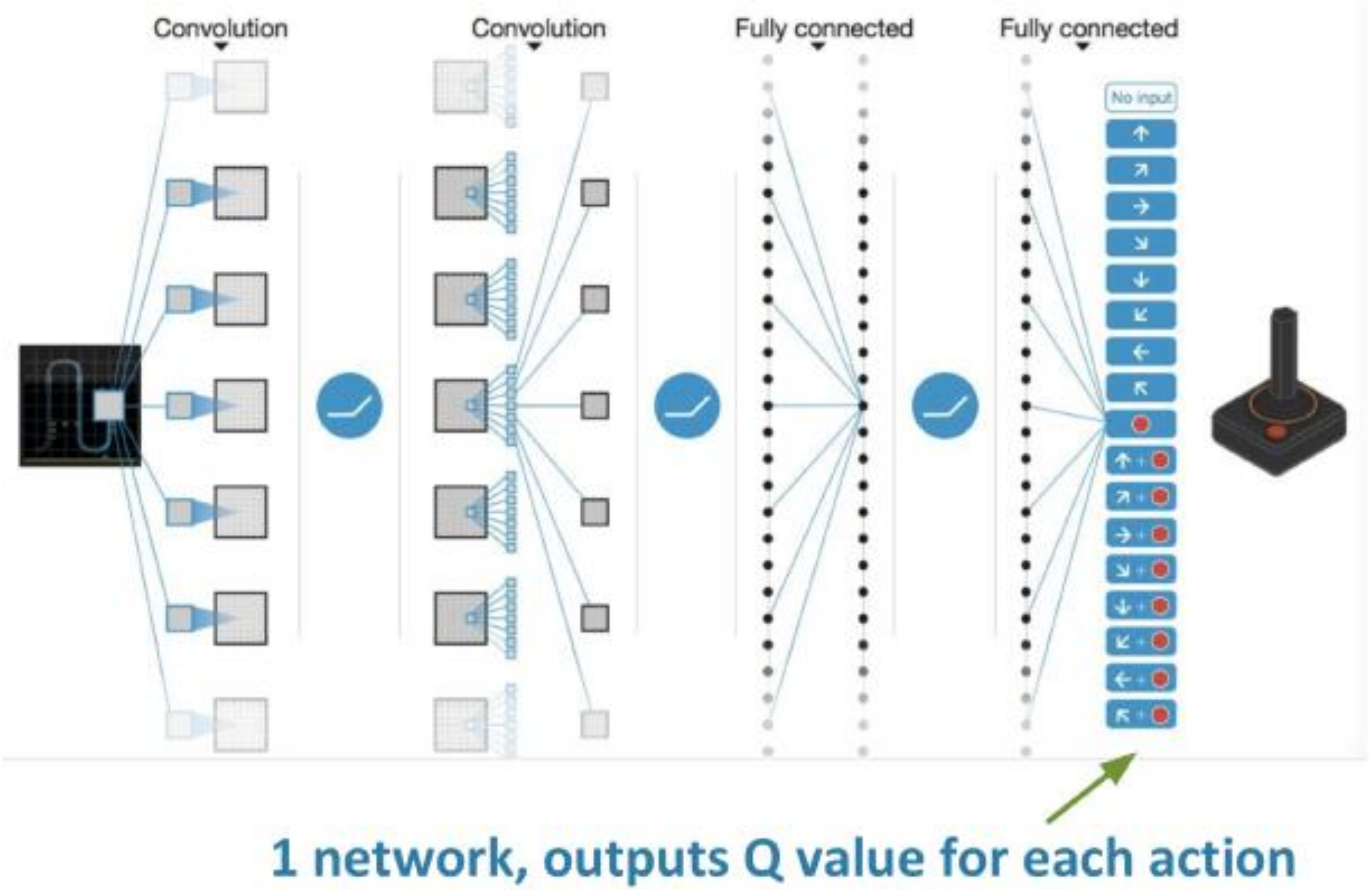


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Taken from Emma Brunskill's slides

Advantages of DQNs

Q-learning:

- Converges to the optimal Q(s, a) using table lookup representation
- But Q-learning with value function approximation can diverge

Advantages of DQNs

Q-learning:

- Converges to the optimal Q(s, a) using table lookup representation
- But Q-learning with value function approximation can diverge

Two of the issues causing problems:

- Correlations between samples
- Non-stationary targets

Advantages of DQNs

Q-learning:

- Converges to the optimal Q(s, a) using table lookup representation
- But Q-learning with value function approximation can diverge

Two of the issues causing problems:

- Correlations between samples
- Non-stationary targets

Deep Q-learning (DQN) addresses both of these challenges by

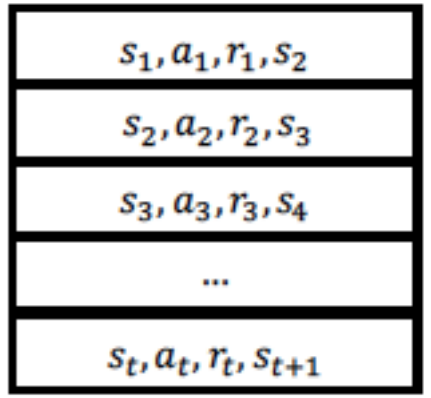
- **Experience replay**
- **Fixed Q-targets**

Experience replay

Issue: correlation between consecutive samples (i.e., current state-action pair affects the success of next state-action)

Solution:

- Store previous experience tuples (s_t, a_t, r_t, s_{t+1}) in a table (**replay buffer**)
- Sample a tuple $(s, a, r, s') \sim D$ // similar to bootstrapping
-



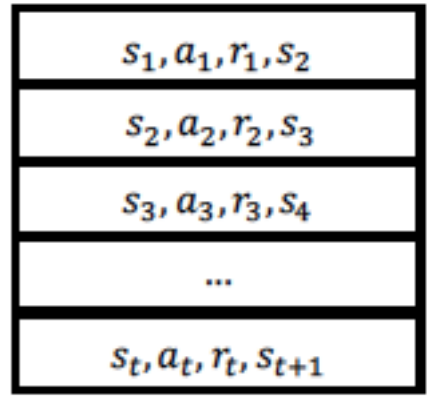
Experience replay

Issue: correlation between consecutive samples (i.e., current state-action pair affects the success of next state-action)

Solution:

- Store previous experience tuples (s_t, a_t, r_t, s_{t+1}) in a table (**replay buffer**)
- Sample a tuple $(s, a, r, s') \sim D$ // similar to bootstrapping
- Compute the target value for the sampled *s*:

$$r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w})$$



- Use SGD to update the network weights **w**:

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

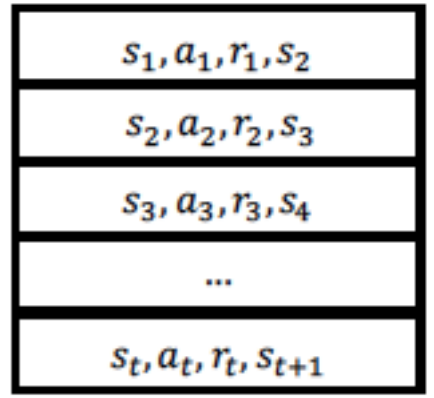
Experience replay

Issue: correlation between consecutive samples (i.e., current state-action pair affects the success of next state-action)

Solution:

- Store previous experience tuples (s_t, a_t, r_t, s_{t+1}) in a table (**replay buffer**)
- Sample a tuple $(s, a, r, s') \sim D$ // similar to bootstrapping
- Compute the target value for the sampled *s*:

$$r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w})$$



- Use SGD to update the network weights **w**:

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

New issue: **w** gets updated on the next round, changing the target value -> not converge to the Q function

Fixed Q-target

Issue: In experience replay, **w** gets updated on the next round, changing the target value -> not converge to the Q function

Solution:

- Fix the weight **w** in target calculation for multiple updates -> improve stability
- $\mathbf{w}^-$: fixed weight used to calculate multiple target values

Fixed Q-target

Issue: In experience replay, **w** gets updated on the next round, changing the target value -> not converge to the Q function

Solution:

- Fix the weight **w** in target calculation for multiple updates -> improve stability
- $\mathbf{w}^-$: fixed weight used to calculate multiple target values

- Compute the target value for the sampled *s* (now with fixed target weights):

$$r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}^-)$$

- Use SGD to update the network weights **w** (with a fixed target):

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}^-) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

DQN vs. other function approximation methods

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

Taken from Emma Brunskill's slides

More advanced deep RL (not covered in this course)

- Double DQN (Deep Reinforcement Learning with Double Q-Learning, Van Hasselt et al, AAAI 2016)
- Prioritized Replay (Prioritized Experience Replay, Schaul et al, ICLR 2016)
- Dueling DQN (best paper ICML 2016) (Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, ICML 2016)