SIT796 Reinforcement Learning

# Function Approximation

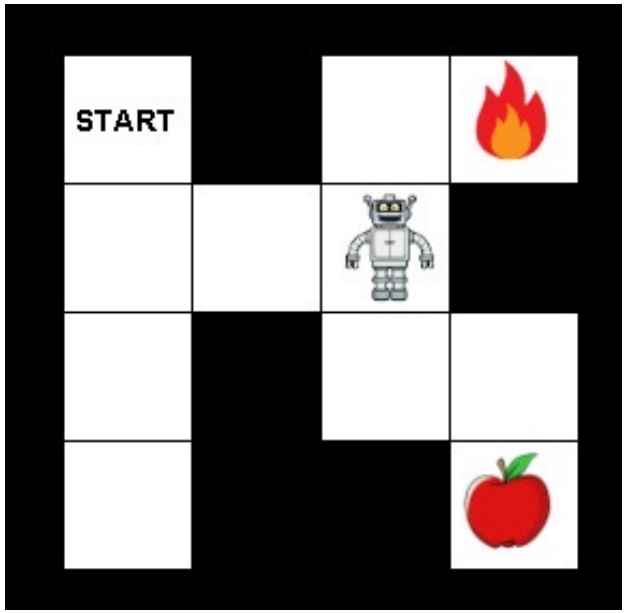
DEAKIN UNIVERSITY

Presented by:
Thommen George Karimpanal
School of Information Technology

# Approach so far

Tabular approach:

- Single entry in a table of values
- States and actions have discrete representations
- Working set of algorithms
- Convergence proofs



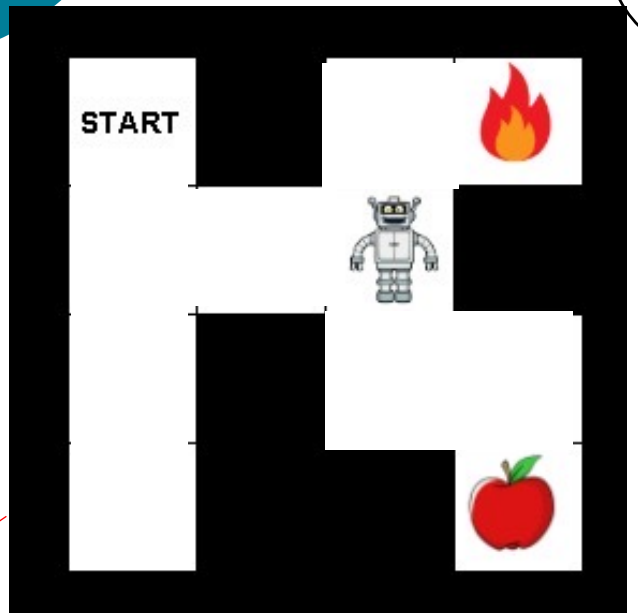

State

Action

| | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| $s_1$ | $Q(s_1,a_1)$ | $Q(s_1,a_2)$ | $Q(s_1,a_3)$ |
| $s_2$ | $Q(s_2,a_1)$ | $Q(s_2,a_2)$ | $Q(s_2,a_3)$ |
| $s_3$ | $Q(s_3,a_1)$ | $Q(s_3,a_2)$ | $Q(s_3,a_3)$ |
| $s_4$ | $Q(s_4,a_1)$ | $Q(s_4,a_2)$ | $Q(s_4,a_3)$ |
| | | | |

$\pi^*$

# Limitations of Tabular Approach




State

Action

| | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| $s_1$ | $Q(s_1,a_1)$ | $Q(s_1,a_2)$ | $Q(s_1,a_3)$ |
| $s_2$ | $Q(s_2,a_1)$ | $Q(s_2,a_2)$ | $Q(s_2,a_3)$ |
| $s_3$ | $Q(s_3,a_1)$ | $Q(s_3,a_2)$ | $Q(s_3,a_3)$ |
| $s_4$ | $Q(s_4,a_1)$ | $Q(s_4,a_2)$ | $Q(s_4,a_3)$ |

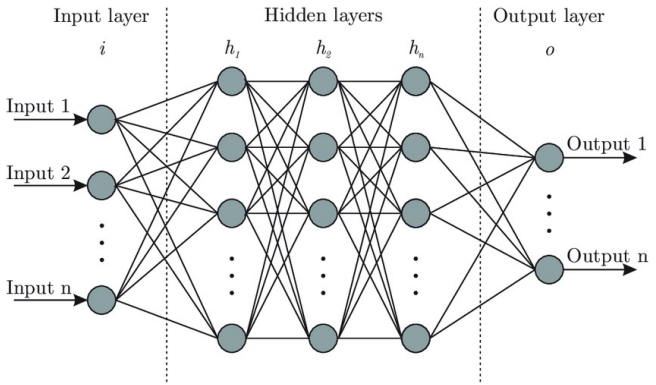
$\pi^*$

Function approximators (Linear, RBF, NN)

# Function Approximation

Let's first consider state value prediction $v_\pi$ with function approximation.

- Instead of representing each state in a table of values, we represent it in a parameterised functional form using a weight vector $\mathbf{w} \in \mathbb{R}^d$
- Hence, we write the value for the approximate value of state $s$ as given the weight vector $\mathbf{w}$: $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
  - The number of dimensions of the vector is strictly much less the number of raw states $d \ll |\mathcal{S}|$.
  - Hence changing the value of one weight changes the value of many weights
  - Problem now is to find a $\mathbf{w}^*$ that best approximates $v_\pi(s)$ (or $Q_\pi(s, a)$)

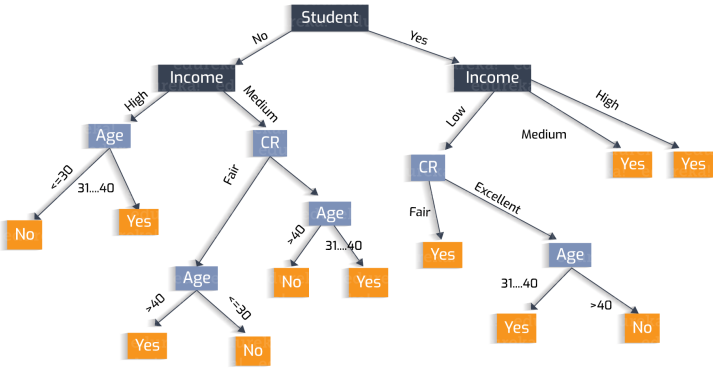



## Examples of what $\hat{v}$ might be

Could be a linear function of features in the state where $\mathbf{w}$ is the weight of each feature

Could be non-linear function of features in the state computed over an *Artificial Neural Network*

Could be function computed by a decision tree where $\mathbf{w}$ is all the numbers defining the split points of the tree








https://miro.medium.com/max/3840/1*jojTznh4HOX_8cGw_04ODA.png

https://www.researchgate.net/profile/Facundo_Bre/publication/321259051/figure/fig1/AS:614329250496529@1523478915726/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o.png

# Linear Methods

A special case of function approximation commonly used Linear function approximation.

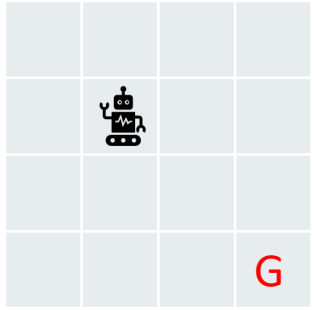
- That is our approximation function $\hat{v}(\cdot, \mathbf{w})$ is linear function over the weight vector $\mathbf{w}$.
- Where each state $s$ is represented with a vector $\mathbf{x}(s) = [x_1(s), x_2(s), \cdots, x_d(s)]^\top$, where $|\mathbf{x}(s)| = |\mathbf{w}|$
- Now we can represent our state-value function using the inner product of $\mathbf{w}$ and $\mathrm{x}(s)$

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

- Here $\mathbf{x}(s)$ is referred to as the *feature vector* for the state $s$.

It worth noting that the new notation can still represent the tabular approaches seen so far.

For instance, represent the state as a vector of the table entries. The feature representing the location of the agent is set to 1 and all others to 0.




$$\mathbf{x}(s) = [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0]$$

$$\mathbf{w} = [0.0,0.1,0.2,0.4,0.1,0.3,0.4,0.6,0.2,0.4,0.6,0.8,0.4,0.6,0.8,1.0]$$

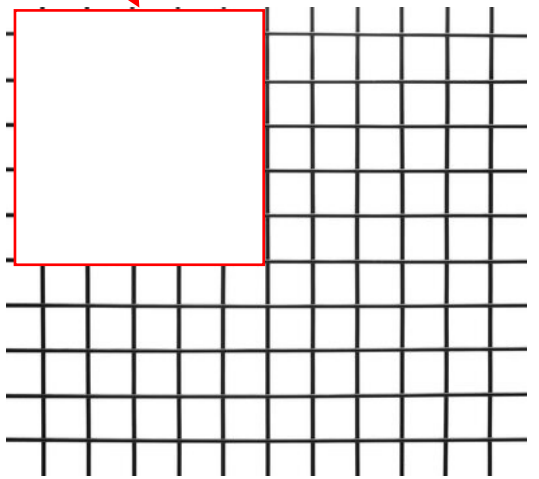
$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = 0.3$$

# State Aggregation

Probably the simplest approach for function approximation is State Aggregation.

- The aim is simply to group similar states together and treat them as a single state
- The grouped states are represented with a single value within the weight vector

This is 1 state now



For example

Image we have a grid world with 10,000 grids in two dimensions = 100 million discrete states.

Recall the more states then the more values to learn and less often we get to visit them relative to all the states

Now if we know an action may move us between one step up to 100 grids during each time step then we may group 50 states of each dimension as a single state.

This will reduce our states to $200 \times 200 = 40{,}000$ states

Faster learning but each group of 50 states will each be assigned a single common value.





https://miro.medium.com/max/3840/1*jojTznh4HOX_8cGw_04ODA.png

https://www.researchgate.net/profile/Facundo_Bre/publication/321259051/figure/fig1/AS:614329250496529@1523478915726/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o.png

SIT796 Reinforcement Learning

# Stochastic Gradient MC

DEAKIN UNIVERSITY

Presented by:
Thommen George Karimpanal
School of Information Technology

# Stochastic Gradient MC

In Gradient Descent our aim is to reduce the error of all our examples.

- However, this is not useful in RL because we do not have all the examples when learning online through interaction.

Stochastic Gradient Descent (SGD) allows us to use a single example by adjusting our weight vector $\mathbf{w}$ in the direction of the estimated error, governed by a small factor $\alpha$.

- To do this we must be able to identify the direction of the error, hence our function $\hat{v}(s, \mathbf{w})$ must be differentiable.
  - We find the slope of a function by finding its partial derivative in this case with respect to $\mathbf{w}$

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_i}, \frac{\partial f(\mathbf{w})}{\partial w_i}, \cdots, \frac{\partial f(\mathbf{w})}{\partial w_i}\right)^\top$$

- Now when calculating an update we move the weight vector towards a smaller error by moving a small amount in the direction of the error.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[G_t - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t)$$

- In the linear case this reduces to:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[G_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(s)$$

E.g. the gradient for the example on the previous slide is

$$\nabla\hat{v}(S_t, \mathbf{w}_t) = [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0]$$

*Input:*
the policy π to be evaluate
A differentiable function $\hat{v}: \mathcal{S}\times\mathbb{R}^d \to \mathbb{R}$
*Algorithm Parameter:*
Step size $\alpha \in (0,1]$
*Initialise:*
$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$
Loop forever (for each episode):
Generate an episode based on π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
Loop for each step of the episode $t = 0,1, \ldots, T-1$:
$\mathbf{w} = \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$

Stochastic Gradient Descent for MC Prediction

# Semi-Gradient $TD(0)$

When using applying these ideas to bootstrapping methods (e.g. TD) then we are basing updates on an *estimate* rather than the true value.

Semi-gradient methods need to replace the true target with an estimate of the target $G_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$

*Input:*
the policy π to be evaluate
A differentiable function $\hat{v}: \mathcal{S}\times\mathbb{R}^d \rightarrow \mathbb{R}$
*Algorithm Parameter:*
Step size $\alpha \in (0,1]$
*Initialise:*
$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$
Loop forever (for each episode):
Initialize $S$
Loop for each step of the episode until $S \in \mathrm{S}^{(\mathrm{Terminal})}$ :
Choose $A \sim \pi(\cdot\,|S)$
Take action $A$, observe $R, S'$
$\mathbf{w} = \mathbf{w} + \alpha[R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})]\nabla\hat{v}(S, \mathbf{w})$
$S \leftarrow S'$

Why semi-gradient?

When we update **w**, we take into account the change in estimate, but don't take into account the change in the target (which also depends on **w**)

So the gradient computed only represents part of the true gradient

Stability issues!

# Semi-Gradient $TD(\lambda)$

Recall we were able to use eligibility traces to define an algorithm that sat between MC and TD(0).

- We obviously also wish to have this ability when using function approximation.
- This will allow us to update the weight vector each iteration, balance computation throughout instead of just at the end, and allow its application to continuing problems.

Instead of defining a trace for each state, we define it as a vector $\mathbf{z}_t \in \mathbb{R}^d, s.t. |\mathbf{z}| = |\mathbf{w}|$

$$\mathbf{z}_{-1} = 0$$
$$\mathbf{z}_t = \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t), \qquad 0 \leq t \leq T$$

Now we can update the weight vector proportionally to the trace vector

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t$$

Where the TD-error is calculated the same as $TD(0)$

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$$

Instead of having to iterate through each $e(s)$ as we did in the tabular case, we simply update the trace vector in a single operation

*Input:*
The policy π to be evaluate
A differentiable function $\hat{v}: \mathcal{S}\times\mathbb{R}^d \to \mathbb{R}$
*Algorithm Parameter:*
Step size $\alpha \in (0,1]$
Trace decay rate $\lambda \in [0,1]$
*Initialise:*
$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$
Loop forever (for each episode):
Initialize $S$
Reset $\mathbf{z} = \mathbf{0}$
Loop for each step of the episode until $S \in \mathrm{S}^{(\mathrm{Terminal})}$ :
Choose $A \sim \pi(\cdot\,|S)$
Take action $A$, observe $R, S'$
$\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S, \mathbf{w})$
$\delta \leftarrow R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
$\mathbf{w} = \mathbf{w} + \alpha\delta\mathbf{z}$
$S \leftarrow S'$

SIT796 Reinforcement Learning

# Function approximation with Control

Presented by:
Thommen George Karimpanal
School of Information Technology

# Semi-Gradient Control

Now that we have a method of prediction, we can investigate a method of control.

- Now we want to approximate the action-value function $\hat{q} \approx q_\pi$.
- That is, we will represent the parameterized functional form using a weight vector $\mathbf{w}$.

Hence, One-step Sarsa with function approximation is defined with the update rule

$$\mathbf{w_{t+1}} = \mathbf{w}_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w_t}) - \hat{q}(S_t, A_t, \mathbf{w_t})]\nabla\hat{q}(S_t, A_t, \mathbf{w_t})$$

Control, however, is not just mapping the state to our vector $\boldsymbol{w}$ but also our actions.

For discrete actions this is perfectly fine – use a matrix where each column represents one of the possible actions.
Can also use a function approximation, such as action aggregation approach on the actions
However, every action increases the dimensionality of our value function

Neither of these are always suitable – Sometimes we want to have precise actions

E.g. The angle we turn the steering wheel on a car must be very precise to avoid an accident
This type of *continuous action* control is very much still an open question.

For now, we will assume there is a manageable discrete set of actions to select from

# Semi-Gradient $Sarsa(0)$

*Input:*

A differentiable state-action value function $\hat{q}: \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

A policy $\pi$ if predicting or $q_\pi$ if estimating (e.g. using $\varepsilon - greedy$)

*Algorithm Parameter:*

Step size $\alpha \in (0,1]$

*Initialise:*

$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$

Loop forever (for each episode):

$S, A \leftarrow$ Initial state and action of episode (e.g. using $\varepsilon - greedy$)

Loop for each step of the episode until $S \in S^{(\text{Terminal})}$ :

Take action $A$, observe $R, S'$

If $S' \in S^{terminal}$ then:

$\mathbf{w} = \mathbf{w} + \alpha[R + \gamma \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$, special case for terminal state can't include future state

else:

Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g. using $\varepsilon - greedy$)

$\mathbf{w} = \mathbf{w} + \alpha[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

$S \leftarrow S'$

$A \leftarrow A'$

# $Sarsa(\lambda)$ with Linear Function Approximation

Sarsa can also be extended to use eligibility traces similar to $TD(\lambda)$ – Note: This assumes binary features

*Input:*

A policy $\pi$ if predicting or $q_\pi$ if estimating (e.g. using $\varepsilon - greedy$)

A feature function $\mathcal{F}(s,a)$ returning the set of active feature indices for $s, a$

*Algorithm Parameter:*

Step size $\alpha \in (0,1]$

Trace decay rate $\lambda \in [0,1]$

*Initialise:*

$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$

Loop for each episode:

Initialise $S$

Choose $A \sim \pi(\cdot\,|S)$ initial action of episode (e.g. using $\varepsilon - greedy$)

Reset $\mathbf{z} = \mathbf{0}$

Loop for each step of the episode until $S'$ is a terminal state:

Take action $A$, observe $R, S'$

$\delta \leftarrow R - \sum_{i \in \mathcal{F}(S,A)} w_i$

for all $i \in \mathcal{F}(S,A)$:

$\mathbf{z} \leftarrow \mathbf{z} + 1$ or $\mathbf{z} \leftarrow 1$ depending on using accumulating or replacing traces

If $S' \in S^{terminal}$ then:

$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$

else:

Choose $A' \sim \pi(\cdot\,|S')$ action (e.g. using $\varepsilon - greedy$)

$\delta \leftarrow \delta + \gamma \sum_{i \in \mathcal{F}(S',A')} w_i$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$

$\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$

$S \leftarrow S';$

$A \leftarrow A';$

# Semi-Gradient Off-Policy Control

There is a function approximation version of Watkins's Q-Learning that has been very popular.

- Below is a binary features version of the approach which aligns with the Sarsa implementation.

*Input:*
A feature function $\mathcal{F}(s,a)$ returning the set of active feature indices for $s,a$
*Algorithm Parameter:*
Step size $\alpha \in (0,1]$
Trace decay rate $\lambda \in [0,1]$
*Initialise:*
$\mathbf{w} \in \mathbb{R}^d$ arbitrarily e.g. $\mathbf{w} = 0$
Loop for each episode:
Initialise $S$
Choose $A \sim \pi(\cdot\,|S)$ initial action of episode (e.g. using $\varepsilon - greedy$)
Reset $\mathbf{z} = \mathbf{0}$
Loop for each step of the episode until $S'$ is a terminal state:
Take action $A$, observe $R, S'$
$\delta \leftarrow R - \sum_{i \in \mathcal{F}(S,A)} w_i$
for all $i \in \mathcal{F}(S,A)$: $\mathbf{z} \leftarrow \mathbf{z} + 1$
for all $a \in \mathcal{A}(s)$:
$Q_a \leftarrow \sum_{i \in \mathcal{F}(S,a)} w_i$
$\delta \leftarrow \delta + \gamma \max_a Q_a$
$\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} + \alpha\delta\mathbf{z}$ update all weight vectors
Choose $A' \sim \pi(\cdot\,|S')$ action (e.g. using $\varepsilon - greedy$)
If $greedy\_action$ selected then:
$\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$
else:
$\mathbf{z} \leftarrow 0$
$S \leftarrow S'$;
$A \leftarrow A'$;

Note: for most of this unit we are using Sutton and Barto (2018) – This algorithm, however is based on Sutton and Barto (1998) version as they no longer include it in the new version of the book. The reasons discussed on the following slides. However, I include as it is still in common use

# The Deadly Triad

Watkins' Q-Learning with function approximation however has been found to be unstable and does not always converge.

"..the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call the deadly triad."

- Function approximation
- Bootstrapping
- Off-Policy training

Can we give up one of them? Presence of any 2 appears to be manageable (leads to stable learning)

Function approximation – Can not be replaced in large state spaces without introducing the Curse-of-Dimensionality

Bootstrapping – Is possible but at the cost of significantly more computation and loss of efficiency with increased memory costs

Off-Policy training – we can just use on-policy methods instead and often that is good. However, there are many use cases where we want to learn multiple policies simultaneously in parallel.

# Stable Off-policy Methods

There are a number of approaches recently that provide stable off-poly methods with function approximation.

- For instance, one approach might be to select off-policy behaviours that are close to the target policy can be effective

Gradient-TD methods.

- Aim to minimize the Projected Bellman Error instead of reducing the TD-error
- They achieve this but effectively double the computational complexity

Emphatic-TD methods.

- These methods rewrite the state transitions using importance sampling s.t. they are appropriate for learning the target policy
  - It does this while using the behaviour policy distributions

These methods however are not explored in these classes.

- They can be explored for your major research task

SIT796 Reinforcement Learning

# Feature Construction

DEAKIN UNIVERSITY

Presented by:
Thommen George Karimpanal
School of Information Technology

# Function Construction

For function approximation, we need to convert the state-value function to feature vectors.

This can be done using any of the following:

- Polynomial features
- Fourier basis
- Coarse coding
- Tile coding
- Sparse coding
- Dictionary Learning

Consider the state $s$ represented to be represented by a vector $\mathbf{x}(s) = [x_1(s), x_2(s), \cdots, x_d(s)]^\top$, where $|\mathbf{x}(s)| = |\mathbf{w}|$

We can represent our state-value function using the inner product of $\mathbf{w}$ and $\mathrm{x}(s)$

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

# Polynomial Features

In a polynomial feature, $\mathbf{x}(s)$ is a polynomial basis, that is, for a set of states $s = \{s_1, s_2, s_3, \dots, s_k\}$, the polynomial features can be written as

$$x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}$$

where $c_{i,j}$ is a non-negative integer such that $c_{i,j} = \{0, 1, 2, \dots, n\}$

Thus, we have

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i \prod_{j=1}^{k} s_j^{c_{i,j}}$$

and $x_i(s)$ is a n-order polynomial basis in a k-dimensional space spanned by $(n+1)^k$ different features

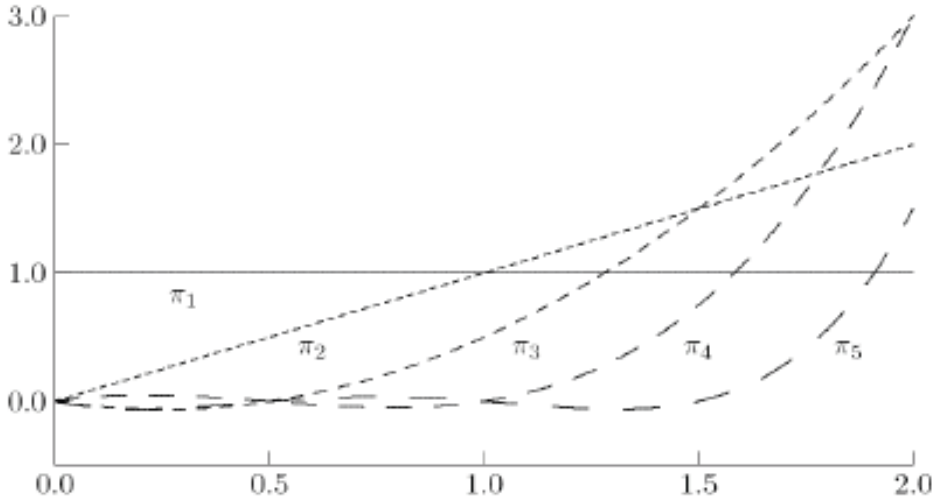
# Polynomial Feature Example

There are several kinds of polynomial bases, such as:

- Lagrange
- Newton
- Orthogonal polynomials
- Chebyshev

The Newton basis functions are:

$$\pi_k(t) = \prod_{j=1}^{k-1}(t - t_j)$$




Which gives, in the third order case, the following

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & t_2 - t_1 & 0 \\ 1 & t_3 - t_1 & (t_3 - t_1)(t_3 - t_2) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

SIT796 Reinforcement Learning

# Fourier Basis

Presented by:
Thommen George Karimpanal
School of Information Technology

# Fourier Transform

- Polynomials are not the best - unstable and not very physically meaningful.
- Easier to talk about “signals” in terms of its “frequencies” (how fast/often signals change, etc).

- **Any** periodic function can be rewritten as a weighted sum of Sines and Cosines of different frequencies (Jean Baptiste Joseph Fourier, 1807).

The aim is then to understand the frequency $\omega$ of our signal.
So, let’s reparametrize the signal by $\omega$ instead of *x*:


For every $\omega$ from 0 to inf, ***F(ω)*** holds the amplitude *A* and phase $\phi$ of the corresponding sine

$$A\sin(\omega x + \phi)$$

# Fourier Transform

Represent the signal as an infinite weighted sum of an infinite number of sinusoids

$$F(u) = \int_{-\infty}^{\infty} f(x) e^{-i 2\pi u x} dx$$

Note: $e^{ik} = \cos k + i \sin k \qquad i = \sqrt{-1}$

Arbitrary function ⟶ Single Analytic Expression

Spatial Domain (*x*) ⟶ Frequency Domain (*u*)

(Frequency Spectrum *F(u)*)

Inverse Fourier Transform (IFT)

$$f(x) = \int_{-\infty}^{\infty} F(u) e^{i 2\pi u x} dx$$

# Fourier Transform

This means that F can encode both using the imaginary numbers.

Consider

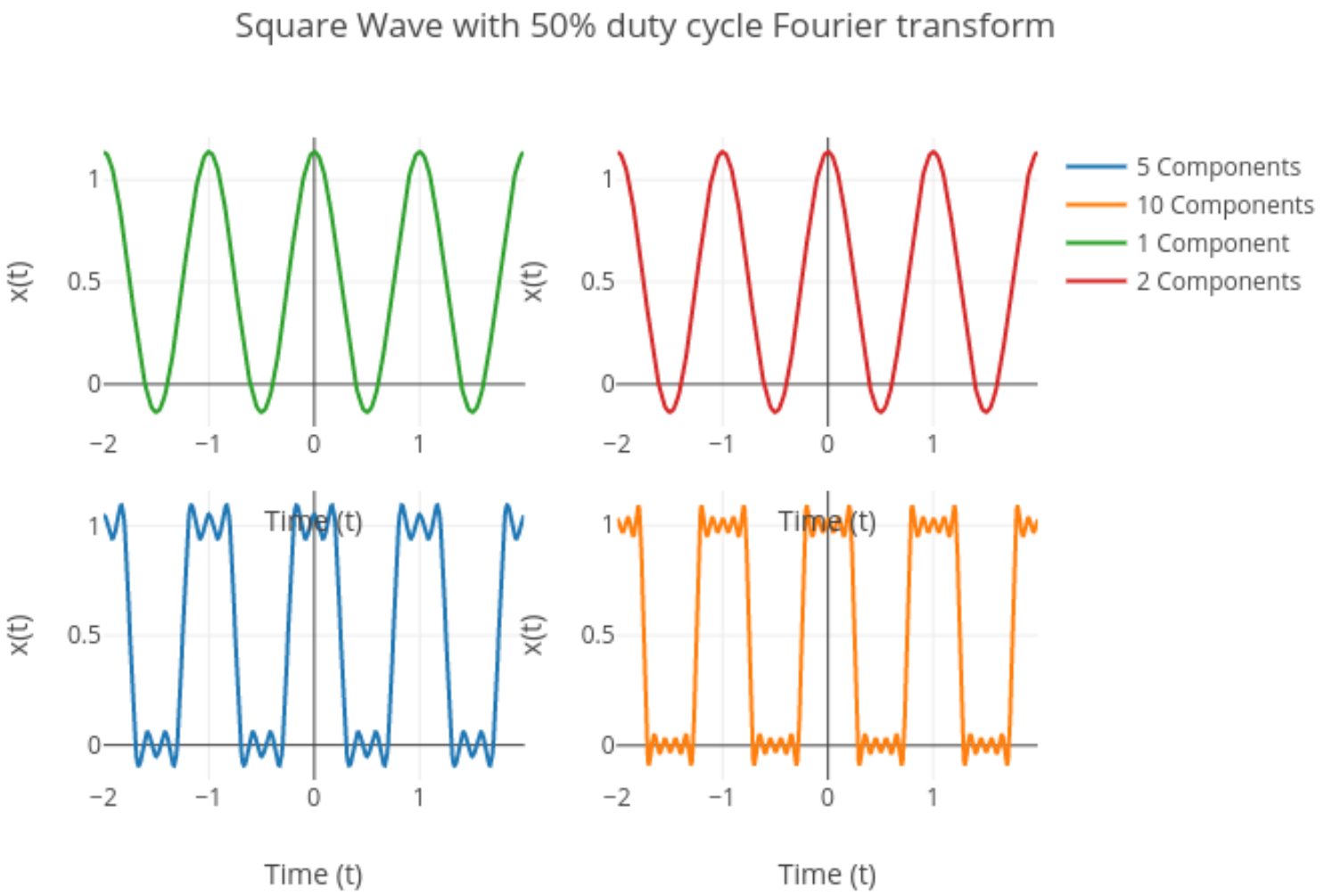
$$F(\omega) = R(\omega) + iI(\omega)$$

The amplitude is given by

$$A = \pm\sqrt{R(\omega)^2 + I(\omega)^2}$$

and the phase is

$$\phi = \tan^{-1}\frac{I(\omega)}{R(\omega)}$$

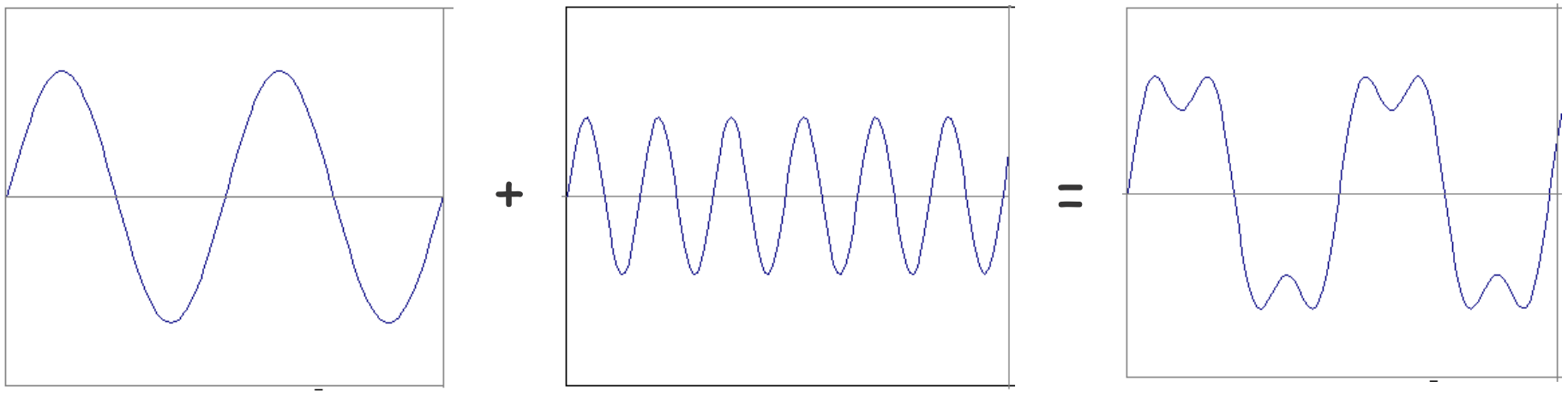


# Fourier Features

In a Fourier basis, the set of states $\mathbf{s} = [s_1, s_2, s_3, \ldots, s_k]^{\mathrm{T}}$ can be expressed using a cosine in the following way

$$x_i(s) = \cos(\pi s^{\mathrm{T}} \mathbf{c}^i)$$

where $\mathbf{c}^i = \left[c_1^i, \ldots, c_k^i\right]^{\mathrm{T}}$ such that $c_j^i \in \{0, 1, 2, \ldots, n\}$



Thus, we have:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^{\mathrm{T}} \mathbf{x}(s) = \sum_{i=1}^{d} w_i \cos(\pi s^{\mathrm{T}} \mathbf{c}^i)$$

SIT796 Reinforcement Learning

# Coding Methods

DEAKIN UNIVERSITY

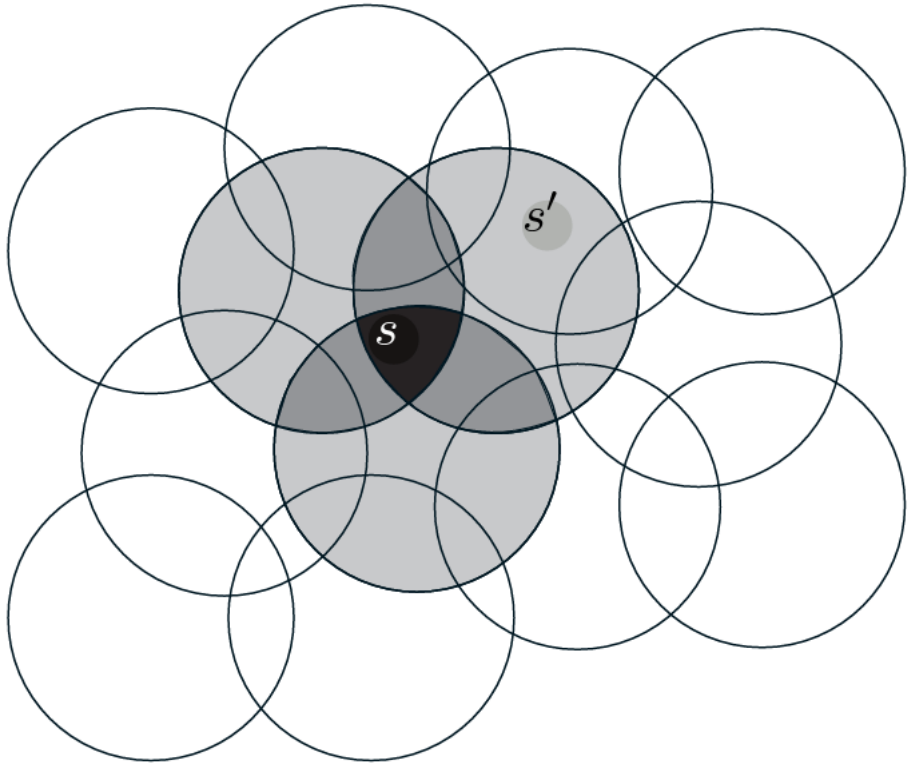
Presented by:
Thommen George Karimpanal
School of Information Technology

# Coarse Coding

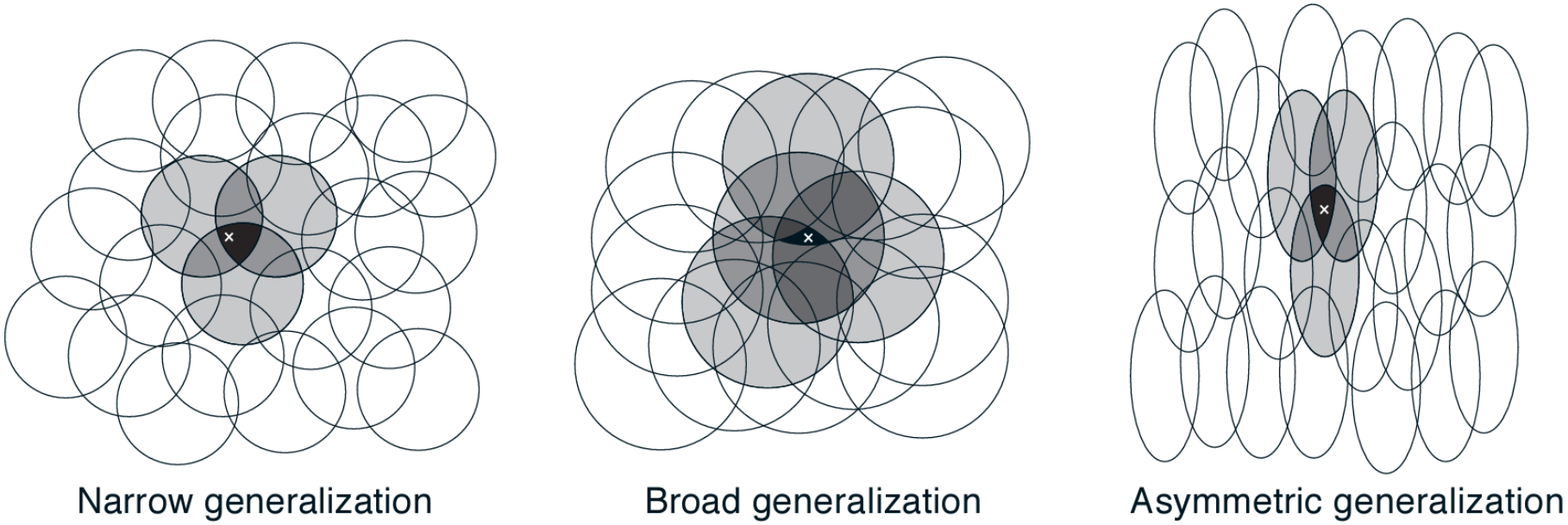
- Coarse coding is used for continuous spaces, but can also be used for state-spaces that are binary or to further encode other features.
- Each ball or sphere is usually called a receptive field.
- Features with large receptive fields give broad generalization, but might yield very coarse approximations
- The trade-off is often a question to be solved for their implementation:

  -Too coarse may not be discriminative enough

  -Too large and the complexity may increase too much

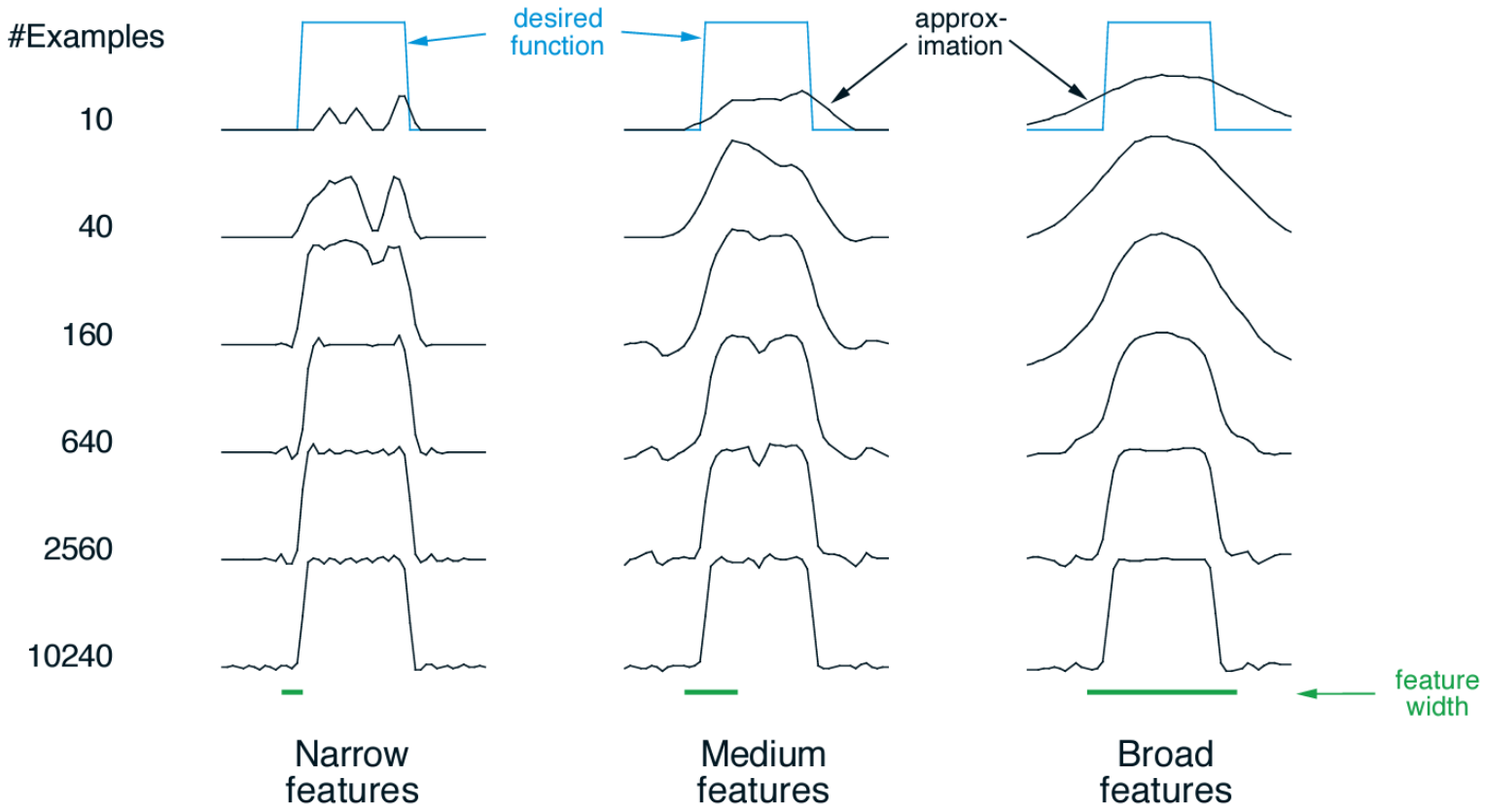


# Coarse Coding



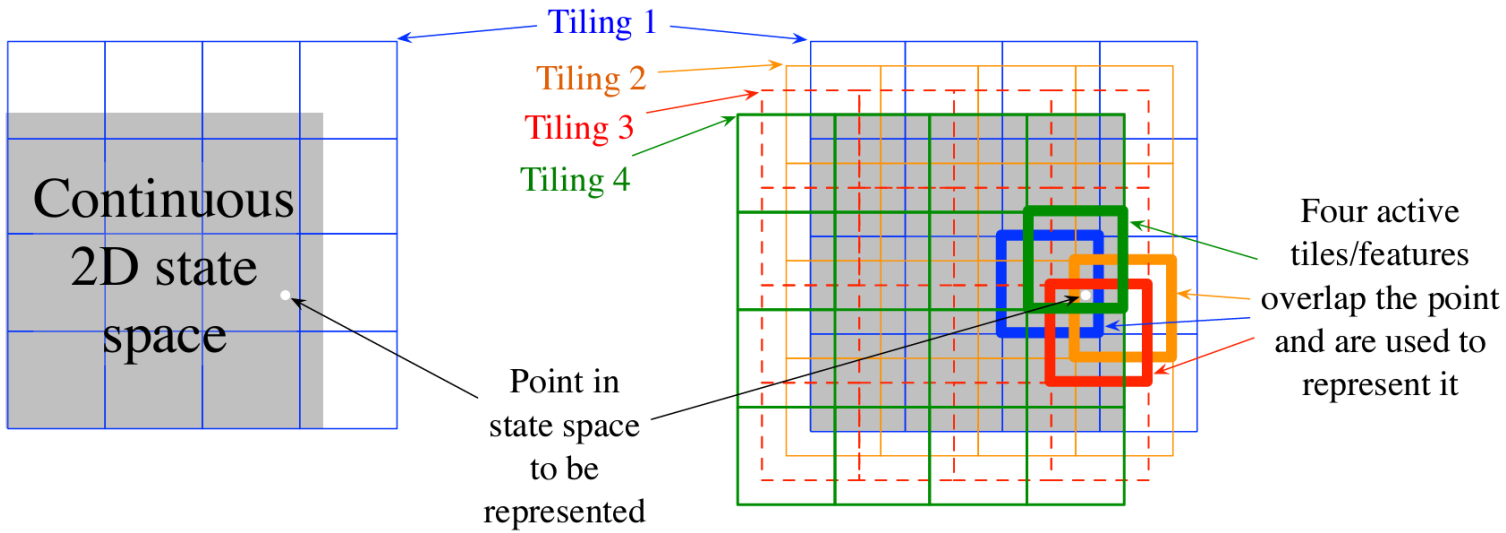

The type of receptive field affects the nature of generalisation

# Coarse Coding



# Tile Coding

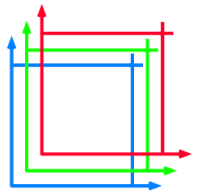
- Makes use of tiles and tilings
- Tiles are elements of tilings
- If only 1 tiling is used – state aggregation
- Many tilings are used to obtain the ability to represent the state space as finely or coarsely as required (Generalisation)
- Uses binary features
- Tilings are offset from each other uniformly in each dimension

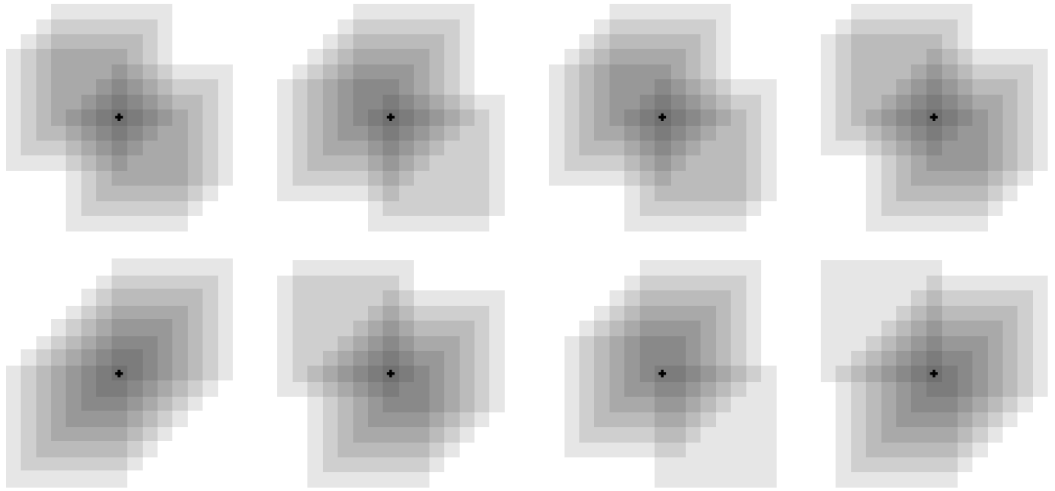


# Tile Coding

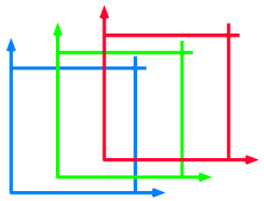
- Tilings are offset from each other uniformly in each dimension
- Diagonal elements become too prominent
- To fix this, the offset can be done unsymmetrically

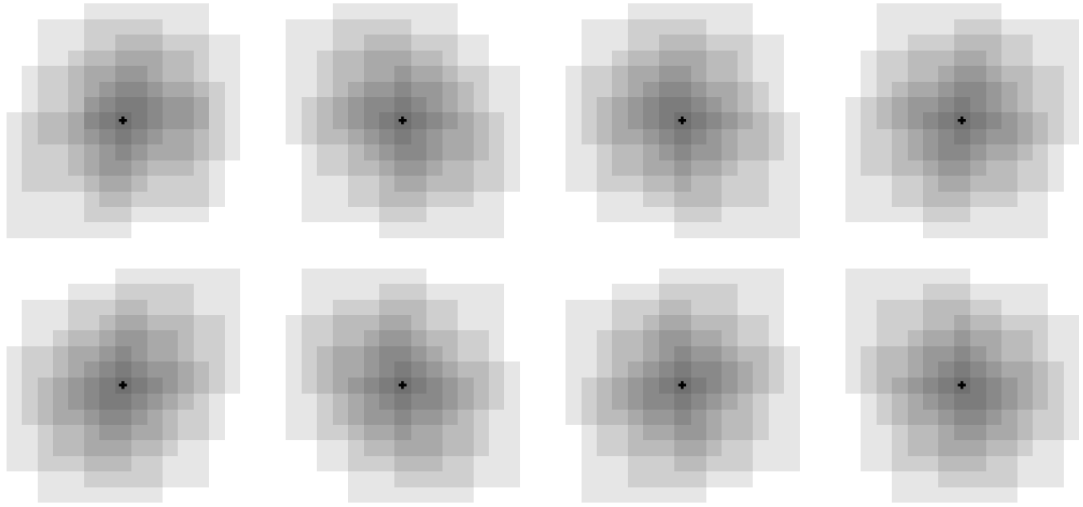
Possible generalizations for uniformly offset tilings





Possible generalizations for asymmetrically offset tilings





# Readings

This lecture focused on exploring how we can use function approximation in RL.

- Future topics will expand on this topic by looking at Deep RL.
- Ensure you understand what was discussed here before doing the following topics

For more detailed information see Sutton and Barto (2018) Reinforcement Learning: An Introduction. Several illustrations here were borrowed from the book.

- *Chapter 9.5: Feature Construction for Linear Methods*
- http://incompleteideas.net/book/RLbook2020.pdf

Other readings:

- Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti Zadeh (Eds.), “Feature Extraction: Foundations and Applications”, Springer, 2006.
- http://www.causality.inf.ethz.ch/ciml/FeatureExtractionManuscript.pdf
- https://sites.fas.harvard.edu/~cs278/papers/ksvd.pdf