Constructing Features for Prediction

By Rohit Pardasani

References:

https://mitpress.mit.edu/books/reinforcement-learning

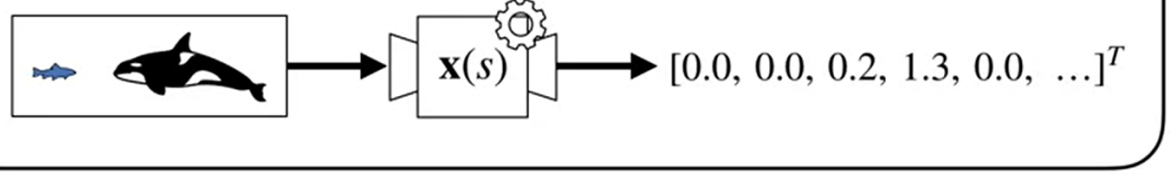
https://www.coursera.org/specializations/reinforcement-learning

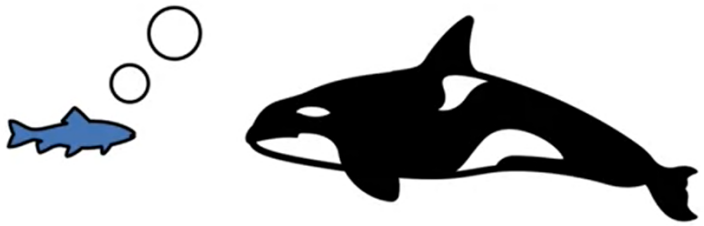
Coarse Coding

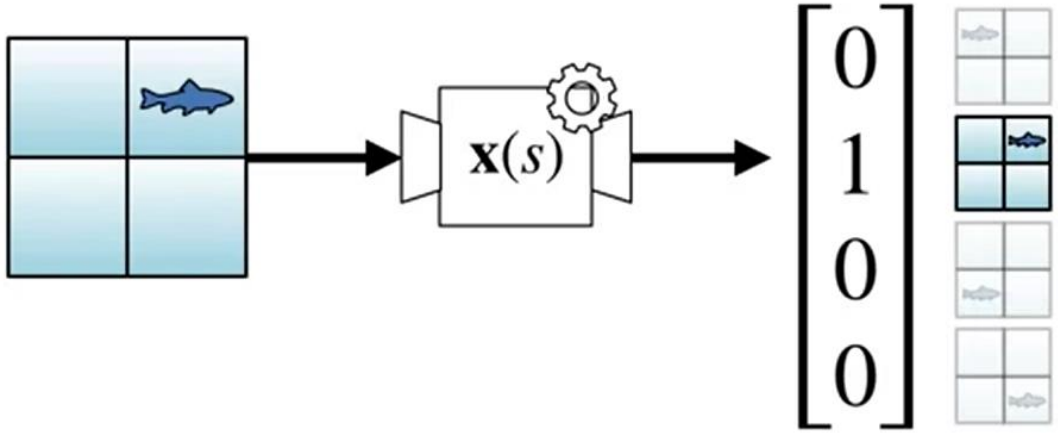
The feature is used to construct value estimates are one of the most important parts of a reinforcement learning agent.
Coarse coding is one simple but effective way to do this.
Coarse coding relates to state aggregation.

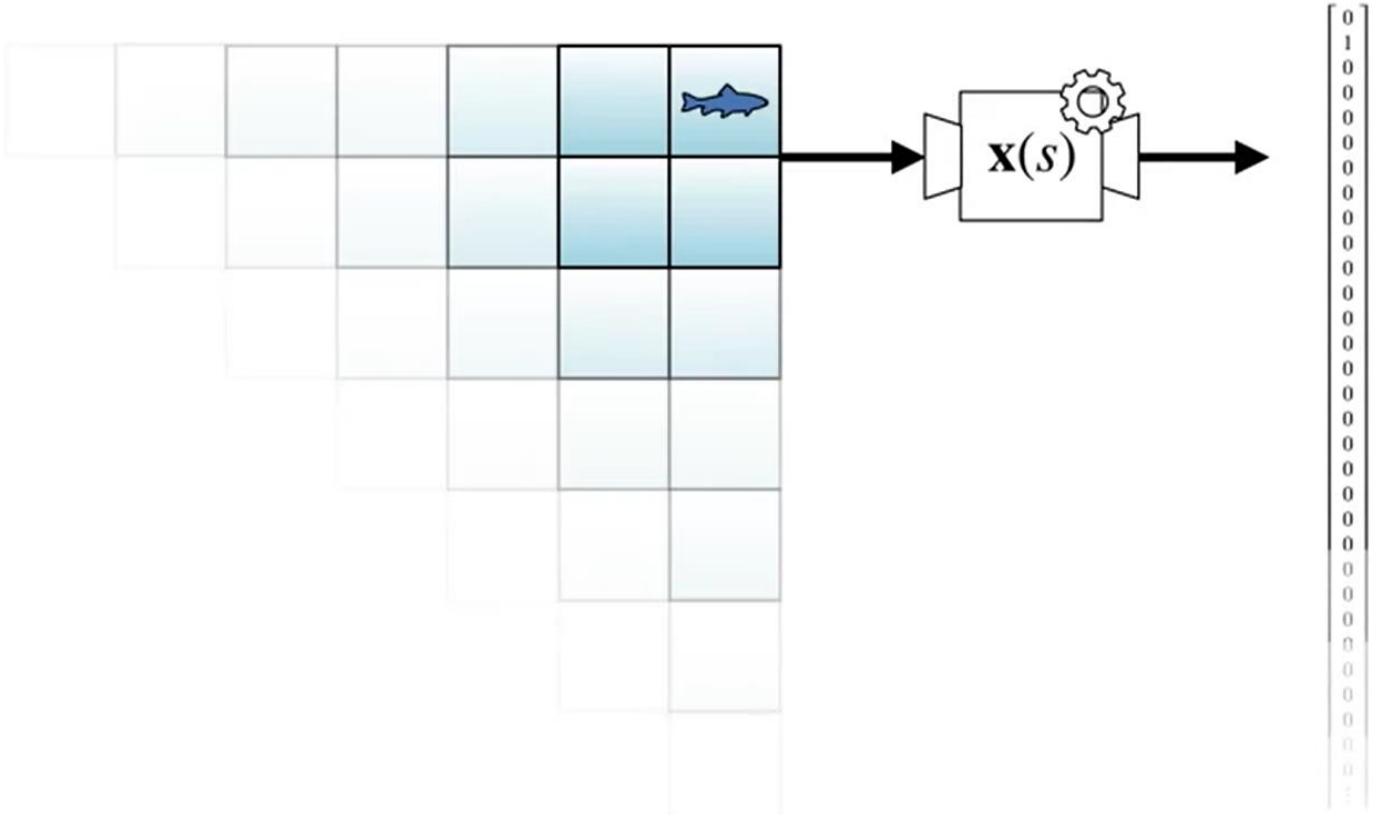
Recall linear value function approximation

$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$$



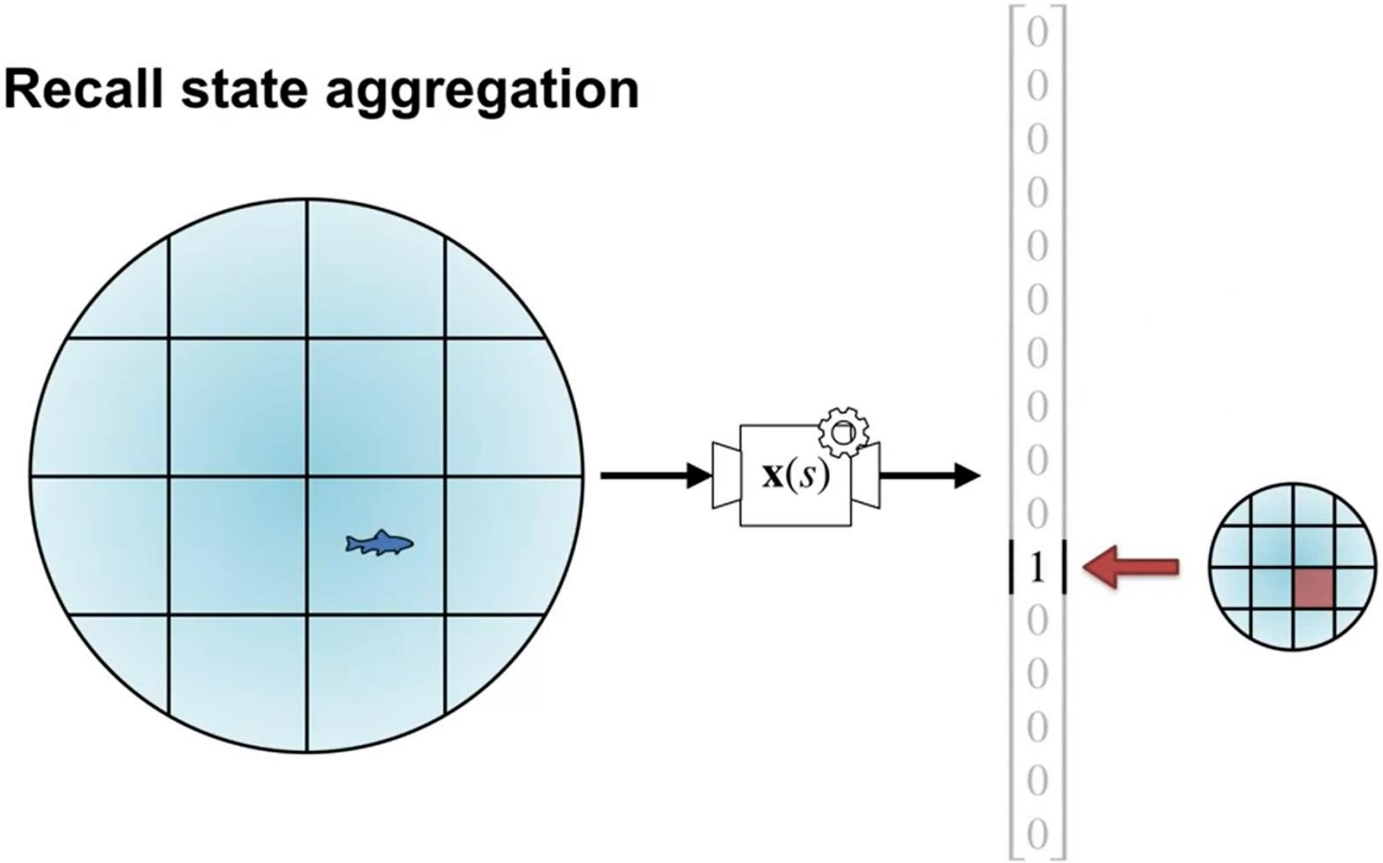






This is not feasible when the size of the state space becomes much larger than the agent's available memory.

Recall state aggregation



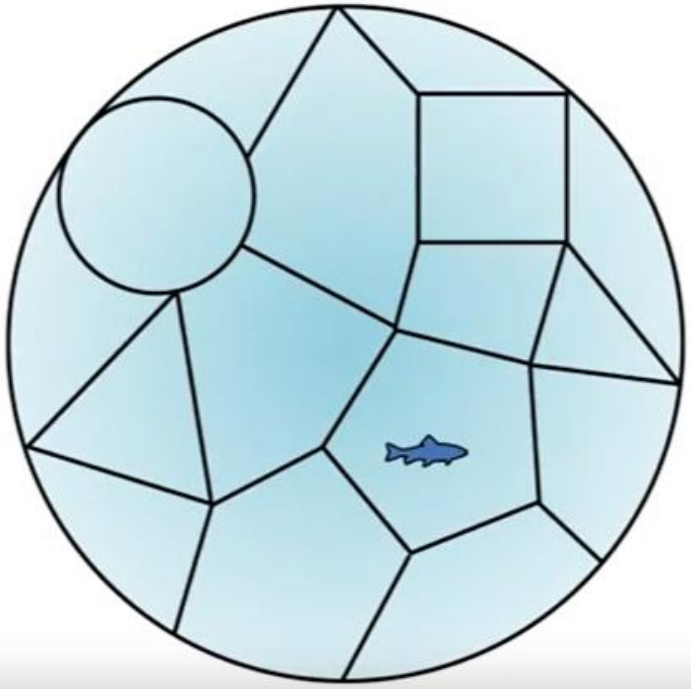
The fish can be one of infinitely many locations.

It's impossible to represent all these locations with a finite lookup table.

Recall that we can use state aggregation to associate nearby states with the same feature.

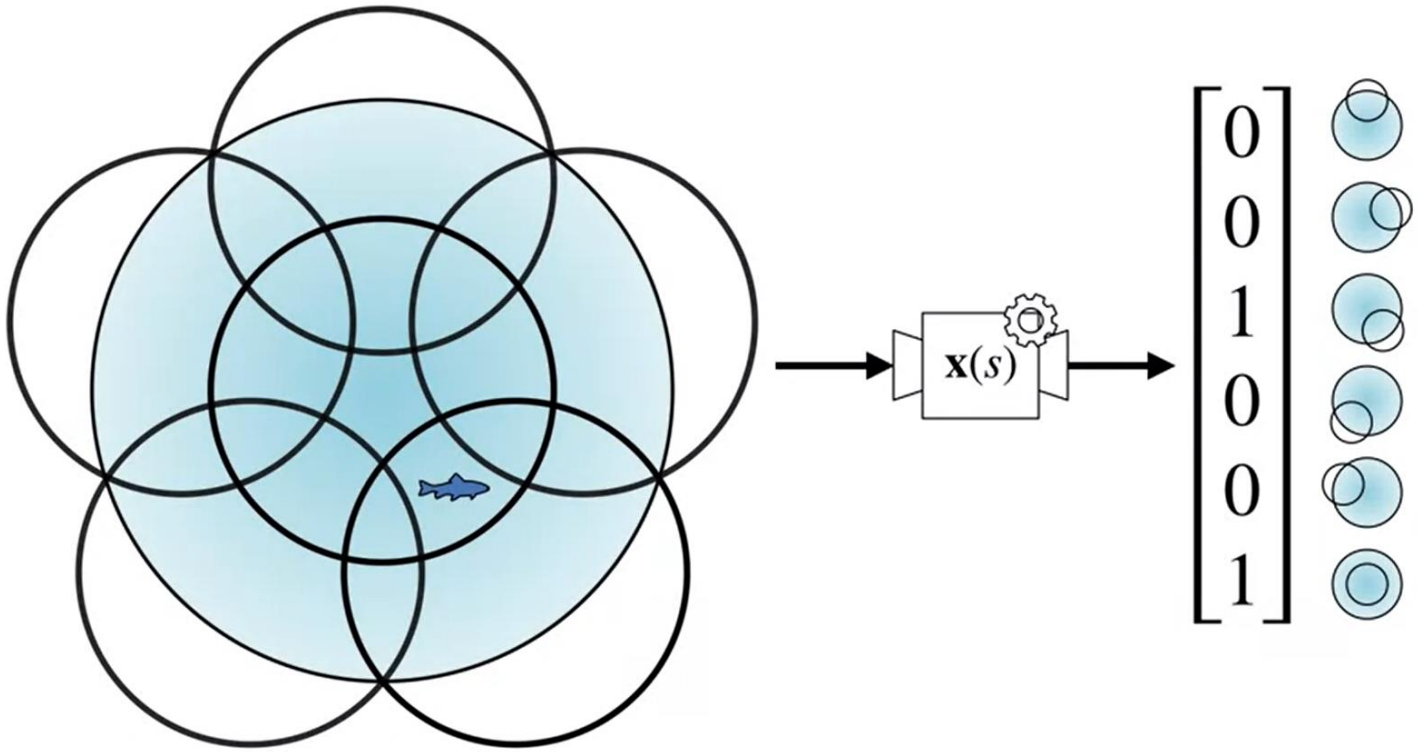
This is like treating all states within each square as the same state.

State aggregation → coarse coding



In general, we can aggregate states using any shapes we want if those shapes do not have any gaps or overlap.

Coarse coding



Summary

- Tabular states can be represented with a binary one-hot encoding
- Coarse coding is a generalization of state aggregation

State aggregation does not usually allow the shapes to overlap.

But this restriction is not necessary.

In fact, by allowing overlap, we obtain a more flexible class of feature representations called course coding.

All the ideas we've discussed so far are not limited to 2D state spaces.

Coarse coding can also be applied to higher dimensional inputs.

Generalization Properties of Coarse Coding

Coarse coding group states into features of arbitrary shapes and sizes.

They can be circles, ellipses, squares or a combination of different shapes.

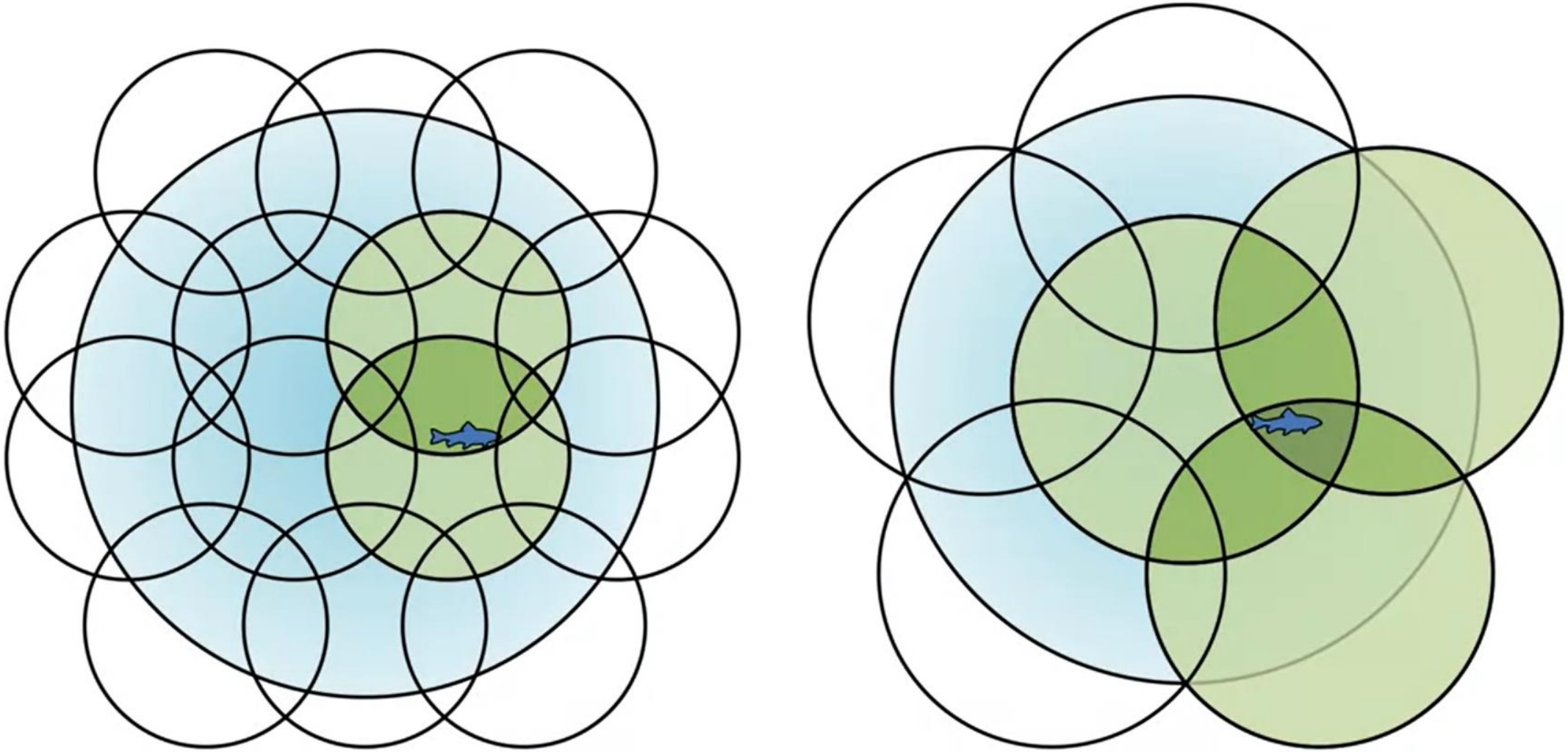
Changing the shapes and sizes of features impacts generalization and discrimination and so affects the speed of learning and the value functions we can represent.

Performing an update to the weights in one state changes the value estimate for all states within the receptive fields of the active features.

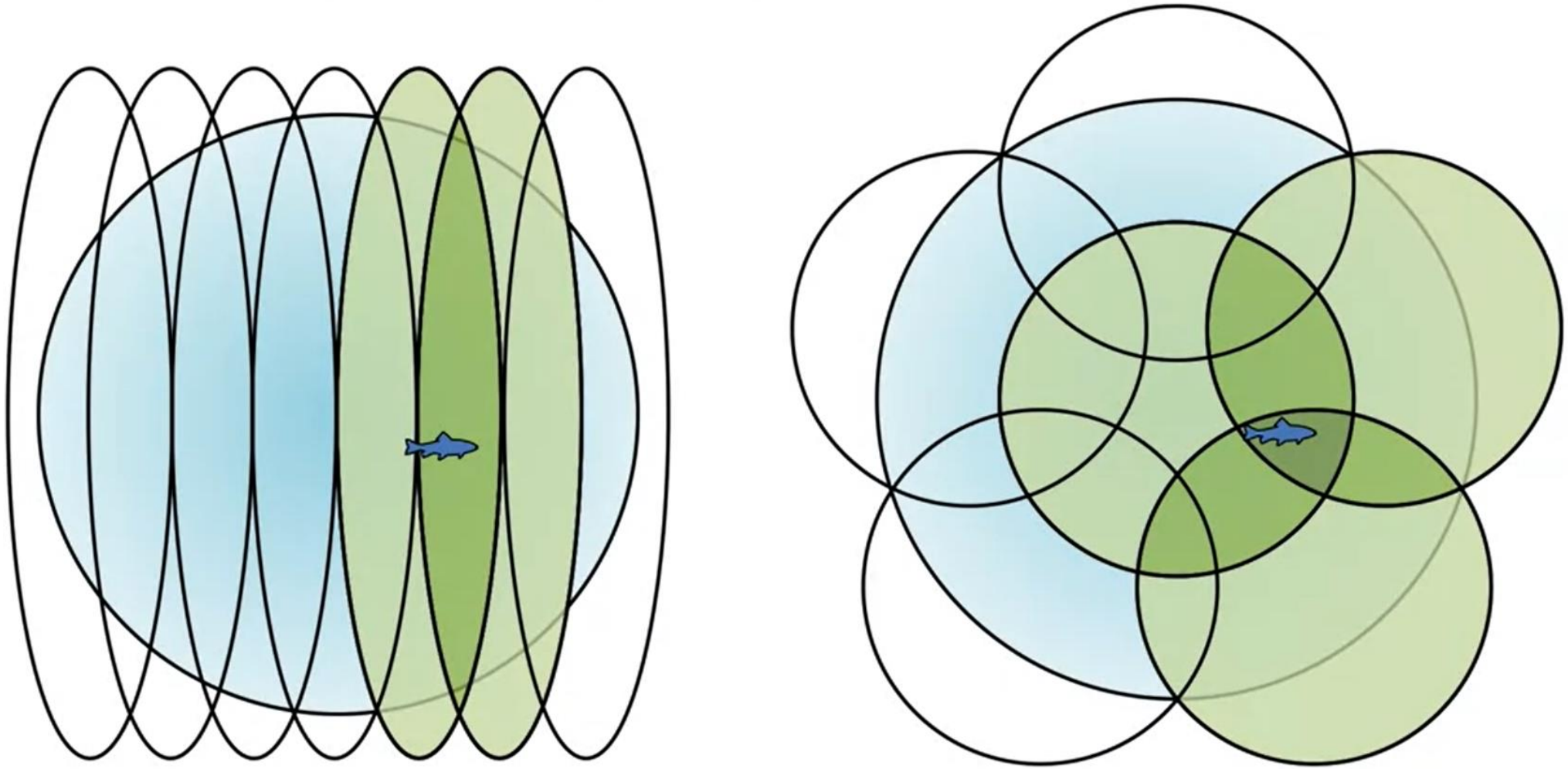
If the union of the receptive fields for the active features is large, the feature representation generalizes more.

Conversely, if the union is small, there's little generalization.

Broadness of generalization



Direction of generalization



State Discrimination

Shape and size of the receptive fields impact generalization and so the speed of learning.

But what about the final accuracy of our estimates?

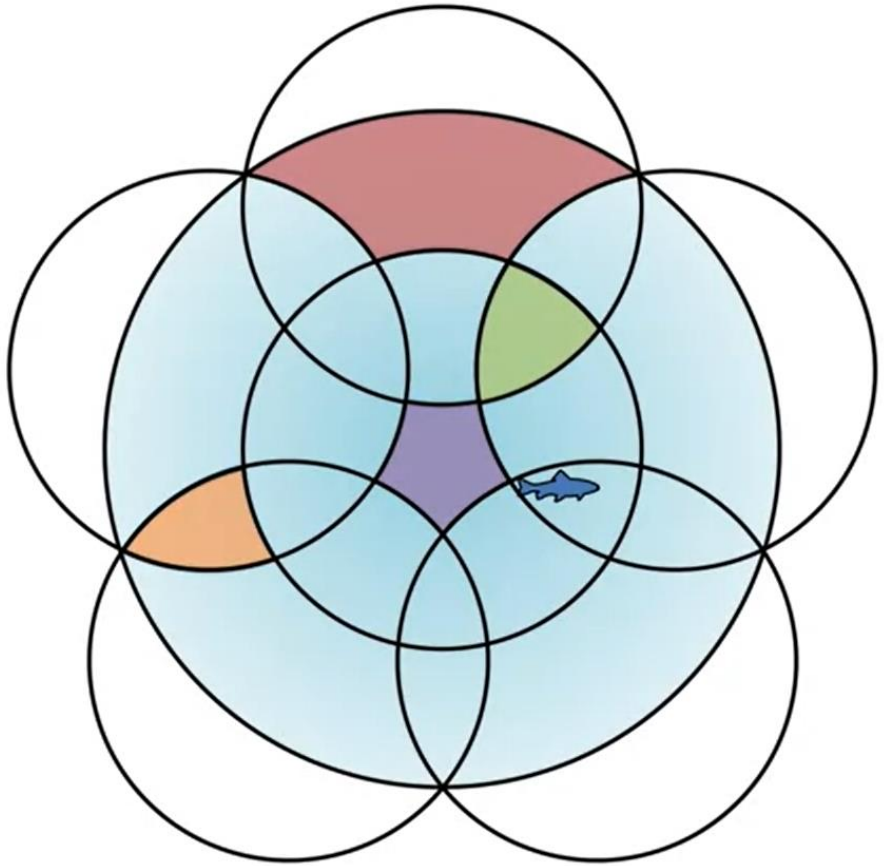
This is where discrimination comes in.

The ability to distinguish between values for two different states is called discrimination.

In coarse coding, the overlap between circles dictates the level of discrimination.

It is impossible to do perfect discrimination because we can never update the value of one state without impacting the values of other states.

State discrimination



The colored shapes depict the discriminative ability of
this coarse coding.

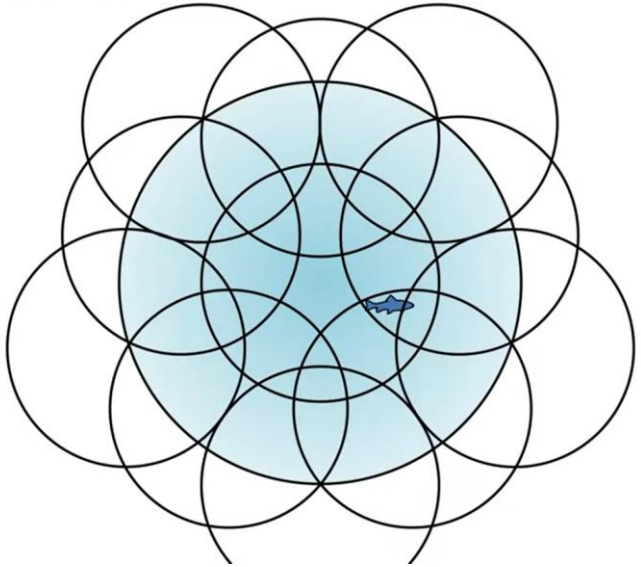
We've only highlighted a few regions to keep the visualization simple.

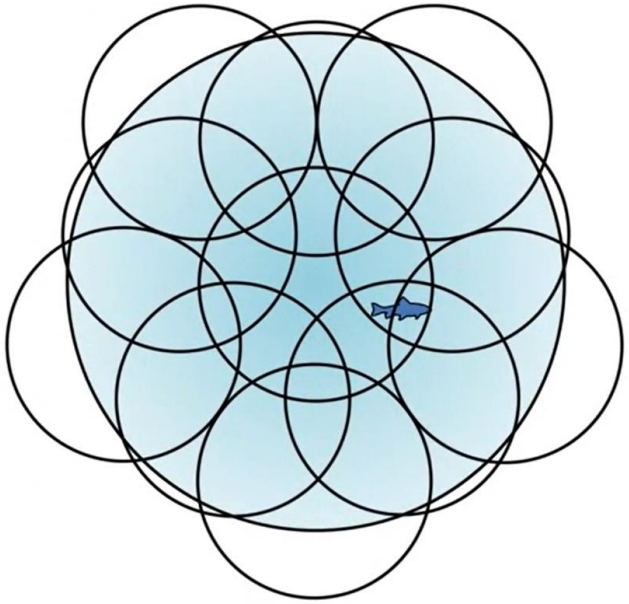
Every state within the same-colored shape will have the exact same feature vector.

As a result, they must all have the same approximate value.

The smaller these regions are, the better we can discriminate.

State discrimination





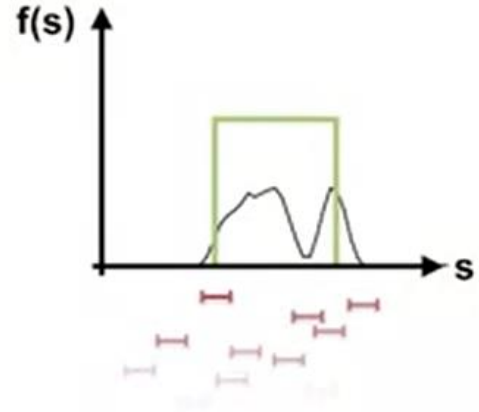
With many circles, the regions becomes smaller and we can discriminate more finely between the values of different states or we can make the circles smaller.

So, the size, number, and shape of the features all affect the discriminative ability of the representation

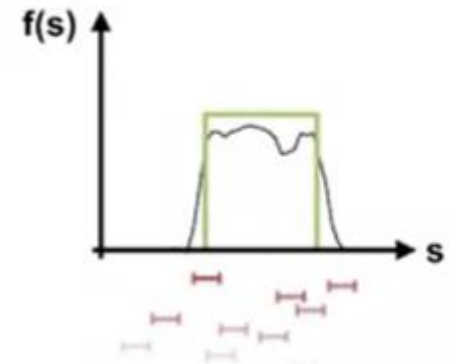
Summary

- The size, number, and shape of the features affects generalization
- The resulting shape intersections affect the ability to discriminate

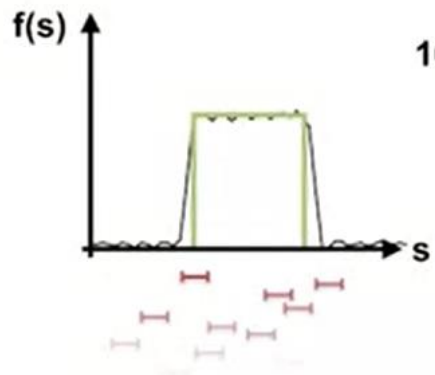
1D Example



40 Samples

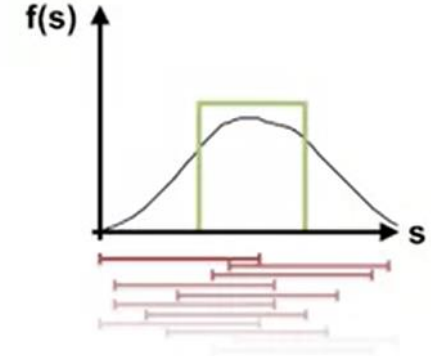


160 Samples

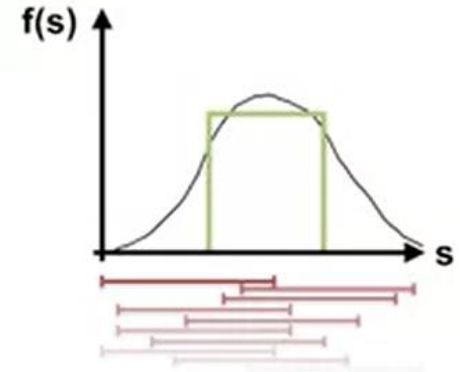


10240 Samples

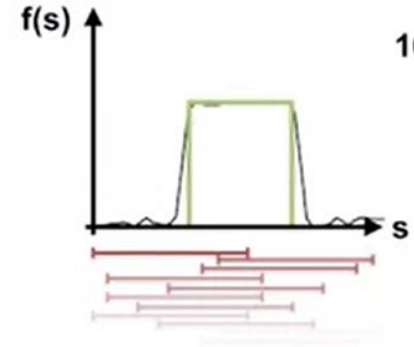
1D Example



40 Samples

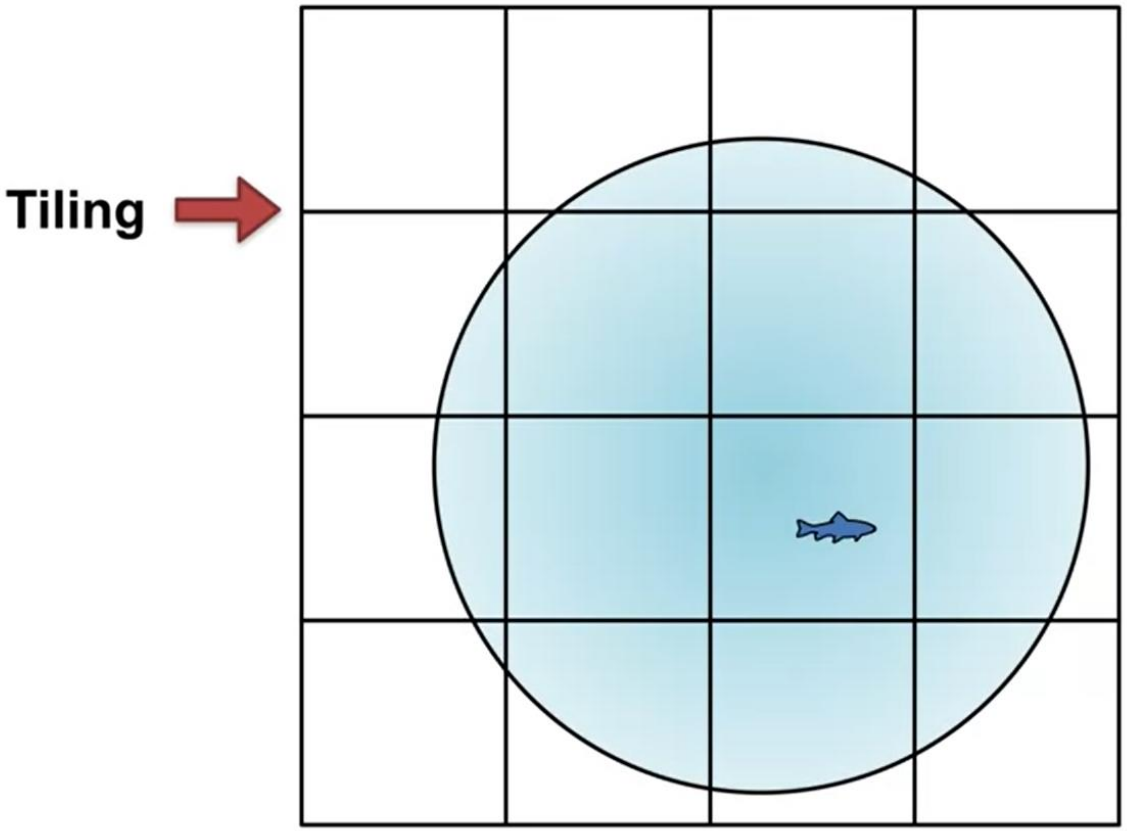


160 Samples



10240 Samples

Tile Coding



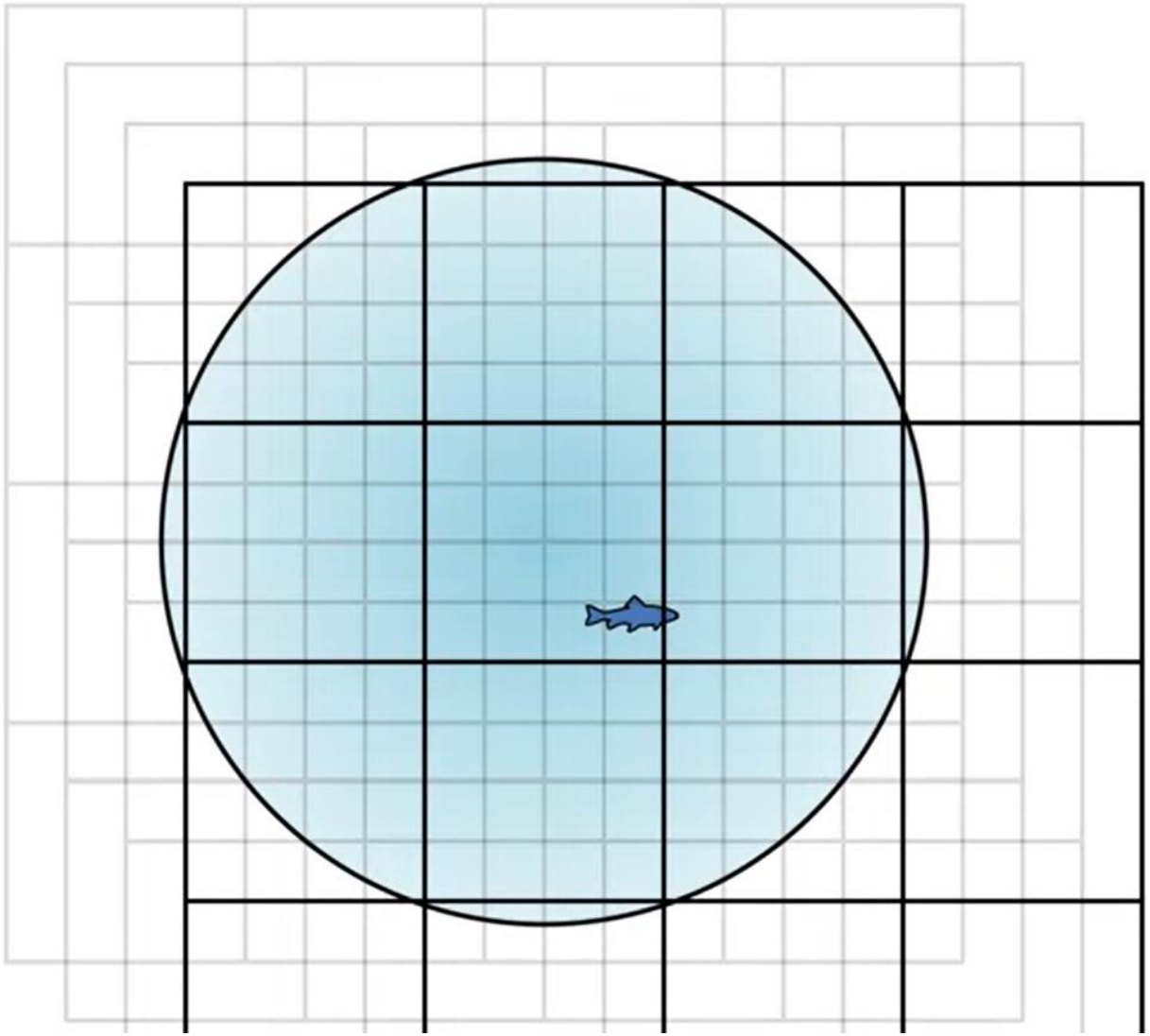
Tile coding uses squares.

Most convenient way to lay out a bunch of squares over space, is a grid. Let's call this grid a tiling.

So far using this is just state aggregation.

Larger tiles will result in increased generalization.

Although the ideal tile size depends on the specific problem,
it's generally a good idea to use larger tiles.



To improve the discriminative ability of our tile coding, we can put several tilings on top of each other.

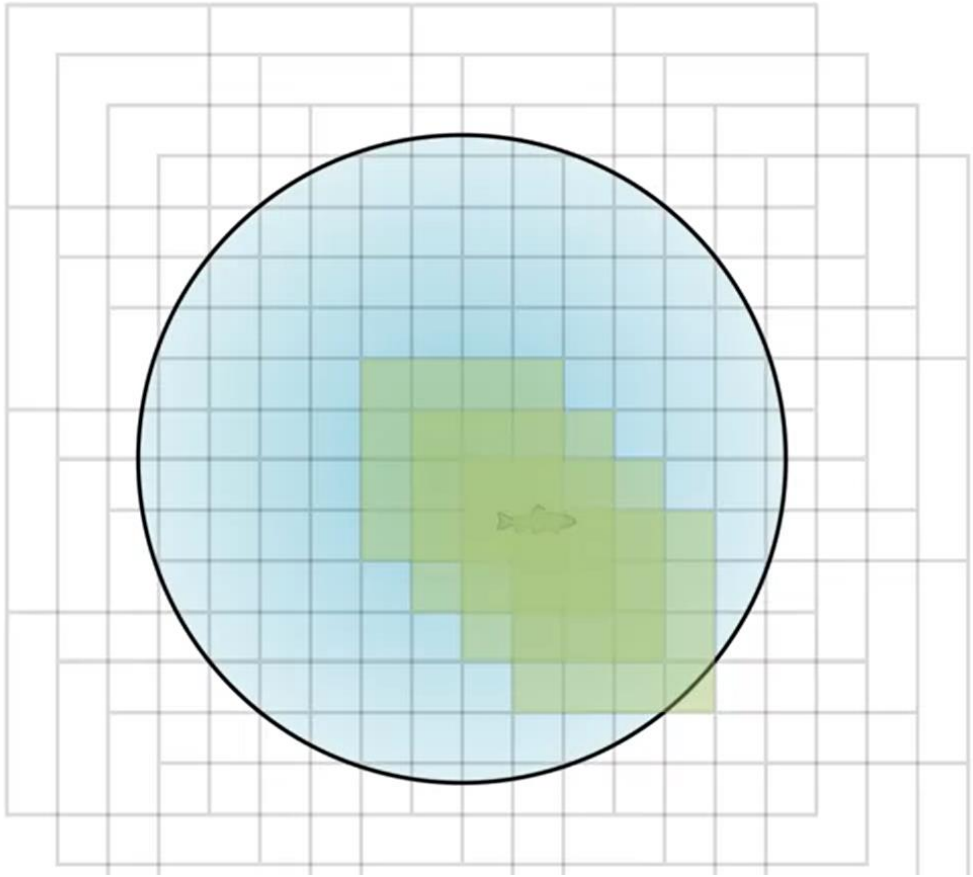
Each tiling is offset by a small amount.

Offsetting each layer of tiling
creates many small intersections.

This results in better discrimination. In practice, it is useful to use a large number of tilings.

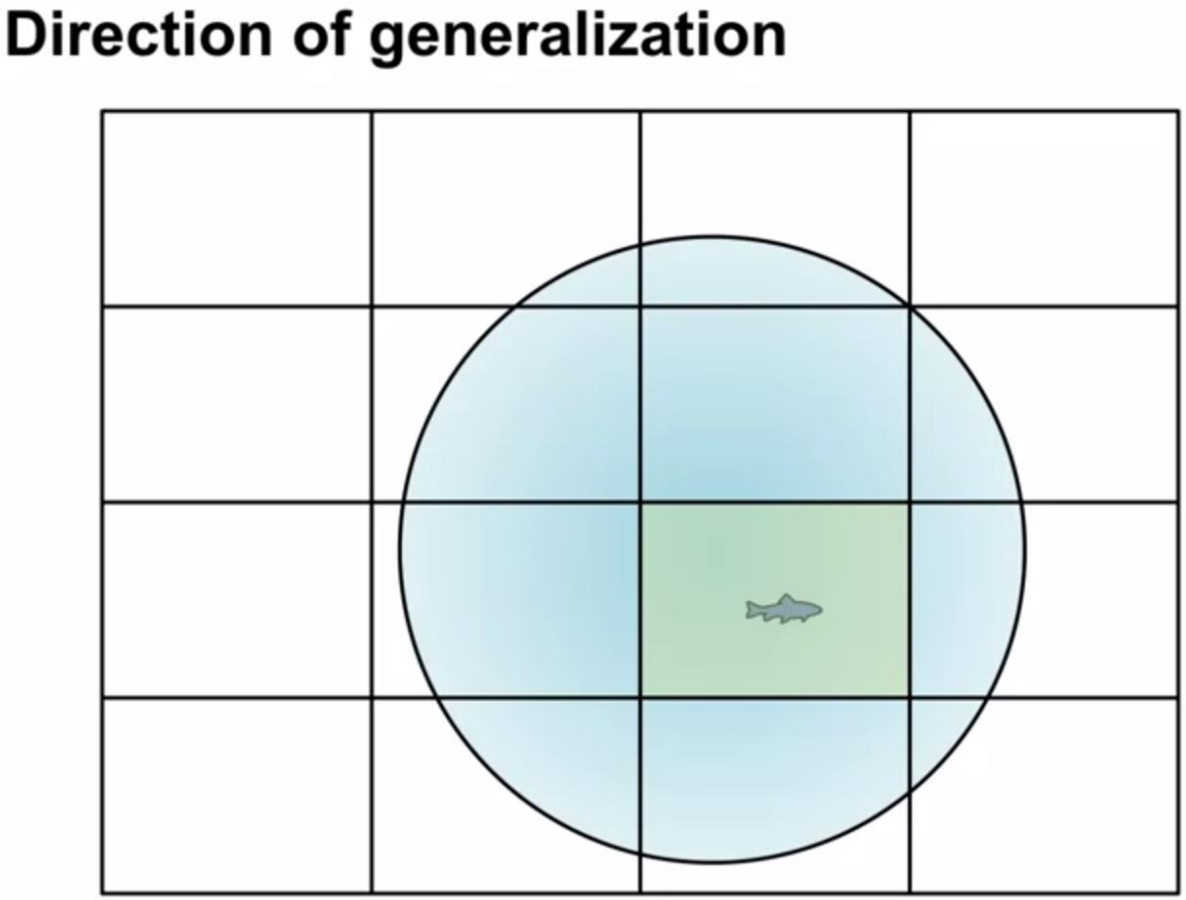
For one tiling, generalization only occurs within the square,
but with multiple tilings, updates in this state generalize to these other states.

Shape of generalization



The generalization in this case is diagonal.

If we had used random offsets, then the generalization would be more spherical and homogeneous.



Let's talk about how to control the generalization properties further.

We can do this by creating a grid of rectangles rather than squares.

An efficient way to do this is to scale each dimension of the state space.

The environment appears to have gotten squished here.

Actually, layering squares over this squished environment is like laying rectangles over the unsquished environment.

By using rectangles, we can control the broadness of the generalization across each dimension of the state-space.

Tile coding can represent a wide range of functions, but its utility does not end there. Tile coding is also computationally efficient.

Since grids are uniform, it's easy to compute which cell the current state is in.

Due to its computational efficiency, tile coding can be used to quickly run preliminary experiments in low dimensional environments.

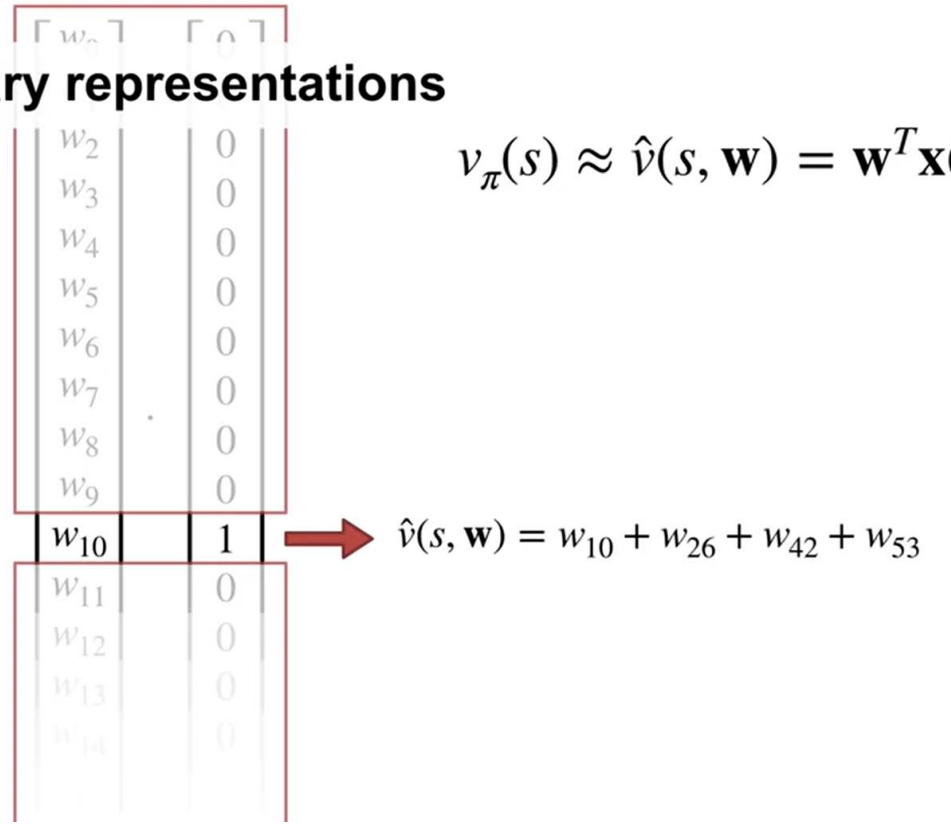
However, as the number of dimensions grows, the number of required tiles grows exponentially.

As a result, it can be necessary to tile input dimension separately.

Whether or not input dimensions can be treated independently depends on the specific problem.

Using Tile Coding in TD

Sparse binary representations



$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$$

$$\hat{v}(s, \mathbf{w}) = w_{10} + w_{26} + w_{42} + w_{53}$$

The number of active tiles is always significantly less than the number of total tiles.

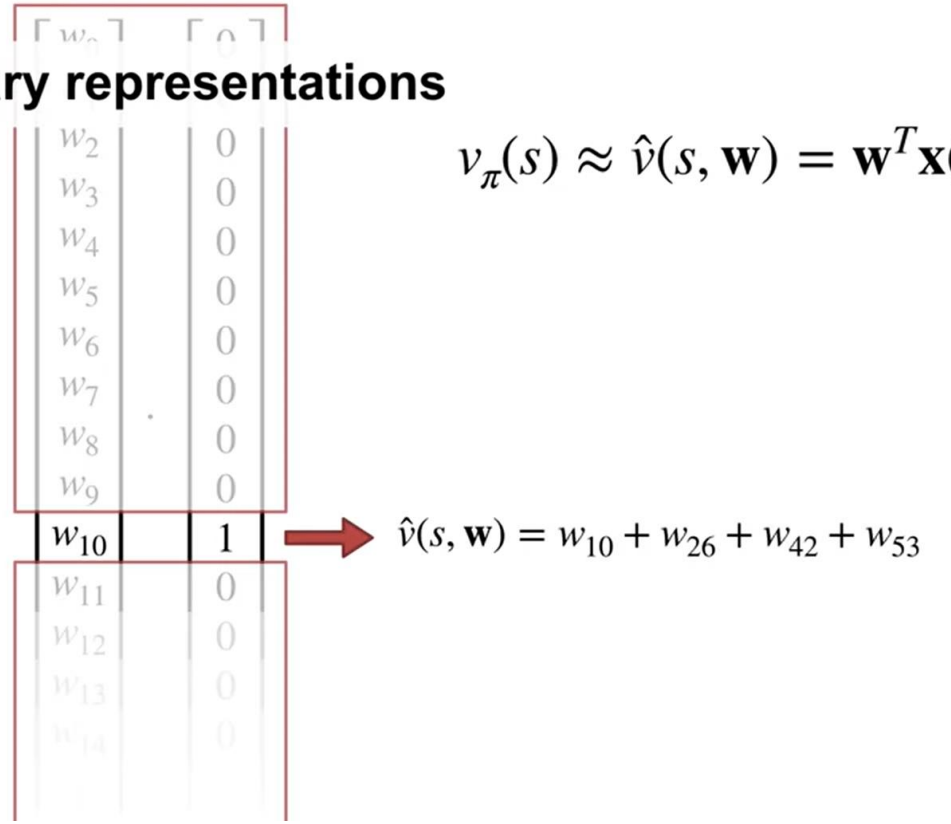
We can use this to calculate the value function efficiently.

With linear function approximation, the value function is a dot product between a weight vector and a feature vector.

Let's see what this dot product looks like with a sparse binary feature vector.

If we were to multiply the two vectors element-wise, many of the element-wise products will be zero.

Sparse binary representations



$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$$

$$\hat{v}(s, \mathbf{w}) = w_{10} + w_{26} + w_{42} + w_{53}$$

This means we only have to consider the weights at the non-zero elements of the feature vector because the features are binary, the weights at the non-zero features are multiplied by one.

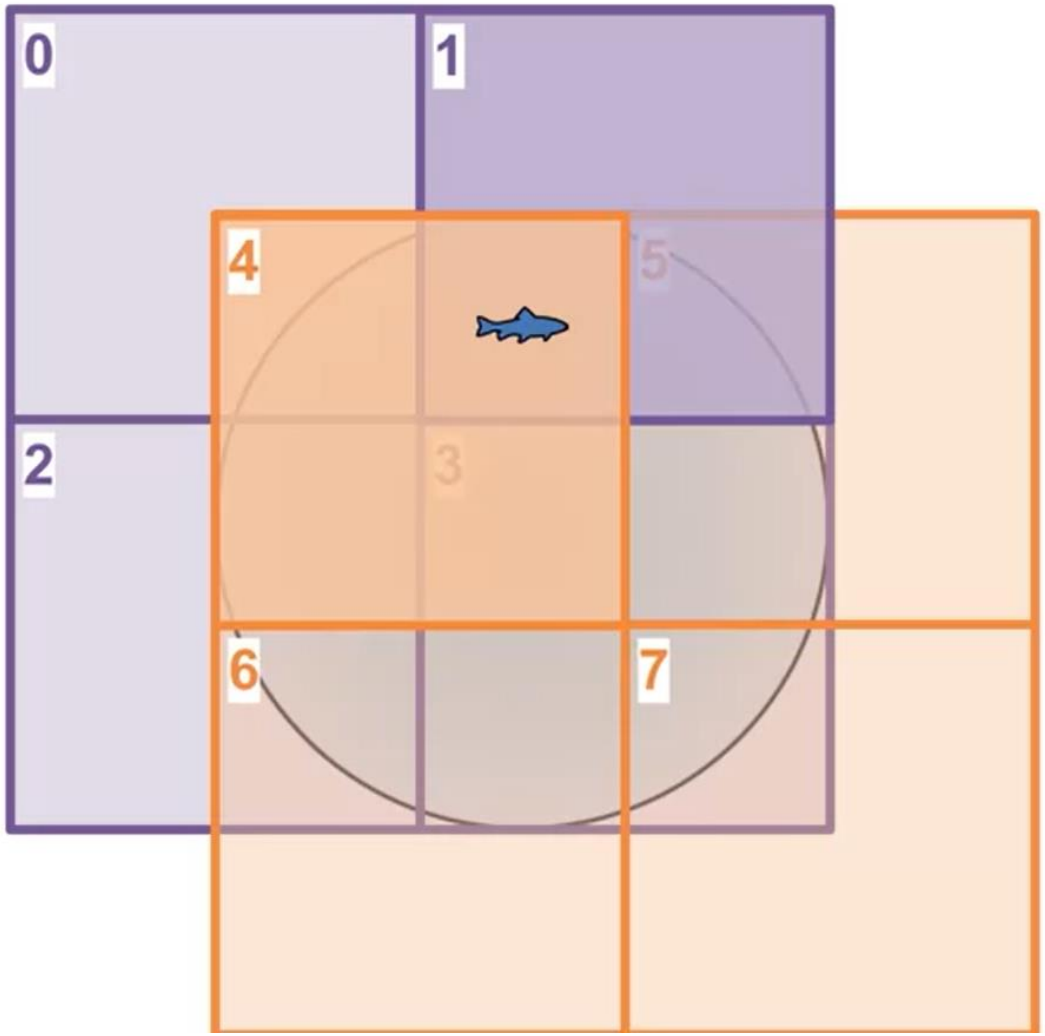
Computing the dot product in the usual way would be expensive.

Instead, we can just sum the weights corresponding to the active features.

Note that this takes a certain amount of time because the number of active features is the same in every state.

The feature vectors produced by tile coding may query in the value function cheap computationally.

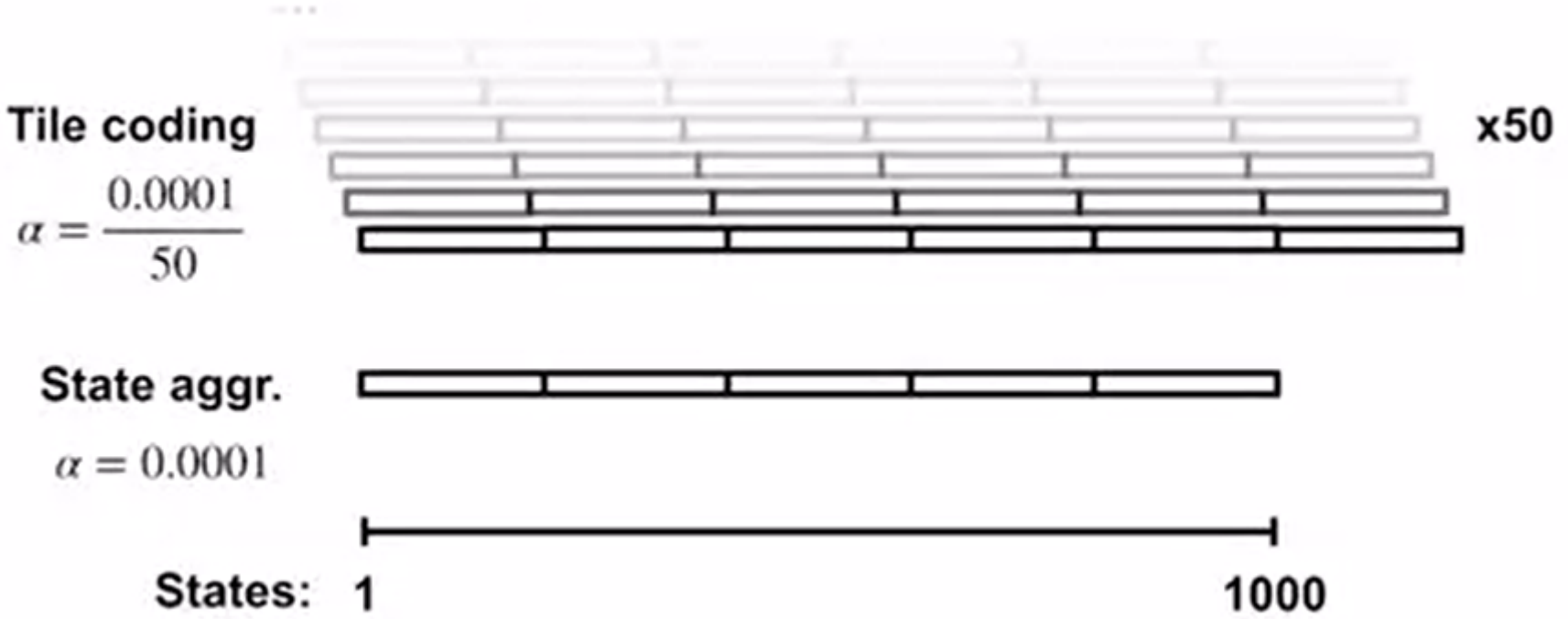
A simple example



$$\mathbf{w} = \begin{bmatrix} 0.0 \\ 1.5 \\ 1.1 \\ 0.2 \\ 0.5 \\ -0.3 \\ 1.3 \\ -0.7 \end{bmatrix} \qquad \mathbf{x}(s) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = 1.5 + 0.5 = 2$$

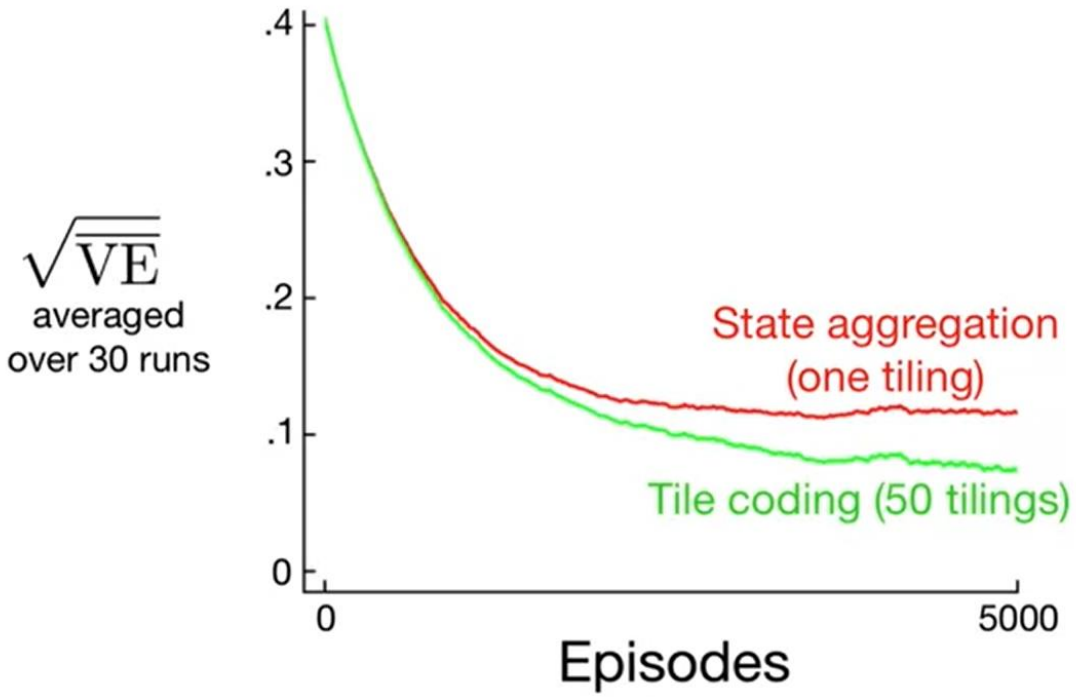
Random Walk : Tile Coding vs State Aggregation



Feature Vector Size = 6 x 50 = 300

Feature Vector Size = 5

Tile coding vs state aggregation



Both agents learn quickly due to aggressive generalization.

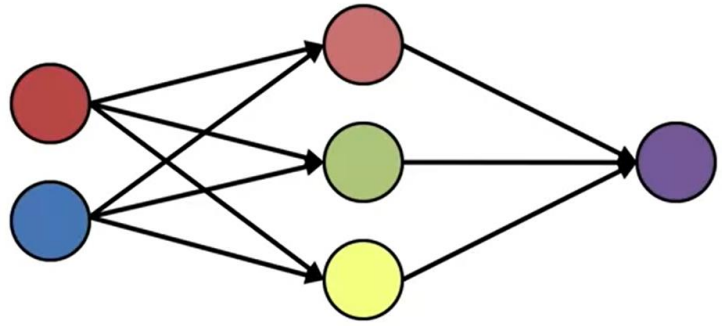
However, the tile code representation is able to better discriminate between states
and achieves a lower value error.

It's interesting that even though the tile coder has many more parameters, it can learn just as quickly as the core state aggregation.

But it also achieves better discrimination because of the small intersections created by multiple overlapping tiles.

What is a Neural Network ?

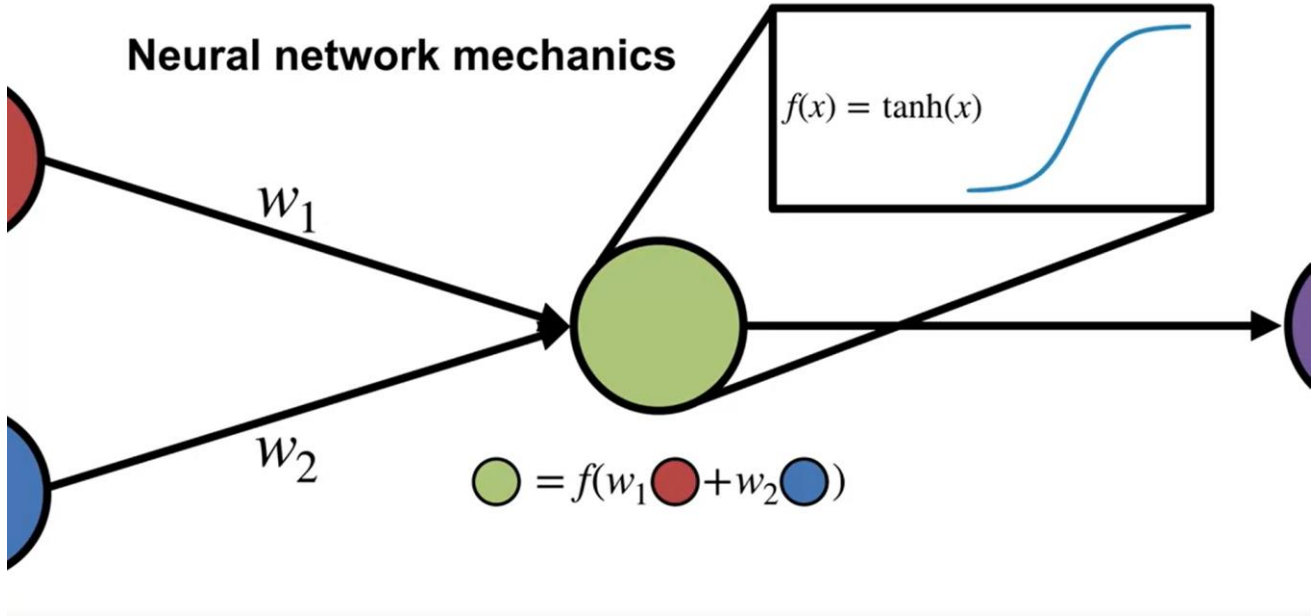
Simple neural network



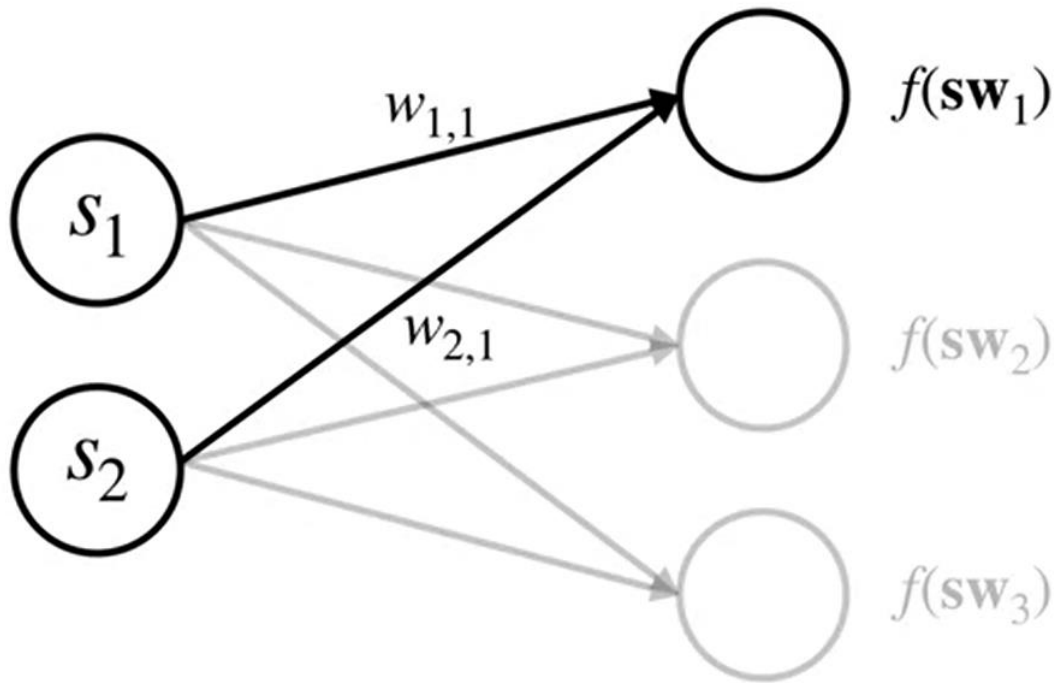
OR …

OR

$$f(x) = \begin{cases} x & x > 0 \\ 0 \end{cases}$$



Neural network implementation



$$\mathbf{s} = [s_1, s_2]$$

$$\mathbf{W} = [\mathbf{w}_1 \quad \mathbf{w}_2 \quad \mathbf{w}_3]$$

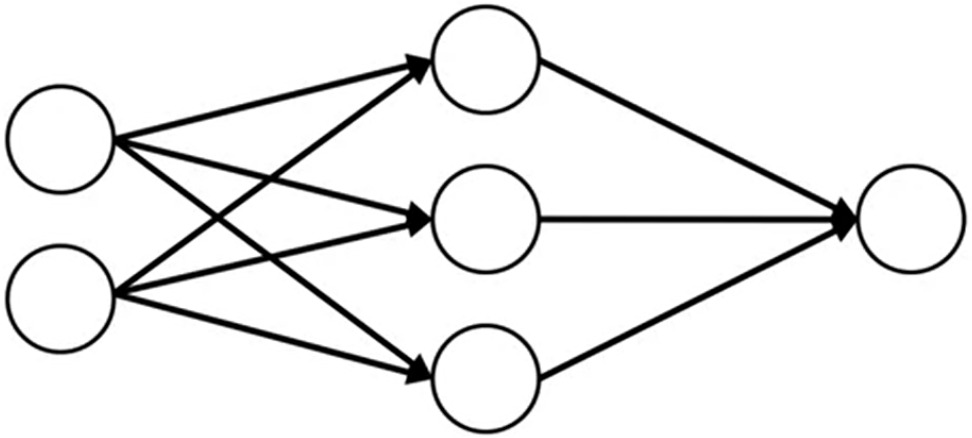
$$= \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \end{bmatrix}$$

$$\text{outputs} = f(\mathbf{s}\mathbf{W})$$

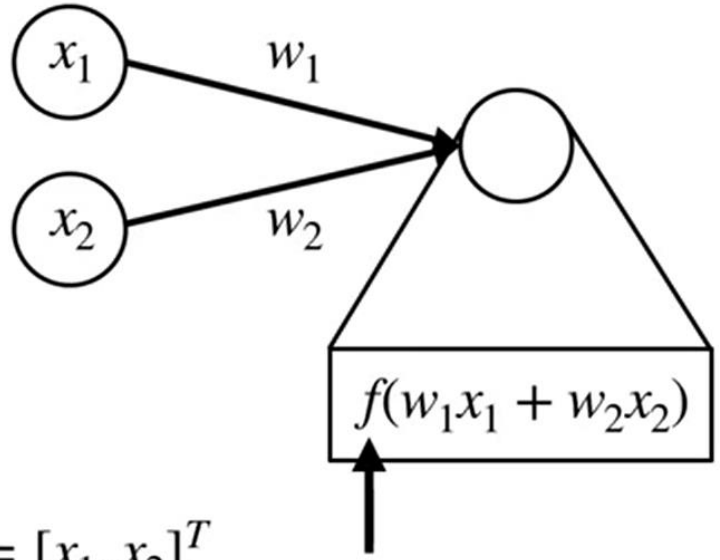
$\mathbf{s} = [s_1, s_2]$ $\quad$ $\mathbf{w}_1 = [w_{1,1}, w_{2,1}]^T$

Non-linear Approximation with Neural Networks?

Non-linear representations

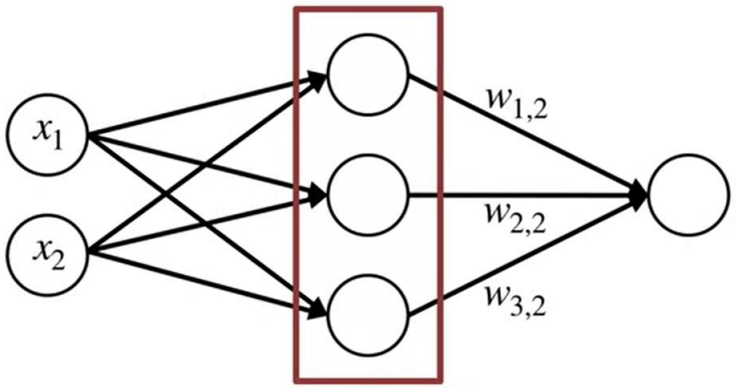


$$\mathbf{w}_{init} \sim \mathcal{N}(\mu, \sigma)$$



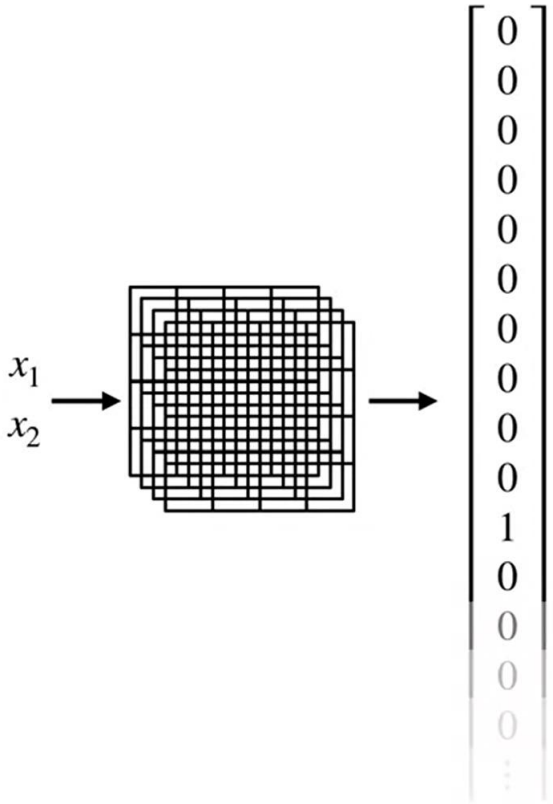
$$\mathbf{x} = [x_1, x_2]^T$$

Non-linear representations



Representation

$\mathbf{x} = [x_1, x_2]^T$



This process is actually not that different from tile coding, we pass inputs to a tile coder and get back a new representation.

So, in both cases, we construct a non-linear mapping of the inputs to produce the features.

In both cases, we take a linear combination of the representation to produce the output, the approximate value of the current state.

Summary

- Neural networks can be viewed as constructing features
- Neural networks are non-linear functions of state

When we created the tile coder we had to set several parameters, the size and shape of the tiles and the number of tilings.

These parameters were fixed before learning.

In a neural network, we have similar parameters corresponding to the number of layers, the number of nodes in each layer, and the activation functions.

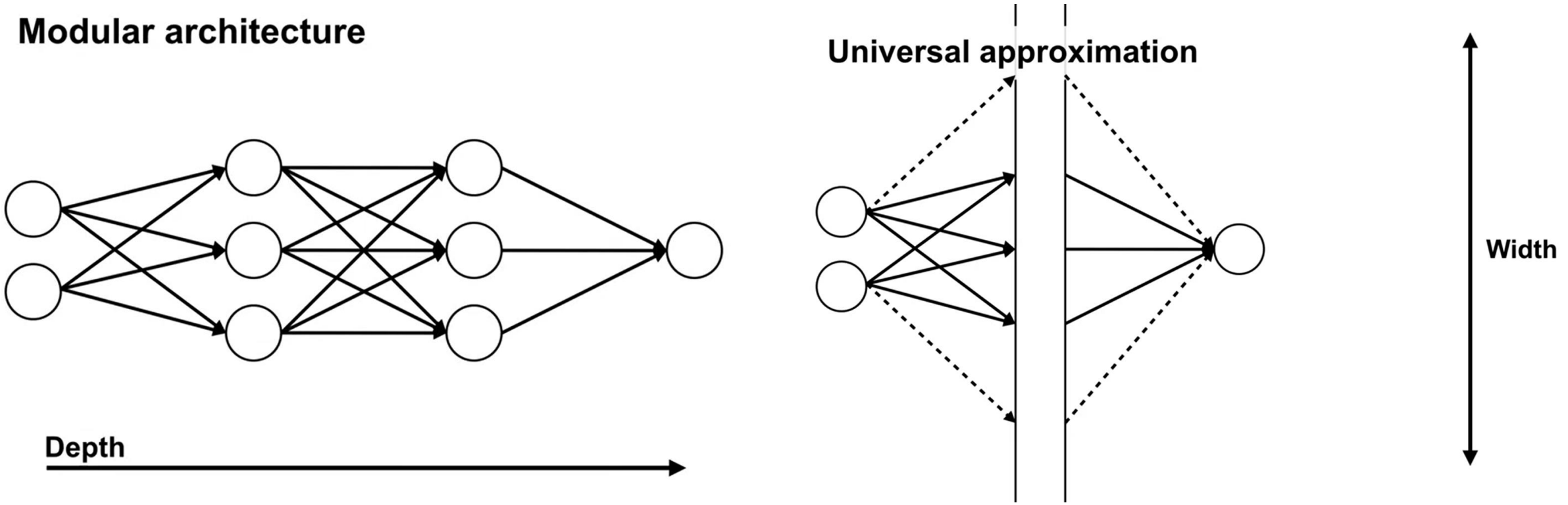
These are all, typically, fixed before learning. In this sense, both use prior knowledge to help in constructing features.

However, in addition, the neural network also has adjustable parameters that change the features during learning.

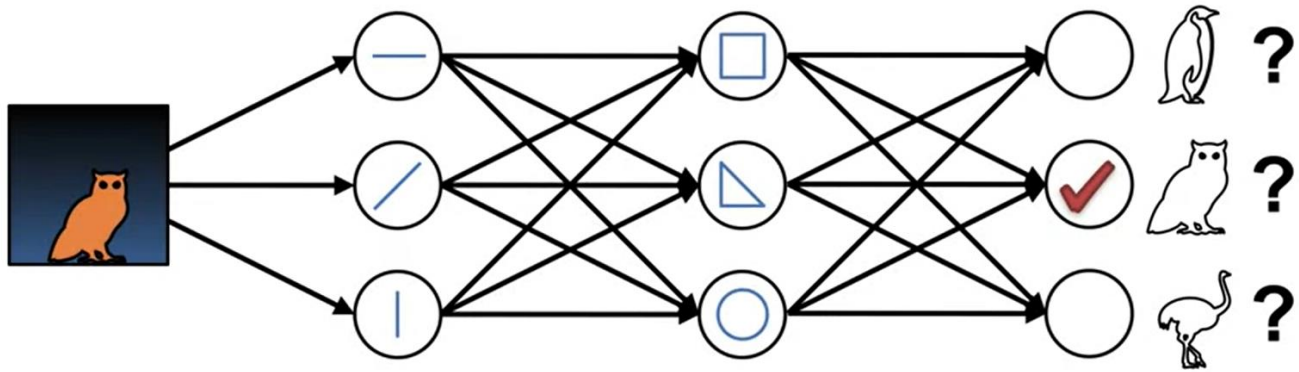
The neural network can use data to improve the features, whereas the tile coder cannot incorporate new information from data.

Both tile coding in neural networks produce features that are non-linear in the input space.

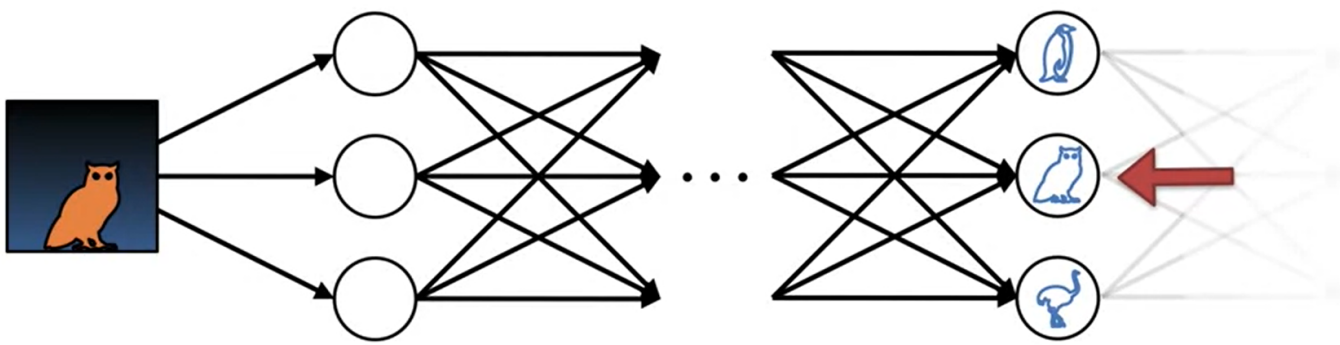
Deep Neural Network



Compositional features



Levels of abstraction

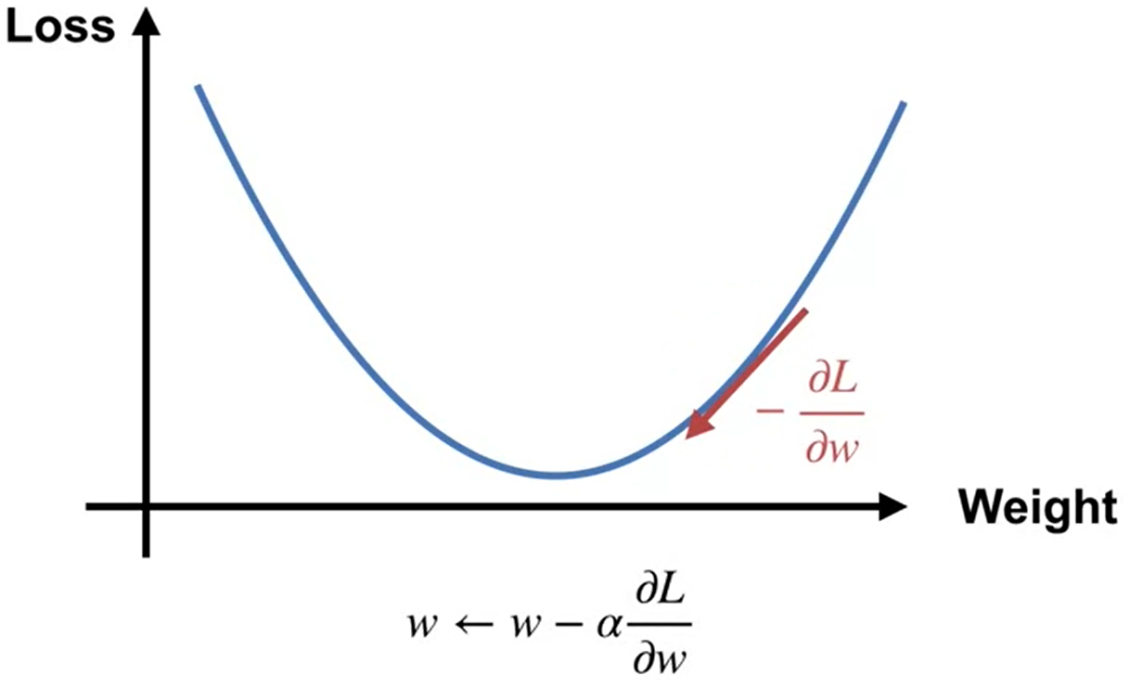


Summary

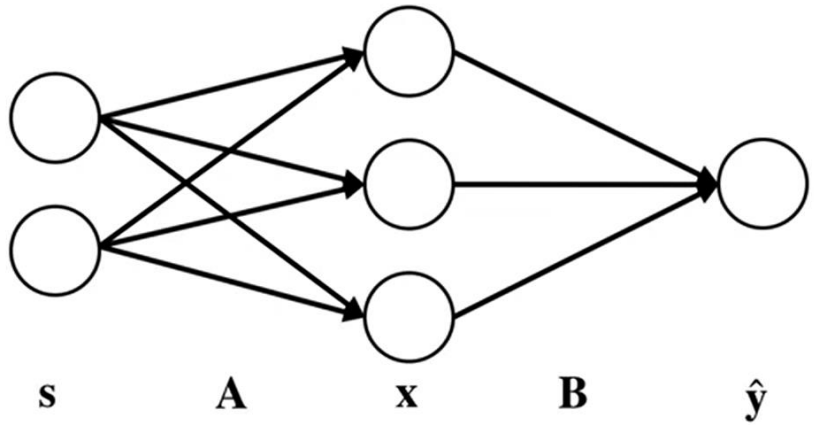
- Neural networks can be composed of multiple layers
- Depth facilitates composition and abstraction

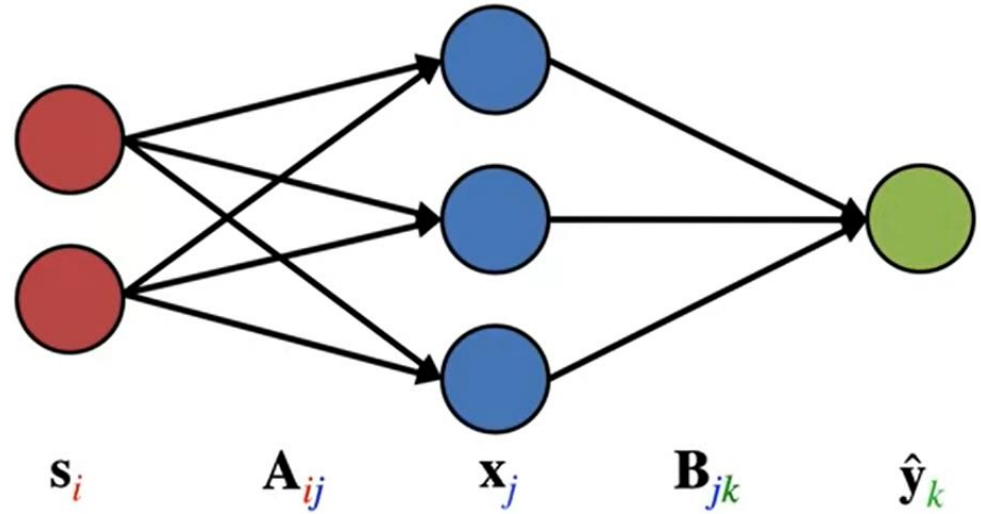
Gradient Descent for Training Neural Networks

Recap on Gradient Descent

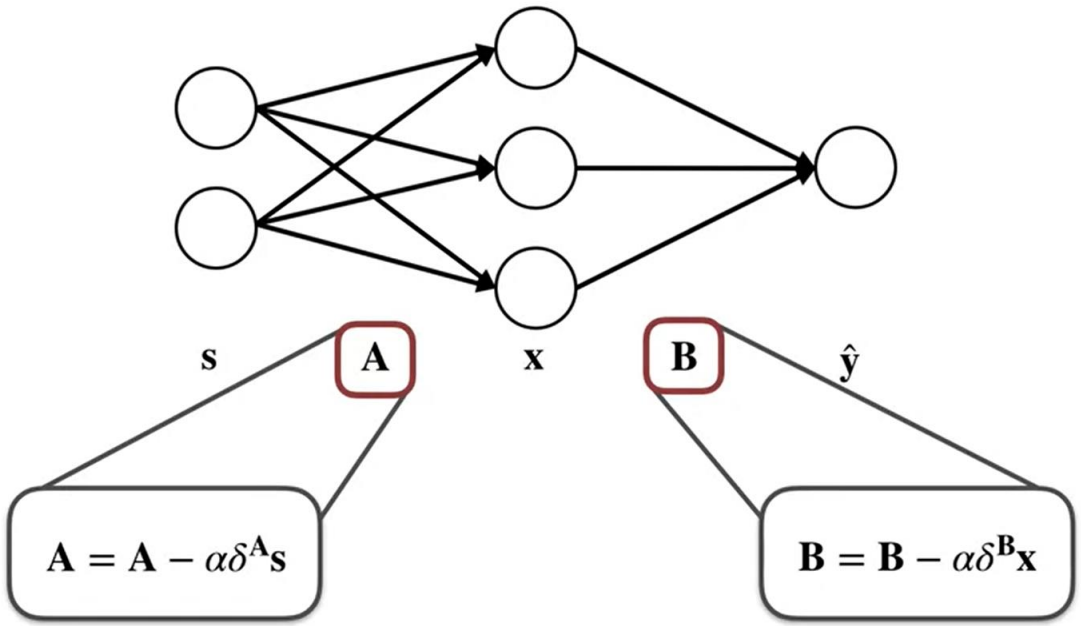


Notation





Goal



$$L(\hat{y}_k, y_k) = (\hat{y}_k - y_k)^2$$

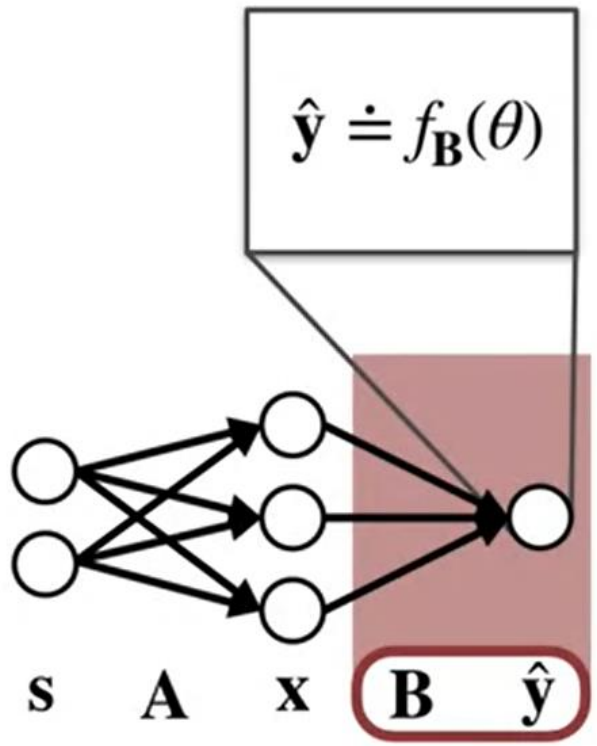
Deriving the gradient

$$\mathbf{x} \doteq f_{\mathbf{A}}(\mathbf{sA})$$
$$\theta \doteq \mathbf{xB}$$
$$\hat{\mathbf{y}} \doteq f_{\mathbf{B}}(\theta)$$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{B}_{jk}}$$

$$= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} \frac{\partial \theta_k}{\partial \mathbf{B}_{jk}}$$

$$= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} \mathbf{x}_j$$

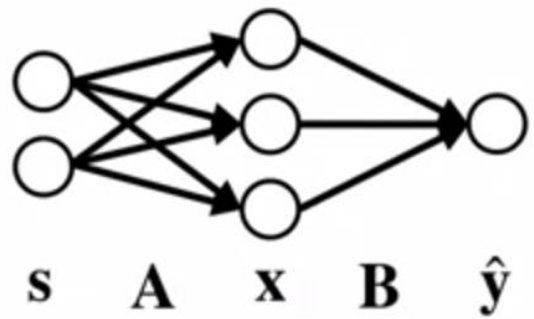


An example of the gradient

$$\mathbf{x} \doteq f_{\mathbf{A}}(\mathbf{sA})$$
$$\theta \doteq \mathbf{xB}$$
$$\hat{\mathbf{y}} \doteq f_{\mathbf{B}}(\theta)$$

Loss:	$L = \frac{1}{2}(\hat{\mathbf{y}}_k - \mathbf{y}_k)^2$	$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} = (\hat{\mathbf{y}}_k - \mathbf{y}_k)$
Activation:	$f_{\mathbf{B}}(\theta_k) = \theta_k$	$\frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} = \frac{\partial \theta_k}{\partial \theta_k} = 1$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} \mathbf{x}_j = (\hat{\mathbf{y}}_k - \mathbf{y}_k)\mathbf{x}_j$$



Deriving the gradient

$$\mathbf{x} \doteq f_{\mathbf{A}}(\mathbf{sA})$$
$$\theta \doteq \mathbf{xB}$$
$$\hat{\mathbf{y}} \doteq f_{\mathbf{B}}(\theta)$$

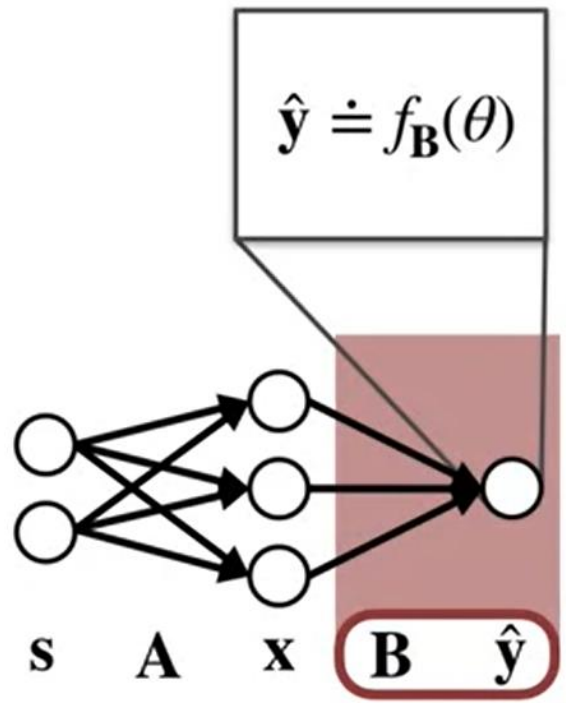
$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{B}_{jk}}$$

$$= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} \frac{\partial \theta_k}{\partial \mathbf{B}_{jk}}$$

$$= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k} \mathbf{x}_j \qquad \delta_k^{\mathbf{B}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k}$$

$$= \delta_k^{\mathbf{B}} \mathbf{x}_j$$

$$\boxed{\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \delta_k^{\mathbf{B}} \frac{\partial \theta_k}{\partial \mathbf{B}_{jk}}}$$



Deriving the gradient

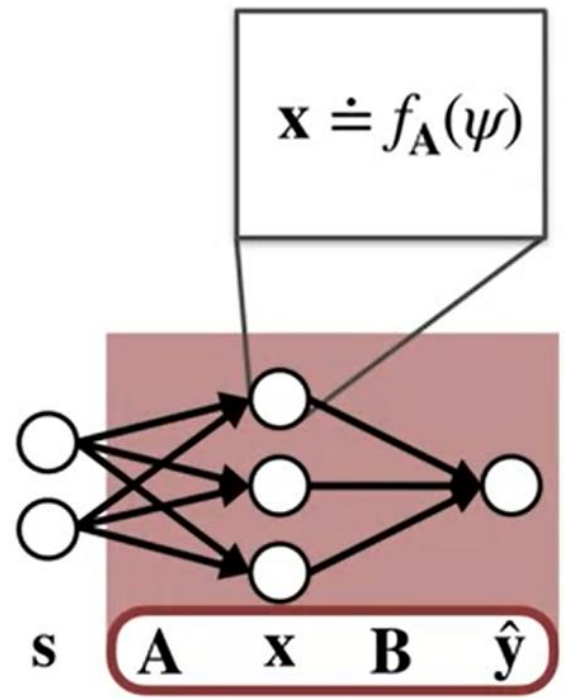
$$\psi \doteq \mathbf{s}\mathbf{A}$$
$$\mathbf{x} \doteq f_{\mathbf{A}}(\psi)$$
$$\theta \doteq \mathbf{x}\mathbf{B}$$
$$\hat{\mathbf{y}} \doteq f_{\mathbf{B}}(\theta)$$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{A}_{ij}} = \delta_k^{\mathbf{B}} \frac{\partial \theta_k}{\partial \mathbf{A}_{ij}}$$

$$= \delta_k^{\mathbf{B}} \mathbf{B}_{jk} \frac{\partial \mathbf{x}_j}{\partial \mathbf{A}_{ij}}$$

$$= \delta_k^{\mathbf{B}} \mathbf{B}_{jk} \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial \mathbf{A}_{ij}}$$

$$= \delta_k^{\mathbf{B}} \mathbf{B}_{jk} \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j} \mathbf{s}_i$$



Deriving the gradient

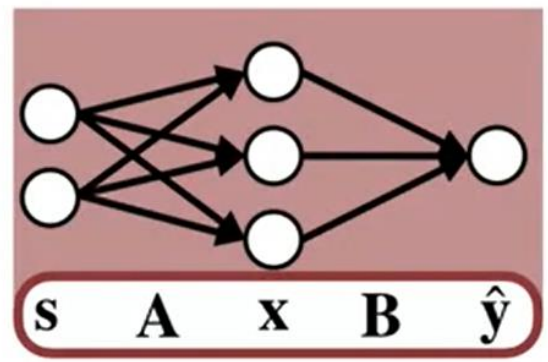
$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{A}_{ij}} = \delta_k^{\mathbf{B}} \mathbf{B}_{jk} \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j} \mathbf{s}_i \qquad \delta_j^{\mathbf{A}} = \left(\mathbf{B}_{jk} \delta_k^{\mathbf{B}}\right) \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j}$$

$$= \delta_j^{\mathbf{A}} \mathbf{s}_i$$

$$\psi \doteq \mathbf{s}\mathbf{A}$$
$$\mathbf{x} \doteq f_{\mathbf{A}}(\psi)$$
$$\theta \doteq \mathbf{x}\mathbf{B}$$
$$\hat{\mathbf{y}} \doteq f_{\mathbf{B}}(\theta)$$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{A}_{ij}} = \delta_j^{\mathbf{A}} \mathbf{s}_i$$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \delta_k^{\mathbf{B}} \mathbf{x}_j$$



The backprop algorithm

for each (s, y) **in** D:

$$\delta_k^{\mathbf{B}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k}$$

$$\nabla_{\mathbf{B}}^{jk} = \delta_k^{\mathbf{B}} \mathbf{x}_j$$

$$\mathbf{B} = \mathbf{B} - \alpha_{\mathbf{B}} \nabla_{\mathbf{B}}$$

Both A and B are updated together in the last step of loop.

$$\delta_j^{\mathbf{A}} = \left(\mathbf{B}_{jk} \delta_k^{\mathbf{B}}\right) \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j}$$

$$\nabla_{\mathbf{A}}^{ij} = \delta_j^{\mathbf{A}} \mathbf{s}_i$$

$$\mathbf{A} = \mathbf{A} - \alpha_{\mathbf{A}} \nabla_{\mathbf{A}}$$

Basically, old value of B is used in the update rule for A

The backprop algorithm

for each (s, y) in D:

$$\delta_k^{\mathbf{B}} = (\hat{\mathbf{y}}_k - \mathbf{y}_k)\mathbf{x}_j \qquad \delta_k^{\mathbf{B}} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_{\mathbf{B}}(\theta_k)}{\partial \theta_k}$$

$$\nabla_{\mathbf{B}}^{jk} = \delta_k^{\mathbf{B}} \mathbf{x}_j$$

$$\mathbf{B} = \mathbf{B} - \alpha_{\mathbf{B}} \nabla_{\mathbf{B}}$$

$$u = \begin{cases} \psi & \psi > 0 \\ 0 & otherwise \end{cases} \qquad u = \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j}$$

$$\delta_j^{\mathbf{A}} = \left(\mathbf{B}_{jk} \delta_k^{\mathbf{B}}\right) u \qquad \delta_j^{\mathbf{A}} = \left(\mathbf{B}_{jk} \delta_k^{\mathbf{B}}\right) \frac{\partial f_{\mathbf{A}}(\psi_j)}{\partial \psi_j}$$

$$\nabla_{\mathbf{A}}^{ij} = \delta_j^{\mathbf{A}} \mathbf{s}_i$$

$$\mathbf{A} = \mathbf{A} - \alpha_{\mathbf{A}} \nabla_{\mathbf{A}}$$

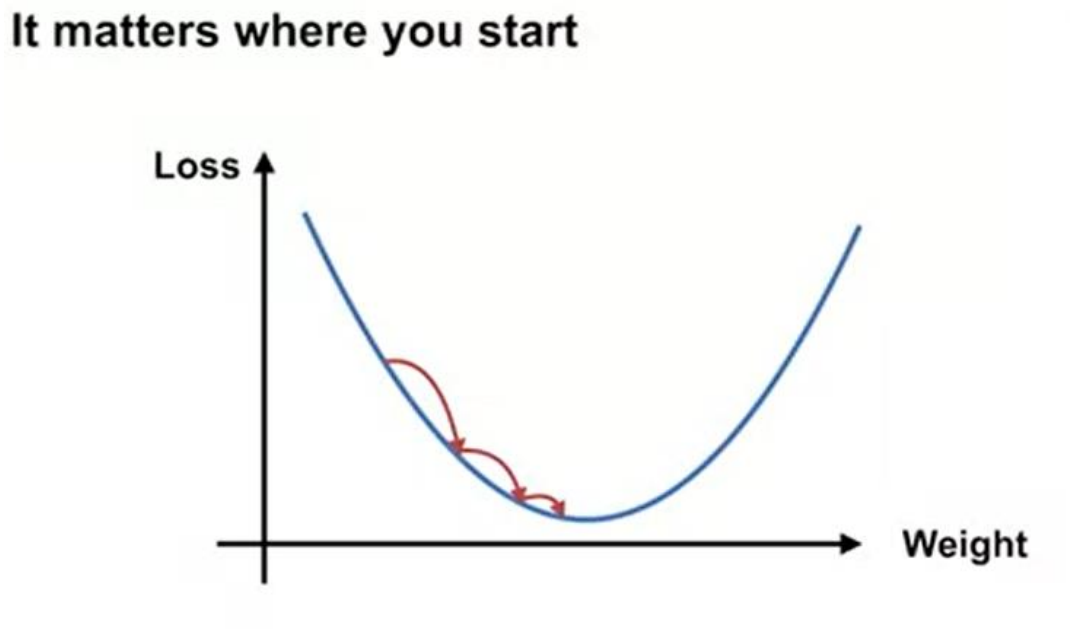
When using ReLU as activation function

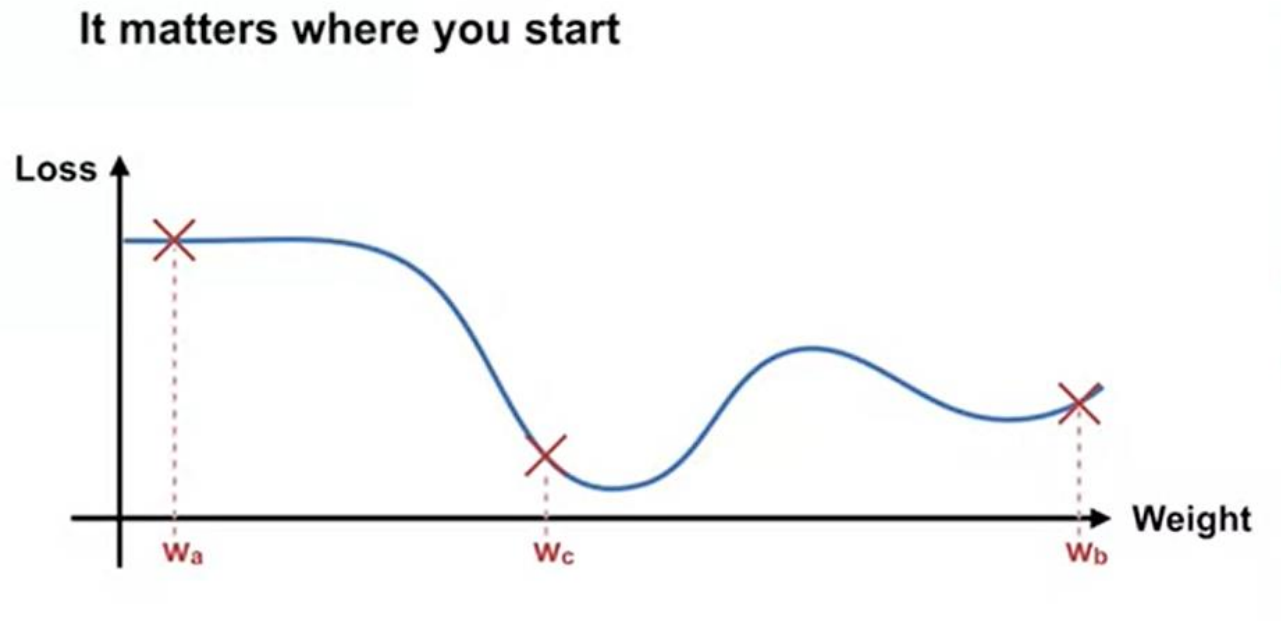
Summary

- The gradient can be used to update the parameters of a neural network with stochastic gradient descent
- Backprop can save computation by computing gradients starting at the output of the network.

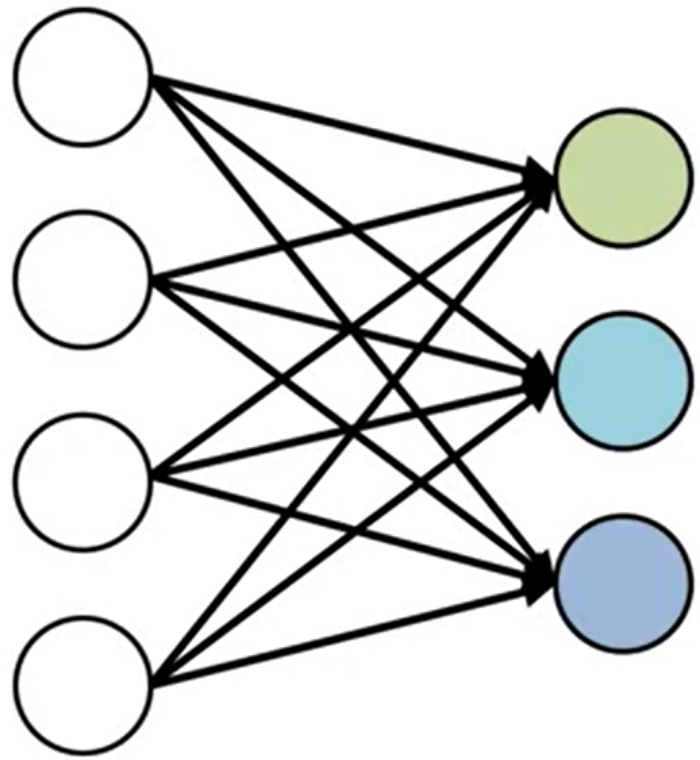
Optimization Strategies for Neural Networks

Like many machine learning methods, neural networks are trained using an iterative process.
This procedure, must have some starting point.
The choice of this starting point, can play a big role in the performance of the neural network.





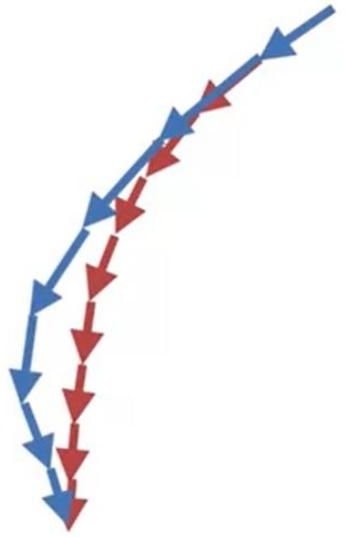
Weight initialization



$$\mathbf{w}_{init} \sim \frac{\mathcal{N}(0,\ 1)}{\sqrt{n_{inputs}}}$$

Update momentum

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t) + \lambda \mathbf{M}_t$$
$$M_{t+1} \leftarrow \lambda M_t - \alpha \nabla_{\mathbf{w}} L$$

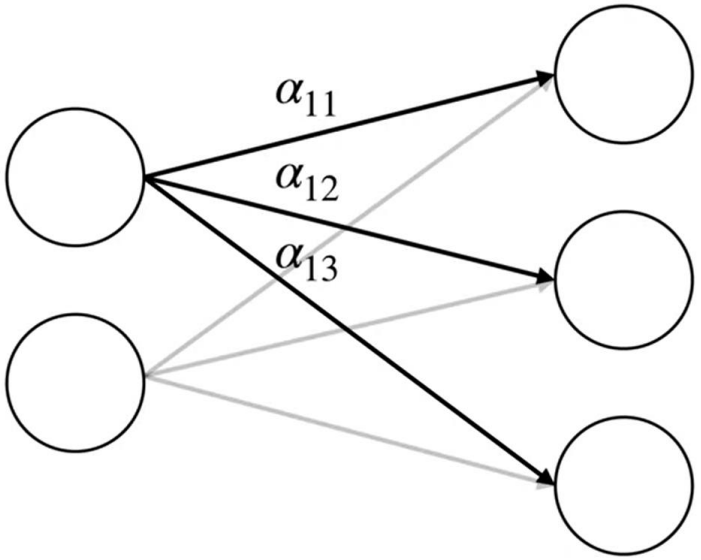


Regular stochastic gradient descent update plus an extra term called the momentum M.
The momentum term summarizes the history of the gradients using a decaying sum of gradients with decay rate Lambda.

If recent gradients have all been in similar directions, then we gained momentum in that direction. This means, we make a large step in that direction.

If recent updates have conflicting directions, then it kills the momentum. The momentum term will have little impact on the update and we will make a regular gradient descent step.

Vector step sizes



- Another potential improvement is to use a separate step size for each weight in the network.
- So far, we have only talked about a global scalar step size.
- This is well-known to be problematic because this can result in updates that are too big for some weights and too small for other weights.
- Adapting the step sizes for each weight, based on statistics about the learning process in practice results in much better performance.
- Instead of updating with a scalar Alpha, there's a vector of step sizes indexed by t to indicate that it can change on each time-step.
- Each dimension of the gradient, is scaled by its corresponding step size instead of the global step size.
- There are a variety of methods to adapt a vector of step sizes

Summary

- Better to initialize weights such that the node outputs have similar variance
- Momentum in weight updates can speed up neural networks optimization