Prediction and Control with Function Approximation

Miguel A. Saavedra-Ruiz

February 2020

1 On-policy Prediction with Approximation

Moving to Parameterized Functions

we've learned about tabular methods. Methods that store table containing separately learn values, for each possible state. In real-world problems, these tables will become intractably large. Imagine a robot that sees the world through a camera. We clearly **cannot store a table entry for every possible image**.

Previously we have spent a lot of time thinking about how to estimate value functions. So far, our value approximations have always had the same form. **For each state, we store separate value in a table**. We look at values, and modify them in the table as learning progresses Fig. 1. But this is not the only way to approximate a value function. In principle, we can use any function that takes a state, and produces a real number.

State	Value
s_1	-4
s_2	6
s_3	12
s_4	5
s_5	53
...	
s_{16}	-9

Figure 1: Storing value functions in a table

For example in a grid like this, our value function approximation could take the X and Y position, and add them together to produce a value estimate Fig. 2.

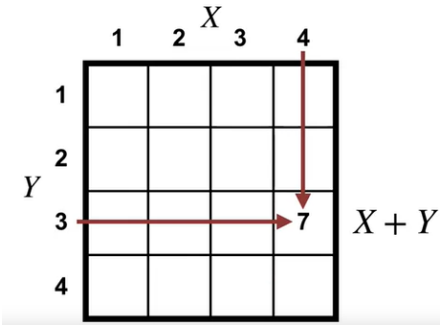

Figure 2: Value functions with a grid

Here is where parameterized function comes in. We incorporate a set of real valued weights, which we can adjust to change the function $\hat{v}(s, w)$.

For example in the grid Fig. 2, instead of using a fixed sum of X and X, we could use a function of the form $\hat{v}(s, w) \doteq W_1 X + W_2 Y$. The weights W_1 and W_2 parameterize our function. They allow us to change the output the function generates. To indicate that this function approximates the true value function, we use the notation $\hat{v}$. W is a vector containing all the weights that parameterize the approximation. We do not have to store a whole table of values. We only have to store two weights to represent our value function approximation.

$$\hat{v}(s, w) \doteq W_1 X + W_2 Y$$

If we change either W_1 or W_2, we will change the value estimate for every state. Now, our learning outcomes will **modify the weights, instead of the individual state values**.

The example discussed before is a special case called **linear value function approximation**. In this case, the value of each state is represented by a linear function of the weights. This simply means that the value of each state, is computed as the sum of the weights multiplied by some fixed attributes of the state called features $X_i(s)$. We can express this compactly. We write that the approximate value is given by the inner product of this feature vector, and the weight vector. We will use $\boldsymbol{X(S)}$ to denote the feature vector.

$$\begin{aligned} \hat{v}(s, w) &\doteq \sum W_i X_i(s) \\ &= < \boldsymbol{W, X(s)} > \end{aligned}$$

Limitation of linear value function approximation

Our choice of feature impacts the kinds of value functions we can represent. The X and Y features cannot represent the true value functions shown in the grid Fig. 3. To represent this values, W_1 and W_2 must be zero. Nonetheless, it is always possible to use more features and accomplish a more reliable value function estimate.

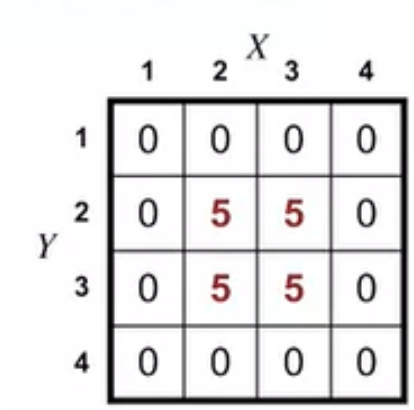

Figure 3: non-linear grid

Tabular value functions can be represented as linear value functions. With the appropriate set of weights as shown in Fig. 4 a tabular value function as the one in Fig. 1 can be represented.

These parameterized values are general, and we can consider lots of different types of functions. Neural networks are an example of a non-linear function of state. The output of the network is our approximate value for a given state. The state is passed to the network as the input. All the connections in the network

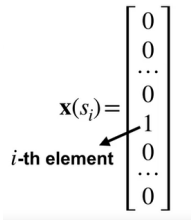

Figure 4: Set of features to create tabular value function

correspond to real valued weights. When data is passed through a connection, the weight is multiplied by its input. This process transforms the input state through a sequence of layers to finally produce the value estimate Fig. 5.

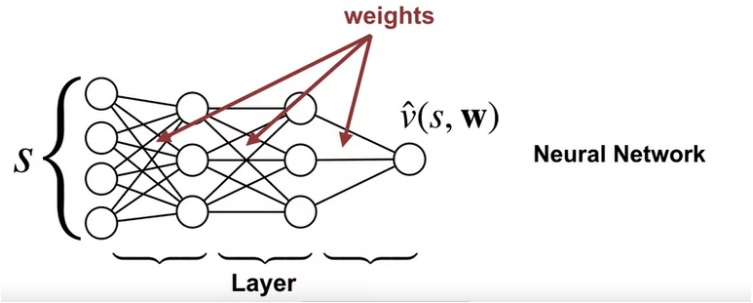

Figure 5: Non linear function approximation

Generalization and Discrimination

Generalization is something that humans do naturally. Once a person learns how to drive one car, they don't have to start from scratch to learn how to drive a different car. They also don't have to start from scratch on a different street or when it is raining. We'd like our agents to be able to generalize too.

Generalization intuitively means applying knowledge about specific situations to draw conclusions about a wider variety of situations. When we talk about generalization in the context of policy evaluation, we mean that **updates to the value estimate of one state influence the value of other states** Fig. 6

State	Value
s_1	-3
s_2	-2
s_3	2
s_4	-1.7
s_5	4

Figure 6: Generalizing a tabular value function

Imagine a robot tasked with collecting cans, observing the world through a set of distance sensors. In many locations, it would take the same amount of time to drive to the nearest can. Even though they correspond to different sensor readings, these locations have similar values. Thus, we might want the value function to generalize across those states. Generalization can speed learning by making better use of the experience we

have. The robot may not have to visit every state as much to get this values correct if it can learn its value from similar states.

Discrimination means the ability to make the values for two states different to distinguish between the values for these two states.

Imagine a robot collecting cans, it is in a state where a can is three feet away, but behind a wall. Contrast this to a state where a can is three feet away, but with a clear paths to reach it. The robot would want to assign different values to these states. So while it is useful to generalize between states with similar distance to the nearest can, it is also important that we discriminate between states based on other information when it is likely to impact their value Fig. 7.

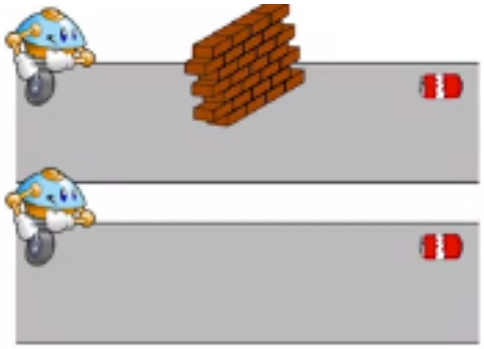

Figure 7: Discrimination with robot example

Let's look at the game of chess as a concrete example. Take the extreme case where we treat all states as the same. This value corresponds to the probability of winning regardless of the state of the game. With equally matched players, this number might be 50 percent. On the opposite extreme, we have the tabular case, where we treat every state as totally different. This is fine for small problems, but in a game like chess, it is impractical to even enumerate all the possible states. There are approximately 10^{46} states. Further, imagine how long it would take to individually learn the value of all these states. We want something in-between where we generalize between states with similar probabilities of winning. Identifying these similarities to get such groupings is a difficult question with no single answer Fig. 8.

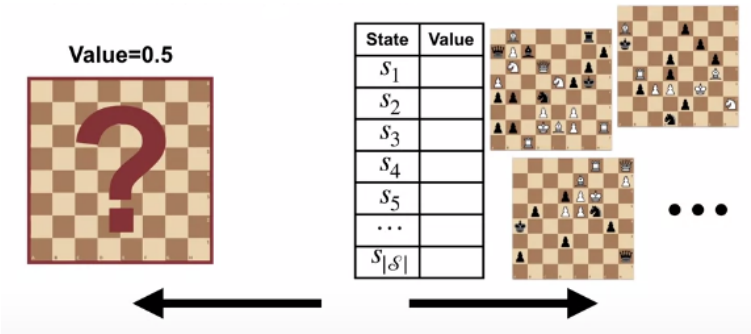

Figure 8: How should we generalize?

- Tabular representations provide good discrimination but no generalization
- Generalization is important for faster learning
- Having both generalization and discrimination is ideal

Framing Value Estimation as Supervised Learning

Supervised learning methods can be useful for handling parts of the reinforcement learning problem. It involves **approximating a function given a dataset of input target pairs**. For example, imagine we had a list of house prices along with a set of attributes for each house. We could use supervised learning to train a function to take the attributes of a house as input and estimate the expected price of the house. The hope is that the function learned will also generalize to approximate the expected price for houses that were not in the training set.

The problem of policy evaluation in reinforcement learning can be framed similarly. This similarity is most obvious in the case of Monte Carlo methods. Remember, the **Monte Carlo** methods estimate the value function using samples of the return. We can think of this as an example of a supervised learning problem, where the **input is the state and the targets are the returns** Fig. 9.

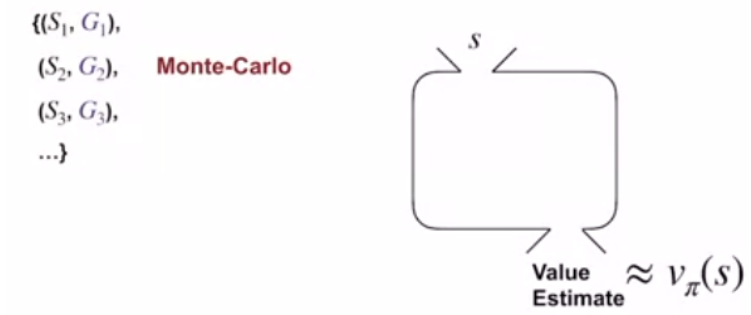

Figure 9: Framing policy evaluation as supervised Learning

TD can also be framed as supervised learning. In this case, the targets are the one-step bootstrap return. In principle, any function approximation technique from supervised learning can be applied to the policy evaluation task. However, not all are equally well-suited. TD with supervised learning is shown in the next equation.

$$(S_1, R_2 + \gamma\hat{v}(S_2, w)), (S_2, R_3 + \gamma\hat{v}(S_3, w)), (S_3, R_4 + \gamma\hat{v}(S_4, w))$$

In reinforcement learning, an agent interacts with an environment and continually generates new data. This is often called the online setting. To distinguish it from the offline setting where the full dataset is available from the start and remains fixed throughout learning. If we want to use a function approximation technique, we should make sure it can work in the online setting. Some methods are not compatible with the online setting because they are either designed for a fixed batch of data or there are not designed for temporally correlated data, and the data in reinforcement learning is always correlated.

$$(S_1, G_1), (S_2, G_2), (S_3, G_3), (S_4, G_4)$$

TD methods introduce an additional complication when applying techniques from supervised learning. TD methods use **bootstrapping**, meaning that our targets now depend on our own estimates. **These estimates change as learning progresses and so our targets continually change by w** $(S_1, R_2 + \gamma\hat{v}(S_2, w))$. This is different than supervised learning where we have access to a ground truth label as the target.

Supervised learning methods are typically not designed for changing targets nor targets computed from the agent's own estimates. Methods compatible with online learning and Bootstrapping are needed.

The Value Error Objective

Start by imagining an idealized scenario. We get a sequence of pairs of states and true values. We want to use this data to find **a parameterized function that closely approximates** v_π. We will do this by adjusting the weights so that the output of the function closely matches the associated value for a given state. In general, we can't expect to perfectly match the value function on all states. The function approximately we choose will limit the value functions we can represent. To make our goal precise, we need to specify some measure of how close our approximation is to the value function Fig. 10.

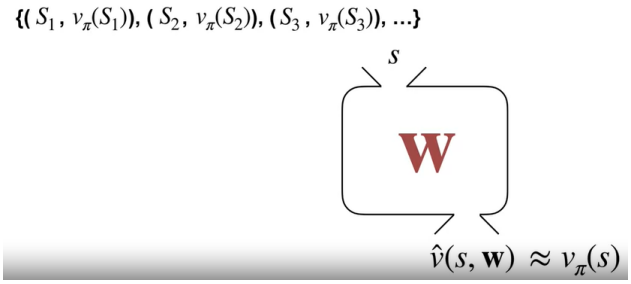

Figure 10: An ideal scenario of parameterized function approximation

Consider a linear value function approximation denoted as $\hat{v}$. Here to keep the visualization simple, the state is a single-dimensional and continuous. Clearly, our approximation of V_* is not perfect, but how far off is it? Let's define this more precisely. Let's start by defining a measure of the error between the value of a state and the approximate value. A natural choice is the squared error $[V_\pi(s) - \hat{v}(s, w)]^2$, the squared difference between the value and our approximation. However, this is not enough to define an objective for function approximation. Making the estimate more accurate in one state will often mean making it less accurate in another state Fig. 11. For this reason, we need to specify how much we care about getting the value right for each state. We will call that $\mu(s)$. We can now write our full objective as a sum of the squared error over the entire state space where each state is weighted by $\mu(s)$. We call this objective the Mean Squared Value Error.

$$\sum_s \mu(s)[V_\pi(s) - \hat{v}(s, w)]^2$$

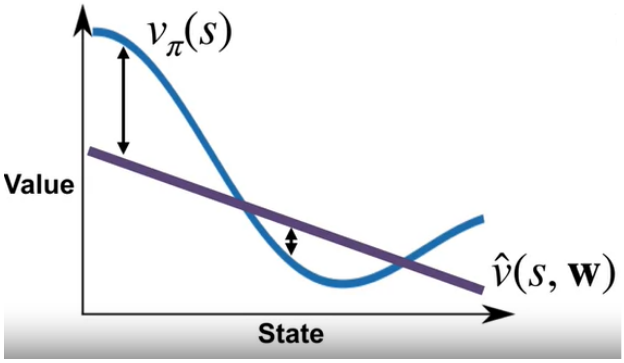

Figure 11: The mean squared value error objective

$\mu(s)$ should tell us how much we care about each state. A natural measure is the fraction of time each state is visited under the policy. That means we want to minimize the average value error for the states we visit while following π. The states that the policy spends more time in have a higher weight in the objective. We care less about errors in the states the policy visits less frequently. $\mu(s)$ **is a probability distribution** .

To adapt the weights and minimize the mean square value error objective denoted as $\overline{VE}$ we follow the next equation:

$$\overline{VE} = \sum_s \mu(s)[V_\pi(s) - \hat{v}(s, w)]^2$$

Introducing Gradient Descent

We want to minimize the mean squared value error, to make our value estimates close to the true value function.

$$\sum_s \mu(s)[V_\pi(s) - \hat{v}(s, w)]^2$$

Recall that our value estimate is given by a function of the state, parameterised by a set of real-valued weights, W. These weights determine how the value's computed for each state, changing the weights will modify the value estimate for many states.

$$W = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_d \end{bmatrix}$$

Here we've plotted a function f, with scalar parameter W. The derivative tells us how to locally change W to increase or decrease f. The sign of the derivative of f at a particular point W, indicates the direction to change W to increase f. The magnitude of the derivative indicates the slope of the function f at the point W. That is, how quickly f will change as we vary W Fig. 12.

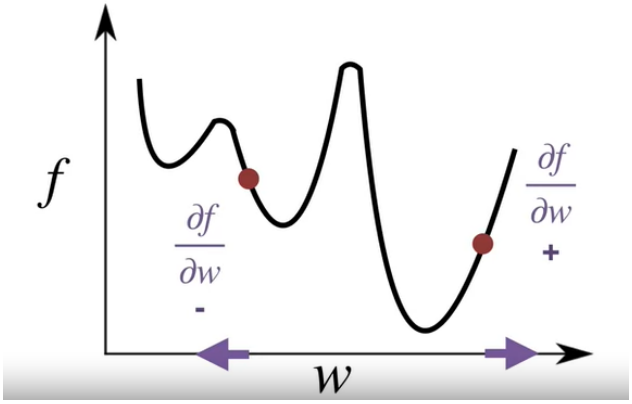

Figure 12: Derivatives

If f is privatized by more than one variable, then W is a vector. In this case, we need to introduce the idea of a gradient to describe how f changes as the vector W changes. The gradient is a vector of partial derivatives, indicating how a local change in each component of W affects the function. Notice that the gradient has to be the same size as the weight vector. The sign of each component of the gradient specifies the direction to change the associated component of W, in order to increase f. The direction is either positive or negative. The magnitude of the component specifies how quickly f changes, as W moves in that direction.

$$W \doteq \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_d \end{bmatrix}$$

$$W = \begin{bmatrix} \frac{\partial f}{\partial W_1} \\ \frac{\partial f}{\partial W_2} \\ \vdots \\ \frac{\partial f}{\partial W_d} \end{bmatrix}$$

Let's derive the gradient of a linear value function. Remember, a linear value approximation is just an inner product of the weights with the feature vector for the state. The partial derivative of that with respect to a single weight, is just the feature associated with that weight. Remember, the features themselves do not depend on the weights. This means the gradient of a linear value approximation is simply the feature vector for that state.

$$\hat{v}(s, w) \doteq \sum W_i X_i(s)$$
$$\frac{\partial \hat{v}(s, w)}{\partial W_i} = x_i(s)$$
$$\nabla \hat{v}(s, w) = x(s)$$

If we want to decrease our objective function $J(w)$, we should move the weights in the direction of the negative of the gradient. This is the idea of gradient descent. Here's the gradient descent update rule that captures this intuition.

$$W_{t+1} \doteq W_t - \alpha \nabla J(W_t)$$

We make small changes in the direction that will most reduce the objective. We use a step size parameter, α, to control how far we move, as otherwise there is a risk of stepping too far. This is because moving in this direction is only guaranteed to decrease the objective locally.

The idea of Gradient Descent is to find a Local minimum, nevertheless, there is always possible to find a unstable solution as a local maximum or a saddle point Fig. 13.

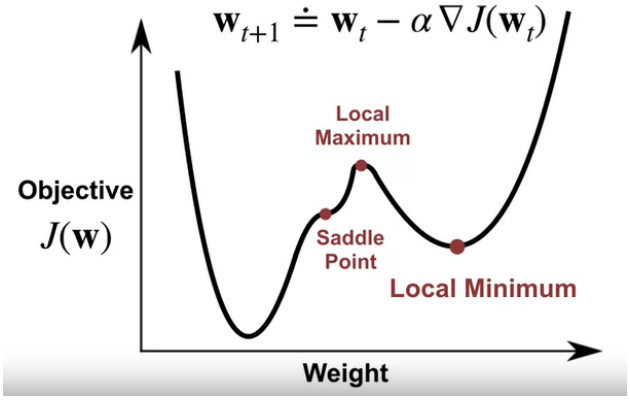

Figure 13: Gradient Descent

Note that a global minimum does not necessarily correspond to the true value function. It is limited by our choice of function parameterization, and depends on our choice of objective.

$$\hat{v} \neq v_*$$

- Grandient descent can be used to find stationary points of objectives
- These solutions are not always globally optimal

Gradient Monte for Policy Evaluation

Let's start by computing the gradient of the mean squared value error, with respect to the weights of our approximation. Remember, this objective is a weighted sum of the squared error over all states. Following the rules of calculus, we can pull the gradient inside the sum. We then take the gradient of each term inside the sum, which requires the chain rule. The gradient of $[V_\pi(s) - \hat{v}(s, w)]$, equals the gradient of $\hat{v}$ because v_π is not a function of w. In other words, changing w does not change $v_\pi(s)$. The gradient of the value function approximation will depend on the particular parameterized function we are using.

$$\nabla \sum_{s \in S} \mu(s)[V_\pi(s) - \hat{v}(s, w)]^2$$

$$\sum_{s \in S} \mu(s) \nabla [V_\pi(s) - \hat{v}(s, w)]^2$$

$$-\sum_{s \in S} \mu(s) 2[V_\pi(s) - \hat{v}(s, w)] \nabla \hat{v}(s, w)$$

$$\triangle_w \propto \sum_{s \in S} \mu(s) 2[V_\pi(s) - \hat{v}(s, w)] \nabla \hat{v}(s, w)$$

Recall that

$$\hat{v}(s, w) = < \boldsymbol{W}, \boldsymbol{X}(\boldsymbol{s}) >$$

$$\nabla \hat{v}(s, w) = x(s)$$

If the difference $[V_\pi(s) - \hat{v}(s, w)]$ is positive, it means the true value is higher than our estimate, so we should change the weights in the direction that increases our estimate. If the current error is negative, we should change the weights in the opposite direction.

Computing the gradient for the mean squared value error requires summing over all states. This is generally not feasible. Also, we likely do not know the distribution μ. Instead, let's approximate this gradient. Imagine an idealized setting where we have access to v_π. Though we do not explicitly have μ, we can sample states from it simply by following the policy. Let's take one of these states, S_1, that occurs while following the policy with target $v_\pi(S_1)$. We can use this pair to make an update to decrease the error on that example.

$$\sum_{s \in S} \mu(s) 2[V_\pi(s) - \hat{v}(s, w)] \nabla \hat{v}(s, w)$$

$$(S_1, V_\pi(s_1)), (S_2, V_\pi(s_2)), (S_3, V_\pi(s_3)), (V_\pi(s_4)) \dots$$

$$w_2 \doteq w_1 + \alpha[V_\pi(s_1) - \hat{v}(s_1, w_1)]\nabla\hat{v}(s_1, w_1)$$

This updating approach is called **stochastic gradient descent**, because it only uses a stochastic estimate of the gradient. In fact, the expectation of each stochastic gradient equals the gradient of the objective. You can think of this stochastic gradient as a noisy approximation to the gradient that is much cheaper to compute, but can nonetheless make steady progress to a minimum.

Stochastic gradient descent allowed us to efficiently update the weights on every step by sampling the gradient. However, there is one remaining practical issue. We do not have access to v_π. Let's see how we can get rid of it from our update rule. Let's replace v_π with an estimate. One option is to use samples of the return G for each visited state, S_t, as we did for Monte Carlo methods. This makes sense because the value function is the expected value of these samples. In fact, the expectation of the gradient when we use a sampled return in place of the true value, is still equal to the gradient of the mean squared value error.

$$w_{t+1} \doteq w_t + \alpha[V_\pi(s_t) - \hat{v}(s_t, w_t)]\nabla\hat{v}(s_t, w_t)$$

$$w_{t+1} \doteq w_t + \alpha[G_t - \hat{v}(s_t, w_t)]\nabla\hat{v}(s_t, w_t)$$

Recall that:

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t | S_t = s\right]$$

$$\mathbb{E}_\pi\left[2[v_\pi(s_t) - \hat{v}(s_t, w_t)]\nabla\hat{v}(s_t, w_t)\right]$$

$$\mathbb{E}_\pi\left[2[G(s_t) - \hat{v}(s_t, w_t)]\nabla\hat{v}(s_t, w_t)\right]$$

This brings us to the Gradient Monte Carlo algorithm for estimating v_π. We take any policy π we wish to evaluate, we choose some function which maps states and weights to a real number that is differentiable in the weights. For a given weight vector, this $\hat{v}$ is a function of state producing the approximate values. We choose a value of the step size α, and initialize the weights parameterizing our estimate however we like. For example, to zero. On each iteration, the agent interacts with the environment to generate a full episode. We compute the sampled return G for each visited state. Then we loop through every step in the episode and perform a stochastic gradients and update based on the sampled returns Fig. 14

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[G_t - \hat{v}(S_t, \mathbf{w})\right]\nabla\hat{v}(S_t, \mathbf{w})$

Figure 14: Gradient Monte Carlo Algorithm

State agregation with Monte Carlo

State aggregation treats certain states as the same. In this table of eight states, we might choose to aggregate states together in groups of four. So now instead of a table of eight entries for the value function, we just have two. When we update the value of any state in the first group, the values of all the other states in that group is updated Fig. 15. State aggregation is another example of linear function approximation. There is one feature for each group of states. Each feature will be 1 if the current state belongs to the associated group, and 0 otherwise. The approximate value of a state is the weight associated with the group that state belongs to.

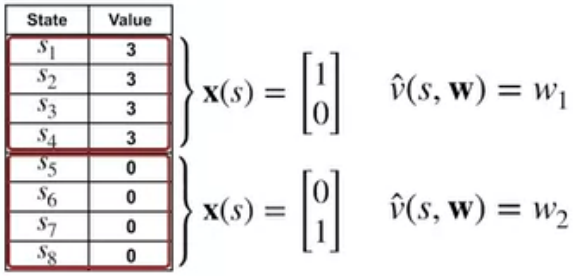

Figure 15: State aggregation example

Let's look at the gradient **Monte Carlo algorithm with state aggregation**. We already know how to compute the value for a given state. Now we need to think about how we compute the gradient of the value function so that we can use our key update formula. State aggregation is an example of linear function approximation, so the gradient is equal to the feature vector. The update rule modifies only the weight corresponding to the current active group. Let's think about how this update rule changes the weight.

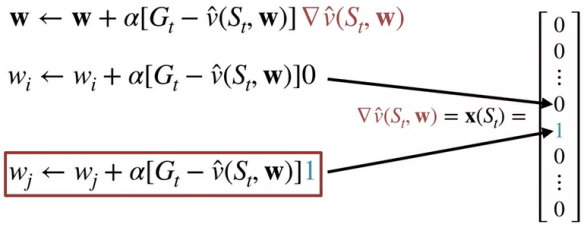

Figure 16: Computing the gradient for Monte Carlo with state aggregation

Semi-gradient TD for policy evaluation

Recall the gradient Monte Carlo update equation. It updates our current value estimate to be closer to a sample of the return G_t, but we can consider using other targets instead of the return. In fact, we can replace the return in this update with any estimate of the value.

$$w \leftarrow w + \alpha[G_t - \hat{v}(s, w)]\nabla \hat{v}(s, w)$$

We can replace the return in the last update with any estimate of the value. Let's call this estimate U_t. If U_t is an unbiased estimate of the true value then our function approximator will converge to a local optimum under the appropriate conditions.

$$w \leftarrow w + \alpha[\boldsymbol{U_t} - \hat{v}(s, w)]\nabla \hat{v}(s, w)$$

$$U_t \text{Unbiased} \rightarrow W \text{ will converge to a local minimum}$$

We can also replace Ut with a bootstrap target, such as the one step TD target.

$$U_t \doteq R_{t+1} + \gamma \hat{V}(S_{t+1}, w)$$

This is still an estimate of the return, but in this case, the estimate is biased. The TD target uses our current value estimate, which will likely not equal the true value function. Because of this we cannot guarantee this algorithm will converge to a local minimum of the value error. The upside is that the TD target often has lower variance than the sample of the return. This means TD will tend to converge in fewer updates. **The TD update is not actually a stochastic gradient descent update**.

In Fig. 17 is mathematically shown why TD can not be used with Stochastic gradient but a semi-gradient method.

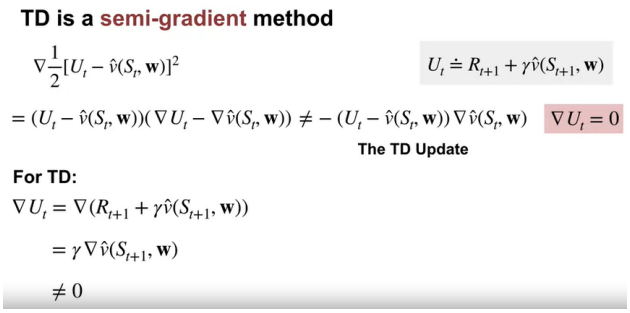

Figure 17: TD as a semi-gradient method

Here is the pseudocode for semi-gradient TD. It's actually very similar to the TD argument for the tabular setting. This algorithm does not have to wait until the end of an episode to make updates. **TD performs an update on each step**, unlike radio Monte Carlo. On each step of the episode the agents selects an action A, in state S. We use the resulting reward in next state to compute the TD target, and update the weights immediately. We continue in this way until we reach the terminal state, which is defined to have value zero, then we start a new episode Fig. 18. The Semi-gradient TD update is also shown in the next equation.

$$w \leftarrow w + \alpha[R + \gamma \hat{v}(S', w) - \hat{v}(s, w)]\nabla \hat{v}(s, w)$$

Remember, TD is not performing gradient descent updates on the squared error due to $U_t = 0$.

Comparing TD and Monte Carlo with State Aggregation

Gradient Monte Carlo will approach a local minimum of the Mean Squared Value Error with more and more samples. This is because it uses an unbiased estimate of the gradient of the value error. In theory, we need to run the algorithm for a very long time and decay the step size parameters to obtain this convergence. In practice, we use a constant step size. So the algorithm **oscillates around a local minimum.**

$$w \leftarrow w + \alpha[G_t - \hat{v}(s, w)]\nabla \hat{v}(s, w)$$

The **TD target** depends on our estimate of the value in the next state. This means **our update could be biased because the estimate in our target may not be accurate**. Since our value approximation will never be perfect even in the limit, the target may remain biased. We cannot guarantee that semi-gradient

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Figure 18: Semi-gradient TD(0)

TD will converge to a local minimum at the Mean Squared value error. Of course, this bias will reduce as their estimates improved.

$$w \leftarrow w + \alpha[R + \gamma\hat{v}(S', w) - \hat{v}(s, w)]\nabla\hat{v}(s, w)$$

TD is less accurate than Monte Carlo but TD converges faster than Monte Carlos because it learns during the episode and has lower variance updates Fig. 19.

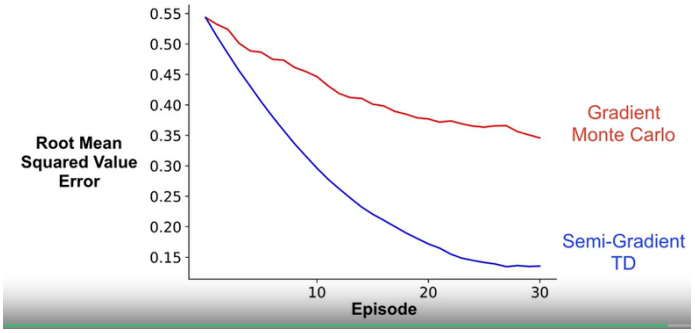

Figure 19: TD convergence vs Monte Carlo

The Linear TD Update

Linear function approximation is an important special case. It is simple enough that we can understand it well, but it is still quite powerful in general. In fact, linear function approximation with TD has been used to build Atari agents that exceed human performance in many games.

Recall the update for semi gradient TD. It adjusts the weights and the direction of the TD target times the gradient of the approximate value function with respect to the weights.

$$w \leftarrow w + \alpha\delta_t\nabla\hat{v}(S_t, w)$$

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(s_t, w)$$

In the linear case, the gradient of the approximate value for a state is just the feature vector for that state.

$$\hat{v}(s_t, w) \doteq w^T x(S_t)$$
$$\nabla \hat{v}(S_t, w) = x(S_t)$$

Finally, the update for semi gradient TD is shown in the next equation.

$$w \leftarrow w + \alpha \delta_t x(S_t)$$

If a feature is large, then the corresponding weight can have a large impact on the prediction. On the other hand if the feature is zero, then that weight has no impact on the prediction and the gradient is therefore zero.

The utility of linear function approximation

- Linear methods are simpler to understand and analyze mathematically
- with good features, linear methods can learn quickly and achieve good prediction accuracy

The True Objective for TD

TD does not precisely optimize the mean squared value error, instead, there's a different objective we can consider for semi gradient TD with linear function approximation. In fact, linear TD provably converges to a well understood approximation called the **TD fixed point**.

To simplify the notation, we'll use X_t to mean the features associated with the state S_t. We can then expand the TD update like this. We can rewrite this using a bit of linear algebra. First, we'll pull X_t into the brackets. We then use the fact that taking the transpose of a scalar, leaves it unchanged. Now, let's think about what this update looks like in expectation. Understanding the expected update is key for proving convergence.

$$\hat{v}(s_t, w) \doteq w^T x(S_t)$$

$$w_{t+1} \doteq w_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(s_t, w_t)]x_t$$

$$w_{t+1} \doteq w_t + \alpha[R_{t+1} + \gamma w_t^T x_{t+1} - w_t^T x_t]x_t$$

$$w_{t+1} \doteq w_t + \alpha[R_{t+1}x_t - x_t(x_t - \gamma x_{t+1})^T w_t]$$

The TD update can be rewritten as the expected update plus a noise term, and so is largely dominated by the behavior of the expected update. The expected update characterizes the expected change in the weight from one time step to the next. The expected TD update can be written as $(b - Aw_t)$. **The matrix A is defined in terms of an expectation over the features, while the vector b is defined in terms of the features and the reward**.

$$\mathbb{E}[\triangle W_t] = \alpha(b - Aw_t)$$

$$b \doteq \mathbb{E}[R_{t+1}x_t]$$

$$A \doteq \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T]$$

The weights are said to converge, when this expected TD update is zero. We call this point W_{TD}. If A is invertible, we can express this as W_{TD}. More generally, $W_{TD} = A^{-1}b$ is a solution to this linear system. We call this solution the **TD fixed point**.

$$W_{TD} = A^{-1}b$$

$$\mathbb{E}[\triangle W_{TD}] = \alpha(b - Aw_{TD}) = 0$$

TD minimizes an objective that is based on A and b as shown in the next equation. This objective extends the connection between TD and Bellman equations, to the function approximation setting. Recall that in the tabular setting, we describe TD as a sample based method for solving the Bellman equation. **Linear TD similarly approximates the solution to the Bellman equation**, minimizing what is called the projected Bellman error. **TD converge to the minimum of a principled objective, based on Bellman equations.**

$$W_{TD} \text{ minimizes } (b - Aw)^T(b - Aw)$$

Nonetheless, we do still want to understand the relationship between the solution found by TD and the minimum value error solution. We can formally characterize this relationship with this equation. The difference between the TD fixed point and the minimum value error solution can be large if γ is close to one. If γ is very close to zero on the other hand, the TD fixed point is very close to the minimum value error solution.

$$\overline{VE}(W_TD) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(w)$$

If we can perfectly represent the value function, then regardless of γ, the TD fixed point is equivalent to the minimum value error solution. This is because both the left and right-hand sides would be zero.

TD fixed point is not equal to the minimum value error solution because of bootstrapping under function approximation. If our estimate of the next state is persistently inaccurate due to function approximation, then TD forever update towards an inaccurate target. On the other hand, if our function approximator is very good, then our estimate of the next state will become very accurate.

2 Constructing Features for Prediction

Coarse coding

The features used to construct value estimates are one of the most important parts of a reinforcement learning agent. Recall that a tabular representation can be expressed as a binary feature vector. Each state is associated with a different feature. If the agent is in a state then the feature corresponding to that state is 1. All other features are 0. The tabular case is just a special case of linear function approximation where the feature vector is an indicator or one hot encoding of the state Fig. 20. **One hot encoding is not feasible when the state space is too big.**

It is important to notice that we can use any shape in state aggregation, not only square ones to aggregrate groups. Nevertheless, it is not possible to overlap shapes Fig. 21.

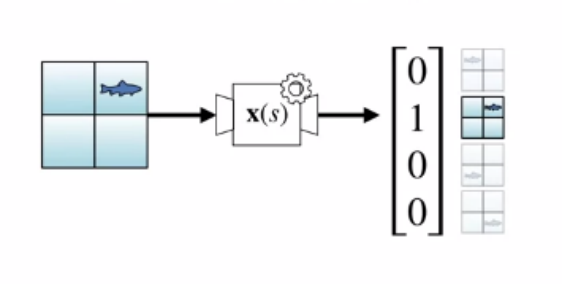

Figure 20: Recalling linear value function approximation

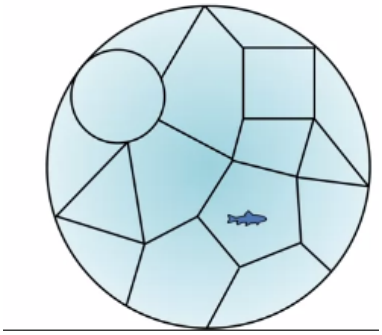

Figure 21: State aggregation with irregular shapes

In general we can aggregate states using any shapes we want as long as those shapes do not have any gaps or overlap. 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 **Coarse coding** Fig. 22.

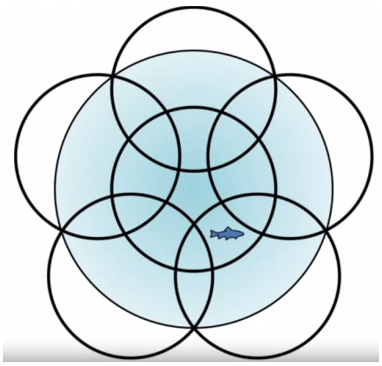

Figure 22: Coarse coding

Let's look at the example feature vector for the fishes current location in the pond. Remember each index in the feature vector corresponds to one of the shapes. The feature corresponding to the circle is active or set to 1 if the fish is within that circle, otherwise the feature set to 0. The features receptive field corresponds to the locations that activate that feature Fig. 23.

Nearby states will have similar feature activation, but they may also have different components active including different numbers of active features.

- Tabular states can be represented with a binary one-hot encoding

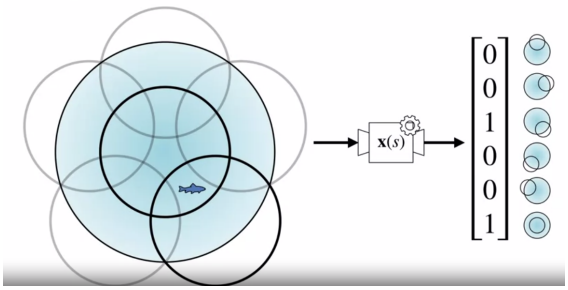

Figure 23: Coarse coding example

- Coarse coding is a generalization of state aggregation

Generalization Properties of Coarse Coding

Coarse coding affects generalization and discrimination. 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. The larger circles on the right generalize more broadly distributing the update across a larger number of states Fig. 24.

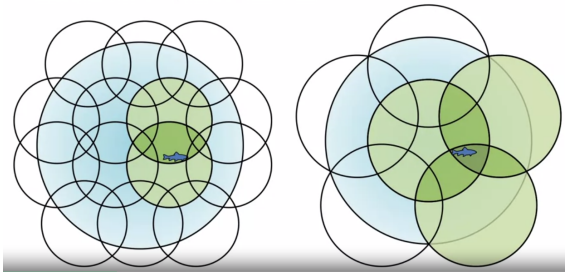

Figure 24: Broadness of generalization

Generalization is not just a scalar quantity however. Using **different shapes in coarse coding** can change the direction of generalization as well.

Coarse coding generalizes with vertically elongated ellipses. The receptive fields made from ellipses are longer than they are wide. With these ellipses, coarse coding primarily generalizes in the vertical dimension Fig. 25.

Shape and size impact generalization and the speed of learning.

Recall that 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. The colored shapes depict the discriminative ability of this particular coarse coding Fig. 26. 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. 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

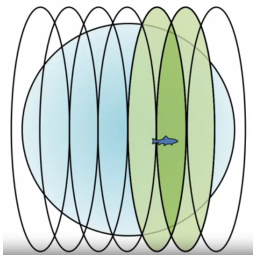

Figure 25: Direction of generalization

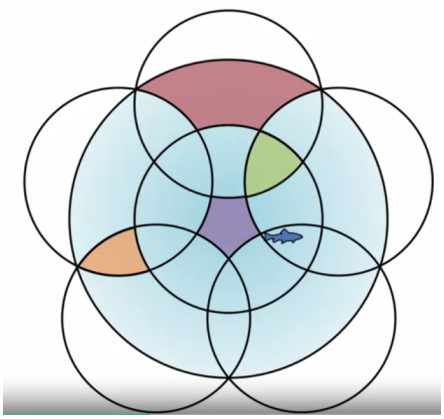

Figure 26: State discrimination

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

Tile coding

Tile coding is a computationally efficient way to perform Coarse Coding. Tile coding is a specific type of Coarse Coding. Tile coding performs an exhaustive partition of the state space using overlapping grids.

Unlike Coarse coding which uses arbitrary shapes, **tile coding uses squares**. What's the most convenient way to lay out a bunch of squares over space? It's 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 Fig. 27.

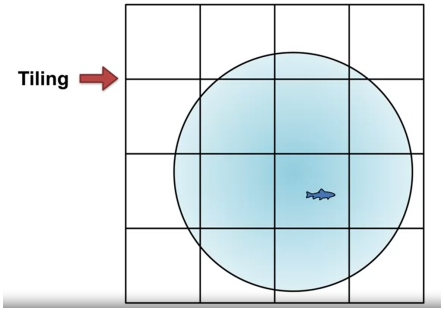

Figure 27: Tiling in Tile coding

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 Fig. 28.

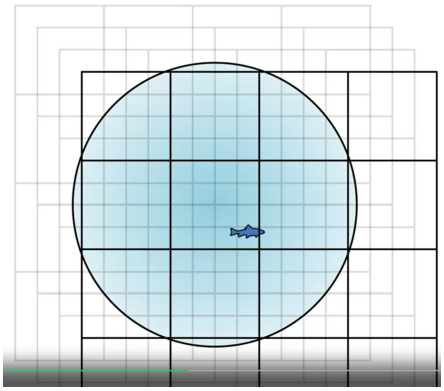

Figure 28: Overlaping Tilings for better discrimination

Let's see 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 Fig. 29.

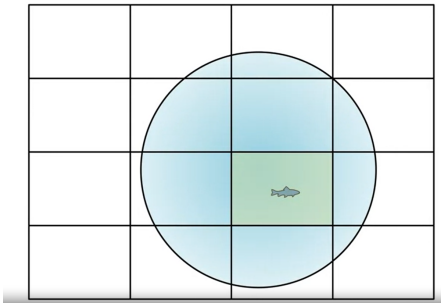

Figure 29: Direction of generalization

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.

Using Tile Coding in TD

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.

Recall the next equation:

$$v_\pi(s) \approx \hat{v}(s_t, w) \doteq w^T x(S)$$

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. 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 Fig. 30.

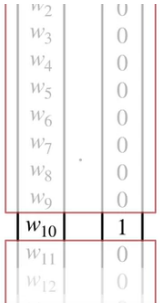

Figure 30: Sparse binary representation

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.

$$\hat{v}(s_t, w) = W_{10} + W_{26} + W_{42} + W_{53}$$

In the next example we can see the fish intersects square one belonging to the purple tiling and square four belonging to the orange tiling. This results in the falling feature vector for this state. Now we simply calculate the state's value with a dot product. The feature vectors binary. So we can just add up the weights at the locations with non-zero features. This gives us a value of two Fig. 31.

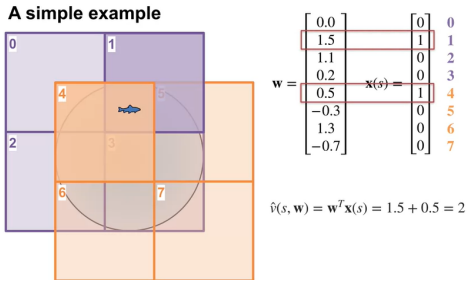

Figure 31: Tile coding example

Tile coding has a better discrimination ability than state aggregation

What is a Neural Network?

Now we will talk about a simple neural network. Each of these circles represents nodes in the network. Each of the lines represents connections between nodes. The nodes are organized into layers. When new data comes in, it starts in the input layer and is sent through the connections to the next layer of nodes. This layer performs some computation on the data then sends the results to the next layer. This process continues until the last layer produces the final output. We call this a feed forward neural network because the data always moves forward through the layers Fig. 32.

When data is passed through a connection, a weight is applied. The node then sums up each of its weighted inputs, and apply some activation function to this sum. The activation function is often a nonlinear transformation. Common activation functions include sigmoidal functions, like tanh or the logistic function, rectified linear units or ReLU, or even thresholding units Fig. 33.

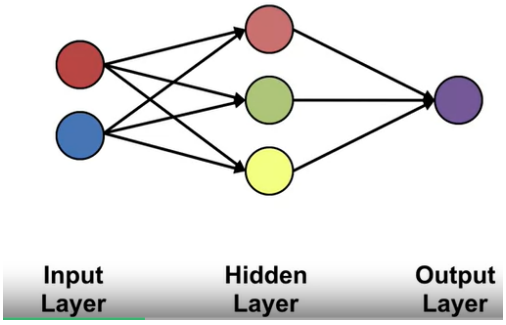

Figure 32: Simple Neural Network

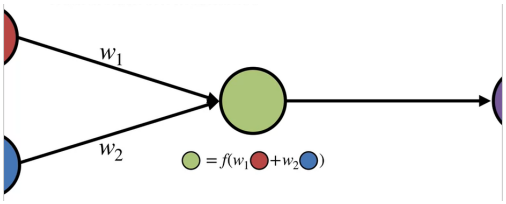

Figure 33: Neural Network computation

Let's take a look at how to write a feed forward pass of a neural network mathematically. At each node we have two vectors, the inputs and the weights for each input. The first subscript refers to the node that connection comes from. The second subscript refers to the node the connection feeds into. We don't product the weights with the input and pass the result through the activation function. Notice here that S is a row vector, so we do not need to write the dot-product with a transpose. Each layer consists of many such nodes. A neural network is just a parameterized function Fig. 34.

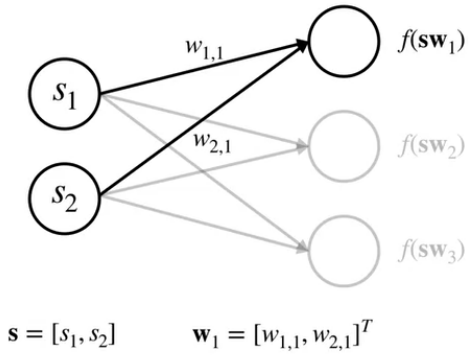

Figure 34: Neural network implementation

The next mathematical expressions represents the math behind a neural network.

$$s = [s_1, s_2] \text{ Input vector}$$

$$W = [W_1 W_2 W_3] \text{ Weights Matrix}$$

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

If the current output is not the final layer, this output vector becomes the input to the next layer, and we repeat the process.

$$\text{Outputs} = f(sW)$$

Non-linear Approximation with Neural Networks

When we first build the neural network, we need to specify the initial weights. The way we initialize the weights is important. For now, let's just assume there are drawn from some random distribution.

$$W_{init} \sim \mathcal{N}(\mu, \sigma)$$

Let's take a look at what happens to a given input vector as we pass it through the network. Let's start by looking at only one node of the network. Each of the inputs are multiplied their corresponding weights. Then, the weighted sum of these inputs is passed to a non-linear activation function resulting in a non-linear function of the inputs Fig. 35. This process is done again and again for each of the nodes in the layer.

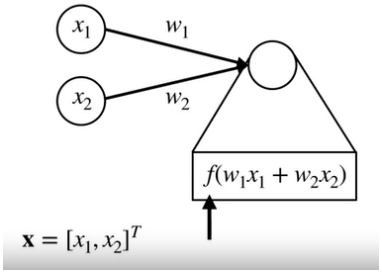

Figure 35: Non-linear representation

Each node has a different set of weights, so will produce a different output, which we call a feature. All of these new features, collectively, are considered the new representation Fig. 36.

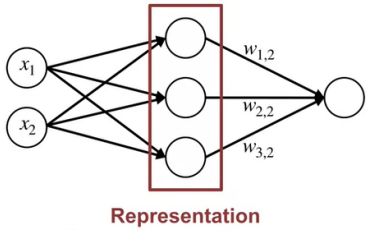

Figure 36: Features in a neural network

This process (neural networks) 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.

Recall that 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.

Deep Neural Networks

We can view a neural network as being a **modular system** where each layer is a module. We can add and remove layers and we can change the types of layers that we use. The depth of the network is defined by the number of hidden layers in the network. In theory, a neural network need not be deep. A neural network with a single hidden layer can approximate any continuous function given that is sufficiently wide (wide = number of nodes in a layer). We call this the universal approximation property.

Practical experience and theory suggests that deep neural networks may make it easier to approximate complex functions. One reason for this is that the depth allows composition of features. Composition can produce more specialized features by combining modular components Fig. 37.

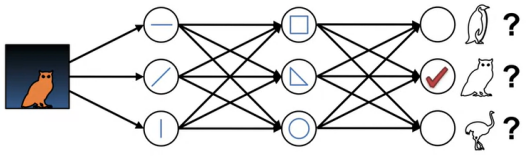

Figure 37: Compositional features

Depth in a network can significantly improve our agent's ability to learn features.

Gradient Descent for Training Neural Networks

The first step is to define a loss on the parameters of the neural network and then derive the gradient. **The loss function specifies how far the networks predictions are from being correct.** Our goal in training the neural network, is to find the parameters which minimize this loss function Fig. 38. The gradient of a function points in the direction of greatest ascend. By moving in the direction opposite the gradient, we move in the direction that most quickly minimizes the loss.

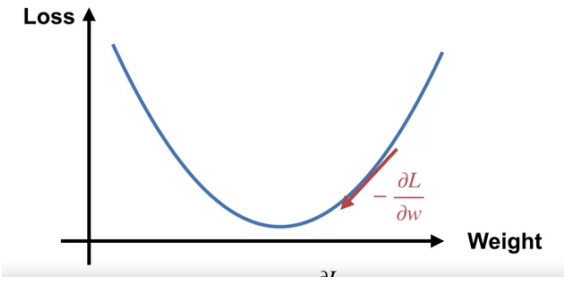

Figure 38: Gradient Descent

$$w \leftarrow w - \frac{\partial L}{\partial w}$$

The input to the network is S and the output is $\hat{Y}$. The hidden layer is the learned features. So let's call it x. The weights A produce the features, the weights B linearly weight S to produce the output. The output of each layer of the network can be represented as a vector. Here, that is X and $\hat{Y}$. The first index of the weight matrices refers to the inputs to that layer, and the second index refers to the outputs Fig. 39.

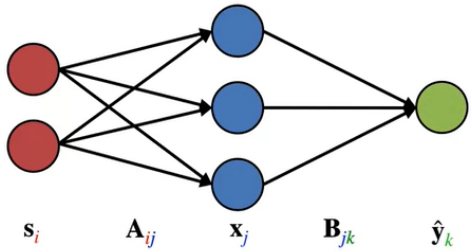

Figure 39: Neural networks notation

We assume for now we have a generic loss L, so we can describe the basic idea. An example of L is the squared error. For the derivation, we will keep L generic.

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

Each matrix of parameters has an update that looks like an error term δ times the input to that layer. For A the input is S, for B the input is x Fig. 39. Further, we will find that δ^A can be efficiently computed as a function of δ^B. The errors δ^B from the output of the network are propagated backwards to this earlier layer to help determine the role A had in producing that error.

$$A = A - \alpha\delta^A s$$

$$B = B - \alpha\delta^B x$$

Let's start at the output of the network and work backwards. Recall:

$$x \doteq f_A(sA)$$
$$\hat{y} \doteq f_B(xB)$$

We start by taking the partial derivative of the loss function with respect to the first set of weights B. We use the chain rule given the derivative of L with respect to $\hat{Y}$ times $\frac{\partial \hat{y}}{\partial B}$. The next step is again to use the chain rule for this derivative.

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial B_{jk}}$$

let's introduce a new variable, θ. θ is the output of the hidden layer times the last set of weights.

$$\theta \doteq xB$$

Thus

$$\hat{y} \doteq f_B(\theta)$$

Rewriting $\hat{y}$ we have:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k} \frac{\partial \theta_k}{\partial B_{jk}}$$

Remember that

$$\frac{\partial \theta_k}{\partial B_{jk}} = x_j$$

Therefore, the gradient of the weights B would be:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k} x_j$$

Let's make some choices for the loss and activation so that we can see a specific instance of this general equation for B. Let's define L to be the squared error and use a linear activation on the last layer.

Loss:

$$L = \frac{1}{2}(\hat{y}_k - y_k)^2$$

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} = (\hat{y}_k - y_k)$$

Activation Function:

$$f_B(\theta_k) = \theta_k$$

$$\frac{\partial f_B(\theta_k)}{\partial \theta_k} = 1$$

Plugin this into the last equation we will have:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = (\hat{y}_k - y_k) x_j$$

To ease notation while computing the gradient with respect to A, let's define a new term δ_k^B. We can replace most of the gradient for B so that it now becomes:

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

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k} x_j = \delta_k^B x_j$$

Now, to compute the gradient with respect to A, we need to go through the same steps as for B. The main difference is that we have one extra chain rule step because the weights A also affect x.

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \delta_k^B \frac{\partial \theta_k}{\partial B_{jk}}$$

For A:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B \frac{\partial \theta_k}{\partial A_{ij}}$$

$$\frac{\partial \theta_k}{\partial A_{ij}} = \frac{\partial \theta_k}{\partial x_j} \frac{\partial x_j}{\partial A_{ij}} = B_{jk} \frac{\partial x_j}{\partial A_{ij}}$$

Therefore:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B B_{jk} \frac{\partial x_j}{\partial A_{ij}}$$

let's introduce another helper variable, ψ.

$$\psi \doteq sA$$

$$x \doteq f_A(\psi)$$

Rewriting and deriving we would have:

$$\frac{\partial x_j}{\partial A_{ij}} = \frac{\partial f_A(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial A_{ij}}$$

Thus:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B B_{jk} \frac{\partial f_A(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial A_{ij}}$$

Finally, as:

$$\frac{\partial \psi_j}{\partial A_{ij}} = s_i$$

Then

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_k^B B_{jk} \frac{\partial f_A(\psi_j)}{\partial \psi_j} s_i$$

We can clean up this derivative by again, defining a term δ_A.

$$\delta_j^A = (B_{jk} \delta_k^B) \frac{\partial f_A(\psi_j)}{\partial \psi_j}$$

The final result will be:

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_j^A s_i$$

Obtaining as a final result for both gradients the next expressions

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial B_{jk}} = \delta_k^B x_j$$

$$\frac{\partial L(\hat{y}_k, y_k)}{\partial A_{ij}} = \delta_j^A s_i$$

Let's take a brief look at some pseudocode Fig. 40. for implementing the backprop algorithm with Stochastic gradient descent. For each data point s, y in our dataset, we first get our prediction $\hat{y}$ from the network. This is typically called the forward pass, then we compute the gradients starting from the output. We first compute δ^B and the gradient for B, then we use this gradient to update the parameters B, the step size α_B. Next, we update the parameters A. We compute δ_A which uses δ_B. Notice, that by computing the gradients of the end of the network first, we avoid recomputing the same terms for A, that were already computed for δ_B. We then compute the gradient for A and update A with this gradient using step size α_A.

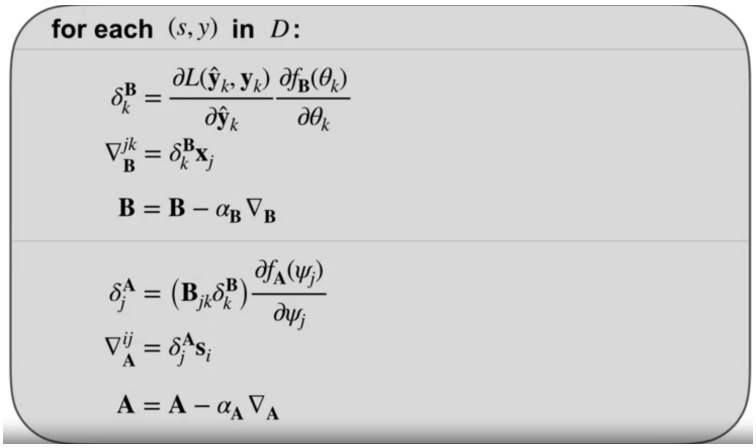

Figure 40: Pseudo-code for back propagation

Here's the pseudocode if we use the ReLU activation on the hidden layer and a linear unit for the output. First, we compute the error for the output layer, then we compute the derivative of the ReLU units with respect to ψ, and finally, we use the aerial signal from the output layer along with you to compute the air signal for the hidden layer, the rest remains the same Fig. 41.

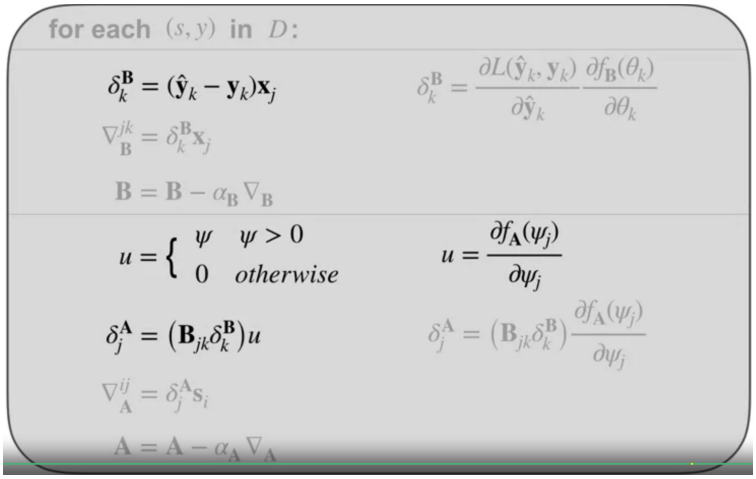

Figure 41: Pseudo-code with example

Optimization Strategies for NNs

One simple yet effective initialization strategy for the weights, is to randomly sample the initial weights from a normal distribution with small variance Fig. 42. This way, each neuron has a different output from other

neurons within its layer. This provides a more diverse set of potential features. By keeping the variants small, we ensure that the output of each neuron is within the same range as its neighbors. One downside to this strategy is that, as we add more inputs to a neuron, the variance of the output grows.

$$W_{init} \sim \mathcal{N}(0, 1)$$

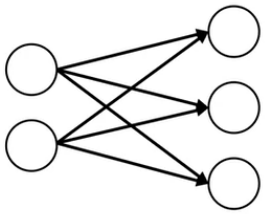

Figure 42: Weights initialization

We can get around the variance output issue by scaling the variance of the weights with the next equation.

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

Another way to improve training, is to consider a more sophisticated update mechanisms. Two common strategies, are to use the heavy-ball method also called momentum and vector step size adaptation. Notice, that this equation is similar to the regular stochastic gradient descent update plus an extra term called the momentum M.

$$W_{t+1} \leftarrow W_t - \alpha \nabla_w L(W_t) + \boldsymbol{\lambda M_t}$$
$$M_{t+1} = \lambda M_t - \alpha \nabla_w L$$

The momentum term summarizes the history of the gradients using a decaying sum of gradients with decay rate λ. 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. Momentum provably accelerates learning, meaning it gets to a stationary point more quickly.

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 Fig. 43.

To implement this, 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.

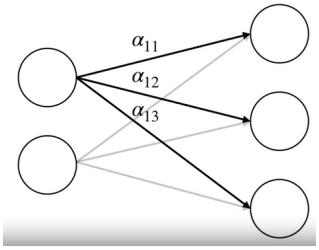

Figure 43: Vector step size

3 Prediction and Control with Function Approximation

Episodic Sarsa with Function Approximation

Recall that the value function approximation has two components, a weight vector and a feature vector. In a given state, the value estimate is the dot product between these two components.

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

To move from TD to Sarsa, we need action value functions. So the feature representation has to represent actions as well.

$$q_\pi(s) \approx \hat{q}(s, a, w) \doteq w^T x(s, a)$$

One way to do this is to have a separate function approximator for each action. This can be accomplished by stacking the features. That is, we can use the same state features for each action, but only activate the features corresponding to that action.

Let's look at an example. Let's say there are four features and three actions. The four features represent the state you are in. But we want to learn a function of both states and actions. We can do this by repeating the four features for each action. Now the feature vector has 12 components. Here, each segment of four features corresponds to a separate action. We call this feature representation stacked because the weights for each action are stacked on top of each other. Thus, only the features for the specified action will be active, while though for the other actions will be set to 0 Fig. 44.

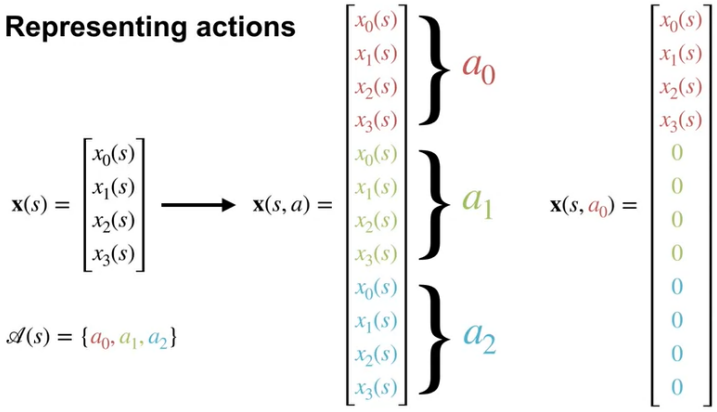

Figure 44: Representing actions

Let's see an example of how to calculate the action values from a given state. Let's say that there are four features and three actions. The weight factor looks like this. With stacked features, we get the following feature vector for action two in state s. We 0 out the features for the other actions. So we extract the segment of the weight vector corresponding to each action. The action values are then the dot products between each segment of the weight vector and the feature vector Fig. 45.

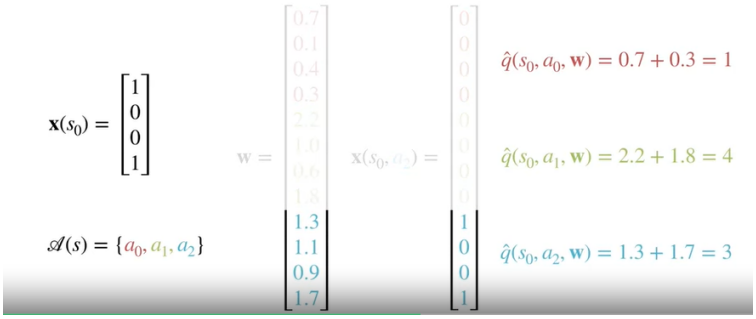

Figure 45: Computing action values

The common way to represent action values with a neural network is to generate multiple outputs, one for each action value. This, however, is equivalent to the stacking procedure described above.

We might want to generalize over actions for the same reason generalizing over state can be useful. We would input both the state and the action to the network. There would only be one output. The approximate action value for that state and action. We can do something similar with tile coding by passing both the state and action as input Fig. 46.

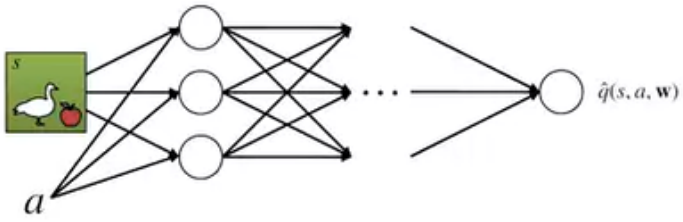

Figure 46: Computing action values with a neural-network

Let's talk about using Sarsa for control of function approximation. The algorithm quite similar to the tab diversion, so we will just review the differences. We use parameterized action value functions for the action value estimates. The update also changes to use the gradient to update the weights similar to similar gradient TD Fig. 47.

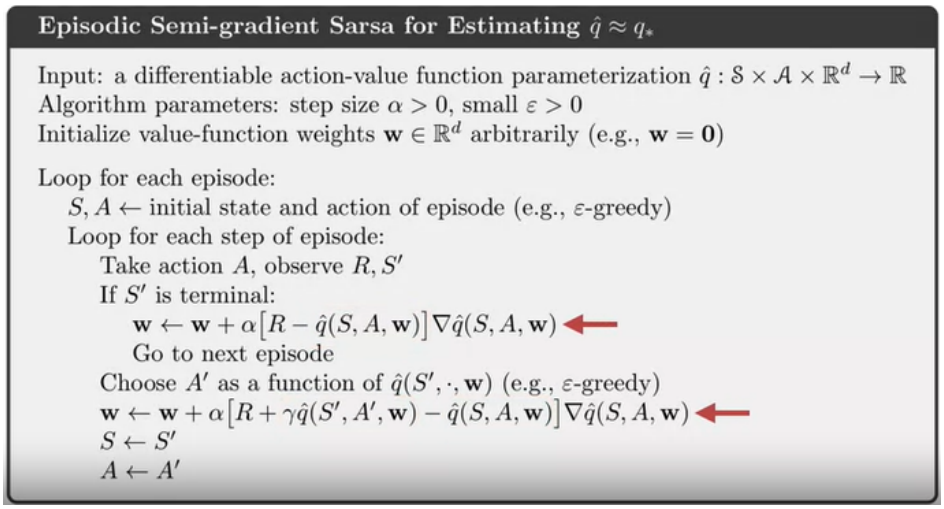

Figure 47: Episodis Semi-gradient Sarsa Pseudo-code

Expected Sarsa with Function Approximation

Let's see how we can turn Sarsa into expected Sarsa when using function approximation. Recall the update equation for Sarsa. Sarsa's update target includes the action value for the next state in action.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Recall that expected Sarsa instead uses the expectation over its target policy.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1})Q(S_{t+1}, a') - Q(S_t, A_t))$$

The same expectation can be computed even when we use function approximation. First, recall the update for Sarsa with function approximation. It looks similar to the tabular setting except the action value estimates are parameterized by the weight factor, W.

Sarsa

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

Expected Sarsa with function approximation follows a similar structure.

Extected Sarsa

$$w \leftarrow w + \alpha[R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1})\hat{q}(S_{t+1}, a', w) - \hat{q}(S_t, A_t, w)]\nabla\hat{q}(S_t, A_t, w)$$

Finally, for Q-learning with function approximation as Q-learning is a special case of expected Sarsa then we have:

Q-learning

$$w \leftarrow w + \alpha[R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', w) - \hat{q}(S_t, A_t, w)]\nabla\hat{q}(S_t, A_t, w)$$

Exploration under Function Approximation

Let's do a brief refresher on how we use optimistic initial values in the tabular setting. We initialize our values to be greater than the true values. This is like the agent imagining that it can get more reward by taking that action than it actually can in reality. Typically, initializing the value function this way causes the agent to systematically explore the state action space. As the agents values become more accurate, they are impacted less and less by this initialization. This is straightforward to implement in a **tabular setting** where the update to each state action pair is independent of all the other state action pairs.

$$Q(s,a) \leftarrow 1000 \ \forall s, a$$

In function approximation, optimistic initial values corresponds to initializing the weights such that the resulting values are optimistic. In some cases this is straightforward, for example, when the features are binary, we simply initialize each weight to be the largest possible return. Then, as long as each state has at least one feature active, the value will be optimistic and likely overly so.

Linear

$$q_\pi(s) \approx \hat{q}(s,a,w) \doteq w^T x(s,a)$$

$$w \leftarrow= \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \\ \vdots \end{bmatrix}$$

In many cases however, it is difficult to initialize optimistically. For example, in a neural network the relationship between the final values and the features can be quite complicated.

Non-Linear

$$q_\pi(s) \approx \hat{q}(s,a,w) = NN(s,a,w)$$

$$w \leftarrow ???$$

Depending on how our features generalize, optimistic initial values may not result in the same kind of systematic exploration we see in the tabular case.

Consider an extreme example, where we have only one feature that is always one. We can initialize optimistically but every update will change the value for all states. This means that before some states are even visited, the value will already have decreased such that it is no longer optimistic. To facilitate systematic exploration, changes to the value function need to be more localized. For example, function approximation with tile coding can produce such localized updates. Neural networks and also provide local updates, but neural networks may also generalize aggressively. In practice without special consideration, a neural network will lose his optimism relatively quickly Fig. 48.

Epsilon greedy is generally applicable and easy to use even in cases with non-linear function approximation. The only thing Epsilon greedy needs are the action value estimates, independent of how they are initialized or approximated. However, Epsilon greedy is not a directed exploration method. It relies on randomness to discover better actions near states followed by the current policy. It is therefore not as systematic as exploration methods that rely on optimism Fig. 49.

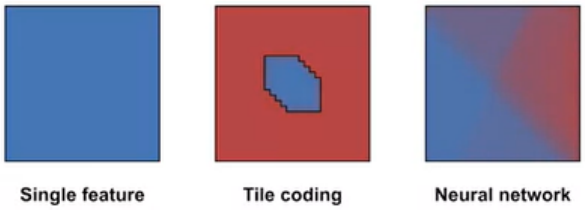

Figure 48: How optimism interacts with generalization

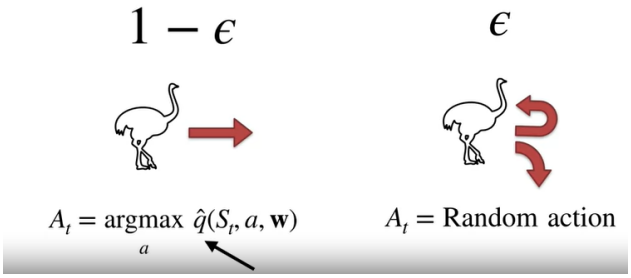

Figure 49: ϵ Greedy

Average Reward: A New Way of Formulating Control Problems

In general, the only way to ensure that the agents actions maximize reward over time is to keep increasing the discount factor towards 1. Depending on the problem, we might need gamma to be quite large. And remember, we can't set it to 1 in a continuing setting because then the return might be infinite. Larger values of gamma can also result in larger and more variables sums, which might be difficult to learn.

Let's discuss a new objective called the average reward. Imagine the agent has interacted with the world for h steps. This is the reward it has received on average across those h steps. In other words, it's rate of reward. If the agents goal is to maximize this average reward, then it cares equally about nearby and distant rewards. We denote the average reward of a policy with $r(\pi)$.

$$r(\pi) \doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathrm{E}[R_t \; S_0, A_{0:t-1} \approx \pi]$$

We can write the average reward using the state visitation, μ. This inner term is the expected reward in a state under policy π. The outer sum takes the expectation over how frequently the policy is in that state. Together, we get the expected reward across states. In other words, the average reward for a policy.

$$r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r$$

In the nearsighted example, the two deterministic possible policies visit either the left loop or the right loop indefinitely. In both cases, the five states in each loop are visited equally many times. In the left loop, the immediate expected reward is 0 for all states except one, which gets 1. This results in an average reward of $\frac{1}{5} = 0.2$. Most states in the right loop also have 0 to expected reward. But this time, the last date gets 2. This gives an average reward of $\frac{2}{5} = 0.4$. We can see the average reward puts preference on the policy that receives more reward in total without having to consider larger and larger discounts Fig. 50.

In the average reward setting, returns are defined in terms of differences between rewards and the average

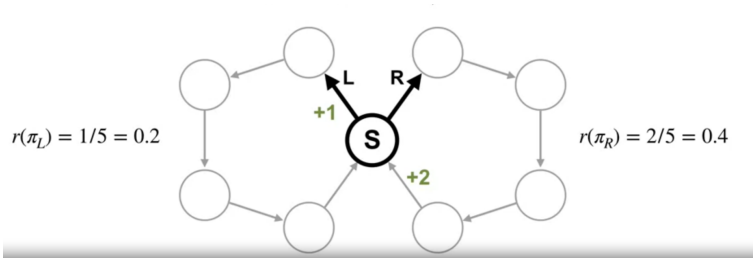

Figure 50: Average reward example

reward r_π, which is called **the differential return**. The differential return represents how much more reward the agent will receive from the current state in action compared to the average reward of the policy.

$$G_t = R_{t+1} - r_\pi + R_{t+2} - r_\pi + R_{t+2} - r_\pi \dots$$

The differential return represents how much better it is to take an action in a state then on average under a certain policy. The differential return can only be used to compare actions if the same policy is followed on subsequent time steps. To compare policies, their average reward should be used instead. Interestingly, the differential return is only a convergent sum if the subtracted constant is equal to the true average reward. If a lower or higher number is subtracted, the sum will diverge to positive or negative infinity.

Now that we have a valid definition of the return for average reward, we define value functions in the usual way, as the expected return. Similarly, we can also define differential value functions as the expected differential return under a policy from a given state or state action pair. This quantity captures how much more reward the agent will get by starting in a particular state than it would get on average over all states if it followed a fixed policy.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Like in the discounted setting, differential value functions can be written as Bellman equations. They only differ in that they subtract $r(\pi)$ from the immediate reward and there is no discounting.

$$q_\pi(s, a) = \sum_{s', r} p(s', r | s, a)(r - r(\pi) + \sum_{a'} \pi(a' | s') q_\pi(s', a')$$

Many algorithms from the discounted case can be rewritten to apply to the average reward case. For example, differential Sarsa is very similar to the Sarsa algorithm you've seen before. Let's step through the differences. A key difference is that differential Sarsa has to track an estimate of the average reward under its policy and subtract it from the sample reward in its update. This implementation does so with the incremental averaging techniques we've seen throughout the course. Given this estimate, it then subtracts $\overline{R}$ from the sampled reward in its update. In practice, we can get better performance with a slight modification to this algorithm. Instead of the exponential average of the reward to compute $\overline{R}$, we use this update which has lower variance Fig. 51.

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Figure 51: Differential Sarsa Pseudo-code

4 Policy Gradient

Learning Policies Directly

Previously, we used epsilon-greedy to convert approximate action values into a policy. But we can also consider **policy which maps states** directly to actions without first computing action values.

For example, in mountain car, we can define such a policy. Simply choose accelerate right when the velocity is positive and otherwise, choose the accelerate left action. Put another way, this policy accelerates in whatever direction we are already moving. In fact, this simple energy pumping policy, is close to optimal Fig. 52. We're not actually going to specify policies by hand. Rather, we will learn them.

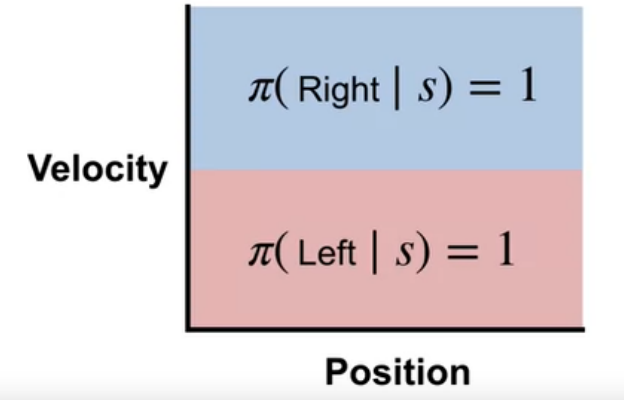

Figure 52: a policy without action values

We can use the language of function approximation to both represent and learn policies directly. We'll use θ for the policies parameter vector. This distinguishes it from the parameters W for the approximate value function. We use the notation Pi of a given s and Theta, to denote the parameterized policy. For a given input state and action, the parameterized policy function will output the probability of taking that action in that state Fig. 53.

The parameterized function, has to generate a valid policy. This means, it has to generate a valid probability

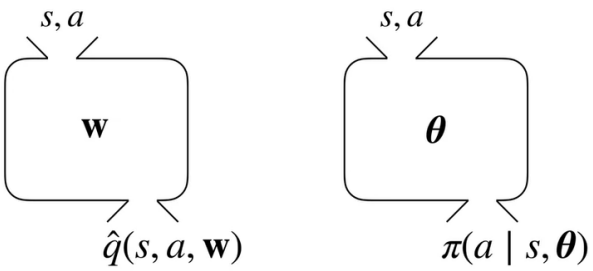

Figure 53: Parametirizing policies directly

distribution over actions for every state. Specifically, the probabilities selecting an action, must be greater than or equal to zero. For each state, the sum of the probabilities over all actions must be one.

$$\pi(a|s,\theta) \geq 0 \quad \forall a \in A \text{ and } s \in S$$

$$\sum a \in A \pi(a|s,\theta) = 1 \quad \forall s \in S$$

Let's consider a simple but effective way to satisfy the last conditions. Here's the definition of a softmax policy. The function $h(s, a, \theta)$ shown here, is called the **action preference**. **A higher preference for a particular action in a state, means that the action is more likely to be selected.** The action preference is a function of the state and action as well as a parameter vector Theta. Computing the probability of selecting an action with the softmax is simple. We take the action preference, exponentiate it, and then divide by the sum over all the actions for the same thing.

$$\pi(a|s,\theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_{b \in A} e^{h(s,b,\theta)}}$$

The action preference, can be parameterized in any way we like since the softmax will enforce the constraints of a probability distribution. For example, the action preferences could be a linear function of the state action features or something more complex like the output of a neural network.

$$< x(s,a), \theta >$$

Here's what we get when we pass a particular set of action preferences through the softmax. The input preferences can be arbitrarily large or even negative. If one preference is much larger than all the others, the action probability would be close to one. No matter how big the preference gets, the probability will never be greater than one. If one preference is very small, the softmax policy will still select the action with non-zero probability. For example, if the preference is negative and the other is positive, the negative action will still have non-zero probability. Finally, actions with similar preferences will be chosen with near equal probability under a softmax policy Fig. 54.

It's important to distinguish between action preferences and action values. **Preferences indicate how much the agent prefers each action but they are not summaries of future reward**. Only the relative differences between preferences are important. For example, we could add plus 100 to all the preferences and it would not matter. An epsilon-greedy policy derived from action values can behave very differently than a softmax policy over action preferences. In epsilon-greedy, the action corresponding to the highest valued action, is selected with high probability. The probability selecting all the other actions, is quite small. Actions with nearly the same but lower action value, are selected with much lower probability. On the other hand, actions with very poor action values, are still selected frequently due to the epsilon exploration step. So even if the agent learns an action has terrible consequences, it will continue to select that action much more frequently than it would under the softmax policy Fig. 55.

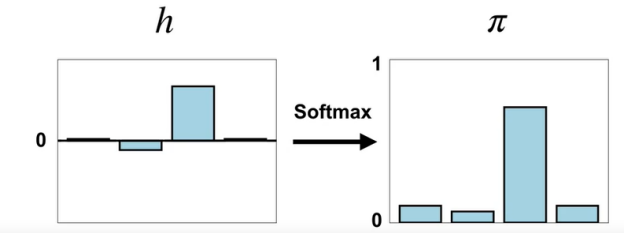

Figure 54: Softmax policies

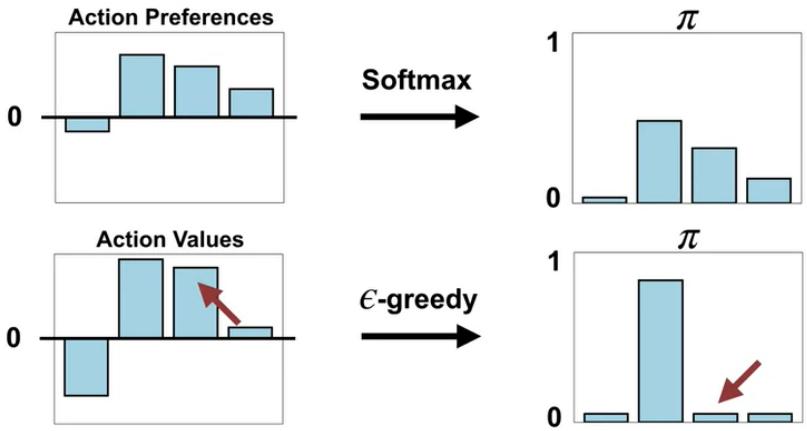

Figure 55: Action preferences are not action values

Advantages of Policy Parameterization

There are many advantages to directly learning the parameters of a policy. **First, the agent can make its policy more greedy over time autonomously.** Why would you want that? Well, in the beginning, the agent's estimates are not that accurate. So you would want the agent to explore a lot. As the estimates become more accurate, the agent should become more and more greedy.

We can avoid the issue of epsilon greedy policies taking random action and not being optimal with parameterized policies. The policy can start off stochastic to guarantee exploration. Then as learning progresses, the policy can naturally converge towards a deterministic greedy policy. A softmax policy can adequately approximate a deterministic policy by making one action preference very large Fig. 56.

$$\pi(a|s,\theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_{b \in A} e^{h(s,b,\theta)}}$$

Stochastic policies can be useful with function approximation.

Sometimes the policy is more simple than the value function. Remember the mountain car problem. We spent a lot of time designing a value-based agent to learn an optimal policy. However, in this problem, the energy pumping policy we saw previously is nearly optimal. The agent simply selects its action in agreement with the current velocity. If the velocity is negative, then the agent takes the accelerate left action. If the velocity is positive, then the agent takes the accelerate right action. This policy allows the agent to quickly escape from the valley. As you can see, this is quite a simple policy Fig. 57.

Parameterized stochastic policies are useful because:

- they can autonomously decrease exploration over time

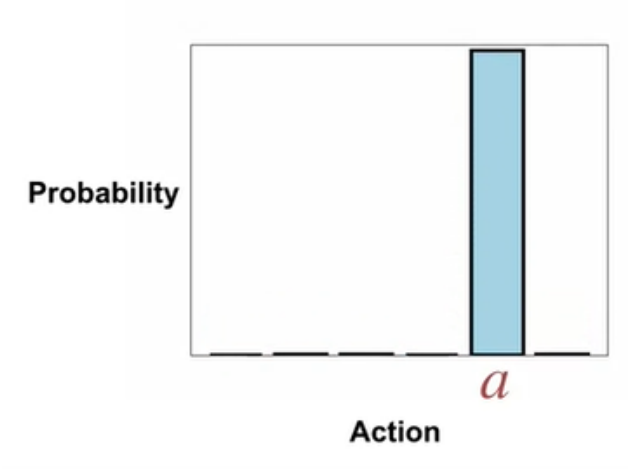

Figure 56: The flexibility of stochastic policies

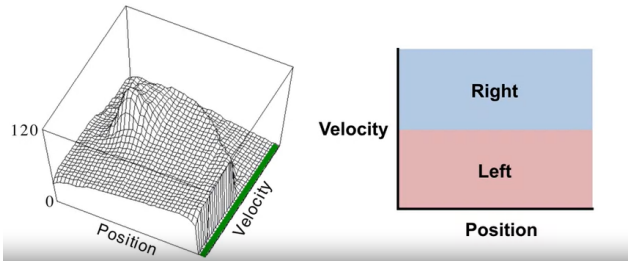

Figure 57: Policy vs Value function

- They can avoid failures due to deterministic policies with limited function approximation.
- Sometimes the policy is less complicated than the value function.

The Objective for Learning Policies

To improve a parameterized policy it is just like with action value-based methods, the basic idea will be to specify an objective, and then figure out how to estimate the gradient of that objective from an agent's experience.

Formulating an objective for learning a parameterized policy is in some sense more straightforward than it was for action value-based methods. When we parameterize our policy directly, we can also use this goal directly as the learning objective.

$$R_t + R_{t+1} + R_{t+2} \dots$$

Recall the next equations of return:

Episodis:

$$G_t = \sum_{t=0}^{T} R_t$$

Continuing:

$$G_t = \sum_{t=0}^{T} \gamma^t R_t$$

Continuing with average reward:

$$G_t = \sum_{t=0}^{T} R_t - r(\pi)$$

We're going to restrict our attention to the continuing setting with average reward. Remember that the objective is to find a way to directly optimize the parameters of a policy. The first step is to write the average reward objective in a form that we can optimize.

Previously, we estimated the average reward to learn action values in an average reward variant of sarsa. Now, our aim is to learn a policy that directly optimizes average reward.

We can write the average reward achieved by a particular policy π.

$$r(\pi) = \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r$$

This inner sum gives the expected reward if we start in state S and take action A. This is simply the sum over all rewards we might receive weighted by their probability from S and A. We sum over next states s' to get marginal probabilities over the reward.

$$\sum_{s',r} p(s',r|s,a) r = \mathbb{E}[R_t | S_t = s, A_t = a]$$

The next level of the summation is over all possible actions weighted by their probability under π. This gives us the expected reward under the policy π from a particular state S.

$$\sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r = \mathbb{E}_\pi[R_t | S_t = s]$$

Finally, we get the overall average reward by considering the fraction of time we spend in state S under policy π. The distribution μ provides these probabilities. The expected reward across states is a sum over S of the expected reward in a state weighted by $\mu(s)$, $r(\pi)$ is our average reward learning objective.

$$\sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r = \mathbb{E}[R_t]$$

Our goal of policy optimization will be to find a policy which maximizes the average reward. Our basic approach will be to estimate the gradient of the objective with respect to the policy parameters and adjust the parameters based on this estimate. The class of methods they use this idea are often referred to as **policy gradient methods.**

$$\nabla r(\pi) = \nabla \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r$$

Now, we're interested in learning policies directly. There's also a superficial difference, in that before we were minimizing the mean squared value error, and now we are maximizing an objective.

There are few challenges in computing this gradient however, the main difficulty is that modifying our policy changes the distribution μ.

$$\mu(s) \text{ depends on } \theta$$

This contrast value function approximation where we minimized means grid value error under a particular policy. There the distribution μ was fixed. It does not change as the weights and the parameterized value function chains. We were therefore able to estimate gradients for states drawn from μ by simply following the policy.

$$\nabla_w \overline{VE} = \nabla_w \sum_s \mu(s)[v_\pi(s) - \hat{v}(s,w)]^w$$

$$\nabla_w \overline{VE} = \sum_s \mu(s) \nabla_w [v_\pi(s) - \hat{v}(s,w)]^w$$

Luckily, there's an excellent theoretical answer to this challenge called **the policy gradient theorem**.

The Policy Gradient Theorem

Policy gradient methods use similar approach to stochastic gradient ascent, but with the average reward objective and the policy parameters θ. We want to maximize the average reward rather than minimizing it. This means we do gradient ascent and move in the direction of the positive gradient. Remember that a simple recipe for solving a problem is to first specify an objective then estimate the gradient of that objective, and finally, adjust the weights in that direction. Step one is done. Now, the next step is to estimate the gradient of our objective Fig. 58.

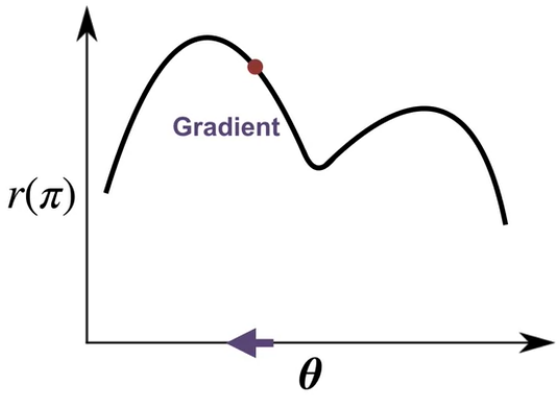

Figure 58: Gradient ascent

Recall the average reward objective, let's compute the gradient. We can apply the product rule of calculus to the objective to yield two terms.

Average reward objective

$$\nabla r(\pi) = \nabla \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r$$

Product rule

$$\nabla(f(x)g(x)) = \nabla f(x) g(x) + f(x) \nabla g(x)$$

Therefore:

$$\nabla r(\pi) = \sum_s \nabla \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r + \sum_s \mu(s) \nabla \sum_a \pi(a|s,\theta) \sum_{s',r} p(s',r|s,a) r$$

The first term $\nabla \mu(s)$involves the gradient of the stationary distribution overstates. Unfortunately, $\nabla \mu(s)$ is not straightforward to estimate. The stationary distribution $\mu(s)$ depends on a long-term interaction between the policy and the environment.

Luckily, the policy gradient theorem gives us a simpler expression for this gradient. The results of the theorem is shown next. In the inner sum, we have the gradient of the policy times the action value function $\nabla \pi(a|s,\theta) q_\pi(s,a)$.

$$\nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla \pi(a|s,\theta) q_\pi(s,a)$$

The gradient of the policy $\nabla \pi(a|s,\theta) q_\pi(s,a)$ is easy to compute as long as our policy parameterization is differentiable. The gradient of the policy tells you how to adjust your parameters to increase the probability of a certain action.

The gradient of the policy tells us how to change the policy parameters to make an action more likely to be selected in the given state. By moving the parameters in the direction of the gradient, we increase the probability for a particular action action. This necessarily means decrease in the probability of some of the other actions. **Different actions have different gradients.**

The whole term mentioned before gives the direction to move the policy parameters to most rapidly increase the overall average reward.

Estimating the Policy Gradient

To approximate the gradient we can do the same thing we did when deriving our stochastic gradient descent rule for policy evaluation. We simply make updates from states we observe while following policy π. This gradient from state s_t provides an approximation to the gradient of the average reward.

Recall:

$$\nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla \pi(a|s,\theta) q_\pi(s,a)$$

So:

$$\sum_a \nabla \pi(a|s_t,\theta_t) q_\pi(s_t,a)$$

As we discussed before for stochastic gradient descent, we can **adjust the weights with this approximation and still guarantee you will reach a stationary point**.

The stochastic gradient descent update looks like this for the policy parameters.

$$\theta_{t+1} = \theta_t + \alpha \sum_a \nabla \pi(a|s_t,\theta_t) q_\pi(s_t,a)$$

We can simplify the last equation further. Let's re-examine this from a perspective based on expectations. Notice that the sum over states waited by μ can be re-written as an expectation under μ. Recall that μ is

the stationary distribution for π which reflects state visitation under π. In fact, the state's we observe while following π are distributed according to μ.

$$\nabla r(\pi) = \mathbb{E}_\mu[\sum_a \nabla \pi(a|s,\theta) q_\pi(s,a)]$$

By computing the gradient from a state s_t, we get an unbiased estimate of this expectation. Thinking about our stochastic gradient as an unbiased estimate suggests one other simplification. Notice that inside the expectation we have a sum over all actions. We want to make this term even simpler and get rid of the sum over all actions. If this was an expectation over actions, we could get a stochastic example of this two and so avoid summing over all actions.

It would be nice if the sum of our actions was weighted by π and so was an expectation under π. That way we could sample it using the agents action selection, which is distributed according to π. It turns out this is an easy problem to solve. To get a weighted sum corresponding to an expectation we can just multiply and divided by $\pi(a|s)$. Now we have an expectation of over actions drawn from π for this term.

$$= \sum_a \nabla \pi(a|s,\theta) q_\pi(s,a)]$$

$$= \sum_a \pi(a|s,\theta) \frac{1}{\pi(a|s,\theta)} \nabla \pi(a|s,\theta) q_\pi(s,a)]$$

$$= \mathbb{E}_\pi[\frac{\nabla \pi(a|s,\theta)}{\pi(a|s,\theta)} q_\pi(s,a)]$$

The new stochastic gradient ascent update now looks like this.

$$\theta_{t+1} \doteq \theta_t + \alpha \frac{\nabla \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)} q_\pi(s_t,a_t)$$

As an aside it is common to rewrite this gradient as the grading of $\ln \pi$.

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(a_t|s_t,\theta_t) q_\pi(s_t,a_t)$$

This is based on a formula from calculus for the gradient of a logarithm.

$$\nabla \ln(f(x)) = \frac{\nabla f(x)}{f(x)}$$

The last thing to talk about is how to actually compute the stochastic gradient for a given state and action. We just need two components, the gradient of the policy and an estimate of the differential values. The first is easy. We know the policy and this parameterization, and so can compute its gradient. The action value can be approximated in a variety of ways. For example, we could use a TD algorithm that learns differential action values.

Actor-Critic Algorithm

Even within policy gradient methods, value-learning methods like TD still have an important role to play. In this setup, the parameterized policy plays the role of an actor, while the value function plays the role of a critic, evaluating the actions selected by the actor. These so-called actor-critic methods, were some of the earliest TD-based methods introduced in reinforcement learning.

Recall this expression for the policy gradient learning rule for the episodic task.

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(a_t|s_t, \theta_t) q_\pi(s_t, a_t)$$

But we don't have access to q_π, so we'll have to approximate it. We can do the usual TD thing, the one-step, bootstrap return. As usual, the parameterize function $\hat{v}$ is learned estimate of the value function.

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(a_t|s_t, \theta_t)[R_{t+1} - \overline{R} + \hat{v}(S_{t+1}, w)]$$

This is the critic part of the actor-critic algorithm. The critic provides immediate feedback. To train the critic, we can use any state value learning algorithm. We will use the average reward version of semi-gradient TD. The parameterized policy is the actor. It uses the policy gradient updates shown below Fig. 59.

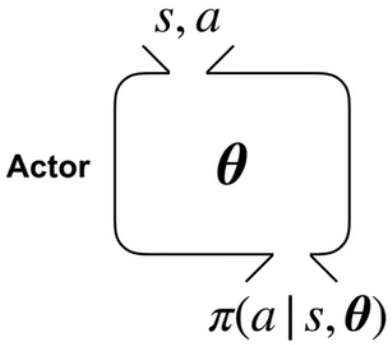

Figure 59: Policy gradient approximation

We could use the last update derived, but there is one last thing we can do to improve the algorithm. We can subtract off what is called a **baseline**. Instead of using the one-step value estimate alone, we can subtract the value estimate for the state, s_t, to get the update that looks like this.

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(a_t|s_t, \theta_t)[R_{t+1} - \overline{R} + \hat{v}(S_{t+1}, w) - \boldsymbol{\hat{v}(s_t, w)}]$$

$\hat{v}(s_t, w)$ is the baseline in this case. Notice that the following expression is equal to the TD error.

$$\delta = [R_{t+1} - \overline{R} + \hat{v}(S_{t+1}, w) - \hat{v}(s_t, w)] = \text{TD error}$$

The expected value of the last update is the same as the previous one defined above.

Subtracting the baseline tends to reduce the variance of the update which results in faster learning.

After we execute an action, we use the TD error to decide how good the action was compared to the average for that state. If the TD error is positive, then it means the selected action resulted in a higher value than expected. Taking that action more often should improve our policy. That is exactly what this update does. It changes the policy parameters to increase the probability of actions that were better than expected according to the critic. Correspondingly, if the critic is disappointed and the TD error is negative, then the probability of the action is decreased. The actor and the critic learn at the same time, constantly interacting. The actor is continually changing the policy to exceed the critics expectation, and the critic is constantly updating its value function to evaluate the actors changing policy Fig. 60.

To start with the algorithm, we specify the policy parameterization and the value function parameterization. For example, we might use tile coding to construct the approximate value function and a softmax policy parameterization. We will need to maintain an estimate of the average reward just like we did in the

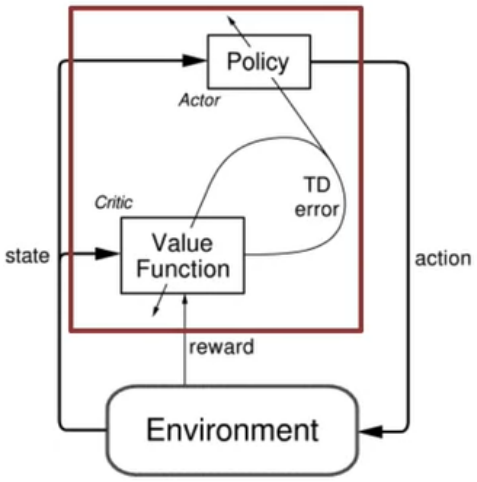

Figure 60: Actor and Critic interaction

differential Sarsa algorithm. We initialize this to zero. We can initialize the weights and the policy parameters however we like. We initialize the step size parameters α for the value estimate, the policy, and the average reward, and they could all be different. We get the initial state from the environment and then begin acting and learning. On each time step, we choose the action according to our policy and receive the next state and reward from the environment. Using this information, we compute the differential TD error and updating our running estimate of the average reward. We update the value function weights using the TD update. Finally, we update the policy parameters using our policy gradient update Fig. 61.

Actor-Critic (continuing), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Initialize $\bar{R} \in \mathbb{R}$ to 0

Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g. to 0)

Algorithm parameters: $\alpha^{\mathbf{w}} > 0, \alpha^{\boldsymbol{\theta}} > 0, \alpha^{\bar{R}} > 0$

Initialize $S \in \mathcal{S}$

Loop forever (for each time step):

$A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$

Take action A, observe S', R

$\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$

$S \leftarrow S'$

Figure 61: Actor-Critic (Continuing) Algorithm

Actor-Critic with Softmax Policies

Actor-critic elegantly mixes direct policy optimization, value functions, and temporal difference learning. We didn't specify the function approximation nor the policy parameterization. We will discuss one specific implementation, actor critic with linear function approximation and a Softmax policy parameterization.

Previously, we introduced the actor critic algorithm. That algorithm combines policy evaluation, which is the critic, and the policy gradient rule to update the policy, which is the actor.

Critic

$$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w)$$

Actor

$$\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A|S, \theta)$$

The critic uses semi-gradient TD, the actor uses the TDR from the critic to update the policy parameters.

A common choice of policy parameterization for finite actions is the Softmax policy. We can use function approximation to represent the preferences $e^{h(s,a,\theta)}$. The parameters of this function are the policy parameters denoted by theta, so our policy gradient learning rule will update theta.

$$\pi(a|s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_{b \in A} e^{h(s,b,\theta)}}$$

Notice the policy is written as a function of the current state. This is like having a different Softmax distribution for each state.

The state-dependent action preferences give rise to potentially different distributions for each state. To select an action, we follow a simple procedure. In the current state we query the Softmax for each action. This generates a vector of probabilities, one entry for each action. Given this vector, we pick an action proportionally to these probabilities Fig. 62.

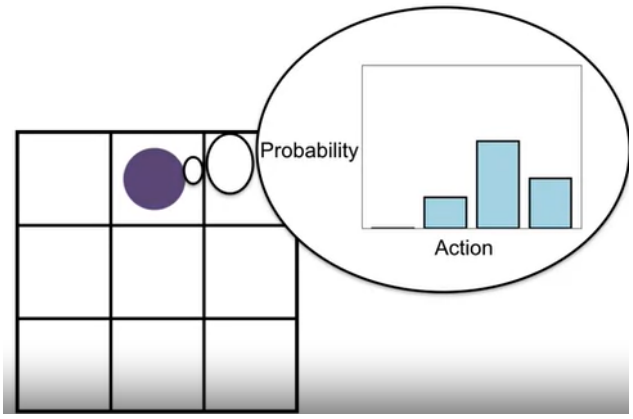

Figure 62: Softmax policy for a state

Let's discuss how our parameterization choices impact the value function, the action preferences, and our update equations. Let's use the same features for our value estimates and the policy. Remember the critic updates an estimate of the state value function, so the critic only requires a feature Vector characterizing the current state.

$$\hat{v}(s, w) \doteq W^T x(s)$$

The actor's action preferences depend on the state and action, so our action preference function requires a state action feature vector. To solve this issue, we can use stack state feature as shown in Fig. 44.

$$h(s, a, \theta) \doteq \theta^T x_h(s, a)$$

The size of the policy parameter vector, θ, is larger than the weights, W, used by the critic. Since there are three actions, this is three times larger.

Now let's look at the update equations for the actor and the critic with our linear Softmax parameterization. The first one is easy. The gradient of the linear value function is just the feature vector.

$$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w)$$

$$\nabla \hat{v}(S, w) = x(s)$$

Therefore the critic is:

$$w \leftarrow w + \alpha^w \delta x(s)$$

The actor's update to the preferences looks as follows.

$$\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A|S, \theta)$$

$$\nabla \ln \pi(A|S, \theta) = x_h(s, a) - \sum_b \pi(b|s, \theta) x_h(s, b)$$

The gradient has two parts. The first is the state action features for the selected action $x_h(s, a)$. The second part is the state action features multiplied by the policy summed over all actions $\sum_b \pi(b|s, \theta) x_h(s, b)$.

Gaussian Policies for Continuous Actions

A Gaussian random variable x has the probability density function shown in the next equation.

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The image Fig. 63 shows examples of PDF with different means and variances. Note that the y-axis in this case is density and not probability. Probability density means that for a given range, the probability of x lying in that range will be the area under the probability density curve. The parameter μ controls the mean of the distribution. If $\mu = 0$, the distribution will be centered around 0. For this curve, $\mu = -2$ so the distribution is centered around -2. The parameter σ is the standard deviation, the square root of the variance. If σ is low like this curve, the distribution will be sharply peaked around the mean value μ. As σ becomes larger, the distribution is more spread out like this curve, and x is drawn from a wider range of values.

To define our policy using a Gaussian over actions. We make the parameters μ and σ functions of the state. This way, the agent can assign different action distributions to different states.

$$\pi(a|s, \theta) \doteq \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} e^{-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}}$$

μ can be any parameterized function. But to keep things simple, let's make it a linear function of the state features. The policy parameters associated with μ are denoted by θ_μ^T

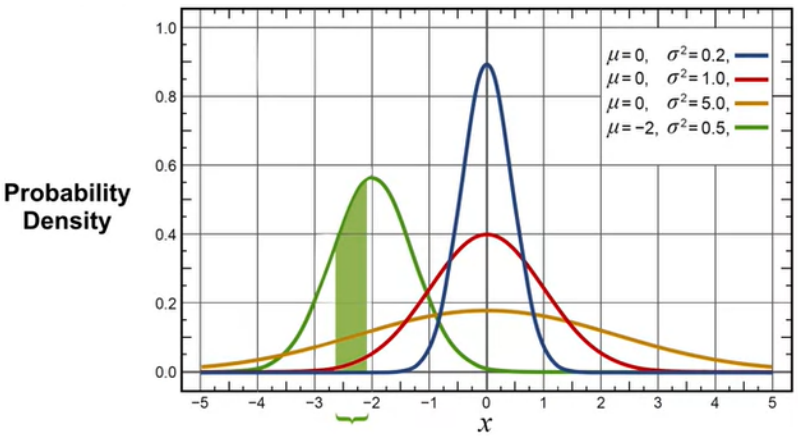

Figure 63: Gaussian distribution

$$\mu(s, \theta) \doteq \theta_\mu^T x(s)$$

The parameter's function σ has one constraint. It must be positive. To enforce this, we will parameterize it as the exponential of a linear function. The policy parameters associated with σ are denoted by θ_σ^T

$$\sigma(s, \theta) \doteq exp(\theta_\sigma^T x(s))$$

The policy parameters now consists of these two stack parameter vectors of equal size.

$$\theta \doteq \begin{bmatrix} \theta_\mu \\ \theta_\sigma \end{bmatrix}$$

Sigma essentially controls the degree of exploration.

We typically initialize the variance to be large so that a wide range of actions are tried. As learning progresses, we expect the variance to shrink and the policy to concentrate around the best action in each state Fig. 64. Like many parameterized policies, the agent can reduce the amount of exploration over time through learning.

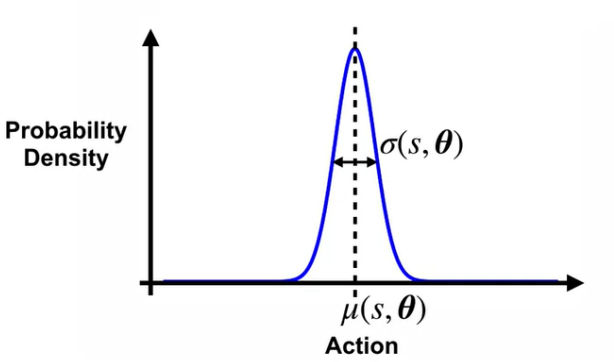

Figure 64: Gausian policies in action

Now we have continuous actions instead of discrete actions, we can use the same actor-critic architecture. The algorithm is actually very similar. The main difference is that the gradient of the policy is different because the parameterization is different.

$$\nabla \ln \pi(a|s, \theta_\mu) = \frac{1}{\sigma(s,\theta)^2}(a - \mu(s,\theta))x(s)$$

$$\nabla \ln \pi(a|s, \theta_\sigma) = (\frac{(a - \mu(s,\theta))^2}{\sigma(s,\theta)^2} - 1)x(s)$$

Adventage of continuous actions

- It might not be straightforward to choose a discrete set of actions.
- Continuous actions also allow us to generalize over actions.
- If the true action set is discrete but very large, it might be better to treat them as a continuous range.

References

[1] University of Alberta. *Prediction and Control with Function Approximation*. 2019. URL: `https://www.coursera.org/learn/prediction-control-function-approximation` (visited on 02/29/2020).

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.