

## Nomenclature

Nomenclature in RL can be inconsistent. Modern literature has largely settled on nomenclature as given below. Historically some policy gradient methods would use  $u$  for action - this is also common in *optimal control* (a related field - see the notes for Section 2).

Following [Thomas & Okal \(2016\) A Notation for Markov Decision Processes](#)

symbol	variable
$s$	state
$s'$	next state
$a$	action
$r$	reward
$G_t$	discounted return after time t
$\gamma$	discount factor $[0, 1)$
$a \sim \pi(s)$	sampling action from a stochastic policy
$a = \pi(s)$	deterministic policy
$\pi^*$	optimal policy
$V \pi(s)$	value function
$Q \pi(s, a)$	value function
$\theta, \omega$	function parameters (weights)
$\mathbf{E}[f(x)]$	expectation of $f(x)$

## Expectations = weighted average of all possible values (the mean)

`expected_value = probability * magnitude`

$$\mathbf{E}[f(x)] = \sum p(x) \cdot f(x)$$

Expectations allow us to **approximate by sampling**

Approximating the expectation for our commute to work can be done by sampling the time across three days and averaging.

Modern reinforcement learning optimizes expectations - expected future reward. RL is trying to take actions that **on average** are the best.

## Conditionals = probability of one thing given another

probability of next state and reward given state & action

$$P(s'|s, a)$$

reward received from a state & action

$$R(r|s, a, s')$$

sampling an action from a stochastic policy

$$a \sim \pi(s|a)$$

## Variance & bias

Model generalization error = bias + variance + noise

### In supervised learning

Variance = overfitting

- error from sensitivity to noise in data set
- seeing patterns that aren't there

Bias = underfitting

- error from assumptions in the learning algorithm
- missing relevant patterns

### In reinforcement learning

Variance = deviation from expected value

- how consistent is my model / sampling
- can often be dealt with by sampling more
- high variance = sample inefficient

Bias = expected deviation vs true value

- how close to the truth is my model
- approximations or bootstrapping tend to introduce bias
- biased away from an optimal agent / policy

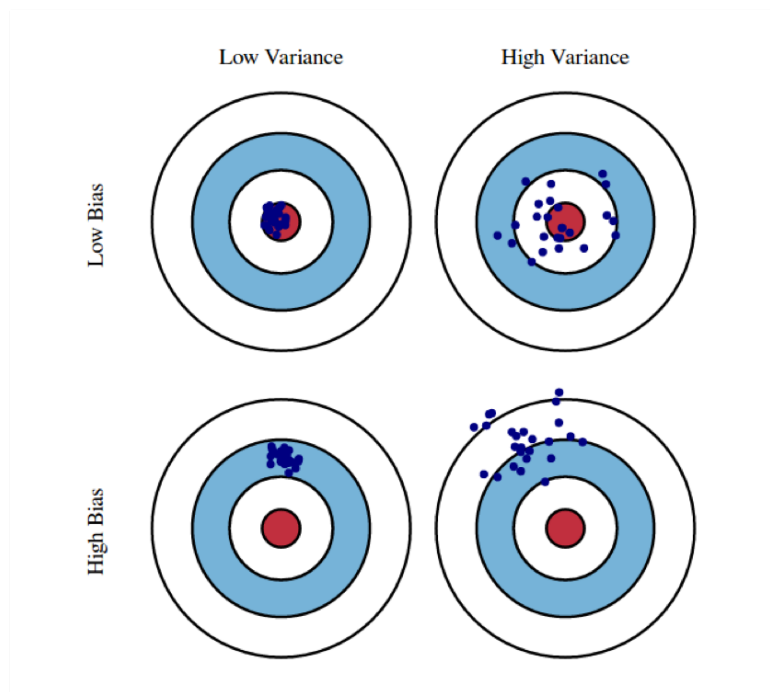


Figure 1: Variance = consistency, bias = error versus the truth

## Bootstrapping

Doing something on your own

- funding a startup with your own capital
- using a function to improve or approximate itself

The Bellman Equation is bootstrapped equation

$$V(s) = r + \gamma V(s')$$
$$Q(s, a) = r + \gamma Q(s', a')$$

Bootstrapping often introduces bias. The bootstrapped approximation gives the agent a chance to mislead itself.

## iid - independent and identically distributed

Fundamental assumption made in statistical learning

Assuming the training set is independently drawn from a fixed distribution

In the context of image classification

- independent sampling = we have photos from a wide range of sources, and each photo is independent of our other photos
- fixed distribution = the photos we have in our training set are the same kinds of photos as we will do prediction for

## curse of dimensionality

Refers to phenomena that occur in high dimensional spaces

High dimensional spaces means we need more data to support these dimensions

Often we need to consider every possible combination of actions - a high dimensional action space (say a robot with multiple arms) means we need to consider a large number of potential actions - this is known as the **combinatorial explosion**

Each additional dimension doubles the effort to consider all of the combinations

Rule of thumb - 5 training examples for each dimension in the representation

## importance sampling

[Wikipedia](#)

Estimate the properties of a distribution we can't sample from (the unknown distribution) using samples from another distribution

Sampling from one distribution allows to lower the variance of our estimate of the expectation of our unknown distribution

Trying to approximate the expected value of a random variable  $X$  under a distribution  $P$  -  $\mathbf{E}[X; P]$

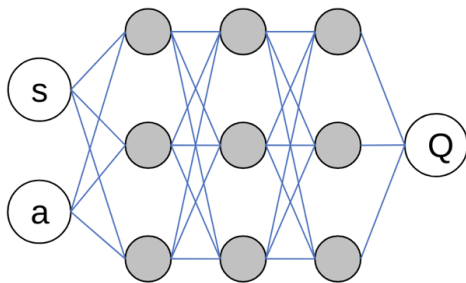
## Function approximation

state	temperature	pressure	estimate
	s1	s2	
0	100	100	-1
1	100	90	1
2	90	100	0
3	90	90	1

Lookup table

$$V_{\pi}(s) = 3s_1 + 4s_2$$

Linear function



Non-linear function  
(neural network)

Figure 2: Three commonly use function approximation methods

### Lookup tables

Imagine a problem where we have two dimensions in the state variable, with each state variable having two discrete options (either high or low). We use this state variable to predict the safety of the system.

```
state = np.array([temperature, pressure])
```

state	temperature	pressure	safety estimate
0	high	high	unsafe
1	low	high	safe
2	high	low	safe
3	low	low	very safe

### Advantages

- Stability
- Each estimate is independent of every other estimate

### Disadvantages

- No sharing of knowledge between similar states/actions
- Curse of dimensionality - high dimensional state and action spaces means large tables

## Linear functions

$$V(s) = 3s_1 + 4s_2$$

Advantages

- Less parameters than a table
- Can generalize across states

Disadvantages

- The real world is often non-linear

## Non-linear functions

Most commonly neural networks

Advantages

- Model complex dynamics
- Convolution for vision
- Recurrency for memory / temporal dependencies

Disadvantages

- Instability
- Difficult to train

## A few things about training neural networks

Larger batch size

- larger learning rate
- decrease in generalization
- increase in batch normalization performance

### Learning rate

Controls the strength of weight updates performed by the optimizer (SGD, RMSprop, ADAM etc)

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(x, \theta^t)}{\partial \theta}$$

where  $E(x, \theta^t)$  is the error backpropagated from sample  $x$

Small learning rate - slow training

High learning rate - overshoot or divergence

### Always intentionally set learning rate

```
from keras.models import Sequential
# don't do this!
model.compile(optimizer='rmsprop', loss='mse')
# do this
from keras.optimizers import RMSprop
opt = RMSprop(lr=0.001)
model.compile(optimizer=opt, loss='mse')
```

### Batch size

Modern reinforcement learning trains neural networks using batches of samples

Below we have a dataset with four samples, of shape (14, 2)

```
>>> import numpy as np
>>> data = np.arange(4*28).reshape(4, -1, 2)
>>> data.shape
(4, 14, 2)
```

The first dimension is the batch dimension - this is foundational in TensorFlow

```
tf.placeholder(shape=(None, 14, 2))
```

Passing in `None` allows us to use whatever batch size we want

## Why use batches

Batches allow us to learn faster - weights are updated more often during each epoch

It is better to take many small steps in the right direction than to make a great leap forward only to stumble backward - Chinese Proverb

Smaller batches can fit onto smaller GPUs - if a large sample dimension we can use less samples per batch

Batches give a less accurate estimate of the gradient - this noise can be useful to escape local minima

Larger batch size -> larger learning rate - more accurate estimation of the gradient (better distribution across batch) - we can take larger steps

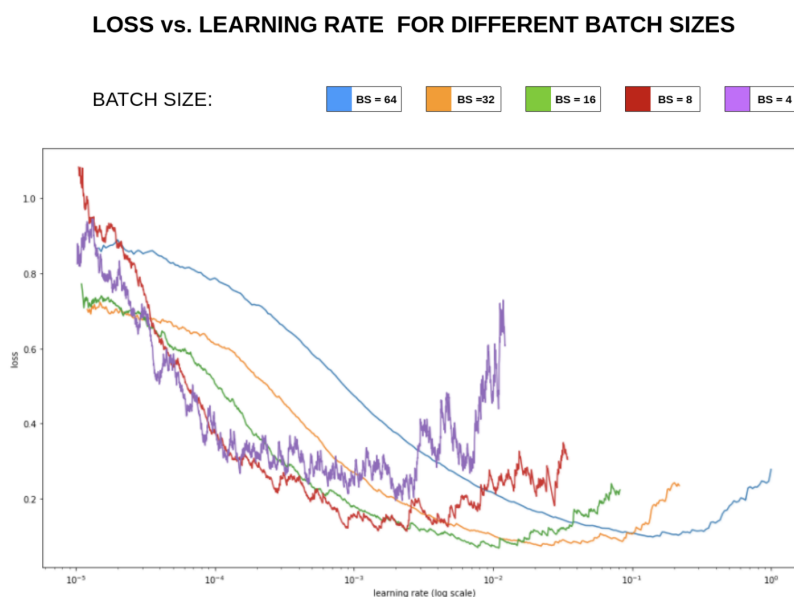


Figure 3: Relationship between learning rate error plotted using batches from 64 to 4

Larger batch size -> larger optimal learning rate

<https://miguel-data-sc.github.io/2017-11-05-first/>

Observed that larger batch sizes decrease generalization performance

Poor generalization due to large batches converging to *sharp minimizers* - areas with large positive eigenvalues  $\nabla^2 f(x)$  - Hessian matrix (matrix of second derivatives) where all eigenvalues positive = positive definite = local minima

Batch size is a **hyperparameter that should be tuned**

<https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network>

## Scaling aka pre-processing

Neural networks don't like numbers on different scales

- improperly scaled inputs or outputs can cause issues with gradients
- anything that touches a neural network needs to be within a reasonable range

We can estimate statistics like min/max/mean from the training set

- these statistics are as much a part of the ML model as weights
- in reinforcement learning we have no training set

**Standardization** = removing mean & scale by unit variance

$$\phi(x) = \frac{x - \mu(x)}{\sigma(x)}$$

Our data now has mean of 0, variance of 1

**Normalization** = min/max scaling

$$\phi(x) = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Our data is now between 0 and 1

## Batch normalization

[Ioffe & Szegedy \(2015\) Batch normalization](#)

[Ioffe \(2017\) Batch renormalization](#)

Batch norm. is additional preprocessing of data as it moves between network layers

- used in very deep convolutional/residual nets

We use the mean and variance of the batch to normalize activations

- standardization is actually used!
- reduces sensitivity to weight & bias initialization
- allows higher learning rates
- originally applied before the activation - but this is a topic of debate

[Batch normalization before or after relu - Reddit](#)

[Ian Goodfellow Lecture \(3:20 onward\)](#)

Vanilla batch norm. struggles with small or non-iid batches - the estimated statistics are worse

- vanilla batch norm. uses two different methods for normalization during training & testing
- batch renormalization uses a single algorithm for both training & testing