

# One - Statistical Background

Variance and bias, IID, function approximation.

---

## 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).

These notes follow [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)
$\mathbb{E}[f(x)]$	expectation of $f(x)$
$X$	a random variable
$f(x; \theta)$	density of $X$ at point $x$ , parameterized by $\theta$
$f(x, y)$	joint density of $X$ and $Y$ at the point $(x, y)$
$f(x y)$	conditional distribution of $X$ given $Y$

## Expectations

**An expectation is simply the mean** (or average for the less statistically careful).

expectation = probability \* magnitude

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

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

Note the difference in notation for paramertizations, joint densities and conditional distributions.

symbol	variable
$f(x; \theta)$	density of $X$ at point $x$ , parameterized by $\theta$
$f(x, y)$	joint density of $X$ and $Y$ at the point $(x, y)$
$f(x y)$	conditional distribution of $X$ given $Y$

## 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

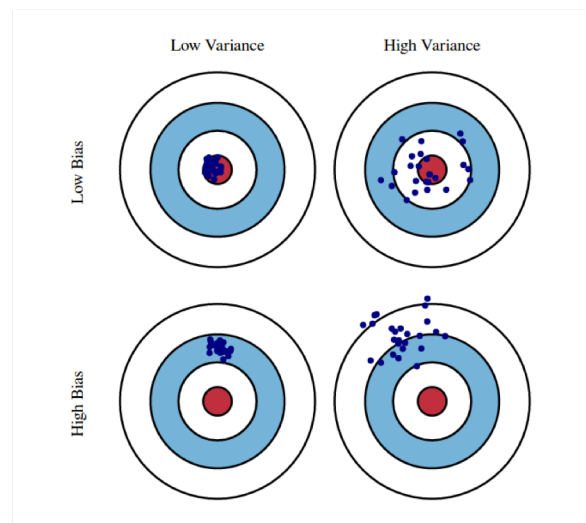


Figure 1: Variance = consistency, bias = error versus the truth

## 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. This is known as the **combinatorial explosion**

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

Rule of thumb - 5 training examples for each dimension in the representation (for supervised learning).

## Importance sampling

[Wikipedia](#) - [Introduction video on Youtube](#) -

*Reinforcement learning context - importance sampling is used in prioritized experience replay*

Importance sampling = a variant of Monte Carlo approximation - it's a method of approximating expectations.

If we had access to a probability distribution we can approximate the expectation analytically. We sum across all the possible values of the function  $f(x)$  and multiply by the probability of that value occurring:

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

If we don't have access to the distribution we could instead approximate the expectation using Monte Carlo, by looking at the sample mean across  $n$  samples. Here are sampling from the true distribution  $x \sim p$ . This means our sample expectation will converge to the true expectation of the distribution  $p(x)$ :

$$\mathbf{E}[f(x)] = \frac{1}{n} \sum_{i=1}^n f(x_i)$$

Now let's imagine that we could only take samples from a different distribution  $x \sim q$ . This will have a different expectation (because the probabilities of drawing a sample  $x$  are different):

$$\mathbf{E}[f(x)] = \sum_x q(x) \cdot f(x)$$

The magic of importance sampling is that we can use these samples  $q(x)$  to improve our approximation of the distribution we can't sample from  $p(x)$ . If we multiply our true distribution by the probabilities of our sample distribution:

$$\mathbf{E}[f(x)] = \sum_x p(x) \cdot f(x) \cdot \frac{q(x)}{q(x)}$$

We end up being able to take a sample expectation according to the sampling distribution  $q(x)$ :

$$\mathbf{E}[f(x)] = \frac{1}{n} \sum_{i=1}^n f(x_i) \frac{p(x_i)}{q(x_i)}$$

The ratio of the two probabilities is called the importance weight:

$$w(x) = \frac{p(x)}{q(x)}$$

$$\mathbf{E}[f(x)] = \frac{1}{n} \sum_{i=1}^n f(x_i) \cdot w(x_i)$$

Note that we do need to know the probabilities for both  $p(x_i)$  and  $q(x_i)$  - i.e. how likely the samples were. But we only need to know  $f(x)$  as sampled from the distribution  $q(x)$  - i.e. we don't need to know what the function was under the true distribution  $p(x)$ .

## Entropy

[Wikipedia - Must know Information Theory concepts in Deep Learning \(blog post\)](#)

*Context - entropy used for decision tree construction*

Like it's thermodynamic counterpart, entropy has a number of interpretations

- measure of uncertainty - more uncertainty is higher entropy
- measure of predictability - less predictable (more random) is higher entropy
- average information gain from observing an experiment - this is a function of the probability of an outcome. More information gained from rarer observations

Deterministic experiment (completely predictable, i.e. a coin with one side) has zero entropy.

Entropy is a measurement of how much information is contained in a distribution. Entropy gives us the theoretical lower bound on the number of bits we would need to encode our information. Knowing this lower bound allows us to quantify how much information is in our data.

Taking  $\log_2$  means we can interpret the entropy as measured in bits (i.e. 0 or 1)

$$H(x) = - \sum_i p(x_i) \cdot \log_2 p(x_i)$$

Some policy gradient based agents will have an entropy maximization term in the loss function - to make the policy as random as possible.

## Cross-entropy

[Wikipedia - Must know Information Theory concepts in Deep Learning \(blog post\)](#)

*Context - cross entropy is used as a loss function in classification neural networks - minimizing the distance between the output softmax and the true classes*

- measures the distance between or similarity of two probability distributions

For the discrete case the cross entropy is given by

$$H(p, q) = - \sum_x p(x) \cdot \log q(x)$$

## Mutual information

[Wikipedia - Must know Information Theory concepts in Deep Learning \(blog post\)](#)

- dependency between two distributions
- how much information about one variable is carried by another
- measures how much knowing one of these variables reduces uncertainty about the other
- Quantifies the bits (ie amount of information) obtained about one variable through the other

More generalized version of the linear correlation coefficient. Mutual dependence = 0 guarantees that random variables are independent, zero correlation doesn't.

$$I(X; Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \left( \frac{p(x, y)}{p(x)p(y)} \right)$$

Note that for continuous variables, the sums are replaced by integrals. If  $\log_2$  is used, the units are bits.

- for independent variables, the mutual information is zero.

- for deterministic functions, all information is shared -> mutual information is the entropy of Y (or X)

## Kullback–Leibler divergence

[Wikipedia - Must know Information Theory concepts in Deep Learning \(blog post\)](#)

*Reinforcement learning context - KL divergence is used in [Trust Region Policy Optimization \(TRPO\)](#) and [Proximal Policy Optimization \(PPO\)](#) to constrain how much a policy changes during learning. Also used in [C51 - A Distributional Perspective on Reinforcement Learning](#) uses the KL divergence, and suggests that the Wasserstein metric might be a fruitful next step.*

- measures the difference between probability distributions
- KL divergence between  $P$  and  $Q$  tells us how much information we lose when we try to approximate data given by  $P$  with  $Q$

$$D_{KL}(P||Q) = \mathbb{E}_x \cdot \log \frac{P(x)}{Q(x)}$$

Often used in reinforcement learning to measure/constrain/penalize the distance between policy updates. Also known as relative entropy or information gain.

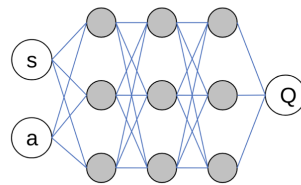
## Function approximation

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

It shows you understand the importance of the hyperparameter!

```
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

Tradeoff batch size vs. number of iterations to train a neural network

Visualizing Learning rate vs Batch size

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

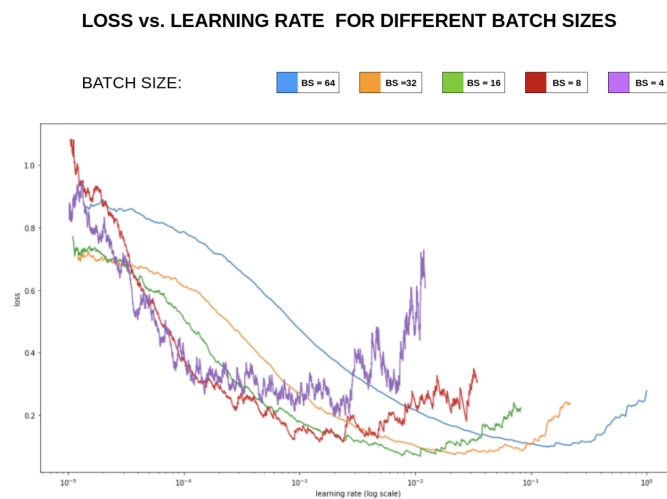


Figure 3: Relationship between learning rate error plotted using batches from 64 to 4 - [Visualizing Learning rate vs Batch size](#)

Larger batch size -> larger optimal learning rate

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**

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

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

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

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

## Residual networks

*Reinforcement learning context - AlphaGo Zero uses residual blocks in the value function and policy network*

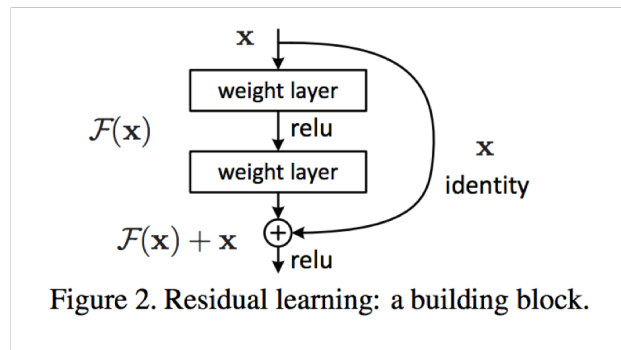


Figure 4: Residual blocks

[He et. al \(2015\) Deep Residual Learning for Image Recognition](#)

Convolutional network with skip connections

Layers are reformulated as residuals of the input

trying to learn  $H(x)$

instead of learning  $F(x) = H(x)$

we learn the residual  $F(x) = H(x) - x$

and can get  $H(x) = F(x) + x$