Introduction to Machine Learning

Prof. (Dr.) Honey Sharma

Reference Books:

- Mitchell M., T., Machine Learning, McGraw Hill (1997) 1st Edition.
- https://www.geeksforgeeks.org/

INTRODUCTION

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn-to improve automatically with experience-the impact would be dramatic.

Imagine computers learning from medical records which treatments are most effective for new diseases, houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants.

We do not yet know how to make computers learn nearly as well as people learn. However, algorithms have been invented that are effective for certain types of learning tasks.

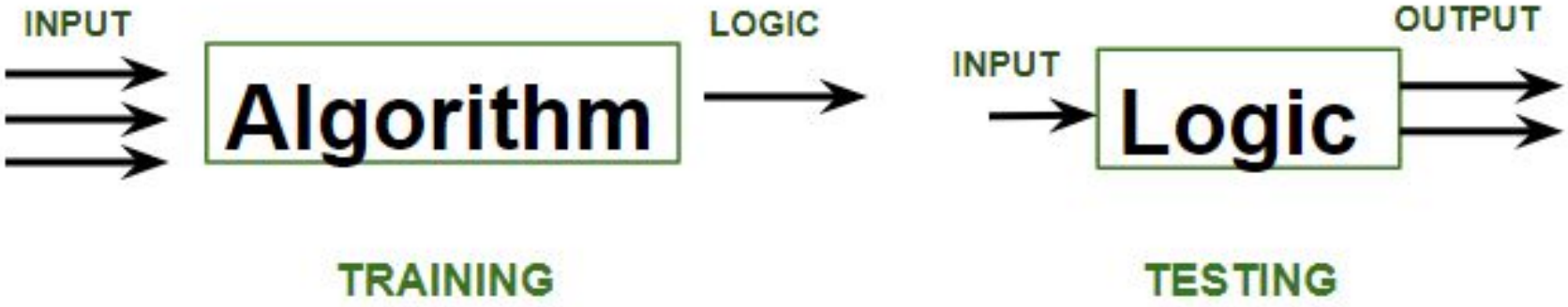
According to Arthur Samuel “Machine Learning enables a Machine to Automatically learn from Data, Improve performance from an Experience and predict things without explicitly programmed.”

In Simple Words, When we fed the Training Data to Machine Learning Algorithm,

This algorithm will produce a mathematical model

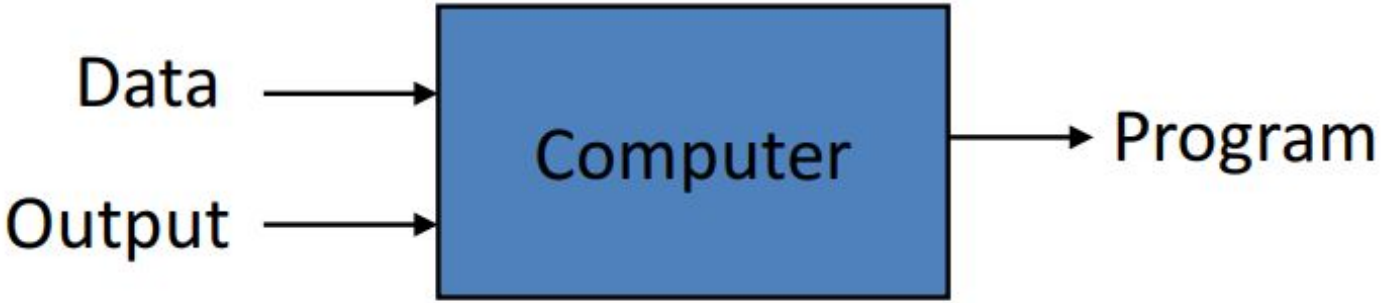
With the help of the mathematical model, the machine will make a prediction and take a decision without being explicitly programmed.

Also, during training data, the more machine will work with it the more it will get experience, the more efficient result will be produced.


INPUT
Algorithm
LOGIC
INPUT
Logic
OUTPUT
TRAINING
TESTING

Traditional Programming

Machine Learning

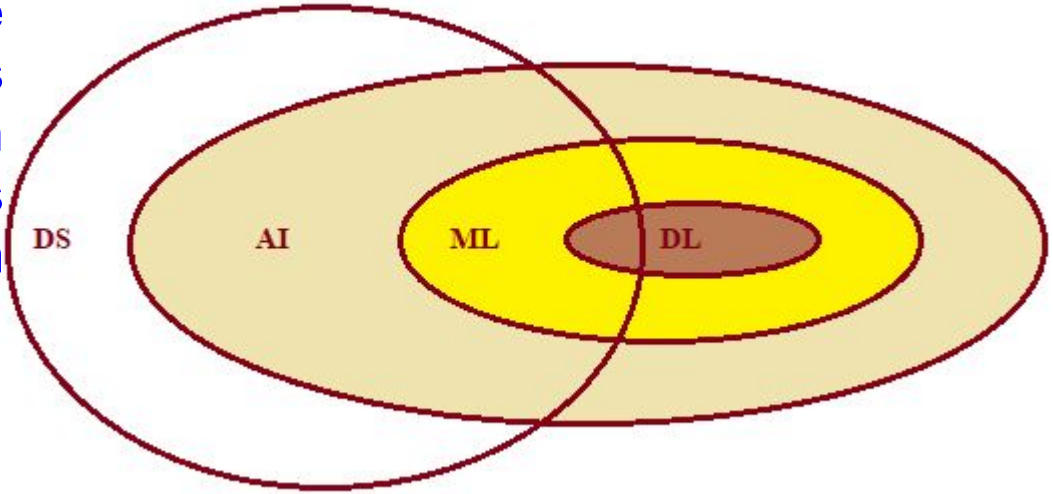


AI enables the machine to think, that is without any human intervention the machine will be able to take its own decision.

Machine Learning is a subset of AI that uses statistical to explore the data and build systems that have the ability to automatically learn and improve from experiences without being explicitly programmed.

Deep learning is a machine learning technique that is inspired by the way a human brain filters information, it is basically learning from examples.

DS uses tools like stat, prob, linear algebra



WELL-POSED LEARNING PROBLEMS

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

A Checkers Learning Problem

Task T: Playing checkers

Performance measure P: Percent of games won against opponents

Training experience E: Playing practice games against itself

A Robot Driving Learning Problem

Task T: Driving on public four-lane highways using vision sensors

Performance measure P: Average distance traveled before an error (as judged by human overseer)

Training experience E: A sequence of images and steering commands recorded while observing a human driver

A Handwriting Recognition Learning Problem

Task T: Recognizing and classifying handwritten words within images

Performance measure P: Percent of words correctly classified

Training experience E: A database of handwritten words with given classifications

DEFINE NEW LEARNING PROBLEMS

Task T:

Performance measure P:

Training experience E:

DESIGNING A LEARNING SYSTEM

- Choosing the Training Experience
- Choosing the Target Function
- Choosing a Representation for the Target Function
- Choosing a Function Approximation Algorithm
- The Final Design

Choosing the Training Experience

The very important and first task is to choose the training data or training experience which will be fed to the Machine Learning Algorithm.

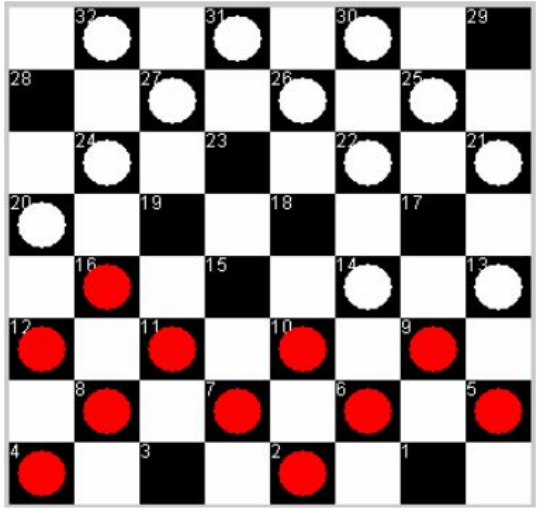
It is important to note that the data or experience that we fed to the algorithm must have a significant impact on the Success or Failure of the Model.

Attributes impacting on Success and Failure of Data

First key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system.

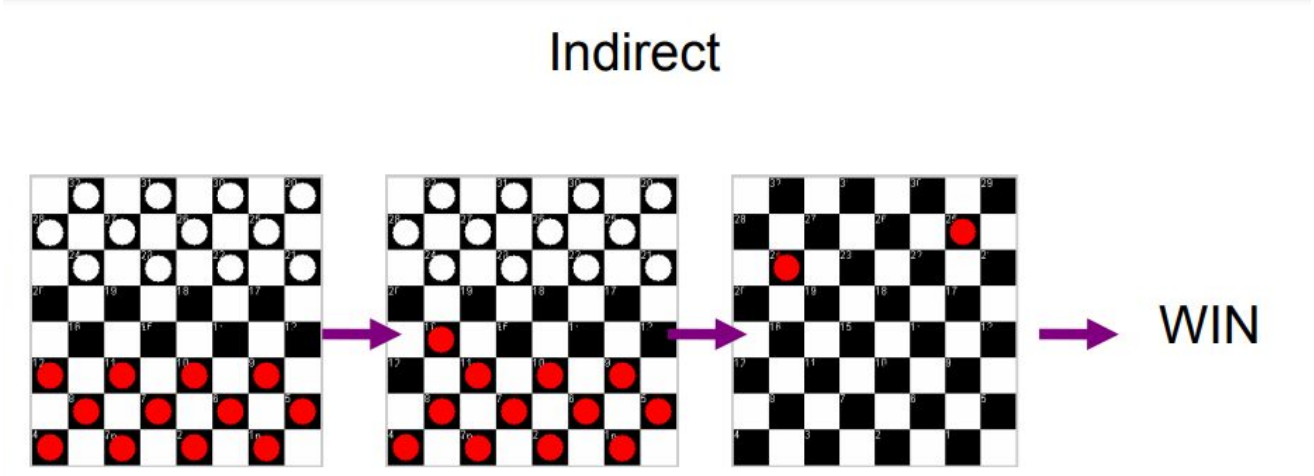
Examples:

Direct feedback: in learning to play checkers, the system might learn from examples consisting of individual checkers board states and the correct move for each.



16 → 19

Indirect feedback: Examples consisting of the move sequences and final outcomes of various games played.



In the last case, the learner faces an additional problem of credit assignment, or **determining the degree to which each move in the sequence deserves credit or blame for the final outcome**.

Credit assignment can be a particularly difficult problem because the **game can be lost even when early moves are optimal**, if these are followed later by poor moves.

Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

Second important attribute of the training experience is the degree to which the learner controls the sequence of training examples.

For example

The learner might rely on the teacher to select informative board states and to provide the correct move for each.

Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move.

The learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher

Third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured.

In general, learning is most reliable when the training examples follow a distribution similar to that of future test examples.

In our checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament.

If its training experience E consists only of games played against itself, there is an obvious danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested.

To proceed with our design, let us decide that our system will train by playing games against itself. This has the advantage that no external trainer need be present, and it therefore allows the system to generate as much training data as time permits.

We now have a fully specified learning task. A checkers learning problem:

Task T: playing checkers

Performance measure P: percent of games won in the world tournament

Training experience E: games played against itself

In order to complete the design of the learning system, we must now choose

- the exact type of knowledge to be learned
- a representation for this target knowledge
- a learning mechanism

Choosing the Target Function

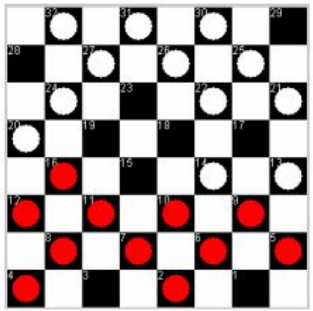
The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program.

A checkers-playing program that can generate the legal moves from any board state. The program needs only to **learn how to choose the best move from among these legal moves**. This learning task is representative of a large class of tasks for which the legal moves that define some large search space are known a priori.

Given this setting where we must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is **a program, or function, that chooses the best move for any given board state.**

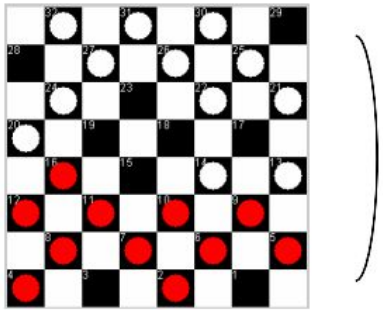
Let us call this function ChooseMove and use the notation ChooseMove : B -> M to indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M.

this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system

ChooseMove () = 16 → 19

An alternative target function is an evaluation function that assigns a numerical score to any given board state.

V : B -> R

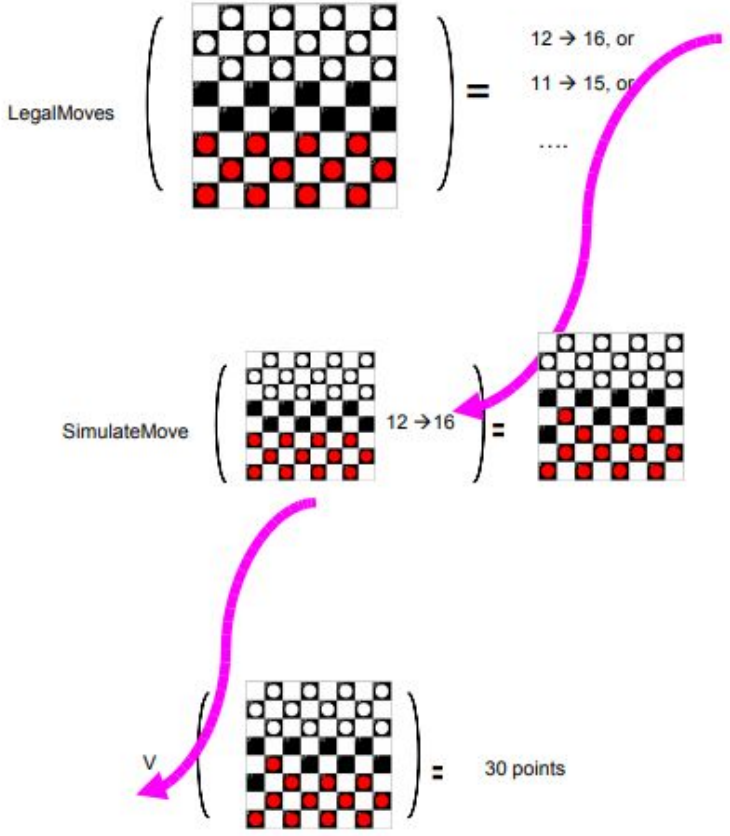
V  = 30 points

For all legal moves:

- simulate that move

- check the value of the result

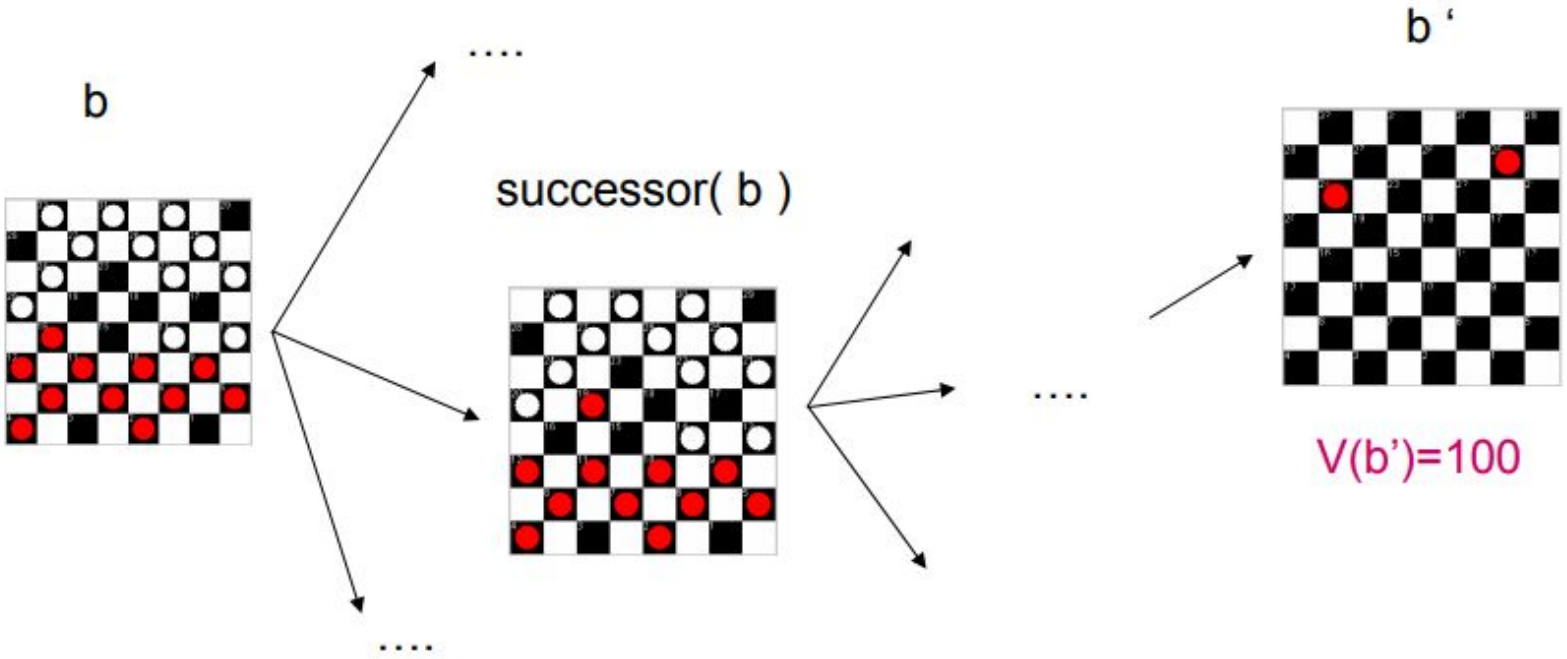
Pick the best move



What values should the target function, V, produce?

Of course any evaluation function that assigns higher scores to better board states will do.

1. if b is a final board state that is won, then V(b) = 100
2. if b is a final board state that is lost, then V(b) = -100
3. if b is a final board state that is drawn, then V(b) = 0
4. if b is a not a final state in the game, then V(b) = V(b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game


b
....
successor(b)
....
b ‘
V(b’)=100

It may be very difficult in general to learn such an operational form of V perfectly.

In fact, we often expect learning algorithms to acquire only some approximation to the target function, and for this reason the process of learning the target function is often called function approximation.

In the current discussion we will use the symbol $\tilde{V}$ to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

Choosing a Representation for the Target Function

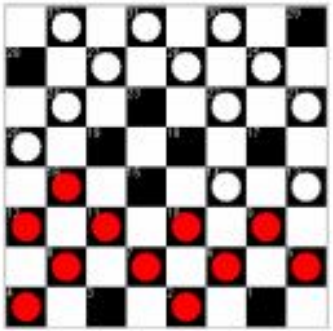
When the machine algorithm will know all the possible legal moves the next step is to choose the optimized move using any representation.

In general, this choice of representation involves a crucial tradeoff. On one hand, we wish to pick a very expressive representation to allow representing as close an approximation as possible to the ideal target function V.

On the other hand, the more expressive the representation, the more training data the program will require.

To keep the discussion brief, let us choose a simple representation:

for any given board state, the function $\tilde{V}$ will be calculated as a linear combination of the following board features:
x1: the number of white pieces on the board;
x2: the number of red pieces on the board;
x3: the number of white kings on the board;
x4: the number of red kings on the board;
x5:the number of white pieces threatened by red (i.e., which can be captured on red's next turn;
x6: the number of red pieces threatened by white



$X_1 = 12$ $X_3 = 0$ $X_5 = 1$

$X_2 = 11$ $X_4 = 0$ $X_6 = 0$

$$\tilde{\mathrm{V}}(b) = w_0 + \sum_{i=1}^{6} w_i x_i$$

here w0 through W6 are numerical coefficients, or weights, to be chosen by the learning algorithm.

Learned values for the weights w0 through W6 will determine the relative importance of the various board features in determining the value of the board, whereas the weight w0 will provide an additive constant to the board value.

Partial design of a checkers learning program:

Task T: playing checkers
Performance measure P: percent of games won in the world tournament
Training experience E: games played against itself
Target function: V:Board ->R
Target function: representation $\tilde{V}$

The first three items above correspond to the specification of the learning task, whereas t**he final two items constitute design choices for the implementation of the learning program**

Notice the net effect of this set of design choices is to reduce the problem of learning a checkers strategy to the problem of learning values for the coefficients w0 through w6 in the target function representation.

Choosing a Function Approximation Algorithm

In order to learn the target function $\tilde{V}$, we require a set of training examples, each describing

- a specific board state b
- the training value Vtrain(b) for b.

In other words, each training example is an ordered pair of the form (b, Vtrain(b)).

For instance, the following training example describes a board state b in which White has won the game (note x2 = 0 indicates that red has no remaining pieces) and for which the target function value Vtrain(b) is therefore +100.

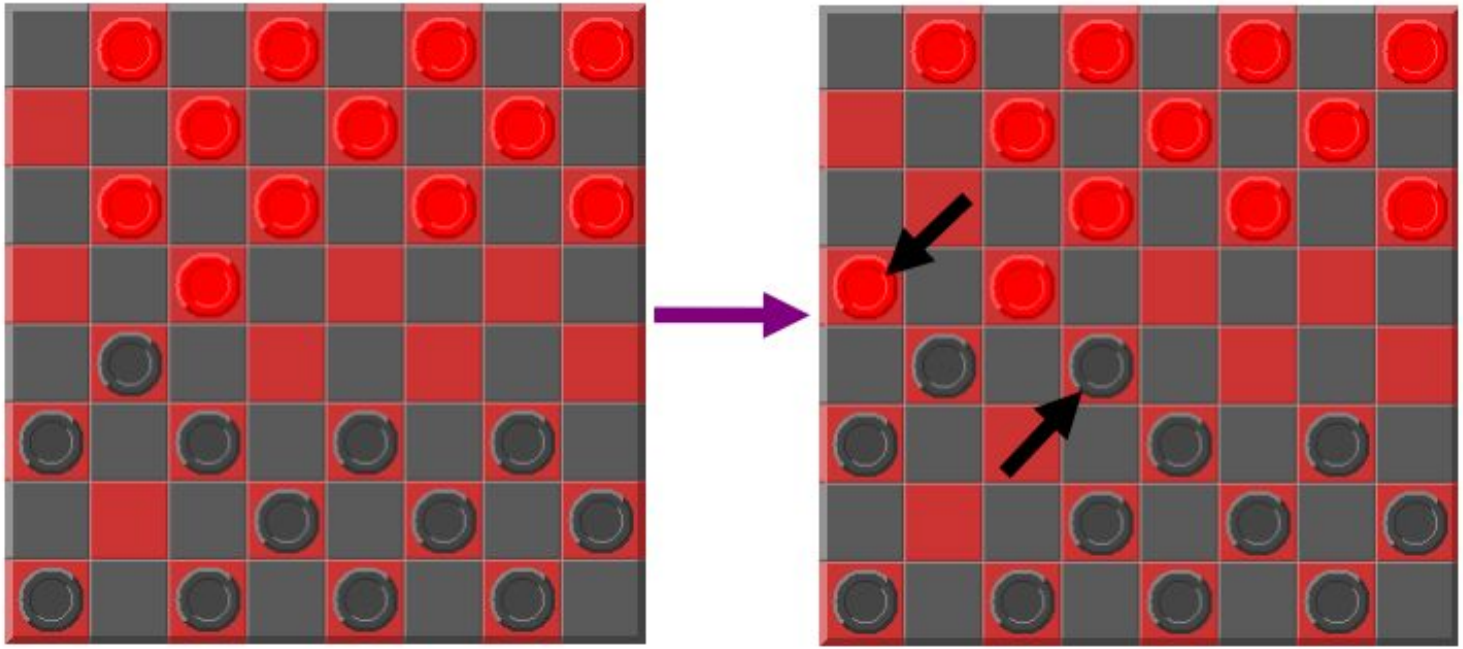
$$((x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, , x_5 = 0, , x_6 = 0), +100),$$

What about intermediate board states?

$$V_{train}(b) < -\tilde{V}(Successor(b))$$

where $\tilde{V}$ is the learner's current approximation to V and where Successor(b) denotes the next board state following b for which it is again the program's turn to move (i.e. the board state following the program's move and the opponents response)

- *b:*
- *Successor(b)*:



Now, we have the training data

All that remains is to specify the learning algorithm for choosing the weights to best fit the set of training examples

Common approach: best set of weights will minimise the squared error, E, between training values Vtrain and value predicted by approximation function $\tilde{V}$

$$E \equiv \sum_{\langle b, V_{train}(b) \rangle \in \text{training examples}} (V_{train}(b) - \hat{V}(b))^2$$

Thus, we seek the weights, or equivalently the $\tilde{V}$, that minimize E for the observed training examples.

We need algorithm that will:

- incrementally refine weights as more training examples become available
- Needs to be robust to errors in training data

One such algorithm is called the least mean squares, or LMS training rule.

LMS weight update rule.

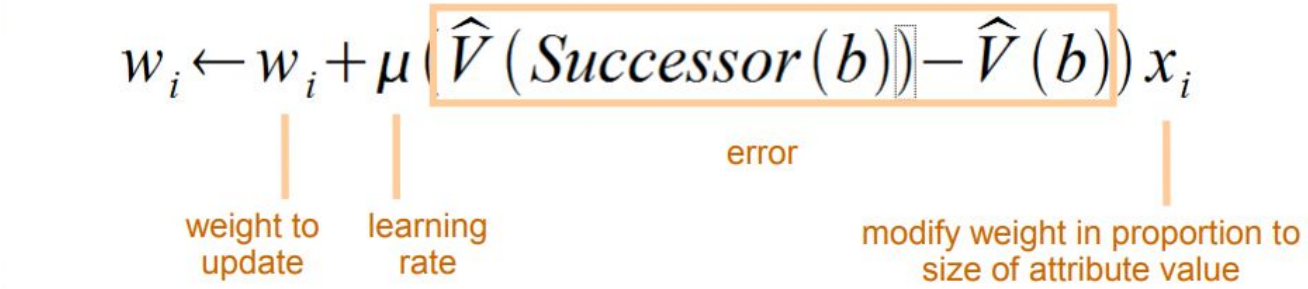
For each training example (b, Vtrain(b))

- Use the current weights to calculate $\tilde{V}(b)$
- For each weight wi, update it as

$$w_i \leftarrow w_i + \mu (V_{train}(b) - \hat{V}(b)) x_i$$

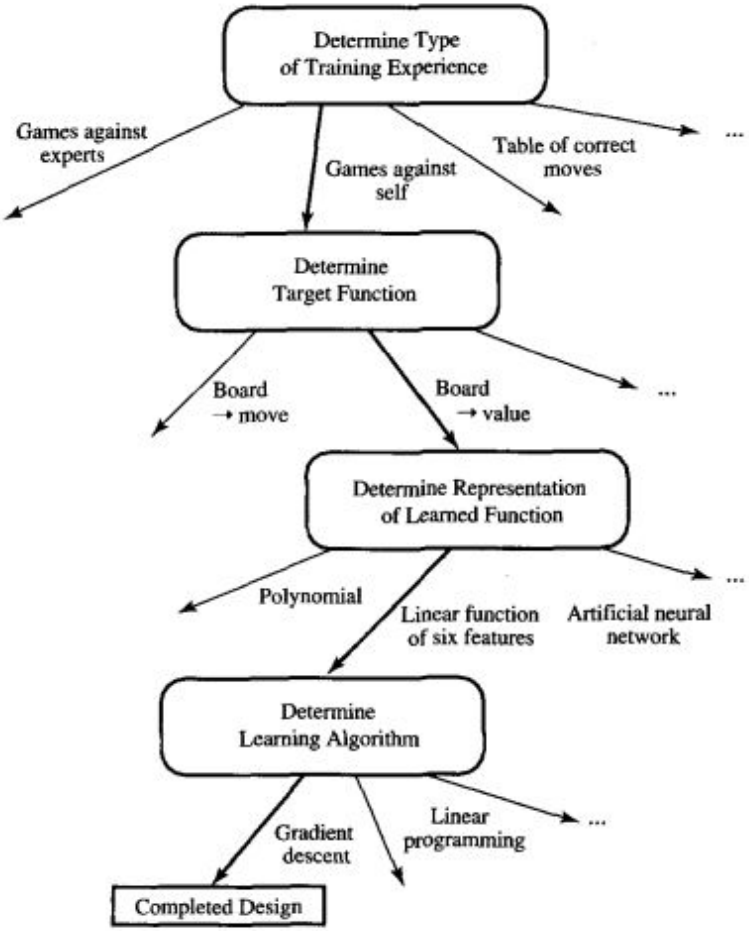
where,

$$V_{train}(b) \leftarrow \hat{V}(Successor(b))$$



- Here learning rate is a small constant (e.g., 0.1) that moderates the size of the weight update.
- Notice that when the error (Vtrain(b) -Ṽ(b)) is zero, no weights are changed.
- When(Vtrain(b) -Ṽ(b))is positive (i.e., when Ṽ(b) is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of Ṽ(b), reducing the error.
- Notice that if the value of some feature xi is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board.

The Final Design



learning process – playing and learning at same time computer will play against itself. Initialise $\tilde{V}(b)$ with random weights (wi) (First time round is essentially random (because we set the weights as random) – as it learns should pick better successors)

start a new game - for each board

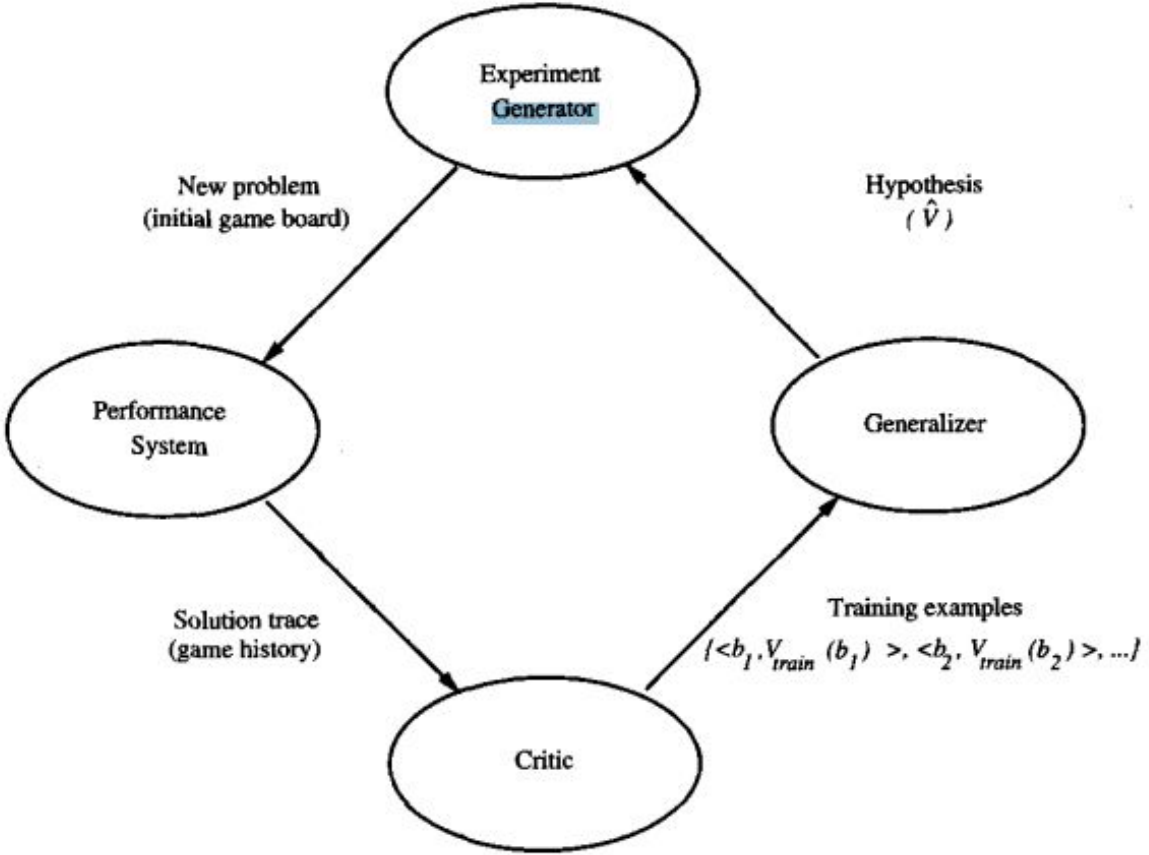
a) calculate $\tilde{V}(b)$ on all possible legal moves

(b) pick the successor with the highest score

(c) evaluate error

(d) modify each weight to correct error

The Final Design



The Performance System is the module that must solve the given performance task. It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.

In our case, the strategy used by the Performance System to select its next move at each step is determined by the $\tilde{V}$ learned evaluation function.

Therefore, we expect its performance to improve as this evaluation function becomes increasingly accurate.

The Critic takes as input the history or trace of the game and produces as output a set of training examples of the target function.

Each training example in this case corresponds to some game state in the trace, along with an estimate Vtrain, of the target function value. In our example, the Critic corresponds to the training rule

$$V_{train}(b) < -\tilde{\mathrm{V}}\left(Successor(b)\right)$$

The Generalizer takes as input the training examples and produces an output hypothesis that is its estimate of the target function.

In our example, the Generalizer corresponds to the LMS algorithm, and the output hypothesis is the function $\tilde{V}$ described by the learned weights w0, . . . , W6.

The Experiment Generator takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore.

Its role is to pick new practice problems that will maximize the learning rate of the overall system.

In our example, the Experiment Generator follows a very simple strategy: It always proposes the same initial game board to begin a new game.