Exam 2 Recap

📖 This is a highlight of important information found from chapter CSP to Machine Learning & Neural Networks.

Constraint Satisfaction Problems (CSP)

Sometimes it is easier to identify a goal state based on meeting a series of conditions.

> **Examples:**
>
> - ~~Student A graduates when all courses are approved.~~ Who cares about the order in which courses were approved?
> - In a Letter Soup, you want to find all words. The order in which you find it is not important.

State is defined by a set of variables. → X_1, X_2, ... , X_n

Each variable X_i has a domain D_i from which it can take on value.

A set of Constraints regulate the possible values for:

- Each X_i (unary)
- Combinations of X_i and X_j (binary)
- Multiple X_i, X_j, ... , X_k

The goal of the search is to find a state where each variable X_i has an assigned value. That all of the constraints are observed.

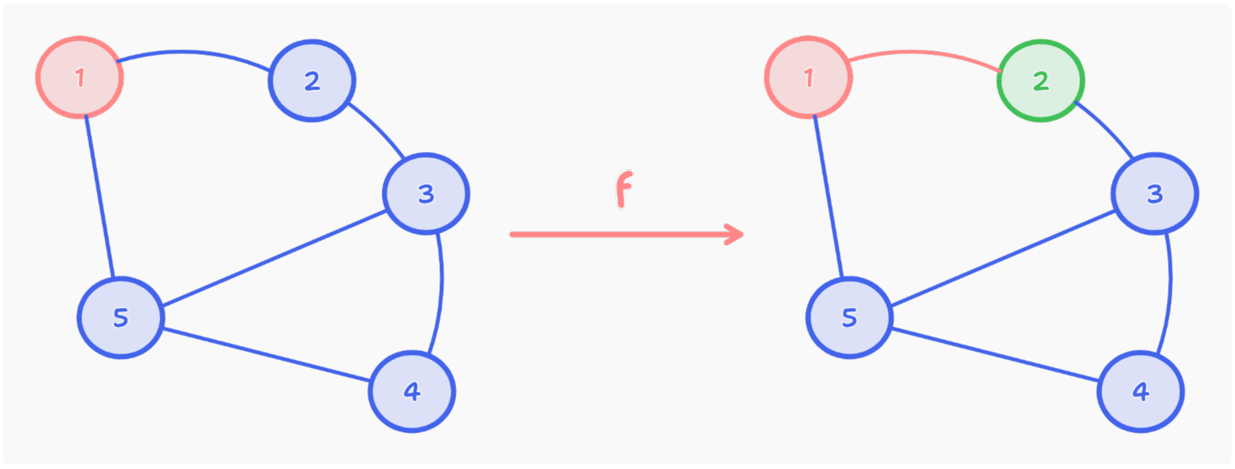
Solving CSP Problems

Previous search methods can be used.

Backtracking Search

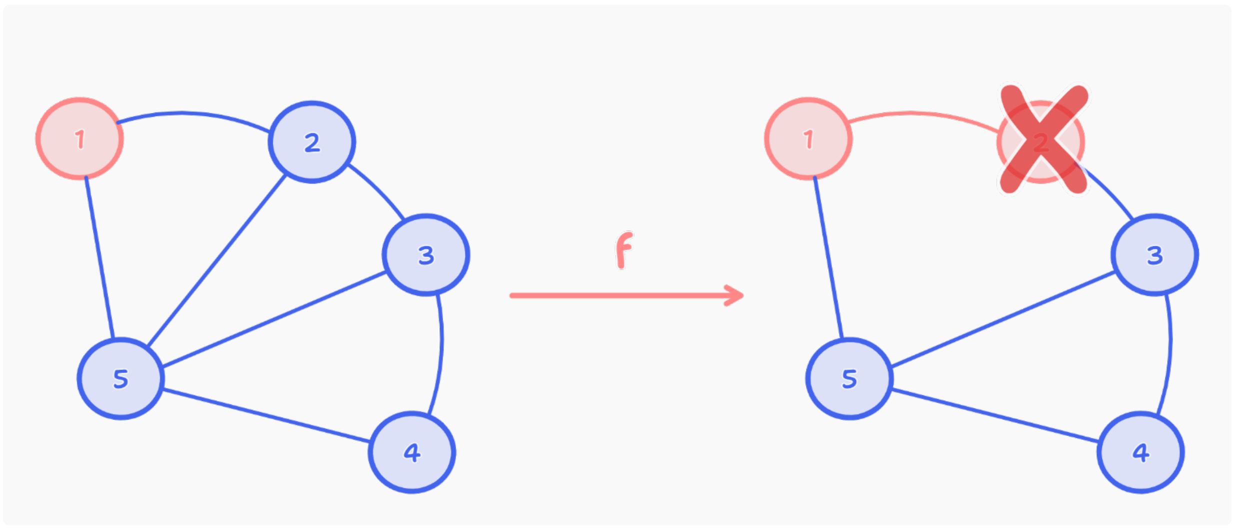
Depth First Search with two additional ideas:

1. Assign one variable at a time.



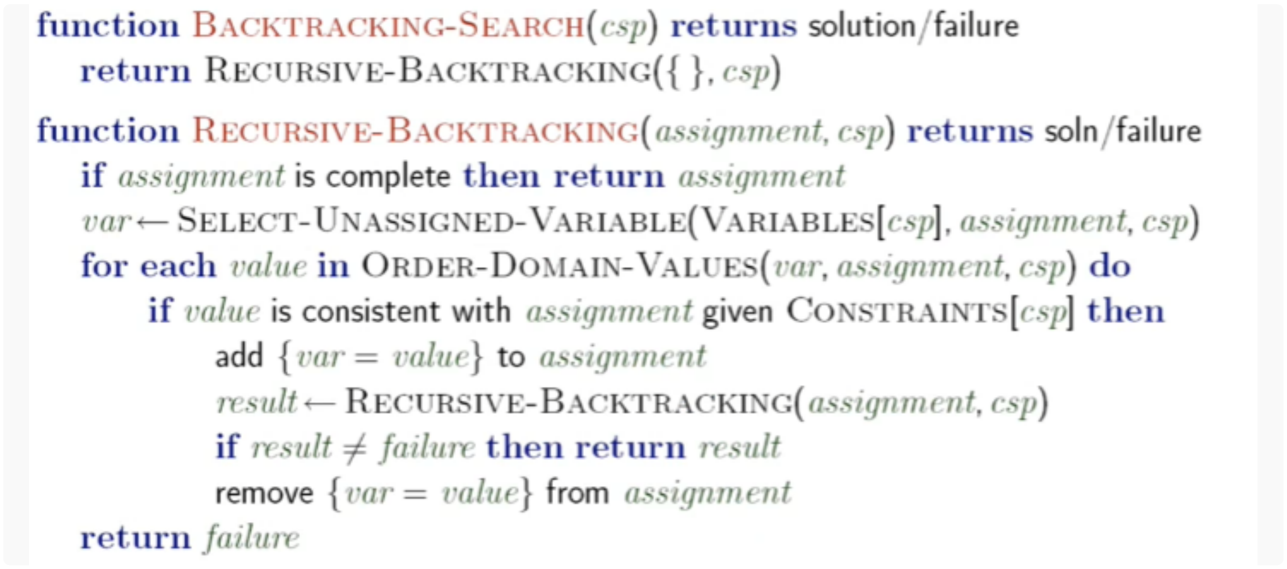
Successor function provides a new successor with one more assigned variable

2. Check constrains as you go. Immediately detect violation and stop search on that sub-tree.



. . .

Pseudo code of how it should work



```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Here is an implementation in Python 🐍

```python
def backtracking_search(csp):
    return recursive_backtracking({}, csp)

def recursive_backtracking(assignment, csp):
    if len(assignment) == len(csp['Variables']):
        return assignment

    var = select_unassigned_variable(csp['Variables'], assignment, csp)
    for value in order_domain_values(var, assignment, csp):
        if is_consistent(var, value, assignment, csp):
            assignment[var] = value
            result = recursive_backtracking(assignment, csp)
            if result is not None:
                return result
            del assignment[var]

    return None

def select_unassigned_variable(variables, assignment, csp):
    for var in variables:
        if var not in assignment:
            return var

def order_domain_values(var, assignment, csp):
    values = csp['Domain'][var]
    if 'Ordering' in csp:
        order_function = csp['Ordering']
        values = sorted(values, key=lambda value: order_function(var,
value, assignment, csp))
    return values

def is_consistent(var, value, assignment, csp):
    for constraint in csp['Constraints']:
        if var in constraint['Scope'] and not constraint['Function'](var,
value, assignment, csp):
            return False
    return True
```

To improve backtracking we could add filtering. (Constraint propagation inference)

- After each valid assignment we remove from domain D_i of variable X_i invalid options to reduce search space.

Markov Decision Process

So far agents can find paths to solve a problem and solutions that meet a set of constraints. But it requires a big assumption in all cases, that the world is deterministic.

Deterministic – predictable, without chance or randomness.

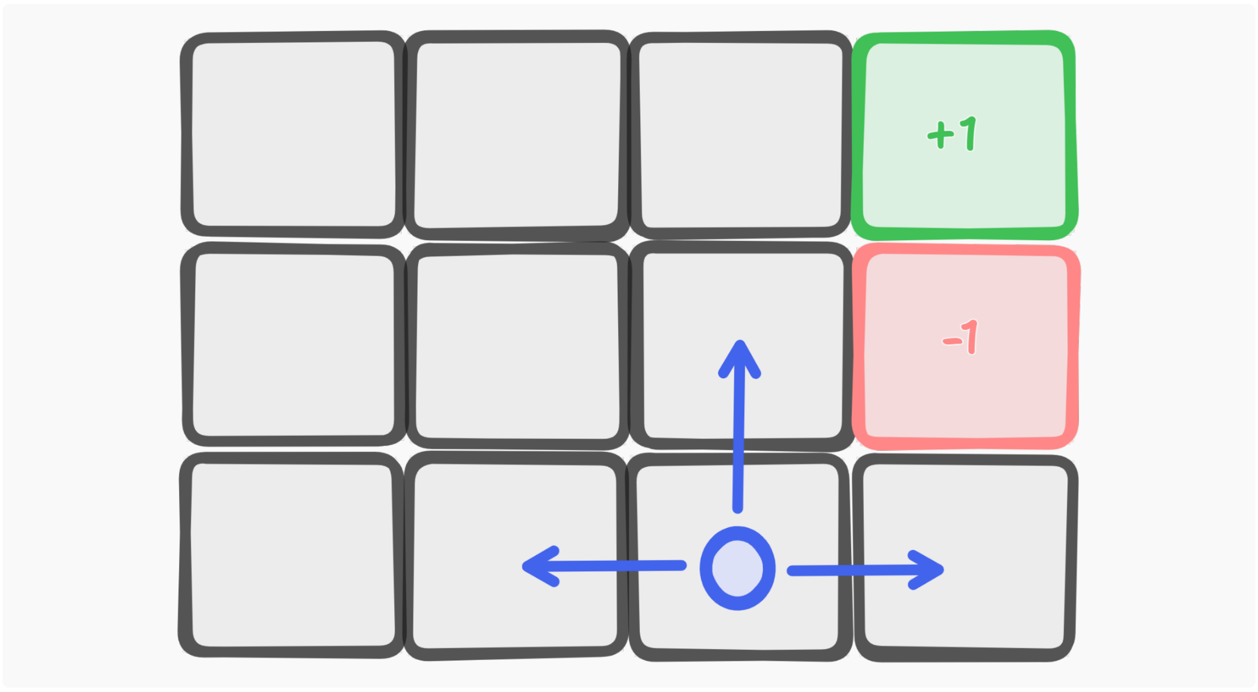
- The action of an agent occurs.
- No surprises

Now we look at the stochastic world.

Stochastic – Non-deterministic, unpredictable with the chance of being random.

- Actions of agents occur with a certain probability `P`.
- Surprises can occur.

Example: Grid World



- Agent moves in a grid world
- The agent's moves are not certain
- When an agent moves north it can move
  - **North** → 80% probability
  - **East** → 10% probability
  - **West** → 10% probability

> - Can be result of external factors.

This uncertainty of the world means that agents now need multiple plans of action. To solve this, agents need a policy. A recipe or standard operating procedure (SOP) telling what to do at each state.

Let's say the following:

- Set of state $s \in S$
- Set of actions $a \in A$
- Transition function $T(s, a, s)$

  $T(s, a, s) = P(s \mid s, a)$– probability of reaching state s given that action a was taken from state s (conditional probability).
- Reward function $R(s, a, s)$

  $R(s, a, s)$– Prize (positive or negative) for taking action $a$from state s to reach s.
- Start state s_0
- Terminal State s_f(could be more than one – positive or negative, or nonexistent).

The key idea in MDP is that given the present state s, the outcome of reaching a future state s does not depend on the past. This means that, it only depends on actions taken from s. The past is in the past.

$P(A \mid B) =$ Probability of event A happening given that B happened.

Policies in MDP

Policies is a function that maps each state to the action(s) that should be done.

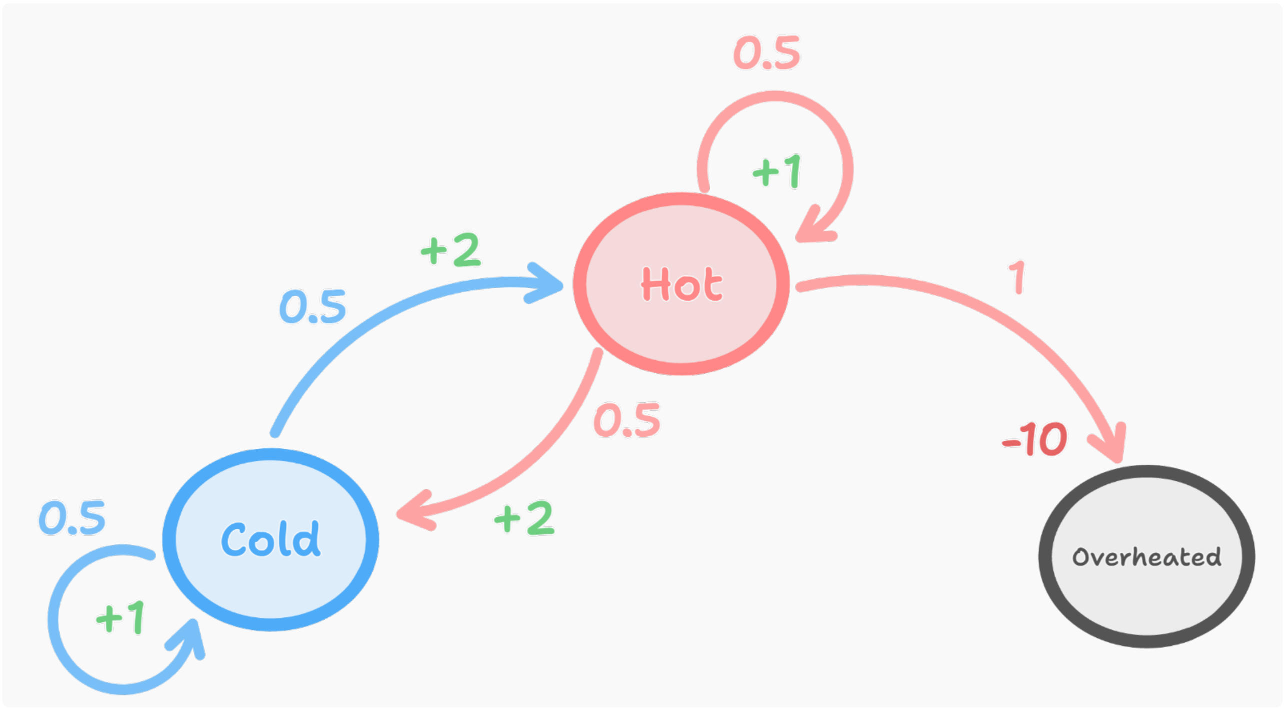
Denoted by the symbol pi π: S → A

Where the optimal policy for a given problem is denoted as pi start π* : S → A

Policies cover all bases: for each state tells the action that should be done. It is a map-type data structure where:

- Map: key, value pairs
- State is the key
- action is the value

To get the policy we use a strategy similar to expectimax.

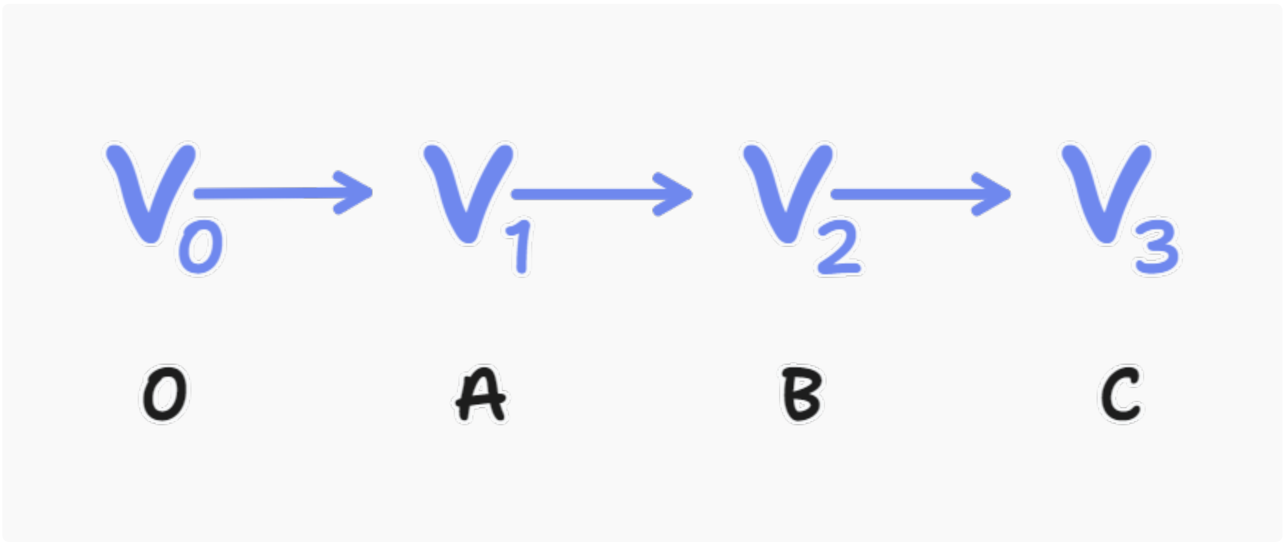


Say we have the following system where we know the probability of an action occurring and the associated reward from performing that action.

To calculate the value to take we perform the following formula:

$$Value_k = Probability \times [Reward_k + Decay \times Value_{k-1}]$$

We assume that our first value is 0.



We then calculate A, B, and C. To determine the best approach for a given value we take the maximum value found from all the options

> **Example:**

> Decay = 0.1, Going from Cold to Cold.
>
> $Value_1 = 0.5 \times [1 + 0.1 \times 0] = 0.5$
>
> Decay = 0.1, Going from Cold to Hot.
>
> $Value_1 = 0.5 \times [2 + 0.1 \times 0] = 1$
>
> This means that we would want to go to Hot if we are at the Cold state.

Machine Learning & Neural Networks

The computer science field devoted to developing algorithms that make computers learn without being explicitly programmed for it.

▼ **Key Ingredients**

- Pattern Recognition
- Statistical Learning
- Artificial Intelligence
- Linear Algebra

There are two broad specializations of Machine Learning:

1. **Supervised Learning –** Algorithm learns to make predictions from a set of examples.
2. **Unsupervised Learning –** Algorithm learns the structure of the data by itself.

Linear Regression

We fit a line $h(x) = \theta_0 + \theta_1 x$

$h(x)$ is called a hypothesis.

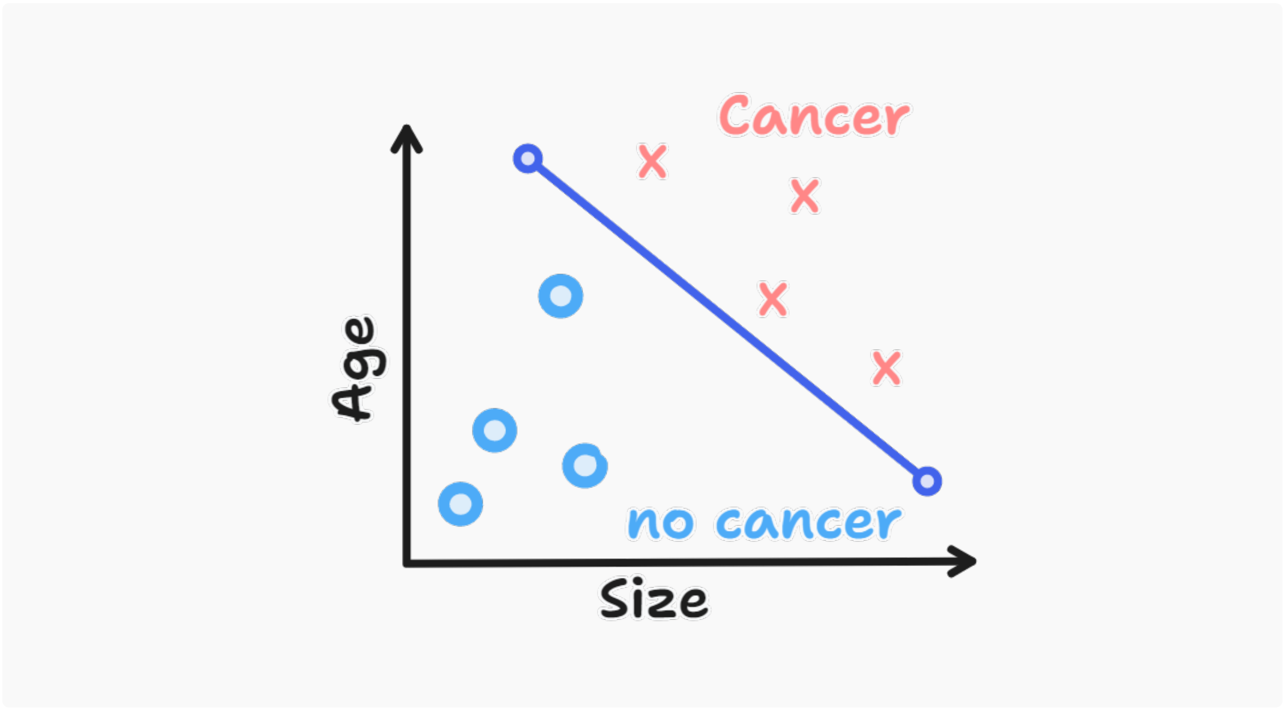
θ_0, θ_1 are called the parameters.

$h(x)$ is linear with respect to the parameters.

Linear regression approximates a continuous value.

Similar to $y = mx + b$

y = Hypothesis m = weight b = bias



The basic classifier has to make a `YES` or `NO` decision.

The linear part in linear regression is in the parameters (weights and biases).

Features can be combined in various ways.

> **Examples:**
>
> $$\hat{y} = w_1 x + w_2 x^2 + b$$
>
> $$\hat{y} = w_1 x_1 x_2 + b$$
>
> $$\hat{y} = w_1 x_1 + w_2 log(x_2) + w_3 x_3^4 + b$$

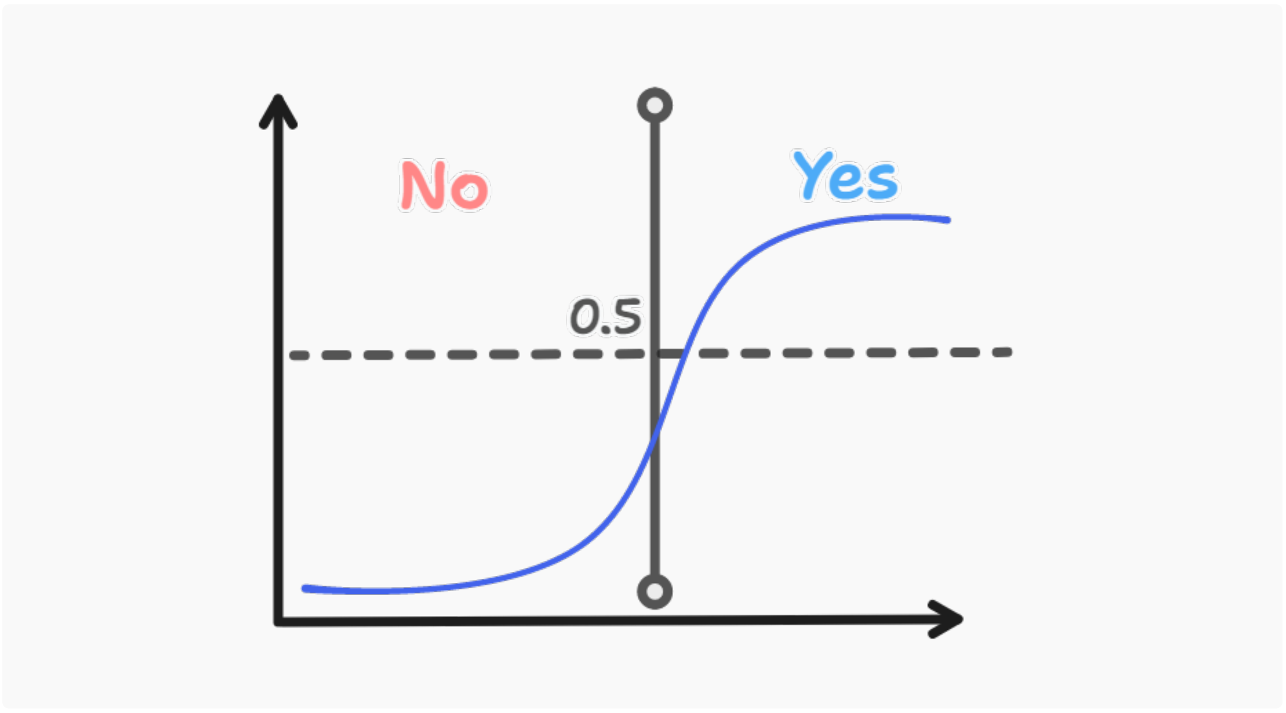
Logistic Regression

Classification method based on logistic function

$$f(z) = \frac{1}{1+e^{-z}}$$

Hypothesis gives probability that sample belongs to 1 class.

$f(z) = \frac{1}{1+e^{-z}}$ $\quad$ $f(z) \geq 0.5 \rightarrow 1$ $\quad$ $f(z) < 0.5 \rightarrow 0$



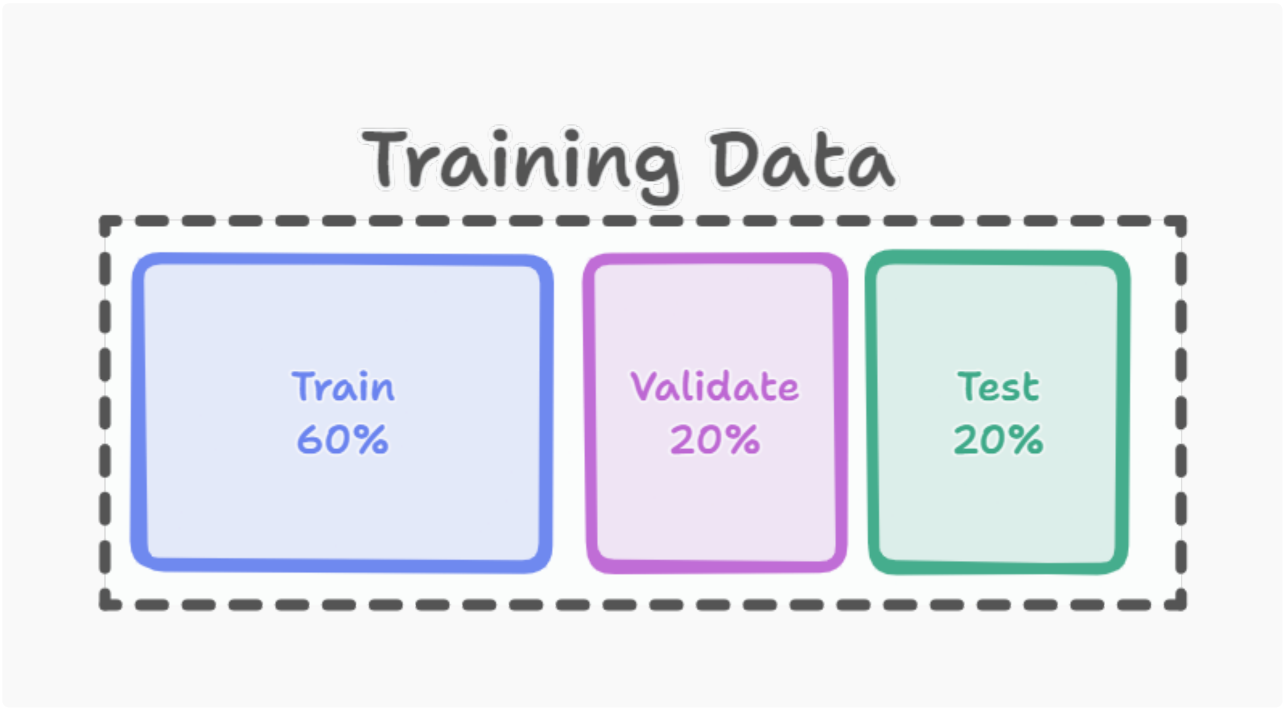
If you need to have more than 2 classes, then you need to run one vs all method.

Training

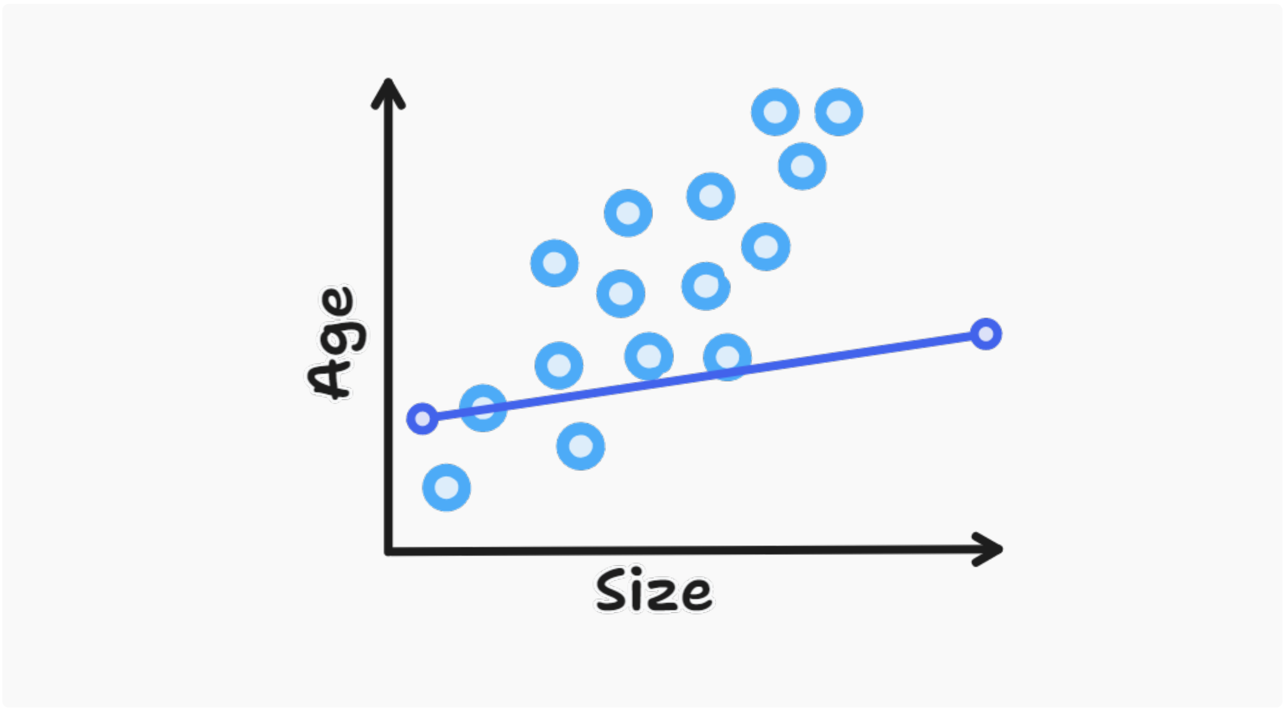
Training refers to the process of choosing weights, bias, and model.

Data is split into three portions:

- **Train –** Used to find parameters.
- **Validate –** Used check for bias/variance.
- **Test –** used to pick among good models.

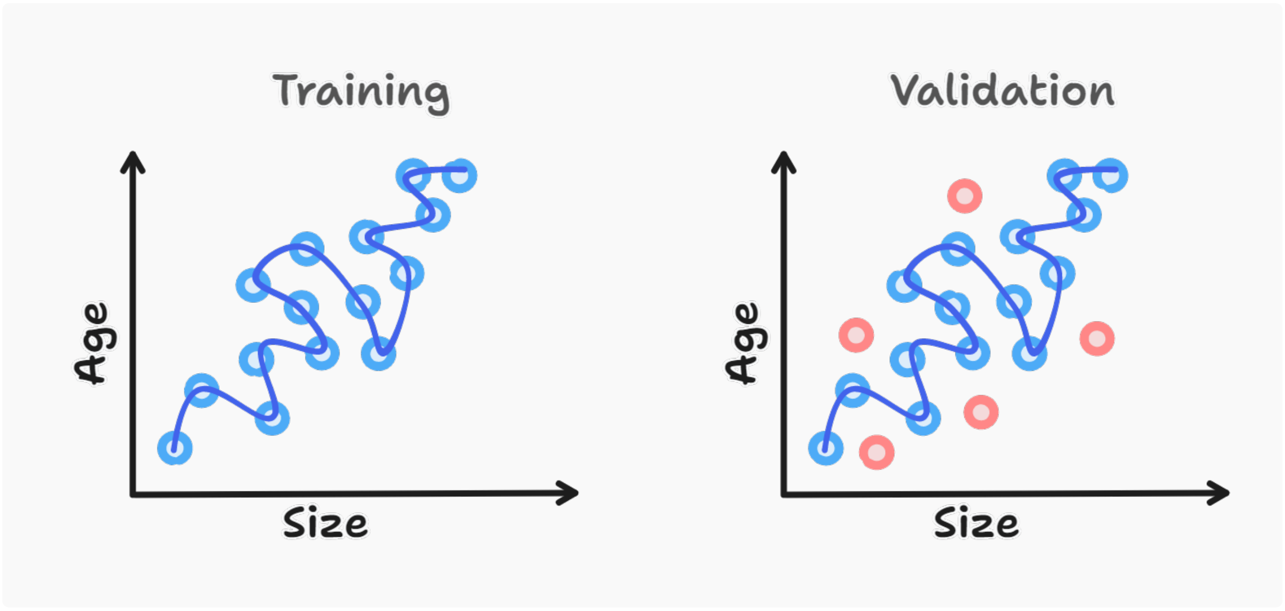


Bias underfitting occurs when a model with bias does not fit well with the training set. It misses on most of the examples.



To fix this you would need more features or another model.

Variance overfitting happens when a model with variance fits the training set but **fails** on the validation set. It does not **generalize** well. (It simply memorized the training set)



To solve this you can use more data. Additionally you can have regularization.

Regularization – Penalize the network whenever it makes a mistake.

Or you could do drop out. This means that the model randomly removes some weights. (Makes them 0). The idea is that the network will not depend on a given input. It should look at all inputs.

When choosing a model, you want to have low bias and low variance. If more than one exists, use the test set to decide.

Metrics and Cost Function

Bias, Variance, etc. are related with performance metrics of the model.

- **Squared Error –** Averaged squared error between prediction and real label.
- **Accuracy –** Averaged error between predicted class label and real label.

Cost function for a model measures the error in the target metric.

For regression:

$$(\widehat{y}^{(i)} - y^{(i)})^2$$

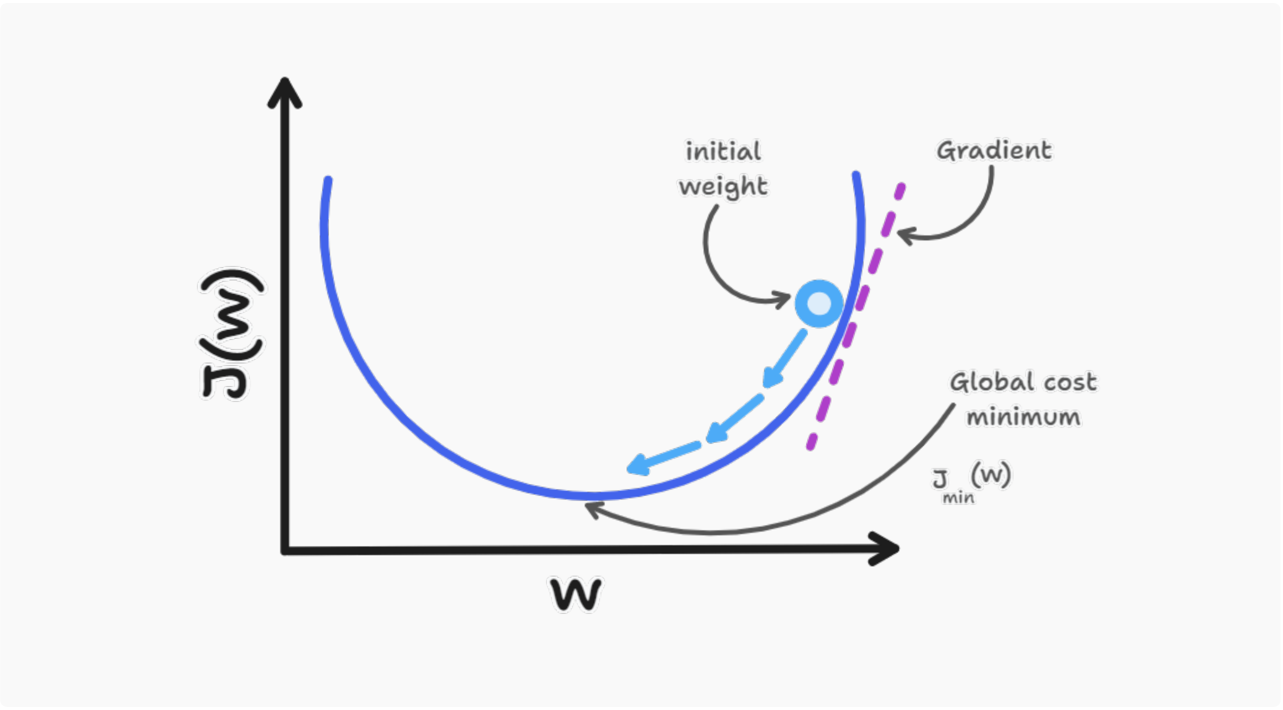
For entire data set:

$$J = \frac{1}{2n} \sum_{i=1}^{n} (\widehat{y}^{(i)} - y^{(i)})^2$$

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

Gradient Descent

```
w = 0
for each epoch e:
    for each example i:
        $w = w - \alpha \frac{dJ}{dw}$
```



α is called the learning rate. It controls how far we "jump". It is a number between 0 and 1.