**PERCEPTRON**

Binary classifier that takes set of input values , multiplies them by corresponding weights and calculates classification based on an activation function. Add bias based on sum you want. Types of activation functions are step, sigmoid, softmax, etc.

Remember that **they can’t learn XOR.**

**STEP FUNCTION**

Output based on if input is above/below a specified threshold.

**SIGMOID FUNCTION**

Squeezes the weighted sum from the perceptron into a value between 0 and 1.

**TANH FUNCTION**

Squeezing the weighted sum from the perceptron into a value between -1 and 1.

**RELU FUNCTION**

Output based on the function

**SOFTMAX FUNCTION**

Turns outputs into probabilities based on

We can use this to determine what class an output is.

**LOSS/COST FUNCTION**

Measures the difference between the output and the expected output and outputs a “cost” or “loss” based on

is also known as **cross-entropy loss**.

**GRADIENT DESCENT**

A process that updates the parameters (weights/biases) of the model in the direction of the *steepest descent* of the cost function.

Calculate partial derivative for slope and intercept from Sum of Squared Residuals (SSR), multiply each by learning rate to get step, do the step (change weights/parameters), get a new SSR, repeat. Stop when step < 0.01, or some other value .

**STOCHASTIC GRADIENT DESCENT**

Regular gradient descent but do it for one point at each step; reduces computation power required and reduces redundancy in calculating SSR when new data comes in. **Minibatch** SGC is just using a set of points, not one.

Where minibatch .

**BACKPROPAGATION**

Involves using the chain rule to calculate first the gradients of the loss function with respect to the output, then the intermediate outputs of each layer. We can use the chain rule here to compute the derivative of a composite function. We then use the gradient descent algorithm, which actually updates the weights/biases.

**GRADIENT EXPLODING/VANISHING**

**Exploding** happens when gradients become too large during backpropagation; causes parameter updates to overshoot optimal values which makes it hard for network to converge.

**Vanishing** happens when gradients become extremely small during BP; updates also become small and it takes forever for the network to learn. Happens when there’s a lot of layers.

**ERRORS**

**Training Error**: Model error on data

**Generalization Error**: Model error on new data

**Underfitting**: When low-capacity models can’t fit the training set (TS)

**Overfitting**: When high-capacity models memorize the TS

---------------------------------------------------------------

**CONVOLUTIONAL NEURAL NETWORKS**

Used to reduce input size before putting it in a conventional neural network. Pass input through multiple layers then return a column vector.

Input:

Kernel:

Output:

You can calculate CNN Layers with

Note that the \* is a convolutional operator **not ­­multiplication**

**POOLING LAYER**

These are the layers in CNNs. There are diff types.

**Max Pooling:** Returns max value from portion of image covered by kernel

**Average Pooling**: Returns avg value from ⬆️

**PADDING AND STRIDE**

Add rows/columns to input to keep edge information and allow for combining of diff filter sizes.

**Output shape** after convolution is

Where is row padding and is column padding.

**Padding**:

A picture containing screenshot, text, diagram, line

Description automatically generated

**Stride**:

A picture containing text, screenshot, diagram, rectangle

Description automatically generated

For multiple input channels:

A picture containing diagram, screenshot, rectangle, plan

Description automatically generated

Output shape is

In order: Input size, kernel/filter size, padding, stride

is height and is width.

---------------------------------------------------------------

**UNINFORMED SEARCH**

US means that we only know the **goal test** and the **succs()** function. Assuming succs() is a tree, we can use many diff strategies, including

**BFS:** Searches all states closest to tree and expands

**Uniform Cost Search**: Basically just Dijkstra’s

**DFS:** Select a node and keep moving deeper; move on to next branch, repeat

**Iterative Deepening:** DFS but with a limit based on first path found to goal state; you can’t go farther than this path length

A picture containing text, screenshot, font, number

Description automatically generated

**INFORMED SEARCH**

**A\* Search**

Put the start state on PQ; call the PQ open.

If OPEN is empty, exit with failure. #

Remove from OPEN and place on CLOSED a node for which is minimum. #

If is a goal node, exit. #

Expand , generating all successors and attach to pointers back to . For each successor of

If is not already on OPEN or CLOSED compute

and place it on OPEN. #

If is already on OPEN or CLOSED, then check if is lower for the new version of . If so, then:

Redirect pointers backwards from along path yielding lower . #

Put on OPEN. #

If is not lower for the new version, do nothing. #

Go to 2. #

---------------------------------------------------------------

**GAME THEORY**

We have an agent that interacts with the world

Agent → Action → World

World → Observations → Agent

**MODELING PROPERTIES**

**Number** **of Agents**: # of players

**Action Space**: Infinite is tennis, Finite is Rock-Paper-Scissors

**Deterministic/Stochastic**: Deterministic means that results of game are predictable, stochastic means that results can be random

**Zero-Sum/General Sum**: Zero-sum means one person wins, other loses

**Sequential/Simultaneous**: Simultaneous means all players take actions at the same time (RPS), sequential means players take turns.

**NORMAL FORM GAME**

Mathematical description of simultaneous game.

players, player chooses strategy from strategy set , player gets reward

Therefore, 2 players with 2 actions will lead to a 2x2 result matrix.

**Strictly Dominant Strategies** are strategies that always does better than other strategies/actions regardless of what other players do.

Sometimes this doesn’t exist. To determine the SDS, find

**Dominant Strategy Equilibrium** occurs when all players have a strictly dominant strategy, i.e., nobody does.

**Nash Equilibrium** occurs when every player has no incentive to change their strategy. There are different types:

**Mixed Equilibrium** occurs when a player decides what they’re going to do based on the probability of a certain outcome happening, e.g., a player decides to serve a tennis ball favoring right because the player is left-handed, so they run a generator based on the probability and decide based on that.

**Deterministic Equilibrium** is just the normal Nash equilibrium: make the best-case decision regardless of what the other player(s) do.

**SIMULTANEOUS VS SEQUENTIAL GAMES**

For simultaneous games, you can form an **outcome matrix**.

For sequential games, you can represent the outcomes as leaves on a tree. You can find the Nash equilibrium by **searching up the tree and using backwards induction**.

**MINIMAX**

Minimax is basically just players trying to find the best worst-case scenario. The algorithm can be implemented as DFS, but it only sometimes examines every node in the game tree, because of alpha-beta pruning.

**ALPHA-BETA PRUNING**

ABP can eliminate entire branches of trees based on the outcome of one of the leaves. For example, if the worst-case outcome for a player in one branch of the tree is 3, and an outcome for the player in the other branch happens to be less than 3, the player will never go down this branch because the worst-case is, well, worse.

Since exploring a game tree can be computationally heavy, the depth of the minimax algorithm is limited. If the depth of the search is reached and the state is not terminal (result when both players play optimally), a score still must be given; this is calculated by a heuristic function.

---------------------------------------------------------------

**REINFORCEMENT LEARNING**

**Building a Theoretical Model**

Set of states

Set of actions

Information: At time , observe state and reward

Goal: Find map from state to actions that maximizes reward. This is called a **policy.**

**MARKOV DECISION PROCESS**

Defining a problem:

State set . Initial state . Action set .

Reward function : must output a scalar, and assume that you have infinite time (if you have finite time your policy might change as time changes for example more time would cause you to try the method with low risk/high rewards, short time is vice versa)

**State transition model**:

Essentially, this is a function that outputs the probability of getting to when the agent takes action when it is currently at state

**Markov’s Assumption**: The transition probability only depends on and , and not earlier history

**The Policy:**

The policy is a function that takes in a state and outputs the best action to be taken