University College London, DCN PhD Project

Notes on ML, Comb Opt, etc.

Supervisor: Dr. George Zervas

Christopher Parsonson

2020

Contents

Part I.

Elementary Reinforcement Learning

1. Introduction to Reinforcement Learning

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

1.1. What is Reinforcement Learning?

- Reinforcement learning is the 'science of decision making'
- Unique aspects of RL compared to other ML paradigms:
  - No supervisor, only reward signal
  - Feedback is delayed > only retrospectively learn good & bad decisions
  - Learns in sequential-time dynamic World rather than static data
  - Agent's actions influence subsequent data it receives & learns from

1.2. Definitions

- **Reward hypothesis**: All goals can be described by the maximisation of expected cumulative reward. This forms the basis of RL, which always assumes that all decisions can be evaluated with a quantitative & determinable reward
- **Reward** R_t: A scalar feedback signal. Indicates how well agent doing at time step t. Agent's goal is to maximise cumulative reward. Examples:
  - Minimising time to complete task: Issue -1 reward each time step, terminate episode after completion/some amount of time > agent learns to minimise task completion time
  - Managing an investment portfolio: +1 each £ made
  - Helicopter stunt manoeuvres: +ve for following desired trajectory, -ve for crashing
- **Sequential decision making**: The process of making decisions one after another.

  - Goal: Select actions such that the long-term future reward is maximised
  - Actions will have consequences on reward, & this reward can be be delayed
  - Whole point of RL is to train agent that can make decisions that may not look good in short term but will enable agent to outperform simple heuristics and/or humans in long term

- **Agent and Environment**: At each time step t, the **agent** executes an action A_t and receives an observation O_t & a reward R_t. The **environment** receives action A_t and emits an observation O_{t+1} and a scalar reward R_{t+1}
- **History** H_t: All the observables that the agent has been exposed to up to time t. Is a sequence of observations, actions, and rewards. Agent uses the history to decide what to do given a state:

$$H_t = O_1, R_1, A_1, ..., A_{t-1}, O_t, R_t \tag{1.1}$$

- **State** S_t: The information given to the agent at time t. Since the state is a function of the history, the history can be represented using the current state:

$$S_t = f(H_t) \tag{1.2}$$

- **2 states in RL**:
  1. **Environment state** S_t^e: The environment's representation of the state. Is the data used by the environment to pick the next observation & reward. Not usually visible to the agent, and often contains irrelevant info for the agent. The agent only gets to see its own representation of the state.
  2. **Agent state** S_t^a: The agent's representation of the state. Is the set of variables that we give to the agent to tell it what is happening to it. It is usually the state we refer to. It can be any function of the history:

$$S_t^a = f(H_t) \tag{1.3}$$

- **Information (Markov) state** S_t: A state that has the **Markov property**, whereby the probability of the next state being S_{t+1} given the current state S_t is the same as the equivalent probability were you given all the previous states in history H_t.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, ..., S_t) \tag{1.4}$$

  - I.e. with only the current state, if the current state has the Markov property, you can make predictions about the future as if you had all the relevant historic information too
  - As such, states with the Markov property can have their environment history disregarded, and you can use the current state to predict the future just as well as if you had the whole history

  - Therefore if the current state is a Markov state, then the historic information is irrelevant
  - The environment state S_t^e and the history H_t are Markov states by definition
  - **Example**: Consider a bag with 3 balls; 2 red and 1 green. In a random experiment, 1 ball was drawn yesterday, 1 ball was drawn today, and 1 ball will be drawn tomorrow. Consider two versions of this experiment: **(1)** all balls are drawn are *without* replacement **(2)** all balls are drawn *with* replacement.
    1. **W/o replacement**: To predict tomorrow's draw, you will make a better prediction if you have the historic information rather than just the present state, therefore this stochastic process of drawing coloured balls does *not* have the Markov property. E.g. if you know today's ball was red but don't know what yesterday's ball was (i.e. you only have present state and not historic info), chance of tomorrow's ball being red is $1/2$. On the other hand, if you know both today and yesterday's balls were red, then you know you will definitely get a green ball tomorrow i.e. your prediction is improved if you have the historic info rather than just the present state, therefore state does not have the Markov property.
    2. **W/ replacement**: Now the ball is replaced after being drawn, therefore the current state holds just as much relevant information as the environment history and can be used just as well as the history to make predictions. Therefore tomorrow's draw *is* only dependent on the present state, and the historic information is irrelevant, therefore this stochastic process of drawing balls *does* have the Markov property.

- **Fully observable environments**: Environments where the agent is exposed directly to the environment state

$$O_t = S_t^a = S_t^e \tag{1.5}$$

  - Here, the agent's state is a Markov state
  - The RL agent's sequential decision process is defined as a **Markov Decision Process** (MDP) (see next lecture)
  - The majority of this course will look at MDPs i.e. problems where the agent has access to the complete environment state and therefore the agent state has the Markov property in that it has complete knowledge of how the environment will generate the next observation & reward given only the current state

- **Partially observable environments**: Environments where the agent is exposed only to part of the environment state

$$S_t^a \neq S_t^e \tag{1.6}$$

  - Here, the agent is not exposed to a Markov state, and it must learn its own state representation S_t^a

  - To make predictions about the future, the agent must learn to construct its own state representation S_a^t such that S_a^t can be approximated as a Markov state
  - The RL agent's sequential decision process is defined as a **partially observable Markov decision process** (POMDP)

- **Major components of an RL agent**: RL agents contain ≥ 1 of the following components:

  1. **Policy function** π: A function mapping states to actions. Is the agent's behaviour. The policy an agent follows can be changed, but the policy itself is stationary/time-independent i.e. a given policy is static. Can have a:
     - *Deterministic policy* where the action is chosen from the policy $\pi(s)$ given state s according to which action has the highest estimated long-term reward

$$a = \pi(s) \tag{1.7}$$

     - *Stochastic policy* where the action is chosen according to a probability distribution (where this probability distribution is the policy) (i.e. each action a has a probability of being chosen given some state s)

$$\pi(a|s) = P(A_t = s|S_t = s) \tag{1.8}$$

  2. **Value function**: A function that predicts what the agent's expected future long-term reward will be, thereby evaluating the agent's current position and assigning it a **value**.
     - It predicts *total expected future reward* (termed the **value** or the **return** G_t (see next lec)), *not* reward for just next step
     - Discount factor γ (see next lec) determines how much future rewards are discounted by and therefore how far into future agent considers when evaluating its current position
     - Two types of value function:
       a) **State-value function** $v_\pi(s)$: The expected return starting from state s and following policy π:

$$\begin{aligned} v_\pi(s) &= E_\pi(R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + ...|S_t = s) && (1.9)\\ &= E_\pi(G_t|S_t = s) && (1.10) \end{aligned}$$

       b) **Action-value function** $q_\pi(s, a)$: The expected return starting from state s, taking action a, and then following policy π:

$$\begin{aligned} q_\pi(s, a) &= E_\pi(R_{t+1}^a + \gamma R_{t+2}^a + \gamma R_{t+3}^a + ...|S_t = s, A_t = a) && (1.11)\\ &= E_\pi(G_t|S_t = s, A_t = a) && (1.12) \end{aligned}$$

  - Value is an *expectation* because Markov states are stachostic therefore cannot be 100% sure about future states therefore just *expect* a certain future return (i.e. expected value) from being in state s

3. **Model $\mathcal{M}$ ($\langle \mathcal{P}, \mathcal{R} \rangle$)**: What the agent predicts the environment will do next. Is the agent's learned representation of the environment/MDP. Assumes state space $\mathcal{S}$ & action space $\mathcal{A}$ are known. Has 2 components:
   - **State transition matrix $\mathcal{P}$**: Defines the transition probabilities from all states s to all successor states s'
     * Rows = current state s, columns = next state s'
     * Each row of the matrix sums to 1 since you will definitely transition to some state in the next time step. For n possible states:

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{1,1} & \cdots & \mathcal{P}_{1,n} \\ \vdots & & \\ \mathcal{P}_{n,1} & \cdots & \mathcal{P}_{n,n} \end{bmatrix}$$

     * Using the state transition matrix, given action a and current state s, can get probability that will transition to state s' (the **state transition probability**):

$$\mathcal{P}^a_{ss'} = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{1.13}$$

   - **Expected immediate reward $\mathcal{R}$**: Expected immediate (next state) reward given current state s and chosen action a

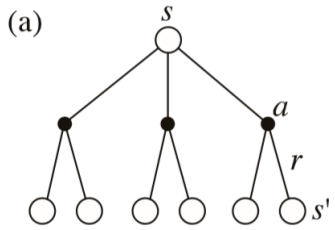
$$\mathcal{R}^a_s = E(R_{t+1} | S_t = s, A_t = a) \tag{1.14}$$

- **5 categories of RL agents**:
  1. **Value based**: Agent picks actions greedily according to which action has the highest value/expected future reward. There is no explicitly defined policy as the policy is implicit in the value function.
  2. **Policy based**: Agent only stores a policy function mapping states to actions. Does not explicitly store a value function since the value of each state is implicit in the policy function.
  3. **Actor critic**: Agent stores both a policy function and a value function. Alternates between each to update both as the agent learns
  4. **Model free**: Agent does not explicitly build a model representation of the environment with $\mathcal{P}$ and $\mathcal{R}$, just uses a policy and/or value function(s) to make decisions. Many RL algorithms are model-free

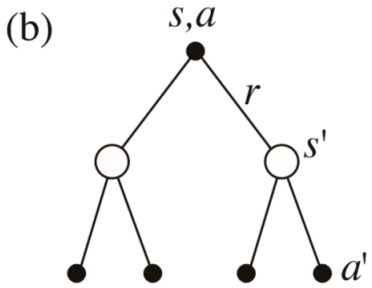
5. **Model based**: Agent builds a model representation of the environment with $\mathcal{P}$ and $\mathcal{R}$ and uses it with its policy and/or value function(s) to make decisions.

- **2 fundamental problems in field of sequential decision making**: There are 2 possible approaches to solving decision problems such that the optimal choice is made
  1. **Learning** (i.e. RL):
     - Environment initially unknown
     - Agent externally interacts with environment to improve its policy
  2. **Planning**:
     - Model of environment is known from the beginning, therefore everything about environment (reward, rules etc.) already known
     - Agent uses known model to perform internal computations without any external interaction with the environment to improve its policy
- **Exploration**: Process used by RL agent to find out more information about its environment
- **Exploitation**: Process used by RL agent to maximise its long-term expected future reward. Usually it's important to both explore and exploit
- Two aspects of sequential decision making & MDPs:
  1. **Prediction**: Given a policy π, find the value function v that correctly evaluates how well you will do if you follow the policy.
     - Input: MDP & policy π (or just an MRP, see next lec)
     - Output: Value function v_π
  2. **Control**: Given a value function v, find the policy π that maximises your future reward. To solve the control problem, must first solve the prediction problem such that you have found the optimum value function v_* (see next lec), and then you will be able to find the optimal policy π_* that maximises reward.
     - Input: MDP
     - Output: Optimum value function v_* & optimal policy π_*
- **Backup diagrams**
  - At the heart of RL is the concept of being in a state, taking an action, observing what happens when that action is taken, and then using that returned information to *backup* (a.k.a. *update*) the information to the state or state-action pair that you were initially in. This process is often represented with **backup diagrams**
  - In backup diagrams, time always flows downwards

- Conventionally, hollow vertices indicate states and solid black vertices indicate actions. Their dependencies are expressed by edges connecting the vertices
- For example, consider the backup diagram below. Each open circle represents a state and each solid circle represents a state-action pair. Starting from state s, the root node at the top of the diagram, the agent could take any of some set of actions (in this case, one of the three possible actions shown). From each of these, the environment could respond by putting the agent in one of several next states s' (according to the environment's transition probability matrix $\mathcal{P}$), and with a returned reward r. This backup diagram is the one-step look-ahead used by the Bellman equation (see next lec) when averaging over all possibilities and weighting each possibility by its probability of occurring when evaluating a policy π with value function v_π:



- Similarly, the backup diagram for the Bellman equation when doing a one-step look-ahead to get q_π involves being in state s and, rather than considering all possible actions, taking a specific action a, ending up in some new state s', then considering possible action a' that can be taken when in s':



- By convention, if we are considering taking the action of a set of actions that will give the maximum expected reward, we draw an arc across the possible actions. E.g. backup diagrams for Bellman optimal v_* and q_*:

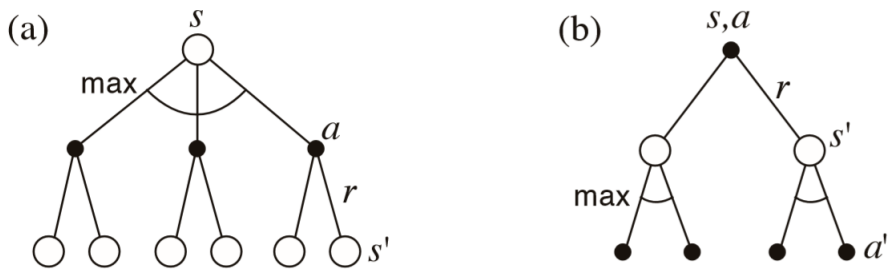


Figure 3.7: Backup diagrams for (a) v_* and (b) q_*

2. Markov Decision Processes

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

2.1. Markov Processes

- **Markov process (a.k.a. Markov chain)**: A memoryless (no historic information explicitly stored) random sequence (process/chain) of states $\{s_1, s_2, ...\}$ where each state has the Markov property
  - A Markov process is fully described by tuple $\langle \mathcal{S}, \mathcal{P} \rangle$
  - $\mathcal{S}$ is a finite set of possible states
  - $\mathcal{P}$ is a state transition probability matrix
- **Sample**: A sequence of states

2.2. Markov Reward Processes

- **MRP reward function** $\mathcal{R}_s$: Quantitatively describes reward we get from being in state s at time t

$$\mathcal{R}_s = E(R_{t+1}|S_t = s) \tag{2.1}$$

- **Discount factor** γ: Factor by which to multiply future rewards to discount their present value
  - $\gamma \in [0, 1]$
  - Higher γ discounts future rewards less and therefore encourages agent to care less about imminent rewards and more about maximising long-term future rewards. This is **far-sighted evaluation**
  - Lower γ will lead to caring about immediate rewards more. This is **myopic evaluation**

  - Use of γ ensures that:
    * Total future reward (return G_t) does not go to ∞
    * The (imperfect) model's uncertainty about the future is factored into its evaluations
    * Humans & animals discount rewards in that they tend to prefer immediate rewards. Since nature has found discounted rewards useful, it is likely a good idea
  - Undiscounted Markov reward processes ($\gamma = 1$) can be used for episodic tasks that will definitely terminate and therefore G_t will not go to ∞ e.g. chess

- **Markov reward process (a.k.a. Markov reward chain)**: A Markov chain where each state in the chain has a **value** that quantitatively describes how good or bad it is to be in that particular state
  - A Markov reward process is fully described by tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
  - $\mathcal{S}$ is a finite set of possible states
  - $\mathcal{P}$ is a state transition probability matrix

$$\mathcal{P}_{ss'} = P(S_{t+1} = s' | S_t = s) \tag{2.2}$$

  - $\mathcal{R}$ is a reward function

$$\mathcal{R}_s = E(R_{t+1} | S_t = s) \tag{2.3}$$

  - γ is a discount factor

- **Return** G_t: The total discounted reward from time step t until the end of the episode.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... \tag{2.4}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

  - I.e. the value of receiving reward R after $k + 1$ time steps is $\gamma^k R$
  - Goal of RL is to maximise G_t i.e. to maximise the long-term reward

- **MRP state value function v(s)**: A function that takes a state s and gives the long-term value (i.e. the total reward we expect to get from t until the end of the episode) from being in state s.
  - The value function is essentially a sequence of cumulative rewards i.e. from time step t to end of episode, sum discounted rewards you get at each time step. This is equivalent to the retun G_t

  - N.B. Since we are talking about a Markov reward process, *not* a Markov decision process, we are not considering policies and actions etc. therefore MRP is $v(s)$ rather than being a function of policy π/chosen action a as we saw in previous lecture

$$v(s) = E(G_t|S_t = s) \tag{2.6}$$

- **The Bellman Equation for MRPs**: The Bellman equation expresses the value $v(s)$ of a state s at a point in time t. This is equivalent to the state value function $v(s)$ we wrote in Eq 2.6. Expanding Eq 2.6 by subbing in Eq 2.4, we get the Bellman equation for MRPs:

$$\begin{aligned} v(s) &= E(G_t|S_t = s) && (2.7)\\ &= E(R_{t+1} + \gamma v(S_{t+1})|S_t = s) && (2.8) \end{aligned}$$

  - The Bellman equation decomposes the value function $v(s)$ into 2 parts: (1) The immediate reward R_{t+1} from being in state $S_t = s$, and (2) the discounted value $\gamma v(S_{t+1})$ of being in successor state S_{t+1}
  - This is useful because it breaks down a dynamic optimisation problem (how to maximise our long-term reward in a dynamic environment) into a simpler sequence of sub-problems
  - A correct value function $v(s)$ learnt by an RL agent will always obey the Bellman equation since it will be able to predict the return accurately given any state s. This is useful because if our $v(s)$ does not obey the Bellman equation, we know we have not yet found the correct $v(s)$
  - The Bellman equation is a *linear equation*. It can be solved with **direct** solutions to find the value function $v(s)$ (i.e. to solve the MRP by finding a function $v(s)$ that correctly maps the current state to a value), however the computationally complexity for n states is $O(n^3)$ (i.e. cubes w/ no. states)
  - Therefore to solve the Bellman equation for large MRPs and find the correct value function $v(s)$, **iterative** solutions such as **dynamic programming**, **Monte-Carlo evaluation**, and **temporal-difference learning** are often used

2.3. Markov Decision Processes

- **Markov Decision Process (MDP)**: A Markov reward process with **decisions**. As with all Markov processes, each state in the MDP environment is a Markov state.
  - An MDP is a sequential decision process where the environment in which decisions are being made is fully observable i.e. the current state given to the decision maker completely captures all the *relevant* history of the environment, and therefore is sufficient to make predictions about the future (i.e. the states in the environment have the Markov property).

  - All RL problems can be approximated to/formalised as being MDPs
  - POMDPs can be converted into MDPs
  - 'Bandits' are MDPs with one state
  - A Markov decision process is fully described by tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
  - $\mathcal{S}$ is a finite set of possible states
  - $\mathcal{A}$ is either a discrete (finite) or continuous (infinite) set of possible actions
  - $\mathcal{P}$ is a state transition probability matrix

$$\mathcal{P}^a_{ss'} = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.9}$$

  - $\mathcal{R}$ is a reward function

$$\mathcal{R}^a_s = E(R_{t+1} | S_t = s, A_t = a) \tag{2.10}$$

  - γ is a discount factor
  - N.B. Now that we are considering MDPs rather than MRPs, our transition probability matrix $\mathcal{P}$ and reward function $\mathcal{R}$ (and consequently value function) are dependent on the policy π/chosen action a as well as the current state s (as we considered in lecture 1)
  - Given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π, the sequence of resultant states from following π in $\mathcal{M}$, $\{S_1, S_2, ...\}$, is a **Markov process** $\langle \mathcal{S}, \mathcal{P}^\pi \rangle$, and the state-reward sequence $\{S_1, R_2, S_2, ...\}$ is a **Markov reward process** $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$ i.e. the building blocks of an MDP are the Markov process and the MRP that we've already seen this lecture

- **The Bellman Equation for MDPs**: Just as with the MRP Bellman equation in Eq 2.8, the MDP Bellman equation expresses the value by decomposing it into two parts: (1) The immediate reward R_{t+1}, and (2) The discounted value of the successor state $\gamma v_\pi(S_{t+1})$. However, because we are now dealing with an MDP rather than and MRP, decisions are involved, therefore the MDP Bellman equation also considers the action a and/or policy π. There is a Bellman equation for the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$:
  - State-value Bellman eq:

$$v_\pi(s) = E_\pi(R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s) \tag{2.11}$$

  - Action-value Bellman eq:

$$q_\pi(s, a) = E_\pi(R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a) \tag{2.12}$$

- **Optimum value function**: The optimum value function will correctly assign a value to each possible position in the MDP, and therefore specify the best possible performance in the MDP. An MDP is 'solved' when we have found the optimal value function.
  - Optimal state-value function $v_*(s)$ (a.k.a. the **Bellman optimality equation for the state-value function**) is the maximum value function over all policies π:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.13}$$

  - Optimal action-value function $q_*(s, a)$ (a.k.a. the **Bellman optimality equation for the state-action function**) is the maximum action-value function over all policies π:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.14}$$

  - The Bellman optimality equation is non-linear, and in general there is no direct closed-form solution, therefore iterative methods must be used to solve it to find the optimum value function. Iterative solution methods include value iteration, policy iteration, Q-learning, and Sarsa, which we shall see more of in later lectures.
- **Optimal policy**: The optimal policy π_* is the policy (method for making decisions) that achieves the optimum value function.
  - A policy π is better than or equal to another policy π' if the value for ***all*** states with π, v_π, is greater than or equal to the value for all states with π', $v_{\pi'}$:

$$\pi \geq \pi' if v_\pi(s) \geq v_{\pi'}(s), \forall s \tag{2.15}$$

  - For any MDP, the following statements are true:
    1. There's always at least 1 optimal policy π_* that's better than or equal to all other policies:

$$\pi_* \geq \pi, \forall \pi \tag{2.16}$$

    2. All optimal policies achieve the optimum value function $v_{\pi_*}(s)$:

$$v_{\pi_*}(s) = v_*(s) \tag{2.17}$$

    3. All optimal policies achieve the optimum action-value function $q_{\pi_*}(s, a)$:

$$q_{\pi_*}(s, a) = q_*(s, a) \tag{2.18}$$

3. Planning by Dynamic Programming

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2^{nd} edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

3.1. What is Dynamic Programming?

- **Principle of Optimality**: An optimal policy solution has the property that whatever the initial conditions and control variables (choices) over some initial period, the control (or decision variables i.e. the policy) chosen over the remaining period must be optimal for the remaining problem.
  - The principle states that any optimal policy solution can be sub-divided into 2 components: (1) An optimal first action A_* (i.e. can greedily choose an action at first time step), and (2) An optimal policy from successor state S' (i.e. can continue choosing greedy actions for all later iterations).
  - I.e. a policy π that chooses action a given state s, $\pi(a|s)$, achieves the optimum value from state s, $v_\pi(s) = v_*(s)$, if and only if for any state s' reached from s, π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$.
  - This is the basic principle behind dynamic programming.
- **Dynamic Programming (DP)**: A general method for solving complex sequential problems that have 2 properties:
  1. Optimal substructure: Must be able to break the solution down into series of sub-problems (i.e. the **principle of optimality** applies)
  2. Overlapping sub-problems: These sub-problems must re-occur many times such that solutions to the sub-problems can be cached and re-used
  - DP works by breaking the complex problem up into sub-problems, solving the sub-problems, then combining the solutions to the sub-problem to 'solve' the complex problem

- MDPs have both the above properties: The Bellman eq. gives recursive decomposition, meaning solving an MDP can be broken down into a series of sub-problems (giving property (1)), and the value function predicts future states & rewards in the MDP and therefore enables solutions to be stored and re-used (giving property (2)). Therefore, MDPs can be solved using DP
- When solving the MDP, DP assumes full knowledge of the MDP from the beginning, therefore it assumes the environment and everything about it is already known, therefore DP is a **planning** (as opposed to a learning) approach to solving MDPs. DP uses the known model to perform internal computations without any external interaction with the environment to improve its policy
- DP is used for e.g. scheduling, finding shortest paths, solving MDPs etc
- In relation to MDPs, it can be used for solving both the prediction and the control problem (see lecture 1)

3.2. Policy Evaluation

- Consider that we want to evaluate a given policy π in an MDP, and that we have full knowledge of the environment/MDP from the beginning
- Since we are evaluating a given policy, this is a **prediction** task (as opposed to a control task). Since we have full knowledge of the MDP from the beginning, this is a **planning** problem (as opposed to a learning problem). Since it is a planning problem in an MDP, we can use **dynamic programming** to solve the prediction task and find the v_π that correctly evaluates the policy π
- **Iterative policy evaluation**:
  - Start with a value function at step $k = 0$, v_0, which assumes a value of 0 for all states
  - Using known model, move to next step $k = 1$ by following given policy π until episode termination. Using reward(s) returned by known environment in this internal simulation, update v_0 to v_1, which is the next iteration of the value function for the policy π i.e. you are updating the value for each state in the MDP predicted by the value function
  - Continue this iterative process until v no longer changes. The value function that you converge on is guaranteed to be the value function v_π given by the Bellman eq. that correctly evaluates policy π (this has been proved mathematically).

3.3. Policy Iteration

- Consider now that we want to find the optimal policy π_* that maximises our reward in the MDP. This is now a **control** task, which again we can use DP for to solve.

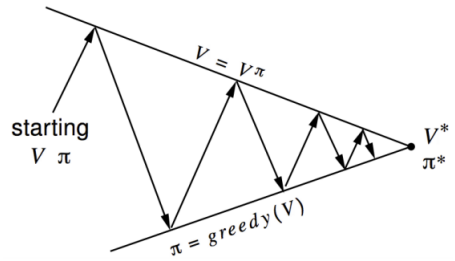
- We've seen that we can use iterative policy evaluation to iteratively find v_π which correctly evaluates a given policy π by using DP

$$v_\pi(s) = E_\pi(R_{t+1} + \gamma R_{t+2} + ...|S_t = s) \tag{3.1}$$

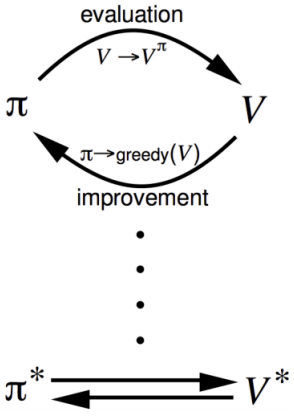
- We can then take our iteratively found v_π and use it to define a new policy π' that uses v_π such that the reward is maximised (**greedy policy improvement**).

$$\pi' = greedy(v_\pi) \tag{3.2}$$

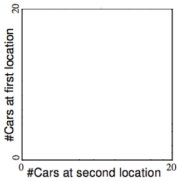
- We can then take this improved policy π' and use iterative policy evaluation to iteratively find $v_{\pi'}$, which can again be used to find an even better policy with greedy policy improvement
- This iterative procedure is **policy iteration**, whereby *policy evaluation* is done to estimate v_π using iterative policy evaluation, and then *policy improvement* is done to generate a new and better policy $\pi' \geq \pi$ with greedy policy improvement. This iterative process can be repeated until the optimal policy π_* and its corresponding optimum value function v_* are converged upon, thereby solving the control problem by solving the Bellman optimality equation iteratively with DP:



Policy evaluation Estimate v_π
Iterative policy evaluation
Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement



- **Real Example: Car Rental**
  - Jack owns 2 car rental shops that each hold $\leq$ 20 cars, and he owns 40 cars in total. People randomly request to rent cars from each location. Each night, Jack can move $\leq$ 5 cars between the locations. If no car is available to rent at a shop when a request comes in, Jack will get \$0. If a car is available, Jack gets \$10. Jack wants to find the optimal policy for moving cars around such that his income is maximised.
  - Formalising problem as an MDP:
    - **States**: 2 locations, maximum of 20 cars at each, therefore $20 \times 20 = 400$ possible states s in S:



* **Actions**: Move up to 5 cars between locations overnight

$$a \in [0, 5] \tag{3.3}$$

* **Reward**: \$10 for each car rented (must be available)

$$R = \begin{cases} 10, & \text{if rented.} \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

* **State transitions**: Cars are requested and returned at random points in time according to a Poisson distribution, where the 1st location has an average of 3 requests and 3 returns, and the 2nd location has an average of 4 requests and 2 returns

– Solving problem:

* Finding the optimal policy is a control problem. If we assume the MDP is initially known (i.e. we know probability of rental requests etc.) and can therefore run internal simulations without interaction with the external environment, then we can use DP to solve this MDP and find the optimal policy.

* Solving with DP policy iteration:

  · Start with initial policy π_0 where move no cars ($a = 0$) for all states, and with initial value function v_0 where estimate 0 reward for all possible states

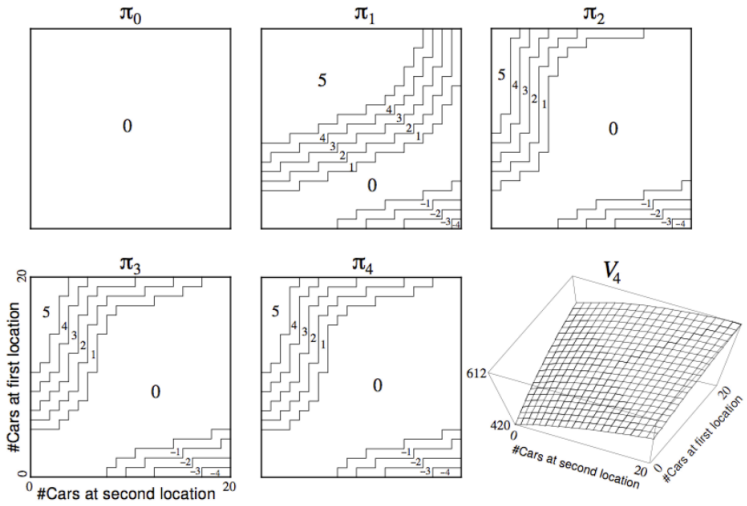
  · Evaluate this policy using iterative policy evaluation using internal simulation of the known MDP until converge on v_{π_0} for policy π_0

  · Use greedy policy improvement to find a new and better policy, π_1, where $\pi_1 = greedy(v_{\pi_0})$

  · Repeat this policy iteration process until converge (i.e. improvements stop) on a policy. At this point the Bellman optimality equation will have been satisfied, therefore the optimal policy π_* that maximises Jack's return will have been found, since anything that satisfies the Bellman optimality equation is optimal by definition.

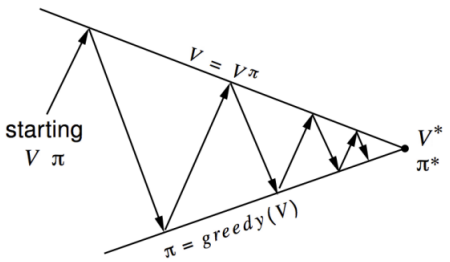
  · From perspective of second location, a +ve number in state space means we're moving cars out of first location and adding to second location, and a -ve number means we're moving cars out of second location and giving

to first location. As we might expect, the policy that's converged on will move more cars away from a location if it is close to maximum capacity (20 cars) since the cars will probably be more useful at the other location, but it will prefer to move cars from location 1 rather than from location 2 since location 2 has a higher average number of requests:

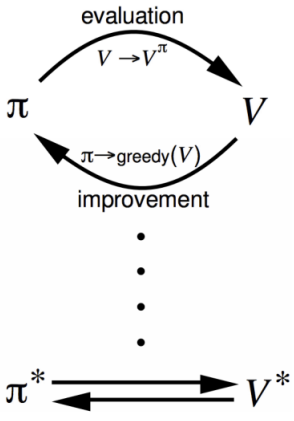


- N.B. The final figure in the above is a plot of the states on the x-y axis and the expected return ($) of following policy π_4 for each possible state, therefore it is a plot of the value function v_4. Since π_4 is our converged on policy, $\pi_4 = \pi_*$ and $v_4 = v_*$. As expected, a higher number of cars at each location leads to a higher expected total reward.

- **Generalised policy iteration**: So far have considered policy iteration as doing policy evaluation by **iterative policy evaluation** and policy improvement by **greedy policy improvement**. The general case of this policy iteration process can use **any** policy evaluation algorithm and **any** policy improvement algorithm:



Policy evaluation Estimate v_π
Any policy evaluation algorithm
Policy improvement Generate $\pi' \geq \pi$
Any policy improvement algorithm



3.4. Value Iteration

- **Value iteration**: Similar to policy iteration, but now we are iteratively updating the value for each state in the MDP until converge on the optimal value function $v_*(s)$
  - Unlike policy iteration, there is no explicit policy that we're evaluating. We are simply evaluating a state i.e. what's the value of being in this state (ignoring the policy)?
  - Do this by starting at initial guess value function v_0 and at each $k+ = 1$ iteration, for all states $s \in S$, update $v_{k+1}(s) from v_k(s')$. Repeat until converge on a value function, which will be the optimum value function v_* which correctly evaluates each state in the MDP

3.5. Extensions to Dynamic Programming

- **Synchronous Dynamic Programming**: Where all states are updated/backed up in **parallel** i.e. update all states at the same time for each $k+ = 1$ step. This is what we've seen so far in this lecture course for policy & value iteration
- **Asynchronous Dynamic Programming**: Where each state is updated/backed up **individually** (in any order).
  - Asynchronous DP can significantly reduce computation time since you don't need to update all steps on each time step. ***So long as all states continue to be selected*** for a backup/update, then asynchronous DP is still guaranteed to converge on the solution
  - 3 key ideas in field of asynchronous DP:
    1. **In-place DP**: Asynchronous DP only stores one copy of the value function, whereas synchronous DP stores two (the old and the new v), therefore asynchronous DP uses less memory
    2. **Prioritised sweeping**: To decide which order individual states should be backed up/updated, can apply a prioritised sweep through all the states whereby you calculate the difference between the states' current value (evaluated with the current value function $v(s)$) and compare it to the known Bellman equation to get the magnitude of the **Bellman error**. The state with the largest remaining Bellman error is the one that should be backed up/updated first
    3. **Real-time DP**: Rather than sweeping through all states as done with prioritised sweeping, put a live agent in the (internally simulated) environment and only update/backup the states that the agent visits. I.e. the agent's experience guides state selection

- **Full-width backups**: Both synchronous and asynchronous DP use full-width backups/updates, whereby *all* actions & *all* states that might occur in the future are considered. Full-width backups are what we've seen throughout this lecture.
  - Full width backups are computationally expensive and therefore can only do on small- to medium-sized problems with up to a few million states
  - For large problems, DP suffers from **Bellman's curse of dimensionality**, whereby the number of states $n = |S|$ grows exponentially with the number of state variables
  - For many real-World problems, even 1 full-width backup/update can be too expensive
  - Full-width backups also require complete knowledge of the MDP transitions and reward function from the beginning, which in many cases is not possible, therefore cannot apply DP to situations where MDP not fully known
- **Sample backups**: This is where a *sample* (rather than the 'full width') of possible future states, actions and rewards are considered
  - Rather than using reward function $\mathcal{R}$ and transition matrix $\mathcal{P}$, sample backups use a sampled set of states, actions and rewards $\langle S, A, R, S' \rangle$
  - Advantages:
    1. Model-free: Don't need to know $\mathcal{R}$ and $\mathcal{P}$, therefore no advanced knowledge of the MDP is required
    2. Breaks the curse of dimensionality through sampling, since no longer need to consider every possible future state & action, only those in our sample
    3. Since the computational cost of the backup is dependent only on the size of the sample, it is constant and independent of the number of states $n = |S|$
  - Sample backups are how we go from DP methods, which can't handle large state problems & which require advanced complete knowledge of the MDP, to using model-free RL methods
- **Approximate DP**: Where we use a **function approximator** to approximate the value function by only considering **sampled** states rather than the full-width of all possible states. This is effectively what RL is.

4. Model-Free Prediction

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2^{nd} edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

4.1. What is Model-Free Prediction?

- Previous lectures have considered the **planning** approach to solving sequential decision problems, where a perfect model of the environment/the MDP is **known** from the beginning. Dynamic programming is a popular planning approach to solving both the prediction and the control tasks in sequential decision making
- Now we are going to consider the **learning** approach to solving prediction and control tasks for a sequential decision making problem, where we are approximating the value function for an **unknown** MDP. This is a **model-free** problem, because we no longer know exactly how the environment works at the beginning; we have to **learn** our own representation of the environment and its inner workings
- This lecture we will look at model-free **prediction** (evaluating a policy). The next lecture will look at model-free **control** (finding the optimum value function and optimal policy)
- The main methods for solving model-free prediction & control tasks are **Monte-Carlo RL** and **Temporal Difference RL**

4.2. Monte-Carlo Learning

- **Episodic task**: A task that will definitely terminate and therefore will always last a finite amount of time before termination. This is as opposed to a **continuous task**, which has no definite termination and therefore can potentially run for an infinite amount of time
- **Bootstrapping**: Agent uses its own value function to **estimate** future returns rather than using actual returns.
  - Updates estimate of state/state-action pair values based on the estimates (value function) of successor states/state-action pair. Therefore to bootstrap, must consider

every state-action possibility since updates state estimate based on estimates of successor states, therefore is computationally expensive and as number of states increases, compute power needed to bootstrap massive increases.

  - DP & TD methods can use bootstrapping since they can be applied to non-episodic tasks, whereas MC methods cannot use bootstrapping and must instead only use the actual returns from sampling

- **Sampling**: Agent uses a **set** of experiences (rather than every possible experience/state-action) to update its value function. In the case of MC, the **actual** return of the sample is used, whereas for TD, the **expected** return of the sample is used.
  - Don't need to consider every state-action possibility, only those that you have sampled, therefore is less computationally expensive than when no sampling is used
  - MC & TD methods sample (i.e. only consider a *sample* set of possible state-action possibilities), DP methods do not (i.e. DP considers *every* state-action possibility)
- **Monte Carlo Reinforcement Learning**: A model-free approach to solving prediction and control tasks in a sequential decision making problem/MDP.
  - Learn directly from **episodes** of experience
  - Learning is **model-free** (no initial knowledge of MDP transitions/rewards)
  - Use **sampling** whereby learning is done from the **actual** (empirical) returns that were gained from experience rather than from the *estimated* future returns. Since it uses actual return rather than expected return, it cannot estimate future returns with bootstrapping, therefore episode must always terminate, therefore **cannot use MC learning for non-episodic tasks**
  - Advantages over TD RL:
    1. **Unbiased**: MC uses the actual return rather than the estimated return, therefore is not biased towards the initial value function used to estimate return
    2. **Convergence with function approximation**: Converges to v_π even with function approximation (not always the case with TD)
    3. **Simple**: Is very simple and easy to use
    4. **Non-Markov environments**: MC can sometimes be better in non-Markov environments where all relevant historic information is *not* contained in current state, since MC averages over historic state visits. N.B. Often real-World states are in between Markov & non-Markov states (i.e. they're POMDPs), therefore must choose between MC & TD RL
- **MC policy evaluation**: Goal is to learn v_π (the value function that correctly evaluates policy π) from episodes of experience under policy π. Recall that in Eq. 1.10 we defined the value function as the *expected* return. MC policy evaluation, however, defines the value function as the **empirical** (real) **mean return**.

- Consider you want to evaluate a state s with MC RL by sampling multiple episodes of an experience under policy π. There are 2 MC policy evaluation methods for doing this by finding the empirical return of the state under the policy:

  1. **First-visit MC policy evaluation**:

     a) On the **first** time step t that state s is visited in an episode, incrementally add to a *state counter* $N(s)$ to keep track of no. times state s has been visited for the first time across all sampled episodes

     $$N(s) += 1 \tag{4.1}$$

     b) Take the actual/empirical total future return G_t and incrementally add it to the empirical total return across all episodes $S(s)$

     $$S(s) += G_t \tag{4.2}$$

     c) Estimate the value of being in state s by finding the mean of the total future return across all episodes

     $$V(s) = \frac{S(s)}{N(s)} \tag{4.3}$$

     d) By the **law of large numbers**, if we sample enough episodes (i.e. visit the state s for the first time across episodes enough times), as the number of times we visit s for the first time, $N(s)$, tends to ∞, $V(s)$ will converge on $v_\pi(s)$ and we will have found the correct evaluation of policy π

  2. **Every-visit MC policy evaluation**:

     a) For **every** time step t that state s is visited in an episode, incrementally add to a *state counter* $N(s)$ to keep track of the total no. times state s has been visited across all sampled episodes

     $$N(s) += 1 \tag{4.4}$$

     b) Take the empirical total future return G_t and incrementally add it to the empirical total return across all episodes $S(s)$

     $$S(s) += G_t \tag{4.5}$$

     c) Estimate the value of being in state s by averaging the total future return across all episodes

     $$V(s) = \frac{S(s)}{N(s)} \tag{4.6}$$

     d) Again, by law of large numbers, as $N(s)$ tends to ∞, $V(s)$ will converge on $v_\pi(s)$ and we will have found the correct evaluation of policy π

- **Real Example: Blackjack**
  - Consider that we want to evaluate a simple Blackjack policy π: That we stick if the sum of the cars is ≥ 20, otherwise we twist. This is a planning task (want to evaluate policy π therefore must find value function v_π), and we want to try to solve it using MC learning. N.B. for purposes of this example, the game of blackjack being formalised is a simplified version.
  - Formalising problem as an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$:
    - **States** $10 \times 10 \times 2 = 200$ possible states:
      - Current sum of cards in your hand (if less than 12, automatically ask for another card, therefore consider this part of environment to reduce number of possible states)
      - Dealer's showing card
      - Whether your hand contains a 'usable' (won't bust you) ace

      $$\text{sum} \in [12, 21], \text{card} \in [ace, 10], \text{ace} \in [1, 0] \tag{4.7}$$

    - **Actions**:

      $$a \in [\text{stick}, \text{twist}] \tag{4.8}$$

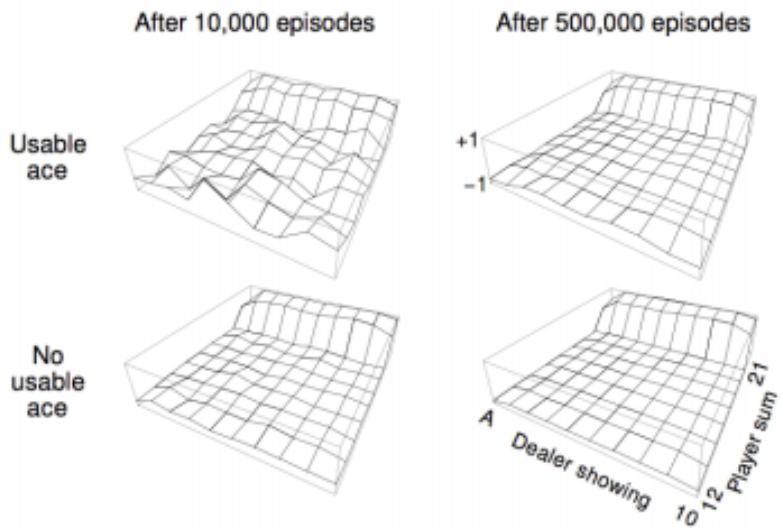
    - **Rewards**:

      $$R_{\text{a=stick}} = \begin{cases} +1, & \text{if sum cards} > \text{sum dealer cards.} \\ 0, & \text{if sum cards} = \text{sum dealer cards.} \\ -1, & \text{if sum cards} < \text{sum dealer cards} \end{cases} \tag{4.9}$$

      $$R_{\text{a=twist}} = \begin{cases} -1, & \text{if sum cards} > 21. \\ 0, & \text{otherwise.} \end{cases} \tag{4.10}$$

    - **Transitions**: Automatically twist if sum of cards < 12
  - Solving with MC learning:
    - Principle: Sample a large number of episodes (e.g. 500,000), take the mean value of each state $v(s)$ over all episodes using either first or every visit MC policy evaluation, and when we've visited all states enough times by sampling enough episodes, we will have converged on v_π
    - E.g. $v(s)$ after sampling 10,000 episodes, we see that because having a usable ace is rarer than having no usable ace, the $v(s)$ for usable ace is more noisy than for no usable ace. After sampling 500,000 episodes, this noise is smoothed out. N.B. height of graph (z axis) is how well we think we'll do long term in

current state, x axis is card dealer is showing, y axis is sum of cards in our hand. The converged on v_π shows that if we have less than sum=20 in our hand, our return is very low, highlighting how this simple policy we've evaluated is not very good. Crucially, this evaluation was learned by *trial-and-error* sampling with *no* initial knowledge of the game/MDP beforehand.

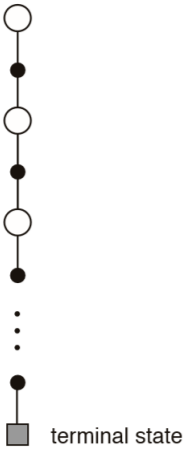


Policy: stick if sum of cards $\geq$ 20, otherwise twist

- **Incremental MC Updates**: Rather than computing mean return all at once, can compute the mean **incrementally** by updating it at the end of each episode rather waiting until finished experiencing all sampled episodes. After each update of mean, can update $v(s)$. Most MC methods use this to incrementally update $v(s)$ until converge on v_π. $v(s)$ can be updated at the end of each episodes with update rule (where α is the learning rate, which determines by how much we change $v(s)$ on each incremental update; if too high will oscillate around v_π and not converge, but if too low will be slow to converge on v_π):

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{4.11}$$

- Backup diagram for Monte Carlo learning estimation of v_π (i.e. follows specific path defined by policy π of state-actions all the way to terminal state):



4.3. Temporal-Difference Learning

- **Temporal Difference Reinforcement Learning**: Another model-free approach to solving prediction and control tasks in a sequential decision making problem/MDP. Crucial difference between TD and MC is that TD can break up tasks and therefore can handle non-episodic tasks with bootstrapping, unlike MC learning
  - Learn directly from **episodes** of experience
  - Learning is **model-free** (no initial knowledge of MDP transitions/rewards)
  - Uses **bootstrapping** whereby learning is done from the **estimated** future returns rather than from the *actual* returns. This allows TD to learn from experiences that have not yet terminated, therefore effectively breaking up the experiences into episodes and enabling it to handle non-episodic tasks. Therefore **can use TD learning for non-episodic tasks**
  - Uses **sampling** whereby learns from a sample set of experiences rather than having to consider every possible state-action (as DP does)
  - Advantages over MC RL:
    1. **Smaller & faster updates**: TD can learn online after every step since it estimates the final outcome to update its value function (i.e. it uses bootstrapping), whereas MC must wait until the end of the episodes before knowing actual return (i.e. it uses sampling)
    2. **Can handle non-episodic tasks**: TD can learn in continuous (non-episodic) environments that never terminate, whereas MC only works with episodic tasks
    3. **Lower variance**: TD updates its value function after just 1 step (1 action, transition & reward), whereas MC updates after all actions, transitions and

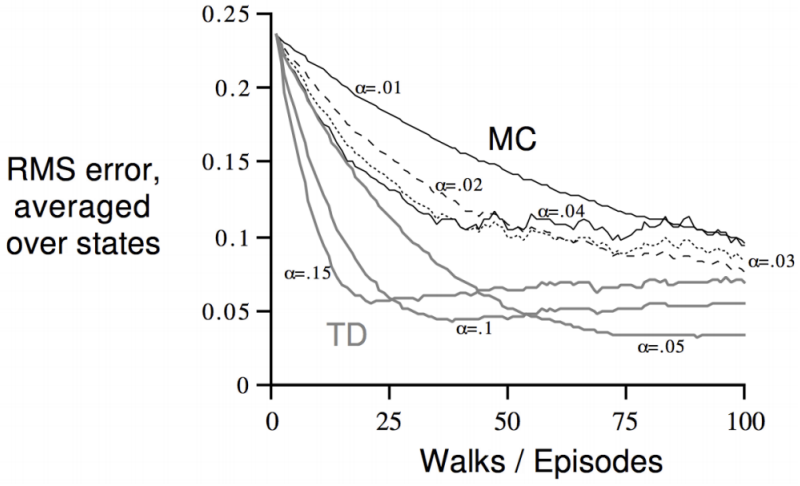
rewards have been experienced in an episode. Therefore, TD has fewer steps between updates at which noise is introduced into the updated value function, therefore lower variance, therefore less likely to get unstable oscillations around v_π

4. **Faster convergence**: TD exploits the Markov property by not considering previous history of states, whereas MC averages across historic state visits, therefore TD tends to be more time-efficient by converging on value function faster than MC

- **TD(0) Policy Evaluation**: Simplest TD RL algorithm. Has no weighting function λ (i.e. $\lambda = 0$, see later). As before, goal is to learn v_π (value function that correctly evaluates policy π) from episodes of experience under policy π. Recall that in Eq. 4.11, we defined the MC update rule as tending towards the *actual* return G_t. With TD policy evaluation, the update rule for the value function tends towards the **estimated** return $R_{t+1} + \gamma V(S_{t+1})$, known as the **TD target**:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{4.12}$$

  - $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is known as the **TD error**, which we want to minimise. When it is 0, $V(S_t)$ will have converged on v_π
  - Again, by law of large numbers, as no. episodes experienced by TD tends to ∞, will converge on v_π
  - Can see that learning curve comparison that TD converges much faster than MC since uses bootstrapping therefore doesn't keep carrying irrelevant information through episodes (here the RMS error is the error between actual v_π found by dynamic programming and the $v(s)$ estimated by TD & MC learning; recall that TD & MC learning are *approximate* DP methods). Note that as learning rate α is decreased, convergence is slower but can also lead to lower converged RMS and therefore a better converged v_π solution to the prediction task:

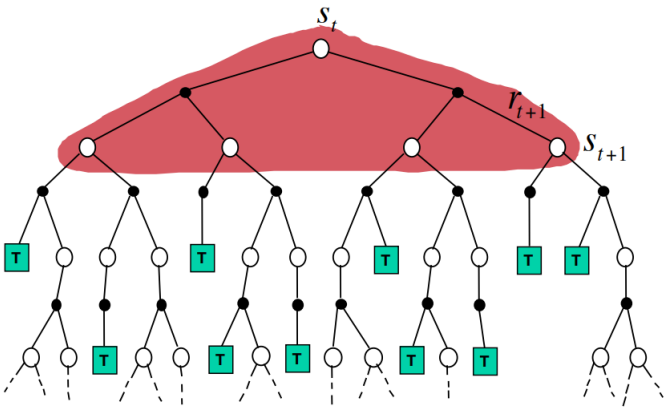


  - TD(0) converges on a solution that best fits the MDP data, whereas MC converges to the solution with the minimum MSE w.r.t. the observed returns, which is a slight difference that can sometimes mean that MC & TD methods converge on different v_π for the same problem, both of which are correct
  - Backup diagram for TD(0) (i.e. only does a 1-step look-ahead):

- **Diagrammatic comparison of DP, MC & TD value function backups to evaluate policy** π: Consider a backup diagram/tree of events, where states are represented with hollow vertices, actions with black vertices, and state-action-state transitions are represented with edges. The initial state is s_t. At s_t, can select 1 of 2 possible actions, a_1 or a_2. Selecting either one of these actions will take you to 1 of the 2 possible next states, state s_{t+1}, where the probability of going to each state given chosen action and current state is defined by the environment's/MDP's transition probability matrix $\mathcal{P}$. This tree of states, actions etc. continues until episode termination. Each of the algorithms will repeat the following processes until their $v(s)$ converges, at which point they will have converged on v_π:

  1. **Dynamic Programming**: Dynamic programming knows the MDP fully, therefore internally does a 1-step look-ahead for all possible states & actions following policy π. It sums the expected values for each new state & weights them by their probability of being transitioned to given current state and respective chosen action. It will then do a backup to update its $v(s)$

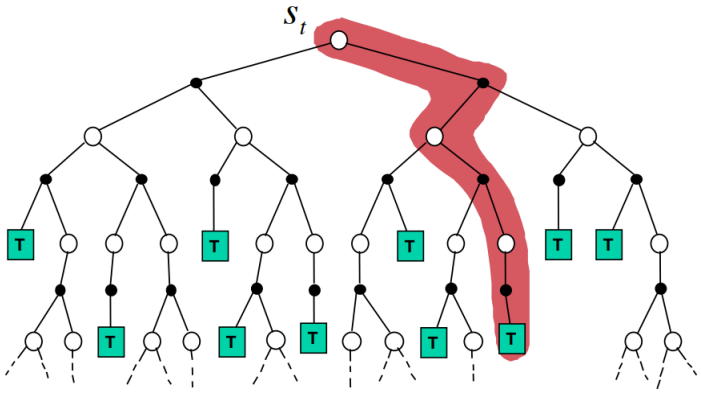
$$V(S_t) \leftarrow E_\pi(R_{t+1} + \gamma V(S_{t+1})) \tag{4.13}$$



  2. **Monte Carlo Reinforcement Learning**: MC RL updates $v(s)$ by looking at the sampled episode from start state s_t all the way until the terminal state (i.e. end of

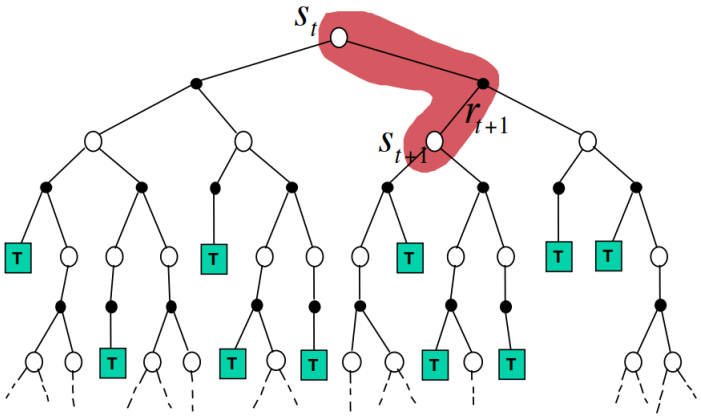
the episode) following policy π

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{4.14}$$

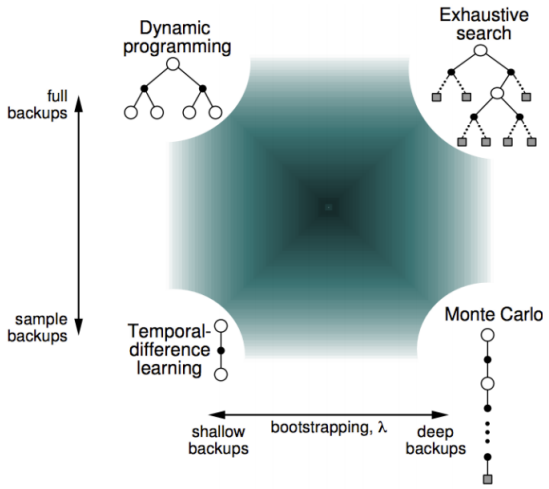


3. **Temporal Difference Reinforcement Learning**: TD RL updates $v(s)$ by sampling from start state s_t to the next state s_{t+1} following policy π

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1} - V(S_t)) \tag{4.15}$$



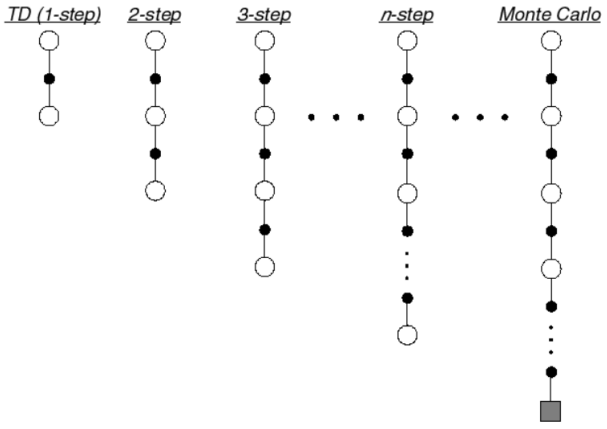
- **A unified view of RL**: The above information can be neatly summarised into a single diagram comparing the key paradigms of sequential decision making & RL:



- One question might be: Can we combine the benefits of Monte Carlo learning & TD(0) learning that we've seen so far? The answer is yes; with **TD(λ) learning**

4.4. TD(λ) Learning

- **n-step TD learning**:
  - So far, we have considered TD learning where we are looking $n = 1$ step into the future before updating $v(s)$. This is as opposed to MC learning, which looks at all the steps to the end of the episode before updating $v(s)$
  - **n-step TD learning** is where we look > 1 step into the future before updating $v(s)$
  - Obviously, as n tends to ∞ (or just to the number of steps in the episode), TD learning becomes MC learning, since it will be using the *actual* return to do its updates rather than the *estimated* return



- n **step return**:

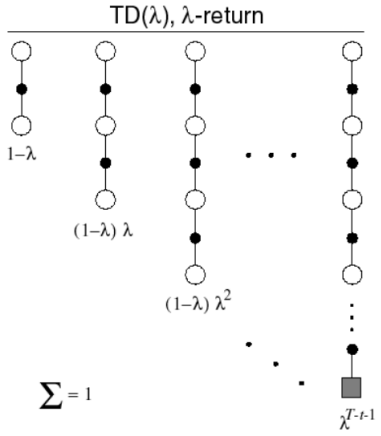
$$G_t^n = R_{t_1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t_n}) \tag{4.16}$$

- **n-step TD learning update rule**:

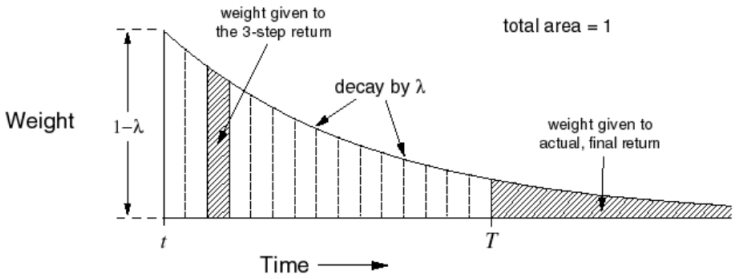
$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^n - V(S_t)) \tag{4.17}$$

- But, how do we choose which value of n is best? The answer is we don't have to. Instead, we can use multiple TD algorithms with different values of n and combine the information from each algorithm (which effectively combines the look-ahead information from different time steps) using **TD(λ) learning**

- **TD(λ) learning**: A method for combining information from different time steps (i.e. from different TD algorithms with different values of n) so that the learning process looks further into the future and therefore learns from expected returns that are closer to the actual returns, whilst still using bootstrapping so can continue to learn efficiently and handle non-episodic tasks
  - Works by defining a λ **weighting** (where $\lambda \in [0, 1]$) and a λ-**return** G_t^λ, where G_t^λ is the λ-weighted average return across all n step returns G_t^n. The weights of the average return get less by factor of λ each n^{th} step into the future:



  - I.e. weighting factor scales with time:



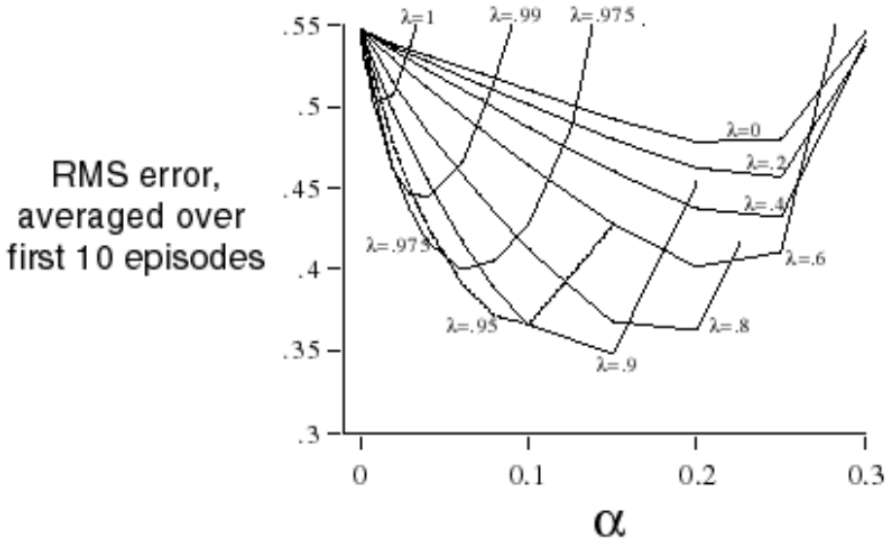
- **Weighted return** across all n-steps:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^n \tag{4.18}$$

- **Forward-view TD(λ)** update rule:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t)) \tag{4.19}$$

- Can tune λ weighting to balance trade-off between bootstrapping & looking ahead into future (i.e. trade-off between benefits of MC & TD learning):



- When $\lambda = 0$, only the current state is used for the update (i.e. what we saw at start of lecture). This is the TD(0) update in Eq. 4.15. When $\lambda \neq 0$, TD(λ) follows update rule in Eq. 4.19. As λ increases, TD waits more & more time steps before backing up to update $v(s)$. When $\lambda = 1$, TD waits until the end of the episode before update, & is equivalent to MC learning and therefore won't be able to handle non-episodic tasks. I.e. **TD(1) is equivalent to every-visit MC learning**.

- **Offline TD updates**: Where $v(s)$ updates are accumulated at each step within the episode, but only applied in batch at the **end** of the episode.
- **Online TD updates**: Where $v(s)$ updates are applied online at each step **during** the episode

5. Model-Free Control

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

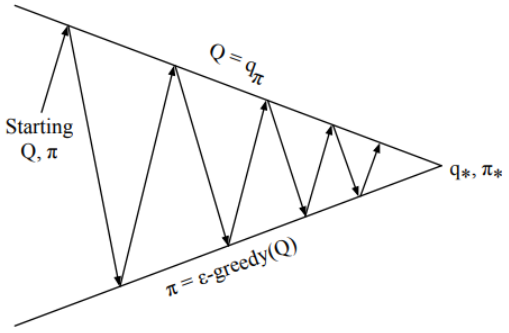
5.1. What is Model-Free Control?

- Previous lecture looked at model-free prediction i.e. how to find the value function v_π which correctly evaluates policy π without any initial information about the MDP
- Now we are considering model-free **control** i.e. how to find the *optimum* value function and *optimal* policy without initial knowledge of the MDP
- This lecture is what the previous 4 lectures have built towards. We will now see how we can put an RL agent into an unknown environment and let it **learn** how to maximise its reward in the environment
- Model-free control is good for solving problems where either (i) MDP model is unknown therefore can't use DP, but experience can be sampled using MC or TD, or (ii) MDP is known, but state-action space is too big to use DP, therefore must sample experiences using MC or TD
- There are 2 approaches to learning a solution to the control problem:
  - **On-policy learning**: Learning with your own **target policy** π. The experiences you learn from are generated by following your own policy π. As you follow your own policy, you evaluate and update it.
  - **Off-policy learning**: Learning with 'someone else's' **behaviour policy** μ. The experiences you learn from are generated by following a behaviour policy μ. You observe the return(s) of following μ and use return(s) to evaluate and update your own target policy π

5.2. On-Policy Learning

5.2.1. On-Policy Monte Carlo Learning

- Saw previously in lecture 3 how **generalised policy iteration** works to find the optimum value function v_* and the optimal policy π_*. We looked specifically at using **dynamic programming** with **iterative policy evaluation** as the *policy evaluation* algorithm to estimate v_π and **greedy policy improvement** as the *policy improvement* algorithm to generate $\pi' \geq \pi$
- We now want to consider how we can do **model-free** policy iteration with **MC learning**
- **Policy iteration for MC learning**:
  - If we use the **state-value function** $v(s)$ for our *policy evaluation* as we did in lecture 3 with DP, we do not take an action a, we merely follow π without explicitly taking actions. As such, to do policy evaluation with $v(s)$, we need a perfect model of the MDP from the beginning, therefore we cannot use $v(s)$ for model-free MC policy iteration
  - Instead, we can use the **action-value function** q_π **(s,a)** for our policy evaluation, which evaluates the explicit action taken and therefore does not require perfect knowledge of the MDP.
  - If we use the **greedy policy improvement** algorithm for our *policy improvement* as we did in lecture 3, we won't explore enough therefore we won't find enough new information out about the unknown MDP environment and thus won't ever fully understand MDP or maximise reward.
  - Instead, we can use the ϵ**-greedy action selection** algorithm for our policy improvement:
    - Method to ensure that don't always follow a greedy policy (since this would mean we never explore and learn about the initially unknown MDP)
    - All m possible actions have a non-zero probability of being selected, with probability $1 - \epsilon$ of choosing the greedy action and probability ϵ of choosing a random action
    - This is a simple way to guarantee that the value function you start with is always better than the value function you end up with, since for any ϵ-greedy policy π, the ϵ-greedy policy π' with respect to q_π is an improvement such that $v_{\pi'} \geq v_\pi(s)$
    - When $\epsilon = 1$, only random actions are chosen (pure exploration), and when $\epsilon = 0$, only greedy actions are chosen (pure exploitation). As will see later, can decay ϵ to 0 as converge on optimal policy π_*
  - MC policy iteration:



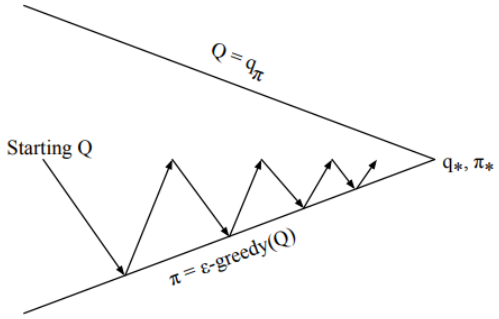
Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$
Policy improvement ϵ-greedy policy improvement

  - In the above diagram, Q is a big table of state-action pair value estimates. These Q-values are *estimated* total rewards you will get between now & end of game in your current state following policy π. When the Q table is converged, $Q \approx q_\pi$. In the control problem, when converged, the learned Q-table is a direct approximate of q_* (recall that model-free MC & TD methods are *approximate* TD methods in that they find approximate value functions/solutions).

- **Greedy in the Limit with Infinite Exploration (GLIE)**: A method for using the ϵ-greedy algorithm for policy improvement whilst balances exploration with exploitation as policy is improved by decaying ϵ to 0 as converge on π_*
  - For an ϵ-greedy algorithm to be GLIE, must have 2 properties:
    1. All state-action pairs must be explored infinitely many times across $k \longrightarrow \infty$ steps
    $$\lim_{k \longrightarrow \infty} N_k(s, a) = \infty \tag{5.1}$$
    2. The policy must converge on a greedy policy (i.e. a policy that always chooses to maximise its reward, there ϵ must eventually decay to 0)
    $$\lim_{k \longrightarrow \infty} \pi_k(a|s) = 1(a = \text{argmax} Q_k(s, a')) \tag{5.2}$$

5.2.2. On-Policy Temporal Difference Learning

- As discussed in lecture 4, unlike MC learning which can only update at the end of each episode, TD learning can update after each time step, which leads to lower variance etc.
- **Policy iteration for TD learning**:
  - As with MC policy improvement, **TD policy iteration** considers the **state-value function** $q_\pi(s, a)$ when doing *policy evaluation* and the **ϵ-greedy** algorithm when doing *policy improvement*, only now these are done after every time step rather than every episode:



Every time-step:
Policy evaluation Sarsa, $Q \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

  - Again, Q is a table of state-action pair value estimates which, when converged, $Q \approx q_\pi$
  - On-policy TD learning uses the **Sarsa** algorithm for *policy evaluation*

- **Sarsa Algorithm**: An algorithm used for evaluating the TD algorithm's policy π (therefore is an **on-policy** algorithm) i.e. it converges on the value function $q_\pi(s, a)$ that correctly evaluates policy π. It is an *update pattern* that we follow for updating our Q-table (which is our action-value function $q(s, a)$), and it stands for **State-Action-Reward-State-Action** (i.e. the pattern that we follow to update Q). Steps:

  1. Initialise a Q-table f 0s, where have a 0 value assigned to each possible state-action pair
  $$Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{5.3}$$
  For each episode:
  2. Choose action A from list of possible actions in current state S using policy derived from Q (e.g. ϵ-greedy)
  For each step of the episode, until S is a terminal state:
  3. Take action A, observe returned reward R and new state S'
  4. Choose new action A' from list of possible actions in state S' using policy derived from Q (e.g. ϵ-greedy)
  5. Update Q-table according to:
  $$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma(S', A') - Q(S, A)] \tag{5.4}$$
  6. Update current state and action:
  $$S \leftarrow S'; A \leftarrow A' \tag{5.5}$$

- The above Sarsa algorithm is equivalent to:

> Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
> Repeat (for each episode):
> Initialize S
> Choose A from S using policy derived from Q (e.g., ε-greedy)
> Repeat (for each step of episode):
> Take action A, observe R, S'
> Choose A' from S' using policy derived from Q (e.g., ε-greedy)
> $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
> $S \leftarrow S'$; $A \leftarrow A'$;
> until S is terminal

- Sarsa is guaranteed to converge on the optimal action-value function q_* if:
  1. The policies are GLIE i.e. ϵ tends to 0
  2. The step sizes α_t get incrementally smaller with time such that they eventually don't change the Q-table at all
- Just as with TD learning, can also use ***n*-step Sarsa** where look n steps into the future. As with TD, do this with **Sarsa(λ)** algorithm, where $\lambda \in [0, 1]$ and determines how far we look into the future when updating our Q-table (higher λ looks further into future)

5.3. Temporal Difference Off-Policy Learning

- **Off-policy** learning updates your policy by evaluating a different policy and subsequently updating your own
- Because MC learning is too high variance since it only updates after every episode rather than after every time step, in practice, off-policy MC learning does not work, therefore when referring to off-policy learning, the method is always a TD learning method
- **Advantages** of off-policy learning vs. on-policy learning:
  1. More efficiently use data since re-uses experiences generated from old policies
  2. Learns an optimal policy by updating target policy π_* whilst simultaneously exploring by observing the behaviour policy μ, therefore get best of both exploration and exploitation
  3. Can simultaneously learn about many different behaviour policies μ whilst improving your own target policy π, therefore get good efficient level of exploration
  4. Behaviour policy μ being observed can be that of a human or another agent, therefore may be more likely to learn e.g. a human-superior policy
- 2 methods for off-policy learning:

1. **Importance sampling**: A method for estimating the expected values of a particular distribution using samples that were generated from a different distribution. In the case of RL, use importance sampling to estimate the value function that correctly evaluates policy π by using samples collected previously from an old policy π'
   - Observe behaviour policy μ and use the returns to estimate the **TD targets** (the estimated return $R_{t+1} + \gamma V(S_{t+1})$) for μ
   - Then, use **importance sampling** to weight the estimated TF target and apply this weighted TD target to evaluate your own target policy π
   - Importance sampling update rule to update value function:

$$V(S_t) \leftarrow V(S_t) + \alpha\left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))\right) \tag{5.6}$$

2. **Q-Learning**: Rather than using importance sampling to estimate the expected TD target using the behaviour policy μ, Q-learning considers the action-values in $Q(s, a)$
   - Since this is off-policy, the next action that's actually taken is still chosen by the behaviour policy i.e. $A_{t+1}\ \mu(|S_t)$, however we also use our target policy to consider an alternative *successor* action A' i.e. $A'\ \pi(|S_t)$. The Q-table is then updated towards the value of the alternative action according to the Q-learning update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t_1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)) \tag{5.7}$$

     Usually when we do Q-learning, we make the target policy π greedy (i.e. chooses highest action-value in Q) and the behaviour policy μ ϵ-greedy

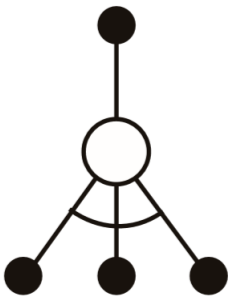
     The full Q-learning control algorithm update rule is:

$$Q(S, A) \leftarrow Q(S, A) + \alpha\left(R + \gamma \max_{a'} Q(S', a') - Q(S, A)\right) \tag{5.8}$$

   - Q-learning algorithm for off-policy control:

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$;
  until S is terminal

  - Q-learning makes better use of the Q-values available than importance sampling and is therefore more efficient
  - Q-learning backup diagram (i.e. follow behaviour policy action to new state, then choose action from new state that maximises value (i.e. following greedy target policy)):



- Comparison of 1-step backup diagrams for DP (which uses full backups i.e. considers every possible state-action pair) and TD (which uses sample backups i.e. only considers sample of possibilities):

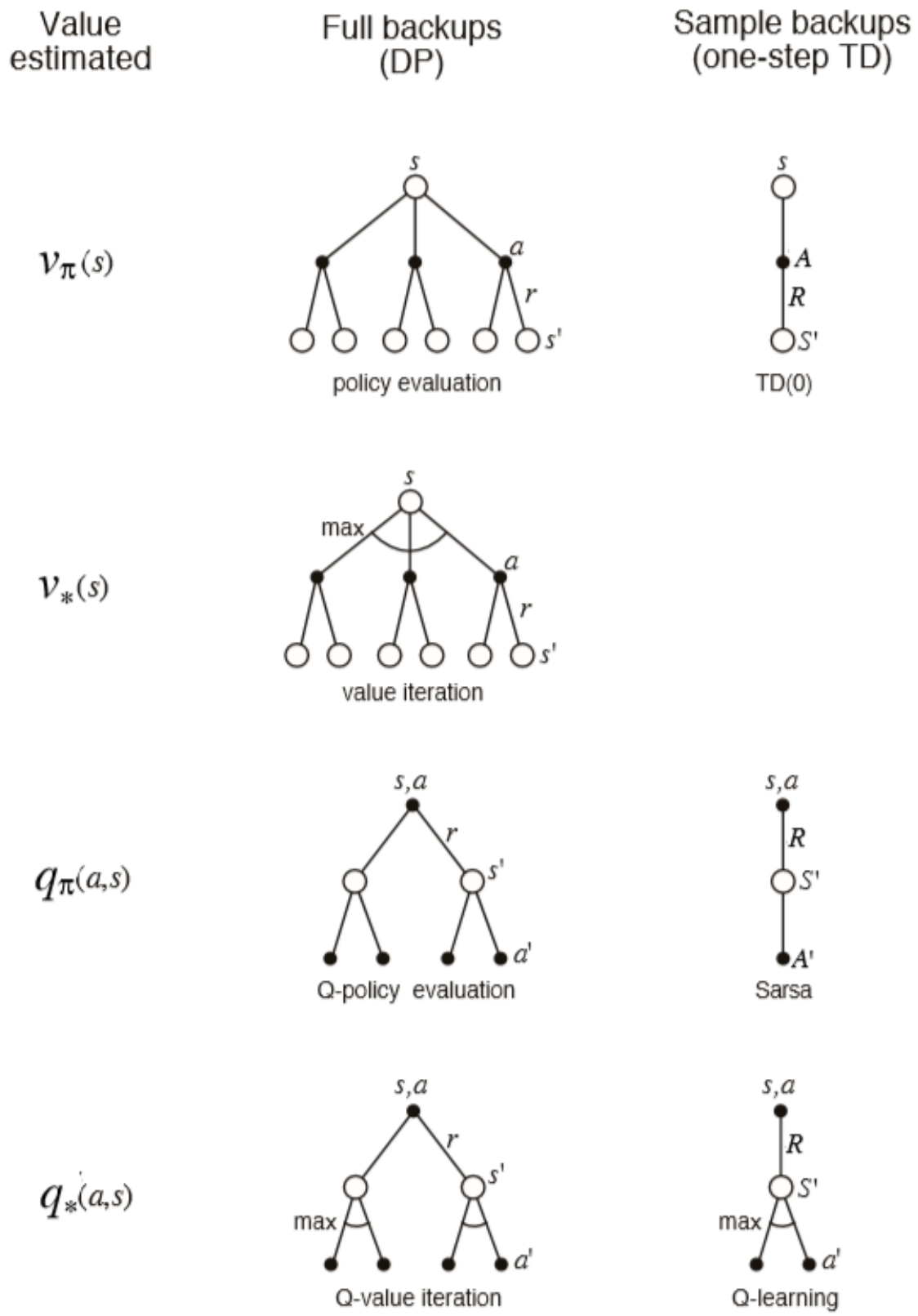


Figure 8.12: The one-step backups.

Part II.

Reinforcement Learning in Practice

6. Value Function Approximation

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

6.1. What is Value Function Approximation?

- So far, have considered RL learning process as learning how to update & use a **look-up table** (e.g. Q-table of state-action value pairs, or V table of state values)
- However, look-up tables are memory & data inefficient, often not possible to create for real problems, and even if they were possible to create would take too long to look-up a single value. E.g. blackgammon has 10^{20} states, Go has 10^{170} states, and helicopters (with a continuous state-action space) have an infinite number of states.
- **Function approximator (FA)**: A function used to approximate a look-up table/true value function. Can **generalise** to unseen states & state-action pairs
  - Different kinds of function approximators e.g.:
    - Linear
    - Non-linear (NNs fall under this category)
    - Decision tree
    - Nearest Neighbour
    - Fourier/wavelet basis
  - FAs defined by set of **parameters**/weights $\boldsymbol{w}$. Use RL methods (MC or TD learning) to optimise $\boldsymbol{w}$ such that function approximator converges on a function that sufficiently approximates the look-up table/true value function it is approximating
  - 3 value function FA architectures:
    1. $\hat{v}(s, \boldsymbol{w})$: Takes state s and learns set of parameters $\boldsymbol{w}$ that accurately estimate a value for how good state s is. When converged, if have correct set of parameters,

FA will approximate the true value function:

$$\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s) \tag{6.1}$$

2. $\hat{q}(s, a, \boldsymbol{w})$: Takes state s and action a and estimates value. When trained:

$$\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a) \tag{6.2}$$

3. $\hat{q}(s, a_1, \boldsymbol{w}), ..., \hat{q}(s, a_m, \boldsymbol{w}$: Learns to approximate value of *all* possible state-action pairs for m actions. Is possible for scenarios where have low number of actions (e.g. Atari games with e.g. 4 actions), but not where have very large action space

– Linear & non-linear FAs (e.g. NNs) are advantageous because they are **differentiable** (i.e. can calculate a gradient/rate of change of their parameters w.r.t. e.g. some other parameter (e.g. loss)), which makes optimisation easier since can change parameters $\boldsymbol{w}$ such that change in gradients are favourable

– 2 approaches for doing RL at scale with differentiable linear & non-linear FAs to approximate the value function:

1. **Incremental methods**: Incrementally update value function approximator **online** (i.e. after each step in episode)
2. **Batch methods**: Look at experiences so far as a single 'batch' and use to update value function in one go

6.2. Incremental Methods

- All updates done online i.e. after each time step/new experience

6.2.1. Prediction

- First will look at prediction problem of finding $v_\pi(s)$
- Work by defining a cost function J (e.g. mean-squared error) which evaluates expected difference between value function approximator $\hat{v}$ and the true value function v_π:

$$J(\boldsymbol{w}) = E_\pi\left[\left(v_\pi(S) - \hat{v}(S, \boldsymbol{w})\right)^2\right] \tag{6.3}$$

- Usually then use some optimisation algorithm (e.g. gradient descent, see Deep Learning parts) which optimises FA by updating parameters $\boldsymbol{w}$ such that cost is minimised
- **Feature vector** $x(S)$: Vector of values ('features') describing/representing state S. A feature vector might describe e.g. distance of robot from landmarks, trends in stock markets, piece configurations in chess, etc.

- Can then pass feature vector through value function approximator (e.g. a linear 'weighted summation' FA) where use weights $\boldsymbol{w}$ to perform transformation on features & output e.g. a single value (in this case as a result of a weighted sum) estimating the value of state S (which is represented by the feature vector). E.g. for each j^{th} feature in feature vector with n features:

$$\hat{v}(S, \boldsymbol{w}) = \boldsymbol{x}(S)^{\mathrm{T}} \cdot \boldsymbol{w} = \sum_{j=1}^{n} \boldsymbol{x}_j(S) \boldsymbol{w}_j \tag{6.4}$$

- For linear FA, update rule (i.e. how we change parameters $\boldsymbol{w}$ after each step such that function approximator is optimised) is simple (where α is the **learning rate** which is a coefficient determining how large we make our updates after each step):

$$\Delta \boldsymbol{w} = \alpha \big(v_\pi(S) - \hat{v}(S, \boldsymbol{w})\big) \cdot \boldsymbol{x}(S) \tag{6.5}$$

- Problem: So far have assumed we know $v_\pi(s)$ (i.e. a ground truth/label/supervisor) when calculating cost to perform updates to $\boldsymbol{w}$ & optimise FA, but for model-free RL problems we do not know true value function $v_\pi(s)$ or a supervisor therefore can't do this. Therefore, must use TD & MC RL to give us a relative target (instead of true value function $v_\pi(s)$) when evaluating cost with which we update $\boldsymbol{w}$
- Solution: Use RL approaches we're familiar with. I.e. the model-free RL approaches (MC or TD learning) learn to provide some ground-truth value as a target for the FA to update/optimise its parameters $\boldsymbol{w}$ towards:
  - **MC Learning**: Substitute return G_t as the target/ground truth value in our equations. In this sense, treating G_t as the target labelled data so that FA learns to approximate the RL target G_t; this is a 'supervised learning' perspective of what we're doing. Example of linear update rule:

$$\Delta \boldsymbol{w} = \alpha \big(G_t - \hat{v}(S_t, \boldsymbol{w})\big) \cdot \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}) \tag{6.6}$$

  - **TD(0) Learning**: Substitute the TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$ as the target. Uses bootstrapping therefore may be faster than MC learning, but is biased (i.e. is just a sample of the true value of $v_\pi(s)$) therefore is not as good a 'ground truth' as provided by MC learning. E.g. linear update rule:

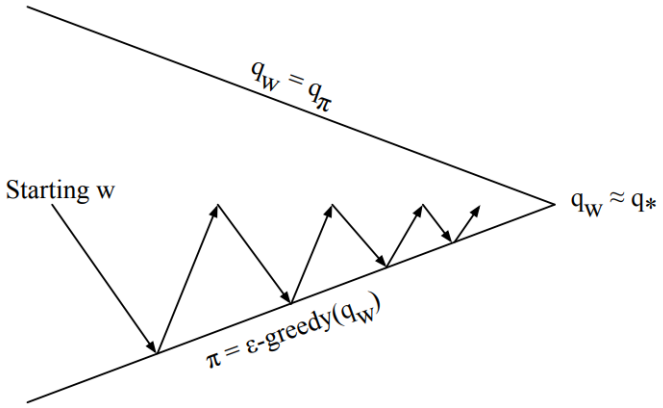
$$\Delta \boldsymbol{w} = \alpha \big(R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})\big) \cdot \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}) \tag{6.7}$$

  - **TD($\boldsymbol{\lambda}$) Learning**: Use λ-return G_t^λ as biased sample of true value $v_\pi(s)$. Unlike TD(0), looks further into future, therefore can reduce bias of target & get more reliable convergence of FA

$$\Delta \boldsymbol{w} = \alpha \big(G_t^\lambda - \hat{v}(S_t, \boldsymbol{w})\big) \cdot \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}) \tag{6.8}$$

6.2.2. Control

- Will now look at control problem of finding optimum q_* & π_*
- Do same as before, but now for action-value function q rather than value function v
- Our iterative diagram for solving the control problem now becomes:



Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

- I.e. just as before, if assume with have the ground truth $q_\pi(S, A)$, our cost function might be:

$$J(\boldsymbol{w}) = E_\pi\left[\left(q_\pi(S, A) - \hat{q}(S, A\boldsymbol{w})\right)^2\right] \tag{6.9}$$

And our update rule:

$$\Delta \boldsymbol{w} = \alpha\left(q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{w})\right) \cdot \boldsymbol{x}(S, A) \tag{6.10}$$

Where $\boldsymbol{x}(S, A)$ is our feature vector but now the features represent state-action pairs rather than just states

- Just as before, can use MC or TD learning to substitute targets as the ground-truth values & make the above practical:

1. **MC Learning**:
$$\Delta \boldsymbol{w} = \alpha\left(G_t - \hat{q}(S, A, \boldsymbol{w})\right) \cdot \boldsymbol{x}(S, A) \tag{6.11}$$
2. **TD(0) Learning**:
$$\Delta \boldsymbol{w} = \alpha\left(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}) - \hat{q}(S, A, \boldsymbol{w})\right) \cdot \boldsymbol{x}(S, A) \tag{6.12}$$
3. **TD(λ) Learning**:
$$\Delta \boldsymbol{w} = \alpha\left(q_t^\lambda - \hat{q}(S, A, \boldsymbol{w})\right) \cdot \boldsymbol{x}(S, A) \tag{6.13}$$

- N.B. In above examples we've used linear function approximator, but this could be anything e.g. non-linear NN FA, & same fundamental methods would apply.
- For many RL problems where FA is involved, find that bootstrapping helps & that if don't use bootstrapping (i.e. if use MC learning), get too high variance & therefore worse performance. TD(0) (update after each time step) still does better than MC, but optimum number of steps to look into future is usually $0 < \lambda_{\text{opt}} < 1$, therefore λ (i.e. parameter to control amount of bootstrapping done with TD(λ)) is a parameter to tune
- **Con** of non-linear FAs (such as NNs) as opposed to linear FAs is that non-linear FAs have much less stable convergence, therefore can't use TD learning with e.g. NNs as often struggles to converge and therefore must use on-policy MC learning. DeepMind 2015 Atari Games paper was one of first to overcome these convergence/stability issues so that could use TD learning *and* NNs to get much more powerful RL agent (see later)

6.3. Batch Methods

- So far have considered performing incremental updates (i.e. update FA's $\boldsymbol{w}$ params after each new time step/experience). Once experience is used, it is discarded, therefore is not data-efficient
- **Batch methods** look across multiple experiences (a 'batch' of experiences i.e. the 'training data') and combine them to update FA's $\boldsymbol{w}$ params such that find FA that **best fits** the batch

6.3.1. Prediction

Least Squares Algorithm for Prediction

- Use **least squares** algorithm to find vector $\boldsymbol{w}$ that minimises **sum-squared error** between FA $\hat{v}(s_t, \boldsymbol{w})$ & the ground-truth true value function v_t^π. I.e. find $\boldsymbol{w}$ that minimises least squares error over all previous experiences/previous time steps experienced, where sum-squared error is given by:

$$LS(\boldsymbol{w}) = \sum_{t=1}^{T} \left(v_t^\pi - \hat{v}(s_t, \boldsymbol{w})\right)^2 \tag{6.14}$$

- To find $\boldsymbol{w}$ that minimises LS, use **experience replay**
- **Experience replay**: Given an **experience** (i.e. training dataset) D made up of $\langle$state, value$\rangle$ pairs, $D = \left\{\langle s_1, v_1^\pi\rangle, ..., \langle s_T, v_T^\pi\rangle\right\}$, repeat:
  1. Randomly sample a state-true value pair from experience dataset D:

$$\langle s, v^\pi \rangle \ D \tag{6.15}$$

2. Use the sampled experience (a state and a true value) and apply stochastic gradient descent to update the FA parameters towards this true value:

$$\Delta \boldsymbol{w} = \alpha\big(v^{\pi} - \hat{v}(s, \boldsymbol{w})\big) \cdot \nabla_{\boldsymbol{w}} \hat{v}(s, \boldsymbol{w}) \tag{6.16}$$

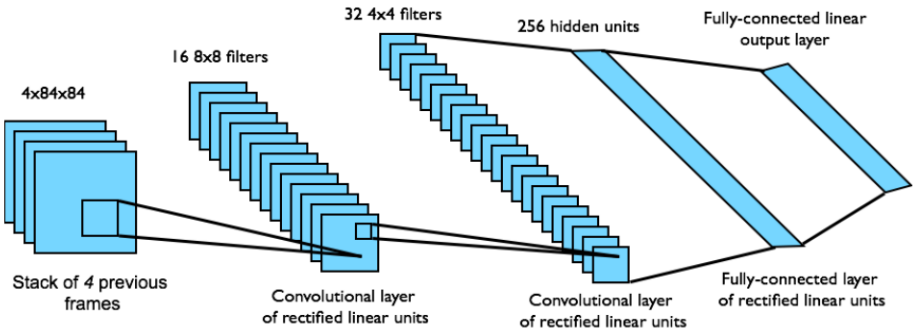
Repeat until $\boldsymbol{w}$ converges on least-squares solution (i.e. set of params that minimise sum-squared error):

$$\boldsymbol{w}^{\pi} = \text{argmin}_{\boldsymbol{w}} LS(\boldsymbol{w}) \tag{6.17}$$

- Advantages of experience replay:
  1. Random sampling **breaks correlations** between samples, therefore learned FA is more robust
  2. Re-use experiences, therefore more **sample-efficient**

Experience Replay in Deep Q-Networks (DQN)

- Was approach used by DeepMind in 2015 paper to play Atari games to super-human level. Was first paper able to use a TD method (i.e. Q-learning) with non-linear FA (a NN) without stability & convergence problems. Did this by using experience replay
- Process:
  1. Have NN $\hat{q}(s, a, \boldsymbol{w})$, termed the **Q-network**, which approximates q_{π}
  2. Use $\hat{q}(s, a, \boldsymbol{w})$ to choose an action a_t in state s_t according to an ϵ-greedy policy
  3. Observe the returned reward r_{t+1} and the new state s_{t+1} as a result of taking action a_t in state s_t, and store this experience $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ in experience replay memory D
  4. Randomly sample a **mini-batch** (e.g. 64) of experiences/transitions $\langle s, a, r, s' \rangle$ from stored experiences D (which has e.g. millions of saved experiences)
  5. Use TD Q-learning to compute the TD targets (the **Q-targets**) for each of the sampled experiences in the mini-batch
  6. Use (variant of) stochastic gradient descent to minimise mean-squared error between NN Q-network $\hat{q}(s, a, \boldsymbol{w})$ & the Q-targets by updating params of Q-network towards Q-target
- DQN architecture used by DeepMind took in 4 stacked game frame images as the input state s & applied convolutional neural network, with DQN outputting estimated values for each state-action pair (e.g. for given game might be e.g. 4 possible actions):

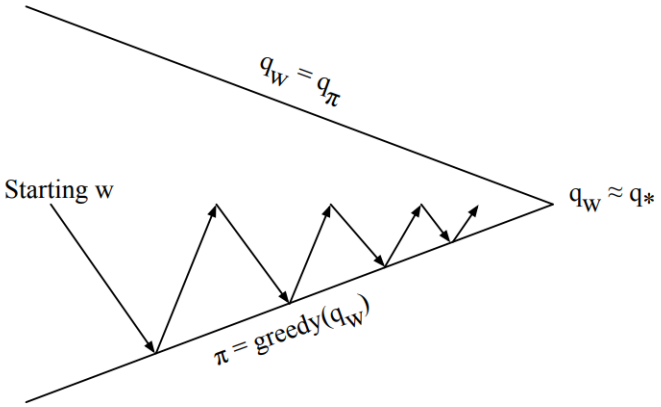


- Most approaches to RL now use batch methods as opposed to incremental methods due to more stable updates & convergence of $\boldsymbol{w}$ parameters of function approximator

6.3.2. Control

Least Squares Algorithm for Control

- Same principle as previously, but now finding/approximating q_* & v_* to solve control problem:



Policy evaluation Policy evaluation by least squares Q-learning
Policy improvement Greedy policy improvement

7. Policy Gradient

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/

7.1. What is Policy Gradient Reinforcement Learning?

- 3 classes of RL methods:
  1. **Value-based**: Learn a value function which *implicitly* defines a policy by following e.g. ϵ-greedy policy to select actions based on value function. Is what we've seen so far (e.g. Sarsa & Q-learning are value-based algorithms).

     Pros:

     - Tend to get larger updates of policy & therefore faster convergence on value function (for scenarios where have small action spaces)

     Cons:

     - Evaluating a policy (e.g. assigning an exact value to every move) is inefficient as often just want 'optimum' move rather than knowing value of all stats & actions
     - Need to find maximum value of all possible state-action pairs therefore can't apply to very large or continuous action spaces
     - Can never incorporate stochastic elements/randomness into agent policy (since will make deterministic choice of action with highest value according to value function), which is disadvantageous for certain scenarios (e.g. in rock-paper-scissors a random policy will beat any deterministic policy, e.g.2. for some partially-observable MDP (POMDP) problems such as Aliased Gridworld problem, agent benefits from some stochasticity in its policy)
  2. **Policy-based**: No value function. Instead, *explicitly* define a policy & learn best policy directly.

     Pros:

  - By only following policy gradient rather than value function gradient to implicitly define a policy, often get smaller step updates & smoother/more stable convergence
  - Don't need to explicitly choose max value action as do with value-based, therefore don't need top consider value of all possible actions, therefore can handle scenarios with very large or continuous (infinite) action spaces. This is number 1 reason people use policy-based RL
  - Capable of learning stochastic policies, which can be good for e.g. POMDPs

  Cons:

  - Smaller updates after each step make agent prone to converging to local rather than global minimum policy
  - Since policy updates at each step are small, although gives smooth convergence, convergence can be slow. Therefore for small-action space problems (e.g. Atari games), can be more beneficial to use value-based RL
  - Use MC methods therefore get reward at end of episode therefore have high variance (e.g. for Atari games, one episode might result in reward of 1,000, and the next a reward of 0), which can make learning difficult

3. **Actor-critic**: Explicitly define both the value function *and* the policy function, & learn to optimise them both. Tries to get best of value-based & policy-based approaches

- Up to now, have considered **value-based** RL approach to solving **control** problem as doing **policy iteration** where first find value function for a given policy & then use this to update the policy with e.g. ϵ-greedy strategy
- In previous chapter, did this by approximating either value function $V_\theta(s) \approx V^\pi(s)$ or action-value function $Q_\theta(s,a) \approx Q^\pi(s,a)$ by optimising parameters θ of the function approximator (N.B. In this chapter, θ parameters are equivalent to $\boldsymbol{w}$ parameters from previous chapter, but use θ when referring to parameters approximating policy, and $\boldsymbol{w}$ when referring to parameters approximating value function). Rather than directly defining a policy π, we just followed an ϵ-greedy policy to choose actions according to our converged-on value function; this is the principle of policy iteration
- **Policy gradient RL**: Use a function approximator to **directly** parameterise/approximate policy π & directly update the policy FA parameters by following the **policy gradient**. Therefore, FA will explicitly give the policy i.e. which action should take for given state. Use a FA rather than look-up table to can scale RL to large state-action spaces

7.1.1. Policy Objective Function

- Given a parameterised policy $\pi_\theta(s, a)$, want to find set of parameters θ that give optimum policy
- Therefore need an objective/cost function to evaluate a given set of policy parameters. 3 approaches:
  1. **Start value**: If start in state s_1, what is expected return from following policy π_θ to the end of the episode?
  $$J_1(\theta) = V^{\pi_\theta}(s_1) = E_{\pi_\theta}[v_1] \tag{7.1}$$
  Use for **episodic environments**
  2. **Average value**: Use for **continuous environments** where don't have explicit start state. Take a given state s, & multiply the **stationary distribution** $d^{\pi_\theta}(s)$ (the probability of ending up in state s when following policy π_θ in a Markov chain) by the value of that state s following π_θ & averaging this over all states:
  $$J_{\text{average}V}(\theta) = \sum_s d^{\pi_\theta}(s) \cdot V^{\pi_\theta}(s) \tag{7.2}$$
  3. **Average reward per time step**: Also use for **continuous environments**, but rather than average value over future time steps, only average over possible immediate rewards 1 step into future:
  $$J_{\text{average}R}(\theta) = \sum_s d^{\pi_\theta}(s) \cdot \sum_a \pi_\theta(s, a) \cdot R_s^a \tag{7.3}$$
- Policy gradient RL methods take one of above objective functions and finds set of parameters θ that maximise the objective function (if higher objective function output indicates higher reward) to find optimum policy. All policy gradient methods perform updates by moving the policy in a direction such that results in the reward/objective function being maximised. I.e. This is an optimisation problem. 2 categories of optimisation algorithms:
  1. **Gradient-free optimisation**:
     - Hill climbing
     - Simplex/amoeba/Nelder Mead
     - Evolutionary algorithms
     - Swarm intelligence algorithms
  2. **Gradient-based optimisation** (often easier & more efficent as just point to direction need to go to improve, but sometimes hard to find gradient signal for certain problems. In this chapter, will focus on gradient-based methods):
     - Gradient descent/ascent

  - Conjugate gradient
  - Quasi-newton

- 3 approaches to policy gradient RL:
  1. Finite-difference policy gradient (**policy-based**)
  2. Monte-Carlo policy gradient (**policy-based**)
  3. Actor-critic policy gradient (**actor-critic**)

7.2. Finite-Difference Policy Gradient

- The **numerical** approach to calculating gradient of a policy; is most simple but nieve approach. Is policy-based policy gradient i.e. only considers policy (no value function)
- To evaluate the gradient of a policy $\pi_\theta(s,a)$ with n dimensions, estimate gradient of objective function w.r.t. for each k^{th} dimension in θ. I.e. for $k \in [1,n]$:

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon \cdot u_k) - J(\theta)}{\epsilon} \tag{7.4}$$

  Where u_k is a unit vector with 1 in the k^{th} component of θ and 0 elsewhere
- Will always work, but for gradient with n dimensions, must do n gradient evaluations, therefore can't realistically scale to problems where θ has many parameters. Since NNs often have millions of parameters, finite-difference policy gradient methods are not scalable in practice

7.3. Monte-Carlo Policy Gradient

- Also policy based (i.e. only consider policy, no value function). Computes policy gradient **analytically** (rather than numerically as did with finite-difference)
- **Likelihood ratio**: A 'trick' used to get an analytical expression for a function by taking the log of the function. Allows us to compute the **expectation** of e.g. the policy gradient rather than having to compute the actual/**empirical** gradient numerically. Uses the identity:

$$\nabla_\theta(s,a) = \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a) \tag{7.5}$$

- In above identity, define the $\nabla_\theta \log \pi_\theta(s,a)$ term as the **score function**, which tells us which direction to move in order to increase something (i.e. how to change policy such that reward is increased)
- Score function looks different for different policies. 2 popular policy functions:

1. **Softmax policy**: Use for when have **discrete** set of actions. Is a probability distribution that gives probability of choosing action a in state s (i.e. is alternative to deterministic ϵ-greedy policy we've seen so far):

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^T} \tag{7.6}$$

   Score function:

$$\nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - E_{\pi_\theta}[\phi(s,a)] \tag{7.7}$$

   Can therefore adjust θ params such that score function (i.e. reward) is maximised

2. **Gaussian policy**: Use for when have **continuous** (i.e. infinite) set of actions. Is a normal/Gaussian probability distribution over all possible actions with mean $\mu(s)$ & standard deviation σ, and can parameterise the distribution & adjust such that score function is maximised:

$$\nabla_\theta \log \pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2} \tag{7.8}$$

   Where a is the action chosen, $\mu(s)$ is the distribution's mean action, $\phi(s)$ is the feature of state s, & σ^2 us the variance of the distribution

- **One-step MDP**: MDP where only take 1 step before episode terminates. Goal is to pick action that maximises 1-step reward $r = R_{s,a}$ when starting in state s.
- MC policy gradient for 1-step MDP: Want set of params θ that give policy that maximises 1-step reward.

  First, define an objective function (here, the expected reward by following policy π_θ):

$$J(\theta) = E_{\pi_\theta}[r] = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) R_{s,a} \tag{7.9}$$

  Then, find gradient of objective function, $\nabla_\theta J(\theta)$. so that know how to change policy params θ such that objective function $J(\theta)$ is maximised:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot r] \tag{7.10}$$

  (i.e. is just expectation under policy π_θ for score $\times$ reward)

- MC policy gradient for multi-step MDP: Do same, but now replace immediate reward r with long-term reward/return $Q^\pi(s,a)$. I.e. we are generalising the likelihood ratio approach to multi-step MDPs, giving the **policy gradient theorem**:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)] \tag{7.11}$$

  This policy gradient theorem forms the basis of policy gradient RL; tells us how to change θ of parameterised policy function such that long-term reward is maximised in a multi-step MDP

REINFORCE Algorithm

- REINFORCE is an old RL algorithm from the 90s, but is still popular. It is just Monte Carlo policy gradient RL (and is sometimes referred to as this). It is the most simple RL algorithm that makes use of the score function/the policy gradient theorem for policy gradient RL
- Is MC RL, therefore can only handle episodic tasks
- Works by sampling the expected Q values from $Q^{\pi_\theta}(s_t, a_t)$ to get a 'sampled' expected return v_t, which can then plug in as the return value in the policy gradient theorem to get the update rule for the θ parameters, which we perform using *stochastic gradient ascent* (i.e. want to maximise objective function/reward therefore use gradient ascent):

$$\Delta\theta_t = \alpha\nabla_\theta \log \pi_\theta(s_t, a_t) \cdot v_t \tag{7.12}$$

- I.e. At each step t, the θ params are updated a small amount in direction of the score function (using stochastic gradient ascent) until converge on $\pi_\theta \approx \pi_*$:

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(s_t, a_t)v_t$
    end for
  end for
  return θ
end function

- **Con** of policy-based methods such as reinforce: Although get smooth convergence, often have very small step updates to policy function, therefore need 100,000s of iterations even to solve simple problems. To reduce the number of iterations needed, instead of policy-based RL, recent movement has been towards **actor-critic** RL. Also have high variance due to high varying of rewards at end of MC learning episodes

7.4. Actor-Critic Policy Gradient

- New class of RL algorithms that are becoming increasingly popular. Incorporates idea from last chapter of using a function approximator to approximate the Q-targets/value function while using ideas from this chapter of updating the parameterised policy directly
- I.e. actor-critic maintain 2 sets of parameters/ 2 different function approximators:
  - **Critic**: Approximates & updates the action-value function parameters $\boldsymbol{w}$. I.e. just takes a policy π and evaluates it using e.g. MC policy evaluation, TD learning etc. as we've seen in previous chapters.

  - **Actor**: Approximates & updates the policy parameters $\boldsymbol{\theta}$. Does this using policy gradient theorem as before, but now using approximated value function $Q_{\boldsymbol{w}}(s,a)$ rather than true return v_t

- Here we are replacing the high-variance return v_t used in MC policy-based methods with a critic used to estimate the action-value function $Q_{\boldsymbol{w}}(s,a) \approx Q^{\pi_\theta}(s,a)$, which reduces the high variance updates and therefore allows for easier learning and larger steps to be taken when updating parameters, therefore can use fewer iterations to solve RL problems

  I.e. Using $Q_{\boldsymbol{w}}(s,a)$ gives an *approximate* policy gradient when plugged into the policy gradient theorem (rather than a *true* policy gradient as we got when used v_t). As before, updates to parameters of function approximators are done with stochastic gradient ascent:

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot Q_w(s,a)] \tag{7.13}$$

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) \cdot Q_w(s,a) \tag{7.14}$$

Q Actor-Critic (QAC) Algorithm

- An example of a simple actor-critic algorithm
- **Critic** updates $\boldsymbol{w}$ to approximate Q^π using TD(0) Q-learning (i.e. is an online algorithm as it updates after each time step in episode), and **actor** updates $\boldsymbol{\theta}$ to approximate π_* using policy gradient theorem
- At each time step, sample an action from actor's policy function π_θ, calculate the TD error δ for critic's estimated value before & after step, & then update both the actor's θ params to move closer to reward-maximising policy & the critic's w params to move closer to the 'true' value function:

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

7.4.1. Reducing Variance using a Baseline

- Can use a **baseline** to reduce variance in actor-critic parameter updates of the policy & the value function

- **Baseline function** $B(s)$: Some relative comparison used to 'rescale' parameter updates so don't e.g. evaluate a policy as 1,000 on one step & 999,999 on next, but rather e.g. +1 on one step and 0 on next. This reduces the variance/noise in the parameter updates which gives smoother, faster & more reliable convergence
- Baseline functions must have the property that operating them on the score function for the policy gradient theorem will not change the expectation value. For this to be true, the baseline function must only be a function of the state s, *not* of the state-action pair
- Often set the baseline function $B(s)$ as the state value function $v_{\pi_\theta}(s)$. This can be parameterised by the critic with parameters $\boldsymbol{v}$ such that $V_v(s) \approx V_{\pi_\theta}(s)$. Can then define an **advantage function** $A(s,a)$ using this approximate value function of our baseline, which is approximated by the critic as:

$$A(s,a) = Q_w(s,a) - V_v(s) \tag{7.15}$$

  Where a positive value returned by the advantage function indicates that the chosen action does better than the baseline suggesting the policy should move policy towards choosing that action in the given state, & a negative result indicates a worse performance than the baseline, indicating should move away from that policy. I.e. use of advantage function/baseline always pushes policy towards where will do better than 'usual'/the baseline, which **reduces variance**

- I.e. policy gradient theorem with advantage function:

$$\Delta_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot A^{\pi_\theta}(s,a)] \tag{7.16}$$

  Where we now have 3 sets of parameters ($\boldsymbol{w}$ and $\boldsymbol{v}$ for the critic, and $\boldsymbol{\theta}$ for the actor):

$$V_{\boldsymbol{v}}(s) \approx V^{\pi_\theta}(s) \tag{7.17}$$

$$Q_{\boldsymbol{w}}(s,a) \approx Q^{\pi_\theta}(s,a) \tag{7.18}$$

$$A(s,a) = Q_{\boldsymbol{w}}(s,a) - V_{\boldsymbol{v}}(s) \tag{7.19}$$

  Where $V_v(s)$ & $Q_w(s,a)$ are updated by e.g. TD learning

- Alternative method to above is to avoid needing parameters w & only use one set of params w; this is what is usually done in practice in research. Do this by replacing advantage function $A(s,a)$ with the critic's estimate of the TD-error δ^{π_θ} which (like the advantage function) can tell the agent how much better/worse than 'usual' a given action/policy is:

$$\Delta_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \cdot \delta^{\pi_\theta}] \tag{7.20}$$

  Where approximate TD error as:

$$\delta^{\pi_\theta} \approx \delta_v = r + \gamma V_v(s') - V_v(s) \tag{7.21}$$

I.e. now we only need one set of parameters v to approximate TD error (using state value function approximation $V_v(s)$) rather than also needing another set of parameters w to approximate $Q_w(s,a)$, which makes learning more simple for agent

- Just as we saw in previous chapters, can combine benefits of bootstrapping & looking ahead by using TD(λ) learning (which can also have this baseline/advantage function/TD error approach)

7.4.2. Natural Policy Gradient

- So far have considered **stochastic policies** where define policy as e.g. Gaussian distribution over all actions with a mean μ & a standard deviation σ. **Con**: Policy can be noisy and have non-zero chance of choosing sub-optimal action
- Solution: **natural policy gradient**; Use the 'limiting case' of the policy gradient (i.e. the most extreme policy gradient value) when considering a deterministic policy function with 0 noise (rather than a stochastic Gaussian function with non-zero standard deviation). Therefore, natural policy gradient methods *always* pick the mean μ of the distribution since don't have any standard deviation/noise across the possible actions, which results in much lower variance in policy updates & prevention of choosing sub-optimal actions
- I.e. this is a deterministic policy gradient method (no noise in policy function across actions) as opposed to a stochastic policy gradient method (has noise due to e.g. Gaussian distribution with non-zero standard deviation)
- Natural actor-critic methods scale very well to high-dimensional problems/continuous action spaces
- Summary of policy gradient methods we've seen this chapter:

  - The policy gradient has many equivalent forms

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ v_t\right] && \text{REINFORCE}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)\right] && \text{Q Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^w(s,a)\right] && \text{Advantage Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta\right] && \text{TD Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic}\\
G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic}
\end{aligned}
$$

  - Each leads a stochastic gradient ascent algorithm
  - Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$

]

8. Integrating Learning & Planning

- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2nd edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/
- Previous chapters have looked at how to use RL for model-free tasks
- Will now look at how to utilise environment model to integrate **learning** & **planning** into one architecture (e.g. Dyna, see later)
- Previously how to learn value function directly from experience (**value-based RL**), how to learn policy function directly from exp (**policy-based RL**), & how to learn both simultaneously from exp (**actor-critic RL**). These are all **model-free** methods. Will now look at how to learn the model (how one state transitions to next & what rewards are returned) directly from exp (**model-based RL**), & then use **planning** to use learned model to construct value function or policy. Finally, will look at **integrated architectures** (e.g. Dyna) which integrated model-free & model-based RL.

8.1. Model-Based Reinforcement Learning

- 3 broad themes in RL:
  - **Model-free RL**: No model. Learn value function and/or policy from real experience
  - **Model-based RL**: Learn model from real experience. Use learned model to plan value function and/or policy from simulated experience (using e.g. sample-based planning)

    Pros:

    * If can learn rules of environment (e.g. chess), can be more efficient than 'blindly' learning policy & value function in unknown environment
    * Can use model to reason about things agent does & doesn't know about, therefore can more efficiently explore areas of MDP that are unknown, resulting in more efficient learning & more robust RL agent

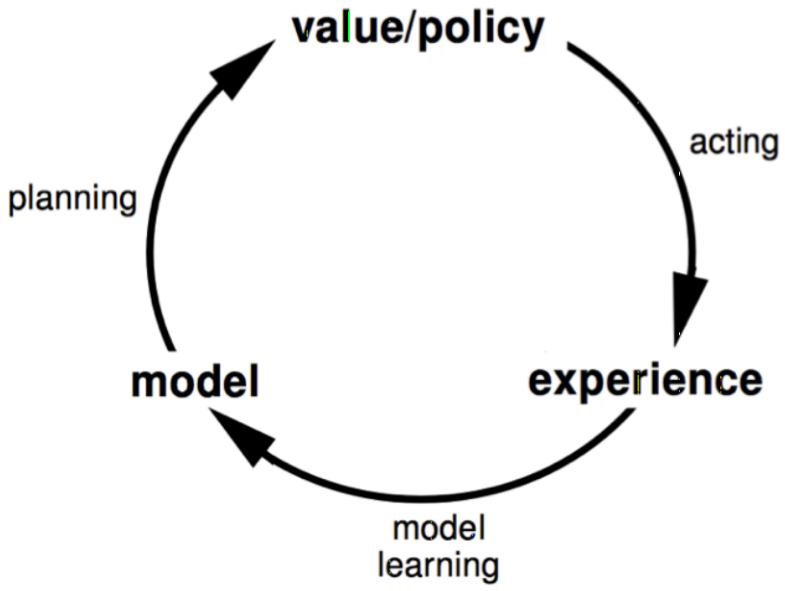
    * With learned model, can generate infinite experiences/training data

  Cons:

    * First learn model, then construct value function, therefore have 2 sources of error being compounded which may make learning harder

  – **Integrated architectures (e.g. Dyna)**: Combine model-free & model-based RL into single architecture to try get benefits of both. Learn model from real experience. Use simulated experience *and* real experience to do learning *and* planning for value function and/or policy

- Workflow of model-based RL:



8.1.1. Learning a Model

- **Model**: See definition of model in first chapter. Can parameterise/approximate model $\mathcal{M}$ with a function approximator with parameters v such that $\mathcal{M}_v = \langle \mathcal{P}_v, \mathcal{R}_v \rangle \approx \langle \mathcal{P}, \mathcal{R} \rangle$.

  Learning a model just means learning rules (e.g. in chess, learning concept of winning & losing, & learning which pieces can move where), *not* how to evaluate states or state-action pairs, therefore learning a model can be more simple than learning value function

- To learn model $\mathcal{M}_v$, consider as supervised learning problem where have experiences $\{S_1, A_1, R_2, ..., S_T\}$ which use to learn what states & rewards happen next etc. Learning which reward r will occur given state s & action a (i.e. learning state transition function $\mathcal{P}$) is a **regression** problem, & learning which next state s' will occur given state s &

action a (i.e. learning reward function $\mathcal{R}$) is a **density estimation** problem. Can adjust params v such that $\mathcal{M}_v$ fits $\langle \mathcal{P}, \mathcal{R} \rangle$ with minimum e.g. mean squared error

- Can use e.g. NN $\mathcal{M}_v$ to approximate model $\mathcal{M}$

8.1.2. Planning with a Model

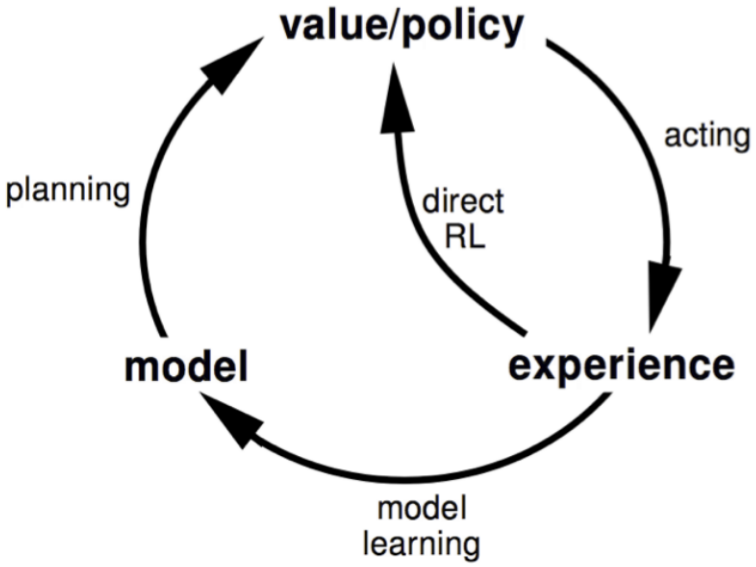
- With learned model/MDP representation $\mathcal{M}_v = \langle \mathcal{P}_v, \mathcal{R}_v \rangle$, can solve MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_v, \mathcal{R}_v \rangle$ using any planning/dynamic programming method (e.g. value iteration, policy iteration, tree search etc.) I.e. we *can* use DP if we learn the environment model!

Sample-Based Planning

- Some of most simple but effective planning methods. Treat learned model $\mathcal{M}_v$ in same way as treated environment in previous chapters; randomly sample experiences from $\mathcal{M}_v$ & learn from sampled experiences using standard model-free RL (e.g. MC learning, Sarsa, Q-learning)
- If learned model representation is in accurate, the trained agent will only ever be as good as the learned model, therefore if model is wrong must either:
  - Stop using model-based RL
  - Use e.g. Bayesian approach to reason about the model uncertainty & factor this in to model-free agent learning

8.2. Integrated Architectures

- Integrated architectures aim to bring together best of model-free & model-based RL approaches by integrating learning & planning
- **Dyna architecture**: Learn a model from real experience. Use simulated experience *and* real experience to do learning *and* planning for value function and/or policy. Dyna architecture:



8.2.1. Dyna-Q Algorithm

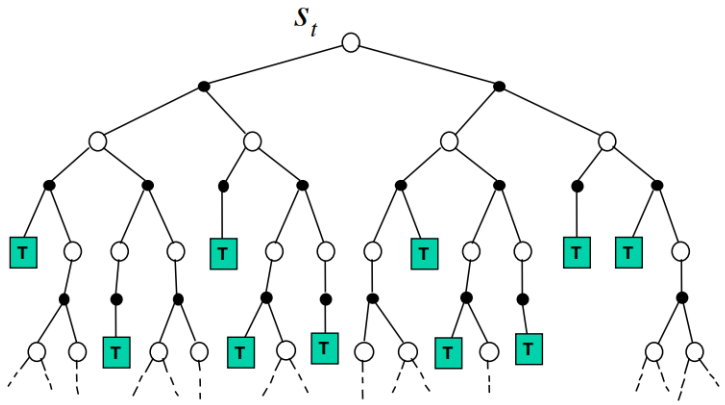
- Simplest algorithm that uses Dyna architecture
- Takes action in real *and* simulated World, compares returns, & updates both model representation & value function (and/or policy) according to difference between real & simulated return. Below in step (f), n is the number of planning steps to perform in simulated model, and is analogous to the model 'thinking' by sampling from its simulated model many different states to update it's Q-values:

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

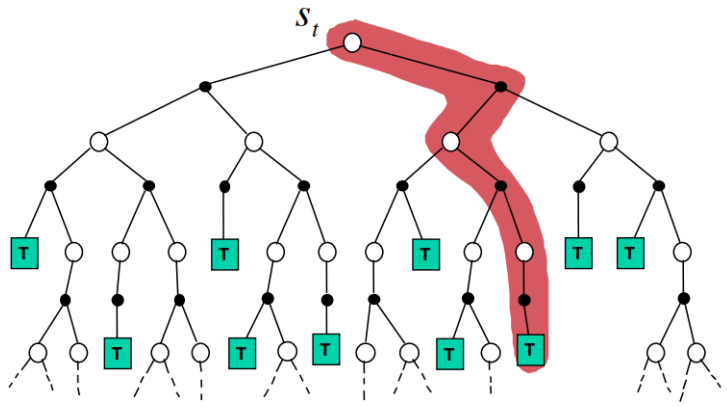
- Variant of Dyna-Q is Dyna-Q+, which does same thing but gives agent bonus for visiting previously unvisited states, encouraging more exploration & therefore finding global optimum faster

8.3. Simulation-Based Search

- **Simulation-based search**: Process of using agent's own internal model to simulate episode(s) of experience from current state s_t to terminal state(s) by applying model-free RL to simulate the episodes (i.e. treat internal model as the environment). RL agent 'imagines' the future, and uses this to enable better planning
- 2 approaches:
  - **Forward search**: Select best action by doing a **lookahead**. Build **search tree** with current state s_t as root node & use internal simulated model to build out search tree to terminal states. With simulated search tree built out, can then consider which terminal state/final long-term return was best, and choose this action in the real-World. This prevents agent from having to solve whole MDP, but rather a sub-MDP starting from now/the current state s_t:



  - **Sampled forward search**: Same as forward search where start from current state s_t, but rather than considering all possible branches when doing lookahead to build out search tree, only consider (sample) branch(es) that are most likely to occur, therefore is more efficient as only consider what matters. Is what we usually refer to when say 'simulation-based search'. E.g. Sampling the most likely branch for forward search:

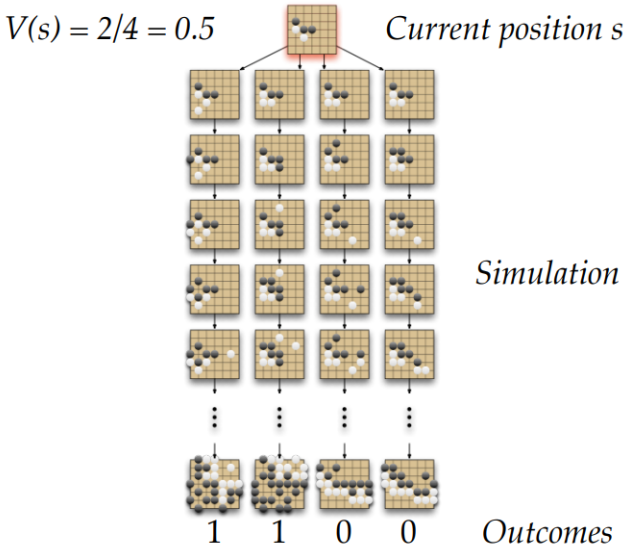


8.3.1. Monte-Carlo Tree Search (MCTS)

- State-of-the-art simulation-based search approach to RL. Used by DeepMind for AlphaGo. Applies MC control (as we've seen so far in previous chapters) to simulated experience (what we've seen in this chapter where start from current state s_t and simulate to most likely terminal state(s) and then use this lookahead search tree to pick best action in real World state s_t)

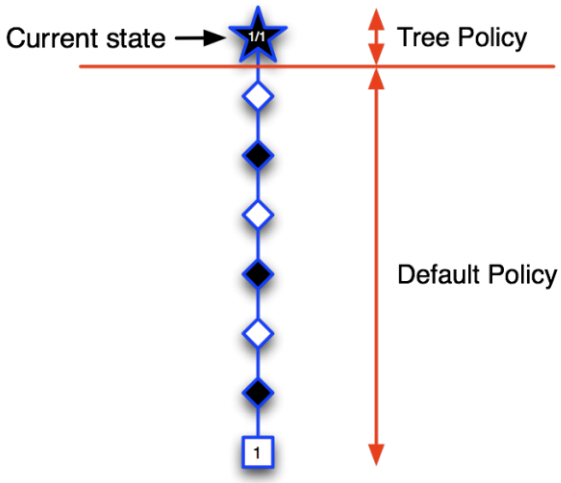
Case Study: The Game of Go

- Goal is for agent to take board state s & know how good the state is for black & white. The state transitions $\mathcal{P}$ are just the rules of Go. The reward function $\mathcal{R}$ returns $R_t = 0$ for all non-terminal steps, $R_t = 1$ if black wins, & $R_t = 0$ if white wins. Policy $\pi = \langle \pi_B, \pi_W \rangle$ selects policy both for black & white player. The value function tells us how likely agent thinks black is to win given current board state s i.e. $v_\pi(s) = E_\pi[R_T|S = s] = P[\text{Black wins}|S = s]$
- **Simple MC evaluation**: Consider applying simple MC **forward search** evaluation we've seen so far (i.e. no tree search)
  - Start at current board state s. Want to know probability black wins.
  - Perform **forward search** by running e.g. 4 internal simulations where follow policy π & play out black & white moves to terminal state:



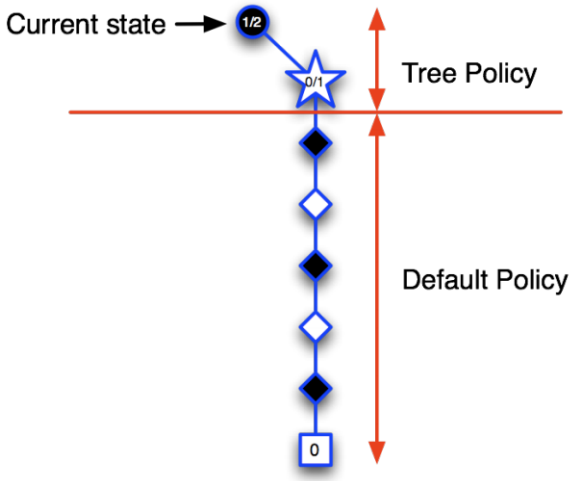
  - Find that 2 of the 4 simulations resulted in wins, therefore value of state s is $V(s) = \frac{2}{4} = 0.5$, i.e. agent will estimate a 0.5 probability of black winning in state s

- **MC Tree Search (MCTS)**: Rather than using simple forward search, use **sampled forward search** by doing a **tree search** combined with MC.
  - Again start at current board state s. Want to know probability black wins.
  - Do first forward-search simulation to terminal state to see who wins, thereby forming the first branch of your simulation tree:

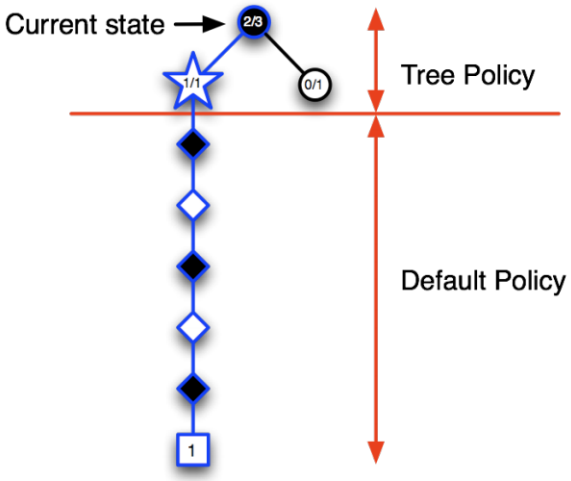


  - Found that $R = 1$ at terminal state, therefore black won. Now go back through chain of events in this branch that led to this result & update the value of the root node in search tree's branch to 1, since agent will now think that from this current state, will have 100% chance of winning

- For second simulation, start from same current state s, but now choose a different first action to explore a new branch, thereby adding a new branch to your simulation tree (with same root node at top, which is current state):

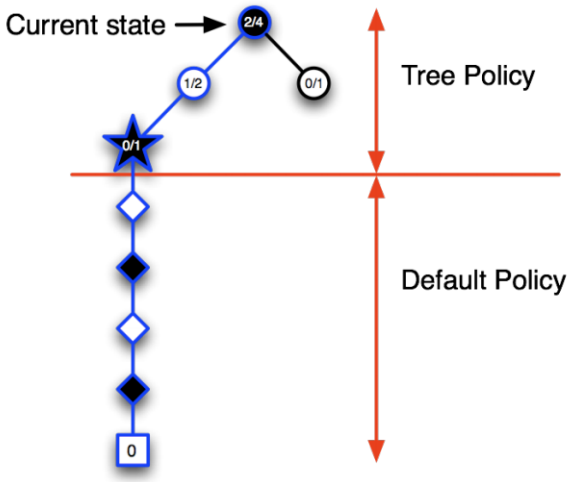


- Found that $R = 0$ at terminal state, therefore black lost. Go back through tree branch & update root node to new mean value i.e. $V(s) = \frac{1+0}{2} = 0.5$, therefore root node/current state now has estimated value of 0.5
- Since we know that this new branch resulted in a loss, following the principle of sampled forward search, we should not consider any further branches that begin from this first action node that resulted in a loss. Instead, for 3rd simulation run, consider a new first action node and run to terminal state to generate another new branch:

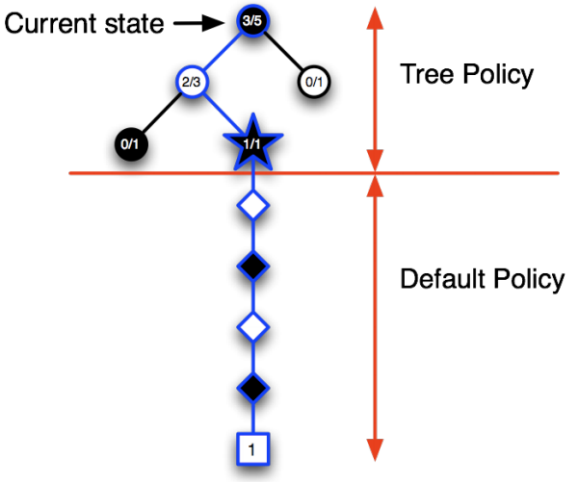


- This time find $R = 1$ at terminal state, therefore go back through branch to source node and update estimated value of current state/source node: $V(s) = \frac{1+0+1}{3} = \frac{2}{3}$

- For 4th run, now seems like we have a branch from root node/current state that looks most promising, therefore this is branch we want to focus on. 4th run starts from root node, but goes to previously added first action node that has turned out to be more promising, and then adds another new node from this node to add a new branch to this promising branch



- Find that results in $R = 0$, therefore go back to root node and update to $V(s) = \frac{1+0+1+0}{4} = \frac{1}{2}$
- Next simulation run might look like e.g.:



- Keep following this principle of only expanding (adding new nodes to) part(s) of search tree that look most promising. By law of large numbers, eventually search tree will converge on $q_\pi(s, a)$

- Pros MCTS:

  - Evaluates states dynamically as it learns (unlike dynamic programming)
  - Uses sampling whereby only consider most promising areas of search tree, therefore can handle large problems with effectively infinite search tree space since only considers sub-section
  - Only requires model (i.e. rules of game) as input. If have model, can black box MCTS & apply it to any scenario
  - Is computationally efficient as can be parallelised

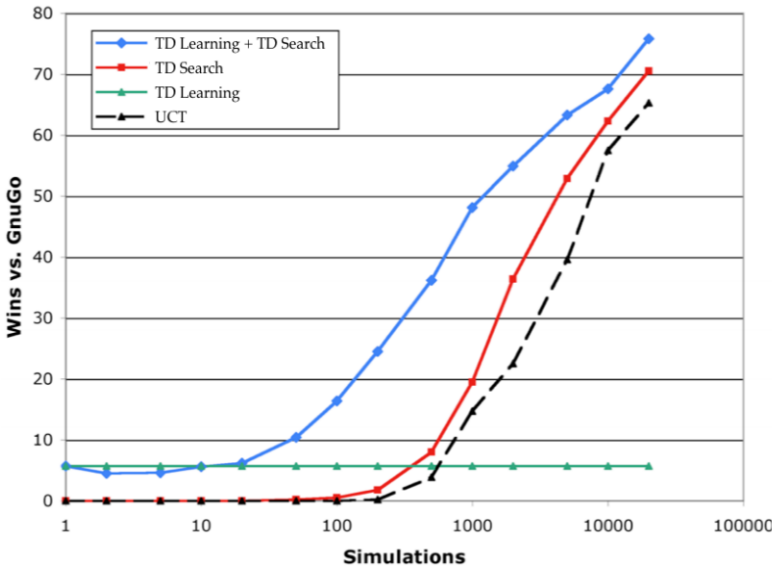
8.3.2. Temporal Difference Search

- Do same as above (i.e. simulation-based search) but now with e.g. TD Sarsa or Q-learning rather than with MC learning. Can therefore bootstrap to be more efficient & reduce variance, and optimise by using TD(λ) by finding optimum λ
- Previously did MC tree search by applying MC control to sub-MDP from current state. Here, do TD search by applying e.g. Sarsa to sub-MDP from current state.
- Works in same way as MCTS (i.e. build out simulated search tree to use as $q_\pi(s, a)$ from which to select best ϵ-greedy action for real-World in current state), but rather than using MC control, use TD bootstrapping to choose & update nodes, therefore no longer have to simulate all the way to end of episode & can handle non-episodic tasks.
- N.B. Rather than storing all Q-values (state-action pair values) in search tree nodes, can approximate search tree node values with function approximator (e.g. NN). This is being used to try to train agent to play e.g. Civilisation where state-action space would be too large to store in an actual search tree.
- TD search has same pros & cons relative to MC search as we've seen for TD vs MC RL in previous chapters

Dyna-2 Architecture

- Previously saw how Dyna architecture combined real & simulated experiences for learning.
- Dyna-2 also combines real & simulated experiences for learning, but now maintains 2 separate function approximation value functions:
  - **Long-term memory**: Update from **real** experience using **TD learning**
  - **Short-term memory**: Update from **simulated** experience using **TD search**
- Then sums values of long & short term memories to get overall value estimate for given state/state-action pair

- When applying to Go, find that Dyna-2 combination of TD learning & TD search performs best (green line 'TD learning' is where no search tree is used therefore not focusing search on promising areas of state-action space therefore TD/RL agent never finds good solutions in very large search space of Go):



9. Exploration & Exploitation

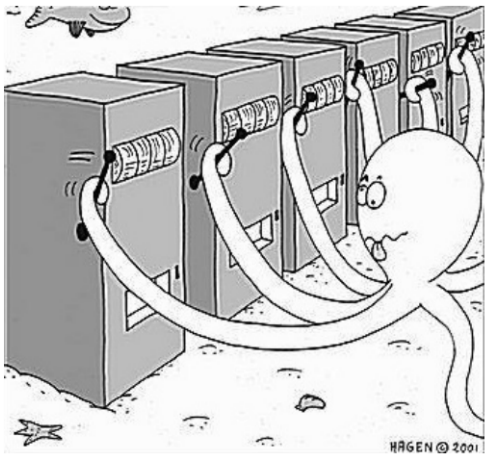
- **Learning Resource(s)**:
  1. *'Reinforcement Learning: An Introduction' (2^{nd} edition), Sutton et al., 2018,* https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
  2. *'Reinforcement Learning UCL Lecture Series', David Silver, UCL, 2015,* https://www.davidsilver.uk/teaching/
- Making decisions online involves fundamental choice:
  - **Exploitation**: Make best decision possible given current information
  - **Exploration**: Potentially give up short-term reward to gather more information and hopefully maximise long-term reward
- 3 main classes of exploration algorithms:
  - **Naive/random exploration algorithms**: Add noise to e.g. ϵ-greedy/softmax/Gaussian policy so sometimes choose random action to 'explore'
  - **Optimism in face of uncertainty algorithms**: Prefer actions with uncertain values to encourage exploring areas of MDP you know less about (e.g. UCB, probability matching with Thompson sampling). Con: Can't use in very large/continuous action spaces since would always have more uncertainty to explore & therefore would never exploit
  - **Information state search algorithms**: Assign 'uncertainty' value to each state to bring agent's uncertainty about its current state's value into its decision making process. Is probably most 'correct' exploration method, but makes state space even larger and computationally more difficult (e.g. Gittins indices, Bayes-adaptive MDPs)
- So far have considered naive approaches (ϵ-greedy, softmax, Gaussian) to balance exploration & exploitation. Now will look at more sophisticated approaches

9.1. Multi-Armed Bandits

Definitions

- **Multi-armed bandit**: A simplification of an MDP. Is tuple $\langle \mathcal{A}, \mathcal{R} \rangle$. Take 1 action, get 1 reward, then episode terminates (i.e. each episode is only 1 time step). I.e. no state space $\mathcal{S}$ or transition function $\mathcal{P}$

Have set of m actions, $\mathcal{A}$. Each action is known as an 'arm', analogous to old-fashioned 1-arm bandit machines:



Initially do not know what reward each action returns i.e. don't know $\langle \mathcal{A}, \mathcal{R} \rangle$. Goal is to maximise cumulative reward across all episodes.

- **Action-value**: The mean reward for action/bandit arm a:

$$Q(a) = E[r|a] \tag{9.1}$$

- **Optimal value** V^*: Value of optimum action:

$$V^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a) \tag{9.2}$$

- **Regret**: The 'opportunity cost'. Evaluates how well agent did over all time steps w.r.t. how well it *could* have done if head made best decision at each time step (for multi-armed bandit, 1 step step == 1 episode). Is difference between optimal value & actual value obtained at given time step:

$$I_t = E[V^* - Q(a_t)] \tag{9.3}$$

- **Total regret**: Total regret/opportunity loss over all time steps:

$$L_t = E\left[\sum_{\tau=1}^{t} V^{*-Q(a_\tau)} \right] \tag{9.4}$$

Goal of agent in multi-armed bandit problem is to **minimise total regret**

- **Count** $N_t(a)$: Total number of times action a was chosen over all time steps
- **Gap** Δ_a: Difference between value of action a chosen and value of best possible optimum action a^* that could have been chosen

$$\Delta_a = V^* - Q(a) \tag{9.5}$$

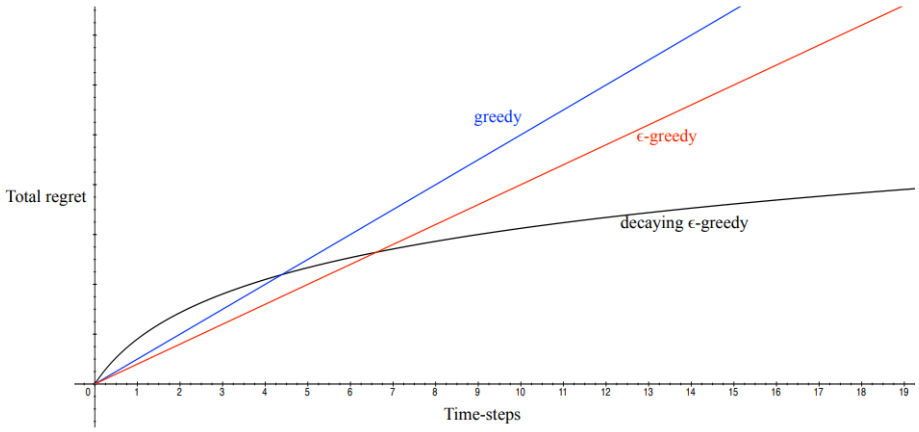
- Can use **count** & **gap** to express the **total regret**:

$$L_t = \sum_{a \in \mathcal{A}} E\big[N_t(a)\big] \cdot \Delta_a \tag{9.6}$$

  Want to have small counts & large gaps to choose an action that's far from the best the minimum number of times

- Problem: We don't now true optimum V^*, therefore can't find gap. Solution: Must learn optimum using RL. To do this, must balance exploration & exploitation whilst minimising the total regret.

- A greedy policy will never explore and therefore total regret will continue to increase linearly with time steps since never find optimum solution. A ϵ-greedy policy always has some chance of exploring therefore it will never always choose optimum solution therefore will always have some regret therefore total regret also increases linearly (but to with a lesser rate). Q: Can we achieve sub-linear total regret (i.e. stop total regret continue to increase linearly)? Turns out using more sophisticated exploration & exploitation algorithms, such as decaying ϵ greedy (where decay ϵ over time so stop exploring/choosing random actions when have learned a good policy), we can:



- N.B. turns out that, according to the **lower bound theorem**, asymptotic total regret is at least logarithmic in the number of steps i.e. can never get a total regret curve that is less than a logarithmic distribution.

9.1.1. Random Exploration Algorithms

Greedy Algorithm

- Algorithm where always choose highest value action. Never explores, therefore never finds true optimum action, therefore has linear total regret.

ϵ-Greedy Algorithm

- Similar to greedy, but now have some probability ϵ of choosing a random action rather than just the greedy action. Explores therefore finds better solutions therefore has lower total regret than greedy algorithm, but always has some chance of choosing random action therefore every makes completely optimum action choices therefore still have linear total regret

Optimistic Initialisation

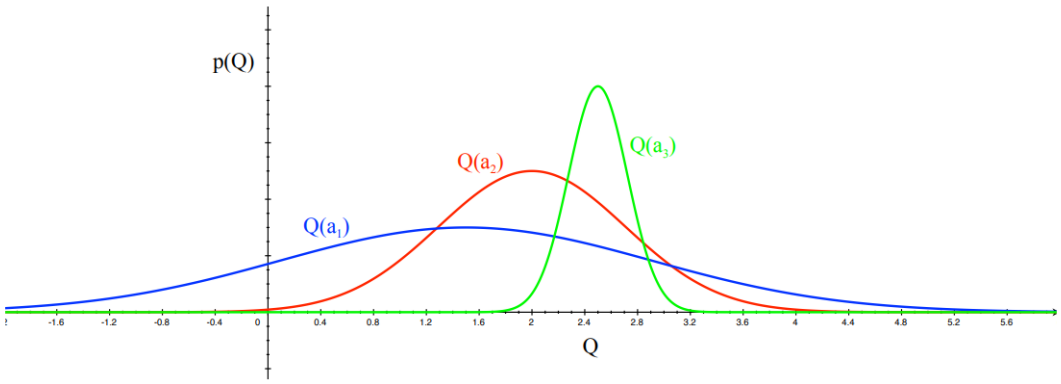
- When initialise values of state-action pairs, assume *all* actions have the maximum possible reward, then update these values as choose actions & find their true values. Can then use greedy or ϵ-greedy policy. Encourages systematic exploration early on, but still has linear total regret.

Decaying ϵ-Greedy Algorithm

- Simply decay ϵ with time such that will be 0 when find optimum policy. Turns out this simple method is highly effective and does achieve sub-linear total regret

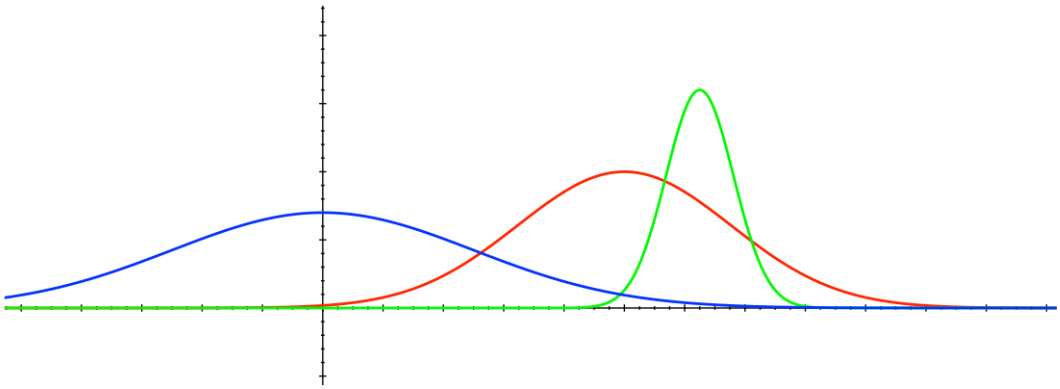
9.1.2. Optimism in Face of Uncertainty Algorithms

- Principle: Rather than picking action that you think is best, pick action that you think *could* be best
- E.g. Consider problem with 3 arms/actions, a_1, a_2, a_3. Consider the current Q-value distribution (i.e. expected reward distribution) of the agent for each action. A narrow distribution indicates agent is more confident in value of this action, whereas wide distribution indicates agent knows less about an arm:



- Following optimism in face of uncertainty principle, agent should choose action a_1, since according to its wide distribution, it *could* have highest value

- After choosing a_1 & observing reward, updated distribution might be:



- I.e. as learned more about a_1, distribution became narrower.
- Now, a_2 is the action which *could* have highest value, therefore pick this as next action
- I.e. we are optimistically picking actions we are less certain about which could potentially have highest value. Tends to give very good exploration-exploitation tradeoff & achieves sub-linear total regret

Upper Confidence Bound (UCB) Algorithm

- Method for constructing the uncertainty distributions of each action for optimism in face of uncertainty approach
- **UCB**: Defines the tail of the distribution. Usually define as being the value that agent is 95% sure the mean of the distribution is less than
- UCB algorithm chooses action with highest UCB value. Eventually, UCB will be equal to mean of distribution as become more confident in action's value and therefore narrow the uncertainty distribution
- **Hoeffding's Inequality**: Gives probability that a random variable x is less than the mean + upper confidence bound value for $x \in [0, 1]$:

$$P[X > \bar{X} + u] \leq e^{-2tu^2} \tag{9.7}$$

  Where u is the upper confidence bound value, $\bar{X}$ is the mean of the random variable probability distribution, and $P[X > \bar{X} + u]$ is the probability that a random variable has a value greater than the mean + the UCB (in our case, 5%)

- Can apply this to our RL problem of calculating upper confidence bound by scaling rewards $r \in [0, 1]$ & compute the e.g. 95% confidence interval

- E.g. Pick a probability p that true value exceeds the UCB (in our case, $p = 0.05$ i.e. only 5% probability that the true value will be higher than our UCB value). Can re-arrange Hoeffding's inequality & solve for u (the upper confidence bound value):

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}} \tag{9.8}$$

- As get more confident in action values, can reduce p to increase UCB and guarantee that we maintain enough exploration to find optimum value and achieve sub-linear regret.
- UCB algorithm performs very well. Unlike e.g. ϵ-greedy, it doesn't matter how you initialise it, will always achieve good total regret performance

9.1.3. Bayesian Bandits: Probability Matching with Thompson Sampling

- With vanilla UCB, made no assumptions about $q(a)$ distribution other than bounds on possible rewards/values.
- If know some prior distribution of what $q(a)$ distributions are, can use **Bayesian bandits** where exploit prior knowledge about rewards distributions. If prior knowledge is accurate will achieve better performance, but if inaccurate then are better off using vanilla UCB for exploration-exploitation
- **Probability matching**: Process of selecting action a according to probability that a is the optimal action, where uncertain actions have higher probabilities of being the optimum value.
- **Thompson sampling**: Method for doing probability matching. Use Bayes law to compute prior posterior value distributions for each action. Sample a reward distribution $\mathcal{R}$ from posterior. Compute action-value function $Q(a) = E[\mathcal{R}_a$. Select action maximising value on sample, $a_t = \text{argmax}_{a \in \mathcal{A}} Q(a)$
- This method achieves optimum total return curve (i.e. asymptotic logarithmic curve)

9.1.4. Information State Search Algorithms

- **Value of information**: We know exploration is useful in RL because it gains more information which maximises long-term reward. But can we quantify the value of this information? Doing so would help RL agent decide how much reward it was willing to give up to acquire new information before making a decision.
- The more uncertain a situation, the more information there is to be gained by exploring it, therefore makes sense to explore uncertain situations more (as we've seen). If we know the *value* of the information that we'll gain, can *optimally* trade-off exploration & exploitation. Information state search algorithms try to find the *optimum* exploration-exploitation strategy. I.e. is no longer just a heuristic

- Approaches to information state search algorithms include Gittins indices & Bayesian-adaptive MDPs. Con: Complicated & computationally expensive to perform

9.2. Contextual Bandits

- Like multi-armed bandits, are a simplification of an MDP, but now consider the states $\mathcal{S}$ (i.e. the 'context' in which decisions are made) as well as actions & rewards. I.e. is tuple $\langle \mathcal{A}, \mathcal{S}, \mathcal{R} \rangle$
- Again, $\mathcal{A}$ is a set of m actions, $\mathcal{S}$ is an unknown distribution over states, & $\mathcal{R}$ is a distribution over rewards. At each step, environment generates state s_t $\mathcal{S}$, agent selects action $a_t \in \mathcal{A}$, & environment generates rewards r_t $\mathcal{R}_{s_t^{a_t}}$. Goal of agent is to maximise cumulative reward
- Advertising has been most successful application of contextual bandit problem. The arms (actions) are set of adverts that can show to users, & the context (states) are the information about the user (e.g. user recently looked at cooking books on Amazon etc.). Have also been successful in e.g. showing which news stories to show to a user. Get +1 reward if user clicks on advert/news story, & 0 otherwise.

9.3. MDPs

- All exploration-exploitation principles used for multi-armed bandits & contextual bandits can also be extended to the full MDP case. I.e. can use naive methods (ϵ-greedy etc), optimistic in face of uncertainty methods & information state search algorithms we've seen to all problems using same principles

Part III.

Deep Learning

10. Neural Network Foundations

- **Learning Resource(s)**:
  1. *'Advanced Deep Learning & Reinforcement Learning, Lecture 3: Neural Networks Foundations', Simon Osindero, UCL-DeepMind Lecture Series, 2018,* https://www.youtube.com/watch?v=5eAXoPSBgnE
  2. *'Deep Learning', Goodfellow et al., 2016,* http://www.deeplearningbook.org
  3. *'Machine Learning', Andrew Ng, Coursera Stanford, 2011,* https://www.coursera.org/learn/machine-learning
  4. *'Neural Networks and Deep Learning', Michael Nielsen,* http://neuralnetworksanddeeplearning.com/
- Neural networks (NNs) are compositions of (i) **linear transforms** & (ii) **non-linear functions**
- NNs are optimised by **learning**, which is the process of optimising the NN's **loss function** over some training data or real-World experience by tuning the NN's tuneable parameters. This optimisation is usually done with **stochastic gradient descent (SGD)**
- Training a NN is learning the set of parameters w (e.g. weights) that take an input x and produce an output y. I.e. the trained NN is a function that approximately maps x to y:

$$y = nn(x, w) \tag{10.1}$$

- Examples of NN inputs x:
  - Images
  - Features
  - Words
  - World observation + reward
- Examples of NN outputs y:
  - Labels
  - Predictions
  - Next state in World

  - Next action to take in World

- Notes on terminology:
  - Neuron == unit
  - non-linearity == activation function == non-linear function
  - Combination of a linear transform and a non-linear function == a 'layer'
  - Operation == op == a 'layer'
  - Traditional sketch of NN layers == TF-style modern computation graphs

- **Feedforward**: Process of passing information forward from the first NN layer to the final output layer. Model outputs are *not* fed back into the model (i.e. NN doesn't have 'feedback' connections). Models that have these feedback connections are **recurrent neural networks** (RNNs) (see later). Feedforward NNs form basis of many types of NNs, including **convolutional neural networks** (CNNs) (see later).

- **Network**: Refer to feedforward NNs as a *network* because they are a composition of different functions all connected in a chain to form a directed acyclic graph. The functions are referred to as the *layers* of the NN, and having multiple functions/layers gives the NN *depth*, hence the name **deep neural network**. Example of simple NN with 1 hidden layer (i.e. *not* deep) represented as sequential chain of functions:

$$f^{nn}(x) = f^{output}(f^{hidden}(f^{input}(x))) \tag{10.2}$$

- **Hidden layers**: All NNs have an input layer and an output layer. When training NNs, we only give the desired final output labels/targets, which is the desired output of the final output layer. We do not give the intermediary layers any target outputs. Therefore, these intermediary layers have 'hidden' targets, hence why we refer to layers between the input and output layer as **hidden layers**.

- **Neural**: Although NNs are loosely based on the brain, modern feedforward networks are best thought of as **function approximation machines** designed to achieve **statistical generalisation** guided by mathematical & engineering principles.

10.1. Single-Layer Neural Networks

10.1.1. Linear Layers

Linear Transformation Units

- Linear layers (a.k.a. **dense layers** or **fully connected layers**) are just a NN layer that applies some **linear transformation** to some data.

- E.g. a **linear unit** might take some input value(s) x, multiply the input by its weight value(s) w, add its bias term b, and linearly sum over the result across all inputs & weights to get the unit's output y:

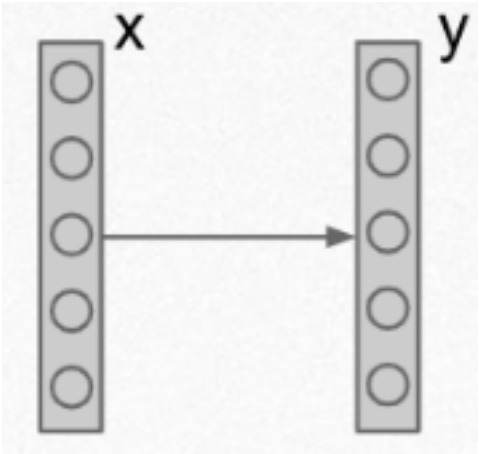
$$y = \sum_i w_i x_i + b \tag{10.3}$$

Linear Transformation Layers

- Multiple of these linear units transforming different parts of input x in parallel form a **linear layer** of linear units performing this linear transformation. We can describe this layer of linear units as a matrix-vector operation (i.e. each unit has its own input(s), bias term, output, and vector of weights, therefore the layer's inputs, outputs and biases are a vector and weights are a matrix):

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \tag{10.4}$$

- We can then use this linear layer as a *linear regressor* to learn the set of weights w for each unit in the linear layer that multiply the inputs x such that the desired final NN output y is achieved.



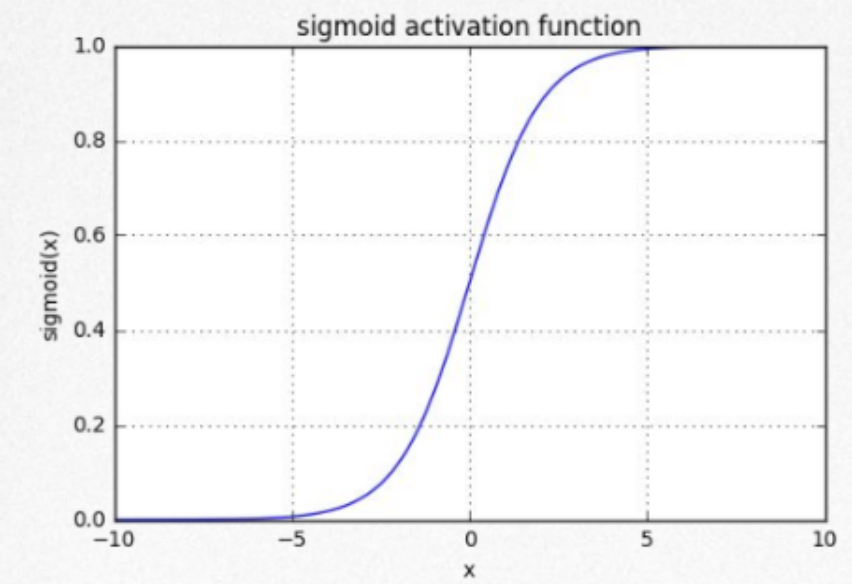
10.1.2. Non-Linear Activation Functions

- **But** to be able to learn more interesting functions & gain deeper insights with our NN, we typically follow this linear transformation layer with a non-linear function, known as the **activation function**. Therefore a single NN layer is usually comprised of a linear transformation layer followed by a non-linear activation function

- A common activation function used for binary classification problems (where want to output 0 or 1) is the **sigmoid** activation function σ:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{10.5}$$

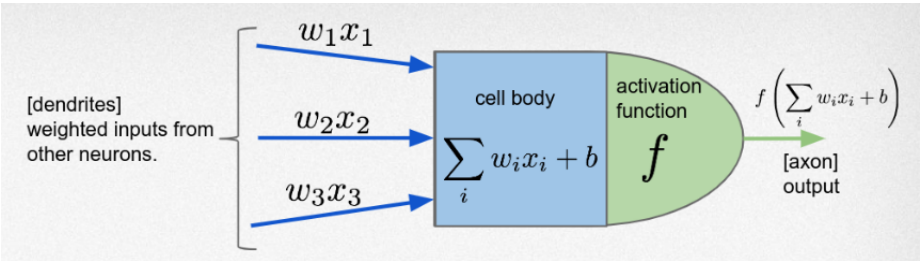
- The sigmoid function is a 'squashing' function in that it will 'squash' any input x to be between 0 and 1:



- Typically, the activation function is applied to the output of the linear transformation layer for each unit, giving the final output of the unit:

$$y = \sigma\left(\sum_i w_i x_i + b\right) \tag{10.6}$$

- The output of the layer is therefore described again by the matrix-vector operation combining parameters across all the units in the layers
- There are many different types of activation function, and the best choice of activation function depends on the specific problem. However, each NN layer is generally made up of a linear transformation layer & a non-linear activation function:



- The sigmoid activation function was popular in the 90s & 2000s, but due to the difficult changing gradient properties of the function, it is less scalable to deep-NNs and therefore less popular today. However, it is still popular in NNs that use **gating**, such as **long short-term memory** (LSTM) RNNs (see later)

Binary Classification with Sigmoid

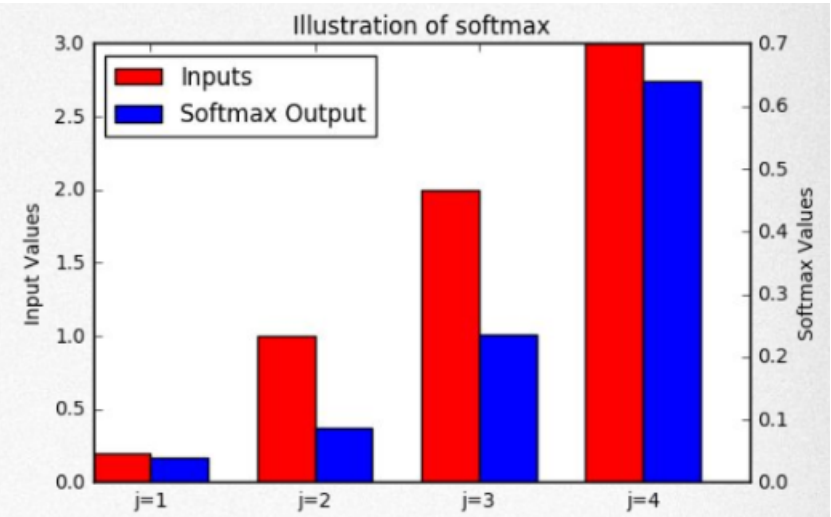
- Since the sigmoid activation function outputs 0 or 1, it can be used for binary classification, where only need to classify an input as 1 of 2 possible classes

Multi-Class Classification with Softmax

- To extend a single-layer NN to multi-class classification where classify inputs into >2 different classes, a common activation function to use is the **softmax** function.
- We know that the argmax function just takes some set of inputs $\mathbf{x}$ and makes the largest element 1 and all other elements 0.
- The softmax function is similar to argmax, but rather than applying a 'hard max' where make one element equal to 1 and all others 0, softmax scales the elements of $\mathbf{x}$ between 0 and 1 depending on how high the input element's value was, with the highest-value element being the closest to 1.
- The inputs x are passed to the softmax function to get a vector of outputs $\mathbf{y}$, where the elements of $\mathbf{y}$ are between 0 and 1 and together sum to 1. This output can therefore be interpreted as the **probability distribution** over all the K indices (classes) of $\mathbf{y}$:

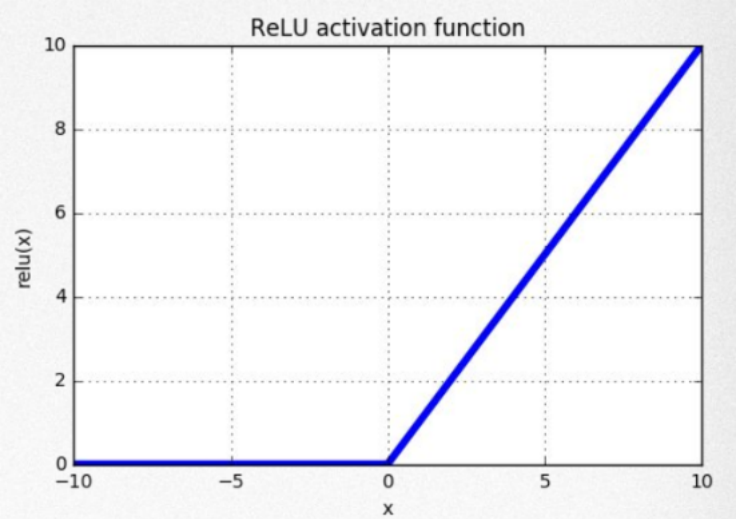
$$\mathbf{y} = \text{softmax}(\mathbf{x}), \text{where:} \tag{10.7}$$

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{10.8}$$



Rectified-Linear Unit (ReLU)

- A disadvantage of the sigmoid function is its changing gradient, which is computationally complicated and expensive to calculate. For deep NNs, this is a particular problem, limiting the scalability of sigmoid.
- The **ReLU** activation function, once activated, has a **constant gradient**, which is much easier to compute and therefore makes learning more efficient:



Other Types of Hidden Unit

- The relu activation function is very popular with deep NNs today. Indeed, usually all the hidden units are relu units. Which hidden unit is best depends on the specific problem, and can only be determined through trial-and-error. Some other hidden units include:
  1. Generalisations of ReLUs:
     - Absolute value rectification units
     - Leaky ReLUs
     - Parametric ReLUs
  2. Maxout units
  3. Radial basis function (RBF) units
  4. Softplus units
  5. Hard tanh units
- Discovering new hidden units is an active area of research and there are likely many different types of hidden units still undiscovered. The key point to remember is that using sigmoid hidden units is discouraged since learning the changing gradient is very

complicated for deep learning models, therefore sigmoid units tend to be used only in the final output layer for feedforward NNs (although, as will be discussed later, they are used extensively in RNNs such as LSTMs). Softmax units are commonly used in the output layer, and are sometimes used in the hidden layer(s) as well. The most common hidden layer unit is the ReLU, but many other types of hidden unit are available for selection.

10.1.3. Loss Functions

Cross-Entropy Loss

- The 'cross-entropy' is often used in machine learning as the loss function that must be minimised by adjusting the NN's trainable parameters with some optimisation algorithm
- **Cross-Entropy**: A measure of the difference between 2 probability distributions. Builds on idea of entropy from information theory; it is the number of bits required to represent/transmit an **average** event from one distribution compared to another
- **Information**: The number of bits required to encode and transmit some *event*. Low-probability events occur less frequently, therefore need more bits to be described and therefore have more information, whereas high-probability events have less information.
- **(Information) Entropy**: The number of bits/ amount of information required to transmit an event that has been selected from a probability distribution. If the distribution is skewed (i.e. events in the distribution have different probabilities of occurring), you have a higher chance of picking a random event that has a high probability (i.e. the event is likely **unsurprising**), therefore the information/number of bits of the high-probability event is low, therefore the **entropy of the distribution is low**. Similarly, for a distribution where events have equal/similar probabilities, no events occur much more frequently than others and have a low probability of occurring (the event will be **surprising**), therefore they all require a relatively large number of bits/information to encode, therefore the **entropy of the distribution is high**.
- Consider we have some distribution Q that is trying to approximate some underlying target distribution P. The **cross-entropy** of Q from P is the **number of additional bits to represent an event using Q instead of P**. The cross-entropy of 2 probability distributions (e.g. Q from P) is defined with the cross-entropy function $H()$::

$$H(P, Q) = -\sum_{x} P(x) \cdot logQ(x) \tag{10.9}$$

- $P(x)$ is the probability of event x in distribution P, and vice-versa for $Q(x)$. The result is a positive number of bits and will be equal to the entropy of the distribution if the two probability distributions were identical.
- Cross-entropy is commonly used as the loss-function in classification models (e.g. logistic linear regression algorithms, neural network algorithms, etc.). To apply to the classification problem, the random variable of our distributions to compare is the input

example/data features for which we require a predicted class label, and the event is each of the possible class labels.

- The known probability of each class label for an example is given by distribution P, and model's predicted probability of each class label for an example is given by distribution Q. We can therefore calculate the cross-entropy for a single prediction to get the 'difference' between the model's prediction and the ground-truth target label. Minimising the difference between these distributions (i.e. minimising their cross-entropy) will bring the model closer to predicting the correct class given an example, therefore cross-entropy can be used as a loss function.
- N.B. The specific cross-entropy functions used for NNs are usually a **negative log-likelihood** (NLL) function for classification problems (as above).
- An alternative to cross-entropy for the loss function is the **mean squared error** function for regression problems.
- It has been found that, for classification problems, using the cross-entropy NLL function as the loss function instead of e.g. mean squared error leads to (i) faster training (ii) improved generalisation.

Choosing a Loss Function for a NN

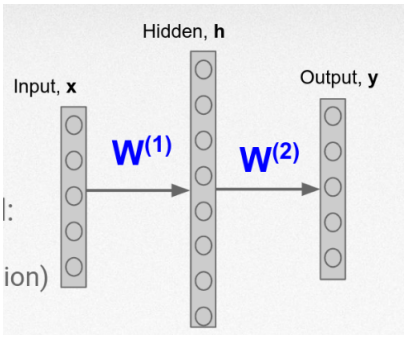
- The activation function in the output layer is chosen depending on how you want to frame your prediction problem. The choice of loss function depends on the activation function used in this output layer.
- Common NN problems & configurations:
  1. **Regression**: The problem of predicting a continuous real-valued quantity.
     - **Output layer config**: One node with a linear activation unit
     - **Loss function**: Mean squared error (MSE)
  2. **Binary Classification**: The problem of classifying an example into 1 of 2 possible classes.
     - **Output layer config**: One node sigmoid activation unit
     - **Loss function**: Cross-entropy NLL function
  3. **Multi-CLass Classification**: The problem of classifying an example into 1 of >2 possible classes.
     - **Output layer config**: One node for each class using a softmax activation unit
     - **Loss function**: Cross-entropy NLL function

10.2. Neural Networks with One Hidden Layer

- So far we have seen a basic single-layer NN. Now we will look at what happens if we add a *hidden layer* to our NN.
- Notes on terminology:
  - Traditionally, a **hidden layer** means a linear transformation + a non-linear activation function combined into a single operation (as we've seen so far and will continue to use for this section). **But** more modern terminology refers to these linear transformations & non-linear activation functions as *separate layers/modules*. This is especially common when looking at NNs as computation graphs in TensorFlow.

10.2.1. Hidden Layers

- As stated, by hidden layer, we mean the compound of:
  1. A linear transformation module
  2. A non-linear module (i.e. an activation function)



- Having a hidden layer allows us to have *intermediate* representations of the original input data by performing transformations on it, which can simplify the problem.

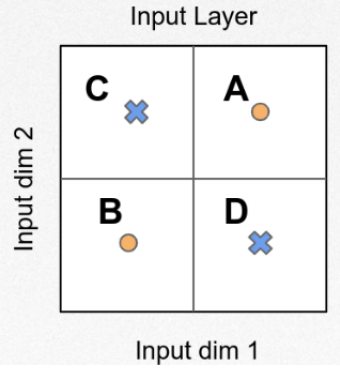
Example: Learning XOR

- To demonstrate the power that just one hidden layer in our NN can add with these intermediate representations, we will look at using a NN with 1 hidden layer to learn the XOR function.
- The XOR ('exclusive or') function takes two inputs x_1 and x_2. If 1 of these inputs is equal to 1, XOR returns 1. Otherwise, it returns 0 (i.e. if both = 1 or if neither = 1, will return 0).
- This is a **binary classification** problem where we want to classify our set of inputs (x_1 and x_2) into class 0 (orange) or class 1 (blue). Consider 4 sets of inputs, A, B, C and D, with their x_1 and x_2 values (in the 'original space') given in a tuple. A and B have the

ground-truth class label 0, and C and D have ground-truth label 1 (ignore the 'hidden space' column for now):

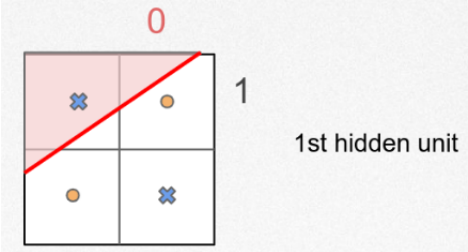
Point	Original Space	Hidden Space	
A	(1,1)	(1,0)	Class 0
B	(-1,-1)	(1,0)	
C	(-1,1)	(0,0)	Class 1
D	(1,-1)	(1,1)	

- There are 2 input dimensions in the original space tuples: x_1 and x_2. Plotting the inputs on this original space, we see that there is no straight (linear) line that can be plotted that will separate these inputs into their ground-truth classes, therefore the **inputs are not linearly separable**:

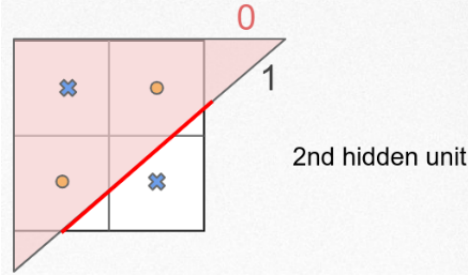


- Imagine that we pass this input data into a hidden layer with 2 units. The first hidden unit generates a new representation for the x_1 dimension, and the second hidden unit generates a new representation for the x_2 dimension. These representations are generated by plotting a linear line somewhere in the original space and generating a new representation for each dimension depending on which region/side of the line the input is:

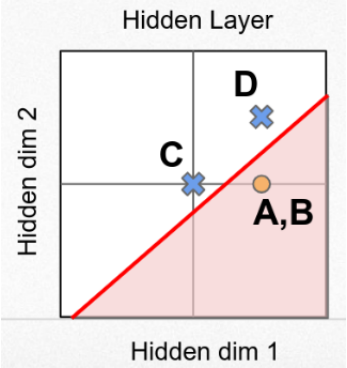
    1. First hidden unit: Plot line. If input data is above line (red region), transform x_1 dimension of example to 0, else transform to 1:



2. Second hidden unit: Plot line. If input data is above line (red region), transform x_2 dimension of example to 0, else transform to 1:



- Combining the transformations of these 2 hidden units, the hidden layer outputs a new **hidden space representation** of the inputs (see above table's 'hidden space' column). This intermediate representation of the input data is now linearly separable, and therefore can be passed to our output layer for easy classification using a simple sigmoid function (outputs 0 or 1):



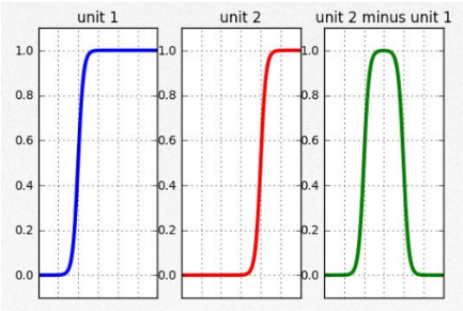
- The XOR function is very simple, therefore we only need 2 hidden units to approximate it with our NN. As we try to approximate more complicated functions with more dimensions, we will need more hidden units, but the basic principle of generating intermediate space representations with a layer of hidden units remains the same. The number of hidden units to use (i.e. the **width** of the layer) depends on the problem, and is a hyperparam to tune.

10.2.2. NNs as Universal Approximators

- It has been shown that, given enough hidden units, a NN with just one hidden layer can approximate any continuous function with a finite number of hidden units (if it has enough hidden units & the correct weights & biases). Hence, NNs with just 1 hidden layer are **universal function approximators**.

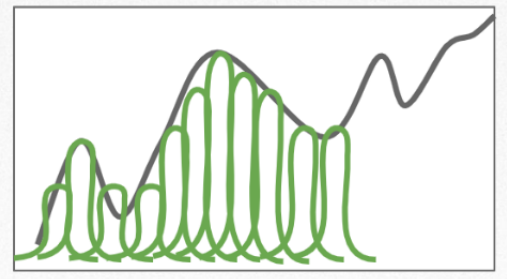
Visual Proof

  - **1D**: Consider 2 hidden units whose sigmoid functions are slightly offset w.r.t. oneanother. When combined, the amplitudes of these functions form a localised 'bump':



  - Imagine that we want to use this localised bump that results from combining 2 hidden units to represent a 1-dimensional function (drawn in grey below)

  - We can do this by using multiple hidden unit pairs to create multiple bumps & adjusting the weights & biases of the hidden units to translate & scale each individual bump. When the magnitudes of the bumps are combined, the hidden units will have formed an approximation of the desired function. As we add more hidden units to the layer, we will have more bumps and therefore more accurately approximate the function:



  - **2D**: Can also scale this same concept of function approximation to higher dimensions, but each local 'bump' will need more dimensions, therefore will need additional hidden units for each bump

- **Con of single hidden layer**: To approximate high-dimensional functions, if we use only 1 hidden layer, we require exponentially many more units to approximate the function.

This is computationally very expensive and not scalable.

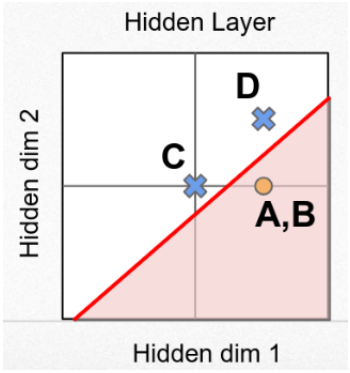
- Therefore, in order to efficiently approximate complex high-dimensional functions, we can add *multiple* hidden layers (rather than exponentially increasing number of hidden units) to form a **deep** NN

10.3. Modern Deep Neural Networks

- We have seen that a NN with just 1 hidden layer is a universal function approximator, but that to approximate high-dimensional functions you need exponentially many more hidden units, which is not computationally efficient or scalable.
- Instead, we can use *multiple* hidden layers. Just as we saw in the single hidden layer XOR example, each hidden layer generates an intermediate representation of the input data. Using multiple hidden layers breaks down the complex functional mapping into even smaller intermediate re-representation steps.

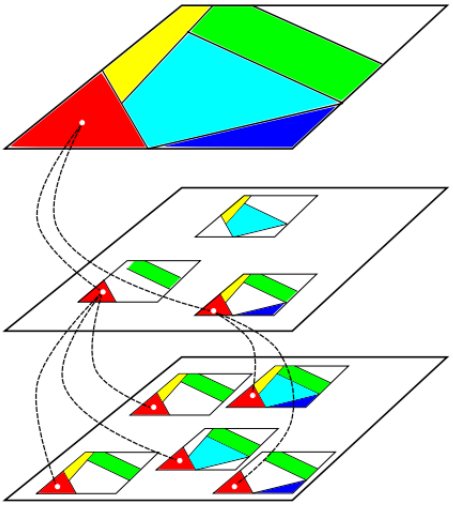
10.3.1. Why Adding More Depth Helps

- Good visualisation & theoretical analysis in *'On the Number of Linear Regions of Deep Neural Networks, G. Montufar et al., 2014,* https://arxiv.org/abs/1402.1869
- Previously in the XOR example we saw how a hidden layer generates a hidden representation of the input. In the XOR example, we generated a hidden representation that mapped 2 different points in the input (A and B) onto the same point in the hidden representation output, allowing us to separate the intermediate space representation into 2 linear regions using just 2 hidden units:



- This can be thought of as **folding** the input space in a non-trivial way such that A and B overlap (i.e. different inputs map to the same output).
- We know that in this hidden folded state representation, each region (linearly separable by a line) represents a label that our NN can assign to our input data's features (e.g. class 0 or class 1).

- It turns out that if we recursively fold the space representation (i.e. use multiple hidden layers rather than just 1), we create **exponentially** many more linearly separable regions, where each region in a subsequent layer contains the regions/labels for all previous layers underlying that particular region:



- Therefore as the number of hidden layers is increased, the number of regions (i.e. assignable feature labels) increases **exponentially**, which allows us to use our (deep) NNs to make much more meaningful insights about the input data, since the NN can assign a wider variety of different feature labels to the input data.
- It also turns out that increasing the number of hidden units in a layer only increases the number of linear regions **polynomially**, therefore we get much more powerful representations using **deep & narrow** NNs (many layers, few units per layer) than **shallow & wide** (few layers, many units per layer) NNs.

10.3.2. NNs as Computation Graphs

- Consider a deep NN composed of the following layers (an input layer, 3 hidden layers, & an output layer):
  - First hidden layer (takes original space representation as input, applies linear transformation + sigmoid activation function):

$$\boldsymbol{h}^{(1)} = \sigma(\boldsymbol{W}^{(1)} \cdot \boldsymbol{x}) \tag{10.10}$$

  - Second hidden layer (also takes original space representation as input, applies linear transformation + relu activation function):

$$\boldsymbol{h}^{(2)} = \text{relu}(\boldsymbol{W}^{(2)} \cdot \boldsymbol{x}) \tag{10.11}$$

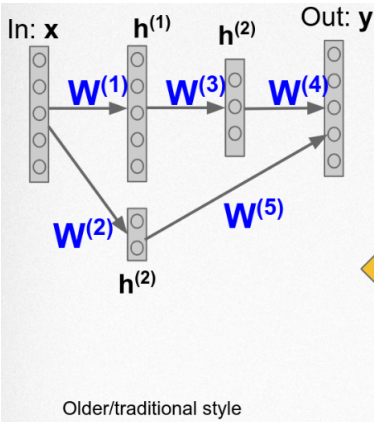
- Third hidden layer (takes hidden space representation generated by first hidden layer, applies linear transformation + sigmoid activation function):

$$\boldsymbol{h}^{(3)} = \sigma(\boldsymbol{W}^{(3)} \cdot \boldsymbol{h}^{(1)}) \tag{10.12}$$

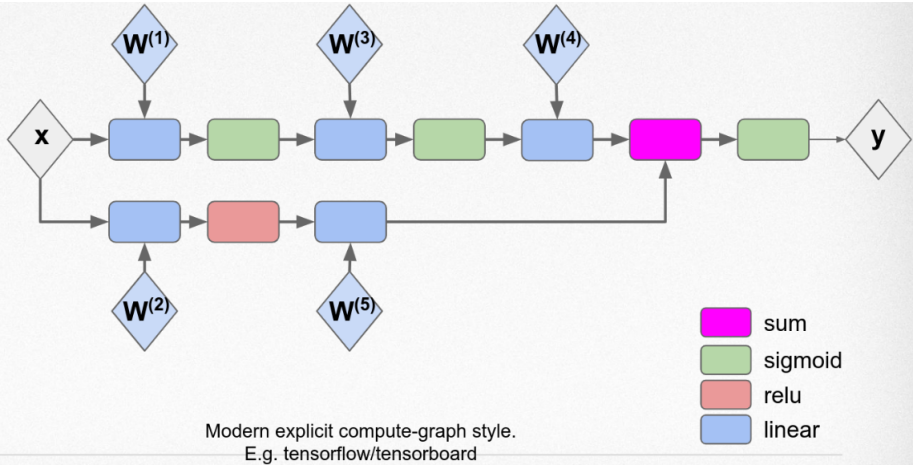
- Output layer (takes hidden state representations generated by (i) second and (ii) third hidden layers, applies linear transformation (sums these 2 representations together) + sigmoid activation function):

$$\boldsymbol{y} = \sigma\left(\boldsymbol{W}^{(4)} \cdot \boldsymbol{h}^{(3)} + \boldsymbol{W}^{(5)} \cdot \boldsymbol{h}^{(2)}\right) \tag{10.13}$$

- N.B. Note how the layers of this NN are not all connected in a chain, but rather have a main chain of layers with an added extra layer of *skip connections* going from the input layer to the second hidden layer to the output layer, skipping hidden layers 1 and 3 (see diagram below); skip connections make it easier for the gradient to flow from the output layer to layers closer to the input, which can make learning easier.
- So far, we have represented NNs in the traditional way; as a sketch of layers made up of an input layer, an output layer, and a hidden intermediate layer(s). E.g. the above deep NN (Figure error: $h^{(2)}$ at bottom is second hidden layer (in parallel with $h^{(1)}$), so top $h^{(2)}$ should be $h^{(3)}$ i.e. 3 hidden layers in total):



- As we start building different kinds of deep NNs, the NN topology can become very complex, and so it becomes more useful to represent our NNs as **computational graphs**. Representing the same NN as above but now as a computational graph (done in e.g. TensorFlow):



- NN computational graphs are made up of building blocks/modules. In object-oriented programming, these blocks are just classes that we initialise as objects and put together to make a NN (analogous to lego!).
- Up to now and in the above traditional NN sketch diagram, we've referred to a 'layer' as being a combination of a linear transform and an activation function. Now, in the context of computation graphs, we refer to a layer as a building block/object, where the linear transformations and the activation functions are now separated into individual modules.
- Each object/block only needs to have 3 core methods/functionalities ($\boldsymbol{J}$ is the *Jacobian matrix*, see later):

  1. forward_pass($\boldsymbol{x}$): Pass the input $\boldsymbol{x}$ that was given to the module into the module's op and get an output $\boldsymbol{y}$

  $$\boldsymbol{y} = f(\boldsymbol{x}) \tag{10.14}$$

  2. backward_pass($\frac{\delta L}{\delta \boldsymbol{y}}$): Given the gradient of the loss L (difference between model output & target output) w.r.t. the final NN output $\boldsymbol{y}$, $\frac{\delta L}{\delta y}$, calculate the gradient of this loss w.r.t. the inputs x, $\frac{\delta L}{\delta x}$

  $$\frac{\delta L}{\delta \boldsymbol{x}} = \frac{\delta L}{\delta \boldsymbol{y}}\left(\boldsymbol{J}^{(\boldsymbol{x})}(\boldsymbol{y})\right), \text{where:} \tag{10.15}$$

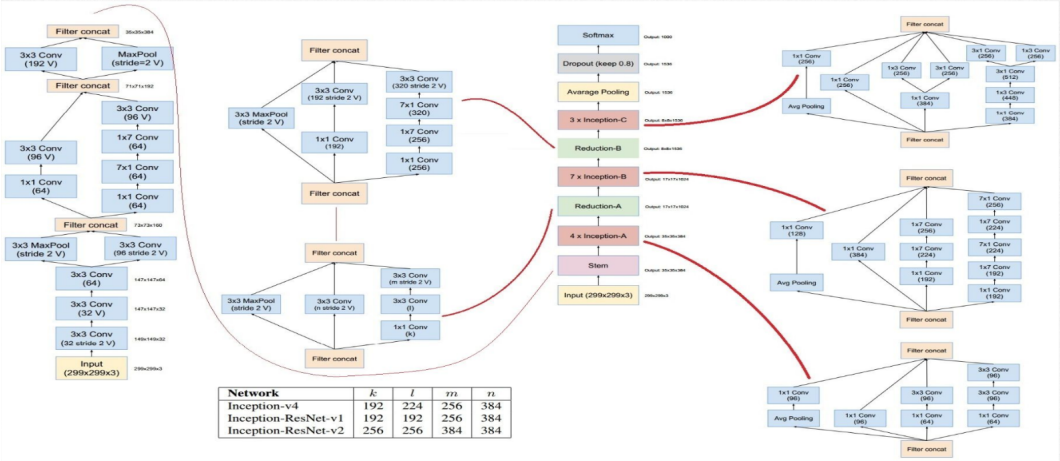
  $$\frac{\delta L}{\delta x_i} = \sum_{j=1}^{J} \frac{\delta L}{\delta yi}\frac{\delta y_j}{\delta x_i} \tag{10.16}$$

  3. compute_gradients($\frac{\delta L}{\delta \boldsymbol{y}}$): Given the gradient of the loss L w.r.t. the final NN output $\boldsymbol{y}$, calculate the gradient of this loss w.r.t. the NN's trainable parameters $\boldsymbol{\theta}$

  $$\frac{\delta L}{\delta \boldsymbol{\theta}} = \frac{\delta L}{\delta \boldsymbol{y}}\left(\boldsymbol{J}^{(\boldsymbol{\theta})}(\boldsymbol{y})\right), \text{where:} \tag{10.17}$$

$$\frac{\delta L}{\delta \theta_i} = \sum_{j=1}^{J} \frac{\delta L}{\delta y i} \frac{\delta y_j}{\delta \theta_i} \tag{10.18}$$

- As an example of how a very complex NN model can be represented as a computational graph made up of simple blocks/modules, here is the **Inception V4** NN model:



Network	k	l	m	n
Inception-v4	192	224	256	384
Inception-ResNet-v1	192	192	256	384
Inception-ResNet-v2	256	256	384	384

- Side note: The NNs we have considered so far have had **fully connected** layers, where every input unit is connected to every output unit between layers. However, to reduce the number of connections between input and output units and therefore reduce (i) number of parameters to optimise (ii) computation cost of training, a common strategy is for each unit in the input layer to only connect to a small subset of units in the output layer (i.e. be **sparsely connected**). E.g. CNNs (see later) use specialised sparse connection patterns.

10.4. Learning

- Make a NN **learn** by defining a **loss function**, and then adjusting the NN model's tuneable parameters $\boldsymbol{\theta}$ such that the loss is minimised. These parameters are changed using some **optimisation** algorithm. Most common optimisation is **stochastic gradient descent (SGD)**. N.B. There are also other 'gradient-free' optimisation methods e.g. evolutionary algorithms, swarm intelligence algorithms etc., which can be useful for e.g. RL scenarios where don't have a good gradient signal

- 2 key types of objects used during learning:

  1. **Gradient vector** $\frac{\delta y}{\delta \boldsymbol{x}}$: The partial derivatives of a **scalar** function w.r.t. **vector** arguments (e.g. derivative of model's loss L (scalar) w.r.t. model's tuneable parameters $\boldsymbol{\theta}$, or w.r.t. model's inputs $\boldsymbol{x}$, or w.r.t. model's outputs $\boldsymbol{y}$)

Gradient

$$y = f(\mathbf{x})$$
$$f : \mathbb{R}^m \to \mathbb{R}$$
$$\frac{\partial y}{\partial \mathbf{x}} = \nabla^{(\mathbf{x})} y = \left[\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_m}\right]$$

2. **Jacobian matrix** $\frac{\delta \boldsymbol{y}}{\delta \boldsymbol{x}}$: The partial derivatives of a **vector** function w.r.t. **vector** arguments

Jacobian

$$\mathbf{y} = f(\mathbf{x})$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{J}^{(\mathbf{x})}(\mathbf{y}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

- **Epoch**: 1 cycle through the entire training dataset

10.4.1. Optimisation with Gradient Descent

- Excellent intro paper: 'An Overview of Gradient Descent Optimisation Algorithms' (https://arxiv.org/pdf/1609.04747.pdf)
- 3 main approaches to performing gradient descent updates:

1. **Stochastic/Online Gradient Descent**: Calc loss for each training example, update model for each example. Perform update after each i^{th} of m training examples (where η is the **learning rate**, which controls how large the updates are and is another hyperparam to tune):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \Delta^{(\boldsymbol{\theta})} L\left(\boldsymbol{\theta}, \left\{\boldsymbol{x}^{(i)}, y^{(i)}\right\}_{j=1}^{m}\right) \tag{10.19}$$

Pros:

a) Simple to understand and implement

b) Model is updated frequently, which can make learning faster in some scenarios

c) Update process is noisy (since updates at every example, even anomalous etc.), which can help avoid local minima

d) Frequent updates mean can constantly monitor performance and rate of improvement during training

Cons:

a) Frequent updates more computationally expensive, therefore takes long time to train on large datasets

b) Frequent updates can create noisy gradient signal which cause model parameters to 'jump-around' > high variance over training epochs

c) Noisy gradient makes convergence on model parameters with minimum loss more difficult

2. **Full-Batch Gradient Descent**: Calc loss for each training example, but only update model after all training examples have been evaluated (i.e. perform updates only after each epoch). Update w/ full training batch $\boldsymbol{x}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \Delta^{(\boldsymbol{\theta})} L\Big(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}\Big) \tag{10.20}$$

Pros:

a) Fewer updates > computationally efficient > shorter training time for large datasets

b) More stable gradient signal > can give more stable convergence for some problems

c) Separates loss calculations from model updates, therefore can calculate losses for each example *in parallel* (since each won't individually affect model update) > very efficient

Cons:

i. Stable gradient makes model prone to converging on a local minimum

ii. Update at end of epoch must accumulate prediction errors across all training examples > more complex to do than just for 1 example

iii. Often need to have entire training set in memory to be available to algorithm after epoch > not practical for very large datasets

iv. Process of updating model over all examples is very slow if have large number of examples > slower training speeds

3. **Mini-Batch Gradient Descent**: Split training dataset into small/mini batches > go through each mini batch > calc loss for each example in mini batch > at end of mini-batch, update model. Balances pros and cons of SGD & full-batch gradient descent. Is most commonly used for deep learning. Smaller batch size > faster convergence, but more noisy updates. Larger > slower convergence but easier convergence with more accurate gradient signal estimates. Often set batch size as power of 2 for GPU/CPU memory requirements e.g. 32, 64, 128, 256. Common batch size is 32. **N.B. 'SGD' (stochastic gradient descent) term is often used to refer to mini-batch GD even though SGD can also refer to single-example online GD.**

Pros:

a) Higher update frequency than full-batch GD > avoid local minima

b) Updates are done in mini-batches rather than per-example > more efficient than SGD

c) Don't need to store whole batch in memory as in full-batch GD > more scalable

Cons:

a) 'Mini-batch size' becomes an additional hyperparam to tune

b) Like full-batchh GD, loss info accumulates across mini-batches > accumulation of these losses is complex

- Disadvantages of above 'vanilla' gradient descent algorithms:
  1. Choosing learning rate can be difficult; too small leads to slow convergence, too large can cause oscillations of loss around optimum or even divergence
  2. Learning rate schedulers that use e.g. annealing (reducing learning rate according to pre-defined 'schedule' or when improvement in objective falls between a threshold) still need pre-defined schedule which is unable to adapt to individual data set characteristics
  3. Vanilla GD algorithms use same learning rate for all NN parameters. If data is **sparse** (i.e. some features have very different frequencies of occurrence in the training data set), may want to apply larger updates for rarely occurring features
  4. **Saddle points** (points in loss function where 1 dimensions slopes up and another slopes down with a surrounding cost plateaus) have close to 0 gradient in all dimensions, which is hard for vanilla GD algorithms to escape > never find true global minimum.
- Some popular more advanced gradient descent algorithms to deal with the above challenges:
  1. Momentum
  2. Nesterov Accelerated Gradient
  3. Adagrad
  4. Adadelta
  5. RMSprop
  6. Adam
  7. AdaMax
  8. Nadam
- Some additional strategies used alongside the above algorithms for optimising gradient descent include:

1. **Shuffling & Curriculum Learning**: Generally, don't want order in which we present training data to our model to bias it. Therefore, after each epoch, before doing another pass-through the training data set, often **shuffle** the data set randomly. Alternatively, if training NN to solve progressively harder problems, giving training data in meaningful order (e.g. easiest first) can improve performance & convergece; this is **curriculum learning**
2. **Batch Normalisation**: Process of scaling & offsetting the data coming out of a linear layer about to be put into an activation layer such that the *mean* of the input data is 0 and the *variance* is 1. Pros:
   a) Can worry less about how we initialise parameters of each layer
   b) Acts as a **regulariser** by adding some noise into data, which can prevent overfitting. **Con**: Normalises across the *mini-batch*, therefore the same image in different batch may get different prediction depending on what other images are in the mini-batch. To help with this, make the scaling and offsetting learnable features for the NN so that learns to minimise this affect. N.B. Batch normalisation is a common source of errors in NNs for this reason!
3. **Early Stopping**: Stop the training early if stop improving loss/error.
4. **Gradient Noise**: Sample noise from a Gaussian distribution and add this noise to each gradient descent parameter update > makes NN more robust to any poor parameter initialisation values, & helps model escape local minima (particularly helpful for deeper NNs where there are many local minima)
5. **Dropout for Regularisation**: Deep NNs can quickly overfit training data if don't have a very large number of examples. To help with overfitting, when doing a forward-propagation randomly ignore or 'dropout' some outputs (i.e. nodes/units) in a given layer > makes training process noisy, forcing nodes to take on more or less responsibility for the inputs

10.4.2. Computing the Gradient with Back-Propagation

- Up to now we have considered the process of **forward-propagation**: Take an input $\boldsymbol{x}$, propagate this information forward from the input layer through the hidden layers performing intermediate transformations on it, and finally produce an output $\boldsymbol{y}$, after which a scalar cost or loss $J(\boldsymbol{\theta})$ can be evaluated.
- We know that to optimise our model using this cost/loss, we often use gradient descent. As we saw above, gradient descent updates require the gradient of the loss L w.r.t. the NN's trainable parameters $\boldsymbol{\theta}$ (J is the Jacobian matrix, not the cost function as above; the cost function is synonymous with the loss function $L = J$):

$$\nabla J(\boldsymbol{\theta}) \equiv \Delta^{(\boldsymbol{\theta})} L = \frac{\delta L}{\delta \boldsymbol{\theta}} = \frac{\delta L}{\delta \boldsymbol{y}} \left(\boldsymbol{J}^{(\boldsymbol{\theta})}(\boldsymbol{y}) \right) \tag{10.21}$$

- And this gradient can then be plugged into a gradient descent algorithm to optimise our NN parameters:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \Delta^{(\boldsymbol{\theta})} L\left(\boldsymbol{\theta}, \left\{\boldsymbol{x}^{(i)}, y^{(i)}\right\}_{j=1}^{m}\right) \tag{10.22}$$

- However, numerically computing the gradient $\frac{\delta L}{\delta \boldsymbol{\theta}}$ in deep NNs can be computationally very expensive, since there are many NN parameters to consider.
- To numerically compute the gradient of the loss w.r.t. the NN parameters in an efficient way, we use **back-propagation**.

Background

- Back-propagation is an efficient way of numerically computing gradients
- Works by computing the **chain rule** with a specific order of **operations** (a function with 1 or more variables) that is highly efficient
- **Composite function:** A function with >=1 other variable(s) which is also a function (i.e. a function of a function)
- **Chain rule:** Formula for computing the derivative of a **composite function**. E.g. for real numbered scalar variable x, if $u = g(x), \mathrm{F}(x) = f\big(g(x)\big) = f(u) = y$, then the derivative of the composite function y w.r.t. variable x written with the chain rule is:

$$F'(x) = f'\big(g(x)\big) \cdot g'(x), \text{ or:} \tag{10.23}$$

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} \tag{10.24}$$

- **Multivariate chain rule**: Formula for computing the derivative of a **multivariate composite function**. E.g. for real numbered scalar variable t, 2 single variable functions $x(t)$ and $y(t)$, and a multivariate composite function $f\big(x(t), y(t)\big)$, then the derivative of the composite multivariate function f w.r.t. variable t is:

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} \tag{10.25}$$

  I.e. To find the derivative of a composite function, the composite function (f) is (1) differentiated w.r.t. each of its variables (x and y) (which themselves are functions) & multiplied by the derivative of that variable w.r.t. its own variable (t) (i.e. apply the **chain rule**), and then (2) all of the derivatives found by the chain rule are **summed** to get the derivative of the composite function f.

- **Product rule:** For $y(x) = u(x) \cdot v(x)$, then:

$$\frac{\partial y}{\partial x} = u\frac{\partial v}{\partial x} + v\frac{\partial u}{\partial x} \tag{10.26}$$

Computing Gradients/Derivatives in a Computation Graph

- Consider the following expression:

$$e = (a + b) \cdot (b + 1) \tag{10.27}$$

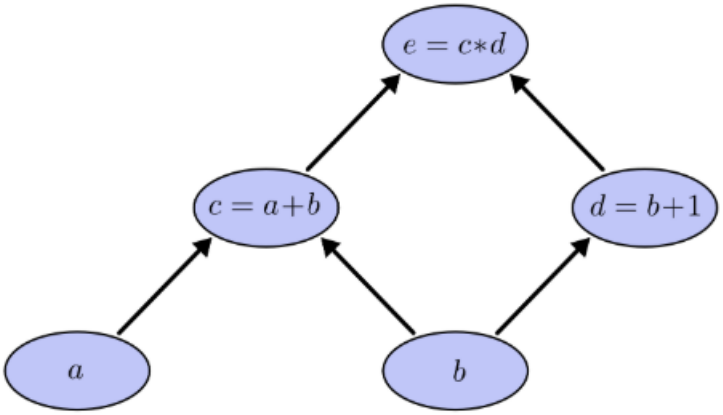
- This expression is made up of 3 ops: 2 additions and 1 multiplication
- Each op has its own output variable:

$$c = a + b \tag{10.28}$$

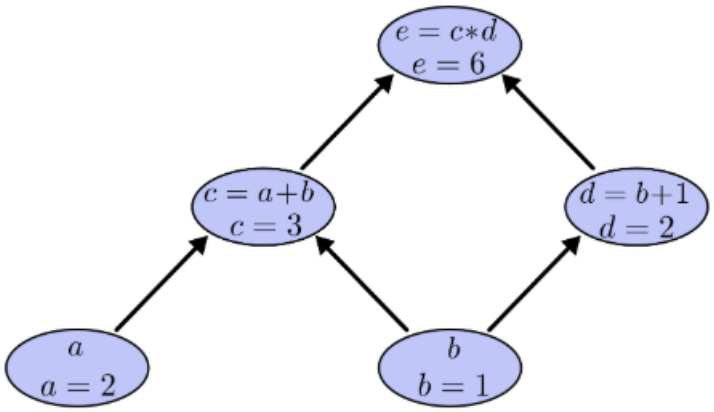
$$d = b + 1 \tag{10.29}$$

$$e = c \cdot d \tag{10.30}$$

- We can represent this expression as a **computation graph**, where each variable is a node and the dependencies of the ops used to generate the variables are represented with directed edges:



- To evaluate this graph, we give the inputs a and b and compute the output e. E.g. for $a = 2, b = 1$, we get output $e = 6$:



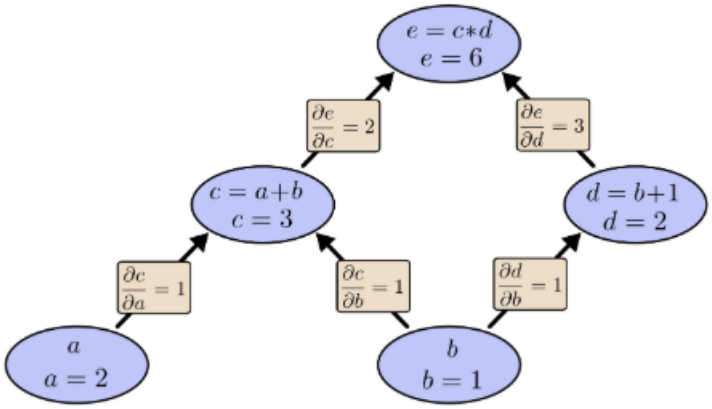
- E.g.1. Consider we want to find e.g. how a affects c (i.e. the partial derivative of c w.r.t. a, $\frac{\partial c}{\partial a}$). This is the derivative of the directed edge connecting a to c. Using the **sum rule**:

$$\frac{\partial c}{\partial a} = \frac{\partial}{\partial a}(a+b) = \frac{\partial}{\partial a}a + \frac{\partial}{\partial a}b = 1 + 0 = 1 \tag{10.31}$$

- E.g.2. Consider that we want e.g. how c affects e, using product rule:

$$\frac{\partial e}{\partial c} = \frac{\partial}{\partial c}(c \cdot d) = c\frac{\partial d}{\partial c} + d\frac{\partial c}{\partial c} = 3\frac{\partial d}{\partial c} + 2\frac{\partial c}{\partial c} = 0 + 2 = 2 \tag{10.32}$$

- Doing this for each edge in the computation graph for nodes/variables directly connected to one another:



- To find the relative gradient/derivative of nodes not directly connected to one another, there are multiple ops/functions that contribute to the gradient, therefore must use the **multivariate chain rule**.

- E.g. Consider how e is affected by b. $e(c,d)$ is a composite function of $c(b)$ and $d(b)$, therefore the derivative of e w.r.t. b is:

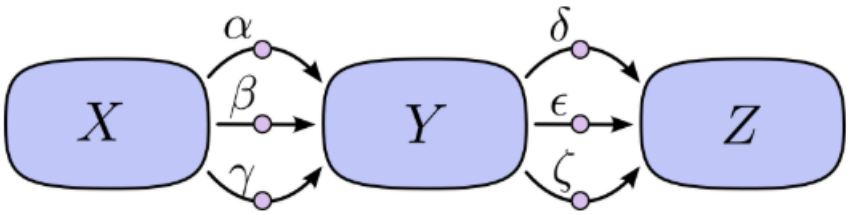
$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = (2 \cdot 1) + (3 \cdot 1) = 5 \tag{10.33}$$

  I.e. Just as before we apply the chain rule to the multivariate function and 'sum over paths' to get the derivative

Back-Propagation

- **Problem**: If we want to use this 'sum over paths' approach to using the multivariate chain rule in order to compute how each variable/parameter in a computation graph influences the output, the number of paths we must sum over grows exponentially with the number of possible paths.

- E.g. consider a simple graph with 3 nodes, X, Y, Z, 3 possible paths between X and Y, and 3 possible paths between Y and Z:



- To find the derivative of the output Z w.r.t. input X, we must sum over all paths, therefore we must sum over $3 \times 3 = 9$ paths.

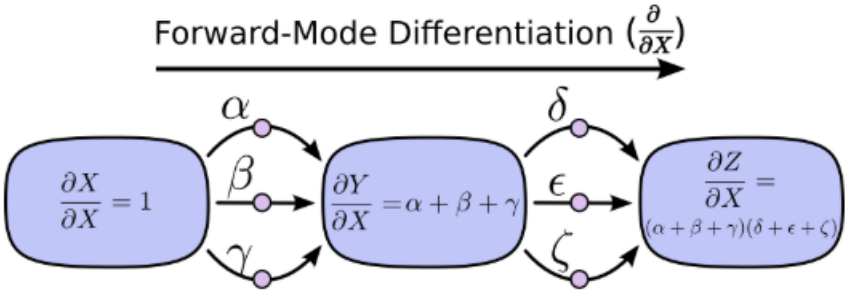
$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta \tag{10.34}$$

- With more complicated graphs, the number of paths to sum over quickly becomes intractable. Therefore, instead of summing over all the paths, it is much more efficient to compute the sum by **factoring** the paths to reduce the number of computations we need to perform to compute the gradient:

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) \cdot (\delta + \epsilon + \zeta) \tag{10.35}$$

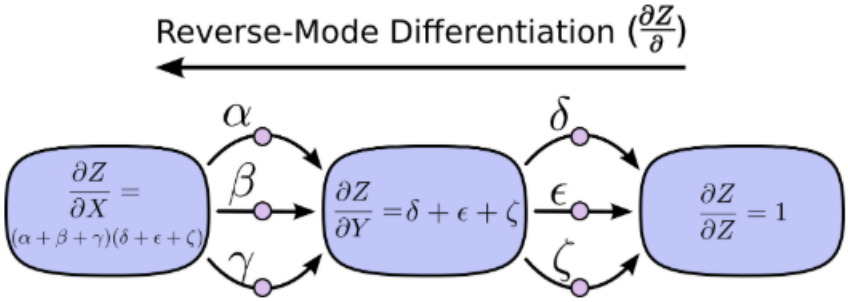
- There are two methods for computing the sum by factoring the paths:

1. **Forward-mode differentiation**: Start at the input X. Go forward through the computation graph, and at each node compute how the variable of the node (e.g. Y) varies with input X (i.e. compute $\frac{\partial Y}{\partial X}$ etc.) by summing all the paths feeding into the node. By the time you reach output Z, you will have found **how 1 input affects every node/variable by applying $\frac{\partial <Node>}{\partial X}$ to every node**. This is very similar to what we did above and what we do in school when using calculus.

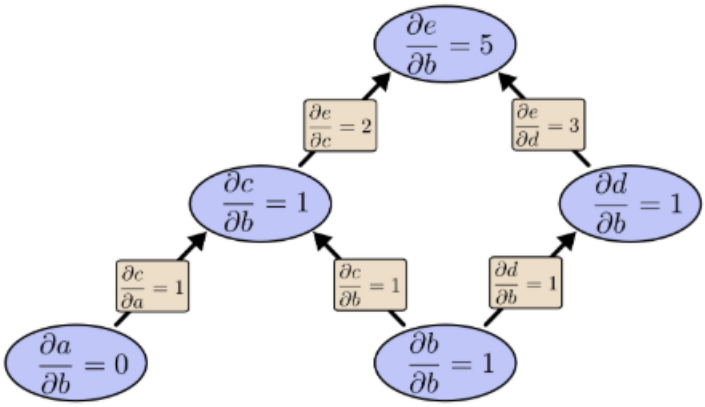


2. **Reverse-mode differentiation**: Start at the output Z. Reverse through the computation graph, and at each node compute how the computation graph output Z varies with the variable of that node (e.g. Y) (i.e. compute $\frac{\partial Z}{\partial Y}$ etc.) by summing all the paths that originate at that node. By time you reach input X, you will have found

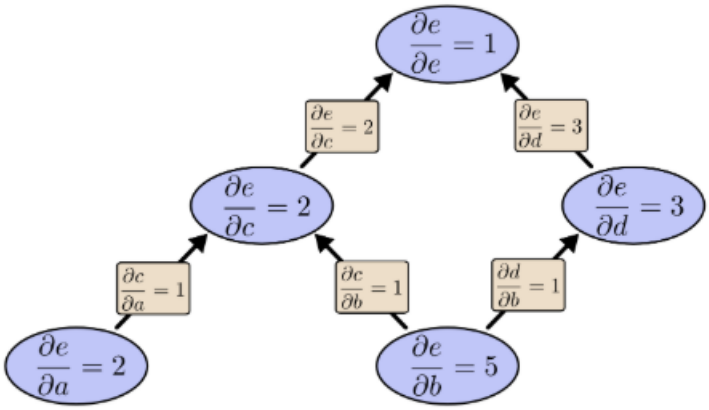
how every node/variable affects 1 output by applying $\frac{\partial Z}{\partial <Node>}$ to every node. This method is also known as **back-propagation**.



- E.g. Working up from b and applying forward-mode differentiation to find how 1 input b affects every node/variable in our computation graph:



- E.g.2. Working down from e and applying reverse-mode differentiation/back-propagation to find how every node/variable affects 1 output e:



- Therefore, while forward-mode differentiation gives us the derivative of our output w.r.t. a single input node (in our case, $\frac{\partial e}{\partial b}$), reverse-mode differentiation gives us the derivative of our output w.r.t. to *every* node, including all the inputs (in our case, $\frac{\partial e}{\partial a}$ and $\frac{\partial e}{\partial b}$) with the same number of summation steps.
- In our simple case, reverse-mode differentiation gives a factor of 2 speedup compared to forward-mode (since there are 2 input nodes). But, for a graph with e.g. 1 million inputs & 1 output, forward-mode would require us to go through the graph 1 million times to compute the derivatives (i.e. how each input affects the output), whereas reverse-mode differentiation only needs to go through graph once, giving a 1 million factor speedup!
- Since with NNs, we often want to find the derivative/gradient of the output's cost/loss w.r.t. millions of parameters, using reverse-mode differentiation/back-propagation is a very efficient way to find $\frac{\partial L}{\partial \boldsymbol{\theta}}$.
- I.e. we use back-propagation to compute the gradient of the loss w.r.t. the NN parameters, and we then use an optimisation algorithm such as gradient descent to use this gradient for learning by updating the NN parameters such that the loss is minimised.
- N.B. Forward-mode differentiation is more efficient for scenarios where the computation graph has few inputs and lots of outputs, since one sweep through the graph will give the derivatives of all outputs w.r.t. 1 input.
- N.B.2. Back-propagation can be used for computing any gradient, not just the gradient of the cost function w.r.t. the NN parameters.

10.5. Common NN Modules

- The following are common modules (layers/functions etc.) used for building NNs in e.g. TensorFlow

10.5.1. Linear Modules

1. Linear layers (i.e. what we saw at start of this chapter)
2. Convolution/deconvolution layers (see next chapter)

10.5.2. Basic Elementwise Ops

- Basic operations performed on each element in a vector/matrix/tensor:
  1. Add:
     - Forward propagation:

$$\boldsymbol{y} = \boldsymbol{a} + \boldsymbol{b} \tag{10.36}$$

   - Backward propagation:

$$\frac{\partial L}{\partial \boldsymbol{a}} = \frac{\partial L}{\partial \boldsymbol{y}} \tag{10.37}$$

2. Multiply:

   - Forward propagation:

$$\boldsymbol{y} = \boldsymbol{a} \cdot \boldsymbol{b} \tag{10.38}$$

   - Backward propagation:

$$\frac{\partial L}{\partial \boldsymbol{a}} = \boldsymbol{b} \cdot \frac{\partial L}{\partial \boldsymbol{y}} \tag{10.39}$$

10.5.3. Basic Groupwise Ops

- Basic operations performed on vectors/matrices/tensors (i.e. groups of elements)

1. Sum:

   - Forward propagation:

$$\boldsymbol{y} = \sum_i x_i \tag{10.40}$$

   - Backward propagation:

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{y}} \mathbf{1}^T \tag{10.41}$$

2. Max:

   - Forward propagation:

$$\boldsymbol{y} = \max_i x_i \tag{10.42}$$

   - Backward propagation:

$$\frac{\partial L}{\partial \boldsymbol{x_i}} = \frac{\partial L}{\partial \boldsymbol{y}} \tag{10.43}$$

3. Switch/conditional (denote 'active branch' using a one-hot vector $\boldsymbol{s}$):

   - Forward propagation:

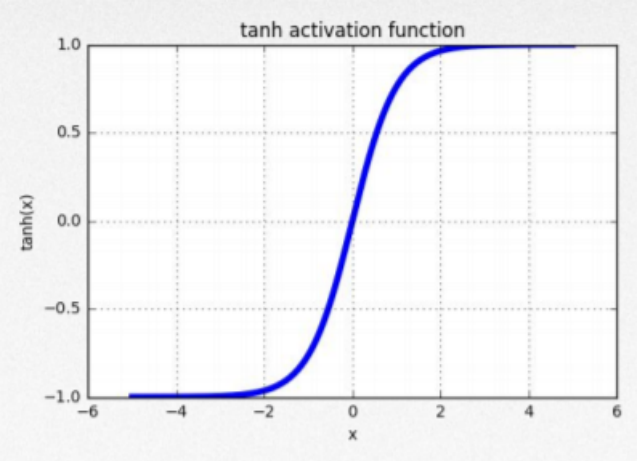
$$\boldsymbol{y} = \boldsymbol{s} \cdot \boldsymbol{x} \tag{10.44}$$

   - Backward propagation:

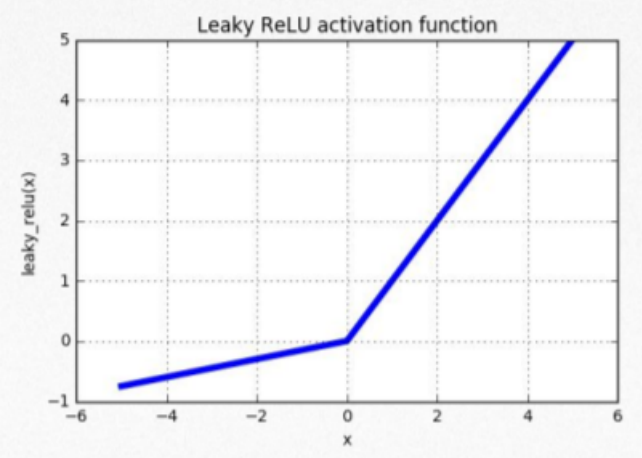
$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{y}} \cdot \boldsymbol{s}^T \tag{10.45}$$

10.5.4. Elementwise Non-Linear Functions (A.K.A. Activation Functions)

1. Sigmoid (see earlier)
2. Softmax (see earlier)
3. Relu (see earlier)
4. tanh (scaled and shifted version of sigmoid):



5. Leaky relu (overcomes possible issue with regular relu that gradients are 0 when inputs are inactive):



10.5.5. Basic Loss Functions

1. Cross-entropy (i.e. negative log-likelihood (NLP)) (see earlier)
2. Mean squared error relative to target $\boldsymbol{t}$ (use for regression problems):

– Forward propagation:

$$L = y = ||\boldsymbol{t} - \boldsymbol{x}||^2 = \sum_j (t_j - x_j)^2 \tag{10.46}$$

– Backward propagation:

$$\frac{\partial L}{\partial \boldsymbol{x}} = -2(\boldsymbol{t} - \boldsymbol{x})^T \tag{10.47}$$

11. Convolutional Neural Networks

- **Learning Resource(s)**:
  1. *'Neural Networks for Visual Recognition, Lecture 5: Convolutional Neural Networks', Serena Yeung, Stanford University, 2017,* https://www.youtube.com/watch?v=bNb2fEVKeEo&t=451s
  2. *'Deep Learning', Goodfellow et al., 2016,* http://www.deeplearningbook.org
  3. *'Deep Learning Lectures, Lecture 3: Convolutional Neural Networks for Image Recognition', Sander Dieleman, UCL-DeepMind Lecture Series, 2020,* https://www.youtube.com/watch?v=shVKhOmT0HE&list=PLqYmG7hTraZCDxZ44o4p3N5Anz3lLRVZF&index=4&t=0s
  4. *'Conv Nets: A Modular Perspective', Christopher Olah, blog post, 2014,* https://colah.github.io/posts/2014-07-Conv-Nets-Modular/
  5. *'Understanding Convolutions', Christopher Olah, blog post, 2014,* https://colah.github.io/posts/2014-07-Understanding-Convolutions/
- So far, we have seen that a *layer* in a NN is defined by the transformation it performs on the data $\boldsymbol{x}$ we put into it
- E.g. A *linear layer* performs a *linear* transformation (matrix multiplication (& addition if have bias value(s)):

$$\boldsymbol{y} = \boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b} \tag{11.1}$$

- E.g.2. A *non-linear layer*, such as a sigmoid activation function layer, performs a *non-linear* transformation:

$$\boldsymbol{y} = \sigma(\boldsymbol{x}) \tag{11.2}$$

- **Convolutional layer**: A layer which performs a **convolution operation** on the input data:

$$\boldsymbol{y} = \boldsymbol{W} * \boldsymbol{x} + \boldsymbol{b} \tag{11.3}$$

  Convolution is a linear/affine transformation.

11.1. The Convolution Operation

- Consider that we are using a laser sensor to track the location of a spaceship

- The laser outputs a single value $x(t)$; the position of the spaceship at time t
- The laser sensor is a bit noisy/unreliable, therefore rather than taking $x(t)$ to be the true position of the spaceship at time t, we want to take the average of several measurements to try to 'smooth over' the noise in the measurements. This will give us a less noisy position at time t
- If we take the simple mean of all measurements up to time t, our average position will be outdated and we will not get an accurate measurement of the current position. Therefore, we instead take the *weighted* average, where more recent measurements are more relevant and therefore contribute more to our average current position
- We do this by applying a *weighting function* $w(a)$ which, for each measurement, takes the age of the measurement a and returns some weighting factor. Multiplying each position by its corresponding weighting factor, summing, and dividing by the total number of positions gives the smoothed estimate of the position of the spaceship:

$$s(t) = \frac{1}{m} \sum_{i=1}^{m} x_i(a) \cdot w_i(t-a) \tag{11.4}$$

- In continuous form, this is equivalent to:

$$s(t) = \int x(a) \cdot w(t-a) da \tag{11.5}$$

- This operation is called a **convolution**. I.e. it is the convolution of x and w at point t. This operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t) \tag{11.6}$$

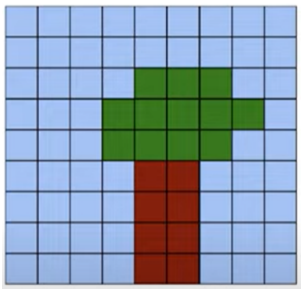
11.1.1. Common Terminology for Convolutions

- **Input**: The **first argument** of the convolution (in the above example, the function $x(a)$). For NNs, is usually a multi-dimensional array/tensor of data.
- **Kernel** (a.k.a. **filter**): The **second argument** of the convolution (in the above example, the function $w(a)$). For NNs, is usually a multi-dimensional array/tensor of tuneable parameters/weights. The kernel should always have the same number of dimensions as the input data (e.g. if the input data is a 2D image, the kernel should be 2D)
- **Feature map** (a.k.a. **activation map**): The **output** of the convolution (in the above example, the function $s(t)$)

11.2. Why Convolution Helps

- Images (and other types of data, including graphs, speech audio, textual data etc.), have 3 important properties:
    1. **Locality**: Nearby pixels tend to be strongly correlated with one another (e.g. for a bird flying in the sky, most of the important pixels are focused around the bird and are strongly correlated with each other (i.e. 1 section makes up wing, another head, collection makes up bird etc., and are not correlated with e.g. pixel on other side of image making up part of sky)).
    2. **Translation invariance**: Meaningful patterns (e.g. the 'pattern' of a bird flying in the sky) can occur anywhere in the image.

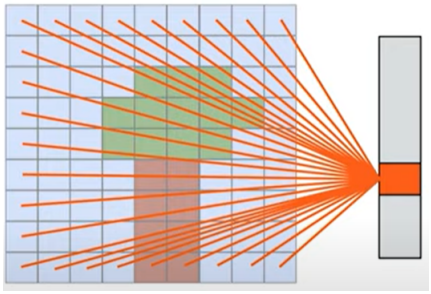
    3. **Hierarchical**: Images tend to be built up with hierarchical detail; at the low level, images are sets of edges, then zooming out a bit to a higher level might reveal a head, and zooming out further might reveal a head with wings and a body, thereby building up to the 'bird' object
- Convolutional layers are able to leverage these 2 key properties better than traditional fully-connected/dense NN layers (explained below).
- Consider a simple image of $9 \times 9 = 81$ pixels representing a tree (N.B. Each pixel would usually also have 3 values (RGB) associated with it, therefore would have $9 \times 9 \times 3 = 243$ values, but for simplicity for this example will ignore RGB values):



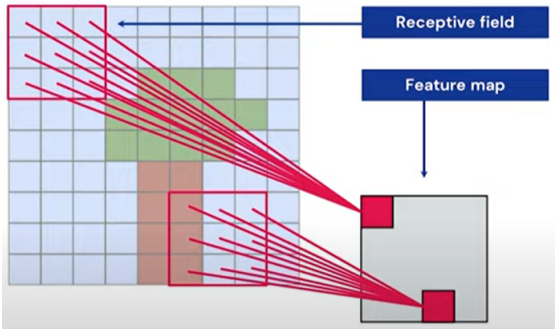
- Convolution leverages 4 key ideas which help NNs w.r.t. learning over different types of data (e.g. images, audio, text, graphs etc.):
    1. Sparse interactions
    2. Parameter sharing
    3. Equivariant representations
    4. Hierarchy

11.2.1. Sparse Interactions

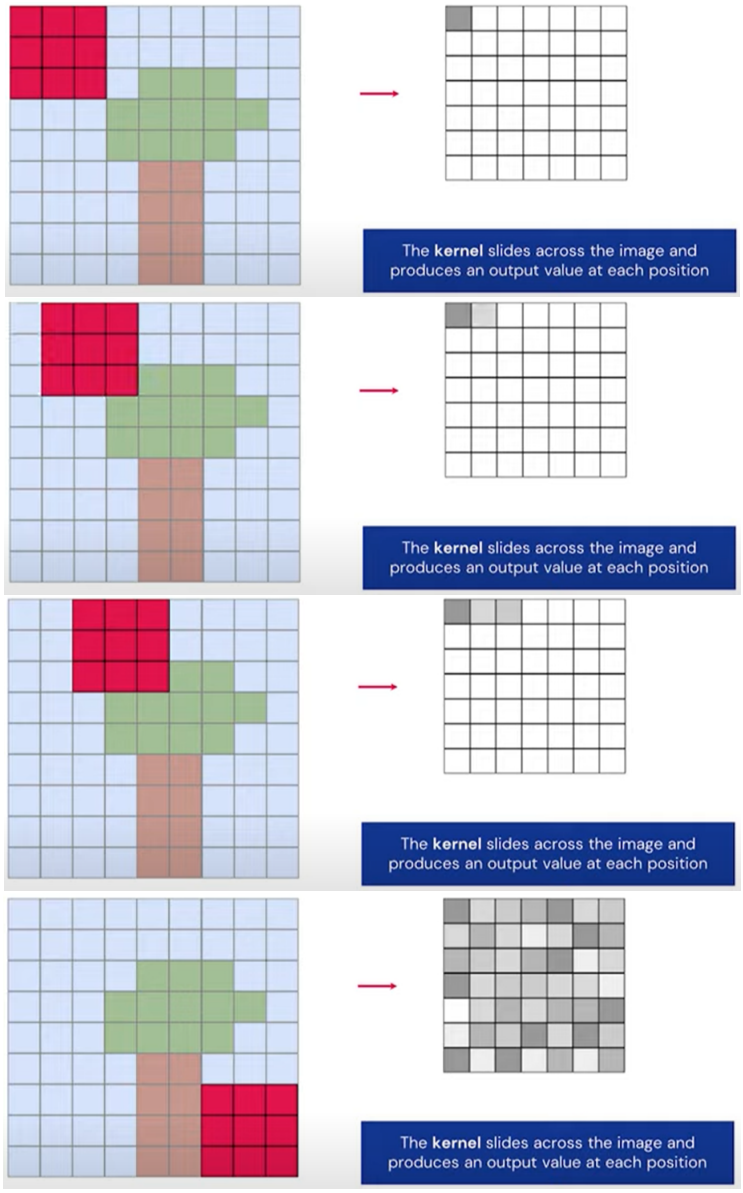
- For a traditional **fully-connected** (every input datum/input unit is connected to every output unit) linear layer, the output of each unit would be the linear transformation of all 81 data points in the image w.r.t. the unit's weights $\boldsymbol{w}$ & bias b, $y = \sum_{i \in \text{image}} \boldsymbol{w}_i \boldsymbol{x}_i + b$:



- Fully-connected layers therefore do not leverage the locality of the input data i.e. they consider every data point rather than just local sections of data.
- As mentioned previously, a convolution operation uses a *kernel* (a.k.a. filter). In the case of CNNs, the kernel/filter is a set of parameters/weights. The kernel should have the same number of dimensions as the input data (i.e. should be 2D for a 2D image input). Convolutional layers typically have **sparse connectivity** (a.k.a. sparse connectivity or sparse weights) by making the kernel *smaller* than the input
- To make the kernel smaller than the input, each output unit in the CNN layer only takes a set of locally-connected inputs from part of the data being input into the layer. The locally connected input data/units of a certain section of the image are connected to a certain value in the CNN layer **feature map's** (a.k.a. activation map) output. This set of locally-connected units of the input units/data are known as the **receptive field** of the output unit/feature map value they are connected to, which we get by **convolving** the receptive field (the input) with the kernel (i.e. the CNN layer's weights & biases). E.g. our 9×9 image/input units being locally connected to a 4×4 set of outputs values/feature map, where each output value/unit in the feature map has a 3×3 receptive field input and has an output $y = \boldsymbol{w} * \boldsymbol{x} + b$ (the below examples demonstrates convolving the kernel $\boldsymbol{w}$ with 2 different receptive fields to produce 2 output values in the feature map as an example):



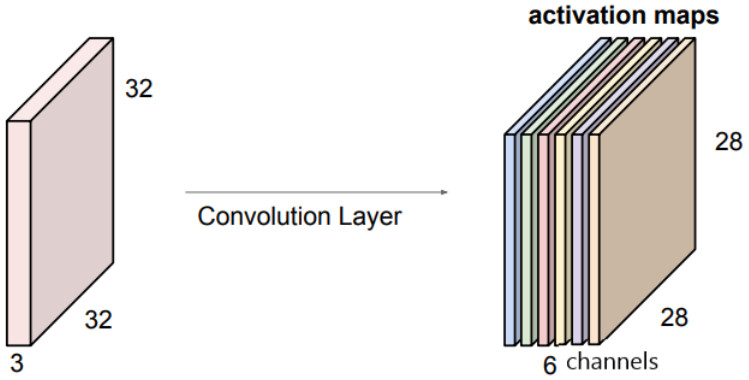
- To get an output value for each receptive field, we 'slide' the kernel across the input image's receptive fields, convolving each receptive field with the kernel. The output values of this convolution process collectively form the **feature map**. The feature/activation map is an abstract representation of the input data, and has the same number of dimensions as the kernel/input data (in this case, 2D). If the output value of the the kernel (plotted as a 'pixel' in the below activation map) was high (i.e. the output unit/feature map value was activated), this suggests that the 'feature' (e.g. a certain edge) that that particular output unit is detecting, was detected in the receptive field input to that particular unit:



- N.B. Images are usually RGB, therefore input data is of size e.g. $9 \times 9 \times 3$, and the kernel

will be of size e.g. $4 \times 4 \times 3$, where feature map value for each colour is e.g. averaged to get one output value for that kernel unit (which will go into the feature map).

- In practice, we usually have *multiple* kernels in our CNN sliding across the input data so that we can learn to represent more features in each local receptive field. E.g. if we have 6 $5 \times 5 \times 3$ kernels sliding across a $32 \times 32 \times 3$ RGB image, we'll get 6 feature maps which collectively form a 'new' abstract representation/image/feature map/activation map (in this case, of size $28 \times 28 \times 6$). Each of the activation maps (both the 3 activation maps from the RGB kernel which are eventually combined, and for each of the multiple kernels being slid across the input data) in the collective output is referred to as a **channel**:



- Key point is that, because the size of the kernel is smaller than the input image, each value in the output feature map represents a *local collection* of input pixels, and therefore leverages the locality of the input data.

11.2.2. Parameter Sharing

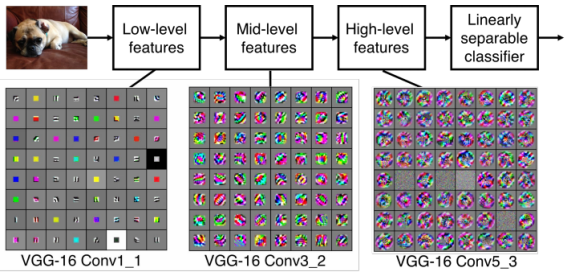
- In a traditional fully-connected/dense NN layer there is a weight matrix $\boldsymbol{W}$, where each vector element of the matrix is a vector of weights of one of the units in the layer. Each of these weights is used exactly once on a given example (e.g. an image), and then never re-used during processing of that example
- In a convolutional NN layer, the parameters/weights of the kernel are used on every receptive field of the input data as the kernel is slid across the input data, convolving all its weights with every receptive field as it goes. Therefore, the parameters/weights of the kernel are *shared* across the receptive fields when calculating the output unit values/the feature map.
- Parameter sharing allows CNNs to store far fewer parameters than fully-connected NNs must store (since rather than learning a set of parameters for each pixel/location in the input image, the NN layer only learns one set) > much more computationally efficient

11.2.3. Translation Equivariant Representations

- **Equivariant function**: A function for which if the input changes, the output changes in the same way.
- Convolutions are equivariant to translations in the input data. E.g. if we move an object in an image by 10 pixels to the right, then the position of the object in the output of the convolution operation will also move by an equivalent amount to the right.
- Therefore, convolutions also leverage the translation invariance of images.
- N.B. Convolutions are *not* equivariant to other types of transformations (e.g. scaling or rotating the object in input image).

11.2.4. Hierarchy

- A single CNN layer uses a kernel/filter to create a higher level filtered abstract representation of the input data/image
- If we use a second CNN layer, the use of another kernel/filter will create another higher level representation
- Therefore by using multiple CNN layers, each layer generates increasingly high-level representations of the input data. Since images are hierarchical, this is useful for learning to go from e.g. edges at the low level to e.g. a dog at the high level



- I.e. The lower/first layers of a convolutional NN model extract low-level pieces of info such as edges in images or high frequencies in audio clips, and the upper/layer layers extract high-level info such as faces in images, text in audio clips, or complex geometric shapes etc.

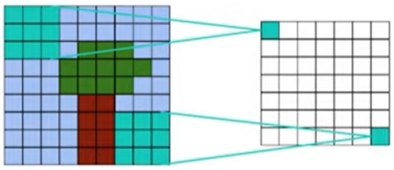
11.3. Common Convolution Implementations

- There are various methods that are used to implement convolution in NNs, which each differ in the size of the output feature map(s)
    1. **Valid convolution**: Simplest implementation. Only compute a feature map value where we are able to fully overlap the kernel with the input image. Con: as stack

valid conv layers on top of one another, feature map will shrink > can't build v deep CNNs > undesirable

$$\text{output size} = \text{input size} - \text{kernel size} + 1 \tag{11.7}$$

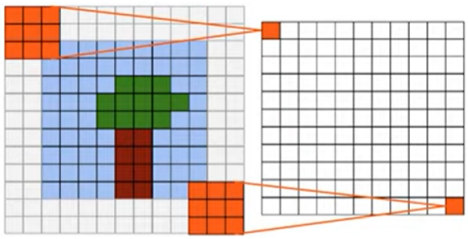
E.g. a 9×9 input image with a 3×3 kernel results in a $9 - 3 + 1 = 7$ (7×7) feature map output:



2. **Full convolution**: Computes a feature map value wherever kernel overlaps a pixel in the input image by ≥ 1 pixel. Do this by **padding** the edge of your input image with 0s. Will produce a feature map with more values than the number of values in the input image. Con: as stack full conv layers on top of one another, feature map will continue to grow > can't build v deep CNNs > undesirable.

$$\text{output size} = \text{input size} + \text{kernel size} - 1 \tag{11.8}$$

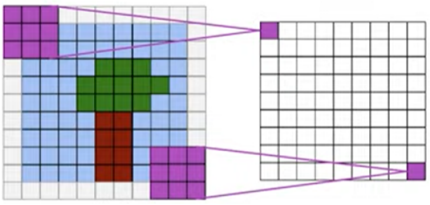
E.g. a 9×9 input image with a 3×3 kernel results in a $9 + 3 - 1 = 11$ (11×11) feature map output:



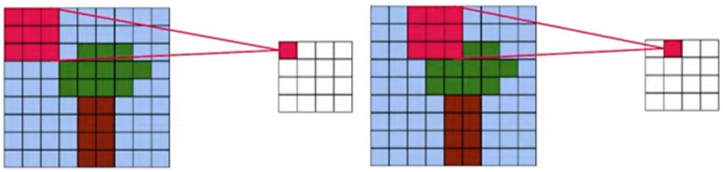
3. **Same convolution**: The most popular convolution approach in today's CNNs. Pads the input image with just enough 0s around the edge such that the feature map will have the *same* size as the input image. Allows us to stack many same conv layers > build v deep CNNs > desirable. N.B. To prevent asymmetric zero padding, need to have a kernel with an odd size, therefore almost all kernels in literature have odd kernel sizes.

$$\text{output size} = \text{input size} \tag{11.9}$$

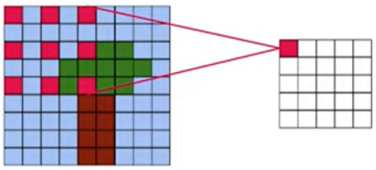
E.g. a 9×9 input image with a 3×3 kernel results in a $9 = 9$ (9×9) feature map output:



4. **Strided convolution**: Slide the kernel along the image in by more than 1 pixel between convolutions/receptive fields, which reduces the number of convolutions we need to perform and is therefore more computationally efficient:



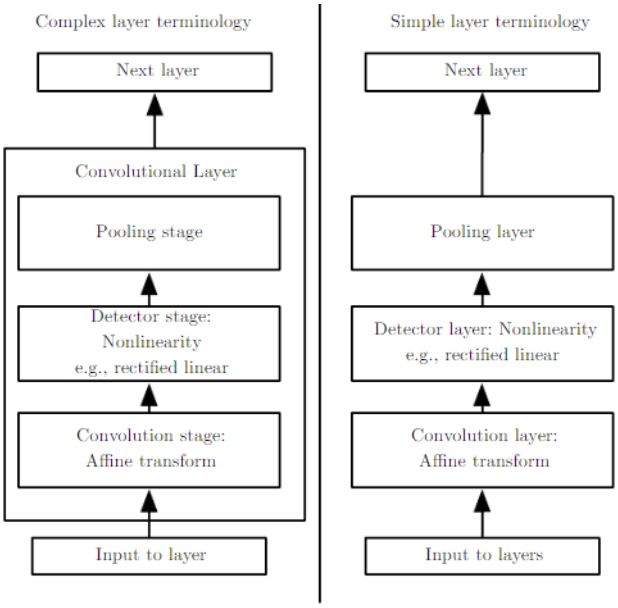
5. **Dilated convolution**: If want to expand the range of pixels that the receptive field of a CNN unit encompasses, rather than just increasing the size of the kernel (which is computationally inefficient), can 'dilate' the receptive field such that spaced out pixels form the input (and make all pixels between the chosen pixels equal to 0):



11.4. Pooling

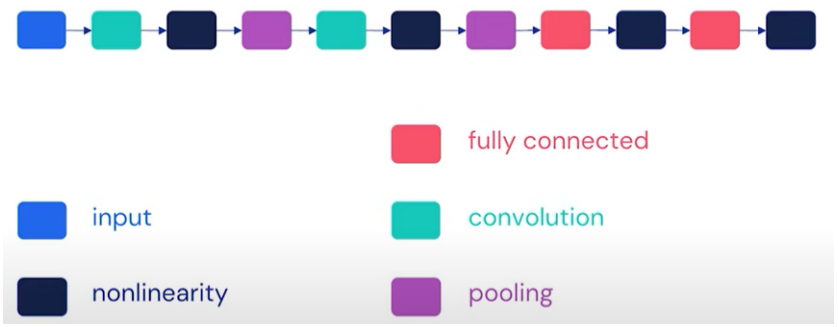
- Typically, when we refer to a CNN layer, the CNN layer has 3 stages/modules:
    1. **Convolution stage**: Use several kernels to perform several convolutions on the receptive fields in parallel, producing an activation map of linear activation values
    2. **Detector stage**: Run each linear activation value through a non-linear activation function (e.g. a relu function)
    3. **Pooling stage**: Run the activation function output values through a **pooling function**.
- I.e. Adding a detector stage and a pooling stage to the CNN layer still makes the CNN layer produce an output feature map just as we've seen previously, but the feature map will now be smaller/lower resolution than that output by the convolution stage (since pooling takes a group of neighbourhood values and outputs a single value).
- N.B. In literature, these 3 stages may be referred to as a convolution layer, or each stage may be referred to as a separate layer (common in modern computation graph

diagrams/terminology):



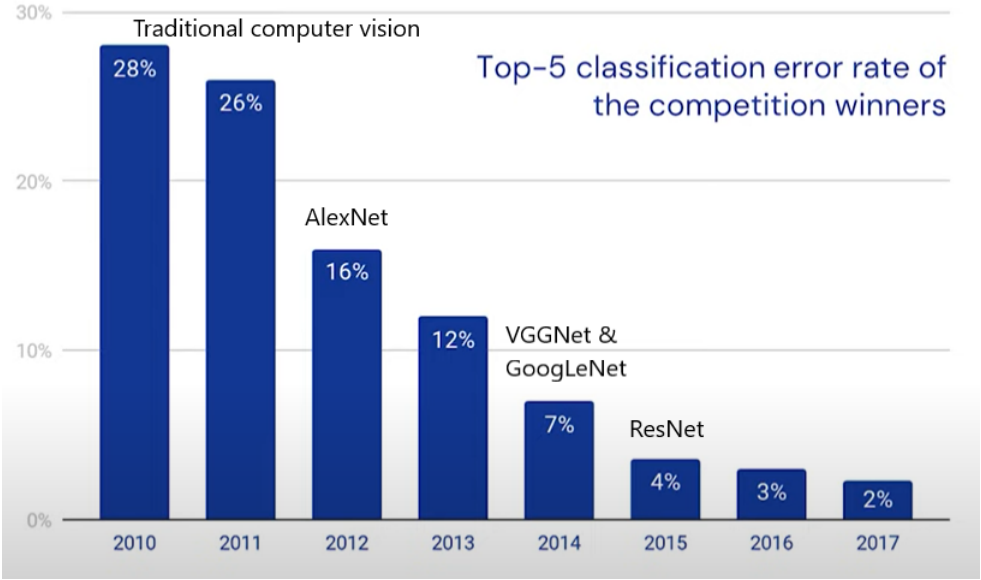
- **Pooling function**: A function which takes a group of nearby values & produces a single 'summary statistic' which summarises these values
- Common pooling functions used in CNNs:
    1. Max pooling (takes a set of values in a 'rectangular neighbourhood' and returns the maximum value)
    2. Average pooling (takes set of values in rectangular neighbourhood and returns average value)
    3. L^2 norm (takes set of values in rectangular neighbourhood and returns L^2 norm value)
    4. Returning weighted average based on distance from a central pixel in rectangular neighbourhood
- By using pooling (e.g. max pooling where take the maximum value across a neighbourhood of values), if the input feature being detected (e.g. a particular corner) is translated by a small amount (i.e. it's not in the exact original position that has been learned, but it is nearby such that its corresponding activation value is still present in the same rectangular neighbourhood of output values), then the output of the pooling function will be the same. Therefore, pooling helps to make CNN layers **invariant to local translations** such as (small) rotations & scaling of the input image, which is useful if we care more about whether some feature is present in an image rather than exactly where it is in the image.
- Typically there are up to 100s of convolution layers dispersed throughout a NN model.

E.g. of a simple CNN model with 2 convolution ops portrayed as a computation graph (for next few examples, will use this colour scheme):



11.5. Famous Convolutional Neural Network Case Studies

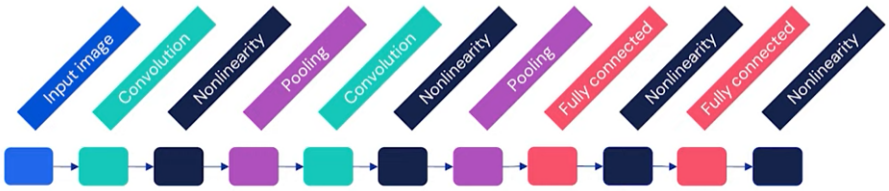
- ImageNet was a classification competition running from 2010 to 2017. Task was to learn to classify a database of 1.4 million labelled images with 1,000 possible classes.
- Summary of yearly winners:



11.5.1. LeNet-5 (1998)

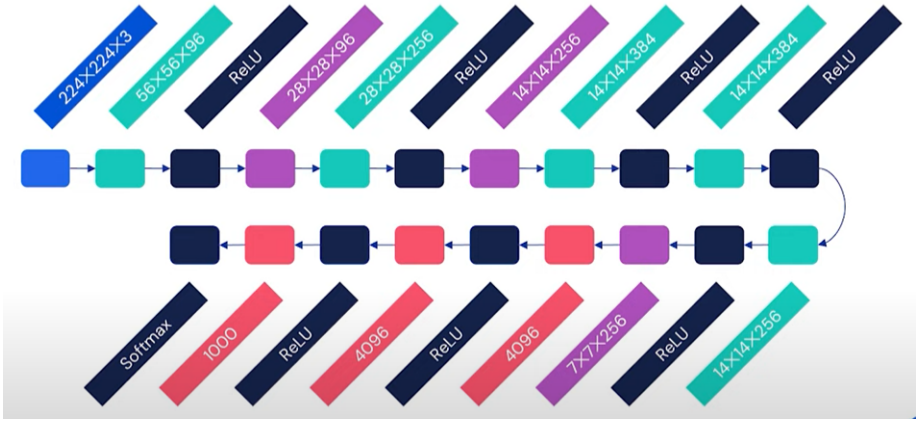
- Simple CNN used to classify handwritten digits
- Followed approach we've seen so far, where follow structure of **convolution | nonlinearity | pooling** for every convolution we want to perform in our model, and a

softmax activation function in the output layer to classify the input into 1 of 10 possible digits. CNN architecture:



11.5.2. AlexNet (2012)

- Breakthrough in image classification with CNNs. Was applied to ImageNet
- From this point on almost all NNs used CNNs
- Was a very large model. Split parameters of each layer into 2 halves, and trained each half on a separate GPU for 6 days (with modern hardware takes minutes!). Architecture of model was 8 layers (5 convolutional layers and 3 fully connected layers), where the non-linearity layers after the linear layers were relu layers. Also made use of **dropout** (randomly removing units during training) and **weight decay** (make sure size of parameters doesn't grow too much such that certain units become overly influential in output) to help with regularisation (prevent overfitting). Architecture:



- Key layers:
  1. Input layer: $224 \times 224 \times 3$ input RGB image (much larger/higher resolution images than CNNs before AlexNet has used, which usually used $32 \times 32 \times 3$ input images)
  2. Convolution (first layer): Used 96 11×11 kernels (note always odd number!) which performed *strided* convolutions with a (large) stride of 4 (i.e. 4 pixels between receptive fields), therefore reduced image size/resolution by a factor of 4 by producing an output feature map of size $56 \times 56 \times 96$, which in turn reduced number of pixels by

factor of $4 \times 4 = 16$, making the size of the input image tractable for rest of NN even though original image was so large/high resolution

3. Relu non-linearity (second layer): Produce non-linear value for each value in the convolution layer's output feature map
4. Max-pooling (third layer): Max-pooling used a window/rectangular neighbourhood input of 2×2 (i.e. took 2×2 grid of local non-linear feature map values and output a single summary (maximum value) statistic), thereby further reducing the image/data size to $28 \times 28 \times 96$.

   I.e. these first modules did the initial work to reduce a $224 \times 224 \times 3$ image down to size $28 \times 28 \times 96$ for rest of NN model, which is similar to what previous implementations of CNNs had handled as the original input; one of key innovations of AlexNet
5. etc etc....
6. Fully connected linear layer (eighth layer): Produced 1,000 outputs (1 for each possible class)
7. Softmax non-linearity (output layer): Output probability distribution (sums to 1) over all 1,000 possible classes

- Key innovations:
  1. Used initial 'compression layers' so that model could handle large/high resolution initial inputs
  2. Was first model to use relu function; before this, people thought that relu function wouldn't work because discontinuity/0 gradient would stop gradient-based optimisation from working (therefore gradient would not be definable everywhere). Turns out using relu actually helped gradient propagation throughout the model, since when activated the gradient of the relu function is constant therefore much more simple than e.g. sigmoid gradient > enabled building of much deeper NNs (which as we've seen previously leads to more powerful models). AlexNet was around 2x as deep as LeNet-5
  3. Was also first model to use dropout to help with regularisation/overfitting
  4. Was first model not to combine every convolutional layer with a pooling layer

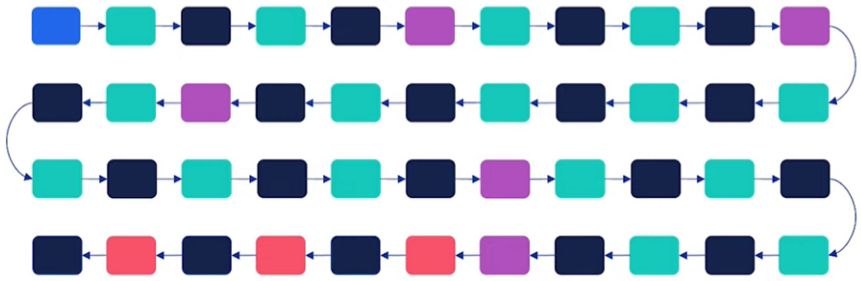
Depth Limitations

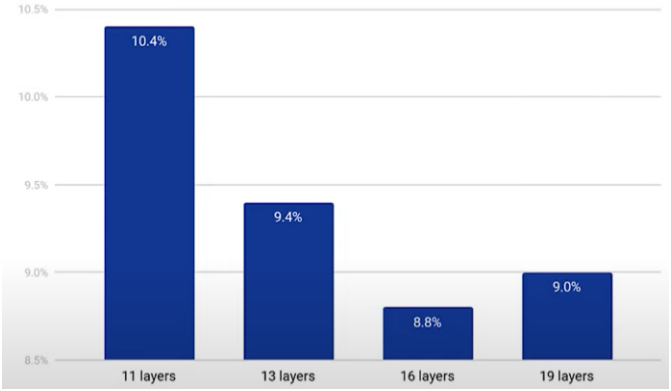
- So far we have seen that, by adding layers, we exponentially increase the number of linear regions that we can classify, and therefore significantly increase the power of the NN model. So what limits the number of layers we can use? Why was AlexNet 8 layers rather than e.g. 100? Answer:

1. More layers means more computation power needed and more training time (don't always have resources to do this)
2. As number of layers grow, becomes harder to know how each unit impacts final output (i.e. to perform backprop) and therefore becomes harder to correctly optimise all the parameters

11.5.3. VGGNet (2014)

- Was able to achieve ×2 depth of AlexNet
- Key innovations:
  1. Took idea from AlexNet further by stacking many convolutional layers on top of one another before doing any pooling and used **same** convolutions. These 2 factors were used to maintain the original image size/resolution throughout model where they didn't want a reduction in the resolution (i.e. resolution was only
  2. Unlike AlexNet, which used different size kernels at different convolutional layers, VGGNet used a constant 3×3 kernel for all convolutions. They used a small kernel, but used multiple kernels stacked on top of one another (performing same convolutions) so that had low number of total parameters to train whilst also being able to maintain large/high resolution images/large receptive fields, which allowed reduction in computational/learning complexity and therefore for a deeper NN model
  3. Unlike AlexNet, which used *model parallelism* (design NN architecture so that can split model across different GPUs and train in parallel), VGGNet used *data parallelism* (split data into batches and train model on parallel batches). Another type of parallelism: **pipeline parallelism**
- Architecture of model was up to layers with 3×3 same convolution kernels (trained for 2-3 weeks on 4 GPUs); tried 11, 13, 16, and 19 layers, where lowest error rate was at 16 layers (19 became too complex/too many par meters to accurately optimise/train):



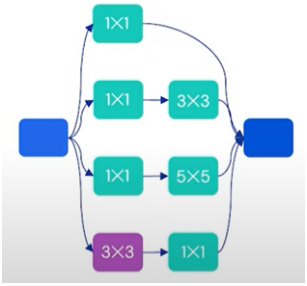


11.5.4. GoogLeNet (2014)

Optimisation in Deep Neural Networks

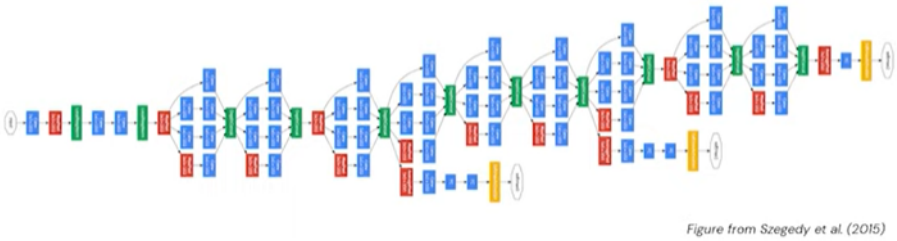
- **Optimisation** (i.e. how we work out how to update our NN weights/parameters) is a key area of research in deep NNs. Modern deep NNs use new optimisation techniques to allow even deeper NNs which are able to better assign credit to specific weights/parameters influencing output, therefore allowing accurate parameter updates that minimise loss
- 4 main approaches to improving optimisation in deep NNs:
    1. Careful initialisation of the parameter values such that don't get **exploding gradients** (where gradients of loss w.r.t. parameters keeps rising; occurs if parameters too large) or **vanishing gradients** (where gradients of loss w.r.t. parameters goes to 0; occurs if parameters too small). Various heuristics available for good initialisation of parameter values.
    2. Use of sophisticated optimisation algorithms
    3. Use of batch normalisation layers (scales activation layers so that values are in right range to make less sensitive to intialised parameter values, and also introduces some noise into data to help with regularisation (helps with overfitting))
    4. Good NN architecture design (e.g. connectivity pattern with residual connections (see ResNets later) such that gradient propagation is easier)
- GoogLeNet was winner of 2014 ImageNet (VGGNet came second)
- Key innovations (helped with optimisation process which allowed for very deep model):
    1. Used much more complicated model architecture than previous models, where branched out architecture so that had multiple convolution ops operating side-by-side. I.e. have multiple convolution ops with different kernel sizes and pooling functions/window sizes being performed in parallel. Referred to these as **inception blocks/modules**. E.g. of an inception block with different convolution opera-

tions with different size kernels and use of pooling (had multiple of these blocks throughout network):



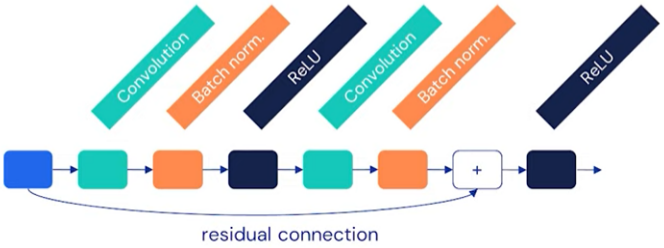
2. First model to use **batch normalisation** (see previous NN foundations chapter) to reduce sensitivity to initialisation and also help with regulariser (to stop overfitting). Batch norm made model converge much faster with much higher accuracy.

- Architecture:



11.5.5. ResNet (2015)

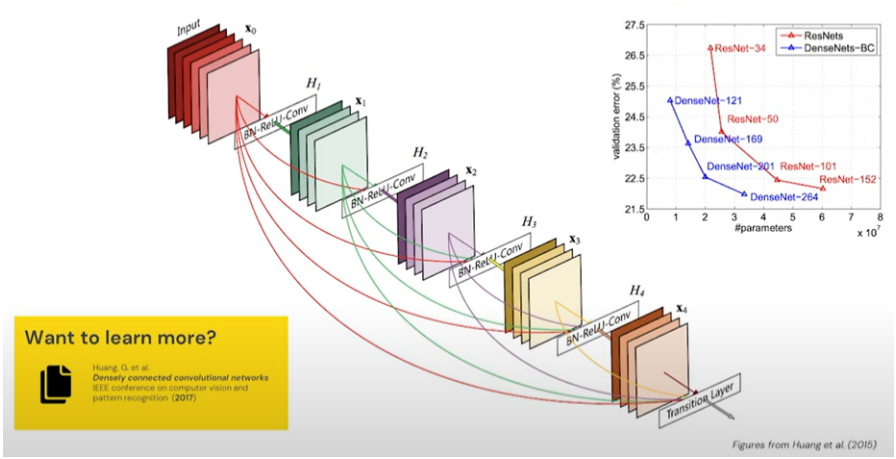
- If adding more layers is difficult because it makes back propagating the gradients through the network more difficult, why not 'skip' certain layers in the network when updating?
- Residual networks do this by adding a **residual connection**, which is a connection that allows the input data to bypass multiple layers in the model (i.e. we essentially 're-insert' the original data into the model by adding the values of the original input to the values of the output after e.g. a convolution layer deeper in the network). This allows back-propagation to go along the residual connection pathway and skip the layers bypassed by the residual connection. Example of a **residual block** that can be stacked many times:



- Residual connections allow the NN to have many more layers (since fewer parameters need to be updated than otherwise would be needed if didn't have residual connections). The ResNet that won in 2015 had 152 layers, which was almost an order of magnitude more than VGGNet or GoogLeNet from previous year!

11.5.6. DenseNet (2016)

- In DenseNet, also made backprop easier, but rather than using residual connections to skip layers, it connected every layer to every other layer in the network to make propagating the gradient easier
- Was able to build deep network that had lower validation error and fewer parameters than ResNet from previous year
- Architecture (i.e. still make up of batch-normalisation | relu | convolution ops/layers/blocks):

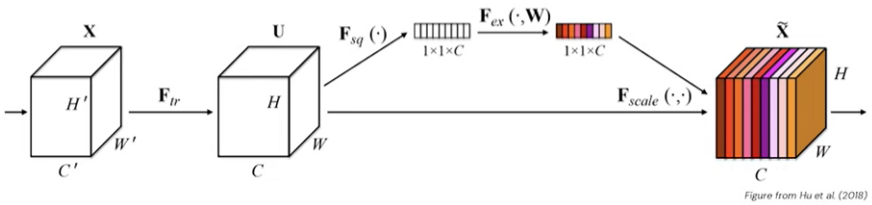


11.5.7. Squeeze-and-Excitation Networks (SENets) (2017)

‘

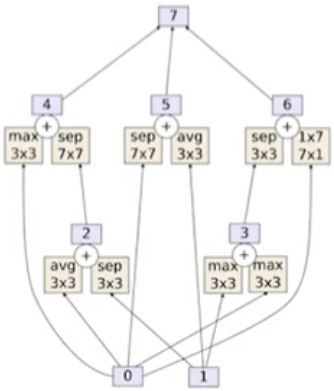
- This was the last year of the ImageNet competition before it was deemed solved
- Convolutions are very good at capturing *local* patterns, but less good at capturing the *global* context i.e. the stuff that's happening elsewhere in the image might give valuable context to the pattern you're trying to classify
- We know that a convolution layer typically works by having multiple kernels detecting different features in the same receptive field, the output of which are combined to produce a single feature map for that convolution layer. Typically, the convolution layer weights each of these kernels (or *channels*) equally to get the combined feature map
- SENets change this approach by *learning* how much each channel should be weighted when contribution to the final output feature map of the convolution layer

- Do this by:
    1. Take ((& save since we will use later) the feature map output by each channel/kernel
    2. Use *average pooling* to summarise each channel feature map into a single value
    3. Combine the summary value of each channel into a vector of length n, where n is the number of channels
    4. Feed the vector as input into a 2-layer NN (first layer is a dense/fully-connected linear op followed by relu op, second layer is a dense/fully-connected linear op followed by a sigmoid op) to get a vector output of weights (again of length n)
    5. Now take this vector of weights and apply each weight to its respective channel in the feature map when combining the channel feature maps into a single feature map being output by the convolution layer
- With training, the 2-layer NNs associated with each convolution layer in the CNN learn how to weight each of their channels/kernels when combining them such that loss is minimised. This effectively allows CNN model to learn global context as will weight different channels more or less strongly depending on context of image i.e. what other stuff is going on in the other channels/feature maps.
- Architecture:



11.5.8. AmoebaNet (2018)

- So far, we have seen how CNNs have been gradually improved by humans intricately changing the CNN architecture, making the architectures increasingly complex
- AmoebaNet was the first model to incorporate *neural architecture search*. This approach uses **evolutionary algorithms** to automatically search for the best NN architecture. The found architecture can then be trained to do e.g. image recognition
- E.g. Researchers pre-defined a set of acyclic computation graphs that were made up of pre-defined layers (e.g. a convolution layer here, a pooling layer there etc.), but let an evolutionary algorithm define how to connect these layers. Showed that evolutionary algorithm could find CNN models that, once trained, performed very well. E.g. of one of the architectures found:



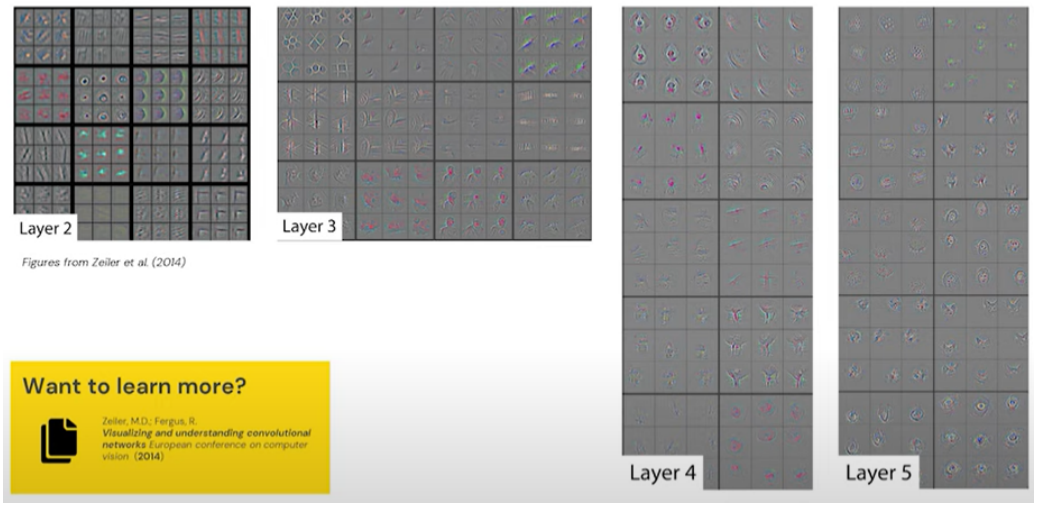
11.6. Advanced Topics

11.6.1. Data Augmentation

- We know that convolutions are translation invariant i.e. if we translate an object's position in an image, the output of the convolution will be translated by a corresponding amount
- *But* they are not invariant to other types of transformations such as rotation, scaling, shearing, warping etc. These types of transformations are very common in images (e.g. taking picture of same object but from different angle/position, picture on bright day vs. dark day etc.)
- To make CNN model robust/invariant to these types of transformation, we can **augment** our training data by taking the original image and applying each of these transformations with random perturbations to generate additional augmented images to train on. E.g. augmenting an image of a tree in a few different ways to increase CNN model robustness to these common transformations:

11.6.2. Visualising CNNs

- We can visualise what each convolution layer is detecting by using gradient descent; take the trained CNN model, apply different images as input (e.g. image of an edge, image of a hair pattern, image of a dog head etc.) and see which units are activated
- E.g. Below can see that layer 2 is detecting low-level features such as edges, whereas layer 4 is detecting high-level features such as a dog's head (top left):



- Can also visualise what specific objects CNN is looking for by only looking at the final output layer and searching for an image that generates the maximum activation response for each unit in the output layer. E.g. below are 6 images that each produced the highest response for 6 different units; these are not 'natural' images, but are what the CNN is 'looking' for:

11.6.3. Pre-Training & Fine-Tuning

- ImageNet has 1.4 million labelled images available, but many real-World scenarios have much less training data. Pre-training and then taking off the top output layer can reduce the total number of training examples you need.

11.6.4. Group Equivariant CNNs

- We saw that can use data augmentation to make CNN models learn to be invariant to transformations such as rotations

- Another approach to achieving this invariance is to use **group equivariant convolution neural networks**, where the models themselves are intrinsically invariant to these transformations

11.6.5. Recurrence & Attention

- Recurrence and attention are alternative methods to convolutions which take advantage of the grid-like topological structure of e.g. image data

11.6.6. Beyond Image Recognition

- There are various other areas that CNNs are being applied to in addition to image classification (because the areas benefit from the locality and transformation invariance that CNNs leverage):
    1. Object detection
    2. Generative models (e.g. **generative adversarial networks (GANs)**, **variational autoencoders**, **autoregressive models** (e.g. PixelCNN))
    3. Representation learning
    4. Self-supervised learning (e.g. how do we learn to classify images when the training dataset is unlabelled?)
    5. Video, audio, text, graphs, etc.

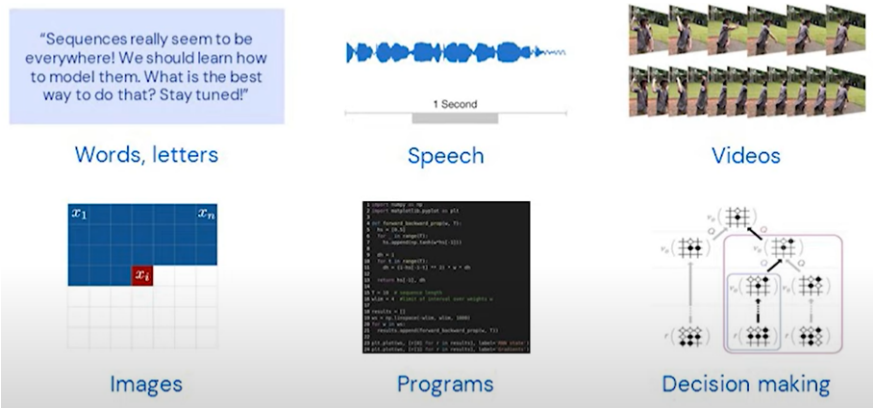
12. Sequence Modelling Neural Networks

- **Learning Resource(s)**:
  1. *'Neural Networks for Visual Recognition, Lecture 10: Recurrent Neural Networks', Justin Johnson, Stanford University, 2017,* https://www.youtube.com/watch?v=6niqTuYFZLQ&t=110s
  2. *'Deep Learning', Goodfellow et al., 2016,* http://www.deeplearningbook.org
  3. *'Deep Learning Lectures, Lecture 6: Sequences and Recurrent Networks', Marta Garnelo, UCL-DeepMind Lecture Series, 2020,* https://www.youtube.com/watch?v=87kLfzmYBy8&list=PLqYmG7hTraZCDxZ44o4p3N5Anz3lLRVZF&index=6
  4. *'Understanding LSTM Networks', Christopher Olah, blog post, 2015,* https://colah.github.io/posts/2015-08-Understanding-LSTMs/
  5. *'The Unreasonable Effectiveness of Recurrent Neural Networks', Andrej Karpathy, blog post, 2015,* http://karpathy.github.io/2015/05/21/rnn-effectiveness/
  6. *'Attention and Augmented Recurrent Neural Networks', Christopher Olah, blog post, 2015,* https://distill.pub/2016/augmented-rnns/
  7. *'A Ten Minutes Introduction to Sequence-to-Sequence Learning in Keras'*, Francois Chollet, The Keras Blog, 2017, https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-html
  8. *'seq2seq: The Clown Car of Deep Learning'*, Dev Nag, Medium blog post, 2016, https://medium.com/@devnag/seq2seq-the-clown-car-of-deep-learning-f88e1204dac3
  9. *'Attention? Attention!'*, Lilian Weng, blog post, 2018, https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html
  10. *'The Transformer Family'*, Lilian Weng, blog post, 2020, https://lilianweng.github.io/lil-log/2020/04/07/the-transformer-family.html
  11. *'Transformers are Graph Neural Networks'*, Chaitanya Joshi, TowardsDataScience blog post, 2020, https://towardsdatascience.com/transformers-are-graph-neural-networks-b

12.1. Why Sequences Matter

- So far, have looked at standard **feed-forward NNs** and **convolutional NNs**

- We have seen how CNNs are good for processing grid data with locality, translational invariance, and (usually) data of fixed input size
- However, CNNs are not good at processing **sequences** of data with **variable input sizes**
- **Sequence**: A collection of elements with 3 key properties:
  1. Elements can be **repeated**
  2. The **order** of the elements is important
  3. The number of elements in the sequence is **variable**
- Although some specialised CNNs can handle variable-sized inputs, they are unable to consider the context/order of elements in the sequence, therefore they cannot deal with sequential data
- Sequences are present in many areas of ML applications:



- Therefore, we want **sequence models** capable of processing and learning over sequential data.

12.2. Fundamentals of Sequence Modelling

- Previously for standard **feed-forward NNs**/CNNs, we have seen how we use a supervised learning approach whereby we take some data x with ground-truth labels y, and we train our NN model $f_\theta(x)$ to take the inputs x and learn the set of parameters θ that correctly produce the desired output y. We do this using some loss function L_θ and some optimisation algorithm which finds the optimum set of parameters θ^* that minimises the loss $L(\theta)$. Summarising:
  - **Data**:
  $$\{x, y\}_i \tag{12.1}$$
  - **Model**:
  $$y \approx f_\theta(x) \tag{12.2}$$

  - **Loss**:
$$L(\theta) \tag{12.3}$$
  - **Optimisation**:
$$\theta^* = \text{argmin}_\theta L(\theta) \tag{12.4}$$

- Sequence NNs use a slightly different approach. They take some sequence of inputs x (e.g. a sequence of words i.e. a sentence) and train a model $f_\theta(x)$ that learns to assign a probability $p(x)$ of that sequence occurring (e.g. learns to guess how probable a given sentence x is of occurring). The rest is similar (i.e. use a loss function and an optimisation algorithm for training to find the optimum set of parameters θ^* that gives us our desired output; the correct probability of a given sequence occurring). Summarising:
  - **Data**:
$$\{x\}_i \tag{12.5}$$
  - **Model**:
$$p(x) \approx f_\theta(x) \tag{12.6}$$
  - **Loss**:
$$L(\theta) \tag{12.7}$$
  - **Optimisation**:
$$\theta^* = \text{argmin}_\theta L(\theta) \tag{12.8}$$

12.2.1. Finding p(x)

- We want to find the probability $p(x)$ of a sequence x occurring
- E.g. Consider the following sequence of words (collectively forming a sentence):
$$\text{'Modelling word probabilities is really difficult'} \tag{12.9}$$
- There are a few different approaches we might consider to finding $p(x)$ for this sequence:
  1. **Independent probability approach**:
     - Naively assume each element in the sequence (i.e. each word in the sentence) is independent of one another. To estimate the probability of a word occurring, you would process a bunch of text and find the probability of each word occurring in the English language. You would find that the word 'the' occurs the most probable, therefore every word in the sentence you predict would be 'the':
$$p(x) = \prod_{t=1}^{T} p(x_t) \tag{12.10}$$

Most likely 6-word sentence:

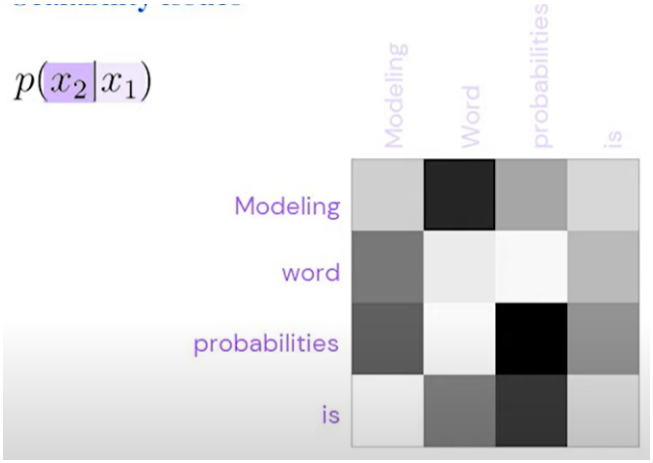
$$\text{'The the the the the the'} \tag{12.11}$$

- This naive approach doesn't work because languages/word sequences are **structured** whereby the elements in the sequence are not independent and their structure/order is important

2. **Conditional probability approach**:
   - Take the previous words in the sentence and predict the probability of a 'target' word conditional on the previous words that occurred in the sentence. I.e. x_1 = 'modelling', x_2 = 'word', ..., x_6 = 'difficult'. We would first predict the probability of the first word x_1, then the probability of the second word x_2 given x_1, then the probability of the third word given x_1 and x_2, etc...

$$p(x) = \prod_{t-1}^{T} p(x_t|x_1, ..., x_{t-1}) \tag{12.12}$$

   - This approach does capture the structure of the sentence and is fine if we have a small number of words to assign a probability to (e.g. if we are given x_1 = '*modelling*' and have to predict which word x_2 from a small list of 4 possible words ('modelling', 'word', 'probabilities', 'is') is most likely to occur next), since if have e.g. 4 possible words, only need a 4×4 conditional probability table to find the probabilities of each word given another word:



   - *But* if the number of possible words is very large (e.g. the whole English vocabulary with 10,000 words) this approach is **not scalable** since have very large number of conditional probabilities to calculate (would e.g. be a $10,000 \times 10,000$ table). If we also have to make predictions given more context (e.g. rather than just probability of 1 word occurring given 1 previous word, predict probability of a word occurring given 2, or 3, or 4 etc. previous words), constructing such a table of conditional probabilities becomes intractable.

3. **Conditional probability with N-grams approach**:
   - Do above, but rather than considering all of the context (i.e. all of the previous words/elements) set a horizon of N elements to look back at in the sequence. Elements beyond N are not considered:

$$p(x) \approx \prod_{t=1}^{T} p(x_t|x_{t-N-1}, ..., x_{t-1}) \tag{12.13}$$

   - E.g. could set $N = 2$ so that only consider previous 2 words in making prediction. This makes the number of conditional probabilities to calculate for our table a bit more tractable, *but* we lose a lot of the context, which is often very important for sequence modelling/prediction, therefore $p(x)$ predictions will often be inaccurate. Also, the conditional probability table to calculate is still very large. E.g. a study in 2006 set $N = 5$ and found that there were $1,176,470,663$ five-word sequences on the internet that occurred ≥ 40 times.

4. **Learning approach**:
   - Instead of these brute-force conditional probability table approaches, can we use sequence data to train a model $f_\theta(x)$ to **learn** how to take a sequence x and return the probability $p(x)$ of the sequence occurring in a more efficient way?

12.2.2. Modelling p(x)

- There are two stages a model needs to make predictions in sequence data:
  1. **Summarising the context**: Take in the context (i.e. the previous elements in the target sequence) and summarise this context into a vector.
     - Do this using a function $f_\theta(x)$, which takes in a sequence x and outputs some vector h which summarises the context of x
     - The h vector output by f_θ should then be able to summarise the context sufficiently well such that:

$$p(x_t|x_1, ..., x_{t-1}) \approx p(x_t|h) \tag{12.14}$$

     - f_θ must therefore:
       a) Be able to take in sequences x of variable sizes (since e.g. sentences do not all have the same number of words)
       b) Consider the order of the elements (words) in the sequence
       c) Be differentiable so that can use e.g. back-prop and an optimisation algorithm to learn the set of parameters θ that output correct the h

   d) Be able to learn that very small changes (e.g. changing just one word in a sentence) can completely change the context/meaning of a sequence

   e) Be able to retain long-term dependencies (i.e. the gap between important elements in a sequence that determine its context/meaning might be very large)

2. **Modelling conditional probabilities**: Take the vectorised summary of the context h, and use it to predict what the probability distribution of the next element is
   - Do this using a function $g_\theta(h)$, which takes the context h and produces a probability distribution over the possible next element(s) in the sequence given this context summary
   - g_θ must therefore:
     a) Be able to learn that very small changes in the context can have very significant changes in the probability distribution over the next element(s)
     b) Be able to return a probability distribution

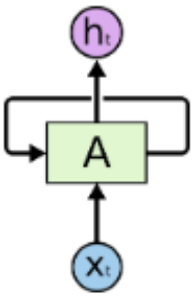
- The most important & successful sequence model has been the **recurrent neural network (RNN)**

12.2.3. Introduction to Recurrent Neural Networks

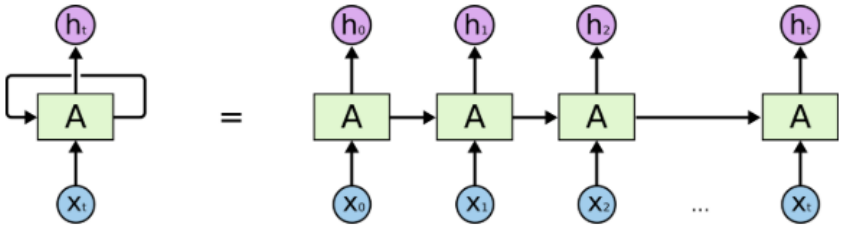
Quick Overview

- RNNs are good for processing **sequences** of data with **variable input size**. They've been very successful for e.g.:
  1. Speech recognition
  2. Language modelling
  3. Translation
  4. Image captioning
  5. Time series prediction (e.g. making predictions about the future) (e.g. predicting: closing price of stock each day, utilisation demand on a server each hour, number of passengers using a station each day, etc.)
- RNNs are a type of NN with a specific architecture. They have a **recursive loop** whereby information is repeatedly input to the *same* set of weights & biases (**parameter sharing**). I.e. an RNN can be thought of as being a string of multiple copies of the same network/NN block, where each copy/block passes its output to the successor block.
- This recursive loop in RNNs enables information to persist, which allows RNNs to process sequences since they can consider past elements in the sequence when reasoning about the current element

- E.g. consider a NN module A, some input data x_t, and an output of A h_t. An RNN has a loop in it whereby it keeps passing data into copies of the same NN block A:



- If we 'unroll' this loop, we see more clearly that RNNs are a string of multiple copies of the same NN block A, each re-using the same set of parameters:



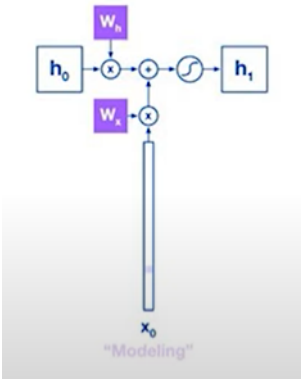
Looking Deeper

- Consider the sequence $\boldsymbol{x}$ of words from Eq. 12.9.
- Let $\boldsymbol{h}$ be a persistent state vector variable which stores the context of the sequence observed so far (i.e. it is a *hidden state* summarising the context of the input sequence seen so far by the model)
- At $t = 0$, $h_t = h_0$ is initialised as a vector of 0s (i.e. no context has yet been observed).
- We input the first element of our sequence, x_0, into our model (in this case, x_0 = 'modelling'. We want our model to predict what the next word in the sequence might be.
- As said previously, there are 2 steps for using our model to predict the probability distribution of the next element:

  1. Summarising the context:

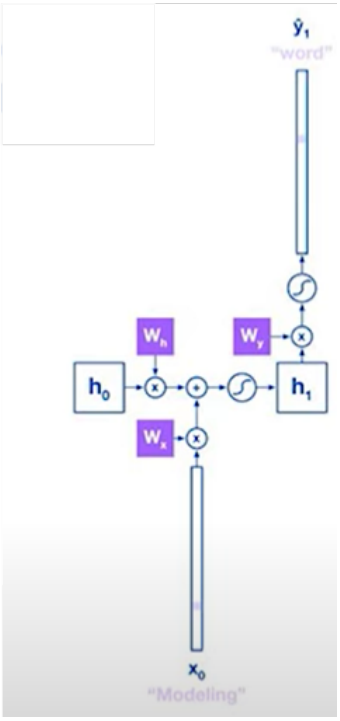
     - Apply a set of weights $\boldsymbol{W_h}$ to our (initialised) stored hidden context state variable h_0, and apply another set of weights $\boldsymbol{W_x}$ to our input sequence x_0. Do this by *matrix multiplication*. Then linearly combine these weighted matrices with *matrix addition*. Finally, use some activation function (e.g. tanh) to take this

linearly combined matrix and output a new updated hidden state summarsing the context of our sequeence given this lates x_0 input, h_1:



$$h_t = \tanh\left(\boldsymbol{W_h h_{t-1}} + \boldsymbol{W_x x_t}\right) \tag{12.15}$$

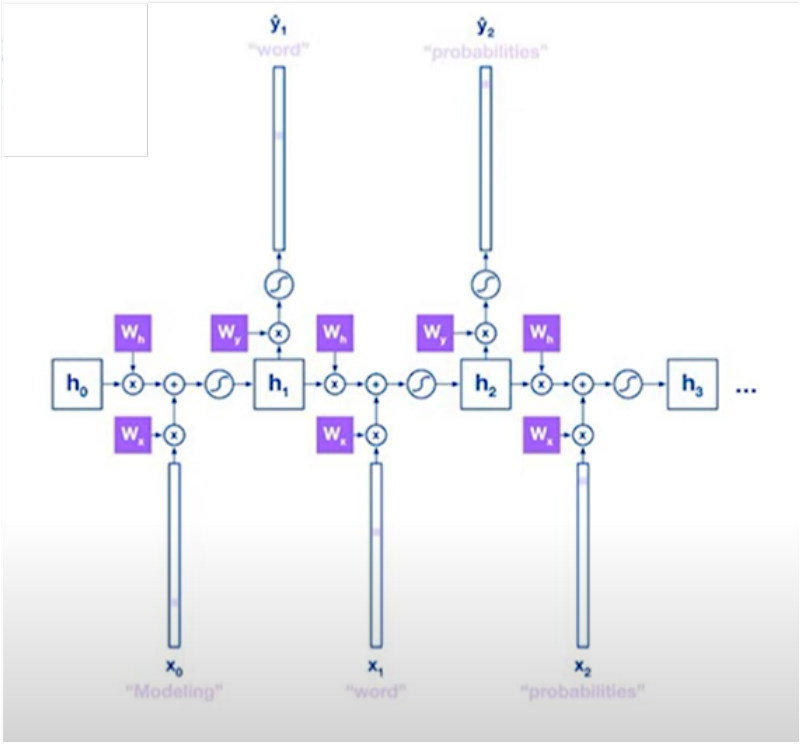
2. Modelling conditional probabilities:
   - Use the most up-to-date stored hidden state context summary h as input to e.g. a softmax function with weights $\boldsymbol{W_y}$ to produce a probability distribution over all possible target values y, where y is e.g. the next word in the sentence:



$$p(\boldsymbol{y}_{t+1}) = \text{softmax}(\boldsymbol{W}_{\boldsymbol{y}}\boldsymbol{h}_t) \tag{12.16}$$

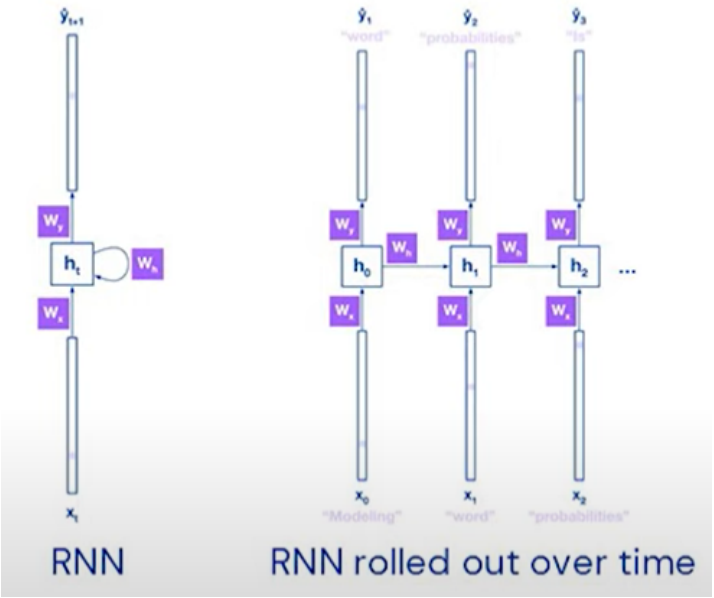
  - I.e. $\boldsymbol{y}_{t+1}$ is a probability distribution (sums to 1) over all possible words for the next word in the sequence given the context of the sequence. The predicted next word is that with the highest probability of occurring.

- This 2-step process is the fundamental building block of an RNN. Can repeat this 2-step process to repeatedly update the hidden context summary state h by giving each element of the sequence x into the model, and can do this as many times as we like:



- I.e. RNNs are able to connect previous information with present information when making predictions. This helps them to classify an event (e.g. what's currently happening in a movie, what's being said in a piece of text, etc.) in the context of previous events that occurred, which (most) CNNs are not able to do.

- I.e.2. Can summarise the above into a simple RNN diagram, which can also be rolled-out over time:



- Important to note that RNNs use the *same* sets of weight parameters $\theta = \{\boldsymbol{W_h}, \boldsymbol{W_x}, \boldsymbol{W_h}\}$ at each stage (parameter sharing); the only thing that is updated is the the hidden state $\boldsymbol{h}$ summarising the context, and the latest element(s) from the input sequence $\boldsymbol{x}$; the RNN uses these & the sets of weight parameters to predict the most likely next element(s) in the sequence $\boldsymbol{y}$
- To summarise, RNN build representations of each element in an input sequence *sequentially* (one element at a time). This is similar to a conveyor belt, **autoregressively processing** (i.e. using previously generated representations as extra input while generating the next representation) each element in a sequence from left to right. At the end, a hidden feature for each element in the sequence is output, which can be passed to next RNN layer etc. until finally get representation we desire (e.g. word prediction, image caption, language translation etc.)

12.2.4. Training Recurrent Neural Networks

Cross-Entropy Loss

- Predicting e.g. the next word in a sentence is a **classification** task, where the number of classes is the size of the vocabulary.
- As we've seen previously, the **cross-entropy** is the best loss function for classification tasks. In this case, we are taking the cross-entropy loss to be the cross-entropy between what the model predicted $\hat{\boldsymbol{y}}$ and what the actual answer was $\boldsymbol{y}$.

  For 1 word:

$$L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}})_t = -\boldsymbol{y_t} \log \hat{\boldsymbol{y}}_t \tag{12.17}$$

For whole sentence:

$$L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}})_t = -\sum_{t=1}^{T} \boldsymbol{y_t} \log \hat{\boldsymbol{y}}_t \tag{12.18}$$

- Therefore, to make our NN learn, we must optimise the set of parameters $\theta = \{\boldsymbol{W_h}, \boldsymbol{W_x}, \boldsymbol{W_h}\}$ such that the cross-entropy loss is minimised.

Differentiating

- As usual, to optimise our set of parameters θ such that loss L is minimised, we must find how the loss varies w.r.t. our parameters with differentiation, and then use some optimisation algorithm to update θ such that the loss is minimised
- Differentiating the loss L w.r.t. the parameters θ is a bit different to the differentiation we've seen with standard feed-forward NNs/CNNs due to the presence of the recursive loop; we must unroll over all the time steps rather than just differentiating once, since each time step/modular unit of the RNN re-uses the weights
- ML libraries (e.g. TensorFlow) take care of this for you

Vanishing Gradients: The Problem of Long-Term Dependencies

- Consider that, given part of a sentence, we want to predict the next word in the sentence given the context (i.e. the previous words)
- E.g.:

'The clouds are in the sky' (12.19)

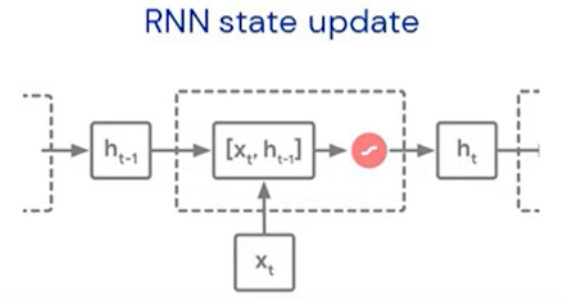
- In this case, the gap between the previous relevant information/context ('the clouds are in the') and the place we need to make a prediction (where 'sky' is) is small, which makes it easier for vanilla RNNs to make predictions
- E.g.2.:

'I grew up in France... I speak French' (12.20)

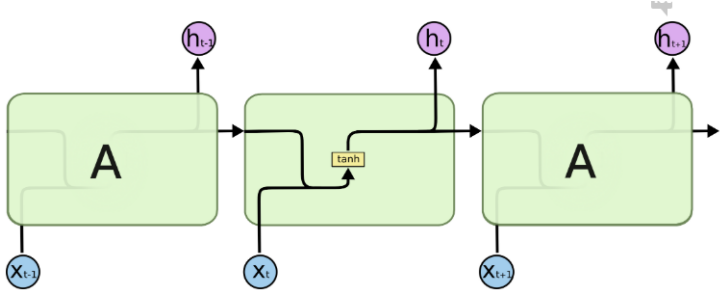
- If predicting the final word, the RNN might think based on the recent information 'I speak' that the word will be a language, but to narrow down *which* language, the RNN needs to look further back at the context of 'I grew up in France'. This gap might be very large, and as the gap grows, vanilla RNNs become unable to learn to connect the information as the gradients vanish/go to 0
- I.e. the larger the distance between relevant information in the chain of elements in a sequence being processed, the more probable the information will be lost in the RNN chain
- To solve this, a special kind of RNN called **long short term memory (LSTM)** networks are used, which are capable of learning long-term dependencies.

12.3. Long Short Term Memory (LSTM) Networks

- As we've seen, a 'vanilla' RNN takes a sequence input x_t and uses it with its stored context h_{t-1} to output/update its stored hidden context state to h_t:

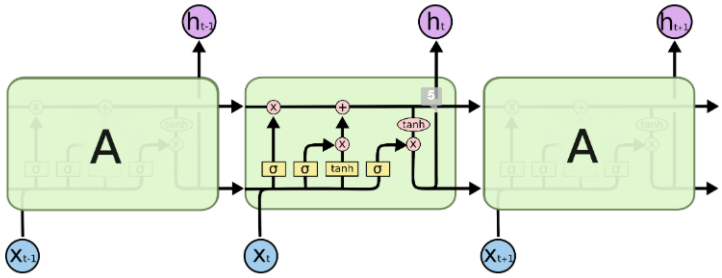


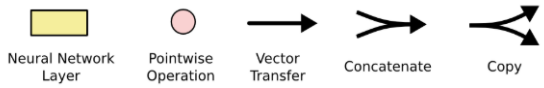
- This simple structure (made up of e.g. a single tanh NN layer) forming a module is repeated as a chain:



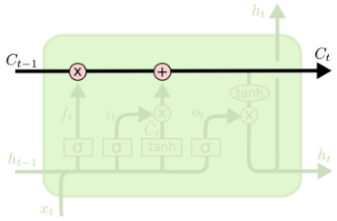
12.3.1. Vanilla LSTMs

- LSTMs are a particular kind of RNN for learning long-term dependencies & therefore helping with the vanishing gradient problem of RNNs. In reality, vanilla RNNs are almost never used, and usually use LSTMs (or some other variant).
- LSTMs also have this chain structure of a repeating module, but the module has a different structure. Rather than having a single tanh NN layer, each module in an LSTM has 4 layers interacting with one another: 3 sigmoid, & 1 tanh layer:

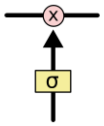




- Key idea behind LSTMs is the **cell state** C (top horizontal line running through the LSTM). The cell state can be considered as the 'long term memory' running through all of the LSTM modules, and is updated by each module:

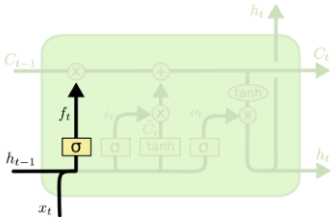


- The long-term memory/cell state C helps the LSTM to learn how to store important information in the long-term such that it can handle long-term dependencies. The long term memory C is updated by a series of **gating mechanisms**
- LSTMs have three **gates**, where each gate is a sigmoid layer and a pointwise multiplication operator; the sigmoid layer outputs a vector a numbers between 0 and 1, deciding how much of each component in the cell state C to allow through by multiplying C by the vector:



- I.e. the 3 gates in the LSTM module are used to control the cell state C
- There are 4 key stages in an LSTM module (which run from left to right):
    1. **Stage 1**: The first sigmoid layer (the **'forget gate'** i.e. the first gate) takes in the sequence input x_t and the hidden context state h_{t-1}n and outputs a vector f_t of numbers between 0 and 1 of length equal to the number of elements in the cell state C_{t-1}, which will be used to update the cell state. A 1 represents 'completely keep this element stored in the cell state', while a 0 represents 'completely forget this element'. E.g. in language if we encounter a new subject (e.g. a man), may want to 'forget' the gender of the old subject (e.g. a woman) by f_t outputting a 0 (or close to 0) for the corresponding element:

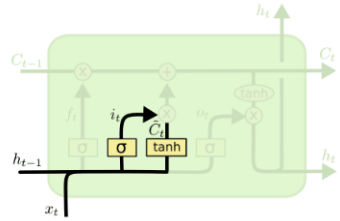
$$f_t = \sigma\big(W_f \cdot [h_{t-1}, x_t] + b_f\big) \tag{12.21}$$



2. **Stage 2**: The second layer (the **'input gate'**) is also a sigmoid layer (i.e. the second gate). It takes the sequence input x_t and the hidden context state h_{t-1} and outputs a vector of numbers between 0 and 1 which will decide which elements of the input sequence we want to use to update our cell state C. The third layer is a tanh layer which takes the sequence input x_t and hidden state h_{t-1} and and creates a new representation of them by generating a vector of values, where these values are 'candidate values' $\widetilde{C}$ (e.g. the gender of the new subject) that could be added to the cell state C:

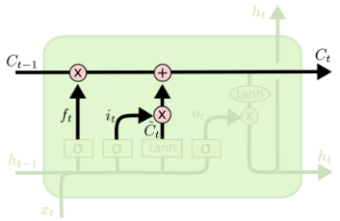
$$i_t = \sigma\big(W_i \cdot [h_{t-1}, x_t] + b_i\big) \tag{12.22}$$

$$\widetilde{C}_t = \tanh\big(W_C \cdot [h_{t-1}, x_t] + b_C\big) \tag{12.23}$$



3. **Stage 3**: Update the old cell state C_{t-1} to the new cell state C_t by:
   a) Multiplying C_{t-1} by f_t to forget old information we no longer want to keep
   b) Multiply i_t by $\widetilde{C}_t$ to get the new information we want to add (i.e. scale the candidate values by how much we want them to contribute to the new cell state), and add this to the cell state

$$C_t = \big(f_t \cdot C_{t-1}\big) + \big(i_t \cdot \widetilde{C}_t\big) \tag{12.24}$$

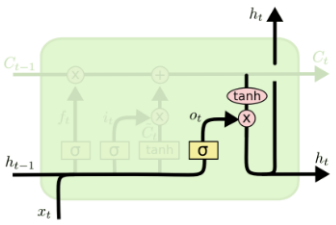


4. **Stage 4**: Decide what new hidden context state h_t to output. First use a sigmoid layer (i.e. the third gate) to output a vector of numbers between 0 and 1 which each decide how much of each element in the cell state C_t we are going to use to update our hidden context state h. Then put the cell state C_t through a tanh layer (to push

the values between -1 and 1) and multiply this output by the output of the sigmoid gate to get the new hidden context state h_t:

$$o_t = \sigma\big(W_o[h_{t-1}, x_t] + b_o\big) \tag{12.25}$$

$$h_t = o_t \cdot \tanh\big(C_t\big) \tag{12.26}$$



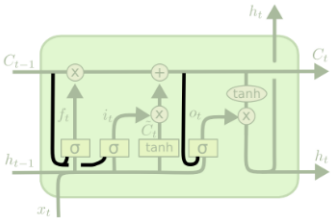
12.3.2. LSTMs with Peephole Connections

- 'Peephole connections' allow the gate layers to 'peep' at the cell state C i.e. the gate layers take C as an additional input. These peepholes can be added to all gates (as below), or only some:

$$f_t = \sigma\big(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f\big) \tag{12.27}$$

$$i_t = \sigma\big(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i\big) \tag{12.28}$$

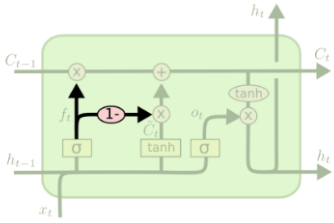
$$o_t = \sigma\big(W_o \cdot [C_t, h_{t-1}, x_t] + b_o\big) \tag{12.29}$$



12.3.3. LSTMs with Coupled Forget & Input Gates

- Instead of separately deciding which info in the cell state to forget and which new info to add to the cell state, make these decisions together so that only forget something if we're going to input something new in its place:

$$C_t = \big(f_t \cdot C_{t-1}\big) + \big((1 - f_t) \cdot \widetilde{C}_t\big) \tag{12.30}$$



12.3.4. Gated Recurrent Unit (GRU) Networks

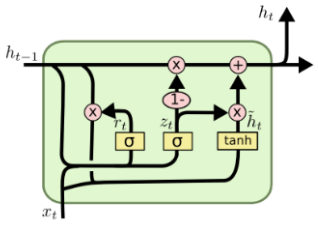
- Combines the forget & input gates into a single 'update gate', and merges the cell state with the hidden state. This is simpler than vanilla LSTM models, and is becoming increasingly popular since have fewer parameters to train and are simpler to implement:

$$z_t = \sigma\big(W_z \cdot [h_{t-1}, x_t]\big) \tag{12.31}$$

$$r_t = \sigma\big(W_r \cdot [h_{t-1}, x_t]\big) \tag{12.32}$$

$$\widetilde{h}_t = \tanh\big(W \cdot [r_t \cdot_{t-1}, x_t]\big) \tag{12.33}$$

$$h_t = \big((1 - z_t) \cdot h_{t-1}\big) + \big(z_t \cdot \widetilde{h}_t\big) \tag{12.34}$$



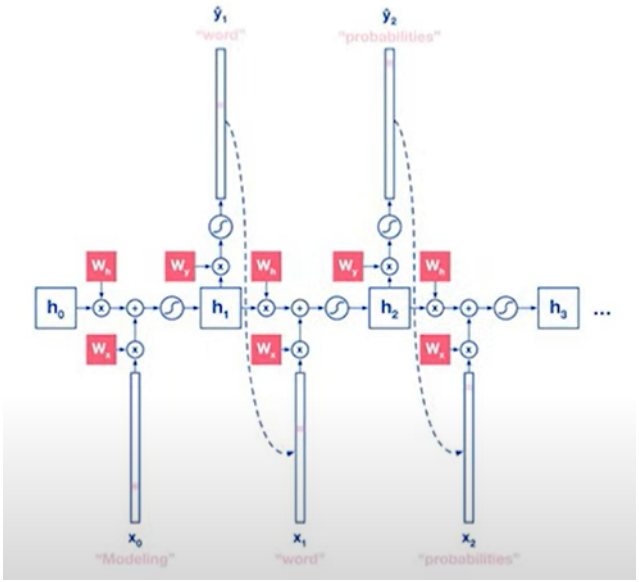
I.e. which specific LSTM you use depends on your application. Other notable variants of LSTMs:

- Depth Gated RNNs
- Clockwork RNNs

12.4. Applications & Examples of Sequence Modelling

12.4.1. Generating Sequences

- So far have looked at how to train a model to *estimate the probability* of a sequence (e.g. a new sentence) occurring. E.g. can estimate the probability distribution of the next word in a sentence
- A more interesting use of our trained model is to *generate* a new (probable) sequence (e.g. a sentence to caption an image)
- Going back to our previous example from Eq 12.9, to generate the sentence rather than just estimate the probability of the next word using e.g. a vanilla RNN, we use a similar process, but now rather than feeding in some external sequence's word to get a probability distribution of the next word, at each module in the RNN we feed the most likely word predicted from the previous module. By the end of the RNN chain, will have generated the most probable sequence of words (sentence) given some initial input (e.g. an image):

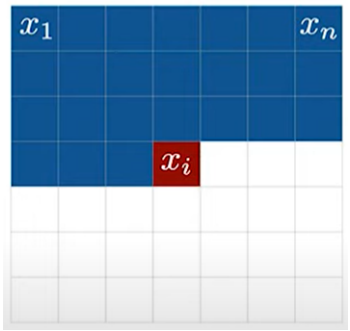


- So far have considered languages/sentences as sequences. Will now look at a few different examples of sequences where ML has been applied.

12.4.2. Images as Sequences

PixelRNN (2016)

- Treated images as a sequence of pixels
- E.g. given the previous sequence of pixels, model the probability distribution for the colour of the next pixel (highlighted in red):



- Can then generate images by e.g. picking the most probable colour pixel for each element in the sequence of pixels
- Was able to learn quite well what a natural real image should look like (nowadays can do much better though)

12.4.3. Language as Sequences: Seq2Seq Models

Sequence-to-Sequence (Seq2Seq) Models (2014)

- Sequence-to-sequence models take in a sequence of elements (e.g. words, letters, time series, etc.) and output another sequence in a different domain. E.g. for translation: take in a sequence of e.g. English words and output sequence of e.g. Japanese words
- Have been used successfully for e.g. translation, text summarising, image captioning etc. Has been used in Google Translate since 2016 ('Google Neural Machine Translation) (N.B. 'Neural Machine Translation' (NMT) is using neural networks for translating languages)

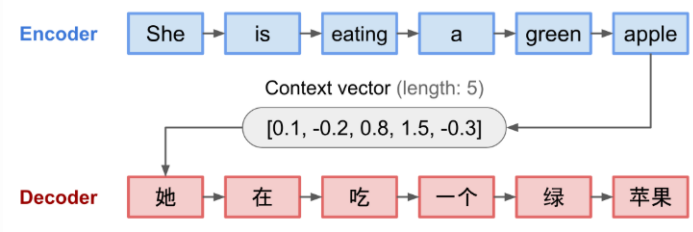
Trivial Case: Same Size Input & Output Sequences

- If the input & output sequence have the same length, can use standard LSTM or GRU networks
- Can use this to train model to e.g. learn how to add numbers (encoding each number as a character string). E.g. given the string sequence '345+452' (size 7), can train LSTM or GRU model to output string sequence '0000797' (also size 7)

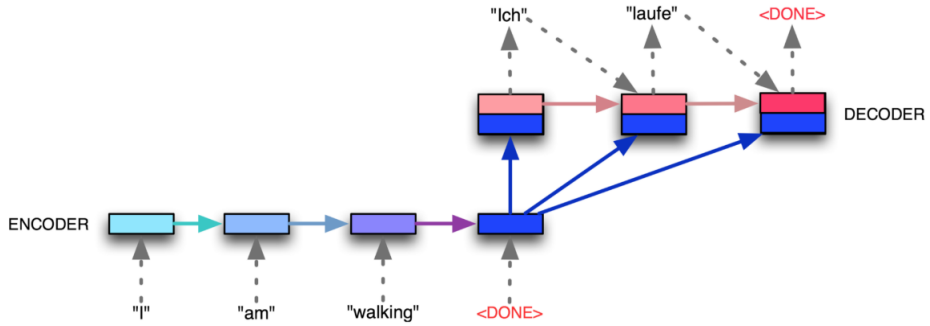
General Case (Canonical Seq2Seq): Different Size Input & Output Sequences

- Most applications of seq2seq (e.g. translation) have different size input & output sequences. This requires a more advanced model setup (which is what most people refer to when referring to 'seq2seq' models)
- Canonical seq2seq models have 2 main components (which are essentially 2 separate RNN models):
    1. **Encoder**: A layer/stack of RNN module/modules. Trained to take the input sequence (e.g. a sentence in English) and use the RNN module(s) to create an 'encoded' state representation (both a hidden state h and, if using LSTM, a long-term memory cell state C). This encoded state is passed on to the second component of the seq2seq model (see below), and therefore is 'internal' to the RNN (i.e. is a hidden state representation of the original sequence). This hidden state should be a good summary of the contextual meaning of the whole original source that was input to the RNN. It will be a hidden context state vector, and will be of *fixed length*
    2. **Decoder**: Also a layer/stack of RNN module/modules. Its initial hidden context state h and cell state C is the output of the encoder component. The decoder learns to go through encoder's output sequence and use it as the initial context &/or long-term memory to generate a new sequence, which does not have to be the same length as h ot C and therefore does not need to be the same length as the sequence that was put into the encoder.

- E.g. seq2seq (i.e. LSTM or GNU units) with hidden context vector of length 5 translating an English sentence into Chinese:



- E.g. In the case on **translating** the English sentence 'I am walking' into German (source format: English sentence, target format: German sentence), the input into the encoder RNN is the English sentence, which generates an internal representation of this sentence as the initial input for the decoder RNN, which generates the equivalent sentence in German:

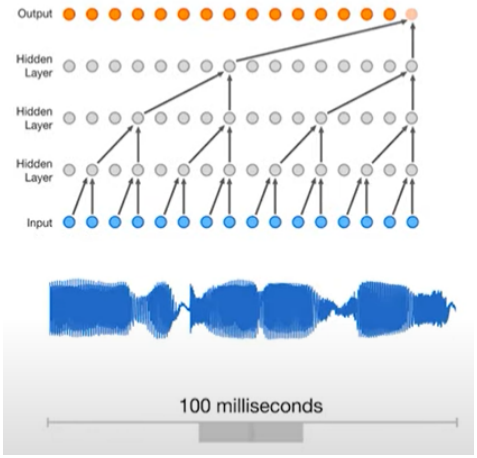


- E.g.2. In the case of **image captioning** (source format: image, target format: sentence) (e.g. taking the below image and generating the sequence 'A person riding a motorcycle on a dirt road'), replace the encoder with a CNN model to generate an internal hidden representation tensor of the input image. The representation output by the CNN will be a high-level representation of the original input image (e.g. summarising a dog's face etc. at a high level). Use this hidden representation as input to the decoder RNN, which will generate the equivalent sequence as a caption

12.4.4. Audio Waves as Sequences

Google WaveNet

- Used in Google's voice assistant
- Rather than using RNNs, uses deep CNNs
- Takes high-resolution audio and gradually extracts high-level representations with lower-resolution convolutions to translate audio into actionable commands:



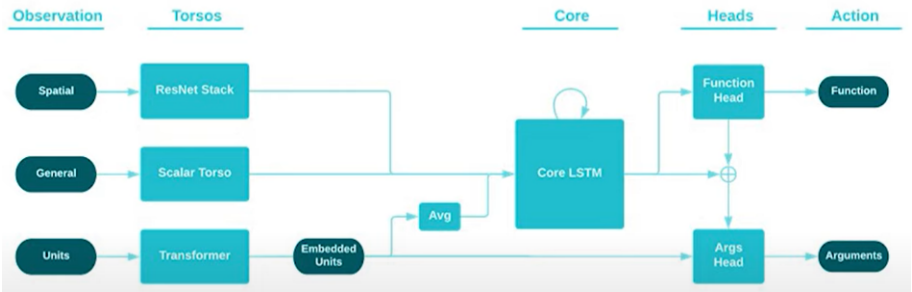
- N.B. only looks back at fixed size of input (i.e. fixed amount of time into past of audio clip), therefore does preserve some long-term memory/info but not as well as LSTMs, therefore struggles with long-term dependencies

12.4.5. RL Agent Policies as Sequences

- RL policies are policies for making sequential decisions, therefore RNNs (LSTMs, GRUs etc) can help

- Help with e.g. deciding where and when to draw on a canvas for RL agents being trained to make paintings. Have doines this to draw quite good human faces with only a few brush strokes:

- E.g.2. OpenAI's RL agent's policy used to play Dota2 was heavily dependent on LSTMs
- E.g.3. DeepMind's Alphastar RL agent to play Starcraft II also used LSTMs. Architecture took the observation and generated some representation(s), and at the core had an LSTM generating a sequence of decisions to take given the observation:



12.5. Attention Mechanisms in Neural Networks

12.5.1. The Problem with Vanilla Seq2Seq Models

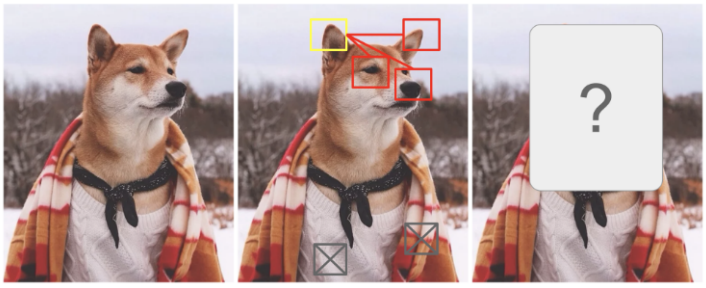
- Previously was saw a simple seq2seq model for e.g. translation, which was composed of an encoder and a decoder, both of which were RNNs i.e. using LSTM or GRU units.
- A disadvantage of these vanilla seq2seq models based on standard LSTMs/GRUs is that once it has finished processing the whole input (e.g. translated the last word of an English sentence into French), it has already forgotten the first part of the sentence that it translated. I.e. there's a similar problem with LSTMs as with RNNs; as distance between relevant info in the chain increases, probability of keeping this info in the chain decreases quickly
- . This is fine for short sentences, but for longer sentences we may need longer memory than this, especially for prediction tasks where e.g. want to predict a word at the end

of a long sentence that can only be predicted if have remembered the beginning of the sentence.

- I.e. General disadvantage of vanilla seq2seq LSTM or GRU networks is limited long term memory. To solve this, the **attention mechanism** was used in 2015 by Bahdanau et al. for neural machine translation of longer sentences

12.5.2. Introduction to Attention

- **Attention**: A mechanism used by NNs to selectively decide which part(s) of a given set of data to pay more & less attention to when making predictions. Is usually a vector of values which can be used as weightings to weigh how much each element in the data should be 'attended to'.
- NN attention mechanisms are inspired by attention mechanisms in human visual system. E.g. consider below image; to identify what image is showing (a dog in a man's outfit), you focus on ('attend to') a small patch of pixels around the face:



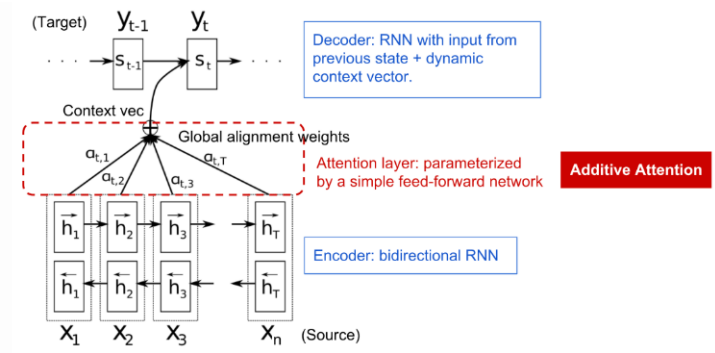
- Attention is useful for many other types of data processing. E.g. in the following sentence:

We pay high attention to the word 'eating', and therefore we know that a food word is probable to occur very soon. The colour 'green' describes the food, but is less related to the word 'eating'. In this way, the ways in which words 'attend' to one another is different, which is a common feature of many different data types/sequences.

- I.e. A NN model with an attention mechanism can predict a certain element (e.g. a pixel in an element or a word in a sentence) by taking the attention vector and using it to estimate how strongly correlated the target element is with other elements in the data/sequence to inform its prediction about what the target might be.

Example: Attention for Neural Machine Translation

- Bahdanau et al. 2015 first used an attention mechanism for NMT to help with memorising long source sentences when making target predictions.
- Previously we saw that vanilla seq2seq models have an encoder and a decoder, where the encoder is responsible for outputting a fixed length vector summarising the whole of the input into a (hidden) context state, which is the initial input for the decoder
- Bahdanau et al. added an attention mechanism to the beginning of the decoder, whereby an attention layer (termed an *alignment model*) would take the original source input data that was originally put into the encoder and weight the inputs by some attention vector output by a standard feed-forward NN, and linearly combining this (via addition) with the hidden context representation output by the encoder, thereby forming the input to the decoder. This therefore informed which parts of the source input the decoder should pay more attention to, relieving the encoder from the burden of having to encode all info in the source sequence into a fixed-length summary context vector:

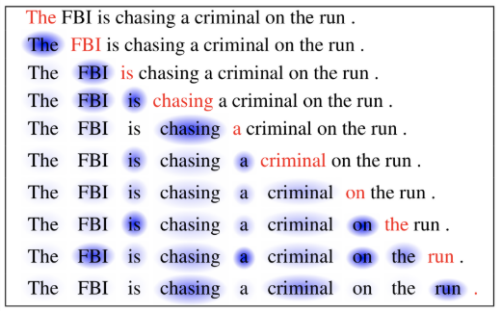


- I.e. If there is a particularly important piece of information at the start of a long sentence, the attention vector enables the decoder to 'attend to'/remember this part of the sentence when making its predictions whilst also considering the hidden context vector output by the encoder.
- This is a type of *attention interface* in that an attention layer interfaces between components (in this case, an RNN encoder and an RNN decoder). See later section on 'Attention Interfaces'

12.5.3. Types of Attention Mechanisms

- There are 3 broad categories of attention mechanism:
    1. **Self/intra attention**: Relates different elements in a single sequence to one another to produce an attention-weighted representation of the same sequence. Is often used synonymously with just 'attention'. Was used by e.g. Cheng et al. 2016 to do machine reading. Self-attention enabled model to learn how the current word being looked at correlated with the previous part(s) of the sentence. E.g. with the current

word being looked at in red, and the size of the blue shade indicating the 'activation level' (i.e. which part(s) of the previously looked at sentences correlated most with the word currently being looked at, and therefore should be most 'attended to'):



2. **Global/soft attention**: Attending to the entire input sequence. Alignment/attention weights are learned and placed 'softly' over the entire source data (e.g. the kind of attention mechanism employed by Bahdanau et al. 2015 for NMT). Models with soft attention mechanisms are smooth and differentiable, but have a high computational cost if the size of the source input is large, since need attention weights for all elements of the input.

   Was first used by Xu et al. 2015 for image captioning. Used a CNN encoder to extract high level features and an LSTM decoder to consume the convolution features weighted by a soft attention mechanism. Below can see the regions the model 'attended to' when generating each word in the caption sequence 'A woman is throwing a frisbee in a park':

3. **Local/hard attention**: Only attend to part of an input sequence at a time. Leads to less computation at inference (prediction) time, but the model is non-differentiable and requires more complicated techniques (e.g. variance reduction, reinforcement learning etc.) to train

- Some of the most popular specific attention mechanisms include:
  - Content-based attention
  - Additive attention
  - Location-based attention
  - General attention
  - Dot-product attention
  - Scaled dot-product attention

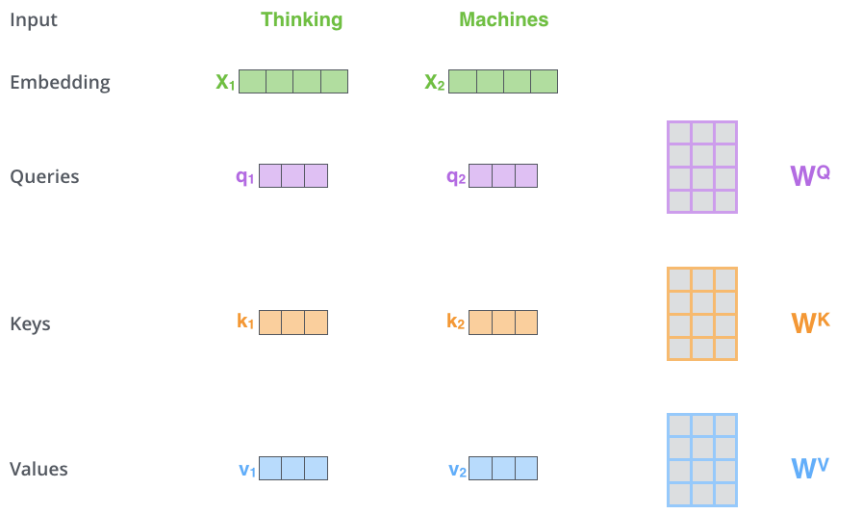
12.5.4. Self-Attention in Detail

- Consider that we have 2 words, 'thinking' and 'machines', which we want to calculate self-attention vectors for (from perspective of solving this with seq2seq model with an encoder and a decoder component). Consider that we want to use an attention layer to determine how much attention to give to the first element in the sequence, 'thinking'

Using Vectors

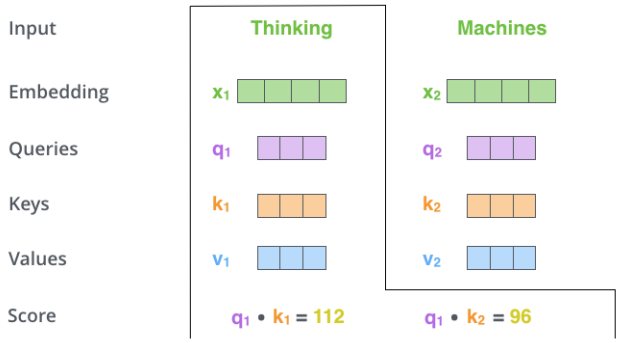
- 7 steps:
  1. As with any NLP process, start by taking each input word and using an embedding algorithm to code each word into a vector of fixed size (e.g. size of each vector = 512).
  2. For each embedded vector, generate 3 new vectors **q**, **k**, and **v**, where each vector is an abstraction useful for thinking about attention/dependencies, which itself is similar to a retrieval process (a process of mapping a *query* (e.g. YouTube search) against a set of *keys* (e.g. video title, description, etc.) in a database of candidate answers/values, then present the best matched *values* (e.g. videos)). In this way, attention can be thought of as computing a weighted some of the **values** dependent on the **query** and the corresponding set of **keys** (the query determines which values to focus on, and is said to 'attend to' the values):
     a) **Value vector**: The attention value being retrieved. Is usually the encoder's hidden states
     b) **Key vector**: The key generated to mark the attention value. Is usually the encoder's hidden states
     c) **Query vector**: The query generated to be matched against the key vector and find the value vector. Is usually the decoder's hidden states

     Generate these 3 vectors for each input by multiplying the embedding vectors by 3 sets of parameters/matrices $\boldsymbol{W}^{Q}$, $\boldsymbol{W}^{K}$, and $\boldsymbol{W}^{V}$ trained during training:



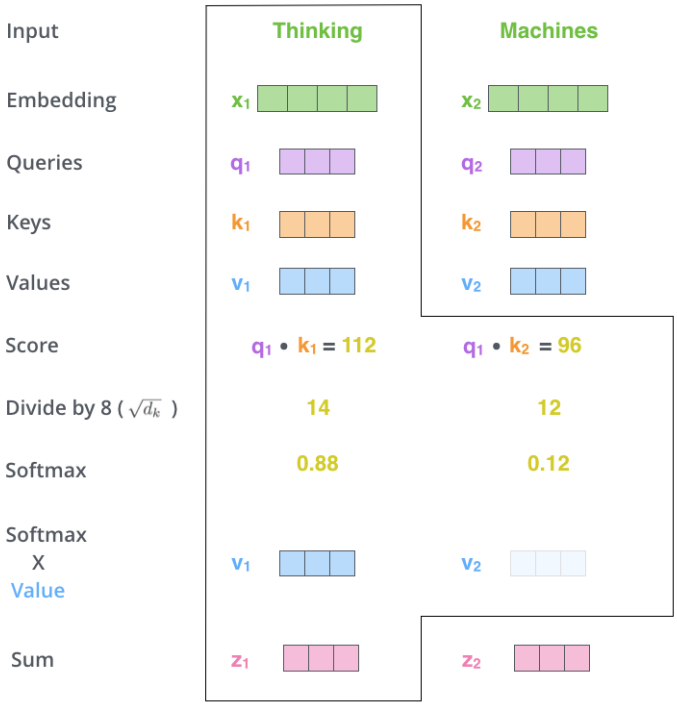
3. Calculate a **score** for each element in the input sequence, which is a score of each element in the sequence against every other element in the sequence. By the end, the element(s) with the highest score are where most attention must be given.

   Usually calc score by taking dot product of query vector **q** for element being scored with key vector **k** for each element in the sequence. In below example, word being focused on is 'thinking', therefore obviously it has a strong dependency/score with itself, but in longer sentences might see interesting distribution of dependencies spread out:



4. Squash scores to between 1 and 0 by passing scores through a softmax function to output probability distribution (sums to 1) over all scores, which is an attention distribution relative to element being considered telling model how it should distribute its attention when e.g. making predictions about the sequence.
5. Apply attention weighting by multiplying each element's value vector **v** by its corresponding score to get attention-weighted value vector for every element relative to element being considered
6. Sum the components of each element's attention-weighted value vector to get. This

is the attention layer's output for the word 'thinking':



- N.B. In above example we considered processing word 'thinking'. Key point about self-attention is that all words in the sentence can be processed in *parallel* i.e. can generate attention layer output for 'machine' at same time as 'thinking' (i.e. don't need to process sequentially), therefore can use **matrices**
- N.B.2. Divide by 8 term is there because in example of Vaswani et al. 2017 Attention is All You Need paper (see later section on 'transformers'), they found that dividing by the square root of the number of dimensions in the key vector ($d_k = 64$) lead to more stable gradients

Using Matrices

- Steps:
  1. Generate word embeddings as usual. Concatenate vector embeddings into matrinx $\mathbf{X}$
  2. Multiply $\mathbf{X}$ by trained matrices $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$, and $\boldsymbol{W}^V$ to get $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ matrices
  3. Get outputs of self-attention layer:

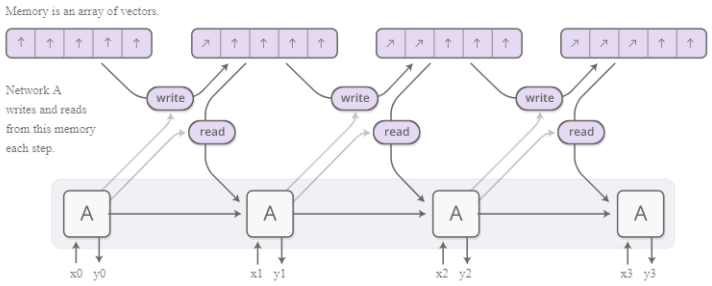
$$\boldsymbol{Z} = \text{softmax}\left(\frac{\boldsymbol{Q} \cdot \boldsymbol{K}^T}{\sqrt{d_k}}\right) \cdot \boldsymbol{V} \tag{12.35}$$

12.6. Augmented Recurrent Neural Networks

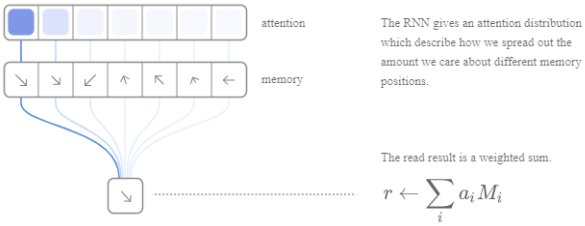
- 4 main directions of augmenting RNNs to give them new properties that allow them to do new interesting sequence processing tasks:
    1. Neural Turing machines (NTMs)
    2. Attentional interfaces
    3. Adaptive computation times
    4. Neural programmers
- All of the above are extensions of RNNs, but they all use the same underlying principle of **attention**

12.6.1. Neural Turing Machines (NTMs)

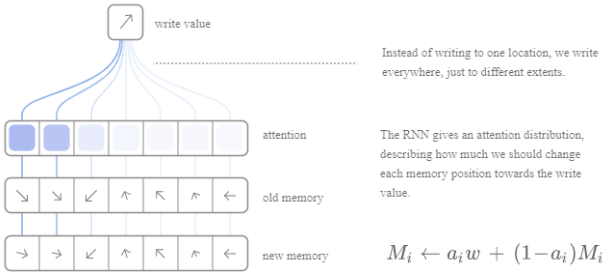
- Augment RNNs by giving them large memory banks which they can read from/write to, enabling a range of new NN applications
- In 1936, Alan Turing proposed a model to simulate any algorithm. It was made up of a long 'tape' of 0s and 1s, and a 'head' to interact with the tape. The head could:
    1. Read symbols
    2. Edit symbols
    3. Move left & right on the tape
- An NTM is a NN model architecture proposed in 2014 with an external memory storage (the 'tape', termed the **memory bank**) coupled to a NN (the 'head', termed the **controller NN**) which can interact with the external memory storage.
- Like Alan Turing's model, the external memory is an array of elements, but rather than a 1 or a 0, each element is a vector
- To decide where in the memory bank to read & write from/to, the NN uses a *soft* attention mechanism, whereby it reads & writes from/to *everywhere* in the memory bank at each step, but by different (attention weighted) extents for each vector element in the bank:



- E.g. To read from the memory bank, the NN outputs an **attention distribution** that describes the spread of the amounts that it cares about the different memory elements for that particular step. The result of every read operation is a weighted sum of all the components of the memory bank:



- E.g.2. Similarly to write, write to everywhere in the memory bank, but use attention distribution to weight how much each element is written to for that particular step:



- The attention distribution used for reading & writing is the output vector of an RNN (i.e. LSTM or GRU)
- The external memory bank of NTMs enables NNs to store much more in memory, enabling applications that were previously beyond NNs. E.g. they can learn to mimic look up tables, store long sequences in memory, sort numbers etc. These might enable future applications such as Neural GPUs etc.

12.6.2. Attentional Interfaces

- As we've seen, attentional interfaces are interfaces between a NN and its input that enable the NN to attend more to the most import part(s)/subset(s) of the input data.
- These have been used successfully in translation, image captioning etc. by relieving responsibility from the encoder and allowing the decoder to look further back at initial part(s) of a sequence that are important for prediction which otherwise might have been forgotten
- Have also been applied to e.g. interface between a CNN and a RNN for image captioning etc. (see previous)
- Attentional interfaces are extremely general and powerful for NNs

12.6.3. Adaptive Computation Times

- Standard RNNs do the same amount of computation for each step/element when processing a sequence. This constrains the RNN to having to do O(n) operations for a sequence of n elements, and also means that can't do more computations for any particularly difficult (& perhaps important) steps
- To solve, train an RNN to output an attention distribution over the number of steps to run, and weight the number of steps to run by this attention distribution. Can tune the adaptive computation time hyperparameter to encourage to NN to learn to prioritise either compute time (only do more ops for most important steps) or performance (do many more ops for most important steps)

12.6.4. Neural Programmers

- **Neural programmer**: A NN that learns how to generate a programme in order to solve a task. Does so without needing examples of correct programmes, by discovering how to produce programmes as a means to accomplishing some task.
- At each step, the RNN controller outputs a probability distribution over what the next op (e.g. 'add the output of the op 2 steps ago and the output of the op 1 step ago') should be
- After each step, rather than running a single op, evaluate the output distribution over all ops by running all of them and then average the outputs together weighted by the probability that we ran that operation
- I.e. the probability distribution over the possible ops is our attention vector that we use to find the weighted average and use this as our output

Key take home message for attention

- Key thing about attention mechanisms is that they output a vector of elements which are **differentiable**, and therefore we can use traditional differentiation of loss w.r.t. the NN parameters that output the attention vector in order to optimise our NN
- Furthermore, attention allows us to 'choose' *all* available actions/predictions etc. (but to varying extents), therefore never really have to 'choose' a single action (as do in e.g. field of RL), which makes things more simple
- Disadvantage of attention mechanisms are that by taking every available action at every step, the computational cost grows linearly with e.g. the amount of memory in a neural Turing machine (hence why might use e.g. local/hard attention mechanisms)
- Using attention mechanisms with ML is a hot active area of research

12.7. Transformer Networks

- Disadvantages of networks with recurrence:

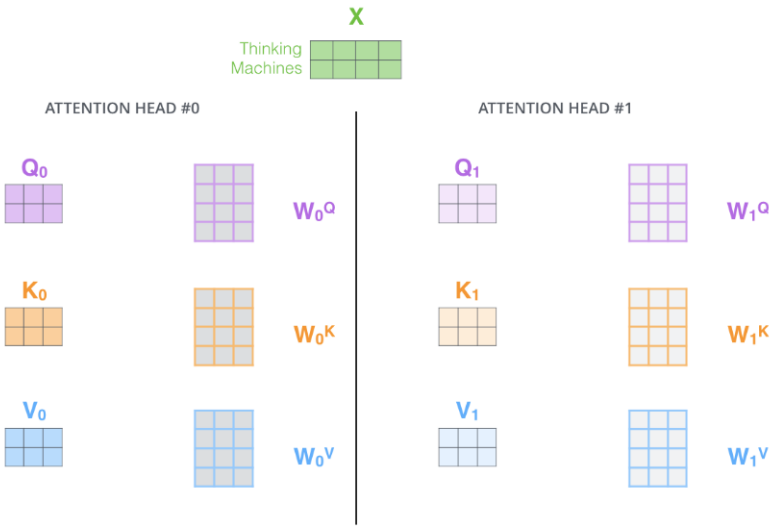
1. Must process each element in the sequence sequentially, therefore long processing time to e.g. translate a sentence (even if self-attention used)
2. Relevant info/dependencies must travel long distances due to sequential nature, making learning long-range dependencies more difficult (can solve with self-attention, since self-attention computes dependencies of all elements with one another in the sequence)
3. Number of computation steps needed per layer scales O(n), where n is length of sequence (can solve with self-attention, since self-attention layers connect all elements in the sequence with a constant number of sequentially executed ops to learn the dependency between elements independently of their position distance in the sequence)

- Transformers are an alternative approach to seq2seq modelling *without* recurrent network units/RNNs/recurrent architectures. They are based entirely on self-attention mechanisms
- First introduced by Vaswani et al. 2017, transformers have replaced RNNs (LSTMs, GRUs etc.) for many sequence processing tasks such as language translation
- Instead of recurrence to get a representation for a given element in a sequence, transformers use self-attention to estimate how important all other elements in the sequence are to the given element to get an attention vector. The element's updated feature values are the attention-weighted sum of the linear transformation of all of the other elements.
- Advantages of transformers:
  1. Self-attention mechanism therefore have shorter long-term dependency propagation distances
  2. Self-attention mechanism therefore have a constant number of computation steps per layer independent of element dependency position distances in sequence
  3. Don't use recurrence (i.e. don't process each element in sequence sequentially) therefore can parallelise to get much faster training and inference
- Transformers use 2 key concepts; **multi-head self-attention** and **positional encoding**

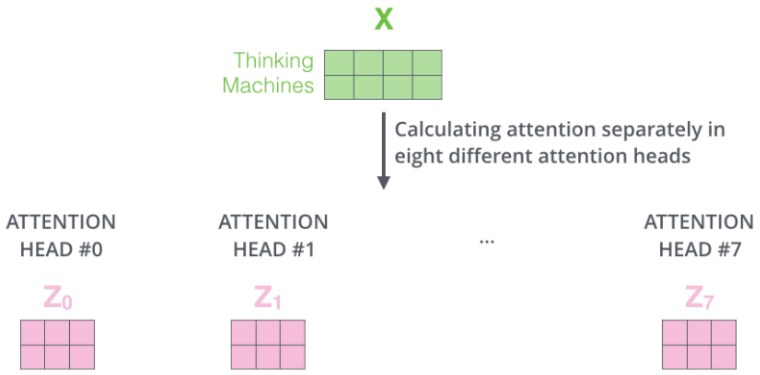
12.7.1. Multi-Head Self-Attention

- Problems with vanilla self-attention looked at earlier:
  - Since the attention layer output for a given element will always strongly 'attend to' itself, the ability of the model to attend to elements in different positions is limited
  - With a single set of Q/K/V matrices, can only project the embeddings into a single attention-weighted representation space, which limits the complexity of element dependencies with different positions in the sequence that the attention layer can represent

- One of key contributions of Vaswani et al. 2017 paper. Can be thought of as multiple attention layers running in parallel, allowing for the attention layer to jointly attend to information from different representation sub-spaces at different positions in the sequence w.r.t. a given element
- Works by initialising and training **multiple** sets of Q/K/V matrices (in Vaswani et al. 2017, used 8 sets), where each set is referred to as a **head**:

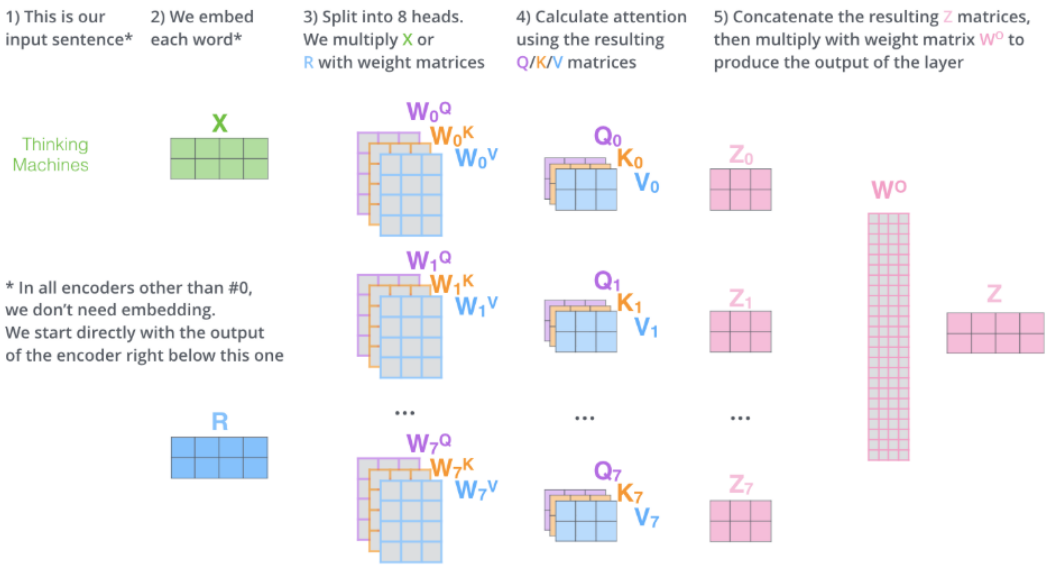


- Then do the same process as previously described with matrices when introducing self-attention, but for the e.g. 8 different heads (sets of Q/K/V matrices) to get a set of attention layer output matrices (calculation done in parallel):



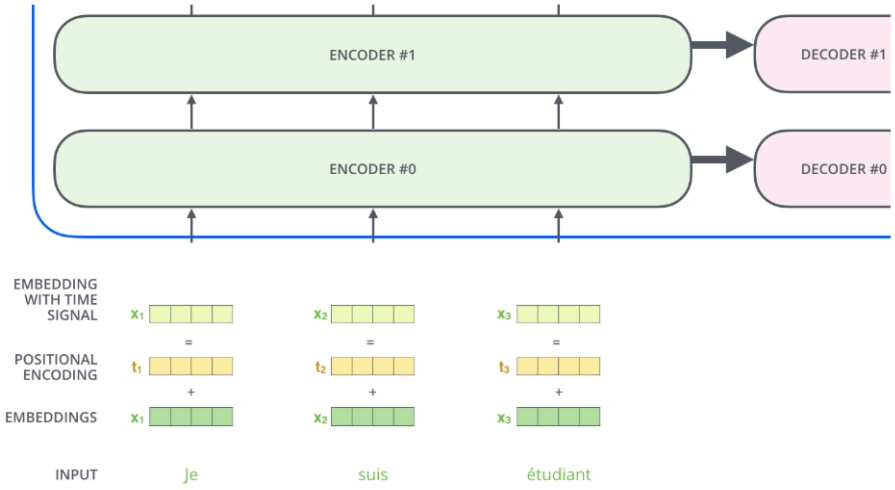
- To make output matrices processable by a NN (which, rather than a set of matrices, expects a single matrix (a vector for each word/element in the sequence)), concatenate the attention head outputs and multiply with a trained weight matrix W^O to output a single matrix Z which captures the information from all the attention heads and can be put into e.g. a feed-forward NN

- Summary:



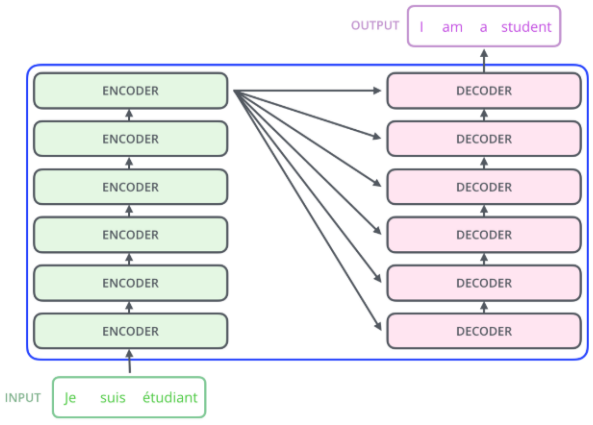
12.7.2. Positional Encoding

- Since transformers do not use recurrence, they do not sequentially process each element of the input sequence one after another. However, as we know, the position/order of each element in a sequence is often very important.
- To address this, transformers add a vector to each input embedding. The added vector contains info related to relative element positions and distances from one another.
- The positional encoder used in the paper was trained and capable of performing positional encoding on sequences longer than any sequence in the training set
- All inputs would have their positions encoded before being passed on to e.g. an encoder layer:



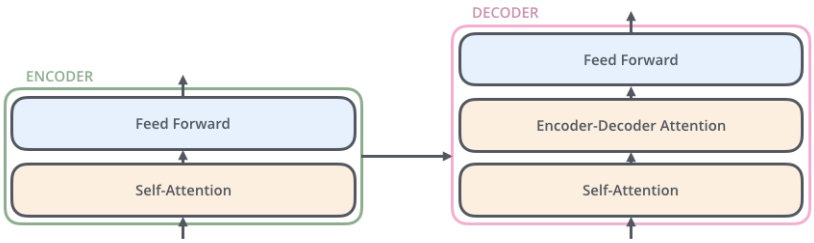
12.7.3. Transformer Architecture

- Useful transformer visualisation/GIF at bottom of this blog post: http://jalammar.github.io/illustrated-transformer/
- Similar to previous seq2seq models, transformer networks are composed to 2 parts; an encoder and a decoder
- Both the encoder and the decoder have $N = 6$ encoder/decoder layers/stacks. The original sequence is input to the first encoder layer, is passed through each layer of the encoder separately, and then the output of the last encoder layer is given as input to all of the decoder layers, which as an additional input also receive the input from the previous decoder layer (except obviously from the bottom decoder layer):

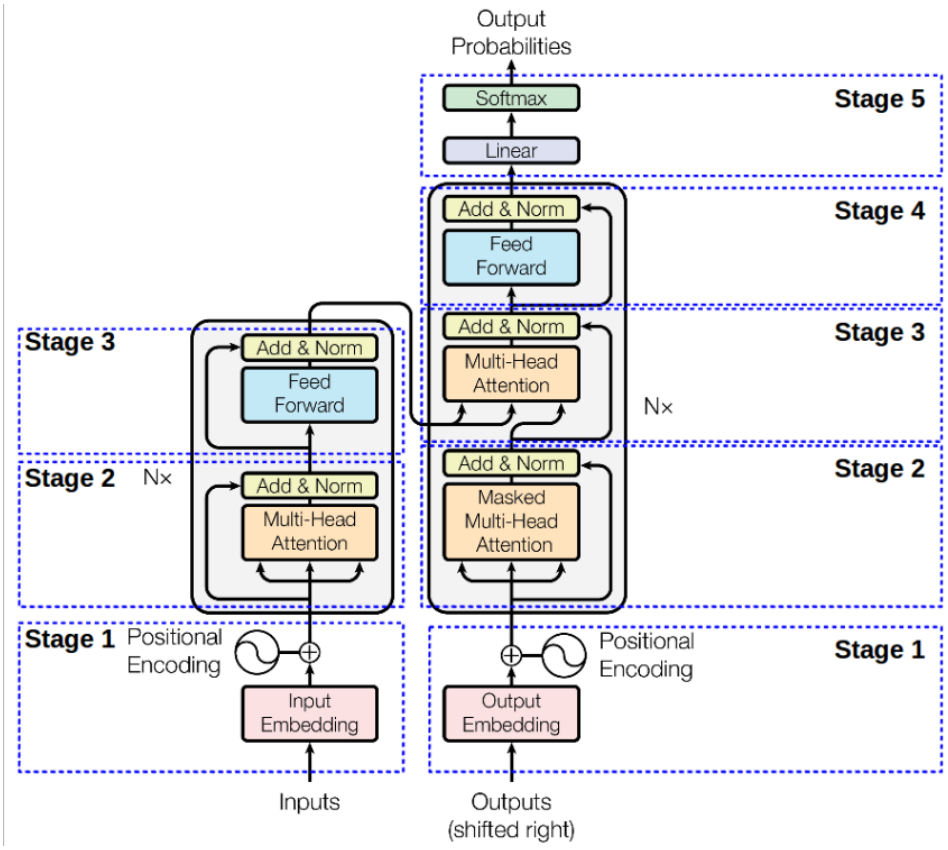


- Each layer in the **encoder** is identical and contains **2 sub-layers** (a *multi-head self-attention mechanism* and a *feed-forward NN*), and each layer in the **decoder** is identical and contains **3 sub-layers** (a *multi-head self-attention mechanism*, a *feed-forward NN*, and

another *multi-head self attention mechanism* for performing multi-head attention over the outputs of the 6-layer stack of encoders):



- Each layer in the encoder has 3 stages, and each layer in the decoder has 5 stages:

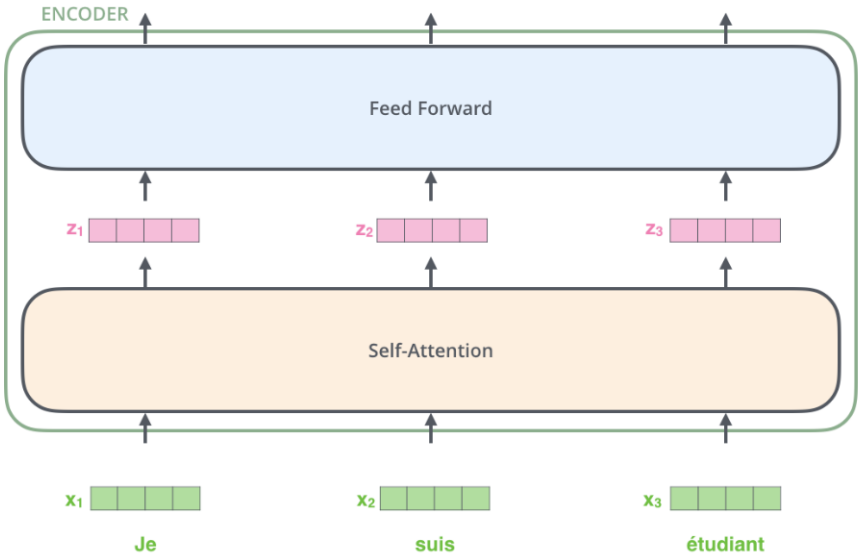


Encoder

- Stages of an encoder layer:
  1. Positional encoding: Encodes the relative positions & distances of each element in the input sequence

2. Multi-head attention: An attention mechanism for summarising the dependencies of each element in the input sequence with one another
3. Feed-forward NN: Creates a hidden representation of the attention-weighted sequence

   Crucially, a key property of transformers is that each element flows through its own path in the encoder, allowing for parallelisation:

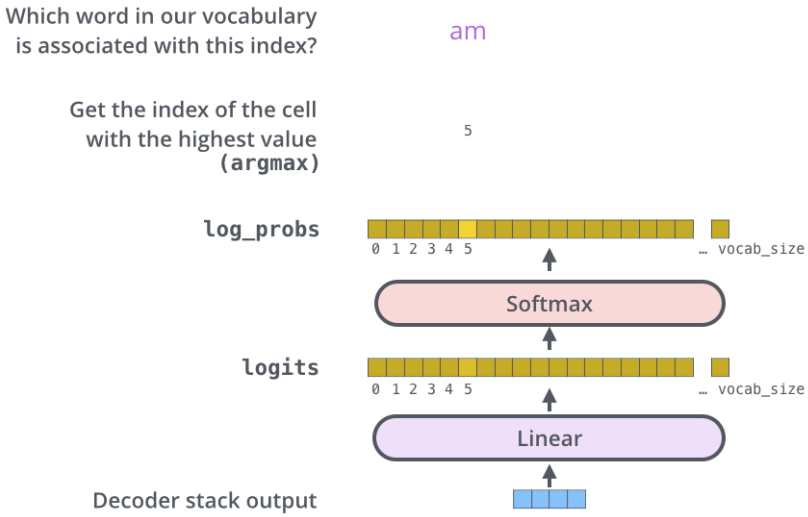


   The output of each encoder layer is passed to the next encoder layer until the top encoder is reached. The top encoder's output is transformed into a set of attention vectors **K** and **V**, which are used by each decoder layer to help focus attention in the appropriate places in the input sequence when processing each element.

Decoder

- Stages of a decoder layer:
  1. Positional encoding: Takes the output sequence from the previous decoder layer and encodes the positions of each element in the sequence. The output sequence's elements being fed into decoder layer are offset by 1 position so that predictions for an element in position i of the sequence can only depend on the known representations of elements in positions less than i i.e. can't look ahead at future elements of sequence when making predictions
  2. 'Masked' multi-head attention: Self-attention layer that has been modified to prevent an element i attending to elements in positions greater than i i.e. can't attend to future elements in sequence

3. Multi-head attention: Takes **K** and **V** attention vector outputs from top encoder layer and uses to focus attention on important parts of sequence for given element being processed (N.B. obviously all elements are being processed in parallel)
4. Feedforward NN: Outputs attention-weighted representation of sequence as a vector of floats
5. Linear + softmax layer: Fully-connected linear layer takes vector of floats and outputs logit vector, where have 1 logit for each possible class of element (e.g. if have 10,000 words in vocabulary, vector would have 10,000 logits), where each cell in vector corresponds to score of unique word in vocabulary. Softmax converts this into probability distribution over possible words (sums to w). Cell with highest probability is chosen and corresponding word is produced as output for current time step



- Note also the residual connections in the transformer network; this allows stacking of many layers in the encoder and decoder to allow for deeper transformer network models than would otherwise be possible

12.8. Pointer Networks

- 'Pointer Networks', Vinyals et al. 2015 https://arxiv.org/pdf/1506.03134.pdf
- Are a variation of seq2seq models
- Standard seq2seq models we've seen so far are made up of e.g. an LSTM encoder coupled with an LSTM decoder. They take an input sequence of any length, the encoder turns it into a fixed-length representation, and the decoder transforms the representation into a target sequence, where the target sequence can be a different length from the source sequence. Used for e.g. language translation

- Rather than translating one sequence into another representation of the sequence with any length (as with standard seq2seq models), pointer networks:
    1. Take a sequence and output 'pointers' to each element in the input sequence, where each pointer is a key/'target class' in the 'target dictionary' e.g. an index (dict key) of an input sequence element (dict value)
    2. The number of elements/target classes in the output sequence is dependent on the number of elements in the input sequence (whose length can vary)
- Use pointer networks for seq2seq problems where solutions are discrete and correspond to positions in the input sequence. The number of target classes in each step of the output depends on the length if the input (which is variable); standard seq2seq models and neural Turing machine cannot handle these types of problems. Common problems suitable for pointer networks:
    1. Sorting sequences (e.g. sorting sequence of numbers from small to large)
    2. Various combinatorial optimisation problems (e.g. travelling salesman problem)

Vanilla Seq2Seq Models

- Have e.g. an LSTM encoder and an LSTM decoder
- The encoder LSTM takes an input sequence P of n embedded vectors and encodes a hidden context state representation of the input sequence, and the decoder takes the hidden state representation and outputs a sequence C
- Softmax output is a probability distribution over all possible classes of elements in the output sequence (e.g. in case of language, might have 10,000 word vocabulary). Therefore, length of output is independent of length of input sequence

Standard Seq2Seq Models with Attention

- Also have LSTM encoder and LSTM decoder, but now have an attention layer that generates an attention distribution over the entire sequence (soft/global attention) of the encoder's output hidden state representation
- Performs much better than vanilla seq2seq, but still can't handle problems where output dictionary size depends on size of input sequences

Pointer Networks

- Is a slightly modified version of standard LSTM-based seq2seq models with attention
- Rather than the softmax of the decoder LSTM outputting a probability distribution over all possible sequence element classes/dictionary for a general problem (e.g. English

language), pointer network decoder's softmax layer outputs a probability distribution over the dictionary of inputs

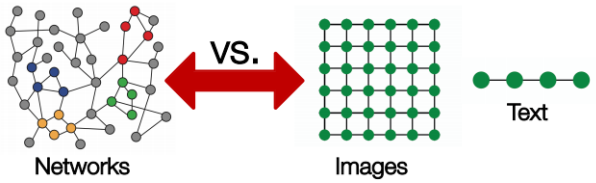
- Each class in the output 'points' exactly to an element in the input sequence
- Can therefore e.g. train pointer network to produce a sequence of elements, where each element points to/indexes one of the elements in the input sequence, and the order of these output elements corresponds to e.g. the sorted order of input elements in terms of size (e.g. from lowest to highest)
- Paper kept architecture of pointer network very simple, using a single layer LSTM with either 256 or 512 hidden units, trained with SGD with learning rate of 1.0, batch size of 128, random uniform weight initialisation from -0.08 to 0.08, and L2 gradient clipping of 2.0. Trained model on 1M labelled examples (found that overfitting did occur for small/simple tasks)

13. Graph Neural Networks

- **Learning Resource(s)**:
  1. *'The Graph Neural Network Model', Franco Scarselli et al., IEEE Transactions on Neural Networks, 2009,* https://ieeexplore.ieee.org/document/4700287
  2. *'Graph Neural Networks: A Review of Methods and Applications', Jie Zhou et al., 2018,* https://arxiv.org/abs/1812.08434
  3. *'A Comprehensive Survey on Graph Neural Networks', 2019, Zonghan Wu et al., 2019,* https://arxiv.org/abs/1901.00596
  4. *'Graph Neural Networks'*, Stanford University Course Notes, 2019, https://snap-stanford.github.io/cs224w-notes/, http://web.stanford.edu/class/cs224w/slides/08-GNN.pdf
  5. *'Graph Database Concepts', neo4j,* https://neo4j.com/docs/getting-started/current/graphdb-concepts/
  6. *'Graph Embeddings: The Summary'*, Primoz Godec, TowardsDataScience blog post, 2018, https://towardsdatascience.com/graph-embeddings-the-summary-cc6075aba007
  7. *'Graph Node Embedding Algorithms', 2019, Jure Leskovec, Stanford University,* https://www.youtube.com/watch?v=7JELX6DiUxQ&feature=youtu.be&fbclid=IwAR2QFJMXXP_y7ZeEzXHzJJbWvPnAYpkOdSa8bvTphgLhdtkxO2emlePK5cM

13.1. Why Graph Neural Networks?

- So far, we have looked at a range of powerful ML techniques. These have all been applied to data in the **Euclidean space** e.g. 1D sentences, 2D grid images, etc.
- Many types of data in real-World are structured as **graphs**, which are in **non-Euclidean space**. Examples of data structured as a graph:
  - Molecules (e.g. atoms are nodes, bonds are edges)
  - Social media networks (e.g. users are nodes, relationships are edges)
  - Communication networks (e.g. server end-points are nodes, optical links are edges)



- From above image, can see that can't apply tradition ML techniques we've seen so far to graph-structured data because graphs have:
  1. No fixed node ordering or reference point
  2. Arbitrarily varying size
  3. Complex topological structure (no regular spatial locality like grids)
  4. Nodes/data points with multiple features intricately related to other nodes via various links/relationships
- There is now growing interest in generalising ML techniques (e.g. convolutions, RNNs, autoencoders etc.) from working in Euclidean space to working in non-Euclidean space so that can apply ML to graph problems. A **graph neural network (GNN)** is a NN model applied to graph data (as opposed to Euclidean grid data or 1D data etc.)
- Examples of predictions tasks GNNs are being applied to:
  1. **Node classification**: Predict parameter(s) of a given node
  2. **Link prediction**: Predict whether 2 nodes are linked by an edge and predict parameter(s) for this edge
  3. **Community detection**: Identify & predict parameter(s) of densely linked local clusters of nodes
  4. **Similarity computation**: Predict the similarity of 2 nodes or 2 networks

13.2. Overview of Graphs

13.2.1. Introduction to Graphs

Basics

- **Network/graph**: A set of objects/vertices/nodes N where some pairs of objects are connected by a set links/edges E to form a network/graph $G(N, E)$
- **Undirected graph**: Graph whose edges between nodes are symmetrical/reciprocal (e.g. friendship on Facebook) i.e. edges have no associated *direction*. **Node degree** k_i is the number of edges adjacent to node i, and the **average degree** is:

$$\bar{k} = \langle k \rangle = \frac{1}{|N|} \sum_{i=1}^{|N|} k_i = \frac{2|E|}{N} \tag{13.1}$$

- **Directed graph**: Graph whose edges between nodes have an associated direction. **In-degree** k_i^{in} is the number of edges entering node i, **out-degree** k_{in}^{out} is the number of edges leaving node i, and the **average degree** is:

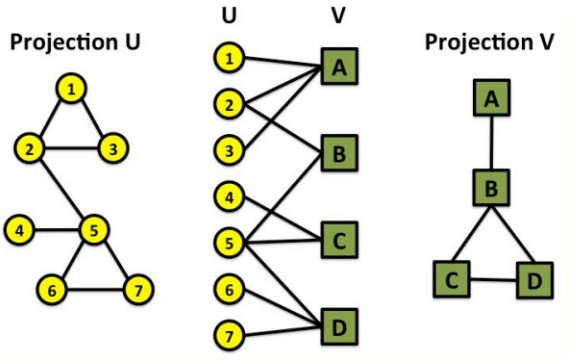
$$\bar{k} - \langle k \rangle = \frac{|E|}{N} \tag{13.2}$$

- **Complete graph**: An undirected graph with the max. no. of edges (i.e. all pairs of nodes are connected):

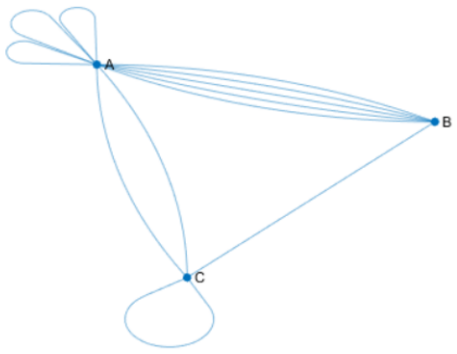
$$|E| = \binom{N}{2} = \frac{N(N-1)}{2} \tag{13.3}$$

$$\bar{k} = |N| - 1 \tag{13.4}$$

- **Edge disjoint paths**: 2 paths that do not share an edge
- **Bipartite graph**: Graph whose nodes can be divided into 2 disjoint sets of U and V (i.e. no edges between nodes in U and no edges between nodes in V), where every edge connects a node in U to a node in V. Can **project** e.g. set of nodes V by creating an edge between nodes in V if they share at least on neighbour in U (& vice versa for U):

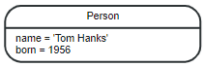


- **Self-loop**: Edge connecting a node with itself, forming a loop
- **Multi-graph**: Graph where multiple edges have same source & target nodes. May or may not contain self-loops. E.g. multi-graph with self-loops:

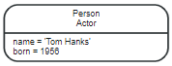


- Often an edge will have some associated *weight* indicating some attribute of the relationship connecting 2 nodes, where the nodes themselves will have some label(s) & attribute
- **Node labels**: Something that can be used to group nodes into a set, where nodes with the same label belong to the same set. Each node can have single or multiple labels.

  E.g. Single label ('person'):



  E.g.2. Multiple labels ('person', 'actor'):



- **Node or edge attributes/properties/features**: Add qualities to nodes/edges. In above example, node has attributes 'name' and 'born'
- **Spatial-temporal graph**: A graph whose node attributes change dynamically over time.

Representing Graphs

- Exact position, length or orientation of graph edges have no meaning, therefore can visualise graph in different ways by rearranging nodes & distorting edges so long as underlying topology/structure/relationships do not change
- **Adjacency matrix**: Way of representing a graph. Graph G can be represented by adjacency matrix $\boldsymbol{A}$ such that $A_{ij} = 1$ if node i and node j are linked by an edge, and $A_{ij} = 0$ otherwise.

  A graph with N nodes will have an $N \times N$ adjacency matrix

  E.g. Graph with 3-clique on nodes 1, 2 and 3 and an edge from node 3 to node 4:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \tag{13.5}$$

  Undirected graphs have a symmetric adjacency matrix

- *But* most real-World graphs are **sparse** (i.e. they have a very low number of connections $|E|$ relative to the maximum number of connections they could have were they a complete graph $E_{\max}$, or they have a very small average node degree $\bar{k}$ relative to the maximum average degree for a complete graph $|N| - 1$). Therefore most elements in adjacency matrix will be 0s, which is undesirable since large amount of memory used just for storing 0s

- To solve above, represent graph as a **list of edges** i.e. list of source-target node pairs. Makes looking up edges harder, but preservers memory

Graph Connectivity

- For undirected graphs:
  - **Connected graph**: An undirected graph with a path between any possible node pair in the graph
  - **Disconnected graph**: An undirected graph made up of ≥ 2 connected sets/sub-graphs
  - **Bridge edge**: An edge which, if removed, would cause undirected graph to become a disconnected graph
  - **Articulation node**: A node which, if removed, would cause undirected graph to become a disconnected graph
- For directed graphs:
  - **Strongly connected graph**: A directed graph with a path from each node to every other node, and vice versa
  - **Weakly connected graph**: A directed graph that is connected (has path between any possible node pair) if we disregard edge directions (i.e. assume is undirected graph)
  - **Strongly connected components (SCCs)**: Strongly connected sub-graphs of graph G. Nodes in G that can reach a given SCC are part of the SCC's **in-component**, and nodes in G that can be reached from a given SCC are part of the SCC's **out-component**

13.2.2. Key Graph Properties

Degree Distribution

Path Length

Clustering Coefficient

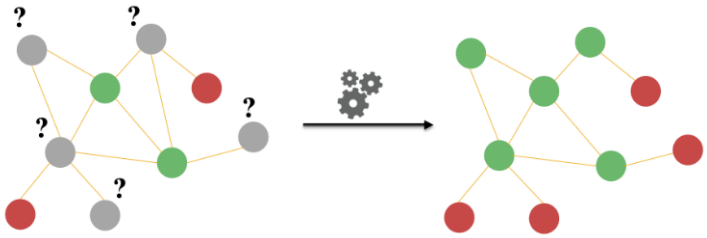
Connected Components

13.3. Machine Learning with Graphs

13.3.1. Node Classification Techniques

- (N.B. Similar methods apply for link classification techniques)

- **Node classification**: Process of taking a graph G and assigning **labels** (i.e. classes/attributes/values) to nodes in the graph given a set of existing node labels within the graph. E.g. (labels here are colours):



  Useful for very large graphs (e.g. social media network) since can take a small sample of labels and generate trustworthy node label predictions for the rest of the nodes in the graph (prevents having to collect ground-truth label for every node, which is often impossible for large graphs)

- **Collective classification**: How node classification is done in practice. First collectively assign labels to all nodes in graph, then propagate information from these labels around graph until have **stable** label assignments for each individual node. Relies on the **Markov assumption for graphs**

Graph Correlations

- **Markov assumption for graphs**: Assume that graph nodes are *correlated* in that the label Y_i of node i depends on the labels of its neighbours N_i:

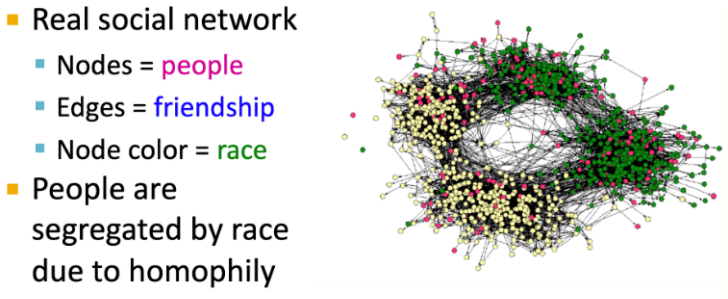
$$P(Y_i|i) = P(Y_i|N_i) \tag{13.6}$$

- 3 main correlation phenomena in graphs:

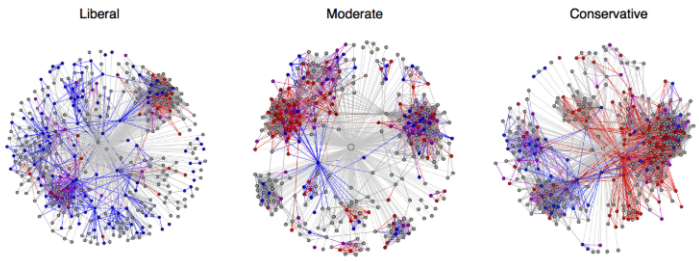
  1. **Homophily**: The tendency of nodes with similar attributes to form connecting edges.

     For e.g. a social network, similar attributes include age, gender, organisational affiliation, taste, race etc.

     E.g. Homophily phenomena by race occurring in social media friendship graph:



E.g.2. Homophily by political views occurring in social media friendship graph:

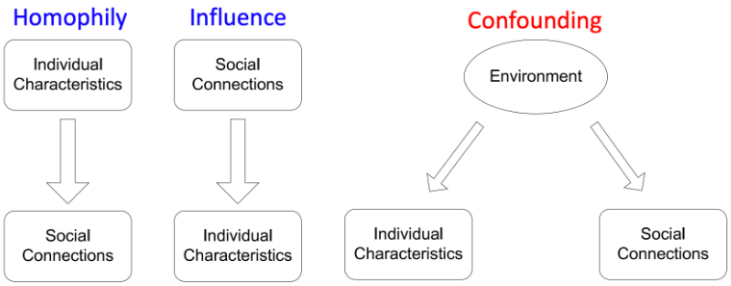


2. **Influence**: The tendency of nodes connected by edges to adopt similar node attributes.

   E.g. A friend recommending a musical preference, which you then become interested in an passing on to your other friends

3. **Confounding**: The tendency of a confounding variable (a variable that influences both the dependent and independent variable) to cause nodes spuriously associated with the confounding variable to adopt a range of similar node attributes.

   E.g. The environment children are brought up in causing them to have a range of similar attributes such as language spoken, music taste, political preferences etc.



Using Graph Correlations for Collective Classification

- The Markov assumption tells us that connected nodes are more likely to share the same attributes. To make node label predictions, must therefore consider the attributes of each node's neighbouring nodes and account for the graph topology
- 3 key components/stages needed for collective classification:
  1. **Local classifier**: Assigns initial node labels. Makes initial label prediction based on existing node attributes/features without considering any network information (i.e. ignores neighbouring nodes, topology etc.)
  2. **Relational classifier**: Predicts node label based on labels and features of neighbouring nodes (i.e. incorporates network information). I.e. captures local correlations

3. **Collective inference**: Propagates correlations through whole network (i.e. propagates correlations beyond immediate neighbours). Graph topology has significant impact on final predictions

- 3 main techniques for node classification:
  1. Relational classification
  2. Iterative classification
  3. Belief classification

Relational Classification

- Initialise by assigning ground-truth labels to known nodes and use a uniform label probability value for all unknown nodes (or some known prior so that have better initial starting position than just uniform probability across all possible labels)
- Choose a random node i and update label Y_i of node i by using the weighted average of the label/class probabilities of its neighbours. Continue updating node labels in random order until converge on label probabilities for each node
- Disadvantages:
  1. Convergence not guaranteed
  2. Does not use node feature information when performing updates, only labels of neighbouring nodes

Iterative Classification

- Uses node attributes as well as node labels to perform label updates, therefore can get more accurate label predictions than relational classifiers
- **Bootstrap phase**: Initialise flat vector a_i for each node i summarising node attributes, train a *local classifier* $f(a_i)$ which takes vector a_i and predicts node label, and then aggregate neighbour labels (using e.g. mode, mean etc.) & combine with predicted label to get initial label Y_i for each node
- **Iteration phase**: For each node i, update node attribute vector a_i and update label assignment Y_i to output of $f(a_i)$. Iterate through nodes repeatedly until converge on node labels
- Disadvantages:
  1. Convergence not guaranteed

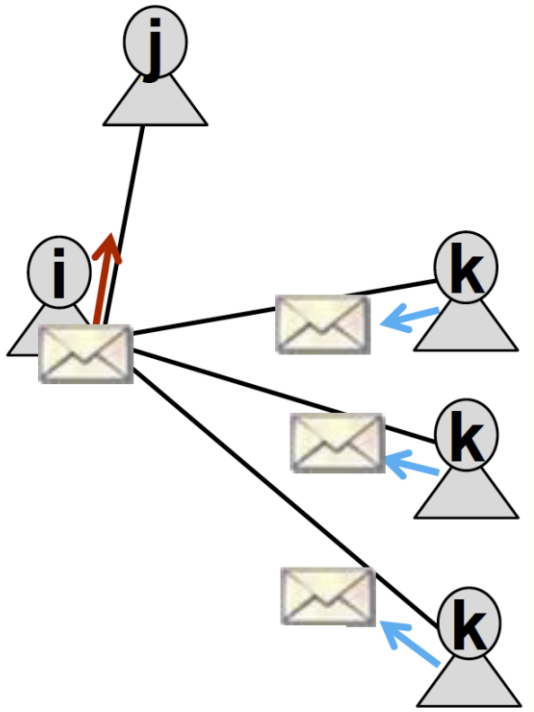
Belief Classification with Message Passing

- **Belief propagation**: Dynamic programming technique to get conditional label (a.k.a. state) probabilities for nodes in graphs. Uses an iterative process where every neighbouring node 'talks' to one another by **message passing**, and a given node label is updated by **aggregating** its neighbouring nodes' messages (e.g. by taking the **weighted average** of its neighbouring nodes' messages). The messages being passed from e.g. node i to node j correspond to i's belief that j is in a certain state.
- Notation for message passing:
  - **Label-label potential matrix** ψ: Captures the correlation/dependency between each node and its neighbours in a graph.

    E.g. $\psi(Y_i, Y_j)$ is the probability of node j being in state Y_j given its neighbour i is in state Y_i
  - **Prior belief** $\phi(Y_i)$: Probability of node i being in state Y_i (N.B. Can also be learned)
  - **Message** $m_{i \longrightarrow j}(Y_j)$: The message being passed from node i to node j, which represents i's estimate of j being in state Y_j
  - L: Set of all possible states
- To get the message that i will send to j about its belief/estimated probability that j is in state Y_j, sum over all states the label-label potential multiplied by our prior, multiplied by the product of all the messages sent by neighbours from previous rounds:

$$m_{i \longrightarrow j}(Y_j) = \alpha \sum \psi(Y_i, Y_j) \cdot \phi_i(Y_i) \prod_{k \in N} m_{k \longrightarrow i} \forall L \tag{13.7}$$

- E.g. Consider we want to find the message being sent from node i to node j. Initialise all messages as being equal to 1. The message i sends to j will depend on its neighbours k, where each neighbour k passes a message to i regarding its belief of the label/state of i. These messages will be aggregated to update the state of i, which will influence the message i passes to j (as described in equation above):



- I.e. To update the state of each node in the graph with message passing, each node will have an associated message passing computation graph (as above) which follows the above equation to compute the messages & the corresponding predicted state for a given node
- Repeat this for each node until converge on node labels
- Advantages:
    1. Each node's message passing process is a separate computation graph therefore can easily parallelise
    2. Can apply to any graph, therefore very general
- Disadvantages:
    1. Convergence not guaranteed

13.3.2. Learning Node Representations

Embeddings

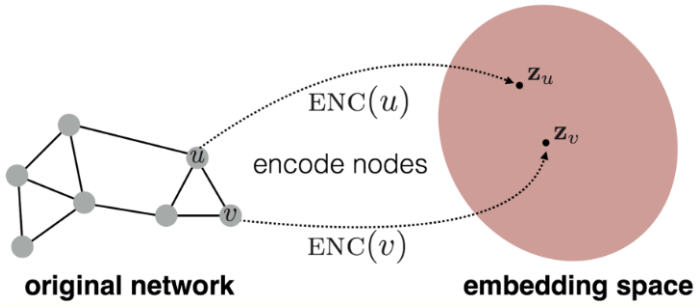
- To make graphs processable by ML methods, we want to create **embeddings** which transform the graph into a vector/set of vectors whilst maintaining the graph's topology,

node relationships, sub-graphs etc. Typically, embedding representations have a lower number of dimensions than the original graph representation

- 2 types of embeddings:
  1. **Node embedding**: Each node is encoded with its own vector representation. Used when want to do node-level prediction (e.g. predicting relationships between people based on node label/attribute similarities)
  2. **Graph embedding**: The whole graph is encoded with a single vector representation. Used when want to do graph-level prediction (e.g. classifying toxic vs. non-toxic chemical structures)
- Use embeddings because:
  1. Allows us to use rich vector-space mathematical tool set & well-developed ML techniques
  2. Embeddings are compressed (lower dimension) representations of graphs. For large graphs with e.g. 1 million nodes, if didn't have embedding representations, would have e.g. a 1 million $\times$ 1 million adjacency matrix, which is almost impossible to store & process
  3. Vector ops are more simple & faster than ops performed on entire graphs
- Challenges with embedding:
  1. Embeddings must preserve original graph properties
  2. Embedding method must be scalable to very large graphs with millions of nodes
  3. Number of dimensions in embeddings must be large enough to sufficiently encapsulate original graph properties whilst being small enough to be computationally tractable to process (literature methods often use embedding vector sizes between 128 and 256)
- Embedding methods have 2 components:
  1. **Encoder/mapping function (ENC)**: A function for mapping e.g. nodes of a graph onto a low-dimensional embedding space.

     E.g. Mapping function embedding nodes u and v to low-dimensional vectors $\boldsymbol{z}_u$ and $\boldsymbol{z}_v$ respectively:

     $$\text{ENC}(v) = \boldsymbol{z}_v \tag{13.8}$$



2. **Similarity Function**: A function that evaluates how similar a given pair of nodes in the original graph are. Encoder function's parameters should be optimised such that similarity of u and v in the original graph is approximately equal to the similarity (e.g. dot product) between their corresponding embedding representation vectors:

$$\text{similarity}(u, v) \approx \boldsymbol{z}_u^T \cdot \boldsymbol{z}_v \tag{13.9}$$

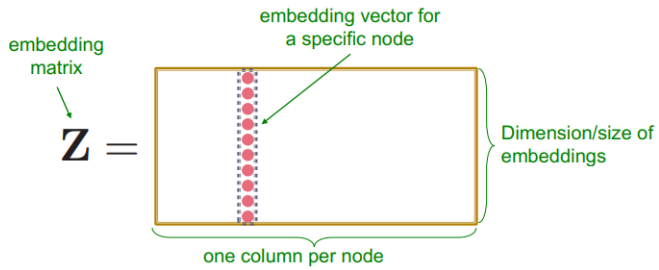
- 2 main methods for defining encoder to map nodes in graph onto embedding space:
  1. **Shallow encoding**: Use a single-layer (e.g. 1 hidden layer) data transformation
  2. **Deep encoding**: Use multiple layer data transformations (i.e. use deep NNs)

Shallow Encoding

- Most simple approach. Encoder is just a look-up matrix, termed the **embedding matrix** $\boldsymbol{Z}$. Each column $\boldsymbol{Z}$ is an embedding vector for a specific node, and each row is a dimension of the embedding vector:

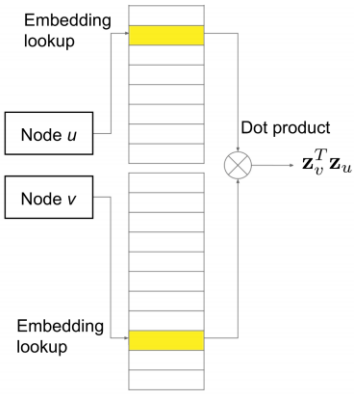
$$\text{ENC}(v) = \boldsymbol{Z} \cdot \boldsymbol{v} \tag{13.10}$$

  Where $\boldsymbol{v}$ is the **indicator vector** whose elements are all 0 except one in column indicating node v



- I.e. shallow encoding methods use a single-layer data transformation. E.g. 1 hidden layer maps node u to embedding vector $\boldsymbol{z}_u$ via hidden layer/function h:

$$\boldsymbol{z}_u = h(\boldsymbol{z}_v, v \in N_R(u)) \tag{13.11}$$



- Many diff methods for generating node embeddings with shallow encoding, defined by how they define node similarity in order to train the node embedding lookup table/single-layer data transformation function:
  1. **Random Walk**: Most simple method
     - Follow random walk strategy R (e.g. choose random node in graph as starting point, from starting point move to random neighbour node v and record that visited v, then move to another random neighbour from v and record, etc etc for fixed number of steps (i.e. DeepWalk algorithm from Perozzi et al. 2013; a fixed-length (e.g.g 32 steps) unbiased random walk strategy from each node)). Use results from random walk experiment to define similarity(u, v) as probability that u and v will both be visited in a random walk over the graph
     - Take defined random walk similarity and use to optimise the embedding matrix vectors such that the similarity (e.g. dot product) between the 2 corresponding vector embeddings is approximately equal to the random walk similarity for all pairs of nodes
  2. **Node2Vec**: Problem with random walk is follow random strategy and therefore do not explore node neighbourhood very well, therefore losing some graph characteristics and insufficiently defining the similarity between nodes. Node2vec is a modified version of DeepWalk where, rather than following unbiased random walk, follow biased random walk strategy
     - Define 2 parameters; Q (probability that random walk moves to unvisited node in graph), and P (probability that random walk returns to a previously visited node). Higher Q encourages exploration of larger neighbourhood of node, whereas larger P encourages exploration of local neighbours of node
     - Rest of algorithm same as DeepWalk above
  3. **TransE**
  4. **Structural Deep Network Embedding (SDNE)**

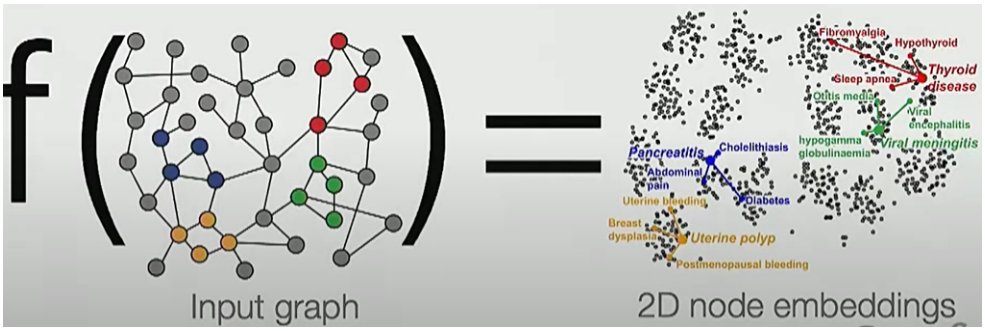
- Disadvantages of shallow encoding:
  1. Every node has its own unique embedding vector, therefore no sharing of embedding vector parameters even though there will be some nodes that could use same, therefore have very large number of parameters to tune when scale to larger graph, leading to difficulty with convergence, performance, and training times
  2. Cannot generate embeddings for nodes at inference for nodes that were not seen during training (since using a look-up matrix therefore must have embedding vectors for each possible node), therefore can't generalise to unseen graphs
  3. Only considers node labels when encoding nodes; does not consider node features

Deep Encoding

- Deep NNs solve above problems
- Use same methods for finding original graph similarity functions as shallow encoding (e.g. random walk, node2vec, TransE, SDNE etc.), but rather than tuning a single-layer transformation function/matrix to learn to map original nodes into vectors in the embedding space, uses multiple layers of non-linear transformations (i.e. deep NNs) to do mapping

13.4. Overview of Graph Neural Networks

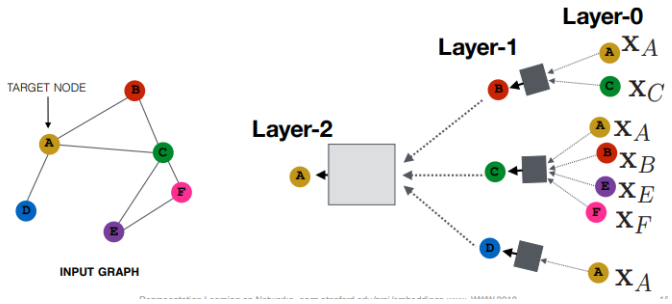
- **Graph Neural Network (GNN)**: Any deep learning approach applied to graph-structured data. Were first used in 2005 & expanded on in 2009, and over last decade have becoming increasingly hot area of research
- Like the graph encoding functions we saw in previous section, GNNs also embed graphs or nodes into vector spaces by learning a mapping function. However, when people refer to GNNs, they usually refer to an end-to-end model that can extract high-level representations from nodes/graphs & make predictions rather than only as a graph encoder.
- Essentially want to learn a mapping function that takes a graph and maps it onto 2D node embeddings, which we can use for various tasks:



- GNNs are based on graph embedding techniques such as DeepWalk, but look to generalise to unseen &/or dynamic graphs. As will be discussed, most GNNs use similar information diffusion process (**message passing**) to aggregate neighbouring node information & update a hidden embedded vector representation h_v of a given node v. This aggregation is done via e.g. CNN kernels, RNN units etc.
- I.e. 'message passing' in graphs or 'information propagation' or 'aggregation' might interchangeably refer to e.g. 'graph convolution' or 'LSTM aggregator' or 'pooling aggregator' (see later)
- Fundamentally, GNNs use the graph structure and associated node features as inputs to learn either a representation vector of a node h_v (node-level) or of an entire graph h_G (graph-level). Modern GNNs usually do this using a neighbourhood aggregation strategy where iteratively update node representations by aggregating neighbouring node representations (with e.g. convolution, pooling etc.) such that that spatial information in the graph is captured in the converged-on node representations. Can then use these representations for e.g. node label prediction

Aggregation & Training

- GNN problems can be framed as taking a graph G with a set of nodes/vertices V and an adjacency matrix A, where X is a matrix of node features (e.g. categorical attributes (profile info for social media etc), node degrees, clustering coefficients, one-hot encoded indicator vectors etc). Goal of GNN is to generate node embeddings by aggregating neighbouring node info
- I.e. To update a given node, aggregate its neighbouring nodes K steps/hops away. E.g. for $K = 2$, to update target node A to update embedding representation z_A, would form the following computation graph:

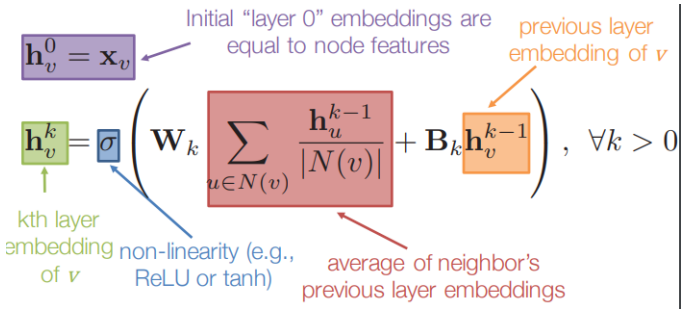


Where each box is an aggregated representation of the node inputs. Layer 0 (furthest layer/highest number of steps away from central node) will have node embeddings as the raw input features x_u, and layer K embedding will get aggregated emmbedded state representation information from nodes that are K hops away. I.e. can think of this as a K-layer NN whose output is the embedded feature representation of node A!

The key thing distinguishing different GNN approaches is *how* information is aggregated across these computation graph layers i.e. what's in the boxes in above diagram? Which aggregators are best is hot area of current research!

- Each node in the graph forms its own one of these computation graphs. In this sense, they key to using ML on GNNs (where don't have regular fixed grid of nodes as with images or vector as with text) is that the input graph data itself (local structure etc.) defines the computation graph and how we aggregate/propagate info/do convolutions/do ML using the features & structure of the local neighbourhood (up to K hops away from the node we're updating):

- Basic approach to aggregation (i.e. what we put in each box in above diagram) is to use a NN by averaging the neighbour's previous layer embeddings and applying a non-linearity (e.g. relu) to get the hidden k-th layer embedding representation of node v, h_v^k. Crucially, aggregation must be **order-invariant** (i.e. if changes order of set of nodes we're aggregating, would get same result), which is why we use e.g. mean pooling, max pooling, sum pooling etc. (below using mean/average pooling):



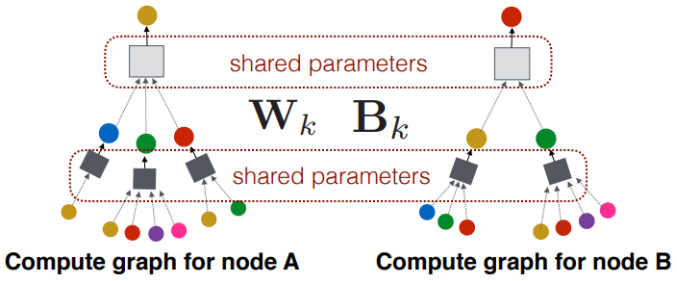
In above equation, $\boldsymbol{W}_k$ are the weights of the NN used to aggregate the neighbour's features, and $\boldsymbol{B}_k$ are the weights of the NN used to aggregate a node's own features (i.e. have 1 W and 1 B matrix per layer in our NN/number of hops K in our aggregation computation graph). Therefore GNN learns how to balance these 2 contributions when generating a node embedding with aggregation

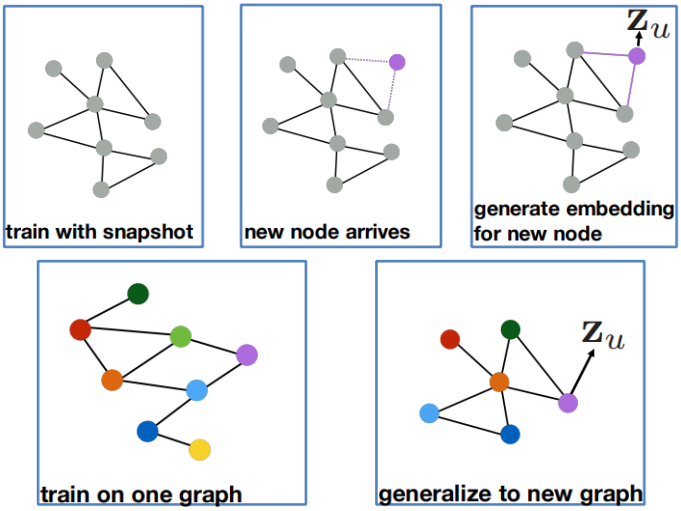
Do this for all k layers (in above diagrams, have up to $K = 2$), at which point will have output embedding for node v, z_v

- Can then train NN (i.e. train parameters $\boldsymbol{W}_k$ and $\boldsymbol{B}_k$ in above equation) by passing output z_v through a loss function, $L(z_u)$ and using e.g. stochastic gradient descent to optimise NN parameters
  - **Unsupervised learning**: Use only the graph structure to make predictions. Loss function could be random walks (e.g. node2vec or DeepWalk from prev section, graph factorisation, similarity function where ensure similar nodes have similar embeddings etc.)
  - **Supervised learning**: Use node attributes to do e.g. node classification using e.g. cross-entropy loss function
- I.e. 2 design choices for GNN:
  1. Define a neighbourhood aggregation function (i.e. the boxes in above diagram)
  2. Define a loss function that takes the output node embeddings z_u and calculates a loss value with which to optimise neighbourhood aggregation function parameters

Inductive Capability

- Many real graphs have millions of nodes, need GNN to be able to generalise to being able to generate embedding representations for unseen nodes as they're encountered
- For GNN to be **inductive**, must share the same aggregation parameters (**parameter sharing**) $\boldsymbol{W}_k$ and $\boldsymbol{B}_k$ for embedding all nodes. This allows trained GNN to generalise to unseen nodes & graphs





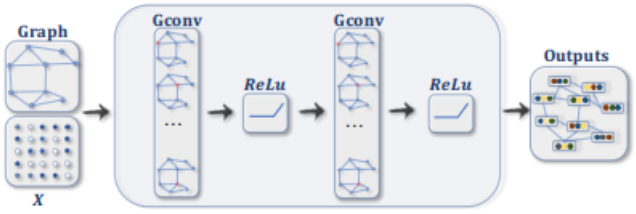
13.4.1. Graph Categories

- There are various different types of graphs which graph-based data can be categorised into (and provided as input to a GNN):
  - **Directed graph**: Most simple graph type to use for ML. Is a graph whose edges have an associated direction
  - **Heterogeneous graph**: A graph with different types of nodes and edges (e.g. some nodes might be an image, other nodes might be text).
  - **Dynamic /spatial-temporal graph**: A graph with a dynamic graph structure &/or dynamic input signals (e.g. dynamic node or edge attributes) varying in time.

13.4.2. GNN Categories

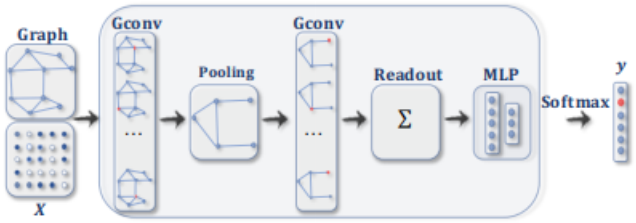
- 4 categories of GNN (see subsequent sections):
  1. **Recurrent graph neural networks (RecGNNs)**: Early GNN implementations were RecGNNs. Use recurrent neural architectures where use iterative **message passing** mechanism (see last section) to exchange information between nodes until a stable equilibrium is found. Although most research has shifted to using convolutions (ConvGNNs & STGNNs, see below), message passing forms the basis of STGNNs, and RecGNNs could still be combined with more modern approaches to gain new insights
  2. **Convolutional graph neural networks (ConvGNNs)**: GNN approaches which use the **convolution** operation to aggregate a given node's features with its neighbours to generate an abstract representation of the node

     E.g. ConvGNN for node-level classification:



(a) A ConvGNN with multiple graph convolutional layers. A graph convolutional layer encapsulates each node's hidden representation by aggregating feature information from its neighbors. After feature aggregation, a non-linear transformation is applied to the resulted outputs. By stacking multiple layers, the final hidden representation of each node receives messages from a further neighborhood.

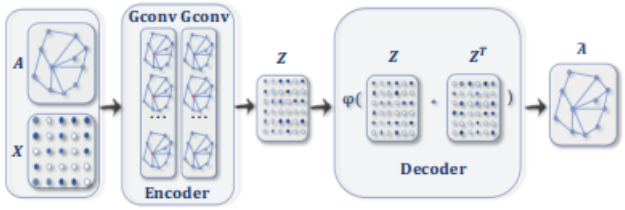
E.g. ConvGNN for graph-level classification:



(b) A ConvGNN with pooling and readout layers for graph classification [21]. A graph convolutional layer is followed by a pooling layer to coarsen a graph into sub-graphs so that node representations on coarsened graphs represent higher graph-level representations. A readout layer summarizes the final graph representation by taking the sum/mean of hidden representations of sub-graphs.

3. **Graph autoencoders (GAEs)**: Learn **embedding** function to encode nodes/graphs into low-dimensional vector space

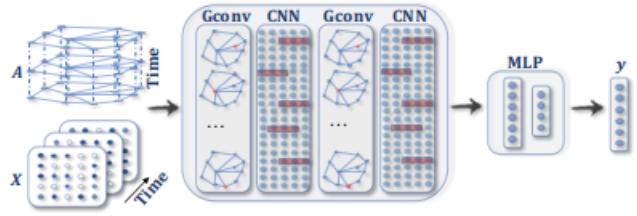
   E.g. GAE for node embedding:



(c) A GAE for network embedding [61]. The encoder uses graph convolutional layers to get a network embedding for each node. The decoder computes the pair-wise distance given network embeddings. After applying a non-linear activation function, the decoder reconstructs the graph adjacency matrix. The network is trained by minimizing the discrepancy between the real adjacency matrix and the reconstructed adjacency matrix.

4. **Spatial-temporal graph neural networks (STGNNs)**: Simultaneously consider node spatial and temporal dependencies to learn hidden patterns for spatial-temporal graphs. Important for e.g. traffic speed forecasting, driver maneuver anticipation, human action recognition etc.

E.g. STGNN for spatial-temporal graph forecasting:



(d) A STGNN for spatial-temporal graph forecasting [74]. A graph convolutional layer is followed by a 1D-CNN layer. The graph convolutional layer operates on A and $X^{(t)}$ to capture the spatial dependency, while the 1D-CNN layer slides over X along the time axis to capture the temporal dependency. The output layer is a linear transformation, generating a prediction for each node, such as its future value at the next time step.

13.4.3. GNN Frameworks

- All GNNs take graph-structure input data with node and/or edge information

GNN Output Mechanisms

- 3 categories of GNN output depending on the graph task being performed:
  1. **Node-level**: Used for when learning to classify nodes. Use e.g. softmax layer at output to get probability distribution over possible classes for each node.
  2. **Edge-level**: Used for when learning to classify edges or predict where links will occur between nodes. GNN produces 2 nodes' hidden representations & inputs into a similarity function/NN, which in turn predicts the label/connection strength of the edge
  3. **Graph-level**: Used for when learning to classify whole graph. To get compact graph representation, often combine GNNs with **pooling** and **readout** operations

GNN Training Frameworks

- Different ways to train GNNs depending on (i) task & (ii) label information available:
  - **Semi-supervised learning for node-level classification**: Given single graph with some nodes labelled and others unlabelled, label the unlabelled nodes (i.e. is what we saw in previous example for colouring the uncoloured nodes). In practice often do this with ConvGNNs
  - **Supervised learning for graph-level classification**: Given sets of labelled graphs, learn to classify whole graphs. In practice often do this by first using ConvGNNs to extract high-level node representations, then using pooling to down-sample these node representations to lower dimensional node representations, and then

using a readout function to collapse the node representations into a single graph representation

  - **Unsupervised learning for graph embedding**: If have no labelled graph data, can learn graph embedding function. Usually 2 possible methods (i) Train a simple autoencoder where encoder uses ConvGNN layers to encode graph into vector embedding representations, and a decoder to use the embedding representations to reconstruct the graph structure (ii) Use *negative sampling* (samples portion of node pairs and sets them as positive pairs, and sets existing node pairs with edges as positive pairs, and uses logistic regression to distinguish between positive and negative pairs)

- Summary of (older) RecGNN techniques with (more modern) (spectral or spatial based) ConvGNN techniques, indicating pooling & readout functions used if they were applied to graph-level classification (node & edge level classification do not require pooling or readout techniques):

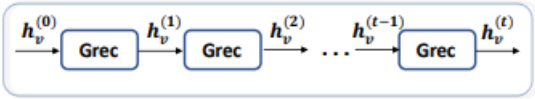
TABLE III: Summary of RecGNNs and ConvGNNs. Missing values ("-") in pooling and readout layers indicate that the method only experiments on node-level/edge-level tasks.

Approach	Category	Inputs	Pooling	Readout	Time Complexity
GNN* (2009) [15]	RecGNN	A, X, X^e	-	a dummy super node	$O(m)$
GraphESN (2010) [16]	RecGNN	A, X	-	mean	$O(m)$
GGNN (2015) [17]	RecGNN	A, X	-	attention sum	$O(m)$
SSE (2018) [18]	RecGNN	A, X	-	-	-
Spectral CNN (2014) [19]	Spectral-based ConvGNN	A, X	spectral clustering+max pooling	max	$O(n^3)$
Henaff et al. (2015) [20]	Spectral-based ConvGNN	A, X	spectral clustering+max pooling		$O(n^3)$
ChebNet (2016) [21]	Spectral-based ConvGNN	A, X	efficient pooling	sum	$O(m)$
GCN (2017) [22]	Spectral-based ConvGNN	A, X	-	-	$O(m)$
CayleyNet (2017) [23]	Spectral-based ConvGNN	A, X	mean/graclus pooling	-	$O(m)$
AGCN (2018) [40]	Spectral-based ConvGNN	A, X	max pooling	sum	$O(n^2)$
DualGCN (2018) [41]	Spectral-based ConvGNN	A, X	-	-	$O(m)$
NN4G (2009) [24]	Spatial-based ConvGNN	A, X	-	sum/mean	$O(m)$
DCNN (2016) [25]	Spatial-based ConvGNN	A, X	-	mean	$O(n^2)$
PATCHY-SAN (2016) [26]	Spatial-based ConvGNN	A, X, X^e	-	sum	-
MPNN (2017) [27]	Spatial-based ConvGNN	A, X, X^e	-	attention sum/set2set	$O(m)$
GraphSage (2017) [42]	Spatial-based ConvGNN	A, X	-	-	-
GAT (2017) [43]	Spatial-based ConvGNN	A, X	-	-	$O(m)$
MoNet (2017) [44]	Spatial-based ConvGNN	A, X	-	-	$O(m)$
LGCN (2018) [45]	Spatial-based ConvGNN	A, X	-	-	-
PGC-DGCNN (2018) [46]	Spatial-based ConvGNN	A, X	sort pooling	attention sum	$O(n^3)$
CGMM (2018) [47]	Spatial-based ConvGNN	A, X, X^e	-	sum	-
GAAN (2018) [48]	Spatial-based ConvGNN	A, X	-	-	$O(m)$
FastGCN (2018) [49]	Spatial-based ConvGNN	A, X	-	-	-
StoGCN (2018) [50]	Spatial-based ConvGNN	A, X	-	-	-
Huang et al. (2018) [51]	Spatial-based ConvGNN	A, X	-	-	-
DGCNN (2018) [52]	Spatial-based ConvGNN	A, X	sort pooling	-	$O(m)$
DiffPool (2018) [54]	Spatial-based ConvGNN	A, X	differential pooling	mean	$O(n^2)$
GeniePath (2019) [55]	Spatial-based ConvGNN	A, X	-	-	$O(m)$
DGI (2019) [56]	Spatial-based ConvGNN	A, X	-	-	$O(m)$
GIN (2019) [57]	Spatial-based ConvGNN	A, X	-	sum	$O(m)$
ClusterGCN (2019) [58]	Spatial-based ConvGNN	A, X	-	-	-

13.5. Recurrent GNNs

13.5.1. Overview

- Early pioneer GNN works used RecGNNs
- As with other recurrent NNs, RecGNNs use the same graph layer (Grec) to update node representation h_v for each node:



13.5.2. RecGNN Case Studies

'Original' Graph Neural Network (GNN*)

- Scarselli et al., 2009, 'The Graph Neural Network Model', https://ieeexplore.ieee.org/document/4700287
- Generalised RNNs to handle graph data based on **message passing**, which allows nodes to be updated based on neighbourhood information recurrently until a stable equilibrium was reached (see prev section). Refer to their model as GNN*
- Hidden representations h_v^t of node v at time t (where $\boldsymbol{h}_v^0$ is randomly initialised) recurrently updated by:

$$\boldsymbol{h}_v^t = \sum_{u \in N(v)} f\left(\boldsymbol{x}_v, \boldsymbol{x}_{(u,v)}^{\boldsymbol{e}}, \boldsymbol{x}_u, \boldsymbol{h}_u^{(t-1)}\right) \tag{13.12}$$

  Where $f()$ is a function (e.g. a NN) recurrently applied and is a contraction mapping function (i.e. projects node representation on low-dimensional space such that distance between 2 points after projection is less than before)
- When node states converge, hidden states passed through a **readout layer**, where readout layer is a trainable function which takes node-level representations and outputs a graph-level representation, which can then be classified by e.g. a softmax layer. Is very similar to pooling layers, difference is mainly that pooling is for downsampling within the network whereas readout is for final step of using node-level representations to get graph-level representations (i.e. only really use readout layer if want graph-level outputs)

Graph Echo State Network (GraphESN)

- **Echo state network (ESN)**: An RNN with a sparsely connected input layer (termed the **reservoir** layer) with around 1% connectivity and whose weights are randomly assigned and not trainable. Weights of output layer (termed **readout** layer) are trainable. Reservoir layer transforms inputs into encoded representations, and embeddings are then forwarded to output layer with its trainable units. ESNs have relatively few parameters to train (due

to untrainable reservoir layer and sparse connections), therefore have quick training times and can handle irregular chaotic data. They are computationally cheap to implement & don't have vanishing/exploding gradients

- GraphESN extended idea of ESNs to handle graphs. Had an encoder layer (the reservoir) and an output layer (the readout layer), and recurrently updates node state representations until convergence (just like GNN*)

Gated Graph Neural Network (GGNN)

- Instead of vanilla RNN, used gated recurrent unit (GRU) as recurrent module
- Used back-prop through time (BPTT) algorithm to learn model parameters, which requires running recurrent function multiple times over all nodes, requiring intermediate hidden node state representations to be stored in memory (not scalable to large graphs)

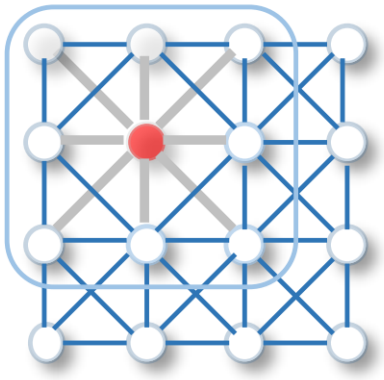
Stochastic Stead-state Embedding (SSE)

- Extends previous ideas to scale to large graphs. Recurrently updates hidden states in stochastic/random & asynchronous fashion by randomly sampling one batch of nodes for state updates and another batch for gradient computation
- Function that is recurrently applied with SSE is a weighted average of historical & new states, where the weights are learned

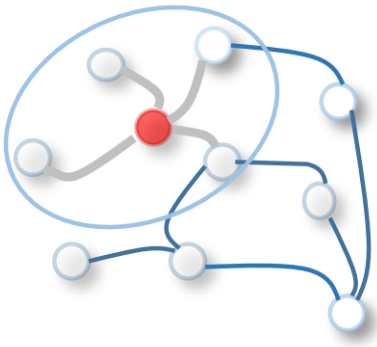
13.6. Convolutional GNNs

13.6.1. Overview

- Consider standard 2D convolution. Can construct image as a graph, where each pixel is a node. We know that 2D convolution applies a kernel/filter of some fixed size to take the weighted average of nodes within its receptive field as it slides along the image, producing a new state representation of the nodes in the receptive region by agglomerating neighbours:



- Similarly, can extend this idea of 2D convolution to **graph convolution**. To generalise to graphs, don't use fixed size kernel, but rather define kernel to include e.g. the immediate neighbours of a given node. Can then agglomerate by applying convolution (i.e. weighted average) of neighbouring nodes to get new state representation of node in red (N.B. This is a **spatial-based graph convolution** (see below)):



- Applying graph convolutions with deep graph convolutional neural networks to extract high-level node/edge/graph-level representations is more efficient than recurrent models and compatible with standard NN models, therefore ConvGNNs have become increasingly popular

Graph Pooling

- Just as with standard CNNs, pooling functions are used on the output of the convolution op (i.e. used on output of graph convolution layer) to produce a single summary statistic of local neighbouring values/nodes to help with:
  1. Invariance to local translations (and therefore improve generality)
  2. Downsample number of dimensions in embedding representation of node to make processing and learning over large graphs with many features more computationally

tractable (and therefore improve scalability)

3. Reducing number of features also helps with overfitting, further helping with generality

- Pooling functions and readout functions are more or less the same thing and often referred to interchangeably
- Just as with standard 2D CNNs, most popular pooling functions are e.g. max pooling, average pooling etc since evaluating the output of the pooling window is fast and simple and therefore scalable to large graphs
- Pooling is an important part of ConvGNNs, and has been combined with attention mechanisms to maximise its effectiveness (e.g. SAGPool considers both node features and graph topology and learns the pooling function in a self-attention manner). How to improve effectiveness and computational complexity of graph pooling is a hot area of research

ConvGNN Categories

- 2 Categories of ConvGNN:
    1. **Spectral-based ConvGNN**: Define convolution kernel/filter in terms of **graph signal processing**, where use the convolution op in the Fourier domain to remove noise from graph signals by computing the eigen-decomposition of the graph Laplacian. All spectral-based methods follow this principle, with difference being how kernel is chosen
    2. **Spatial-based ConvGNN**: Define convolution kernel in terms of **information propagation** (similar to RecGNNs) by considering the nodes' spatial relations (as opposed to graph signals); the convolution operation servers a similar node aggregation purpose of propagating neighbouring node information to update a central node's state as message passing serves for RecGNNs. Spatial-based ConvGNNs are analogous to conventional CNNs on images (as above). Tend to be most popular approach recently due to 3 key advantages:
        a) **Efficiency**: Spectral models either use eigenvectors or use whole graph for every computation, whereas spatial models perform convolutions in the graph domain via spatial information propagation (i.e. spatial convolution) and can therefore be scaled to larger graphs
        b) **Generality**: Spectral models using Fourier methods assume a fixed graph and therefore cannot generalise to different sized graphs without changing the eigenbasis, whereas spatial models perform convolutions on each node where weights used by conv op can be shared across different locations and structures and therefore can generalise to unseen & different sized graphs

c) **Flexibility**: Spectral models always assume graphs are undirected, but spatial models can easily incorporate directed graphs into aggregation function

13.6.2. Spectral ConvGNN Case Studies

Spectral Convolution Neural Network (Spectral CNN)

- Defines filter as set of learnable params & considers graph signals with multiple channels

Chebyshev Spectral CNN (ChebNet)

- Approximates filter with Chebyshev polynomials. Filters are therefore localised in space, therefore can extract local neighbourhood features independently of the size of the graph, making more scalable than spectral CNN

CayleyNet

- Approximates filter with Cayley polynomials to capture narrower frequency bands of graph signals. Turns out that ChebNet is a special case of CayleyNet

Graph Convolutional Network (GCN)

- Filter defined as first-order approximation of ChebNet filter, which restrains number of parameters (helps with learning etc.) and prevents over-fitting
- Can also be considered as a spatial-based method since it aggregates feature information from a node's neighbourhood with convolution

Adaptive Graph Convolutional Network (AGCN)

- Is a modified version of GCN. Uses a learned 'residual graph adjacency matrix' which takes 2 nodes' features as inputs & outputs a hidden state expressing the structural relationship between the 2 nodes

Dual Graph Convolutional Network (DGCN)

- SUses *two* graph convolution layers in parallel. Is able to encode both local and global graph structural information without needing to stack multiple graph convolutional layers

13.6.3. Spatial ConvGNN Case Studies

Neural Network for Graphs (NN4G)

- Deep convolution graph neural network i.e. had multiple layers with different params. Performed graph convolutions by summing nodes' neighbourhood information. Used skip & residual NN connections between layers so that information was memorised across layers

Diffusion Convolutional Neural Network (DCNN)

- Regards graph convolutions as a **diffusion** process where neighbouring nodes transfer info to one another with some transition probability so that can reach information distribution equilibrium after only a few rounds of information propagation.

Message Passing Neural Network (MPNN)

- A general framework for spatial-based ConvGNNs. Treats graph convolutions as **message passing** process where info passed from one node to another via edges
- Message passing function (i.e. the spatial convolution) generates hidden representations for each node, which is passed to an output layer (for node-level predictions) or a readout layer (for graph-level predictions) (N.B. Both output layer and readout layer have learnable parameters)

Graph Isomorphism Network (GIN)

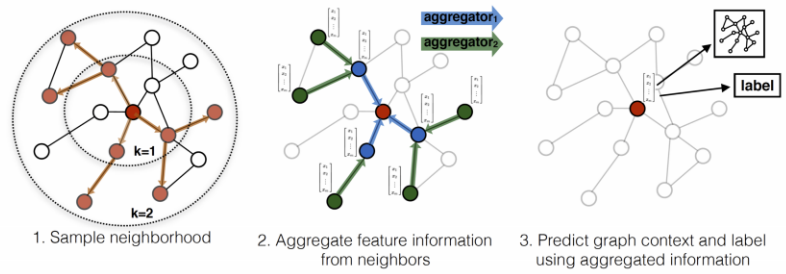
- An extension of MPNN. Found that, from the embedding representations produced by MPNN, couldn't distinguish between different graph structures, therefore some graph structure information had been lost. GIN fixed this by learning a weight to apply to the central node being updated

GraphSage

- W. Hamilton et al., 2018, 'Inductive Representation Learning on Large Graphs', https://arxiv.org/pdf/1706.02216.pdf
- Previous ConvGNN used full-batch approaches to training where store all graph data and hidden states of all nodes in memory and applied each convolution op to a node's entire neighbourhood. This is not scalable to large graphs which might have 1,000s of neighbours for a given central node.
- GraphSage ('sample and aggregate') fixes the number of neighbouring nodes to be considered for each convolution, where the specific neighbour nodes used for the convolution/aggregation update of a central node are randomly sampled. Did this by introducing

mini-batch training algorithm for ConvGNNs: Randomly selects fixed mini-batch size number of nodes, goes through each node in mini-batch and sets as root node, and samples a tree rooted at node by recursively expanding the root node's neighbourhood by K steps. For each tree, GraphSage computes root node's hidden state by aggregating its neighbouring tree node info with convolution op(s)

E.g. of how aggregation is node for K steps around central node's neighbours (drawings for $K = 1$ and $K = 2$):

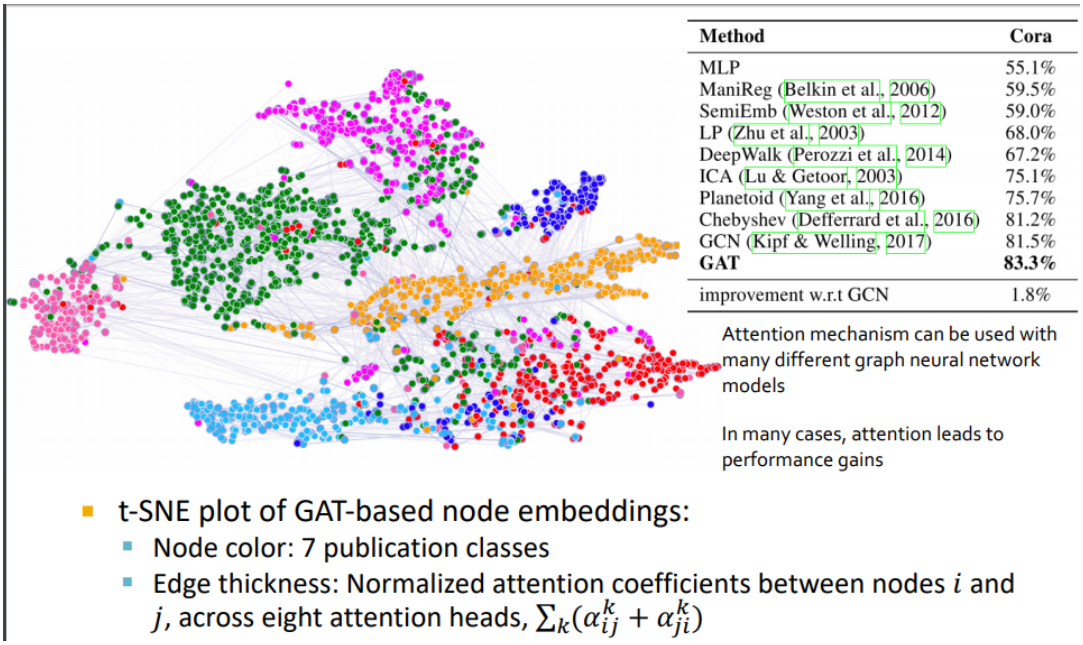


- All learning is done by learning an aggregation function which learns to aggregate neighbouring node features (e.g. text attributes, node profile information, node degrees etc) so that learned node embedding encompasses topological info of graph
- Applied to open-access Web of Science citation (predicting academic paper categories), Reddit (predicting post categories/subreddits), & the protein-protein interaction (PPI; classifying protein functions into PPI graphs) databases for node classification & had much better performance & generality & scalability than previous works
- Tested GraphSage by learning 4 diff aggregator functions; GraphSage-GCN, GraphSage-mean, GraphSage-LSTM (to use LSTM as aggregator and maintain order-invariance, must train LSTM over randomised ordering of nodes), & GraphSage-pool (mean & max). LSTM- & pool-based aggregators performed best, but pool is around 2x faster than LSTM
- GraphSage much greater generality and scalability than previous GNN approaches

Graph Attention Network (GAT)

- P. Velickovic et al., 2017, 'Graph Attention Networks', https://arxiv.org/abs/1710.10903
- GraphSage assumes that neighbour node contributions to updating the central node are all equally valuable, and GCN pre-determines which neighbouring nodes are used to update the central node
- GAT uses an attention mechanism whereby it learns how important each neighbouring node is to the central node and weights their contributions accordingly. Uses a softmax function to produce attention weights (i.e. all weights sum to 1)

- GAT also went further by using multi-head attention (i.e. multiple attention layers with separate trainable parameters) which improved the model's ability to express complex graphs in its hidden state representations
- Benchmarked on 4 open-access datasets; Cora, Citeseer, Pubmed (all citation databases), & PPI (protein-protein interaction dataset)
- E.g. applying to Cora citation network (each node is a paper, colour corresponds to class of the paper), GAT performs much bettern than previous attempts (MLP: Only consider text information, no graph structure. DeepWalk: Only consider graph structure, not node information. GCN: Consider neighbouring node information equally when aggregating, no attention distribution):



Method	Cora
MLP	55.1%
ManiReg (Belkin et al., 2006)	59.5%
SemiEmb (Weston et al., 2012)	59.0%
LP (Zhu et al., 2003)	68.0%
DeepWalk (Perozzi et al., 2014)	67.2%
ICA (Lu & Getoor, 2003)	75.1%
Planetoid (Yang et al., 2016)	75.7%
Chebyshev (Defferrard et al., 2016)	81.2%
GCN (Kipf & Welling, 2017)	81.5%
GAT	**83.3%**
improvement w.r.t GCN	1.8%

- Advantages of using attention mechanism in GNNs such that learn which neighbours are more important when updating given node:
  1. Computationally efficient (can parallelise across all edges and nodes in graph)
  2. Storage efficient (amount of info you have to store only scales linearly with number of edges + number of nodes)
  3. Localised (only attends over local network neighbourhoods when updating a node rather than having to attend over e.g. whole graph
  4. Inductive capability (can share attention layer parameters across network since does not depending on global graph structure)

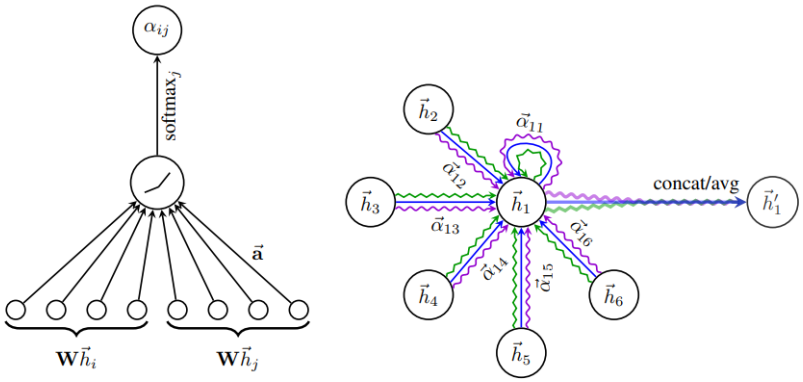


Figure 1: **Left:** The attention mechanism $a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_1$.

Gated Attention Network (GAAN)

- GAT assumed each head of the multi-head attention mechanism contributed equally to updates.
- GAAN introduced a new self-attention mechanism which computed an additional self-attention vector containing a score for each head

GeniePath

- Extended idea of GAANs by using spatial attention mechanism, but further applied an LSTM-like gating mechanism to control which information flowed across graph convolution layers, which is a different approach to prioritising information than e.g. attention mechanisms

Mixture Model Network (MoNet)

- Adopts different approach to assign different weights to node's neighbours. Uses 'pseudo coordinates' of each node to evaluate relative position between each node and its neighbour, then learns a function that maps these coordinates to a relative weight to apply between the nodes.
- Since different node pairs across the graph will have the same relative pseudo coordinates (i.e. relative position to one another), this allows the learned parameters of the relative weight mapping function to be shared, enabling much more efficient training
- Also introduced a Gaussian convolution kernel with learnable parameters to learn how to apply weighted average/convolution op

PATCHY-SAN

- Another approach which, like MoNet, was able to share weights across different locations in the graph.
- Ordered neighbours of each node according to their 'scores' (where each score summarised the node's degree, centrality, and Weisfeiler-Lehman colour, which are all parameters describing the graph structure) and selected the top q neighbours to contribute to central node update, therefore all updates had fixed number q of neighbouring nodes to consider.
- Could therefore take the q nodes and consider them as a grid structure (just like 2D image at start of this section) and apply standard 1D convolution kernel to aggregate the neighbourhood feature information. To weight contribution of each neighbour, applied weights corresponding to order to list of the q neighbours

Large Graph Convolutional Network (LGCN)

- PATCHY-SAN only considered graph structure when ranking node neighbours, which is computationally expensive to do and therefore cannot scale to larger graphs
- LGCN ranked node neighbours according to node feature information as opposed to graph structure parameters (degree, centrality etc). Then applied same method as PATCHY-SAN, but could also be scaled to large graphs since didn't need to calculate graph structures

Approaches to Increasing Training Efficiency:

Fast Graph Convolutional Network (Fast-GCN)

- Whereas GraphSage sampled fixed number of neighbours for convolution, Fast-GCN samples fixed number of nodes

Stochastic Training of Graph Convolutional Networks (StoGCN

- Reduced size of convolution kernel's receptive field to small scale using historical node representations as a control variate. Achieved good performance even with just 2 neighbours per central node in the receptive field

Cluster Graph Convolutional Network (Cluster-GCN)

- Samples sub-graph from graph using clustering algorithm and performs convolution on sub-graph to update node, where the sampled sub-graph has a restricted number of nodes. Can scale to larger graphs and use deeper NN architectures

13.7. Graph Autoencoder GNNs

13.7.1. Overview

- Use to map nodes from a graph into a low-dimensional embedding space, and to decode these low-dimensional embedding space representations and recover the original graph structure
- 2 applications of GAEs:
  1. **Networking embedding**: Processing of mapping a graph to a low-dimensional vector representation of either the nodes in the graph or the whole graph.

     GAEs have 2 components:

     a) **Encoder**: Maps graph nodes onto low-dimensional embedding space

     b) **Decoder**: Enforces that embeddings preserve the graph topological information (e.g. adjacency matrix, PPMI matrix etc.) such that can recover the original graph structure from the encoded graph
  2. **Generating new graphs**: Train GAE on multiple graphs to learn how to generate graphs by (1) encoding graphs into hidden representations and then (2) decoding the encoded representation to reveal the hidden representation. Used for e.g. generating new molecule graphs to discover new types of drugs.

     Can do this in either a **sequential** manner (generate nodes and edges step-by-step and then concatenate to form generated graph i.e. turns graphs into sequences before concatenating, therefore can lost structural information due to cyclic nature of graph generation), or in a **global** manner (output generated graphs all at once, but is not scalable to large graphs as takes up too much memory space to process/output/store whole graph all at once)

13.7.2. Network Embedding GAE Case Studies

Deep Neural Network for Graph Representations (DNGR)

- One of first attempts to build a GAE. Like many early attempts, used standard linear/dense/fully-connected NN layers to form multi-layer perceptron NN (MLP)
- Used stacked denoising autoencoder made up of MLPs to encode & decode PPMI matrix

Structural Deep Network Embedding (SDNE)

- Used same structure as DNGR, but applied 2 loss functions to output during training; one to enable learning how to make embedding preserve node first-order proximity by minimising distance between node's network embedding and its neighbour's network embedding, and the other to enable learning how to make network embedding

preserve second order proximity by minimising distance between node's inputs and its reconstructed inputs

Graph Autoencoder (GAE*)

- DNGR and SDNE only use node structural information (i.e. the connectivity between node pairs), but ignore node attributes
- GAE* leverages GCN to encode node structural information *and* attribute information simultaneously using 2 graph convolution layers and a decoder layer that reconstructed graph adjacency matrix

Variational Graph Autoencoder (VGAE)

- Found that GAE* only reconstructing adjacency matrix would make GAE* overfit and not generalise to unseen graphs
- VGAE is a variant of GAE where scale the second loss function output with a normal distribution so that graphs with similar adjacency matrices get mapped to similar places in the embedding representation, which helps with generality and scalability

Adversarially Regularised Variational Graph Encoder (ARVGA)

- Uses same training scheme as generative adversarial networks (GANs), where have a competition between a generator model and a discriminator model and train generator (i.e. the encoder) to generate as accurate embedding representations as possible

Deep Recursive Network Embedding (DRNE)

- Previously discussed models learn network embeddings by solving link prediction problem. BUT sparsity of real graphs can cause number of positive node pairs to be << negative node pairs, which can make learning difficult
- DRNEs take different approach to link prediction models to do network embedding. Work by converting graph into sequences by doing random walks, and therefore can use standard sequence ML models directly on graphs.
- DRNE used LSTM network to aggregate random sequence of node neighbours ordered by their node degree

Network Representations with Adversarially Regularised Autoencoders (NetRA)

- Also used LSTM for encoder and decoder with node sequences generated by random walks and orders them to provide ordered sequence as input to model. Regularises learned

network embeddings with prior distribution via adversarial training similar to ARVGA

13.7.3. Graph Generation GAE Case Studies

1. Sequential Approaches:

SMILES

- Use deep CNN has encoder and deep RNN as decoder
- Generate a string representation of molecular graphs
- Is specific to domain of molecular graph generation

Deep Generative Model of Graphs (DeepGMG)

- Iteratively adds nodes and edges to a growing graph until some criterion is met
- Generates graphs by a sequence of decisions (whether to add a node, which node to add, and whether to add and edge, and which node to connect to the new node). Use a RecGNN to update the growing graph
- Can generalise to any graph/any domain

GraphRNN

- Uses a graph-level RNN & an edge-level RNN to generate nodes & edges; graph-level RNN adds a new node to a node sequence at each time step, while edge-level RNN produces a binary sequence indicating where connections should occur between the new node and the nodes previously generated in the sequence

2. Global Approaches:

Graph Variational Autoencoder (GraphVAE)

- Uses ConvGNN as encoder and simple MLP as decoder and outputs whole graph with adjacency matrix & node & edge attributes

Regularised Graph Variational Autoencoder (VGVAE)

- GraphVAE has difficulty controlling connectivity, validity, and node compatibility of its output graphs. VGVAE gives more control by imposing validity constraints to regularise the output of the decoder

Molecular Generative Adversarial Network (MolGAN)

- Integrates ConvGNNs, GANs, & reinforcement learning objects to generate graphs with desired properties
- Generator tries to generate a fake graph with its feature matrix, & discriminator tries to distinguish fake graph from empirical/real graphs. Additionally uses rewards to encourage graphs generated by discriminator to have certain properties defined by some external evaluator

NetGAN

- Trains discriminator to generate plausible random walks through an LSTM network and uses discriminator to try tell difference between fake random walks & real ones.
- After training, outputs a generated random walk which is used to generate a matrix of nodes

13.8. Spatial-Temporal GNNs

13.8.1. Overview

- STGNNs are graphs which are dynamic i.e. whose structures/characteristics change in time
- Use for forecasting future node labels
- 2 approaches to STGNNs:
  1. **RNN-based approaches**: Generate representations for spatial-temporal dependencies in graphs using a recurrent unit. Usually use graph convolutions to filter the input & the hidden states given to the recurrent unit.

     Suffer from time-consuming iterative propagation & gradient explosion/vanishing issues
  2. **CNN-based approaches**: Generate spatial-temporal dependency representations in non-recursive manner. Are easy to parallelise, have stable gradients, and have low memory requirements, therefore often advantageous over RNN-based approaches. Do this by interleaving 1D-CNN layers with graph convolution layers, where 1D-CNN layers learn temporal dependencies and graph convolution layers learn spatial dependencies of spatial-temporal graphs

13.8.2. RNN-Based STGNN Case Studies

Gaph Convolutional Recurrent Network (GCRN)

- Combines LSTM network with ChebNet

Diffusion Convolutional Recurrent Neural Network (DCRNN)

Integrates diffusion graph convolutional layer into a GRU network, & uses encoder-decoder framework to predict node values K steps into future

Structural-RNN

- Made up of a node-RNN & an edge-RNN. Edge-RNN generates temporal representation of graph, which is taken as input into the node-RNN, which then combines this representation with a spatial representation of the graph

GaAN

- Sometimes want to learn dynamic spatial dependencies between nodes. GaAN does this by using an attention mechanism to learn dynamic spatial dependencies with an RNN-based approach. Attention function is used to update edge weights between 2 nodes
- Have to calculate spatial dependency weight between every pair of nodes, which is not scalable to large graphs

13.8.3. CNN-Based STGNN Case Studies

Graph WaveNet

- Learns a self-adaptive adjacency matrix to perform graph convolutions, and can perform well without being given the adjacency matrix

13.9. Applications of GNNs

- Since GNNs by definition can be applied to a large range of application areas, often use certain evaluation datasets to see how well different GNN architectures perform on certain task. 2 common GNN tasks are:
    1. **Node classification**: Each node v in set of graph nodes V has an associated label y_v. Want to use GNN to learn a representation vector h_v for each node v such that its label can be predicted with $y_v = f(h_v)$

2. **Graph classification**: For a set of graphs $\{G_1, ..., G_N\}$ and their corresponding labels $\{y_1, ..., y_N\}$, learn a representation vector h_G that predicts the label of an entire graph, $y_G = g(h_G)$

Common open-access datasets to test node classification on include Core, Citeseer, Pubmed, PPI, and Reddit:

TABLE VI: Summary of selected benchmark data sets.

Category	Data set	Source	# Graphs	# Nodes(Avg.)	# Edges (Avg.)	#Features	# Classes	Citation
Citation Networks	Cora	[117]	1	2708	5429	1433	7	[22], [23], [25], [41], [43], [44], [45] [49], [50], [51], [53], [56], [61], [62]
	Citeseer	[117]	1	3327	4732	3703	6	[22], [41], [43], [45], [50], [51], [53] [56], [61], [62]
	Pubmed	[117]	1	19717	44338	500	3	[18], [22], [25], [41], [43], [44], [45] [49], [51], [53], [55], [56], [61], [62] [70], [95]
	DBLP (v11)	[118]	1	4107340	36624464	-	-	[64], [70], [99]
Bio-chemical Graphs	PPI	[119]	24	56944	818716	50	121	[18], [42], [43], [48], [45], [50], [55] [56], [58], [64]
	NCI-1	[120]	4110	29.87	32.30	37	2	[25], [26], [46], [52], [57], [96], [98]
	MUTAG	[121]	188	17.93	19.79	7	2	[25], [26], [46], [52], [57], [96]
	D&D	[122]	1178	284.31	715.65	82	2	[26], [46], [52], [54], [96], [98]
	PROTEIN	[123]	1113	39.06	72.81	4	2	[26], [46], [52], [54], [57]
	PTC	[124]	344	25.5	-	19	2	[25], [26], [46], [52], [57]
	QM9	[125]	133885	-	-	-	-	[27], [69]
	Alchemy	[126]	119487	-	-	-	-	-
Social Networks	Reddit	[42]	1	232965	11606919	602	41	[42], [48], [49], [50], [51], [56]
	BlogCatalog	[127]	1	10312	333983	-	39	[18], [55], [60], [64]
Others	MNIST	[128]	70000	784	-	1	10	[19], [23], [21], [44], [96]
	METR-LA	[129]	1	207	1515	2	-	[48], [72], [76]
	Nell	[130]	1	65755	266144	61278	210	[22], [41], [50]

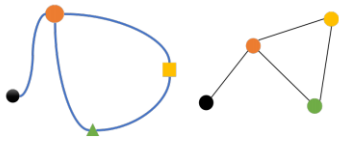
Also, open source libraries for GNNs exist. E.g. Deep Graph Library (DGL) allows you to rapidly implement many different types of GNN on top of deep learning platforms such as PyTorch. In DGL release paper (2019) (https://rlgm.github.io/papers/49.pdf), compares itself to other graph libraries including DeepMind's GraphNet, NGra, Euler and Pytorch Geometric. DGL turns out to be more flexible than these libraries, which try to maximise performance for specific models and are therefore not future-proof. DGL API was modelled on NetworkX, and it is therefore very similar to use. Can even create networkX graphs and convert to DGL graphs easily (https://docs.dgl.ai/tutorials/basics/3_pagerank.html) Is 'framework agnostic', therefore should work on top of any ML platform e.g. PyTorch, TensorFlow etc

13.9.1. Structural Scenarios

- Encapsulates scenarios where data is naturally graph structured. Examples include social network prediction, traffic prediction, recommender systems etc.

Physics

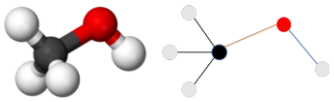
- Try to model real-World physical systems, with objects as nodes & relations between objects as edges. E.g. various objects & their interaction represented as a graph:



(a) physics

Chemistry & Biology

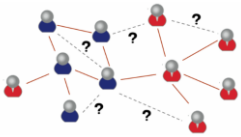
- Model molecules as graphs & learn to e.g. calculate **molecular fingerprints** (feature vectors representing the molecules) for e.g. classifying molecules into toxic & non-toxic, or **predict protein interfaces** for e.g. drug discovery & design



(b) molecule

Social Networks

- Use social network graph to e.g. predict connections between nodes (people)



Traffic

- Use for forecasting e.g. traffic speed, volume, or density on traffic networks or taxi demand using STGNNs. Usually consider sensors in traffic network (e.g. speed monitor, car counter etc) as nodes and the distance between the sensors as edges, where dynamic input feature to node is e.g. average traffic speed across a given period of time

Recommender Systems

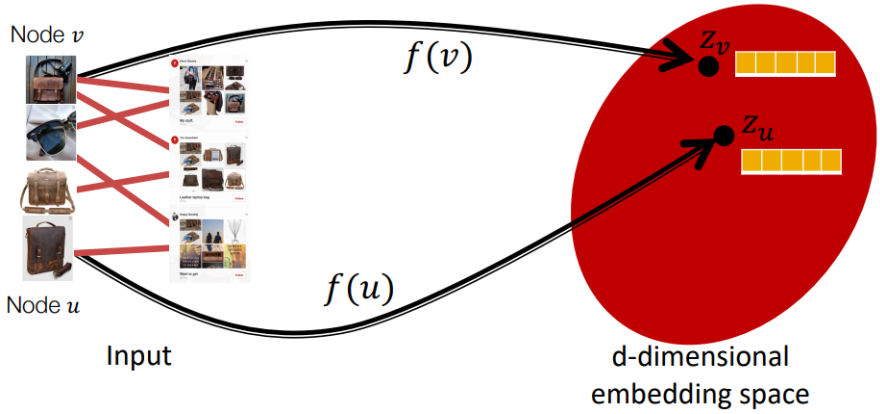
- E.g. take e.g. movies (items) and people as nodes and do **link predictions** for item-item, people-people, and people-item nodes to assign & predict 'scores' of how important different items are to different people, therefore can recommend items to people
- E.g. **Pinterest**: Has 300 million users uploading ideas ('pins') to various categories ('boards'). There are 4 billion pins and 2 billions boards. Pins might be e.g. an image, some text, a link etc. Boards will be a collection of these ideas/pins (i.e. pins on same board all have something in common).

I.e. pins and boards form nodes connected by edges. Is a massive **heterogeneous graph**! Want to train models so can recommend pins to users, classify pins and boards, cluster, do ranking etc. to help with e.g. presenting related pins, answering search queries, making shopping suggestions, advertising etc.

But if e.g. apply standard MLP where learn on image pins, might e.g. classify bed with railing and a normal railing into same category:

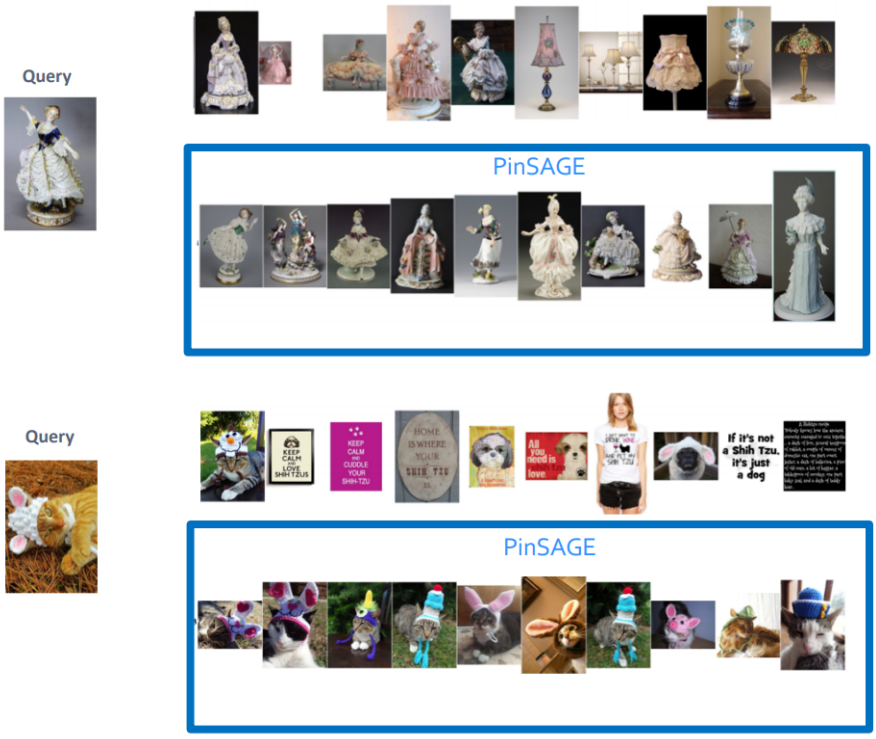
If had considered graph structure, might have seen that bed and railing are not connected/similar in graph, therefore may not have made this error. Can we leverage Pinterest database as a graph structure to do e.g. better classification, better pin recommendation etc.? Can we do this node embedding on a graph with billions of nodes & edges??

- Pinterest data forms a bipartite graph where have pin nodes (e.g. images) in one set and board nodes (collections) in another connected to one another. We can learn to map these onto d-dimensional space just as we've seen before:



- Key innovations used by PinSage to learn over graph with billions of nodes:
  - **Sub-sample neighbourhoods** with random walks (see prev GraphSage case study) for efficient GPU batching
  - Use **'producer-consumer'** CPU-GPU training pipeline
  - Use **curriculum learning** for negative samples
  - Use **MapReduce** for efficient inference
- Example of how a visual recommender only leverages visual information for classification to recommend people similar results to a given search query) compares to a graph

recommender (PinSage leverages graph info of Pinterest database to enrich its embedding representations):

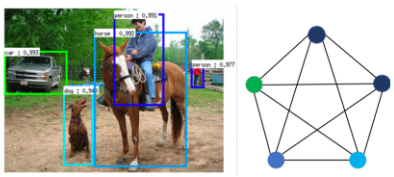


13.9.2. Non-Structural Scenarios

- Encapsulates scenarios not naturally represented as graphs (e.g. images, text, programming source code, multi-agent systems etc.). Usually use GNNs to tackle these scenarios by either creating a graph representation of e.g. a sentence and then applying GNN

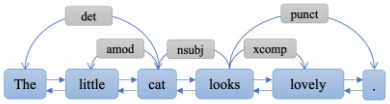
Computer Vision (Images)

- Use graphs to do e.g. **image classification**, **visual reasoning** to combine semantic (relevant/meaningful) info with spatial info observed for e.g. self-driving cars or human action/gesture recognition, **sematic segmentation** where assign category/label to each pixel in an image (a.k.a. dense classification) for e.g. 3D image classification



Natural Language Processing (Text)

- Can do e.g. **text classification** (learn embeddings of sentences or whole documents), **sequence labelling** (word-level tasks), **neural machine translation** (seq2seq tasks), **relation extraction** (extract semantic relations between entities in text), **event extraction** (recognise certain events occurring in text) etc. Works with GNNs because although text is made up of data with a sequential order, also contains an internal graph structure



13.9.3. Other

Generative Models

- E.g. generating new drugs, modelling social interactions etc

Combinatorial Optimisation

- Tackle NP-hard problems with GNNs e.g. traveling salesman problem (TSP), minimum spanning trees (MST) etc.

13.10. Open Problems

13.10.1. Model Depth

- Traditional NNs benefit from adding depth to NN to get greater expressive power capabilities (i.e. more linear regions etc). However, GNNs are usually no more than 3 layers (i.e. are relatively shallow) since stacking too many e.g. GCN layers results in too much smoothing (i.e. nodes converge to the same value). Designing deep GNNs to get more expressive capability is an exciting area of future research.

13.10.2. Dynamic Graphs

- How to model dynamic graphs with e.g. changing structures (nodes/edges disappearing/reappearing) &/or node attributes. Handling dynamic graphs would be big step forward in developing stable and adaptable generalised GNNs. Need e.g. STGNNs to be able to perform graph convolutions w.r.t. *dynamic* spatial relations.

13.10.3. Non-Structural Scenarios

- Currently not optimal method to turn raw non-graph-structured data (e.g. images) into graphs. New graph generation approaches might enable GNNs to tackle an even wider

range of fields

13.10.4. Scalability

- How to create embedding representations of e.g. social networks or recommendation systems with millions of nodes. Difficult since there are many core computational steps required for GNNs, which don't scale to processing very large graphs

13.10.5. Heterogeneity

- Most current GNNs can only handle homogeneous graphs i.e. where all nodes & edges in graph have same type. Need GNNs that can handle heterogeneous graphs with e.g. images & text as different types of node

13.11. GNN Implementation Notes

- Important to pre-process data. Should use:
  - Re-normalisation tricks
  - Variance-scaled initialisation
  - Network data whitening
- Use e.g. Adam optimiser to take care of decaying learning rate when optimising NN
- Relu activation function tends to work well
- Don't put an activation function in output layer
- Every layer should include a bias term
- Using GCN layer with e.g. 64 or 128 units is already plenty
- Should always be able to overfit a small batch of training data to get close to 100% accuracy/0 error (if don't, know something is wrong)

Part IV.

Combinatorial Optimisation