15-887 Planning, Execution, and Learning
Deep Reinforcement Learning

Devin Schwab

November 3, 2016

Outline

Binary Perceptron

- The simplest unit in a neural network is a perceptron
- perceptrons are made up of:
  - a set of inputs, X
  - a weight for each input, w_i
  - a threshold, b
  - an activation function, $\begin{cases} 1 & \text{if } \sum x_i w_i + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$

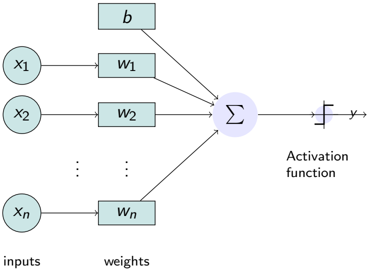


Figure: Binary Threshold Perceptron

Perceptron Decision Surface

Despite having a non-linear activation,
the decision surface is still a hyper plane.

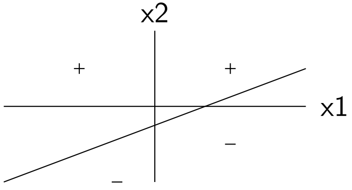


Figure: Example perceptron decision boundary

Limits of Perceptrons

- The decision boundary of a single perceptron is still linear, so the data must be linearly separable
- Even simple functions like XOR cannot be learned

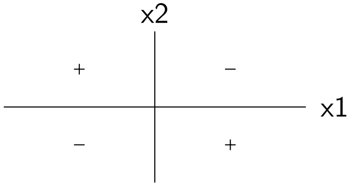


Figure: XOR data

Limits of Perceptrons

- The decision boundary of a single perceptron is still linear, so the data must be linearly separable
- Even simple functions like XOR cannot be learned

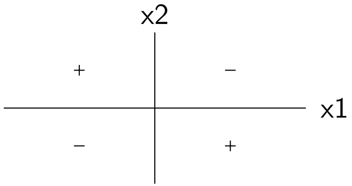


Figure: XOR data

How do we solve this?

Limits of Perceptrons

- The decision boundary of a single perceptron is still linear, so the data must be linearly separable
- Even simple functions like XOR cannot be learned

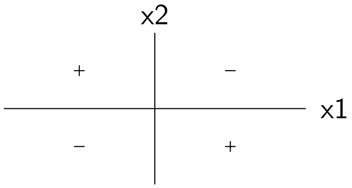


Figure: XOR data

How do we solve this?

Use multiple perceptrons

Multilayer Perceptrons (MLPs)

- A single perceptron has a linear decision boundary
- The composition of non-linear functions leads to non-linear decision boundaries
- So stack layers of perceptrons to get a non-linear decision boundary.

Neural Network Architecture

- Networks are constructed from multiple layers of perceptrons
- Networks have an input layer, an output layer and one or more hidden layers
- This architecture is very general
  - Connectivity between layers can vary
  - Activation functions in each layer can vary
  - Number of units in each layer can vary

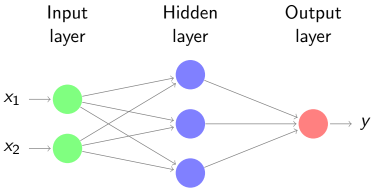
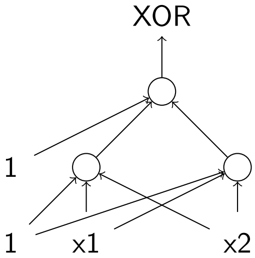


Figure: Multilayer Perceptron with a single hidden layer. This particular network is fully connected.

XOR Network



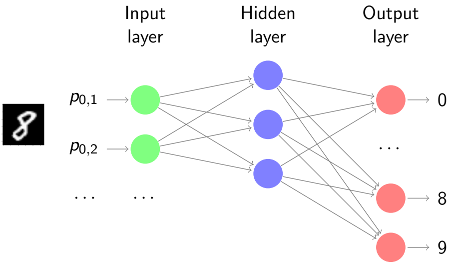
What makes it a deep network?

- Deep Learning really just means a neural network with more than one hidden layer
  - Nowadays, many, many more hidden layers
  - More layers, and more neurons means more representation power
- Why have deep neural networks only recently become popular?
  - Better optimization techniques
  - Better regularization
  - Better computing power (i.e. GPUs)

Example Network - MNIST Dataset

- Input: 28x28 black and white images of digits

- Output: The digit shown in the image



What is needed for training?

- Training data $(X^{(i)}, t^{(i)})$
  - $X^{(i)}$ are the data dimensions
  - $t^{(i)}$ is a target value
- A loss function. This a positive, real-valued function where the bigger the number of the bigger the error in the classification label
  - Mean Squared Error (MSE) works: $L = \frac{1}{N2}\sum_i \left(t^{(i)} - y(x^{(i)})\right)^2$
  - $y(x^{(i)})$ is the output of your neural network for input $x^{(i)}$
- A method for optimizing a continuous, non-convex function (usually a gradient descent variant)

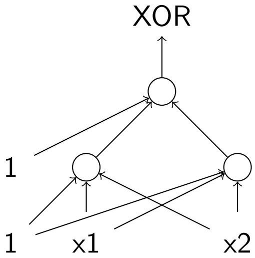
Gradient Descent

- To update the weights and the bias terms use the gradients
  - $w_i \leftarrow w_i - \alpha \frac{\partial L}{\partial w_i}$
  - $b_i \leftarrow b_i - \alpha \frac{\partial L}{\partial b_i}$
  - α is a learning rate, and L is the loss function.
- How do you actually compute these gradients? **Chain Rule!**
  - $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \cdots \frac{\partial z_i}{\partial w_i}$
  - where z_k is the output of the k-th layer

Backpropagation

- To compute the partials with respect to each layer's parameters, the partial derivatives of the layers all the way to the output are needed
  - i.e. all of the $\frac{\partial z_k}{\partial z_{k-1}}$ terms for k greater than the current layer)
- It would be very expensive to compute these partial gradients every single time for every single parameter
  - So keep the partial derivatives as you go back through the layers.

XOR Network

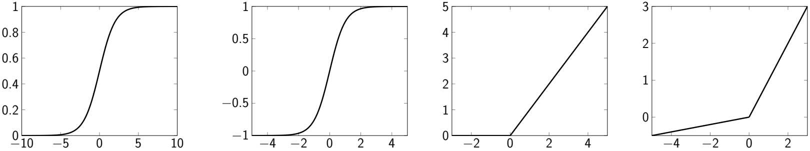


Activation Functions

- The binary perceptron is not usually used in neural nets
- The only constraint on activation functions is that they must be non-linear
- However, there are some desirable properties:
  - Large gradients
  - Differentiable (at least over most of the domain)
  - Easy to compute gradients
  - Some activation functions have a probabilistic interpretation
  - Sometimes, having a fixed output range is desirable

Common Activation Functions

- Sigmoid: $(1 + e^{-z})^{-1}$
- Hyperbolic Tangent: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Rectified Linear Units (ReLu): $\mathrm{relu}(z) = \max\{z, 0\}$
- Leaky ReLu: $\begin{cases} z & z > 0 \\ \epsilon z & \text{otherwise} \end{cases}$, where $\epsilon \ll 1$



Sigmoid **tanh** **ReLu** **Leaky ReLu**

Summary

- Stack lots of perceptrons in layers to make a neural network
- Use backpropagation to train the weights in the network
- Network learns a non-linear function approximation

Basic Definitions

- The world is modeled as a Markov Decision Processes (MDPs).
  Defined as a tuple: (S, A, P, R)
  - S: Set of all states
  - A: Set of all actions
  - P: Transition Function. $P : S \times A \times S \rightarrow [0, 1]$
  - R: Reward Function. $R : S \times A \rightarrow \mathbb{R}$
- Policy: $\pi : S \rightarrow A$. Maps states to actions.

Q-Learning

- We don't know the model so learn it!
- After each action, use the observed (s, a, r, s') to update the estimate of the Q-function
  - $\hat{Q}^*(s, a) \leftarrow (1 - \alpha)\hat{Q}^*(s, a) + \alpha \left(r + \gamma \max_{a' \in A} \hat{Q}^*(s', a') \right)$
- What if:
  - There are an infinite number of states, or the states are continuous?
  - Many states are all similar and should have similar actions?

Q-Learning

- We don't know the model so learn it!
- After each action, use the observed (s, a, r, s') to update the estimate of the Q-function
  - $\hat{Q}^*(s, a) \leftarrow (1 - \alpha)\hat{Q}^*(s, a) + \alpha \left(r + \gamma \max_{a' \in A} \hat{Q}^*(s', a') \right)$
- What if:
  - There are an infinite number of states, or the states are continuous?
  - Many states are all similar and should have similar actions?

Create a function that maps $(s, a) \rightarrow Q(s, a)$

Neural Networks as Function Approximators

Idea

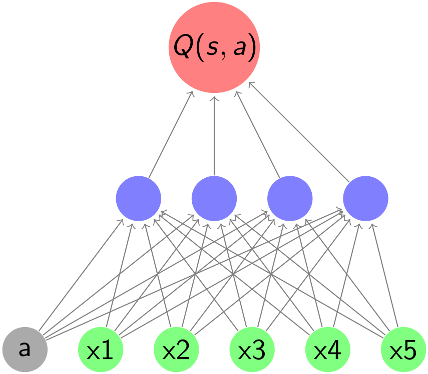
Approximate the Q-function with a neural network

Deep Q-Network DQN

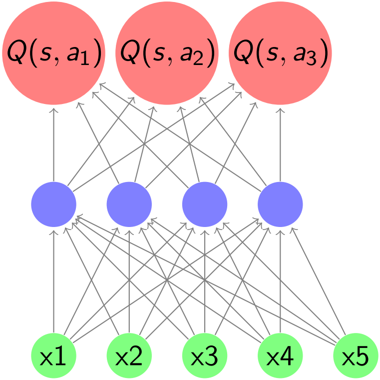
- **Basic idea:** Approximate with a deep neural network[1]
- Neural Nets had been tried before, with some success
- Major contributions
  - Network architecture that requires one pass for each state
  - The loss function used with the network
  - The use of replay memory
  - The use of a target network

[1] Mnih et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529533, 2015. URL: http://dx.doi.org/10.1038/nature14236, arXiv:1312.5602, doi:10.1038/nature14236

Naïve Network Architecture



Network Architecture



DeepQ Loss Function

- In order to perform back-propagation on the DeepQ network, a loss function $L(\theta)$ is needed.
- Use the mean-squared error of the Bellman Equation for Q-functions
  - $L_i(\theta_i) = \mathbb{E}\left[\left(y - \hat{Q}(s, a; \theta)\right)^2\right]$
  - Where the target value y is $y = r + \gamma \max_{a'} Q^*(s', a')$

DeepQ Loss Function

- In order to perform back-propagation on the DeepQ network, a loss function $L(\theta)$ is needed.
- Use the mean-squared error of the Bellman Equation for Q-functions
  - $L_i(\theta_i) = \mathbb{E}\left[\left(y - \hat{Q}(s, a; \theta)\right)^2\right]$
  - Where the target value y is $y = r + \gamma \max_{a'} Q^*(s', a')$
- Unfortunately, $Q^*(s', a')$ is not known apriori, so y cannot be computed

DeepQ Loss Function

- In order to perform back-propagation on the DeepQ network, a loss function $L(\theta)$ is needed.
- Use the mean-squared error of the Bellman Equation for Q-functions
  - $L_i(\theta_i) = \mathbb{E}\left[\left(y - \hat{Q}(s, a; \theta)\right)^2\right]$
  - Where the target value y is $y = r + \gamma \max_{a'} Q^*(s', a')$
- Unfortunately, $Q^*(s', a')$ is not known apriori, so y cannot be computed

How do we solve this?

DeepQ Loss Function

- In order to perform back-propagation on the DeepQ network, a loss function $L(\theta)$ is needed.
- Use the mean-squared error of the Bellman Equation for Q-functions
  - $L_i(\theta_i) = \mathbb{E}\left[\left(y - \hat{Q}(s, a; \theta)\right)^2\right]$
  - Where the target value y is $y = r + \gamma \max_{a'} Q^*(s', a')$
- Unfortunately, $Q^*(s', a')$ is not known apriori, so y cannot be computed

How do we solve this?

Use a target network

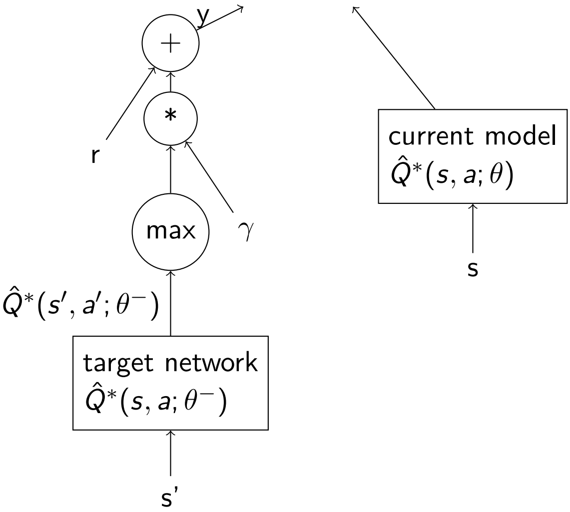
Target Network

- Approximate $Q^*(s', a')$ with your previous estimate of $Q^*(s', a')$
  - i.e. $\hat{Q}^*(s', a'; \theta^-)$
- Now you have two models
  - Your original model with weights θ
  - A copy of this model with a previous iterations weights θ^-
- The copy is known as the target network.

Target Network

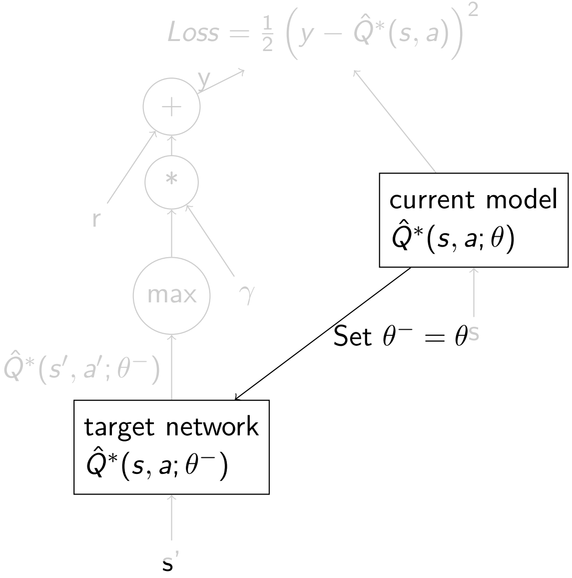
Starting in state s, took action a, received reward r and ended in state s'

$$Loss = \frac{1}{2}\left(y - \hat{Q}^*(s,a)\right)^2$$



Target Network

After an update, copy the weights from the model to the target network



Hard Target Updates

- Updating the target network weights every time you update your model weights can have some stability issues
- Instead, only perform the copying of the weights every k steps

Soft Target Updates

- More recent work has shown that soft target updates work better
- Take a small step from your current target weights towards the current Q-network weights.

$$\theta^- \leftarrow (1-\tau)\theta^- + \tau\theta, \text{ where } \tau \ll 1$$

DeepQ Loss Function

- Now we can calculate our loss at each step
- To backpropagate, just take the gradient of the loss with respect to the network parameters:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

Replay Memory

- Instead of directly using the experience samples $e_t = (s, a, s', r)$ put each sample in a ring buffer of size N
- Now to update the θ parameters, select k samples uniformly from this buffer and treat this as a training mini-batch
- Three major benefits
  1. **Data efficiency**: Each sample is used multiple times
  2. **Reduced correlation between updates**: Most sequential samples have a lot of redundant information, which can lead the agent to over fitting
  3. **Reduces feedback loops**: on-policy samples are chosen according to current parameters, which can lead lots of similar samples, which can get the agent stuck in local minima or divergence of $\hat{Q}$.

Sequences of States

- Many domains are not fully Markovian or fully observable
  - For example, in an Atari game, taking a picture of a single screen means the agent cannot tell which directions the sprites are moving
- So instead of giving a single images, store the previous frames in a sequence and give all frames at once to the agent
  - Now the agent can learn the concept of velocity by looking at how the pictures have changed between frames

Preprocessor

- Many times the raw state is not in a convenient form for learning
- The DQN algorithm uses ϕ to represent a preprocessor
- The preprocessor is run on every state, and is fixed
  - i.e. the output of the preprocessor will always be the same for the same input
- Useful for operations like:
  - Converting images to gray scale
  - Down-scaling images
  - etc.

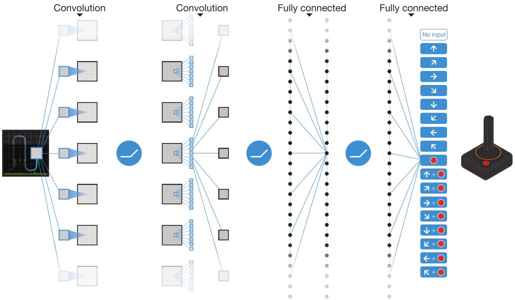
DQN Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation
  end for
end for

DQN Atari

- Input is raw pixel values from Atari
- Reward is just the score received for an action
- Outputs represent the estimated Q-function value for the given input state and the action associated with that neuron.



Atari Games

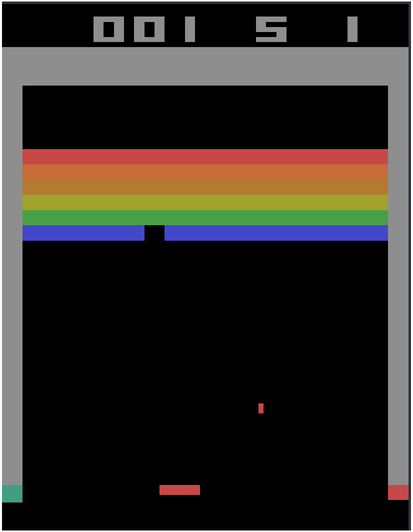


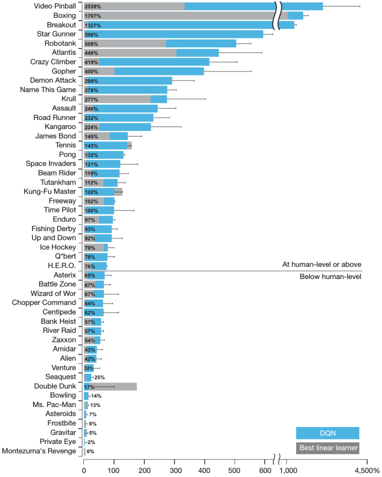
Figure: Atari Breakout

Figure: Atari River Raid

DQN Results

- On 29 of tested games, the agent achieved better performance than human-players
  - The games varied wildly in genre (e.g. side scrolling shooter vs boxing)
  - The same architecture, and hyperparameters were used across all networks in these experiments
- In some games (like Breakout), the agent was able to learn long-term expert level strategy

DQN Results

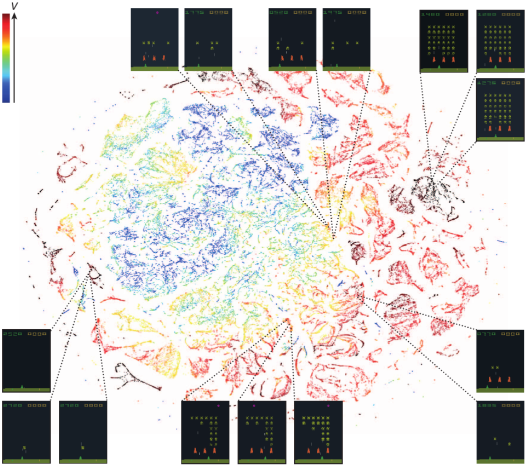


Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*bert 78%
H.E.R.O. 76%
At human-level or above
Below human-level
Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pac-Man 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,500%

Breakout Demo

https://www.youtube.com/watch?v=V1eYniJ0Rnk

DQN



Summary

- Deep Learning can learn from data to represent complex functions
- DQN is the basis for most current DeepRL algorithms
- DQN is incredibly versatile
  - It learns a wide variety of Atari games with no domain knowledge
- Tons of open research questions

Backup

Universal Approximation Theorem

- What types of functions can this framework learn?
- Informally: a neural network can learn any multidimensional, continuous function with a single hidden layer, given the hidden layer contains enough units
- Formally proved by the **Universal Approximation Theorem**
- This theorem tells what types of functions neural networks can learn, but not how compact the network is, how easy the training is, or what activation functions are used.

Backpropagation Pseudocode

Algorithm 1 Backpropagation Pseudocode

1: **procedure** $\textsc{Backprop}(X, y, W, b)$
2: Compute $\hat{y}$ for the given input X ▷ Forward Pass
3: Compute the loss function
4: Compute the partial derivative of each layer's output with respect to its input (i.e. $\frac{\partial z_k}{\partial z_{k-1}}$) ▷ Start backwards pass
5: Compute the partial derivative of each layer's output with respect to its parameters (i.e. $\frac{\partial z_k}{\partial w_i}$ and $\frac{\partial z_k}{\partial b_i}$)
6: For each parameter compute the gradient $\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \cdots \frac{\partial z_i}{\partial b_i}$ using the previously computed partials
7: **return** Parameter gradients

Feed-forward Computational Costs

- Mathematically each neuron computes the function activation$(W^T X + b)$
- Computing $W^T X + b$ takes $|X|$ multiplications and additions
- The cost of the activation depends on the function used
  - Say it adds j multiplications and k additions
- For a layer with n neurons, that is densely connected there will be $n(|X| + j)$ multiplications and $n(|X| + k)$ additions
- This is the cost of a single layer, in deep networks there can be hundreds of layers

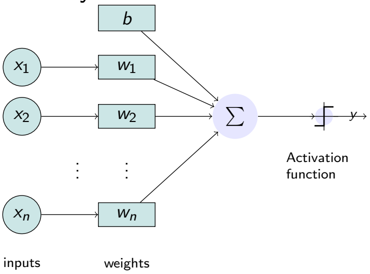


Figure: A single neuron

Cost Depends on Activation Function

- The computational costs can vary a lot based on the activation functions used
- Sigmoid and tanh activations contain exponentials in both the forward direction and in their gradients
- On the other hand rectified linear units add very little over head
  - Just a max operation in the forward direction
  - The gradient is either zero or one depending on the sign of the input

A Simple Example

- Consider a network with the following layers
  - A layer going from 8 inputs to 400 outputs with ReLu activation
  - A layer going from 400 units to 300 units with ReLu activation
  - A layer going from 300 units to 2 units with a tanh activation
- This *small* network has approximately 125,000 parameters
  - $125,000 \approx 8 \cdot 400 + 400 \cdot 300 + 300 \cdot 2$
- That means approximately 125,000 multiplications and additions plus the cost for computing the activation functions, every time this network is used

GPUs for Feed-Forward Networks

- With GPUs we can greatly parallelize this computation
- Each layer is run sequentially, but within a layer all of the neuron's can be computed in parallel
- If multiple items are being fed through the network in succession then the network can be treated as a pipeline with layers being run in parallel on different inputs

Loss Function

- Feed-forward neural networks map an input vector X to an output vector $\hat{y}$
- The job of the loss function is to quantify how close the output value $\hat{y}$ is to the true value y
- The job of the training algorithm is to adjust the weights in order to minimize the sum of the loss from all of the training examples. $\min_{W,b} \text{loss}(\hat{y}, y)$

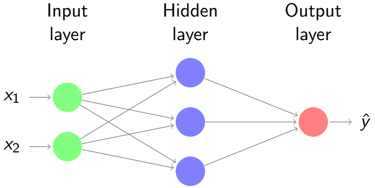


Figure: Example network

Common Loss Functions

- Mean Squared Error (MSE): $\sum_i^n \frac{1}{2n}||\hat{y}^{(i)} - y^{(i)}||^2$
  - Works well for regression problems
- Cross Entropy: $-\frac{1}{n}\sum_i^n [yln\hat{y} + (1-y)ln(1-\hat{y})]$
  - This is used with a sigmoid or soft-max activation
  - Works well for classification problems