10-703: Recitation 1

Fall 2023

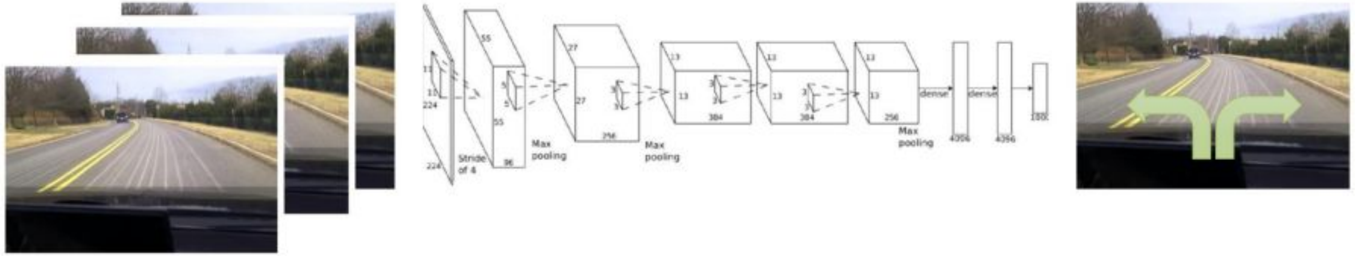
Athiya Deviyani & Amy Lin

Topics We Will Be Covering

- Neural Networks
  - MLPs, CNNs, RNNs, Transformers
- Tensorflow, Keras and Pytorch
- Open AI Gym
- Bandits

Why deep learning?

“Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level.” *– LeCun, Bengio, Hinton*



Scales well with large amounts of data

Less reliant on human feature engineering

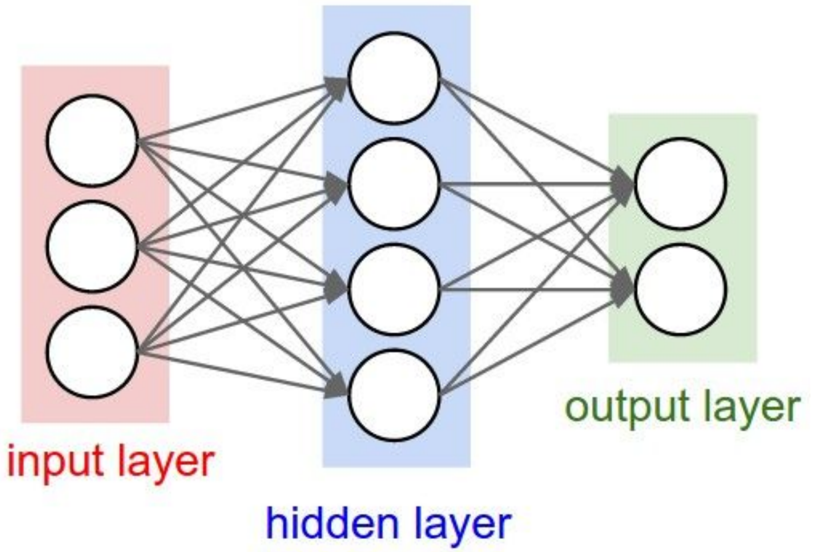
Source: 10-403 recitation slides, https://cmudeeprl.github.io/403website_s22/assets/lectures/s22/recitation_1-neural_nets.pdf

Why “deep” RL?

- Distinction between *classical* and *deep* RL
  - Traditionally, restricted to simpler function approximators (e.g. discrete lookup tables)
    - **Restricted to tabular methods**: problems in which the state and actions spaces are small enough for approximate value functions to be represented as arrays and tables
  - Modern deep learning methods can handle much more complex functions
- Function approximation is super useful for RL
  - Represent agents, reward functions, dynamics models, etc. using deep networks
  - **Handle richer input modalities** like images and text; classical methods often rely on hand-crafted low-dimensional features
- The use of deep learning is most useful in problems with high-dimensional state space
  - This means that with deep learning, Reinforcement Learning is able to solve more complicated tasks with lower prior knowledge because of its ability to learn different levels of abstractions from data

Multi-layer perceptrons (MLPs)

- A universal function approximator
  - Very general-purpose
  - Limited inductive bias
- Stack of *feedforward* layers
  - Input layer, hidden layer(s), output layer
  - **Nonlinear** activation function
- Hierarchical representation

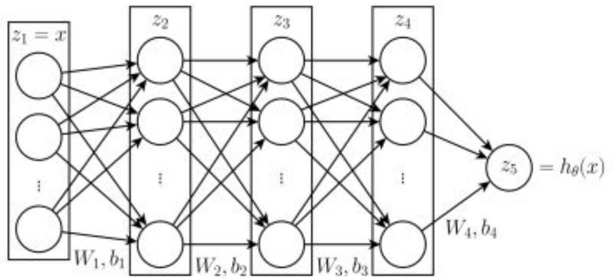


This is most of what you need for the homeworks (architecture-wise)!

Source: https://cs231n.github.io/neural-networks-1/

Multi-layer perceptrons (MLPs)

- Some notation
  - Input features: $x \in \mathbb{R}^d$
  - Outputs: $y \in \mathcal{Y}$
  - Network parameters: $\theta \in \mathbb{R}^k$
  - Network function: $h_\theta : \mathbb{R}^d \to \mathcal{Y}$
  - Activation function: $f \in \{\sigma, \tanh, \mathrm{relu}, \dots\}$
  - Loss function: $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+ \in \{\mathrm{mse}, \mathrm{nll}, \mathrm{bce}, \dots\}$



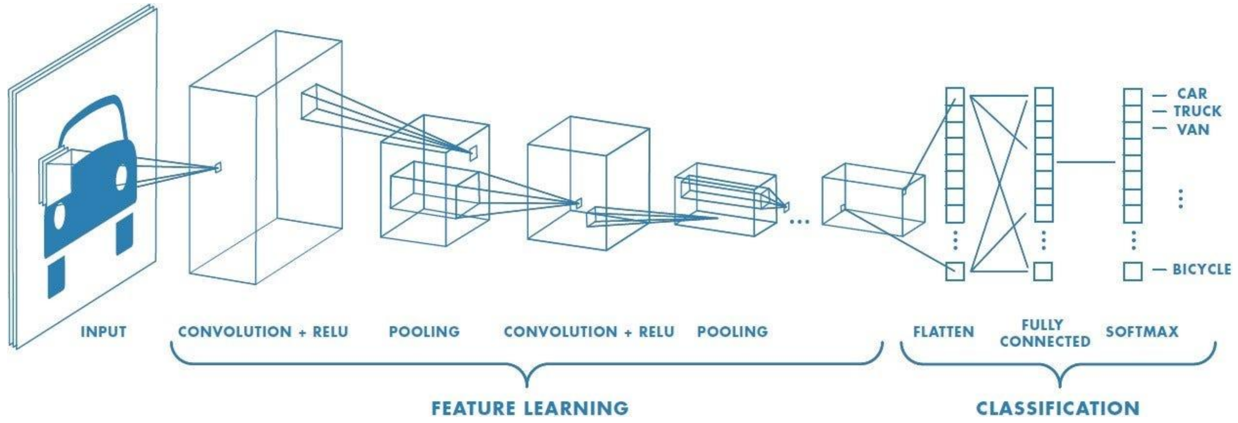
- Computing the network output

$$z_{i+1} = f_i(W_i z_i + b_i), \quad i = 1, \dots, k-1, \quad z_1 = x$$

$$h_\theta(x) = z_k$$

Source: 10-403 recitation slides, https://cmudeeprl.github.io/403website_s22/assets/lectures/s22/recitation_1-neural_nets.pdf

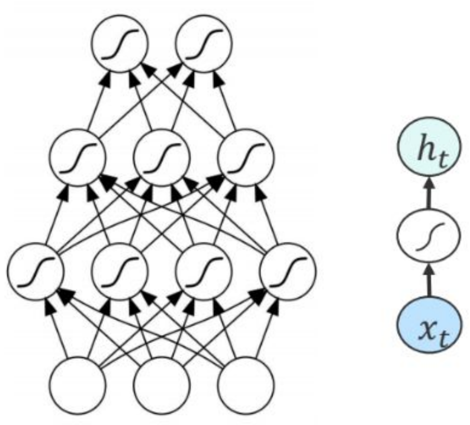
Convolutional Neural Networks



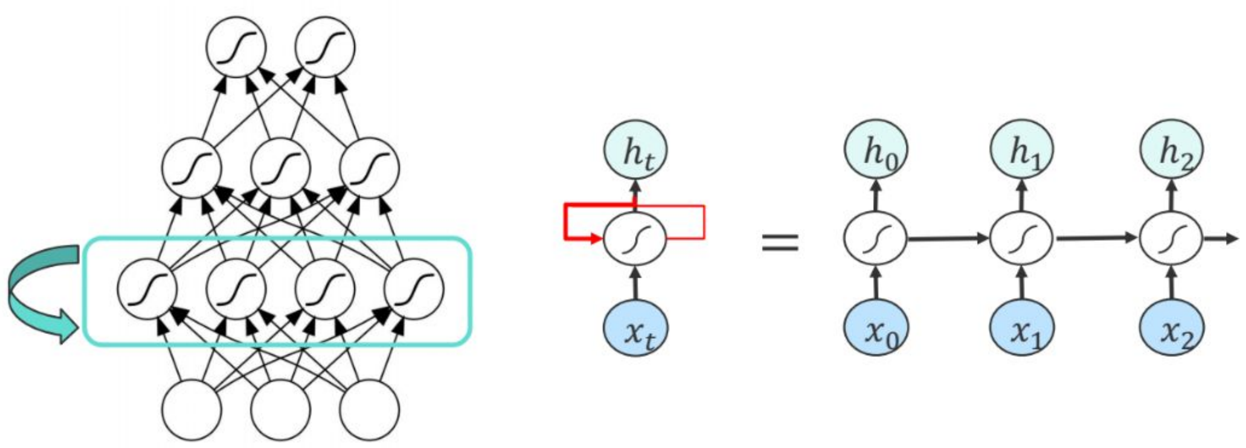
- A convolutional neural network can have several layers that **each learn to detect different features of an image**. Filters are applied to each training image at different resolutions, and the output of each convolved image is used as the input to the next layer. The filters can start as very simple features, such as brightness and edges, and increase in complexity to features that uniquely define the object.
- Why not flatten an image and pass it through an MLP?
  - CNNs are able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters
  - CNNs are able to fit the image dataset better due to the reduction in the number of parameters and weights reusability

Source: A Comprehensive Guide to Convolutional Neural Networks (https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/)

Recurrent neural networks (RNNs)



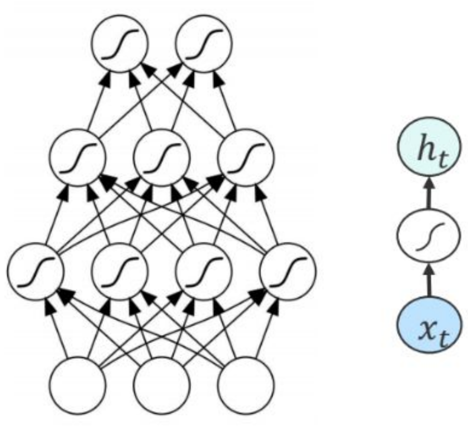
MLP: at each timestep, the computation is independent of previous timesteps (*stateless*)

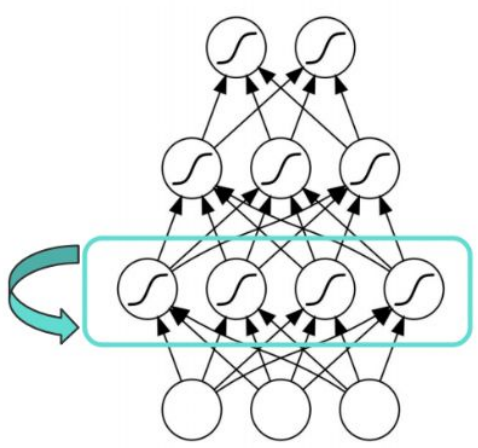


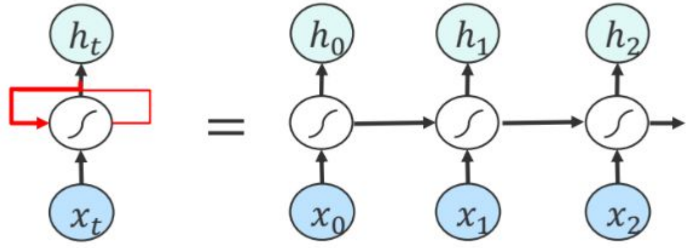
RNN: at each timestep, the computation is dependent on previous timesteps (*stateful*)

Source: Eric Xing, https://www.dropbox.com/s/uetn2j5opxjg0x8/Deep%20Sequence%20Models.pdf?dl=0

Recurrent neural networks (RNNs)







- Natural fit for sequential decision-making problems
- Vanishing / exploding gradient problem (alleviated by LSTMs / GRUs)
- Hard to parallelize due to sequential computation

Source: Eric Xing, https://www.dropbox.com/s/uetn2j5opxjg0x8/Deep%20Sequence%20Models.pdf?dl=0

Attention

Basic building block used in transformer models

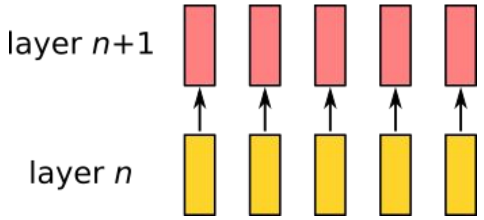
Can think of as a *differentiable dictionary lookup* that takes three arguments:

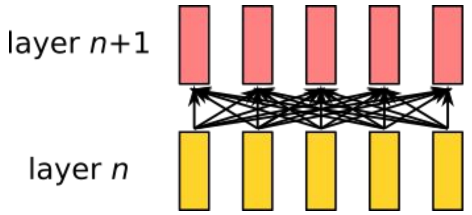
Query (**Q**), Key (**K**), Value (**V**)

Each argument (and output) is a sequence of vectors with same sequence length

Two main differences from standard feedforward layers:

1. Attention can selectively attend to / ignore specific sequence elements
   a. Standard feedforward layers can do this too, but the key difference is *sparsity*
2. Handles *arbitrary length* sequences of vectors





Dot-product attention

Inputs: a query q and a set of key-value (k-v) pairs to an output

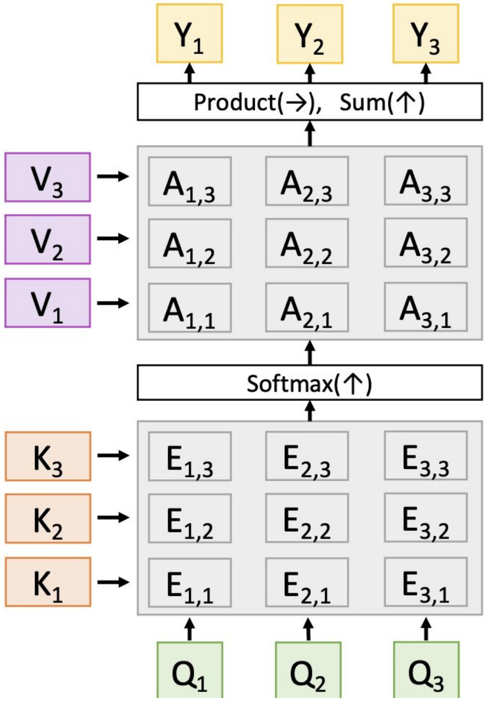
Query, keys, values, and output are all vectors

Output is weighted sum of values, where

Weight of each value is computed by an inner product of query and corresponding key

Queries and keys have same dimensionality d_k value have d_v

$$A(Q, K, V) = softmax(QK^T)V$$



Source: Richard Socher, https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf, UMich Deep Learning for Computer Vision https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/598_WI2022_lecture18.pdf

Scaled dot-product attention

Inputs: a query q and a set of key-value (k-v) pairs to an output

Query, keys, values, and output are all vectors

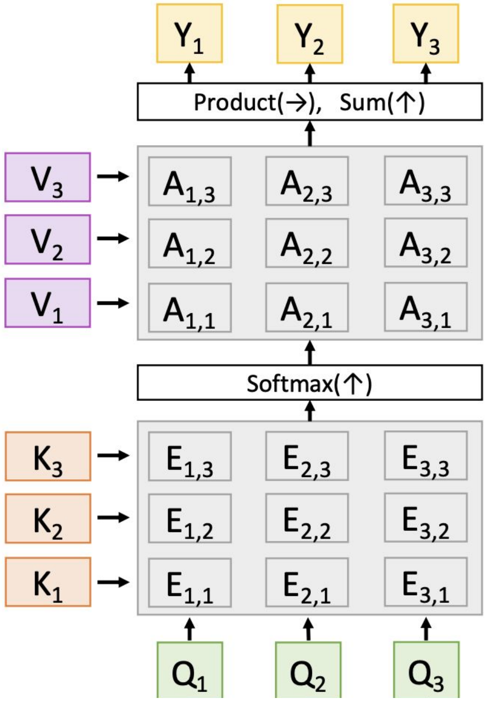
Output is weighted sum of values, where

Weight of each value is computed by an inner product of query and corresponding key

Queries and keys have same dimensionality d_k value have d_v

$$\cancel{A(Q,K,V) = softmax(QK^T)V}$$

$$A(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Source: Richard Socher, https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf, UMich Deep Learning for Computer Vision https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/598_WI2022_lecture18.pdf

Self-attention vs Cross-attention

- Where do Q, K, V come from?
  - Recall that Q is the query, K is the key, and V is the value (all are vectors)

- **Self-attention**: Q, K, V are all computed from the same input vector

- **Cross-attention**: Q comes from input vector v1, and (K,V) comes from another input vector v2 (we say v1 *cross-attends* to v2)

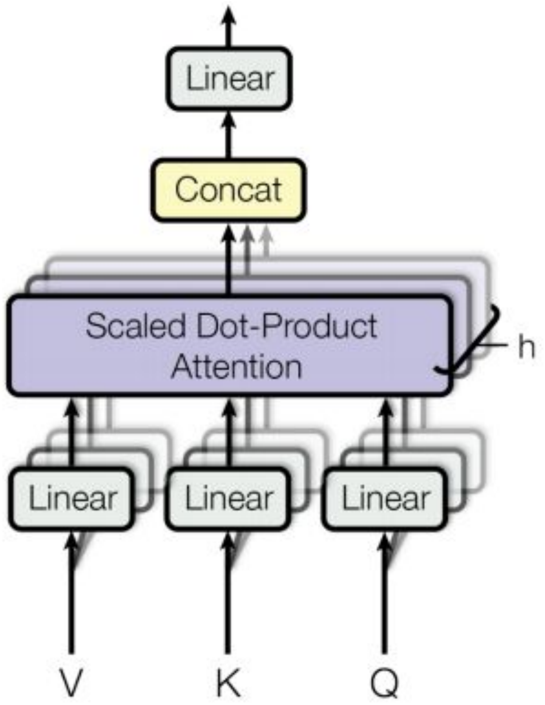
$$A(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Source: Richard Socher, https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf

Multi-headed attention

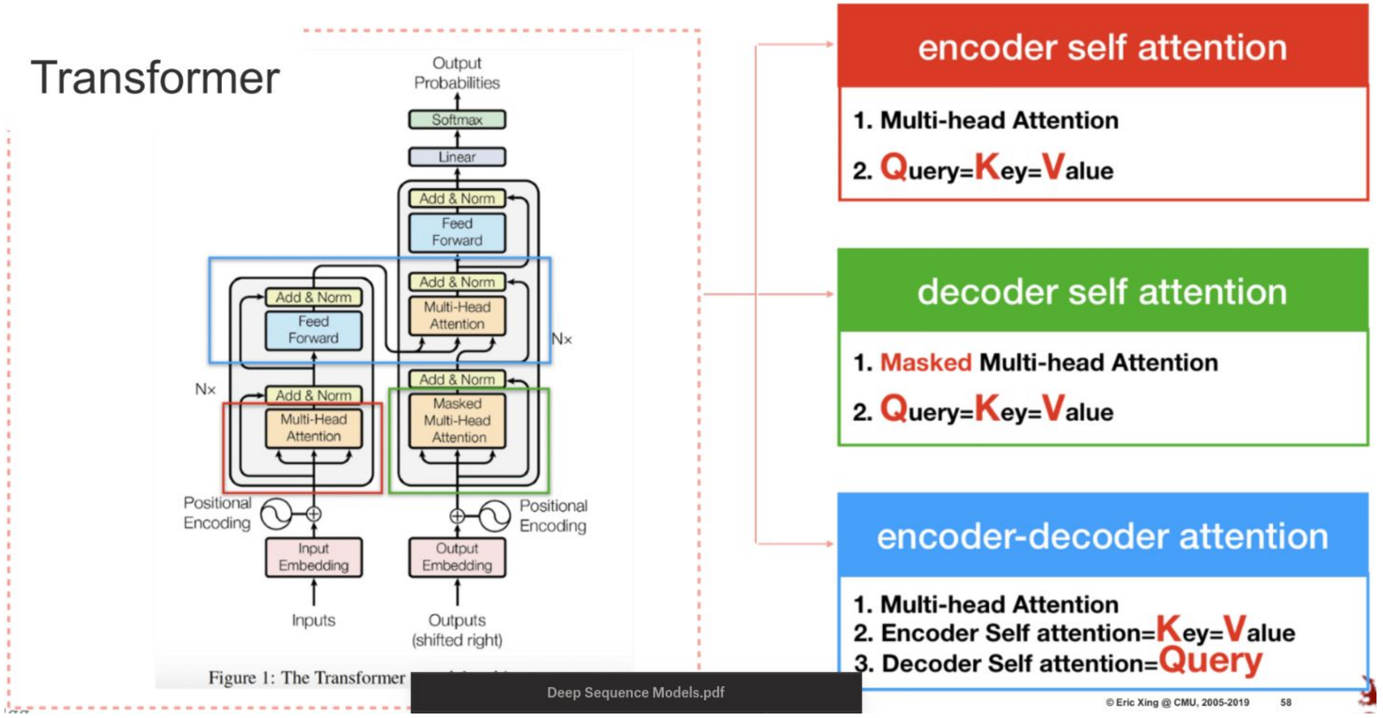
For an input vector, we can generate an arbitrary number of Q, K, V vectors

If we generate *h* sets of Q,K,V vectors, then we have multi-headed attention with *h* heads



Source: Richard Socher, https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf

Transformers



encoder self attention

1. Multi-head Attention
2. Query=Key=Value

decoder self attention

1. Masked Multi-head Attention
2. Query=Key=Value

encoder-decoder attention

1. Multi-head Attention
2. Encoder Self attention=Key=Value
3. Decoder Self attention=Query

© Eric Xing @ CMU, 2005-2019 58

Image Source: Eric Xing, Attention is all you need: https://arxiv.org/pdf/1706.03762.pdf

Transformers in RL

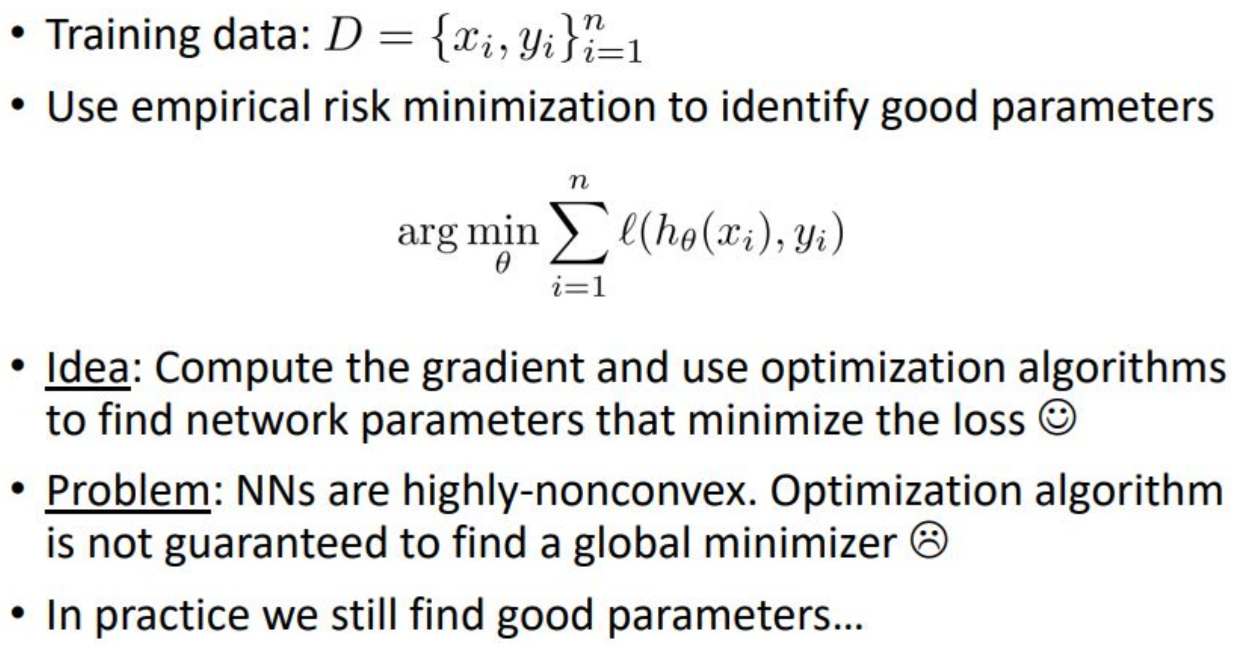
- Developed for natural language processing, but extremely general modeling capabilities
  - Good at capturing long-horizon sequences; useful for learning dynamics and imitation
  - Permutation invariance is useful for many real-world observation spaces
  - Learn to model trajectories and "condition on goodness"
    - See *Decision Transformer* and other related work

- Related to more recent trends in deep RL
  - Learning policies from large offline datasets (*offline RL*)
  - Handling multimodal tasks and data (e.g. *vision and language*)

- The bad news:
  - Hard to train, especially when doing RL
  - Really data / compute hungry

Source: Chen et al. '21, https://arxiv.org/pdf/2106.01345

Supervised learning



- Training data: $D = \{x_i, y_i\}_{i=1}^n$
- Use empirical risk minimization to identify good parameters

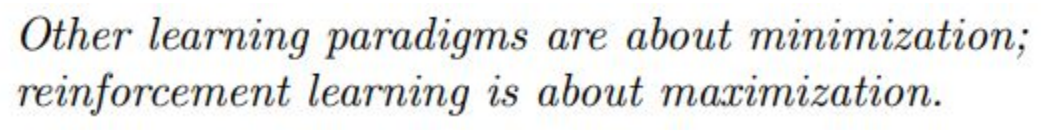
$$\arg\min_{\theta} \sum_{i=1}^{n} \ell(h_\theta(x_i), y_i)$$

- Idea: Compute the gradient and use optimization algorithms to find network parameters that minimize the loss ☺
- Problem: NNs are highly-nonconvex. Optimization algorithm is not guaranteed to find a global minimizer ☹
- In practice we still find good parameters…

Source: 10-403 recitation slides, https://cmudeeprl.github.io/403website_s22/assets/lectures/s22/recitation_1-neural_nets.pdf

Supervised learning vs Reinforcement learning

- Many deep RL methods still use supervised learning
  - e.g. training a dynamics model is just supervised learning
- Reinforcement learning (usually) involves *active learning* – the agent controls the data distribution it is trained on
- Supervised learning mimics demonstrated behaviors, reinforcement learning is *open-ended* discovery of optimal behaviors


Other learning paradigms are about minimization;
reinforcement learning is about maximization.

- In practice, these differences can severely complicate training and evaluation (e.g. fewer convergence guarantees, volatile gradients, sensitive to hyperparameters, etc.)

Source: Lu et al ‘22, https://arxiv.org/pdf/2103.04047

PyTorch and Tensorflow

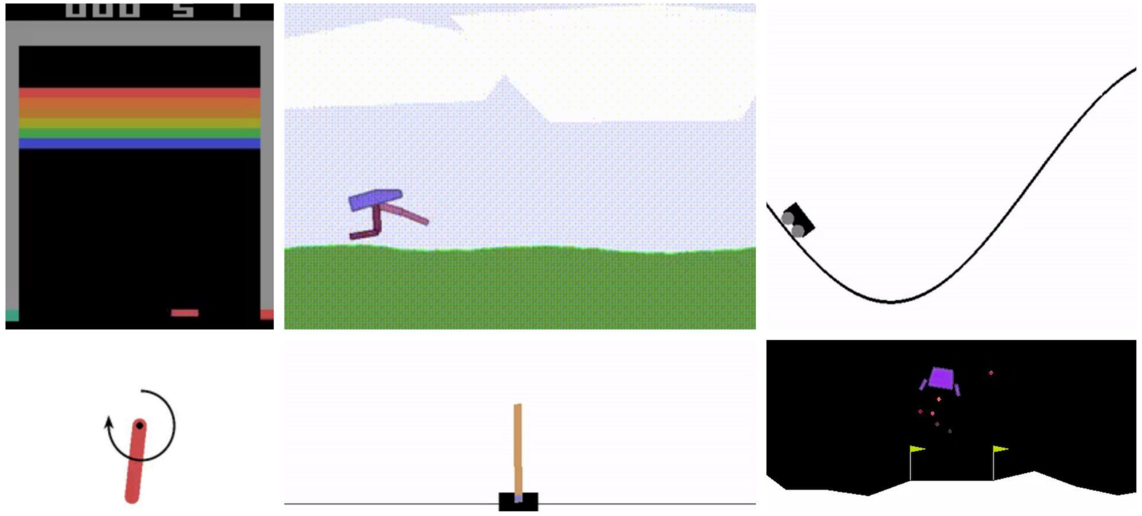
- All homeworks can be completed using either PyTorch or Tensorflow
- Read documentation and Tutorials
- PyTorch Installation: https://pytorch.org/get-started/locally/
- PyTorch Tutorials: https://pytorch.org/tutorials/
- Tensorflow Installation: https://www.tensorflow.org/install
- Tensorflow Tutorial: https://www.tensorflow.org/tutorials
- Will not need to know for HW1, but is necessary later in the course

PyTorch demo

https://colab.research.google.com/drive/1EZVQMCKYRFLVKOSejWYGjetfCBomrR9q

Gym

- Gym is a Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments
- Many popular environments such as brick-breaker, cartpole and mountain car are used to evaluate the performance of an algorithm



Useful Notation in Gym

- **`env = gym.make('ENVIRONMENT_NAME')`**: initializes environment
- **`env.reset()`**: resets environment to initial state
- **`env.step(action)`**: updates an environment by taking a specific action; returns the next agent observation, reward for taking action, and a termination flag, and other information such as metrics
- **`env.nA`**: number of actions in the environment
- **`env.nS`:** number of states in the environment
- **`env.P[s][a]`:** list of transition tuples (prob, next_state, reward, done)
- **`env.action_space`**: describes the numerical structure of the legitimate actions that can be applied to the environment
- **`env.observation_space`**: defines the structure as well as the legitimate values for the observation of the state of the environment

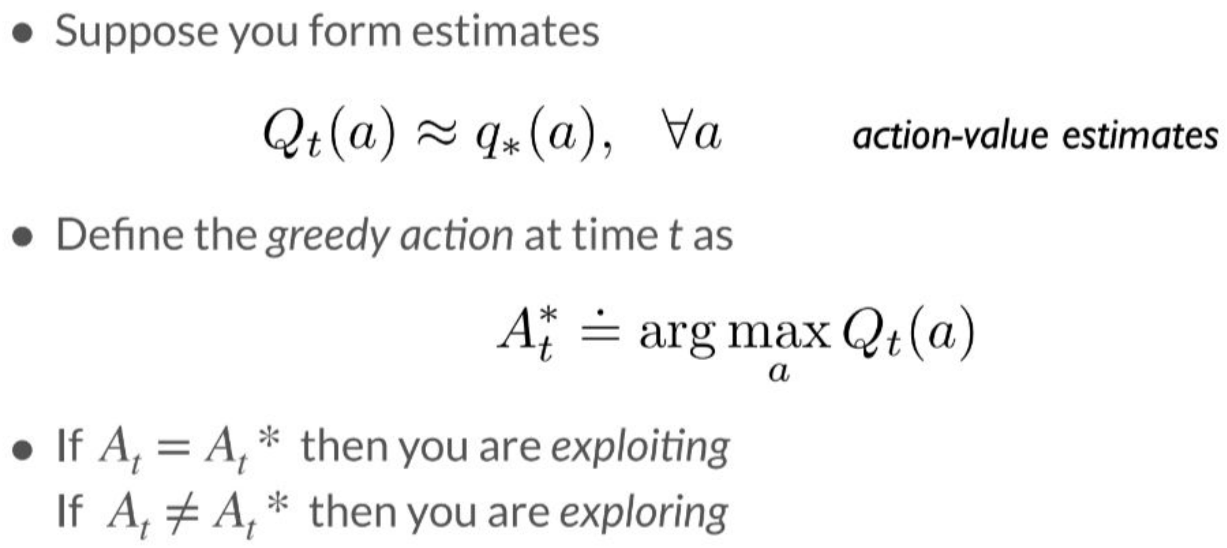
Read documentation (https://gymnasium.farama.org/) for more methods

Gym demo

https://colab.research.google.com/drive/1YsaZ024U6Hwr7uY_oL49ks9Tq6q4yuXF?usp=sharing

Multi Arm Bandits

- Exploration vs Exploitation



- Suppose you form estimates

$$Q_t(a) \approx q_*(a), \quad \forall a$$ *action-value estimates*

- Define the *greedy action* at time *t* as

$$A_t^* \doteq \arg\max_a Q_t(a)$$

- If $A_t = A_t{}^*$ then you are *exploiting*
  If $A_t \neq A_t{}^*$ then you are *exploring*

Bandit Algorithms

- **Epsilon Greedy**
  - Always exploit, but with probability epsilon choose a random action for exploration
  - Pros: simplest algorithm, Cons: Over time you are still exploring and choosing bad options (linear regret)
- **Optimistic Initialization**
  - Set very high initial values, that will be reduced as exploration progresses
  - The greedy action is always picked, so exploration reduces over time
- **UCB**
  - Set a upper bound on the true value of the action value
  - Allow bound to become smaller as an action is picked more often and we become more confident of the true value
  - Exploration reduces over time
- **Boltzmann Exploration**
  - Uses a Boltzmann distribution to pick actions, very similar to a softmax function
  - By using a probability distribution that is updated, it balances exploration and exploitation

Bandit Algorithms

- **Epsilon Greedy**
  - Always exploit, but with probability epsilon choose a random action for exploration
  - Pros: simplest algorithm, Cons: Over time you are still exploring and choosing bad options (linear regret)
- **Optimistic Initialization**
  - Set very high initial values, that will be reduced as exploration progresses
  - The greedy action is always picked, so exploration reduces over time
- **UCB**
  - Set a upper bound on the true value of the action value
  - Allow bound to become smaller as an action is picked more often and we become more confident of the true value
  - Exploration reduces over time
- **Boltzmann Exploration**
  - Uses a Boltzmann distribution to pick actions, very similar to a softmax function
  - By using a probability distribution that is updated, it balances exploration and exploitation

Bandit Algorithms

- **Epsilon Greedy**
  - Always exploit, but with probability epsilon choose a random action for exploration
  - Pros: simplest algorithm, Cons: Over time you are still exploring and choosing bad options (linear regret)
- **Optimistic Initialization**
  - Set very high initial values, that will be reduced as exploration progresses
  - The greedy action is always picked, so exploration reduces over time
- **UCB**
  - Set a upper bound on the true value of the action value
  - Allow bound to become smaller as an action is picked more often and we become more confident of the true value
  - Exploration reduces over time
- **Boltzmann Exploration**
  - Uses a Boltzmann distribution to pick actions, very similar to a softmax function
  - By using a probability distribution that is updated, it balances exploration and exploitation

Bandit Algorithms

- **Epsilon Greedy**
  - Always exploit, but with probability epsilon choose a random action for exploration
  - Pros: simplest algorithm, Cons: Over time you are still exploring and choosing bad options (linear regret)
- **Optimistic Initialization**
  - Set very high initial values, that will be reduced as exploration progresses
  - The greedy action is always picked, so exploration reduces over time
- **UCB**
  - Set a upper bound on the true value of the action value
  - Allow bound to become smaller as an action is picked more often and we become more confident of the true value
  - Exploration reduces over time
- **Boltzmann Exploration**
  - Uses a Boltzmann distribution to pick actions, very similar to a softmax function
  - By using a probability distribution that is updated, it balances exploration and exploitation

Questions?