

Lecture 1: Theoretical foundations of ML

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning

Supervised learning

- Set of possible input instances X
- Categories C , say $\{0, 1\}$
- Build a classification model $h : X \rightarrow C$
- Restrict the types of models; usually to bound computational costs.
 - Hypothesis space \mathcal{H} — e.g., linear separators
 - Search for best $h \in \mathcal{H}$
- How do we find the best h ?
 - Labelled training data (training set)
 - Choose h to minimize error (**loss**) with respect to the training set
 - Why should h generalize well to arbitrary data?

No free lunch

- ML algorithms minimize **training error** (with respect to the training set)
- The actual goal is to minimize **generalization error** (w.r.t all (unseen) inputs)

No Free Lunch Theorem [Wolpert, Macready 1997]

Averaged over all possible data distributions, every classification algorithm has the same error rate when classifying previously unobserved points.

- Is the situation hopeless?
- NFL theorem refers to prediction inputs coming from **all possible** distributions
- ML assumes training set is **representative** of overall data
 - Unseen instances follow roughly the same distribution as training set

A theoretical framework for ML

- X is the space of input **instances**
 - $C \subseteq X$ is the **target concept** to be learned
 - e.g., X is all emails, C is the set of spam emails
 - X is equipped with a probability distribution D
 - Any instance is a random sample from X drawn using D
 - In particular, training set and test set contain such random samples
-
- \mathcal{H} is a set of **hypotheses**
 - Each $h \in \mathcal{H}$ identifies a subset of X
 - Choose the best $h \in \mathcal{H}$ as model
 - **True error:** Probability that h incorrectly classifies $x \in X$ drawn randomly according to D
 - $\text{err}_D(h) = \text{Prob}(h \Delta C)$
 - $h \Delta C = (h \setminus C) \cup (C \setminus h)$
symmetric diff
 - **Training error:** Given a (finite) training sample $S \subseteq X$
 - $\text{err}_S(h) = |S \cap (h \Delta C)| / |S|$



A theoretical framework for ML

- X , inputs with distribution D
- $C \subseteq X$, target concept
- $h \in \mathcal{H}$, hypothesis (model) for C
- True error: $\text{err}_D(h) = \text{Prob}(h \Delta C)$
- Training error:
 $\text{errs}(h) = |S \cap (h \Delta C)| / |S|$

Goal

Minimizing training error should correspond to minimizing true error

- **Overfitting** Low training error but high true error
- **Underfitting** Cannot achieve low training/true error
- These errors are related to the **representational capacity** of \mathcal{H}
 - How expressive is \mathcal{H} ? How many different concepts can it capture?
 - Capacity too high — overfitting
 - Capacity too low — underfitting

Probably Approximately Correct (PAC) learning

- Assume \mathcal{H} is finite — use $|\mathcal{H}|$ for capacity
- **Probably Approximately Correct learning**
With high probability, the hypothesis h that fits the sample S also fits the concept C approximately correctly

Theorem (PAC learning guarantee)

Let $\delta, \epsilon > 0$. Let S be a training set of size $n \geq \frac{1}{\epsilon}(\ln |\mathcal{H}| + \ln(1/\delta))$ drawn using D . With probability $\geq 1 - \delta$, every $h \in \mathcal{H}$ with training error zero has true error $< \epsilon$.

- Size of the sample required for PAC guarantee determined by parameters δ, ϵ
 - Smaller δ means higher probability of find a good hypothesis
 - Smaller ϵ means better performance with respect to generalization

Probably Approximately Correct (PAC) learning

Theorem (Uniform convergence)

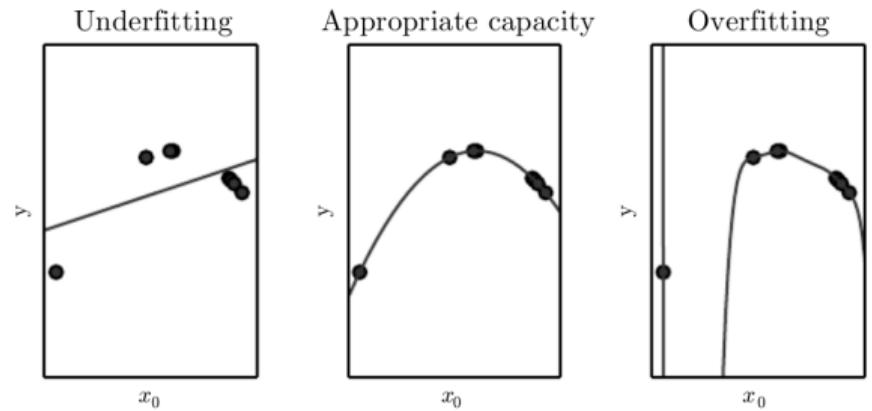
Let $\delta, \epsilon > 0$. Let S be a training set of size $n \geq \frac{1}{2\epsilon^2}(\ln |\mathcal{H}| + \ln(2/\delta))$ drawn using D . With probability $\geq 1 - \delta$, every $h \in \mathcal{H}$ satisfies $|\text{err}_S(h) - \text{err}_D(h)| \leq \epsilon$.

- Stronger guarantee: even if we cannot achieve zero training error, the additional generalization error is bounded
- What if \mathcal{H} is not finite?
- Other measures of capacity — e.g. VC-dimension
- Analogous convergence theorems in terms of VC-dimension

Overfitting and underfitting

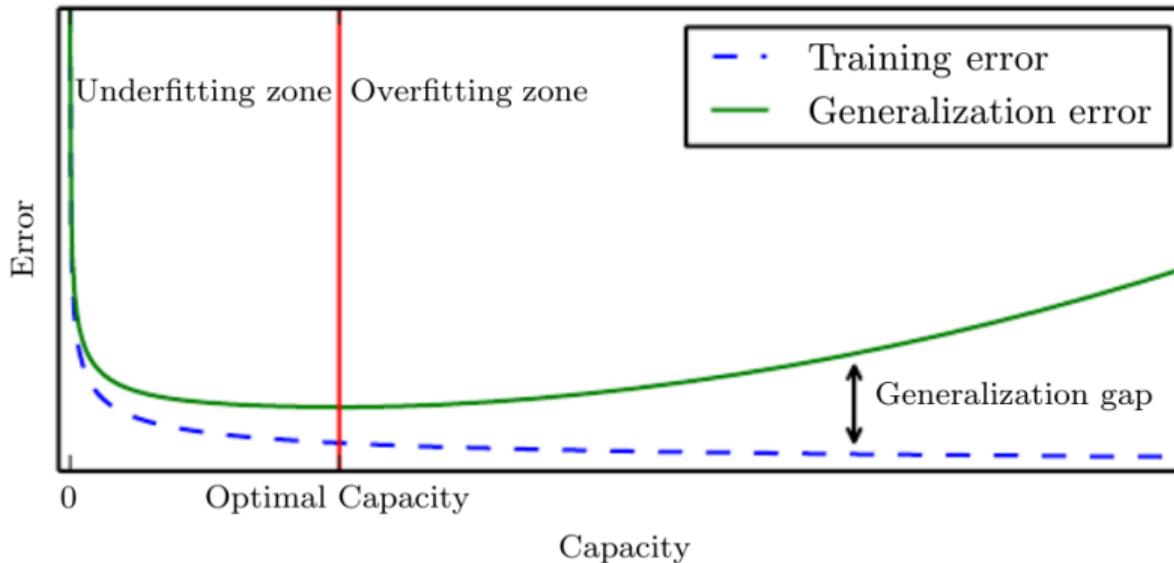
Example: Regression

- \mathcal{H}_d is set of polynomials of degree d
- Increasing d increases expressiveness — higher representational capacity
- Using too high a d results in overfitting
- Using too low a d results in underfitting



- Random points lying along a quadratic
- Linear function underfits
- Quadratic fits and generalizes well
- Degree 9 polynomial overfits

Capacity and error



- As capacity increases, training error decreases
- Initially, generalization error also decreases
- At some point, generalization error starts increasing
- Optimum capacity is not where training error is minimum

Theory and practice

- Deep learning models are too complex to compute representational capacity explicitly
- May not even be able to achieve true representational capacity
- Effective capacity limited by capabilities of parameter estimation algorithm (backpropagation with optimization)
- Parameter estimation is a complex nonlinear optimization

Regularization

- Add a penalty for model complexity to the loss function
- Trade off lower training error against penalty

Hyperparameters

- Settings that adjust the capacity — e.g., degree of polynomial
- Set externally, not learned
- Search hyperparameter combinations for optimal settings

Summary

- Supervised learning builds a model that minimize training error
- Real goal is to minimize generalization error
- PAC learning provides a theoretical framework to justify this
- Discrepancies in representational capacity of models can cause underfitting or overfitting
- In practice, use regularization and hyperparameter search to identify optimum capacity

Lecture 2 Part A: VC Dimension

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning

Representational capacity

PAC learning guarantee

Let \mathcal{H} be a hypothesis class, $\delta, \epsilon > 0$ and S a training set of size $n \geq \frac{1}{\epsilon}(\ln |\mathcal{H}| + \ln(1/\delta))$ drawn using D . With probability $\geq 1 - \delta$, every $h \in \mathcal{H}$ with true error $\text{err}_D > \epsilon$ has training error $\text{err}_S > 0$.

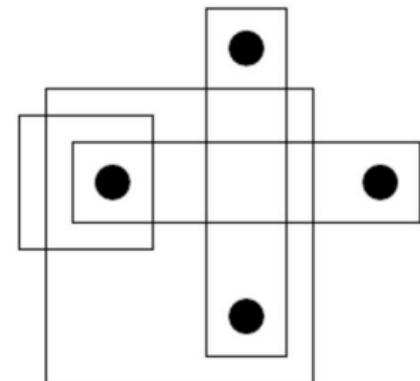
Uniform convergence

Let \mathcal{H} be a hypothesis class, $\delta, \epsilon > 0$. If a training set S of size $n \geq \frac{1}{2\epsilon^2}(\ln |\mathcal{H}| + \ln(2/\delta))$ is drawn using D , then with probability $\geq 1 - \delta$, every $h \in \mathcal{H}$ satisfies $|\text{err}_S(h) - \text{err}_D(h)| \leq \epsilon$.

- $|\mathcal{H}|$ is **representational capacity**, when \mathcal{H} is finite
- How do we adapt and apply these bounds when \mathcal{H} is infinite?

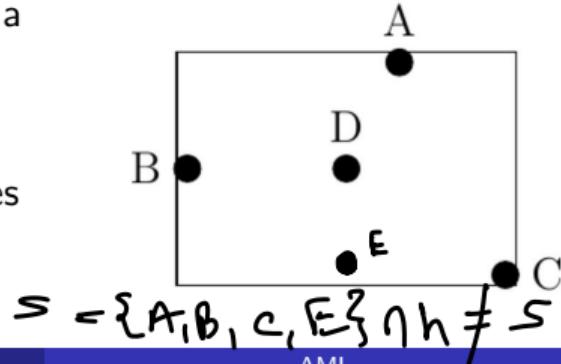
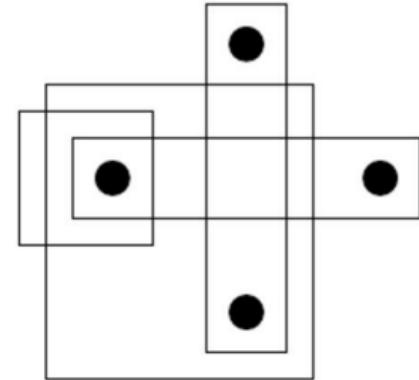
Shattering

- Set system: (X, \mathcal{H})
 - X is a set — instance space
 - \mathcal{H} , set of subsets of X — set of possible classifiers / hypotheses
- $A \subseteq X$ is shattered by \mathcal{H} if every subset of A is given by $A \cap h$ for some $h \in \mathcal{H}$
 - Every way of splitting A is captured by a hypothesis in \mathcal{H}
 - $2^{|A|}$ different subsets of A
- Example:
 - $X = \mathbb{R} \times \mathbb{R}$
 - \mathcal{H} : Axis-parallel rectangles
 - A : Four points forming a diamond
 - \mathcal{H} shatters A



VC-Dimension [Vapnik-Chervonenkis]

- VC-Dimension of \mathcal{H} — size of the largest subset of X shattered by \mathcal{H}
 - For axis-parallel rectangles, VC-dimension is at least 4
 - Not a **universal** requirement — some sets of size 4 may not be shattered
- No set of size 5 can be shattered by axis-parallel rectangles
 - Draw a **bounding box** rectangle — each edge touches a boundary point
 - At least one point lies inside the bounding box
 - Any set that includes the boundary points also includes the interior point



VC-Dimension, Examples

- Intervals of reals have VC-dimension 2
 - $X = \mathbb{R}$, $\mathcal{H} = \{[a, b] \mid a \leq b \in \mathbb{R}\}$
 - Cannot shatter 3 points: consider subset with first and third point
- Pairs of intervals of reals have VC-dimension 4
 - $X = \mathbb{R}$, $\mathcal{H} = \{[a, b] \cup [c, d] \mid a \leq b, c \leq d \in \mathbb{R}\}$
 - Cannot shatter 5 points: consider subset with first, third and fifth point
- Finite sets of real numbers
 - $X = \mathbb{R}$, $\mathcal{H} = \{Z \mid Z \subseteq \mathbb{R}, |Z| < \infty\}$
 - Can shatter any finite set of reals — VC-dimension is infinite
- Convex polygons, $X = \mathbb{R} \times \mathbb{R}$
 - For any n , place n points on unit circle
 - $\mathcal{H} =$ Each subset of these points is a convex polygon — VC-dimension is infinite

VC-dimension and machine learning

PAC learning guarantee

Let \mathcal{H} be a hypothesis class, $\delta, \epsilon > 0$ and S a training set of size $n \geq \frac{1}{\epsilon}(\ln |\mathcal{H}| + \ln(1/\delta))$ drawn using D . With probability $\geq 1 - \delta$, every $h \in \mathcal{H}$ with true error $\text{err}_D > \epsilon$ has training error $\text{err}_S > 0$.

- We can rewrite this using VC-dimension. Can similarly restate uniform convergence.

Sample bound using VC-dimension

For any class \mathcal{H} and distribution D , if a training sample S is drawn using D of size $O\left(\frac{1}{\epsilon} \left[\text{VC-dim}(\mathcal{H}) \ln \frac{1}{\epsilon} + \ln \frac{1}{\delta} \right]\right)$, then with probability $\geq 1 - \delta$,

- every $h \in \mathcal{H}$ with true error $\text{err}_D(h) \geq \epsilon$ has training error $\text{err}_S(h) > 0$,
- i.e., every $h \in \mathcal{H}$ with training error $\text{err}_S(h) = 0$. has true error $\text{err}_D(h) < \epsilon$

Summary

- PAC learning and uniform convergence use size of finite hypothesis set as measure of representational capacity
- VC-dimension provides a way of measuring capacity for infinite hypothesis sets
- VC-dimension may be finite or infinite
- For finite VC-dimension, we have analogues of PAC learning guarantee and uniform convergence
- Note that these theoretical bounds are hard to use in practice
- Difficult, if not impossible, to compute VC-dimension for complex models

Lecture 2 Part B: Loss functions

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning

Loss Functions for Gradient descent

- What loss functions should we use?
- Learning algorithms estimate parameters of a model based on training data
- Parameter estimate is through gradient descent
 - Define a **loss function** measuring the error with respect to training data
 - Compute gradients of the loss function with respect to each parameter
 - Adjust parameters by a small step in direction opposite to gradients
- Typical loss functions include mean squared error (MSE) and cross entropy
- How do arrive at these loss functions?

Maximum likelihood estimators (MLE)

- Build a model M from training data $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- Learning — define M by computing parameters θ
- Model predicts value \hat{y}_i on input x_i with probability $P_{\text{model}}(\hat{y}_i | x_i, \theta)$
This is a probability distribution
- Probability of predicting correct value is $P_{\text{model}}(y_i | x_i, \theta)$
- Likelihood is
$$\prod_{i=1}^n P_{\text{model}}(y_i | x_i, \theta)$$
- Find M that maximizes the likelihood

Maximizing Log likelihood

- Maximize the likelihood $\prod_{i=1}^n P_{\text{model}}(y_i | x_i, \theta)$
- Since \log is an increasing function, so we can equivalently maximize \log likelihood

$$\log \left(\prod_{i=1}^n P_{\text{model}}(y_i | x_i, \theta) \right) = \sum_{i=1}^n \log(P_{\text{model}}(y_i | x_i, \theta))$$

- Log likelihood is a function of the parameters θ of the model M

$$\mathcal{L}(\theta) = \sum_{i=1}^n \log(P_{\text{model}}(y_i | x_i, \theta))$$

- To find θ maximizing $\mathcal{L}(\theta)$, solve $\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = 0$ (gradient is zero at optimum)
- Or, do gradient descent using $\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$

Regression and MSE loss

- Training input is $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
 - Noisy outputs from a linear function $y_i = w^T x_i + \epsilon$
 - $\epsilon \sim \mathcal{N}(0, \sigma^2)$: Gaussian noise, mean 0, fixed variance σ^2
 - $y_i \sim \mathcal{N}(\mu_i, \sigma^2)$ where $\mu_i = w^T x_i$
- Model gives us an estimate for w , so regression learns μ_i for each x_i

$$P_{\text{model}}(y_i | x_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \mu_i)^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - w^T x_i)^2}{2\sigma^2}}$$

- Log likelihood (assuming natural logarithm)

$$\mathcal{L}(\theta) = \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - w^T x_i)^2}{2\sigma^2}} \right) = n \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \sum_{i=1}^n \frac{(y_i - w^T x_i)^2}{2\sigma^2}$$

Note: Here $\theta = w$ are the model parameters

Regression and MSE loss

- Log likelihood: $\mathcal{L}(\theta) = n \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \sum_{i=1}^n \frac{(y_i - w^T x_i)^2}{2\sigma^2}$
- Let $\hat{y}_i = w^T x_i$ is value predicted by the model
- To maximize $\mathcal{L}(\theta)$ with respect to w , ignore all terms that do not depend on w
- Optimum value of w is given by

$$w_{MSE} = \arg \max_w \left[- \sum_{i=1}^n (y_i - \hat{y}_i)^2 \right] = \arg \min_w \left[\sum_{i=1}^n (y_i - \hat{y}_i)^2 \right]$$

Minimizing Squared Error!

- Assuming data points are generated by linear function and then perturbed by Gaussian noise, MSE is the “correct” loss function to maximize likelihood (and minimize cross entropy)

Cross entropy

- Let $X = \{x_1, x_2, \dots, x_k\}$ with a probability distribution P

- Entropy is defined as
$$H(P) = - \sum_{i=1}^k P(x_i) \log P(x_i)$$

- Average number of bits to encode each element of X under dist P
- Given two distributions P and Q over X , cross entropy is defined as

$$H(P, Q) = - \sum_{i=1}^k P(x_i) \log Q(x_i)$$

→ distribution output of model's

- Imagine an encoding based on Q where true distribution is P
- Again, average number of bits to encode each element of X in this situation
- Note that cross entropy is not symmetric: $H(P, Q) \neq H(Q, P)$

Maximizing Log likelihood

- Recall: Log likelihood is a function of the learned parameters θ

$$\mathcal{L}(\theta) = \sum_{i=1}^n \log(P_{\text{model}}(y_i | x_i, \theta))$$

- Define $P_{\text{data}}(y | x_i)$ as follows: $P_{\text{data}}(y | x_i) = \begin{cases} 1 & \text{if } y = y_i \\ 0 & \text{otherwise} \end{cases}$
- For each x_i , $P_{\text{data}}(y_i | x_i) = 1$, so rewrite log likelihood as

$$\sum_{i=1}^n \log(P_{\text{model}}(y_i | x_i, \theta)) = \sum_{i=1}^n P_{\text{data}}(y_i | x_i) \cdot \log(P_{\text{model}}(y_i | x_i, \theta))$$

- We can rewrite the log-likelihood as

$$\mathcal{L}(\theta) = \sum_{i=1}^n P_{\text{data}}(y_i | x_i) \log(P_{\text{model}}(y_i | x_i, \theta))$$

Cross entropy and MLE

- Maximum likelihood estimator (MLE) — maximize

$$\mathcal{L}(\theta) = \sum_{i=1}^n P_{\text{data}}(y_i | x_i) \log(P_{\text{model}}(y_i | x_i, \theta))$$

- P_{model} is an estimate for the true distribution P_{data}

- $H(P_{\text{data}}, P_{\text{model}}) = - \sum_{i=1}^k P_{\text{data}}(y | x_i) \log(P_{\text{model}}(y | x_i, \theta))$
- $H(P_{\text{data}}, P_{\text{model}}) = -\mathcal{L}(\theta)$
- Minimizing cross entropy is the same as maximizing likelihood
- The **cross entropy loss function** is a special form of this generic observation

Binary classification

- Compute linear output $z_i = w^T x_i$, then apply sigmoid $\sigma(z) = \frac{1}{1 + e^{-z}}$
- Let $a_i = \sigma(z_i)$. So, $P_{\text{model}}(y_i = 1) = a_i$, $P_{\text{model}}(y_i = 0) = 1 - a_i$
- Cross entropy: $\sum_{i=1}^n \sum_{j \in \{0,1\}} P_{\text{data}}(y_i = j) \log(P_{\text{model}}(y_i = j | x_i, \theta))$
- Expand:
$$\sum_{i=1}^n P_{\text{data}}(y_i = 0) \log P_{\text{model}}(y_i = 0 | x_i, \theta) + P_{\text{data}}(y_i = 1) \log P_{\text{model}}(y_i = 1 | x_i, \theta)$$
- Equivalently, $\sum_{i=1}^n (1 - y_i) \cdot \log(1 - a_i) + y_i \cdot \log a_i$
- Recommended loss function, directly minimizes cross entropy

Summary

- Our goal is to find a maximum likelihood estimator
- Gradient descent uses a loss function to optimize parameters
- Finding MLE is equivalent to minimizing cross entropy $H(P_{\text{data}}, P_{\text{model}})$
- Applying this to a given situation, we arrive at concrete loss functions
 - Mean square error for regression
 - “Cross entropy” for classification

Lecture 3: Deep Neural Networks

Pranabendu Misra
Chennai Mathematical Institute

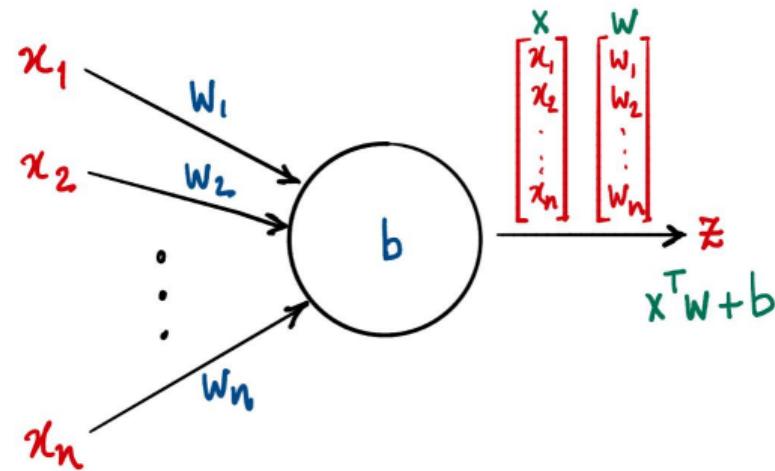
Advanced Machine Learning 2023

Linear separators and perceptrons

- Perceptrons define linear separators

$$x^T w + b$$

- $x^T w + b > 0$, classify Yes (+1)
- $x^T w + b < 0$, classify No (-1)



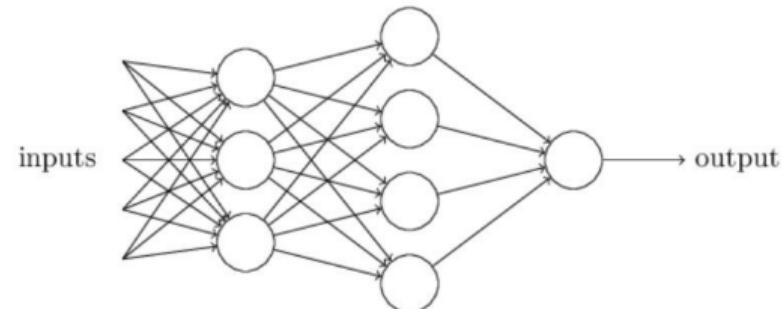
Linear separators and perceptrons

- Perceptrons define linear separators

$$x^T w + b$$

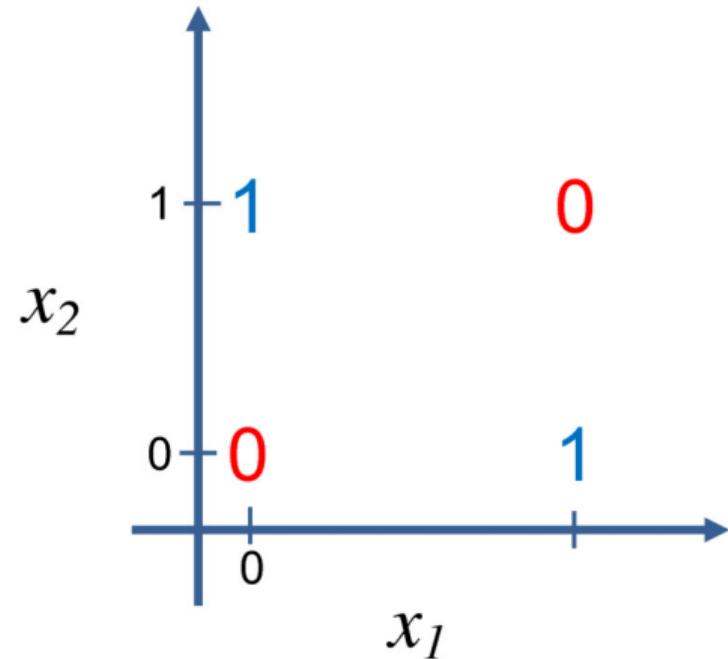
- $x^T w + b > 0$, classify Yes (+1)
- $x^T w + b < 0$, classify No (-1)

- **Unfortunately!** Network of perceptrons still defines only a linear separator



Linear separators and perceptrons

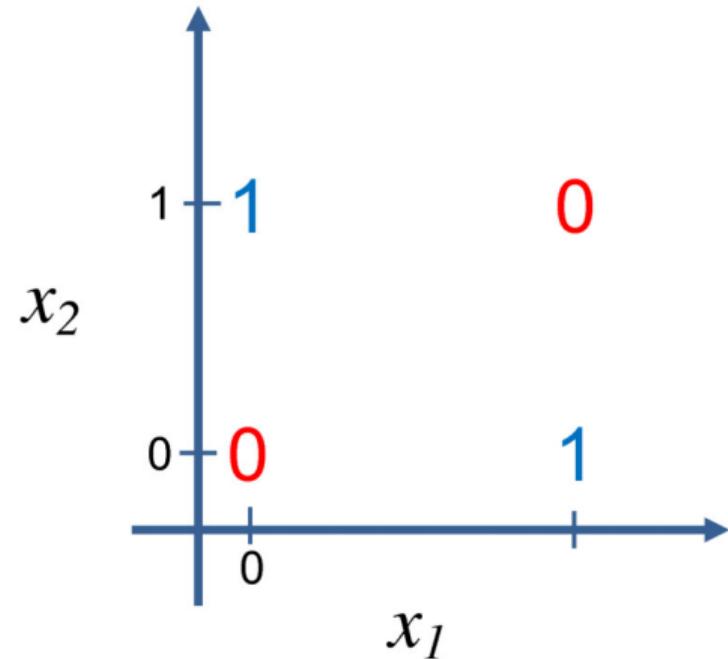
- Perceptrons define linear separators
$$x^T w + b$$
 - $x^T w + b > 0$, classify Yes (+1)
 - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator
- Linear separators cannot describe XOR



Linear separators and perceptrons

- Perceptrons define linear separators
$$x^T w + b$$
 - $x^T w + b > 0$, classify Yes (+1)
 - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator
- Linear separators cannot describe XOR

We need non-linearity!



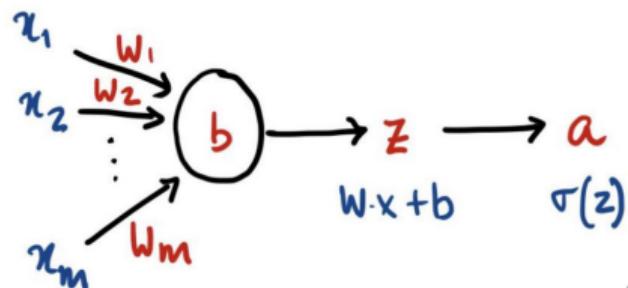
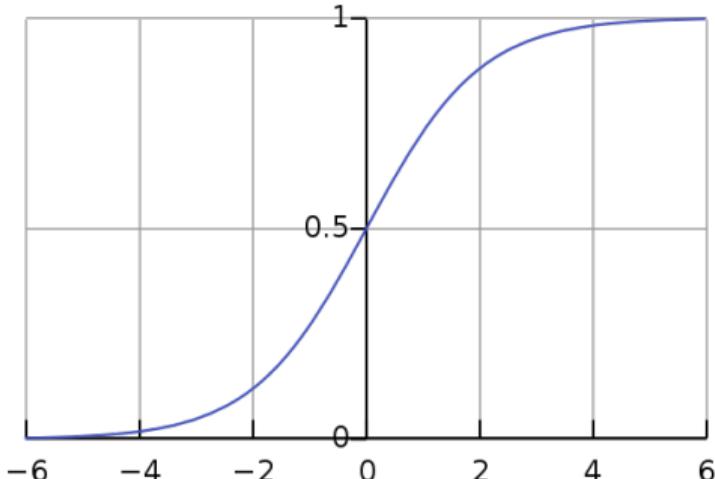
Linear separators and perceptrons

- Perceptrons define linear separators
 $x^T w + b$
 - $x^T w + b > 0$, classify Yes (+1)
 - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator
- Linear separators cannot describe XOR

We need non-linearity!

- Introduce a non-linear **activation** function
 - Traditionally sigmoid,
 $\sigma(z) = 1/(1 + e^{-z})$

This is a neuron!



(Feed forward) Neural networks

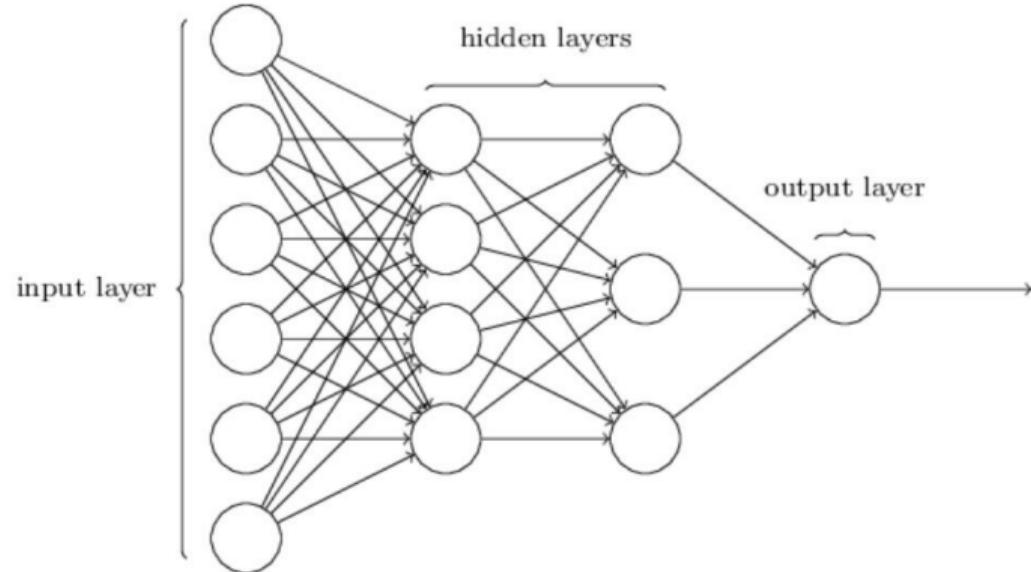
- Acyclic network of perceptrons with non-linear activation functions

- **Universal Approximation**

Theorem: With just 1 hidden layer, a neural network can approximate any function for any degree of precision.

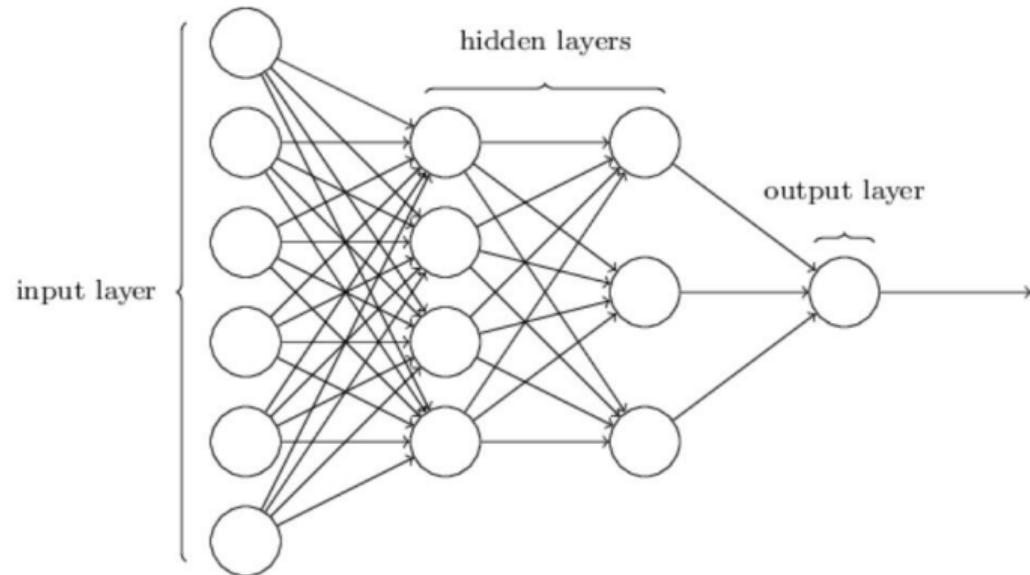
- For any function f , it is possible to construct such a neural network (by hand).

For example the XOR function.



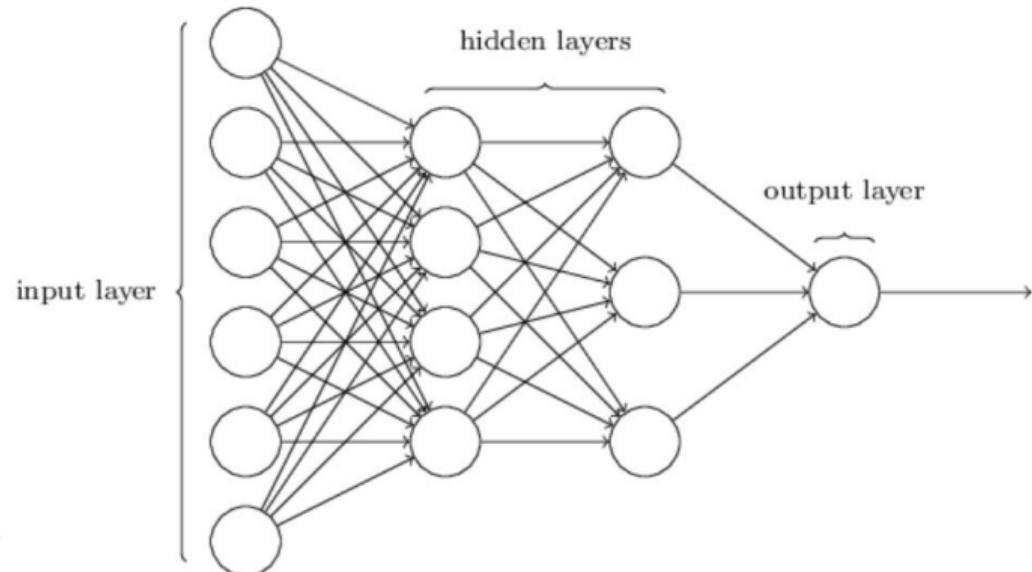
(Feed forward) Neural networks

- Acyclic network of perceptrons with non-linear activation functions
 - We decide the structure or **Hyper-Parameters** of the network.
 - The **Loss** function to use and any **Regularization** are also hyper-parameters.
 - The value of it's **Parameters θ** (weights and biases of each neuron) are found by **training** it.
 - **Objective:** Given a training set S , we train a neural network to give **least** generalization error.
 - We estimate the generalization error using a test set T .



(Feed forward) Neural networks

- How do we **train** a neural network?
 - We have already chosen the hyperparameters of the neural network, based on the ML task at hand.
 - Initialize the parameters θ to some suitable random numbers.
 - Let the loss-function be $\ell(\theta, S)$ on the output of the neural network given the set S as input.
 - **To Solve Optimization problem:** Given training set S , find the values for θ with least $\ell(\theta, S)$.



(Feed forward) Neural networks

- How do we compute a neural network?
 - Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

Highly Non-Trivial Problem!

(Feed forward) Neural networks

- How do we compute a neural network?

- Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

Highly Non-Trivial Problem!

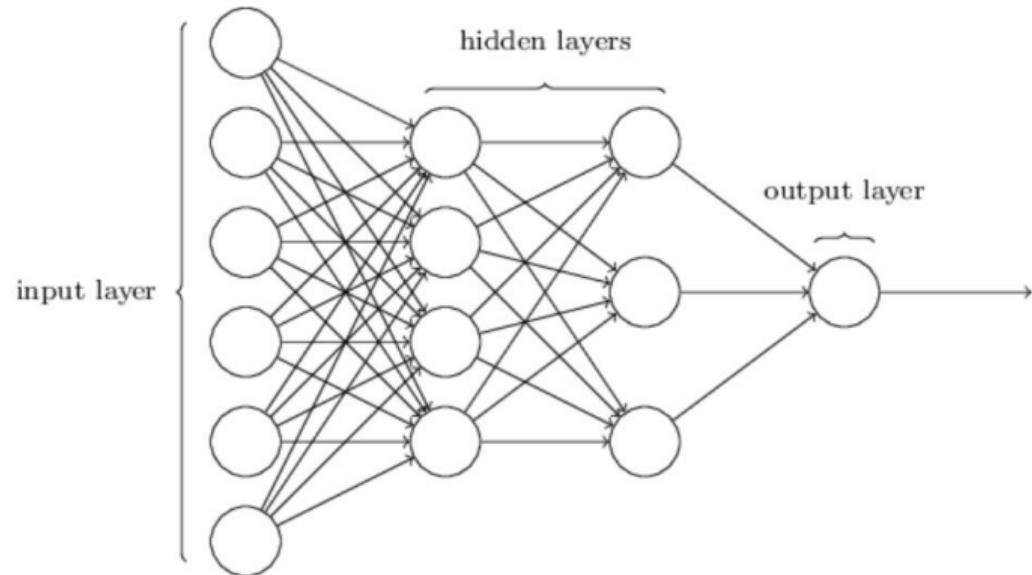
- We use the most basic optimization method:

Gradient Descent

- Update θ so that $\ell(\theta, S)$ decreases

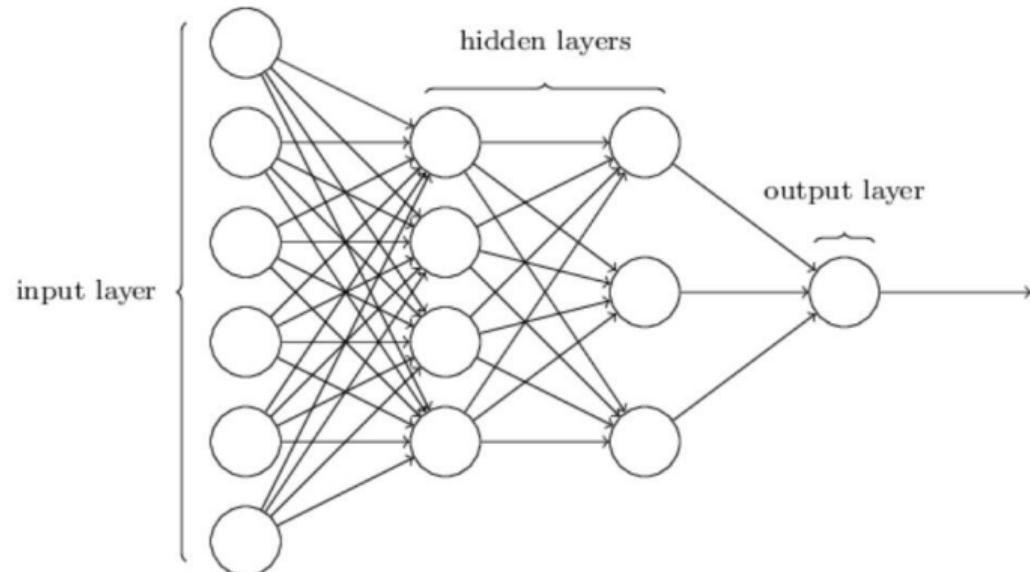
$$\theta \leftarrow \theta - \alpha \cdot \frac{d\ell(\theta, S)}{d\theta}$$

- α is the learning rate.



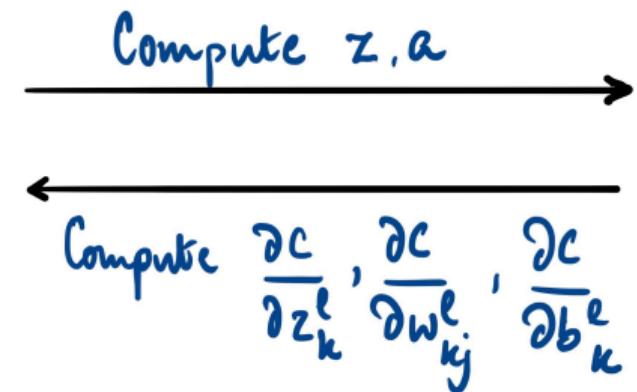
(Feed forward) Neural networks

- Acyclic network of perceptrons with non-linear activation functions
- Hyper-Parameters
 - Network architecture:
Interconnection of layers
 - Hidden layer activation functions
 - Output layer activation function
 - Loss function for gradient descent
 - Initial values of weights and biases
 - The variant of Gradient Descent to use.



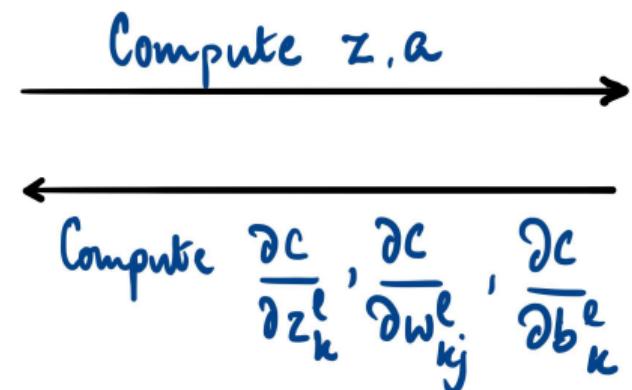
Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks



Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks
- Forward pass, compute outputs, activation values
- Backward pass, use chain rule to compute all gradients in one scan



Training a neural network

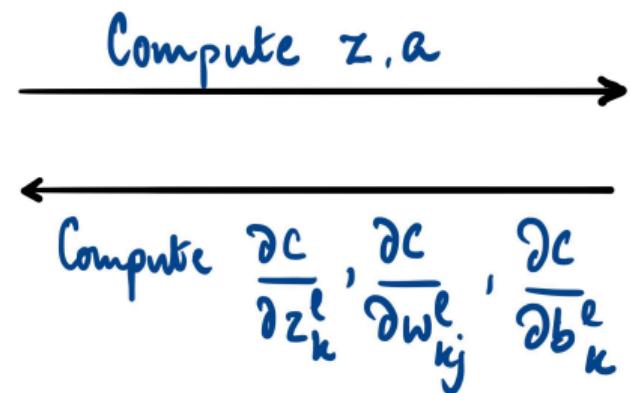
- Backpropagation — efficient implementation of gradient descent for neural networks
 - Forward pass, compute outputs, activation values
 - Backward pass, use chain rule to compute all gradients in one scan
 - Stochastic gradient descent (SGD)
 - Update parameters in minibatches
 - Epoch: set of minibatches that covers entire training data

Compute z, a

Compute $\frac{\partial C}{\partial z_k^e}$, $\frac{\partial C}{\partial w_{kj}^e}$, $\frac{\partial C}{\partial b_k^e}$

Training a neural network

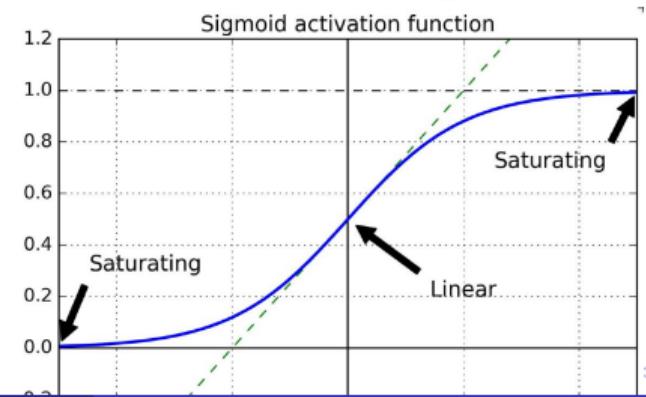
- Backpropagation — efficient implementation of gradient descent for neural networks
- Forward pass, compute outputs, activation values
- Backward pass, use chain rule to compute all gradients in one scan
- Stochastic gradient descent (SGD)
 - Update parameters in minibatches
 - Epoch: set of minibatches that covers entire training data
- Difficulties: slow convergence, vanishing and exploding gradients



Unstable gradients

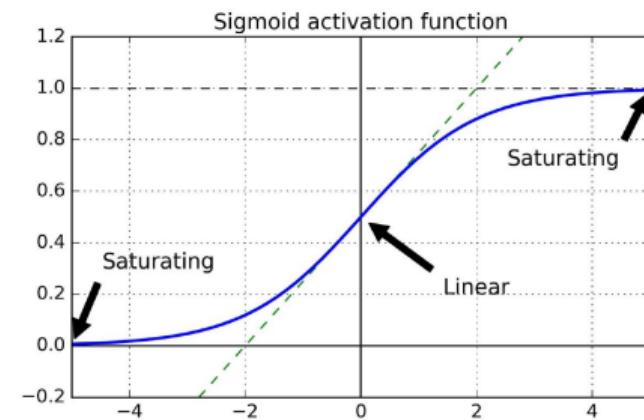
- **Vanishing gradients** — gradients become smaller towards lower layers (closer to input)
 - Gradient descent updates leave these layers' parameters virtually unchanged
- Also exploding gradients in deep networks
- In general, unstable gradients, different layers learn at different speeds
- Choose hyper-parameters so that Gradient Descent works well.
- [Xavier Glorot and Joshua Bengio, 2010]
 - Random initialization, e.g. $\mathcal{N}(0, 1)$
 - Variance keeps increasing going forward
 - Saturating sigmoid function

Compute z, a →
← Compute $\frac{\partial C}{\partial z_k^e}, \frac{\partial C}{\partial w_{kj}^e}, \frac{\partial C}{\partial b_k^e}$



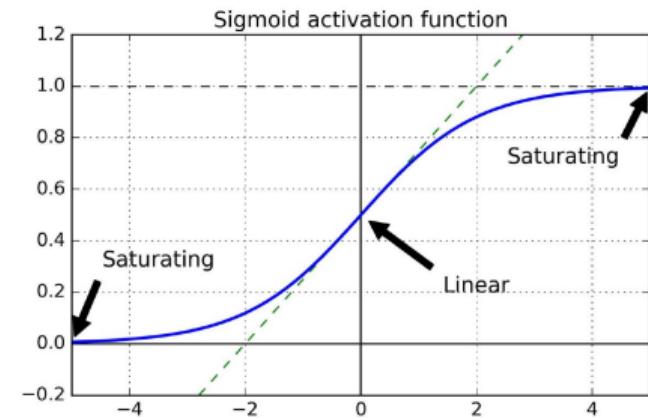
Initializing neural networks

- Want “signal” to flow well in both directions during backpropagation
 - Signal should not die out, explode, saturate
- [Glorot,Bengio] Gradients should have equal variance before and after flowing through a layer in both directions
 - Equal variance requires $fan_{in} = fan_{out}$
- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
 - Gaussian, $\mathcal{N}(0, 1/fan_{avg})$
 - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\frac{3}{fan_{avg}}}$



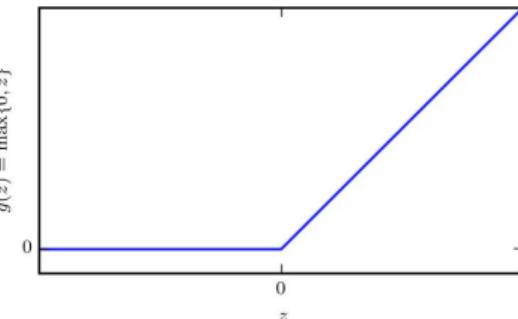
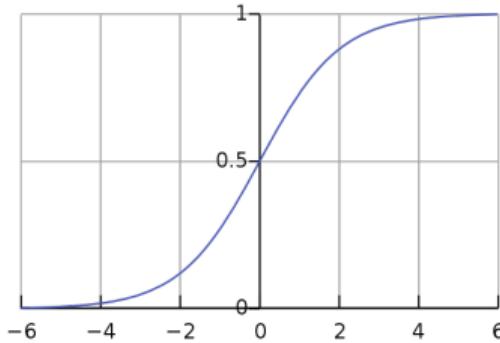
Initializing neural networks

- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
 - Gaussian, $\mathcal{N}(0, 1/fan_{avg})$
 - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\frac{3}{fan_{avg}}}$
- [Yann LeCun, 1990s] earlier proposed the same with fan_{avg} replaced by fan_{in}
 - Equivalent if $fan_{in} = fan_{out}$
- Use Other Activation Functions**
 - ReLU, [He et al, 2015], $\mathcal{N}(0, 2/fan_{in})$



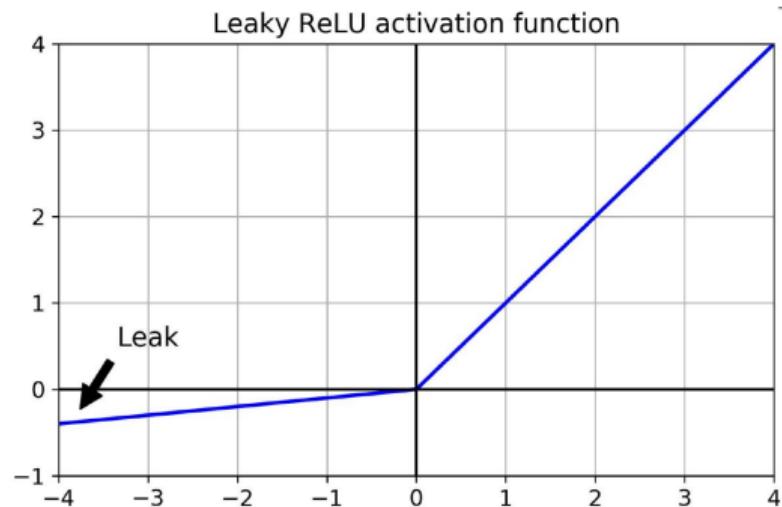
Non-saturating activation functions

- Sigmoid was initially chosen as a “smooth” step
- Rectified linear unit (ReLU):
$$g(z) = \max(0, z)$$
 - Fast to compute
 - Non-differentiable point not a bottleneck
- “Dying ReLU”
 - Neuron dies — weighted sum of outputs is negative for all training samples
 - With a large learning rate, half the network may die!



Non-saturating activation functions

- Leaky ReLU, $\max(\alpha z, z)$
 - “Leak” α is a hyperparameter
- RReLU — random leak
 - Pick α from a random range during training
 - Fix to an average value when testing
 - Seems to work well, act as a regularizer
- PReLU — parametric ReLU [He et al, 2015]
 - α is learned during training
 - Often outperforms ReLU, but could lead to overfitting



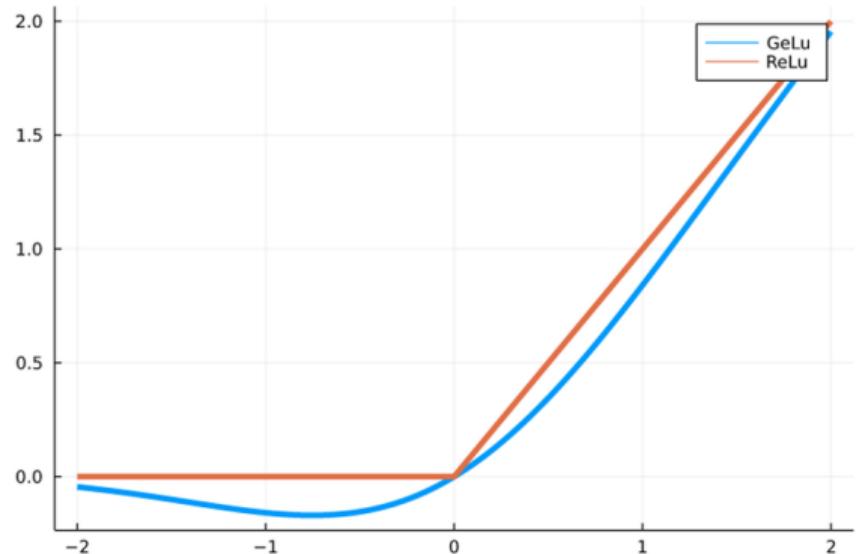
Non-saturating activation functions

- GELU — Gaussian Error Linear Unit
[Hendrycks and Gimpel, 2016]

$$\text{GELU}(x) = x \cdot P(Y \leq x)$$

P is the Probability Density Function of the Gaussian Distribution $Y \sim \mathcal{N}(0, 1)$

- Differential at zero with non-zero gradient
- Fast Implementation possible
- Used in most Transformer like GPT, BERT, ...



Lecture 4: Deep Neural Networks

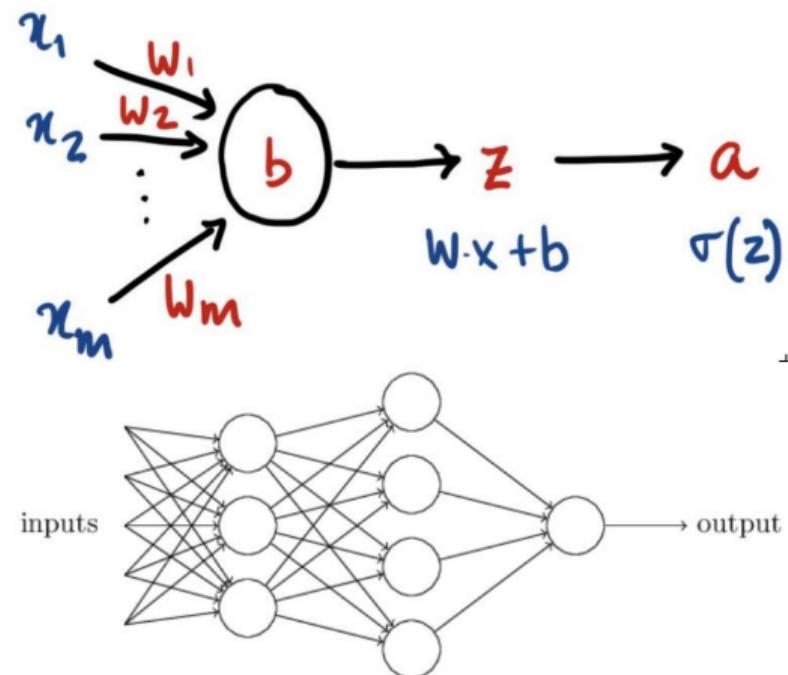
Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

Linear separators and perceptrons

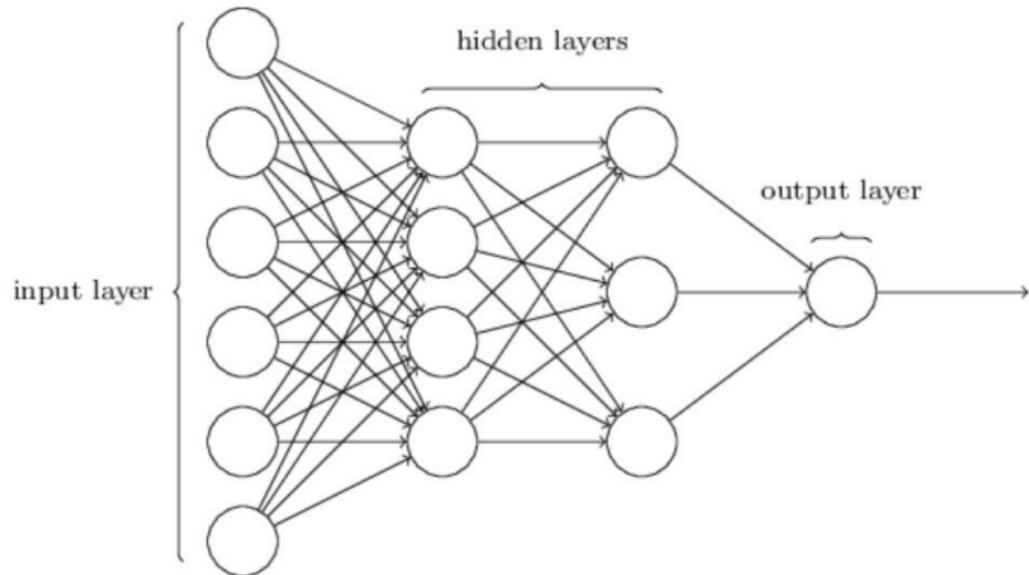
- Perceptrons define linear separators
 $x^T w + b$
 - $x^T w + b > 0$, classify Yes (+1)
 - $x^T w + b < 0$, classify No (-1)
- Network of perceptrons still defines only a linear separator
- Introduce a non-linear **activation** function
 - Traditionally sigmoid,
 $\sigma(z) = 1/(1 + e^{-z})$

This is a neuron!



(Feed forward) Neural networks

- **Neural Network:** Acyclic network of perceptrons with non-linear activation functions.
- **Universal Approximation Theorem:** With just 1 hidden layer, a neural network can approximate any function for any degree of precision.
- However, the network needs to be **deep** with many hidden layers. Shallow networks will require a very large number of neurons.

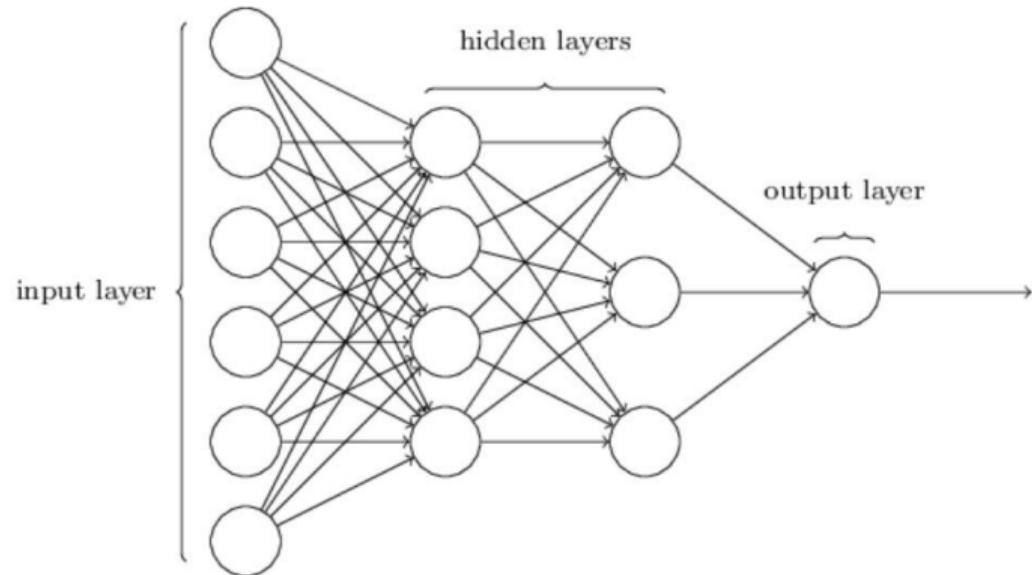


(Feed forward) Neural networks

- How do we train a neural network?
 - Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

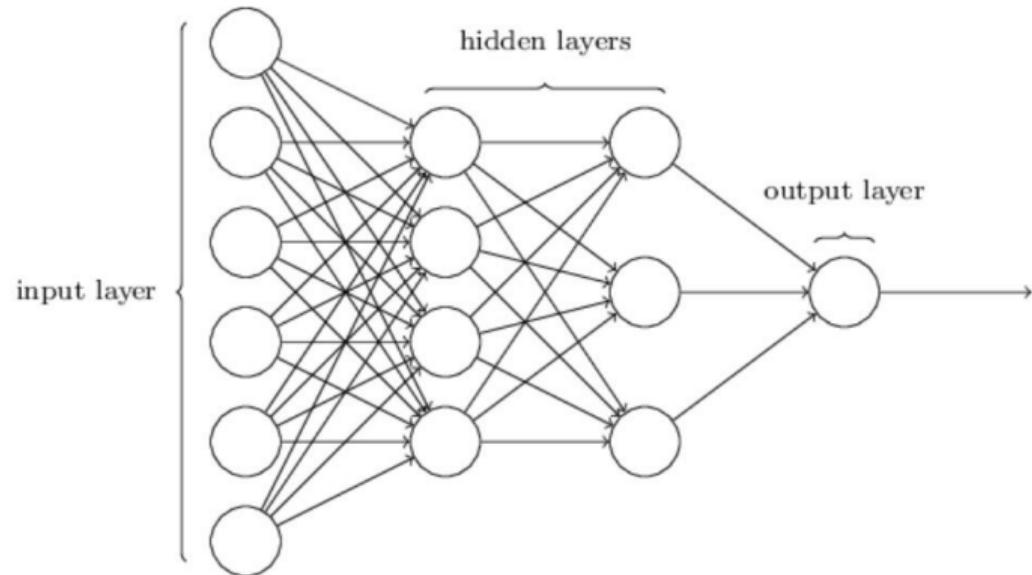
Highly Non-Trivial Problem!

- We use the most basic optimization method:
Gradient Descent
- Update θ so that $\ell(\theta, S)$ decreases
$$\theta \leftarrow \theta - \alpha \cdot \frac{d\ell(\theta, S)}{d\theta}$$
- α is the **learning rate**.



(Feed forward) Neural networks

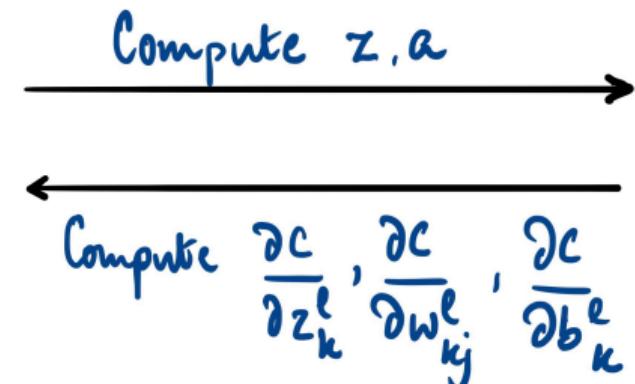
- Acyclic network of perceptrons with non-linear activation functions
 - We decide the structure or **Hyper-Parameters** of the network.
 - The **Loss** function to use and any **Regularization** are also hyper-parameters.
 - The value of it's **Parameters θ** (weights and biases of each neuron) are found by **training** it.
 - **Objective:** Given a training set S , we train a neural network to give **least** generalization error.
 - We estimate the generalization error using a test set T .



An important consideration in these choices is the effect on the **Gradient Descent** process.

Training a neural network

- **Backpropagation** — efficient implementation of gradient descent for neural networks
- **Forward pass:** compute outputs for each item in Training Set S and the loss function value over S
- **Backward pass:** use Chain rule to compute gradients of the loss function with respect to every neural network parameter.
- Stochastic gradient descent (SGD)
 - Gradient Descent using whole S is too slow
 - Sample a **mini-batches** S' from S and do one step of Gradient Descent with S'
 - **Epoch:** A sequence of minibatches that covers entire training set S



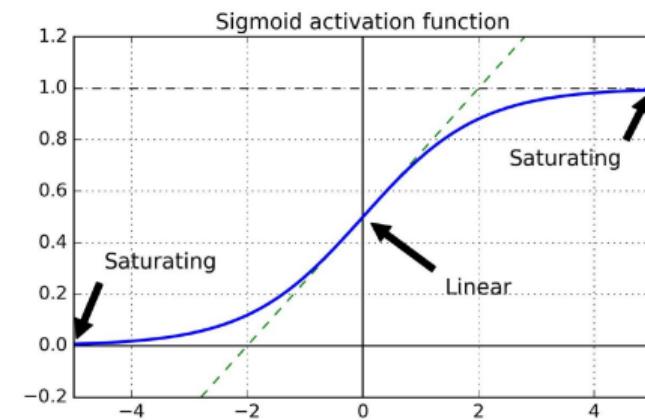
Unstable gradients

- **Vanishing gradients** — gradients become smaller towards lower layers (closer to input)
 - Gradient descent updates leave these layers' parameters virtually unchanged
- Also **exploding gradients** in deep networks
 - The parameter values rapidly change in each step of gradient descent, leading to non-convergence
- We must avoid these problems to successfully train the neural network
- One choice we make is to set the initial values of our network parameters.
- Another choice we make is the activation function itself.

Compute z, a 
Compute $\frac{\partial C}{\partial z_k^l}, \frac{\partial C}{\partial w_{kj}^l}, \frac{\partial C}{\partial b_k^l}$ 

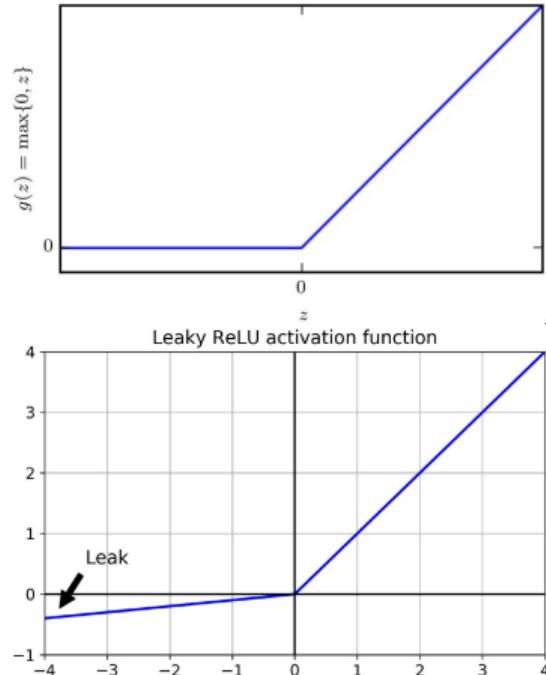
Initializing neural networks

- Want “signal” to flow well in both directions during backpropagation
 - Signal should not saturate the sigmoid activation
 - The gradients should not vanish or explode
- [Glorot, Bengio] Gradients should have equal variance before and after flowing through a layer in both directions
 - Equal variance requires $fan_{in} = fan_{out}$
- Initialize with
 - Gaussian, $\mathcal{N}(0, 1/fan_{avg})$
 - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\frac{3}{fan_{avg}}}$



Non-saturating activation functions

- Sigmoid was initially chosen as a “smooth” step
- Rectified linear unit (ReLU):
$$g(z) = \max(0, z)$$
 - Fast to compute
 - Very popular in modern networks
- “Dying ReLU”
 - Neuron dies — weighted sum of outputs is negative for all training samples
 - With a large learning rate, half the network may die!
- Leaky ReLU, $\max(\alpha z, z)$
 - “Leak” α is a hyperparameter



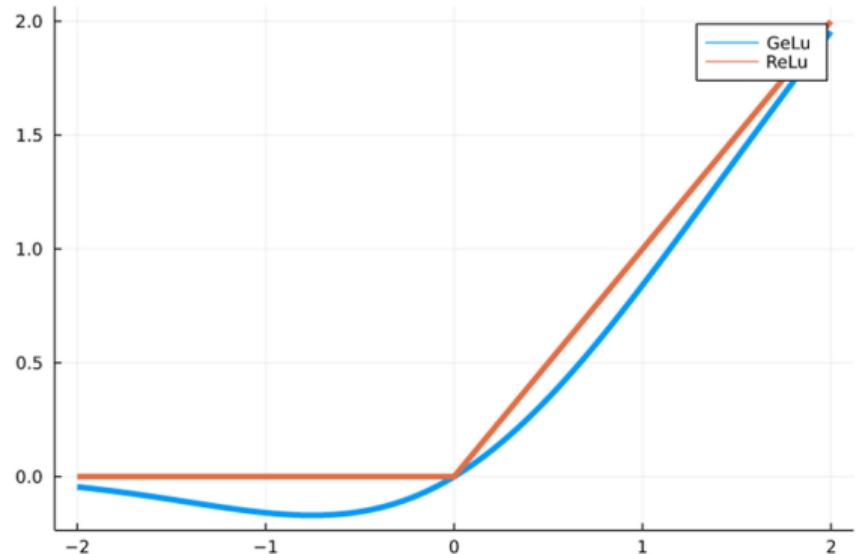
Non-saturating activation functions

- GELU — Gaussian Error Linear Unit
[Hendrycks and Gimpel, 2016]

$$\text{GELU}(x) = x \cdot P(Y \leq x)$$

P is the Probability Density Function of the Gaussian Distribution $Y \sim \mathcal{N}(0, 1)$

- Differential at zero with non-zero gradient
- Fast Implementation possible
- Used in most Transformer like GPT, BERT, ...



Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization partially mitigates vanishing/exploding gradients
- But may still recur during training
- Add batch normalization (BN) layers

- Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
- Zero-center and normalize each input

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
- Learn optimal scaling and shifting parameters for each layer

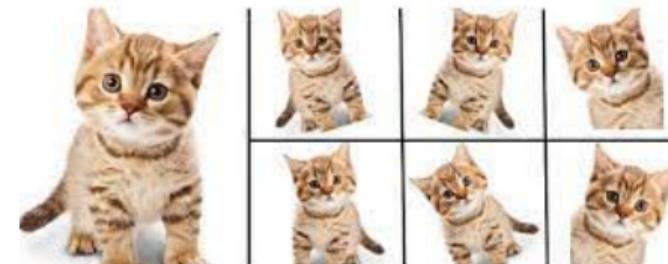
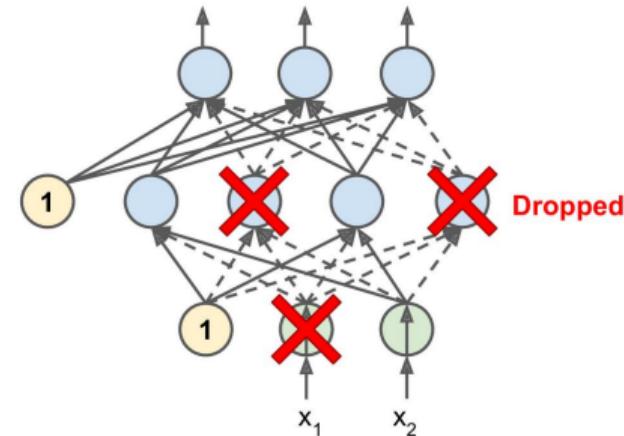
It smoothens the objective function

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization partially mitigates vanishing/exploding gradients
- But may still recur during training
- Add batch normalization (BN) layers
 - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
 - Zero-center and normalize each input
$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
 - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
 - Learn optimal scaling and shifting parameters for each layer
- At input, BN layer avoids need for standardizing
- Difficulties
 - Mean and variance differ across minibatches
 - How to estimate parameters for entire dataset?
 - Practical solution: maintain a moving average of means and standard deviations for each layer
- Batch normalization greatly speeds up learning rate
- Even works as a regularizer!

Regularization

- Recall, **Regularization** restricts the values the network parameters can take, to avoid overfitting and improve generalization to unseen data.
- Usual ℓ_1 and ℓ_2 regularization on the parameter values.
- **Dropout:** Disable nodes with probability p during training, so the network is not dependent on any particular node.
- **Data Augmentation**



Enlarge your Dataset

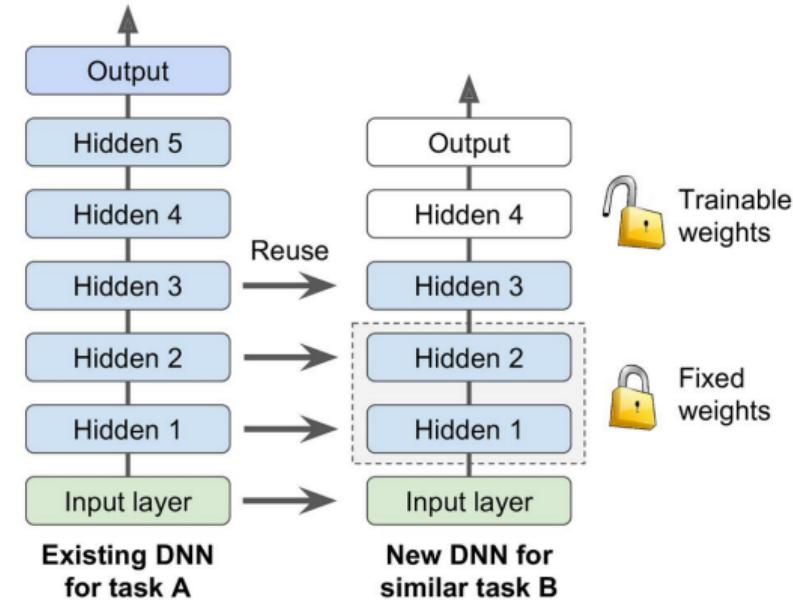
Lecture 4: Transfer Learning

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, ...)
- New DNN to classify types of vehicles
- Tasks similar, even overlapping
- Lower layers identify basic features, upper layers combine them to classify
- Freeze weights of lower layers, re-learn upper layers
- Unfreeze in stages to determine how much to reuse



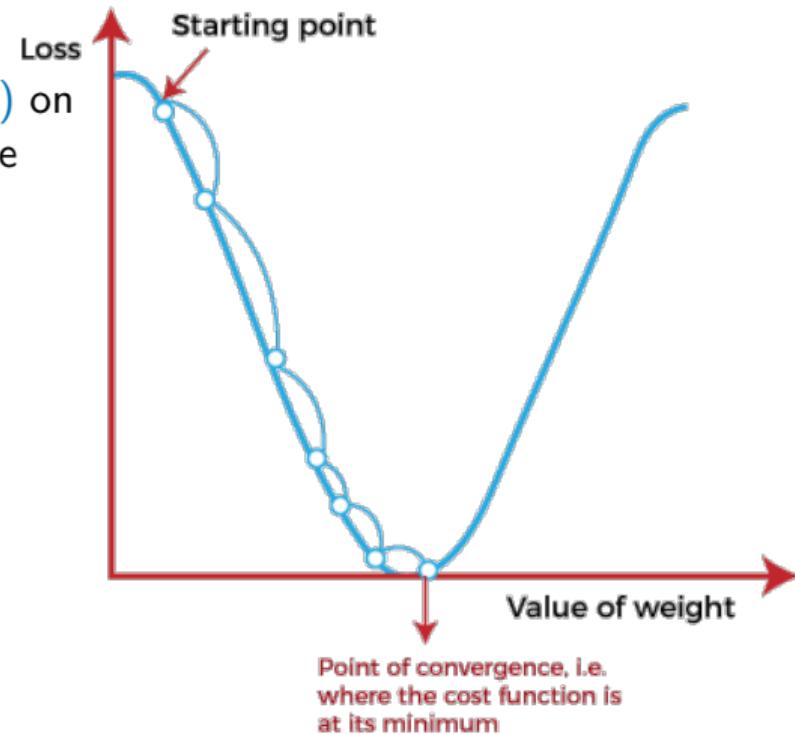
Lecture 5: Training Deep Neural Networks

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

Gradient Descent

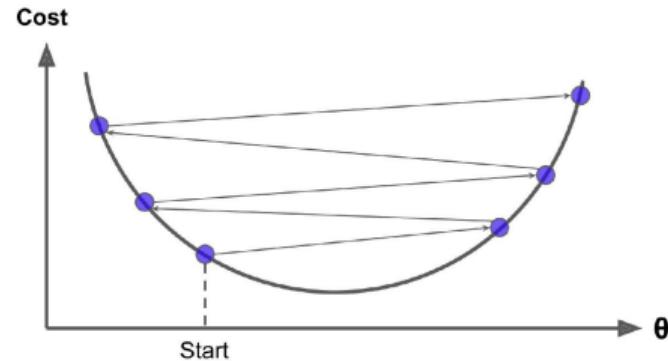
- The NN parameters θ determine the model.
- The loss-function value $J(\theta) = J(\theta, S)$ on the training set S tells us how well the model performs currently.
- Gradient $\nabla J(\theta_t) = \frac{\partial}{\partial \theta} J(\theta_t)$
- Update Parameters:
$$\theta_t \leftarrow \theta_{t-1} - \alpha \nabla J(\theta_{t-1}).$$
- α is the learning rate.



III conditioning

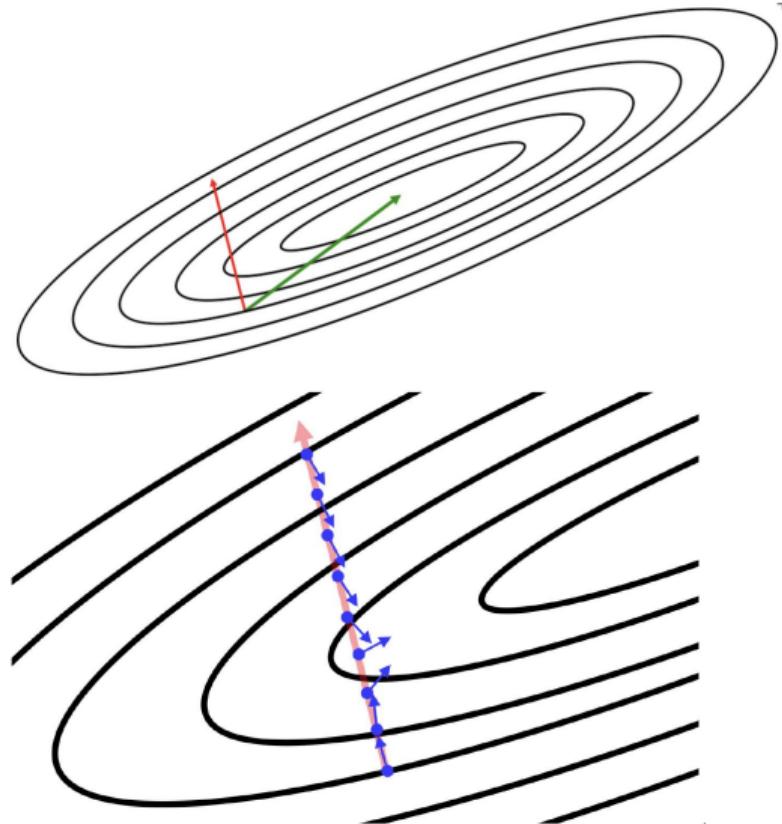
ILL :)

- III conditioning — small change in input produces a large change in output
- What is the impact of update $\theta - \alpha \nabla_{\theta}$ on cost $J(\theta)$?
- Depends on curvature, given by second derivative
Hessian: $H_{\theta} = \frac{\delta^2}{\delta \theta_i \delta \theta_j} J(\theta)$
- Using Taylor expansion, impact of update $\theta - \alpha \nabla_{\theta}$,
$$J(\theta) - \nabla_{\theta}^T \nabla_{\theta} + \frac{1}{2} \nabla_{\theta}^T H_{\theta} \nabla_{\theta}$$
- Analyze H_{θ} to check for ill conditioning



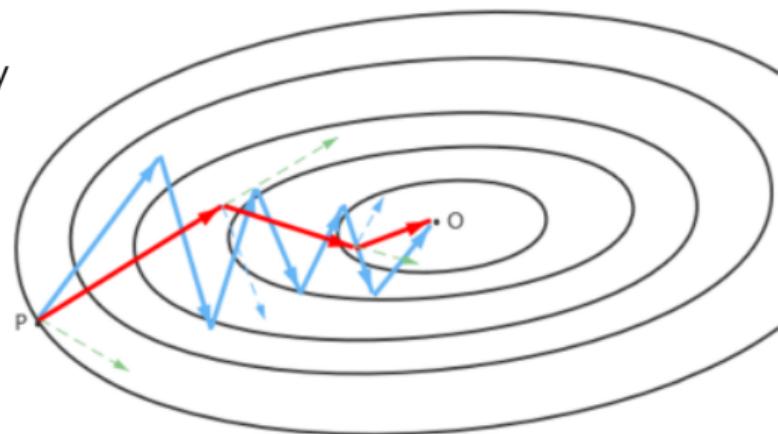
Directing gradient descent

- Locally steepest direction of descent may be far from the optimum
 - Elliptical contours vs circular contours
- Gradient changes rapidly along the direction of steepest descent
 - Taking large steps is problematic
- Ill-conditioned Hessian H — second derivatives
 - Computing Hessian is expensive
 - “Second order” methods are not used in practice
- Instead, heuristics like momentum and adaptive learning rates



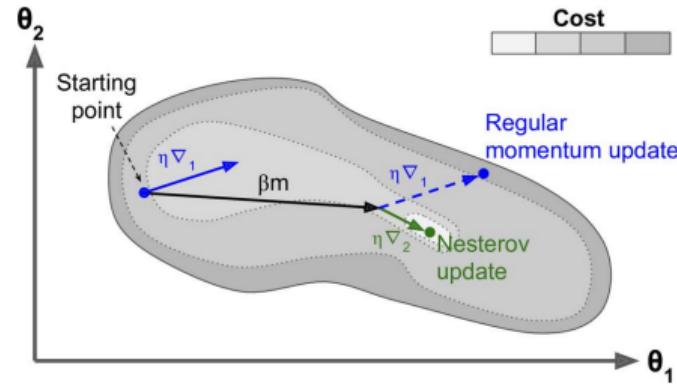
Momentum

- SGD convergence can be very slow
- Momentum in physics — mass \times velocity
- Introduce **velocity** v in SGD — assume unit mass
 - Moving average of past gradients, exponential decay
 - If gradient remains steady, velocity increases
- Update rule
 - $v_t \leftarrow \epsilon v_{t-1} - \alpha \nabla J(\theta_{t-1}) ; v_0 = 0$
 - $\theta_t \leftarrow \theta_{t-1} + v_t$
- Hyperparameter $\epsilon \in [0, 1)$ — friction exponentially decaying history



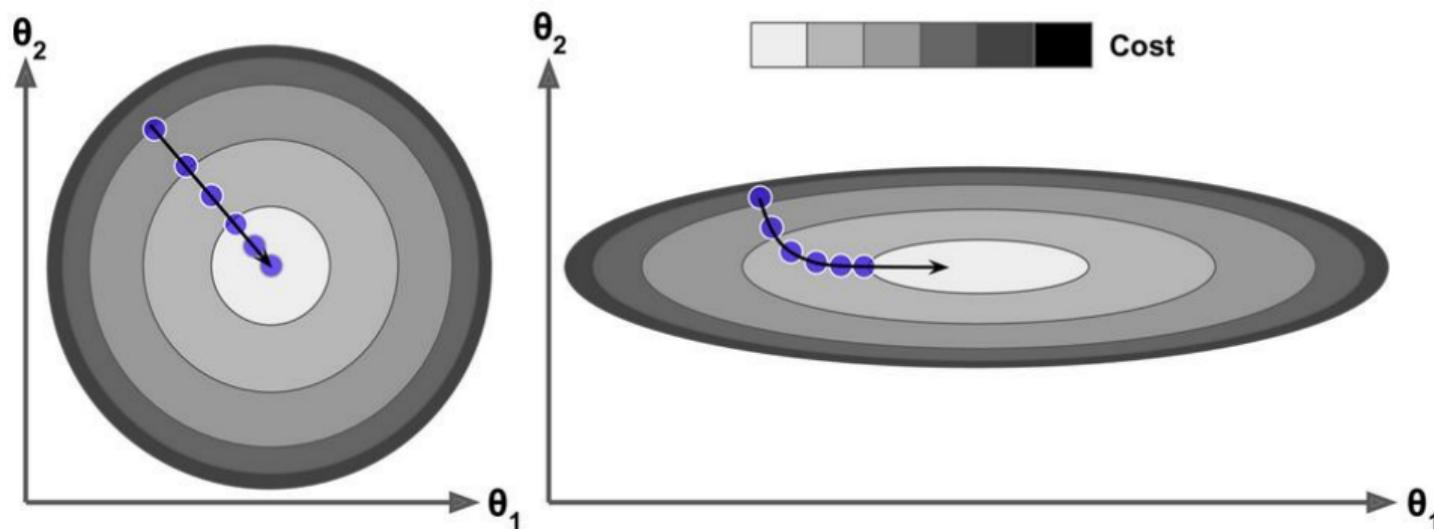
Nesterov momentum optimization

- Measure cost function slightly ahead, in direction of momentum
- Update rule
 - $v_t \leftarrow \epsilon v_{t-1} - \alpha \nabla J(\theta_{t-1} + \epsilon v_{t-1})$
 - $\theta_t \leftarrow \theta_{t-1} + v_t$
- Look before you leap!
- Stronger Convergence Guarantees.



Adjusting the trajectory

- If features have different scales, gradient descent is steeper in some dimensions
- How can we correct for this?

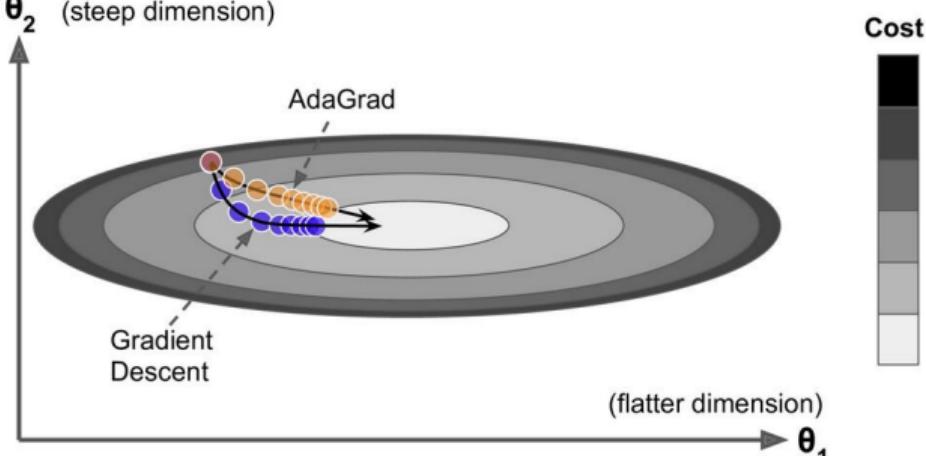


Adagrad

- Adagrad update rule

- $g_{t-1} \leftarrow \nabla J(\theta_{t-1})$
- $r_t \leftarrow r_{t-1} + g_{t-1} \cdot g_{t-1}$: $\gamma_0 = \theta_2$ (steep dimension)
- $\Delta\theta \leftarrow \frac{\alpha}{\delta + \sqrt{r_t}} \cdot g_{t-1}$, where
 $\delta \approx 10^{-7}$, for numerical stability
- $\theta_t \leftarrow \theta_{t-1} - \Delta\theta$

- Shrink learning rate in each dimension according to entire history of the gradient



Adaptive learning rates

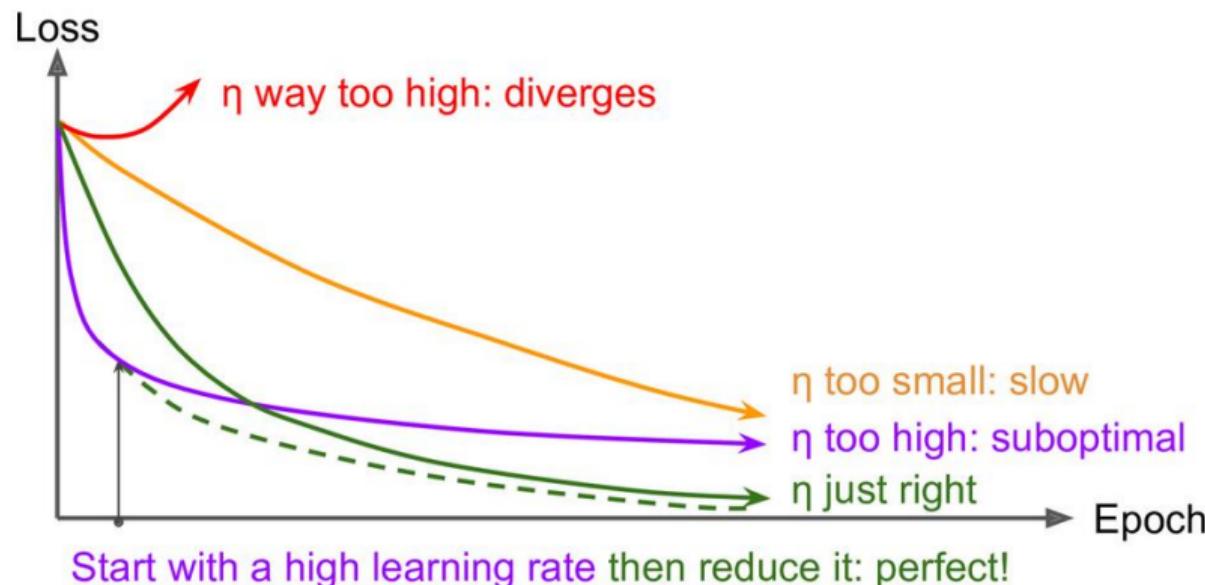
RMSProp

- Using entire history shrinks learning rate too much
- Exponentially decaying average, discard extreme past
- Update rule
 - $g_{t-1} \leftarrow \nabla J(\theta_{t-1})$
 - $r_t \leftarrow \rho r_{t-1} + (1 - \rho)(g_{t-1} \cdot g_{t-1})$, where ρ is decay rate (*Small : $10^{-5}, 10^{-6}$*)
 - $\Delta\theta \leftarrow \frac{\alpha}{\sqrt{\delta + r_t}} \cdot g_{t-1}$, where $\delta \approx 10^{-6}$
 - $\theta_t \leftarrow \theta_{t-1} - \Delta\theta$
- New hyperparameter ρ

- Adaptive moments — combines RMSProp and momentum
- v and r are the first and second moments of θ
- Update rule
 - Two decay rates ρ_1, ρ_2 ; two moments, $v_t = r_t = 0$ at time step $t = 0$
 - $g_{t-1} \leftarrow \nabla J(\theta_{t-1})$
 - $v_t \leftarrow \rho_1 v_{t-1} + (1 - \rho_1) g_{t-1}$ (momentum)
 - $r_t \leftarrow \rho_2 r_{t-1} + (1 - \rho_2) (g_{t-1} \cdot g_{t-1})$ (RMSProp)
 - Correct bias in first and second moments: $\hat{v} \leftarrow \frac{v}{1 - \rho_1^t}, \hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$
 - $\Delta\theta = \frac{\alpha}{\sqrt{\hat{r}} + \delta} \hat{v}_t$
 - $\theta_t \leftarrow \theta_{t-1} - \Delta\theta$
- Most popular choice of optimizer

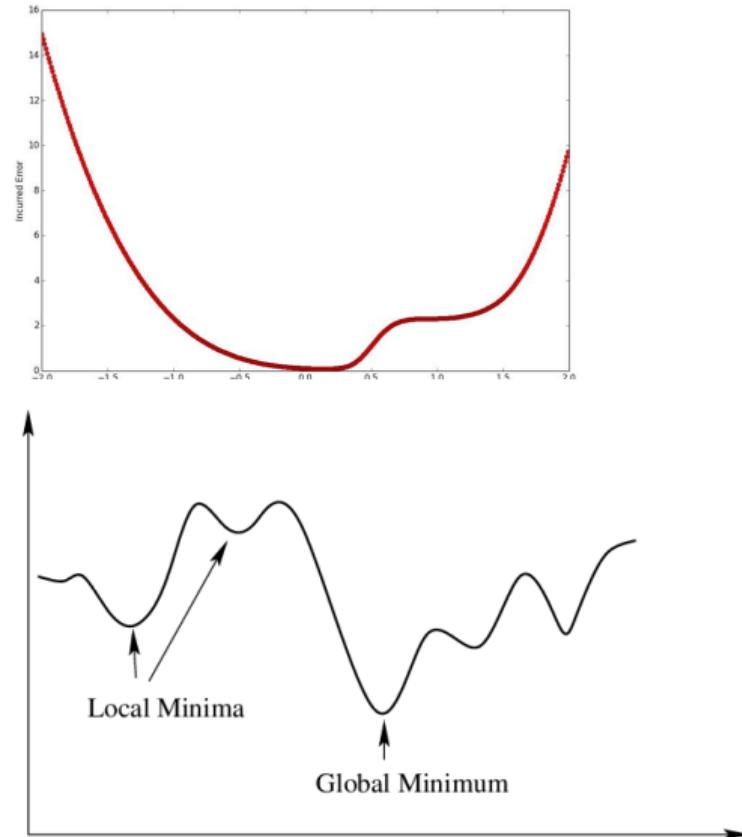
Learning rate schedulers

- Choosing a good fixed learning rate is hard
- Adjust learning rates as a function of number of epochs.
- Power scheduling, exponential scheduling, piecewise constant scheduling



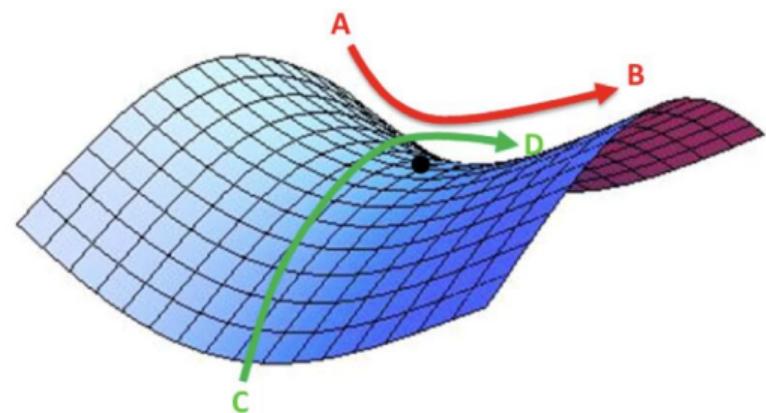
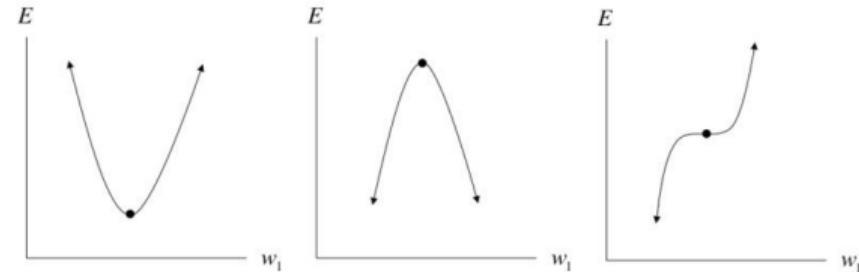
Loss Landscape and Local minima

- Gradient Descent assumes that the Loss function is Convex. Then it can converge to the global minima
- Loss function for neural networks is not convex.
- So gradient descent only finds a local minima



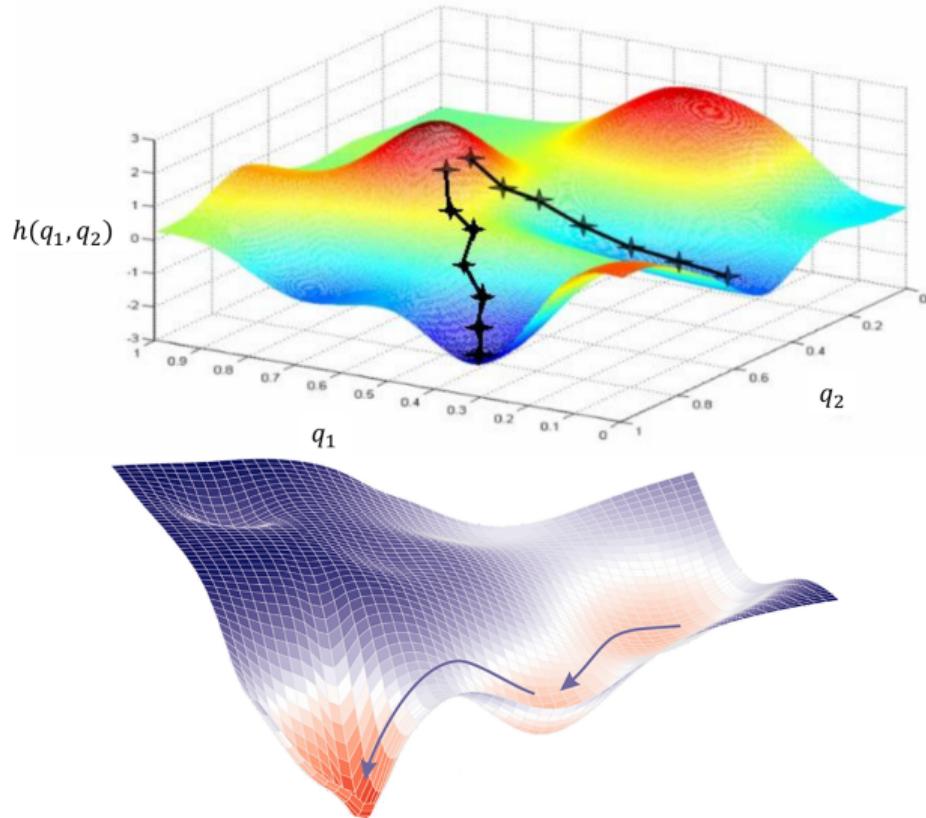
Loss Landscape: Saddle points

- Critical points — zero gradient
 - Minimum, maximum or inflection point
 - k critical points $\rightarrow k/3$ are minima
- In d dimensions
 - Should be minimum in d directions
 - k critical points $\rightarrow k/3^d$ are minima
- Large fraction of critical values are saddle points
- Does not seem to be a problem for SGD
- Solving directly for zero gradient is problematic



Loss Landscape

- In practice, Adam etc do very well even with non-convex loss functions
- The choice of the initial parameter values can lead to different local minima
- Adam et.al. seem to avoid “shallow” local minima, and converge to “deep” local minima.
- “Deep” local minima well approximate the global minima
→ real-world success of DNN!



Lecture 6: Hands On

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

<https://www.cmi.ac.in/~pranabendu/aml23/Lec06-DNN-MNIST.html>

<https://www.cmi.ac.in/~pranabendu/aml23/Lec06-DNN-CIFAR10.html>

Lecture 7: Convolutional Neural Networks

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

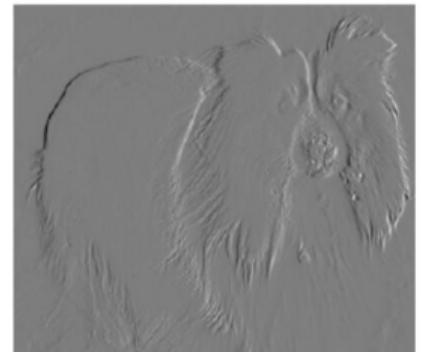
Deep Neural Networks for Recognizing Images

Last Lecture:

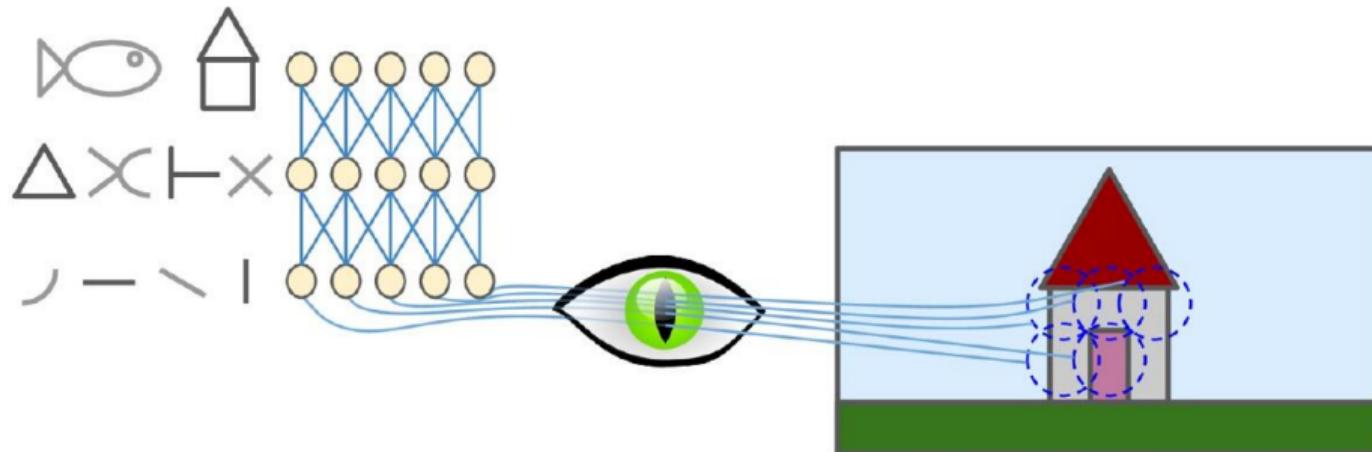
- Fully-Connected DNN for CIFAR-10
- Performed OK ($\sim 50\%$ Test Accuracy)
- However, we also saw that the Fully-Connected Network doesn't use *Visual Information*.
 - We scrambled all training and test images with a fixed scrambler function.
 - The resulting images were no longer recognizable as digits (by us humans).
 - We trained the network with scrambled train images
 - The Fully-Connected network still managed a 50% Test accuracy on the scrambled test images
 - This means that the network was not using visual information.
- Can we do better by using some visual information in the images?
- How can this be done?

How the brain recognizes images

- Visual cortex processes images
- Experiments on cats and monkeys [Hubel, Wiesel 1959], Nobel Prize 1981
- Visual cortex organized in layers
 - Each layer detects **features**
 - Initial layers detect simple features — edges
 - Later layers combine features of earlier layers — detect contours, shapes, entire object
- Convolutional neural network (CNN) — layered network



Receptive field



- Each neuron focuses on a small region — **receptive field**
- FC-NN reads entire image as input
 - MNIST — 28×28 pixels b/w
 - CIFAR10 — 32×32 pixels color

- Imagenet, 224×224
 - Three colours — $224 \times 224 \times 3$ inputs
 - Each neuron in first layer has $> 120,000$ input weights
 - Multiple such neurons
- Issues: Parameter blowup, overfitting

Filters and convolution

- Aggregate values over a region
 - Smoothening — take average
 - Vertical lines — difference between adjacent columns
 - Horizontal lines — difference between adjacent rows
- Pass a **filter f** over the image
 - Convolution — $I * f$
 - Sometimes, filter is called a **convolution kernel** — $I * K$
- Light to dark vertical edges
- Dark to light vertical edges

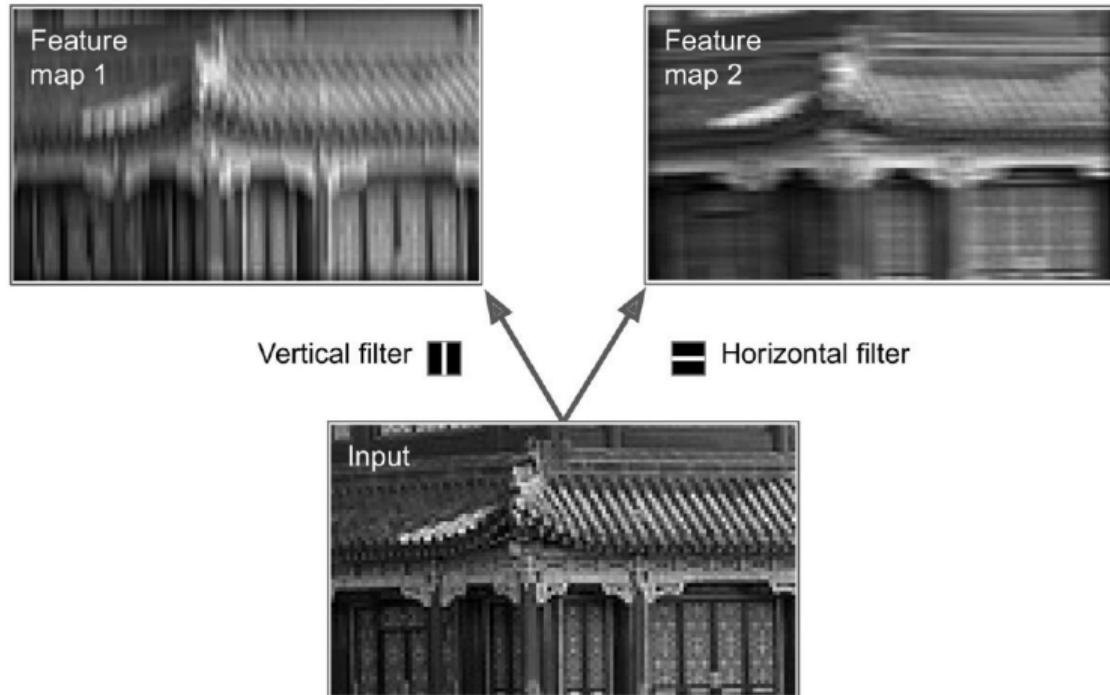
$$\begin{matrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{matrix} \quad * \quad \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix} = \begin{matrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{matrix} \quad * \quad \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix} = \begin{matrix} 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \end{matrix}$$

The diagrams illustrate the convolution process. In the top example, the input image consists of a 6x6 grid of 10s, and the filter is a 3x3 kernel with values 1, 0, -1. The result is a 4x4 output matrix where every element is 30. In the bottom example, the input image has 0s in the first two columns and 10s in the last four columns, and the same filter is applied. The result is a 4x4 output matrix where every element is -30. The filter is shown as a 3x3 grid of 1, 0, -1, and the input is shown as a 6x6 grid of 10s or 0s. The output is shown as a 4x4 grid of 30s or -30s.

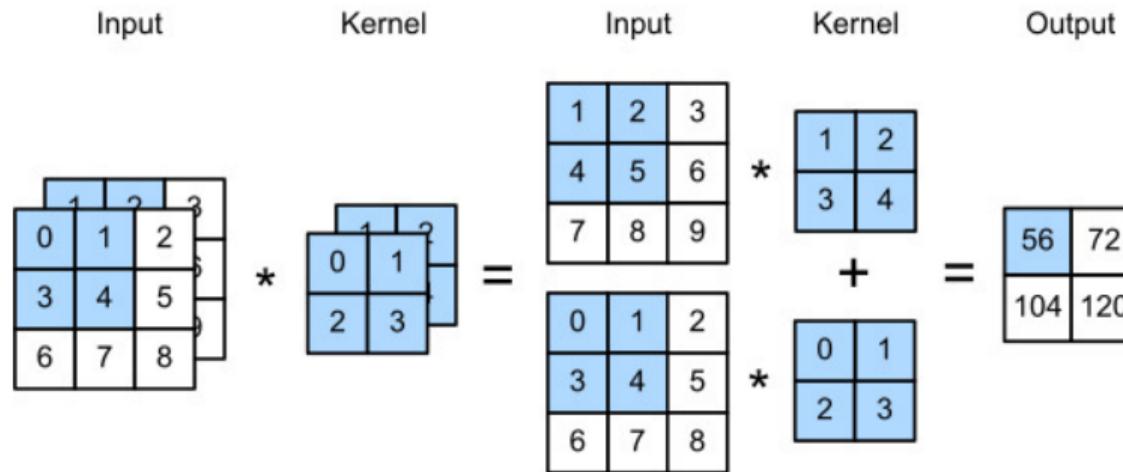
Feature maps

- Filters produce feature maps



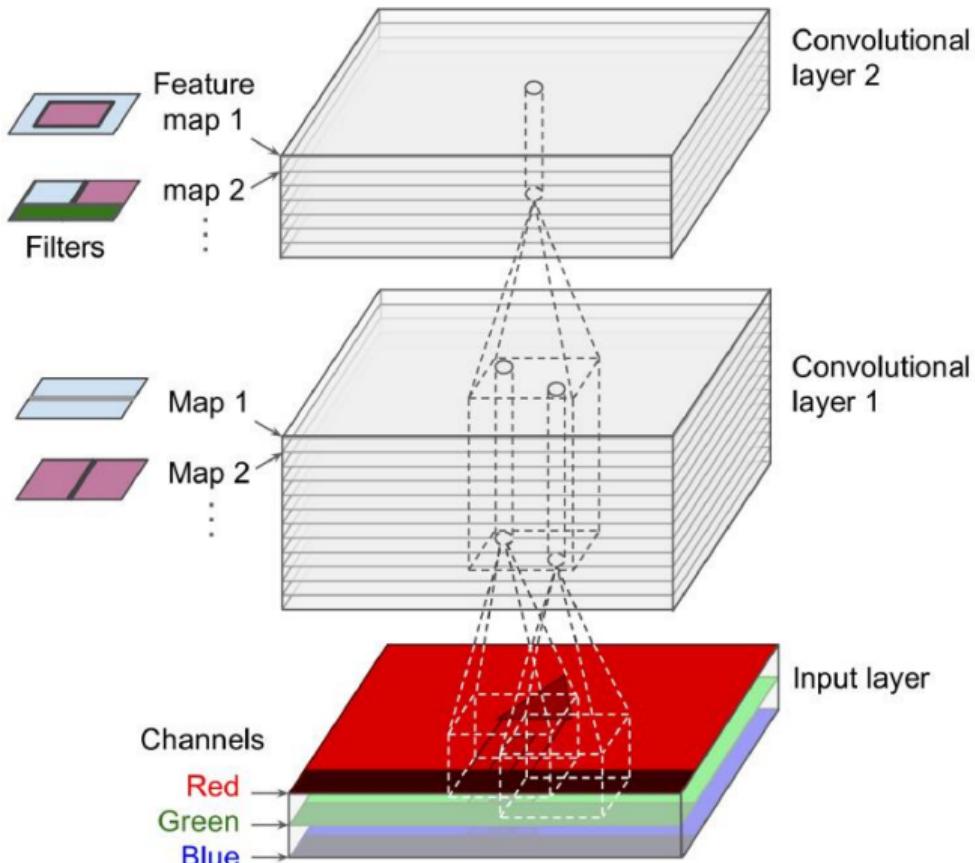
Multi-Channel Inputs

- Real World Images are in color (3-channels)
- Similarly convolution filters in higher layer will need to work with feature maps produced by multiple lower layer convolutions.
- So we need Multi-Channel Convolution



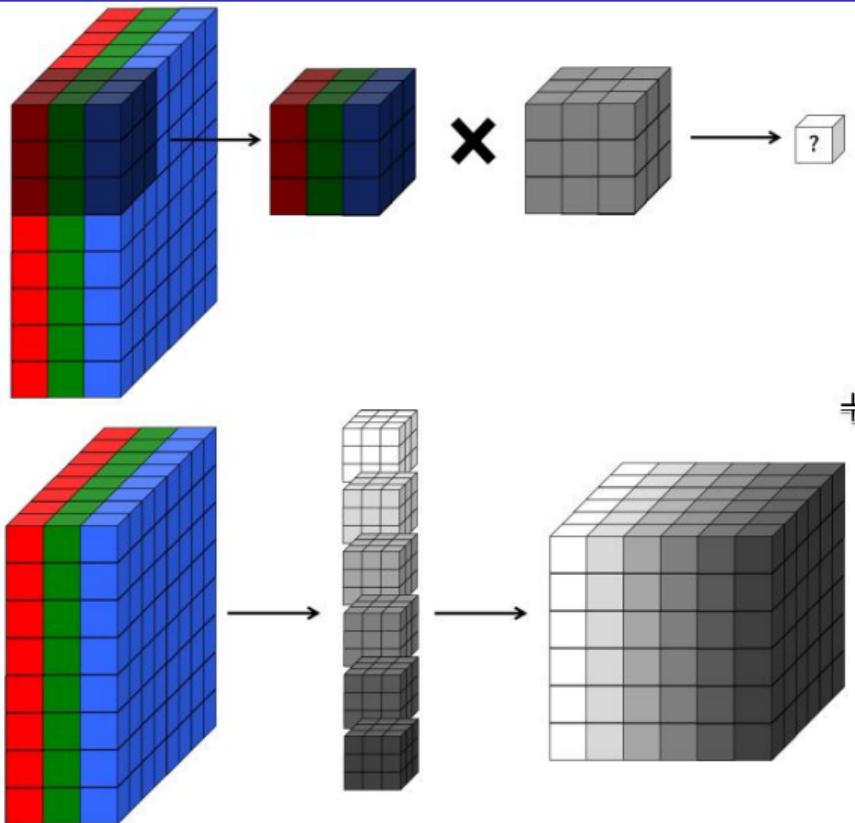
Feature maps

- Filters produce feature maps
- Colour images are split by **channel**
- Each layer has many feature maps
 - Array of filters, each connected to a different region
- Higher layers combine features discovered by lower layers



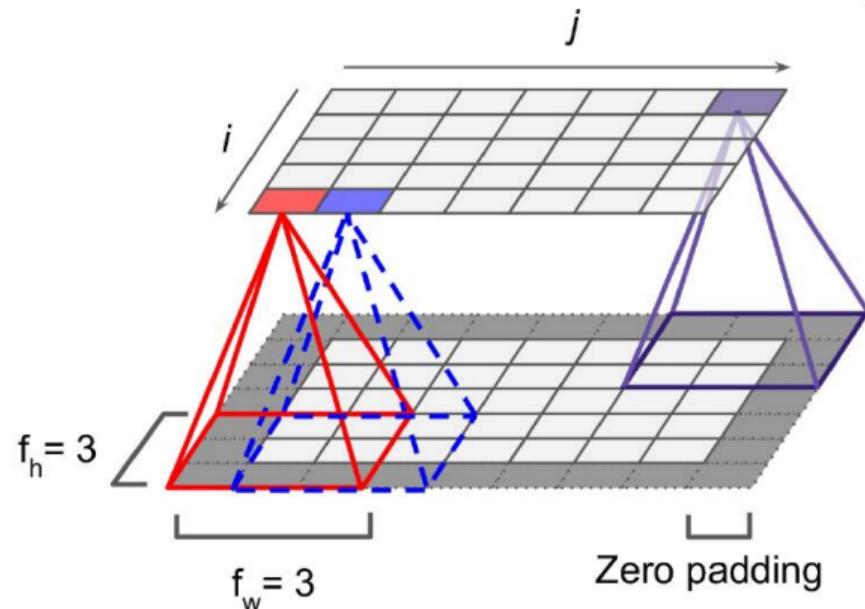
Volumetric view

- Each filter processes a **volume** of inputs
- Each layer has sublayers
 - A sublayer is an array of such filters
- Each layer produces a block of outputs



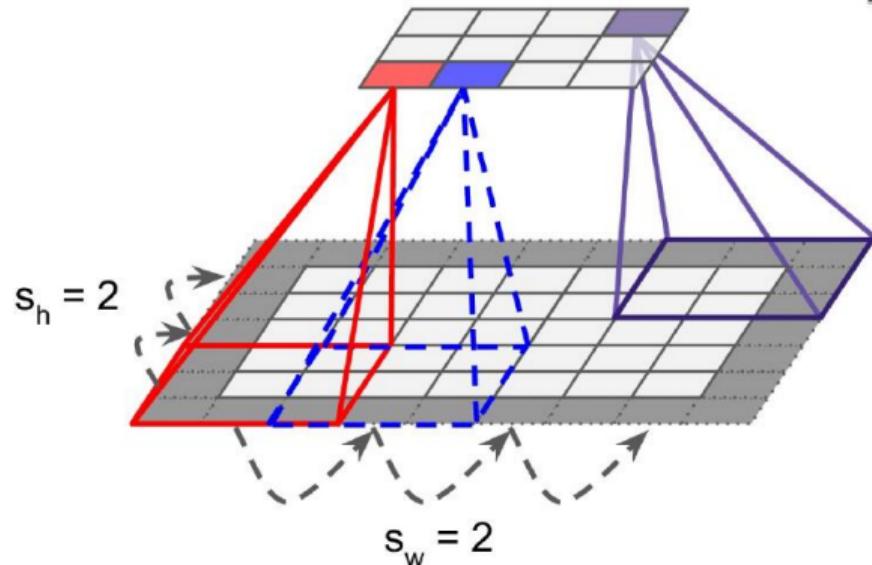
Zero padding, stride

- Each filter f has height f_h , width f_w
 - Receptive field of f
- Need to extend the boundary for filter to work properly at the edges
 - Zero padding
- With padding, feature map has same dimension as input
- Note: In CNNs, filters are learnt during training.
 - We only specify f_h and f_w , but weights are learned from training data



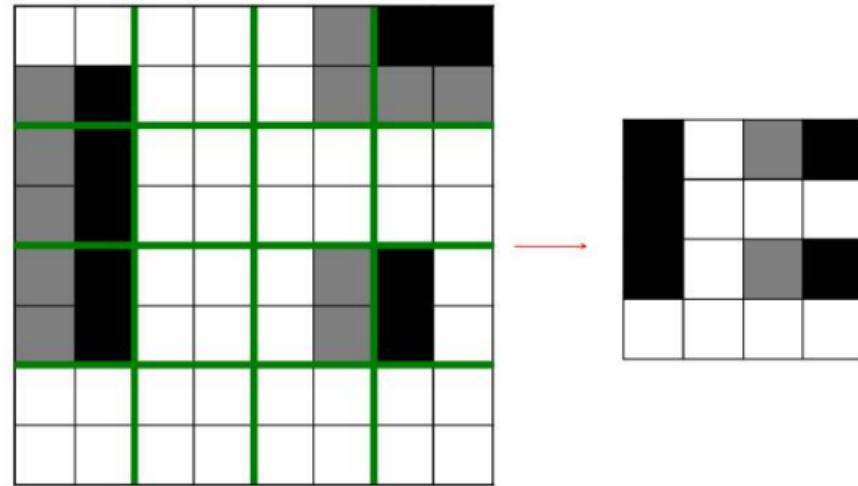
Zero padding, stride

- Each filter f has height f_h , width f_w
 - Receptive field of f
- Need to extend the boundary for filter to work properly at the edges
 - Zero padding
- With padding, feature map has same dimension as input
- To reduce dimension, we can space out the receptive fields
 - Horizontal and vertical **stride**



Pooling

- Filters process overlapping regions
- **Pooling** processes partitions
 - Subsampling, reduce dimensionality
- Most common is **max-pool** over 2×2 window



Pooling

- Filters process overlapping regions
- Pooling processes partitions
 - Subsampling, reduce dimensionality
- Most common is max-pool over 2×2 window
- Here, max-pooling reduces an image to half its size

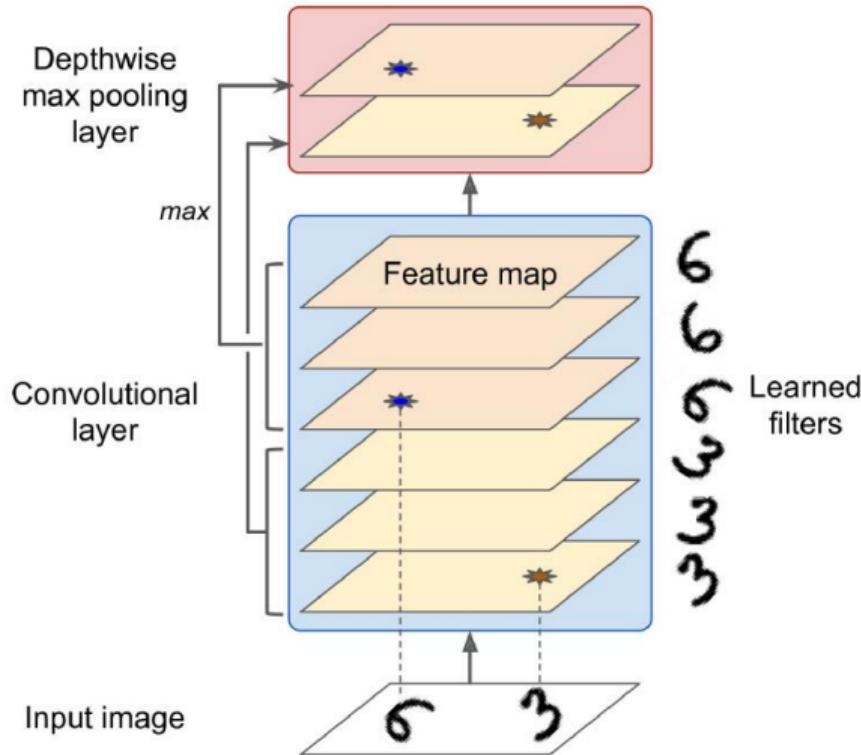
$$\text{Output Size} = \left\lceil \frac{n_{in} + 2p - K}{s} \right\rceil + 1$$

*K: kernel size n_{in}: number of input features
P: padding size S: stride*



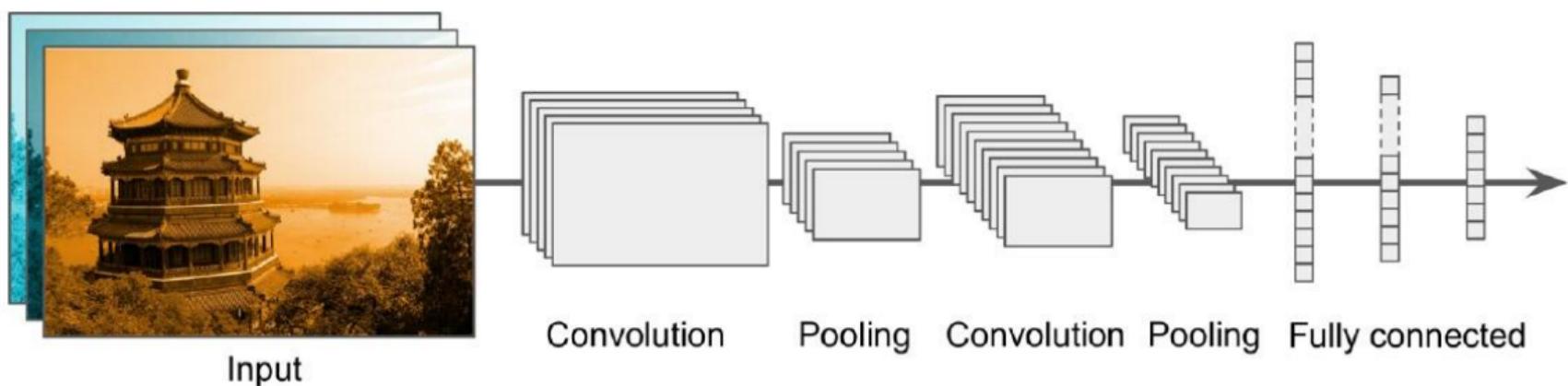
Pooling

- Filters process overlapping regions
- **Pooling** processes partitions
 - Subsampling, reduce dimensionality
- Most common is **max-pool** over 2×2 window
- Here, max-pooling reduces an image to half its size
- Can also pool depthwise — for instance, to learn features invariant to rotation



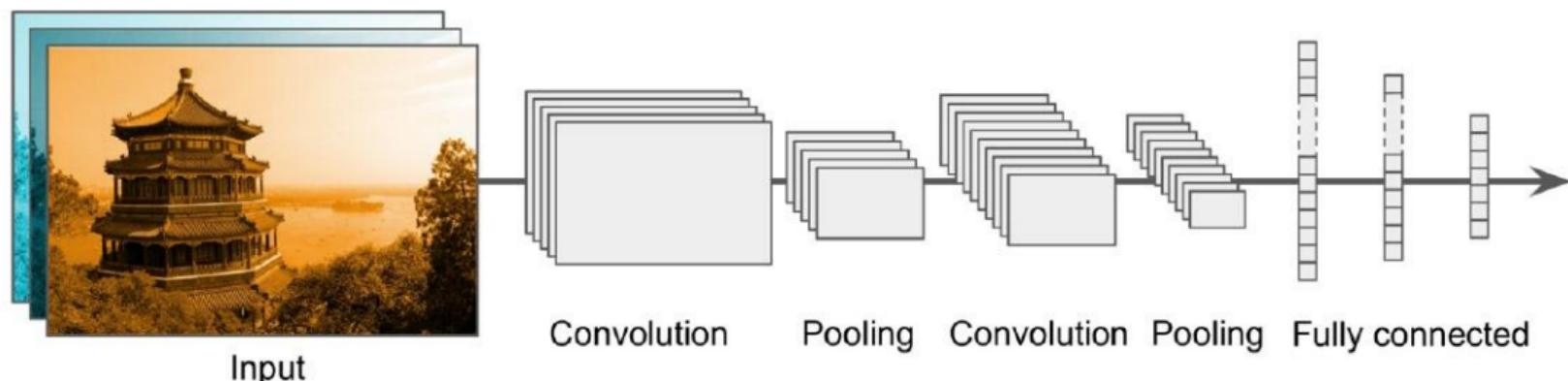
Typical CNN Architecture

- A typical CNN has multiple iterations of convolution followed by pooling
- After final pooling, conventional completely connected network



Parameter sharing

- A filter is a layer of identical nodes operating on different regions (receptive fields)
- All these nodes should behave the same
- While training, their weights are tied to each other — parameter sharing
- Thus, backward pass of backpropagation calculation is reduced
- Forward pass needs to compute individual outputs — still expensive

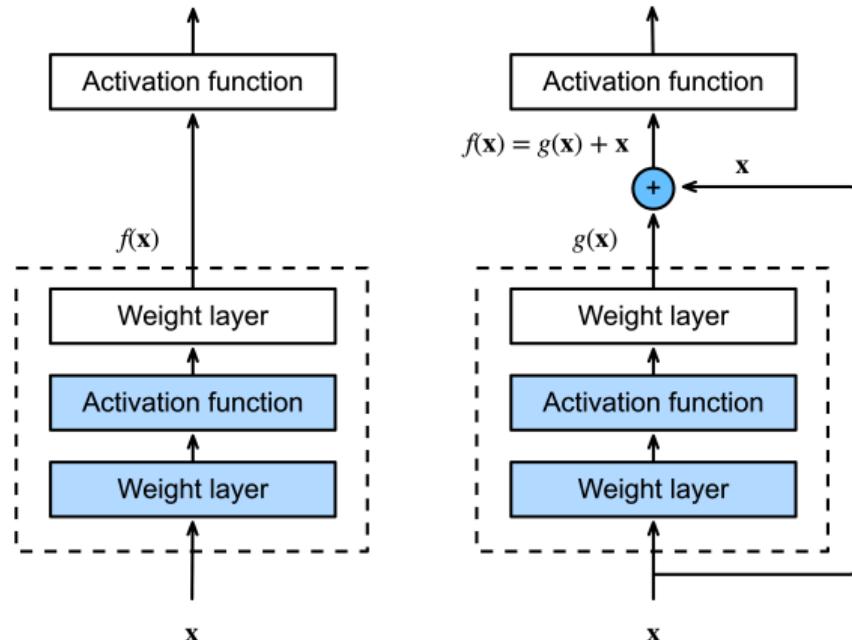


CNNs through the ages

- **LeNet-5** — Yann LeCun, 1998
 - Handwritten digits, MNIST data set from US postal service
- **AlexNet** — Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton, 2012
 - ImageNet, 14 million images, 20,000 categories, hand annotated
 - Top-five error rate — at least one of top 5 predicted labels is correct
 - 2012 ImageNet challenge, AlexNet reduced top-five error rate from 26% to 17%
 - First to add multiple convolution layers between pooling layers
- **GoogLeNet** — Christian Szegedy et al, 2014
 - 2014 ImageNet challenge, reduced top-five error rate to 7%
 - **Inception layer** with 1×1 filters, operates in depth dimension, **cross-channel features**

CNNs through the ages

- ResNet — Kaiming He et al, 2015 152 layers!
 - 2014 ImageNet challenge, reduced top-five error rate to under 3.6%
 - Residual Networks that use Skip Connections
 - Learn $g(x) = f(x) - x$ rather than $f(x)$
 - This mitigates vanishing or exploding gradients.

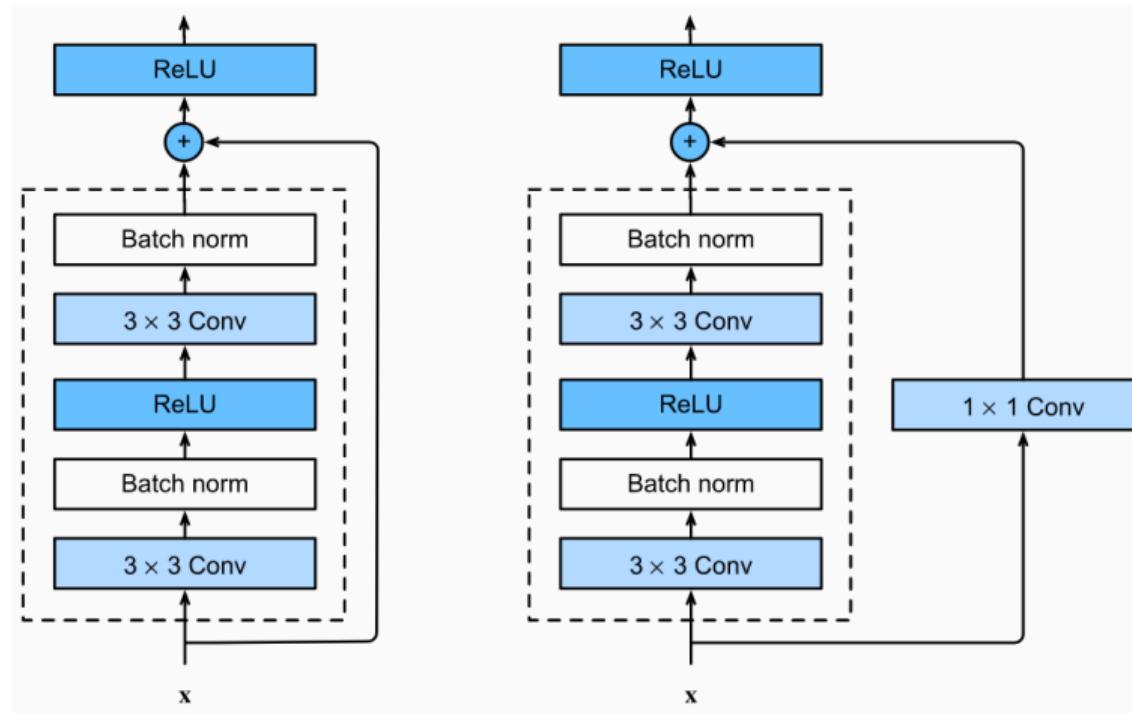


- Vision Transformers

RESNET

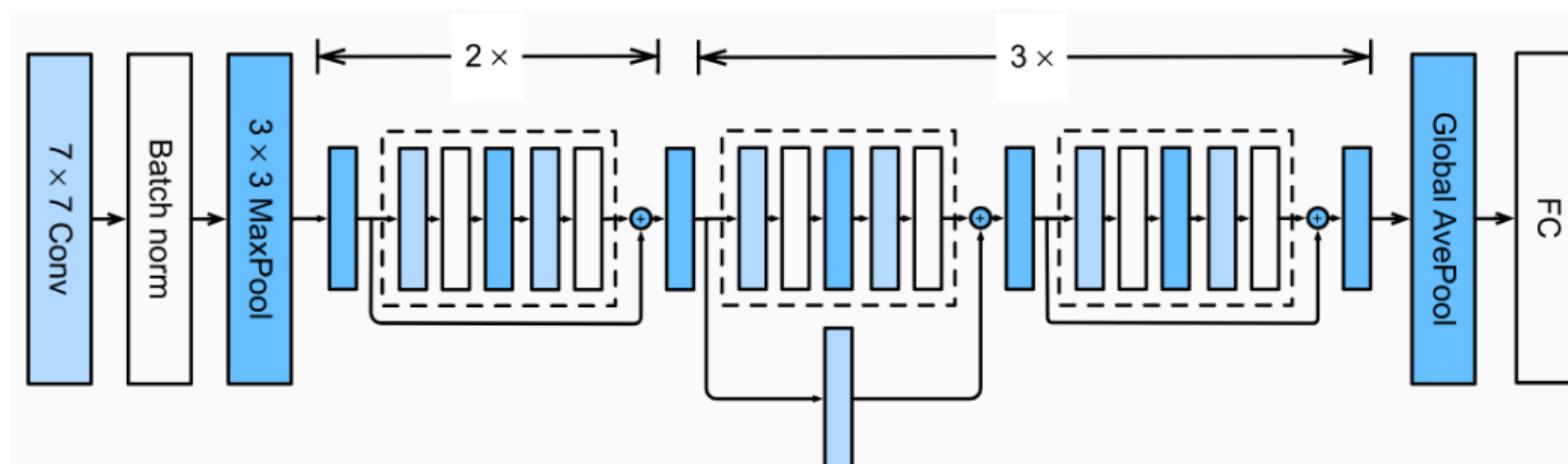
■ Residual Networks that use Skip Connections

- Learn $g(x) = f(x) - x$ rather than $f(x)$
- Made of Blocks
- Very Deep Networks with Skip Connections



- Residual Networks that use Skip Connections

- Learn $g(x) = f(x) - x$ rather than $f(x)$
- Made of Blocks
- Very Deep Networks with Skip Connections



Summary

- CNN architecture mimics the visual cortex
 - Processing is done in layers
 - Filters aggregate information across a small receptive field to capture features
 - Higher layers combine features at lower levels
- Pooling layers use subsampling for dimension reduction
- Within a filter layer, parameter sharing reduces number of parameters to be learned
- AlexNet signalled resurgence of CNNs with huge margin of victory in ImageNet 2012
- CNNs continue to evolve with specialized hacks

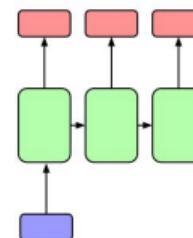
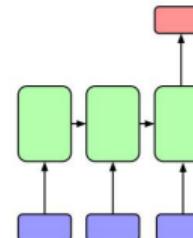
Recurrent Neural Networks

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2023

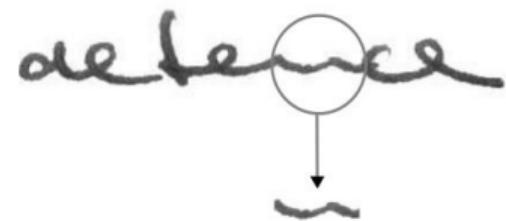
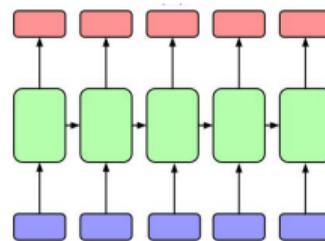
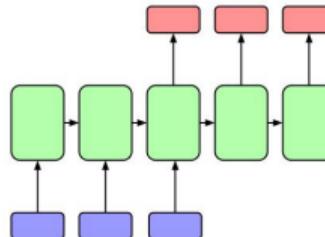
Dealing with sequences

- Conventional neural networks map single inputs to single outputs
 - Each input/output may be a vector of values
- These are *Feed Forward Networks*.
- Some classification tasks require mapping a sequence of inputs to an output
 - Identifying a music or video clip
- Others require mapping a single input to a sequence of outputs
 - Generating a caption for an image

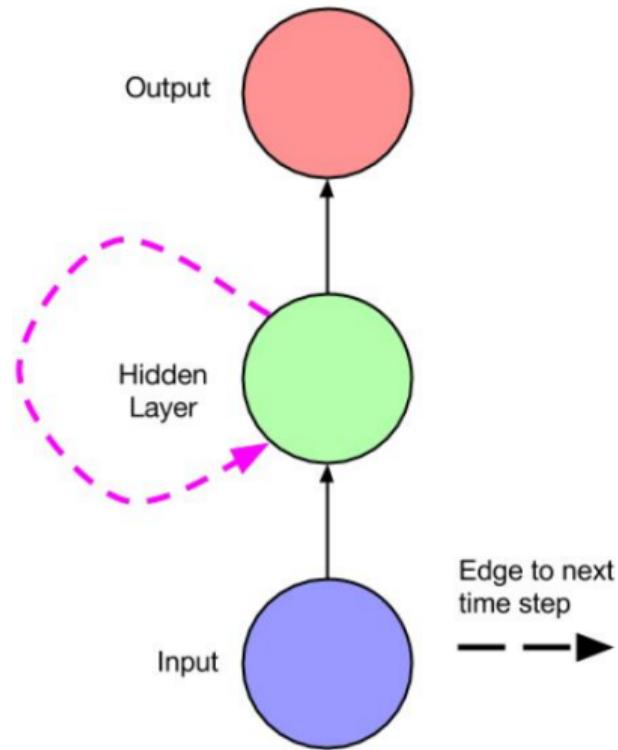


Dealing with sequences

- Mapping sequences to sequences
 - Language translation — read an entire input sentence, then generate output
- Mapping sequences to sequences on the fly
 - Predict the next word in a sentence
- Context is important
 - The handwritten word is clearly **defence**
 - The **n** in isolation is illegible

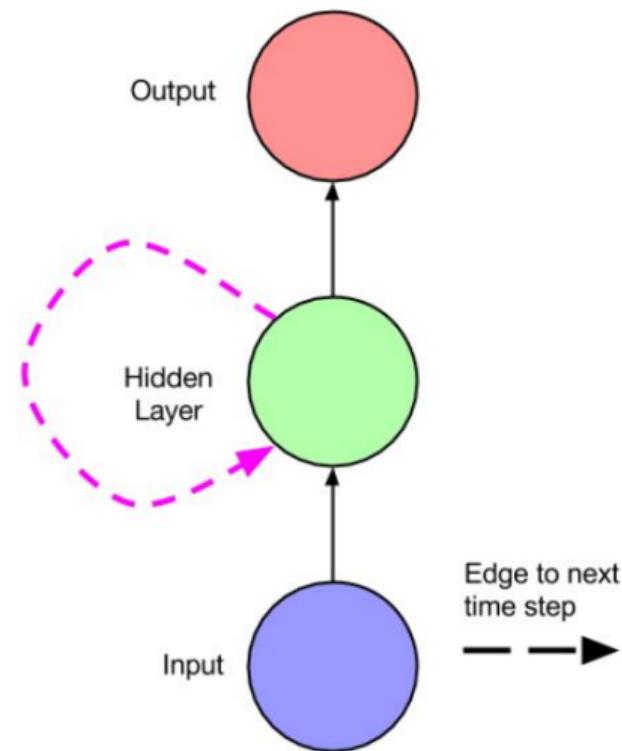


Idea: Lets also remember the past!

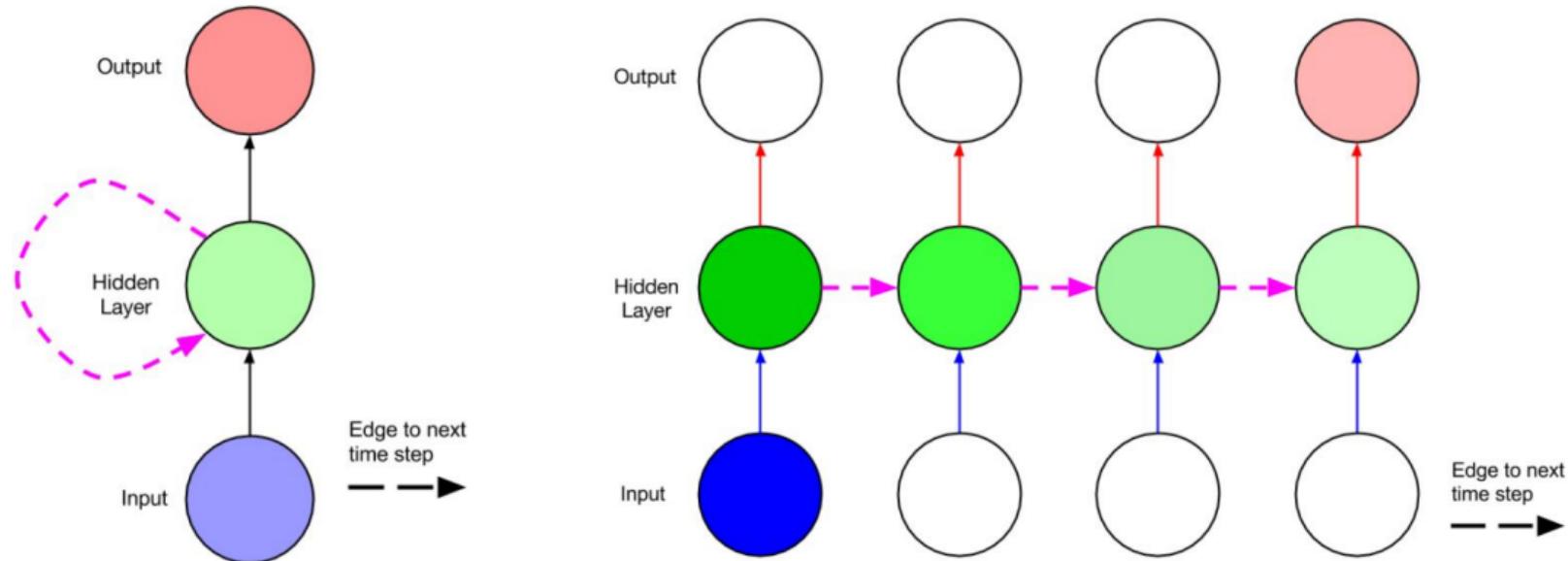


Incorporating memory

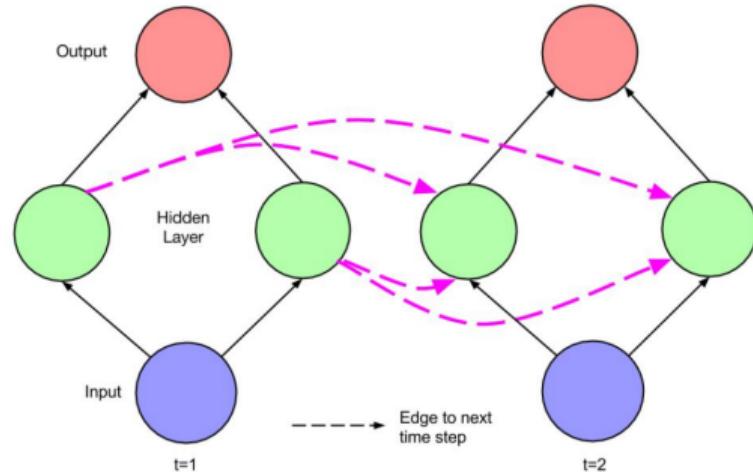
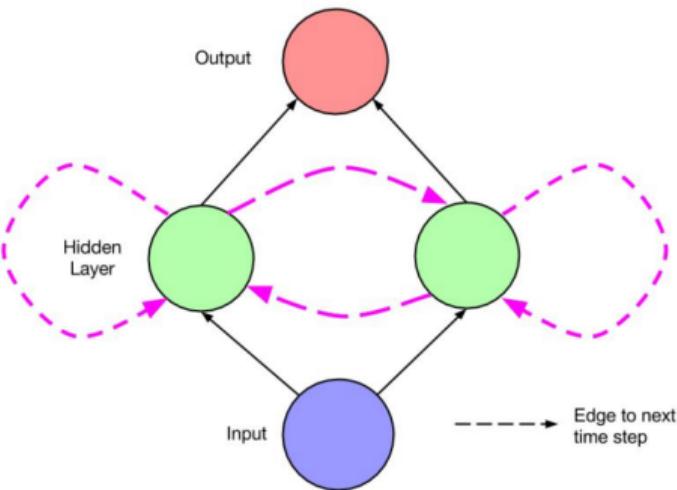
- Input sequence $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
- Output sequence $\hat{y}^{(1)}, \hat{y}^{(2)}, \dots, \hat{y}^{(t)}$
- Allow $\hat{y}^{(t)}$ to also depend on previous inputs $x^{(1)}, x^{(2)}, \dots, x^{(t-1)}$
- Hidden state : $h^{(t)}$
 - $h^{(t)}$ depends on current input and previous state
 - $$h^{(t)} = f(W^{hx}x^{(t)} + W^{hh}h^{(t-1)} + b_h)$$
- Output is a function of the current state
 - $$y^{(t)} = g(W^{yh}h^{(t)} + b_y)$$



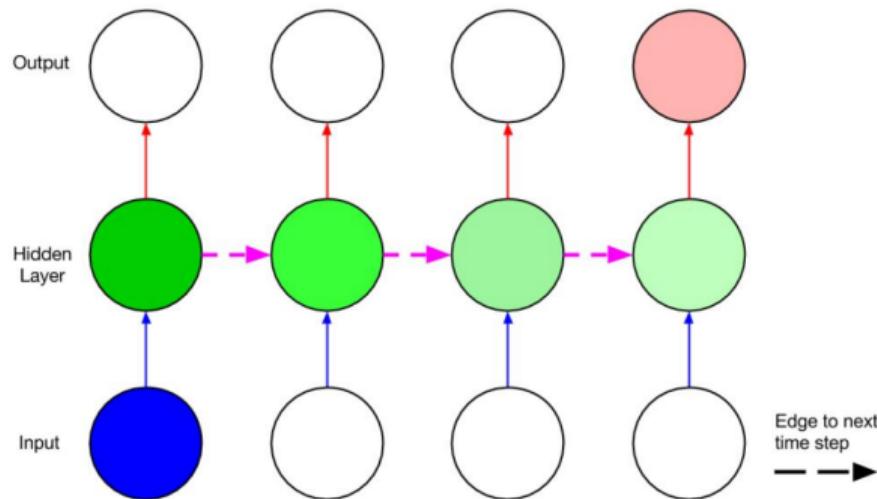
Time unrolling



Time unrolling

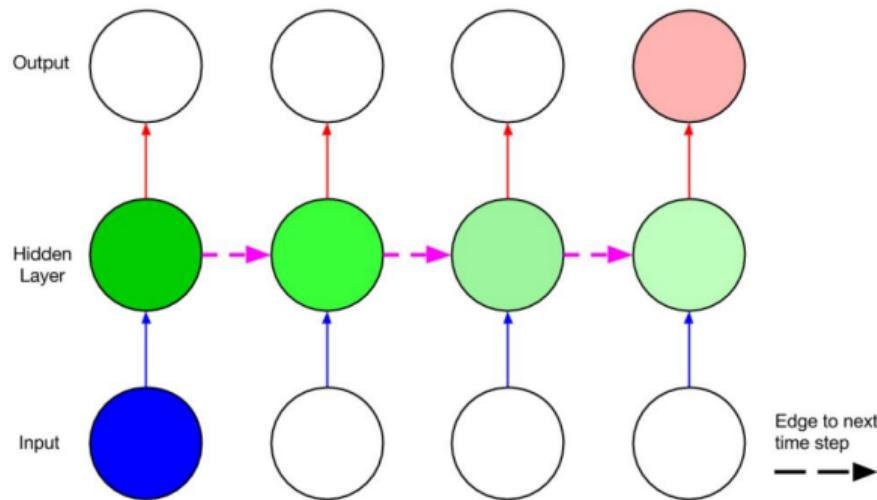


Time Unrolling and Back Propagation Through Time



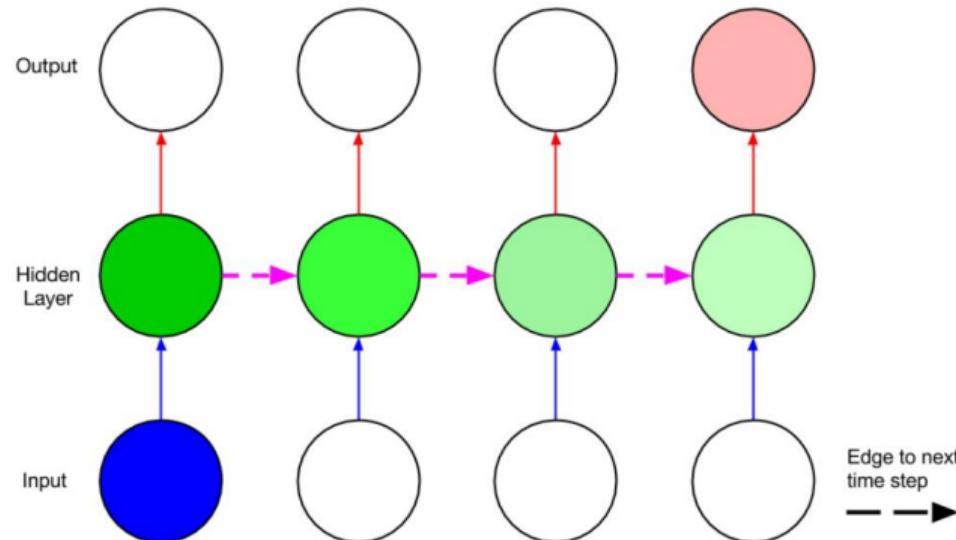
- Time Unrolling makes it a (larger) Feed-Forward Network
- but all *copies* share the parameters, so number of parameters doesn't increase.
- So we can do back-propagation to update the weights

Vanishing and Exploding Gradients gradients



- Unfortunately, we end-up with a very deep network
- Only the most recent parts of the input sequence are remembered; earlier parts are forgotten
- Back-Propagation suffers from vanishing or exploding gradients when unrolled over many time steps.

Vanishing and Exploding Gradients gradients



- Also, can't unroll infinitely far into the future.
- Truncated BPTT is what is done in practice, and also addresses these issues to an extent.
- Unfortunately, long-term context is lost. *Back propagation through time*

Problem: We are unable to remember important parts of information far into the past.

Why this happens:

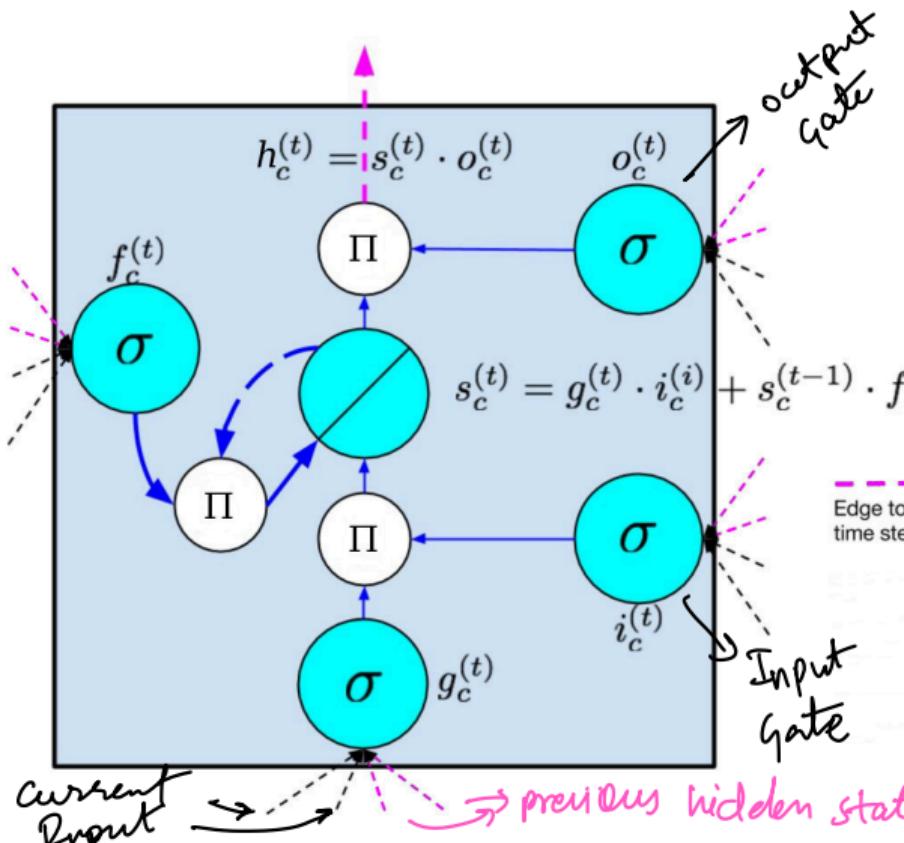
- Every piece x_i of the input sequence is treated in the same manner by the RNN.
- So important and non-important pieces both modify the hidden state, causing important pieces of the input sequences further in the past to be forgotten.

Problem: We are unable to remember important parts of information far into the past.

Idea: A mechanism that learns to distinguish between important and not-important parts, and remembers the important parts.

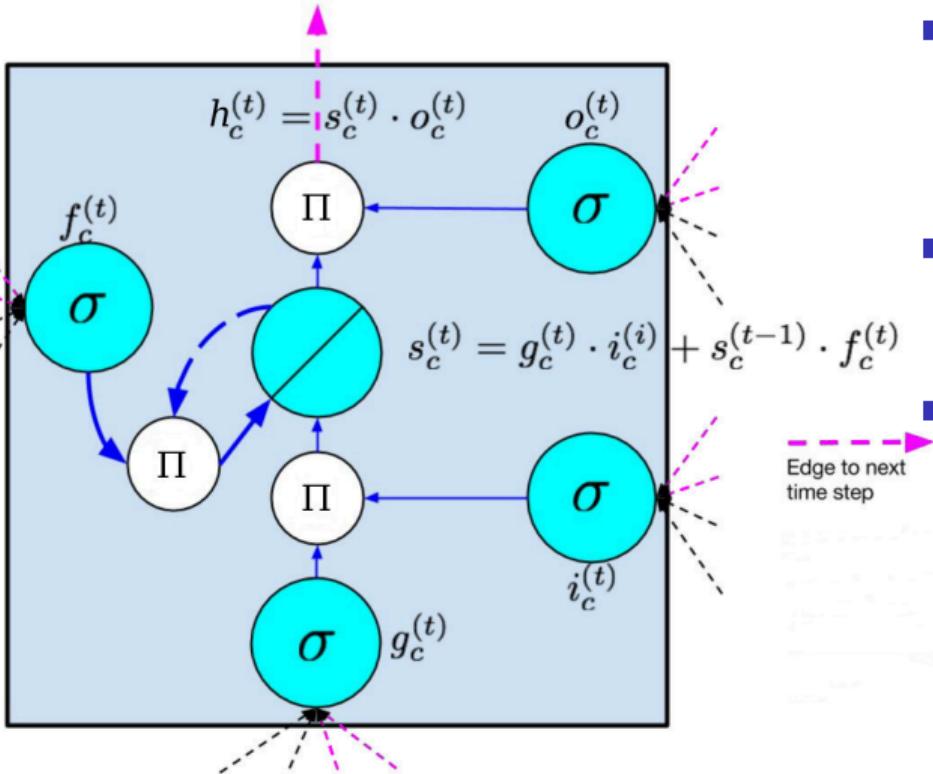
- Don't update the hidden state all the time, but only when necessary.
Use gates to control information flow
- A **gate** is a simple neural network with sigmoid activation on the output. It takes the current input $x^{(t)}$ and the previous state $h^{(t-1)}$ and outputs a vector with values in $[0, 1]$.
0 means gate is closed; 1 means gate is open
- We thus arrive at **Gated RNNs**. Intuitively, gates learn to distinguish important and non-important information.
Only let important information update the internal-state.

LSTM



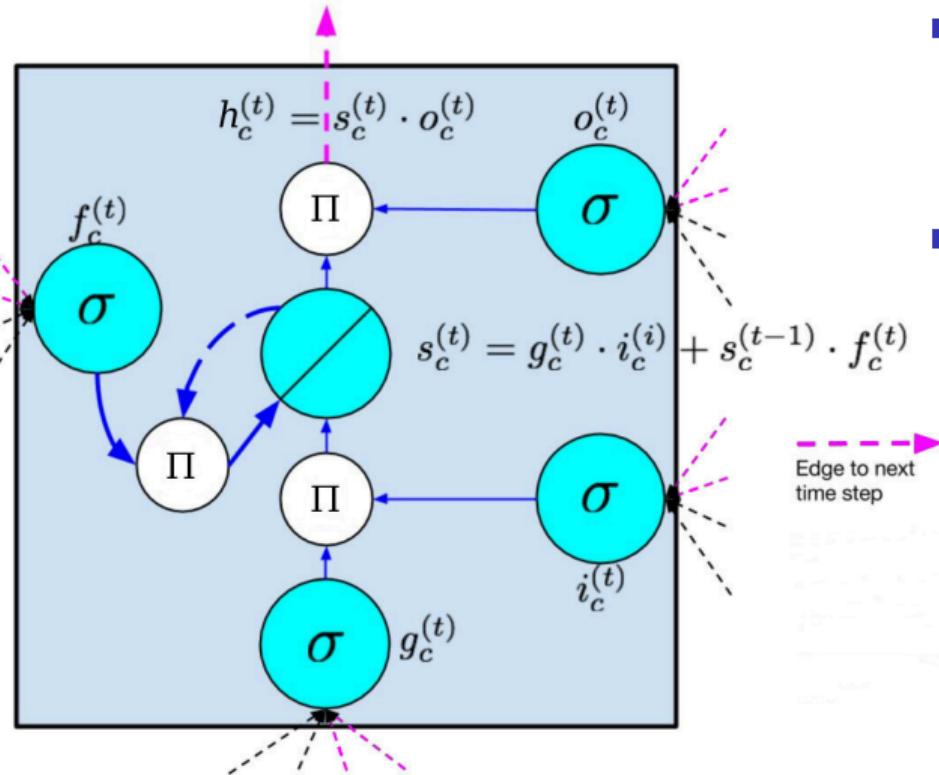
- Long Short Term Memory (LSTM) are a popular variant of gated RNNs.
- Has multiple gates to decide how the **internal state / memory** $s_c^{(t)}$ is updated.
- Here, $x_c^{(t)}$ is the **input** at time t , and $h_c^{(t-1)}$ is the previous **hidden state**.
- Both $x_c^{(t)}$ and $h_c^{(t-1)}$ are given to all the gates.
- Π represents point-wise product of two vectors.

LSTM



- $g(t)_c$ is the **Input Node** which encodes the input $x^{(t)}$. Its output is what is actually added to $s_c^{(t)}$ instead of $x^{(t)}$.
- $i_c^{(t)}$ is the **Input Gate**. Decides what bits of the input $x_c^{(t)}$ are added to $s_c^{(t)}$.
- $f_c^{(t)}$ is the **Forget Gate**. Decides what bits of $s_c^{(t-1)}$ is retained in $s_c^{(t)}$ and what is forgotten.

LSTM



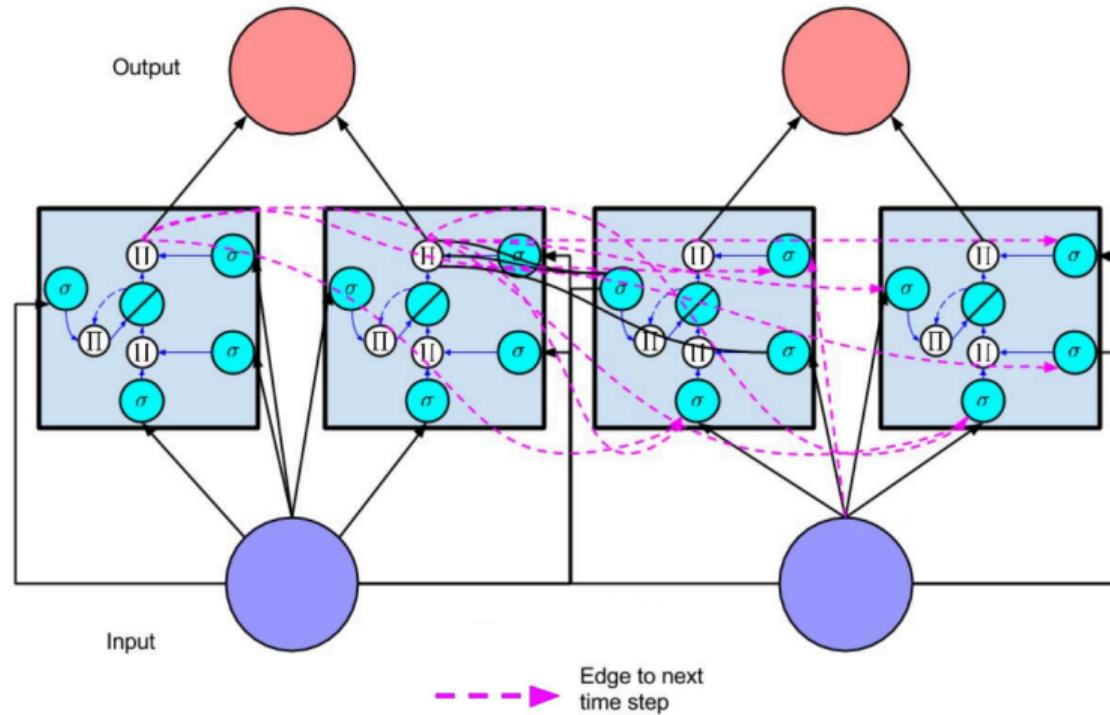
- We update the internal state as

$$s_c^{(t)} = \Pi(g_c^{(t)}, i_c^{(t)}) + \Pi(s_c^{(t-1)}, f_c^{(t)})$$

- The Output Gate $o_c^{(t)}$ decides what bits of the internal state $s_c^{(t)}$, is output to the hidden state $h_c(t)$

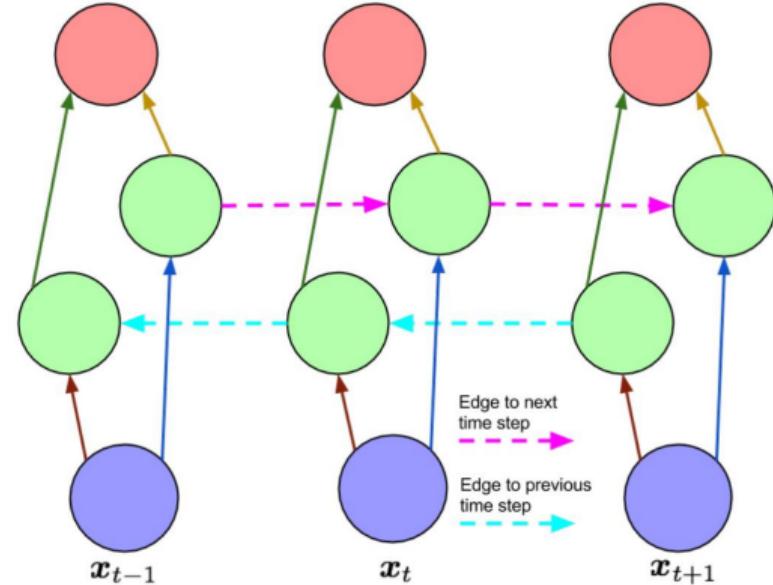
$$h_c^{(t)} = \Pi(s_c^{(t)}, o_c^{(t)})$$

LSTM unrolled in time



Bidirectional RNN

- Useful when we need context from both past and “future”
 - translation
 - handwriting recognition
 - Fill in the blank
- The whole input must be available
- Training via BPTT



Lecture 11: Autoencoders

Pranabendu Misra

Advanced Machine Learning 2023
Chennai Mathematical Institute

Learning (Sensible) Compressed Representations

Learning a sensible and compressed representation of a given dataset is an important task:

- Given an image, learn about the objects in the image
 - Then we can add / remove / modify the objects in the image, thereby generating a new image
 - We can generate similar images
 - We can create sensible interpolation between two images

- Given a music file, learn about the instruments in the music
 - Remove noise and distortions
 - Add or remove other instruments
 - and so on ...

This is an unsupervised learning task

- Consider the following two images



- A usual interpolation between the two images could be:



- A more sensible interpolation between the two images would be:



- This is something that, e.g. a child will imagine, a dog transforming into bird looks like.

- Example: shifting the image



- Example: Night to Day



■ Super Resolution



Garcia (2016) srez --- github.com/david-gpu/srez

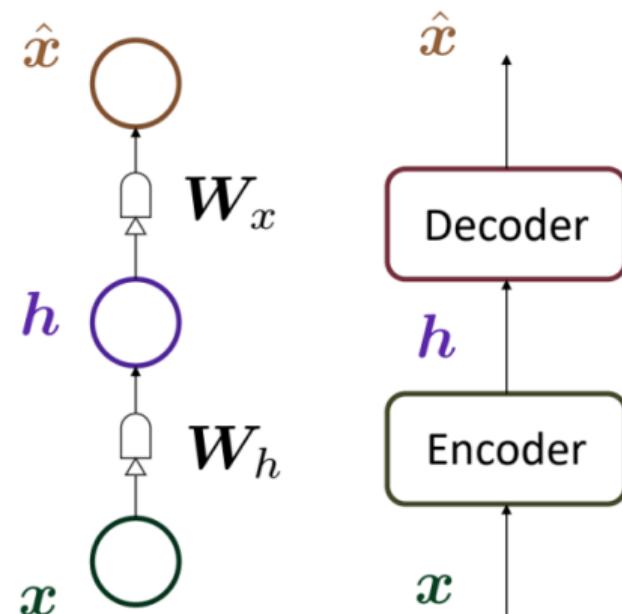
- Caption to Image:



Input Caption: A bright blue bird with white belly

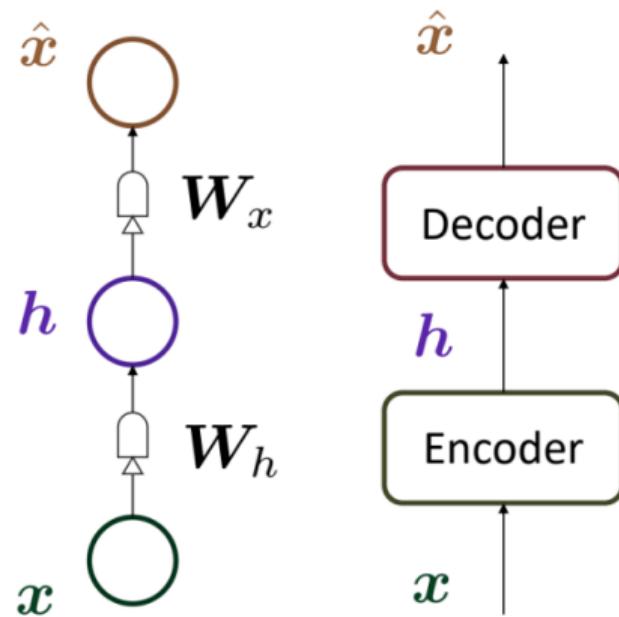
Autoencoders

- The idea is to learn how to encode the input into a compressed form, and then re-generate the input from the encoding
- An **encoder** network learns how to map the input x to a **code** h in the **latent space** of far smaller dimension.
- And at the same time, a **decoder** network learns to map the code h back to an approximate from of the input \hat{x}
- This is an *Unsupervised Learning* task.



Autoencoders

- In equations: $h = f(W_h x)$
 $\hat{x} = g(W_x h)$
- Here f and g are the non-linear activation functions
- Without these non-linear activations, and $W_x = W_h^\top$ the above equations would define PCA (Principal Component Analysis), where the dimension of the vector h is the number of principal components.
- So Autoencoders can be thought of as a generalization of PCA

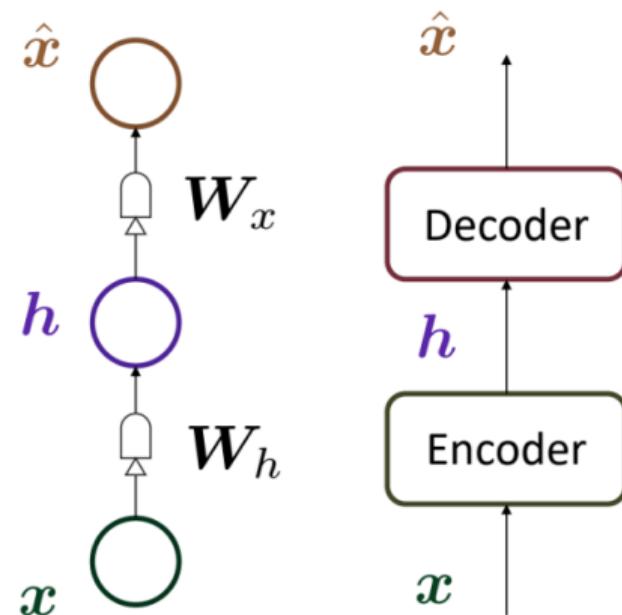


Autoencoders

Loss functions:

- Mean Squared Error: $\ell(x, \hat{x}) = \frac{1}{2} \|x - \hat{x}\|^2$
Suitable for $x \in \mathbb{R}^n$, e.g. an image.

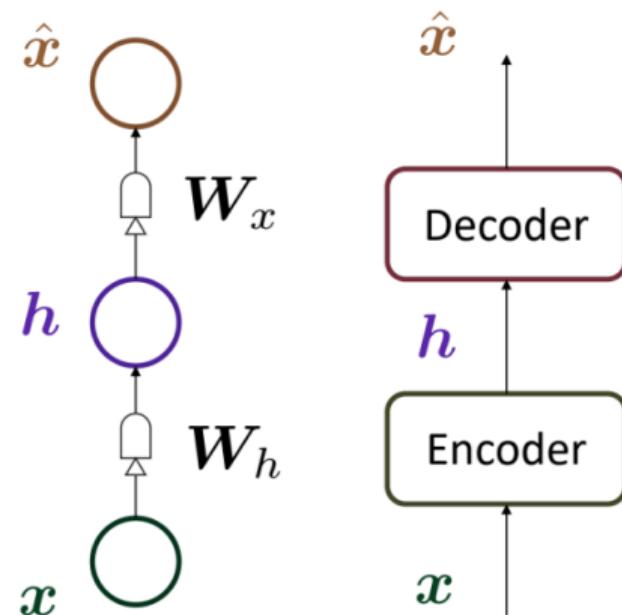
Reconstruction Error



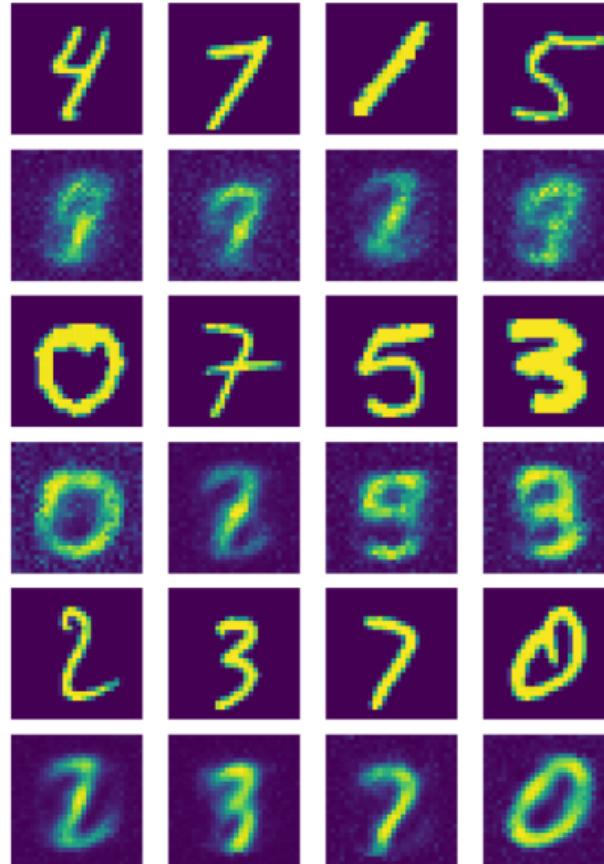
Autoencoders

Loss functions:

- Mean Squared Error: $\ell(x, \hat{x}) = \frac{1}{2} \|x - \hat{x}\|^2$
Suitable for $x \in \mathbb{R}^n$, e.g. an image.
- Clearly, minimizing the loss-function means that the encoder and decoder learn to compress and reconstruct the inputs in the training set.
- We can then validate using the test set.



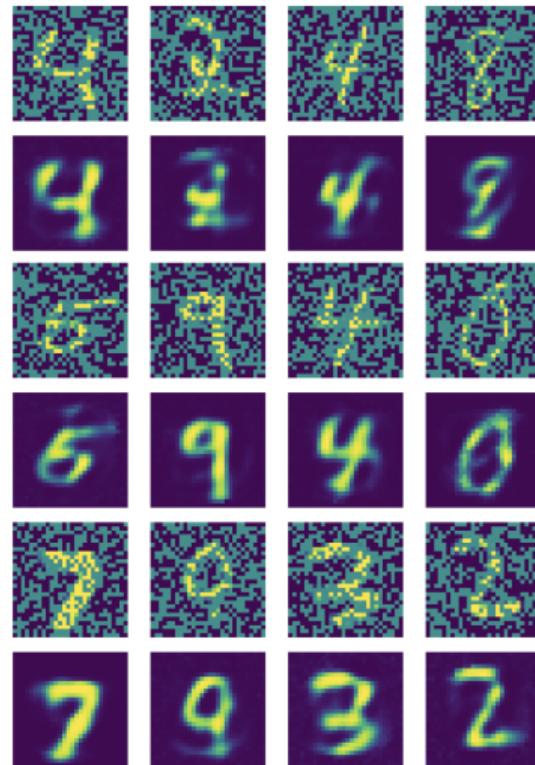
The outputs generated by an autoencoder trained on MNIST.



Autoencoders: Variants

De-noising Autoencoder:

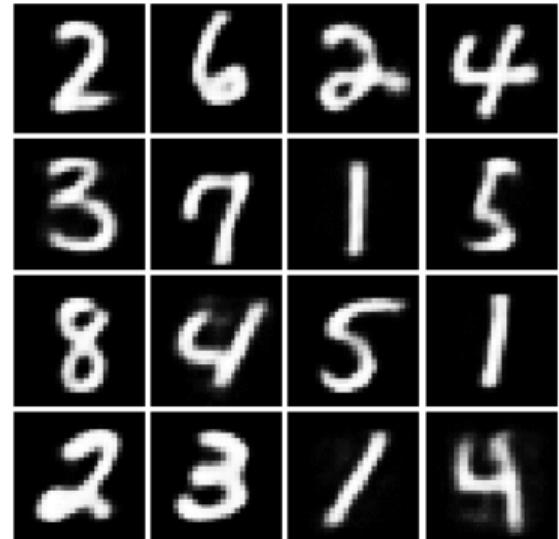
- The input to the encoder is $x + r$ where r is some small amount of random noise.
- The Loss Function still compares x and \hat{x}
- The idea is the encoder will learn to separate out the noise r from x , thus learning useful features of the input and generating \hat{x} from those.
- The AE learns to filter out noise that present in the input data.



Autoencoders: Variants

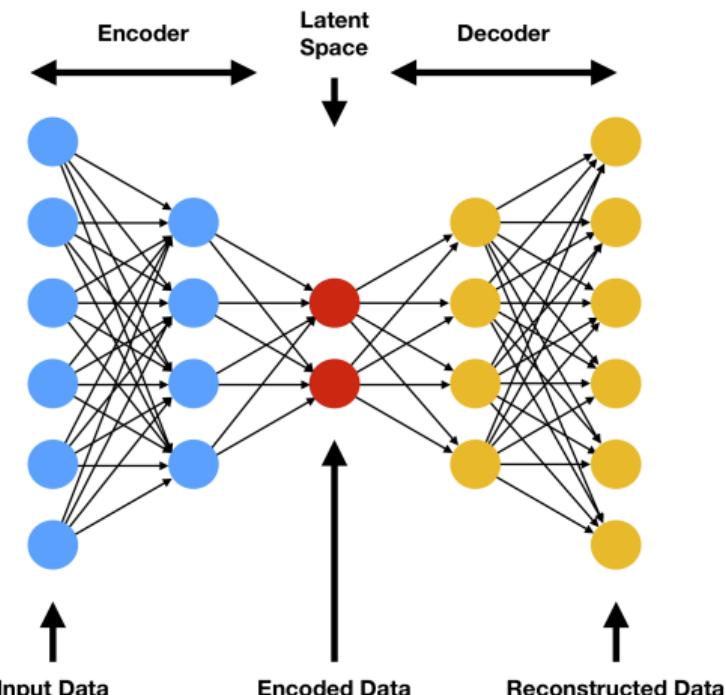
Contractive Autoencoder:

- Enforces the notion that two similar inputs x and x' should map to similar codes h and h' in the latent space.
- The Loss Function has an extra **Regularization term**, which is $\lambda \cdot \|J_h(x)\|^2$
- $J_h(x)$, the Jacobian, is the matrix of partial derivatives of h w.r.t. x .
- So the above term in the loss forces smaller derivatives for h . Hence, small changes in x should lead to small changes in h .



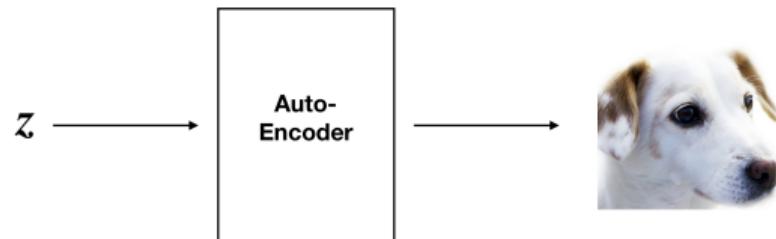
Autoencoders: The Latent Space

- The input data-points x are mapped to points h in the latent-space.
- We choose the dimension of the latent-space to reflect the representational dimension of the input data.
e.g. for a data-set of human faces, a latent space dimension of 50 would be reasonable.
- Is the latent space sensible? If we sample a point from the latent-space, will the decoder produce an interesting image out of it?

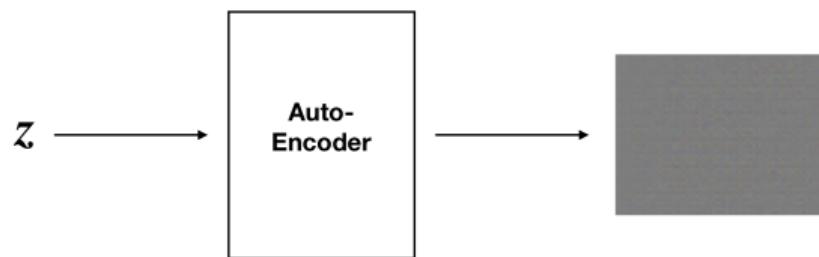


Autoencoders: Data generation from random point in Latent Space?

Expectation:



Reality:



Turns out that auto-encoders learn something like jpeg compression, most random latent-space points are meaning-less.

Variational Autoencoders (VAE)

The generation aspect is at the center of this model.

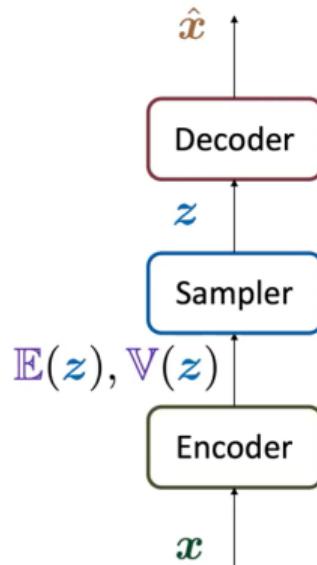
- The encoder maps x to a code h which has two parts $E(z)$ and $V(z)$.
 E and V stand for expectation and variance
- The next step is to sample a point z from a *Gaussian distribution* with expectation $E(z)$ and variance $V(z)$.

$$z = E(z) + \epsilon \odot \sqrt{V(z)}$$

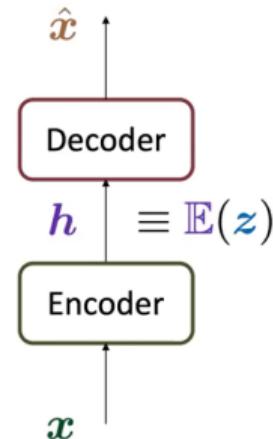
where $\epsilon \sim N(0, I_d)$ and d is the dimension of the latent-space.

- Finally, the decoder maps this randomly sampled point z to \hat{x}

Variational auto-encoder



Classic auto-encoder



Variational Autoencoders (VAE)

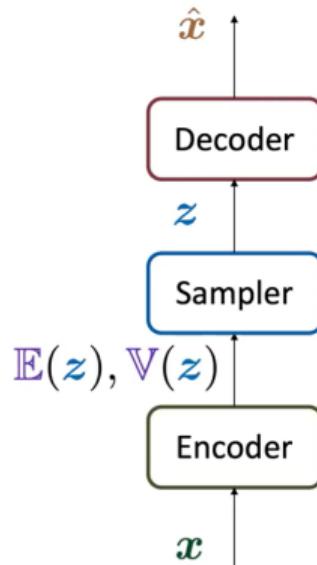
The generation aspect is at the center of this model.

- The encoder maps x to a code h which has two parts $E(z)$ and $V(z)$.
 E and V stand for expectation and variance
- The next step is to sample a point z from a *Gaussian distribution* with expectation $E(z)$ and variance $V(z)$.

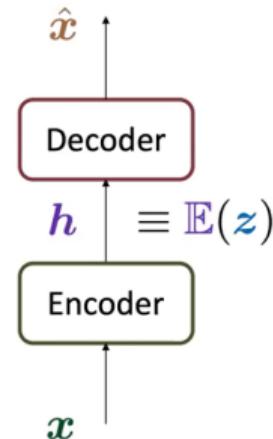
$$z = E(z) + \epsilon \odot \sqrt{V(z)}$$

where $\epsilon \sim N(0, I_d)$ and d is the dimension of the latent-space.

Variational auto-encoder



Classic auto-encoder



So the code in the latent-space specifies a probability distribution.

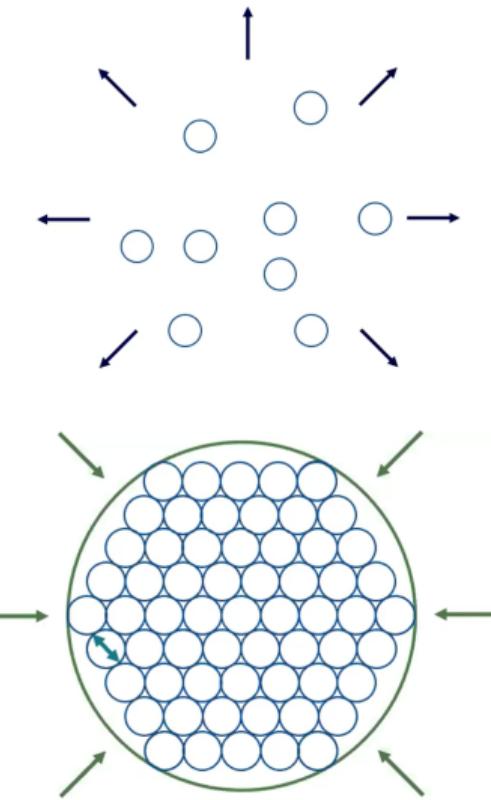
Variational Autoencoders (VAE)

Loss function:

- The reconstruction error $\ell(x, \hat{x}) = \frac{1}{2} \|x - \hat{x}\|^2$
 - It pushes the “probability balls” apart, because it penalizes overlapping.
 - We also have no control over the “size” of the probability balls.
- Regularization term

$$\beta \cdot \ell_{KL}(z, N(0, I_d))$$

- It brings the probability balls closer
- It enforces a size on the probability balls



Variational Autoencoders (VAE)

Loss functions for VAE:

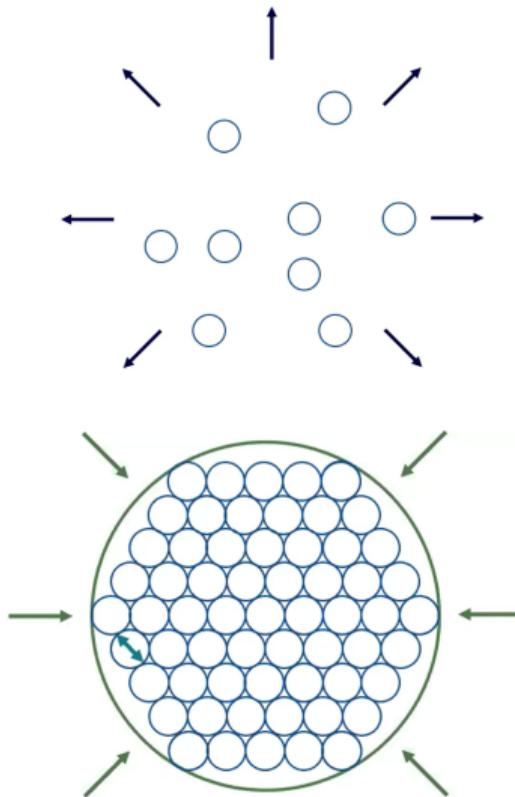
- Apart from the usual loss, there is an extra regularization term in the loss

$$\ell_{KL}(z, N(0, I_d)) =$$

$$\frac{1}{2} \sum_{i=1}^d (V(z_i) - \log(V(z_i)) - 1 + E(z_i)^2)$$

divergence b/w distribution of z and $N(0, I_d)$.

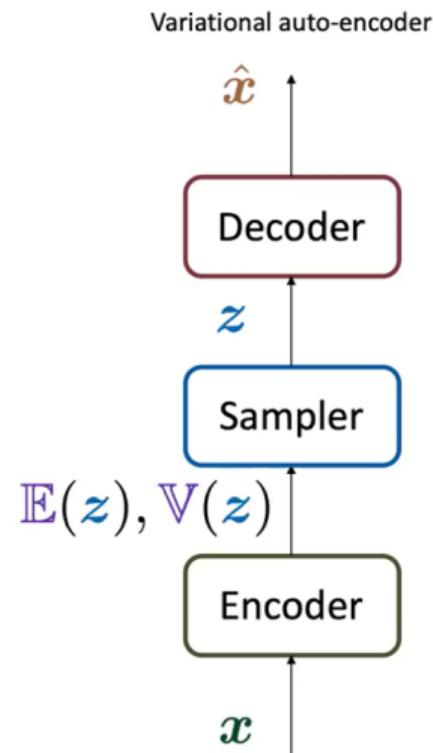
- The term $(V(z_i) - \log(V(z_i)) - 1)$ is minimized when $V(z_i) = 1$. This makes the “size” of the balls radius-1.
- The term $E(z_i)^2$ penalizes large values, so the balls are tightly packed as much as possible.



Variational Autoencoders (VAE)

Loss functions for VAE:

- Overall, the loss function leads to a *clustering* of similar inputs in the latent-space, and these clusters *densely packed*
- This means that if we pick a point in the latent-space, it can generate a good-quality image.
- We can also *interpolate* between two input-points in the latent-space to generate new images.



Examples: VAE trained on Celebrity Faces

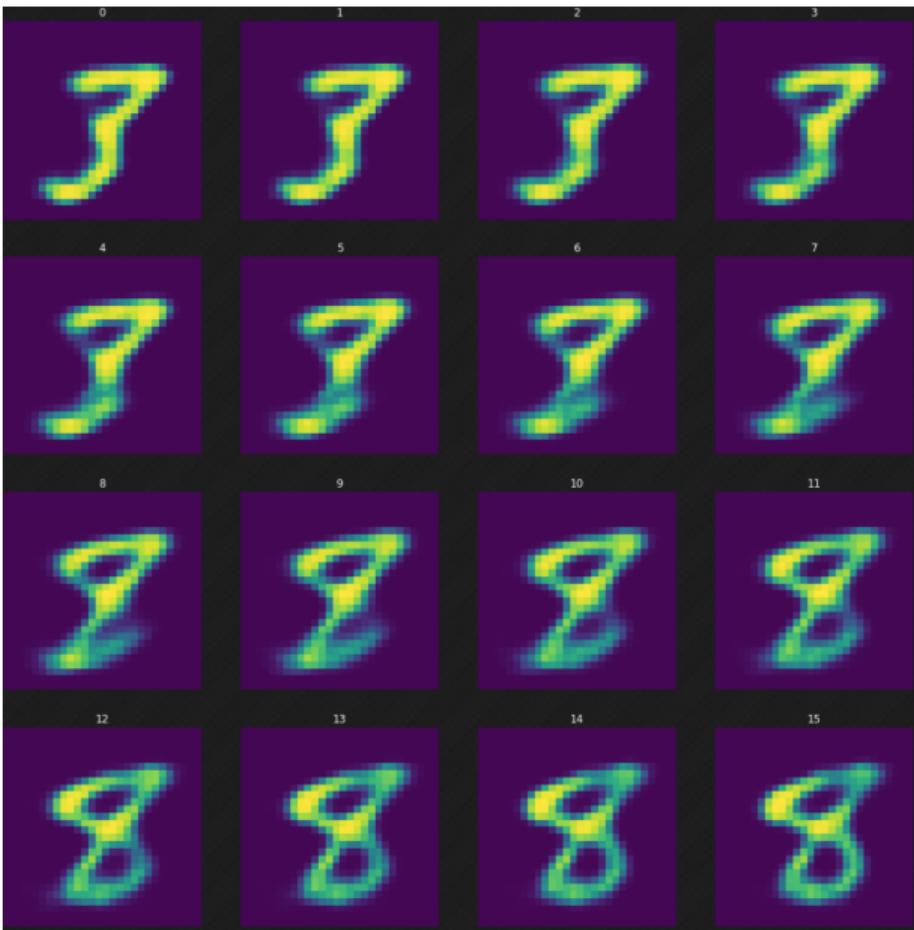
Interpolating between hair-colors:



Interpolating between no glasses and glasses







References:

These slides are based on:

- <https://www.compthree.com/blog/autoencoder/>
- <https://atcold.github.io/pytorch-Deep-Learning/en/week07/07-3/>
- <https://atcold.github.io/pytorch-Deep-Learning/en/week08/08-3/>

Generative Adversarial Models

Pranabendu Misra

Advanced Machine Learning 2023

Generative Adversarial Network

- Another type of *Generative* models, like autoencoders.
- **The generation step is at the heart of this model.**

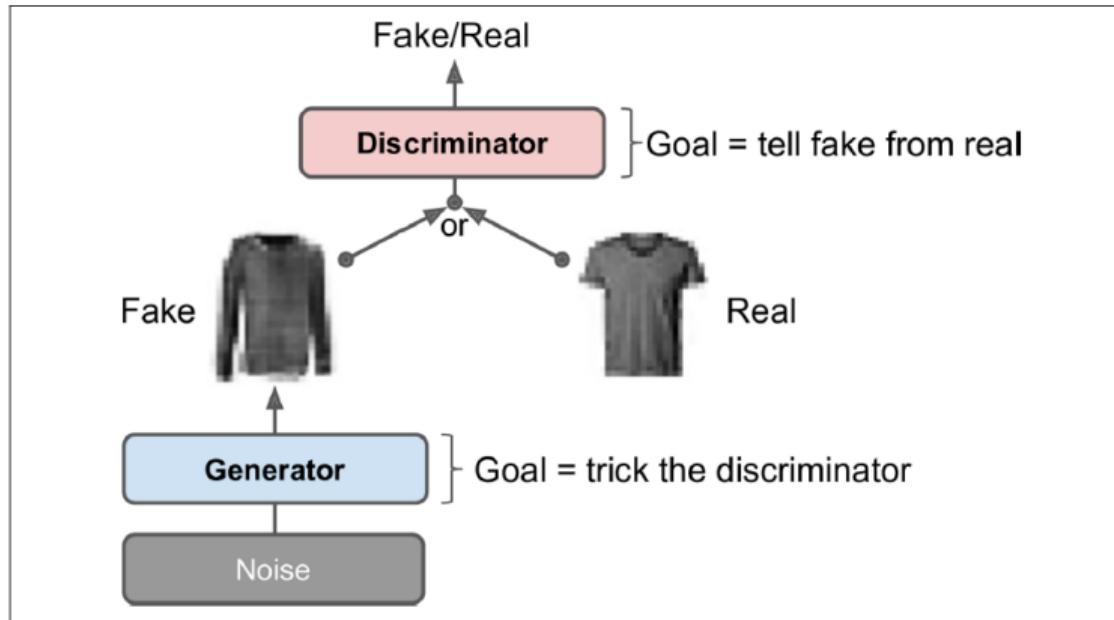


Figure 17-15. A generative adversarial network

Generative Adversarial Network

- Based on a turn-based-game between 2 players
- **Generator:** A Neural Network which given some *random noise* it generates a *fake datapoint*.
- **Discriminator:** A Neural Network that given an image determines if it is real or fake.

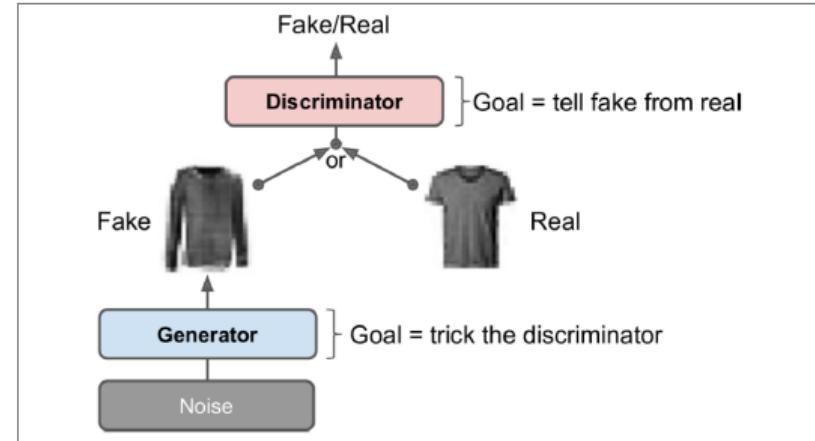


Figure 17-15. A generative adversarial network

This is a game-theoretic modeling of the problem.

- Generator and Discriminator are both trying to outwit each other.
- In this process, the Generator learns to produce very good *fake data*.

Generator



Discriminator

Generator



Real data

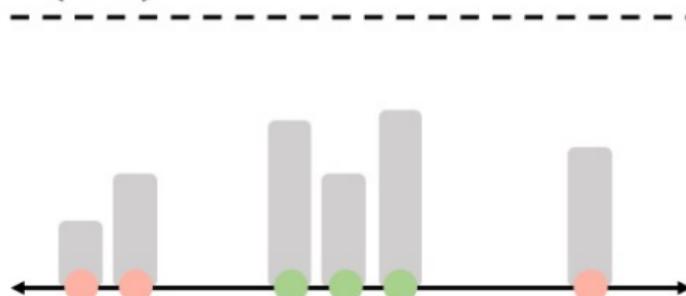


Fake data

GAN: Learning

Discriminator

$$P(\text{real}) = 1$$



Generator



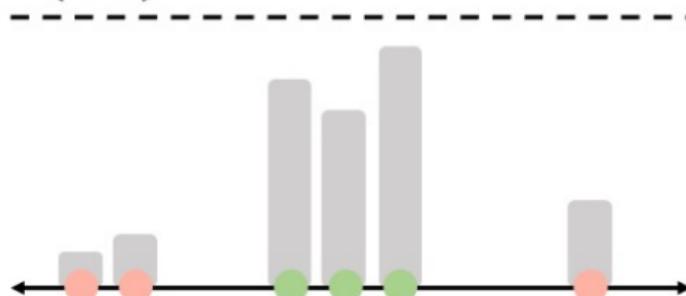
Real data

Fake data

GAN: Learning

Discriminator

$$P(\text{real}) = 1$$



Generator



Real data

Fake data

Discriminator

$$P(\text{real}) = 1$$



Generator

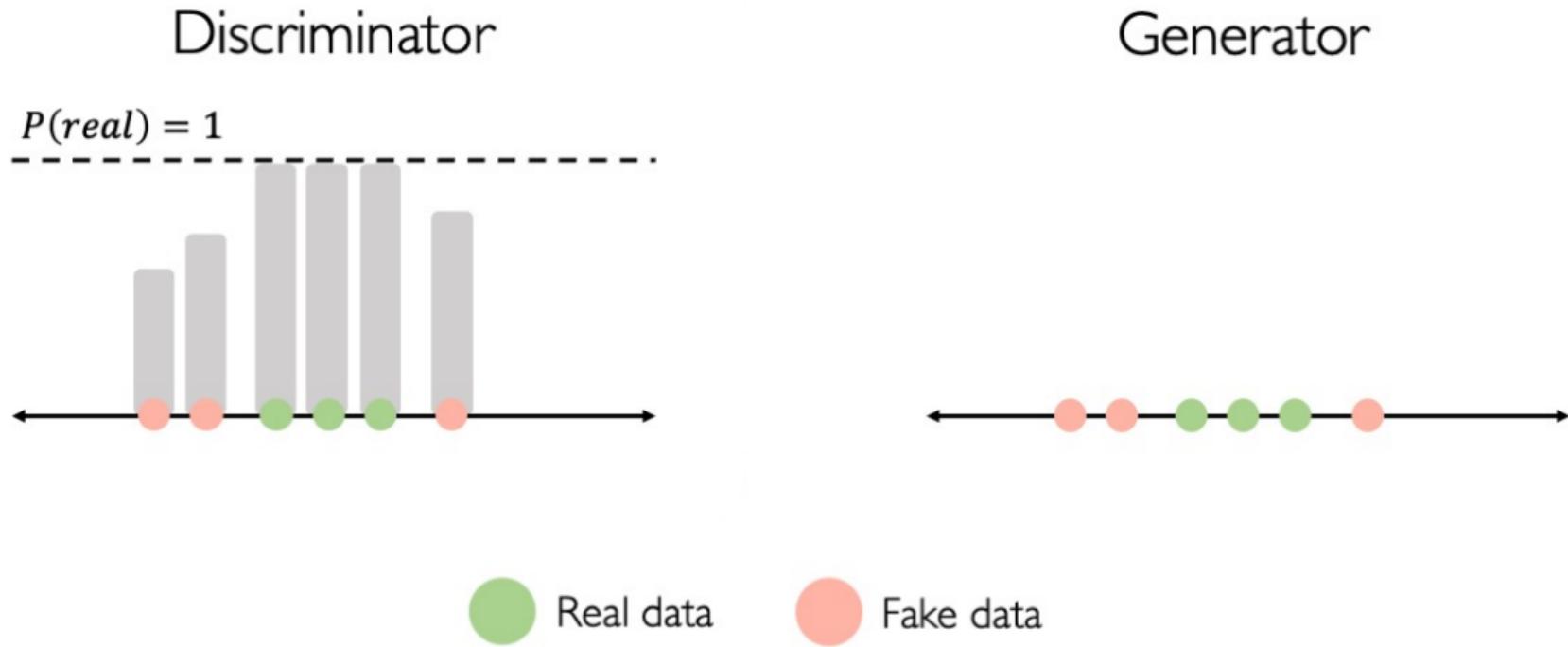


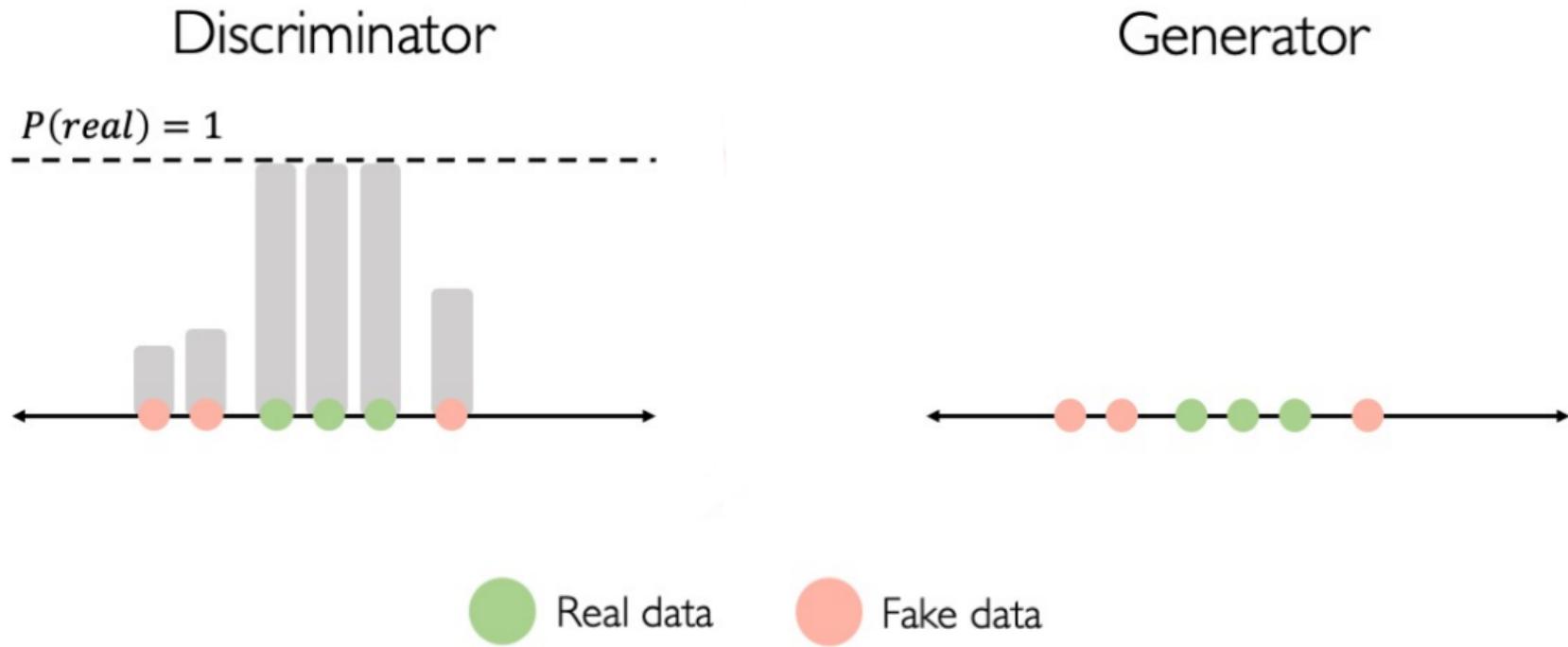
Real data



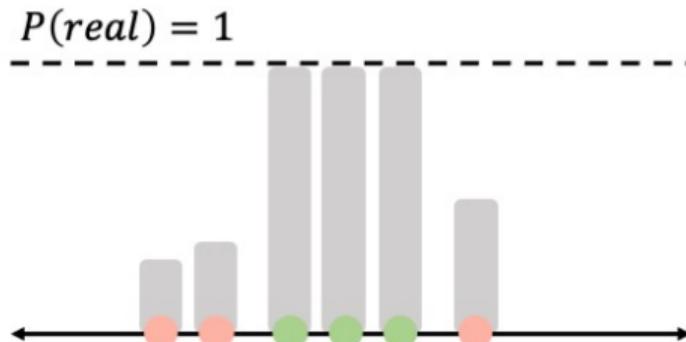
Fake data

GAN: Learning





Discriminator



Generator



Real data

Fake data

Generative Adversarial Network

- **Generator:** A Neural Network which given some *random noise* it generates a *fake datapoint*.
- **Discriminator:** A Neural Network that given an image determines if it is real or fake.

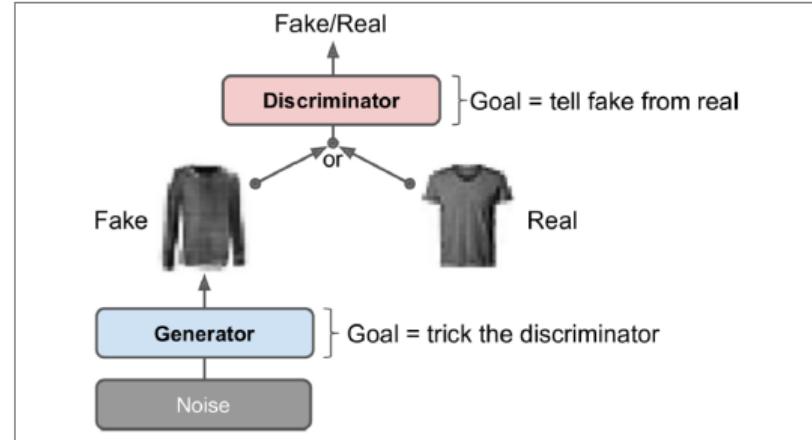


Figure 17-15. A generative adversarial network

Generative Adversarial Network: Loss Functions and Training

- A *Sampler* S samples a vector z in the *latent space* using a normal distribution.
- The Generator G maps z to a data point.

$$\hat{x} = G(z)$$

- The Discriminator D gets $\{\hat{x}, x\}$ where x is a training sample, i.e. a real datapoint.
- D outputs a probability for both \hat{x} and x of them being real (i.e. not generated by G)

$$\hat{y} = D(\hat{x})$$

$$y = D(x)$$

- Note that \hat{y} and y are probabilities

Generative Adversarial Network: Loss Functions and Training

- Discriminator's Loss:

$$\log(D(G(z))) + (1 - \log(D(x)))$$

- Generator's Loss:

$$1 - \log(D(G(z)))$$

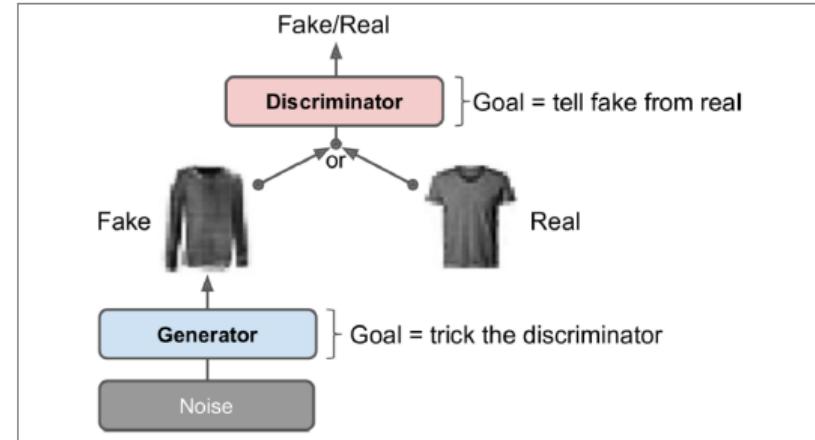


Figure 17-15. A generative adversarial network

- We train in rounds
- The Generator produces a batch of fake data $G(z)$
- A batch of real data x and $G(z)$ are both run through the discriminator, and loss $\log(D(G(z))) + (1 - \log(D(x)))$ is computed.
- We then update D via a gradient descent step, keeping G fixed.

Generative Adversarial Network: Loss Functions and Training

- Discriminator's Loss:

$$\log(D(G(z))) + (1 - \log(D(x)))$$

- Generator's Loss:

$$1 - \log(D(G(z)))$$

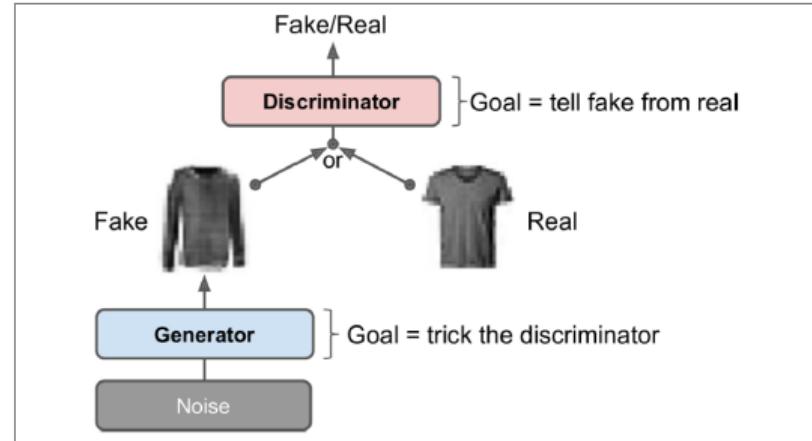


Figure 17-15. A generative adversarial network

- Next, the generator then produces a new batch of fake data $G(z')$
- These are processed by the discriminator to get the loss $\log(D(G(z')))$.
- We then update G by a gradient descent step, keeping D fixed.

Generative Adversarial Network: Loss Functions and Training

- Training GANs can be difficult, and we have to be very careful
- The training succeeds if we reach a good equilibrium between the Generator and the Discriminator.

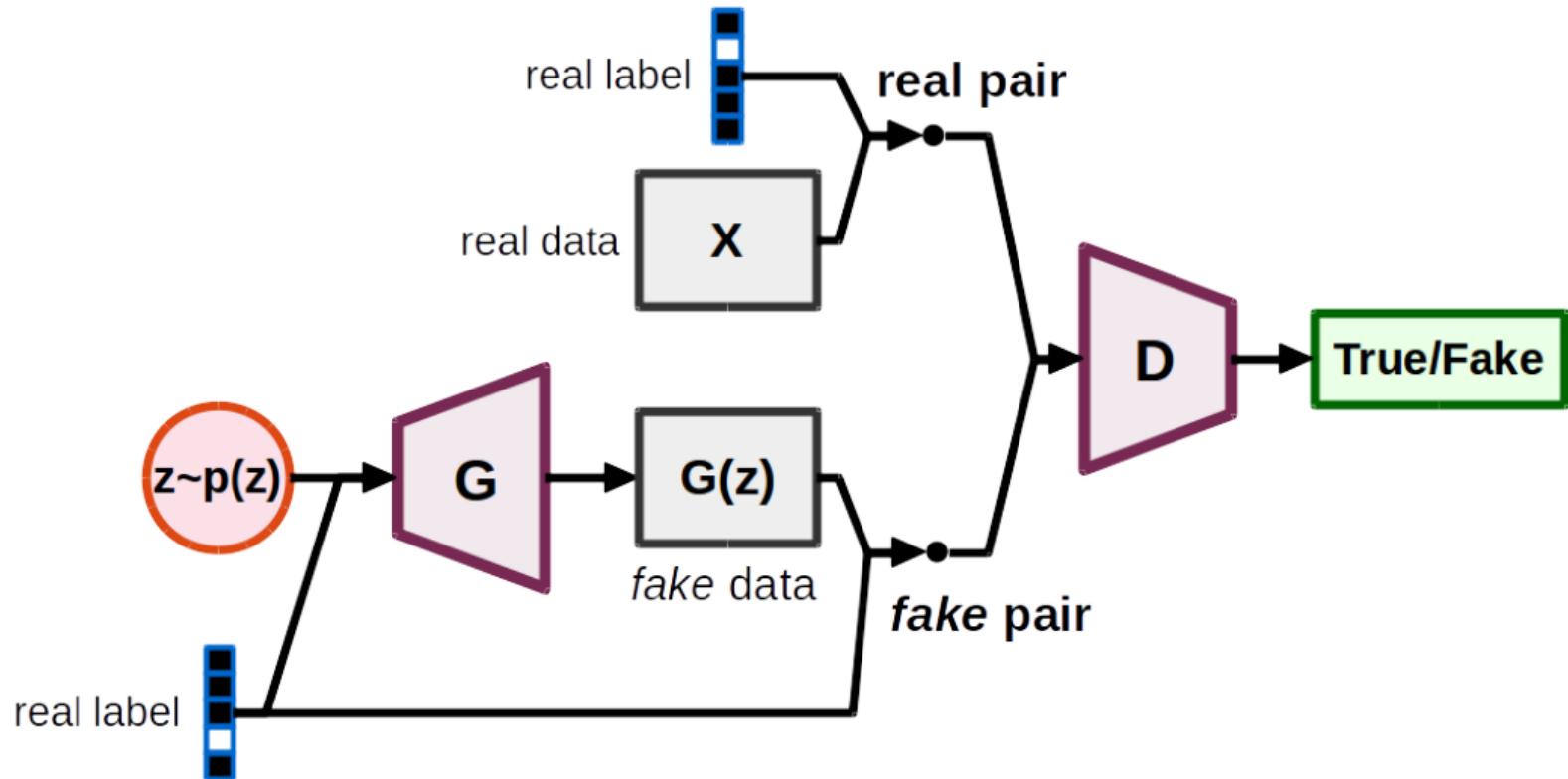
Some issues that may arise:

- *Mode Collapse*: The Generator only produces a small set of distinct images.
- *Convergence Failure*: The Generator produces poor quality images even after training for a long time. Could happen because the opponent network always counters whatever improvements you make, blocking every direction of progress in gradient descent
- *Losses don't indicate progress*: Even as the generator is improving, so is the discriminator. So the loss of the Generator may keep increasing even though it is getting better.

GANs for image synthesis: latest results



Conditional GAN

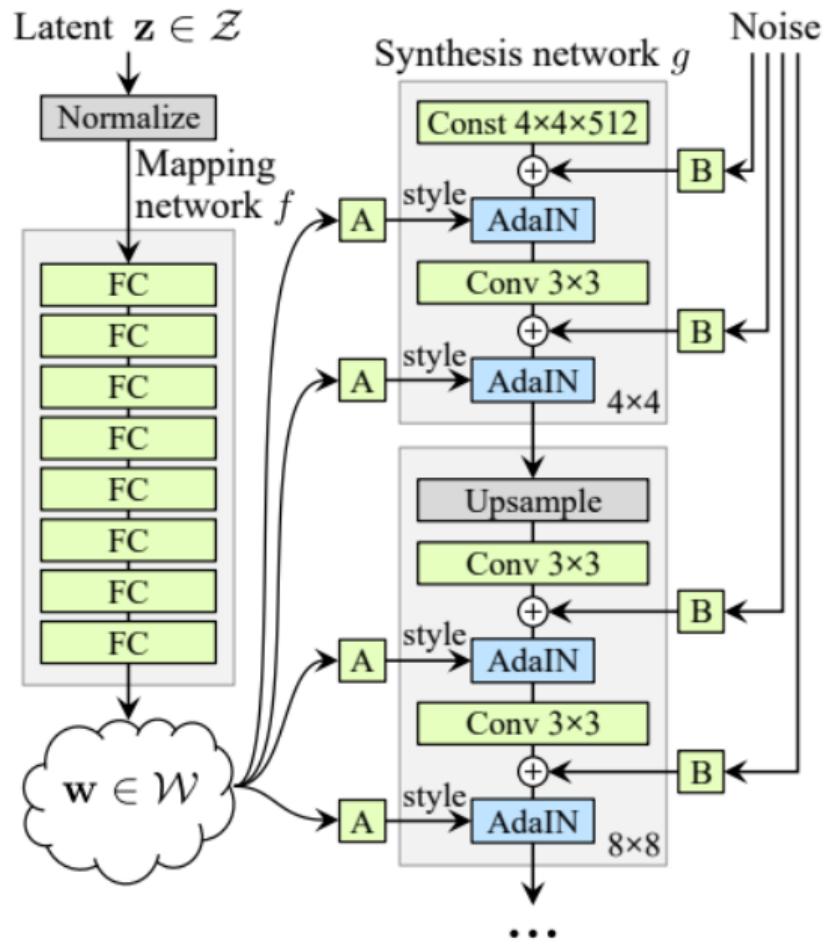


StyleGAN

- Progressively generates larger and larger images.

$4 \times 4 \rightarrow 8 \times 8 \rightarrow 16 \times 16 \rightarrow \dots 1024 \times 1024$

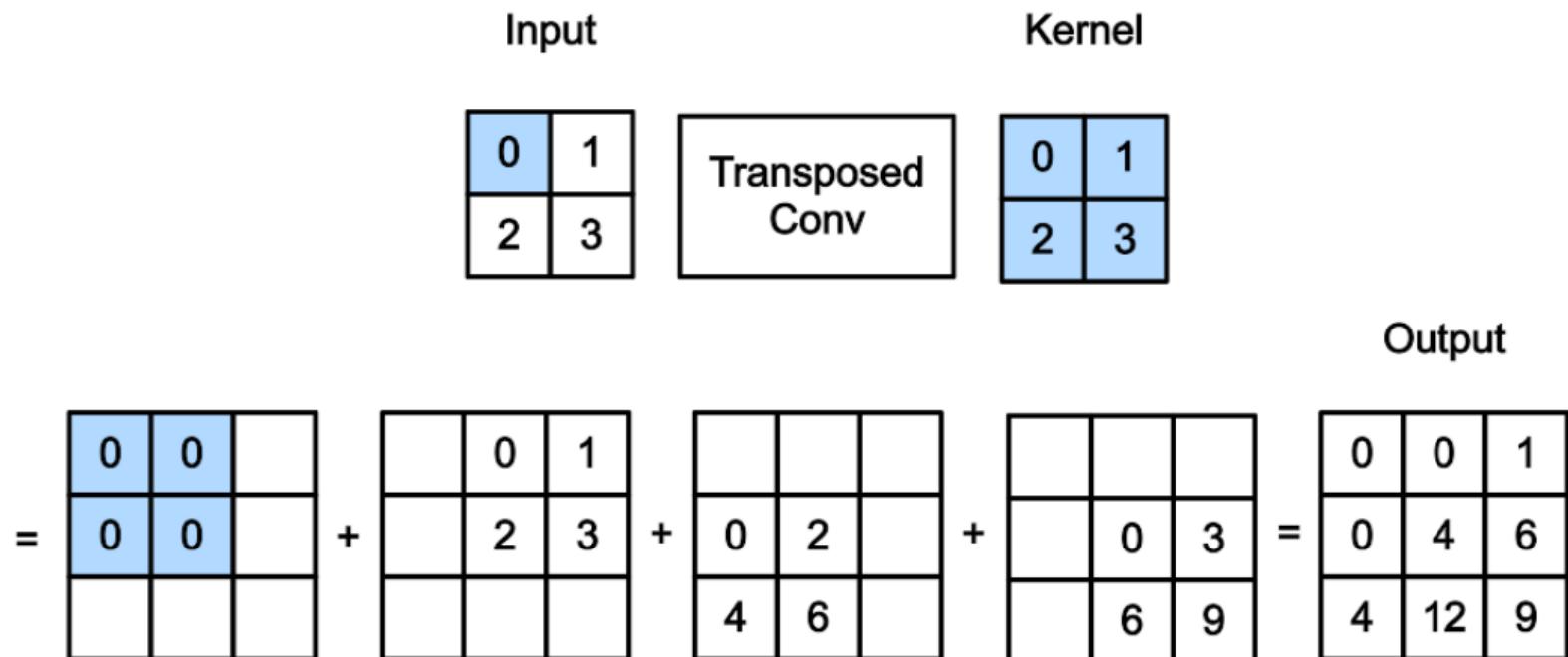
- Each stage takes the output of the previous stage and upscales it.
- Maps the sampled noise z to a **style code** w
- The style code w is supplied to every stage



StyleGAN: Style Mixing



Upsampling Images: Transposed Convolution



References:

These slides are based on:

- http://introtodeeplearning.com/slides/6S191/MIT_DeepLearning_L4.pdf
- Generative Adversarial Networks, Goodfellow et al, Communications of ACM, November 2020

Reinforcement Learning

Pranabendu Misra

based on slides by Madhavan Mukund.

Advanced Machine Learning 2023

An alternative approach to learning

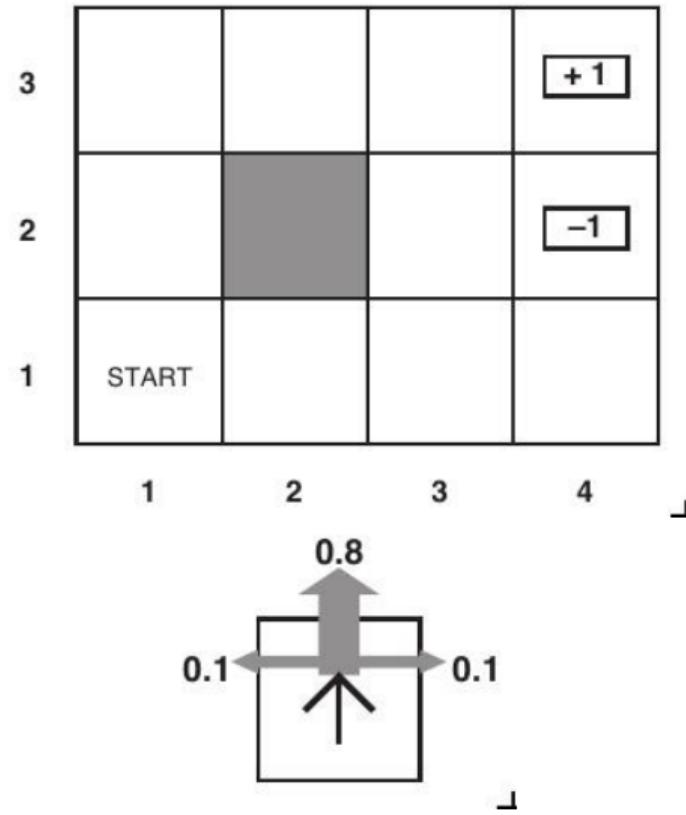
- Supervised learning — use labelled examples to learn a classifier
- Unsupervised learning — search for patterns, structure in data
- Reinforcement learning — learning through interaction
 - Choose actions in an uncertain environment
 - Actions change state, yield rewards
 - Learn optimal strategies to maximize long term rewards
- Examples
 - Playing games — AlphaGo, reward is result of the game
 - Motion planning — robot searching for an optimal path with obstacles
 - Feedback control — balancing an object

The components

- **Policy** What action to take in the current state
 - “Strategy”, can be probabilistic
- **Reward** In response to taking an action
 - Short-term outcome, may be negative or positive
- **Value** Accumulation of rewards over future actions
 - Long-term outcome, goal is to maximize value
- **Environment Model** How the environment will behave
 - Given a state and action, what is the **next state**, reward?
 - Probabilistic, in general
 - Use models for *planning*
 - Can also use RL without models, trial-and-error learners

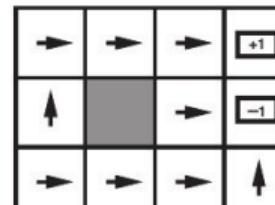
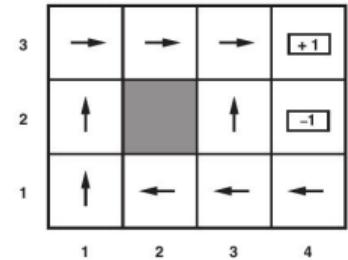
Motion planning example

- A reward when we reach a new state.
 - Two terminal states with rewards $+1, -1$
 - All other states have reward -0.04
 - Start from START state
 - Move till you reach a terminal state
 - Maximize the sum of the rewards seen
- Policy: direction to move from current box
 - Outcome of action is non-deterministic.
 - With probability 0.8 , go in intended direction
 - With probability 0.1 , go left or go right of the intended direction
 - Collision with boundary = stationary

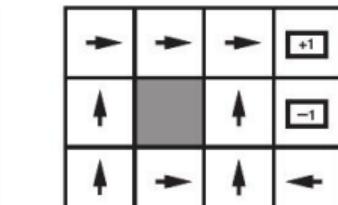


Motion planning example

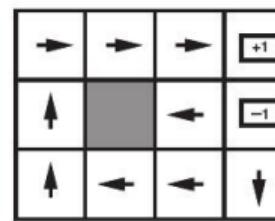
- Optimal policy learned by repeatedly moving on the board
 - From bottom right, conservatively follow the long route around the obstacle to avoid -1
- $R(s)$: reward for non-final states s
 - If $R(s) < -1.6284$, terminate as fast as possible
 - If $-0.4278 < R(s) < -0.0850$, risk going past -1 to reach $+1$ quickly
 - If $-0.0221 < R(s) < 0$, take no risks, avoid -1 at all cost
 - $R(s) = 0$ is shown above
 - If $R(s) > 0$ avoid terminating



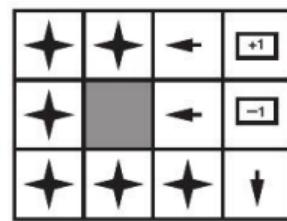
$$R(s) < -1.6284$$



$$-0.4278 < R(s) < -0.0850$$



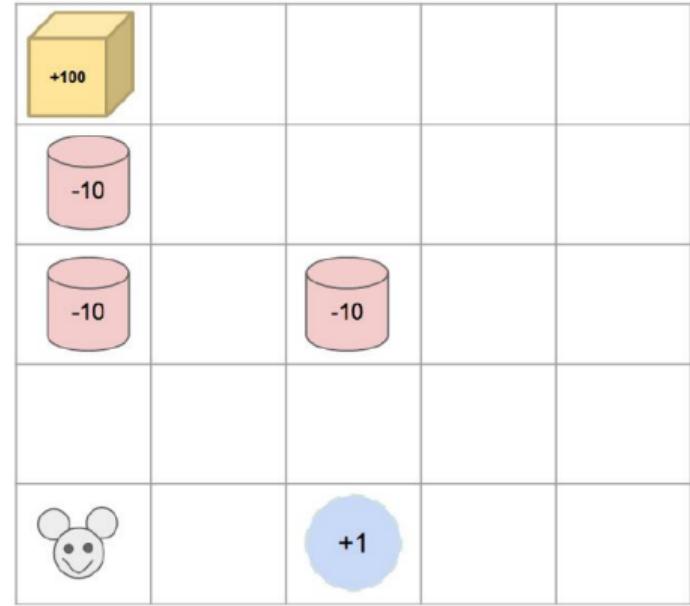
$$-0.0221 < R(s) < 0$$



$$R(s) > 0$$

Exploration vs exploitation

- Policy evolves by experience
- Greedy strategy is to always choose best known option
- Using this we may get stuck in a local optimum
 - Greedy strategy only allows the mouse to discover water with reward +1
 - Mouse never discovers a path to cheese with +100 because of negative rewards en route
- How to balance **exploitation** (greedy) vs **exploration**?
- Formalize these ideas using **Markov Decision Processes**



Bandits

- One-armed bandit — a slot machine in a casino
 - Put in a coin and pull a lever (the arm)
 - With high probability, lose your coin (the bandit steals your money)
 - With low probability, get varying reward, rewards follow some probability distribution
- k-armed bandit
 - Each arm has a different reward probability; we can pull one arm
 - Goal is to maximize total reward over a sequence of plays
- Action corresponds to choosing the arm a
 - For each action a , $q_*(a)$ is expected reward if we choose a
 - A_t is action chosen at time t , with reward R_t
 - If we knew $q_*(a)$ we would always choose $A_t = \arg \max_a q_*(a)$
 - Assume $q_*(a)$ is unknown — build an estimate $Q_t(a)$ of $q_*(a)$ at time t

Exploration and exploitation

- Build $Q_t(a)$, estimate of $q_*(a)$ at time t , from past observations (sample average)

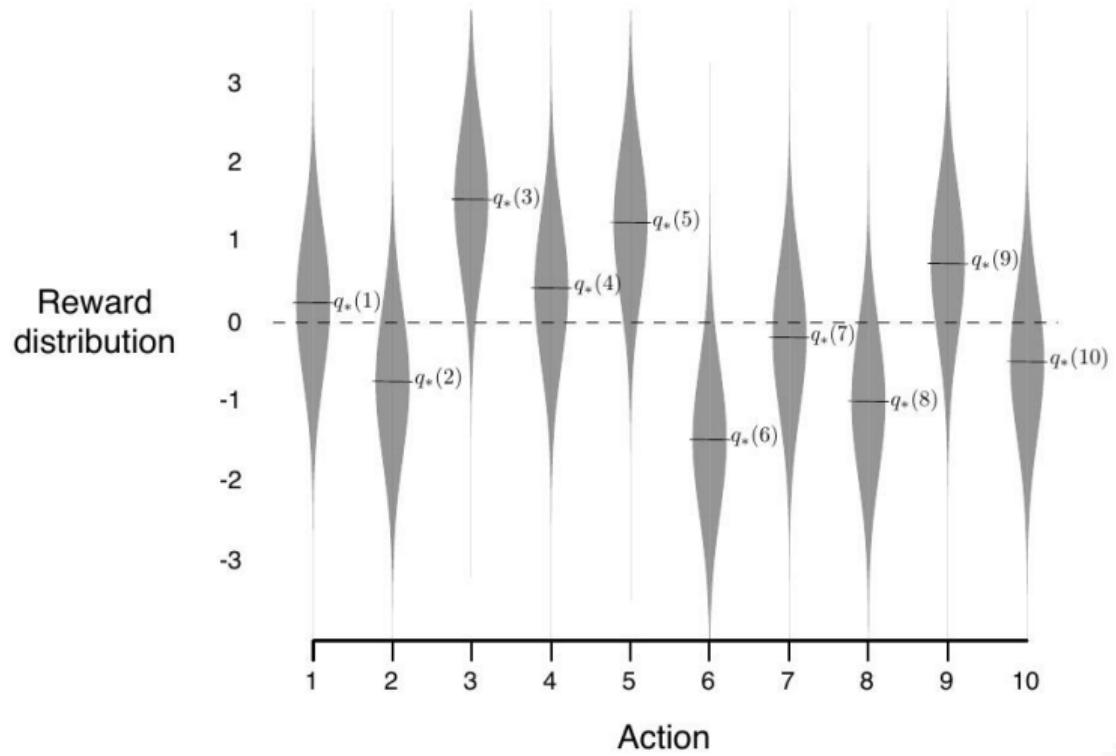
$$\frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_t=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_t=a}}$$

- **Greedy policy** chooses $\arg \max_a Q_t(a)$
- But it fails to explore other options.
How will we learn about all actions?
- ε -greedy policy
 - With small probability ε , choose a random action (uniform distribution)
 - With probability $1 - \varepsilon$, follow greedy
- ε -greedy is a simple way to balance exploitation with exploration
 - Theoretically, explores all actions infinitely often

Exploration and exploitation

10 bandit experiment

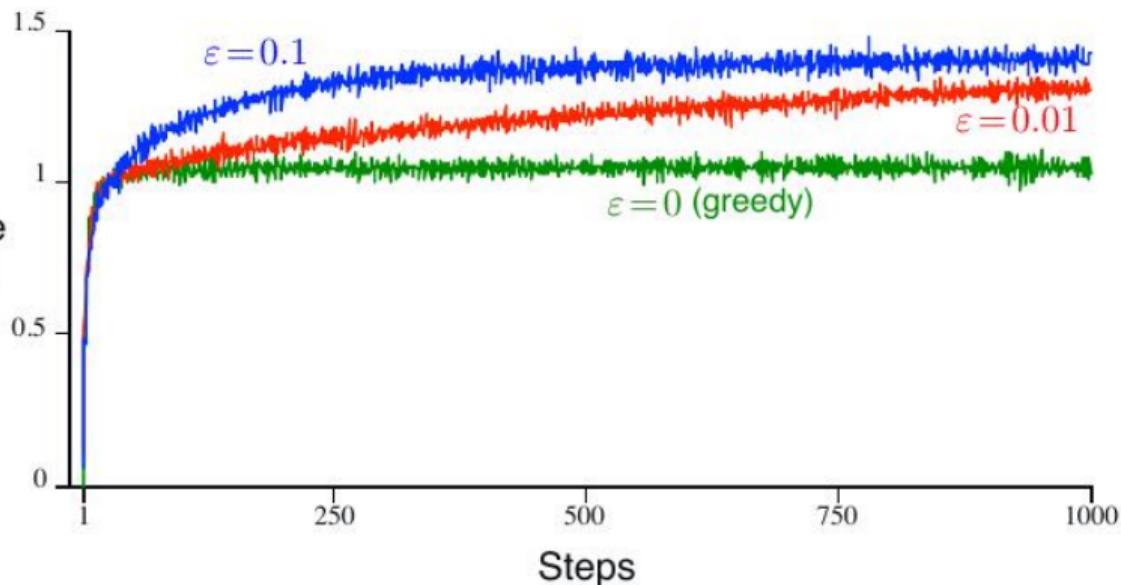
- Each bandit's reward follows Gaussian distribution
- Same variance, mean is chosen randomly



Exploration and exploitation

ε -greedy strategies results:

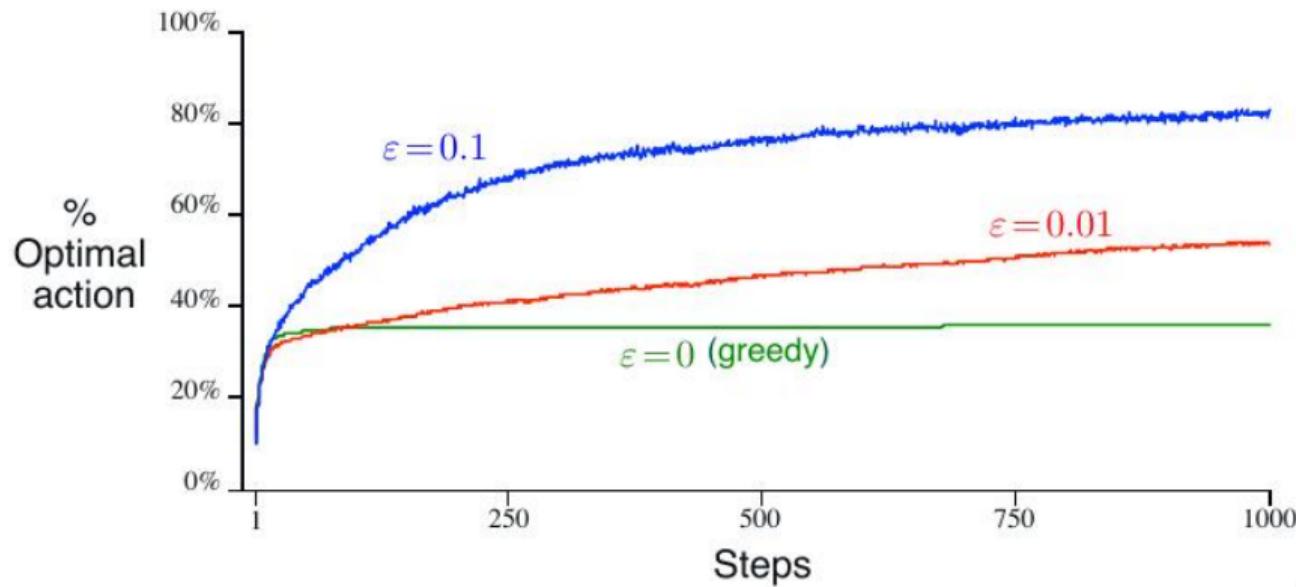
- Pure greedy strategy is sub-optimal
- Initial “learning rate” is Average reward more or less equal



Exploration and exploitation

Discovery of optimal actions

- Pure greedy strategy discovers optimal action only 1/3 of the time



Sidenote: Efficient Incremental calculation of Estimated Rewards

- Focus on a single action a . Sample average is $\frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_t=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_t=a}}$
- R_i — reward when a is selected for i th time
- Q_n — estimate of action value after a has been selected $n - 1$ times
- $$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n - 1}$$
- $$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i = \frac{1}{n} \left(R_n + \sum_{i=1}^{n-1} R_i \right) = \frac{1}{n} \left(R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} (R_n + (n-1)Q_n) = \frac{1}{n} (R_n + nQ_n - Q_n) = Q_n + \frac{1}{n} [R_n - Q_n] \\ &= Q_n + \frac{1}{n} [R_n - Q_n] \end{aligned}$$
- NewEstimate = OldEstimate + Step [Target - OldEstimate]

Stationary vs non-stationary Rewards

- Non-stationary Rewards: the Reward probability distribution changes over time
- Assume the distribution changes gradually.
This means recent rewards are more important.
- Exponentially decaying weighted average of rewards
- Use a constant step $\alpha \in (0, 1]$ —
$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$
- $$\begin{aligned} Q_{n+1} &= Q_n + \alpha[R_n - Q_n] = \alpha R_n + (1 - \alpha)Q_n \\ &= \alpha R_n + (1 - \alpha)[\alpha R_{n-1} + (1 - \alpha)Q_{n-1}] \\ &= \alpha R_n + \alpha(1 - \alpha)R_{n-1} + (1 - \alpha)^2 Q_{n-1} \\ &= \alpha R_n + \alpha(1 - \alpha)R_{n-1} + \alpha(1 - \alpha)^2 R_{n-2} + \cdots + \alpha(1 - \alpha)^{n-1} R_1 + (1 - \alpha)^n Q_1 \\ &= (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha(1 - \alpha)^{n-i} R_i \end{aligned}$$
- Initial value Q_1 affects the calculation — different heuristics possible

Summary

- k -armed bandit is the simplest interesting situation to analyze
- ε -greedy strategy balances exploration and exploitation
- Incremental update rule for estimates

$$\text{NewEstimate} = \text{OldEstimate} + \text{Step} [\text{Target} - \text{OldEstimate}]$$

- Exponentially decaying weighted average when rewards change over time (non-stationary)

Markov Decision Processes

Pranabendu Misra

based on slides by Madhavan Mukund

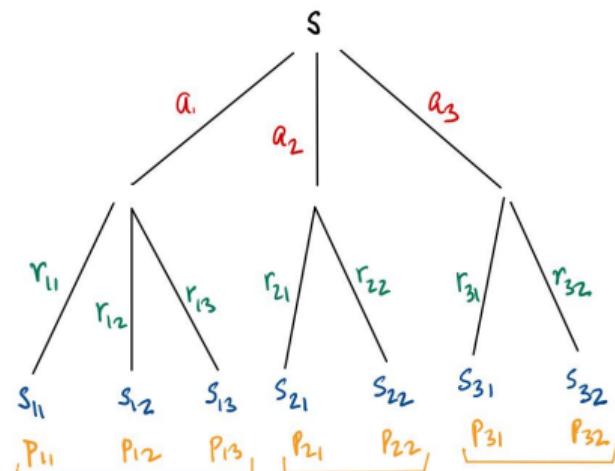
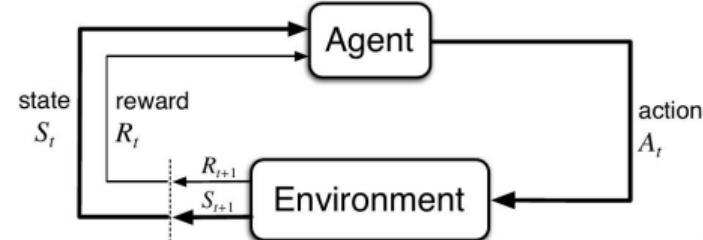
Advanced Machine Learning 2023

Markov Decision Processes

- Set of states S , actions A , rewards R
- At time t , agent in state S_t selects action A_t , moves to state S_{t+1} and receives reward R_{t+1}

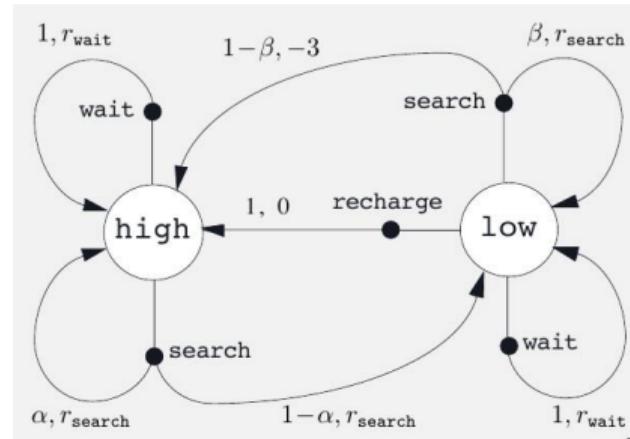
Trajectory $S_0, A_0, R_1, S_1, A_1, R_2, S_2, \dots$

- Probabilistic transition function:
 $p(s', r | s, a)$
 - Probability of moving to state s' with reward r if we choose a at s
 - For each (s, a) , $\sum_{s'} \sum_r p(s', r | s, a) = 1$
 - Backup diagram
- Typically assume finite MDPs — S , A and R are finite



MDP Example: Robot that collects empty cans

- State — battery charge: **high**, **low**
- Actions: **search** for a can, **wait** for someone to bring can, **recharge** battery
 - No **recharge** when **high**
- α , β , probabilities associated with change of battery state while searching
- 1 unit of reward per can collected
- $r_{\text{search}} > r_{\text{wait}}$ — cans collected while searching, waiting
- Negative reward for requiring rescue (**low** to **high** while searching)



s	a	s'	$p(s' s, a)$	$r(s, a, s')$
high	search	high	α	r_{search}
high	search	low	$1 - \alpha$	r_{search}
low	search	high	$1 - \beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-

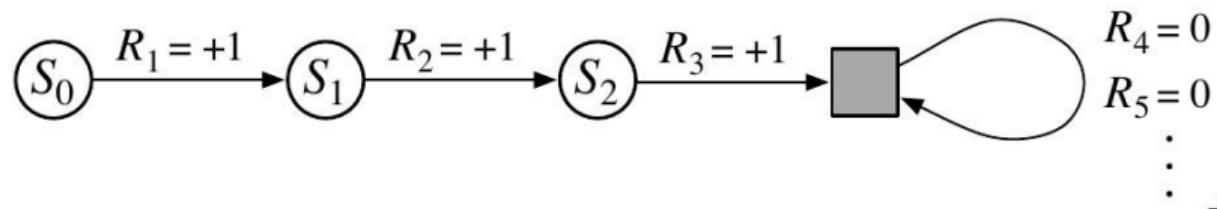
Long term rewards

- How do we formalize long term rewards?
- Assume that each trajectory is a finite **episode**
- Episode with T steps, expected reward at time t : $G_t \stackrel{\Delta}{=} R_{t+1} + R_{t+2} + \dots + R_T$
 - Each episode is independent: rewards are reset after each episode
- In some situations, trajectories may be (potentially) infinite
 - **Discounted** rewards: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, $0 \leq \gamma \leq 1$
- Inductive calculation of expected reward

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

Long term rewards

- Can make all episodes infinite by adding a self-loop with reward 0



- Allow $\gamma = 1$ only if sum converges to a finite number

- Alternatively, $G_t \triangleq \sum_{k=t+1}^T \gamma^{k-t-1} R_k$,

where we allow $T = \infty$ or $\gamma = 1$, but not both at the same time

- Example: If $T = \infty$, $R_k = +1$ for each k , $\gamma < 1$, then $G_t = \frac{1}{1-\gamma}$

Policies and value functions

- A **policy** π describes how the agent chooses actions at a state

- $\pi(a | s)$ — probability of choosing a in state s , $\sum_a \pi(a | s) = 1$

- State value function at s , following policy π

$$v_\pi(s) \triangleq \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right]$$

- Action value function on choosing a at s and then following policy π

$$q_\pi(s, a) \triangleq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

- Note that $v_\pi(s) = \sum_a \pi(a | s) q_\pi(s, a)$
- Goal is to find an optimal policy, that maximizes state/action value at every state

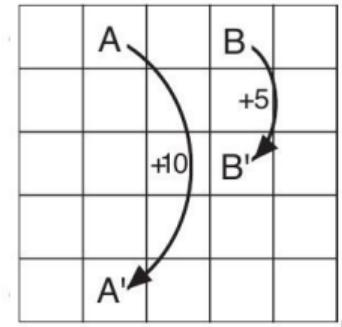
Bellman equation

- $v_\pi(s) \stackrel{\triangle}{=} \mathbb{E}_\pi[G_t | S_t = s]$
 $= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s]$
 $= \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']]$
 $= \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_\pi(s')]$

- Bellman equation relates state value at s to state values at successors of s
- Value function v_π is unique solution to the equation

Gridworld Example

- Actions in each cell are {N,S,E,W}, with usual interpretation
- Reward is 0, except at boundaries
- Colliding with boundary — position unchanged, reward -1
- Special squares A and B — all four actions move as indicated, with rewards +10 and +5, respectively
- Policy π — choose each action with uniform probability 0.25
- Solving Bellman equations, we obtain v_π for each square
- Values at boundary are negative
- Value at A is less than 10 because next move takes agent to boundary square with negative value
- Value at B is more than 5 because next move is to a square with positive value



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Optimal policies and value functions

- Compare policies π, π' : $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for every state s
- Optimal policy π_* , $\pi_* \geq \pi$ for every π
 - Always exists, but may not be unique
- Optimal state value function, $v_*(s) \triangleq \max_\pi v_\pi(s) = v_{\pi_*}(s)$
- Optimal action value function, $q_*(s, a) \triangleq \max_\pi q_\pi(s, a) = q_{\pi_*}(s, a)$
- Bellman optimality equation for v_*

$$\begin{aligned}v_*(s) &= \max_a q_{\pi_*}(s, a) \\&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\&= \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_*(s')]\end{aligned}$$

Bellman optimality equations

- $v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$
 $= \max_a \sum_{s',r} p(s',r | s,a) [r + \gamma v_*(s')]$

- Likewise, for action value function

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = t, A_t = a] \\ &= \sum_{s',r} p(s',r | s,a) [r + \max_{a'} \gamma q_*(s',a')] \end{aligned}$$

- For finite state MDPs, can solve explicitly for v_*
 - n states, n equations in n unknowns, (assuming we know p)
- However, n is usually large, computationally infeasible
 - State space of a game like chess or Go
- Instead, we will explore iterative methods to approximate v_*

Dynamic Programming for MDP: Policy and Value Iteration

Pranabendu Misra

based on slides by Madhavan Mukund

Advanced Machine Learning

2023

Policy evaluation

- Given a policy π , compute its state value function v_π
- Use the *Bellman equations*: $v_\pi(s) = \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_\pi(s')]$
 - For MDP with n states: n equations in n unknowns
 - Can solve to get v_π , but computationally infeasible for large n
- Instead, use the Bellman equations as an iterative update rule.
 - Initialize $v_\pi^0(s)$: set $v_\pi^0(\text{term}) = 0$ for terminal state term , arbitrary values for other s
 - Update v_π^k to v_π^{k+1} using: $v_\pi^{k+1}(s) = \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_\pi^k(s')]$
 - Stop when incremental change $\Delta = |v_\pi^{k+1} - v_\pi^k|$ is below threshold θ

We have now computed v_π approximately

Policy evaluation

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π , the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$

Policy evaluation example



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R_t = -1$
on all transitions

$k = 0$

v_k for the
random policy

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Policy improvement

- Using v_π , can we find a better policy π' ?
- Is there a state s where we can update $\pi(s)$ by a better action a ?
- Recall the action-value function

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')] \end{aligned}$$

- If $q_\pi(s, a) > v_\pi(s)$, modify π so that $\pi(s) = a$
- The new policy π' is strictly better

Policy improvement

Policy Improvement Theorem

For policies π, π' :

- If $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ for all s , then $\pi' \geq \pi$,
 - If $\pi' \geq \pi$ and $q_\pi(s, \pi'(s)) > v_\pi(s)$ for some s , then $v_{\pi'}(s) > v_\pi(s)$.
-
- $q_\pi(s, \pi'(s))$ is $q_\pi(s, a)$ for $a = \pi'(s)$
 - Proof of the theorem is not difficult for deterministic policies
 - The theorem extends to probabilistic policies also
 - Provides a basis to iteratively improve the policy

Policy iteration

- Start with an arbitrary policy π_0
- Use policy evaluation to compute v_{π_0}
- Use policy improvement to construct a better policy π_1
- Policy iteration: Alternate between policy evaluation and policy improvement



- Finite MDPs — can improve π only finitely many times,
 - Must converge to optimal policy (requires proof)
- Nested iteration — each policy evaluation is itself an iteration
 - Speed up by using v_{π_i} as initial state to compute $v_{\pi_{i+1}}$

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in \mathcal{S}$:

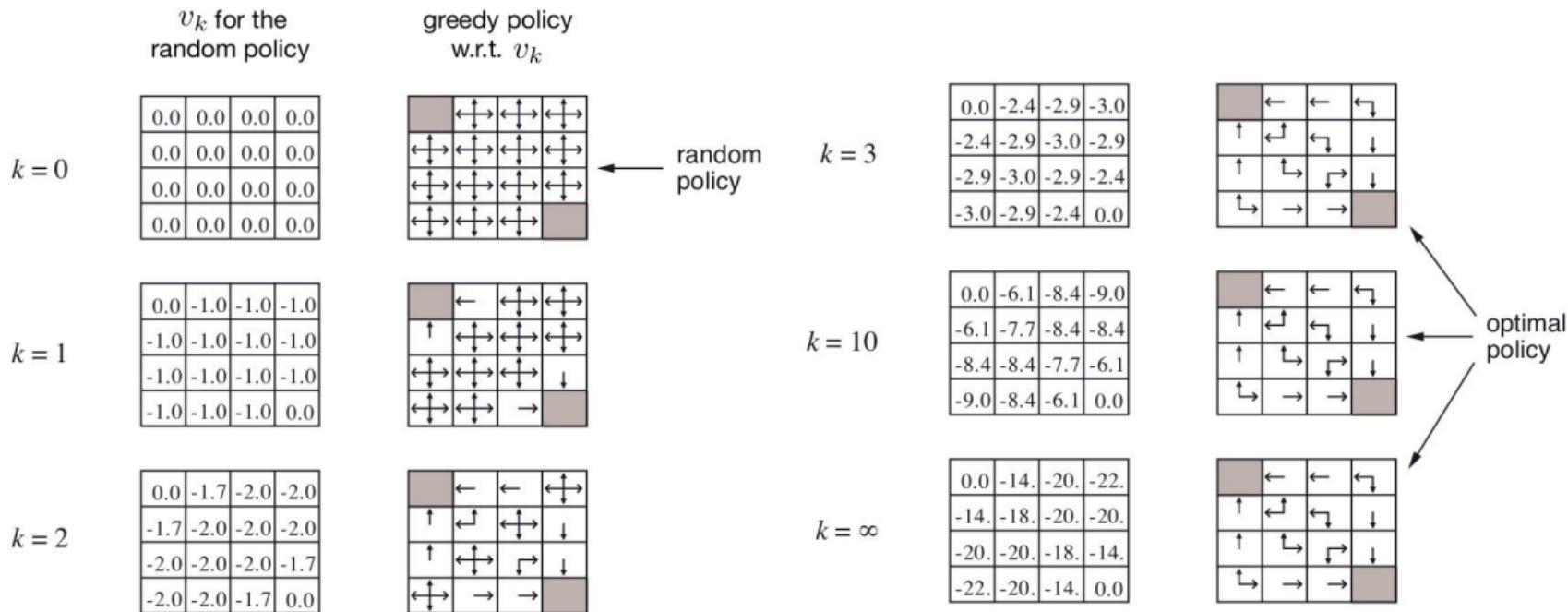
$$\text{old-action} \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

If $\text{old-action} \neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Policy Iteration



Value iteration

- Policy iteration — policy evaluation is expensive
- But even a single iteration in the computation of v_{π_k} is sufficient to point towards optimal actions for a state – enough for policy improvement
- **Value iteration:** Directly estimate the optimal value function at each state.

Iterative update to the value of a state :

$$v_{\pi_{k+1}}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi_k}(S_{t+1}) | S_t = s, A_t = a]$$

$$= \max_a \sum_{s',r} p(s', r | s, a) [r + \gamma v_{\pi_k}(s')]$$

- Again, stop when incremental change $\Delta = |v_{\pi_{k+1}} - v_{\pi_k}|$ is below threshold θ
- To compute π^* from v_{π^*} , at each state s simply take the action a that maximizes $q_{\pi^*}(s, a)$.

Value iteration

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

| $\Delta \leftarrow 0$

| Loop for each $s \in \mathcal{S}$:

| | $v \leftarrow V(s)$

| | $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

| | $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Dynamic programming

- In the literature, policy iteration and value iteration are referred to as **dynamic programming** methods
- Requires knowledge of the model — $p(s', r | s, a)$
- These algorithms are correct because of **Bellman Optimality Equation**, which states that if no improvements are possible then the current policy is optimal.
- How to combine policy evaluation and policy improvement is flexible
 - Value iteration is policy iteration with policy evaluation truncated to a single step
 - **Generalized policy iteration** — simultaneously maintain and update approximations of π_* and v_*
- **Asynchronous dynamic programming** for large state spaces

Monte Carlo Methods

Pranabendu Misra

based on slides by Madhavan Mukund

Advanced Machine Learning 2023

Monte Carlo methods

- If we know the model ($p(s, a, s', r)$), then use generalized policy iteration (dynamic programming) to approximate π_* , v_*
- What if the model is a (partial) black box?
- IDEA: Generate random episodes to estimate the given quantities
- Learning through **experience**
- **Monte Carlo** algorithms — compute estimates through random sampling

Monte Carlo policy evaluation — estimating v_π

- Estimate v_π for a given policy π
- Generate an episode following π , compute $v_\pi(s)$ backwards from end
- Average out values across episodes
- First-visit MC — compute average for first visit to s in each episode
- Every-visit MC — remove **Unless** condition

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in S_0, S_1, \dots, S_{t-1} :

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Action value estimate — $q_\pi(s, a)$

- In the absence of a model, we have to directly estimate action values $q_\pi(s, a)$
 - Policy improvement requires $q_\pi(s, a)$
 - $v_\pi(s) = \sum_a \pi(a | s) q_\pi(s, a)$
- Need to ensure that all pairs (s, a) are visited
- IDEA (**Exploring starts**): Each pair (s, a) has non-zero probability of being start of an episode
 - This is not realistic assumption.
- With exploring starts, algorithm for estimating q_π is similar to the one for v_π

Monte Carlo Policy Iteration

- As before, alternate between policy evaluation and policy improvement

$$\pi_0 \xrightarrow{\text{evaluate}} q_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluate}} q_{\pi_1} \xrightarrow{\text{improve}} \pi_2 \xrightarrow{\text{evaluate}} \dots$$

- Improve:** Given estimate q_{π_k} , update $\pi_{k+1}(s) = \arg \max_a q_{\pi_k}(s, a)$

- Evaluate:** Estimate q_{π_k} from π_k

- Iterate over large number of episodes to estimate average values
- Exploring starts

Monte Carlo Policy Iteration, estimating π_*

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

ε -soft policies

- To avoid exploring starts, use a version of ε greedy
- ε -soft policy
 - Let $\mathcal{A}(s)$ be set of actions available at state s
 - Choose non-greedy action with probability $\frac{\epsilon}{|\mathcal{A}(s)|}$ — uniform
 - Choose greedy action with probability $(1 - \varepsilon) + \frac{\epsilon}{|\mathcal{A}(s)|}$
 - This is an On-Policy method.
We use policy π to generate episodes to estimate q_π .

Monte Carlo Policy Iteration with ε -soft policies

On-policy first-visit MC control (for ε -soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε -soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)

$A^* \leftarrow \arg \max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon / |\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon / |\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Off policy methods

- Use a different policy b to generate episodes to estimate v_π
- **Coverage** If $\pi(a | s) > 0$ then $b(a | s) > 0$
- Consider the probability of a trajectory $A_t, S_{t+1}, A_{t+1}, \dots, S_T$ from S_t

- For π , $\pi(A_t | S_t)p(S_{t+1} | S_t, A_t)\pi(A_{t+1} | S_{t+1}) \cdots p(S_T | S_{T-1}, A_{T-1})$

$$= \prod_{k=t}^{T-1} \pi(A_k | S_k)p(S_{k+1} | S_k, A_k)$$

- For b , $b(A_t | S_t)p(S_{t+1} | S_t, A_t)b(A_{t+1} | S_{t+1}) \cdots p(S_T | S_{T-1}, A_{T-1})$

$$= \prod_{k=t}^{T-1} b(A_k | S_k)p(S_{k+1} | S_k, A_k)$$

- $p(s' | s, a)$ are unknown model probabilities

- Take ratio, these cancel out $\frac{\prod_{k=t}^{T-1} \pi(A_k | S_k)p(S_{k+1} | S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k | S_k)p(S_{k+1} | S_k, A_k)} = \frac{\prod_{k=t}^{T-1} \pi(A_k | S_k)}{\prod_{k=t}^{T-1} b(A_k | S_k)}$

Off Policy: Importance sampling

- Use ratio $\rho_{t:T} = \frac{\prod_{k=t}^{T-1} \pi(A_k | S_k)}{\prod_{k=t}^{T-1} b(A_k | S_k)}$ to “adjust” estimates learnt via b for this episode
 - Let G_t be an estimate learned by exploring using b
 - The corresponding estimate with respect to π for this episode is $\rho_{t:T} G_t$
- Then $v_\pi(S) = \frac{\sum_{t \in \mathcal{T}(S)} \rho_{t:T} G_t}{|\mathcal{T}(S)|}$, where $\mathcal{T}(S)$ is the set of trajectories starting with S in the episodes.
- **Importance sampling**
 - Generate episodes using b and compute adjusted estimates for q_π and v_π
 - Contribution of each episode is proportional to its likelihood with respect to π
- Variations — ordinary importance sampling (above), weighted importance sampling
- Off policy methods still to be fully analyzed

Temporal Difference Learning

Pranabendu Misra

Based on slides by Madhavan Mukund

Advanced Machine Learning 2023

Adding bootstrapping to Monte Carlo methods

- Dynamic programming: use generalized policy iteration to approximate π_* , v_*
 - Bootstrap from an initial estimate through incremental updates
 - Need to know the model
- Monte Carlo methods: random exploration to estimate π_* , v_*
 - Works with black box models
 - Need to complete an episode before applying updates
- Temporal Difference (TD) learning
 - Learn immediately from the ongoing episode.

From Monte Carlo to TD

- Monte Carlo update for non-stationary environments
 - $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$, $\alpha \in (0, 1)$ is a constant.
 $G_t - V(S_t)$ is like the current gradient of $V(S_t)$
 - G_t is available only after end of the episode — calculate backwards starting from G_T
- Instead
 - Observe that, $R_{t+1} + \gamma V(S_{t+1})$ is our current estimate for G_t .
 - Revised update rule: $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
 - R_{t+1} is available after choosing A_t
 - Use current estimate for $V(S_{t+1})$
 - Update $V(S_t)$ on the fly, as the episode evolves
- Also called TD(0), because it has zero lookahead
 - More generally, can look ahead n steps to update, TD(n)
 - Most general version is called TD(λ).

TD(0) algorithm for policy evaluation

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

$A \leftarrow$ action given by π for S

 Take action A , observe R, S'

$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

 until S is terminal

TD(0) example: Driving home from work

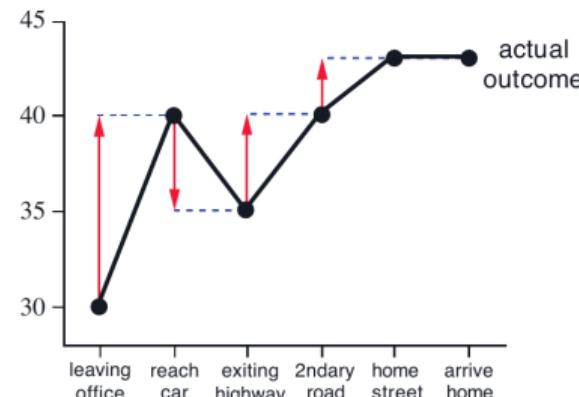
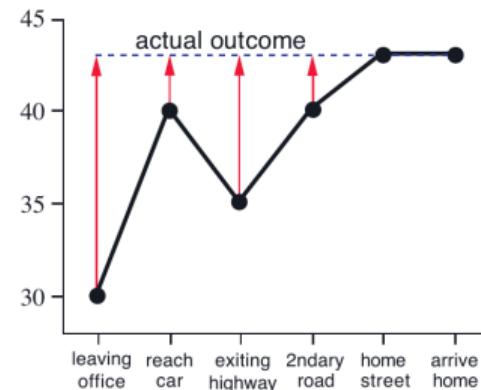
- Predict how long it will take you to drive home from work
- Leave office on Friday at 6:00 pm, initial estimate 30 minutes from now
- Reach car at 6:05 pm, raining, revise estimate to 35 minutes from now, total 40
- At 6:20 pm, complete highway stretch smoothly, cut estimate of total to 35 minutes
- Stuck behind slow truck, follow till 6:40 pm
- Turn off onto home street, arrive at 6:43 pm

<i>State</i>	<i>Elapsed Time (minutes)</i>	<i>Predicted Time to Go</i>	<i>Predicted Total Time</i>
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

TD(0) example: Driving home

- Rewards: elapsed time on each leg
- No discounting: $\gamma = 1$, return at a state is actual time remaining

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

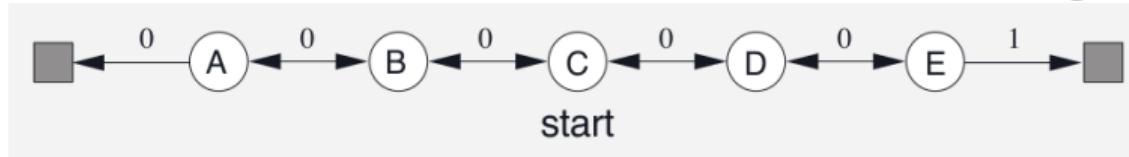


MC vs TD(0) updates to the state values.

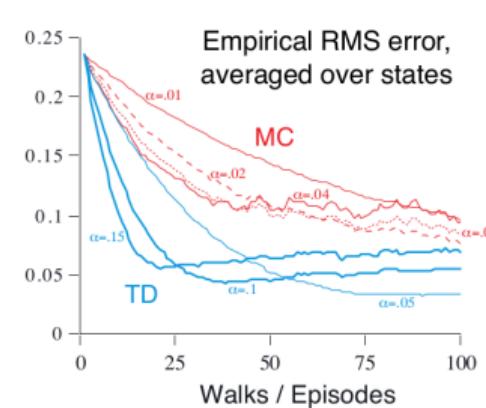
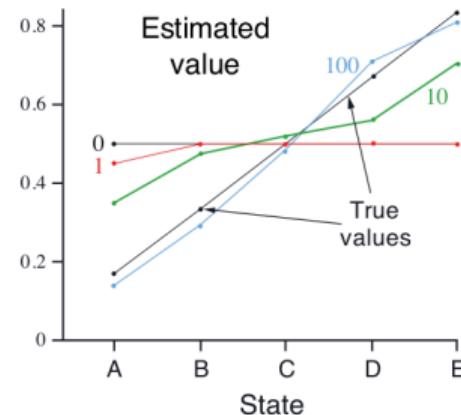
MC updates happen only at the end of the episode, TD(0) updates happen immediately.

Comparing MC and TD(0)

- Markov Reward Process: MDP without actions, environment changes automatically.



- Reward is probability of reaching right hand side: $1/6, 2/6, 3/6, 4/6, 5/6$



Updates to state values and RMS error VS num of episodes.

Comparing MC and TD(0) ...

Predict the values of states A and B , given the following eight episodes

A, 0, B, 0

B, 1

B, 1

B, 1

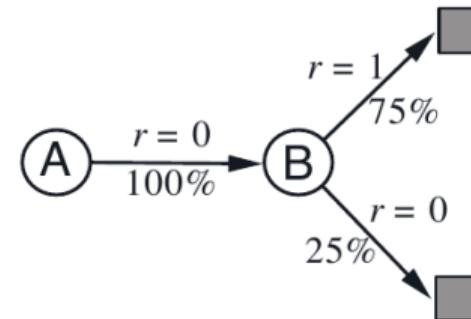
B, 1

B, 1

B, 1

B, 0

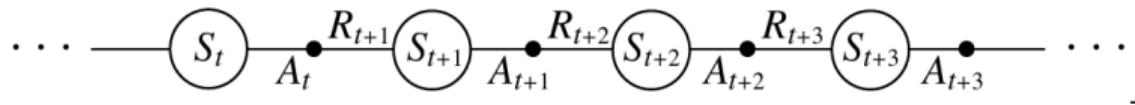
- $V(B) = 6/8 = 0.75$
- What about $V(A)$?
- MC: only one episode with A with total reward 0, hence $V(A) = 0$
- TD(0): $V(A) = 0.75$, because based on data, we always go from A to B with reward 0, and $V(B) = 0.75$.



SARSA: On policy TD control, estimating π_*

- For π_* , better to estimate q_π rather than v_π

- Structure of an episode



- Use the following update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

- Update uses $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, hence the name SARSA
- As with Monte Carlo estimation, use ϵ -soft policies to balance exploration and exploitation

SARSA algorithm on-policy TD control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Loop for each step of episode:

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$S \leftarrow S'; A \leftarrow A'$;

 until S is terminal

Q-learning: Off policy TD control, estimating π_*

- Directly estimate q_* independent of policy being followed

- Use the following update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

- Observe that we use the *greedy policy* at S_{t+1} , unlike SARSA which uses π .
This comes from the *Bellman equations*.
- Underlying policy still needs to be designed to visit all state-action pairs
- With suitable assumptions, Q-learning provably converges to q_*

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$$S \leftarrow S'$$

 until S is terminal

Summary

- Temporal difference methods combine bootstrapping with Monte Carlo exploration of state space
- SARSA is a TD(0) algorithm for on-policy control — estimating π_*
- Q-learning is an off-policy algorithm that provably converges to q_*
- TD-based approaches apply beyond reinforcement learning
 - General methods to make long term predictions about dynamical systems
- Theoretical properties such as convergence still an area of research

Deep Reinforcement Learning

Pranabendu Misra

Advanced Machine Learning 2023

Deep Reinforcement Learning

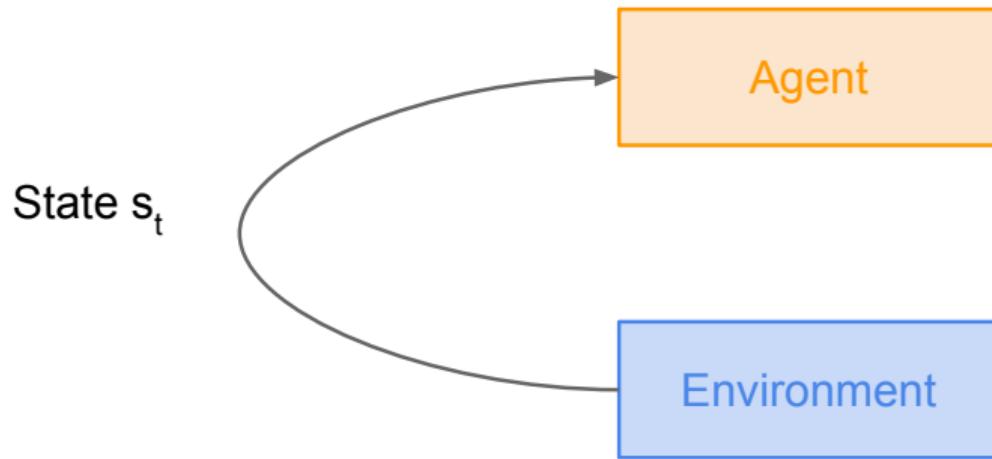
We use slides by Fei-Fei Li & Justin Johnson & Serena Yeung
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Reinforcement Learning

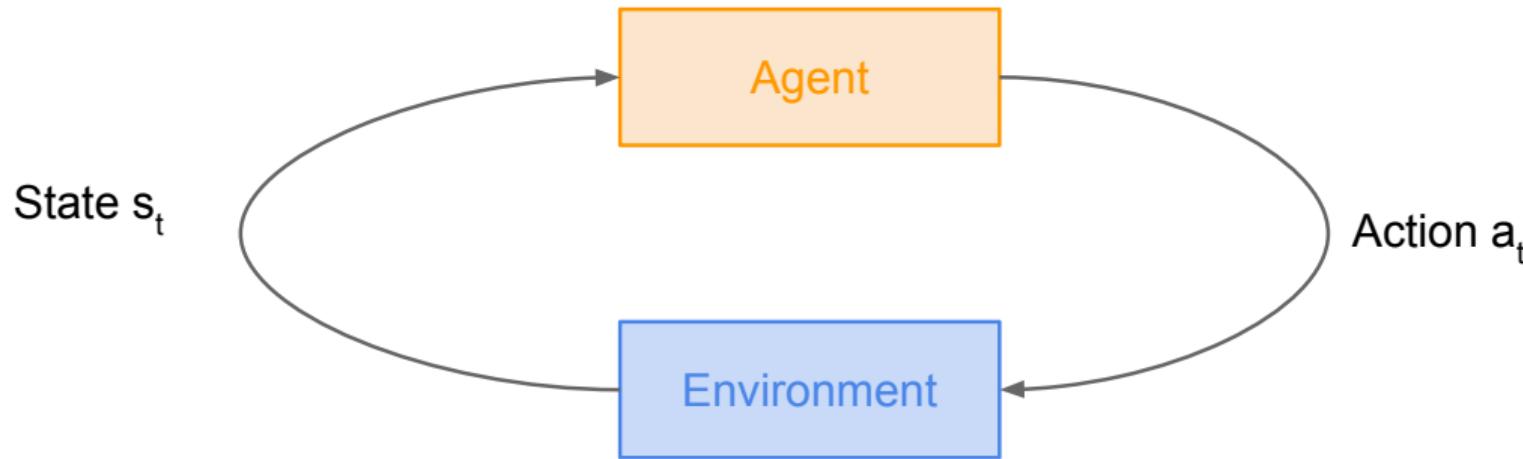
Agent

Environment

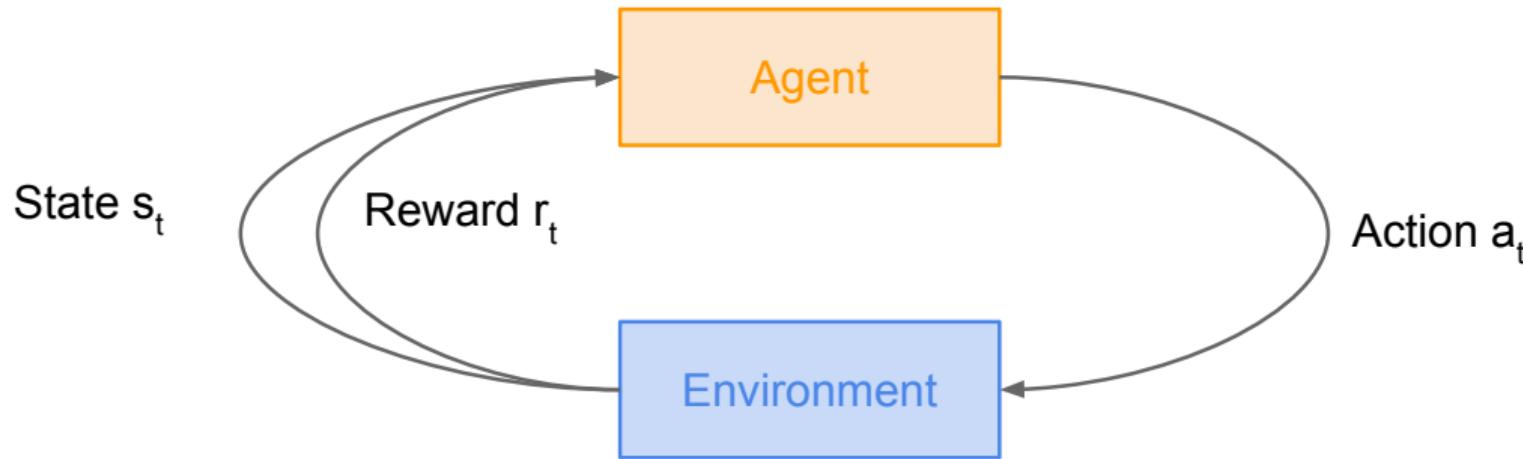
Reinforcement Learning



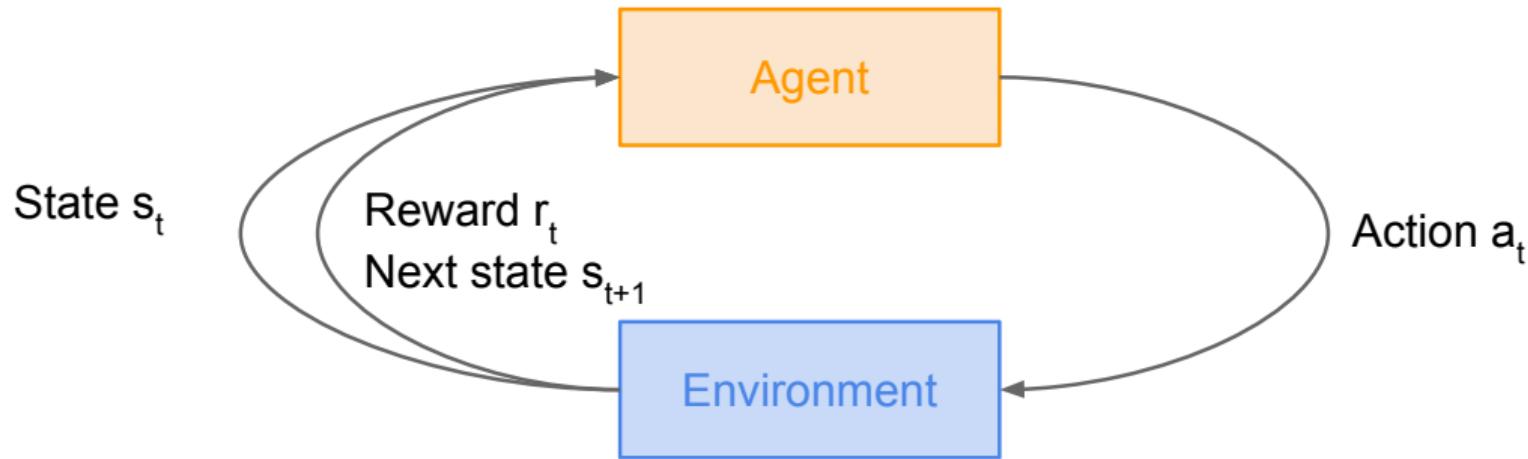
Reinforcement Learning



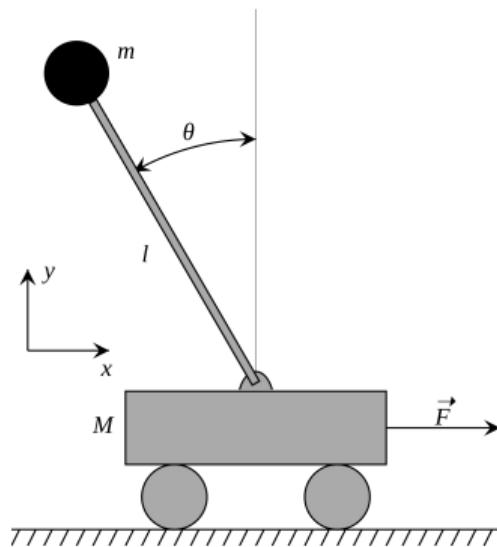
Reinforcement Learning



Reinforcement Learning



Cart-Pole Problem



Objective: Balance a pole on top of a movable cart

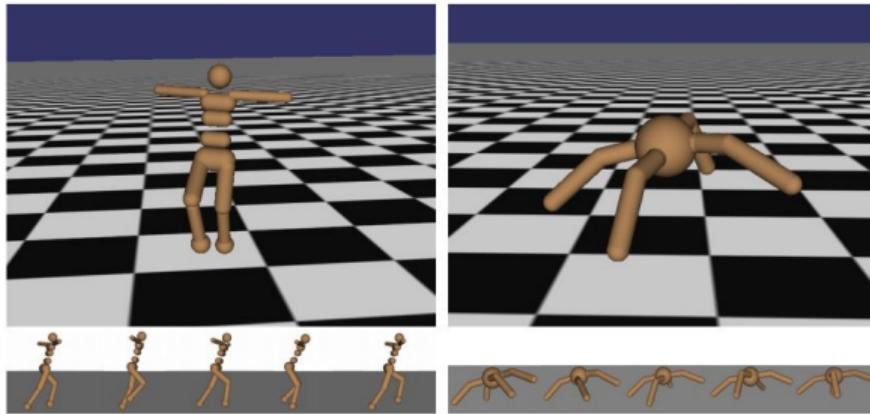
State: angle, angular speed, position, horizontal velocity

Action: horizontal force applied on the cart

Reward: 1 at each time step if the pole is upright

This image is CC0 public domain

Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints

Action: Torques applied on joints

Reward: 1 at each time step upright + forward movement

Figures copyright John Schulman et al., 2016. Reproduced with permission.

Atari Games



Objective: Complete the game with the highest score

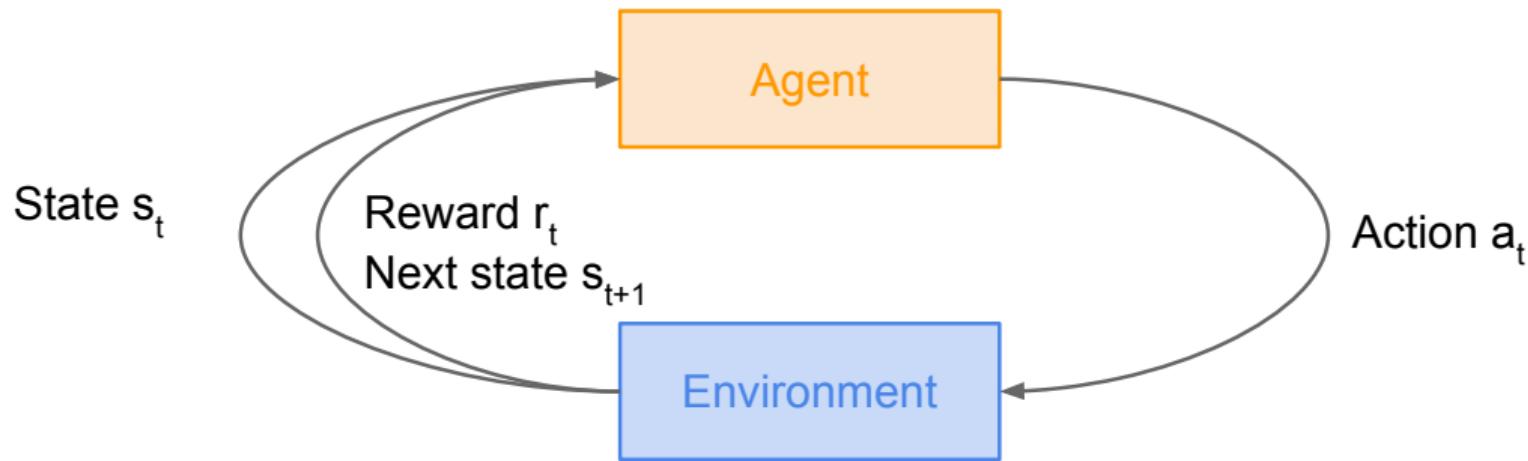
State: Raw pixel inputs of the game state

Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

How can we mathematically formalize the RL problem?



Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property:** Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

\mathcal{S} : set of possible states

\mathcal{A} : set of possible actions

\mathcal{R} : distribution of reward given (state, action) pair

\mathbb{P} : transition probability i.e. distribution over next state given (state, action) pair

γ : discount factor

Markov Decision Process

- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
 - Agent selects action a_t
 - Environment samples reward $r_t \sim R(\cdot | s_t, a_t)$
 - Environment samples next state $s_{t+1} \sim P(\cdot | s_t, a_t)$
 - Agent receives reward r_t and next state s_{t+1}
- A policy π is a function from S to A that specifies what action to take in each state
- **Objective:** find policy π^* that maximizes cumulative discounted reward:
$$\sum_{t \geq 0} \gamma^t r_t$$

The optimal policy π^*

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

How good is a state?

The **value function** at state s , is the expected cumulative reward from following the policy from state s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a , is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s, a)$. E.g. a neural network!

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning!**

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

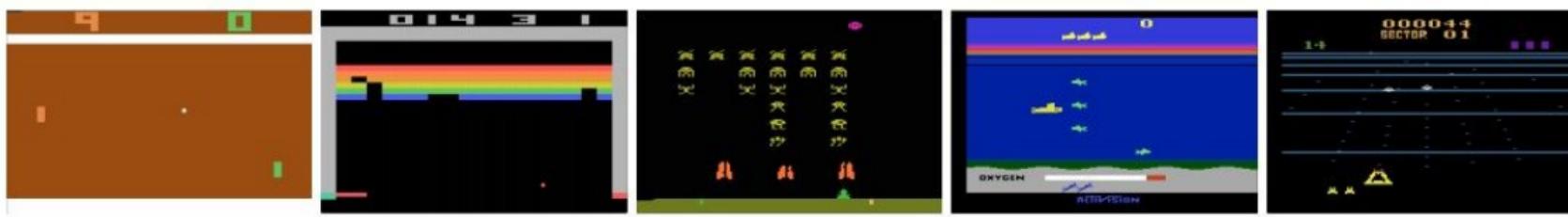
Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Case Study: Playing Atari Games



Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state

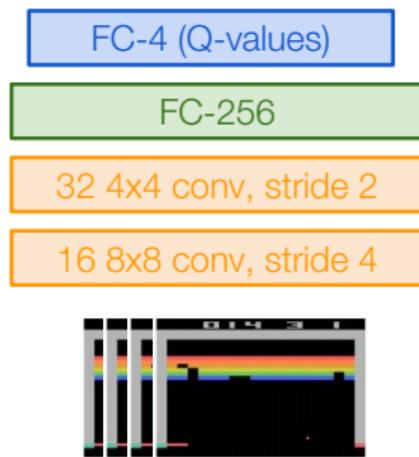
Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

Q-network Architecture

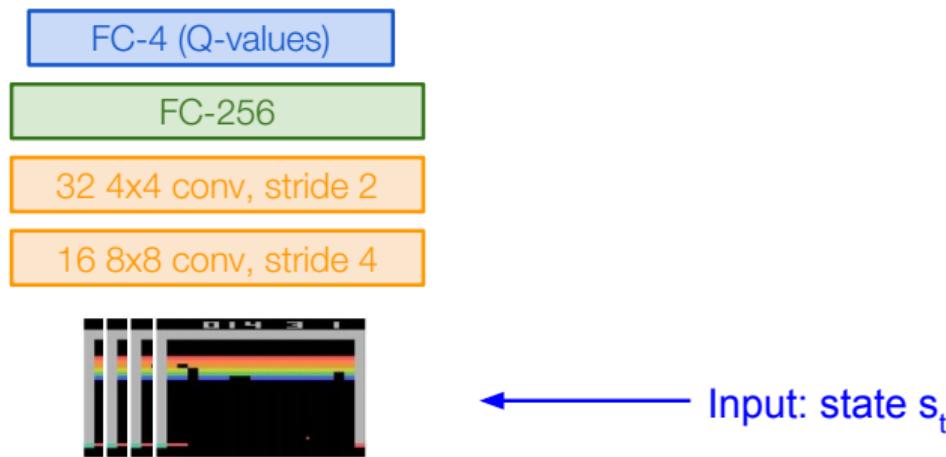
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

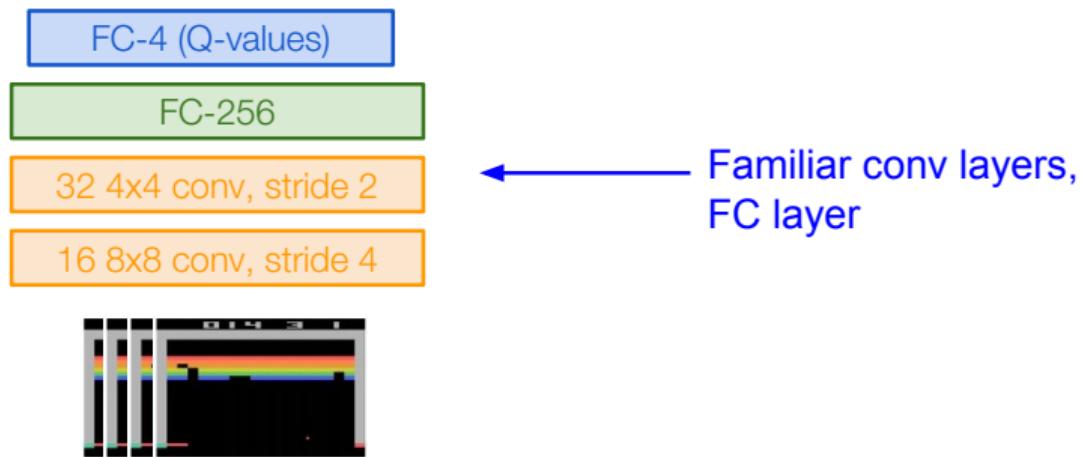
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

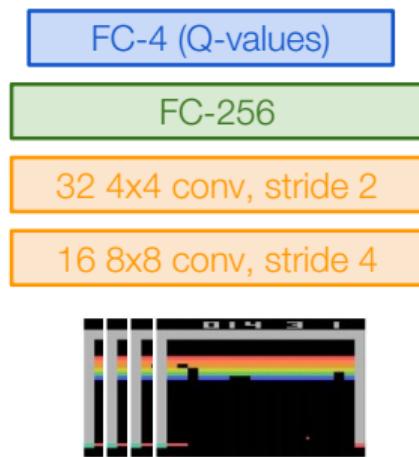
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

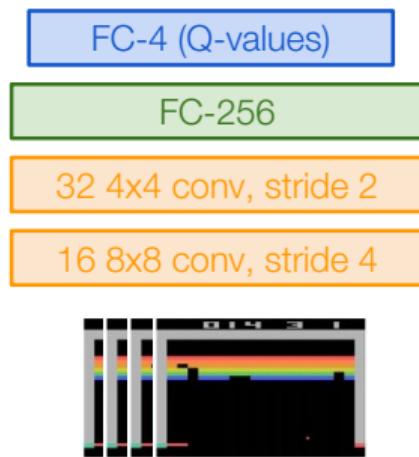
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1), Q(s_t, a_2), Q(s_t, a_3), Q(s_t, a_4)$

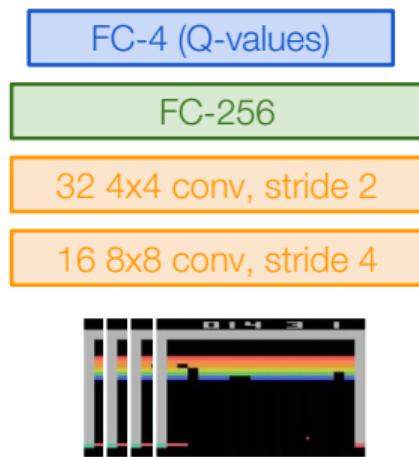
Number of actions between 4-18 depending on Atari game

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

A single feedforward pass
to compute Q-values for all
actions from the current
state => efficient!



Last FC layer has 4-d
output (if 4 actions),
corresponding to $Q(s_t, a_1), Q(s_t, a_2), Q(s_t, a_3), Q(s_t, a_4)$

Number of actions between 4-18
depending on Atari game

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Training the Q-network: Loss function (from before)

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
 - Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples
- Each transition can also contribute
to multiple weight updates
=> greater data efficiency

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N ← Initialize replay memory, Q-network
 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 3
end for
end for

Putting it together: Deep Q-Learning with Experience Replay

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**

← Play M episodes (full games)

 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 3

end for

end for

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for

Initialize state
(starting game
screen pixels) at the
beginning of each
episode

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for



For each timestep t
of the game

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for

With small probability,
select a random
action (explore),
otherwise select
greedy action from
current policy

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for



Take the action (a_t) ,
and observe the
reward r_t and next
state s_{t+1}

Putting it together: Deep Q-Learning with Experience Replay

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 3

end for

end for



Store transition in
replay memory

Putting it together: Deep Q-Learning with Experience Replay

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 $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

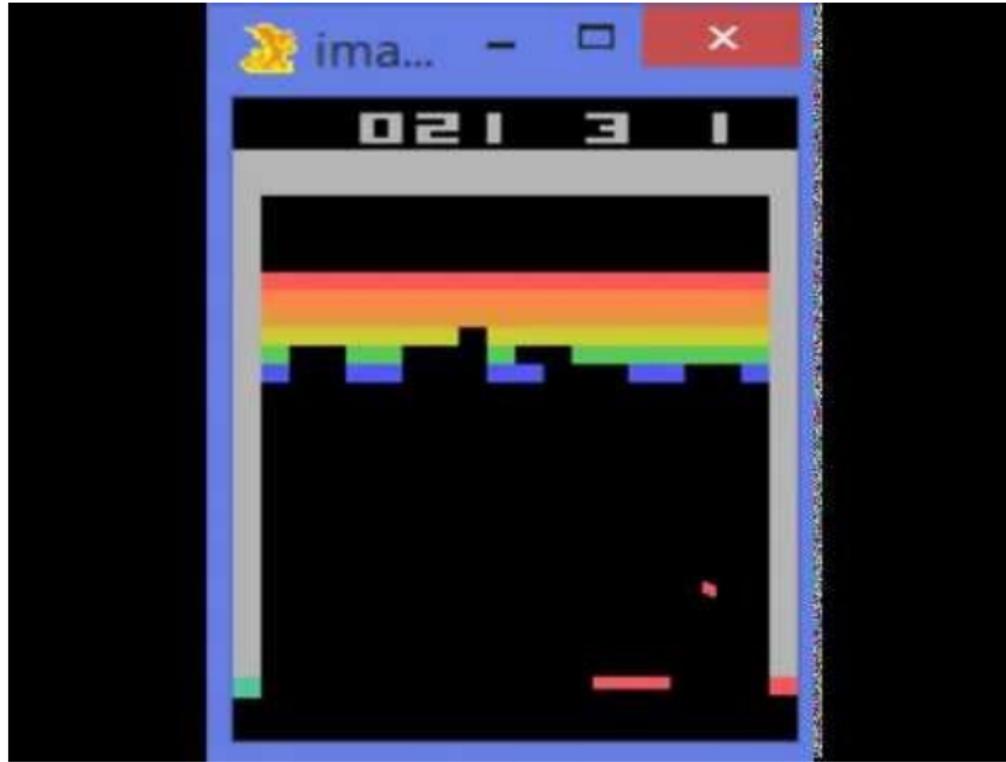
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Experience Replay:
Sample a random
minibatch of transitions
from replay memory
and perform a gradient
descent step



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

Video by Károly Zsolnai-Fehér. Reproduced with permission.

References:

- http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf
- <https://www.youtube.com/watch?v=lvoHnicueoE>

Advanced Machine Learning

Bayesian Optimization

Based on Slides by
Sourish Das and Madhavan Mukund

Chennai Mathematical Institute

2022

Pranabendu Misra

cmi

Introduction

Bayesian optimization (BayesOpt) is a class of ML optimization methods focused on solving the problem

$$\max_{\mathbf{x} \in A} f(\mathbf{x}),$$

where

- ▶ $\mathbf{x} \in \mathbb{R}^d$, typically $d \leq 20$
- ▶ Typically $A = \{\mathbf{x} \in \mathbb{R}^d | a_i \leq x_i \leq b_i\}$ is a hyper-rectangle
- ▶ f is expensive to evaluate.

Ex: f is a deep network model and with L many layers and q many nodes in each layer; $L = 2, 3, 4, \dots$; $q = 2, 3, 4, \dots$; so $\mathbf{x} = (L, q)$ and $f = RMSE$ in validation dataset



Nature of f

- ▶ f is expensive to evaluate – each evaluation may that maybe performed may take substantial amount of time and/or monetary cost (e.g., buying cloud computing power)
 - ▶ f lacks known special structure like concavity or linearity
 - ▶ When we evaluate f , we observe on $f(\mathbf{x})$ and no first and second order derivatives available
 - ▶ so gradient descent type algorithms are not possible
 - ▶ f is a '**black box**.'
- ▶ **Goal:** Find a global rather than local optimum.

cmi

Overview of BayesOpt

- ▶ BayesOpt is designed for black-box derivative free global optimization.
- ▶ BayesOpt consists of two main components:
 1. Bayesian statistical model for modeling the objective function f
 2. Acquisition function for deciding where to sample next.

cmi

Basic pseudo-code for Bayesian optimization

- ▶ Place a Gaussian process prior model on f
- ▶ Set $n = n_0$, observe f at n_0 different points, i.e.,
 $f(\mathbf{x}_1), f(\mathbf{x}_2), \dots, f(\mathbf{x}_{n_0})$
- ▶ while $n \leq N$ do
 - 1. Update the posterior probability distribution on f
 - 2. Let \mathbf{x}_n be a maximizer of the acquisition α function over \mathbf{x}
- Note Acquisition function $\alpha(\mathbf{x})$ is computed using the current posterior distribution.
- 3. Observe $y_n = f(\mathbf{x}_n)$
- 4. $n = n + 1$
- ▶ end while
- ▶ Return a solution: the point evaluated with the largest $f(\mathbf{x})$

smallest **cmi**

- we have $(x_i, y_i = f(x_i))$ for $i \in \{1, 2, \dots, n\}$

- Assume we have a multivariate gaussian dist-

$$[y_1, y_2, \dots, y_n] \sim N(\mu, \Sigma)$$

$$\mu = [m_1, \dots, m_n]$$

mean

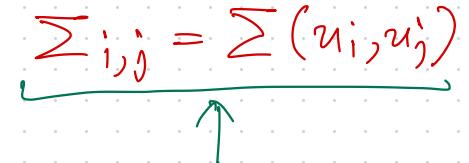
$$\Sigma = \begin{bmatrix} \Sigma_{1,1} & \Sigma_{1,2} & \dots & \Sigma_{1,n} \\ \Sigma_{2,1} & \Sigma_{2,2} & \dots & \Sigma_{2,n} \\ \vdots & & & \\ \Sigma_{n,1} & \Sigma_{n,2} & \dots & \Sigma_{n,n} \end{bmatrix}$$

$\Sigma_{i,j}$ is the Covariance of y_i and y_j

- We assume that f can be approximated by a multivariate gaussian dist.

Co-variances

Gaussian process $\equiv n \rightarrow \infty$

- We choose some suitable initial value for μ_i
 Mean function $\mu(u_i) = \mu^i$ Typically $\mu^i = 0$
- The choice of the co-variances Σ_{ij} is more important
 - covariance function or Kernel $\Sigma_{ij} = \sum(u_i, u_j)$

 - Gaussian Kernel

$$\sum(u_i, u_j) = \alpha e^{-\left(\frac{\|u_i - u_j\|^2}{l^2}\right)}$$

A popular choice

directly compute covariances without evaluating
 $y_i = f(u_i)$

- we have $(x_i, y_i = f(x_i))$ for $i \in \{1, 2, \dots, n\}$
 - Assume we have a multivariate gaussian dist-
- $$[y_1, y_2, \dots, y_n] \sim N(\mu, \Sigma)$$
- if we had x_1, x_2, \dots, x_{n-1} then we get
a Prob dist on possible values of x_n

$$y_n | [x_1, \dots, x_{n-1}] \sim N(\hat{\mu}_n, \hat{\sigma}_n^2)$$

$$\hat{\mu}_n = [\sum_{n,1}, \dots, \sum_{n,n-1}] \Sigma^{-1} [y_i - \mu_i]_{i=1 \dots n-1} + \mu_n$$

$$\hat{\sigma}_n^2 = \sum_{n,n} - [\sum_{n,1}, \dots, \sum_{n,n-1}] \Sigma^{-1} [\sum_{1,n}, \dots, \sum_{n-1,n}]$$

Posterior Prob Distribution

Modeling objective function with GP Regression

- ▶ Consider the following

$$\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}$$

- ▶ Represents $f(\mathbf{x})$ as

$$f(\mathbf{x}) = \sum_{j=1}^K \phi_j(\mathbf{x})\beta_j = \boldsymbol{\phi}\boldsymbol{\beta},$$

we say $\boldsymbol{\phi}$ is a basis system for $f(\mathbf{x})$, where $\phi_j(\mathbf{x})$ is completely known.

- ▶ Problem is $\boldsymbol{\beta}$ is unknown - hence we estimate $\boldsymbol{\beta}$.

cmi

Modeling objective function with GP Regression

Can be
infinite

- ▶ We are writing the function with its basis expansion

$$\mathbf{y} = \phi\beta + \epsilon$$

- ▶ The basis ϕ is fully known, such as
 - ▶ $\phi = \{1, \sin(\omega\mathbf{x}), \cos(\omega\mathbf{x}), \sin(2\omega\mathbf{x}), \cos(2\omega\mathbf{x}) \dots\}$, ω is known
 - ▶ $\phi = \{1, \exp(-\lambda_1(\mathbf{x} - c_1)^2), \exp(-\lambda_2(\mathbf{x} - c_2)^2) \dots\}$
- ▶ Problem is β is unknown - hence we estimate β .

cmi

Bayesian method

- ▶ Model:

$$\mathbf{y} = f(\mathbf{x}) + \epsilon$$

$$\epsilon \sim \mathbf{N}(0, \sigma^2 \mathbf{I}) \implies \mathbf{y} \sim \mathbf{N}(f(\mathbf{x}), \sigma^2 \mathbf{I}),$$

$$f(\mathbf{x}) = \phi\beta = \sum_{k=1}^K \phi_k(\mathbf{x})\beta_k + \sum_{k=K+1}^{\infty} \phi_k(\mathbf{x})\beta_k,$$

where $|\sum_{k=K+1}^{\infty} \phi_k(\mathbf{x})\beta_k| < \epsilon$; $\epsilon \geq 0$

- ▶ β is unknown and we want to estimate

Assuming β 's are uncorrelated random variable and $\phi_k(\mathbf{x})$ are known deterministic real-valued functions.

- ▶ Then due to **Kosambi-Karhunen-Loeve** theorem, we can say that $f(\mathbf{x})$ is a random realisation from a stochastic process.

Gaussian Process Prior

- ▶ As $f(\mathbf{x})$ is a stochastic process, if we assume $\boldsymbol{\beta} \sim \mathbf{N}(0, \sigma^2 \mathbf{I})$ then $f(\mathbf{x}) = \boldsymbol{\phi}\boldsymbol{\beta}$ follow Gaussian process.
- ▶ Since $f(\mathbf{x})$ is unknown function; therefore induced process on $f(\mathbf{x})$ is known as '**Gaussian Process Prior**'.

Prior on $\boldsymbol{\beta}$:

$$p(\boldsymbol{\beta}) \propto \exp\left(-\frac{1}{2\sigma^2}\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

Induced Prior on $f = \boldsymbol{\phi}\boldsymbol{\beta}$:

$$p(f) \propto \exp\left(-\frac{1}{2\sigma^2}\boldsymbol{\beta}^T\boldsymbol{\phi}^T\mathbf{K}^{-1}\boldsymbol{\phi}\boldsymbol{\beta}\right)$$

cmi

Gaussian Process Prior

- ▶ The prior mean and covariance of $f(\mathbf{x})$ are given by

$$\mathbf{E}[f(\mathbf{x})] = \phi(\mathbf{x})\mathbf{E}[\boldsymbol{\beta}] = \phi\boldsymbol{\beta}_0$$

$$\begin{aligned}\text{cov}[f(\mathbf{x})] &= \mathbf{E}[f(\mathbf{x}).f(\mathbf{x}')^T] = \phi(\mathbf{x})\mathbf{E}[\boldsymbol{\beta}\boldsymbol{\beta}^T]\phi(t')^T \\ &= \sigma^2\phi(\mathbf{x})\phi(\mathbf{x}')^T = \mathbf{K}(\mathbf{x}, \mathbf{x}')\end{aligned}$$

$$f(\mathbf{x}) \sim \mathcal{N}_n(\phi(\mathbf{x})\boldsymbol{\beta}_0, \mathbf{K}(\mathbf{x}, \mathbf{x}')), \quad \epsilon \sim \mathcal{N}_n(0, \sigma^2 \mathbf{I})$$

$$y(\mathbf{x}) \sim \mathcal{N}_n\left(\phi(\mathbf{x})\boldsymbol{\beta}_0, \mathbf{K}(\mathbf{x}, \mathbf{x}') + \sigma^2 \mathbf{I}\right)$$

cmi

Gaussian Process Prior

- ▶ If $\beta_0 = 0$ then

$$\mathbf{E}[f(\mathbf{x})] = \phi(\mathbf{x})E[\beta] = \phi\beta_0 = 0$$

$$f(\mathbf{x}) \sim \mathcal{N}_n(\mathbf{0}, \mathbf{K}(\mathbf{x}, \mathbf{x}')), \quad \epsilon \sim \mathcal{N}_n(0, \sigma^2 \mathbf{I})$$

$$y(\mathbf{x}) \sim \mathcal{N}_n\left(\mathbf{0}, \mathbf{K}(\mathbf{x}, \mathbf{x}') + \sigma^2 \mathbf{I}\right)$$

c^{m_i}

Gaussian Process Regression

- ▶ The estimated value of \mathbf{y} for a given \mathbf{x}_* is the mean (expected) value of the functions sampled from the posterior at that value of \mathbf{x}_* .
- ▶ Suppose $\mu(\mathbf{x}) = \phi(\mathbf{x})\beta_0 = 0$, then expected value of the estimate at a given \mathbf{x}_* is given by

$$\begin{aligned}\hat{f}(\mathbf{x}_*) &= \mathbf{E}(f(\mathbf{x}_*)|\mathbf{x}, \mathbf{y}) \\ &= \mathbf{K}(\mathbf{x}_*, \mathbf{x}) \cdot \underbrace{[\mathbf{K}(\mathbf{x}, \mathbf{x}) + \sigma^2 \cdot \mathbf{I}]^{-1}}_{\text{Matrix of order } n} \cdot \mathbf{y}\end{aligned}$$

- ▶ The time complexity of the matrix inversion is $\mathcal{O}(n^3)$

cmi

- mean function $\mu(u_i) = \mu_i$
- covariance function or Kernel $\Sigma_{ij} = \sum(u_i, u_j)$

- Gaussian Kernel

$$\sum(u_i, u_j) = \alpha e^{-\left(\frac{\|u_i - u_j\|^2}{l^2}\right)}$$

A popular choice

- How can we choose the parameters
 μ, α, l ?

Likelihood Method: Gaussian Process Prior Model

- ▶ Data model:

$$\mathbf{y}(\mathbf{x}) \sim \mathcal{N}_n\left(\mathbf{0}, \mathbf{K}_{\alpha, \rho}(\mathbf{x}, \mathbf{x}') + \sigma^2 \mathbf{I}\right)$$

- ▶ Static or Hyperparameters: $\theta = \{\alpha, \rho, \sigma^2\}$
- ▶ Likelihood function:

$$f(\beta | \mathbf{y}, \phi, \sigma^2) \propto (\sigma^2)^{-p/2} \exp\left(-\frac{1}{2\sigma^2} (\mathbf{y} - f)^T [\mathbf{K} + \sigma^2 \mathbf{I}]^{-1} (\mathbf{y} - f)\right)$$

- ▶ Negative Log-likelihood function:

$$l(\beta) \propto \frac{1}{2\sigma^2} \mathbf{y}^T [\mathbf{K} + \sigma^2 \mathbf{I}]^{-1} \mathbf{y}$$

cmi

Gaussian Process Prior Model

- ▶ Negative log-posterior:

$$p(\beta) \propto \frac{1}{2\sigma^2} \left(\mathbf{y}^T [\mathbf{K} + \sigma^2 \mathbf{I}]^{-1} \mathbf{y} + \beta^T \phi^T \mathbf{K}^{-1} \phi \beta \right)$$

- ▶ Hence the induced penalty matrix in the Gaussian process prior is identity matrix
- ▶ Still hyperparameters: $\theta = \{\alpha, \rho, \sigma^2\}$ are unknown.
- ▶ One can use **optimization** routine to estimate the MLE/MAP.

Pick θ that maximizes the probability of **cmi**
the seen data

Experiment with GP Regression

- ▶ Model:

$$y = \frac{\sin(x)}{x} + \epsilon,$$

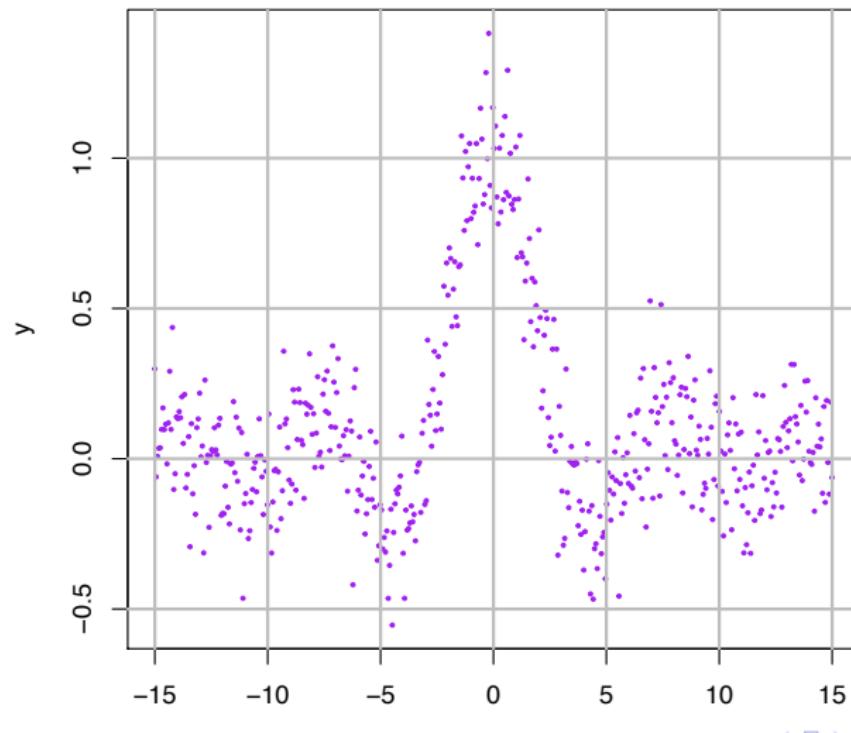
where $\epsilon \sim N(0, \tau)$.

- ▶ Simulate data from the above model and pretend we don't know the true function.
- ▶ **Objective** is to estimate/learn the function.

cmi

Experiment with GP Regression

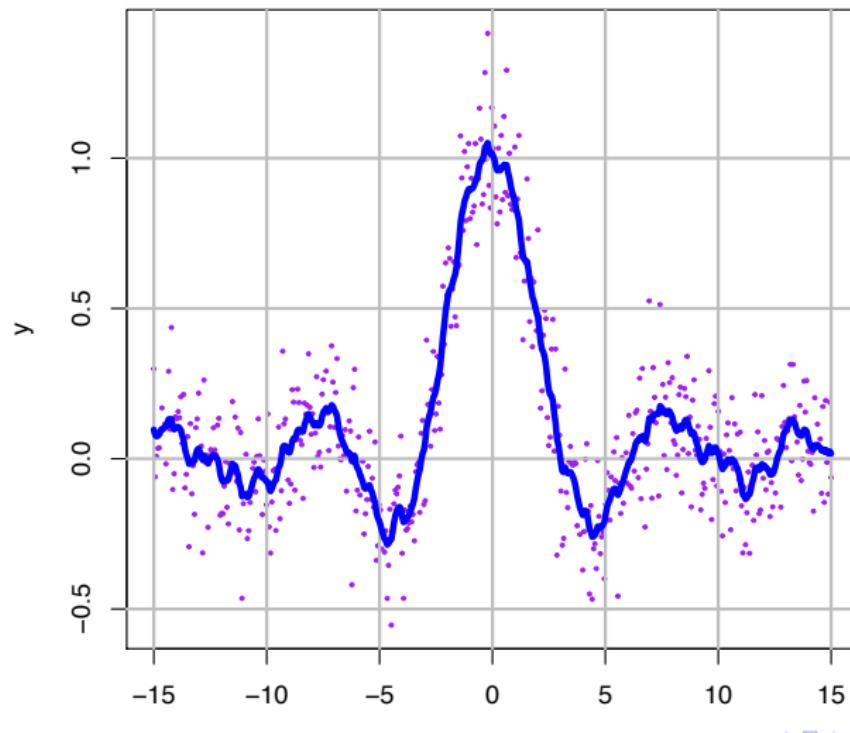
Objective is to estimate/learn the function.



cmi

Experiment with GP Regression

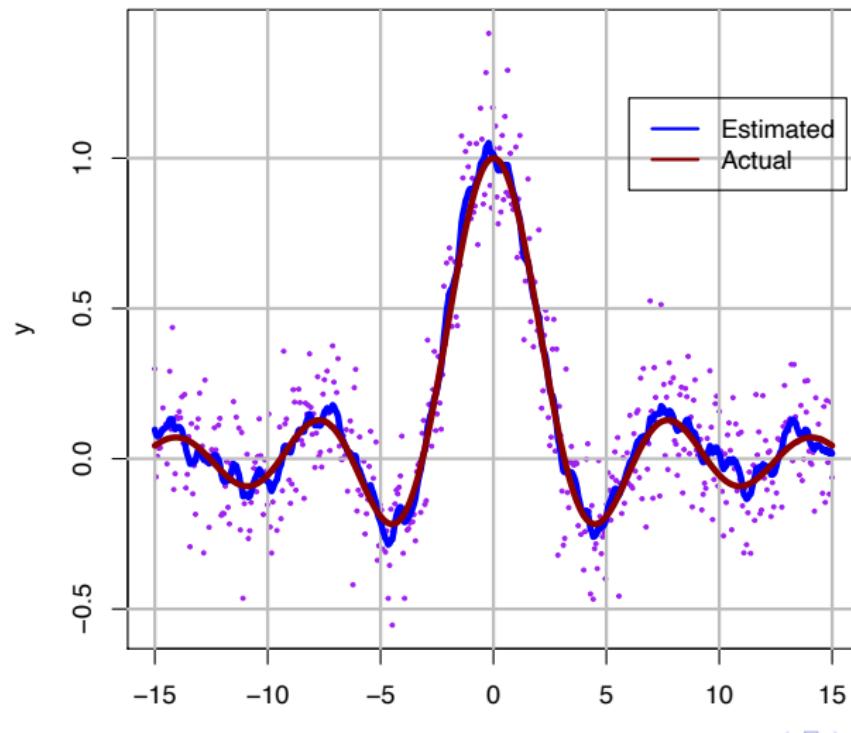
Objective is to estimate/learn the function.



cmi

Experiment with GP Regression

Objective is to estimate/learn the function.



cmi

Back to BayesOpt

Obs: Only need to model near the optimum, not everywhere

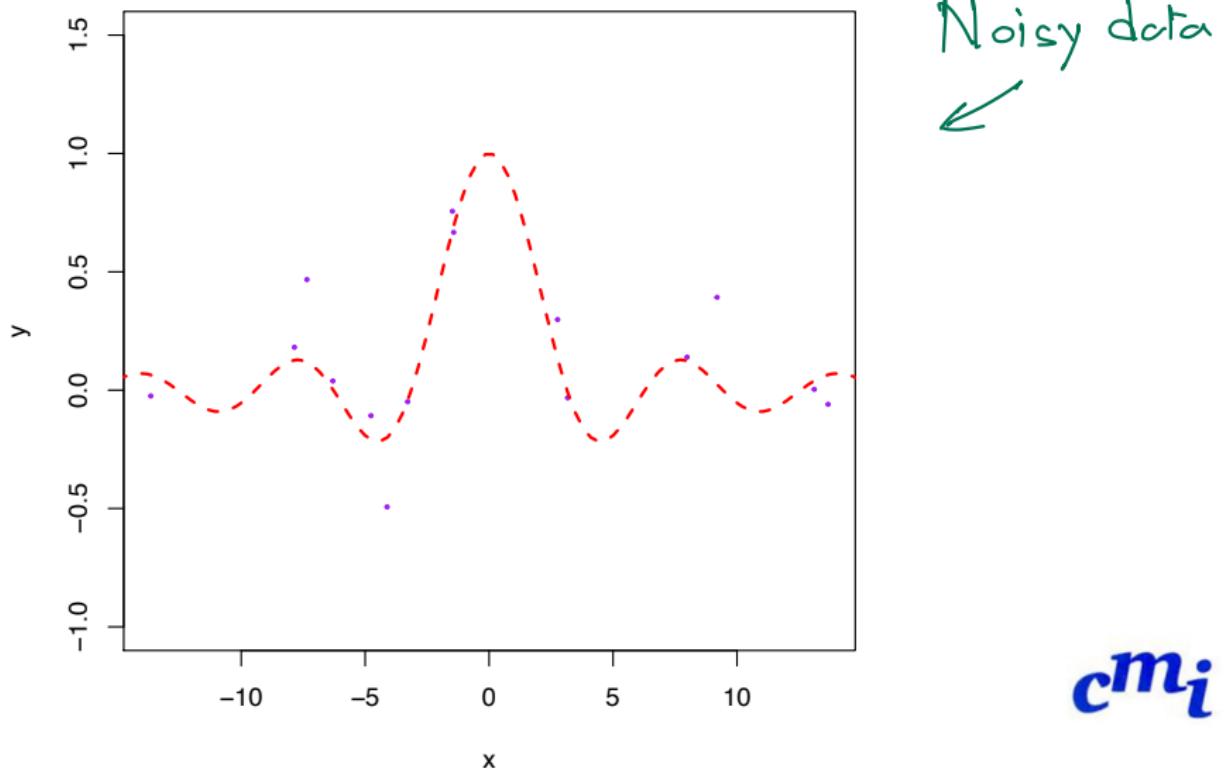
- ▶ As we modeled the objective function f by \hat{f}
- ▶ With f we try to predict the performance of the deep network model for a possible choices of hyper-parameter x .
- ▶ Next we model the acquisition function which recommend where will be the next point of hyper-parameter will be
 - ▶ One can use the \hat{f} directly as **acquisition function** or one can sample the acquisition function $\alpha(x)$ from the posterior distribution of f , i.e.,

where to sample $f(x)$
next

$$\alpha(x) \sim \mathcal{N}(\hat{f}, \text{cov}(\hat{f}))$$

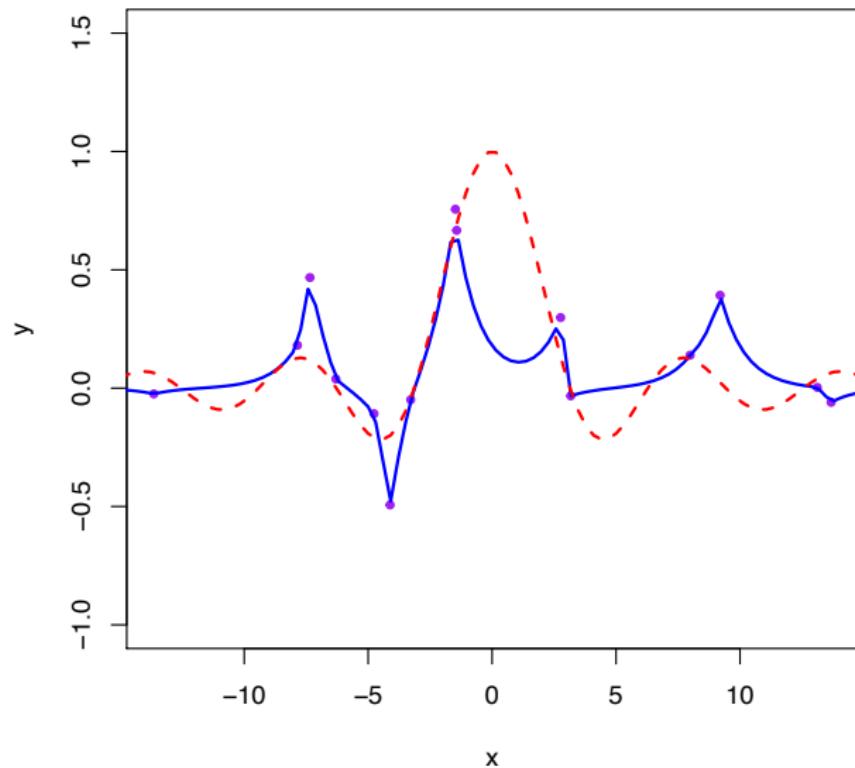
cmi

Bayesian Optimization: First Iteration (maximization)

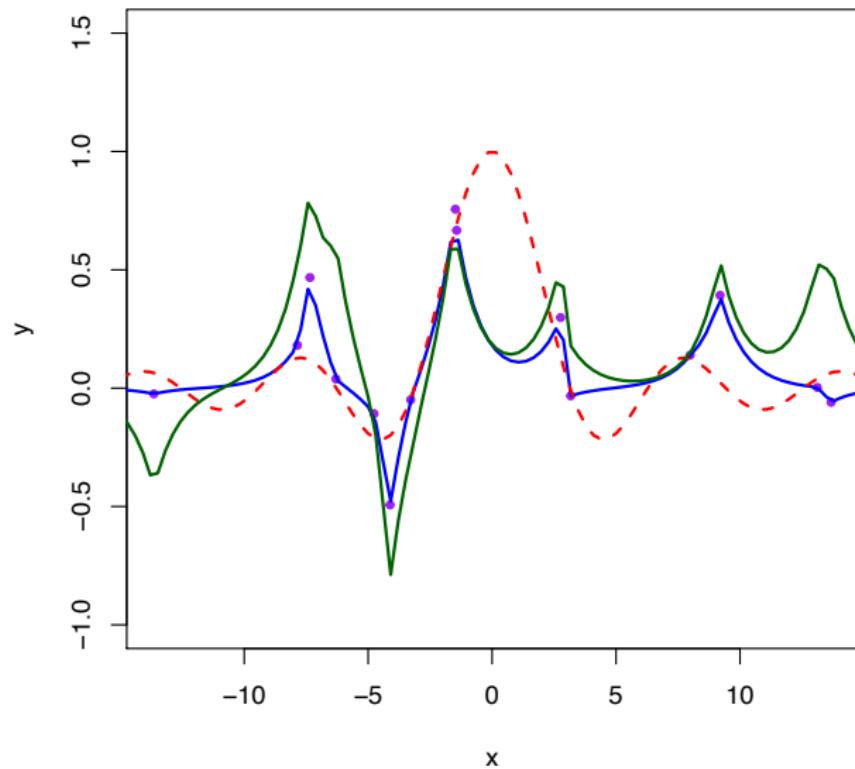


cmi

Bayesian Optimization: First Iteration



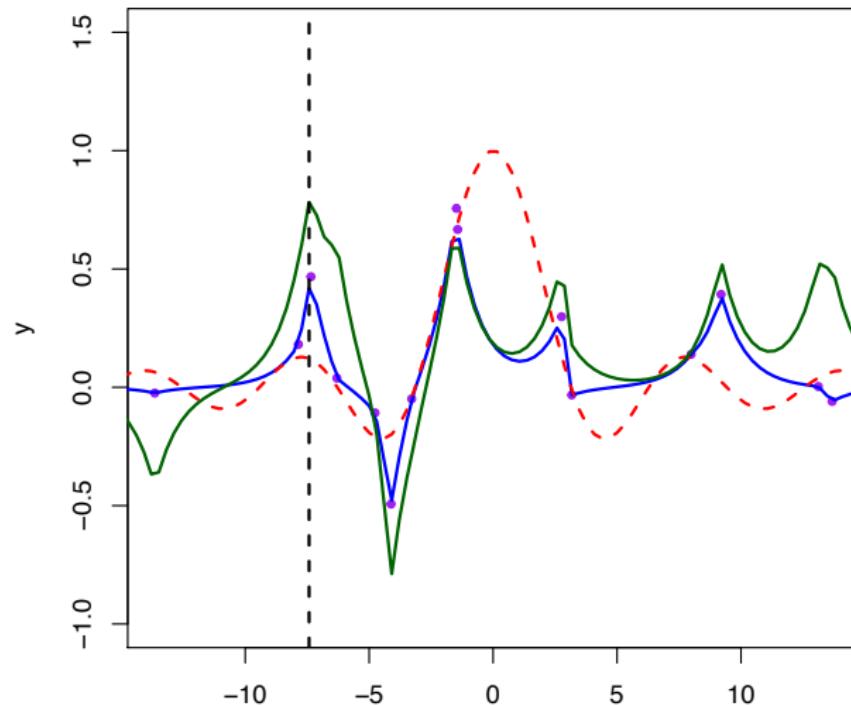
Bayesian Optimization: First Iteration



cmi

Bayesian Optimization: First Iteration

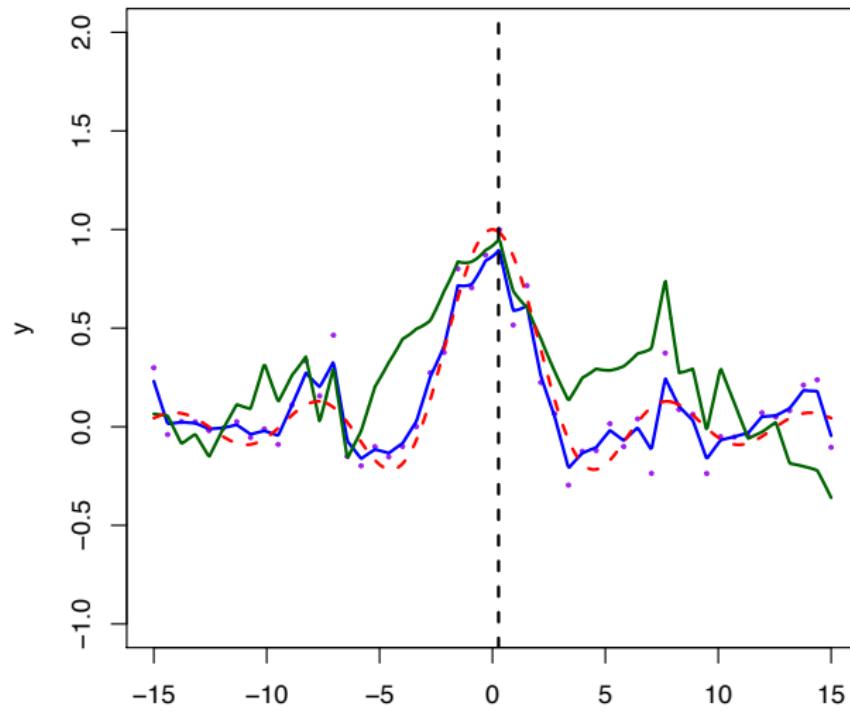
[1] -7.424242



cmi

Bayesian Optimization: Iteration = 50

[1] 0.2705411



cmi