Multi-Agent Reinforcement Learning

Deep Learning

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer
Slides by: Leonard Hinckeldey

The MARL Book

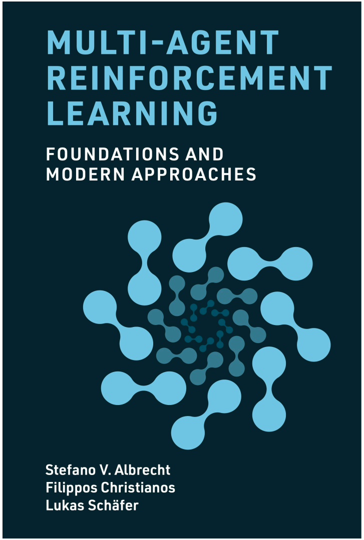
This lecture is based on

Multi-Agent Reinforcement Learning: Foundations and Modern Approaches

by Stefano V. Albrecht, Filippos Christianos and Lukas Schäfer

MIT Press, 2024

Download book, slides, and code at:
www.marl-book.com



MULTI-AGENT REINFORCEMENT LEARNING

FOUNDATIONS AND MODERN APPROACHES

Stefano V. Albrecht
Filippos Christianos
Lukas Schäfer

Lecture Outline

- Function approximation
- Feed forward neural networks
- Training neural networks
- Additional neural network architectures

Function Approximation

Motivation for Function Approximation in RL

Problem

Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair.

Motivation for Function Approximation in RL

Problem

Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair. This has 2 major limitations:

1. The table grows linearly with the number of possible inputs

Motivation for Function Approximation in RL

Problem

Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair. This has 2 major limitations:

1. The table grows linearly with the number of possible inputs → difficult in complex environments (e.g., Go has approximately 10^{170} possible states)

Motivation for Function Approximation in RL

Problem

Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair. This has 2 major limitations:

1. The table grows linearly with the number of possible inputs → difficult in complex environments (e.g., Go has approximately 10^{170} possible states)
2. Each value is updated in isolation

Motivation for Function Approximation in RL

Problem

Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair. This has 2 major limitations:

1. The table grows linearly with the number of possible inputs → difficult in complex environments (e.g., Go has approximately 10^{170} possible states)
2. Each value is updated in isolation → agents need to visit each state-action pair to obtain accurate values

Motivation for Function Approximation in RL

Problem

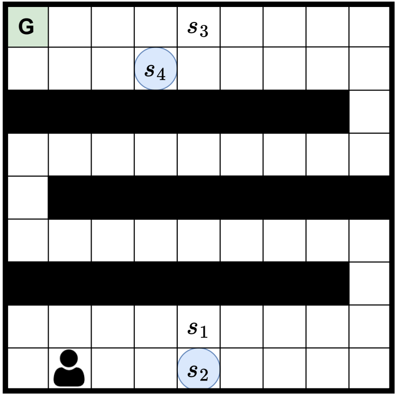
Previously introduced RL and MARL algorithms maintain **tabular** value functions with a value for each state-action pair. This has 2 major limitations:

1. The table grows linearly with the number of possible inputs → difficult in complex environments (e.g., Go has approximately 10^{170} possible states)
2. Each value is updated in isolation → agents need to visit each state-action pair to obtain accurate values

Solution

We need value functions that **generalise** across inputs.

Generalization



- The agent must reach the goal location (G)
- The agent has already seen states s_1 and s_3 during past trajectories, but not s_2 and s_4
- States s_2 and s_4 are similar to s_1 and s_3 respectively
- It would be beneficial if the agent could generalize from s_1 and s_3 to estimate the values of s_2 and s_4

Function Approximation

Function approximation is one way to find generalizable representations for value functions.

- Function approximations learns a **parameterized** function $f(x;\theta)$ for some input $x \in \mathbb{R}^{d_x}$ such that:

$$\forall x : f(x;\theta) \approx f^*(x)$$

- Training a function approximator involves optimizing the parameters θ such that f accurately **approximates** f^*
- In RL, f^* might be the true value function of the environment representing the expected return given a state s as input

Linear Function Approximation

One approach for function approximation is to represent $f(x;\theta)$ as a linear function. For instance, a linear state-value function:

$$\hat{V}(s;\theta) = \theta^\top x(s) = \sum_{k=1}^{d} \theta_k x_k(s)$$

- $x(s)$ denotes the state feature vector
- We can then optimize the parameters using a **loss function** like mean square error

Linear Function Approximation

One approach for function approximation is to represent $f(x;\theta)$ as a linear function. For instance, a linear state-value function:

$$\hat{V}(s;\theta) = \theta^\top x(s) = \sum_{k=1}^{d} \theta_k x_k(s)$$

- $x(s)$ denotes the state feature vector
- We can then optimize the parameters using a **loss function** like mean square error

Note

The function is **linear** in **parameters** but we can transform the input features $x(s)$ to approximate non-linear value functions e.g. $x(s^2)$ for quadratic functions

Linear Function Approximation Optimization

We might optimize $f(x;\theta)$ using a **mean squared error** loss.

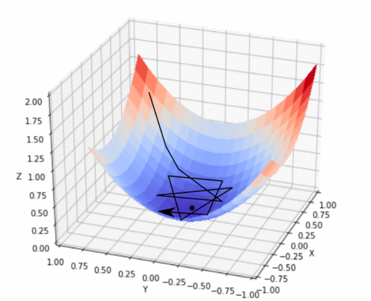
$$\theta^* = \arg\min_{\theta} \mathbb{E}_{s \in S}\left[\left(V^{\pi}(s) - \hat{V}(s;\theta)\right)^2\right]$$

Linear Function Approximation Optimization

We might optimize $f(x;\theta)$ using a **mean squared error** loss.

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{s \in S}\left[\left(V^{\pi}(s) - \hat{V}(s;\theta)\right)^2\right]$$

- We can find optimal θ values by computing gradients of all $\theta_i \in \theta$ with respect to the loss and descend along the gradient



Linear Function Approximation Optimization

We might optimize $f(x;\theta)$ using a **mean squared error** loss.

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{s \in S}\left[\left(V^{\pi}(s) - \hat{V}(s;\theta)\right)^2\right]$$

- We can find optimal θ values by computing gradients of all $\theta_i \in \theta$ with respect to the loss and descend along the gradient
- With optimal parameters $\hat{V}(s;\theta^*)$ might be close to the true value function, allowing us to estimate values by 'plugging in' new potentially unseen states

Linear Function Approximation Optimization

We might optimize $f(x;\theta)$ using a **mean squared error** loss.

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{s \in S}\left[(V^\pi(s) - \hat{V}(s;\theta))^2\right]$$

- We can find optimal θ values by computing gradients of all $\theta_i \in \theta$ with respect to the loss and descend along the gradient
- With optimal parameters $\hat{V}(s;\theta^*)$ might be close to the true value function, allowing us to estimate values by 'plugging in' new potentially unseen states
- Linear function approximation is a simple and powerful way to find generalizable functions

Deep Learning for Function Approximation

The main benefit of linear value functions is their simplicity and generalization ability. However, linear function approximation heavily relies on the selection of state features.

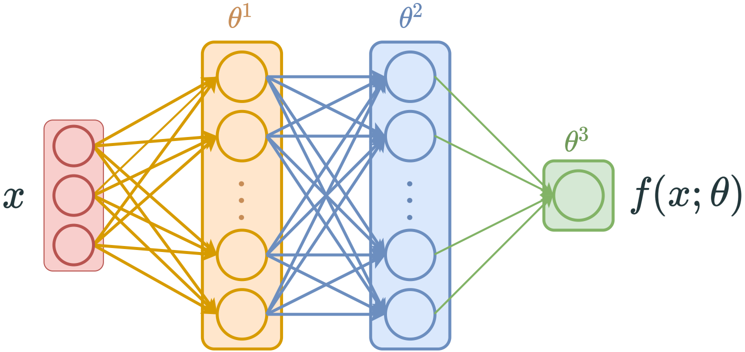
Deep Learning Advantages:

- Universal method, leveraging **neural networks** for function approximation
- Automatically learns feature representations of states
- Able to represent non-linear, complex functions
- Is effective with high-dimensional data such as images and text

Neural Networks

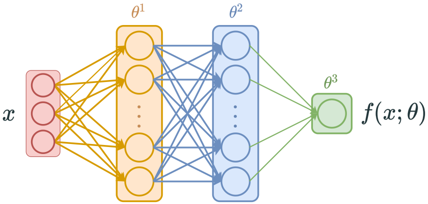
Feedforward Neural Networks

Feedforward neural networks, fully-connected neural networks, or multi-layer perceptrons can be considered as the building block of most neural networks.



Feedforward Neural Networks

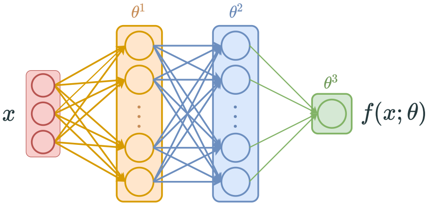
Feedforward neural networks, fully-connected neural networks, or multi-layer perceptrons can be considered as the building block of most neural networks.



- A feedforward NN consists of multiple sequential layers
- The first layer processes input *x*, subsequent layers process the outputs of previous layers
- Each layer is defined as a parameterized function and is composed of many neural units
- We typically refer to the first layer as the **input layer** and the last layer as the **output layer** and all layers in between as **hidden layers**

Feedforward Neural Networks

Feedforward neural networks, fully-connected neural networks, or multi-layer perceptrons can be considered as the building block of most neural networks.



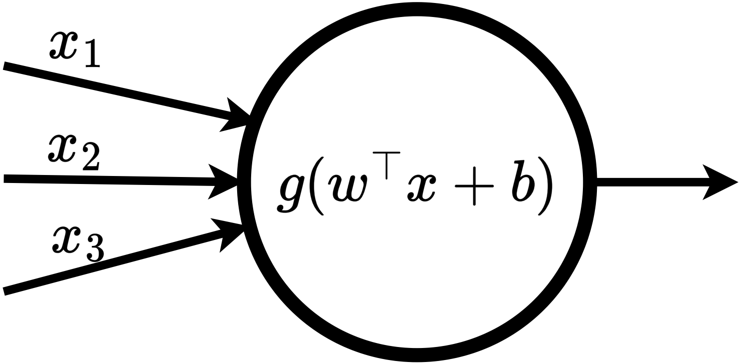
Mathematically, this can be represented as a compositional function:

$$f(x;\theta) = (f_2(f_1(x;\theta^1);\theta^2))$$

where f_k corresponds to the function of layer k with parameters θ^k

Neural Unit

An individual unit of a neural network in layer k represents a parameterized function $f_{k,u} : \mathbb{R}^{d_{k-1}} \to \mathbb{R}$ which processes the output of the previous layer.



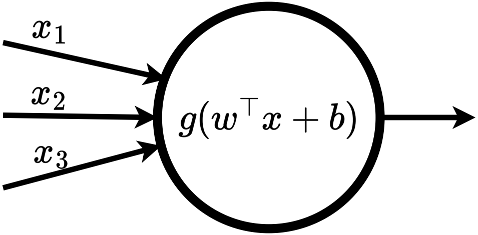
Neural Unit

An individual unit of a neural network in layer k represents a parameterized function $f_{k,u} : \mathbb{R}^{d_{k-1}} \to \mathbb{R}$ which processes the output of the previous layer.

- The previous layer's output is the concatenation of all its units' outputs
- Formally this is:

$$f_{k,u}(x; \theta^k_u) = g_k(w^\top x + b)$$

- θ^k_u is the weight vector w as well as the scalar bias b
- g_k is a **non-linear** activation function



Neural Unit

An individual unit of a neural network in layer k represents a parameterized function $f_{k,u} : \mathbb{R}^{d_{k-1}} \to \mathbb{R}$ which processes the output of the previous layer.

- The previous layer's output is the concatenation of all its units' outputs
- Formally this is:

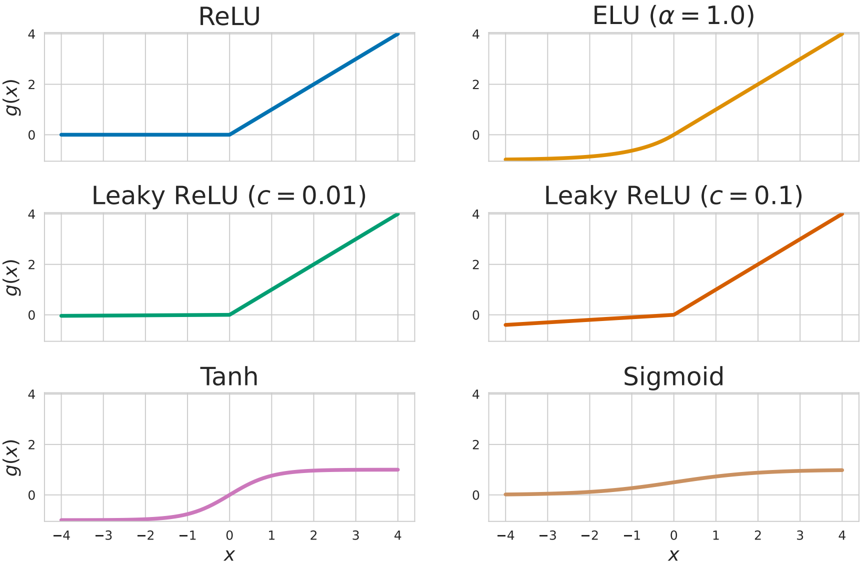
$$f_{k,u}(x; \theta_u^k) = g_k(w^\top x + b)$$

- θ_u^k is the weight vector w as well as the scalar bias b
- g_k is a **non-linear** activation function

Note

Applying a **non-linear** activation function is essential, because the composition of two linear function f and g can only represent a linear function itself.

Activation Functions



Computation of Neural Network Layer

We can use vector notation to aggregate the computation of all neural units of layer k as:

$$f_k(x_{k-1}; \theta^k) = g_k(W_k^\top x_{k-1} + b_k)$$

- $W_k \in \mathbb{R}^{d_{k-1} \times d_k}$ is a weight matrix
- $b_k \in \mathbb{R}^{d_k}$ is the bias vector

The high-dimensional linear transformation of a single layer k can now be described as a single matrix multiplication with W_k, followed by vector addition with b_k and applying an activation function g_k element-wise, resulting in an output vector.

Universal Function Approximation Theory

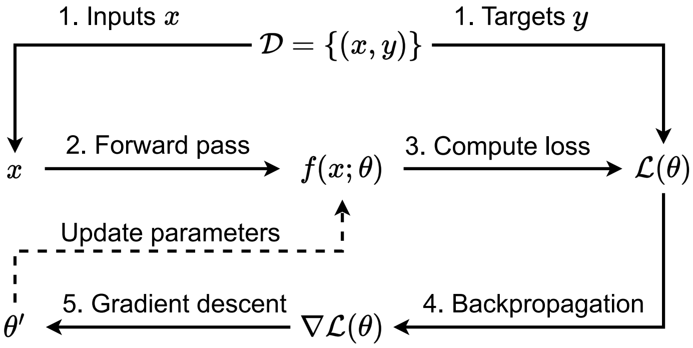
The **universal function approximation theory** states that with as little as one hidden layer, a feedforward neural network can approximate **any** function.

- This theory assumes arbitrary **width** of the hidden layer
- We refer to the number of neural units within hidden layers as the **width** of the network
- The number of layers is referred to as the **depth**
- In practice, **deep** neural networks can outperform shallow ones and in some cases, require fewer parameters to do so

Training Neural Networks

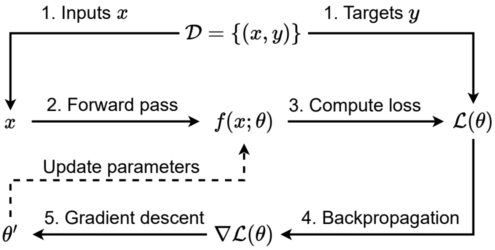
Gradient Based Optimization

To train a neural network, we repeatedly update the parameters θ such that $f(x;\theta)$ becomes better at approximating a target function.



Gradient Based Optimization

To train a neural network, we repeatedly update the parameters θ such that $f(x;\theta)$ becomes better at approximating a target function.



Steps of the optimization process

1. Gather batch of input-target (x, y) pairs
2. Compute network output for given inputs $f(x;\theta)$
3. Compute the loss over network outputs and targets y
4. Backpropagation (compute gradients)
5. Gradient descent (update θ)

Loss Function

The loss function indicates how ’wrong’ the neural network estimates are. In optimization the objective is to find θ such that the loss is minimized:

$$\theta^* \in \arg\min_\theta \mathcal{L}(\theta)$$

Loss Function

The loss function indicates how ’wrong’ the neural network estimates are. In optimization the objective is to find θ such that the loss is minimized:

$$\theta^* \in \arg\min_\theta \mathcal{L}(\theta)$$

- Knowing the true values (supervised learning) allows us to construct a loss by comparing the neural network estimated value with the true value, e.g., MSE:

$$\mathcal{L}(\theta) = \frac{1}{B}\sum_{k=1}^{B}(V^\pi(s_k) - \widehat{V}(s_k;\theta))^2$$

- Where $\mathcal{B}$ is a **batch** (B is batch size) of states and true value $V^\pi(s_k)$ pairs
- Minimizing the loss leads to parameters θ such that $\widehat{V}(s_k;\theta)$ more accurately approximates the true value function

Loss Function

The loss function indicates how ’wrong’ the neural network estimates are. In optimization the objective is to find θ such that the loss is minimized:

$$\theta^* \in \arg\min_\theta \mathcal{L}(\theta)$$

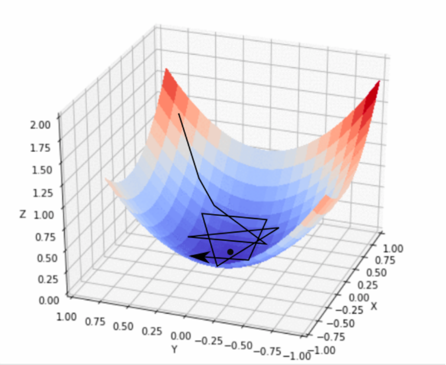
- Knowing the true values (supervised learning) allows us to construct a loss by comparing the neural network estimated value with the true value, e.g., MSE:

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{k=1}^{B} (V^\pi(s_k) - \widehat{V}(s_k; \theta))^2$$

Minimizing the loss will optimize θ such that $\widehat{V}$ will gradually provide more accurate state-value estimates.

Gradient Descent

Gradient descent sequentially updates the parameters θ by following the **negative** (hence ”descent”) **gradients** of the loss function with respect to the parameters for given data.



Gradient Descent

Gradient descent sequentially updates the parameters θ by following the **negative** (hence "descent") **gradients** of the loss function with respect to the parameters for given data.

- The gradient is defined as a **vector** of partial derivatives one for each $\theta_i \in \theta$

$$\nabla \mathcal{L}(\theta) = \left(\frac{\partial \mathcal{L}(\theta)}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{L}(\theta)}{\partial \theta_d} \right)$$

- In the simplest case, gradient descent updates the parameters as follows:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta \mid \mathcal{D})$$

- α is the learning rate and $\mathcal{D}$ the training data

Stochastic Gradient Descent

Problem

The application of (vanilla) gradient descent has some downsides:

- Computing gradients across the entire dataset $\mathcal{D}$ is costly (often doesn't fit in memory)
- Convergence is slow due to computational overhead of computing gradients

Solution

Stochastic gradient descent (SGD) instead computes gradients over randomly sampled data pairs $d \sim \mathcal{U}(\mathcal{D})$ (speeds up convergence but increases variance):

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta \mid \mathrm{d})|_{d \sim \mathcal{U}(\mathcal{D})}$$

Mini-batch Gradient Descent

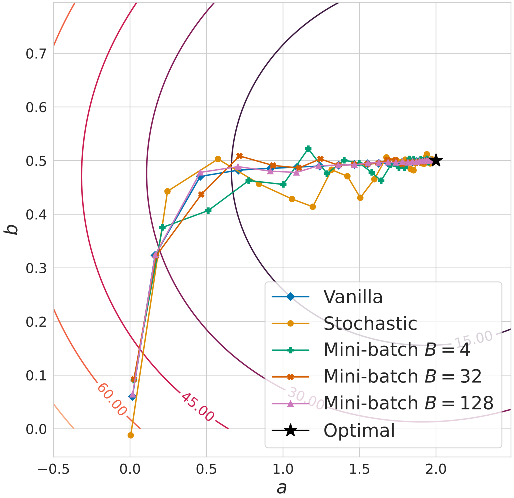
Mini-batch gradient descent is a middle ground between SGD and GD.

- Randomly samples a predefined **batch** of data pairs
- This reduces variance compared to SGD while still converging much faster than GD

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta|\mathcal{B})|_{\mathcal{B}=\{d_i \sim \mathcal{U}(\mathcal{D})\}_{i=1}^{B}}$$

- $\mathcal{B}$ denotes a **batch** and the **batch size** is a hyperparameter determined by B
- The **batch size** can be adjusted depending on the computational power available and the type of data being worked with

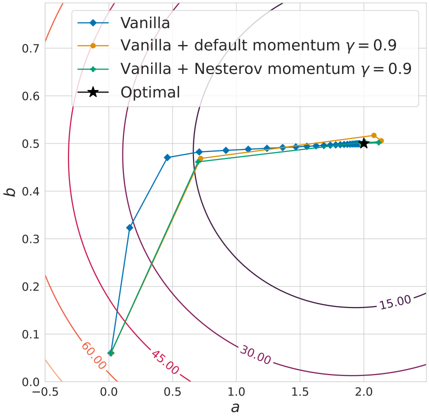
GD vs SGD vs MGD



- **Vanilla gradient descent:** smooth optimization but each step is expensive
- **Stochastic gradient descent:** less stable optimization but is notably faster per step
- **Mini-batch gradient descent:** stable optimization for larger batch sizes and efficient

Momentum

Modern optimization algorithms often also use **momentum**.



- **Momentum** computes moving averages over past gradients to ”accelerate” convergence
- This works particularly well if gradients consistently follow a similar direction
- There are more recent optimisers which also dynamically adjust the learning rate; one popular such optimiser is **Adam** (Kingma and Ba 2015)

Backpropagation

Backpropagation is the algorithm that computes the gradients for **every** parameter in the neural network.

- The algorithm applies the **chain rule** making use of the compositional function of the neural network (i.e. $f(x;\theta) = f_2(f_1(x;\theta^1);\theta^2)$)

Reminder

The chain rule states that for $y = g(x)$ and $z = f(y) = f(g(x))$:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z$$

Here $\frac{\partial y}{\partial x}$ is the Jacobian matrix of function g.

Backpropagation

Backpropagation is the algorithm that computes the gradients for **every** parameter in the neural network.

- The algorithm applies the **chain rule** making use of the compositional function of the neural network (i.e. $f(x;\theta) = f_2(f_1(x;\theta^1);\theta^2)$)

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z$$

- To efficiently find all of the gradients, we start at the output layer and traverse the network backwards layer by layer, applying the chain rule
- This **backward pass** propagates all the gradients through the entire network
- These gradients are then used in **gradient descent**

Other Neural Network Architectures

Convolution and Recurrent Neural Networks

Feedforward neural networks can, in principle, be applied to any data. There are, however, architectures designed to perform well with specific data.

Architectures often used in RL and MARL are:

1. **Convolutional neural networks**
   - Used to process images
   - Aims to make use of spatial correlations
   - Often incorporate inductive biases such as invariance to local image-based translations
2. **Recurrent neural networks**
   - Used to process temporal data
   - Makes use of temporal correlations

Convolutional Neural Networks

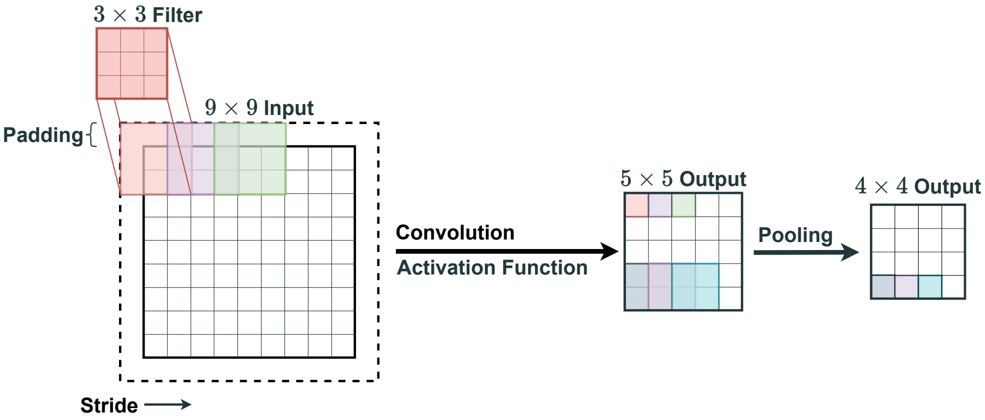
Convolutional neural networks (CNNs) are neural networks for image processing

- Blocks of parameters (called **filters** or **kernels**) are moved and repeatedly applied across an image

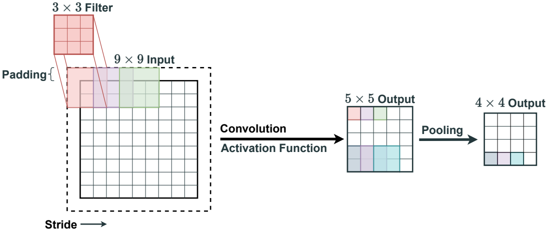
Some advantages to this architecture include:

- **Parameter sharing:** kernel significantly reduces the total number of parameters compared to a feedforward NN
- For example to process a 128×128 RGB image:
  - Feedforward NN: input layer with 128 hidden units has 6,291,584 parameters
  - CNN: with sixteen 5×5 filters has 1,216 parameters
- **Spatial correlations:** nearby pixels are likely related (e.g., edges) $\rightarrow$ same spatial patterns may occur at different locations of an image

Convolutional Neural Networks



Convolutional Neural Networks



- Filter or kernels (a collection of parameters) are moved across an image
- The convolution can be thought of as reducing the dimensional (e.g. taking in 3×3 patch of pixels and returning one single value)

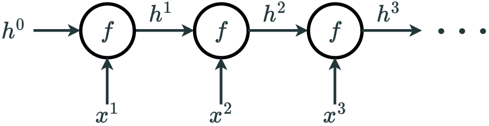
- Convolution of a filter with parameters $W \in \mathbb{R}^{d_w \times d_w}$ over input $x \in \mathbb{R}^{d_x \times d_x}$ is:

$$y_{i,j} = \sum_{a=1}^{d_w} \sum_{b=1}^{d_w} W_{a,b} x_{i+a-1,j+b-1}$$

Recurrent Neural Networks

Recurrent neural networks (RNNs) are designed to process **sequential** data with several advantages:

- Learning sequences may require passing in the entire sequence as an input, which requires lots of parameters
- It is also reasonable to assume a correlation between time points in a sequence
- Sharing parameters might help pick up on recurring patterns in the data that occur at different timesteps

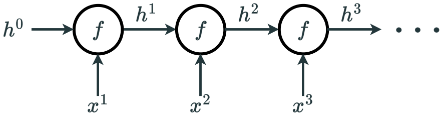


Recurrent Neural Networks

RNNs use a **hidden state**, passed as an additional input.

- RNNs sequentially process inputs (i.e., first x^1 then x^2 ...)
- The RNN also conditions on the **hidden state** h^t
- h^t can be thought of as a compact representation of the entire history of past inputs

Function: $h^t = f(x^t, h^{t-1}; \theta)$



Summary

We covered:

- Function approximation
- Feed forward neural networks
- Loss functions, backpropagation and gradient descent
- CNNs and RNNs

Next we'll cover:

- Deep **reinforcement** learning