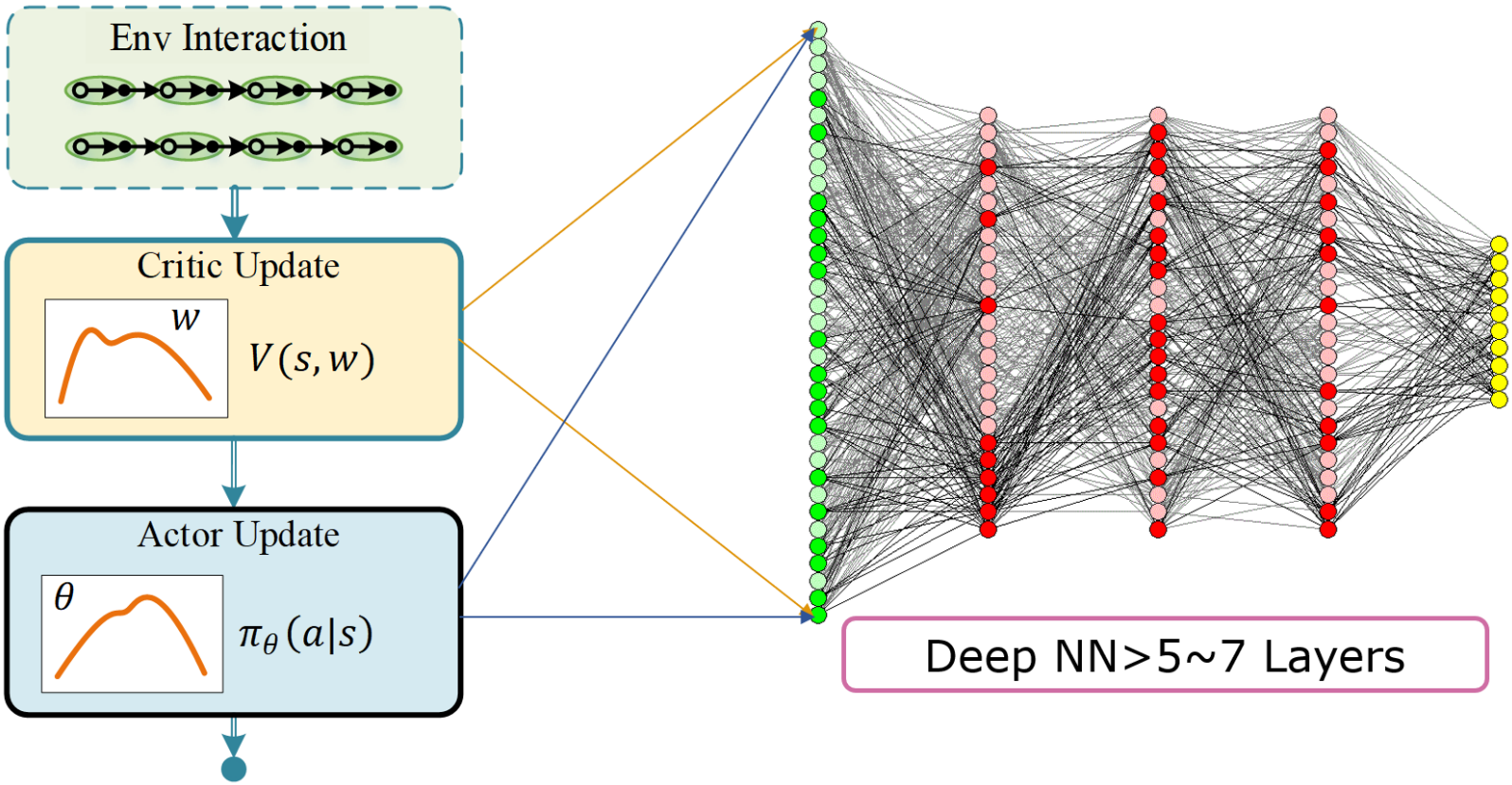
Lecture 11: Deep Reinforcement Learning

Dr. Wen Fuxi

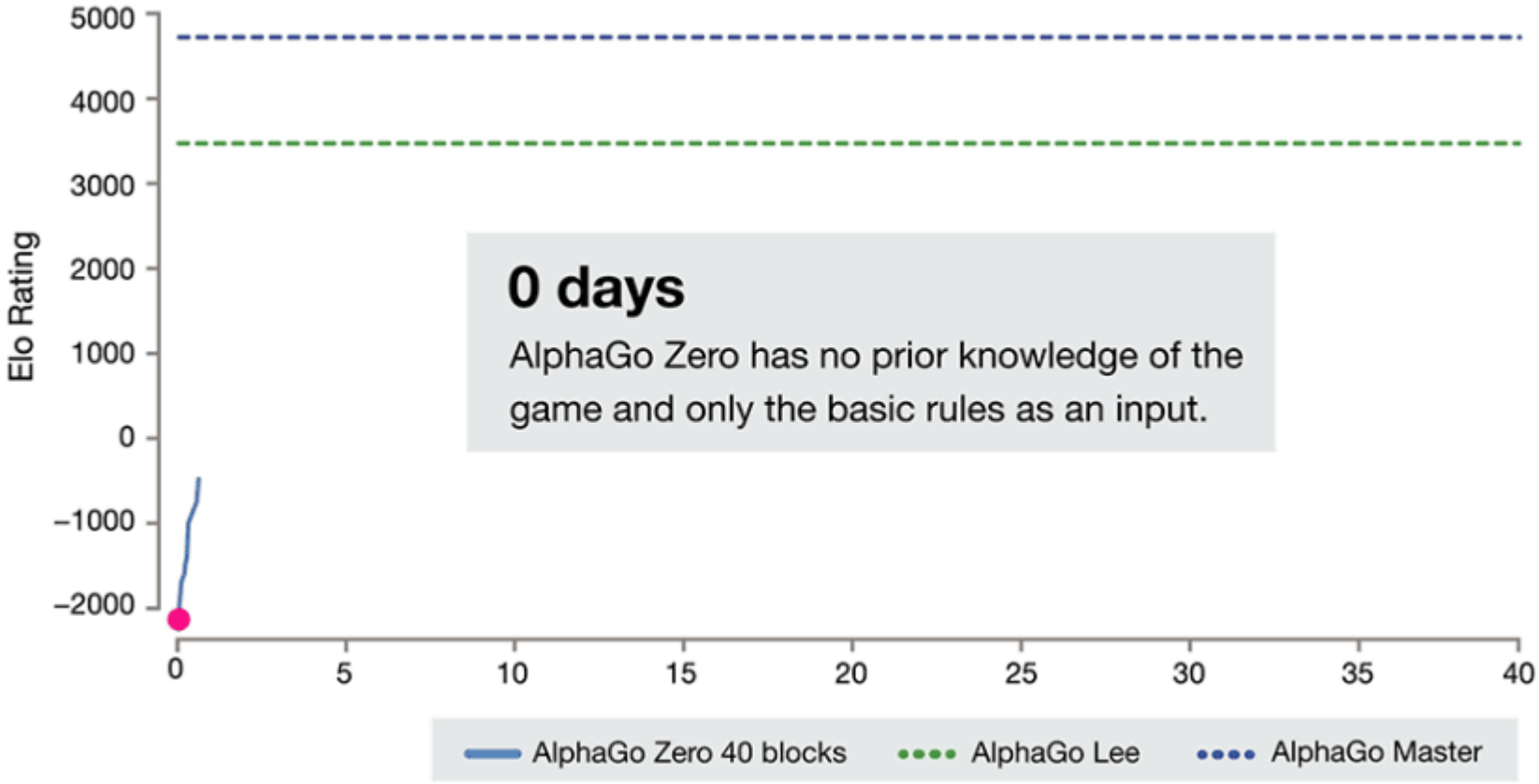
What is Deep RL? Deep RL = RL + Deep NN

$$\max_{\pi} / \min J(\pi) = \mathbb{E}_{s \in d_{\text{init}}(s)} \{v^{\pi}(s)\}$$

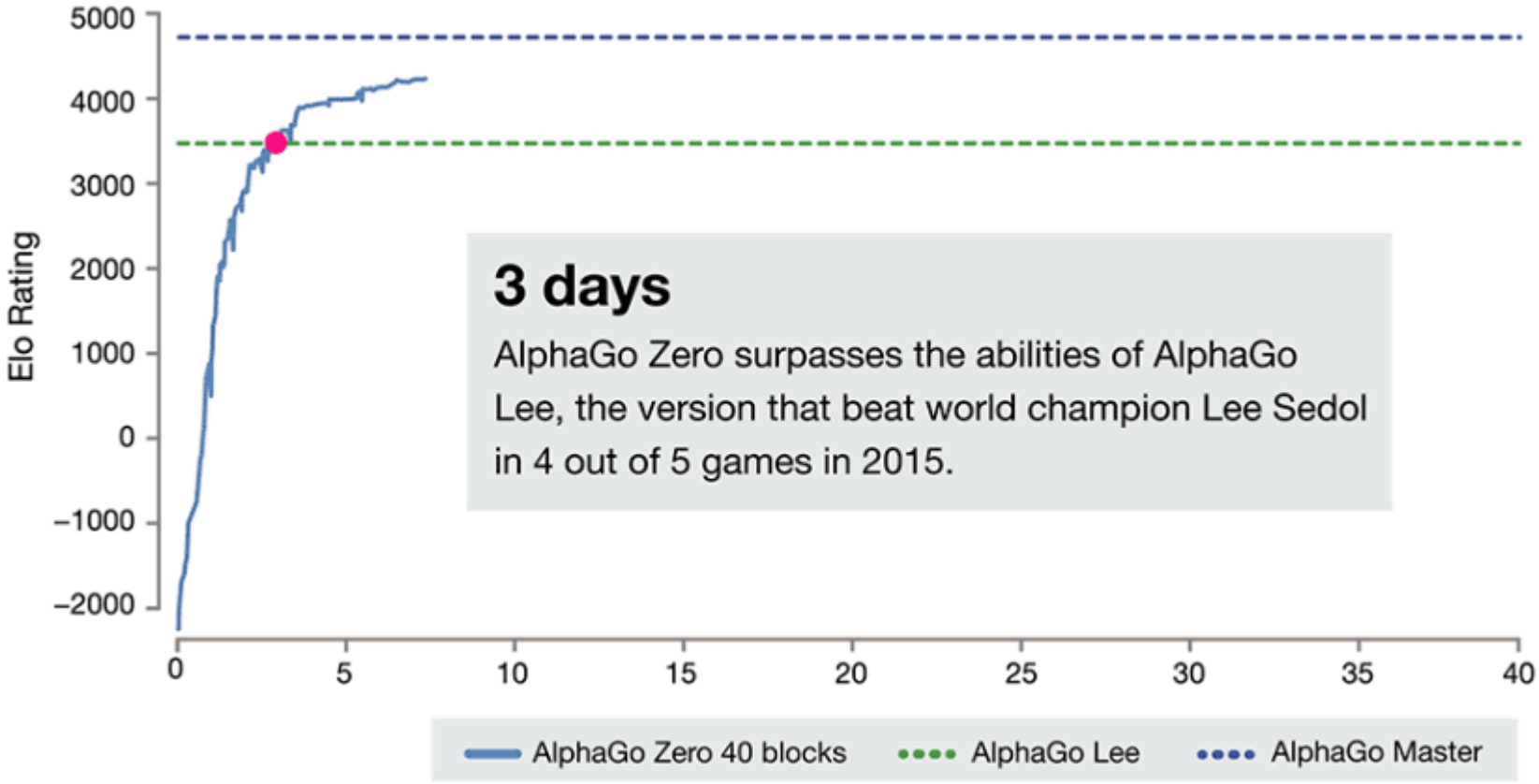
$$\downarrow$$



Deep RL in games

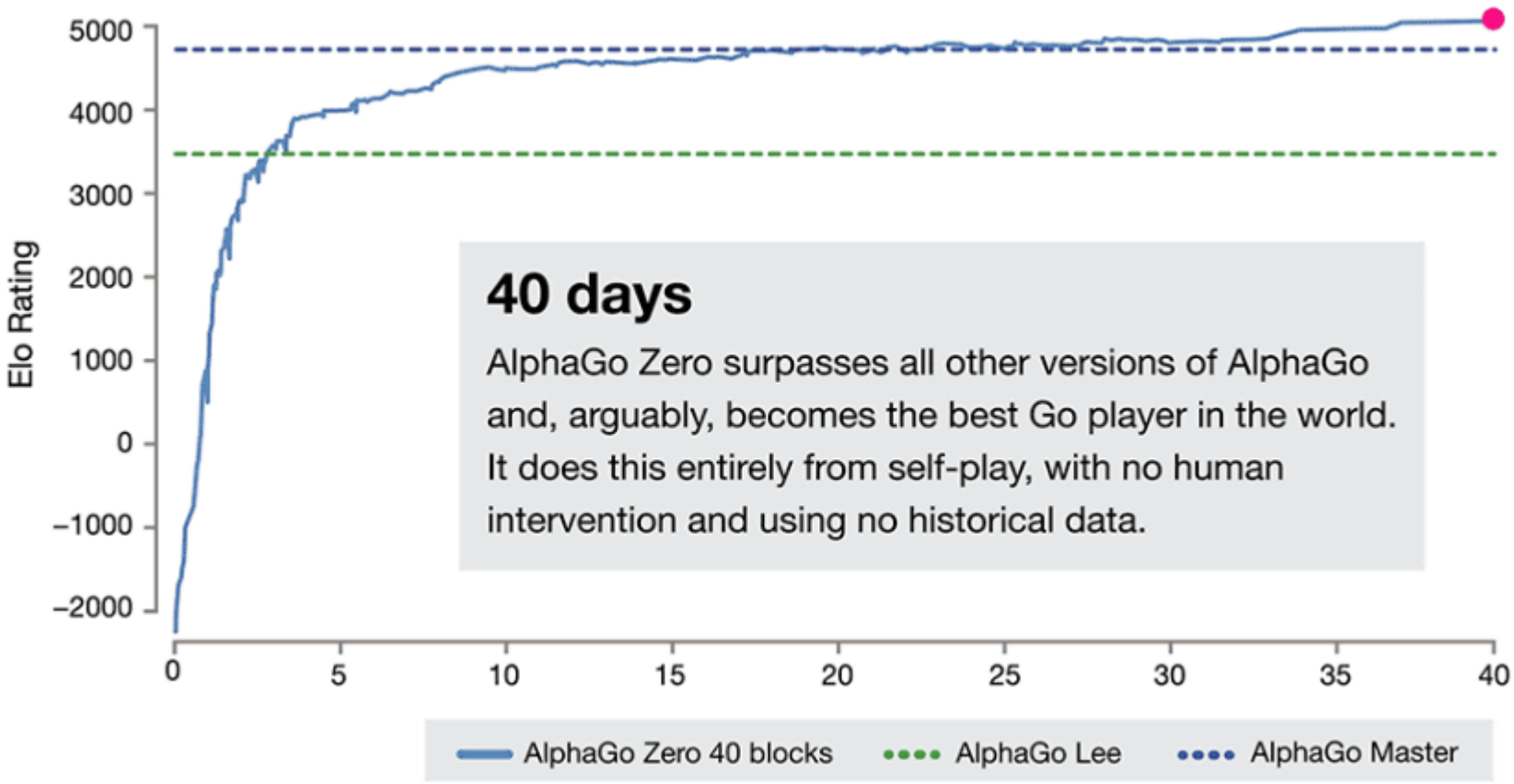


Deep RL in games

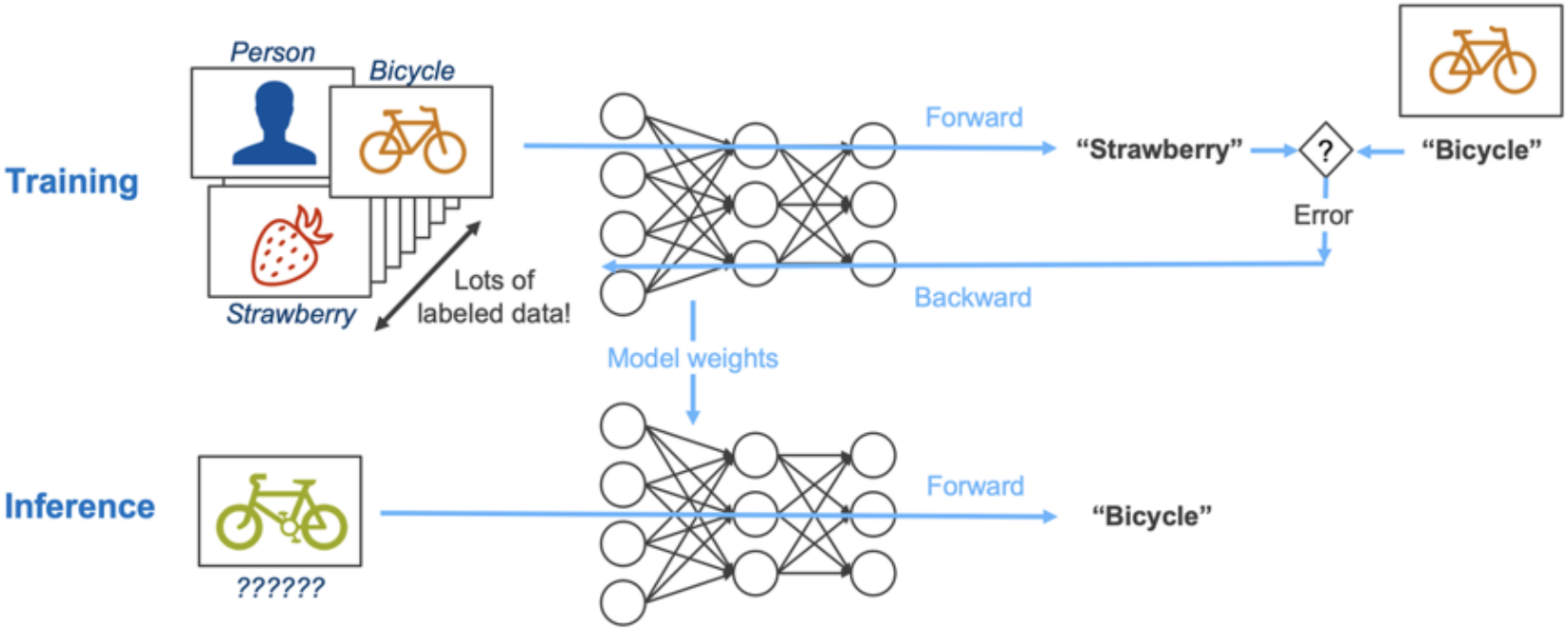


Deep RL in games

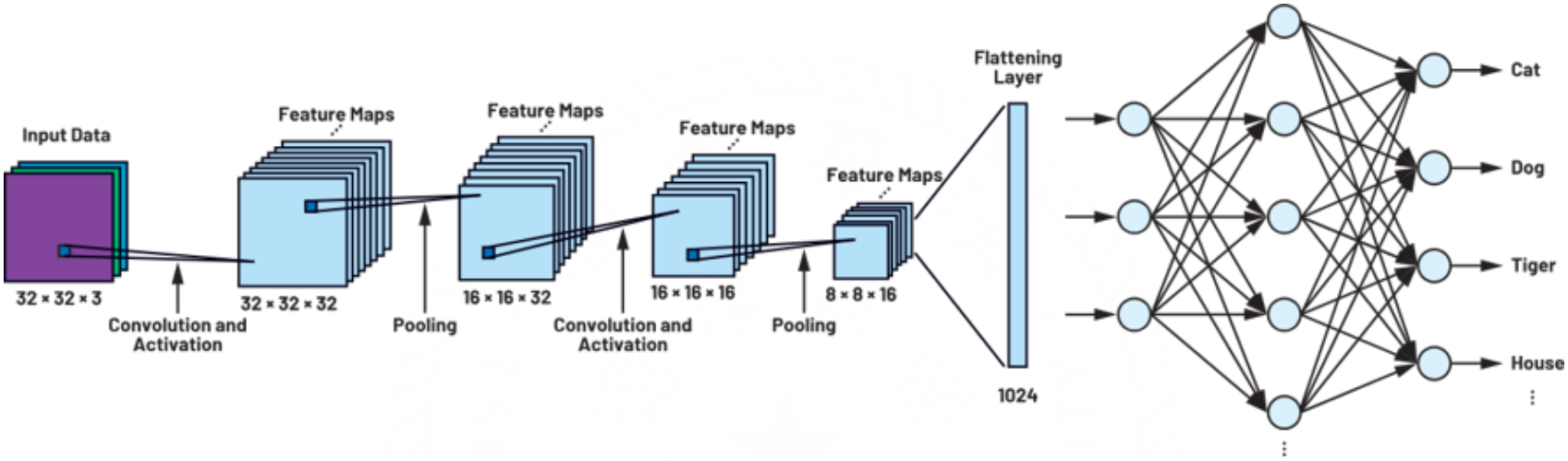
Deep RL in games



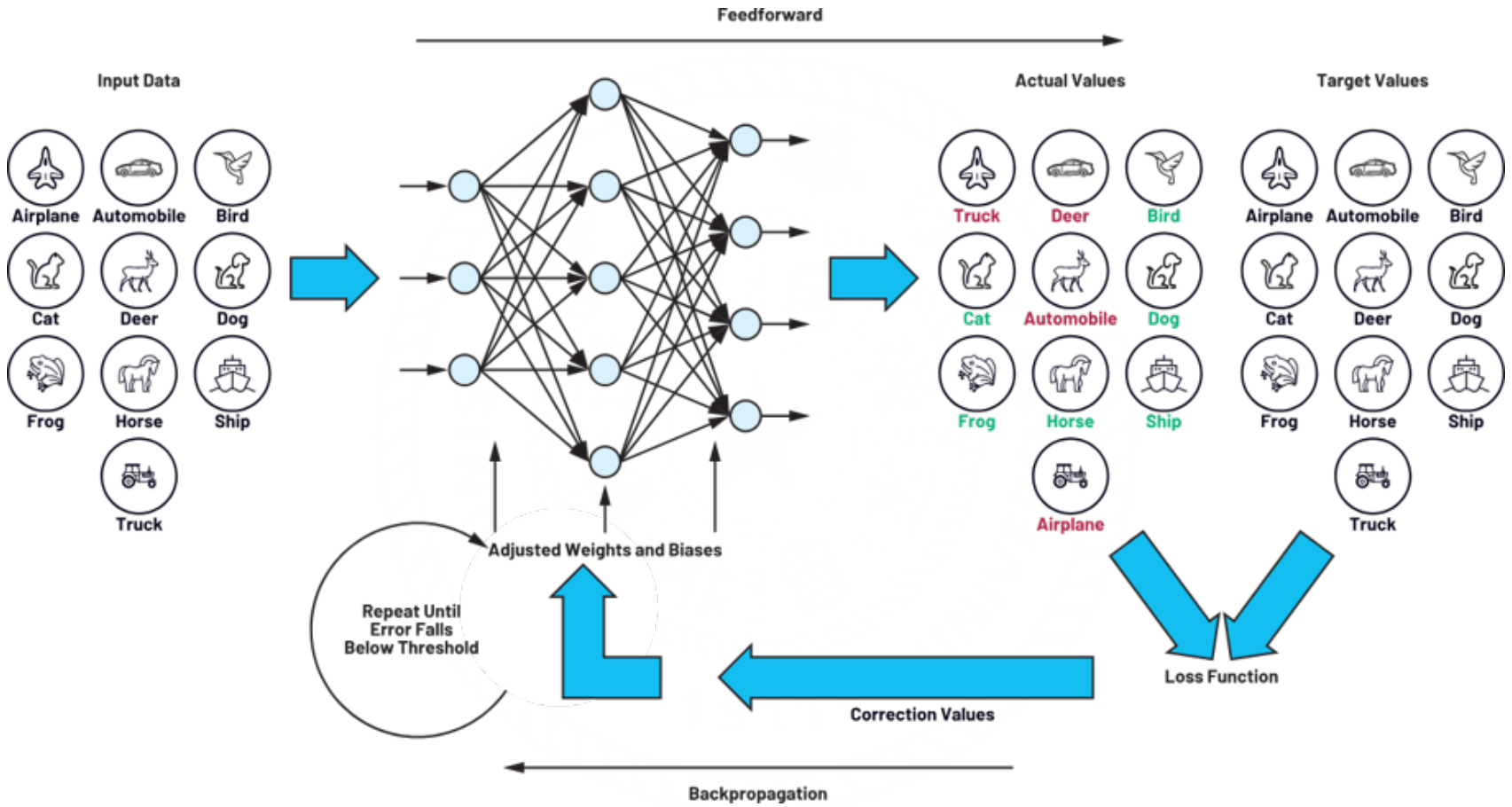
Training and Inference



Convolutional Neural Networks

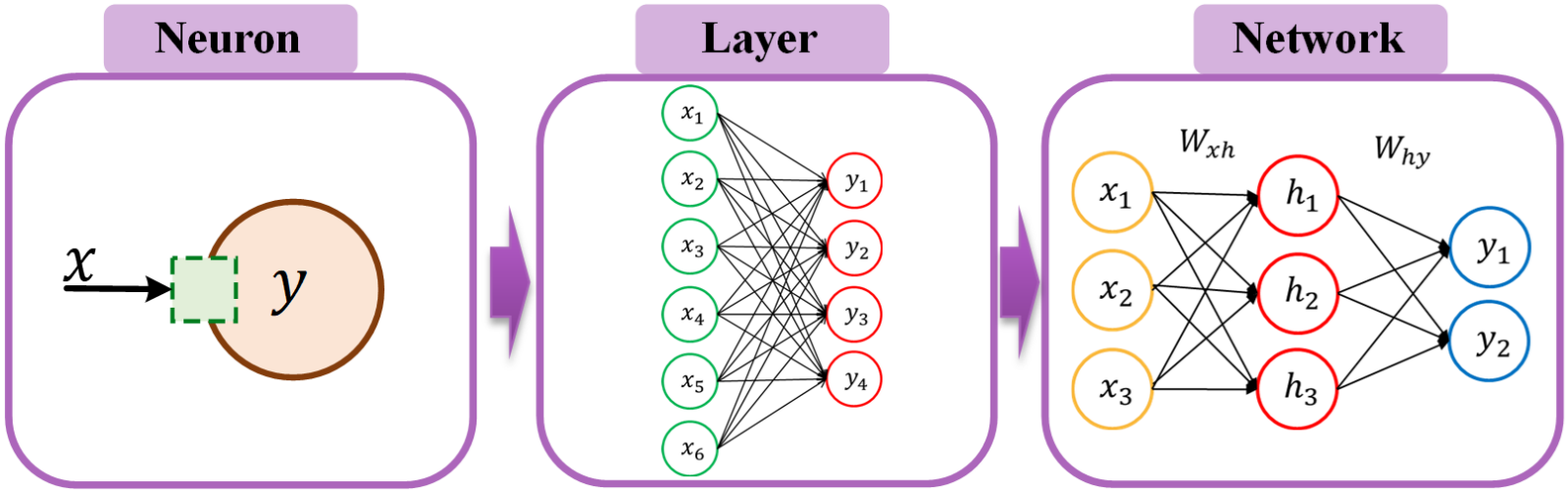


A training loop with feedforward and backpropagation



Artificial Neural Network - Layered Structure

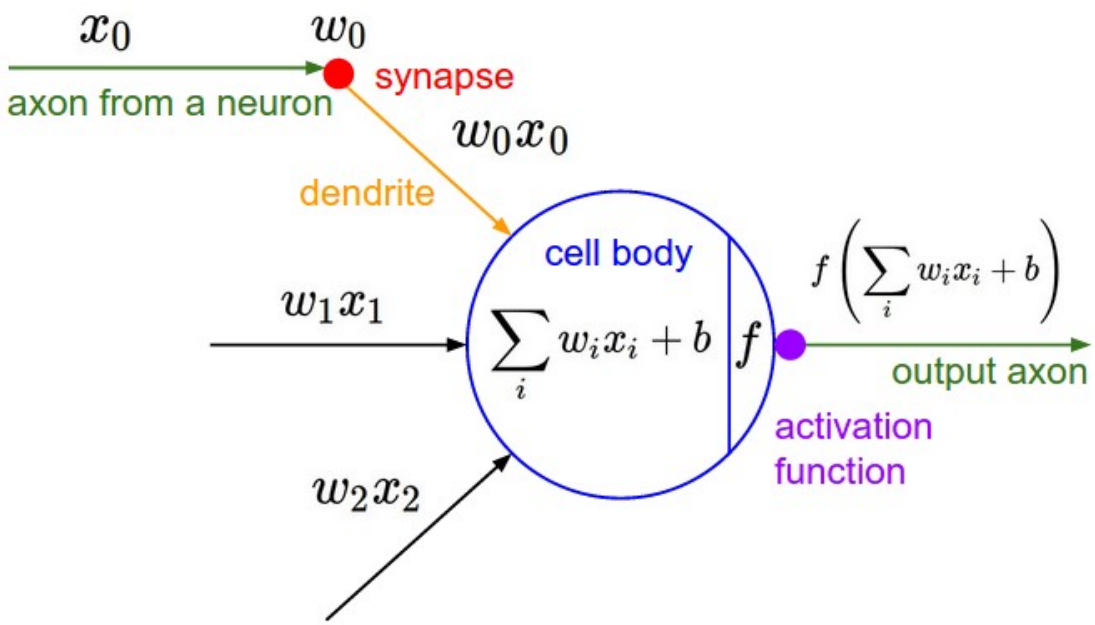
- Neuron: basic computing unit of a neural network
- Layer: combination of neurons in the width direction
- Network: connect layers along the depth direction



Artificial Neural Network - Neuron

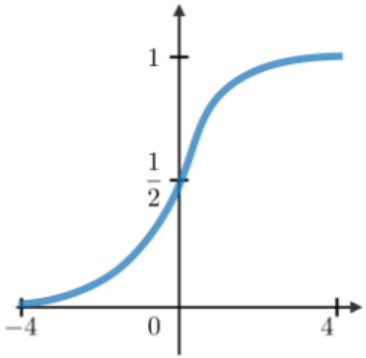
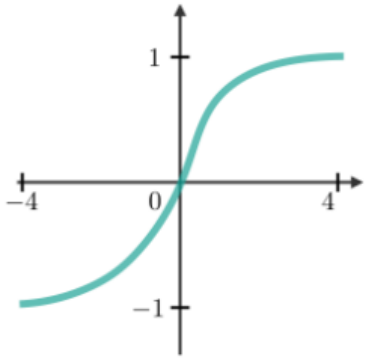
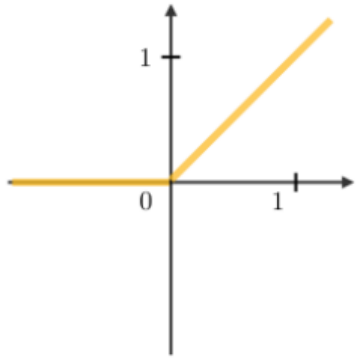
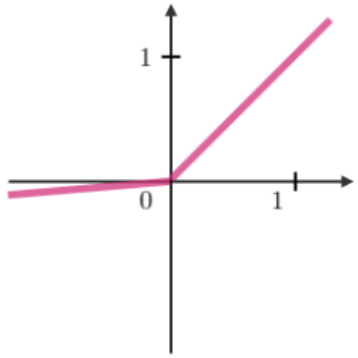
Neuron

- Affine transformation
  - Linear transformation + offset
  - Superposition of linear functions is still linear
- Non-linear activation function



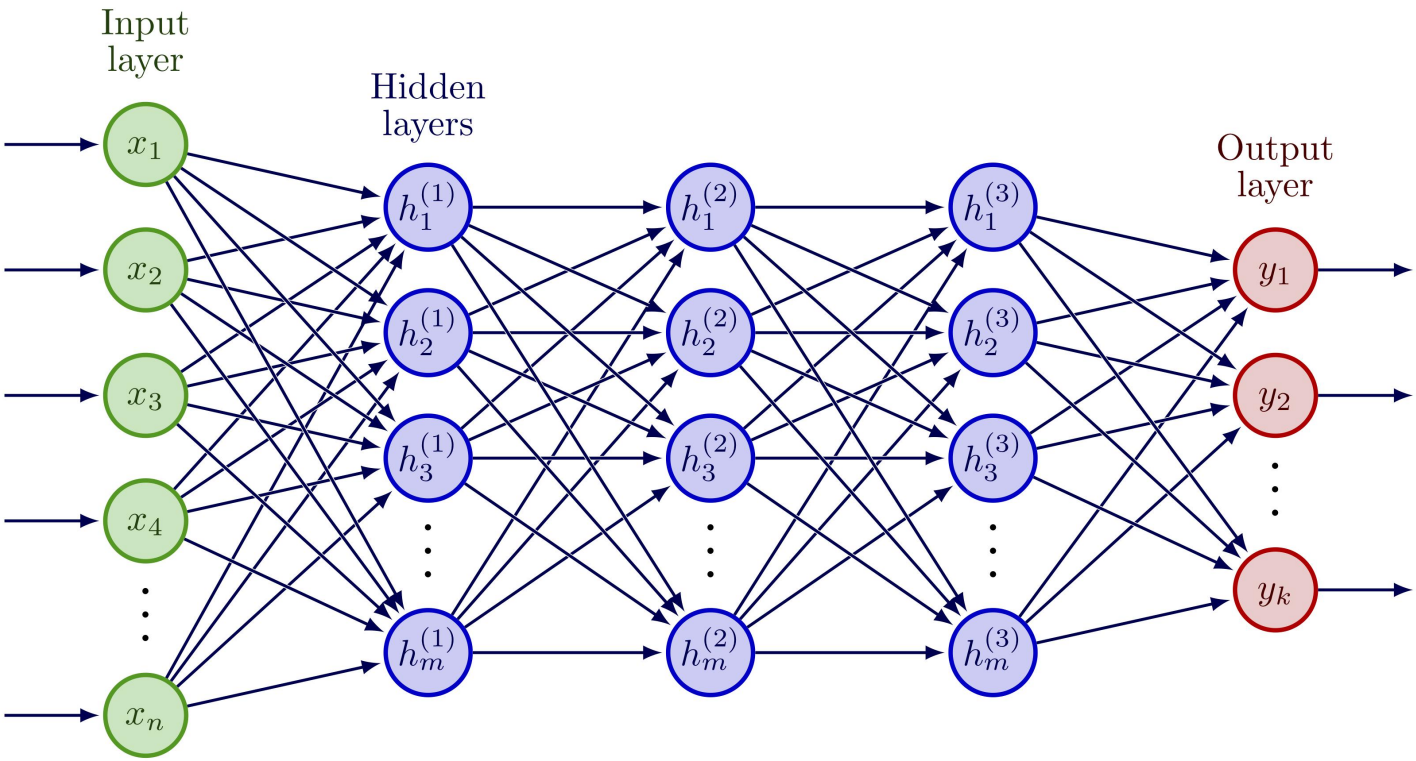
Artificial Neural Network - Activation Function

Non-linear Activation Function

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

Artificial Neural Network - Layers

Fully connected (FC) layer



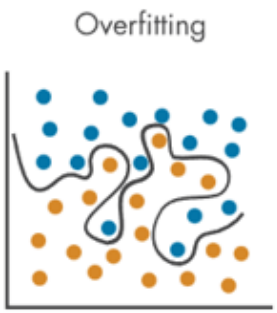
Overfitting vs. Underfitting

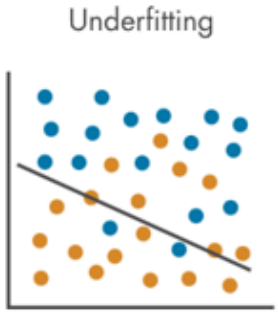
What Is Overfitting?

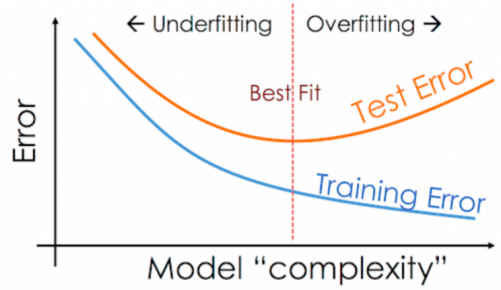
Overfitting occurs when the model is so closely aligned to the training data that it does not know how to respond to new data.

Why?

- The machine learning model is too complex; it memorizes very subtle patterns in the training data that don't generalize well.
- The training data size is too small for the model complexity and/or contains large amounts of irrelevant information.





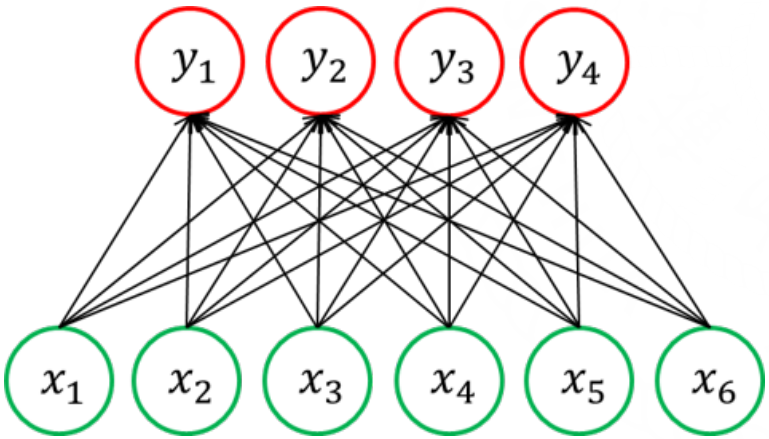


Overfitting - Dropout technique

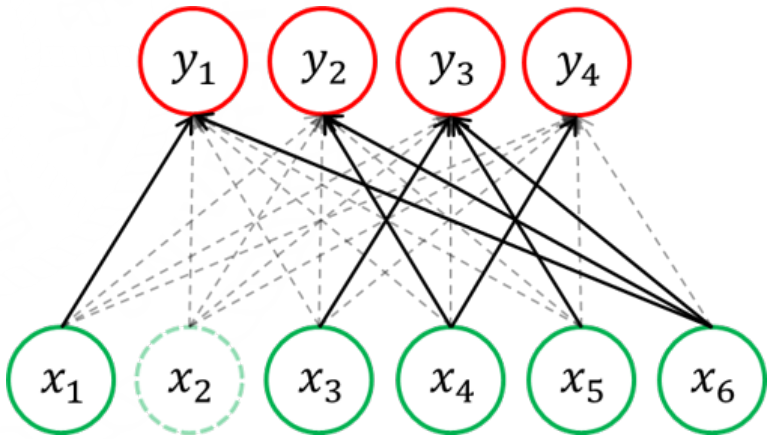
"Dropout technique" to conquer the overfitting issue

- Randomly remove some neurons.
- Randomly remove some connections.

Trainable weights



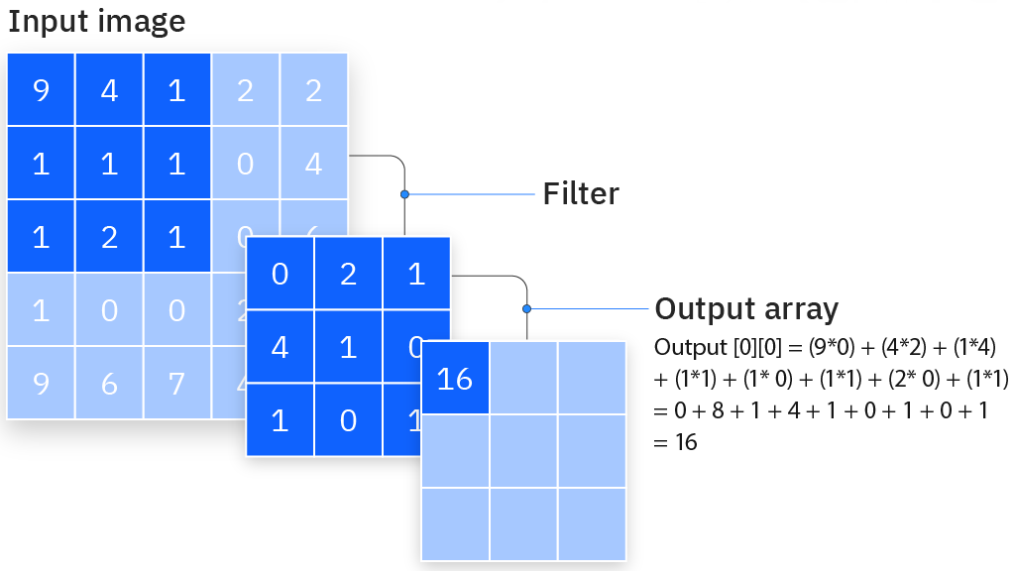
Dropout technique

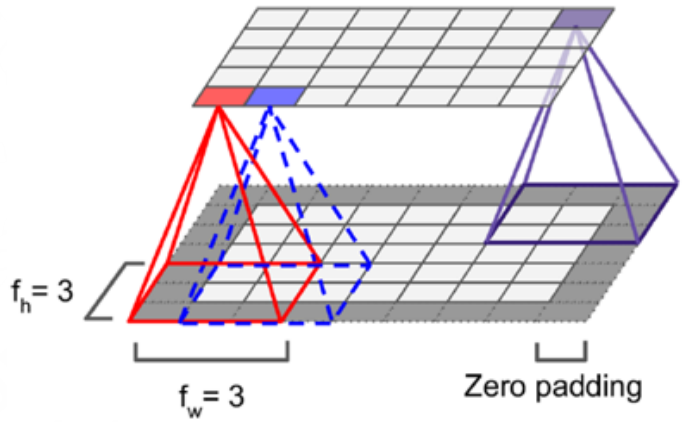


Convolutional Layer

Key parameters of convolutional layer

- kernel size, number of kernels, number of kernel channels
- sliding stride, padding size





- 3X3 kernel
- One sliding stride
- Zero padding

Largely reduce the number of trainable weights

Each receptive field shares the same kernels

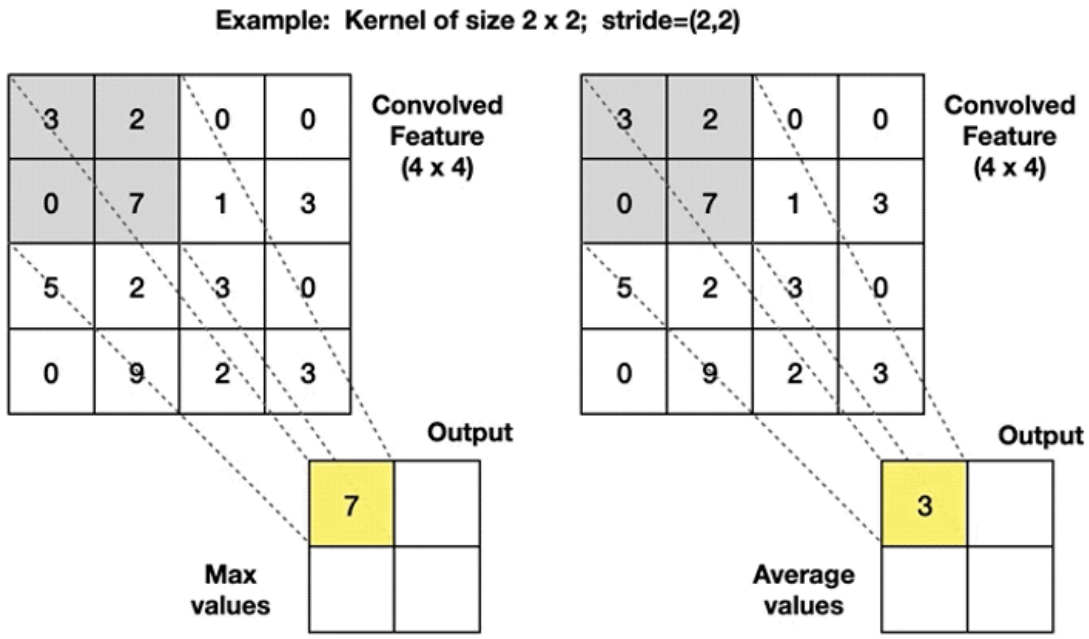
Max and Average Pooling layers

Max Pooling

Take the highest value from the area covered by the kernel.

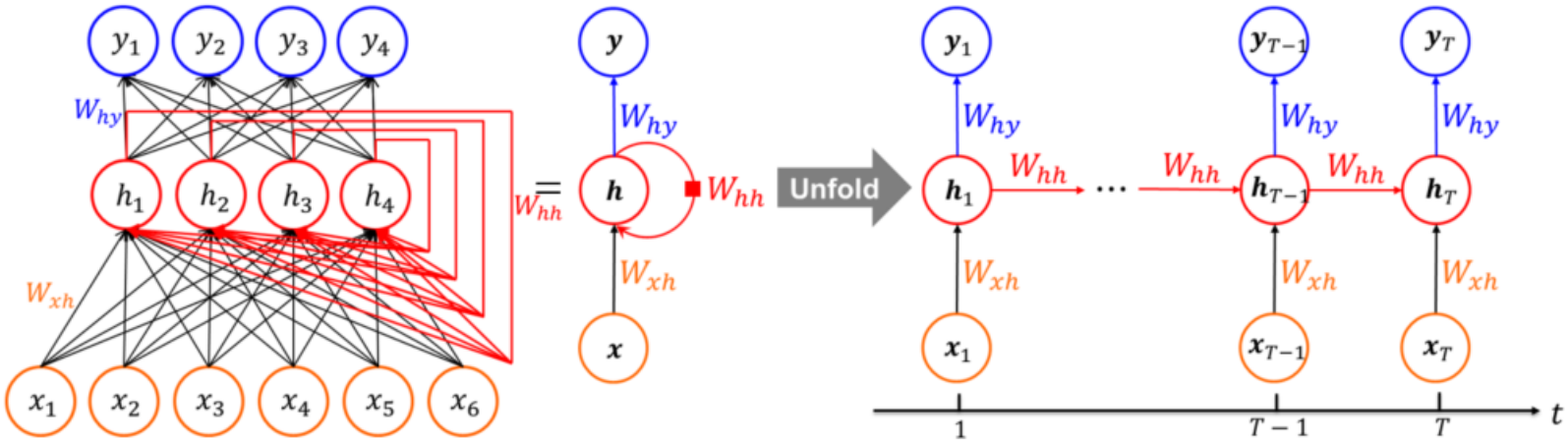
Average Pooling

Calculate the average value from the area covered by the kernel.

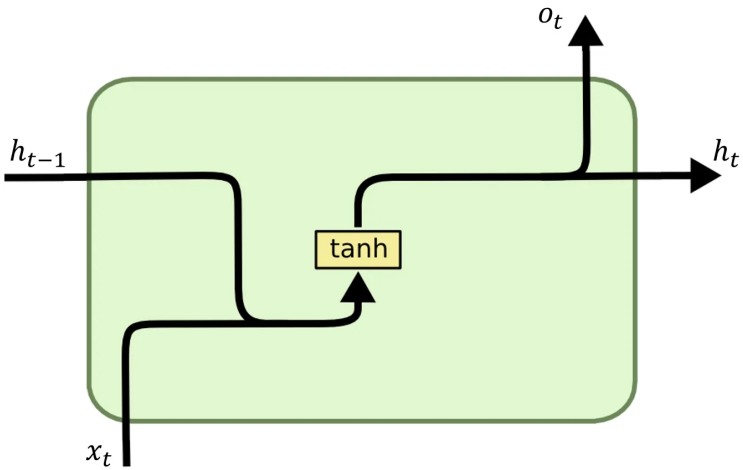


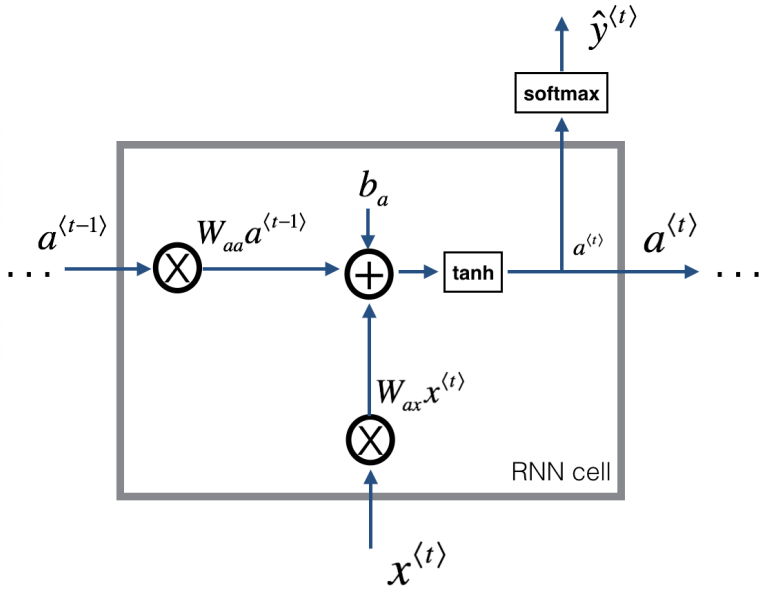
Recurrent layer

- The output of the previous step is fed as the input to the current step.
- The hidden state carries pertinent information of previous inputs.
- Once unfolded in time, feedforward networks have every layer sharing weights.
- When training, gradients easily explode or vanish



Recurrent Neural Network - RNN



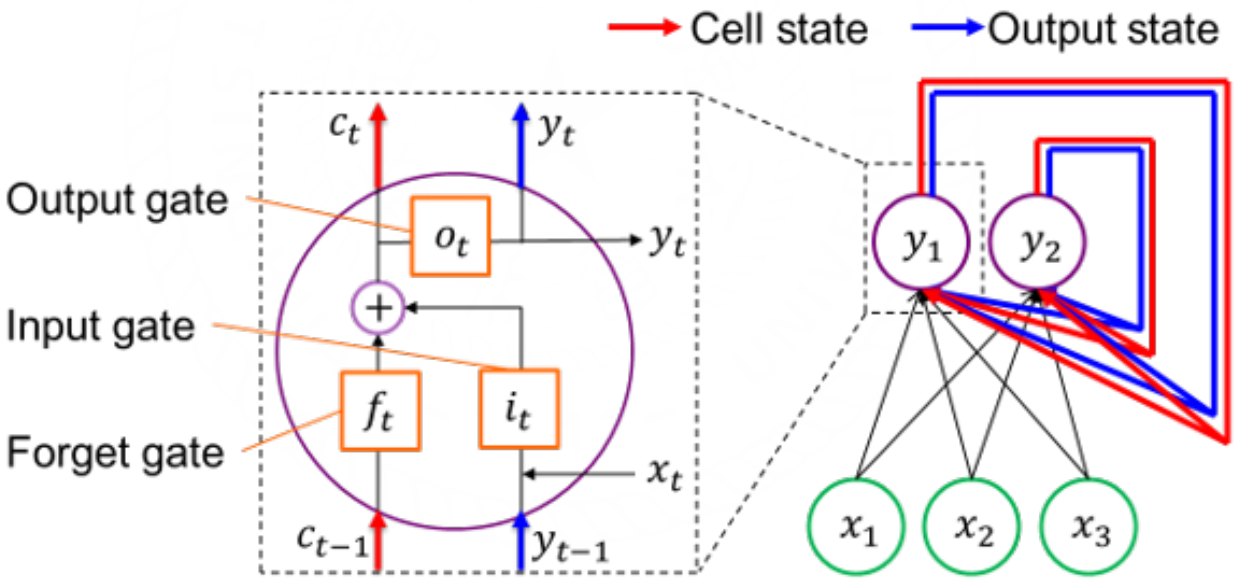


$$a^{\langle t \rangle} = \tanh\left(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a\right)$$

$$\hat{y}^{\langle t \rangle} = \text{soft max}\left(W_{ya}a^{\langle t \rangle} + b_y\right)$$

Recurrent layer

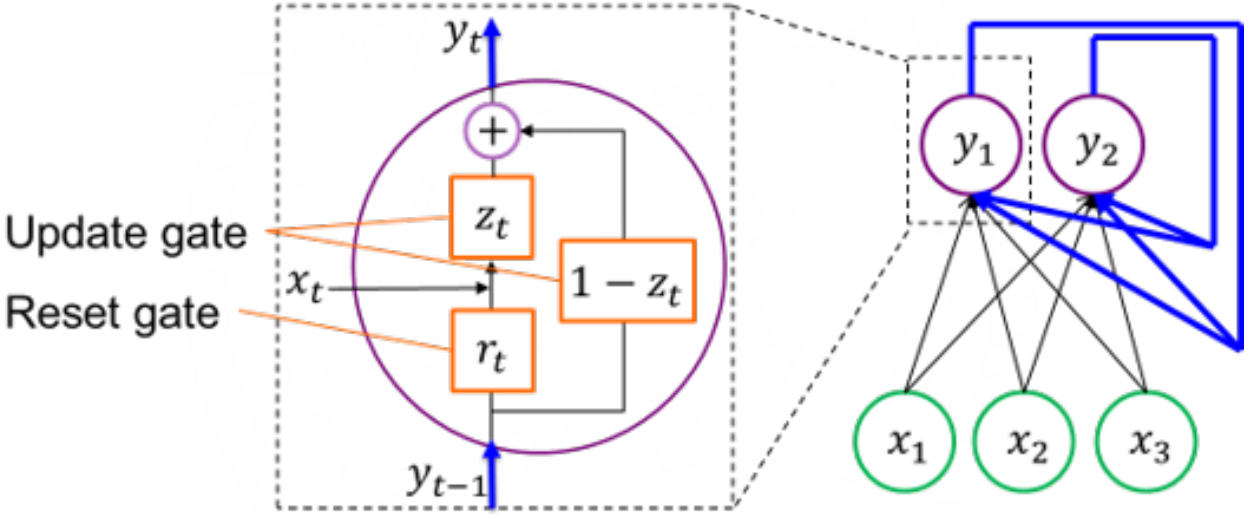
- (1) Long short-term memory (LSTM)
  - Gradients neither explode nor vanish through cell path
  - A cell is composed by input, forget and output gates
  - Gates regulate the information flow into and out of the cell



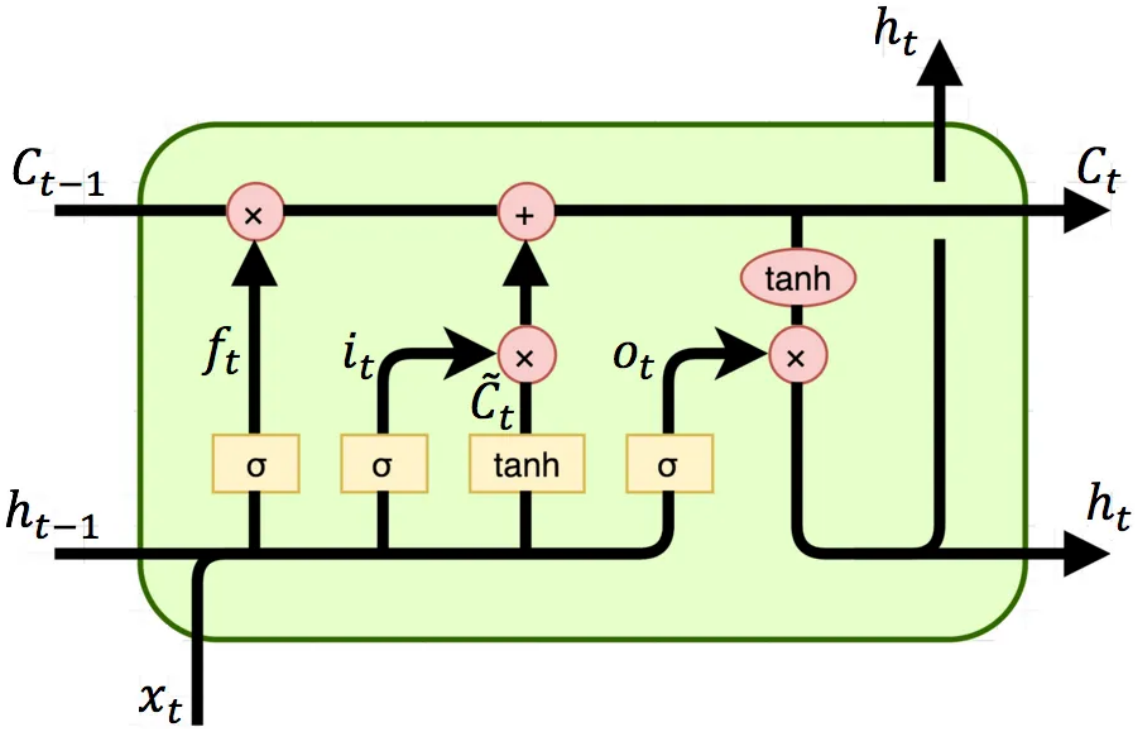
Recurrent layer

- (2) Gated recurrent unit (GRU)

  A cell is composed by an update gate and a reset gate



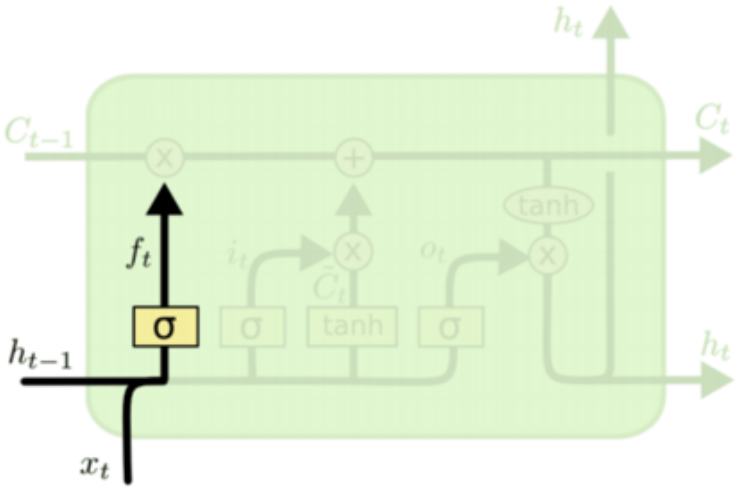
Recurrent layer - Long short-term memory (LSTM)



Long short-term memory (LSTM) – Forget Gate

The first gate determines whether to carry over the history or to forget it

- More precisely, how much of the history to carry over
- Also called the "forget" gate

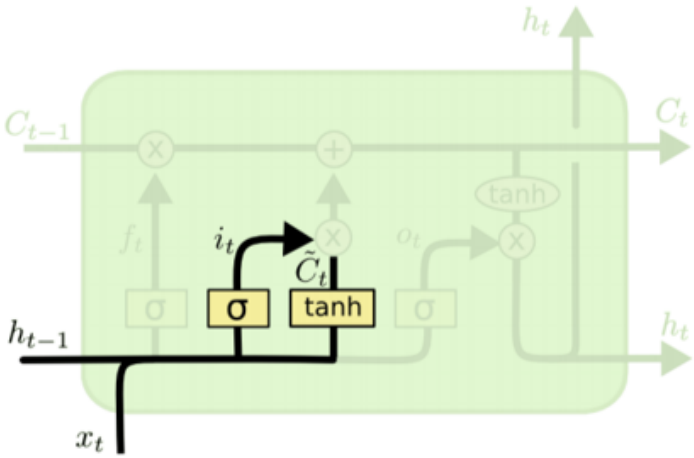


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Long short-term memory (LSTM) – Input Gate

The first gate determines whether to carry over the history or to forget it

- A perceptron layer that determines if something is interesting in the input
- A gate that decides if it's worth remembering



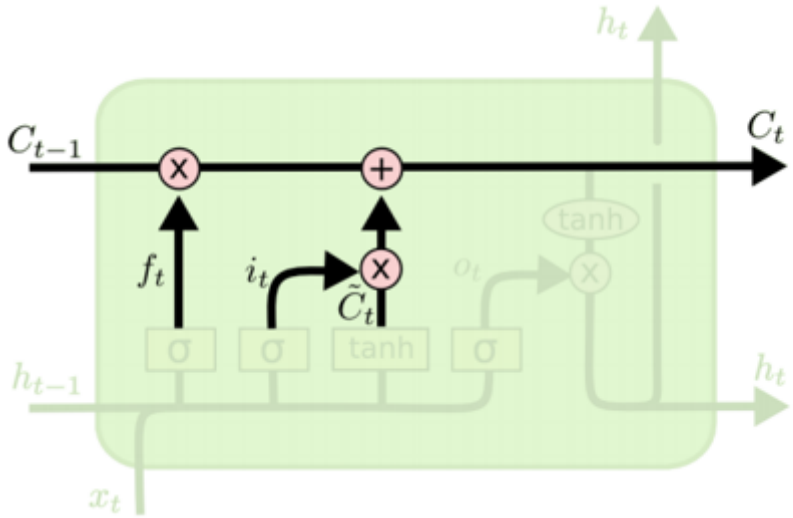
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Long short-term memory (LSTM) – Memory Cell Update

The second gate has two parts

- A perceptron layer that determines if there's something interesting in the input
- A gate that decides if its worth remembering
- If so its added to the current memory cell



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Long short-term memory (LSTM) – Output Gate

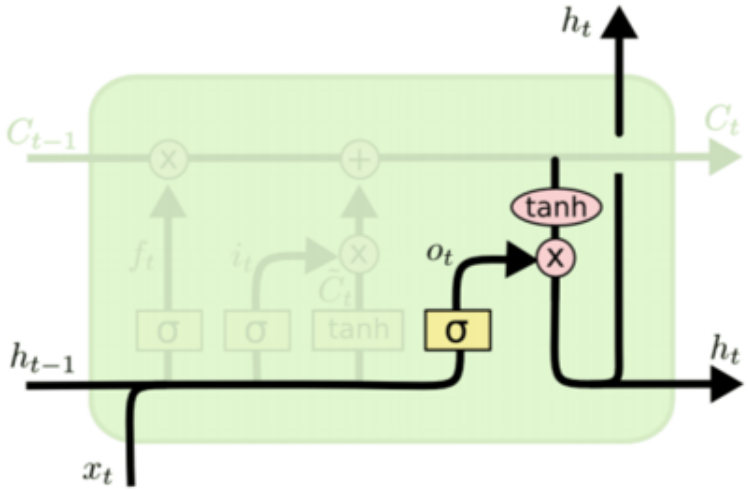
The output of the cell

- Simply compress it with tanh to make it lie between 1 and -1

  Note that this compression no longer affects our ability to carry memory forward

- While we're at it, lets toss in an output gate

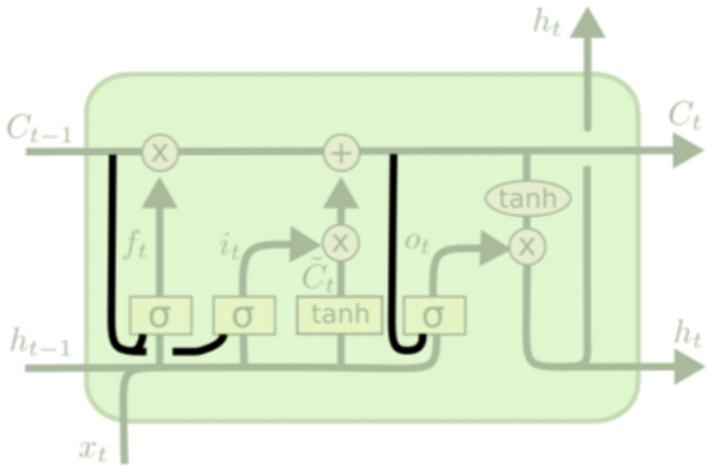
  To decide if the memory contents are worth reporting at this time



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

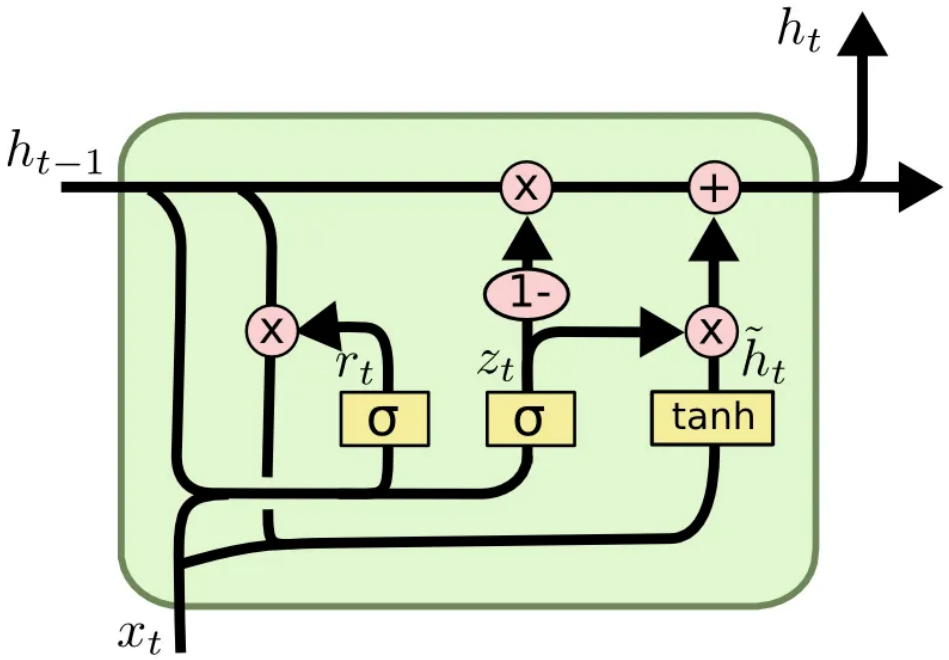
$$h_t = o_t * \tanh\left(C_t\right)$$

Long short-term memory (LSTM) – Peephole Connection



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Gated recurrent unit (GRU)



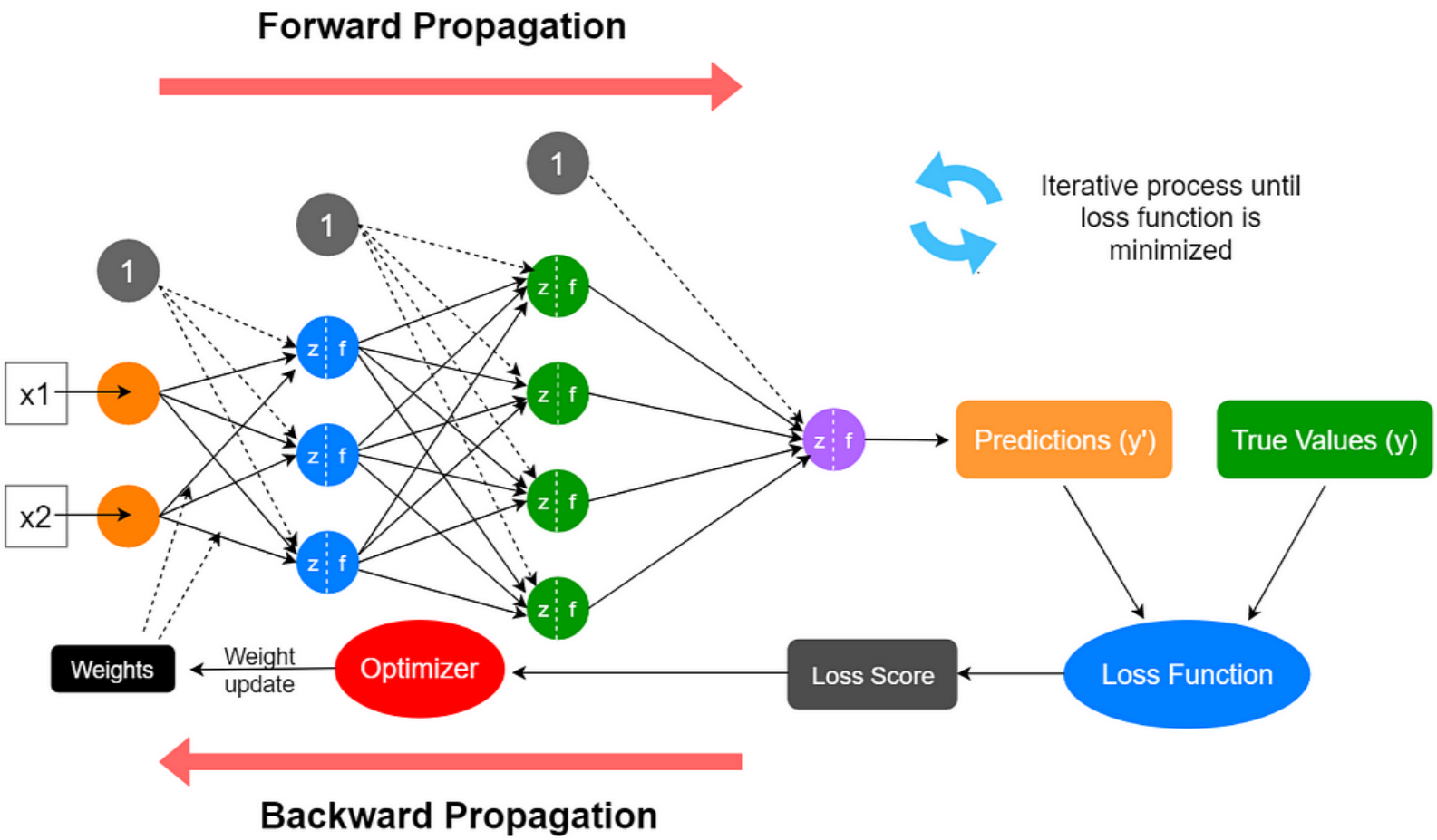
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

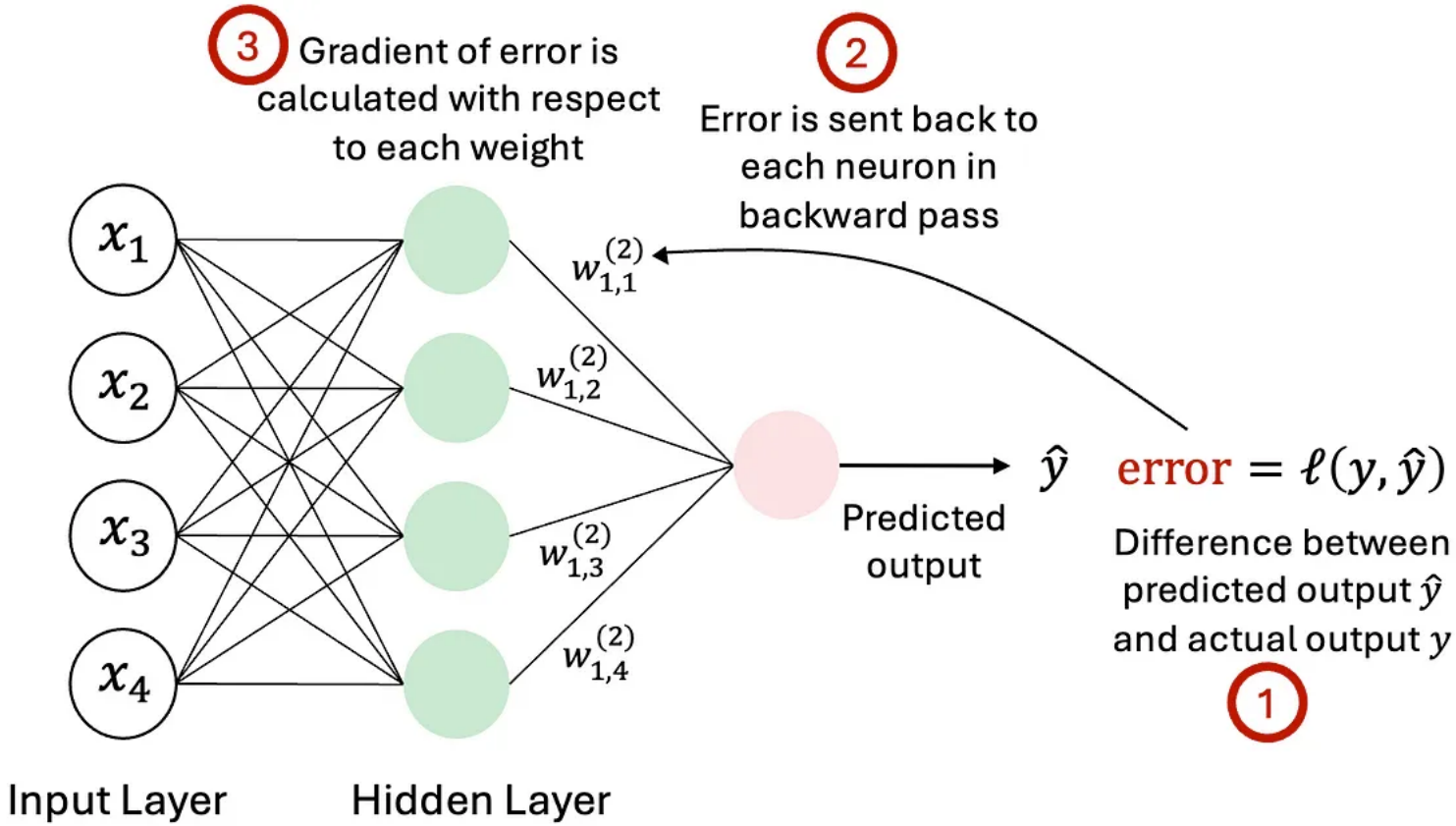
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Understanding Forward and Backward Propagation in Neural Networks



Understanding Forward and Backward Propagation in Neural Networks



Loss Functions in Training NN

Cross entropy

- Measure the error between predicted distribution p and target distribution p^{target}
- Often used for classification, where outputs are interpreted as membership probabilities

$$J \overset{\text{def}}{=} \mathcal{H}\left(p^{\text{target}}, p\right) = -\sum_i p_i^{\text{target}} \log\left(p_i\right)$$

Mean square error (MSE)

- The average of the squared differences between predicted values y and target values y^{target}
- Often used for regression, where outputs are quantities

$$J \overset{\text{def}}{=} \text{MSE}\left(y^{\text{target}}, y\right) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i^{\text{target}} - y_i\right)^2$$

$\rightarrow$ J is a function of y, and y is a function of w.

Training of Neural Network

Gradient Descent (GD)

- A first-order optimization algorithm that adjusts trainable weights according to the network error

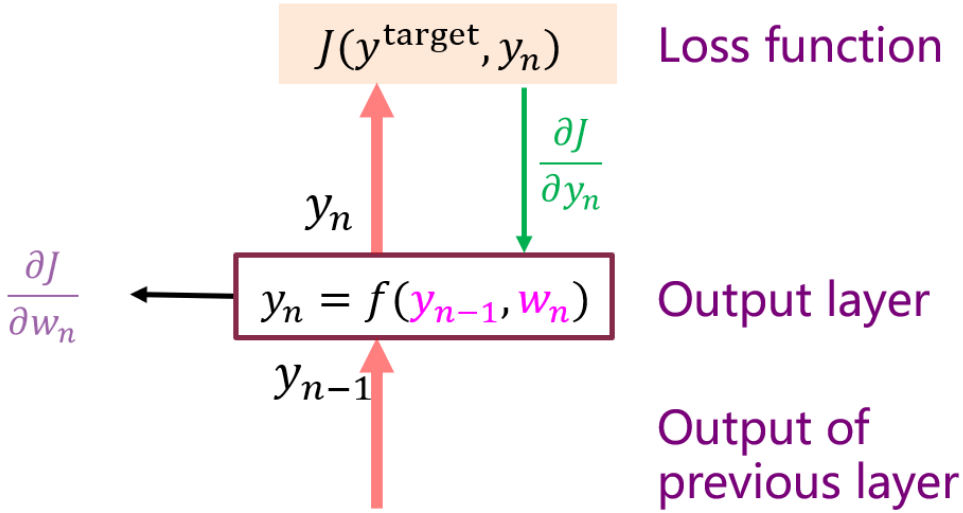
$$w \leftarrow w - \eta \frac{\partial J\left(y^{\text{target}}, y\right)}{\partial w}$$

$$\frac{\partial J\left(y^{\text{target}}, y\right)}{\partial w} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial w}$$

Backpropagation (BP)

- Allow the error information from the loss function to flow backward through each layer
- Compute the gradients of each layer by the chain rule with a specific order of operations

Backpropagation (BP) - for output layer



Only use in output layer

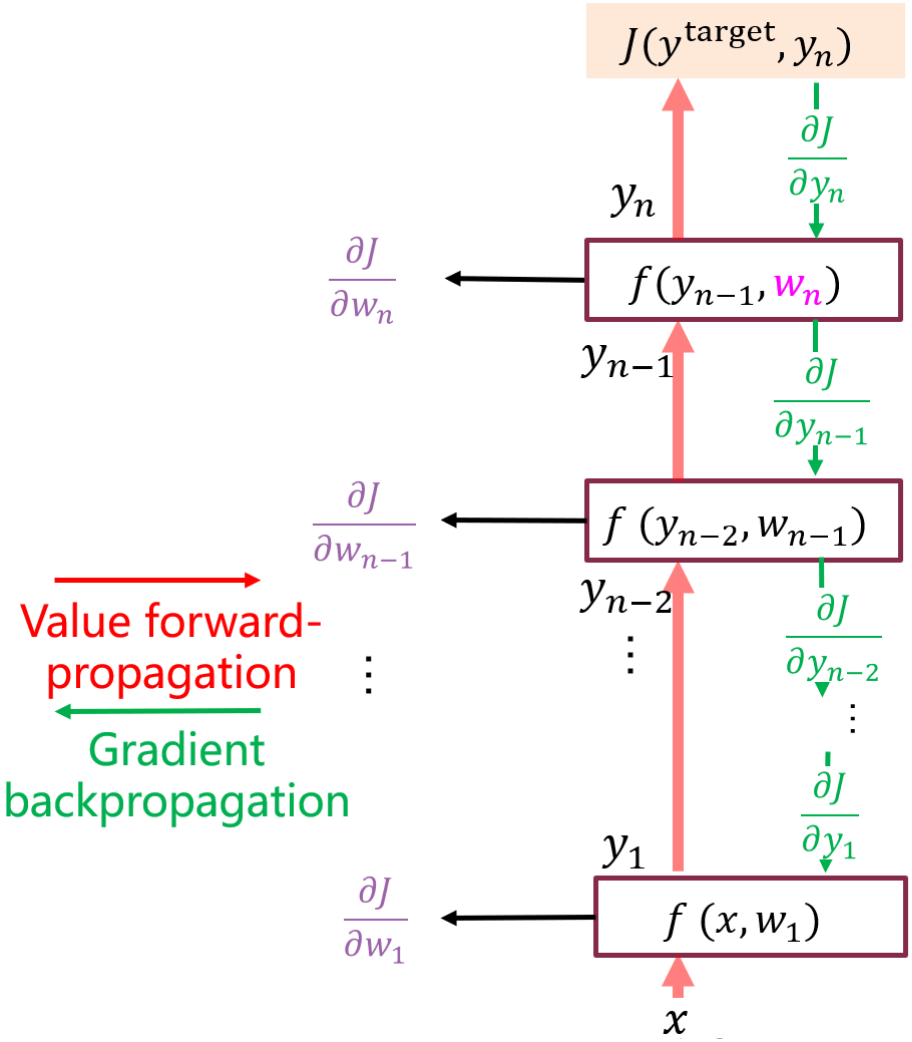
$$\frac{\partial J}{\partial w_n} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_n}$$

$$+$$

$$\frac{\partial J}{\partial y_{n-1}} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial y_{n-1}}$$

Reuse in subsequent operations

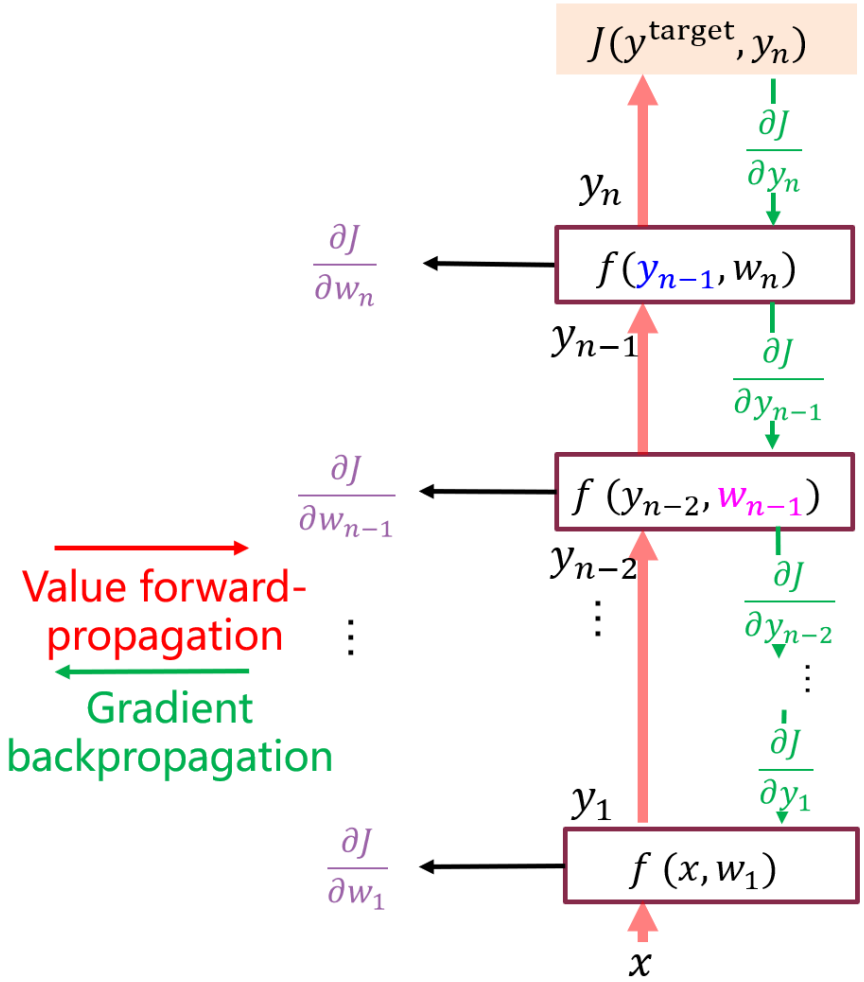
Backpropagation (BP)-1



$$\frac{\partial J}{\partial y_n} = \frac{\partial J(y^{\text{target}}, y_n)}{\partial y_n}$$

$$\frac{\partial J}{\partial w_n} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_n}$$

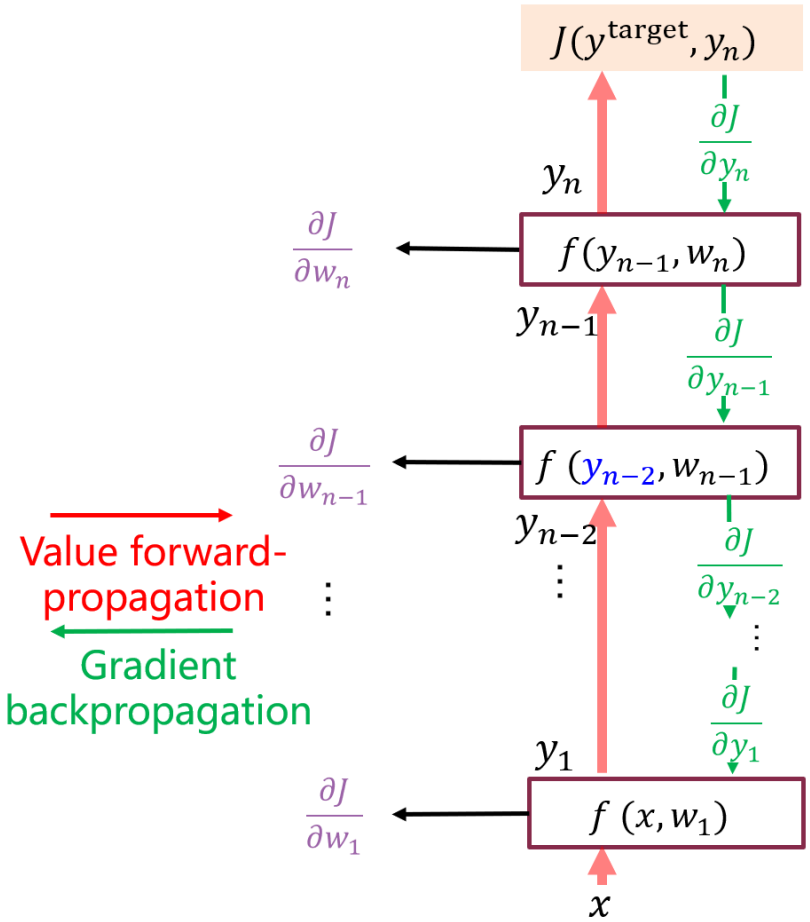
Backpropagation (BP)-2



$$\frac{\partial J}{\partial y_n} = \frac{\partial J(y^{\text{target}}, y_n)}{\partial y_n}$$

$$\frac{\partial J}{\partial w_n} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_n} \Rightarrow \text{Only use in output layer}$$

Backpropagation (BP)-3



$$\frac{\partial J}{\partial y_n} = \frac{\partial J(y^{\text{target}}, y_n)}{\partial y_n}$$

$$\frac{\partial J}{\partial w_n} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_n}$$ ➡ Only use in output layer

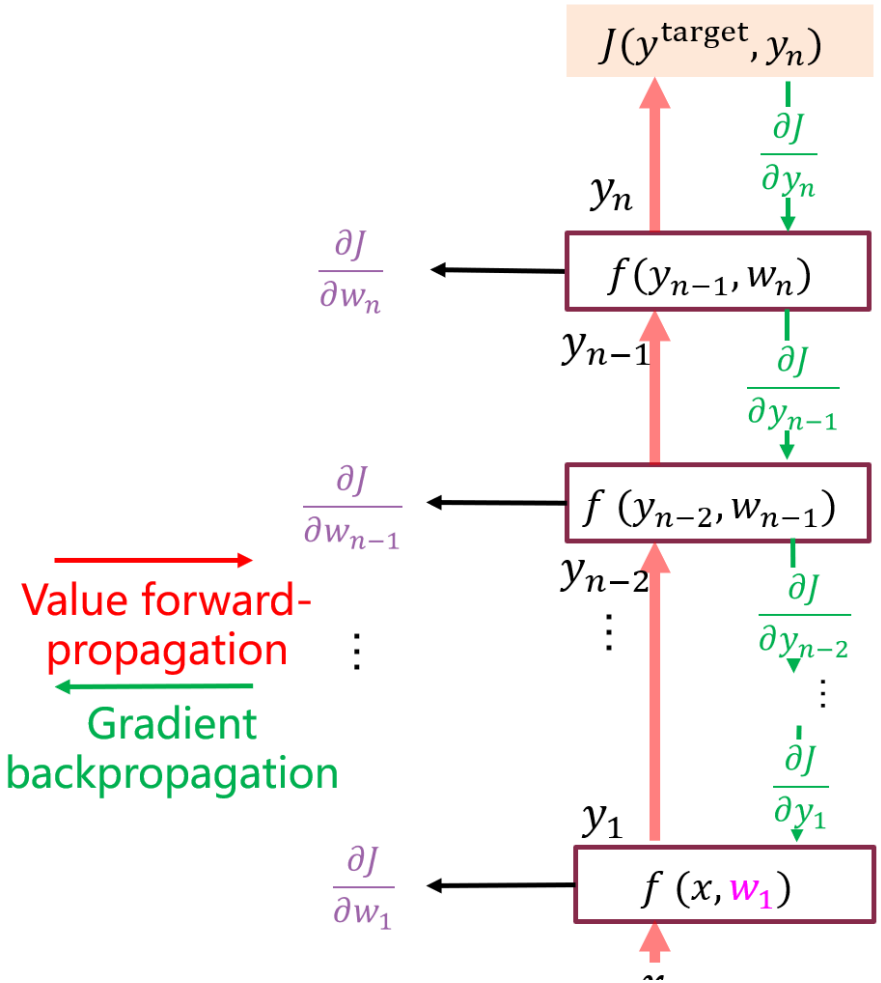
$$\frac{\partial J}{\partial y_{n-1}} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial y_{n-1}}$$

$$\frac{\partial J}{\partial w_{n-1}} = \frac{\partial J}{\partial y_{n-1}}\frac{\partial y_{n-1}}{\partial w_{n-1}}$$ ➡ Only use in Second last layer

$$\frac{\partial J}{\partial y_{n-2}} = \frac{\partial J}{\partial y_{n-1}}\frac{\partial y_{n-1}}{\partial y_{n-2}}$$

$$\frac{\partial J}{\partial w_{n-2}} = \frac{\partial J}{\partial y_{n-2}}\frac{\partial y_{n-2}}{\partial w_{n-2}}$$

Backpropagation (BP)-4



$$\frac{\partial J}{\partial w_n} = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_n}$$

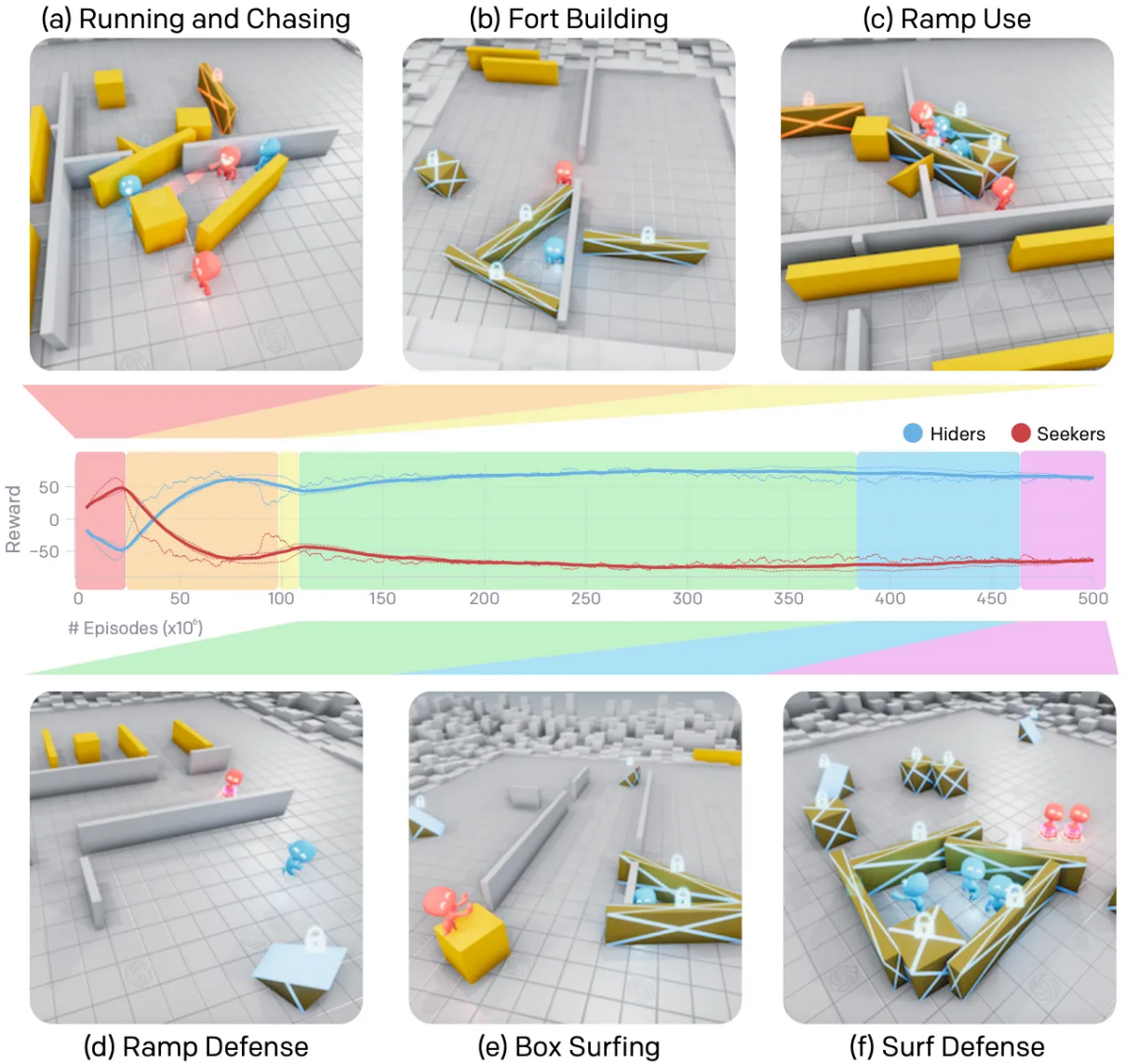
$$\frac{\partial J}{\partial w_{n-1}} = \frac{\partial J}{\partial y_{n-1}}\frac{\partial y_{n-1}}{\partial w_{n-1}}$$

... ...

$\frac{\partial J}{\partial w_1}$ → Loss function

$\frac{\partial J}{\partial w_1}$ → Weight of 1st layer

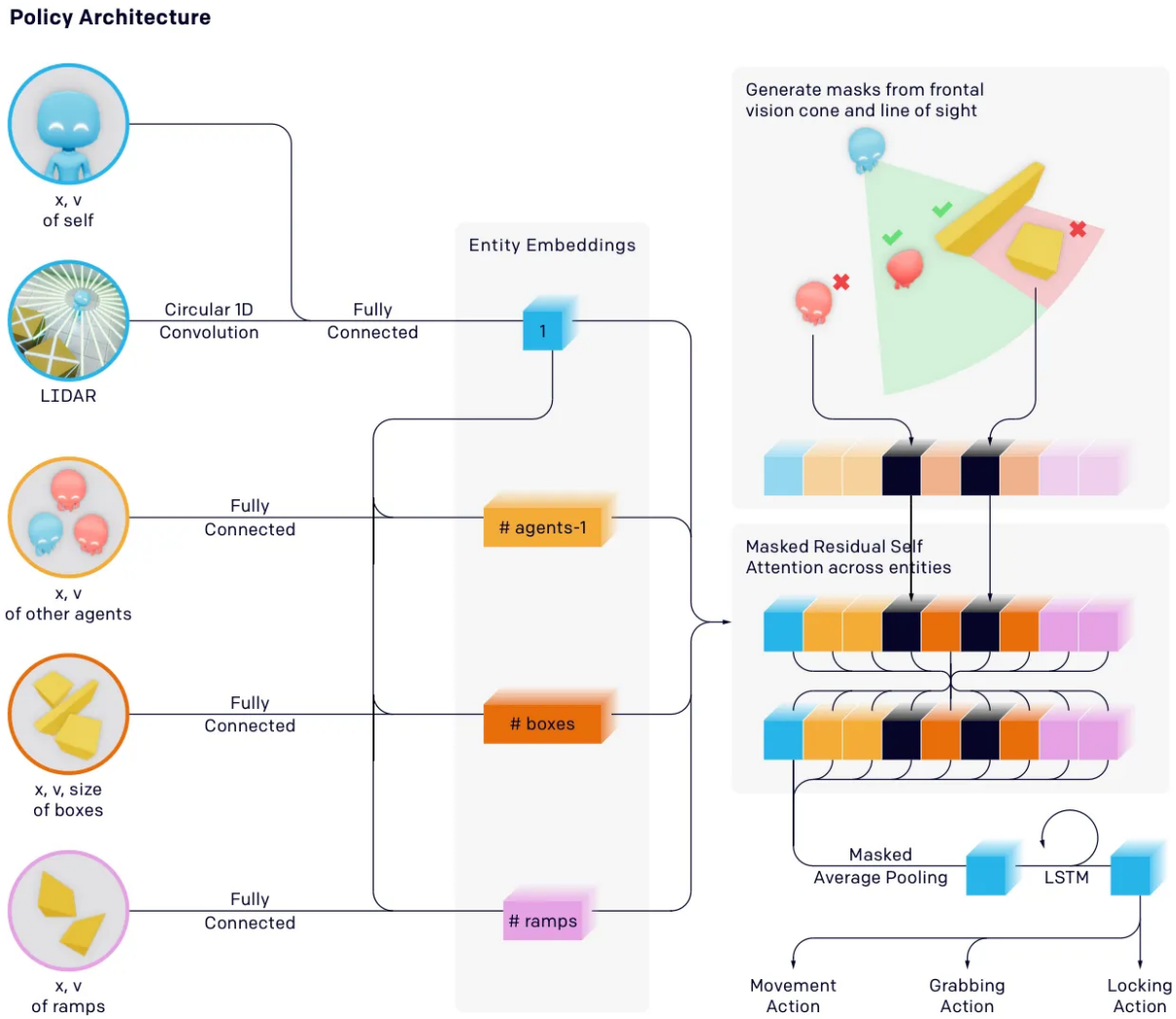
RL for multi-agent competitive game Hide-and-Seek (OpeAI Google Brain ICLR 20)



RL for multi-agent competitive game Hide-and-Seek

- Hide-and-seek is a multi-agent competitive game where hider agents are tasked to hide from the line of sight of seeker agents.
- The environment contains many objects scattered around, which can be grabbed and locked in place by both seekers and hiders.
- Agents are given a team-based reward
  - Hiders receive a reward of 1 if all hiders are hidden; otherwise, -1. Seekers receive a reward of 1 if any hider is found and -1 if all hiders are hidden.
  - Each episode lasts for 240 steps. The first 40% of the steps are for the hider's preparation, during which all agents receive 0 reward.
- Algorithm: All agents share the same policy but observe and act independently of one another. A centralized critic (value function) is used, where the input of the value function contains more information than that of the policy network.

State representation and policy network (source: original paper)



RL for multi-agent competitive game Hide-and-Seek

State components:

1. Agents observe the position (x), velocity (v), and size of themselves, objects, and other agents.
2. If entities are masked out if not in the line-of-sight or not in a 135-degree cone in front.
3. Agents also have 30 range sensors (LiDAR) evenly around them.
4. Each agent observes its team.
5. Each agent knows how much time is left in the preparation phase.

RL for multi-agent competitive game Hide-and-Seek

State encoding:

1. States are encoded as embeddings through multilayer perceptrons (MLPs).
2. Objects belonging to the same object types (boxes, agents, ramps) are encoded by the shared MLP.
3. one MLP for each object type.
4. All embeddings are concatenated and fed into the self-attention net layer, where out-of-sight objects (embeddings) are masked out.

RL for multi-agent competitive game Hide-and-Seek

Action space:

1. Agent can set discretized force along x, y and torque around z axis.
2. The agent has a binary action to either grab or lock objects.
3. Objects may be unlocked only by agents on the same team.
4. Agent may only grab or lock the object in front within a small radius.

RL for multi-agent competitive game Hide-and-Seek

Policy Network Architecture:

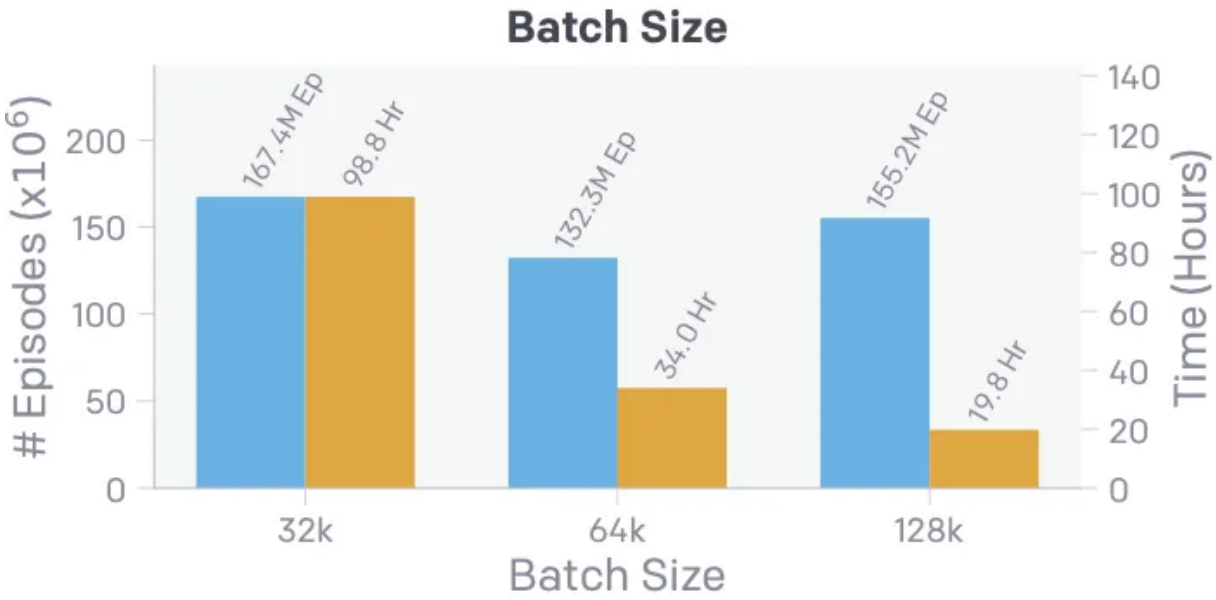
1. One MLP for each object type.
2. Self-attention layer.
3. LSTM layer.
4. MLP heads for move action, grab action and lock action, respectively.

Size of embedding layer	128
Size of MLP layer	256
Size of LSTM layer	256
Residual attention layer	4 attention heads of size 32
Weight decay coefficient	10^{-6}

RL for multi-agent competitive game Hide-and-Seek

This paper highlights the impact of batch size on performance and convergence.

- 64K batch size and 1.6 million parameters require 132.2 million episodes and 34 hours to converge
- A larger batch size leads to significantly quicker training time.
- Environments are randomized for each episode.



RL for multi-agent competitive game Hide-and-Seek

Performance stages

- Initially, crudely run away and chase
- 25 million episodes, hiders learn to use tools
- 75 million episodes seekers learn to use tools
- 85 million hiders learn to defend against the tool use strategy used by the seeker
- 380 million seekers learns to surf
- Eventually, hiders learn to lock all boxes before constructing a shelter to avoid the seekers' search.

https://openai.com/index/emergent-tool-use/