

## CS 540 Introduction to Artificial Intelligence **Review**

University of Wisconsin-Madison Spring 2024

## **Final Information**

- Time: May 7th 10:05 AM-12:05 PM
- Location (by section\*\*):
  - Noland 132: Section 001
  - Engineering Hall 1800: Section 003
  - Microbial Sciences 1220: Section 002

\*\*To find your section go to MyUW->Course Schedule->It will say "LEC 00\_". Do not use canvas to find your section (everyone will see CS540 001 since we merged the canvas site for all three sections).

- Format: The final exam will be entirely multiple choice.
- Cheat sheet: you will be allowed a cheat sheet of a single piece of paper (8.5" x 11", front and back). The exam will focus on conceptual and applied AI reasoning.
- Calculator: fine if it doesn't have an Internet connection
- Detailed topic list + practice: <u>https://piazza.com/class/lrjf9oinrox1zf/post/833</u>



# **Neural Networks**

## How to classify

Cats vs. dogs?





Neural networks can also be used for regression.

- Typically, no activation on outputs, mean squared error loss function.



#### Perceptron

• Given input **x**, weight **w** and bias *b*, perceptron outputs:  $o = \sigma(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b) \qquad \sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$ Activation function

Cats vs. dogs?









#### **Single Hidden Layer**

### How to classify Cats vs. dogs?







#### Neural networks with one hidden layer

Key elements: linear operations + Nonlinear activations



## Single Hidden Layer



 Normalize the output into probability using sigmoid 1

$$p(\mathbf{y} = 1 \,|\, \mathbf{x}) = \frac{1}{1 + e^{-f}}$$





#### **Multi-class classification**

Turns outputs *f* into *k* probabilities (sum up to 1 across *k* classes)



$$p(y|\mathbf{x}) = softmax(\mathbf{f})$$
$$= \frac{\exp f_y(x)}{\sum_{i}^{k} \exp f_i(x)}$$

## Deep neural networks (DNNs)



 $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$  $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$  $\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$  $\mathbf{f} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$  $\mathbf{y} = \text{softmax}(\mathbf{f})$ 

NNs are composition of nonlinear functions

## How to train a neural network?

Update the weights W to minimize the loss function

$$L = \frac{1}{|D|} \sum_{i} \ell(\mathbf{x}_i, y_i)$$

Use gradient descent!



## **Gradient Descent**

- Choose a learning rate  $\alpha > 0$
- Initialize the model parameters  $w_0$
- For *t* =1, 2, ...





• Repeat until converges

## **Minibatch Stochastic Gradient Descent**

- Choose a learning rate  $\alpha > 0$
- Initialize the model parameters  $w_0$
- For *t* =1, 2, ...
  - Randomly sample a subset (mini-batch)  $B \subset D$ Update parameters:

$$\mathbf{w}_{t} = \mathbf{w}_{t-1} - \alpha \frac{1}{|B|} \sum_{\mathbf{x} \in B} \frac{\partial \ell(\mathbf{x}_{i}, y_{i})}{\partial \mathbf{w}_{t-1}}$$

Repeat



- Want to compute  $\frac{\partial \ell(\mathbf{x}, y)}{\partial w_{11}}$
- Data point:  $((x_1, x_2), y)$



Use chain rule!



















# **Numerical Stability**

#### **Gradients for Neural Networks**

• Compute the gradient of the loss  $\ell$  w.r.t.  $\mathbf{W}_t$ 

$$\frac{\partial \ell}{\partial \mathbf{W}^{t}} = \frac{\partial \ell}{\partial \mathbf{h}^{d}} \frac{\partial \mathbf{h}^{d}}{\partial \mathbf{h}^{d-1}} \dots \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^{t}} \frac{\partial \mathbf{h}^{t}}{\partial \mathbf{W}^{t}}$$

Multiplication of *many* matrices



Wikipedia

#### **Two Issues for Deep Neural Networks**



#### Gradient Exploding



#### **Gradient Vanishing**



 $1.5^{100} \approx 4 \times 10^{17}$ 



#### **Issues with Gradient Exploding**

- Value out of range: infinity value (NaN)
- Sensitive to learning rate (LR)
  - Not small enough LR  $\rightarrow$  larger gradients
  - Too small LR  $\rightarrow$  No progress
  - May need to change LR dramatically during training

#### **Gradient Vanishing**

Use sigmoid as the activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \ \sigma'(x) = \sigma(x)(1 - \sigma(x))$$



#### **Issues with Gradient Vanishing**

- Gradients with value 0
- No progress in training
  No matter how to choose learning rate
- Severe with bottom layers (those near the input)
  Only top layers (near output) are well trained
  No benefit to make networks deeper



#### How to stabilize training?

#### **Stabilize Training: Practical Considerations**

- Goal: make sure gradient values are in a proper range
  - E.g. in [1e-6, 1e3]
- Multiplication  $\rightarrow$  plus
  - Architecture change (e.g., ResNet)
- Normalize
  - Batch Normalization, Gradient clipping
- Proper activation functions

Quiz. Which of the following are TRUE about the vanishing gradient problem in neural networks? Multiple answers are possible.

A.Deeper neural networks tend to be more susceptible to vanishing gradients.

B.Using the ReLU function can reduce this problem.

C. If a network has the vanishing gradient problem for one training point due to the

sigmoid function, it will also have a vanishing gradient for every other training point.

D. Networks with sigmoid functions don't suffer from the vanishing gradient problem if

trained with the cross-entropy loss.

Quiz. Which of the following are TRUE about the vanishing gradient problem in neural networks? Multiple answers are possible?

A.Deeper neural networks tend to be more susceptible to vanishing gradients.

B.Using the ReLU function can reduce this problem.

C. If a network has the vanishing gradient problem for one training point due to the

sigmoid function, it will also have a vanishing gradient for every other training point.

D. Networks with sigmoid functions don't suffer from the vanishing gradient problem if

trained with the cross-entropy loss.

Quiz. Let's compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Quiz. Let's compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Q5. A Leaky ReLU is defined as f(x)=max(0.1x, x). Let f'(0)=1. Does it have non-zero gradient everywhere??

A.Yes

B. No
Q5. A Leaky ReLU is defined as f(x)=max(0.1x, x). Let f'(0)=1. Does it have non-zero gradient everywhere??



B. No





# Generalization & Regularization

# How good are the models?



### **Training Error and Generalization Error**

- Training error: model error on the training data
- Generalization error: model error on new data
- Example: practice a future exam with past exams
  - Doing well on past exams (training error) doesn't guarantee a good score on the future exam (generalization error)

### **Influence of Model Complexity**



Model complexity

Quiz Break: When training a neural network, which one below indicates that the network has overfit the training data?

A. Training loss is low and generalization loss is high.

- B. Training loss is low and generalization loss is low.
- C. Training loss is high and generalization loss is high.
- D. Training loss is high and generalization loss is low.
- E. None of these.

Quiz Break: When training a neural network, which one below indicates that the network has overfit the training data?

A. Training loss is low and generalization loss is high.

- B. Training loss is low and generalization loss is low.
- C. Training loss is high and generalization loss is high.
- D. Training loss is high and generalization loss is low.
- E. None of these.

# Quiz Break: Adding more layers to a multi-layer perceptron may cause \_\_\_\_\_.

- A. Vanishing gradients during back propagation.
- B. A more complex decision boundary.
- C. Underfitting.
- D. Higher test loss.
- E. None of these.

Quiz Break: Adding more layers to a multi-layer perceptron may cause \_\_\_\_\_. (Multiple answers)

- A. Vanishing gradients during back propagation.
- B. A more complex decision boundary.
- C. Underfitting.
- D. Higher test loss.
- E. None of these.

# How to regularize the model for better generalization?

# Weight Decay

### Ó Training Error: 0.100 Test Error: 0.259 Bayes Error: 0.210

Neural Network - 10 Units, No Weight Decay

#### Neural Network - 10 Units, Weight Decay=0.02





### **Squared Norm Regularization as Hard Constraint**

 Reduce model complexity by limiting value range

 $minL(\mathbf{w}, b)$  subject to  $\|\mathbf{w}\|^2 \le B$ 

- Often do not regularize bias b
  - Doing or not doing has little difference in practice
- A small *B* means more regularization



#### **Squared Norm Regularization as Soft Constraint**

• We can rewrite the hard constraint version as

$$minL(\mathbf{w},b) + \frac{\lambda}{2} \parallel \mathbf{w} \parallel^2$$

### **Squared Norm Regularization as Soft Constraint**

• We can rewrite the hard constraint version as

$$minL(\mathbf{w},b) + \frac{\lambda}{2} \parallel \mathbf{w} \parallel^2$$

- Hyper-parameter  $\lambda$  controls regularization importance
- $\lambda = 0$ : no effect  $\lambda \to \infty, \mathbf{w}^* \to \mathbf{0}$

### **Illustrate the Effect on Optimal Solutions**

$$\mathbf{w}^* = \operatorname{argminL}(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$

$$\tilde{\mathbf{w}}^* = \operatorname{argminL}(\mathbf{w}, b)$$

### **Dropout** Hinton et al.



## **Apply Dropout**

• Often apply dropout on the output of hidden fully-connected layers



## Dropout



Figure 2: Left: A unit at training time that is present with probability p and is connected to units in the next layer with weights w. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

## Dropout

Hinton et al.



Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.



## Convolutional Neural Networks (CNNs)

### How to classify

Cats vs. dogs?







12MP

wide-angle and telephoto cameras

36M floats in a RGB image!

### **Fully Connected Networks**



~ 36M elements x 100 = ~3.6B parameters!

Where is Waldo?





### Why Convolution?

- Translation
   Invariance
- Locality



**2-D Convolution** 



 $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$   $1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$   $3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$  $4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$ 



(vdumoulin@ Github)

### **2-D Convolution Layer**



- **X**:  $n_h \times n_w$  input matrix
- **W**:  $k_h \times k_w$  kernel matrix
- b: scalar bias

• **Y**: 
$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$
 output matrix

 $\mathbf{Y} = \mathbf{X} \star \mathbf{W} + b$ 

• W and b are learnable parameters

### 2-D Convolution Layer with Stride and Padding

- Stride is the #rows/#columns per slide
- Padding adds rows/columns around input
- Output shape



$$\begin{bmatrix} (n_h - k_h + p_h + s_h)/s_h \end{bmatrix} \times \begin{bmatrix} (n_w - k_w + p_w + s_w)/s_w \end{bmatrix}$$
  
Input size Pad Stride

### **Multiple Input Channels**

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Have a kernel for each channel, and then sum results over channels

Input Kernel



### **Multiple Input Channels**

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Have a 2D kernel for each channel, and then sum results over channels



### **Multiple Input Channels**

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Also call each 3D kernel a "filter", which produce only one output channel (due to summation over channels)



### Multiple filters (in one layer)

- Apply multiple filters on the input
- Each filter may learn different features about the input
- Each filter (3D kernel) produces one output channel





### **Multiple Output Channels**

- The # of output channels = # of filters
- Input **X**:  $c_i \times n_h \times n_w$
- Kernel **W**:  $c_o \times c_i \times k_h \times k_w$
- Output **Y**:  $c_o \times m_h \times m_w$

$$Y_{i,:,:} = X \star W_{i,:,:,:}$$
  
for  $i = 1, ..., c_o$ 

### Pooling

Let us assume filter is an "eye" detector.

**Q.:** how can we make the detection robust to the exact location of the eye?

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

### Pooling

By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

### 2-D Max Pooling

• Returns the maximal value in the sliding window



max(0,1,3,4) = 4

### 2-D Max Pooling

 Returns the maximal value in the sliding window

Input







Output



max(0,1,3,4) = 4
### Padding, Stride, and Multiple Channels

- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel

### **#output channels = #input channels**

### Padding, Stride, and Multiple Channels

- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel



### **#output channels = #input channels**

### **Average Pooling**

- Max pooling: the strongest pattern signal in a window
- Average pooling: replace max with mean in max pooling
  - The average signal strength in a window

Max pooling



Average pooling



Consider a convolution layer with 16 filters. Each filter has a size of 11x11x3, a stride of 2x2. Given an input image of size 22x22x3, if we don't allow a filter to fall outside of the input, what is the output size?

- 11x11x16
- 6x6x16
- 7x7x16
- 5x5x16

Consider a convolution layer with 16 filters. Each filter has a size of 11x11x3, a stride of 2x2. Given an input image of size 22x22x3, if we don't allow a filter to fall outside of the input, what is the output size?

#### • 11x11x16

• 6x6x16

$$[(n_h - k_h + p_h + s_h)/s_h] \times [(n_w - k_w + p_w + s_w)/s_w]$$

- 7x7x16
- 5x5x16

# **Evolution of CNNs**

### ImageNet competition (error rate)



Credit: Stanford CS 231n

# Simple Idea: Add More Layers

VGG: 19 layers. ResNet: 152 layers. **Add more layers**... sufficient?

- No! Some problems:
  - i) Vanishing gradients: more layers  $\rightarrow$  more likely
  - ii) Instability: deeper models are harder to optimize





He et al: "Deep Residual Learning for Image Recognition"

# **Residual Connections**

- **Idea**: Identity might be hard to learn, but zero is easy!
- Make all the weights tiny, produces zero for output
- Can easily transform learning identity to learning zero:



Left: Conventional layers block

Right: Residual layer block

To learn identity f(x) = x, layers now need to learn  $f(x) = 0 \rightarrow$  easier



# **Uninformed Search**









queue (fringe, OPEN)  $\rightarrow$  [A]  $\rightarrow$ 





queue (fringe, OPEN)  $\rightarrow$  [CB]  $\rightarrow$  A





queue (fringe, OPEN)  $\rightarrow$  [EDC]  $\rightarrow$  B





queue (fringe, OPEN)  $\Box$ [GFED]  $\rightarrow$  C

If G is a goal, we've seen it, but we don't stop!

Use a queue (First-in First-out)

- 1. en\_queue(Initial states)
- 2. While (queue not empty)
- 3. s = de\_queue()
- 4. if (s==goal) success!
- **5.** T = succs(s)
- 6. en\_queue(T)
- 7. endWhile



queue □[] **→**G

... until much later we pop G.

Looking foolish? Indeed. But let's be consistent...

Use a queue (First-in First-out)

- 1. en\_queue(Initial states)
- 2. While (queue not empty)
- 3. s = de\_queue()
- 4. if (s==goal) success!
- **5**. T = succs(s)
- 6. en\_queue(T)
- 7. endWhile



queue □[] **→**G

... until much later we pop G.

We need back pointers to recover the solution path.

Looking foolish? Indeed. But let's be consistent...

#### **Performance of search algorithms on trees**

b: branching factor (assume finite)

d: goal depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if <sup>1</sup>	O(b <sup>d</sup> )	O(b <sup>d</sup> )

1. Edge cost constant, or positive non-decreasing in depth

#### **Uniform-cost search**

- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path).
- Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item

#### Example



(All edges are directed, pointing downwards)

#### Performance of search algorithms on trees

b: branching factor (assume finite)

d: goal depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if <sup>1</sup>	O(b <sup>d</sup> )	O(b <sup>d</sup> )
Uniform-cost search <sup>2</sup>	Y	Y	O(b <sup>C*/ε</sup> )	O(b <sup>C*/ε</sup> )

1. edge cost constant, or positive non-decreasing in depth

2. edge costs  $\geq \varepsilon > 0$ . C\* is the best goal path cost.

#### **Depth-first search (DFS)**





stack (fringe)

1. A, [B, C] 2. B, [D, E, C] 3. D, [E, C] 4. E, [C] 5. C, [F, G] 6. F, [G] 7. G

#### **Performance of search algorithms on trees**

b: branching factor (assume finite)

d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if <sup>1</sup>	O(b <sup>d</sup> )	O(b <sup>d</sup> )
Uniform-cost search <sup>2</sup>	Y	Y	O(b <sup>C*/ε</sup> )	O(b <sup>C*/ε</sup> )
Depth-first search	Ν	Ν	O(b <sup>m</sup> )	O(bm)

1. edge cost constant, or positive non-decreasing in depth

2. edge costs  $\geq \varepsilon > 0$ . C\* is the best goal path cost.

#### **Iterative deepening**

- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Preferred uninformed search method

#### Example



(All edges are directed, pointing downwards)

#### Nodes expanded by:



- Breadth-First Search: S A B C D E G Solution found: S A G
- Uniform-Cost Search: S A D B C E G Solution found: S B G (This is the only uninformed search that worries about costs.)
- Depth-First Search: S A D E G Solution found: S A G
- Iterative-Deepening Search: S A B C S A D E G Solution found: S A G

#### **Performance of search algorithms on trees**

b: branching factor (assume finite)

d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if <sup>1</sup>	O(b <sup>d</sup> )	O(b <sup>d</sup> )
Uniform-cost search <sup>2</sup>	Y	Y	O(b <sup>C*/ε</sup> )	O(b <sup>C*/ε</sup> )
Depth-first search	Ν	Ν	O(b <sup>m</sup> )	O(bm)
Iterative deepening	Y	Y, if <sup>1</sup>	O(b <sup>d</sup> )	O(bd)

1. edge cost constant, or positive non-decreasing in depth

2. edge costs  $\geq \varepsilon > 0$ . C\* is the best goal path cost.



# **Informed Search**

# Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost **g**(s) from start to node s
- Successors.



Informed search. Know:

- All uninformed search properties, plus
- Heuristic h(s) from s to goal (recall game heuristic)

### **Recap and Examples**

**Example** for A\*:



## **Recap and Examples**





# Games

## Games Setup

### Games setup: multiple agents

—



## Normal Form Game

Mathematical description of simultaneous games.

- *n* players {1,2,...,*n*}
- Player *i* chooses strategy  $a_i$  from action space  $A_i$ .
- Strategy profile: *a* = (*a*<sub>1</sub>, *a*<sub>2</sub>, ..., *a<sub>n</sub>*)
- Player *i* gets rewards *u<sub>i</sub>*(*a*)
  Note: reward depends on other players!
- We consider the simple case where all reward functions are common knowledge.

# Example of Normal Form Game **Ex**: Prisoner's Dilemma

Player 2	Stay silent	Betray
Player 1		
Stay silent	-1, -1	-3, 0
Betray	0, -3	-2, -2

- •2 players, 2 actions: yields 2x2 payoff matrix
- Strategy set: {Stay silent, betray}

## Strictly Dominant Strategies

Let's analyze such games. Some strategies are better than others!

 Strictly dominant strategy: if a<sub>i</sub> strictly better than b regardless of what other players do, a<sub>i</sub> is strictly dominant

• I.e., 
$$u_i(a_i, a_{-i}) > u_i(b, a_{-i}), \forall b \neq a_i, \forall a_{-i}$$

All of the other entries of *a* excluding *i* 

• Sometimes a dominant strategy does not exist!

## Strictly Dominant Strategies Example

### Back to Prisoner's Dilemma

- Examine all the entries: betray strictly dominates
- Check:

Player 2 Player 1	Stay silent	Betray
Stay silent	-1, -1	-3, 0
Betray	0, -3	-2, -2
Dominant Strategy Equilibrium  $a^*$  is a (strictly) dominant strategy equilibrium (DSE), if every player *i* has a strictly dominant strategy  $a_i^*$ 

• Rational players will play at DSE, if one exists.

Player 2	Stay silent	Betray
Player 1		
Stay silent	-1, -1	-3, 0
Betray	0, -3	-2, -2

Dominant Strategy: Absolute Best Responses Player *i*'s best response to strategy to  $a_{-i}BR(a_{-i}) = \arg\max_{b} u_i(b, a_{-i})$ 

BR(player2=silent) = betray BR(player2=betray) = betray

Player 2	Stay silent	Betray
Player 1		
Stay silent	-1, -1	-3, 0
Betray	0, -3	-2, -2

 $a_i^*$  is the dominant strategy for player *i*, if  $a_i^* = BR(a_{-i}), \forall a_{-i}$ 

### **Dominant Strategy Equilibrium**

# Dominant Strategy Equilibrium does not always exist.

Player 2 Player 1	L	R
Т	2, 1	0, 0
В	0, 0	1, 2

### Nash Equilibrium

# a\* is a Nash equilibrium if no player has an incentive to unilaterally deviate

$$u_i(a_i^*, a_{-i}^*) \ge u_i(a_i, a_{-i}^*) \quad \forall a_i \in A_i$$



### Nash Equilibrium: Best Response to Each Other

 $a^*$  is a Nash equilibrium:

 $\forall i, \forall b \in A_i: u_i(a_i^*, a_{-i}^*) \ge u_i(b, a_{-i}^*)$ 

(no player has an incentive to **unilaterally deviate**)

- Equivalently, for each player i:  $a_i^* \in BR(a_{-i}^*) = argmax_b u_i(b, a_{-i}^*)$
- Compared to DSE (a DSE is a NE, the other direction is generally not true):

 $a_i^* = BR(a_{-i}), \forall a_{-i}$ 

### Nash Equilibrium: Best Response to Each Other

 $a^*$  is a Nash equilibrium:

 $\forall i, \forall b \in A_i: u_i(a_i^*, a_{-i}^*) \ge u_i(b, a_{-i}^*)$ 

(no player has an incentive to **unilaterally deviate**)

- Pure Nash equilibrium:
  - A **pure strategy** is a deterministic choice (no randomness).
  - Later: we will consider **mixed** strategies
  - In pure Nash equilibrium, players can only play pure strategies.

### Finding (pure) Nash Equilibria by hand

• As player 1: For each column, find the best response, underscore it.

Player 2	L	R
Паусі і		
Т	2, 1	0, 0
В	0, 0	1, 2

### Finding (pure) Nash Equilibria by hand

• As player 2: For each row, find the best response, upper-score it.

Player 2	I	R
Player 1	_	
Т	2, 1	0, 0
В	0, 0	1, 2

### Finding (pure) Nash Equilibria by hand

• Entries with both lower and upper bars are pure NEs.

Player 2	1	D	
Player 1		Λ	
Т	2, 1	0, 0	
В	0, 0	1, 2	

### Pure Nash Equilibrium may not exist

So far, pure strategy: each player picks a deterministic strategy. But:

Player 2	rock	paper	scissors
Player 1			
rock	0, 0	-1, 1	<u>1, -1</u>
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

### **Mixed Strategies**

Can also randomize actions: "mixed"

• Player *i* assigns probabilities  $x_i$  to each action

$$x_i(a_i)$$
, where  $\sum_{a_i \in A_i} x_i(a_i) = 1, x_i(a_i) \ge 0$ 

• Now consider **expected rewards** 

$$u_i(x_i, x_{-i}) = E_{a_i \sim x_i, a_{-i} \sim x_{-i}} u_i(a_i, a_{-i}) = \sum_{a_i} \sum_{a_{-i}} x_i(a_i) x_{-i}(a_{-i}) u_i(a_i, a_{-i})$$

## Mixed Strategy Nash Equilibrium Example: $x_1^*(\cdot) = x_2^*(\cdot) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$

Player 2	rock	naper	scissors
Player 1	TOCK	рарсі	30/330/3
rock	0, 0	-1, 1	1, -1
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

### Sequential-Move Games

More complex games with multiple moves

- Instead of normal form, extensive form
- Represent with a **tree**
- Rewards at leaves
- Find strategies: perform search over the tree

- Nash equilibrium still well-defined
  - Backward induction

U'

 $D^{\prime}$ 

U'

D'

U

D

2

(0,0)

(2,1)

(1,2)

(3,1)







#### min

The execution on the terminal nodes is omitted.





















### Our Approach So Far

We find the minimax value/strategy bottom up

- Minimax value: score of terminal node when both players play optimally
  - Max's turn, take max of children
  - Min's turn, take min of children

• Can implement this as depth-first search: **minimax algorithm** 

### Minimax Algorithm

```
function Max-Value(s)
inputs:
```

```
s: current state in game, Max about to play
output: best-score (for Max) available from s
```

```
if ( s is a terminal state )
then return ( terminal value of s )
else
```

```
\alpha := - infinity
for each s' in Succ(s)
\alpha := max(\alpha, Min-value(s'))
return \alpha
```

```
function Min-Value(s)
output: best-score (for Min) available from s
```

```
if (s is a terminal state)
then return (terminal value of s)
else
```

```
 \begin{array}{l} \beta := infinity \\ for each s' in Succs(s) \\ \beta := min(\beta, Max-value(s')) \\ return \beta \end{array}
```

Time complexity?

```
• O(b<sup>m</sup>)
```

Space complexity?

• O(bm)

### Break & Quiz

**Q 2.1**: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- **B. 65**
- C. 41
- D. 2

### Break & Quiz

**Q 2.1**: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- B. 65 (1 + 4 + 4\*3 + 4\*3\*2 + 4\*3\*2 = 65. Note the root and leaf nodes.)
- C. 41
- D. 2



## **Reinforcement Learning**

### **Building The Theoretical Model**

Basic setup:

- Set of states, S
- Set of actions A



- Information: at time *t*, observe state  $s_t \in S$ . Get reward  $r_t$
- Agent makes choice  $a_t \in A$ . State changes to  $s_{t+1}$ , continue

Goal: find a map from states to actions maximize rewards.

### Markov Decision Process (MDP)

The formal mathematical model:

- State set S. Initial state s<sub>0.</sub> Action set A
- State transition model:  $P(s_{t+1}|s_t, a_t)$ 
  - Markov assumption: transition probability only depends on  $s_t$  and  $a_t$ , and not previous actions or states.
- Reward function: **r**(s<sub>t</sub>)
- **Policy**:  $\pi(s) : S \to A$ , action to take at a particular state.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

### **Discounting Rewards**

One issue: these are infinite series. **Convergence**?

• Solution

$$U(\mathbf{s}_0, \mathbf{s}_1 \dots) = r(\mathbf{s}_0) + \gamma r(\mathbf{s}_1) + \gamma^2 r(\mathbf{s}_2) + \dots = \sum_{t \ge 0} \gamma^t r(\mathbf{s}_t)$$

- Discount factor γ between 0 and 1
  - Set according to how important present is VS future
  - Note: has to be less than 1 for convergence

### Values and Policies

- Now that  $V^{\pi}(s_0)$  is defined what *a* should we take?
  - First, set V\*(s) to be expected utility for **optimal** policy from s
  - What's the expected utility of an action?
    - Specifically, action a in state s?



### **Obtaining the Optimal Policy**

Assume, we know the expected utility of an action.

• So, to get the optimal policy, compute

$$\pi^{*}(s) = \operatorname{argmax}_{a} \sum_{s'} P(s'|s, a) V^{*}(s')$$
  
All the states we Transition Expected could go to probability rewards


#### **Bellman Equations**

Let's walk over one step for the value function:



### Q-Learning

- Our **next** reinforcement learning algorithm.
- Does not require knowing r or P. Learn from data of the form:{(s<sub>t</sub>, a<sub>t</sub>, r<sub>t</sub>, s<sub>t+1</sub>)}.
- Learns an action-value function Q\*(s,a) that tells us the expected value of taking a in state s.

• Note: 
$$V^*(s) = \max_a Q^*(s, a)$$
.

• Optimal policy is formed as  $\pi^*(s) = \arg\max_a Q^*(s, a)$ 

#### **Q-Learning Iteration**

#### How do we get Q(*s*,*a*)?

• Iterative procedure  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r(s_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$ Learning rate

**Idea**: combine old value and new estimate of future value. Note: We are using a policy to take actions; based on the estimated Q!

#### Q-Learning

Estimate  $Q^{*}(s,a)$  from data {( $s_t, a_t, r_t, s_{t+1}$ )}:

- 1. Initialize Q(.,.) arbitrarily (eg all zeros)
  - 1. Except terminal states Q(s<sub>terminal</sub>,.)=0
- 2. Iterate over data until Q(.,.) converges:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$
  
Learning rate

#### **Exploration Vs. Exploitation**

General question!

• **Exploration:** take an action with unknown consequences

- Pros:

- Get a more accurate model of the environment
- Discover higher-reward states than the ones found so far

**Cons**:

- When exploring, not maximizing your utility
- Something bad might happen
- **Exploitation:** go with the best strategy found so far

- Pros:

- Maximize reward as reflected in the current utility estimates
- Avoid bad stuff

Cons:

Might prevent you from discovering the true optimal strategy

#### Q-Learning: ε-Greedy Behavior Policy

Getting data with both **exploration and exploitation** 

 With probability ε, take a random action; else the action with the highest (current) Q(s,a) value.

$$a = \begin{cases} \operatorname{argmax}_{a \in A} Q(s, a) & \operatorname{uniform}(0, 1) > \epsilon \\ \operatorname{random} a \in A & \operatorname{otherwise} \end{cases}$$

## Q-learning Algorithm

Input: step size  $\alpha$ , exploration probability  $\epsilon$ 

- 1. set Q(s,a) = 0 for all s, a.
- 2. For each episode:
- 3. Get initial state s.
- 4. While (*s* not a terminal state):

Explore: take action to see what happens.

5. Perform  $a = \epsilon$ -greedy(Q, s), receive r, s'

6. 
$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'}Q(s',a'))$$

Update action-value based on result.

8. End While

7.  $s \leftarrow s'$ 

9. End For



# Thank you and good luck!