A Gentle Introduction to Machine Learning

Second Lecture

Part I

Olov Andersson, AIICS

LINKÖPINGS UNIVERSITET

Recap from Last Lecture

Last lecture we talked about **supervised Learning**

- Definition
  - Learn **unknown function** y=f(x) given examples of (x,y)
- Choose a model, e.g. NN, and **train** it on examples
  - Set **loss function** (e.g. square loss) between model and examples
  - Train model parameters via gradient descent (local minima)
- Trend: Neural Networks and Deep Learning

Artificial Neural Networks – Summary

Advantages

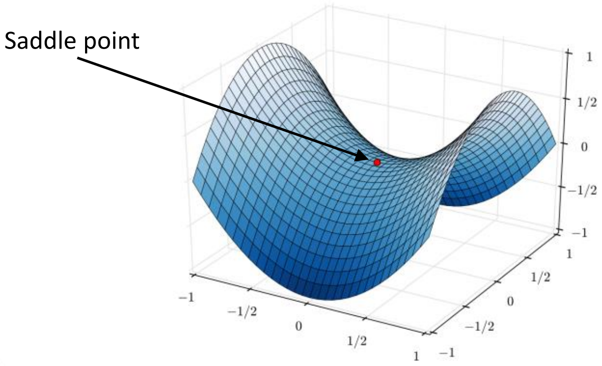
- Under some conditions it is a **universal approximator** to any function **f(x)**
  - E.g. It is very flexible, a large ”hypothesis space” in book terminology
- Some biological justification (real NNs more complex)
- Can be layered to capture abstraction (**deep learning**)
  - Used for speech, object and text recognition at Google, MSFT etc.
  - For best results use architectures tailored to input type (see DL lecture)
  - Often using millions of neurons/parameters and GPU acceleration.
- Modern **GPU-accelerated tools** for large models and Big Data
  - Tensorflow (Google), PyTorch (Facebook), Theano etc.

Disadvantages

- Training is a **non-convex** problem with **saddle points** and **local minima**
- **Many tuning parameters** (number of neurons, layers, starting weights, gradient scaling...)
- **Difficult to interpret** or debug weights in the network

What Was a Saddle Point Again?

- Gradient is zero, but not a minima
  - Loss could be decreased but gradient descent is stuck
- Believed to be a more common problem than local minima for ANN



Outline of This Lecture

Wrap up **supervised learning**

- **Pitfalls & Limitations**
- **Robotics applications**

Reinforcement Learning

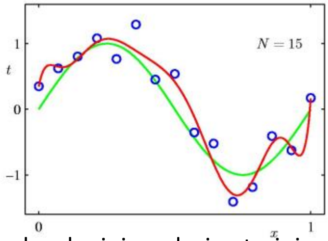
- **Introduction**
- **Q-Learning (lab5)**

Next lecture

- Deep learning, a closer look

Machine Learning Pitfall - Overfitting

- Models can overfit if you have **too many parameters in relation to the training set size**.
- Example: 9th degree polynomial regression model (10 parameters) on 15 data points:



Green: True function (unknown)
Blue: Training examples (noisy!)
Red: Trained model

(Bishop, 2006)

- This is **not** a local minima during training, it **is** the best fit possible on the given training examples!
- The trained model captured ”**noise**” in data, variations independent of f(**x**)

Overfitting – Where Does the Noise Come From?

- Noise is small variations in the data due to **ignored** or **unknown variables,** that cannot be predicted via chosen feature vector **x**
  - Example: Predict the temperature based on season and time-of-day. What about atmospheric changes like a cold front? As they are not included in the model, nor entirely captured by other input features, this variation will show up as **random noise** for the model!

- With few data points to go on, the model can mistake the effect that unmodeled variables has on y as coming from variables **x** that **are** included.

- Since this relationship is merely chance, the model will **not generalize** well to future situations

Overfitting - Demo

- See the interactive example of ANN training again http://playground.tensorflow.org/
  - 2D input x -> 1D y (binary classification or regression)

Exercise:

- Pick the bottom-left data set, two (Gaussian) clusters
- Make a flexible network, e.g. 2 hidden layers w/ 8 neurons each
- Set ”Ratio of training to test data” to 10%
- Max out noise
- Train for a while, can adjust ”learning rate”
- Compare result to ”Show test data”
- How well does this model generalize?

Up next: How do we fix it?

Model Selection – Choosing Between Models

- In conclusion, we want to avoid **unnecessarily complex** models
- This is a fairly general concept throughout science and is often referred to as **Ockham's Razor**:

  "Pluralitas non est ponenda sine necessitate"

  -Willian of Ockham

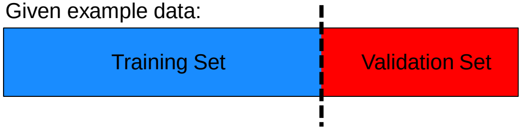
  "Everything should be kept as simple as possible, but no simpler."

  -Albert Einstein (paraphrased)

- There are several mathematically principled ways to **penalize** model **complexity** during training, which we will not cover here.
- A simple approach is to use a separate **validation set** with examples that is **only** used for evaluating models of different complexity.

Model Selection – Hold-out Validation

- This is called a **hold-out validation set** as we keep the data away from the training phase
- Measuring performance on such a validation set is a **better metric of actual generalization** error to unseen examples
- With the validation set we can compare models of **different complexity** to select the one which generalizes best.
- Examples could be polynomial models of different order, the number of neurons or layers in an ANN etc.



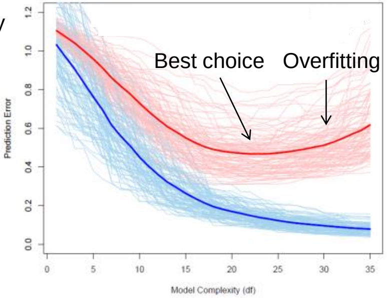
Model Selection – Selection Strategy

- As the number of parameters increases, the size of the hypothesis space also increases, allowing a **better fit to training data**
- However, at some point it is **sufficiently flexible** to capture the underlying patterns. Any more will just capture noise, leading to **worse generalization to new examples!**

Example: Prediction error vs. model complexity over many (simulated) data sets. (Hastie et al., 2009)

Red: Validation set (generalization) error

Blue: Training set error



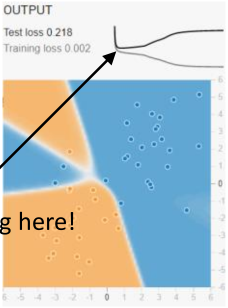
- Do we need to train and test many models of different complexity?
  - Various tricks to avoid this

Early Stopping: Model Complexity Trick with Neural Networks

- Training neural networks tends to progress from simple functios to more complex ones
- This comes from initializing the parameter values **w** close to zero
  - Remember, a neuron's output = g(**w***x)
  - Common activation functions g (e.g. sigmoid) are linear around zero
  - This makes the NN effectively "start out" as a linear model
- **Early stopping** NN trick: Can make a model complexity vs. validation loss curve **while training**, stop when validation error starts increasing

Exercise: Back to the NN demo app

- Observe "test loss" plot
- Reset network
- Train again, but keep an eye on test loss
- Try to pause at low test loss
  - Can adjust "learning rate"



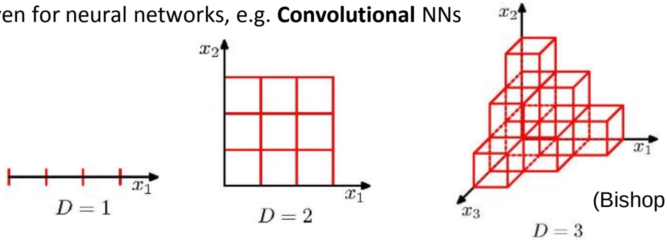
Stop training here!

Limitations of Supervised Learning

- We noted earlier that the first phase of learning is traditionally to select the ”**features**” to use as input vector **x** to the algorithm
- In the spam classification example we restricted ourselves to **a set of relevant words** (bag-of-words), but even that could be thousands
- Even for such binary features we would have needed $O(2^{\#features})$ examples to cover all **possible** combinations
- In a continuous feature space, there might be a difficult non-linear case where we need a grid with 10 examples along each feature dimension, which would require $O(10^{\#features})$ examples.

The Curse of Dimensionality

- This is known as the **curse of dimensionality** and also applies to reinforcement learning as we shall see later
- However, this is a **worst case** scenario.
  - **The true amount of data needed for supervised learning depends on the model and the complexity of the function we are trying to learn**
  - **Deep learning** *may* overcome this since it can capture hierarchical abstractions
- Usually, learning works rather well even for many features
  - However, selecting features and model that reflects problem structure **can** be the difference between success and failure
  - Even for neural networks, e.g. **Convolutional** NNs



(Bishop, 2006)

Some Application Examples of Dimensionality

Computer Vision – Object Recognition

- One HD image can be 1920x1080 = 2 **million** pixels
- If **each pixel** is naively treated as one dimension, learning to classify images (or objects in them) can be a **million-dimensional problem**.
- Much of computer vision involves clever ways to extract a small set of descriptive features from images (edges, contrasts)
  - Recently **deep convolutional networks** dominate most benchmarks

Data Mining – Product models, shopping patterns etc

- Can be anything from a few key features to millions
- Can often get away with using **linear models**, for the very high-dimensional cases there are few easy alternatives, although **NNs gaining popularity**

Some Application Examples of Dimensionality II

Robotics

- For perception, see the computer vision considerations, but need **real-time**
- For control, e.g. learning robot motion
  - Moderate dimension, but **non-linear** and require **high accuracy (robustness)**
  - Ground robots have at least a handful dimensions
  - Air vehicles (UAVs) have at least a dozen dimensions
  - Humanoid robots have at least 30-60 dimensions
  - The human body is said to have over 600 *muscles*
- Traditionally use tailored models based on e.g. physics approximations
  - Learning is gaining ground but **data is not as cheap to collect**

Applications in Physical Agents: Robotics

Humorous reminder from IEEE Spectrum: The DARPA 2015 Humanoid Challenge “Fail Compilation”

- **Engineering** robot behavior can be **fragile** and **time consuming**
- Things humans do without thinking require **extremely detailed instructions** for a robot. Even robust locomotion is hard.
- Can we **learn** robot behaviors?

LINKÖPINGS UNIVERSITET

Learning for Robot Behavior?

- Powerful models such as deep neural networks can require millions of examples
- As seen in pancake flipping video, **supervised learning** with robots quickly gets tedious as you need to baby-sit it in real-time
- Can we remove the human from the loop?

  1. An **automated teacher** like a **planning or optimal control** algorithm can generate supervised examples **if it has a model of the world**
     - Mordatch et al, https://www.youtube.com/watch?v=IxrnT0JOs4o
     - Our research w/ real nano-quadcopters (deep ANN on-board the microcontroller)
  2. Reinforcement learning generalizes this to completely **unknown worlds**

A Gentle Introduction to Machine Learning

Part II - Reinforcement Learning

Olov Andersson

Artificial Intelligence and Integrated Computer Systems
Department of Computer and Information Science
Linköping University

Introduction to Reinforcement Learning

- Remember:
  - In Supervised Learning agents learn to act given **examples** of correct choices.
- What if an agent is given **rewards** instead?
- Examples:
  - In a game of chess, the agent may be rewarded when it wins.
  - A soccer playing agent may be rewarded when it scores a goal.
  - A helicopter acrobatics agent may be rewarded if it performs a loop.
  - A pet agent may be given a reward if it fetches its masters slippers.
- These are all examples of **Reinforcement Learning**, where the agent itself figures out *how* to solve the task.

Defining the domain

- How do we formally define this problem?
- An agent is given a sensory **input** consisting of:

State	$s \in \mathcal{S}$ (from type problem domain)
Reward	$R(s) \in \mathbb{R}$ (our way to encode objective *in* domain)

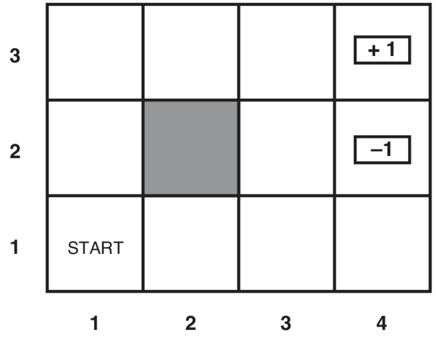
- It should pick an **output**

Action	$a \in \mathcal{A}$ (based on type of robot/agent)

- It wants to learn the "best" action for each state.

What do we need to solve?

- An example domain...
- $\mathcal{S}$ = {squares}
- $\mathcal{A}$ = {N,W,S,E}
- $R(s)$ = 0 except for the two terminal states on the right



- Considerations:
  - It may not know the effect of actions yet $p(s'|s,a)$
  - It may not know the rewards $R(s)$ in all states yet
  - Reward will be zero for all actions in all states not adjacent to the two terminal states.
  - **Need to consider reward of future moves!**

Rewards and Utility

- We define the reward for reaching a state s_i as $R(s_i)$
- To **plan ahead** it must look at a sum of rewards over a **sequence** of states $R(s_{i+1}), R(s_{i+2}), R(s_{i+2}), \ldots$
- This can be formalized as the **utility** U for the sequence

$$U = \sum_{t=0}^{\infty} \gamma^t R(s_t), \text{ where } 0 < \gamma < 1 \tag{1}$$

- Where $\gamma < 1$ is the **discount factor** making the utility finite even for infinite sequences.
- A low γ makes the agent very short-sighted and greedy, while a gamma close to one makes it very patient ($\gamma \approx$ planning horizon).

The Policy Function

- We now have a utility function for a sequence of states
- ...but the sequence of states depends on the actions taken!
- We need one last concept, a **policy** function $\pi(s)$ decides which action to take in **each** state

$$a = \pi(s) \tag{2}$$

- Clearly, a good policy function is what we set out to find

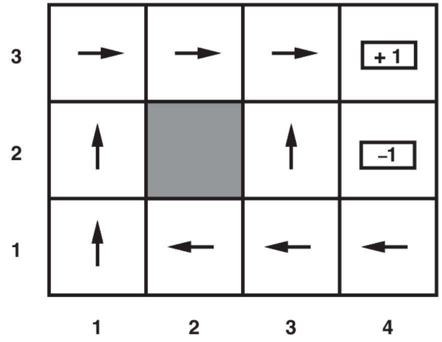
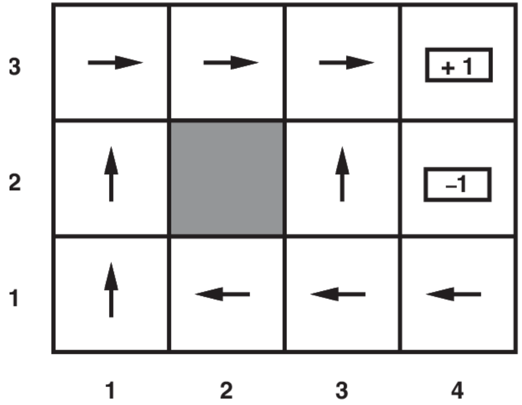
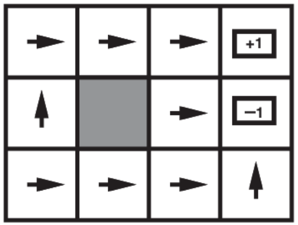


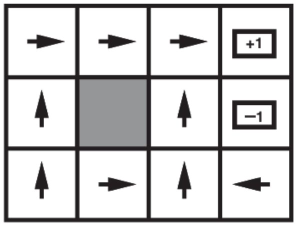
Figure: A policy function maps states to actions (arrows)

Examples of optimal policies for different R(s)

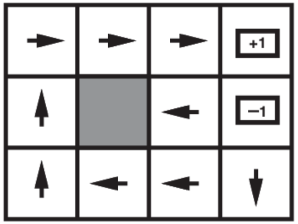




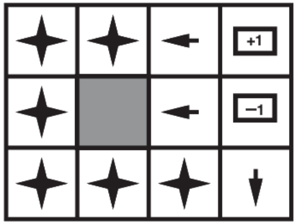
$R(s) < -1.6284$



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

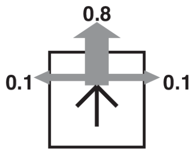


$-0.0221 < R(s) < 0$



$R(s) > 0$

Assuming random transition function (for each direction):



How to find such an optimal policy?

- There are two different philosophies for solving these problems
- Model-based reinforcement learning
  - Learn $R(s)$ and $f(s, a) = s'$ using supervised learning.
  - Solve a (probabilistic) planning problem using an algorithm like value iteration (see book, not included in this course).
- Model-free reinforcement learning
  - Use an iterative algorithm that *implicitly* both adapts to the environment and solves the planning problem.
  - Q-learning is a popular such algorithm that has a very simple implementation. (lab5)

Q-Learning

- In Q-learning, all we need to keep track of is the "Q-table" $Q(s,a)$, a table of **estimated utilities** for taking action a in state s.
- Each time an agent moves, the Q-values can be updated by nudging them towards the *observed* reward and the Q-value of the *observed* next state.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{\alpha' \in \mathcal{A}} Q(s',a') - Q(s,a)) \qquad (3)$$

  where α is the **learning rate** and γ the discount factor.
- This comes from the recursive definition of utility and samples the transition function $p(s'|s,a)$ so we don't have to learn it. This means that the Q-function can **only** be updated when the agent actually **interacts** with the environment.
- If the transitions are **deterministic**, $\alpha = 1$ is optimal, but approximating continuous state by discrete ones can make actions appear non-deterministic (lab5).

Q-Learning

- In Q-learning, all we need to keep track of is the "Q-table" $Q(s,a)$, a table of **estimated utilities** for taking action a in state s.
- *If* we knew the long-term value of an action, solving the planning problem to compute policy $\pi(s)$ reduces to just taking the best action in the Q-table: $\max_{\alpha \in \mathcal{A}} Q(s,a)$
- Turns out one can learn the Q-table for the optimal policy by applying an iterative update rule on the Q-table as the agent moves
- In a simpler **deterministic** world (no randomness) this is:

$$Q(s,a) \leftarrow Q(s,a) + R(s) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \tag{4}$$

  where γ is the discount factor.
- An intuition is to remember that Q-value is estimated utility. We are using a recurrence relation from the definition of utility in Eq.(1), *updating the best estimate in the current state with the observed reward, and the best estimate in the next state*.

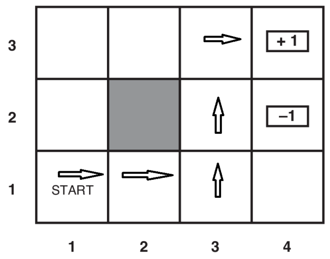
Q-Learning II - Final Version

- The full update rule for non-deterministic environments is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)) \tag{5}$$

  where α is the **learning rate** and γ is the discount factor.
- Each time an agent moves, the Q-values can be updated by **nudging** them towards the Q-value targets, the α weighting acting as a kind of moving average over all (random) next states for an action.
- Even just approximations of the state space, like the discretization in lab5, can cause non-determinism from the point of view of the agent.
- For full proof, see the book (not needed for exam).

The Q-table Update - An Example



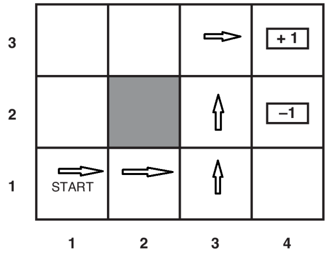
Where actions are N,E,S,W (North = up) and $\lambda = 0.9$. For simplicity the agent *repeatedly* executes the actions above, ending each episode in the terminal +1 state and restarting. Transitions are deterministic so we use learning rate $\alpha = 1$.

Begin by initializing all terminal $Q(s_T, *) =$ reward, all other $Q(s, a) = 0$
For each step the agent updates Q(s,a) for the *previous* state/action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a))$$

After a while the Q-values will converge to the true utility

The Q-Learning Update - An Example



$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a))$$

First run (clarified): $Q(s_{3,3},E) = 0 + 1 \cdot (0 + 0.9 \cdot \max(1,1,1,1) - 0) = 0.9$.
(Remember, all action Q-vals for terminal $s_{4,4}$ initialized to +1)
Second run: $Q(s_{3,2},N) = 0 + 1 \cdot (0 + 0.9 \max(0,0.9,0,0) - 0) = 0.81$,
$Q(s_{3,3},E) = 0.9$ (unchanged due to learning rate $\alpha = 1$)
Third run: $Q(s_{3,1},N) = 0 + 1 \cdot (0 + 0.9 \max(0.81,0,0,0) - 0) = 0.729$,
$Q(s_{3,2},N) = 0.81$, $Q(s_{3,3},E) = 0.9$ (both unchanged). And so on...

Action selection while learning: Exploration

- That was assuming **fixed** actions. The agent should ideally pick the action with **highest utility** (Q-value).
- However, always taking the highest estimated utility action while still learning will get the agent stuck in a sub-optimal policy.
- In the previous example, once the Q-table has been updated all the way to the start position, following that path will always be the only non-zero (and therefore best) choice.
- The agent needs to balance taking the *currently* highest Q-value actions with exploring the other options!
- Simple ϵ-greedy strategies do random movements $\epsilon\%$ of the time.

Curse of Dimensionality for Q-Learning

- Need to discretize continuous state and action spaces.
- The Q-table will grow exponentially with their dimension!
- Workaround: Approximate Q-table by supervised learning.
    - "Fitted" Q-iteration. See Q-table as unknown f(x), (state,action) as examples of input x, and the Q-value *after* update as example output y. Can learn this from new examples as the agent moves.
- If approximation **generalizes** well, we get large gains in scalability.
- Use deep learning $\rightarrow$ **deep reinforcement learning**
    - Deep ANN was used for the video game example (plus some tricks)
    - Google's Go champion combines several approaches, deep convolutional nets for approximating the game board, a tree-search planning approach for updating utilities and more...
- Caveat: Non-linear approximations may impede convergence.

Q-Learning - Final Words

- Implementation is very simple, having no model of the environment.
    - It only needs a table of Q(s,a) values!
- Once the Q(s,a) function has converged, the optimal policy $\pi^*(s)$ is simply the action with highest utility in the table for each s
- Technically the learning rate α actually needs to decrease over time for perfect convergence.
- Q-learning must also be combined with exploration
- Q-learning requires very little computational overhead per step
- The curse of dimensionality: The Q-table grows exponentially with dimension. A good approximation can avoid this.
- Model-free methods may require more interactions with the world than model-based, and much more than a human.