Nature Inspired Algorithms (1): Neural Networks

CE811 Game Artificial Intelligence

Dr Michael Fairbank

18th October 2021

University of Essex

Overview

Introduction

- Last week we talked about *Ad-hoc behaviour authoring*.
- Along with Pathfinding and Obstacle avoidance (Lecture 4) these form the basis of most game AI
- However we could use more modern, adaptive and efficient techniques in our games.

Introduction

There are two main types of algorithms to consider for automatic game playing:

1. Algorithms that **learn a model** for this particular game. The model tells the agent what to do in each situation.
    - (E.g. A neural network that learns ”jump whenever you get here”)
2. Algorithms that do not need to learn anything about the game, but do need to be given a ”forward model” of the game, which allows them to do a **look-ahead of future possible actions**, and have the ability to choose the best method.
    - (E.g. MCTS, A*, Minimax)

Key Concepts

There are at least two concepts which are common to all of the algorithms used:

Representation How to store, maintain and represent the knowledge gathered about a particular task.

1. How do we represent the algorithm?
2. How do we represent the information the algorithm acts on?

Utility Evaluation of a game state, representing how good it is. This could be used, e.g. to determine which path to take, based on the knowledge about the task at stake

Representation of Knowledge

- **Knowledge Representation** is one of the most crucial and difficult aspects of AI.
  - We already talked about blackboards, and how difficult it is to keep them well organised.
- Knowledge (both algorithms and data) must stored in a way that can be accessed and processed by the machine
- The better the form we store them in, the easier it is to do AI
- The representation selected for a problem suffers from the no free lunch theorem:[1] there is no single representation that is ideal for all possible tasks.

[1] Wolpert + Macready, 1997: No free lunch theorems for optimization.

Representation Examples

Some different representations of algorithms include:

Graphs	Finite State Machines, Probabilistic models
Connectionism	Artificial Neural Networks
Trees	Decision Trees, Behaviour Trees, Genetic programming
Genetic	Evolutionary Algorithms
Tabular	TD-Learning and Q-Learning
Grammars	Grammatical Evolution

Utility

- In game theory, **Utility** is the measure of rational choice when playing a game.
- In a basic sense - how **good** a state (or region of the state space) is.
- This information about the ‘goodness’ of a state can be used to inform decisions or moves to play
- Again, no free lunches here either

Utility Examples

Some variations on utility include:

Objective/Loss/Cost/Error	In Machine Learning, it represents how well the approach maps to the training examples
Fitness	In evolutionary computation measures the degree to which a solution fits the goals.
Heuristic	In tree search, A function that approximates the value of the state by analysing it’s features.
Reward	In reinforcement learning, function that the agent tries to maximise when learning with actions to execute a given state

Outline

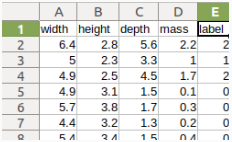
Supervised learning

- Supervised Learning is the process of trying to *approximate* an underlying function between some *labelled data* and their corresponding attributes or features.
- The goal is not to determine a relationship between the properties and labels but derive a function which approximates it.
- To evaluate its effectiveness, the algorithm is tested on previously **unseen examples** (test examples)

Supervised learning

So supervised leaning consists of:

- A list of input vectors $\{x_1, x_2, \ldots, x_n\}$
- and corresponding target output vectors (or labels): $\{y_1, y_2, \ldots, y_n\}$
- Note that each pair (x_i, y_i) is called a training **pattern**.



	A	B	C	D	E
1	width	height	depth	mass	label
2	6.4	2.8	5.6	2.2	2
3	5	2.3	3.3	1	1
4	4.9	2.5	4.5	1.7	2
5	4.9	3.1	1.5	0.1	0
6	5.7	3.8	1.7	0.3	0
7	4.4	3.2	1.3	0.2	0
8	5.4	3.4	1.5	0.4	0

- Q: In this data set, what is x_2?
- Q: In this data set, what is y_2?

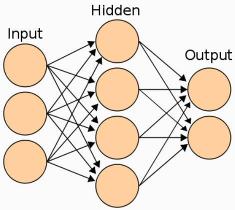
Supervised learning

- E.g. might be trying to distinguish between different fruit (apple vs. banana vs orange) given some of their measurements and attributes (e.g. colour, maximum diameter, weight).
- The training data consists of lots of examples of the measurements, along side the label (apple / banana / orange) for each set of attributes.
- Target values (labels) are necessary for supervised learning.

Outline

Artificial Neural Networks

- An ANN is a function that maps a set of inputs to a set of outputs
- These are represented as vectors
- The network always contains input and output layers
- There may also be hidden layers between them
- A layer contains nodes
- Nodes in one layer are connected to nodes in the next layer, by connections called weights.
- Usually there are no backward-facing connections, so we have a “feed-forward neural network” (FFNN).



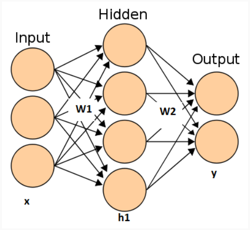
Artificial Neural Networks

As shown on the diagram:

- Let W_1 be the matrix of weights in the first layer of connections.
- Let W_2 be the matrix of weights in the second layer of connections.
- Let the input vector be x
- The hidden layer has activations h_1
- The output layer has activations y

Plus (but not shown on diagram):

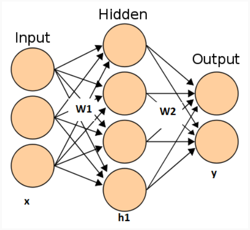
- Let b_1 be a vector of bias values for each node in the hidden layer.
- Let b_2 be a vector of bias values for each node in the output layer.



Artificial Neural Networks

- Then the hidden layer's activations are set by: $h_1 = \mathtt{tanh}(xW_1 + b_1)$
- And the output layer's activations are set by: $y = h_1 W_2 + b_2$

where ordinary Matrix Multiplication is being used here.



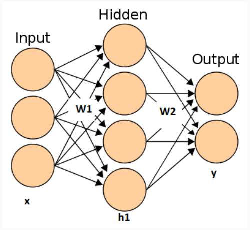
Artificial Neural Networks

For these matrix products to make sense (for this particular network shown on the right):

- We need x to be a row of length 3
- W_1 has 3 rows and 4 columns
- h_1 and b_1 have length 4
- W_2 has 4 rows and 2 columns
- h_2 and b_2 have length 2

Also,

- We assume the *tanh* is applied "element-wise" here.
- So the ith element of $\mathrm{tanh}(x)$ is equal to $\mathrm{tanh}(x_i)$



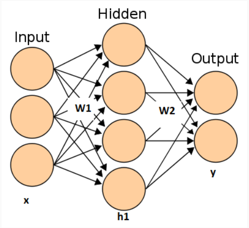
Artificial Neural Networks

If more hidden layers were present ("deep learning") then we could have:

- $h_1 = \tanh(xW_1 + b_1)$
- $h_2 = \tanh(h_1 W_2 + b_2)$
- $h_3 = \tanh(h_2 W_3 + b_3)$
- $\ldots$
- $y = h_{n-1} W_n + b_n$

tanh is called the "activation function" or "non-linearity"

- We don't normally have an activation function on the final layer.
- This lets the output range of the neural network exceed $-1 < y < 1$.



Tensorflow FFNN

To Build a feed-forward neural network (FFNN) in TensorFlow:

```
import tensorflow as tf
# Build our random weight and bias matrices, of appropriate shapes
W1 = tf.Variable(tf.random.truncated_normal([3,4], stddev=0.1))
b1 = tf.Variable(tf.random.truncated_normal([1,4], stddev=0.1))
W2 = tf.Variable(tf.random.truncated_normal([4,2], stddev=0.1))
b2 = tf.Variable(tf.random.truncated_normal([1,2], stddev=0.1))

def run_network(x):
    h1=tf.tanh(tf.matmul(x,W1) + b1)
    y=tf.matmul(h1,W2) + b2
    return y
```

Keras FFNN

To Build the same feed-forward neural network (FFNN) in Keras:

```
from tensorflow import keras
layer1=keras.layers.Dense(4, activation='tanh', input_shape=(3,))
layer2=keras.layers.Dense(2, activation=None)
model = keras.Sequential([layer1,layer2])
def run_network(x):
    return model(x)
```

Artificial Neural Networks

- So we’ve seen how the ANN receives an input vector x, and produces an output vector y.
- They won’t do anything useful as we’ve described them yet.
- We first need to “train” the neural network, i.e. choose the values of the weight and bias matrices such that the neural network does something useful.
- Give it enough weights, each appropriately set, and the neural network can represent any smooth function, to arbitrary accuracy.
    - Universal function approximator

Universal function approximator

E.g. could be a vision system:

- Input = grid of pixel intensities of image
- Output = classification of what the image is (car, house, etc)

E.g. Could be a language translation system:

- Input = vector of words in an English sentence
- Output = vector of words in an French sentence

E.g. It could be a game player:

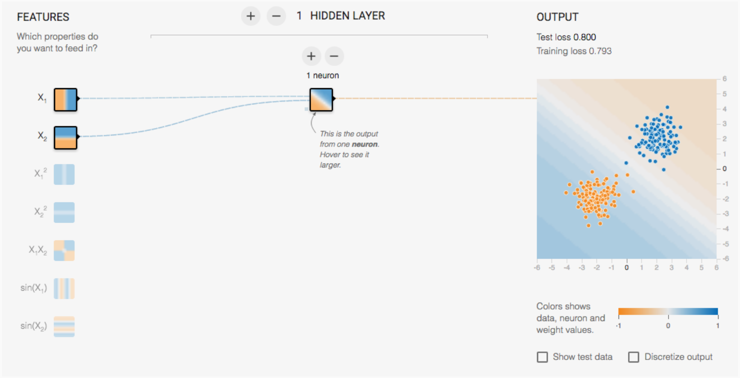
- Input = The state of the game
- Output = Action to take

Q1: What would the state be in a driving game?

Q2: What would the state be in a poker game?

Tensorflow Playground

`http://playground.tensorflow.org`



We'll play with this in one of the moodle quizzes.

Outline

Neural-Network Training

In supervised learning we have

- A list of input vectors $\{x_1, x_2, ..., x_n\}$
- and corresponding target output vectors $\{y_1, y_2, ..., y_n\}$

We want the NN to choose weights and biases so that these match up,

- so that $f(x_p) \approx y_p$,
    - for all training pattern pairs (x_p, y_p),
        - where $f(...)$ is our neural network function.

Loss Function

Define an objective “loss function” that we seek to minimise:

$$L = \frac{1}{n}\sum_{p=1}^{n}(y_p - f(x_p))^2 \tag{1}$$

where $f(x)$ is the neural network function.

- We seek to minimise L with respect to $W_1, b_1, W_2, b_2, ...$
- The loss function is variously known as an “error function”, ”loss function”, ”objective function” or ”cost function”.
- This particular loss function is called the “mean-squared error”. The other common loss function is the “cross-entropy loss”

Neural-Network Training

There is no closed-form solution to train a NN in one step.

- We have to use an iterative method.

Define $\vec{w} = \mathit{flatten}([W_1, b_1, W_2, b_2, ...])$

- Then we can update all of the weights and biases in the network, *iteratively*, by

$$\Delta\vec{w}^i = -\eta\frac{\partial L}{\partial\vec{w}^i} \qquad \text{(for all components } i\text{)} \qquad (2)$$

- Here $\eta > 0$ is a small “learning rate” (η=“eta”)
- This is “gradient descent” on L with respect to $\vec{w}$

Gradient Descent

- We do gradient descent on the loss-function in “weight space”
- If weight-space was 2D then this is what gradient descent might look like.

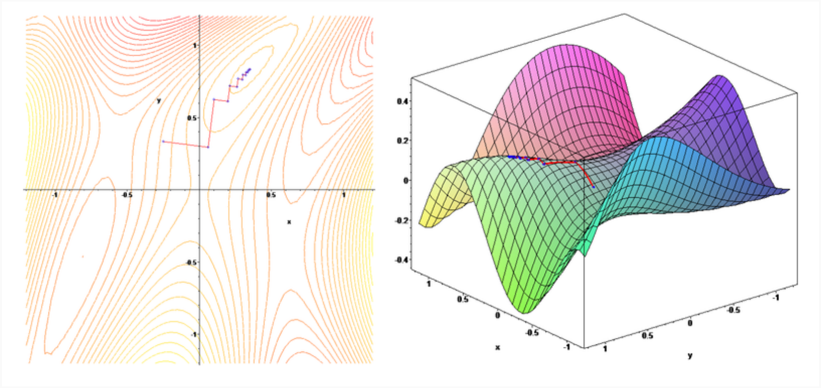
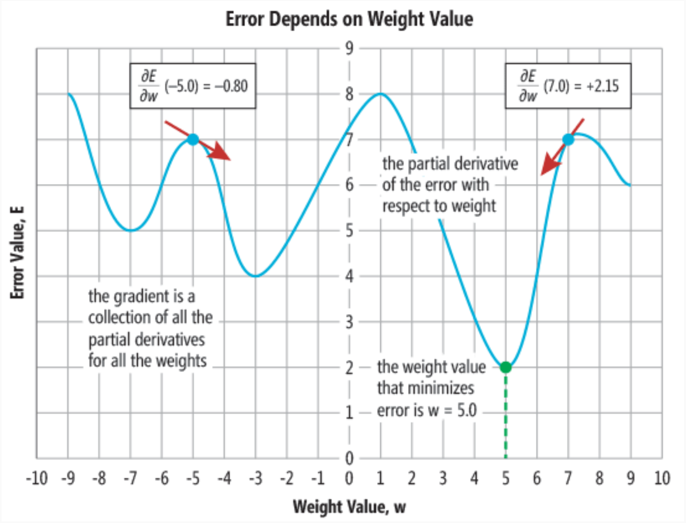


Image source: `https://en.wikipedia.org/wiki/Gradient_descent`Gradient-descent

Partial Derivatives and Gradient Decent

Consider a 1D simplification of weight space:

Here the loss function is “E”.



Backpropagation / Automatic Differentiation

- We want to do $\Delta\vec{w}^i = -\eta\frac{\partial L}{\partial \vec{w}^i}$
- How do we compute this derivative?
- Reminder $L = \frac{1}{n}\sum_{p=1}^{n}(y_p - f(x_p))^2$, and $f(x) = ((\tanh(xW_1 + b_1))W_2 + b_2)$.
- Could you work out how to differentiate this?

Backpropagation / Automatic Differentiation

- The “backpropagation” algorithm[2][3] computes $\frac{\partial L}{\partial \vec{w}^i}$ for all components *i* very efficiently.
- It is an application of “automatic differentiation”.
- It is often combined with gradient descent[4]

[2]Backpropagation was described by Werbos, P. J. ”Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph. D. thesis, Harvard University, Cambridge, MA, 1974.” (1974).

[3]Most notably popularised for neural networks by Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. ”Learning representations by back-propagating errors.” nature 323.6088 (1986): 533-536.

[4]Many people incorrectly call an application of gradient descent as an ”applicaton of backpropagation”, however I take ”backpropagation” to be the computation of $\frac{\partial L}{\partial \vec{w}^i}$ and the gradient-descent step to be a separate operation.

Automatic Differentiation

- Automatic differentiation (AD) is a set of algorithms that allow us to compute the derivatives of any computation that can be written as a “graph” of mathematical operation, such as a neural network (which is a graph which links together lots of multiplication, additions, tanh functions), or even the loss function *L*.
- All of those component operations themselves have simple known derivatives.

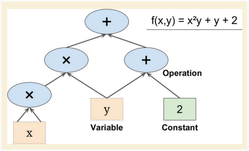
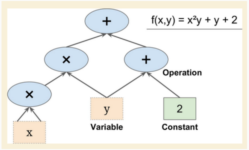


Image source: Hands-On Machine Learning with Scikit-Learn and TensorFlow

Automatic Differentiation

- So if we were clever enough, we could methodically combine all of these derivatives (using the chain rule, product rule, etc)
  - to compute the derivative of the function *L* with respect to any of its arguments.
- This is what automatic differentiation does.
- Modern libraries do AD for us (TensorFlow, Keras, PyTorch).



Automatic Differentiation with TensorFlow

Suppose we have a function defined as follows:

$$f(x,y) = 3x^2 + y$$

As we know from basic calculus, its partial derivatives are:

$$\Rightarrow \partial f / \partial x = 6x,$$
$$\partial f / \partial y = 1$$

We can compute these derivatives efficiently using AD in tensorflow:

```
> import tensorflow as tf
> x=tf.constant(4.0,tf.float32)
> y=tf.constant(2.0,tf.float32)
> with tf.GradientTape() as tape:
>     tape.watch([x,y]) # Tells the gradient tape to prepare to find derivatives with respect to x and y
>     f=tf.pow(x,2.0)*3.0+y
> [dfdx, dfdy]=tape.gradient(f, [x,y])
> print(dfdx.numpy(), dfdy.numpy())
24.0 1.0
```

Automatic Differentiation with TensorFlow

- It also works if *x* and *y* are vectors or matrices.
- It will do it component-wise in those cases.
- So that's what we need to compute $\frac{\partial L}{\partial \vec{w}^i}$.
- Hence backpropagation with gradient descent is very easy to implement in TensorFlow
- All that calculus work is done for you.

Trianing the Network in Keras

- Keras extends tensorflow - uses AD within TensorFlow
- To iterate Gradient-Descent in Keras, we use:

```
model.compile(
     optimizer=keras.optimizers.SGD(0.01),
     loss=keras.losses.MeanSquaredError()
)
history = model.fit(
    x_train,
    y_train,
    batch_size=len(x_train),
    epochs=1000
)
```

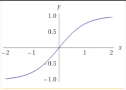
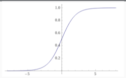
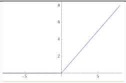
- This will run 1000 iterations of Gradient Descent (“SGD”), with learning rate $\eta = 0.01$
- The loss function is ”Mean-squared error”
- “model” is our NN we built before in keras.
- The input data is stored in a numpy array x_train
- The target output data is stored in a numpy array y_train

Outline

Subtleties: Activation Functions

We can use different activation functions, other than tanh.

The most common ones are:

Activation Function	Equation	TensorFlow Function	Graph	Range
Hyperbolic Tangent Function	$y = tanh(x)$	tf.tanh		$-1 < y < 1$
Logistic Sigmoid Function	$y = \frac{1}{1+e^{-x}}$	tf.nn.sigmoid		$0 < y < 1$
Rectified Linear Function	$y = max(x, 0)$	tf.relu		$0 \leq y < \infty$

Subtleties: Optimizer

- Gradient descent uses a fixed step size η.
- This can get very slow on a flattish part of of the loss-function surface.
- We can use different “optimizers”, some which attempt to speed up and slow down on flat / curvy parts of the surface.
- The main two ones are:
  - keras.optimizers.SGD: Ordinary fixed step-size “stochastic Gradient Descent”
  - keras.optimizers.Adam: Dynamically and intelligently adjusts the learning rate to handle tight corners and plateaus.[5]

[5] See nice blog article on Adam

Subtleties: Loss Function

- If the target values y are real values, then:
  1. You are doing a “regression” task.
  2. Use $L =$”mean-squared-error”.
  3. Do not use an activation function on your final layer.
- If the target labels y are categories (e.g. dog vs cat), then:
  1. You are doing a “classification” task.
  2. Use $L =$”cross-entropy loss”.
  3. Use softmax activation function on your final layer (implicitly or explicitly).

Classification Tasks

Q: Why classification networks need a different loss function and softmax on final layer, compared to regression networks?

A: The combination of cross-entropy with softmax ensures the neural network outputs *probabilities*.

Softmax and Cross-Entropy are described on the next two slides...

Classification Tasks: Softmax

Softmax Function:

- Suppose we have 3 output categories (0/1/2). Our neural network needs 3 outputs, y_0,y_1,y_2.
- The softmax function on the final layer converts these outputs to

$$p_i = \frac{e^{y_i}}{\sum_k e^{y_k}} \tag{3}$$

- The Softmax function on the final layer ensures the outputs are all positive and add to 1 (like probabilities should behave).
- In keras we can just specify the final layer with: *tf.keras.layers.Dense(num_classes, activation='softmax')*

Classification Tasks: Cross Entropy

The Cross-Entropy Loss function:

- The cross entropy loss function, for one output vector $(p_0, p_1, p_2, ..., p_{n-1})$ with target category $t \in \{0, 1, 2, ..., n-1\}$, is
$$L = -log(p_t) \tag{4}$$
- Hence minimising L must maximise p_t, i.e. the probability of the true category t, as estimated by the neural network.
- This is summed over the entire training batch.
- In keras, we can specify Cross-Entropy Loss in the *model.compile* line with: *loss=keras.losses.SparseCategoricalCrossentropy()*[6].

[6]Technical Note: It's most numerically stable to pass this function the argument *from_logits=True*, which will then do the softmax on the final layer for you (and you should therefore use no activation function on your final layer)

Subtleties: Input Feature Preparation

Real valued inputs:

- Generally the inputs to a NN should all be rescaled to be in roughly the range [-1,1]
- Commonly subtract the mean and divide by the standard deviation to achieve this.

Categorical inputs:

- Generally one-hot encode categorical inputs.
- E.g. if there are 3 possible input categories for car manufacturer, ”BMW”, ”Ford” or ”Fiat”, then encode them as:
    1. BMW=(1,0,0)
    2. Ford=(0,1,0)
    3. Fiat=(0,0,1)

Note: Most of the skill in making neural networks work is preparing your input / output data appropriately.

Subtleties: Mini-batching

- If your training-set size has 1bn items in it, then just computing *L* over the whole training set will take too long.
- We must train on “mini-batches”.
- Every iteration, pick 20 or so patterns randomly from the 1bn, and train on those.
- That is what the variable “batch_size” does in the keras fit loop. It is the mini-batch size.
- Terminology:
    1. One **epoch** is 1 full pass through the dataset, which might consist of many iterations.
    2. One **iteration** is one application of the weight update equation, calclulated for one mini-batch.

Subtleties: Mini-batching

- Shuffling mini-batches like this gives a stochastic element into training neural networks.
- Hence the name “Stochastic Gradient Descent” (SGD)
- Can shake gradient descent out of local minima
- And improve generalisation
- Hence SGD can be the best learning algorithm (despite being slow)

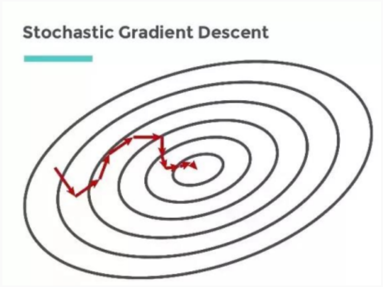
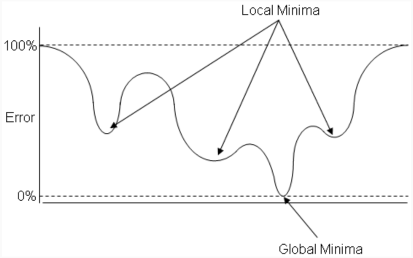


Image source: https://qph.ec.quoracdn.net/mainqimgd76baf475801ee1a0b850b11f932e719

Subtleties: Local Minima



- It’s worth noting that gradient descent is not guaranteed to find the global minimum of E, given its local search property.
- Besides, the ANN can get stuck in **local minima**

Subtleties: Local Minima

Typical approaches to overcome convergence to local minima include:

- Random restarts: rerun the algorithm with new random values for w_{ij}.
- Mini batching: Can shake the solution out of a local minimum
- High dimensional weight space (lots of weights): Allows for tunnels to exist in high dimensions which can lead out of minima
- Suitable optimizer (E.g. adam/sgd)

Subtleties: Initialising weights

You need to randomise your weights before training.

- Why?
    1. If all the weights are zero, then due to symmetry, it's likely that you are starting gradient descent from the exact top of a hill.
    2. To avoid local minima we might have to attempt learning from several start points in weight space.
- Keras does this for us.
- Research has proven that the initial choice of magnitude for the random weights is very important.[7]

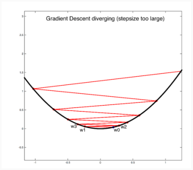
[7] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

Subtleties: Learning Rates

Getting the learning rate right can be tricky.

Observations:

- If eta too high, it fails
- If eta too low, solution is very slow.



Recommendation:

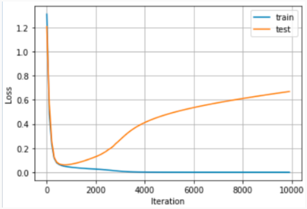
- Plot the value of the function you are trying to minimise vs. iteration number
- Make sure it is decreasing
- Reduce η if necessary
- Modern optimisers (Adam) help by adapting the learning rates for you.

8

[8] Image source: http://www.cs.cornell.edu/courses/cs4780/2017sp/lectures/images/gradient_de

Subtleties: Overfitting

- Another common problem of ANN is **overfitting**: it refers to a model that learns the *training* data too well ($L \approx 0$) but performs *poorly* on test data.
- We normally split the data into a training set, validation set and possibly a test set too.
- We plot the performance on both the training set and validation set



- Here we see overfitting has happened badly

Subtleties: Overfitting

Overfitting typically happens when:

- Not enough training data
- Powerful model (lots of weights) can simply memorise the training data.

We can try to address overfitting by any of:

- Get more training data
- Stop training when the orange curve starts increasing (“early stopping”)
- Add some regularization, e.g. L2 Regularisation.
- Use Dropout layers
- Train 50 neural networks and use the average of their outputs (create an ensemble)

Subtleties: Test set versus validation set

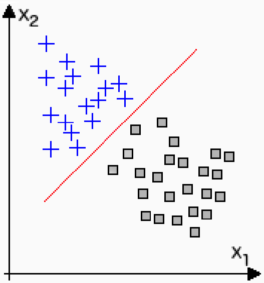
- During training we peep at the validation set, e.g. by graphing it, and using it to tune learning rates and other hyper-parameters.
- Hyper parameters are things such as learning rates, network size, number of hidden layers, number of hidden nodes.
- We should keep a third batch of data separate that we have never peeped at, this is called the test set.

Subtleties: K-fold Cross-Validation

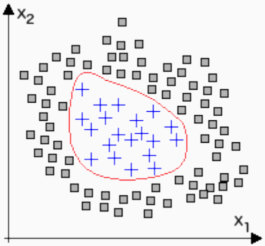
- What if we don’t have enough data to separate out a test set?
- We can try k-fold cross validation.
- E.g. for 10-fold cross validation, we split the data set into 10 portions.
    - We keep one out, this is the current test set.
    - We train on the other 90% of the data
    - We repeat the process 10 times, with a different portion held-out as test-set each time.
    - We quote the mean of our 10 test-set scores, as our final test-set performance.

Subtleties: Class Separability

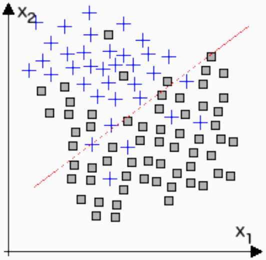
Input: x_1, x_2; output: 2 classes.



Linear



Non Linear

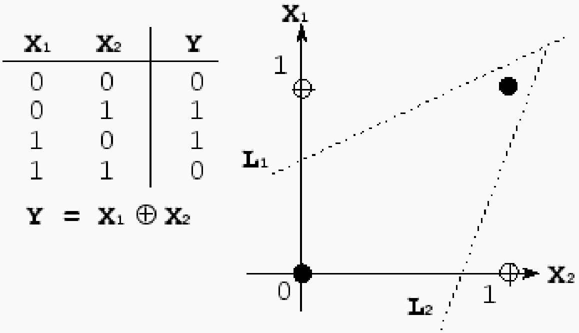


Non separable

- A neural network needs at least one hidden layer, with a non-linear activation function, to be able to learn a non-linearly separable cluster of points.

Subtleties: Linear Separability

XOR is not linearly separable, and as a result, cannot be represented by a NN without a hidden layer.



X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

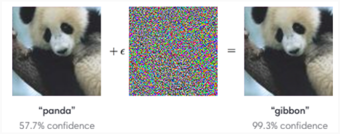
$$Y = X_1 \oplus X_2$$

Figure 1: XOR separability problem

- The first neural networks had 0 zero hidden layers, and their activation function was a step-function.
- They were called a “perceptron”.

Subtleties: Accountability

- Neural networks do have one (possibly major) drawback
- It's quite difficult to know *why* it did something (a black box)
- Often clients won't trust a black box! They might prefer decision trees which are more explainable.
- Also, we can trick trained neural networks by adding carefully-crafted image noise (creating "advaserial images")



[9]

[9]https://blog.openai.com/adversarial-example-research/

Outline

Game Applications - PacMan

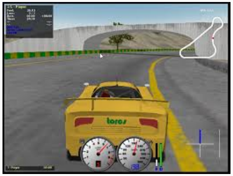
- Consider a game of pacman.
- We could record what expert players did in each situation. Form a database of their games.
- This converts the problem into a supervised learning problem - the expert's actions form the data labels we need.
- How to represent each game situation? (The representation problem.)

Game Applications - PacMan

- Could record the 4 locations of the ghosts (4 pairs of numbers (x,y))
- Plus the location of the player (one more (x,y) pair)
- And record which way the expert moved (up,down,left or right)
- Inputs: (x1,y1,x2,y2,x3,y3,x4,y4,xp,yp)
- Target outputs: ("up/down/left/right")
- We could train a neural network to play pacman!
- Q: Is this a regression problem or a classification problem?
- Q: How many inputs and outputs should our neural network have?
- Q: How many hidden layers?

Game Applications - Driving Game

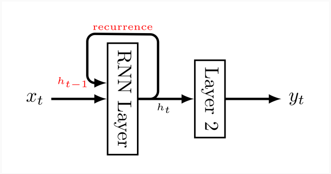
- Inputs:
  1. Location of car relative to the centre of the road
  2. Orientation of car relative to the orientation of road.
- Target outputs:
  1. Steering wheel angle.
- Will this work?
- What if we represented the inputs differently?
- E.g. just have a 640*480 screenshot flattened into an input vector!???


(Game: TORCS)

Outline

Recurrent Neural Networks

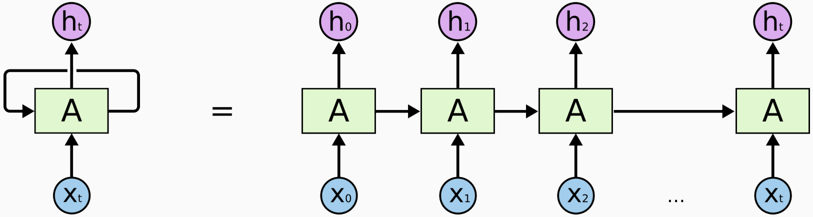
- A recurrent neural network is a graph with cycles in it, i.e. a network with one or more backward-linking connections.
- A very simple example is below.
- This allows the network to learn dependencies between temporal data
- They are built for processing time sequences of data, $\{x_t\}$, for $t = 0, 1, \ldots, T - 1$.
- Useful for video and text input



Q: Is this a ”feed-forward neural network”?

Recurrent Neural Networks

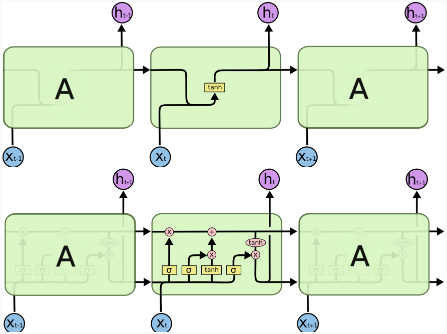
Typically, RNN are depicted in an ’unrolled’ way (time progresses to the right):



- Unfortunately, it can be difficult to train neural networks to process time sequences longer than about 10 time-steps long.
- LSTM (”Long-Short Term Memory”) networks are recurrent networks designed to ease this difficulty.

Long-Short Term Memory

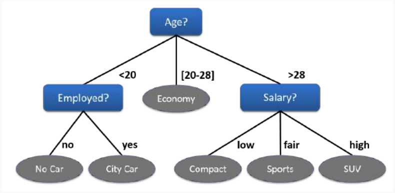
- LSTM implements 4 NN layers designed in a particular way
- Interesting post about them



Unrolled RNN (above) and LSTM (below) Neural Networks

Outline

Decision Tree Learning



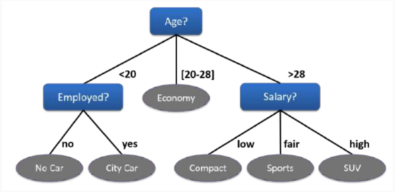
The above decision tree represents an algorithm to recommend a car to a client

- Q: What input attributes does this algorithm require about the client?
- Q: How many output categories are there?

Given a dataset of past recommendations of cars recommended by an expert salesperson, is there an algorithm to build a decision tree to represent those past recommendations?

- It's a supervised learning problem again!

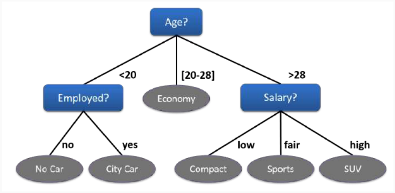
Decision-Tree Learning



What’s a good criterion for choosing the questions?

- We want each question to split the dataset as evenly as possible, where in each split there are different groups of output class.
- Look at that last question, “Employed?” That question works well in this case. But another question ”hair colour of client?” might not work as well in that position of the tree!

Decision-Tree Learning



Each question is chosen to add as much **information** as possible.

- ”Hair colour of client?” clearly adds no new information.

Mathematically, the information gain is measured by the reduction in **entropy**.

- Algorithms to build decision trees work recursively, by splitting the tree on the question that adds most “information”.

Iterative Dichotomiser 3 (ID3) i

The most notable version of decision tree learning is ID3 (and its successor, C4.5). These are the main steps of the algorithm:

1. At the start, all training examples are at the root of the tree.
2. Select the attribute (feature) with the highest heuristic value. Typical heuristics are :
   - Information Gain G(A) is based on the concept of **entropy** from information theory. It measures the different in entropy *H* before and after the dataset *D* is split on attribute *A*.
   - Gain Ratio (C4.5 only) reduces the bias of information gain (due to attributes with large number of values) by taking into account the number and size of branches when choosing an attribute.

Iterative Dichotomiser 3 (ID3) ii

3. Based on the selected attribute, construct a new node of the tree and split the dataset into subset. The possible values of the attribute become the branches of the node.
4. Repeat 2 and 3 until all samples for a node belong to the same class, there are no remaining attributes to further split or no data samples left.

ID3's successor is C4.5, an improvement over ID3 that (i) allows use of continuous input; (ii) handles incomplete data points; and (iii) uses *pruning* to avoid overfitting.

Decision Trees: Sklearn implementation

- The Python package “sklearn” includes a decision tree classifier similar to the C4.5 algorithm.
- See https://scikit-learn.org/stable/modules/tree.html for details.

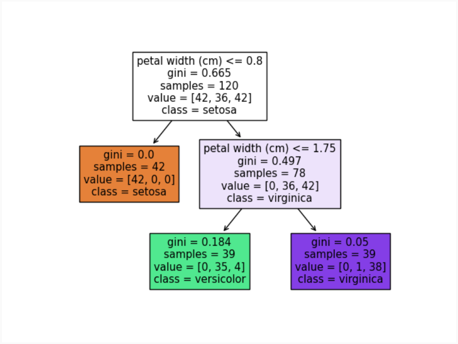
```
from sklearn import tree
clf = tree.DecisionTreeClassifier(max_depth=2, min_samples_leaf=1)
clf.fit(inputs_train, labels_train)
output_predictions=clf.predict(inputs_val)
print("output_predictions",output_predictions)

#measure accuracy of validation-set predictions:
from sklearn.metrics import accuracy_score
print("Accuracy", accuracy_score(labels_val, output_predictions))
```

Decision Trees: Visualisation

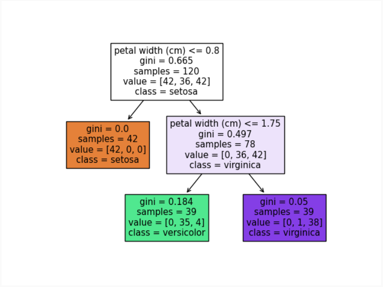
- The Python package “sklearn.tree” includes a plotter for the learned decision trees.
- Left arrows mean True, Right arrows mean False.

```
import matplotlib.pyplot as plt
fig = plt.figure()
tree.plot_tree(clf, feature_names = [...], class_names=[...], filled = True);
fig.savefig('decision_tree.png')
```



Decision Trees: Accountability

- Decision trees are liked because the final algorithm is a tree which the client can inspect and understand.
- Not a black box
- Accountable AI.
- E.g. We could invent an explainable classifier for spy-detection In the Resistance.
- Q: Can you ”explain” how this particular decision tree is making its decisions?



Outline

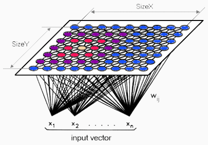
Unsupervised Learning

- In unsupervised learning, there are no labels on the data.
- We just aim to cluster common patterns input patterns into groups.
- Could be useful for clustering all the different types of playing strategies there are to play our game.
- Or could cluster the players themselves, based on their features, for potential market research.[10]

[10] See Yannakakis + Togelius, AI and Games, sec 5.6-5.7 for an example

Outline

Self-Organising Maps



- SOM perform **unsupervised** learning
- The SOM they work by applying *competitive learning*: Each time a new input is seen, the weight in the network that most closely matches the new input, and all its most closely neighbouring weights, is moved slightly closer to the new input.
- Over time, the weights will form clusters around the most commonly seen input features.

Outline

Clustering Algorithms

- Other unsupervised learning algorithms exist to do clustering, e.g. k-means clustering.
- See https://scikit-learn.org/stable/modules/clustering.html for how to do clustering easily in Python.

Outline

Lecture Quizzes

There’s a lot of information to take in this week!

- **Four** lecture quizzes this week:
    - Try to do them all before next week’s lab.
    - Usual 9-day deadline

This Week's Lab

- We will create a keras classifer neural network / decision tree in Python.
- Use Jupyter-Notebook for a further introduction in neural networks
- Before next week's lab please watch the 2 introductory videos on Introducing Neural Networks with Keras
- If you're working from home then make sure you computer is set up with jupyter notebooks plus all of the python libraries

Questions?