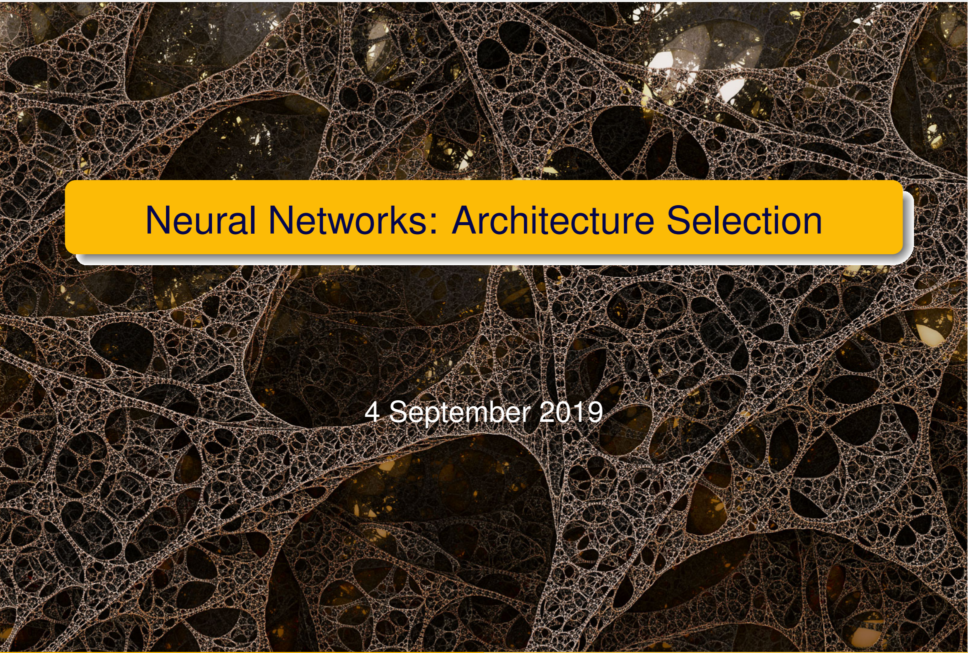
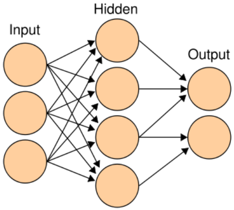


Neural Networks: Architecture Selection

4 September 2019

Architecture Selection

How many layers? How many neurons? How many weights?



- Occam's razor: the simplest network is always the best
- Too few neurons: insufficient complexity, poor (underfit) model
- Too many neurons: excessive complexity, poor (overfit) model
- What do we know about the complexity of any given problem?
    - Number of training patterns
    - Input/output dimensionality
    - Complexity of input-output relationship (function)

Architecture Selection
Making guesstimates based on the number of data patterns

- Rule of thumb: The number of training patterns should always exceed the number of free parameters.
- Why?
- Because otherwise overfitting may easily occur: free parameters will match exact patterns rather than extracting the big picture
- *Food for thought: knowledge is a result of data compression*
- If the number of patterns is the same as the number of parameters, we might as well do k-nearest-neighbour classification (no learning)

Architecture Selection

Making guesstimates based on dimensionality

Let N be the input dimensionality, M be the output dimensionality, T be the number of training patterns, and N_h be the number of hidden neurons.

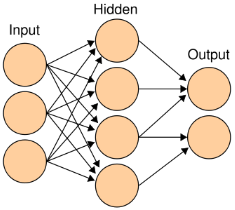
1. $N_h = (N + M)/2$
   - No mathematical justification
   - Used by default in Weka (scientific tool)
2. $N_h = T/\big(5 * (N + M)\big)$
   - Five training samples per weight (magic numbers!)
   - Used by default in Neuralware (commercial tool)
3. $N_h = \sqrt{T/(N \log T)}$
   - Optimises the mean integrated squared error for some classes of smooth functions
   - Smoothness assumption?
4. $N_h = \sqrt{N * M}$
   - Pyramidal structures (# neurons reducing from layer to layer) have shown good generalisation performance

Architecture Selection

Making guesstimates based on dimensionality

- ...Out of the four rules, the second one gave the most reliably good results on a binary benchmark of varied complexity/dimensionality
- Reason: Rule # 2 generated the largest architectures
- Larger architectures tend to make the problem easier for gradient descent
  - "Blessing of dimensionality: mathematical foundations of the statistical physics of data", A.N. Gorban, I.Y. Tyukin
  - The number of local minima reduces with an increase in dimensionality
  - Instead of geting out of local minima, gradient descent needs to get over multiple saddle points
  - Saddle points are easier to deal with than local minima!

Architecture Selection



- What about the layers?
  - More can be better (deep learning), but is harder to train
  - How much is enough?
  - `https://playground.tensorflow.org`
- What about the weights?
  - If the training algorithm is good, it should be capable of setting irrelevant weights to zero
  - Regularisation: minimise not only the error, but also the complexity

Regularisation

Penalizing complexity

- Add a penalty term to the objective function:
  - $E_{NN} = E + \lambda E_p$
- Now we are minimizing both the **error** and the **complexity**

How do you measure complexity?

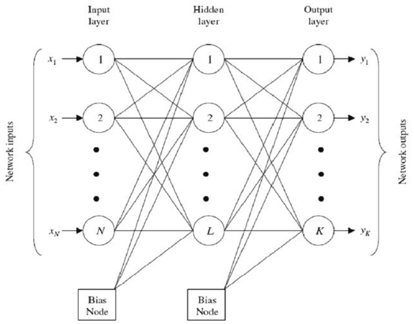
- Weight decay:
  - $E_p = \sum_{i=1}^{W} w_i^2$
  - Minimize weight vector magnitude; only constantly reinforced weights will survive
- Weight elimination:
  - $E_p = \sum_{i=1}^{W} \frac{w_i^2/w_0^2}{1+w_i^2/w_0^2}$
  - w_0 determines the “significance” of weights
  - $|w_i| >> w_0$ => high complexity, penalize more
  - $|w_i| << w_0$ => low complexity, penalize less

Regularisation

Penalizing complexity

- Laplace: L1 regularisation
  - $E_p = \sum_{i=1}^{W} |w_i|$
  - Contribution of each w to the penalty term increases linearly with the increase of the weight
- Multiple other penalty functions were proposed
- Consider the objective function:
  - $E_{NN} = E + \lambda E_p$
- How do we choose λ?
  - Cross-validation
  - Make it adaptive?

Regularisation and the Bias Weights



Should we penalise the biases?

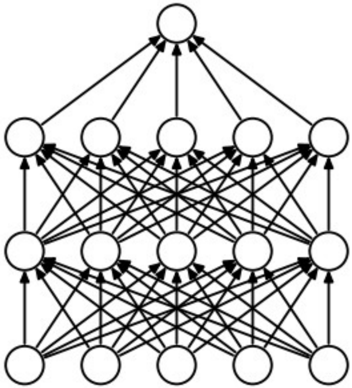
- Regularisation makes the function smoother, i.e. less sensitive to changes in the input
- Biases provide constant input
- In practice, we usually **do not** regularise the bias weights
- Regularising biases may cause underfitting

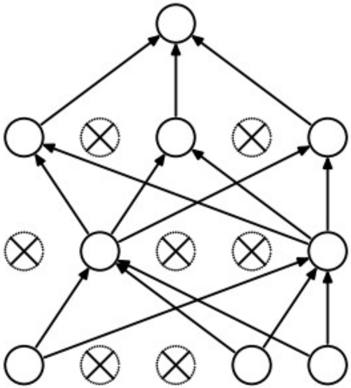
Regularisation

Dropout: a new form of regularization

- Hinton, 2012: overfitting occurs because the model is too complex, eg. each hidden unit relies on neighbour units to make the final prediction
- “Dropout”: On each presentation of each training pattern, each hidden unit is randomly omitted from the network with a probability of 0.5, so a hidden unit cannot rely on other hidden units being present
- Force hidden units to “take responsibility”
- More robust models are obtained by preventing “co-adaptation”
- Can be combined with other regularisation methods

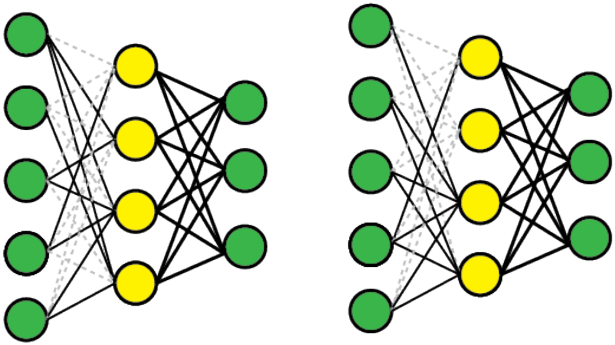
Regularisation
Dropout: a new form of regularization


(a) Standard Neural Net


(b) After applying dropout.

Regularisation
Dropconnect: a generalisation of Dropout

You can also “disable” weights rather than neurons:



Essentially, we train an ensemble that looks like a single NN

Neural Network Construction

How to automate architecture selection

How do we automatically construct an optimal architecture?

Minimalistic approach

- Start with just a few neurons, add more when stagnation occurs
  - Cascade correlation NNs embraced this principle
  - The NN contains a working model when a new neuron is added => integrating the new neuron may slow down training
  - How do we decide when to add a neuron, and when to stop growing?
  - How do we expand this to adding layers?
  - What about adding recurrent connections?

Neural Network Construction

Evolutionary approach

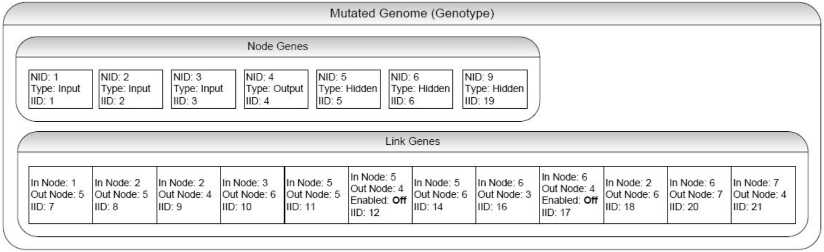
- Optimise the weights and/or the architecture using a genetic algorithm
    - Evolutionary algorithms were successfully used to “evolve” NN architectures
    - If you can represent it, you can evolve it
    - Probably the best “growing” approach
    - Start with a perceptron-like architecture: inputs + outputs
    - Allow the algorithm to establish new neurons and connections
    - Evolving NNs is a slow process

Evolutionary pruning

- Make different architectures compete for survival
- Assign higher fitness to smaller architectures

Neural Network Construction

NEAT: NeuroEvolution of Augmenting Topologies



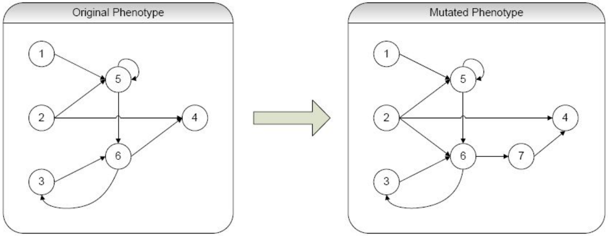
Mutated Genome (Genotype)

Node Genes

NID: 1 Type: Input IID: 1	NID: 2 Type: Input IID: 2	NID: 3 Type: Input IID: 3	NID: 4 Type: Output IID: 4	NID: 5 Type: Hidden IID: 5	NID: 6 Type: Hidden IID: 6	NID: 9 Type: Hidden IID: 19

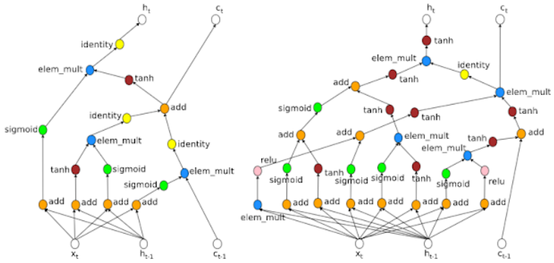
Link Genes

In Node: 1 Out Node: 5 IID: 7	In Node: 2 Out Node: 5 IID: 8	In Node: 2 Out Node: 4 IID: 9	In Node: 3 Out Node: 6 IID: 10	In Node: 5 Out Node: 5 IID: 11	In Node: 5 Out Node: 4 Enabled: **Off** IID: 12	In Node: 5 Out Node: 6 IID: 14	In Node: 6 Out Node: 3 IID: 16	In Node: 6 Out Node: 4 Enabled: **Off** IID: 17	In Node: 2 Out Node: 6 IID: 18	In Node: 6 Out Node: 7 IID: 20	In Node: 7 Out Node: 4 IID: 21



Neural Network Construction

- NEAT uses direct encoding: every node/connection is stored in the representation
- Indirect encoding:
  - Define primitves (layers, neuron types, activation functions...)
  - Representation is a graph of primitives
  - Allows to re-use the primitives/subgraphs iteratively/recursively
  - The research is ongoing!



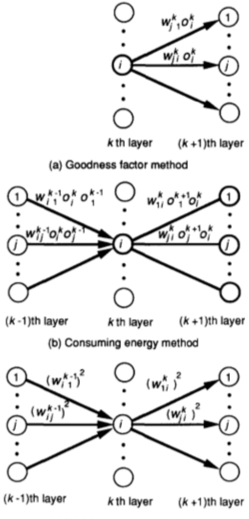
Neural Network Pruning

To grow or to prune?

Applying Occam's Razor

- Start with an oversized architecture, remove unnecessary parameters
  - Weights
  - Hidden units
  - Input units
  - Need a way of quantifying relevance of each parameter
- Large architectures have large functional flexibility => a lot of potential for a good fit
- And a lot of potential for an over-fit?..

Neural Network Pruning



Intuitive pruning

- Determine the “active” neurons, remove inactive ones
  - “An important unit is the one that fires frequently and has strong connections to other units”
  - $G_i = \sum_P \sum_j (w_{ji} o_i)^2$ - Goodness factor
  - $E_i = \sum_P \sum_j (w^l_{ji} o^l_i o^{l+1}_j)$ - Consuming energy
- Units that output 0 more often than 1 are considered irrelevant - is it fair?
- Weight magnitude pruning: remove small weights

Neural Network Pruning

Information Matrix pruning

- Fisher information: a way of measuring the amount of information that an observable random variable X carries about an unknown parameter θ upon which the probability of X depends.
  - $I = \frac{1}{P}\sum_{p=1}^{P} \frac{\partial f_{NN}}{\partial w}(\frac{\partial f_{NN}}{\partial w})^T$ - approx. information matrix
  - Calculates the covariance of the weights
  - Captures curvature, just like the Hessian
  - May be time- (and memory-) consuming to compute
  - Prune the weights that bear the least information
- Principal Component Analysis (PCA): prune parameters (weights) that do not account for data variance
- All of these techniques do not scale very well to large NNs

Neural Network Pruning

Hypothesis Testing

- Use statistical tests to calculate the significance of weights/hidden units
  - Null hypothesis: a subset of weights is equal to zero
  - If weights associated with a neuron are not statistically different from zero, prune the neuron
- Input pruning: Inject a noisy input
  - If the statistical significance of an original parameter is not higher than that of random noise, prune the parameter
- Assume that weights are $\approx$ normally distributed
  - Remove the weights that are in the distribution tails

Neural Network Pruning

Sensitivity analysis pruning

- Saliency: the influence small perturbations to a parameter have on the approximated error/output function
- Prune parameters with low saliency
- Optimal Brain Damage (OBD), introduced by Yann LeCun:
  1. Choose a reasonable NN architecture
  2. Train until a reasonable solution is obtained
  3. Compute second order derivatives h_{kk} for each parameter (diagonal of the Hessian matrix)
  4. Compute the saliencies for each parameter: $s_k = h_{kk} w^2 / 2$
  5. Sort parameters by saliency and delete low-saliency ones
  6. Go back to step 2
- Optimal Brain Surgeon (OBS) - adjust weights
- Optimal Cell Damage (OCD) - prune inputs
- Hessians are expensive to calculate

Neural Network Training
Passive VS Active

- NN architecture and training algorithm are important, but so is the data
- Data contributes to the complexity of the model

Passive learning

Neural network passively accepts the training data, and tries to fit the data as well as possible

Active learning

Neural network is presented with a candidate training set. Heuristics are then used to choose the patterns that are most informative

Active Learning

- Redundant data may slow down the training
- If one class is over-represented, it may bias the NN
- Choosing most informative and relevant patterns:
    - Decrease training time
    - Improve generalisation
- Two main active learning approaches:
    - Selective learning
    - Incremental learning

Selective learning

Selecting patterns for training

- Given a candidate set, a subset of informative patterns is chosen as the training set
- The model is trained until convergence/stopping criteria
- New cycle starts by selecting a new subset for training

- Selective Updating:
  - Start training on the candidate set
  - At each epoch, see which patterns had the most influence on the weights, and discard the patterns that had the least influence
  - Training set may change from epoch to epoch
- Discard the patterns that have been classified correctly: this knowledge has already been absorbed
- Engelbrecht: choose patterns that are close to decision boundaries (sensitivity analysis)

Incremental learning

Training incrementally

- Given a candidate set, a subset of informative patterns is chosen as the training set
- That subset of patterns is removed from the candidate set
- The model is trained until convergence/stopping criteria
- New cycle starts by adding more patterns from the candidate set to the training set
- As training progresses, the candidate set decreases, and the training set grows
- Incremental learning does not discard patterns. Rather, it attempts to get the "best" ones first, and uses "weaker" ones to tweak a working model later
- Eventually, the entire candidate set may be used for training

Incremental learning

Information theory

- Most incremental learning approaches are based on information theory (Fisher information matrix)
- Optimal Experiment Design:
  - At each iteration, choose a pattern from the candidate set that minimizes the expected value of the error
  - Expensive: need to calculate the information matrix inverse
- A problem: Fukumizu showed that the Fisher information matrix may be singular
  - What if it does not have an inverse?
  - Same paper: Fisher matrix is singular iff the are redundant units
  - Remove units => solve the problem
  - 2-in-1: architecture selection + incremental learning
  - Very complex and computationally heavy

Incremental learning

Simpler approaches

- Information gain can be maximized by simply choosing patterns that yield the largest error
- Use Robel's factor ($\frac{E_G}{E_T}$): when overfitting is observed, add patterns that yield the largest errors
- Engelbrecht: Patterns that yield midrange sigmoid outputs are the most informative
- Many more methods exist, but all suffer from the following:
  - Overhead of using a heuristic
  - If we use more time to pick patterns than we save on training, was it worth it?
  - The data set should be bad/hard enough to justify these techniques

The End

- Questions?
- Next week: Wednesday is Spring Day, no lecture
- An extra lecture may be scheduled for a later slot (18:30 - 20:30) on Mon, Tue, or Fri next week
- Next lecture: Deep Learning