Course 395: Machine Learning - Lectures

Lecture 1-2: Concept Learning (M. Pantic)

Lecture 3-4: Decision Trees & CBC Intro (M. Pantic & S. Petridis)

Lecture 5-6: Evaluating Hypotheses (S. Petridis)

Lecture 7-8: Artificial Neural Networks I (S. Petridis)

Lecture 9-10: Artificial Neural Networks II (S. Petridis)

Lecture 11-12: Instance Based Learning (M. Pantic)

Lecture 13-14: Genetic Algorithms (M. Pantic)
Why squared error is not a good choice for sigmoid output activation functions

- See http://neuralnetworksanddeeplearning.com/chap3.html

- For output units: \( \Delta w_{ki} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(\text{net}_k)}{\partial \text{net}_k} x_{ki} \)

- \( \frac{\partial E}{\partial o_k} = -(t_k - o_k) \) when the error function is the squared loss

- \( \frac{\partial \sigma(\text{net}_k)}{\partial \text{net}_k} = \sigma(\text{net}_k)(1 - \sigma(\text{net}_k)) = o_k(1 - o_k) \) when \( \sigma \) is sigmoid
Why squared loss is not a good choice for sigmoid output activation functions

• See http://neuralnetworksanddeeplearning.com/chap3.html

• For output units: \( \Delta w_{ki} = \eta (t_k - o_k) o_k (1 - o_k) x_{ki} \)

• When the output is 0 or 1 then \( \Delta w \) is 0 as well

• If target is 1 and network’s output is 1 then \( \Delta w = 0 \) (good)

• If target is 1 and network’s output is 0 then \( \Delta w = 0 \) (bad!!!)
Cross Entropy Error as Error Function

- A good error function when the output activation functions are sigmoid is the binary cross entropy defined as follows:

\[ E = -\frac{1}{D} \sum_{d=1}^{D} \left( t_d \ln o_d + (1 - t_d) \ln(1 - o_d) \right) \]

\( D = \) number of training examples

- For output units: \( \Delta w_{ki} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(\text{net}_k)}{\partial \text{net}_k} x_{ki} \)

- \( \frac{\partial E}{\partial o_k} = \frac{o_k - t_k}{o_k(1 - o_k)} \)

- \( \frac{\partial \sigma(\text{net}_k)}{\partial \text{net}_k} = \sigma(\text{net}_k)(1 - \sigma(\text{net}_k)) = o_k(1 - o_k) \)
Cross Entropy Error as Error Function

• For output units: \( \Delta w_{ki} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(\text{net}_k)}{\partial \text{net}_k} x_{ki} \)

\[
\Delta w_{ki} = -\eta \frac{o_k - t_k}{o_k(1 - o_k)} o_k(1 - o_k)x_{ki} = \eta(t_k - o_k)x_{ki}
\]

• The higher the error the higher the weight update
**Softmax output activation functions**

- A popular output activation function for classification is
  \[ o_k = \frac{e^{net_k}}{\sum_k e^{net_k}} \]

- The output can be interpreted as a discrete probability distribution.

- The right error function is the negative log likelihood cost
  \[ E = -\sum_k t_k \ln o_k \]

- Target vectors = [0 0 1 … 0] \(\rightarrow\) \(E = -\ln o_L\) where \(L\) is the position of the active target, i.e., it is 1.

- It is equivalent to the binary cross entropy for 2 classes.
Output activation functions: Summary

• For each output activation function the right error function should be selected

• Sigmoid $\rightarrow$ Cross entropy error (useful for classification)

• Softmax $\rightarrow$ negative log likelihood cost (useful for classification)

• Both combinations work well for classification problems, Softmax has the advantage of producing a discrete probability distribution over the outputs

• Linear $\rightarrow$ MSE (useful for regression)
**Backpropagation with momentum**

- Standard backpropagation

  \[ w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]

- If the error surface is a long and narrow valley, gradient descent goes quickly down the valley walls, but very slowly along the valley floor.

Backpropagation with momentum

• Standard backpropagation

\[ w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]

• Backpropagation with momentum

\[ \Delta w_i(t) = \mu \Delta w_i(t - 1) + (1 - \mu) \left( -\eta \frac{\partial E}{\partial w_i(t)} \right) \] OR

\[ \Delta w_i(t) = \mu \Delta w_i(t - 1) + \left( -\eta \frac{\partial E}{\partial w_i(t)} \right) \]

• \( \mu = \) momentum constant, usually 0.9, 0.95
• It is like giving momentum to the weights
• We do not take into account only the local gradient but also recent trends in the error surface
• Matlab function: \texttt{traingdm}
Backpropagation with adaptive learning rate

- It is good that the learning rate is not fixed during training
  \[ w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]

- Simple heuristic
  1. If error decreases, increase learning rate: \( \eta = \eta \times \eta_{inc} \)
  2. If error increases, decrease learning rate and don’t update the weights: \( \eta = \eta \times \eta_{dec} \)

- Typical values for \( \eta_{inc} = 1.05, 1.1 \)
- Typical values for \( \eta_{dec} = 0.5, 0.7 \)
- Matlab function: \texttt{traingda}
Resilient Backpropagation

- The weight change depends on the learning rate and the value of the partial derivative. We have no control over the partial derivative.

- The effect of the learning rate can be disturbed by the unforeseeable behaviour of the derivative.

- Resilient backpropagation uses only the sign of the derivative!!

- For each weight $w_i$ we define an individual update value $\Delta_i$ which depends only on the sign of the derivative and ignores its actual value
Resilient Backpropagation

\[
\Delta_i(t) = \begin{cases} 
\Delta^{inc} \cdot \Delta_i(t-1) & \text{if } \frac{\partial E^t}{\partial w_i} \cdot \frac{\partial E^t}{\partial w_i} > 0 \\
\Delta^{dec} \cdot \Delta_i(t-1) & \text{if } \frac{\partial E^t}{\partial w_i} \cdot \frac{\partial E^t}{\partial w_i} < 0 
\end{cases}
\]

- Every time the partial derivative changes its sign, i.e., last update was too big, the update value is decreased.

- If the derivative retains its sign, the update value is increased in order to accelerate convergence.
Resilient Backpropagation

\[
\Delta w_i(t) = \begin{cases} 
  -\Delta_i(t) & \text{if } \frac{\partial E^t}{\partial w_i} > 0 \\
  \Delta_i(t) & \text{if } \frac{\partial E^t}{\partial w_i} < 0
\end{cases}
\]

- We also need to initialise the update values \( \Delta_i \)
- We usually define an upper limit for \( \Delta_i \)
- Typical values for \( \Delta_{inc} = 1.2 \)
- Typical values for \( \Delta_{dec} = 0.5 \)
- Matlab function: \texttt{trainrp}
Other Training Algorithms

- Conjugate gradient
- Levenberg-Marquardt
- Hessian-free
- And many others…not covered in the lecture
**Batch/Mini-batch/Stochastic Gradient Descent**

- **Batch**: All examples are fed to the network. Weights are updated only after all examples have been presented to the network.

- For each weight the corresponding gradient (or \( \Delta w \)) is computed (for each example).

- The weights are updated based on the average gradient over all examples.

- **Stochastic/Incremental/On-line**: One example at a time is fed to the network.

- Weights are updated after each example is presented to the network.
**Batch/Mini-batch/Stochastic Gradient Descent**

- Mini-Batch: $M$ (usually 100) randomly examples are fed to the network.

- For each weight the corresponding gradient (or $\Delta w$) is computed (for each example).

- The weights are updated based on the average gradient over all $M$ examples.

- Set of $M$ examples is called mini-batch.

- Popular approach in deep neural networks.

- Sometimes called stochastic gradient descent (NOT to be confused with online/incremental gradient descent).
(Hyper)Parameters / Weights

- (Hyper)Parameters are what the user specifies, e.g. number of hidden neurons, learning rate, number of epochs etc

- They need to be optimised

- Weights are the weights of the network

- They are also parameters but they are optimised automatically via gradient descent
Ways to avoid overfitting

• Early stopping (see slide 52)
• L1 Regularisation
• L2 Regularisation
• Dropout
• Max-norm Constraint
• Data augmentation
L2 Regularisation

- \( E = E_0 + \lambda \sum_{all\ weights} w^2 \)

\( E_0 \) is the original error function, e.g., squared loss, negative log-likelihood

- It is NOT applied to the bias

- We wish to minimise the original error function \( (E_0) \)

- We also wish to penalise large weights, keep the weights small (second term)

- Small \( \lambda \) \( \rightarrow \) we prefer to minimise \( E_0 \)

- Large \( \lambda \) \( \rightarrow \) we prefer small weights
L1 Regularisation

- $E = E_0 + \lambda \sum_{all \text{ weights}} |w|$

- $E_0$ is the original error function, e.g., squared loss, negative log-likelihood

- It is NOT applied to the bias

- We wish to minimise the original error function ($E_0$)

- We also wish to penalise large weights, keep the weights small (second term)

- Small $\lambda \rightarrow$ we prefer to minimise $E_0$

- Large $\lambda \rightarrow$ we prefer small weights
L1/L2 Regularisation

• So what’s the difference between L1 and L2 regularisation?

• L2: \( \frac{\partial E}{\partial w} = \frac{\partial E_0}{\partial w} + \lambda w \rightarrow \Delta w = -\eta \frac{\partial E_0}{\partial w} - \eta \lambda w \)

• L1: \( \frac{\partial E}{\partial w} = \frac{\partial E_0}{\partial w} + \lambda \text{sign}(w) \rightarrow \Delta w = -\eta \frac{\partial E_0}{\partial w} - \eta \lambda \text{sign}(w) \)

• L1: The weights shrink by a constant amount towards 0

• L2: The weights shrink by an amount proportional to \( w \)

• L1 drives small weights to zero
L1/L2 Regularisation

• Why small weights prevent overfitting?

• When weights are 0 or close to zero this equivalent to removing the corresponding connection between the neurons

• Simpler architecture $\rightarrow$ avoids overfitting

• Network has the right capacity

• It is like we start with a high capacity (complex) network until we find a network with the right capacity for the problem
**Dropout**

- We don’t modify the error function but the network itself.
- During training neurons are randomly dropped out.
- The probability that a neuron is present is $p$.

From Dropout: A simple way to prevent neural networks from overfitting by Srivastava et al., JMLR 2014.
Dropout

• Dropout prevents overfitting because it provides a way of approximately combining exponentially many different neural network architectures.

• Typical values for $p$: 0.8/0.5 for input/hidden neurons

• At test time the outgoing weights of a neuron are multiplied by $p$

From Dropout: A simple way to prevent neural networks from overfitting by Srivastava et al., JMLR 2014
Max-Norm Regularisation

- Constrain the norm of the incoming weight vector at each hidden unit to be upper bounded by a fixed constant $c$.

- Weight vector length: $L = \sqrt{w_{j1}^2 + w_{j2}^2 + \ldots + w_{jN}^2}$

- $w_{ji}$ corresponds to incoming weights to neuron $j$ from the $N$ neurons of the previous layer.

- If $L > c$ then multiply all the incoming weights by $c/L$.

- The new vector length is $c$.

- Another approach to keep the weights small.

- Usually used in combination with dropout.
Data Augmentation

• One of the best ways to avoid overfitting is more data

• So we can artificially generate more data, usually a bit noisy, so we introduce more variation

• We should apply operations that correspond to real-world variations.

• For images: flip left-right, rotate, translate, etc
Vanishing/Exploding gradient

\[ \Delta w_{ji} = \eta \delta_j x_{ji} \]
\[ \delta_j = \sum_{k=\text{outputNeuronsConnectedTo}j} \delta_k w_{kj} \frac{\partial \sigma(\text{net}_j)}{\partial \text{net}_j} \]

• As we backpropagate through many layers:
  1. If the weights are small \( \rightarrow \delta_i \) shrink exponentially
  2. If the weights are big \( \rightarrow \delta_i \) grow exponentially
• So either the network stops learning (case 1) or becomes unstable (case 2)
• That is why it is not possible to train deep networks with backpropagation
Deep NNs

3-layer feed-forward network

4-layer feed-forward network

- There is a pre-training phase where weights are initialised to a good starting point.
- Pre-training is performed per layer using Restricted Boltzmann Machines or Stacked Denoising Autoencoders.
- Then backpropagation is used to fine-tune the weights starting from a good initialisation point.
Stacked Denoising Autoencoder

- Train a network to reproduce its input
- This network is called an Autoencoder (AE)
- The idea is that the middle layer represents the main variations in the data
- The problem is that the AE may simply learn the identity function

From https://www.mql5.com/en/articles/1103#2_2
Stacked Denoising Autoencoder

- Denoising AE: we add noise to the input so the network learns to reconstruct (output) the “denoised” input
- We usually set as many as half of the inputs to 0
- The network tries to reconstruct the input and undo the effect of noise
- The hidden layer is “forced” to learn the main variations in the data

From https://www.mql5.com/en/articles/1103#2_2
Stacked Denoising Autoencoder

- The hidden layer weights of the AE are copied to the feed-forward NN
- The output of the hidden layer is used as input for the 2\textsuperscript{nd} AE
- Noise is added to this new input and the 2\textsuperscript{nd} AE learns to “denoise” its input

From https://www.mql5.com/en/articles/1103#2_2
Stacked Denoising Autoencoder

- The hidden layer weights of the 2\textsuperscript{nd} AE are copied to the feed-forward NN.
- The output of the hidden layer is used as input for the 3\textsuperscript{rd} AE.
- Using this approach we can add as many as layers as we want.

From https://www.mql5.com/en/articles/1103#2_2
Stacked Denoising Autoencoder

- This approach is used to initialise the NN
- This is called pre-training
- It results in good initialisation of the weights
- Then we fine-tune the network using stochastic gradient descent

From https://www.mql5.com/en/articles/1103#2_2
Deep Networks for Time Series

- Deep feedforward NNs are good at various tasks but not at handling time series data
- Recurrent Neural Networks are suitable for time series
- They also suffer from the vanishing gradient problem
LSTMs

• A type of recurrent network that can be effectively trained is the Long-Short Term Memory Recurrent Neural Network (LSTM-RNN). Introduced in 1990s

• We replace the neuron with a memory cell

• There are input, output and forget gates which control when information flows in / out of the cell and when to reset the state of the cell
LSTMs

From LSTM: A search space odyssey by Greff et al., arXiv Mar 2015
Convolutional Neural Networks

• Convolutional Neural Networks (CNNs) have been very successful in computer vision

• First version was introduced in 1980s (neocognitron)

Convolutional Neural Networks

- Became popular in 2012 after winning the ImageNet competition

- “ImageNet Classification with Deep Convolutional Neural Networks”, by Krizhevsky et al., NIPS 2012

- Tricks: Data augmentation, Dropout, ReLu + GPUs
Convolutional Neural Networks

- It’s a deep network = many layers
- Each layer is either a convolutional layer or subsampling layer
- Final layers are fully connected layers
Convolutional Layers


- The kernel, i.e., weights, are fixed. After convolving the image with the kernel we end up with a feature map.

Convolutionsal Layers

- Different kernels, i.e., weights, are applied to the image and each of them produces a feature map (kernel is fixed for each feature map)

- The weights are learned during training
Subsampling Layers

- Each feature map is downsamples using average pooling or max-pooling

- Use non-overlapping patches of 2x2 and take the average or maximum of the pixels as the output
Deep NNs Applications

• Deep Face by Facebook – State of the art in Face verification
  - DeepFace: Closing the Gap to Human-Level Performance in Face Verification, Taigman, Ming, Ranzato, Wolf

• State-of-the-art performance in Speech Recognition
  - Microsoft has done a lot of research on this topic
ImageNet Competition – Object Classification

- Classification of 1000+ objects
- State-of-the-art before 2012: ~26%
- New state-of-the-art in 2012 with deep networks: ~15%
Practical Suggestions: Activation Functions

- Continuous, smooth (essential for backpropagation)
- Nonlinear
- Saturates, i.e. has a min and max value
- Monotonic (if not then additional local minima can be introduced)
- Sigmoids are good candidate (log-sig, tan-sig) or ReLu
Practical Suggestions: Activation Functions

- In case of regression, then output layer should have linear activation functions.
Practical Suggestions: Scaling Inputs

• It is not desirable that some inputs are orders of magnitude larger than other inputs

• Map each input $x(i)$ to $[-1, +1]$ using this formula

$$y = 2 \frac{x-x_{min}}{x_{max}-x_{min}} - 1$$

• Matlab function: *mapminmax*
  - Rows: features, columns: examples
  - Normalises each row
  - Useful for continuous inputs/targets
Practical Suggestions: Scaling Inputs

• Standardize inputs to mean=0 and 1 std. dev.=1
  \[ y = \frac{x - x_{\text{mean}}}{x_{\text{std}}} \]

• Useful for continuous inputs/targets

• Matlab function: `mapstd`

• Scaling is needed if inputs take very different values. If e.g., they are in the range \([-3, 3]\) then scaling is probably not needed
Practical Suggestions: Scaling Inputs

- The scaling values, $x_{min}, x_{max}, x_{mean}, x_{std}$ are computed on the training set and then applied to the validation and test sets.

- It is not correct to scale each set separately.
Practical Suggestions: Scaling Inputs

• Matlab automatically scales the inputs to [-1, 1] and removes the inputs/outputs that are constant.

• Think if you wish to scale the inputs, if not you should disable the automatic scaling

• Check http://www.mathworks.co.uk/help/nnet/ug/choose-neural-network-input-output-processing-functions.html
Practical Suggestions: Target Values

• Binary Classification
  - Target Values: -1/0 (negative) and 1 (positive)
  - 0 for log-sigmoid, -1 for tan-sigmoid

• Multiclass Classification
  - [0,0,1,0] or [-1, -1, 1, -1]
  - 0 for log-sigmoid, -1 for tan-sigmoid

• Regression
  Target values: continuous values [-inf, +inf]
Practical Suggestions: Number of Hidden Layers

• Networks with many hidden layers are prone to overfitting and they are also harder to train

• For most problems one hidden layer should be enough

• 2 hidden layers can sometimes lead to improvement

• If you want to use more layers then you should follow the deep learning methodology to initialise the weights and use strong regularisation (dropout etc)
Division of data

- Matlab automatically divides the dataset into training/validation/test sets.

- You should force matlab to use an empty test set (you have your own) and use the same validation set as yours.

- You can provide the indices of your validation set and your test set (=empty array)

Matlab Examples

• Create feedforward network
  - net = feedforwardnet(hiddenSizes, trainFcn)
  - hiddenSizes = [10, 10, 10] – 3 layer network with 10 hidden neurons in each layer
  - trainFcn = ‘trainlm’, ‘traingdm’, ‘trainrp’ etc

• Configure (set number of input/output neurons)
  - net = configure(net, x, t)
  - x: input data, noFeatures x noExamples
  - t: target data, noClasses x noEx
Matlab Examples

• Train network
  - \([\text{net}, \text{tr}] = \text{train} (\text{net}, P, T)\)
  - P: input data
  - T: target data

• Simulate network
  - \([\text{Y}, \text{Pf}, \text{Af}, \text{E}, \text{perf}] = \text{sim} (\text{net}, P)\)
Matlab Examples

- Batch training: Train
- Incremental Training: Adapt
- Use Train for the CBC
Questions

- Questions from book: 4.1, 4.2, 4.5, 4.8, 4.10

- You should read chapter 4

- Examinable material: Slides, Manual (part 3A), Chapter 4