# COMP9444
## Neural Networks and Deep Learning
### 17. Course Review

### Assessment
Assessment will consist of:
- Assignments 40%
- Written Exam 60%

In order to pass the course, you must score:
- at least 16/40 for the assignments
- at least 24/60 for the written exam
- a combined mark of at least 50/100

### Planned Topics
- Neuroanatomy (9.10)
- Perceptrons, Backpropagation (5.4-5.5, 6.1-6.5)
- Hidden Unit Dynamics (8.2-8.3)
- Recurrent Networks (10.2)
- Convolutional Networks (7.12-7.13, 9.1-9.4)
- Long Short Term Memory (10.5-10.7, 10.10)
- Autoencoders (14.1-14.5)
- Unsupervised Learning (5.8, 15.1)
- Hopfield Networks
- Restricted Boltzmann Machines (16.7, 20.1-20.3)
- Generative Models (19.4, 20.9, 20.10)

### Topics Covered
1. Neuroanatomy
2. Perceptrons
3. Backpropagation
4. Variations on Backprop
5. Geometry
6. Convolutional Networks
7. Image Processing
8. Recurrent Networks
9. Language Processing
10. Reinforcement Learning
11. Deep Reinforcement Learning
12. Boltzmann Machines
13. Autoencoders
14. Adversarial Training and GANs
15. Coevolution
16. Evolutionary Art
Not Examinable

- Neuroanatomy (Lect 1)
- Coevolution (Lect 15)
- Evolutionary Art (Lect 16)

Final Exam

- 2-hour exam, centrally managed
- worth 60% of final mark
- NO textbook or course notes
- approved calculators may be used
- Exercises and Quizzes on the Course Web site give a good indication of the kind of questions to expect in the exam
- Sample Exam now available on Course Web Site

Structure of a Typical Neuron

The brain is made up of neurons (nerve cells) which have
- a cell body (soma)
- dendrites (inputs)
- an axon (outputs)
- synapses (connections between cells)

Synapses can be excitatory or inhibitory and may change over time.
When the inputs reach some threshold an action potential (electrical pulse) is sent along the axon to the outputs.
Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)

Possible solution:

\[ x_1 \text{ XOR } x_2 \text{ can be written as: } (x_1 \text{ AND } x_2) \text{ NOR } (x_1 \text{ NOR } x_2) \]

Recall that AND, OR and NOR can be implemented by perceptrons.

McCulloch & Pitts Model of a Single Neuron

\[ s = w_1x_1 + w_2x_2 - \text{th} \]

\[ x_1, x_2 \text{ are inputs} \]
\[ w_1, w_2 \text{ are synaptic weights} \]
\[ \text{th is a threshold} \]
\[ w_0 \text{ is a bias weight} \]
\[ g \text{ is transfer function} \]

Perceptron Learning Rule

Adjust the weights as each input is presented.

Recall: \[ s = w_1x_1 + w_2x_2 + w_0 \]

\[
\begin{align*}
  \text{if } g(s) &= 0 \text{ but should be 1,} & \text{if } g(s) &= 1 \text{ but should be 0,} \\
  w_k &\leftarrow w_k + \eta x_k & w_k &\leftarrow w_k - \eta x_k \\
  w_0 &\leftarrow w_0 + \eta & w_0 &\leftarrow w_0 - \eta \\
  \text{so } s &\leftarrow s + \eta (1 + \sum_k x_k^2) & \text{so } s &\leftarrow s - \eta (1 + \sum_k x_k^2) \\
  \text{otherwise, weights are unchanged. } (\eta > 0 \text{ is called the } \text{learning rate})
\end{align*}
\]

Theorem: This will eventually learn to classify the data correctly, as long as they are linearly separable.

Multi-Layer Neural Networks

Problem: How can we train it to learn a new function? (credit assignment)
Types of Learning

- Supervised Learning
  - agent is presented with examples of inputs and their target outputs
- Reinforcement Learning
  - agent is not presented with target outputs, but is given a reward signal, which it aims to maximize
- Unsupervised Learning
  - agent is only presented with the inputs themselves, and aims to find structure in these inputs

Ockham’s Razor

“The most likely hypothesis is the simplest one consistent with the data.”

Since there can be noise in the measurements, in practice need to make a tradeoff between simplicity of the hypothesis and how well it fits the data.

Two-Layer Neural Network

Output units  \( a_i \)

Hidden units  \( a_j \)

Input units  \( a_k \)

Normally, the numbers of input and output units are fixed, but we can choose the number of hidden units.

Local Search in Weight Space

Problem: because of the step function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and “shoulders”, with occasional discontinuous jumps.
Key Idea

Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

\[ g(s) = \frac{1}{1 + e^{-s}} \]

or hyperbolic tangent

\[ g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\left(\frac{1}{1 + e^{-2s}}\right) - 1 \]

Variations on Backprop

- Cross Entropy
  - problem: least squares error function unsuitable for classification, where target = 0 or 1
  - mathematical theory: maximum likelihood
  - solution: replace with cross entropy error function
- Weight Decay
  - problem: weights “blow up”, and inhibit further learning
  - mathematical theory: Bayes’ rule
  - solution: add weight decay term to error function
- Momentum
  - problem: weights oscillate in a “rain gutter”
  - solution: weighted average of gradient over time

Gradient Descent (4.3)

Recall that the error function \( E \) is (half) the sum over all input patterns of the square of the difference between actual output and desired output

\[ E = \frac{1}{2} \sum (z - t)^2 \]

The aim is to find a set of weights for which \( E \) is very low.

If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

\[ w \leftarrow w - \eta \frac{\partial E}{\partial w} \]

Parameter \( \eta \) is called the learning rate.

Cross Entropy

For classification tasks, target \( t \) is either 0 or 1, so better to use

\[ E = -t \log(z) - (1 - t) \log(1 - z) \]

This can be justified mathematically, and works well in practice – especially when negative examples vastly outweigh positive ones. It also makes the backprop computations simpler

\[ \frac{\partial E}{\partial z} = \frac{z - t}{z(1 - z)} \]

if \( z = \frac{1}{1 + e^{-s}} \),

\[ \frac{\partial E}{\partial s} = \frac{\partial E}{\partial z} \frac{\partial z}{\partial s} = z - t \]
**Bayes’ Rule (3.11)**

The formula for conditional probability can be manipulated to find a relationship when the two variables are swapped:

\[ P(a \land b) = P(a \mid b)P(b) = P(b \mid a)P(a) \]

\[ \rightarrow \text{Bayes’ rule } P(a \mid b) = \frac{P(b \mid a)P(a)}{P(b)} \]

This is often useful for assessing the probability of an underlying cause after an effect has been observed:

\[ P(\text{Cause} \mid \text{Effect}) = \frac{P(\text{Effect} \mid \text{Cause})P(\text{Cause})}{P(\text{Effect})} \]

**Bayes Rule in Machine Learning**

\( H \) is a class of hypotheses

\[ P(D \mid h) = \text{probability of data } D \text{ being generated under hypothesis } h \in H. \]

\[ P(h \mid D) = \text{probability that } h \text{ is correct, given that data } D \text{ were observed.} \]

Bayes’ Theorem:

\[ P(h \mid D)P(D) = P(D \mid h)P(h) \]

\[ P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)} \]

\( P(h) \) is called the prior.

**Weight Decay (5.2.2)**

Assume that small weights are more likely to occur than large weights, i.e.

\[ P(w) = \frac{1}{Z}e^{-\frac{1}{2} \sum w_j^2} \]

where \( Z \) is a normalizing constant. Then the cost function becomes:

\[ E = \frac{1}{2} \sum_i (z_i - t_i)^2 + \frac{\lambda}{2} \sum_j w_j^2 \]

This can prevent the weights from “saturating” to very high values.

Problem: need to determine \( \lambda \) from experience, or empirically.

**Momentum (8.3)**

If landscape is shaped like a “rain gutter”, weights will tend to oscillate without much improvement.

Solution: add a momentum factor

\[ \delta w \leftarrow \alpha \delta w + (1 - \alpha) \frac{\partial E}{\partial w} \]

\[ w \leftarrow w - \eta \delta w \]

Hopefully, this will dampen sideways oscillations but amplify downhill motion by \( \frac{1}{1-\alpha} \).
Limitations of Two-Layer Neural Networks

Some functions cannot be learned with a 2-layer sigmoidal network.

For example, this Twin Spirals problem cannot be learned with a 2-layer network, but it can be learned using a 3-layer network if we include shortcut connections between non-consecutive layers.

Vanishing / Exploding Gradients

Training by backpropagation in networks with many layers is difficult. When the weights are small, the differentials become smaller and smaller as we backpropagate through the layers, and end up having no effect. When the weights are large, the activations in the higher layers will saturate to extreme values. As a result, the gradients at those layers will become very small, and will not be propagated to the earlier layers. When the weights have intermediate values, the differentials will sometimes get multiplied many times in places where the transfer function is steep, causing them to blow up to large values.

Dropout (7.12)

Nodes are randomly chosen to not be used, with some fixed probability (usually, one half).

Hinton Diagrams

- used to visualize higher dimensions
- white = positive, black = negative
**Activation Functions (6.3)**

- Sigmoid
- Rectified Linear Unit (ReLU)
- Hyperbolic Tangent
- Scaled Exponential Linear Unit (SELU)

**Softmax (6.2.2)**

Consider a classification task with $N$ classes, and assume $z_j$ is the output of the unit corresponding to class $j$.

We assume the network’s estimate of the probability of each class $j$ is proportional to $\exp(z_j)$. Because the probabilities must add up to 1, we need to normalize by dividing by their sum:

$$\text{Prob}(i) = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$

$$\log \text{Prob}(i) = z_i - \log \sum_{j=1}^{N} \exp(z_j)$$

If the correct class is $i$, we can treat $-\log \text{Prob}(i)$ as our cost function. The first term pushes up the correct class $i$, while the second term mainly pushes down the incorrect class $j$ with the highest activation (if $j \neq i$).

**Convolutional Network Components**

- **convolution layers**: extract shift-invariant features from the previous layer
- **subsampling or pooling layers**: combine the activations of multiple units from the previous layer into one unit
- **fully connected layers**: collect spatially diffuse information
- **output layer**: choose between classes

**Convolutional Neural Networks**

$$Z_{j,k}^i = g(b^i + \sum_l \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} K_{l,m,n} V_{j+m+k+n}^i)$$

The same weights are applied to the next $M \times N$ block of inputs, to compute the next hidden unit in the convolution layer (“weight sharing”).
Weight Initialization

In order to have healthy forward and backward propagation, each term in the product must be approximately equal to 1. Any deviation from this could cause the activations to either vanish or saturate, and the differentials to either decay or explode exponentially.

\[ \text{Var}[z] \approx \left( \prod_{i=1}^{D} G_{0} n_{i}^{\text{in}} \text{Var}[w(i)] \right) \text{Var}[x] \]

\[ \text{Var}[\frac{\partial}{\partial x}] \approx \left( \prod_{i=1}^{D} G_{1} n_{i}^{\text{out}} \text{Var}[w(i)] \right) \text{Var}[\frac{\partial}{\partial z}] \]

We therefore choose the initial weights \{w_{jk}(i)\} in each layer (i) such that

\[ G_{1} n_{i}^{\text{out}} \text{Var}[w(i)] = 1 \]

Stride with Zero Padding

When combined with zero padding of width \( P \),

\( j \) takes on the values 0, \( s \), 2\( s \), \ldots, \((J + 2P - M)\)

\( k \) takes on the values 0, \( s \), 2\( s \), \ldots, \((K + 2P - N)\)

The next layer is \((1 + (J + 2P - M)/s)\) by \((1 + (K + 2P - N)/s)\)

Convolutional Filters

First Layer Second Layer Third Layer

Batch Normalization

We can normalize the activations \( x_{k}^{(i)} \) of node \( k \) in layer (i) relative to the mean and variance of those activations, calculated over a mini-batch of training items:

\[ x_{k}^{(i)} = \frac{x_{k}^{(i)} - \text{Mean}[x_{k}^{(i)}]}{\sqrt{\text{Var}[x_{k}^{(i)}]}} \]

These activations can then be shifted and re-scaled to

\[ y_{k}^{(i)} = \beta_{k}^{(i)} + \gamma_{k}^{(i)} x_{k}^{(i)} \]

\( \beta_{k}^{(i)}, \gamma_{k}^{(i)} \) are additional parameters, for each node, which are trained by backpropagation along with the other parameters (weights) in the network.

After training is complete, \text{Mean}[x_{k}^{(i)}] and \text{Var}[x_{k}^{(i)}] are either pre-computed on the entire training set, or updated using running averages.
Neural Texture Synthesis

1. pretrain CNN on ImageNet (VGG-19)
2. pass input texture through CNN; compute feature map $F^l_i$ for $i^{th}$ filter at spatial location $k$ in layer (depth) $l$
3. compute the Gram matrix for each pair of features
   \[ G^l_{ij} = \sum_k F^l_{ik}F^l_{jk} \]
4. feed (initially random) image into CNN
5. compute L2 distance between Gram matrices of original and new image
6. backprop to get gradient on image pixels
7. update image and go to step 5.

Residual Networks

Idea: Take any two consecutive stacked layers in a deep network and add a “skip” connection which bypasses these layers and is added to their output.

Dense Networks

Recently, good results have been achieved using networks with densely connected blocks, within which each layer is connected by shortcut connections to all the preceding layers.

Neural Style Transfer

content + style → new image
Neural Style Transfer

For Neural Style Transfer, we minimize a cost function which is

\[ E_{\text{total}} = \alpha E_{\text{content}} + \beta E_{\text{style}} \]

\[ = \frac{\alpha}{2} \sum_{i,k} \| F^l_{ik}(x) - F^l_{ik}(x_c) \|^2 + \frac{\beta}{4} \sum_{l=0}^{L} w_l N_l^2 M_l^2 \sum_{i,j} (G^l_{ij} - A^l_{ij})^2 \]

where

- \( x_c, x \) = content image, synthetic image
- \( F^l_{ik} \) = \( i \)th filter at position \( k \) in layer \( l \)
- \( N_l, M_l \) = number of filters, and size of feature maps, in layer \( l \)
- \( w_l \) = weighting factor for layer \( l \)
- \( G^l_{ij}, A^l_{ij} \) = Gram matrices for style image, and synthetic image

Recurrent Networks

- Processing Temporal Sequences
- Sliding Window
- Recurrent Network Architectures
- Hidden Unit Dynamics
- Long Short Term Memory

Sliding Window

The simplest way to feed temporal input to a neural network is the “sliding window” approach, first used in the NetTalk system (Sejnowski & Rosenberg, 1987).

Simple Recurrent Network (Elman, 1990)

- at each time step, hidden layer activations are copied to “context” layer
- hidden layer receives connections from input and context layers
- the inputs are fed one at a time to the network, it uses the context layer to “remember” whatever information is required for it to produce the correct output
Back Propagation Through Time

- We can “unroll” a recurrent architecture into an equivalent feedforward architecture, with shared weights.
- Applying backpropagation to the unrolled architecture is referred to as “backpropagation through time.”
- We can backpropagate just one timestep, or a fixed number of timesteps, or all the way back to the beginning of the sequence.

Oscillating Solution for \(a^n b^n\)

- Simple Recurrent Networks (SRNs) can learn medium-range dependencies but have difficulty learning long-range dependencies.
- Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) can learn long-range dependencies better than SRN.

Hidden Unit Dynamics for \(a^n b^n c^n\)

- SRN with 3 hidden units can learn to predict \(a^n b^n c^n\) by counting up and down simultaneously in different directions, thus producing a star shape.

Long Range Dependencies

- Simple Recurrent Networks (SRNs) can learn medium-range dependencies but have difficulty learning long-range dependencies.
- Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) can learn long-range dependencies better than SRN.
**Long Short Term Memory**

![Diagram of LSTM](image1)

**Gated Recurrent Unit**

![Diagram of GRU](image2)

**Co-occurrence Matrix (2-word window)**

<table>
<thead>
<tr>
<th>word</th>
<th>all</th>
<th>and</th>
<th>bought</th>
<th>caught</th>
<th>crooked</th>
<th>found</th>
<th>house</th>
<th>little</th>
<th>lived</th>
<th>man</th>
<th>mile</th>
<th>mouse</th>
<th>sixpence</th>
<th>stile</th>
<th>there</th>
<th>they</th>
<th>together</th>
<th>upon</th>
<th>walked</th>
<th>who</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bought</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caught</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>crooked</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>found</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>house</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>little</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lived</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>man</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mile</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mouse</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sixpence</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>stile</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>there</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>they</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>together</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>upon</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>walked</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>who</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Word Embeddings**

![Word Embedding Graph](image3)
### Singular Value Decomposition

Co-occurrence matrix $X$ can be decomposed as $X = USV^T$ where $U$, $V$ are unitary (all columns have unit length) and $S$ is diagonal.

Columns $1$ to $n$ of row $k$ of $U$ then provide an $n$-dimensional vector representing the $k^{th}$ word in the vocabulary.

SVD is computationally expensive, proportional to $L \times M^2$ if $L \geq M$. Can we do something similar with less computation, and incrementally?

### word2vec Skip-Gram Model

- try to predict the context words, given the center word
- this skip-gram model is similar to CBOW, except that in this case a single input word is used to predict multiple context words
- all context words share the same hidden-to-output weights

### Continuous Bag Of Words

- If several context words are each used independently to predict the center word, the hidden activation becomes a sum (or average) over all the context words
- Note the difference between this and NetTalk – in word2vec (CBOW) all context words share the same input-to-hidden weights

### Hierarchical Softmax

$$ [n' = \text{child}(n)] = \begin{cases} +1, & \text{if } n' \text{ is left child of node } n, \\ -1, & \text{otherwise}. \end{cases} $$

$$ \sigma(u) = 1/(1 - \exp(-u)) $$

$$ \text{prob}(w = w_i) = \prod_{j=1}^{L(u)-1} \sigma([n(w, j + 1) = \text{child}(n(w, j))]\nabla_{n(w, j)}^T h) $$
Negative Sampling

The idea of negative sampling is that we train the network to increase its estimation of the target word \( j^\ast \) and reduce its estimate not of all the words in the vocabulary but just a subset of them \( W_{\text{neg}} \), drawn from an appropriate distribution.

\[
E = -\log \sigma(v'_j^T h) - \sum_{j \in W_{\text{neg}}} \log \sigma(-v'_j^T h)
\]

This is a simplified version of Noise Constrastive Estimation (NCE). It is not guaranteed to produce a well-defined probability distribution, but in practice it does produce high-quality word embeddings.

Word Vector Arithmetic

![Country and Capital Vectors Projected by PCA](image)

Attention Mechanism

![Attention Mechanism](image)
Reinforcement Learning Framework

- An agent interacts with its environment.
- There is a set $S$ of states and a set $A$ of actions.
- At each time step $t$, the agent is in some state $s_t$.
  It must choose an action $a_t$, whereupon it goes into state $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r_t = R(s_t, a_t)$.
- Agent has a policy $\pi : S \rightarrow A$. We aim to find an optimal policy $\pi^*$ which maximizes the cumulative reward.
- In general, $\delta$, $R$, and $\pi$ can be multi-valued, with a random element, in which case we write them as probability distributions:
  $$\delta(s_{t+1} = s_t | a_t) \quad R(r_t = r_t | s_t, a_t) \quad \pi(a_t = a_t | s_t)$$

Models of optimality

- Is a fast nickel worth a slow dime?
  
  Finite horizon reward: $\sum_{i=0}^{h-1} r_{t+i}$
  Infinite discounted reward: $\sum_{i=0}^{\infty} \gamma^i r_{t+i}, \quad 0 \leq \gamma < 1$
  Average reward: $\lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h-1} r_{t+i}$

- Finite horizon reward is simple computationally.
- Infinite discounted reward is easier for proving theorems.
- Average reward is hard to deal with, because can’t sensibly choose between small reward soon and large reward very far in the future.

RL Approaches

- Value Function Learning
  - TD-Learning
  - Q-Learning
- Policy Learning
  - Hill Climbing
  - Policy Gradients
  - Evolutionary Strategy
- Actor-Critic
  - combination of Value and Policy learning

Exploration / Exploitation Tradeoff

Most of the time we should choose what we think is the best action. However, in order to ensure convergence to the optimal strategy, we must occasionally choose something different from our preferred action, e.g.

- choose a random action 5% of the time, or
- use Softmax (Boltzmann distribution) to choose the next action:
  $$P(a) = \frac{e^{R(a)/T}}{\sum_{b \in A} e^{R(b)/T}}$$
**Temporal Difference Learning**

Let’s first assume that \( R \) and \( \delta \) are deterministic. Then the (true) value \( V* (s) \) of the current state \( s \) should be equal to the immediate reward plus the discounted value of the next state

\[
V^*(s) = R(s,a) + \gamma V^*(\delta(s,a))
\]

We can turn this into an update rule for the estimated value, i.e.

\[
V(s_t) \leftarrow r_t + \gamma V(s_{t+1})
\]

If \( R \) and \( \delta \) are stochastic (multi-valued), it is not safe to simply replace \( V(s_t) \) with the expression on the right hand side. Instead, we move its value fractionally in this direction, proportional to a learning rate \( \eta \)

\[
V(s_t) \leftarrow V(s_t) + \eta [r_t + \gamma V(s_{t+1}) - V(s_t)]
\]

**Policy Gradients**

If \( r_{total} = +1 \) for a win and \(-1\) for a loss, we can simply multiply the log probability by \( r_{total} \). Differentials can be calculated using the gradient

\[
\nabla_{\theta} r_{total} \sum_{t=1}^{m} \log \pi_{\theta}(a_t|s_t) = r_{total} \sum_{t=1}^{m} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)
\]

The gradient of the log probability can be calculated nicely using Softmax.

If \( r_{total} \) takes some other range of values, we can replace it with \((r_{total} - b)\) where \( b \) is a fixed value, called the baseline.

**Q-Learning**

For a deterministic environment, \( \pi^*, Q^* \) and \( V^* \) are related by

\[
\pi^*(s) = \text{argmax}_a Q^*(s,a)
\]

\[
Q^*(s,a) = R(s,a) + \gamma V^*(\delta(s,a))
\]

\[
V^*(s) = \max_b Q^*(s,b)
\]

So

\[
Q^*(s,a) = R(s,a) + \gamma \max_b Q^*(\delta(s,a),b)
\]

This allows us to iteratively approximate \( Q \) by

\[
Q(s_t, a_t) \leftarrow r_t + \gamma \max_b Q(s_{t+1},b)
\]

If the environment is stochastic, we instead write

\[
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta [r_t + \gamma \max_b Q(s_{t+1},b) - Q(s_t, a_t)]
\]

**REINFORCE Algorithm**

We then get the following REINFORCE algorithm:

for each trial

run trial and collect states \( s_t \), actions \( a_t \), and reward \( r_{total} \)

for \( t = 1 \) to length(trial)

\[
\theta \leftarrow \theta + \eta (r_{total} - b) \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)
\]

end

This algorithm has successfully been applied, for example, to learn to play the game of Pong from raw image pixels.
Double Q-Learning

- if the same weights $w$ are used to select actions and evaluate actions, this can lead to a kind of confirmation bias
- could maintain two sets of weights $w$ and $\overline{w}$, with one used for selection and the other for evaluation (then swap their roles)
- in the context of Deep Q-Learning, a simpler approach is to use the current “online” version of $w$ for selection, and an older “target” version $\overline{w}$ for evaluation; we therefore minimize
  $$[r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)]^2$$
- a new version of $\overline{w}$ is periodically calculated from the distributed values of $w$, and this $\overline{w}$ is broadcast to all processors.

Deep Q-Learning with Experience Replay

- choose actions using current Q function ($\varepsilon$-greedy)
- build a database of experiences $(s_t, a_t, r_t, s_{t+1})$
- sample asynchronously from database and apply update, to minimize
  $$[r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)]^2$$
- removes temporal correlations by sampling from variety of game situations in random order
- makes it easier to parallelize the algorithm on multiple GPUs

Advantage Actor Critic

Recall that in the REINFORCE algorithm, a baseline $b$ could be subtracted from $r_{total}$
$$\theta \leftarrow \theta + \eta (r_{total} - b) \nabla \log \pi_\theta (a_t | s_t)$$
In the actor-critic framework, $r_{total}$ is replaced by $Q(s_t, a_t)$
$$\theta \leftarrow \theta + \eta \nabla \log \pi_\theta (a_t | s_t)$$
We can also subtract a baseline from $Q(s_t, a_t)$. This baseline must be independent of the action $a_t$, but it could be dependent on the state $s_t$. A good choice of baseline is the value function $V_\theta(s)$, in which case the $Q$ function is replaced by the advantage function
$$A_{\theta}(s, a) = Q(s, a) - V_\theta(s)$$
Asynchronous Advantage Actor Critic

- use policy network to choose actions
- learn a parameterized Value function $V_a(s)$ by TD-Learning
- estimate Q-value by n-step sample
  $$Q(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_a(s_{t+n})$$
- update policy by
  $$\theta \leftarrow \theta + \eta \nabla_{\theta} \log \pi(\theta | s_t) \left[ Q(s_t, a_t) - V_a(s_t) \right]$$
- update Value function my minimizing
  $$\left[ Q(s_t, a_t) - V_a(s_t) \right]^2$$

Hill Climbing

- Initialize “champ” policy $\theta_{\text{champ}} = 0$
- for each trial, generate “mutant” policy
  $$\theta_{\text{mutant}} = \theta_{\text{champ}} + \text{Gaussian noise (fixed } \sigma)$$
- champ and mutant play up to $n$ games, with same game initial conditions (i.e. same seed for generating dice rolls)
- if mutant does “better” than champ,
  $$\theta_{\text{champ}} \leftarrow \theta_{\text{champ}} + \alpha (\theta_{\text{mutant}} - \theta_{\text{champ}})$$
- “better” means the mutant must score higher than the champ in the first game, and at least as high as the champ in each subsequent game.

Evolved Behavior

- initialize mean $\mu = \{\mu_i\}_{1 \leq i \leq m}$ and standard deviation $\sigma = \{\sigma_i\}_{1 \leq i \leq m}$
- for each trial, collect $k$ samples from a Gaussian distribution
  $$\theta_i = \mu_i + \eta_i \sigma_i \quad \text{where} \quad \eta_i \sim \mathcal{N}(0, 1)$$
- sometimes include “mirrored” samples $\overline{\theta}_i = \mu_i - \eta_i \sigma_i$
- evaluate each sample $\theta$ to compute score or “fitness” $F(\theta)$
- update mean $\mu$ by
  $$\mu \leftarrow \mu + \alpha(F(\theta) - \overline{F})(\theta - \mu)$$
  $\alpha =$ learning rate, $\overline{F} =$ baseline
- sometimes, $\sigma$ is updated as well

Evolutionary / Variational Methods
Boltzmann Machine (20.1)

The Boltzmann Machine uses exactly the same energy function as the Hopfield network:

\[
E(x) = -\left( \sum_{i \neq j} w_{ij} x_i x_j + \sum_i b_i x_i \right)
\]

The Boltzmann Machine is very similar to the Hopfield Network, except that

- components (neurons) \( x_i \) take on the values 0, 1 instead of \(-1, +1\)
- used to generate new states rather than retrieving stored states
- update is not deterministic but stochastic, using the sigmoid

Restricted Boltzmann Machine (16.7)

If we allow visible-to-visible and hidden-to-hidden connections, the network takes too long to train. So we normally restrict the model by allowing only visible-to-hidden connections.

This is known as a Restricted Boltzmann Machine.
**Quick Contrastive Divergence**

It was noticed in the early 2000’s that the process can be sped up by taking just one additional sample instead of running for many iterations.

- \(v_0, h_0\) are used as positive sample, and \(v_1, h_1\) as negative sample
- this can be compared to the Negative Sampling that was used with word2vec – it is not guaranteed to approximate the true gradient, but it works well in practice

**Restricted Boltzmann Machine**

- inputs are binary vectors
- two-layer bi-directional neural network
  - visible layer \(v\)
  - hidden layer \(h\)
- no vis-to-vis or hidden-to-hidden connections
- all visible units connected to all hidden units

\[ E(v, h) = -(\sum_i b_i v_i + \sum_j c_j h_j + \sum_{i,j} v_i w_{ij} h_j) \]

- trained to maximize the expected log probability of the data

**Alternating Gibbs Sampling**

With the Restricted Boltzmann Machine, we can sample from the Boltzmann distribution as follows:

- choose \(v_0\) randomly
- then sample \(h_0\) from \(p(h|v_0)\)
- then sample \(v_1\) from \(p(v|h_0)\)
- then sample \(h_1\) from \(p(h|v_1)\)
- etc.

**Autoencoder Networks**

- output is trained to reproduce the input as closely as possible
- activations normally pass through a bottleneck, so the network is forced to compress the data in some way
- like the RBM, Autoencoders can be used to automatically extract abstract features from the input
Variational Autoencoder (20.10.3)

Instead of producing a single $z$ for each $x^{(i)}$, the encoder (with parameters $\phi$) can be made to produce a mean $\mu_{z|x^{(i)}}$ and standard deviation $\Sigma_{z|x^{(i)}}$.

This defines a conditional (Normal) probability distribution $q_\phi(z|x^{(i)})$.

We then train the system to maximize

$$
\mathbb{E}_{z \sim q_\phi(z|x^{(i)})}\left[ \log p_\theta(x^{(i)}|z) \right] - D_{KL} (q_\phi(z|x^{(i)}) || p(z))
$$

- the first term enforces that any sample $z$ drawn from the conditional distribution $q_\phi(z|x^{(i)})$ should, when fed to the decoder, produce something approximating $x^{(i)}$
- the second term encourages $q_\phi(z|x^{(i)})$ to approximate $p(z)$
- in practice, the distributions $q_\phi(z|x^{(i)})$ for various $x^{(i)}$ will occupy complementary regions within the overall distribution $p(z)$

Generative Models

Sometimes, as well as reproducing the training items $\{x^{(i)}\}$, we also want to be able to use the decoder to generate new items which are of a similar “style” to the training items.

In other words, we want to be able to choose latent variables $z$ from a standard Normal distribution $p(z)$, feed these values of $z$ to the decoder, and have it produce a new item $x$ which is somehow similar to the training items.

Generative models can be:
- explicit (Variational Autoencoders)
- implicit (Generative Adversarial Networks)
Generative Adversarial Networks

Generator (Artist) $G_\theta$ and Discriminator (Critic) $D_\psi$ are both Deep Convolutional Neural Networks.

Generator $G_\theta : z \mapsto x$, with parameters $\theta$, generates an image $x$ from latent variables $z$ (sampled from a Normal distribution).

Discriminator $D_\psi : x \mapsto D_\psi(x) \in (0, 1)$, with parameters $\psi$, takes an image $x$ and estimates the probability of the image being real.

Generator and Discriminator play a 2-player zero-sum game to compute:

$$\min_{\theta} \max_{\psi} \left( E_{x \sim p_{\text{data}}} \left[ \log D_\psi(x) \right] + E_{z \sim p_{\text{model}}} \left[ \log \left( 1 - D_\psi(G_\theta(z)) \right) \right] \right)$$

Discriminator tries to maximize the bracketed expression, Generator tries to minimize it.

Alternate between:

- Gradient ascent on Discriminator:
  $$\max_{\psi} \left( E_{x \sim p_{\text{data}}} \left[ \log D_\psi(x) \right] + E_{z \sim p_{\text{model}}} \left[ \log \left( 1 - D_\psi(G_\theta(z)) \right) \right] \right)$$

- Gradient descent on Generator, using:
  $$\min_{\theta} E_{z \sim p_{\text{model}}} \left[ \log \left( D_\psi(G_\theta(z)) \right) \right]$$

This formula puts too much emphasis on images that are correctly classified. Better to do gradient ascent on Generator, using:

$$\max_{\theta} E_{z \sim p_{\text{model}}} \left[ \log \left( D_\psi(G_\theta(z)) \right) \right]$$

This puts more emphasis on the images that are wrongly classified.

GAN Generated Images
Related Courses

- COMP3411/9414 Artificial Intelligence
- COMP9417 Machine Learning and Data Mining
- COMP4418 Knowledge Representation and Reasoning
- COMP3431 Robotic Software Architecture
- COMP9517 Machine Vision
- 4th Year Thesis topics

UNSW myExperience Survey

Please remember to fill in the UNSW myExperience Survey.

Possible 4th Year Projects

- deep learning combined with evolution
- deep learning for signal processing
- self-normalizing activation functions
- CNN with internal weight symmetries
- adversarial coevolution of HERCL programs
- other topics in deep learning, evolution, games

Neural Networks and Deep Learning

QUESTIONS?
Neural Networks and Deep Learning

GOOD LUCK!