Today’s quiz

• **Q1.** What is the role of the hidden layers in a neural network?

• **Q2.** A 3-input neuron is trained to output a zero when the input is 110 and a one when the input is 111. After generalisation, the output will be zero when and only when the input is:
  
  (a) 000 or 110 or 011 or 101
  
  (b) 010 or 100 or 110 or 101
  
  (c) 000 or 010 or 110 or 100

• **Q3.** Consider the cross-entropy loss function:

\[
Err(w) = - \left[ \sum_{i=1}^{n} y_{i} \log(h(x_{i})) + (1-y_{i}) \log(1-h(x_{i})) \right]
\]

Derive the backpropagation update equation for the output node of a feed-forward neural network assuming this loss function.
Feed-forward neural networks

Notation:

- $w_{ji}$ denotes weight on connection from unit $i$ to unit $j$.
- By convention, $x_{j0} = 1$, $\forall j$.
- Output of unit $j$, denoted $o_j$, is computed using a sigmoid:

  $o_j = \sigma(w_j \cdot x_j)$

  where
  - $w_j$ is vector of weights entering unit $j$.
  - $x_j$ is vector of inputs to unit $j$.
- By definition, $x_{ji} = o_i$.

Weights are trained by the Backpropagation algorithm.

---

Stochastic gradient descent

- Initialize all weights to small random numbers.
- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$.
  - For the output unit, compute the correction:
    $\delta_{N+H+1} = o(1-o)(y-o)$
  - For each hidden unit $h$, compute its share of the correction:
    $\delta_h = o_h(1-o_h)w_{N+H+1,h}\delta_{N+H+1}$
  - Update each network weight:
    $w_{h,i} = w_{h,i} + \alpha_h \delta_h x_{h,i}$
A family of sigmoid functions

\[ \sigma(z) = \tanh(z) \]
\[ \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
\[ \frac{\partial \sigma(z)}{\partial z} = 1 - \sigma(z)^2 \]

Encoding the input: Discrete inputs

- Discrete inputs with \( k \) possible values are often encoded using a **1-hot** or **1-of-\( k \)** encoding:
  - \( k \) input bits are associated with the variable (one for each possible value).
  - For any instance, all bits are 0 except the one corresponding to the value found in the data, which is set to 1.
  - If the value is missing, all inputs are set to 0.
Encoding the input: Real-valued inputs

- Important to scale the inputs, so they have a common, reasonable range.
- Standard transformation: normalize the data
  - To get mean=0, variance=1, subtract the mean and divide by the standard deviation.
  - Works well if the data is roughly normal, but bad if we have outliers.

- Alternatives:
  - 1-to-n encoding: discretize the variable into a given number of intervals $n$.
  - Thermometer encoding: like 1-to-n but if the variable falls in the $i$-th interval, all bits 1..i are set to 1.
  - The thermometer encoding is usually better than 1-to-n encoding.

Encoding the output

- Multi-class domains:
  - Use a network with several output units: one per class.
  - Compared to training multiple 1-vs-all classifiers, this allows shared weights at the hidden layers.

- Regression tasks:
  - Use a network with several output sigmoid units, corresponding to encoding of different output ranges of output value.
  - Use an output unit without a sigmoid function (i.e. just the weighted linear combination) to get full range of output values.
Learning the identity function

• Also called auto-regression.
• This a case of unsupervised learning.

![Input vs Output Table]

Learning the identity function

• Neural network structure:

![Neural Network Diagram]

• Learned hidden layer weights:
  (capture an alternate encoding of the data.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Layer</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000000</td>
<td>.99 .97 .99</td>
<td>.99 .97 .99</td>
</tr>
<tr>
<td>00100000</td>
<td>.98 .97 .98</td>
<td>.98 .97 .98</td>
</tr>
<tr>
<td>00010000</td>
<td>.99 .97 .99</td>
<td>.99 .97 .99</td>
</tr>
<tr>
<td>00001000</td>
<td>.99 .97 .99</td>
<td>.99 .97 .99</td>
</tr>
<tr>
<td>00000001</td>
<td>.99 .97 .99</td>
<td>.99 .97 .99</td>
</tr>
</tbody>
</table>
Network architecture

- Overfitting occurs if there are too many parameters compared to the amount of data available.
- Choosing the number of hidden units
  - Too few hidden units do not allow the concept to be learned.
  - Too many lead to slow learning and overfitting.
  - If the $m$ inputs are binary, $\log m$ is a good heuristic choice.
- Choosing the number of layers
  - Always start with one hidden layer.
  - Don’t go beyond 2 hidden layers, unless the task structure suggests something different.

Convergence of backpropagation

- Backpropagation = gradient descent over all parameters in network.
- If the learning rate is appropriate, the algorithm is guaranteed to converge to a local minimum of the cost function.
  - NOT the global minimum.
  - Can be much WORSE than global minimum.
  - There can be MANY local minimum.
  - Solution: random restarts = train multiple nets with different initial weights.
  - In practice, the solution found is often very good (do a few parallel restarts).
- Training can take thousands of iterations - VERY SLOW! But using network after training is very fast.
- Can we find solution faster (i.e. in fewer iterations)?
Overtraining

- Traditional overfitting is concerned with the number of parameters vs. the number of instances.
- In neural networks: related phenomenon called overtraining occurs when weights take on large magnitudes, pushing sigmoids into saturation.
  - As learning progresses, the network has more active parameters.
- Use a validation set to decide when to stop training.
  >> Training horizon is a hyper-parameter.
- Regularization is also effective.

Regularization in neural networks

- Incorporate an L2 penalty: $J(w) = 0.5(y-h_w(x))^2 + 0.5\lambda w^T w$
  - Select $\lambda$ with cross-validation.
- Can also use different penalties $\lambda_1, \lambda_2$ for each layer.
  - Can be interpreted as a Bayesian prior over weights.
Choosing the learning rate

• Backprop is very sensitive to the choice of learning rate.
  – Too large ⇒ divergence.
  – Too small ⇒ VERY slow learning.
  – The learning rate also influences the ability to escape local optima.

• Very often, different learning rates are used for units in different layers. Hard to tune by hand!

• **Heuristic**: Track performance on validation set, when it stabilizes, divide learning rate by 2.

Optimization method: Adagrad

• Calculate **adaptive** learning rate per parameter.

• Intuition: Adapt learning rate depending on previous updates to that parameter.
  – Learn slowly for frequent features.
  – Learn faster for rare but informative features.

• Can add regularization term.

See: Duchi, Hazan, Singer (2011) *Adaptive subgradient methods for online learning and stochastic optimization*. JMLR.
Adding momentum

• On the t-th training sample, instead of the update:
  \[ \Delta w_{ij} \leftarrow \alpha_{ij} \delta_j x_{ij}. \]
  We do:  \[ \Delta w_{ij}(t) \leftarrow \alpha_{ij} \delta_j x_{ij} + \beta \Delta w_{ij}(t-1) \]
  The second term is called momentum.

Advantages:
  – Easy to pass small local minima.
  – Keeps the weights moving in areas where the error is flat.
  – Increases the speed where the gradient stays unchanged.

Disadvantages:
  – With too much momentum, it can get out of a global maximum!
  – One more parameter to tune, and more chances of divergence.

Optimization: Repeated Line Searches

• Given the optimization problem:  \( w^* = \text{argmax}_w f(w) \)
  – Think of \( f(w) \) as your neural network.
• Consider an update:  \( w_j = \text{argmin}_w f(w_1, w_2, \ldots, w_{j-1}, w_j, w_{j+1}, \ldots, w_m) \)
  – Optimizing a single dimension \( w_j \) holding the rest constant.
• Repeat for all weights.
• Then, repeat again until all weights are simultaneously optimal.

• This works well when
  \[ \frac{\partial^2 f}{\partial w_i \partial w_k} = 0 \text{ for } i \neq j \]
Optimization: Conjugate gradient descent

- Condition is not always true:
  \[ \frac{\partial^2 f}{\partial w_i \partial w_k} = 0 \quad \text{for} \quad i \neq j \]
- Find a coordinate system where it is satisfied?
- Yes! The eigenvectors of the Hessian matrix:
  \[ H_{j,k} = \frac{\partial^2 f}{\partial w_j \partial w_k} \]
- Apply repeated line search in this coordinate system.

Optimization: Second-order method

- Newton-Raphson method looks at both the first and second-order derivatives of the error function:
  \[ G_j = \frac{\partial f}{\partial w_j} \]
  \[ H_{j,k} = \frac{\partial^2 f}{\partial w_j \partial w_k} \]
- Approximate first-order derivative:
  \[ C(w_{t+1}) \approx G + H(w_{t+1} - w_t) \]
- Solve for \( G(w_{t+1}) = 0 \):
  \[ w_{t+1} = w_t - H^{-1}G \]

More application-specific tricks

• If there is too little data, it can be perturbed by random noise; this helps escape local minima and gives more robust results.
  – In image classification and pattern recognition tasks, extra data can be generated, e.g., by applying transformations that make sense.

• Weight sharing can be used to indicate parameters that should have the same value based on prior knowledge.
  – Each update is computed separately using backpropagation, then the tied parameters are updated with an average.

Several applications

• Speech recognition and synthesis.
• Natural language understanding.
• Image classification, digit recognition.
• Financial prediction.
• Game playing strategies.
• Robotics.
• .....
When to consider using NNs

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input).
- Output is discrete or real valued, or a vector of values.
- Possibly noisy data.
- Training time is not important.
- Form of target function is unknown.
- Human readability of result is not important.
- The computation of the output based on the input has to be fast.

What you should know

- Training by backpropagation.
- Overfitting (and how to avoid it).
- When to use NNs.
- Some strategies for successful application of NNs.