Deep Neural Networks (1)
Hidden layers; Back-propagation

Steve Renals

Machine Learning Practical — MLP Lecture 3
4 October 2017 / 9 October 2017
Recap: Softmax single layer network

MLP Lecture 3
Deep Neural Networks (1)
Single-layer network, 1 output, 2 inputs

$$+$$

$$x_1 \quad x_2$$
Geometric interpretation

Single-layer network, 1 output, 2 inputs

$$y(w; x) = 0$$

$$x_2$$

$$x_1$$

$$w$$

$$b = \frac{-b}{||w||}$$
Single layer network

Single-layer network, 3 outputs, 2 inputs

![Diagram of a single-layer network with 2 inputs and 3 outputs. The inputs $x_1$ and $x_2$ are connected to the outputs, illustrating the network's structure.]
Example data (three classes)
Classification regions with single-layer network

Single-layer networks are limited to linear classification boundaries
Single layer network trained on MNIST Digits

Output weights define a “template” for each class
Hinton Diagrams

Visualise the weights for class $k$

400 (20x20) inputs
Hinton diagram for single layer network trained on MNIST

- Weights for each class act as a “discriminative template”
- Inner product of class weights and input to measure closeness to each template
- Classify to the closest template (maximum value output)
Multi-Layer Networks
From templates to features

- Good classification needs to cope with the variability of real data: scale, skew, rotation, translation, ....
- Very difficult to do with a single template per class
- Could have multiple templates per task... this will work, but we can do better

Use features rather than templates

(images from: Nielsen, chapter 1)
Incorporating features in neural network architecture

Layered processing: inputs - features - classification

How to obtain features? - learning!
Incorporating features in neural network architecture

![Diagram showing neural network architecture with numbers and symbols]
Multi-layer network

\[ y_k = \text{softmax} \left( \sum_{r=1}^{H} w_{kr}^{(2)} h_r^{(1)} + b_k \right) \]

\[ h_j^{(1)} = \text{sigmoid} \left( \sum_{s=1}^{d} w_{js}^{(1)} x_s + b_j \right) \]
Multi-layer network for MNIST

Hidden units make training the weights more complicated, since the hidden units affect the error function indirectly via all the outputs.

The credit assignment problem:
- What is the “error” of a hidden unit?
- How important is input-hidden weight $w_{ji}^{(1)}$ to output unit $k$?
- What is the gradient of the error with respect to each weight?

Solution: back-propagation of error (backprop)

Backprop enables the gradients to be computed. These gradients are used by gradient descent to train the weights.
Training output weights

Outputs

Hidden units

\[ y_1 \]

\[ y_t \]

\[ y_K \]

\[ g_1^{(2)} \]

\[ g_t^{(2)} \]

\[ g_K^{(2)} \]

\[ w_{1j}^{(2)} \]

\[ w_{tj}^{(2)} \]

\[ w_{Kj}^{(2)} \]

\[ \frac{\partial E^n}{\partial w_{Kj}^{(2)}} = g_k^{(2)} h_j^{(1)} \]

\[ h_j \]

\[ w_{ji}^{(1)} \]

\[ x_i \]

MLP Lecture 3

Deep Neural Networks (1)
Training MLPs: Error function and required gradients

- **Cross-entropy error function:**

\[
E^n = - \sum_{k=1}^{C} t^n_k \ln y^n_k
\]

- **Required gradients:**

\[
\frac{\partial E^n}{\partial w_{kj}^{(2)}}, \quad \frac{\partial E^n}{\partial w_{ji}^{(1)}}, \quad \frac{\partial E^n}{\partial b_k^{(2)}}, \quad \frac{\partial E^n}{\partial b_j^{(1)}}
\]

- **Gradient for hidden-to-output weights** similar to single-layer network:

\[
\frac{\partial E^n}{\partial w_{kj}^{(2)}} = \frac{\partial E^n}{\partial a_k^{(2)}} \cdot \frac{\partial a_k^{(2)}}{\partial w_{kj}} = \left( \sum_{c=1}^{C} \frac{\partial E^n}{\partial y_c} \cdot \frac{\partial y_c}{\partial a_k^{(2)}} \right) \cdot \frac{\partial a_k^{(2)}}{\partial w_{kj}}
\]

\[
= (y_k - t_k) h_j^{(1)} g_k^{(2)}
\]
Back-propagation of error: hidden unit error signal

$$g_j^{(1)} = \left( \sum_{\ell} g_{\ell j}^{(2)} w_{\ell j} \right) h_j (1 - h_j)$$

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = g_j^{(1)} x_i$$
Training MLPs: Input-to-hidden weights

\[
\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \frac{\partial E^n}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}
\]

To compute \( g_j^{(1)} = \frac{\partial E^n}{\partial a_j^{(1)}} \), the error signal for hidden unit \( j \), we must sum over all the output units’ contributions to \( g_j^{(1)} \):

\[
g_j^{(1)} = \sum_{c=1}^{K} \frac{\partial E^n}{\partial a_c^{(2)}} \cdot \frac{\partial a_c^{(2)}}{\partial a_j^{(1)}} = \left( \sum_{c=1}^{K} g_c^{(2)} \cdot \frac{\partial a_c^{(2)}}{\partial h_j^{(1)}} \right) \cdot \frac{\partial h_j^{(1)}}{\partial a_j^{(1)}}
\]

\[
= \left( \sum_{c=1}^{K} g_c^{(2)} w_c^{(2)} \right) h_j^{(1)} (1 - h_j^{(1)})
\]
Training MLPs: Gradients

\[
\frac{\partial E^n}{\partial w_{kj}^{(2)}} = (y_k - t_k) \cdot h_j^{(1)}
\]

\[
\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \left( \sum_{c=1}^{k} g_c^{(2)} w_{cj}^{(2)} \right) h_j^{(1)} (1 - h_j^{(1)}) \cdot x_i
\]

- Exercise: write down expressions for the gradients w.r.t. the biases

\[
\frac{\partial E^n}{\partial b_k^{(2)}} \quad \frac{\partial E^n}{\partial b_j^{(1)}}
\]
Back-propagation of error: hidden unit error signal

\[ g_j^{(1)} = \left( \sum_{\ell} g_{\ell j}^{(2)} w_{\ell j} \right) h_j (1 - h_j) \]

\[ \frac{\partial E_n}{\partial w_{ji}^{(1)}} = g_j^{(1)} x_i \]
Back-propagation of error

- The back-propagation of error algorithm is summarised as follows:
  1. Apply an input vectors from the training set, \( x \), to the network and forward propagate to obtain the output vector \( y \)
  2. Using the target vector \( t \) compute the error \( E^n \)
  3. Evaluate the error gradients \( g_k^{(2)} \) for each output unit
  4. Evaluate the error gradients \( g_j^{(1)} \) for each hidden unit using back-propagation of error
  5. Evaluate the derivatives for each training pattern

- Back-propagation can be extended to multiple hidden layers, in each case computing the \( g^{(\ell)} \)s for the current layer as a weighted sum of the \( g^{(\ell+1)} \)s of the next layer
Training with multiple hidden layers

\[
g^{(2)}_k = \left( \sum_m g^{(3)}_m w_{mk} \right) h^{(2)}_k (1 - h^{(2)}_k)
\]

\[
\frac{\partial E^n}{\partial w^{(2)}_{kj}} = g^{(2)}_j h^{(2)}_j
\]
Are there alternatives to Sigmoid Hidden Units?
Sigmoid function

Logistic sigmoid activation function $g(a) = \frac{1}{1 + \exp(-a)}$
Sigmoid Hidden Units

- Compress unbounded inputs to (0,1), saturating high magnitudes to 1
- Interpretable as the probability of a feature defined by their weight vector
- Interpretable as the (normalised) firing rate of a neuron

However...

- Saturation causes gradients to approach 0: If the output of a sigmoid unit is \( h \), then the gradient is \( h(1 - h) \) which approaches 0 as \( h \) saturates to 0 or 1 - and hence the gradients it multiplies into approach 0. Very small gradients result in very small parameter changes, so learning becomes very slow.
- Outputs are not centred at 0: The output of a sigmoid layer will have mean > 0. This is numerically undesirable.
tanh

tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}

sigmoid(x) = \frac{1 + \tanh(x/2)}{2}

Derivative:
\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)
tanh hidden units

- tanh has same shape as sigmoid but has output range $\pm 1$
- Results about approximation capability of sigmoid networks also apply to tanh networks
- Possible reason to prefer tanh over sigmoid: allowing units to be positive or negative allows gradient for weights into a hidden unit to have a different sign
- Saturation still a problem
Rectified Linear Unit – ReLU

relu(x) = max(0, x)

Derivative: \[
\frac{d}{dx} \text{relu}(x) = \begin{cases} 
0 & \text{if } x \leq 0 \\
1 & \text{if } x > 0
\end{cases}
\]
ReLU hidden units

- Similar approximation results to tanh and sigmoid hidden units
- Empirical results for speech and vision show consistent improvements using relu over sigmoid or tanh
- Unlike tanh or sigmoid there is no positive saturation – saturation results in very small derivatives (and hence slower learning)
- Negative input to relu results in zero gradient (and hence no learning)
- Relu is computationally efficient: \( \max(0, x) \)
- Relu units can “die” (i.e. respond with 0 to everything)
- Relu units can be very sensitive to the learning rate
Summary

- Understanding what single-layer networks compute
- How multi-layer networks allow feature computation
- Training multi-layer networks using back-propagation of error
- Tanh and ReLU activation functions
- Multi-layer networks are also referred to as *deep neural networks* or *multi-layer perceptrons*

**Reading:**
- Nielsen, chapter 2
- Goodfellow, sections 6.3, 6.4, 6.5
- Bishop, sections 3.1, 3.2, and chapter 4