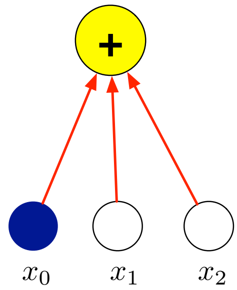
Multi-layer networks

Steve Renals

Machine Learning Practical — MLP Lecture 3
7 October 2015

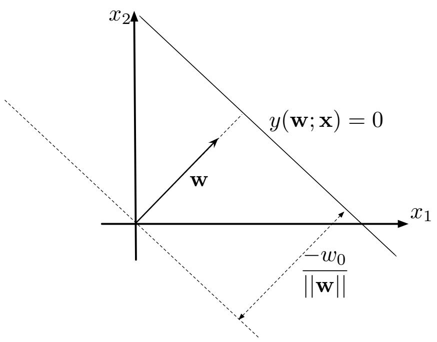
Single layer network

Single-layer network, 1 output, 2 inputs + bias



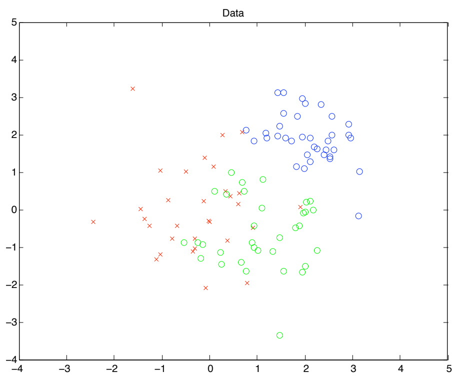
Geometric interpretation

Single-layer network, 1 output, 2 inputs + bias

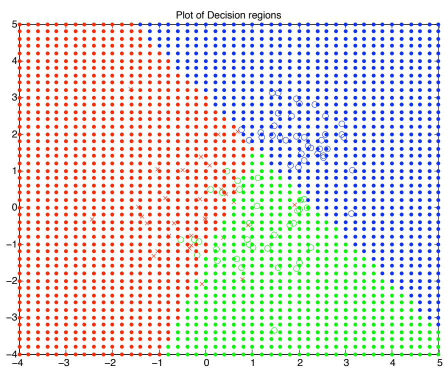


Bishop, sec 3.1

Example data (three classes)

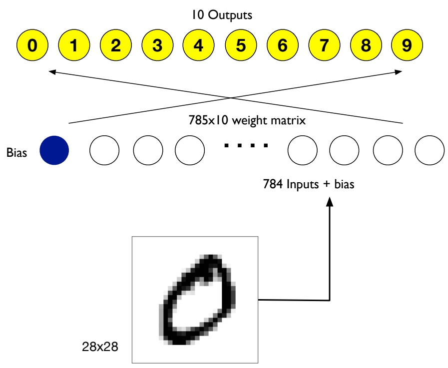


Classification regions with single-layer network



Single-layer networks are limited to linear classification boundaries

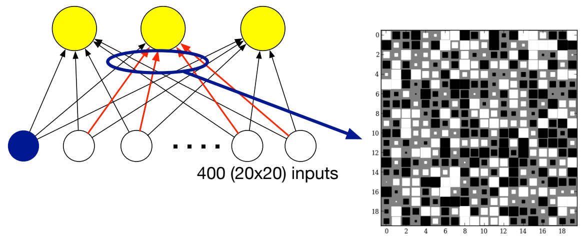
Single layer network trained on MNIST Digits



Output weights define a "template" for each class

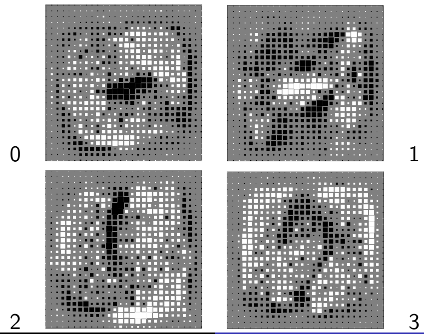
Hinton Diagrams

Visualise the weights for class k

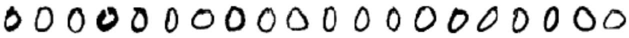


Hinton diagram for single layer network trained on MNIST

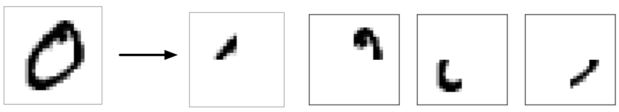
- Weights for each class act as a "discriminative template"
- Inner product of class weights and input to measure closeness to each template
- Classify to the closest template (maximum value output)



From templates to features



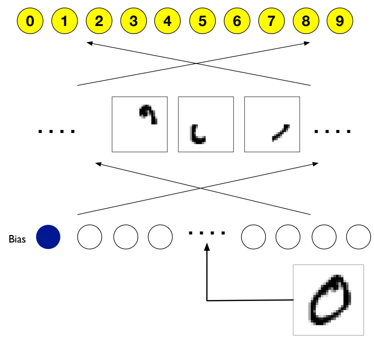
- Good classification needs to cope with the variability of real data: scale, skew, rotation, translation, ...
- Very difficult to do with a single template per class
- Could have multiple templates per task... this will work, but we can do better
- Use features rather than templates



(images from: Michael Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/chap1.html)

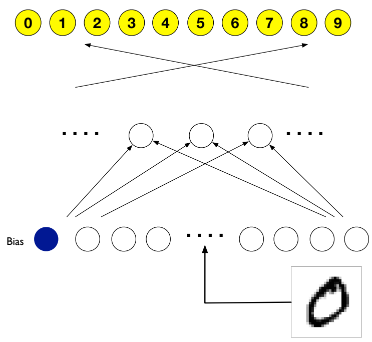
Incorporating features in neural network architecture

- Layered processing: inputs - features - classification
- How to obtain features - learning!



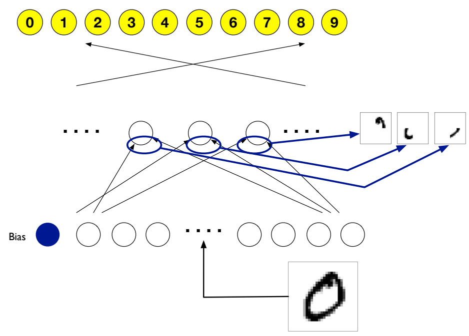
Incorporating features in neural network architecture

- Layered processing: inputs - features - classification
- How to obtain features - learning!

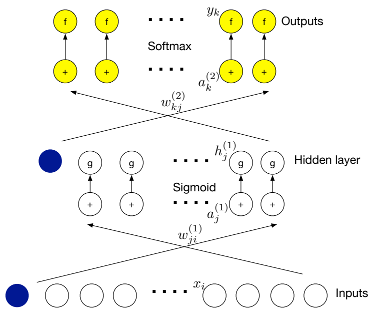


Incorporating features in neural network architecture

- Layered processing: inputs - features - classification
- How to obtain features - learning!

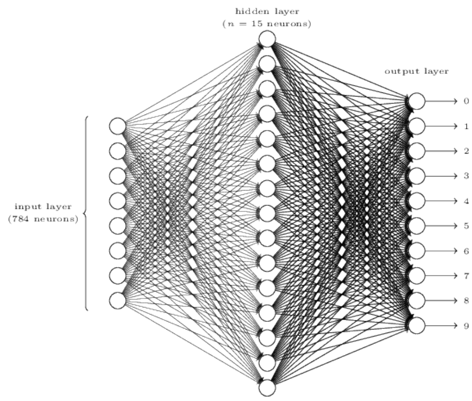


Multi-layer network



$$y_k = \text{softmax}\left(\sum_{r=0}^{H} w_{kr}^{(2)} h_r^{(1)}\right) \quad ; \quad h_j^{(1)} = \text{sigmoid}\left(\sum_{s=0}^{d} w_{js}^{(1)} x_s\right)$$

Multi-layer network for MNIST



(image from: Michael Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/chap1.html)

Training MLPs: Credit assignment

- 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?
- Solution: Gradient descent – requires derivatives of the error with respect to each weight
- Algorithm: *back-propagation of error* (backprop)
- Backprop gives a way to compute the derivatives. These derivatives are used by an optimisation algorithm (e.g. gradient descent) to train the weights.

Training MLPs: Error function and required gradients

- Cross-entropy error function:

$$E^n = -\sum_{k=1}^{C} t_k^n \ln y_k^n$$

- Required gradients:

$$\frac{\partial E^n}{\partial w_{kj}^{(2)}} \quad ; \quad \frac{\partial E^n}{\partial w_{ji}^{(1)}}$$

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

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

$$= \underbrace{(y_k - t_k)}_{\delta_k^{(2)}} h_j^{(1)}$$

Training MLPs: Input-to-hidden weights

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \underbrace{\frac{\partial E^n}{\partial a_j^{(1)}}}_{\delta_j^{(1)}} \cdot \underbrace{\frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}}_{x_i}$$

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

$$\delta_j^{(1)} = \sum_{c=1}^{K} \frac{\partial E^n}{\partial a_c^{(2)}} \cdot \frac{\partial a_c^{(2)}}{\partial a_j^{(1)}}$$

$$= \left(\sum_{c=1}^{K} \delta_c^{(2)} \cdot \frac{\partial a_c^{(2)}}{\partial h_j^{(1)}}\right) \cdot \frac{\partial h_j^{(1)}}{\partial a_j^{(1)}}$$

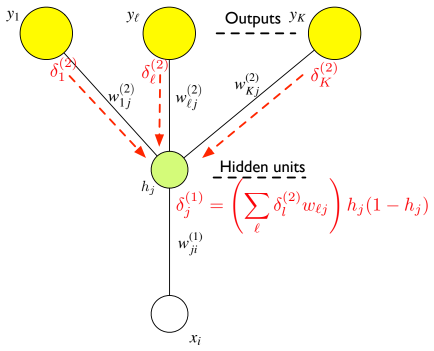
$$= \left(\sum_{c=1}^{K} \delta_c^{(2)} w_{cj}^{(2)}\right) h_j^{(1)}(1 - h_j^{(1)})$$

Training MLPs: Gradients

$$\frac{\partial E^n}{\partial w_{kj}^{(2)}} = \underbrace{(y_k - t_k)}_{\delta_k^{(2)}} \cdot h_j^{(1)}$$

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \underbrace{\left(\sum_{c=1}^{k} \delta_c^{(2)} w_{cj}^{(2)}\right) h_j^{(1)}(1 - h_j^{(1)})}_{\delta_j^{(1)}} \cdot x_i$$

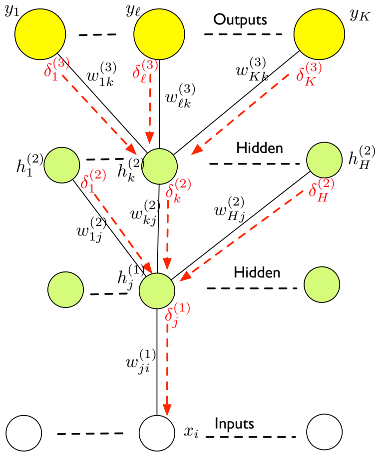
Back-propagation of error: hidden unit error signal



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 signals $\delta_k^{(2)}$ for each output unit
  4. Evaluate the error signals $\delta_j^{(1)}$ for each hidden unit using back-propagation of error
  5. Evaluate the derivatives for each training pattern, summing to obtain the overall derivatives
- Back-propagation can be extended to multiple hidden layers, in each case computing the $\delta^{(\ell)}$s for the current layer as a weighted sum of the $\delta^{(\ell+1)}$s of the next layer

Training with multiple hidden layers



Summary

- Understanding what single-layer networks compute
- How multi-layer networks allow feature computation
- Training multi-layer networks using back-propagation of error
- Reading:
  Michael Nielsen, chapter 1 of *Neural Networks and Deep Learning*
  `http://neuralnetworksanddeeplearning.com/chap1.html`
  Chris Bishop, Sections 3.1, 3.2, and Chapter 4 of *Neural Networks for Pattern Recognition*