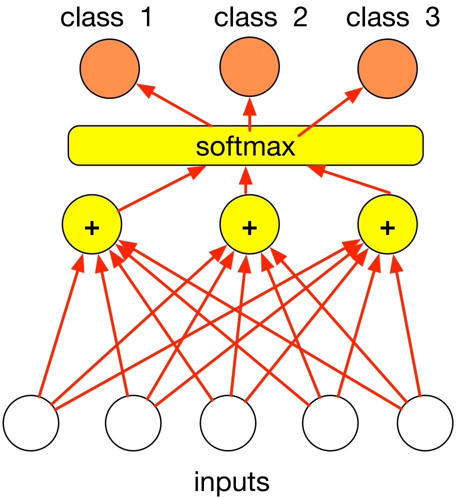
Deep Neural Networks (1)
Hidden layers; Back-propagation

Steve Renals

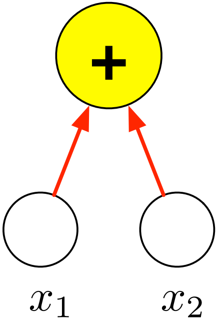
Machine Learning Practical — MLP Lecture 3
4 October 2017 / 9 October 2017

Recap: Softmax single layer network



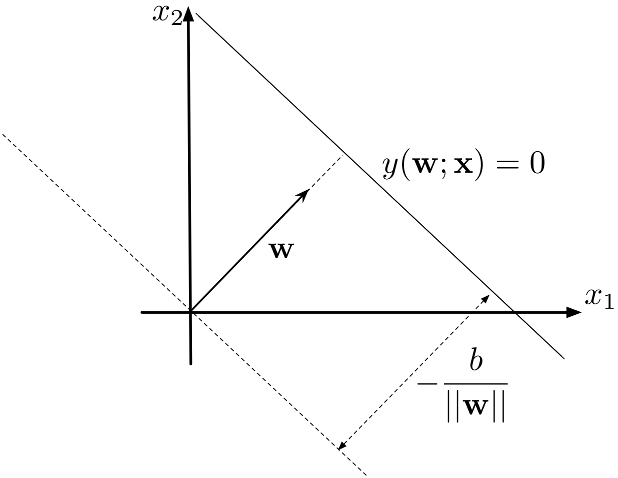
Single layer network

Single-layer network, 1 output, 2 inputs



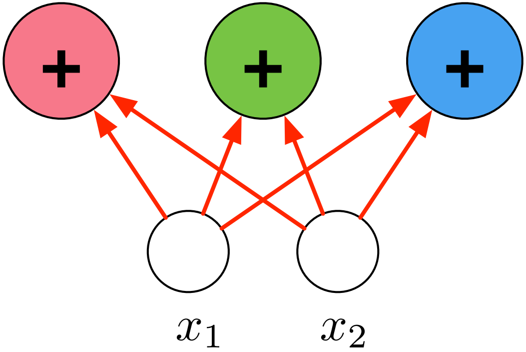
Geometric interpretation

Single-layer network, 1 output, 2 inputs

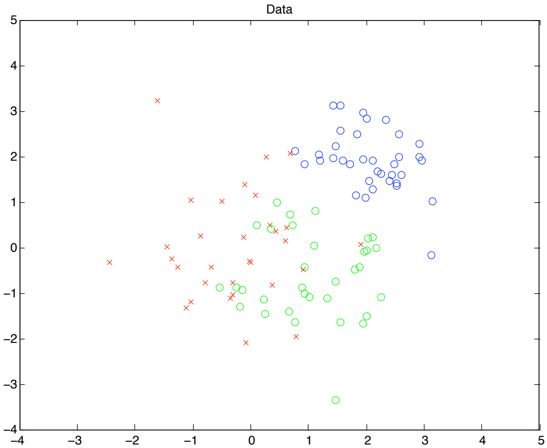


Single layer network

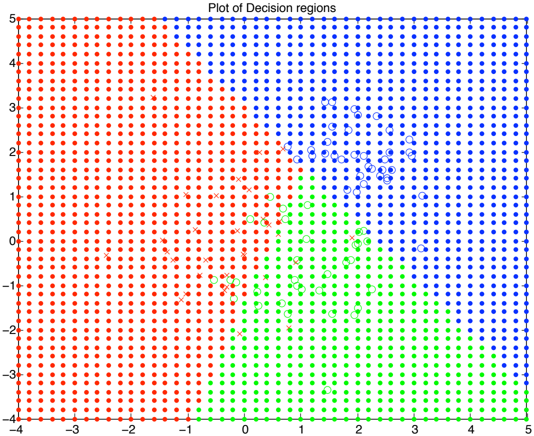
Single-layer network, 3 outputs, 2 inputs



Example data (three classes)

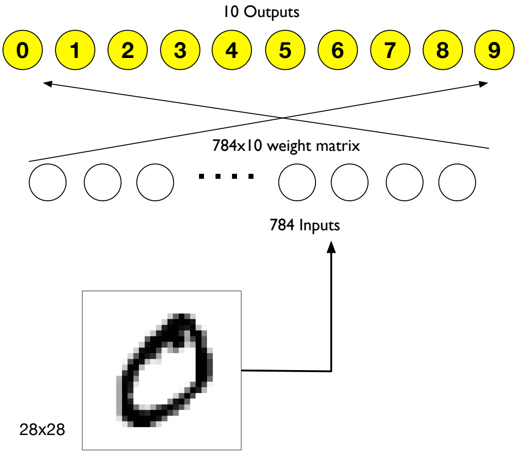


Classification regions with single-layer network



Single-layer networks are limited to linear classification boundaries

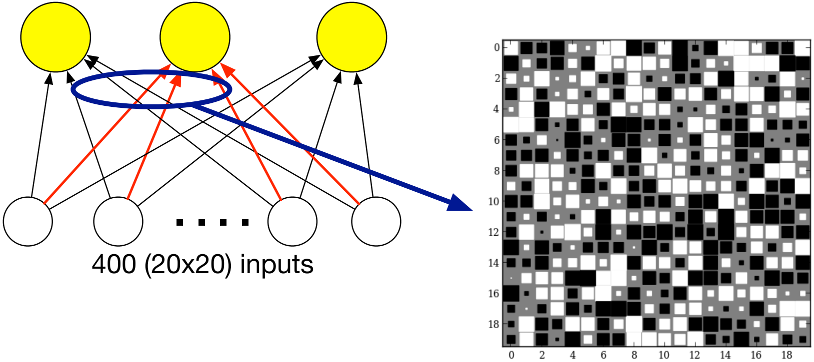
Single layer network trained on MNIST Digits



Output weights define a "template" for each class

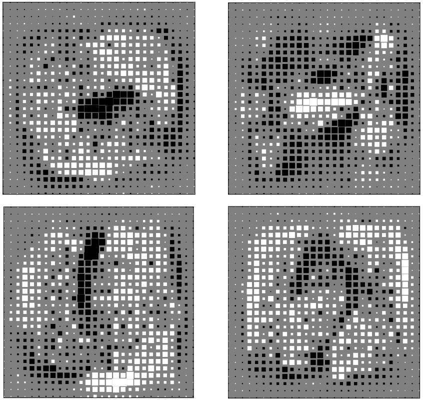
Hinton Diagrams

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)



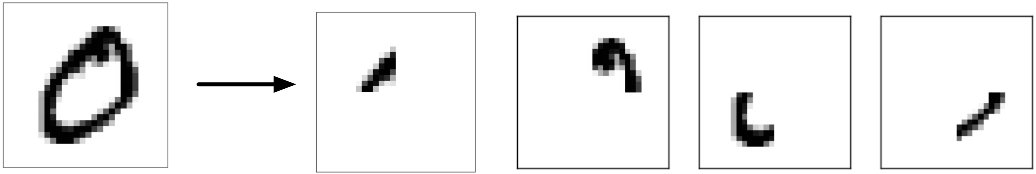
0 1 2 3

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

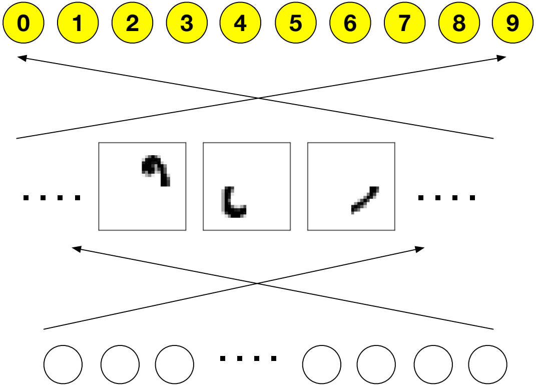
Use features rather than templates



(images from: Nielsen, chapter 1)

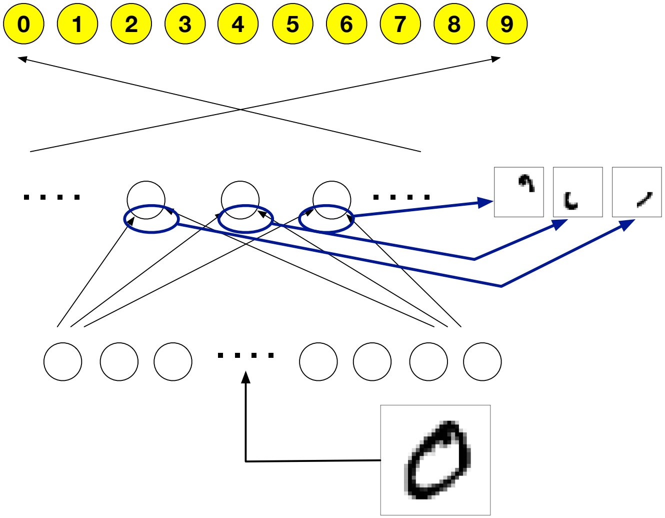
Incorporating features in neural network architecture

Layered processing: inputs - features - classification

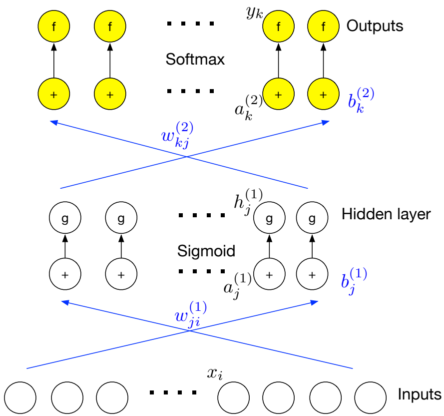


How to obtain features? - learning!

Incorporating features in neural network architecture

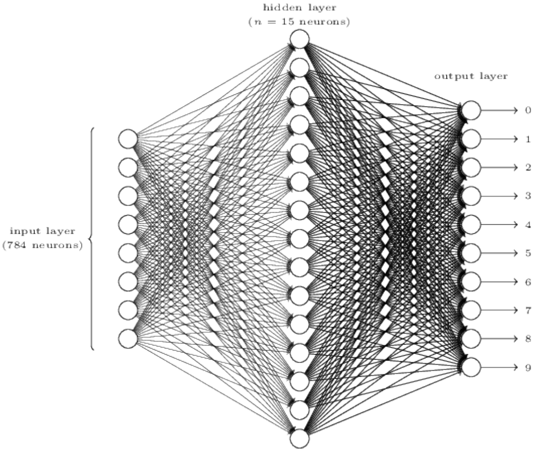


Multi-layer network



$$y_k = \mathrm{softmax}\left(\sum_{r=1}^{H} w_{kr}^{(2)} h_r^{(1)} + b_k\right) \qquad h_j^{(1)} = \mathrm{sigmoid}\left(\sum_{s=1}^{d} w_{js}^{(1)} x_s + b_j\right)$$

Multi-layer network for MNIST

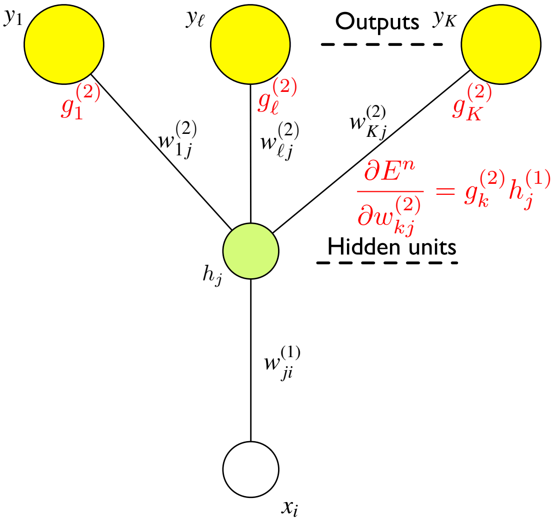


(image from: Michael Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/chap1.html)

Training multi-layer networks: 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?
  - *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



Training MLPs: Error function and required gradients

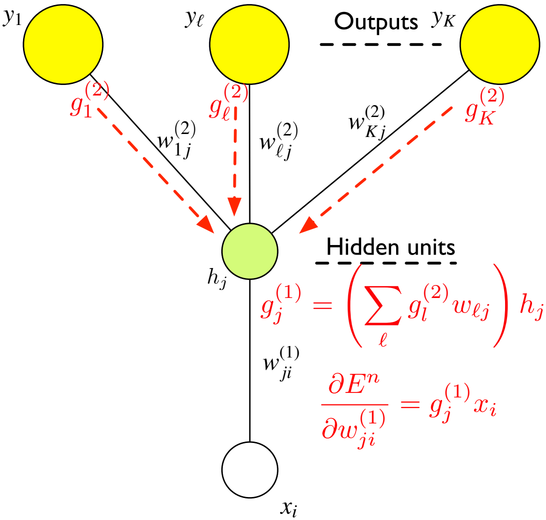
- Cross-entropy error function:

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

- Required gradients: $\dfrac{\partial E^n}{\partial w_{kj}^{(2)}} \quad \dfrac{\partial E^n}{\partial w_{ji}^{(1)}} \quad \dfrac{\partial E^n}{\partial b_k^{(2)}} \quad \dfrac{\partial E^n}{\partial b_j^{(1)}}$
- **Gradient for hidden-to-output weights** similar to single-layer network:

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

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



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

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = g_j^{(1)} x_i$$

Training MLPs: Input-to-hidden weights

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

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

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

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

Training MLPs: Gradients

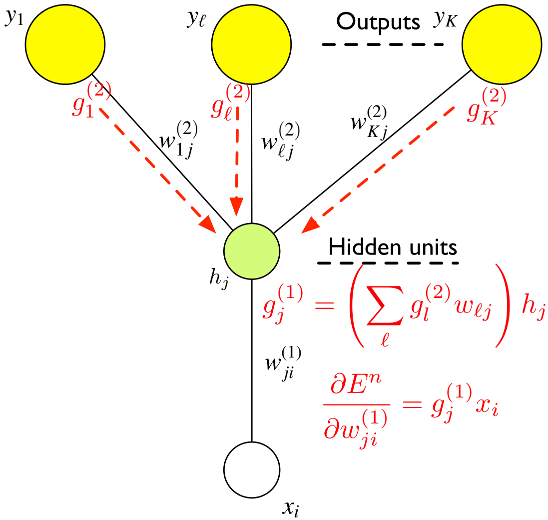
$$\boxed{\frac{\partial E^n}{\partial w_{kj}^{(2)}}} = \underbrace{(y_k - t_k)}_{g_k^{(2)}} \cdot h_j^{(1)}$$

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

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

$$\frac{\partial E^n}{\partial b_k^{(2)}} \qquad \frac{\partial E^n}{\partial b_j^{(1)}}$$

Back-propagation of error: hidden unit error signal



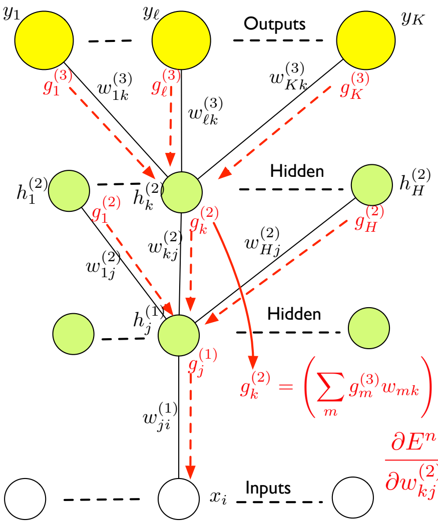
$$g_j^{(1)} = \left(\sum_\ell g_l^{(2)} w_{\ell j} \right) h_j (1 - h_j)$$

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = g_j^{(1)} x_i$$

Back-propagation of error

- The back-propagation of error algorithm is summarised as follows:
  1. Apply an input vectors from the training set, $\mathbf{x}$, to the network and forward propagate to obtain the output vector $\mathbf{y}$
  2. Using the target vector $\mathbf{t}$ 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

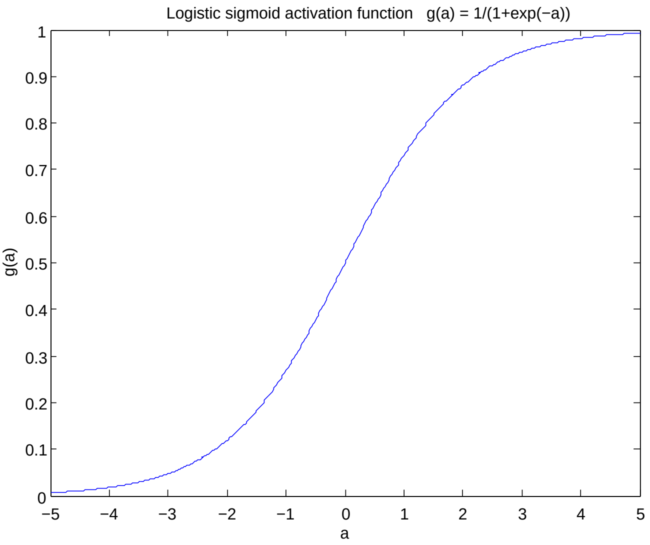


$$g_k^{(2)} = \left(\sum_m g_m^{(3)} w_{mk} \right) h_k^{(2)} (1 - h_k^{(2)})$$

$$\frac{\partial E^n}{\partial w_{kj}^{(2)}} = g_k^{(2)} h_j^{(1)}$$

Are there alternatives to Sigmoid Hidden Units?

Sigmoid function



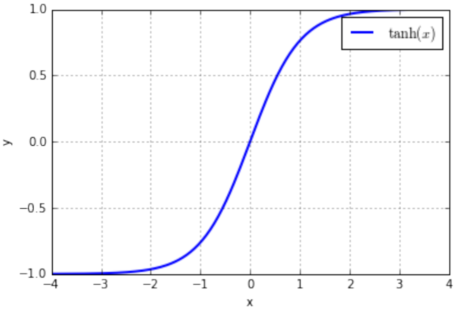
Sigmoid Hidden Units

- 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 then 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



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

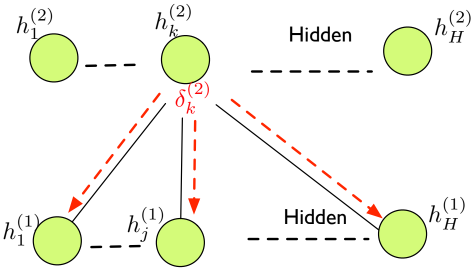
$$\text{sigmoid}(x) = \frac{1 + \tanh(x/2)}{2}$$

Derivative:

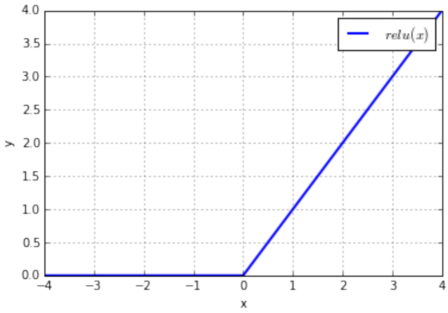
$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

tanh hidden units

- tanh has same shape as sigmoid but has output range ± 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



$$\text{relu}(x) = \max(0, x)$$

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