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

Hakan Bilen

Machine Learning Practical — MLP Lecture 3 1 October 2019 http://www.inf.ed.ac.uk/teaching/courses/mlp/

#### **Recap:** Softmax single layer network



## What Do

## Neural Networks Do?

# What Do Single Layer

Neural Networks Do?

#### Single-layer network, 1 output, 2 inputs





#### Single-layer network, 3 outputs, 2 inputs



#### Example data (three classes)



#### Classification regions with single-layer network

Plot of Decision regions



Single-layer networks are limited to linear classification boundaries

#### Single layer network trained on MNIST Digits



#### Weights of each output unit define a "template" for the class

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)





## What are the pros and cons of using a single-layer network for MNIST classification?

## Multi-Layer Networks

## 

- 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

## 

- 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



Layered processing: inputs - features - classification



Layered processing: inputs - features - classification



How to obtain features? - learning!







#### Multi-layer network



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

#### 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  $w_{ii}^{(1)}$  to output unit k?

#### 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  $w_{ii}^{(1)}$  to output unit k?
- What is the gradient of the error with respect to each weight? (How to compute grads\_wrt\_params?)

#### 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  $w_{ii}^{(1)}$  to output unit k?
- What is the gradient of the error with respect to each weight? (How to compute grads\_wrt\_params?)
- 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 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 (grads\_wrt\_params):  $\frac{\partial E^n}{\partial w_{\iota_i}^{(2)}}$   $\frac{\partial E^n}{\partial w_{\iota_i}^{(1)}}$   $\frac{\partial E^n}{\partial b_{\iota}^{(2)}}$   $\frac{\partial E^n}{\partial b_{\iota}^{(1)}}$
- Gradient for hidden-to-output weights similar to single-layer network:

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

$$= \underbrace{(y_{k} - t_{k})}_{g_{k}^{(2)}} h_{j}^{(1)}$$

#### Training MLPs: Error function and required gradients

• Cross-entropy error function:

gı

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

- Required gradients (grads\_wrt\_params):  $\frac{\partial E^n}{\partial w_{ki}^{(2)}}$   $\frac{\partial E^n}{\partial w_{ii}^{(1)}}$   $\frac{\partial E^n}{\partial b_k^{(2)}}$   $\frac{\partial E^n}{\partial b_i^{(1)}}$
- Gradient for hidden-to-output weights similar to single-layer network:

$$\underbrace{\left[\frac{\partial E^{n}}{\partial w_{kj}^{(2)}}\right]}_{\text{rads_wrt_params}} = \frac{\partial E^{n}}{\partial z_{k}^{(2)}} \cdot \frac{\partial z_{k}^{(2)}}{\partial w_{kj}} = \left(\sum_{c=1}^{C} \frac{\partial E^{n}}{\partial y_{c}} \cdot \frac{\partial y_{c}}{\partial z_{k}^{(2)}}\right) \cdot \frac{\partial z_{k}^{(2)}}{\partial w_{kj}}$$

$$= \underbrace{(y_{k} - t_{k})}_{\text{error.grad}} h_{j}^{(1)}$$

#### Back-propagation of error: hidden unit error signal



#### Training MLPs: Input-to-hidden weights



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

$$\begin{split} \boxed{g_j^{(1)}} &= \sum_{c=1}^{K} \frac{\partial E^n}{\partial z_c^{(2)}} \cdot \frac{\partial z_c^{(2)}}{\partial z_j^{(1)}} = \left(\sum_{c=1}^{K} g_c^{(2)} \cdot \frac{\partial z_c^{(2)}}{\partial h_j^{(1)}}\right) \cdot \frac{\partial h_j^{(1)}}{\partial z_j^{(1)}} \\ &= \left(\sum_{c=1}^{K} g_c^{(2)} w_{cj}^{(2)}\right) h_j^{(1)} (1 - h_j^{(1)}) \end{split}$$

#### Training MLPs: Input-to-hidden weights



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

$$\begin{aligned}
\left[ g_{j}^{(1)} \right] &= \sum_{c=1}^{K} \frac{\partial E^{n}}{\partial z_{c}^{(2)}} \cdot \frac{\partial z_{c}^{(2)}}{\partial z_{j}^{(1)}} = \left( \sum_{c=1}^{K} g_{c}^{(2)} \cdot \frac{\partial z_{c}^{(2)}}{\partial h_{j}^{(1)}} \right) \cdot \frac{\partial h_{j}^{(1)}}{\partial z_{j}^{(1)}} \\
&= \underbrace{\left( \sum_{c=1}^{K} g_{c}^{(2)} w_{cj}^{(2)} \right)}_{\text{AffineLayer.bprop}} \underbrace{h_{j}^{(1)} (1 - h_{j}^{(1)})}_{\text{SigmoidLayer.bprop}} \\
&= \underbrace{MIP}_{i} \left[ \text{struck} 3 / 1 \text{ October 2019} \right] \quad \text{Deen Neural Networks (1)}
\end{aligned}$$

#### **Training MLPs: Gradients**

• grads\_wrt\_params:

$$\boxed{\begin{array}{l} \displaystyle \frac{\partial E^n}{\partial w_{kj}^{(2)}} = (y_k - t_k) \cdot h_j^{(1)} \\ \\ \displaystyle \frac{\partial E^n}{\partial w_{ji}^{(1)}} = \left(\sum_{c=1}^k g_c^{(2)} w_{cj}^{(2)}\right) \cdot h_j^{(1)} (1 - h_j^{(1)}) \cdot x_i \end{array}$$

#### **Training MLPs: Gradients**

• grads\_wrt\_params:

$$\frac{\partial E^{n}}{\partial w_{kj}^{(2)}} = \underbrace{(y_{k} - t_{k})}_{\text{CrossEntropySoftmaxError.grad}} \cdot h_{j}^{(1)}$$

$$\frac{\partial E^{n}}{\partial w_{ji}^{(1)}} = \underbrace{\left(\sum_{c=1}^{k} g_{c}^{(2)} w_{cj}^{(2)}\right)}_{\text{AffineLayer.bprop}} \cdot \underbrace{h_{j}^{(1)}(1 - h_{j}^{(1)})}_{\text{SigmoidLayer.bprop}} \cdot x_{i}$$

#### **Training MLPs: Gradients**

• grads\_wrt\_params:

$$\begin{array}{c}
 \overline{\frac{\partial E^{n}}{\partial w_{kj}^{(2)}}} = \underbrace{(y_{k} - t_{k})}_{\text{CrossEntropySoftmaxError.grad}} \cdot h_{j}^{(1)} \\
 \overline{\frac{\partial E^{n}}{\partial w_{ji}^{(1)}}} = \underbrace{\left(\sum_{c=1}^{k} g_{c}^{(2)} w_{cj}^{(2)}\right)}_{\text{AffineLayer.bprop}} \cdot \underbrace{h_{j}^{(1)}(1 - h_{j}^{(1)})}_{\text{SigmoidLayer.bprop}} \cdot x_{i}
\end{array}$$

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

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

- The back-propagation of error algorithm is summarised as follows:
  - 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  ${\bf t}$  compute the error  $E^n$
  - 3. Evaluate the error gradients  $g_k^{(2)}$  for each output unit using error.grad
  - 4. Evaluate the error gradients  $g_j^{(1)}$  for each hidden unit using AffineLayer.bprop and SigmoidLayer.bprop
  - 5. Evaluate the derivatives (grads\_wrt\_params) 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



#### Training with multiple hidden layers



#### Training with multiple hidden layers



Are there alternatives to Sigmoid Hidden Units?

#### Sigmoid function



#### Sigmoid Hidden Units (SigmoidLayer)

- 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

#### Sigmoid Hidden Units (SigmoidLayer)

- $\bullet\,$  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 − hence the gradients it multiplies into approach 0.
  - Small gradients result in small parameter changes, so learning becomes slow
- Outputs are not centred at 0
  - The output of a sigmoid layer will have mean > 0 numerically undesirable.

#### tanh



$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
$$sigmoid(x) = \frac{1 + tanh(x/2)}{2}$$

Derivative:  $rac{d}{dx} anh(x) = 1 - anh^2(x)$ 

- tanh has same shape as sigmoid but has output range  $\pm 1$
- Results about approximation capability using sigmoid layers also apply to tanh layers

- tanh has same shape as sigmoid but has output range  $\pm 1$
- Results about approximation capability using sigmoid layers also apply to tanh layers
- 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

- tanh has same shape as sigmoid but has output range  $\pm 1$
- Results about approximation capability using sigmoid layers also apply to tanh layers
- 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



- tanh has same shape as sigmoid but has output range  $\pm 1$
- Results about approximation capability using sigmoid layers also apply to tanh layers
- 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**



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

#### Derivative:

$$\frac{d}{dx}\operatorname{relu}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

#### ReLU hidden units (ReluLayer)

- 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)

#### ReLU hidden units (ReluLayer)

- 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

Lab 3 covers how multiple layer networks are modelled in the mlp framework.

- Representing activation functions as layer, applied after a linear (affine) layer
- Stacking layers together to build multi-layer networks with non-linear activation functions (e.g. sigmoid) for the hidden layer
- Using Layer.fprop methods to implement the forward propagation and and Layer.bprop methods to implement back propagation to compute the gradients
- Training softmax models on MNIST
- Training deeper multi-layer networks on MNIST
- Using various non-linear activation functions for the hidden layer (sigmoid, tanh, relu)

- 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