Regularisation (cont.)
Hidden Unit Transfer Functions

Steve Renals

Machine Learning Practical — MLP Lecture 5
21 October 2015

Training with Momentum

$$\Delta \mathbf{w}(t) = -(1-\alpha)\eta \frac{\partial E}{\partial \mathbf{w}} + \alpha \Delta \mathbf{w}(t-1)$$

- α is the *momentum*
- Weight changes start by following the gradient
- After a few updates they start to have *velocity* – no longer pure gradient descent
- Momentum term encourages the weight change to go in the previous direction
- Damps the random directions of the gradients, to encourage weight changes in a consistent direction

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*
- Time-dependent schedules
  - **Piecewise constant**: pre-determined η for each epoch)
  - **Exponential**: $\eta(t) = \eta(0)\exp(-t/r)$ ($r \sim$ training set size)
  - **Reciprocal**: $\eta(t) = \eta(0)(1 + t/r)^{-c}$ ($c \sim 1$)
- **Performance-dependent** η – e.g. "NewBOB": fixed η until validation set stops improving, then halve η each epoch (i.e. constant, then exponential)
- **Weight-dependent** η (e.g. AdaGrad, RMSProp) – control η based on moving average of gradients for the weight

Recap: Backprop Training with Weight Decay

$$\begin{aligned}
\frac{\partial E^n}{\partial w_i} &= \frac{\partial (E^n_{\text{train}} + E_{L2})}{\partial w_i} \\
&= \frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta \frac{\partial E_{L2}}{\partial w_i} \\
&= \frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta w_i \\
\Delta w_i &= -\eta \left(\frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta w_i \right)
\end{aligned}$$

- Weight decay corresponds to adding $E_{L2} = 1/2 \sum_i w_i^2$ to the error function
- Addition of complexity terms is called *regularisation*
- Weight decay is sometimes called L2 regularisation

L1 Regularisation

- **L1 Regularisation** corresponds to adding a term based on summing the absolute values of the weights to the error:

$$\begin{aligned}
E^n &= \underbrace{E^n_{\text{train}}}_{\text{data term}} + \underbrace{\beta E^n_{L1}}_{\text{prior term}} \\
&= E^n_{\text{train}} + \beta |w_i|
\end{aligned}$$

- Gradients

$$\begin{aligned}
\frac{\partial E^n}{\partial w_i} &= \frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta \frac{\partial E_{L1}}{\partial w_i} \\
&= \frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta \operatorname{sgn}(w_i)
\end{aligned}$$

Where $\operatorname{sgn}(w_i)$ is the sign of w_i:
$\operatorname{sgn}(w_i) = 1$ if $w_i > 0$ and $\operatorname{sgn}(w_i) = -1$ if $w_i < 0$

L1 vs L2

- L1 and L2 regularisation both have the effect of penalising larger weights
  - In L2 they shrink to 0 at a rate proportional to the size of the weight (βw_i)
  - In L1 they shrink to 0 at a constant rate ($\beta \operatorname{sgn}(w_i)$)
- Behaviour of L1 and L2 regularisation with large and small weights:
  - when $|w_i|$ is large L2 shrinks faster than L1
  - when $|w_i|$ is small L1 shrinks faster than L2
- So L1 tends to shrink some weights to 0, leaving a few large important connections – L1 encourages *sparsity*
- $E_{L1}(0)$ is undefined; we take $\operatorname{sgn}(0) = 0$

Adding "fake" training data

- Generalisation performance goes with the amount of training data (change `MNISTDataProvider` to give training sets of 1 000 / 5 000 / 10 000 examples to see this)
- Given a finite training set we could *create* further training examples...
  - Create new examples by making small rotations of existing data
  - Add a small amount of random noise
- Using "realistic" distortions to create new data is better than adding random noise

Model Combination

- Combining the predictions of multiple models can reduce overfitting
- Model combination works best when the component models are *complementary* – no single model works best on all data points
- Creating a set of diverse models
  - Different NN architectures (number of hidden units, number of layers, hidden unit type, input features, type of regularisation, ...)
  - Different models (NN, SVM, decision trees, ...)
- How to combine models?
  - Average their outputs
  - Linearly combine their outputs
  - Train another "combiner" neural network whose input is the outputs of the component networks
  - Architectures designed to create a set of specialised models which can be combined (e.g. mixtures of experts)

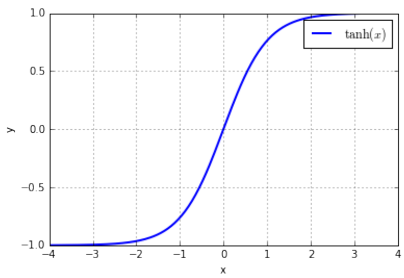
Dropout

- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
- Dropout training:
  - Each mini-batch randomly delete a fraction (say half) of the hidden units (and their related weights and biases)
  - Them process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random sunset of hidden units to delete and repeat the process
- When training is complete the network will have learned a complete set of weights and biases, all learned when half the hidden units are missing. (To compensate for this, in the final network we halve the values of the outgoing weights from each hidden unit)

Why does Dropout work?

- Each mini-batch is like training a different network, since we randomly select to dropout half the neurons
- So we can imagine dropout as combining an exponential number of networks
- Since the component networks will be complementary and overfit in different ways, dropout is implicit model combination
- Also interpret dropout as training more robust hidden unit features – each hidden unit cannot rely on all other hidden unit features being present, must be robust to missing features
- Dropout has been useful in improving the generalisation of large-scale deep networks
- **Annealed Dropout:** Dropout rate schedule starting with a fraction p units dropped, decreasing at a constant rate to 0
  - Initially training with dropout
  - Eventually fine-tune all weights together

tanh

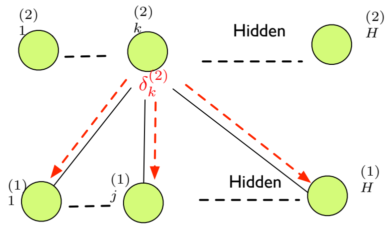


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad ; \quad \mathrm{sigmoid}(x) = \frac{1 + \tanh(x/2)}{2}$$

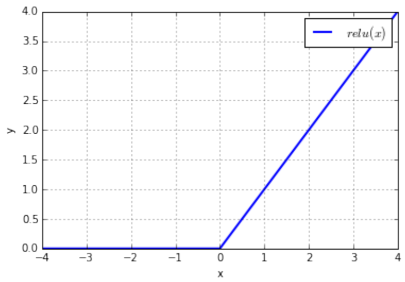
$$\text{Derivative:} \quad \frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

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



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

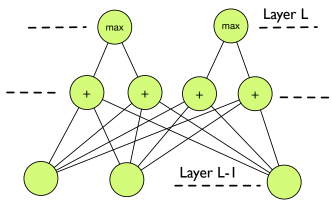
- Similar approximation results to tanh and sigmoid hidden units
- Empirical results for speech and vision show consistent improvements using relu over sigmoid or tanh
- Unlike tanh or sigmoid there is no positive saturation – saturation results in very small derivatives (and hence slower learning)
- 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

Maxout units

- Unit that takes the max of two linear functions $z_i = \mathbf{w}^i \mathbf{h}^{\mathbf{L}-\mathbf{1}}$:

$$h = \max(z_1, z_2)$$

  (if $\mathbf{w}^2 = 0$ then we have Relu)
- Has the benefits of Relu (piecewise linear, no saturation), without the drawback of dying units
- Twice the number of parameters



Generalising maxout

- Units can take the max over G linear functions z_i:

$$h = \max_{i=0}^{G}(z_i)$$

- Maxout can be generalised to other functions, e.g. p-norm

$$h = ||\mathbf{z}||_p = \left(\sum_{i=0}^{G} |z_i|^p\right)^{1/p}$$

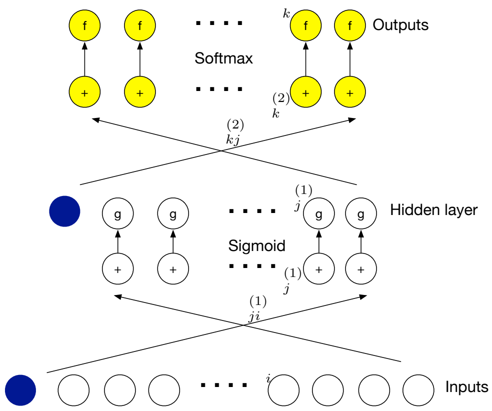
  Typically $p = 2$
- p can be learned by gradient descent.
  (Exercise: What is the gradient $\partial E / \partial p$ for a p-norm unit?)

Summary

- Further approaches to improve generalisation
  - L1 regularisation
  - Creating additional training data
  - Model combination
  - Dropout
- Hidden unit transfer functions
  - tanh
  - ReLU
  - Maxout
- Reading:
  Michael Nielsen, chapter 3 of *Neural Networks and Deep Learning*
  http://neuralnetworksanddeeplearning.com/chap3.html

Postscript:
Derivatives of Transfer Functions

Sigmoid hidden layer



Jacobian of the transfer function Sigmoid

(The Jacobian $\mathbf{J}$ is the matrix of partial derivatives)

$$\mathbf{J} = \begin{pmatrix} \frac{\partial h_0}{\partial a_0} & \cdot & \frac{\partial h_0}{\partial a_i} & \cdot & \frac{\partial h_0}{\partial a_H} \\ & & \cdots & & \\ \frac{\partial h_i}{\partial a_0} & \cdot & \frac{\partial h_i}{\partial a_i} & \cdot & \frac{\partial h_i}{\partial a_H} \\ & & \cdots & & \\ \frac{\partial h_H}{\partial a_0} & \cdot & \frac{\partial h_H}{\partial a_i} & \cdot & \frac{\partial h_H}{\partial a_H} \end{pmatrix} = \begin{pmatrix} \frac{\partial h_0}{\partial a_0} & \cdot & 0 & \cdot & 0 \\ & & \cdots & & \\ 0 & \cdot & \frac{\partial h_i}{\partial a_i} & \cdot & 0 \\ & & \cdots & & \\ 0 & \cdot & 0 & \cdot & \frac{\partial h_H}{\partial a_H} \end{pmatrix}$$

$$= \begin{pmatrix} h_0(1-h_0) & \cdot & 0 & \cdot & 0 \\ & & \cdots & & \\ 0 & \cdot & h_i(1-h_i) & \cdot & 0 \\ & & \cdots & & \\ 0 & \cdot & 0 & \cdot & h_H(1-h_H) \end{pmatrix}$$

Recap: Backprop with a sigmoid hidden layer

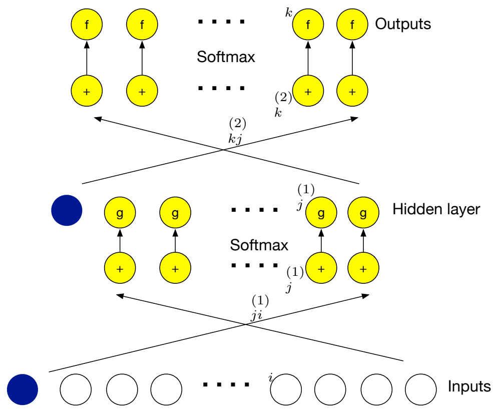
$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \underbrace{\frac{\partial E^n}{\partial a_j^{(1)}}}_{\delta_j^{(1)}} \cdot \underbrace{\frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}}_{x_i}$$

For a sigmoid hidden unit:

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

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

Softmax hidden layer



Jacobian of the transfer function Softmax

$$\mathbf{J} = \begin{pmatrix} \frac{\partial h_0}{\partial a_0} & \cdot & \frac{\partial h_0}{\partial a_i} & \cdot & \frac{\partial h_0}{\partial a_H} \\ & & \cdots & & \\ \frac{\partial h_i}{\partial a_0} & \cdot & \frac{\partial h_i}{\partial a_i} & \cdot & \frac{\partial h_i}{\partial a_H} \\ & & \cdots & & \\ \frac{\partial h_H}{\partial a_0} & \cdot & \frac{\partial h_H}{\partial a_i} & \cdot & \frac{\partial h_H}{\partial a_H} \end{pmatrix}$$

$$= \begin{pmatrix} h_0(1-h_0) & \cdot & -h_0 h_i & \cdot & -h_0 h_H \\ & & \cdots & & \\ -h_i h_0 & \cdot & h_i(1-h_i) & \cdot & -h_i h_H \\ & & \cdots & & \\ -h_H h_0 & \cdot & -h_H h_i & \cdot & h_H(1-h_H) \end{pmatrix}$$

$$J_{ij} = h_i(\boldsymbol{\delta_{ij}} - h_j)$$

Backprop with a softmax hidden layer

For softmax, the normlisation term makes it more complicated

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

$$= \sum_{c \in \text{Layer } 2} \delta_c^{(2)} \cdot \sum_{k \in \text{Layer } 1} \frac{\partial a_c^{(2)}}{\partial h_k^{(1)}} \cdot \frac{\partial h_k^{(1)}}{\partial a_j^{(1)}}$$

$$= \sum_{c \in \text{Layer } 2} \delta_c^{(2)} \cdot \sum_{k \in \text{Layer } 1} w_{ck}^{(2)} h_k^{(1)} (\boldsymbol{\delta}_{kj} - h_j^{(1)})$$