Training with Momentum

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

- \( \alpha \) 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

Recap: Backprop Training with Weight Decay

\[
\begin{align*}
\frac{\partial E^n}{\partial \mathbf{w}_i} &= \frac{\partial E_{\text{train}}^n}{\partial \mathbf{w}_i} + \beta \frac{\partial E_{\text{L1}}}{\partial \mathbf{w}_i} \\
&= \frac{\partial E_{\text{train}}^n}{\partial \mathbf{w}_i} + \beta \text{sgn}(\mathbf{w}_i)
\end{align*}
\]

- Weight decay corresponds to adding \( E_{\text{L2}} = 1/2 \sum w_i^2 \) 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:
  \[ E^n = E_{\text{train}}^n + \beta E_{\text{L1}}^n \]
  \[ = E_{\text{train}}^n + \beta |\mathbf{w}_i| \]
- Gradients
  \[
  \begin{align*}
  \frac{\partial E^n}{\partial \mathbf{w}_i} &= \frac{\partial E_{\text{train}}^n}{\partial \mathbf{w}_i} + \beta \frac{\partial E_{\text{L1}}}{\partial \mathbf{w}_i} \\
  &= \frac{\partial E_{\text{train}}^n}{\partial \mathbf{w}_i} + \beta \text{sgn}(\mathbf{w}_i)
  \end{align*}
  \]
  Where \( \text{sgn}(w_i) \) is the sign of \( w_i \):
  \( \text{sgn}(w_i) = 1 \) if \( w_i > 0 \) and \( \text{sgn}(w_i) = -1 \) 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 (\( \beta \text{sgn}(w_i) \))
  - In L1 they shrink to 0 at a constant rate (\( \beta \text{sgn}(w_i) \))
- Behaviour of L1 and L2 regularisation with large and small weights:
  - when \( |w| \) is large L2 shrinks faster than L1
  - when \( |w| \) 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
- \( E_{\text{L1}}(0) \) is undefined; we take \( \text{sgn}(0) = 0 \)
Adding “fake” training data

- Generalisation performance goes with the amount of training data (change MNISTDataProvider to give training sets of 1000 / 5000 / 10000 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)

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

```
<table>
<thead>
<tr>
<th>h_k^2 (2)</th>
<th>h_j^2 (1)</th>
<th>( h_k^2 ) (2)</th>
<th>h_j^2 (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H_k (2)</td>
<td>H_j (1)</td>
<td>H_k (2)</td>
<td>H_j (1)</td>
</tr>
</tbody>
</table>
```

\[
tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \text{sigmoid}(x) = \frac{1 + \tanh(x/2)}{2}
\]

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

```
tanh

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

```
tanh hidden units

- \( \tanh \) has same shape as \( \text{sigmoid} \) but has output range ±1
- Results about approximation capability of \( \text{sigmoid} \) networks also apply to \( \tanh \) networks
- Possible reason to prefer \( \tanh \) over \( \text{sigmoid} \): 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 \( z_i = \mathbf{w}_i \mathbf{h}_{L-1} \): 
\[ h = \max(z_1, z_2) \]
  
  (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 
\[ h = ||z||_p = \left( \sum_{i=0}^{G} |z_i|^p \right)^{1/p} \]

Typically \( p = 2 \)
- \( p \) can be learned by gradient descent.
  (Exercise: What is the gradient \( \frac{\partial E}{\partial p} \) 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*

Postscript:
Derivatives of Transfer Functions
Recap: Backprop with a sigmoid hidden layer

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

For a sigmoid hidden unit:

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

\[ J = \begin{bmatrix} \frac{\partial h_0}{\partial x_0} & \cdots & \frac{\partial h_0}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_0} & \cdots & \frac{\partial h_k}{\partial x_n} \end{bmatrix} \]

\[ J = \begin{bmatrix} h_0(1 - h_0) & -h_0h_1 & -h_0h_k \\ -h_1h_0 & h_1(1 - h_1) & -h_1h_k \\ \vdots & \vdots & \vdots \\ -h_kh_0 & -h_kh_1 & h_k(1 - h_k) \end{bmatrix} \]

\[ J_0 = h_0(\delta_0 - h_0) \]

Softmax hidden layer

\[ J = \begin{bmatrix} \frac{\partial h_0}{\partial x_0} & \cdots & \frac{\partial h_0}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_0} & \cdots & \frac{\partial h_k}{\partial x_n} \end{bmatrix} \]

\[ J = \begin{bmatrix} h_0(1 - h_0) & -h_0h_1 & -h_0h_k \\ -h_1h_0 & h_1(1 - h_1) & -h_1h_k \\ \vdots & \vdots & \vdots \\ -h_kh_0 & -h_kh_1 & h_k(1 - h_k) \end{bmatrix} \]

\[ J_0 = h_0(\delta_0 - h_0) \]

Backprop with a softmax hidden layer

For softmax, the normalisation term makes it more complicated

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

\[ = \sum_{c \in \text{Layer 2}} \delta_c^{(2)} \cdot \sum_{k \in \text{Layer 1}} \frac{\partial a_k^{(3)}}{\partial a_j^{(1)}} \cdot \frac{\partial h_k^{(1)}}{\partial a_j^{(1)}} \]

\[ = \sum_{c \in \text{Layer 2}} \delta_c^{(2)} \cdot \sum_{k \in \text{Layer 1}} \psi_k^{(3)} h_k^{(1)} (\delta_{kj} - h_j^{(1)}) \]

\[ J_0 = h_0(\delta_0 - h_0) \]