Regularisation and Hidden Unit Transfer Functions

Steve Renals

Machine Learning Practical — MLP Lecture 5
19 October 2016

Recap: Overtraining

- Overtraining corresponds to a network function too closely fit to the training set (too much flexibility)
- Undertraining corresponds to a network function not well fit to the training set (too little flexibility)
- Solutions
    - If possible increasing both network complexity in line with the training set size
    - Use prior information to constrain the network function
    - Control the flexibility: **Structural Stabilisation**
    - Control the *effective flexibility*: **early stopping** and **regularisation**

Regularisation

Weight Decay (L2 Regularisation)

- Weight decay puts a “spring” on weights
- If training data puts a consistent force on a weight, it will outweigh weight decay
- If training does not consistently push weight in a direction, then weight decay will dominate and weight will decay to 0
- Without weight decay, weight would walk randomly without being well determined by the data
- Weight decay can allow the data to determine how to reduce the effective number of parameters

Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E^n_{\text{train}}}_{\text{data term}} + \underbrace{\beta E_W}_{\text{prior term}}$$

Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E^n_{\text{train}}}_{\text{data term}} + \underbrace{\beta E_W}_{\text{prior term}}$$

- E_{train} is the usual error function:

$$\boxed{E^n_{\text{train}}} = -\sum_{k=1}^{K} t^n_k \ln y^n_k$$

Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E^n_{\text{train}}}_{\text{data term}} + \underbrace{\beta E_W}_{\text{prior term}}$$

- E_{train} is the usual error function:

$$\boxed{E^n_{\text{train}}} = -\sum_{k=1}^{K} t^n_k \ln y^n_k$$

- E_W should be a differentiable flexiblity/complexity measure, e.g.

$$\boxed{E_W = E_{L2}} = \frac{1}{2}\sum_i w_i^2$$

$$\boxed{\frac{\partial E_{L2}}{\partial w_i}} = w_i$$

Gradient Descent Training with Weight Decay

$$\boxed{\frac{\partial E^n}{\partial w_i}} = \frac{\partial (E^n_{\text{train}} + E_{L2})}{\partial w_i} = \left(\frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta \frac{\partial E_{L2}}{\partial w_i} \right)$$

$$= \left(\frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta w_i \right)$$

$$\Delta w_i = -\eta \left(\frac{\partial E^n_{\text{train}}}{\partial w_i} + \beta w_i \right)$$

- 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:

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

- Gradients

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

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*
- $\partial E_{L1}/\partial w$ is undefined when $w = 0$; assume it is 0 (i.e. take $\operatorname{sgn}(0) = 0$ in the update equation)

Data Augmentation – 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

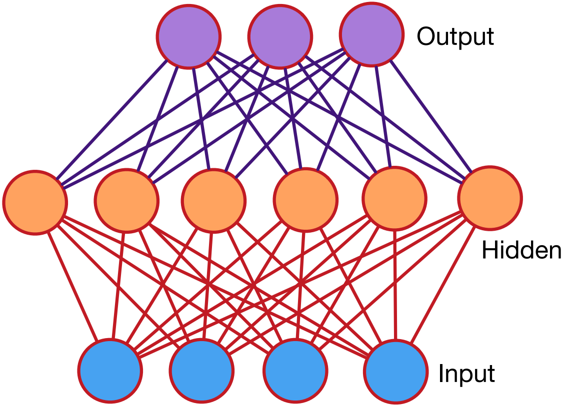
- Combining the predictions of multiple models can reduce overfitting
- Model combination works best when the component models are *complementary* – no single model works best on all data points
- Creating a set of diverse models
  - Different NN architectures (number of hidden units, number of layers, 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

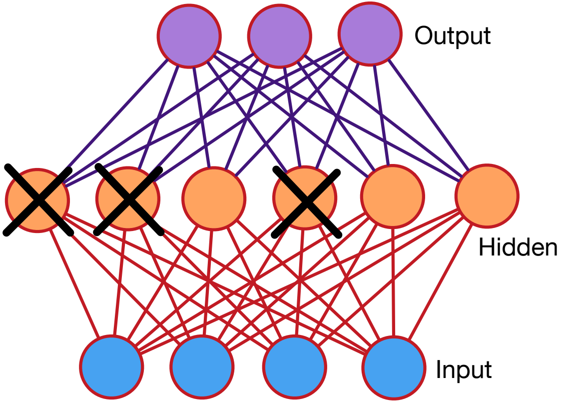
Dropout

- **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 ($p \sim 0.5$) of the hidden units (and their related weights and biases)
  - Then process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random subset of hidden units to delete and repeat the process

Dropout Training - Complete Network

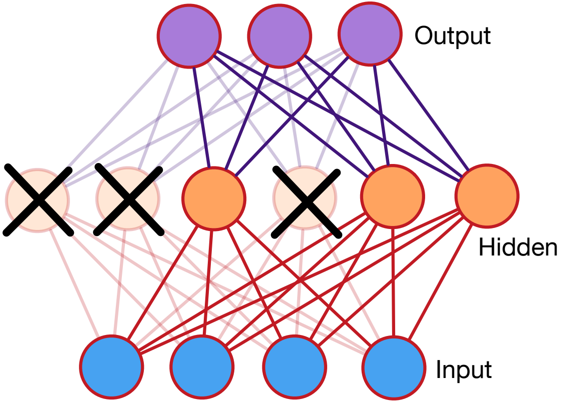


Dropout Training - First Minibatch



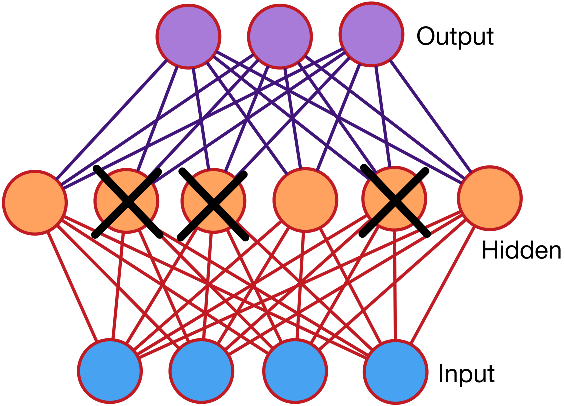
$p = 0.5$

Dropout Training - First Minibatch



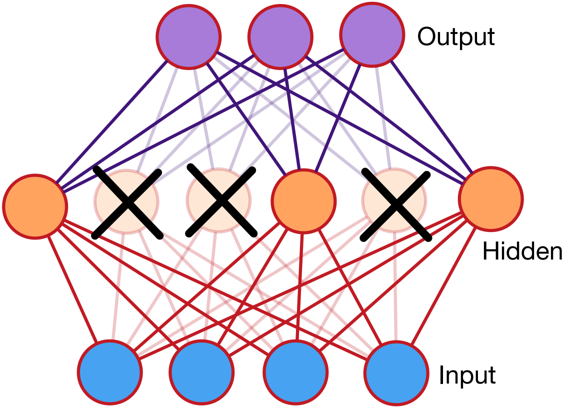
$p = 0.5$

Dropout Training - Second Minibatch



$p = 0.5$

Dropout Training - Second Minibatch



$p = 0.5$

Dropout

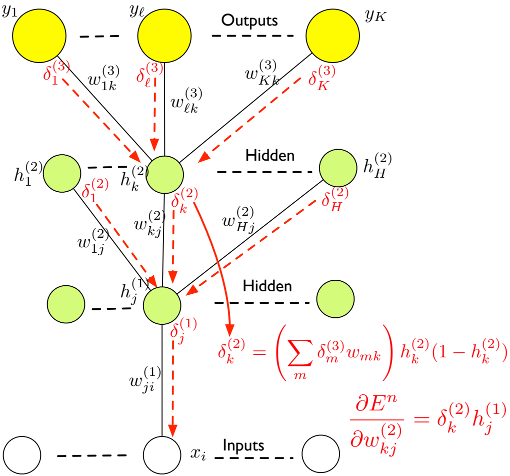
- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
- Dropout training:
  - Each mini-batch randomly delete a fraction ($p \sim 0.5$) of the hidden units (and their related weights and biases)
  - Then process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random subset 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 a fraction p of the hidden units are missing. To compensate for this, in the final network we scale hidden unit activations by p.
- Inverted dropout: scale by $1/p$ when training, no scaling in final network.

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

Are there alternatives to Sigmoid Hidden Units?

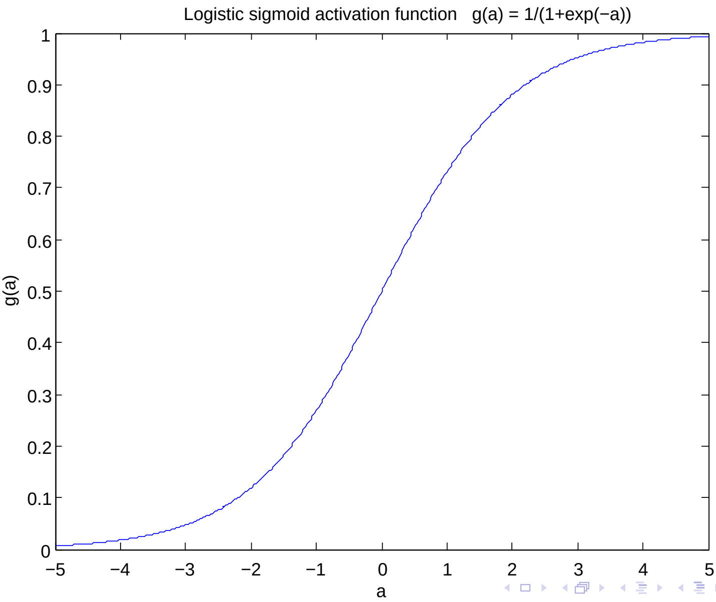
Recap: Architectures of multi-layer networks



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

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

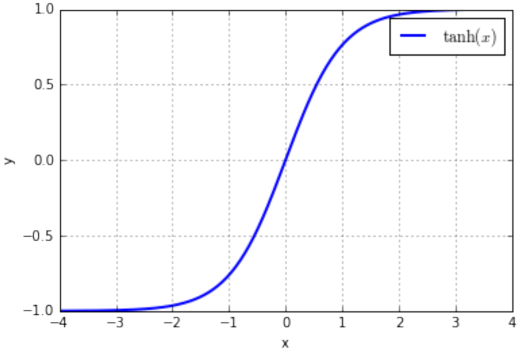
Sigmoid function



Sigmoid Hidden Units

- In their favour
  - 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

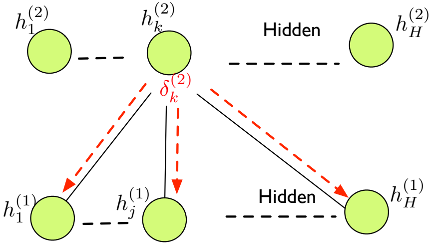


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad ; \quad \text{sigmoid}(x) = \frac{1 + \tanh(x/2)}{2}$$

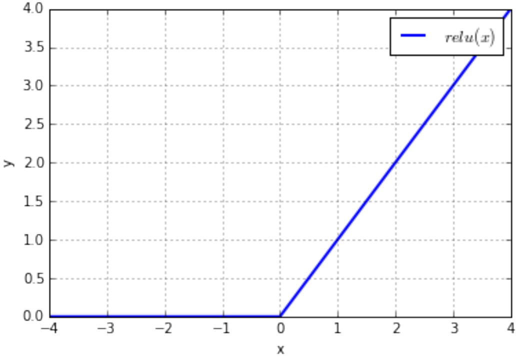
$$\text{Derivative:} \quad \frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

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
- Saturation still a problem



Rectified Linear Unit – ReLU



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

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

ReLU hidden units

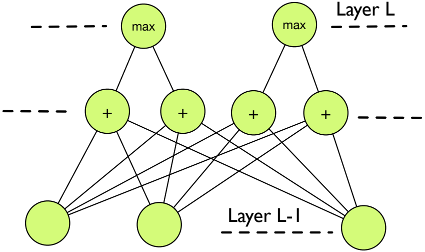
- Similar approximation results to tanh and sigmoid hidden units
- Empirical results for speech and vision show consistent improvements using relu over sigmoid or tanh
- Unlike tanh or sigmoid there is no positive saturation – saturation results in very small derivatives (and hence slower learning)
-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

$$z_i = \mathbf{w}_i \mathbf{h}^{\mathbf{L-1}} + b_i \qquad i = \{1, 2\}$$
$$h = \max(z_1, z_2)$$

  (if $\mathbf{w}_2 = 0, b_2 = 0$ 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

- Regularisation
    - L2 regularisation – weight decay
    - L1 regularisation – sparsity
    - 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