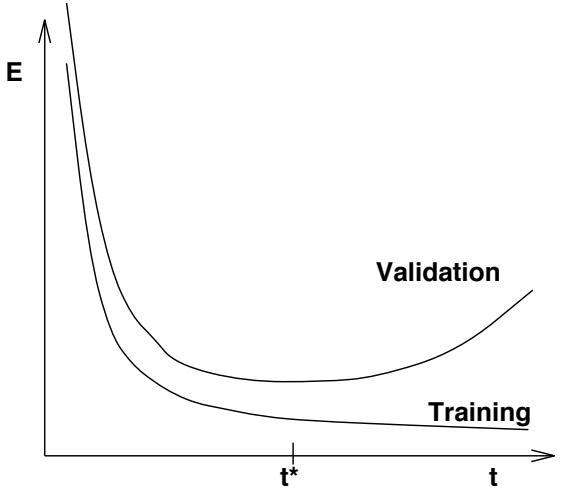
Deep Neural Networks (3)
Regularisation and Normalisation

Steve Renals

Machine Learning Practical — MLP Lecture 5
18 October 2017 / 23 October 2017

Recap: Early stopping



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:

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

- 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

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

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

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

- 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:

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

- Gradients

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

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 (βw_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 $|w_i|$ 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 $\text{sgn}(0) = 0$ 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

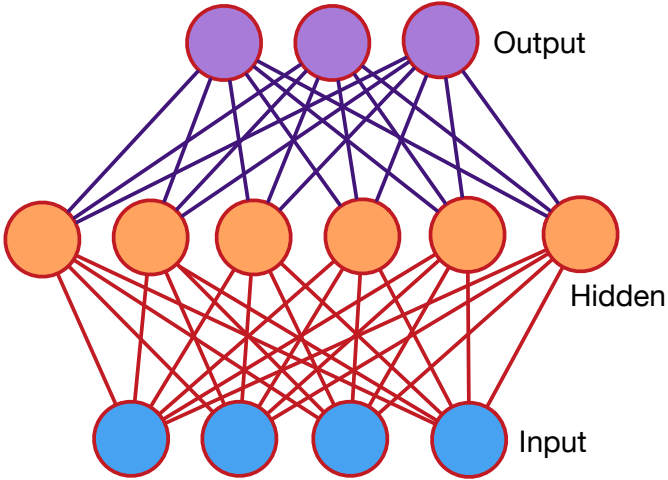
Model Combination

- 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)

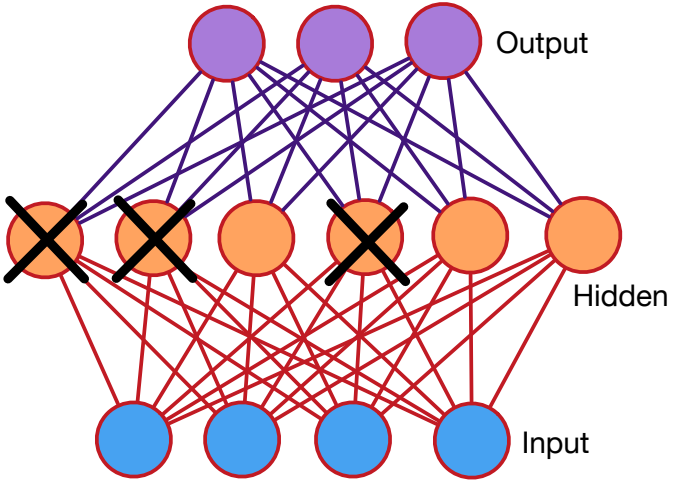
Dropout

- **Dropout** 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

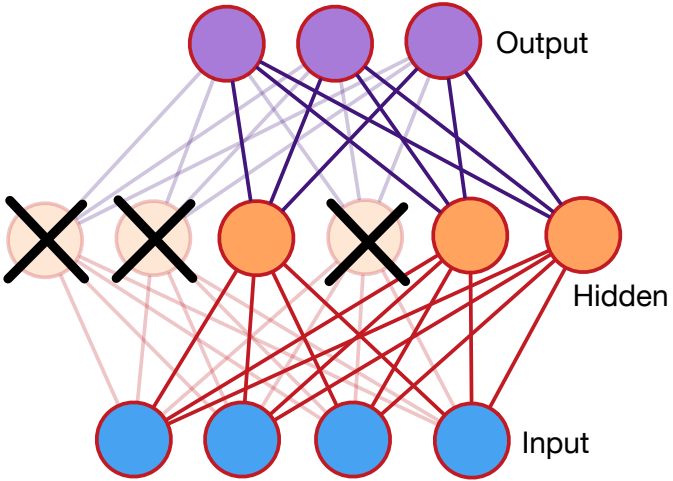


Dropout Training - First Minibatch



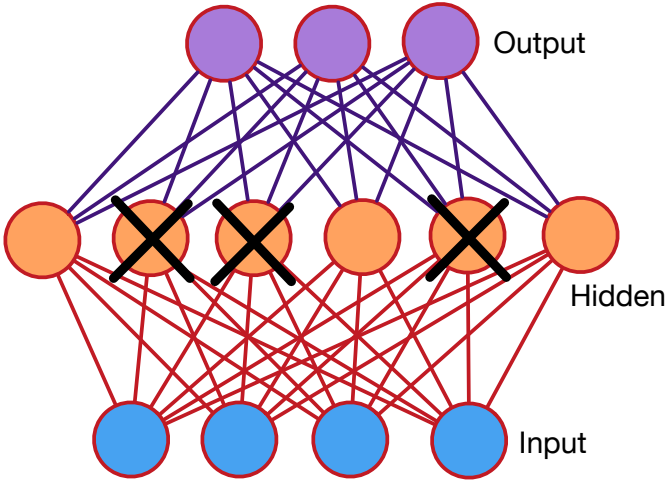
$p = 0.5$

Dropout Training - First Minibatch



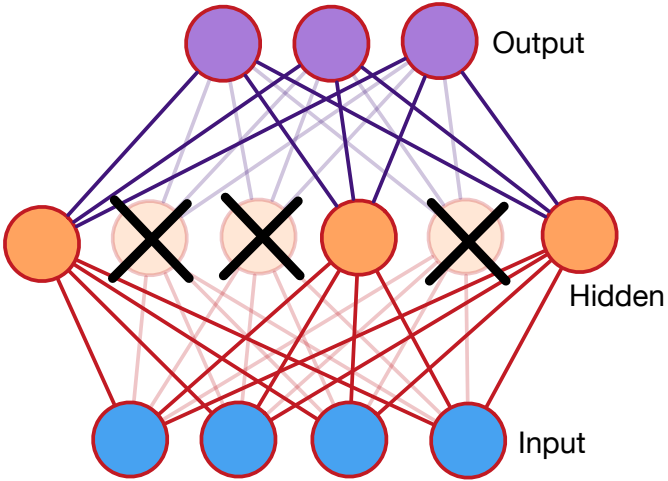
$p = 0.5$

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 $(1 - p)$ (i.e. the probability that a weight is included).
- Inverted dropout: scale by $1/(1 - p)$ 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

Data Preprocessing

- Normalisation
  - Subtract the mean of the input data from every feature, and scale by its standard deviation

$$\hat{x_i^n} = \frac{x_i^n - \text{mean}(x_i)}{\text{sd}(x_i)}$$

- PCA - Principal Components Analysis
  - Decorrelate the data by projecting onto the principal components
  - Also possible to reduce dimensionality by only projecting onto the top P principal components
- Whitening
  - Decorrelate by PCA
  - Scale each dimension

PCA and Whitening

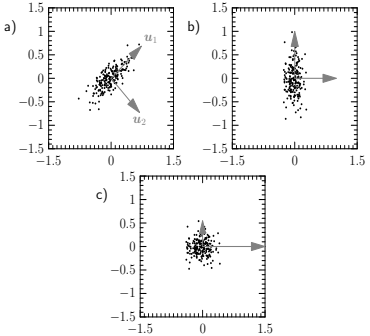
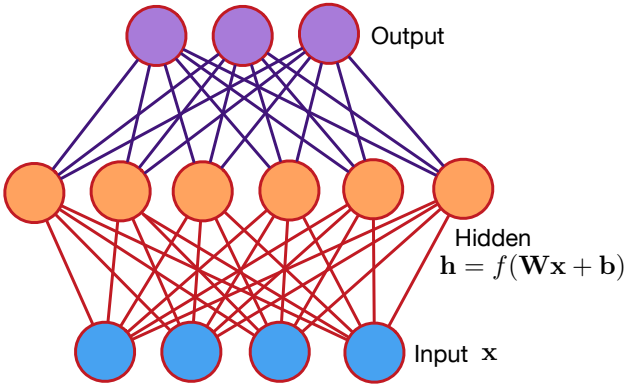


Figure 3: Illustration of PCA and whitening. a) The original data "cloud". The arrows show the principal components. The first one points in the direction of the largest variance in the data, and the second in the remaining orthogonal direction. b) When the data is transformed to the principal components, i.e. the principal components are taken as the new coordinates, the variation in the data is aligned with those new axes, which is because the principal components are uncorrelated. c) When the principal components are further normalized to unit variance, the data cloud has equal variance in all directions, which means it has been whitened. The change in the lengths of the arrows reflects this normalization; the larger the variance, the shorter the arrow.

from Hyvärinen et al, *Natural Image Statistics*, Springer, 2009.

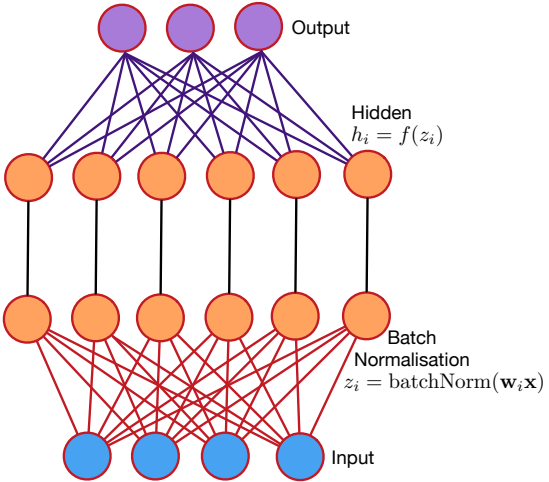
Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

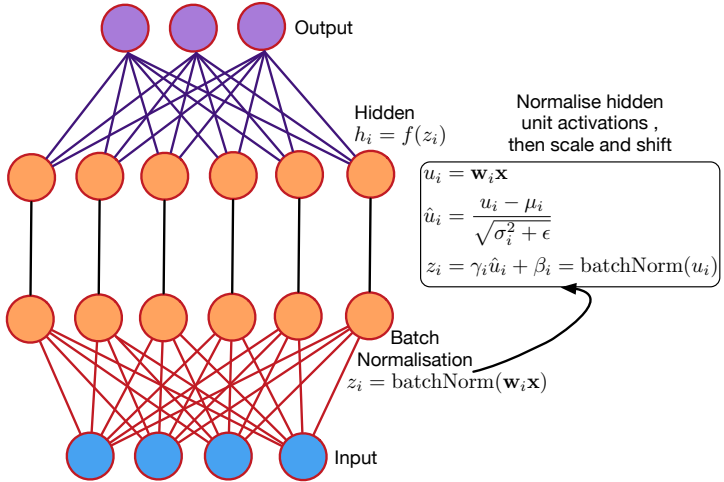
Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

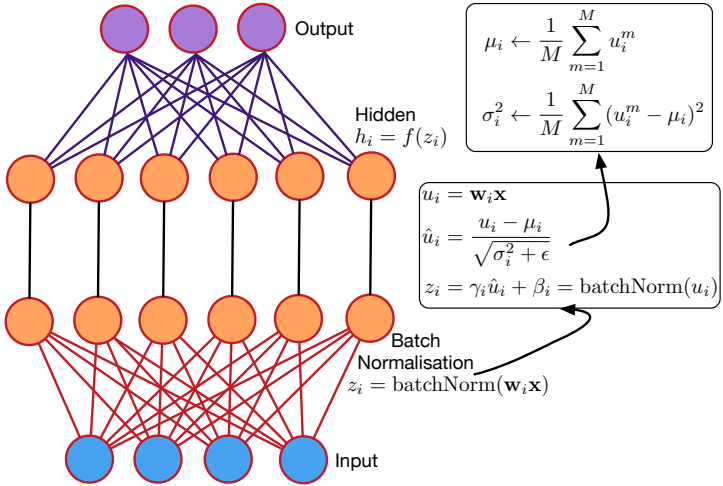
Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

Batch Normalisation



Compute mean and variance of each hidden unit activation across the minibatch (size M)

Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

Batch normalisation

- Use minibatch statistics to normalise activations of each layer (activations are the argument of the transfer function)
- Parameters γ and β can scale and shift the normalised activations; β can also play the role of bias
- batchNorm depends on the current training example – and on examples in the minibatch (to compute mean and variance)
- Training
  - Set parameters γ and β by gradient descent – require gradients $\frac{\partial E}{\partial \gamma}$ and $\frac{\partial E}{\partial \beta}$
  - To back-propagate gradients through the batchNorm layer also require: $\frac{\partial E}{\partial \hat{u}}$ $\frac{\partial E}{\partial \sigma^2}$ $\frac{\partial E}{\partial \mu}$ $\frac{\partial E}{\partial u_i}$
- Runtime - use the sample mean and variance computed over the complete training data as the mean and variance parameters for each layer – fixed transform:

$$\hat{u}_i = \frac{u_i - \text{mean}(u_i)}{\sqrt{\text{Var}(u_i) + \epsilon}}$$

Batch normalisation – gradients (for reference)

$$\frac{\partial E}{\partial \hat{u}_i^m} = \frac{\partial E^m}{\partial z_i^m} \cdot \gamma_i$$

$$\frac{\partial E}{\partial \sigma_i^2} = \sum_m \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot (u_i^m - \mu_i) \cdot \frac{-1}{2} \left(\sigma_i^2 + \epsilon\right)^{-3/2}$$

$$\frac{\partial E}{\partial \mu_i} = \left(\sum_m \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \right) + \frac{\partial E}{\partial \sigma_i^2} \cdot \frac{1}{M} \sum_m -2(u_i - \mu_i)$$

$$\frac{\partial E}{\partial u_i^m} = \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot \frac{1}{\sqrt{\sigma_i^2 + \epsilon}} + \frac{\partial E}{\partial \sigma_i^2} \cdot \frac{2(u_i - \mu_i)}{M} + \frac{\partial E}{\partial \mu_i} \cdot \frac{1}{M}$$

$$\frac{\partial E}{\partial \gamma_i} = \sum_m \frac{\partial E^m}{\partial z_i^m} \cdot \hat{u}_i^m$$

$$\frac{\partial E}{\partial \beta_i} = \sum_m \frac{\partial E^m}{\partial z_i^m}$$

see also http://cthorey.github.io/backpropagation/

Benefits of batch normalisation

- Makes training many-layered networks easier
  - Allows higher learning rates
  - Weight initialisation less crucial
- Can act like a regulariser – maybe reduces need for techniques like dropout
- Can be applied to convolutional networks
- In practice (image processing) – achieves similar accuracy with many fewer training cycles
- Very widely used, and very useful for many-layered networks (e.g. visual object recognition)

Summary

- Regularisation
  - L2 regularisation – weight decay
  - L1 regularisation – sparsity
  - Creating additional training data
  - Model combination
  - Dropout
- Feature normalisation
- Batch normalisation
- Reading:
  - Nielsen, chapter 3 of *Neural Networks and Deep Learning* http://neuralnetworksanddeeplearning.com/chap3.html
  - Goodfellow et al, chapter 7 *Deep Learning* (sections 7.1–7.5, 7.12) http://www.deeplearningbook.org/contents/regularization.html