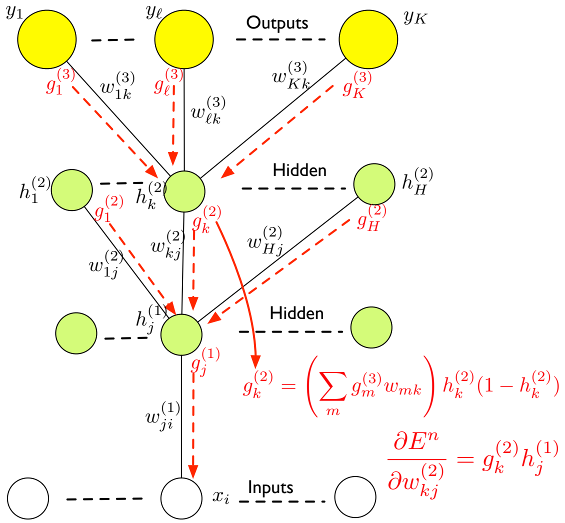
Deep Neural Networks (2)
Generalisation, Training algorithms, Initialisation

Steve Renals

Machine Learning Practical — MLP Lecture 4
11 October 2017 / 16 October 2017

Recap: Training multi-layer networks



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

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

$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \eta (g_k^{(2)} h_j^{(1)})$$

Generalization

- How many hidden units (or, how many weights) do we need?
- How many hidden layers do we need?
- Generalization: what is the expected error on a test set?
- Causes of error
  - Network too "flexible": Too many weights compared with number of training examples
  - Network not flexible enough: Not enough weights (hidden units) to represent the desired mapping

  When comparing models, it can be helpful to compare systems with the same number of *trainable parameters* (i.e. the number of trainable weights in a neural network)
- Optimizing training set performance does not necessarily optimize test set performance....

Training / Test / Validation Data

- Partitioning the data...
  - **Training** data – data used for training the network
  - **Validation** data – frequently used to measure the error of a network on "unseen" data (e.g. after each epoch)
  - **Test** data – less frequently used "unseen" data, ideally only used once
- Frequent use of the same test data can indirectly "tune" the network to that data (e.g. by influencing choice of *hyperparameters* such as learning rate, number of hidden units, number of layers, ...)

Measuring generalisation

- Generalization Error – The predicted error on unseen data. How can the generalization error be estimated?
  - Training error?

$$E_{\text{train}} = - \sum_{\text{training set}} \sum_{k=1}^{K} t_k^n \ln y_k^n$$

  - Validation error?

$$E_{\text{val}} = - \sum_{\text{validation set}} \sum_{k=1}^{K} t_k^n \ln y_k^n$$

Cross-validation

- Optimize network performance given a fixed training set
- *Hold out* a set of data (validation set) and predict generalization performance on this set
    1. Train network in usual way on training data
    2. Estimate performance of network on validation set
- If several networks trained on the same data, choose the one that performs best on the validation set (**not** the training set)
- *n-fold* Cross-validation: divide the data into n partitions; select each partition in turn to be the validation set, and train on the remaining $(n-1)$ partitions. Estimate generalization error by averaging over all validation sets.

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 Stabilization**
  - Control the *effective flexibility*: **early stopping** and **regularization**

Structural Stabilization

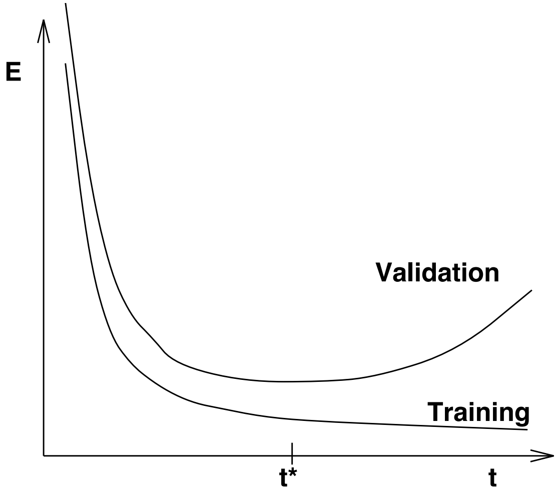
Directly control the number of weights:

- Compare models with different numbers of hidden units
- Start with a large network and reduce the number of weights by pruning individual weights or hidden units
- Weight sharing — use prior knowledge to constrain the weights on a set of connections to be equal.
  $\rightarrow$ Convolutional Neural Networks

Early Stopping

- Use validation set to decide when to stop training
- Training Set Error monotonically decreases as training progresses
- Validation Set Error will reach a minimum then start to increase

Early Stopping



Early Stopping

- Use validation set to decide when to stop training
- Training Set Error monotonically decreases as training progresses
- Validation Set Error will reach a minimum then start to increase
- Best generalization predicted to be at point of minimum validation set error
- "Effective Flexibility" increases as training progresses
- Network has an increasing number of "effective degrees of freedom" as training progresses
- Network weights become more tuned to training data
- Very effective — used in many practical applications such as speech recognition and optical character recognition

Generalisation by design

- Regularisation – penalise the weights: L1 (sparsity), L2 (weight decay)
- Dropout – randomly delete a fraction of hidden units each minibatch
- Data augmentation – generate additional (noisy) training data
- Model combination – smooth together multiple networks
- Parameter sharing – e.g. convolutional networks

To be covered in future lectures and labs...

Weight Updates

- Let $g_i(t) = \partial E / \partial w_i(t)$ be the gradient of the error function E with respect to a weight w_i at update time t
- "Vanilla" gradient descent updates the weight along the negative gradient direction:

$$\Delta w_i(t) = -\eta g_i(t)$$
$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

  Hyperparameter η - *learning rate*
- Initialise η, and update as the training progresses (learning rate schedule)

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)

Training with Momentum

$$\Delta w_i(t) = -\eta g_i(t) + \alpha \Delta w_i(t-1)$$

- $\alpha \sim 0.9$ 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

Adaptive Learning Rates

- Tuning learning rate (and momentum) parameters can be expensive (hyperparameter grid search) – it works, but we can do better
- Adaptive learning rates and per-weight learning rates
  - AdaGrad – normalise the update for each weight
  - RMSProp – AdaGrad forces the learning rate to always decrease, this constraint is relaxed with RMSProp
  - Adam – “RMSProp with momentum”

Well-explained by Andrej Karpathy at
`http://cs231n.github.io/neural-networks-3/`

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$\begin{aligned} S_i(0) &= 0 \\ S_i(t) &= S_i(t-1) + g_i(t)^2 \\ \Delta w_i(t) &= \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} g_i(t) \end{aligned}$$

  $\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors
- The update step for a parameter w_i is normalised by the (square root of) the sum squared gradients for that parameter
  - Weights with larger gradient magnitudes will have smaller effective learning rates
  - S_i cannot get smaller, so the effective learning rates monotonically decrease
- AdaGrad can decrease the effective learning rate too aggressively in deep networks

Duchi et al, http://jmlr.org/papers/v12/duchi11a.html

RMSProp

- RProp (http://dx.doi.org/10.1109/ICNN.1993.298623) is a method for batch gradient descent which uses an adaptive learning rate for each parameter and only the sign of the gradient (equivalent to normalising by the gradient)
- RMSProp is a stochastic gradient descent version of RProp (Hinton, http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) normalised by a moving average of the squared gradient – similar to AdaGrad, but replacing the sum by a moving average for S:

$$S_i(t) = \beta S_i(t-1) + (1-\beta) g_i(t)^2$$
$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} g_i(t)$$

  $\beta \sim 0.9$ is the decay rate
- Effective learning rates no longer guaranteed to decrease

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"
- Adam (Kingma & Ba, https://arxiv.org/abs/1412.6980) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha) g_i(t)$$
$$S_i(t) = \beta S_i(t-1) + (1-\beta) g_i(t)^2$$
$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingman and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Random weight initialisation

- Initialise weights to small random numbers r, sampling weights independently from a Gaussian or from a uniform distribution
    - control the initialisation by setting the mean (typically to 0) and variance of the weight distribution
- Biases may be initialised to 0
    - output (softmax) biases can be normalised to $\log(p(c))$, log of prior probability of the corresponding class c
- Calibration – variance of the input to a unit independent of the number of incoming connections ("fan-in", n_{in})
- Heuristic: $w_i \sim U(-\sqrt{1/n_{in}}, \sqrt{1/n_{in}})$ [U is uniform distribution]
    - Corresponds to a variance $\mathrm{Var}(w_i) = 1/(3n_{in})$
    - (Since, if $x \sim U(a, b)$, then $\mathrm{Var}(x) = (b-a)^2/12$ so if $x \sim U(-n, n)$, then $\mathrm{Var}(x) = n^2/3$)

Why $\mathrm{Var}(w) \sim 1/n$?

Consider a linear unit:

$$y = \sum_i w_i x_i$$

if w and x are zero-mean, then

$$\mathrm{Var}(y) = \mathrm{Var}(\sum_i w_i x_i) = n\,\mathrm{Var}(x_i)\,\mathrm{Var}(w_i)$$

if w and x are iid (independent and identically distributed)

So, if we want variance of inputs and outputs to be the same, set

$$\mathrm{Var}(w_i) = \frac{1}{n}$$

Nicely explained at `http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization`

“GlorotInit” (“Xavier initialisation”)

- We would like to constrain the variance of each layer to be $1/n_{in}$, thus

$$w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$$

- However we need to take the backprop into account, hence we would also like $\mathrm{Var}(w_i) = 1/n_{out}$
- As compromise set the variance to be $\mathrm{Var}(w_i) = 2/(n_{in} + n_{out})$
- This corresponds to Glorot and Bengio's normalised initialisation

$$w_i \sim U\left(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}\right)$$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward networks”, *AISTATS*, 2010.
http://www.jmlr.org/proceedings/papers/v9/glorot10a.html

Summary

- Basics of generalisation
- Learning rate schedules and gradient descent variants
- Initialising the weights
- **Reading**
  - Goodfellow et al, sections 5.2, 5.3, 8.3, 8.5, 7.8
  - Andrej Karpathy, CS231n notes (Stanford)
    `http://cs231n.github.io/neural-networks-3/`
- **Additional Reading**
  - Kingma and Ba, “Adam: A Method for Stochastic Optimization”, ICLR-2015
    `https://arxiv.org/abs/1412.6980`
  - Glorot and Bengio, “Understanding the difficulty of training deep feedforward networks”, AISTATS-2010.
    `http://www.jmlr.org/proceedings/papers/v9/glorot10a.html`