Learning Multi-layer Networks

Steve Renals

Machine Learning Practical — MLP Lecture 4
12 October 2016
Recap: Training multi-layer networks

\[ y_1, y_T, y_K \]

\[ h_1^{(2)}, h_k^{(2)}, h_H^{(2)} \]

\[ \delta_1^{(3)}, \delta_k^{(3)}, \delta_H^{(3)} \]

\[ w_{1k}^{(3)}, w_{kk}^{(3)}, w_{Hj}^{(3)} \]

\[ \delta_1^{(2)}, \delta_k^{(2)}, \delta_H^{(2)} \]

\[ w_{1j}^{(2)}, w_{kj}^{(2)}, w_{Hj}^{(2)} \]

\[ \delta_j^{(1)} \]

\[ w_{ji}^{(1)} \]

\[ x_i \]

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

\[ w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \eta (\delta_k^{(2)} h_j^{(1)}) \]
How to set the learning rate?
Weight Updates

- Let $D_i(t) = \partial E / \partial w_i(t)$ 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:

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

Hyperparameter $\eta$ - learning rate
- Initialise $\eta$, 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 $\eta$ 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** $\eta$ – e.g. “NewBOB”: fixed $\eta$ until validation set stops improving, then halve $\eta$ each epoch (i.e. constant, then exponential)
Training with Momentum

\[ \Delta w_i(t) = -\eta D_i(t) + \alpha \Delta w_i(t - 1) \]

- \( \alpha \sim 0.9 \) 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
AdaGrad

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

\[
S_i(0) = 0 \\
S_i(t) = S_i(t - 1) + D_i(t)^2 \\
\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t) + \epsilon}} D_i(t)
\]

$\epsilon \sim 10^{-8}$ 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 weight updates; small gradients result in larger updates
  - The effective learning rate for a parameter is forced to monotonically decrease since the normalising term $S_i$ cannot get smaller

RMSProp

- RProp ([http://dx.doi.org/10.1109/ICNN.1993.298623](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](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 $S$:

$$S_i(t) = \beta S_i(t-1) + (1-\beta)D_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} D_i(t)$$

- $\beta \sim 0.9$ 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:

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

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingman and Ba recommend \(\alpha \sim 0.9, \beta \sim 0.999\)
Coursework 1

MNIST classification, working on a standard architecture (2 hidden layers, each with 100 hidden units)

- **Part 1** Learning Rate Schedules – investigate either exponential or reciprocal learning rate schedule
- **Part 2** Training with Momentum – investigate using a gradient descent learning rule with momentum
- **Part 3** Adaptive Learning Rules – implement and investigate two of AdaGrad, RMSProp, Adam

Submit a report (PDF), along with your notebook and python code. Primarily assessed on the report which should include:

- A clear description of the methods used and algorithms implemented
- Quantitative results for the experiments you carried out including relevant graphs
- Discussion of the results of your experiments and any conclusions you have drawn
Generalisation in practice
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.
  → 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

Validation

Training

E

t*

t
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
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
Summary

- Learning rates and weight updates
- Coursework 1
- Generalisation in practice
- Reading:
  Michael Nielsen, chapters 2 & 3 of *Neural Networks and Deep Learning*
  Andrej Karpathy, CS231n notes (Stanford)