Training tips/recent advances

A brief history of Neural networks

1. McCulloch and Pitts (1943): all-or-nothing model of neurons.
2. Rosenblatt (1957): perceptron – mechanism for learning based on Hebb (1949).
3. Limitations of perceptrons: Minksy and Papert (1969)
4. 1973: Lighthill report led to first AI winter.
5. Backpropagation (Werbos; 1974). Popularised by Hinton et al in 1980s.
6. Limitations in hardware led to 2nd AI winter late 1990s.
7. 2012: resurgence due to hardware and some new “tricks”.
8. Computational biology now a big user (Angermueller et al 2016), e.g. protein folding (ALphaFold) and biomedicine (Ching et al 2018).
9. 2023: Neural networks everywhere...

See Schmidhuber (2015) for further history.

Why now?

1. Advances in computational hardware (GPU, CPU, TPU)
2. Some algorithmic developments, help in training
3. Advent of big data: many more samples now than ever before

Practical matters

- Like computational biology, 80% of the work is mundane but critical (data collection, cleaning, hyper-parameter selction).
- The mathematics of the systems are clearly defined; engineering them to work is a real challenge.
- Good news: many frameworks. We will use Keras (with Tensor Flow backend).
- GPU vs CPU
- Desktop vs Cloud

Classification: modifications

Three tweaks:

1. **1 HOT encoding** of training signal
2. **Softmax** function on output layer:

$$y_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$

   Converts output layer into probability distribution.
   Effectively consider another layer, with “lateral interactions” to compute sum term.

3. **Cross-entropy** loss function:

$$E = -\sum_{i=1}^{N} t_i \log(y_i)$$

   for comparing two probability distributions. Minimal ( ) when t=y. See Stone book, eq 4.45.

Additional terms

Many additions have been suggested to back-propagation to help prevent e.g. overfitting, local minima:

- Weight decay/ L2 regularization:

$$E = \frac{1}{2}(t - y)^2 + \beta \sum_i w_i^2$$

- Momentum: add history to weight changes:

$$\Delta w_{ij}(t+1) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t)$$

  where $0 < \alpha < 1$.

Several other methods available (conjugate gradient, rmsprop, adam) that improve convergence.
See Ruder (2017) for overview.

Merit function

Error minimisation can also be a maximisation of a "merit function". e.g. Stone and Bray (1995) form of "self-supervised activity".

$$F = \log \frac{V}{U}$$

Recent advances

If early methods (family trees, nettalk) worked so well, why did research stop in the early 1990s?
And what made it work again?

Awareness of gradient descent

- Not strictly a new innovation, but watching the 'vanishing' gradient, or 'exploding' gradient helped.
- Avoiding saturation of hidden units. RELU helped here.
- Weight initialisation schemes suggested to help avoid saturation (Glorot and Bengio, 2010). Unsupervised pre-learning of weights.

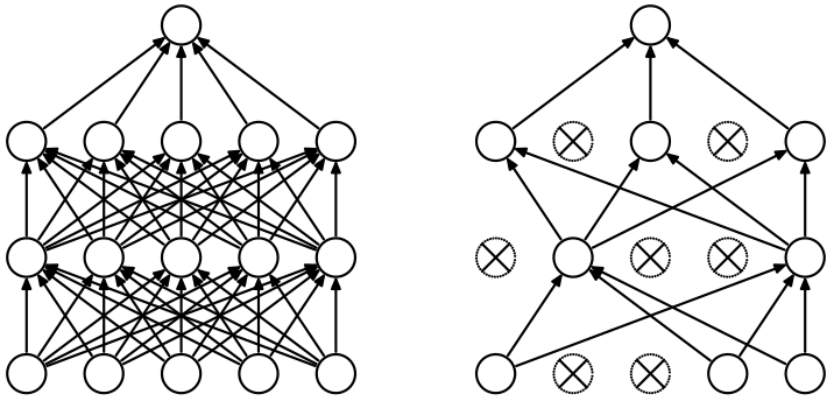
GPUs

Compute resource is improving all the time. GPUs especially useful, given parallel nature of computation.

Increased volumes of training data

- We now live in an age of data... large data sets are everywhere. Large networks needs lots of data.
- Labelled data were sparse in the 1990s, but now relatively abundant. ImageNet competition being a prime example (see later).
- Remember to clean up the input data, and try to normalise it somehow, e.g. so inputs are all scaled to same distribution (zero mean, unit variance).

Dropout (Srivastava et al 2014, fig 1)



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

- **Dropout** introduced as a way to reduce overfitting. Bank-teller analogy.
- During training, on each iteration silence some fraction of units.
- Implemented during training:
  1. once the activation of a layer calculated, set output of half of neurons to zero.
  2. Double the activation of rest of the neurons.

Hyper parameters

How do you decide on the hyper-parameters of the model?

1. Network architecture: how many hidden units? how many hidden layers?
2. Learning rates (and other parameters)
3. Error functions
4. Feature selection
5. Data set size

One approach is to overfit then regularize. Sophisticated methods exist, but often more pragmatic approaches taken.

Autograd tools

1. We would like to find $\frac{\partial E}{\partial w_i}$ for every weight in the network.
2. Autograd tools can now do the hard work for us. They do not calculate the symbolic derivative. See:
   `https://non-contradiction.github.io/autodiffr/`
   See also the Julia implementation:
   `https://www.youtube.com/watch?v=vAp6nUMrKYg`. Snippet in autograd.jl
3. `https://en.wikipedia.org/wiki/Automatic_differentiation`

Dual numbers

An 'easy' way of computing derivative for free is using dual numbers
$x = a + \epsilon b$
where $\epsilon^2 = 0$.
Demo: `autograd.jl`

Forward vs Reverse mode

- If you have few inputs (n) and many outputs (m), forward-mode is cheap and requires little memory. But you have to re-run it for each of n inputs.
- Reverse-mode is like backprop; it propagates activity forward, noting dependencies, and then fills in gradient backward. If you have only one output (m, e.g. error signal), this calculates all derivatives in one pass.
- Reverse mode is therefore more common in practice.

Summary

1. Recent developments: ReLu, DropOuts
2. More labelled training data
3. More compute
4. Autograd / Autodiff tools
5. Performant packages (pytorch/julia)