neural networks: practical considerations

Patrick van der Smagt

some good and some bad news

universal approximation

an MLP with at least two layers of weights can approximate arbitrarily well any given mapping from one finite input space with a finite number of discontinuities to another, if we have enough hidden units [Cybenko 1989; Funahashi 1989; Hornik et al 1989, 1991; Hartman et al 1990].

parameter finding

finding the optimum set of weights $\boldsymbol{w}$ for an MLP is an NP-complete problem, i.e. it cannot be solved exactly within a time $t \propto |\boldsymbol{w}|^x$ [Blum et al., 1992].

how did deep learning evolve?

In the 1990's, FFNN's have been used in many areas, including process control, classification, stock exchange prediction, diagnosis, robot control, sensory data fusion, etc. But they did not live up to their expectation, and after their hype (1986–1995) they lost in popularity.

In fact, their restricted representation and generalisation properties was one of the biggest problems.

$\rightarrow$ IEEE journals, ICML had policies not to accept NN papers

Recent results with deep neural networks have shown that the don't-give-up approach can give great results. Neural networks now can be used to solve real-world problems (google speech recognition, Street View house number recognition, image recognition, ...)

a big problem: the vanishing gradient

It is usually true that

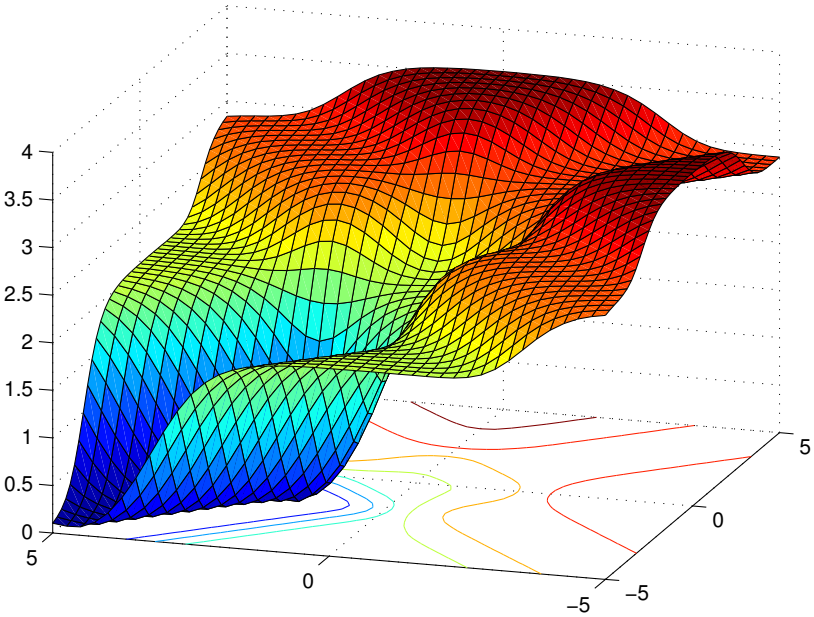
$$\partial E(\boldsymbol{w})/\partial w_{ij}^{(H)} \gg \partial E(\boldsymbol{w})/\partial w_{ij}^{(H-1)}$$

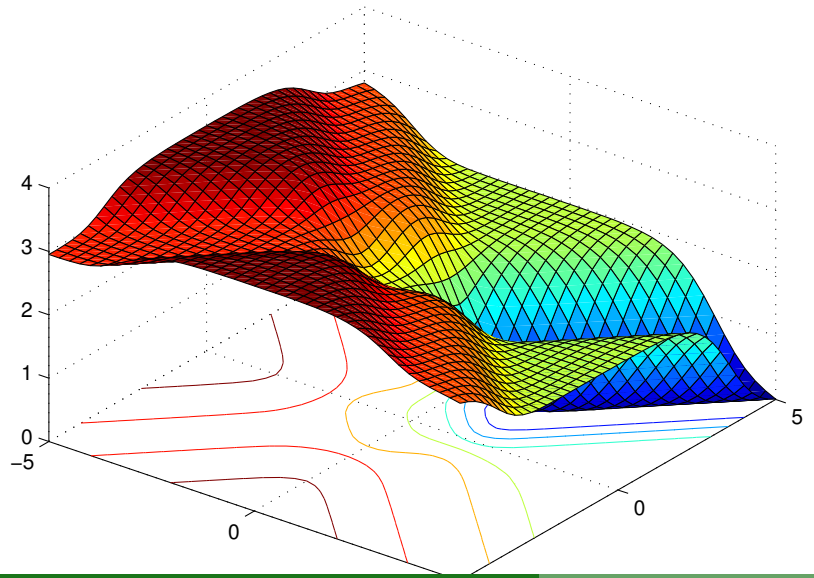
i.e., the lower you get in the network, the more the gradient vanishes.

After all,

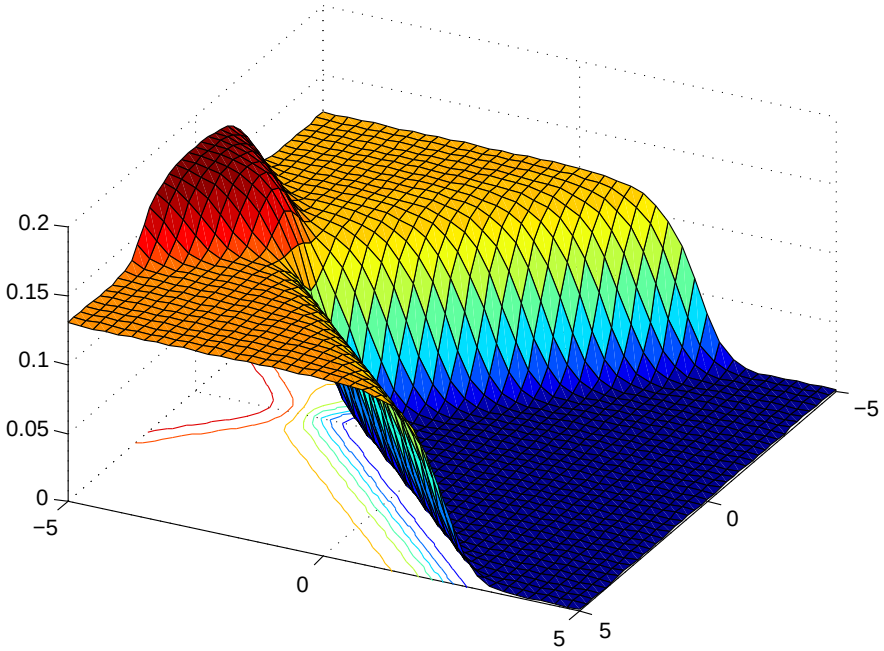
$$\frac{\partial E(\boldsymbol{w})}{\partial w_{H-1,i,j}} = \delta_{H-1,j} x_i = \sum_l \underbrace{\delta_{H,l}}_{\text{small}} \times \underbrace{w_{Hlk} x_i}_{\text{small}} = \text{smaller!}$$

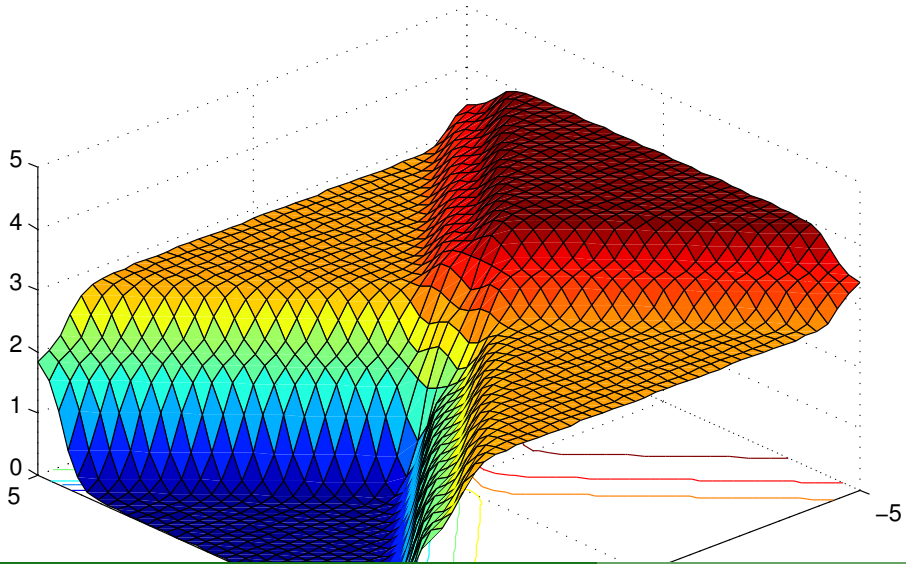
E(XOR), 1 hidden layer



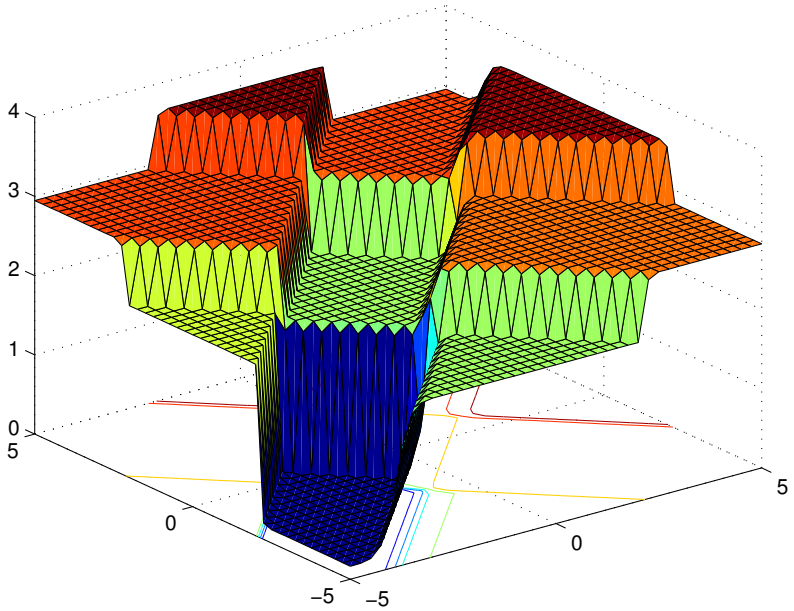


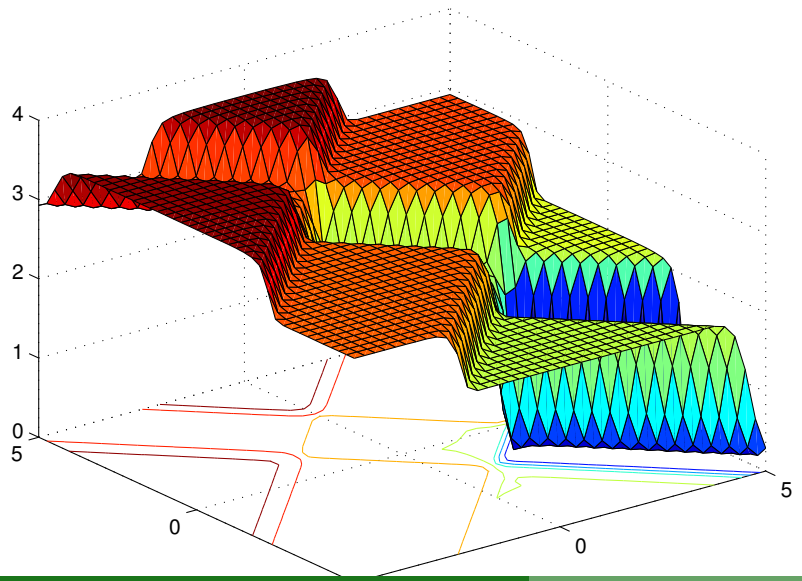
E(XOR), 3 hidden layers





E(XOR), 10 hidden layers





way out

Back-propagation is difficult for multiple hidden layers due to the vanishing gradient.

Unsupervised pretraining with *Restricted Boltzmann Machines* (Hinton et al., 2006) solved that.

However, it was later found that DNN can also be successfully trained:

- use many labelled data (e.g., now well possible for images);
- train "longer" (possible with GPUs);
- better weight initialisation (new methods were developed);
- regularise with "batch normalisation" or "layer normalisation" or "dropout".

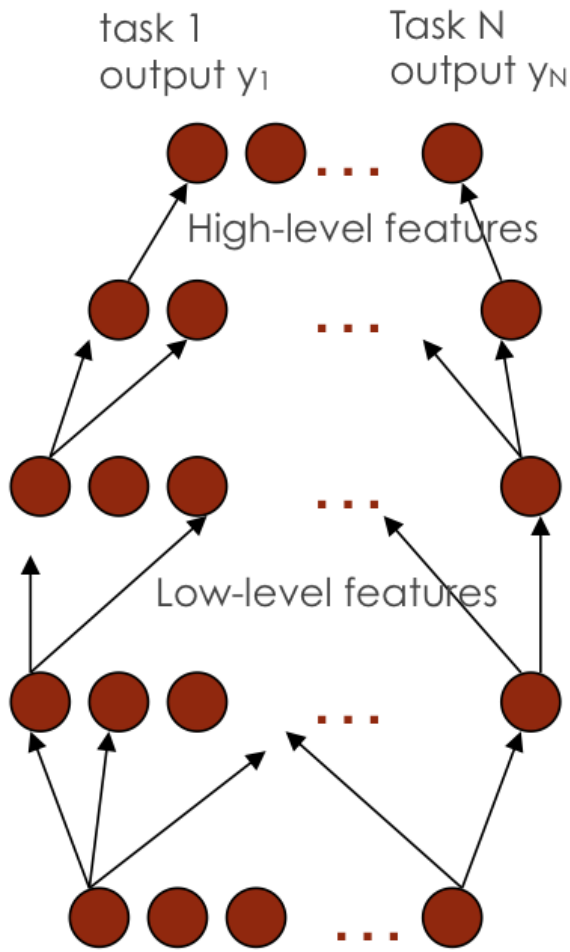
Sometimes the use of rectified linear units $\max(0, x)$ or $\log(1 + e^x)$ improve things, too.

why do multiple hidden layers improve generalisation?

Functions compactly represented with k layers may require exponential size with $k-1$ layers.

Multiple levels of latent variables allow combinatorial sharing of statistical strength.

Different high-level features share lower-level features.



from: *Understanding and Improving Deep Learning Algorithms*, Yoshua Bengio, ML Google Distinguished Lecture, 2010

rectifier linear units

Here is a trick to make logistic neurons more powerful, but keeping the number of parameters constant:

1. make many copies of each neuron
2. all neurons have the same parameters, but have a fixed offset to the bias: -1, -0.5, 0.5, ...

$$\sum_{n=1}^{\infty} \text{logistic}(x + 0.5 - n) \quad \approx \quad \log(1 + e^x)$$

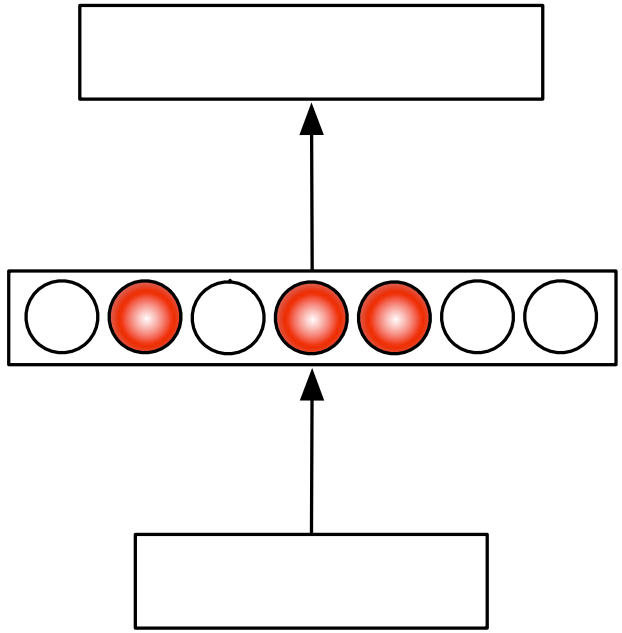
Apart from saving parameters, this also reduces the vanishing gradient.

(but there is no guarantee that this improves things)

dropout

Let's look at a neural network with one hidden layer.

Each time a learning sample is learned, we randomly put to 0 each hidden unit with probability 0.5.



We are therefore randomly sampling from 2^H different architectures, but these share the same weights.

dropout: how is the output computed?

To compute the output, one could average all possible models.

That is too expensive, however.

Instead we take the half of the outgoing weights to get the same results.
This computes the geometric mean of the predictions of all 2^H models.

Quote

if your deep neural network is overfitting, use dropout.
If it isn't, it will probably be too small!.

algorithm for backprop ("on-line" aka "stochastic" learning)

back-propagation algorithm:

```
initialise the weights
repeat
  for each training sample (x, z) do
  begin
     o = M(w, x); forward pass
     calculate error z − o at the output units
     for all w^(2) compute δ_{w^(2)}; backward pass
     for all w^(1) compute δ_{w^(1)}; backward pass continued
     update the weights using ∂E/∂w_ij = δ_j φ'(·) x_i
  end
  (this is called one epoch)
until stopping criterion satisfied
```

What is wrong with this learning method?

better algorithm for backprop ("(mini) batch learning")

back-propagation algorithm:

initialise the weights
repeat
  randomly select samples $(\boldsymbol{x}, \boldsymbol{z})$ from a mini–batch do
  begin
    $\boldsymbol{o} = \mathcal{M}(\boldsymbol{w}, \boldsymbol{x})$; forward pass
    calculate error $\boldsymbol{z} - \boldsymbol{o}$ at the output units
    for all $w^{(2)}$ compute $\delta_{w^{(2)}}$; backward pass
    for all $w^{(1)}$ compute $\delta_{w^{(1)}}$; backward pass continued
    sum the delta weights using $\partial E / \partial w_{ij} = \delta_j \phi'(\cdot) x_i$
  end
  update the weights using summed delta weights
until stopping criterion satisfied

initialising the weights

If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.

- So they can never learn to be different features.
- We break symmetry by initialising the weights to have small random values.

If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.

- We generally want smaller incoming weights when the fan-in is big, so initialise the weights to be proportional to $\sqrt{(\text{fan-in})}$.

We can also scale the learning rate the same way.

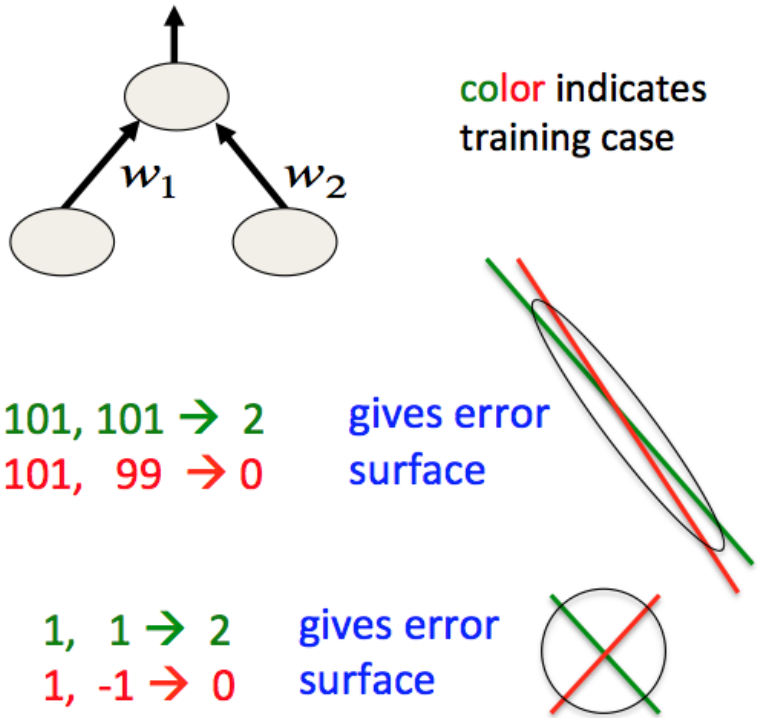
shifting the inputs

When using steepest descent, shifting the input values makes a big difference.

- It usually helps to transform each component of the input vector so that it has zero mean over the whole training set.

The tanh() produces hidden activations that are roughly zero mean.

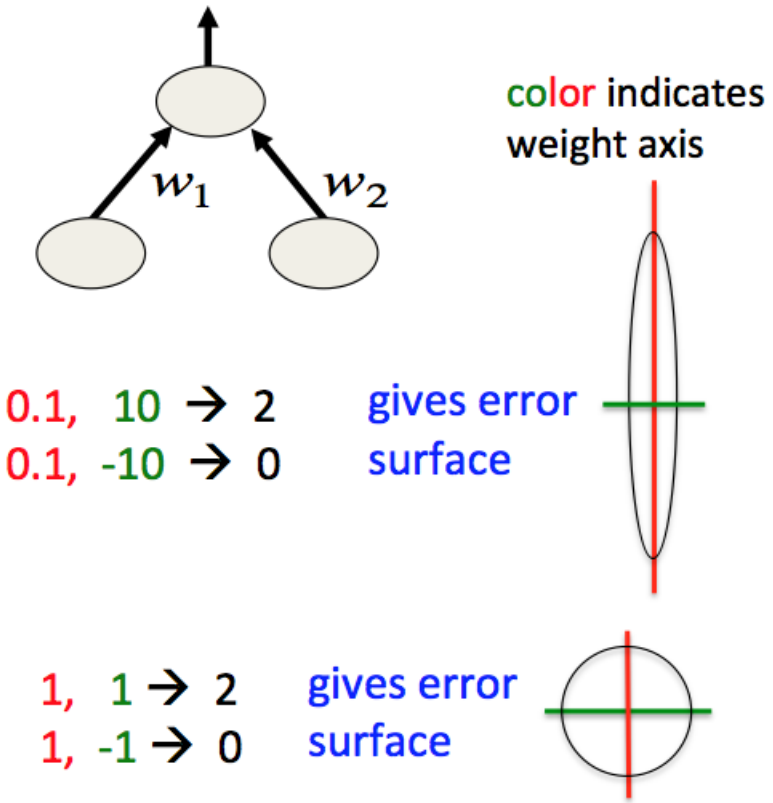
- In this respect it is better than the sigmoid.



scaling the inputs

When using steepest descent, scaling the input values makes a big difference.

- It usually helps to transform each component of the input vector so that it has unit variance over the whole training set.



A more thorough method: Decorrelate the input components

For a linear neuron, we get a big win by decorrelating each component of the input from the other input components.

There are several different ways to decorrelate inputs. A reasonable method is to use Principal Components Analysis (PCA).

- Drop the principal components with the smallest eigenvalues.
- Divide the remaining principal components by the square roots of their eigenvalues. For a linear neuron, this converts an axis aligned elliptical error surface into a circular one.

For a circular error surface, the gradient points straight towards the minimum.

speed-up tricks

If we start with a very large learning rate, weights get very large and one suffers from "saturation" in the neurons. This leads to a vanishing gradient, and learning is stuck.

Furthermore,

1. use momentum (this will be explained in the "optimisation" lecture);
2. use separate learning rates per parameter;
3. rmsprop;
4. use a second-order method (also here, see the "optimisation" lecture).

separate adaptive learning rates

In a multilayer net, the appropriate learning rates can vary widely between weights:

- The magnitudes of the gradients are often very different for different layers, especially if the initial weights are small.
- The fan-in of a unit determines the size of the "overshoot" effects caused by simultaneously changing many of the incoming weights of a unit to correct the same error.

So use a global learning rate (set by hand) multiplied by an appropriate local gain that is determined empirically for each weight.

how to determine the individual learning rates

start with a local gain of 1 for every weight.
Increase the local gain if the gradient for that weight does not change sign.
Use small additive increases and multiplicative decreases (for mini-batch)

- this ensures that big gains decay rapidly when oscillations start.
- If the gradient is totally random the gain will hover around 1 when we increase by plus δ half the time and decrease by times $1 - \delta$ half the time

$$\Delta w_{ij} = -\alpha g_{ij} \frac{\partial E}{\partial w_{ij}}$$

if $\left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$

then $g_{ij}(t) = g_{ij}(t-1) + 0.05$

else $g_{ij}(t) = g_{ij}(t-1) \cdot 0.95$

rprop: using only the sign of the gradient

The size of the gradient can be very different for different weights and can change during learning. Also remember the condition of the Hessian.

- This makes it hard to choose a single global learning rate.

Idea: use only the sign of the gradient.

- The weight updates are all of the same magnitude.
- This escapes from plateaus with tiny gradients quickly.

rprop: This combines the idea of only using the sign of the gradient with the idea of adapting the step size separately for each weight.

- Increase the step size for a weight multiplicatively (e.g. times 1.2) if the signs of its last two gradients agree.
- Otherwise decrease the step size multiplicatively (e.g. times 0.5).
- Don't make the step size too large or too small.

rmsprop: a mini-batch version of rprop

rprop is equivalent to using the gradient but also dividing by the size of the gradient.

The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?

rmsprop: Keep a moving average of the squared gradient for each weight

$$\text{meanSquare}(w,t) = 0.9\,\text{meanSquare}(w,t-1) + 0.1\left(\frac{\partial E}{\partial w}(t)\right)^2$$

Dividing the gradient by $\sqrt{\text{meanSquare}(w,t)}$ makes the learning work much better.

summary of learning methods

For small datasets (e.g., 10,000 cases) or bigger datasets without much redundancy, use a full-batch method.

- conjugate gradient (with Powell restarts), LBFGS, ...
- adaptive learning rates, rprop, ...

For big, redundant datasets use mini-batches.

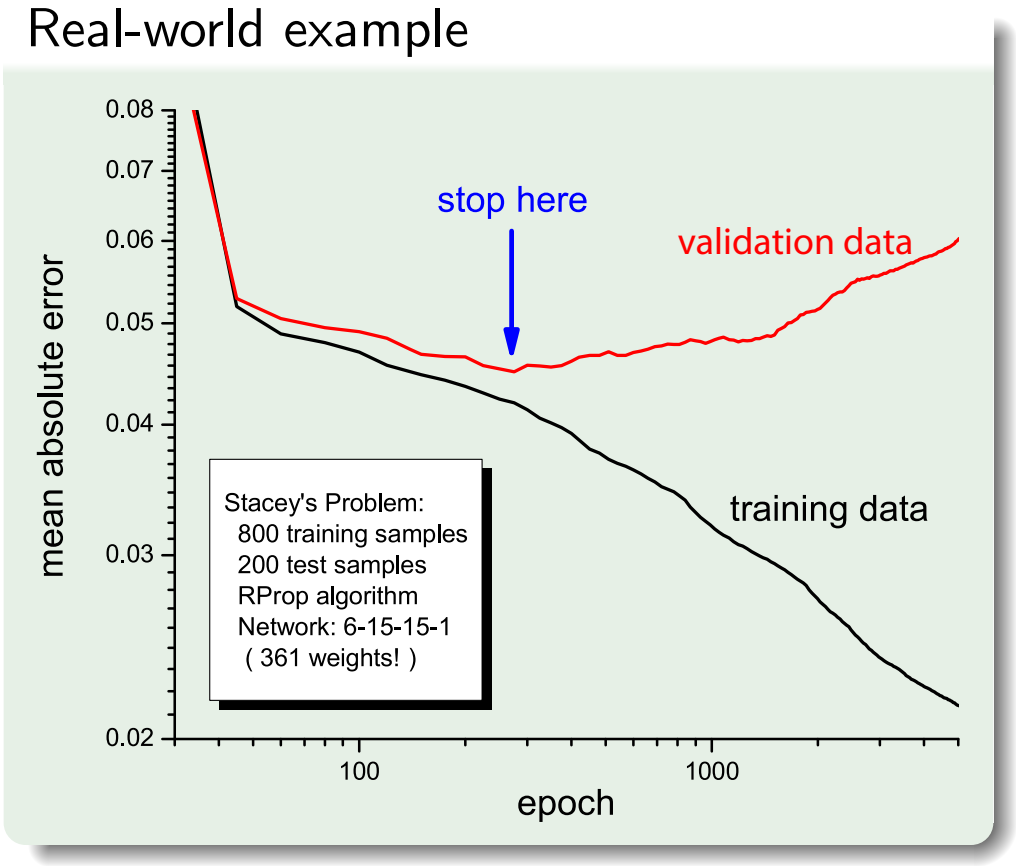
1. Try Adam, Adadelta;
2. Try rmsprop (with momentum?);
3. Try gradient descent with momentum.

Why there is no simple recipe:

- Neural networks differ a lot: Very deep networks (especially ones with narrow bottlenecks); Recurrent networks; Wide shallow networks.
- Tasks differ a lot: Some require very accurate weights, some don't; Some have many very rare cases (e.g., words).

early stopping

Early stopping is the standard regularisation technique for MLPs.



Note: Typically, use about 10% to 30% of the data for cross-validation.

early stopping recipe

Note: You also need to reserve data for testing! So the recipe is like this:

early stopping:

split the data D in D_L, D_X, D_T
repeat
  train the network one epoch on D_L
  cross-validating: test network on D_X
until D_X-loss did not decrease anymore for long
select the weights with the lowest D_X
testing: report the error on D_T

Typically we use 60% for D_L / 30% for D_X / 10% for D_T.

hyperparameter optimisation

getting the last out of your NN, you need to optimise:

- number of hidden layers (1, 2, 3, ...)
- number of hidden units (50, 100, 200, ...)
- type of activation function (sigmoid, tanh, rectifier, ...)
- learning method (adam, adadelta, rprop, ...)
- learning parameters
- data set split
- data preprocessing
- ...

We often start finding these by "playing around" with some reasonable estimates.
In the end, one often uses hyperparameter optimisation to find a good set.
Random search or Bayesian Optimisation are both viable candidates.

FFNN for classification

Take a set of classification data $\mathcal{C}_0 = \{(\boldsymbol{x}, z = 0)\}$ and $\mathcal{C}_1 = \{(\boldsymbol{x}, z = 1)\}$

If we take an FFNN with sigmoidal outputs

$$y = \phi(a) = \frac{1}{1 + \exp(-a)}$$

we can interpret the output $y(\boldsymbol{x}, \boldsymbol{w})$ as the conditional probability $p(\mathcal{C}_0 \mid \boldsymbol{x})$ while $p(\mathcal{C}_1 \mid \boldsymbol{x}) = 1 - y(\boldsymbol{x}, \boldsymbol{w})$.

But then the likelihood is a Bernoulli distribution

$$p(t \mid \boldsymbol{x}, \boldsymbol{w}) = y(\boldsymbol{x}, \boldsymbol{w})^z \big[1 - y(\boldsymbol{x}, \boldsymbol{w})\big]^{1-z}$$

The negative log likelihood then leads to

$$E(\boldsymbol{w}) = -\sum_{p=1}^{n} \Big\{ z_p \log y(\boldsymbol{x}_p, \boldsymbol{w}) + (1 - z_p) \log \big[1 - y(\boldsymbol{x}_p, \boldsymbol{w})\big] \Big\}$$

Has been shown to lead to better results!

use one-hot encoding

When classifying data, use one output per class: *one-hot encoding*.

Imagine the other case. If one encodes class 1 as $z = 0$ and class 2 by $z = 1$, then the continuous output is difficult to interpret. What does $z = 0.5$ mean?

This gets even worse with 3, . . . classes.

We prefer to use a one-hot encoding! Following the previous slide, each output encodes the likelihood of $\boldsymbol{x}$ belonging to that class.

How I train my neural network

1. get and scale the data, get a neural network that I hope is too wide and too deep, but I still want it to keep it small enough to train fast.
2. train the neural network to see if it can approximate the training data. Plot the reconstruction on your training data. Make it overfit as well as possible. If you don't reach that stage, make your network deeper, wider, stronger, and fiddle with adam vs adagrad vs. nadam vs. . . . the whole shebang. If you still fail, your data or your underlying problem is wrong.
3. now that I know you can represent the data in the NN, I try to represent the underlying function in the NN. By optimising towards the validation set. So, make the network less strong, fiddle with activation functions, initialisation, and I add regularisation, until the results are good enough on the validation set. Hyperparameter search may work. If all fails, there may be too few training data.
4. If all is good, I keep my fingers crossed for the test set. If there the results are not to my liking, I need to get more data / shuffle data better (between train and validation set). After trying with a few different seeds, knowing that that is cheating. In the end, I despair.