Backpropagation

Anti Alman

University of Tartu

23.09.2014

Learning

Perceptron
- Weights get closer to a good set of weights
- Average of two good solutions may be a bad solution in more complex networks
- Not used in „multi-layer“ neural networks

Alternative
- Show that output gets closer to the target
- Individual weights may get worse
- Example – linear neuron with squared error measure

Linear neurons

Also called linear filters

The neuron has a real-valued output which is a weighted sum of its inputs

The aim of learning is to minimize the error summed over all training cases.

$$\underset{\substack{\Big\uparrow\\\text{neuron's}\\\text{estimate of the}\\\text{desired output}}}{y} = \sum_i w_i x_i = \overset{\substack{\text{weight}\\\text{vector}\\\Big\downarrow}}{\mathbf{w}}^T \underset{\substack{\Big\uparrow\\\text{input}\\\text{vector}}}{\mathbf{x}}$$

Learning the weights

Basic idea

- Change weights proportionally to the error we made
- Change each weight proportionally to corresponding input value

Learning the weights

Define the error as the squared residuals summed over all training cases:

$$E = \tfrac{1}{2} \sum_{n \in training} (t^n - y^n)^2$$

Now differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \tfrac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n} = -\sum_n x_i^n (t^n - y^n)$$

The batch **delta rule** changes the weights in proportion to their error derivatives summed over all training cases

$$\mathrm{D} w_i = -\mathit{e} \frac{\partial E}{\partial w_i} = \sum_n \mathit{e}\, x_i^n (t^n - y^n)$$

Toy example

Each day you buy fish, chips, and ketchup

- several portions of each

The cashier **only** tells you the total price of the meal

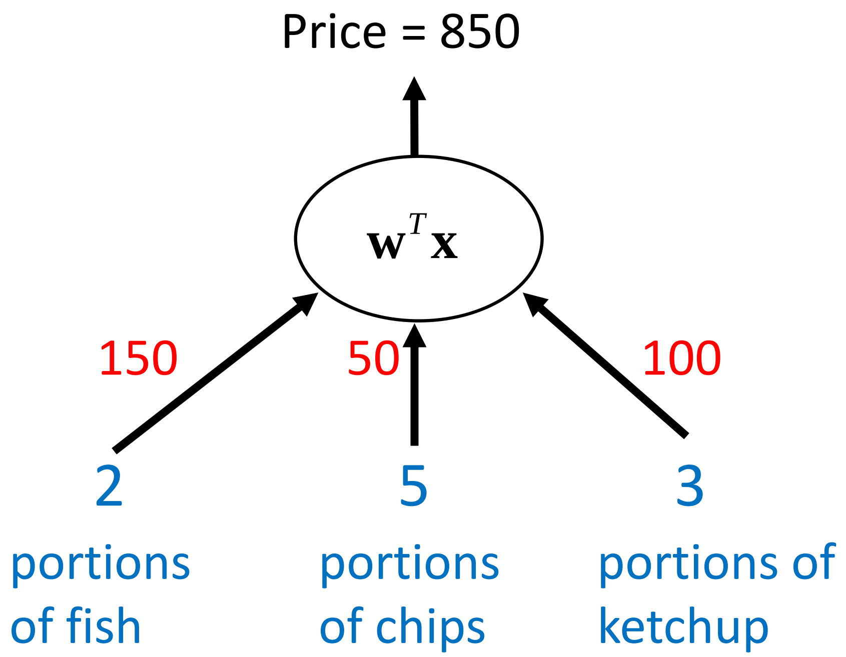
- $price = x_{fish} w_{fish} + x_{chips} w_{chips} + x_{ketchup} w_{ketchup}$

After several days, you can figure out the price of each portion

- $\mathbf{w} = (w_{fish}, w_{chips}, w_{ketchup})$

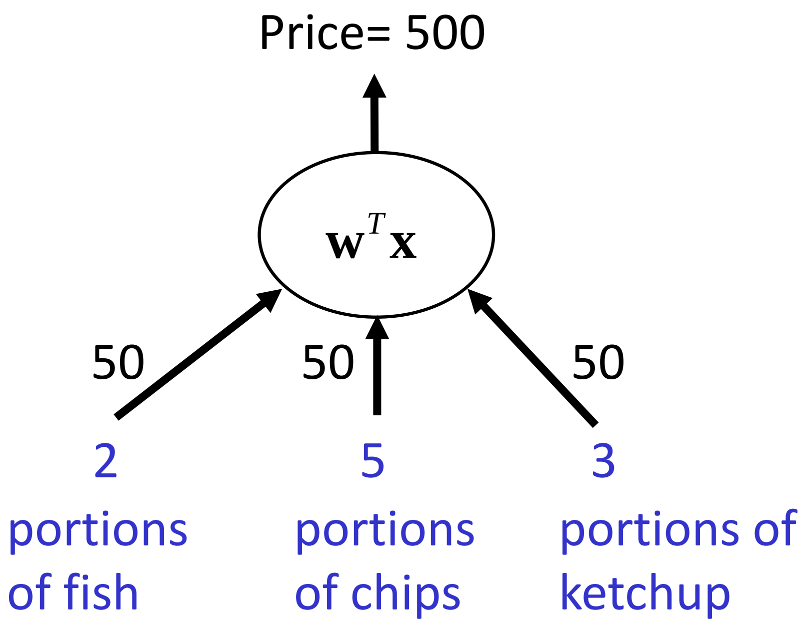
Toy example

Lets say that the true weights and our purchase is the following



Toy example

Start with a guess that everything costs 50



We use the delta rule

- $\Delta w_i = \varepsilon x_i(t - y)$
- learning rate ε = 1/35

The weight changes

- +20, +50, +30

New weights

- 70, 100, 80

Prediction with new weights

- 880
- Closer to target even tough weight for chips got worse

Behavior of the learning procedure

Does the learning procedure eventually get the right answer?

- There may be no perfect answer.
- By making the learning rate small enough we can get as close as we desire to the best answer.

How quickly do the weights converge to their correct values?

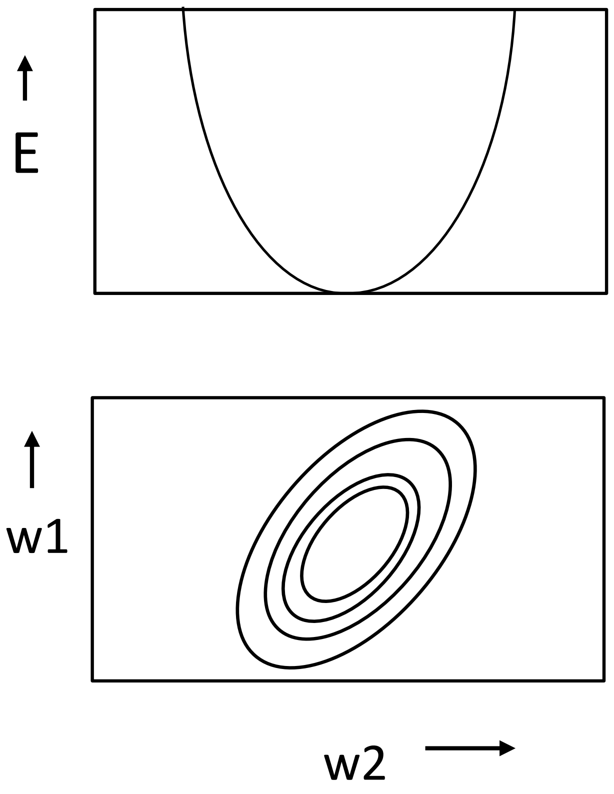
- It can be very slow if two input dimensions are highly correlated.
- If you almost always have the same number of portions of ketchup and chips, it is hard to decide how to divide the price between ketchup and chips.

The error surface

The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error

- For a linear neuron with a squared error, it is a quadratic bowl
- Vertical cross-sections are parabolas
- Horizontal cross-sections are ellipses

For multi-layer, non-linear nets the error surface is much more complicated.

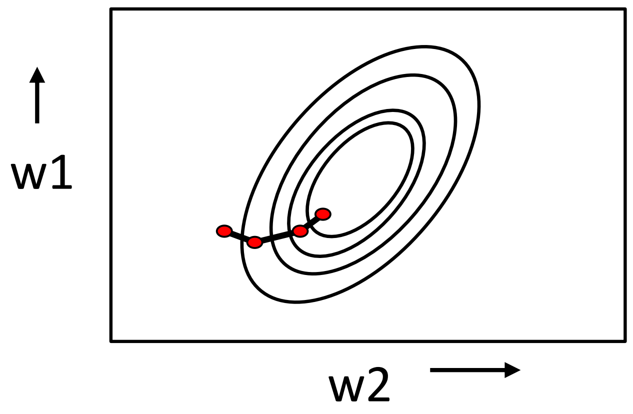


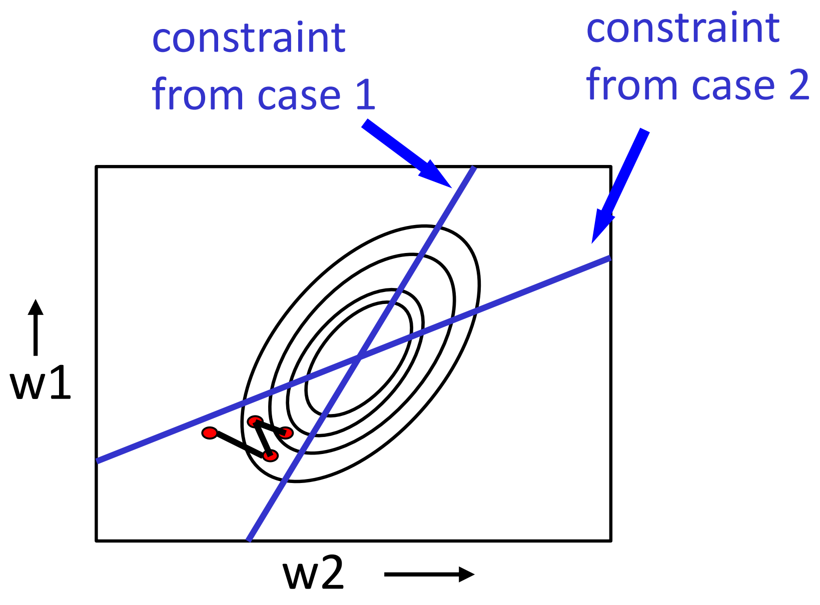
Online versus batch learning

The simplest kind of batch learning does steepest descent on the error surface

- This travels perpendicular to the contour lines.

The simplest kind of online learning zig-zags around the direction of steepest descent

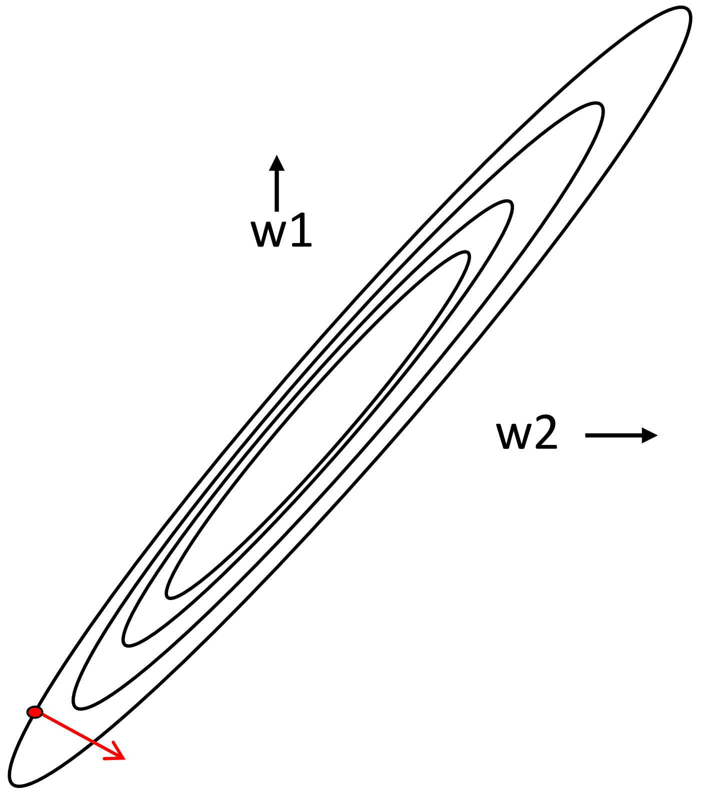




Why learning can be slow

If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!

- The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
- This is just the opposite of what we want.



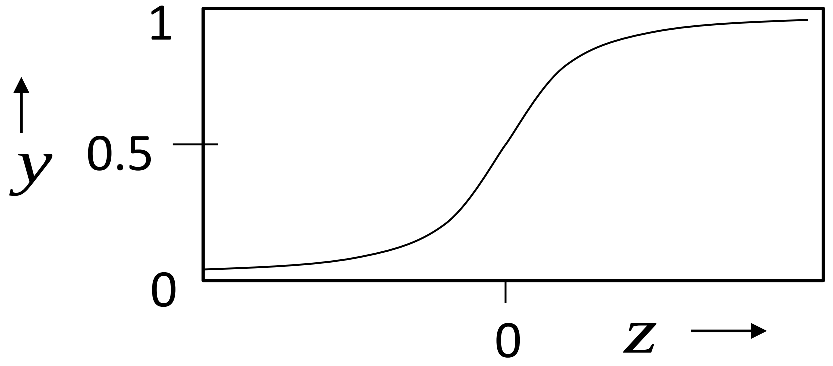
Logistic neurons

These give a real-valued output that is a smooth and bounded function of their total input

- They have nice derivatives which make learning easy

$$z = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



The derivatives of a logistic neuron

The derivatives of z with respect to the inputs and the weights

$$z = b + \sum_i x_i w_i$$

$$\frac{\partial z}{\partial w_i} = x_i \qquad \frac{\partial z}{\partial x_i} = w_i$$

The derivative of the output with respect to the logit

$$y = \frac{1}{1 + e^{-z}}$$

$$\frac{dy}{dz} = y\,(1 - y)$$

Derivative of the output with respect to each weight

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{dy}{dz} = x_i\, y\,(1-y)$$

delta-rule

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E}{\partial y^n} = -\sum_n \boxed{x_i^n}\,\boxed{y^n\,(1-y^n)}\,\boxed{(t^n-y^n)}$$

extra term = slope of logistic

Backpropagation Intuitive understanding

How to adjust the weights?

For output neuron the desired and target output is known, so the adjustment is simple

For hidden neurons

- Intuitively: if a hidden neuron is connected to output with large error, adjust its weights a lot, otherwise don’t alter the weights too much
- Mathematically: weights of a hidden neuron are adjusted in direct proportion to the error in the neuron to which it is connected

The idea behind backpropagation

We don’t know what the hidden units ought to do

- We can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t. hidden activities to train the hidden units
- Each hidden activity can affect many output units and can therefore have many separate effects on the error

We can compute error derivatives for all the hidden units efficiently at the same time

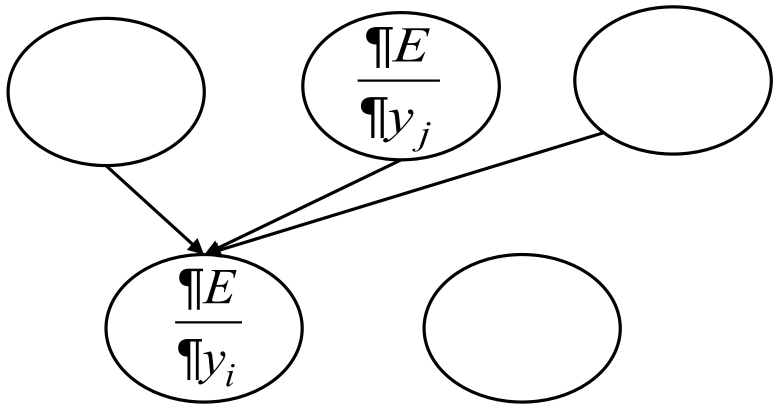
- Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit

Backpropagation on a single case

1. Input the training case
2. Convert the discrepancy between each output and its target value into an error derivative
3. Compute error derivatives in each hidden layer from error derivatives in the layer above
4. Use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights

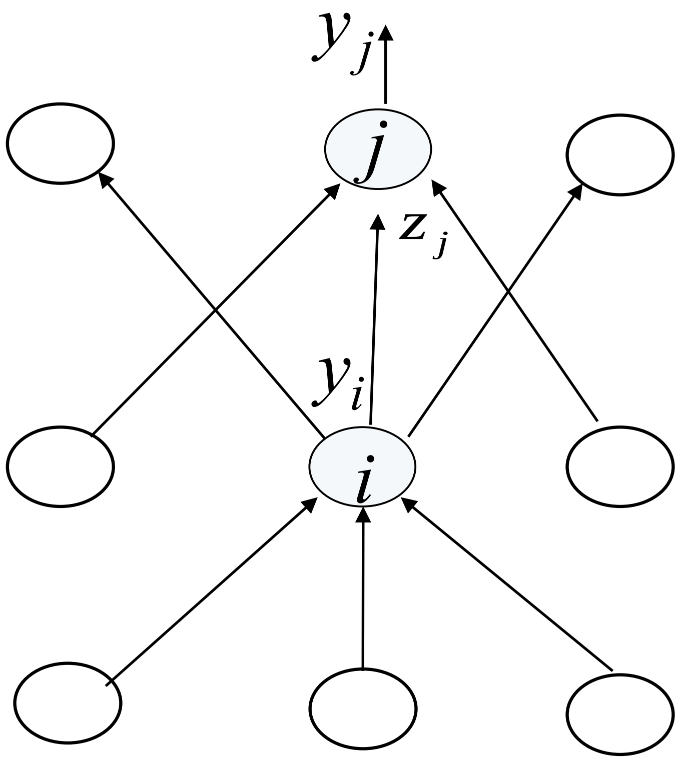
$$E = \tfrac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



$\frac{\partial E}{\partial y_j}$
$\frac{\partial E}{\partial y_i}$

Backpropagating



$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j}\frac{\partial E}{\partial y_j} = y_j\,(1-y_j)\,\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\,\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\,\frac{\partial E}{\partial z_j} = y_i\,\frac{\partial E}{\partial z_j}$$

Converting error derivatives into a learning procedure

The backpropagation algorithm is an efficient way of computing the error derivative dE/dw for every weight on a single training case.

To get a fully specified learning procedure, we still need to make a lot of other decisions about how to use these error derivatives:

- Optimization issues: How do we use the error derivatives on individual cases to discover a good set of weights?
- Generalization issues: How do we ensure that the learned weights work well for cases we did not see during training?

We now have a very brief overview of these two sets of issues.

Optimization

How often to update the weights

- Online: after each training case.
- Full batch: after a full sweep through the training data.
- Mini-batch: after a small sample of training cases.

How much to update

- Use a fixed learning rate?
- Adapt the global learning rate?
- Adapt the learning rate on each connection separately?
- Don’t use steepest descent?

Generalization

The training data contains information about the regularities in the mapping from input to output. But it also contains two types of noise.

- The target values may be unreliable (usually only a minor worry).
- There is sampling error. There will be accidental regularities just because of the particular training cases that were chosen.

When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.

- So it fits both kinds of regularity.
- If the model is very flexible it can model the sampling error really well. This is a disaster.

Learning by perturbing weights

Randomly perturb one weight and see if it improves performance. If so, save the change.

- a form of reinforcement learning
- Very inefficient
- Towards the end of learning, large weight perturbations will nearly always make things worse

Learning by using perturbations

We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes.

- we need lots of trials on each training case to “see” the effect of changing one weight through the noise created by all the changes to other weights

A better idea: Randomly perturb the activities of the hidden units

- Once we know how we want a hidden activity to change on a given training case, we can compute how to change the weights.
- There are fewer activities than weights, but backpropagation still wins by a factor of the number of neurons.

Materials

Neural Networks for Machine Learning, Lecture 3, Geoffrey Hinton

- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec3.pdf

COMP4302/5322 Neural Networks, w4, s2 2003

- http://sydney.edu.au/engineering/it/~comp4302/ann4-6s.pdf