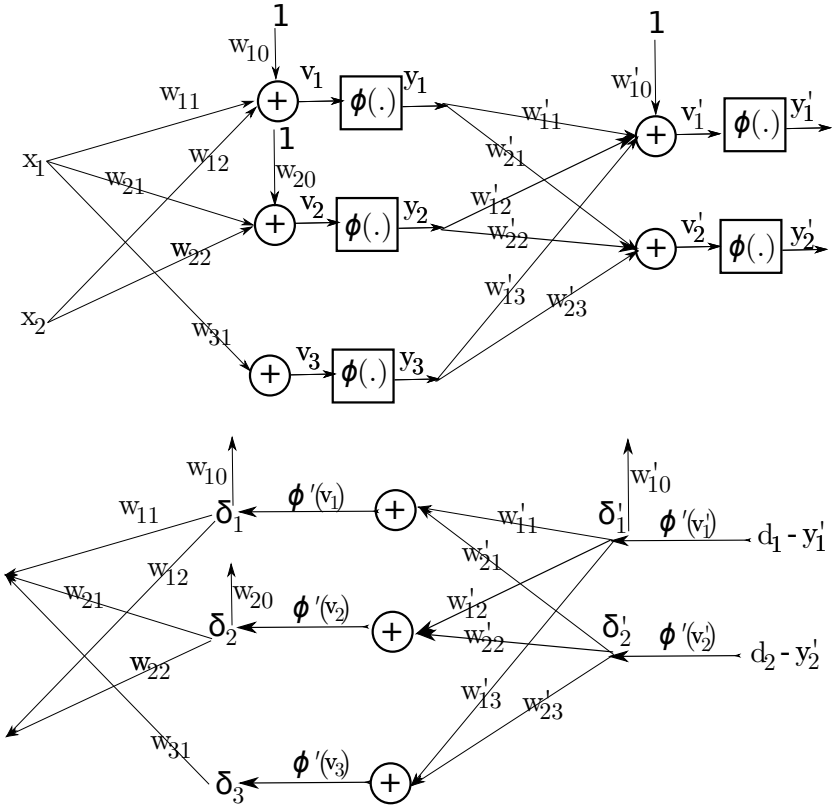
ECE/CS 559 - Neural Networks Lecture Notes : The Backpropagation Algorithm

Erdem Koyuncu

1 The algorithm

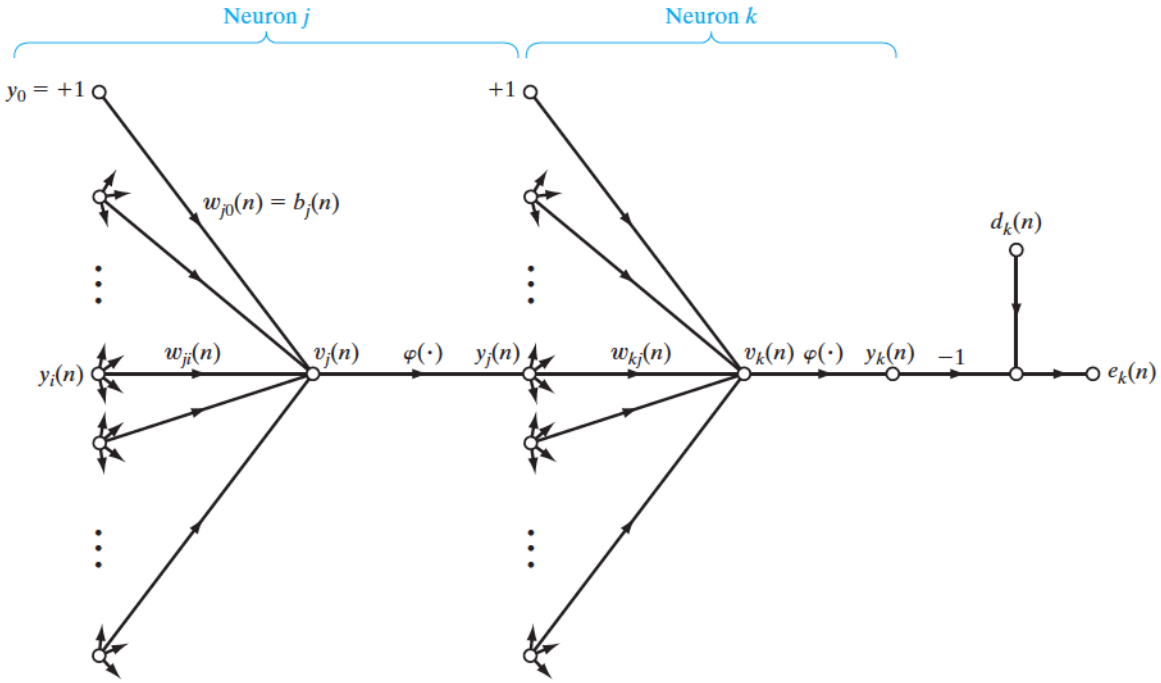
- Developed in the 1960s-1970s.
- Basically a method to calculate the partial derivatives (with respect to individual weights) of a multi-layer feedforward network.
- We begin with an example that shows how the algorithm works.



- Here we have input $\mathbf{x} = [x_1 \ x_2]^T$ and the corresponding outputs $f(\mathbf{x}, \mathbf{w}) \triangleq \mathbf{y}' \triangleq [y'_1 \ y'_2]^T$. We wish to calculate the derivatives of $E = \frac{1}{2}\|\mathbf{d} - \mathbf{y}'\|^2$ with respect to the weights.
- First, we feed pattern $\mathbf{x}$ to the network and obtain all the output information as shown in the first figure.
- We then do the backpropagation as shown in the second figure.
- $\frac{\partial E}{\partial w} = -$ (the signal before multiplication by w in the feedforward network) $\times$ (the signal before multiplication by w in the feedback network).
- Note that we would then do the update $w \leftarrow w - \eta \frac{\partial E}{\partial w}$ in the gradient descent.

2 Why does it work?

- We simply use the chain rule of calculus for the calculation. The algorithm just reveals itself.
- Chain Rule: $\frac{\partial f(g(x))}{\partial x} = f'(g(x))g'(x)$.
- Now consider the final two layers of the network (with a different notation):



(Image taken from the course book (Haykin))

- Let us calculate the partial derivative for any one of the weights connecting a neuron in the second last layer to the last layer. We have

$$\frac{1}{2}\frac{\partial E}{\partial w_{k'j'}} = \frac{1}{2}\frac{\partial \sum_{k\in K}(d_k - y_k)^2}{\partial w_{k'j'}} \tag{1}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial y_k}{\partial w_{k'j'}} \tag{2}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial \phi(\sum_{j\in J} w_{kj}y_j)}{\partial w_{k'j'}} \tag{3}$$

$$= -\sum_{k\in K}(d_k - y_k)\phi'(v_k)\frac{\partial \sum_{j\in J} w_{kj}y_j}{\partial w_{k'j'}} \tag{4}$$

$$= -\sum_{k\in K}(d_k - y_k)\phi'(v_k)y_{j'}\mathbf{1}(k = k',\, j = j') \tag{5}$$

$$= -(d_{k'} - y_{k'})\phi'(v_{k'})y_{j'} \tag{6}$$

$$\triangleq -\delta_{k'}y_{j'} \tag{7}$$

Hence,

$$\frac{1}{2}\frac{\partial E}{\partial w_{kj}} = -e_k\phi'(v_k)y_j = -y_j\delta_k \tag{8}$$

where $\delta_k \triangleq (d_k - y_k)\phi'(v_k)$.

- Similarly, we can calculate the partial derivative for any one of the weights connecting a neuron in the third last layer to the second layer. We have

$$\frac{1}{2}\frac{\partial E}{\partial w_{ji}} = \frac{1}{2}\frac{\partial \sum_{k\in K}(d_k - y_k)^2}{\partial w_{ji}} \tag{9}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial y_k}{\partial w_{ji}} \tag{10}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial \phi(\sum_{j\in J} w_{kj}y_j)}{\partial w_{ji}} \tag{11}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\frac{\partial y_j}{\partial w_{ji}} \tag{12}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\frac{\partial \phi(\sum_{i\in I} w_{ji}y_i)}{\partial w_{ji}} \tag{13}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)w_{kj}\phi'(v_j)y_i \tag{14}$$

$$= -y_i\phi'(v_j)\sum_{k\in K}\delta_k w_{kj} \tag{15}$$

$$\triangleq -y_i\delta_j \tag{16}$$

- If there were another layer (say indexed via $z \in Z$), we would go

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\phi'(v_j)w_{ji}\phi'(v_i)y_z \tag{17}$$

$$= -y_z\phi'(v_i)\sum_{j\in J}\delta_j w_{ji} \tag{18}$$

$$\triangleq -y_z\delta_i \tag{19}$$

  and so on.

3 Matrix representation of the algorithm

- The following matrix representation is useful for computation purposes.
- Suppose we have m_0 input nodes, m_1 nodes at layer 1, m_2 nodes at layer 2, and finally m_L nodes at layer L. Let $L \geq 1$.
- The input signal is $\mathbf{y}_0 \triangleq \mathbf{x} \in \mathbb{R}^{m_0\times 1}$.
- Neural network weight matrices $\mathbf{W}_\ell \in \mathbb{R}^{m_\ell \times (m_{\ell-1}+1)}$ including the biases.
- Then, the induced local fields are $\mathbf{v}_\ell \triangleq \mathbf{W}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix} \in \mathbb{R}^{m_\ell \times 1}$, $\ell = 1, \ldots, L$.
- And the outputs are $\mathbf{y}_\ell \triangleq \phi(\mathbf{v}_\ell)$, $\ell = 1, \ldots, L$.
- Define $E = \frac{1}{2}\|\mathbf{d} - \mathbf{y}_L\|^2$, where $\mathbf{d}$ is some certain desired output vector.
- The goal is to calculate $\frac{\partial E}{\partial [\mathbf{W}_\ell]_{i,j}}$ for the purpose of using it with gradient descent.

- Backpropagation equations:

$$\begin{aligned}\boldsymbol{\delta}_L &\triangleq (\mathbf{d} - \mathbf{y}_L) \cdot \phi'(\mathbf{v}_L) \\ \boldsymbol{\delta}_\ell &= \underline{\mathbf{W}_{\ell+1}^T \boldsymbol{\delta}_{\ell+1}} \cdot \phi'(\mathbf{v}_\ell),\ \ell = 1, \ldots, L-1 \\ \frac{\partial E}{\partial \mathbf{W}_\ell} &= -\boldsymbol{\delta}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix}^T,\ \ell = 1, \ldots, L.\end{aligned}$$

- In the above equations, $\cdot$ is the Kronecker/elementwise product, i.e., for vectors $a = [a_1 \cdots a_n]^T$ and $b = [b_1 \cdots b_n]^T$, we define $a \cdot b = [a_1 b_1 \cdots a_n b_n]^T$. Also, $\underline{a}$ is the "lose my first component operator", i.e. $\underline{a} = [a_2 \cdots a_n]$. The reason you need this operator is because of the biases: once you backpropagate signals for the biases, you no longer need these signals and have to destroy them...

- Hence, the weight updates for online learning would be

$$\mathbf{W}_\ell \leftarrow \mathbf{W}_\ell + \eta \boldsymbol{\delta}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix}^T,\ \ell = 1, \ldots, L$$

.

- Note that in the expression for E, usually we have a summation over a training sequence (over different input patterns, and their corresponding desired outputs, i.e. $E = \frac{1}{2|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \|\mathbf{d}_i - \mathbf{y}_{i,L}\|^2$.)

- So, given η, for epochs = 1 to infinity, for $i = 1$ to training samples, do the forward and backward computations given the training example $\mathbf{x}_i$ as shown above, and perform the corresponding weight updates, loop until some stopping criterion is met. The criterions may be:
  - When the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.
  - When the absolute rate of change in the average squared error per epoch is sufficiently small.

4 Heuristics to make backpropagation perform better

- **Stochastic vs batch update:** Stochastic (online) mode of learning is computationally faster than batch (offline) learning. This is because, in stochastic learning, you make use of the redundancy in the training set. For example, suppose you have 1000 images, 10 copies of the same pattern, then online learning will approximately be 10x faster than offline learning. Even though real life samples do not contain identical images, a similar idea applies. So, the advantage of stochastic learning is much faster convergence on large data set that contain a lot of redundancy. Also, a stochastic trajectory of weight allows escape from local minima. The disadvantage of stochastic learning is that you typically keep bouncing around a local minima unless learning rate is reduced. Moreover, theoretical conditions for convergence are not as trivial. For batch learning, one can easily guarantee convergence to local minimum under simple conditions. But, batch learning is slow on large problems with large datasets.

- **Maximizing information content:** Use an example that results in the largest training error, or use an example that is radically different than all those previously used.

- Usually odd functions perform better than other types of activation functions. $\phi(-v) = -\phi(v)$. The tangent hyperbolic $\beta \tanh \alpha v$ satisfies this property. Some authors even propose suitable values for α, β for general purposes, e.g. $1.7159 \tanh \frac{2}{3} v$. This satisfies $\phi(1) = 1$ and $\phi(-1) = -1$. The second derivative of the function attains its maximum at 1.

- Target values. Obviously, the target values should be within range of the activation functions. In fact, there should be a certain guard interval to avoid saturation or divergence problems. In fact, if you set the target values on the asymptotes of the sigmoid, (1) the output weights will be driven to $-\infty$ or $+\infty$, (and thus saturate), (2), and therefore, if a training sample does not saturate the outputs (this can be an outlier, for example), it will produce enormous gradients due to large weights, resulting in

incorrect updates. and (3) outputs will tend to be binary even when they are wrong - a mistake that will be very difficult to correct. To avoid saturation, people sometimes pick activation functions like $\tanh x + \alpha x$.

- Normalizing the inputs: The training samples should be preprocessed so that their mean values are close to 0. E.g. if the training inputs are strictly positive, the weights of a neuron in the first hidden layer can only increase together or decrease together (not enough degree of freedom to move anywhere we like). For example, think about weights w_1, w_2 in the two dimensional plane. You can only move to the NE direction or the SW direction. If you then wanted to move towards SE or NW you can only do so by zigzagging.

  The input variables should be uncorrelated (to avoid correlations in the different weights of the neuron); this can be done via what is called principal component analysis (PCA) to be discussed later.

  The input variable should also have equal variances, so that the neurons will learn at approximately the same speed.

- Weight initialization: Large weights saturate the units, leading to small gradients and thus slow learning. Small weights correspond to a very flat are of the error surface and also lead to slow learning. For example, think about learning in a one input, one output, many neurons in a hidden layer network with all weights initialized to 0 with tanh activation function. The weights will remain as 0 except for the bias of the output neuron.

  For a heuristic for a good initialization of weights, suppose that the induced local field of neuron j is say $v_j = \sum_{i=1}^{m} w_{ji} y_i$. Suppose the inputs applied to each neuron in the net has 0 mean and variance 1, i.e. $E[y_i] = 0$, and $E[(y_i - E[y_i])^2] = E[y_i^2] = 1$. Also, assume y_i, y_k are uncorrelated, and suppose the synaptic weights are drawn from a distribution with 0 mean and variance σ^2. Then, the mean and variance of the induced local field is $E[v_j] = 0$ and $E[v_j^2] = E[\sum_i \sum_k w_{ji} w_{jk} y_i y_k] = \sum_i \sum_k E[w_{ji} w_{jk}] E[y_i y_k] = \sum_i \sigma^2 1 = m\sigma^2$. A good point to put $m\sigma^2$ is between the linear and saturation parts of the activation function. This point was point 1 for the special tanh function discussed above, so we obtain $\sigma^2 = \frac{1}{m}$.

- Learning rates. All neurons in the multilayer perceptron should ideally learn at the same rate. The last layers usually have larger local gradients than the layers at the front end of the network. Hence, the learning-rate parameter η should be assigned a smaller value in the last layers than in the front layers of the multilayer perceptron. Neurons with many inputs should have a smaller learning-rate parameter than neurons with few inputs so as to maintain a similar learning time for all neurons in the network. It is often suggested that for a given neuron, the learning rate should be inversely proportional to the square root of synaptic connections made to that neuron.

- The main problem with having too many problems is usually the vanishing or exploding gradient problems: Think about a line network with $y = f(w_1 f(w_2 f(w_3 f(w_4 x))))$ for example. To update w_4, we have $w_4 \leftarrow w_4 +$ many f's and $w_1 w_2 w_2$ product. f's are usually less than 1 and w_is are initialized to have mean less than 1: We have the vanishing gradient problem. Otherwise, we have exploding gradient.

  The real problem is the unstable gradients as a result of having many multiplications.

5 Overfitting

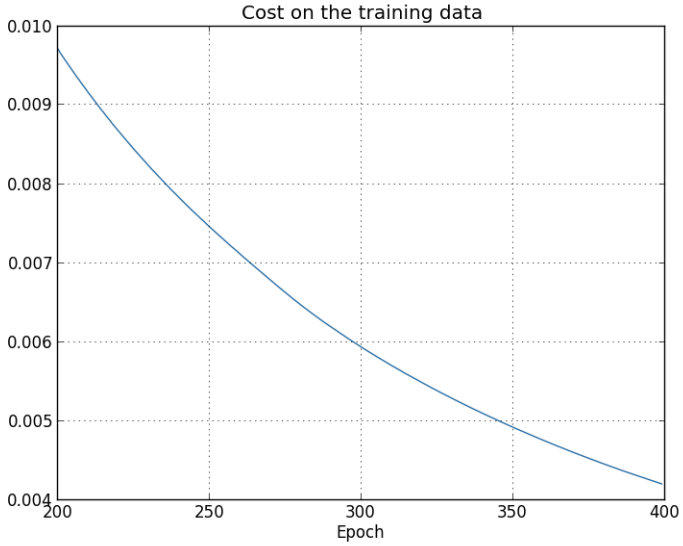
- A model with a large number of free parameters can describe a wide array of phenomena. Von Neumann: "With four parameters I can fit an elephant and with five I can make him wiggle his trunk."

- Just because the model fits so well with the available data does not make it a good model.

- For example, samples of a straight line; one intuitive way to fit it is via a straight line (which requires 2 params), while another way to fit is is via a crazy curve (if you have a sufficiently large number of

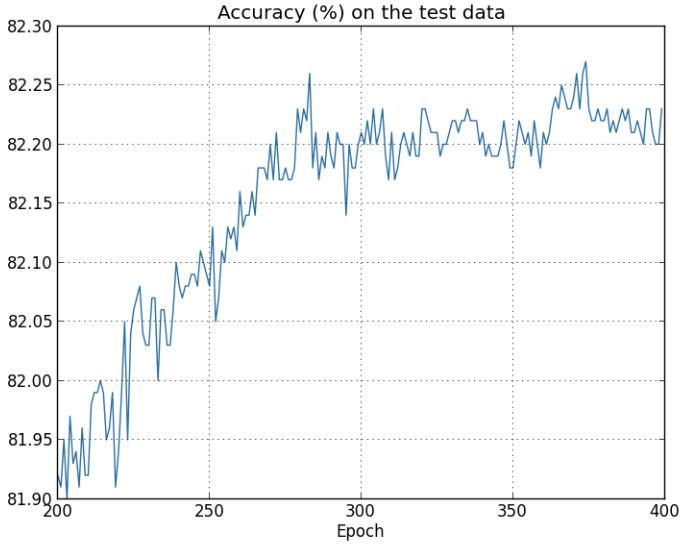
params). Occam's razor: Among competing hypotheses, the one with the fewest assumptions should be selected. It should be mentioned that sometimes a complex explanation is the correct one.

- If we for example train our neural network (with backpropagation) using a training set with a few samples (say 100s or thousand), we will see that the energy function (mean-squared error) will be smoothly decaying to zero as the epochs increase:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

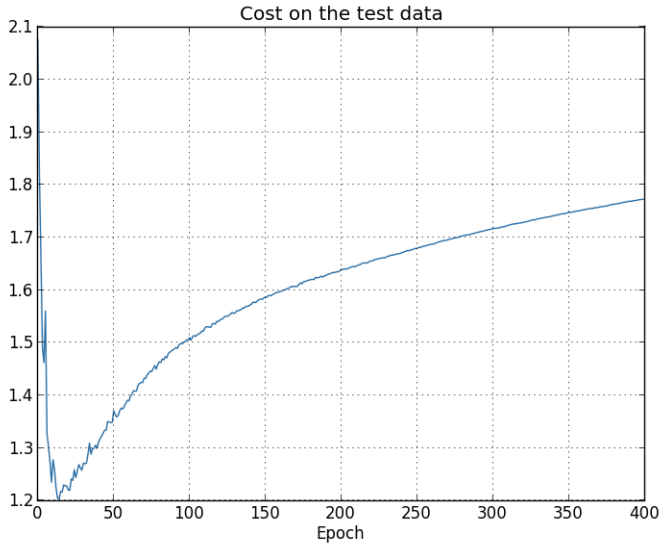
- When the resulting network is actually tested however, one observes a non-zero (and actually high) inaccuracy on the test data:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

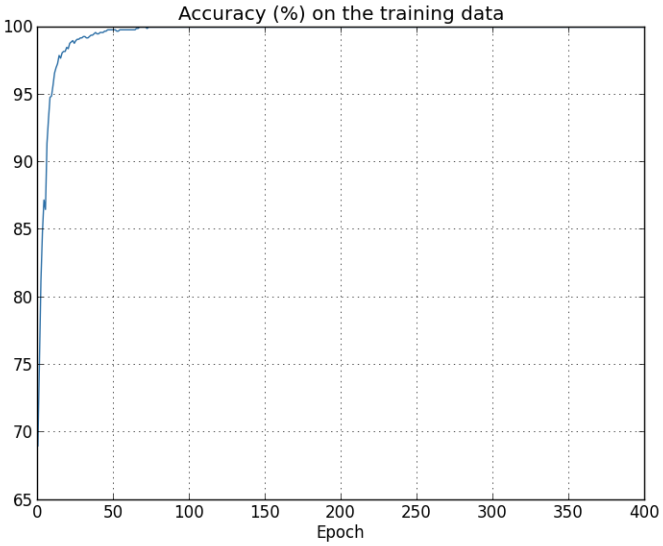
- From the training data, it appears that the network is getting better, although, on the test data the performance of the network saturates and fluctuates around a fixed value after around 280 epochs. We say that the network is overfitting after 280 epochs.

- You may think that the it is not fair to look at MSE on one graph and percent inaccuracy on another graph, but in fact, if we draw the cost on test data, we see a similar behavior:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

- Now the points where things get worse are different if we take MSE or % inaccuracy as performance metric, but since we care about % inaccuracy at the end, it makes more sense to define the point where we start overtraining as 280.
- Another sign of overfitting is when we plot % inaccuracy on the test data:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

- So our network really is learning about peculiarities of the training set (as if it is just memorizing the training set), without "understanding" the problem well-enough to solve/classify a previously-not-shown test example.
- **One of the simplest ways to prevent overfitting is to stop training when the precent errors (accuracy) in the test set stop improving.** In practice, one continues training a little more to be confident that the accuracy has in fact saturated.
- Now, say from one experiment, you got one set of weights (you made sure you prevented overfitting). You may want to play with different values of η, the initial condition, the network topology etc to further optimize the performance. These parameters are called the hyperparameters of the optimization

problem. For each hyperparameter, the achievable performance and the corresponding weights will be different. Obviously at the end, you will pick the weights that provide the lowest final classification error on the training set. On the other hand, by trying many different hyperparameters, we fall into a danger of this time overfitting with respect to the hyperparameters. That is why the available data set is usually broken down further to three parts called the training data, validation data and the test data. Usually the training, validation, test data follow a 5:1:1 scale ratio in terms of their size.

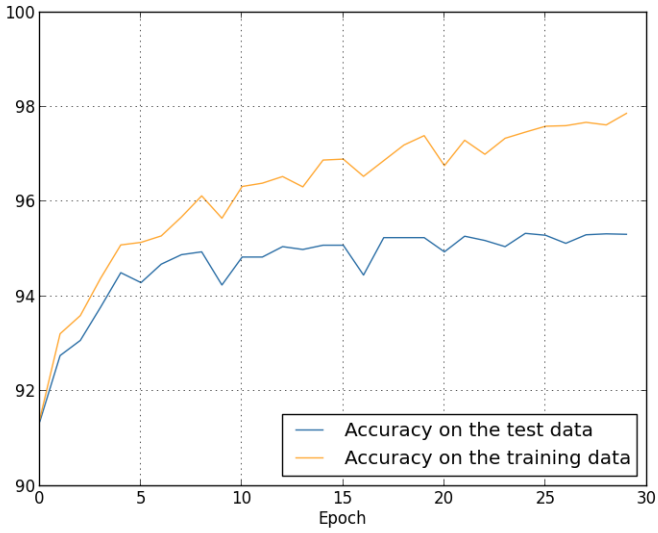
- **Cross-validation and the hold-out method:** Hence, we train the network using the training data as usual. We compute the classification accuracy on the validation data at the end of each epoch. Once the classification accuracy on the validation data has saturated/stops to get worse, we stop training. This strategy is called early stopping. We do this for different hyperparameters, and then pick the hyperparameters + weights that provide the best accuracy on the validation data. We can now use these parameters on the actual test data to see how well we perform (Now we are more confident that we do not do overfitting). **This called the hold-out method (the validation data is held out from the training data. This is thus another way to prevent overfitting**. The entire procedure is called cross-validation.

- Now what if the performance on the test data is unsatisfactory? We will go back to the drawing board, train using new hyperparameters and test on the data set. But then, at least technically, we can overfit on the test data! Fortunately, that seldom happens in practice, and the above 3-set procedure works fine. In fact, typically one does not even have enough data so that one just uses 2-set approach consisting of a training set and a test set.

- **Multi-fold cross validation:** Note that there are other variants of cross-validation. We may use multifold cross-validation by dividing the available set of N examples into K subsets, where $K > 1$; this procedure assumes that K is divisible into N. The model is trained on all the subsets except for one, and the validation error is measured by testing it on the subset that is left out. This procedure is repeated for a total of K trials, each time using a different subset for validation. The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment.There is a disadvantage to multifold cross-validation: It may require an excessive amount of computation, since the model has to be trained K times.

- **Leave-one-out method:** When the available number of labeled examples, N, is severely limited, we may use the extreme form of multifold cross-validation known as the leave-one-out method. In this case, N - 1 examples are used to train the model, and the model is validated by testing it on the example that is left out. The experiment is repeated for a total of N times, each time leaving out a different example for validation.The squared error under validation is then averaged over the N trials of the experiment.

- Now we were talking about ways to prevent overfitting. **Another way is obviously to use a larger test set:**



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

We see that there is still overfitting going on but it is on a much less serious scale: The performance with the test data and the training data are much closer. Unfortunately, training data is expensive so using a larger data set is not always an option.

- **Another way to prevent overfitting is regularization.**

6 Regularization

- We know how to determine the partial derivatives of $E(\mathbf{w})$ using the backpropagation algorithm. We then use gradient descent to update the weights.
- L_2 regularization: Consider the following function $E'(\mathbf{w}) \triangleq E(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$. The weight updates are $\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla\mathbf{E}(\mathbf{w}) - \lambda\eta\mathbf{w} = (1-\eta\lambda)\mathbf{w} - \eta\nabla\mathbf{E}(\mathbf{w})$.
- Why does this work? Suppose our network mostly has small weights, as will tend to happen in a regularized network. The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data.
- Also, small weights imply a usually lesser chance of saturation of neurons, implying better performance.
- L_1 **regularization**: $E'(\mathbf{w}) \triangleq E(\mathbf{w}) + \lambda\|\mathbf{w}\|_1$. We have $\mathbf{w} \leftarrow \mathbf{w} - \lambda\text{sgn}(\mathbf{w}) - \eta\nabla\mathbf{E}(\mathbf{w})$
- **Dropout:** Get first training sample, remove randomly half the neurons in the hidden layers (or some fraction of), forward and back propagate. Restore all connections. Get second training sample, do the same thing, again and again.

  Why does this work? We are trying to make the system robust to loss of neurons. Like brain damage.
- **Artificially expanding the training data:** Small rotations of images, elastic distortions (emulate random oscillations found in hand muscles). Or, if you are looking for a speech recognizer, you can add random noise to your voice samples to expand the training data.
- **Momentum method:** Initialize a velocity vector to 0, and then perform the updates $\mathbf{v} \leftarrow \mu\mathbf{v} - \eta\nabla E(\mathbf{w})$ with $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$. The factor $(1-\mu)$ can be thought of as the friction of the descent, you build speed as you roll down the valley, that may improve your speed of convergence (you avoid the slowness of gradient descent as you reach the global minimum), but you may overshoot if μ is too large.