Feed-forward Neural Networks

MICHAEL WOLLOWSKI

Introduction

Last time, we studied perceptrons.

They are single layer feed-forward networks.

They work well for domains which can be linearly separated

An example for which they do not work is the Boolean XOR function.

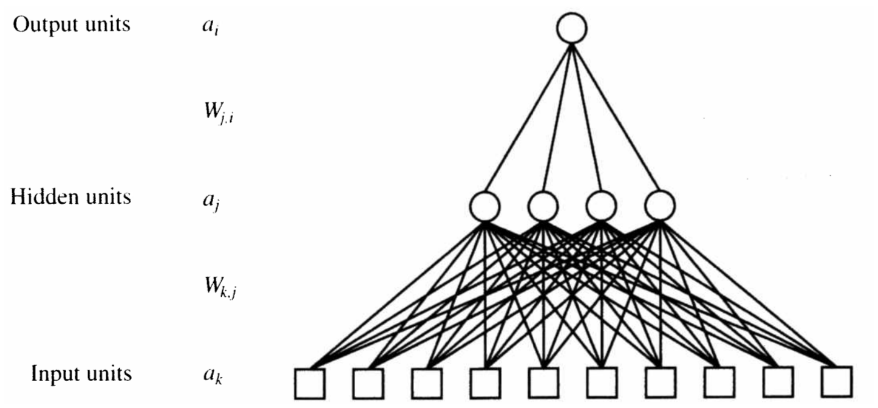
In this presentation, we will study multi-layer feed-forward networks.

Most of the concepts are the same as for perceptrons.

A key difference is in the way in which we determine errors in the non-output layers

Architecture

Here is the architecture of a basic feed-forward network.



Architecture

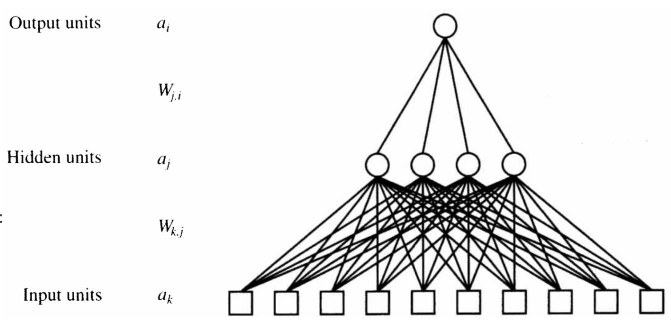
As with perceptrons, it consists of:
- input units
- an output unit
- we may have more than one output unit.
- weights between units

Additionally, there are hidden units.

We now have a weight matrix.

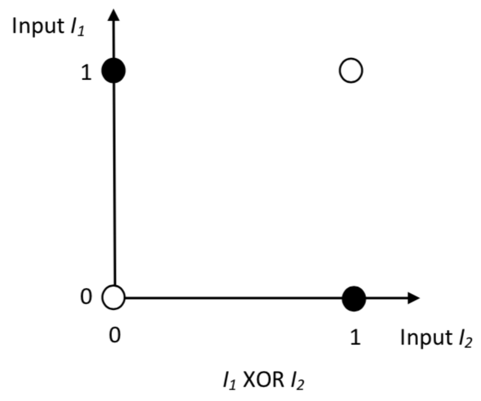
It contains the "knowledge" of the NN

The weight matrix changes as a result of training



XOR

Let’s revisit XOR and how one might design a NN that solves it



XOR Solution

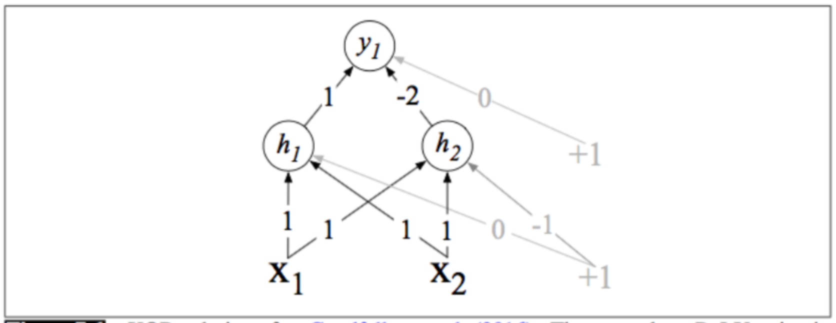


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we’ve called them h_1, h_2 (h for “hidden layer”) and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

Figure source: Jurafsky and Martin, Speech and Language Processing, 3rd Ed.

XOR Solution

The bias “nodes” can be used to adjust the threshold of activation functions.

We can safely ignore their role.
Please complete the XOR exercise posted on our website and come back when finished.

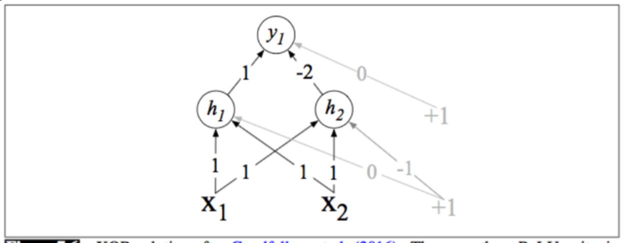


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we’ve called them h_1, h_2 (h for “hidden layer”) and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

Figure source: Jurafsky and Martin, Speech and Language Processing, 3rd Ed.

XOR Solution

Let’s have a look at what hidden layers do for us.

Consider the two figures.

The one on the left shows the value space of the original XOR problem.

The one on the right shows how the hidden layer produces a value space that is linearly separable.

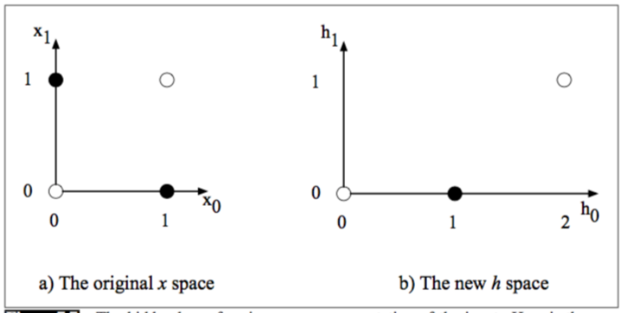
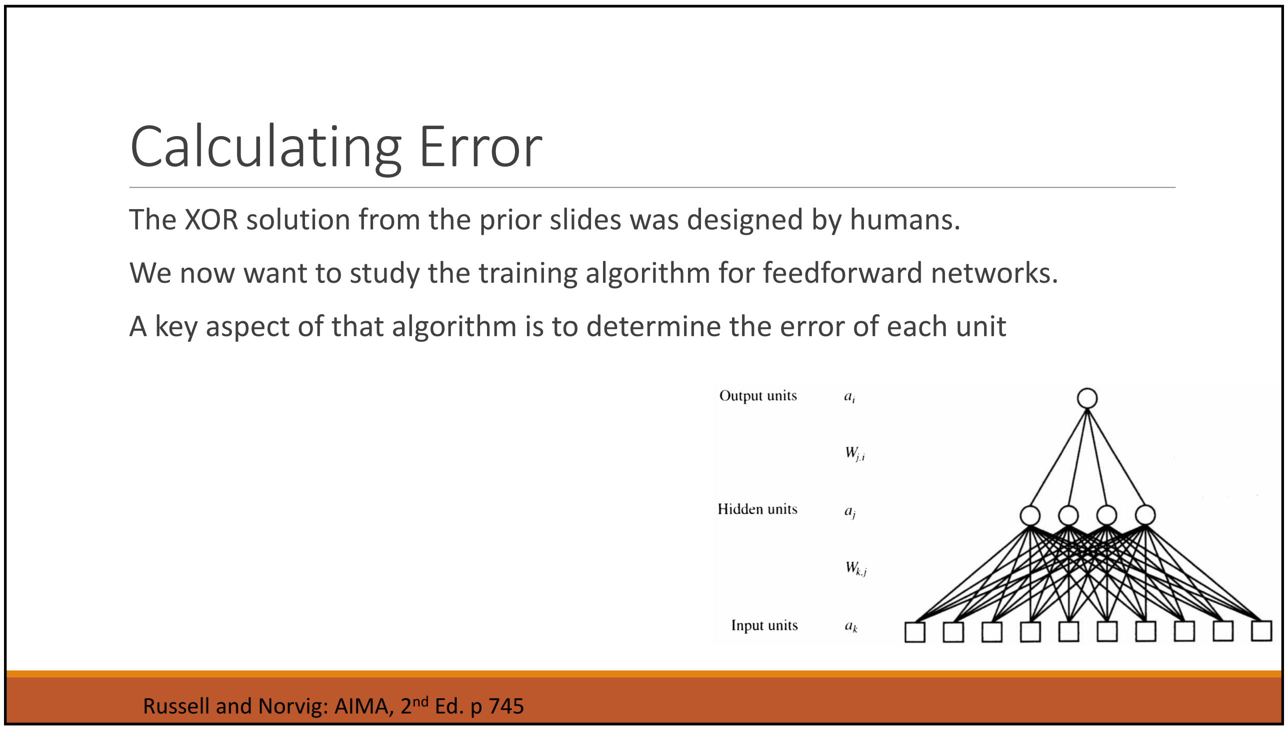


Figure 7.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

Figure source: Jurafsky and Martin, Speech and Language Processing, 3rd Ed.

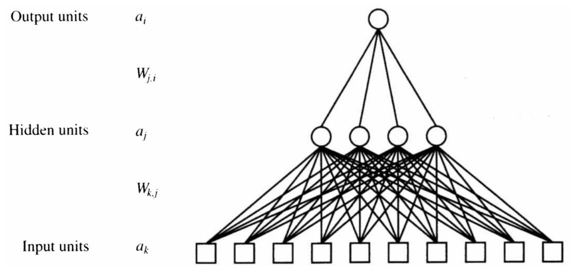


Calculating Error

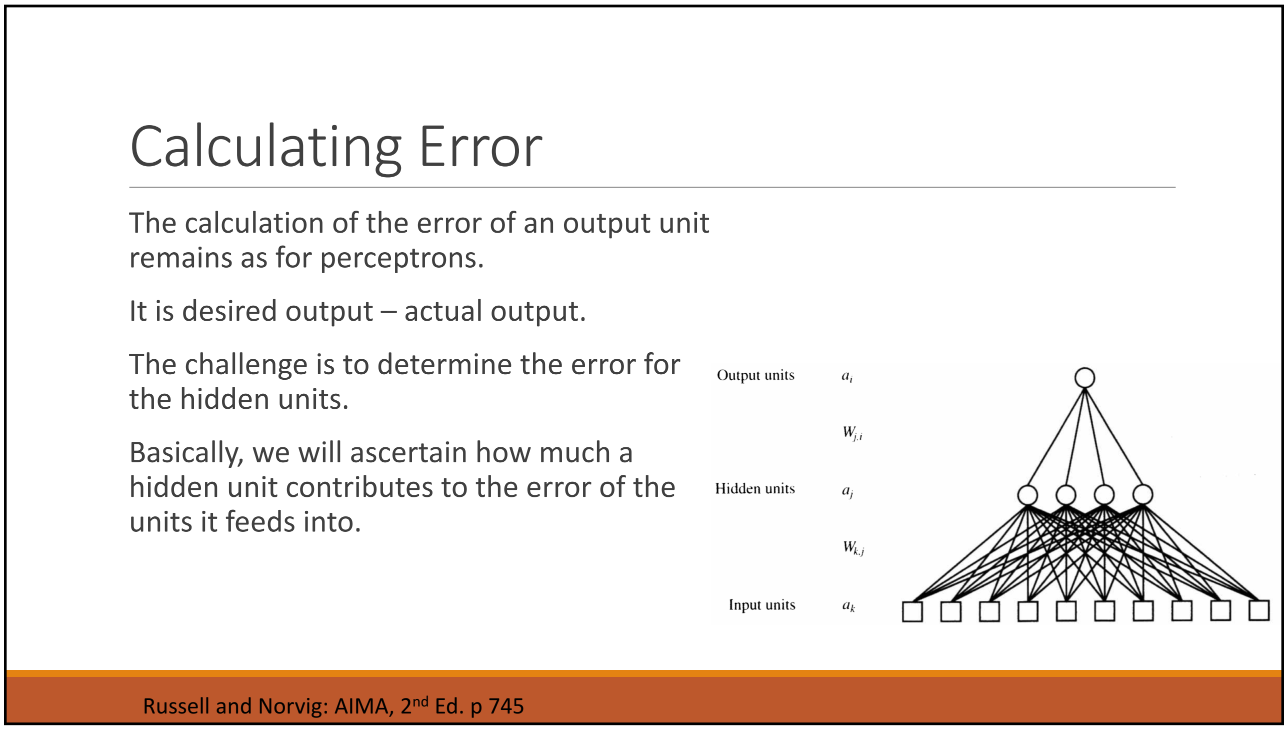
The XOR solution from the prior slides was designed by humans.

We now want to study the training algorithm for feedforward networks.

A key aspect of that algorithm is to determine the error of each unit



Russell and Norvig: AIMA, 2nd Ed. p 745



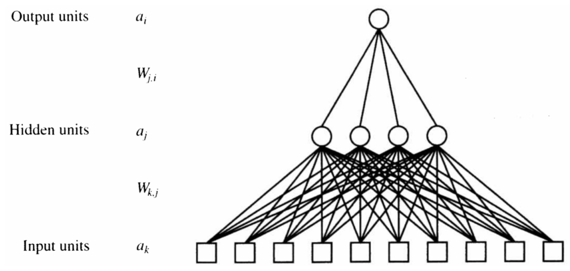
Calculating Error

The calculation of the error of an output unit remains as for perceptrons.

It is desired output – actual output.

The challenge is to determine the error for the hidden units.

Basically, we will ascertain how much a hidden unit contributes to the error of the units it feeds into.



Russell and Norvig: AIMA, 2nd Ed. p 745

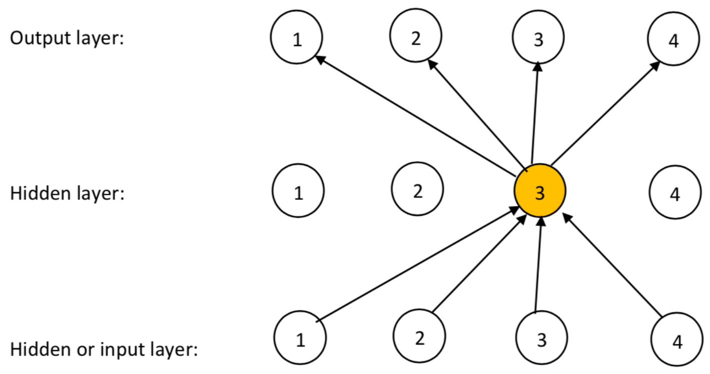
Calculating Error of Hidden Layer Nodes

Let’s focus on node 3 of the hidden layer of network on the right.

Suppose we already calculated the error of each of the four output units.

Node 3 sends its output to all four nodes of the output layer.

As such, we say that it contributes to any error of those four output units.

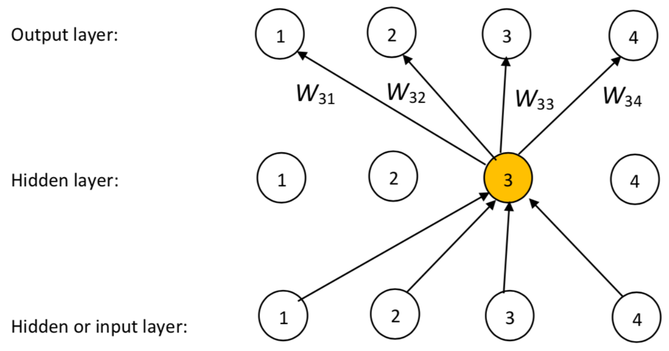


Calculating Error of Hidden Layer Nodes

In order to determine the magnitude of node 3’s contribution to the errors, we consider the weights.

The error of the shaded unit 3 is then the weighted sum of the errors of units 1-4 of the output layer.

$\sum_{i:1..n} W_{3i}\, \text{Error}_i$



Backpropagation Learning Algorithm

Just as with perceptrons, there are two stages:

1. We run the network on an input to produce an output.
2. We calculate errors of the units and adjust weights.

Notice that the error is calculated backwards, from the output layer through the hidden layer.

Hence "back"propagation.

Backpropagation Learning Algorithm

Let's have a look at the first portion, in which we run the network on a given input.

M is the number of layers.

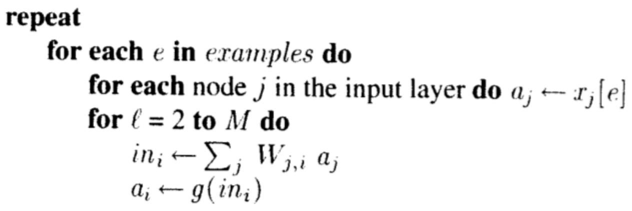
The input layer is layer 1.

As you can see, for each unit at a given layer, the algorithm

- calculates the input
- Using the activation function, it calculate the output.

It does this for all layers.

Typically, FFNs use sigmoid functions



```
repeat
    for each $e$ in $examples$ do
        for each node $j$ in the input layer do $a_j \leftarrow x_j[e]$
        for $\ell = 2$ to $M$ do
            $in_i \leftarrow \sum_j W_{j,i}\, a_j$
            $a_i \leftarrow g(in_i)$
```

Figure source: Russell and Norvig: AIMA, 2nd Ed. p 746

Backpropagation Learning Algorithm

Let’s have a look at the backpropagation portion, in which we adjust the weights.

In order to adjust the weights, we need to know the error.

The algorithm first calculates the errors of the output layer by subtracting the output from the desired output value.

Since the activation function is differentiable, the difference is multiplied with the derivative of the function value, i.e. the slope of the activation.

The algorithm then calculates the error of the hidden layers *j.*

Next the algorithm adjusts the weights for the prior layer *i.*

```
for each node $i$ in the output layer do
    $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
for $\ell = M - 1$ to 1 do
    for each node $j$ in layer $\ell$ do
        $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$
        for each node $i$ in layer $\ell + 1$ do
            $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
until some stopping criterion is satisfied
```

Figure source: Russell and Norvig: AIMA, 2nd Ed. p 746

Backpropagation Learning Algorithm

Here is the complete algorithm.

If you rather much see the algorithm in code, here is a link to an annotated Java implementation of it.

```
repeat
    for each $e$ in $examples$ do
        for each node $j$ in the input layer do $a_j \leftarrow x_j[e]$
        for $\ell = 2$ to $M$ do
            $in_i \leftarrow \sum_j W_{j,i} a_j$
            $a_i \leftarrow g(in_i)$
        for each node $i$ in the output layer do
            $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
        for $\ell = M - 1$ to 1 do
            for each node $j$ in layer $\ell$ do
                $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$
                for each node $i$ in layer $\ell + 1$ do
                    $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
until some stopping criterion is satisfied
```

Figure source: Russell and Norvig: AIMA, 2nd Ed. p 746

Loss Functions and Gradient Descent

When talking about NN, we need to understand two terms:

1. Loss functions and
2. Gradient descent

There are many loss functions, see the Wikipedia entry on Loss Functions.

Ours is very simple, just the sum of the differences between desired and actual output.

Here are some common loss functions used in Deep Learning.

We wish to reduce the error or minimize the loss function.

Finding the Min of a Loss Function

Gradient descent is an iterative optimization algorithm for finding a local minimum in a differentiable function.

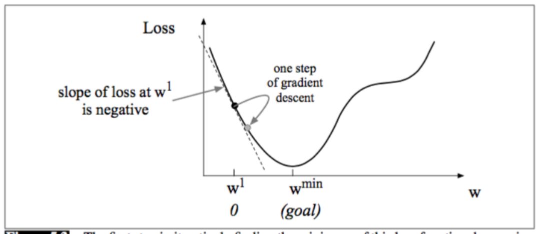


Figure 5.3 The first step in iteratively finding the minimum of this loss function, by moving *w* in the reverse direction from the slope of the function. Since the slope is negative, we need to move *w* in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of *w* (which is 0), w^2 at the second step, and so on.

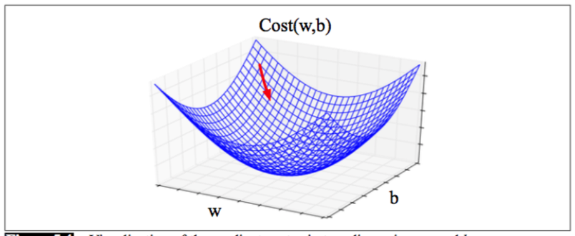


Figure 5.4 Visualization of the gradient vector in two dimensions *w* and *b*.

Figure source: Jurafsky and Martin, Speech and Language Processing, 3rd Ed.

Nettalk

Minksy and Papert's book *Perceptrons*, published in 1969 had the effect of bringing research into NN to a standstill.

In the 80s, some researchers showed renewed interest in NN.

Sejnowski and Rosenberg developed a system that could learn to read out aloud.

Their system was very successful, turned a lot of heads and jumpstarted research into NN.

Incidentally, Convolutional Neural Networks (CNN) were developed in the late 80s.

Nettalk

Nettalk was trained on English text at the input units and phonemes, i.e. ways of sounding characters of text at the output

The input text was presented on a sliding window of 7 characters

There were 7 groups of 29 units at the input layer.

The hidden layer consisted of 80 units

The output layer consisted of 26 units.

The network was fully connected.

They used bias inputs to each unit.

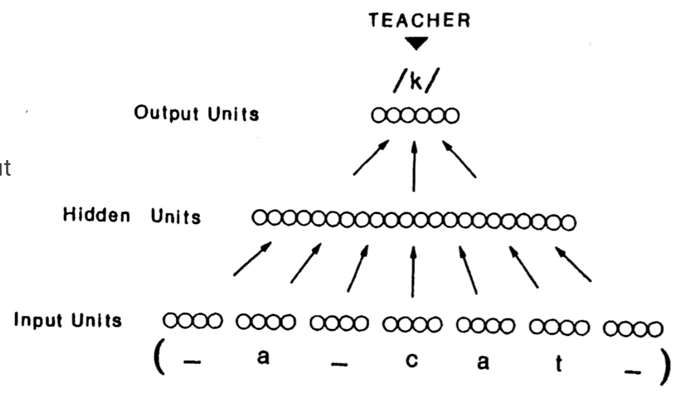
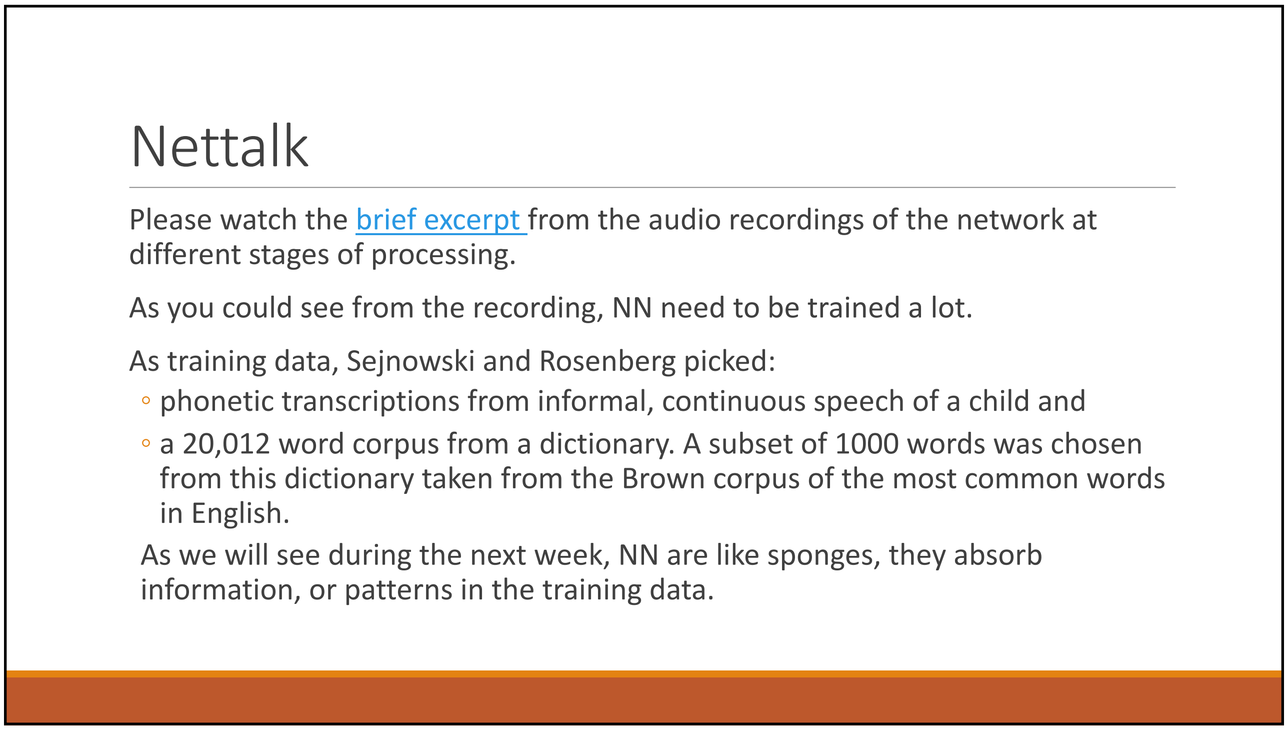


Figure source: Sejnowski and Rosenberg NETtalk: a parallel network that learns to read aloud , Figure 2



Nettalk

Please watch the brief excerpt from the audio recordings of the network at different stages of processing.

As you could see from the recording, NN need to be trained a lot.

As training data, Sejnowski and Rosenberg picked:

- phonetic transcriptions from informal, continuous speech of a child and
- a 20,012 word corpus from a dictionary. A subset of 1000 words was chosen from this dictionary taken from the Brown corpus of the most common words in English.

As we will see during the next week, NN are like sponges, they absorb information, or patterns in the training data.