Multilayer networks & backprop

CS530
Chapman

Spring 2021

1

MORE ABOUT FROM SINGLE LAYER TO MULTILAYER NETWORKS

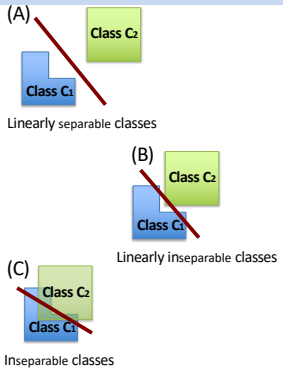
2

Who needs multi-layer networks?

So far, we focused more on 1-layer networks and saw that they can classify only linearly separable problems, either using the perceptron or delta rules.

What if our classes are not linearly separable, or not even perfectly separable using any hyper-curve?

The perceptron rule would never converge, forever striving to find a line that perfectly separates the classes. The delta rule would converge to some local minimum, resulting in a line that would separate the classes, to some extent and with errors.

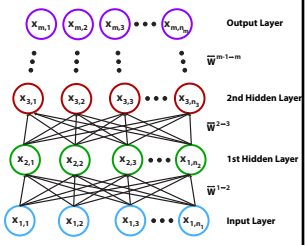


3

Who needs multi-layer networks?

More-general architectures can achieve non-linear separators and much more. Such a network has an input layer, whose neurons' states are the patterns input into the network. It has 1 or more hidden layers, called hidden because we do typically directly manipulate their inputs or outputs. Last, they possess an output layer, where the output classes are represented.

In the general sense, the network strives to transform the states of its input layer into states of its output layer that would match its classification goals.

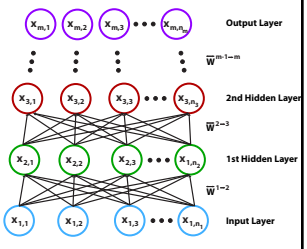


4

Who needs multi-layer networks?

If the network contains only linear neurons, it practically reduces to a single layer. The output would then just be a series of matrix multiplications of the inputs. And all those matrix multiplications would end up as a single resulting matrix, hence practically one layer.

To get more processing power than a single layer, non-linearities must be introduced into the network. This is done in the form of non-linear activation functions for at least some of the neurons.



5

How to represent multi-layer networks?

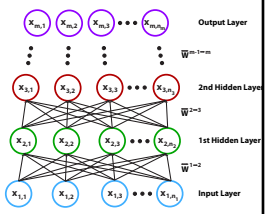
A general layer in a network does the following computation:

$\vec{x}^{k+1} = f\left(\vec{W}^{k\to k+1}\vec{x}^{k}\right)$.

So the weights an n-layer network are an n-1 ordered series:

$\left(\vec{W}^{1\to 2}, \vec{W}^{2\to 3}, \cdots, \vec{W}^{n-1\to n+1}\right)$

Such a series of matrices is termed a **tensor**.

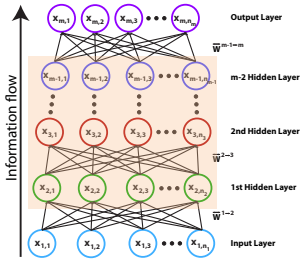


6

Function of hidden layers

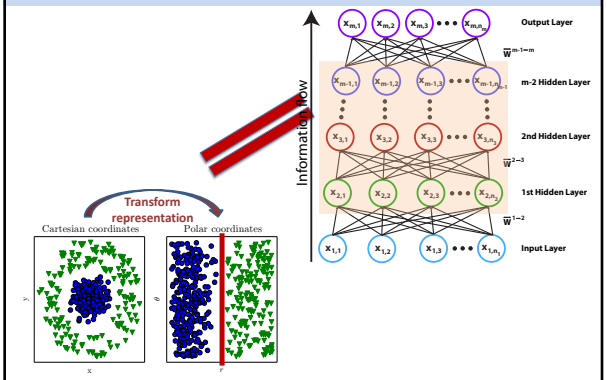
The hidden layers in a multilayer neural network serve as feature detectors. As the network weights are trained, the hidden-layer neurons take on salient features that characterize the training data.

They carry this out using nonlinear transformations on the input patterns. In the resulting non-linear feature space, the classification problem is typically more easily achieved than in the original space. In this respect, multilayer neural networks learn the nonlinear transformation kernels that they use on their own.



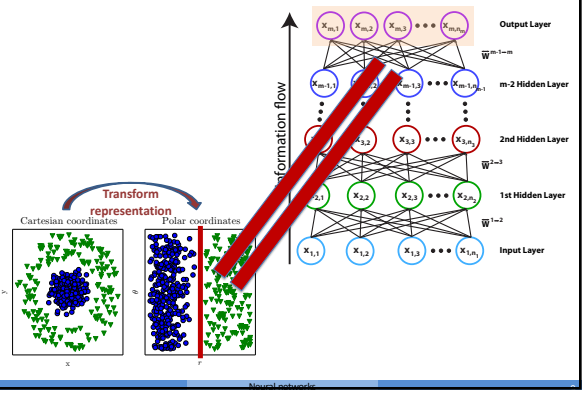
7

Function of hidden layers



8

Function of output layer



9

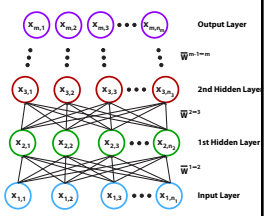
TRAINING MULTILAYER NETWORKS: THE BACKPROP

10

How to train multi-layer networks?

We can train multi-layer neural networks using gradient descent (or SGD), finding at least a local minimum. But how do we compute the gradient on such multi-layer networks?

Using back propagation (or the "backprop").

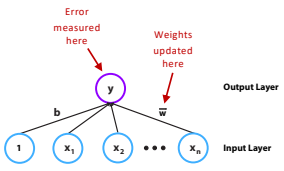


11

From single- to multi-layer networks

The perceptron and delta rules are effective error-correction methods using gradient descent for single-layer networks. But they both depend on:

1. Being able to measure the error between desired and actual output at the output neuron
2. The fact that only the weights into a single neuron contribute to the error for that neuron



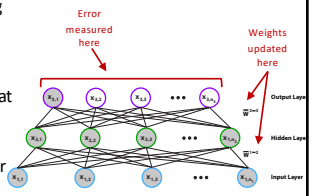
12

From single- to multi-layer networks

The perceptron and delta rules are effective error-correction methods using gradient descent for single-layer networks. But they both depend on:

1. Being able to measure the error between desired and actual output at the output neuron
2. The fact that only the weights into a single neuron contribute to the error for that neuron

In multilayer networks the error could be due to weights into that neuron from the previous layer. But it could also be due to all the weights from any downward layers. So which weights caused the error? This is a difficult credit-assignment problem.



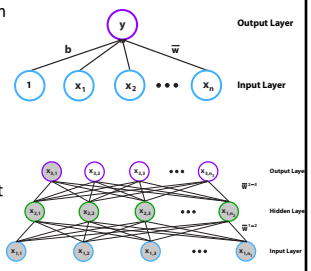
Any gray neuron could contribute to the error in $x_{3,1}$

13

From single- to multi-layer networks

In sum:

- For single-layer networks, the gradient of the error (or loss) function can easily and directly be computed
- For multilayer networks computing the gradient is not so trivial.
- The backpropagation (backprop) algorithm computers the gradient of the error function
- And we then use (stochastic) gradient descent to decrease the error



Any gray neuron could contribute to the error in $x_{3,1}$

14

BACK PROPAGATION ALGORITHM (OR "THE BACKPROP")

15

Reminders

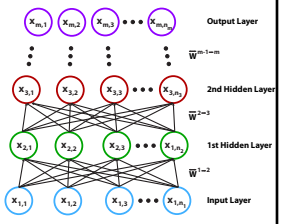
- In this network, each layer is a direct function of the layer below it

$$\vec{x}^{i+1} = f(\vec{W}^{i\to i+1} \cdot \vec{x}^{i})$$

- So,

$$\hat{\vec{y}} = \vec{x}^{m}$$

$$\vec{x}^{m} = f(\vec{W}^{m-1\to m} \cdot \vec{x}^{m-1})$$
$$= f(\vec{W}^{m-1\to m} \cdot f(\vec{W}^{m-2\to m-1} \cdot \vec{x}^{m-2}))$$
$$= f\Big(\vec{W}^{m-1\to m} \cdot f(\vec{W}^{m-2\to m-1} \cdot f(\cdots f(\vec{W}^{1\to 2} \cdot \vec{x}^{1}) \cdots))\Big)$$



16

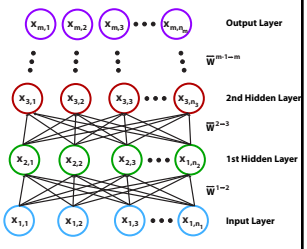
Reminders

- Collecting the weights (and biases) connecting all the layers into one 3D matrix, or tensor, we get:

$$\underset{\equiv}{W} = [\vec{W}^{1\to 2}, \vec{W}^{2\to 3}, \ldots, \vec{W}^{m-1\to m}]$$

- So, $\underset{\equiv}{W}$ holds all the weights in the neural network
- Now, we define

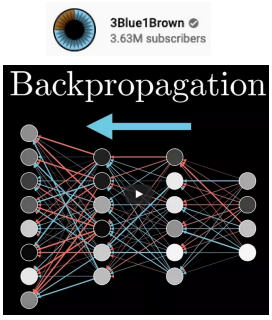
$$g\left(\underset{\equiv}{W}, \vec{x}^{1}\right)$$
$$= f\Big(\vec{W}^{m-1\to m}$$
$$\cdot f(\vec{W}^{m-2\to m-1}$$
$$\cdot f(\cdots f(\vec{W}^{1\to 2} \cdot \vec{x}^{1}))\cdots)\Big)$$



17

Animation intro to backprop

- Good (though not perfect) animation introduction to the backprop
- https://www.youtube.com/watch?v=Ilg3gGewQ5U



18

Intro to backprop

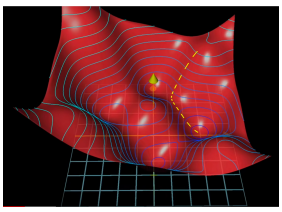
- Learning in a NN is minimizing a cost function (that is a function of all the weights, $\underline{\underline{w}}$) over the unseen test set.
- The cost function is

$$e\left(\underline{\underline{w}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\vec{\hat{y}}_i - \vec{y}_i\right)^2,$$

where $\vec{\hat{y}}_i = g\left(\underline{\underline{w}}, \vec{x}_i\right)$ and m is the size of the training set.

- So, for a given training set $\{\vec{x}_i\}_{i=1}^{m}$,

$$e\left(\underline{\underline{w}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left[g\left(\underline{\underline{w}}, \vec{x}_i\right) - \vec{y}_i\right]^2$$



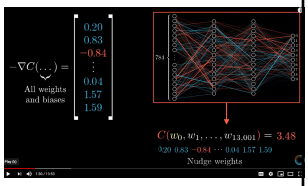
https://www.youtube.com/watch?v=Ilg3gGewQ5U

19

Intro to backprop

- Backprop computes the gradient of the cost function, $\nabla e\left(\underline{\underline{w}}\right)$
- We then (typically) use SGD to go down the gradient in steps of

$$\Delta\underline{\underline{w}} = -\eta\nabla e\left(\underline{\underline{w}}\right)$$

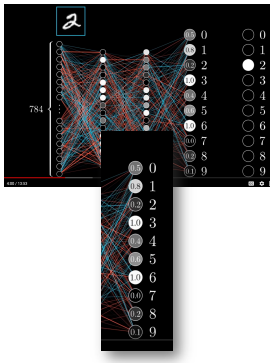


https://www.youtube.com/watch?v=Ilg3gGewQ5U

20

The output layer in the movie

- In the movie, are the network outputs probabilities or not?
  - Output layer not softmax. So, activations between 0 & 1, but do not add up to 1
  - With softmax: $[.5\ .8\ .2\ 1\ .4\ .6\ 1\ 0\ .2\ .1] \Rightarrow \begin{bmatrix}.1\ .13\ .07\ .16\ .09\ .11\ .16\\ .06\ .07\ .06\end{bmatrix}$



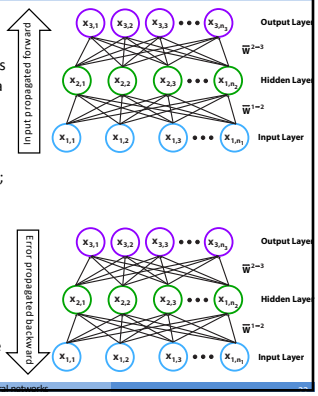
21

Backpropagation in multi-layer networks

The backpropagation algorithm devises a rule to solve the credit assignment problem for feed-forward neural networks with non-linear neurons. Using this rule, a multilayer neural network can modify its weights to reduce its error term.

Backpropagation works in two stages:

1. Forward phase: the weights are fixed; the input is propagated forward from the input layer, though the hidden layers, to the output layer.
2. Backward phase: An error term is computed at the output layer, by comparing it to the desired output. This error term is then propagated back though the hidden layers to the input layer.



22

Backpropagation in multi-layer networks

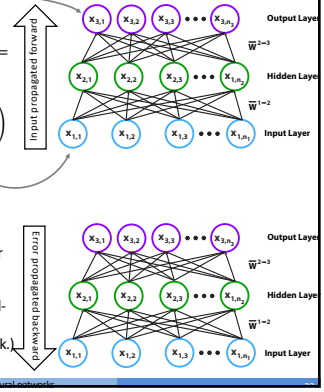
$$\hat{\vec{y}} = \vec{x}^m = f\left(\vec{\vec{w}}^{m-1\to m}\cdot\vec{x}^{m-1}\right)$$
$$= f\left(\vec{\vec{w}}^{m-1\to m}\cdot f\left(\vec{\vec{w}}^{m-2\to m-1}\cdot\vec{x}^{m-2}\right)\right) =$$
$$f\left(\vec{\vec{w}}^{m-1\to m}\cdot f\left(\vec{\vec{w}}^{m-2\to m-1}\cdot\right.\right.$$
$$\left.\left.f\left(\vec{\vec{w}}^{m-3\to m-2}\cdot f\left(\cdots f\left(\vec{\vec{w}}^{1\to 2}\cdot\vec{x}^{1}\right)\right)\right)\right)\right)$$

Written differently,

$$\vec{\hat{y}}_i = g\left(\underline{\underline{w}}, \vec{x}_i\right)$$

The index i is to stress that this is just for sample i of overall M samples.

(Do not confuse the m layers of the feed-forward neural network with the M samples over which we train the network.)



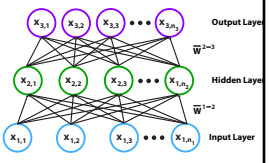
23

Backpropagation in multi-layer networks

Backpropagation, or "backprop" stands for backward propagation of errors. Backpropagation calculates the gradient of the error or loss function with respect to the weights

$$\nabla e\left(\underline{\underline{w}}\right)$$

Traversing down the gradient is then left to a (stochastic) gradient-descent algorithm.



24

Deriving the backpropagation: notation

Below, $x_{i,j}$ is the activity of the j'th neuron in layer i.

$$x_{i,j} = f\left(\sum_{k=1}^{n_{i-1}} x_{i-1,k}\left(\vec{W}^{i-1\to i}\right)_{j,k}\right) = f\left(\sum_{k=1}^{n_{i-1}} x_{i-1,k} w_{j,k}\right)$$

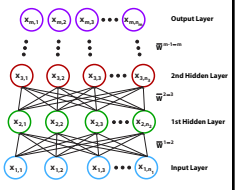
So we implicitly assume that $w_{j,k}$ leads from the k'th neuron in layer i-1 into the j'th neuron in layer i.

We define $\vec{W}^{i-1\to i} = \left[\vec{w}^{(1\downarrow)} \mid \vec{w}^{(2\downarrow)} \mid \cdots \mid \vec{w}^{(n_i\downarrow)}\right]$.

So column vector $\vec{w}^{(q\downarrow)} = \left(w_{1,q}, w_{2,q}, \cdots, w_{n_{i-1},q}\right)^T$ holds all outputs from q'th neuron in layer i-1 to layer i.

We similarly define $\vec{W}^{i-1\to i} = \begin{bmatrix} \vec{w}^{(\vec{1})} \\ \vec{w}^{(\vec{2})} \\ \vdots \\ \vec{w}^{(\vec{n}_{i-1})} \end{bmatrix}$.

Row vector $\vec{w}^{(\vec{r})} = \left(w_{r,1}, w_{r,2}, \cdots, w_{r,n_{i-1}}\right)$ holds all inputs into the r'th neuron in layer i from layer i-1.



25

Deriving the backpropagation: notation

Following the above,

$$x_{i,j} = f\left(\sum_{k=1}^{n_{i-1}} x_{i-1,k} w_{j,k}\right) = f\left(\vec{x}^{i-1} \cdot \vec{w}^{(j)}\right).$$

Let us focus on $x_{3,1}$ for example,

the activity of the first neuron in layer 3,

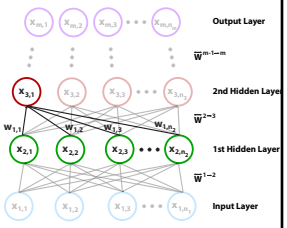
$$x_{3,1} = f\left(\sum_{k=1}^{n_2} x_{2,k} w_{1,k}\right) = f\left(\vec{x}^2 \cdot \vec{w}^{(\vec{1})}\right).$$

Here, again,

$$\vec{x}^2 = \left(x_{2,1}, x_{2,2}, \cdots, x_{2,n_2}\right),$$

$$\vec{w}^{(\vec{1})} = \left(w_{1,1}, w_{1,2}, \cdots, w_{1,n_1}\right) = \left(\left(\vec{W}^{2\to 3}\right)_{1,1}, \left(\vec{W}^{2\to 3}\right)_{1,2}, \cdots, \left(\vec{W}^{2\to 3}\right)_{1,n_1}\right).$$

So, again, we take $\vec{w}$ to refer to the correct $\vec{W}^{i-1\to i}$ that connects $\vec{x}^{i-1}$'s layer to the one above it.



26

Deriving the backpropagation

For some neuron $x_{i,j}$, we denote its desired output by $y_{i,j}$.

So the squared error for that neuron is

$$e_{i,j}^2 = \left(y_{i,j} - x_{i,j}\right)^2.$$

We impelement gradient descent to reduce

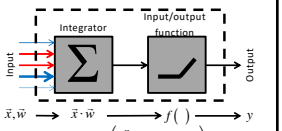
$$e^2 = \sum_{i=1}^{M}\sum_{j=1}^{n_i} e_{i,j}^2.$$ (M is number of samples.)

So we need to compute

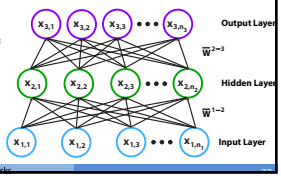
$$\Delta w_{j,k} = -\eta \frac{\partial\left(e^2\right)}{\partial\left(w_{j,k}\right)}.$$

We can break this up into 2, using the chain rule

$$\frac{\partial\left(e^2\right)}{\partial\left(w_{j,k}\right)} = \underbrace{\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}}_{\text{Input into } \vec{x}^i} \frac{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}{\partial\left(w_{j,k}\right)}.$$



$$x_{i,j} = f\left(\sum_{k=1}^{n} x_{i-1,k} w_{j,k}\right)$$



27

Deriving the backpropagation

$$\frac{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}{\partial\left(w_{j,k}\right)} = \frac{\partial\left(\sum_{c=1}^{n_{i-1}} w_{j,c} \cdot x_{i-1,c}\right)}{\partial\left(w_{j,k}\right)} = x_{i-1,k}.$$

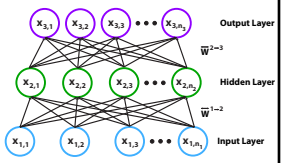
Also, wanting to go downhill, we define

$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)},$$

a generalized error term for $x_{i,j}$. So now

$$\Delta w_{j,k} = -\eta \frac{\partial\left(e^2\right)}{\partial\left(w_{j,k}\right)} = -\eta \frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)} \frac{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}{\partial\left(w_{j,k}\right)}$$

$$= \eta \delta_j x_{i-1,k}.$$



28

Deriving the backpropagation

Scrutinizing δ_j, by applying the chain rule again,

$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)} = -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} \frac{\partial\left(x_{i,j}\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}.$$

For a linear neuron,

$$x_{i,j} = \vec{w}^{(j)} \cdot \vec{x}^{i-1} = \sum_{k=1}^{n_{i-1}} w_{i,k} \cdot x_{i-1,k} \Rightarrow \frac{\partial\left(x_{i,j}\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)} = 1$$

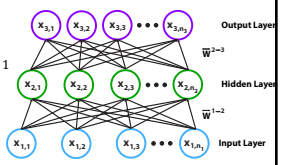
Accomplishing nothing (see below).

But, for a non-linear neuron,

$$x_{i,j} = f\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)$$

$$\Rightarrow \frac{\partial\left(x_{i,k}\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)} = \frac{\partial\left(f\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)} = f'\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right).$$

Backprop requires differentiable activation function



29

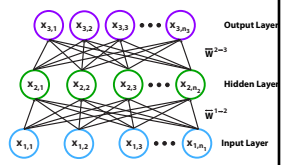
Backpropagation – deriving δ

We therefore derived

$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}$$

$$= -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} \frac{\partial\left(x_{i,j}\right)}{\partial\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right)}$$

$$= -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} f'\left(\vec{w}^{(j)} \cdot \vec{x}^{i-1}\right).$$



30

Backpropagation – output layer

In the output layer,

$$\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} = 2e\frac{\partial e}{\partial\left(x_{i,j}\right)} = 2e\frac{\partial\left(y_{i,j}-x_{i,j}\right)}{\partial\left(x_{i,j}\right)} = -2e$$

$$\Rightarrow \delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,k}\right)} f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right) = 2e\cdot f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)$$

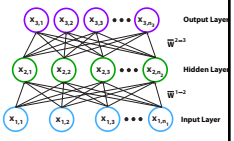
$$= 2\left(y_{i,j}-x_{i,j}\right) f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)$$

And

$$\Delta w_{j,k} = \eta\delta_j x_{i-1,k} = 2\eta\left(y_{i,j}-x_{i,j}\right) f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right) x_{i-1,k}.$$

As in the delta rule, the update, Δw, is a product of the input activity, $x_{i-1,k}$, the error, e, and f'.

Though for the linear neuron in the delta rule $f' = 1$.



31

Backpropagation – earlier layer

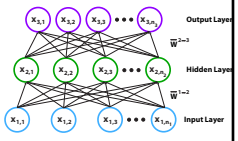
We now propogate the error from layer $(i+1)$ to layer i.

Assuming we know the error in layer $(i+1)$,

we will compute δ_j in layer i. Using the chain rule,

$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)} = -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)}\frac{\partial\left(x_{i,j}\right)}{\partial\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)}$$

$$= -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right).$$



We know the connection strengths between layers $i+1$ and i, because we assume that these are the same as the connections between layers i and $i+1$ (a very non-biological assumption, by the way).

32

Backpropagation – earlier layer

$x_{i,j}$ affects the activity of $x_{i+1,k}$ $\forall k$, or $\vec{x}^{i+1}$.

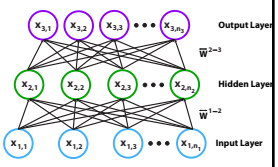
In other words, a neuron's activity affects all neurons in the layer above it. Therefore, to find the error due to $x_{i,j}$, we must sum up all the errors from layer $i+1$, because $x_{i,j}$ can affect the errors in all these units. Hence

$$e^2 = \sum_{c=1}^{n_{i+1}}\left(y_{i+1,c}-x_{i+1,c}\right)^2 = \sum_{c=1}^{n_{i+1}}\left[y_{i+1,c}-f\left(\vec{w}^{(c)}\cdot\vec{x}^{i}\right)\right]^2.$$

Now, differentiating this expression w.r.t. $x_{i,j}$,

$$\frac{\partial\left(e^2\right)}{\partial x_{i,j}} = \sum_{c=1}^{n_{i+1}}\left[\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(c)}\cdot\vec{x}^{i}\right)}\frac{\partial\left(\vec{w}^{(c)}\cdot\vec{x}^{i}\right)}{\partial x_{i,j}}\right]$$

$$= \sum_{c=1}^{n_{i+1}}\left[\frac{\partial\left(e^2\right)}{\partial\left(\vec{w}^{(c)}\cdot\vec{x}^{i}\right)} w_{c,j}\right] = -\sum_{c=1}^{n_{i+1}}\left(\delta_c\cdot w_{c,j}\right).$$



33

Backpropagation – earlier layer

Putting it together,

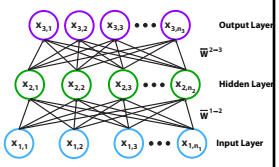
$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)$$

$$= f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)\sum_{c=1}^{n_{i+1}}\left(\delta_c\cdot w_{c,j}\right).$$

So, for an earlier layer,

$$\Delta w_{j,k} = \eta\delta_j x_{i-1,k}.$$

Therefore, following all the math, we know how to compute the weight changes in the output layer. And we know how to go from the output layer back, one layer at a time.



34

Backpropagation – earlier layer

Putting it together,

$$\delta_j = -\frac{\partial\left(e^2\right)}{\partial\left(x_{i,j}\right)} f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)$$

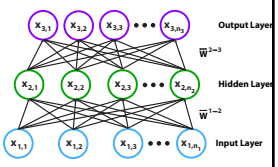
$$= f'\left(\vec{w}^{(j)}\cdot\vec{x}^{i-1}\right)\sum_{c=1}^{n_{i+1}}\left(\delta_c\cdot w_{c,j}\right).$$

So, for an earlier layer,

$$\Delta w_{j,k} = \eta\delta_j x_{i-1,k}.$$

Therefore, following all the math, we know how to compute the weight changes in the output layer. And we know how to go from the output layer back, one layer at a time.

Above is the formula to compute weight changes for a single sample. How do we compute the weight changes over the entire training set?



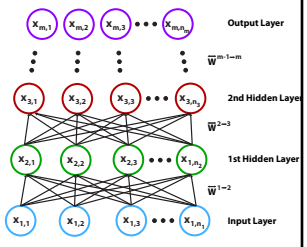
35

Modes of learning in multilayer NN

Two modes of learning are possible:

1. Batch learning: All weight updates over all samples are computed and then GD is run once, at the end of the epoch. Computation of weights can be carried out in parallel, saving computation time.
2. Stochastic (online) learning: GD computed immediately, after each weight update for each sample. This induces noise due to local gradient computation, so reduces chances of getting stuck in local minima.

Modern applications often use the "mini-batches compromise": batch learning with small batch sizes and stochastically selected samples in each batch.



36

Backpropagation – earlier layer

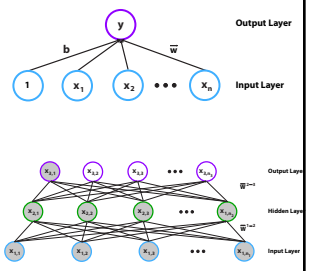
Note that the update rule for weights in a multilayer neural network as derived from the backprop,

$$\Delta w_{j,k} = \eta \delta_j x_{i-1,k},$$

is not that different in form from the Widrow-Hoff (delta) rule in the case of the (gradient-descent) perceptron,

$$\Delta \vec{w} = \eta (y - \vec{w} \cdot \vec{x}) \vec{x}.$$

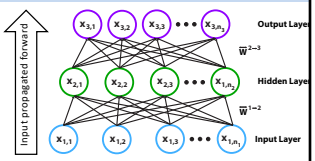
Though for the backprop in multilayer networks, $\delta = y - \vec{w} \cdot \vec{x}$.

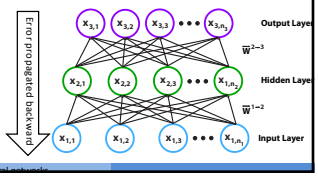


37

Backpropagation

Variants of this back-propagation algorithm, or "backprop" are at the heart of training virtually all multilayer networks and most other architectures. So, it is at the heart of the deep-learning field.

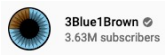


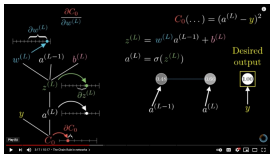


38

Animation explanation of math

- If you want more information, you could consider watching an animation explaining the math behind backprop
- Note: some of the notation used is different from ours
- https://www.youtube.com/watch?v=tIeHLnjs5U8





39