CSCI-567: Machine Learning

Prof. Victor Adamchik

U of Southern California

July 9, 2020

Your model is only as good as your data.

Outline

Outline

Problem 3

Show that one-versus-all can be seen as a special case of error-correcting-output-code (ECOC). Specifically, write down the code matrix M for ECOC for a problem with C labels so that executing ECOC is the same as doing one-versus-all. (Note: the entry of M should be either −1 or +1.)

Problem 4

Assume we have a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, the probability of seeing out come y is given by

$$P(y|x_n) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - \mathbf{w}^T \mathbf{x}_n)^2}{2\,\sigma^2})\right)$$

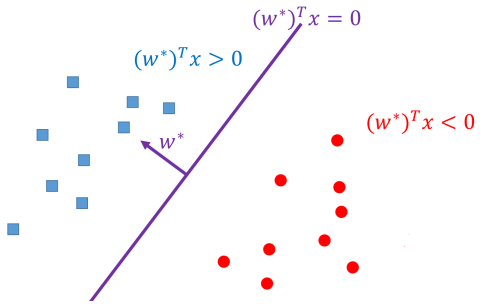
Find the maximum likelihood estimations for $\mathbf{w}$ and σ

Solution

Outline

The Perceptron

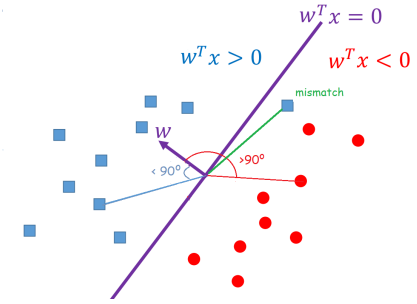
The Perceptron (introduced by Rosenblatt in 1957) is a linear model for classification. Its model is a hyperplane that partitions space into two regions.



Intuition

The product of two vectors is defined by

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} = \|\boldsymbol{w}\|\|\boldsymbol{x}\|\cos(\alpha)$$



For a point to be classified as a blue square we want $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0$, and the angle $\alpha < 90^\circ$ between $\boldsymbol{x}$ and $\boldsymbol{w}$. In order to decrease the alpha value, we update weights by $\boldsymbol{w}' = \boldsymbol{w} + \boldsymbol{x}$

Why does it make sense?

If the current weight $\boldsymbol{w}$ makes a mistake

$$y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n < 0$$

then after the update $\boldsymbol{w}' = \boldsymbol{w} + y_n\boldsymbol{x}_n$ we have

$$y_n\boldsymbol{w}'^{\mathrm{T}}\boldsymbol{x}_n = y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n + y_n^2\boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n = y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n + \|\boldsymbol{x}_n\|^2 \geq y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n$$

Thus it is more likely to get it right after the update.

The Perceptron Algorithm

Repeat:

- Pick a data point $\boldsymbol{x}_n$ uniformly at random
- If $\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$$

Note:

- The algorithm is online and error driven.
- If the prediction is correct, it does nothing.
- $\boldsymbol{w}$ is always a *linear combination* of the training examples.

Applying GD to perceptron loss

Mathematically, find the minimizer of

$$F(\boldsymbol{w}) = \sum_{n=1}^{N} \ell_{\text{perceptron}}(y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) = \sum_{n=1}^{N} \max\{0, -y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n\}$$

Gradient is

$$\nabla F(\boldsymbol{w}) = \sum_{n=1}^{N} -\mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \leq 0]y_n\boldsymbol{x}_n$$

(only misclassified examples contribute to the gradient)

GD update

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \lambda\sum_{n=1}^{N} \mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \leq 0]y_n\boldsymbol{x}_n$$

Any theory?

- If training set is linearly separable, Perceptron *converges in a finite number of steps*
- How long does it take to converge?
- By "how long", what we really mean is "how many updates".
- One way to make this definition is through the notion of margin.
- The margin is the distance between the hyperplane and the nearest point.

Perceptron Convergence Theorem

Suppose the perceptron algorithm is run on a linearly separable data set $\mathcal{D}$ with margin $\gamma \geq 0$. Assume that $\|x\| = 1$. Then the algorithm will converge after at most $\frac{1}{\gamma^2}$ updates.

Exercise 1

The following table shows a binary classification training set and the number of times each point is misclassified during a run of the perceptron algorithm. Which of the following is the final output of the algorithm? Assume $\boldsymbol{w}$ initialized with zeros.

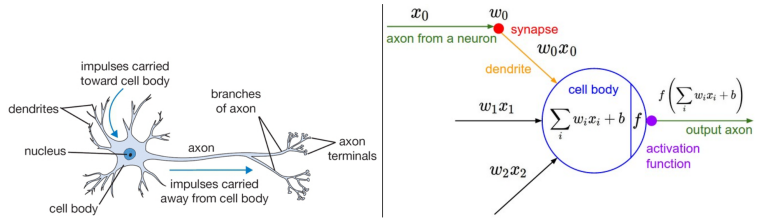
$\boldsymbol{x}$	y	Times misclassified
(-3, 2)	+1	5
(-1, 1)	-1	5
(5, 2)	+1	3
(2, 2)	-1	4
(1, -2)	+1	3

Outline

Biological motivation

Perceptron is a rough model for how individual neurons in the brain work. The basic computational unit of the brain is a **neuron**. Approximately 86 billion neurons can be found in the human brain. Neurons are connected with each other via **synapses**. Each neuron receives input signals from its **dendrites** and produces output signals along its **axon**.

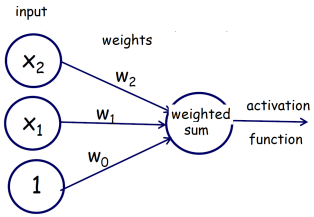
Biological motivation



a biological neuron (left) and its mathematical model (right).

In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact with the dendrites of the other neuron (e.g. $w_0 x_0$). In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can *fire*. We model the firing rate of the neuron by an activation function.

Perceptron as a linear classifier

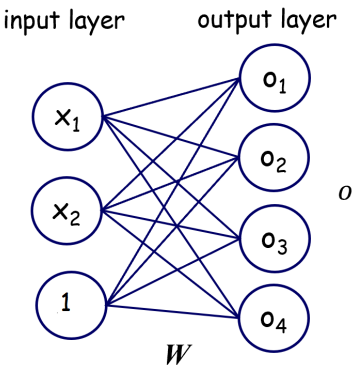


The perceptron works in three simple steps:

- All the inputs x are multiplied with their weights w.
- All the multiplied values are summed to weighted sum.
- An activation function is applied to that weighted sum.

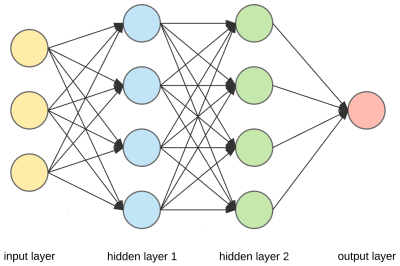
An activation function must be a nonlinear function!

More output nodes



$o = h(\boldsymbol{W}\,\boldsymbol{x})$

More layers



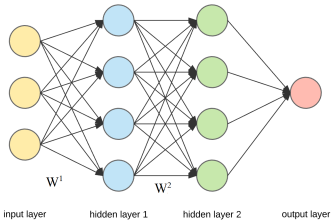
Neural Network:

- each node is called a **neuron**
- each hidden layer has an **activation function**
- #layers refers to #hidden_layers (plus 1 or 2 for input/output layers)
- **deep** neural nets can have many layers and *millions* of parameters
- this is a **feedforward, fully connected** neural net, called multilayer perceptron (MLP).

Math formulation

An L-layer neural net can be written as

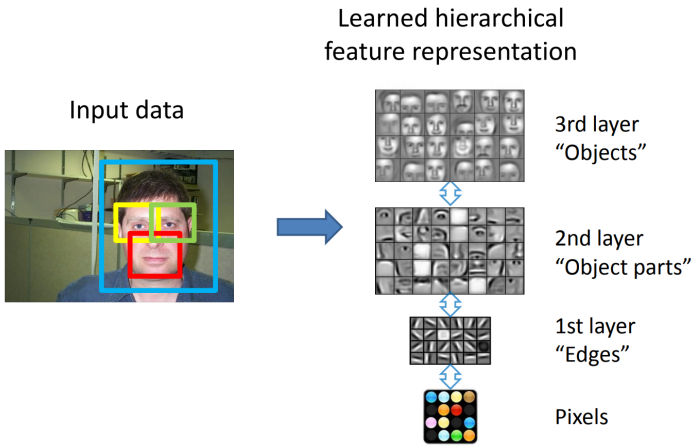
$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathrm{L}}\left(\boldsymbol{W}^{L}\boldsymbol{h}_{\mathrm{L}-1}\left(\boldsymbol{W}^{L-1}\cdots\boldsymbol{h}_{1}\left(\boldsymbol{W}^{1}\boldsymbol{x}\right)\right)\right)$$



To learn this model we need to learn weights $W^1, W^2, \ldots, W^L$.
They are learnt by Gradient Descent using *the backpropagation algorithm*.

Example from Honglak Lee (NIPS 2010)

Face recognition:



How powerful are neural nets?

Universal approximation theorem (Cybenko, 89; Hornik, 91):

A feedforward neural net with a single hidden layer and finite number of neurons can approximate any continuous functions under mild assumptions on the activation (sigmoidal) function.

It might need a huge number of neurons though, and *depth helps!* With more layers, you may need less neurons.

Designing network architecture is important and very complicated

- for feedforward network, need to decide number of hidden layers, number of neurons at each layer, activation functions, etc.

There is no theory for that, but some rules of thumb in practice.

Deep Feedforward Networks: Historical Notes

In the 1940s, approximation techniques were used to motivate machine learning models such as the perceptron.

Inability to learn the XOR function, led to a backlash against the neural network.

Following the success of backpropagation (1986), NN gained popularity.

The core ideas behind modern feedforward networks have not changed substantially since the 1980s.

The same backpropagation algorithm and the same approaches to gradient descent are still in use.

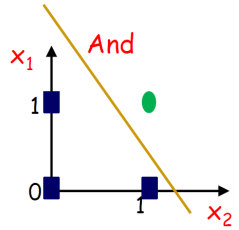
The modern revival of deep learning started in 2006.

Until 2012, it was widely believed that feedforward networks would not perform well.

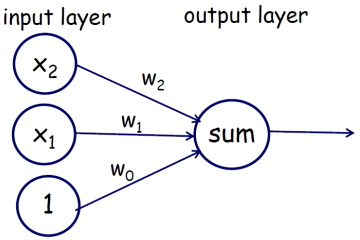
Today, it is now known that with the right resources and engineering practices, feedforward networks perform very well.

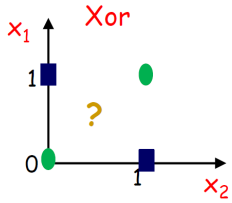
Exercise 2

Perceptron can represent *some* Boolean functions.



Prove why perceptron cannot represent XOR.





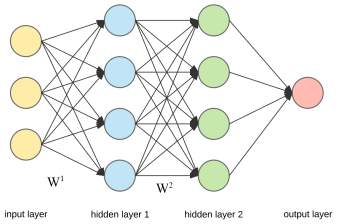
Solution

Outline

Math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathsf{L}}\left(\boldsymbol{W}^{L}\boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}^{L-1}\cdots \boldsymbol{h}_1\left(\boldsymbol{W}^{1}\boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, (one training point) define recursively

$$\boldsymbol{o}^0 = \boldsymbol{x}, \qquad \boldsymbol{a}^\ell = \boldsymbol{W}^\ell \boldsymbol{o}^{\ell-1}, \qquad \boldsymbol{o}^\ell = \boldsymbol{h}_\ell(\boldsymbol{a}^\ell) \qquad (\ell = 1, \ldots, \mathsf{L})$$

where

- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_{\mathsf{L}}$ are numbers of neurons at each layer
- $\boldsymbol{W}^\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights for layer ℓ
- $\boldsymbol{a}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output to layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

Learning the model

No matter how complicated the model is, our goal is the same: minimize

$$\mathcal{E}(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \sum_{n=1}^{\mathsf{N}} \mathcal{E}_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}})$$

where

$$\mathcal{E}_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \begin{cases} \|\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n\|_2^2 & \text{for regression} \\ \ln\left(1 + \sum_{k \neq y_n} e^{f(\boldsymbol{x}_n)_k - f(\boldsymbol{x}_n)_{y_n}}\right) & \text{for classification} \end{cases}$$

How to optimize such a complicated function?

Same thing: apply **SGD**! even if the model is *nonconvex*.

What is the gradient of this complicated function?

Chain rule:

- for a composite function $f(g(w))$

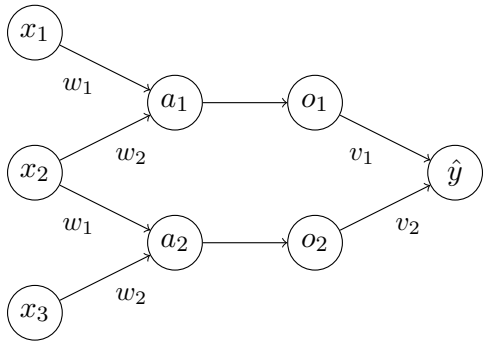
$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w}$$

- for a composite function $f(g_1(w), \ldots, g_d(w))$

$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial w}$$

Simple example

Consider the following mini neural net, where (x_1, x_2, x_3) is the input, followed by a layer with a filter (w_1, w_2), a ReLU layer, and another layer with weight (v_1, v_2).



where $(\boldsymbol{x}, y) \in \mathbb{R}^3 \times \{-1, +1\}$, and $\hat{y}$ is prediction, y is ground truth.

Simple example

More concretely, the computation is specified by

$$\begin{aligned} a_1 &= x_1 w_1 + x_2 w_2, \quad & a_2 &= x_2 w_1 + x_3 w_2 \\ o_1 &= \max\{0, a_1\}, & o_2 &= \max\{0, a_2\} \\ \hat{y} &= o_1 v_1 + o_2 v_2 \end{aligned}$$

We define a loss function by

$$\ell = \ln(1 + \exp(-y\hat{y}))$$

Simple example

We start with the last layer and compute $\frac{\partial \ell}{\partial v_1}$ and $\frac{\partial \ell}{\partial v_2}$.

$$\begin{aligned} \frac{\partial \ell}{\partial v_1} &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_1} \\ &= \frac{-ye^{-y\hat{y}}}{1 + e^{-y\hat{y}}} o_1 = -\sigma(-y\hat{y}) y o_1 = (\sigma(y\hat{y}) - 1) y o_1 \end{aligned}$$

$$\begin{aligned} \frac{\partial \ell}{\partial v_2} &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_2} \\ &= \frac{-ye^{-y\hat{y}}}{1 + e^{-y\hat{y}}} o_2 = -\sigma(-y\hat{y}) y o_2 = (\sigma(y\hat{y}) - 1) y o_2 \end{aligned}$$

Simple example

Next we compute $\frac{\partial \ell}{\partial w_1}$ and $\frac{\partial \ell}{\partial w_2}$. We will be using $H(a) = \mathbb{I}[a > 0]$.

$$\begin{aligned} \frac{\partial \ell}{\partial w_1} &= \frac{\partial \ell}{\partial a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial \ell}{\partial a_2} \frac{\partial a_2}{\partial w_1} \\ &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_1} \frac{\partial o_1}{\partial a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_2} \frac{\partial o_2}{\partial a_2} \frac{\partial a_2}{\partial w_1} \\ &= (\sigma(y\hat{y}) - 1) y (v_1 H(a_1) x_1 + v_2 H(a_2) x_2) \end{aligned}$$

$$\frac{\partial \ell}{\partial w_2} = (\sigma(y\hat{y}) - 1) y (v_1 H(a_1) x_2 + v_2 H(a_2) x_3).$$

Backpropagation algorithm.

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)$, learning rate λ
Initialize: w_1, w_2, v_1, v_2 randomly
While not converged:

- randomly pick an example $(\boldsymbol{x}_n, y_n)$
- Forward propagation: compute

$$\begin{aligned} &a_1 = x_1 w_1 + x_2 w_2, a_2 = x_2 w_1 + x_3 w_2 \\ &o_1 = \max\{0, a_1\}, o_2 = \max\{0, a_2\}, \hat{y} = o_1 v_1 + o_2 v_2 \end{aligned}$$

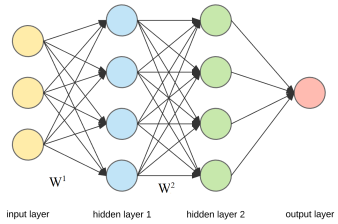
- Backward propagation: update

$$\begin{aligned} w_1 &\leftarrow w_1 - \lambda(\sigma(y\hat{y}) - 1) y (v_1 H(a_1) x_1 + v_2 H(a_2) x_2) \\ w_2 &\leftarrow w_2 - \lambda(\sigma(y\hat{y}) - 1) y (v_1 H(a_1) x_2 + v_2 H(a_2) x_3) \\ v_1 &\leftarrow v_1 - \lambda(\sigma(y\hat{y}) - 1) y o_1 \\ v_2 &\leftarrow v_2 - \lambda(\sigma(y\hat{y}) - 1) y o_2 \end{aligned}$$

General Case: math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathsf{L}}\left(\boldsymbol{W}^{L}\boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}^{L-1}\cdots\boldsymbol{h}_1\left(\boldsymbol{W}^{1}\boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, define recursively

$$\boldsymbol{o}^0 = \boldsymbol{x}, \qquad \boldsymbol{a}^\ell = \boldsymbol{W}^\ell \boldsymbol{o}^{\ell-1}, \qquad \boldsymbol{o}^\ell = \boldsymbol{h}_\ell(\boldsymbol{a}^\ell) \qquad (\ell = 1, \ldots, \mathsf{L})$$

where

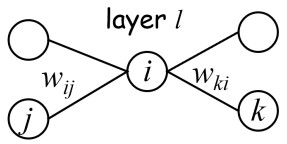
- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_{\mathsf{L}}$ are numbers of neurons at each layer
- $\boldsymbol{W}^\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights for layer ℓ
- $\boldsymbol{a}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output to layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

General Case: computing derivatives

An equation for the rate of change of the cost with respect to any weight in the network.

Find the **derivative of $\mathcal{E}_n$ w.r.t. to $\boldsymbol{W}$**

$$a^l = W^l\, o^{l-1}$$
$$o^l = h_l(a^l)$$



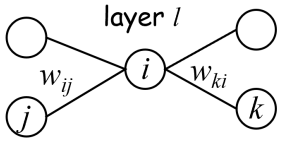
$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}^\ell} = \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}\frac{\partial \boldsymbol{a}^\ell}{\partial \boldsymbol{W}^\ell} = \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}(\boldsymbol{o}^{\ell-1})^{\mathrm{T}}$$

General Case: computing derivatives

Computing the error in terms of the next layer errors.

Find the **derivative of $\mathcal{E}_n$ w.r.t. to $\boldsymbol{a}^\ell$**

$$a^l = W^l\, o^{l-1}$$
$$o^l = h_l(a^l)$$



$$\frac{\partial \mathcal{E}_n}{\partial a_i^\ell} = \frac{\partial \mathcal{E}_n}{\partial o_i^\ell}\frac{\partial o_i^\ell}{\partial a_i^\ell} = \left(\sum_k \frac{\partial \mathcal{E}_n}{\partial a_k^{\ell+1}}\frac{\partial a_k^{\ell+1}}{\partial o_i^\ell}\right) h'_\ell(a_i^\ell) = \left(\sum_k \frac{\partial \mathcal{E}_n}{\partial a_k^{\ell+1}} w_{ki}^{\ell+1}\right) h'_\ell(a_i^\ell)$$

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell} = \left((\boldsymbol{W}^{\ell+1})^{\mathrm{T}}\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^{\ell+1}}\right) \circ h'_\ell(\boldsymbol{a}^\ell)$$

where $\boldsymbol{x} \circ \boldsymbol{y} = (x_1 y_1, \cdots, x_n y_n)$ is the Hadamard product.

General Case: computing derivatives

Computing the error for the last layer. We know that

$$\mathcal{E}_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \begin{cases} \|\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n\|_2^2 & \text{for regression} \\ \ln\left(1 + \sum_{k \neq y_n} e^{f(\boldsymbol{x}_n)_k - f(\boldsymbol{x}_n)_{y_n}}\right) & \text{for classification} \end{cases}$$

where $\boldsymbol{f}(\boldsymbol{x}) = h_L(\boldsymbol{a}^L)$.

The derivative is :

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^{\mathsf{L}}} = \nabla \mathcal{E}_n \circ h'_{\mathsf{L}}(\boldsymbol{a}^{\mathsf{L}})$$

General Case: backpropagation

Backpropagation is based on the following three equations:

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}^\ell} = \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}(\boldsymbol{o}^{\ell-1})^{\mathrm{T}}$$

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell} = \begin{cases} ((\boldsymbol{W}^{\ell+1})^{\mathrm{T}} \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^{\ell+1}}) \circ \boldsymbol{h}'_\ell(\boldsymbol{a}^\ell) & \text{if } \ell < \mathsf{L} \\ \nabla \mathcal{E}_n \circ \boldsymbol{h}'_{\mathsf{L}}(\boldsymbol{a}^{\mathsf{L}}) & \text{else} \end{cases}$$

where $\boldsymbol{x} \circ \boldsymbol{y} = (x_1 y_1, \cdots, x_n y_n)$ is the Hadamard product.

Verify dimensions for the above equations.

- $\boldsymbol{W}^\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights for layer ℓ
- $\boldsymbol{a}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output to layer ℓ

Putting everything into SGD

The backpropagation algorithm (1986) computes the gradient of the cost function for a single training example.

Initialize $\boldsymbol{W}^1, \ldots, \boldsymbol{W}^{\mathsf{L}}$ (all $\mathbf{0}$ or randomly). Repeat:

1. randomly pick one data point $n \in [\mathsf{N}]$
2. **forward propagation**: for each layer $\ell = 1, \ldots, \mathsf{L}$
   - compute $\boldsymbol{a}^\ell = \boldsymbol{W}^\ell \boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^\ell = \boldsymbol{h}_\ell(\boldsymbol{a}^\ell)$ $\quad (\boldsymbol{o}^0 = \boldsymbol{x}_n)$
3. **backward propagation**: for each $\ell = L, \ldots, 1$
   - compute
   $$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}$$
   - update weights
   $$\boldsymbol{W}^\ell \leftarrow \boldsymbol{W}^\ell - \lambda \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}^\ell} = \boldsymbol{W}^\ell - \lambda \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}(\boldsymbol{o}^{\ell-1})^{\mathrm{T}}$$

More tricks to optimize neural nets

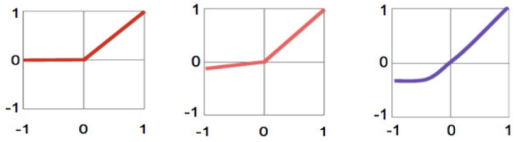
Using ReLU (instead of sigmoid) as an activation function. It solves the vanishing gradient problem.

There are several improved versions of ReLU that have been proposed which provide even better accuracy

One of them is Leaky ReLU: $y = \max(ax, x)$, here a is a small constant, $a = 0.1$.

Another popular activation function is Exponential Linear Unit (ELU) defined by

$$y = \begin{cases} x, & x \geq 0 \\ a(e^x - 1), & x < 0 \end{cases}$$



Overfitting

Overfitting is very likely since the models are too powerful.

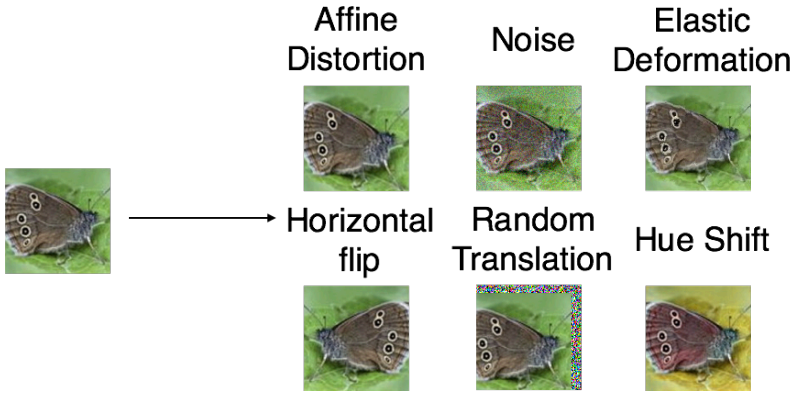
Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- ...

Data augmentation

Data: the more the better. How do we get more data?

Exploit prior knowledge to add more training data



Regularization

L2 regularization: minimize

$$\mathcal{E}'(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \mathcal{E}(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) + \eta \sum_{\ell=1}^{\mathsf{L}} \|\boldsymbol{W}_\ell\|_2^2$$

Simple change to the gradient:

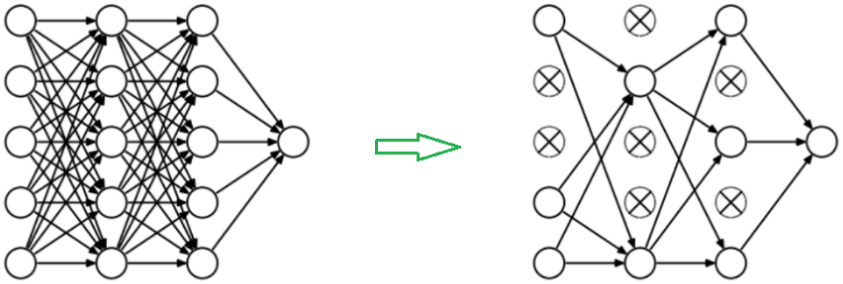
$$\frac{\partial \mathcal{E}'}{\partial w_{ij}} = \frac{\partial \mathcal{E}}{\partial w_{ij}} + 2\eta w_{ij}$$

Introduce *weight decaying effect*:

$$\boldsymbol{W}^\ell \leftarrow \boldsymbol{W}^\ell - \lambda \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}^\ell} - 2\eta\lambda \boldsymbol{W}^\ell$$

Dropout

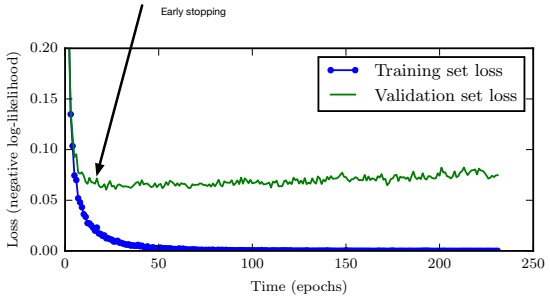
Randomly delete neurons during training



Very effective, makes training faster as well

Early stopping

Stop training when the performance on validation set stops improving



Conclusions for neural nets

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- take *a lot of time* to train (need GPUs for massive parallel computing)
- take some work to select architecture and hyperparameters
- are still not well understood in theory

Conclusions for backpropagation learning

(due to Yann LeCun)

- gradient descent is unreliable when the network is small
- the solution is to make the network much larger than necessary and regularize it
- Although there are lots of local minima, many of them are equivalent
- It doesn't matter which one you fall into with large nets, backpropagation yields very consistent results
- Many local minima are due to symmetries in the system
- Breaking the symmetry in the architecture solves many problems. This may be why convolutional nets work so well