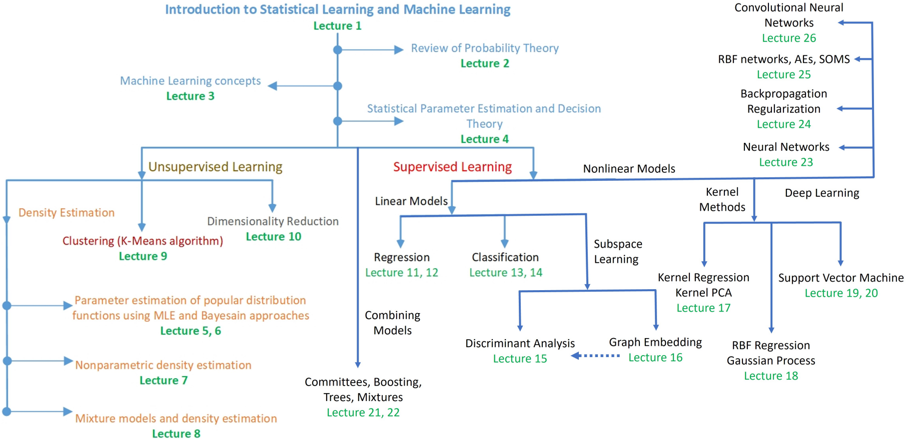
Statistical Learning and Machine Learning

Lecture 24 - Neural Networks 2

October 13, 2021

Course overview and where do we stand



Why Neural Networks?

Neural networks:

- Adaptive (parametric) basis functions
- Hierarchical data transformations
- Usually more compact (and efficient) than kernel methods of similar performance
- Non-convex optimization
- Neural networks comprising of many hidden layers (*Deep Learning*) have proven to be effective in many machine learning problems

The artificial neuron

The basic building block of a neural network is called *neuron*:

- It receives as input a vector, e.g. $\mathrm{x} \in \mathbb{R}^D$ and applies the following transformation:

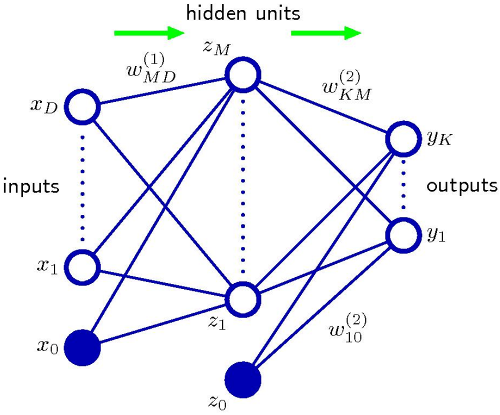
$$\alpha = \sum_{i=1}^{D} w_i x_i + w_0 = \mathrm{w}^T \mathrm{x} + w_0 \tag{1}$$

$$z = h(\alpha) \tag{2}$$

where:

- $\{\mathrm{w}, w_0\}$ are the parameters of the neuron
- w is called *weight* and w_0 is called *bias*
- α is known as the *activation*
- $h(\cdot)$ is a nonlinear *activation function*
- $h(\cdot)$ can take many forms, depending on the position of the neuron in the neural network and the problem at hand

A two-layer feed-forward neural network

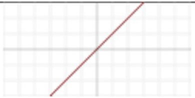
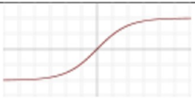
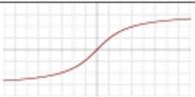
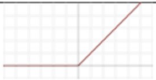


$$y_k(\mathrm{x}, \mathrm{w}) = \sigma\left(\sum_{j=1}^{M} W^{(2)}_{kj}\, h\left(\sum_{i=1}^{D} W^{(1)}_{ji} x_i + w^{(1)}_{j0}\right) + w^{(2)}_{k0}\right) \qquad (3)$$

Activation functions

Name	Equation	Derivative
Identity	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
HyperbTan	$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan	$f(x) = \tanh^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
ReLU	$f(x) = \begin{cases} 0, \text{if } x \leq 0 \\ x, \text{if } x > 0 \end{cases}$	$f(x) = \begin{cases} 0, \text{if } x \leq 0 \\ 1, \text{if } x > 0 \end{cases}$
Softmax	$f_i(\mathbf{x}) = \frac{e^{x_k}}{\sum_{l=1}^{K} e^{x_l}}, \quad k = 1, \ldots, K$	$\frac{\vartheta f_i(\mathbf{x})}{\vartheta x_j} = f_i(\mathbf{x})(\delta_{ij} - f_j(\mathbf{x}))$

Activation functions

Name	Plot
Identity	
Logistic	
HypTan	
ArcTan	
ReLU	

Multilayer Perceptron

The above neural network is called *Multilayer Perceptron* (or MLP):

- an important property is that the activation functions of all neurons are differentiable w.r.t. their parameters

The use of nonlinear activation functions is crucial:

- If we use linear activation functions in the above two-layer neural network:

$$y_k(\mathrm{x}, \mathrm{w}) = \mathrm{W}^{(2)T}\mathrm{W}^{(1)T}\tilde{\mathrm{x}} = \mathcal{W}^T\tilde{\mathrm{x}} \tag{4}$$

  where $\mathcal{W} = \mathrm{W}^{(2)T}\mathrm{W}^{(1)T} \in \mathbb{R}^{(D+1)\times K}$.

The error function is the negative log-likelihood (discarding the terms not depending on w and scaling factors):

$$E(\mathrm{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(\mathrm{x}_n, \mathrm{w}) - t_n\Big)^2 \tag{5}$$

Parameter optimization: Gradient descent

Gradient descent updates w using:

$$\mathrm{w}^{(\tau+1)} = \mathrm{w}^{\tau} - \eta \nabla E(\mathrm{w}^{(\tau)}) \tag{6}$$

To find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point.

When

$$E(\mathrm{w}) = \sum_{n=1}^{N} E_n(\mathrm{w}) \tag{7}$$

stochastic gradient descent (SGD) can be used:

$$\mathrm{w}^{(\tau+1)} = \mathrm{w}^{\tau} - \eta \nabla E_n(\mathrm{w}^{(\tau)}) \tag{8}$$

Mini-batch gradient descent uses small chunks (e.g. 64) data points for each update.

Error Backpropagation

Iterative process for updating the weights of a neural network formed by two steps:

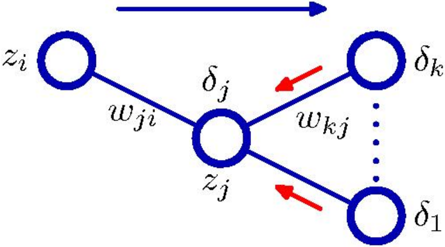
- Step 1: Evaluate the derivatives of the error function with respect to the weights
- Step 2: Adjust the weights using the derivatives

The contribution of the Backpropagation algorithm is that it provides a computationally efficient method for evaluating the derivatives.

Backpropagation algorithm can be used for:

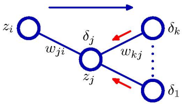
- a general network having arbitrary feed-forward topology
- arbitrary *differentiable nonlinear activation functions*,
- a broad class of error functions

Error Backpropagation



Forward and backward pass at a neuron

Error Backpropagation



At neuron j the forward pass is:

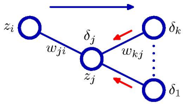
$$\alpha_j = \sum_i w_{ij} z_i \qquad z_j = h(\alpha_j). \tag{9}$$

The gradient of E_n w.r.t. w_{ji} can be expressed as (*chain rule*):

$$\frac{\theta E_n}{\theta w_{ji}} = \underbrace{\frac{E_n}{\theta \alpha_j}}_{\delta_j} \underbrace{\frac{\theta \alpha_j}{\theta w_{ji}}}_{z_i} = \delta_j z_i \tag{10}$$

Calculating the derivative of the error at neuron j is equal to mutliplying the value of the *error signal* at neuron j (δ_j) with the value of the output of neuron j (z_j, for 'dummy' neurons interacting with the bias $z = 1$).

Error Backpropagation



The error signal at neuron j is given by:

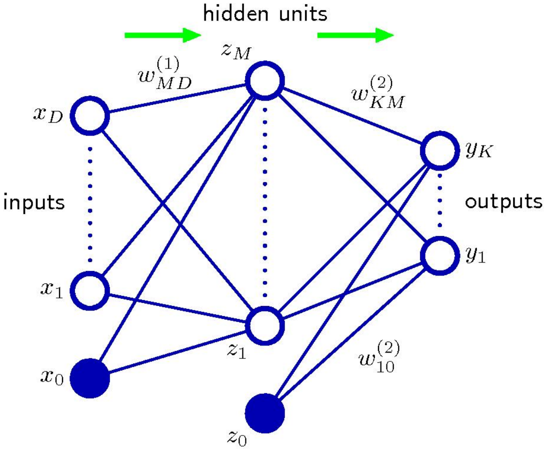
$$\delta_j = \frac{\theta E_n}{\theta \alpha_j} = \sum_k \frac{\theta E_n}{\theta \alpha_k} \frac{\theta \alpha_k}{\theta \alpha_j} = h'(\alpha_j) \sum_k w_{kj} \delta_k \quad (11)$$

Thus, the error signals at a neuron j are obtained by *bakcpropagating* the error signals from units higher up in the network.

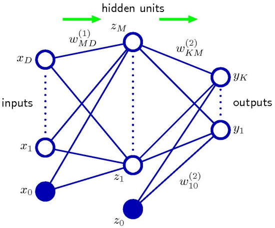
For (mini-)batch methods, when $E(\mathrm{W}) = \sum_n E_n(\mathrm{w})$ we have:

$$\frac{\theta E}{\theta w_{ji}} = \sum_n \frac{\theta E_n}{\theta w_{ji}} \quad (12)$$

Error Backpropagation: Example



Error Backpropagation: Example



We use:

- Sum-of-squares error:

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2 \tag{13}$$

- linear output neurons
- hidden neurons with activation function:

$$h(\alpha) = tanh(\alpha) = \frac{e^{\alpha} - e^{-\alpha}}{e^{\alpha} + e^{-\alpha}} \tag{14}$$

Error Backpropagation: Example

Forward pass:

- For each input data point x_n calculate:

$$\alpha_j = \sum_{i=0}^{D} W_{ji}^{(1)} x_i, \qquad z_j = tanh(\alpha_j), \qquad y_k = \sum_{j=1}^{M} W_{kj}^{(2)} z_j \tag{15}$$

Calculate the error signals for each output neurons: $\delta_k = y_k - t_k$.

Backward pass:

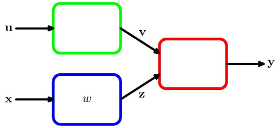
- Backpropagate the error signals:

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k \tag{16}$$

- Calculate the gradients w.r.t. the weights $\mathrm{W}^{(1)}$ and $\mathrm{W}^{(2)}$:

$$\frac{\theta E_n}{\theta W_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\theta E_n}{\theta W_{ji}^{(2)}} = \delta_k z_j \tag{17}$$

Error Backpropagation: The Jakobian matrix



The error w.r.t. w:

$$\frac{\theta E}{\theta w} = \sum_{k,j} \frac{\theta E}{\theta y_k} \frac{\theta y_k}{\theta z_j} \frac{\theta z_j}{\theta w} \tag{18}$$

where $\frac{\theta E_n}{\theta z_j}$ is the Jakobian of the red module.

For small errors: $\Delta_{y_k} \simeq \sum_i \frac{\theta y_k}{\theta x_i} \Delta x_i$.

In general:

$$J_{ki} = \frac{\theta y_k}{\theta x_i} = \sum_j \frac{\theta y_k}{\theta \alpha_j} \frac{\theta \alpha_j}{\theta x_i} = \sum_j w_{ji} \frac{\theta y_k}{\theta \alpha_j} = \left(\sum_j w_{ji} h'(\alpha_j) \left(\sum_l w_{lk} \frac{\theta y_k}{\theta \alpha_l} \right) \right) \tag{19}$$

Error Backpropagation: The Hessian matrix

The Hessian matrix has elements $\frac{\theta^2 E}{\theta w_{ji} \theta w_{lk}}$.

Usually H^{-1} is of interest:

- Diagonal approximation:

$$\frac{\theta^2 E_n}{\theta \alpha_j^2} = h'(\alpha_j)^2 \sum_k \sum_{k'} w_{kj} w_{k'j} \frac{\theta^2 E_n}{\theta \alpha_k \theta \alpha_{k'}} + h''(\alpha_j) \sum_k w_{kj} \frac{\theta E_n}{\theta \alpha_k} \quad (20)$$

- Outer product approximation:
  - For the sum-of-squares error function: $\mathrm{H} \simeq \sum_{n=1}^{N} \mathrm{b}_n \mathrm{b}_n^T$
  - For the cross-entropy error function with logistic sigmoid output neurons, and $\mathrm{b}_n = \nabla y_n = \nabla \alpha_n$:

$$\mathrm{H} \simeq \sum_{n=1}^{N} y_n(1 - y_n) \mathrm{b}_n \mathrm{b}_n^T \quad \text{and} \quad \mathrm{H}_{L+1}^{-1} = \mathrm{H}_L^{-1} - \frac{\mathrm{H}_L^{-1} \mathrm{b}_{L+1} \mathrm{b}_{L+1}^T \mathrm{H}_L^{-1}}{1 + \mathrm{b}_{L+1}^T \mathrm{H}_L^{-1} \mathrm{b}_{L+1}} \quad (21)$$

Regularization in Neural Networks

Weight decay:

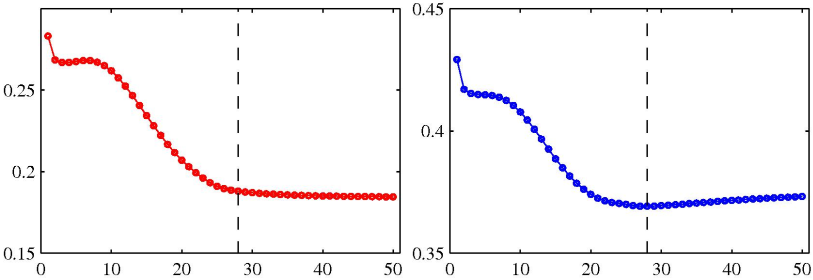
$$\tilde{E}(\mathrm{w}) = E(\mathrm{w}) + \frac{\lambda}{2}\mathrm{w}^T\mathrm{w} \tag{22}$$

To allow the regularizer to be invariant under linear transformations of the input data (for a network with L layers):

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2 + \cdots + \frac{\lambda_L}{2}\sum_{w\in\mathcal{W}_L} w^2 \tag{23}$$

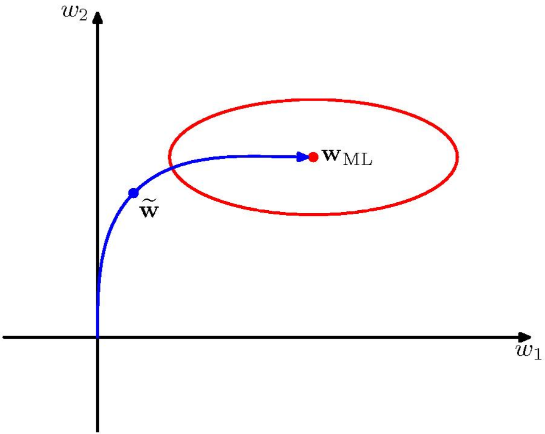
where $\mathcal{W}_l$ denotes the set of weights in layer l.

Early stopping:



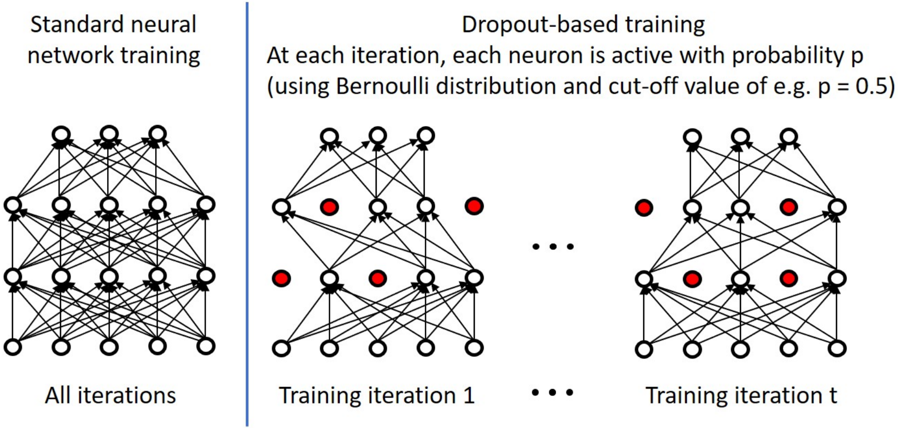
Left: Error function on the training data
Right: Error function on the validation data

Early stopping vs. weight decay regularization



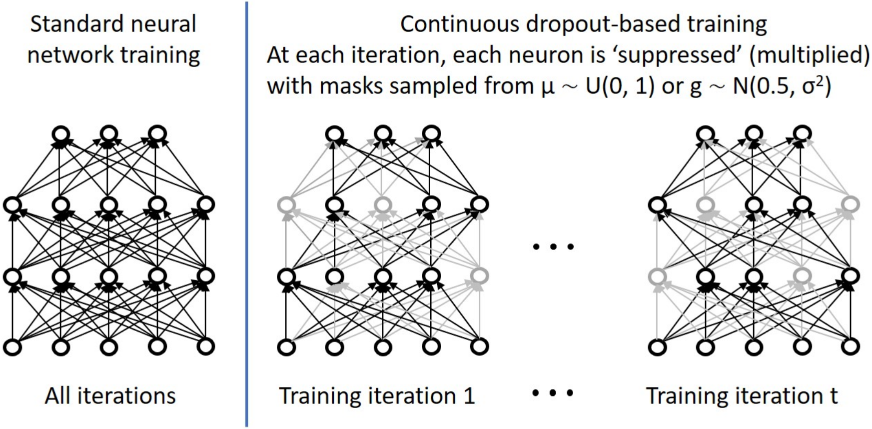
Regularization in Neural Networks: Dropout

Dropout in iterative optimization: A probabilistic process to 'augment' the training set during iterative training and increase invariance:



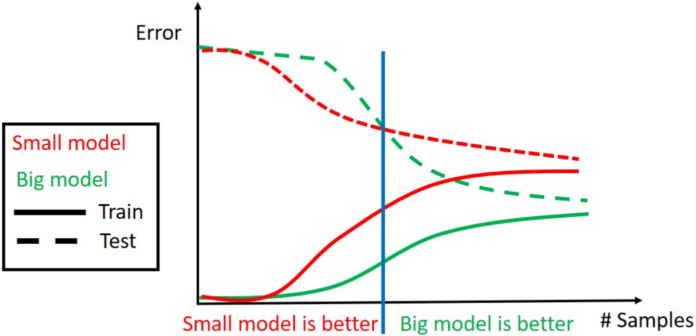
Regularization in Neural Networks: Dropout

Continuous Dropout-based iterative optimization:



Regularization in Neural Networks: Training set size

When the number of training samples is small (smaller than the number of the model's parameters) the model tends to overfit (under-determined problem).



Regularization in Neural Networks: Training set size

In neural networks, this problem is usually addressed by using:

- Data augmentation: create new samples by applying small variations on the training data. For example, for images: geometric variations (shift, rotations, scaling), crops, noise
- Transfer learning:
    1. Initialize the model's parameters with those of a pre-trained model on a big data set (having similar properties to the problem we want to solve)
    2. fine-tune the model using the small data set