Statistical Machine Learning

Christian Walder

Machine Learning Research Group
CSIRO Data61

and

College of Engineering and Computer Science
The Australian National University

Canberra
Semester One, 2020.

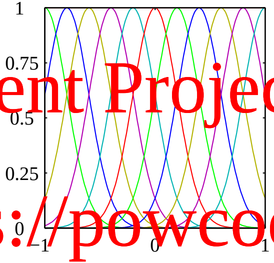
(Many figures from C. M. Bishop, "Pattern Recognition and Machine Learning")

Outlines

Part VII

Neural Networks 1

Basis Functions



- Play a crucial role in the algorithms explored so far.
- Previously (*e.g.* Linear Regression and Linear Classification): were fixed before learning starts.
- Now for Neural Networks: number of basis functions fixed, parameters of the basis functions are adaptive
- Later in kernel methods: center basis functions on the data / have an infinite number of effective basis functions (*e.g.* Support Vector Machines).

Outline

©2020
Ong & Walder & Webers
Data61 | CSIRO
The Australian National University

- The functional form of the network model (including special parametrisation of the basis functions).
- How to determine the network parameters within the maximum likelihood framework?
(Solution of a nonlinear optimisation problem.)

- Error backpropagation : efficiently evaluate the derivatives of the log likelihood function with respect to the network parameters.

- Various approaches to regularise neural networks.

Feed-forward Network Functions

- Same goal as before: *e.g.* for regression, decompose

$$t(\mathbf{x}) = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

  where ϵ is the noise.

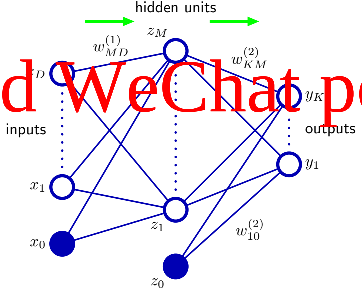
- (Generalised) Linear Model

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_j \phi_j(\mathbf{x})\right)$$

  where $\boldsymbol{\phi} = (\phi_0, \dots, \phi_M)^T$ is the fixed model basis and $\mathbf{w} = (w_0, \dots, w_M)^T$ are the model parameter.
- For regression: $f(\cdot)$ is the identity function.
- For classification: $f(\cdot)$ is a nonlinear activation function.
- Goal : Let $\phi_j(\mathbf{x})$ depend on parameters, and then adjust these parameters together with $\mathbf{w}$.

Feed-forward Network Functions

- Goal : Let $\phi_j(\mathbf{x})$ depend on parameters, and then adjust these parameters together with $\mathbf{w}$.
- Many ways to do this.
- Neural networks use basis functions which follow the same form as the (generalised) linear model.
- EACH basis function is itself a nonlinear function of an adaptive linear combination of the inputs.



Functional Transformations

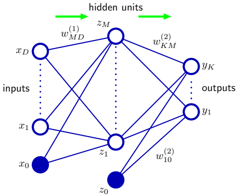
- Construct M linear combinations of the input variables $x_1, \dots, x_D$ in the form

$$\underbrace{a_j}_{\text{activations}} = \sum_{i=1}^{D} \underbrace{w_{ji}^{(1)}}_{\text{weights}} x_i + \underbrace{w_{j0}^{(1)}}_{\text{bias}} \qquad j = 1, \dots, M$$

- Apply a differentiable, nonlinear activation function $h(\cdot)$ to get the output of the hidden units

$$z_j = h(a_j)$$

- $h(\cdot)$ is typically sigmoid, $\tanh$, or more recently $\mathrm{ReLU}(x) = \max(x, 0)$



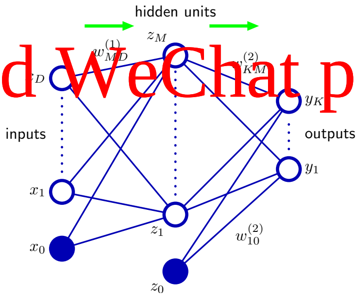
Functional Transformations

- Outputs of the hidden units are again linearly combined

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \qquad k = 1, \dots, K$$

- Apply again a differentiable, nonlinear activation function $g(\cdot)$ to get the network outputs y_k

$$y_k = g(a_k)$$



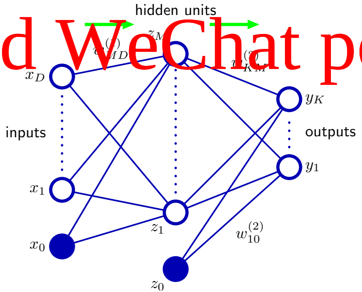
Functional Transformations

- The activation function $g(\cdot)$ is determined by the nature of the data and the distribution of the target variables.
- For standard regression: $g(\cdot)$ is the identity so $y_k = a_k$.
- For multiple binary classification, $g(\cdot)$ is a logistic sigmoid:

$$y_k = \sigma(a_k) = \frac{1}{1 + \exp(-a_k)}$$

- Recall from generative classification model perspective:

$$a_k(\mathbf{x}) = \ln \frac{p(\mathbf{x}, \mathcal{C}_{k_1})}{p(\mathbf{x}, \mathcal{C}_{k_2})}$$

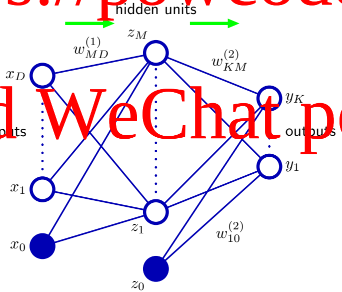


Functional Transformations

- Combine all transformations into one formula

$$y_k(\mathbf{x}, \mathbf{w}) = g\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

where $\mathbf{w}$ contains all weight and bias parameters.

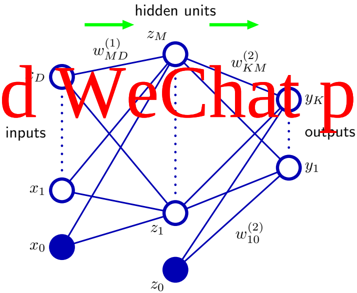


Functional Transformations

- As before, the biases can be absorbed into the weights by introducing an extra input $x_0 = 1$ and a hidden unit $z_0 = 1$.

$$y_k(\mathbf{x}, \mathbf{w}) = g\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

where $\mathbf{w}$ now contains all weight and bias parameters.

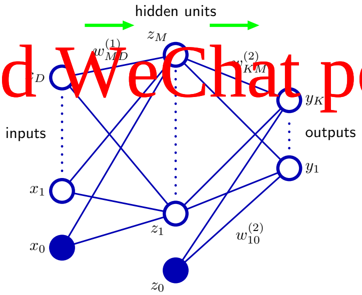


Comparison to Perceptron

- A neural network looks like a multilayer perceptron.
- But perceptron's nonlinear activation function was a step function — neither smooth nor differentiable.

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

- The activation functions $h(\cdot)$ and $g(\cdot)$ of a neural network are smooth and differentiable.



Linear Activation Functions

- If all activation functions are linear functions then there exists an equivalent network without hidden units. (Composition of linear functions is a linear function.)
- But if the number of hidden units in this case is smaller than the number of input or output units, the resulting linear function are not the most general.
- Dimensionality reduction.
- *c.f.* Principal Component Analysis (upcoming lecture).
- Generally, most neural networks use nonlinear activation functions as the goal is to approximate a nonlinear mapping from the input space to the outputs.

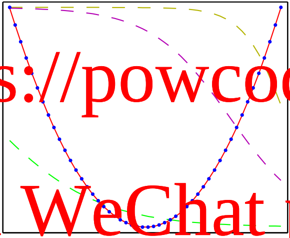
Neural Networks as Universal Function Approximators

- Feed-forward neural networks are universal approximators.
- Example: A two-layer neural network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy if it has enough hidden units.
- Holds for a wide range of hidden unit activation functions
- Remaining big question : Where do we get the appropriate settings for the weights from? With other words, how do we learn the weights from training examples?

Aproximation Capabilities of Neural Networks

- Neural network approximating

$$f(x) = x^2$$

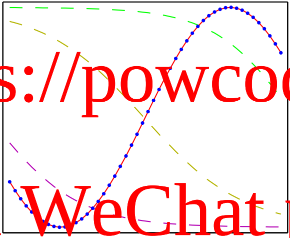


Two-layer network with 3 hidden units ($\tanh$ activation functions) and linear outputs trained on 50 data points sampled from the interval $(-1, 1)$. Red: resulting output. Dashed: Output of the hidden units.

Aproximation Capabilities of Neural Networks

- Neural network approximating

$$f(x) = \sin(x)$$

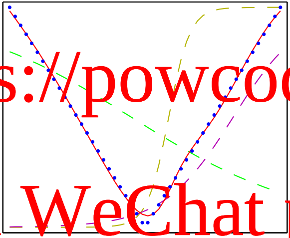


Two-layer network with 3 hidden units ($\tanh$ activation functions) and linear outputs trained on 50 data points sampled from the interval $(-1, 1)$. Red: resulting output. Dashed: Output of the hidden units.

Aproximation Capabilities of Neural Networks

- Neural network approximating

$$f(x) = |x|$$

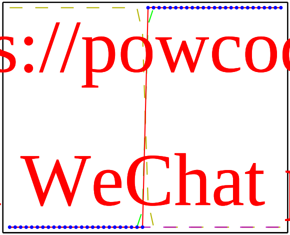


Two-layer network with 3 hidden units ($\tanh$ activation functions) and linear outputs trained on 50 data points sampled from the interval $(-1, 1)$. Red: resulting output. Dashed: Output of the hidden units.

Aproximation Capabilities of Neural Networks

- Neural network approximating Heaviside function

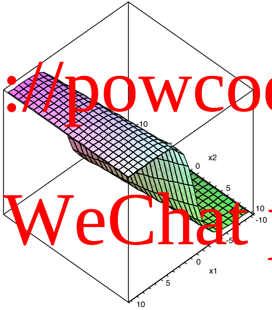
$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



Two-layer network with 3 hidden units ($\tanh$ activation functions) and linear outputs trained on 50 data points sampled from the interval $(-1, 1)$. Red: resulting output. Dashed: Output of the hidden units.

Variable Basis Functions in a Neural Networks

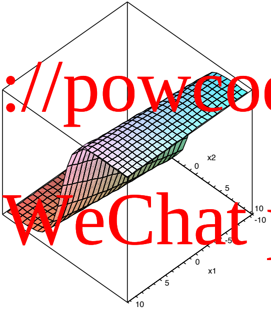
- Hidden layer nodes represent parametrised basis functions



$z = \sigma(w_0 + w_1 x_1 + w_2 x_2)$ for $(w_0, w_1, w_2) = (0.0, 1.0, 0.1)$

Variable Basis Functions in a Neural Networks

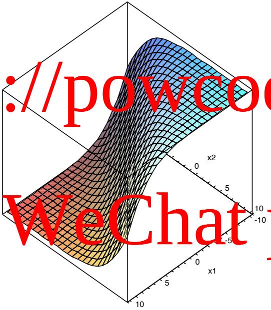
- Hidden layer nodes represent parametrised basis functions



$z = \sigma(w_0 + w_1 x_1 + w_2 x_2)$ for $(w_0, w_1, w_2) = (0.0, 0.1, 1.0)$

Variable Basis Functions in a Neural Networks

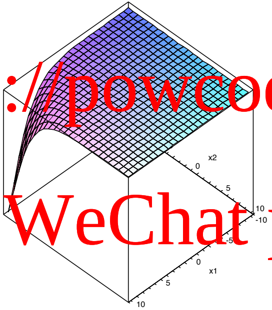
- Hidden layer nodes represent parametrised basis functions



$z = \sigma(w_0 + w_1 x_1 + w_2 x_2)$ for $(w_0, w_1, w_2) = (0.0, -0.5, 0.5)$

Variable Basis Functions in a Neural Networks

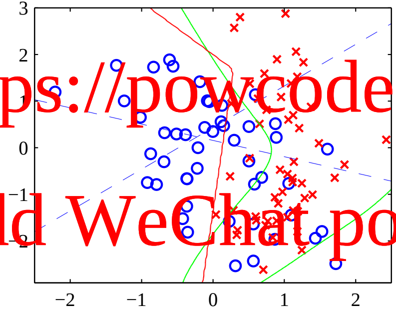
- Hidden layer nodes represent parametrised basis functions



$z = \sigma(w_0 + w_1 x_1 + w_2 x_2)$ for $(w_0, w_1, w_2) = (10.0, -0.5, 0.5)$

Aproximation Capabilities of Neural Networks

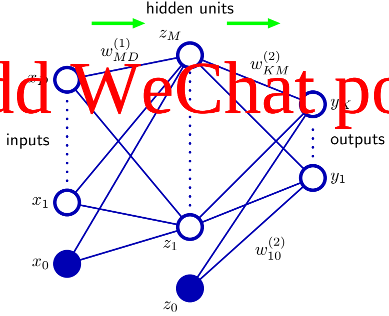
- Neural network for two-class classification.
- 2 inputs, 2 hidden units with $\tanh$ activation function, 1 output with logistic sigmoid activation function.



Red: $y = 0.5$ decision boundary. Dashed blue: $z = 0.5$ hidden unit contours. Green: Optimal decision boundary from the known data distribution.

Weight-space Symmetries

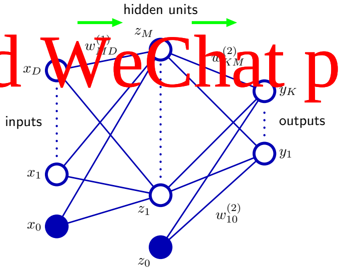
- Given a set of weights $\mathbf{w}$. This fixes a mapping from the input space to the output space.
- Does there exist another set of weights realising the same mapping?
- Assume $\tanh$ activation function for the hidden units. As $\tanh$ is an odd function: $\tanh(-a) = -\tanh(a)$.
- Change the sign of all inputs to a hidden unit and outputs of this hidden unit: Mapping stays the same.



Weight-space Symmetries

- M hidden units, therefore 2^M equivalent weight vectors.
- Furthermore, exchange all of the weights going into and out of a hidden unit with the corresponding weights of another hidden unit. Mapping stays the same. $M!$ symmetries.
- Overall weight space symmetry : $M!\,2^M$

M	1	2	3	4	5	6	7
$M!\,2^M$	2	8	48	384	3840	46080	645120



- Assume the error $E(\mathbf{w})$ is a smooth function of the weights.
- Smallest value will occur at a critical point for which

$$\nabla E(\mathbf{w}) = 0.$$

- This could be a minimum, maximum, or saddle point.
- Furthermore, because of symmetry in weight space, there are at least $M!\,2^M$ other critical points with the same value for the error.

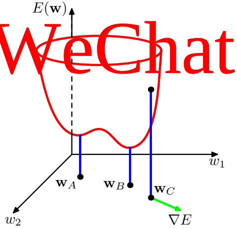
Parameter Optimisation

Definition (Global Minimum)

A point $\mathbf{w}^*$ for which the error $E(\mathbf{w}^*)$ is smaller than any other error $E(\mathbf{w})$.

Definition (Local Minimum)

A point $\mathbf{w}^*$ for which the error $E(\mathbf{w}^*)$ is smaller than any other error $E(\mathbf{w})$ in some neighbourhood of $\mathbf{w}^*$.

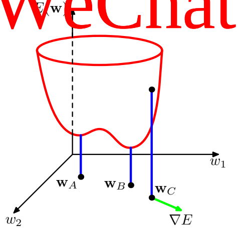


Parameter Optimisation

- Finding the global minimum is difficult in general (would have to check everywhere) unless the error function comes from a special class (e.g. smooth convex functions have only one local minimum).

- Error functions for neural networks are not convex (symmetries!).
- But finding a local minimum might be sufficient.
- Use iterative methods with weight vector update $\Delta\mathbf{w}^{(\tau)}$ to find a local minimum.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$



Local Quadratic Approximation

- Around a stationary point $\mathbf{w}^*$ we can approximate

$$E(\mathbf{w}) \simeq E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^*),$$

where the Hessian $\mathbf{H}$ is evaluated at $\mathbf{w}^*$ so that

$$H_{ij} = \left. \frac{\partial^2}{\partial w_i \partial w_j} \right|_{\mathbf{w}=\mathbf{w}^*} E(\mathbf{w}).$$

- Using a set $\{\mathbf{u}_i\}$ of orthonormal eigenvectors of $\mathbf{H}$,

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i,$$

to expand

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{u}_i.$$

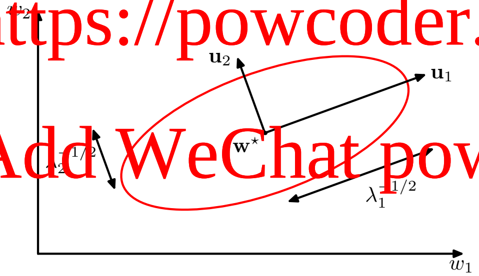
- We get

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2.$$

Local Quadratic Approximation

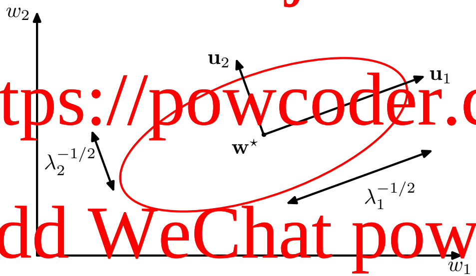
- Around a minimum $\mathbf{w}^*$ we can approximate

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2.$$



Local Quadratic Approximation

- Around a minimum $\mathbf{w}^*$, the Hessian $\mathbf{H}$ must be positive definite if evaluated at $\mathbf{w}^*$.



- This explains why the Laplace approximation always yields a valid covariance matrix.

Gradient Information improves Performances

- Hessian is symmetric and contains $W(W+1)/2$ independent entries where W is the total number of weights in the network.
- If we use function evaluations only:
  - Need to gather this $O(W^2)$ pieces of information by doing $O(W^2)$ number of function evaluations each of which cost $O(W)$ time, for an overall cost of order $O(W^3)$.
- If we use gradients of the function:
  - Surprisingly the gradient ∇E also costs only $O(W)$ time, although it provides W pieces of information.
  - We now need only $O(W)$ steps, so the order of time complexity is reduced to $O(W^2)$.

FYI only: In general we have the “cheap gradient principle”. See (Griewank, A., 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Section 5.1).

Gradient Descent Optimisation

- Batch processing : Update the weight vector with a small step in the direction of the negative gradient

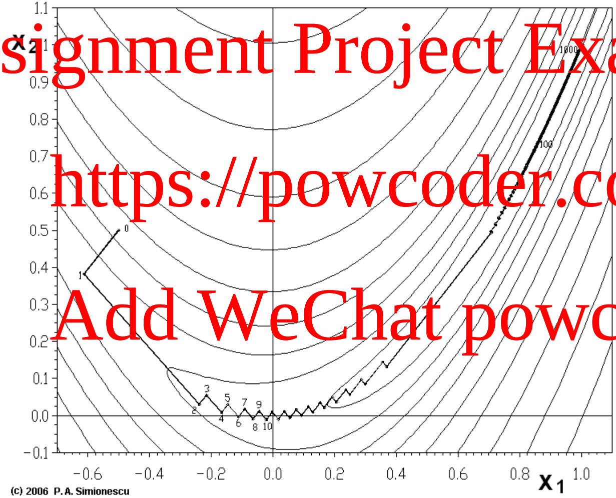
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

where η is the learning rate.

- After each step, re-evaluate the gradient $\nabla E(\mathbf{w}^{(\tau)})$ again.
- Gradient Descent has problems in long valleys.

Gradient Descent Optimisation

- Gradient Descent has problems in 'long valleys'.



Example of zig-zag of Gradient Descent Algorithm.

Nonlinear Optimisation

- Use Conjugate Gradient Descent instead of Gradient Descent to avoid zig-zag behaviour.
- Use Newton method which also calculates the inverse Hessian in each iteration (but inverting the Hessian is usually costly).
- Use Quasi-Newton methods (e.g. BFGS) which also calculates an estimate of the inverse Hessian while iterating.
- Even simpler are *momentum* based strategies.
- Run the algorithm from a set of starting points to find the smallest local minimum.

Nonlinear Optimisation

- Remaining big problem: Error function is defined over the whole training set. Therefore, need to process the whole training set for each calculation of the gradient $\nabla E(\mathbf{w}^{(\tau)})$.
- If the error function is a sum of errors for each data point

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

we can use on-line gradient descent (also called sequential gradient descent or stochastic gradient descent) to update the weights by one data point at a time

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}).$$

FYI Only: NN Extensions

- Add more hidden layers (deep learning). To make it work we need many of the following tricks:
- Clever weight initialisation to ensure the gradient is flowing through the entire network.
- Some links may additively skip over one or several subsequent layer(s).
- Favour ReLU over e.g. the sigmoid, to avoid vanishing gradients.
- Clever regularisation methods such as dropout.
- Specific architectures, not further considered here:
  - Parameters may be shared, notably as in convolutional neural networks for images.
  - A state space model with neural network transitions is a recurrent neural network.
  - Attention mechanisms learn to focus on specific parts of an input.