

STK2100- Machine Learning and Statistical Methods for Prediction and Classification

Spring 2026

Neural networks

Covers Sections 11.1, 11.3-11.8 and 11.10

Ingrid Hobæk Haff
Department of Mathematics
University of Oslo

Neural networks

- We have already seen that we can construct more flexible models for regression and classification through the construction of derived covariates
- **Neural networks** are constructed as non-linear transformations of linear combinations of the covariates
- As opposed to basis expansions in Chapter 5, the parameters of the basis functions are learned from data
- This results in very flexible models for prediction tasks, where interpretation is not important

Neural networks

- Recall principal components regression, where we constructed new covariates $z_1, \dots, z_M$, as linear combinations of the original covariates $x_1, \dots, x_p$:

$$z_{im} = \sum_{j=1}^p \alpha_{jm} x_{ij} = \boldsymbol{\alpha}_m^T \mathbf{x}_i \quad m = 1, \dots, M$$
$$y_i = \beta_0 + \sum_{m=1}^M \beta_j z_{im} + \varepsilon_i = \underbrace{\beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i}_{T_i} + \varepsilon_i$$

- Here, the z_{im} s can be considered as **latent** or **hidden variables**

Neural networks

- This can be generalised to a **one-layer neural network**, also called **one-layer perceptron**:

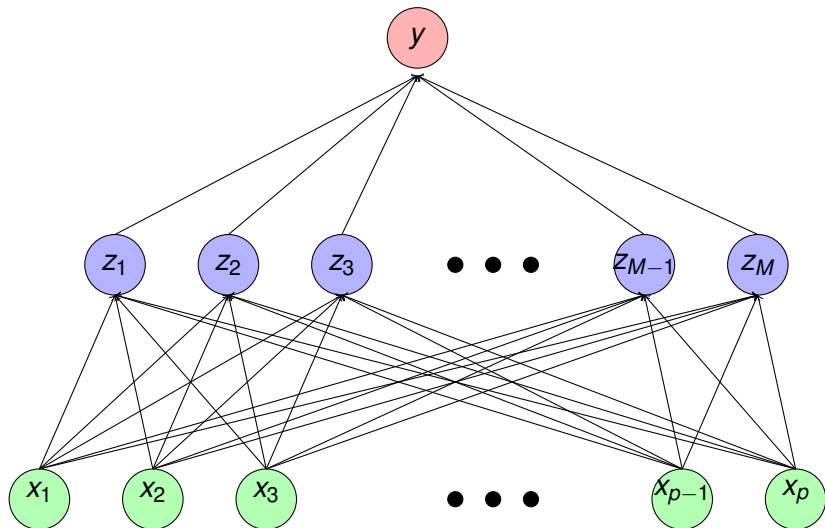
$$z_{jm} = \sigma(\alpha_j^T \mathbf{x}_i), \quad j = 1, \dots, M$$

$$T_i = \beta_0 + \beta^T \mathbf{z}_i$$

$$Y_i = g(T_i) + \varepsilon_i = g(\beta_0 + \beta^T \mathbf{z}_i) + \varepsilon_i$$

- The α_{jm} s and β_m s are called **weights**
- The z_{jm} s are called **hidden units** because they are not directly observed
- Both the **activation function** $\sigma(\cdot)$ and the **output function** $g(\cdot)$ can be non-linear
- The x_j s constitute the **input layer**, the z_m s the **hidden layer** and Y the **output layer**

Visualisation of neural network with one hidden layer



Rewriting the model

- We have

$$\begin{aligned}Y_i &= g\left(\beta_0 + \sum_{m=1}^M \beta_m z_{im}\right) + \varepsilon_i \\&= g\left(\beta_0 + \sum_{m=1}^M \beta_m \sigma\left(\sum_{j=1}^p \alpha_{mj} x_{ij}\right)\right) + \varepsilon_i \\&= f(\mathbf{x}_i) + \varepsilon_i\end{aligned}$$

with

$$f(\mathbf{x}_i) = g\left(\beta_0 + \sum_{m=1}^M \beta_m \sigma\left(\sum_{j=1}^p \alpha_{mj} x_{ij}\right)\right)$$

- Neural nets are usually presented through predictions, without specifying the distribution of the ε_i s, i.e.

$$\begin{aligned}z_{im} &= \sigma(\alpha_j^T \mathbf{x}_i), \quad m = 1, \dots, M, \quad \text{and} \quad T_i = \beta_0 + \boldsymbol{\beta}^T \mathbf{z}_i \\ \hat{Y}_i &= g(T_i) = f(\mathbf{x}_i)\end{aligned}$$

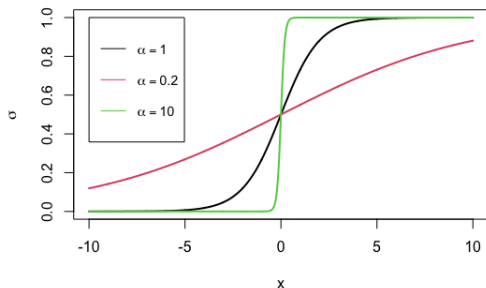
Choice of σ and g

- There are many possible choices for the activation and output functions
- The most common in the regression setting are
 - the **sigmoid** function for σ : $\sigma(x) = \frac{1}{1+\exp(-x)}$
 - the **identity** function for g : $g(z) = z$
- Alternatives for σ are
 - the **hyperbolic tangent**: $\sigma(x) = \tanh(x)$
 - the **RELU** function: $\sigma(x) = \max\{0, x\}$
 - the **leaky RELU**: $\sigma(x) = \max\{ax, x\}$, with $0 < a < 1$

The sigmoid function

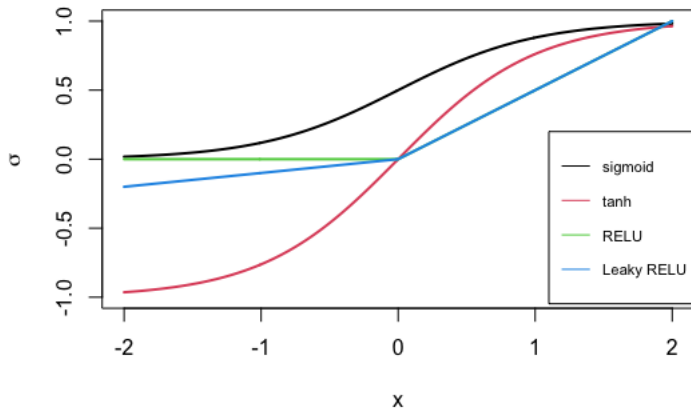
- The sigmoid function has the form

$$\sigma(x) = \frac{1}{1 + \exp(-\alpha x)}$$



- For small values of α , it is almost linear
- For large values of α , it is almost a threshold function

Comparison between different activation functions



Estimation of parameters

- Neural networks have many parameters that must be estimated:

$$\theta = \{\beta_0, \beta_m, \alpha_{jm}, m = 1, \dots, M, j = 1, \dots, p\}$$

- Criterion:

$$R(\theta) = \sum_{i=1}^N (y_i - f(\mathbf{x}_i))^2 = \sum_{i=1}^N \left(y_i - g \left(\beta_0 + \sum_{m=1}^M \beta_m \cdot \sigma \left(\sum_{j=1}^p \alpha_{jm} x_{ij} \right) \right) \right)^2$$

- Due to possible overfitting, we introduce a penalty term

$$R_0(\theta) = R(\theta) + \lambda J(\theta), \quad \text{with} \quad J(\theta) = \sum_{m=1}^M \sum_{j=1}^p \alpha_{jm}^2 + \sum_{m=1}^M \beta_m^2$$

- For the regularisation to affect the covariates equally, the covariates should be standardised before fitting the neural network

Gradient descent and back-propagation

- Estimation of the parameters is typically done through **gradient descent methods**
- In order, to minimise $R_0(\theta)$ w.r.t. θ , we compute the derivatives, assuming that both $\sigma(\cdot)$ and $g(\cdot)$ are smooth and have smooth derivatives
- We get

$$\frac{\partial R_0(\theta)}{\partial \beta_{m^*}} = -2 \sum_{i=1}^N (y_i - f(\mathbf{x}_i)) \frac{\partial f(\mathbf{x}_i)}{\partial \beta_{m^*}} + 2\lambda \beta_{m^*}$$

with

$$\frac{\partial f(\mathbf{x}_i)}{\partial \beta_{m^*}} = g' \left(\beta_0 + \sum_{m=1}^M \beta_m \sigma \left(\sum_{j=1}^p \alpha_{jm} x_{ij} \right) \right) \cdot \sigma \left(\sum_{j=1}^p \alpha_{jm^*} x_{ij} \right) = g'(T_i) z_{im^*}$$

Gradient descent and back-propagation

- Further,

$$\frac{\partial R_0(\theta)}{\partial \alpha_{j^*m^*}} = -2 \sum_{i=1}^N (y_i - f(\mathbf{x}_i)) \frac{\partial f(\mathbf{x}_i)}{\partial \alpha_{j^*m^*}} + 2\lambda \alpha_{j^*m^*}$$

with

$$\begin{aligned} \frac{\partial f(\mathbf{x}_i)}{\partial \alpha_{j^*m^*}} &= g' \left(\beta_0 + \sum_{m=1}^M \beta_m \sigma \left(\sum_{h=1}^p \alpha_{hj} x_{ih} \right) \right) \cdot \beta_{m^*} \cdot \sigma' \left(\sum_{j=1}^p \alpha_{jm^*} x_{ij} \right) \cdot x_{ij^*} \\ &= g'(T_i) \cdot \beta_{m^*} \cdot \sigma' \left(\sum_{j=1}^p \alpha_{jm^*} x_{ij} \right) \cdot x_{ij^*} \end{aligned}$$

- Gradient descent is then performed by iterating the following parameter updates until convergence (to a local optimum)

$$\begin{aligned} \beta_m^{r+1} &= \beta_m^r - \gamma_r \frac{\partial R_0(\theta^r)}{\partial \beta_m} \\ \alpha_{jm}^{r+1} &= \alpha_{jm}^r - \gamma_r \frac{\partial R_0(\theta^r)}{\partial \alpha_{jm}} \end{aligned}$$

Gradient descent and back-propagation

- The iterated parameter updates can be implemented as a two-pass procedure, known as **back-propagation**
 - The **forward pass** consists in calculating the z_{im} s and T_i s
 - The **backward pass** consists in calculating the $g'(T_i)$ s and $\sigma' \left(\sum_{j=1}^p \alpha_{jm} x_{ij} \right)$ s
- The tuning parameter γ_r is called the **learning rate**
- The learning rate is usually constant, but can be a sequence decreasing slowly towards zero
- Convergence can be made faster by using second derivatives, but this requires more computation in each iteration

Extensions to several response variables and to classification

- Neural nets can be extended to handle several response variables $Y_1, \dots, Y_K$ through

$$z_{im} = \sigma(\alpha_m^T \mathbf{x}_i), \quad m = 1, \dots, M$$

$$T_{ik} = \beta_{0k} + \beta_k^T \mathbf{z}_i, \quad k = 1, \dots, K$$

$$y_{ik} = g(T_{ik}) + \varepsilon_{ik} = g(\beta_{0k} + \beta_k^T \mathbf{z}_i) + \varepsilon_{ik}, \quad k = 1, \dots, K$$

- For classification, one starts by defining the dummy variables

$$y_{ik} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases}$$

so that $y_i = \arg \max_k y_{ik}$

Extensions to several response variables and to classification

- The predictions of y_i and $\Pr(Y_i = k|\mathbf{x}_i)$ are then given by

$$z_{im} = \sigma(\boldsymbol{\alpha}_m^T \mathbf{x}_i), \quad m = 1, \dots, M$$

$$T_{ik} = \beta_{0k} + \boldsymbol{\beta}_k^T \mathbf{z}_i, \quad k = 1, \dots, K$$

$$\hat{y}_i = \arg \max_k g(T_{ik})$$

$$\hat{\Pr}(Y_i = k|\mathbf{x}_i) = g(T_{ik}) = f_k(\mathbf{x}_i)$$

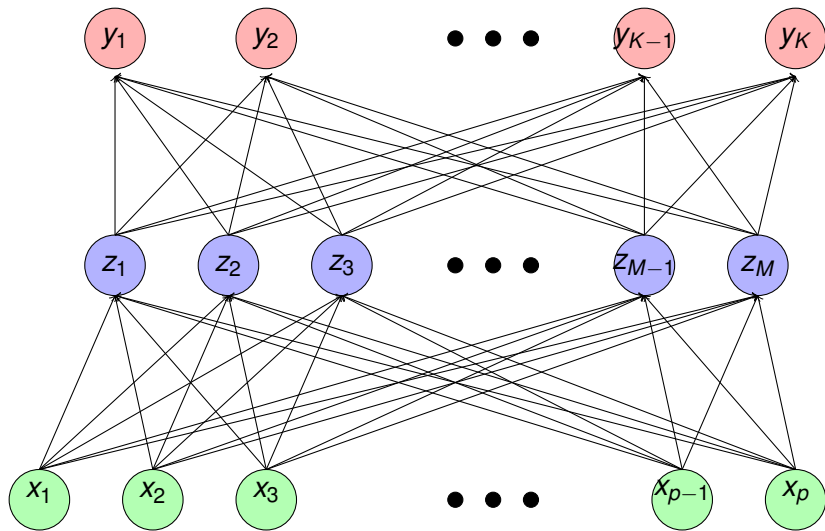
- When using neural networks for classification, one typically uses the **softmax** function as the output function, i.e.

$$g(T_{ik}) = \frac{e^{T_{ik}}}{\sum_{l=1}^K e^{T_{il}}}$$

- Further, the **cross-entropy** can be used as the loss function

$$R(\boldsymbol{\theta}) = - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log f_k(\mathbf{x}_i)$$

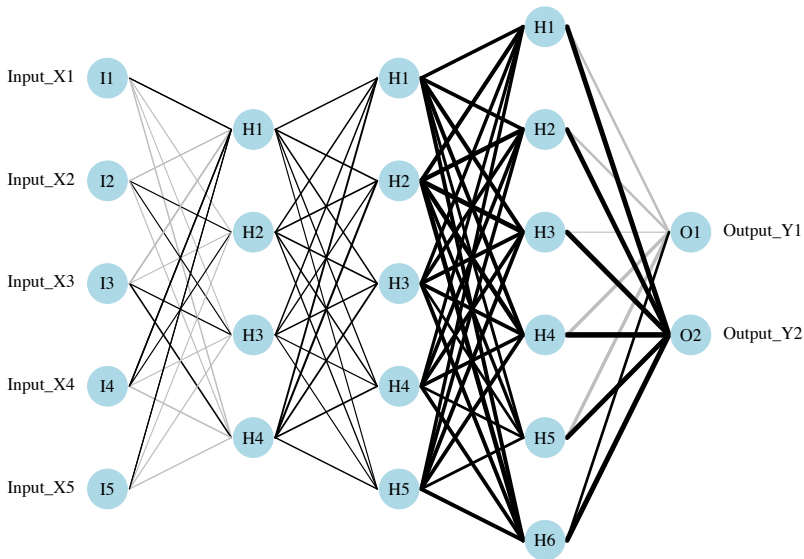
Neural net for K -class classification



Deep learning – multi-layer neural networks

- So far, we have only considered neural networks with one layer of latent variables
- In principle, these have the **universal approximation property**, meaning that they can approximate most functions well, as long as they are large enough
- In practice, it is much easier to find a good solution by constructing a neural network with several layers of smaller size
- This gives more flexibility and more possibilities for interactions and non-linearities

Illustration of multi-layer neural network



Mathematical formulation

- We can still write

$$y_i = f(\mathbf{x}_i) + \varepsilon_i,$$

but now with

$$f(\mathbf{x}_i) = g(\dots\sigma_{L-1}(\dots\sigma_{L-2}(\dots)))$$

- Expressions for the gradients are obtained using the chain rule, and are similar to the ones for one-layer networks
- Define $\mathbf{z}_L$ to be the variables from the last hidden layer
- Then the network model can be seen as a model g applied to the derived covariates $\mathbf{z}_L$
- This is then a complex method for constructing derived covariates

Challenges with neural networks

- Neural networks are very flexible models that may have many parameters, which can lead to overfitting
- The fitting criterion R is nonconvex with many local minima
- When the weights are small, the model is almost linear
- Hence, one can start with small weights, and introduce nonlinearities when needed
- Another option is to try different starting values
- One could also use several networks for prediction

Avoiding overfit

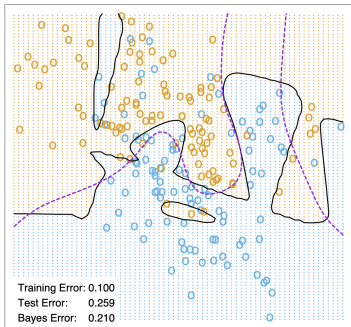
- There are several strategies to avoid overfit
- Large weights tend to cause sharp transitions in the activation functions, and thus large changes in output for small changes in inputs
- Hence, one strategy is to use a penalty term/**weight decay**

$$J(\theta) = \sum_{m=1}^M \sum_{j=1}^p \alpha_{jm}^2 + \sum_{m=1}^M \beta_m^2$$

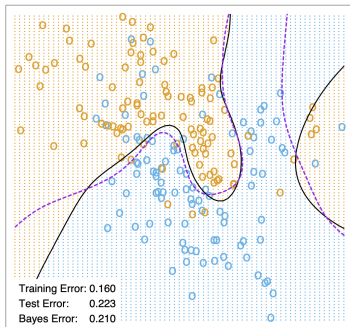
- Another strategy is **early stopping** to avoid reaching a global minimum
- Yet another option is **dropout**, where some of the weights are turned off during training
- Finally, one may **add noise** to the inputs

Illustration of weight decay

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Properties neural networks

- Advantages of neural networks:
 - They are very flexible
 - Impressive results have been obtained in many large scale applications, such as image analysis and text processing
- Disadvantages:
 - Fitting can be time-consuming and require large computer resources
 - They are unstable in the estimation stage
 - It is difficult to choose the number of latent variables and the penalty value λ
 - It is difficult to obtain standard errors and uncertainty measures
 - They are difficult to interpret
- They typically work well for large training data with large signal to noise ratio
- For tabular data, alternative methods, such as gradient boosted trees, are often better

Example: Simulated example from Section 11.6

- Data sets with
 - training sample 100
 - test sample 10 000

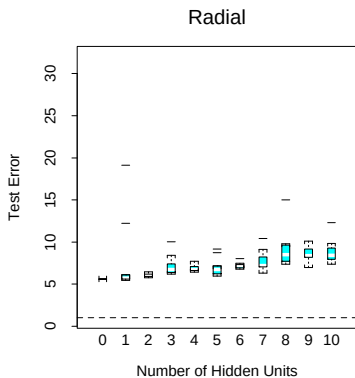
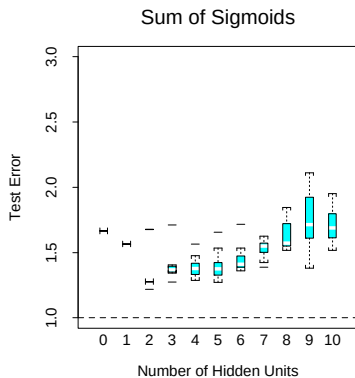
were simulated from two models

$$Y = \sigma(\mathbf{a}_1^T \mathbf{X}) + \sigma(\mathbf{a}_2^T \mathbf{X}) + \varepsilon_1; \quad p = 2$$

$$Y = \prod_{m=1}^{10} \phi(X_m) + \varepsilon_2 \quad p = 10$$

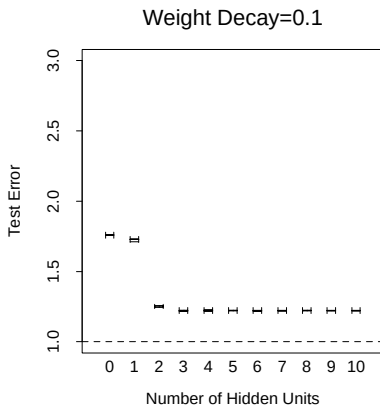
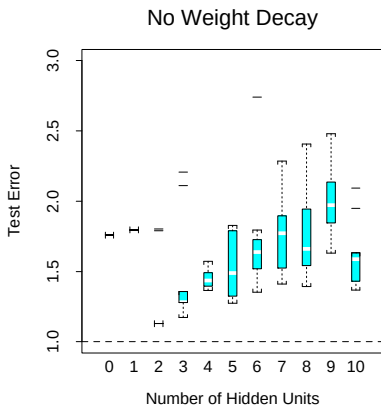
- The parameter values were set such that the signal to noise ratio $\frac{\text{Var}(f(\mathbf{X}))}{\text{Var}(\varepsilon)}$ is approximately the same for the two models
- Neural networks with 10 random starting weights were trained on the training sets, and the test error $E_{\text{Test}}(Y - \hat{f}(\mathbf{X}))^2$ was computed

Example: Simulated example from Section 11.6



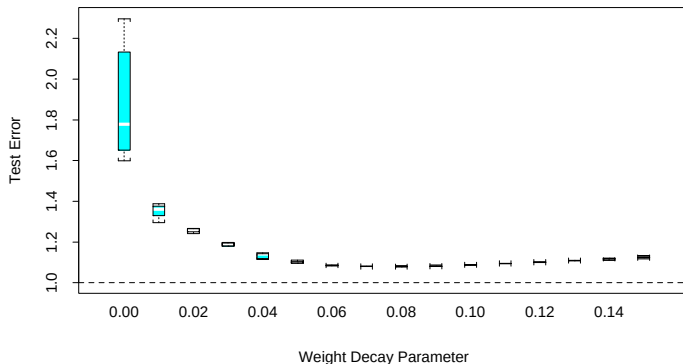
- Sum of sigmoids:
 - Zero hidden units corresponds to linear regression
 - Typical U-shape
- Radial function
 - Difficult for neural networks
 - Spherically symmetric, no preferred direction
 - Constant fit seems to do best

Example: Simulated example from Section 11.6: sigmoids with no/strong decay



Example: Simulated example from Section 11.6: sigmoids with varying decay

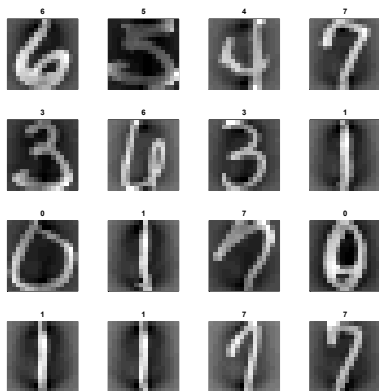
Sum of Sigmoids, 10 Hidden Unit Model



- There are two free parameters:
 - the weight decay λ
 - the number of hidden units M
- Learning strategy:
 - fix one of the parameters at the least constrained model
 - use cross-validation to choose the other

Example: Zip data

- These are 16×16 greytone images of handwritten numbers
- Response: written digit $\in \{0, 1, \dots, 9\}$ (c1)
- Explanatory variables: intensities of the $p = 256$ pixels ($x_1 - x_{256}$)



We want to predict the written digit based on the pixel intensities

See R code