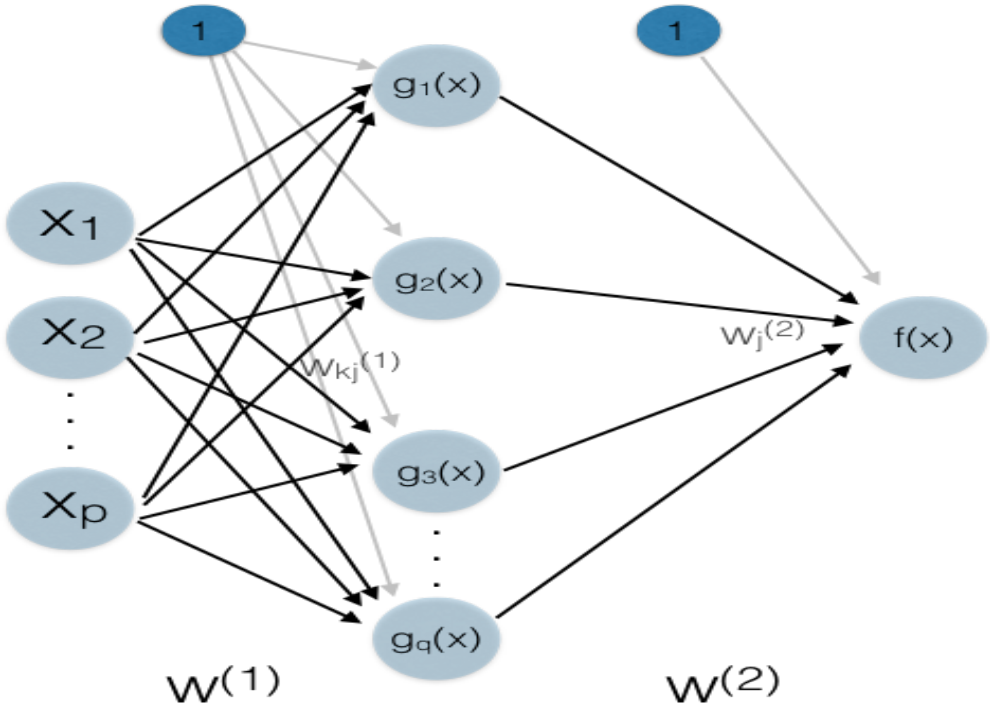
STATS 235: Modern Data Analysis
Neural Networks

Babak Shahbaba

Department of Statistics, UCI

Neural Networks (NN)



- Multilayer perceptron (MLP) with p input variables, one hidden layer with q hidden units, and a single output.
- Here, $w^{(1)}$ represents the connection weight matrix between the input layer and the hidden layer, and $w^{(2)}$ is the vector of connection weights between the hidden layer and the output.

Neural Networks (NN)

- A multilayer perceptron (MLP, aka feedforward NN) is comprised of an input layer, output layer and a number of hidden layers in between
- The hidden layers creates a set of basis by applying nonlinear transformations, g, to their input and pass their results to the next layer until we reach the output layer.
- We refer to g as the activation or transfer function, which is usually set to the sigmoid (aka logistic) function

$$\text{sigm}(a) = \frac{1}{1 + e^{-a}}$$

or the hyperbolic tangent function

$$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$

Neural Networks (NN)

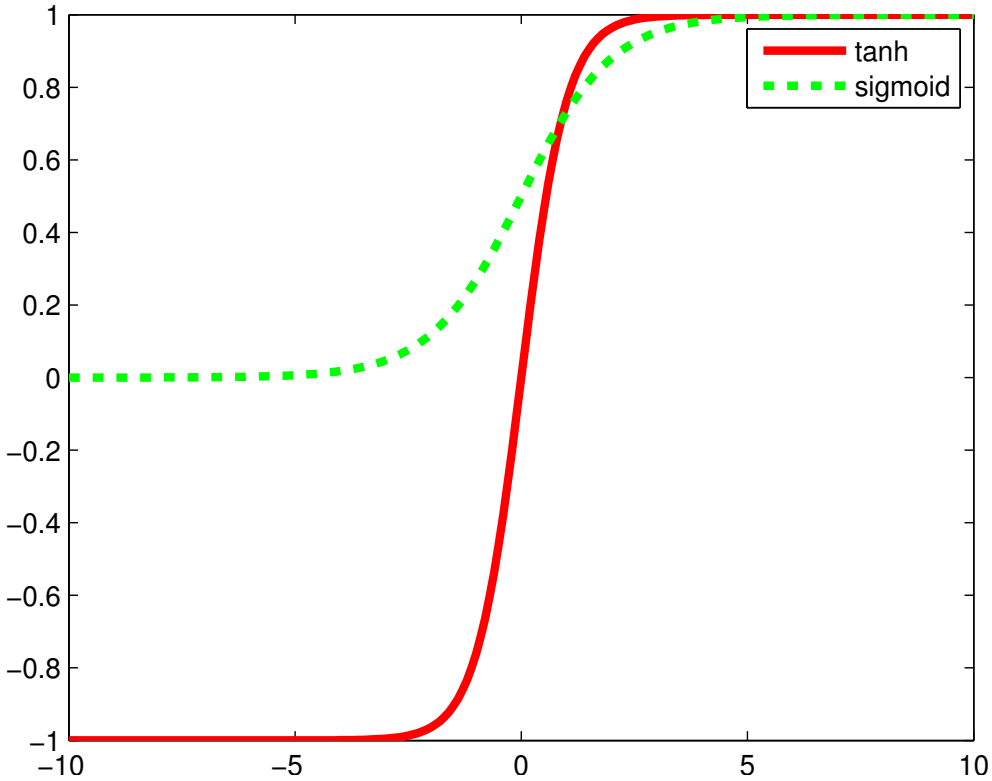


Figure 16.6 in Murphy (2012).

Neural Networks (NN)

- The output $f(x)$, which is a function used to approximate y, is a linear combination of basis defined by the hidden layers
- For linear regression models (continuous outcome)

$$P(y|x, w) = N(y|f(x), \sigma^2)$$

- For logistic regression models (binary outcome)

$$P(y|x, w) = \mathrm{Ber}(y|\mathrm{sigm}(f(x)))$$

- For multiple categories, we use the multinomial logit model, which is also known as the softmax function

Neural Networks (NN)

- For a MLP with one hidden layer and tanh activation function, we have

$$
\begin{aligned}
g_j(x) &= \tanh[w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k], \qquad \text{for } j = 1, \ldots, q \\
f(x) &= w_0^{(2)} + \sum_{j=1}^{q} w_j^{(2)} g_j(x)
\end{aligned}
$$

- Here w_0's, which play the role of the intercept in regression models, are called biases

Learning

- To train a MLP, we first need to specify the negative log-likelihood, which is also known as the energy function, E
- For regression, we have (squared error)

$$E = \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

for binary classification models, we have (cross entropy)

$$E = -\sum_{i=1}^{n} y_i [\log \operatorname{sigm}(f(x_i))] + (1 - y_i)[\log(1 - \operatorname{sigm}(f(x_i)))]$$

- To estimate the weights, we minimize the energy function with respect to w

Learning

- The parameters of a neural network model are not identifiable
  - Permuting the order of hidden units does not change the model
  - If we change the sign of weights entering a hidden unit, the model remains the same as long as we also change the sign of the weights going out of that unit since $\tanh(-a) = -\tanh(a)$
- Also, in general the energy function for MLP is non-convex
- Nevertheless, we can still use common iterative optimization methods (e.g., gradient descent algorithms) to obtain locally optimal estimates
- Using the chain rule, it is easy to find the gradient

Backpropagation

- For learning (i.e., estimating the weights), we start with initializing the weights (including the biases) to some random numbers (all different values) and iteratively perform the following steps
  - At each iteration, we use *forward propagation* to find the values going to each unit, before and after transformation, until we reach the output layer
  - We find the derivatives of E with respect to each unit starting from the output and *backpropagate* using the chain rule to find the derivatives with respect to hidden units
  - Using the chain rule again, we find the derivatives with respect to the weights
  - We then update the weights by moving in the direction of the negative gradient (see the notes on optimization)
  - We repeat the above steps until some stopping criterion is reached

Backpropagation

- For a MLP with one hidden layer and tanh activation function, given the current weights, forward propagation involves finding the following values at each hidden unit before and after transformation

$$\begin{aligned} z_j &= w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k \\ g_j &= \tanh(z_j) \end{aligned}$$

- For the output unit, we hve

$$f = w_0^{(2)} + \sum_{j=1}^{q} w_j^{(2)} g_j$$

Backpropagation

- Backpropagation starts with finding $\partial E/\partial f$; For regression model

$$\frac{\partial E}{\partial f} = -\frac{1}{\sigma^2}\sum_{i=1}^{n}(y - f(x_i))$$

- Next, using the chain rule we have

$$\frac{\partial E}{\partial g_j} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial g_j} = w_j^{(2)}\frac{\partial E}{\partial f}$$

- We then find the derivatives with respect to z_j

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = (1 - g_j^2)\frac{\partial E}{\partial g_j}$$

Recall that $\frac{d}{da}\tanh(a) = 1 - \tanh^2(a)$

Backpropagation

- Finally, we find the derivatives with respect to the weights
- For connection weights between the hidden layer and output we have

$$\frac{\partial E}{\partial w_j^{(2)}} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial w_j^{(2)}} = g_j \frac{\partial E}{\partial f}$$

Note that $g_0 = 1$ when evaluating $\partial E / \partial w_0^{(2)}$

- For the weights connecting the input layer to the hidden layer we have

$$\frac{\partial E}{\partial w_{kj}^{(1)}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{kj}^{(1)}} = x_k \frac{\partial E}{\partial z_j}$$

$x_0 = 1$ when evaluating $\partial E / \partial w_0^{(1)}$

Backpropagation

- After we find the gradient ∇E, we update the parameters by taking a step in a direction of negative gradient, with stepsize t

$$\begin{aligned} \Delta w &= -\nabla E \\ w &\leftarrow w + t\Delta w \end{aligned}$$

- The stepsize (aka learning rate) is found by trial-and-error
- We could run the algorithm until the approximation error falls below a desired threshold; however, this could lead to *overfitting*
- Two common strategies to avoid this issue are *early stopping* and *weight decay*

Early stopping

- Overfitting occurs when the model performs well on the training data and performs poorly on the test (future) data
- In the "early stopping" method, we start with some initial weights close to zero and monitor the performance of the neural network model throughout the training process based on an independent *validation set* (usually 20% of the data; this is separate from any test set used for model evaluation); we stop the algorithm when the model's performance on the validation set starts to decline substantially (a sign of overfitting)
- We can use the prediction error or average log probability on the validation set as a measure of performance
- This method could be very successful for avoiding overfitting; however it is *ad hoc* and wasteful since some of the data points are not used in the training directly

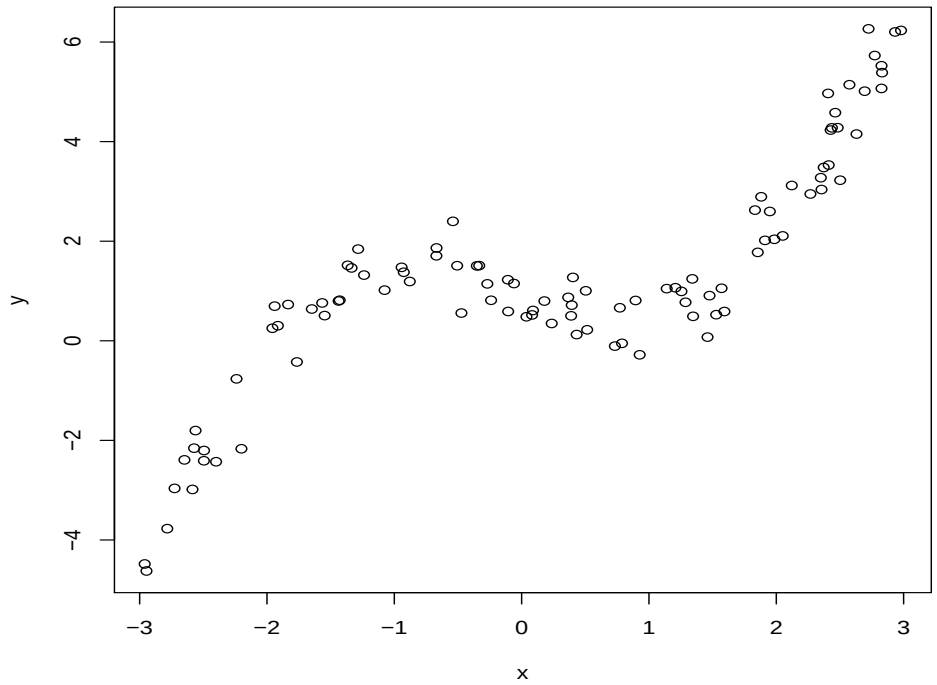
Weight decay

- Alternatively, to avoid overfitting, we can penalize models agains complexity (similar to ridge regression and Lasso)
- To this end, instead of minimizing the energy function, we minimize the penalized version of it by adding the following penalty terms:

$$\lambda_1 \sum_{k=1}^{p} \sum_{j=1}^{q} [w_{kj}^{(1)}]^2 + \lambda_2 \sum_{j=1}^{q} [w_j^{(2)}]^2$$

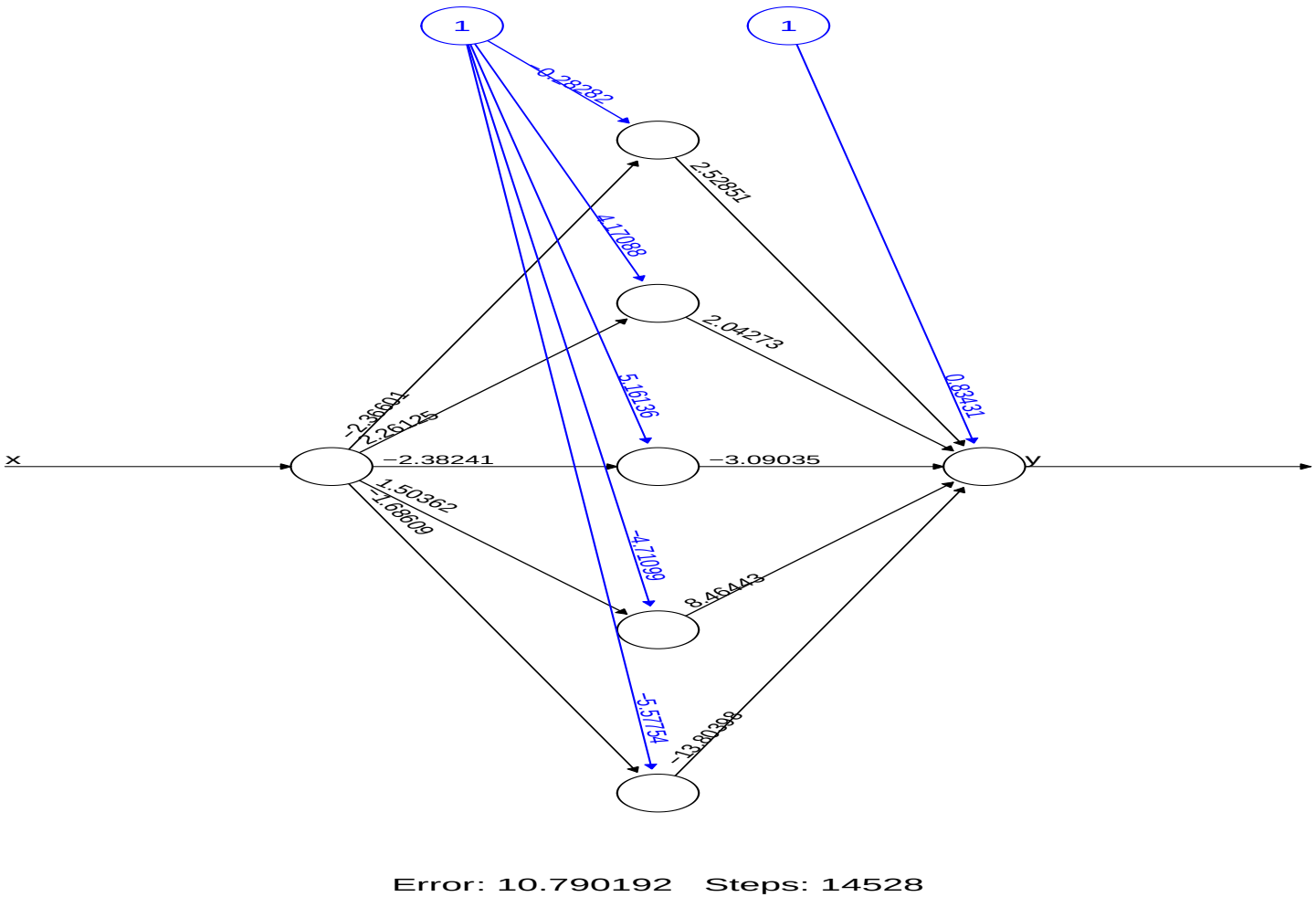
- This is known as "weight decay" since it shrinks the weights towards zero to encourage simpler models
- To set the values of λ_1 and λ_2, we can use an independent validation set as before or use cross-validation when the sample size is small

Illustrative Example



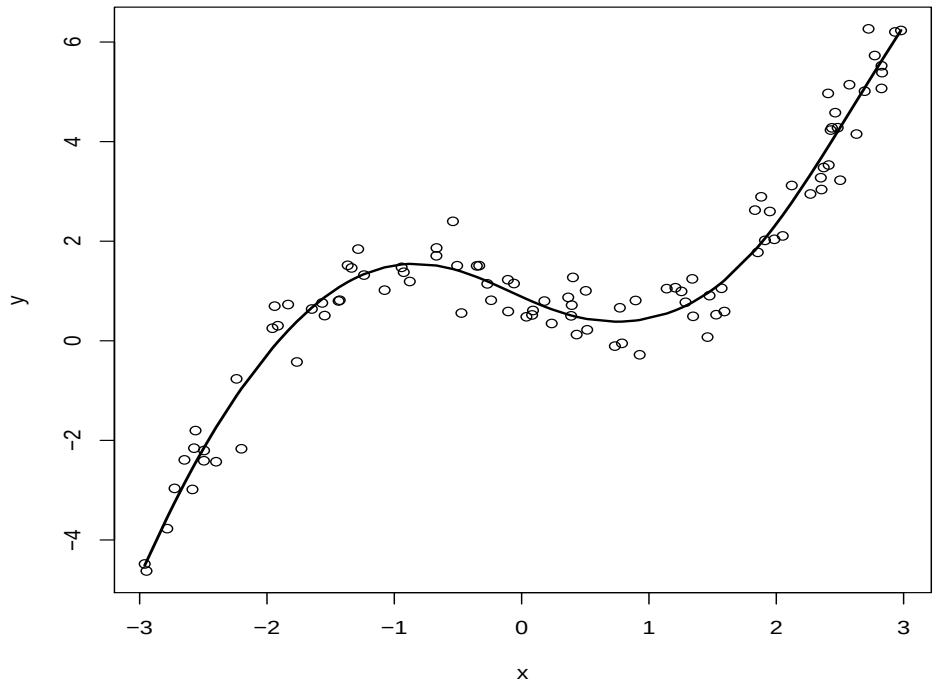
Observed data

Illustrative Example



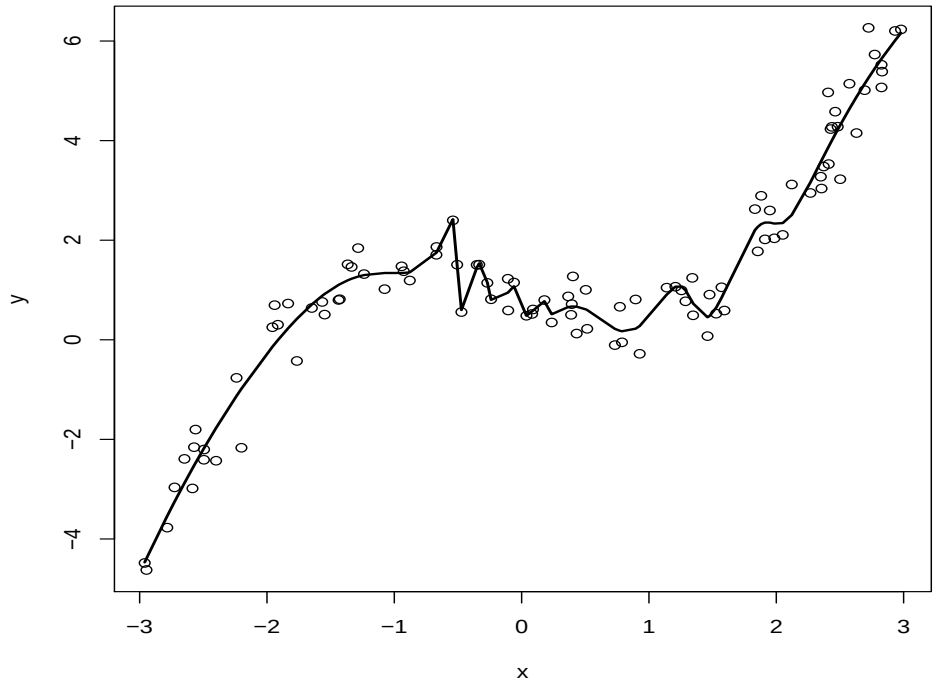
A neural network with 5 hidden units

Illustrative Example



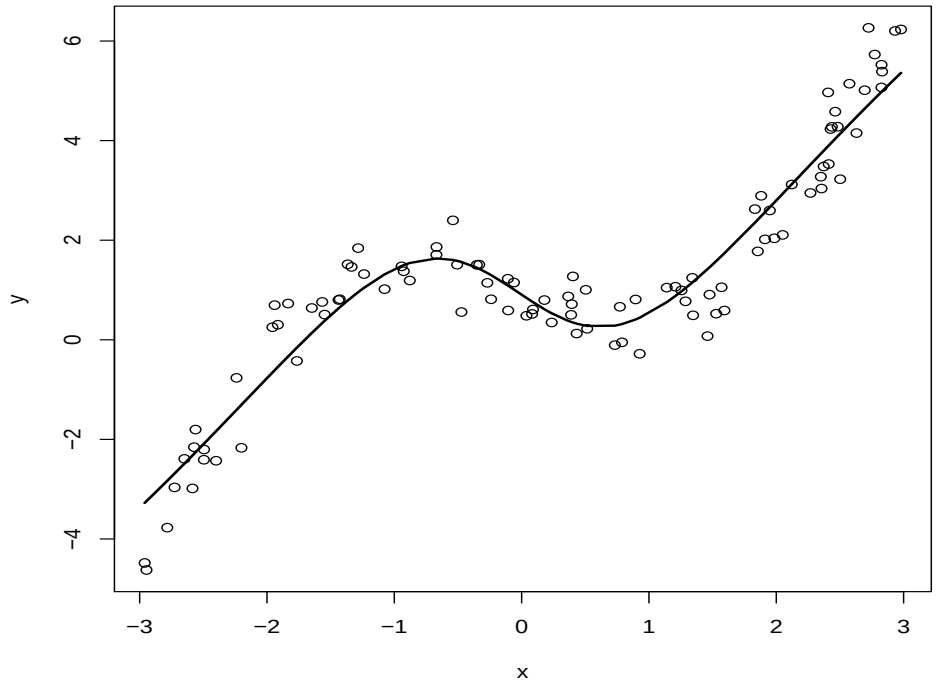
Estimated function, $f(x)$, with 5 hidden units

Illustrative Example



Estimated function, $f(x)$, with 50 hidden units

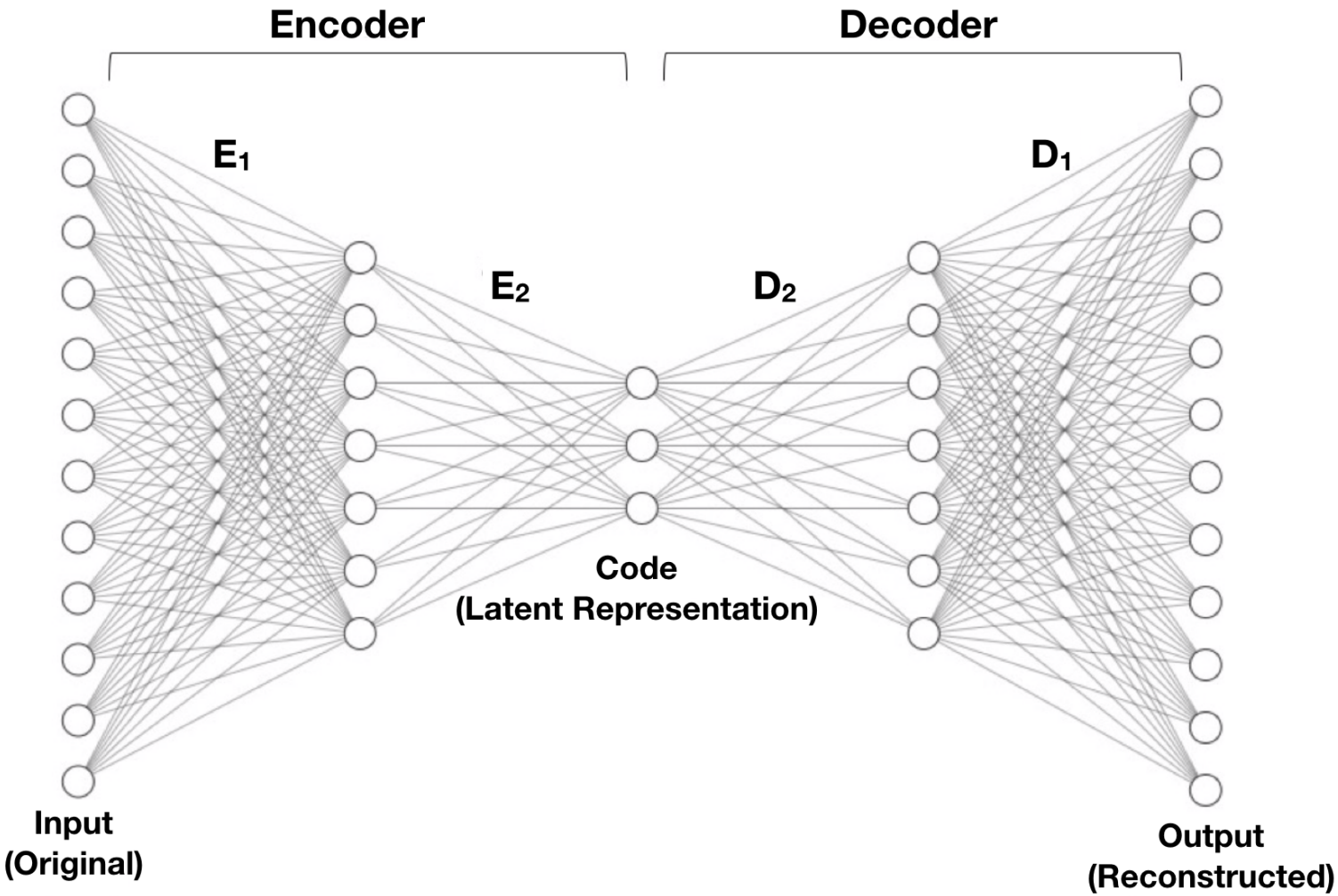
Illustrative Example



Estimated function, $f(x)$, with 50 hidden units using weight decay

Autoencoder

- Autoencoder is a special type of feed-forward neural network commonly used for learning latent representations of the data.



Autoencoder

- The data are fed from the input layer and encoded into a low-dimensional latent representation; this is known as *coding*.
- The *code* is then *decoded* into a reconstruction of the original data.
- The goal of an autoencoder is to learn an identity map such that the output (reconstruction) is close to the input (original) data.
- The model is trained to minimize the difference between the original and the reconstructed data.

Autoencoder

- Theoretically, an auto-encoder with suitable activation functions could represent an identity map.
- Denoting the encoder as ϕ and the decoder as ψ, idieally we want $\phi \circ \psi = I$.
- This way, an accurate reconstruction of the data implies a good low-dimensional representation at the bottleneck (middle layer).
- Therefore, autoencoders are commonly used for complicated nonlinear dimensionality reduction.
- See my codes based on the `h2o` package to build autoencoders.

Example

- The following plots shows low-dimensional representations of neural data obtained from rats learning a sequence of odors (A, B, C), and whether the odors are presented in the right sequence (InSeq) or out of sequence (OutSeq). The data are collected at the Fortin Lab at UCI.