Artificial Neural Networks

Carlos Valle

Departamento de Informática
Universidad Técnica Federico Santa María

cvalle@inf.utfsm.cl

December 23, 2015

Overview

Artificial Neural Networks

- ANN were inspired by scientists who attempt to answer questions such as:
  - What makes the human brain such a formidable machine in processing cognitive thought?
  - What is the nature of this thing called intelligence?
  - And, how do humans solve problems?
- There are many different theories and models for how the mind and brain work.

Connectionism

- One such theory, called connectionism, uses analogues of neurons and their connections together with the concepts of activation functions, and the ability to modify those connections to create learning algorithms.
- Now, ANNs are treated more abstractly, as a network of highly interconnected nonlinear computing elements.
- Problems of speech recognition, handwritten character recognition, face recognition, and robotics are important applications of ANNs.

Multilayer perceptron architecture

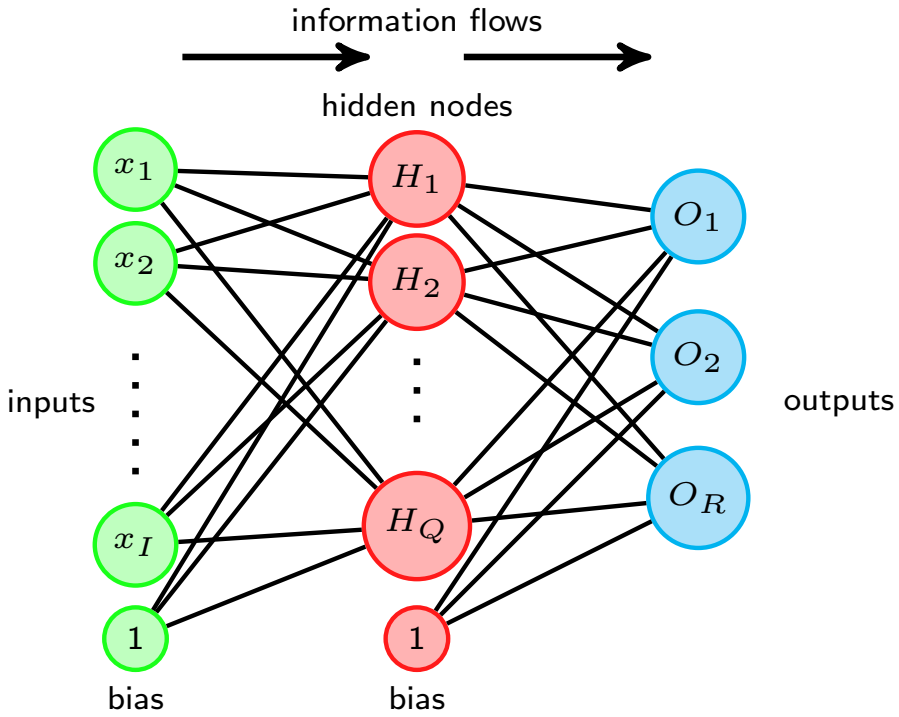


Figure 1: Schematic ANN feed-forward

- Each feature of a single input vector $\mathbf{x} = (x_1, \ldots, x_I)^T$ feeds a input layer node.

Multilayer perceptron architecture (2)

- Each node in the hidden layer contains a derivate feature H_q, which is a function of the linear combination of the inputs

$$\begin{aligned} H_q &= \sigma\left(w_{q0}^{(1)} + \sum_{i=1}^{I} w_{qi}^{(1)} x^{(i)}\right), \quad q = 1, ..., Q, \\ H_q &= \sigma\left({\mathbf{w}_q^{(1)}}^T \mathbf{x}\right), \quad q = 1, \ldots, Q, \end{aligned}$$

- where $H = (H_1, H_2, ..., H_Q)^T$, $w_{q0}^{(1)}$ is the hidden layer bias, $w_{qi}^{(1)}$ is the hidden layer weight, and $\sigma(\varphi)$ is the activation function which is usually chosen to be the sigmoid $\sigma(\varphi) = \frac{1}{1+e^{-\varphi}}$.
- As we have previously shown the sigmoid function has an operation range close to $[-1, 1]$. For this reason the data is usually normalized to that range.

Multilayer perceptron architecture (3)

- Each node $O_r, r = 1, \ldots, R$ in the output layer is a function of the linear combination of the hidden nodes $H_q, q = 1, \ldots, Q$

$$
\begin{aligned}
O_r &= g_r\left(w_{r0}^{(2)} + \sum_{q=1}^{Q} w_{rq}^{(2)} H_q\right), \quad r = 1, ..., R, \\
O_r &= g_r\left(\mathbf{w}_r^{(2)^T}\mathbf{H}\right), \quad r = 1, ..., R,
\end{aligned}
$$

  where $O = (O_1, O_2, ..., O_R)$, $w_{r0}^{(2)}$ is the *output layer bias*, and $g_r(O)$ is the function which transforms the output.

- The identity function is typically used.
- The biases represent the intercepts of the model.
- This can be interpreted as adding a node with a constant "1" in the output.

Multilayer perceptron architecture (4)

- In classification is popular as well, however, the softmax function $g_r(O) = \frac{e^{O_r}}{\sum_{k=1}^{K} e^{O_k}}$ is also well-used.
- Since $\sigma(\varphi)$ and $g_r(O)$ can be both the identity function.
- An ANN can be viewed as the non-linear generalization of the linear model.
- The error of the network in every single pattern is

$$E_m(\mathbf{w}) = \sum_{r=1}^{R} \sum_{m=1}^{M} \left(y_{mr} - f_r(\mathbf{x}_m)\right)^2, \tag{1}$$

- For classification we use either squared error or cross-entropy (deviance):

$$E_m(\mathbf{w}) = -\sum_{m=1}^{M} \sum_{r=1}^{R} I(y_{mr} == r) \log f_r(\mathbf{x}_m), \tag{2}$$

Multilayer perceptron architecture (5)

- where $\mathbf{w}$ is the weight vector which contains both $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$, the hidden layer weights vector and the output layer weights vector, respectively.
- Commonly the number of output nodes are one $(R = 1)$ in regression settings.
- While in classification problems R corresponds to the number of classes j, where each node models the probability of the class c_j.
- The weights are real-valued numbers and usually initialized randomly in a small range, for example $[-0.5, 0.5]$.

Backpropagation

- The most widely used technique for updating the weights is the backpropagation algorithm.
- Gradient descent is used in order to update the output layer weights

$$w_{rq}^{(2)\text{new}} = w_{rq}^{(2)\text{old}} - \gamma \frac{\partial E_m(\mathbf{w})}{\partial w_{rq}^{(2)\text{old}}}, \tag{3}$$

- where

$$\frac{\partial E_m(\mathbf{w})}{\partial w_{rq}^{(2)}} = -2(y_{mr} - g_r(w_r^{(2)^T} H))g'(w_r^{(2)^T} H)H_q$$

Backpropagation (2)

- and the hidden layer weights

$$w_{qi}^{(1)^{\text{new}}} = w_{qi}^{(1)^{\text{old}}} - \gamma \frac{\partial E_m(\mathbf{w})}{\partial w_{qi}^{(1)^{\text{old}}}}, \tag{4}$$

- where

$$\frac{\partial E_m(\mathbf{w})}{\partial w_{qi}^{(1)}} = -\sum_{r=1}^{R} 2(y_{mr} - g_r(w_r^{(2)^T} H)) g'(w_r^{(2)^T} H) w_{rq}^{(2)} \sigma' \left(\mathbf{w}_q^{(1)^T} \mathbf{x} \right) x_m^{(i)}$$

- The learning rate γ parameter controls the step size of the gradent descent algorithm. An epoch refers to use the backpropagation rule along the whole training set.

Delta Rule

- Observe that for a single point $(\mathbf{x}_m, y_m)$, we can write

$$\begin{aligned} \frac{\partial E_m(\mathbf{w})}{\partial w_{rq}^{(2)}} &= \delta_{rm} H_{qm} \\ \frac{\partial E_m(\mathbf{w})}{\partial w_{qi}^{(1)}} &= s_{qm} x_m^{(i)} \end{aligned} \tag{5}$$

- Note that both δ_{rm} and s_{qm} are "errors" from the current model at the output and hidden layer neurons respectively.
- Additionally, we can express

$$s_{qm} = \left(\sum_{r=1}^{R} \delta_{rm} w_r^{(2)^T} \right) \sigma' \left(\mathbf{w}_q^{(1)^T} \mathbf{x} \right)$$

Delta Rule

- Equations (3) and (4) are computed by using the delta rule (5) with a two-pass algorithm:
  1. In the forward pass $\hat{f}_r(\mathbf{x}_m) = O_r$ is computed.
  2. In the backward pass δ_{rm} is computed and the output layer nodes are updated to compute s_{qm}
- Sometimes Hyperbolic tangent is used as the activation function, but it can be written as a logistic function: $\tanh(\phi) = 2\sigma(2\phi) - 1$
- When we use cross-entropy and the sof-max function we have

$$\frac{\delta E_m(\mathbf{w})}{\delta O_k} = (p_k - y_k),$$

where $y_r = I(y_m = r)$ and $p_k = P(y = k|x) = \frac{e^{O_k}}{\sum_{k=1}^{K} e^{O_k}}$

Practical considerations

- The error function $E(\mathbf{w})$ is non convex, then, it has many local minima.
- Therefore the final solution obtained is quite dependent on the choice of starting weights.
- One must at least try a number of random starting configurations, and choose the solution giving lowest error.
- Probably a better approach is to use the average predictions over the collection of networks as the final prediction.
- In practice, this type of ANN tend to overfit.
- As we discussed in the first chapter, for a fixed value of γ, stochastic gradient descent may not converge to the optimum.
- progressive decay could be used: At each iteration $s : \eta(s) = \eta_0/(1 + s\eta_d)$, where η_0 is the learning rate, and η_d is the learning rate decay.

Weight decay

- We can use a regularization term

$$E(\mathbf{w}) + \lambda \left(\sum_{rq} {w_{rq}^{(2)}}^2 + \sum_{qi} {w_{qi}^{(1)}}^2 \right)$$

- This method is called weight decay which is analogous to ridge regression used for linear models.
- Larger values of $\lambda > 0$ will tend to shrink the weights toward zero.
- This penalty adds terms $2w_{rq}^{(2)}$ and $2w_{qi}^{(1)}$ to the respective gradient expressions.

Vanishing gradient descent

- If we use more hidden layers, stochastic gradient descent by using backpropagation. We would see that our deep networks not performing much (if at all) better than shallow networks.
- This behavior seems strange. Intuitively, extra hidden layers should make the network able to learn more complex classification functions, and thus do a better job classifying.
- Several authors have reported that the different layers in our deep network are learning at different speeds.
- In particular, when later layers in the network are learning well, early layers often get stuck during training, learning almost nothing at all.
- There are fundamental reasons the learning slowdown occurs, connected to our use of gradient-based learning techniques.

Vanishing gradient descent (2)

- Let's consider the following neural network

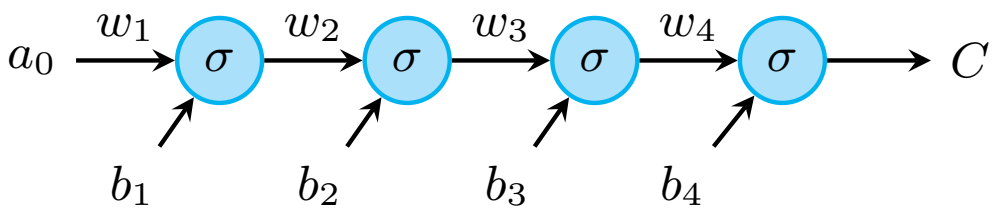


Figure 2: ANN with 4 hidden layers example

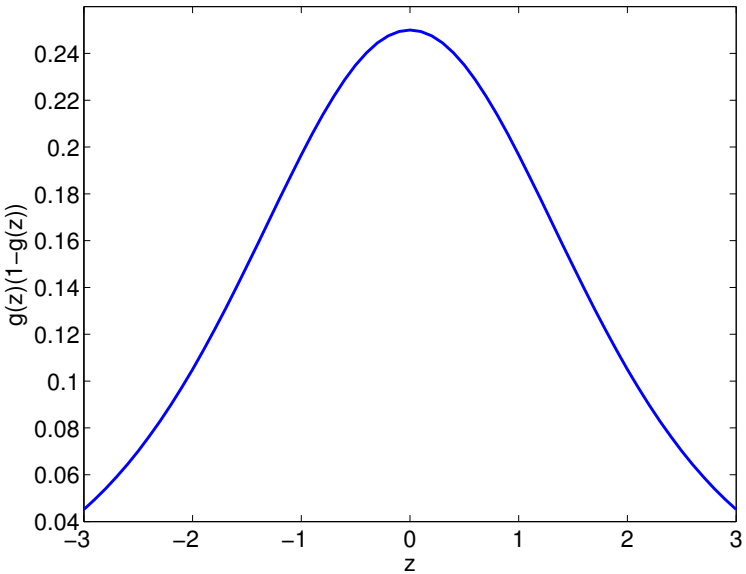
- Here w_i, $i = 1, \ldots, 4$ are the weights of each layer.
- b_i, $i = 1, \ldots, 4$ are the biases of each layer.
- C is the cost function.
- The output a_i from the ith neuron is $\sigma(z_j)$,
- And z_j is the weighted input of the ith neuron.

Vanishing gradient descent (3)

- Recall that $a_4 = \sigma(z_4) = \sigma(w_4\sigma(z_3) + b_4)$
- Calculating the derivative respect to the bias b_1 we have

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4}\sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot w_3 \cdot \sigma'(z_2) \cdot w_2 \cdot \sigma'(z_1) \cdot 1$$

- Except for the first and the last terms, this quantity is a product of terms of the form $\sigma'(z_j)w_j$.
- To study how these terms behave, let's observe the derivative of the sigmoid function



Vanishing gradient descent (4)

- As we saw the $|w_j|$s are commonly set between -0.5 and 0.5.
- Thus $|\sigma'(z_j)w_j| \leq \frac{1}{8}$.
- Note that
$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial b_3}\sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3)$$
- Hence,
$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial b_3} \cdot w_3 \cdot \sigma'(z_2) \cdot w_2 \cdot \sigma'(z_1)$$
- $\frac{\partial C}{\partial b_1}$ will be a factor of 32 (or more) smaller than $\frac{\partial C}{\partial b_3}$.
- However, we could fix this behavior if the terms get greater than 1.
- Instead, the gradient will actually grow exponentially as we move backward through the layers.
- Instead of a vanishing gradient problem, we'll have an exploding gradient problem.

Exploding gradient descent

- There are two steps to get an exploding gradient.
- First, we choose all the weights in the network to be large, say $w_1 = w_2 = w_3 = w_4 = 100$.
- Second, we'll choose the biases so that the $\sigma'(z_j)$ terms are not too small:
- To do this, we will choose the biases to ensure that the weighted input to each neuron is $z_j = 0$ (and so $\sigma'(z_j) = 1/4$).
- So, we want $z_j = w_j a_{j-1} + b_j = 0$. We can achieve this by setting the biases $b_j = -w_j \cdot a_{j-1}$.
- When we do this, we see that all the terms $w_j \sigma'(z_j)$ are equal to $100 \cdot (1/4) = 25$.
- Thus, we got an exploding gradient.

The unstable gradient problem

- Both the vanishing gradient problem or the exploding gradient problem are consequences of the fact that the gradient in early layers is the product of terms from all the later layers.
- When there are many layers, that's an intrinsically unstable situation.
- The only way all layers can learn at close to the same speed is if all those products of terms come close to balancing out.
- Without some mechanism or underlying reason for that balancing to occur, it's highly unlikely to happen simply by chance.
- As a result, if we use standard gradient-based learning techniques, different layers in the network will tend to learn at wildly different speeds.
- To give a deeper understanding of this discussion please refer to
  1. Glorot and Bengio. *Understanding the difficulty of training deep feedforward neural networks*
  2. Sutskever et al. *On the importance of initialization and momentum in deep learning*.

Autoencoders

- The difference with the MLP is that in an autoencoder, the output layer has equally many nodes as the input layer, and instead of training it to predict some target value y given inputs x.
- An autoencoder is trained to reconstruct its own inputs x.
- The cost function is the quadratic reconstruction error: $C = \sum_i (x_i - \hat{x}_i)^2$

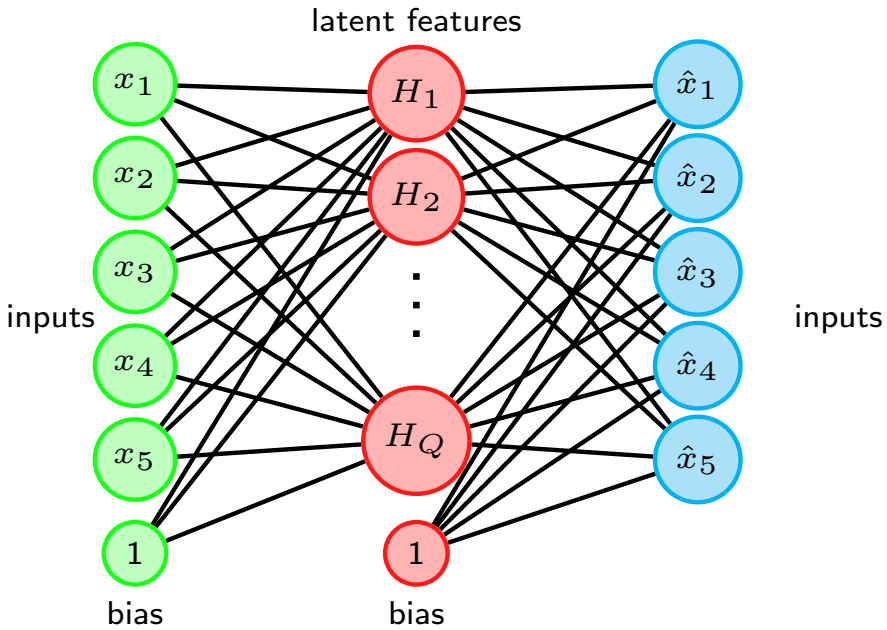


Figure 4: Autoencoder scheme

Extreme Learning Machines

- This ANN is too much faster than the classic MLP feedforward.
- This ANN has one hidden layer, and the weights connecting inputs to hidden nodes are randomly assigned and fixed.
- The weights between hidden nodes and outputs are updated in a single step to minimize the cost function.
- If we fix the hidden layer weights. We can directly compute H the matrix of the outputs of the hidden nodes for each training example

Extreme Learning Machines (2)

- In regression problem, the outputs of the model

$$\hat{\mathbf{Y}} = \sigma(\mathbf{W}^{(1)} X)\beta = H\beta$$

- where β is the output layer weight vector.
- we now simply select the output weights β to minimize the quadratic error: $\hat{\beta} = (H^T H)^{-1} H^T Y$
- While in classification problems, H is defined

$$\mathbf{H} = \begin{bmatrix} h(\mathbf{x}_1) \\ \vdots \\ h(\mathbf{x}_M) \end{bmatrix} = \begin{bmatrix} h_1(\mathbf{x}_1) & \cdots & h_Q(\mathbf{x}_1) \\ \vdots & \cdots & \vdots \\ h_1(\mathbf{x}_M) & \cdots & h_Q(\mathbf{x}_M) \end{bmatrix}$$

- Here, $h_q(\mathbf{x}_m)$ is the output of the hidden node q evaluated with $\mathbf{x}_m$
- And, we select the output weights β to minimize the error norm:

$$||H\beta - T||^2$$

- As we know, its solution is $\hat{\beta} = (H^T H)^{-1} H^T T$

Any questions?