

Appendix

A Artificial neural networks

Artificial neural networks (ANNs) are a class of learning models inspired by the central nervous system, in particular the neurons and synapses of the brain. An ANN comprises many interconnected neurons, and each connection is associated with a weight. Figure 5 shows a model of a single neuron in an ANN. Each neuron works by summing the weighted activations of the incoming connections and an optional bias, then it applies the activation function to the result. Let A be the set of neurons with a connection to a neuron β , then the input function for the neuron β is

$$x_\beta = b + \sum_{\alpha \in A} w_{\alpha\beta} y_\alpha, \quad (1)$$

where x_β is the input to the neuron's activation function, $w_{\alpha\beta}$ is the weight of the connection from neuron α to this neuron β , and b is an optional bias. The bias can also be described as simply another neuron that always returns 1 along with a weight. The output of the neuron β after applying the activation function is denoted y_β , i.e. $y_\beta = f(x_\beta)$.

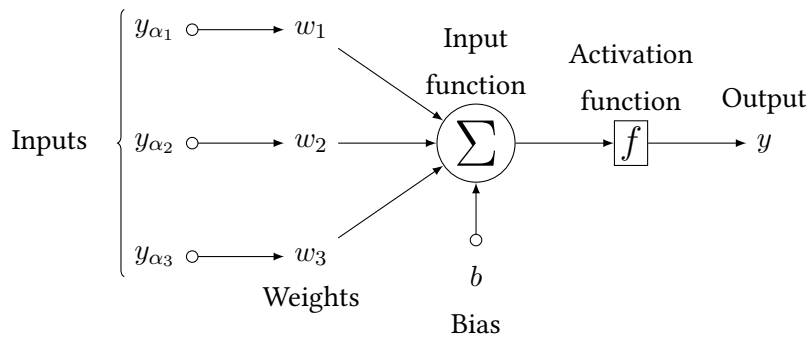


Figure 5: A mathematical model of a neuron

Common activation functions are sigmoid, hyperbolic tangent and softplus. Definition and properties of these activation functions are shown in Table 8. Both sigmoid and hyperbolic tangent *squash* the output to a specific range (e.g. $[0, 1]$ for sigmoid) making them useful for binary classification problems. To train a network using a gradient descent (e.g. backpropagation described in Appendix A.2), the chosen activation function must be differentiable, and the choice may influence the speed and effectiveness of the algorithm.

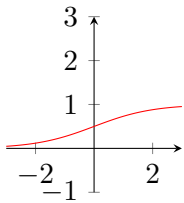
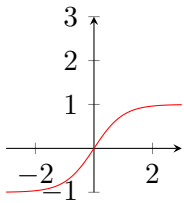
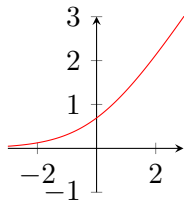
	Sigmoid	Hyperbolic tangent	Softplus
Formula	$\sigma(x) = \frac{1}{1 + e^{-x}}$	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\text{softplus}(x) = \ln(1 + e^x)$
Range	$[0, 1]$	$[-1, 1]$	$[0, \infty[$
Plot			

Table 8: *Typical activation functions*

A.1 Feedforward neural network

There are many ways to connect neurons together, and each network type has its own properties. ANNs can have different layers of neurons. Single-layer networks are connected directly from the input neurons to the output neurons. Multi-layer networks have one or more layers of hidden neurons that are between the input and output layers, see Figure 6. The simplest ANN is the feedforward network. It has connections in one direction like a directed acyclic graph. The information only moves in one direction, from input layer toward the output layer. Finding the optimal network size, including the distribution of neurons in the hidden, input and output layers, is a challenging issue [13, p. 736].

When referring to the neurons of a network we will use i for input neurons, o for output neurons, and h for hidden neurons.

A.2 Backpropagation

ANNs are trained with learning algorithms that calculate the error between calculated output and the sample output data. The error is then used to create an adjustment of the weights. A popular learning algorithm for multi-layer networks is backpropagation [13, p. 733] in conjunction with gradient descent. Backpropagation is a method for

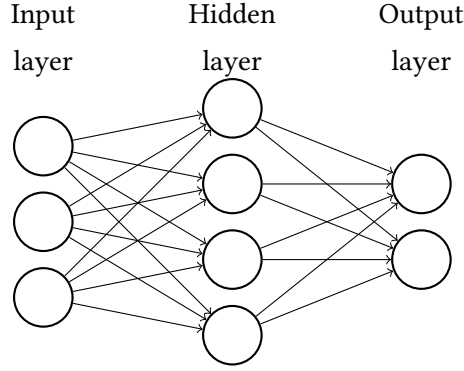


Figure 6: A neural network with hidden neurons

supervised learning that calculates the gradient of an error function with respect to all the weights in the network, in order to update those weights and minimize the error function.

To calculate the total error of the network, the squared error for each output neuron is summed and then divided in half:

$$E = \frac{1}{2} \sum_{o \in O} (y_o - \hat{y}_o)^2,$$

where o is an output neuron in the output layer O , y_o is the output from neuron o , and \hat{y}_o is the target output from neuron o . This is just one type of error function (sum of squared errors (SSE)), and other error functions may be used for different results.

In order to minimize the above error function we need to calculate weight updates as follows:

$$\Delta w_{\alpha\beta} = -\eta \frac{\partial E}{\partial w_{\alpha\beta}},$$

where α and β are neurons, and η is the learning rate, a constant which regulates the rate of change.

To calculate the derivatives, we calculate an error term, δ , for each neuron, starting with the output neurons. The error term is used as follows:

$$\frac{\partial E}{\partial w_{\alpha\beta}} = \delta_\beta y_\alpha,$$

where δ_β is the error term for neuron β , and y_α is the output of neuron α .

For output neurons the error term is calculated by subtracting the target output from the actual output, then multiplying it by the derivative of the neuron activation:

$$\delta_o = f'(x_o)(y_o - \hat{y}_o),$$

where o is a single output neuron, f' is the derivative of the activation function, and x_o is the weighted sum of inputs to neuron o .

We use the error terms of the output layer to backpropagate the error back to the hidden layer:

$$\delta_h = f'(x_h) \sum_{o \in O} \delta_o w_{ho} ,$$

where δ_h is the error term of a hidden neuron h , x_h is the weighted sum of inputs to neuron h , f' is the derivative of the activation function, w_{ho} is the weight of the connection from neuron h to neuron o . The error terms for any additional hidden layers can be calculated in the same way.

To apply the weight changes, they are added to each of the original weights:

$$w'_{\alpha\beta} = w_{\alpha\beta} + \Delta w_{\alpha\beta} .$$

Each training sample in the set of training samples can be used to calculate new weight updates. After all training samples have been used, an *epoch* has passed. Depending on the learning rate, the size of the network, and the number of training samples, many epochs may be necessary in order to properly train the network. It is common to stop the algorithm when the network error is below a certain user-defined threshold, or when no further improvements are made.

A.3 Momentum

A simple extension of the backpropagation algorithm is the use of a momentum term. Let e be the index of an epoch (at which all weights of the network are updated), then

$$\Delta w_{\alpha\beta}^{(e)} = -\eta \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} + \mu \Delta w_{\alpha\beta}^{(e-1)} ,$$

where $\Delta w_{\alpha\beta}^{(e-1)}$ is the previous weight update and μ is the momentum. Using this update we can calculate the weight at the next epoch:

$$w_{\alpha\beta}^{(e+1)} = w_{\alpha\beta}^{(e)} + \Delta w_{\alpha\beta}^{(e)} .$$

A.4 Resilient backpropagation

Resilient backpropagation (Rprop) completely changes how weight updates are made. Instead of having a global learning rate, the algorithm calculates updates independently on each weight based on the sign of the partial derivative [12]. Rprop has been shown to be one of the fastest weight update mechanisms.

First a delta is calculated for each weight based on the previous delta. If the sign of the partial derivative, $\frac{\partial E}{\partial w_{\alpha\beta}}^{(e)}$, is the same as in the previous epoch, the delta is increased. If the sign is different, the delta is decreased.

$$\Delta_{\alpha\beta}^{(e)} = \begin{cases} \min\left(\eta^+ \cdot \Delta_{\alpha\beta}^{(e-1)}, \Delta_{\max}\right) & \text{if } \frac{\partial E}{\partial w_{\alpha\beta}}^{(e-1)} \cdot \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} > 0 \\ \max\left(\eta^- \cdot \Delta_{\alpha\beta}^{(e-1)}, \Delta_{\min}\right) & \text{if } \frac{\partial E}{\partial w_{\alpha\beta}}^{(e-1)} \cdot \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} < 0 \\ \Delta_{\alpha\beta}^{(e-1)} & \text{otherwise} \end{cases},$$

where η^+ is the rate at which the delta is increased, η^- is the rate at which the delta is decreased, and Δ_{\max} , Δ_{\min} are the upper and lower limits of the delta to avoid overflow/underflow problems when implementing the algorithm.

The deltas are used to calculate the weight updates as follows:

$$\Delta w_{\alpha\beta}^{(e)} = \begin{cases} -\Delta_{\alpha\beta}^{(e)} & \text{if } \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} > 0 \\ +\Delta_{\alpha\beta}^{(e)} & \text{if } \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} < 0 \\ 0 & \text{otherwise} \end{cases}$$

Several modifications and variations exist. One such variation is iRprop^+ , which adds weight backtracking [9]. The only difference is that if $\frac{\partial E}{\partial w_{\alpha\beta}}^{(e-1)} \cdot \frac{\partial E}{\partial w_{\alpha\beta}}^{(e)} < 0$ and $E^{(e)} > E^{(e-1)}$ the weight update is calculated as

$$\Delta w_{\alpha\beta}^{(e)} = -\Delta w_{\alpha\beta}^{(e-1)}.$$

In other words, if the entire network error, E , has increased since the last epoch, all contributing weights (where the sign of the gradient has changed) are reverted.

One of the advantages of Rprop and its variations is the elimination of additional training parameters. Although the algorithm uses two factors, η^+ and η^- , they have been shown to be more or less independent of the network and problem domain, and the values $\eta^+ = 1.2$ and $\eta^- = 0.5$ are used throughout the literature. Additionally, the initial deltas and partial derivatives are defined as:

$$\Delta_{\alpha\beta}^{(0)} = \Delta_0 = 0.1 \text{ and } \frac{\partial E}{\partial w_{\alpha\beta}}^{(0)} = 0.$$

B Recurrent neural networks

Another type of ANN is a recurrent neural network (RNN) which feeds its outputs back into its own inputs [13, p. 729]. Thus, the response of the network depends on previous states, which makes the network able to support short-term memory, giving it an advantage when dealing with temporal data. There are many different kinds of RNNs. In what follows we will describe the prominent ones.

B.1 The Elman network

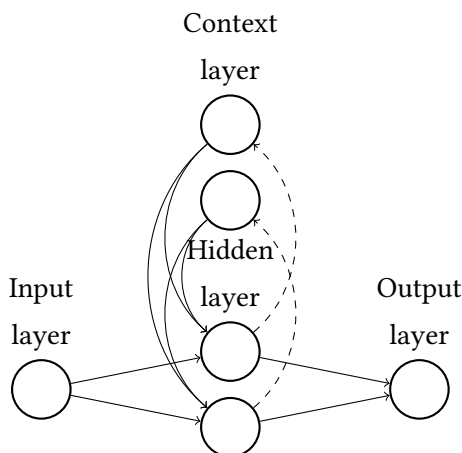


Figure 7: *An Elman network. Each hidden neuron has a recurrent connection to a context neuron, that stores its output. Context neurons influence the next iteration by being connected to all hidden neurons.*

An Elman network, depicted in Figure 7, is a simple type of recurrent network that provides memory in the form of a context layer [3]. Input and context neurons are connected to all hidden units, which are also connected to all output neurons. When an input is propagated as feedforward at each timestep, hidden neurons are activated by both input and context neurons. The hidden neurons then activate a context neuron, saving the hidden neuron value for the next iteration as a sort of state. There is one context neuron for each hidden neuron. The recurrent connections (dashed lines) have a fixed weight of 1.0 and are not adjusted. The output of a hidden neuron h in the set of hidden neurons H is thus calculated as

$$y_h(t) = f(x_h(t)) = f \left(b + \sum_{i \in I} w_{ih} y_i(t) + \sum_{h' \in H} w_{h'h} y_{h'}(t-1) \right).$$

The activation function f is applied to the weighted sum of inputs consisting of two sums and an optional bias. The first sum is over each input neuron i in I (similar to Equation (1)). The second sum is over each hidden neuron h' in H where $w_{h'h}$ is the weight from the context neuron of h' to the hidden neuron h , and $y_{h'}(t-1)$ is the information from the previous time step stored in the context neuron.

B.2 Backpropagation through time

Backpropagation through time (BPTT) is a learning algorithm similar to regular backpropagation as described in Appendix A.2. It can be used to train several types of RNNs,

e.g. the simple Elman network described in the previous section. The algorithm works by unfolding the network structure for a given amount of time steps, essentially giving us a feedforward network to use regular backpropagation on, as illustrated in Figure 8. The recurrent weight is duplicated spatially in the unfolded structure.

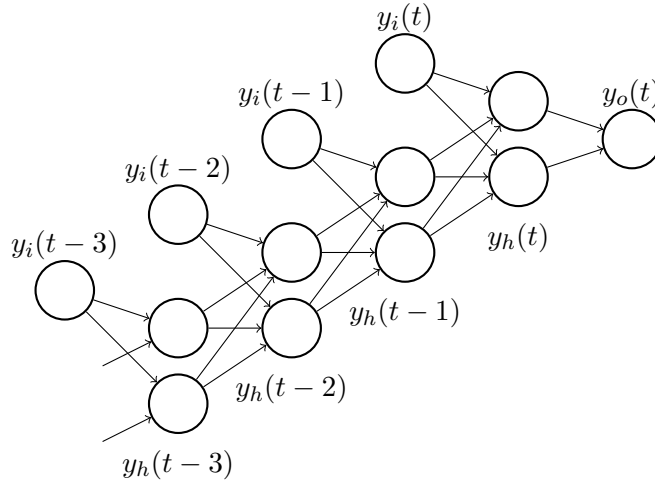


Figure 8: An unfolded recurrent network. y_o is the output of an output neuron. y_h is the output of a hidden neurons and y_i is the output of an input neuron, both unfolded for each time step t .

The algorithm then works like regular backpropagation but proceeding back through time to propagate the error terms for the hidden neurons, which are calculated as

$$\delta_h(t) = f'(x_h(t)) \sum_{h' \in H} \delta_{h'}(t+1) w_{hh'} ,$$

where x_h is the weighted input sum to neuron h , f' is the derivative of the activation function for the output of neuron h , and $w_{hh'}$ is the weight from neuron h to neuron h' . The weight updates can then be calculated using one of the methods described in Appendices A.2 to A.4.

B.3 Long short-term memory

Long short-term memory (LSTM) is an RNN consisting of gated recurrent neural units called *memory cells*.

The original LSTM memory cell introduced in [8] is illustrated in Figure 9. The memory cell receives a number of weighted inputs and calculates a sum $x(t)$ as in Equation (1). The *input squashing* function, f_{is} (e.g. sigmoid or tanh), is applied to the sum. Next it follows a multiplicative unit that multiplies the squashed input $f_{is}(u)$ with the

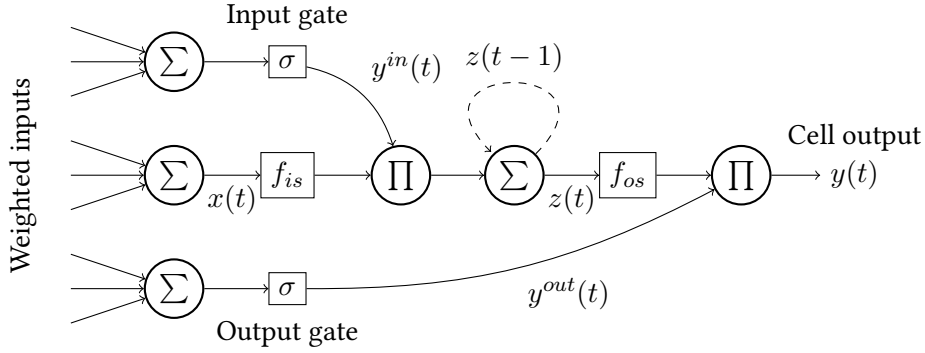


Figure 9: A memory cell with an input gate and an output gate, the dashed connection is recurrent.

activation of an *input gate*, $y^{in}(t)$. The input gate is similar to a regular neuron, and also receives weighted inputs from other units in the network, but the activation function (the *gate squashing* function) should be the sigmoid function (σ). Thus, the gate is fully closed when the output of the gate unit is 0 and fully open when the output is 1. The cell state $z(t)$ of the memory cell is defined as

$$z(0) = 0 ,$$

$$z(t) = z(t - 1) + y^{in}(t) f_{is}(x(t)) ,$$

where t is a discrete time step.

To get the final output of the cell, $y(t)$, the current cell state is squashed using the *output squashing* function f_{os} and gated using another gate, the *output gate*, y^{out} :

$$y(t) = y^{out}(t) f_{os}(z(t)) .$$

Several extensions to the model exists, such as *forget gates* [5]. Figure 10 shows a memory cell with a forget gate, $y^{\varphi}(t)$. The forget gate gates the recurrent connection on the cell state such that the cell may decide to partially forget its earlier states:

$$z(t) = y^{\varphi}(t) z(t - 1) + y^{in}(t) f_{is}(x(t)) .$$

If the forget gate is fully closed (*i.e.* $y^{\varphi}(t) = 0$) the calculation of $z(t)$ does not use the previous cell state, thus making the cell “forget” its previous states entirely.

Several memory cells can share the same input, output, and forget gates. This is called a *memory block*: one input gate, one output gate, one forget gate, and multiple memory cells. A complete LSTM network consists of a number of input neurons, a number of output neurons, and a number of memory blocks each containing a number of memory cells.

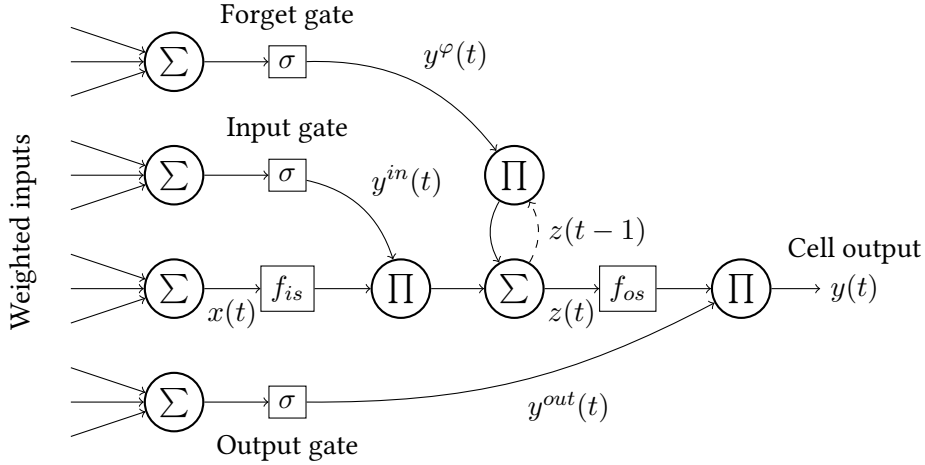


Figure 10: A memory cell with an input gate, an output gate, and a forget gate, the dashed connection is recurrent.

LSTM Learning

To learn the weights of an LSTM block, we can use backpropagation through time as previously described in Appendix B.2. To use backpropagation, we need the error terms for each of the cells in the block as well as each of the gates [6].

Let $c \in C$ be a memory cell in the set of cells in the memory block. The memory cell has state $z_c(t)$ and output $y_c(t)$. The gates $y^{in}(t)$, $y^{out}(t)$, and $y^\varphi(t)$ are shared among all the cells in the block.

To calculate the error terms we first need to calculate a weighted error sum for each cell in the block:

$$\forall c \in C \quad \varepsilon_c(t) = \sum_{\alpha \in A_c} w_{c\alpha} \delta_\alpha(t),$$

where A_c is the set of neurons that receive the output $y_c(t)$ of a memory cell c , and $w_{c\alpha}$ is the weight of the connection from memory cell c to neuron α .

The error sum is then used to calculate the partial derivative of the error function with respect to each cell state as

$$\forall c \in C \quad \frac{\partial E}{\partial z_c}(t) = \varepsilon_c(t) y_c^{out}(t) f'_{os}(z_c(t)) + \frac{\partial E}{\partial z_c}(t+1) y_c^\varphi(t+1).$$

The error term for each cell as a whole (as used to update the weights of each cell's inputs) is

$$\forall c \in C \quad \delta_c(t) = y_c(t) f'_{is}(x_c(t)) \frac{\partial E}{\partial z_c}(t).$$

Finally we can calculate the error terms for the output gate, forget gate, and input

gate as

$$\begin{aligned}\delta^{out}(t) &= f'(x^{out}(t)) \sum_{c \in C} \varepsilon_c(t) f_{os}(z_c(t)) , \\ \delta^\varphi(t) &= f'(x^\varphi(t)) \sum_{c \in C} \frac{\partial E}{\partial z_c}(t) z_c(t-1) , \\ \delta^{in}(t) &= f'(x^{in}(t)) \sum_{c \in C} \frac{\partial E}{\partial z_c}(t) f_{is}(x_c(t)) ,\end{aligned}$$

where x^{out} , x^φ , x^{in} are the weighted sums of inputs to the output gate, forget gate, and input gate, and f' is the derivative of the gate squashing function, *i.e.* sigmoid.

The weight updates for the connections going into the cell and the gates can then be calculated using one of the methods described in Appendices A.2 to A.4.

C Notational conventions

C.1 Neural networks

Let $net = (N, I, O, W)$ be a neural network, then

- N is a set of neuron indices such that $|N|$ is the total number of neurons in the network,
- $I \subset N$ is the indices of the input neurons,
- $O \subset N$ is the indices of the output neurons, and
- W is a weight matrix such that $w_{\alpha\beta}$ is the weight of the connection from a neuron α to a neuron β where $\alpha \in N$ and $\beta \in N$.

H may be used to refer to a set of hidden neurons (*i.e.* $H \not\subseteq I \cup O$), *e.g.* all hidden neurons or a subset.

Let $\alpha \in N$ be the index of a single neuron, then

- x_α is the weighted sum of inputs to the neuron (see Equation (1)),
- y_α is the output of the neuron after applying the activation function, and
- f is the activation function.

C.2 Recurrent neural networks

Let $\alpha \in N$ be the index of a single neuron, and t a discrete time step, then

- $x_\alpha(t)$ is the weighted sum of inputs to the neuron at time t , and
- $y_\alpha(t)$ is the output of the neuron after applying the activation function.

C.3 Long short-term memory

Let $c \in C$ be the index of a memory cell in the set of cells in a memory block, and t a discrete time step, then

- $x_c(t)$ is the weighted sum of inputs to the memory cell c at time t ,
- $y_c(t)$ is the output of the memory cell c ,
- $z_c(t)$ is the cell state of the memory cell c ,
- $y^{in}(t)$ is the output of the input gate,
- $y^{\varphi}(t)$ is the output of the forget gate,
- $y^{out}(t)$ is the output of the output gate,
- f_{is} is the input squashing function,
- f_{os} is the output squashing function, and
- f is the gate squashing function.

C.4 Training

Let $(\mathbf{X}, \hat{\mathbf{Y}})$ be a single training sample, then

- \mathbf{X} is the input vector (of size $|I|$),
- $\hat{\mathbf{Y}}$ is the expected output vector (of size $|O|$), and
- \hat{y}_o is the expected output of an output neuron $o \in O$.

Let e be a training epoch (where $e = 1$ is the first epoch) then

- $E^{(e)}$ is the network error at epoch e , and
- $\Delta w_{\alpha\beta}^{(e)}$ is the weight update for the connection from a neuron $\alpha \in N$ to a neuron $\beta \in N$.