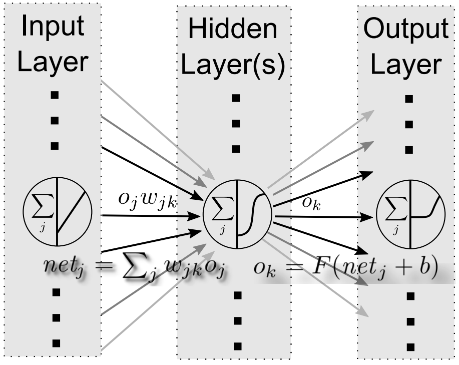
Deep Learning Basics

Eric MSP Veith <eric.veith@offis.de>

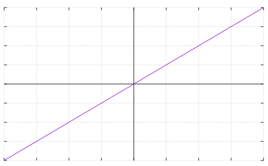
1 Artificial Neural Network Basics



An Artificial Neural Network (ANN) is a directed graph consisting of neurons and connections between them. ANNs are usually subdivided into layers: Input from the outside world is propagates from neurons in the **input layer** via one or more intermediate, so-called **hidden layers**—because they are not visible from the outside—to the output layer, which contains the neurons whose output is the output of the ANN. These networks are also called **feed-forward** networks, because they are constructed as a directed, acyclic graph. Examples of feed-forward networks are the **Perceptron**—which is a general-purpose network structure for pattern matching and to approximate any continuous function—and convolutional neural networks (**ConvNets**), which are used for object detection in images. ANNs can also feed the result of a neuron's activation back to itself, potentially remembering the values between two activations. These ANNs are then called Recurrent Artificial Neural Networks (RNNs), and can in theory approximate any dynamic systems. Examples include the ancestory **Elman Network** and the modern **Long-Short Term Memory** cells and **Gated Recurrent Units**. RNNs excel at time series prediction, e.g., for natural language processing or power output prediction, because through their context layers that save the last run's activation, they incorporate the concept of a series.

The input of each neuron—except for the input layer—is the weighted sum of the results of all other neurons that are connected to it. To this sum, an **activation function** is applied. The activation function for all non-input layer neurons is usually not the identity function, to introduce nonlinearity. For feed-forward networks, the recified linear unit (**ReLU**) is recommended; other possible choices, especially for RNNs, are the **logistic** function and **tanh**.

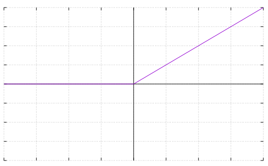
Identity



$$f(x) = x$$

$$f'(x) = 1$$

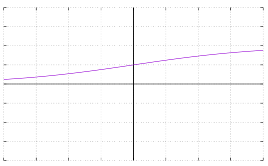
ReLU



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

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

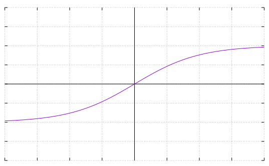
Sigmoid



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

tanh



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

$$\tanh'(x) = 1 - \tanh^2(x)$$

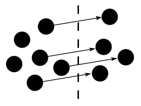
The activation of a layer is the result of a function taking an argument vector; the transition between two layers the product of the previous layer's output vector and the corresponding weight matrix.

$$\mathbf{o}_k = f(\mathbf{x}) \ , \qquad \mathbf{net}_j = \mathbf{o}_j \mathbf{W}_{jk} \ . \tag{1}$$

2 Training

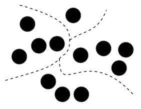
ANNs are approximators with a probability distribution. In order to create their model of the function or system they approximate, every ANN and RNN needs to be trained. Training happens by adjusting the **weights** of the ANN.

Supervised Learning



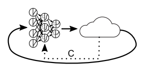
Maps features (inputs) to known labels (teaching outputs): $\mathbf{x} \mapsto \hat{\mathbf{y}}$. Calculates error to adjust weights: $c = f(y, \hat{y})$.

Unsupervised Learning



Clusters a set of inputs without a teaching output/error feedback.

Reinforcement Learning



The ANN acts as an agent upon the world, receives a feedback, and then tries to maximize its internal reward function accordingly by training.

One of the most used algorithms is **backpropagation of error**. It builds on the non-linearity of the activation functions, which need to be differentiable, to make most cost functions **non-convex**. It attributes a portion of the cost to a particular weight. Specifically, it understands the activation of an ANN as a nesting of activation functions and uses the gradient to travel the downward slope of the error function.

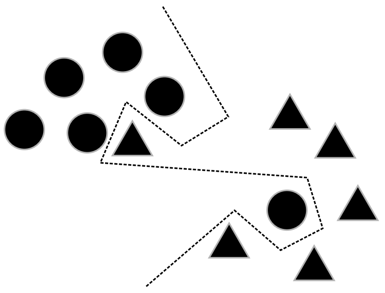
$$\delta_k = \frac{\partial c}{\partial o_k} f'(net_j) \tag{2}$$

A weight is updated stepwise according to the learning rate η and the momentum. The momentum helps the algorithm to escape local minima and be more efficient; in the most current version of backpropagation, **Nadam**, it is adaptive and based on previous and the guessed next step sizes.

Looking at the sigmoid function, we can imagine that the gradient is very small in extreme values. This is called the **vanishing gradient problem** and lets the learning stagnate. By choosing a cost function that penalizes big error values, yet stays out of the extreme regions of an activation function's derivative, by remaining in the $[0;1]$ interval. The **cross-entropy function** is based on the information potential of the ANN for an input x over all n training items:

$$c = -\frac{1}{n} \sum_x \left[y \ln o + (1-y) \ln(1-o) \right] \ . \tag{3}$$

Any ANN's most important ability is to generalize. However, if the **information capacity** of the ANN becomes too big and it is able to learn the training set perfectly, it might overfit. Then, it has perfectly memorized the statistical noise that makes up the training data. To detect this, we subdivide the training patterns in supervised learning into a training and a test set. After training, the ANN is fed the test set, but not to modify its weights: If the training error is very low, but the test error is high, it has overfitted.



Two straightforward approaches to reduce overfitting are increasing the training set, and to reduce the number of neurons/connections (**optimal brain damage**). However, the best way is to regularize the cost function over all weights (**weight decay**), i.e., to scale it according to the information capacity of the ANN:

$$c' = c + \frac{\lambda}{2n} \sum_w w^2 \ . \tag{4}$$

Another option is, instead of permanently reducing the number of neurons/connections, we delete a randomly chosen ($p = 0.5$) subset of them only temporarily for training; this is achieved using a **dropout layer**.

References

Dozat, T. (2016). Incorporating nesterov momentum into adam.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Orr, G. B. and Müller, K.-R. (2003). *Neural networks: tricks of the trade.* Springer.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747.*