Natural Language Processing Introduction

Unit 3: Deep Learning

March 2020

Deep learning

Introduction

Deep learning is an extended field of machine learning that has proven to be highly useful in the domains of text, image, and speech, primarily. The collection of algorithms implemented under deep learning have similarities with the relationship between stimuli and neurons in the human brain.

Why is relevant?

- It is the state of the art for complex problems

Why is relevant?

- It is the state of the art for complex problems
- Can deal with huge amount of data and find complex patterns

Why is relevant?

- It is the state of the art for complex problems
- Can deal with huge amount of data and find complex patterns
- In some tasks can achieve super human performance

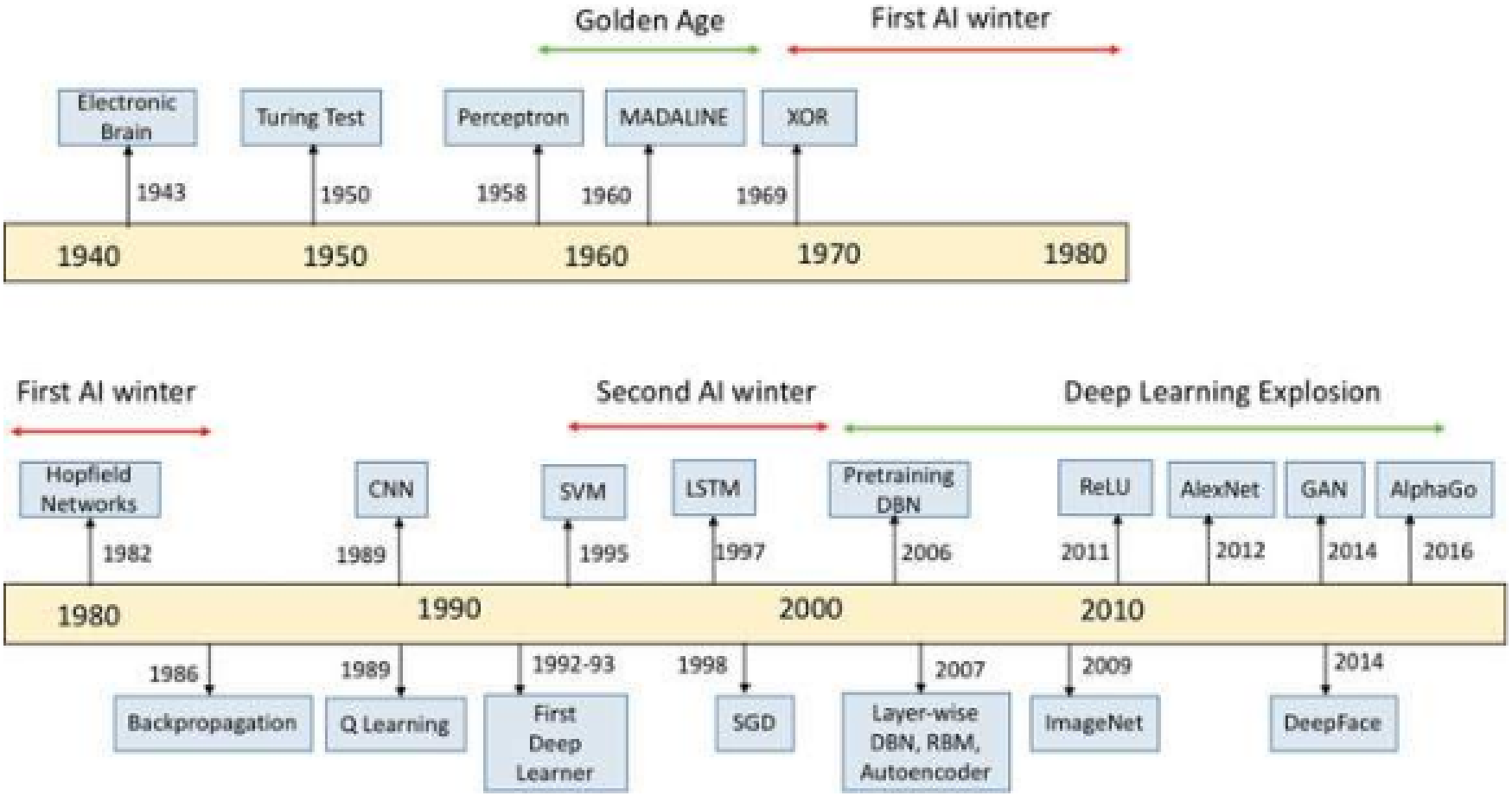
Why is relevant?

- It is the state of the art for complex problems
- Can deal with huge amount of data and find complex patterns
- In some tasks can achieve super human performance
- Can model complex data using sequence modeling.

Why is relevant?

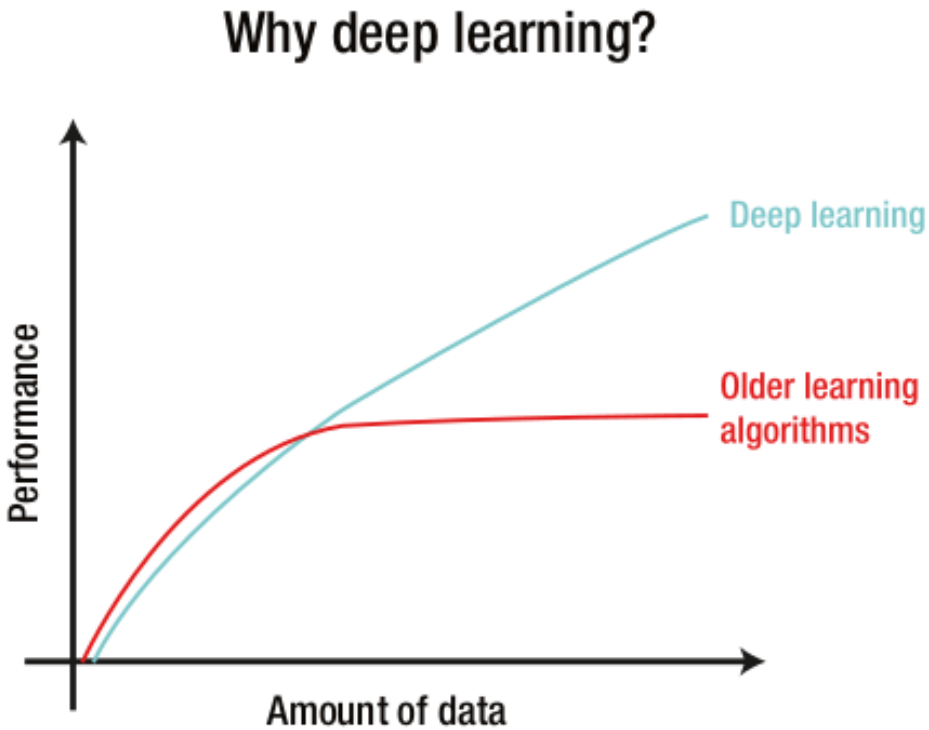
- It is the state of the art for complex problems
- Can deal with huge amount of data and find complex patterns
- In some tasks can achieve super human performance
- Can model complex data using sequence modeling.
- Can perform memory tasks by design adding memory cells to the network.

Deep learning evolution



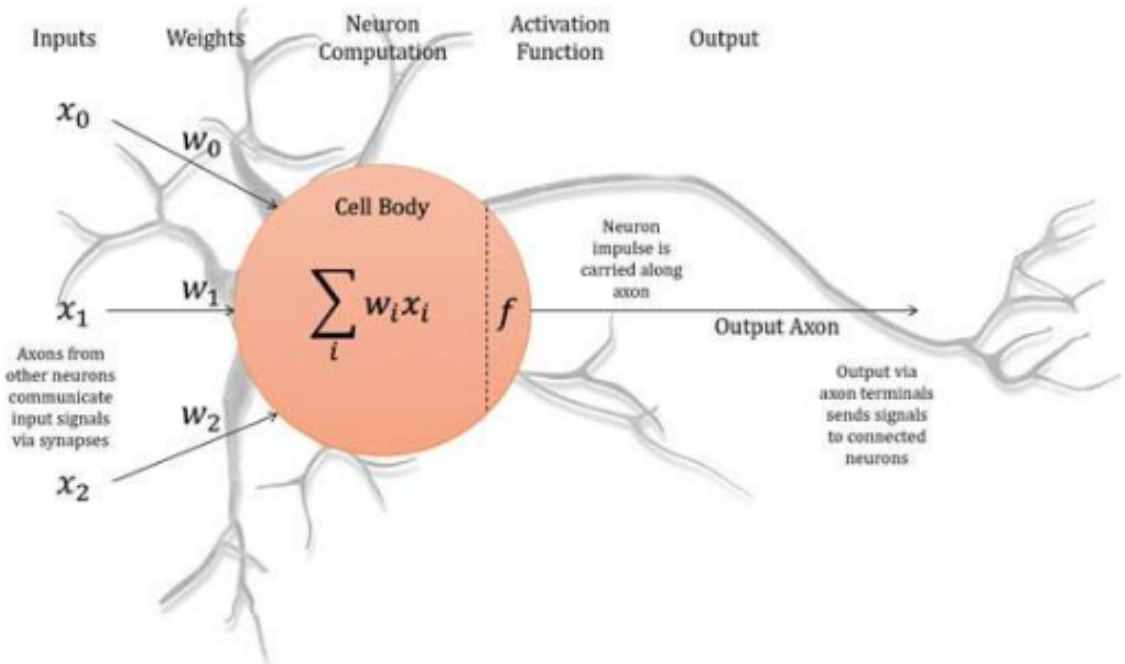
Why deep learning

The term deep in deep learning refers to the depth of the artificial neural network architecture, and learning stands for learning through the artificial neural network itself. The figure below is an accurate representation of the difference between a deep and a shallow network and why the term deep learning gained currency.



Neural networks

Neural networks are networks of artificial neurons gathered to create models where information is passed between them (synapses). Each of these neurons learns a different function of its input, giving the network an extremely diverse representational power.



Deep learning

Definition

Deep learning is a field in Machine Learning which extends concepts in Neural Networks to generate complex models which can't be learned with traditional techniques. Uses a deep (many layers) neural networks architecture and optimization methods in order to achieve great results in image, signal and natural language processing.

Perceptron

Deep learning in its simplest form is an evolution of the perceptron algorithm, trained with a gradient-based optimizer. The perceptron algorithm is one of the earliest supervised learning algorithms, dating back to the 1950s. Much like a biological neuron, the perceptron algorithm acts as an artificial neuron, having multiple inputs, and weights associated with each input, each of which then yields an output.

Perceptron

The basic form of the perceptron algorithm for binary classification is:

$$y(x_1, x_2, x_3, \cdots, x_n) = f(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n)$$

We individually weight each x_i by a learned w_i to map the input $x \in R^n$ to an output value y, where $f(x)$ is defined as the step function shown below:

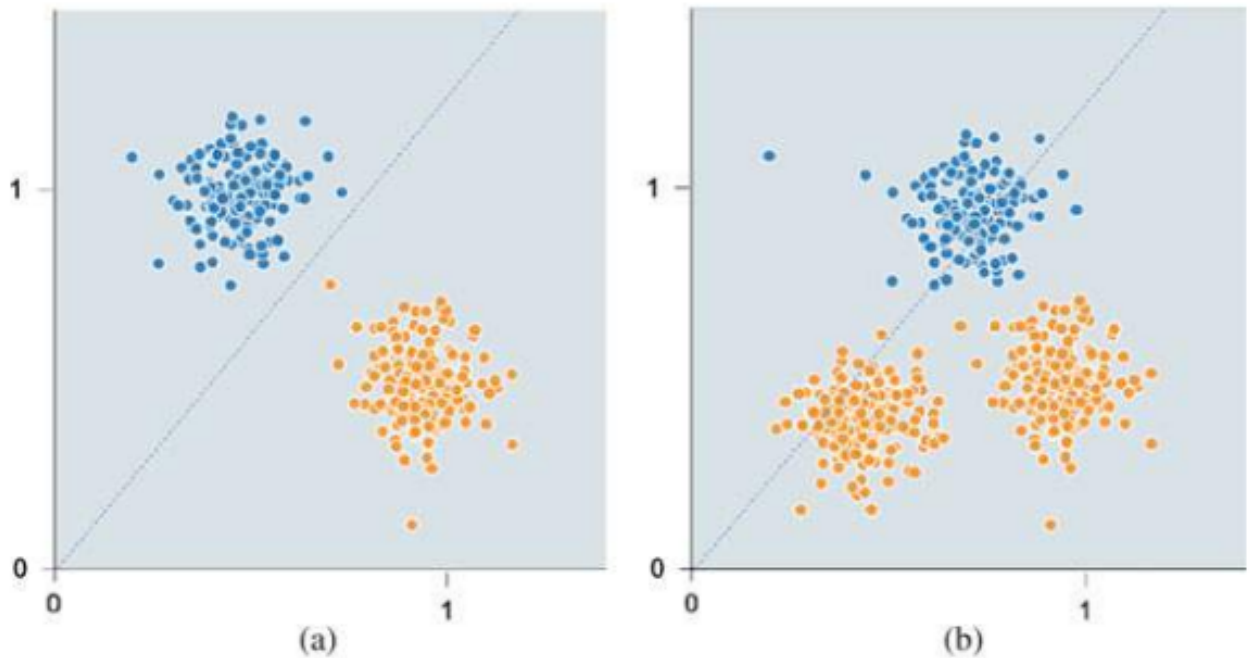
$$f(v) = \begin{cases} 1 \ if \ v \geq 0.5 \\ 0 \ if \ v < 0.5 \end{cases}$$

Bias

The perceptron algorithm learns a hyperplane that separates two classes. However, at this point, the separating hyperplane cannot shift away from the origin. Restricting the hyperplane in this fashion causes issues, when the data is linearly separable, but the perceptron algorithm is not able to separate the data. This is due to the restriction of the separating plane needing to pass through the origin.

Bias

Linearily separable data. Perceptron boundary passing in th origin:

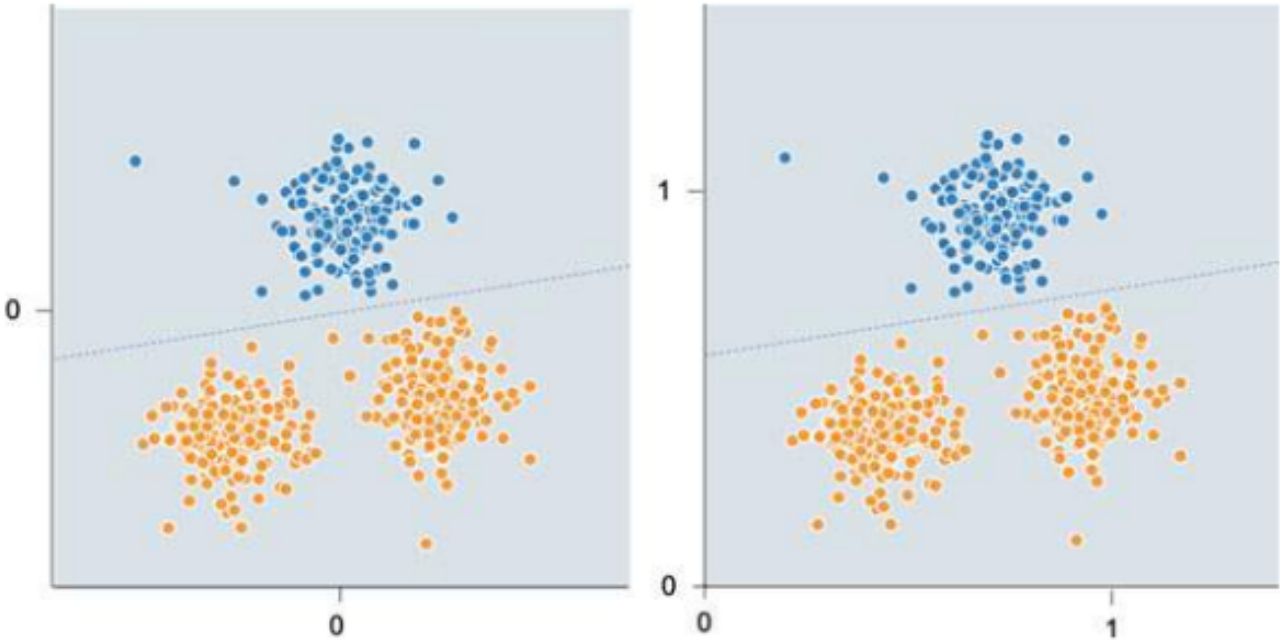


Bias

One solution is to ensure that our data is learnable if we normalize the method to center around the origin as a potential solution to alleviate this issue or add a bias term b to allowing the classification hyperplane to move away from the origin.

Bias

Linearily separable data. Percetron adding a bias term:



Perceptron function

We can write the perceptron with a bias term as:

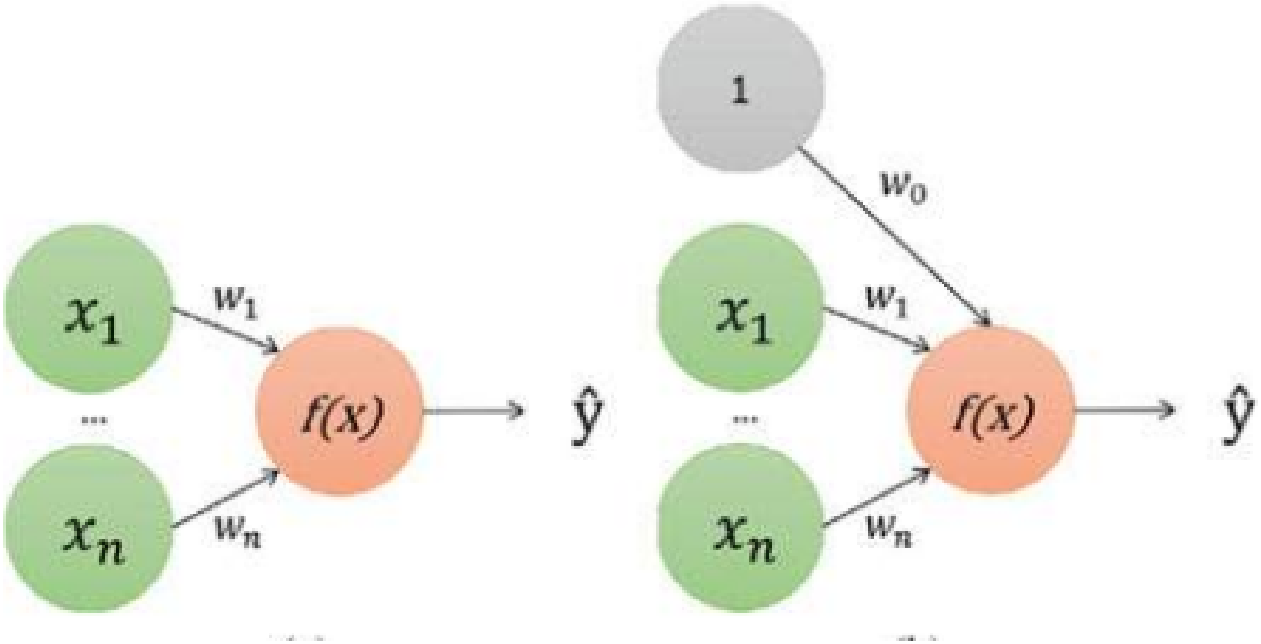
$$y(x_1, x_2, x_3, \cdots, x_n) = f(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b)$$

Alternatively, we can treat b as an additional weight w_0 tied to a constant input of 1 and write it as:

$$y(x_1, x_2, x_3, \cdots, x_n) = f(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + w_0)$$

Bias

Percetron adding a bias term:



Perceptron vector notation

Some authors describe this as adding an input constant $x_0 = 1$, allowing the learned value for $b = w_0$ to move the decision boundary away from the origin. We can write the perceptron function using vector notation as:

$$y(x) = f(wx + b)$$

Perceptron learning

The learning process for the perceptron algorithm is to modify the weights w to achieve a low error on the training set. For example, suppose we need to separate sets of points A and B. Starting with random weights w, we incrementally improve the boundary through each iteration with the aim of achieving $E(w, b) = 0$.

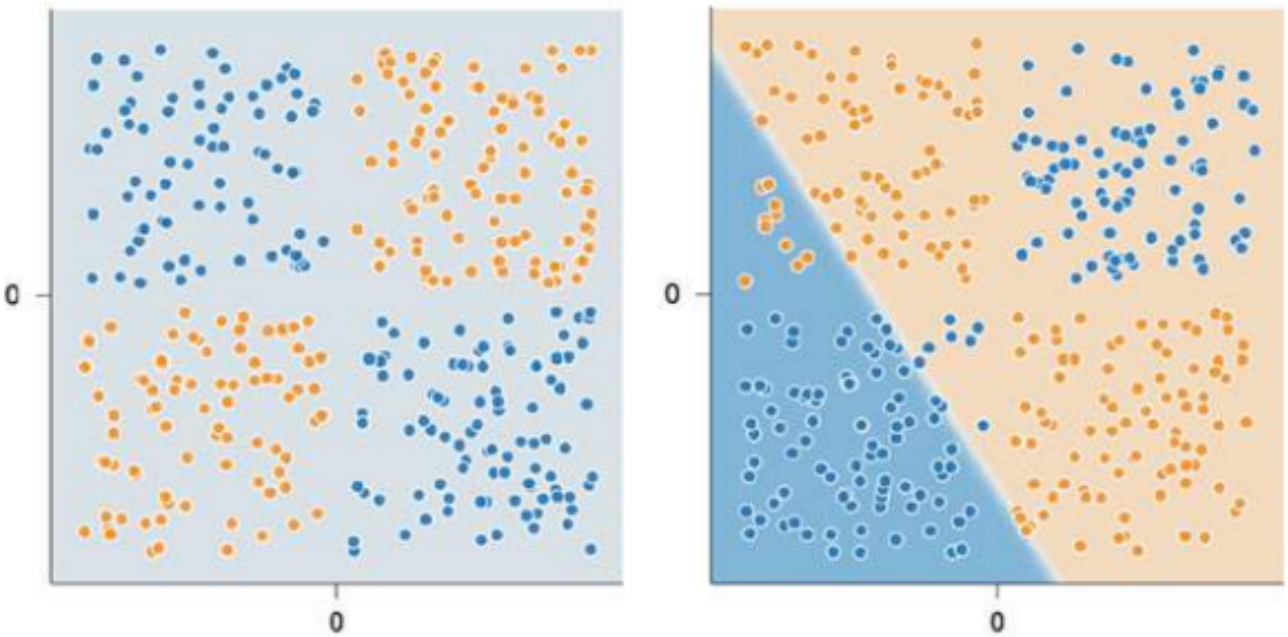
$$E(w, b) = \sum_{x \in A} (1 - f(wx + b)) + \sum_{x \in B} f(wx + b)$$

Linear separability

Two sets of data are linearly separable if a single decision boundary can separate them. For example, two sets, A and B, are linearly separable if, for some decision threshold t, every $x_i \in A$ satisfies the inequality $\sum_i w_i x_i \geq t$ and every $y_j \in B$ satisfies $\sum_j w_j x_j < t$. Conversely, two sets are not linearly separable if separation requires a non linear decision boundary.

Linear separability

Non linear separable data:



Multilayer perceptron

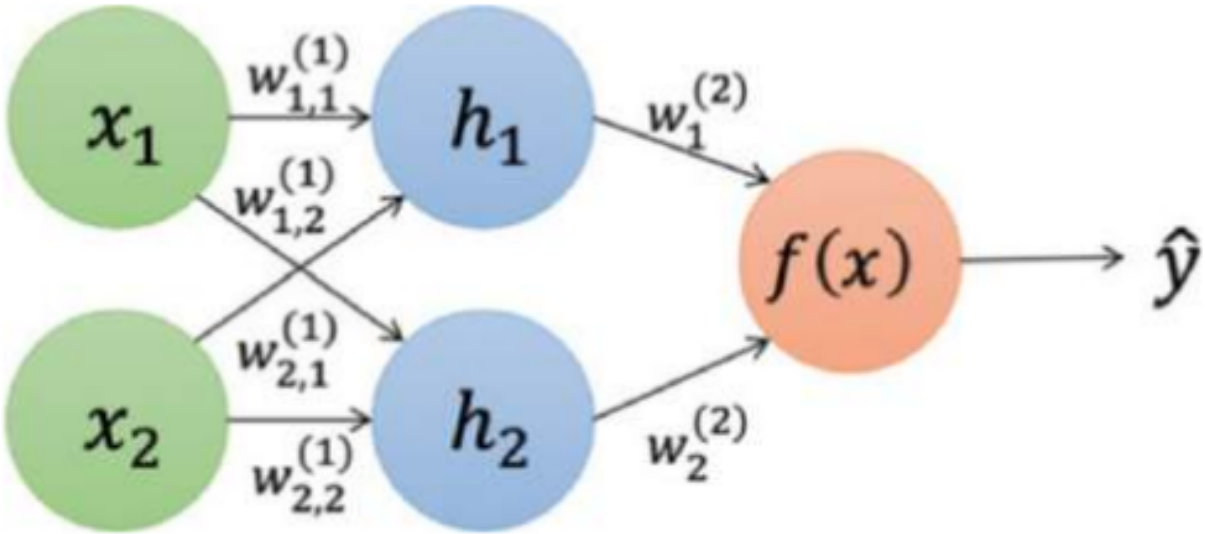
The multilayer perceptron (MLP) links multiple perceptrons (commonly referred to as neurons) together into a network. Neurons that take the same input are grouped into a layer of perceptrons. Instead of using the step function, as seen previously, we substitute a differentiable, non-linear function called **activation function**.

Multilayer perceptron

Activation function or non-linearity, allows the output value to be a non-linear, weighted combination of its inputs, thereby creating non-linear features used by the next layer. In contrast, using a linear function as the activation function restricts the network to only being able to learn linear transforms of the input data. Furthermore, it is shown that any number of layers with a linear activation function can be reduced to a 2-layer MLP

Multilayer perceptron

The MLP is composed of interconnected neurons and is, therefore, a neural network. Specifically, it is a feed-forward neural network, since there is one direction to the flow of data through the network (no cycles—recurrent connections).



Steps for training a neural network

- 1. **Forward propagation**: Compute the network output for an input example.
- 2. **Error computation**: Compute the prediction error between the network prediction and the target.
- 3. **Backpropagation**: Compute the gradients in reverse order with respect to the input and the weights.
- 4. **Parameter update**: Use stochastic gradient descent to update the weights of the network to reduce the error for that example.

Forward propagation

The first step in training an MLP is to compute the output of the network for an example from the dataset. We use the sigmoid function, represented by $\sigma(x)$, as the activation function for the MLP:

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

The forward propagation step is very similar to steps 3 and 4 of the perceptron algorithm. The goal of this process is to compute the current network output for a particular example x, with each output connected as the input to the next layer's neuron(s).

Forward propagation

The layer's weights are combined into a single weight matrix, W_l , representing the collection of weights in that layer, where l is the layer number. The linear transform performed by the layer computation for each weight is an inner product computation between x and W_l . This type is regularly referred to as a "fully connected," "inner product," or "linear" layer because a weight connects each input to each output. Computing the prediction $\hat{y}$ for an example x where h_1 and h_2 represent the respective layer outputs becomes:

$$f(v) = \sigma(v)$$
$$h_1 = f(W_1 x + b_1)$$
$$h_2 = f(W_2 x + b_2)$$
$$\hat{y} = h_2$$

Error computation

The error computation step verifies how well our network performed on the example given. We use mean squared error (MSE) as the loss function used in this example (treating the training as a regression problem). MSE is defined as:

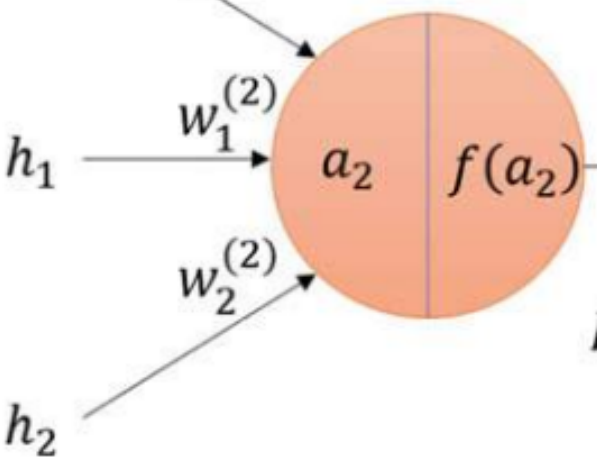
$$E(\hat{y}, y) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Error computation

This error function is commonly used for regression problems. The squaring function forces the error to be non-negative and functions as a quadratic loss with the values closer to zero, yielding a polynomially smaller error than values further from zero.

$$a_2 = w_1^{(2)} h_1 + w_2^{(2)} h_2 + b^{(2)}$$

$$\hat{y} = f(a_2) = \sigma(a_2) = \frac{1}{1 - e^{-a_2}}$$



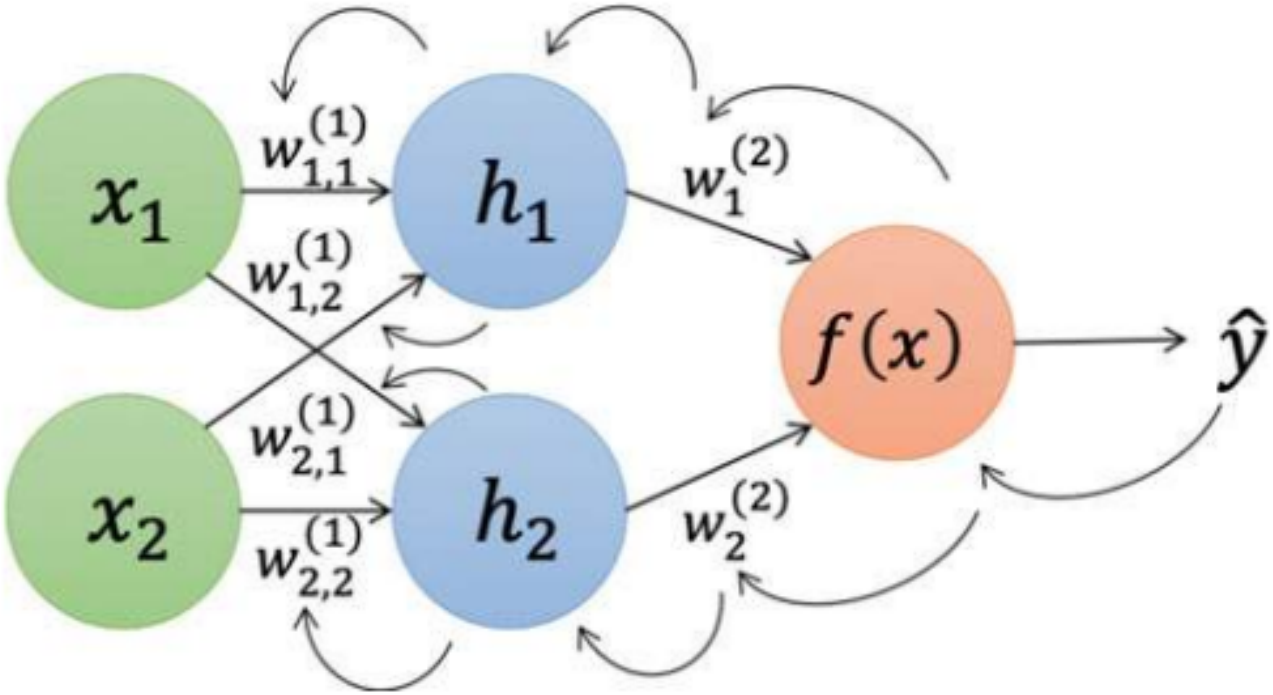
$$E_{out}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Backpropagation

During forward propagation, an output prediction $\hat{y}$ is computed for the input x and the network parameters θ. To improve our prediction, we can use SGD to decrease the error of the whole network.

Backpropagation

Determining the error for each of the parameters can be done via the chain rule of calculus. We can use the chain rule of calculus to compute the derivatives of each layer (and operation) in the reverse order of forward propagation.



Backpropagation

In our previous example, the prediction $\hat{y}$ was dependent on W_2 . We can compute the prediction error with respect to W_2 , by using the chain rule:

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_2}$$

The chain rule allows us to compute the gradient of the error for each of the learnable parameters θ, allowing us to update the network using stochastic gradient descent.

Backpropagation

We begin by computing the gradient on the output layer with respect to the prediction.

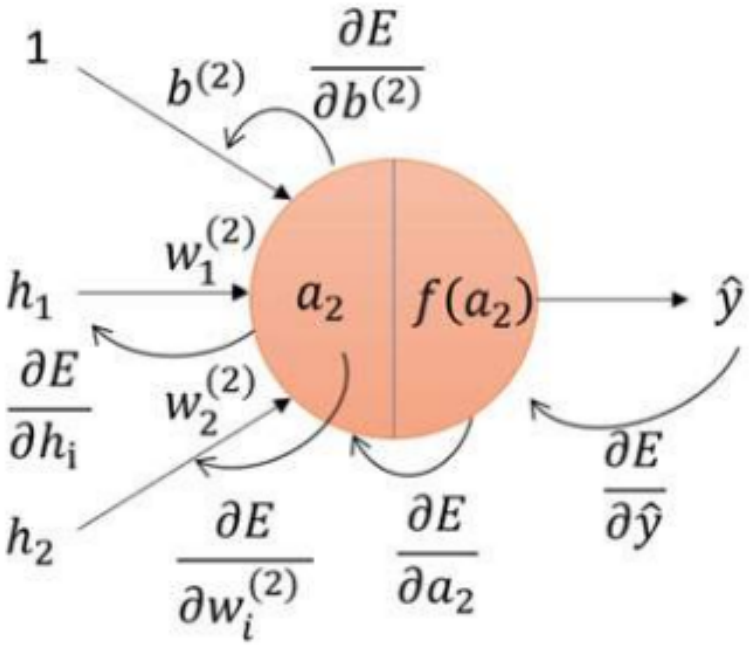
$$\nabla \hat{y} E(\hat{y}, y) = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

We can then compute error with respect to the layer 2 parameters. We currently have the "post-activation" gradient, so we need to compute the preactivation gradient:

$$\nabla a_2 E = \frac{\partial E}{\partial a_2} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_2}$$

Backpropagation

Backpropagation process:



$$\nabla_{\hat{y}} E = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

$$\frac{\partial E}{\partial a_2} = \frac{\partial E}{\partial \hat{y}} f'(a_2)$$

$$\frac{\partial E}{\partial w_i^{(2)}} = \frac{\partial E}{\partial a_2} h_i$$

$$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial a_2}$$

$$\frac{\partial E}{\partial h_i} = \frac{\partial E}{\partial a_2} w_i^{(2)}$$

Parameter update

The last step in the training process is the parameter update. After obtaining the gradients with respect to all learnable parameters in the network, we can complete a single SGD step, updating the parameters for each layer according to the learning rate α:

$$\theta = \theta - \alpha \nabla_\theta E$$

The value of α is particularly vital in SGD and affects the speed of convergence. Too small of a learning rate and the network converges very slowly and can potentially get stuck in local minima near the random weight initialization. Too larges learning rates could make the weights may grow too quickly, becoming unstable and failing to converge at all.

Multi layer perceptron algorithm

Algorithm 1: Neural network training

Data: Training Dataset $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_1, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y_n})\}$
Neural network with l layers with learnable parameters
$\theta = (\{\mathbf{W}_1, \ldots \mathbf{W}_l\}, \{\mathbf{b}_1, \ldots \mathbf{b}_l\})$
Activation function $f(\mathbf{v})$
Learning rate α
Error function $E(\hat{\mathbf{v}}, \mathbf{v})$
Initialize neural network parameters $\theta = (\{\mathbf{W}_1, \ldots \mathbf{W}_l\}, \{\mathbf{b}_1, \ldots \mathbf{b_l}\})$
for $e \leftarrow 1$ *to* e *epochs* **do**
  for $(\mathbf{x}, \mathbf{y})$ *in* $\mathcal{D}$ **do**
    for $i \leftarrow 1$ *to* l **do**
      if i=l **then**
        $\mathbf{h}_{i-1} = \mathbf{x}$
      $\mathbf{a}_i = \mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i$
      $\mathbf{h}_i = f(\mathbf{a}_i)$
    $\hat{\mathbf{y}} = \mathbf{h}_l$
    $error = E(\hat{\mathbf{y}}, \mathbf{y})$
    $\mathbf{g}_{\mathbf{h}_{i+1}} = \nabla_{\hat{\mathbf{y}}} E(\hat{\mathbf{y}}, \mathbf{y})$
    for $i \leftarrow l$ *to* 1 **do**
      $\mathbf{g}_{\mathbf{a}_i} = \nabla_{\mathbf{a}_i} E = \mathbf{g}_{\mathbf{h}_{i+1}} \circ f'(\mathbf{a}_i)$
      $\nabla_{\mathbf{W}_i} E = \mathbf{g}_{\mathbf{a}_i} \mathbf{h}_{i-1}^{\mathsf{T}}$
      $\nabla_{\mathbf{b}_i} E = \mathbf{g}_{\mathbf{a}_i}$
      $\mathbf{g}_{\mathbf{h}_i} = \nabla_{\mathbf{h}_{i-1}} E = \mathbf{W}_i^{\mathsf{T}} \mathbf{g}_{\mathbf{a}_i}$
    $\theta = \theta - \alpha \nabla_\theta E$

Deep learning

The term “deep learning” is somewhat ambiguous. In many circles deep learning is a re-branding term for neural networks or is used to refer to neural networks with many consecutive (deep) layers. However, the number of layers to distinguish a deep network from a shallow network is relative.

Deep learning

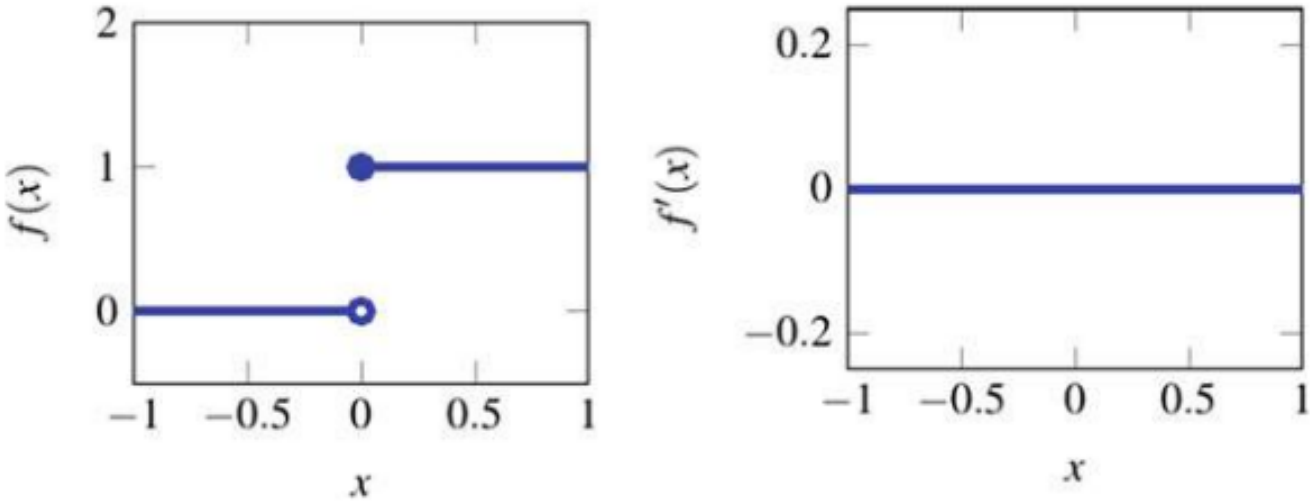
In general, deep networks are still neural networks (trained with backpropagation, learning hierarchical abstractions of the input, optimized using gradient-based learning), but typically with more layers. The distinguishing characteristic of deep learning is its application to problems previously infeasible to traditional methods and smaller neural networks, such as the MLP.

Deep learning

Deeper networks allow for more layers of hierarchical abstractions to be learned for the input data, thus becoming capable of learning higher-order functions in more complex domains.

Activation functions

When computing the gradient of the output layer, it becomes apparent that the step function is not exactly helpful when trying to compute a gradient. The derivative is 0 everywhere which means any gradient descent is useless. Therefore we wish to use a non-linear activation function that provides a meaningful derivative in the backpropagation process.

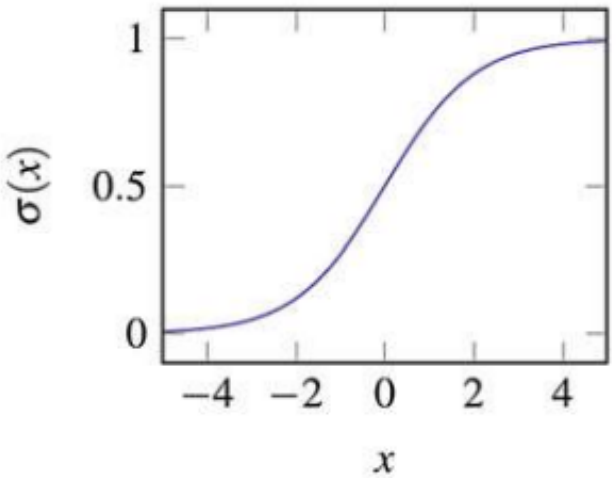


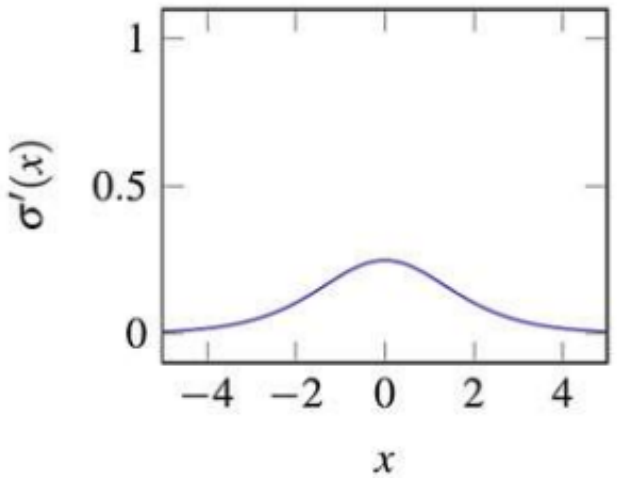
Activation functions: sigmoid

The sigmoid function is a useful activation for a variety of reasons. This function acts as a continuous squashing function that bounds its output in the range (0, 1).

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

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





Activation functions: sigmoid

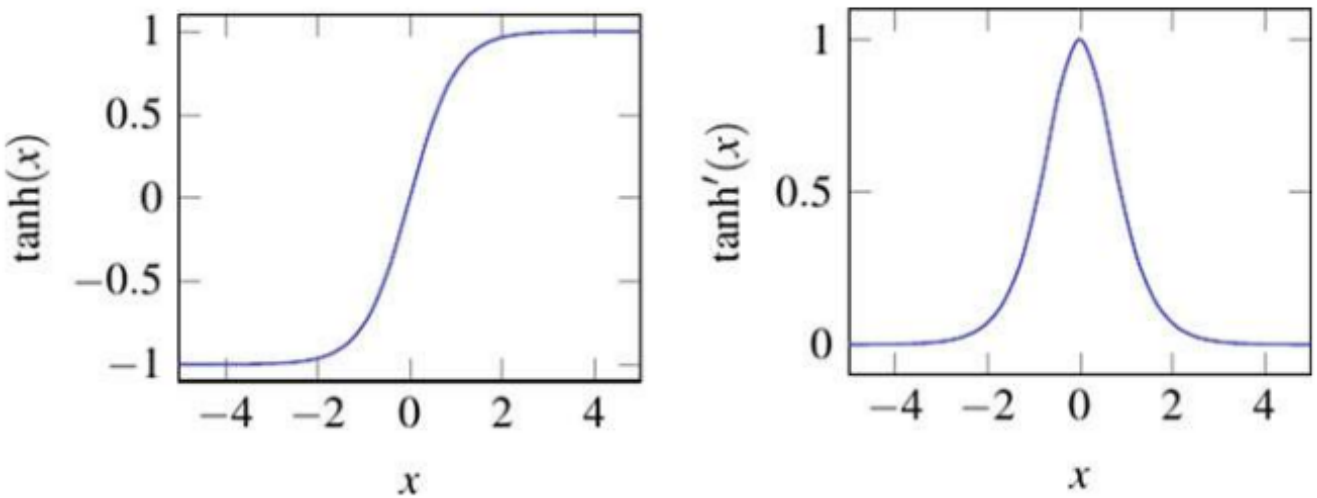
There are, however, some undesirable properties of the sigmoid function:

- Saturation of the sigmoid gradients at the ends of the curve (very close to $\sigma(x) \leftarrow 0$ or $\sigma(x) \leftarrow 1$) will cause the gradients to be very close to 0. As backpropagation continues the small gradient is multiplied by the postactivation output forcing it smaller still. Preventing this can require careful initialization of the network weights or other regularization strategies.
- The outputs of the sigmoid are not centered around 0, but instead around 0.5. This introduces a discrepancy between the layers because the outputs are not in a consistent range. This is often referred to as "internal covariate shift".

Activation functions: tanh

The tanh function is another common activation function. It also acts as a squashing function, bounding its output in the range $(-1, 1)$. It can also be viewed as a scaled and shifted sigmoid:

$$tanh(x) = 2\sigma(2x) - 1$$



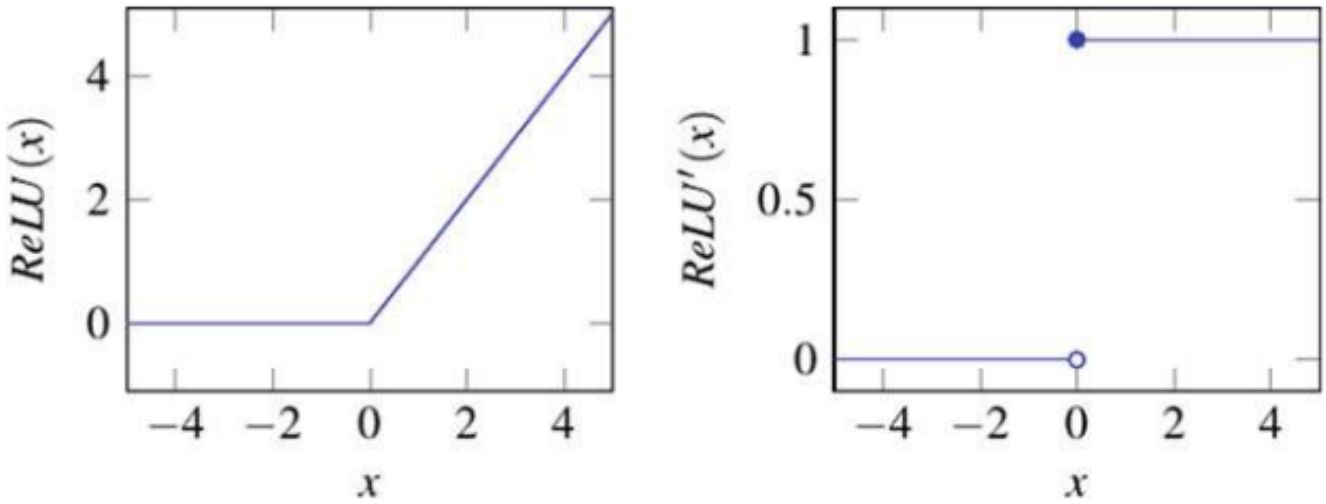
Activation functions: tanh

The tanh function solves one of the issues with the sigmoid non-linearity because it is zero centered. However, we still have the same issue with the gradient saturation at the extremes of the function.

Activation functions: ReLU

The rectified linear unit (ReLU) is a simple, fast activation function typically found in computer vision. The function is a linear threshold, defined as:

$$f(x) = max(0, x)$$



Activations functions: ReLU

This simple function has become popular because it has shown faster convergence compared to sigmoid and tanh, possibly due to its non-saturating gradient in the positive direction. Also the ReLU function is much faster computationally. The sigmoid and tanh functions require exponentials which take much longer than a simple max operation.

Activations functions: ReLU

One drawback from the simplicity of the gradient updates being 0 or 1 is that it can lead to neurons "dying" during training. If a large gradient is backpropagated through a neuron. Some studies have shown that as many as 40 % of the neurons in a network can "die" with the ReLU activation function if the learning rate is set too high.

Activation functions: other activation functions

Other activation functions have been incorporated to limit negative effects previously described:

- Hard tanh: computationally cheaper than the tanh. It has the disadvantage of gradient saturation at the extremes.
  $f(x) = max(-1, min(1, x))$

Activation functions: other activation functions

Other activation functions have been incorporated to limit negative effects previously described:

- Leaky ReLU: introduces an α parameter that allows small gradients to be backpropagated when the activation is not active, thus eliminating the "death" of neurons during training:

$$f(x) = \begin{cases} x \ if \ x \geq 0 \\ \alpha x \ if \ x < 0 \end{cases}$$

Activation functions: other activation functions

Other activation functions have been incorporated to limit negative effects previously described:

- PRELU: uses an α parameter to scale the slope of the negative portion of the input; however, an α parameter is learned for each neuron (doubling the number of learned weights). When $\alpha = 0$ this is the ReLU function and when the α is fixed, it is equivalent to the Leaky ReLU:

$$f(x) = \begin{cases} x \ if \ x \geq 0 \\ \alpha x \ if \ x < 0 \end{cases}$$

Activation functions: other activation functions

Other activation functions have been incorporated to limit negative effects previously described:

- ELU: Modification of the ReLU that allows the mean of activations to push closer to 0, which therefore potentially speeds up convergence:

$$f(x) = \begin{cases} x \ if \ x > 0 \\ \alpha(e^x - 1) \ if \ x \leq 0 \end{cases}$$

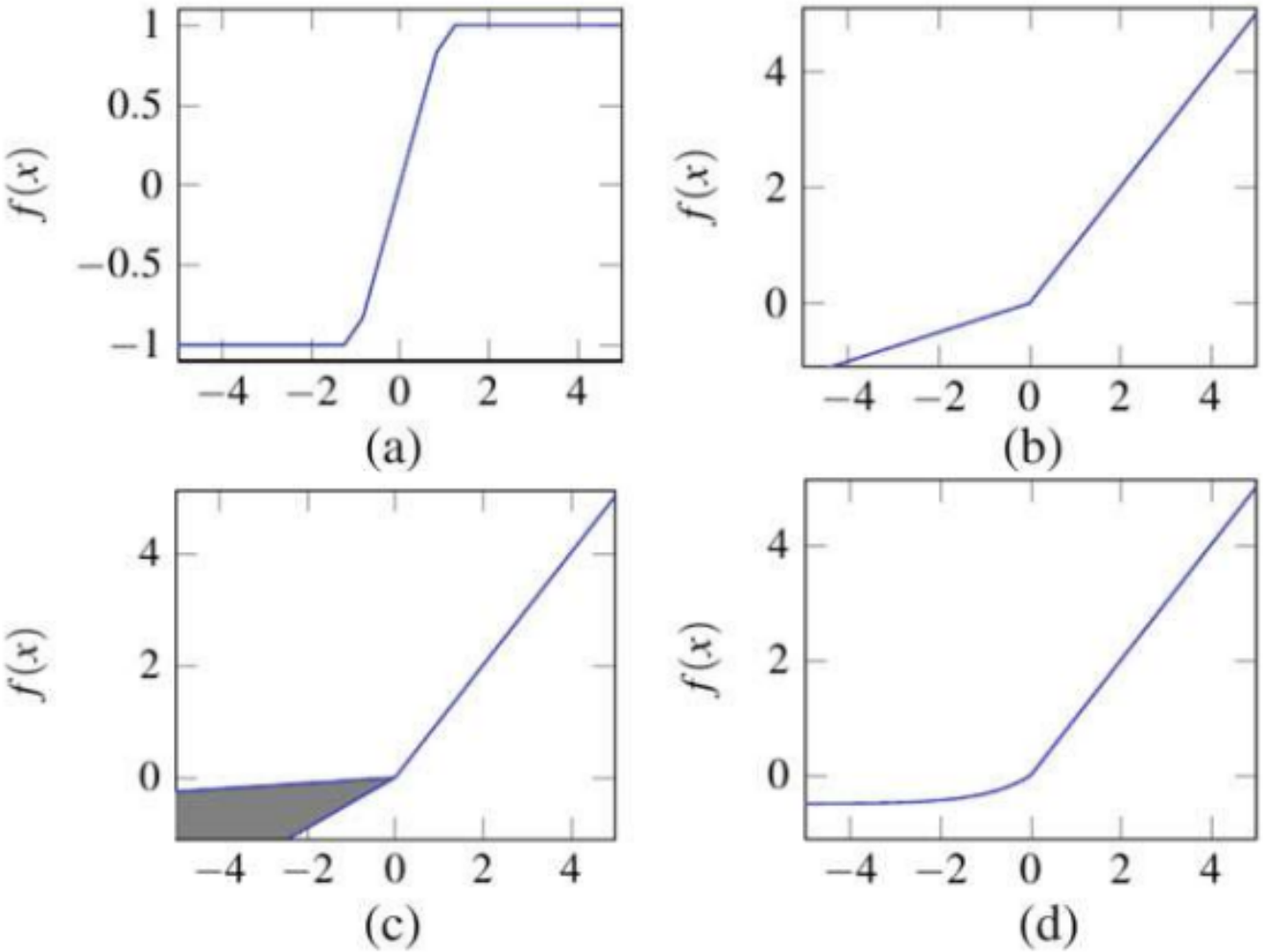
Activation functions: other activation functions

Other activation functions have been incorporated to limit negative effects previously described:

- maxout: The maxout function takes a different approach to activation functions. It differs from the element-wise application of a function to each neuron output. Instead, it learns two weight matrices and takes the highest output for each element:
  $f(x) = max(w_1 x_1 + b_1, w_2 x_2 + b_2)$

Activation functions: other activation functions

Hard tanh (a), leaky ReLU (b), PRELU (c) and ELU (d) functions:



Softmax

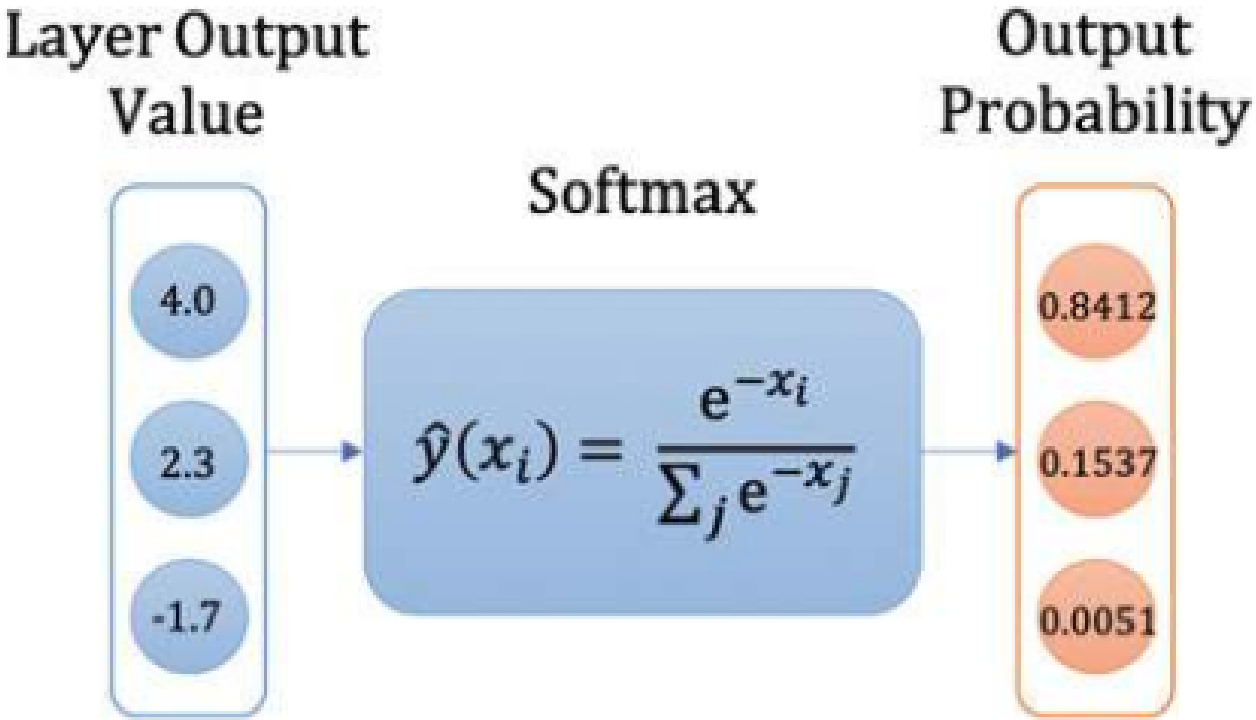
The squashing concept of the sigmoid function is extended to multiple classes by way of the softmax function. The softmax function allows us to output a categorical probability distribution over K classes.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

We can use the softmax to produce a vector of probabilities according to the output of that neuron. In the case of a classification problem that has $K = 3$ classes, the final layer of our network will be a fully connected layer with an output of three neurons.

Softmax

The softmax probabilities can become very small, especially when there are many classes and the predictions become more confident. Most of the time a log based softmax function is used to avoid underflow errors.



Mean square error

Mean squared error(MSE) computes the squared error between the classification prediction and the target. Training with it minimizes the difference in magnitude. One drawback to MSE is that it is susceptible to outliers since the difference is squared.

$$E(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Mean abosulute error

Mean absolute error gives a measure of the absolute difference between the target value and prediction. Using it minimizes the magnitude of the error without considering direction, making it less sensitive to outliers.

$$E(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Negative log likelihood

Negative log likelihood (NLL), is the most common loss function used for multiclass classification problems. It is also known as the multiclass cross-entropy loss. The softmax provides a probability distribution over the output classes. The entropy computation is a weighted-average log probability over the possible events or classifications in a multiclass classification problem. This causes the loss to increase as the probability distribution of the prediction diverges from the target label.

$$E(\hat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n}(y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i))$$

Hinge loss

The hinge loss is a max-margin loss classification taken from the SVM loss. It attempts to separate data points between classes by maximizing the margin between them. Although it is not differentiable, it is convex, which makes it useful to work with as a loss function.

$$E(\hat{y}, y) = \sum_{i=1}^{n} max(0, 1 - y_i \hat{y}_i)$$

Kullback–Leibler (KL) Loss

Additionally, we can optimize on functions, such as the KL-divergence, which measures a distance metric in a continuous space. This is useful for problems like generative networks with continuous output distributions. The KL-divergence error can be described by:

$$E(\hat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n} D_{KL}(y_i||\hat{y}_i)$$

$$= \frac{1}{n}\sum_{i=1}^{n}(y_i \cdot log(y_i)) - \frac{1}{n}\sum_{i=1}^{n}(y_i \cdot log(\hat{y}_i))$$

Optimization methods

The training process of neural networks is based on gradient descent methods, specifically SGD. However, as we have seen in the previous section, SGD can cause many undesirable difficulties during the training process. We will explore additional optimization methods in addition to SGD and the benefits associated with them. We consider all learnable parameters including weights and biases as θ.

Stochastic gradient descent

Stochastic gradient descent is the process of making updates to a set of weights in the direction of the gradient to reduce the error:

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta E$$

Momentum

One issue that commonly arises with SGD is that there are areas of feature space that have long shallow ravines, leading up to the minima. SGD will oscillate back and forth across the ravine because the gradient will point down the steepest gradient on one of the sides rather than in the direction of the minima. Thus, SGD can yield slow convergence. Momentum is one modification of SGD to move the objective more quickly to the minima. The parameter update equation for momentum is:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta E$$

$$\theta^{t+1} = \theta^t - v_t$$

Adagrad

Adagrad is an adaptive gradient-based optimization method. It adapts the learning rate to each of the parameters in the network, making more substantial updates to infrequent parameters, and smaller updates to frequent ones. This makes it particularly useful for learning problems with sparse data. Removes the need to tune the learning rate manually. This does, however, come at the cost of having an additional parameter for every parameter in the network. Adagard equation is:

$$g_{t,i} = \nabla_\theta E(\theta_{t,i})$$

$$\theta^{t+1,i} = \theta^{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \circ g_{t,i}$$

Adagrad

g_t is the gradient at time t along each component of θ, G_t is the diagonal matrix of the sum of up to t time steps of past gradients w.r.t. to all parameters θ on the diagonal, η is the general learning rate, and ϵ is a smoothing term (usually $1e-8$) that keeps the equation from dividing by zero.

Adagrad

The main drawback to adagrad is that the accumulation of the squared gradients is positive, causing the sum to grow, shrinking the learning rate, and stopping the model from further learning. Additional variants, such as Adadelta, have been introduced to alleviate this problem.

RMS prop

RMS-prop was developed by Geoffrey Hinton and introduced to solve the inadequacies of adagrad. It also divides the learning rate by an average of squared gradients, but it also decays this quantity exponentially.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2$$

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

where $\rho = 0.9$ and the learning rate $\eta = 0.001$ is suggested.

ADAM

Adaptive moment estimation, referred to as Adam is another adaptive optimization method. It too computes learning rates for each parameter, but in addition to keeping an exponentially decaying average of the previous squared gradients, similar to momentum, it also incorporates an average of past gradients m_t:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

ADAM

Empirical results show that Adam works well in practice in comparison with other gradient-based optimization techniques. While Adam has been a popular technique, some criticisms of the original proof have surfaced showing convergence to sub-optimal minima in some situations. Each work proposes a solution to the issue, however the subsequent methods remain less popular than the original Adam technique.

Let's code

References

[1] Jurafsky D., Martin J.: Speech and Language Processing 2nd. ed. (2009).

[2] Bird S., Klein E., Loper, E.: Natural Language Processing with Python, (2009). ISBN: 978-0-596-51649-9

[3] Indurkhya N., Damerau F. Handbook of Natural Language Processing, Second Edition (2010). ISBN: 978-1-4200-8593-8

[4] Kao, A., Poteet S. Natural Language Processing and Text Mining (2007). ISBN: 78-1-84628-175-4

[5] Sipser, M. Introduction to the theory of computation (2013). ISBN: 978-1-133-18779-0

[6] https://www.sketchengine.eu/corpora-and-languages/corpus-types/