

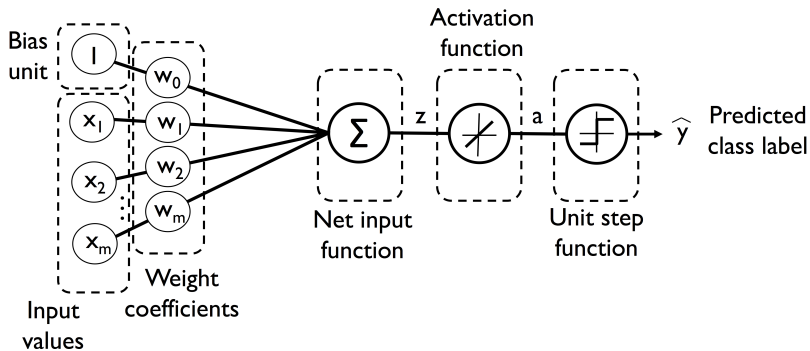
Chapter 12

Multilayer Artificial Neural Network from Scratch

August 19, 2018

- Set of algorithms to train neural networks
- Python libraries available
- Outline
 - Review of Perceptron + Adaline
 - Getting a conceptual understanding of multilayer neural networks
 - Implementing the fundamental backpropagation algorithm for neural network training from scratch
 - Training a basic multilayer neural network for image classification
 - Logistic cost function and the backpropagation algorithm

Single neuron review



- Perceptron

- Update all weights, then recompute \hat{y}
- Weight update done after seeing each sample

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

- Adaline

- Weight update done after entire training set has been seen
- In every epoch, update all weights as follows:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \text{where } \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

- I.e. compute the gradient based on all samples in the training set (this is known as batch gradient descent)
- SGD updates for each training sample
- Mini-batch: middle ground between SGD and batch GD

Weight update details

Partial derivative for each weight w_j in the weight vector \mathbf{w} :

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

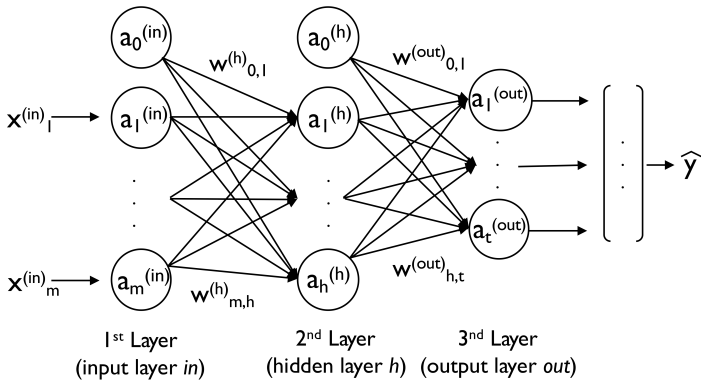
Here $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the *activation* of the neuron, which is a linear function in the case of Adaline: Remember that we defined the *activation function* $\phi(\cdot)$ as follows:

$$\phi(z) = z = a$$

Here, the net input z is a linear combination of the weights that are connecting the input to the output layer:

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

Multi-layer feedforward neural network



Multi-layer Neural Network Architecture

- One input layer, one hidden layer, and one output layer.
- The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer.
- If such a network has more than one hidden layer, we also call it a deep artificial neural network.

- We denote the i th activation unit in the l th layer as $a_i^{(l)}$
- The activation units $a_0^{(1)}$ and $a_0^{(2)}$ are the *bias units*, respectively, which we set equal to 1
- The activation of the units in the input layer:

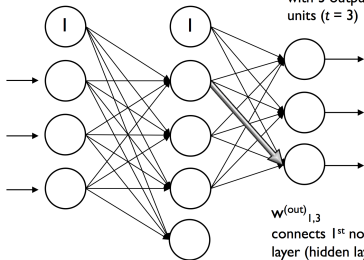
$$\mathbf{a}^{(i)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

Notation summary

Input layer with 3
input units plus
bias unit ($m = 3+1$)

Hidden layer with 4
hidden units plus bias
unit ($d = 4+1$)

Output layer
with 3 output
units ($t = 3$)



Number of layers: $L = 3$

$w^{(out)}_{1,3}$
connects 1st non-bias neuron in the 2nd
layer (hidden layer h) to the 3rd unit in
the 3rd layer (output layer out)

MLP learning procedure

- 1 Starting at the input layer, forward propagate the patterns of the training data ($\mathbf{x}^{(i)}$) through the network to generate an output
- 2 Based on the network output, calculate the error that we want to minimize using a cost function
- 3 **Backpropagate** the error, find its derivative with respect to each weight in the network, and update the model
- 4 Update the weights

In summary: After repeat the four steps for multiple epochs and learn the weights of the MLP, we use **forward propagation** to calculate the network output and apply a threshold function to obtain the predicted class labels.

Forward propagation

- 1 Assume, input has m dimensions
- 2 Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation unit $a_1^{(2)}$ for unit 1 in the hidden layer:

$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \dots + a_m^{(1)} w_{1,m}^{(1)}$$

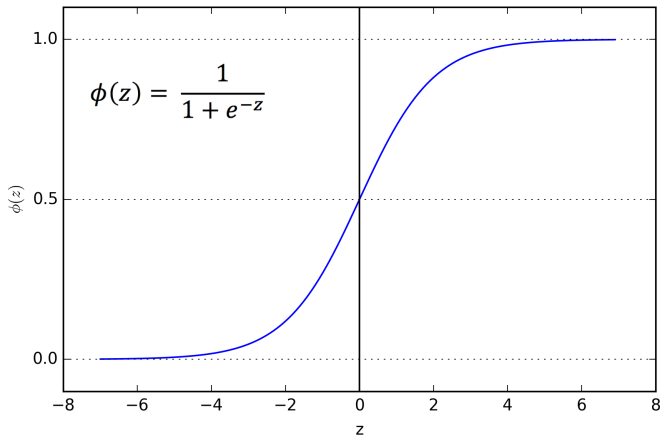
- 3 Compute the activation for unit 1 in the hidden layer:

$$a_1^{(2)} = \phi(z_1^{(2)})$$










- 4 Here $\phi(\cdot)$ is the activation function which has to be differentiable to learn the weights that connect the neurons using GD.
- 5 However, to solve complex problems as image classification, we need non-linear activation function as sigmoid:

$$\phi(z) = \frac{1}{1 + e^{-z}}.$$

Sigmoid function



Activation function

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Backward Propagation

- We compare the result with actual output
- The task is to make the output to neural network as close to actual (desired) output
- How do you reduce the error?
 - We try to minimize the value/weight of neurons those are contributing more to the error and this happens while traveling back to the neurons of the neural network and finding where the error lies.
 - This process is known as **Back Propagation**

Backward Propagation

- Works by determining the loss (or error) at the output and then propagating it back into the network
- The weights are updated to minimize the error resulting from each neuron
- The first step in minimizing the error is to determine the gradient (Derivatives) of each node w.r.t. the final output.
- The round of forward and back propagation iteration is known as one training iteration aka "Epoch"

- For purposes of efficiency and readability, the activation will be written in a matrix form
- Net inputs for the hidden layer:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{a}^{(1)}$$

- Dimensions (ignoring bias units for simplicity)

$$[h \times 1] = [h \times m][m \times 1]$$

- Activations for the hidden layer:

$$\mathbf{a}^{(2)} = \phi(\mathbf{z}^{(2)})$$

Matrix notation

- Generalize computation to all n samples in the training set

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(1)} [\mathbf{A}^{(1)}]^T$$

- Matrix dimensions

$$[h \times n] = [h \times m][n \times m]^T$$

- Activation matrix

$$\mathbf{A}^{(2)} = \phi(\mathbf{Z}^{(2)})$$

- Now activation of the output layer

$$\mathbf{Z}^{(3)} = \mathbf{W}^{(2)} \mathbf{A}^{(2)}$$

- Matrix dimensions

$$[t \times n] = [t \times h][h \times n]$$

- Output of the network

$$\mathbf{A}^{(3)} = \phi(\mathbf{Z}^{(3)}), \mathbf{A}^{(3)} \in \mathbb{R}^{t \times n}.$$

Cost function

Let's dig a little bit deeper into some of the concepts useful in the implementation of our classifier in this Chapter. Such as the logistic cost function and the backpropagation algorithm that we implemented to learn the weights. The logistic cost function is presented in Chapter 3 (sum of squared errors (SSE) was used in Adaline) and is useful to compute the cost of all training samples for each epoch:

$$J(\mathbf{w}) = - \sum_{i=1}^n y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$$

Here, $a^{(i)}$ is the sigmoid activation of the i th unit $a^{(i)} = \phi(z^{(i)})$, $y^{(i)}$ is the real value of a sample i in training set. L2 Regularization term is defined as:

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

In case, we are evaluating the output layer, $m = t$

By adding the L2 regularization term to our logistic function:

$$J(\mathbf{w}) = - \left[\sum_{i=1}^n y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Cost function for all units in output layer

So, we need to generalize the logistic cost function to all activation units t in our network. The cost function (without the regularization term) becomes:

$$J(\mathbf{w}) = - \sum_{i=1}^n \sum_{j=1}^t y_j^{(i)} \log(a_j^{(i)}) + (1 - y_j^{(i)}) \log(1 - a_j^{(i)})$$

Superscript i is the index of a particular sample in training set.

Minimizing the cost function

We want to minimize the cost function $J(\mathbf{w})$, so we calculate the partial derivative with respect to each weight for every layer in the network:

$$\frac{\partial J(\mathbf{w})}{\partial w_{j,i}^{(l)}}$$

Backpropagation algorithm allows us to calculate those partial derivatives to minimize the cost function.

Developing your intuition for backpropagation

- One of the most widely used algorithms to train artificial neural networks very efficiently
- Very computationally efficient approach to compute the partial derivatives of a complex cost function in multilayer neural networks.
- Use those derivatives to learn the weight coefficients for parameterizing such a multilayer ANN.
- The challenge in the parameterization of neural networks is that we are typically dealing with a very large number of weight coefficients in a high-dimensional feature space.
- Many challenges to overcome in this high-dimensional cost surface (local minima) in order to find the global minimum of the cost function.

Developing your intuition for backpropagation

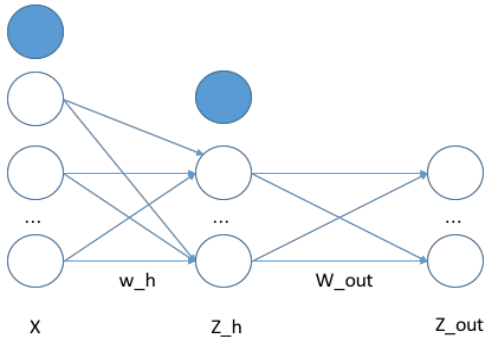
- Backpropagation is simply just a special case of **reverse mode** automatic differentiation (chain rule to compute the derivative of a complex, nested function).
- The trick of **reverse mode** is that we start from right to left: we multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on.
- Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in neural network training.
- To fully understand backpropagation, we need to borrow certain concepts from differential calculus, which is beyond the scope of this course.

One hot representation

Since we implemented an MLP for multiclass classification that returns an output vector of t elements that we need to compare to the $t \times 1$ dimensional target vector in the one-hot encoding representation, for example, the activation of the third layer and the target class (here, class 2) for a particular sample may look like this:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Multilayer Artificial Neural Network for MNIST Example



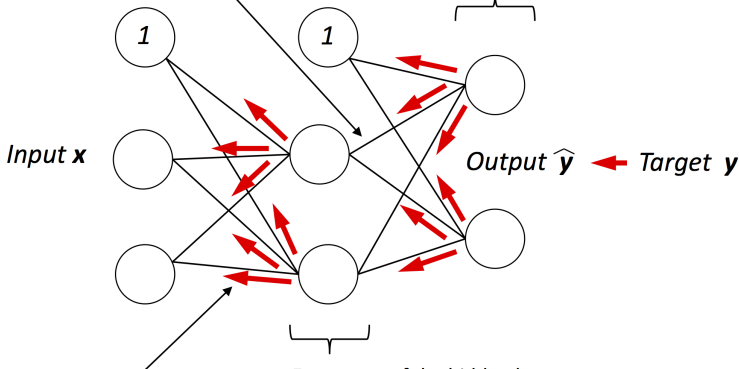
$$\begin{aligned} X &: [n_{samples}, n_{features}], w_h: [n_{features}, n_{hidden}] \\ z_h &= X * w_h + b_h \rightarrow [n_{samples}, n_{hidden}], a_h = \text{sigmoid}(z_h) \\ w_{out} &: [n_{hidden}, n_{output}], z_{out} = a_h * w_{out} + b_{out}, \\ a_{out} &= \text{sigmoid}(z_{out}) \end{aligned}$$

Compute the gradient:

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\mathbf{W}) = a_j^{(h)} \delta_i^{(out)}$$

Error term of the output layer:

$$\delta^{(out)} = \mathbf{a}^{(out)} - \mathbf{y}$$



Compute the gradient:

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\mathbf{W}) = a_j^{(in)} \delta_i^{(h)}$$

Error term of the hidden layer:

$$\delta^{(h)} = \delta^{(out)} (\mathbf{W}^{(out)})^T \odot \frac{\partial \phi(z^{(h)})}{\partial z^{(h)}}$$

Multilayer Artificial Neural Network for MNIST Example

Note: \odot means element-wise multiplication in this context.

$$\Delta w_h = (gradw_h + \lambda * w_h)$$

$$\Delta b_h = gradb_h \text{ (**bias is not regularized**)}$$

$$w_{h-} = \eta * \Delta w_h$$

$$b_{h-} = \eta * \Delta b_h$$

$$\Delta w_{out} = (gradw_{out} + \lambda * w_{out})$$

$$\Delta b_{out} = gradb_{out} \text{ (**bias is not regularized**)}$$

$$w_{out-} = \eta * \Delta w_{out}$$

$$b_{out-} = \eta * \Delta b_{out}$$

MNIST Classification

► [iPython notebook on github](#)