A simple neural network with backpropagation using Python

2019 - 2020 - 2021

Ando Ki, Ph.D.
adki@future-ds.com

Contents

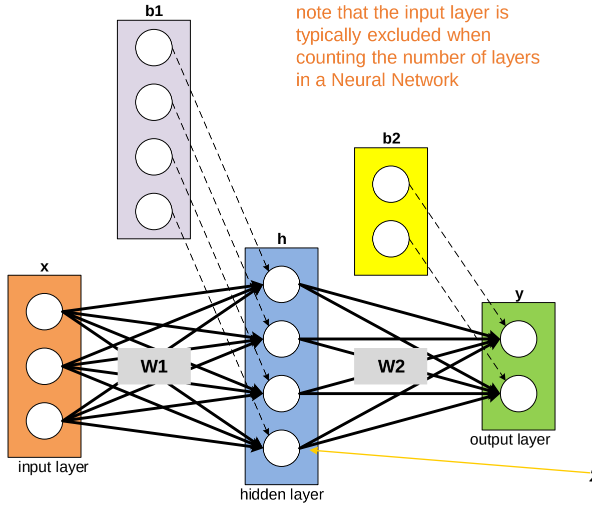
Install modules for Python3.5 on Ubuntu 16.04

- Install necessary Python modules/packages on Python 3.5 (not Python 3.7)
  - $ sudo apt-get update
  - $ sudo apt-get install python3-numpy
  - $ sudo apt-get install python3-scipy
  - $ sudo apt-get install python3-matplotlib

Install modules for Python3.7 on Ubuntu 18.04

- Install necessary Python modules/packages on Python 3.7
  - $ sudo apt update
  - $ sudo apt install python3-pip
  - $ pip3 install numpy
  - $ pip3 install scipy
  - $ pip3 install matplotlib

A simple two-layer neural network

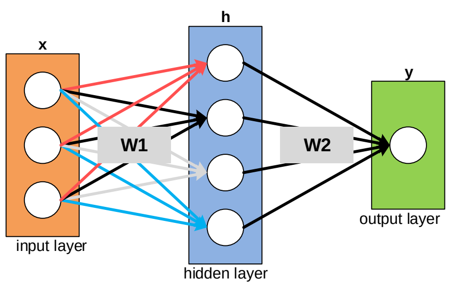


- Input layer, x
- Hidden layer, h
- Output layer, y
- Weight, W1 and W2
- Biases, b1 and b2
- Activation function, f
- Loss function, L

$$z_k = f\left(b_k + \sum_{i=1}^{n\_x}\left(x_i \times W_{(k,i)}\right)\right)$$

Creating neural network (1/2)

- Simplified version removing biases




```
import numpy as np
class NeuralNetwork:
    def __init__(self, n_x, n_h, n_y):
        """ n_x: number of input nodes
            n_h: number of hidden nodes
            n_y: number of output nodes"""
        self.W1     = np.random.rand(n_x, n_h)
        self.W2     = np.random.rand(n_h, n_y)
        self.hidden = np.zeros((1,n_h))
        self.output = np.zeros((n_y, 1))
        self.activation = sigmoid
```

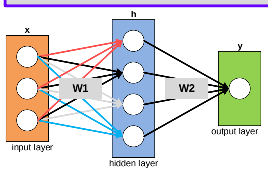
```
nn  =  NeuralNetwork(3,4,1)
```

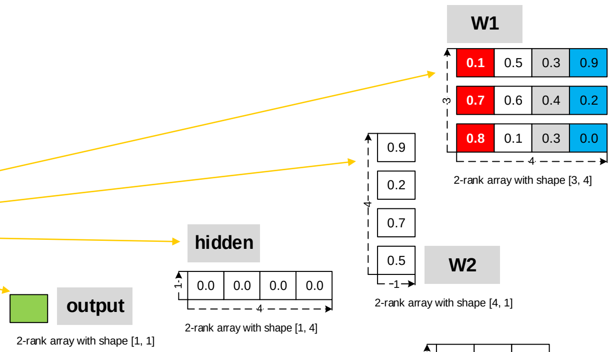

Creating neural network (2/2)

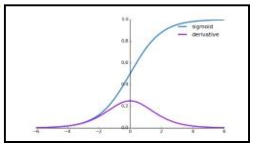
- Simplified version removing biases

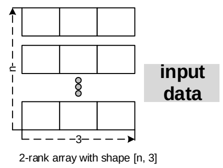
```
import numpy as np
class NeuralNetwork:
    def __init__(self, n_x, n_h, n_y):
        """ n_x: number of input nodes
            n_h: number of hidden nodes
            n_y: number of output nodes"""
        self.W1     = np.random.rand(n_x, n_h)
        self.W2     = np.random.rand(n_h, n_y)
        self.hidden = np.zeros((1, n_h))
        self.output = np.zeros((n_y, 1))
        self.activation = sigmoid
```

```
nn = NeuralNetwork(3,4,1)
```







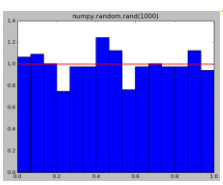


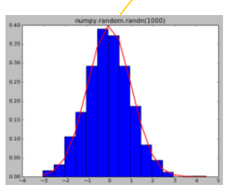
Initialize parameters (i.e., weights)

```
import numpy as np
class NeuralNetwork:
    def __init__(self, n_x, n_h, n_y):
        """ n_x: number of input nodes
            n_h: number of hidden nodes
            n_y: number of output nodes"""
        self.W1     = np.random.rand(n_x, n_h)
        self.W2     = np.random.rand(n_h, n_y)
        self.hidden = np.zeros((1, n_h))
        self.output = np.zeros((n_y, 1))
        self.activation = sigmoid
```

- numpy.random.rand(d0, d1, ...)
  - creates shape (d0, d1, ...) array initialized with uniform distributed over **[0, 1)**.
- numpy.random.randn(d0, d1, ...)
  - create shape (d0, d1, ...) array initialized with standard normal distribution (i.e., Gaussian distribution, mean 1)

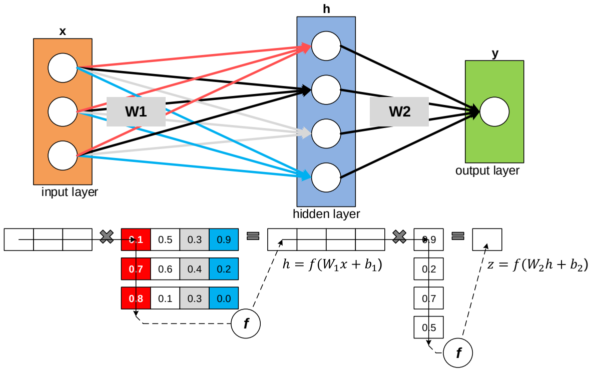
- randint(low[, high, size, dtype]): Return random integers from low (inclusive) to high (exclusive).
- random_integers(low[, high, size]): Random integers of type np.int between low and high, inclusive.
- random_sample([size]): Return random floats in the half-open interval [0.0, 1.0).
- random([size]): Return random floats in the half-open interval [0.0, 1.0).
- ranf([size]): Return random floats in the half-open interval [0.0, 1.0).
- sample([size]): Return random floats in the half-open interval [0.0, 1.0).





Forward propagation

```
class NeuralNetwork:
...
    def feedforward(self, In):
        self.hidden = self.activation(np.dot(In, self.W1))
        self.output = self.activation(np.dot(self.hidden, self.W2))
        return self.output
```



- It calculates the predicted output.
  - numpy.dot(a, b)
    - Dot product of two arrays (i.e., matrix multiplication)

$$h = f(W_1 \times x + b_1)$$

$$z = f(W_2 \times h + b_2) \\ = f\big(W_2 \times \big(f(W_1 \times x + b_1)\big) + b_2\big)$$

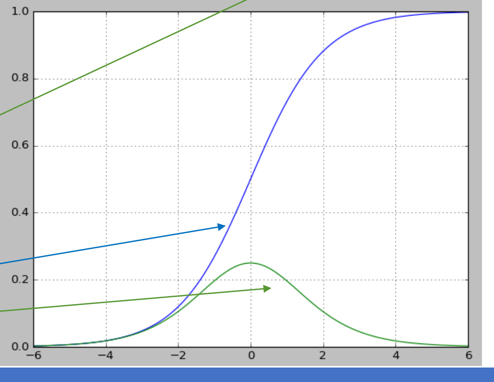
Activation function

- Use one of many activation function

sigmoid(z) =s(z)= $\frac{1}{(1+e^{-z})}$

$$\frac{ds(z)}{dz} = \frac{d}{dz}\left[\frac{1}{(1+e^{-z})}\right] = \frac{d}{dz}(1+e^{-z})^{-1} = -(1+e^{-z})^{-2} \times (-e^{-z}) = \underline{s(z) \times (1 - s(z))}$$

```
import numpy as np
from matplotlib import pyplot as plt

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def dsigmoid(z):
    return sigmoid(z) * ( 1 - sigmoid(z))

if __name__ == "__main__":
   z = np.linspace(-10, 10, 200)
   plt.grid()
   plt.plot(z, sigmoid(z))
   plt.plot(z, dsigmoid(z))
   plt.show()
```



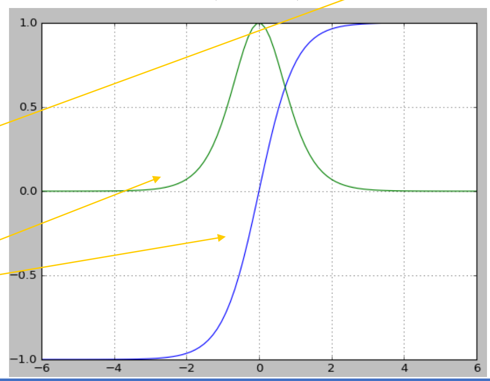
Activation function

- Use one of many activation function

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{(e^z - e^{-z})}{(e^z + e^{-z})}$$

$$\frac{\partial \tanh(z)}{\partial z} = \frac{(e^z + e^{-z}) \times (e^z + e^{-z}) + (e^z - e^{-z}) \times (e^z - e^{-z})}{(e^z + e^{-z})^2} = 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2} = 1 - \tanh(z)^2$$

```
import numpy as np
from matplotlib import pyplot as plt

def tanh(z):
    return np.tanh(z)

def dtanh(z):
    return 1.0 - np.tanh(z)**2

if __name__ == "__main__":
   z = np.linspace(-6, 6, 100)
   plt.grid()
   plt.plot(z, tanh(z))
   plt.plot(z, dtanh(z))
   plt.show()
```



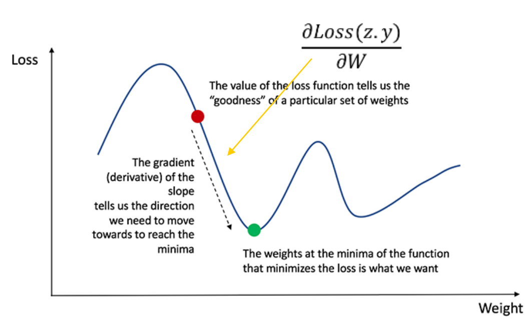
Loss function

- Select one of many loss functions.
  - a way to evaluate the "goodness" of our predictions (i.e. how far off are our predictions)
  - ➔ *loss: measure the error the prediction*

$$\mathrm{SSE} = \sum_{i=1}^{n} \left((z - y) \right)^2$$

- Sum of Squared Error (SSE)
  - where 'y' for desired value, 'z' for calculated value.

- Our goal in training is to find the best set of weights and biases that minimizes the loss function.
- Calculate the derivative of the loss function with respect to the weights and biases.



Loss function and backpropagation

Loss(y, z) = $\sum_{i=1}^{n}(z-y)^2$

Where x=input, z=output, y=desired output

$$\frac{\partial Loss(z-y)}{\partial W} = \frac{\partial \sum_{i=1}^{n}(z-y)^2}{\partial W}$$

$$= \frac{\partial \sum_{i=1}^{n}(z-y)^2}{\partial z} \times \frac{\partial z}{\partial m} \times \frac{\partial m}{\partial W}$$

$$m = W_1 x + b_1$$

$$= \frac{\partial \sum_{i=1}^{n}(z-y)^2}{\partial z} \times \frac{\partial z}{\partial m} \times \frac{\partial (Wx+b)}{\partial W}$$

$$\begin{aligned} z &= f(W_2 h + b_2) \\ &= f(W_2 f(W_1 x + b_1) + b_2) \\ &= f(W_2 f(m) + b_2) \\ &= f(....) \\ &= activation\ function \end{aligned}$$

$$= 2(z-y) \times \frac{\partial z}{\partial m} \times x$$

$$= 2(z-y) \times derivative\_of\_activatio\_function \times x$$

https://youtu.be/tIeHLnjs5U8

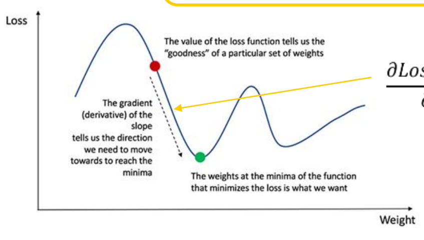
Backward propagation

```
class NeuralNetwork:
...
    def backprop(self, In, Out, Desired):
        diff = Out - Desired
        d_W2 = np.dot(self.hidden.T, (2*diff*self.activation(Out, True)))
        d_W1 = np.dot(In.T, np.dot(2*diff*self.activation(Out, True), self.W2.T)*self.activation(self.hidden, True))
        self.W1 -= d_W1
        self.W2 -= d_W2
```

x * 2 * (z – y) * df(x)/dx

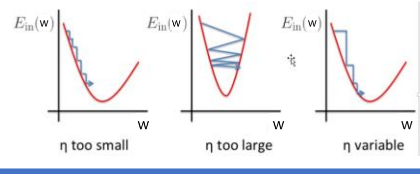
$$\frac{dL}{dW^1} = a^{l-1} \times \delta^{l+1} \times W^{l+1} \times \frac{dg(z)}{dz}$$

Update weights
*negative slope causes increase weights
*positive slope causes decrease weight



$$\frac{\partial Loss(z.y)}{\partial W}$$

- The ratio of updating parameters.
  - Learning rate.
  - W = W – η ·ΔW



Running an example

```
class NeuralNetwork:
...
if __name__ == "__main__":
    X = np.array([[0,0,1],
                  [0,1,1],
                  [1,0,1],
                  [1,1,1]])

    Y = np.array([[0],
                  [1],
                  [1],
                  [0]])

    nn = NeuralNetwork(X.shape[1],4,Y.shape[1])

    for i in range(1000):
        z = nn.feedforward(X)
        nn.backprop(X, z, Y)

    print(nn.output)
```

Input data shape (4, 3)

Desired output shape (4, 1)

Building network 3:4:1

Training 1000 iterations

Forward calculation

Backward propagation

Print final result

```
adki@AndoUbuntu: ~/work/seminars/2
[adki@AndoUbuntu] python simple.py
[[ 0.01250298]
 [ 0.96425908]
 [ 0.9650013 ]
 [ 0.04393822]]
[adki@AndoUbuntu]
```

Compare with the desired output Y

All together (1/3)

```
import numpy as np
from matplotlib import pyplot as plt

def sigmoidFunction(z, derivative=False):
    """derivative: get normal when false, while derivative when true"""
    if derivative: return z * (1.0 – z)
    else         : return 1.0/(1.0 + np.exp(-z))

def lossFunction(Out, Desired):
    """Out: result of forward
      Desired: desired result
      It calculate Sum of Squared Error."""
    return ((Desired - Out)**2).sum()

class NeuralNetwork:
    def __init__(self, n_x, n_h, n_y):
        """n_x: number of input nodes
          n_h: number of hidden nodes
          n_y: number of output nodes"""
        self.W1     = np.random.rand(n_x, n_h)
        self.W2     = np.random.rand(n_h, n_y)
        self.hidden = np.zeros((1, n_h))
        self.output = np.zeros((n_y, 1))
        self.activation = sigmoidFunction
```

Use numpy

Use mathplot

$$y = \frac{1}{1+e^{-z}}$$

Get derivative of sigmoid when True

Compare with the desired output Y

SSE= $\sum_{i=1}^{n}\left((z-y)\right)^2$

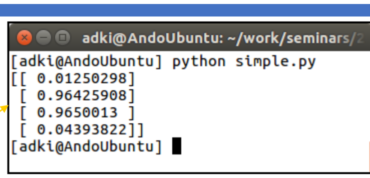
Sigmoid

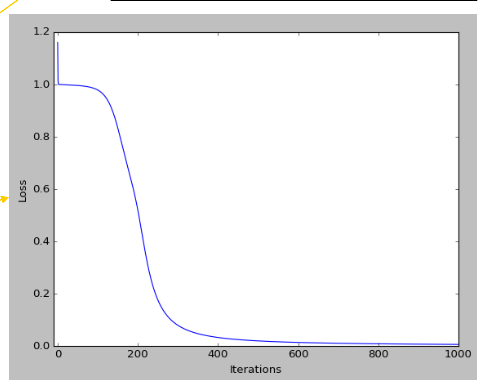
All together (2/3)

```
def feedforward(self, In):
    """In: input data"""
    self.hidden = self.activation(np.dot(In, self.W1))
    self.output = self.activation(np.dot(self.hidden, self.W2))
    return self.output

def backprop(self, In, Out, Desired):
    """In: input data
       Out: the result of forwared propagation
       Desired: desired value
       application of the chain rule to find derivative of the loss function
       with respect to W2 and W1"""
    diff = Out - Desired
    d_W2 = np.dot(self.hidden.T, (2*diff*self.activation(Out, True)))
    d_W1 = np.dot(In.T,\
              np.dot(2*diff*self.activation(Out, True), self.W2.T)*self.activation(self.hidden, True))
    # update the weights with the derivative (slope) of the loss function
    self.W1 -= d_W1
    self.W2 -= d_W2
```


All together (3/3)

```
if __name__ == "__main__":
    X = np.array([[0,0,1],
                  [0,1,1],
                  [1,0,1],
                  [1,1,1]])
    Y = np.array([[0],[1],[1],[0]])

    nn = NeuralNetwork(X.shape[1],4,Y.shape[1])

    loss_values = []
    for i in range(1000):
        z = nn.feedforward(X)
        nn.backprop(X, z, Y)
        loss = lossFunction(z, Y)
        loss_values.append(loss)

    print(nn.output)

    plt.plot(loss_values)
    plt.xlabel("Iterations"); plt.xlim(-10, len(loss_values))
    plt.ylabel("Loss")
    plt.show()
```





Running 'simple.py' example

- This example shows how to program a simple neural network with backpropagation
  - Step 1: (ignore this step if you do not use virtual environment) go to your project directory and invoke Python virtual environment
    - [user@host] cd $(PROJECT)/codes/python-projects/nn_backpropagation
    - [user@host] source ~/my_python/bin/activate
  - Step 2: see the codes
  - Step 3: run
    - [user@host] python simple.py
    - or
    - [user@host] python3 simple.py

```
[user@host] cd $(PROJECT)/codes/python-projects/nn_backpropagation
[user@host] source ~/my_python/bin/activate
(my_python)$ python simple.py
(my_python)$ deactivate
[user@host]
```


Considerations

- Initial value issues
  - Determines local or global minima
- Activation function issues
- Error/loss function issues
- Learning rate issues
- Optimizing function issues
- How to save and load trained results, i.e., weights.
- How to separate training and inference steps.
- How to expend multi-layer more than two.

A full version (1/4)

```
import numpy as np
from matplotlib import pyplot as plt
import pickle

def sigmoidFunction(z, derivative=False):
    """derivative: get normal when false, while derivative when true"""
    if derivative: return z * (1.0 –  z)
    else        : return 1.0/(1.0 + np.exp(-z))

def lossFunction(Out, Desired):
    """Out: result of forward
      Desired: desired result
      It calculate Sum of Squared Error."""
    return ((Desired - Out)**2).sum()

class NeuralNetwork:
    def __init__(self, n_x, n_h, n_y, init=True):
        """n_x: number of input nodes
          n_h: number of hidden nodes
          n_y: number of output nodes"""
        if init:
          self.W1  = np.random.rand(n_x, n_h)
          self.W2  = np.random.rand(n_h, n_y)
        else:
          self.W1  = np.zeros((n_x, n_h))
          self.W2  = np.zeros((n_h, n_y))
          self.hidden = np.zeros((1, n_h))
          self.output = np.zeros((n_y, 1))
          self.activation = sigmoidFunction
          self.inference = self.feedforward
```

binary protocols for serializing and de-serializing a Python object structure (i.e., provides methods to save and restore Python objects.

initialized weights using random number when 'init' is 'True'.

A full version (2/4)

```
    def feedforward(self, In):
        """In: input data"""
        self.hidden = self.activation(np.dot(In, self.W1))
        self.output = self.activation(np.dot(self.hidden, self.W2))
        return self.output

    def backprop(self, In, Out, Desired):
        """In: input data
          Out: the result of forwared propagation
          Desired: desired value
          application of the chain rule to find derivative of the loss function
          with respect to W2 and W1"""
        diff = Out - Desired
        d_W2 = np.dot(self.hidden.T, (2*diff*self.activation(Out, True)))
        d_W1 = np.dot(In.T,\
                  np.dot(2*diff*self.activation(Out, True), self.W2.T)*self.activation(self.hidden, True))
        # update the weights with the derivative (slope) of the loss function
        self.W1 -= d_W1
        self.W2 -= d_W2
```

A full version (3/4)

```
def train(self, In, Desired, iter=1000):
    """In: input data
      Desired: desired values
      iter: iteration"""
    self.loss_values = []
    for i in range(1000):
        z = self.feedforward(In)
        self.backprop(In, z, Desired)
        loss = lossFunction(z, Desired)
        self.loss_values.append(loss)

def save(self, file):
    """file: file name to write weights to"""
    with open(file, 'wb') as f:
        params = { "W1" : self.W1, "W2": self.W2 }
        pickle.dump(params, f)

def load(self, file):
    """file: file name to read weights from"""
    with open(file) as f:
        params = { "W1" : [], "W2": []}
        params = pickle.load(f)
    self.W1 = params["W1"]
    self.W2 = params["W2"]
```

Perform training for a certain iterations

Save resultant weights

Load trained weights

A full version (4/4)

```
if __name__ == "__main__":
    X = np.array([[0,0,1],
                  [0,1,1],
                  [1,0,1],
                  [1,1,1]])
    Y = np.array([[0],[1],[1],[0]])

    nn = NeuralNetwork(X.shape[1],4,Y.shape[1])

    loss_values = []
    for i in range(1000):
        z = nn.feedforward(X)
        nn.backprop(X, z, Y)
        loss = lossFunction(z, Y)
        loss_values.append(loss)
    nn.save('weight.txt')

    print(nn.output)

    #plt.figure()
    plt.plot(loss_values)
    plt.xlabel("Iterations")
    plt.xlim(-10, len(loss_values))
    plt.ylabel("Loss")
    plt.show()
```

Standalone training code

```
import sys
import numpy as np
from matplotlib import pyplot as plt
import simple_all as nn

if __name__ == "__main__":
    X = np.array([[0,0,1],
                  [0,1,1],
                  [1,0,1],
                  [1,1,1]])
    Y = np.array([[0],
                  [1],
                  [1],
                  [0]])

    if len(sys.argv)==1: wfile='weight.txt'
    else               : wfile=sys.argv[1]

    net = nn.NeuralNetwork(X.shape[1],4,Y.shape[1])

    net.train(X, Y, 1000)
    net.save(wfile)
```

```
#plt.figure()
plt.plot(net.loss_values)
plt.xlabel("Iterations")
plt.xlim(-10, len(net.loss_values))
plt.ylabel("Loss")
plt.show()
```

Prepare data-set to train

Prepare desired data corresponding to the train-data

Get file name to store trained weights to

Standalone inference code

```
import sys
import numpy as np
import simple_all as nn

if __name__ == "__main__":
   net = nn.NeuralNetwork(3, 4, 1, False)

   if len(sys.argv)==1: wfile='weight.txt'
   else               : wfile=sys.argv[1]

   net.load(wfile)

   X = np.array([[0,1,0]])

   z = net.inference(X)
   print "X: ", X, "==>", z

   X = np.array([[1,1,0]])
   z = net.inference(X)
   print "X: ", X, "==>", z
```

Prepare data-set to train

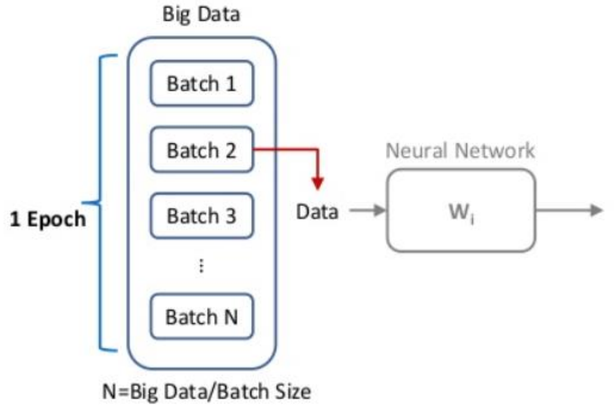
Get file name to read trained weights from

Load pre-trained weights

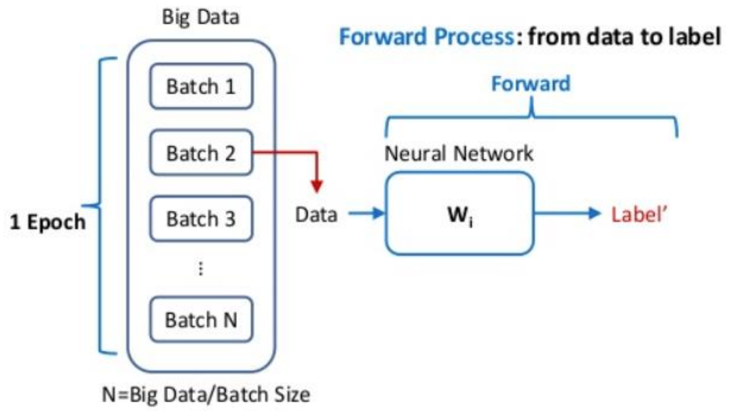
Prepare new data

Run inference

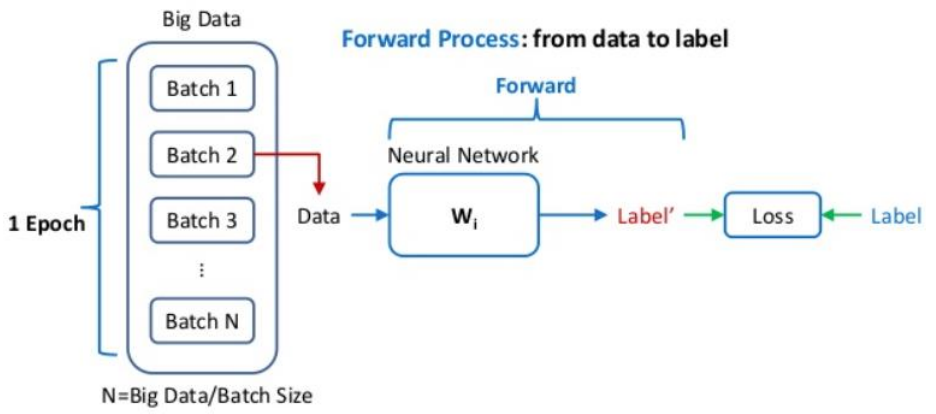
Neural network in brief



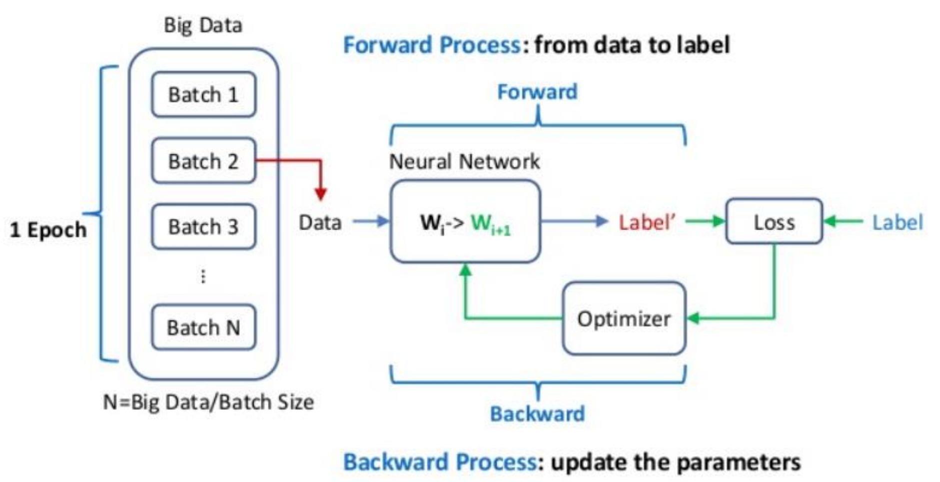
Neural network in brief



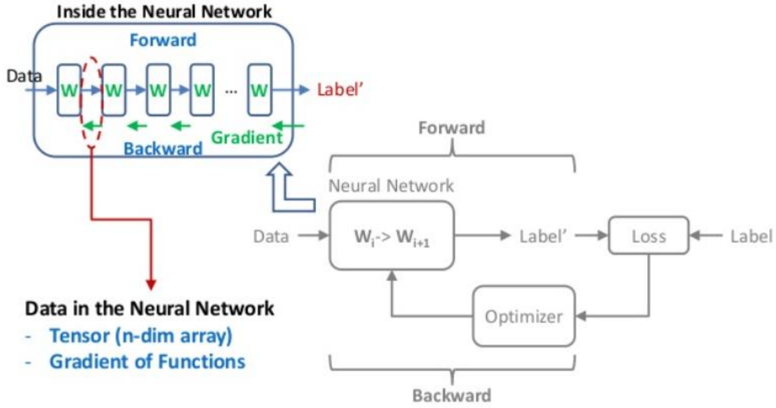
Neural network in brief



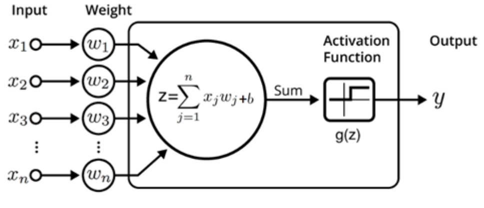
Neural network in brief



Neural network in brief



Backpropagation: single-neuron case (1/2)



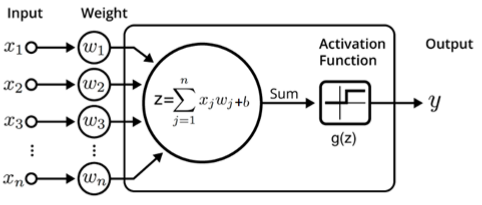
- xj: input
- Wj: weight
- b: bias
- g(): activation function
- z: sum of xj*Wj+b
- y: output of activation function
- e: expected value

- Loss function
  - $L(y, e) = ½ \cdot (y - e)^2$
    - y: output value
    - e: expected value
    - ½: make life easy
- variation of Wi causes z to vary,
- variation of z causes g(z) to vary,
- variation of g(z) causes L() to vary.

- Let get gradient of y against Wi
  - $\frac{\partial L(y,e)}{\partial Wi} = \frac{\partial L(y,e)}{\partial y} \times \frac{\partial g(z)}{\partial z} \times \frac{\partial \sum(xj \cdot Wj+b)}{\partial Wi}$
    - L(y,e)=1/2*(y-e)**2
    - y = g(z)
    - z=sum(xj*Wj+b)

refer to: https://towardsdatascience.com/back-propagation-the-easy-way-part-1-6a8cde653f65

Backpropagation: single-neuron case (2/2)

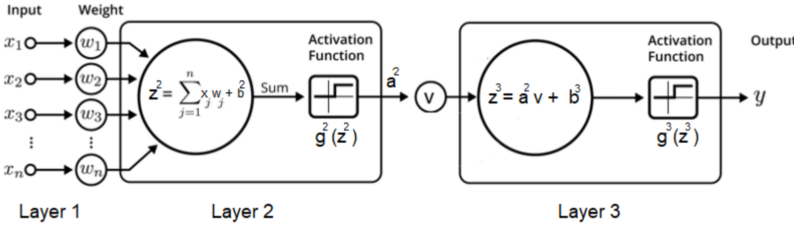


- gradient of L against Wi (applying chain rule)
  - $\frac{\partial L(y,e)}{\partial Wi} = \frac{\partial L(y,e)}{\partial y} \times \frac{\partial g(z)}{\partial z} \times \frac{\partial \sum(xj \cdot Wj+b)}{\partial Wi}$
    - L(y,e)=1/2*(y-e)**2
    - y = g(z)
    - z=sum(xj*Wj+b)

- $\frac{\partial L(y,e)}{\partial y}$
  - $\frac{d(1/2 \cdot (y-e)^2)}{dy} = \frac{d(1/2 \cdot (y-e)^2)}{dy} = (y-e)$
- $\frac{\partial g(z)}{\partial z}$
  - derivative of activation function g(z)
- $\frac{\partial \sum(xj \cdot Wj+b)}{\partial Wi}$
  - $\frac{d(x1 \cdot W1+x2 \cdot W2+\cdots+xn \cdot Wn)}{dWi} = \frac{d(xi \cdot Wi)}{dWi} = xi$

- $\boxed{\frac{\partial L(y,e)}{\partial Wi} = (y-e) \cdot \frac{\partial g(z)}{\partial z} \cdot xi}$

Backpropagation: two-neuron case (1/3)

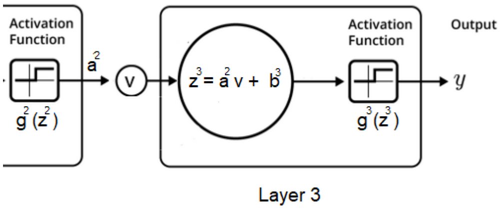


- x: input
- W: weight
- b: bias
- g(): activation function
- z: sum of x*W+b
- y: output of activation function
- e: expected value

- for layer 3
  - variation of v causes z3 to vary,
  - variation of z3 causes g3(z3) to vary,
  - variation of g3(z3) causes L(y,e) to vary

refer to: https://towardsdatascience.com/back-propagation-the-easy-way-part-1-6a8cde653f65

Backpropagation: two-neuron case (2/3) – layer 3

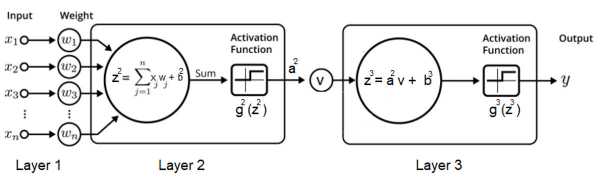


- Gradient of L against v
  - $\frac{\partial L(y,e)}{\partial v} = \frac{\partial L(y,e)}{\partial b} \times \frac{\partial g(z)}{\partial z} \times \frac{\partial \sum(a \cdot v + b)}{\partial v}$
    - L(y,e)=1/2*(y-e)**2
    - b = g(z) = y
    - z=sum(xj*Wj+b)
  - where v is one of W2

- $\frac{\partial L(y,e)}{\partial b} = \frac{\partial L(y,e)}{\partial y}$
  - $\frac{d(1/2 \cdot (y-e)^2)}{dy} = \frac{d(1/2 \cdot (y-e)^2)}{dy} = (y-e)$
- $\frac{\partial g(z)}{\partial z}$
  - derivative of activation function g(z)
- $\frac{\partial \sum(a \cdot v + b)}{\partial v}$
  - $\frac{d(a1 \cdot W1 + a2 \cdot W2 + \cdots + an \cdot Wn)}{dv} = \frac{d(a \cdot v)}{dv} = a$

- $\boxed{\frac{\partial L(y,e)}{\partial v} = (y-e) \cdot \frac{\partial g(z)}{\partial z} \cdot a}$
  - where a is result of previous layer

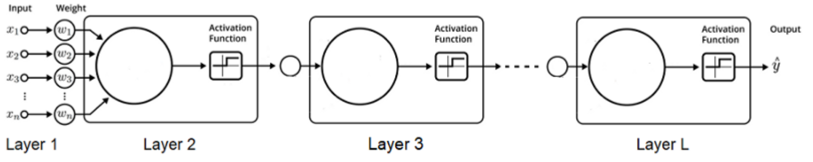
Backpropagation: two-neuron case (3/3) – layer 2



- for layer 2
  - variation of $\mathbf{z^2}$ affects $\mathbf{g^2(z^2)}$
  - variation of $\mathbf{g^2(z^2)}$ affects $\mathbf{z^3}$ (note that at this point **v** is considered fixed)
  - variation of $\mathbf{z^3}$ affects $\mathbf{g^3(z^3)}$
  - variation of $\mathbf{g^3(z^3)}$ affects $\boldsymbol{\mathcal{L}(y, \hat{y})}$

- gradient of L against Wi (applying chain rule)
  - $\partial\mathcal{L}/\partial w_i = (\partial\mathcal{L}/\partial a^3 * \partial a^3/\partial z^3 * \partial z^3/\partial a^2) * \partial a^2/\partial z^2 * \partial z^2/\partial w_i$
    - $\partial z^3/\partial a^2 = \partial(a^2 * v))/\partial a^2 = v$
    - $\partial a^2/\partial z^2 = \partial g^2(z^2)/\partial z^2 = g^{2\prime}(z^2)$
    - $\partial z^2/\partial w_i = x_i$
  - $\partial\mathcal{L}/\partial w_i = \delta^3 * v * g^{2\prime}(z^2) * x_i$
    - $\delta^3 = (a^3 - y) * g^{3\prime}(z^3)$

Sequence of layers



- For any layer $\boldsymbol{l} \leq L$
  $\partial\mathcal{L}/\partial w^l = \delta^l * a^{l-1}$
  $\partial\mathcal{L}/\partial b^l = \delta^l$
  where a^{l-1} is the output of the layer $\boldsymbol{l}$-1, or if we are at layer 1 it will be the input x.

- For layer L
  $\delta^L = (a^L - y) * g^{L\prime}(z^L)$

- For any other layer $\boldsymbol{l} < L$
  $\delta^l = \delta^{l+1} * w^{l+1} * g^{l\prime}(z^l)$

