Neural Networks from Scratch. Easy vs hard

Sivasurya S Follow

Aug 17, 2018 · 10 min read

Artificial Neural Networks are like magic for those who do not understand how they work. why be a spectator, when you can be a magician!. I would like to discuss how easy it is to get started with a standard machine learning library (`keras`) and how interesting it could get, if we try to implement a neural network from scratch(`numpy`) along with a 'little' bit of math.

What's new!

Well, There are already several articles on how to develop a neural network from scratch. But, In most of those articles keeping beginners in mind, a simple network without any discussion of cost or activation function is implemented. When the problem at hand changes, the network cannot function properly due to the range of input values, type of activation functions, error function. So, let's dig deeper.

Here's what we are going to do. Prepare some dataset, create a suitable network architecture, implement the architecture in the easy way and then the hard way.

Data preparation

Let us use a digits dataset from sklearn. It is a multi-class classification.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split


dig = load_digits()
plt.gray()
plt.matshow(dig.images[25])
```

Looks like a mini version of mnist. It should be fine. This dataset contains inputs in `data` and outputs in `target` variable. `target` values ranges from

0–9. It has to be converted to one-hot representation, where the array contains all zeros, except the index of the value as 1. i.e, value 4 will be represented as [0,0,0,0,1,0,0,0,0]. `pandas` does it in a single function. Such multi-class classification problems must be translated into one-hot representations for the NN model to train, due to the nature of error function.

```
onehot_target = pd.get_dummies(dig.target)
```

That's it!. The dataset is ready. Now, let's split it into training and testing datasets. `train_test_split` also randomizes the instances

```
x_train, x_val, y_train, y_val = train_test_split(dig.data,
onehot_target, test_size=0.1, random_state=20)

# shape of x_train :(1617, 64)
# shape of y_train :(1617, 10)
```

There are 1617 instances in the training set. Each input instance contains an array of 64 values and the respective output contains an array of 10 values.

```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

%matplotlib inline

dig = load_digits()
plt.gray()
plt.matshow(dig.images[25])

onehot_target = pd.get_dummies(dig.target)

x_train, x_val, y_train, y_val = train_test_split(dig.data, onehot_target, test_size=0.1, random
# shape of x_train :(1617, 64)
# shape of y_train :(1617, 10)
```

nn_dataprep.py hosted with ♡ by GitHub view raw

Dataset preparation

. . .

Artificial Neural Network Architecture:

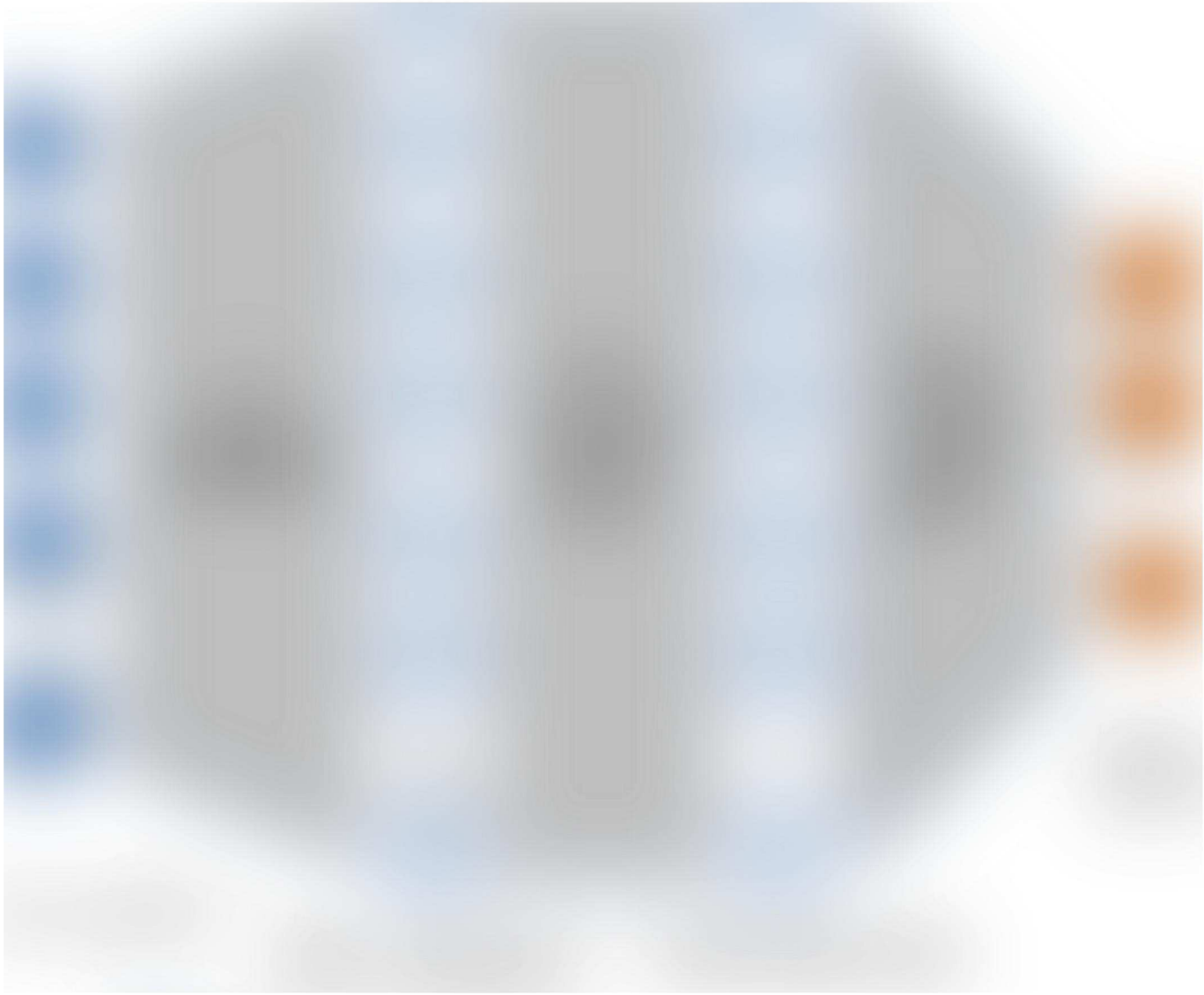
Let us construct a four-layer network (input, 2-hidden layers, output).

Input layer - 64 neurons (input image array)

Hidden layer 1 - 128 neurons (arbitrary)

Hidden layer 2 - 128 neurons (arbitrary)

Output layer - 10 neurons (output one-hot array)



ANN architecture

. . .

Keras — The Easy way:

`Keras` is a fantastic library to get started. Quick and easy.

```
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop, Adadelta, Adam


model = Sequential()
model.add(Dense(128, input_dim=x_train.shape[1],
activation='sigmoid'))
model.add(Dense(128, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
```

In our case, we are going to build a fully connected NN, which means all the neurons in the current layer are connected to all the neurons in the next layer. Thus, we have defined the model as `sequential` and added three

consecutive layers. It is important to note that there is no need to add a separate layer for input. As input layer will be automatically initialized when the first hidden layer is defined.

The model adds the first hidden layer with 128 neurons, where also the `input_dim` specifies the size of the input layer. And then the second hidden layer with the same 128 neurons. Finally, the output layer with 10 neurons.

```
model.summary()
```


The model is not complete, without specifying the cost function and gradient descent optimization.

```
model.compile(optimizer=Adadelta(), loss='categorical_crossentropy',
metrics=['categorical_accuracy'])
model.fit(x_train, y_train, epochs=50, batch_size=64)
```

Well. That's the easy way of doing it. Putting it altogether,

```
import pandas as pd
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop, Adadelta, Adam
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

dig = load_digits()
onehot_target = pd.get_dummies(dig.target)
x_train, x_val, y_train, y_val = train_test_split(dig.data, onehot_target, test_size=0.1, random

model = Sequential()
model.add(Dense(128, input_dim=x_train.shape[1], activation='sigmoid'))
model.add(Dense(128, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))

model.summary()

model.compile(optimizer=Adadelta(), loss='categorical_crossentropy', metrics=['categorical_accur
model.fit(x_train, y_train, epochs=50, batch_size=64)

scores = model.evaluate(x_val, y_val)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

nn_keras.py hosted with ♡ by GitHub view raw

ANN using keras

. . .

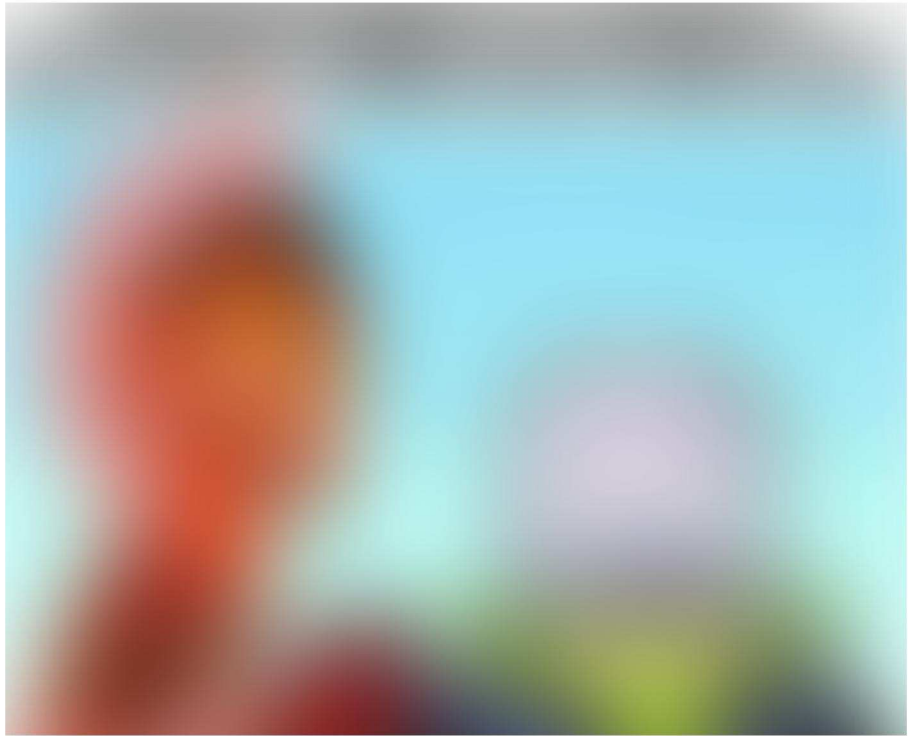
ANN from Scratch — The hard way

Gear up!. It's the hard way. The algorithm trains the model with two main processes. Feed-forward and back propagation. Feed-forward predicts the output for the given input with some weights and back-propagation trains the model by adjusting the weights. So, it is essential to initialize the weights first.

```
import numpy as np

class MyNN:
    def __init__(self, x, y):
        self.input = x
        self.output = y
        neurons = 128       # neurons for hidden layers
        self.lr = 0.5       # user defined learning rate
        ip_dim = x.shape[1] # input layer size 64
        op_dim = y.shape[1] # output layer size 10
        self.w1 = np.random.randn(ip_dim, neurons) # weights
        self.b1 = np.zeros((1, neurons))           # biases
        self.w2 = np.random.randn(neurons, neurons)
        self.b2 = np.zeros((1, neurons))
```

```
self.w3 = np.random.randn(neurons, op_dim)
self.b3 = np.zeros((1, op_dim))
```

Starting with a good set of weights will surely provide a local minima quickly. Apart from slowly reaching a local minima, bad set of weights some times may never converge. Initial weights should be randomized in-order to do symmetry breaking. The values should never be zero but closer to zero, so that the output does not skyrocket. And it is better to have positive as well as negative values so that apart from magnitude, direction also differs. So go with normal distribution. Initializing bias vectors to zero is fine.



source: Machine learning memes for convolutional teens, facebook

Feed-forward

Forward pass

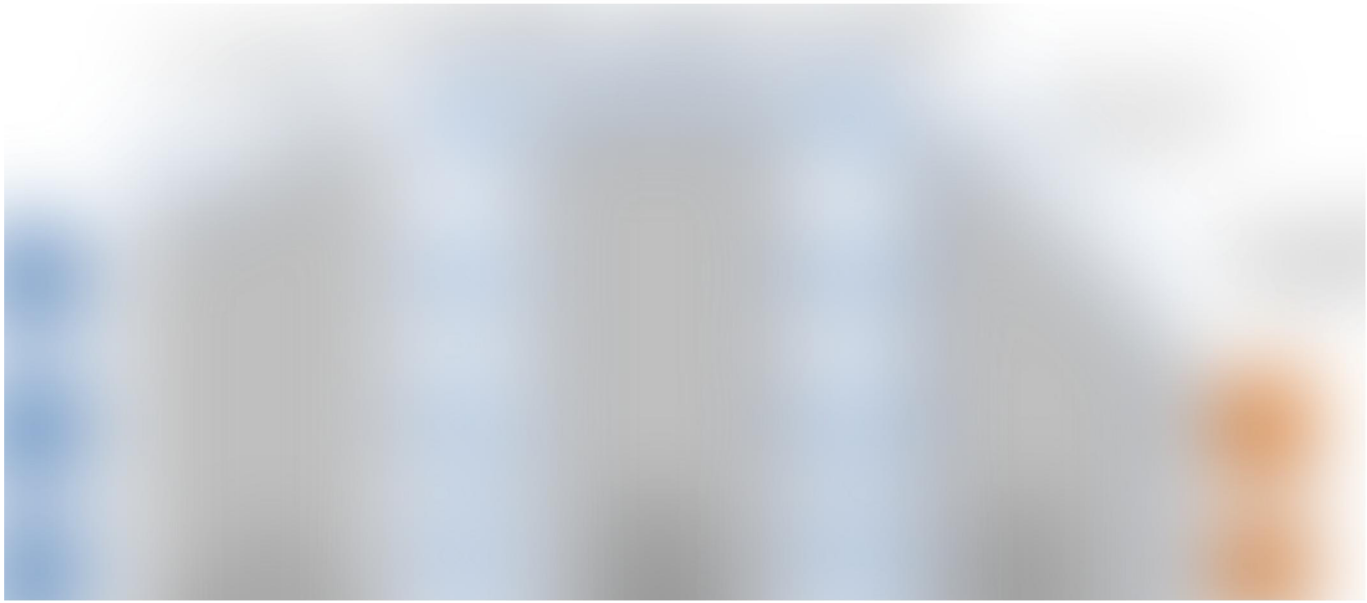
```
def sigmoid(s):
    return 1/(1 + np.exp(-s))


# for numerical stability, values are normalised
def softmax(s):
    exps = np.exp(s - np.max(s, axis=1, keepdims=True))
    return exps/np.sum(exps, axis=1, keepdims=True)


def feedforward(self):
    z1 = np.dot(self.x, self.w1) + self.b1
    self.a1 = sigmoid(z1)
    z2 = np.dot(self.a1, self.w2) + self.b2
    self.a2 = sigmoid(z2)
    z3 = np.dot(self.a2, self.w3) + self.b3
    self.a3 = softmax(z3)
```

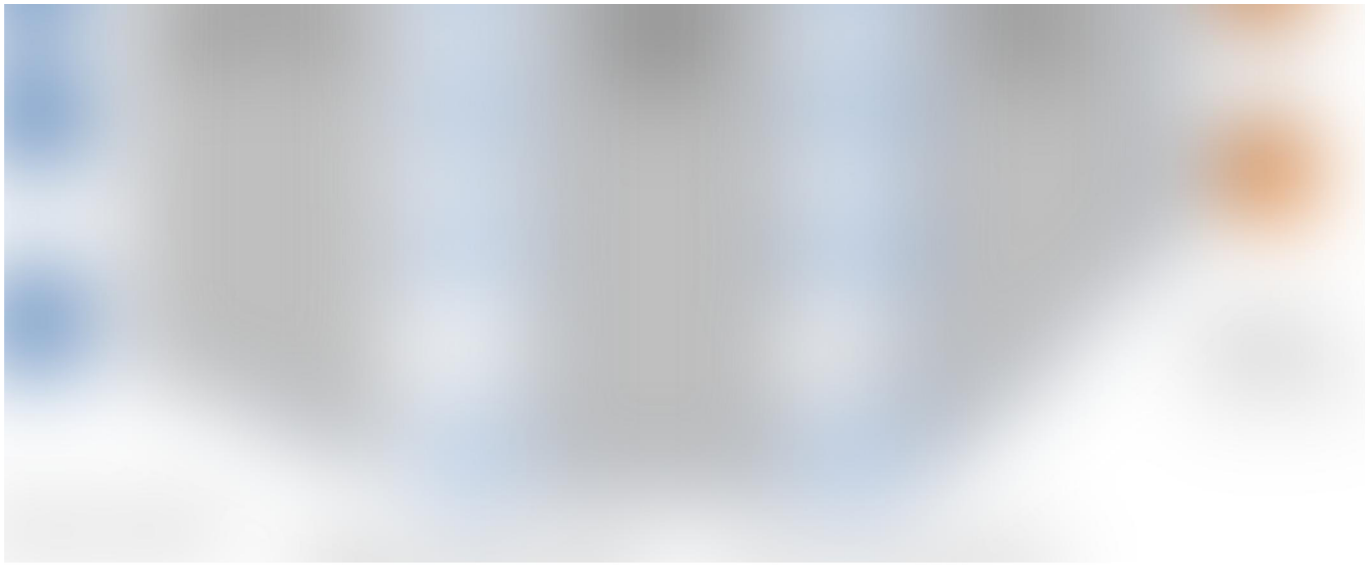
Feed-forward process is quite simple. `(input x weights) + bias` computes `z` and it is passed into the layers, which contains specific activation functions. These activation functions produces the output `a` . The output of the current layer will be the input to the next layer and so on. As you could see, the first and second hidden layer contains *sigmoid* function as the activation function and the output layer has *softmax* as the activation

function. The final result `a3` produced by the softmax is the output of the neural network.

The type of function applied on the layer makes a lot of difference. Sigmoid function squishes the input to (0, 1) and Softmax also does the same. But, Softmax ensures the sum of the outputs equal to 1. In case of our output, we would like to measure what is the probability of the input to each class. for instance, if an image has a probability of 0.9 to be digit 5, it is sensible to have 0.1 probability distributed among the other classes which is done by softmax.

Back-propagation





Backward pass

Okay. Back-prop may look tricky, as you are dealing with multiple layers, multiple weights, loss function, gradients. Do not worry, we will try out it mathematically, intuitively and with code implementation. Fundamentally back-prop computes the error from the output of feed-forward and the true value. This error is back-propagated to all the weight matrices through computing gradients in each layer and these weights are updated. Sounds simple right. Ummm. let's see.

Loss function:

Cost functions and loss functions are interchangeably used. Strictly speaking loss function is when computed for a single instance, and cost function is for the whole network. Cost function is a way of measuring how well your network is performing comparing with the true values. Actually, we will never use cost function anywhere in the training, but we have to compute the gradient of the cost function w.r.t weights and biases for their update. So, computing the loss is just to see, how good are we doing with each epoch.

The most commonly used loss function is mean squared error. As we are dealing with Multi-class classification problem, the output will be a probability distribution. we have to compare it with our real values, which is also a probability distribution and find the error. For such conditions, it is highly recommended to go with cross-entropy as the cost function. Why because, cross-entropy function is able to compute error between two probability distributions.

Chain — rule:

Let us consider the cost function as c, From the feed forward, we know that

```
z = xw + b                  -> z = function(w)
a = sig(z) or softmax(z)    -> a = function(z)
c = -(y*log(a3))            -> c = function(a)
```

Thus, from the output all we have to do is find the error and how much does each weights influences the output. In other words, find the derivative of cost function w.r.t w3. Yes, Back propagation is nothing but calculation of derivatives using chain rule.

Outer layer:

$$\frac{dc}{dw3} = \frac{dc}{da3} \cdot \frac{da3}{dz3} \cdot \frac{dz3}{dw3}$$

```
z3 = a2w3 + b3
a3 = softmax(z3)
```

```
dz3/dw3 = a2
da3/dz3 = softmax derivative(z3)
dc/da3  = cost function derivative(a3) = -y/a3
```

Surprisingly, as cross-entropy is often used with softmax activation function, we do not really have to compute both of these derivatives.

Because, some of the parts of these derivatives cancel each other as clearly explained here. Thus, `predicted value − real value` is the result of their product.

```
Let, a3_delta be the product of these terms as it will be needed in
the upcoming chain rules.
            dc    da3
a3_delta = --- . ---
            da3   dz3


Thus, a3_delta = a3-y (the error to be propagated)


dc
---  = (a3 - y) . a2
dw3
w3 = w3 - dc/dw3


For changes in biases,
dc     dc    da3   dz3
---  = --- . --- . ---
db3    da3   dz3   db3


dz3/db3 = 1. Rest is already calculated
b3 = b3 - dc/db3 => b3 = b3 - a3_delta
```

Hidden layers:

In hidden layers, the partial derivative of cost function w.r.t hidden layers will also follow a chain rule, since cost function is a function of outer layer.

```
z2 = a1w2 + b2
a2 = sigmoid(z2)

dc     dc    da2   dz2
---  = --- . --- . ---
dw2    da2   dz2   dw2

dz2/dw2 = a1
da2/dz2 = sigmoid_derv(z2)

dc     dc    da3   dz3
---  = --- . --- . --- => dc/da2 = a3_delta.w3
da2    da3   dz3   da2

w2 = w2 - dc/dw2
and set a2_delta = dc/da2 . da2/dz2

dc     dc    da2   dz2
---  = --- . --- . ---
db2    da2   dz2   db2

dz2/db2 = 1
b2 = b2 - dc/db2 => b2 = b2 - a2_delta
```

Similarly for derivative of cost function w.r.t w1

```
z1 = x.w1 + b1
a1 = sigmoid(z1)
c  = a1.w2 + b2


dc     dc    da1   dz1
---  = --- . --- . ---
dw1    da1   dz1   dw1


dz1/dw1 = x
da1/dz1 = sigmoid_derv(z1)


dc     dc    da2   dz2
---  = --- . --- . --- => dc/da1 = a2_delta.w2
da1    da2   dz2   da1


w1 = w1 - dc/dw1
and set a1_delta = dc/da1 . da1/dz1


dc     dc    da1   dz1
---  = --- . --- . ---
db1    da1   dz1   db1


dz1/db1 = 1
b1 = b1 - dc/db1 => b1 = b1 - a1_delta
```

Collecting all the above equations and laying it down in a single shot will provide a better overview. Comparing it with forward pass makes sense intuitively too.

```
Feed forward equations:
z1 = x.w1+b1
a1 = sigmoid(z1)


z2 = a1.w2+b2
a2 = sigmoid(z2)


z3 = a2.w3+b3
a3 = softmax(z3)


Back propagation equations:


There is no z3_delta and softmax_derv(a3), as explained before.
a3_delta = a3-y
```

```
z2_delta = a3_delta.w3.T
a2_delta = z2_delta.sigmoid_derv(a2)

z1_delta = a2_delta.w2.T
a1_delta = z1_delta.sigmoid_derv(a1)
```

Note: The derivative of sigmoid function can be implemented in two ways, depending upon the input. There is always a confusion with this. If the input is already an output of sigmoid i.e, `a`, then

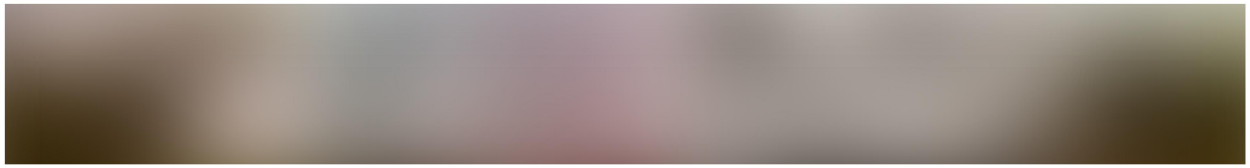
```
def sigmoid_derv(x):
    return x * (1 - x)
```

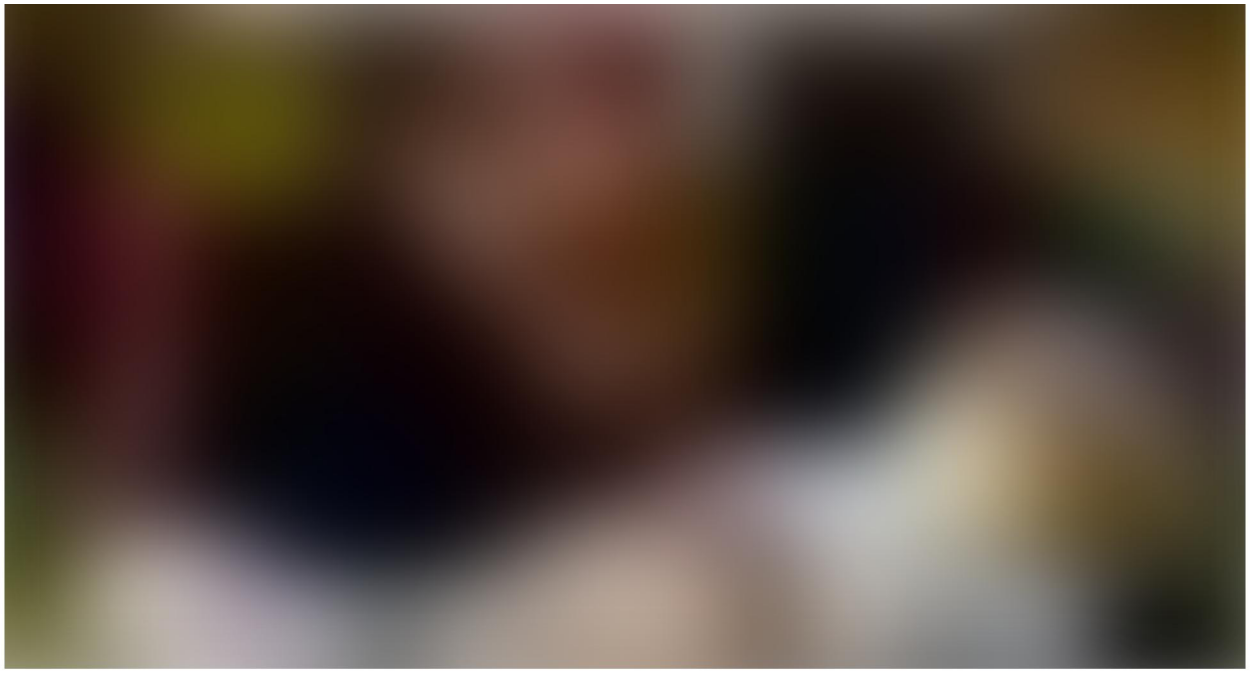
If the input is `z` and has not gone through softmax activation, then

```
def sigmoid_derv(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

In chain rule calculation, sigmoid derivative of `z` is used. But in implementation, sigmoid derivative of `a` is used. So, the former equation is used. Hope, that is clear enough. Now with numpy implementation.

```
def sigmoid_derv(s):
    return s * (1 - s)


def cross_entropy(pred, real):
    n_samples = real.shape[0]
    res = pred - real
    return res/n_samples


def error(pred, real):
    n_samples = real.shape[0]
    logp = - np.log(pred[np.arange(n_samples), real.argmax(axis=1)])
    loss = np.sum(logp)/n_samples
    return loss


def backprop(self):
    loss = error(self.a3, self.y)
    print('Error :', loss)
    a3_delta = cross_entropy(self.a3, self.y) # w3
    z2_delta = np.dot(a3_delta, self.w3.T)
    a2_delta = z2_delta * sigmoid_derv(self.a2) # w2
    z1_delta = np.dot(a2_delta, self.w2.T)
    a1_delta = z1_delta * sigmoid_derv(self.a1) # w1


    self.w3 -= self.lr * np.dot(self.a2.T, a3_delta)
    self.b3 -= self.lr * np.sum(a3_delta, axis=0, keepdims=True)
    self.w2 -= self.lr * np.dot(self.a1.T, a2_delta)
    self.b2 -= self.lr * np.sum(a2_delta, axis=0)
    self.w1 -= self.lr * np.dot(self.x.T, a1_delta)
    self.b1 -= self.lr * np.sum(a1_delta, axis=0)
```





Source: neuralnetmemes, picbear

Prediction phase:

Once the model is trained enough, prediction is straight forward. The query input has to be passed to the feed forward network and predict the output.

```
def predict(self, data):
    self.x = data
    self.feedforward()
    return self.a3.argmax()
```

Putting it altogether,

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

dig = load_digits()
onehot_target = pd.get_dummies(dig.target)
x_train, x_val, y_train, y_val = train_test_split(dig.data, onehot_target, test_size=0.1, random

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

def sigmoid_derv(s):
    return s * (1 - s)

def softmax(s):
    exps = np.exp(s - np.max(s, axis=1, keepdims=True))
    return exps/np.sum(exps, axis=1, keepdims=True)

def cross_entropy(pred, real):
    n_samples = real.shape[0]
    res = pred - real
    return res/n_samples

def error(pred, real):
    n_samples = real.shape[0]
    logp = - np.log(pred[np.arange(n_samples), real.argmax(axis=1)])
    loss = np.sum(logp)/n_samples
    return loss

```

```
class MyNN:
    def __init__(self, x, y):
        self.x = x
        neurons = 128
        self.lr = 0.5
        ip_dim = x.shape[1]
        op_dim = y.shape[1]

        self.w1 = np.random.randn(ip_dim, neurons)
        self.b1 = np.zeros((1, neurons))
        self.w2 = np.random.randn(neurons, neurons)
        self.b2 = np.zeros((1, neurons))
        self.w3 = np.random.randn(neurons, op_dim)
        self.b3 = np.zeros((1, op_dim))
        self.y = y

    def feedforward(self):
        z1 = np.dot(self.x, self.w1) + self.b1
        self.a1 = sigmoid(z1)
        z2 = np.dot(self.a1, self.w2) + self.b2
        self.a2 = sigmoid(z2)
        z3 = np.dot(self.a2, self.w3) + self.b3
        self.a3 = softmax(z3)

    def backprop(self):
        loss = error(self.a3, self.y)
        print('Error :', loss)
        a3_delta = cross_entropy(self.a3, self.y) # w3
        z2_delta = np.dot(a3_delta, self.w3.T)
        a2_delta = z2_delta * sigmoid_derv(self.a2) # w2
        z1_delta = np.dot(a2_delta, self.w2.T)
        a1_delta = z1_delta * sigmoid_derv(self.a1) # w1

        self.w3 -= self.lr * np.dot(self.a2.T, a3_delta)
```

```
            self.b3 -= self.lr * np.sum(a3_delta, axis=0, keepdims=True)
            self.w2 -= self.lr * np.dot(self.a1.T, a2_delta)
            self.b2 -= self.lr * np.sum(a2_delta, axis=0)
            self.w1 -= self.lr * np.dot(self.x.T, a1_delta)
            self.b1 -= self.lr * np.sum(a1_delta, axis=0)

        def predict(self, data):
            self.x = data
            self.feedforward()
            return self.a3.argmax()

    model = MyNN(x_train/16.0, np.array(y_train))

    epochs = 1500
    for x in range(epochs):
        model.feedforward()
        model.backprop()

    def get_acc(x, y):
        acc = 0
        for xx,yy in zip(x, y):
            s = model.predict(xx)
            if s == np.argmax(yy):
                acc +=1
        return acc/len(x)*100

    print("Training accuracy : ", get_acc(x_train/16, np.array(y_train)))
    print("Test accuracy : ", get_acc(x_val/16, np.array(y_val)))
```

nn_numpy.py hosted with ♡ by GitHub view raw

ANN using numpy

Final thoughts

- Tweak parameters such as learning rate, epochs, initial weights, activation functions to see how the system reacts
- If your model is not working, i.e, error gets sky-rocketed or weights are all NaNs or simply it predicts all inputs to a same category:
  - Check each function if they are normalized
  - Train with a single class and see how it works
  - Check the dimensions of matrices and their transposes
  - Verify the type of product used, dot product or hadamard product

. . .

Thank you for reading the post. Give some claps, if you liked it. If you spotted an error or having doubts, please let me know in the comments.

Feel free to contact me via Github, Twitter and Linkedin. Cheers!.

Machine Learning Neural Networks Data Science Keras Numpy