**Neural Networks from scratch using python and numpy**

Description

The purpose of this lab is to develop a NN from scratch using just python. It will be a very simple model, however the learning will transition over to a scaled up version of this as well. Creating a model from scratch will teach you the fundamental principles about how neural networks work on the inside. You will also find that the core concept is fairly simple and the power comes from the mass scaling.

We will attempt to create a NN processing engine from scratch in about 100 lines of code. This engine will be able to define models in a modular fashion, and have support for forward and backward passes. This includes back propagation using gradient descent. No library aside from numpy will be used.

Background (easily found on google)

* Derivative calculus (basic)
* How NN work on the inside
* Forward propagation
* Gradient descent and backward propagation
* Matrix multiplication

Basic principles

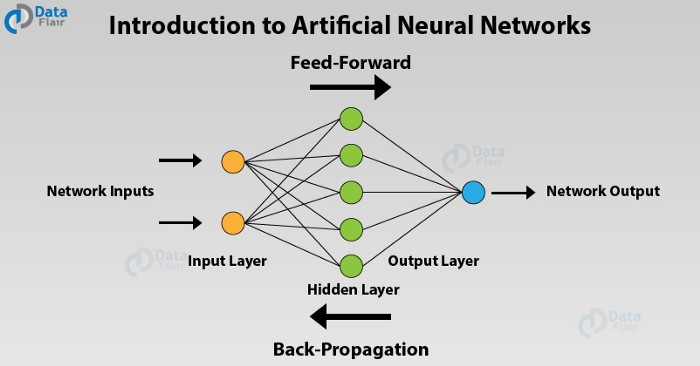


image Source: Data Flair

We will build a similar NN model with 2 inputs and 4 hidden neurons and one final output neuron.

Part 1: Forward pass through single layer

For this part we need to define the model layers and incorporate weights and biases. We will do this in a modular fashion so that we can stack many layers (if we wanted). We will define a layer class, this will hold one single layer for our neural network, along with support functions for forward propagation.

|  |
| --- |
| import numpy as np import math import logging   class **NNLayer**:  def **\_\_init\_\_**(self,inputs,outputs):  #Initialize the weights based on the inputs and outputs dimensions  self.inputs = inputs  self.outputs = outputs  self.weights = np.random.rand(inputs,outputs)    def **forward**(self,x):  if x.shape[0] != self.inputs:  logging.warn("Dimensional mismatch")  return  return self.sigmoid(np.dot(x,self.weights))    def **sigmoid**(self,x):  '''Create sigmoid activation function'''  return 1/(1+np.exp(-x)) |

**\_\_init\_\_() :** Here we will initialize all the parameters for our layer. For us to propagate data forward we need to know what inputs are coming in (size) and also the size of the outputs. In one case for example we could be building the first layer of our network, which would have 2 inputs and 4 outputs. This would create a set of weights like follow for this layer:

array([[0.38137423, 0.12811791, 0.74957231, 0.03831145],

[0.82606829, 0.6338447 , 0.50923346, 0.15618227]])

We have 2 inputs and 4 neurons, this would mean we have a total of 8 connections. We then need to take the dot product of the inputs to this array to get the output on layer 1.

**forward():** Here we will take the input, check whether it is appropriate and perform the dot product to compute the output. BUT let’s not forget that we need our activation function, which in our case is sigmoid.

Part 2: Forward pass through entire model

Now that we have a method to define each individual layer, we will build a model on top of this. Our main goal will be to concat multiple layers together and define functions that let us propagate data through all of them.

--------------------------------------------------------------------------------------

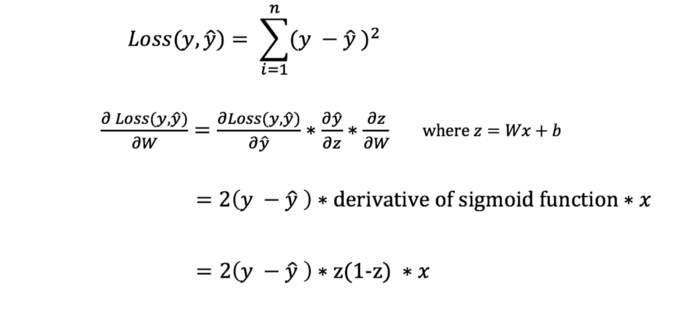
import numpy as np  
import math  
import logging  
   
class **NNLayer**:  
 def **\_\_init\_\_**(self,inputs,outputs):  
 #Initialize the weights based on the inputs and outputs dimensions  
 self.inputs = inputs  
 self.outputs = outputs  
 self.weights = np.random.rand(inputs,outputs)  
   
 def **forward**(self,x):  
 if x.shape[0] != self.inputs:  
 logging.warn("Dimensional mismatch")  
 return  
 return self.sigmoid(np.dot(x,self.weights))  
   
 def **sigmoid**(self,x):  
 '''Create sigmoid activation function'''  
 return 1/(1+np.exp(-x))

|  |
| --- |
| **class createModel:  def \_\_init\_\_(self,layers):  '''Given a list of layers define the model [2,4,2] in our case'''  self.model = self.buildModel(layers)    def buildModel(self,layers):  '''Given a list of layers define the model [2,4,2] in our case'''  model = np.array([])  for i in range(len(layers)-1):  l = NNLayer(layers[i],layers[i+1])  model = np.append(model,l)  return model    def forward(self,x):  '''Forward pass through all the layers.'''  for layer in self.model:  x = layer.forward(x)  return x**  **model = createModel([2,4,2])**  **model.forward(np.array([1,1]))**  -------------------------------------------------------------------------------------- |

In our last two lines we can see how nicely we’ve extrapolated the results, we simply define the entire model based on one single array. Our class will then go ahead and initialize the model and give us one simple forward function to propagate data through.

Part 3: Loss calculation

After we have our model defined we will need to define a loss function. To keep this modular we need to create our code in a way that the loss function can be easily modified and edited, so what we will do is pass a loss function to our model class. This will be done during initialization and can be referenced back when we run something like model.loss(). For our specific example we will use a fairly simple loss function called “Mean squared error”, the following formulas will help us define the loss function as well as a method to compute the gradient descent which we will need later on.



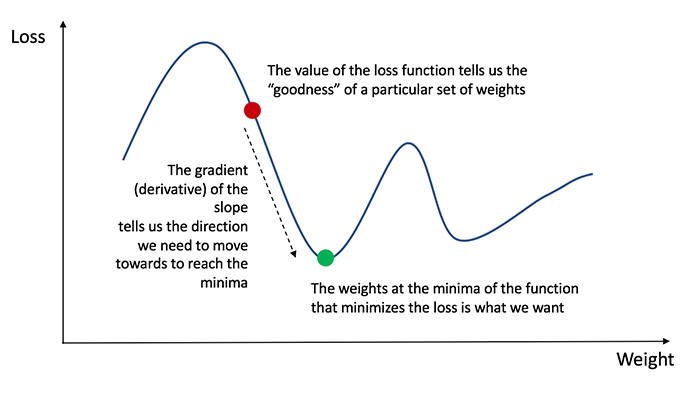
def lossFunc(self,y,output):

|  |
| --- |
| '''Forward pass through all the layers.'''  if y.shape[0] != output.shape[0]:  logging.warn("Dimensional mismatch")  return  self.loss = ((y-output)\*\*2).sum() |

Now we have our full forward pass finished. We can propagate data forward, compute the loss and define the model in a very very modular manner. The next thing we want to do is define everything needed for backprop.

Part 4: Gradient update through neurons and weights

Arguably this is the most important and powerful aspect of a neural network, it can also be the most complex. We will implement stochastic gradient descent which essentially means we will not be shuffling our data in as batches, but one input at a time. This is perfectly fine as we don’t require this for our toy NN engine. Gradient descent is a fairly simple concept, we have a function (our neural network) and we will attempt to find the global minimum in this NN+loss function.



**Our Goal:** Compute the derivative of the weights in each layer with respect to the output loss. We will then change the weights using this derivative, when we do this iteratively our weights will move to a position that trends toward a better function.

**How?:** Since we cannot simply compute the derivative of the entire neural network we will use the chain rule to propagate the loss backward through the model. To understand more about how backprop and chain rule works in the domain of neural networks see the following link. <https://cs231n.github.io/optimization-2/>

The only new function we need to add in out layer is the following:

|  |
| --- |
| def **back\_prop**(self,back\_x):  '''Propogate the loss backwards using chain rule and return variable.'''  self.back\_x = self.\_sigmoid\_derivative(self.output)\*back\_x   self.d\_weights = np.dot(self.input.reshape(-1,1),self.back\_x.reshape(-1,1).T)  self.back\_x = np.dot(self.d\_weights.reshape(-1,1),self.back\_x.reshape(-1,1).T).T[0]  return self.back\_x |

Explained: To this function we feed the backwards value being propagated, aka the derivative of the term after it with respect to the final loss. We then use this to compute the derivative of the weights of the current layer w.t.r (with respect to) the loss. We also compute a new back prop value so that the layer behind us can use it to compute the it’s own derivative.

The entire process will look like this:

|  |
| --- |
| #define model layer1 = NNLayer(2,4) layer2 = NNLayer(4,1)  #define input data input = np.array([1,1]) target = 1  def **train\_loop**():  #propagate forward  x = layer1.forward(input)  x = layer2.forward(x)    #compute the loss  loss = (target-x)\*\*2  print("output:",x,"target:",target,"loss:",loss)    #gradient descent and back prop  back\_x = 2\*(target-x)  back\_x = layer2.back\_prop(back\_x)  back\_x = layer1.back\_prop(back\_x)   #update weights  layer2.weights += layer2.d\_weights  layer1.weights += layer1.d\_weights |

And it works! If we call train\_loop about 10 times the loss will slowly drop down and trend to zero. All while the value at the output trends to the target value (1 in our example).

Part 5: Training loop

To make the final code as clean as possible, we will wrap our back prop into our createModel class as well. This will allow us to define, train and run a NN model in very few lines of code.

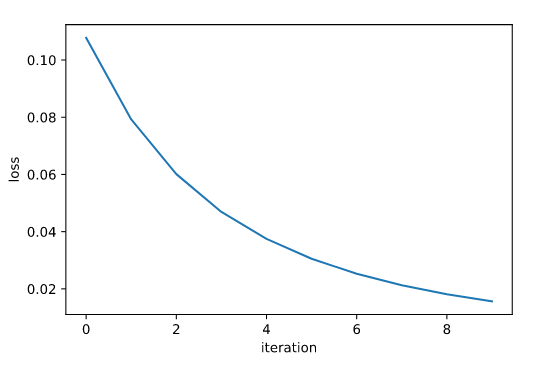
Our final loop becomes very very simple and we only need 4 lines to initialize the model and run a forward and backward pass:

|  |
| --- |
| model = createModel([2,3,1]) output = model.forward(input) loss = model.compute\_loss(target,output) model.backward() |

In a training loop this would look like the following:

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  #define input data  input = np.array([1,1])  target = 1  loss\_ = []    model = createModel([2,3,1])  for i in range(10):  output = model.forward(input)  loss = model.compute\_loss(target,output)  model.backward()  loss\_.append(loss)  print("output:",output,"loss:",loss)  # plot\_graph(loss\_) |

If we uncomment the last line then we can visualize the results of the training:



**Total Code:**

|  |
| --- |
| import numpy as np import logging  class **NNLayer**:  def **\_\_init\_\_**(self,inputs,outputs):  #Initialize the weights based on the inputs and outputs dimensions  self.input\_dim = inputs  self.outputs\_dim = outputs  self.weights = np.random.rand(inputs,outputs)    def **forward**(self,x):  self.input = x  if x.shape[0] != self.input\_dim:  logging.warn("Dimensional mismatch")  return  self.output = self.\_sigmoid(np.dot(x,self.weights))  return self.output    def **\_sigmoid**(self,x):  '''Create sigmoid activation function'''  return 1/(1+np.exp(-x))    def **\_sigmoid\_derivative**(self,x):  '''Derivative of sigmoid activation'''  z = self.\_sigmoid(x)  return z\*(1-z)    def **\_transpose\_vector**(self,x):  if len(x.shape) == 1:  input = x.reshape(-1,1)  else:  input = x.T  return input    def **back\_prop**(self,back\_x):  '''Propogate the loss backwards using chain rule and return variable.'''  self.back\_x = self.\_sigmoid\_derivative(self.output)\*back\_x   self.d\_weights = np.dot(self.input.reshape(-1,1),self.back\_x.reshape(-1,1).T)  self.back\_x = np.dot(self.d\_weights.reshape(-1,1),self.back\_x.reshape(-1,1).T).T[0]  return self.back\_x   class **createModel**:  def **\_\_init\_\_**(self,layers):  '''Given a list of layers define the model [2,4,2] in our case'''  self.model = self.build\_model(layers)    def **build\_model**(self,layers):  '''Given a list of layers define the model [2,4,2] in our case'''  model = np.array([])  for i in range(len(layers)-1):  l = NNLayer(layers[i],layers[i+1])  model = np.append(model,l)  return model    def **forward**(self,x):  '''Forward pass through all the layers.'''  for layer in self.model:  x = layer.forward(x)  self.output = x  return self.output   def **backward**(self):  try:  back\_x = self.loss\_backward  for layer in reversed(self.model):  back\_x = layer.back\_prop(back\_x)  layer.weights += layer.d\_weights  except:  logging.warning("loss may not yet be defined.")  return    def **\_sigmoid**(self,x):  '''Create sigmoid activation function'''  return 1/(1+np.exp(-x))    def **\_sigmoid\_derivative**(self,x):  '''Derivative of sigmoid activation'''  z = self.\_sigmoid(x)  return z\*(1-z)    def **compute\_loss**(self,y,output):  '''Forward pass through all the layers.'''  self.loss = ((y-output)\*\*2).sum()  #needed for backprop  self.loss\_backward = 2\*(y-output)  return self.loss  def **detailed\_train\_loop**():  #define model  layer1 = NNLayer(2,4)  layer2 = NNLayer(4,1)   #define input data  input = np.array([1,1])  target = 1   for i in range(10):  #propogate forward  x = layer1.forward(input)  x = layer2.forward(x)    #compute the loss  loss = (target-x)\*\*2  print("output:",x,"target:",target,"loss:",loss)    #gradient decent and back prop  back\_x = 2\*(target-x)  back\_x = layer2.back\_prop(back\_x)  back\_x = layer1.back\_prop(back\_x)   #update weights  layer2.weights += layer2.d\_weights  layer1.weights += layer1.d\_weights  def **plot\_graph**(data):  import matplotlib.pyplot as plt  plt.plot(data) # plotting by columns  plt.xlabel("iteration")  plt.ylabel("loss")  plt.show()   if \_\_name\_\_ == '\_\_main\_\_':  #define input data  input = np.array([1,1])  target = 1  loss\_ = []    model = createModel([2,3,1])  for i in range(10):  output = model.forward(input)  loss = model.compute\_loss(target,output)  model.backward()  loss\_.append(loss)  print("output:",output,"loss:",loss)    # plot\_graph(loss\_) |