



Artificial intelligence

Article

Neural networks from scratch

Learn the fundamentals of how you can build neural networks without the help of the frameworks that might make it easier to use

☆ Save 👍 Like

By Casper Hansen

Updated March 18, 2023 | Published March 18, 2020



Site feedback

- APIs
- Articles
- Blogs
- Courses
- Data Sets
- Learning Paths
- Open Projects
- Patterns
- Series
- Tutorials
- Videos

This content also appears in:

Related Topics

Creating complex [neural networks](#) with different architectures in Python should be a

About cookies on this site

Our websites require some cookies to function properly (required). In addition, other cookies may be used with your consent to analyze site usage, improve the user experience and for advertising.

For more information, please review your [cookie preferences](#) options. By visiting our website, you agree to our processing of information as described in IBM's [privacy statement](#).

To provide a smooth navigation, your cookie preferences will be shared across the IBM web domains listed [here](#).

Accept all

Legend ⓘ

Required only

Data science

Machine Learning

Deep learning

PyTorch

notebook on [GitHub](#) and run the code at the same time.

Prerequisites

In this article, I explain how to make a basic deep neural network by implementing the forward and backward pass (backpropagation). This requires some specific knowledge about the functions of neural networks.

It's also important to know the fundamentals of linear algebra to be able to understand why I perform certain operations in this article. My best recommendation is to watch 3Blue1Brown's series [Essence of linear algebra](#) .

NumPy

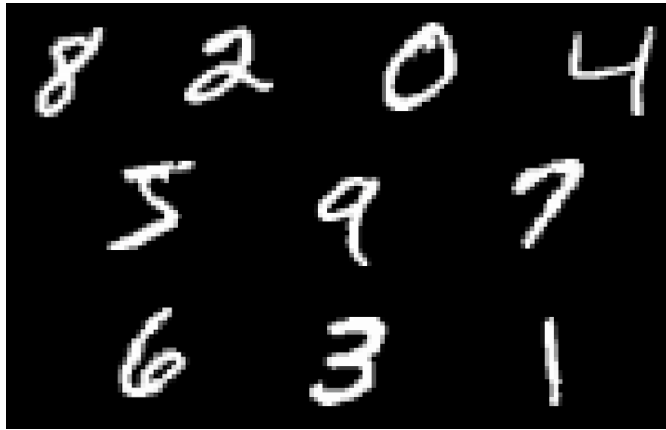
In this article, I build a *basic* deep neural network with 4 layers: 1 input layer, 2 hidden layers, and 1 output layer. All of the layers are fully connected. I'm trying to classify digits from 0 - 9 using a data set called [MNIST](#) . This data set consists of 70,000 images that are 28 by 28 pixels each. The data set contains one label for each image that specifies the digit that I see in each image. I say that there are 10 classes because I have 10 labels.

Interested in generative AI?

[Learn generative AI skills](#) →

Table of Contents ↓

Resources ↓



10 examples of the digits from the MNIST data set, scaled up 2x

For training the neural network, I use stochastic gradient descent, which means I put one image through the neural network at a time.

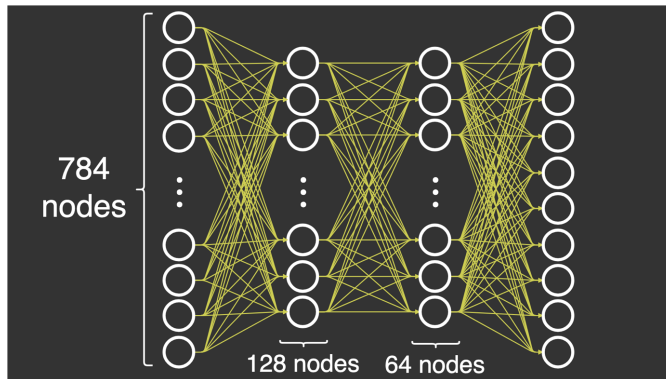
Let's try to define the layers in an exact way. To be able to classify digits, you must end up with the probabilities of an image belonging to a certain class after running the neural network because then you can quantify how well your neural network performed.

1. Input layer: In this layer, I input my data set consisting of 28x28 images. I *flatten* these images into one array with $28 \times 28 = 784$ elements. This means that the input layer will have 784 nodes.
2. Hidden layer 1: In this layer, I reduce the number of nodes from 784 in the input layer to 128 nodes. This creates a

challenge when you are going forward in the neural network (I'll explain this later).


3. Hidden layer 2: In this layer, I decide to go with 64 nodes, from the 128 nodes in the first hidden layer. This is no new challenge because I've already reduced the number in the first layer.
4. Output layer: In this layer, I reduce the 64 nodes to a total of 10 nodes so that I can evaluate the nodes against the label. This label is received in the form of an array with 10 elements, where one of the elements is 1 while the rest are 0.


You probably realize that the number of nodes in each layer decreases from 784 nodes to 128 nodes to 64 nodes to 10 nodes. This is based on [empirical observations](#) that this yields better results because we're not overfitting nor underfitting, only trying to get just the right number of nodes. The specific number of nodes chosen for this article were chosen at random, although decreasing to avoid overfitting. In most real-life scenarios, you would want to optimize these parameters by brute force or good guesses, usually by grid search or random search, but this is outside the scope of this article.



Imports and data set

For the entire NumPy part, I specifically wanted to share the imports used. Note that I use libraries other than NumPy to more easily load the data set, but they are not used for any of the actual neural networks.

```
from sklearn.datasets import fetch_o   
from keras.utils.np_utils import to_  
import numpy as np  
from sklearn.model_selection import t  
import time
```

Show more 

Now, I must load the data set and preprocess it so that I can use it in NumPy. I do normalization by dividing all images by 255 and make it such that all images have values between 0 - 1 because this removes some of the numerical stability issues with activation functions later on. I use one-hot encoded labels because I can more easily subtract these labels from the output of the neural network. I also choose to load the inputs as flattened arrays of $28 * 28 = 784$ elements

because that is what the input layer requires.

```
x, y = fetch_openml('mnist_784', ver 1)
x = (x/255).astype('float32')
y = to_categorical(y)
```

```
x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=0.1)
```

Show more ▾

Initialization


The initialization of weights in the neural network is a little more difficult to think about. To really understand how and why the following approach works, you need a grasp of linear algebra, specifically dimensionality when using the dot product operation.


The specific problem that arises when trying to implement the feedforward neural network is that we are trying to transform from 784 nodes to 10 nodes. When instantiating the `DeepNeuralNetwork` class, I pass in an array of sizes that defines the number of activations for each layer.

```
dnn = DeepNeuralNetwork(sizes=[784, 10])
```

Show more ▾

This initializes the `DeepNeuralNetwork` class by the `init` function.

```
def __init__(self, sizes, epochs=10,   
    self.sizes = sizes  
    self.epochs = epochs  
    self.l_rate = l_rate  
  
    # we save all parameters in the n  
    self.params = self.initialization
```

Show more 

Let's look at how the sizes affect the parameters of the neural network when calling the `initialization()` function. I am preparing $m \times n$ matrices that are "dot-able" so that I can do a forward pass, while shrinking the number of activations as the layers increase. I can only use the dot product operation for two matrices M1 and M2, where m in M1 is equal to n in M2, or where n in M1 is equal to m in M2.

With this explanation, you can see that I initialize the first set of weights W1 with $m=128$ and $n=784$, while the next weights W2 are $m=64$ and $n=128$. The number of activations in the input layer A0 is equal to 784, as explained earlier, and when I dot W1 by the activations A0, the operation is successful.

```
def initialization(self):  
    # number of nodes in each layer  
    input_layer=self.sizes[0]  
    hidden_1=self.sizes[1]  
    hidden_2=self.sizes[2]  
    output_layer=self.sizes[3]  
  
    params = {  
        'W1':np.random.randn(hidden_1  
        'W2':np.random.randn(hidden_2  
        'W3':np.random.randn(output_1  
    }  
  
    return params
```



Show more ▾

Feedforward

The forward pass consists of the dot operation in NumPy, which turns out to be just matrix multiplication. I must multiply the weights by the activations of the previous layer. Then, I must apply the activation function to the outcome.

To get through each layer, I sequentially apply the dot operation followed by the sigmoid activation function. In the last layer, I use the `softmax` activation function because I want to have probabilities of each class so that I can measure how well the current forward pass performs.

Note: I chose a numerically stable version of the `softmax` function. You can read more from the course at Stanford called [CS231n](#)

.


```
def forward_pass(self, x_train):  
    params = self.params  
  
    # input layer activations becomes  
    params['A0'] = x_train  
  
    # input layer to hidden layer 1  
    params['Z1'] = np.dot(params["W1"  
    params['A1'] = self.sigmoid(param  
  
    # hidden layer 1 to hidden layer  
    params['Z2'] = np.dot(params["W2"  
    params['A2'] = self.sigmoid(param  
  
    # hidden layer 2 to output layer  
    params['Z3'] = np.dot(params["W3"  
    params['A3'] = self.softmax(param  
  
    return params['A3']
```



Show more ▾

The following code shows the activation functions used for this article. As can be observed, I provide a derivative version of the sigmoid because I need that later on when backpropagating through the neural network.

```
def sigmoid(self, x, derivative=False):  
    if derivative:  
        return (np.exp(-x))/((np.exp(  
    return 1/(1 + np.exp(-x))  
  
def softmax(self, x):  
    # Numerically stable with large e  
    exs = np.exp(x - x.max())  
    return exs / np.sum(exs, axis=0)
```




Show more ▾

Backpropagation

The backward pass is hard to get right because there are so many sizes and

operations that must align for all of the operations to be successful. Here is the full function for the backward pass. I go through each weight update below.

```
def backward_pass(self, y_train, out   
    '''  
        This is the backpropagation a  
        of the neural network's param  
    ...  
    params = self.params  
    change_w = {}  
  
    # Calculate W3 update  
    error = output - y_train  
    change_w['W3'] = np.dot(error, pa  
  
    # Calculate W2 update  
    error = np.multiply( np.dot(param  
    change_w['W2'] = np.dot(error, pa  
  
    # Calculate W1 update  
    error = np.multiply( np.dot(param  
    change_w['W1'] = np.dot(error, pa  
  
    return change_w
```

Show more 

W3 update

The update for W_3 can be calculated by subtracting the ground truth array with labels called `y_train` from the output of the forward pass called `output`. This operation is successful because `len(y_train)` is 10 and `len(output)` is also 10. An example of `y_train` might be the following code, where the 1 is corresponding to the label of the output.

```
y_train = np.array([0, 0, 1, 0, 0, 0]
```

Show more ▾

An example of output is shown in the following code, where the numbers are probabilities corresponding to the classes of `y_train`.

```
output = np.array([0.2, 0.2, 0.5, 0.]
```

Show more ▾

If you subtract them, you get the following.

```
>>> output - y_train  
array([ 0.2,  0.2, -0.5,  0.3,  0.6,
```

Show more ▾

The next operation is the dot operation that dots the error (which I just calculated) with the activations of the last layer.

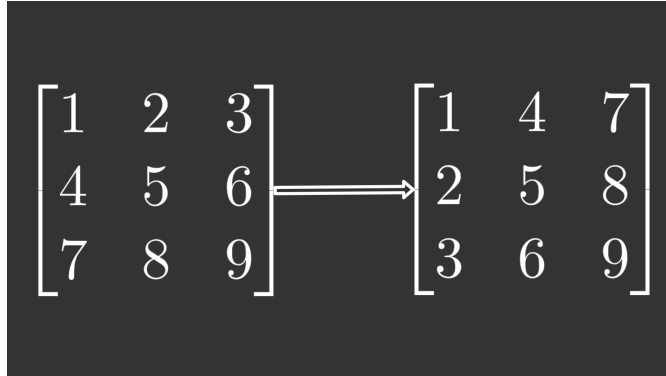
```
error = output - y_train  
change_w['W3'] = np.dot(error, param_
```

Show more ▾

W2 update

Next is updating the weights `w2`. More operations are involved for success. First, there is a slight mismatch in shapes because

w_3 has the shape $(10, 64)$ and error has $(10, 64)$, that is, the exact same dimensions. Therefore, I can use a [transpose operation](#) on the w_3 parameter by the `.T` such that the array has its dimensions permuted and the shapes now align up for the dot operation.


$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

An example of the transpose operation. Left: The original matrix. Right: The permuted matrix


w_3 now has shape $(64, 10)$ and error has shape $(10, 64)$, which are compatible with the dot operation. The result is [multiplied element-wise](#) (also called Hadamard product) with the outcome of the derivative of the sigmoid function of z_2 . Finally, I dot the error with the activations of the previous layer.


```
error = np.multiply( np.dot(params['change_w['W2']'] = np.dot(error, param
```

Show more \checkmark

W1 update

Likewise, the code for updating w_1 is using the parameters of the neural network one step earlier. Except for other parameters, the code is equivalent to the W_2 update.

```
error = np.multiply(np.dot(params['change_w['W1']] = np.dot(error, param
```

Show more 

Training (stochastic gradient descent)


I have defined a forward and backward pass, but how can I start using them? I must make a training loop and use stochastic gradient descent (SGD) as the optimizer to update the parameters of the neural network. There are two main loops in the training function. One loop for the number of epochs, which is the number of times I run through the entire data set, and a second loop for running through each observation one by one.

For each observation, I do a forward pass with x , which is one image in an array with the length 784, as explained earlier. The output of the forward pass is used along with y , which are the one-hot encoded labels (the ground truth) in the backward pass. This gives me a dictionary of updates to the

weights in the neural network.

```
def train(self, x_train, y_train, x_
    start_time = time.time()
    for iteration in range(self.epoch
        for x,y in zip(x_train, y_tra
            output = self.forward_pas
            changes_to_w = self.backw
            self.update_network_param

        accuracy = self.compute_accur
        print('Epoch: {0}, Time Spent
              iteration+1, time.time()
        ))
```

Show more 

The `update_network_parameters()` function has the code for the SGD update rule, which just needs the gradients for the weights as input. And to be clear, SGD involves calculating the gradient using backpropagation from the backward pass, not just updating the parameters. They seem separate, and they should be thought of separately because the two algorithms are different.

```
def update_network_parameters(self,
    """
    Update network parameters acc
    Stochastic Gradient Descent.


     $\theta = \theta - \eta * \nabla J(x, y)$ ,
        theta  $\theta$ :          a net
        eta  $\eta$ :              the l
        gradient  $\nabla J(x, y)$ : the g
                                i.e.
    """
    ...

    for key, value in changes_to_w.it
        for w_arr in self.params[key]
            w_arr -= self.l_rate * va
```

Show more 


After having updated the parameters of the neural network, I can measure the accuracy on a validation set that I prepared earlier to validate how well the network performs after each iteration over the whole data set.

The following code uses some of the same pieces as the training function. To start, it does a forward pass then finds the prediction of the network and checks for equality with the label. After that, I sum over the predictions and divide by 100 to find the accuracy. Next, I average out the accuracy of each class.

```
def compute_accuracy(self, x_val, y_val)   
    """  
    This function does a forward  
    of the maximum value in the o  
    y. Then it sums over each pre  
    """  
    predictions = []  
  
    for x, y in zip(x_val, y_val):  
        output = self.forward_pass(x)  
        pred = np.argmax(output)  
        predictions.append(pred == y)  
  
    summed = sum(pred for pred in pre  
    return np.average(summed)
```

Show more 


Finally, I can call the training function after knowing what will happen. I use the training and validation data as input to the training function, and then wait.

```
dnn.train(x_train, y_train, x_val, y_val) 
```

Show more 

Note that the results might vary a lot depending on how the weights are initialized. My results range from an accuracy of 0% - 95%.

Following is the full code for an overview of what's happening.

```
from sklearn.datasets import fetch_o   
from keras.utils.np_utils import to_c  
import numpy as np  
from sklearn.model_selection import t  
import time  
  
x, y = fetch_openml('mnist_784', vers  
x = (x/255).astype('float32')  
y = to_categorical(y)  
  
x_train, x_val, y_train, y_val = trai  
  
class DeepNeuralNetwork():  
    def __init__(self, sizes, epochs=  
        self.sizes = sizes  
        self.epochs = epochs  
        self.l_rate = l_rate  
  
        # we save all parameters in t  
        self.params = self.initializa  
  
    def sigmoid(self, x, derivative=F  
        if derivative:  
            return (np.exp(-x))/((np.  
            return 1/(1 + np.exp(-x))  
  
    def softmax(self, x):  
        # Numerically stable with lar  
        exps = np.exp(x - x.max())  
        return exps / np.sum(exps, ax  
  
    def initialization(self):  
        # number of nodes in each lay  
        input_layer=self.sizes[0]  
        hidden_1=self.sizes[1]  
        hidden_2=self.sizes[2]  
        output_layer=self.sizes[3]  
  
        params = {  
            'W1':np.random.randn(hidd  
            'W2':np.random.randn(hidd  
            'W3':np.random.randn(outp  
        }
```



```

        return params

    def forward_pass(self, x_train):
        params = self.params

        # input layer activations bec
        params['A0'] = x_train

        # input layer to hidden layer
        params['Z1'] = np.dot(params[
        params['A1'] = self.sigmoid(p

        # hidden layer 1 to hidden la
        params['Z2'] = np.dot(params[
        params['A2'] = self.sigmoid(p

        # hidden layer 2 to output la
        params['Z3'] = np.dot(params[
        params['A3'] = self.softmax(p

        return params['A3']

    def backward_pass(self, y_train,
        '''
        This is the backpropagati
        of the neural network's p

        Note: There is a stabilit
        caused by the dot

        RuntimeError: inv
        RuntimeError: ove
        RuntimeError: ove
        ...
        params = self.params
        change_w = {}

        # Calculate W3 update
        error = output - y_train
        change_w['W3'] = np.dot(error

        # Calculate W2 update
        error = np.multiply( np.dot(p
        change_w['W2'] = np.dot(error

        # Calculate W1 update
        error = np.multiply( np.dot(p
        change_w['W1'] = np.dot(error

        return change_w

    def update_network_parameters(sel
        '''
        Update network parameters
        Stochastic Gradient Desce

         $\theta = \theta - \eta * \nabla J(x, y),$ 
        theta  $\theta$ :          a
        eta  $\eta$ :              t

```

```

        gradient ∇J(x, y): t
            i
        ...

    for key, value in changes_to_w.items():
        for w_arr in self.params[key]:
            w_arr -= self.l_rate * value

def compute_accuracy(self, x_val, y_val):
    """
    This function does a forward pass
    of the maximum value in the output
    y. Then it sums over each prediction.
    """
    predictions = []

    for x, y in zip(x_val, y_val):
        output = self.forward_pass(x)
        pred = np.argmax(output)
        predictions.append(pred == y)

    summed = sum(predictions)
    return np.average(summed)

def train(self, x_train, y_train, x_val, y_val,
          start_time = time.time(),
          iterations = 1000):
    for iteration in range(iterations):
        for x,y in zip(x_train, y_train):
            output = self.forward_pass(x)
            changes_to_w = self.backward_pass(y, output)
            self.update_network_params(changes_to_w)

        accuracy = self.compute_accuracy(x_val, y_val)
        print('Epoch: {0}, Time Spent: {1}, Accuracy: {2}'.format(
            iteration+1, time.time() - start_time, accuracy))

dnn = DeepNeuralNetwork(sizes=[784, 10])
dnn.train(x_train, y_train, x_val, y_val)

```

Show more 

Good exercises in NumPy

You might have noticed that the code is very readable, but it takes up a lot of space and could be optimized to run in loops. Here is a chance to optimize and improve it. If you're

new to this topic, the difficulties of the following exercises are easy to hard, where the last exercise is the hardest.

1. Easy: Implement the ReLU activation function or any other activation function. Check how the sigmoid functions are implemented for reference, and remember to implement the derivative as well. Use the ReLU activation function in place of the sigmoid function.
2. Easy: Initialize biases and add them to Z before the activation function in the forward pass, and update them in the backward pass. Be careful of the dimensions of the arrays when you try to add biases.
3. Medium: Optimize the forward and backward pass such that they run in a `for` loop in each function. This makes the code easier to modify and possibly easier to maintain.
 - Optimize the initialization function that makes weights for the neural network such that you can modify the `sizes=[]` argument without the neural network failing.
4. Medium: Implement mini-batch gradient descent, replacing stochastic gradient descent. Instead of making an update to a parameter for each sample, make an

update based on the average value of the sum of the gradients accumulated from each sample in the mini-batch. The size of the mini-batch is usually below 64.

5. Hard: Implement the Adam optimizer. This should be implemented in the training function.
 1. Implement Momentum by adding the extra term
 2. Implement an adaptive learning rate, based on the AdaGrad optimizer
 3. Combine step 1 and 2 to implement Adam

My belief is that if you complete these exercises, you will have a good foundation. The next step is implementing convolutions, filters, and more, but that is left for a future article.

As a disclaimer, there are no solutions to these exercises.

PyTorch

Now that I've shown how to implement these calculations for the feedforward neural network with backpropagation, let's see how easy and how much time PyTorch saves us in comparison to NumPy.

Loading MNIST data set

One of the things that seems more complicated or harder to understand than it should be is loading data sets with PyTorch.


You start by defining the transformation of the data, specifying that it should be a tensor and that it should be normalized. Then, you use the `DataLoader` in combination with the data sets `import` to load a data set. This is all you need. You'll see how to unpack the values from these loaders later.

```
import torch
from torchvision import datasets, tra

transform = transforms.Compose([
    transforms.ToTensor()
    transforms.Normalize(
])

train_loader = torch.utils.data.DataL
datasets.MNIST('data', train=True

test_loader = torch.utils.data.DataLo
datasets.MNIST('data', train=Fals
```

Show more 

Training

I have defined a class called `Net` that is similar to the `DeepNeuralNetwork` class written in NumPy earlier. This class has some

of the same methods, but you can clearly see that I don't need to think about initializing the network parameters nor the backward pass in PyTorch because those functions are gone, along with the function for computing accuracy.

```
import time
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self, epochs=10):
        super(Net, self).__init__()
        self.linear1 = nn.Linear(784,
        self.linear2 = nn.Linear(128,
        self.linear3 = nn.Linear(64,

        self.epochs = epochs

    def forward_pass(self, x):
        x = self.linear1(x)
        x = torch.sigmoid(x)
        x = self.linear2(x)
        x = torch.sigmoid(x)
        x = self.linear3(x)
        x = torch.softmax(x, dim=0)
        return x

    def one_hot_encode(self, y):
        encoded = torch.zeros([10], d
        encoded[y[0]] = 1.
        return encoded

    def train(self, train_loader, opt
        start_time = time.time()
        loss = None

        for iteration in range(self.e
            for x,y in train_loader:
                y = self.one_hot_enco
                optimizer.zero_grad()
                output = self.forward
                loss = criterion(outp
                loss.backward()
                optimizer.step()

        print('Epoch: {0}, Time S
              iteration+1, time.tim
        ))
```

Show more 

When reading this class, notice that PyTorch has implemented all of the relevant activation functions for us, along with different types of layers. You don't even have to think about it. You can just define some layers like `nn.Linear()` for a fully connected layer.

I have imported optimizers earlier, and now I specify which optimizer I want to use, along with the criterion for the loss. I pass both the optimizer and criterion into the training function, and PyTorch starts running through the examples just like in NumPy. I could even include a metric for measuring accuracy, but that is left out in favor of measuring the loss instead.

```
model = Net()
optimizer = optim.SGD(model.parameters)
criterion = nn.BCEWithLogitsLoss()
model.train(train_loader, optimizer,
```



Show more ▾

TensorFlow 2.0 with Keras

For the TensorFlow/Keras version of the neural network, I chose to use a simple approach, minimizing the number of lines of code. That means I am not defining any class, but instead using the high-level API of Keras

to make a neural network with just a few lines of code. If you are just starting to learn about neural networks, you will find that the bar to entry is the lowest when using Keras.

Therefore, I recommend it.

I start by importing all of the functions I need for later.

```
import tensorflow as tf
from tensorflow.keras.datasets import
from tensorflow.keras.utils import to
from tensorflow.keras.layers import F
from tensorflow.keras.losses import B
```

Show more ▾

I can load the data set and preprocess it with just these few lines of code. Note that I only preprocess the training data because I'm not planning on using the validation data for this approach. Later, I explain how we can use the validation data.

```
(x_train, y_train), (x_val, y_val) =
x_train = x_train.astype('float32') /
y_train = to_categorical(y_train)
```

Show more ▾

The next step is defining the model. In Keras, this is extremely simple after you know which layers you want to apply to your data. In this case, I'm going for the fully connected layers, as in the NumPy example. In Keras, this is done by the `Dense()` function.

After I have defined the layers of the model, I compile the model and define the optimizer, loss function, and metric. Finally, I can tell Keras to fit to the training data for 10 epochs, just like in the other examples.

```
model = tf.keras.Sequential([  
    Flatten(input_shape=(28, 28)),  
    Dense(128, activation='sigmoid'),  
    Dense(64, activation='sigmoid'),  
    Dense(10)  
])  
  
model.compile(optimizer='SGD',  
              loss=BinaryCrossentropy,  
              metrics=['accuracy'])  
  
model.fit(x_train, y_train, epochs=10
```

Show more ▾

If you want to use the validation data, you could pass it in using the `validation_data` parameter of the fit function:

```
model.fit(x_train, y_train, epochs=1
```

Show more ▾

Conclusion

This article gave you the fundamentals of how you can build neural networks without the help of the frameworks that might make it easier to use. I built a *basic* deep neural network with 4 layers, and I explained how to make a basic deep neural network by implementing the forward and backward

pass (backpropagation).



Build
Smart ↓
Build
Secure ↑

IBM
Developer

About
FAQ
Third-party
notice

Follow Us

Twitter
LinkedIn
YouTube

Explore

Newsletters
Patterns
APIs
Articles
Tutorials
Open
source
projects
Videos
Events

Community

Career
Opportunities

Privacy

Terms of
use

Accessibility

Cookie
preferences

Sitemap