archive.today
webpage capture

Saved from https://levelup.gitconnected.com/create-your-own-artificial-neural-network-fo search
no other snapshots from this url
Redirected from https://medium.com/gitconnected/create-your-own-artificial-neural-network-fc
no other snapshots from this url
All snapshots from host levelup.gitconnected.com
from host medium.com

9 Aug 2023 05:00:39 UTC

Webpage Screenshot

share download .zip report bug or abuse Buy me a coffee

This is your **last free member-only story** this month. Sign up for Medium and get an extra one.

Member-only story

Create Your Own Artificial Neural Network for Multi-class Classification (With Python)

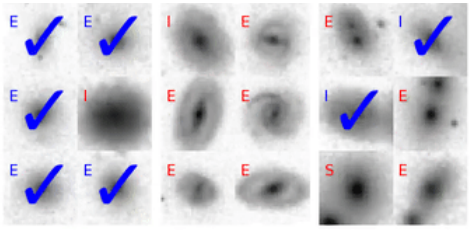
Philip Mocz · Follow
Published in Level Up Coding · 10 min read · Jul 23

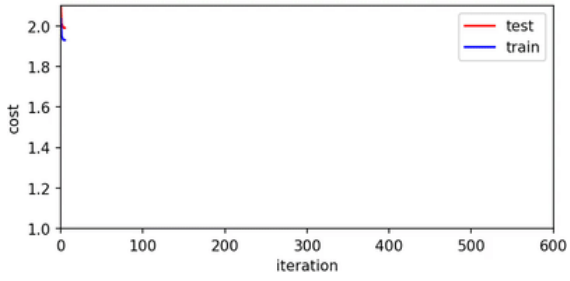
340 1

In today's recreational coding exercise, we will build an Artificial Neural Network from scratch and train it to classify galaxies. In machine learning, the multi-class classification problem is the problem of classifying objects into one of three or more classes. In this example, we will classify images of galaxies as Elliptical, Spiral, or Irregular.

You may find the accompanying artificial neural network Python code on GitHub.

Before we begin, below is a gif of what running our AI multi-class classification algorithm looks like:



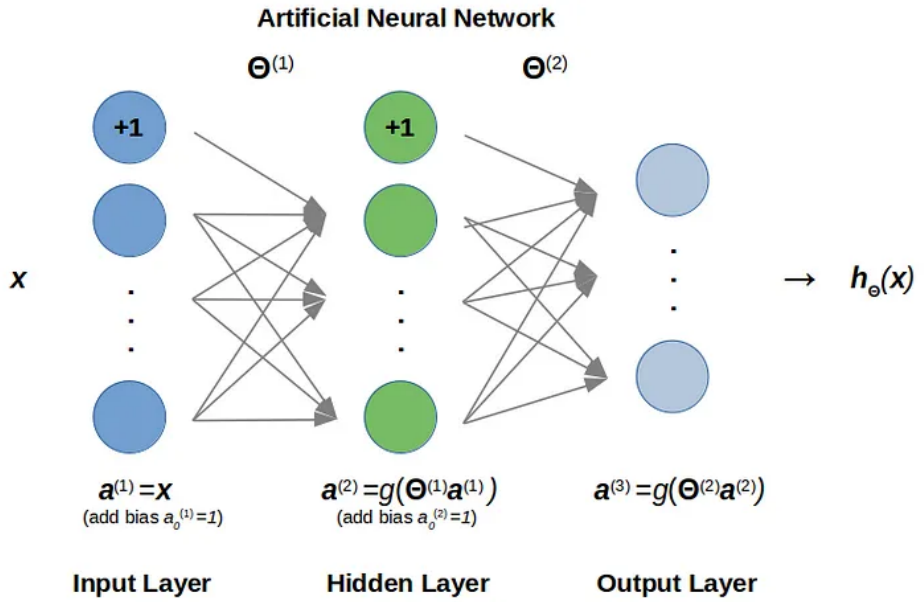


Artificial Neural Networks

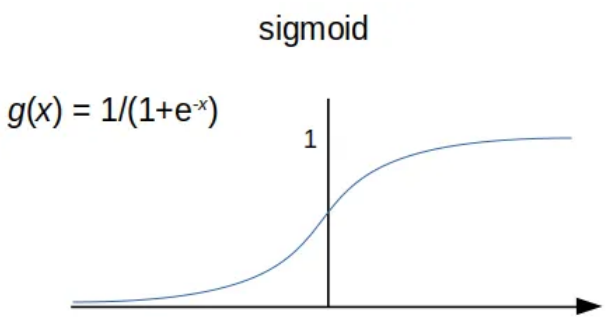
In AI/deep learning, an artificial neural network is a mathematical operator that takes an input vector x (**features**) and produces an output vector $h_\Theta(x)$ (**labels**), where Θ represents a collection of matrices (**weights**) whose values are trained on a dataset with known labels. The input x may be, for example, an image flattened as a vector, and the output could be an array of probabilities where the i-th entry is the probability that the input belongs to a class i.

The neural network consists of several layers. There is the **input layer**, one or more **hidden layers**, and the final **output layer**. Data is passed from one layer to the next with a matrix multiplication Θ^(i) followed by the application of a nonlinear activation function $g(x)$ that returns values between 0 and 1 (think of values that are close to 1 sort of like a firing neuron in the human brain). Additionally, a **bias** term of 1 is prepended to the vector output of each layer, which shifts the activation function towards the positive or negative side, as needed, when the model is trained.

A neural network may look something like the following schematic, which shows an example with three layers:



For our purposes, we will use a sigmoid activation function for $g(x)$:



implemented as:

```
def g(x):
  """ sigmoid function """
  return 1.0 / (1.0 + np.exp(-x))
```

sigmoid.py hosted with ❤ by GitHub view raw

It is important the activation function is *nonlinear* because it introduces nonlinearity into the neural network, which allows it to approximate complex representations and functions based on the inputs that are not possible with simple linear regression models. Other (nonlinear) choices for the activation function exist too, such as ReLU or SoftMax.

We will end up needing the derivative of the sigmoid later too (for training), so we implement it:

```
def grad_g(x):
  """ gradient of sigmoid function """
  gx = g(x)
  return gx * (1.0 - gx)
```

grad_sigmoid.py hosted with ❤ by GitHub view raw

We have also chosen to encode the output labels y (class ID numbers: 0=Elliptical, 1=Spiral, 2=Irregular, in our example problem) as follows. Given training data features x (grayscale images flattened as vectors in our case) the output labels y are vectors with a single '1' in the ith place if the image belongs to class i.

$$y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right\}$$

Feedforward and Model Prediction

Feedforward propagation is the computation of the model output from the input, given the set of weights Θ. We implement it for a 3-layer neural network as follows:

```python
def predict(Theta1, Theta2, X):
  """ Predict labels in a trained three layer classification network.
  Input:
    Theta1       trained weights applied to 1st layer (hidden_layer_size x input_layer_s
    Theta2       trained weights applied to 2nd layer (num_labels x hidden_layer_size+1)
    X            matrix of training data      (m x input_layer_size)
  Output:
    prediction   label prediction
  """

  m = np.shape(X)[0]                    # number of training values
  num_labels = np.shape(Theta2)[0]

  a1 = np.hstack((np.ones((m,1)), X))   # add bias (input layer)
  a2 = g(a1 @ Theta1.T)                 # apply sigmoid: input layer --> hidden layer
  a2 = np.hstack((np.ones((m,1)), a2))  # add bias (hidden layer)
  a3 = g(a2 @ Theta2.T)                 # apply sigmoid: hidden layer --> output layer

  prediction = np.argmax(a3,1).reshape((m,1))
  return prediction
```

predict.py hosted with ❤️ by GitHub view raw

The algorithm cycles between adding the bias term, applying matrix multiplication by the weights, and applying the sigmoid. Our function is implemented in a vectorized fashion, to avoid 'for loops', which can be slow in Python. It takes as input the weights Theta1 and Theta2 which transform the first two layers, and a matrix X of several feature (row) vectors x.

The prediction from the neural network returns for each input label an output $h_\Theta(x)$, which is a vector where each entry i is proportional to the probability the input belongs to class i. To return a single class for our prediction, we simply take the entry with the largest value.

Cost Function

We next define the **cost function,** $J(\Theta)$, that we wish to minimize in order to train the neural network. The cost function operates on the set of m training features x. For a multi-class classification problem, we can use the following equation:

$$
\begin{aligned}
J(\Theta) &= \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] \\
&+ \frac{\lambda}{2m}\sum_{l=1}^{L}\left[\sum_{j=1}^{J_l}\sum_{k=1}^{K_l}\left(\Theta_{j,k}^{(l)}\right)^2\right]
\end{aligned}
$$

where the first line introduces a penalty when the neural network incorrectly predicts the label (motivated by the likelihood function used in logistic regression), and the second line is a **regularization term**, with a scaling parameter λ, designed to add a penalty when the values of Θ are large. This term is meant to reduce the overall complexity of the model and help prevent overfitting.

We can implement the cost function as follows. First, we will need a helper function that flattens the Θ training weight matrices into a single vector, for convenience (as we'll see later, that Python's optimization functions take only a single vector as input).

```python
def reshape(theta, input_layer_size, hidden_layer_size, num_labels):
  """ reshape theta into Theta1 and Theta2, weights in our neural network """
  ncut = hidden_layer_size * (input_layer_size+1)
  Theta1 = theta[0:ncut].reshape(hidden_layer_size, input_layer_size+1)
  Theta2 = theta[ncut:].reshape(num_labels, hidden_layer_size+1)
  return Theta1, Theta2
```

reshape.py hosted with ❤ by GitHub view raw

Then, we code the cost function equation (again in a vectorized way for numerical efficiency). The code first calculates the predicted data labels using forward propagation, then calculates the cost function, including the regularization term.

```python
def cost_function(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lmbda):
  """ Neural net cost function for a two layer classification network.
  Input:
    theta               flattened vector of neural net model parameters
    input_layer_size    size of input layer
    hidden_layer_size   size of hidden layer
    num_labels          number of labels
    X                   matrix of training data
    y                   vector of training labels
    lmbda               regularization term
  Output:
    J                   cost function
  """

  # unflatten theta
  Theta1, Theta2 = reshape(theta, input_layer_size, hidden_layer_size, num_labels)

  # number of training values
  m = len(y)

  # Feedforward: calculate the cost function J:

  a1 = np.hstack((np.ones((m,1)), X))
  a2 = g(a1 @ Theta1.T)
  a2 = np.hstack((np.ones((m,1)), a2))
  a3 = g(a2 @ Theta2.T)

  y_mtx = 1.*(y==0)
  for k in range(1,num_labels):
        y_mtx = np.hstack((y_mtx, 1.*(y==k)))

  # cost function
  J = np.sum( -y_mtx * np.log(a3) - (1.0-y_mtx) * np.log(1.0-a3) ) / m

  # add regularization
  J += lmbda/(2.*m) * (np.sum(Theta1[:,1:]**2)  + np.sum(Theta2[:,1:]**2))

  return J
```

cost.py hosted with ❤ by GitHub view raw

We are getting close to having a working neural network model. So far, we have a way to take any image x and, given a set of weights Θ, return the predicted classification for the image (the weights are not yet trained at this point, so the predictions will be bad!). We can also evaluate the cost function of the training weights Θ, given all the training data. We now need to train the neural network, that is, find a good set of parameters Θ that minimizes the cost function. We would also like our model to be general enough to remain accurate for new unseen data. How do we do this? The next two sections will answer that.

Backpropagation

To perform the optimization procedure in the training of the neural network, we will also need to compute the gradient for the cost function. This is where **backpropagation** comes in, an efficient way to compute such gradients, $\Delta^{(l)}$ with respect to every element of each training weight matrix $\Theta^{(l)}$. Below I will describe the algorithm and implementation, but will not cover the derivation though will mention it comes from the chain rule for derivatives in calculus.

The backpropagation algorithm first computes a forward pass for each of the m input training data pairs (x_k, y_k). Then, for each layer l, starting from the last, it computes an error vector $\delta^{(l)}$ for the layer that goes into updating the computation of the gradient. As the errors are found, the gradients are updated as:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}\left(a^{(l)}\right)^T$$

Care has to be taken in the calculation to always remove the 0th entry of $\delta^{(l)}$ that is due to the bias term we have previously added into each layer.

For the third (output) layer in our model, the error is simply the difference between the predicted labels and the actual labels:

$$\delta^{(3)} = \mathbf{a}^{(3)} - \mathbf{y}$$

And for the second layer, it is:

$$\delta^{(2)} = \left(\mathbf{\Theta}^{(2)}\right)^T \delta^{(3)} * g'(\mathbf{\Theta}^{(1)} a^{(1)})$$

where * is the element-by-element multiplication operator.

We implement backpropagation as follows:

```python
def gradient(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lmbda):
  """ Neural net cost function gradient for a three layer classification network.
  Input:
    theta               flattened vector of neural net model parameters
    input_layer_size    size of input layer
    hidden_layer_size   size of hidden layer
    num_labels          number of labels
    X                   matrix of training data
    y                   vector of training labels
    lmbda               regularization term
  Output:
    grad                flattened vector of derivatives of the neural network
  """

  # unflatten theta
  Theta1, Theta2 = reshape(theta, input_layer_size, hidden_layer_size, num_labels)

  # number of training values
  m = len(y)

  # Backpropagation: calculate the gradients Theta1_grad and Theta2_grad:

  Delta1 = np.zeros((hidden_layer_size,input_layer_size+1))
  Delta2 = np.zeros((num_labels,hidden_layer_size+1))

  for t in range(m):

    # forward
    a1 = X[t,:].reshape((input_layer_size,1))
    a1 = np.vstack((1, a1))   #  +bias
    z2 = Theta1 @ a1
    a2 = g(z2)
    a2 = np.vstack((1, a2))   #  +bias
    a3 = g(Theta2 @ a2)

    # compute error for layer 3
    y_k = np.zeros((num_labels,1))
    y_k[y[t,0].astype(int)] = 1
    delta3 = a3 - y_k
    Delta2 += (delta3 @ a2.T)

    # compute error for layer 2
    delta2 = (Theta2[:,1:].T @ delta3) * grad_g(z2)
    Delta1 += (delta2 @ a1.T)

  Theta1_grad = Delta1 / m
  Theta2_grad = Delta2 / m

  # add regularization
  Theta1_grad[:,1:] += (lmbda/m) * Theta1[:,1:]
  Theta2_grad[:,1:] += (lmbda/m) * Theta2[:,1:]

  # flatten gradients
  grad = np.concatenate((Theta1_grad.flatten(), Theta2_grad.flatten()))

  return grad
```

gradient.py hosted with ❤️ by GitHub view raw

There are alternative ways to calculate the gradient. For example, one could use simple **finite difference.** This, however, scales inefficiently, because it has to be performed with respect to each of the training parameters (but the

method is simple and general and so it can be used to check your answers!). Another approach is to use **automatic differentiation**, which automatically creates a gradient function to any computer function by employing the chain rule to all the sequences of elementary arithmetic operations (e.g., addition, subtraction, multiplication, division). Libraries such as JAX provide easy ways to add automatic differentiation for any function in your code, and it is a very powerful and efficient method to calculate the gradient of any function.

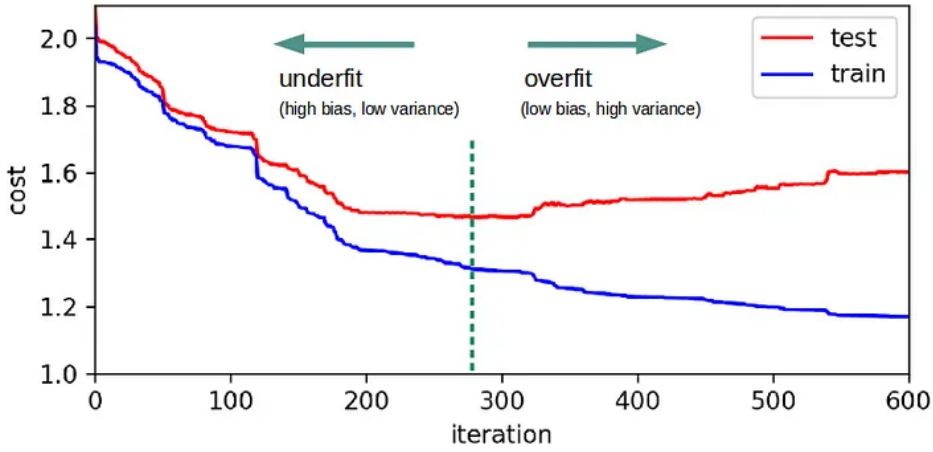
Training the Neural Network

With the gradient function in place, we can now train our neural network. To do so, we will use an advanced optimizer, the *conjugate gradient* algorithm — fmin_cg — that is built into the scipy library in Python. The method is a variant of gradient descent, except the search direction is set to always be orthogonal to the previous step. The gradient function is used to determine an adaptive step size as the algorithm searches for the minimum of the cost function.

To fit the neural network, we will employ a strategy to prevent **overfitting.** Overfitting is what happens when a machine learning model gives accurate predictions for training data but not for new data. To avoid it, we split the data into two sets:

- **training set**
- **test/validation set**

The training set will typically be comprised of ~80% of the data and will be fed into the optimizer to find a good fit for the model weights. However, the neural network is a complex function with a lot of parameters, and may eventually learn to correctly classify up to 100% of the data in a way that is not general to other datasets (note the regularization term we added to the cost function can prevent this to an extent by keeping weights small). What we do to prevent overfitting is we also evaluate the cost function on the test set. As iterations proceed in the optimization process, the cost of fitting the training set is expected to decrease. But at some point, if we have overfitted the model, the cost of the test set may start to increase! This is a good stopping point for our training. At that point, we can go off and apply our trained model to brand-new data. The fitting process may produce learning curves that conceptually look like the following:



In general, having more (good quality) data to train deep learning models improves the model's accuracy and generality because it sees enough variation to capture critical patterns in the data. Also, the more data you have, the more layers and nodes you can add to the neural network and still avoid overfitting.

We implement the training procedure below:

```python
def main():
  """ Artificial Neural Network for classifying galaxies """

  # set the random number generator seed
  np.random.seed(917)

  # Load the training and test datasets
  train = np.genfromtxt('train.csv', delimiter=',')
  test = np.genfromtxt('test.csv', delimiter=',')

  # get labels (0=Elliptical, 1=Spiral, 2=Irregular)
  train_label = train[:,0].reshape(len(train),1)
  test_label = test[:,0].reshape(len(test),1)

  # normalize image data to [0,1]
  train = train[:,1:] / 255.
  test = test[:,1:] / 255.

  # Construct our data matrix X (2700 x 5000)
  X = train

  # Construct our label vector y (2700 x 1)
  y = train_label

  # Three layer Neural Network parameters:
  m = np.shape(X)[0]
  input_layer_size = np.shape(X)[1]
  hidden_layer_size = 8
  num_labels = 3
  lmbda = 1.0    # regularization parameter

  # Initialize random weights:
  Theta1 = np.random.rand(hidden_layer_size, input_layer_size+1) * 0.4 - 0.2
  Theta2 = np.random.rand(num_labels, hidden_layer_size+1) * 0.4 - 0.2

  # flattened initial guess
  theta0 = np.concatenate((Theta1.flatten(), Theta2.flatten()))

  # Minimize the cost function using a nonlinear conjugate gradient algorithm
  args = (input_layer_size, hidden_layer_size, num_labels, X, y, lmbda)  # parameter val
  theta = optimize.fmin_cg(cost_function, theta0, fprime=gradient, args=args,  maxiter=6

  return 0
```

ann.py hosted with ❤ by GitHub view raw

That's it! There you have it, a functioning 3-layer neural network for multi-class classification.

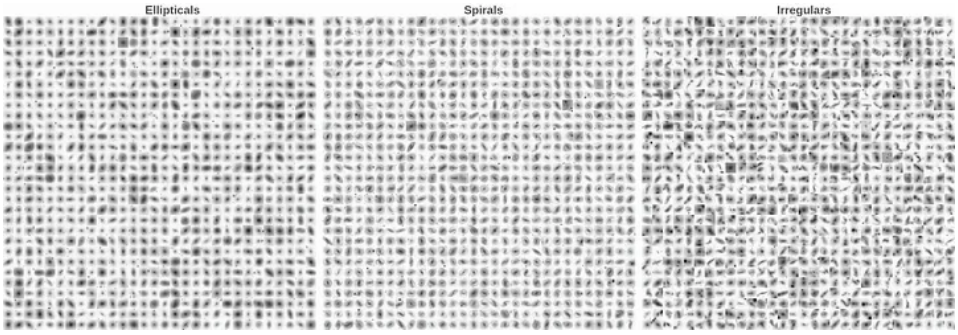
Neural networks are really cool because they can successfully approximate any ideal function according to the Universal Approximation Theorem. That is to say, if you had access to every single piece of data (input and output), there would be an ideal function that could reproduce it all; and a neural network is mathematically guaranteed to get closer to approximating that ideal function as you add more layers/nodes.

Galaxy Image Training Data

Included in this tutorial is a training dataset of images of galaxies I created from the Galaxy Zoo Project, which is a research program that used citizen science to classify tens of thousands of galaxies by their morphologies. The

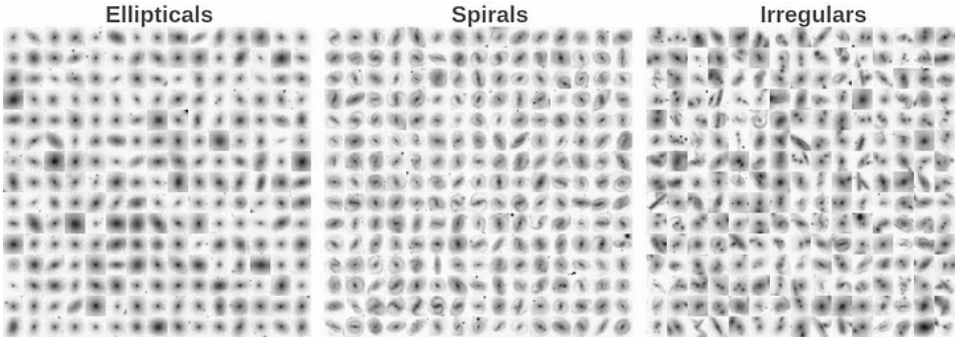
images in this example are cropped, converted to grayscale, and reduced to 32 x 32 pixels in order to create a simplified dataset that is not too expensive to work with on a laptop. Furthermore, we consider just three morphologies: Elliptical, Spiral, and Irregular. A training set of 2700 galaxies are used, as well as a test/validation set of 675 galaxies. Our example is meant to be pedagogical and quick-to-run, rather than scientifically detailed.

Below is our training set of galaxy images organized by type:



Training dataset of 2700 galaxies (900 Elliptical, 900 Spiral, and 900 Irregular)

And our test/validation set:

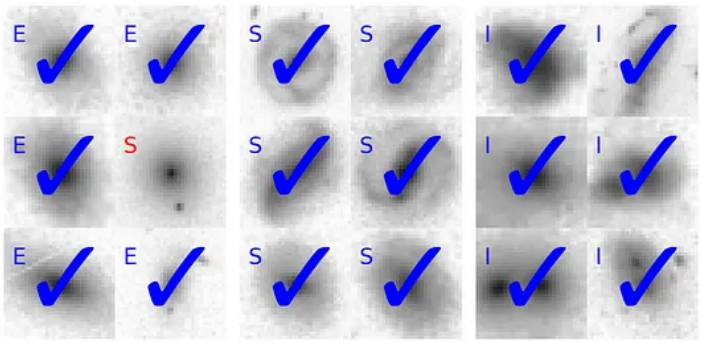


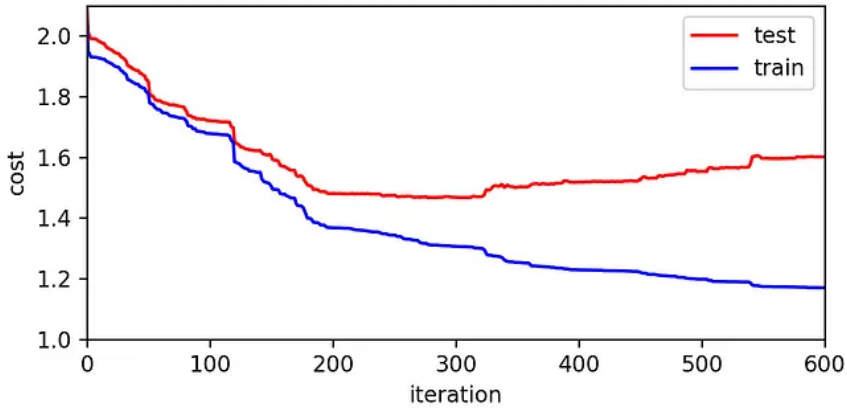
Test/validation dataset of 675 galaxies (225 Elliptical, 225 Spiral, and 225 Irregular)

We consider a 3-layer artificial neural network with a single hidden layer length of 8. We train the data for 600 iterations and fixed the regularization parameter λ to 1.

Running the code allows you to visualize the training of the Artificial Neutral Network in real-time as it learns to classify images of galaxies. At every iteration cycle, the code plots a random sample of galaxies and the predictions given by the current model (and whether it is correct or not). Additionally, the cost function of the training and testing data sets is calculated and plotted as a function of the iteration count. The cost function of the training data set decreases with time, while the cost function of the test/validation data set has a minimum point, to the left of which the model underfits the data and to the right of which the model overfits. The final

trained model has an accuracy of 74% on the test data. Further improvements may be achieved by considering different values for the length of the hidden layer and the regularization parameter, and more training data.





Neural Networks for multi-class classification problems are abundant. A well-known example is hand-writing recognition, and the MNIST database is a commonly used one for teaching a computer how to interpret images of digits and is often used in introductory AI courses:

Sample images from the MNIST dataset used for training a hand-written digit classifier

Neural Networks for multi-class classification can be used for all sorts of applications, including image and speech recognition, medical diagnosis, fraud detection, and sentiment analysis just to name a few. Large language models such as OpenAI's GPT-4 or Meta's LLama-2 are also based on neural network architecture and train models with billions of parameters using massive GPU clusters.

We wrote our Python code with the goal to be pedagogical and dive into the details of the main concepts that underpin neural networks. However, pure Python implementations can be slow (even when we avoid 'for' loops). For performant applications, the following two deep-learning libraries are most commonly used:

- TensorFlow. An open-source machine learning library released by Google, with support for CPUs, GPUs, and TPUs.
- PyTorch. An open-source is a machine learning framework based on the Torch library, originally developed by Meta AI.

Download the Python code on GitHub for our Artificial Neural Network algorithm to visualize the training in real-time and perhaps modify it to improve its accuracy further or apply it to your own classification problems. Enjoy!

Artificial Intelligence | Python | Machine Learning | Data Science | Astronomy

340 1

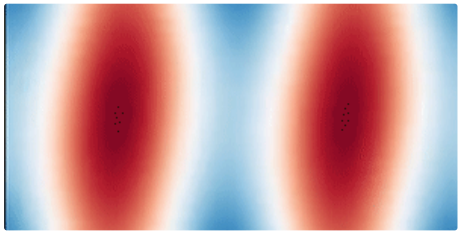
Written by Philip Mocz

1.1K Followers · Writer for Level Up Coding

Computational Physicist. Sharing intro tutorials on creating your own computer simulations! Harvard '12 (A.B), '17 (PhD). Connect with me @PMocz

Follow

More from Philip Mocz and Level Up Coding



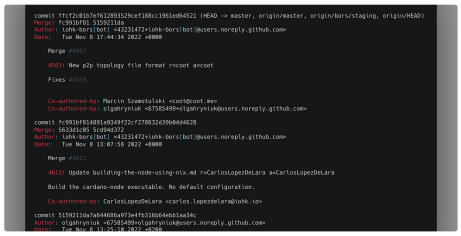
Philip Mocz in Level Up Coding

Create Your Own Navier-Stokes Spectral Method Fluid Simulation...

For today's recreational coding exercise, we solve the Navier-Stokes equations for an...

7 min read · 4 days ago

491 1



Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

4 min read · Nov 15, 2022

8.7K 86

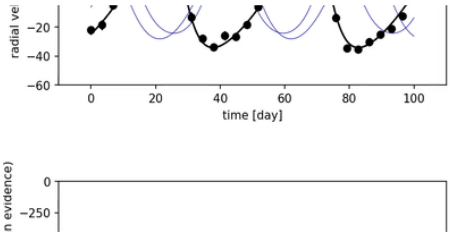
Sanjay Priyadarshi in Level Up Coding

I Spent 30 Days Studying A Programmer Who Built a $230...

Steal This Programmer Blueprint

13 min read · 2 days ago

397 3



Philip Mocz in Level Up Coding

Create Your Own Nested Sampling Algorithm for Bayesian Paramete...

In today's recreational coding exercise, we learn a more advanced and robust Monte...

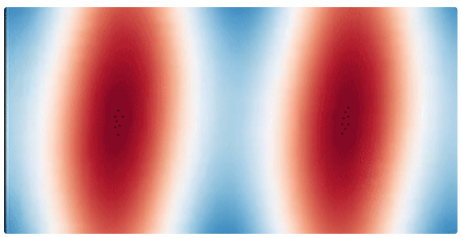
6 min read · Jun 8

283

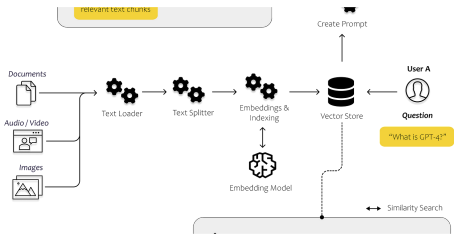
See all from Philip Mocz

See all from Level Up Coding

Recommended from Medium



Philip Mocz in Level Up Coding



Dominik Polzer in Towards Data Science

Create Your Own Navier-Stokes Spectral Method Fluid Simulation...

For today's recreational coding exercise, we solve the Navier-Stokes equations for an...

· 7 min read · 4 days ago

491 1

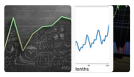
All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt...

· 26 min read · Jun 22

4K 37

Lists



Predictive Modeling w/ Python

18 stories · 226 saves

Practical Guides to Machine Learning

10 stories · 237 saves

Natural Language Processing

476 stories · 101 saves

ChatGPT

21 stories · 91 saves

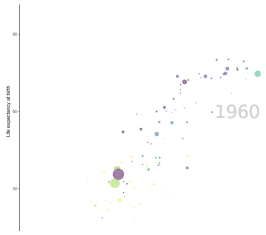
Sydney Nye in Towards Data Science

Mastering Monte Carlo: How To Simulate Your Way to Better...

How a Scientist Playing Solitaire Forever Changed the Game of Statistics

· 25 min read · 6 days ago

538 8



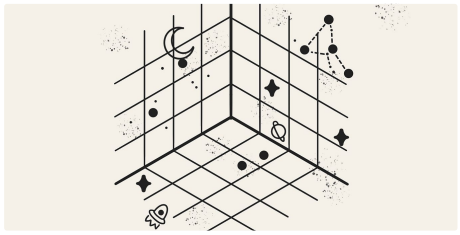
Keith McNulty

How to create Hans Rosling's famous animated bubble chart

A great little learning exercise that illustrates the range and flexibility of the R language

· 6 min read · Aug 1

227 1



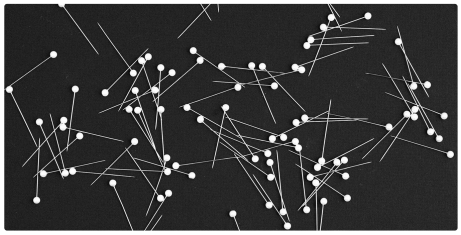
Leonie Monigatti in Towards Data Science

Explaining Vector Databases in 3 Levels of Difficulty

From noob to expert: Demystifying vector databases across different backgrounds

· 8 min read · Jul 4

2K 21



Vikas Kumar Ojha in Geek Culture

Binary Neural Networks: A Game Changer in Machine Learning

This blog explains about binary neural networks which have potential of...

· 9 min read · Feb 19

111 1

See more recommendations