Build Hand Gesture Recognition from Scratch using Neural Network — Machine Learning Easy and Fun

From Self Captured images to Learning the Neural Network Model

Gavril Ognjanovski Dec 24, 2018 · 8 min read ★

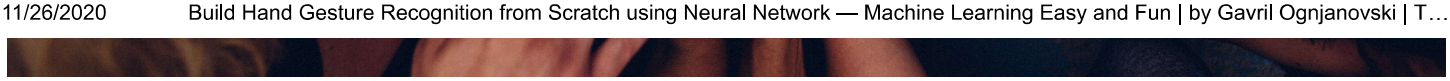

Photo by Perry Grone on Unsplash

Introduction

This algorithm should be working with all the different kinds of skin color, just make sure to position your hand in the middle.

For this solution I used the GNU Octave and Visual Studio Code.

The data and the code are available on my Github repository.

Gago993/HandGestureRecognitionNeuralNetwork

Hand Gesture Recognition software using Neural Networks - Gago993/HandGestureRecognitionNeuralNetwork

github.com

All the work that we need to do can be split in 5 steps:

1. Generate and Prepare the Data
2. Generate Features
3. Generate ML Model
4. Test the ML Model
5. Predict with ML Model

So let's start…

Generate and Prepare the Data

Since we are building this project from the bottom. First thing we need to do is to create the data that we are going to use for training the Neural Network model.

For this step I used my computer build-in camera. I captured 78 images from my hand showing 4 different gestures and they are split in 4 folders. I crop some of the images so they are better "fit" for training our model later. All of the training (prepared) images are stored in ***dataset*** folder.

- left — contains 27 images of hand pointing left
- right — contains 24 images of hand pointing right
- palm — contains 11 images of hand palm
- peace — contains 14 images of peace hand (V sign)

Image 1: Dataset Example (Peace V Sign)

Generate Features

After the training images are ready we can continue with the next step which is processing all the images and creating the features. The main algorithm for separating the hand from the image is done in few simple steps:

1. Load the image
2. Resize the image to 50x50
3. Convert image from RGB to YCbCr color

4. Pick the central point color (expect the hand to be placed in the middle of the image)
5. Segment the hand by skin color range defined from step 3.
6. Mark the selected pixels with white color and others with black color

This algorithm is placed in ***processSkinImage.m*** file. I mark down each Step within the code.

This image skin processing is used by the ***create_image_dataset.m*** file which go through all the images process them using the code above and write them in separate folder called ***dataset_resized*** in left, right, palm, peace folders respectively.

At the end we need to prepare our images so they can be used for generating and testing our Neural Network model later. Since we got 78 images and they are 50x50 pixels we are going to save them as 78x2500 size matrices where each column represent pixel from our image . Also we are going to randomly split the matrices in two as two separate sets. A training matrix will be saved in ***x_features_train*** matrix and will contain 80% of the images and the test matrix in ***x_features_test*** with other 20% of the images. The labels will be saved in ***y_labels_train*** and ***y_labels_test*** respectively.

The code for creating the features matrices looks like

```
...
% Generate random number from 1 to 10
randNum = ceil(rand() * 10);

% Split the images in 80%-20% train-test set
if randNum > 2
% Create the features for the image
X_train = [X_train; image_out(:)'];
y_train = [y_train; label];
else
X_test = [X_test; image_out(:)'];
y_test = [y_test; label];
endif
...
```

There are four types of labels:

- [1 0 0 0] — left pointing hand image
- [0 1 0 0] — right pointing hand image
- [0 0 1 0] — palm hand image
- [0 0 0 1] — peace sign hand image

This labels are created in ***create_image_dataset.m*** as shown in the code below. Where ***folder*** is the folder name where the image is contained and ***ismember*** returns 1 of the 4 options from the bullet list above.

```
...
label_keys = { 'left', 'right', 'palm', 'peace'};
...
label = ismember(label_keys, folder);
...
```

After this script is finished we can check the in ***dataset_resized*** folder for the processed images and we should see something like this

Image 2: Processed Image (Peace V Sign)

There also should be ***x_features_train*** and ***y_labels_train*** files that we are going to use to train our model in the next step and also ***x_features_test*** and ***y_labels_test*** to test our model later.

Before going on, in this blog I assume that you are already familiar with Neural Networks and that's why I'm not going to deep dive into Neural Network explanation.

In order to fully understand the concepts and formulas I suggest reading my post for deeper understanding of Neural Networks.

Everything you need to know about Neural Networks and Backpropagation — Machine Learning Made Easy...

Neural Network explanation from the ground including understanding the math behind it

towardsdatascience.com

Generate ML Model

Let's start with defining our NN structure. We will use one hidden layer in the network. The input layer size will be 2500 nodes, since our images are 50x50 pixels. The hidden layer size will be 25 nodes and the output will be 4 nodes (4 type of signs).

Defining the hidden layer size has no strict formula but usually it depends on the question "How well does it fits the data?"

Image 3: Andrew Ng on Neural Network Size

Here we are going to use the ***main.m*** file and we will:

- Load the features and labels
- Randomly initialize Theta values (NN nodes weights)
- Create the cost function and forward propagation

- Create the gradient for the NN cost function (*Backpropagation*)
- Minimize the cost function using ***fmincg*** minimizer

Load the features and labels

So let's start step one which is loading the features and the labels. We do that by using ***dlmread*** function.

```
...
X = dlmread('x_features_train');

% Labels for each processed training image
%[1 0 0 0] - left, [0 1 0 0] - right, [0 0 1 0] - palm, [0 0 0 1] -
peace
y = dlmread('y_labels_train');
...
```

Randomly initialize Theta values (NN nodes weights)

Next we need to initialize the Theta values using ***randInitializeWeights.m*** function. Which is represented by the following code

```
epsilon = sqrt(6) / (L_in + L_out);
W = zeros(L_out, 1 + L_in);
W = (rand(L_out, 1 + L_in) * 2 * epsilon) - epsilon;
```

Where the generated values are between [-epsilon, epsilon]. This code is related with statistics formula for "Uniform Distribution Variance". If you are more interested about this formula I will leave links at the end of this blog or you can post a question.

Create the cost function and forward propagation

Our next goal is to implement the *Cost Function* defined by the equation below.

Image 4: Regularized Cost Function

Where *g* is the activation function (Sigmoid function in this case)

In order co compute the cost we need to use Feedforward computation. The code is implemented in ***nnCostFunction.m.*** We will use a for-loop over the examples to compute the cost also we need to add the column of 1's to the X matrix which represent the "bias" values. The θ_1 (Theta 1) and θ_2 (Theta 2) values are parameters for each unit in the NN, the first row of θ_1 corresponds to the first hidden unit in the second layer.

Create the gradient for the NN cost function (*Backpropagation*)

To be able to minimize the cost function we need to compute the gradient for the NN cost function. For that we are going to use the *Backpropagation* algorithm*,* short for "backward propagation of errors", is used for minimizing our cost function which means minimizing error for our NN and minimizing the error for each output neuron. This calculation is part of the code implemented in ***nnCostFunction.m***

Minimize the cost function using *fmincg* minimizer

One we computed the gradient, we can train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as *fmincg.* This function is not a part of the Octave so I got it from Machine Learning Course by Andrew Ng. As fur as I know this function is faster than the ones implemented in Octave and it uses Conjugate gradient method.

fmincg takes 3 arguments as shown in the code example below. It takes the cost function, initial θ (Theta) parameters concatenated into a single vector and the options parameter.

After running the *fmincg* on our test examples we get the θ values in one vector and we need to reshape them in matrices for easier matrix multiplication.

Test the ML Model

Now we successfully generated our Theta (weight) values from the NN. Next thing to do is to check how well our model fits the training and how well it performs on the the test data.

In order to predict we are going to use the ***predict*** function located in ***predict.m*** file.

This function takes the \theta parameters and features matrix. Then it ads column of 1's which represent the "bias" values to the features matrix. And just multiply the features with the \theta values for both of the NN layers. Then it gets the vector h2 with size (number_of_images x 4), where each column (4) represent the possibility of that image being in that class (left, right, palm, peace). At the end if returns the one with the highest probability.

Let's perform train set accuracy and test set accuracy.

```
...
pred = predict(Theta1, Theta2, X_train);
% Compare the prediction with the actual values
[val idx] = max(y_train, [], 2);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == idx)) *
100);

...
pred = predict(Theta1, Theta2, X_test);
% Compare the prediction with the actual values
[val idx] = max(y_test, [], 2);
fprintf('\nTest Set Accuracy: %f\n', mean(double(pred == idx)) *
100);
```

What we get is

```
Training Set Accuracy: 100.000000%
Test Set Accuracy: 90.909091%
```

Predict with ML Model

I also made a few additional images and put them in the ***test*** folder. They represent full images (not edited) so I could test the performance of the NN model. What I got was write prediction for all 4 of them.

```
pred =  2
Type: right

pred =  1
Type: left
```

```
pred =  3
Type: palm

pred =  4
Type: peace
```

You can try this out with your images and even with your own training images. Just put your training images under the ***dataset*** folder and call the ***create_image_dataset.m*** file to create the train and test matrices. Also all the ***test*** images under ***test*** folder and you are ready to call the ***main.m*** script.

Conclusion

Congratulations on building your Machine Learning Neural Network Model from scratch. Hopefully this will help understand the big picture when working with Neural Networks and how to make the first step. For any questions or suggestions please post a comment or contact me.

Hope you enjoyed it!

Useful Links

Statistics/Distributions/Uniform - Wikibooks, open books for an open world

The (continuous) uniform distribution, as its name suggests, is a distribution with probability densities that are the…

en.wikibooks.org

What are good initial weights in a neural network?

I have just heard, that it's a good idea to choose initial weights of a neural network from the range $(\frac{-1}{\sqrt…

stats.stackexchange.com

A Step by Step Backpropagation Example

Background Backpropagation is a common method for training a neural network. There is no shortage of papers online that…

mattmazur.com