Neural Net from scratch (using Numpy)

Sanjay.M Follow
Nov 14, 2018 · 8 min read

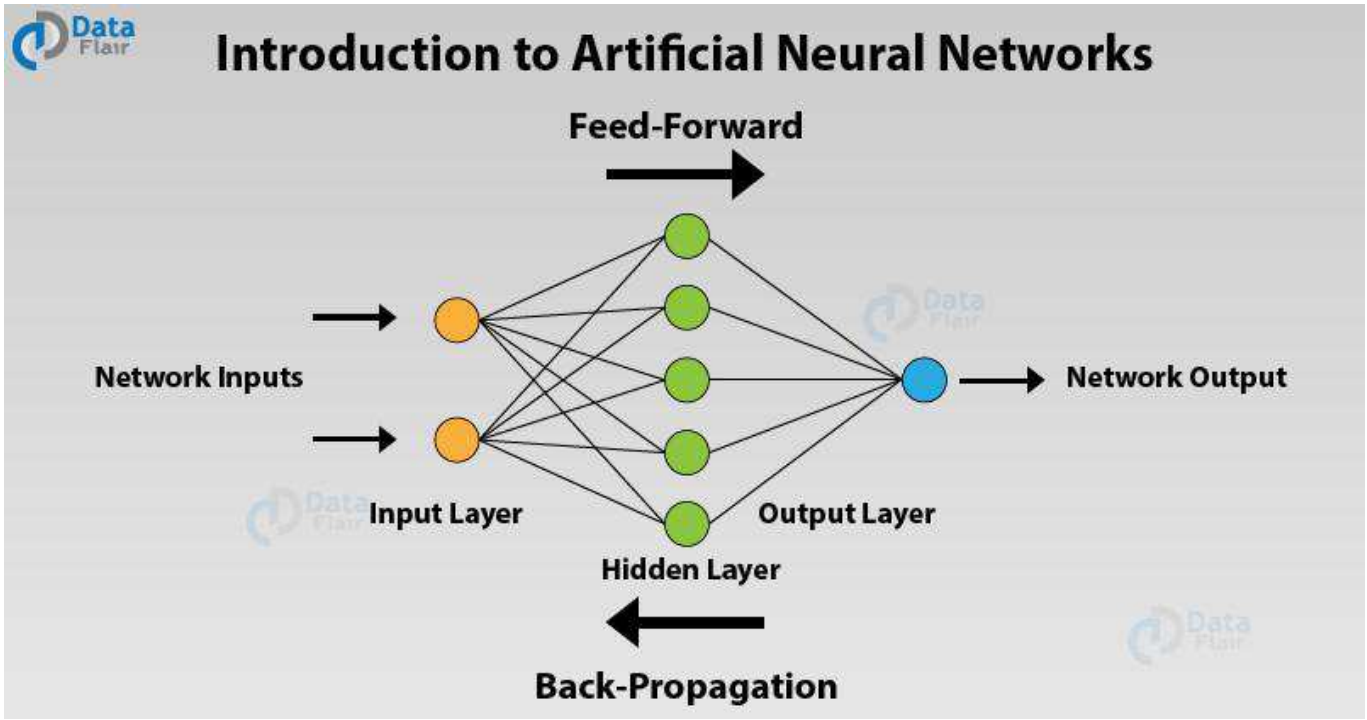


image Source: Data Flair

This post is about building a shallow NeuralNetowrk(nn) from scratch (with just 1 hidden layer) for a classification problem using numpy library in Python and also compare the performance against the LogisticRegression (using scikit learn).

Building a nn from scratch helps in understanding how nn works in the back-end and it is essential for building effective models. Without delay lets dive into building our simple shallow nn model from scratch.

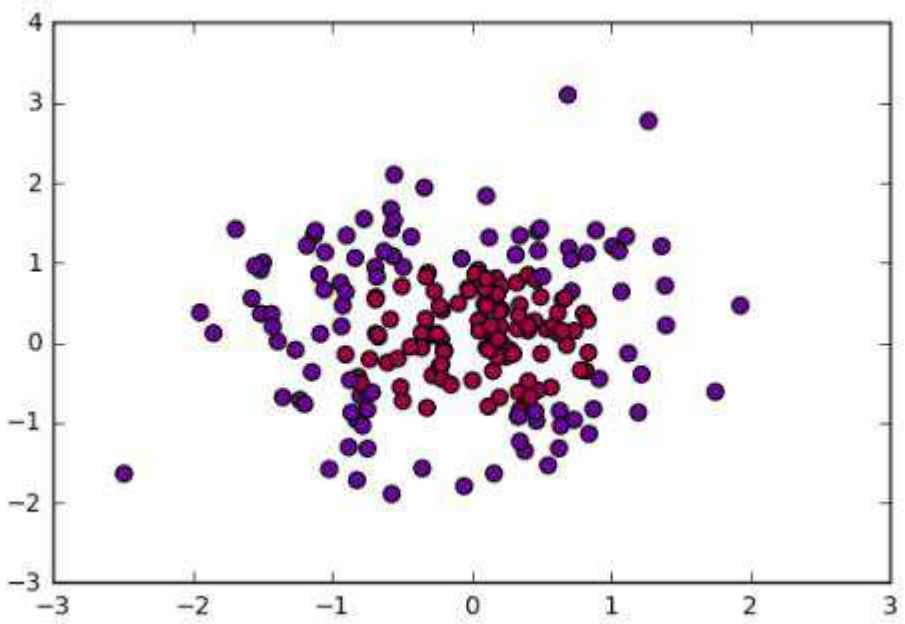
Complete code is available **here**.

For this task I am generating a dataset using the scikit learn dataset generator **make_gaussian_quantiles** function (Generate isotropic Gaussian and label samples by quantile). Generated input dataset will have have two features ('X1' and 'X2' and output 'Y' will have 2 classes (red: 0, blue:1), with total of 200 examples.

```
def load_extra_datasets():
    N = 200
    gaussian_quantiles = sklearn.datasets.make_gaussian_quantiles
     (mean=None, cov=0.7, n_samples=N, n_features=2, n_classes=2,
shuffle=True, random_state=None)
    return  gaussian_quantiles
```

```
gaussian_quantiles= load_extra_datasets()
X, Y = gaussian_quantiles
X, Y = X.T, Y.reshape(1, Y.shape[0])


# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



The dataset generated has two classes, plotted as red and blue points. Our goal is to build a Machine Learning classifier that predicts the correct class given the X- and Y- coordinates. As we can see in the graph, data is not linearly separable, so we can't draw a straight line that separates the two classes. This means that linear classifiers such as Logistic Regression, won't

be able to fit these kind of data properly. In these cases nn comes to our rescue. In nn feature engineering is not required, as hidden layers will automatically learn feature patterns to classifies the data accurately.

Logistic Regression (LR):

First lets train a LR classifier using the inputs x- and y-values and the output will be the predicted class (0 or 1). We will use the Regression class from *scikit-learn*

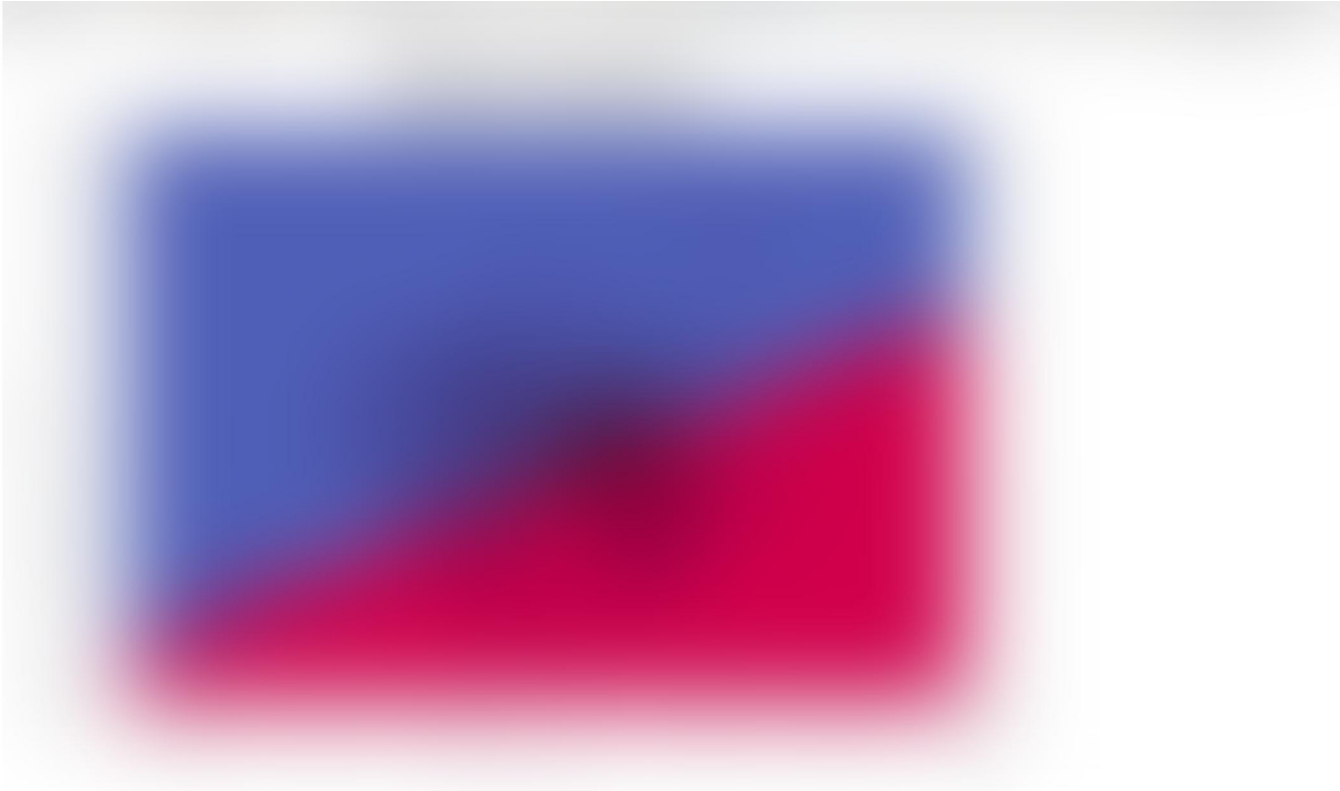
```
clf = sklearn.linear_model.LogisticRegressionCV()
clf.fit(X.T, Y.T)


# Plot the decision boundary for logistic regression


plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")


# Print accuracy


LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' %
float((np.dot(Y,LR_predictions) + np.dot(1-Y,1-
LR_predictions))/float(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```



As we can see from the above LR could able to classify just 53% of the data points correctly, as these data points are not linearly separable.

Neural Network:

Lets now build a simple nn with 1 hidden layer with 4 neurons. Input layer will have 2 nodes as our data has two features (X1 and X2) and output layer will have one node , based on the probability threshold we will classify the output as either red or blue (0 or 1).

Source: Andrew Ng's Coursera

We need to do the below steps to build our nn model.

- Define Network structure (# of input units, # of hidden units, etc).
- Initialise the model's parameters
- Perform the below steps in loop until we get minimum cost/optimal parameters.

1. Implement forward propagation
2. Compute loss

3. Implement backward propagation to get the gradients
4. Update parameters

- Then merge all the above steps into one function we call `nn_model()`. Once we built `nn_model()` and learnt the right parameters,we can make predictions on new data.

1. **Define Network Structure**: As mentioned earlier, for input layer number of nodes will be 2, and for hidden layer i set it to 4. By choosing more nodes in this layer, we can make model learn complex functions. But it comes at a cost of heavy computation to make predictions and learn the network parameters. More number of hidden layers and nodes could also lead to over-fitting of our data.

```
#X and Y are the input and output variables

    n_x = X.shape[0] # size of input layer`
    n_h = 4
    n_y = Y.shape[0] # size of output layer
```

2. Initialize the model's parameters: W1 (weight matrix for hidden layer) and W2(wight matrix for output layer) parameters are initialized randomly

using the numpy random function. Multiplied by 0.01 as we do not want the initial weights to be large, because it will lead to slower learning. b1 and b2 are initialized to zeros.

W1 — weight matrix of shape (n_h, n_x) for hidden layer
b1 — bias vector of shape (n_h, 1)
W2 — weight matrix of shape (n_y, n_h) for output layer
b2 — bias vector of shape (n_y, 1)

```
W1 = np.random.randn(n_h,n_x) * 0.01
b1 = np.zeros(shape=(n_h, 1))
W2 = np.random.randn(n_y,n_h) * 0.01
b2 = np.zeros(shape=(n_y, 1))
```

3. Forward Propagation: During forward propagation the input feature matrix is fed to the every neuron in the hidden layer. Which will be multiplied by the respective initial set of weights(W1) and bias(b1) will be added to form Z1 matrix (linear transformations of the given inputs). Then we apply the non-linearity to Z1 by feeding it through an activation function (to apply non-linearity). We chose 'tanh' as our activation function as it fits to many scenarios. The output of this activation function/hidden layer will

be A1 (which is a matrix of size (4,1) contains the activations from the 4 neurons i.e a1, a2, a3 a4).

For the next layer which is the final output layer in our case, we multiply the inputs from the previous layer (A1) with the initial weights of output layer(W2), add bias(b2) to form Z2. Then apply the sigmoid activation function on Z2 to produce out final output A2 (which is our predictions). We used sigmoid for our final layer as we want our output to be between 0 and 1. Based on the probability threshold we can decide weather the output is red or blue. This is how nn makes predictions during forward propagation, which is just a sequence of matrix multiplications and application of activation function(s).

```
# Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1,X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2,A1) + b2
    A2 = sigmoid(Z2) # Final output prediction
```

4. Compute Loss: Now that we have our predictions, next step would be to check how much our predictions differ from the actual values, i.e loss/error. Here we do not use mean square error (MSE) to compute our loss as our

prediction function is non-linear(sigmoid). Squaring the prediction will results in non-convex function with many local minimums. In such case gradient descent many not find the optimal global minimum. Hence we use the binary **Cross_Entropy** loss (log-likelihood method for error estimate), this cost function is convex in nature, so reaching the global minimum point (minimum loss point) will be easier. Below is the cost function formula and the code.

m : Number of training examples


```
# Compute the cross-entropy cost

logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1
- A2)
cost = - np.sum(logprobs) / m
```

5. BackPropagation/Gradient Descent (GD): Back propagation is used to calculate the gradients(slope/derivatives) of the loss function with respect

to the model parameters(w1,b1,w2,b2). To minimize our cost we use the GD algorithm, which uses the computed gradients to update the parameters so that the our cost keeps reducing over iterations, i.e it help move towards global minimum.

- Below are the gradient/slope computing formulae for each of the model parameters. 'm' is number of training examples.

Sourced from Coursera

```
dZ2 = A2 - Y
dW2 = (1 / m) * np.dot(dZ2, A1.T)
db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)
dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
```

```
dW1 = (1 / m) * np.dot(dZ1, X.T)
db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)
```

6. Update the parameters: Once we have computed our gradients, we multiply them with a factor called learning-rate (converging rate) and subtract from the initial parameters to get the updated parameters(weights and biases). Learning rate should be minimal so that we will not miss the global minimum point.

- Multiply the gradients by learning rate
- Subtract from weights

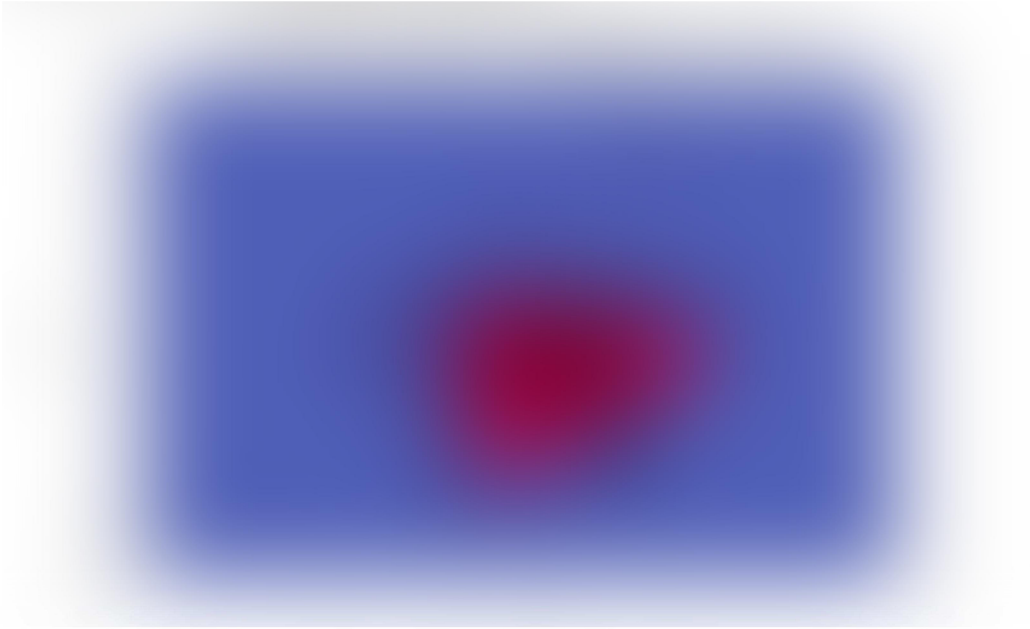
```
W1 = W1 - learning_rate * dW1
b1 = b1 - learning_rate * db1
W2 = W2 - learning_rate * dW2
b2 = b2 - learning_rate * db2
```

Now we have performed one round of forward propagation and backward propagation for all the training examples, i.e we completed 1 epoch. We need to repeat these steps over multiple epochs till our cost is minimum(model reaches global minimum point)or the learning stops (no updates to the parameters).

Below is the function ‘nn_model’ which performs all the above operations repeatedly over a given number of epochs(num_iterations) and prints the cost after every 1000 epochs. The output of this function will be the final set of optimised parameters(weights/baiases).

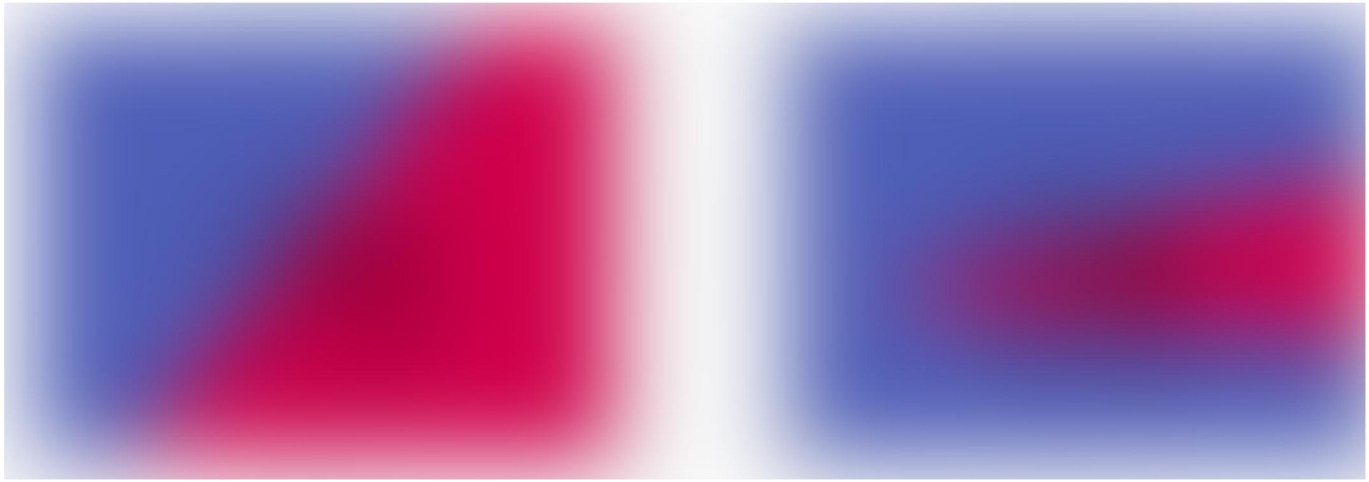
7. Predictions: Below is the function which makes prediction using the learned parameters, by doing just a forward propagation. We are setting a threshold as 0.5, if the output of the final layer (A2) is > 0.5 then we are classifying it as 1:blue else 0:red.

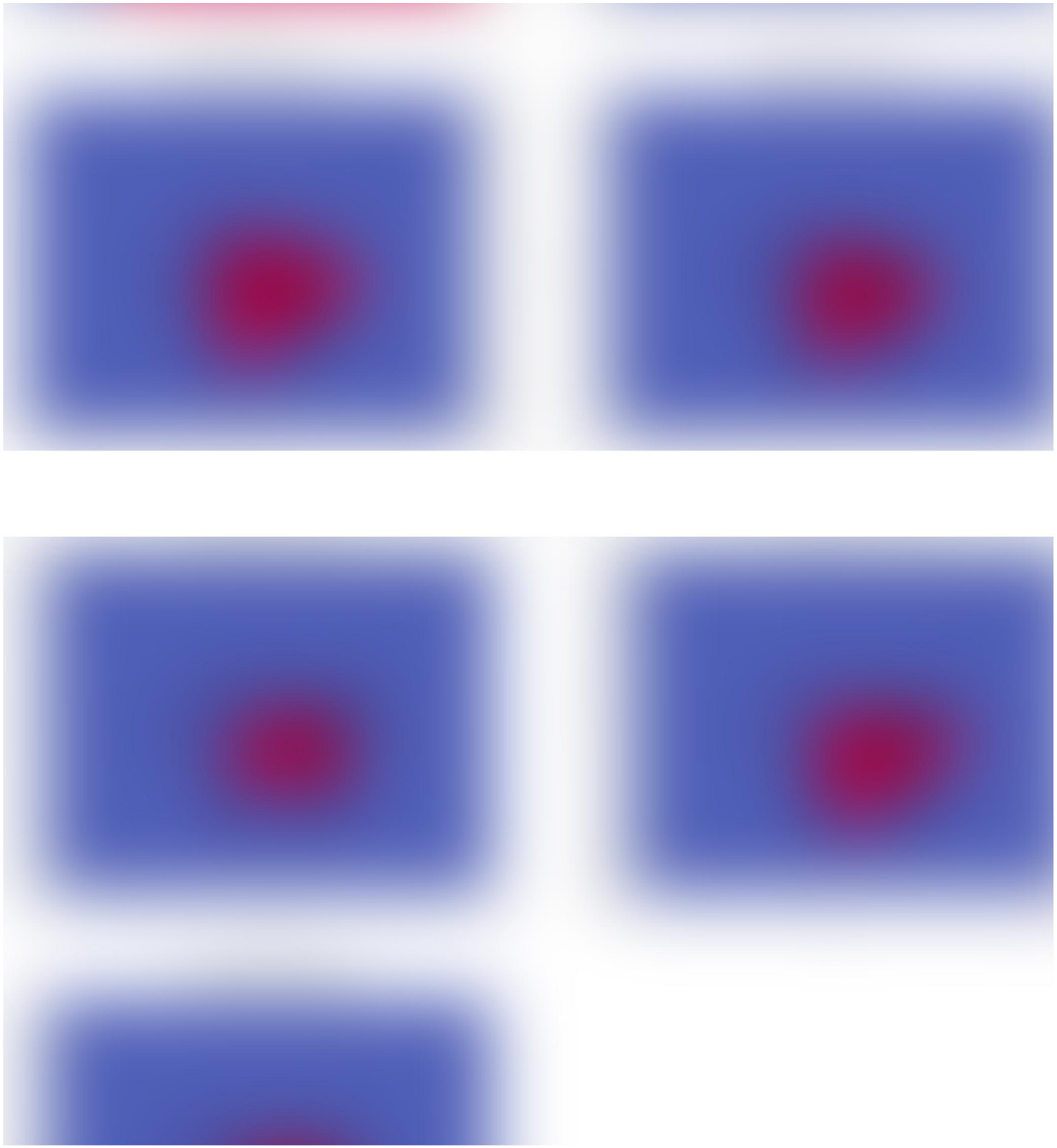
Now lets train our final model by running the function **nn_model** over 5000 epochs and see the results.

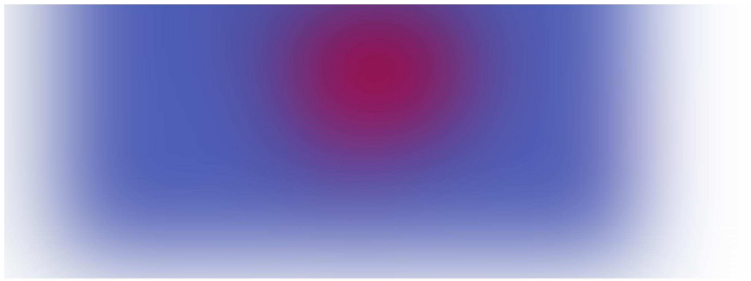


As we can see from the results, our nn model has learnt the patterns well. Which is able to learn non-linear decision boundaries which separates the classes. And our cost started from 0.69 and reached to 0.089 after 4000 epochs. Final accuracy came to 93% which is much higher compare to what we achieved from Logistic regression which is just 53%.

Tuning the Hidden Layer size: Next step would be to decide the optimal number of neurons for the hidden layer to see if our model can do better without over-fitting. For this lets train the model with different number of nodes (1,2,3,4,5,20,50) and see the results.







Below are the findings from this test.

- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models over-fit the data. We can see that at n_h = 50 the model seems over-fitting (100% accuracy).
- The best hidden layer size seems to be around n_h = 4. Indeed, a value around here seems to fits the data well without also incurring noticeble over-fitting.

Thanks a lot for reading till end, please comment for any suggestions/changes. And please do 👏 if you like the post… 😉

Machine Learning Neural Networks Deep Learning Numpy Data Science