CS164 Final Project

December 2020

Akmarzhan Abylay

Problem Description

Machine learning using neural networks (NNs) is becoming more and more accessible as various tutorials and packages, such as Tensorflow, Keras, and PyTorch, come in, allowing us to create complex models in just a few lines of code. It takes a few hours of research to build NNs that perform exceptionally well, which is why the tools are so popular among computer scientists. However, not all people understand how exactly NNs work, treating them as a black-box model. This is why it is interesting for me to explore the exciting world of NNs and the underlying mechanisms that allow them to be so successful in all types of regression/classification tasks, specifically. In this assignment, I will build a neural net from scratch based on various algebra concepts and algorithms (e.g., forward and backward propagation) to classify a linearly non-separable 2d dataset. I will also compare the performance of a manual NN with that from Keras.

Solution Specification

Optimization

I explain all of the steps in the next part, but to summarize what I want to find:

1. **Objective function:** The average binary cross-entropy loss function.
2. **Optimization problem:** Minimizing the objective function.
3. **Decision variables:** The weights and the biases in the model.
4. **Constraints:** There are no explicit constraints on the system, other than the order in which the weights are updated (i.e., from the last layer all the way to the back) since the weights and biases can take any value. However, there are some edge cases. For example,
   (a) The cross-entropy score is built in a way that the value of log(0) is undefined, which means that if the specific value of $\hat{y}$ is exactly 0 or 1, the value of the loss at that point will be undefined. This is very rare as we get probabilities outside 0 and 1 almost all the time. One way to deal with it would be to add some small number (e.g., $\epsilon = 1e-5$) to the expressions, so that they are defined, and the loss function is not too far from the actual value.
   (b) "Breaking symmetry problem," which we will discuss further in the assignment.
5. **Feasible Set**: As mentioned, there are no explicit constraints for my type of optimization problem (i.e., non-linear classification on a 2d dataset), which is why the weights and biases can be of any value. However, it only works because the points in our dataset don't really have meaning (i.e., very simplistic). Depending on the type of problem we are trying to solve, there might be different constraints. For example, if we wanted to use the neural network for harder tasks such as part-of-speech tagging, then we would have to assign some natural constraints. For instance, there is an intrinsic structure to the words in the sentence, which is why we wouldn't see a verb followed by a verb (unless separated by a comma), so we could let the NN know about these nuances.

   Additionally, although the cost function might be convex in $\hat{y}$, it might not be in most cases convex in parameters θ (weights and biases) as $\hat{y}$ itself is a function of a collection of weights and biases.

We care about the convexity in the parameters because these are our decision variables. Although NNs are made of convex parts, the composition of convex functions is not convex, so deep neural networks are not convex (Cornell University, 2017). Even the most simplistic NNs are neither convex nor concave, and the deeper the NN goes, the less convex it becomes. This is why it typically hardly converges to global optimum points, so we will get local optima (i.e., and rarely, global). However, we have a very simplistic dataset, so we'll see how well the model performs.

Context

What is a neural network? How does it work? To build a model from scratch, we need to understand how NNs work first. As the name suggests, NNs draw inspiration from the actual neural system, where each node represents a neuron, and the whole model simulates layers of densely interconnected nodes, which allow the model to process, learn and analyze information. NNs can be seen as universal function approximators and they can learn to approximate any function with enough nodes. They usually include the following general parts:

1. **input layer:** receives the input and passes it through the rest of the artificial neural network.
2. **hidden layers:** apply weights and biases to the input (i.e., the input of each new layer is the output from the previous layer) and outputs them through an activation function.
3. **output layer:** last layer, which produces the outputs of the model.

There are some fundamental variables we need to define for the layers:

1. **[W] weights:** parameters that transform the input.
2. **[b] bias:** constant added to the transformed input to affect the value out of an activation function.
3. **[Z] weighted input + bias:** intermediary value, which is fed into the activation function. It is defined as $WX + b$.
4. **[A] activation:** function that defines the output of a given input and determines whether a node will fire up or not. They introduce non-linear properties into the NN, since without them, the whole model would just be a combination of linear functions.

Let's grab a simple two-layer example to see the predicted output for a given input. For convenience, we will use the sigmoid activation function.

$$\hat{y} = \sigma(W_2\sigma(W_1X + b_1) + b_2)$$

As we see, the output of the previous layer is then fed into the new layer as an input. The output of the second layer is then the predicted final value (i.e., for classification, we can just turn probabilities into classes). This process is called **forward propagation**, and it is defined as calculating the output by passing it through the layers. Now, to optimize the model, we need some way to find out how good our model's predictions are. In this assignment, I am going to work with binary classification. I will optimize the **average binary cross-entropy score as my objective function**, which is similar to using logistic regression that usually optimizes the log loss for all observations (i.e., training) ("Cross entropy," n.d.). It is defined as:

$$E(\hat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n}(ylog\hat{y} + (1-y)log(1-\hat{y}))$$

Here y is the true output, and $\hat{y}$ is the predicted output. E basically gives us an understanding of how far these two are. If the prediction is perfect, the value goes to 0, which is why our task is to minimize the average binary cross-entropy function score.

This is where the optimization part comes in, as we can tweak the weights and biases so that they give us better and better predictions after each iteration. We will do that through a technique called back-propagation. In short, we use gradient descent, and after each iteration, we calculate the loss function's gradient with respect to NN weights and update the weights and biases accordingly.

We would calculate the loss function E and propagate the loss to all of the previous layers, starting from the last layer by changing the layers' associated weights and biases. Let's start from the weight update equations: $w = w - \alpha\frac{\partial E}{\partial w}$ Here α is the learning rate (i.e., the amount by which we update the weights and biases during training). To find $\frac{\partial E}{\partial \hat{y}}$ we need to use the chain rule recursively. For convenience, I will write down the general equations and chain rule (i.e., in the code, it is all correctly linked).

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w}$$

To find the derivatives, I used the derivation rules (i.e., addition, reciprocal) and standard formulas. Below is the derivative of the loss function with respect to the output.

$$\frac{\partial E}{\partial \hat{y}} = \frac{1}{n}\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{1}{n}\frac{-y + y\hat{y} + \hat{y} - y\hat{y}}{\hat{y}(1-\hat{y})} = \frac{1}{n}\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

Below $\hat{y}$ is the same thing as A that we defined in the beginning as it is simply the output of the last layer (i.e., after activation). Also, I am using a lower-case z for convenience. Since we have different activations, we will use a different term depending on whether we have ReLu or sigmoid. The derivative of the sigmoid activation function with respect to z looks like this:

$$\begin{aligned}\frac{\partial \hat{y}}{\partial z} &= -\frac{1}{(1+e^{-z})^2}(-e^{-z}) \\ &= \frac{e^{-z}}{(1+e^{-z})(1+e^{-z})} \\ &= \frac{1}{1+e^{-z}}\frac{1+e^{-z}-1}{1+e^{-z}} \\ &= \hat{y}(1-\hat{y})\end{aligned}$$

This is true because by the reciprocal rule, if we take $1 + e^{-z}$ as some $s(z)$, then we know that $\frac{1}{s(z)}' = -\frac{s'(z)}{s(z)^2}$. Alsom by the addition and chain rules $(1+e^{-z})' = 0 + -e^{-z} = -e^{-z}$. For ReLu, things are a little easier. We will denote the activation as a for easier coordination. The following holds:

$$\frac{\partial a}{\partial z} = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases}$$

The derivative at 0 is undefined, but the actual values rarely are 0, and even if that is the case, we can add some minimal value (e.g., $\epsilon = 1e - 5$) to it so that it doesn't break.

Then, the last bit of information we need is:

$$\frac{\partial z}{\partial \hat{w}} = x$$

x here is the input of that specific layer, but we should note that the subsequent layers take the output of the previous layers as input. So, in the end, we get:

$$\frac{\partial L}{\partial w} = \frac{1}{n} \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y})x = \frac{1}{n}(\hat{y} - y)x$$

This works similarly for the rest of the layers with ReLu activation function, but considering that the output of the layer was defined by ReLu. Also, the update equations for the bias are almost the same but instead of finding $\frac{\partial z}{\partial w}$, we just find $\frac{\partial z}{\partial b}$ (i.e., which is 1, but we should sum over all data points so that we get a single bias term for each weight).

Analysis

When initializing the weights, an important thing to notice is that we shouldn't set all weights with the same values as the model will never learn (i.e., if unequal weights have to be developed). This is called the "breaking symmetry problem" and happens because when the error is propagated back, all weights will move identically and stay the same. Thus, we initialized the weights with some small random numbers. A similar thing is true for biases. We can also re-train the model several times with different initial weights to get better results. However, it is not required in our case, as our dataset is simple.

I used the logic from the previous part and also got inspired by the equations in this resource and built a manual neural network without the use of packages. I also built a Keras NN to compare (see Appendix for code and the loss function plot).

I used a dataset from the `scikit` library called moons. The original dataset looked like this (i.e., I only added little noise to see how well it would classify clean data):

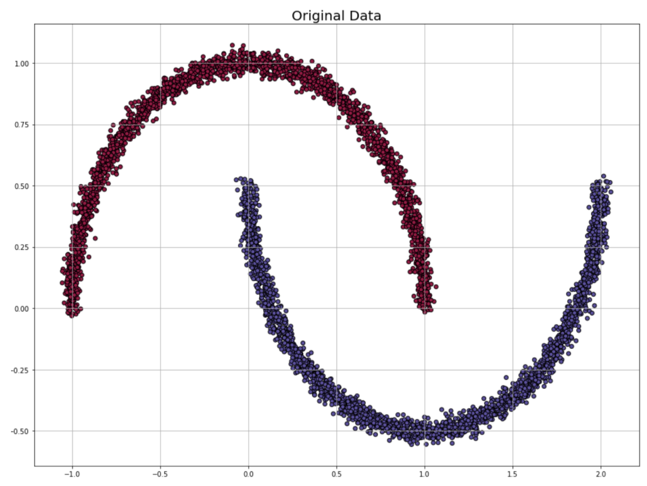


Figure 1: Original Dataset

I then used a simple NN architecture with 25 weights (nodes) in the the first layer, 50 nodes in both the second and the third layers, 25 node in the fourth layer and one node in the last layer (i.e., output). The results were good. The manual NN got 100% accuracy and an average cross-entropy loss of 0.002 after 10000 iterations, while the Keras NN had the same accuracy but no loss at all after 150 epochs (i.e., I am pretty sure it would have had 0 loss after around 40 epochs as we see from the GIF). I made some visualizations inspired by the code from here. Click on the references below the figures to see GIFs.

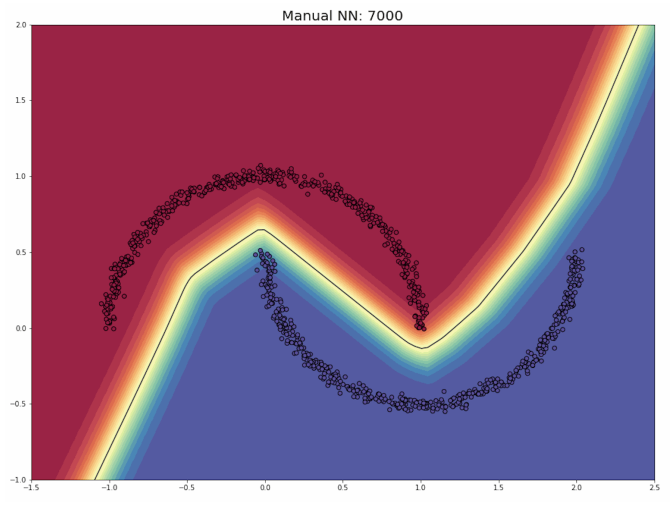


Figure 2: Manual NN (click for a visualization)

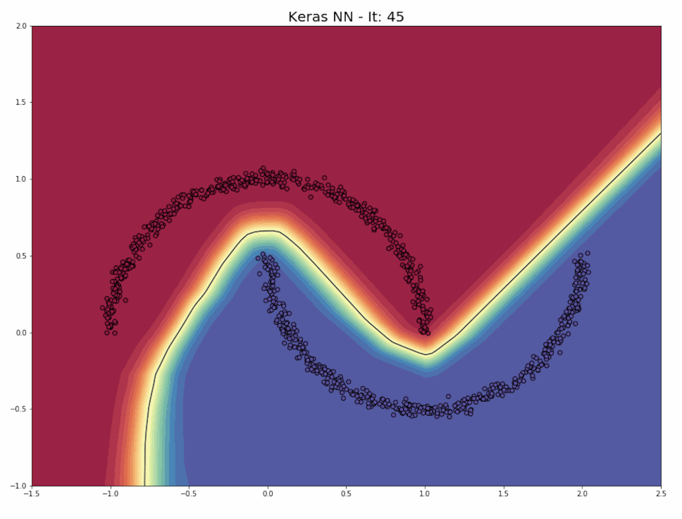


Figure 3: Keras NN (click for a visualization)

As we see from the diagrams above, both the manual NN and the Keras NN (i.e., with an equivalent architecture) performed well (i.e., partially because the dataset was super clear). We can see that the red part refers to one class and the blue part refers to another class. Both models were able to separate this non-linearly separable points. However, on more complex datasets, the manual NN might not be as successful due to the simplicity (e.g., even for this simple data, it took around 7000 iterations to fit). I ended up trying the manual NN on different datasets, but it takes much more time (i.e., iterations) and more complex NN architectures to get good results. As mentioned, the Keras library takes around 175 times fewer iterations to get the same accuracy (i.e., about 7000 vs. 40 epochs), which is why the Keras library is superior. Still, it was a fun experience to understand how a NN works on the inside.

References

1. "Backpropagation." (n.d.) Retrieved from https://ml-cheatsheet.readthedocs.io/en/latest/ backpropagation.html
2. Cornell University. (2017). Non-Convex Optimization. Retrieved from https://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture7.pdf
3. "Cross entropy." (n.d.) Retrieved from https://en.wikipedia.org/wiki/Cross_entropy
4. Jindal, D. (2019). Augmenting Neural Networks with Constrained Optimization. Retrieved from https://towardsdatascience.com/augmenting-neural-networks-with-constraints-optimization-ac747408432f
5. SkalskiP. (n.d.). ILearnDeepLearning.py. Retrieved from https://github.com/SkalskiP/ ILearnDeepLearning.py
6. Skalski, P. (2018). Deep Dive into Math Behind Deep Networks. Retrieved from https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba

LOs

- **#nonconvexprogramming:** effectively applied non-convex programming techniques (i.e., deep neural networks and gradient descent) in a way that addresses the relevant goals (i.e., classify the points in a 2d dataset) and demonstrated a deep grasp of the method by analyzing and explaining it (i.e., what NNs are and how exactly they work).

Appendix

Manual Neural Network

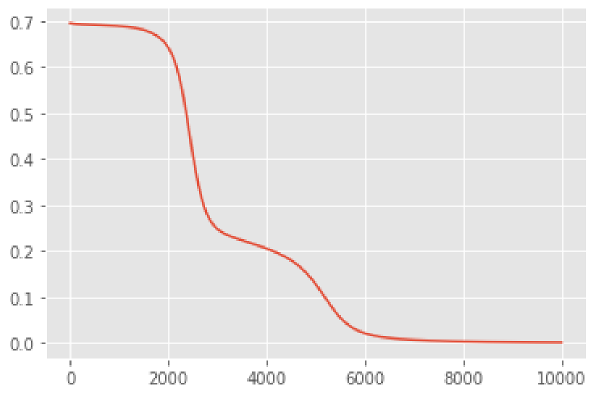
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(Z):
    """
    Sigmoid activation function.
    """
    return 1/(1+np.exp(-Z))

def relu(Z):
    """
    ReLu activation function.
    """
    return np.maximum(0,Z)

def der_sigmoid(Z):
    """
    Derivative of sigmoid activation function.
    """
    return sigmoid(Z) * (1 - sigmoid(Z))

def der_relu(Z):
    """
    Derivative of ReLu activation function.
    """
    return np.sign(np.maximum(0, Z))

def layers(structure):
    """
    Function for initializing the weights and biases based on the NN structure.
    """

    #a dictionary where all of the weights and biases will be stored
    parameters = {}

    #iterating through the given layers
    for idx, layer in enumerate(structure, 1):

        #initializing weights and biases
        parameters['W'+str(idx)] = np.random.randn(
            layer["output"], layer["input"])*0.1
        parameters['b'+str(idx)] = np.random.randn(
            layer["output"], 1)*0.1

    return parameters
```

```python
def map_activation(Z, activation, typ):
    """
    Function for mapping the activation functions.
    """
    if activation=="relu":
        if typ == "backward":
            return der_relu(Z)
        else:
            return relu(Z)
    else:
        if typ == "backward":
            return der_sigmoid(Z)
        else:
            return sigmoid(Z)

def forw_prop(X, parameters, structure):
    """
    Function for forward propagation.
    """

    #storing the values of A and Z
    #for the backpropagation step.
    storage = {}

    #initializing (i.e., first 'layer' is input)
    A_c = X

    #going through all layers given in the structure
    for idx, layer in enumerate(structure, 1):

        A_p = A_c #p: previous, c: current
        W_c = parameters["W"+str(idx)]
        b_c = parameters["b"+str(idx)]
        Z_c = np.dot(W_c, A_p) + b_c
        A_c = map_activation(Z_c, layer["activation"], "forward")
        #adding to the storage
        storage["A"+str(idx-1)] = A_p
        storage["Z"+str(idx)] = Z_c

    #returning the current layer and the temporary storage
    return A_c, storage

def cost(y_hat, y):
    """
    Function for calculating the average binary cross-entropy.
    """

    n = y_hat.shape[1]
    J = -1 / n * (np.dot(y, np.log(y_hat).T) + np.dot(1 - y, np.log(1 - y_hat).T))
    return np.squeeze(J)
```

```
def acc(y_hat, y):
    """
    Function for calculating the accuracy of the model.
    """

    y_prob = np.copy(y_hat)
    y_prob[y_prob>0.5] = 1
    y_prob[y_prob<=0.5] = 0
    return np.mean((y_prob == y).all(axis=0))

def back_prop(y_hat, y, storage, parameters, structure, lr):
    """
    Function for backward propagation.
    """

    y = y.reshape(y_hat.shape)

    dA_p = - (np.divide(y, y_hat) - np.divide(1 - y, 1 - y_hat))

    #going through all layers in a reversed direction
    for idx, layer in reversed(list(enumerate(structure))):

        dA_c = dA_p

        A_p = storage["A"+str(idx)]
        Z_c = storage["Z"+str(idx+1)]
        W_c = parameters["W"+str(idx+1)]
        b_c = parameters["b"+str(idx+1)]

        dZ_c = map_activation(Z_c, layer["activation"], "backward")*dA_c
        dW_c = np.dot(dZ_c, A_p.T) / A_p.shape[1]
        db_c = np.sum(dZ_c, axis=1, keepdims=True) / A_p.shape[1]
        dA_p = np.dot(W_c.T, dZ_c)

        #updating the weights and biases
        parameters["W"+str(idx+1)] -= lr * dW_c
        parameters["b"+str(idx+1)] -= lr * db_c

    return parameters

def train(X, y, structure, epochs, lr=1e-3, visual=False):
    """
    Function for the training process.
    """

    #initializing the parameters
    parameters = layers(structure)

    #storing the accuracy and loss
    loss = []
    accuracy = []

    #iteratively training the model
    for idx in range(epochs):

        #forward propagation
        y_hat, storage = forw_prop(X, parameters, structure)

        #checking the loss and accuracy
        loss.append(cost(y_hat, y))
        accuracy.append(acc(y_hat, y))

        #backward propagation
        parameters = back_prop(y_hat, y, storage, parameters, structure, lr)

        #for the plot
        if visual:
            if idx%100==0:
                viz(idx, parameters)
    #returning the parameters, loss, and accuracy
    return parameters, loss, accuracy
```

[2]:
```
#importing the libraries
from sklearn.model_selection import train_test_split
from sklearn import datasets

N = 10000
#initializing the dataset
(X, y) = datasets.make_moons(n_samples=5000, noise=0.025, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=0)
```

[3]:
```
#structure of the neural network
structure = [{"input": 2, "output": 25, "activation": "relu"},
             {"input": 25, "output": 50, "activation": "relu"},
             {"input": 50, "output": 50, "activation": "relu"},
             {"input": 50, "output": 25, "activation": "relu"},
             {"input": 25, "output": 1, "activation": "sigmoid"}]

#training the model N times
parameters, loss, accuracy = train(X_train.T, (y_train.reshape(((len(y_train),␣
 ↪1)))).T, structure, N, 0.01)

#plotting the loss
with plt.style.context('ggplot'):
    plt.plot(np.arange(0, N, 1), loss)
```



[4]:
```
#testing accuracy and cross-entropy loss
y_hat, _ = forw_prop(np.transpose(X_test), parameters, structure)
test = acc(y_hat, np.transpose(y_test.reshape((y_test.shape[0], 1))))
loss = cost(y_hat, np.transpose(y_test.reshape((y_test.shape[0], 1))))
print('Testing Accuracy: {0} \nCross-Entropy Loss: {1}'.format(np.round(test*100,␣
 ↪3), np.round(loss, 3)))
```

```
Testing Accuracy: 100.0
Cross-Entropy Loss: 0.002
```

Keras Neural Network

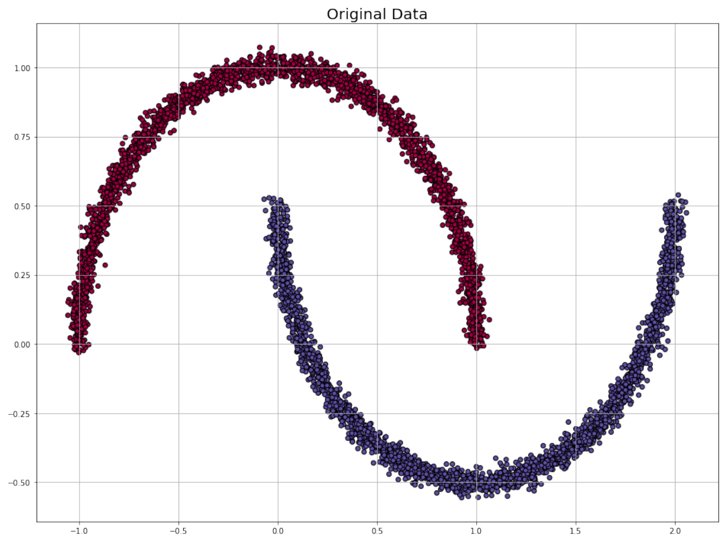
[5]:
```
#importing the Neural Network packages
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

#initializing the same NN
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics='accuracy')
```

[6]:
```
#checking the accuracy and loss
model.fit(X_train, y_train, epochs=150, batch_size=10, verbose=0)
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print('Testing Accuracy: {0} \nCross-Entropy Loss: {1}'.format(round(accuracy*100,␣
 ↪3), round(loss, 3)))
```

```
Testing Accuracy: 100.0
Cross-Entropy Loss: 0.0
```

Visualizations

```
[7]: import os
import matplotlib.pyplot as plt

#plotting function
def plot(X, y, name, file_name=None, preds=None):

    #initializing the figure
    plt.figure(figsize=(16,12))
    plt.gca()
    plt.title(name, fontsize=20)

    #plotting
    if preds is not None:
        b1, b2 = some_grid
        plt.contourf(b1, b2, preds.reshape(b1.shape), 25, alpha = 1, cmap=plt.cm.
 ↪Spectral)
        plt.contour(b1, b2, preds.reshape(b1.shape), levels=[.5], cmap="Greys",␣
 ↪vmin=0, vmax=.6)
    if not file_name:
        plt.grid()
    plt.scatter(X[:, 0], X[:, 1], edgecolors='black', c=y.ravel(), cmap=plt.cm.
 ↪Spectral)

    #saving the pictures
    if(file_name):
        plt.savefig(file_name)
        plt.close()

#plotting the original dataset
plot(X, y, "Original Data")
```



[ ]:
```
#initializing the grid
some_grid = np.mgrid[-1.5:2.5:100j, -1:2:100j]

#visualizations for the manual NN
def viz(idx, parameters):
    title = "Manual NN: {:04}".format(idx)
    name = "manual_{:04}.png".format(idx//100)
    file = os.path.join('viz', name)
    pred_prob, _ = forw_prop(np.transpose(some_grid.reshape(2, -1).T), parameters,␣
 ↪structure)
    pred_prob = pred_prob.reshape(pred_prob.shape[1], 1)
    plot(X_test, y_test, title, file_name=file, preds=pred_prob)
```

[ ]:
```
params_values = train(np.transpose(X_train), np.transpose(y_train.reshape((y_train.
 ↪shape[0], 1))), structure, 8000, 0.01, visual=True)
```

[ ]:
```
import keras

#visualizations for the Keras NN
def keras_viz(epoch, logs):
    title = "Keras NN - It: {:02}".format(epoch+1)
    name = "keras_nn_{:02}.png".format(epoch)
    file = os.path.join('viz', name)
    pred_prob = model.predict_proba(some_grid.reshape(2, -1).T, batch_size=32,␣
 ↪verbose=0)
    plot(X_test, y_test, title, file_name=file, preds=pred_prob)

keras_visual = keras.callbacks.LambdaCallback(on_epoch_end=keras_viz)

#I deleted this part so that it looks cleaner but it is the same code as in NN␣
 ↪initialization
model.fit(X_train, y_train, epochs=45, verbose=0, callbacks=[keras_visual])
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print('Testing Accuracy: {0} \nCross-Entropy Loss: {1}'.format(round(accuracy*100,␣
 ↪3), round(loss, 3)))
```

[ ]:
```
#libraries
import glob
from PIL import Image

#paths
viz = "viz/*"
graph = "viz/graph_keras.gif"

#processing the images as GIF
img, *imgs = [Image.open(f) for f in sorted(glob.glob(viz))]
img.save(fp=graph, format='GIF', append_images=imgs, save_all=True, duration=80,␣
 ↪loop=0)
```