Deliverables

For this assignment you need to complete the Julia programs `activations.jl` and `neural_net_training.jl`. For the code review you will also need to prepare (and eventually submit) a Jupyter notebook or other cleaned-up demonstration of your code. Your GitHub repository will become your submission, so commit your work as you go.

Partners and Groups

You are required to work with your assigned partner on this assignment. You are responsible for ensuring that you and your partner both fully understand all aspects of your submission; you are expected to work collaboratively and take responsibility for each other's learning.

You will also be assigned a code-review partner next week. To prepare for that code review, make sure you understand all of your code and that it is readable to people outside your group. See the code review guidelines for more information.

Objectives

The goal of this project is to understand how neural networks compute predictions and how to train a neural network using the backpropagation algorithm. The version of a neural network we are implementing is deliberately inefficient so that we can think through the operations being performed by each neuron. You have therefore been provided with an overly object-oriented neural network that encourages you to do all the computations one node at a time. In future projects we will implement much more efficient neural networks using tensors. This is the last project where you will be expected to deliberately use loops instead of matrix/vector operations.

Provided Functionality

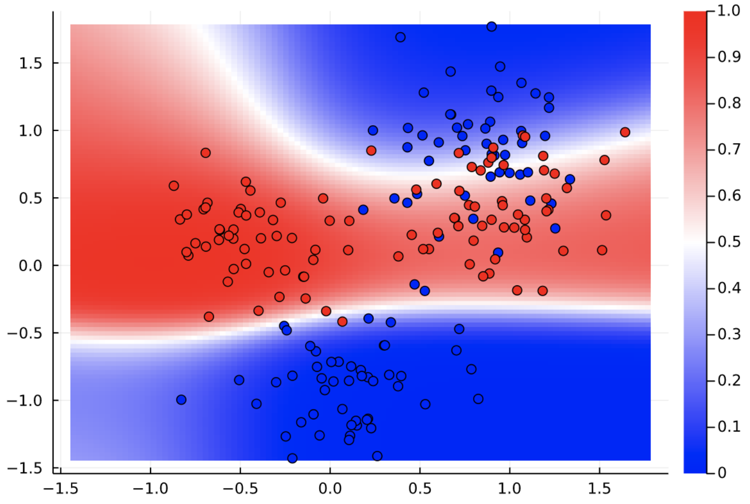
The file `neural_net_model.jl` defines several structs for building a neural network and functions for creating them:

- `Input`: a node that stores an activation but doesn't compute anything.

- `Neuron` a similar struct to what we saw in our single-neuron models, but augmented with fields a and δ for storing activations and derivatives, respectively.
- `NeuralNetwork`: represents a densely-connected neural network that can have arbitrary input and output dimensions, any number and size of hidden layers, and several possible activation functions.
- functions for creating copies of various structs.

You should not need to modify the code in `neural_net_model.jl`, but note that all of the structs are mutable, which will allow the functions you implement to modify their data.

The files `generate_data.jl` and `plot_models.jl` are the same ones we saw in the single-neuron project, but with neural networks, we can use them on more interesting data sets. For example, the following plot is based on data generated with 4 clusters and a neural network with a single 20-neuron hidden layer.



Implementation Tasks

Activation Functions

The program `activations.jl` contains a number of activation functions and their derivatives. The ones we've seen before (linear and sigmoid) are implemented for you. Your first task is to implement tanh and ReLU neurons.

Neural Network Training

Most of your implementation work will happen in `neural_net_training.jl` Many of the functions in this file appear similar to ones we implemented for single-neuron models, but some of the details differ, so pay close attention to the requirements.

The two `predict` functions resemble those we had for single-neuron models, in that one takes a single data point and the other takes a collection of data points. However they differ in that the first, called `predict!` does not return a prediction and instead modifies `node.a` for each node in the network (including the output neurons from which the prediction can later be read). The second `predict` function is much more like last time, in that it produces an array of predictions, and doesn't modify the model.

The `gradient!` function is similar to what we saw last time, except that instead of returning a vector of partial derivatives, it updates `node.`δ for each neuron. This places slightly more work on the `update!` function which has to translate the deltas into weight and bias updates.

In the `train!` function, we have replaced the `iterations` argument with two arguments: `batch_size` and `epochs`. In stochastic gradient descent, each *epoch* processes the entire data set (in a random order). The shuffled data set is split into *batches* where for each batch, we calculate gradients and perform an update. The starting-point code specifies a default batch size of 1. In your implementation, you are encouraged to start by ignoring the `batch_size` argument and assuming that you're always operating on a single data point per batch. We'll discuss the right way to perform batch-processing soon!

The `train!` function has less logging than last time, but you are still expected to keep track of losses (if the user requests it), appending the current MSE loss on the entire data set at the start of training, and at the end of each epoch. The `MSE` function is similar to what you implemented last time, only there can now be multiple outputs! Therefore, we will need to sum the squared errors for each data point before taking a mean across the data set.

Many of the functions you will be implementing can get rather complicated. You are encouraged to think about helper functions or other ways of decomposing them into more manageable tasks.

Testing

You have been provided with two Jupyter notebooks: one called `scratchwork` that is intended for trying things out as you go, and one called `testing` that is intended for exercising your implementations once they're mostly working. The expectation is that the `testing` notebook will be cleaned up as a part of your submission so that you can show off the functionality of your implementation to your code-review partner and for grading. The first thing you should do for testing is compare your implementation to your single-neuron model. If you create a neural network with zero hidden neurons and one output neuron, its behavior should be identical to what we implemented before.

Afterwards, you should include multiple examples of both larger neural networks and more-interesting data sets. You should plot 1-D and 2-D outputs, but you should also be testing on higher-dimensional inputs and outputs, and thinking about how to demonstrate that it's working! You may want to implement something similar to the `accuracy` function we had for single-neuron models (or other types of tests) but think about how they should generalize to this case!

Optional Extensions

There are a number of ways to go beyond the basics with this project, and you are encouraged to explore whatever aspects you find most interesting! Here are a few suggestions for things to try:

- Read in one of the data sets we worked with last week and try your neural network out on some real data (Julia has lots of helpful packages, such as `DataFrames.jl` which provides much of the same functionality as `Pandas`).
- Implement `batch_size` > 1. You will need to think through how to keep track of activations and deltas, since the `NeuralNetwork` struct only saves individual floating-point values. We'll learn much more about batch processing soon.