Final Project

In this project we will implement a simple 3-layer neural network from scratch. Implementing a network from scratch at least once is an extremely valuable exercise. It helps you gain an understanding of how neural networks work, and that is essential for designing effective models.

Generating a dataset

Let's start by generating a dataset we can play with. scikit-learn has some useful dataset generators, so we don't need to write the code ourselves. We will go with the make_moons function as follows:

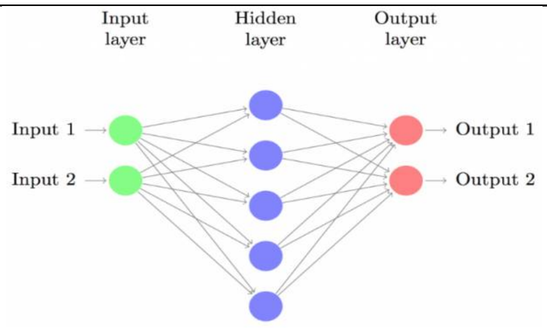
```
# Generate a dataset and plot it
np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)
```

The dataset we generated has two classes, plotted as red and blue points. You can think of the blue dots as male patients and the red dots as female patients, with the x- and y- axis being medical measurements.

Our goal is to train a classifier that predicts the correct class (male of female) given the x- and y- coordinates. Note that the data is not linearly separable, we can't draw a straight line that separates the two classes..In fact, that's one of the major advantages of Neural Networks. You don't need to worry about feature engineering. The hidden layer of a neural network will learn features for you.

Training a Neural Network

Let's now build a 3-layer neural network with one input layer, one hidden layer, and one output layer. The number of nodes in the input layer is determined by the dimensionality of our data, 2. Similarly, the number of nodes in the output layer is determined by the number of classes we have, also 2. (Because we only have 2 classes we could actually get away with only one output node predicting 0 or 1, but having 2 makes it easier to extend the network to more classes later on). The input to the network will be x- and y- coordinates and its output will be two probabilities, one for class 0 ("female") and one for class 1 ("male"). It looks something like this:



We can choose the dimensionality (the number of nodes) of the hidden layer. The more nodes we put into the hidden layer the more complex functions we will be able fit. But higher dimensionality comes at a cost. First, more computation is required to make predictions and learn the network parameters. A bigger number of parameters also means we become more prone to overfitting our data.

How to choose the size of the hidden layer? While there are some general guidelines and recommendations, it always depends on your specific problem and is more of an art than a science.

We also need to pick an *activation function* for our hidden layer. The activation function transforms the inputs of the layer into its outputs. A nonlinear activation function is what allows us to fit nonlinear hypotheses. Common chocies for activation functions are tanh, the sigmoid function, or ReLUs. We will use ***tanh***, which performs quite well in many scenarios. A nice property of these functions is that their derivate can be computed using the original function value.
Because we want our network to output probabilities the activation function for the output layer will be the softmax, which is simply a way to convert raw scores to probabilities.

How our network makes predictions

Our network makes predictions using forward propagation, which is just a bunch of matrix multiplications and the application of the activation function(s) we defined above. If x is the 2-dimensional input to our network then we calculate our prediction $\hat{y}$ (also two-dimensional) as follows:

$$\begin{aligned} z_1 &= xW_1 + b_1 \\ a_1 &= \tanh(z_1) \\ z_2 &= a_1W_2 + b_2 \\ a_2 &= \hat{y} = \mathrm{softmax}(z_2) \end{aligned}$$

z_i is the input of layer i and a_i is the output of layer i after applying the activation function. W_1, b_1, W_2, b_2 are parameters of our network, which we need to learn from our training data. You can think of them as matrices transforming data between layers of the network.

Learning the Parameters

Learning the parameters for our network means finding parameters ($\boldsymbol{W_1}, \boldsymbol{b_1}, \boldsymbol{W_2}, \boldsymbol{b_2}$) that minimize the error on our training data. But how do we define the error? We call the function that measures our error the *loss function*. A common choice with the softmax output is the categorical cross-entropy loss (also known as negative log likelihood). If we have N training examples and C classes then the loss for our prediction $\boldsymbol{\hat{y}}$ with respect to the true labels $\boldsymbol{y}$ is given by:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i}$$

The formula looks complicated, but all it really does is sum over our training examples and add to the loss if we predicted the incorrect class. The further away the two probability distributions $\boldsymbol{y}$ (the correct labels) and $\boldsymbol{\hat{y}}$ (our predictions) are, the greater our loss will be. By finding parameters that minimize the loss we maximize the likelihood of our training data.

We can use gradient descent to find the minimum and I will implement the most vanilla version of gradient descent, also called batch gradient descent with a fixed learning rate. Variations such as SGD (stochastic gradient descent) or minibatch gradient descent typically perform better in practice. So if you are serious you'll want to use one of these, and ideally you would also decay the learning rate over time.

As an input, gradient descent needs the gradients (vector of derivatives) of the loss function with respect to our parameters: $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial W_2}$, , $\frac{\partial L}{\partial b_2}$. To calculate these gradients we use the famous *backpropagation algorithm*, which is a way to efficiently calculate the gradients starting from the output. Applying the backpropagation formula we find the following:

$$\delta_3 = \hat{y} - y$$
$$\delta_2 = (1 - \tanh^2 z_1) \circ \delta_3 W_2^T$$
$$\frac{\partial L}{\partial W_2} = a_1^T \delta_3$$
$$\frac{\partial L}{\partial b_2} = \delta_3$$
$$\frac{\partial L}{\partial W_1} = x^T \delta 2$$
$$\frac{\partial L}{\partial b_1} = \delta 2$$

Implementation

Now we are ready for our implementation. We start by defining some useful variables and parameters for gradient descent:

```
num_examples = len(X) # training set size
nn_input_dim = 2 # input layer dimensionality
nn_output_dim = 2 # output layer dimensionality

# Gradient descent parameters (I picked these by hand)
epsilon = 0.01 # learning rate for gradient descent
reg_lambda = 0.01 # regularization strength
```

First let’s implement the loss function we defined above. We use this to evaluate how well our model is doing:

```
# Helper function to evaluate the total loss on the dataset
def calculate_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation to calculate our predictions
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Calculating the loss
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    # Add regulatization term to loss (optional)
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss
```

We also implement a helper function to calculate the output of the network. It does forward propagation as defined above and returns the class with the highest probability.

```
# Helper function to predict an output (0 or 1)
def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```

Finally, here comes the function to train our Neural Network. It implements batch gradient descent using the backpropagation derivates we found above.

```
# This function learns parameters for the neural network and returns the model.
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradient descent
# - print_loss: If True, print the loss every 1000 iterations
def build_model(nn_hdim, num_passes=20000, print_loss=False):
```

```
# Initialize the parameters to random values. We need to learn these.
np.random.seed(0)
W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
b1 = np.zeros((1, nn_hdim))
W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
b2 = np.zeros((1, nn_output_dim))

# This is what we return at the end
model = {}

# Gradient descent. For each batch...
for i in xrange(0, num_passes):

    # Forward propagation
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # Backpropagation
    delta3 = probs
    delta3[range(num_examples), y] -= 1
    dW2 = (a1.T).dot(delta3)
    db2 = np.sum(delta3, axis=0, keepdims=True)
    delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
    dW1 = np.dot(X.T, delta2)
    db1 = np.sum(delta2, axis=0)

    # Add regularization terms (b1 and b2 don't have regularization terms)
    dW2 += reg_lambda * W2
    dW1 += reg_lambda * W1

    # Gradient descent parameter update
    W1 += -epsilon * dW1
    b1 += -epsilon * db1
    W2 += -epsilon * dW2
    b2 += -epsilon * db2

    # Assign new parameters to the model
    model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}

    # Optionally print the loss.
    # This is expensive because it uses the whole dataset.
    if print_loss and i % 1000 == 0:
       print "Loss after iteration %i: %f" %(i, calculate_loss(model))

return model
```

What you should do:

Here are some exercises you should do them as your final project:

1. Build a model with a 3-dimensional hidden layer and plot the decision boundary.
2. In the above we picked a hidden layer size of 3. Let’s now get a sense of how varying the hidden layer size (1,2,3,4,5,20,40) affects the result. For each hidden layer size, please plot the decision boundary.
3. Instead of batch gradient descent, use minibatch gradient descent (more info) to train the network. Minibatch gradient descent typically performs better in practice.

4. We used a fixed learning rate for gradient descent. Implement an annealing schedule for the gradient descent learning rate.
5. We used a tanh activation function for our hidden layer. Experiment with other activation functions (some are mentioned above). Note that changing the activation function also means changing the backpropagation derivative.
6. Extend the network from two to three classes. You will need to generate an appropriate dataset for this.
7. Extend the network to four layers. Experiment with the layer size. Adding another hidden layer means you will need to adjust both the forward propagation as well as the backpropagation code.

Good luck