Using Neural Networks to Classify American Sign Language (ASL) Signs

Chris Walker

Problem Statement and Introduction

Machine learning plays a unique role in improving accessibility in digital communication. This includes services like text-to-speech, language translation, and alt-text generation. Machine learning could also be used to detect American Sign Language (ASL) signs and convert them to text data. **Sign language detection is not currently implemented into the modern ecosystem of video call tools.** Imagine if FaceTime was able to convert signs to text in real time. An ASL user could sign into the camera, convert their message to text, and have it read back as text-to-speech. This could create more natural telecommunication for people with a hearing disability. With this concept in mind, I set out to create a primitive sign language detection model by implementing the learning algorithms from scratch.

Beyond this report, this project is an R package (called `deepspace`) with a C++ back end which aids in the collection, preprocessing, and modeling of images for the purpose of classification. I hosted and documented this package on my personal GitHub profile here. Additionally, it is important to set reasonable expectations for this class project; while a fully featured model would capture the entirety of ASL, this model classifies several letters of the ASL alphabet as a proof of concept. Users can install this package on their system and launch a webcam stream to classify a handful of ASL digits if they choose.

Data Sources

This model was exclusively trained on images captured using my webcam. At the time of capture, I pre-select a sign which I was going to perform. I called a function called `deepspace::capture_images()` which launches a webcam session and captures n images in rapid succession. I performed the sign in a variety of lighting conditions and angles while the webcam is capturing. Because I *pre-selected* the sign I am capturing, there was no need to label images individually as all images are stored in a directory that corresponds to that sign. I repeated this process for ASL letters A, B, and C.

<insert example images here>

Stemming from other assignments this semester, I *vectorized* all images captured by the webcam. I can point to a directory which corresponds to a given sign using `deepspace::load_images()`. This function loads, vectorizes, and labels all images as R matrices for use in training. The matrices of images were split into training and testing for validation.

<discuss image dimensions here>

Methodology

The process of capturing images as part of this process is fairly linear. In general, I captured images following the procedure described in *Data Sources* and stored them for model validation. However, the process of modeling the images was more iterative in nature. It involved making updates to the underlying algorithm while also tuning hyperparameters such as the number of hidden layer neurons, the learning rate, and preprocessing methodologies (i.e., normalization).

While the code package only contains four core functions used for image capture and model estimation, the package contains many other functions designed to aid the model estimation procedure written in C++. Most of these functions correspond to a specific linear algebra operation needed to fit and validate a neural network.

`normal_matrix(row, col)`

This function initializes a matrix with dimensions *row* and *col* such that each element is randomly drawn from a normal distribution with mean zero and a standard deviation of one. While other methods exist to initialize a neural network for training, I find that randomly distributed noise works well in many cases. I also supply a random seed which ensures I can reproduce random noise and achieve the same estimations given the same training data and hyperparameter configurations. Let X be a matrix with i rows and j columns.

$$X_{ij} \approx \mathcal{N}(0, 1)$$

`dot(X, Y)`

Dot-products are an extremely common operation in machine learning and fitting a neural network is no exception. This function considers two matrices where matrix X must have the same number of rows as matrix Y has columns. It returns a matrix with the number of rows of X and the number of columns as Y. This function is highly optimized and benchmarks at speeds equal to or faster than R's base dot-product implementation.

$$XY$$

`transpose(X)`

Like dot-products, the transpose operation is a core linear algebra operation. This function simply considers matrix X and returns a new matrix which has as many rows as X has columns and as many columns as X has rows. It inverts the positions of row-column pairs as iterates across the matrix.

$$X^T$$

`multiply(X, Y)`

Unlike the dot-product, this function performs element-wise multiplication of two matrices. Because this is an element level multiplication, matrices X and Y must share the same dimensions. As such, this function returns a new matrix with the same dimensions as the input matrices.

$$X \times Y$$

`subtract(X, Y)`

This function operates like the `multiply()` function to perform element-wise subtraction of two matrices. Thus, the input matrices X and Y must again share the same dimensions such that a new matrix with the same dimensions can be returned. It is important to note that matrix Y is always subtracted *from* matrix X.

$$X - Y$$

`sub_scalar(x, Y)`

Unlike `subtract()`, this function allows for the subtraction of a matrix *from* a scalar value. Because C++ is highly procedural, this function is needed to define behavior separately from element-wise subtraction. It accepts a scalar parameter x and a matrix Y and returns a new matrix with the same dimensions as Y.

$$x - Y$$

`mul_scalar(x, Y)`

Like `sub_scalar()`, this function implements a scalar multiplication procedure. However, because it is multiplication the explicit order of the operation does not matter. In either case, this function accepts a scalar x and a matrix Y and returns a new matrix with the same dimensions as Y.

$$x \times Y$$

`add_ones(X)`

This function does not implement a standard linear algebra operation but is rather a convenience functionality used during network feed-forward to introduce a new column vector of ones to a matrix of arbitrary dimensions. This is needed to allow for a *bias* to be estimated for each layer in the network as we will see in later sections. Let X be a matrix with i rows and j columns. Matrix Y represents matrix X with a column vector of ones appended to the right side.

$$Y = \begin{bmatrix} X_{11} & \dots & X_{1j} & 1 \\ \vdots & \ddots & \vdots & 1 \\ X_{i1} & \dots & X_{ij} & 1 \end{bmatrix}$$

`activation(X)`

This function considers element-wise activation during model feed-forward. Specifically, it implements the logistic function (sometimes called the sigmoid function). This is needed to take unbounded values and *squish* them between values of zero and one. It is the same function that logistic regression uses to convert $X\beta$ values to a probability. This is necessary for neural networks to keep values sensible as they traverse the network. It also helps demonstrate the similarities between neural networks and logistic regression!

$$X_a = \frac{e^X}{1 + e^X}$$

`feed_forward(X)`

The concept of *feeding forward* is an important concept in neural network fitting and prediction. The core concept it is to take the dot product of the weight matrix with data from the prior step. The resulting matrix will have the same rows as the input matrix and the same number of columns as there are classes in our training label matrix (one-hot encoded). The `deepspace` package fits neural networks with and two hidden layers each with an arbitrary number of neurons.

Let our training data be denoted as X, the weights for hidden layer one as H_1, the weights for hidden layer two as H_2, post activation matrices are denoted as A and finally the weights for the output layer as W. We also need to denote functions $f()$ which corresponds to `add_ones()` from above. Other functions such as `activation()` and `dot()` are also used in this operation in code but those have been obscured by the notation here.

$$Z_1 = XH_1$$

$$A_1 = \frac{e^{f(Z_1)}}{1 + e^{F(Z_1)}}$$

$$Z_2 = A_1 H_2$$

$$A_2 = \frac{e^{f(Z_1)}}{1 + e^{f(Z_1)}}$$

$$Z_3 = A_2 W$$

$$A_3 = \frac{e^{f(Z_3)}}{1 + e^{f(Z_3)}}$$

`gradient()`

This function is the heart of the back propagation algorithm. It is used to compute the gradient such that the weights and biases can be adjusted to maximize <what are we maximizing?> through the process of gradient descent. It represents the derivative of the cost function with respect to each term in the network. If a gradient can be found, we *subtract* the gradient from the current matrix of weights to better the model fit. Let W be a matrix of weights for the current layer, D is a matrix of differences, and A is the activation matrix from the current layer.

$$G = WD^T \times (A \times (1 - A))$$

Model Fitting Procedure

The core function of this package is `deepspace::fit_network()`. This function accepts training data and labels and fits a neural network. During initialization, all weight matrices are constructed using `normal_matrix()` in accordance with network design criteria. Suppose we have a matrix of training data denoted as X with 6 predictors and n observations. Additionally, we will be fitting a binary classification outcome (which is one hot encoded in the label matrix Y with 2 columns). The network will have two hidden layers each with 3 neurons. Matrix T represents the training matrix X with an intercept column vector added. All W matrices represent weights for the hidden layers and the output layer.

$$\text{Let } f(i,j) = X_{i \times j} \approx \mathcal{N}(0,1)$$

$$T = \begin{bmatrix} X_{1,1} & ... & X_{1,6} & 1 \\ \vdots & \ddots & \vdots & 1 \\ X_{n,1} & ... & X_{n,6} & 1 \end{bmatrix}$$

$$W_1 = f(7, 3)$$

$$W_2 = f(6, 3)$$

$$W_3 = f(6, 2)$$

The learning procedure takes this baseline network and begins backpropagation. It repeats this process for a predetermined number of epochs or until some other stop condition is applied (i.e., a minimum change denoted as ϵ). Matrices denoted as D are known as *difference* matrices.

$$D_3 = A_3 - Y$$

$$D_2 = W_3 D_3^T \times (A_2 \times (1 - A_2))$$

$$D_1 = W_2 D_2^T \times (A_1 \times (1 - A_1))$$

We can now apply these matrices to update the three weight matrices. An additional parameter is introduced here as γ which represents the learning rate. Currently, `deepspace` supports only a constant learning rate across all three weight matrix computations.

$$W_1 = W_1 - \gamma X^T D_1$$

$$W_2 = W_2 - \gamma A_1^T D_2$$

$$W_3 = W_3 - \gamma A_2^T D_3$$

Hypothesis

Before setting out on model estimations, I wanted to outline some key hypothesis statements to better my intuition for these models. First, I expect that there are diminishing marginal returns to model performance for an increase in the number of hidden layer neurons. Therefore, a network with 2 hidden layers each with 20 neurons may be sufficient. Additionally, I expect that the logistic activation function will be sufficient for model training.

While an increased number of neurons, hidden layers, or training epochs may marginally improve performance, I expect that a fairly simple model will perform adequately for the purposes of image classification. I detailed the performance of each model training configuration and identified a candidate model in the section below.

Evaluation and Final Results

<robust discussion of results>