Neural Networks for Multi-Class Classification

Joey Shi

July 2020

1 Abstract

In this paper, we will be formalizing the process of training a neural network classification model. We will be covering the softmax loss function, computing the gradient of the network, and gradient descent.

2 Notation

1. X is a $n \times d$ feature matrix of real numbers, such that each row $x_i \in \mathbb{R}^d$ is an example.
2. y is an n-dimensional label vector, such that each $y_i \in \{1, 2, \ldots, k_L\}$ is a class label.
3. $h(z)$ is a non-linear function. We will call it an activation function.
4. $W^{(l)}$ is a $k_l \times k_{l-1}$ weight matrix.
5. $b^{(l)}$ is a k_l-dimensional bias vector.
6. $p(y_i \mid D, x_i)$ is the probability of predicting y_i from x_i and parameters D
7. $\delta_{a,b} = 1$ if $a = b$, $\delta_{a,b} = 0$ otherwise [Kronecker delta function].

3 Maximum Likelihood Estimate

We begin by defining our model's probability mass distribution:

$$p(y_i \mid D, x_i) = \frac{\exp\left(z_{i,y_i}^{(L)}\right)}{\sum_{c=1}^{k} \exp\left(z_{i,c}^{(L)}\right)}, \qquad \left[z_i^{(L)} \text{ is defined in section 4}\right]$$

We choose this probability mass function for our classification problem because of its behaviour: $p(y_i \mid D, x_i)$ is larger if $z_{i,y_i}^{(L)} = \max_c z_{i,c}^{(L)}$ and small otherwise. For our model, we want to maximize this probability across all training example, so we maximize $p(y \mid D, X)$. Assuming that each example is i.i.d., we can equivalently minimize $-\log(p(y \mid D, X))$ to avoid underflow during computation.

$$\begin{aligned}
-\log(p(y \mid D, X)) &= -\log\left(\prod_{i=1}^{n} p(y_i = c \mid D, x_i)\right) \\
&= \sum_{i=1}^{n} -\log(p(y_i \mid D, x_i)) \\
&= \sum_{i=1}^{n} -z_{i,y_i}^{(L)} + \log\left(\sum_{c=1}^{k} \exp\left(z_{i,c}^{(L)}\right)\right) \qquad \text{[Softmax Loss Function]}
\end{aligned}$$

4 Objective Function

Consider the following feed-forward neural network. For an input layer defined by x_i, our hidden layers are defined by the activation vectors $a_i^{(l)}$ and our output layer is defined by $z_i^{(L)}$.
Note: We add hidden layers so our model can learn about latent factors in the data during training.

$$\begin{aligned} a_i^{(0)} &= x_i & z_i^{(1)} &= W^{(1)} a_i^{(0)} + b^{(1)} \\ a_i^{(1)} &= h\left(z^{(1)}\right) & z_i^{(2)} &= W^{(2)} a_i^{(1)} + b^{(2)} \\ &\vdots & &\vdots \\ a_i^{(L-1)} &= h\left(z^{(L-1)}\right) & z_i^{(L)} &= W^{(L)} a_i^{(L-1)} + b^{(L)} \end{aligned}$$

After some weight initialization, given an example x_i, our model would predict $\hat{y}_i = \arg\max_c z_{i,c}^{(L)}$.
Using softmax loss, our objective function becomes

$$f(W^{(1)}, b^{(1)}, \ldots, W^{(L)}, b^{(L)}) = \sum_{i=1}^{n} \left[-z_{i,y_i}^{(L)} + \log\left(\sum_{c=1}^{k} \exp\left(z_{i,c}^{(L)}\right) \right) \right]$$

5 Gradients of the Objective Function

Theorem 1. *Let $A^{(l)}$ be a matrix such that each row is an activation vector $a_i^{(l)}$ and let $Z^{(l)}$ be defined similarly. Define a new matrix R by the recursion relation $R^{(l-1)} = R^{(l)} W^{(l)} \circ h'\left(Z^{(l-1)}\right)$ with a base case of $R^{(L)}$, where $r_{i,c}^{(L)} = p(y_i = c \mid D, x_i) - \delta_{y_i,c}$. Then the derivatives of the weights and biases of our network are given by*

$$\frac{\partial f}{\partial W^{(l)}} = \left(R^{(l)}\right)^T A^{(l-1)} \qquad \frac{\partial f}{\partial b^{(l)}} = \sum_{i=1}^{n} r_i^{(l)}$$

Gradient of $W^{(L)}$ and $b^{(L)}$

$$\begin{aligned} \frac{\partial f}{\partial w_{c,j}^{(L)}} &= \sum_{i=1}^{n} \left[-a_{i,j}^{(L-1)} \delta_{y_i,c} + \frac{1}{\sum_{c_0=1}^{k_l} \exp\left(z_{i,c_0}^{(L)}\right)} \cdot \exp\left(z_{i,c}^{(L)}\right) \cdot a_{i,j}^{(L-1)} \right] && \text{[Chain Rule]} \\ &= \sum_{i=1}^{n} [p(y_i = c \mid D, x_i) - \delta_{y_i,c}] a_{i,j}^{(L-1)} \\ &= \sum_{i=1}^{n} r_{i,c}^{(L)} a_{i,j}^{(L-1)} \\ \frac{\partial f}{\partial W^{(L)}} &= \left(R^{(L)}\right)^T A^{(L-1)} \\ \frac{\partial f}{\partial b_c^{(L)}} &= \sum_{i=1}^{n} r_{i,c}^{(L)} \cdot 1 && \text{[Similar work as above]} \\ \frac{\partial f}{\partial b^{(L)}} &= \sum_{i=1}^{n} r_i^{(L)} \end{aligned}$$

Gradient of $W^{(L-1)}$ and $b^{(L-1)}$

$$\begin{aligned}
\frac{\partial f}{\partial w_{j,k}^{(L-1)}} &= \sum_{i=1}^{n}\sum_{c=1}^{k_L}[p(y_i = c \mid D, x_i) - \delta_{y_i,c}] \cdot w_{c,j}^{(L)} h'\left(z_{j,k}^{(L-1)}\right) \cdot a_{j,k}^{(L-2)} && \text{[Chain Rule]} \\
&= \sum_{i=1}^{n}\sum_{c=1}^{k_L} r_{i,c}^{(L)} \cdot w_{c,k}^{(L)} h'\left(z_{i,k}^{(L-1)}\right) \cdot a_{i,k}^{(L-2)} \\
&= \sum_{i=1}^{n}\left[\left(W^{(L)}\right)^T r_i^{(L)} \circ h'\left(z_i^{(L-1)}\right)\right] \cdot a_i^{(L-2)} \\
&= \sum_{i=1}^{n} r_i^{(L-1)} \cdot a_i^{(L-2)} \\
&= \left(R^{(L-1)}\right)^T A^{(L-2)} \\
\frac{\partial f}{\partial b_k^{(L-1)}} &= \sum_{i=1}^{n} r_{i,k}^{(L-1)} \cdot 1 && \text{[Similar work as above]} \\
\frac{\partial f}{\partial b^{(L-1)}} &= \sum_{i=1}^{n} r_i^{(L-1)}
\end{aligned}$$

The following derivations are trivial and left as an easy exercise for the reader

6 Stochastic Gradient Descent Algorithm

Define D to be a vector containing all the weights and biases in our model. Then $f(D)$ is our loss function. Given E epoches, B number of batches, and a learning rate α^t, we proceed with minimizing $f(D)$ by stochastic gradient descent. Randomly initialize D^0. For each epoch, for each batch, sample $\lfloor n/B \rfloor$ training examples without replacement and compute $\nabla f(D^t)$. Then,

$$D^{t+1} = D^t - \alpha^t \nabla f(D^t)$$

We will use the parameters D^E to make predictions for the model.

7 Using the Model

Load a dataset X and y, shuffle the examples, and split the dataset into a training set X_{train} and y_{train} and testing set X_{test} and y_{test}. Instantiate our model with a specific number of hidden units for each hidden layer. Fit our model over X_{train} and y_{train}. Compute training and testing error to evaluate the quality of our model. We can then pass in our own example for our model to predict. Pseudocode is provided below.

Golden Rule: The testing set must not affect the training phase in any way.

```
X, y = load(dataset)
X_train, y_train, X_test, y_test = shuffle_and_split(X, y)
model = NNClassifier(hidden_layers)
model.fit(X_train, y_train)

training_error = count(model.predict(X_train) != y_train) / n_train
testing_error = count(model.predict(X_test) != y_test) / n_test

for any custom example x, we can predict y_hat = model.predict(x)
```