CS 189 Project T - Perceptrons

Daniel Abraham, Ganesh Vurimi,
Matthew Bronars, Warren Deng, Rasika Iyer

November 2020

1 What are Perceptrons?

Perceptrons are the building blocks of neural networks that can be derived using basic linear algebra tools. In an analogy to human biology, perceptrons can be thought of as a single neuron; they receive various inputs and fire after passing a certain threshold. We normally use perceptrons for the problem of binary classification. This is when we are trying to assign a class/label, typically represented as either positive or negative scalar, to some new input. For example, if we had a bunch of old emails and we wanted to classify new, incoming emails as either spam or ham (not spam), we could train a perceptron on the old emails. Then, we could break down new emails into a finite number of parts (called features), feed these parts into our trained perceptron as input, and receive an output labelling the email as either spam or ham. More specifically, a perceptron has inputs 1 to n (these are the features) which we can represent as a n dimensional vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix}$$

We assign each input a weight w_i. Stacking these individual weights in a vector, we have another n dimensional vector

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ ... \\ w_n \end{bmatrix}$$

We are interested in the weighted sum of the inputs, which is the same as taking the dot product between the inputs and their weights

$$\sum_{i=1}^{n} x_i * w_i = \mathbf{x}^T \mathbf{w}$$

Using this weighted sum, we apply an activation function $f(x)$ in order to generate the predicted class/label of the input. In the future when you learn about neural networks, this activation function can be picked from a wide variety of functions. In the case of perceptrons and binary classification, we use the activation function

$$f(x) = \begin{cases} +, \mathbf{x}^T \mathbf{w} \geq 0, \\ -, \mathbf{x}^T \mathbf{w} < 0 \end{cases}$$

This activation function separates any input into one of two classes: a positive value if the dot product of $\mathbf{w}$ and $\mathbf{x}$ is non-negative, and a negative value if the dot product is negative. Note that we could have also picked an activation function that outputs 1 and 0 (step function) instead of 1 and -1 since in both cases we have two labels.

By setting the correct weights $\mathbf{w}$ for our data, we can classify any new input $\mathbf{x}^*$ by doing what we did previously: computing the dot product and applying the activation function. To generate the weights for a set of data, we run the Perceptron Learning Algorithm (described in full in the following section) on our entire data set (for the email example, this would be all of the old emails we have).
While the perceptron does not seem very complicated, we have essentially built a basic neural network. In fact, perceptrons are just single layer neural networks; by stringing together multiple perceptions, we can build more complicated neural networks that aren't restricted to just binary classification (these are called multilayer perceptrons and will be covered in a future note). Understanding perceptrons and their basic properties is key to grasping the concepts of neural networks and machine learning, and all it takes is a little understanding of linear algebra.

2 Modeling a Perceptron

2.1 Update Algorithm

The Perceptron Learning Algorithm can be used to determine the weights of a perceptron, where r is a hyper-parameter called the learning rate that can take on any value between 0 and 1. We'll learn more about hyper-parameters later on in the course, but for the purpose of this algorithm, the learning rate allows us to control how much we care about incorrectly classifying a point when updating our weights. For simplicity of notation, we also assume that we are working with raw data $\mathbf{x}$ instead of features lifted from the data. We define a data set D to consist of training points $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, where $\mathbf{x}_i$ is the i-th input vector and y_i is its output/desired classification. For binary classification on a set of linearly separable data centered around the origin, the algorithm is as follows:

1. Initialize the weight vector $\mathbf{w}$ to the zero vector $\mathbf{0}$
2. For each training point ($\mathbf{x_i}$, y_i) in dataset D:
   (a) Calculate the current classification using weights $\mathbf{w}$

$$y_i^* = \begin{cases} 1, \mathbf{w}^T\mathbf{x_i} \geq 0, \\ -1, \mathbf{w}^T\mathbf{x_i} < 0 \end{cases}$$

   (b) If the point is incorrectly classified, update weights using

$$\mathbf{w}_{new} = \mathbf{w}_{old} + r * y_i^* * \mathbf{x_i}$$

3. Repeat step 2 until no weights are updated

2.2 Alternate Derivation

We can derive the Perceptron Learning Algorithm from an alternative viewpoint using gradient descent to optimize some cost function. While gradient descent may sound difficult, this derivation only requires some basic multi-variable calculus techniques. Our cost function should penalize us for misclassifying points, and do nothing for correctly classifying points. Then if we minimize this cost function with respect to the weights, we have found our optimal weight vector (since we would have the best weight vector in terms of correctly classifying our training set). Let's define our cost function as

$$C(w) = \sum_{i \epsilon V} -y_i * \mathbf{x}_i^T * \mathbf{w}$$

where V is the set of all indices such that $-y_i * \mathbf{x}^T * \mathbf{w} < 0$ (incorrectly classified). If all of our points are correctly classified, V is an empty set so our cost ends up being 0. However, if V is not empty, our costs

ends up being a positive ($y_i \mathbf{x}^T \mathbf{w} < 0$ so $-y_i \mathbf{x}^T \mathbf{w} > 0$) linear combination of the weight vector $\mathbf{w}$. Using what we know from multi-variable calculus, we can take the gradient of this cost function with respect to $\mathbf{w}$ in order to find the direction of steepest ascent. The gradient is simply

$$\nabla C(w) = \sum_{i \epsilon V} -y_i * \mathbf{x}_i^T$$

We want to travel in the opposite of this direction since we are minimizing the cost function (direction of steepest descent). Thus, to update our weights we can use

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \sum_{i \epsilon V} y_i * \mathbf{x}_i^T$$

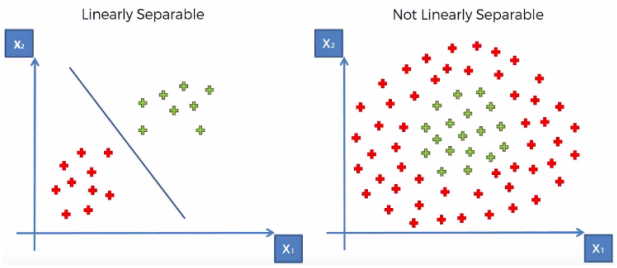
We can do this update step iteratively until our weights converge. Sometimes, we might not want to travel the entire gradient in each update step; we may want to just travel a small portion of it in order to not overshoot the optimal weight. We can multiply the gradient by parameter r, which we refer to as the learning rate/ step size, to accommodate for this. Again, r is bounded by 0 and 1. Thus our final update algorithm is

$$\mathbf{w}_{new} = \mathbf{w}_{old} + r * \sum_{i \epsilon V} y_i * \mathbf{x}_i^T$$

Looking closely, this is very similar to same update scheme that we use in step $2b$ of the original Perception Learning Algorithm. However, here we update the weight vector for all training points simultaneously instead of updating each weight individually for every training point

2.3 Convergence

The Perceptron Learning Algorithm is guaranteed to converge under a few assumptions of the training data. Particularly, it will fail if the data is not linearly separable. For two dimensional input, linear separability can be thought of as if we can draw a line between the data points such that all data with one label lie on one side of the line and all data with the other label lie on the other side.



If the data is not linearly separable, the Perceptron Learning Algorithm will oscillate in the updating step, and will not converge. If the data is linearly separable, the algorithm still is not guaranteed to find an optimal solution but it will find some dividing hyperplane to separate the data (line in the case of two dimensional input). In fact, there is an upper bound to the maximum iterations before convergence, but this is beyond the scope of the note.

2.4 Bias

Previously we had assumed that all of our data was centered around the origin. This means that if we were to draw a dividing hyperplane to linearly separate our data, the hyperplane would intersect with the origin. The equation for this hyperplane is actually just the weight vector $\mathbf{w}$. However, we might have linearly

separable data that requires a hyperplane that does not go through the origin. Our algorithm currently would fail to learn the correct boundary on this sort of data. To amend this, we can add a bias term to the weight vector. What we want is to shift our computed value $\mathbf{w}^T\mathbf{x}$ by some scalar value b.

$$\mathbf{w}^T\mathbf{x} + b$$

This is now the same hyperplane, but shifted (when $b = 0$, the hyperplane passes through the origin). We can represent this value b as an extra term in our weight vector called w_{n+1}

$$\mathbf{w}_{bias} = \begin{bmatrix} w_1 \\ w_2 \\ ... \\ w_n \\ w_{n+1} \end{bmatrix}$$

Because we changed added one to the dimension of $\mathbf{w}$, we have to add an extra element to all of our input vectors $\mathbf{x}$. This is simply the constant term 1

$$\mathbf{x}_{bias} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \\ 1 \end{bmatrix}$$

Now, taking the dot product of $\mathbf{w}_{bias}$ and $\mathbf{x}_{bias}$ we have the relation we wanted previously

$$\mathbf{w}_{bias}^T\mathbf{x}_{bias} = \mathbf{w}^T\mathbf{x} + 1 * w_{n+1} = \mathbf{w}^T\mathbf{x} + b$$

We are now free to classify data regardless of whether it is origin centered.

2.5 Multiclass Perceptron

Our perceptron currently only works for binary classification, where we can only predict one of two labels. We can predict more labels by slightly modifying the way we represent things. We assume that the possible labels range from 0 to m. For binary classification, we could get away with only using one weight vector because classified based off of a threshold between $\mathbf{w}_i^T\mathbf{x}$. For more than two outputs, instead of a single weight vector we have a weight vector for each possible class for a total of m vectors. To get the predicted label of some input $\mathbf{x}$, we compute

$$y = \underset{i=0,1,...m}{\arg\max}\, \mathbf{w}_i^T\mathbf{x}$$

This means that we take the dot product between $\mathbf{x}$ and each weight vector, and classify based on the weight vector that produced the largest dot product. The update step for each weight vector is also similar to that of binary classification, but with some slight modifications

1. Initialize each weight vector $\mathbf{w}_j$ to the zero vector $\mathbf{0}$
2. For each training point ($\mathbf{x_i}$, y_i) in dataset D:
   (a) Calculate the current classification

$$y_i^* = \underset{i=0,1,...m}{\arg\max}\, \mathbf{w}_i^T\mathbf{x}$$

   (b) If the point is incorrectly classified:

i. Update the desired weight vector $\mathbf{w}^*$ using

$$\mathbf{w}^*_{new} = \mathbf{w}^*_{old} + r * \mathbf{x_i}$$

ii. Update the rest of the weight vectors using

$$\mathbf{w}_{new} = \mathbf{w}_{old} - r * \mathbf{x_i}$$

3. Repeat step 2 until no weights are updated

Here when we misclassify a training point, we increase the value of the correct label's weight vector and decrease the rest. This way when we compute the argmax step again, the value of the dot product between the input and the correct weight vector should be higher while the rest are lower.

3 Application: Audio Classification

Audio classification is a classic ML (machine learning) problem that can be solved with different types of ML algorithms. We will focus on binary classification using a perceptron. Specifically, let us pretend that we have a device that needs to classify the word **dog** from the word **cat**.

To begin working on this problem, we first need to accumulate many data points of humans saying the word dog and the word cat. When we record someone saying such a word, much pre-processing needs to be done in order for the data to be useful. For example, not all recordings will be the same length, some may be louder than others, and it may even have an offset/bias. We will assume that the data we are getting is **clean**, which just means that the audio data is normalized and trimmed correctly.

Let us represent our data with matrices and vectors. We will have our main data matrix, $X \in \mathbb{R}^{n \times d}$ which contains n audio recordings which are stored as row vectors. Each audio recording has d samples taken. So, our data matrix X and labels $\vec{y}$ can be represented as:

$$X = \begin{bmatrix} \vec{x_1}^T \\ \vec{x_2}^T \\ \vec{x_3}^T \\ \vec{x_4}^T \\ \vec{x_5}^T \\ . \\ . \\ . \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ . \\ . \\ . \end{bmatrix}$$

Where $\vec{x_i} \in \mathbb{R}^d$ is the ith audio recording with label given by y_i. We assign label 1 for dogs and 0 for cats. Our classification scheme will have us chose $\vec{w}$ such that $f(X\vec{w}) = \vec{y_{pred}} \approx \vec{y}$ where f is defined in section 1 above and is applied to each element of the vector $X\vec{w}$.

This all seems very straight forward and follows much of the theory that we stated in the formal definitions. However, there is a very large problem here. The set of points with label dog and the set of points with label cat are often not linearly separable. Why is this bad? Recall from section 2.1 that our data needs to be linearly separable for the algorithm to work.

However, we have a work around that we can use. This work around is what allows perceptrons to be applied to even non linearly separable data. The work around/technique is called data featurization. This is a fancy word, but all it means is to transform the data before putting it into our data matrix. For example, if our data vector takes the form $\vec{x_i}^T = [x_1, x_2]$, then a potential featurization may be $\phi(\vec{x_i})^T = [1, x_1, x_2, x_1^2, x_2^2]$. We are using the same data points, however we are just manipulating/transforming them before stuffing them into our data vector.

Back to audio classification: What is a good featurization to use here? In 16B we learned about the DFT. If we recall correctly, the DFT is able to take discrete time domain signals and transform them into frequency domain signals. So, a smart move may be to first perform the DFT on the audio vectors before putting them into our data matrix. That is:

$$X_{DFT} = \begin{bmatrix} DFT\{\vec{x_1}^T\} \\ DFT\{\vec{x_2}^T\} \\ DFT\{\vec{x_3}^T\} \\ DFT\{\vec{x_4}^T\} \\ DFT\{\vec{x_5}^T\} \\ . \\ . \\ . \end{bmatrix} \qquad \vec{y_{DFT}} = \vec{y}$$

But why would we even think about doing this? Why would this help? It turns out that the best audio classification algorithm lives inside of the human brain. The input data comes from our ears. Human ear-brain interface actually performs a pseudo fourier transform before the brain starts decoding what these signals mean. So, we learn from the best and mimic the way that our ears work.

This technique of performing the DFT on our input data will result in linearly separable data in the case of cats and dogs. So, the big takeway here is that even though perceptrons are used for linearly separable data, we can also use them on non-linearly separable data given that we use a clever featurization.