Optimization, Gradient Descent, and Backpropagation

CSE 4308/5360: Artificial Intelligence I

University of Texas at Arlington

Optimization

- In AI (and many other scientific and engineering areas), our goal is oftentimes to construct a “good” function F for a certain task.
- For example, we may want to construct:
  - a “good” decision tree.
  - a “good” mixture of Gaussians.
  - a “good” neural network
- How do we define what “good” is?
- We have an **optimization criterion**, that quantitatively measures how good a function is.
  - When we have choices to make about how to construct the function, the optimization criterion is used to pick the best choice.

Optimization Criteria

- What examples of optimization criteria have we seen?
- For decision trees:
- For mixtures of Gaussians:

Optimization Criteria

- What examples of optimization criteria have we seen?
- For decision trees:
  - Information gain.
- For mixtures of Gaussians:
  - Log likelihood of the training data.

Optimization Criteria

- What optimization criterion can we use for neural networks?

Optimization Criteria

- What optimization criterion can we use for neural networks?
  - Training error: the sum, over all training data x_j, of absolute differences between the output $h(x_j)$ of the neural network and the actual class label y_j of x_j.

$$\boxed{E_1 = \sum_{j=1}^{N} |h(x_j) - y_j|}$$

  - Squared error: the sum of **squared** differences between the output $h(x_j)$ of the neural network and the actual class label y_j of x_j.

$$\boxed{E_2 = \sum_{j=1}^{N} \left(h(x_j) - y_j\right)^2}$$

- For reasons that will be clarified later, we like squared errors better.
  - Preview: Absolute values are not differentiable at 0.
  - We like optimization criteria that are differentiable everywhere.

Perceptron - Notation

- For this presentation, we will assume that the bias input is always equal to 1.
  - Note that in the slides the bias is set to -1, in the textbook it is 1.
- Suppose that each pattern x is D-dimensional.
  - $x = (x_1, \dots, x_D)$.
- To account for the perceptron's bias input, we will represent x as a D+1 dimensional vector:
  - $x = (1, x_1, \dots, x_D)$.
  - So, for all x, $x_0 = 1$.

Perceptron - Notation

- The perceptron has a (D+1)-dimensional vector w of weights: w = (w_0, …, $w_{D)}$.
  - w_0 is the weight for the bias input.
- We will denote as <w, x> the dot product of w and x.
- $< \ w, x > = \sum_{d=0}^{D} w_d x_d$
- Note that the textbook and slides use the variable **in** for that dot product.

Perceptron - Notation

- Suppose that we are given N training data, x_1, …, x_N, together with their associated class labels y_1, …, y_N.
- Each class label y_j is either 0 or 1.

Optimization Criterion - Perceptron

- The perceptron output on any x is denoted as h(x).
- The training squared error E of the perceptron is:

$$\boxed{E = \sum_{j=1}^{N} \left(h(x_j) - y_j\right)^2}$$

- Note that h(x) depends on the weight vector w.
- "Learning" or "training" the perceptron essentially means finding good weights for w.
- The output h(x) and the error E both depend on w.
- To show this dependency, we re-write the error equation as:

$$\boxed{E(w) = \sum_{j=1}^{N} \left(h_w(x_j) - y_j\right)^2}$$

Optimization Criterion - Perceptron

$$\boxed{E(w) = \sum_{j=1}^{N} \left(h_w(x_j) - y_j\right)^2}$$

- So, the error E is a function of w.
- We want to find a w that minimizes the error.
- This is a classic optimization problem:
  - We have a function E(w), taking as input a D-dimensional vector w, and outputting a real number.
  - We want to find the w that minimizes (or maximizes, in some problems) E(w).
  - We will talk about how to minimize E(w), but the process for maximizing E(w) is similar.

Globally and Locally Optimal Solutions

$$\boxed{E(w) = \sum_{j=1}^{N} \left(h_w(x_j) - y_j\right)^2}$$

- We want to find the w that minimizes E(w).
- In general, there are two types of solutions we can look for:
  - Finding the **globally optimal** w, such that E(w) <= E(w') for any w' != w.
  - Finding a **locally optimal** w, such that E(w) <= E(w') for all w' within some distance ε of w.
- Usually, finding the globally optimal w is infeasible:
  - Takes time exponential to D: the number of dimensions of w.
  - Essentially we need to try a lot of values for w.
- There are exceptions: specific problems where we can find globally optimal solutions.
- For most problems, we just live with locally optimal solutions.

Gradient Descent

$$\boxed{E(w) = \sum_{j=1}^{N} \left(h_w(x_j) - y_j\right)^2}$$

- We want to find the w that minimizes E(w).
- How can we find a locally optimal solution here?
- There is a standard recipe, applicable in lots of optimization problems, that is called **<u>gradient descent</u>**.
- To apply gradient descent, we just need E(w) to be differentiable, so that we can compute its **gradient vector**.

Gradient Descent

- Gradient descent is performed as follows:
  1. Let w be some initial value (chosen randonly or manually).
  2. Compute the gradient $\frac{\partial E}{\partial w}$.
  3. If $\frac{\partial E}{\partial w} < t$, where t is some predefined thershold, **exit**.
  4. Update w: $w = w + s\frac{\partial E}{\partial w}$.
  5. Go to step 2.
- Note parameter s at step 4, called the **<u>learning rate</u>**:
  - It must be chosen carefully.
  - If s is too large, we may overshoot and miss the minimum.
  - If s is too small, it may take too many iterations to stop.

Learning a Perceptron

- Suppose that a perceptron is using the step function as its activation function.
- Can we apply gradient descent in that case?
- No, because E(w) is not differentiable.
  - Small changes of w usually lead to no changes in $h_w(x)$, until we make a change large enough to cause $h_w(x)$ to switch from 0 to 1 (or from 1 to 0).
- This is why we use the sigmoid activation function g:
  - $h_w(x) = g(< w, x >) = \frac{1}{1+ e^{-<w,x>}}$
  - Given an input x, we compute the weighted sum <w, x>, and feed that to the sigmoid g.
  - The output of g is the output of the perceptron.

Learning a Perceptron

- $h_w(x) = g(< w, x >) = \frac{1}{1+ e^{-<w,x>}}$
- Then, measured just on the single training object x, the error E(w) is defined as:

$$E(w) = \left(y - h_w(x)\right)^2$$

$$= \left(y - \frac{1}{1 + e^{-<w,x>}}\right)^2$$

- In this form, E(w) is differentiable, and we can compute the gradient $\frac{\partial E}{\partial w}$.

Learning a Perceptron

- Measured just on x, the error E(w) is defined as:

$$E(w) = \left(y - h_w(x)\right)^2$$

- Computing the gradient $\frac{\partial E}{\partial w}$ is a bit of a pain, so we will skip it.
  - The textbook has all the details.
  - The details involve applications of relatively simple and well known rules of computing derivatives, such as the chain rule.

Learning a Perceptron

- Measured just on x, the error E(w) is defined as:

$$E(w) = \left(y - h_w(x)\right)^2$$

- We will skip to the solution:

$$\frac{\partial E}{\partial w} = \left(y - h_w(x)\right) * h_w(x) * (1 - h_w(x)) * x$$

- Note that $\frac{\partial E}{\partial w}$ is a (D+1) dimensional vector. It is a scalar (shown in red) multiplied by vector x.

Weight Update

$$\frac{\partial E}{\partial w} = \left(y - \ h_w(x)\right) * h_w(x) * (1 - h_w(x)) * x$$

- So, to apply the gradient descent update rule, we update the weight vector w as follows:

$$w = w + s * \left(y - \ h_w(x)\right) * h_w(x) * (1 - h_w(x)) * x$$

- Remember that s is the learning rate, it is a positive real number that should be chosen carefully, so as not to be too big or too small.
- In terms of individual weights w_d, the update rule is:

$$w_d = w_d + s * \left(y - \ h_w(x)\right) * h_w(x) * (1 - h_w(x)) * x_d$$

Perceptron Learning Algorithm

- Inputs:
  – N D-dimensional training objects x_1, …, x_N.
  – The associated class labels y_1, …, y_N, which are 0 or 1.

1. Extend each x_j to a (D+1) dimensional vector, by adding the bias input as the value for the zero-th dimension.
2. Initialize weights w_d to small random numbers.
   – For example, set each w_d between -1 and 1.
3. For j = 1 to N:
   1. Compute $h_w(x_j)$.
   2. For d = 0 to D:

$$w_d = w_d + s * \left(y - h_w(x_j)\right) * h_w(x_j) * (1 - h_w(x_j)) * x_{j,d}$$

4. If some stopping criterion has been met, **exit**.
5. Else, go to step 3.

Updates for Each Example

- One interesting thing in the perceptron learning algorithm is that weights are updated every time we see a training example.
- This is different from learning decision trees, Gaussians, or mixtures of Gaussians, where we have to look at all examples before we make an update.

Stopping Criterion

- At step 4 of the perceptron learning algorithm, we need to decide whether to stop or not.
- One thing we can do is:
  - Compute the cumulative squared error E(w) of the perceptron at that point:

$$\boxed{E(w) = \sum_{j=1}^{N} \left(h_w(x_j) - y_j\right)^2}$$

  - Compare E(w) with the cumulative error we have computed at the previous iteration.
  - If the difference is too small (e.g., smaller than 0.00001) we stop.

Using Perceptrons for Multiclass Problems

- A perceptron outputs a number between 0 and 1.
- This is sufficient only for binary classification problems.
- For more than two classes, there are many different options.
- We will follow a general approach called **one-versus-all classification**.

One-Versus-All Perceptrons

- Suppose we have M classes C_1, …, C_M, where L > 2.
- For each class C_m, train a perceptron h_m by using:
  - $y_j = 0$ if the class of x_j is not C_m.
  - $y_j = 1$ if the class of x_j is C_m.
- So, perceptron h_m is trained to recognize if an object is of class C_m or not.
- In total, we train M perceptrons, one for each class.

One-Versus-All Perceptrons

- To classify a test pattern x:
  - Compute the responses $h_m(x)$ for all M perceptrons.
  - Find the class $C_{m'}$ such that the response $h_{m'}(x)$ is higher than all other responses.
  - Output that the class of x is $C_{m'}$.
- So, we assign x to the class whose perceptron gave the highest response for x.

Neural Network Structure

- Perceptrons are organized into layers:
- There is the input layer.
  - Here, there are no actual perceptrons, just D+1 inputs, which are set to the values of each example x that is fed to the network.
  - Each input is connected to some perceptrons in the first hidden layer.
- There are one or more hidden layers.
  - Each perceptron here receives as inputs the outputs of allperceptrons from the previous layer.
  - Each perceptron provides its output as input to all perceptrons in the next layer.
- There is an output layer.
  - Each perceptron here receives as inputs the outputs of all perceptrons from the previous layer.
  - If we have a binary classification problem, we have one output perceptron.
  - Otherwise, we have as many output perceptrons as the number of classes.

Neural Network Notation

- Our training and test data is again D-dimensional.
- We extend our data to be (D+1) dimensional, so as to include the bias input.
- We have U perceptrons.
- For each perceptron P_u, we denote by a_u its output.
- We denote by $w_{u,v}$ the weight of the edge connecting the output of perceptron P_u with an input of perceptron P_v.
- Each class label y_j is now a vector.
- To make notation more convenient, we will treat y_j as a U-dimensional vector. (U is the total number of perceptrons).
- If P_u is the m-th output vector, and x_j belongs to class m, then:
  - y_j will have value 1 in the u-th dimension.
  - y_i will have values 0 in all other dimensions.

Squared Error for Neural Networks

- Let $h_w(x)$ be the output of a neural network. The output now is a vector, since there can be many output perceptrons.
- The optimization criterion is the squared error, but it must be adjusted to account for vector output:

$$\boxed{E(w) = \sum_{j=1}^{N} \sum\nolimits_{u:P_u \in \text{output layer}} \left(y_{j,u} - a_u\right)^2}$$

- This is now a double summation.
  - We sum over all training examples x_j.
  - For each x_j, we sum over all perceptrons in the output layer.
  - We sum the squared difference between the actual output, and what it should be.
  - We denote by $y_{j,u}$ the u-th dimension of class label y_j.

Error on a Single Training Example

- As we did for single perceptrons, we can measure the error of the neural network on a single training example x, and its associated class label y.
  - Note that now we denote by y_u the u-th dimension of y.

$$\boxed{E(w) = \sum_{u:P_u \in \text{output layer}} \left(y_{j,u} - a_u\right)^2}$$

- Assuming that each unit in the network uses the sigmoid activation function, E(w) is differentiable again.
- We can compute the gradient $\frac{\partial E}{\partial w}$.
- Based on the gradient, we can update all weights.

Backpropagation

- We will skip the actual calculation of $\frac{\partial E}{\partial w}$.
  - The textbook has all the details.
- We will just show the formulas for how to update weights after we see a training example x.
- There is a different formula for the weights of edges leading to the output layer, and a different formula for the rest of the edges.
- The algorithm that uses these formulas for training neural networks is called **<u>backpropagation</u>**.
- The next slides describe this algorithm.

Step 1: Compute Outputs

- Given a training example x, and its class label y, we first must compute the outputs of all units in the network.
- We follow this process:

```
// Update the input layer, set inputs equal to x.
1.  For u = 0 to D:
    – $a_u = x_u$(where $x_u$ is the u-th dimension of x).
// Update the rest of the layers:
2.  For l = 2 to L (where L is the number of layers):
    – For each perceptron $P_v$ in layer l:
        • $\text{in}_v = \sum_{u:\, P_u \in \text{ layer } l-1} w_{u,v} a_u$
        • $a_v = g(\text{in}_v)$, where g is the sigmoid activation function.
```

Step 2: Update Weights

- For each perceptron P_v in the output layer:
  - $\Delta[v] = g(\text{in}_v) * (1 - g(\text{in}_v)) * (y_v - a_v)$
  - For each perceptron P_u in the preceding layer L-1:
    - $w_{u,v} = w_{u,v} + s * a_u * \Delta[\text{v}]$

- For l = L-1 to 2:
  - For each perceptron P_v in layer l:
    - $\Delta[v] = g(\text{in}_v) * (1 - g(\text{in}_v)) * \sum_{z:\, P_z \in \text{ layer } l+1} (w_{v,z} * \Delta[\text{z}])$
    - For each perceptron P_u in the preceding layer l-1:
      - $w_{u,v} = w_{u,v} + s * a_u * \Delta[\text{v}]$

Backpropation Summary

- Inputs:
  - N D-dimensional training objects $x_1, \ldots, x_N$.
  - The associated class labels $y_1, \ldots, y_N$, which are U-dimensional vectors.

1. Extend each x_j to a (D+1) dimensional vector, by adding the bias input as the value for the zero-th dimension.
2. Initialize weights $w_{u,v}$ to small random numbers.
   - For example, set each $w_{u,v}$ between -1 and 1.
3. last_error = E(w)
4. For j = 1 to N:
   - Update weights $w_{u,v}$ as described in the previous slides.
4. err = E(w)
5. If |err – last_error| < threshold, **exit**. // threshold can be 0.00001.
6. Else: last_error = err, go to step 3.

Classification with Neural Networks

- Suppose we have M classes C_1, …, C_M.
- Each class C_m corresponds to an output perceptron P_u.
- Given a test pattern x = (x_0, …, x_D) to classify:
- Compute outputs for all units, as we did in training.

1. For u = 0 to D: $a_u = x_u$
2. For l = 2 to L (where L is the number of layers):
   - For each perceptron P_v in layer l:
     - $\text{in}_v = \sum_{u:\, P_u \in \text{ layer } l-1} w_{u,v} a_u$
     - $a_v = g(\text{in}_v)$, where g is the sigmoid activation function.

- Find the output unit P_u with the highest response a_u.
- Return the class that corresponds to P_u.

Structure of Neural Networks

- Backpropagation describes how to learn weights.
- However, it does not describe how to learn the structure:
  - How many layers?
  - How many units at each layer?
- These are parameters that we have to choose somehow.
- A good way to choose such parameters is by using a validation set, containing examples and their class labels.
  - The validation set should be separate (disjoint) from the training set.

Structure of Neural Networks

- To choose the best structure for a neural network using a validation set, we try many different parameters (number of layers, number of units per layer).
- For each choice of parameters:
  – We train several neural networks using backpropagation.
  – We measure how well each neural network classifies the validation examples.
  – Why not train just one neural network?

Structure of Neural Networks

- To choose the best structure for a neural network using a validation set, we try many different parameters (number of layers, number of units per layer).
- For each choice of parameters:
  - We train several neural networks using backpropagation.
  - We measure how well each neural network classifies the validation examples.
  - Why not train just one neural network?
  - Each network is randomly initialized, so after backpropagation it can be different from the other networks.
- At the end, we select the neural network that did best on the validation set.