Machine Learning: Feed-Forward Neural Nets

Overview

Perceptrons. A perceptron is a linear classifier of the form \( y = \text{sign}(\sigma_{i=1}^{d} w_i x_i + b) \) where the weights \( w = (w_1, \ldots, w_d) \) are trained using stochastic gradient descent. A perceptron is guaranteed to converge to some hyperplane separating two classes if the two classes are linearly separable (i.e., if there exists at least one hyperplane such that all points from Class 1 are on one side of it and all points from Class 2 are on the other side).

Support Vector Machines. A Support Vector Machine is a perceptron enhanced with the ability to classify non-linearly separable datasets.

General Support Vector Machines enhance perceptrons in two ways:

- Support Vector Machines optimize the hinge-loss (or quadratic hinge-loss) of the dataset w.r.t. a given separating plane. This allows for classifying non-linearly separable data sets.

- Support Vector Machines can use something called the kernel trick. Specifically, SVMs can generalize the \( \sum_{i=1}^{d} w_i x_i = \bar{w} \cdot \bar{x} \) inner product of the vector of weights and the data to be any, possibly non-linear, function that exhibits properties of the inner product. Because of the kernel trick, SVMs that use non-linear kernels instead of dot-products can build non-linear separating curves.

Gradient Descent. Most of the classification and prediction problems we have seen so far turned out to be multivariate optimization problems. Gradient Descent is a well-known iterative (approximation) method for searching for (local) optima of multivariate functions.
Given a function \( f(x_1, \ldots, x_d) \) that needs to be minimized, gradient descent starts with a point \( \bar{x}_0 = (x_{01}, \ldots, x_{0d}) \), and a learning rate \( \nu > 0 \), and proceeds as follows:

\[
x_{i+1} = \bar{x}_i - \nu \nabla f(\bar{x}_i),
\]
or

\[
x_{x+1, j} = x_{i, j} - \nu \frac{\partial f}{\partial x_j}(\bar{x}_i)
\]
for each \( j = 1, \ldots, d \).

**Stochastic Gradient Descent.** The stochastic gradient descent is a variation of the gradient descent method, where the gradient is computed after selecting a small subsample of data points from the dataset, rather than from going through the entire dataset in a single pass. Stochastic Gradient Descent is not as fast at convergence, but it is faster in processing as it samples a small number of points for each step of the gradient descent process.

**Non-linear transition functions.** The key idea behind the perceptron is that the class of a data point \( \bar{x} \) can be computed as a linear combination \( w \cdot \bar{x} \), against which a learned threshold \( b \) can be applied: if \( w \cdot \bar{x} > b \), \( \bar{x} \) belongs to one class, otherwise - to the other.

The disadvantage of a linear transition function that is thresholded at some value \( b \) is two-fold:

- The function \( f(x) = 1 \) if \( x > b \) and \( f(x) = -1 \) if \( x < b \) is not differentiable at \( x = b \), which means that one cannot use gradient descent methods to approximate it.

- The linear function \( f(\bar{x}) = w \cdot \bar{x} \) by itself is slow-growing.

We would like to replace the threshold function with a differentiable function which transitions from -1 to +1 value very fast.

We have seen two such functions already, when discussing logistic regression:

\[
\sigma(x) = \frac{1}{1 + e^x}
\]
and

\[
tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}
\]

**Neuron.** A neuron is essentially a perceptron with a non-linear transition function. Where a perceptron supplied with a vector \( w = (w_1, \ldots, w_d) \) of weights and a threshold \( \theta \) produces the following computations:

\[
y = f(\bar{x}) = \sum_{i=1}^{d} w_i x_i = w \cdot \bar{x}
\]

followed by
\[
\text{output}(\bar{x}) = \begin{cases} 
+1 & \text{if } y > \theta; \\
-1 & \text{if } y < \theta;
\end{cases}
\]

The neuron, replaces the second step with the computation

\[
\text{output}(\bar{x}) = h(y),
\]

where \( h \) is a predefined transition function. For example, a neuron using the sigmoid function \( \sigma(x) \) has the following output:

\[
\text{output}(\bar{x}) = \frac{1}{1 + e^{-y}} + \frac{1}{1 + e^{w \cdot \bar{x} - \theta}}
\]

**Feed-Forward Neural Networks**

The idea behind Support Vector Machines is to make a single perceptron into a more complex decision procedure.

The idea behind Feed-forward Neural Networks is to use many simple neurons together and to layer them.

**Feed-Forward Neural Network.** A feed-forward neural network consists of a set of input variables \( X_1, \ldots, X_d \), a set of hidden layers \( L_1, \ldots, L_M \), where each layer \( l \) consists of a set \( \{N_{l1}, \ldots, N_{lm_l}\} \) of nodes or neurons, and an output layer \( Y_1, \ldots, Y_K \) of neurons. The following applies:

- Each input variable \( X_i \) is connected to all nodes from the first hidden layer: i.e., the neural net contains the edges of the form \( (X_i, N_{1j}) \) for all \( i = 1, \ldots, d \) and \( j = 1, \ldots m_1 \).
- Each for \( j = 1, \ldots M - 1 \) each neuron in hidden layer \( L_j \) is connected to each neuron in hidden layer \( L_{j+1} \): i.e., the neural net contains the edges of the form \( (N_{ji}, N_{j+1,k}) \) for \( i = 1, \ldots, m_j \) and \( j = 1, \ldots, m_{j+1} \).
- Each neuron in the hidden layer \( L_M \) is connected to each output \( Y_k \): i.e., the neural net contains the edges of the form \( (X_{Mi}, Y_k) \) for \( i = 1, \ldots, m_M \) and \( k = 1, \ldots, K \).
- With each neuron \( N_{ji} \) and output neuron \( Y_j \) we associate a linear transformation function:
  \[
  a_{ji} = w_{ji} \cdot \bar{x} + b_{ji},
  \]
  where
  \[
  w_{ji} = (w_{ji,1}, \ldots, w_{ji,k_j})
  \]
  and \( b_{ji} \) are the weights and the bias associated with the neuron \( N_{ji} \) (the \( i \)th neuron in hidden layer \( j \)), and \( k_j \) is defined as follows:

  \[
  k_j = \begin{cases} 
  d & \text{if } j = 1 \\
  m_{j-1} & \text{if } j > 1
  \end{cases}
  \]

Here, we implicitly consider the output neurons to form the \( M + 1 \)st layer of the network.
• With each neuron \( N_{j,i} \) and output neuron \( Y_j \) we associate an activation function \( h_{j,i} \):
\[
z_{j,i} = h_{j,i}(a_{ji}) = h_{ji}(w_{ji} \cdot \bar{x} + b_{ji}).
\]

Here, \( h_{ji} \) are non-linear sigmoid functions (e.g., \( \sigma(x) \) or \( \tanh(x) \)).

Usually, all neurons use the same activation function, although in some cases, the activation function for output layer may be different than those of the hidden layers.

**Notation.** We use \( h_j \) to denote the vector \((h_{j,1}, \ldots, h_{j,k_j})\) and \( a_j \) to denote the vector \((a_{j1}, \ldots, a_{jk_j})\).

Based on this description, a feed-forward neural network represents a non-linear transformation of \( d \) inputs \( X_1, \ldots, X_d \) into \( K \) outputs \( Y_1, \ldots, Y_K \) produced as follows:
\[
\bar{y} = h_{M+1}(a_{M+1}),
\]

or
\[
y_k = h_{M+1,k}(a_{M+1,k}) = h_{M+1,k} \left( \sum_{i=1}^{k_M} w_{Mi} \cdot h(a_M) + b_{Mi} \right).
\]

**Example. Two-layer feed-forward network.** Consider a standard example of a neural network with an input layer \( X_1, \ldots, X_d \), a single hidden layer \( N_1, \ldots, N_M \) and an output layer \( Y_1, \ldots, Y_K \) and \( \sigma(x) = \frac{1}{1+e^{-x}} \) as the activation function for each neuron. Let us denote as \( w_1, \ldots, w_M \) and \( b_1, \ldots, b_M \) the weights and the biases for neurons in the hidden layer, and as \( v_1, \ldots, v_K \) and \( c_1, \ldots, c_K \) the weights and the biases of the neurons in the output layer.

Then, each output \( y_k \) can be computed as follows:
\[
y_k = \sigma \left( \sum_{i=1}^{M} v_{ki} \cdot \sigma \left( \sum_{j=1}^{d} w_{ij} x_j + b_i \right) + c_k \right)
\]

**Training Feed-Forward Network**

To train a neural network we need a dataset \( X = \{\bar{x}_1, \ldots, \bar{x}_n\} \), and a set \( T = \{t_1, \ldots, t_n\} \) where \( t_i = \text{class}(\bar{x}_i) \). In a more general case, we consider the output \( T \) to be a set of vectors \( T = \{\bar{t}_1, \ldots, \bar{t}_n\} \), where \( t_{ij} \) is the output of the \( j \)th classifier (represented by the neural network output element \( Y_j \)) on input \( \bar{x}_i \).

Given a neural network \( Q \) with \( k \) outputs \( Y = \{Y_1, \ldots, Y_k\} \), let vector \( \bar{y}_i = (y_{i1}, \ldots, y_{ik}) \) denote the outputs produced by \( Q \) on input vector \( \bar{x}_i \).

We want to compare the vectors \( \bar{t}_i \) and \( \bar{y}_i \), and we want these two vectors to be as close to each other as possible for all \( i = 1, \ldots, n \).

Our standard metric for this is the SSE: sum squared errors. For a given output \( Y_j \):
\[ E_j(\bar{x}_i) = (t_{ij} - y_{ij})^2 \]

The error is additive, so
\[ E(\bar{x}_i) = \sum_{j=1}^{k} E_j(\bar{x}_i) = \sum_{i=1}^{k} (t_{ij} - y_{ij})^2 \]

Finally, the full error of the dataset is
\[ E_X = \sum_{i=1}^{n} E(\bar{x}_i) = \sum_{i=1}^{n} \sum_{j=1}^{k} E_j(\bar{x}_i) = \sum_{i=1}^{n} \sum_{j=1}^{k} (t_{ij} - y_{ij})^2 \]

Let \( W \) represent the vector of all weights for all neurons in the network \( Q \). As the network is trained, the above error computation is parameterized by \( W \):
\[ E_X(W) = \sum_{i=1}^{n} E(\bar{x}) = \sum_{i=1}^{n} \sum_{j=1}^{k} E_j(\bar{x}_i) = \sum_{i=1}^{n} \sum_{j=1}^{k} (t_{ij} - y_{ij})^2 \]

Because our error is additive, we can apply both the gradient descent and stochastic gradient descent to approximate the optimal (or a sufficiently good) value.

How do we apply this to the shape of our function(s) computing values \( y_{ik} \).

**Backpropagation Algorithm**

The basic outline of our training process is as follows.

1. **Initialization.** Select a starting set of parameters \( w_{lj} \) for each neuron in layers \( L_1, \ldots, L_M \) and \( Y \) of the neural net.

2. **Step.** Each learning step is done in two stages.
   - **Stage 1: Forward propagation.** On step \( s \) Select a batch of input points \( X_s \subseteq X \). For each \( \bar{x} \in X_s \) compute the outputs of each layer using current vectors of weights \( w_{lj} \).
   - **Stage 2: Back propagation.** Starting with the output layer, apply gradient descent on the given batch \( X_s \) of points to change the weights of the neurons in each layer.

3. **Stoppage condition.** Stop when the error is small, or when the error stops changing significantly from round to round.

   The key observation here is that the actual gradient descent does not have to be computed on the entirety of the functions used for computing \( y_{ik} \); rather, it can be done by unwrapping the \( h(w \cdot \bar{x}) \) components representing computations on individual neurons, one-by-one, through, what we can call, local information exchange.

   Here is the mechanics of it for a single layer.
Consider a single output neuron $Y$. For a data point $\bar{x}$, let $t$ be $\text{class}(\bar{x})$ and $y$ be the output of the network’s $Y$ neuron on $\bar{x}$.

Let the prior hidden layer $L$ consist of neurons $N_1, \ldots, N_m$, and let $w = (w_0, \ldots, w_m)$ be weights associated with $Y$, with $w_0$ representing the bias of node $Y$. Let $\bar{z} = 1, z_1, \ldots, z_m$ be the outputs of neurons $N_1, \ldots, N_m$ on input $\bar{x}$ (with 1 corresponding to the ever-present bias).

The output $y$ is then produced as follows:

$$y = h \left( \sum_{j=0}^{m} w_j z_j \right),$$

where $h()$ is a differentiable non-linear sigmoid activation function.

Our error function is then

$$E(\bar{z}, w) = \frac{1}{2} (t - y)^2 = \frac{1}{2} \left( t - h \left( \sum_{j=0}^{m} w_j z_j \right) \right)^2$$

(note: we added the $\frac{1}{2}$ fraction to make differentiating easier).

Let us differentiate $E(\bar{z}, w)$ on $w$ now.

$$\nabla E = (y - t) \cdot \nabla y = (y - t) \cdot \nabla \left( h \left( \sum_{j=0}^{m} w_j z_j \right) \right) =$$

$$= (y - t) \cdot h' \left( \sum_{j=0}^{m} w_j z_j \right) \nabla \left( \sum_{j=0}^{m} w_j z_j \right)$$

To compute partial differentials, let us denote outputs of neurons prior to activation:

$$a = w \cdot \bar{z},$$

where $\bar{z}$ is a vector of inputs from the previous layer (or is $\bar{x}$ for the first hidden layer). This lets us represent the gradient as:

$$\nabla E = (t - y) h' (a) \nabla (a) = (t - y) h' (a).$$

We now need to evaluate the partial derivatives $\frac{\partial E}{\partial w_j}$. Using the chain rule, we can write:

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial a} \cdot \frac{\partial a}{\partial w_j}$$

Let us denote $\frac{\partial E}{\partial a}$ as $\delta$:

$$\delta = \frac{\partial E}{\partial a}.$$
We refer to $\delta$ as the error of the neuron.

$$\frac{\partial a}{\partial w_j} = \frac{\partial}{\partial w_j} (w_1 z_1 + w_2 z_2 + \ldots + w_j z_j + \ldots + w_m z_m) = z_j$$

In the output layer:

$$\delta = y - t$$

In other layers:

$$\delta = \frac{\partial E}{\partial a} = \frac{\partial}{\partial a} (t - y) h'(a)$$

References

