ML Recap

Klejdi Sevdari

June 2022

Perceptron

We want to find a hyperplane that separates the points into two classes: $y \in \{1, -1\}$. For each data point we want that $y_i \beta^T x_i \geq 1$.The parameter β is updated when a point is misclassified in the following way:

$$\beta_{\text{new}} \leftarrow \beta_{\text{old}} + \eta y_i x_i$$

Convergence
It can be shown that if the data is linearly separable, there exists a β_{sep} such that $y_i \beta_{\text{sep}}^T \frac{x_i}{\|x_i\|} \geq 1$. Additionally converence is proved by showing that:

$$\|\beta_{\text{new}} - \beta_{\text{sep}}\|^2 \leq \|\beta_{\text{old}} - \beta_{\text{sep}}\|^2 - 1$$

Capacity

We focus on a given finite number of P vectors $\{x^1, \ldots, x^P\}$. Capacity is defined as the number of dichotomies that the perceptron can implement on these P points. An assumption necessary for linear separability is general position, which states that no subset of less than or equal to N points can be linearly dependent. If this assumption holds, we have that:

$$C(P, N) = 2 \sum_{k=0}^{N-1} \binom{P-1}{k}$$

If P = 2N: $C(P, N) = 2^{P-1}$. If $P \to \infty$, then $C(P, N) \sim AP^N$. If $P < N$, then all dichotomies can be implemented.

Clustering

In Clustering, we are given N data-points and we want to group them into K disjoint groups. A good clustering is one that achieves high within-cluster similarity and low inter-cluster similarity.

Agglomerative Clustering

Agglomerative clustering is based on the union between the two nearest clusters. It can be visualized using a dendogram.

Distance between clusters can be defined in numerous ways:

- Single-linkage: $d(c_i, c_j) = \min d(a, b), \forall a \in c_i, b \in c_j$.
- Complete-linkage: $d(c_i, c_j) = \max d(a, b), \forall a \in c_i, b \in c_j$.
- Average-linkage: $d(c_i, c_j) = \frac{1}{|c_i||c_j|} \sum d(a, b)$.

Some disadvantages:

- They do not scale well. Time complexity $O(n^2)$.
- It is a greedy algorithm. Local optima are a problem.

K-Means

Definition 1 (Within-Point Scatter)**.**

$$W(C) = \sum_{k=1}^{K} N_k \sum_{C(i)=k} \|x_i - \bar{x_k}\|^2$$

Definition 2. (Algorithm for K-Means).

- For a given cluster assignment C, the total cluster variance is minimized with respect to $\{m_1, \ldots, m_k\}$ yielding the means of the currently assigned clusters:

$$m_k = \frac{\sum \mathbb{1}_{c_i=k} x_j}{N_k}$$

- Given a current set of means $\{m_1, \ldots, m_K\}$, we assign each observation to the closest (current) cluster mean:

$$C(i) = \arg\min_{1 \le k \le K} \|x_i - m_k\|^2$$

.

- Steps 1 and 2 are repeated until the assignments do not change.

K-Means is efficient $O(tKN)$, simple to implement and easy to debug. However it has some disadvantages:

- Applicable only when the mean is defined.
- Needs to have K specified in advance.
- Sensitive to outliers.
- Can only discover clusters with convex shapes.

KNNs

Idea: data points that are sufficiently close to one another should be of the same class. To predict a data point x, we compute the k closest training data points to x and find the most common class. k is usually selected using Cross Validation. A Voronoi diagram shows how the input space is divided into classes. We require $O(n)$ space to store a training set of size n. Predictions also take $O(n)$ time, which is costly. KNN can be adapted to output probabilities and can also be used for regression. KNN is a nonparametric model because the number of parameters grows with n. KNN does not behave well in high dimensions.

If some attributes have larger ranges, more importance is attributed to them, thus normalizing might be an important and natural step when using KNN. However, sometimes scale matters.

Neural Networks (less famously known as MLPs)

Activation Functions

- ReLU: $y = \max(0, x)$.
- Soft ReLU: $y = \log(1 + e^x)$.
- Hard Threshold: $y = 1$ if $x > 0$ and 0 otherwise.
- Logistic: $y = \frac{1}{1+\exp(-x)}$.
- Tanh: $y = \frac{\exp(x)-\exp(-x)}{\exp(x)-\exp(-x)}$.

Loss Functions

Universality

NNs are universal approximators: they can approximate any function arbitrarily well. A hidden layer (unbounded in size) is in principle enough. However, not all functions can be represented compactly. Hornik's theorem proves that for any continuous function, there exists a neural network of finite size that approximates the function. However, nothing can be said about the architecture of this NN or its weights. Current research shows that introducing more layers improves expressivity better than introducing more neurons per layer.

Surrogate Loss Function

What we ideally want is to minimize the number of classifications. That is we want to find w^* such that:

$$w^* = \min_w \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{y_i = \hat{y}_i}$$

However, this is problem is NP-hard. We want to choose a reasonable (i.e differentiable, convex) loss function l and choose w^* such that it minimizes:

$$w^* = \min_w \frac{1}{N} \sum_{i=1}^{N} l(w, x_i, y_i)$$

Minimizing the loss function should help with minimizing the classification loss. In most cases the loss function is an upper bound of the classification loss and they have a similar shape.

Things to keep in mind in the context of NNs

Weight Initialization
If all weights are initialized to 0 (or any other constant), the derivatives will remain the same for every w. The network will fail to break symmetry. Thus the weights should be initialized in a stochastic manner.
Learning Rate
Using a learning rate that is too large might cause the gradient of the activation functions to saturate. (think about the sigmoid function)

SGD

You select m training instances and estimate the gradient using those points. In choosing m there is a trade off between a lower variance and more computation. When using SGD, the learning rate is large in the beginning and then is decayed later on.

Regularization Techniques

Early Stopping can be used to terminate training if the validation loss becomes much greater than the training loss. This can be done by setting a patience parameter.
l_2 Regularization:

$$w^* = \arg\min \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{f}(w))^2 + \frac{1}{2}\lambda \|w\|^2$$

Other Regularization techniques include Dropout, Batch Normalization etc.

Multi Class Classification

Softmax:
For $k \in \{1, \ldots, K-1\}$, we have:

$$P(Y = k) = \frac{\exp(w_k^T x_k)}{1 + \sum_{k=1}^{K-1} \exp(w_k^T x_k)}$$

and for $k = K$:

$$P(Y = K) = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(w_k^T x_k)}$$

Useful Identities for Gradients

- $z = Wx,\ \partial z/\partial x = W$. Similarly, if $z = xW,\ \partial z/\partial x = W^T$.
- $z = f(x)$ (functions applied coordinate wise). $\partial z/\partial x = \operatorname{diag} f'(x)$.
- $J = f(z),\ z = Wx,\ \partial J/\partial W = \delta^T x^T$ where $\delta = \partial J/\partial z$.
- $J = \mathrm{CE}(y, \hat{y}),\ \hat{y} = \mathrm{softmax}(\theta),\ \partial J/\partial \theta = (\hat{y} - y)$.

Autoencoders

An autoencoder's primary purpose is learning an informative representation of the data. Training an autoencoder means finding the functions $g_w(\cdot)$(encoder) and $f_{w'}(\cdot)$(decoder) that satisfy

$$w, w' = \arg\min_{w,w'} \mathbb{E}\left[\Delta(x_i, f_{w'}(g_w(x_i)))\right]$$

Regularization of the autoencoder can be achieved by l_1 or l_2 regularization. Another approach is to tie the weights of the encoder to the weights of the decoder and have a symmetric structure.
Loss Functions for the Autoencoder
Mean Squared Error:

$$L_{\mathrm{MSE}} = \frac{1}{N} \sum_{i=1}^{M} \|x_i - \tilde{x_i}\|^2$$

Binary Cross-Entropy (! inputs and outputs should be normalized to be between 0 and 1):

$$L_{CE} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{d} [x_{i,j} \log \tilde{x}_{i,j} + (1 - x_{i,j}) \log(1 - \tilde{x}_{i,j})]$$

Some applications of the autoencoders include anomaly detection, dimensionality reduction, denoising etc.

Bias Variance Trade Off

We want to choose a model that captures the regularities in our training data and is able to generalize well. The bias error comes from wrong assumptions in the learning algorithm. The variance error arises when an algorithm models the noise in the data. The Bias-Variance decomposition is given below:

$$\mathbb{E}_{D,\epsilon}\left[(y - \hat{f}(x))^2\right] = \mathrm{Bias}_D[\hat{f}(x)]^2 + \mathrm{Var}_D[\hat{f}(x)]^2 + \sigma^2$$

If we model $y = f(x) + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$ and our fitted model is $\hat{f}(x)$ and use the MSE as a measure for error then:

$$\mathbb{E}\left[(y - \hat{f})^2\right] = \mathbb{E}\left[(f + \epsilon - \hat{f} + \mathbb{E}\hat{f} - \mathbb{E}\hat{f})^2\right] = (\mathrm{Bias}\ \hat{f})^2 + \mathrm{Var}\ \hat{f} + \sigma^2$$

Miscellaneous

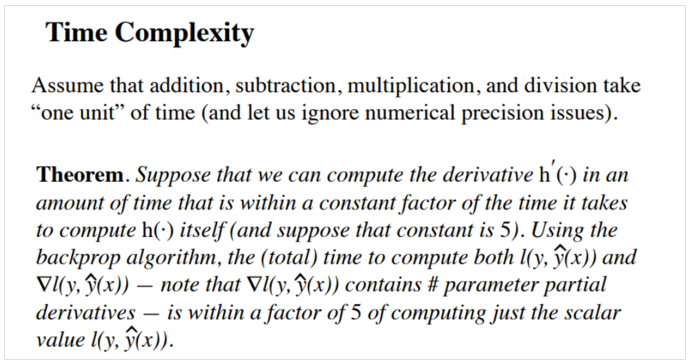

Figure 1: Time Complexity of Back Propagation