Summary

sagisk

November 2, 2021

1 Chapter 2: Machine Learning with Shallow Neural Networks

1.1 ML models:

Perceptron:
Input: $X = \{x_1, ..., x_d\}$.
Labels: $y \in \{-1, 1\}$
Weights: $W = \{w_1, ..., w_d\}$.
Prediction: $\hat{y} = \text{sign}\{WX\} = \text{sign}\{\sum_j w_j x_j\}$
Error: $E(X) = (y - \hat{y}) \in \{-2, 0, 2\}$
Loss: minimize $L = \sum_{(X,y)\in D}(y - \hat{y})^2 = \sum_{(X,y)\in D}(y - \text{sign}(WX))^2$
However since this loss is non-differentiable we will use a smother loss:
$L = \max\{0, -y(WX)\}$
Weight update:

- For non-differentiable loss: $W = W + aE(X)X = W + a(y - \hat{y})X$
- For smother loss: $W = W + ay_iX_i$ where we update only for misclassified points i.e. for $I(y_i\hat{y}_i) < 0$.

Least-Squares Regression
Input: $(X_1, y_1), ...(X_n, y_n)$ where each $X_i \in R^d, y_i \in R$.
Labels: y_i are real-valued targets.
Weights: $W = (w_1, ..., w_d) \in R^d$
Predictions: $\hat{y_i} = WX_i$
Error: $e_i = (y_i - \hat{y_i})$
Loss:

- i-th element loss: $L_i = e_i^2 = (y_i - \hat{y_i})^2$
- Total loss: $L = \sum_{L_i} = \sum_i e_i^2 = \sum_i (y_i - \hat{y_i})^2$.

Weight update:

- $W = W - a(-e_iX) = W + a(y_i - \hat{y_i})X$
- With regularization: $W = W(1 - a\lambda) - a(-e_iX) = W(1 - a\lambda) + a(y_i - \hat{y_i})X$

Widrow-Hoff Learning or Least-squares Classification: Here input, weights and predictions are the same as for the linear regression. Label: $y_i \in \{-1, +1\}$.
Loss:

- i-th element loss: $L_i = (y_i - \hat{y_i})^2 = y_i^2(y_i - \hat{y_i})^2 = (y_i^2 - \hat{y_i}y_i)^2 = (1 - \hat{y_i}y_i)^2$

Weight update:

- $W = W + ay_i(1 - \hat{y_i}y_i)X$ for numeric as well as binary responses.

- With regularization: $W = W(1 - a\lambda) + ay_i(1 - \hat{y}_i y_i)X$ only for binary responses.

Logistic Regression:
Input: $(X_1, y_1), ...(X_n, y_n)$ where each $X_i \in R^d, y_i \in \{-1, 1\}$.
Labels: $y_i \in \{-1, 1\}$.
Weights: $W = (w_1, ..., w_d) \in R^d$
Predictions: $\hat{y}_i = P(y_i = 1) = \frac{1}{1+e^{-WX_i}}$
Error: For positive samples we want to maximize $P(y_i = 1)$ and for the negative ones we want to maximize $P(y_i = -1)$. This casewise maximization is equivalent to maximizing $|y_i/2 - 0.5 + \hat{y}_i|$.
Loss: The likelihood is given as $L = \prod_{i=1}^{n} |y_i/2 - 0.5 + \hat{y}_i|$ and we want to minimize the log-likelihood $LL = \sum_{i=1}^{n} -\log(|y_i/2 - 0.5 + \hat{y}_i|)$
Weight update: $\frac{dL_i}{dW} = -\frac{\text{sign}(y_i/2-0.5+\hat{y}_i)}{|y_i/2-0.5+\hat{y}_i|}\frac{d\hat{y}_i}{dW} = -\frac{\text{sign}(y_i/2-0.5+\hat{y}_i)}{|y_i/2-0.5+\hat{y}_i|}\frac{X_i}{1+e^{-WX_i}}\frac{1}{1+e^{WX_i}} = -\frac{y_i X_i}{1+e^{y_i WX_i}}$
Hence, $W = W(1 - a\lambda) + a\frac{y_i X_i}{1+e^{y_i(WX_i)}}$
Support Vector Machines (SVM): Input, labels, weights, predictions are identical to the least-squares classification.
Loss: $L_i = \max\{0, 1-y_i\hat{y}_i\}$. The loss function is similar to the smoothed version of the loss function for the Perceptron algorithm. The main difference here is that SVM also penalizes the datapoints that are predicted correctly but are withing the margin. In comparison with Widrow-Hoff loss SVM does not penalize the too positive values while Widrow-Hoff is doing it (which is undesirable).
Weight-update: $\frac{dL_i}{dW} = -y_i X_i$ if $y_i\hat{y}_i < 1$ else is 0. So, $W = W(1 - a\lambda) + ay_i X_i$ for $y_i\hat{y}_i < 1$.

1.2 Multiclass Models:

Perceptron: Input: $(X_1, c(1)), ...(X_k, c(k))$ where $X_i \in R^d$
Labels: $c(i) \in \{1, ...k\}$ is the index of the class to which i-th data item belongs to.
Weights: $W_1, ..., W_k$ where $W_i = \{w_1^i, ..., w_d^i\} \in R^d$
Predictions: The goal is to get the largest $W_{c(i)}X_i$ so that it is larger then $W_r X_i$ for $r \neq c(i)$.
Loss: $L_i = \max_{r:r\neq c(i)} \max(W_r X_i - W_{c(i)}X_i, 0)$. For the i-th data point loss is incured by predicting it to be in the r-th class which is the most incorrectly predicted to the true $c(i)$-th class.
Weight update: $\frac{dL_i}{dW_r} = -X_i$ if $r = c(i)$, X_i if $r \neq c(i)$ else 0. So, $W_r = W_r + aX_i$ if $r = c(i)$. $W_r = W_r - aX_i$ if $r \neq c(i)$. $W_r = W_r$ otherwise.
Weston-Watkins SVM (WW SVM): Input, labels, and weights are the same as for the multiclass perceptron model.
Loss: $L_i = \sum_{r:r\neq c(i)} \max(W_r X_i - W_{c(i)}X_i + 1, 0)$. Here the difference from the multiclass perceptron is that it aggregates all losses for all classes r that lag on the true class $c(i)$ be less than a margin amount of 1. While the multiclass perceptron just uses the smallest loss (i.e. the r-th class most similar to the true $c(i)$-th class).
Weight update: WW SVM updates on any class that is predicted more favorable than the true class. $\frac{dL_i}{dW_r} = -X_i[\sum_{j\neq r} \sigma(j, X_i)]$ if $r = c(i)$ and $X_i[\sigma(r, X_i)]$ otherwise.
Hence, $W_r = W_r(1 - a\lambda) + aX_i[\sum_{j\neq r} \sigma(j, X_i)]$ if $r = c(i)$ and $W_r = W_r(1 - a\lambda) + a(-X[\sigma(r, X_i)])$ if $r \neq c(i)$
Logistic Regression: Input, labels, weights are the same as for the above two models. Predictions: $P(r|X_i) = \frac{e^{W_r X_i}}{\sum_j e^{W_j X_i}}$
Loss: The cross-entropy loss: $L_i = -\log[P(c(i)|X_i)] = -W_{c(i)}X_i + \log[\sum_j e^(W_j X_i)]$
Weight update:

1.3 Backpropagetion for Interpretability and Feature Selection:

The set up is that we have a test instance $X = (x_1, ..., x_d)$ for which the multilabel output scores of the neural network are $\{o_1, ..., o_k\}$. $Furthermore, let the output of the winning class among the k outputs be o_m$, where $m \in \{1...k\}$. Then to get how relevant a x_i-feature is we can get $\frac{do_m}{dx_i}$. The features for which this partial derivative has a large absolute value have the greatest influence on the classification to the winning class.

1.4 Autoencoders:

The basic idea of autoencoders is to have the same number of output nodes as input nodes. Autoencoder replicates data from input to output. However, it does not output the same data as in the input layer since the hidden layers of it are constraint for example by having less nodes than in the input layer. Hence, the replication is lossy. Typically (not necessary) there is a symmetry between hidden layers i.e. the k-th layer is the same as $M-k+1$-th layer for M-layer autoencoder. Very often, M-is odd to have $(M+1)/2$ layer the most constricted layer.

Autoencoders and Singular value decomposition (SVD): Input: $n \times d$ matrix D. Architecture: A single hidden layer network with dimensionality $k << d$. Weights: From input to hidden matrix W^T. From hidden to output: $k \times d$ matrix V^T Prediction: DWV^T. Loss: $\|DW^TV^T - D\|^2$ which has a closed form solution $W = (V^TV)^{-1}V^T$. Hence, $DW^T = U(V^TW^T) = U$. Difference from SVD: In SVD the matrix V is orthonormal. However, here the matrix with orthonormal columns is just one of the possible solutions. However, the subspace spanned by the columns of V will be the same as spanned by the basis vectors of SVD. If the weights of encoder and decoder are shared i.e. From input to hidden: W = V From hidden to output: V^T. Then first V of weights are used to transform d-dimesnional data points X to k-dimensional then the weights of V^T transform them back to orginial representation. So, $V^TV = I$ i.e. V-is orthogonal so we get exactly SVD.

Other decompoisitons: The same architecture can be applied to PCA just D should be mean-centered before computations.

Logistic matrix factorization: Input: D binary data matrix. Architecture: A single layer with sigmoid function applied at the output layer. Weights:

- From input to hidden matrix W^T.
- From hidden to output: $k \times d$ matrix V^T

Prediction: $B = Bern(\text{sigmoid}(UV^T))$ where B_{ij} is from Bern(sigmoid($[UV^T]_{ij}$)

Autoencoder's applications: Outlier Detection: If $D \approx D' = UV^T$ then D' is an approximation to D and $(D - D')$ represent the outlier scores of the matrix entries If one add the squared scores in each column of D then one can get outlier features.

1.5 Recommender Systems:

Input: $n \times d$ sparse rating matrix D with n users and d items. The input to recommender systems can be represented as a set of triplets:
< RowId >, < ColmnId >,< Rating > Architecture:

- Input layer: n - one hot encoded input nodes (corresponding to n users).
- Hidden layer: k-units where k is the rank of factorization.
- Output layer: d-untis, where d is the number of columns (corresponding to items).

Weights:

- Input to hidden: An $n \times k$ matrix U
- Hidden to output: An $k \times d$ matrix V^T.

Loss: The sum of the squares of the errors. The **BIG peculiarity** here is that we compute the sum only over those output nodes which have a corresponding known entries. This is a result of the sparsity of the input matrix D.

Prediction. Though, the loss is computed only over known entries the prediction can be given for all d outputs by e_rUV^T for the r-th row (r-th user).

1.6 Word2Vec:

word2vec methods are used to create word embedding. There are two variants:

1. Continuous Bag Of Words (CBOW):

Task: predict i-th word from the t context words. Input: $w_1, ..., w_m$ where $m = 2t$ where t is the number of words before and after the i-th word. Prediction: $P(w|w_1, w_2, ..., w_m)$. Architecture: Input layer: $m \times d$ nodes each word is one-hot encoded. Specifically, the input $x_{ij} \in \{0, 1\}$ where $i \in \{1, ..., m\}$ the position of the context and $j \in \{1, ..., d\}$ the identifier of the word in a lexicon of size d. Hidden layer: with p nodes. $h_q = \sum_{i=1}^{m} \sum_{j=1}^{d} u_{jq} x_{ij}$ where to get the q-th hidden node output we average the words in the context. Output layer: with d nodes. The output after application of softmax is: $\hat{y} = \frac{e^{h_q v_{qj}}}{\sum_{q=1} e^{h_q v_{qj}}}$

Weights:

- Input to hidden: The weights between the m context words. And u_{jq} is the weight for the j-th word in the context to the q-th hidden layer node. So, we have $d \times p$ matrix U.
- Hidden to output: The weight matrix is $p \times d, V$.

Loss:

- For a target word $w = r$ we have $L_r = -\log P(y_r = 1|w_1, .., w_m) = -\log(\hat{y_r})$
- The total loss: $L = -\sum_{i=1}^{m} y_i \log(\hat{y_i})$.

Weight update: Using the derivative of the softmax function one can show that $\frac{dL}{d(h_q v_{qj})} = \sum \frac{dL}{d\hat{y_i}} \frac{d\hat{y_i}}{d(h_q v_{qj})} = y_j - \hat{y_j}$

- $u_i = u_i - a \frac{dL}{du_i}$ wher u_i is the column of U.
- $v_i = v_i - a \frac{dL}{dv_i}$

1. 2. Skip-gram:

Task: Predict the context words from the i-th word. Input: the same as in CBOW. Prediction: $P(w_1, w_2, ..., w_m|w)$ Architecture: Input layer: One-hot encodied d-binary inputs $x_1, ..., x_d$ corresponding to d possible values of a single input word. Hidden layer: p-units Output layer: One-hot encoded $m \times d$ values where $y_{ij} \in \{0, 1\}$ where i ranges over $1, ..m$ (size of context window), and j ranges from $1, .., d$ (lexicon size.

Weights:

- Input to hidden: Each input x_j is connected to all the hidden nodes with a $d \times p$ matrix U.
- Hidden to output: The hidden layer with a $p \times d$ matrix $V = [v_{qj}]$. Each of these m groups of $d$output nodes computes the probabilities of the various words for a particular context word. V is shared among possible m-groups.

Loss: $L = -\sum_{i=1}^{m} \sum_{j=1}^{d} y_{ij} \log(\hat{y_{ij}})$

Weight update: $u_i = u_i - a \frac{dL}{du_i}$ wher u_i is the column of U. $v_i = v_i - a \frac{dL}{dv_i}$

References