Supervised Machine Learning

DESKTOP-H800RKQ

November 11, 2024

Contents

1 Linear Regression **1**

2 Linear Classifier **3**

3 Neural Networks **3**
3.1 Activation Functions 4
3.1.1 Step Function 4
3.1.2 Sigmoid Function 5
3.1.3 Rectified Linear Unit (ReLU) 5
3.1.4 Leaky ReLU 5
3.2 Backpropagation 5
3.3 Generalization and Optimization 6
3.4 Modeling 7
3.4.1 Sequence Modeling 7
3.4.2 Composite Models 7

1 Linear Regression

A model in which the output is a linear function of the input features:

$$\hat{Y}_{\vec{w}}(e) = \sum_{i=0}^{n} w_i X_i(e)$$

where $\vec{w} = \langle w_0, \ldots, w_n \rangle$ and $X_0 = 1$.

The **sum of squares error** on examples E for output Y is

$$\text{Error}(E, \vec{w}) = \sum_{e \in E} (Y(e) - \hat{Y}_{\vec{w}}(e))^2$$

The goal is to find weights that minimize the sum of squares error.

The minimum error can be found analytically. For output features of M examples $\vec{y}$, X being a matrix with column j holding the values of the input features for example j, and weights $\vec{w}$,

$$\vec{y} = \vec{w}X \implies \vec{y}X^T(XX^T)^{-1} = \vec{w}$$

The minimum error can also be found iteratively, which works for more problems. This uses **gradient descent**:

$$w_i \leftarrow w_i - \eta \frac{\partial \text{Error}(E, \vec{w})}{\partial w_i}$$

where η is the gradient descent step size, called the **learning rate**. Using sum of squares error here gives the rule

$$w_i \leftarrow w_i + \eta \sum_{e \in E} \left(Y(e) - \sum_{i=0}^{n} w_i X_i(e) \right) X_i(e)$$

where $\eta \to 2\eta$ is an arbitrary scale.

Algorithm 1 LinearLearner

```
1:  Input: Set of input features, X = {X_1, ..., X_n}
2:  Input: Output feature, Y
3:  Input: Set of examples, E
4:  Input: Learning rate, η
5:  Initialize weights w⃗ = ⟨0, 0, ..., 0⟩ randomly
6:  repeat
7:      for each example e ∈ E do
8:          δ ← Y(e) − Σ_{i=0}^{n} w_i X_i(e)
9:          for each i ∈ {0, ..., n} do
10:             w_i ← w_i + ηδX_i(e)
11:         end for
12:     end for
13: until Stop criteria are true
```

With this algorithm, if examples are chosen at random on line 7, this becomes **stochastic gradient descent**.

With **batched gradient descent**:

1. process a batch of size n before updating weights
2. if n is all the data, this is gradient descent
3. if $n = 1$, this is incremental gradient descent

Incremental can be more efficient than batch, but it is not guaranteed to converge. This is because weight updates are done immediately but this might undo the work of other weight updates, creating an oscillation.

2 Linear Classifier

For binary classification between 0 and 1, the **squashed linear function** is of the form:

$$\hat{Y}_{\vec{w}}(e) = f\left(\sum_{i=0}^{n} w_i X_i(e)\right)$$

where f is an **activation function**.

If the activation function is differentiable, gradient descent can be used to update the weights. The partial derivative of the sum of squares error for an example e becomes

$$\frac{\partial \text{Error}(e, \vec{w})}{\partial w_i} = -2\delta f'\left(\sum_i w_i X_i(e)\right) X_i(e)$$

where δ is the square root of the sum of squares error. Thus, each example e updates each weight w_i by

$$w_i \leftarrow w_i + \eta\delta f'\left(\sum_i w_i X_i(e)\right) X_i(e)$$

The logistical activation function is

$$f(x) = \frac{1}{1+e^{-x}},\ f'(x) = f(x)(1-f(x))$$

3 Neural Networks

Inspired by networks in the brain. Idea is to connect simple units that have a threshold and fire.

Neural networks can learn the same things as a decision tree but imposes different learning bias. Linear and sigmoid layers are treated as a single layer.

Backpropagation learning: errors made are propagated backwards to change the weights

Each node j:

- has a set of weights $w_j = [w_{j0}, \ldots, w_{jN}]$
- receives inputs $v = [v_0, \ldots, v_N]$ which are either example features $[x_0, \ldots, x_N]$ or the output of a previous layer ℓ $[z_0^{(\ell)}, \ldots, z_N^{(\ell)}]$.

The number of weights is one more than the number of parents. The output is the activation function output

$$z_j = f\left(\sum_i w_{ij} v_i\right)$$

which is necessarily non-linear.

When connecting neurons into a network:

- **feedforward network**
  - forms a directed acyclic graph
  - have connections only in one direction
  - represents a function of its inputs
- **recurrent network**
  - feeds outputs back into inputs
  - supports short-term memory, so for given inputs, the behaviour of the network depends on the initial state, which may depend on previous inputs

3.1 Activation Functions

3.1.1 Step Function

$f(x) = 1$ if $x > 0$ else $f(x) = 0$. Simple to use but not differentiable so not used in practice.

3.1.2 Sigmoid Function

$$f(x) = \frac{1}{1 + e^{-kx}}$$

For very large or small x, $f(x)$ approaches 1 or 0. Tuning k approximates the step function, with higher values of k leading to a steeper sigmoid. Sigmoid is differentiable but computationally expensive.

Vanishing gradient problem: when x is very large/small, $f(x)$ responds very little to changes in x, so the network does not learn further or learns very slowly.

3.1.3 Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

Computationally efficient and network converges quickly. ReLU is differentiable.

Dying ReLU problem: when inputs approach 0 or are negative, the gradient becomes 0 and the network cannot learn.

3.1.4 Leaky ReLU

$$f(x) = \max(0, x) + k \cdot \min(0, x)$$

Small positive slope k in the negative area enables learning for negative input values.

3.2 Backpropagation

Backpropagation implements stochastic gradient descent.

The **backpropagation algorithm** is an efficient method of calculating the gradients in a multi-layer neural network. Given training examples $(\vec{x}_n, \vec{y}_n)$ and an error/loss function $\text{Error}(\hat{Y}, Y)$, perform 2 passes:

- **forward pass**: compute the error given the inputs and weights
- **backward pass**: compute the gradients $\frac{\partial \text{Error}}{\partial W^{(2)}_{j,k}}$ and $\frac{\partial \text{Error}}{\partial W^{(1)}_{i,j}}$

Update each weight by the sum of the partial derivatives for all the training examples.

3.3 Generalization and Optimization

For unit j of layer ℓ

$$\delta_j^{(\ell)} = \begin{cases} \frac{\partial \text{Error}}{\partial z_j^{(\ell)}} \times f'(a_j^{(\ell)}) & \text{base case, } j \text{ is an output unit} \\ \left(\sum_k \delta_k^{(\ell+1)} W_{j,k}^{(\ell+1)}\right) \times f'(a_j^{(\ell)}) & \text{recursive case, } j \text{ is a hidden unit} \end{cases}$$

Using this, update weights using

$$W_{j,k}^{(\ell)} \leftarrow W_{j,k}^{(\ell)} - \eta \frac{\partial \text{Error}}{\partial W_{j,k}^{(\ell)}}$$

where η is the learning rate

$$\frac{\partial \text{Error}}{\partial W_{j,k}^{(\ell)}} = \delta_k^{(\ell)} z_j^{(\ell-1)}$$

for all layers beyond the input layer, or

$$\frac{\partial \text{Error}}{\partial W_{j,k}^{(\ell)}} = \delta_k^{(1)} x_j$$

at the input layer.

To improve optimization:

- **momentum**: weight changes accumulate over iterations
- **RMS-prop**: rolling average of square of gradient
- **Adam**: combination of momentum and RMS-prop
- **initialization**: randomly set params to start

Regularized Neural nets prevent overfitting, increased bias for reduced variance through:

- param norm penalties added to the objective function
- dataset augmentation
- early stopping
- dropout
- param tying

- **convolutional** neural nets: used for images, params tied across space
- **recurrent** neural nets: used for sequences, params tied across time

3.4 Modeling

3.4.1 Sequence Modeling

Word Embedding: latent vector spaces that represent the meaning of words in context

RNNs: neural net repeats over time and has inputs from previous time step

LSTM: RNN with longer-term memory

Attention: uses expected embeddings to focus updates on relevant parts of the network

Transformers: multiple attention mechanisms

LLMs: very large transformers for language

3.4.2 Composite Models

In **random forests**, each decision tree in the forest is different, with different features, splitting criteria, and training sets. The average or majority vote determines the output.

In **support vector machines**, find the classification boundary with the widest margin. Combine this with the kernel trick.

Ensemble learning is a combination of base-level algorithms.

In **boosting**, a sequence of learners fit the examples the previous learner did not fit well. This way, learners are progressively biased towards higher precision. Early learners have many false positives but reject all clear negatives. Late learners have a more difficult problem, but the set of examples is more focused.