Notes on Deep Learning

Qiang Hu

January 10, 2018

Abstract

In this note, the concepts and knowledge on Deep Learning are recorded.

1 Introduction to Deep Learning

- **Deep learning** focuses on using **neural networks** for complex practical problems.
- Deep neural networks are used for *object recognition* and *image analysis*, for various modules of self-driving cars, for chatbots and natural language understanding problems.
- Linear models and stochastic optimization methods are crucial for training deep neural networks.
- Popular building blocks of neural networks includes **fully connected layers**, *convolutional* and *recurrent layers*. These blocks are used to define complex modern architectures in **TensorFlow** and **Keras** frameworks.
- Prerequisite knowledge:
  - **Linear Regression:** a supervised machine learning algorithm where the predicted output is continuous and has a constant slope.
  - **Mean Squared Error** (MSE): as a cost function, it is used to optimize the weights. MSE measures the average squared difference between an observation's actual and predicted values. The output is a single number representing the cost, or score, associated with our current set of weights.

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - y_i^{'})^2 \tag{1}$$

y_i is the actual value of an observation, and $y_i^{'}$ is the prediction.

– **Gradient descent**: a method to minimize MSE. Calculates the gradient of the cost function. To capture the impact of each weight on the final prediction, we use **partial derivatives**. To find the partial derivatives, we use the **Chain rule**. E.g.,

$$f`(m, b) = \begin{bmatrix} \frac{df}{dm} \\ \frac{df}{db} \end{bmatrix} \tag{2}$$

To minimize the cost function, we iteratively move in the direction of steepest descent as defined by the negative of the gradient.

– **Learning rate**: it is size of the update on the weights. The calculated gradient tells us the slope of our cost function at our current position (i.e., weights) and the direction we should update to reduce our cost function. We should move in the direction opposite the gradient.

– **Training**: training a model is the process of iteratively improving your prediction equation by looping through the dataset multiple times, each time updating the weights values in the direction indicated by the slope of the cost function. Training is completed when we reach an acceptable error threshold, or when subsequent training iterations fail to reduce our cost. Before training, we need to initializing our weights (set default values), set our **hyperparameters** (i.e., learning rate and number of iterations).

The optimization problem can be formed as,

$$\min_w L(w) = \min_w \sum_{i=1}^{l} L(w; x_i, y_i) \tag{3}$$

L(w) is the loss (cost) function.
w^0 is the initial weights.
while True,

$$w^t = w^{t-1} - \eta_t \nabla L(w^{t-1}) \tag{4}$$

if $||w^t - w^{t-1}|| < \epsilon$, break

Note that, l gradients should be computed on each step. If the dataset doesn`t fit in memory, it should be read from the disk on every GD step.

– **Normalization**: As the number of features grows, calculating gradient takes longer to compute. We can speed this up by "normalizing" our input data to ensure all values are within the same range. This is especially important for datasets with high standard deviations or differences in the ranges of the attributes. Our goal now will be to normalize our features so they are all in the range -1 to 1. First, subtract the mean of the column (mean normalization); second, divide by the range of the column (feature scaling).

– **Logistic regression**: a classification algorithm used to assign observations to a discrete set of classes. Logistic regression transforms its output using the logistic sigmoid function to return a *probability* value which can then be mapped to two or more discrete classes.

– **Sigmoid activation**: it is used to map predicted values to probabilities. The function maps any real value into another value between 0 and 1.

$$S(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

It's easy to calculate the derivative of the sigmoid function.

$$s`(z) = s(z)(1 - s(z)) \tag{6}$$

– **Decision boundary**: the threshold value or tipping point used to map the probability value to a discrete class.

– **Prediction function**: in logistic regression, a prediction function returns the probability of our observation being positive, True, or "Yes". We call this class 1 and its notation is $P(class = 1)$. As the probability gets closer to 1, our model is more confident that the observation is in class 1.

– **Cross-Entropy**: a.k.a., **Log Loss**. It can be divided into two separate cost functions: one for $y = 1$ and one for $y = 0$.

$$\begin{array}{ll} J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) & \\ \quad Cost(h_\theta(x), y) = -\log h_\theta(x) & \text{if} y = 1 \\ Cost(h_\theta(x), y) = -\log\left(1 - h_\theta(x)\right) & \text{if} y = 0 \end{array} \tag{7}$$

The key thing to note is the cost function penalizes confident and wrong predictions more than it rewards confident and right predictions!

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta(x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right) \tag{8}$$

And the derivative of the cost function is:

$$C` = x \cdot (s(z) - y) \tag{9}$$

– **Multiclass logistic regression**: When $y = 0, 1, 2, ..., n$,

1. Divide the problem into $n + 1$ binary classification problems;
2. For each class ...
3. Predict the probability the observations are in that single class.
4. prediction= $\max$ (probability of the classes)

– **Softmax transform**: transfer the score to probability.

$$z = (w_1^T x, ..., w_K^T x) \Rightarrow (e^{z_1}, ..., e^{z_K}) \tag{10}$$

$$\sigma(z) = \left(\frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}, ..., \frac{e^{z_K}}{\sum_{k=1}^{K} e^{z_k}} \right) \tag{11}$$

– **Model regularization**: add a regularizer $R(w)$ to the original loss function $L(w)$. The regularizer penalizes the model for large weights. λ is the regularization strength which controls the model quality on a training set and model complexity.

$$L_{\text{reg}}(w) = L(w) + \lambda R(w) \Rightarrow \min_w L_{\text{reg}}(w) \tag{12}$$

To minimize the regularized loss function $L_{\text{reg}}(w)$, it equals to the optimization problem,

$$\begin{cases} \min_w L(w) \\ s.t.\ R(w) \leq C \end{cases} \tag{13}$$

If the L1 penalty $||w|| = \sum |w_i|$ is used,

* Drives some weights exactly to zero;
* Learns sparse models;
* Cannot be optimized with simple gradient methods.

If the L2 penalty $||w||^2 = \sum w_j^2$ is used,

* Drives all weights closer to zero;
* Can be optimized with gradient methods.

Other regularization techniques:

* Dimensionality reduction, e.g., remove features, principal component analysis;
* Data augmentation, e.g., upon image data, flip, rotate the image to get more data;
* Dropout;
* Early stopping;
* Collect more data.

– **Stochastic gradient descent**: Similar to the regular GD, the training starts with an initial w_0, but in every step, we randomly choose an example with index i between 1 and l to update the weight. This process leads to very noisy approximations. But if we make enough numbers of iterations, it converges to some minimum.

* Noisy updates lead to fluctuations;
* Needs only one example on each step;
* Can be used in online setting;

  - Learning rate η_t should be chosen very carefully.

- **Mini-batch gradient descent**: Similar to the stochastic GD, but in very step, we randomly choose m examples from the dataset to calculated the gradient.
  - Still can be used in online setting;
  - Reduces the variance of gradient approximations;
  - Learning rate η_t should be chosen very carefully.
- **Gradient descent extensions**: to optimize GD for difficult functions with complex level sets, we can define the **momentum** function $h_t = \alpha h_{t-1} + \eta_t g_t$, where g_t is the gradient calculated. Then we update the weight using h_t, i.e., $w^t = w^{t-1} - h_t$.
  - It tends to move in the same direction as on previous steps;
  - h_t accumulates values along dimensions where gradients have the same sign;
  - Usually, $\alpha = 0.9$.

  Nesterov momentum: $h_t = \alpha h_{t-1} + \eta_t \nabla L(w^{t-1} - \alpha h_{t-1})$
- **AdaGrad**: choose learning rate adaptively.

$$\begin{aligned} G_j^t &= G_j^{t-1} + g_{t,j}^2 \\ w_j^t &= w_j^{t-1} - \frac{\eta_t g_{t,j}}{\sqrt{G_j^t + \epsilon}} \end{aligned} \tag{14}$$

  where $g_{t,j}$ is the gradient with respect to j-th parameter.
  - Separate learning rates for each dimension;
  - Suits for sparse data
  - Learning rate can be fixed as $\eta_t = 0.01$;
  - G_j^t always increases, leads to early stops.
- **RMSprop**:

$$\begin{aligned} G_j^t &= \alpha G_j^{t-1} + (1-\alpha) g_{t,j}^2 \\ w_j^t &= w_j^{t-1} - \frac{\eta_t g_{t,j}}{\sqrt{G_j^t + \epsilon}} \end{aligned} \tag{15}$$

  - α is about 0.9;
  - Learning rate adapts to latest gradient steps.
- **Adam**:

$$\begin{aligned} m_j^t &= \frac{\beta_1 m_j^{t-1} + (1-\beta_1) g_{t,j}}{1-\beta_1^t} \\ v_j^t &= \frac{\beta_2 v_j^{t-1} + (1-\beta_2) g_{t,j}^2}{1-\beta_2^t} \\ w_j^t &= w_j^{t-1} - \frac{\eta_t m_j^t}{\sqrt{v_j^t + \epsilon}} \end{aligned} \tag{16}$$

  Note in the first step, the normalization is very large, but as the iteration goes on, the denominator is close to 1. This is used to remove the bias of v closing to 0 at the beginning. It also combines momentum and individual learning rates.