Optimization on Deep learning

MATH 818.01 Midterm Survey

Jaeheun Jung

Abstract

Rescently, the deep neural networks succeded in many machine learning tasks and began to be used in mary research areas. To make useful deep learning model we should gather the data and train the proposed model. This survey presents an brief overview of training in deep learning in various perspectives including loss function, optimizer, parallelism and training strategies. The survey also includes the post training process for model compression.

1 Introduction

The deep learning methods based on multilayer perceptrons are recently succeded in various kinds of tasks in machine learning.

For example, the convolutional neural network method had been succeeded in computer vision that treats image data and recuurent neural network method worked well to treat serial datastes. Every deep learning methods include the deep neural net architecture inside their models as an efficient function estimator.

To perform the training of deep learning architecture, we need to define optimization problem and apply the optimization with optimizers. Note that there are some configurations needed to be decides by users. More precisely we need to decide the loss function, batch size and optimizer with optimizer configurations such as learning rate. Moreover there are several algorithms for applying above parameter updates due to parallelism of dataset and model for backpropagation.

In this paper we will see how it works and the differences betwwen various options for each configurations.

2 Deep learning architecture

The DNN architecture is an universal function approximator consist of multiple artificial neurons and their connections. More precisely, there are multiple hidden layers and nodes located on each layers. Each nodes recieves input and sents the output information to nodes on next layer. The input of nodes on next layer can be represented as weighted sum of outputs of previous nodes.

This DNN architecture was deribed by imitating biological brain which is complicated complex consist of connection of huge number of neurons.

As an mathematical object, the DNN architecture can be described as composition of sequence of nonlinear activation and weighted summation. Since the weighted sum can be written as matrix multiplication we can consider the weights as the matrix formed parameters and DNN architecture as class of functions parametrized by matrices.

We train the DNN model from data by minimizing some real-valued function called loss fucntion. There are various kind of loss functions represents the measurement of similarity between model function and real data distribution.

Let $y_n^{true} = (y_{n1}^{true}, \cdots, y_{nk}^{true}) \in \mathbb{R}^k$ be real data and y_n^{pred} be output of the model function with respect to the input x. Note that y_n^{pred} is parametrized by parmaeters θ.

There are most commonly used loss functions for deep learning:

1. The mean-squared error mse is defined with

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i^{true} - y_i^{pred})^2$$

2. The cross-entropy loss is defined as

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} -y_{ij}^{true} log(y_{ij}^{pred})$$

Applying optimization to minimize the loss function we can train our model fits to the given data.

3 Gradient descent optimization algorithm

To train our deep learning model, we should minimize the loss function with respect to the parameters. Our goal is to apply optimization as efficiently as possible.

To achieve the goal, we use an first-order iterative optimization algorithm which called gradient descent method in general. Of course there are another optimization methods (e.g. optimizers based on Newton's method) but they require huge computation resources due to large amounts of parameters.

Even the second order method is hard to be implemented because hessian matrix is too large and difficult to compute.

The gradient descent algorithm allows us to find local minimum with by taking steps which proportional to negative direction of gradient because gradient direction represents fastest increasing direction of objective function.

The gradient descent algorithm can be written as

Algorithm 1: Gradient Descent

Input loss function $L(\theta)$ and learning rate μ;
initialize weight $\bar{\theta} = (\theta_1, \cdots, \theta_n) \in \mathbb{R}^n$;
initialize timestamp $t \longleftarrow 0$;
repeat
  $\theta^{new} \longleftarrow \bar{\theta}$;
  for $i = 1, \cdots, n$ **do**
    $\theta_i^{new} \longleftarrow \theta_i^{new} - \mu \frac{\partial L}{\partial \theta_i}|_{\theta=\bar{\theta}}$
  end
  $\bar{\theta} \longleftarrow \theta^{new}$
until *converge*;

This algorithm had been improved in many ways which would be introduced in later sections.

4 Advanced optimizers

Recently, there are many improvements have made by idea of adaptive learning rate and momentum to accelerate the optimization. They update the parameters by referring previous gradients.

Since training methods are based on the gradient descent, we can only find local minima. This can be problem if the model defined on poor local minima of loss function will perform bad accuracy. However, the advance optimizers described in this section allows us to avoid this situation that trapped in local minima.

Also, it is considered that advanced optimizers are faster in convergence.

4.1 SGD with momentum

The momentum method computes momentum of the gradient which represents the exponentially weighted sum of gradietns. This allows to avoid some local minima because the v^{t+1} can be nonzero at local minima θ^t. The update rule of momentum method can be written as

$$v_{t+1} = \gamma v_t - \eta \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

The Momentum method helps to accelerate SGD in the relevant direction and dampen the oscillations which can be observerd with large learning rates.

4.2 [DHS11] Adagrad

The adaptive gradient algorithm(Adagrad) computes adaptive learning rates by reffering size of previous gradients. Its update rule can be described as

$$G_\theta^t = \sum_{i=0}^{t} (\nabla L(\theta_i))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_\theta^t + \epsilon}} \nabla L(\theta_t)$$

The G_θ^t is sum of squares of all previous gradients. Thus G_θ^t is increasing with respect to timestamp t. Therefore it affects the learinng rate to decrease. However, this became problem because G_θ^t is getting too large for large t.

4.3 [HSS] RMSprop

Root mean square propagation introduces idea of moving average to prevent the learning rate from getting too small. The update rule for RMSprop is given with:

$$G_\theta^t = \gamma G_\theta^{t-1} + (1-\gamma)(\nabla L(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_\theta^t + \epsilon}} \nabla L(\theta^t)$$

The new update rule for G_θ^t sets smaller weight for past information and this gives us some upper bound of G_θ^t. Therefore, the adaptive learning rate won't be too small.

4.4 [KB14] Adam

The Adam(adaptive moment estimation) optimizer is very popular optimizer which takes advatages of momentum and RMSprop. Adam optimizer computes first and second moment estimation with bias-correction. The update rule for Adam optimizer is written as:

$$\hat{m_\theta} = \frac{m_\theta^{t+1}}{1-\beta_1^{t+1}} \text{ where } m_\theta^{t+1} = \beta_1 m_\theta^t + (1-\beta_1)\nabla L(\theta^t)$$
$$\hat{G_\theta} = \frac{G_\theta^{t+1}}{1-\beta_2^{t+1}} \text{ where } G_\theta^{t+1} = \beta_2 G_\theta^t + (1-\beta_2)(\nabla L(\theta^t))^2$$
$$\theta^{t+1} = \theta^t - \eta \frac{\hat{m_\theta}}{\sqrt{\hat{G_\theta}} + \epsilon}$$

In general, Adam optimizer converges faster than any other optimizers. We can use $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as default.

4.5 [LJH+19] RAdam

There is new state-of-the-art optimizer introduced rescently. It is called Rectified Adam(RAdam) optimizer. It rectifies the variance of adaptive learning rate. The update rule for RAdam is written as:

$$
\begin{aligned}
&v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(\theta^t))^2 \\
&\hat{m_t} = \frac{m_t}{1-\beta_1^t} \text{ where } m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(\theta_t) \\
&\rho_t = \rho_\infty - 2t\frac{\beta_2^t}{1-\beta_2^t} \text{ where } \rho_\infty = \frac{2}{1-\beta_2} - 1 \\
&\text{if} \quad \rho_t > 4 \text{ then} \\
&\qquad \ell_t = \sqrt{\frac{1-\beta_2^t}{v_t}} \\
&\qquad r_t = \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}} \\
&\qquad \theta_t = \theta_{t-1} - \alpha_t r_t \ell_t \hat{m}_t \text{ where } \alpha_t \text{ is step size} \\
&\text{else } \theta_t = \theta_{t-1} - \alpha_t \hat{m}_t
\end{aligned}
$$

The RAdam optimizer computes the variance of adaptive learing rate and decide to use adaptive update rule for learning rate or not. This makes the model to avoid the bad local minima occur in early stage of traininig.

5 Update strategy

5.1 stochastic,batch,minibatch trianinig

There are three kinds of trainign methods for trainingwhich classified according to the batch sizes.

At first, The batch gradient descent computes the average of loss functions at all datapoints provided. In this case, we update parameters only once for single epoch. The batch gradient uses exact gradient for training.

The batch gradient descent is computationally efficient due to fewer updates and its gradient is more stable in some problems. However, it is required to store all of data in the memory and easy to converge in local minima.

Secondly, the stochastic gradient descent or online learning uses single datapoint for single weight update. This is noisy update for weights and it allows to avoid local minima.

The SGD updates the parameters very frequently and this can result faster learning. But this can be hard for parallelism because of the bottleneck in parameter servers.

Finally, The minibatch updates splits the training dataset into small batches which called mini-batch and updates the parameters using each minibatchs. It updates the parameters (# of batches)-times for single epoch.

The minibatch training takes the advantages of the batch gradient descent and stochastic gradient descent. However, we need to decide the size of minibatch to implement the algorithm. The batch size is commonly chosen for 32 but it can be tuned in validation.

5.2 data parallelism

All of optimizers described in previous sections updates parameters according to gradients of loss function. Because of the computation time issue, we parallelize the training process with respect to the data or model itself. In this section we will focus on the data parallelism.

Since the loss function is additive with respect to the partition of data, we can divide the dataset into partition and provide each partitions to different workers.Then the workers would compute the gradient of the loss function using provided data.

After workers compute gradients, they send their outputs to parameter server and the parameter server will collect the gradients to update the parameters. For parameter server there are various update rules to update weights from the gradient informations. Typically, there are famous updates rules called synchronous updates and asynchronous updates.

1. Synchronous SGD
   In Synchronous SGD update, SSGD updates parameters using true gradient $G^{(t)} = \frac{1}{N}\sum_{l=1}^{M} G_l^{(t)}$ by waiting all workers to finish computing $G_l^{(t)}$.

Algorithm 2: SSGD worker m

Input dataset $\mathcal{X}$, minibatch size $\mathcal{B}$
for $t = 0, 1, \cdots$ **do**
  Wait to read $\theta^{(t)}$ from parameter server;
  $G_m^{(t)} := 0$;
  for $i = 1, \cdots, \mathcal{B}$ **do**
    Sample data $x_{k,i}^{\sim}$ from $\mathcal{X}$;
    $G_m^{(t)} \longleftarrow G_m^{(t)} + \frac{1}{\mathcal{B}}\nabla L(x_{k,i}^{\sim}, \theta^{(t)})$
  end
  Send $G_m^{(t)}$ to parameter server
end

Algorithm 3: SSGD parameter server

Input learning rate μ_t and number of workers M;
initialize $t \longleftarrow 0$;
initialize model $\theta^{(0)}$;
repeat
  Send $\theta^{(t)}$ to each workers;
  Wait for $G_1^{(t)}, \cdots G_M^{(t)}$ from each workers;
  $\theta^{(t+1)} \longleftarrow \theta^{(t)} - \frac{\mu_t}{N} \sum_{l=1}^{M} G_l^{(t)}$;
  $t \longleftarrow t + 1$
until *converges*;

Since SSGD waits for all of the workers, the training speed depends on the slowest worker. Thus there is waste of time if the difference between fastest worker and slowest worker.

To make SSGD faster, additional workers can be helpful. If we use k more workers and wait only first M gradients we can save training time.

However, since SSGD computes the true gradient of timestamp t, SSGD performs similarly with SGD without parallelism.

2. Asynchronous SGD [CPM+16]
   The ASGD worker computes the gradient of loss function similarly with SSGD worker. The different between SSGD and ASGD comes from the parameter server.

   The ASGD parameter server doesn't wait for the workers to finish computing gradients. It updates parameter θ^t immediately when the gradient g_m is provided. This makes ASGD faster but leads to the new problem called stale gradient or delayed gradient problem.

   The ASGD algorithm can be written as

Algorithm 4: ASGD worker m

Input dataset $\mathcal{X}$, minibatch size $\mathcal{B}$
for $t = 0, 1, \cdots$ **do**
  Read $\theta^{(t)}$ from parameter server;
  $g_m := 0$;
  for $i = 1, \cdots, \mathcal{B}$ **do**
    Sample data $\tilde{x_i}$ from $\mathcal{X}$;
    $g_m \longleftarrow g_m + \frac{1}{\mathcal{B}} \nabla L(\tilde{x_i}, \theta^{(t)})$
  end
  Send g_m to parameter server
end

Algorithm 5: ASGD parameter server

Input learning rate μ_t and number of workers M;
initialize $t \longleftarrow 0$;
initialize model $\theta^{(0)}$;
repeat
  Wait for g_m from any worker;
  $\theta^{(t+1)} \longleftarrow \theta^{(t)} - \mu_t g_m$;
  $t \longleftarrow t + 1$
until *converges*;

More precisely, if timestamp of parameter server is $t+\tau$, the provided gradient g_m with timestamp t is not the gradient which is required at position $\theta^{(t+\tau)}$. This penalizes the accuracy of the model. However, there are various solutions for the stale gradient problem were introduced later.

6 Post training

The training process described in previous sections optimizes(minimzes) the loss function until it converges to the local minima. After the training, there are some additional methods for model compression. In this section we will how they works.

6.1 Pruning

The network pruning process eliminates the connection or nodes which affects little on prediction. In detail, we repeat two steps:

1. eliminate connection with small weight(i.e. smaller than chosen threshold)
2. train changed model using training data

By pruning insignificant connection we can reduce the number of parameters and save memory and computational resources. According to ([HMD15]) The pruning can reduce number of parameters nearly 90% without big penalty on accuracy.

From the success of pruning, it is possible to presume that training method described in previous sections can be improved.

6.2 Low rank approximation

Since the parameters of deep learning are marix formed, they can be approximated by low rank matrices. This can be performed by truncated singular value decomposition. More precisely, for given

matrix W we compute the singular value decomposition to represent W as sum of rank 1 matrices

$$W = U\Sigma V^T = \sum_{i=1}^{n} \sigma_i U_i V_i^T \text{ where } \sigma_1 \geq \cdots \geq \sigma_n$$

and choose first R of them we can find $\bar{W} = \sum_{i=1}^{R} \sigma_i U_i V_i^T$ which is closest matrix with W of rank at most R.

The set of low rank matrices can be easily parametrized by fewer parameters because

$$\{W \in Mat_{n\times n}(\mathbb{R}) \mid rank(W) \leq R\} = \{W_1 W_2 \mid W_1 \in Mat_{n\times R}(\mathbb{R}), W_2 \in Mat_{R\times n}(\mathbb{R})\}.$$

this parametrization is easily implemented by inserting new hidden layer consist of R nodes and linear activations.

Therefore, we can approximate the trained weights with low rank matrices and additionally train on reparametrized model function. This would reduce the number of parameters without large panalty of accuracy.

On CNN architecture, the weight used in convolutional layer is given with order 4 tensor. Similarly with the case of matrix formed weights, it is possible to decompose the tensors into sum of simple tensors using canonical polyadic decomposition, which is generalization of svd in tensors. In [CWZZ17], It is reported that the tensor rank decomposition approach succeded to reduce the number of parameters without large penalty of accuracy.

I will see more about this method in final survey of this course.

References

[Alp20] Ethem Alpaydin. *Introduction to machine learning.* 2020.

[CPM+16] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[CWZZ17] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[HSS] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[LJH+19] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.