Adaptive Optimisation Methods

Tirtharaj Dash

Dept. of CS & IS and APPCAIR
BITS Pilani, Goa Campus

September 21, 2021

Where we are:

Summary of the previous lecture:

- Gradient Descent (SGD)
- Mini-batch GD
- SGD with momentum
- Parameter initialisation

Today, we will see some adaptive optimisation methods for fast and stable training of deep networks:

- RMSProp optimisation
- Adam optimisation
- Batch normalisation

Adaptive optimisation methods I

- SGD with momentum is one of the most commonly used method training DL models.
- In SGD, every single weight in the network is updated using an equation with same learning rate (global γ).
- There are other innovative methods that are shown to be useful for various problems. These fall under the class of adaptive methods.

Adaptive optimisation methods II

- Intuition:
  - Adaptive methods adapt a learning rate for each weight individually.
  - In modern day neural networks, the structure is deep. The layer which is closer to the output layer are somewhat directly interacting with the loss where as the layers towards the inputs are far away.
  - So, the update for the deeper layers (layers close to outputs) will be updated nicely even if the learning rate small. However, this will not be the case with the shallow layers.
  - The primary purpose of adaptive methods is to adaptive the learning a proper gradient update.
- Here we will look at:
  - RMSprop (Root Mean Square propagation)
  - Adam (Adaptive Moment Estimation)

Adaptive optimisation methods III

RMSprop:

- Idea: The gradient is normalized by its root-mean-square.
- Here is the equation:

$$\mathbf{v}_k = \beta \mathbf{v}_{k-1} + (1-\beta)\nabla L_i(\mathbf{w}_{k-1})^2$$

  (the squaring is element-wise)
- Notice that: The momentum factor that we computed in SGD-with-momentum was the 'first moment' (moving average). Here, we are calculating the second moment (squaring the gradient).

Adaptive optimisation methods IV

- The update then is:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma \frac{\nabla L_i(\mathbf{w}_{k-1})}{\sqrt{\mathbf{v}_k} + \epsilon}$$

- The parameter γ is the global learning rate. The parameter ϵ is a very small parameter to handle divided-by-0 issues during computation (usually: 10^{-8})

Adaptive optimisation methods V

Adam Optimisation:

- Idea: Combine the RMSprop and momentum algorithm together in some fashion.
- First moment:

$$\mathbf{m}_k = \beta_1 \mathbf{m}_{k-1} + (1 - \beta_1) \nabla L_i(\mathbf{w}_{k-1})$$

- Second moment:

$$\mathbf{v}_k = \beta_2 \mathbf{v}_{k-1} + (1 - \beta_2) \nabla L_i(\mathbf{w}_{k-1})^2$$

- Update the weights as:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma \frac{\mathbf{m}_k}{\sqrt{\mathbf{v}_k} + \epsilon}$$

Adaptive optimisation methods VI

- Usually, there will be a bias-correction term computed before the update. But, it is not shown here (and it won't matter as k increases). If you are bothered, here are these:

$$\mathbf{m}_k^{bc} = \frac{\mathbf{m}_k}{1 - \beta_1^k}$$

and similarly:

$$\mathbf{v}_k^{bc} = \frac{\mathbf{v}_k}{1 - \beta_2^k}$$

Adaptive optimisation methods VII

The update rule is then:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma \frac{\mathbf{m}_k^{bc}}{\sqrt{\mathbf{v}_k^{bc}} + \epsilon}$$

- Some good defaults for the hyperparameters are: $\gamma = 0.001$; $\beta_1 = 0.9$; $\beta_2 = 0.999$; and ϵ is fixed 10^{-8}.

Adaptive optimisation methods VIII

Is bias correction necessary?

- For implementation, the initial value of the first and second moments are set to 0; i.e. $\mathbf{m}_0 = \mathbf{0}$ and $\mathbf{v}_0 = \mathbf{0}$.
  - The initial values of the computed moments are lower than the true averages.
  - This error keeps cascading with following search itearations.
- Bias correction fixes the above error and allows to compute a more accurate moving average.
- Bias correction does play some role during the initial search iterations as the denominator is < 1; however, when k starts to be large, the denominator is just 1. Some implementations just ignore the bias correction step.

Normalising activations I

- It is a method of adaptive reparameterisation, motivated by the difficulty of training very deep models.
- Deep models are composed of several functions or layers. The backpropagation procedure tells us how to update the parameters of a functions and layers, under an assumption that other layers don't change.
- However, in practice, we update all of the layers simultaneously.
- When we make the update, unexpected results can happen because many functions composed together are changed simultaneously, using updates that were computed under the assumption that the other functions remain constant.

Normalising activations II

- Let's assume an activation at a layer ℓ be written as (ignoring the activation functions):

$$\mathbf{w}^{(1)}\mathbf{w}^{(2)}\dots\mathbf{w}^{(\ell)}\mathbf{x}$$

  When we do the backprop, the updates to the parameters will have ℓ-order effect on the activation, that is:

$$(\mathbf{w}^{(1)} - \gamma\nabla L)(\mathbf{w}^{(2)} - \gamma\nabla L)\dots(\mathbf{w}^{(\ell)} - \gamma\nabla L)\mathbf{x}$$

  (assume the gradients appropriately)
- This update could lead to either negligible activation or a very large activation.

Normalising activations III

- So, what we are seeing is that the update to a parameter in any layer is strongly influencing the activation at a latter stage. And, no matter what learning rate we set, this effect will not go away easily.
- Normalising the activations is an approach to neutralise such effects by forcing the activation to follow a mean of 0 and a variance.

Normalising activations IV

Batch Normalisation:

- A normalisation strategy to reparameterise any deep network.
- Let $\mathbf{Z}$ be a minibatch of activations of the layer to normalise (let it be ℓ).
- Each row of $\mathbf{Z}$ consists of net sums corresponding to an example of the minibatch. That is:

$$\mathbf{Z} = \begin{bmatrix} z_{1,1}^{(\ell)} & z_{1,2}^{(\ell)} & \cdots & z_{1,m^{(\ell)}}^{(\ell)} \\ z_{2,1}^{(\ell)} & z_{2,2}^{(\ell)} & \cdots & z_{2,m^{(\ell)}}^{(\ell)} \\ \vdots & \vdots & \vdots & \vdots \\ z_{|B_k|,1}^{(\ell)} & z_{|B_k|,2}^{(\ell)} & \cdots & z_{|B_k|,m^{(\ell)}}^{(\ell)} \end{bmatrix}$$

where $z_{i,j}^{(\ell)}$ denotes the netsum at node j of the layer ℓ for the ith instance in the minibatch B_k; k denotes iteration.

Normalising activations V

- Note that: Some implementations apply batch-normalisation (BN) after applying the activation function. Here we are doing it on pre-activated values (netsums). It doesn't really matter whether we apply BN before or after activation.
- The normalised matrix $\mathbf{Z}$ is then

$$\mathbf{Z}' = \frac{\mathbf{Z} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

  where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit.
- You may see that: $\boldsymbol{\mu} \in \mathbb{R}^{m^{(\ell)}}$, where $m^{(\ell)}$ is the number of nodes in the layer ℓ; and $\boldsymbol{\sigma} \in \mathbb{R}^{m^{(\ell)}}$. That is both these vectors of sizes equal to the number of nodes in layer ℓ.

Normalising activations VI

- The arithmetic here is based on broadcasting the vector $\boldsymbol{\mu}$ and the vector $\boldsymbol{\sigma}$ to be applied to every row of $\mathbf{Z}$.
- Within each row the arithmetic is element-wise, so $Z_{i,j}$ is normalised by substracting μ_j and dividing by σ_j:

$$Z'_{i,j} = \frac{Z_{i,j} - \mu_j}{\sigma_j}$$

- The rest of the network then operates on $\mathbf{Z}'$ the same way it would have operated on $\mathbf{Z}$ (that is, activation and feedforward to the next layer).

Normalising activations VII

- At training time:
  - The vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are calculated as:

$$\mu_j = \frac{1}{|B_k|} \sum_{i=1}^{|B_k|} \mathbf{Z}_{i,j}$$

$$\sigma_j = \sqrt{\delta + \frac{1}{|B_k|} \sum_{i=1}^{|B_k|} (\mathbf{Z}_{i,j} - \mu_j)^2}$$

where δ is a small positive value such as 10^{-8}, to avoid the undefined gradient when the denominator is 0.

Normalising activations VIII

- At the test time:
  - The vectors μ and σ may be replaced by running averages that were collected during the training time.
  - This allows the model to be evaluated on a single example without needing to use definitions of μ and σ that depends on an entire minibatch.

Normalising activations IX

- What we saw here is that: We are enforcing that the activations in a layer must follow a Gaussian distribution with mean at 0, that is, $\mathcal{N}(0, \boldsymbol{\sigma}^2)$.
  - What if this is not a right distribution for the layer? That is, by enforcing this we are taking away some expressive power of the network.
  - Rather, what we should do is enfore a Gaussian distribution on the activations but allow the network to pick the distribution parameters.
  - That is:

    $$\mathbf{Z}'' = \alpha \mathbf{Z}' + \beta$$

    where α and β are learnable parameters to be obtained by using backpropagation, requiring additional gradient computations.
  - This equation is linearly transforming the normalised values to new values suitable for the network.

Normalising activations X

- Effect of BN on training of deep networks:
  1. *Relation to internal covariate shift:* Internal covariate shift refers the change in the distribution of layer inputs caused by updates to the preceding layers. These continual changes impacts training negatively. BN reduces this effect.

     Note: Although the above relation is(was) widely accepted, there is little evidence to support it. Refer How does batch normalization help optimization? NIPS 2018.

Normalising activations XI

2. *BN helps optimisation*: BN reparametrises the underlying optimisation problem to make it more stable and smooth. This implies that the gradients used in training are more predictive and well-behaved, which enables faster and more effective optimization.

3. *BN improves generalisation*: The smoothening effect of BN's reparametrisation encourages the training process to converge to more flat minima. Such minima are believed to facilitate better generalization.