Optimisation

Tirtharaj Dash

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

September 18, 2021

Where we are:

Summary of the previous lecture:

- Dropout regularisation
- Regularisation via early stopping

Today, we will see some standard optimisation approaches for deep network training:

- Gradient descent and its variants
- Parameter initialisation strategies

Gradient Descent I

- We have already done some basic tutorials on Gradient Descent (GD).
- We saw that our objective is:

$$\min_{\mathbf{w}} L(\mathbf{w})$$

  Meaning: find out the parameters $\mathbf{w}$ that minimises the loss L.
- The gradient descrent equation to update the parameters $\mathbf{w}$ at some iteration k (denoted as $\mathbf{w}_k$) is:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma_k \nabla L(\mathbf{w}_{k-1})$$

Gradient Descent II

- This is an iterative update equation that updates $\mathbf{w}$ to its new values based on the direction provided by the gradient of L at the older value of $\mathbf{w}$.
- The parameter γ_k is the learning rate (or the step-size parameter) at the present iteration k.

Note 1: Earlier we have not used iteration-specific γ_k. But, here we are keeping it more general.

Note 2: Gradient ∇L is a vector. Recall that the gradient of a scalar w.r.t. a vector is a vector.

Gradient Descent III

Few things to consider:

- We assumed that the function L is continuous.
- L may have a convex or non-convex surface.
- We aim to find the lowest point of this surface.

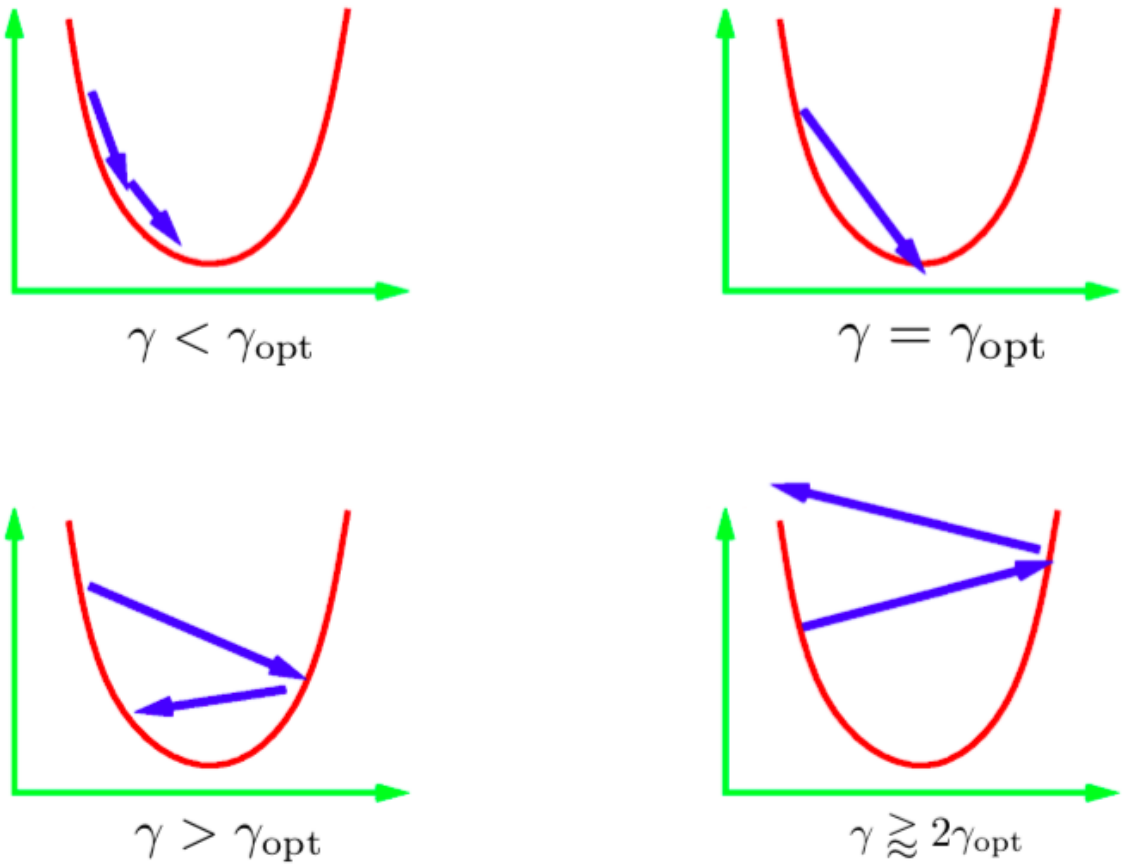
Gradient Descent IV

Further:

- We don't know the direction of the lowest point.
- Therefore, we take a local step based on the gradient (or slope) at our present point $\mathbf{w}_k$.
- Iteratively, we are taking a step locally and moving (slowly) towards the lowest point.
- If the surface is convex, we will reach a point which will be lowest glabally; otherwise, it will be a locally lowest point.

Gradient Descent V

The parameter γ_k:



(Source: https://atcold.github.io/pytorch-Deep-Learning/)

Gradient Descent VI

- $\gamma < \gamma_{opt}$: slow convergence; but, will reach minimum
- $\gamma = \gamma_{opt}$: ideal case; difficult to know what is γ_{opt}
- $\gamma \geq 2\gamma_{opt}$: diverges from the minimum (The search becomes somewhat like random search)
- $\gamma > \gamma_{opt}$: a good setting (but, again we don't know γ_{opt}; therefore, we search through a set of γ values to determine which one suits to our search needs)

Stochastic GD I

- We replace the actual gradient vector with stochastic estimation of this.
- For minimising a loss function during training of a neural network, this means that the gradient of the loss function is the gradient of the loss for a single stochastically (randomly) drawn training instance.

Stochastic GD II

- Let L_i denote the loss of the network for ith training instance:

$$L_i = loss(y_i, f(\mathbf{x}_i; \mathbf{w}))$$

The function that we eventually want to minimize is L, the total loss over all instances.

$$L = \frac{1}{n} \sum_{i}^{n} L_i$$

Stochastic GD III

- In SGD, we update the weights according to the gradient over L_i (as opposed to the gradient over L):

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

  i is chosen uniformly at random.

- If i is chosen randomly, then L_i is a noisy but unbiased estimator of L. That is:

$$\mathbb{E}[\nabla L_i(\mathbf{w}_{k-1})] = \nabla L(\mathbf{w}_{k-1})$$

- We can therefore write:

$$\mathbb{E}[\mathbf{w}_k] = \mathbf{w}_{k-1} - \gamma_k \mathbb{E}[\nabla L_i(\mathbf{w}_{k-1})] = \mathbf{w}_{k-1} - \gamma_k \nabla L(\mathbf{w}_{k-1})$$

Stochastic GD IV

- This noisy version of update also allows the search to be more effective (practically) allowing searching over multiple local minimum surfaces:

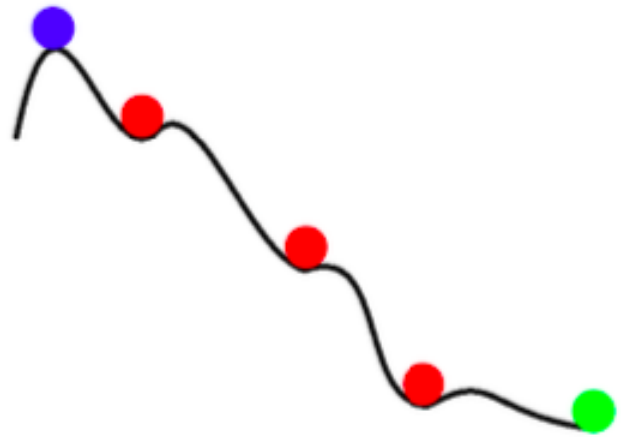


Fig: annealing process

(source: https://atcold.github.io/pytorch-Deep-Learning/)

Stochastic GD V

Summary:

- There is a lot of redundant information across instances.
- SGD prevents a lot of these redundant computations which GD would have made.
- SGD step is virtually as good as a GD step given the information in the gradient.

Stochastic GD VI

- SGD searches over multiple local minimum surfaces (annealing). Note that the loss surface for a neural network consists of a large number of local minima situated very closed to each other.
- SGD is drastically cheaper to compute (as you don't go over all data points).
  - Example: Imagine a problem with 5,000,000 data points, each with 1,000,000 features.

Mini-batch GD I

- We compute loss over multiple randomly selected instances instead of calculating it over just one instance.
- This reduces the *noise* in the step update.

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma_k \frac{1}{|B_k|} \sum_{j \in B_k} \nabla L_j(\mathbf{w}_{k-1})$$

where B_k is a (mini-)batch of instances at training iteration k.

Note: $|B_k| = 1$ results in SGD, and $|B_k| = n$ (where n denotes the total number of instances) results in Batch-GD.

Mini-batch GD II

- The (forward propagation $+$) loss computation for each instance is independent of other instances, and therefore, $|B_k|$ instances can be processed in parallel.
- Gradients are then collected and backpropagated once.
- This parallel computation exploits the power of GPUs or TPUs and distributed training.

Note: Don't use GD for full-size batch. Use Limited-memory BFGS algorithm allowing construction of matrix-vector product on-the-fly. Refer some PyTorch examples, for more details.

SGD with momentum I

- Speeds up the convergence by using a moving average of the gradient:

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

  The update to $\mathbf{w}$ is then:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma_k \mathbf{m}_k$$

- $\mathbf{m}$ is called the momentum factor.

SGD with momentum II

- At each update step we add the stochastic gradient to the old value of the momentum, after dampening it by a factor $\hat{\beta}$ ($\in [0, 1)$).
- **m** can be thought of as a running average of the gradients.
- Finally we move **w** in the direction of the new momentum **m**.

SGD with momentum III

- Alternative form:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma_k \nabla L_i(\mathbf{w}_{k-1}) + \beta_k(\mathbf{w}_{k-1} - \mathbf{w}_{k-2})$$

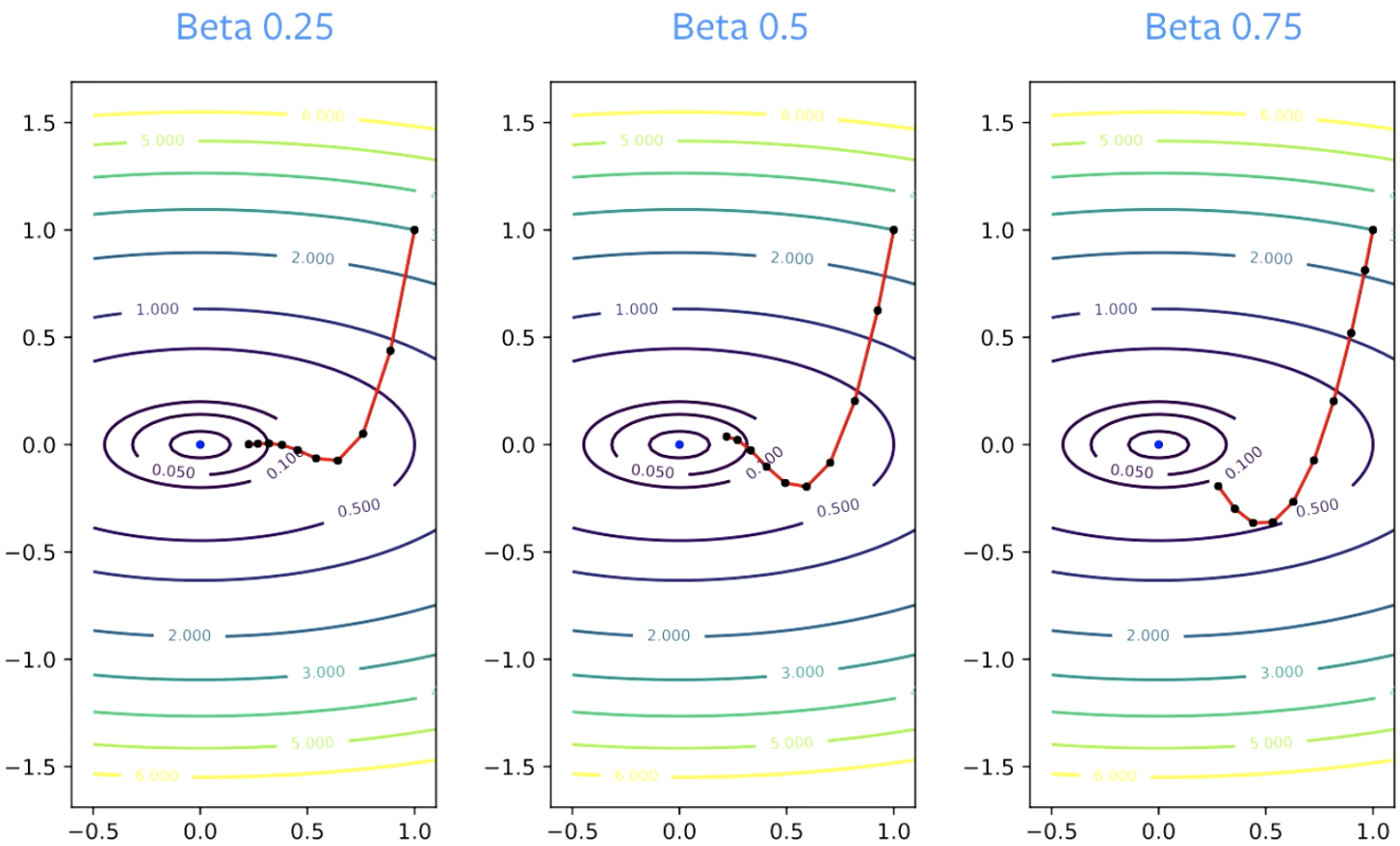
- That is: the next step is a combination of previous step's direction $(\mathbf{w}_{k-1} - \mathbf{w}_{k-2})$ and the new negative gradient.

SGD with momentum IV

How does it work?

SGD with momentum V

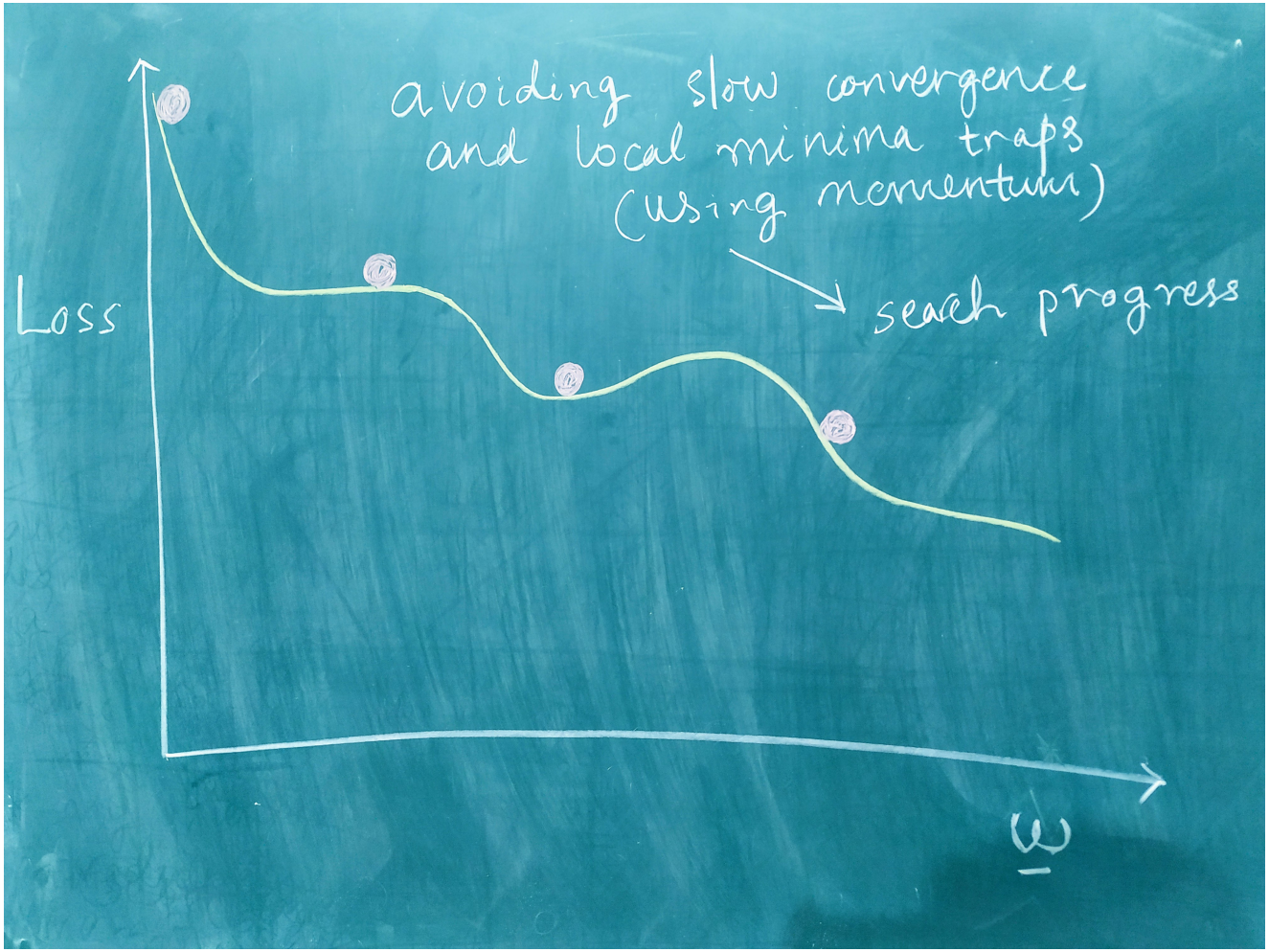
- The parameter β behaves as a damping factor ('friction' or 'braking' parameter) during oscillation of the movement.



(Source: https://atcold.github.io/pytorch-Deep-Learning/)

SGD with momentum VI

- Can navigate complex loss surfaces: helps the search process retain speed in flat regions of the loss surface and avoid local optima.



Parameter Initialisation I

Choosing a good starting point is important:

- *Symmetry:* If all the parameters are initialised to the same value (for any layer ℓ, $\mathbf{w}^{(\ell)} = c$ for some constant $c \in \mathbb{R}$), they do not differ through out training and we might not realise the network's expressive power. This is called the symmetry problem.
  - A good starting point breaks the symmetry.
- *Local minimum trap:* If the parameters are initialised to a bad starting point, GD gets into a trap in some local minimum.
  - A good starting point could allow GD to result in a good solution.

Parameter Initialisation II

There are some heuristics for choosing a good starting point for $\mathbf{w}$:

- We can think of initializing the parameters $\mathbf{w}$ to $\mathbf{w}_0$ as being similar to imposing a Gaussian prior $\mathbf{w}$ with mean $\mathbf{w}_0$. That is:

$$g(\mathbf{w}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{w}_0, \sigma^2)$$

- Usually $\boldsymbol{\mu} = \mathbf{0}$: says that it is more likely that the units do not interact with each other.

Parameter Initialisation III

- Let $w_{ji}^{(\ell)}$ refer to the weight parameter for the connection between unit i in some layer $(\ell - 1)$ and a unit j in layer (ℓ).
- For parameters for a fully connected layer with n_{in} inputs and n_{out} outputs, some heuristics look like:
  - For all i, j: $w_{ji} \sim U\left(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right)$
  - For all i, j: $w_{ji} \sim U\left(-\frac{6}{\sqrt{n_{in}+n_{out}}}, \frac{6}{\sqrt{n_{in}+n_{out}}}\right)$ (Xavier initialisation)

Parameter Initialisation IV

- If you look at some papers, there are roughly the following initialisation heuristics suggested:

Activation function	Uniform distro $[-r, +r]$	Normal distro $(\mu = 0)$
tanh	$r = \sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = \sqrt{\frac{2}{n_{in}+n_{out}}}$
sigmoid	$r = 4\sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = 4\sqrt{\frac{2}{n_{in}+n_{out}}}$
ReLU (and variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{in}+n_{out}}}$

- At any rate, we are setting the weights to very small random numbers. In moderate problem sizes, default initialisation (Uniform distro $[-1, +1]$) works alright.

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 eqution 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.