Deep Learning Refresher

Week 5

Tirtharaj Dash

Gradient Descent I

- We have already done some basic tutorial on Gradient Descent (GD).
- The objective is:

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

- Gradient descent requires computation of gradient* of f w.r.t. $\mathbf{w}$ at some iteration k: $\nabla f(\mathbf{w}_k)$
- Then, the iterative solution to the above minimisation objective is:

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

Here, γ_k is the step size parameter.

*: Gradient is a vector.

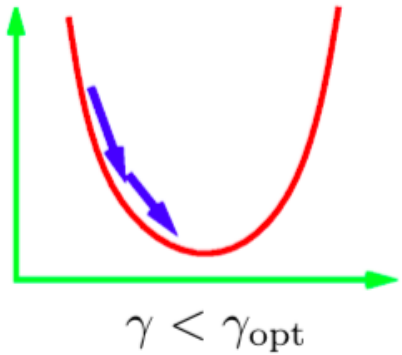
Gradient Descent II

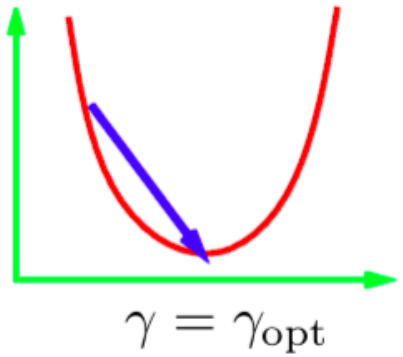
Few things to consider:

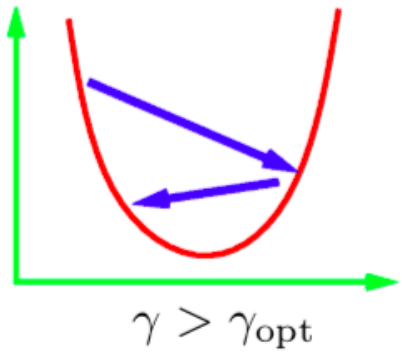
- We assumed that the function f is continuous.
- f may have a convex or non-convex surface.
- We aim to find the lowest point of this surface.
- We don't know the direction of the lowest point.
- Therefore, we take a local step based on the gradient (or slope) at our present point $\mathbf{w}_k$.
- 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.

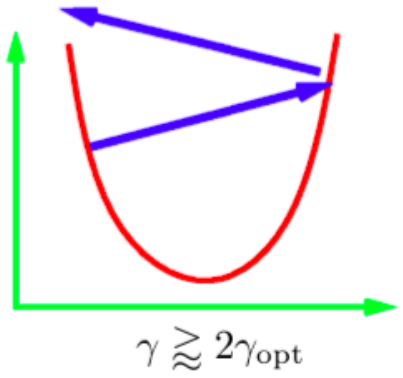
Gradient Descent III

The parameter γ_k:









Gradient Descent IV

- $\gamma < \gamma_{opt}$: 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 repalce 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.
- Let f_i denote the loss of the network for ith training instance:

$$\mathcal{L}_i = loss(y_i, g(\mathbf{x}_i, \mathbf{w}))$$

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

$$\mathcal{L}_{avg} = \frac{1}{n} \sum_{i}^{n} \mathcal{L}_i$$

Stochastic GD II

- In SGD, we update the weights according to the gradient over $\mathcal{L}_i$ (as opposed to the gradient over $\mathcal{L}_{avg}$):

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

  i is chosen uniformly at random.

- If i is chosen randomly, then $\mathcal{L}_i$ is a noisy but unbiased estimator of $\mathcal{L}$, which is written as:

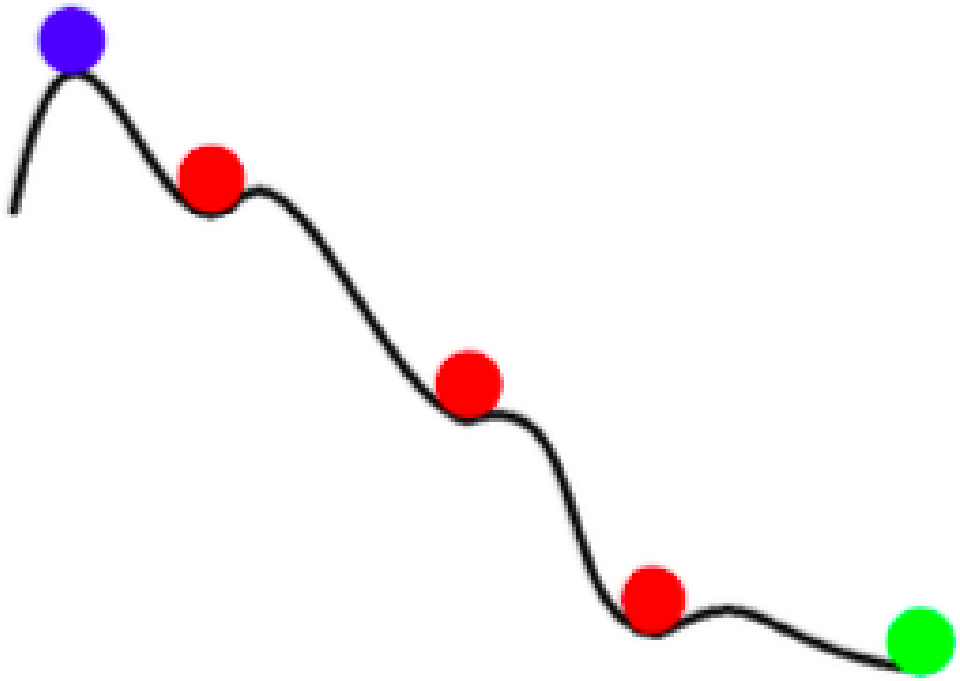
$$\mathbb{E}[\nabla \mathcal{L}_i(\mathbf{w}_k)] = \nabla \mathcal{L}_{avg}(\mathbf{w}_k)$$

- We can therefore write:

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

Stochastic GD III

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



(annealing process)

Stochastic GD IV

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.
- SGD searches over multiple local minimum surfaces (annealing)
- Stochastic Gradient Descent is drastically cheaper to compute (as you don't go over all data points).
  Example: Imaging 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.

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma_k \frac{1}{|B_i|} \sum_{j \in B_i} \nabla \mathcal{L}_j(\mathbf{w}_k)$$

- The (forward processing+) loss computation for each instance is independent of other instances, and therefore, $|B_i|$ instances can be processed in parallel.
- Gradient are then collected and backpropagated once.
- Example: Parallel computation in GPUs/TPUs; distributed training
- Note Don't use GD for full-size batch. Use Limited-memory BFGS algorithm (refer some PyTorch examples)

SGD with momentum I

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

$$\mathbf{p}_{k+1} = \hat{\beta}\mathbf{p}_k + \nabla\mathcal{L}_i(\mathbf{w}_k)$$

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

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

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

SGD with momentum II

- Alternative form:

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

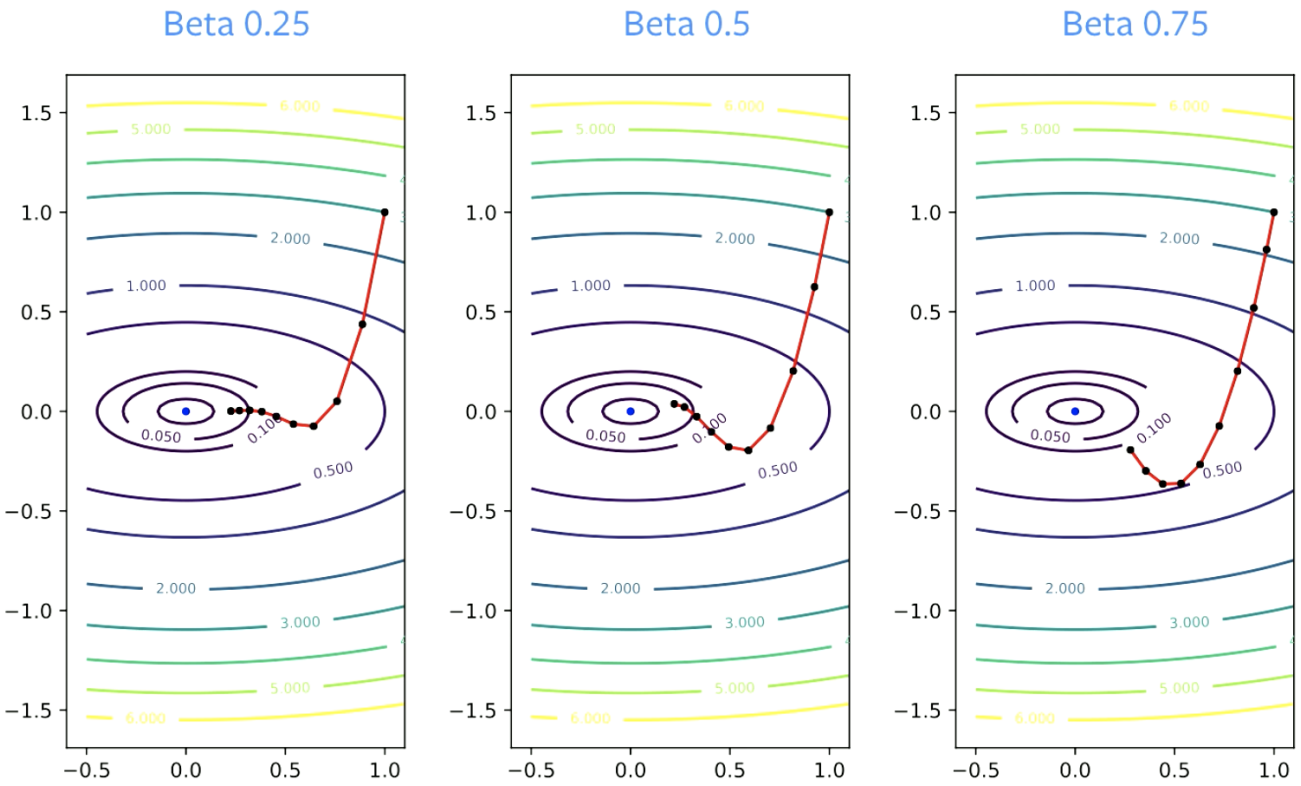
- That is: the next step is a combination of previous step's direction $(\mathbf{w}_k - \mathbf{w}_{k-1})$ and the new negative gradient.

SGD with momentum III

How does it work?

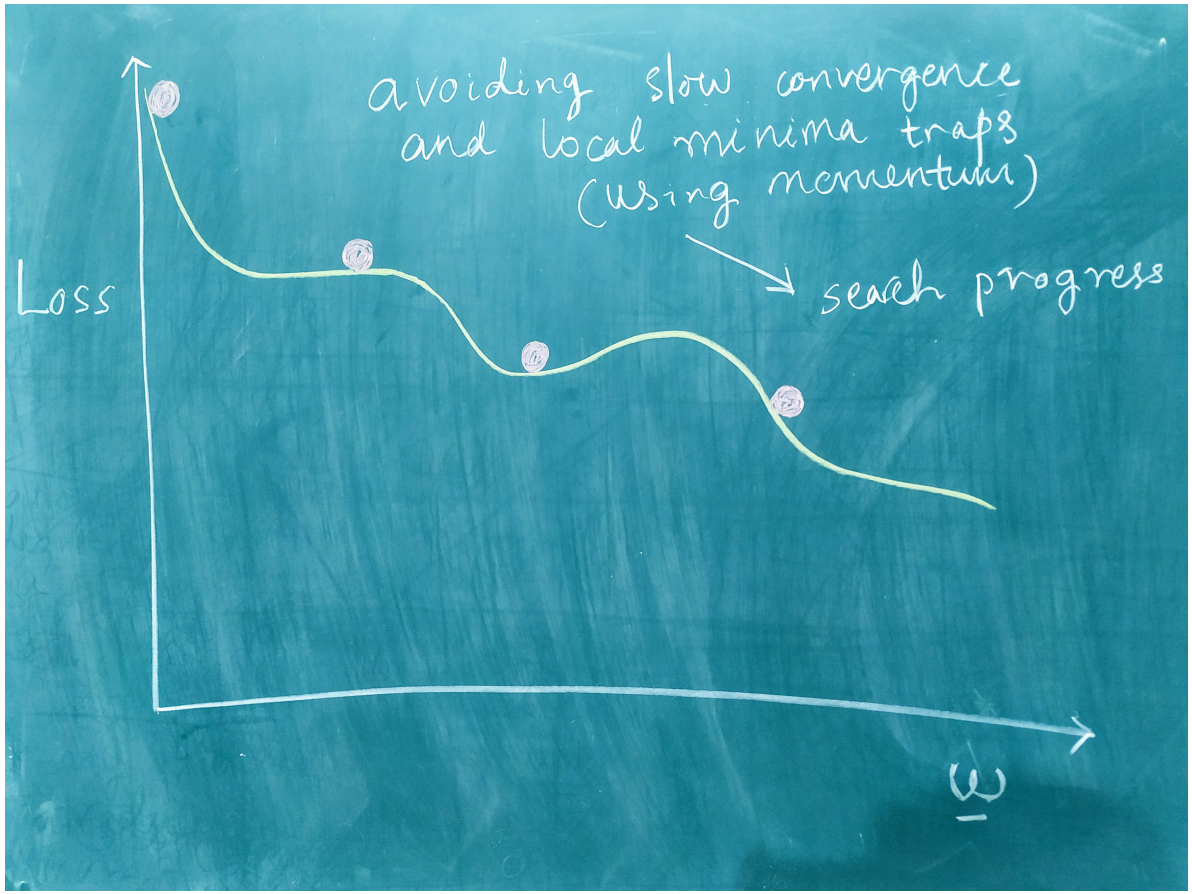
SGD with momentum IV

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



SGD with momentum V

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



Chosing a good starting point I

- GD ends a bad local optima solution if the starting point is bad.
- Therefore, it would benefit us if we could start the search from a good starting point.
- There are some heuristics for choosing a good starting point for the $\mathbf{w}$ parameters (includes bias).
- 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 $\mathbf{w}_0$, i.e. $f(\mathbf{w}) = \mathcal{N}(\boldsymbol{\mu} = \mathbf{w}_0, \sigma^2)$.
- Usually $\boldsymbol{\mu} = \mathbf{0}$: says that it is more likely that units do not interact with each other than that they do interact. ($w_{ij}^{(l)}$ refers to the weight parameter for the connection between unit i in some layer $(l-1)$ and a unit j in layer (l))

Chosing a good starting point II

- For parameters for a fully connected layer with n_{in} inputs and n_{out} outputs, some heuristics look like:
  - For all $i, j,$: $w_{ij} \sim U\left(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right)$
  - for all i, j: $w_{ij} \sim U\left(-\frac{6}{\sqrt{n_{in}+n_{out}}}, \frac{6}{\sqrt{n_{in}+n_{out}}}\right)$ (Xavier initialisation)
- 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.

Adaptive optimisation methods I

- SGD with momentum is one of the most commonly used method in ML optimisation.
- 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.
- These 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.

Adaptive optimisation methods II

- 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 are the equations:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + (1 - \beta) \nabla \mathcal{L}_i(\mathbf{w}_t)^2$$

(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:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \frac{\nabla \mathcal{L}_i(\mathbf{w}_t)}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

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

Adaptive optimisation methods V

Adam Optimisation:

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

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \nabla \mathcal{L}_i(\mathbf{w}_t)$$

- Second moment:

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2) \nabla \mathcal{L}_i(\mathbf{w}_t)^2$$

- Update the weights as:

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

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 you t increases). If you are bothered, here are these:

$$\mathbf{m}_{t+1}^{bc} = \frac{\mathbf{m}_{t+1}}{1 - \beta_1^{t+1}}$$

and similarly:

$$\mathbf{v}_{t+1}^{bc} = \frac{\mathbf{v}_{t+1}}{1 - \beta_2^{t+1}}$$

The update rule is then:

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

- 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 VII

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 $\mathbf{v}_0 = \mathbf{0}$. So, what happens is that the initial values of the computed moments are lower than the true averages. This error keeps cascading with following search itearations. Bias correction fixes this error and allowes 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 t 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 reparametrization, 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 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):

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

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

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

(assume the gradients appropriately)

- This update could lead to either negligible activation or a very large activation. 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 activations III

- Normalising the activations is an approach to neutralise such effects by forcing the activation to follow a mean of 0 and a variance.
- The calculations are thus (at any layer the netsums for m units be $z_1, z_2, \ldots, z_m$):

$$\mu = \frac{1}{m} \sum_{i}^{m} z_i$$

$$\sigma^2 = \frac{1}{m} \sum_{i}^{m} (z_i - \mu)^2$$

and then the normalised activation is:

$$z_i^{normalised} = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Normalising activations IV

- There is still a problem though: By enforcing a layer to have activations around mean of 0 and a spread of σ^2 may not be good idea from the model's point of view. That is: what if mean of 0 for layer 5 is not really practical; but a mean of say 1.76 is.
- So, in such a case, we can ask the model to learn its own mean and spread from the activations that it has computed by using:

$$\tilde{z}_i = \alpha z_i^{normalised} + \beta$$

  Here, the parameters α and β are trainable parameters that model can learn using backprop.
- So what the above equation doing is: it is linearly transforming the normalised activations to some other value that is appropriate given the data.