Gradient Checks
Use the centered formula: $\frac{df(x)}{dx} = \frac{f(x+h)-f(x-h)}{2h}$, because it as error terms on order of O(h^2) (i.e. it is a second order approximation).

Relative error: $\frac{|f'_a - f'_n|}{max(|f'_a|, |f'_n|)}$, which considers their ratio of the differences to the ratio of the absolute values of both gradients.

Stick around active range of floating point: make sure that the numbers you are comparing are not extremely small.

Use only few datapoints: One fix to the problem of kinks.

Be careful with the step size h

Don’t let the regularization overwhelm the data.

Remember to turn off dropout/augmentations. When performing gradient check, remember to turn off any non-deterministic

Check only few dimensions: in practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient and assume that the others are correct.

Before learning: sanity checks Tips/Tricks
- Look for correct loss at chance performance.
- Overfit a tiny subset of data.

Babysitting the learning process: these plots are the window into the training process and should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.
- The first quantity that is useful to track during training is the loss
- The second important quantity to track while training a classifier is the validation/training accuracy. This plot can give you valuable insights into the amount of overfitting in your model.
- The last quantity you might want to track is the ratio of the update magnitudes to the value magnitudes. Note: updates, not the raw gradients.
- Activation / Gradient distributions per layer: an incorrect initialization can slow down or even completely stall the learning process. Luckily, this issue can be diagnosed relatively easily. One way to do so is to plot activation/gradient histograms for all layers of the network.

Parameter updates:
Vanilla update: $x \mathrel{+}= -\ learning\_rate * dx$
Momentum update:
- $v = mu * v - learning\_rate * dx \# integrate\ velocity$

- x $+= v$ *# integrate position*

Nesterov Momentum: It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistently works slightly better than standard momentum.

Annealing the learning rate:

- Step decay: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.
- Exponential decay. has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where α0,k are hyperparameters and t is the iteration number (but you can also use units of epochs).
- $1/t$ decay has the mathematical form α=α0/(1+kt) where a0,k are hyperparameters and t is the iteration number.

Second order methods:

- Newton’s method, which iterates the following update: $x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$. Here, Hf(x) is the Hessian matrix, which is a square matrix of second-order partial derivatives of the function. The term ∇f(x) is the gradient vector, as seen in Gradient Descent. Intuitively, the Hessian describes the local curvature of the loss function, which allows us to perform a more efficient update. In particular, multiplying by the inverse Hessian leads the optimization to take more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature. Note, crucially, the absence of any learning rate hyperparameters in the update formula, which the proponents of these methods cite this as a large advantage over first-order methods.