1. Gradient descent

* The method we use to train all our models (ANNs, CNNs, RNNs, etc.)
* Backbone of deep learning
* Can be used to train other ML models
* K-mean clustering
* Hidden Markov models
* Matrix factorization
* If you’re an expert in mathematics, you may be interested in:
* ‘Gradient Descent: Convergence Analysis’ in extra\_reading.txt
* Why do we need gradient descent
* What problem we are trying to solve
* For model (e.g., linear regression), we come up with a cost/loss
* We want to minimize the cost (L) with respect to parameters (w)
* Calculus – how do we minimize a function in calculus?
* We find its derivative and set it to 0
  + E.g.:
* Works because: **Slope at the minimum / maximum of a function is 0**
* How do we know if it’s a minimum or maximum of a function?
* We built the loss function
* What about local minima or saddle points?
* Not a problem in modern deep learning
* Why can’t we just take the gradient and set it to 0, then solve for w?
* Some equations are simply not solvable
* Numerical approximations
* What do we do when we can’t solve an equation?
* Approximate it
* If you can’t solve an integral, you can simply draw little trapezoids and find the area of the trapezoids instead
* Estimation of area under the curve
* How does gradient descent work?
* Idea: repeatedly take small steps in the direction of the gradient to find a new w
* At each step, L(w) decreases provided the step size is small enough
* Eventually, it will converge to the minimum
* Gradient descent in code

A close-up of a black text

Description automatically generated

* Example
* Minimize
* Gradient is
* Inside the loop:
* The solution is
* Hyperparameters
* Each iteration of the loop is called an epoch, and we must choose the number of epochs high enough so the cost converges
* is the learning rate – must be small enough so that the cost does not blow up, but large enough so that you don’t have to wait longer than necessary
* Trial and error is how you choose them
* You can also use automated methods like Bayesian optimization
* Practice is best

1. Stochastic gradient descent (SGD)

* A simpler problem
* Suppose we want to measure the average height of everyone in the world
* 7 billion people
* Too many to survey
* What if just asked 1000 people (chosen randomly)
* Do we expect the average of this 1000 to be similar to the population average? Yes
* Advantage: asking 1000 people takes less time
* How is this related to deep learning?
* Recall: we must calculate the cost and the gradient of the cost
* Cost depends on number of samples in the dataset
* What if we are using the ImageNet dataset and we have N = 1 million
* Take long time to add up errors of 1 million sample
* What if we just took the average error over 1000 images instead?
* This average error is probably close to the average error on the entire dataset
* Gradient is also similar, but 1000x less computation!
* Stochastic (batch) gradient descent
* Typically we use smaller batch sizes, like 32, 64, or 128

A screenshot of a computer code

Description automatically generated

1. Momentum

* Most performance-improving in plain SGD
* Without momentum:
* Gradient descent momentum
* Every time we want the box to move, we have to push it again
* Difficult
* Gradient descent, without momentum
* **If is 0 -> parameter doesn’t change!**
* Gradient descent, with momentum
* 2 steps: sliding on ice & pushing the box
* Typical values of are 0.9, 0.95, 0.99, etc.
* **Without any g, the box ‘slows down’**
* Effect of
* We just get back regular gradient descent
* Effect of momentum – speeds up training

A graph of a line

Description automatically generated with medium confidence

* Another perspective

A diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of

Description automatically generated

1. Variable and adaptive learning rates

* We’ve seen that momentum can greatly speed up training
* Variable learning rates
* Learning rates is a function of time, e.g.,
* Learning rate scheduling
* #1: Step decay: periodically reduce the learning rate by a constant factor
* #2: exponential decay – learning rate follows an exponential curve
* #3: 1/t decay – drop off is slower than exponential decay
* Learning rate decreases with time
* Initially, weights are far from optimal – as we get closer to the minimum, learning rate may be too large
* Babysitting method
* Do a few epochs, see how it goes (remember, we’re plotting the costs)
* Too slow -> increase learning rate
* Be careful! May hit a plateau temporarily
* Caveat:
* More hyperparameters you have to optimize
* Even choosing between these methods is a hyperparameter choice
* Adaptive learning rate techniques
* AdaGrad
* Dependence of cost on each param is not the same
* Steep gradient in one direction, flat in another
* Adapt the learning rate for each param individually, based on how much ‘learning’ it has done so far
* AdaGrad
* Each parameter of the neural network has its own cache
* E.g. 1 weight matrix of size 3 x 4 -> 1 cache matrix of size 3x4
* Typically, epsilon is small, around
* Cache will always be positive
* If a parameter has had large gradients in the past -> large cache -> effective learning rate will be very small -> change more slowly in the future
* If a parameter has had small gradients in the past -> small cache -> effective learning rate will remain large
* Everything is element-wise
* Each scalar param and its learning rate is updated independently of the others
* RMSProp
* It has been observed that AdaGrad decreases learning rate too aggressively
* Still more learning to be done
* Introduced Geoff Hinton + team
* Since cache is growing too fast, let’s decrease it on each update:
* Typical values for decay: 0.99, 0.999, etc.
* We say the cache is ‘leaky’
* Note: there is some ambiguity in the RMSProp update
* What is the initial value of cache?
* One might automatically assume 0
  + Let decay = 0.999
  + Initial cache =
  + Initial update (ignoring epsilon):
* Very large initial update
* Compensate by making learning rate smaller than usual
* One solution: initialize cache = 1 instead
* Approximate equals to no RMSProp at the beginning
* Which one is right? Neither! (it was never specified)
  + Major packages have implemented both
  + Tensorflow initializes cache = 1
  + Keras initializes cache = 0
* Summary pseudocode

A white background with black text

Description automatically generated

1. Adam

* Adaptive moment estimation
* Go-to-method for optimizing neural networks today
* Often used as a default choice w/o considering others
* It works well, and its robust with default settings (learning rate, etc.)
* Is a successor to RMSProp
* ‘RMSProp with momentum’, but not exactly the same
* Review so far – 2 helpful techniques:
* Momentum – alternate way of representing momentum
* Adaptive learning rate (e.g., RMSProp) – cache is now v
* A digression into moving averages
* Why are we doing this?
* To make sense of why momentum and RMSProp take these forms
* Let’s begin with the regular average
* Given
* Problems with this calculation
  + What if we have so much data, it can’t fit into memory
  + Or imagine a robot interacting with the world. At each time step T, we collect a new measurement . If I need to calculate the average value of X, just add all X’s so far / T
* This computation gets longer as more X’s are collected
* O(T) time complexity – number of steps is proportional to T
* We can make this constant by using previous computations
* A bit of algebra
* A bit more algebra
* Replace 1/T with a constant
* This will change the answer (but may be useful)
* 1/T gives us the sample mean (notice: 1/T decreases as T increases)
* If 1/T is a constant, then we no longer have the sample mean!
* Instead, we have the **exponentially weighted moving average**
* Exercise: manipulate the below equation to represent the mean in terms of X’s only to show that the ‘weights’ for each X decay exponentially
* A minor replacement
* Moments
* In statistics, the first and second moments are quantities we typically care about (related to mean and variance)
* Momentum and RMSProp

A math equations with numbers and symbols

Description automatically generated with medium confidence

* Building Adam: combining momentum and RMSProp

A diagram of mathematical equations

Description automatically generated

* One more step to Adam
* We saw the exponential weighted moving average (EWMA) as an alternative to the simple average
* Useful when data is nonstationary (changes over time)
* What do we get y(t) when we apply EWMA to some time series input x(t)?
* Smoothing
* Y(t) is a smoothed version of x(t)
* In signal processing, this is also called a low pass filter
* Low frequency movements are typically larger in magnitude
* High frequency movements are small
* problem with our low pass filter
  + y(t) always depends on the last value
  + What is the ‘first last value’?
* There is a bias towards 0, takes time to catch up
* Bias correction

A math equations and numbers

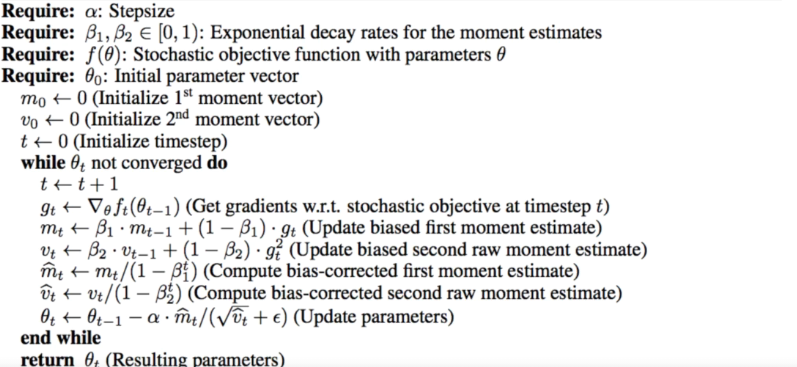
Description automatically generated with medium confidence

* Plugging in some numbers

A math problem with numbers and a green bubble

Description automatically generated

* Incorporating bias correction into Adam
* Replace
* **Adam pseudocode**



* **Default hyperparams values**
* Tensorflow and PyTorch already use these default values
* learning\_rate = 0.001
* beta\_1 = 0.9
* beta\_2 = 0.999
* epsilon = 1e-7 (Tensorflow) or 1e-8 (PyTorch)

A screenshot of a computer

Description automatically generated

* Is Adam always the best?
* SGD with momentum
* ML is experimentation, not philosophy
* Try and see for yourself