Quiz 7

1. Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update. Please describe the algorithm and principle of the following techniques:
   (a) SGD (Vanilla update).
   (b) Momentum update
   (c) Nesterov Momentum
   (d) Adagrad
   (e) RMSprop.

Hints:

- **Vanilla update**. The simplest form of update is to change the parameters along the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function). Assuming a vector of parameters x and the gradient dx, the simplest update has the form:

```
# Vanilla update
x += - learning_rate * dx
```

  where `learning_rate` is a hyperparameter - a fixed constant. When evaluated on the full dataset, and when the learning rate is low enough, this is guaranteed to make non-negative progress on the loss function.

- **Momentum update** is another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem. In particular, the loss can be interpreted as the height of a hilly terrain (and therefore also to the potential energy since $U = mgh$ and therefore $U \propto h$). Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location. The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

  Since the force on the particle is related to the gradient of potential energy (i.e. $F = -\nabla U$), the force felt by the particle is precisely the (negative) gradient of the loss function. Moreover, $F = ma$ so the (negative) gradient is in this view proportional to the acceleration of the particle. Note that this is different from the SGD update shown above, where the gradient directly integrates the position. Instead, the physics view suggests an update in which the gradient only directly influences the velocity, which in turn has an effect on the position:

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

  Here we see an introduction of a v variable that is initialized at zero, and an additional hyperparameter (mu). As an unfortunate misnomer, this variable is in optimization referred to as momentum (its typical value is about 0.9), but its physical meaning is more consistent with the coefficient of friction. Effectively, this variable damps the velocity and reduces the kinetic energy of the system, or otherwise the particle would never come to a stop at the bottom of a hill. When cross-validated, this parameter is usually set to values such as [0.5, 0.9, 0.95, 0.99]. Similar to annealing schedules for learning rates (discussed later, below), optimization can sometimes benefit a little from momentum schedules, where the momentum is increased in later stages of learning. A typical setting is to start with momentum of about 0.5 and anneal it to 0.99 or so over multiple epochs.

- **Nesterov Momentum** is a slightly different version of the momentum update that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistenly works slightly better than standard momentum.

The core idea behind Nesterov momentum is that when the current parameter vector is at some position x, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by `mu * v`. Therefore, if we are about to compute the gradient, we can treat the future approximate position `x + mu * v` as a "lookahead" – this is a point in the vicinity of where we are soon going to end up. Hence, it makes sense to compute the gradient at `x + mu * v` instead of at the "old/stale" position x.

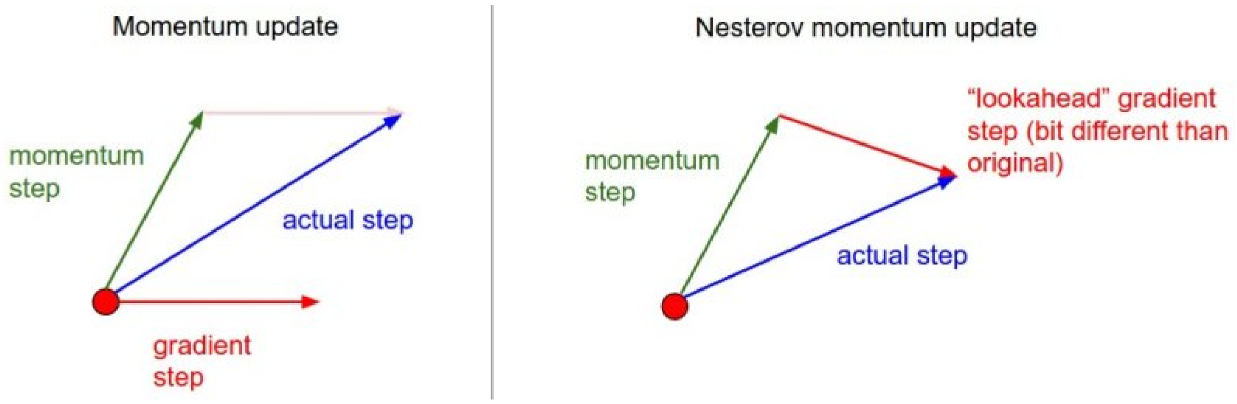

Figure 1: Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

That is, in a slightly awkward notation, we would like to do the following:

```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

However, in practice people prefer to express the update to look as similar to vanilla SGD or to the previous momentum update as possible. This is possible to achieve by manipulating the update above with a variable transform `x_ahead = x + mu * v`, and then expressing the update in terms of `x_ahead` instead of `x`. That is, the parameter vector we are actually storing is always the ahead version. The equations in terms of `x_ahead` (but renaming it back to `x`) then become:

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

All previous approaches we've discussed so far manipulated the learning rate globally and equally for all parameters. Tuning the learning rates is an expensive process, so much work has gone into devising methods that can adaptively tune the learning rates, and even do so per parameter. Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate. In this section we highlight some common adaptive methods you may encounter in practice:

- **Adagrad** is an adaptive learning rate method originally proposed by Duchi et al..

  ```
  # Assume the gradient dx and parameter vector x
  cache += dx**2
  x += - learning_rate * dx / (np.sqrt(cache) + eps)
  ```

  Notice that the variable `cache` has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. Notice that the weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. Amusingly, the square root operation turns out to be very important and without it the algorithm performs much worse. The smoothing term `eps` (usually set somewhere in range from 1e-4 to 1e-8) avoids division by zero. A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

- **RMSprop**. RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites slide 29 of Lecture 6 of Geoff Hinton's Coursera class. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:

  ```
  cache = decay_rate * cache + (1 - decay_rate) * dx**2
  x += - learning_rate * dx / (np.sqrt(cache) + eps)
  ```

  Here, `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999]. Notice that the `x+=` update is identical to Adagrad, but the `cache` variable is a "leaky". Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

- **Adam**. Adam is a recently proposed update that looks a bit like RMSProp with momentum. The (simplified) update looks as follows:

  ```
  m = beta1*m + (1-beta1)*dx
  v = beta2*v + (1-beta2)*(dx**2)
  x += - learning_rate * m / (np.sqrt(v) + eps)
  ```

  Notice that the update looks exactly as RMSProp update, except the "smooth" version of the gradient m is used instead of the raw (and perhaps noisy) gradient vector dx. Recommended values in the paper are eps = 1e-8, beta1 = 0.9, beta2 = 0.999. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative. The full Adam update also includes a bias correction mechanism, which compensates for the fact that in the first few time steps the vectors m,v are both initialized and therefore biased at zero, before they fully "warm up". With the bias correction mechanism, the update looks as follows:

```
# t is your iteration counter going from 1 to infinity
m = beta1*m + (1-beta1)*dx
mt = m / (1-beta1**t)
v = beta2*v + (1-beta2)*(dx**2)
vt = v / (1-beta2**t)
x += - learning_rate * mt / (np.sqrt(vt) + eps)
```

2. We can begin to train the network using the following initialization approaches:

   (a) **all constant initialization**,

   (b) **small random numbers**,

   (c) **calibrating the variances with 1/sqrt(n) or sqrt(2/n)**.

   Please explain the pro and cons of each initialization approach.

Hints

(a) **all constant initialization**. If every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

(b) **small random numbers**. It is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. The implementation for one weight matrix might look like `W = 0.01* np.random.randn(D,H)`, where `randn` samples from a zero mean, unit standard deviation gaussian. With this formulation, every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional gaussian, so the neurons point in random direction in the input space. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice.

    It's not necessarily the case that smaller numbers will work strictly better. For example, a Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the "gradient signal" flowing backward through a network, and could become a concern for deep networks.

(c) **calibrating the variances with 1/sqrt(n) or sqrt(2/n)**. One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs). That is, the recommended heuristic is to initialize each neuron's weight vector as: `w = np.random.randn(n) / sqrt(n)`, where `n` is the number of its inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

    The sketch of the derivation is as follows: Consider the inner product $s = \sum_{i=1}^{n} w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity.

We can examine the variance of s:

$$\begin{aligned}
\mathrm{Var}(s) &= \mathrm{Var}\Big(\sum_i^n w_i x_i\Big) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}$$

where in the first 2 steps we have used properties of variance. In third step we assumed zero mean inputs and weights, so $E[x_i] = E[w_i] = 0$.

Note that this is not generally the case: For example ReLU units will have a positive mean. In the last step we assumed that all w_i, x_i are identically distributed. From this derivation we can see that if we want s to have the same variance as all of its inputs x, then during initialization we should make sure that the variance of every weight w is $1/n$. And since $\mathrm{Var}(aX) = a^2\mathrm{Var}(X)$ for a random variable X and a scalar a, this implies that we should draw from unit gaussian and then scale it by $a = \sqrt{1/n}$, to make its variance $1/n$. This gives the initialization `w = np.random.randn(n) / sqrt(n)`.

A more recent paper on this topic, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al., derives an initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons in the network should be $2.0/n$. This gives the initialization `w = np.random.randn(n) * sqrt(2.0/n)`, and is the current recommendation for use in practice in the specific case of neural networks with ReLU neurons.