1. Optimizer

1.1. **Momentum.** Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{1}$$

$$\theta = \theta - v_t, \tag{2}$$

where the momentum term is usually set to 0.9 or a similar value.
The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

1.2. **Nesterov.** We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an *approximation* of the next position of the parameters. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{3}$$

$$\theta = \theta - v_t, \tag{4}$$

Momentum first computes the current gradient ($J(\theta)$), and then takes a big jump in the direction of the updated accumulated gradient ($\theta = \theta - v_t$). Nesterov accelerated gradient (NAG) first makes a big jump in the direction of the previous accumulated gradient ($\theta - \gamma v_{t-1}$), measures the gradient ($J(\theta - \gamma v_{t-1})$) and then makes a correction ($\theta = \theta - v_t$) which results in the complete NAG update.

1.3. **Adagrad.** Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. Adagrad uses a different learning rate for every parameter θ_i at every time step t. For brevity, we use g_t to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla J(\theta_{t,i}). \tag{5}$$

Adagrad sets learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}, \tag{6}$$

where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix. Each diagonal element in G_t is the sum of the squares of the gradients w.r.t. θ_i up to time step t. ϵ is a smoothing term that avoids division by zero (e.g., 10^{-8}).
As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing a matrix-vector product $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t, \tag{7}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

1.4. **Adadelta.** Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size ω.
Instead of inefficiently storing ω previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2, \tag{8}$$

where γ can be set as a similar value as the momentum term, around 0.9.
Based on Eq. (7), we simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$, and get

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t. \tag{9}$$

Note that the denominator is just the root mean squared (RMS) of the gradient, we can rewrite it as

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} g_t. \tag{10}$$

Note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, we define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\theta_t^2, \tag{11}$$

where

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{12}$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}. \tag{13}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t. \tag{14}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t. \tag{15}$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

1.5. **RMSprop.** RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2, \tag{16}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}\, g_t. \tag{17}$$

1.6. **Adam.** Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t ike Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{18}$$

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t^2 \tag{19}$$

m_t and v_t are are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initilizaed as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small. They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{20}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{21}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{22}$$