AMPDUP notes: ADAM and ADAMW optimizers

Jack Ramina

April 2026

1 Speaker Notes

https://arxiv.org/pdf/1711.05101 - ADAMW

https://arxiv.org/pdf/1412.6980 - ADAM

Gradient Descent: requires objective function to minimize, usually a Loss function such as cross entropy. $L[\theta]$ where θ is the set of all parameters. Calculate the gradient for each input, find the average of the gradients, then take a step in the opposite direction.

Gradient points in direction of greatest increase so stepping in opposite direction moves parameters towards a lower Loss.

$$g_t = \nabla_\theta L[\theta]$$

$$\theta_t = \theta_{t-1} - \alpha g_t$$

g_t is the gradient and α is the step size, hyperparameter that you determine.

Stochastic Gradient Descent is similar, but instead of taking the gradient of all data points, take the gradient of a small batch of randomly selected points then step. Computational time is a lot less, many small steps in an approximately correct direction instead of one step in the 100% correct direction. Slight errors can bring you out of local minima and could move towards global minima.

We all know SGD and that it is better than GD. For the sake of the talk though I will refer to them both as just GD since the underlying math is the same.

Now from GD we will derive the ADAM method. First step: introduce momentum. Consider minimizing a parabola: $y = x^2$. With GD, as we near the minimum of the parabola, the gradient starts to become very small, so each step the solver takes also becomes very small. It will take a long time to actually converge once we get close.

Momentum aims to speed up this process by getting closer to the minimum faster. Instead of forgetting the previous direction you came from, you use the prior gradients to inform your new direction:

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

$$\theta_t = \theta_{t-1} - \alpha m_t$$

β_1 is the memory term, usually around 0.9. At each step we look back and say "We've been coming this way a lot, the minimum is probably still this way" and take 90% of the previous step direction, with a slight adjustment of 10% of the current gradient.

As long as the directions are similar, this will compound and our speed towards the minimum will add up, allowing us to move faster. Eventually we'll (likely) overshoot our true minimum and climb up the other side of the parabola. At this point though the first few iterations with steep slopes have been multiplied by so many β_1 that their contribution is almost negligible, so we turn around quickly and start heading back in the right direction.

Often times we work with large numbers of parameters, and some of those parameters might reach their minimums before others. Consider the paraboloid: $z = 10x^2 + y^2$ The gradient for this function is $20x + 2y$. This is very steep in the x direction and much more shallow in the y direction. With a fixed step size, we

would expect to see x descending quickly to its basin while y slowly trudges along toward the true minimum. However, if we use a variable (adaptive) step size that changes for each parameter we can avoid this.

We want to decrease the step size for parameters that regularly see large gradients, and increase the step size for parameters for small gradients. This way we have them approach the minimum in a more balanced manner:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\theta_t = \theta_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

β_2 is the memory term for the second raw moment, usually about 0.99 or 0.999. ϵ is some small scalar to avoid dividing by zero. We take the square of the gradient to make big gradients bigger and small gradients smaller, as well as to make everything positive. We then divide our iterative step by $\sqrt{v_t}$. This decreases the step size for parameters with consistently large gradients, and increases the step size for parameters with consistently small gradients. Using a memory term close to 1 means that the gradients are cumulative, so a parameter with a large gradient at only one iteration will not have its step size decreased too much.

One small issue with our construction so far: m_0 and v_0 are initialized to 0, and since each successive m_t and v_t depend on these initializations, they tend to bias towards 0. We greatly underestimate our gradients:

$$m_1 = 0.9 \times 0 + 0.1 g_1 = 0.1 g_1$$

$$v_1 = 0.999 \times 0 + 0.001 g_1^2$$

and so on. We need to introduce some bias correction terms to negate this effect:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

so now:

$$\hat{m}_1 = \frac{0.1 g_1}{0.1} = g_1 \qquad \hat{v} = \frac{0.001 g_1^2}{0.001} = g_1^2$$

Note that as we take more steps, the effect of the bias is naturally minimized. For m_t we are repeatedly multiplying the bias by $0.9 < 1$ so it has little effect on later iterations. Therefore, we do not need to correct as much. In the correction terms we raise β to the power t. As t increases, β^t gets closer to 0 so we are essentially dividing by 1.

So putting this all together we get the ADAM algorithm:

$$g_t = \nabla_\theta L[\theta]$$

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

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

with $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $L[\theta]$ being cross entropy (or any other objective function) with parameters θ.

Now we can examine ADAMW and weight decay. Weight decay is used to subtract a small percentage of each weight's current value at every step.

$$\theta_t = (1-\lambda)\theta_{t-1} - \alpha g_t$$

This formula first decreases the weights by a small amount and then moves them in the opposite direction of the gradient as usual. This is useful because it can help to prevent overfitting. Without decay, the parameter weights can grow to encode specific cases in the training data which might not be representative of the testing data. Additionally, weights might have the tendency to get very large. This makes models very sensitive to small changes in the input. In general it is not great to have weights be too large.

For non-adaptive models you can introduce a penalty term into the loss function. This is called L_2 regularization. We will look back at GD:

$$L_{reg} = L[\theta] + \frac{\lambda'}{2}||\theta||^2$$

This is the regularized loss. The new term gets bigger as the weights get bigger, which causes the loss to increase. Bad! The gradient now becomes $\nabla_\theta L[\theta] + \lambda'\theta = g_t + \lambda'\theta$. Now the GD iterative step is:

$$\theta_t = \theta_{t-1} - \alpha(g_t + \lambda'\theta_{t-1}) \implies \theta_t = (1-\alpha\lambda')\theta_{t-1} - \alpha g_t$$

If we set $\lambda' = \frac{\lambda}{\alpha}$ then we get $\theta_t = (1-\lambda)\theta_{t-1} - \alpha g_t$ as desired. In this case, α and λ are coupled hyperparameters.

In GD and other non adaptive methods, L_2 regularization is equivalent to weight decay. The same can not be said for ADAM. If we apply the L_2 regularization to the loss and find the gradient as before, we get:

$$g_t^{reg} = g_t + \lambda'\theta_{t-1}$$

Now plug this into our equations for m and v:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)(g_t + \lambda'\theta_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t + \lambda'\theta_{t-1})^2$$

Here the decay is getting scaled by both the momentum and the adaptive scaling. The decay will no longer be proportional to θ_{t-1} as it was with GD. Instead it is getting warped by $\frac{1}{\sqrt{\hat{v}_t}+\epsilon}$. Since $\sqrt{\hat{v}_t}$ works parameter-wise, we have a different decay for every parameter. Additionally, the parameters with the largest gradients are being decayed by the smallest amounts. The main issue here is the decay is now directly tied to the adaptive scaling, whereas the only thing the decay should be dependent on is the current size of the weights. This is the case for all adaptive gradient methods.

ADAMW attempts to decouple the decay from the adaptive scaling. Instead of modifying the loss function by adding a penalty term, ADAMW simply adds the weight decay term directly into the update step:

$$\theta_t = \theta_{t-1} - \alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} - \alpha\lambda'\theta_{t-1}$$

So

$$\theta_t = (1-\alpha\lambda')\theta_{t-1} - \alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$$

Setting $\lambda' = \frac{\lambda}{\alpha}$ as before yields the desired weight decay term. We see that the decay term does not interact at all with the adaptive scaling, which solves our problems.

When I ran a test on the MNIST fashion data set I could not reproduce results that showed ADAMW working better than ADAM. This could be for a few reasons:

1. ADAMW works best for models/datasets that tend toward overfitting. ADAMW corrects the overfitting better than ADAM can. However, the MNIST data set might be too clean and not prone to overfitting.

2. The weight decay value used might be too large. I originally ran the code with a weight decay value of $\lambda = 0.001$. However after 40 epochs there was not enough of time for the weight decay to have much of an effect when it was that small. I could have ran it for more epochs but that would have taken too long, and 40 epochs is already enough time to reach a plateau. So instead I ran it with a higher decay rate of $\lambda = 0.01$. This decay might be too high. If the weights are already "well behaved" then cutting them down by too much would only hinder their ability to solve the problem correctly.

3. The paper goes into a little more detail about schedulers, a topic I skipped over for this presentation. From what I understand, the scheduler changes the learning rate/decay rate over time to produce better results. I will be looking deeper into this for future projects this semester.