Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2023

Contents

CHAPTER 7

Neural Networks 2 - Training Techniques, Regularization

7.7 Optimizing neural network parameters

We now understand both how neural networks work, and how to **train** them. We can use gradient descent to **optimize** their parameters.

But, we can do **better** than a simple SGD approach with step size $\eta(t)$. We'll try out some **modifications** that can speed up our training, and make better models.

7.7.1 Mini-batch

7.7.1.1 Review: Gradient Descent Notation

Let's review some gradient descent notation. We want to **optimize** our objective function J using W.

We do this using the gradient. This gradient depends on our current weights at time t, W_t.

$$\overbrace{\nabla_W J}^{\text{General Gradient}} \longrightarrow \overbrace{\nabla_W J(W_t)}^{\text{Gradient at time t}} \tag{7.1}$$

Our update rule is:

$$W_{new} = W_{old} - \eta \overbrace{\left(\nabla_W J(W_{old})\right)}^{\text{Gradient}} \tag{7.2}$$

Or, using timestep t:

$$W_{t+1} = W_t - \eta \overbrace{\left(\nabla_W J(W_t)\right)}^{\text{Gradient}} \tag{7.3}$$

What is our objective function J? Without regularization, it's based on our **loss** function. We can get loss for each of our data points:

We won't define J here, because it is slightly different for SGD and BGD. We'll get to that below.

$$\mathcal{L}^{(i)} = \overbrace{\mathcal{L}(g^{(i)}, y^{(i)})}^{\text{Loss for data point i}} \tag{7.4}$$

Our guess $g^{(i)}$ depends on both our current data point $x^{(i)}$, and the current weights W_t:

$$\mathcal{L}^{(i)}(W_t) = \mathcal{L}(\overbrace{h(x^{(i)}; W_t)}^{g^{(i)}}, y^{(i)}) \tag{7.5}$$

7.7.1.2 Review: BGD vs. SGD

Let's review our two main types of gradient descent, using the equation

$$W_{t+1} = W_t - \eta \overbrace{\left(\nabla_W J(W_t)\right)}^{\text{Gradient}} \tag{7.6}$$

First, we have **batch gradient descent**, where we use our **whole** training set each time we take a step.

Definition 1

Batch Gradient Descent (BGD) is a form of gradient descent where we get the **gradient** of our loss function using **all of our training data**.

$$\nabla_W J(W_t) = \sum_{i=1}^{n} \overbrace{\nabla_W \left(\mathcal{L}^{(i)}(W_t) \right)}^{\text{Each data point}}$$

We get the gradient for each data point, and then **add** all of those gradients up. We use this **combined gradient** to take **one step**.

We **repeat** this process every time we want to take a new step.

Then, we have **stochastic gradient descent**, where we use only **one** data point for each step we take.

Definition 2

Stochastic Gradient Descent (SGD) is a form of gradient descent where we get the **gradient** of our loss function using **one data point at a time**.

$$\nabla_W J(W_t) = \overbrace{\nabla_W \left(\mathcal{L}^{(i)}(W_t) \right)}^{\text{One data point}}$$

We **randomly** choose one data point $(x^{(i)}, y^{(i)})$ and get the **gradient**. Based on this one gradient, we take our **step**.

For each step, we choose a new **random** data point.

These two approaches have tradeoffs:

Concept 3

There are **tradeoffs** between **SGD** and **BGD**:

- Each step is **faster** in **SGD**: we only use one data point.
  - Meanwhile, **BGD** is **slower**: each step uses all of our data.
  - **SGD** could improve a lot with only a **small subset** of a data.
- Because **BGD** uses all our data, its gradient is much more **accurate**.
  - **SGD** often uses **smaller** steps: the gradient is less accurate, with less data.
  - This is worse if the data is **noisy**: each SGD step becomes less effective.
- **SGD randomly** chooses data points: this random noise makes it harder to overfit.
  - **BGD** uses all of the data, so we don't reduce overfitting.

7.7.1.3 Mini-batch

Rather than picking one or the other, one might think, "why do we have to pick **every** data point or **one** data point? Couldn't we pick only a **few**?"

This is the premise of **mini-batch**: instead of making a batch out of the entire training set, we **randomly** select a few data points, and use that as our batch.

Definition 4

Mini-batch is a way to **compromise** between SGD and BGD.

To create a mini-batch, we **randomly** select K data points from our training data.

We treat this mini-batch the same way we would a regular **batch**: get the **gradient** of each data point, **add** those gradients, and take one step of gradient descent.

$$\nabla_W J(W_t) = \overbrace{\sum_{i=1}^{K}}^{\text{K data points in a mini-batch}} \nabla_W \left(\mathcal{L}^{(i)}(W_t) \right)$$

We gather a **new** mini-batch for each step we want to take.

Mini-batch is the **default** used in most modern packages: it gives us more **control** over our

algorithm, and can often find the **best** of both worlds.

We do have to be careful to randomly select data in an efficient way, though. Packages usually take care of this.

Concept 5

Mini-batch has a lot of benefits of both SGD and BGD:

- Steps are **faster** than BGD: we only need to get the gradient for K points.
  - The **speed** no longer depends on the total training data size (more data, more gradients): instead, it depends on our **batch size** K.
- Steps are more **accurate** than SGD: with more data, we have a better **gradient**.
  - This means we can take **bigger** steps.
- Our batches are **random**, like SGD: we reduce overfitting and escape local minima.

One more important benefit:

- If we find that a particular problem is better suited for something closer to BGD or SGD, we can **adjust** our batch size K.
  - This gives us more **control** over our learning algorithm.

7.7.2 Adaptive Step Size - Challenges

We'll stop discussing mini-batches, and the SGD vs. BGD problem. Instead, let's improve our **step size**.

Step size η is a difficult problem:

- If η is **small**, then our training can take a long **time**.
- If η is too **large**, we might **diverge**: our answer gets way too large.
- A **large** step size might also cause **oscillation**: most of our step is wasted going back and forth, so we go **slowly** again.

SGD and mini-batch have a step size-related problem, too:

- In order to **converge** according to our theorems (see chapter 3), the step size $\eta(t)$ has to be **decreasing** in a certain way.

Check chapter 3 for the exact requirements of the theorem.

We'll spend the following sections coming up with solutions.

7.7.3 Vanishing/Exploding Gradient

Now, neural networks have one more **problem**, that we've ignored so far: **deep** neural networks can cause a problem called "**exploding/vanishing gradient**".

By "deep", we just mean "many layers".

Here's an example: suppose you have a long chain rule, with 8 terms. Our chain rule gets **longer** with more layers, because each layer needs its own derivatives.

$$\frac{\partial A}{\partial H} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \cdot \left(\cdots \right) \cdot \frac{\partial G}{\partial H} \tag{7.7}$$

This chain rule gets **longer** as we move "**backwards**" through our network, so the chain rule is longest for the "**early**" layers: $\ell = 1, 2$, and so on.

Suppose all of our derivatives are roughly .1. What happens when we multiply them **together**?

$$\frac{\partial A}{\partial H} = .1 \cdot .1 \cdot \left(\cdots \right) \cdot .1 = 10^{-8} \tag{7.8}$$

The derivative becomes really, really **tiny**! This is the case of the **vanishing** gradient: if our gradients are less than one, then as we append more layers, they multiply to get smaller and smaller.

- This is a problem: if our gradients in our earlier layers become too **small**, we'll never make any progress! They'll hardly change.

Definition 6

Vanishing gradient occurs when a deep neural network ends up with **very small gradients** in the **earlier** layers.

This happens because a deeper neural network has a **longer chain rule**: if all of the terms are **less than one**, they'll multiply into a very small value, "**vanishing**".

This means that our gradient descent will have **almost no effect** on these earlier weights, **slowing down** our algorithm considerably.

What if the gradients are larger than 1? Let's say our derivatives are 10 each.

$$\frac{\partial A}{\partial H} = 10 \cdot 10 \cdot \left(\cdots \right) \cdot 10 = 10^{8} \tag{7.9}$$

Now, the early derivatives are becoming **huge**! This is the case of **exploding** gradient: if our gradients are greater than one, then as we add layers, they multiply to get bigger.

- This is also a problem: we don't want to take **huge** steps, or we will **diverge**, or **oscillate**, and jump huge distances across the **hypothesis space**.

Definition 7

Exploding gradient occur when a deep neural network ends up with **very large gradients** in the **earlier** layers.

This happens because a deeper neural network has a **longer chain rule**: if all of the terms are much **greater than one**, they'll multiply into a very large value, "**exploding**".

This means that our gradient descent will take **huge steps** in the hypothesis space. This can cause us to **diverge**, miss local minima, or **oscillate**.

So, to avoid this, we can't just blindly multiply our gradients and keep a fixed step size.

The solution? Each **weight** gets its own step size η.

Concept 8

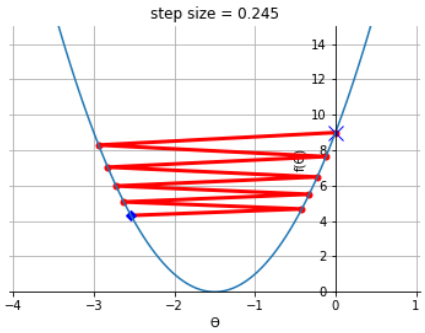
In order to avoid **vanishing/exploding** gradient problems, we give each **weight** in our network its own **step size** η.

This allows us to **adjust** the step size for some weights more than others: if our gradient is too large or small, we can fix it.

7.7.4 Momentum

7.7.4.1 Solving Oscillation

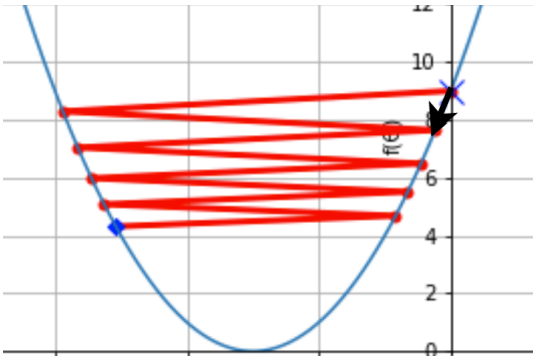
Let's look at one common problem we have with gradient descent: **oscillation**.



We overshoot our target, and then have to take another step that **undoes** most of what happened in the previous step. So, we waste a lot of time correcting the last step.

For example, our first two steps land us in almost the same place we started!

This can significantly **slow** down how quickly we converge.



The black arrow shows the combined effect of our first two steps: almost nothing!

We don't want to waste time, so we want to remove the "part" of the gradient that is likely to **cancel** out.

The **next** gradient cancels out some of the previous. Our first two steps add up, or "**average** out" to a small improvement.

If our steps are effectively "averaging", we'll speed up that process: we'll average together the gradients *before* taking our step!

This means we can take a bigger step in a direction we won't have to cancel!

Concept 9

Since our gradient descent steps **combine** to give us our new model, we can think of them as adding, or "**averaging**" to a more accurate improvement.

When our function **oscillates**, we get the same pattern **multiple** times: past steps indicate the sort of pattern we'll see in **future** steps.

The only real difference between adding and averaging is whether we divide by the number of terms.

So, we'll average our current gradient with past gradients: that way, the **component** that gets cancelled out is **removed**, and we won't have to undo our mistake over and over.

Concept 10

Oscillation causes us to move back and forth over the same region **multiple** times, where each step mostly **cancels** out the last.

One solution is to **average out** multiple of our gradients: the part that is "**cancelled** out" should be eliminated by the average.

So, we average our **past** gradients (past oscillation) with our **current** gradient, so we move in a more **efficient** direction, speeding up our algorithm.

Another way to think about it: when we **average** out our current and previous gradient, we're cancelling out what they "**disagree**" on, and keeping what they **agree** on.

So, we're taking a step in the direction that multiple gradients agree will **improve** our model!

7.7.4.2 Weighted averages

We could naively average all of our gradients **equally**. But, this would be a bad idea:

- It doesn't give you as much control of the algorithm: what if we care more about the **present** gradient, than the previous one?
- Gradients further in the **past** are **less likely** to matter: we've moved further away from those positions.
  - We also need to **scale** down past terms, so they don't take up most of the average.

If you're averaging 100 terms, and you add one more... it's not going to change much.

The first problem is easy to solve: we'll **weigh** each of our terms differently.

Concept 11

A **weighted average** is used when we want some terms to affect our **average** more than others.

We represent this with **weights**: each weight represents the **proportion** of our average from that term.

$$\text{Weighted Average} = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

These weights are **separate** from the weights inside our neural network.

They do, however, represent the same type of concept: the NN weights scale the **input**, while these weights scale the **gradients**.

Example: If $w_1 = .6$, that means 60% of the average comes from x_1.

Note that, since we're talking about **proportions**, they need to **sum to 1**: it wouldn't make sense to have more than 100% of the average.

At each time step, we're adding one new gradient: the **present** one.

We'll simplify our average to those two terms: the **present** gradient, versus all the **past** gradients.

- We represent the importance (**weight**) of our **past** gradients using the variable γ.
- We want the two terms to add to 1: so, the importance of **current** gradient is $1-\gamma$.

$$\overbrace{A_t}^{\text{Average}} = \overbrace{\gamma G_{t-1}}^{\text{Old gradients}} + \overbrace{(1-\gamma) g_t}^{\text{New gradient}} \tag{7.10}$$

Now, we have **control** over how much the present or past gradient matters: we just have to adjust γ.

7.7.4.3 Running Average

We still have some work to do: first, we haven't made it clear how we're incorporating our old gradients: we lumped them into one term.

Let's try building up from $t = 1$. We'll assume our previous gradients are 0, for simplicity.

$$A_0 = g_0 = 0 \tag{7.11}$$

Our first step will average this with our **first** gradient:

$$A_1 = \gamma g_0 \quad + \quad (1-\gamma) g_1 \tag{7.12}$$

Simplifying to:

$$A_1 = (1-\gamma)g_1 \tag{7.13}$$

What about our second step?

$$A_2 = \overbrace{\gamma G_{t-1}}^{\text{Old gradients}} + (1-\gamma)g_2 \tag{7.14}$$

We *could* just plug in g_1. But, A_1 contains the information about our first gradient g_1, **and** the gradient before it, g_0.

$$A_2 = \overbrace{\gamma A_1}^{\text{Contains } g_1,\, g_0} + (1-\gamma)g_2 \tag{7.15}$$

We can repeat this process:

$$A_3 = \overbrace{\gamma A_2}^{\text{Contains } g_2, g_1,\, g_0} + \overbrace{(1-\gamma)g_3}^{\text{New gradient}} \tag{7.16}$$

And so, we've created a general way to **average** as our program **runs** through different gradients.

$$A_t = \gamma A_{t-1} + (1-\gamma)g_t \tag{7.17}$$

To allow more flexibility, we'll allow γ to **vary in time**, as γ_t.

$$A_t = \gamma_t A_{t-1} + (1-\gamma_t)a_t \tag{7.18}$$

- We call this a **running average**.

Definition 12

A **running average** is a way to average past data with present data **smoothly**.

Our **initial** value for the average is typically zero:

$$A_0 = 0$$

Then, we begin introducing **new** data points.

- You use the parameter γ_t to indicate how much you want to prioritize **past data**.
- Thus, $1 - \gamma_t$ indicates the value of **new data**.

$$\overbrace{A_t}^{\text{Average}} = \overbrace{\gamma_t A_{t-1}}^{\text{Old gradients}} + \overbrace{(1-\gamma_t) g_t}^{\text{New gradient}}$$

- Note that instead of γ, we write γ_t: this "discount factor" can vary with time.

Clarification 13

This is technically only one kind of **running average**: here, we use an "**exponential moving average**".

There are different ways to average past data points, with different **weighting** schemes.

- For example, you could do a "**simple moving average**", where you average equally over the last n data points.

7.7.4.4 Running Averages: The Distant Past

So, how does this "running average" approach affect our different data points, further in the past? Let's find out.

For simplicity, let's assume $\gamma_t = \gamma$: it's a **constant**.

$$A_t = \gamma A_{t-1} + (1-\gamma) g_t \tag{7.19}$$

We can expand A_{t-1} to see further in the past:

$$A_t = \gamma\Big(\gamma A_{t-2} + (1-\gamma) g_{t-1}\Big) + (1-\gamma) g_t \tag{7.20}$$

And even further:

This is starting to get messy: don't worry if it's hard to read.

$$A_t = \gamma\left(\gamma\left(\gamma A_{t-3} + (1-\gamma)g_{t-2}\right) + (1-\gamma)g_{t-1}\right) + (1-\gamma)g_t \tag{7.21}$$

Let's rewrite this.

$$\gamma^3 A_{t-3} \;+\; \gamma^2(1-\gamma)g_{t-2} \;+\; \gamma(1-\gamma)g_{t-1} \;+\; (1-\gamma)g_t \tag{7.22}$$

We see a "stacking" effect for γ:

- We only partly include our newest data point: we scale it by $1-\gamma$, to make room for the past.

$$(1-\gamma)g_t \tag{7.23}$$

- But if your gradient is 1 time unit in the past, we apply γ **once**, "forgetting" some more of that gradient.

$$\gamma(1-\gamma)g_{t-1} \tag{7.24}$$

- But if your gradient is 2 units in the past, we apply γ **twice**: we've "forgotten" some of it twice.

$$\gamma^2(1-\gamma)g_{t-2} \tag{7.25}$$

Each time we do an average, we scale down our older data points by γ. So, the further in the past, the less effect they have.

- This is exactly the kind of design we wanted!

Concept 14

A **running average** tends to pay less attention to data further in the **past**.

- In general, if you are k time units in the past, we apply a factor of γ^k.

Because $\gamma < 1$, this **exponentially** decays to 0.

If we want to fully expand A_t, it's easiest to use a sum:

$$A_T = \sum_{t=0}^{T} \gamma^{(T-t)} \cdot (1-\gamma) \cdot g_t$$

You can compare this formulation against what we computed above.

7.7.4.5 Momentum

Applying a running average to our gradients gives us **momentum**.

- This analogy to **physics** represents how our point "moves" through the **weight space**, to optimize J.
- The gradient gives us a "direction" of motion. So, our **momentum** represents the direction we were "already moving": the **previous** averaged gradient.

Gradient descent adjusts our weights: we go from one list of weights, to another. That's why we say we're moving through the "weight space".

Because our hypothesis is determined by our weights, this is also a "hypothesis space".

We use M to represent the "averaged gradient" that we use to move. Our initial momentum is 0:

$$M_0 = 0 \tag{7.26}$$

And we want to average our current gradient with past gradients:

$$M_t = \gamma M_{t-1} + (1-\gamma) g_t \tag{7.27}$$

What is our **past** gradient g_t? Well, we want to use W to modify J: $\frac{\partial J}{\partial W}$, or $\nabla_W J$.

And we're moving through the **weight space**, so our input to the gradient is the previous set of weights, W_{t-1}.

$$g_t = \nabla_W J(W_{t-1}) \tag{7.28}$$

And finally, M_t, our "averaged gradient", determines how we move.

$$W_t = W_{t-1} - \eta M_t \tag{7.29}$$

Definition 15

Momentum is a technique for gradient descent where we do a **running average** between our current gradient, and our older gradients.

This approach reduces **oscillation**, and thus aims to improve the speed of convergence for our models.

- Our initial "momentum" (averaged gradient) is **0**.
- The amount we value new data is given by γ_t.
- Old data is scaled by $1-\gamma_t$.

$$M_0 = 0$$

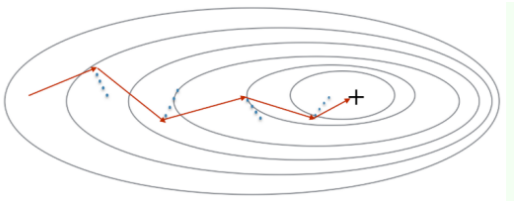
$$M_t = \overbrace{\gamma_t M_{t-1}}^{\text{Old gradients}} + \overbrace{(1-\gamma_t)\nabla_W J(W_{t-1})}^{\text{New gradient}}$$

We use this momentum to take our step:

$$W_t = W_{t-1} - \eta M_t$$

This approach puts more emphasis on newer data points, and less on older ones.

Example: We can see this "dampened oscillation" in action below:



The blue lines indicate the more "severe" path that we would've taken with normal gradient descent. the orange line is the line we take with momentum.

Notice that generally, the orange path is closer to correct! We still oscillate somewhat, but much less than we would have otherwise.

7.7.5 Adadelta

7.7.5.1 Per-weight step sizes

Earlier, we discussed creating **separate** step sizes for different weights in our network:

- Different weights could have different **sensitivities**: one weight could have a much larger/smaller impact on J, based on **structure**.
- We already have the challenge of figuring out what **step sizes** cause **slow convergence/divergence**: it would be even harder to find a step size that fit **all** of our weights, at the same time.

So, instead, we decided that each weight gets its **own step size**.

- But, we never elaborated on **how** we compute that (adjustable) step size.

Notation 16

Our base step size is indicated as η.

We'll call the **modified** step size for weight j, η_j.

7.7.5.2 Scaling

Our goal now, is to come up with a **systematic** way to **assign step sizes**. This allows us to **adjust**, rather than run our whole gradient descent with a "bad" step size.

So, we have less of a problem if we choose η poorly.

Why do we adjust our step size? To avoid slow convergence, oscillation, or divergence.

- We might expect **slow convergence** if the derivative is too **small**: we carefully take small steps, but we aren't having much of an impact on J.
  - Across this "flatter" region of the surface, in one direction, we might expect it to be safer to move further.
- We might expect **divergence** or **oscillation** if the derivative is too **large**.
  - We might end up "missing" possible solutions/local minima by **overshooting** them.
  - So, "steeper" regions might be riskier.

Concept 17

Our goal is to **scale** our step size, so that it **adapts** to the situation:

- We want to **shrink** our step for **large** gradients
- We want to **increase** our step for **small** gradients

And we're interested in the **magnitude** of these gradients.

This sort of behavior is easily captured by including a factor of $1/\|g_t\|$. However, this has a smoothness problem: so, we'll use $1/\|g_t\|^2$ instead.

Though, we need to keep it separate for each of our data points.

Notation 18

The gradient for **weight** j at **time** t is given as

$$g_{t,j} = \nabla_W J(W_{t-1})_j$$

Note the double-subscript.

- By isolating weight j, we have a constant, not a vector.

We can now write our gradient update rule:

$$W_{t,j} = W_{t-1,j} - \eta_j \cdot g_{t,j} \tag{7.30}$$

And we're currently using the step size

$$\eta_j = \frac{\eta}{g_{t,j}^2} \tag{7.31}$$

Don't save this equation! It isn't our final formula.

7.7.5.3 Averaging

But, we don't necessarily know how "steep" our region **generally** is, based on the current gradient g_t. g_t only gives us **one point** in space.

It would be helpful to include information from the **past**: we'll be re-using the **weighted average**, once again.

$$G_t = \gamma G_{t-1} + (1-\gamma) g_t^2 \qquad \text{(Maybe?)}$$

Once again, it's helpful that the weighted average gradually "forgets" older information: we care less about gradients which are "further" from the present.

- Note that we're averaging the **squared** magnitude.

Technically this is still **incorrect**: we need the j notation.

It's incorrect because g_t is a vector: we can't square it directly, we have to square its magnitude.

$$G_{t,j} = \gamma G_{t-1,j} + (1-\gamma) g_{t,j}^2 \qquad \text{(Fixed!)}$$

7.7.5.4 Division by zero

There's a problem with our weight adjustment:

$$\eta_j = \frac{\eta}{G_{t,j}} \tag{7.32}$$

What happens if the denominator is near zero? It'll explode to a huge number! And at zero, it's undefined.

To solve this, we'll add a very small constant, ϵ.

We won't prescribe any particular choice of ϵ here.

$$\eta_j = \frac{\eta}{G_{t,j} + \epsilon} \tag{7.33}$$

Now, our scaling factor will never be bigger than $1/\epsilon$.

7.7.5.5 Square root

One last concern, to wrap up: currently, we're diving by the **squared** gradient. This is actually somewhat overkill.

Remember that our goal is to do the following operation:

$$W_{t,j} = W_{t-1,j} - \underbrace{\eta_j \cdot g_{t,j}}_{\text{Update}} \tag{7.34}$$

With our current formula, our update has

- a factor of $g_{t,j}$ in the **numerator**
- from η_j, a factor proportional to $g_{t,j}^2$ in the **denominator**

This is "scaling" our gradient by more than we want to. So, we'll take the **square root** of the denominator.

Our goal is to make the scales of different axes more similar, not to neglect dimensions with high gradient (high effect on loss)

$$\eta_j = \frac{\eta}{\sqrt{G_{t,j} + \epsilon}} \tag{7.35}$$

This is the completed form of our **adadelta step size rule!**

Definition 19

Adadelta is a technique for **adaptive step size**, which:

- **Decreases** step size in dimensions with a history of **high-magnitude** gradients
- **Increases** step size in dimensions with a history of **low-magnitude** gradients

Suppose our gradient for weight W_j at time t is represented by

$$g_{t,j} = \nabla_W J(W_{t-1})_j$$

This is accomplished by **scaling** the step size η to create η_j:

$$\eta_j = \frac{\eta}{\sqrt{G_{t,j} + \epsilon}}$$

Where $G_{t,j}$ is a "**running average**" of the previous gradients for weight W_j.

$$G_{0,j} = 0$$
$$G_{t,j} = \gamma G_{t-1,j} + (1-\gamma) g_{t,j}^2$$

So, our completed gradient descent rule takes the form:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j} + \epsilon}} \cdot g_{t,j}$$

7.7.5.6 Sparse Data

One major advantage of adadelta is its use for **sparse datasets**, where many variables only show up in a small percentage of the data.

- If a variable is much less frequent, then the weighted average G_t will be much smaller.
- So, when those data points **do** appear, the step size is much larger.

This allows our model to learn more from variables that don't show up as frequently.

Concept 20

Adadelta often works well with **sparse data**: datasets where many variables rarely show up as non-zero.

The step sizes for these variables become much larger, so our model can learn more from less-common information.

However, this can run the risk of paying attention to variables that are sparse, but **not** especially meaningful.

- It's important to choose your variables carefully, so the model doesn't "learn" from noise.

7.7.5.7 Adagrad

Originally, adadelta derives from a **simpler** method, known as **adagrad**, or "adaptive gradient".

- The main difference with this approach is, rather than find the **weighted average** G_t, we simply **sum** the previous gradients.

The main problem with this approach is that, over time, G_t becomes too **large**. So, our step size becomes too small, and our algorithm slows down.

By averaging, we don't run into this problem of G_t "accumulating": older data is gradually **forgotten**.

7.7.6 Adam

Momentum and Adadelta both bring some benefits:

- **Momentum** averages our current gradient with **previous gradients**, to reduce **oscillation** and make a more direct path to the solution.
- **Adadelta** modifies our **step sizes**: it takes smaller steps in directions of high gradient (reduce overshooting) and takes bigger steps in directions of low gradient (converge faster).

There's nothing structurally incompatible between them. So, why not incorporate both?

- This combination is called **Adam**: it has become the most popular way to handle step sizes in neural networks.

Concept 21
Adam integrates the techniques of both **momentum** and **adadelta**.

7.7.6.1 Momentum and Adadelta

We used two different **running averages**, both using γ for their "**discount factor**": how much we discount the effect of older data.

It's "discounting", or reducing the effect of older data, because $\gamma < 1$.

- We'll replace γ with B_1 (momentum) and B_2 (adadelta).

Notation 22
In the adam algorithm, B_1 and B_2 are **discount factors** replacing γ from momentum and adadelta.

We use the same notation for gradients:

$$g_{t,j} = \nabla_W J(W_{t-1})_j \tag{7.36}$$

First, we'll keep track of our **averaged gradient**, $m_{t,j}$. This will be the **direction** we plan to move.

$$m_{t,j} = B_1 m_{t-1,j} + (1 - B_1) g_{t,j} \tag{7.37}$$

Then, we'll keep track of the **average squared gradient**, $v_{t,j}$. This will tell us whether to scale up/down our **step size** on each variable.

$$v_{t,j} = B_2 v_{t-1,j} + (1 - B_2) g_{t,j}^2 \tag{7.38}$$

We can combine these into our equation, the way you use momentum and adadelta:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{v_{t,j} + \epsilon}} \cdot m_{t,j} \tag{7.39}$$

We're not quite done yet, though.

7.7.6.2 Normalization

There's one issue with our running average, that we should address.

Suppose we have a sequence of numbers:

$$[1, 1, 1] \tag{7.40}$$

What's the running average of this sequence of numbers, with $\gamma = .1$? You'd expect it to be 1 for all elements, right?

- We start with $a_0 = 0$.

$$\begin{aligned} a_1 &= .1 * 0 + .9 * 1 = .9 \\ a_2 &= .1 * .9 + .9 * 1 = .99 \\ a_3 &= .1 * .99 + .9 * 1 = .999 \end{aligned} \tag{7.41}$$

It turns out not to be true! Why is that?

Because of our initial term, a_0: it has nothing to do with the data. Because it'll always be 0, it **deflates** the value of all the remaining averages.

How much does it deflate the average? Well, let's consider the general equation:

$$A_T = \sum_{t=0}^{T} \gamma^{(T-t)} (1 - \gamma)^t g_t \tag{7.42}$$

What "fraction" of the total is missing, thanks to $A_0 = 0$?

- Well, all of our weighted terms $\gamma^{(T-t)}(1-\gamma)^t$ should add up to 1: each one represents the "percent/fraction" of the average which comes from that g_t term.

A_0 matches $t = 0$, so we find:

$$A_T = \overbrace{A_0 \gamma^T}^{A_0 = 0} + \sum_{t=1}^{T} \gamma^{(T-t)} (1 - \gamma)^t g_t \tag{7.43}$$

So, out of our total 1, or 100%, we're missing γ^T.

- Our new total is $1 - \gamma^T$.
- In order to correct for the "zeroed out" part of our average, we multiply by

$$\frac{\text{Desired total}}{\text{Real total}} = \frac{1}{1-\gamma^T} \tag{7.44}$$

Concept 23

We've chosen $A_0 = 0$: this is **unrelated** to our real data.

As a result, A_t doesn't accurately reflect our weighted average: it's slightly **smaller**.

- A_0 is "included" as a 0, even though it's not a **real** data point.
- So, whatever **percent** of our data is represented by A_0, is "falsely empty".

A_0 is scaled by γ^t, so that's the amount **removed** from our "true" weighted average.

The "real" weighted average, that ignores A_0, has to un-do the deflation caused by including fake, starting data:

$$\widehat{A}_t = A_t \cdot \frac{1}{1-\gamma^t}$$

We'll make this correction for our running averages:

$$\begin{aligned} \widehat{m}_{t,j} &= \frac{m_{t,j}}{1-B_1^t} \\ \widehat{v}_{t,j} &= \frac{v_{t,j}}{1-B_2^t} \end{aligned} \tag{7.45}$$

7.7.6.3 Adam

Now, we have our complete equation for adam:

Definition 24

Adam is a technique for improving **gradient descent** that integrates two other techniques. Our computed gradient is given as

$$g_{t,j} = \nabla_W J(W_{t-1})_j$$

- **Momentum** averages our previous gradients with our current one, to reduce oscillation and get a "averaged" view of data. B_1 gives the weight of old data.

$$m_{0,j} = 0 \qquad m_{t,j} = B_1 m_{t-1,j} + (1 - B_1) g_{t,j}$$

- **Adadelta** estimates how "steep" our gradient has been recently, and uses it to adjust our step size, for better convergence. B_2 gives the weight of old data.

$$v_{0,j} = 0 \qquad v_{t,j} = B_2 m_{t-1,j} + (1 - B_2) g_{t,j}^2$$

We include a **correction** factor for the zeroed initial conditions: $m_0 = v_0 = 0$.

$$\widehat{m}_{t,j} = \frac{m_{t,j}}{1 - B_1^t} \qquad \widehat{v}_{t,j} = \frac{v_{t,j}}{1 - B_2^t}$$

Finally, our update rule takes the form:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{\widehat{v}_{t,j} + \epsilon}} \cdot \widehat{m}_{t,j}$$

Impressively, adam is not very sensitive to the initial conditions for ϵ, B_1 and B_2.

It's able to "self-correct" its step sizes and gradient based on data.

To implement this for NNs, we just need to keep track of each value, for each weight. If you do it systematically, this is easier than it sounds.

7.8 Regularization

Something we haven't discussed in a while, that we might investigate, is **regularization**: techniques against overfitting.

- We teach our model using "training data", which, by chance, may not perfectly reflect the **true** distribution.

We want to apply this to our modern, deep neural networks, with their huge number of **parameters**, and huge amount of **data**. And yet...

These modern neural nets **don't** tend to have as much problem with **overfitting**, and we're not sure **why**!

Regardless, we do still have *some* methods for regularization: this will be our focus for the rest of this chapter.

7.8.1 Methods related to ridge regression

We'll start with methods that we can bring back from classic ridge regression.

7.8.1.1 Early Stopping

One component built into our learning algorithm for regression is **early stopping**: we check if the model is still making **progress**. If it isn't, then we **stop** training.

- The longer we train our model, the more time it has to "**memorize**" the exact structure of our training data: including **noise**.

We would typically either measure the size of the **gradient**, or the change in the **loss**. If either was small, then we might be in a local minimum: we've finished training.

So, we do the same here: after a period of training (over the whole dataset, called an **epoch**), we measure the **loss** on a validation set.

- If the loss stops decreasing, or begins **increasing**, our model is probably **overfitting**. We **stop early**.

Then, you return the weights with the lowest error.

Definition 25

An **epoch** is the time frame during which your model sees your whole **training data** set, **once**.

With **early stopping**, you evaluate your model using your **validation data**, computing the loss.

- If the loss has decreased from the last epoch, you **continue** training.
- If the loss has stopped decreasing, or is increasing, you **stop** training.

This continues until you've either run out of **epochs**, or you've met your **termination** condition, and stopped early.

- Note that sometimes, "epoch" just refers to how long you train before you check your loss.
- In this case, it might be smaller than the whole dataset.

7.8.1.2 Weight Decay

We can also use the same kind of regularization term that we used for linear regression: penalizing the **squared magnitude**.

- Starting with our loss function:

$$J_{loss} = \sum_{i=1}^{n} \mathcal{L}(NN(x^{(i)}),\ y^{(i)};\ W) \tag{7.46}$$

- And we penalize based on the square magnitude of our weights:

$$J = \lambda \|W\|^2 + J_{loss} \tag{7.47}$$

If we take the gradient, we get:

$$\nabla_W J = 2\lambda W + \nabla_W J_{loss} \tag{7.48}$$

Let's see how the regularization affects our step:

$$W_t = W_{t-1} - \eta\left(2\lambda \|W_{t-1}\| + \nabla_W J_{loss}\right) \tag{7.49}$$

It directly subtracts from our weight, **decaying** it.

$$W_t = W_{t-1}\Big(1 - 2\lambda\eta\Big) - \eta\nabla_W J_{loss} \tag{7.50}$$

Thus, we call it **weight decay**.

Concept 26

When we apply **square magnitude** regularization to the weights of a neural network, we call it **weight decay**.

$$J_{reg} = J_{loss} + \lambda\|W\|^2$$

That's because, when you take the gradient, it directly subtracts from weight W, causing it to **decay** by a factor of $(1 - 2\lambda\eta)$.

$$W_t = W_{t-1}\Big(1 - 2\lambda\eta\Big) - \eta\nabla_W J_{loss}$$

7.8.1.3 Perturbation

One last way to reduce overfitting is to add some **random noise** to our data: each variable has a small, random number added to it.

This value is typically **zero-mean** and **normally distributed**:

- Zero-mean: it has 0 effect, on average, so it doesn't bias the data high or low.
- Normally distributed: the noise is **symmetric**: +2 and -2 are equally likely.

The "normal distribution" contains more information than that, but the symmetry is important.

How large the noise is, depends on the chosen **variance**.

This reduces overfitting, because if the data is slightly different each time you see it, it's harder to perfectly "memorize" the exact shape and structure.

Definition 27

Perturbation is a technique where you slightly modify your system.

- In our case, we're **randomly** adding small amount of **noise** to our input data.

This makes it more difficult for the model to **overfit**, because the patterns aren't always exactly the same.

- Only the "general", high-level patterns are preserved, each time you view the dataset.

Of course, if you perturb your data too strongly, you can miss real patterns. Your perturbations shouldn't be too large.

7.8.2 Dropout

We also have **structural** ways of dealing with overfitting. We discussed perturbing the **dataset**, but we could, instead, perturb the **model** itself!

We do this by randomly **removing** some weights from the neural network, and training.

- Each weight has a probability p of being "turned off": the **activation** is set to zero.

$$a_j^\ell = 0 \tag{7.51}$$

- Thus, that neuron's output is **ignored** by the next layer, and receives no training.
- At the next step, we remove a **different** random selection of weights.

Because our model keeps changing slightly, it's harder to exactly **overfit** to the data.

This particular approach also addresses a **different** kind of overfitting:

- One model might heavily "rely" on a **small** number of neurons to make decisions.
- This makes our model less flexible, uses the weights less efficiently.

To solve this, we prevent the neural network from using some of these weights, **randomly**.

Thus, the whole network "**shares**" some responsibility for getting the right answer.

Definition 28

Dropout is a process where, at each training interval, you **randomly** "drop out", or de-activate, some of the weights in the network.

- Each neuron has probability p of being turned off.
- These neurons have their **activation** set to zero: $a_j^\ell = 0$.

This process is designed to reduce **overfitting**. As the network randomly changes, it's harder for it to perfectly match the data structure.

- When the network is finished training, we multiply all the weights by p. Why?
- Because during training, only p fraction of the neurons were active. We want to replicate that average activity level, even when we use all of our neurons.

This process is also designed to create "collective responsibility" for your neurons. It prevents your network from relying on a small number of neurons to solve problems.

It generally improves **robustness** against random variations in the data.

This approach has, in recent years, become somewhat less popular, for a couple reasons:

- Very **large** networks don't struggle as much with overfitting.
- CNNs tend not to benefit from this procedure, because of **weight-sharing**: the same weights are re-used in multiple places.
- Like most ML techniques, their usefulness depends on the situation.

We'll discuss CNNs in our next chapter.

It still finds use in some smaller models, RNNs, etc.

In many places, it has been replaced by **batch normalization**.

7.8.3 Batch Normalization

7.8.3.1 Covariate Shift

Our last approach related to regularization was designed to handle a new type of problem we call **covariate shift**:

When you run **gradient descent** on a neural network, you're adjusting the weights of all of our layers, at the **same time**.

Let's focus on layer 1 and layer 2. By updating layer 1, we've changed the outputs it creates: the same $x^{(i)}$ now creates a different output, going from a^1_{old} to a^1_{new}.

But, this output is the **input** of layer 2.

- This means that layer 2 is now receiving **different inputs** than it was before.
- This is a problem: layer 2 just received training based on the **old** inputs a^1_{old}!

This makes life a lot harder for our layer 2: not only is it learning to make better predictions, it also has to adjust for the change in layer 1. This is a form of **covariate shift**.

Definition 29

Covariate shift occurs when the distribution of input variables **changes** over time.

- This can cause our original model to become inaccurate, or "outdated": it was trained on **different** data.

Internal covariate shift occurs because of changes to the network itself.

- The distribution of inputs to **later layers** changes, because **earlier layers** have changed through training.

Example: If the weights in layer 1 all get smaller (as a side effect of correcting them), layer 2 may have to make all of its weights bigger to compensate.

- Our expectation is that this extra work would slow down training.

7.8.3.2 Layer Normalization

So, we want to counter the problem of how the input to layer 2 **changes** based on layer 1's **learning**.

- But, at the same time, we don't want to **undo** the work that we did **training** layer 1.
- So, we want to preserve the information in layer 1, while making it easier to use.

In our example, we mentioned that the scale of the inputs might get larger: they all get bigger or smaller, at the same time.

But, we often want to know what makes these inputs **different** from each other, so we can compare them: it's not helpful if all of them become larger/smaller.

So, we'll **standardize** each of our mini-batches of data, between each layer:

- **Subtracting the mean**: we take the mean of the mini-batch input, and subtract it from each data point. So, our standardized data is always centered at 0.
- **Dividing by standard deviation**: we compute how "spread out" the data is, and scale by that factor. Our standardized data is always the same amount "spread out".

This is exactly the same as when we standardized in the Feature Transformation chapter.

We repeat this process in between each layer of our network. Each layer receives a set of inputs with mean 0, standard deviation 1, no matter how the earlier layers change.

For now, we'll apply this to the pre-activation Z.

Below, we exclude the ℓ superscript in z^{ℓ}, μ^{ℓ}, σ^{ℓ}, and n^{ℓ} for readability.

Notation 30

We'll represent the element in the i^{th} row, j^{th} column, as Z_{ij}.

Concept 31

In order to deal with **internal covariate shift**, we'll take each mini-batch of k **pre-activations** to each **layer** of our neural network, and **standardize**/normalize it.

- Each **dimension** of the input is standardized **separately**.

Here, we focus on a single element: dimension i of the j^{th} data point in our batch, z_{ij}.

- We **compute** the mean μ and standard deviation σ, for batch size k.

$$\mu_i = \frac{1}{K}\sum_{j=1}^{K} Z_{ij} \qquad \sigma_i^2 = \frac{1}{K}\sum_{j=1}^{K}(Z_{ij} - \mu_i)^2$$

- We **subtract** the mean, divide by the standard deviation (include ϵ term to avoid dividing by zero)

$$\overline{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

After **standardizing**, this data is sent forward through the network.

- This is equivalent to using a "standardizing function" **after** the weight function $Z = W^T A$, and **before** the activation function $f(Z)$ (now $f(\overline{Z})$).
- We could also standardize **after** activation, but it's unclear which approach is better.

We can insert our new module into our existing model:

Normalization/standardization can be treated just like any other module.

Note that this preserves the **information** in our input data:

- If one data point has one feature much larger than another, you'll still see that: the gap will just be shifted over to zero, and normalized.

Example: Suppose you have some data: $[1, 2, 100, 3, 4, 5]$. If you standardize, you get

$$[-0.458, -0.433, 2.04, -0.408, -0.383, -0.358,] \tag{7.52}$$

The larger data point still stands out above the rest!

We need to be careful of dimensions:

Clarification 32

Normalization relies on the distribution (mean, s.d.) of our **mini-batch**.

But that means we can't just compute one data point at a time: we need to include the whole mini-batch of k **at the same time**.

So, we have to change the dimensions of Z^ℓ.

k is our **batch size**, while n is the number of **dimensions**.

- Z^ℓ without norm: $(n^\ell \times 1)$
- Z^ℓ with norm: $(n^\ell \times k)$

We use Z^ℓ_{ij} to indicate the i^{th} dimension of the j^{th} data point.

7.8.3.3 Post-Normalization: Choose Mean and S.D.

Now, we've adjusted it so that our distribution doesn't "drift", based on our training.

But, now, we've **restricted** our model:

- We don't necessarily want our mean and standard deviation to be 0 and 1: it would be better to be able to **control** it.

To accomplish this, we'll **scale** and **shift** our input. Thus, we're choosing our mean/s.d. in a deliberate way.

- Each dimension needs its own mean and standard deviation.
- We have n dimensions, and we need one variable to handle mean (or s.d.) for each: we'll need an $(n \times 1)$ vector.

Concept 33

By doing **normalization**, we've transformed Z into $\overline{Z}$.

- This "resets" our mean and standard deviation to 0 and 1.

However, we want to be able to **control** our mean and standard deviation. To do so, we introduce two new parameters:

- G: An $(n \times 1)$ vector that **scales** each of our dimensions $\overline{Z}_i$, giving our **standard deviations**.
- B: An $(n \times 1)$ vector that **shifts** each of our dimensions $\overline{Z}_i$, giving our **means**.

Thus, we get the true output of **batch normalization**:

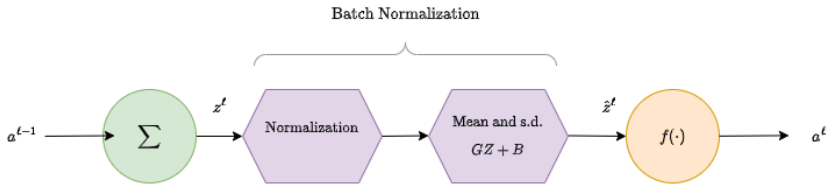
$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ij} + B_i$$

Example: Here's a sample example using a vector $\overline{z}_j$: only considering one, post-normalization data point j.

$$\begin{bmatrix} \widehat{Z}_{1j} \\ \widehat{Z}_{2j} \\ \vdots \\ \widehat{Z}_{kj} \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_k \end{bmatrix} * \begin{bmatrix} \overline{Z}_{1j} \\ \overline{Z}_{2j} \\ \vdots \\ \overline{Z}_{kj} \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{bmatrix} \tag{7.53}$$

Where $*$ indicates element-wise multiplication.

If we include this in our neuron graph, we now have two new modules:



7.8.3.4 Full definition

Definition 34

Batch Normalization is a process where we

- Standardize the pre-activation for each layer using the mean μ_i and standard deviation σ_i (for the i^{th} dimension). Select ϵ: $0 < \epsilon << 1$.

$$\overline{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

- Choose the new mean and standard deviation for the pre-activation using $(n \times 1)$ vectors G and B

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ij} + B_i$$

This process is meant to accomplish the following:

- Remove possible **internal covariance shift**: training earlier layers may change the scale of inputs to later layers.
  - This could make training more difficult.
- Allow our model to **select** a particular mean and s.d. for its pre-activation values, rather than arriving at them by chance.

It also tends to have a regularizing effect, and, in some learning algorithms, has replaced dropout.

7.8.3.5 Effectively Perturbs Data

We're not actually sure why normalization helps our models train. We originally designed it for **internal covariate shift**, but we're not sure if that's really **why** it works.

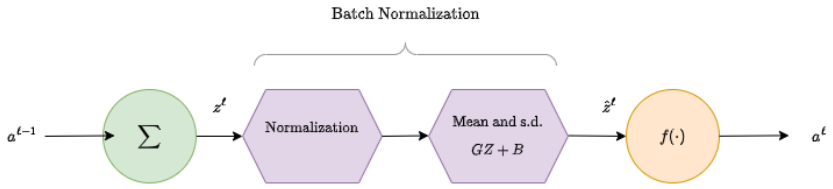
One explanation might be that, due to random sampling, each mini-batch ends up slightly **modified** by our normalization.

- Since each batch is likely to have a slightly different mean/standard deviation, each one ends up differently "perturbed" by normalization.

7.8.4 Applying batch normalization to backprop

Remark (Optional)
The following section mostly deals with the details of computing batch normalization derivatives.

As we showed with our figure,



Batch normalization adds two units that **interrupt** our chain of nested functions.

That means we need to figure out how to do backprop, bridging across the new gap between Z and $\widehat{Z}$.

So, that only leaves a couple problems:

- "Bridging the gap" between derivatives before and after BN, with $\partial \widehat{Z}^\ell / \partial Z^\ell$.
- Gradients for G and B: they're parameters now, too, so we need to train them.

Concept 35
Introducing batch normalization add new functions **in between** our old ones.

- Our input data has to travel through those additional layers.
- This **changes** the relationship between our current value, and the output loss.

So, in order to do backprop correctly, we have to figure out the **derivatives** of those functions.

7.8.4.1 Bridging the gap

We want to connect the start and end of batch normalization:

$$\frac{\partial \widehat{Z}}{\partial Z} \tag{7.54}$$

As usual, with the chain rule, we'll connect them by considering any values/function **between** them.

In this case, Z is normalized ($\overline{Z}$), and **then** we apply G and B ($\widehat{Z}$). We're missing the "normalized" step.

$$\frac{\partial \widehat{Z}}{\partial Z} = \frac{\partial \widehat{Z}}{\partial \overline{Z}} \cdot \frac{\partial \overline{Z}}{\partial Z} \tag{7.55}$$

Let's compare any two of these : $\widehat{Z}$ and $\overline{Z}$, for example.

- These are both $(n \times k)$ matrices.
- That means that each has two dimensions of variables. If we were to take the derivative between them, we would need $2 * 2$ axes: a **4-tensor**.

That sounds terrible. Instead, we'll compute these derivatives **element-wise**.

$$\frac{\partial \widehat{Z}_{ab}}{\partial Z_{ef}} = \frac{\partial \widehat{Z}_{ab}}{\partial \overline{Z}_{cd}} \cdot \frac{\partial \overline{Z}_{cd}}{\partial Z_{ef}} \tag{7.56}$$

7.8.4.2 Indexing

First, let's simplify these indices: only some pairs of elements matter.

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ik} + B_i \tag{7.57}$$

It seems that $\widehat{Z}_{ik}$ is only affected by the element with the same indices: $\overline{Z}_{ik}$.

Concept 36

$\widehat{Z}_{ik}$ is only a function of the same index element $\overline{Z}_{ik}$.

- Any other elements from $\overline{Z}$ has no effect.

$$(a \neq c \text{ or } b \neq d) \implies \frac{\partial \widehat{Z}_{ab}}{\partial \overline{Z}_{cd}} = 0$$

So, we write our remaining derivatives as $\partial \widehat{Z}_{ik} / \partial \overline{Z}_{ik}$.

What about the other derivative?

$$\overline{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon}$$

μ_i and σ_i include various different data points Z_{ik}, but only the i^{th} dimension. Z_{ik} requires the exact same indices.

Concept 37

$\overline{Z}_{ik}$ is only a function of elements in the same dimension i, Z_{ij}.

$$c \neq e \implies \frac{\partial \overline{Z}_{cd}}{\partial Z_{ef}} = 0$$

Our remaining derivatives take the form $\partial \overline{Z}_{ik} / \partial Z_{ij}$.

If we boil this down, we get all of our non-zero derivatives:

$$\frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} = \frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} \cdot \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} \tag{7.58}$$

7.8.4.3 Computing $\partial \widehat{Z}_{ik} / \partial \overline{Z}_{ik}$

We return to our previous equation:

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ik} + B_i \quad \implies \quad \frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} = G_i \tag{7.59}$$

7.8.4.4 Computing $\partial \overline{Z}_{ik} / \partial Z_{ij}$

For the other derivative:

$$\overline{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon} \tag{7.60}$$

This gets a bit complicated, because Z_{ij} can affect three different terms: Z_{ik}, μ_i, and σ_i.

We'll solve this by using the multi-variable chain rule.

We're linearly adding the effect due to each of these three variables, separately.

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \overbrace{\frac{\partial \overline{Z}_{ik}}{\partial Z_{ik}} \cdot \frac{dZ_{ik}}{dZ_{ij}}}^{Z_{ik}\text{'s effect}} + \overbrace{\frac{\partial \overline{Z}_{ik}}{\partial \mu_i} \cdot \frac{d\mu_i}{dZ_{ij}}}^{\mu_i\text{'s effect}} + \overbrace{\frac{\partial \overline{Z}_{ik}}{\partial \sigma_i} \cdot \frac{d\sigma_i}{dZ_{ij}}}^{\sigma_i\text{'s effect}} \tag{7.61}$$

In each of these terms, we treat the other two variables as "constant".

7.8.4.5 Lots of mini-derivatives

Let's compute the $\overline{Z}$ derivatives.

$$\overline{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon} \tag{7.62}$$

gives us

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ik}} = \frac{1}{\sigma_i + \epsilon} \qquad \frac{\partial \overline{Z}_{ik}}{\partial \mu_i} = \frac{-1}{\sigma_i + \epsilon} \qquad \frac{\partial \overline{Z}_{ik}}{\partial \sigma_i} = -\left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \tag{7.63}$$

Now, let's compute the Z_{ij} derivatives.

$$\boxed{\frac{\partial Z_{ik}}{\partial Z_{ij}} = \delta_{jk}} = \mathbf{1}(j = k) = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases} \tag{7.64}$$

$$\mu_i = \frac{1}{K}\sum_{j=1}^{K} Z_{ij} \implies \boxed{\frac{\partial \mu_i}{\partial Z_{ij}} = \frac{1}{K}} \tag{7.65}$$

$$\sigma_i^2 = \frac{1}{K}\sum_{j=1}^{K} (Z_{ij} - \mu_i)^2 \implies \boxed{\frac{\partial \sigma_i}{\partial Z_{ij}} = \frac{Z_{ij} - \mu_i}{K\sigma_i}} \tag{7.66}$$

7.8.4.6 Assembling our derivatives

Now, we plug them in.

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \frac{\partial \overline{Z}_{ik}}{\partial Z_{ik}} \cdot \frac{dZ_{ik}}{dZ_{ij}} + \frac{\partial \overline{Z}_{ik}}{\partial \mu_i} \cdot \frac{d\mu_i}{dZ_{ij}} + \frac{\partial \overline{Z}_{ik}}{\partial \sigma_i} \cdot \frac{d\sigma_i}{dZ_{ij}} \tag{7.67}$$

First, the $\overline{Z}$ derivatives.

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \frac{dZ_{ik}}{dZ_{ij}} - \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \frac{d\mu_i}{dZ_{ij}} - \left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \cdot \frac{d\sigma_i}{dZ_{ij}} \tag{7.68}$$

And now the Z_{ij} derivatives.

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \delta_{jk} - \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \left(\frac{1}{K}\right) - \left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \cdot \left(\frac{Z_{ij} - \mu_i}{K\sigma_i}\right) \tag{7.69}$$

Key Equation 38

We have found the batch normalization derivatives

$$\frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} = G_i$$

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \frac{1}{K(\sigma_i + \epsilon)}\left(K\delta_{jk} - 1 - \frac{(Z_{ik} - \mu_i)(Z_{ij} - \mu_i)}{\sigma_i(\sigma_i + \epsilon)}\right)$$

Which we multiply together to find $\partial \widehat{Z}_{ik}/\partial Z_{ij}$.

Once we've computed these derivatives, we can use them to extend the chain of $\partial L/\partial \widehat{Z}_{ik}$.

We're aiming to handle both of the batch normalization functions, with $\partial L/\partial Z_{ij}$.

We're going from $\widehat{Z}_{ik}$, which is post-BN, to Z_{ij}, which is pre-BN.

- Z_{ij} affects every data point $\widehat{Z}_{ik}$, and every data point affects L.
- So, we'll have to consider every data point k using the multi-variable chain rule:

$$\frac{\partial L}{\partial Z_{ij}} = \underbrace{\sum_{k=1}^{K}}_{\text{MV Chain Rule}} \overbrace{\frac{\partial L}{\partial \widehat{Z}_{ik}} \cdot \frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}}}^{\text{Data point k's effect}} \tag{7.70}$$

We can use this to go further back in our layers: as far as we want, as long as we don't forget this derivative!

7.8.4.7 Gradients for B and G

We want $\partial L/\partial B$ and $\partial L/\partial G$. Thanks to the work we did just now, we can travel backwards through numberous layers, to reach any B^{ℓ} and G^{ℓ}.

So, we'll assume we have $\partial L/\partial \widehat{Z^{\ell}}$. Once again, we'll omit the layer notation.

- We'll focus on a single bias, B_i: this biases one dimension of $\widehat{Z}_{ik}$.

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ik} + B_i \quad \Longrightarrow \quad \frac{\partial \widehat{Z}_{ik}}{\partial B_i} = 1 \tag{7.71}$$

- This bias is applied to every of our K data points: every data point affects the loss L. So, we'll have to sum them with the multi-variable chain rule.

This is exactly the same as how we did $\partial L/\partial Z_{ij}$.

$$\frac{\partial L}{\partial B_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} \cdot \frac{\partial \widehat{Z}_{ik}}{\partial B_i} = \boxed{\sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}}} \tag{7.72}$$

Now, we focus on a single scaling factor G_i:

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ik} + B_i \quad \Longrightarrow \quad \frac{\partial \widehat{Z}_{ik}}{\partial G_i} = \overline{Z}_{ik} \tag{7.73}$$

And once again, we add across different data points:

$$\frac{\partial L}{\partial G_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} \cdot \frac{\partial \widehat{Z}_{ik}}{\partial G_i} = \boxed{\sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} \overline{Z}_{ik}} \tag{7.74}$$

We're finished.

Key Equation 39

Here, we have the gradients for the batch scaling coefficients, B and G.

$$\frac{\partial L}{\partial B_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial \hat{Z}_{ik}}$$

$$\frac{\partial L}{\partial G_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial \hat{Z}_{ik}} \overline{Z}_{ik}$$

7.9 Terms

Section 7-1

- Neuron (Unit, Node)
- Neural Network
- Series and Parallel
- Linear Component
- Weight w
- Offset (Bias, Threshold) w_0
- Activation Function f
- Pre-activation z
- Activation a
- Identity Function
- Acyclic Networks
- Feed-forward Networks
- Layer
- Fully Connected
- Input dimension m
- Output dimension n
- Weight Matrix
- Offset Matrix
- Layer Notation A^{ℓ}
- Step function
- ReLU function
- Sigmoid function
- Hyperbolic tangent function
- Softmax function

Section 7-1.5

- Forward pass
- Back-Propagation
- Weight gradient
- Matrix Derivative
- Partial Derivative
- Multivariable Chain Rule
- Total Derivative
- Size of a matrix
- Planar Approximation
- Scalar/scalar derivative
- Vector/scalar derivative
- Scalar/vector derivative
- Vector/vector derivative

Section 7-2

- Mini-batch
- Vanishing/Exploding Gradient
- Weighted Average
- Running Average
- Exponential Moving Average
- Discount Factor
- Momentum
- Adadelta
- Adagrad (Optional)
- Adam
- Normalization
- Early stopping (Review)
- Weight Decay

- Perturbation
- Dropout
- Covariate Shift
- Internal Covariate Shift
- Batch Normalization
- Multivariable Chain Rule (Review)