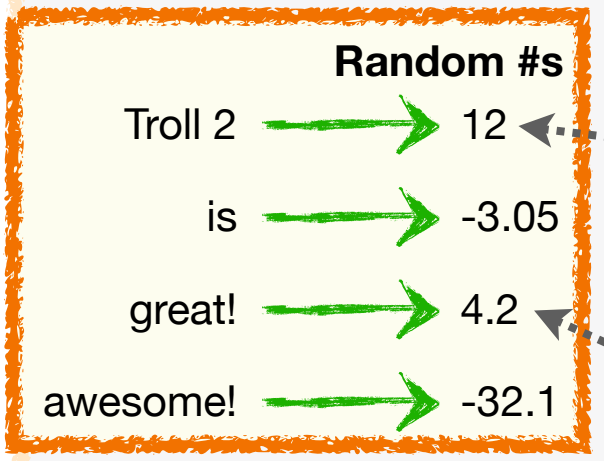
Chapter 09

Converting Words to Numbers with Word Embedding!!!

Word Embedding: Main Ideas

1 **A Problem:** Words are great, and we can use them to communicate all kinds of cool ideas, but neural networks work with numbers, not words.

So if we want to plug words into a neural network, we need a way to turn them into numbers.



In theory, we could just assign random numbers to words…

…but then, even though **great** and **awesome** mean similar things, they have very different numbers associated with them, **4.2** and **-32.1**…

…and that means the neural network will probably need a lot more complexity and training because learning how to correctly use the word **great** won't help the neural network correctly use the word **awesome**.

So it would be nice if similar words that are used in similar ways could be given similar numbers so that learning how to use one word would help the neural network learn how to use the other word at the same time.

And because the same words can be used in different contexts, or made plural or used in some other way, it might be nice to assign each word more than one number, so that the neural network can more easily adjust to different contexts.

2 **A Solution:** Surprisingly simple neural networks can assign similar numbers to similar words, and we call these numbers **Word Embeddings**. These simple networks can also create multiple Word Embeddings per word to handle the different contexts that a word can be used in.

For example, the words **great** and **awesome** have similar meanings and are often used in similar contexts. They can be used in a positive way…

StatQuest is **great!**

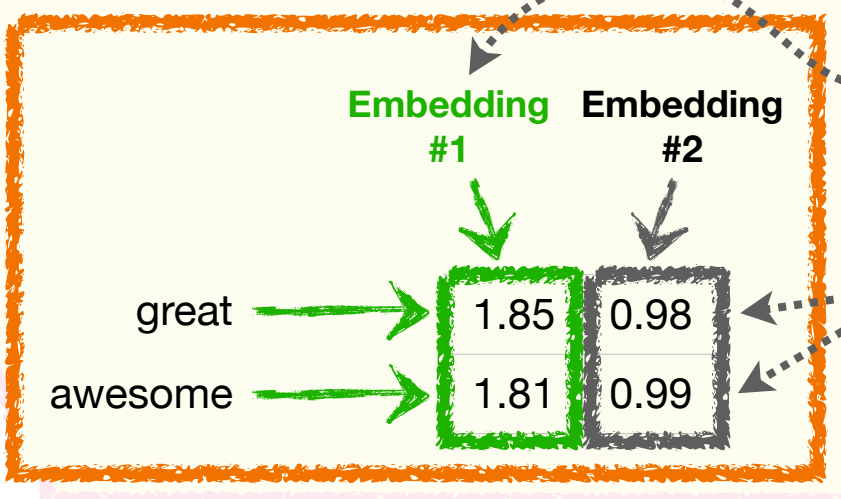
This book is **awesome!**

…and they can also be used in a sarcastic, negative way…

My cellphone's broken, **great**.

My dad's sense of humor is **awesome**.

…and a simple Word Embedding network can create two Word Embedding numbers for each word. The first number, **Embedding #1**, could capture the way these words are used in positive contexts.



And the second number, **Embedding #2**, could capture the way these words are used in negative contexts.

Now let's dive into the details of how Word Embeddings are created.

Word Embedding: Details

1 To show how we can get a super simple Word Embedding network to figure out what numbers should go with different words, let's imagine we have a simple training dataset that contains two phrases.

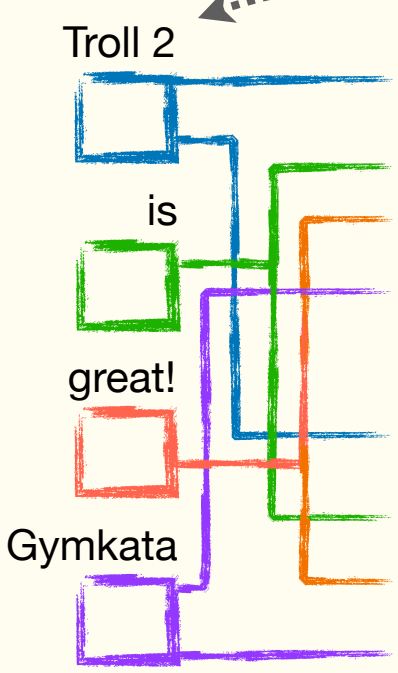
Training Data

Troll 2 is great!

Gymkata is great!

Both phases have two words, **Troll 2** and **Gymkata**, in similar contexts, because someone, and I'm not going to name names, thinks these two movies that have been consistently voted the worst movies ever made, are great!

2 In order to create a neural network to figure out which numbers we should associate with each word, the first thing we do is create inputs for each unique word.



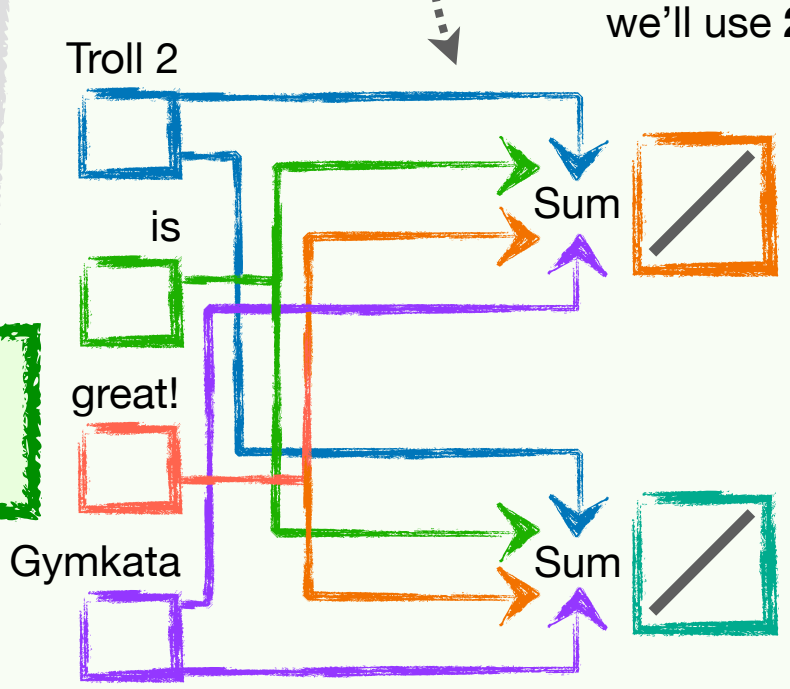
In this case, we have **4** unique words in the training data, so we have **4** inputs.

NOTE: In this example, we're going to treat **Troll 2** as a single word.

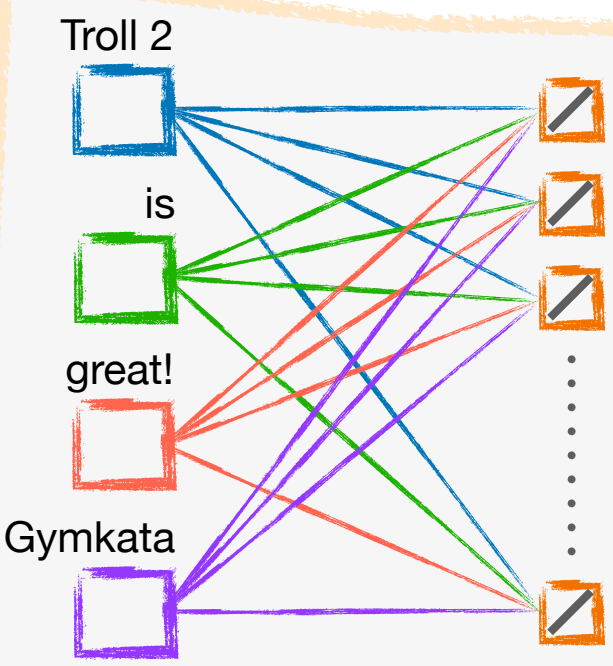
3 Now we connect each input to at least **1** activation function.

The number of activation functions corresponds to how many numbers we want to associate with each word.

In this example, we want to associate **2** numbers with each word, so that means we'll use **2** activation functions.



NOTE: These activation functions use the **Identity** function, and thus, the input values are the same as the output values. In other words, these activation functions don't do anything except give us a place to do a summation.

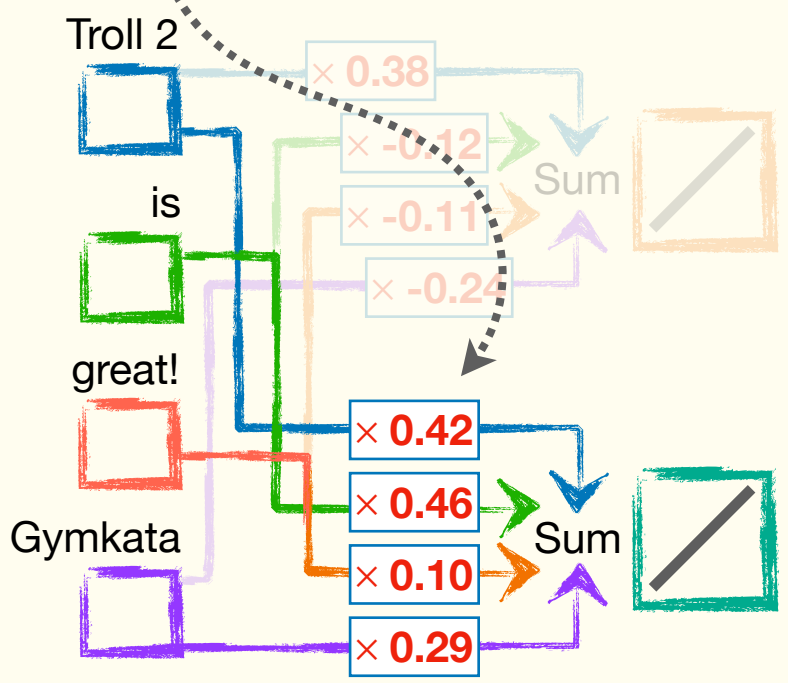


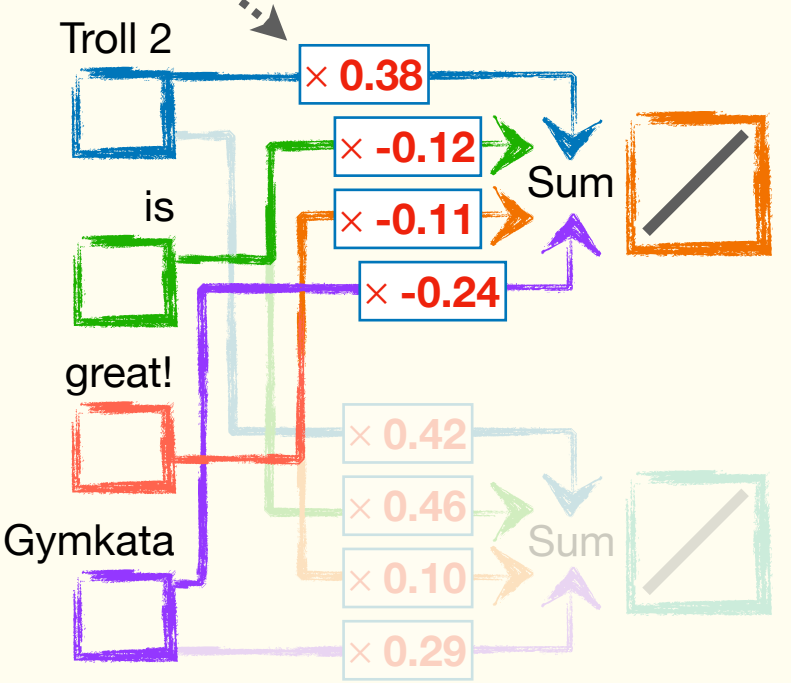
4 Also, although our example only uses **2** activation functions, so will only give us **2** Word Embedding values per input word, in practice, it's common to use *several hundred* activation functions, resulting in *several hundred* Word Embedding values for each word.

Word Embedding: Details

5 The weights on the connections to the top activation function will be the first set of Word Embeddings that we associate with each word.

And the weights on the connections to the bottom activation function will be the second set of Word Embeddings that we associate with each word.

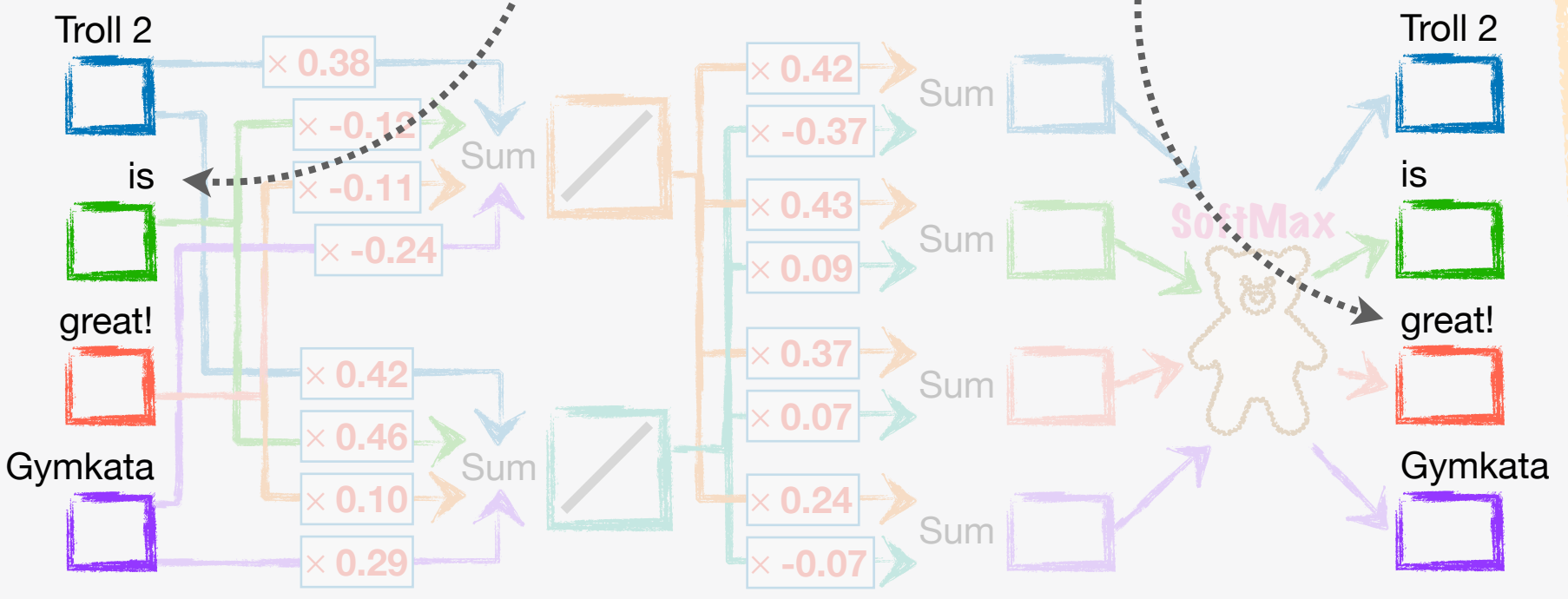




Like always, these weights start out with random values that we optimize with backpropagation.

6 And in order to do backpropagation, we have to make predictions, so, in this example, we'll make each input word…

…predict the next word in the phrase.



Word Embedding: Details

Training Data

Troll 2 is great!

Gymkata is great!

7 Thus, given the two phrases in the training dataset…

…we want **Troll 2** to predict **is**…

…and we want **is** to predict **great!**…

…and we want **Gymkata** to predict **is** because the word that comes after **Gymkata** is **is**.

Troll 2

is

great!

Gymkata

Troll 2

is

great!

Gymkata

8 Now, to see what word **Troll 2** predicts, we put a **1** in the input for **Troll 2**…

…and **0s** into all of the other inputs.

Troll 2: 1

is: 0

great!: 0

Gymkata: 0

TERMINOLOGY ALERT!!!

When one input gets a **1** and all of the other inputs get a **0**, this is called **One-Hot Encoding**.

Fun facts: **Troll 2** is awesome, but it's not a sequel and it's not about trolls!!!

9 Then we do the math, running the outputs from the activation functions through connections to the outputs…

…and running the outputs through the SoftMax function.

Troll 2: 1

is: 0

great!: 0

Gymkata: 0

× 0.38, × -0.12, × -0.11, × -0.24 → Sum

× 0.42, × 0.46, × 0.10, × 0.29 → Sum

× 0.42, × -0.37, × 0.43, × 0.09 → Sum → 0.01

× 0.37, × 0.07, × 0.24, × -0.07 → Sum → 0.17, 0.06

Sum → 0.20

SoftMax

Troll 2: 0.23

is: 0.27

great!: 0.26

Gymkata: 0.24

Word Embedding: Details

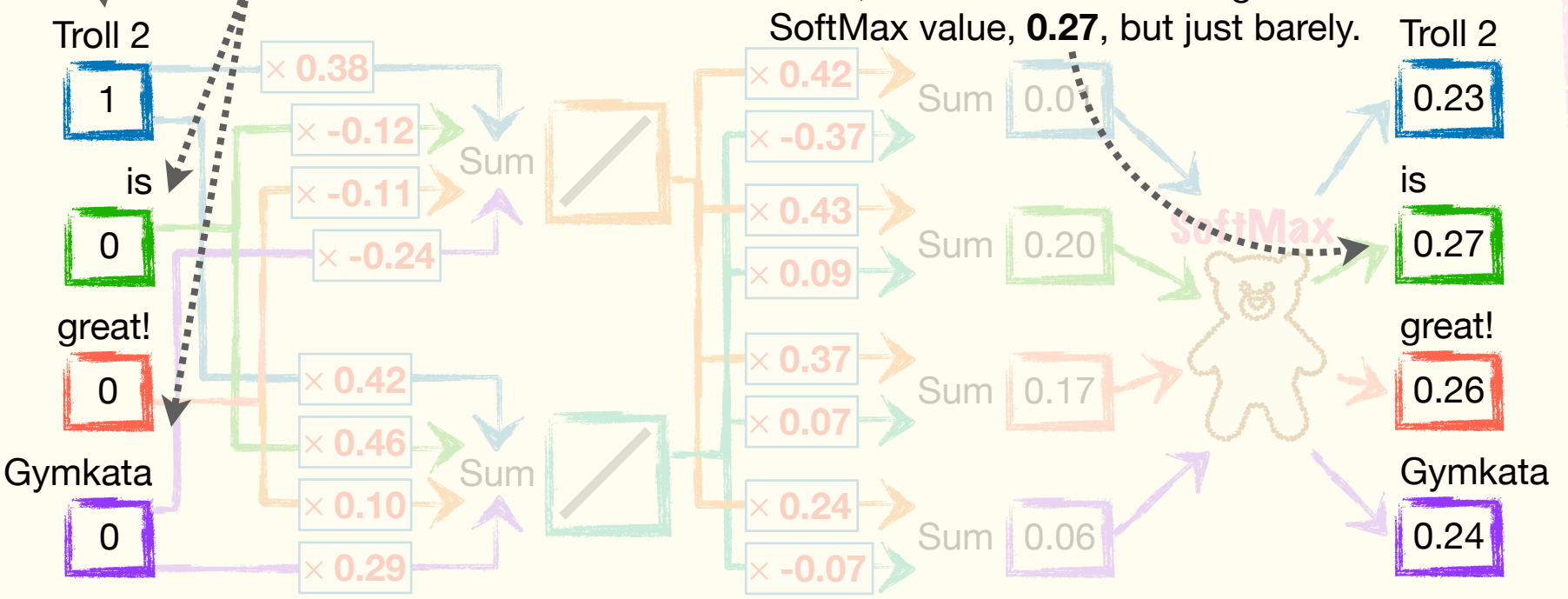
Training Data

Troll 2 is great!

Gymkata is great!

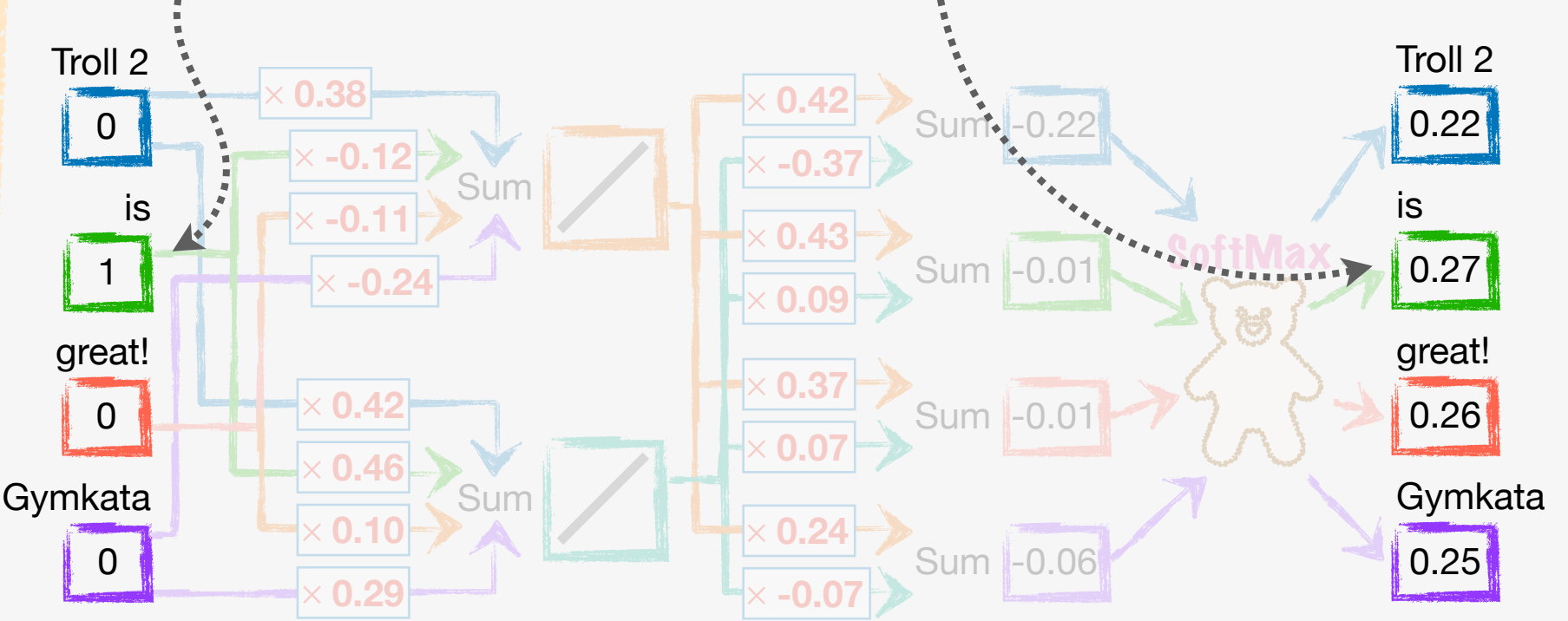
10 Thus, using the untrained weights, putting a **1** in the input for **Troll 2** and **0s** in all of the other inputs…

…correctly predicts the next word, **is**, because it has the largest SoftMax value, **0.27**, but just barely.



11 To see what the word **is** predicts, we put a **1** in the input for **is**, and **0s** in all of the other inputs…

…and the untrained weights incorrectly predict the next word to be **is** instead of the correct word: **great!**

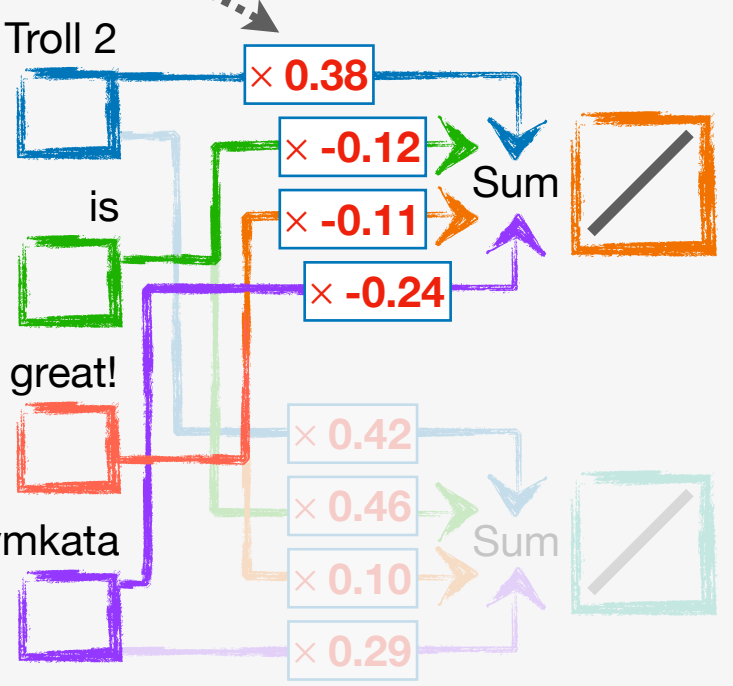


12 So, just as we expected, we're going to have to train the weights in this neural network.

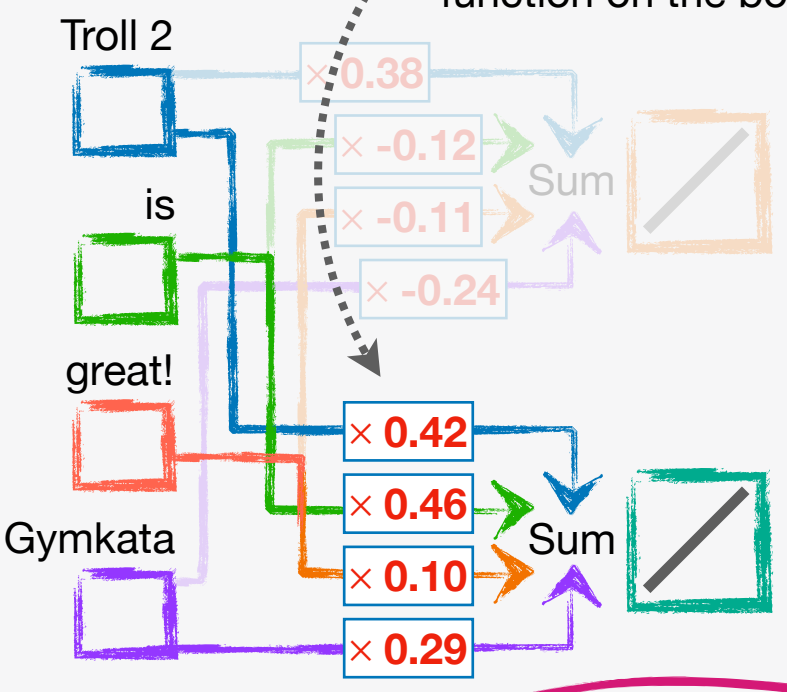
However, before we do that, we're going use the Word Embedding values to draw a cool graph.

Word Embedding: Details

13 Since we have one Word Embedding per word in our vocabulary going to the activation function on the top…

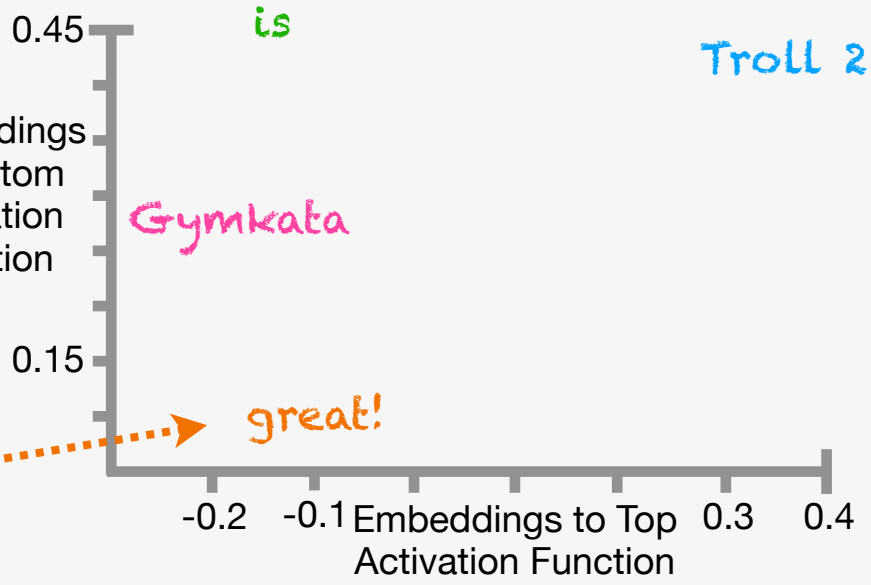


…and one Word Embedding per word going to the activation function on the bottom…



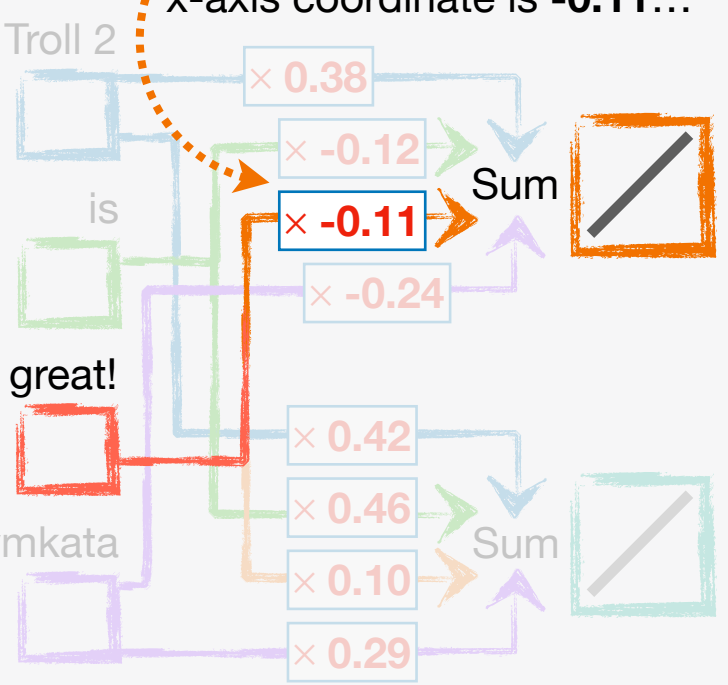
…we can plot each word on a graph, with the Word Embedding to the top activation function on the x-axis…

…and the Word Embedding to the bottom activation function on the y-axis.

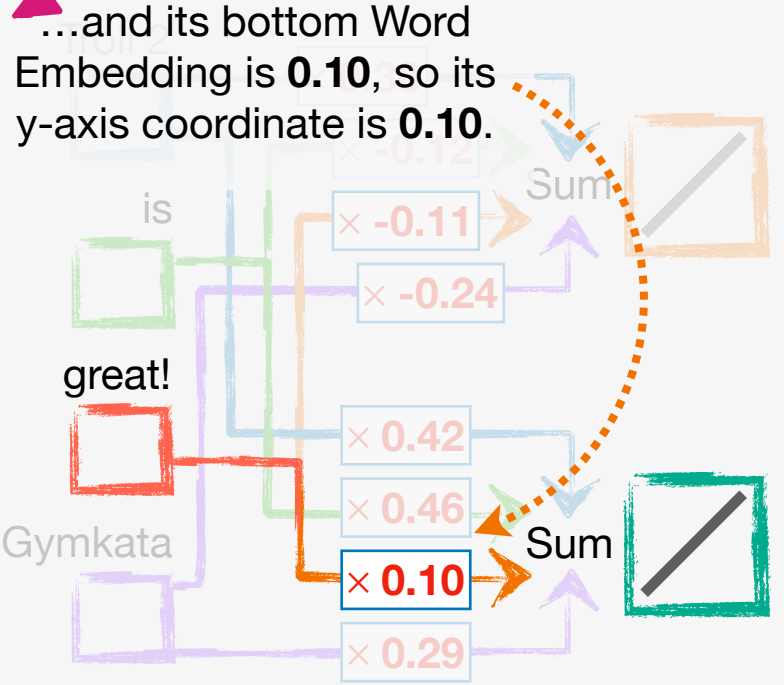


For example, the word **great!** goes here…

…because its top Word Embedding is **-0.11**, so its x-axis coordinate is **-0.11**…



…and its bottom Word Embedding is **0.10**, so its y-axis coordinate is **0.10**.



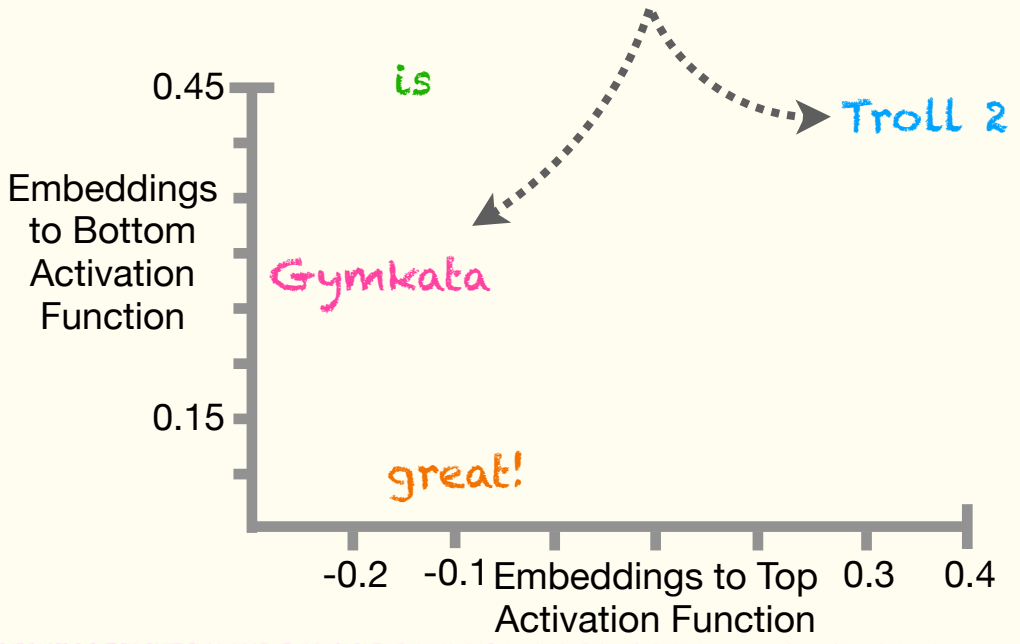
Word Embedding: Details

Training Data

Troll 2 is great!

Gymkata is great!

14 Now, with this graph based on the untrained Word Embeddings, we see that the words **Troll 2** and **Gymkata**…

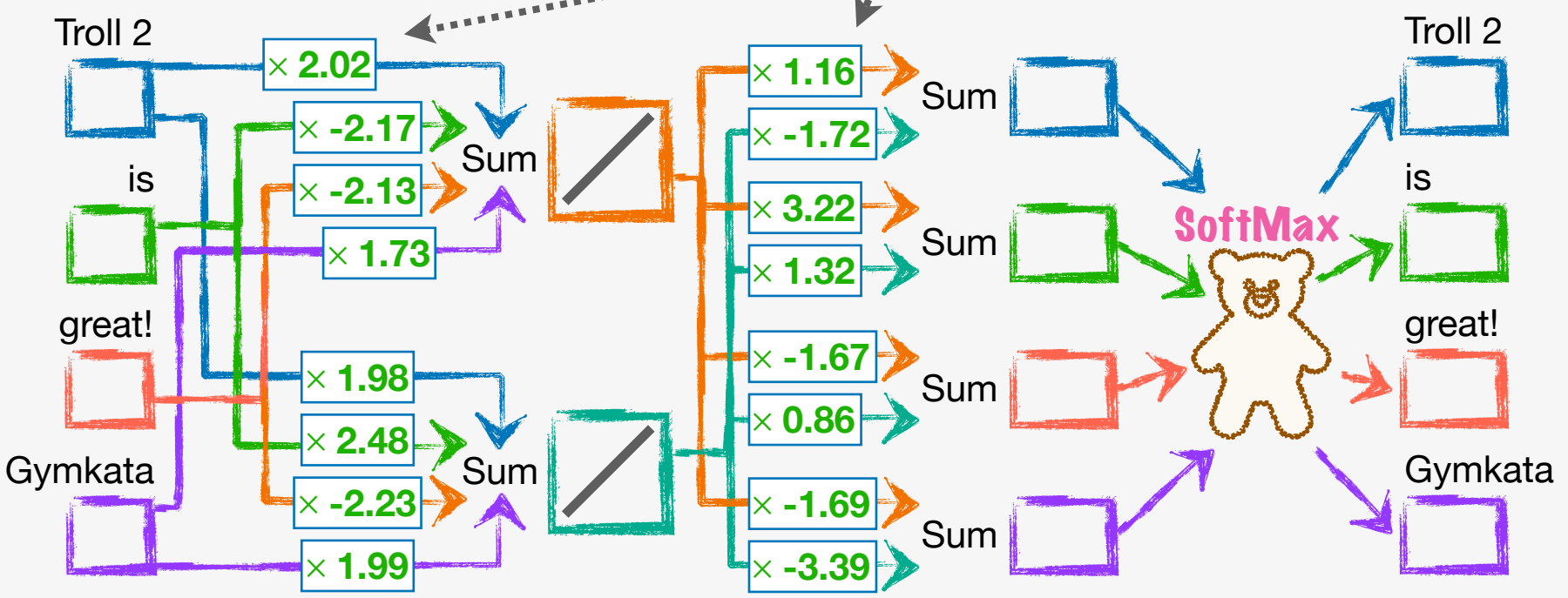


…are no more similar to each other as they are to any of the other words in our vocabulary.

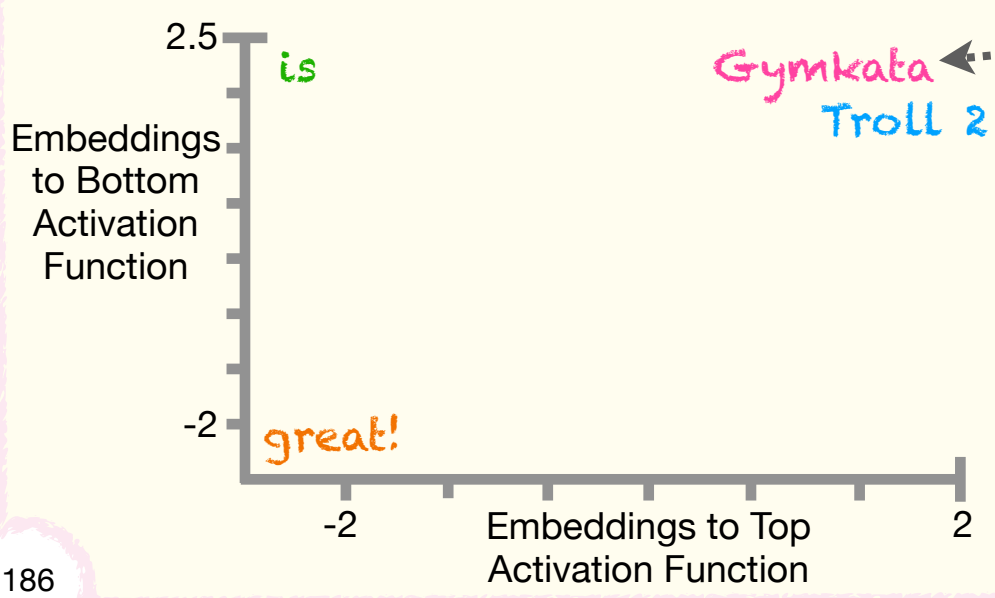
However, because both words appear in the same context in the training data…

…optimizing the Word Embeddings with backpropagation should make their weights more similar.

15 And when we use the training data to have one word predict the next word, and combine that with the Cross Entropy loss function to do backpropagation, we end up with these new weights.



16 The new weights on the connections from the inputs to the activation functions are the new Word Embeddings.



And when we plot the words using the new Word Embeddings, we see that **Troll 2** and **Gymkata** are now relatively close to each other compared to the other words in the training data.

BAM!

NOTE: Unlike the neural networks we looked at earlier, we only use the outputs of this Word Embedding network for training. Instead, the interesting stuff is the weights themselves!

Word Embedding: Details

Training Data

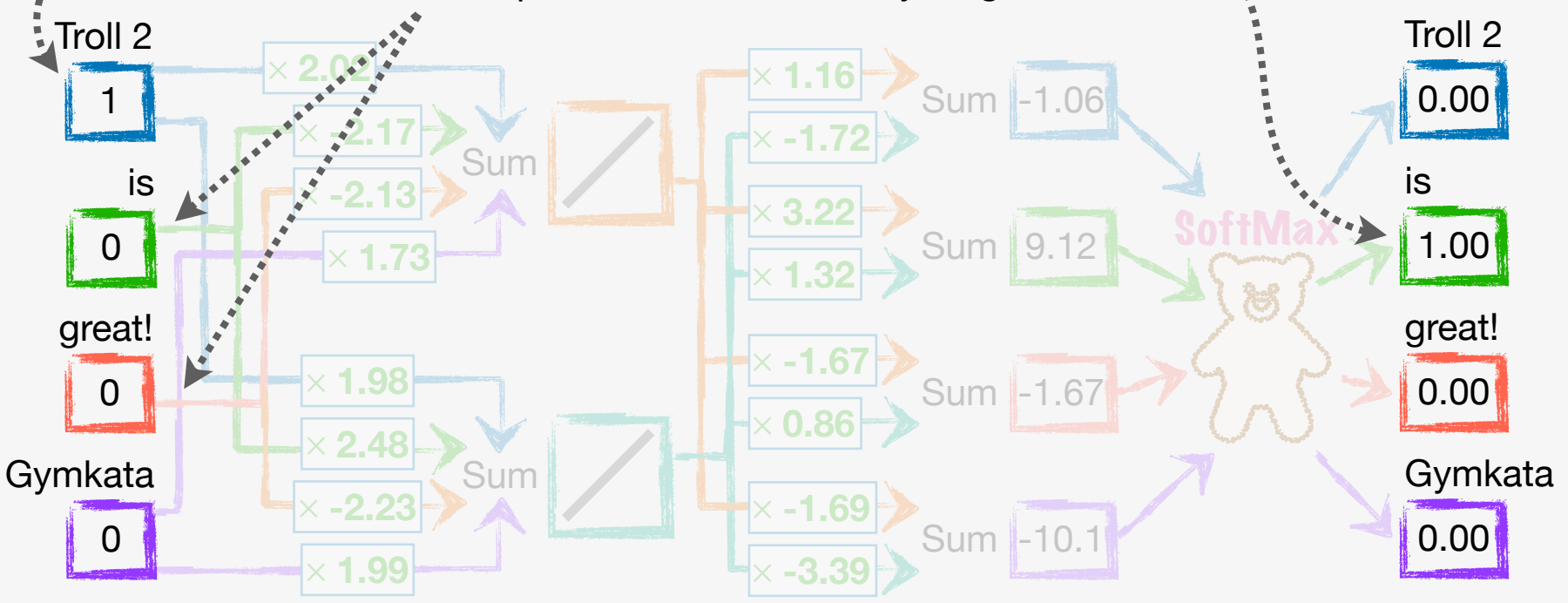
Troll 2 is great!

Gymkata is great!

17 Now that we've trained the neural network, we can plug a **1** into the input for **Troll 2** and **0s** into all of the other inputs…

…and the output for the next word, **is**, is **1,** and everything else is **0**.

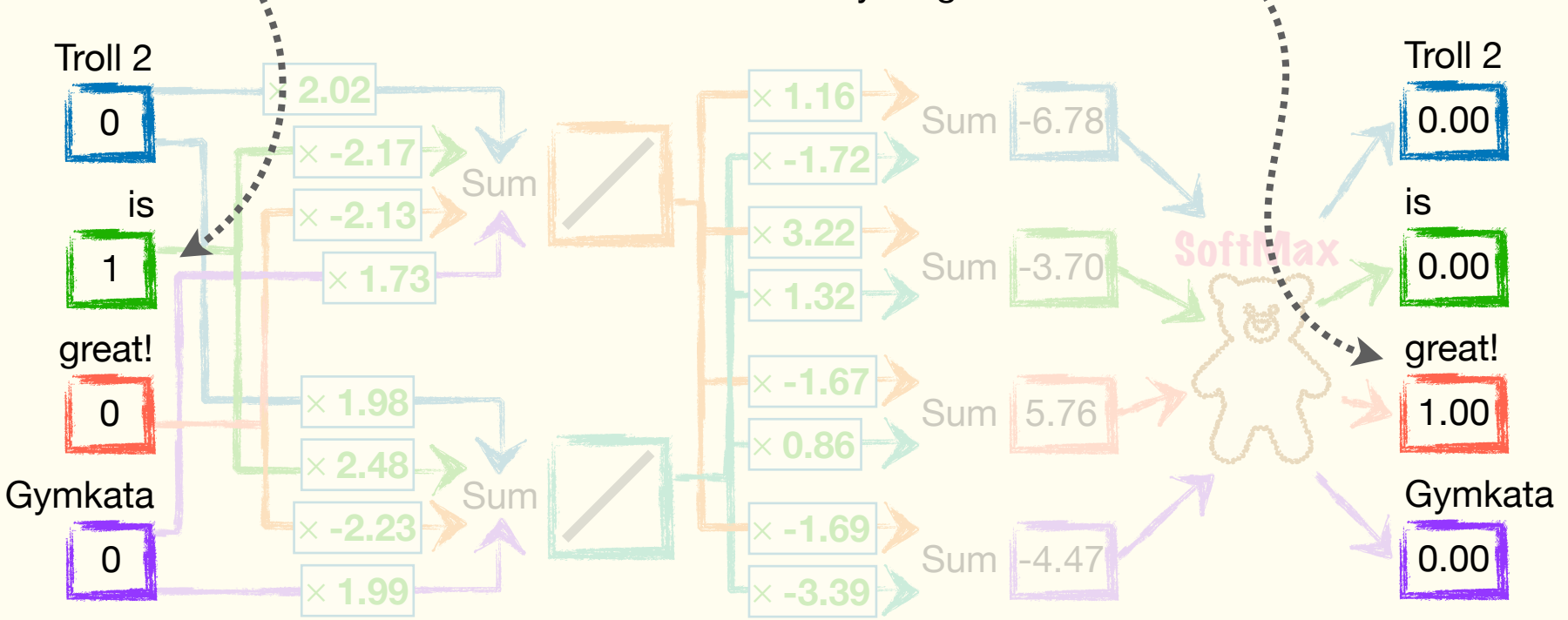
And this is exactly what we wanted.



18 And when we plug the word **is** into the input…

…then the output for the next word in both phrases, **great!**, is **1**, and everything else is **0**.

And this is exactly what we wanted.



DOUBLE BAM!!!

Word Embedding: Details

19 So far, we've shown we can train a neural network to predict the next word in each phrase in our training data, but just predicting the next word doesn't give the network a lot of context to understand each one.

A relatively popular Word Embedding network called **word2vec** uses two methods to make more context. The first is called the **Continuous Bag of Words** method, and it uses the surrounding words to predict what occurs in the middle.

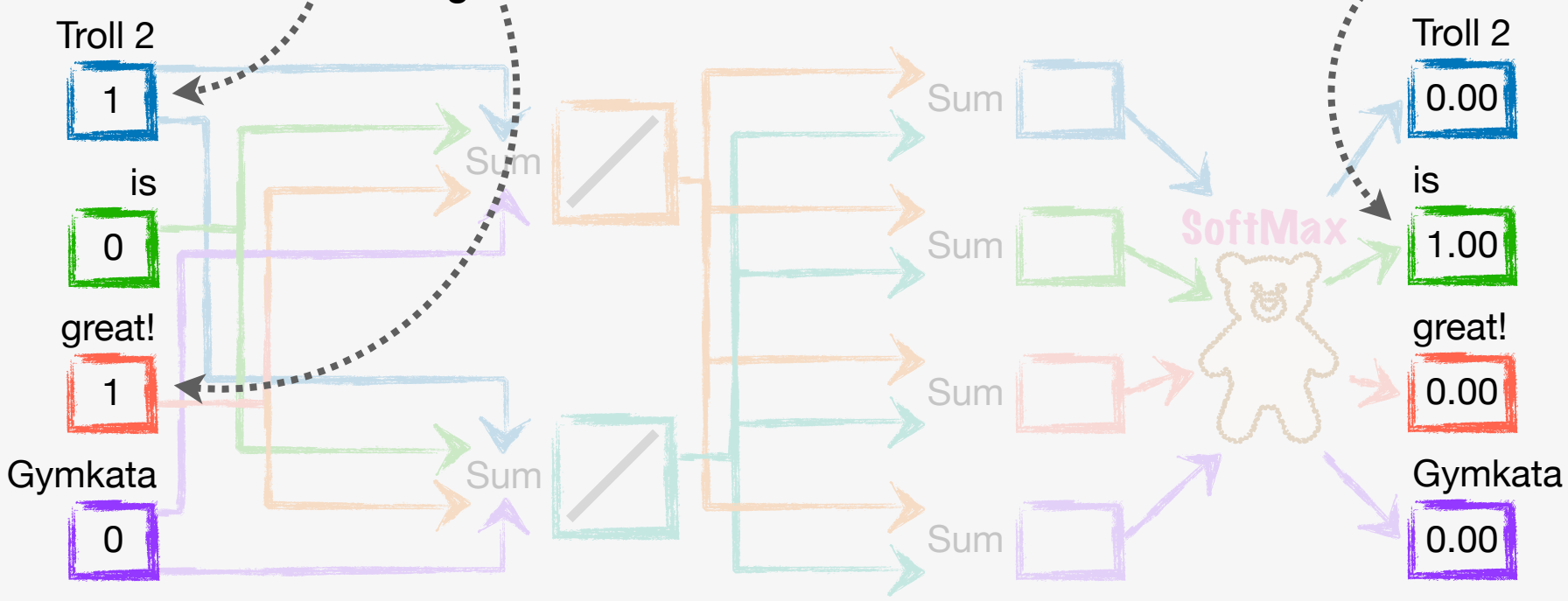
Training Data

Troll 2 is great!

Gymkata is great!

For example, the **Continuous Bag of Words** method could use the words **Troll 2** and **great!**…

…to predict the word that occurs between them, **is**.



20 The second method, called **Skip Gram**, increases the context by using the word in the middle to predict the surrounding words.

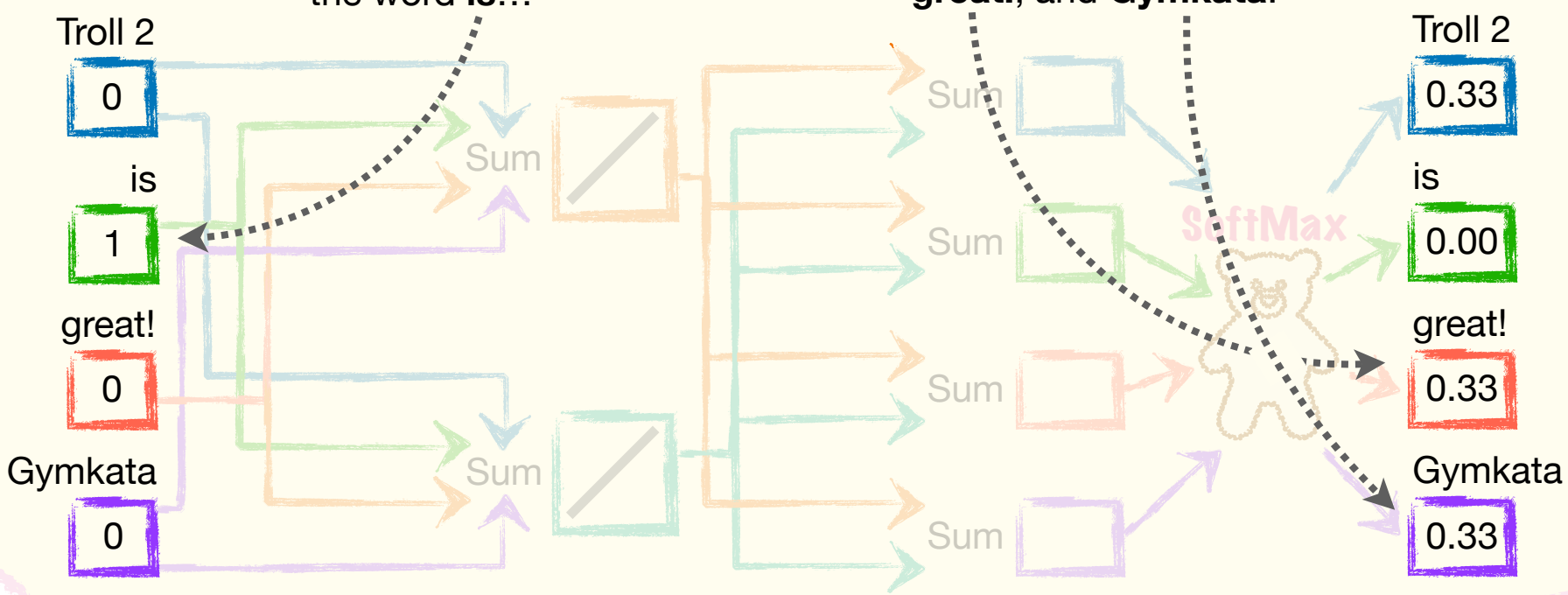
Training Data

Troll 2 is great!

Gymkata is great!

For example, the **Skip Gram** method could use the word **is**…

…to predict the surrounding words, **Troll 2**, **great!**, and **Gymkata**.



Word Embedding: Details

21 Lastly, in practice, instead of using a tiny training dataset with only two sentences to build a Word Embedding network, it's relatively common to use something large like the entire Wikipedia to build a Word Embedding network that has hundreds of activation functions to create hundreds of Word Embedding values for each word in a vocabulary with over 10,000 words.

Entire Wikipedia!!!

An aardvark eats…

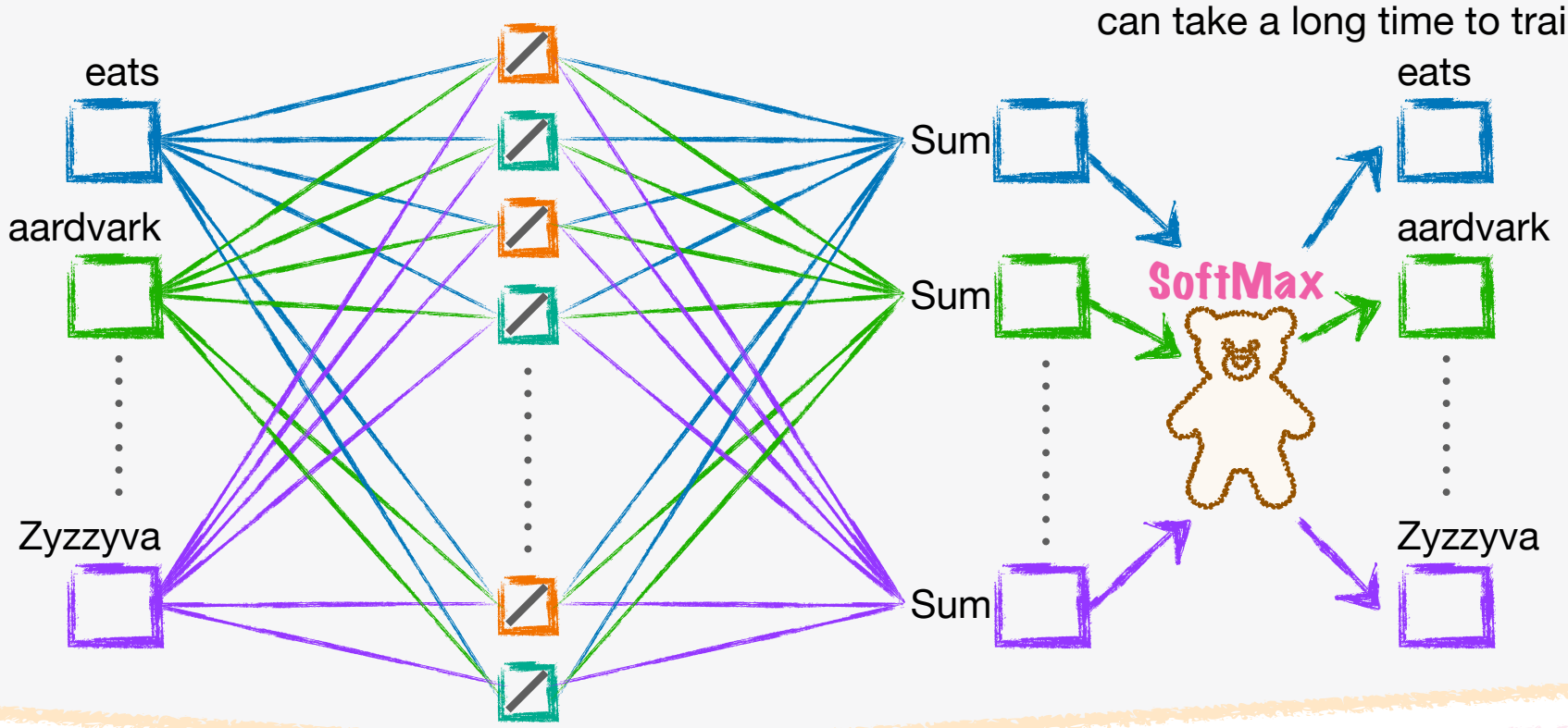
A pineapple is tasty…

StatQuest is a cool…

⋮

The Zyzzyva weevils…

And such a large network can take a long time to train.



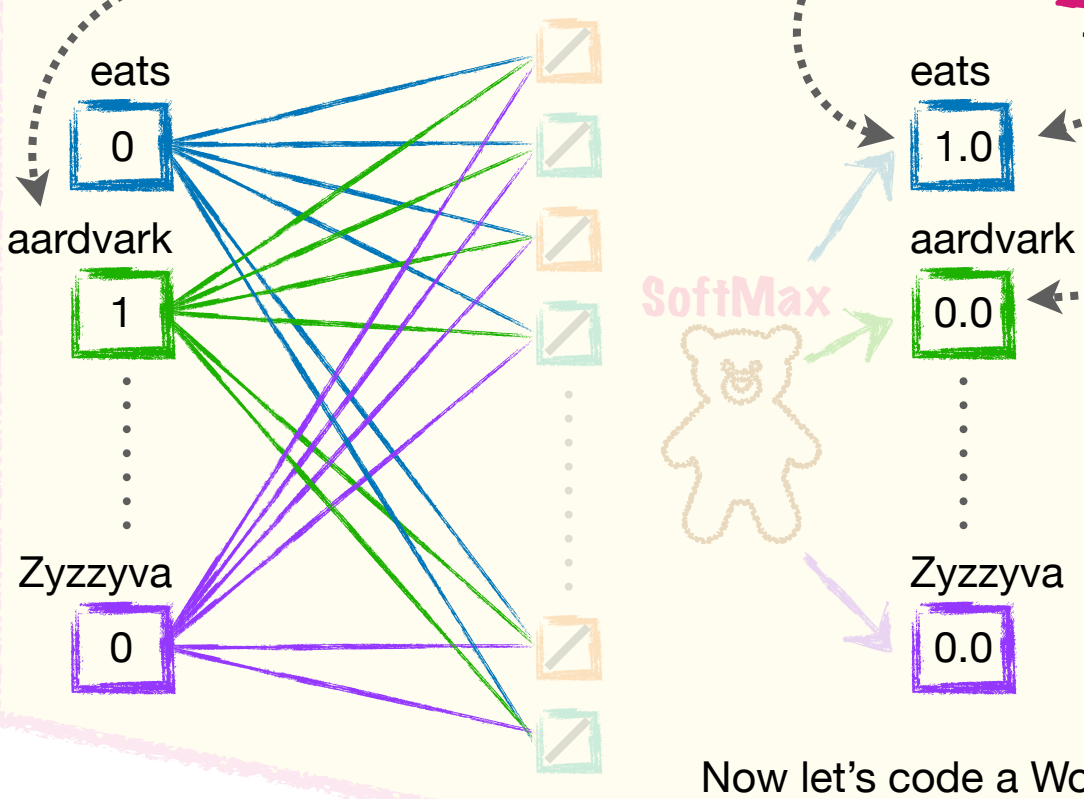
22 The good news is that **word2vec** introduced a method called **Negative Sampling** that can speed up training a large Word Embedding network.

Negative Sampling works by selecting a subset of output values to focus on, rather than focusing on all of them, which reduces the number of weights that need to be adjusted during each step of backpropagation.

For example, if we wanted the word **aardvark** to predict the word **eats**…

…then that means we want the output for **eats** to be **1.0**…

…and we want everything else to output **0.0**.



So **Negative Sampling** would speed up training by only focusing on the output for **eats**, which should output **1.0**, and just a few of the other randomly selected outputs, which should all output **0.0**, instead of *all* of the other outputs. Bam!

Now let's code a Word Embedding network in **PyTorch**!