Daily Insights

1. Walkthrough: The Transformer Architecture [Part 1/2]
2. Walkthrough: The Transformer Architecture [Part 2/2]
3. Understanding Batch Normalization
4. Rethinking Batch Normalization
5. A Survey of Early Impact Measures
6. Understanding Recent Impact Measures
7. Four Ways An Impact Measure Could Help Alignment
8. Why Gradients Vanish and Explode
9. A Primer on Matrix Calculus, Part 1: Basic review
10. A Primer on Matrix Calculus, Part 2: Jacobians and other fun
11. A Primer on Matrix Calculus, Part 3: The Chain Rule

Walkthrough: The Transformer Architecture [Part 1/2]

This is the first post in a new sequence in which I walk through papers, concepts, and blog posts related to machine learning and AI alignment.

In the process of increasing my own understanding of these topics, I have decided to implement a well known piece of advice: the best way to learn is to teach. This sequence is therefore a culmination of what I have learned or reviewed very recently, put into a learning format.

My thoughts on various topics are *not* polished. I predict there will be mistakes, omissions, and general misunderstandings. I also believe that there are better places to learn the concepts which I walk through. I am not trying to compete with anyone for the best explanations. The audience I have in mind is a version of myself from a few weeks or days ago, and therefore some things which I take for granted may not be common knowledge to many readers. That said, I think this sequence may be useful for anyone who wants a deeper look at the topics which I present.

If you find something wrong, leave a comment. Just try to be respectful.

If you've been following machine learning or natural language processing recently, you will likely already know that the Transformer is currently all the rage. Systems which are based on the Transformer have struck new records in natural language processing benchmarks. The GLUE benchmark is currently filled by models which, as far as I can tell, are all based on the Transformer.

With the Transformer, we now have neural networks which can write coherent stories about unicorns in South America, and an essay about why recycling is bad for the world.

The paper describing the Transformer, Attention Is All You Need, is now the top paper on Arxiv Sanity Preserver, surpassing the popularity of GANs and residual nets. And according to Google scholar, the paper has attracted 2588 citations (and counting).

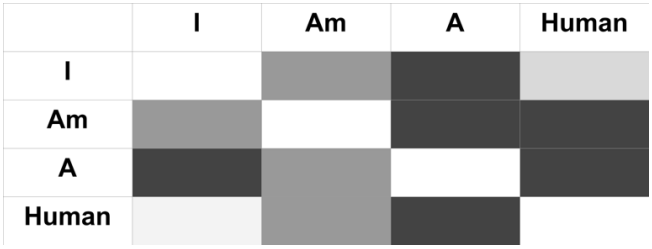
So what makes the Transformer so powerful, and why is it such a break from previous approaches? Is it all just hype? Clearly not. But it is worth looking into the history behind the architecture first, in order to see where these ideas first came from. In this post, I give a rough sketch for what the transformer architecture looks like. In the following post, I will provide a detailed description of each step in the forward pass of the architecture. First, however, we have to know what we are looking at.

The central heart of the Transformer is the attention mechanism, hence the name of the original paper. As I understand, attention is a mechanism first designed in order to improve the way that recurrent neural networks understood text. I found this post helpful for providing an intuitive breakdown of how the attention mechanism works in pre-Transformer architectures.

The way I view attention in my head, and the way that many people illustrate it, is to imagine a table linking every word in a sentence to every other word in another sentence. If the two sentences are the same, then this is called self-attention.

Attention allows us to see which parts of the sentence are relevant to the other parts. If there's a strong link between "it" and "car" then it's possible that this link is a way for the model to say that **it** *is* the **car**.

Consider the sentence "I am a human." Attention might look at this sentence and construct the following links:



The brighter the cell in the table, the more connected the two words are. Here, the exact meaning of the shades of grey aren't really important. All you need to notice is that there's some sort of relationship between words, and this relationship isn't identical between every word. It's not a symmetric relationship either. The exact way that attention is calculated will come later, which should shed light on why attention works the way it does.

If we were using a Transformer on the previous sentence, it might figure out that "I" was referring to "human." And in a sense, that's exactly what we want it to do. "I" and "human" are quite related, since they both point in the same direction. We will later see how we can use these linkages in a clever way to incorporate them into the Transformer architecture.

As a brief digression, I will note that the idea of coming up with a vague abstraction for how we want to interpret our neural networks is a concept that comes up repeatedly in deep learning. In the case of convolutional neural networks, for instance, we frequently describe neural networks as having some sort of structured model for the images we are classifying. This is why authors will sometimes talk about the model recognizing smaller pieces of the picture, like wheels, and then combining these smaller features into a coherent whole, such as a car.

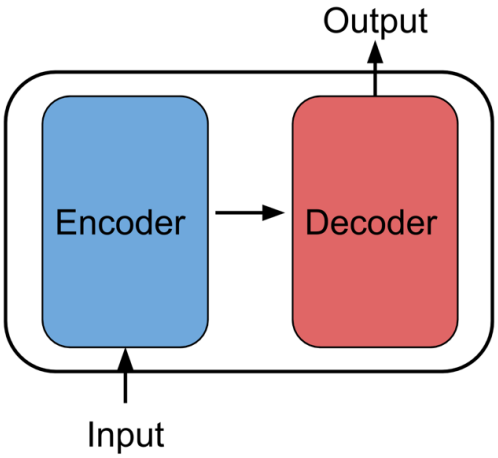
The way that I think about attention is similar. The hope is that the neural network will be able to capture those parts of the text that are related to each other in some way. Perhaps the words that are strongly linked together are synonyms, or stand for each other in some way, such as because one of them is a pronoun of another word. No part of the links are hard-coded, of course. Finding out which words are related is where most of the learning happens.

Unlike previous approaches that use attention, the Transformer is unique because of how it uses attention for virtually everything. Before, attention was something that was used in conjunction with another neural network, allowing some form of computation by the neural network on a a pre-processed structured representation. The Transformer takes this concept further by repeatedly applying attention to an input, and relying very little on traditional feed forward neural networks to turn that input into something useful. Transformers still *use* regular neural networks, their importance is just diminished. And no RNNs or CNNs need be involved.

Other than attention, the other main thing to understand about the Transformer is its general structure, the encoder-decoder architecture. This architecture is not unique to

the Transformer, and from what I understand, has been the dominant method of performing sequence to sequence modeling a few years before the Transformer was ever published. Still, it is necessary to see how the encoder-decoder architecture works in order to get any idea how the Transformer does its magic.

Below, I have illustrated the basic idea.



Simple enough, but how does it work? That middle arrow is conveying something useful. This isn't just a deep architecture with a strange standard representation.

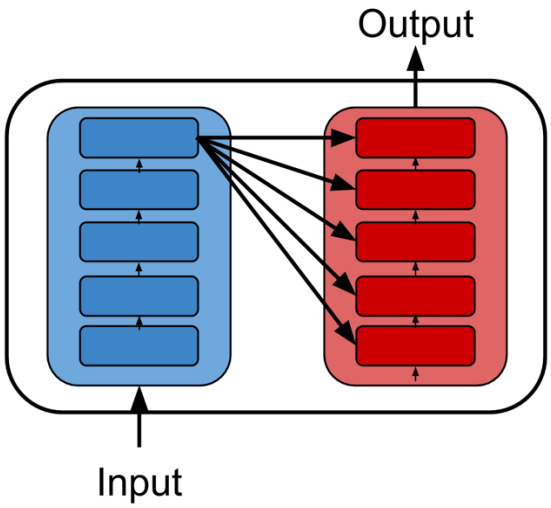
By thinking first about traditional neural networks, the justification for having this representation will become clear. In an RNN, we have the same neural network performing a computation several times on each part of a sequence, returning an output for each step. It is this sequential nature of the computation which limits the output of an RNN. If we want an RNN to translate a sentence in English with 5 words in it, to a sentence in French with 7 words in it, there seems to be no natural way to do it. This is because, intuitively, as we go along the input sequence, we can only translate each element in the input to one element in the output.

By contrast, the encoder-decoder mechanism gets around this by constructing two different networks which work together in a special way. The first network, the encoder, takes in an input and places that input into a very high dimensional space, the *context*. After the input has been translated into this high dimensional space, it is then put into a much lower dimension using the decoder. By having an initial ramp up to a high dimension, and then back down, we are free to translate between sequences of different lengths.

The exact way that it works is like this: in the decoder network, we first take in the context and an initial hidden state as inputs. We then repeatedly apply an RNN to these inputs, creating an output and passing along the hidden state information to the next step. We repeat this until the network has produced an end token, which indicates that it has reached the end of the translated text.

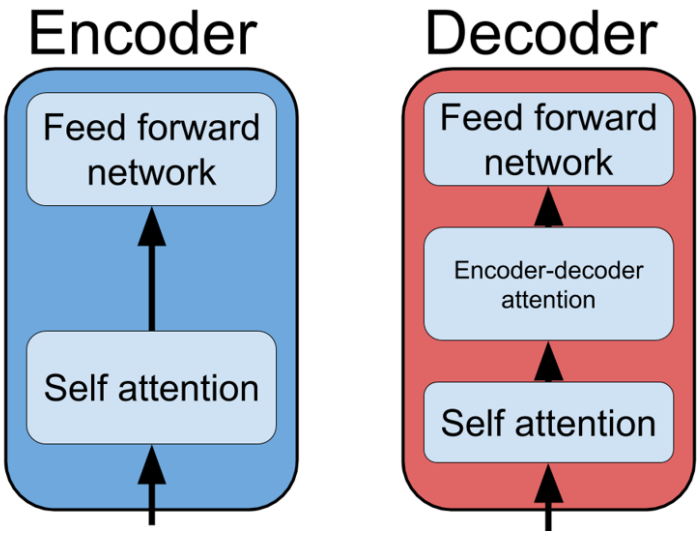
In the Transformer, we drop the RNN part, but keep the benefit of being able to map arbitrarily long sequences to other arbitrarily long sequences. Under the hood, the Transformer is really more of a stack of encoders and decoders, which are themselves composed of self-attention components and a feedforward neural network. The size of this stack is the main design choice involved in creating a Transformer, and contributes to the simplicity of tuning it.

Since there are so many small details in the Transformer, it's important to get a rough visual of how data is flowing through the network before you start to think about the exact computations that are being performed. We can look at the whole model like this.



The little stacked boxes I have put in encoder and decoder represent the layers in the network. The input is first fed up through the encoder, and then at the top step, we somehow provide some information to each layer of the decoder. Then, we start going through the decoder, at each step using the information we just got from the encoder.

Unlike an RNN, we do not share weights in each of these layers. The little boxes represent individual parts of the architecture that we are training separately. If we look under the hood at these little blocks, we find that encoder and decoder blocks are pretty similar.



These things are stacked inside the encoder and decoder, and feed upwards.

On the left, we have an encoder layer. This layer has two sublayers. The first layer is the aforementioned self-attention. Although I have not given the details yet as to how

attention works, we can visualize this block as calculating the values of the table I have above, followed by some computation which uses the numbers in the table to weight each of the words in the sequence. This weighting then *bakes* in some information into each word, before it is used by the feed forward network. If this currently doesn't make sense, it should hopefully become more apparent once we get into the actual vector and matrix operations.

On the right, we have a decoder layer. The decoder layer is almost identical to the encoder layer except that it adds one middle step. The middle step is the encoder-decoder attention. This sublayer is the one that's going to use the information carried over from the last step of the encoder layer.

Both the encoder and decoder layers feed into a neural network before shipping the values onto the next step.

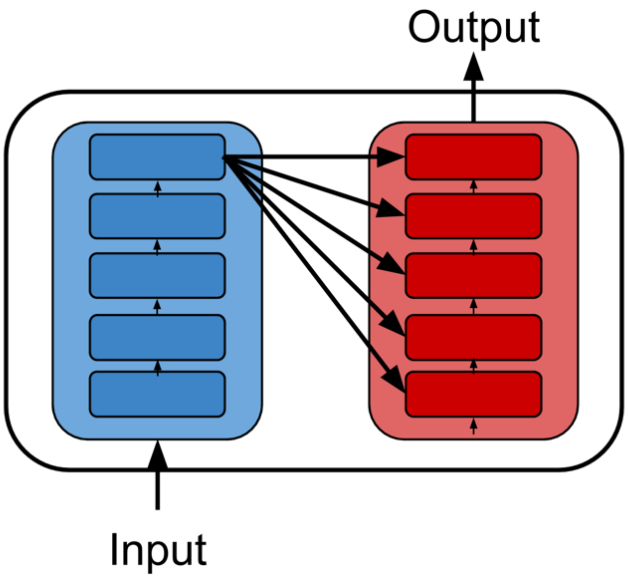
If you're anything like me, looking at this very conceptual picture has probably left you eager to dive into a more concrete analysis in order to truly understand what's going on. For that, we'll need to go into each step, and look at the architecture like an algorithm, rather than a picture. Part 2 of this post will do just that.

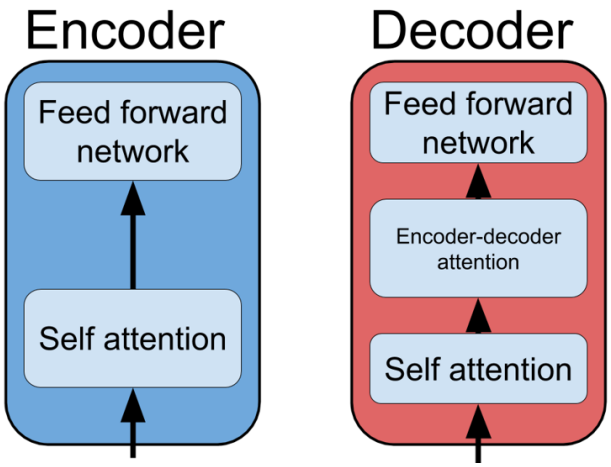
Just keep in mind that as long as we have a rough idea of what's going on, it will make all the little matrix computations and notation a little less painful. In my opinion, learning benefits immensely from an abstract first pass through the material followed by a more detailed second pass. So with that, join me tomorrow as I unravel the deeper mysteries of the Transformer.

Walkthrough: The Transformer Architecture [Part 2/2]

If you are already sort of familiar with the Transformer, this post can serve as a standalone technical explanation of the architecture. Otherwise, I recommend reading part one to get the gist of what the network is doing.

Yesterday, I left us with two images of the Transformer architecture. These images show us the general flow of data through the network. The first image shows the stack of encoders and decoders in their bubbles, which is the basic outline of the Transformer. The second image shows us the sublayers of the encoder and decoder.





Now, with the picture of how the data moves through the architecture, I will fully explain a forward pass with an example. Keep the general structure in mind as we go through the details, which can be a bit mind-numbing at parts.

The task

The Transformer is well suited for translating sentences between languages. Since I don't speak any language other than English, I have decided to translate between sarcastic sentences and their intended meaning. In the example, I am translating the phrase "Yeah right" to "That's not true." This way all those robots who only understand the literal interpretation of English can simply incorporate a Transformer into their brain and be good to go.

The embedding

Before the sentence "Yeah right" can be fed into the architecture, it needs to take a form that is understood by the network. In order to have the architecture read words, we therefore need embed each word into a vector.

We *could* just take the size of the vocabulary of the document we are working with, and then one-hot encode each word into a fantastically sparse and long vector. But this approach has two problems:

1. The high dimension that these vectors are in makes them harder to work with.

2. There's no natural interpretation of proximity in this vector space.

Ideally, we want words that are similar to be related in some way when they are embedded into a vector format. For instance, we might want all the fruits {apple, orange, pear} to be close to each other in some cluster, and far enough away from the vehicle cluster {car, motorcycle, truck} to form distinct groups.

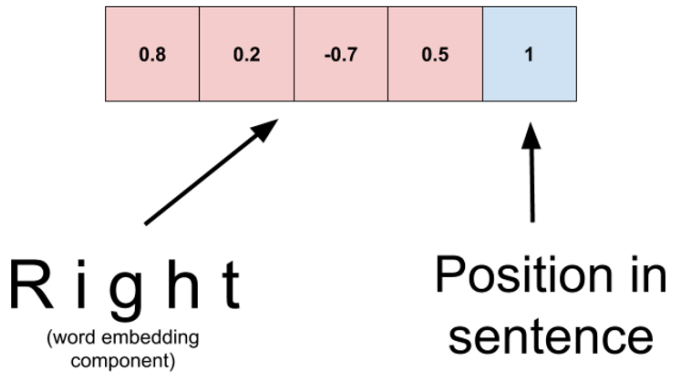
To be brief, the way that we do this is to use some word embedding neural network that has already been trained on English sentences.

Positional encoding

After we have obtained a set of word-embedded vectors for each word in our sentence, we still must do some further pre-processing before our neural network is fully able to grasp English sentences. The Transformer doesn't include a default method for analyzing the order of words in a sentence that it is fed. That means that we must somehow add the relative order of words into our embedding.

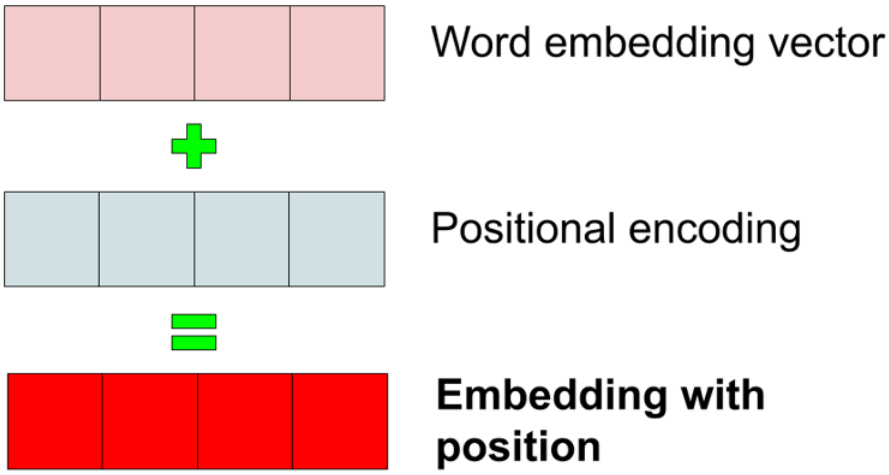
Order is of course necessary because we don't want the model to mistake "Yeah right" with "Right yeah." A mistake like that could be so catastrophic that we would get the *opposite* result of what we want from the network.

Before I just show you how positions are encoded, let me first show you the naive way that someone might do it, and how I first thought about it. I thought the picture would somehow look like this.



Here I have displayed the word embedding as the left four entries in the vector, and concatenated it with the position of the word in the sentence. I want to be very clear: this is *not* the way that it works. I just wanted to show you first the bad first pass approach before telling you the correct one, just to save you the trouble of being confused about why it doesn't work the way you imagined it in your head.

The correct way that the positional encoding works is like this.



Rather than just being a number, the positional encoding is another vector whose dimension is equal to the word embedding vector length. Then, instead of concatenating, we perform an element wise vector addition with the word embedding.

The way that we compute the positional encoding vector is a bit tricky, so bear with me as I describe the formula. In a moment, I will shed light on how this formula will allow the model to understand word order in a justified manner. We have

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Where PE stands for "positional encoding," *pos* refers to the position of the word in the sentence, and *i* stands iterator which we use to construct this vector. *i* runs from 0 to $d_{model}/2$.

Let me show you how this works by applying this formula on the example sentence.

Let's say that $d_{model} = 4$, the same dimension as I have shown in the picture above.

And further, let's encode the word "right" in the sentence "Yeah right," which is at position 1. In that case, then we will compute that the positional encoding is the following vector:

$$[\sin(pos/10000^{(2*0)/4}), \cos(pos/10000^{(2*0)/4}),$$

$$\sin(pos/10000^{(2*0)/4}), \sin(pos/10000^{(2*0)/4}))]^T$$

$$= [\sin(pos), \cos(pos), \sin(pos/100), \sin(pos/100))]^T$$

$$= [\sin(1), \cos(1), \sin(1/100), \sin(1/100))]^T$$

Why does the positional encoding use this formula? According to the paper,

> We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

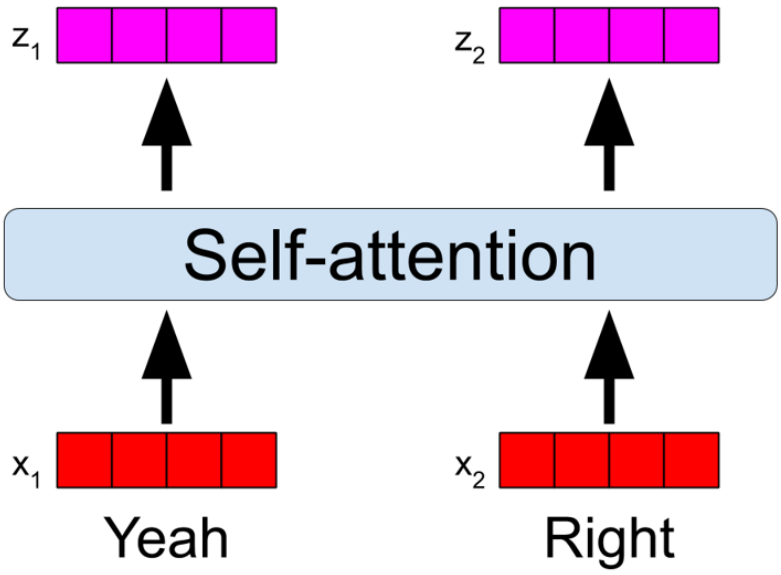
In other words, if we take the positional encoding of any word, we can easily construct the positional encoding of some other word *elsewhere in the sentence* by some linear scaling, rotation and stretching the encoding vector.

To see why this is useful, consider the fact that the word embedding allowed us to group words together which were close in meaning to each other. Now we have a way of transforming the vector in a linear fashion in order to obtain the position of the another word in the sentence. This means that we have a sort of language where a linear function translates to "the last word" or "the word 10 words ago." If we add these two encodings, the hope is that the model should be able to incorporate *both* the relative meaning of the words, and the relative positions, all in a single compact vector. And empirically, the model seems to be able to do just that.

Now that these two ways of representing the vector are combined, the words are ready to interact by being fed into the self-attention module.

Self-attention

Recall that the whole point of the Transformer architecture is that it is built on attention modules. So far we have only focused on processing the word vectors in a way that will be understood by the model. Now, we must understand what is done to these embedded vectors.



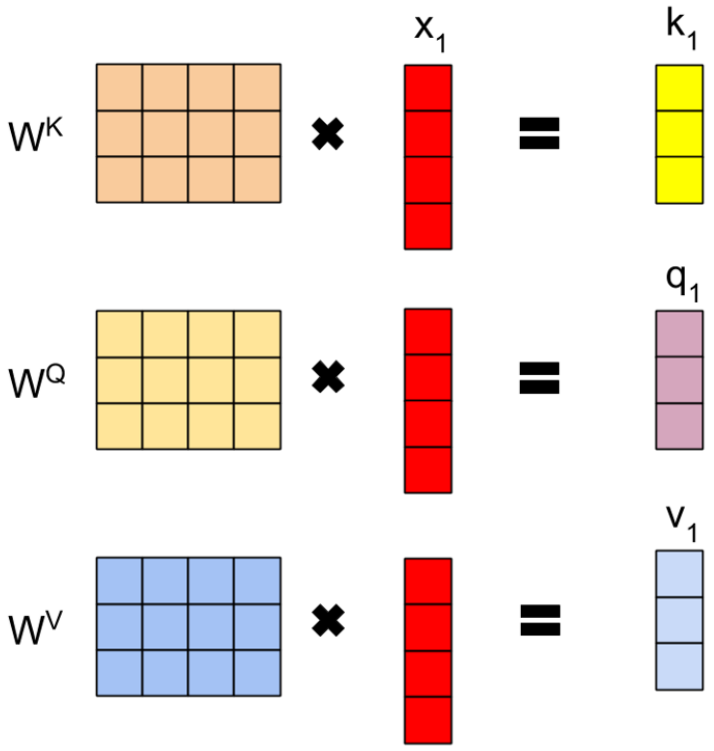
As we can see in the image above, each vector in the sentence is fed through the self attention module. In turn, the self attention module takes this sequence and returns yet another sequence, ready to be handed off to the next part of the network (shown as the vectors z_1 and z_2). This new sequence can be seen as a modified version of the original sequence, with some new information baked into each word. These new vectors contain information about how the words relate to each other.

In the last post, I described how self attention allowed the Transformer to construct links between words, which are intended to represent the relationship between words. However, it still isn't clear exactly how we find the strength of these links.

In order to create the links, we first must understand the idea of **keys**, **queries** and **values**. I will borrow the analogy that nostalgebraist uses in their explanation of the Transformer. You can imagine keys as vectors which represent a dating profile for a word, which includes information that might be useful for connecting them to other words. Queries, by contrast, are vectors which include information about what each word is looking for in each other word. Values are vectors which contain some more information about the word that isn't already represented. Our job is to use these keys and queries, which are learned for each word, in order to construct a table of matches. Each link in the table represents the relationship status between the words. Once we have these links, the strength of these links allow us to weight the words by their values. By doing this, each word sort of becomes an infusion of all the words they are dating, which could include themselves.

That is quite a lot to take in. Luckily, I will go through the process of constructing keys, queries and values, and the appropriate weighting, step by step.

The first thing to understand is that the keys and queries are not explicitly represented in the Transformer. Instead keys, queries, and values are constructed by multiplying the embedded vectors by a matrix of weights, which *is* directly learned. This is how I visualize the process.



Here, I have repeated the input vector for the first word three times, once for each computation. It is multiplied by three *separate* matrices, in order to obtain the keys, queries and values. W^K represents the weights for generating the keys. W^Q is the matrix for queries, and W^V is the matrix for generating values.

Once we have our keys, queries, and values for each word, we are ready to link each word up, or in the analogy, find their dates.

Since the dot product can be viewed as a way of measuring the similarity between vectors, the way that we check whether two words are compatible is by calculating the inner product between their keys and values. This dot product will represent the score, or compatibility between two words. In particular, the score between the first word and the second word is the dot product $q_1 \cdot k_2$. If we were to consider the general case, then the score between word *n* and word *m* would be $q_n \cdot k_m$. Remember, these scores are not symmetric. This asymmetry is due to the fact that the score from word *n* to word *m* is calculated differently than the score from word *m* to word *n*. By analogy, just because one words likes another word doesn't mean that they are liked back!

Once we have the scores from every word to every other word, we now must incorporate that information into the model. The way we do this is the following:

First, we divide the scores by the square root of the dimension of the key. Then we take a softmax over the scores for each word, so that they are all positive and add up to one. Then, for each word, we sum over the values of every other word, weighted by

the softmax scores. Then we are done. Here is what that looks like for the first word in our sentence.

	Yeah	Right
Obtain score	$q_1 \cdot k_1 = 10$	$q_1 \cdot k_2 = 5$
Divide by sqrt(d_k)	10 / sqrt(3)	5 / sqrt(3)
Softmax	0.95	0.05
Multiply by value	$0.95 * v_1$	$0.05 * v_2$
Sum over values	$0.95 * v_1 + 0.05 * v_2$	

The intuitive justifications for each of these steps may not first be apparent. The most confusing step, at least to me at first, was why we needed to divide the score by the square root of d_k. According to the paper, this makes gradients easier to work with.

The reason is because for large values of d_k , the dot products can become so large that the softmax function ends up having extremely small gradients.

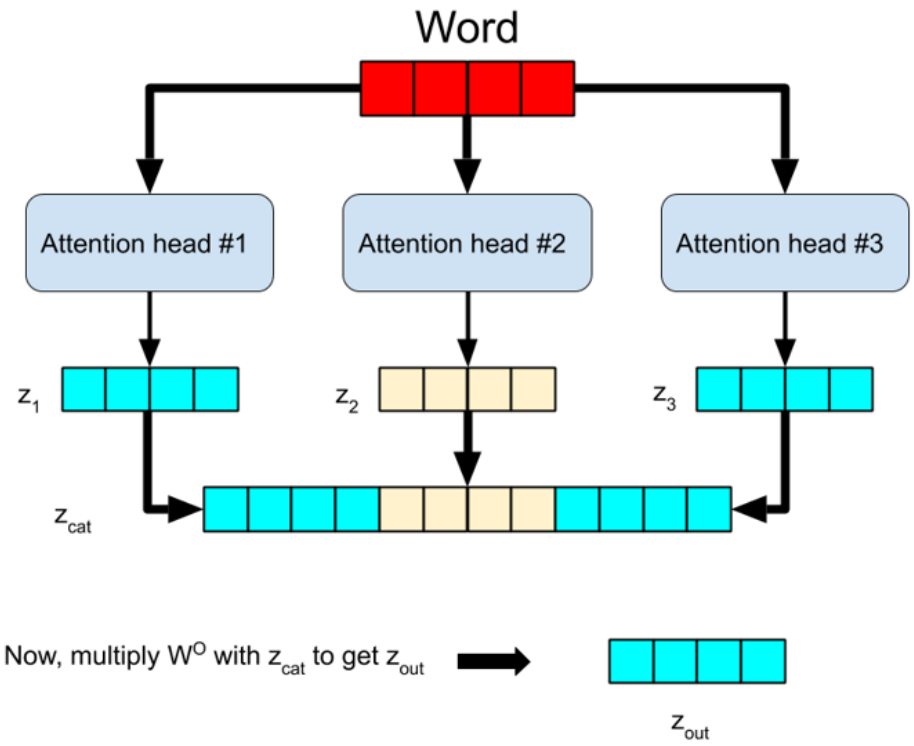
Once we have these weighted values, we are ready to pass them off to the next part of the Transformer. But not so fast! I temporarily skipped over a key component to the Transformer, and what allows it to be so powerful.

The step I missed was that I didn't tell you that that there's actually *multiple* keys, queries, and values for each word. This allows the model to have a whole bunch of different ways to represent the connections between words. Let me show you how it works.

Instead of thinking about finding the sum over values for each word, we should instead think about multiple, parallel processes each computing different sums over values. Each *attention head* computes a value (the weighted sum), using weight matrices which were all initialized differently. Then when all of the attention heads have calculated their values, we combine these into a single value which summarizes all the contextual information that each attention head was able to find.

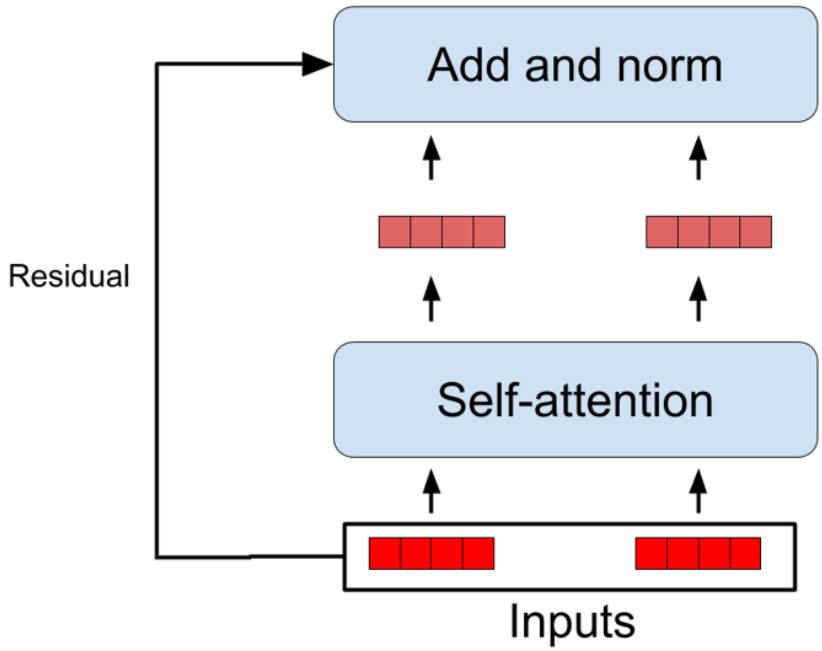
In order to combine the values found by the attention heads, we concatenate each of the values, and multiply the concatenated matrix by another learned matrix. We call

this new learned matrix W^O for the output weights. Once completed, we now have the final output word in this attention mechanism. This process looks like this.



Add and norm

Now, it looks like we're ready to get to the next part of the encoder. But before we do so, I must point out that I left something out of my initial picture. Before we can put z_{out} into the feed forward neural network, we must apply a residual connection first. What is a residual connection? It looks a bit like this.



In other words, the inputs are fed into the output of the self-attention module. The way that this works is by implementing a layer normalization step to the sum of the outputs and inputs. If you don't know what layer normalization is, don't worry. Layer normalization is an operation based on batch normalization, intended to reduce some of batch normalization's side effects. Tomorrow's post will cover batch normalization in detail, so the exact specification is omitted here.

Once we have added and normed, we are finally ready to push these words onto the neural network. But first, a note on notation.

Notation

This will likely come as no surprise, but the way that the above operations are performed are above the level of vectors. In order to speed up computation and simplify the way we write the algorithm, we really have to put all the steps into higher order arrays. This notation can indicate which operations are done in parallel.

In practice, this means that instead of writing the multi-head attention as a bunch of simultaneous steps, we simplify the whole routine by writing the following expression.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Here, the matrices Q, K, and V stand for the matrices of queries, keys and values respectively. The *rows* in the matrices are the vectors of queries, keys and values, which we have already discussed above. It should be understood that any step which allows for some form of parallelization and simplification, a matrix representation is preferred to vectors.

The feed forward neural network

We are almost done with one slice of an encoder. Even though most of the time was spent on the previous step (the attention and norm sublayer), this next one is still quite important. It is merely simpler to understand. Once we have the words generated by the self attention module, and it is handed to the layer-norm step, we must now put each word through a feed forward neural network.

This is an important point, so it's important not to miss: the same neural network is applied independently to each of the words produced by the previous step. There are many neural networks at this slice of the encoder, but they all share weights. On the other hand, neural networks *between* layers do not share weights. This allows the words to solidify their meaning from some non-linearity in the neural network before being shipped off to the next step, which is an identical slice of the encoder.

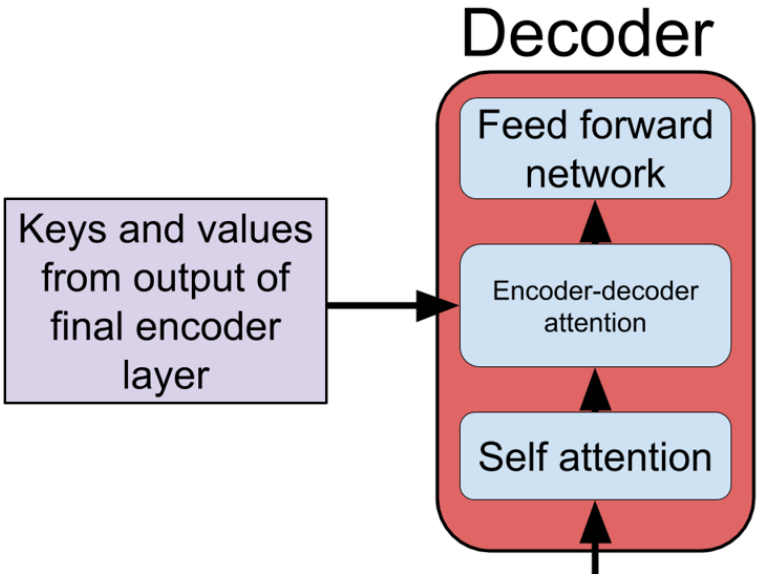
The stack

As mentioned before, the encoder and decoder are made up of a stack of attention mechanism/neural network modules. Words are passed through the bottom of a single slice of this stack, and words appear out the top. The *same* number of words are outputted from the encoder stack as inputted from the encoder stack. In other words, the vectors pass through the layers in the same shape that they started. In this sense, you can trace a single word through the stack, going up from the encoder.

Encoder-decoder crossover

At this point, the encoder is pretty much fully explained. And with that, the decoder is *nearly* explained, since the two share most properties. If you have been following this far, you should be able to trace how an embedded sentence is able to go through an encoder, by passing through many layers, each of which have a self attention component followed by a feed-forward neural network. The way that data travels through a decoder is similar, but we still must discuss how the encoder and decoder interact, and how the output is produced.

I said in the previous post that when the encoder produces its output, it somehow is able to give information to each of the layers of the decoder. This information is incorporated into the encoder-decoder attention module, which is the main difference between the encoder and decoder. The exact mechanism for how this happens is relatively simple. Take a look at the decoder structure one more time.



I have added the box on the left to clarify how the decoder takes in the input from the encoder. In the encoder-decoder attention sublayer, the keys and values are inherited from the output of the last encoder layer. These keys and values are computed the same way that we computed keys and values above: via a matrix of weights which act on the words. In this case, just as in the last, there is a *different* matrix acting on each layer, even as these matrices act on the *same* set of words: the final output of the encoder. This extra module allows the decoder to attend to all of the words in the input sequence.

In line with the previous description, the *values* in the encoder-decoder attention sublayer are derived from the outputs of the self-attention layer below it, which works exactly the same as self-attention worked in the encoder.

In the first pass through the network, the words "Yeah right" are passed through the encoder, and two words appear at the top. Then we use the decoder to successively work from there, iteratively applying self attention to an input (which input? this will be elaborated in just a moment) and plugging its output into the encoder-decoder attention. The encoder then attends to the output from the encoder, along with the output from self-attention. This is then fed into a neural network step, before being handed off to the *next* decoder in the stack. At the top of the decoder, we feed the output one last time through a linear layer and a softmax layer, in order to produce a single word, the output of the network on a single step.

Then, after a single word is produced, we put that back into the input of the decoder and run the decoder again, now with its ouput being parsed as input. We stop whenever the decoder outputs a special symbol indicating that it has reached the end of its translation, the *end* token. This process would look end like this.

That’s not true <end>

Before the decoder has produced an output, we don't want it to be able to attend to tokens which have not been seen yet. Therefore, in our *first* pass through the decoder,

we set all the softmax inputs in self-attention to $-\infty$. Recall that when the softmax is applied, we use the resulting output to weight the values in the self-attention module. If we set these inputs to the softmax to $-\infty$, the result is that the softmax outputs a zero, which correspondingly gives those values no weight. In *future* passes, we ensure that every input that corresponds to a future word is treated similarly, so that there cannot be connections between words that we have not outputted yet.[1]

Now, with the mechanism for producing outputs fully elucidated, we are ready to stare at the full network, with all of its tiny details. I do not want to compete with the original paper for illustrations on this one. Therefore, I have incorporated figure 1 from the paper and placed it here.

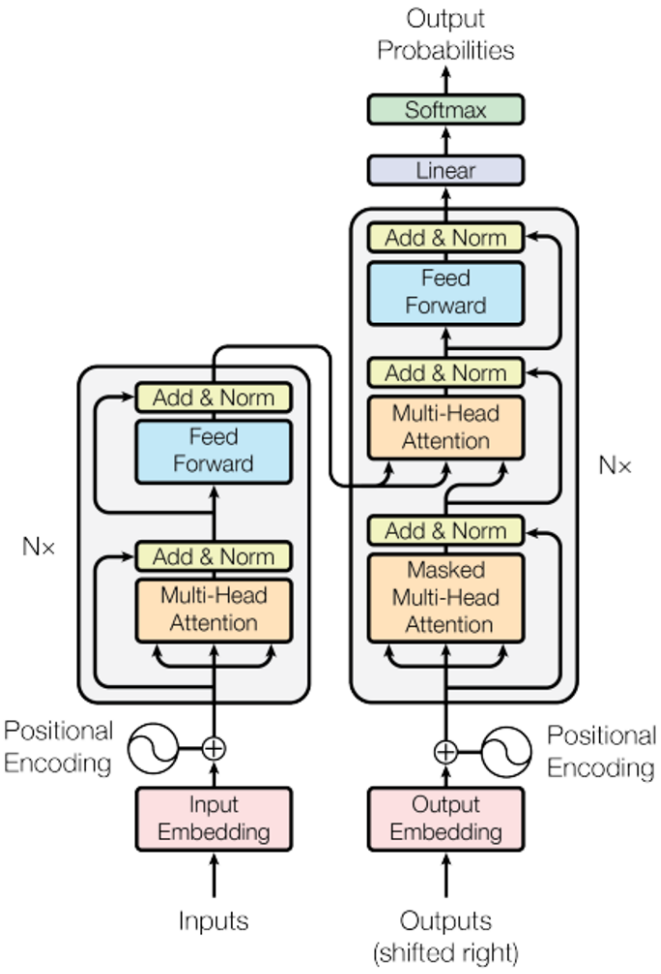


Figure 1: The Transformer - model architecture.

As you might have noticed, I neglected to tell you about some extra residual connections. Regardless, they work the same way that I described above, so there should be no confusion there! Other than that, I have covered pretty much part of the Transformer design. You can now gaze your eyes upon the above image and feel the

full satisfaction of being able to understand every part (if you've been paying *attention*).

Of course, we haven't actually gone into how it is trained, or how we generate the single words from the final softmax layer and output probabilities. But this post is long enough, and such details are not specific to the Transformer anyway. Perhaps one day I shall return.

For now, I will be leaving the transformer architecture. In tomorrow's post I will cover batch normalization, another one of those relatively new ML concepts that you should know about. See you then.

---

[1] If this discussion of exactly how the decoder incorporates its own output, and masks illegal connections confuses you, you're not alone. I am not confident that these paragraphs are correct, and will possibly correct them in the future when I have a better understanding of how it is implemented.

Understanding Batch Normalization

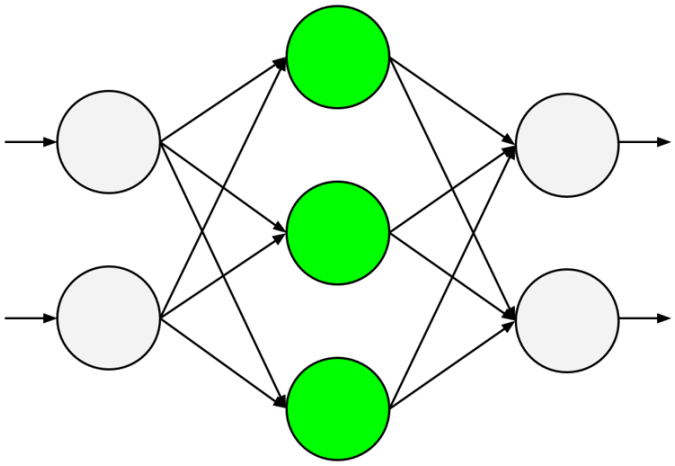
Batch normalization is a technique which has been successfully applied to neural networks ever since it was introduced in 2015. Empirically, it decreases training time and helps maintain the stability of deep neural networks. For that reason practitioners have adopted the technique as part of the standard toolbox.

However, while the performance boosts produced by using the method are indisputable, the underlying reason *why* batch normalization works has generated some controversy.

In this post, I will explore batch normalization and will outline the steps of how to apply it to an artificial neural network. I will cover what researchers initially suspected were the reasons why the method works. Tomorrow's post will investigate new research which calls these old hypotheses into question.

To put it in just a few sentences, batch normalization is a transformation that we can apply at each layer of a neural network. It involves normalizing the input of a layer by dividing the layer input by the activation standard deviations and subtracting the activation mean. After batch normalization is applied it is recommended to apply an additional transformation to the layer with learned parameters which allow the neural network to learn useful representations of the input. All of these steps are then incorporated into the backpropagation algorithm.

The mechanics of batch normalization can be better understood with an example. Here, I have illustrated a simple feed-forward neural network. Our goal is to apply batch normalization to the hidden layer, indicated in green.



Let the vector h stand for the input to the hidden layer. The input h is processed in the layer by applying an activation function element-wise to the vector computed from the previous layer. Let H stand for a mini-batch of activations for the hidden layer, each row corresponding to one example in the mini-batch.

What batch normalization does is subtract the activation unit mean value from each input to the hidden layer, and divides this expression by the activation unit standard deviation. For a single unit, we replace h_i with

$$h_i' = (h_i - \mu_i)/\sigma_i$$

where μ_i is the mean input value for h_i across the mini-batches. In symbolic form, in order to calculate μ_i we compute

$$\mu_i = \frac{1}{m} \sum_j H_{j,:}$$

similarly, we calculate σ_i by computing

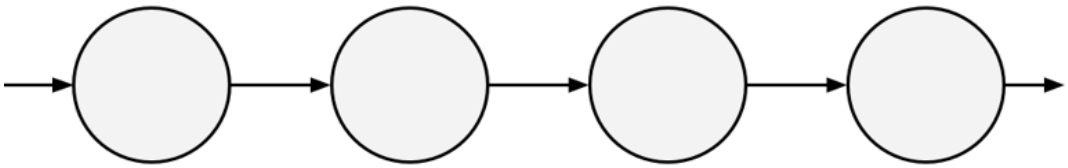
$$\sigma_i = \sqrt{\delta + \frac{1}{m} \sum_i (H - \mu)_i^2}$$

The above expression is the standard deviation for the input to the ith activation unit with an additional constant value δ. This delta component is kept at a small positive value, like 10^{-8}, and is added only to avoid the gradient becoming undefined where the true standard deviation is zero.

At test time, we can simply use the running averages for μ and σ discovered during training, as mini-batch samples will not always be available.

The above computations put the distribution of the input to a layer into a regime where the gradients for each layer are all reasonably sized. This is useful for training because we don't want the gradient descent step to vanish or blow up. Batch normalization accomplishes this because the weights no longer have an incentive to grow to extremely large or small values. In the process, batch normalization therefore also increases our ability to train with activations like the sigmoid, which were previously known to fall victim to vanishing gradients.

I will borrow an example from the Deep Learning Book (section 8.7.1) to illustrate the central issue, and how we can use batch normalization to fix it. Consider a simple neural network consisting of one neuron per layer. We denote the length of this network by l.



Imagine that we designed this neural network such that it did not have an activation function at each step. If we chose this implementation then the output would be $\hat{y} = xw_1w_2w_3 \dots w_l$. Suppose we were to subtract a gradient vector g obtained in the course of learning. The new value for $\hat{y}$ will now be $\hat{y} = x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l)$. Due to the potential depth of this neural network, the gradient descent step could now have altered the function in a disastrous way. If we expand the product for the updated $\hat{y}$ expression, we find that there are *n*-order terms which could blow up if they are too large. One of these *n*-order terms is $\epsilon g_1 \prod_{i=2}^{l} w_i$. This expression is now subtracted from $\hat{y}$, which can cause an issue. In particular, if the terms w_i from i = 2 to i = l are all greater than one, then this previous expression becomes exponentially large.

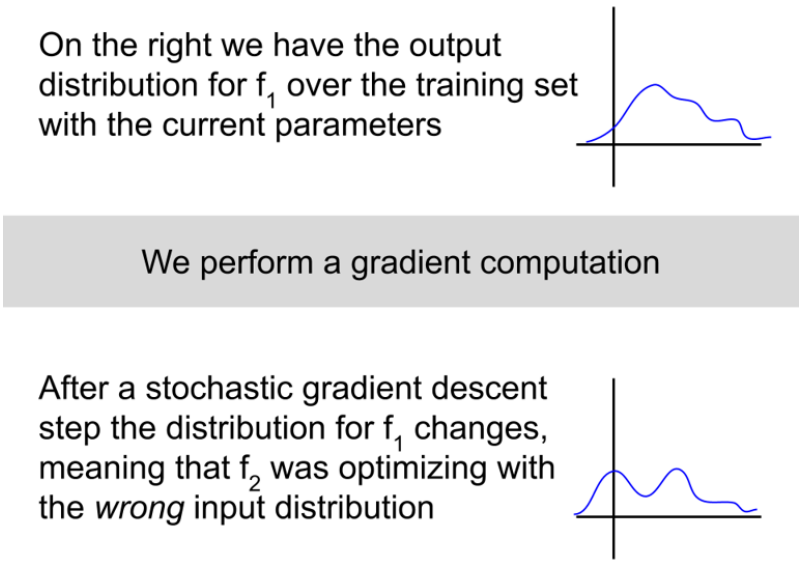
Since a small mistake in choosing the learning rate can result in an exponential blow up, we must choose the the rate at which we propagate updates wisely. And since this network is so deep, the effects of an update to one layer may dramatically affect the other layers. For example, whereas an appropriate learning rate at one layer might be 0.001, this might simultaneously cause a vanishing gradient at some another layer!

Previous approaches to dealing with this problem focused on adjusting ϵ at each layer in order to ensure that the effect of the gradient was small enough to cancel out the large product, while remaining large enough to learn something useful. In practice, this is quite a difficult problem. The *n*-order terms which affect the output are too numerous for any reasonably quick model to take into account all of them. By using this technique, the only options we have left are to shrink the model so that there are few layers, or to slow down our gradient computation excessively.

The above difficulty of coordinating gradients between layers is really a specific case of a more general issue which arises in deep neural networks. In particular, the issue is termed an *internal covariate shift* by the original paper. In general a *covariate shift* refers to a scenario in which the input distribution for some machine learning model changes. Covariate shifts are extremely important to understand for machine learning because it is difficult to create a model which can generalize beyond the input distribution that it was trained on. Internal covariate shifts are covariate shifts that happen within the model itself.

Since neural networks can be described as function compositions, we can write a two layer feedforward neural network as $f_2(f_1(x, \theta_1), \theta_2)$ where x is the input to the network, and θ defines the parameters at each layer. Writing this expression where $u = f_1(x, \theta_1)$ we obtain $f_2(u, \theta_2)$. We can see, therefore, that the final layer of the network has an input distribution defined by the output of the first layer, $f_1(x, \theta_1)$. Whenever the parameters θ_1 and θ_2 are modified simultaneously, then f_2 has

experienced an internal covariate shift. This shift is due to the fact that f_1 now has a different output distribution.



It's as if after being told how to change in response to its input distribution, the very ground under f_2's feet has changed. This has the effect of partially canceling out assumption of we are making about the gradient, which is that each element of the gradient is defined as the the rate of change of a parameter *with everything else held constant*. Gradients are only defined as measuring some slope over an infinitesimal region of space — and in our case, we are only estimating the gradient using stochastic mini-batch descent. This implies that we should automatically assume that this basic assumption for our gradient estimate will be false in practice. Even still, a difference in the way that we approach the gradient calculation can help alleviate this problem to a large degree.

One way to alleviate the issue would be to encourage each layer to output similar distributions across training steps. For instance, we *could* try to add a penalty to the loss function to encourage the activations from each layer to more closely resemble a Gaussian distribution, in particular the *same* Gaussian at each step during the training process, like a whitened distribution. This would have the intended effect of keeping the underlying distribution of each layer roughly similar across training steps, minimizing the downsides of internal covariate shift. However this is an unnecessarily painful approach, since it is difficult to design a loss penalty which results in the exact desired change.

Another alternative is to modify the parameters of a layer after each gradient descent step in order to point them in a direction that will cause their output to be more Gaussian. Experiments attempting this technique resulted in neural networks that would waste time repeatedly proposing an internal covariate shift only to be reset by the intervention immediately thereafter (see section 2 in the paper).

The solution that the field of deep learning has settled on is roughly to use batch normalization as described above, and to take the gradients while carefully taking into

account these equations. The batch normalization directly causes the input activations to resemble a Gaussian, and since we are using backpropagation through these equations, we don't need any expensive tug of war with the parameters.

One more step is however needed in order to keep the layers from losing their representation abilities after having their output distributions normalized.

Once we have obtained the batch of normalized activations H', we in fact use $\gamma H' + \beta$ as the input for the layer, where γ and β are learned scalar parameters. In total, we are applying a transform at each layer, the *Batch Normalizing Transform*. If you consider a layer computation defined by $z = g(Wu + b)$ where g is an activation function, u is the output from the previous layer, W is the weight matrix, and b is the bias term, then the layer now becomes written as $z = g(BT(Wu))$ where BT is the batch transformation defined by $BT(h) = \gamma h' + \beta$. The bias term is removed because the distribution shift is now fully defined by β.

It may seem paradoxical that after normalizing H we would now alter the matrix to make its standard deviation $\neq 1$ and its mean $\neq 0$. Didn't we want to minimize the effect of a covariate shift? However, this new step allows more freedom in the way the input activations can be represented. From the paper,

> Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity.

With the new learned parameters, the layers are more expressive. With this additional parameterization the model can figure out the appropriate mean and standard deviation for the input distribution, rather than having it set to a single value automatically, or worse, having it be some arbitrary value characterized by the layers that came before.

With these properties batch normalization strikes a balance between minimizing the internal covariate shift in the neural network while keeping the representation power at each layer. All we need to do is incorporate the new learned parameters, compute the gradients via a new set of backpropagation equations and apply the normalization transformation at each layer. Now we hit run and our neural network trains faster, and better. It's that easy.

Did my explanation above make perfect sense? Is internal covariate shift really the big issue that batch normalization solves? Tomorrow I will investigate potential flaws with the reasons I gave above.

Tune in to find out what those issues might be. Or just read the paper I'll be summarizing.

Rethinking Batch Normalization

Yesterday we saw a glimpse into the inner workings of batch normalization, a popular technique in the field of deep learning. Given that the effectiveness of batch normalization has been demonstrated beyond any reasonable doubt, it may come as a surprise that researchers don't really know how it works. At the very least, we sure didn't know how it worked when the idea was *first* proposed.

One might first consider that last statement to be unlikely. In the last post I outlined a relatively simple theoretical framework for explaining the success of batch normalization. The idea is that batch normalization reduces the internal covariate shift (ICS) of layers in a network. In turn, we have a neural network that is more stable, and robust to large learning rates, and allows much quicker training.

And this was the standard story in the field for years, until a few researchers decided to actually investigate it.

Here, I hope to convince you that the theory really is wrong. While I'm fully prepared to make additional epistemic shifts on this question in the future, I also fully expect to never shift my opinion *back*.

When I first read the original batch normalization paper, I felt like I really understood the hypothesis. It felt simple enough, was reasonably descriptive, and intuitive. But I didn't get a perfect visual of what was going on — I sort of hand-waved the step where ICS contributed to an unstable gradient step. Instead I, like the paper, argued by analogy, that since controlling for covariate shifts were known for decades to help training, a technique to reduce internal covaraite shift is thus a natural extension of this concept.

It turned out this theory wasn't even a little bit right. It's not that covariate shifts aren't important *at all*, but that the entire idea is based on a false premise.

Or at least, that's the impression I got while reading Shibani Santurkar et al.'s How Does Batch Normalization Help Optimization? Whereas the original batch normalization paper gave me a sense of "I kinda sorta see how this works," this paper completely shattered my intuitions. It wasn't just the weight of the empirical evidence, or the theoretical underpinning they present; instead what won me over was the surgical precision of their rebuttal. They saw how to formalize the theory of improvement via ICS reduction and tested it on BatchNorm directly. The theory turned out to be simple, intuitive, and false.

In fairness, it wasn't laziness that prohibited researchers from reaching our current level of understanding. In the original batch normalization paper, the authors indeed proposed a test for measuring batch normalization's effect on ICS.

The problem was instead twofold: their *method* for measuring ICS was inadequate, and failed to consistently apply their proposed mechanism for how ICS reduction was supposed to work in their testing conditions. More importantly however, they *didn't even test the theory that ICS reduction contributed to performance gains*. Instead their argument was based on a simple heuristic: we *know* that covariate shifts are bad, we *think* that batch normalization reduces ICS, and we *also* know batch normalization increases performance charactersitics — therefore batch normalization works due to ICS reduction. As far as I can tell, most the articles that came after the

original paper just took this heuristic at face value, citing the paper and calling it a day.

And it's not a bad heurstic, all in all. But perhaps it's a tiny bit telling that on yesterday's post, Lesswrong user crabman was able to anticipate the true reason for batch normalization's success, defying both my post and the supposed years that it took researchers to figure this stuff out. Quoth crabman,

> I am imagining this internal covariate shift thing like this: the neural network together with its loss is a function which takes parameters θ as input and outputs a real number. Large internal covariate shift means that if we choose ε>0, perform some SGD steps, get some θ, and look at the function's graph in ε-area of θ, it doesn't really look like a plane, it's more curvy like.

In fact, the above paragraph doesn't actually describe internal covariate shift, but instead the smoothness of the loss function around some parameters θ. I concede, it is perhaps possible that this is really what the original researchers meant when they termed internal covariate shift. It is therefore also possible that this whole critique of the original theory is based on nothing but a misunderstanding.

But I'm not buying it.

Take a look at how the original paper defines ICS,

> We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training.

This definition can't merely refer to the smoothness of the gradient around θ. For example, the gradient could be extremely bumpy and have sharp edges and yet ICS could be absent. Can you think of an example of a neural network like this? Here's one: think of a network with just one layer whose loss function is some extremely contorted shape because its activation function is some crazy non-linear function. It wouldn't be smooth, but its input distribution would be constant over time, given that it's only one layer.

I can instead think of two interpretations of the above definition for ICS. The first interpretation is that ICS simply refers to the change of activations in a layer during training. The second interpretation is that this definition specifically refers to change of activations *caused* by changes in network parameters at previous layers.

This is a subtle difference, but I believe it's important to understand. The first interpretation allows ease of measurement, since we can simply plot the mean and variance of the input distributions of a layer during training. This is in fact how the paper (section 4.1) tests batch normalization's effect on ICS. But really, the second interpretation sounds closer to the hypothesized mechanism for how ICS was supposed to work in the first place.

On the level of experimentation, the crucial part of the above definition is the part that says "change [...] *due* to the change in network parameters." Merely measuring the change in network parameters *over time* is insufficient. Why? Because the hypothesis was that if activation distributions change too quickly, then a layer will have its gradient pushed into a vanishing or exploding region. In the first interpretation, a change over time could still be slow enough for each layer to adapt

appropriately. Therefore, we need additional information to discover whether ICS is occurring in the way that is described.

To measure ICS under the second interpretation, we have to measure the counterfactual change of parameters — in other words, the amount that the network activations change as a result of other parameters being altered. And we also need a way of seeing whether the gradient is being pushed into extreme regions as a result of these parameters being changed. Only then can we see whether this *particular* phenomenon is actually occurring.

The newer paper comes down heavily in favor of this interpretation, and adds a level of formalization on top of it. Their definition focuses on measuring the difference between two different gradients: one gradient with all of the previous layers altered by back propagation, and one gradient where all of the previous layers have been unaltered. Specifically, let L by a loss function for a neural network of k layers. Then,

their definition of ICS for the activation i and time t is $||G_{t,i} - G'_{t,i}||_2$ where

$$G_{t,i} = \nabla_{W_i^{(t)}} L(W_1^{(t)}, \ldots, W_k^{(t)}; x^{(t)}, y^{(t)})$$

$$G'_{t,i} = \nabla_{W_i^{(t)}} L(W_1^{(t+1)}, \ldots, W_{i-1}^{(t+1)}, W_i^{(t)}, W_{i+1}^{(t)}, \ldots, W_k^{(t)}; x^{(t)}, y^{(t)}))$$

and $(x^{(t)}, y^{(t)})$ is the batch of input-label pairs to train the network at time t.

The first thing to note about this definition is that it allows a clear, precise measurement of ICS, which is based solely on the change of the gradient due to shifting parameters beneath a layer during backpropagation.

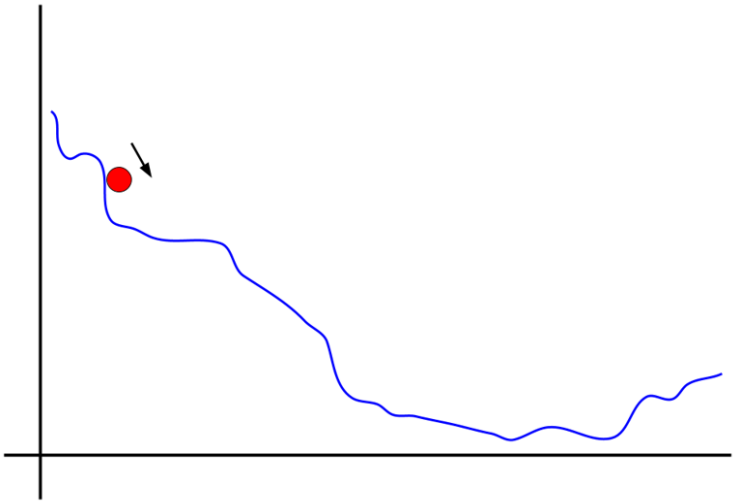
What Shibani Santurkar et al. found when they applied this definition was a bit shocking. Not only did batch normalization fail to decrease ICS, in some cases it even increased it when compared to naive feedforward neural networks. And to top that off, they found that even in networks where they artificially increased ICS, performance barely suffered.

In one experiment they applied batch normalization to each hidden layer in a neural network, and at each step, they added noise after the batch normalization transform in order to induce ICS. This noise wasn't just Gaussian noise either. Instead they chose the noise such that it was a *different* Gaussian at every time step and every layer, such that the Gaussian parameters (specifically mean and variance) varied according to a yet another meta Gaussian distribution. What they discovered was that even though this increased measured ICS dramatically, the time it took to train the networks to the baseline accuracy was almost identical to regular batch normalization.

And remember that batch normalization actually does work. In all of the experiments for mere performance increases, batch normalization has passed the tests with flying colors. So clearly, since batch normalization works, it must be for a different reason

than simply reducing ICS. But that leaves one question remaining: how on Earth does it work?

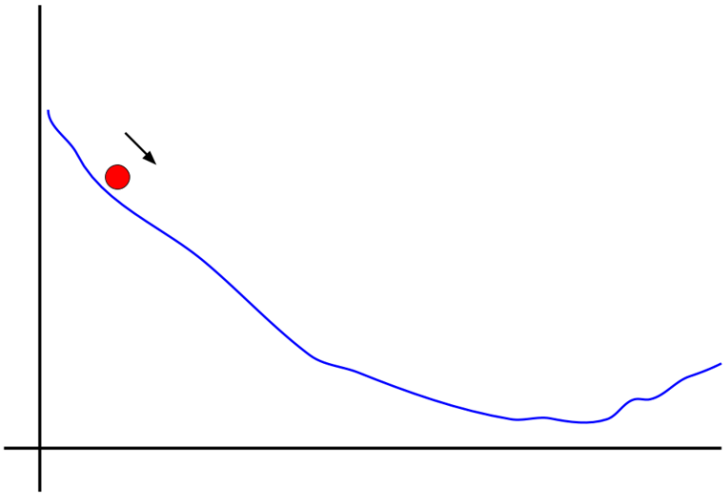
I have already hinted at the reason above. The answer lies in something even simpler to understand than ICS. Take a look at this plot.



Imagine the red ball is rolling down this slope, applying gradient descent at each step. And consider for a second that the red ball isn't using any momentum. It simply looks at each step which direction to move and moves in that direction in proportion to the slope at that point.

A problem immediately arises. Depending on how we choose our learning rate, the red ball could end up getting stuck almost immediately. If the learning rate is too slow, then it will probably get stuck on the flat plane to the right of it. And in practice, if its learning rate is too high, then it might move over to another valley entirely, getting itself into an exploding region.

The way that batch normalization helps is by changing the loss landscape from this bumpy shape into one more like this.



Now it no longer matters that much what we set the learning rate to. The ball will be able to find its way down even if its too small. What used to be a flat plane has now been rounded out such that the ball will roll right down.

The specific way that the paper measures this hypothesis is by applying pretty standard ideas from the real analysis toolkit. In particular, the researchers attempted to measure the Lipschitzness of the loss function around the parameters θ for various types of deep networks (both empirically and theoretically). Formally a function is L-Lipschitz if $|f(x_1) - f(x_2)| \leq L||x_1 - x_2||$ for all x_1 and x_2. Intuitively, this is a measure of how smooth the function is. The smaller the constant L, the function has fewer and less extreme jumps over small intervals in some direction.

This way of thinking about the smoothness of the loss function has the advantage of including a rather natural interpretation. One can imagine that the magnitude of some gradient estimate is a prediction of how much we expect the function to fall if we move in that direction. We can then evaluate how good we are at making predictions across different neural network schemes and across training steps. When gradient predictiveness was tested, there were no surprises — the networks with batch normalization had the most predictive gradients.

Perhaps even more damning is that not only did the loss function become more smooth, the gradient landscape itself became more smooth, a property known as $\beta-$ smoothness. This had the effect of not only making the gradients more predictive of the loss, but the gradients themselves were easier to predict in a certain sense — they were fairly consistent throughout training.

Perhaps the way that batch normalization works is by simply smoothing out the loss function. At each layer we are just applying some normalizing transformation which helps remove extreme points in the loss function. This has the additional prediction that other transformation schemes will work just as well, which is exactly what the researchers found. Before, the fact that we added some parameters γ and β was

confusing, since it wasn't clear how this contributed to ICS reduction. Now, we can see that ICS reduction shouldn't even be the goal, perhaps shedding light on why this works.

In fact, there was pretty much nothing special with the exact way that batch normalization transforms the input, other than the properties that contribute to smoothness. And given that so many more methods have now come out which build on batch normalization despite using quite different operations, isn't this exactly what we would expect?

Is this the way batch normalization really works? I'm no expert, but I found this interpretation much easier to understand, and also a much simpler hypothesis. Maybe we should apply Occam's razor here. I certainly did.

In light of this discussion, it's also worth reflecting once again that the argument "We are going to be building the AI so of course we'll understand how it works" is not a very good one. Clearly the field can stumble on solutions that *work*, and yet the reason why they work can remain almost completely unknown for years, even when the answer is hiding in plain sight. I honestly can't say for certain whether happens a lot, or too much. I only have my one example here.

In the next post, I'll be taking a step back from neural network techniques to analyze generalization in machine learning models. I will briefly cover the basics of statistical learning theory and will then move to a framing of learning theory in light of recent deep learning progress. This will give us a new test bed to see if old theories can adequately adapt to new techniques. What I find might surprise you.

A Survey of Early Impact Measures

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

In the context of AI alignment an impact penalty is one way of avoiding large negative side effects from misalignment. The idea is that rather than specifying negative impacts, we can try to avoid catastrophes by avoiding large side effects altogether.

Impact measures are ways to map a policy to a number which is intended to correspond to "how big of an impact will this action have on the world?" Using an impact measure, we can regularize any system with a lot of optimization power by adding an impact term to its utility function.

This post records and summarizes much of the early research on impact penalties. I emphasize the aims of each work, and the problems associated with each approach, and add occasional commentary along the way. In the next post I will dive more deeply into recent research which, at least in my opinion, is much more promising.

The mathematics of reduced impact: help needed, by Stuart Armstrong (2012)

This is the first published work I could find which put forward explicit suggestions for impact measures and defined research directions. Armstrong proposed various ways that we could measure the difference between worlds, incorporate this information into a probability distribution, and then use that to compare actions.

Notably, this post put a lot of emphasis on comparing specific ontology-dependent variables between worlds in a way that is highly sensitive to our representation. This framing of low impact shows up in pretty much all of the early writings on impact measures.

One example of an impact measure is the "Twenty (million) questions" approach, where humans define a vector of variables like "GDP" and "the quantity of pesticides used for growing strawberries." We could theoretically add some L_1 regularizer to the utility function, which measures the impact difference between a proposed action and the null action, scaled by a constant factor. The AI would then be incentivized to keep these variables as close possible to what they would have been in the counterfactual where the AI had never done anything at all.

The flaw with this specific approach was immediately pointed out, both by Armstrong, and in the comments below. Eliezer Yudkowsky objected to the approach since it implied that some artificial intelligence would try to manage the state of affairs of the entire universe in order to keep the state of the world to be identical to the counterfactual where the AI never existed,

> Coarse-grained impact measures end with the AI deploying massive-scale nanotech in order to try and cancel out butterfly effects and force the world onto a coarse-grained path as close as possible to what it would've had if the AI "hadn't existed" however that counterfactual was defined. [...] giving an AI a huge penalty function over the world to minimize seems like an obvious recipe for building something that will exert lots and lots of power.

I agree that this is an issue with the way that the impact measure was defined — in particular, the way that it depended on some metric comparing worlds. However, the last line sounds a bit silly to me. If your impact measure provides an artificial intelligence an incentive to "exert lots and lots of power" then it doesn't really sound like an impact measure at all.

This critique is expanded in Yudkowsky's Arbital article which came out four years later, which I also go into below.

Low Impact Artificial Intelligences, by Stuart Armstrong and Benjamin Levinstein (2015)

This work starts from fundamentally the same starting point as the Lesswrong post above, but expands upon it by having more explicit and rigorous approaches.

One approach outlined in the article is the approach of viewing impact as a form of news which informs us of the world that we are living in.

The idea is simple enough: if you were notified tomorrow that Deepmind had constructed an advanced superintelligence, this would rightfully change the way that you thought about the world, and would inform you that some radical things are soon arriving which could alter the trajectory of what we value. Putting an emphasis on importance and the value of information forms the intuition behind this method of measuring impact. This also enables us to look at importance by putting us in the shoes of an agent which is given information.

> How would we measure importance? One approach is to look at a large set U of different utility functions, and check that knowing X makes little expected difference for any of them.

In other words, conditional on the action taking place, how does this affect utility?

A concept which neighbors this approach is to imagine that impact is a measure of how detectable an action is. Presumably, the exact way that hydrogen atoms are distributed throughout the center of the sun shouldn't affect the impact measure very much, since their arrangement gives us pretty much no actionable information here on Earth. Consequently, knowing a lot about the particular way that atoms in the sun are arranged doesn't change what we predict will happen in the world — at least unless you are so powerful that it allows you to simulate the entire universe flawlessly, and feeds into the final input into this simulation.

By shifting the focus from particle positions and state representations, this paper gets closer to the way that I personally interpret impact measurement, and how I understand impact is defined in more recent research.

The paper moves then towards describing very abstract ways of measuring impact, including measuring general changes in probability distributions, such as some generalized cross-entropy between two probability distributions. The intuition here is that worlds with a low-impact artificial intelligence will have a broad probability distribution, whereas worlds with high-impact artificial intelligence will have almost all the probability density on a specific narrow region. If we could somehow measure the divergence between these probability distributions in a way that was natural and was resilient to the choice of representation, this would provide a way of measuring impact that clearly isn't value laden.

The weakness of the abstract approach is that there is no clear interpretation of what's going on. The choice of how we compare probability distributions seems a bit arbitrary, and it also doesn't seem like the type of thing a human would care about if they were naively minimizing impact. Furthermore, this approach, like the one that came before, requires some baseline weak prediction capability in order for it to be applied consistently. To see why, consider that a sufficiently advanced superintelligence will always have essentially all of its probability distribution on a single future — the actual future.

Armstrong and Levinstein wrote a brief discussion for how we can calibrate the impact measure. The way that machine learning practitioners have traditionally calibrated regularizers is by measuring their effect on validation accuracy. After plotting validation accuracy against some μ scaling factor, practitioners settle on the value which allows their model to generalize the best. In AI alignment we must take a different approach, since it would be dangerous to experiment with small scaling values for the impact penalty without an idea of the magnitude of the measurement.

The paper points to an additional issue: if the impact measure has sharp, discontinuous increases, then calibrating the impact measure may be like balancing a pen on the tip of a finger.

> It is conceivable that we spend a million steps reducing μ through the ‘do nothing’ range, and that the next step moves over the ‘safe increase of u’, straight to the ‘dangerous impact’ area.

This above problem motivates the following criterion for impact measures. An impact measure should scale roughly linearly in the measurement of impact on the world. Creating one paperclip might have some effect X on the world, and creating two paperclips might have some effect $X + Y$ on the world, but creating three paperclips should have some effect that is close to $X + 2Y$ or else the impact measure is broken.

Since measuring impact is plausibly something that can be done without the help of a superintelligence, this provides a potential research opportunity. In other words, we can check *ex ante* whether any impact measure is robust to these types of linear increases in impact. On the other hand, if we find that some impact penalty *requires* superintelligent capabilities in order to measure, then it may cast doubt on the method of measurement, and our ability to trust it. And of course, it couldn't reflect any algorithm which humans are running in their head.

After this point in time, specifying desiderata in order to avoid failure modes becomes a shift in the way that impact measures are formulated.

Concrete Problems in AI Safety (Section 3), by Dario Amodei and Chris Olah et al. (2016)

The main contribution in this paper is that it proposes a way to learn impact. In this sense, impact is less of an actual thing that we add to the utility function, and more of something that is discovered by repeated demonstration. The intention behind this shift is to move focus away from explicit ways of representing impact, which can be brittle if we focus too much on exactly how the environment is represented. The downside is that it doesn't appear to scale to superintelligent capabilities.

If I understand this proposal correctly, an example of impact in this case would be to penalize some ML system each time it makes a large error. Over time the ML system would have an implicit penalty term for errors of that type, such that in the future it won't be very likely to do something which has a large impact on the world. Of course, if we consider that as AI systems grow in competence they are likely to try strategies which we had not even thought about before, this approach is particularly susceptible to edge instantiation.

The paper also discusses using empowerment to measure influence before immediately rejecting it as a solution. The authors consider that researching simple and already well-trodden mathematical functions could yield fruitful results, providing optimism for future research into this area.

> Despite these issues, the example of empowerment does show that simple measures (even purely information-theoretic ones!) are capable of capturing very general notions of influence on the environment.

Low impact, by Eliezer Yudkowsky (2016)

This Arbital article summarizes impact measurements more succinctly and comprehensively any previous work. It outlines the aim of impact research and provides useful critiques for impact penalties which are based on measuring the distance between state representations. In particular, the article expands on three foreseeable difficulties in an impact measure:

1. An artificial intelligence could try to minimize its influence by doing bad things that were low impact relative to the normal course of reality. An example would be an AI that tried to cure cancer but kept death rates constant, since this would have happened if we didn't cure cancer.

2. Due to the existence of chaotic systems, an AI would be indifferent to particular systems which are nearly impossible to control directly, such as the weather. In this case, since the AI is indifferent, it might as well deploy nanobots in the sky since there's not much we can do to keep weather constant.

3. The AI will end up wanting to keep everything in complete stasis, which introduces the incentive of taking over the entire world in order to keep things the way they would have turned out if the AI didn't exist. For instance, if we constructed a low impact AI in order to learn more about alignment experimentally, the low-impact AI

would want us to learn as little as possible from the experiment because every new insight we gain would be something we would not have gotten if the AI did not exist.

As I have indicated above, I think that these types of errors are quite important to consider, but I do think that impact can be framed differently in order to avoid them. In particular, there is a lot of focus on measuring the distance between worlds in some type of representation. I am skeptical that this will forever remain a problem because I don't think that humans are susceptible to this mistake, and I also think that there are agent-relative definitions of impact which are more useful to think about.

To provide one example which guides my intuitions, I would imagine that being elected president is quite impactful from an individual point of view. But when I make this judgement I don't primarily think about any particular changes to the world. Instead, my judgement of the impact of this event is focused more around the type of power I gain as president, such as being able to wield the control of the military. Conversely, being in a nuclear war is quite impactful because of how it limits our current situation. Having the power to exert influence, or being able to live a safe and happy life is altered dramatically in a world affected by nuclear war.

This idea is related to instrumental convergence, which is perhaps more evidence that there is a natural core to this concept of measuring impact. In one sense, they could be part of a bigger whole: collecting money is impactful because it allows me to do more things as I become wealthier. And indeed, there may be a better word than "impact" for the exact concept which I am imagining.

Penalizing side effects using stepwise relative reachability, by Victoria Krakovna et al. (2018)

From what I am aware, the current approaches which researchers are most optimistic about are the impact measures based on this paper. In the first version of this paper, which came out in 2018 (updated in 2019 for attainable utility), the authors define relative reachability and compare it against a baseline state, which is also defined. I will explore this paper, and the impact measures which are derivative to this one in the next post.

In the last post in this sequence, I promised to "cover the basics of statistical learning theory." Despite the ease of writing those words, I found it to be much more difficult than I first imagined, delaying me a few days. In the meantime, I will focus the next post on surveying recent impact research.

Understanding Recent Impact Measures

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

In the first five years after Stuart Armstrong posted his first research suggestions for impact measures, very little published work expanded on the idea. The last post in this sequence was intended to somewhat comprehensively review this literature, but it surveyed only four papers total, including the original article.

In the last two years, research has now picked up pace by a significant margin. The two papers which are most significant are Penalizing side effects using stepwise relative reachability by Victoria Krakovna et al. and Conservative Agency by Alexander Turner et al. In that time a few blog posts have come out explaining the approaches in more detail, and public debate over the utility of impact measures has become much more visible.

Here I will briefly explain the two most prominent measures, relative reachability and attainable utility. We will see that they diverge conceptually from earlier research. By being different, they also end up satisfying some desirable properties. I will then consider some recent notable critiques of impact measures more generally. A personal analysis of these critiques will wait one more day. This post will only cover the surface.

Preliminaries

Before I can explain either of the two measures, I must first introduce the language which allows me to precisely define each approach. Both impact measures have quite simple natural language descriptions, but it is easy to feel as though one is not getting the full story if it is explained using English alone. The specific way that the two methods are represented takes place within a Markov decision process (MDP).

Intuitively, an MDP is just a way of representing actions that an agent can take in a stochastic environment, which is made up of a set of states. Formally, an MDP is defined by a tuple (S, A, r, p, γ). S is the set of states in the environment. A is the set of actions that the agent can take. r is a function which maps state-action pairs to a real number reward. p is a function which returns the probability of transitioning into one state given the previous state and an action, $p(s_{t+1}|s_t, a_t)$. γ is the discount factor for the rewards, $\gamma \in [0, 1]$.

Relative reachability

In Victoria Krakovna's blog post introducing relative reachability, she explains that relative reachability was a synthesis of two related ideas: preserving reversibility of the environment, and penalizing impact over states. The central insight was that these two ideas can be combined to avoid the downsides of either of them alone.

The idea behind preserving reversibility is simple. We don't want our artificial intelligence to do something that would make us unable to return things to the way that they were previously. For example, if we wanted the AI to create a waste disposal facility, we might not want it to irrevocably pollute a nearby lake in the process.

The way we formalize state reversibility is by first introducing a *reachability measure*. This reachability measure essentially takes in two states and returns 1 if there is some sequence of actions that the agent could take in order to go from the first state to the final state, and 0 if there is no such sequence of actions. But this is not yet the full description of the reachability measure. In order to take into account uncertainty in the environment, and a discount factor, reachability is actually defined as the following function of two states x and y

$$R(x;y) := \max_{\pi} E[\gamma_r^{N_\pi(x;y)}]$$

where π is some policy, γ_r is the reachability discount factor $\in (0, 1]$, and N a function which returns the number of steps it takes to reach y from x when following π. In English, this is stating that reachability between two states is the expected value of the the discount factor raised to the power of the number of states it would take if one were to follow an optimal policy from the first state to the final state. The more steps we are expected to take in order to go from x to y, the closer reachability is to zero. If there is no sequence of actions which can take us from x to y, then reachability is exactly zero. On the other hand, if x = y, and they are the same state, then the reachability between them is one.

An *unreachability deviation* is a penalty that we can add to actions which incentivizes against taking some irreversible action. This penalty is simply defined as $1 - R(s, s')$ where s' is some baseline state. In other words, if we are very close to the baseline state, then the penalty is close to zero (since reachability would be close to one).

The exact way that we define the baseline state is not particularly important for understanding a first pass through. Naively, the baseline could simply refer to the first step in the episode. It is, however, better to think about the baseline state as some type of reference world where the agent had done decided to do nothing. We can take this concept further by defining "doing nothing" as either a counterfactual reality where the agent was never turned on, or the result of an infinite sequence of nothing actions which began in the last time step. The second interpretation is preferred for a number of reasons, but this isn't crucially important for understanding relative reachability. (Read the paper for more details).

The problem with penalizing actions with the unreachability deviation is that it yields the maximum possible penalty for all actions which result in some irreversibility. This is clearly an issue in a complex environment, since all actions are in some sense irreversible. See section 2.2 in the paper for a specific toy example of why using mere unreachability won't work.

The contribution that Krakovna makes is by introducing a measure which is sensitive to the magnitude of irreversibility. Relative reachability is defined as the average reduction in reachability of all states from the current state compared to the baseline.

This is written as the following, where $d_{RR}(s_t; s_t')$ represents the relative reachability deviation from a state at time t compared to a baseline state s_t'

$$\frac{1}{|S|} \sum_{s \in S} \max(R(s_t'; s) - R(s_t, s), 0)$$

Take a moment to pause and inspect the definition above. We are summing over all states in the environment, and taking a difference between the reachability between the baseline and our current state. This feels like we are determining how far we are from the set of all states in the environment that are close to the baseline. For some particularly irreversible action, relative reachability will assign a high penalty to this action because it reduced the reachability to all the states we could have been in if we had done nothing. The idea is that presumably we should not try to go into regions of the state space which will make it hard to set everything back to "normal." Conversely, we shouldn't enter states that would be hard to get to if we never did anything at all.

Attainable utility

Alexander Turner expanded upon relative reachability by generalizing it to reward functions rather than states. As I understand, it was not Turner's initial intention to create a general version of reachability, but the way that the two approaches ended up being similar allowed for a natural abstraction of both (see the section on Value-difference measures in the relative reachability paper).

Attainable utility is the idea that, rather than caring about the average reduction of state reachability, we should instead care about the average reduction of *utility reachability*. The central insight guiding attainable utility is summed up nicely in a single sentence in his introductory post, Towards A New Impact Measure.

> Impact is change to our ability to achieve goals.

"Goals" in this case refers to some set of arbitrary utility functions. They don't need to be *our* utility functions. They could instead be any sufficiently diverse set of utility functions. In the above post Turner uses the set of all computable utility functions weighted by their complexity. In general these reward functions are referred to as the *auxiliary set*.

There are a few ways that attainable utility has been represented formally. In Conservative Agency the penalty is written as

$$P(s,a) := \sum_{r \in R} |Q_r(s,a) - Q_r(s,\varnothing)|$$

where $\varnothing$ refers to the baseline "do nothing" action and $Q_r(s)$ refers to the Q-value of an action taken at some state s, or in other words the expected cumulative value of taking that action and following an optimal policy from the point of view of the particular reward function. This penalty is then scaled by some constant factor before being incorporated into a utility function.

The way that we choose the scaling depends on an operational choice. We can either measure the impact of some mild reference action, or we can scale by the Q-value over all the reward functions in the auxillary set: $\sum_{r \in R} Q_r(s,\varnothing)$. As mentioned in the last post, the advantage of the first method is that it allows us to avoid the problem of catastrophic miscalibration of impact penalties. Turner tentatively proposes the following,

> Construct a device which, upon receiving a signal (a_{unit}), expends a tiny amount of energy to manufacture one paperclip. The agent will then set ImpactUnit := Penalty($h_{<t}a_{unit}$), re-estimating the consequences of taking the privileged a_{unit} at each time step. To prevent the agent from intentionally increasing ImpactUnit, simply apply 1.01 penalty to any action which is expected to do so.

In both relative reachability and attainable utility preservation we modify the reward function by adding a regularized term. This is represented as the following:

$$R'(s,a) = R(s,a) - \lambda \frac{Penalty(s,a)}{Scale(s)}$$

where λ is some parameter that controls the strength of the impact penalty, perhaps representing the operator's belief in the power of the impact penalty.

What does this solve?

In the introductory post to attainable utility preservation, Turner claims that by using attainable utility, we are able to satisfy a number of desirable properties which were unsatisfied in earlier approaches. Yesterday, I outlined a few notable critiques to impact measurements, such as incentives for keeping the universe in a stasis. Turner sought to outline a ton of potential desiderata for impact measures, some of which were only discovered after realizing that other methods like whitelisting were difficult to make work.

Among the desirable properties are some obvious ones that had already been recognized, like value-agnosticism, natural kind, and the measure being apparently rational. Turner contributed some new ones like dynamic consistency and efficiency, which allowed him to provide tests for his new approach. (It is interesting to compare the computational efficiency of calculating relative reachability and attainable utility).

Some people have disagreed with the significance of some items on the list, and turned to simpler frameworks. Rohin Shah has added,

> My main critique is that it's not clear to me that an AUP-agent would be able to do anything useful. [...]
>
> Generally, I think that it's hard to satisfy the conjunction of three desiderata -- objectivity (no dependence on values), safety (preventing any catastrophic plans) and non-trivialness (the AI is still able to do some useful things).

By contrast Daniel Filan has compiled a list of test cases for impact measures. While both the relative reachability paper and the paper describing attainable utility preservation provided tests on AI safety gridworld environments, it is not clear to me at the moment whether these are particularly significant. I am driven to study impact measurements mainly because of the force of intuitive arguments for each approach, rather than due to any specific empirical test.

The post Best reasons for pessimism about impact of impact measures? is the most comprehensive collection of critiques from the community. So far I have not been able to find any long-form rebuttals to the specific impact measurements. Instead, the best counterguments come from this post above.

In general there is a disagreement about the *aim* of impact measures, and how we could possibly apply them in a way that meaningfully helps align artificial intelligence. In the top reply from the "Best reasons for pessimism" post, lesswrong user *Vaniver* is primarily concerned with our ability to reduce AI alignment into a set of individual issues such that impact measurements helps solve a particular one of these issues.

The state of the debate over impact measurement is best described as informal and scattered across many comments on Lesswrong and the Alignment Forum.

In the next post I will continue my discussion of impact measures by providing what I view as finer grained intuitions for what I think impact measures are good for. This should hopefully provide some insight into what problem we can actually solve by taking this approach, and whether the current impact measures rise to the challenge.

Four Ways An Impact Measure Could Help Alignment

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Impact penalties are designed to help prevent an artificial intelligence from taking actions which are catastrophic.

Despite the apparent simplicity of this approach, there are in fact a plurality of different frameworks under which impact measures could prove helpful. In this post, I seek to clarify the different ways that an impact measure could ultimately help align an artificial intelligence or otherwise benefit the long-term future.

It think it's possible some critiques of impact are grounded in an intuition that it doesn't help us achieve X, where X is something that the speaker *thought* impact was supposed to help us with, or is something that would be good to have in general. The obvious reply to these critiques is then to say that it was never intended to do X, and that impact penalties aren't meant to be a complete solution to alignment.

My hope is that in distinguishing the ways that impact penalties can help alignment, I will shed light on why some people are more pessimistic or optimistic than others. I am not necessarily endorsing the study of impact measurements as an especially tractable or important research area, but I do think it's useful to gather some of the strongest arguments for it.

Roughly speaking, I think that that an impact measure could potentially help humanity in at least one of four main scenarios.

1. Designing a utility function that roughly optimizes for what humans reflectively value, but with a recognition that mistakes are possible such that regularizing against extreme maxima seems like a good idea (ie. Impact as a regularizer).

2. Constructing an environment for testing AIs that we want to be extra careful about due to uncertainty regarding their ability to do something extremely dangerous (ie. Impact as a safety protocol).

3. Creating early-stage task AIs that have a limited function, but are not intended to do any large scale world optimization (ie. Impact as an influence-limiter).

4. Less directly, impact measures could still help humanity with alignment because researching them could allow us to make meaningful progress on deconfusion (ie Impact as deconfusion).

Impact as a regularizer

In machine learning a regularizer is a term that we add to our loss function or training process that reduces the capacity of a model in the hopes of being able to generalize better.

One common instance of a regularizer is a scaled L_2 norm penalty of the model parameters that we add to our loss function. A popular interpretation of this type of regularization is that it represents a prior over what we think the model parameters should be. For example, in Ridge Regression, this interpretation can be made formal by invoking a Gaussian prior on the parameters.

The idea is that in the absence of vast evidence, we shouldn't allow the model to use its limited information to make decisions that *we the researchers* understand would be rash and unjustified given the evidence.

One framing of impact measures is that we can apply the same rationale to artificial intelligence. If we consider some scheme where an AI has been the task of undertaking ambitious value learning, we should make it so that whatever the AI initially believes is the true utility function U, it should be extra cautious not to optimize the world so heavily unless it has gathered a very large amount of evidence that U really is the right utility function.

One way that this could be realized is by some form of impact penalty which eventually gets phased out as the AI gathers more evidence. This isn't *currently* the way that I have seen impact measurement framed. However, to me it is still quite intuitive.

Consider a toy scenario where we have solved ambitious value learning and decide to design an AI to optimize human values in the long term. In this scenario, when the AI is first turned on, it is given the task of learning what humans want. In the beginning, in addition to its task of learning human values, it also tries helping us in low impact ways, perhaps by cleaning our laundry and doing the dishes. Over time, as it gathers enough evidence to fully understand human culture and philosophy, it will have the confidence to do things which are much more impactful, like becoming the CEO of some corporation.

I think that it's important to note that this is not what I currently think will happen in the real world. However, I think it's useful to imagine these types of scenarios because they offer concrete starting points for what a good regularization strategy might look like. In practice, I am not too optimistic about ambitious value learning, but more narrow forms of value learning could still benefit from impact measurements. As we are still somewhat far from any form of advanced artificial intelligence, uncertainty about which methods will work makes this analysis difficult.

Impact as a safety protocol

When I think about advanced artificial intelligence, my mind tends to forward chain from current AI developments, and imagines them being scaled up dramatically. In these types of scenarios, I'm most worried about something like mesa optimization, where in the process of making a model which performs some useful task, we end up searching over a very large space of optimizers that ultimately end up optimizing for some other task which we never intended for.

To oversimplify things for a bit, there are a few ways that we could ameliorate the issue of misaligned mesa optimization. One way is that we could find a way to

robustly align arbitrary mesa objectives with base objectives. I am a bit pessimistic about this strategy working without some radical insights, because it currently seems really hard. If we could do that, it would be something which would require a huge chunk of alignment to be solved.

Alternatively, we could whitelist our search space such that only certain safe optimizers could be discovered. This is a task where I see impact measurements could be helpful.

When we do some type of search over models, we could construct an explicit optimizer that forms the core of each model. The actual parameters that we perform gradient descent over would need to be limited enough such that we could still transparently see what type of "utility function" is being inner optimized, but not so limited that the model search itself would be useless.

If we could constrain and control this space of optimizers enough, then we should be able to explicitly add safety precautions to these mesa objectives. The exact way that this could be performed is a bit difficult for me to imagine. Still, I think that as long as we are able to perform some type of explicit constraint on what type of optimization is allowed, then it should be possible to penalize mesa optimizers in a way that could potentially avoid catastrophe.

During the process of training, the model will start unaligned and gradually shift towards performing better on the base objective. At any point during the training, we wouldn't want the model to try to do anything that might be extremely impactful, both because it will initially be unaligned, and because we are uncertain about the safety of the trained model itself. An impact penalty could thus help us to create a safe testing environment.

The intention here is not that we would add some type of impact penalty to the AIs that are *eventually* deployed. It is simply that as we perform the testing, there will be some limitation on much power we are giving the mesa optimizers. Having a penalty for mesa optimization can then be viewed as a short term safety patch in order to minimize the chances that an AI does something extremely bad that we didn't expect.

It is perhaps at first hard to see how an AI could be dangerous *during* the training process. But I believe that there is good reason to believe that as our experiments get larger, they will require artificial agents to understand more about the real world while they are training, which incurs significant risk. There are also specific predictable ways in which a model being trained could turn dangerous, such as in the case of deceptive alignment. It is conceivable that having some way to reduce impact for optimizers in these cases will be helpful.

Impact as an influence-limiter

Even if we didn't end up putting an impact penalty directly into some type of ambitiously aligned AGI, or use it as a safety protocol during testing, there are still a few disjunctive scenarios in which impact measures could help construct limited AIs. A few examples would be if we were constructing Oracle AIs and Task AGIs.

Impact measurements could help Oracles by cleanly providing a separation between "just giving us true important information" and "heavily optimizing the world in the process." This is, as I understand, one of the main issue with Oracle alignment at the

moment, which means that intuitively an impact measurement could be quite helpful in that regard.

One rationale for constructing a task AGI is that it allows humanity to perform some type of important action which buys us more time to solve the more ambitious varieties of alignment. I am personally less optimistic about this particular solution to alignment, as in my view it would require a very advanced form of coordination of artificial intelligence. In general I incline towards the view that competitive AIs will take the form of more service-specific machine models, which might imply that even if we succeeded at creating some low impact AGI that achieved a specific purpose, it wouldn't be competitive with the other AIs which that themselves have no impact penalty at all.

Still, there is a broad agreement that if we have a good theory about what is happening within an AI then we are more likely to succeed at aligning it. Creating agentic AIs seems like a good way to have that form of understanding. If this is the route that humanity ends up taking, then impact measurements could provide immense value.

This justification for impact measures is perhaps the most salient in the debate over impact measurements. It seems to be behind the critique that impact measurements need to be *useful* rather than just safe and value-neutral. At the same time, I know from personal experience that there at least one person currently thinking about ways we can leverage current impact penalties to be useful in this scenario. Since I don't have a good model for how this can be done, I will refrain from specific rebuttals of this idea.

Impact as deconfusion

The concept of impact appears to neighbor other relevant alignment concepts, like mild optimization, corrigibility, safe shutdowns, and task AGIs. I suspect that even if impact measures are never actually used in practice, there is still some potential that drawing clear boundaries between these concepts will help clarify approaches for designing powerful artificial intelligence.

This is essentially my model for why some AI alignment researchers believe that deconfusion is helpful. Developing a rich vocabulary for describing concepts is a key feature of how science advances. Particularly clean and insightful definitions help clarify ambiguity, allowing researchers to say things like "That technique sounds like it is a combination of X and Y without having the side effect of Z."

A good counterargument is that there isn't any particular reason to believe that *this* concept requires priority for deconfusion. It would be bordering on a motte and bailey to claim that some particular research will lead to deconfusion and then when pressed I appeal to research in general. I am not *trying* to do that here. Instead, I think that impact measurements are potentially good because they focus attention on a subproblem of AI, in particular catastrophe avoidance. And I also think there has empirically been demonstrable progress in a way that provides evidence that this approach is a good idea.

Consider David Manheim and Scott Garrabrant's Categorizing Variants of Goodhart's Law. For those unaware, Goodhart's law is roughly summed up in the saying "Whenever a measure becomes a target, it ceases to become a good measure." This

paper tries to catalog all of the different cases which this phenomenon could arise. Crucially, it isn't necessary for the paper to actually present a solution to Goodhart's law in order to illuminate how we could avoid the issue. By distinguishing ways in which the law holds, we can focus on addressing those specific sub-issues rather than blindly coming up with one giant patch for the entire problem.

Similarly, the idea of impact measurement is a confusing concept. There's one interpretation in which an "impact" is some type of distance between two representations of the world. In this interpretation, saying that something had a large impact is another way of saying that the world changed a lot as a result. In newer interpretations of impact, we like to say that an impact is really about a difference in what we are able to achieve.

A distinction between "difference in world models" and "differences in what we are able to do" is subtle, and enlightening (at least to me). It allows a new terminology in which I can talk about the *impact* of artificial intelligence. For example, in Nick Bostrom's founding paper on existential risk studies, his definition for existential risk included events which could

> permanently and drastically curtail [humanity's] potential.

One interpretation of this above definition is that Bostrom was referring to *potential* in the sense of the second definition of impact rather than the first.

A highly unrealistic way that this distinction could help us is if we had some future terminology which allowed us to unambiguously ask AI researchers to "see how much impact this new action will have on the world." AI researchers could then boot up an Oracle AI and ask the question in a crisply formalized framework.

More realistically, the I could imagine that the field may eventually stumble on useful cognitive strategies to frame the alignment problem such that impact measurement becomes a convenient precise concept to work with. As AI gets more powerful, the way that we understand alignment will become nearer to us, forcing us to quickly adapt our language and strategies to the specific evidence we are given.

Within a particular subdomain, I think an AI researcher could ask questions about what they are trying to accomplish, and talk about it using the vocabulary of well understood topics, which could eventually include impact measurements. The idea of impact measurement is simple enough that it will (probably) get independently invented a few times as we get closer to powerful AI. Having thoroughly examined the concept *ahead* of time rather than afterwards offers future researchers a standard toolbox of precise, deconfused language.

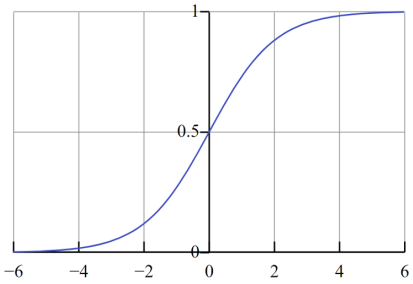
I do not think the terminology surrounding impact measurements will ever quite reach the ranks of terms like "regularizer" or "loss function" but I do have an inclination to think that *simple* and *common sense* concepts should be rigorously defined as the field advances. Since we have intense uncertainty about the type of AIs that will end up being powerful, or about the approaches that will be useful, it is possibly most helpful at this point in time to develop tools which can reliably be handed off for future researchers, rather than putting too much faith into one particular method of alignment.

Why Gradients Vanish and Explode

Epistemic status: Confused, but trying to explain a concept that I previously thought I understood. I suspect much of what I wrote below is false.

Without taking proper care of a very deep neural network, gradients tend to suddenly become quite large or quite small. If the gradient is too large, then the network parameters will be thrown completely off, possibly causing them to become NaN. If they are too small, then the network will stop training entirely. This problem is called the vanishing and exploding gradients problem.

When I first learned about the vanishing gradients problem, I ended up getting a vague sense of why it occurs. In my head I visualized the sigmoid function.



I then imagined this being applied element-wise to an affine transformation. If we just look at one element, then we can imagine it being the result of a dot product of some parameters, and that number is being plugged in on the x-axis. On the far left and on the far right, the derivative of this function is very small. This means that if we take the partial derivative with respect to some parameter, it will end up being extremely (perhaps vanishingly) small.[1]

Now, I know the way that I was visualizing this was very wrong. There are a few mistakes I made:

1. This picture doesn't tell me anything about why the gradient "vanishes." It's just showing me a picture of where the gradients get small. Gradients also get small when they reach a local minimum. Does this mean that vanishing gradients are sometimes good?

2. I knew that gradient vanishing had something to do with the depth of a network, but I didn't see how the network being deep affected why the gradients got small. I had a rudimentary sense that each layer of sigmoid compounds the problem until there's no gradient left, but this was never presented to me in a precise way, so I just ignored it.

I now think I understand the problem a bit better, but maybe not a whole lot better.

(Note: I have gathered evidence that the vanishing gradient problem is not linked to sigmoids and put it in this comment. I will be glad to see evidence which proves I'm wrong on this one, but I currently believe this is evidence that machine learning professors are teaching it incorrectly).

First, the basics. Without describing the problem in a very general sense, I'll walk through a brief example. In particular, I'll show how we can imagine a forward pass in a simple recurrent neural network that enables a feedback effect to occur. We can then immediately see how gradient vanishing can become a problem within this framework (no sigmoids necessary).

Imagine that there is some sequence of vectors which are defined via the following recursive definition,

$$h^{(t)} = W h^{(t-1)}$$

This sequence of vectors can be identified as the sequence of hidden states of the network. Let W admit an orthogonal eigendecomposition. We can then represent this repeated application of the weights matrix as

$$h^{(t)} = Q \Lambda^t Q^\intercal h^{(0)}$$

where Λ is a diagonal matrix containing the eigenvalues of W, and Q is an orthogonal matrix. If we consider the eigenvalues, which are the diagonal entries of Λ, we can tell that the ones that are less than one will decay exponentially towards zero, and the values that are greater than one will blow up exponentially towards infinity as t grows in size.

Since Q is orthogonal, the transformation $Q^\intercal h^{(0)}$ can be thought of as a rotation transformation of the vector $h^{(0)}$ where each coordinate in the new transformation reflects $h^{(0)}$ being projected onto an eigenvector of W. Therefore, when t is very large, as in the case of an unrolled recurrent network, then this matrix calculation will end up getting dominated by the parts of $h^{(0)}$ that point in the same direction as the exploding eigenvectors.

This is a problem because if an input vector ends up pointing in the direction of one of these eigenvectors, the loss function may be very high. From this, it will turn out that in these regions, stochastic gradient descent may massively overshoot. If SDG overshoots, then we end up reversing all of the descent progress that we previously had towards descending down to a local minimum.

As Goodfellow et al. note[2], this error is relatively easy to avoid in the case of non-recurrent neural networks, because in that case the weights aren't shared between layers. However, *in* the case of vanilla recurrent neutral networks, this problem is almost unavoidable. Bengio et al. showed that in cases where a simple neural network is even a depth of 10, this problem will show up with near certainty.

One way to help the problem is by simply clipping the gradients so that they can't reverse all of the descent progress so far. This helps the symptom of exploding gradients, but doesn't fix the problem entirely, since the issue with blown up or vanishing eigenvalues remains.

Therefore, in order to fix this problem, we need to fundamentally re-design the way that the gradients are backpropagated through time, motivating echo state networks, leaky units, skip connections, and LSTMs. I plan to one day go into all of these, but I first need to build up my skills in matrix calculus, which are currently quite poor.

Therefore, I intend to make the next post (and maybe a few more) about matrix calculus. Then perhaps I can revisit this topic and gain a deeper understanding.

---

[1] This may be an idiosyncratic error of mine. See page 105 in these lecture notes to see where I first saw the problem of vanishing gradients described.

[2] See section 10.7 in the Deep Learning Book for a fuller discussion of vanishing and exploding gradients.

A Primer on Matrix Calculus, Part 1: Basic review

Consider whether this story applies to you. You went through college and made it past linear algebra and multivariable calculus, and then began your training for deep learning. To your surprise, much of what they taught you in the previous courses is not very useful to the current subject matter.

And this is fine. Mathematics is useful in its own right. You can expect a lot of stuff isn't going to show up on the deep learning final, *but* it's also quite useful for understanding higher mathematics.

However, what isn't fine is that a lot of important stuff that you do need to know was omitted. In particular, the deep learning course requires you to know matrix calculus, a specialized form of writing multivariable calculus (mostly differential calculus). So now you slog through the notation, getting confused, and only learning as much as you need to know in order to do the backpropagation on the final exam.

This is not how things should work!

Matrix calculus can be beautiful in its own right. I'm here to find the beauty for myself. If I may find it beautiful, then perhaps I will find new joy in reading those machine learning papers. And maybe you will too.

Therefore, I dedicate the next few posts in this sequence to covering this paper, which boldly states that it is, "an attempt to explain all the matrix calculus you need in order to understand the training of deep neural networks." *All* the matrix calculus we need? Perhaps it's enough to understand training neural networks, but it isn't enough matrix calculus for deep learning more generally — I just Ctrl-F'd and found no instance of "hessian"! [1]

Since it's clearly not the full picture, I will supplement my posts with material from chapter 4 of the Deep Learning Book, and Wikipedia.

This subsequence of my daily insights sequence will contain three parts. The first part is this post, the introduction. For the posts in the sequence I have outlined the following rubric:

Part 1 (this one) will be reviewing some multivariable calculus and will introduce the matrix calculus notation.

Part 2 will cover Jacobians, derivatives of element-wise binary operators, derivatives involving scalar expansions, vector sum reduction, and some common derivatives encountered in deep learning.

Part 3 will cover the hessian matrix, higher order derivatives and Taylor approximations, and we will step through an example of applying the chain rule in a neural network.

First, what's important to understand is that most of the calculus used in deep learning is not much more advanced than what is usually taught in a first course in calculus. For instance, there is rarely any need for understanding integrals.[2]

On the other hand, even though the mathematics itself is not complicated, it takes the form of specialized notation, enabling us to write calculus using large vectors and matrices, in contrast to a single variable approach.

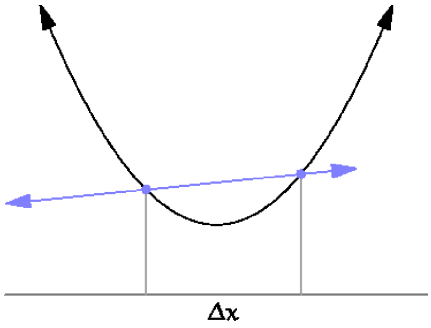
Given this, we might as well start from somewhat of a beginning, with limits, and then build up to derivatives. Intuitively, a *limit* is a way of filling in the gaps of certain functions by finding what value is "approached" when we evaluate a function in a certain direction. The formal definition for a limit given by the epsilon-delta definition, which was provided approximately 150 years after the limit was first introduced.

Let f be a real valued function on a subset D of the real numbers. Let c be a limit point of D and let L be a real number. We say that $\lim_{x \to c} f(x) = L$ if for every $\epsilon > 0$ there exists a δ such that, for all $x \in D$, if $0 < |x - c| < \delta$ then $|f(x) - L| < \epsilon$. For an intuitive explanation of this definition, see this video from 3Blue1Brown.

It is generally considered that this formal definition is too cumbersome to be applied every time to elementary functions. Therefore, introductory calculus courses generally teach a few rules which allow students to quickly evaluate the limits of functions that we are familiar with.

I will not attempt to list all of the limit rules and tricks, since that would be outside of the scope of this single blog post. That said, this resource provides much more information than what would be typically necessary for succeeding in a machine learning course.

The *derivative* is defined on real valued functions by the following definition. Let f be a real valued function, then the derivative of f at a is written $f'(a) = \lim_{h \to 0} \frac{f(a+h)-f(a)}{h}$. The intuitive notion of a derivative is that it measures the slope of a function at a point a. Since slope is traditionally defined as the rate of change between *two* points, it may first appear absurd to a beginner how we can measure slope at a single point. But this absurdity can be visually resolved by viewing the following GIF [3]



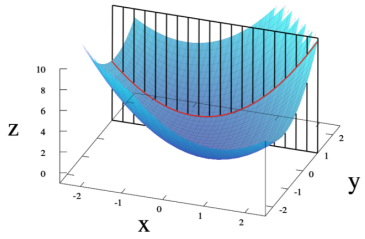
Just as in the case of limits, we are usually not interested in applying the formal definition to functions except when pressed. Instead we have a list of common derivative rules which can help us simplify derivatives of common expressions. Here is a resource which lists common differentiation rules.

In machine learning, we are most commonly presented with functions which whose domain is multi-dimensional. That is, instead of taking the derivative of a function f(x) where x is a real valued variable, we are interested in taking the derivative of functions f(x⃗) where x⃗ is a vector, which is intuitively something that can be written as an ordered list of numbers.

To see why we work with vectors, consider that we usually interested in finding a local minimum of the loss function of a neural network (a measure of how badly the neural network is performing) where the parameters of the neural network are written as an ordered list of numbers. During training, we can write the loss function as a function of its weights and biases. In general, all of deep learning can be reduced notation-wise to simple multidimensional functions, and compositions of those simple functions.

Therefore, in order to understand deep learning, we must understand the multidimensional generalization of the derivative, the gradient. First, in order to construct the gradient, however, we must briefly consider the notion of *partial derivatives*. A quick aside, I have occasionally observed that some people seem at first confused by partial derivatives, imagining them to some type of fractional notion of calculus. [4]

Do not be alarmed. As long as you understand what a derivative is in one dimension, partial derivatives should be a piece of cake. A partial derivative is simply a derivative of a function with respect to a particular variable, *with all the other variables held constant*. To visualize, consider the following multidimensional function that I have ripped from Wikimedia.



Here, we are taking the partial derivative of the function with respect to x where y is held constant at 1 and the z axis represents the co-domain of the function (ie. the axis that is being mapped to). I think about partial derivatives in the same way as the image above, by imagining taking a slice of the function in the x-direction, and thereby reducing the function to one dimension, allowing us to take a derivative. Symbolically, we can evaluate a function's partial derivative like this: say $f(x, y) = 2xy + x^2y$. If we want to take the derivative of f with respect to x, we can treat the y's in that expression as constants and write $\frac{\partial f}{\partial x} = 2y + 2xy$. Here, the symbol ∂ is a special symbol indicating that we are taking the partial derivative rather than the total derivative.

The *gradient* is simply the column vector of partial derivatives of a function. In the previous example, we would have the gradient as $[2y + 2xy \quad 2x + x^2]^\intercal$. The notation I will use here is that the gradient is written as ∇f for some function. The gradient is important because it generalizes the concept of slope to a higher dimension. Whereas the single variable derivative provided us the slope of the function at a single point, the gradient provides us a vector which points in the direction of greatest ascent at a point and whose magnitude is equal to the rate of increase in this direction. Also, just as a derivative allows us to construct a local linear approximation of a function about a point, the gradient allows us to construct a linear approximation of a multivariate function about a point in the form of a hyperplane. From this notion of a gradient, we can "descend" down a loss function by repeatedly subtracting the gradient starting at some point, and in the process find neural networks which are better at doing their assigned tasks.

In deep learning we are often asked to take the gradient of a function $f : R^{n \times m} \to R$ (this notation is just saying that we are mapping from a space of matrices to the real number line). This may occur because the function in question has its parameters organized in the form of an n by m matrix, representing for instance the strength of connections from neuron i to neuron j. In this case, we treat the gradient exactly as we did before, by collecting all of the partial derivatives. There is no difference, except in notation.

Some calculus students are not well acquainted with the proof for why the gradient points in the direction of greatest ascent. Since it is simply a list of partial derivatives, this fact may seem surprising. Nonetheless, this fact is what makes the gradient centrally important in deep learning, so it is therefore worth repeating here.

In order to see why the gradient points in the direction of steepest ascent, we first need a way of measuring the ascent in a particular direction. It turns out that multivariable calculus offers us such a tool. The *directional derivative* is the rate of change of a function f along a direction $\vec{v}$. We can imagine the directional derivative as being conceptually similar to the partial derivative, except we would first change the basis while representing the function, and then evaluate the partial derivative with respect to a basis vector which is on the span of $\vec{v}$. Similar to the definition for the derivative, we define the directional derivative $\nabla_{\vec{v}}$ as

$$\nabla_{\vec{v}} f(x) = \lim_{h \to 0} \frac{f(x+h\vec{v})-f(x)}{h}$$

Additionally, we can employ the multivariable chain rule to re-write the directional derivative in a way that uses the gradient.[5] In single variable calculus, the chain rule can be written as $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$. In the multivariable case, for a function $f : R^k \to R$, we write $\frac{d}{dx} f(g(x)) = \sum_{i=1}^{k} \frac{d}{dx} g_i(x) D_i f(g(x))$ where D_i is the partial derivative of f with respect to its ith argument. This can be simplified by employing the following notation, which uses a dot product: $\frac{d}{dx} f(g(x)) = \nabla f \cdot g'(x)$.

If we rewrite the definition of the directional derivative as $\nabla_{\vec{v}} f(x) = \frac{d}{dt} f(x + t\vec{v})$, and then apply the multivariate chain rule to this new formulation, we find that $\nabla_{\vec{v}} f(x) = \frac{d}{dt} f(x_0 + t\vec{v}_0, x_1 + t\vec{v}_1, \ldots, x_k + t\vec{v}_k) = \nabla f \cdot \vec{v}$.

Given that $\nabla_{\vec{v}} f(x) = \nabla f \cdot \vec{v}$, the unit vector $\vec{v}$ which maximizes this dot product is the unit vector which points in the same direction as ∇f. This previous fact can be proven by a simple inspection of the definition of the dot product between two vectors, which is that $\vec{u} \cdot \vec{v} = ||\vec{u}||\,||\vec{u}|| \cos(\theta)$ where θ is the angle between the two vectors.

$||\vec{u}||\,||\vec{u}|| \cos(\theta)$ is maximized when $\theta = 0$. For more intuition on how to derive the dot product, I recommend this video from 3Blue1Brown. I also recommend this one for intuitions on why the gradient points in the direction of maximum increase.

---

[1] For more depth I recommend this part four of this pdf text. For even more depth I recommend this book (though I have not read it). For even more depth, I recommend

seeing the footnote below. For even *more* depth than that, perhaps just try to complete a four year degree in mathematics.

[2] Even this 1962 page behemoth called a book, intended to introduce all of the mathematics needed for a computer science education, includes very little information on integration, despite devoting full chapters to topics like tensor algebras and topology. However, if or when I blog about probability theory, integration will become relevant again.

[3] If this wasn't enough for you, alternatively you can view the 3Blue1Brown video on derivatives.

[4] Interestingly, fractional calculus is indeed a real thing, and is very cool.

[5] For a proof which is does not use the multivariable chain rule, see here. I figured given the primacy of the chain rule in deep learning, it is worth mentioning now.

A Primer on Matrix Calculus, Part 2: Jacobians and other fun

I started this post thinking that I would write all the rules for evaluating Jacobians of neural network parameters in specific cases. But while this would certainly be useful for grokking deep learning papers, frankly it's difficult to write that in Latex and the people who have written The Matrix Calculus You Need For Deep Learning paper have already done it much better than I can do.

Rather, I consider my comparative advantage here to provide some expansion on why we should use Jacobians in the first place. If you were to just read the paper above, you might start to think that Jacobians are just notational perks. I hope to convince you that they are much more than that. In at least one setting, Jacobians provide a mathematical framework for analyzing the input-output behavior of deep neural networks, which can help us see things which we might have missed without this framework. A specific case of this phenomenon is a recently discovered technique which was even more recently put into a practical implementation: Jacobian regularization. Here we will see some fruits of our matrix calculus labor.

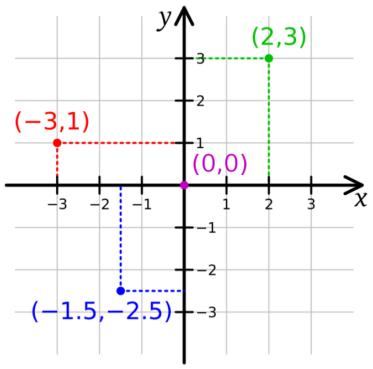
Deep learning techniques require us to train a neural network by slowly modifying parameters of some function until the function begins returning something close to the intended output. These parameters are often represented in the form of matrices. There are a few reasons for this representation: the matrix form is compact, and it allows us to use the tools of linear algebra directly. Matrix computations can also be processed in parallel, and this standardization allows programmers to build efficient libraries for the training of deep neural networks.

One quite important matrix in deep learning is the Jacobian.

In one sense, the Jacobian matrix is just a way of organizing gradient vectors. Gradient vectors, in turn, are just ways of organizing partial derivatives of an expression. Therefore, the Jacobian matrix is just a big matrix which allows us to organize a lot of partial derivatives. Formally, the Jacobian of $f : R^n \to R^m$ is defined by the following matrix. We denote $f_i(x)$ as the mapping from $R^n \to R_i$, where R_i is the real number line in the ith coordinate of the output vector R^m. Then the Jacobian is simply

$$\begin{bmatrix} \nabla f_1^T \\ \nabla f_2^T \\ \vdots \\ \nabla f_m^T \end{bmatrix}$$

But describing the Jacobian as just some big rectangular box of partial derivatives of vector-valued functions hides the intuition for why Jacobians are important. To truly grasp the Jacobian, we will first build up to it by imagining a function that maps between two real coordinate spaces: $f : \mathbb{R}^n \to \mathbb{R}^m$. If n = m, this function has a natural interpretation. We can see it by considering the case where n = m = 2.

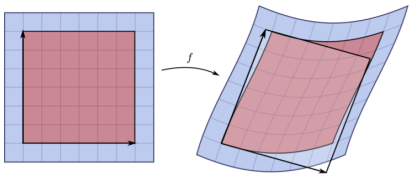


Now imagine stretching, warping, and contracting the plane so that the points are moved around in some manner. This means that every point is mapped to some other point on the graph. For instance, (2,5) could be mapped to (1,3). Crucially, make sure this transformation doesn't cause any sort of sharp discontinuity: we don't want the graph to rip. I am not good with illustrating that type of thing, so I encourage you to imagine it instead in your head, or alternatively watch this video.

There is a special set of such mappings, which we call *linear transformations*. Linear transformations have the property that when we perform the stretching, the gridlines are kept straight. We could still rotate the graph, for instance. But what we can't do is bend some axis so that it takes on a curvy shape after the transformation. If we drew a line on the graph before the linear transformation, it must remain a straight line after the transformation.

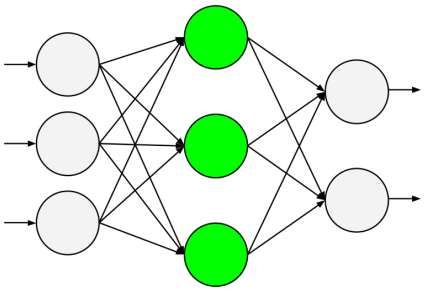
What does this have to do with the Jacobian? To see we first must ask why derivatives are useful. In the most basic sense, the derivative is getting at trying to answer the questoin, "What is the local behavior of this function?"

First, if you will consider by analogy, we could ask for the local behavior of some differentiable function $f : \mathbb{R} \to \mathbb{R}$. This would be a line whose slope is provided by the derivative. Similarly we could ask for the local behavior of some multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, which would be a hyperplane whose direction is determined by the gradient. Now when we ask what the local behavior of some vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ is, we get a linear transformation described by the Jacobian.



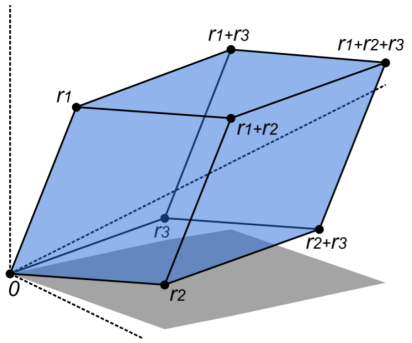
In the above illustration, the Jacobian is evaluated at the point in the bottom left corner of the red tiles. The linear transformation implied by the Jacobian is represented by the translucent square after the function is applied, which is a rotation transformation some angle clockwise. As we can see, while f is some extra curvy function, the Jacobian can approximate the local behavior at a point quite well.

To see how this visualization is useful, consider the case of applying a Jacobian to a neural network. We could be designing a simple neural network to predict the output of two variables, representing perhaps normalized class probabilities, given an input of three variables, representing perhaps input pixel data. We now illustrate the neural network.



This neural networks implements a particular function from $R^3 \rightarrow R^2$. However, the exact function that is being implemented depends crucially on the parameters, here denoted by the connections between the nodes. If we compute the Jacobian of this neural network with respect to the input, at some input instance, we would end up getting a good idea of how the neural network changes within the neighborhood of that particular input.

One way we can gain insight from a Jacobian is by computing its *determinant*. Recall, a determinant is a function from square matrices to scalars which is defined recursively as the alternating sum and subtraction of determinants of the minors of the matrix multiplied by elements in the top row. On second thought, *don't* recall that definition of determinant; that's not going to get you anywhere. Despite the determinant's opaque definition, we can gain deeper insight into what the determinant represents by instead viewing it geometrically. In a few words, the determinant computes the scaling factor for a given linear transformation of a matrix.

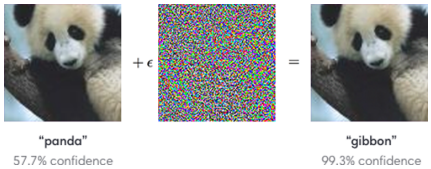


Above, I have pulled from Wikimedia a parallelepiped, which was formed from some linear mapping of a cube. The volume of this parallelepiped is some multiple of the volume of the cube before the transformation. It turns out that no matter which region of space we look at, this linear transformation will generate the *same* ratio from post-transformed regions to pre-transformed regions. This ratio is given by the determinant of the matrix representing the linear mapping. In other words, the determinant tells us how much some transformation is expanding or contracting space.

What this means is for the Jacobian is that the determinant tells us how much space is being squished or expanded in the *neighborhood* around a point. If the output space is being expanded a lot at some input point, then this means that the neural network is a bit unstable at that region, since minor alterations in the input could cause huge distortions in the output. By contrast, if the determinant is small, then some small change to the input will hardly make a difference to the output.

This very fact about the Jacobian is behind a recent development in the regularization of deep neural networks. The idea is that we could use this interpretation of the Jacobian as a measure of robustness to input-perturbations around a point to make neural networks more robust off their training distribution. Traditional approaches like L^2 regularization have emphasized the idea of keeping some parameters of the neural network from wandering off into extreme regions. The idea here is that smaller parameters are more likely *a priori*, which motivates the construction of some type of penalty on parameters that are too large.

In contrast to L^2 regularization, the conceptual framing of Jacobian regularization comes from a different place. Instead of holding a leash on some parameters, to keep them from wandering off into the abyss, Jacobian regularization emphasizes providing robustness to small changes in the input space. The motivation behind this approach is clear to anyone who has been paying attention to adversarial examples over the last few years. To explain, adversarial examples are cases where we provide instances of a neural network where it performs very poorly, even if it had initially done well on a non-adversarial test set. Consider this example, provided by OpenAI.



The first image was correctly identified as a panda by the neural network. However, when we added a tiny bit of noise to the image, the neural network spit out garbage, confidently classifying a nearly exact copy as a gibbon. One could imagine a hypothetical adversary using this exploit to defeat neural network systems in practice. In the context of AI safety, adversarial attacks constitutes a potentially important subproblem of system reliability.

In Jacobian regularization, we approach this issue by putting a penalty on the size of the entries in the Jacobian matrix. The idea is simple: the smaller the values of the matrix, the less that tiny perturbations in input-space will affect the output. Concretely, the regularizer is described by taking the frobenius norm of the Jacobian matrix, $||J(x)||_F^2$. The frobenius norm is nothing complicated, and is really just a way of describing that we square all of the elements in the matrix, take the sum, and then take the square root of this sum. Put another way, if we imagine concatenating all the gradient vectors which compose the Jacobian, the frobenius norm is just describing the L^2 penalty of this concatenated vector.

Importantly, this technique is subtly different from taking the L^2 norm over the *parameters*. In the case of a machine learning algorithm with no linearity, this penalty does however reduce to L^2 regularization. Why? Because when we take the Jacobian of a purely affine function, we obtain the global information about how the function stretches and rotates space, excluding the translation offset. This global information precisely composes the parameters that we would be penalizing. It is theoretically similar to how if we take the derivative of a line, we can reconstruct the line from the derivative and a bias term.

If while reading the last few paragraphs, you starting thinking *how is this just now being discovered?* you share my thoughts exactly. As far as I can tell, the seeds of Jacobian regularization have existed since at least the 1990s. However, it took until 2016 for a team to create a full implementation. Only recently, as I write this in August 2019, has a team of researchers claimed to have discovered an efficient algorithm for applying this regularization penalty to neural networks.

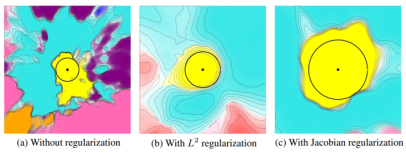
The way that researchers created this new method is by using random projections to approximate the Frobenius norm. Whereas the prior approach *mentioned* random projections, it was never put into practice. The new paper succeeded by devising an algorithm to approximate Jacobians efficiently with minimal overhead cost.

How efficiently? The paper states that there is

> only a negligible difference in model solution quality between training with the exact computation of the Jacobian as compared to training with the approximate algorithm, even when using a single random projection.

If this technique really works as its described, this is a significant result. The paper claims that by applying Jacobian regularization, training uses only 30 percent more computation compared to traditional stochastic gradient descent without regularization at all. And for all that, we get some nice benefits: the system was significantly more robust to a PGD attack, and it was apparently much better than

vanilla L^2 regularization due to the distance between decision cells in the output space.



(a) Without regularization (b) With L^2 regularization (c) With Jacobian regularization

I recommend looking at the paper for more details.

A Primer on Matrix Calculus, Part 3: The Chain Rule

This post concludes the subsequence on matrix calculus. Here, I will focus on an exploration of the chain rule as it's used for training neural networks. I initially planned to include Hessians, but perhaps for that we will have to wait.

Deep learning has two parts: *deep* and *learning.* The *deep* part refers to the fact that we are composing simple functions to form a complex function. In other words, in order to perform a task, we are mapping some input x to an output y using some long nested expression, like $y = f_1(f_2(f_3(x)))$. The *learning* part refers to the fact that we are allowing the properties of the function to be set automatically via an iterative process like gradient descent.

Conceptually, combining these two parts is easy. What's hard is making the whole thing efficient so that we can get our neural networks to actually train on real world data. That's where the backpropagation enters the picture.

Backpropagation is simply a technique to train neural networks by efficiently using the chain rule to calculate the partial derivatives of each parameter. However, backpropagation is notoriously a pain to deal with. These days, modern deep learning libraries provide tools for automatic differentiation, which allow the computer to automatically perform this calculus in the background. However, while this might be great for practitioners of deep learning, here we primarily want to understand the notation as it would be written on paper.[1] Plus, if we were writing our own library, we'd want to know what's happening in the background.

What I have discovered is that, despite my initial fear of backpropagation, it is actually pretty simple to follow if you just understand the notation. Unfortunately, the notation can get a bit difficult to deal with (and was a pain to write out in Latex).

We start by describing the single variable chain rule. This is simply $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$. But if we write it this way, then it's in an opaque notation and hides which variables we are taking the derivative with respect to. Alternatively we can write the rule in a way that makes it more obvious what we are doing: $\frac{d}{dx}f(g(x)) = \frac{df}{dg}\frac{dg}{dx}$, where g is meant as shorthand for g(x). This way it is intuitively clear that we can cancel the fractions on the bottom, and this reduces to $\frac{df}{dx}$, as desired.

It turns out, that for a function $f : R^n \to R^m$ and $g : R^k \to R^n$, the chain rule can be written as $\frac{\partial}{\partial x}f(g(x)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ where $\frac{\partial f}{\partial g}$ is the Jacobian of f with respect to g.

Isn't that neat. Our understanding of Jacobians has now well paid off. Not only do we have an intuitive understanding of the Jacobian, we can now formulate the vector

chain rule using a compact notation — one that matches the single variable case perfectly.[2]

However, in order to truly understand backpropagation, we must go beyond mere Jacobians. In order to work with neural networks, we need to introduce the *generalized Jacobian.* If the Jacobian from yesterday was spooky enough already, I recommend reading no further. Alternatively if you want to be able to truly understand how to train a neural network, read at your own peril.

First, a vector can be seen as a list of numbers, and a matrix can be seen as an ordered list of vectors. An ordered list of matrices is... a tensor of order 3. Well not exactly. Apparently some people are actually disappointed with the term tensor because a tensor means something very specific in mathematics already and isn't *just* an ordered list of matrices.[3] But whatever, that's the term we're using for this blog post at least.

As you can probably guess, a list of tensors of order n is a tensor of order n + 1. We can simply represent tensors in code using multidimensional arrays. In the case of the Jacobian, we were taking the derivative of functions between two vector spaces, R^n and R^m. When we are considering mapping from a space of tensors of order n to a space of tensors of order m, we denote the relationship y = f(x) as between the spaces $R^{(M_1 \times M_2 \times \ldots \times M_n)} \to R^{(M_1 \times M_2 \times \ldots \times M_m)}$.

The generalized Jacobian J between these two spaces is an object with shape $(M_1 \times M_2 \times \ldots \times M_n) \times (N_1 \times N_2 \times \ldots \times N_m)$. We can think of this object as a generalization of the matrix, where each row is a tensor with the same shape as the tensor y and each column has the same shape as the tensor x. The intuitive way to understand the generalized Jacobian is that we can index J with vectors $\vec{i}$ and $\vec{j}$. At each index in J we find the partial derivative between the variables $y_{\vec{i}}$ and $x_{\vec{j}}$, which are scalar variables located in the tensors y and x.

Formulating the chain rule using the generalized Jacobian yields the same equation as before: for z = f(y) and y = g(x), $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$. The only difference this time is that $\frac{\partial z}{\partial x}$ has the shape $(K_1 \times \ldots \times K_{D_z}) \times (M_1 \times \ldots \times M_{D_x})$ which is itself formed by the result of a generalized matrix multiplication between the two generalized matrices, $\frac{\partial z}{\partial y}$ and $\frac{\partial y}{\partial x}$. The rules for this generalized matrix multiplication is similar to regular matrix multiplication, and is given by the formula:

$$\left(\frac{\partial z}{\partial x}\right)_{i,j} = \sum_{k} \left(\frac{\partial z}{\partial y}\right)_{i,k} \left(\frac{\partial y}{\partial x}\right)_{k,j}$$

However, where this differs from matrix multiplication is that i, j, k are vectors which specify the location of variables within a tensor.

Let's see if we can use this notation to perform backpropagation on a neural network. Consider a neural network defined by the following composition of simple functions: $f(x) = W_2(\text{relu}(W_1 x + b_1)) + b_2$. Here, relu describes the activation function of the first layer of the network, which is defined as the element-wise application of $\text{relu}(x) = \max(x, 0)$. There are a few parameters of this network: the weight matrices, and the biases. These parameters are the things that we are taking the derivative with respect to.

There is one more part to add before we can train this abstract network: a loss function. In our case, we are simply going to train the parameters with respect to the loss function $L(\hat{y}, y) = ||\hat{y} - y||_2^2$ where $\hat{y}$ is the prediction made by the neural network, and y is the vector of desired outputs. In full, we are taking $\frac{\partial}{\partial w} L(f(x), y)$, for some weights w, which include W_1, W_2, b_1, b_2. Since this loss function is parameterized by a constant vector y, we can henceforth treat the loss function as simply $L(f(x))$.

Ideally, we would not want to make this our loss function. That's because the true loss function should be over the entire dataset — it should take into account how good the predictions were for each sample that it was given. The way that I have described it only gave us the loss for a single prediction.

However, taking the loss over the entire dataset is too expensive and converges slowly. Alternatively, taking the loss over a single point (ie: stochastic gradient descent) is also too slow because it doesn't allow us to take into account parallel hardware. So, actual practitioners use what's called mini-batch descent, where their loss function is over some subset of the data. For simplicity, I will just show the stochastic gradient descent step.

For $\frac{\partial}{\partial b_2} L(f(x))$ we have $\frac{\partial}{\partial b_1} L(f(x)) = \frac{\partial L}{\partial f} \frac{\partial f}{\partial b_2}$. From the above definition of f, we can see that $\frac{\partial f}{\partial b_2} = I$, where I is the identity matrix. From here on I will simply assume that the partial derivatives are organized in some specific manner, but omitted. The exact way it's written doesn't actually matter too much as long as you understand the *shape* of the Jacobian being represented.

We can now evaluate $\frac{\partial f}{\partial W_2}$. Let U be (relu($W_1 x + b_1$)). Then computing the derivative $\frac{\partial f}{\partial W_2}$comes down to finding the generalized Jacobian of $W_2 U$with respect to W_2. I will illustrate what this generalized Jacobian would look like by building up from analogous, lower order derivatives. The derivative $\frac{dy}{dx}$ of $y = cx$ is c. The gradient $\nabla_x c^\top x$ is c. The Jacobian J_x of Ux is U. We can therefore see that the generalized Jacobian J_{W_2} of $W_2 U$ will be some type of order 3 tensor which would look like a simple expression involving U.

The derivatives for the rest of the weight matrices can be computed similarly to the derivatives I have indicated for b_2 and W_2. We simply need to evaluate the terms later on in the chain $\frac{\partial l}{\partial f} \cdots \frac{\partial v}{\partial W_1}$where v is shorthand for the function $v = W_1 x$.

We have, however, left out one crucial piece of information, which is how to calculate the derivative over the relu function. To do that we simply separate the derivative into a piecewise function. When the input is less than zero, the derivative is 0. When the input is greater than zero, the derivative is 1. But since the function is not differentiable at 0, we just pretend that it is and make it's derivative 0; this doesn't cause any issues.

$$\frac{\partial}{\partial x} \text{relu}(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

This means that we are pretty much done, as long as you can fill in the details for computing the generalized Jacobians. The trickiest part in the code is simply making sure that all the dimensions line up. Now, once we have computed by derivatives, we can incorporate this information into some learning algorithm like Adam, and use this to update the parameters and continue training the network.

There are, however, many ways that we can make the algorithm more efficient than one might make it during a naive implementation. I will cover one method briefly.

We can start by taking into account information about the direction we are calculating the Jacobians. In particular, if we consider some chain $\frac{\partial l}{\partial f} \cdots \frac{\partial v}{\partial W_1}$, we can take advantage of the fact that tensor-tensor products are associative. Essentially, this means that we can start by computing the last derivative $\frac{\partial v}{\partial W_1}$ and then multiplying forward. This is called *forward accumulation*. We can also compute this expression in reverse, which is referred to as *reverse accumulation*.

Besides forward and reverse accumulation, there are more complex intracacies for fully optimizing a library. From Wikipedia,

> Forward and reverse accumulation are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ with a minimum number of arithmetic operations is known as the *optimal Jacobian accumulation* (OJA) problem, which is NP-complete.

Now if you've followed this post and the last two, and filled in some of the details I (sloppily) left out, you should be well on your way to being able to implement efficient backpropagation yourself. Perhaps read this famous paper for more ways to make it work.

---

[1] This is first and foremost my personal goal, rather than a goal that I expect the readers here to agree with.

[2] If you want to see this derived, see section 4.5.3 in the paper.

[3] The part about people being disappointed comes from my own experience, as it's what John Canny said in CS 182. The definition of Tensor can be made more precise as a multidimensional array that satisfies a specific transformation law. See here for more details.