Artificial Neural Networks for "Kids"

Jordan Bennett

Artificial Neural Networks for "kids":

The easiest most intuitive neural network tutorial you'll probably ever find.

Jordan Bennett

Contents

Life is strange

I wrote an artificial neural network from scratch 2 years ago, and at the same time, I didn't grasp how an artificial neural network actually worked.

But how??

▽

▽

So two years ago, I saw a nice artificial neural network tutorial on youtube by David Miller in c++.

I didn't know any c++ back then, but java and c++ are somewhat similar looking.

So back then I watched the tutorial, then for a little while I began to think about an overall picture of what was going on.

After 2 weeks or so of thinking ***about how David's code was organized*** back then, I ***had developed a mental model*** of the neural network, including all the functions, and all the relevant attributes (Sometimes my memory doesn't betray me). So, from scratch, I then transcribed what I learnt in the form of java code.

Surprisingly, based on ***my mental model,*** the neural net worked well!!

▽

▽

Admittedly, that ***was not the best way*** of learning how neural nets actually work. ***That was not an appropriate mental model to have of neural nets!***

I may have gotten the model to nicely work back then, but I can't say I possessed an intuitive picture of what was going on.

Of-course this has now changed, as reflected in the detailed intuition provided in this book.

▽

▽

Turns out to understand things ***effectively***, takes more than actually being able to successfully write an elementary artificial neural network, based on thoughts about how some artificial neural net code implementation may be organized!!

Key thing is, if one has an ***intuitive understanding***, the programming language or the way in which one organizes the code, has little bearing on actually understanding the fundamental mechanism of artificial neural nets!

This means that ***studying how some code somewhere is organized***, likely ***won't win you an intuitive grasp*** of the basic neural network layout. As a result, you may not be motivated to further pursue the ***exciting field*** of artificial neural networks, which is a core part of perhaps mankind's most important task …. primarily because you may lack the intuitive building blocks needed to ***properly venture beyond*** the scope of elementary artificial neural networks. ***This book is all about a clear/intuitive understanding (i.e. Fun Introduction + Detailed Maths)!***

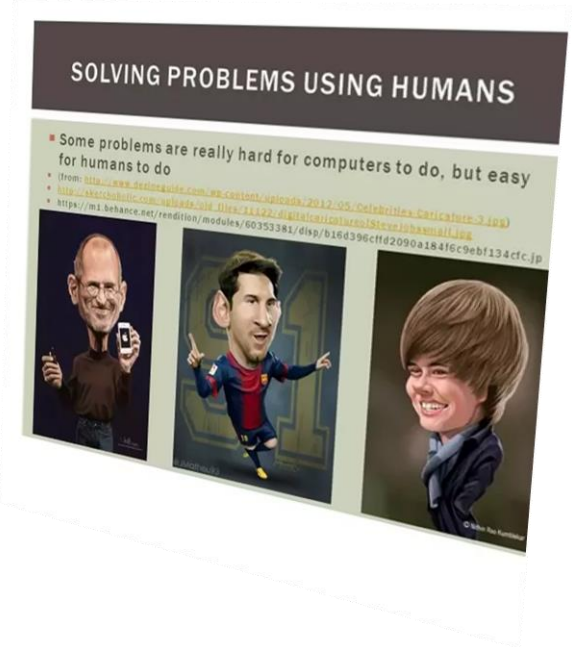
Preface

This short book is like a nice extra topping over what is the probably the **clearest**, and **most intuitive short 4-video neural network** series on youtube by a channel called "3blue1brown".

1. **First** I will explain the **intuitive** extra topping I describe above while avoiding math.
2. **Then** I will attach a youtube link to one of 3blue1brown's relevant video, while going into **some maths** that will be easy to grasp, and easier if one watches said video.
3. This book **(my extra topping**) should make 3blue1brown's **enchantingly clear** videos**, even clearer!!**

What the heck is a neural network?

Let's say we want to classify some digits from individual pictures of digits (aka enable a model to tell what digits are on pictures it is fed).



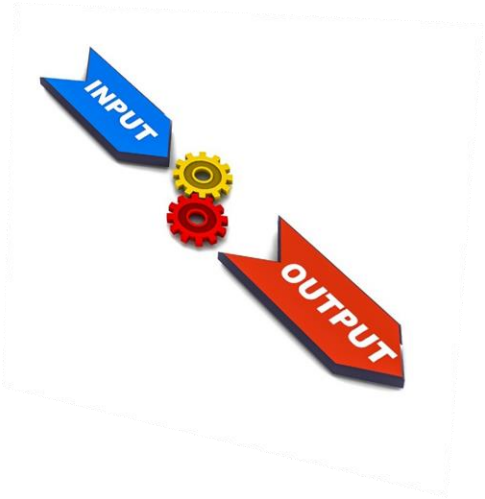
Humans find it **easy** to tell the difference between many creatively written 2's and 1's, but this is non-trivial for the computer to do. Well, we can make it trivial by thinking about the structure of a simple neural net to do it.

What we want to do is make the neural net learn what the digits are, by **adjusting** the neural net's structure **based on** many inputs which are actually examples of correctly labelled digits.

But **why** adjust its structure? And what the heck is this structure anyway?

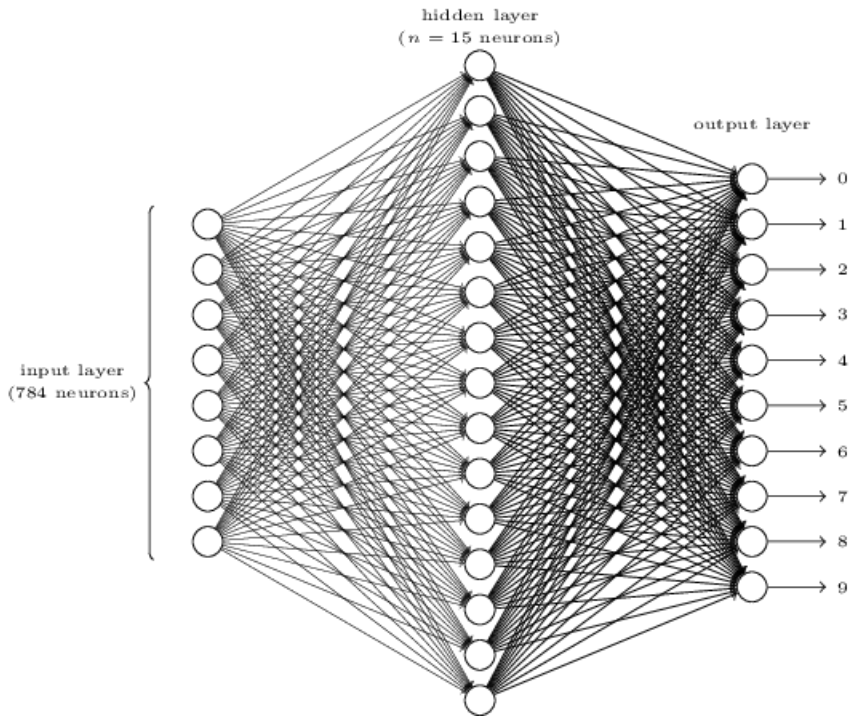
The structure is a collection of data or "parameters" that are simply a way to hold aka store or memorize what the images are, based on labels.

Otherwise where else to store what each picture of a digit means? Certainly not nowhere.

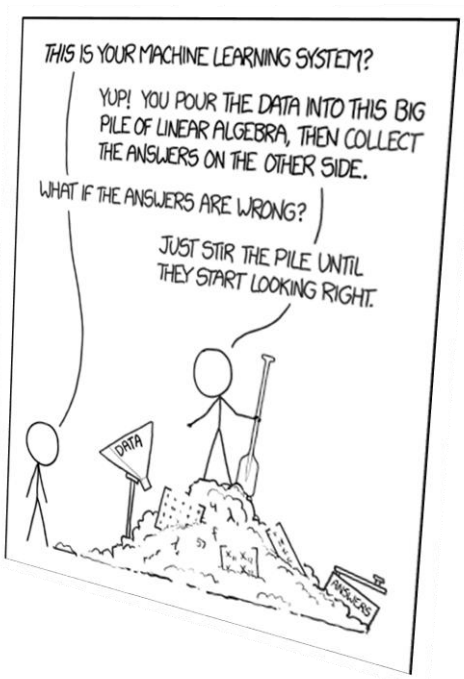


A neural network will have **input structure** to **receive values** from the input pictures with digits written on them, a **"hidden" structure** which acts as **an extra way to represent the inputs**, and a **final output structure** to tag or store answers about digit picture.

The more **hidden** structure there is, is the more opportunity there is to memorize our digits correct labels.



Neural nets will have layers of each type of structure, and multiple **nodes (aka neurons)** per layer. Apart from that, there is also more structure **connecting** each layer, by forming **paths** from **nodes** of one layer, to nodes of another layer. These **connections** are **weights**. So in the end there are weights and neurons, a set of neurons making up each layer, and weights connecting them all.

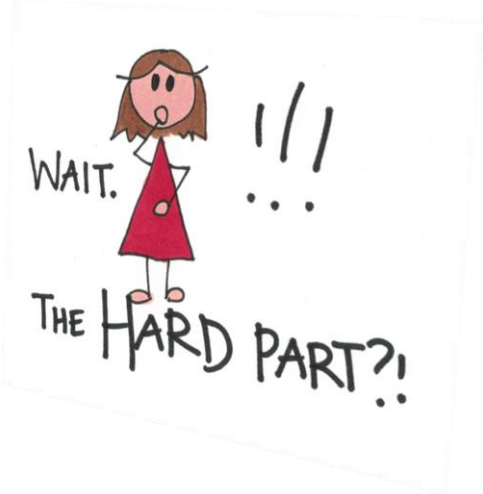


What we do is we pass input data such as picture of a digit, in the form of a collection of numbers representing each pixel from the picture, straight to the layer of input receiver nodes/neurons. There is then a **memorization** process that uses the hidden or extra storage layer of neurons, and then the weights communicate this information to the final layer, which has 10 neurons.

The classification of digits involves 10 possible answers (0-9), and so the neural net will have 10 output neurons, each corresponding to 0 or 1 or 2...up to 9.

The digit classifier will output 1 for any digit it thinks it’s seeing, otherwise 0 because it doesn't think it’s seen another number.

The “hard” part aka some nice maths



I urgently advise you to watch these short nice videos in order (video 1, video 2, video 3, and video 4), whether you don’t yet grasp the overall idea of the neural net, or not. Videos 1-3 explain the overall idea with **nice animations**, then video 4 gets into the work. They belong to the same guy I was telling you about in the preface, and are the most intuitive lessons I’ve seen on youtube thus far. I just add the cherry on top here to make it super clearer!

Now let’s discuss video 4, where all the “magic”/calculus happens:

Now, if one payed close attention, one can formulate what is occurring in massive "plain" neural networks, in the 3 simple parts below:

Part I -- "Trailing hypersurfaces" & "Training set averages":

It is useful to note that the when neural nets work, this work can be observed as simply computing measurements of change, from the context of **one neuron at a time** with respect to some **target value aka our correct label of an input image**. (i.e. changes that should occur in order to alter the structure of the neural net in the long run - See "**Part III - Error correction"** after reading the rest of **Part I**)

For single neuron per layer neural network, we get a **simple hypersurface** for each computation on a current context neuron, but for multiple neurons per layer neural networks (*which we mostly discuss below, because such multiple neuron per layer neural nets are far more powerful, and can do things such as digit detection*), we get a "**trailing hypersurface**" for each computation on a current context neuron, because we need to incorporate the behaviour of other neurons in the same layer as our **current neuron**. Each neuron per layer will have something to say about what changes need to take place in the structure of biases and weights in the neural net, so unlike single neuron per layer neural nets where we could care less about same layer neurons (as single neuron per layer neural nets only have one neuron per layer) in contrast, in

multiple neuron per layer neural nets, neurons in the same layer as some current neuron from which changes are being measured, are **considerable as partners** that have some say in what that **current neuron is thinking.** (The "computations" will be explained below)

These hypersurfaces are components for computing costs. (Where cost changes tell us how badly our artificial neural net is doing)

<u>So, for example, for some cost-change, we compute:</u>

1. For **toyish** single neuron per layer artificial neural nets, in our derivatives, there are sums forming "**hypersurfaces**" spanning from prior neurons/weights, to some current neuron. {i.e. $z = (w^L \cdot a^{L-1} + b^L)$} where a is an activation over some **prior hypersurface**, and thus written with $L-1$ while w is a weight corresponding to the same current context layer neuron for which change is being measured, and hence written with L. **However,** for more practical artificial neural nets that can do more sophisticated tasks like digit detection, we have to incorporate indices j and k to indicate that we are dealing with multiple weights and multiple neurons per layer L, where L indicates some layer with a neuron at j, where j, k indicates one weight that may correspond with that neuron j or another neuron $j+n$ in layer L {i.e. $z_j = \left(w^L_{j,k} \cdot a^{L-1}_k + w^L_{j,k+1} \cdot a^{L-1}_{k+1} + b^L\right)\ldots$} So as you can see, in the prior equation, we constrain to $L-1$, but increment k in the weight notation $w_{j,k}$ as we consider activations of each neuron k in $L-1$, with respect to any neuron that shares weights j, k with our current neuron j in L. **Note that any term looking like "**$w^L \cdot a^{L-1} + b^L$**"** (for single neuron per layer artificial neural nets) and "($w^L_{j,k} \cdot a^{L-1}_k + w^L_{j,k+1} \cdot a^{L-1}_{k+1} + b^L$)" (for multiple neuron per layer artificial neural nets) **is called a hypersurface in machine learning.**
2. For both simple **single** neuron per layer neural nets, and **multiple** neuron per layer neural nets, from the context of any current neuron from which changes occur, we always compute ***changes* in biases**, and **weights**, **changes in activations** for our **current** layer L, but also ***changes* in activations from *prior* layers**, which we ***don't use*** for the context of our **current neuron**, but **save** for **later use** when we ***switch our context*** to some neuron in some **prior layer**. For **toyish** single neuron artificial neural nets, where each layer comprises of a single neuron, from **the**

context of some current neuron L (*where I will **interchangeably** refer to a neuron as either a layer or a neuron L, since for single neuron per layer neural net there is only one neuron per layer, and hence a neuron can be seen as a layer, so when you see me say things like "neuron in our prior layer L"…take it to mean simply that that layer L is the neuron.):* We compute the "negative gradient" aka our cost $-\nabla C$ by equation $C = [\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial W^L}, \frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial B^L}, moreWeightChangeAverages..]$ where we compute **changes in activations for the prior layer** L: $\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial A^{L-1}}$ (notice A^{L-1}) and store it in some variable, and save it for later use for some neuron in some prior layer $L-1$. (Also, note that I use C_g, instead of C_k as seen in video 4, to avoid confusion with neuron k's index label; for g represents a general cost count over any change measured for the bias, weight or activation, and does not need to encompass neuron indices.) For the current neuron L, C will **store the derivatives of Cost** $(a^L - y)^2$ with respect to **changes in weights**: $\frac{\partial C_g}{\partial W^L}$ (notice W^L), and **changes in biases** $\frac{\partial C_g}{\partial B^L}$ (notice B^L), which incorporate the **changes in activations at L but not the changes in activations of prior layers** involving A^{L-1} introduced above (when I said "notice A^{L-1}")…**(***looking at the cost tree [at minute 5:49 in video 4](), **,** you see cost C_g as the bottom item in the tree, and an a^L as the item above it. Looking at the equation $\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_g}{\partial a^L}$ you may notice that a^L is the a^L in the cost tree too, which represents activation on current layer L, but not of the prior layer $L-1$**)**. **But why** compute **changes in activations** for some **prior layer**, and also **changes in activations** wrt **our current layer**? You **already know** why we computed **changes in activations** with respect to our **current layer**. (As seen in the discussion about at minute **5:49 in video 4** above) **But what the heck** will we do with those ***activation* changes computed for the *prior* layer**? We compute $\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial A^{L-1}}$, because it represents the change with respect to our activation in our **prior layer;** when it is time to **switch context** to some neuron, in our prior layer $L-1$, we compute $\frac{\partial C_g}{\partial w^{L-1}}$ (notice w^{L-1}) instead of $\frac{\partial C_n}{\partial w^L}$ (notice w^L), that is, our weight change in our new layer $L-1$ adjacent to our old current context neuron L in our prior layer. So, we don't use the **old** $\frac{\partial C_g}{\partial a^L}$ term (from $\frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_g}{\partial a^L}$) which pertained to our output layer L, but instead we utilize the new $\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial A^{L-1}}$ term, which

was computed when we were at some old current context neuron L. (In fact, going up in our change cost tree at minute 2:27 of video 4, you may notice that **instead of** a^L, we have A^{L-1}!!!) Of course, following the tradition, we then compute the cost activation change for our prior layer $L-2$ with respect to our current context neuron at $L-1$, so we can save it when we are ready to **switch contexts** again to some neuron from some prior layer $L-2$. It would look like this: $\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial A^{L-2}}$ notice A^{L-2} has changed from (A^{L-1}) , looking at the old equation term: $\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial A^{L-1}}$. Anyway, all these **changes aka gradients** will eventually be used to **alter the weights and biases and activations** in our neural net's structure. *(we don't change **activations directly**, they are computed from the structure of weights and biases, by transformation functions like σ on the **hypersurfaces** we defined in **Part I (1) above,** in some "forward-pass" process where we simply pass signals from input picture throughout the net without computing any costs, instead of the mutation sequence seen in **Part III - Error correction**)* So now you may notice that C doesn't actually exist as neurons/nodes or weights in our neural network, but its values are used to **affect the neural net's structure** of weights biases and activations. **Anyway,** for **more practical artificial neural nets that can do more sophisticated tasks like digit detection**, aka **multiple** neurons per layer neural nets; where each layer comprises of a **multiple** neurons, from **the context of some current neuron** j (*right away you notice we no longer refer to neuron by L, but j in L, as **multiple** neurons are in a layer here*)**:** As usual, we compute the "negative gradient" aka our cost $-\nabla C$ by equation

$$C = [\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_k}{\partial W_{j,k}^L}, \frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial B_j^L}, moreWeightChangeAverages..]$$

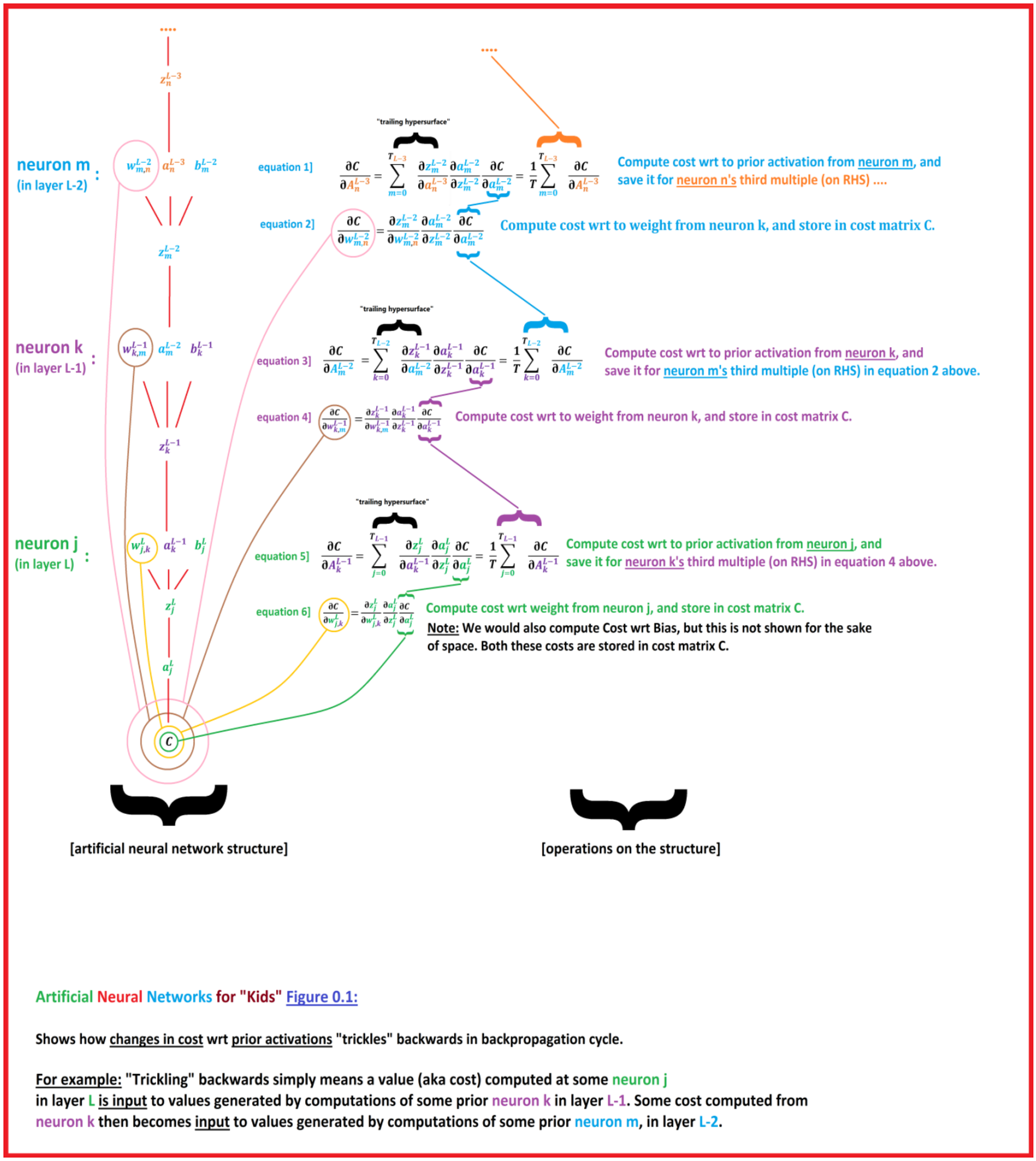
where we compute **changes in activations for the prior layer** L**:** $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$ (notice A_k^{L-1}) and store it in some variable, and save it for later use for some neuron in some prior layer $L-1$. For the current neuron L, C will **store the derivatives of Cost** $(a^L - y)^2$ with respect to **changes in weights**: $\frac{\partial C_g}{\partial W_{j,k}^L}$ (notice $W_{j,k}^L$), and **changes in biases** $\frac{\partial C_g}{\partial B_j^L}$ (notice B_j^L), which incorporate as usual, the **changes in activations but not the changes in activations of prior layers** involving A_k^{L-1}. (Similar to when we discussed the cost tree at minute *at minute 5:49 in video 4* for single neuron per layer neural nets above) **Anyway, by now, you know why we** compute **changes in activations** for some **prior layer**, and also

changes in activations wrt **our current layer;** In addition to **already knowing** why we computed **changes in activations** with respect to our **current layer,** you know that with ***activation* changes computed for the *prior* layer,** we compute something like $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$, because it represents the change with respect to our activation in our **prior layer;** so that when it is time to **switch context** to some neuron k in our prior layer $L-1$, we compute $\frac{\partial C_g}{\partial w_{j,k}^{L-1}}$ } (notice $w_{j,k}^{L-1}$) instead of $\frac{\partial C_g}{\partial w_{j,k}^{L}}$ (notice $w_{j,k}^{L}$), that is, our weight change in our new context layer $L-1$ adjacent to our old current context neuron j in layer L. In this **new scenario** or **context or neuron** from which we are measuring change, we don't use the old $\frac{\partial C_g}{\partial a_j^L}$} term (from $\frac{\partial z_j^L}{\partial w_{j,k}^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial C_g}{\partial a_j^L}$) which pertained to our output layer L, but instead we utilize the new $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$ term, which was computed when we were at some **old current context** neuron j in our output layer L (notice j in $\sum_{j=0}^{n_{L-1}}$). (Similar to when we discussed minute 2:27 of video 4, for single neuron per layer neural nets above) Of course, following the tradition, we then compute the cost activation change for our prior layer $L-2$ with respect to our current context neuron k at $L-1$, so we can save it when we are ready to **switch contexts** again to some neuron from some prior neuron p in $L-2$. It would look like this: $\frac{1}{n}\sum_{k=0}^{n_{L-2}} \frac{\partial C_g}{\partial A_p^{L-2}}$ notice $\sum_{j=0}$ has changed to $\sum_{k=0}$, and also, notice A_p^{L-2} has changed from (A_k^{L-1}) , looking at the old equation term: $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$. (You may notice that C_g doesn't change in our multiple neuron per layer neural net, from how it appeared in our single neuron per layer neural net, because this g is **not about layer L or neuron j versus layer $L-1$ or neuron k referencing**, but regards a general cost count, which will naturally cover any layer.) **Finally, notice that there is a slight difference** between **costs** for **single** neuron per layer neural networks: $\frac{1}{n}\sum_{j=0}^{n-1}$, and **multiple** neurons per layer neural nets: $\frac{1}{n}\sum_{j=0}^{n_{L-1}}$; as you can see multiple neurons per layer neural nets retain a n_{L-1} at the boundary of the sum, while single neuron per layer neural nets only sum up to $n-1$. This is because we simply don't need to care about considering the number of neurons per layer in a single neuron per layer neural network, so technically, layers don't really exist, each neuron in the single neuron per layer neural net acts as a layer. This is why we don't need the n_{L-1} to bound our sum in single neuron per

layer neural nets, as seen in the more complicated multiple neurons per layer neural net.

So, essentially I am saying that that:

- For single neuron per layer artificial neural nets, we have a "**simple hypersurface**" per computation on a neuron because we may sum in this way: $\{ z = (w^L \cdot a^{L-1} + b^L)\}$ with respect to some prior neuron, and a "**trailing hypersurface**" per computation on a neuron because we sum like this: $\{z_j = \left(w^L_{j,k} \cdot a^{L-1}_k + w^L_{j,k+1} \cdot a^{L-1}_{k+1} + b^L\right)\ldots\}$ with respect to prior neurons, for multiple neurons per layer artificial neural nets.
- The $\frac{1}{n}$ notation shows that we deal with **"training set averages"** which are actually our costs aka **"negative gradients"** $-\nabla C$. This cost occurs for both single neuron per layer artificial neural nets, $C = [\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_g}{\partial W^L}, \ldots]$ and multiple neurons per layer artificial neural nets: $C = [\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial W^L_{j,k}}, \ldots]$. For either case, the $\frac{1}{n}$ is needed, because we are summing over all training cases aka any number of input pictures processed so far, so we have to divide by n, where n corresponds to the total number of past training cases. Anyway, for more practical neural nets, which are the multiple neurons per layer artificial neural nets, we do the same.



(See high quality version of the diagram: https://imgur.com/NjT9EDt)

Also, notice that in the image above, at particular sums, I use notation $\frac{1}{T}$, where T is the number of training samples. In contrast, $\frac{1}{n}$ is used throughout ***Part I*** prior to the image above per sum, because there was no $n'th$ neuron discussed prior to the image above, and so n could be used distinctively for the training sample total without confusion with neuron indices.

Part II - "Partner neuron sums - An emphasis on “trailing hypersurfaces”:

This part will refer to **Part I** section (1), in regards to the discussion about where we “constrain to $L-1$”.

This part is just to emphasize that we have **trailing hypersurfaces** in **multiple neuron per layer neural nets** (which are useful neural nets for tasks like digit detection).

Unlike single neuron per layer neural nets, multiple neuron per layer neural nets require trailing hypersurfaces (instead of a single hypersurface) for any neuron sharing weights with any other neuron in a layer.

Considering some “**current neuron**” j, aka a neuron from which change may be computed with respect to neighbouring neurons $j+n$ in the same layer, affected by some number of neurons $k+n$ in our prior layer $L-1$; we can think of these computations as involving **partner neurons** (where j and some number of neurons $j+n$ may be “partners” through some number of prior neurons $k+n$**)**, that partner up by sharing the burden of hypersurface computation, which involves a sum that includes:

- **Neurons** $k+n$ connected by **weights** $j, k+n$**,** which connect to a current layer **neuron** j, ***from which change is being measured.***
- Some **bias** associated with **neuron** j above, pertaining to layer L.

- **Multiple activations** relating to **weights** $j, k+n$ connecting to **neurons** $j+n$ in the same layer as **neuron** j above. (Where **neuron** j is also connected to any **weights** connecting to **neurons** $j+n$).

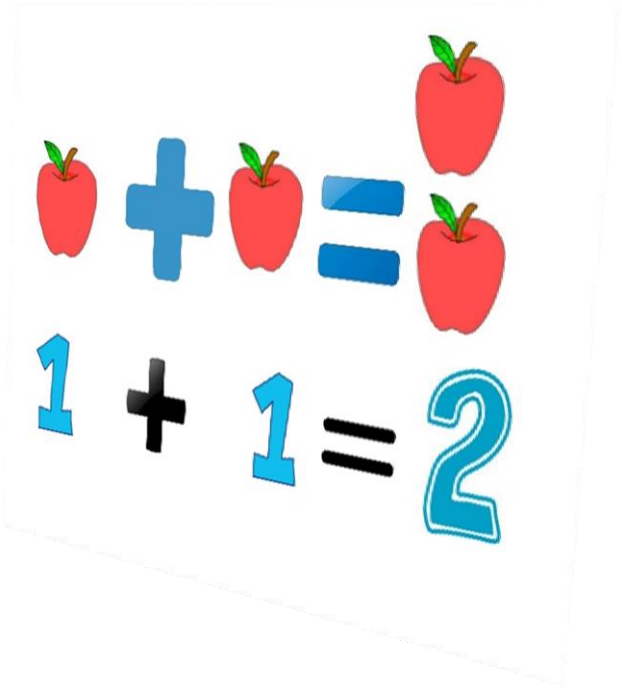
Part III - "Error correction - Application of costs from the trailing hypersurfaces":

Now, the whole reason for the cost function computation, and the back propagation algorithm, was to make our artificial neural net learn **better weights** in order to produce **better and better answers**; hence slowly but surely **removing its mistakes** as it is exposed to more and more examples of correctly labelled input data. ***(Note that although I just now mentioned this step, it is done iteratively together with the steps above)***

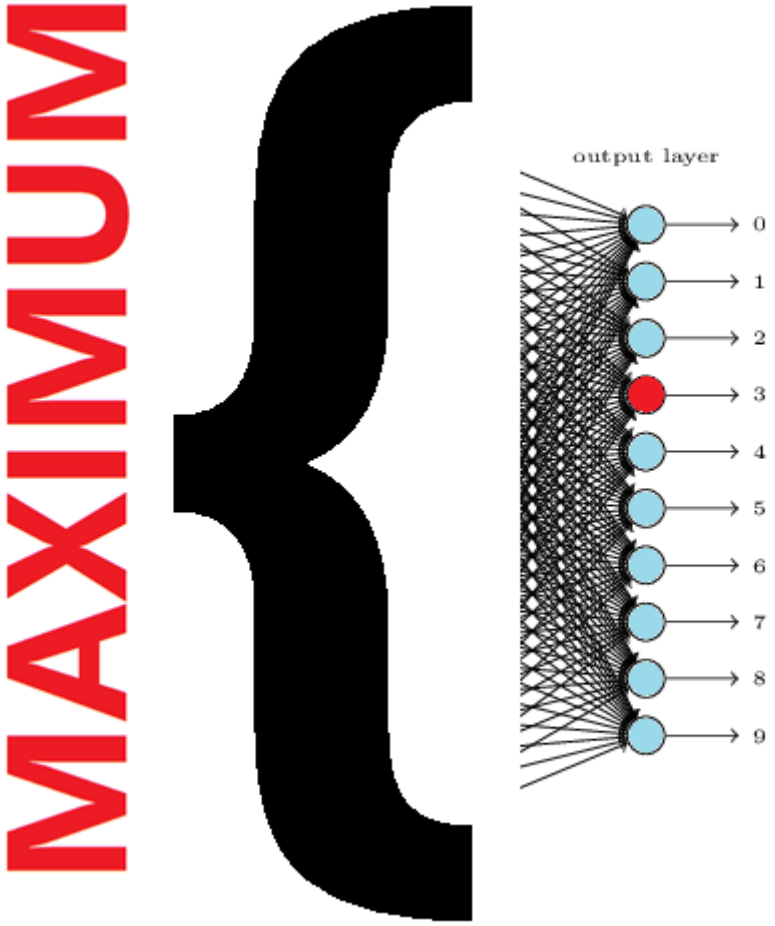
1. Remember the cost function was the negative gradient of the error contained in matrix C.
2. At each cycle's end, we would have computed the changes w.r.t. to the cost function, ∇C, and placed a sequence of changes aka "**negative gradients**" in C. (**Reminder**: Producing regular negative gradient is **steepest descent** in the direction of better and better answers, and producing mini-batch based negative gradient is **stochastic gradient decent**)

3. Remember also that there is another crucial matrix, our matrix of weights M, which we can say contained our **weights**, but also related **biases** and **activations**!!!! (This is why we can add our list of changes in weights, biases, and activations $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial W_{j,k}^{L}}, \frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial B_{j}^{L}}, \frac{1}{n}\sum_{j=0}^{n_{L-1}}$ in C to M, because M would have contained the corresponding weights biases and activations of our neural network. Remember, C doesn't actually exist as neurons/nodes or weights in our neural network, but merely as a list of changes or cost which we must incorporate into our model.) So, although we compute cost change at term A_k^{L-1} (seen in $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$), we don't actually store it in C, instead it was used as explained in Parts I item 2.
4. Computing the cost matrix C showed where the neural net went wrong, but the neural net itself is unchanged, and has not learnt from its mistakes yet.
5. To enable the neural net to change its dirty old erroneous ways, we propagate the **error signal back into the network.** (i.e. back propagate) To propagate the error signal backwards, is another way of saying we adjust the weights M of the neural network, such that that mistakes are reflected in the error in terms of C. So we simply add C to M, i.e. $M = M + C$. (**Note: Looking on the step 2**, although the contents of C are called "**negative gradients**", we don't **intentionally** create a subtraction equation $M = M - C$. **Crucially**, if we decided to update the weights by subtraction equation, $M = M - C$ we would not be updating the weights by the correct changes, because then if a value in C is negative, an addition would happen, and that **negative change aka negative value would not be reflected** in the update**.** Instead, creating an addition equation $M = M + C$ for weight update, ensures that we update the weights correctly, changing our weights in terms of the actual values of the cost matrix C, whether that value of C is positive or negative. Notice that even if a value in C is negative, in our **addition equation** that negative value **will still be subtracted**, as positive plus a negative is actually a subtraction. The name "**negative gradients**' simply applies because the cost function that populates matrix C is demonstrably observed in literature and practice to **decrease** over some range of input as our derivative processes in Parts I, II, then the **additive process** here in Part III occur. This **decrease** gives us the name of our gradients in matrix C.
6. Looking at a cost change tree, at minute 2:23 of video 4, although that tree referred to **single** neuron per layer neural net if we **pretend** that the

tree contained the indices for a **multiple** neuron per layer neural net, **then another crucial thing to take note of**, is that we compute triplets **of costs per cycle,** including **changes in weights:** $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial Wj,k^L}$, **changes in biases:** $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial B_j^L}$**,** and finally **changes in activations for prior layer** $L-1$: $\frac{1}{n}\sum_{j=0}^{n_{L-1}} \frac{\partial C_g}{\partial A_k^{L-1}}$. You already know that these changes in activations for or prior layer is saved for computation of changes in **biases** and **weights** when we **switch contexts** to some **prior layer**. (As discussed in ***Part I, item 2***) Anyway, our **changes in activations** on $L-1$, **containing the** A_k^{L-1} term**, is what triggers the "trailing hypersurface"** phenomena (This is because for **multiple** neuron per layer neural nets, z looks like this: $z = (w_{j,k}^L \cdot a_k^{L-1} + w_{j,k+1}^L \cdot a_{k+1}^{L-1} + b^L)$ instead of this: $z = (w^L \cdot a^{L-1} + b^L)$, as seen in **single** neuron per layer neural nets). As such, this "**trailing hypersurface"** computation **of changes/averages on activations** (of prior layer $L-1$) generate some "**partner neuron**" relationship. (*See Part II***)** The quick explanation is that we can see from looking at **single** neuron per layer neural nets, that the "**partner neuron**" relationship needs to occur in our **multiple** neuron per layer neural net, because now instead of considering **merely** the activations from prior layers $L-1$ (as seen in single neuron per layer neural nets), we must also now consider activations that affect any neighbouring neuron (or neuron in the same layer as our current context neuron) relating to any other neuron at a point from which change is being measured. So in the end, in **multiple** neuron per layer neural nets, we must apply this "**partner neuron**" relationship, as we are dealing with many neurons per layer per **training case** aka each time the neural net is exposed to a picture of a digit. We would still compute triplets for **single** neuron per layer neural nets too, (as seen in a cost change tree, at minute 2:23 of video 4) but we would have a small number of cost triplets computed, proportional to that small number of neurons. Typically, the larger the neural network, the more triplets will be required.



So after the above process is done (i.e. many cycles of M updates in $M + C$ terms, where every time this addition happens, we step **closer and closer** to a better neural network), the artificial neural net will be able to guess what **unlabelled** input is saying. In this way the neural net has **generalized** beyond the "training data" or "correctly labelled samples" it's seen in steepest descent, and thus it has learnt the task of digit detection! Now **you see** that baseline neural nets are just **many many additions of big** matrices of data (comprised of **function compositions** i.e. derivatives over many ratios with respect to the weights and activations)!!



But how do we know our neural net has "generalized" beyond the training set (aka how do we know whether the neural net actually learnt to detect digits from the input pictures)? We know when we actually try to "test" our neural net. When we are done correcting our weights, w.r.t. to some correctly labelled inputs aka training examples, we can now test our model by taking a value from the **output layer**, where our **answers** are stored.

We test our model by showing it a picture of a digit that was not a part the training set, to see if it really did learn (Showing it a picture of a digit in the training set would not entirely reflect if it really did learn…so to be sure, it's better to expose it to sets of pixels aka images of digits it's **not yet seen…**then again, the entire point is to make a digit detector, not just a digit detector on predefined examples of correctly labelled images, but a digit detector overall, that is something that can detect **freshly seen digits**, in particular, unlabelled digits!!!! Would it make sense if we always showed our artificial neural net only correctly labelled images aka **always tell it what each image is**

representing?? How then would we know it has learnt anyway???? **So this is why we test on fresh unlabelled data of unseen pictures of digits**).

Now, for an input with a digit 3, if steepest descent aka gradient descent worked out well, although all **output neurons aka answer neurons** would have a value, the neuron for 3 would have an especially higher value than the other neurons, because the neural net thinks it's seeing a 3 (i.e. counting from the zeroeth neuron, we find the **answer neuron for 3** in the fourth neuron, because the neuron count started at 0).

So to get our final answer, all we do is take the maximum of the set of numbers stored in our output layer! (where each **answer/output neuron** has a number)

Review

Although all the nuts and bolts of an elementary digit-detection-capable artificial neural net are easy to understand in the long run, the neural net unfortunately consists of thousands of moving parts, so it is perhaps tedious to grasp the whole picture.

▼

As such, the entirety of an elementary yet powerful artificial neural network can be compacted into merely **3 parts:**

1) "Part I — Trailing hypersurfaces" & "Training set averages":

https://i.imgur.com/yCNJo99.png

2) "Part II — Partner neuron sums — An emphasis on "trailing hypersurfaces":

https://i.imgur.com/yBmDBYT.png

3) "Part III — Error correction — Application of costs from the trailing hypersurfaces":

https://i.imgur.com/PMohvFy.png

▼

Notably, Part II is merely a way to clarify part I, so basically the neural network is just **2 things**:

1) A sequence of "**hypersurface**" computations wrt to some cost function.

2) An **application of costs** aka "negative gradients" (using the **hypersurface** computations) to update the neural network structure as it is exposed to more and more training examples. And thus the neural net improves over time. (aka **error correction**)

That's it, that's the infamous back-prop algorithm, together with an entire elementary but powerful artificial neural network!!

Now you're ready to take on Geoffrey Hinton's "hard" online course in neural nets!!

Thanks

If your mind fancied that scrumptious thought food, please leave a thoughtful review.

Your support is greatly appreciated.

Also, don't forget to share the book with a friend.