Continuous Classification using Deep Neural Networks

Nick Strayer

2017-12-08

Contents

Chapter 1

Introduction

1.1 Continuous Classification

Imagine you are watching a movie. A friend walks in late and asks "what did I miss?" You tell them the main character has just escaped from a nasty predicament and has defeated the antagonist. What you have done is classification on a sequence. The sequence in this case is the frames of the movie and your classification was what was occurring in the movie at that moment. You *could* have given the same answer if you just saw a single frame, but most likely your assessment of the state of the movie depended on events you saw before and the context in which they placed the most recent frame.

Continuous classification in the context of statistics and machine learning is training models to observe data over time, like you watched the movie, and classify the status of the generating system at any given point. Sometimes seeing the most recent data is all that is needed, but more interesting and challenging problems need the algorithm to be able to make decisions about a current time while leveraging context from previous history to do so.

This report is a brief run through past attempts at continuous classification and a deeper exploration of the current state of the art methods.

1.2 Potential applications of continuous classification models

The following are just a few examples of biomedical applications made possible with effective continuous classification models.

1.2.1 Activity Prediction

With the advent of wearable devices such as fitbits and apple watches, the amount of high temporal resolution data we have streaming from individuals is exploding and showing no sign of letting up.

Continuous classification models could use these data to classify the state of the wearer at any moment. A simple example of this is detecting different exercise types (e.g. running vs. swimming); which is implemented (by unpublished methods) internally at companies such as fitbit.

More advanced, and potentially impactful, applications include extending the predictions to more subtle but medically relevant states such as dehydration or sleep apnea (**?**). Preliminary work in these areas using deep learning has shown surprising success with data as limited as heart-rate and motion indication being enough

to predict sleep apnea and various cardiovascular risk states with a c-statistic of 0.94: comparable to invasive gold standards.

1.2.2 EHR monitoring

With more and more information on patients being accrued in government and hospital databases we have a clearer than ever picture of a patient's health over long periods of time. Unfortunately, due to a combination of overwhelming quantities and noise levels in the data, our ability to make use of these data has not kept up with their quantity.

Sequential models can help ease the burden on health practitioners in making use of these data. For instance, a model could be trained on a patient's records to predict the likelihood of cardiovascular events. This model could then alert a doctor of potential risk in order to facilitate timely interventions. This could be especially helpful in large clinical settings where personal doctor-patient relationships may not be common. For a review of the performance of deep learning models in electronic health record contects, see **?**.

1.2.3 Hospital Automation

Patient monitoring systems already have alarms to alert staff of occurring anomaly for a patient. Continuous classification methods could extend these systems to warn *before* the anomaly occurs (e.g. patient has a high change of going into afribulation in the next five mins), or to more subtle actions (patient is experiencing pain and needs a change in the medications administered by their IV). These methods, if successfully implemented could help hospitals more efficiently allocate resources and potentially save lives.

1.3 History of methods

While sources of data well suited to it have recently greatly expanded, interest in performing continuous classification is not a new topic. Many methods have been proposed for the task to varying degrees of success. Below is a brief review of some of the more successful methods and their advantages and limitations.

1.3.1 Windowed regression

Perhaps the most intuitive approach to the problem of incorporating context from previous time points into your prediction is to use a windowed approach. Broadly, in these approaches a window of some width (in previous observation numbers or time length) is sequentially run over the series. The data obtained from the window may have some form of summary applied to it. This could be a mean, median, or any other function which is then used to predict with.

By summarizing the multiple data-points into a single (or few) values noise can be removed, but at the cost of potentially throwing away useful information captured by the interval (such as trajectory.)

If the data are kept intact more advanced methods are available. These include dynamic time warping (**?**) or kernel methods (see next section). This allows more information to be retained in the sample but at the cost of setting a limit on how far back your model can learn dependencies in the data. For instance, if your window is one hour long but an activity lasts two hours your model will have a very hard time recognizing it. This is equivalent to an infinitely strong prior on the interaction timeline (**?**).

1.3.2 Transformation methods

As mentioned before, when a window is scanned across the time dimension of data, one of the ways of extracting information is by performing some transformation on the data. Common examples include wavelet

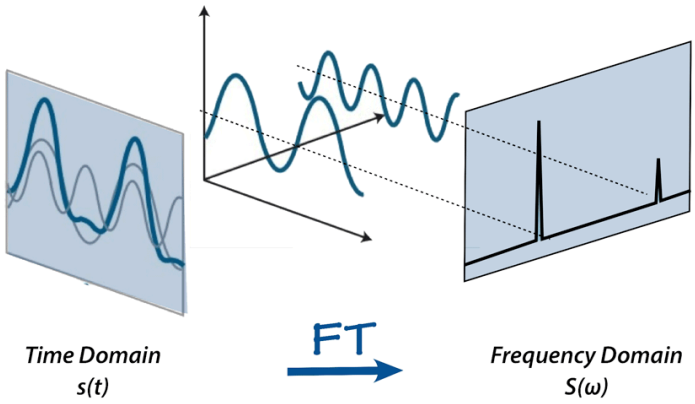

Figure 1.1: Example of transforming data from the data-domain to the frequency-domain for time series data. Image courtousy of [Allen D. Elster, MD FACR](http://mriquestions.com/index.html).

or Fourier transforms. These methods attempt to separate the data into separate components. For instance, Fourier transforms applied to accelerometer data from an individuals wrist can be used to detect the frequencies associated with walking and running (**?**). These methods have also been used extensively in electrical systems and signal processing to help determine the state of the system.

A few limitations are imposed by these methods. First, as previously mentioned, they are subject to the windowing constraints. Secondly, they rely on the data to be periodic or oscillatory in nature. For instance, accelerometer data oscillates back and forth as the individual swings their arms and electrical systems are inherently oscillatory. Data such as heart-rate or step counts produced by devices like apple watches and fitbits are a rather stable signal [1] and thus transformation methods are unable to separate them into frequency domains at small time scales. In addition, these methods are unable to deal with non-numeric data which severally limits them in heterogeneous data domains such as EHR data.

1.3.3 Hidden Markov Models

In an attempt to deal with the fact that in most scenarios the classification of time point t is dependent on that of previous time points, hidden Markov models (or HMMs) model data as a series of observations generated by a system transitioning between some unobserved (or latent) states. This is done by constructing a transition matrix that denotes the probability of transitioning from one state to another and conditioning it on whatever observed data you have.

$$P(s_a -> s_b | x_t) = ...$$

This allows the model to learn time dependencies in the data. For instance, if a person is running now their next state is probably going to be walking rather than sitting or swimming.

HMMs were the state of the art models on continuous classification problems until very recently and are still very valuable for many problems. However, their greatest advantage is also their greatest disadvantage.

The Markov property (or the first 'M' in HMM) states that the next state of the system being modeled depends exclusively on the current state. This means that the model is 'memory-less.' For instance, returning to our running example, say an individual had been running in the previous time point, the model will most likely pick walking as their next state (ignoring any conditional data for simplicity) but what if before they were running they were swimming? This fact from multiple time-steps before would strongly hint that the next state would in fact be biking and not walking (they are running a triathlon.)

[1] Although the raw data the sensors receive may not be.

There are ways to fix this such as extending the model's transition probabilities to multiple time-steps, however the number of parameters needed to estimate transition probabilities for m previous time steps is (# of classes) to the k^{th} power, which rapidly becomes untenable. In addition, we have to a priori decide the number of time steps in the past that matter.

1.3.4 Advantages of deep learning methods

Before we dive into the mathematical underpinnings of deep learning methods we will go over how they solve many of the aforementioned issues from traditional methods.

1.3.4.1 Less Domain Knowledge Needed

One of the ways that it helps to think about deep learning is as a computer program that programs itself given an objective and examples. In his popular blog post *Software 2.0* Andrej Karapathy makes the argument that deep learning is powerful because it helps avoid traditionally tedius processes like explicitly defining cases for the computer to deal with. One of the ways this is applicable to our problems is the ability for deep learning models to adapt to a wide range of problem/ data domains without much human-defined customization.

This can be seen in the context of the input data form. If you had data from an accelerometer it could be fit into the same neural network as data from a more static heart-rate sensor would. The models are flexible enough to learn how to deal with these input patterns without requiring the researcher to explicitly define a transformation based on the data. One advantage of this independence from large amounts of human intervention has the potential to make performance assessments more accurate (**?**).

1.3.4.2 Can find and deal with arbitrary time dependencies

Deep learning models are theoretically capable of learning time dependencies of infinite length and strength (**?**). While obviously it is impossible to supply a network with enough data to fit the number of parameters neccesary to do so, the fact remains that deep learning methods are capable of handling long-term time dependencies. In addition to being able to model these dependencies they do so without any need for explicitly telling the model the length of the dependencies and also using substantially fewer parameters than an extended hidden Markov model (**?**).

For example, a recurrent neural network (RNN) can automatically learn that if a person swims and then runs, they will most likely be biking next, but it could also remember that a patient was given a flu vaccine three months prior and thus their symptoms most likely don't indicate the flu but a cold. This flexibility to automatically learn arbitrary time dependency patterns is powerful in not only its ability to create accurate models, but potentially for exploration of causal patterns.

1.3.4.3 Multiple architectures for solving traditional problems

In a similar vein, one of the decisions that does need to be made with deep learning: which network architecture to use, conveniently is rather robust to the problem of continuous classification. For instance: convolutional neural networks that have achieved great success in computer vision were actually originally designed for time series data, and recent advanced such as dilated convolutions (**?**) allow for them to search as far back in the time-series as needed to find valuable information for classification. Recurrent neural networks (which will be elaborated on in the following sections) are also fantastic for time-series data, as they explicitly model the autocorrelation found in the data via a recurrent cycle in their computation graph. This allows them to read data much like one reads a book, selectively remembering past events that have applicability to the current state.

1.3.4.4 Downsides

As a result of being so flexible deep learning models require a lot of data to properly tune all their parameters without over fitting. This results in not only more data being needed (with some exceptions such as Bayesian methods) but also, when combined with their non-convexity, requires a large amount of computation power.

Another side effect, although one shared by many other approaches described here, is that neural networks are not amenable to inference on specific factors contributing to their classifications.

These and other downsides and potential solutions are explored in the last chapter.

In the next chapter we will go over the basics of modern deep neural networks.

Chapter 2

Neural Networks

> A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler function. We can think of each application of a different mathematical function as providing a new representation of the input. (**?**)

Neural networks (sometimes referred to as multilayer perceptrons) are at their core very simple models. Traditional modern neural networks simply pass data forward through a "network" that at each layer, performs a linear (also referred to as affine) transformation of its inputs followed by a element-wise non-linear transformation (also called an activation function). In doing this they can build up successively more complex representations of data and use those to make decisions about it.

This can be thought about in the analogy of recognizing a cat. First you see ears, a nose, two eyes, four feet, and a fluffy tail; next, you recognize the ears, nose and eyes as a head, the tail and legs as a body; and lastly the head and body as a cat. In performing this 'classification' of a cat you first constructed small features and successively stacked them to figure out what you were looking at. While this is obviously a stretched definition of how neural networks work, it actually is very close to how a special variant called Convolutional Neural Networks work for computer vision techniques (**?**).

2.1 History

While neural networks' popularity has taken off in recent years they are not a new technique. The neuron or smallest unit of a neural network was first introduced in 1943 (**?**). It was then another 15 years until the perceptron (now commonly called 'neural network') was introduced (**?**) that tied together groups of neurons to represent more complex relationships.

Another ten years later, in a textbook (**?**) it was shown that a simple single layer perceptron was incapable of solving certain classes of problems like the "And Or" (XOR) problem (2.1), due to their being linearly inseperable. The authors argued that the only way for a perceptron to overcome this hurdle would be to be stacked together, which, while appealing, was not possible to be trained effectively at the time.

It wasn't until 1986 that a realistic technique for training these multi-layer perceptrons was introduced (**?**). Finally all of the algorithmic pieces were in place for deep neural networks, but interest stagnated due to the computational expense of training the networks, a lack of data, and the success of other competing machine learning algorithms.

Interest in the field of deep learning has had a massive resurgence in the second decade of the 21st century. Driven by growing stores of data and innovations in neural network architectures. One commonly cited tipping point for the current "deep learning revolution" was the 2012 paper (**?**) in which a deep convolutional neural network won the ImageNet prize and showed massive improvements over traditional methods.

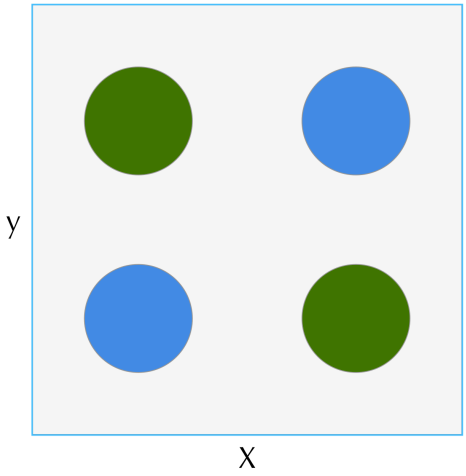

Figure 2.1: Example of the XOR problem. Classes encoded by color are not linearly seperable.

2.1.1 Biological Inspirations

The word 'neural' in the 'neural network' is reference to the fact that these models derive inspiration from how the brain works. With the individual nodes in a hidden 'layer' frequently being called a 'neuron'. While the broad concepts may be similar between the way animal brains and neural networks work, it is important to note that the similarities end approximately at the network-ness of both systems. There has however, been some more recent work on trying to more closely mimic the brain structure with architectures such as capsule networks (**?**). In addition, neuroscience experiments have demonstrated that at least part of our visual system does truly perform these hierarchical stacks of features when recognizing objects (**?**).

2.1.2 Geometric Interpretation

Another way of thinking of how neural networks work is as a building up a series of successive transformations of the data-space that attempt to eventually let the data be linearly separable ((2.2). In this interpretation each layer can be seen as a shift and rotation of the data (the linear transformation), followed by a warping of the new space (the activation function). In his excellent blog post: Neural Networks, Manifolds, and Topology, Chris Olah gives an excellent visual demonstration of this.

2.2 Universal Approximation Theorem

One powerful theoretical result from neural networks is that they are universal approximators (**?**). A neural network with a single hidden layer and non-linear activations functions on that layer can represent *any* borel-measurable function. This result means that there are no theoretical limits on the capabilities of neural networks. Obviously, in real-world situations this is not the case. We can not have infinite width hidden layers, infinite parameters requires infinite data, and even more limiting are our inefficient learning methods. All constraints considered though, the universal approximation theorem does provide confidence that, as long as they are properly constructed and trained, neural networks are amazingly flexible models.

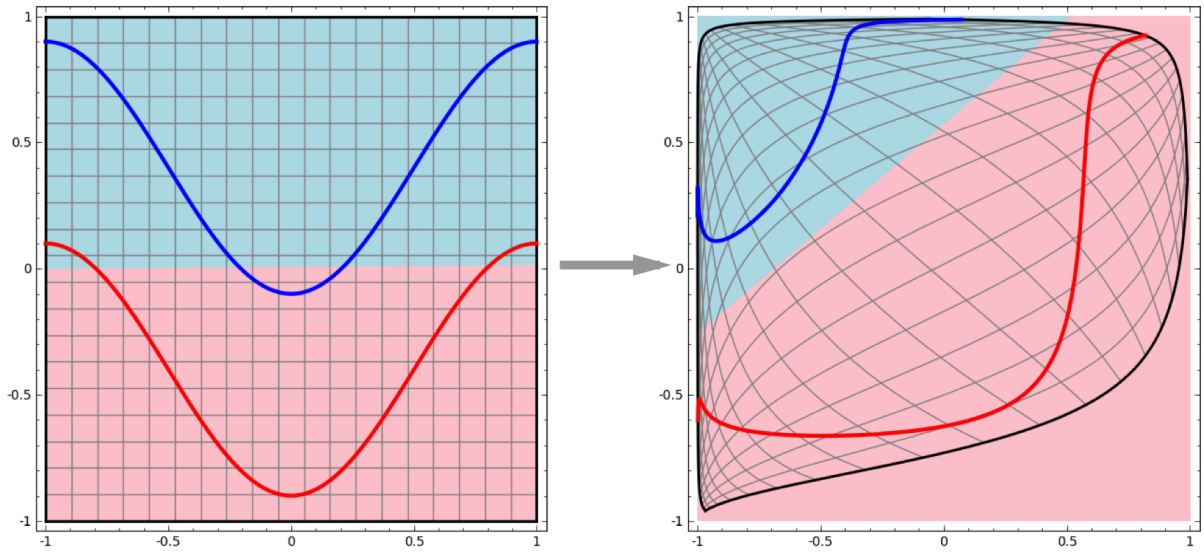

Figure 2.2: Example of how a neural network can, through a series of affine transformations followed by non-linear squashings, turn a linearly inseperable proble into a linearly seperable one. Image courtesy of [Chris Olah's blog](http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/)

2.3 The Computation Graph

While the building blocks of neural networks are simple, often complete models are composed of hundreds to even thousands of neurons and millions of connections and representing them in mathematical notation becomes exceedingly difficult. A method of dealing with this complexity, along with also helping in the intuition of many other properties, is to represent the networks as a 'computation graph.'

A computation graph is simply directed acyclic diagram that shows the flow of data through the model. Each neuron is usually represented as a circle with the weights both in and out of the neuron's value represented as edges.

Sometimes, when the models get even larger, the layers (or groups of neurons) will get lumped into a single node in the graph (as on the right of the figure.)

2.4 Terminology

When covering the basic mathematical operations of a neural network it helps to have a reference for some of the terms that get used. This list provides the most commonly used terms for the models we will be describing.

Neuron: An individual node in the network. Has two values: activation, or the value of the linear function of all inputs, and the post-activation-function value, or a simple transformation of the activation by the activation function.

Affine Transformation: A linear transformation of an input (either data input or a hidden layer's output). Essentially a linear regression.

Bias Term: A constant term added to the affine transformation for a given neuron. Also known as an 'intercept term.'. For notational simplicity in most of our formulas we will omit this.

Activation Function: A non-linear function that takes an input and 'squashes' it to some range. A common activation function is the sigmoid, which takes an unbounded real-valued input and returns a value between -1 and 1.

Layer: A collection of neurons who's inputs typically share the same inputs (either another layer's output or the data).

Hidden Layer: A layer who's input is the output of a previous layer and who's output is another layer. E.g. Input layer -> hidden layer -> output layer.

2.5 Mathematical Operations

The basic operations that one does on a neural network really fall into two categories. Forward propagation, or the passing of data into and through subsequent layers of the model to arrive at an output, and back-propagation, or the calculation of the gradient of each parameter in the model by stepping back through the model from the loss function to the input. For a more thorough treatment of these steps see **?** chapter six.

2.5.1 Forward Propagation

Let a neural network with l layers and k dimensional input X and m dimensional output $\hat{y}$ attempting to predict the true target y. Each layer is composed of $s_i, i \in \{1, 2, ..., l\}$ neurons and has respective non-linear activation function f_i. The output of a layer after affine transformation is represented as a vector of length s_i: $\underline{a_i}$ and the layer ouput vector post-activation function outputs: $\underline{o_i}$. The weights representing the affine

transition from one layer i to layer j are a matrix W_i of of size $s_i \times s_j$. Finally the network has a differentiable with respect to $\hat{y}$ loss function: $L(\hat{y}, y)$.

Forward propagation then proceeds as follows.

1. Input X $(1 \times k)$ is multiplied by W_1 (size $(k \times s_i)$) to achieve the *activation values* of the first hidden layer.
   - $X \cdot W_1 = \underline{a_1}$
2. The $(1 \times s_1)$ activation vector of the first layer is then run element-wise though the first layer's non-linear activation function to achieve the output of layer 1.
   - $\underline{o_1} = f_1(\underline{a_1})$
3. This series of operations is then repeated through all the layers, (using the subsequent layers output vector as the input to the next layer,) until the final layer is reached.
   - $\underline{o_i} = f_i(\underline{o_{(i-1)}} \cdot W_i)$
4. Finally, the loss is calculated from the output of our final layer.
   - $L_n = L(\underline{o_l}, y) = L(\hat{y}, y)$

While not strictly necessary for forward propagation, the intermediate layer activations and output vectors are kept stored so they can be used in the later calculation of the gradient via back propagation.

2.5.2 Back Propagation

If we are just looking to gather predictions from our model we can stop at forward propagation. However, most likely we want to train our model first. The most common technique for training neural networks is using a technique called back propagation (**?**). In this algorithm the chain rule is used to walk back through the layers of the model starting from the loss function to the input weights in order to calculate each weight's gradient with respect to the loss. This gradient is then descended using any number of gradient descent algorithms.

2.5.2.1 The Chain Rule

Back propagation is nothing more than a repeated application of the chain rule from calculus. Let x be a single dimensional real valued input that is mapped through first equation f and then g, both of which map from a single dimensional real number to another single dimensional real number: $f(x) = z, g(z) = y$ or $g(f(x)) = y$. The chain rule states that we can calculate the derivative of the outcome with respect to the input by a series of multiplications of the derivatives of the composing functions.

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx} \tag{2.1}$$

This single dimensional example could be thought of as a neural network composed of two layers, each with a single dimension. The single dimensional case is illuminating, but the value of the chain rule comes when it is applied to higher dimensional values.

2.5.2.2 Expanding to higher dimensions

Now, let $\curvearrowleft \in \mathbb{R}^m$ and $\digamma \in \mathbb{R}^n$. The function f maps from $\mathbb{R}^m \to \mathbb{R}^n$ and $g : \mathbb{R}^n \to \mathbb{R}$. Further, let $\mathbf{z} = f(\mathbf{x})$ and $y = g(\mathbf{z})$. The chain rule can then be expressed as:

$$\frac{dy}{dx_i} = \sum_j \frac{dy}{dz_j}\frac{dz_j}{dx_i} \tag{2.2}$$

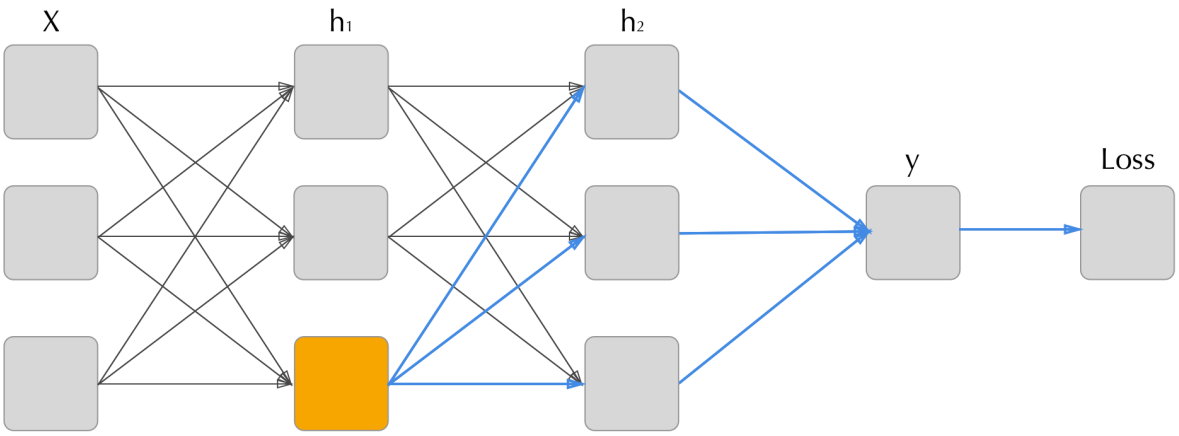

Figure 2.3: How back Propagation steps back from the output to the input. To calculate the gradient with respect to the loss of the orange neuron the we need to aggregate the gradients of all connected points further along in the computation graph (blue connections).

Or that the derivative of y with respect to the i^{th} element of $\mathbf{x}$ is the sum of the series of products of the derivatives of result $\mathbf{z}$ that sits between the two values in the function composition.

Another way of thinking of this is, the derivative of the output of the function composition $f \circ g$ with respect to some element of the input is the sum of all of the derivatives of all of the paths leading from the input element to the output.

2.5.2.3 Applied to Neural Networks

To apply this technique to neural networks we need to make sure all components of our network are differentiable and then walk back from the loss function to the last (or output) layer, calculating the gradients of the output layer's neurons with respect to the loss. Once we have calculated the gradients with respect to the weights of the output layer we only need use those calculated gradients to calculate the gradients of the preceding layer. We can then proceed layer by layer, walking back through the model filling out each neuron's weight gradients until we reach the input.

To calculate the gradient of the weights for hidden layer i we can recall that the hidden layers output can be represented as $\mathbf{a}_i = f_i(\mathbf{W} \cdot \mathbf{a}_{(i-1)})$ (we're omitting the bias term here for simplicity). Thus to calculate the gradient's on the weights we can set $\mathbf{g}_i^* = \mathbf{g}_{i+1} \odot f'(a_{i+1})$ to be the gradient un-activated by our layer's activation function's derivative. Then to find the derivative with respect to each neuron's weights within the layer we multiply this un-activated gradient by the transpose of the layer's output vector: $\mathbf{g}_i = \mathbf{g}_i^* \mathbf{o}_i^t$.

The fact that the calculation of these gradients is so simple is fundamental to deep learning. If it were more complicated, extremely large networks (such as those used in computer vision) with millions of parameters to tune would simply be computationally infeasible to calculate gradients for. Conveniently, the run time of the back propigation is a simple product of the number of neurons in each layer and the number of layers in the whole model.