Final Project Code

Any code, calculations, graphics, etc. that need to be submitted as part of the capstone paper should be submitted within this notebook.

Student Response

References:
Primary resource, referred to throughout as 'Paper':
https:/ / arxiv.org/ pdf/ 1802.01528.pdf
The Matrix Calculus You Need For Deep Learning
Terence Parr and Jeremy Howard July 3, 2018
Intro - Perceptron: https:/ / becominghuman.ai/ from-perceptron-to-deep-neural-nets-504b8ff616e
Intro - Affine function: https:/ / math.stackexchange.com/ questions/ 275310/ what-is-the-difference-between-linear-and-affine-function
Intro - Affine activation: https:/ / arxiv.org/ pdf/ 1603.07285.pdf
Intro - Non-linear activation: https:/ / stackoverflow.com/ questions/ 9782071/ why-must-a-nonlinear-activation-function-be-used-in-a-backpropagation-neural-net
Intro - Udacity: https:/ / github.com/ udacity/ deep-learning-v2-pytorch/ blob/ master/ intro-to-pytorch/ Part%202%20-%20Neural%20Networks%20in%20PyTorch%20(Solution).ipynb
Intro - Perceptorn: http:/ / ataspinar.com/ wp-content/ uploads/ 2016/ 11/ perceptron_schematic_overview.png
Intro - Dot product: https:/ / stats.stackexchange.com/ questions/ 291680/ can-any-one-explain-why-dot-product-is-used-in-neural-network-and-what-is-the-in on the dot product
4 - http:/ / web.mit.edu/ wwmath/ vectorc/ scalar/ intro.html
4 - http:/ / tutorial.math.lamar.edu/ Classes/ CalcIII/ VectorFunctions.aspx
4.5 - QuickSort: https:/ / www.geeksforgeeks.org/ quick-sort/
4.5 - QuickSort: https:/ / www.tutorialspoint.com/ data_structures_algorithms/ quick_sort_algorithm.htm
General:
Backpropagation & Neural Networks
http:/ / cs231n.stanford.edu/ slides/ 2017/ cs231n_2017_lecture4.pdf
Fei-Fei Li & Justin Johnson & Serena Yeung Lecture 4 - 1 April 13, 2017
https:/ / atmos.washington.edu/ ~dennis/ MatrixCalculus.pdf
=https:/ / arxiv.org/ pdf/ 1603.07285.pdf
https:/ / www.comp.nus.edu.sg/ ~cs5240/ lecture/ matrix-differentiation.pdf
Level Curves: https:/ / mathinsight.org/ level_sets

Student Response

Summary of Approach: For this capstone, I explore the fundamental mathematics of applied deep learning by working to understand how matrix calculations produce convolutional neural networks. To do so, I use resources from established machine learning practicioners for a series of mini-investigations into how neural nets achieve pattern recognition.

Throughout the paper *"The Matrix Calculus You Need For Deep Learning"* is excerpted at key points as 'Paper' , and additional resources used to provide a fuller understanding of the concepts involved, with visualization and sample formulae added intermittently. The paper is in "Blue" (except where embedded formulae object) and supporting resources are quoted in "Red" .

Student Response

Paper: Introduction, pg 3

"For example , the activation of a single computation unit in a neural network is typically calculated using the dot product (from linear algebra) of an edge weight vector w with an input vector x plus a scalar bias (threshold): $z(x) = Sum\ wixi + b = w \cdot x + b$ "

Comment:

The single neural node is also referred to as a 'perceptron' . It is the summing of the results of multiplying a series of

weights W_i by an input value X_i plus a bias value b added after the summing. This is the equivalent of simplying multiplying a vector W composed of all weights from i to n as W_i^n and a vector X of all values i to X_i^n. The bias value b can be thought of as an additional input µ similar to x_n but with a weight edge W_b of 1 so that the impact is simply adding the value of this bias.
The input values X_i^n are the known independent variables we want to send through our deep learning model and are commonly things like the RGB values for an image at each pixel or a stock value at a point in time. To start, the weights W_i^n are commonly random values that are then adjusted as the network 'learns', with the final weights comprising the model after training.

Expanding on the importance of the dot product for neural nets, for a single node the inputs are vectors. Referencing https://stats.stackexchange.com/questions/291680/can-any-one-explain-why-dot-product-is-used-in-neural-network-and-what-is-the-in on the dot product:

"One way of seeing it is that the use of dot product in a neural network originally came from the idea of using dot product in linear regression. The most frequently used definition of a line is $y = ax + by = ax + b$. But this is the same as saying $b = y - axb = y - ax$, which is the same as saying $b = (y, x) \cdot (1, -a) b = (y, x) \cdot (1, -a)$. So mathematically, a line is expressed with a dot product between the coordinate axes y, xy, x and some other vector. And lines are useful for linear regression. And you can view neural networks as a linear model with a nonlinear activation tacked on top."

So we have: $z[x, w, b] = \sum_i^n (w_i x_i + b)$ as our fundamental equation.

Resource: https://becominghuman.ai/from-perceptron-to-deep-neural-nets-504b8ff616e

Student Response

Below is an example showing that the sum of elementwise multiplication of two vectors is the same as the dot product of those two vectors. Also we see the effect of varying the bias value in both tabular and graphic form. Here I arbitrarily set values for the tensors *"x"* and *"w"* along with a bias *"b"* and present them first as tensors, or rank 1 matrices, vertical vectors, and then confirm the mode of the dot product. Last I plot various values of b and show how the fundamental equation above is impacted.

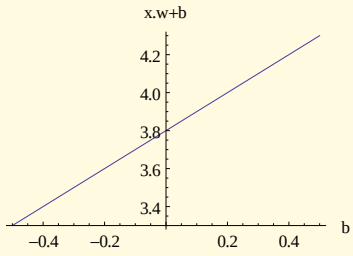
```
Clear[w, x, y, z, b, n]
x = {1, 2, 3};
w = {.5, .6, .7};
b = .5;
{MatrixForm[x] * MatrixForm[w]}
1*.5 + 2*.6 + 3*.7 + .5
z[x_, w_, b_] = w.x + b
Clear[b]
Table[x.w + b, {b, -.5, .5, .1}]
Plot[x.w + b, {b, -.5, .5}, AxesLabel -> {"b", "x.w+b"}]
```

$$\left\{\begin{pmatrix} 0.5 \\ 0.6 \\ 0.7 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}\right\}$$

```
4.3
```

```
4.3
```

```
{3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4., 4.1, 4.2, 4.3}
```

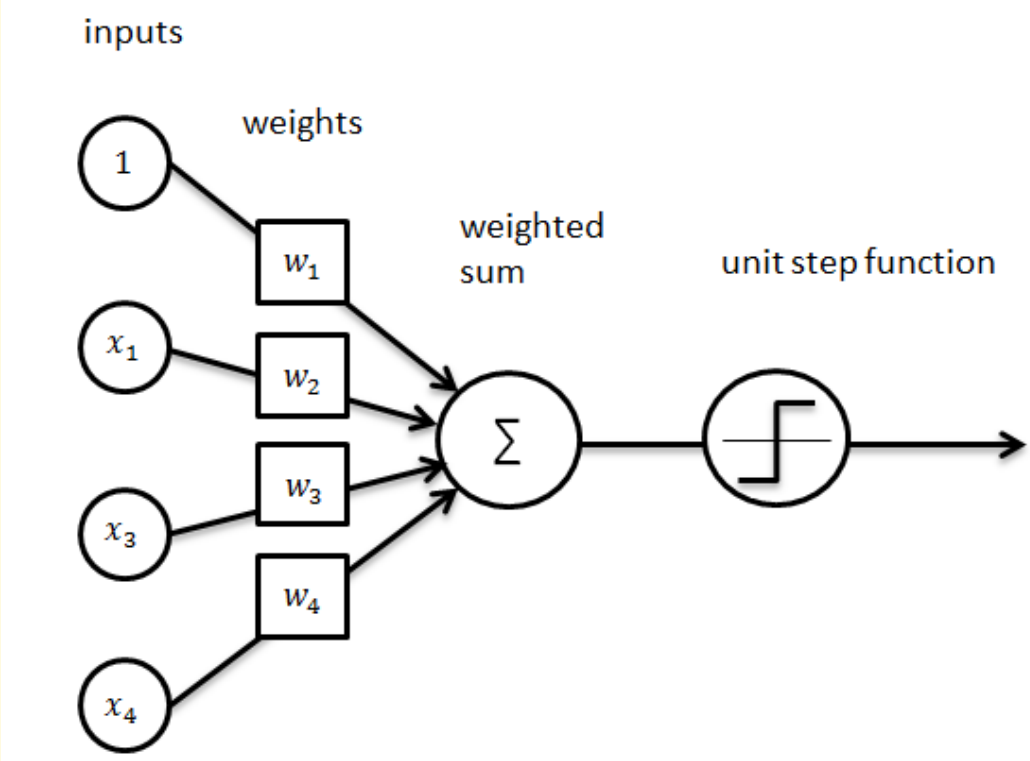


Student Response

aper: *"Function $z(x)$ is called the unit 's affine function and is followed by a rectified linear unit , which clips negative values to zero : $max(0, z(x))$. Such a computational unit is sometimes referred to as an "artificial neuron ."*

Comment:
An affine function is a linear function that does not necessarily traverse the origin. The key, as I have read, is that z(x) is linear and the activation function is not.
Not every activation function is a rectified linear unit. Some deep learning architectures use the hyperbolic tangent function tanh or the sigmoid . The key for deep learning is that the activation function be a non-linear function -- if only linear functions were used this would limit the applications as without a non-linear activation function a neural network would essentially be no different (or better) than a single perceptron.
From a paper on convolutional arithmetic we have another definition of the affine function:
"The bread and butter of neural networks is affine transformations : a vector is received as input and is multiplied with a matrix to produce an output (to which a bias vector is usually added before passing the result through a nonlinearity)."
From Udacity's Pytorch course we have more on activation functions:
"So far we've only been looking at the softmax activation , but in general any function can be used as an activation function . The only requirement is that for a network to approximate a non – linear function , the activation functions must be non – linear . Here are a few more examples of common activation functions : Tanh (hyperbolic tangent), and ReLU (rectified linear unit)."
.
Resources:
Affine: https:/ / math.stackexchange.com/ questions/ 275310/ what-is-the-difference-between-linear-and-affine-function
Affine: https:/ / arxiv.org/ pdf/ 1603.07285.pdf
Non-linear activation: https:/ / stackoverflow.com/ questions/ 9782071/ why-must-a-nonlinear-activation-function-be-used-in-a-backpropagation-neural-net
Udacity: https:/ / github.com/ udacity/ deep-learning-v2-pytorch/ blob/ master/ intro-to-pytorch/ Part%202%20-%20Neural%20Networks%20in%20PyTorch%20(Solution).ipynb
Paper: Neural networks consist of many of these units, organized into multiple collections of neurons called layers. The activation of one layer's units become the input to the next layer's units. The activation of the unit or units in the final layer is called the network output.
Comment: A diagram of a single perceptron, or node in the neural network, is below.

```
Import["http://ataspinar.com/wp-content/uploads/2016/11/perceptron_schematic_overview.png"]
```

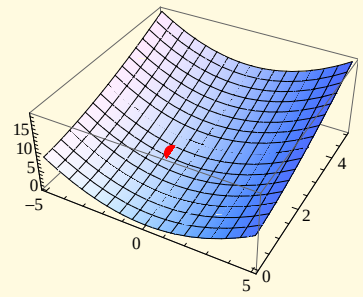


Student Response

Comment: Now I'm going to introduce a function and use a set values of X_i and random weights W_i to see the output of the loss function. I have a 3d surface and one point.

```
Clear[f, g, test1, point1, point2]
test1[f_, g_] = (f^2)/3 + g^(3/2);
firstplot = Plot3D[test1[f, g], {f, -5, 5}, {g, 0, 5}];
test1[-1, 2]
test1[1.5, 4];
point1 = Graphics3D[{PointSize[.05], Red, Point[{-1, 2, (1/3 + 2 Sqrt[2])}]}];
point2 = Graphics3D[{PointSize[.05], Green, Point[{1.5, 4, 8.75}]}];
Show[firstplot, point1]
```

$$\frac{1}{3} + 2\sqrt{2}$$



Student Response

We know what the neural net doesn't -- that the function that produced this surface is surface[f_, g_] = (f^2)/3 + g^(3/2) and therefore for test inputs {−1, 2} the corresponding point on the surface will be $\left\{-1, 2, \frac{1}{3}+2\sqrt{2}\right\}$ (in red above). For our single-node perceptron which has two inputs, f and g, we need two random weights. I'll generate as a pair. I set them randomly and am sticking with the same for reproducability. Weights W_1 & W_2 : {0.944979, 0.485514}

Returning to the single node and the surface above, the point we're working with is {-1, 2, 1/3 + 2 Sqrt[2]}, the weights are as above, the starting bias is .5, the initial output from this single-node neural net is .526 and that is 2.64 away from the actual.

```
point1 = {-1, 2, (1/3 + 2 Sqrt[2])}
z = N[1/3 + 2 Sqrt[2]]
x1 = -1;
x2 = 2;
weight1 = 0.9449794122289528;
weight2 =  0.48551359173806574;
b = .5;
h = {-1, 2}.{0.9449794122289528, 0.48551359173806574} + b;
{h, z - h}
```

$$\left\{-1, 2, \frac{1}{3}+2\sqrt{2}\right\}$$

```
3.16176
```

```
{0.526048, 2.63571}
```

Student Response

So above we have a single perceptron initiated with random weights and inputs that we know are on the surface. For this scenario we can pretend we are training a classifier that will return either 0 for inputs that produce output points on this surface or returns one for non-surface points. We also set a non-zero bias term. As we see, this is really far away from the true value of the function: 2.635 units away! This is just the framework, soon we will see how to train the network using the gradient to move these initially-random weights towards reliable values and/or change the bias value b.

From https://becominghuman.ai/from-perceptron-to-deep-neural-nets-504b8ff616e:

"For generating the output , Rosenblatt introduced a simple rule by introducing the concept of weights . Weights are basically real numbers expressing the importance of the respective inputs to the output . The neuron depicted ... will generate two possible values , 0 or 1, and it is determined by whether the weighted sum of each input , ∑ wjxj , is less than or greater than some threshold value . Therefore , the main idea of a perceptron algorithm is to learn the values of the weights w that are then multiplied with the input features in order to make a decision whether a neuron fires or not ."

Comment: So for this single node which takes as inputs a point on the surface above, random weights for the edges connecting the inputs to the node, and a bias value, the intermediate value is .5260.

P a p e r :

"Training this neuron means choosing weights w and bias b so that we get the desired output for all N inputs x. To do that, we minimize a loss function that compares the network's final activation (x) with the target (x) (desired output of x) for all input x vectors. To minimize the loss, we use some variation on gradient descent, such as plain stochastic gradient descent (SGD), SGD with momentum, or Adam. All of those require the partial derivative (the gradient) of activation (x) with respect to the model parameters w and b. Our goal is to gradually tweak w and b so that the overall loss function keeps getting smaller across all x inputs. If we're careful, we can derive the gradient by differentiating the scalar version of a common loss 3 function (mean squared error)"

Comment: We are given the loss function as $\text{loss}[x\_] = \frac{\sum_{x}(\text{target}[x]-\text{activation}[x]^2)}{N}$ and for the Relu activation this is equivalent to the first term minus the larger (max) of either 0 or $\sum_{x=i}^{|x|} w_i\, x_i$. Relu, or Rectified Linear Unit, is a type of step function where all values of x < 0 become 0 and values of x > 0 are x. I believe Mathematica has this integrated as Ramp, but I haven't investigated its application. As the output is greater than zero, the output of the single node with Relu activation is .5260. Of course the actual output of test1[f, g] is 3.16, so the loss is 2.6. Not useful at this point, pre-training!

Again from the same becominghuman.ai article, we are reminded that the goal is to find some combination of weights w_i and bias b_i that minimize the loss function. We won't implement this until the end of the paper.

Student Response

Paper 4 - p.6: *"Let's compute partial derivatives for two functions, both of which take two parameters. We can keep the same f(x, y) = 3x∘2 y from the last section, but let's also bring in g(x, y) = 2x + y∘2"*

Comment: Here we apply some vector calculus. As we saw in Math241, the gradient for any function f[x, y] is found through $\left\{\frac{\partial f}{\partial x}[x, y], \frac{\partial f}{\partial y}[x, y]\right\} = \nabla f[x, y]$ and the paper is now going to apply this at scale. Below we see the equations f[x, y] and g[x, y] and their gradients in a matrix form.

```
f[x_, y_] = 3 x^2 y
g[x_, y_] = 2 x + y^8
gradf[x_, y_] = {D[f[x, y], x], D[f[x, y], y]};
gradg[x_, y_] = {D[g[x, y], x], D[g[x, y], y]};
MatrixForm[{gradf[x, y], gradg[x, y]}]
```

$$\begin{pmatrix} 6\,x\,y & 3\,x^2 \\ 2 & 8\,y^7 \end{pmatrix}$$

Student Response

Comment:

And the gradient for our test function $\text{test1}[f\_, g\_] = \frac{f^2}{3} + g^{3/2}$ is $\left\{\frac{2f}{3}, \frac{3\sqrt{g}}{2}\right\}$ which will output scalars as we see highlighted later in the paper. The gradient at point {-1, 2} is below: {-2/ 3, 3/ Sqrt[2]}.

```
Clear[f, g];
{f, g} = {-1, 2};
{(2 * f) / 3, (3 * Sqrt[g]) / 2}
N[{(2 * f) / 3, (3 * Sqrt[g]) / 2}]
```

$$\left\{-\frac{2}{3}, \frac{3}{\sqrt{2}}\right\}$$

```
{-0.666667, 2.12132}
```

Student Response

Below I plotted the level curves of the test function and the corresponding gradient. I would like to return to this because I don't intuitively understand the contour plotting at work.

"A level curve of a function f(x,y)f(x,y) is the curve of points (x,y)(x,y) where f(x,y)f(x,y) is some constant value."

The level curves for the test function are of the type we saw in Calc II, parabolic in shape, whereas the level curves for the gradient are all rectangular. The gradient is {2f/3, 3 Sqrt[g]/2} and I "think" the reason for the rectangles is that the output jumps in and out of real numbers (ex of complex below).

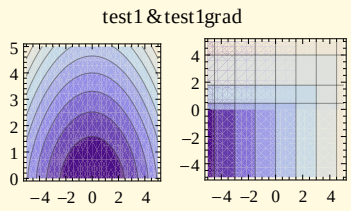
Resource: https://mathinsight.org/level_sets

```
test1grad[0, -2]
```

$$\left\{0, \frac{3\,i}{\sqrt{2}}\right\}$$

```
{test1[f_, g_] = (f^2)/3 + g^(3/2),  test1grad[f_, g_] = {D[test1[f, g], f], D[test1[f, g], g]}}
p1 = ContourPlot[test1[f, g], {f, -5, 5}, {g, -5, 5}, PlotRange -> All];
p2 = ContourPlot[test1grad[f, g], {f, -5, 5}, {g, -5, 5}, PlotRange -> All];
GraphicsRow[{p1, p2}, PlotLabel -> "test1 & test1grad"]
```

$$\left\{\frac{f^2}{3} + g^{3/2}, \left\{\frac{2\,f}{3}, \frac{3\sqrt{g}}{2}\right\}\right\}$$



Student Response

Paper Section 4.1: *"With multiple scalar−valued functions, we can combine them all into a vector just like we did with the parameters. Let y = f(x) be a vector of m scalar−valued functions that each take a vector x of length n = |x| where |x| is the cardinality (count) of elements in x. Each f,i function within f returns a scalar just as in the previous section:"*

$$\begin{pmatrix} y1 = f1\,x \\ y2 = f2\,x \\ ym = fm[x] \end{pmatrix}$$

Style["For instance, we'd represent f(x, y) = 3x^2 y and g(x, y) = 2x + y^8 from the last section as y1= f1[x_] = Subsuperscript[3 x,1, 2] (substituting x1 for x, x2 for y) y2 = f2[x_] = Subscript[2x, 1] + Subsuperscript[x, 2, 8] which reminds me of working with differential equations and linearizing the oscillator equation. It's not the same, but the substitution has a similar feel for me.", Italic, FontColor -> Blue]

Comment: From http://web.mit.edu/wwmath/vectorc/scalar/intro.html defines 'scalar valued function' as *"a function that takes one or more values but returns a single value (one that has only magnitude). f(x,y,z) = x2+2yz5 is an example of a scalar valued function. A n−variable scalar valued function acts as a map from the space Rn to the real number line. That is, f:Rn−>R."*

For our example perhaps we would stack the gradient as individual scalars in an 2 x 1 vector? If so, it would look like:

$$\begin{pmatrix} \frac{2f}{3} \\ \frac{3\sqrt{g}}{2} \end{pmatrix}$$ which if we replaced g with f_2 would be:

$\begin{pmatrix} \frac{2 f_1}{3} \\ \frac{3\sqrt{f_2}}{2} \end{pmatrix}$ so now we have the functions in a vector with remapped variables.

Resource: http://web.mit.edu/wwmath/vectorc/scalar/intro.html

Student Response

Paper Section 4.1 p.7:

"It's very often the case that m = n because we will have a scalar function result for each element of the x vector. For example, consider the identity function y = f(x) = x: y1 = f1(x) = x1, y2 = f2(x) = x2, yn = fn(x) = xn So we have m = n functions and parameters, in this case."

Comment: Intuitively I don't see why the dimensions would be equal. M is a vector of scalar-valued functions that each take a vector x of length n = |x| where |x| is the cardinality (count) of elements in x, and in my opinion there is no reason why the number of elements in each function comprising m would be the same as the number of functions in m. If I read this correctly, Jeremy is saying that the number of elements (individual variables) in each function is often equal to the number of functions underpinning the solution. I see no reason why this would be true. [question sent to Bruce 1/20]

Paper: *"For example, consider the identity function y = f(x) = x:"*

Comment: The identity function seems to me to be one of the only cases where the dimensions would be equal. Perhaps the identity function is frequently used in deep learning? This remains the largest question I have on the paper, but the full implications remain uncertain to me.

Student Response

Paper p. 7 (bottom of page):

"Generally speaking, though, the Jacobian matrix is the collection of all m × n possible partial derivatives (m rows and n columns), which is the stack of m gradients with respect to x:"

Comment: The diagram doesn't copy, but the item of note is that continuing to present each variable in each equation as x_i rather than $\{f_i, g_i, h_i\}$ drives home the point that the gradient is iterating over a vector of variables x_i each vector associated with an individual function f_i with each function grouped into a vector of functions of length **m** to give us the m x n matrix.

Paper: *"Each ∂ ∂x fi(x) , the partial derivative with respect to each x , is a horizontal n−vector because the partial derivative is with respect to a vector, x, whose length is n = |x|. The width of the Jacobian is n if we're taking the partial derivative with respect to x because there are n parameters we can wiggle, each potentially changing the function's value. Therefore, the Jacobian is always m rows for m equations. It helps to think about the possible Jacobian shapes visually:"*

"The Jacobian of the identity function f(x) = x, with fi(x) = xi , has n functions and each function has n parameters held in a single vector x. The Jacobian is, therefore, a square matrix since m = n:"

Comment: Again I don't see the link between the number of elements n in each function and the number of functions m involved, but i agree that the Jacobian will always be m rows for m equations and n columns for n variables, just not that the Jacobian will always be square. The Jacobian of the identity function will be square, but not all Jacobians will be.

On page 29 the paper mentions

"The Jacobian of a vector−valued function that is a function of a vector is an m × n (m = |f| and n = |x|) matrix containing all possible scalar partial derivatives"

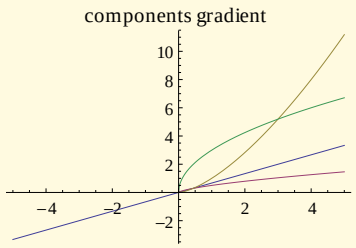
and pulling from http://tutorial.math.lamar.edu/Classes/CalcIII/VectorFunctions.aspx a vector function is "A vector function is a function that takes one or more variables and returns a vector."

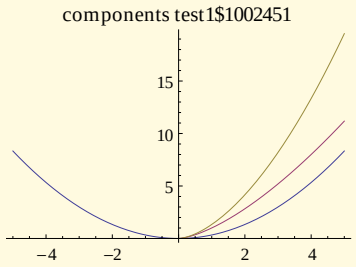
At any rate, this is a key point in the paper. We have already remapped all our equations above to be f_m and imagined them in a vector **{x}** which we present as having dimensions [n x 1] , so it will be as tall (long) as there are variables in the equation. For $\text{test1}[f\_, g\_] = \frac{f^2}{3} + g^{3/2}$ this would give a vector of $\left\{\frac{f_1^2}{3}, f_2^{3/2}\right\}$ which we could represent as $\begin{pmatrix} \frac{f_1^2}{3} \\ f_2^{3/2} \end{pmatrix}$ with has dimensions m x n of [2, 1]. In this example the output is, as the paper suggests, a vector of scalar functions each outputting **x** .

From there we start looking at the partial derivatives. If $\text{test1}[f_1\_, f_2\_] = \left\{\frac{f_1^2}{3} + f_2^{3/2}\right\}$ then our matrix of partial deriva-

tives will be:
$\begin{pmatrix} \frac{2}{3} f_1 f_2^{3/2} \\ \frac{1}{2} f_1^{2/3} 3\sqrt{f_2} \end{pmatrix}$ and each term does output scalars. And I note that this is a square matrix where m = n = 2. At a later point I would like to use additional test functions to investigate the squareness of the Jacobian...
In the remainder of the paper it looks like Jeremy assumes square matrices where the Jacobian is always m rows for m equations and n columns for n variables where m = n. I'll continue that. assumption as, for my test equation above it was true, and it does simplify the approach.

```
f11 = 2 f / 3;
f21 = f^(3 / 2);
f12 = f^(2 / 3) / 2;
f22 = 3 Sqrt[f];
Plot[{f11, f12, f21, f22}, {f, -5, 5}, PlotLabel -> gradient components]
Plot[{f^2 / 3, f^(3 / 2), f^2 / 3 + f^(3 / 2)}, {f, -5, 5}, PlotLabel -> test1 components]
```

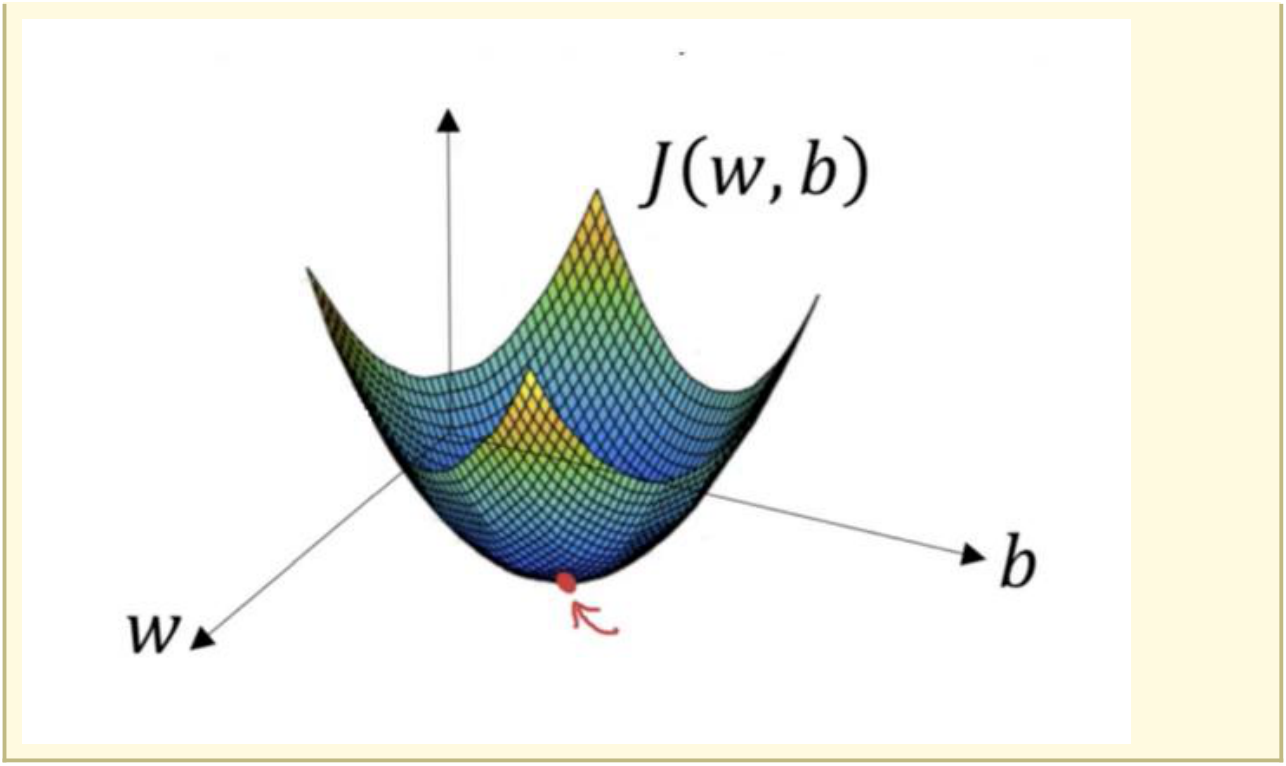




Student Response

Above I am plotting the curves to get a sense for how the individual components feature. Over time I would like to develop an intuitive sense of how these relate, but there's not a lot that jumps out at me now. Below is an import of the loss function for y = x^2. I'll come back to this later. This is a visual I pulled showing the goal of the loss function -- to find the minimum. Familiar from Calc, but a nice reminder.

```
Import["http://p1.pstatp.com/large/6c3c00008c4a37cffca6"]
```



Student Response

Paper: *"Also make sure you pay attention to whether something is a scalar–valued function, y , or a vector of functions (or a vector–valued function), y."*

Comment: When I first read these terms it was perhaps my first exposure to the term 'scalar-valued', but it's simple enough: a scalar-valued function is a function that outputs a value, a vector of functions is just that exactly, and a vector-valued function is a function that outputs a vector (e.g. y=x). The only example that comes to mind is a function that outputs a line. I understand that we need scalar-valued functions because our goal is to have matrices of weights that are real numbers and anything except scalar-valued functions won't generate real numbers.

Paper: 4.2 Derivatives of vector element-wise binary operators

p. 9 - *"Element–wise binary operations on vectors, such as vector addition w + x, are important because we can express many common vector operations, such as the multiplication of a vector by a scalar, as element–wise binary operations. By "element–wise binary operations" we simply mean applying an operator to the first item of each vector to get the first item of the output, then to the second items of the inputs for the second item of the output, and so forth. This is how all the basic math operators are applied by default in numpy or tensorflow , for example. Examples that often crop up in deep learning are max(w , x) and w > x (returns a vector of ones and zeros)."*

Comment: Ok. Vector element-wise operations are a prominent feature of R and other data-science array-manipulating software, makes sense this is part of deep learning given the speed with which element-wise operations can be performed along arrays or vectors.

Paper: *"We can generalize the element–wise binary operations with notation y = f(w) g(x) where m = n = |y| = |w| = |x| We write n (not m) equations vertically to emphasize the fact that the result of element–wise operators give m = n sized vector results"*

Comment: Again not fully following the logic of this but agree to assume squareness. If we agree that the number of functions y/ m is equal to the number of components of each function n then we know that y/ m = n = x = w because we have already rewritten the functions f(x) g(y) h(z) as f_i x_i and there will be a corresponding weight w_i for each term.

Paper p. 10: *"Fortunately the Jacobian is very often a diagonal matrix, a matrix that is zero everywhere but the diagonal. Because this greatly simplifies the Jacobian, let's examine in detail when the Jacobian reduces to a diagonal matrix for element–wise operations."*
"Those partials go to zero when fi and gi are not functions of w ,ji – fi(wi) and gi(xi) look like constants to the partial differentiation operator with respect to wj when j is not i so the partials are zero off the diagonal."
"We'll take advantage of this simplification later and refer to the constraint that fi(w) and gi(x) access at most wi and xi , respectively, as the element–wise diagonal condition."

Comment: It seems that we are essentially arbitrarily assuming the Jacobian will be a diagonal matrix, but it's interesting to see that the general derivative of each term through each function with respect to either the inputs x_i or weights w_i will be a diagonal matrix with every term except the diagonal being 0 and the diagonal being 1. As I understand the point of these paragraphs, the essential is that where the derivative isn't on the diagonal of the matrix the other terms are treated as constants and go to zero, with multiplication inherent to the term driving those terms to zero. Now that we have re-rendered the functions in common terms this is a really interesting point that I look at in the input cell below.

Paper: *"More succinctly, we can write:"* ∂y ∂w = diag ∂ ∂w1 (f1(w1) g1(x1)), ∂ ∂w2 (f2(w2) g2(x2)), . . . , ∂ ∂wn (fn(wn) gn(xn))

Comment: Below isn't quite the right syntax, but we do see how the function f (w) g plays out from $f_1\, w_1\, g_1\, x_1$ to $f_n\, w_n\, g_n\, x_1$.
Paper: ∇y = h ∂x1 ∂x1 , ∂x2 ∂x2 , . . . , ∂xn ∂xn i = 1, 1, . . . , 1 = ~1 T Notice that the result is a horizontal vector full of 1s, not a vertical vector, and so the gradient is ~1 T
Comment: Again the emphasis on the horizontal nature of the vector and referring to this as the transpose is not something I would naturall emphasize, but it's clearly important when we start taking dot products of these vectors.

Student Response

First, the sample equation pair f[x,y] & g[x,y] given on p.6. Below we have:
1 - Vector of the original equations
2 - Vector of the equations rewritten as f_i .
3 - Matrix of the gradient of the original equation
4 - Matrix of the rewritten equations

```
f[x_, y_] = 3 x^2 y;
g[x_, y_] = 2 x + y^8;
gradf[x_, y_] = {D[f[x, y], x], D[f[x, y], y]};
gradg[x_, y_] = {D[g[x, y], x], D[g[x, y], y]};
MatrixForm[{3 x^2 y, 2 x + y^8}]
MatrixForm[{3 * (Subscript[f, 1])^2 Subscript[f, 2], (2 * (Subscript[f, 1])) + Subscript[f, 2]^8}]
MatrixForm[{gradf[x, y], gradg[x, y]}]
MatrixForm[{{(6 * Subscript[f, 1]) Subscript[f, 2], 3 * (Subscript[f, 1]^2)}, {2, 8 * Subscript[f, 2]^7}}]
```

$$\begin{pmatrix} 3\,x^2\,y \\ 2\,x + y^8 \end{pmatrix}$$

$$\begin{pmatrix} 3\,f_1^2\,f_2 \\ 2\,f_1 + f_2^8 \end{pmatrix}$$

$$\begin{pmatrix} 6\,x\,y & 3\,x^2 \\ 2 & 8\,y^7 \end{pmatrix}$$

$$\begin{pmatrix} 6\,f_1\,f_2 & 3\,f_1^2 \\ 2 & 8\,f_2^7 \end{pmatrix}$$

Student Response

So if I read this correctly, for the example above when we add in variables w_i (on this page the paper seems to be using w for the variable in the functions f_1 and not to refer to weights) to our 2x2 gradient we will get a diagonal matrix for

the partial derivative of our function vector y with respect to w: $\frac{dy}{dw} = \begin{pmatrix} \frac{dd}{dw} 6(f_1 w_1) f_2 & 0 \\ 0 & 8 f_2^{\,7} w_2 \end{pmatrix}$. And if we look at the partial derivative for our function vector y with respect to x (the variable in the functions g_i) we get: $\frac{dy}{dx} = \begin{pmatrix} 6 f_1 f_2 x_1 & 0 \\ 0 & 8 f_2^{\,7} x_2 \end{pmatrix}$.

Student Response

Paper: *"We'll take advantage of this simplification later and refer to the constraint that fi(w) and gi(x) access at most wi and xi , respectively, as the element−wise diagonal condition."*

Comment: This seems like another key point in the paper. When taking the gradient of a series of functions every component except N_i will be 0.

Paper 4.3 p.12: *"Using the usual rules for scalar partial derivatives, we arrive at the following diagonal elements of the Jacobian for vector−scalar addition: ∂ /∂ xi_(fi(xi) + gi(z)) = ∂ (xi + z) ∂ xi_=_∂ xi/∂ xi_+_∂ z/∂ xi_=_(1 + 0)_=_1 So, ∂ /∂ x_(x + z) = diag(~1) = I."*

Comment: I'm not 100% sure about the Identity matrix in this example. Obviously the matrix of partial derivatives will be a diagonal matrix of 1s, so I am assuming that the paper means that when multiplied in a chain rule of partial derivatives this will leave the other party unchanged.

Computing the partial derivative with respect to the scalar parameter z, however, results in a vertical vector, not a diagonal matrix. The elements of the vector are: ∂ ∂z (fi(xi) + gi(z)) = ∂(xi + z) ∂z = ∂xi ∂z + ∂z ∂z = 0 + 1 = 1 Therefore, ∂ ∂ z (x + z) = ~1.

Comment: In words, the partial derivative of function f[x + z] $F_i\, x_i + z$ with respect to x_i is equal to the partial derivative of x_i with respect to x_i + the partial derivative of z with respect to x_i which is $1 + 0 = 1$. And for $d x_i$ we get a diagonal. The partial derivative for z is vertical, I think, because of what the paper mentions slightly above, that adding a scalar z to a vector is really extending the scalar into a vector first, which would make that a vertical vector as only the scalar can be differentiated.

Again the orientation of the derivative vector is highlighted. The diagonality of the derivative w/ r/ t xi makes sense as only the ith terms will be non-zero, but for me the verticality of the derivative of the scalar constant is non-intuitive and I would like to research the impact of this further.

Student Response

Paper: 4.4 Vector sum reduction

"Summing up the elements of a vector is an important operation in deep learning, such as the network loss function, but we can also use it as a way to simplify computing the derivative of vector dot product and other operations that reduce vectors to scalars. ... Notice that the result is a horizontal vector full of 1s, not a vertical vector, and so the gradient is_~1_T."

Comment: The gradient, described as a 1xN Jacobian, simplifies to a horizontal (not vertical) vector of 1s. This seems useful for future dot product work, but not clear at this time how it applies.

https:/ / medium.com/ data-science-group-iitr/ loss-functions-and-optimization-algorithms-demystified-bb92daff331c

Future goal: Investigate loss functions.

Student Response

4.5 The Chain Rules p.14:

P a p e r :

"We can't compute partial derivatives of very complicated functions using just the basic matrix calculus rules we've seen so far. For example, we can't take the derivative of nested expressions like sum(w + x) directly without reducing it to its scalar equivalent. We need to be able to combine our basic vector rules using what we can call the vector chain rule. Unfortunately, there are a number of rules for differentiation that fall under the name "chain rule" so we have to be careful which chain rule we're talking about. Part of our goal here is to clearly define and name three different chain rules and indicate in which situation they are appropriate. To get warmed up, we'll start with what we'll call the single−variable chain rule, where we want the derivative of a scalar function with respect to a scalar. Then we'll move on to an important concept called the total derivative and use it to define what we'll pedantically call the single−variable total−derivative chain rule. Then, we'll be ready for the vector chain rule in its full glory as needed for neural networks. The chain rule is conceptually a divide and conquer strategy (like Quicksort) that breaks complicated expressions into subexpressions whose derivatives are easier to compute. Its power derives from the fact that we can process each simple subexpression in isolation yet still combine the intermediate results to get the correct overall result."

Comment: Quicksort is a classic computer science algorithm that can sort a list of values in O(n^2) tasks. It works by breaking the list to be sorted into sub-lists, each of which is sorted, so I see the parallel.

Resources:

https://www.geeksforgeeks.org/quick-sort/

https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

Paper:

"The chain rule comes into play when we need the derivative of an expression composed of nested subexpressions. For example, we need the chain rule when confronted with expressions like d/dx(sin(x2)). The outermost expression takes the sin of an intermediate result, a nested subexpression that squares x. Specifically, we need the single−variable chain rule, so let's start by digging into that in more detail."

"It's better to define the single−variable chain rule of f(g(x)) explicitly so we never take the derivative with respect to the wrong variable. Here is the formulation of the single−variable chain rule we recommend: dy/dx = dy/du ∗ du/dx"

Comment: Jeremy launches into an explanation of the chain rule focused on u-substitution, rehashing intermediate calculus. No comment.

Paper p.16: *"The chain rule is, by convention, usually written from the output variable down to the parameter(s), dy/dx = dy/du du/dx . But, the x−to−y perspective would be more clear if we reversed the flow and used the equivalent dy/dx = du/dx dy/du."*

Comment: Not sure I agree with this, but I'll try this method of visualizing the flow, which I don't find intuitive.

Paper: *"Many readers can solve d/dx(sin(x◦2)) in their heads, but our goal is a process that will work even for very complicated expressions. This process is also how automatic differentiation works in libraries like PyTorch. So, by solving derivatives manually in this way, you're also learning how to define functions for custom neural networks in PyTorch."*

Comment: Not to brag, I can take the derivative of sin(x^2) in my head. :-)

Resources:

https://towardsdatascience.com/getting-started-with-pytorch-part-1-understanding-how-automatic-differentiation-works-5008282073ec

https://openreview.net/pdf?id=BJJsrmfCZ

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Paper 4.5.2 p.18: *"Single−variable total−derivative chain rule Our single−variable chain rule has limited applicability because all intermediate variables must be functions of single variables. But, it demonstrates the core mechanism of the chain rule, that of multiplying out all derivatives of intermediate subexpressions. To handle more general expressions such as y = f(x) = x + x2 , however, we need to augment that basic chain rule. Of course, we immediately see dy dx = d dx x + d dx x 2 = 1 + 2x, but that is using the scalar addition derivative rule, not the chain rule"*

Comment: A good reminder on the scalar addition derivative rule and how it differs from chain rule.

Paper p.19:

"In practice, just keep in mind that when you take the total derivative with respect to x, other variables might also be functions of x so add in their contributions as well. The left side of the equation looks like a typical partial derivative but the right−hand side is actually the total derivative. It's common, however, that many temporary variables are functions of a single parameter, which means that the single−variable total−derivative chain rule degenerates to the single−variable chain rule."

Comment: Shows how the total derivative can be thought of as the sum of chain ruled derivatives for each x_i . Again, a review of fundamental calculus at some length.

Paper: *"We can achieve that by simply introducing a new temporary variable as an alias for x:"* $u_n + 1 = x$.

"Then, the formula reduces to our final form. A bit of dramatic foreshadowing, notice that the summation" $\sum_i^{n+1} \frac{df}{du}\frac{du}{dx}$

sure looks like a vector dot product, $\partial f/\partial u \cdot \partial u/\partial x$, or a vector multiply $\partial f/\partial u\, \partial u/\partial x$.

Comment: Foreshadowing indeed! Quite nifty, looking forward to seeing where this goes.

Paper p.22: ... *"That means that the Jacobian is the multiplication of two other Jacobians, which is kinda cool. Our complete vector chain rule is:"* $\partial\, \partial x\, f(g(x)) = \partial f/\partial g\, \partial g/\partial x$

Comment: It is pretty cool.More examination to follow. The big lesson from this portion of the paper is that rather than deriving scalars the same approach works with vectors, and those vectors can be multiplied as matrices.

Student Response

Paper p.22: Even within this $\partial f/\partial g\, \partial g/\partial x$ formula, we can simplify further because, for many applications, the Jacobians are square (m = n) and the off-diagonal entries are zero.

Comment: Again the comment that 'many' applications result in square Jacobians. Would like to nail down what applications these are.

Paper: As we saw in a previous section, element-wise operations on vectors w and x yield diagonal matrices with elements $\partial wi/\partial xi$ because wi is a function purely of xi but not xj for j / = i. The same thing happens here when fi is purely a function of gi and gi is purely a function of xi.

Comment: Agreed

Paper: 5 The gradient of neuron activation p. 23

"We now have all of the pieces needed to compute the derivative of a typical neuron activation for a single neural network computation unit with respect to the model parameters, w and b: activation(x) = max(0, w · x + b) (This represents a neuron with fully connected weights and rectified linear unit activation. There are, however, other affine functions such as convolution and other activation functions, such as exponential linear units, that follow similar logic.)"

Comment: The payoff! Here we are returning to our concept of the perceptron with a single node, a vector of weights w, a weight b, and an activation function.

Paper:

Let's worry about max later and focus on computing $\partial/\partial w\ (w \cdot x + b)$ and $\partial/\partial b\ (w \cdot x + b)$. (Recall that neural networks learn through optimization of their weights and biases.) We haven't discussed the derivative of the dot product yet, y = f(w)· g(x), but we can use the chain rule to avoid having to memorize yet another rule. (Note notation y not y as the result is a scalar not a vector.)

Comment: I think this should be straightforward in theory but in practice I feel like there is complexity. Interested in where this goes.

Paper: The dot product w · x is just the summation of the element-wise multiplication of the elements: $\sum_i^n w_i\, x_i$ = sum(w ⊗ x). (You might also find it useful to remember the linear algebra notation w · x = wT x.)

Comment: I don't recall using the Transpose of w in the definition of a dot product in linear algebra.

Paper p.24:

"We know how to compute the partial derivatives of sum(x) and w ⊗ x but haven't looked at partial derivatives for sum(w ⊗ x). We need the chain rule for that and so we can introduce an intermediate vector variable u just as we did using the single−variable chain rule: u = w ⊗ x and y = sum(u)"

"Once we've rephrased y, we recognize two subexpressions for which we already know the partial derivatives: ∂u/∂w = ∂/∂w (w ⊗ x) = diag(x) & ∂y/∂u = ∂/∂u sum(u) = ~1 T "

The vector chain rule says to multiply the partials: ∂y/∂w = ∂y/∂u ∂u/∂w = ~1 T diag(x) = x T

Comment: We saw early on that the partial derivative with respect to w of (w ⊗ x) is a diagonal matrix of x. This was useful to see that the product of these two partial derivatives will be not a diagonal matrix but a vector, in this case a horizontal vector of x.

Paper: *"Let's tackle the partials of the neuron activation, max(0, w · x + b). The use of the max(0, z) function call on scalar z just says to treat all negative z values as 0. The derivative of the max function is a piecewise function. When z ≤ 0, the derivative is 0 because z is a constant. When z > 0, the derivative of the max function is just the derivative of z, which is 1."*

Comment: Initially I didn't find the equivalence of wx + b to z to be straightforward, but obviously the output of w.x + b will be a scalar z, and the derivative with respect to z will be 1.

Paper p.25: *"When the activation function clips affine function output z to 0, the derivative is zero with respect to any weight wi . When z > 0, it's as if the max function disappears and we get just the derivative of z with respect to the weights."*

Comment: That derivative is a horizontal vector of X.

Paper: Style["Turning now to the derivative of the neuron activation with respect to b, we get:
∂activation/∂b = (0 ∂z/∂b = 0 w · x + b ≤ 0 &1 ∂z/∂b = 1 w · x + b > 0
Let's use these partial derivatives now to handle the entire loss function.", Italic, FontColor -> Blue]

Student Response

Paper 6 The gradient of the neural network loss function p. 25

"Training a neuron requires that we take the derivative of our loss or "cost" function . with respect to the parameters of our model, w and b. Then the cost equation becomes:"

$$\frac{\sum_{i}^{n}(y-\text{activation}[x_i]^2)}{N} = \frac{\text{Sum}\left[y_i-\text{Max}\left[0,w.x_i+b\right]\right]^2}{N}$$

"Following our chain rule process introduces these intermediate variables:"

u(w, b, x) = max(0, w · x + b)
v(y, u) = y - u
C(v) = 1/ N Subm[v^2, {i, N}]

"Let's compute the gradient with respect to w first."

Paper 6.1 The gradient with respect to the weights p.26:

Overall gradient:

$$\frac{2\sum_{i}^{N}(w\cdot x_i+b-y_i)\,x_i^T}{N} \text{ when } w\cdot x_i + b > 0$$

To interpret that equation, we can substitute an error term $e_i = w\cdot x_i + b - y_i$:

Comment: Until this point this section was walking through a straightforward u-substitution. The use of the phrase 'error term' here is initially odd, but I think that this is equivalent to $\hat{y} - y_i$ to show the difference between the predicted value and the actual value.

Paper: ...yielding dC/ dw = $\frac{2\sum_{i}^{N} e_i\, x_i^T}{N}$

From there, notice that this computation is a weighted average across all xi in X. The weights are the error terms, the difference between the target output and the actual neuron output for each xi input.

Comment: It's 2 over the number of x_i terms and we sum the error (w.x + b - y) times the x for each term i to N. I think this is just multiplication of a scalar with a scalar (e.g. Hadamard) as we sum this.

Student Response

Paper: 6.2 The derivative with respect to the bias

$\partial C/\partial b = \frac{2\sum_{i}^{N} e_i}{N}$ (for the nonzero activation case)

where e is the error term above. The partial derivative is then just the average error or zero, according to the activation level. To update the neuron bias, we nudge it in the opposite direction of increased cost: bt+1 = bt - $\eta\, \partial C / \partial b$

Paper p. 29: In practice, it is convenient to combine $\mathbf{w}$ and b into a single vector parameter rather than having to deal with two different partials: $\hat{w} = [\mathbf{w}^T, b]T$. This requires a tweak to the input vector $\mathbf{x}$ as well but simplifies the activation function. By tacking a 1 onto the end of x, $\hat{x} = [x\,T, 1]$, $w \cdot x + b$ becomes $\hat{w} \cdot x$.
This finishes off the optimization of the neural network loss function because we have the two partials necessary to perform a gradient descent.

Comment: This seems like a hugely important tweak but the paper ends without an explanation of how to apply this to the gradient descent optimization problem. And, as Bruce C. advised, much of the notation above is superluous at best, as all we're doing is taking the 'vertical vector' and transposing then tagging b to it and then re-transposing to turn it back into a vertical vector.

At any rate, now to apply this! Going all the way back to my simple sample equation test1 $[f\_, g\_] = (f\wedge 2)/3 + g\wedge(3/2)$ and a test point $\left\{-1, 2, \frac{1}{3} + 2\sqrt{2}\right\}$, I will use $\{-1, 2\}$ as my vector x and my random weights weight1 = 0.9449 & weight2 = 0.48551 with initial b = .5. First I will run some iterations updating the weights alone to see how the model approximates the output.

```
point1 = {-1, 2, (1/3 + 2 Sqrt[2])};
weight1v1 = 0.9449794122289528;
weight2v1 = 0.48551359173806574;
w1 = {weight1v1, weight2v1};
x1 = -1;
x2 = 2;
x = {x1, x2};
yi = N[1/3 + 2 Sqrt[2]];
b1 = .5;
yi;
h1 = x.w1 + b1;
{yi, h1, yi - h1}
```

```
{3.16176, 0.526048, 2.63571}
```

Student Response

One approach, following the Pytorch gradient descent approach in Udacity
activation function : 1/ 1 + E^ (−x)) -- use sigmoid here instead of ReLu.
$\hat{y} = act\,(w.x + b)$
error = $(w.x + b) - y_i$
update functions:
$w_i = w_i + act\,(y - \hat{y})\, x_i$
$b = b + act\,(y - \hat{y})$

Resource: https:/ / github.com/ udacity/ deep-learning-v2-pytorch/ blob/ 9f2e2700030c59edb91d34d8e9fa710824aacf75/ intro-neural-networks/ gradient-descent/ GradientDescent.ipynb

Using the random weights set above and the input values {x, y} for the test function and known output z as x_1, x_2 and y_1 along with arbitrary bias term b the first manual round of weight setting looks like the below.
$\hat{y}$ the predicted value of y_1 is .6

```
Clear[x, yhat, w1, x, b];
weight1v1 = 0.9449794122289528;
weight2v1 = 0.48551359173806574;
w1 = {weight1v1, weight2v1};
```

```
x1 = -1;
x2 = 2;
x = {x1, x2};
y1 = N[1 / 3 + 2 Sqrt[2]];
b1 = .5;
yhatstep1 = (w1.x + b1);
yhat = 1 / (1 + E^(-yhatstep1))
error = (w1.x + b1) - y1
int =  (y1 - yhat)
weight1v2 = 2 weight1v1 + x1 (1 / (1 + E^(-int)))
weight2v2 = 2 weight2v1 + x2 (1 / (1 + E^(-int)))
b2 = b1 (1 / (1 + E^(-int)))
```

```
0.628561
```

```
-2.63571
```

```
2.5332
```

```
0.963522
```

```
2.8239
```

```
0.463218
```

```
w2 = {weight1v2, weight2v2};
yhat2step1 = (w2.x + b2)
yhat2 = 1 / (1 + E^(-yhat2step1))
error = (w2.x + b2) - y1
int =  (y1 - yhat2)
```

```
5.1475
```

```
0.99422
```

```
1.98574
```

```
2.16754
```

Student Response

And after one round of training we see that the error is decreasing as is the distance between the known value y_1 and the prediction $\hat{y}$. Let's try another round.

```
weight1v3 = 2 weight1v2 + x1 (1 / (1 + E^(-int)));
weight2v3 = 2 weight2v2 + x2 (1 / (1 + E^(-int)));
b3 = b2 (1 / (1 + E^(-int)))
w3 = {weight1v3, weight2v3}
error = (w3.x + b3) - y1
yhat3step1 = (w3.x + b3)
yhat3 = 1 / (1 + E^(-yhat3step1))
int =  (y1 - yhat3)
```

```
0.415644
```

```
{1.02975, 7.44239}
```

```
11.1089
```

```
14.2707
```

```
0.999999
```

```
2.16176
```

```
weight1v4 = 2 weight1v3 + x1 (1 / (1 + E^(-int)));
weight2v4 = 2 weight2v3 + x2 (1 / (1 + E^(-int)));
b4 = b3 (1 / (1 + E^(-int)))
w4 = {weight1v4, weight2v4}
error = (w4.x + b4) - y1
yhat4step1 = (w4.x + b4)
yhat4 = 1 / (1 + E^(-yhat4step1))
int =  (y1 - yhat4)
```

```
0.372734
```

```
{1.16273, 16.6783}
```

```
29.4049
```

```
32.5666
```

```
1.
```

```
2.16176
```

Student Response

Hmm. After three rounds our loss decrease is slowing. Let's try to script this and do more rounds. Update: Hilbert isn't quite the same as Mathematica and I've been unable to get a successful For[] loop export output as Print[] is disabled, etc. Let's try the loss function in the paper next.

dC/ dw = $\frac{2\sum_{i}^{N} e_i \, x_i^T}{N}$ so to do this manually I need to multiply the error term (w.x + b - z) for each xi times a horizontal matrix of xi. I think the goal is that the output is a scalar, which is then multiplied by 2 over the number of x terms n to get an average.
As an example, then, the first xi is the x-component of our point, -1, and the first w is the weight associated with it. So we take the dot product of x1 and w1, add our bias b, subtract our target y value, and multiply by the x1 term. We then repeat with x2, w2, the same b and y, again multiply, add the two, multiply by 2, divide by N =2, and that's our scalar derivative with respect to w.

```
z =  3.16176;
x1 = -1;
x2 = 2;
w1 = {weight1v1, weight2v1};
dw1 = ((weight1v1*x1 + b1 - z) * x1) +  ((weight2v1*x2 + b1 - z) * x2)
db1 = (w1.x + b1 - z)
```

```
0.225274
```

```
-2.63571
```

Student Response

Above is that first derivative w/ r/ t w. To use this we need to set a learning rate which is a small fraction of the opposite sign which will hep us steer to the minimum. Then I think we apply dw1 * lr to the two weights and repeat.

```
lr = -.02;
(* updating weights and bias*)
weight1v2 = (weight1v1 + lr) * dw1;
weight2v2 = (weight2v1 + lr) * dw1;
b2 = b1 + lr db1
w2 = {weight1v2, weight2v2};
dw2 = ((weight1v2 * x1 + b2 - z) * x1) + ((weight2v2 * x2 + b2 - z) * x2)
db2 = (w2.x + b2 - z)
h2 = {weight1v2, weight2v2}.{x1, x2} + b2
weight1v3 = (weight1v2 + lr) * dw2;
weight2v3 = (weight2v2 + lr) * dw2;
b3 = b2 + lr db2
w3 = {weight1v3, weight2v3};
dw3 = ((weight1v3 * x1 + b3 - z) * x1) + ((weight2v3 * x2 + b3 - z) * x2)
db3 = (w3.x + b3 - z)
h3 = {weight1v3, weight2v3}.{x1, x2} + b3
weight1v4 = (weight1v3 + lr) * dw3;
weight2v4 = (weight2v3 + lr) * dw3;
b4 = b3 + lr db3
w4 = {weight1v4, weight2v4};
dw4 = ((weight1v4 * x1 + b4 - z) * x1) + ((weight2v4 * x2 + b4 - z) * x2)
db4 = (w4.x + b4 - z)
h4 = {weight1v4, weight2v4}.{x1, x2} + b4
```

```
0.552714
```

```
-1.9812
```

```
-2.60768
```

```
0.554077
```

```
0.604868
```

```
-3.60266
```

```
-2.51997
```

```
0.641793
```

```
0.655267
```

```
1.62132
```

```
-2.56747
```

```
0.594293
```

Student Response

So we see some oscillation in the direction we want to go, with the output going from .55 to .64 (better) to .59 (not quite so good). I ported this to Python and ran it for 100 iterations:

```
for a in range(0,100):
  dw = (weight1 * x1 + b - z) * x1 + (weight2*x2 + b - z) * x2
  db = np.dot(w1,x) + b - z
  weight1 = (weight1 + lr) * dw
  weight2 = (weight2 + lr) * dw
  b = b + lr*db
  z_hat = (weight1 * x1) + (weight2 * x2) + b
  z_hat_list.append(z_hat[0])
```

After 100 iterations the predicted value (z_hat) was 3.137 which is only .024 away from z. The final weights were 0.024 & 0.0017 while the weight was 3.135. Not what I expected!

Student Response

Table of z_hat values:

[0.7469118045092145, 1.2178249962015135, 0.9531418330554644, 1.650280434181965, 1.2518144528381812, 1.417234611254642, 1.9042755001811758, 2.0964442728327555, 2.116486822606119, 2.329553246218378, 2.30537101291832, 2.484407222494278, 2.4676520690758856, 2.6008844908686126, 2.6020500867017478, 2.6951056527201818, 2.7104872511977787, 2.773870696410698, 2.796407517401047, 2.840117318650754, 2.864018944041885, 2.8953910775705385, 2.917409975822796, 2.940984700308126, 2.959905278253386, 2.9782511462693004, 2.99396769917868, 3.008541950854574, 3.0213955103662036, 3.0330937888057, 3.0435364278050545, 3.052970524691438, 3.061432559145767, 3.0690563520807723, 3.075907378276226, 3.0820738721870558, 3.0876193700240524, 3.092609329757934, 3.097098144002712, 3.1011370363496065, 3.1047708231180144, 3.1080404365542234, 3.1109823423220337, 3.1136295039283555, 3.1160114727889154, 3.1181548720410692, 3.1200836216556067, 3.1218192487957475, 3.1233811105181264, 3.1247866226634953, 3.126051450573761, 3.127189688098215, 3.1282140145034587, 3.1291358375115648, 3.1299654208280576, 3.1307119993134185, 3.131383882253817, 3.1319885463543513, 3.1325327192883132, 3.1330224548686054, 3.1334632006350587, 3.133859858653351, 3.1342168401973467, 3.1345381149398697, 3.134827255202177, 3.1350874757621394, 3.135321669667491, 3.1355324404567417, 3.1357221311488512, 3.135892850326628, 3.1360464956058087, 3.1361847747524143, 3.1363092246844437, 3.136421228570235, 3.136522031214413, 3.13661275290316, 3.1366944018632714, 3.1367678854739367, 3.1368340203562313, 3.136893541452764, 3.1369471101986317, 3.136995321874684, 3.1370387122249905, 3.137077763412168, 3.1371129093768646, 3.1371445406610414, 3.1371730087487175, 3.1371986299724783, 3.1372216890291917, 3.13724244214405, 3.137261119918113, 3.1372779298910283, 3.1372930588474217, 3.1373066748925993, 3.137318929320641, 3.137329958295659, 3.1373398843648963, 3.1373488178205045, 3.13735685792512, 3.1373640940148744]

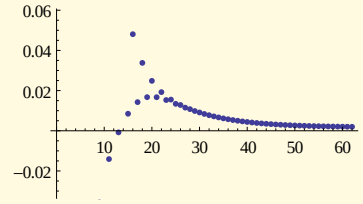
List of weight1 values:

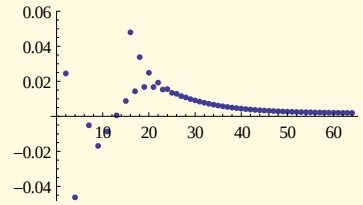
[0.19035131845537182, -0.168094486496656, 0.5981668782769539, -0.3363893243174112, 0.9935664812558096, 0.6641139683652943, -0.03945200497331897, 0.1877895495128278, -0.03569769456183224, 0.15611524264486254, -0.014085191684955723, 0.1028902542277942, -0.0007965102049963742, 0.07001282621019932, 0.00847734894515622, 0.04807419954047648, 0.014260712764875434, 0.03379447533498846, 0.01673291615926989, 0.024848035296563237, 0.016702769961909527, 0.01923582342759507, 0.015298851341837375, 0.01551344352368381, 0.013409874179823035, 0.012834014593128376, 0.01152101188862831, 0.010768663623149084, 0.009828283173139638, 0.009112015648877208, 0.008381563280662582, 0.007759877936252322, 0.007171708382865261, 0.006650317710926523, 0.00617067866013049, 0.005739573849335245, 0.005347232836869351, 0.004993137606234742, 0.004672302382948222, 0.004382510208584671, 0.00412048090786058, 0.0038838598818500447, 0.0036701546336631733, 0.003477266325587657, 0.0033031902631897314, 0.003146149084681 53, 0.003004502311623107, 0.002876772702927828, 0.002761613351354 2807, 0.0026578065301656765, 0.0025642473705297544, 0.0024799365320725 46, 0.002403969410695816, 0.0023355282448439 1, 0.0022738738047744338, 0.0022183383092290346, 0.002168318666812282, 0.002123270442812443, 0.002082702284693465, 0.002046170882113474 6, 0.002013276363981382, 0.0019836581261466167, 0.001956991039628 3006, 0.0019329820146887988, 0.001911366886279724, 0.001891907594597 3917, 0.001874389633468658, 0.0018586197428870062, 0.001844423823147 5401, 0.001831645050259937, 0.0018201421738434928,

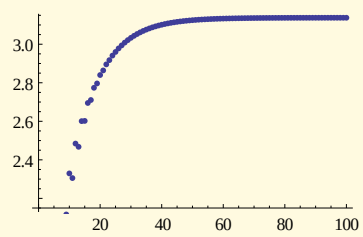
0.0018097879804110741, 0.0018004679064064686, 0.0017920787867670882, 0.001784527726057322,
0.0017777310804055555, 0.0017716135395582314, 0.0017661072993600843, 0.0017611513158776052,
0.0017566906332113989, 0.0017526757777995882, 0.0017490622212700717, 0.0017458098459709889,
0.0017428825878156218, 0.0017402479517128827, 0.0017378766695170017, 0.0017357424961988394,
0.0017338216619769357, 0.0017320928665062012, 0.0017305369143986 43, 0.0017291366019769213, 0.0017278761568653082,
0.0017267418034405944, 0.0017257208698781435, 0.0017248020168706 99, 0.0017239750390308638,
0.0017232307506965356, 0.0017225608844900458, 0.0017219579994726165, 0.0017214153985572861]
List of weight2 values:
[0.08684592714067586, 0.024472549603355893, 0.16851528660042628, -0.04626524156854489, 0.33301511585463855,
0.17318083939522963, -0.005117992110953513, 0.14155451108778083, -0.016897230395983714, 0.13448599511368872,
-0.008656148111073394, 0.09799395116443889, 0.0005528366168899172, 0.06907557566108927, 0.008742308891349868,
0.047935023749819866, 0.01429893549725152, 0.03377940969555788, 0.01673672387898442, 0.024846899020984678,
0.016703022502873383, 0.019235765108328192, 0.015298862389034686, 0.01551344150033378, 0.013409874500973175,
0.012834014545528889, 0.011521011894919686, 0.010768663622383367, 0.009828283173223977, 0.009112015648868686,
0.008381563280663367, 0.007759877936252256, 0.007171708382865267, 0.006650317710926523, 0.00617067866013049,
0.005739573849335245, 0.005347232836869351, 0.004993137606234742, 0.004672302382948222, 0.004382510208584671,
0.004120480907860588, 0.0038838598818500447, 0.0036701546336631733, 0.003477266325587657,
0.0033031902631897314, 0.00314614908468153, 0.003004502311623107, 0.002876772702927828, 0.002761613513542807,
0.002657806530165 6765, 0.0025642473705297544, 0.002479936532072546, 0.002403969410695816, 0.0023352824484391,
0.0022573873804774 4338, 0.0022183383092290346, 0.002168318666812282, 0.002123270442812443, 0.002082702284693465,
0.0020461708821134746, 0.0020013276363981382, 0.0019836581261466167, 0.0019569910396283006,
0.0019329820146887988, 0.001911366886279724, 0.0018919075945973917, 0.001874389633468658,
0.0018586197428870062, 0.0018444238231475401, 0.001831645050259937, 0.0018201421738434928,
0.0018097879804110741, 0.0018004679064064686, 0.0017920787867670882, 0.001784527726057322,
0.0017777310804055555, 0.0017716135395582314, 0.0017661072993600843, 0.0017611513158776052,
0.0017566906332113989, 0.0017526757777995882, 0.0017490622212700717, 0.0017458098459709889,
0.0017428825878156218, 0.0017402479517128827, 0.0017378766695170017, 0.0017357424961988394,
0.0017338216619769357, 0.0017320928665062012, 0.0017305369143986 43, 0.0017291366019769213, 0.0017278761568653082,
0.0017267418034405944, 0.0017257208698781435, 0.0017248020168706 99, 0.0017239750390308638,
0.0017232307506965356, 0.0017225608844900458, 0.0017219579994726165, 0.0017214153985572861]

```
zhat = ListPlot[{0.7469118045092145, 1.2178249962015135, 0.9531418330554644, 1.650280434181965,
    1.2518144528381812, 1.417234611254642, 1.9042755001811758, 2.0964442728327555,
    2.116486822606119, 2.329553246218378, 2.30537101291832, 2.484407222494278, 2.4676520690758856,
    2.6008844908686126, 2.6020500867017478, 2.6951056527201818, 2.7104872511977787,
    2.773870696410698, 2.796407517401047, 2.840117318650754, 2.864018944041885, 2.8953910775705385,
    2.917409975822796, 2.940984700308126, 2.959905278253386, 2.9782511462693004, 2.99396769917868,
    3.008541950854574, 3.0213955103662036, 3.0330937888057, 3.0435364278050545, 3.052970524691438,
    3.061432559145767, 3.0690563520807723, 3.075907378276226, 3.0820738721870558, 3.0876193700240524,
    3.092609329757934, 3.097098144002712, 3.1011370363496065, 3.1047708231180144, 3.1080404365542234,
    3.1109823423220337, 3.1136295039283555, 3.1160114727889154, 3.1181548720410692,
    3.1200836216556067, 3.1218192487957475, 3.1233811105181264, 3.1247866226634953,
    3.126051450573761, 3.127189688098215, 3.1282140145034587, 3.1291358375115648, 3.1299654208280576,
    3.1307119993134185, 3.131383882253817, 3.1319885463543513, 3.1325327192883132,
    3.1330224548686054, 3.1334632006350587, 3.133859858653351, 3.1342168401973467,
    3.1345381149398697, 3.134827255202177, 3.1350874757621394, 3.135321669667491, 3.1355324404567417,
    3.1357221311488512, 3.135892850326628, 3.1360464956058087, 3.1361847747524143,
    3.1363092246844437, 3.136421228570235, 3.136522031214413, 3.13661275290316, 3.1366944018632714,
    3.1367678854739367, 3.1368340203562313, 3.136893541452764, 3.1369471101986317, 3.136995321874684,
    3.1370387122249905, 3.137077763412168, 3.1371129093768646, 3.1371445406610414,
    3.1371730087487175, 3.1371986299724783, 3.1372216890291917, 3.13724244214405, 3.137261119918113,
    3.1372779298910283, 3.1372930588474217, 3.1373066748925993, 3.137318929320641, 3.137329958295659,
    3.1373398843648963, 3.1373488178205045, 3.13735685792512, 3.1373640940148744}];
weight1 = ListPlot[{0.19035131845537182, -0.168094486496656, 0.5981668782769539,
    -0.3363893243174112, 0.9935664812558096, 0.6641139683652943, -0.03945200497331897,
    0.1877895495128278, -0.03569769456183224, 0.15611524264486254, -0.014085191684955723,
    0.1028902542277942, -0.0007965102049963742, 0.07001282621019932, 0.008477348945155622,
    0.04807419954047648, 0.014260712764875434, 0.03379447533498846, 0.01673291615926989,
    0.024848035296563237, 0.016702769961909527, 0.01923582342759507, 0.015298851341837375,
    0.01551344352368381, 0.013409874179823035, 0.012834014593128376, 0.01152101188862831,
    0.010768663623149084, 0.009828283173139638, 0.009112015648877208, 0.008381563280662582,
    0.007759877936252322, 0.007171708382865261, 0.006650317710926523, 0.00617067866013049,
```

```
     0.005739573849335245,  0.005347232836869351,  0.004993137606234742,  0.004672302382948222,
     0.004382510208584671,  0.004120480907860588,  0.0038838598818500447,  0.0036701546336631733,
     0.003477266325587657,  0.003303190263189731 4,  0.00314614908468153,  0.003004502311623107,
     0.002876772702927828,  0.002761613351354280 7,  0.002657806530165676 5,  0.002564247370529754 4,
     0.002479936532072546,  0.002403969410695816,  0.00233552824484391,  0.002273873804774433 8,
     0.002218338309229034 6,  0.002168318666812282,  0.002123270442812443,  0.002082702284693465,
     0.002046170882113474 6,  0.002013276363981382,  0.001983658126146616 7}];
weight2 = ListPlot[{0.08684592714067586,  0.024472549603355893,  0.16851528660042628,
     -0.04626524156854489,  0.33301511585463855,  0.17318083939522963,  -0.005117992110953513,
     0.14155451108778083,  -0.016897230395983714,  0.13448599511368872,  -0.008656148111073394,
     0.09799395116443889,  0.0005528366168899172,  0.06907557566108927,  0.008742308891349868,
     0.047935023749819866,  0.01429893549725152,  0.03377940969555788,  0.01673672387898442,
     0.024846899020984678,  0.016703022502873383,  0.019235765108328192,  0.015298862389034686,
     0.01551344150033378,  0.013409874500973175,  0.012834014545528889,  0.011521011894919686,
     0.010768663622383367,  0.009828283173223977,  0.009112015648868686,  0.008381563280663367,
     0.007759877936252256,  0.007171708382865267,  0.006650317710926523,  0.00617067866013049,
     0.005739573849335245,  0.005347232836869351,  0.004993137606234742,  0.004672302382948222,
     0.004382510208584671,  0.004120480907860588,  0.0038838598818500447,  0.0036701546336631733,
     0.003477266325587657,  0.003303190263189731 4,  0.00314614908468153,  0.003004502311623107,
     0.002876772702927828,  0.002761613351354280 7,  0.002657806530165676 5,
     0.002564247370529754 4,  0.002479936532072546,  0.002403969410695816,  0.00233552824484391,
     0.002273873804774433 8,  0.002218338309229034 6,  0.002168318666812282,
     0.002123270442812443,  0.002082702284693465,  0.002046170882113474 6,  0.002013276363981382,
     0.001983658126146616 7,  0.001956991039628300 6,  0.001932982014688798 8}];
Show[weight1]
Show[weight2]
Show[zhat]
```







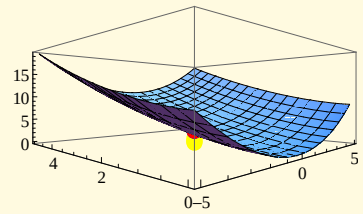
Student Response

Same plot as at the very beginning but showing the first few yhat points in yellow starting out below the surface and then moving up to the surface later.

```
Clear[f, g, test1, point1, point2]
test1[f_, g_] = (f^2)/3 + g^(3/2);
firstplot = Plot3D[test1[f, g], {f, -5, 5}, {g, 0, 5}];
test1[-1, 2]
point1 = Graphics3D[{PointSize[.05], Red, Point[{-1, 2, (1/3 + 2 Sqrt[2])}]}];
```

```
tp1 = Graphics3D[{PointSize[.05], Yellow, Point[{-1, 2, 0.7469118045092145}]}];
tp2 = Graphics3D[{PointSize[.05], Yellow, Point[{-1, 2, 1.2178249962015135}]}];
tp3 = Graphics3D[{PointSize[.05], Yellow, Point[{-1, 2, 0.9531418330554644}]}];
tp4 = Graphics3D[{PointSize[.05], Yellow, Point[{-1, 2, 1.650280434181965}]}];
tp5 = Graphics3D[{PointSize[.05], Blue, Point[{-1, 2, 3.1373066748925993}]}];
Show[firstplot, point1, tp1, tp2, tp3, tp4, tp5, ViewPoint -> {-1, -1, 0}]
```

$$\frac{1}{3} + 2\sqrt{2}$$



Student Response

Thank you!