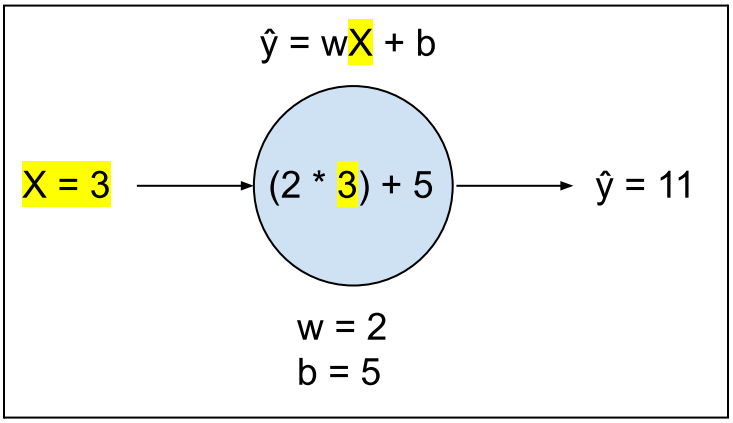
Chapter 08| Actual Neural Networks

~~~~~~

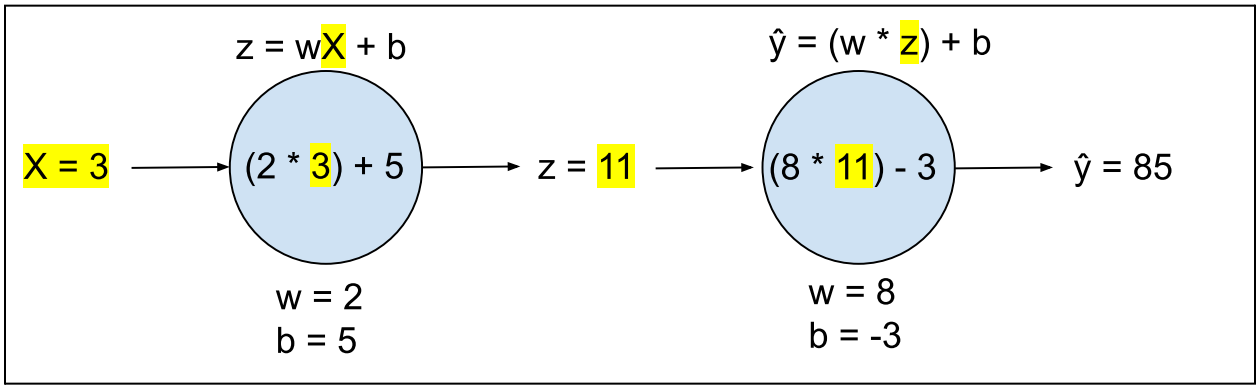
Problem With Just Simply Stacking Dense Layers On Top of Each Other

To start let's recall how only one dense layer works by itself alone:



We simplify the dense layer to one input feature and one neuron since the problem that we will present would actually be easier to explain this way. Recall that the output of a neuron for one input feature is given by the equation: ŷ = wX + b. If X is 3, w is 2, and b is 5, then the output of the neuron (ŷ) is 11.

Now if we end another layer right after our current layer:



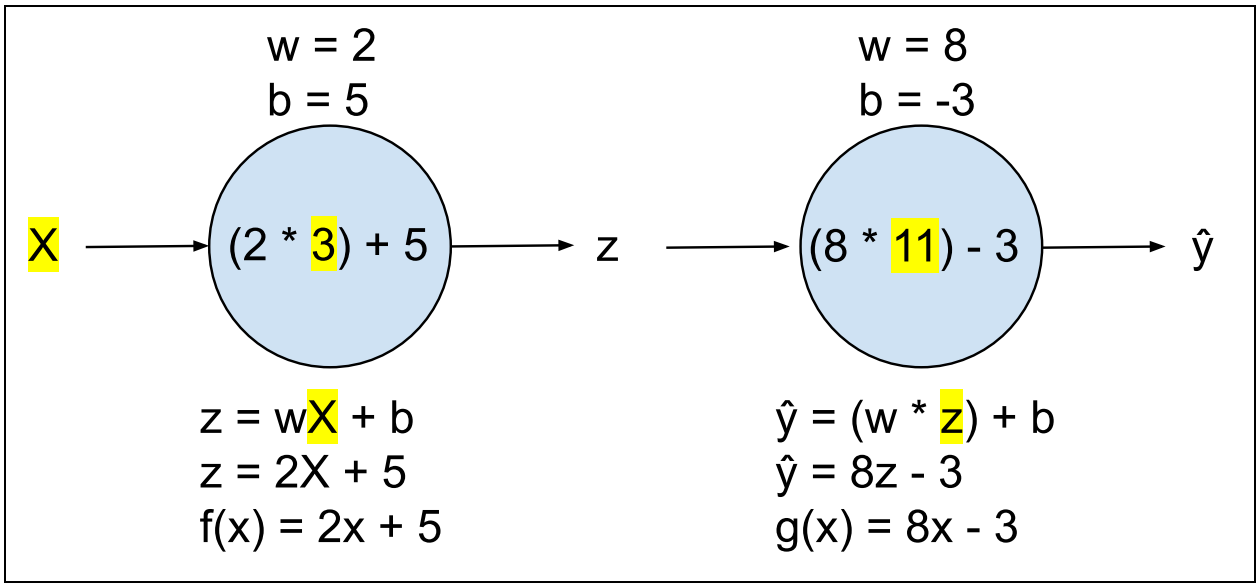
We see that for the first neuron's (in the first layer) output is given by z = wX + b, where we replace ŷ with z. The reason is because ŷ is reserved for the output of the model, which comes from the outputs of the neurons in the output layer (second layer in the second layer). For the first neuron, if X is 3, w is 2, and b is 5, then the output of the first neuron (z) is 11. To connect two layers of neurons together, the **input** for the **next** layer would be the **output** of the **current** layer. As a result, the output for the second layer is ŷ = (w * z) + b, where X is replaced by z since z is the output of the first neuron.
~~~~~~

Continuing with z = 11, if the second neuron's weight (w) is 8 and bias (b) is -3, the output of the model/second neuron (ŷ) is 85.

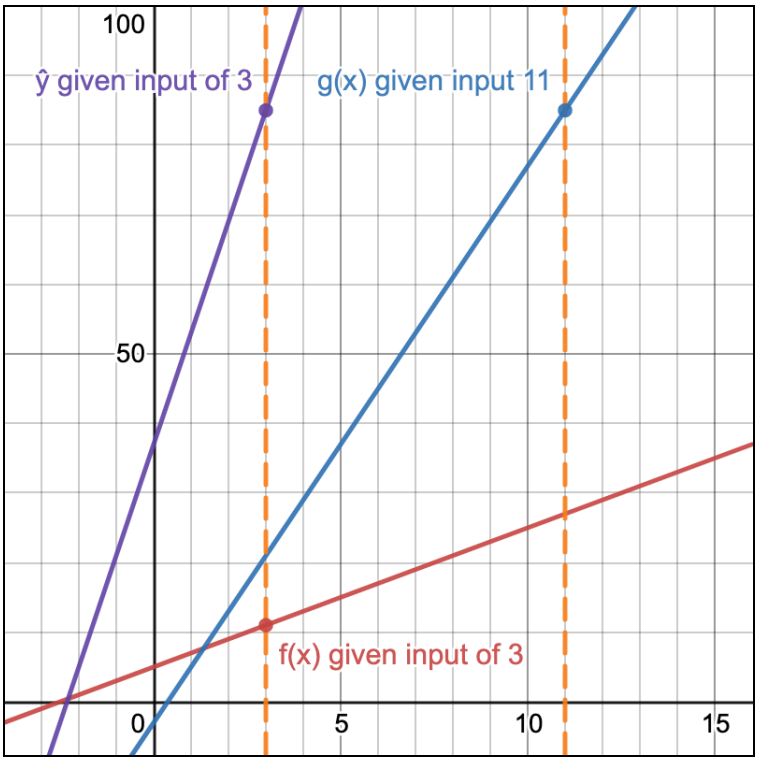
To understand where the problem is, let's substitute 2X + 5 in place of z for the calculation of the second of the second neuron: ŷ = [8 * (2X + 5)] - 3
Simplifying: ŷ = 16X + 40 - 3 → ŷ = 16X + 37

To check that this equation still works, plug in 3 in place of X and the result is still 85. Take a look at this new equation for the output of the model: ŷ = 16X + 37. What aspects of this equation seem familiar?

In reality, the output of the model is technically just the output of one neuron where the weight is 16 and the bias is 37. At the same time, this equation would still output a line based on the input of X. Another way to understand the output of the model is to express the outputs of each neuron as functions of X:



Here the output of the first neuron is f(x) and the output of the second neuron is g(x). Since the output of g(x) is the model's prediction, g(x) = ŷ. The input to g(x), the second neuron, is the output of f(x), the first neuron, whose input is X. As a result, ŷ = g(f(x)) or ŷ = (g ◦ f)(x). Pronounced "y hat is equal to g of f of x". Continuing, ŷ = g(2x + 5), and in mathematics, that is the same as ŷ = 8(2x + 5) - 3 = 16x + 40 - 3 = 16x - 7. Since, f(x) and g(x) are functions we can graph them to better understand how a linear function of a linear function still results in a linear function:



The red line represents f(x) and the output of 11 is the output of the first neuron given that x = 3.

The blue line represents g(x) and the output of 85 is the output of the second neuron given that the input of 11 is the output of the first neuron.

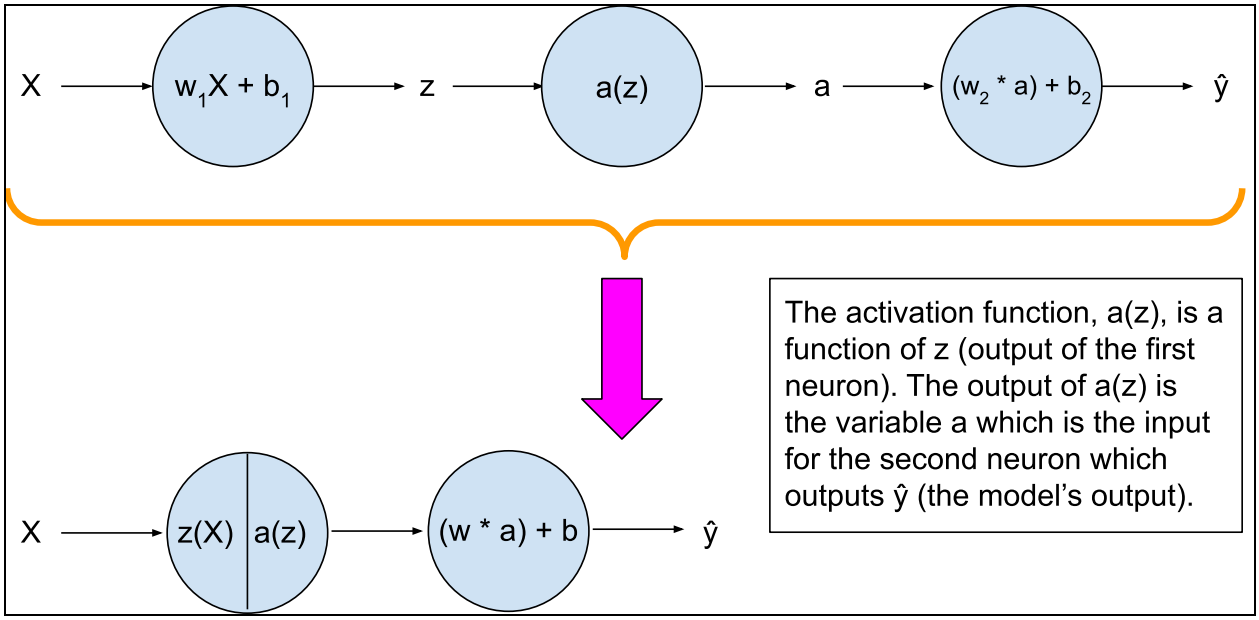
As a result, the purple line represents $\hat{y} = 16x - 7$ and the output of 85 is the output of the model given that x = 3.

Hence, simply stacking dense layers on top of each other does not achieve the goal of getting the model to predict nonlinear functions.

To get the model to predict nonlinear functions we must add in a nonlinear function **in-between** dense layers. These specific nonlinear functions that **"activate"** the model to output nonlinear functions are called **"activation functions"**. In the next section, we will introduce some common/prominent types of **activation functions**.

Solution to Making Neural Networks Actually Work: Activation Functions

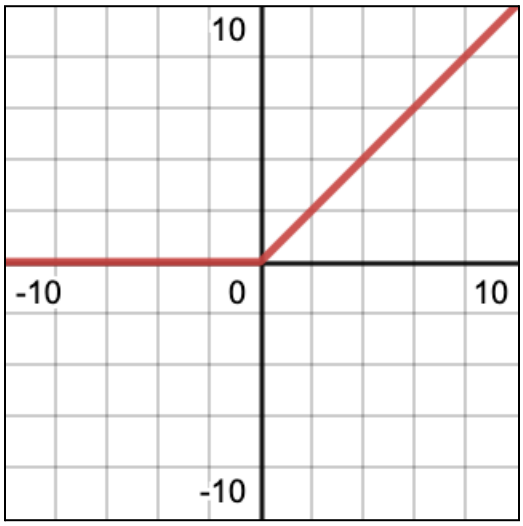
Here is how we would add an activation function in between two dense layers:



The subscripts of 1, on w and b are the weights and biases of the 1st neuron respectively. The subscripts of 2, on w and b are the weights and biases of the 2nd neuron respectively. Now the first neuron is actually a combination of z(X), the standard neuron function that outputs z given X, and a(z). So the first neuron outputs the output of the activation function. The output layer may or may not have an activation function depending on the task the neural network is being used for. Now let's start introducing the possible a(z) functions.

In the last few years the **Re**ctified **L**inear **U**nit (**ReLU**) has become very popular. The function itself is defined as a(z) = max(0, z). As long as x is greater than 0, ReLU's output value would just be its input value. Otherwise, ReLU outputs 0.

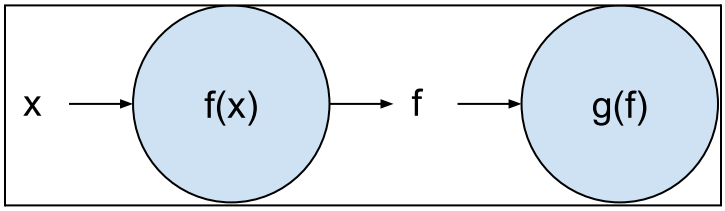
Here is a graph of ReLU, where the y-axis is the output of the ReLU function (the variable: *a*) given by the x-axis which is the input of the ReLU function (the variable: *z*).



Its popularity mainly comes from the simplicity of the nonlinear function. ReLU is nonlinear since a linear function has a constant slope, for input values greater than or equal to 0, the slope is 1 and for any other input feature, the slope is 0. However, the slope of 0 for any other input

values is a downfall of ReLU. Recall the chain rule from calculus used in training to determine the update for the parameters. If the output of our function. f(x), g(x) where x is the output of f(x):

The derivative of **g(f)** w.r.t. (with respect to) x is equal to the derivative of **g(f)** w.r.t. to f multiplied by the derivative of f(x) w.r.t. x. $\frac{dg}{dx} = \frac{dg}{df} * \frac{df}{dx}$



So we are trying to find the slope of x on the function **g** at the point where x is (imagine a graph with **g** on the y-axis and x on the x-axis). This is equal to the slope of f on the function **g** at the point where **f** is (imagine a graph with **g** on the y-axis and f on the y-axis) multiplied by the slope of x on the function f at the point where x is (imagine a graph with f on the y-axis and x on the x-axis).

Applying the chain rule our context, if we were to find the derivative of the first neuron w.r.t. the weight and bias: $\frac{da}{dw} = \frac{da}{dz} * \frac{dz}{dw}$ and $\frac{da}{db} = \frac{da}{dz} * \frac{dz}{db}$

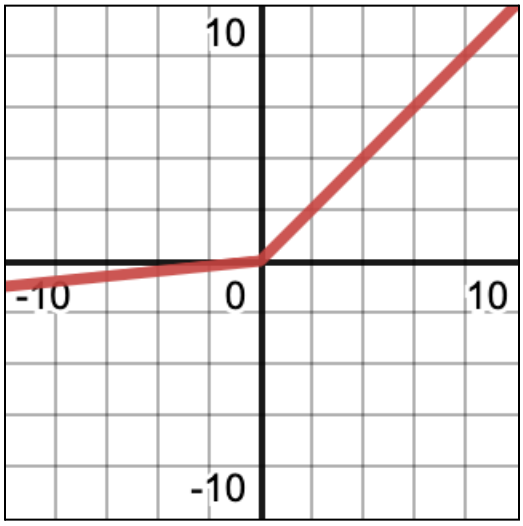
Here $\frac{da}{dz}$ is the equivalent of the slope of *z* on the activation function *a* which is the graph of the ReLU. Since the slope for z < 0 is 0, $\frac{da}{dz} = 0$, and multiplying $\frac{dz}{dw}$ or $\frac{dz}{db}$ by 0 would result in 0. Since the derivative is 0, the neuron's weight and bias of that neuron would not update, this is trouble!

One way to prevent z to be less than 0 is to not have a learning rate too high. A high learning rate may update the weights and biases in the initial stages of training much too aggressively which would make z less than 0 on any datapoint. That neuron would then be considered **"dead"** since it would always output zero after the activation function no matter the input as if the input does not matter to the neuron.

A fix to this, would be to use **Leaky ReLU**: a(z) = max(α*z, x) where α is a positive value less than 1. For example, if α = 0.1, here is the graph with the output of Leaky ReLU on the y-axis and the input of Leaky ReLU on the x-axis:

The negative input values now have a slope of 0.1 (α) and any other input values still have a slope of 1. Hence, Leaky ReLU is also a nonlinear function because its slope is not the same everywhere. To understand why the right side of the graph remains

unchanged but the left side of the graph changes, let's input -5, 0, and 5.



a(-5) = max((0.1 * -5), -5) = max(-0.5, -5) = -0.5
a(0) = max((0.1 * 0), 0) = max(0, 0) = 0
a(5) = max((0.1 * 5), 5) = max(0.5, 5) = 5

As you can see, for negative inputs, Leaky ReLU just makes the negative less negative due to α (a positive value less than 1). Less negative values are greater than negative values by itself, so Leaky ReLU outputs the less negative values.

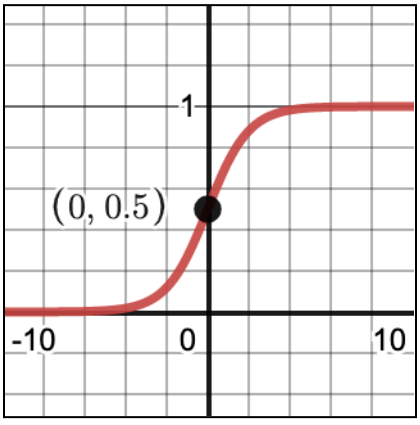
On the other hand, Leaky ReLU makes the positive less positive due to α (a positive value less than 1). Positive values by itself are greater than less positive values, so Leaky ReLU outputs the positive values by itself. An input of zero, still gives 0 since zero multiplied by α is still zero which is the exact same value as the original input of zero.

Despite that now $\frac{da}{dz}$ would never be 0 which may prevent **"dead" neurons**, but the results of using Leaky ReLU are inconsistent due to only some people reporting success by using Leaky ReLU in place of ReLU's limitation.

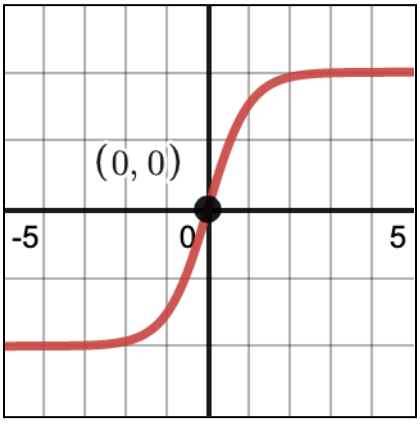
So far we covered two activation functions which are defined by 2 unique linear functions, making them nonlinear. Now let's cover **Sigmoid** and **Tanh** that are nonlinear in nature (not composed of unique linear functions).

Sigmoid: a(z) = $\frac{1}{1+e^{-z}}$

Tanh: a(z) = $\frac{2}{1+e^{-2z}} - 1$



The *e* is not another variable, instead the *e* is an irrational number, more specifically, Euler's number ≈ 2.71828183.



Graph of output of sigmoid on y-axis and input to sigmoid on x-axis.

Squashes input to between 0 and 1.

Graph of output of tanh on y-axis and input to tanh on the x-axis.

Squashes input to between -1, and 1

A problem with sigmoid is that it will output data to the next layer centered at 0.5 rather than 0. As a result, we would also have to do normalization or standardization in between the neurons of the layers. Tanh on the other hand, does output data to the next layer centered at 0.

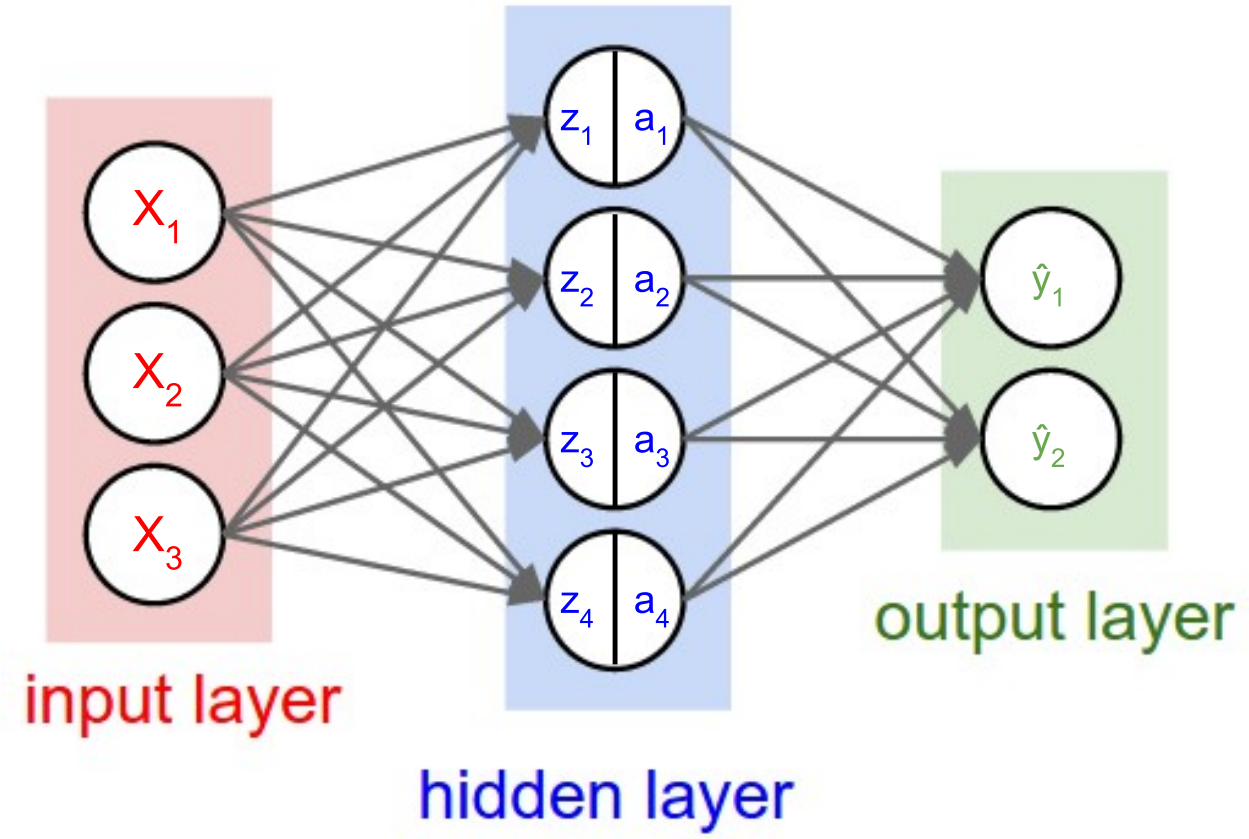
A problem for both Sigmoid and Tanh however, is that for inputs more and more negative or inputs more and more positive, the slope at these points gets closer and closer to a value of 0 as seen from the far left and right of both graphs, the graph is almost a horizontal line. As a result, at very negative/positive input values, $\frac{da}{dz}$ would be close to 0, which would **"kill"** the weight and bias gradient since the updates would be very small.

We are going to use the ReLU activation function for our neurons in our regression task, since a lower learning rate can mitigate the amount of **"dead neurons"**.

In the next section, we will bring the concept of activation functions into the neurons of each layer to stack layers together, creating an **architecture** that receives multiple input features and uses multiple hidden layers.

Neural Network Architecture

Let's start with a neural network with only one hidden layer:



Original Image From CS231n Course Notes from Stanford University

Here in the hidden layer: $z_1 = (w_{1,1} * X_1) + (w_{1,2} * X_2) + (w_{1,3} * X_3) + b_1$ and $a_1 = \max(0, z_1)$, $z_2 = (w_{2,1} * X_1) + (w_{2,2} * X_2) + (w_{2,3} * X_3) + b_2$ and $a_2 = \max(0, z_2)$, and so on. Assume that the "w" and "b" referred to correspond to only the hidden layer's weights and biases respectively.

In the output layer: $\hat{y}_1 = (w_{1,1} * a_1) + (w_{1,2} * a_2) + (w_{1,3} * a_3) + (w_{1,4} * a_4) + b_1$ and $\hat{y}_2 = (w_{2,1} * a_1) + (w_{2,2} * a_2) + (w_{2,3} * a_3) + (w_{2,4} * a_4) + b_2$. Assume that the "w" and "b" referred to correspond to only the output layer's weights and biases respectively. Notice that the outputs of the hidden layer acts as input features to the output layer.

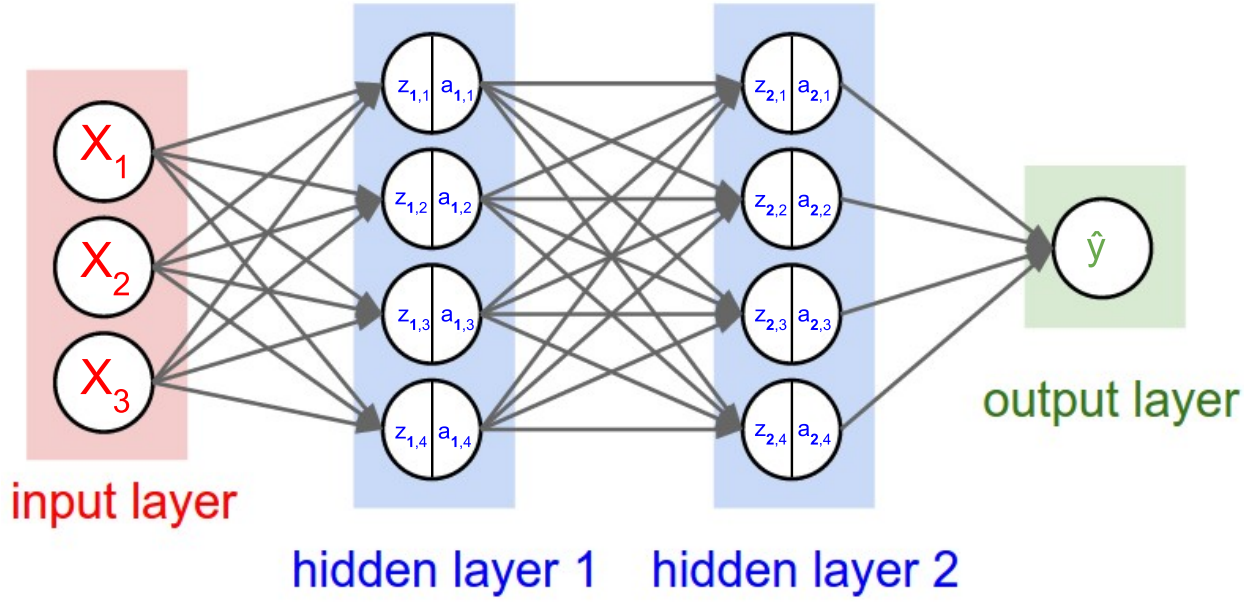
As a result, there are 3 input features in the input layer, 4 **hidden units** (neurons) in the hidden layer and 2 output features in the output layer. As you can see, **the activations** ("a") for each neuron do not depend on the "z" of other neurons. However, the

activations of each neuron do affect the output layer. Each **activation** is passed as input to the neurons in the output layer, just like how each input feature is passed as input to the neurons in the hidden layer. So the output layer receives 4 input features due to the 4 hidden units (neurons) in the hidden layer.

The purpose of the activation functions are mainly to create nonlinear functions but you can also think of activation functions creating better input features so that the output layer can predict better.

In summary: since there are 3 input features in the input layer, each neuron in the hidden layer receives 3 input features as input. Since there are 4 **hidden units** (neurons) in the hidden layer, each neuron in the output layer receives 3 input features as input. Notice that the number of input features and output features of the neural network **still depend on the dataset** but the number of hidden units of the hidden layer **is determined by the human creating the neural network**.

Now let's talk about a neural network with multiple hidden layers:



Original Image From CS231n Course Notes from Stanford University

Here in the **first** hidden layer: $z_{1,1} = (w_{1,1} * X_1) + (w_{1,2} * X_2) + (w_{1,3} * X_3) + b_1$ and $a_{1,1} = \max(0, z_{1,1})$, the first subscript on "z" and "a" represents the number of the hidden layer the neuron belongs to (hidden layer **1**) and the second subscript represents the neuron number (neuron 1) in that layer. $z_{1,2} = (w_{2,1} * X_1) + (w_{2,2} * X_2) + (w_{2,3} * X_3) + b_2$ and

$a_{1,2} = \max(0, z_{1,2})$, and so on. The first subscript on "z" and "a" represents the number of the hidden layer the neuron belongs to (hidden layer **1**) and the second subscript represents the neuron number (neuron 2) in that layer. Assume that the "w" and "b" referred to correspond to only hidden layer 1's weights and biases respectively.

In the **second** hidden layer: $z_{2,1} = (w_{1,1} * a_{1,1}) + (w_{1,2} * a_{1,2}) + (w_{1,3} * a_{1,3}) + b_1$ and $a_{2,1} = \max(0, z_{2,1})$. The first subscript on "w" represents the neuron number (1) in that layer and the second subscript represents the "weight" number of that neuron. $z_{2,2} = (w_{2,1} * a_{1,1}) + (w_{2,2} * a_{1,2}) + (w_{2,3} * a_{1,3}) + b_2$ and $a_{2,2} = \max(0, z_{2,2})$, and so on. The first subscript on "w" represents the neuron number (2) in that layer and the second subscript represents the "weight" number of that neuron. Assume that the "w" and "b" referred to correspond to only hidden layer 2's weights and biases respectively. Notice that the outputs of hidden layer 1 acts as input features to hidden layer 2.

Here we only have one output feature in the output layer where $\hat{y} = (w_1 * a_{2,1}) + (w_2 * a_{2,2}) + (w_3 * a_{2,3}) + (w_4 * a_{2,4}) + b$. Assume that the "w" and "b" referred to correspond to only hidden layer 2's weights and biases respectively. Notice that the outputs of hidden layer 2 acts as input features to the output layer.

As a result, there are 3 input features in the input layer, 4 **hidden units** (neurons) in each hidden layer and 1 output feature in the output layer. As you can see, **the activations** ("a") for each neuron **in the same layer** do not depend on the "z" of other neurons **in the same layer**. However, the **activations** of each neuron do affect the next layer. Each **activation** is passed as input to the neurons in the next layer, just like how each input feature is passed as input to the neurons in hidden layer 1. So hidden layer 2 receives 4 input features due to the 4 hidden units (neurons) in hidden layer 1 and the output layer receives 4 input features due to the 4 hidden units (neurons) in hidden layer 2.

In summary: since there are 3 input features in the input layer, each neuron in hidden layer 1 receives 3 input features as input. Since there are 4 **hidden units** (neurons) in hidden layer 1, each neuron in hidden layer 2 receives 4 input features as input. Since there are 4 **hidden units** (neurons) in hidden layer 2, the neuron in the output layer receives 4 input features as input. Notice that the number of hidden units in each hidden layer is still **determined by the human creating the neural network**. To keep things simple, in this book, our neural networks with multiple hidden layers will have the same number of hidden units in each hidden layer.

Now that we understand the properties/characteristics of neural network architectures that use multiple hidden layers and hidden units, in the next section we will be implementing the ReLU activation function forward and backward methods. At the same time, the dense layer's backward method needs a modification.

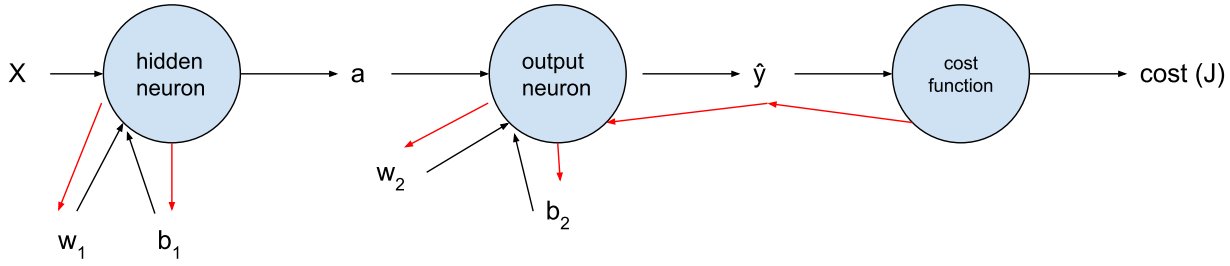
ReLU Activation Function & Dense Layer Backward Modification

Here is the code for the ReLU activation function:

```
class ReLU_Activation:
    def forward(self, inputs):    # inputs are outputs from dense layer
        self.inputs = inputs    # save inputs for backward method
        self.outputs = np.maximum(0, inputs)

    def backward(self, dvalues):    # dvalues has same shape as inputs
        self.dinputs = dvalues * (self.inputs >= 0)
```

Since the slope depends on the input, we save the inputs in the forward method so we no longer need to have another argument in the backward. Next, we can find the maximum between 0 and each element in the *inputs* matrix using *np.maximum()*. As a result, *self.outputs* has the same shape as *inputs*. In the backward method, we get a boolean matrix from numpy by the inequality (self.inputs >= 0). Elements that are greater than equal to 0 would receive a value of True (1), other elements would receive a value of False (0). Now when we multiply *dvalues* by this, the elements that received a value of False would have a slope of 0. This is due to how numpy converts booleans to binary values. A value of "True" converts to 1, and a value of "False" converts to 0.

Now that we covered ReLU, why exactly do we have to modify the dense layer's backward method? Let's say we have one input feature, one hidden unit for only one hidden layer and one output feature:



The red lines represent that we have already covered the calculations of finding the derivatives/slopes. We have covered the backward method of the cost function, and the backward method of each neuron to their parameters (weights and biases) which can help us determine the update direction and update amount for these parameters. These derivatives are multiplied together starting from the cost function due to the calculus chain rule.

However, the output neuron in the output layer has no derivative connection to the hidden neuron in the hidden layer so the derivatives of the hidden neuron in the hidden layer have no connection to the cost function. This is a problem, since the cost function is how we determine the amount of error in the model and without connection, the updates to the parameters are meaningless as they have no idea if increasing or decreasing the parameter would decrease the cost.

As a result, in addition to calculating the derivatives of each neuron w.r.t. their parameters in the backward method, we need to calculate the derivatives of each neuron w.r.t. the input. For the output neuron, this would connect "a" (the output of the hidden neuron) to the derivative of the output neuron which is connected to the cost function. So that when we do backpropagation, *dvalues* for the hidden layer would be *dinputs* from the output neuron. The *dinputs* of the hidden neuron, however, would just be useless since the input layer has no neurons which mean no parameters to update.

So how exactly do we find the derivatives of each neuron w.r.t. the input?
Recall that ŷ is given by the function, D(X, w, b), so let's find the analytical derivative w.r.t. X (the input) by using a **tiny** interval, [X-ε, X+ε]:

$$\frac{\partial D(X,w,b)}{\partial X} = \frac{D(X+\epsilon,w,b) - D(X-\epsilon,w,b)}{(X+\epsilon)-(X-\epsilon)} = \frac{[w(X+\epsilon)+b] - [w(X-\epsilon)+b]}{(X+\epsilon)+(-X+\epsilon)}$$

Substitute [w(X + ε) + b] in place of D(X + ε, w, b).
Substitute [w(X - ε) + b] in place of D(X - ε, w, b).

Distribute the negative in front of the denominator's second term (x_1): X + ε.

$$= \frac{[w(X+\epsilon)+b] + [-w(X-\epsilon)-b]}{(X-X)+(\epsilon+\epsilon)} = \frac{[wX+w\epsilon+b] + [-wX+w\epsilon-b]}{2\epsilon}$$

After distributing the negative in front of the numerator's second term (y_1): [w(X - ε)+b] and grouping the like terms in the denominator, distribute w to (X + ε) for the

numerator's first term (y_2) and distribute -w to (X - ε) for the numerator's first term (y_1). Next we simplify the denominator by combining the grouped like terms.

~~~~~~

$$= \frac{(wX - wX) + (w\epsilon + w\epsilon) + (b - b)}{2\epsilon} = \frac{2w\epsilon}{2\epsilon} = \frac{w\epsilon}{\epsilon} = w$$

Now we can group the like terms in the numerator which can be simplified by being combined. To continue simplifying, we divide 2 (from numerator) by 2 (from denominator) and divide ε (from numerator) by ε (from denominator) to get **w**.

Here is a PDF of steps that got us the analytical derivative. Now let's understand why the analytical derivative is so simple. Remember that D(X, w, b) = wX + b, which originates from y = mx + b. In "y = mx + b" m is the slope, **when** X is plotted on the x-axis and y is plotted on the y-axis. Since "w" represents "m" in AI, the slope of D(X, w, b) w.r.t. the input (X) is just **w**.

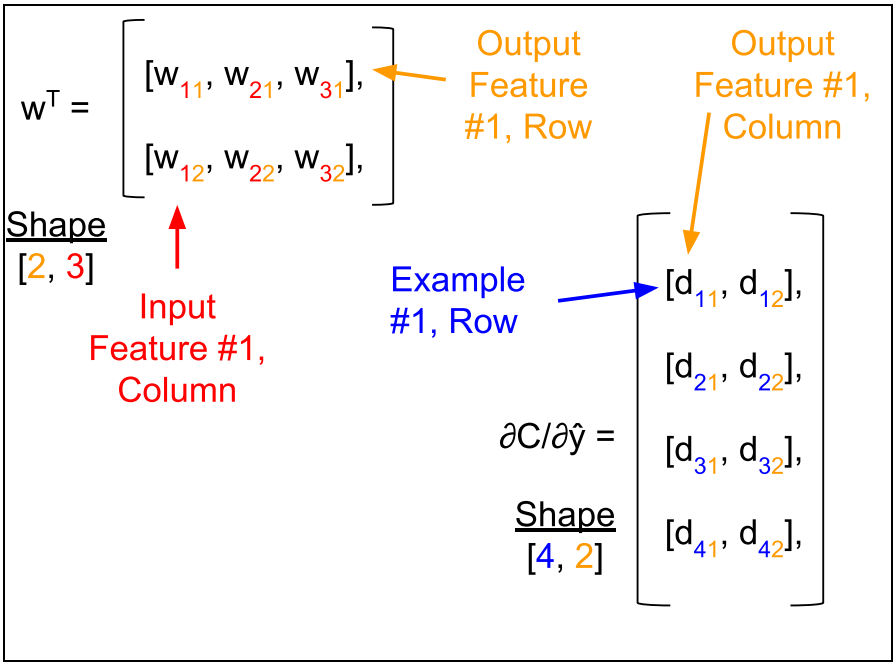
As a result, to get *dinputs* of a neuron we need to multiply the *dvalues* by "w". Since both "w" and *dvalues* are matrices, we must consider their shapes in order to complete matrix multiplication where a matrix of shape (**a**, b) multiplied by another matrix of shape (b, **c**) results in a matrix of shape (**a**, **c**). Also, *dinputs* must have the same shape as X (the input).

dinputs shape = X shape: (# of examples, # of input features)
"w" has shape: (# of input features, # of neurons/# of output features)
dvalues has shape: (# of examples, # of neurons/# of output features)

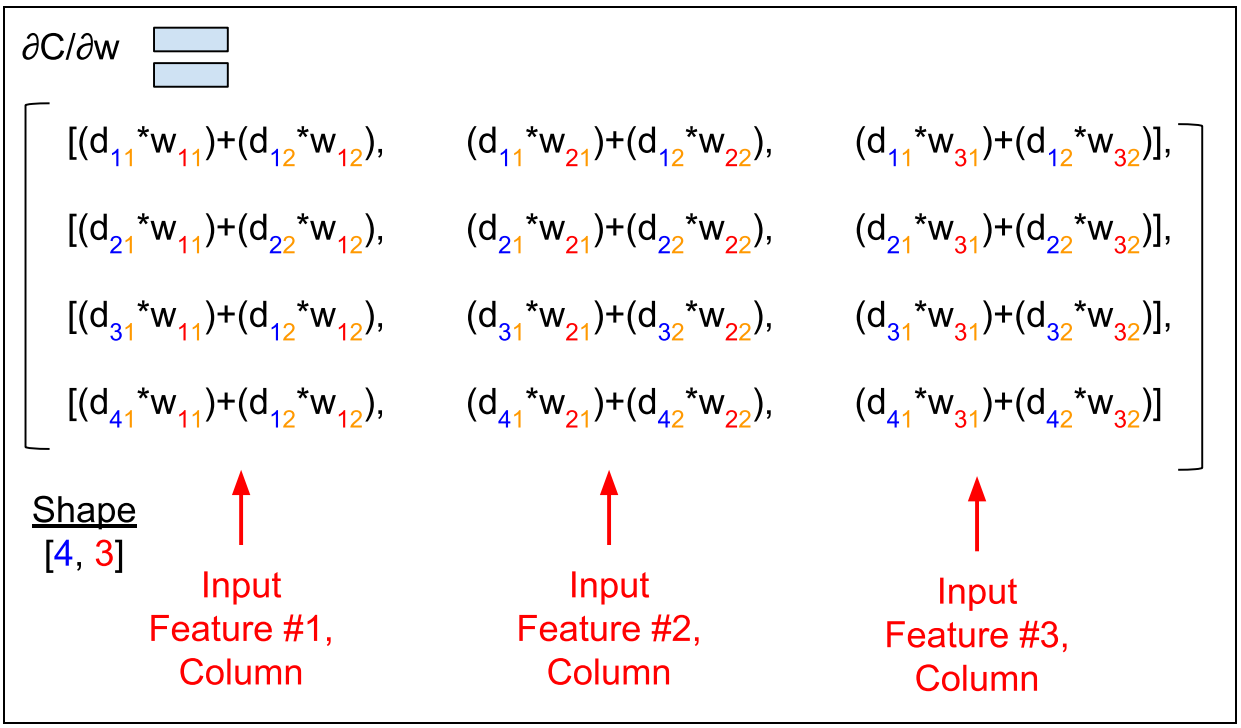
We need to have *dvalues* for sure as the first matrix since its rows are equal to the rows of *dinputs*. For matrix multiplication to work, the first matrix's column has to equal the second matrix's rows. To do this, we need to transpose "w" in which w's rows and columns are flipped. So we have (# of examples, # of neurons/# of output features) * (# of neurons/# of output features, # of input features). Before moving on to the code, let's understand what exactly is happening in the dot product.

This is what we have if we have a model that takes in 3 input features and 2 output features. Also, let's say we have 4 examples:

Note that in w^T our indices are based on the format of the **original matrix** (w)
~~~~~~



Each element in each matrix is indexed by **subscripts** where the first number indicates the row and the second number indicates the column. To make notation easier, in the $\partial C/\partial \hat{y}$ matrix we use indices of a matrix "d", so assume that the $\partial C/\partial \hat{y}$ matrix is the **same exact** as the "d" matrix.



Now there is a lot to unpack but understand that each examples' $\partial C/\partial w$ which is a column vector has 3 elements (one for each **input feature**). Let's first understand the building block, for example: $(d_{11}*w_{11})$, we multiply the derivative of the cost function w.r.t. first example's 1st output feature by the weight of the 1st neuron corresponding to the first input feature to get the derivative of the cost function on the first neuron w.r.t. the 1st example's 1st input feature. As a result, the derivative of the cost function w.r.t. the

1st example's 1st input feature, $(\partial C/\partial X)_{11}$, is the **sum** of all the derivatives of the cost function on each neuron w.r.t. the 1st example's 1st input feature. The same rule applies to other examples.

To understand why we add, we need to understand that the derivative of the cost function w.r.t. some variable **x** is the same as the **effect on the cost** when **slightly increasing x** (by definition of the analytical derivative). When we **slightly increase** the 1st example's 1st input feature, the output of **each neuron's** output feature will change, and as a result, will **effect the cost**. As seen from the numerical derivative section in unit 5, when two variables change (say *f* and *g*), the **effect on the cost** is about the sum of the **effect on the cost** when slightly increasing *f* (leave other independent variables constant, $\partial C/\partial f$) with the **effect on the cost** when slightly increasing *g* (leave other independent variables constant, $\partial C/\partial g$). This is my interpretation of the sum, so keep your eyes open if you can find a better interpretation that is more intuitive.

Now that we understand what the dot product does, here is the code:

```
class Dense_Layer:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros([1, n_neurons])

    def forward(self, inputs):    # inputs is X
        self.inputs = inputs
        self.outputs = np.dot(inputs, self.weights) + self.biases

    def backward(self, dvalues):
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        self.dinputs = np.dot(dvalues, self.weights.T)
```

Everything is the same except the last line where we added the definition of *self.dinputs* as equal to the dot product of *dvalues* and *self.weights* transposed.

Now we are finally ready to create neural networks that have at least one hidden layer.

Finally Training An Actual Neural Network (One Hidden Layer)

In the next few pages, we will be going over the code that will create our one hidden layer neural network, so here is the full code, if you'd rather see it all at once:

```
import matplotlib.pyplot as plt    # version 3.4.0
import numpy as np    # version 1.22.2
import pandas as pd    # version 1.4.0
np.random.seed(0)    # For repeatability

class Dense_Layer:…

class ReLU_Activation:…

class MSE_Cost:…

class SGD_Optimizer:…
```

First, the imports and seeding of Numpy's random generator.

Next we have the **modified** dense layer class, ReLU activation class, MSE cost class, and SGD optimizer class.

The contents of each class have been collapsed to save space. Again full code is available here.

```
dataset = pd.read_csv("2639600.csv")
dataset = dataset.drop(labels=['STATION', 'NAME', 'DATE'], axis=1)
datset = dataset.to_numpy()

LENGTH_OF_MOVING_AVERAGE = 125
modified = np.empty([(len(datset)-LENGTH_OF_MOVING_AVERAGE)+1, 2])
for i in range(len(modified)):
    modified[i] = np.mean(dataset[i:i+125], axis=0)

X = np.expand_dims(np.array(range(len(modified))), axis=1)
y = modified.copy()

def normalize(X, min=0):
    X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))    # normalize to [0, 1] range
    if min == -1:    # if the minimum value for the range is -1 (assuming [-1, 1] range)
        X = (2 * X) - 1
    return X

X = normalize(X, min=-1)    # normalize data to [-1, 1] range
```

Continuing on, we read the csv file, and convert only the important data to a NumPy array. After that we modify the original data, by replacing it by using a 125-day moving average, we normalize the data to a [-1, 1] by using a user-defined function. Now that we have data, we need a model to pass in the data.
This model is now a..., neural network!

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer = Dense_Layer(n_inputs, n_hidden)
        self.activation = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.cost_function = MSE_Cost()

        self.trainable_layers = [self.hidden_layer, self.output_layer]

    def forward(self, inputs, y_true):
        self.hidden_layer.forward(inputs)
        self.activation.forward(self.hidden_layer.outputs)
        self.output_layer.forward(self.activation.outputs)
        self.cost = self.cost_function.forward(self.output_layer.outputs, y_true)

    def backward(self, y_true):
        self.cost_function.backward(self.output_layer.outputs, y_true)
        self.output_layer.backward(self.cost_function.dinputs)
        self.activation.backward(self.output_layer.dinputs)
        self.hidden_layer.backward(self.activation.dinputs)
```

Now we define the model in a class that initiates a hidden layer, activation function, an output layer, and a cost function. The __init__ method takes 3 parameters, *n_inputs* (the number of input features), *n_hidden* (the number of hidden units), and *n_outputs* (the number of output features). Lastly, *self.trainable_layers* is a list containing the layers that have parameters which are helpful to have for the optimizer during training.

In the forward method, we require 2 arguments, *inputs*, the input data to the model and *y_true* to use in the cost function. First, we get the outputs of the hidden layer, then the output of the activation function, and finally the output of the model (output layer). Lastly, we find the cost using the cost function and store it in *self.cost*.

In the backward method, we only require 1 argument, *y_true* for the backward method of the cost function. Use the chain rule of *dinputs* from the cost function to call the backward method of the output layer, *dinputs* from the output layer to call the activation function backward method, and *dinputs* from the activation function backward method.

Creating a class allows us to get the output of the model and cost in one line (by calling the forward method) and the derivatives of the cost function w.r.t. to each parameter in one line (by calling the backward method). So now we prevent the need of copy-and-pasting code and make our code in the training loop cleaner.

```
model = Neural_Network(1, 4, 2)    # 1 input feature, 4 hidden units, 2 output features
optimizer = SGD_Optimizer(learning_rate=0.01)   # learning rate is 0.01

model.forward(X, y)
print("Initial Cost:", model.cost)
```

Now we initialize the model with 1 input feature, 4 hidden units, and 2 output features. To be cautious, since we are using ReLU, we do not want to update too aggressively so that our 4 neurons "die" (output less than 0 for any input), so the learning rate is set to 0.001. Next get the predictions of the model to see its starting cost.

```
cost_history = []    # append to this in training loop
for epochs in range(10):
    # forward pass
    model.forward(X, y)
    cost_history.append(model.cost)

    # check for dead neurons
    n = sum((model.hidden_layer.outputs < 0).all(axis=0))    # number of dead neurons
    percentage = (n / model.hidden_layer.biases.size) * 100    # percentage of dead neurons
    print(f"Cost: {model.cost} - Percentage of Dead neurons: {percentage}%")

    # backward pass
    model.backward(y)
    for layer in model.trainable_layers:
        optimizer.update_params(layer)

# Check New Cost
model.forward(X, y)
cost_history.append(model.cost)
print("Final Cost:", model.cost)
```

Before the training loop we create an empty list as a placeholder to store cost values as we train. We train for 10 epochs and after that check the new cost. In the training loop, we do a forward pass and backward pass. Another for-loop is used to loop over each of the layers that have parameters (*model.trainable_layers*) and pass that as input to the optimizer to update. The code in between the forward and backward pass checks for dead neurons.

If all (*.all()*) rows (examples, axis 0) for a column of outputs of the hidden layer are less than 0, that means if all outputs for an example of a neuron are less than 0, the neuron is dead. We count the number of columns (neurons) . This is true by using *sum()* to get *n*. To figure out what percentage of the neurons are dead, we divide *n* by the number of neurons in the hidden layer and then multiply by 100. Since the number of neurons in

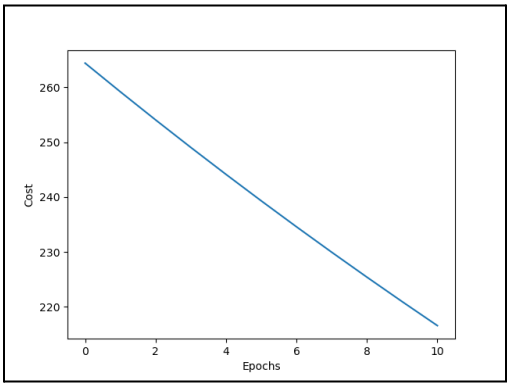
the hidden layer is equal to the number of biases we use *.size* on the vector of biases to get the number of elements in that vector.

```
# Model Predictions vs. Correct Answer Graph

fig = plt.figure()    # create a graphing space
# plot correct values
plt.plot(y[:, 0], label="TMAX")
plt.plot(y[:, 1], label="TMIN")

# plot predicted values
plt.plot(model.output_layer.outputs[:, 0], label="ŷ -- TMAX")
plt.plot(model.output_layer.outputs[:, 1], label="ŷ -- TMIN")

# customization
plt.xlabel("MODIFIED day of the year")
plt.ylabel("temperature (°C)")
plt.legend()

# Save and Close Graph
fig.savefig("training.png")
plt.close()

# Cost History Graph
fig = plt.figure()
plt.plot(cost_history)
plt.xlabel("Epochs")
plt.ylabel("Cost")
fig.savefig("cost.png")
plt.close()
```

Lastly, we print out the cost and the percentage of dead neurons together. Note that we only check the hidden layer for dead neurons since in the output layer there is no activation function that could result in the parameters not being able to update.

At the end, we compare the model predictions and the correct answers using a graph and then create a graph to visualize the cost during training.

Here is the cost graph:

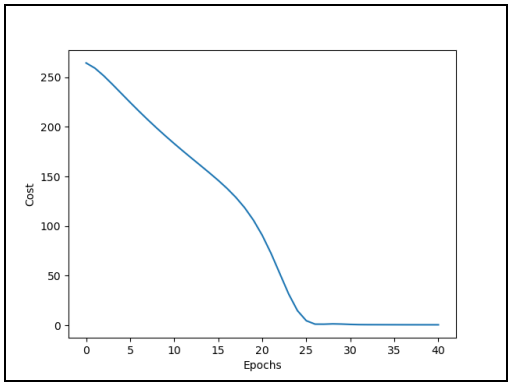


Let's speed up training with a momentum of 0.5 (continue to be cautious for dead neurons) and increase the number of epochs to 40.

Here is the code. Now in the training loop we modify the code block in between the forward and backward passes:

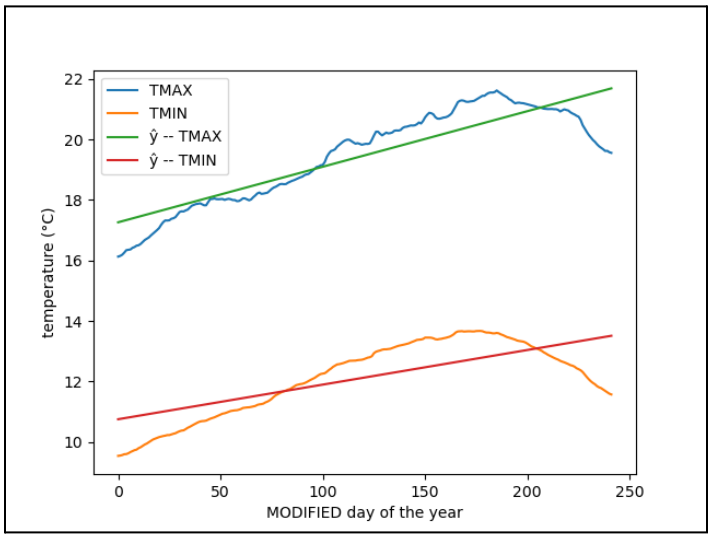
```
# check for dead neurons
if epochs % 4 == 0:
    n = sum((model.hidden_layer.outputs < 0).all(axis=0))    # number of dead neurons
    percentage = (n / model.hidden_layer.biases.size) * 100    # percentage of dead neurons
    print(f"Cost: {model.cost} - Percentage of Dead neurons: {percentage}%")
```

We print every 4 epochs so that your terminal is not flooded with outputs of the print statements. Now let's look at the cost graph:



The model's prediction is still linear!!!! Why is this the case? At the same time, none of our neurons in the hidden layer are dead. In reality, the problem is simple.

It seems like the model has learned but looking at the graph of the model's predictions versus the correct answers, we have a problem:



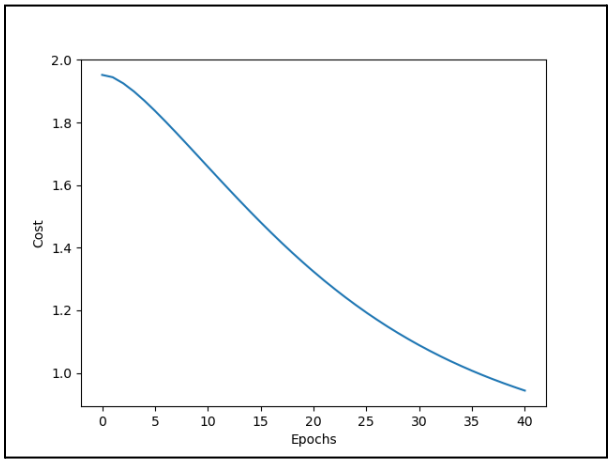
In the output layer the bias for TMAX and TMIN has to update from 0 all the way to about 16 and 10 respectively while the weights are relatively small. The temperature increases by about 5 in 150 days, which indicates that small weights are needed so the weight does not have to update as much as the bias. This means that the weights have to wait for the biases to reach close to their optimal value before being able to obtain a value that would make sense.

Andrej Karpathy (Previous Director of AI at Tesla) suggested on his blog that for regression, it is best to set the output layer biases as the mean of the data. So here is the new code:

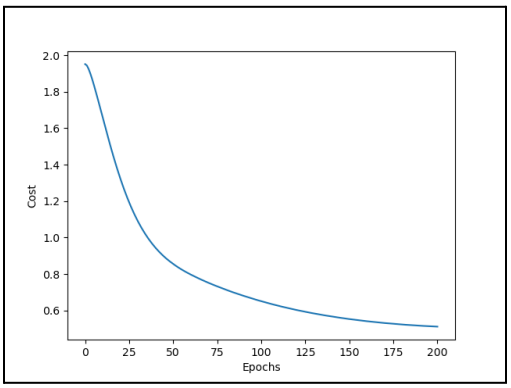
```
model = Neural_Network(1, 4, 2)    # 1 input feature, 4 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.01)

# set the biases of the output layer to the mean of the training data
model.output_layer.biases = y.mean(axis=0, keepdims=True)

model.forward(X, y)
print("Initial Cost:", model.cost)
```

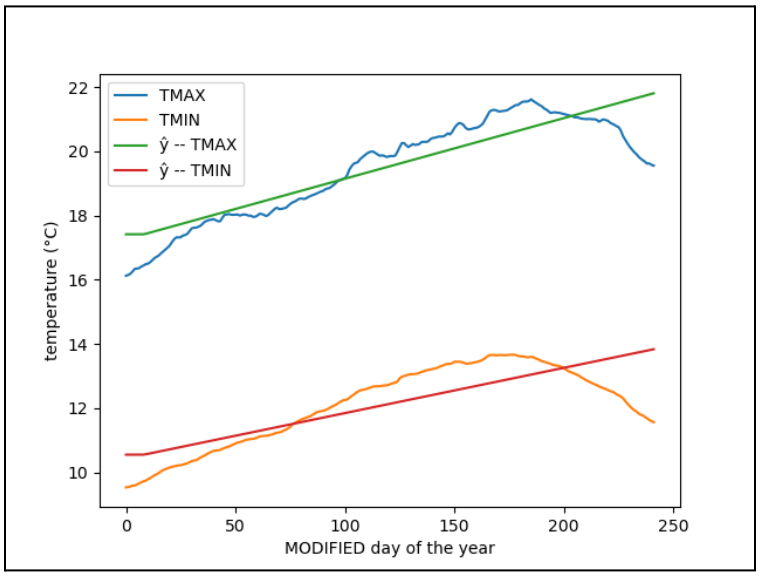
Right after we initialize the model, and right before we get the initial cost, we find the average of the correct answer by using the rows (axis 0). Since there are 2 columns, we get a vector of 2 elements but set *keepdims* equal to True so that the output is still a matrix which is the exact shape of the output layer's biases. At the same time, we

switched to RMSProp with a learning rate of 0.01 due to the different amount of updates required between the weights and biases. Here is the cost graph:



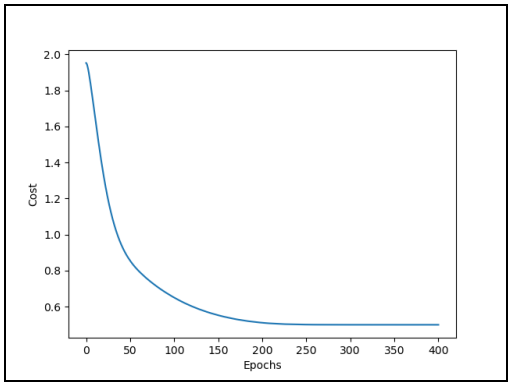
More training is required. Let's push to 200 epochs and print every 20 epochs. Here is the code and cost graph:



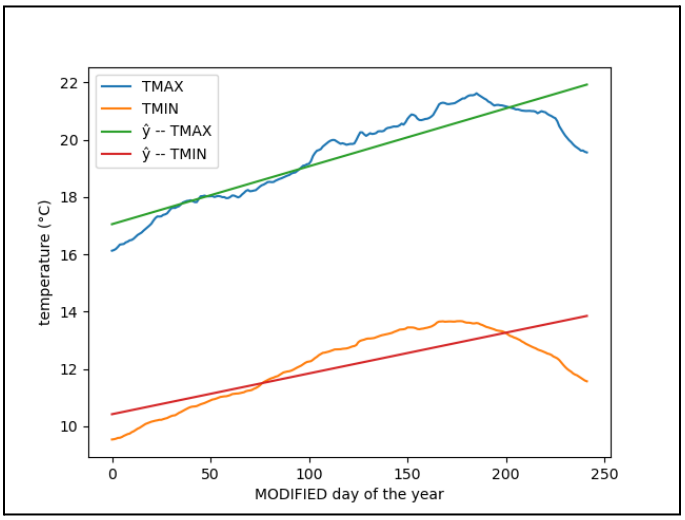


The model has almost **converged** (a model **converges** when it is **smart**, which is when there is no need for more training). The model's predictions compared to the correct answer graph is on the right.

Now we see the nonlinear graph, very close to the shape of the ReLU graph where smaller input values result in the same output and other values result in outputs that increase linearly. Now let's train it for 400 epochs so that the model can converge, and now print every 40 epochs. Here is the code and cost graph:



The model has converged at a cost of about 0.5 without any dead neurons. Now let's compare the model's predictions with the correct answers:



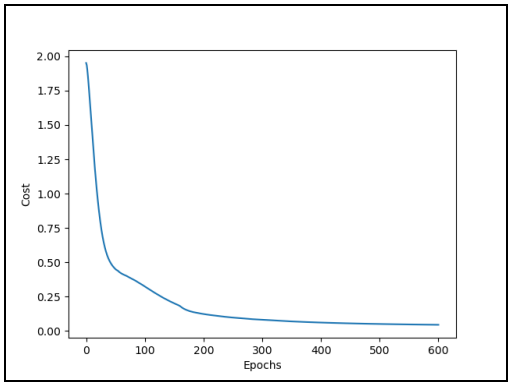
The model's predictions became linear!!!! Still no dead neurons, so what's causing this? Recall that during training (at 200 epochs), the model had used the ReLU activation so the hidden neurons have decided that the lowest cost by using **this neural network architecture** (one hidden layer of 4 hidden units) can be achieved best through a linear prediction.

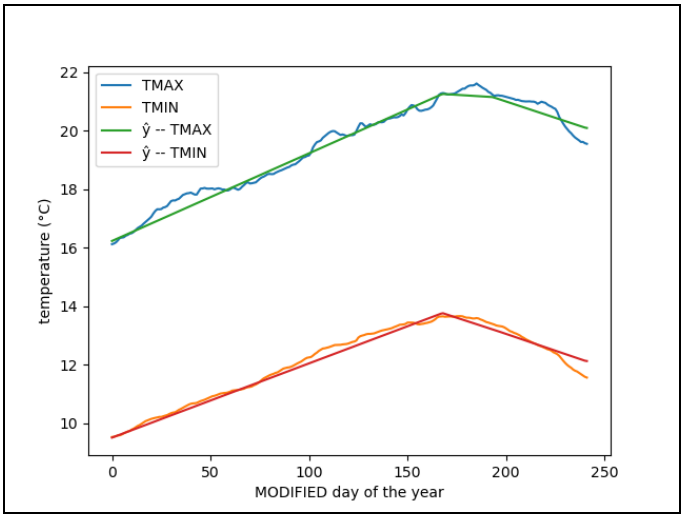
Only adding a hidden layer with 4 hidden units for this dataset does not make a big difference compared to a model without a hidden layer. Essentially, adding a hidden layer with 4 hidden units is not sufficient for the model to produce more complex functions to fit the dataset. Time to increase the number of hidden units!!!!

To increase the number of hidden units to 16, simply pass in 16 as the second argument to defining the model. At the same time increase the number of epochs to 600 since the model needs more time to train more neurons and print every 60 epochs.
Here is the code:

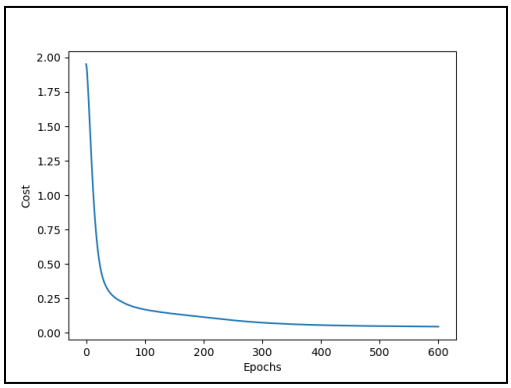
```
model = Neural_Network(1, 16, 2)    # 1 input feature, 16 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.01)
```

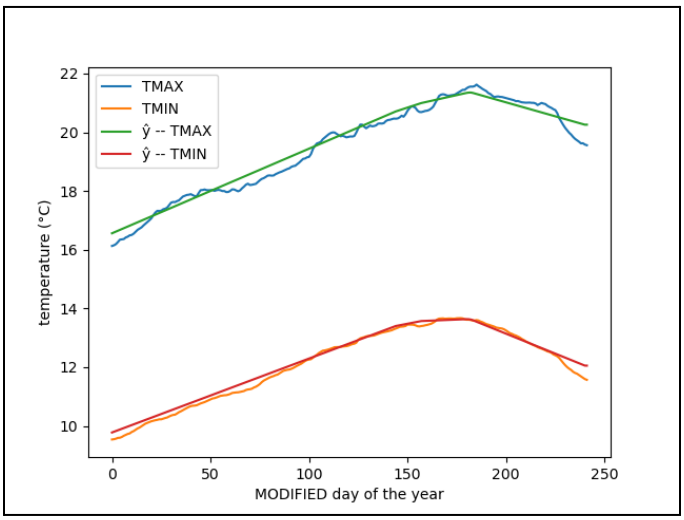
Here is the cost graph and the graph that puts the model's predictions into perspective:



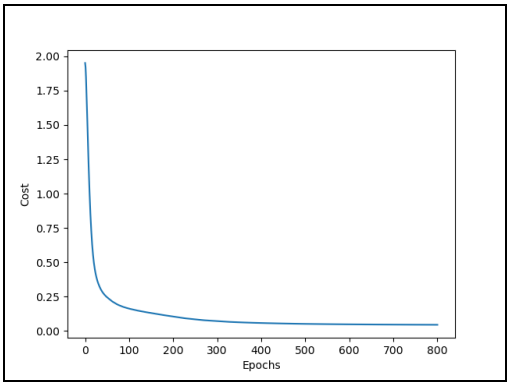


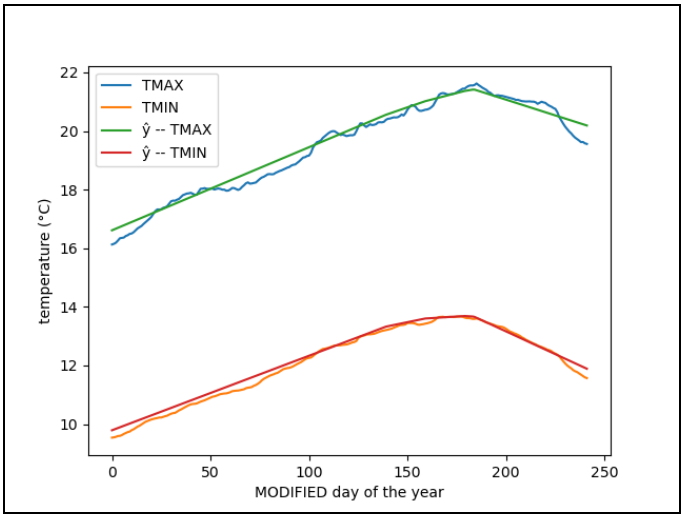
Wow!!!! Take a few moments to analyze the changes. The model has figured out the increasing to decreasing trend of the data and the nonlinear nature of its function is easy to spot. Taking a look at training, the cost is under 0.05 but 56.25% of the neurons in the hidden layer are dead. Meaning the model in reality is using 7 hidden units (16 neurons * [1 - 0.5625]). Now let's try 24 hidden units. Here is the code and its graphs:





We see that the prediction for TMIN is now more smooth while the model needs to work on TMAX more. The cost dropped to about 0.045 with 50% of the neurons dead (the model is using 12 hidden units). Let's try 36 hidden units, push the number of epochs to 800 and print every 80 epochs. Here is the code and its graphs:





The cost is about 0.044 which is a very small decrease while using 200 more epochs has 12 more hidden units. The model's predictions on the graph that compares it to the right answers, see barely an improvement. This is a problem, since for the model to improve, we add more hidden units which make the model take longer to train and improvements get smaller and smaller. At this point, when increasing the number of hidden units results in less and less improvements, it is time to start increasing the number of hidden layers. This will increase the variety of different functions the model can be able to produce.

An analogy is that, increasing the number of neurons, makes the brain bigger while increasing the number of hidden layers, makes the brain able to think more critically. Well how can we prove that this analogy is true? We have to experiment by increasing the number of hidden layers and interpret our results!! So in the next section, we will train neural networks with multiple hidden layers (more than one hidden layer this time). See you there! :)

Neural Networks with Multiple Hidden Layers

Let's start with 2 hidden layers. Here is the code if you prefer to read the script at once:

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation2 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.cost_function = MSE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2, self.output_layer]

    def forward(self, inputs, y_true):
        self.hidden_layer1.forward(inputs)
        self.activation1.forward(self.hidden_layer1.outputs)
        self.hidden_layer2.forward(self.activation1.outputs)
        self.activation2.forward(self.hidden_layer2.outputs)
        self.output_layer.forward(self.activation2.outputs)
        self.cost = self.cost_function.forward(self.output_layer.outputs, y_true)

    def backward(self, y_true):
        self.cost_function.backward(self.output_layer.outputs, y_true)
        self.output_layer.backward(self.cost_function.dinputs)
        self.activation2.backward(self.output_layer.dinputs)
        self.hidden_layer2.backward(self.activation2.dinputs)
        self.activation1.backward(self.hidden_layer2.dinputs)
        self.hidden_layer1.backward(self.activation1.dinputs)
```

2 hidden layers required 2 activation functions, making all methods of the class more complex and longer. At the same time, since we have multiple hidden layers now, we define a function to get the number of dead neurons when receiving a specific layer:

```
def get_dead(layer):
    n = sum((layer.outputs < 0).all(axis=0))
    return (n / layer.biases.size) * 100
```

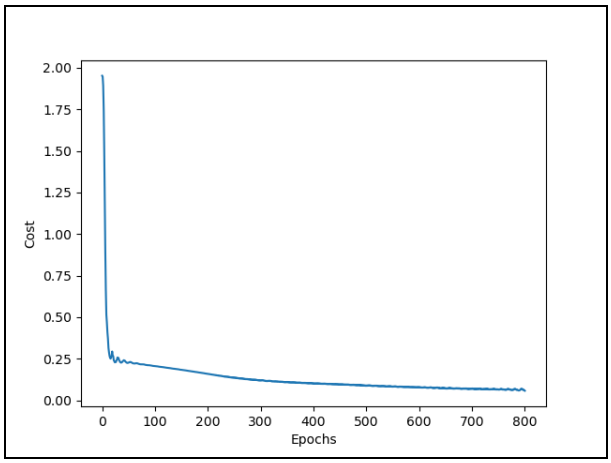
We can define a function since all dense layers are structured the same way.

```
model = Neural_Network(1, 36, 2)    # 1 input feature, 36 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.01)
```

There are no changes required when initializing the model for 2 hidden layers since we have already changed the __init__ method of the class. Continuing with the same learning rate, number of hidden units for each hidden layer, and epochs, the last change is to use the *get_dead()* function for the model's hidden layers:

```
# check for dead neurons
if epochs % 80 == 0:
    # cost & percentage of dead neurons
    print(f"Cost: {model.cost} - Dead1: {get_dead(model.hidden_layer1)}% - " + \
        f"Dead2: {get_dead(model.hidden_layer2)}%")
```

Dead1 refers to the percentage of dead neurons in the first hidden layer, while Dead2 refers to the percentage of the dead neurons in the second hidden layer. Since the print statement would be too long in one whole line, we add an explicit line break in the string. Here is the cost graph:

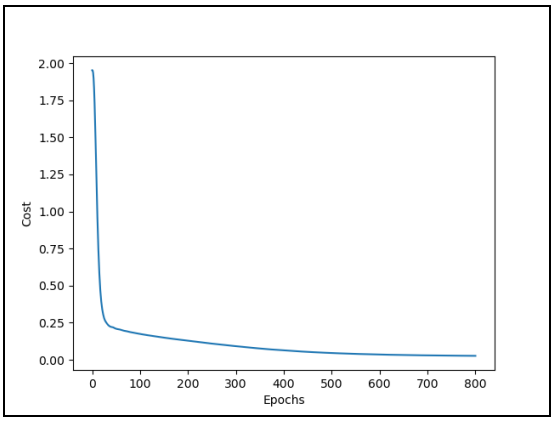


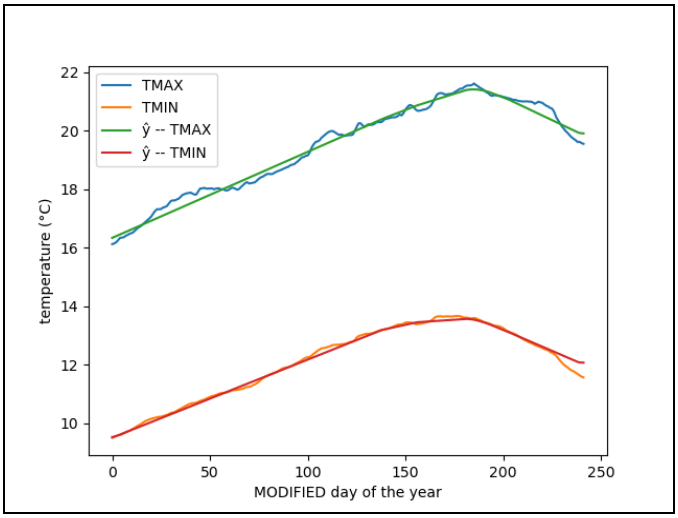
We see that the cost is jumping around in the beginning due to a high learning rate and jumping around in the end due to the learning rate being too high towards the end of training. Remember, that to decrease the learning rate during training, we need to use learning rate decay. So copy-and-paste the class of the learning rate decayer right after the class of the RMSProp optimizer and right before we define the dataset. Here is the code.

Now we define the optimizer with a learning rate of 0.006 and use a decay rate of 0.01.

```
model = Neural_Network(1, 36, 2)    # 1 input feature, 36 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.006)
# every 100 epochs, denominator increases by 1
decayer = Learning_Rate_Decayer(optimizer, 0.01)
```

Recall that when we use a learning rate decay, the learning rate as a function of the number of epoch is defined as $\alpha(t) = \frac{a_0}{1+kt}$ where α is the new learning rate, α_0 is the initial learning rate (0.006), t is the number of epochs, and k is the decay rate (0.01). As a result, the denominator increases by 1 every time the product of k and t increases by 1, and by setting k equal to 0.01, we have the denominator to increase by 1 every 100 epochs. Everything else stays the same so here are the graphs:



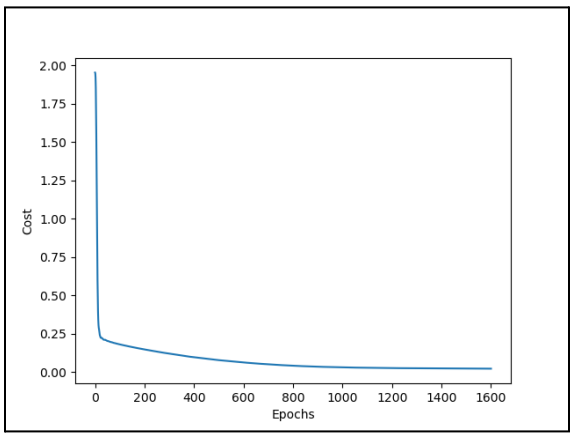


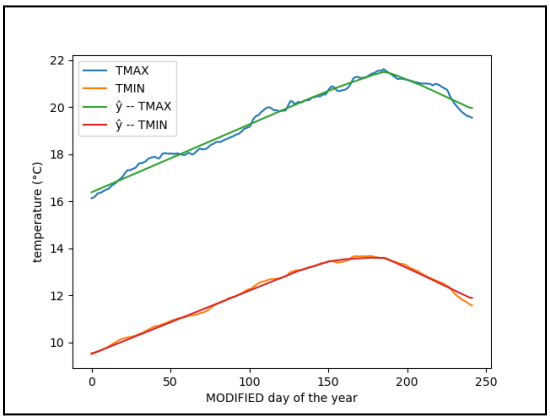
Now the cost is decreasing much more smoothly (about 0.025 now) and a nice improvement in the prediction for TMAX. The prediction for the last few days of TMIN still needs improvement.

Now let's increase the number of hidden units from 36 to 64. Here is the code:

```
model = Neural_Network(1, 64, 2)    # 1 input feature, 64 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.006)
# every 100 epochs, denominator increases by 1
decayer = Learning_Rate_Decayer(optimizer, 0.01)
```

To do this we change the second argument to initialize the model to 64. Since we have doubled the number of hidden units, the model might need more time to train since there are more parameters to update. As a result, double the number of epochs (1600 epochs) and print every 160 epochs. Here are the graphs:





The cost is now about 0.022 and the model's prediction has not changed much. Familiar? Time to increase the number of hidden layers!

Moving on to 3 hidden layers. As we have seen from the last few examples, when we increase the number of hidden layers, we need to also increase the number of hidden units and number of epochs for training. Also, the learning rate has to decrease.

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation2 = ReLU_Activation()
        self.hidden_layer3 = Dense_Layer(n_hidden, n_hidden)
        self.activation3 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.cost_function = MSE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2,
                                 self.hidden_layer3, self.output_layer]

    def forward(self, inputs, y_true):
        self.hidden_layer1.forward(inputs)
        self.activation1.forward(self.hidden_layer1.outputs)
        self.hidden_layer2.forward(self.activation1.outputs)
        self.activation2.forward(self.hidden_layer2.outputs)
        self.hidden_layer3.forward(self.activation2.outputs)
        self.activation3.forward(self.hidden_layer3.outputs)
        self.output_layer.forward(self.activation3.outputs)
        self.cost = self.cost_function.forward(self.output_layer.outputs, y_true)

    def backward(self, y_true):
        self.cost_function.backward(self.output_layer.outputs, y_true)
        self.output_layer.backward(self.cost_function.dinputs)
        self.activation3.backward(self.output_layer.dinputs)
        self.hidden_layer3.backward(self.activation3.dinputs)
        self.activation2.backward(self.hidden_layer3.dinputs)
        self.hidden_layer2.backward(self.activation2.dinputs)
        self.activation1.backward(self.hidden_layer2.dinputs)
        self.hidden_layer1.backward(self.activation1.dinputs)
```

3 hidden layers require 2 activation functions and now we have 4 dense layers with parameters the model has to update during training. As the complexity of the model changes, we have to add more and more steps to the forward and backward methods.

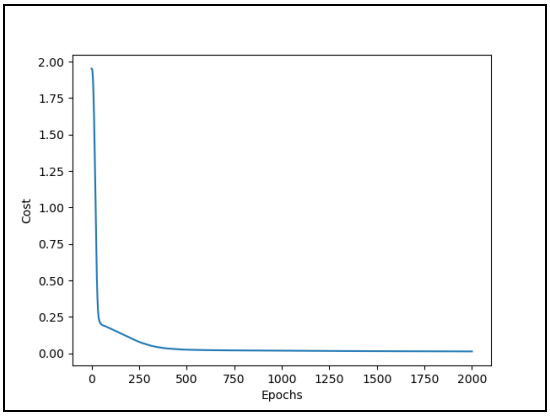
We initialize the model with 128 hidden units for each hidden layer and the learning rate has decreased to 0.001 (equivalent of 1e-3). Since we are starting with an even smaller learning rate, we do not have to decay the learning rate as aggressively, so now the decay rate is 0.005 (which you can interpret as increasing the denominator by 1 every 200 epochs). Here is the code:

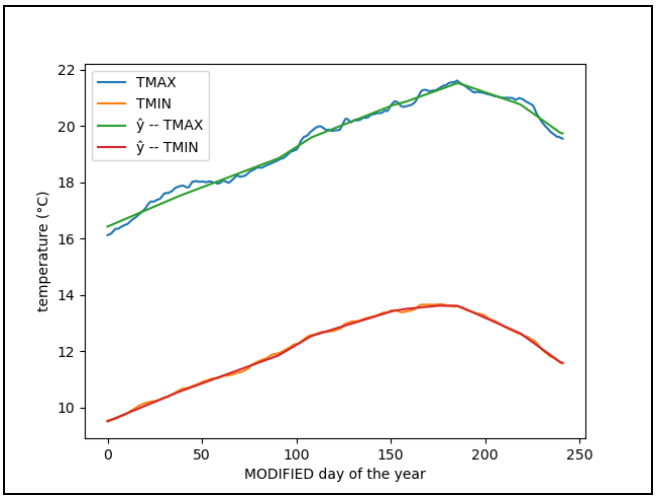
```
model = Neural_Network(1, 128, 2)    # 1 input feature, 128 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.001)
# every 200 epochs, denominator increases by 1
decayer = Learning_Rate_Decayer(optimizer, 0.005)
```

Continuing on, we train for 2000 epochs and print every 200 epochs. Since there are 3 hidden layers, we need to change the print statement to include the percentage of dead neurons in the 3rd hidden layer (represented by Dead3).

```
# check for dead neurons
if epochs % 200 == 0:
    # cost & percentage of dead neurons
    print(f"Cost: {model.cost} - Dead1: {get_dead(model.hidden_layer1)}% - " + \
        f"Dead2: {get_dead(model.hidden_layer2)}% - Dead3: {get_dead(model.hidden_layer3)}%")
```

Everything else stays the same and the execution time for this script should be less than 30 seconds. The cost is now about 0.014 and here are the graphs:

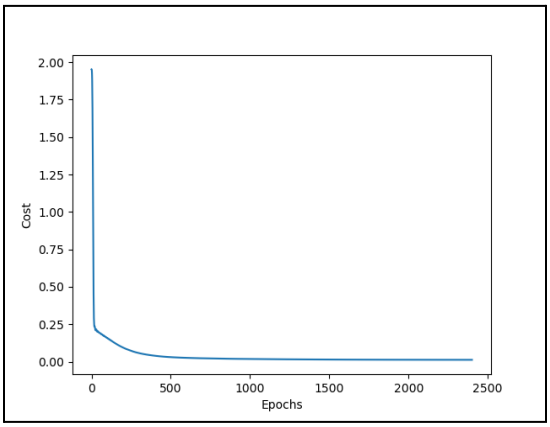


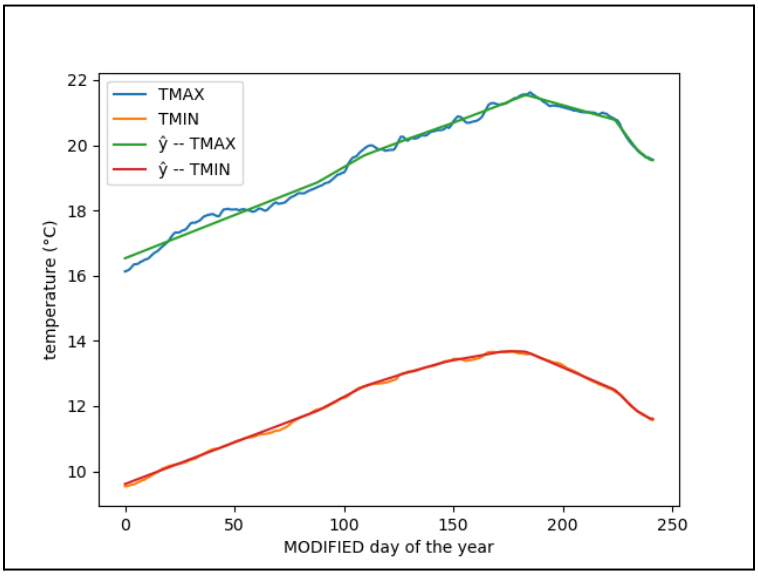


The model's prediction for TMIN is good enough for production purposes now and we can clearly see now that the model is making some effort to improve its predictions for TMAX. Let's increase the number of neurons (hidden units) to 256 to see more improvement in the model's predictions for TMAX. Here is the code:

```
model = Neural_Network(1, 256, 2)    # 1 input feature, 256 hidden units, 2 output features
optimizer = RMSProp_Optimizer(learning_rate=0.001)
# every 200 epochs, denominator increases by 1
decayer = Learning_Rate_Decayer(optimizer, 0.005)
```

Again, to do this we change the second argument in initializing the model to 256. With more hidden units, comes with more neurons that have more parameters that need to be updated. As a result, we will use 2400 epochs, printing every 240 epochs. Here are the graphs:





The execution time for this script should be less than a minute and the cost is now about 0.012, but we see that the model has made significant improvement in the prediction of the last few days of TMAX. Since there has not been much change in the cost after doubling the number of hidden layers, it's time to increase the number of hidden layers again. Here is the code:

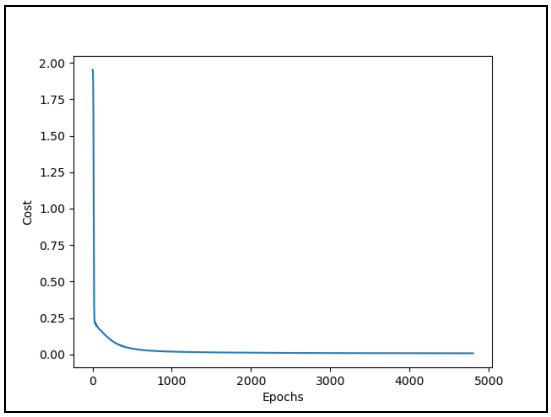
```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation2 = ReLU_Activation()
        self.hidden_layer3 = Dense_Layer(n_hidden, n_hidden)
        self.activation3 = ReLU_Activation()
        self.hidden_layer4 = Dense_Layer(n_hidden, n_hidden)
        self.activation4 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.cost_function = MSE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2,
                                 self.hidden_layer3, self.hidden_layer4,
                                 self.output_layer]
```

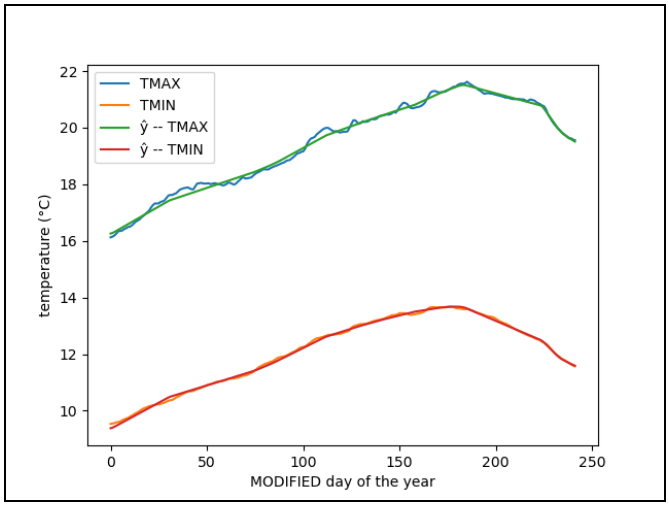
```
def forward(self, inputs, y_true):
    self.hidden_layer1.forward(inputs)
    self.activation1.forward(self.hidden_layer1.outputs)
    self.hidden_layer2.forward(self.activation1.outputs)
    self.activation2.forward(self.hidden_layer2.outputs)
    self.hidden_layer3.forward(self.activation2.outputs)
    self.activation3.forward(self.hidden_layer3.outputs)
    self.hidden_layer4.forward(self.activation3.outputs)
    self.activation4.forward(self.hidden_layer4.outputs)
    self.output_layer.forward(self.activation4.outputs)
    self.cost = self.cost_function.forward(self.output_layer.outputs, y_true)

def backward(self, y_true):
    self.cost_function.backward(self.output_layer.outputs, y_true)
    self.output_layer.backward(self.cost_function.dinputs)
    self.activation4.backward(self.output_layer.dinputs)
    self.hidden_layer4.backward(self.activation4.dinputs)
    self.activation3.backward(self.hidden_layer4.dinputs)
    self.hidden_layer3.backward(self.activation3.dinputs)
    self.activation2.backward(self.hidden_layer3.dinputs)
    self.hidden_layer2.backward(self.activation2.dinputs)
    self.activation1.backward(self.hidden_layer2.dinputs)
    self.hidden_layer1.backward(self.activation1.dinputs)
```

We are going to keep the number of hidden units the same, since 256 is already a huge amount, decrease the learning rate to 0.0006 (equivalent to 6e-4) but we still decay the learning rate by the same amount with a decay rate of 0.05. With more layers, we have more neurons and consequently more parameters need to be updated, so give the model 4800 epochs, printing every 480 epochs. Include the percentage of dead neurons in the 4^{th} hidden layer in the print statement.

```
# check for dead neurons
if epochs % 480 == 0:
    # cost & percentage of dead neurons
    print(f"Cost: {model.cost} - Dead1: {get_dead(model.hidden_layer1)}% - " + \
        f"Dead2: {get_dead(model.hidden_layer2)}% - " + \
        f"Dead3: {get_dead(model.hidden_layer3)}% - " + \
        f"Dead4: {get_dead(model.hidden_layer4)}%")
```

The execution time for this script should be less than a minute and the cost is now about 0.008, making us pass another milestone, cost of under 0.001! Here are the graphs:





Loads of improvement on the model's prediction for TMAX!!! This model, based on its cost and predictions compared to the correct answers, is smart, and has **converged** (no more training necessary)! This is an example of a model that can be deployed into production as a product or whatever it is needed for.

Congratulations on finally training a regression model, more specifically a neural network containing multiple hidden layers, based on a real dataset!!!!

Before moving on to the next big thing, classification in AI, let's synthesize what adding more hidden neurons (hidden units) and increasing the number of hidden layers mean.

When building a neural network model, think of buying ice-cream that you can decide to customize yourself. You decide how many scoops (amount) you would like and decide the flavors (deliciousness of the ice-cream) you would like to have on your ice-cream.

Adding more scoops to the ice-cream means that the ice-cream man would take more time to prepare as each addition of a scoop of ice-cream requires time to add. Even though you may have to wait longer, you would be more happy or satisfied since you get to eat more ice-cream. Adding more scoops is equivalent to adding more hidden layers to your model, your model gets to learn more complex functions to fit the data, but we have to increase the number of training epochs since each new neuron requires time for each of the parameters to update to its optimal value.

Adding more flavors to the ice-cream, means that the ice-cream man has to create better flavors that go in your ice-cream, to make it more delicious. Making new flavors takes a lot of time and there will be a point where it gets harder and harder to make even better flavors, so slowly the quality of adding more flavors stops improving a lot. Adding more flavors is equivalent to adding more hidden neurons/units to each hidden layer since initially an increase of hidden units makes the model smarter but later an increase of hidden units barely impacts the performance of the model. Also, each time the number of hidden units is increased, the model takes more training time, which represents the increase in the time the ice-cream man needs to take to finish preparing your ice-cream.

Congratulations again, and see you in the last few units that focus on the classification problem, which has fewer differences than the regression problem!