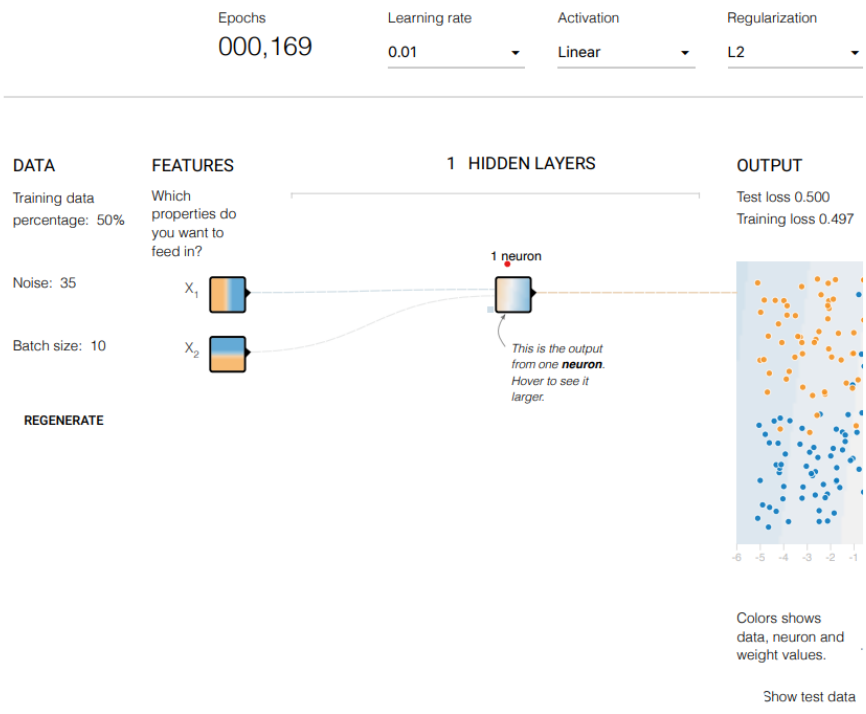
FML-CIE2-MOD-5

PART-A

1. Explain whether the model learn any Non-Linearity. A Neural Net model is combining two input features into a single neuron.

ANSWER:



The Activation is set to Linear, so this model cannot learn any nonlinearities. The loss is very

Source on GitHub (https://github.com/tensorflow/playground)

Epochs

000,169

Learning rate Activation Regularization

(https://www.tensorflow.org/)

eplay lay_arrow kip_next 0.01 0 Linear 0 L2 0

Training data

percentage: 50%

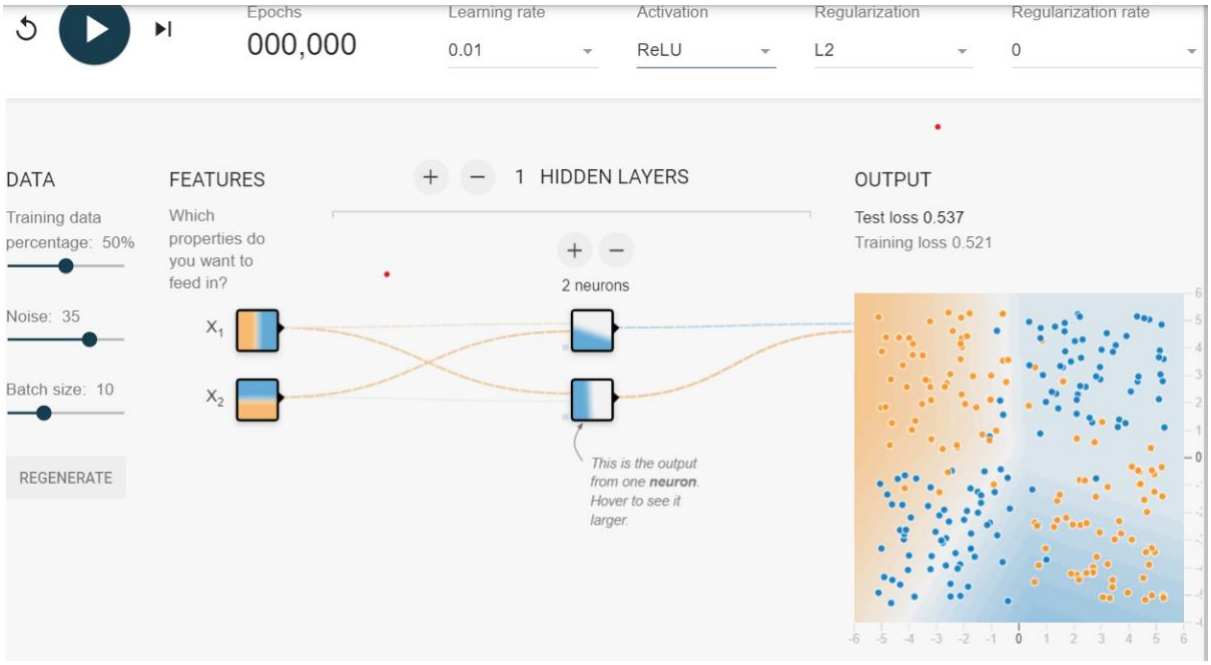
Noise: 35

Batch size: 10

REGENERATE
Colors shows
data, neuron and
weight values. -1
DATA
Which
properties do
you want to
feed in?
FEATURES ddemove 1 HIDDEN LAYERS OUTPUT
Test loss 0.500
Training loss 0.497
-6 -5 -4 -3 -2 -1
Show test data
X1
X2 This is the output
from one neuron.
Hover to see it
larger.
1 neuron
ddemove
4
high, and we say the model underfits the data

2. Illustrate,in a Neural Net model the number of neurons are increased in the hidden layer from 1-2 and include the activation function like ReLU. Verify whether this model can support non-linearity and Data Modeling effectively

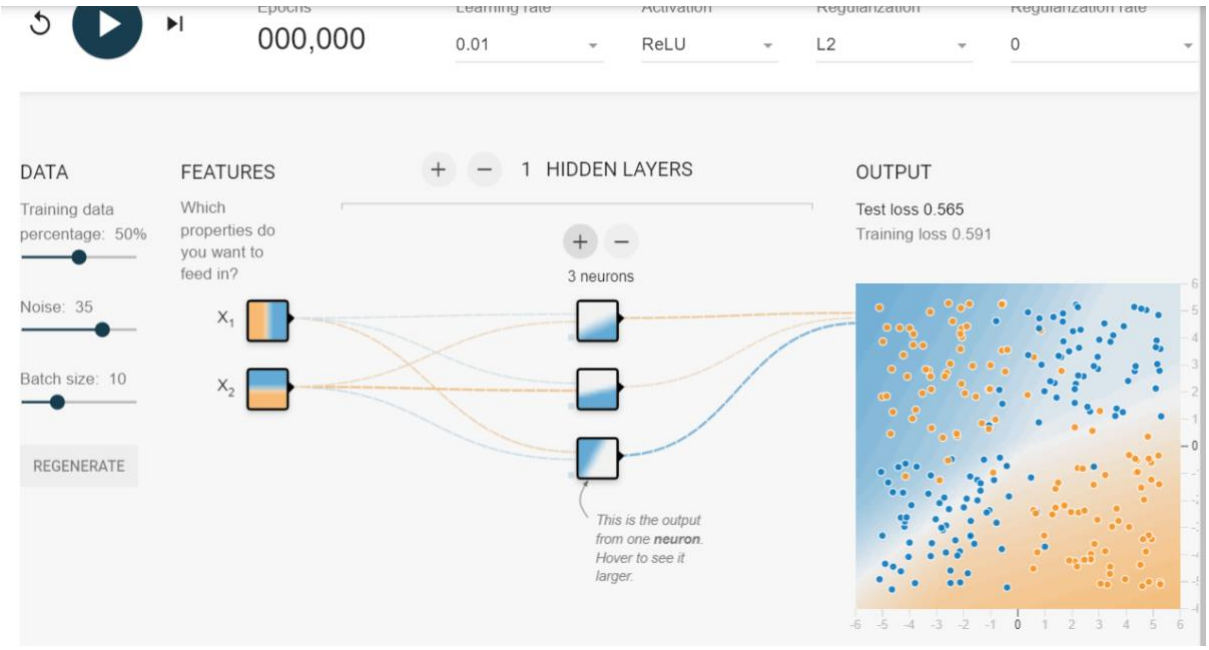
ANSWER:



The nonlinear activation function can learn nonlinear models. However, a single hidden layer with 2 neurons cannot reflect all the nonlinearities in this data set, and will have high loss even without noise: it still underfits the data. These exercises are nondeterministic, so some runs will not learn an effective model, while other runs will do a pretty good job. The best model may not have the shape you expect!

3. Illustrate,in the Neural Network defined above by increasing neurons in the hidden layers from 2-3 and verify the quality, Data Modeling Effectiveness

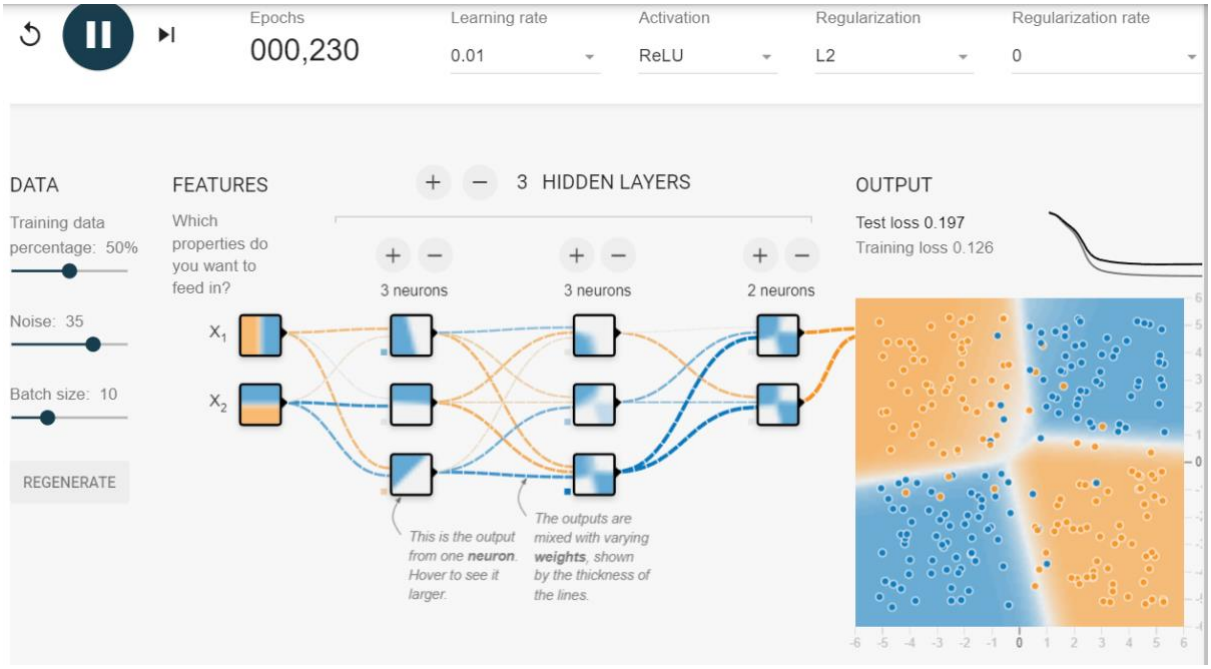
ANSWER:



Playground's nondeterministic nature shines through on this exercise. A single hidden layer with 3 neurons is enough to model the data set (absent noise), but not all runs will converge to a good model. 3 neurons are enough because the XOR function can be expressed as a combination of 3 halfplanes (ReLU activation). You can see this from looking at the neuron images, which show the output of the individual neurons. In a good model with 3 neurons and ReLU activation, there will be 1 image with an almost vertical line, detecting X1 being positive (or negative; the sign may be switched), 1 image with an almost horizontal line, detecting the sign of X2, and 1 image with a diagonal line, detecting their interaction. However, not all runs will converge to a good model. Some runs will do no better than a model with 2 neurons, and you can see duplicate neurons in these cases

4. What is the smallest number of neurons and layers you can use that gives test loss of 0.177 or lower?Continue experimenting by adding or removing hidden layers and neurons per layer. Also feel free to change learning rates, regularization, and other learning settings.

ANSWER:



- First hidden layer with 3 neurons.
- Second hidden layer with 3 neurons.
- Third hidden layer with 2 neurons.

A single hidden layer with 3 neurons can model the data, but there is no redundancy, so on many runs it will effectively lose a neuron and not learn a good model. A single layer with more than 3 neurons has more redundancy, and thus is more likely to converge to a good model. As we saw, a single hidden layer with only 2 neurons cannot model the data well. If you try it, you can see then that all of the items in the output layer can only be shapes composed of the lines from those two nodes. In this case, a deeper network can model the data set better than the 2rst hidden layer alone: individual neurons in the second layer can model more complex shapes, like the upper-right quadrant, by combining neurons in the 2rst layer. While adding that second hidden layer can still model the data set better than the 2rst hidden layer alone, it might make more sense to add more nodes to the 2rst layer to let more lines be part of the kit from which the second layer builds its shapes

However, a model with 1 neuron in the 2rst hidden layer cannot learn a good model no matter how deep it is. This is because the output of the 2rst layer only varies along one dimension (usually a diagonal line), which isn't enough to model this data set well. Later layers can't compensate for this, no matter how complex; information in the input data has been irrecoverably lost. What if instead of trying to have a small network, we had lots of layers with lots of neurons, for a simple problem like this? Well, as we've seen, the 2rst layer will have the ability to try lots of different line slopes. And the second layer will have the ability to accumulate them into lots of different shapes, with lots and lots of shapes on down through the subsequent layers. By allowing the model to consider so many different shapes through so many different hidden neurons, you've created enough space for the model to start easily over0tting on the

noise in the training set, allowing these complex shapes to match the foibles of the training data rather than the generalized ground truth. In this example, larger models can have complicated boundaries to match the precise data points. In extreme cases, a large model could learn an island around an individual point of noise, which is called memorizing the data. By allowing the model to be so much larger, you'll see that it actually often performs worse than the simpler model with just enough neurons to solve the problem.

5. How you can use the ReLU activation function to separate the two classes. Consider the case of the XOR function in which the two points (0, 0),(1, 1) belong to one class, and the other two points (1, 0),(0, 1) belong to the other class. Show ANSWER:

Here's one way of doing this, using a hidden layer with two nodes. In Python:

```
1. def relu(x):
2.     return max(0, x)
3.
4. def x(u, v):
5.     return relu(u - v)
6.
7. def y(u, v):
8.     return relu(-u + v)
9.
10. def f(u, v):
11.     return relu(x(u, v) + y(u, v))
```

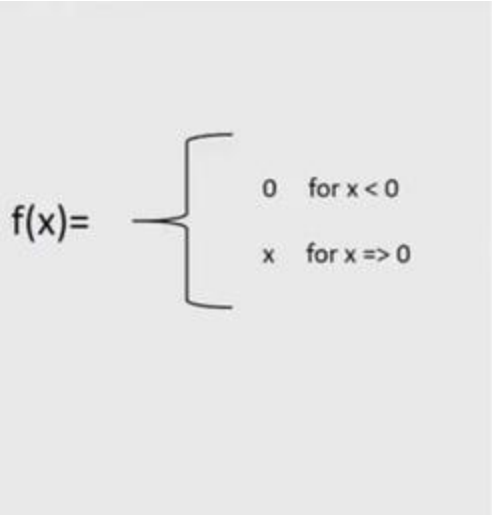
Here *u* and *v* are your input variables. The hidden nodes are *x* and *y*, and the output node is *f*. In actual practice one would implement this differently, but I've written this in terms of functions in the hope that this will make the structure clearer. Step through this as follows and convince yourself that it does what it needs to do:

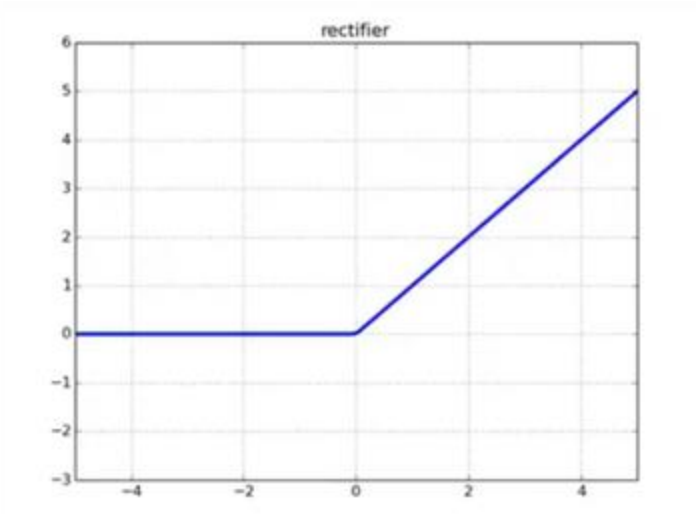
```
1. for u, v in [(0,0), (0,1), (1,0), (1,1)]:
2.     print(u, v, ' ', x(u,v), y(u,v), ' ', f(u, v))
```

ReLU or Rectified Linear Unit ia node or unit that implements the Rectified Linear activation function. Often, networks that use the rectifier function for the hidden layers are referred to as rectified networks. The rectified linear activation function is a simple

calculation that returns the value provided as input directly, or the value 0.0 if the input is 0.0 or less.

To summarise,



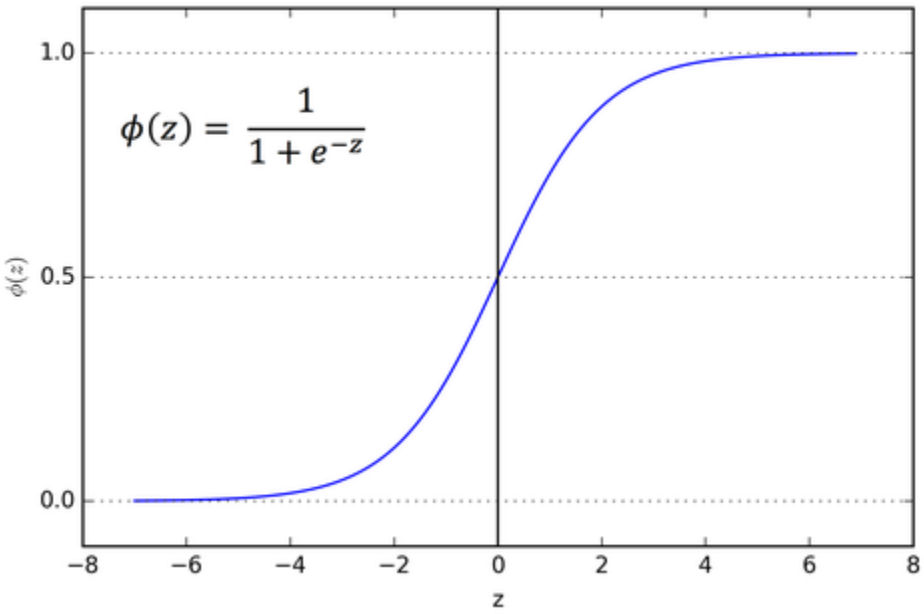


In machine learning, we pass the output of every layer in the model through a non linear "activation" function, before we pass it on to the next layer. The reason we do that is a result of:

1. Each layer in the model is a linear function by itself (multiplying the output of the previous layer with the weights of the current layer).
2. If f(x) and g(x) are both linear, f(g(x)) is also linear.

So the combination of (1) and (2) means that even if you create 1000000 layers that feed into each other - their combined composite representation power is that of a single linear function applied on the input values. So we introduce non linearity between the layers, in order to allow the model to represent complex non linear functions of the input values.

Historically, the logistic or sigmoid function was commonly used to introduce this non linearity:

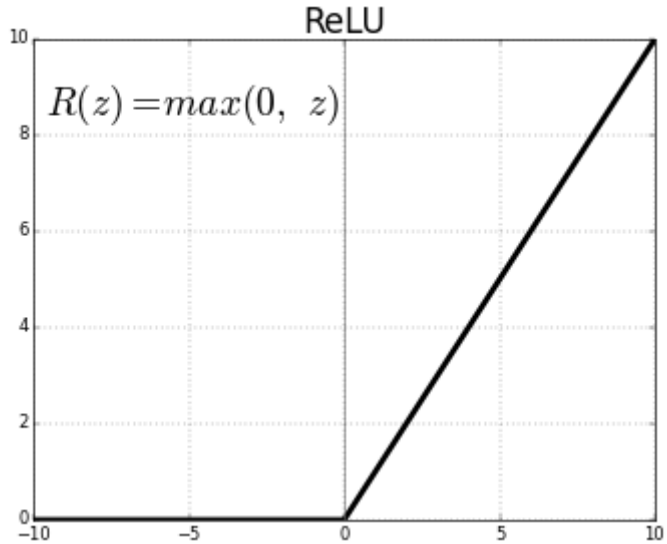


Unfortunately, as you can see, this function flattens out pretty quickly on both the positive and negative sides of the horizontal axis. So when the input is a fairly large positive number or a fairly large negative - the function returns pretty much the same value (1 and 0 respectively). This can slow learning significantly, or even never converge.

That's why in recent years, the ReLU function was introduced, and quickly became the most frequently used non linear activation function for ML.

What is it:

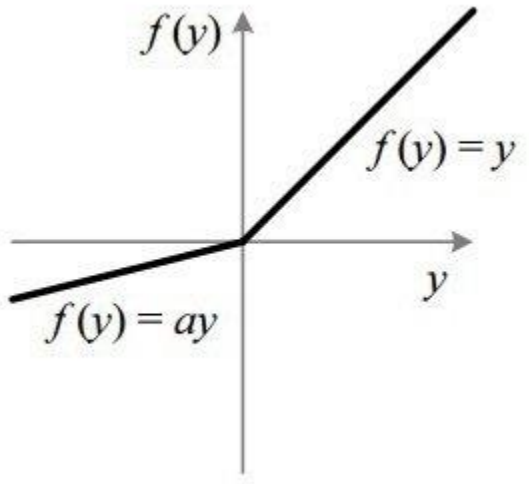
ReLU - the Rectified Linear Unit activation function is a very simple function that outputs 0 for any z<0, and is a simple linear function with slope of 1 ,R(z)=z for any z>=0:



It still has the nice properties of a differential monotonic function, but at least for positive weight input values, it escapes the problem the sigmoid function has.

Unfortunately, for negative input values - it still has the same issue, as all values are mapped to 0.

In order to address that, another version of ReLU was introduced - "Leaky ReLU". In this version, the flat slope for values <0 is replaced by a small positive slope:



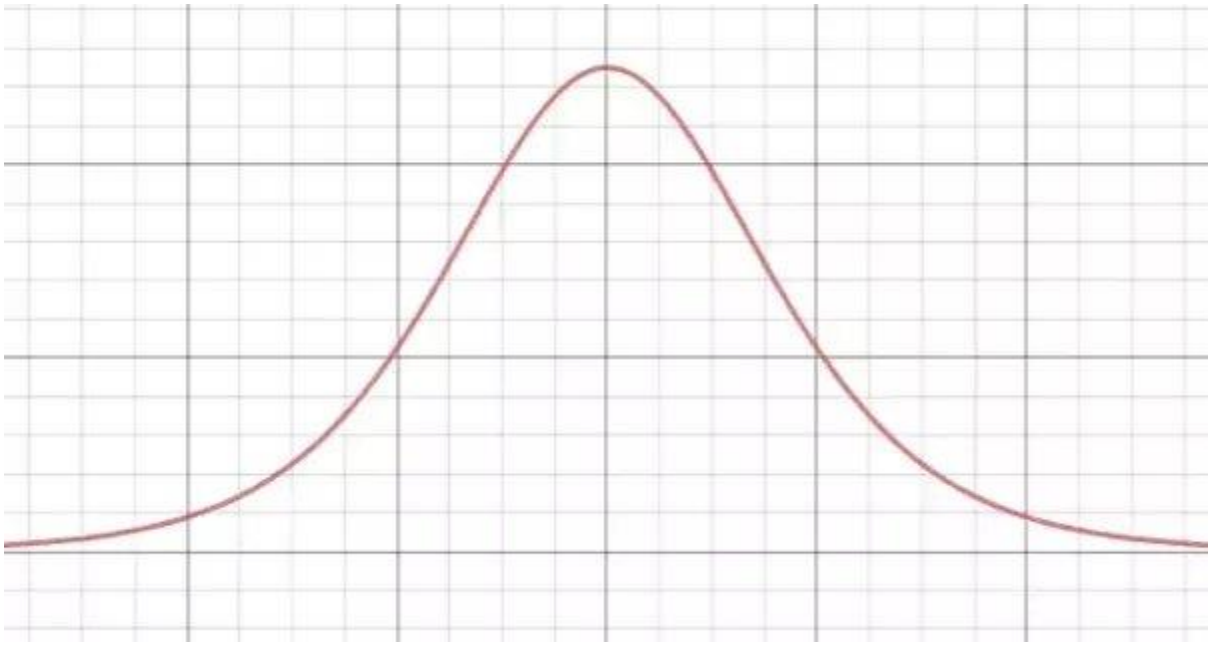
So now you know why we need a non linear activation function between every layer in a model, and why ReLU is considered a good candidate for such a function

6. Show that the derivative of the sigmoid activation function is at most 0.25, irrespective of the value of its argument. At what value of its argument does the sigmoid activation function take on its maximum value?

ANSWER:

Well a good start is to differentiate it!

$f(x)=\frac{1}{1+e^{-x}}$ f(x)=11+e−x
Let $u=1+e^{-x}$ u=1+e−x, so $f(x)=\frac{1}{u}$ f(x)=1u
Chain rule: $\frac{dy}{dx}=\frac{dy}{du}\cdot\frac{du}{dx}$ dydx=dydu·dudx
$y=u^{-1}$ y=u−1, so $\frac{dy}{du}=-1u^2$ dydu=−1u2
$u=1+e^{-x}$ u=1+e−x, so $\frac{du}{dx}=-e^{-x}$ dudx=−e−x
Combine: $\frac{dy}{dx}=\frac{1}{u^2}\cdot e^{-x}$ dydx=1u2·e−x
Revert uu terms and simplify: $\frac{dy}{dx}=\frac{e^{-x}}{(1+e^{-x})^2}$ dydx=e−x(1+e−x)2
Now we can look for the range. Consider the graph:



As x tends to $\pm\infty$, y approaches 0. You can see from the equation that to make $x=0$, the denominator must tend to ∞. Since the denominator is a squaring function, the contents of the brackets can be negative or positive, and the squaring will force it to become positive. Therefore, we can either make the $1+e^{-x}$ 'very negative' or 'very positive'. Therefore, as x, tends to $\pm\infty$, the entire denominator tends to ∞, so the value of y will tend to 0.
To show the function never exceeds 0.25, we can do 2 things. We can cheat by looking at the graph and seeing that the peak is where $x=0$, and just plug that into the equation, or we can mathematically show that this peak is where $x=0$. To find it, we need to find the coordinates of where the gradient of our new graph is 0. Finding this second derivative is messy and long, so I'll state it as:
$\frac{d^2y}{dx^2}=\frac{e^{-2x}(1-e^x)}{(1+e^{-x})^3}$
Finding where this equals zero simply means setting the numerator equal to zero:

$e^{-2x}(1-e^x)=0$
And solving for x:
$e^{-2x}=0$ has no real solutions.
$1-e^x=0$, $e^x=1$, $x=\ln(1)$, $x=0$.
So, our peak for $\frac{dy}{dx}$ is where $x=0$, as we earlier observed from the graph.
Plugging this in:

$\frac{e^{-x}}{(1+e^{-x})^2}=\frac{e^{-0}}{(1+e^{-0})^2}=\frac{1}{(1+1)^2}=\frac{1}{2^2}=\frac{1}{4}=0.25$.
Proof that the derivative of the sigmoid function ranges between 0 and 0.25, QED.

OR

Presumably you have in mind the particular sigmoid function more specifically called the "logistic function", $f(x)=\frac{1}{1+e^{-x}}$f(x)=11+e−x (note: there are other functions often called "sigmoid functions" as well, whose derivatives are not constrained to the range you ask about).
For the logistic function, you can verify by straightforward algebra and calculus that $f(x)+f(-x)=1$f(x)+f(−x)=1 and $f'(x)=f(x)f(-x)$f′(x)=f(x)f(−x).
But two values which sum to $1$1 can have a product of at most $0.25$0.25 (as the two values must be of the form $0.5+\delta$0.5+δ and $0.5-\delta$0.5−δ, so that their product is $0.25-\delta^2$0.25−δ2).
As for why the derivative is never negative, this would only occur if $f(x)$f(x) and $f(-x)$f(−x) ever had opposite signs, but both are always positive.

7. Show that the derivative of the tanh activation function is at most 1, irrespective of the value of its argument. At what value of its argument does the tanh activation take on its maximum value?

ANSWER:

The Hyperbolic Tangent Activation Function

Though the logistic sigmoid has a nice biological interpretation, it turns out that the logistic sigmoid can cause a neural network to get "stuck" during training. This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero. Since neural networks use the feed-forward activations to calculate parameter gradients (again, see this this post for details), this can result in model parameters that are updated less regularly than we would like, and are thus "stuck" in their current state.

An alternative to the logistic sigmoid is the hyperbolic tangent, or $\tanh$tanh function (***Figure 1***, green curves):

$$
\begin{aligned}
g_{\tanh}(z) &= \frac{\sinh(z)}{\cosh(z)} \\
&= \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}
\end{aligned}
$$

Like the logistic sigmoid, the tanh function is also sigmoidal ("s"-shaped), but instead outputs values that range $(-1,1)$. Thus strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get "stuck" during training. Calculating the gradient for the tanh function also uses the quotient rule:

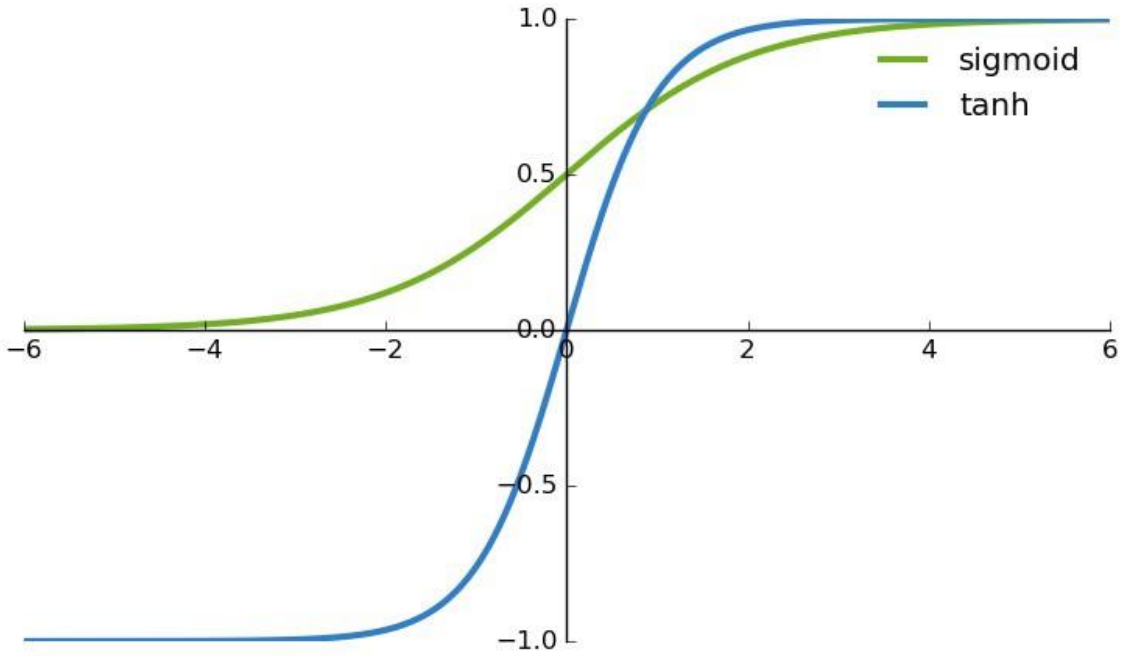
$$
\begin{aligned}
g'_{\tanh}(z) &= \frac{\partial}{\partial z}\frac{\sinh(z)}{\cosh(z)} \\
&= \frac{\frac{\partial}{\partial z}\sinh(z)\times\cosh(z)-\frac{\partial}{\partial z}\cosh(z)\times\sinh(z)}{\cosh^{2}(z)} \\
&= \frac{\cosh^{2}(z)-\sinh^{2}(z)}{\cosh^{2}(z)} \\
&= 1-\frac{\sinh^{2}(z)}{\cosh^{2}(z)} \\
&= 1-\tanh^{2}(z)
\end{aligned}
$$

Similar to the derivative for the logistic sigmoid, the derivative of $g_{\tanh}(z)$ is a function of feed-forward activation evaluated at z, namely $(1-g_{\tanh}(z)^{2})$. Thus the same caching trick can be used for layers that implement tanh activation functions.

The tanh function is just another possible function that can be used as a non-linear activation function between layers of a neural network. It shares a few things in common with the sigmoid activation function. Unlike a sigmoid function that will map input values between 0 and 1, the Tanh will map values between -1 and 1. Similar to the sigmoid function, one of the interesting properties of the tanh function is that the derivative of tanh can be expressed in terms of the function itself.

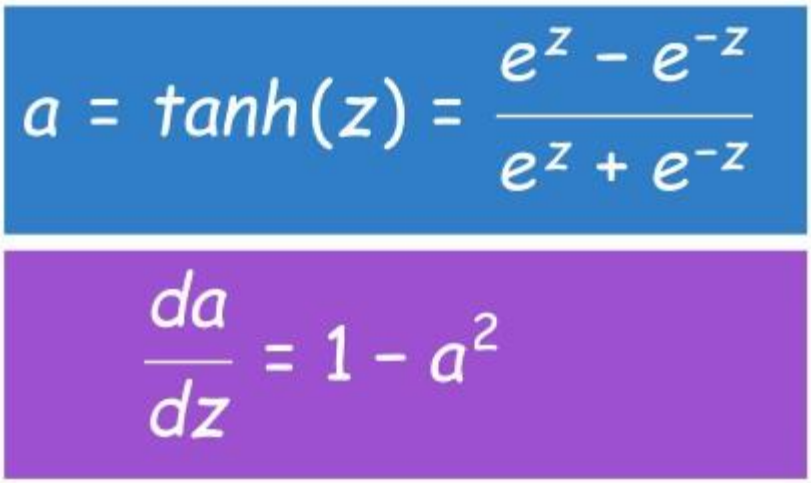
The tanh function is just another possible functions that can be used as a nonlinear activation function between layers of a neural network. It actually shares a few things in common with the sigmoid activation function. They both look very similar. But while a sigmoid

function will map input values to be between 0 and 1, Tanh will map values to be between -1 and 1.



You will also notice that the tanh is a lot steeper.

Like the sigmoid function, one of the interesting properties of the tanh function is that the derivative can be expressed in terms of the function itself. Below is the actual formula for the tanh function along with the formula for calculating its derivative.



Derivative of tanh(z):

a=(e^z-e^(-z))/(e^z+e^(-z)

use same u/v rule

da=[(e^z+e^(-z))*d(e^z-e^(-z))]-[(e^z-e^(-z))*d((e^z+e^(-z))]/[(e^z+e^(-z)]²

da=[(e^z+e^(-z))*(e^z+e^(-z))]-[(e^z-e^(-z))*(e^z-e^(-z))]/[(e^z+e^(-z)]²

da=[(e^z+e^(-z)]²-[(e^z-e^(-z)]²/[(e^z+e^(-z)]²

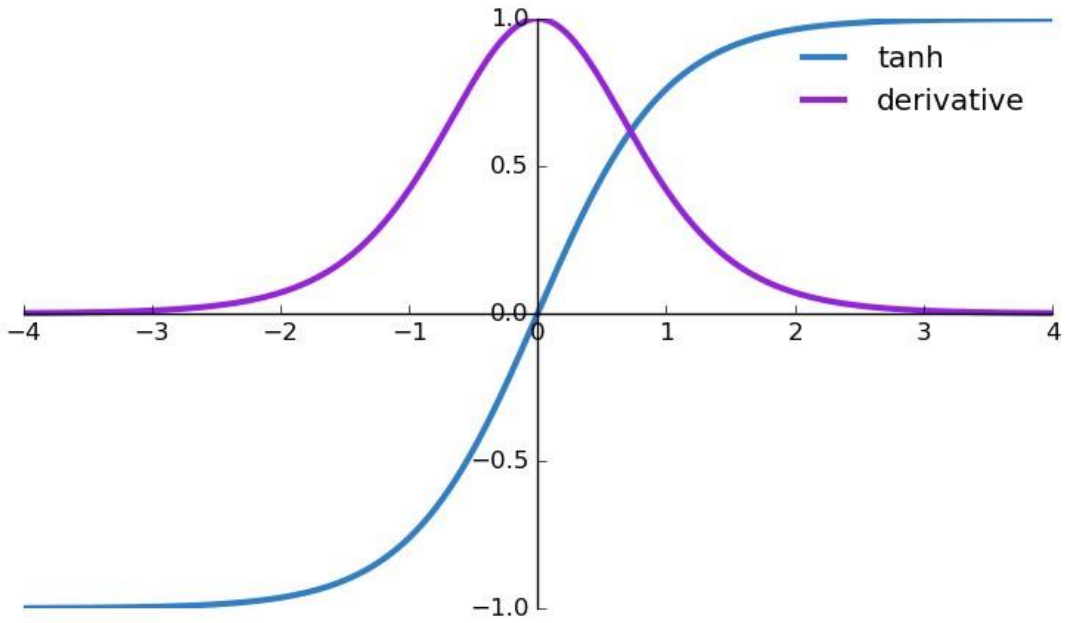
da=1-[(e^z-e^(-z))/(e^z+e^(-z)]²

da=1-a²

Python code:

```
import matplotlib.pyplot as plt
import numpy as np

def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    dt=1-t**2
    return t,dtz=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')# Create and show plot
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3,
label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3,
label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```



Observations:

(i)Now it's output is zero centered because its range in between -1 to 1 i.e -1 < output < 1 .

(ii) Hence optimization is *easier* in this method hence in practice it is always preferred over Sigmoid function .

But still it suffers from Vanishing gradient problem.

When will use:

Usually used in hidden layers of a neural network as it's values lies between**-1 to 1** hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.

8. Choose a network with two inputs x1 and x2. It has two hidden layers, each of which contain two units. Assume that the weights in each layer are set so that top unit in each layer applies sigmoid activation to the sum of its inputs and the bottom unit in each layer applies tanh activation to the sum of its inputs. Finally, the single output node applies ReLU activation to the sum of its two inputs. Write the output of this neural network in closed form as a function of x1 and x2. This exercise should give you an idea of the complexity of functions computed by neural networks.

ANSWER: