Vanishing Gradients : Ungraded Lecture Notebook

In this notebook you'll take another look at vanishing gradients, from an intuitive standpoint.

Background

Adding layers to a neural network introduces multiplicative effects in both forward and backward propagation. The back prop in particular presents a problem as the gradient of activation functions can be very small. Multiplied together across many layers, their product can be vanishingly small! This results in weights not being updated in the front layers and training not progressing.

Gradients of the sigmoid function, for example, are in the range 0 to 0.25. To calculate gradients for the front layers of a neural network the chain rule is used. This means that these tiny values are multiplied starting at the last layer, working backwards to the first layer, with the gradients shrinking exponentially at each step.

Imports

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

Data, Activation & Gradient

Data

I'll start be creating some data, nothing special going on here. Just some values spread across the interval -5 to 5.

- Try changing the range of values in the data to see how it impacts the plots that follow.

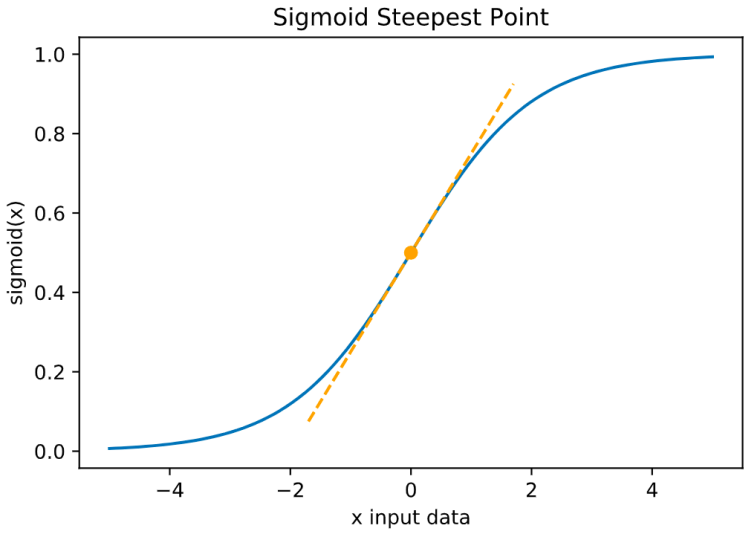
Activation

The example here is sigmoid() to squish the data x into the interval 0 to 1.

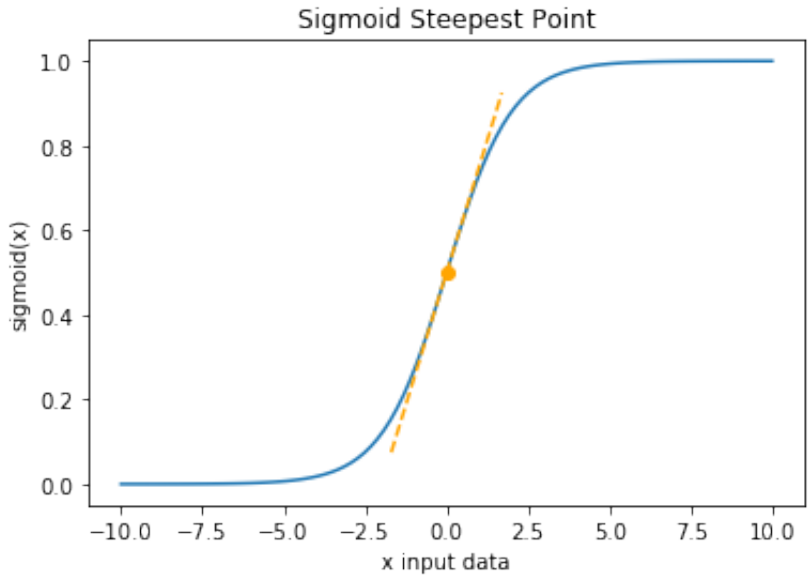
Gradient

This is the derivative of the sigmoid() activation function. It has a maximum of 0.25 at x = 0, the steepest point on the sigmoid plot.

- Try changing the x value for finding the tangent line in the plot.



In [14]:
```
# Data
# Interval [-5, 5]
### START CODE HERE ###
x = np.linspace(-10, 10, 100)  # try changing the range of values i
n the data. eg: (-100,100,1000)
### END CODE HERE ###
# Activation
# Interval [0, 1]
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

activations = sigmoid(x)

# Gradient
# Interval [0, 0.25]
def sigmoid_gradient(x):
    return (x) * (1 - x)

gradients = sigmoid_gradient(activations)

# Plot sigmoid with tangent line
plt.plot(x, activations)
plt.title("Sigmoid Steepest Point")
plt.xlabel("x input data")
plt.ylabel("sigmoid(x)")

# Add the tangent line
### START CODE HERE ###
x_tan = 0   # x value to find the tangent. try different values wit
hin x declared above. eg: 2
### END CODE HERE ###
y_tan = sigmoid(x_tan)  # y value
span = 1.7              # line span along x axis
data_tan = np.linspace(x_tan - span, x_tan + span)  # x values to p
lot
gradient_tan = sigmoid_gradient(sigmoid(x_tan))     # gradient of t
he tangent
tan = y_tan + gradient_tan * (data_tan - x_tan)     # y values to p
lot
plt.plot(x_tan, y_tan, marker="o", color="orange", label=True)  # m
arker
plt.plot(data_tan, tan, linestyle="--", color="orange")         # l
ine
plt.show()
```



Plots

Sub Plots

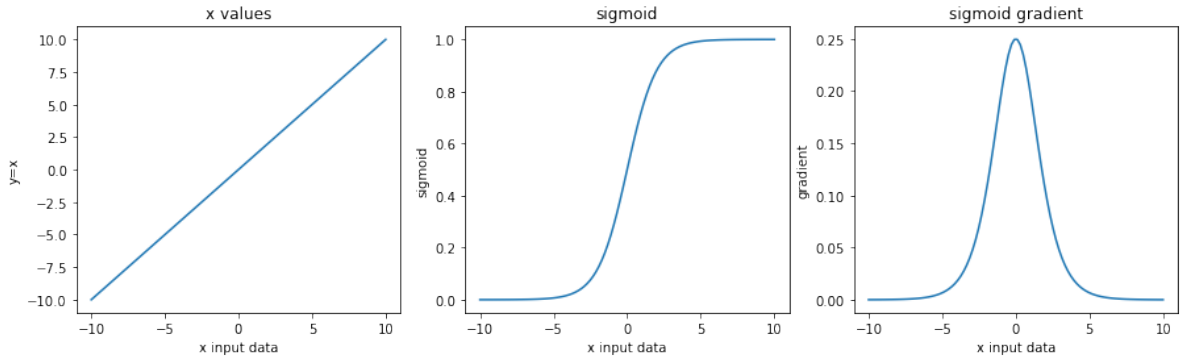
Data values along the x-axis of the plots on the interval chosen for x, -5 to 5. Subplots:

- x vs x
- sigmoid of x
- gradient of sigmoid

Notice how the y axis keeps compressing from the left plot to the right plot. The interval range has shrunk from 10 to 1 to 0.25. How did this happen? As |x| gets larger the sigmoid approaches asymptotes at 0 and 1, and the sigmoid gradient shrinks towards 0.

- Try changing the range of values in the code block above to see how it impacts the plots.

In [15]:
```
# Sub plots
fig, axs = plt.subplots(1, 3, figsize=(15, 4), sharex=True)

# X values
axs[0].plot(x, x)
axs[0].set_title("x values")
axs[0].set_ylabel("y=x")
axs[0].set_xlabel("x input data")

# Sigmoid
axs[1].plot(x, activations)
axs[1].set_title("sigmoid")
axs[1].set_ylabel("sigmoid")
axs[1].set_xlabel("x input data")

# Sigmoid gradient
axs[2].plot(x, gradients)
axs[2].set_title("sigmoid gradient")
axs[2].set_ylabel("gradient")
axs[2].set_xlabel("x input data")

fig.show()
```

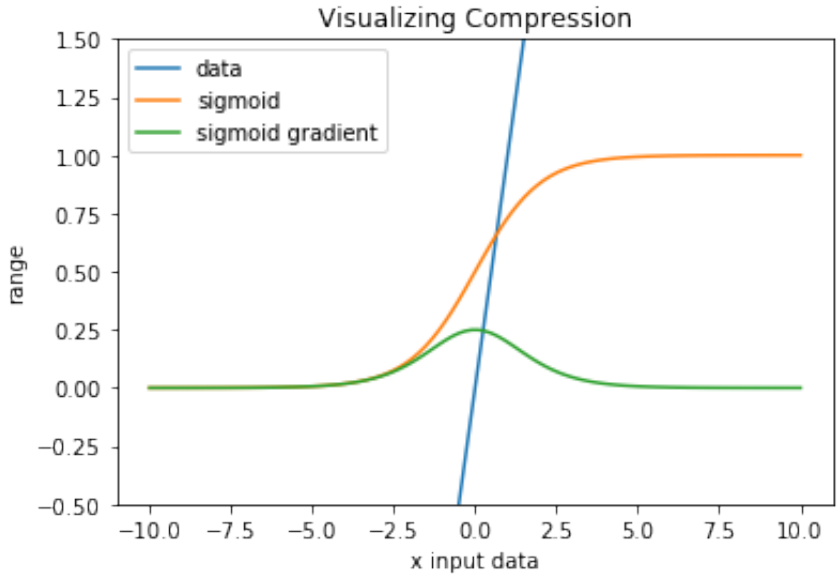


Single Plot

Putting all 3 series on a single plot can help visualize the compression. Notice how hard it is to interpret because sigmoid and sigmoid gradient are so small compared to the scale of the input data x.

- Trying changing the plot ylim to zoom in.

In [16]:
```
# Single plot
plt.plot(x, x, label="data")
plt.plot(x, activations, label="sigmoid")
plt.plot(x, gradients, label="sigmoid gradient")
plt.legend(loc="upper left")
plt.title("Visualizing Compression")
plt.xlabel("x input data")
plt.ylabel("range")
### START CODE HERE ###
plt.ylim(-.5, 1.5)    # try shrinking the y axis limit for better v
isualization. eg: uncomment this line
### END CODE HERE ###
plt.show()

# Max, Min of each array
print("")
print("Max of x data :", np.max(x))
print("Min of x data :", np.min(x), "\n")
print("Max of sigmoid :", "{:.3f}".format(np.max(activations)))
print("Min of sigmoid :", "{:.3f}".format(np.min(activations)), "\n
")
print("Max of gradients :", "{:.3f}".format(np.max(gradients)))
print("Min of gradients :", "{:.3f}".format(np.min(gradients)))
```




```
Max of x data : 10.0
Min of x data : -10.0

Max of sigmoid : 1.000
Min of sigmoid : 0.000

Max of gradients : 0.249
Min of gradients : 0.000
```

Numerical Impact

Multiplication & Decay

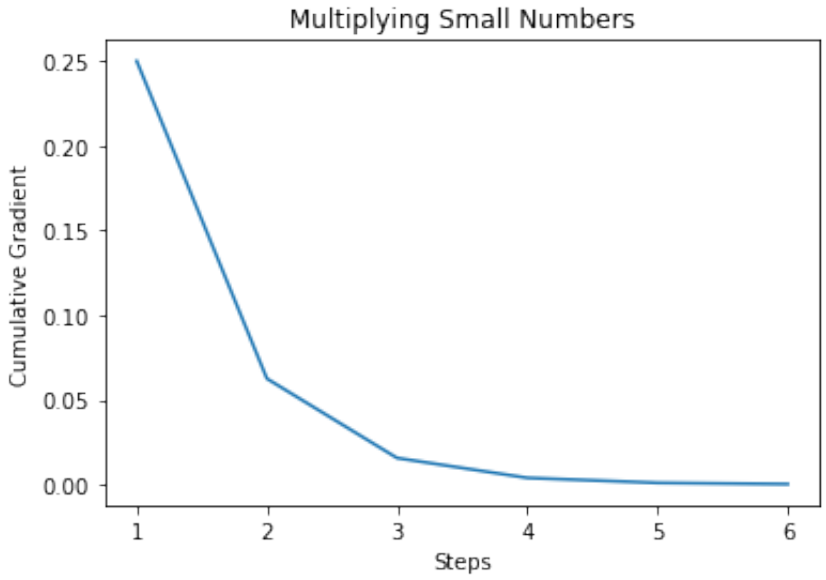
Multiplying numbers smaller than 1 results in smaller and smaller numbers. Below is an example that finds the gradient for an input x = 0 and multiplies it over n steps. Look how quickly it 'Vanishes' to almost zero. Yet sigmoid(x=0)=0.5 which has a sigmoid gradient of 0.25 and that happens to be the largest sigmoid gradient possible!

(Note: This is NOT an implementation of back propagation.)

- Try changing the number of steps n.
- Try changing the input value x. Consider the impact on sigmoid and sigmoid gradient.

In [17]:
```
# Simulate decay
# Inputs
### START CODE HERE ###
n = 6  # number of steps : try changing this
x = 0  # value for input x : try changing this
### END CODE HERE ###
grad = sigmoid_gradient(sigmoid(x))
steps = np.arange(1, n + 1)
print("-- Inputs --")
print("steps :", n)
print("x value :", x)
print("sigmoid :", "{:.5f}".format(sigmoid(x)))
print("gradient :", "{:.5f}".format(grad), "\n")

# Loop to calculate cumulative total
print("-- Loop --")
vals = []
total_grad = 1  # initialize to 1 to satisfy first loop below
for s in steps:
    total_grad = total_grad * grad
    vals.append(total_grad)
    print("step", s, ":", total_grad)

print("")

# Plot
plt.plot(steps, vals)
plt.xticks(steps)
plt.title("Multiplying Small Numbers")
plt.xlabel("Steps")
plt.ylabel("Cumulative Gradient")
plt.show()
```

```
-- Inputs --
steps : 6
x value : 0
sigmoid : 0.50000
gradient : 0.25000

-- Loop --
step 1 : 0.25
step 2 : 0.0625
step 3 : 0.015625
step 4 : 0.00390625
step 5 : 0.0009765625
step 6 : 0.000244140625
```



Solution

One solution is to use activation functions that don't have tiny gradients. Other solutions involve more sophisticated model design. But they're both discussions for another time.