Applications of Gradient Descent In Machine Learning

by Timothy Gao

Table Of Contents:

Section 1: Logistic Regression

1.1: Introducing Weights

We wish to predict a probability that a cell is cancerous based on a set of input variables, called features (e.g., cell radius, texture, concavity). In machine learning, this is called a binary classification problem: we wish to predict the probability for a binary class (yes or no cancer, 1 or 0).

Let each feature be an element of the vector $x=<x_1,x_2,\dots,x_n>$

Ultimately, we desired a scalar value. How can we convert this n-dimensional vector of features into a single scalar? Dot Product!

Let weights be the vector $w=<w_1,w_2,\dots w_n>$. Then $s=w\cdot x=\Sigma_{i=1}^{n}w_ix_i$. Visually:

Utilizing a visually example, we have:

Cell Radius	Cell Smoothness	Cell Concavity	% of cancerous
10	4.7	0.2	80%
x1.5	x2.0	x1.3	
15 +	9.4 +	0.26 =	24.66

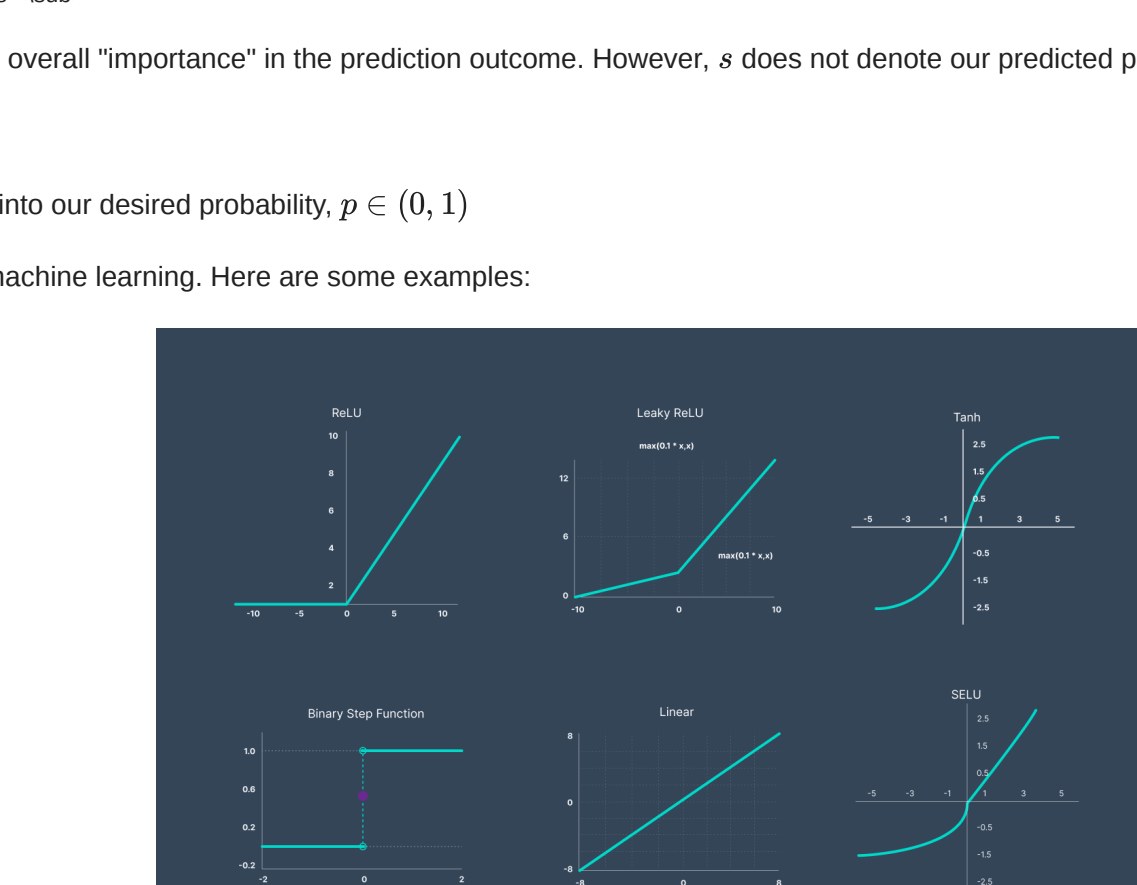
note these are made-up numbers for demonstration purposes <\sub>

Intuitively, the weight of each feature represents its overall "importance" in the prediction outcome. However, s does not denote our predicted probability.

1.2: Activation Function

Now, we need a function $\sigma(s)$ to transform $s\in\mathbb{R}$ into our desired probability, $p\in(0,1)$

These functions are called activation functions in machine learning. Here are some examples:

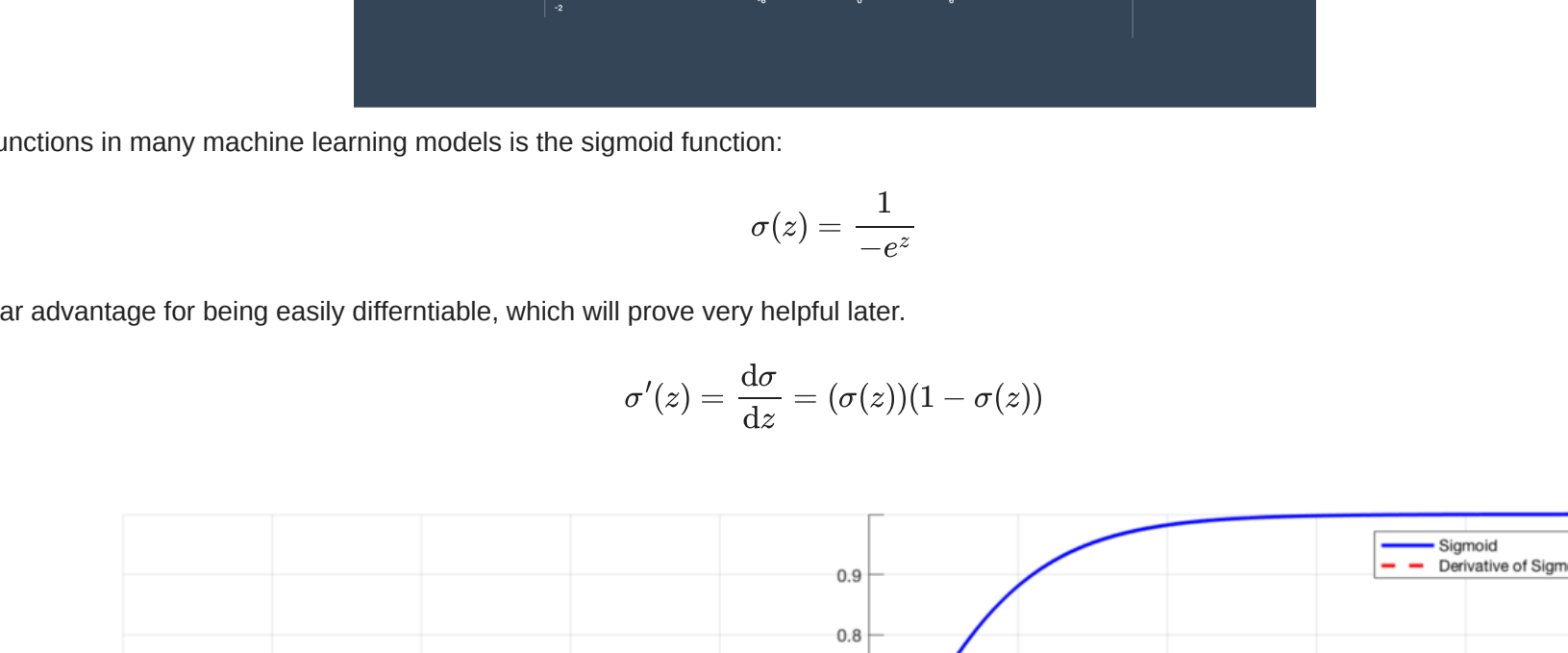


One of the most used activation functions in many machine learning models is the sigmoid function:

$$\sigma(z)=\frac{1}{-e^{z}}$$

The sigmoid function has a peculiar advantage for being easily differntiable, which will prove very helpful later.

$$\sigma'(z)=\frac{\mathrm{d}\sigma}{\mathrm{d}z}=(\sigma(z))(1-\sigma(z))$$



1.3: Introducing Biases

In order to horizontally shift s such that it fits within the "useful" domain of the sigmoid function (so it doesn't just evaluate to 0.00001 or 0.99998 every time), we add an additional bias, a scalar b. Putting everything together, we have:

$$z=w\cdot x+b$$

1.4: Summary

$$p=\sigma(z)=\frac{1}{-e^{w\cdot x+b}}$$

In machine learning, we usually denote model parameters, in this case w and b, as θ. More formally,

$$\theta=<w,b>$$

As a function of theta, our model makes predictions on

$$f(x,\theta)=\frac{1}{-e^{\theta_w}\cdot x+\theta_b}$$

Now, the problem of predicting probability is reduced to the problem of finding optimal weights (w) and bias (b). The finding of optimal weights and bias is the "learning" part of machine learning. We will focus on a particular algorithm called called Logistic Regression, that makes predictions based on $\sigma(w\cdot x+b)$.

Section 2: Gradient Descent

2.1: Introducting Loss Functions

In order to find the optimal weights and bias, we first need to define what "optimal" means. In machine learning, optimal means minimizing error. This error, depends on the difference between our predicted outcome and the expected outcome, and can be defined as a function called the loss function or cost function. The smaller our loss function, the better our model is doing. In your AP Statistics class you may have learned about least squared error ($\sum(p-y)^2$), which is indeed a type of loss function, but it's typically used in another type of task in machine learning called regression. For binary classification, there are a few popular options:

- Cross Entropy: $L_{CE}(y,p)=-(y\log(p)+(1-y)\log(1-p))$
- Negative Loglikelihood: $L_{NL}(y,p)=-\log(p(y))$
- Hinge Loss: $L_{HL}(y,p)=max(0,1-y\cdot(\lceil p+0.5\rceil))$

Note y is the actual (expected) binary 0/1 classes, while p is our predicted probability $\in[0,1]$ <\sub>

Logistic Regression uses Cross Entropy, a standard and utilitarian loss function in binary classification.

Recall from the previous section that $p=\sigma(w\cdot x+b)$. Plugging this in:

$$L_{CE}(w,b,y)=y\log(\sigma(w\cdot x+b)))+(1-y)\log(1-\sigma(w\cdot x+b))$$

Again, y is our actual/expected binary classes, w is the vector of our weights, and b is the scalar of our bias.

To improve our model, we need to minimize this function... This sounds like Multivariable Calculus!

2.2: Minimizing the Loss Function

In order for our model to be able to "intellegently" deal with any input we give it, it needs to train on large amounts of diverse data. In general, we want to select weights and biases that will minimize L_{CE} over m sets of observations, each containing n features and 1 expected output (y). For example, here is what UCI's breast cancer dataset:

Features | Expected Class

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	fractal_dimension_mean	...	texture_worst	perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	fractal_dimension_worst	Cancer Diagnosis
0	15.270	12.91	98.17	725.5	0.08182	0.06230	0.05892	0.031570	0.1359	0.05526	...	15.92	113.70	932.7	0.12220	0.2186	0.29620	0.10350	0.2320	0.07474	0
1	9.295	13.90	59.96	257.8	0.13710	0.12250	0.03332	0.024210	0.2197	0.07696	...	17.84	67.84	326.6	0.18500	0.2097	0.09996	0.07262	0.3681	0.08982	0
2	20.640	17.35	134.80	1335.0	0.09446	0.10760	0.15270	0.089410	0.1571	0.05478	...	23.17	166.80	1946.0	0.15620	0.3055	0.41590	0.21120	0.2689	0.07055	1
3	11.320	27.08	71.76	395.7	0.06883	0.03813	0.01633	0.003125	0.1869	0.05628	...	33.75	79.82	452.3	0.09203	0.1432	0.10890	0.02083	0.2849	0.07087	0
4	12.750	16.70	82.51	493.8	0.11250	0.11170	0.03880	0.029950	0.2120	0.06623	...	21.74	93.63	624.1	0.14750	0.1979	0.14230	0.08045	0.3071	0.08557	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
564	12.770	29.43	81.35	507.9	0.08276	0.04234	0.01997	0.014990	0.1539	0.05637	...	36.00	88.10	594.7	0.12340	0.1064	0.08653	0.06498	0.2407	0.06484	0
565	16.140	14.86	104.30	800.0	0.09495	0.08501	0.05500	0.045280	0.1735	0.05875	...	19.58	115.90	947.9	0.12060	0.1722	0.23100	0.11290	0.2778	0.07012	0
566	9.847	15.68	63.00	293.2	0.09492	0.08419	0.02330	0.024160	0.1387	0.06891	...	22.99	74.32	376.5	0.14190	0.2243	0.08434	0.06528	0.2502	0.09209	0
567	11.080	18.83	73.30	361.6	0.12160	0.21540	0.16890	0.063670	0.2196	0.07950	...	32.82	91.76	508.1	0.21840	0.9379	0.84020	0.25240	0.4154	0.14030	1
568	13.340	15.86	86.49	520.0	0.10780	0.15350	0.11690	0.069870	0.1942	0.06902	...	23.19	96.66	614.9	0.15360	0.4791	0.48580	0.17080	0.3527	0.10160	0

569 rows × 31 columns

We have 31 features ($n=31$) and a total of 569 observations/rows ($m=569$).

To make our model more versatile and adaptable, we need to minimize the loss function over all m observations. More formally, we aim to minimize:

$$\frac{1}{m}\sum_{i=1}^{m}L_{CE}(\theta,y_i)$$

After putting everything together, we wish to find:

$$\text{argmin}_{\theta}\frac{1}{m}\sum_{i=1}^{m}y_i\log(\sigma(\theta_w\cdot x_i+\theta_b))+(1-y_i)\log(\sigma(\theta_w\cdot x_i+\theta_b))$$

This is exactly the task of gradient descent.

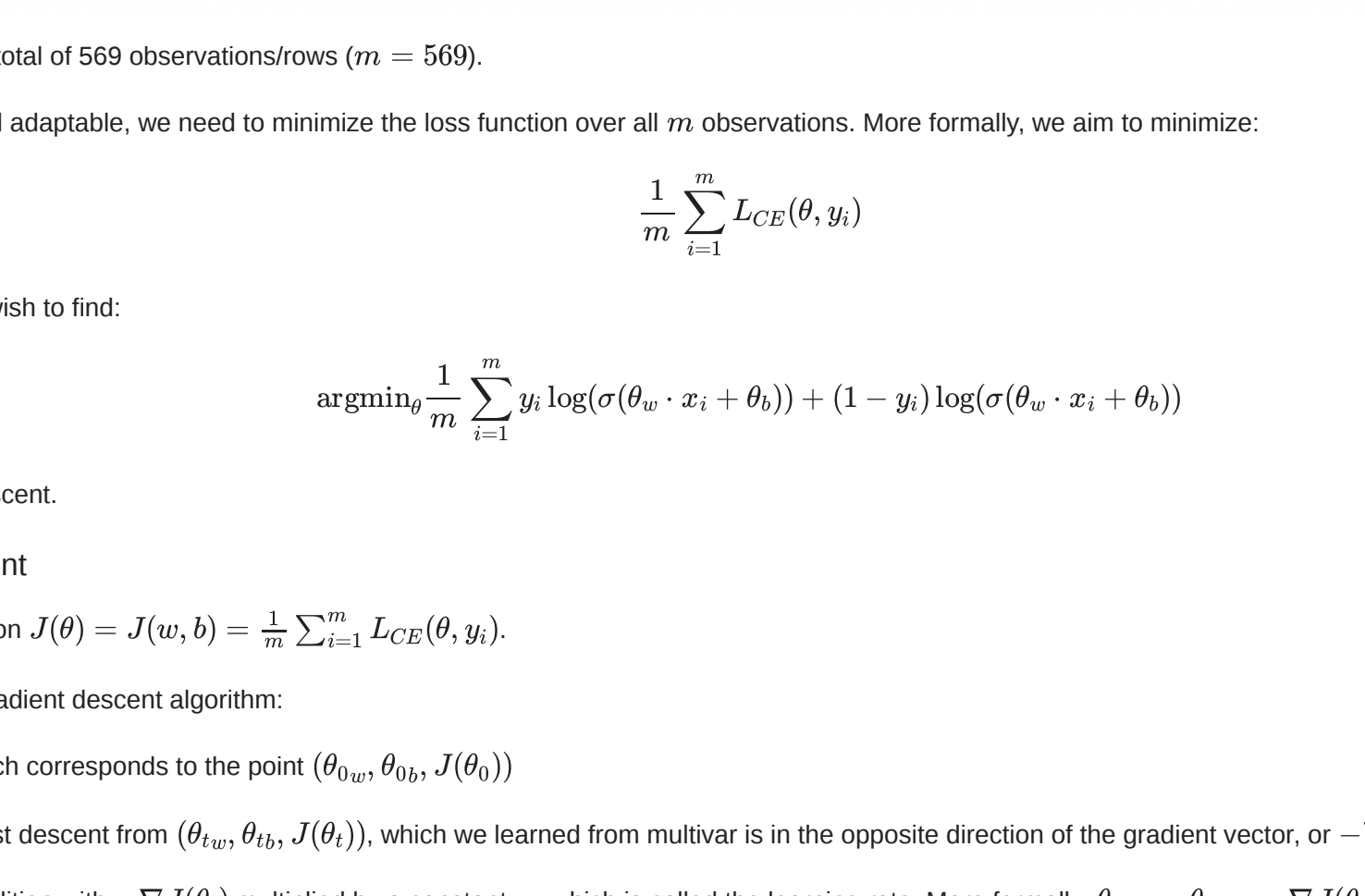
2.3: Introducting Gradient Descent

In gradient descent, we call this function $J(\theta)=J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L_{CE}(\theta,y_i)$.

Here are the steps of a bare-bones gradient descent algorithm:

1. Generate a random $\theta_t=\theta_0$, which corresponds to the point $(\theta_{0w},\theta_{0b},J(\theta_0))$
1. Determine the direction of greatest descent from $(\theta_{tw},\theta_{tb},J(\theta_t))$, which we learned from multivar is in the opposite direction of the gradient vector, or $-\nabla J(\theta_t)$
1. Modify θ by performing vector addition with $-\nabla J(\theta_t)$ multiplied by a constant, η, which is called the learning rate. More formally, $\theta_{t+1}=\theta_t-\eta\cdot\nabla J(\theta_t)$
1. Repeat from step 2 with θ_t iff $\theta_t\neq\theta_{t+1}$ and $t<C$, where C is some constant indicating the maximum number of iterations

Here is a visualization of the outlined process, with vectors indicating each iteration of the algorithm:



In this example, we took 13 steps from our initial θ_0, so $\theta_{final}=\theta_{13}$.

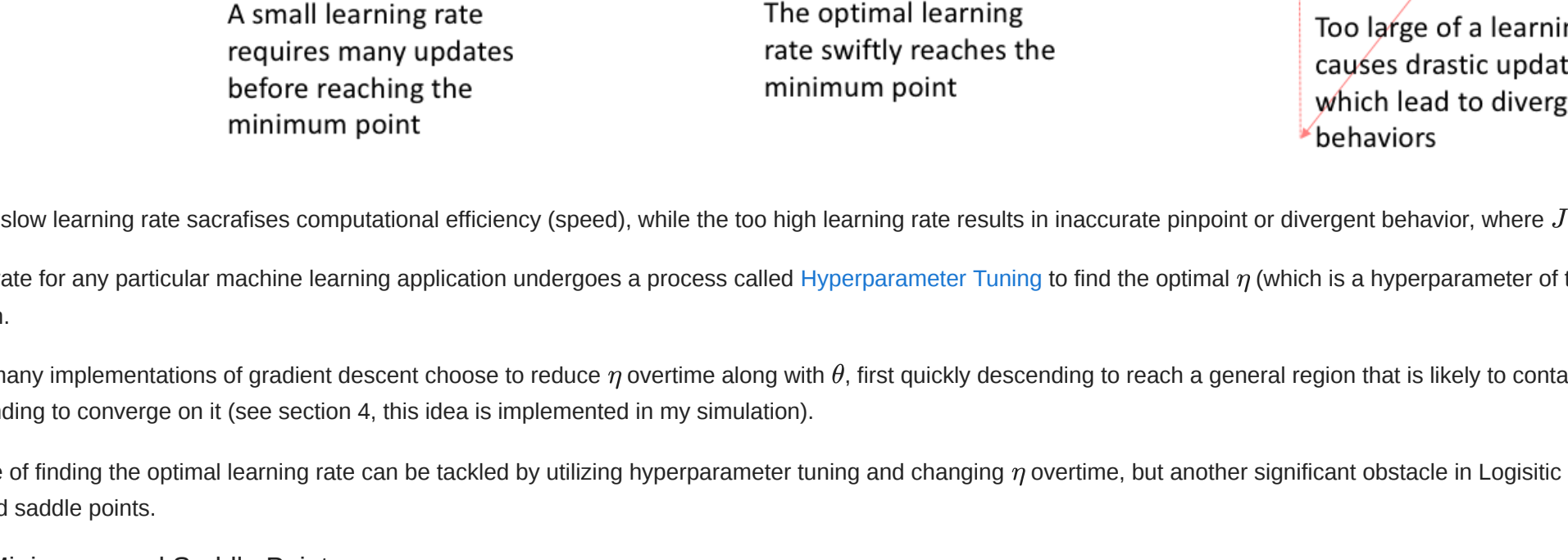
Informally, we can think of this as incremental taking steps down a "mountain," which is the surface curve of our function g. Each step, each iteration, we are getting closer and closer to reaching the global minimum.

2.4: Learning Rate

The learning rate, η, determines how fast we descend down J. Recall the equation for modifying θ:

$$\theta_{i+1}=\theta_i-\eta\cdot\nabla J(\theta_i)$$

The larger the η, the faster we descent. However, this comes at the cost of less accuracy in pinpointing the exact location of the minimum. Here's an oversimplified but helpful visualization in 2D:



Now note how the slow learning rate sacrafises computational efficiency (speed), while the too high learning rate results in inaccurate pinpoint or divergent behavior, where $J(\theta_{i+1})>J(\theta_i)$

The learning rate for any particular machine learning application undergoes a process called Hyperparameter Tuning to find the optimal η (which is a hyperparameter of the model) for its particular application and g function.

Additionally, many implementations of gradient descent choose to reduce η overtime along with θ, first quickly descending to reach a general region that is likely to contain the global minimum, then more slowly descending to converge on it (see section 4, this idea is implemented in my simulation).

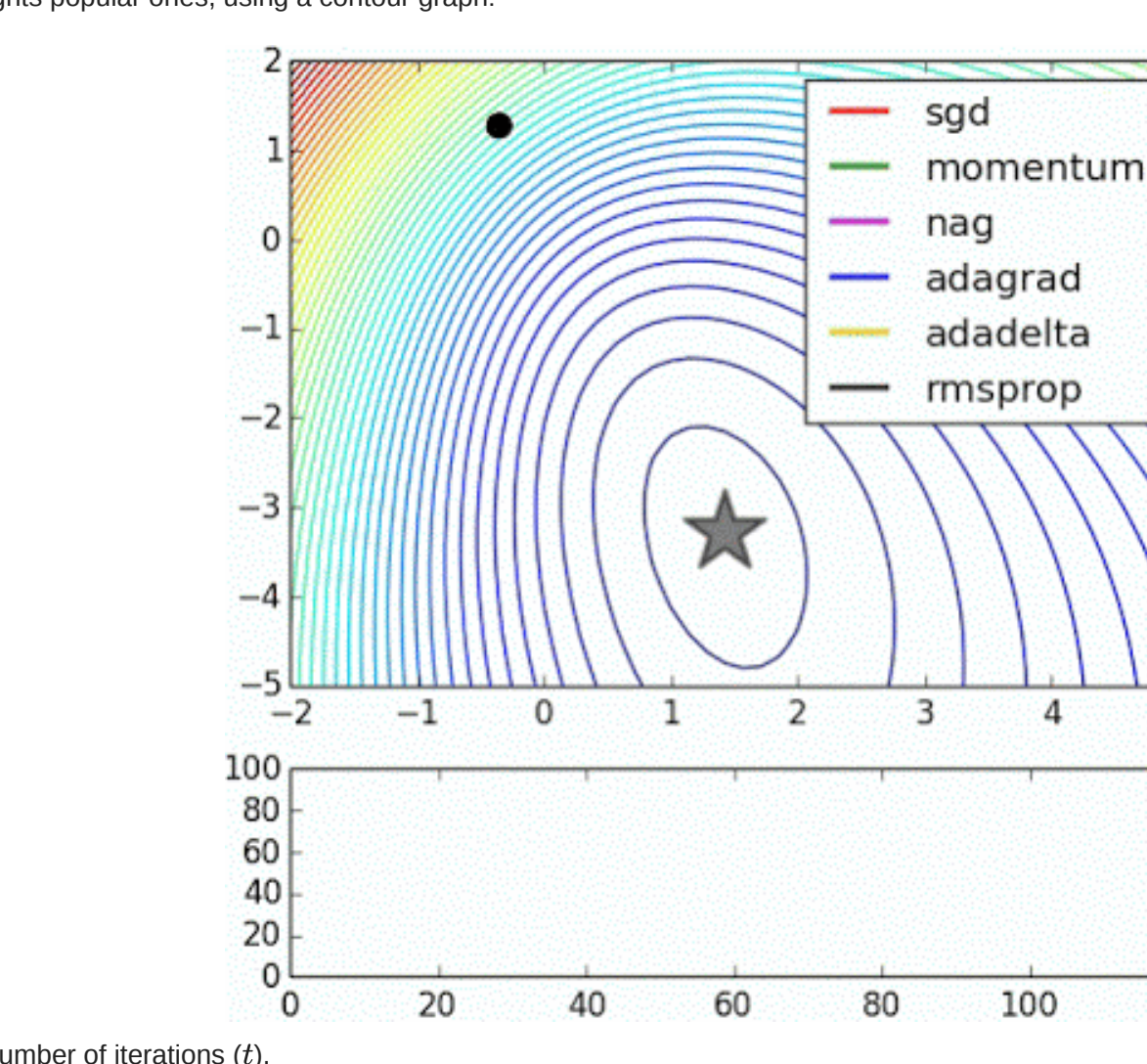
The challenge of finding the optimal learning rate can be tackled by utilizing hyperparameter tuning and changing η overtime, but another significant obstacle in Logisitic Regression is escaping local minimums and saddle points.

2.5: Local Minimums and Saddle Points

As we've learned in multivar, saddle points and local critical points ($f_x=0, f_y=0$) have the same ∇J as global minimums. Many variations of the bare-bones, classical gradient descent have sprung up to address this issue. For example:

- Stochastic gradient descent: we add noise on top of our loss function, and perform gradient descent on what's effectively the "noisy gradient", so as to introduce potential paths to escape out of a local minimum/saddle point.
- Momentum gradient descent: we treat θ intuitively as a ball that is rolling downhill, adding the idea of momentum so that it can escape local minimums/saddle points as it "rolls downhill".

There are other variations, and this gif highlights popular ones, using a contour graph:



The bottom graph demonstrates $J(\theta)$ over number of iterations (t).

Another simple idea is to perform gradient descent on many random points to maximize our chance of finding the global minimum on one of them, but this comes at the cost of computational efficiency.

2.6: Deriving and Finding ∇J

First, recall from section 2.2 that

$$J(\theta)=J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L_{CE}(\theta,y_i)=\frac{1}{m}\sum_{i=1}^{m}y_i\log(\sigma(\theta_w\cdot x_i+\theta_b))+(1-y_i)\log(\sigma(\theta_w\cdot x_i+\theta_b))$$

Second, recall from section 1.2 the derivative of a sigmoid:

$$\sigma'(z)=\frac{\mathrm{d}\sigma}{\mathrm{d}z}=(\sigma(z))(1-\sigma(z))$$

Because both the loss function (cross entropy) and activation function (sigmoid) are differentiable, we find that

$$J'(\theta)=\begin{bmatrix}\frac{\partial J}{\partial w}\\ \frac{\partial J}{\partial b}\end{bmatrix}=[\dots]=\begin{bmatrix}\frac{1}{N}\sum 2x_i(\hat{y}-y_i)\\ \frac{1}{N}\sum 2(\hat{y}-y_i)\end{bmatrix}$$

Note that y-hat, or predicted output, is equivalent to p, the predicted probability. Here, N denotes the number of observations/samples/rows in our training dataset instead of m.

Let's start by taking the partial derivative of the loss function with respect to a single weight w_j:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial \mathbf{w}_j}&=\frac{\partial}{\partial \mathbf{w}_j}-[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))]\\&=-\left[\frac{\partial}{\partial \mathbf{w}_j}y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+\frac{\partial}{\partial \mathbf{w}_j}(1-y)\log[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\right]\end{aligned}$$

Then, using the chain rule and the fact that $\frac{d}{dx}ln(x)=\frac{1}{x}$:

$$\frac{\partial L_{CE}}{\partial \mathbf{w}_j}=-\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial \mathbf{w}_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b)-\frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial \mathbf{w}_j}1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)$$

Rearranging:

$$\frac{\partial L_{CE}}{\partial \mathbf{w}_j}=-\left[\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}-\frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\right]\frac{\partial}{\partial \mathbf{w}_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b)$$

Now plugging in derivative of a sigmoid and using the chain rule again:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial \mathbf{w}_j}&=-\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\frac{\partial(\mathbf{w}\cdot\mathbf{x}+b)}{\partial \mathbf{w}_j}\\&=-\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\mathbf{x}_j\\&=-[y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\mathbf{x}_j\\&=[\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]\mathbf{x}_j\end{aligned}$$

For biases, we follow a very similar procedure, but take the partial derivative of b-in respect to J instead.

Section 3: Ideas in Practice (Implementation With Detailed Comments)

We also test our implementation on a real dataset provided by UC Irvine, predicting breast cancer based on a set of cell parameters. More details on the dataset here. The dataset is avaliable for download here.

In [1]:
```python
#  numpy for fast math operations, like dot product
import numpy as np

# sklearn for breast cancer dataset, a function to split the dataset, confusion matrix, accuracy score
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, plot_confusion_matrix

# pandas for handling breast cancer dataset as a DataFrame
import pandas as pd

# seaborn and matplotlib for graphing
import seaborn as sn
import matplotlib.pyplot as plt
```

```
/Users/gaofamily/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of S
ciPy (detected version 1.23.5)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is required for this version of "
```

In [2]:
```python
# learning rate for gradient descent
LR = 0.0015

# number of random starting points to take. We are utilizing the idea in 2.5, taking many random starting points
n_iters = 25

# number of steps to take in GD
n_steps = 1000

# weights: relative importances of features
# bias: "shift"/location of the activation function
# activation function is sigmoid
weights, biases = 0, 0
```

In [3]:
```python
# activation function
def sigmoid(x):
    """
    Computes sigmoid function to calculate probability based on w * x + b
    Input: Scalar, w * x + b
    Returns: Scalar, probability in (0, 1)
    """
    return 1.0/(1.0 + np.exp(-x))

# loss function:
def cross_entropy(predictions, targets, epsilon = 1e-14):
    """
    Computes cross entropy between predicted probabilities and actual 0/1 classifications.
    Input: predictions vector of predicted classes' probabilities
           targets vector of actual 0/1 classes
    Returns: scalar representing loss
    """
    n = predictions.shape[0]
    return -(1/n) * np.sum(targets*np.log(predictions + epsilon) + (1-targets)*np.log(1 - predictions + epsilon))

# calculating the derivative of J
def calc_dir(x, bias, predicted, actual):
    """
    Computes the gradient descent vector or maximum direction of descent for use to update theta
    Input: weights = coefficients for features
           bias = single-value added to prediction output
           predicted = predicted 0/1 classes for classification output
           actual = actual 0/1 classes for classification output
    Returns: tuple of 2 elements
             gradient descent vector <dw, db> = vector of greatest descent for weights and biases, respectively
    """
    N = len(weights)

    # compute partial derivative for weights
    dW = (1/N) * 2 * np.dot(x.T, predicted - actual)
    # compute partial derivative for bias
    dB = (1/N) * 2 * np.sum(predicted - actual)

    return [dW, dB]
```

In [4]:
```python
# gradient descent algorithm, as discussed in section 2
def gradient_descent(weights, bias, x, y, LR, report=False):
    """
    Performs gradient descent to optimize initial starting weights and bias from a given starting point

    Input: x = training set containing input features
           y = training set containing expected classification
           weights = n-d vector/numpy_array
           bias = scalar
           LR = scalar, learning rate
           report = wether prin
    Returns: tuple of 2 elements, (weights, bias)
    """

    for i in range(n_steps):
        # for each row of features, we generate a corresponding number by multiplying each weight by feature value
        predicted_vals = np.dot(x, weights.T) + bias

        # for each feature value, we predict its class probability utilizing the sigmoid function
        predicted_probs = sigmoid(predicted_vals)

        if report:
            print("On step " + str(i) + "th step, loss equals " + str(cross_entropy(predicted_probs, y)))

        # calculated derivatives
        derivs = calc_dir(x, bias, predicted_probs, y)
        dW = derivs[0]
        dB = derivs[1]

        # "descent" down using learning rate
        weights -= LR * dW
        bias -= LR * dB

    return [weights, bias]
```

In [5]:
```python
# train function that puts everything together
def train(x, y, report = False):
    """
    Optimizes weights and biases given initial training set. Utilizes entire training set (must train-test-split beforehand)

    Input: x = model input of training dataset, 2darray containing m samples (entries) and n features
           y = expected model output of training dataset, 1darray containing n classification results (0/1)

    """
    # declare variables to use
    global weights, bias
    m, n = x.shape

    # we take n_iters random starting points
    for _ in range(n_iters):
        # random starting points: random vector/np_array of size n
        weights = np.random.rand(n)

        # random starting points: random bias
        bias = np.random.rand(1)[0]

        # gradient descent from this random starting point
        res = gradient_descent(weights, bias, x, y, LR, report)

        # extract content from result
        weights = res[0]
        bias = res[1]
```

In [6]:
```python
# predicts probability of a given test set (that is completely different from the training set), utilizing weights, bias, and the sigmoid function
def predict_probs(x):
    """
    Utilizes pre-optimized weights and biases to outputs class probability given m entries of nd-vector of features

    Input: m by n matrix of feature
    Output: m-d vector of class probabilities for each entry
    """

    # utilize formula discussed in section 1 to acquire values: w * x + b
    predicted_vals = np.dot(x, weights.T) + bias
    # apply the sigmoid function to convert value to probability
    predicted_probs = sigmoid(predicted_vals)

    return predicted_probs

# rounds predicted probabilities to the nearest integer (0/1) to predict actual binary classes
def predict(x):
    """
    Utilizes pre-optimized weights and biases to outputs class given m entries of nd-vector of features

    Input: m by n matrix of feature
    Output: m-d vector of 0/1 class for each entry
    """

    # call predict_probs function to predict probabilities
    x = predict_probs(x)
    # round the numbers
    x = [0 if i < 0.5 else 1 for i in x]

    return x
```

In [7]:
```python
%%time
# ^ the above displays time for this cell to run

# load dataset
data = load_breast_cancer()
# make variables x, the features, and y, the output class
x, y = data.data, data.target
# perform train test split with 80% to 20%, dividing our original dataset. The training set will be used to optimize weights and biases, then the completely untouche
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=69)
# call the train function to optimize weights and biases based on the training dataset
train(x_train, y_train)

print("Time for training Logistic Regression Model:")
```

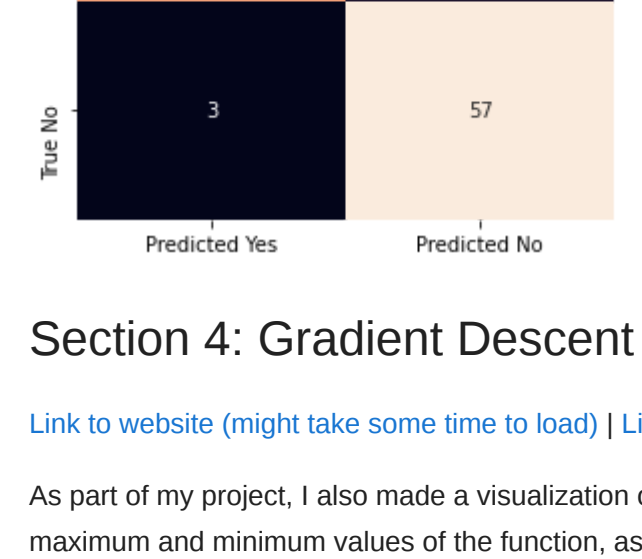
```
<ipython-input-3-eca20362b6bd>:8: RuntimeWarning: overflow encountered in exp
  return 1.0/(1.0 + np.exp(-x))
Time for training Logistic Regression Model:
CPU times: user 2.46 s, sys: 5.65 s, total: 8.1 s
Wall time: 1.68 s
```

In [8]:
```python
%%time
# ^ the above displays time for this cell to run

# call the predict function on testing dataset
y_pred = predict(x_test)

print("The final accuracy of the model is " + str(100 * accuracy_score(y_test, y_pred)) + "%.")
print("Time for testing Logistic Regression Model:")
```

```
The final accuracy of the model is 91.22807017543859%.
Time for testing Logistic Regression Model:
CPU times: user 665 µs, sys: 2.35 ms, total: 3.02 ms
Wall time: 635 µs
<ipython-input-3-eca20362b6bd>:8: RuntimeWarning: overflow encountered in exp
  return 1.0/(1.0 + np.exp(-x))
```

In [9]:
```python
# make a confusion matrix to better understand our results
array = confusion_matrix(y_test, y_pred)
# turn the confusion matrix into a DataFrame to plot it
df_cm = pd.DataFrame(array, index = ['True Yes', 'True No'],
                    columns = ['Predicted Yes', 'Predicted No'])

# use matplotlib and seaborn to plot it
plt.figure()
# use matplotlib and seaborn to plot it
sn.heatmap(df_cm, annot=True)
```

Out[9]:
```
<AxesSubplot:>
```



Section 4: Gradient Descent Visualization

Link to website (might take some time to load) | Link to code

As part of my project, I also made a visualization of the gradient descent algorithm on an interactive 3D graph. The user can customize the graph by entering their own equation for the function $f(x,y)$, the maximum and minimum values of the function, as well as the learning rate η. After graphing, a visual similar to that in section 2.3 will be displayed, where vectors scaled to the magnitude of each step taken in gradient descent will be shown on the 3D graph. The user can use their cursor to rotate the graph, zoom in and out (by scrolling), pan the graph, take a screenshot, and play around with different camera angles. The minimum value found by gradient descent by also be displayed at the bottom.

In my simulation, I employed in the idea in section 2.4, utilizing

$$\eta_{t+1}=\eta_t\cdot\frac{1}{e^{\alpha}}$$

to reduce the learning rate overtime. α is a constant that I experimentally found to be optimal at around ~0.3. This exact formula is inspired by Simulated Annealing, a meta-heuristic for optimization problems whose core idea is quite similar to gradient descent.

To find the global minimum, I also generated many random points to maximize the chance of finding a global minimum on one of them, as in section 2.5.

To actually build the graph itself, I utilized the plotly graphing library, computing many disjoint points on a 2D meshgrid, then connecting them to form the 3D graph. The core idea is that of Riemann Sums, but extended to three dimensions, and rather than finding the area, we're connecting points to build a visualization. As a consequence of this method of generating the graph, 1) the global minimum found may be off by a tiny amount (as the actual minimum can only be achieved iff it is one of the points computed) and 2) the visualization struggles with graphs with a large domain because it'll have to compute quadratically more points, so the max and min are capped (note however that the gradient descent algorithm could perfectly handle a larger domain, as it does for real machine learning loss function such as the one it faces in this notebook's breast cancer dataset).

Section 5: References

Fernandes, Antônio Alves Tôrres, et al. "Read this paper if you want to learn logistic regression." Revista de sociologia e politica 28 (2021).

Sanderson, Grant. "Gradient Descent, How Neural Networks Learn | Chapter 2, Deep Learning." YouTube, YouTube, 16 Oct. 2017, https://www.youtube.com/watch?v=IHZwWFHWa-w.

Sebastian Ruder. "An Overview of Gradient Descent Optimization Algorithms." Sebastian Ruder, Sebastian Ruder, 20 Mar. 2020, https://ruder.io/optimizing-gradient-descent/.

Cloud Education, IBM. "What Is Gradient Descent?" IBM, https://www.ibm.com/cloud/learn/gradient-descent.

Chapter Logistic Regression - Stanford University. https://www.web.stanford.edu/~jurafsky/slp3/5.pdf.

Kumar, Satyam. "Overview of Various Optimizers in Neural Networks." Medium, Towards Data Science, 9 June 2020, https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5.

Soni, Devin. "Improving Vanilla Gradient Descent." Medium, Towards Data Science, 16 July 2019, https://towardsdatascience.com/improving-vanilla-gradient-descent-f9d91031ab1d.