Mathematics For Machine Learning

Directed Reading Project

Student names redacted for privacy

Mentored by Karan Srivastava
Fall 2021 Directed Reading Program

Objectives

Primary Objective: To obtain a general understanding of the basic mathematical methods for machine learning algorithms.

Secondary Objective: Create a basic regression neural network using Keras.

- Primary text:

  Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

Cholesky Decomposition

definition

A symmetric, positive definite matrix A can be factorized into a product of $A = LL^T$, where L is a lower triangular matrix with positive diagonal elements.

On the book "Mathematics for Machine Learning" we read in this project, there is an example of a 3×3 matrix example. According to the book, the algorithm can be written as:

Example 4.10 (Cholesky Factorization)
Consider a symmetric, positive definite matrix $\boldsymbol{A} \in \mathbb{R}^{3\times 3}$. We are interested in finding its Cholesky factorization $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top$, i.e.,

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \boldsymbol{L}\boldsymbol{L}^\top = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix} . \quad (4.45)$$

Multiplying out the right-hand side yields

$$\boldsymbol{A} = \begin{bmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{bmatrix} . \quad (4.46)$$

We will demonstrate it with python codes in the next slide.

Code Demonstration

The following code is a demonstration of the Cholesky Decomposition to fit $y = ax^2 + bx + c$ for the following 10 discrete points.

```
x = np.array([[4.1702],[7.2032],[0.0011],[3.0233],[1.4676],[0.9234],[1.8626],[3.4556],[3.9677],[5.3882]])
y = np.array([[53.9964],[152.1652],[0.5497],[31.1123],[7.7602],[5.8707],[13.1319],[38.6460],[47.4939],[86.0607]])
```

The actual process can be shown below:

```
def cholesky2(A):
    n = len(A)

    # Create zero matrix for L
    L = [[0.0] * n for i in range(n)]

    for i in range(n):
        for k in range(i+1):
            tmp_sum = sum(L[i][j] * L[k][j] for j in range(k))

            if (i == k): # Diagonal elements
                #L_{kk} = \sqrt{ a_{kk} - \sum^{k-1}_{j=1} L^2_{kj}}
                L[i][k] = math.sqrt(A[i][i] - tmp_sum)
            else:
                #L_{ik} = \frac{1}{L_{kk}} \left( a_{ik} - \sum^{k-1}_{j=1} L_{ij} L_{kj} \right)
                L[i][k] = (1 / L[k][k] * (A[i][k] - tmp_sum))
    return L
L=cholesky2(b)
L
```

```
[[65.79411909533059, 0.0, 0.0],
 [11.315179081105018, 3.6845670997450135, 0.0],
 [2.1523095726051, 1.9294347356270625, 1.2825151479137313]]
```

From here we can approximate a,b and c.

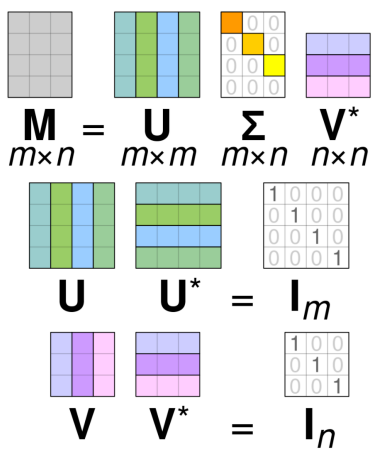
Singular Value Decompositions

Definition

Let $A \in \mathbb{R}^{n \times m}$ be a matrix with $m \leq n$. A singular value decomposition of A (SVD) is a matrix in the form:

$$A = U\Sigma V^T = \sum_{j=1}^{r} \sigma_j u_j V_j^T$$

with orthogonal matrix $U \in \mathbb{R}^{m \times m}$ with column vectors $u_j = 1, \ldots, m$ and $V \in \mathbb{R}^{n \times n}$ with column vectors $v_j = 1, \ldots, n$. Moreover, $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix. The graph below will provide a more intuitive explanation.



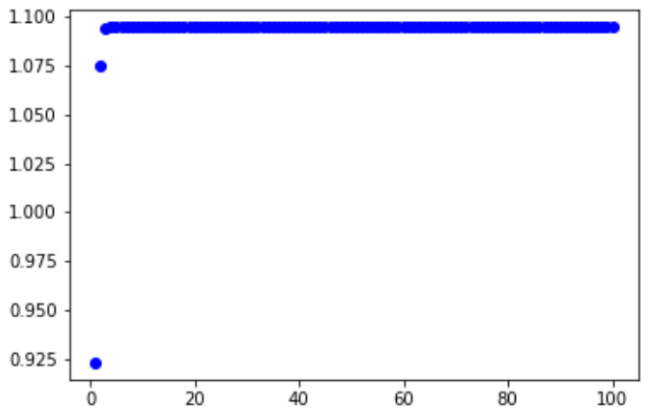
SVD Through Power Iteration

In general, SVD cannot be solved through an exact method. Iterative methods are usually used. Power iteration is one way to solve for SVD. The following code will demonstrate solving the biggest singular value of a random 3×4 matrix using power iteration.

```
A = np.random.rand(3,4)
```

```
np.random.seed(0)
def power_iteration(A,n):
    #As shown in Math535_Lecture_III_b.pdf
    b = np.transpose(A).dot(A)
    v_new_1 = np.random.rand(A.shape[1])

    for _ in range(n):
        v_new = b.dot(v_new_1)/ np.linalg.norm(b.dot(v_new_1))
        v_new_1 = v_new
    return v_new_1

y_l = []
for n in range(100):
    #compute svd
    u=power_iteration(A.T,n)
    v = power_iteration(A,n)
    #we want to have the orange value as on the pdf
    #if we multiply U (the first left comlun as from power iteration), M, V (the first right comlun)
    #we will get the orange value
    x = u.reshape((1,-1)) @ A @ v.reshape((-1,1))
    y_l.append(x[0][0])

x = list(range(1,101))
plt.plot(x,y_l,'ob')
plt.show()
```

```
1.0947415795641948
```




```
#to check my answer
my_svd = y_l[-1]
u,s,vh = np.linalg.svd(A)
true_svd = np.diag(s)[0]
print(my_svd,true_svd)
```

```
1.0947415795641948 [1.09474158 0.         0.         ]
```

Example: Movie Rating

Consider three viewers (Ali, Beatrix, Chandra) rating four different movies (Star Wars, Blade Runner, Amelie, Delicatessen). Their ratings are values between 0 (worst) and 5 (best) and encoded in a matrix $\boldsymbol{A} \in \mathbb{R}^{4\times 3}$

$$\boldsymbol{A} = \begin{array}{c} \begin{array}{ccc} \text{Ali} & \text{Beatrix} & \text{Chandra} \end{array} \\ \begin{bmatrix} 5 & 4 & 1 \\ 5 & 5 & 0 \\ 0 & 0 & 5 \\ 1 & 0 & 4 \end{bmatrix} \end{array} \begin{array}{c} \\ \text{Star Wars} \\ \text{Blade Runner} \\ \text{Amelie} \\ \text{Delicatessen} \end{array}$$

Example: Movie Rating

We interpret the left-singular vectors $\boldsymbol{u}_i$ as stereotypical movies and the right-singular vectors $\boldsymbol{v}_j$ as stereotypical viewers. The SVD of $\boldsymbol{A}$ is shown below:

$$\boldsymbol{A} = \underbrace{\begin{bmatrix} -0.6710 & 0.0236 & 0.4647 & -0.5774 \\ -0.7197 & 0.2054 & -0.4759 & 0.4619 \\ -0.0939 & -0.7705 & -0.5268 & -0.3464 \\ -0.1515 & -0.6030 & 0.5293 & -0.5774 \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} 9.6438 & 0 & 0 \\ 0 & 6.3639 & 0 \\ 0 & 0 & 0.7056 \\ 0 & 0 & 0 \end{bmatrix}}_{\Sigma} \underbrace{\begin{bmatrix} -0.7367 & -0.6515 & -0.1811 \\ 0.0852 & 0.1762 & -0.9807 \\ 0.6708 & -0.7379 & -0.0743 \end{bmatrix}}_{V^T}$$

Movie recommendation: Code Demonstration

We are using data sets from MovieLens, which contains 1 million ratings from 6000 users on 4000 movies.
Load the data sets with Pandas:

```
[10] data = pd.io.parsers.read_csv('ratings.dat',
         names=['user_id', 'movie_id', 'rating', 'time'],
         engine='python', delimiter='::')
     movie_data = pd.io.parsers.read_csv('movies.dat',
         names=['movie_id', 'title', 'genre'],
         engine='python', delimiter='::')
```

Create the ratings matrix with rows as movies and columns as users and normalize the matrix:

```
[11] ratings_mat = np.ndarray(
         shape=(np.max(data.movie_id.values), np.max(data.user_id.values)),
         dtype=np.uint8)
     ratings_mat[data.movie_id.values-1, data.user_id.values-1] = data.rating.values
```

```
[12] normalised_mat = ratings_mat - np.asarray([(np.mean(ratings_mat, 1))]).T
```

Movie recommendation: Code Demonstration

Compute SVD:

```
[13] A = normalised_mat.T / np.sqrt(ratings_mat.shape[0] - 1)
     U, S, V = np.linalg.svd(A)
```

Helper functions to select and display the most similar movies:

```
[14] def top_cosine_similarity(data, movie_id, top_n=10):
         index = movie_id - 1 # Movie id starts from 1
         movie_row = data[index, :]
         magnitude = np.sqrt(np.einsum('ij, ij -> i', data, data))
         similarity = np.dot(movie_row, data.T) / (magnitude[index] * magnitude)
         sort_indexes = np.argsort(-similarity)
         return sort_indexes[:top_n]

     # Helper function to print top N similar movies
     def print_similar_movies(movie_data, movie_id, top_indexes):
         print('Recommendations for {0}: \n'.format(
         movie_data[movie_data.movie_id == movie_id].title.values[0]))
         for id in top_indexes + 1:
             print(movie_data[movie_data.movie_id == id].title.values[0])
```

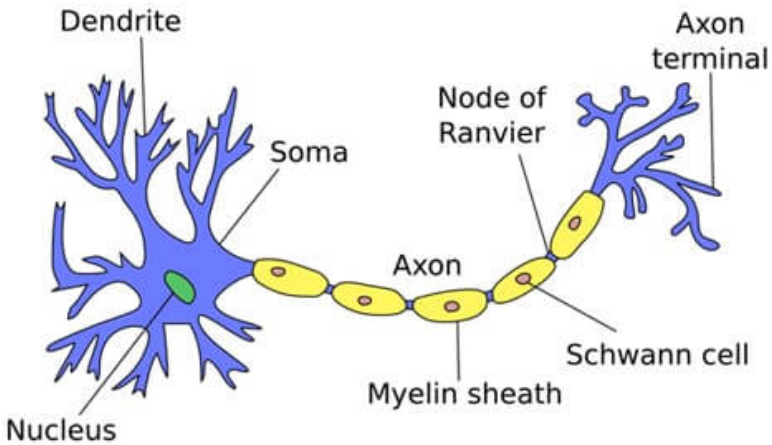
Movie recommendation: Code Demonstration

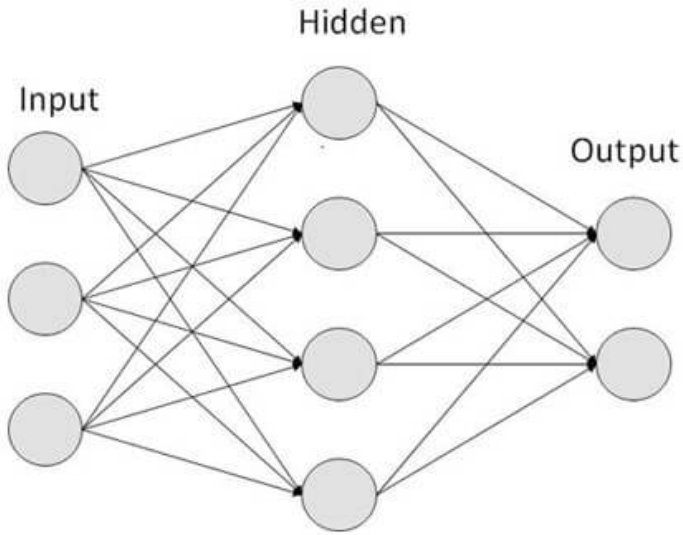
Test with one movie ID

```
[15] k = 50
     movie_id = 521 # Grab an id from movies.dat
     top_n = 10

     sliced = V.T[:, :k] # representative data
     indexes = top_cosine_similarity(sliced, movie_id, top_n)
     print_similar_movies(movie_data, movie_id, indexes)
```

```
Recommendations for Romeo Is Bleeding (1993):

Romeo Is Bleeding (1993)
Priest (1994)
Tough and Deadly (1995)
Chungking Express (1994)
Three Colors: White (1994)
Hideaway (1995)
Little Big League (1994)
Malice (1993)
National Lampoon's Senior Trip (1995)
Calendar Girl (1993)
```

Intro to Neural Network



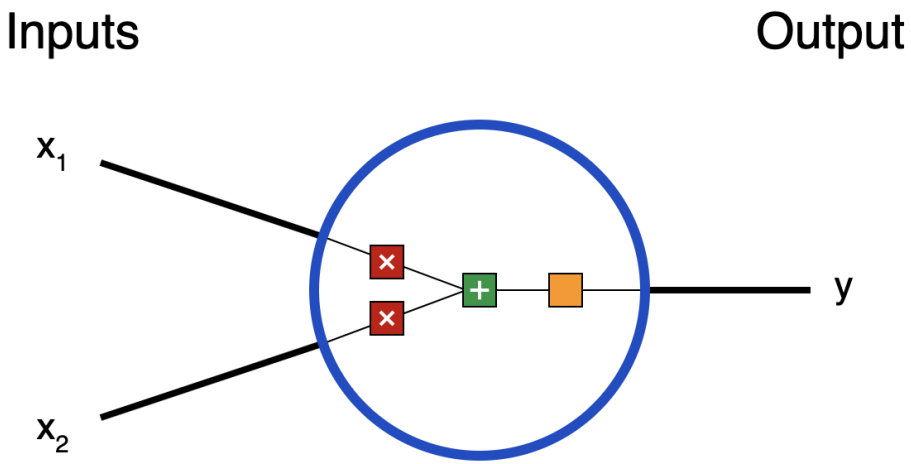


Intro to Neural Network

Key components of Neural Network:

- Input Nodes (input layer)
- Hidden nodes (hidden layer)
- Output Nodes (output layer)
- Connections and weights
- Activation function

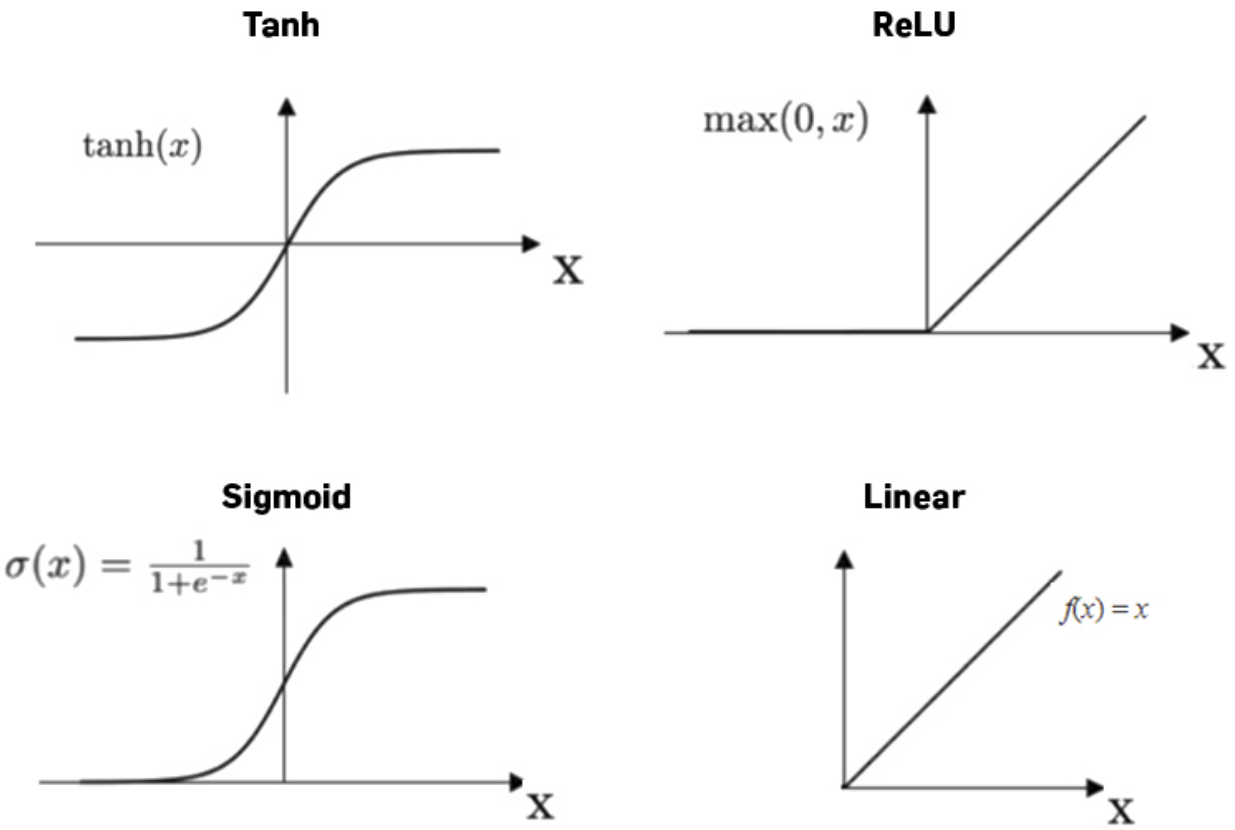
Intro to Neural Network



- Red: Multiply by weights w_i: $w_1 * x_1$ and $w_2 * x_2$
- Green: Add the terms together: $w_1 * x_1 + w_2 * x_2 + b$
- Orange: pass through an activation function: $f(w_1 * x_1 + w_2 * x_2 + b)$

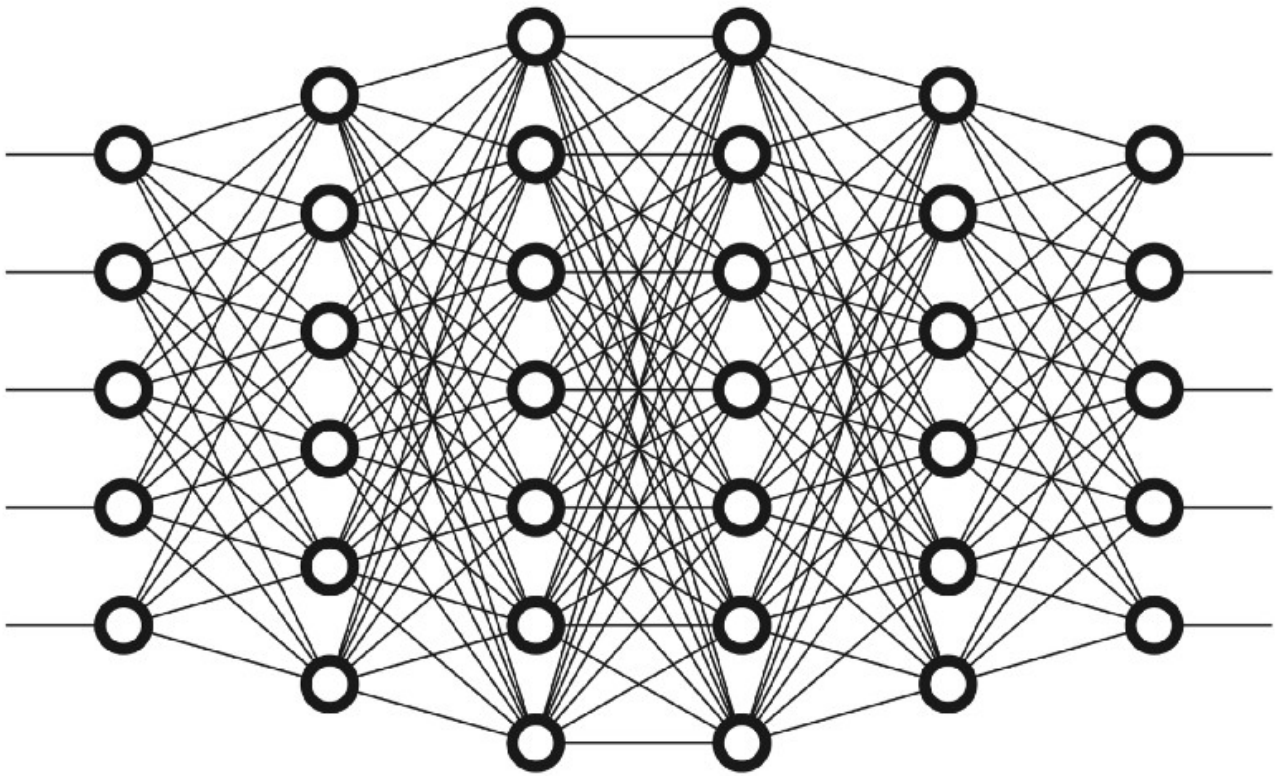
Intro to Neural Network

Commonly used activation functions:



Intro to Neural Network

Deep Learning: multiple hidden layers



Intro to Neural Network

How to select: number of layers, number of neurons per layer, activation function, weights?
Minimize loss function when training! Commonly used loss functions:

- Ordinary Least Squared Loss Function: $L(\theta) = \sum_{i=1}^{i=N} (y_i - \hat{y_i})^2$
- Cross-Entropy Loss Function: $L(\theta) = -\sum_{i=1}^{i=N} \hat{y_i} \times log(y_i)$
- Mean Absolute Percentage Error: $L(\theta) = \frac{100\%}{N} \sum_{i=1}^{i=N} \frac{|y_i - \hat{y_i}|}{y_i}$

How to minimize the loss function? Take derivative and set to zero!

Taking Derivatives w.r.t. a Matrix

Gradient Definition Deisenroth et al. [2020]

For a function $f : \mathbb{R}^n \to \mathbb{R}$, $x \mapsto f(x)$, $x \in \mathbb{R}^n$ of n variables $x_1, x_2, \ldots, x_n$, define the partial derivatives as

$$\frac{\partial f}{\partial x_1} = \lim_{h \to 0} \frac{f(x_1 + h, x_2, \ldots, x_n) - f(x)}{h}$$

$$\vdots$$

$$\frac{\partial f}{\partial x_n} = \lim_{h \to 0} \frac{f(x_1, x_2, \ldots, x_n + h) - f(x)}{h}$$

and collect them in the row vector

$$\nabla f = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}.$$

Derivatives w.r.t. a Matrix

Gradient Definition Deisenroth et al. [2020]

Similarly, if $f : \mathbb{R} \to \mathbb{R}^m$ with m outputs $f_1, \ldots, f_m$, define

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \\ \vdots \\ \frac{\partial f_m}{\partial x} \end{bmatrix}.$$

Jacobian Matrix

Informal Construction of the Jacobian

Jacobian Matrix

Based on the prior two definitions, we can generalize the derivative of any function $f : \mathbb{R}^n \to \mathbb{R}^m$, $x \mapsto f(x)$, $x \in \mathbb{R}^n$ in the form of the Jacobian matrix. Letting $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ and $f = (f_1, f_2, \ldots, f_m) \in \mathbb{R}^m$, we have

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} \qquad \text{(Def. of Gradient)}$$

$$= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \qquad \text{(Def. of Gradient)}$$

Jacobian Matrix as a Linear Transform

One can verify that the Jacobian matrix behaves in the manner we expect a derivative to behave. Notice that, for some differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$ and $x, x_0 \in \mathbb{R}^n$ with $x \neq x_0$,

$$\lim_{x \to x_0} \frac{\|f(x) - f(x_0) - \partial_x f(x_0) \cdot (x - x_0)\|_2}{\|x - x_0\|_2} = 0,$$

where $\partial_x f(x_0)$ is the Jacobian matrix of f with respect to x evaluated at the x_0, and $(\cdot)$ is the standard matrix multiplication operator. This limit implies that the Jacobian is the best locally linear approximation of f at x_0, as expected.

Visual

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of n variables $x_1, \ldots, x_n$. We wish to find a quick way to find

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{or} \quad \max_{x \in \mathbb{R}^n} f(x).$$

In the context of neural networks, we typically seek to find minimums. To do so, one may choose to employ a standard gradient descent algorithm.

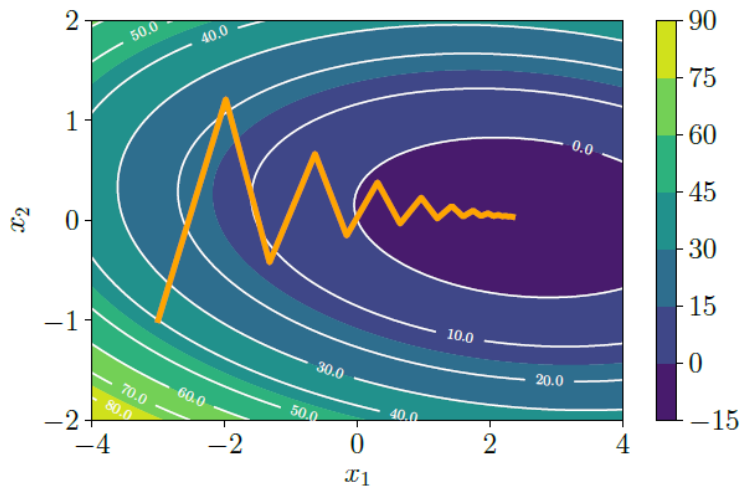


Figure: A sample gradient descent on a two-dimensional quadratic surface. Deisenroth et al. [2020]

Optimization with Gradient Descents

Gradient Descent

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function of n variables. Let $x \in \mathbb{R}^n$ be the column vector such that $f(x)$ is a local minimum. Then x is the limit of the sequence $\{x_i\}_{i=0}^{\infty}$ given recursively by

$$x_{i+1} = x_i - \gamma_i (\nabla f(x_i))^\top,$$

for some choice of x_0. Each $\gamma_i \in \mathbb{R}$ in the sequence $\{\gamma_i\}_{i=0}^{\infty}$ is called the step size.

Bringing Everything Together: Matrix Backpropagation

For sake of example, construct a neural network with the following properties:

- Two inputs and two outputs
- One hidden layer containing two "neurons" (nodes)
- There are $2 * 2 * 2 = 8$ weights given by $w_1, \ldots, w_8$,
- The activation function for each neuron is given by the sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{1}$$

The Example NN

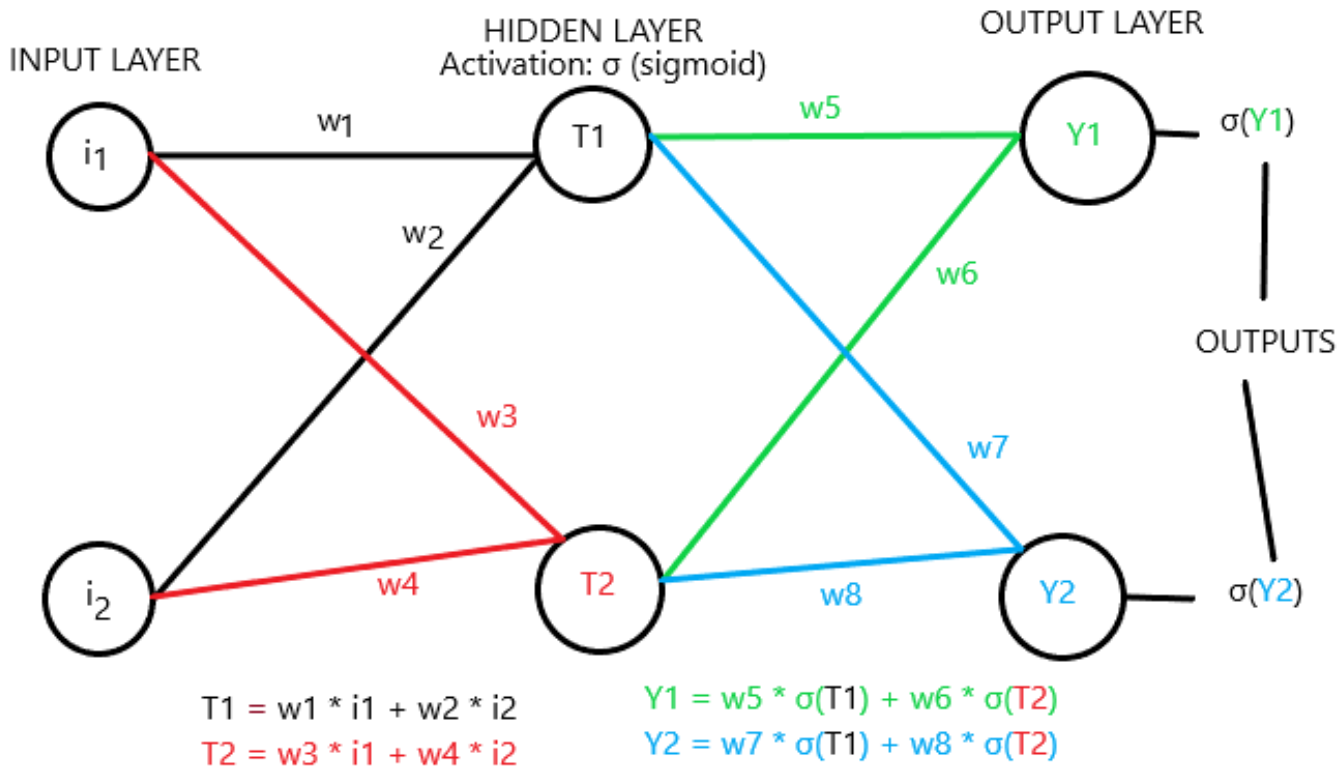


Figure: Sample NN with two inputs, one hidden layer with two nodes, and two outputs.

Matrix Backpropagation

Suppose we have initialized our weights $w_1, w_2, \ldots, w_8$ at random and passed our data through the NN using the described process. Suppose the data points rendered by our NN are given by the vector $Y = [\sigma(Y_1), \sigma(Y_2)]^\top$, and the expected data values $\overline{Y_1}$ and $\overline{Y_2}$ are given by the vector $\overline{Y} = [\overline{Y_1}, \overline{Y_2}]^\top$. We then compute our error according to some loss function, suppose the mean squared error

$$E = \frac{1}{2}\|Y - \overline{Y}\|^2.$$

Matrix Backpropagation

We would like the minimize the error with respect to a given weight. Suppose we want to minimize error with respect to weight w_5. To do so, we perform a gradient descent! Letting $\sigma(Y_1) = \Sigma_1$ and $\sigma(Y_2) = \Sigma_2$ and $\sigma(T_1) = \sigma_1$ and $\sigma(T_2) = \sigma_2$,

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial \Sigma_1} \frac{\partial \Sigma_1}{\partial \sigma_1} \frac{\partial \sigma_1}{\partial w_5}.$$

We know each of those derivatives!

$$\frac{\partial E}{\partial \Sigma_1} = \Sigma_1 - \overline{Y}_1$$
$$\frac{\partial \Sigma_1}{\partial \sigma_1} = \Sigma_1(1 - \Sigma_1)$$
$$\frac{\partial \sigma_1}{\partial w_5} = \sigma_1$$

Matrix Backpropagation

Using all these values, we use the gradient descent algorithm to update w_5 to a new weight w_5':

$$w_5' = w_5 - \gamma \frac{\partial E}{\partial w_5}.$$

After repeating this process for all 8 weights, we run the NN again (the second epoch), find the errors, and update weights again. We proceed for however many epochs we wish. Once we run the last epoch, we have "trained" the neural net.

Comments on Step Sizes

The step size sequence should generally follow the below rules:

- After one iteration, if the value of the function increases, the step size was too large.
- If the function value decreases, increase the step size.

The process of finding a good step size sequence can be a delicate balancing act.

Momentum

Gradient descents work poorly on surfaces with higher curvatures. Consider the rolling ball analogy:

- Imagine a ball rolling down a U-shaped ramp
- If the ramp is really steep, the ball will continue past the lowest point and rapidly oscillate up and down the ramp before settling at the bottom
- If the ramp is not very steep, the ball will gently slide to a rest at the bottom of the ramp

To account for the "momentum" of the ball, one might include an error term. In the context of gradient descents, we have:

$$x_i = x_{i-1} - \gamma_i (\nabla f(x_i))^\top + \alpha \Delta x_i,$$

with

$$\Delta x_i := x_i - x_{i-1}$$

for some $\alpha \in [0, 1]$. The momentum term is particularly useful when the gradient can only be approximated.

Stochastic Gradient Descents (SGD)

Oftentimes computing function gradients can be too computationally taxing to be viable. In such cases, we must settle for approximations. In such cases, we use SGDs. Consider a data set containing N values. Suppose that the parameter we wish to optimize over is θ. We consider the loss function L defined by

$$L(\theta) := \sum_{n=1}^{N} L_n(\theta),$$

and update the gradient descent algorithm accordingly:

$$x_{i+1} = x_i - \gamma_i \sum_{n=1}^{N} (\nabla L_n(\theta_i))^\top.$$

The benefit is that we can choose the batch size N to be however small as we wish, increasing computational efficiency at the cost of accuracy. Therefore, SGD is useful when tackling large-scale ML projects.

Sample Keras NN

```
import numpy as np
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense

# load the dataset

f = lambda x : x*x #Defining the function
LOWER_LIMIT = -10 # Inclusive limit
UPPER_LIMIT = 10 # Exclusive limit

X = np.random.randint( low=LOWER_LIMIT , high=UPPER_LIMIT , size=( 10000 , 1 ) )

# Convert the data type to float32
X = X.astype( np.float32 )

y_function = np.vectorize( f )
Y = y_function( X ) #Creating vector of y values
```

Sample Keras NN

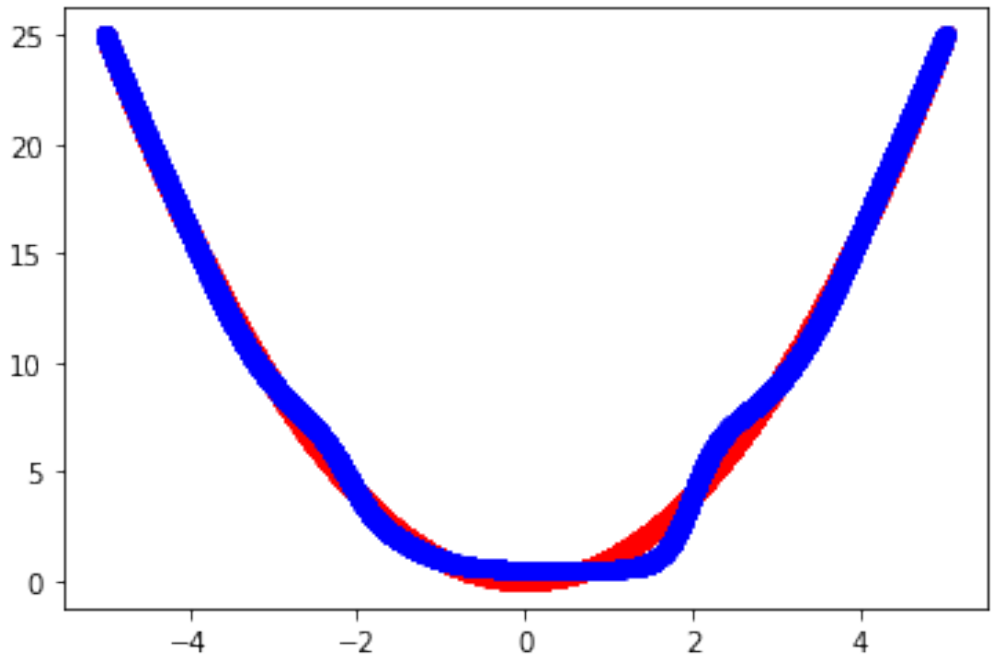
```
# define the keras model
model = Sequential()
model.add(Dense(32, input_dim=1, activation='sigmoid')) #NOTE 1: Sigmoid activation
model.add(Dense(64, activation='sigmoid'))
model.add(Dense(1))
# compile the keras model
model.compile(loss='mean_squared_error', optimizer='SGD', metrics=['mae']) #NOTE 2: Different Metric 'mae' instead of
#'accuracy'. See article 1.

# fit the keras model on the dataset
model.fit(X, Y, epochs=500, batch_size=200)
```

Sample Keras NN

```
#Evaluating the model
LOW = -10
HIGH = 10
test_size = 10000

sample_X = np.random.randint(LOW,HIGH,test_size, dtype = int)  #Creating the training inpute
sample_X.reshape(-1,1)  #This is just making sure things are formatted nicely, ignore this
sample_Y = np.vectorize(f)(sample_X) #True 'desired' outputs

predictions = model.predict( sample_X ) #Making predictions with the model
rounded_pred = np.round( predictions )
rounded_pred = rounded_pred.astype(int)   #Rounding things and converting to a nice datatype
```

Visual

Accuracy: 95.06%



References I

Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.