
ML Cheatsheet Documentation

Team

Jun 01, 2022

1	Linear Regression	3
2	Gradient Descent	21
3	Logistic Regression	25
4	Glossary	39
5	Calculus	45
6	Linear Algebra	57
7	Probability	67
8	Statistics	69
9	Notation	71
10	Concepts	75
11	Forwardpropagation	81
12	Backpropagation	91
13	Activation Functions	97
14	Layers	105
15	Loss Functions	117
16	Optimizers	121
17	Regularization	127
18	Architectures	137
19	Classification Algorithms	151
20	Clustering Algorithms	161

21 Regression Algorithms	163
22 Reinforcement Learning	165
23 Datasets	169
24 Libraries	185
25 Papers	215
26 Other Content	221
27 Contribute	227

Brief visual explanations of machine learning concepts with diagrams, code examples and links to resources for learning more.

Warning: This document is under early stage development. If you find errors, please raise an [issue](#) or [contribute](#) a better definition!

CHAPTER 1

Linear Regression

- *Introduction*
- *Simple regression*
 - *Making predictions*
 - *Cost function*
 - *Gradient descent*
 - *Training*
 - *Model evaluation*
 - *Summary*
- *Multivariable regression*
 - *Growing complexity*
 - *Normalization*
 - *Making predictions*
 - *Initialize weights*
 - *Cost function*
 - *Gradient descent*
 - *Simplifying with matrices*
 - *Bias term*
 - *Model evaluation*

1.1 Introduction

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. It's used to predict values within a continuous range, (e.g. sales, price) rather than trying to classify them into categories (e.g. cat, dog). There are two main types:

Simple regression

Simple linear regression uses traditional slope-intercept form, where m and b are the variables our algorithm will try to “learn” to produce the most accurate predictions. x represents our input data and y represents our prediction.

$$y = mx + b$$

Multivariable regression

A more complex, multi-variable linear equation might look like this, where w represents the coefficients, or weights, our model will try to learn.

$$f(x, y, z) = w_1x + w_2y + w_3z$$

The variables x, y, z represent the attributes, or distinct pieces of information, we have about each observation. For sales predictions, these attributes might include a company's advertising spend on radio, TV, and newspapers.

$$Sales = w_1Radio + w_2TV + w_3News$$

1.2 Simple regression

Let's say we are given a [dataset](#) with the following columns (features): how much a company spends on Radio advertising each year and its annual Sales in terms of units sold. We are trying to develop an equation that will let us to predict units sold based on how much a company spends on radio advertising. The rows (observations) represent companies.

Company	Radio (\$)	Sales
Amazon	37.8	22.1
Google	39.3	10.4
Facebook	45.9	18.3
Apple	41.3	18.5

1.2.1 Making predictions

Our prediction function outputs an estimate of sales given a company's radio advertising spend and our current values for *Weight* and *Bias*.

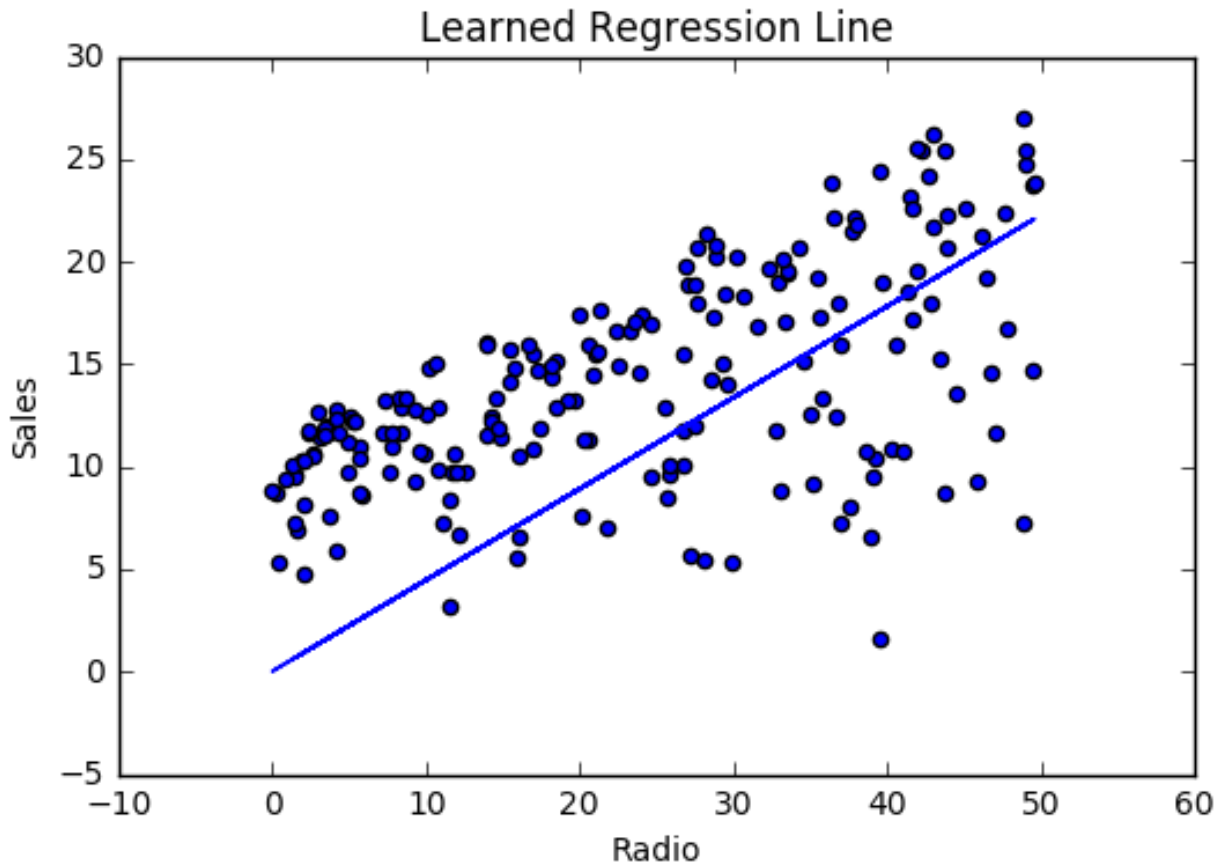
$$Sales = Weight \cdot Radio + Bias$$

Weight the coefficient for the Radio independent variable. In machine learning we call coefficients *weights*.

Radio the independent variable. In machine learning we call these variables *features*.

Bias the intercept where our line intercepts the y-axis. In machine learning we can call intercepts *bias*. Bias offsets all predictions that we make.

Our algorithm will try to *learn* the correct values for Weight and Bias. By the end of our training, our equation will approximate the *line of best fit*.



Code

```
def predict_sales(radio, weight, bias):  
    return weight*radio + bias
```

1.2.2 Cost function

The prediction function is nice, but for our purposes we don't really need it. What we need is a *cost function* so we can start optimizing our weights.

Let's use *MSE (L2)* as our cost function. MSE measures the average squared difference between an observation's actual and predicted values. The output is a single number representing the cost, or score, associated with our current set of weights. Our goal is to minimize MSE to improve the accuracy of our model.

Math

Given our simple linear equation $y = mx + b$, we can calculate MSE as:

$$MSE = \frac{1}{N} \sum_{i=1}^n (y_i - (mx_i + b))^2$$

Note:

- N is the total number of observations (data points)
 - $\frac{1}{N} \sum_{i=1}^n$ is the mean
 - y_i is the actual value of an observation and $mx_i + b$ is our prediction
-

Code

```
def cost_function(radio, sales, weight, bias):
    companies = len(radio)
    total_error = 0.0
    for i in range(companies):
        total_error += (sales[i] - (weight*radio[i] + bias))**2
    return total_error / companies
```

1.2.3 Gradient descent

To minimize MSE we use *Gradient Descent* to calculate the gradient of our cost function. Gradient descent consists of looking at the error that our weight currently gives us, using the derivative of the cost function to find the gradient (The slope of the cost function using our current weight), and then changing our weight to move in the direction opposite of the gradient. We need to move in the opposite direction of the gradient since the gradient points up the slope instead of down it, so we move in the opposite direction to try to decrease our error.

Math

There are two *parameters* (coefficients) in our cost function we can control: weight m and bias b . Since we need to consider the impact each one has on the final prediction, we use partial derivatives. To find the partial derivatives, we use the *Chain rule*. We need the chain rule because $(y - (mx + b))^2$ is really 2 nested functions: the inner function $y - (mx + b)$ and the outer function x^2 .

Returning to our cost function:

$$f(m, b) = \frac{1}{N} \sum_{i=1}^n (y_i - (mx_i + b))^2$$

Using the following:

$$(y_i - (mx_i + b))^2 = A(B(m, b))$$

We can split the derivative into

$$\begin{aligned} A(x) &= x^2 \\ \frac{df}{dx} &= A'(x) = 2x \end{aligned}$$

and

$$\begin{aligned} B(m, b) &= y_i - (mx_i + b) = y_i - mx_i - b \\ \frac{dx}{dm} &= B'(m) = 0 - x_i - 0 = -x_i \\ \frac{dx}{db} &= B'(b) = 0 - 0 - 1 = -1 \end{aligned}$$

And then using the *Chain rule* which states:

$$\begin{aligned}\frac{df}{dm} &= \frac{df}{dx} \frac{dx}{dm} \\ \frac{df}{db} &= \frac{df}{dx} \frac{dx}{db}\end{aligned}$$

We then plug in each of the parts to get the following derivatives

$$\begin{aligned}\frac{df}{dm} &= A'(B(m, f))B'(m) = 2(y_i - (mx_i + b)) \cdot -x_i \\ \frac{df}{db} &= A'(B(m, f))B'(b) = 2(y_i - (mx_i + b)) \cdot -1\end{aligned}$$

We can calculate the gradient of this cost function as:

$$\begin{aligned}f'(m, b) &= \begin{bmatrix} \frac{df}{dm} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N} \sum -x_i \cdot 2(y_i - (mx_i + b)) \\ \frac{1}{N} \sum -1 \cdot 2(y_i - (mx_i + b)) \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{N} \sum -2x_i(y_i - (mx_i + b)) \\ \frac{1}{N} \sum -2(y_i - (mx_i + b)) \end{bmatrix}\end{aligned}\tag{1.1}$$

Code

To solve for the gradient, we iterate through our data points using our new weight and bias values and take the average of the partial derivatives. The resulting gradient tells us the slope of our cost function at our current position (i.e. weight and bias) and the direction we should update to reduce our cost function (we move in the direction opposite the gradient). The size of our update is controlled by the *learning rate*.

```
def update_weights(radio, sales, weight, bias, learning_rate):
    weight_deriv = 0
    bias_deriv = 0
    companies = len(radio)

    for i in range(companies):
        # Calculate partial derivatives
        # -2x(y - (mx + b))
        weight_deriv += -2*radio[i] * (sales[i] - (weight*radio[i] + bias))

        # -2(y - (mx + b))
        bias_deriv += -2*(sales[i] - (weight*radio[i] + bias))

    # We subtract because the derivatives point in direction of steepest ascent
    weight -= (weight_deriv / companies) * learning_rate
    bias -= (bias_deriv / companies) * learning_rate

    return weight, bias
```

1.2.4 Training

Training a model is the process of iteratively improving your prediction equation by looping through the dataset multiple times, each time updating the weight and bias values in the direction indicated by the slope of the cost function (gradient). Training is complete when we reach an acceptable error threshold, or when subsequent training iterations fail to reduce our cost.

Before training we need to initialize our weights (set default values), set our *hyperparameters* (learning rate and number of iterations), and prepare to log our progress over each iteration.

Code

```
def train(radio, sales, weight, bias, learning_rate, iters):
    cost_history = []

    for i in range(iters):
        weight, bias = update_weights(radio, sales, weight, bias, learning_rate)

        #Calculate cost for auditing purposes
        cost = cost_function(radio, sales, weight, bias)
        cost_history.append(cost)

        # Log Progress
        if i % 10 == 0:
            print "iter={:d}    weight={:.2f}    bias={:.4f}    cost={:.2f}".format(i,
↪weight, bias, cost)

    return weight, bias, cost_history
```

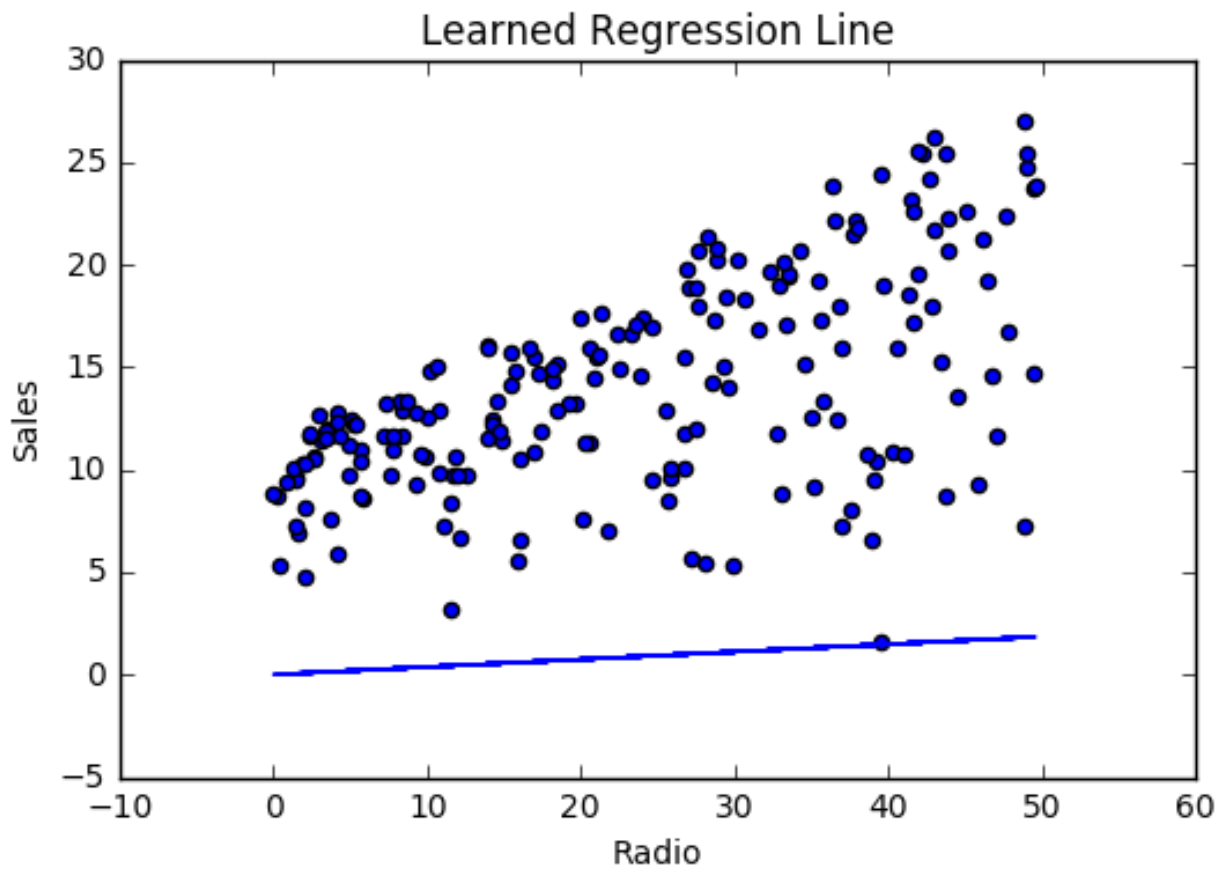
1.2.5 Model evaluation

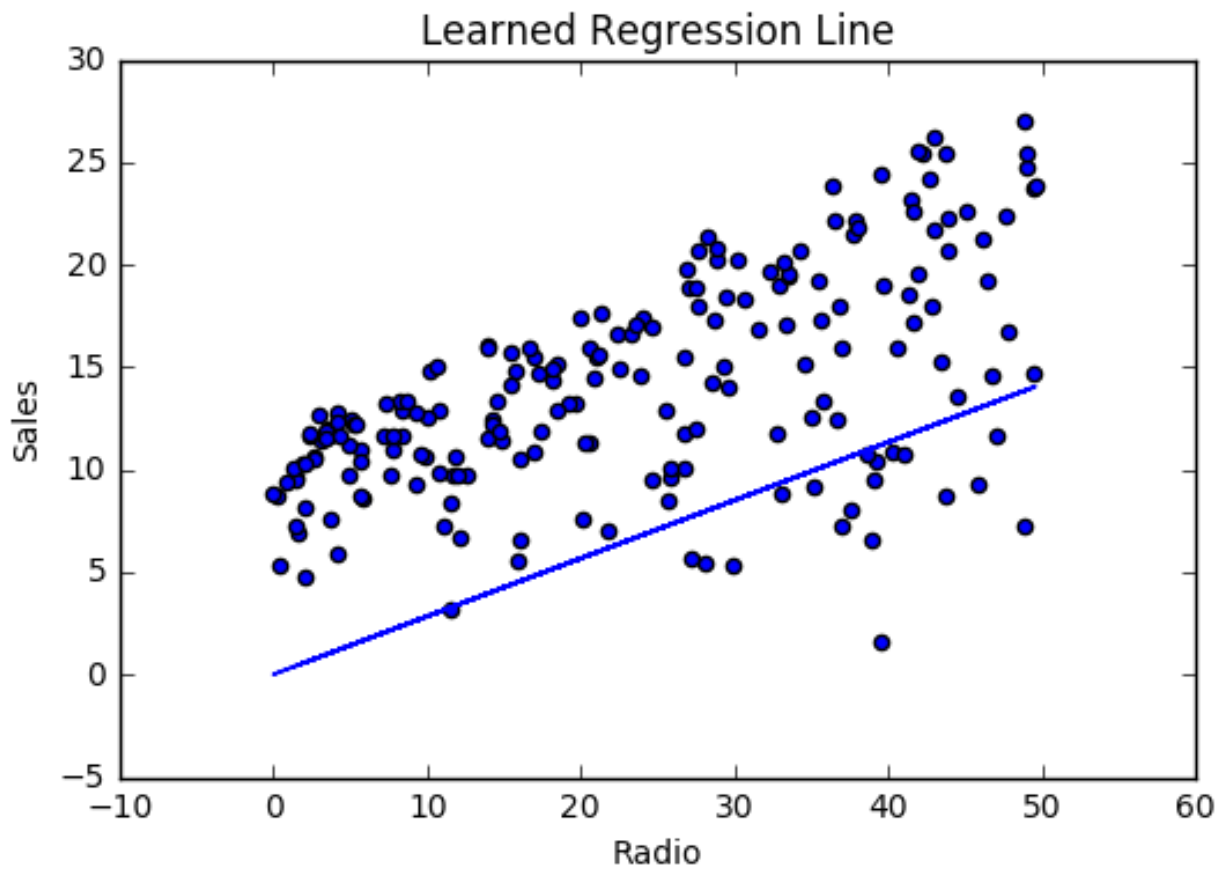
If our model is working, we should see our cost decrease after every iteration.

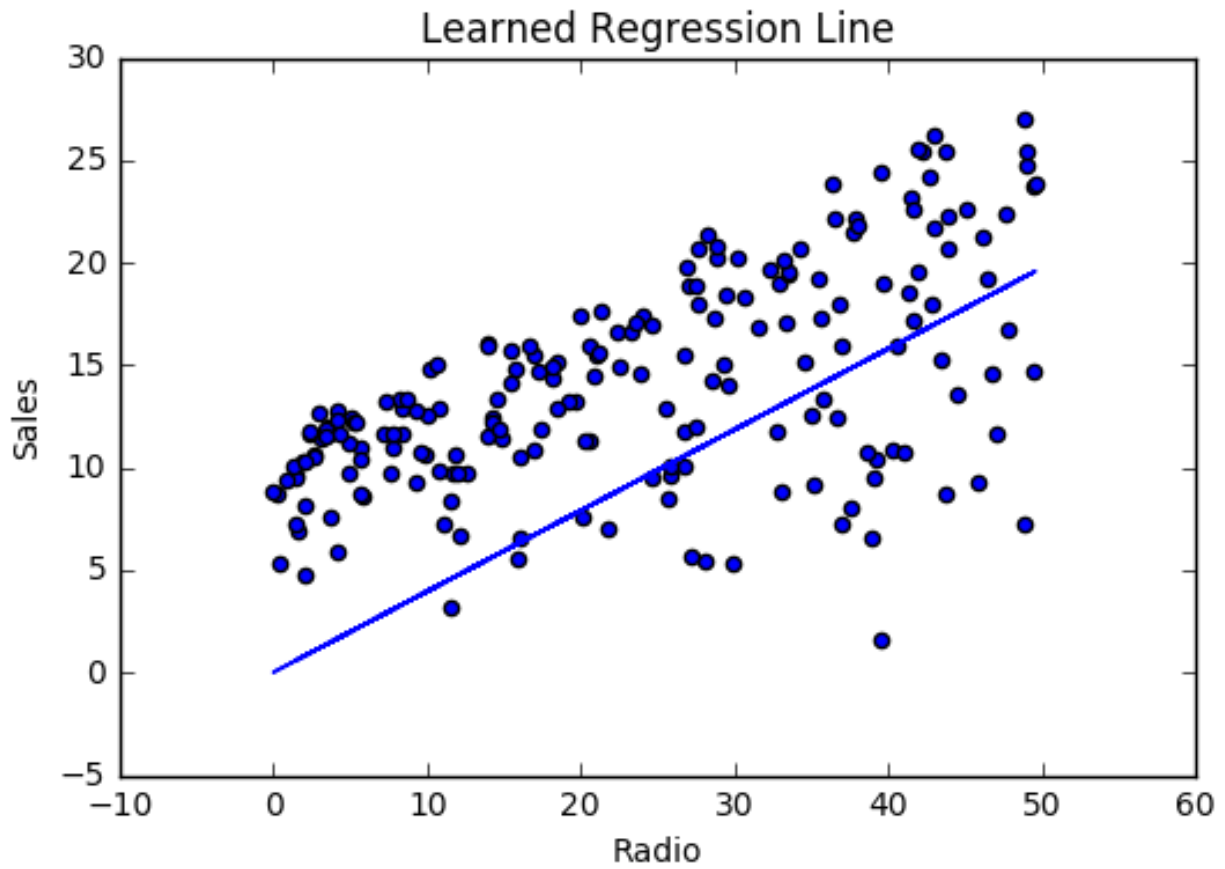
Logging

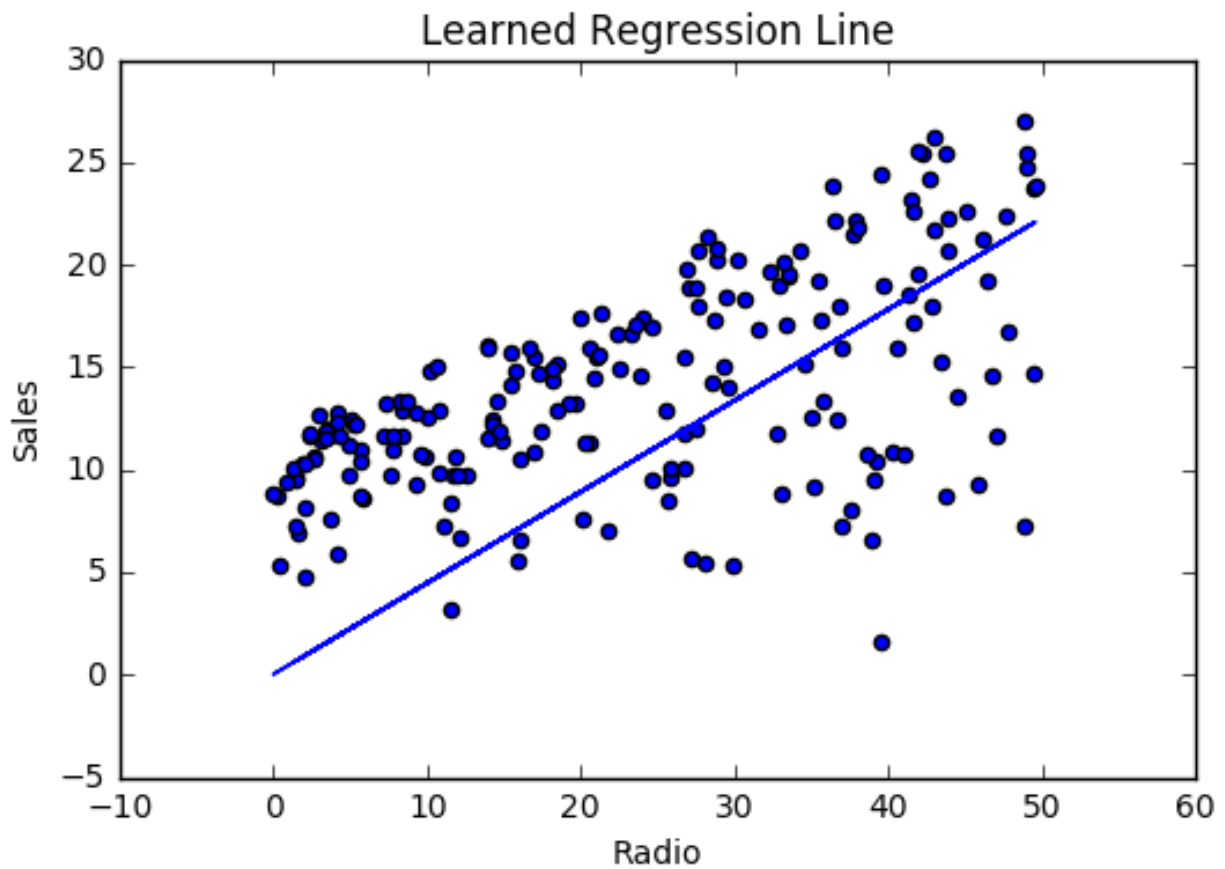
iter=1	weight=.03	bias=.0014	cost=197.25
iter=10	weight=.28	bias=.0116	cost=74.65
iter=20	weight=.39	bias=.0177	cost=49.48
iter=30	weight=.44	bias=.0219	cost=44.31
iter=30	weight=.46	bias=.0249	cost=43.28

Visualizing

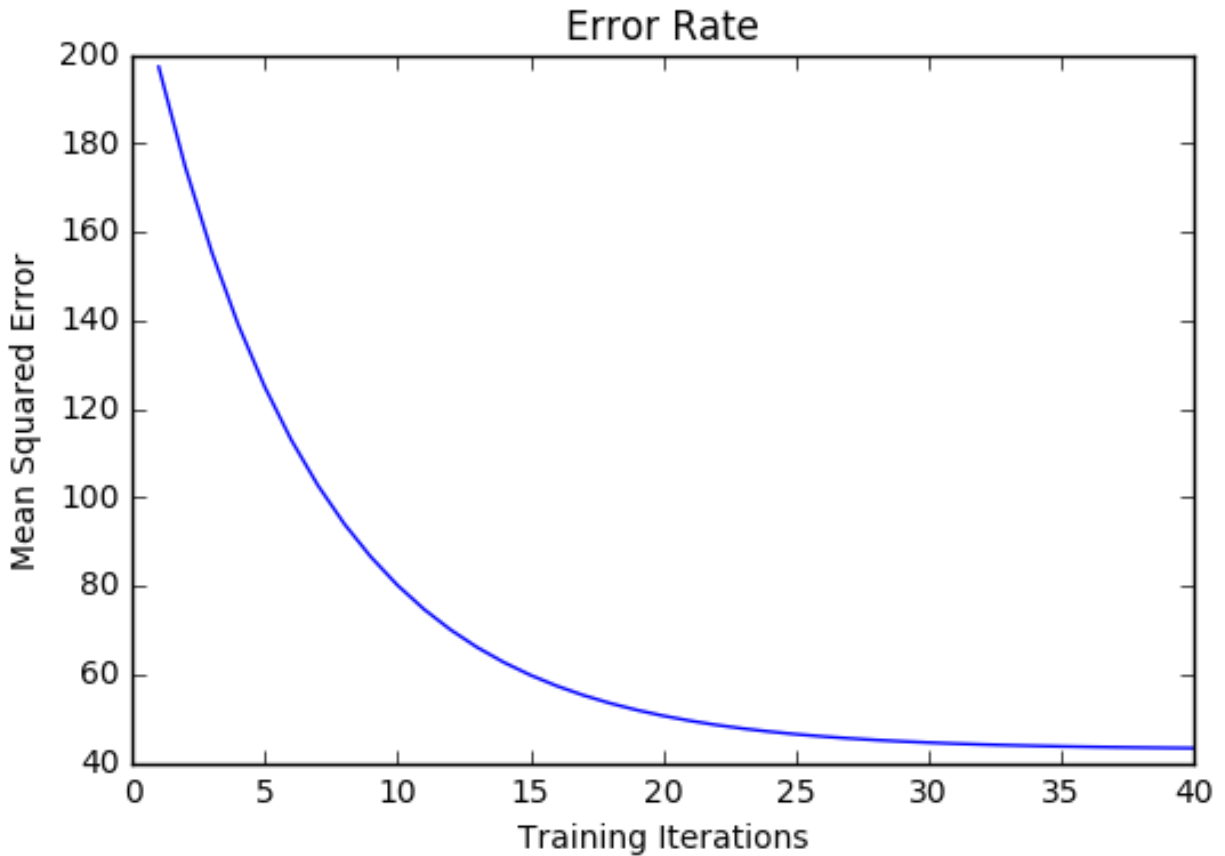








Cost history



1.2.6 Summary

By learning the best values for weight (.46) and bias (.25), we now have an equation that predicts future sales based on radio advertising investment.

$$Sales = .46Radio + .25$$

How would our model perform in the real world? I'll let you think about it :)

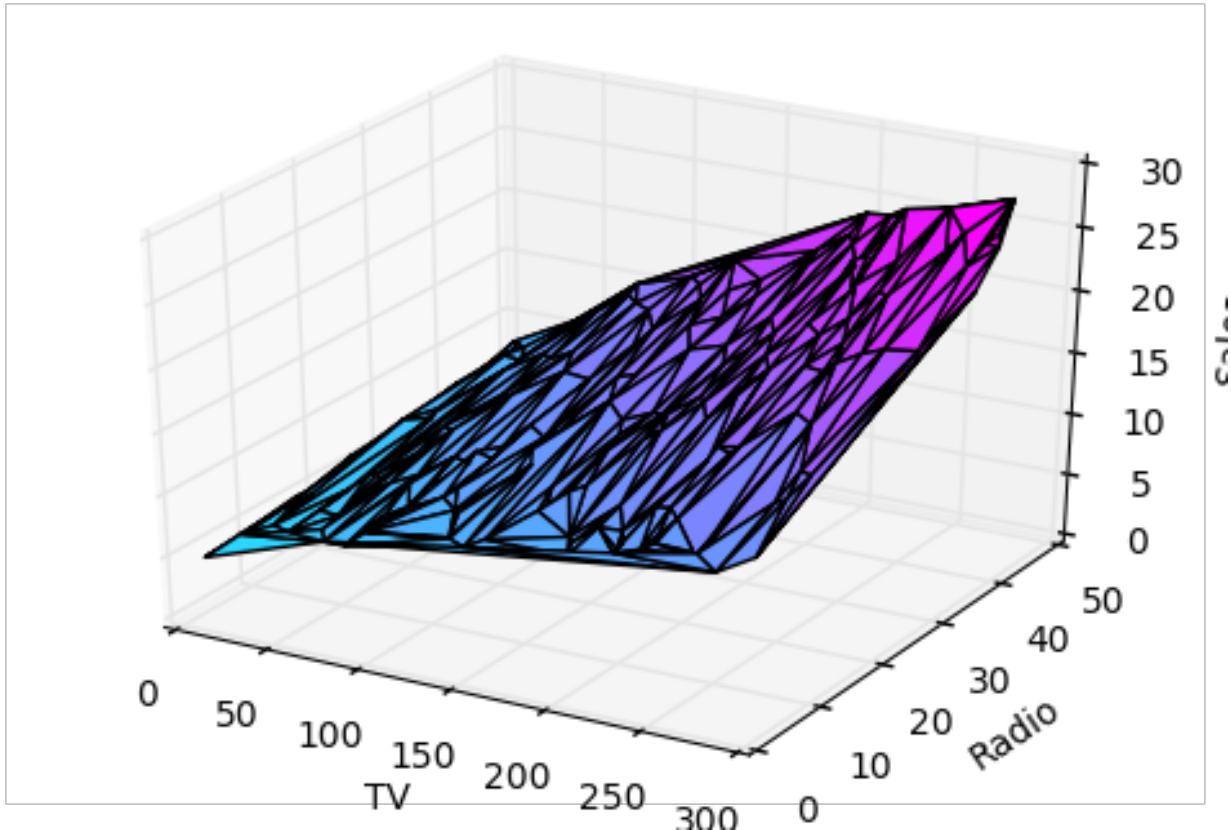
1.3 Multivariable regression

Let's say we are given [data](#) on TV, radio, and newspaper advertising spend for a list of companies, and our goal is to predict sales in terms of units sold.

Company	TV	Radio	News	Units
Amazon	230.1	37.8	69.1	22.1
Google	44.5	39.3	23.1	10.4
Facebook	17.2	45.9	34.7	18.3
Apple	151.5	41.3	13.2	18.5

1.3.1 Growing complexity

As the number of features grows, the complexity of our model increases and it becomes increasingly difficult to visualize, or even comprehend, our data.



One solution is to break the data apart and compare 1-2 features at a time. In this example we explore how Radio and TV investment impacts Sales.

1.3.2 Normalization

As the number of features grows, calculating gradient takes longer to compute. We can speed this up by “normalizing” our input data to ensure all values are within the same range. This is especially important for datasets with high standard deviations or differences in the ranges of the attributes. Our goal now will be to normalize our features so they are all in the range -1 to 1.

Code

```
For each feature column {  
    #1 Subtract the mean of the column (mean normalization)  
    #2 Divide by the range of the column (feature scaling)  
}
```

Our input is a 200 x 3 matrix containing TV, Radio, and Newspaper data. Our output is a normalized matrix of the same shape with all values between -1 and 1.

```
def normalize(features):
    """
    features      -   (200, 3)
    features.T    -   (3, 200)

    We transpose the input matrix, swapping
    cols and rows to make vector math easier
    """

    for feature in features.T:
        fmean = np.mean(feature)
        frange = np.amax(feature) - np.amin(feature)

        #Vector Subtraction
        feature -= fmean

        #Vector Division
        feature /= frange

    return features
```

Note: Matrix math. Before we continue, it's important to understand basic *Linear Algebra* concepts as well as numpy functions like `numpy.dot()`.

1.3.3 Making predictions

Our predict function outputs an estimate of sales given our current weights (coefficients) and a company's TV, radio, and newspaper spend. Our model will try to identify weight values that most reduce our cost function.

$$Sales = W_1TV + W_2Radio + W_3Newspaper$$

```
def predict(features, weights):
    """
    features - (200, 3)
    weights - (3, 1)
    predictions - (200,1)
    """
    predictions = np.dot(features, weights)
    return predictions
```

1.3.4 Initialize weights

```
W1 = 0.0
W2 = 0.0
W3 = 0.0
weights = np.array([
    [W1],
    [W2],
    [W3]
])
```

1.3.5 Cost function

Now we need a cost function to audit how our model is performing. The math is the same, except we swap the $mx + b$ expression for $W_1x_1 + W_2x_2 + W_3x_3$. We also divide the expression by 2 to make derivative calculations simpler.

$$MSE = \frac{1}{2N} \sum_{i=1}^n (y_i - (W_1x_1 + W_2x_2 + W_3x_3))^2$$

```
def cost_function(features, targets, weights):
    """
    features: (200, 3)
    targets: (200, 1)
    weights: (3, 1)
    returns average squared error among predictions
    """
    N = len(targets)

    predictions = predict(features, weights)

    # Matrix math lets us do this without looping
    sq_error = (predictions - targets)**2

    # Return average squared error among predictions
    return 1.0 / (2*N) * sq_error.sum()
```

1.3.6 Gradient descent

Again using the *Chain rule* we can compute the gradient—a vector of partial derivatives describing the slope of the cost function for each weight.

$$\begin{aligned} f'(W_1) &= -x_1(y - (W_1x_1 + W_2x_2 + W_3x_3)) \\ f'(W_2) &= -x_2(y - (W_1x_1 + W_2x_2 + W_3x_3)) \\ f'(W_3) &= -x_3(y - (W_1x_1 + W_2x_2 + W_3x_3)) \end{aligned} \tag{1.3}$$

```
def update_weights(features, targets, weights, lr):
    """
    Features: (200, 3)
    Targets: (200, 1)
    Weights: (3, 1)
    """
    predictions = predict(features, weights)

    # Extract our features
    x1 = features[:, 0]
    x2 = features[:, 1]
    x3 = features[:, 2]

    # Use dot product to calculate the derivative for each weight
    d_w1 = -x1.dot(targets - predictions)
    d_w2 = -x2.dot(targets - predictions)
    d_w3 = -x3.dot(targets - predictions)

    # Multiply the mean derivative by the learning rate
```

(continues on next page)

(continued from previous page)

```

    # and subtract from our weights (remember gradient points in direction of ↘
    ↪steepest ASCENT)
    weights[0][0] -= (lr * np.mean(d_w1))
    weights[1][0] -= (lr * np.mean(d_w2))
    weights[2][0] -= (lr * np.mean(d_w3))

    return weights

```

And that's it! Multivariate linear regression.

1.3.7 Simplifying with matrices

The gradient descent code above has a lot of duplication. Can we improve it somehow? One way to refactor would be to loop through our features and weights—allowing our function to handle any number of features. However there is another even better technique: *vectorized gradient descent*.

Math

We use the same formula as above, but instead of operating on a single feature at a time, we use matrix multiplication to operate on all features and weights simultaneously. We replace the x_i terms with a single feature matrix X .

$$\text{gradient} = -X(\text{targets} - \text{predictions})$$

Code

```

X = [
    [x1, x2, x3]
    [x1, x2, x3]
    .
    .
    .
    [x1, x2, x3]
]

targets = [
    [1],
    [2],
    [3]
]

def update_weights_vectorized(X, targets, weights, lr):
    **
    gradient = X.T * (predictions - targets) / N
    X: (200, 3)
    Targets: (200, 1)
    Weights: (3, 1)
    **
    companies = len(X)

    #1 - Get Predictions
    predictions = predict(X, weights)

```

(continues on next page)

(continued from previous page)

```
#2 - Calculate error/loss
error = targets - predictions

#3 Transpose features from (200, 3) to (3, 200)
# So we can multiply w the (200,1) error matrix.
# Returns a (3,1) matrix holding 3 partial derivatives --
# one for each feature -- representing the aggregate
# slope of the cost function across all observations
gradient = np.dot(-X.T, error)

#4 Take the average error derivative for each feature
gradient /= companies

#5 - Multiply the gradient by our learning rate
gradient *= lr

#6 - Subtract from our weights to minimize cost
weights -= gradient

return weights
```

1.3.8 Bias term

Our train function is the same as for simple linear regression, however we're going to make one final tweak before running: add a *bias term* to our feature matrix.

In our example, it's very unlikely that sales would be zero if companies stopped advertising. Possible reasons for this might include past advertising, existing customer relationships, retail locations, and salespeople. A bias term will help us capture this base case.

Code

Below we add a constant 1 to our features matrix. By setting this value to 1, it turns our bias term into a constant.

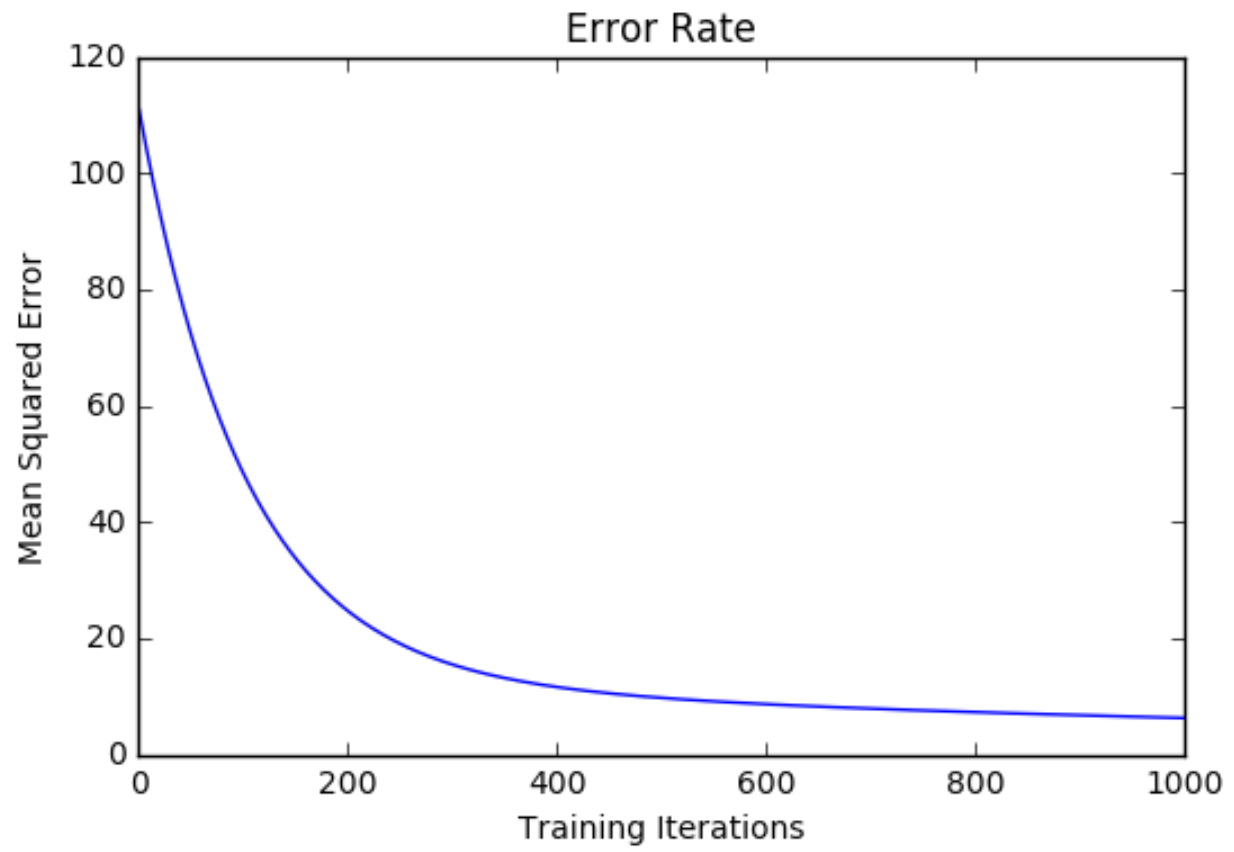
```
bias = np.ones(shape=(len(features),1))
features = np.append(bias, features, axis=1)
```

1.3.9 Model evaluation

After training our model through 1000 iterations with a learning rate of .0005, we finally arrive at a set of weights we can use to make predictions:

$$Sales = 4.7TV + 3.5Radio + .81Newspaper + 13.9$$

Our MSE cost dropped from 110.86 to 6.25.

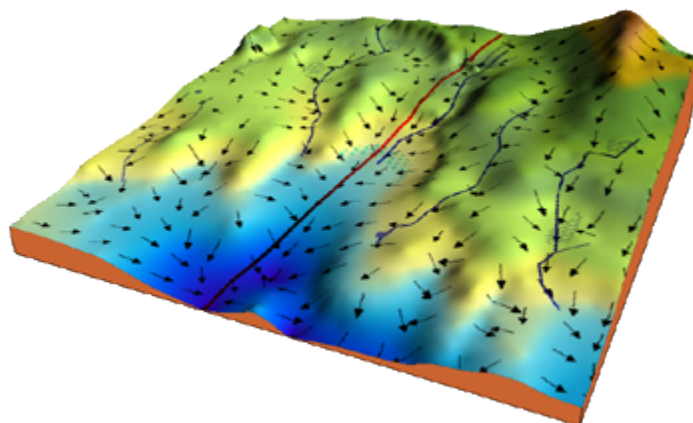


References

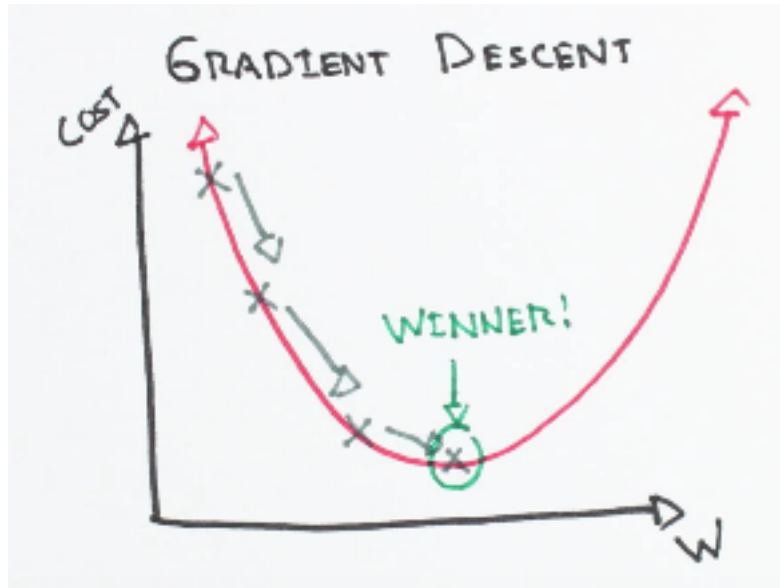
Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the *parameters* of our model. Parameters refer to coefficients in *Linear Regression* and *weights* in neural networks.

2.1 Introduction

Consider the 3-dimensional graph below in the context of a cost function. Our goal is to move from the mountain in the top right corner (high cost) to the dark blue sea in the bottom left (low cost). The arrows represent the direction of steepest descent (negative gradient) from any given point—the direction that decreases the cost function as quickly as possible. [Source](#)



Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill—a local minimum. [image source](#).



2.2 Learning rate

The size of these steps is called the *learning rate*. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

2.3 Cost function

A *Loss Functions* tells us “how good” our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.

2.4 Step-by-step

Now let's run gradient descent using our new cost function. There are two parameters in our cost function we can control: m (weight) and b (bias). Since we need to consider the impact each one has on the final prediction, we need to use partial derivatives. We calculate the partial derivatives of the cost function with respect to each parameter and store the results in a gradient.

Math

Given the cost function:

$$f(m, b) = \frac{1}{N} \sum_{i=1}^n (y_i - (mx_i + b))^2$$

The gradient can be calculated as:

$$f'(m, b) = \begin{bmatrix} \frac{df}{dm} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N} \sum -2x_i(y_i - (mx_i + b)) \\ \frac{1}{N} \sum -2(y_i - (mx_i + b)) \end{bmatrix}$$

To solve for the gradient, we iterate through our data points using our new m and b values and compute the partial derivatives. This new gradient tells us the slope of our cost function at our current position (current parameter values) and the direction we should move to update our parameters. The size of our update is controlled by the learning rate.

Code

```
def update_weights(m, b, X, Y, learning_rate):
    m_deriv = 0
    b_deriv = 0
    N = len(X)
    for i in range(N):
        # Calculate partial derivatives
        # -2x(y - (mx + b))
        m_deriv += -2*X[i] * (Y[i] - (m*X[i] + b))

        # -2(y - (mx + b))
        b_deriv += -2*(Y[i] - (m*X[i] + b))

    # We subtract because the derivatives point in direction of steepest ascent
    m -= (m_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m, b
```

References

- *Introduction*
 - *Comparison to linear regression*
 - *Types of logistic regression*
- *Binary logistic regression*
 - *Sigmoid activation*
 - *Decision boundary*
 - *Making predictions*
 - *Cost function*
 - *Gradient descent*
 - *Mapping probabilities to classes*
 - *Training*
 - *Model evaluation*
- *Multiclass logistic regression*
 - *Procedure*
 - *Softmax activation*
 - *Scikit-Learn example*

3.1 Introduction

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Unlike linear regression which outputs continuous number values, logistic regression transforms its output using the logistic sigmoid

function to return a probability value which can then be mapped to two or more discrete classes.

3.1.1 Comparison to linear regression

Given data on time spent studying and exam scores. *Linear Regression* and logistic regression can predict different things:

- **Linear Regression** could help us predict the student's test score on a scale of 0 - 100. Linear regression predictions are continuous (numbers in a range).
- **Logistic Regression** could help use predict whether the student passed or failed. Logistic regression predictions are discrete (only specific values or categories are allowed). We can also view probability scores underlying the model's classifications.

3.1.2 Types of logistic regression

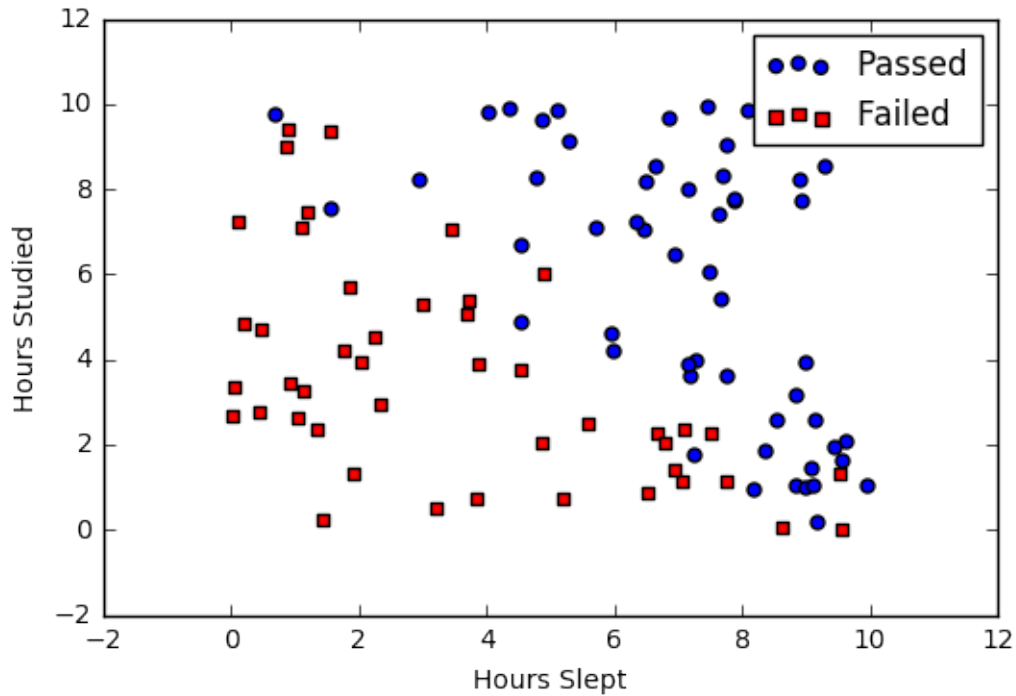
- Binary (Pass/Fail)
- Multi (Cats, Dogs, Sheep)
- Ordinal (Low, Medium, High)

3.2 Binary logistic regression

Say we're given *data* on student exam results and our goal is to predict whether a student will pass or fail based on number of hours slept and hours spent studying. We have two features (hours slept, hours studied) and two classes: passed (1) and failed (0).

Studied	Slept	Passed
4.85	9.63	1
8.62	3.23	0
5.43	8.23	1
9.21	6.34	0

Graphically we could represent our data with a scatter plot.



3.2.1 Sigmoid activation

In order to map predicted values to probabilities, we use the *sigmoid* function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.

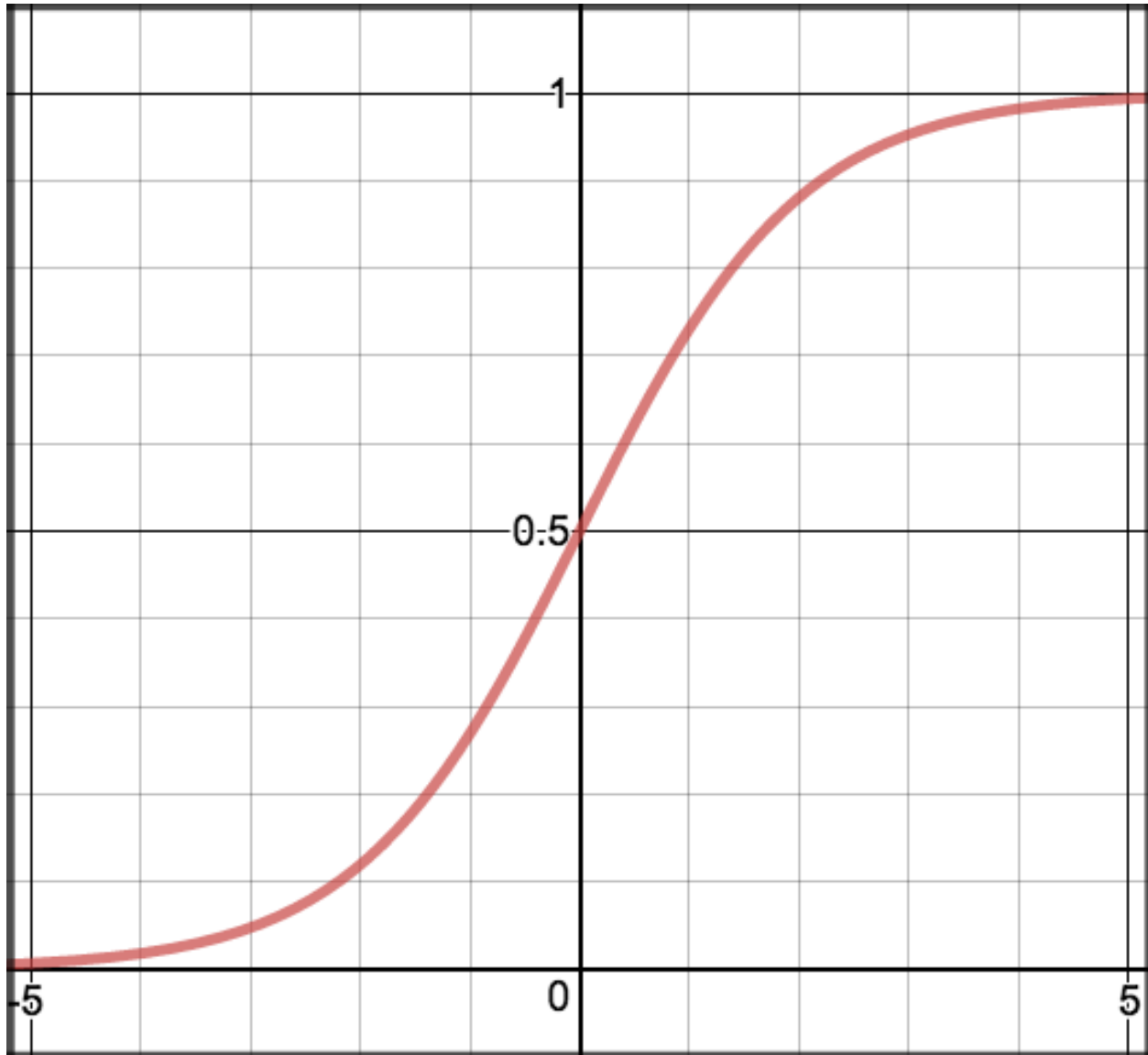
Math

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

Note:

- $s(z)$ = output between 0 and 1 (probability estimate)
 - z = input to the function (your algorithm's prediction e.g. $mx + b$)
 - e = base of natural log
-

Graph



Code

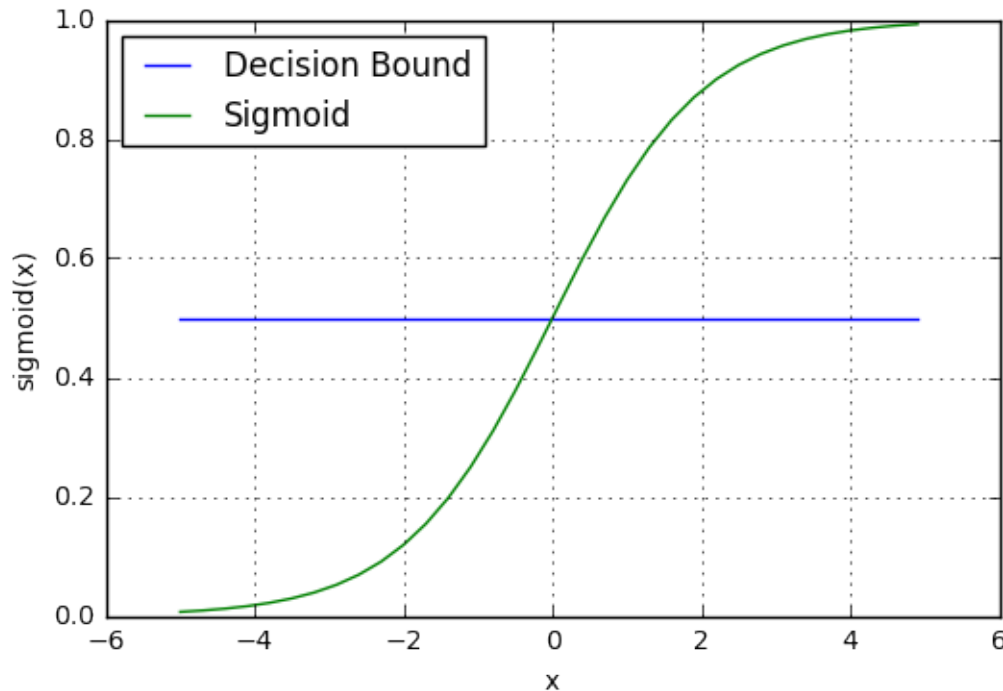
```
def sigmoid(z):  
    return 1.0 / (1 + np.exp(-z))
```

3.2.2 Decision boundary

Our current prediction function returns a probability score between 0 and 1. In order to map this to a discrete class (true/false, cat/dog), we select a threshold value or tipping point above which we will classify values into class 1 and below which we classify values into class 2.

$$\begin{aligned} p &\geq 0.5, \text{class} = 1 \\ p &< 0.5, \text{class} = 0 \end{aligned}$$

For example, if our threshold was .5 and our prediction function returned .7, we would classify this observation as positive. If our prediction was .2 we would classify the observation as negative. For logistic regression with multiple classes we could select the class with the highest predicted probability.



3.2.3 Making predictions

Using our knowledge of sigmoid functions and decision boundaries, we can now write a prediction function. A prediction function in logistic regression returns the probability of our observation being positive, True, or “Yes”. We call this class 1 and its notation is $P(class = 1)$. As the probability gets closer to 1, our model is more confident that the observation is in class 1.

Math

Let’s use the same *multiple linear regression* equation from our linear regression tutorial.

$$z = W_0 + W_1 \text{Studied} + W_2 \text{Slept}$$

This time however we will transform the output using the sigmoid function to return a probability value between 0 and 1.

$$P(class = 1) = \frac{1}{1 + e^{-z}}$$

If the model returns .4 it believes there is only a 40% chance of passing. If our decision boundary was .5, we would categorize this observation as “Fail.”

Code

We wrap the sigmoid function over the same prediction function we used in *multiple linear regression*

```
def predict(features, weights):  
    '''  
    Returns 1D array of probabilities  
    that the class label == 1  
    '''  
    z = np.dot(features, weights)  
    return sigmoid(z)
```

3.2.4 Cost function

Unfortunately we can't (or at least shouldn't) use the same cost function *MSE (L2)* as we did for linear regression. Why? There is a great math explanation in chapter 3 of Michael Neilson's deep learning book⁵, but for now I'll simply say it's because our prediction function is non-linear (due to sigmoid transform). Squaring this prediction as we do in MSE results in a non-convex function with many local minimums. If our cost function has many local minimums, gradient descent may not find the optimal global minimum.

Math

Instead of Mean Squared Error, we use a cost function called *Cross-Entropy*, also known as Log Loss. Cross-entropy loss can be divided into two separate cost functions: one for $y = 1$ and one for $y = 0$.

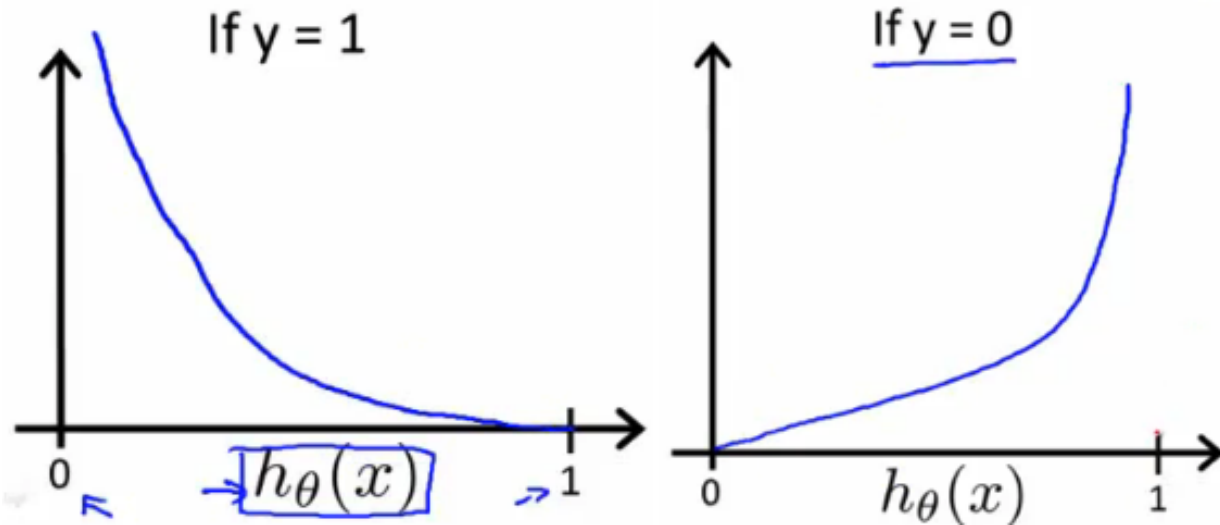
$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$$
$$\begin{aligned} \text{Cost}(h_{\theta}(x), y) &= -\log(h_{\theta}(x)) && \text{if } y = 1 \\ \text{Cost}(h_{\theta}(x), y) &= -\log(1 - h_{\theta}(x)) && \text{if } y = 0 \end{aligned}$$

The benefits of taking the logarithm reveal themselves when you look at the cost function graphs for $y=1$ and $y=0$. These smooth monotonic functions⁷ (always increasing or always decreasing) make it easy to calculate the gradient and minimize cost. Image from Andrew Ng's slides on logistic regression¹.

⁵ <http://neuralnetworksanddeeplearning.com/chap3.html>

⁷ https://en.wikipedia.org/wiki/Monotonic_function

¹ http://www.holehouse.org/mlclass/06_Logistic_Regression.html



The key thing to note is the cost function penalizes confident and wrong predictions more than it rewards confident and right predictions! The corollary is increasing prediction accuracy (closer to 0 or 1) has diminishing returns on reducing cost due to the logistic nature of our cost function.

Above functions compressed into one

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

Multiplying by y and $(1 - y)$ in the above equation is a sneaky trick that let's us use the same equation to solve for both $y=1$ and $y=0$ cases. If $y=0$, the first side cancels out. If $y=1$, the second side cancels out. In both cases we only perform the operation we need to perform.

Vectorized cost function

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m} \cdot (-y^T \log(h) - (1 - y)^T \log(1 - h))$$

Code

```
def cost_function(features, labels, weights):
    '''
    Using Mean Absolute Error

    Features: (100,3)
    Labels: (100,1)
    Weights: (3,1)
```

(continues on next page)

(continued from previous page)

```
Returns 1D matrix of predictions
Cost = (labels*log(predictions) + (1-labels)*log(1-predictions) ) / len(labels)
'''
observations = len(labels)

predictions = predict(features, weights)

#Take the error when label=1
class1_cost = -labels*np.log(predictions)

#Take the error when label=0
class2_cost = (1-labels)*np.log(1-predictions)

#Take the sum of both costs
cost = class1_cost - class2_cost

#Take the average cost
cost = cost.sum() / observations

return cost
```

3.2.5 Gradient descent

To minimize our cost, we use *Gradient Descent* just like before in *Linear Regression*. There are other more sophisticated optimization algorithms out there such as conjugate gradient like *BFGS*, but you don't have to worry about these. Machine learning libraries like Scikit-learn hide their implementations so you can focus on more interesting things!

Math

One of the neat properties of the sigmoid function is its derivative is easy to calculate. If you're curious, there is a good walk-through derivation on stack overflow⁶. Michael Neilson also covers the topic in chapter 3 of his book.

$$s'(z) = s(z)(1 - s(z)) \quad (3.1)$$

Which leads to an equally beautiful and convenient cost function derivative:

$$C' = x(s(z) - y)$$

Note:

- C' is the derivative of cost with respect to weights
 - y is the actual class label (0 or 1)
 - $s(z)$ is your model's prediction
 - x is your feature or feature vector.
-

Notice how this gradient is the same as the *MSE (L2)* gradient, the only difference is the hypothesis function.

⁶ <http://math.stackexchange.com/questions/78575/derivative-of-sigmoid-function-sigma-x-frac11e-x>

Pseudocode

```
Repeat {

    1. Calculate gradient average
    2. Multiply by learning rate
    3. Subtract from weights

}
```

Code

```
def update_weights(features, labels, weights, lr):
    '''
    Vectorized Gradient Descent

    Features: (200, 3)
    Labels: (200, 1)
    Weights: (3, 1)
    '''
    N = len(features)

    #1 - Get Predictions
    predictions = predict(features, weights)

    #2 Transpose features from (200, 3) to (3, 200)
    # So we can multiply w the (200,1) cost matrix.
    # Returns a (3,1) matrix holding 3 partial derivatives --
    # one for each feature -- representing the aggregate
    # slope of the cost function across all observations
    gradient = np.dot(features.T, predictions - labels)

    #3 Take the average cost derivative for each feature
    gradient /= N

    #4 - Multiply the gradient by our learning rate
    gradient *= lr

    #5 - Subtract from our weights to minimize cost
    weights -= gradient

    return weights
```

3.2.6 Mapping probabilities to classes

The final step is assign class labels (0 or 1) to our predicted probabilities.

Decision boundary

```
def decision_boundary(prob):
    return 1 if prob >= .5 else 0
```

Convert probabilities to classes

```
def classify(predictions):  
    '''  
    input  - N element array of predictions between 0 and 1  
    output - N element array of 0s (False) and 1s (True)  
    '''  
    decision_boundary = np.vectorize(decision_boundary)  
    return decision_boundary(predictions).flatten()
```

Example output

```
Probabilities = [ 0.967, 0.448, 0.015, 0.780, 0.978, 0.004]  
Classifications = [1, 0, 0, 1, 1, 0]
```

3.2.7 Training

Our training code is the same as we used for *linear regression*.

```
def train(features, labels, weights, lr, iters):  
    cost_history = []  
  
    for i in range(iters):  
        weights = update_weights(features, labels, weights, lr)  
  
        #Calculate error for auditing purposes  
        cost = cost_function(features, labels, weights)  
        cost_history.append(cost)  
  
        # Log Progress  
        if i % 1000 == 0:  
            print "iter: "+str(i) + " cost: "+str(cost)  
  
    return weights, cost_history
```

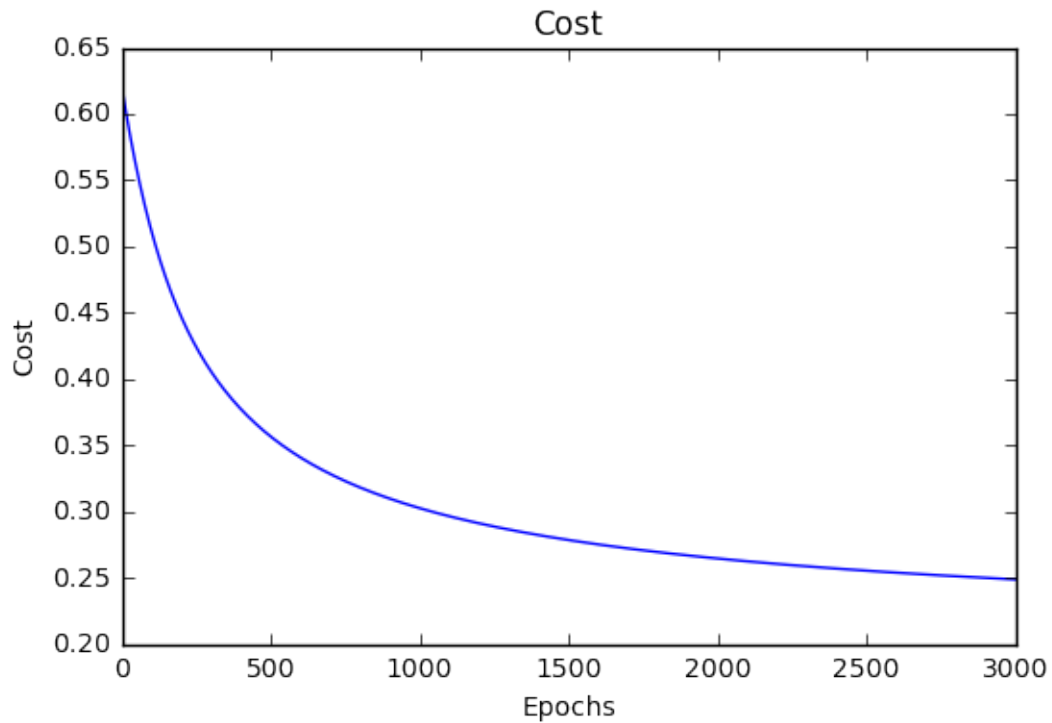
3.2.8 Model evaluation

If our model is working, we should see our cost decrease after every iteration.

```
iter: 0 cost: 0.635  
iter: 1000 cost: 0.302  
iter: 2000 cost: 0.264
```

Final cost: 0.2487. **Final weights:** [-8.197, .921, .738]

Cost history



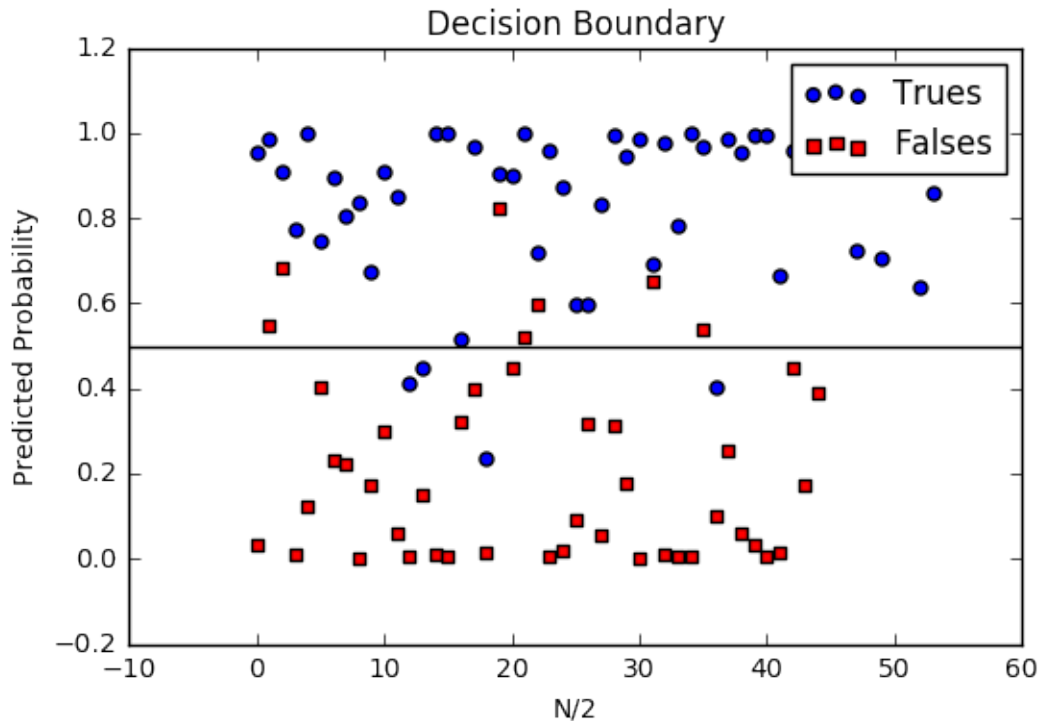
Accuracy

Accuracy measures how correct our predictions were. In this case we simply compare predicted labels to true labels and divide by the total.

```
def accuracy(predicted_labels, actual_labels):  
    diff = predicted_labels - actual_labels  
    return 1.0 - (float(np.count_nonzero(diff)) / len(diff))
```

Decision boundary

Another helpful technique is to plot the decision boundary on top of our predictions to see how our labels compare to the actual labels. This involves plotting our predicted probabilities and coloring them with their true labels.



Code to plot the decision boundary

```
def plot_decision_boundary(trues, falses):
    fig = plt.figure()
    ax = fig.add_subplot(111)

    no_of_preds = len(trues) + len(falses)

    ax.scatter([i for i in range(len(trues))], trues, s=25, c='b', marker="o", label=
    ↪ 'Trues')
    ax.scatter([i for i in range(len(falses))], falses, s=25, c='r', marker="s",
    ↪ label='Falses')

    plt.legend(loc='upper right');
    ax.set_title("Decision Boundary")
    ax.set_xlabel('N/2')
    ax.set_ylabel('Predicted Probability')
    plt.axhline(.5, color='black')
    plt.show()
```

3.3 Multiclass logistic regression

Instead of $y = 0, 1$ we will expand our definition so that $y = 0, 1 \dots n$. Basically we re-run binary classification multiple times, once for each class.

3.3.1 Procedure

1. Divide the problem into $n+1$ binary classification problems (+1 because the index starts at 0?).
2. For each class...
3. Predict the probability the observations are in that single class.
4. prediction = $\max(\text{probability of the classes})$

For each sub-problem, we select one class (YES) and lump all the others into a second class (NO). Then we take the class with the highest predicted value.

3.3.2 Softmax activation

The softmax function (softargmax or normalized exponential function) is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval $[0, 1]$, and the components will add up to 1, so that they can be interpreted as probabilities. The standard (unit) softmax function is defined by the formula

$$(z_i) = \frac{e^{z(i)}}{\sum_{j=1}^K e^{z(j)}} \text{ for } i = 1, \dots, K \text{ and } z = z_1, \dots, z_K \quad (3.2)$$

In words: we apply the standard exponential function to each element z_i of the input vector z and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector (z) is 1.⁹

3.3.3 Scikit-Learn example

Let's compare our performance to the LogisticRegression model provided by scikit-learn⁸.

```
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split

# Normalize grades to values between 0 and 1 for more efficient computation
normalized_range = sklearn.preprocessing.MinMaxScaler(feature_range=(-1,1))

# Extract Features + Labels
labels.shape = (100,) #scikit expects this
features = normalized_range.fit_transform(features)

# Create Test/Train
features_train, features_test, labels_train, labels_test = train_test_split(features,
    ↪ labels, test_size=0.4)

# Scikit Logistic Regression
scikit_log_reg = LogisticRegression()
scikit_log_reg.fit(features_train, labels_train)

#Score is Mean Accuracy
```

(continues on next page)

⁹ https://en.wikipedia.org/wiki/Softmax_function

⁸ http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

(continued from previous page)

```
scikit_score = clf.score(features_test, labels_test)
print 'Scikit score: ', scikit_score

#Our Mean Accuracy
observations, features, labels, weights = run()
probabilities = predict(features, weights).flatten()
classifications = classifier(probabilities)
our_acc = accuracy(classifications, labels.flatten())
print 'Our score: ', our_acc
```

Scikit score: 0.88. Our score: 0.89

References

Definitions of common machine learning terms.

Accuracy Percentage of correct predictions made by the model.

Algorithm A method, function, or series of instructions used to generate a machine learning *model*. Examples include linear regression, decision trees, support vector machines, and neural networks.

Attribute A quality describing an observation (e.g. color, size, weight). In Excel terms, these are column headers.

Bias metric What is the average difference between your predictions and the correct value for that observation?

- **Low bias** could mean every prediction is correct. It could also mean half of your predictions are above their actual values and half are below, in equal proportion, resulting in low average difference.
- **High bias** (with low variance) suggests your model may be underfitting and you're using the wrong architecture for the job.

Bias term Allow models to represent patterns that do not pass through the origin. For example, if all my features were 0, would my output also be zero? Is it possible there is some base value upon which my features have an effect? Bias terms typically accompany weights and are attached to neurons or filters.

Categorical Variables Variables with a discrete set of possible values. Can be ordinal (order matters) or nominal (order doesn't matter).

Classification Predicting a categorical output.

- **Binary classification** predicts one of two possible outcomes (e.g. is the email spam or not spam?)
- **Multi-class classification** predicts one of multiple possible outcomes (e.g. is this a photo of a cat, dog, horse or human?)

Classification Threshold The lowest probability value at which we're comfortable asserting a positive classification. For example, if the predicted probability of being diabetic is > 50%, return True, otherwise return False.

Clustering Unsupervised grouping of data into buckets.

Confusion Matrix Table that describes the performance of a classification model by grouping predictions into 4 categories.

- **True Positives:** we *correctly* predicted they do have diabetes

- **True Negatives:** we *correctly* predicted they don't have diabetes
- **False Positives:** we *incorrectly* predicted they do have diabetes (Type I error)
- **False Negatives:** we *incorrectly* predicted they don't have diabetes (Type II error)

Continuous Variables Variables with a range of possible values defined by a number scale (e.g. sales, lifespan).

Convergence A state reached during the training of a model when the *loss* changes very little between each iteration.

Deduction A top-down approach to answering questions or solving problems. A logic technique that starts with a theory and tests that theory with observations to derive a conclusion. E.g. We suspect X, but we need to test our hypothesis before coming to any conclusions.

Deep Learning Deep Learning is derived from a machine learning algorithm called perceptron or multi layer perceptron that is gaining more and more attention nowadays because of its success in different fields like, computer vision to signal processing and medical diagnosis to self-driving cars. Like other AI algorithms, deep learning is based on decades of research. Nowadays, we have more and more data and cheap computing power that makes this algorithm really powerful in achieving state of the art accuracy. In modern world this algorithm is known as artificial neural network. Deep learning is much more accurate and robust compared to traditional artificial neural networks. But it is highly influenced by machine learning's neural network and perceptron networks.

Dimension Dimension for machine learning and data scientist is different from physics. Here, dimension of data means how many features you have in your data ocean(data-set). e.g in case of object detection application, flatten image size and color channel(e.g 28*28*3) is a feature of the input set. In case of house price prediction (maybe) house size is the data-set so we call it 1 dimensional data.

Epoch An epoch describes the number of times the algorithm sees the entire data set.

Extrapolation Making predictions outside the range of a dataset. E.g. My dog barks, so all dogs must bark. In machine learning we often run into trouble when we extrapolate outside the range of our training data.

False Positive Rate Defined as

$$FPR = 1 - Specificity = \frac{FalsePositives}{FalsePositives + TrueNegatives}$$

The False Positive Rate forms the x-axis of the *ROC curve*.

Feature With respect to a dataset, a feature represents an *attribute* and value combination. Color is an attribute. "Color is blue" is a feature. In Excel terms, features are similar to cells. The term feature has other definitions in different contexts.

Feature Selection Feature selection is the process of selecting relevant features from a data-set for creating a Machine Learning model.

Feature Vector A list of features describing an observation with multiple attributes. In Excel we call this a row.

Gradient Accumulation A mechanism to split the batch of samples—used for training a neural network—into several mini-batches of samples that will be run sequentially. This is used to enable using large batch sizes that require more GPU memory than available.

Hyperparameters Hyperparameters are higher-level properties of a model such as how fast it can learn (learning rate) or complexity of a model. The depth of trees in a Decision Tree or number of hidden layers in a Neural Networks are examples of hyper parameters.

Induction A bottoms-up approach to answering questions or solving problems. A logic technique that goes from observations to theory. E.g. We keep observing X, so we infer that Y must be True.

Instance A data point, row, or sample in a dataset. Another term for *observation*.

Label The "answer" portion of an *observation* in *supervised learning*. For example, in a dataset used to classify flowers into different species, the features might include the petal length and petal width, while the label would be the flower's species.

Learning Rate The size of the update steps to take during optimization loops like *Gradient Descent*. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

Loss $\text{Loss} = \text{true_value}(\text{from data-set}) - \text{predicted value}(\text{from ML-model})$ The lower the loss, the better a model (unless the model has over-fitted to the training data). The loss is calculated on training and validation and its interpretation is how well the model is doing for these two sets. Unlike accuracy, loss is not a percentage. It is a summation of the errors made for each example in training or validation sets.

Machine Learning Mitchell (1997) provides a succinct definition: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.” In simple language machine learning is a field in which human made algorithms have an ability learn by itself or predict future for unseen data.

Model A data structure that stores a representation of a dataset (weights and biases). Models are created/learned when you train an algorithm on a dataset.

Neural Networks Neural Networks are mathematical algorithms modeled after the brain’s architecture, designed to recognize patterns and relationships in data.

Normalization Restriction of the values of weights in regression to avoid overfitting and improving computation speed.

Noise Any irrelevant information or randomness in a dataset which obscures the underlying pattern.

Null Accuracy Baseline accuracy that can be achieved by always predicting the most frequent class (“B has the highest frequency, so let’s guess B every time”).

Observation A data point, row, or sample in a dataset. Another term for *instance*.

Outlier An observation that deviates significantly from other observations in the dataset.

Overfitting Overfitting occurs when your model learns the training data too well and incorporates details and noise specific to your dataset. You can tell a model is overfitting when it performs great on your training/validation set, but poorly on your test set (or new real-world data).

Parameters Parameters are properties of training data learned by training a machine learning model or classifier. They are adjusted using optimization algorithms and unique to each experiment.

Examples of parameters include:

- weights in an artificial neural network
- support vectors in a support vector machine
- coefficients in a linear or logistic regression

Precision In the context of binary classification (Yes/No), precision measures the model’s performance at classifying positive observations (i.e. “Yes”). In other words, when a positive value is predicted, how often is the prediction correct? We could game this metric by only returning positive for the single observation we are most confident in.

$$P = \frac{\text{TruePositives}}{\text{TruePositives} + \text{FalsePositives}}$$

Recall Also called sensitivity. In the context of binary classification (Yes/No), recall measures how “sensitive” the classifier is at detecting positive instances. In other words, for all the true observations in our sample, how many did we “catch.” We could game this metric by always classifying observations as positive.

$$R = \frac{\text{TruePositives}}{\text{TruePositives} + \text{FalseNegatives}}$$

Recall vs Precision Say we are analyzing Brain scans and trying to predict whether a person has a tumor (True) or not (False). We feed it into our model and our model starts guessing.

- **Precision** is the % of True guesses that were actually correct! If we guess 1 image is True out of 100 images and that image is actually True, then our precision is 100%! Our results aren't helpful however because we missed 10 brain tumors! We were super precise when we tried, but we didn't try hard enough.
- **Recall**, or Sensitivity, provides another lens which with to view how good our model is. Again let's say there are 100 images, 10 with brain tumors, and we correctly guessed 1 had a brain tumor. Precision is 100%, but recall is 10%. Perfect recall requires that we catch all 10 tumors!

Regression Predicting a continuous output (e.g. price, sales).

Regularization Regularization is a technique utilized to combat the overfitting problem. This is achieved by adding a complexity term to the loss function that gives a bigger loss for more complex models

Reinforcement Learning Training a model to maximize a reward via iterative trial and error.

ROC (Receiver Operating Characteristic) Curve A plot of the *true positive rate* against the *false positive rate* at all *classification thresholds*. This is used to evaluate the performance of a classification model at different classification thresholds. The area under the ROC curve can be interpreted as the probability that the model correctly distinguishes between a randomly chosen positive observation (e.g. "spam") and a randomly chosen negative observation (e.g. "not spam").

Segmentation It is the process of partitioning a data set into multiple distinct sets. This separation is done such that the members of the same set are similar to each other and different from the members of other sets.

Specificity In the context of binary classification (Yes/No), specificity measures the model's performance at classifying negative observations (i.e. "No"). In other words, when the correct label is negative, how often is the prediction correct? We could game this metric if we predict everything as negative.

$$S = \frac{TrueNegatives}{TrueNegatives + FalsePositives}$$

Supervised Learning Training a model using a labeled dataset.

Test Set A set of observations used at the end of model training and validation to assess the predictive power of your model. How generalizable is your model to unseen data?

Training Set A set of observations used to generate machine learning models.

Transfer Learning A machine learning method where a model developed for a task is reused as the starting point for a model on a second task. In transfer learning, we take the pre-trained weights of an already trained model (one that has been trained on millions of images belonging to 1000's of classes, on several high power GPU's for several days) and use these already learned features to predict new classes.

True Positive Rate Another term for *recall*, i.e.

$$TPR = \frac{TruePositives}{TruePositives + FalseNegatives}$$

The True Positive Rate forms the y-axis of the *ROC curve*.

Type 1 Error False Positives. Consider a company optimizing hiring practices to reduce false positives in job offers. A type 1 error occurs when candidate seems good and they hire him, but he is actually bad.

Type 2 Error False Negatives. The candidate was great but the company passed on him.

Underfitting Underfitting occurs when your model over-generalizes and fails to incorporate relevant variations in your data that would give your model more predictive power. You can tell a model is underfitting when it performs poorly on both training and test sets.

Universal Approximation Theorem A neural network with one hidden layer can approximate any continuous function but only for inputs in a specific range. If you train a network on inputs between -2 and 2, then it will work well for inputs in the same range, but you can't expect it to generalize to other inputs without retraining the model or adding more hidden neurons.

Unsupervised Learning Training a model to find patterns in an unlabeled dataset (e.g. clustering).

Validation Set A set of observations used during model training to provide feedback on how well the current parameters generalize beyond the training set. If training error decreases but validation error increases, your model is likely overfitting and you should pause training.

Variance How tightly packed are your predictions for a particular observation relative to each other?

- **Low variance** suggests your model is internally consistent, with predictions varying little from each other after every iteration.
- **High variance** (with low bias) suggests your model may be overfitting and reading too deeply into the noise found in every training set.

References

- *Introduction*
- *Derivatives*
 - *Geometric definition*
 - *Taking the derivative*
 - *Step-by-step*
 - *Machine learning use cases*
- *Chain rule*
 - *How It Works*
 - *Step-by-step*
 - *Multiple functions*
- *Gradients*
 - *Partial derivatives*
 - *Step-by-step*
 - *Directional derivatives*
 - *Useful properties*
- *Integrals*
 - *Computing integrals*
 - *Applications of integration*
 - * *Computing probabilities*
 - * *Expected value*

5.1 Introduction

You need to know some basic calculus in order to understand how functions change over time (derivatives), and to calculate the total amount of a quantity that accumulates over a time period (integrals). The language of calculus will allow you to speak precisely about the properties of functions and better understand their behaviour.

Normally taking a calculus course involves doing lots of tedious calculations by hand, but having the power of computers on your side can make the process much more fun. This section describes the key ideas of calculus which you'll need to know to understand machine learning concepts.

5.2 Derivatives

A derivative can be defined in two ways:

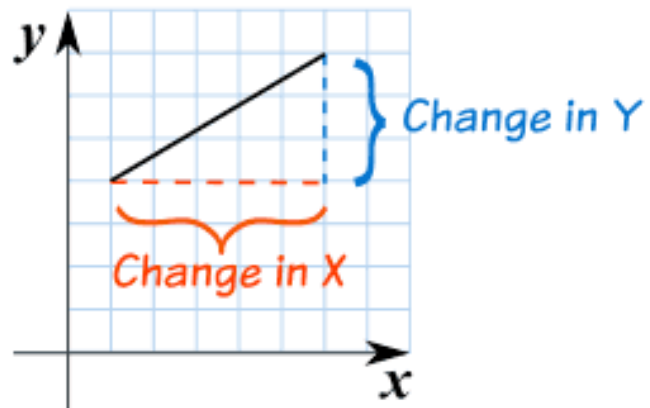
1. Instantaneous rate of change (Physics)
2. Slope of a line at a specific point (Geometry)

Both represent the same principle, but for our purposes it's easier to explain using the geometric definition.

5.2.1 Geometric definition

In geometry slope represents the steepness of a line. It answers the question: how much does y or $f(x)$ change given a specific change in x ?

$$\text{Slope} = \frac{\text{Change in Y}}{\text{Change in X}}$$

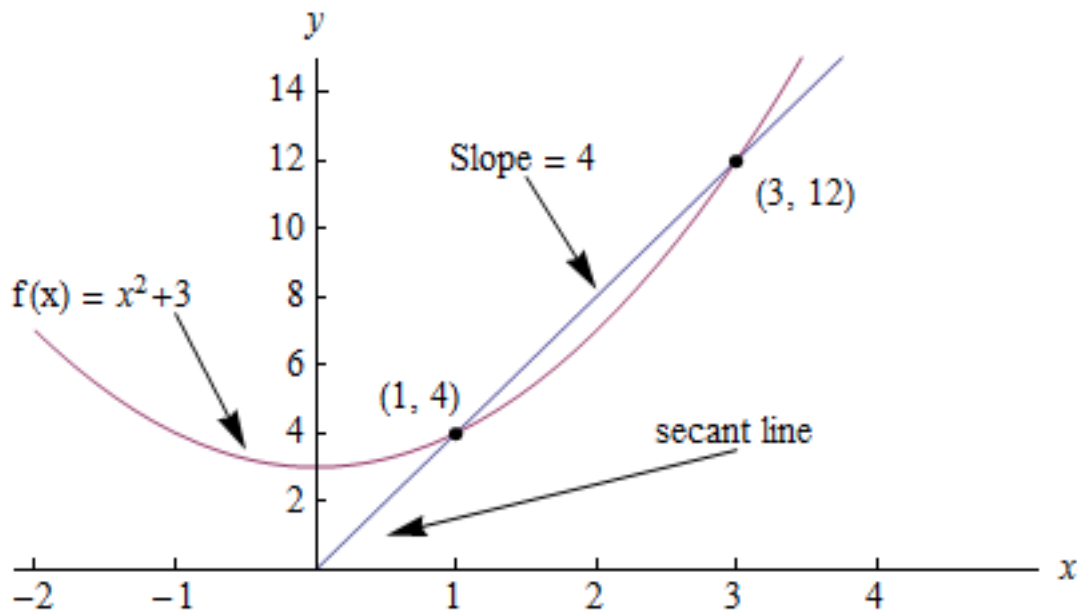


Using this definition we can easily calculate the slope between two points. But what if I asked you, instead of the slope between two points, what is the slope at a single point on the line? In this case there isn't any obvious "rise-over-run" to calculate. Derivatives help us answer this question.

A derivative outputs an expression we can use to calculate the *instantaneous rate of change*, or slope, at a single point on a line. After solving for the derivative you can use it to calculate the slope at every other point on the line.

5.2.2 Taking the derivative

Consider the graph below, where $f(x) = x^2 + 3$.



The slope between (1,4) and (3,12) would be:

$$\text{slope} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{12 - 4}{3 - 1} = 4$$

But how do we calculate the slope at point (1,4) to reveal the change in slope at that specific point?

One way would be to find the two nearest points, calculate their slopes relative to x and take the average. But calculus provides an easier, more precise way: compute the derivative. Computing the derivative of a function is essentially the same as our original proposal, but instead of finding the two closest points, we make up an imaginary point an infinitesimally small distance away from x and compute the slope between x and the new point.

In this way, derivatives help us answer the question: how does $f(x)$ change if we make a very very tiny increase to x ? In other words, derivatives help *estimate* the slope between two points that are an infinitesimally small distance away from each other. A very, very, very small distance, but large enough to calculate the slope.

In math language we represent this infinitesimally small increase using a limit. A limit is defined as the output value a function approaches as the input value approaches another value. In our case the target value is the specific point at which we want to calculate slope.

5.2.3 Step-by-step

Calculating the derivative is the same as calculating normal slope, however in this case we calculate the slope between our point and a point infinitesimally close to it. We use the variable h to represent this infinitesimally distance. Here are the steps:

1. Given the function:

$$f(x) = x^2$$

2. Increment x by a very small value h ($h = x$)

$$f(x + h) = (x + h)^2$$

3. Apply the slope formula

$$\frac{f(x+h) - f(x)}{h}$$

4. Simplify the equation

$$\frac{x^2 + 2xh + h^2 - x^2}{h}$$

$$\frac{2xh + h^2}{h} = 2x + h$$

5. Set h to 0 (the limit as h heads toward 0)

$$2x + 0 = 2x$$

So what does this mean? It means for the function $f(x) = x^2$, the slope at any point equals $2x$. The formula is defined as:

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Code

Let's write code to calculate the derivative of any function $f(x)$. We test our function works as expected on the input $f(x) = x^2$ producing a value close to the actual derivative $2x$.

```
def get_derivative(func, x):  
    """Compute the derivative of `func` at the location `x`."""  
    h = 0.0001 # step size  
    return (func(x+h) - func(x)) / h # rise-over-run  
  
def f(x): return x**2 # some test function f(x)=x^2  
x = 3 # the location of interest  
computed = get_derivative(f, x)  
actual = 2*x  
  
computed, actual # = 6.0001, 6 # pretty close if you ask me...
```

In general it's preferable to use the math to obtain exact **derivative formulas**, but keep in mind you can always compute derivatives numerically by computing the rise-over-run for a "small step" h .

5.2.4 Machine learning use cases

Machine learning uses derivatives in optimization problems. Optimization algorithms like *gradient descent* use derivatives to decide whether to increase or decrease weights in order to maximize or minimize some objective (e.g. a model's accuracy or error functions). Derivatives also help us approximate nonlinear functions as linear functions (tangent lines), which have constant slopes. With a constant slope we can decide whether to move up or down the slope (increase or decrease our weights) to get closer to the target value (class label).

5.3 Chain rule

The chain rule is a formula for calculating the derivatives of composite functions. Composite functions are functions composed of functions inside other function(s).

5.3.1 How It Works

Given a composite function $f(x) = A(B(x))$, the derivative of $f(x)$ equals the product of the derivative of A with respect to $B(x)$ and the derivative of B with respect to x .

composite function derivative = outer function derivative * inner function derivative

For example, given a composite function $f(x)$, where:

$$f(x) = h(g(x))$$

The chain rule tells us that the derivative of $f(x)$ equals:

$$\frac{df}{dx} = \frac{dh}{dg} \cdot \frac{dg}{dx}$$

5.3.2 Step-by-step

Say $f(x)$ is composed of two functions $h(x) = x^3$ and $g(x) = x^2$. And that:

$$\begin{aligned} f(x) &= h(g(x)) \\ &= (x^2)^3 \\ &= x^6 \end{aligned} \tag{5.1}$$

The derivative of $f(x)$ would equal:

$$\begin{aligned} \frac{df}{dx} &= \frac{dh}{dg} \frac{dg}{dx} \\ &= \frac{dh}{d(x^2)} \frac{dg}{dx} \end{aligned} \tag{5.4}$$

Steps

1. Solve for the inner derivative of $g(x) = x^2$

$$\frac{dg}{dx} = 2x$$

2. Solve for the outer derivative of $h(x) = x^3$, using a placeholder b to represent the inner function x^2

$$\frac{dh}{db} = 3b^2$$

3. Swap out the placeholder variable (b) for the inner function ($g(x)$)

$$\begin{aligned} &3(x^2)^2 \\ &3x^4 \end{aligned}$$

4. Return the product of the two derivatives

$$3x^4 \cdot 2x = 6x^5$$

5.3.3 Multiple functions

In the above example we assumed a composite function containing a single inner function. But the chain rule can also be applied to higher-order functions like:

$$f(x) = A(B(C(x)))$$

The chain rule tells us that the derivative of this function equals:

$$\frac{df}{dx} = \frac{dA}{dB} \frac{dB}{dC} \frac{dC}{dx}$$

We can also write this derivative equation f' notation:

$$f' = A'(B(C(x))) \cdot B'(C(x)) \cdot C'(x)$$

Steps

Given the function $f(x) = A(B(C(x)))$, let's assume:

$$\begin{aligned} A(x) &= \sin(x) \\ B(x) &= 5x^2 \\ C(x) &= 4x \end{aligned} \tag{5.6}$$

The derivatives of these functions would be:

$$\begin{aligned} A'(x) &= \cos(x) \\ B'(x) &= 10x \\ C'(x) &= 4 \end{aligned} \tag{5.9}$$

We can calculate the derivative of $f(x)$ using the following formula:

$$f'(x) = A'((4x)^2) \cdot B'(4x) \cdot C'(x)$$

We then input the derivatives and simplify the expression:

$$\begin{aligned} f'(x) &= \cos((4x)^2) \cdot 2(4x) \cdot 4 \\ &= \cos(16x^2) \cdot 8x \\ &= 8x \cos(16x^2) \end{aligned} \tag{5.12}$$

5.4 Gradients

A gradient is a vector that stores the partial derivatives of multivariable functions. It helps us calculate the slope at a specific point on a curve for functions with multiple independent variables. In order to calculate this more complex slope, we need to isolate each variable to determine how it impacts the output on its own. To do this we iterate through each of the variables and calculate the derivative of the function after holding all other variables constant. Each iteration produces a partial derivative which we store in the gradient.

5.4.1 Partial derivatives

In functions with 2 or more variables, the partial derivative is the derivative of one variable with respect to the others. If we change x , but hold all other variables constant, how does $f(x, z)$ change? That's one partial derivative. The next variable is z . If we change z but hold x constant, how does $f(x, z)$ change? We store partial derivatives in a gradient, which represents the full derivative of the multivariable function.

5.4.2 Step-by-step

Here are the steps to calculate the gradient for a multivariable function:

1. Given a multivariable function

$$f(x, z) = 2z^3x^2$$

2. Calculate the derivative with respect to x

$$\frac{df}{dx}(x, z)$$

3. Swap $2z^3$ with a constant value b

$$f(x, z) = bx^2$$

4. Calculate the derivative with b constant

$$\begin{aligned} \frac{df}{dx} &= \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} & (5.15) \\ &= \lim_{h \rightarrow 0} \frac{b(x+h)^2 - b(x^2)}{h} & (5.16) \\ &= \lim_{h \rightarrow 0} \frac{b((x+h)(x+h)) - bx^2}{h} & (5.17) \\ &= \lim_{h \rightarrow 0} \frac{b((x^2 + xh + hx + h^2)) - bx^2}{h} & (5.18) \\ &= \lim_{h \rightarrow 0} \frac{bx^2 + 2bxh + bh^2 - bx^2}{h} & (5.19) \\ &= \lim_{h \rightarrow 0} \frac{2bxh + bh^2}{h} & (5.20) \\ &= \lim_{h \rightarrow 0} 2bx + bh & (5.21) \\ &= 2bx & (5.22) \end{aligned}$$

As $h \rightarrow 0 \dots$

$$2bx + 0$$

5. Swap $2z^3$ back into the equation, to find the derivative with respect to x .

$$\begin{aligned} \frac{df}{dx}(x, z) &= 2(2z^3)x & (5.23) \\ &= 4z^3x & (5.24) \end{aligned}$$

6. Repeat the above steps to calculate the derivative with respect to z

$$\frac{df}{dz}(x, z) = 6x^2z^2$$

7. Store the partial derivatives in a gradient

$$\nabla f(x, z) = \begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dz} \end{bmatrix} = \begin{bmatrix} 4z^3x \\ 6x^2z^2 \end{bmatrix}$$

5.4.3 Directional derivatives

Another important concept is directional derivatives. When calculating the partial derivatives of multivariable functions we use our old technique of analyzing the impact of infinitesimally small increases to each of our independent variables. By increasing each variable we alter the function output in the direction of the slope.

But what if we want to change directions? For example, imagine we're traveling north through mountainous terrain on a 3-dimensional plane. The gradient we calculated above tells us we're traveling north at our current location. But what if we wanted to travel southwest? How can we determine the steepness of the hills in the southwest direction? Directional derivatives help us find the slope if we move in a direction different from the one specified by the gradient.

Math

The directional derivative is computed by taking the dot product¹¹ of the gradient of f and a unit vector \vec{v} of “tiny nudges” representing the direction. The unit vector describes the proportions we want to move in each direction. The output of this calculation is a scalar number representing how much f will change if the current input moves with vector \vec{v} .

Let's say you have the function $f(x, y, z)$ and you want to compute its directional derivative along the following vector²:

$$\vec{v} = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

As described above, we take the dot product of the gradient and the directional vector:

$$\begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dy} \\ \frac{df}{dz} \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

We can rewrite the dot product as:

$$\nabla_{\vec{v}} f = 2 \frac{df}{dx} + 3 \frac{df}{dy} - 1 \frac{df}{dz}$$

This should make sense because a tiny nudge along \vec{v} can be broken down into two tiny nudges in the x-direction, three tiny nudges in the y-direction, and a tiny nudge backwards, by 1 in the z-direction.

5.4.4 Useful properties

There are two additional properties of gradients that are especially useful in deep learning. The gradient of a function:

1. Always points in the direction of greatest increase of a function ([explained here](#))
2. Is zero at a local maximum or local minimum

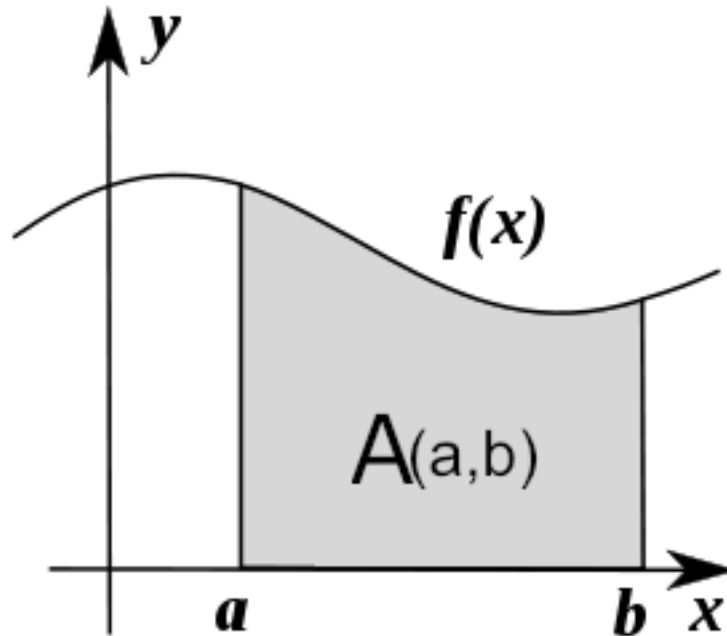
5.5 Integrals

The integral of $f(x)$ corresponds to the computation of the area under the graph of $f(x)$. The area under $f(x)$ between the points $x = a$ and $x = b$ is denoted as follows:

$$A(a, b) = \int_a^b f(x) dx.$$

¹¹ https://en.wikipedia.org/wiki/Dot_product

² <https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/directional-derivative-introduction>



The area $A(a, b)$ is bounded by the function $f(x)$ from above, by the x -axis from below, and by two vertical lines at $x = a$ and $x = b$. The points $x = a$ and $x = b$ are called the limits of integration. The \int sign comes from the Latin word summa. The integral is the “sum” of the values of $f(x)$ between the two limits of integration.

The *integral function* $F(c)$ corresponds to the area calculation as a function of the upper limit of integration:

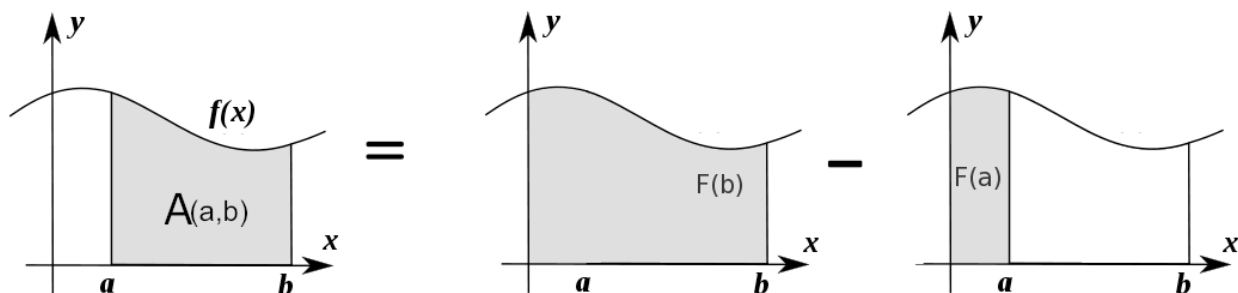
$$F(c) \equiv \int_0^c f(x) dx.$$

There are two variables and one constant in this formula. The input variable c describes the upper limit of integration. The *integration variable* x performs a sweep from $x = 0$ until $x = c$. The constant 0 describes the lower limit of integration. Note that choosing $x = 0$ for the starting point of the integral function was an arbitrary choice.

The integral function $F(c)$ contains the “precomputed” information about the area under the graph of $f(x)$. The derivative function $f'(x)$ tells us the “slope of the graph” property of the function $f(x)$ for all values of x . Similarly, the integral function $F(c)$ tells us the “area under the graph” property of the function $f(x)$ for *all* possible limits of integration.

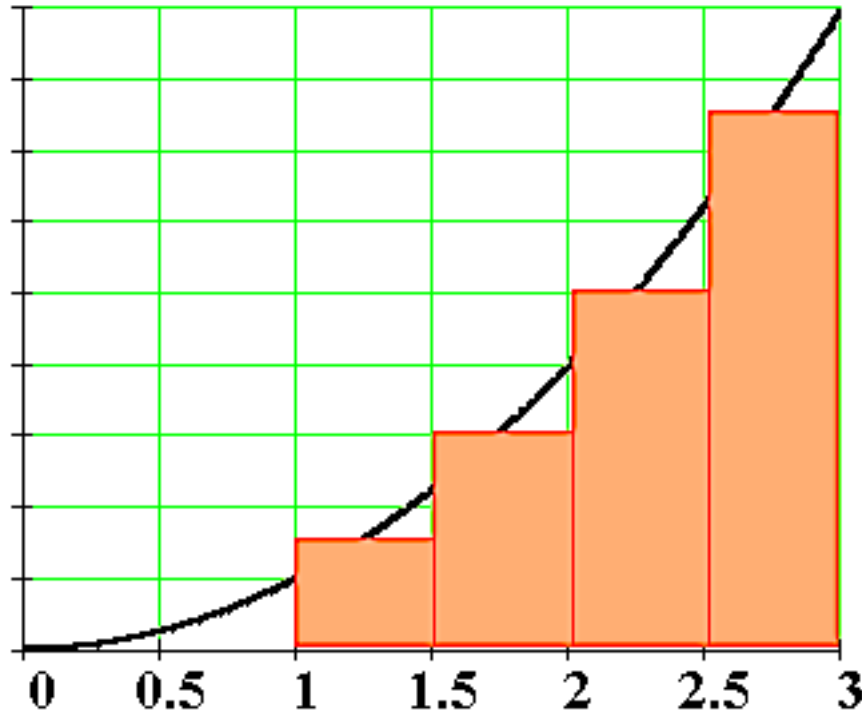
The area under $f(x)$ between $x = a$ and $x = b$ is obtained by calculating the *change* in the integral function as follows:

$$A(a, b) = \int_a^b f(x) dx = F(b) - F(a).$$



5.5.1 Computing integrals

We can approximate the total area under the function $f(x)$ between $x = a$ and $x = b$ by splitting the region into tiny vertical strips of width h , then adding up the areas of the rectangular strips. The figure below shows how to compute the area under $f(x) = x^2$ between $x = 1$ and $x = 3$ by approximating it as four rectangular strips of width $h = 0.5$.



Usually we want to choose h to be a small number so that the approximation is accurate. Here is some sample code that performs integration.

```
def get_integral(func, a, b):
    """Compute the area under `func` between x=a and x=b."""
    h = 0.0001          # width of small rectangle
    x = a                # start at x=a
    total = 0
    while x <= b:        # continue until x=b
        total += h*func(x) # area of rect is base*height
        x += h
    return total

def f(x): return x**2          # some test function f(x)=x^2
computed = get_integral(f, 1, 3)
def actualF(x): return 1.0/3.0*x**3
actual = actualF(3) - actualF(1)
computed, actual    # = 8.6662, 8.6666    # pretty close if you ask me...
```

You can find integral functions using the derivative formulas and some reverse engineering. To find an integral function of the function $f(x)$, we must find a function $F(x)$ such that $F'(x) = f(x)$. Suppose you're given a function $f(x)$ and asked to find its integral function $F(x)$:

$$F(x) = \int f(x) dx.$$

This problem is equivalent to finding a function $F(x)$ whose derivative is $f(x)$:

$$F'(x) = f(x).$$

For example, suppose you want to find the indefinite integral $\int x^2 dx$. We can rephrase this problem as the search for some function $F(x)$ such that

$$F'(x) = x^2.$$

Remembering the derivative formulas we saw above, you guess that $F(x)$ must contain an x^3 term. Taking the derivative of a cubic term results in a quadratic term. Therefore, the function you are looking for has the form $F(x) = cx^3$, for some constant c . Pick the constant c that makes this equation true:

$$F'(x) = 3cx^2 = x^2.$$

Solving $3c = 1$, we find $c = \frac{1}{3}$ and so the integral function is

$$F(x) = \int x^2 dx = \frac{1}{3}x^3 + C.$$

You can verify that $\frac{d}{dx} [\frac{1}{3}x^3 + C] = x^2$.

You can also verify Integrals using maths. Here is a set of [formulas](#) for your reference

5.5.2 Applications of integration

Integral calculations have widespread applications to more areas of science than are practical to list here. Let's explore a few examples related to probabilities.

Computing probabilities

A continuous random variable X is described by its probability density function $p(x)$. A probability density function $p(x)$ is a positive function for which the total area under the curve is 1:

$$p(x) \geq 0, \forall x \quad \text{and} \quad \int_{-\infty}^{\infty} p(x) dx = 1.$$

The probability of observing a value of X between a and b is given by the integral

$$\Pr(a \leq X \leq b) = \int_a^b p(x) dx.$$

Thus, the notion of integration is central to probability theory with continuous random variables.

We also use integration to compute certain characteristic properties of the random variable. The *expected value* and the *variance* are two properties of any random variable X that capture important aspects of its behaviour.

Expected value

The *expected value* of the random variable X is computed using the formula

$$\mu = \int_{-\infty}^{\infty} x p(x) dx.$$

The expected value is a single number that tells us what value of X we can expect to obtain on average from the random variable X . The expected value is also called the *average* or the *mean* of the random variable X .

Variance

The *variance* of the random variable X is defined as follows:

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 p(x).$$

The variance formula computes the expectation of the squared distance of the random variable X from its expected value. The variance σ^2 , also denoted $\text{var}(X)$, gives us an indication of how clustered or spread the values of X are. A small variance indicates the outcomes of X are tightly clustered near the expected value μ , while a large variance indicates the outcomes of X are widely spread. The square root of the variance is called the *standard deviation* and is usually denoted σ .

The expected value μ and the variance σ^2 are two central concepts in probability theory and statistics because they allow us to characterize any random variable. The expected value is a measure of the *central tendency* of the random variable, while the variance σ^2 measures its *dispersion*. Readers familiar with concepts from physics can think of the expected value as the *centre of mass* of the distribution, and the variance as the *moment of inertia* of the distribution.

References

- *Vectors*
 - *Notation*
 - *Vectors in geometry*
 - *Scalar operations*
 - *Elementwise operations*
 - *Dot product*
 - *Hadamard product*
 - *Vector fields*
- *Matrices*
 - *Dimensions*
 - *Scalar operations*
 - *Elementwise operations*
 - *Hadamard product*
 - *Matrix transpose*
 - *Matrix multiplication*
 - *Test yourself*
- *Numpy*
 - *Dot product*
 - *Broadcasting*

Linear algebra is a mathematical toolbox that offers helpful techniques for manipulating groups of numbers simulta-

neously. It provides structures like vectors and matrices (spreadsheets) to hold these numbers and new rules for how to add, subtract, multiply, and divide them. Here is a brief overview of basic linear algebra concepts taken from my linear algebra [post](#) on Medium.

6.1 Vectors

Vectors are 1-dimensional arrays of numbers or terms. In geometry, vectors store the magnitude and direction of a potential change to a point. The vector $[3, -2]$ says go right 3 and down 2. A vector with more than one dimension is called a matrix.

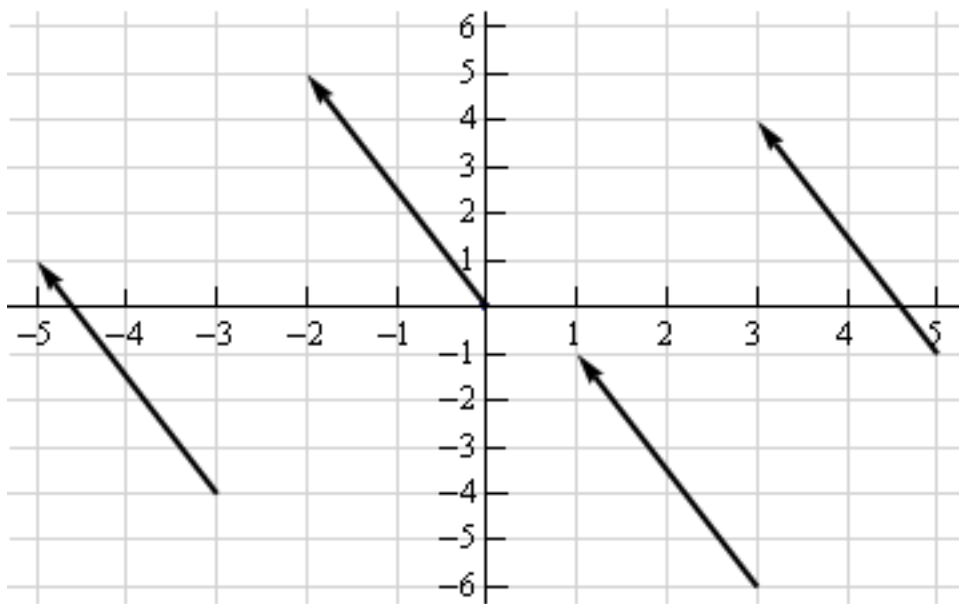
6.1.1 Notation

There are a variety of ways to represent vectors. Here are a few you might come across in your reading.

$$v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = [1 \quad 2 \quad 3]$$

6.1.2 Vectors in geometry

Vectors typically represent movement from a point. They store both the magnitude and direction of potential changes to a point. The vector $[-2, 5]$ says move left 2 units and up 5 units¹.



A vector can be applied to any point in space. The vector's direction equals the slope of the hypotenuse created moving up 5 and left 2. Its magnitude equals the length of the hypotenuse.

¹ http://mathinsight.org/vector_introduction

6.1.3 Scalar operations

Scalar operations involve a vector and a number. You modify the vector in-place by adding, subtracting, or multiplying the number from all the values in the vector.

$$\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} + 1 = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}$$

6.1.4 Elementwise operations

In elementwise operations like addition, subtraction, and division, values that correspond positionally are combined to produce a new vector. The 1st value in vector A is paired with the 1st value in vector B. The 2nd value is paired with the 2nd, and so on. This means the vectors must have equal dimensions to complete the operation.*

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix}$$

```
y = np.array([1,2,3])
x = np.array([2,3,4])
y + x = [3, 5, 7]
y - x = [-1, -1, -1]
y / x = [.5, .67, .75]
```

See below for details on broadcasting in numpy.

6.1.5 Dot product

The dot product of two vectors is a scalar. Dot product of vectors and matrices (matrix multiplication) is one of the most important operations in deep learning.

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = a_1 b_1 + a_2 b_2$$

```
y = np.array([1,2,3])
x = np.array([2,3,4])
np.dot(y,x) = 20
```

6.1.6 Hadamard product

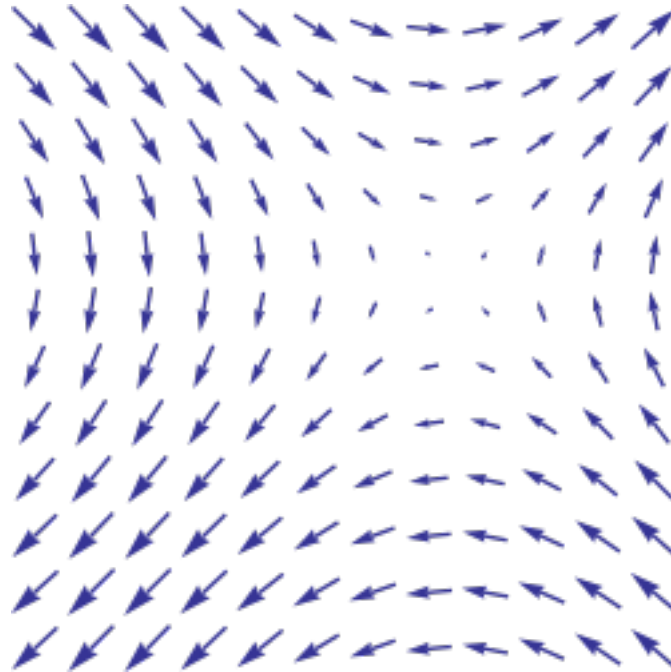
Hadamard Product is elementwise multiplication and it outputs a vector.

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 \\ a_2 \cdot b_2 \end{bmatrix}$$

```
y = np.array([1,2,3])
x = np.array([2,3,4])
y * x = [2, 6, 12]
```

6.1.7 Vector fields

A vector field shows how far the point (x,y) would hypothetically move if we applied a vector function to it like addition or multiplication. Given a point in space, a vector field shows the power and direction of our proposed change at a variety of points in a graph².



This vector field is an interesting one since it moves in different directions depending the starting point. The reason is that the vector behind this field stores terms like $2x$ or x^2 instead of scalar values like -2 and 5. For each point on the graph, we plug the x-coordinate into $2x$ or x^2 and draw an arrow from the starting point to the new location. Vector fields are extremely useful for visualizing machine learning techniques like Gradient Descent.

6.2 Matrices

A matrix is a rectangular grid of numbers or terms (like an Excel spreadsheet) with special rules for addition, subtraction, and multiplication.

6.2.1 Dimensions

We describe the dimensions of a matrix in terms of rows by columns.

$$\begin{bmatrix} 2 & 4 \\ 5 & -7 \\ 12 & 5 \end{bmatrix} \begin{bmatrix} a & 2a & 8 \\ 18 & 7a - 4 & 10 \end{bmatrix}$$

The first has dimensions (3,2). The second (2,3).

```
a = np.array([
    [1, 2, 3],
    [4, 5, 6]
```

(continues on next page)

² https://en.wikipedia.org/wiki/Vector_field

(continued from previous page)

```

])
a.shape == (2,3)
b = np.array([
    [1,2,3]
])
b.shape == (1,3)

```

6.2.2 Scalar operations

Scalar operations with matrices work the same way as they do for vectors. Simply apply the scalar to every element in the matrix—add, subtract, divide, multiply, etc.

$$\begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} + 1 = \begin{bmatrix} 3 & 4 \\ 3 & 4 \\ 3 & 4 \end{bmatrix}$$

```

# Addition
a = np.array([
    [1,2],
    [3,4]])
a + 1
[[2,3],
 [4,5]]

```

6.2.3 Elementwise operations

In order to add, subtract, or divide two matrices they must have equal dimensions. We combine corresponding values in an elementwise fashion to produce a new matrix.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$$

```

a = np.array([
    [1,2],
    [3,4]])
b = np.array([
    [1,2],
    [3,4]])

a + b
[[2, 4],
 [6, 8]]

a -- b
[[0, 0],
 [0, 0]]

```

6.2.4 Hadamard product

Hadamard product of matrices is an elementwise operation. Values that correspond positionally are multiplied to produce a new matrix.

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \odot \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_2 \cdot b_2 \\ a_3 \cdot b_3 & a_4 \cdot b_4 \end{bmatrix}$$

```
a = np.array([
    [2, 3],
    [2, 3]])
b = np.array([
    [3, 4],
    [5, 6]])

# Uses python's multiply operator
a * b
[[ 6, 12],
 [10, 18]]
```

In numpy you can take the Hadamard product of a matrix and vector as long as their dimensions meet the requirements of broadcasting.

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \odot \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 \\ a_2 \cdot b_3 & a_2 \cdot b_4 \end{bmatrix}$$

6.2.5 Matrix transpose

Neural networks frequently process weights and inputs of different sizes where the dimensions do not meet the requirements of matrix multiplication. Matrix transposition (often denoted by a superscript 'T' e.g. M^T) provides a way to “rotate” one of the matrices so that the operation complies with multiplication requirements and can continue. There are two steps to transpose a matrix:

1. Rotate the matrix right 90°
2. Reverse the order of elements in each row (e.g. [a b c] becomes [c b a])

As an example, transpose matrix M into T:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \Rightarrow \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

```
a = np.array([
    [1, 2],
    [3, 4]])

a.T
[[1, 3],
 [2, 4]]
```

6.2.6 Matrix multiplication

Matrix multiplication specifies a set of rules for multiplying matrices together to produce a new matrix.

Rules

Not all matrices are eligible for multiplication. In addition, there is a requirement on the dimensions of the resulting matrix output. Source.

1. The number of columns of the 1st matrix must equal the number of rows of the 2nd
2. The product of an $M \times N$ matrix and an $N \times K$ matrix is an $M \times K$ matrix. The new matrix takes the rows of the 1st and columns of the 2nd

Steps

Matrix multiplication relies on dot product to multiply various combinations of rows and columns. In the image below, taken from Khan Academy's excellent linear algebra course, each entry in Matrix C is the dot product of a row in matrix A and a column in matrix B³.

$$\begin{array}{ccc}
 & \vec{b}_1 & \vec{b}_2 \\
 & \downarrow & \downarrow \\
 \begin{array}{l} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \end{array} & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix} \\
 A & B & C
 \end{array}$$

The operation $a_1 \cdot b_1$ means we take the dot product of the 1st row in matrix A (1, 7) and the 1st column in matrix B (3, 5).

$$a_1 \cdot b_1 = \begin{bmatrix} 1 \\ 7 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \end{bmatrix} = (1 \cdot 3) + (7 \cdot 5) = 38$$

Here's another way to look at it:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1a + 3b & 2a + 4b \\ 1c + 3d & 2c + 4d \\ 1e + 3f & 2e + 4f \end{bmatrix}$$

6.2.7 Test yourself

1. What are the dimensions of the matrix product?

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{bmatrix} = 2 \times 3$$

2. What are the dimensions of the matrix product?

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 3 & 0 \\ 2 & 1 \end{bmatrix} = 3 \times 2$$

³ <https://www.khanacademy.org/math/precaculus/precacalc-matrices/properties-of-matrix-multiplication/a/properties-of-matrix-multiplication>

3. What is the matrix product?

$$\begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 19 & 23 \\ 17 & 24 \end{bmatrix}$$

4. What is the matrix product?}

$$\begin{bmatrix} 3 \\ 5 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 9 \\ 5 & 10 & 15 \end{bmatrix}$$

5. What is the matrix product?

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 32 \end{bmatrix}$$

6.3 Numpy

6.3.1 Dot product

Numpy uses the function `np.dot(A,B)` for both vector and matrix multiplication. It has some other interesting features and gotchas so I encourage you to read the documentation here before use. Also, to multiply two matrices A and B, you can use the expression `A @ B`.

```
a = np.array([
    [1, 2]
])
a.shape == (1,2)
b = np.array([
    [3, 4],
    [5, 6]
])
b.shape == (2,2)

# Multiply
mm = np.dot(a,b) # or a @ b
mm == [13, 16]
mm.shape == (1,2)
```

6.3.2 Broadcasting

In numpy the dimension requirements for elementwise operations are relaxed via a mechanism called broadcasting. Two matrices are compatible if the corresponding dimensions in each matrix (rows vs rows, columns vs columns) meet the following requirements:

1. The dimensions are equal, or
2. One dimension is of size 1

```
a = np.array([
    [1],
    [2]
])
b = np.array([
    [3,4],
    [5,6]
```

(continues on next page)

(continued from previous page)

```
)  
c = np.array([  
    [1,2]  
)  
  
# Same no. of rows  
# Different no. of columns  
# but a has one column so this works  
a * b  
[[ 3, 4],  
 [10, 12]]  
  
# Same no. of columns  
# Different no. of rows  
# but c has one row so this works  
b * c  
[[ 3, 8],  
 [5, 12]]  
  
# Different no. of columns  
# Different no. of rows  
# but both a and c meet the  
# size 1 requirement rule  
a + c  
[[2, 3],  
 [3, 4]]
```

Tutorials

- [Khan Academy Linear Algebra](#)
- [Deep Learning Book Math](#)
- [Andrew Ng Course Notes](#)
- [Linear Algebra Better Explained](#)
- [Understanding Matrices Intuitively](#)
- [Intro To Linear Algebra](#)
- [Immersive Math](#)

References

CHAPTER 7

Probability

- *Links*
- *Screenshots*
- *License*

Basic concepts in probability for machine learning.

This cheatsheet is a 10-page reference in probability that covers a semester's worth of introductory probability.

The cheatsheet is based off of Harvard's introductory probability course, Stat 110. It is co-authored by former Stat 110 Teaching Fellow William Chen and Stat 110 Professor Joe Blitzstein.

7.1 Links

- [Probability Cheatsheet PDF](<http://www.wzchen.com/probability-cheatsheet/>)

7.2 Screenshots

![First Page](<http://i.imgur.com/Oa73huL.jpg>) ![Second Page](<http://i.imgur.com/dyvW2rB.jpg>)

7.3 License

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.][by-nc-sa].

[![Creative Commons License][by-nc-sa-img]][by-nc-sa]

References

CHAPTER 8

Statistics

Basic concepts in statistics for machine learning.

References

Commonly used math symbols in machine learning texts.

- *Algebra*
- *Calculus*
- *Linear algebra*
- *Probability*
- *Set theory*
- *Statistics*

Note: Use the [table generator](#) to quickly add new symbols. Import current tables into tablesgenerator from `figures/*.tgn`. Export and save your changes. Also see helpful [multiline editing](#) in Sublime.

9.1 Algebra

Symbol	Name	Description	Example
(fg)	composite function	a nested function	$(f \circ g)(x) = f(g(x))$
	delta	change / difference	$x = x_1 - x_0$
e	Euler's number	$e = 2.718281828$	$s = \frac{1}{1+e^{-z}}$
\sum	summation	sum of all values	$x_i = x_1 + x_2 + x_3$
\prod	capital pi	product of all values	$x_i = x_1 x_2 x_3$
ϵ	epsilon	tiny number near 0	$lr = 1e-4$

9.2 Calculus

Symbol	Name	Description	Example
x'	derivative	first derivative	$(x^2)' = 2x$
x''	second derivative	second derivative	$(x^2)'' = 2$
\lim	limit	function value as x approaches 0	
	nabla	gradient	$f(a,b,c)$

9.3 Linear algebra

Symbol	Name	Description	Example
$[]$	brackets	matrix or vector	$M = [135]$
\cdot	dot	dot product	$(Z = X \cdot W$
\odot	hadamard	hadamard product	$A = B \odot C$
X^T	transpose	matrix transpose	$W^T \cdot X$
\vec{x}	vector	vector	$v = [123]$
X	matrix	capitalized variables are matrices	X, W, B
\hat{x}	unit vector	vector of magnitude 1	$\hat{x} = [0.20.50.3]$

9.4 Probability

Symbol	Name	Description	Example
$P(A)$	probability	probability of event A	$P(x=1) = 0.5$

9.5 Set theory

Symbol	Name	Description	Example
	set	list of distinct elements	$S = \{1, 5, 7, 9\}$

9.6 Statistics

Symbol	Name	Description	Example
	population mean	mean of population values	
\bar{x}	sample mean	mean of subset of population	
σ^2	population variance	variance of population value	
s^2	sample variance	variance of subset of population	
σ	standard deviation	population standard deviation	
s	sample std dev	standard deviation of sample	
X	correlation	correlation of variables X and Y	
\tilde{x}	median	median value of variable x	

References

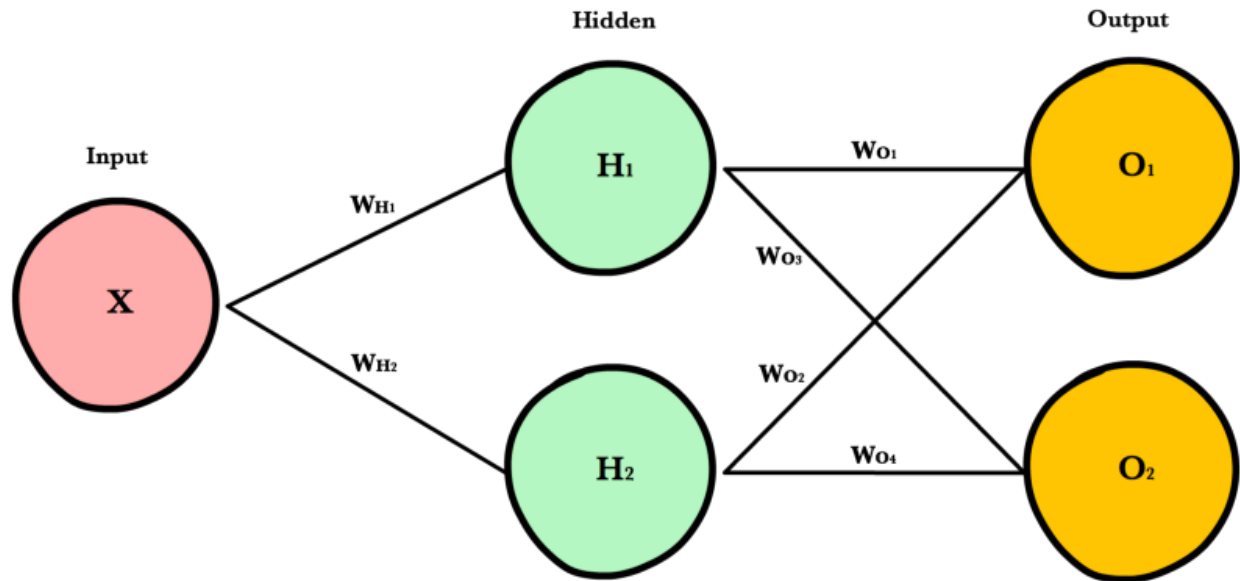
CHAPTER 10

Concepts

- *Neural Network*
- *Neuron*
- *Synapse*
- *Weights*
- *Bias*
- *Layers*
- *Weighted Input*
- *Activation Functions*
- *Loss Functions*
- *Optimization Algorithms*
- *Gradient Accumulation*

10.1 Neural Network

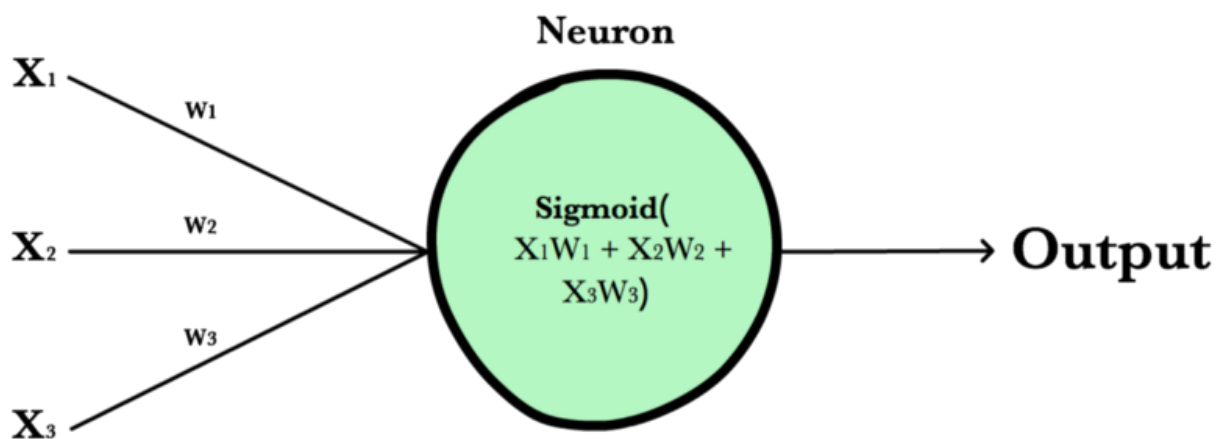
Neural networks are a class of machine learning algorithms used to model complex patterns in datasets using multiple hidden layers and non-linear activation functions. A neural network takes an input, passes it through multiple layers of hidden neurons (mini-functions with unique coefficients that must be learned), and outputs a prediction representing the combined input of all the neurons.



Neural networks are trained iteratively using optimization techniques like gradient descent. After each cycle of training, an error metric is calculated based on the difference between prediction and target. The derivatives of this error metric are calculated and propagated back through the network using a technique called backpropagation. Each neuron's coefficients (weights) are then adjusted relative to how much they contributed to the total error. This process is repeated iteratively until the network error drops below an acceptable threshold.

10.2 Neuron

A neuron takes a group of weighted inputs, applies an activation function, and returns an output.



Inputs to a neuron can either be features from a training set or outputs from a previous layer's neurons. Weights are applied to the inputs as they travel along synapses to reach the neuron. The neuron then applies an activation function to the "sum of weighted inputs" from each incoming synapse and passes the result on to all the neurons in the next layer.

10.3 Synapse

Synapses are like roads in a neural network. They connect inputs to neurons, neurons to neurons, and neurons to outputs. In order to get from one neuron to another, you have to travel along the synapse paying the “toll” (weight) along the way. Each connection between two neurons has a unique synapse with a unique weight attached to it. When we talk about updating weights in a network, we’re really talking about adjusting the weights on these synapses.

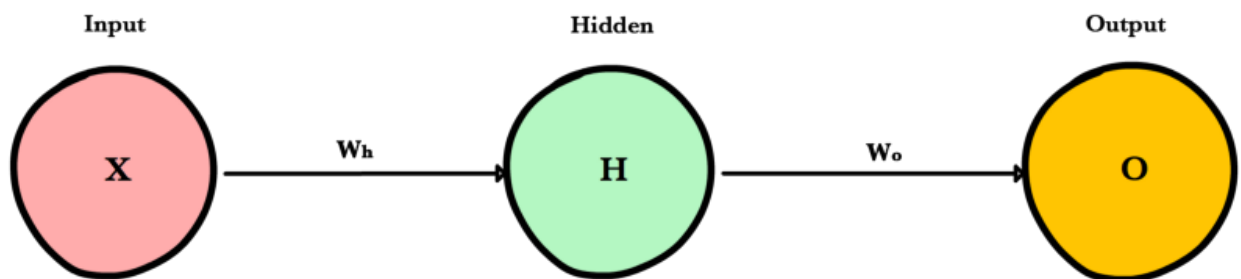
10.4 Weights

Weights are values that control the strength of the connection between two neurons. That is, inputs are typically multiplied by weights, and that defines how much influence the input will have on the output. In other words: when the inputs are transmitted between neurons, the weights are applied to the inputs along with an additional value (the bias)

10.5 Bias

Bias terms are additional constants attached to neurons and added to the weighted input before the activation function is applied. Bias terms help models represent patterns that do not necessarily pass through the origin. For example, if all your features were 0, would your output also be zero? Is it possible there is some base value upon which your features have an effect? Bias terms typically accompany weights and must also be learned by your model.

10.6 Layers



Input Layer

Holds the data your model will train on. Each neuron in the input layer represents a unique attribute in your dataset (e.g. height, hair color, etc.).

Hidden Layer

Sits between the input and output layers and applies an activation function before passing on the results. There are often multiple hidden layers in a network. In traditional networks, hidden layers are typically fully-connected layers—each neuron receives input from all the previous layer’s neurons and sends its output to every neuron in the next layer. This contrasts with how convolutional layers work where the neurons send their output to only some of the neurons in the next layer.

Output Layer

The final layer in a network. It receives input from the previous hidden layer, optionally applies an activation function, and returns an output representing your model's prediction.

10.7 Weighted Input

A neuron's input equals the sum of weighted outputs from all neurons in the previous layer. Each input is multiplied by the weight associated with the synapse connecting the input to the current neuron. If there are 3 inputs or neurons in the previous layer, each neuron in the current layer will have 3 distinct weights—one for each each synapse.

Single Input

$$\begin{aligned} Z &= Input \cdot Weight \\ &= XW \end{aligned}$$

Multiple Inputs

$$\begin{aligned} Z &= \sum_{i=1}^n x_i w_i \\ &= x_1 w_1 + x_2 w_2 + x_3 w_3 \end{aligned}$$

Notice, it's exactly the same equation we use with linear regression! In fact, a neural network with a single neuron is the same as linear regression! The only difference is the neural network post-processes the weighted input with an activation function.

10.8 Activation Functions

Activation functions live inside neural network layers and modify the data they receive before passing it to the next layer. Activation functions give neural networks their power—allowing them to model complex non-linear relationships. By modifying inputs with non-linear functions neural networks can model highly complex relationships between features. Popular activation functions include *relu* and *sigmoid*.

Activation functions typically have the following properties:

- **Non-linear** - In linear regression we're limited to a prediction equation that looks like a straight line. This is nice for simple datasets with a one-to-one relationship between inputs and outputs, but what if the patterns in our dataset were non-linear? (e.g. x^2 , \sin , \log). To model these relationships we need a non-linear prediction equation.¹ Activation functions provide this non-linearity.
- **Continuously differentiable**—To improve our model with gradient descent, we need our output to have a nice slope so we can compute error derivatives with respect to weights. If our neuron instead outputted 0 or 1 (perceptron), we wouldn't know in which direction to update our weights to reduce our error.
- **Fixed Range**—Activation functions typically squash the input data into a narrow range that makes training the model more stable and efficient.

10.9 Loss Functions

A loss function, or cost function, is a wrapper around our model's predict function that tells us “how good” the model is at making predictions for a given set of parameters. The loss function has its own curve and its own derivatives. The

slope of this curve tells us how to change our parameters to make the model more accurate! We use the model to make predictions. We use the cost function to update our parameters. Our cost function can take a variety of forms as there are many different cost functions available. Popular loss functions include: *MSE (L2)* and *Cross-entropy Loss*.

10.10 Optimization Algorithms

Be the first to [contribute!](#)

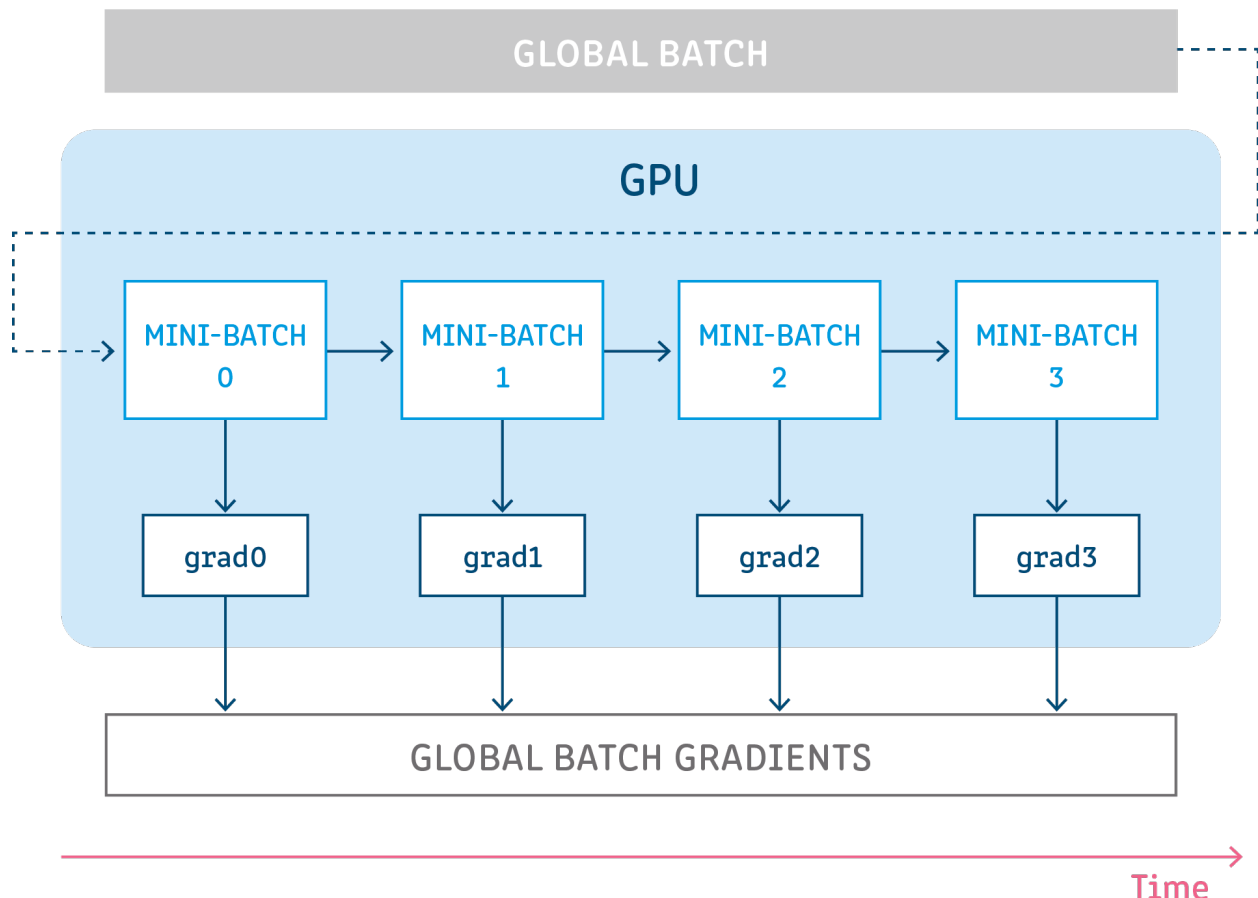
10.11 Gradient Accumulation

Gradient accumulation is a mechanism to split the batch of samples—used for training a neural network—into several mini-batches of samples that will be run sequentially.

This is used to enable using large batch sizes that require more GPU memory than available. Gradient accumulation helps in doing so by using mini-batches that require an amount of GPU memory that can be satisfied.

Gradient accumulation means running all mini-batches sequentially (generally on the same GPU) while accumulating their calculated gradients and not updating the model variables - the weights and biases of the model. The model variables must not be updated during the accumulation in order to ensure all mini-batches use the same model variable values to calculate their gradients. Only after accumulating the gradients of all those mini-batches will we generate and apply the updates for the model variables.

This results in the same updates for the model parameters as if we were to use the global batch.



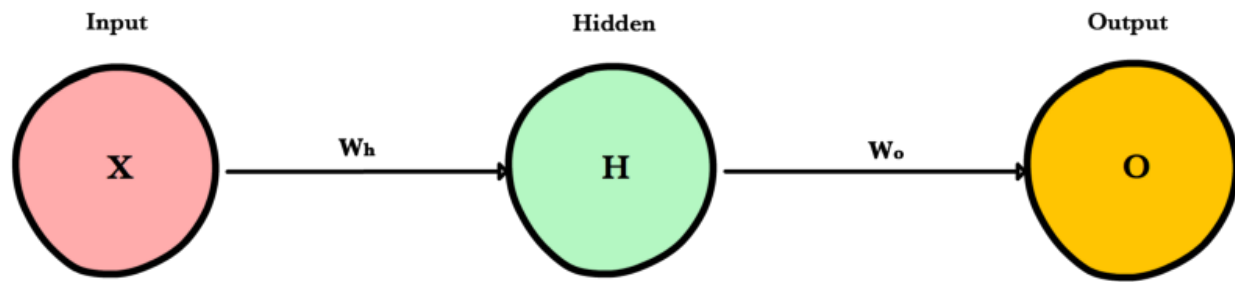
More details, a technical and algorithmical deep-dive, how-to tutorials, and examples can be found at [2].

References

Forwardpropagation

- *Simple Network*
 - *Steps*
 - *Code*
- *Larger Network*
 - *Architecture*
 - *Weight Initialization*
 - *Bias Terms*
 - *Working with Matrices*
 - *Dynamic Resizing*
 - *Refactoring Our Code*
 - *Final Result*

11.1 Simple Network



Forward propagation is how neural networks make predictions. Input data is “forward propagated” through the network layer by layer to the final layer which outputs a prediction. For the toy neural network above, a single pass of forward propagation translates mathematically to:

$$\text{Prediction} = A(A(XW_h)W_o)$$

Where A is an activation function like *ReLU*, X is the input and W_h and W_o are weights.

11.1.1 Steps

1. Calculate the weighted input to the hidden layer by multiplying X by the hidden weight W_h
2. Apply the activation function and pass the result to the final layer
3. Repeat step 2 except this time X is replaced by the hidden layer’s output, H

11.1.2 Code

Let’s write a method `feed_forward()` to propagate input data through our simple network of 1 hidden layer. The output of this method represents our model’s prediction.

```
def relu(z):
    return max(0, z)

def feed_forward(x, Wh, Wo):
    # Hidden layer
    Zh = x * Wh
    H = relu(Zh)

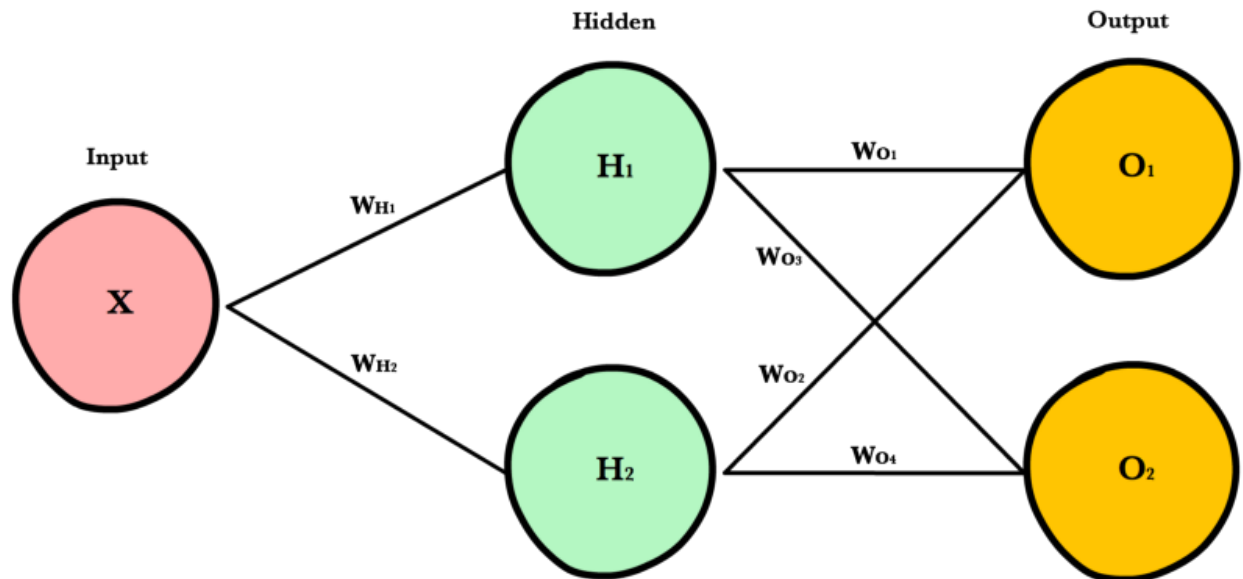
    # Output layer
    Zo = H * Wo
    output = relu(Zo)
    return output
```

x is the input to the network, Z_o and Z_h are the weighted inputs and W_o and W_h are the weights.

11.2 Larger Network

The simple network above is helpful for learning purposes, but in reality neural networks are much larger and more complex. Modern neural networks have many more hidden layers, more neurons per layer, more variables per input,

more inputs per training set, and more output variables to predict. Here is a slightly larger network that will introduce us to matrices and the matrix operations used to train arbitrarily large neural networks.



11.2.1 Architecture

To accommodate arbitrarily large inputs or outputs, we need to make our code more extensible by adding a few parameters to our network's `__init__` method: `inputLayerSize`, `hiddenLayerSize`, `outputLayerSize`. We'll still limit ourselves to using one hidden layer, but now we can create layers of different sizes to respond to the different inputs or outputs.

```
INPUT_LAYER_SIZE = 1
HIDDEN_LAYER_SIZE = 2
OUTPUT_LAYER_SIZE = 2
```

11.2.2 Weight Initialization

Unlike last time where W_h and W_o were scalar numbers, our new weight variables will be numpy arrays. Each array will hold all the weights for its own layer—one weight for each synapse. Below we initialize each array with the numpy's `np.random.randn(rows, cols)` method, which returns a matrix of random numbers drawn from a normal distribution with mean 0 and variance 1.

```
def init_weights():
    Wh = np.random.randn(INPUT_LAYER_SIZE, HIDDEN_LAYER_SIZE) * \
        np.sqrt(2.0/INPUT_LAYER_SIZE)
    Wo = np.random.randn(HIDDEN_LAYER_SIZE, OUTPUT_LAYER_SIZE) * \
        np.sqrt(2.0/HIDDEN_LAYER_SIZE)
```

Here's an example calling `random.randn()`:

```
arr = np.random.randn(1, 2)

print(arr)
>> [[-0.36094661 -1.30447338]]
```

(continues on next page)

(continued from previous page)

```
print(arr.shape)
>> (1,2)
```

As you'll soon see, there are strict requirements on the dimensions of these weight matrices. The number of *rows* must equal the number of neurons in the previous layer. The number of *columns* must match the number of neurons in the next layer.

A good explanation of random weight initialization can be found in the Stanford CS231 course notes¹ chapter on neural networks.

11.2.3 Bias Terms

Bias terms allow us to shift our neuron's activation outputs left and right. This helps us model datasets that do not necessarily pass through the origin.

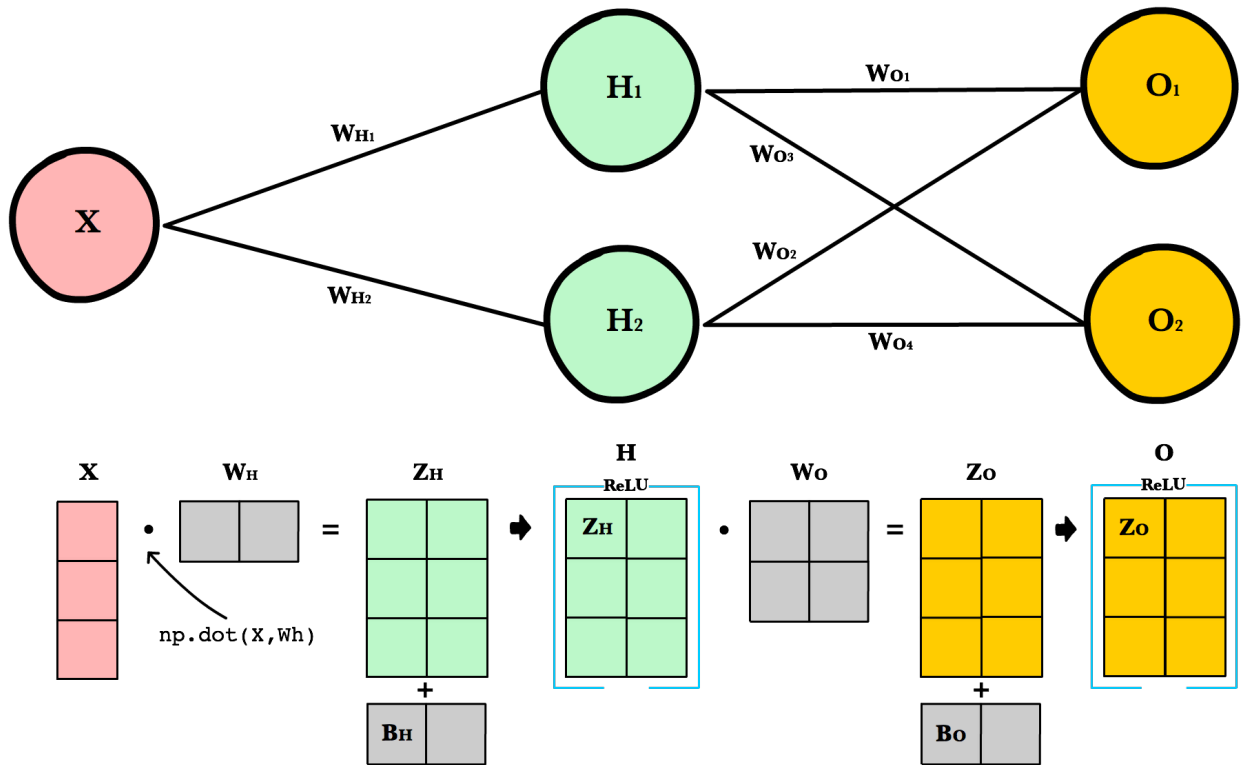
Using the numpy method `np.full()` below, we create two 1-dimensional bias arrays filled with the default value `0.2`. The first argument to `np.full` is a tuple of array dimensions. The second is the default value for cells in the array.

```
def init_bias():
    Bh = np.full((1, HIDDEN_LAYER_SIZE), 0.1)
    Bo = np.full((1, OUTPUT_LAYER_SIZE), 0.1)
    return Bh, Bo
```

11.2.4 Working with Matrices

To take advantage of fast linear algebra techniques and GPUs, we need to store our inputs, weights, and biases in matrices. Here is our neural network diagram again with its underlying matrix representation.

¹ <http://cs231n.github.io/neural-networks-2/#init>



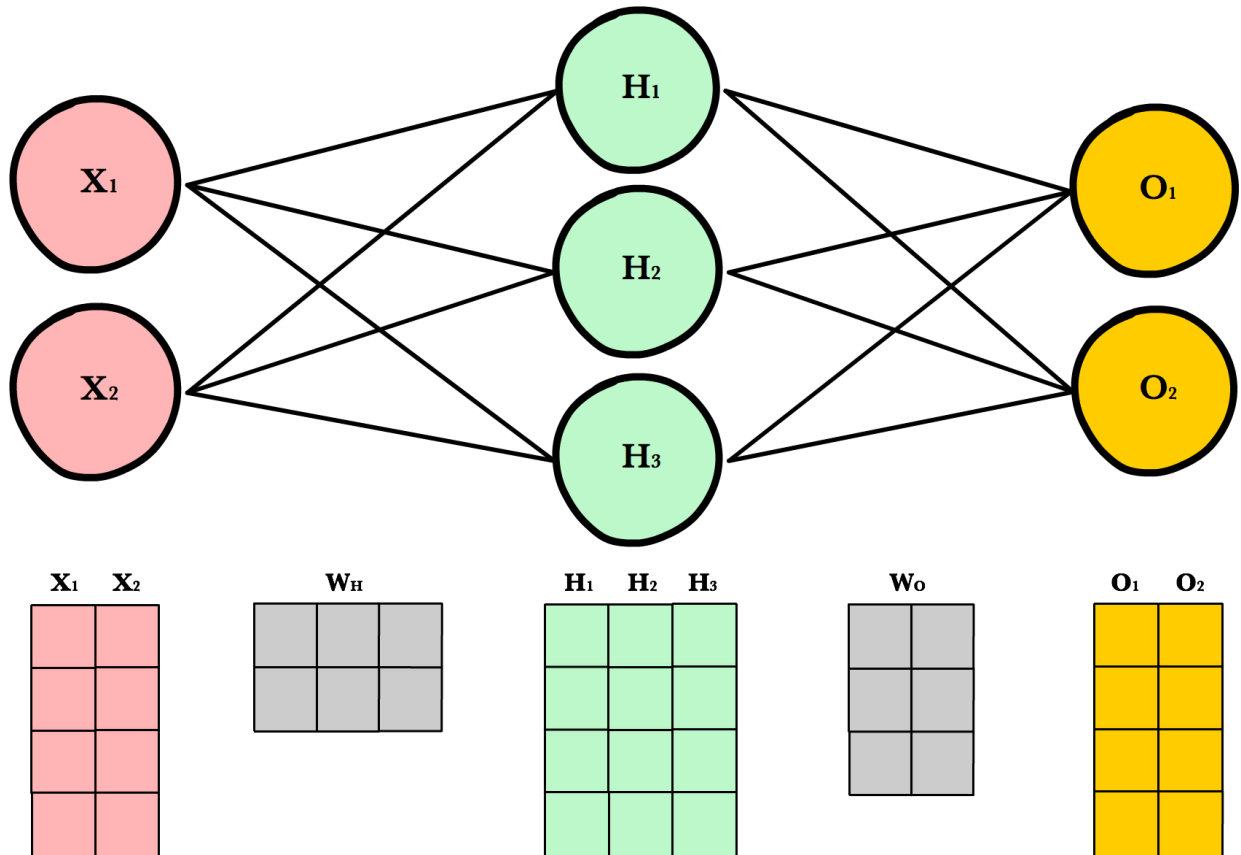
What's happening here? To better understand, let's walk through each of the matrices in the diagram with an emphasis on their dimensions and why the dimensions are what they are. The matrix dimensions above flow naturally from the architecture of our network and the number of samples in our training set.

Matrix dimensions

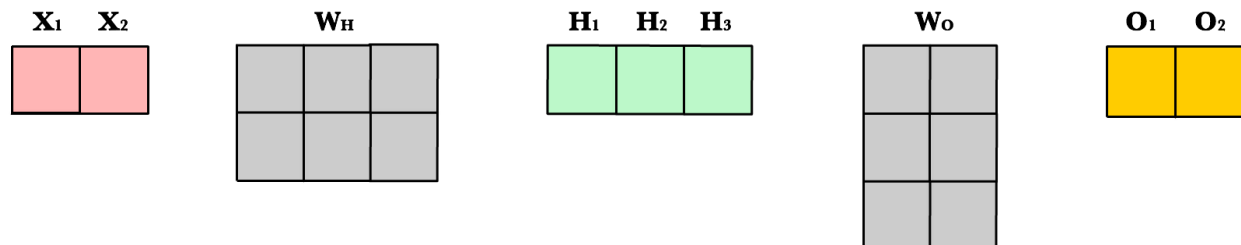
Var	Name	Dimensions	Explanation
X	Input	(3, 1)	Includes 3 rows of training data, and each row has 1 attribute (height, price, etc.)
Wh	Hidden weights	(1, 2)	These dimensions are based on number of rows equals the number of attributes for the observations in our training set. The number columns equals the number of neurons in the hidden layer. The dimensions of the weights matrix between two layers is determined by the sizes of the two layers it connects. There is one weight for every input-to-neuron connection between the layers.
Bh	Hidden bias	(1, 2)	Each neuron in the hidden layer has its own bias constant. This bias matrix is added to the weighted input matrix before the hidden layer applies ReLU.
Zh	Hidden weighted input	(1, 2)	Computed by taking the dot product of X and Wh. The dimensions (1,2) are required by the rules of matrix multiplication. Zh takes the rows of in the inputs matrix and the columns of weights matrix. We then add the hidden layer bias matrix Bh.
H	Hidden activations	(3, 2)	Computed by applying the Relu function to Zh. The dimensions are (3,2)—the number of rows matches the number of training samples and the number of columns equals the number of neurons. Each column holds all the activations for a specific neuron.
Wo	Output weights	(2, 2)	The number of rows matches the number of hidden layer neurons and the number of columns equals the number of output layer neurons. There is one weight for every hidden-neuron-to-output-neuron connection between the layers.
Bo	Output bias	(1, 2)	There is one column for every neuron in the output layer.
Zo	Output weighted input	(3, 2)	Computed by taking the dot product of H and Wo and then adding the output layer bias Bo. The dimensions are (3,2) representing the rows of in the hidden layer matrix and the columns of output layer weights matrix.
O	Output activations	(3, 2)	Each row represents a prediction for a single observation in our training set. Each column is a unique attribute we want to predict. Examples of two-column output predictions could be a company's sales and units sold, or a person's height and weight.

11.2.5 Dynamic Resizing

Before we continue I want to point out how the matrix dimensions change with changes to the network architecture or size of the training set. For example, let's build a network with 2 input neurons, 3 hidden neurons, 2 output neurons, and 4 observations in our training set.



Now let's use same number of layers and neurons but reduce the number of observations in our dataset to **1 instance**:



As you can see, the number of columns in all matrices remains the same. The only thing that changes is the number of rows the layer matrices, which fluctuate with the size of the training set. The dimensions of the weight matrices remain unchanged. This shows us we can use the same network, the same lines of code, to process any number of observations.

11.2.6 Refactoring Our Code

Here is our new feed forward code which accepts matrices instead of scalar inputs.

```
def feed_forward(X):
    '''
    X      - input matrix
    Zh     - hidden layer weighted input
    Zo     - output layer weighted input
    H      - hidden layer activation
```

(continues on next page)

(continued from previous page)

```

y      - output layer
yHat   - output layer predictions
'''

# Hidden layer
Zh = np.dot(X, Wh) + Bh
H = relu(Zh)

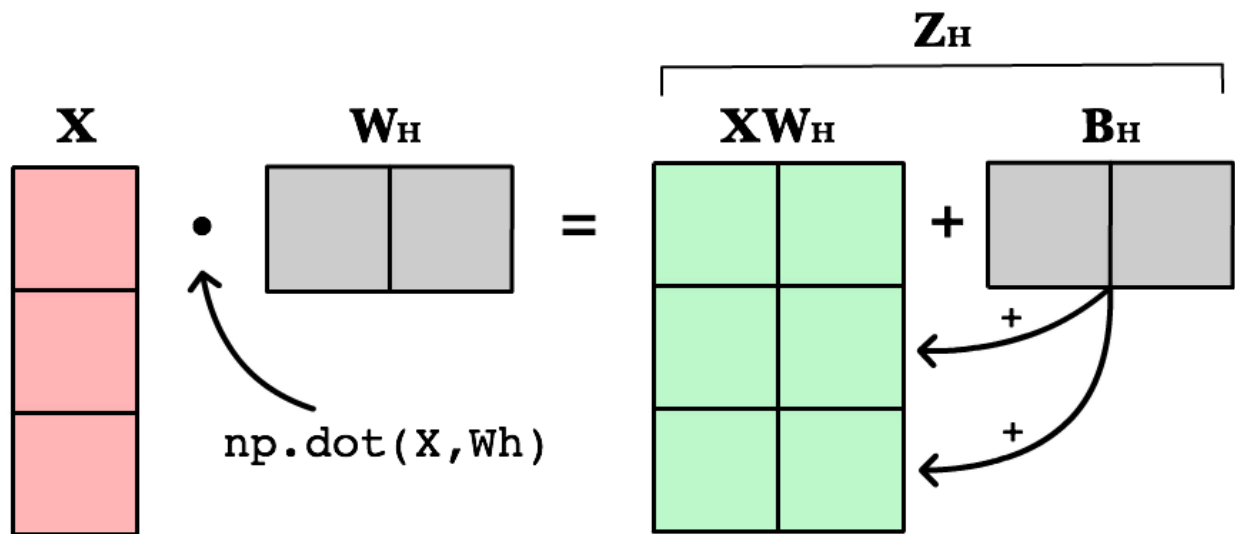
# Output layer
Zo = np.dot(H, Wo) + Bo
yHat = relu(Zo)
return yHat

```

Weighted input

The first change is to update our weighted input calculation to handle matrices. Using dot product, we multiply the input matrix by the weights connecting them to the neurons in the next layer. Next we add the bias vector using matrix addition.

```
Zh = np.dot(X, Wh) + Bh
```



The first column in B_H is added to all the rows in the first column of resulting dot product of X and W_H . The second value in B_H is added to all the elements in the second column. The result is a new matrix, Z_H which has a column for every neuron in the hidden layer and a row for every observation in our dataset. Given all the layers in our network are *fully-connected*, there is one weight for every neuron-to-neuron connection between the layers.

The same process is repeated for the output layer, except the input is now the hidden layer activation H and the weights W_O .

ReLU activation

The second change is to refactor ReLU to use elementwise multiplication on matrices. It's only a small change, but it's necessary if we want to work with matrices. `np.maximum()` is actually extensible and can handle both scalar and array inputs.

```
def relu(Z):
    return np.maximum(0, Z)
```

In the hidden layer activation step, we apply the ReLU activation function `np.maximum(0, Z)` to every cell in the new matrix. The result is a matrix where all negative values have been replaced by 0. The same process is repeated for the output layer, except the input is `Zo`.

11.2.7 Final Result

Putting it all together we have the following code for forward propagation with matrices.

```
INPUT_LAYER_SIZE = 1
HIDDEN_LAYER_SIZE = 2
OUTPUT_LAYER_SIZE = 2

def init_weights():
    Wh = np.random.randn(INPUT_LAYER_SIZE, HIDDEN_LAYER_SIZE) * \
        np.sqrt(2.0/INPUT_LAYER_SIZE)
    Wo = np.random.randn(HIDDEN_LAYER_SIZE, OUTPUT_LAYER_SIZE) * \
        np.sqrt(2.0/HIDDEN_LAYER_SIZE)

def init_bias():
    Bh = np.full((1, HIDDEN_LAYER_SIZE), 0.1)
    Bo = np.full((1, OUTPUT_LAYER_SIZE), 0.1)
    return Bh, Bo

def relu(Z):
    return np.maximum(0, Z)

def relu_prime(Z):
    '''
    Z - weighted input matrix

    Returns gradient of Z where all
    negative values are set to 0 and
    all positive values set to 1
    '''
    Z[Z < 0] = 0
    Z[Z > 0] = 1
    return Z

def cost(yHat, y):
    cost = np.sum((yHat - y)**2) / 2.0
    return cost

def cost_prime(yHat, y):
    return yHat - y

def feed_forward(X):
    '''
    X - input matrix
    Zh - hidden layer weighted input
    Zo - output layer weighted input
    H - hidden layer activation
    y - output layer
```

(continues on next page)

(continued from previous page)

```
yHat - output layer predictions
'''

# Hidden layer
Zh = np.dot(X, Wh) + Bh
H = relu(Zh)

# Output layer
Zo = np.dot(H, Wo) + Bo
yHat = relu(Zo)
```

References

- *Chain rule refresher*
- *Applying the chain rule*
- *Saving work with memoization*
- *Code example*

The goals of backpropagation are straightforward: adjust each weight in the network in proportion to how much it contributes to overall error. If we iteratively reduce each weight's error, eventually we'll have a series of weights that produce good predictions.

12.1 Chain rule refresher

As seen above, forward propagation can be viewed as a long series of nested equations. If you think of feed forward this way, then backpropagation is merely an application of *Chain rule* to find the *Derivatives* of cost with respect to any variable in the nested equation. Given a forward propagation function:

$$f(x) = A(B(C(x)))$$

A, B, and C are activation functions at different layers. Using the chain rule we easily calculate the derivative of $f(x)$ with respect to x :

$$f'(x) = f'(A) \cdot A'(B) \cdot B'(C) \cdot C'(x)$$

How about the derivative with respect to B? To find the derivative with respect to B you can pretend $B(C(x))$ is a constant, replace it with a placeholder variable B, and proceed to find the derivative normally with respect to B.

$$f'(B) = f'(A) \cdot A'(B)$$

This simple technique extends to any variable within a function and allows us to precisely pinpoint the exact impact each variable has on the total output.

12.2 Applying the chain rule

Let's use the chain rule to calculate the derivative of cost with respect to any weight in the network. The chain rule will help us identify how much each weight contributes to our overall error and the direction to update each weight to reduce our error. Here are the equations we need to make a prediction and calculate total error, or cost:

Function	Formula	Derivative
Weighted input	$Z = XW$	$Z'(X) = W$ $Z'(W) = X$
ReLU activation	$R = \max(0, Z)$	$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$
Cost function	$C = \frac{1}{2}(\hat{y} - y)^2$	$C'(\hat{y}) = (\hat{y} - y)$

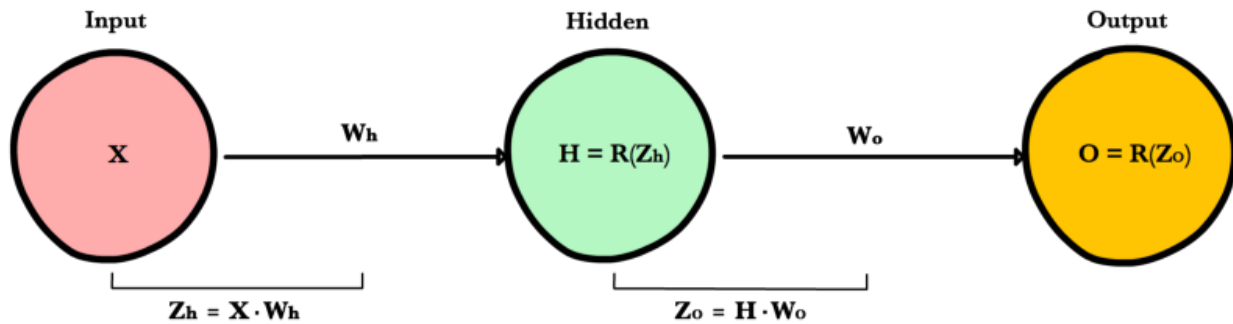
Given a network consisting of a single neuron, total cost could be calculated as:

$$Cost = C(R(Z(XW)))$$

Using the chain rule we can easily find the derivative of Cost with respect to weight W .

$$\begin{aligned} C'(W) &= C'(R) \cdot R'(Z) \cdot Z'(W) \\ &= (\hat{y} - y) \cdot R'(Z) \cdot X \end{aligned}$$

Now that we have an equation to calculate the derivative of cost with respect to any weight, let's go back to our toy neural network example above



What is the derivative of cost with respect to W_o ?

$$\begin{aligned} C'(W_o) &= C'(\hat{y}) \cdot \hat{y}'(Z_o) \cdot Z'_o(W_o) \\ &= (\hat{y} - y) \cdot R'(Z_o) \cdot H \end{aligned}$$

And how about with respect to W_h ? To find out we just keep going further back in our function applying the chain rule recursively until we get to the function that has the W_h term.

$$\begin{aligned} C'(W_h) &= C'(\hat{y}) \cdot O'(Z_o) \cdot Z'_o(H) \cdot H'(Z_h) \cdot Z'_h(W_h) \\ &= (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h) \cdot X \end{aligned}$$

And just for fun, what if our network had 10 hidden layers. What is the derivative of cost for the first weight w_1 ?

$$C'(w_1) = \frac{dC}{d\hat{y}} \cdot \frac{d\hat{y}}{dZ_{11}} \cdot \frac{dZ_{11}}{dH_{10}} \cdot \frac{dH_{10}}{dZ_{10}} \cdot \frac{dZ_{10}}{dH_9} \cdot \frac{dH_9}{dZ_9} \cdot \frac{dZ_9}{dH_8} \cdot \frac{dH_8}{dZ_8} \cdot \frac{dZ_8}{dH_7} \cdot \frac{dH_7}{dZ_7} \cdot \frac{dZ_7}{dH_6} \cdot \frac{dH_6}{dZ_6} \cdot \frac{dZ_6}{dH_5} \cdot \frac{dH_5}{dZ_5} \cdot \frac{dZ_5}{dH_4} \cdot \frac{dH_4}{dZ_4} \cdot \frac{dZ_4}{dH_3} \cdot \frac{dH_3}{dZ_3} \cdot \frac{dZ_3}{dH_2} \cdot \frac{dH_2}{dZ_2} \cdot \frac{dZ_2}{dH_1} \cdot \frac{dH_1}{dZ_1} \cdot \frac{dZ_1}{dW_1}$$

See the pattern? The number of calculations required to compute cost derivatives increases as our network grows deeper. Notice also the redundancy in our derivative calculations. Each layer's cost derivative appends two new terms to the terms that have already been calculated by the layers above it. What if there was a way to save our work somehow and avoid these duplicate calculations?

12.3 Saving work with memoization

Memoization is a computer science term which simply means: don't recompute the same thing over and over. In memoization we store previously computed results to avoid recalculating the same function. It's handy for speeding up recursive functions of which backpropagation is one. Notice the pattern in the derivative equations below.

$$C'(W_3) = (O - y) \cdot R'(Z_3) \cdot H_2$$

$$C'(W_2) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot H_1$$

$$C'(W_1) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot H_0$$

$$C'(W_0) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot W_1 \cdot R'(Z_0) \cdot X$$

Each of these layers is recomputing the same derivatives! Instead of writing out long derivative equations for every weight, we can use memoization to save our work as we backprop error through the network. To do this, we define 3 equations (below), which together encapsulate all the calculations needed for backpropagation. The math is the same, but the equations provide a nice shorthand we can use to track which calculations we've already performed and save our work as we move backwards through the network.

Function	Formula	Derivative
Weighted input	$Z = XW$	$Z'(X) = W$ $Z'(W) = X$
ReLU activation	$R = \max(0, Z)$	$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$
Cost function	$C = \frac{1}{2}(\hat{y} - y)^2$	$C'(\hat{y}) = (\hat{y} - y)$

We first calculate the output layer error and pass the result to the hidden layer before it. After calculating the hidden layer error, we pass its error value back to the previous hidden layer before it. And so on and so forth. As we move back through the network we apply the 3rd formula at every layer to calculate the derivative of cost with respect that layer's weights. This resulting derivative tells us in which direction to adjust our weights to reduce overall cost.

Note: The term *layer error* refers to the derivative of cost with respect to a layer's *input*. It answers the question: how does the cost function output change when the input to that layer changes?

Output layer error

To calculate output layer error we need to find the derivative of cost with respect to the output layer input, Z_o . It answers the question—how are the final layer's weights impacting overall error in the network? The derivative is then:

$$C'(Z_o) = (\hat{y} - y) \cdot R'(Z_o)$$

To simplify notation, ml practitioners typically replace the $(\hat{y} - y) \cdot R'(Z_o)$ sequence with the term E_o . So our formula for output layer error equals:

$$E_o = (\hat{y} - y) \cdot R'(Z_o)$$

Hidden layer error

To calculate hidden layer error we need to find the derivative of cost with respect to the hidden layer input, Z_h .

$$C'(Z_h) = (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h)$$

Next we can swap in the E_o term above to avoid duplication and create a new simplified equation for Hidden layer error:

$$E_h = E_o \cdot W_o \cdot R'(Z_h)$$

This formula is at the core of backpropagation. We calculate the current layer's error, and pass the weighted error back to the previous layer, continuing the process until we arrive at our first hidden layer. Along the way we update the weights using the derivative of cost with respect to each weight.

Derivative of cost with respect to any weight

Let's return to our formula for the derivative of cost with respect to the output layer weight W_o .

$$C'(W_o) = (\hat{y} - y) \cdot R'(Z_o) \cdot H$$

We know we can replace the first part with our equation for output layer error E_o . H represents the hidden layer activation.

$$C'(W_o) = E_o \cdot H$$

So to find the derivative of cost with respect to any weight in our network, we simply multiply the corresponding layer's error times its input (the previous layer's output).

$$C'(w) = \text{CurrentLayerError} \cdot \text{CurrentLayerInput}$$

Note: *Input* refers to the activation from the previous layer, not the weighted input, Z .

Summary

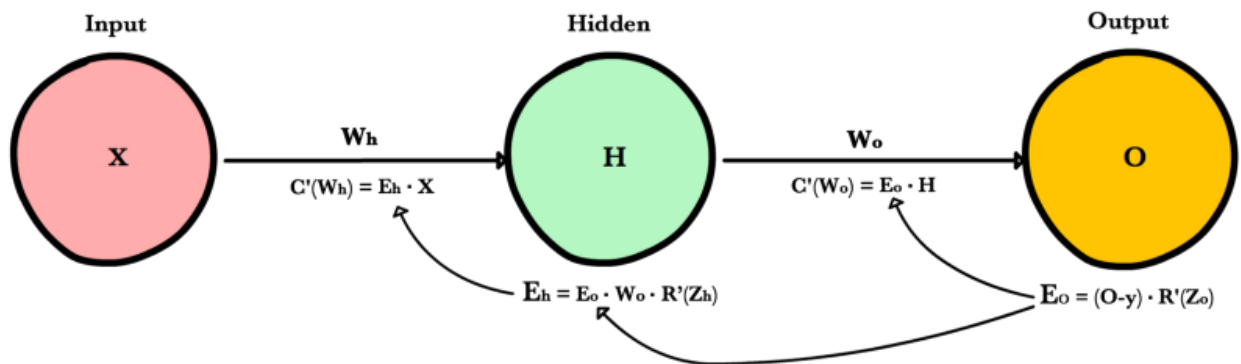
Here are the final 3 equations that together form the foundation of backpropagation.

$$\text{Output Layer Error} \quad E_o = (O - y) \cdot R'(Z_o)$$

$$\text{Hidden Layer Error} \quad E_h = E_o \cdot W_o \cdot R'(Z_h)$$

$$\text{Cost-Weights Deriv} \quad \text{LayerError} \cdot \text{LayerInput}$$

Here is the process visualized using our toy neural network example above.



12.4 Code example

```
def relu_prime(z):
    if z > 0:
        return 1
    return 0

def cost(yHat, y):
    return 0.5 * (yHat - y)**2

def cost_prime(yHat, y):
    return yHat - y

def backprop(x, y, Wh, Wo, lr):
    yHat = feed_forward(x, Wh, Wo)

    # Layer Error
    Eo = (yHat - y) * relu_prime(Zo)
    Eh = Eo * Wo * relu_prime(Zh)

    # Cost derivative for weights
    dWo = Eo * H
    dWh = Eh * x

    # Update weights
```

(continues on next page)

(continued from previous page)

```
Wh -= lr * dWh  
Wo -= lr * dWo
```

References

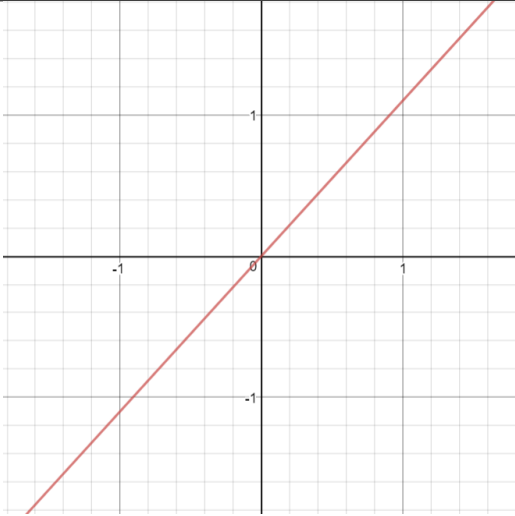
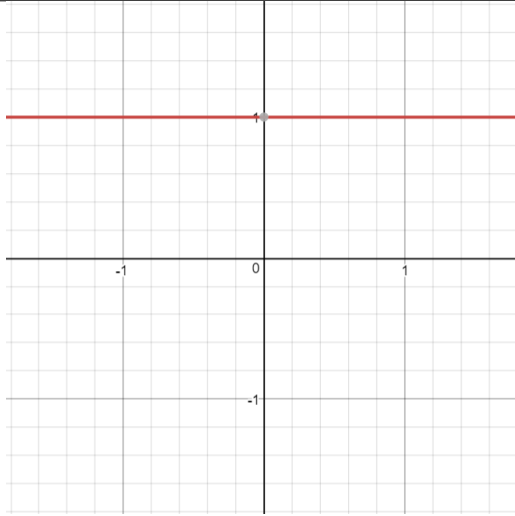
CHAPTER 13

Activation Functions

- *Linear*
- *ELU*
- *ReLU*
- *LeakyReLU*
- *Sigmoid*
- *Tanh*
- *Softmax*

13.1 Linear

A straight line function where activation is proportional to input (which is the weighted sum from neuron).

Function	Derivative
$R(z, m) = \{z * m\}$	$R'(z, m) = \{m\}$
	
<pre>def linear(z, m): return m * z</pre>	<pre>def linear_prime(z, m): return m</pre>

Pros

- It gives a range of activations, so it is not binary activation.
- We can definitely connect a few neurons together and if more than 1 fires, we could take the max (or softmax) and decide based on that.

Cons

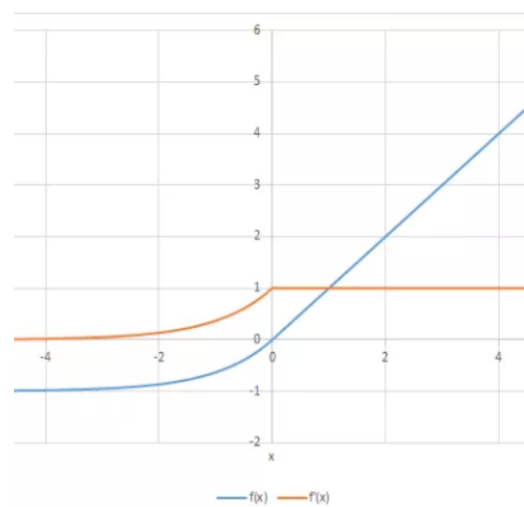
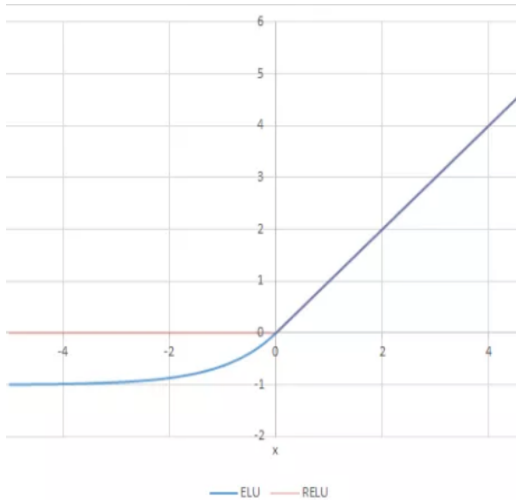
- For this function, derivative is a constant. That means, the gradient has no relationship with X.
- It is a constant gradient and the descent is going to be on constant gradient.
- If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input delta(x) !

13.2 ELU

Exponential Linear Unit or its widely known name ELU is a function that tend to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has a extra alpha constant which should be positive number.

ELU is very similar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to $-\alpha$ whereas RELU sharply smooths.

Function	Derivative
$\frac{1}{1 + e^{-z}}$	$z > 0$
$\frac{1}{1 + e^{-z}}$	$z \leq 0$
$R'(z) = \begin{cases} 1 & z > 0 \\ e^z & z \leq 0 \end{cases}$	



```
def elu(z, alpha):
    return z if z >= 0 else alpha*(e^z - 1)

def elu_prime(z, alpha):
    return 1 if z > 0 else alpha*np.exp(z)
```

Pros

- ELU becomes smooth slowly until its output equal to $-\alpha$ whereas ReLU sharply smooths.
- ELU is a strong alternative to ReLU.
- Unlike to ReLU, ELU can produce negative outputs.

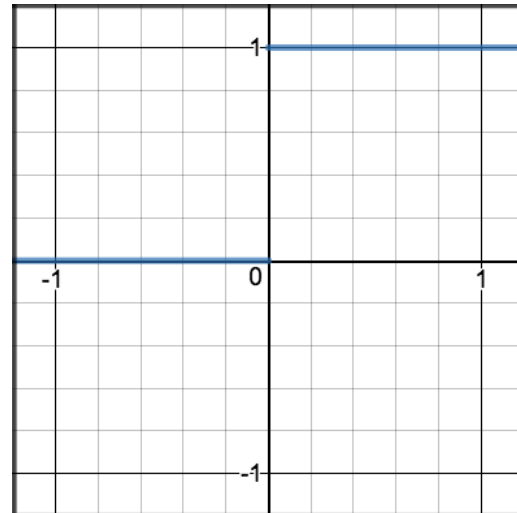
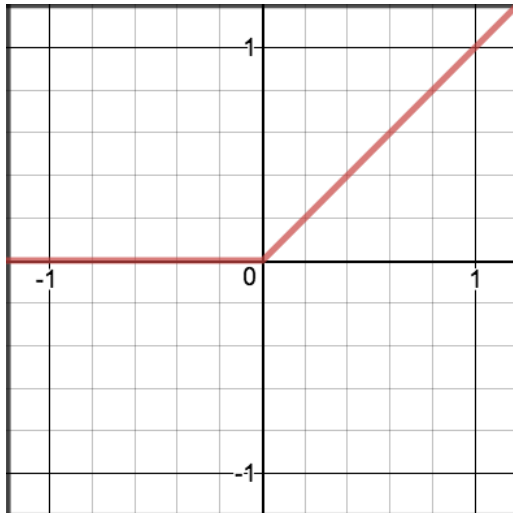
Cons

- For $x > 0$, it can blow up the activation with the output range of $[0, \infty]$.

13.3 ReLU

A recent invention which stands for Rectified Linear Units. The formula is deceptively simple: $\max(0, z)$. Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid (i.e. the ability to learn nonlinear functions), but with better performance.

Function	Derivative
0	$z > 0$
	$z \leq 0$
$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$	



```
def relu(z):
    return max(0, z)

def relu_prime(z):
    return 1 if z > 0 else 0
```

Pros

- It avoids and rectifies vanishing gradient problem.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons

- One of its limitations is that it should only be used within hidden layers of a neural network model.
- Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. In other words, ReLu can result in dead neurons.
- In another words, For activations in the region ($x < 0$) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called the dying ReLu problem.
- The range of ReLu is $[0, \infty)$. This means it can blow up the activation.

Further reading

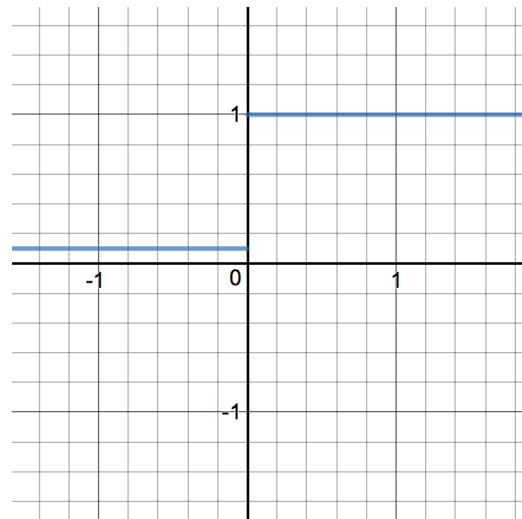
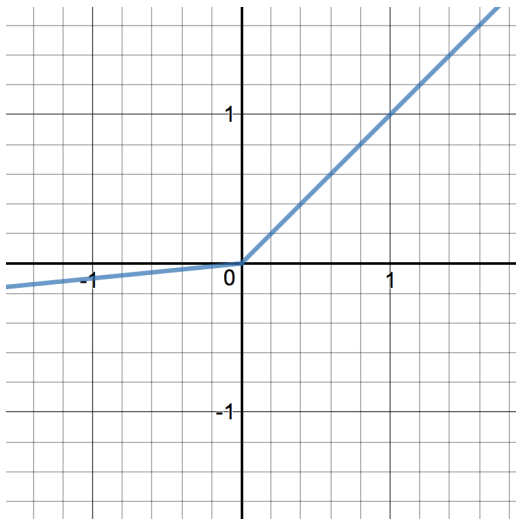
- Deep Sparse Rectifier Neural Networks Glorot et al., (2011)
- Yes You Should Understand Backprop, Karpathy (2016)

13.4 LeakyReLU

LeakyRelu is a variant of ReLU. Instead of being 0 when $z < 0$, a leaky ReLU allows a small, non-zero, constant gradient α (Normally, $\alpha = 0.01$). However, the consistency of the benefit across tasks is presently unclear.¹

Function	Derivative
αz	$z > 0$
	$z \leq 0$

$$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$$



```
def leakyrelu(z, alpha):
    return max(alpha * z, z)

def leakyrelu_prime(z, alpha):
    return 1 if z > 0 else alpha
```

Pros

- Leaky ReLUs are one attempt to fix the “dying ReLU” problem by having a small negative slope (of 0.01, or so).

¹ <http://cs231n.github.io/neural-networks-1/>

Cons

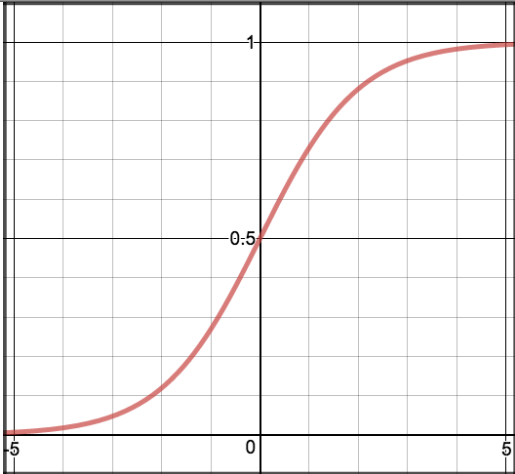
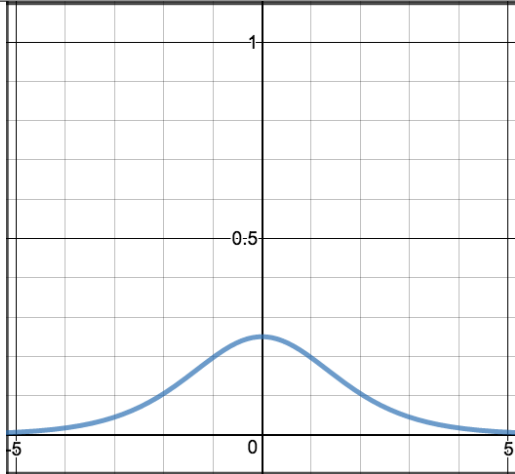
- As it possess linearity, it can't be used for the complex Classification. It lags behind the Sigmoid and Tanh for some of the use cases.

Further reading

- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#), Kaiming He et al. (2015)

13.5 Sigmoid

Sigmoid takes a real value as input and outputs another value between 0 and 1. It's easy to work with and has all the nice properties of activation functions: it's non-linear, continuously differentiable, monotonic, and has a fixed output range.

Function	Derivative
$S(z) = \frac{1}{1 + e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$
	
<pre>def sigmoid(z): return 1.0 / (1 + np.exp(-z))</pre>	<pre>def sigmoid_prime(z): return sigmoid(z) * (1-sigmoid(z))</pre>

Pros

- It is nonlinear in nature. Combinations of this function are also nonlinear!
- It will give an analog activation unlike step function.
- It has a smooth gradient too.
- It's good for a classifier.

- The output of the activation function is always going to be in range (0,1) compared to $(-\infty, \infty)$ of linear function. So we have our activations bound in a range. Nice, it won't blow up the activations then.

Cons

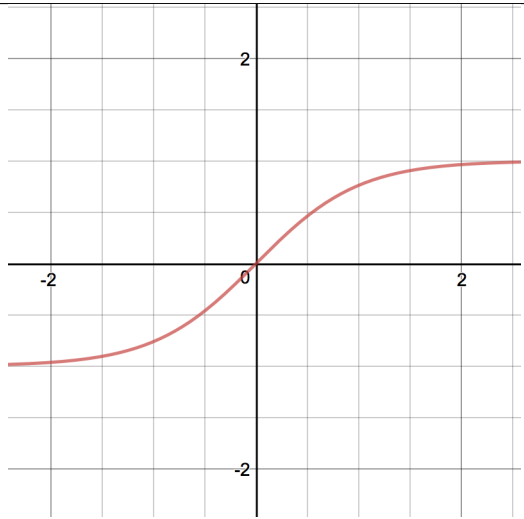
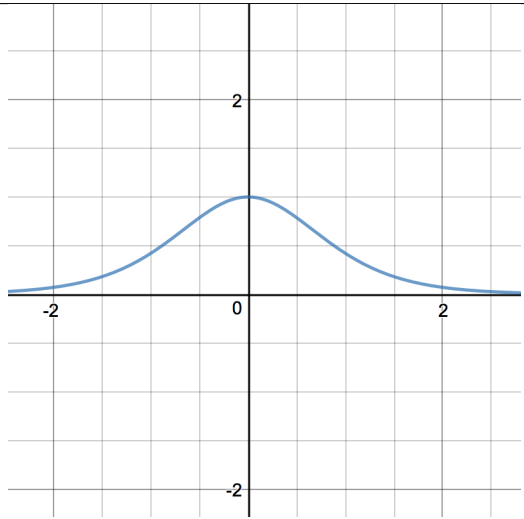
- Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.
- It gives rise to a problem of “vanishing gradients”.
- Its output isn't zero centered. It makes the gradient updates go too far in different directions. $0 < \text{output} < 1$, and it makes optimization harder.
- Sigmoids saturate and kill gradients.
- The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

Further reading

- [Yes You Should Understand Backprop](#), Karpathy (2016)

13.6 Tanh

Tanh squashes a real-valued number to the range $[-1, 1]$. It's non-linear. But unlike Sigmoid, its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.¹

Function	Derivative
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$\tanh'(z) = 1 - \tanh(z)^2$
	
<pre>def tanh(z): return (np.exp(z) - np.exp(-z)) / ↪ (np.exp(z) + np.exp(-z))</pre>	<pre>def tanh_prime(z): return 1 - np.power(tanh(z), 2)</pre>

Pros

- The gradient is stronger for tanh than sigmoid (derivatives are steeper).

Cons

- Tanh also has the vanishing gradient problem.

13.7 Softmax

Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

References

- *BatchNorm*
- *Convolution*
- *Dropout*
- *Pooling*
- *Fully-connected/Linear*
- *RNN*
- *GRU*
- *LSTM*

14.1 BatchNorm

BatchNorm accelerates convergence by reducing internal covariate shift inside each batch. If the individual observations in the batch are widely different, the gradient updates will be choppy and take longer to converge.

The batch norm layer normalizes the incoming activations and outputs a new batch where the mean equals 0 and standard deviation equals 1. It subtracts the mean and divides by the standard deviation of the batch.

Code

Code example from [Agustinus Kristiadi](#)

Further reading

- [Original Paper](#)

- Implementing BatchNorm in Neural Net
- Understanding the backward pass through Batch Norm

14.2 Convolution

In CNN, a convolution is a linear operation that involves multiplication of weight (kernel/filter) with the input and it does most of the heavy lifting job.

Convolution layer consists of 2 major component 1. Kernel(Filter) 2. Stride

1. Kernel (Filter): A convolution layer can have more than one filter. The size of the filter should be smaller than the size of input dimension. It is intentional as it allows filter to be applied multiple times at difference point (position) on the input. Filters are helpful in understanding and identifying important features from given input. By applying different filters (more than one filter) on the same input helps in extracting different features from given input. Output from multiplying filter with the input gives Two dimensional array. As such, the output array from this operation is called “Feature Map”.
2. Stride: This property controls the movement of filter over input. when the value is set to 1, then filter moves 1 column at a time over input. When the value is set to 2 then the filter jump 2 columns at a time as filter moves over the input.

Code

```
# this code demonstate on how Convolution works
# Assume we have a image of 4 X 4 and a filter fo 2 X 2 and Stride = 1

def conv_filter_ouput(input_img_section,filter_value):
    # this method perfomas the multiplication of input and filter
    # returns singular value

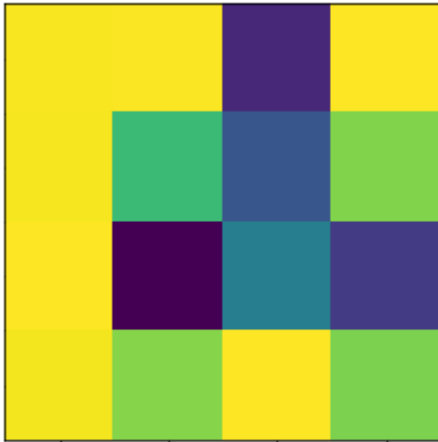
    value = 0
    for i in range(len(filter_value)):
        for j in range(len(filter_value[0])):
            value = value + (input_img_section[i][j]*filter_value[i][j])
    return value

img_input = [[260.745, 261.332, 112.27 , 262.351],
             [260.302, 208.802, 139.05 , 230.709],
             [261.775, 93.73 , 166.118, 122.847],
             [259.56 , 232.038, 262.351, 228.937]]

filter = [[1,0],
          [0,1]]

filterX,filterY = len(filter),len(filter[0])
filtered_result = []
for i in range(0,len(img_mx)-filterX+1):
    clm = []
    for j in range(0,len(img_mx[0])-filterY+1):
        clm.append(conv_filter_ouput(img_mx[i:i+filterX,j:j+filterY],filter))
    filtered_result.append(clm)

print(filtered_result)
```



A. Original input image



B. Image after applying filter

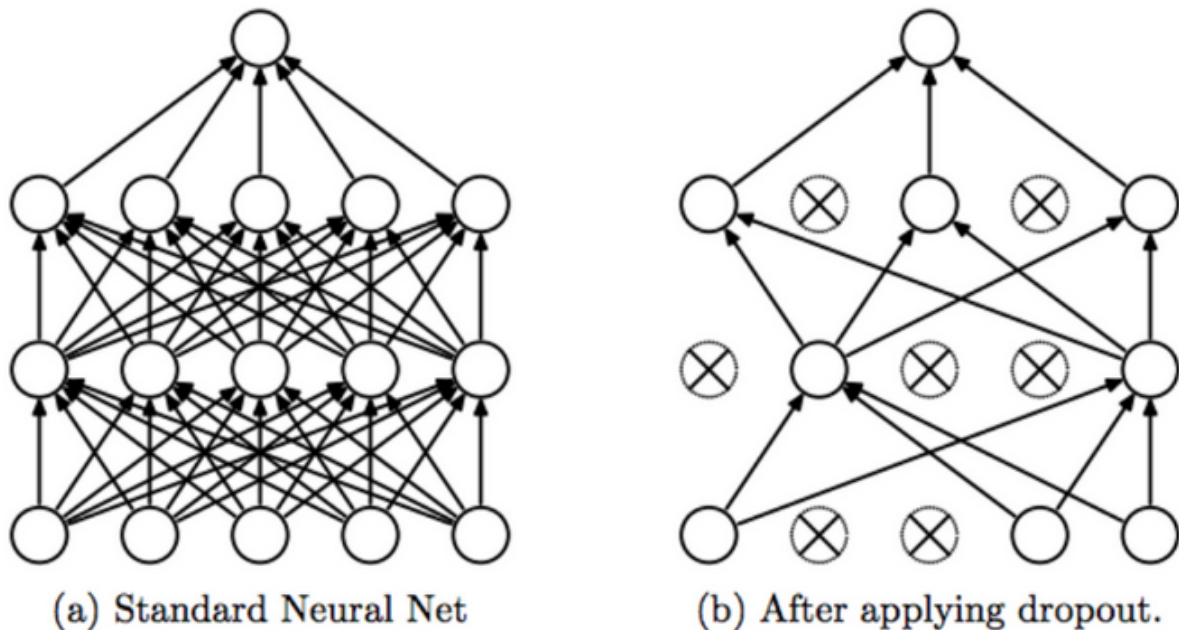
Further reading

- [cs231n reference](#)

14.3 Dropout

A dropout layer takes the output of the previous layer's activations and randomly sets a certain fraction (dropout rate) of the activations to 0, cancelling or 'dropping' them out.

It is a common regularization technique used to prevent overfitting in Neural Networks.



Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

The dropout rate is the tunable hyperparameter that is adjusted to measure performance with different values. It is typically set between 0.2 and 0.5 (but may be arbitrarily set).

Dropout is only used during training; At test time, no activations are dropped, but scaled down by a factor of dropout rate. This is to account for more units being active during test time than training time.

For example:

- A layer in a neural net outputs a tensor (matrix) A of shape $(\text{batch_size}, \text{num_features})$.
- The dropout rate of the layer is set to 0.5 (50%).
- A random 50% of the values in A will be set to 0.
- These will then be multiplied with the weight matrix to form the inputs to the next layer.

The premise behind dropout is to introduce noise into a layer in order to disrupt any interdependent learning or coincidental patterns that may occur between units in the layer, that aren't significant.

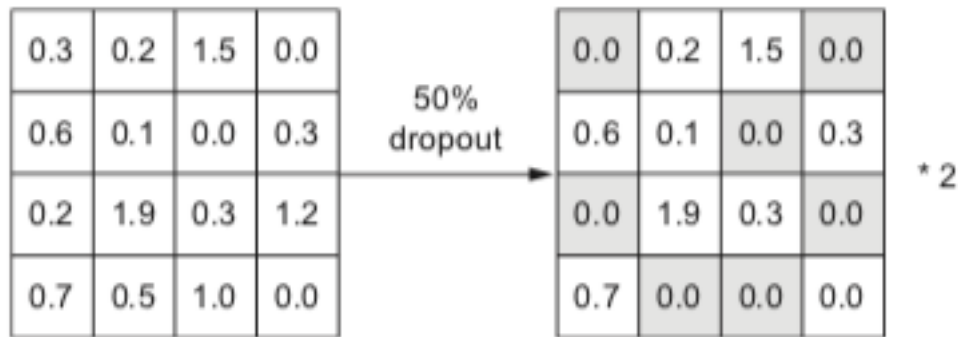
Code

```
# layer_output is a 2D numpy matrix of activations

layer_output *= np.random.randint(0, high=2, size=layer_output.shape) # dropping out
↳ values

# scaling up by dropout rate during TRAINING time, so no scaling needs to be done at
↳ test time
layer_output /= 0.5
# OR
layer_output *= 0.5 # Scaling down during TEST time.
```

This results in the following operation.



All reference, images and code examples, unless mentioned otherwise, are from section 4.4.3 of [Deep Learning for Python](#) by François Chollet.

14.4 Pooling

Pooling layers often take convolution layers as input. A complicated dataset with many object will require a large number of filters, each responsible finding pattern in an image so the dimensionality of convolutional layer can get large. It will cause an increase of parameters, which can lead to over-fitting. Pooling layers are methods for reducing this high dimensionality. Just like the convolution layer, there is kernel size and stride. The size of the kernel is smaller than the feature map. For most of the cases the size of the kernel will be 2X2 and the stride of 2. There are mainly two types of pooling layers.

The first type is max pooling layer. Max pooling layer will take a stack of feature maps (convolution layer) as input. The value of the node in the max pooling layer is calculated by just the maximum of the pixels contained in the window.

The other type of pooling layer is the Average Pooling layer. Average pooling layer calculates the average of pixels contained in the window. Its not used often but you may see this used in applications for which smoothing an image is preferable.

Code

```
def max_pooling(feature_map, size=2, stride=2):
    """
    :param feature_map: Feature matrix of shape (height, width, layers)
    :param size: size of kernel
    :param stride: movement speed of kernel
    :return: max-pooled feature vector
    """
    pool_shape = (feature_map.shape[0]//stride, feature_map.shape[1]//stride, feature_
    ↪map.shape[-1]) #shape of output
    pool_out = numpy.zeros(pool_shape)
    for layer in range(feature_map.shape[-1]):
        #for each layer
        row = 0
        for r in numpy.arange(0, feature_map.shape[0], stride):
            col = 0
            for c in numpy.arange(0, feature_map.shape[1], stride):
                pool_out[row, col, layer] = numpy.max([feature_map[c:c+size,
                ↪r:r+size, layer]])
```

(continues on next page)

(continued from previous page)

```

        col = col + 1
        row = row + 1
    return pool_out

```

Convolution Layer

1	2	3	4
5	1	3	8
5	1	4	7
9	5	8	2



Max Pool Output

5	8
9	8

14.5 Fully-connected/Linear

In a neural network, a *fully-connected layer*, also known as *linear layer*, is a type of layer where all the inputs from one layer are connected to every activation unit of the next layer. In most popular machine learning models, the last few layers in the network are fully-connected ones. Indeed, this type of layer performs the task of outputting a class prediction, based on the features learned in the previous layers.

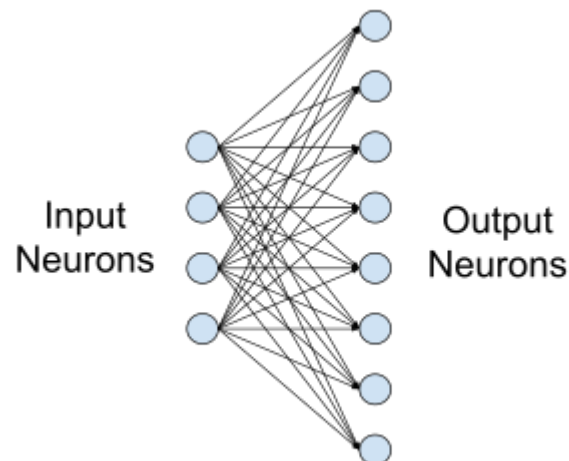


Fig. 1: Example of a fully-connected layer, with four input nodes and eight output nodes. Source [4].

The fully-connected layer receives in input a vector of nodes, activated in the previous convolutional layers. This

vector passes through one or more dense layers, before being sent to the output layer. Before it reaches the output layer, an activation function is used for making a prediction. While the convolutional and pooling layers generally use a ReLU function, the fully-connected layer can use *two types* of activation functions, based on the type of the classification problem:

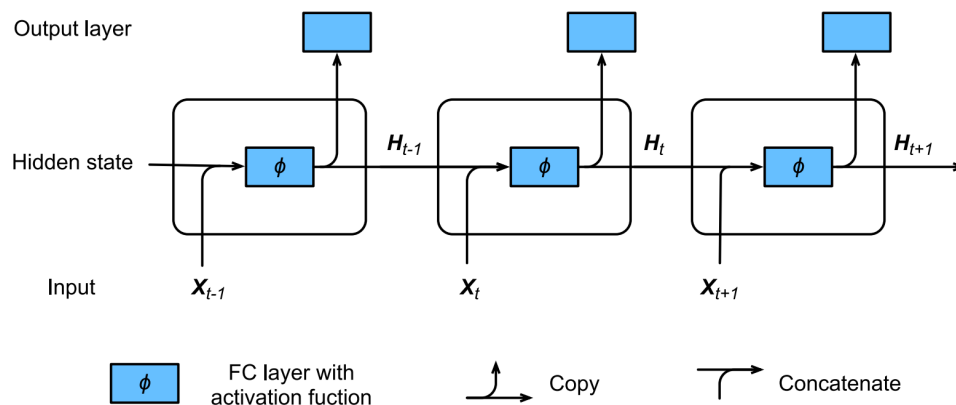
- **Sigmoid:** A logistic function, used for binary classification problems.
- **Softmax:** A more generalized logistic activation function, it ensures that the values in the output layer sum up to 1. Commonly used for multi-class classification.

The activation function outputs a vector whose dimension is equal to the number of classes to be predicted. The output vector yields a probability from 1 to 0 for each class.

14.6 RNN

RNN (Recurrent Neural Network) is the neural network with hidden state, which captures the historical information up to current timestep. Because the hidden state of current state uses the same definition as that in previous timestep, which means the computation is recurrent, hence it is called recurrent neural network.(Ref 2)

The structure is as follows:



Code

For detail code, refer to [layers.py](#)

```
class RNN:
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int, batch_
    size=1) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.out_dim = output_dim
        self.batch_size = batch_size
        # initialization
        self.params = self._init_params()
        self.hidden_state = self._init_hidden_state()

    def _init_params(self) -> List[np.array]:
        scale = 0.01
        Waa = np.random.normal(scale=scale, size=[self.hidden_dim, self.hidden_dim])
        Wax = np.random.normal(scale=scale, size=[self.hidden_dim, self.input_dim])
```

(continues on next page)

(continued from previous page)

```

        Wy = np.random.normal(scale=scale, size=[self.out_dim, self.hidden_dim])
        ba = np.zeros(shape=[self.hidden_dim, 1])
        by = np.zeros(shape=[self.out_dim, 1])
        return [Waa, Wax, Wy, ba, by]

    def _init_hidden_state(self) -> np.array:
        return np.zeros(shape=[self.hidden_dim, self.batch_size])

    def forward(self, input_vector: np.array) -> np.array:
        """
        input_vector:
            dimension: [num_steps, self.input_dim, self.batch_size]
        out_vector:
            dimension: [num_steps, self.output_dim, self.batch_size]
        """
        Waa, Wax, Wy, ba, by = self.params
        output_vector = []
        for vector in input_vector:
            self.hidden_state = np.tanh(
                np.dot(Waa, self.hidden_state) + np.dot(Wax, vector) + ba
            )
            y = softmax(
                np.dot(Wy, self.hidden_state) + by
            )
            output_vector.append(y)
        return np.array(output_vector)

if __name__ == "__main__":
    input_data = np.array([
        [
            [1, 3]
            , [2, 4]
            , [3, 6]
        ]
        , [
            [4, 3]
            , [3, 4]
            , [1, 5]
        ]
    ])
    batch_size = 2
    input_dim = 3
    output_dim = 4
    hidden_dim = 5
    time_step = 2
    rnn = RNN(input_dim=input_dim, batch_size=batch_size, output_dim=output_dim,
    ↪hidden_dim=hidden_dim)
    output_vector = rnn.forward(input_vector=input_data)
    print("RNN:")
    print(f"Input data dimensions: {input_data.shape}")
    print(f"Output data dimensions {output_vector.shape}")
    ## We will get the following output:
    ## RNN:
    ## Input data dimensions: (2, 3, 2)
    ## Output data dimensions (2, 4, 2)

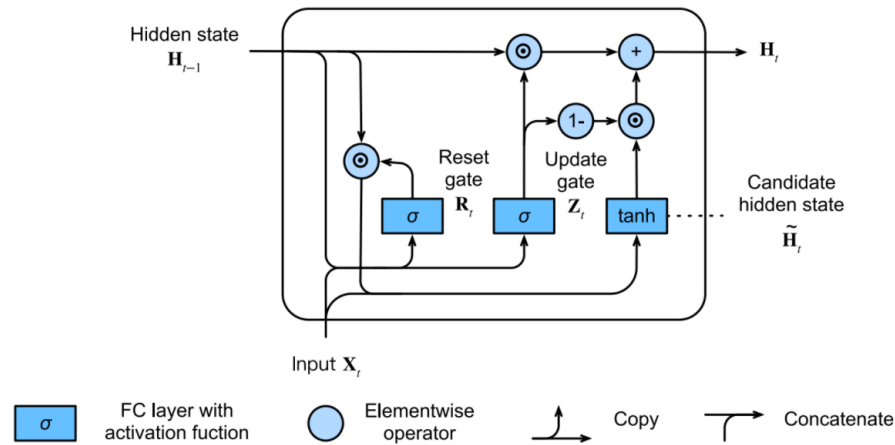
```

14.7 GRU

GRU (Gated Recurrent Unit) supports the gating of hidden state:

1. Reset gate controls how much of previous hidden state we might still want to remember
2. Update gate controls how much of current hidden state is just a copy of previous state

The structure and math are as follow:



1. Reset gate: $R_t = \text{sigmoid}(W_r[\frac{h^{<t-1>}}{x_t}] + b_r)$
2. Update gate: $Z_t = \text{sigmoid}(W_z[\frac{h^{<t-1>}}{x_t}] + b_u)$
3. Candidate hidden state: $\tilde{h}^{<t>} = \tanh(R_t * h^{<t-1>})$
4. Hidden state: $h^{<t>} = Z_t * h^{<t>} + (1 - Z_t) * \tilde{h}^{<t>}$
5. Output layer: $\hat{y}_t = \text{softmax}(W_y a^{<t>} + b_y)$

Code

For detail code, refer to [layers.py](#)

```
class GRU:
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int, batch_
    size=1) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.out_dim = output_dim
        self.batch_size = batch_size
        # initialization
        self.params = self._init_params()
        self.hidden_state = self._init_hidden_state()

    def _init_params(self) -> List[np.array]:
        scale = 0.01
        def param_single_layer():
            w = np.random.normal(scale=scale, size=(self.hidden_dim, self.hidden_
            dim+input_dim))
```

(continues on next page)

(continued from previous page)

```

        b = np.zeros(shape=[self.hidden_dim, 1])
        return w, b

    # reset, update gate
    Wr, br = param_single_layer()
    Wu, bu = param_single_layer()
    # output layer
    Wy = np.random.normal(scale=scale, size=[self.out_dim, self.hidden_dim])
    by = np.zeros(shape=[self.out_dim, 1])
    return [Wr, br, Wu, bu, Wy, by]

def _init_hidden_state(self) -> np.array:
    return np.zeros(shape=[self.hidden_dim, self.batch_size])

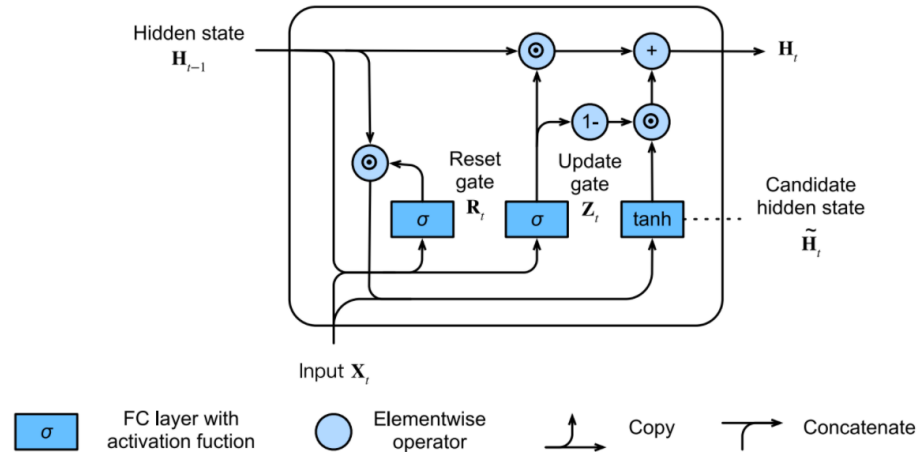
def forward(self, input_vector: np.array) -> np.array:
    """
    input_vector:
        dimension: [num_steps, self.input_dim, self.batch_size]
    out_vector:
        dimension: [num_steps, self.output_dim, self.batch_size]
    """
    Wr, br, Wu, bu, Wy, by = self.params
    output_vector = []
    for vector in input_vector:
        # expit in scipy is sigmoid function
        reset_gate = expit(
            np.dot(Wr, np.concatenate([self.hidden_state, vector], axis=0)) + br
        )
        update_gate = expit(
            np.dot(Wu, np.concatenate([self.hidden_state, vector], axis=0)) + bu
        )
        candidate_hidden = np.tanh(
            reset_gate * self.hidden_state
        )
        self.hidden_state = update_gate * self.hidden_state + (1-update_gate) * ↵
candidate_hidden
        y = softmax(
            np.dot(Wy, self.hidden_state) + by
        )
        output_vector.append(y)
    return np.array(output_vector)

```

14.8 LSTM

In order to address the **long-term information preservation** and **shor-term skipping** in latent variable model, we introduced LSTM. In LSTM, we introduce the memory cell that has the same shape as the hidden state, which is actually a fancy version of a hidden state, engineered to record additional information.

The structure and math are as follow:



1. Forget gate: $F_t = \text{sigmoid}(W_f[\frac{h^{<t-1>}}{x_t}] + b_f)$
2. Input gate: $I_t = \text{sigmoid}(W_i[\frac{h^{<t-1>}}{x_t}] + b_i)$
3. Output gate: $O_t = \text{sigmoid}(W_o[\frac{h^{<t-1>}}{x_t}] + b_o)$
4. Candidate memory: $\tilde{c} = \tanh(W_c[\frac{h^{<t-1>}}{x_t}] + b_c)$
5. Memory: $c^{<t>} = U_t * \tilde{c} + F_t * c^{<t-1>}$
6. Hidden state: $h^{<t>} = O_t * \tanh(c^{<t>})$
7. Output layer: $\hat{y}_t = \text{softmax}(W_y h^{<t>} + b_y)$

Code

For detail code, refer to [layers.py](#)

```
class LSTM:
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int, batch_
    size=1) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.out_dim = output_dim
        self.batch_size = batch_size
        # initialization
        self.params = self._init_params()
        self.hidden_state = self._init_hidden_state()
        self.memory_state = self._init_hidden_state()

    def _init_params(self) -> List[np.array]:
        scale = 0.01
        def param_single_layer():
            w = np.random.normal(scale=scale, size=(self.hidden_dim, self.hidden_
            dim+input_dim))
            b = np.zeros(shape=[self.hidden_dim, 1])
            return w, b

        # forget, input, output gate + candidate memory state
        Wf, bf = param_single_layer()
```

(continues on next page)

(continued from previous page)

```

Wi, bi = param_single_layer()
Wo, bo = param_single_layer()
Wc, bc = param_single_layer()
# output layer
Wy = np.random.normal(scale=scale, size=[self.out_dim, self.hidden_dim])
by = np.zeros(shape=[self.out_dim, 1])
return [Wf, bf, Wi, bi, Wo, bo, Wc, bc, Wy, by]

def _init_hidden_state(self) -> np.array:
    return np.zeros(shape=[self.hidden_dim, self.batch_size])

def forward(self, input_vector: np.array) -> np.array:
    """
    input_vector:
        dimension: [num_steps, self.input_dim, self.batch_size]
    out_vector:
        dimension: [num_steps, self.output_dim, self.batch_size]
    """
    Wf, bf, Wi, bi, Wo, bo, Wc, bc, Wy, by = self.params
    output_vector = []
    for vector in input_vector:
        # expit in scipy is sigmoid function
        foget_gate = expit(
            np.dot(Wf, np.concatenate([self.hidden_state, vector], axis=0)) + bf
        )
        input_gate = expit(
            np.dot(Wi, np.concatenate([self.hidden_state, vector], axis=0)) + bi
        )
        output_gate = expit(
            np.dot(Wo, np.concatenate([self.hidden_state, vector], axis=0)) + bo
        )
        candidate_memory = np.tanh(
            np.dot(Wc, np.concatenate([self.hidden_state, vector], axis=0)) + bc
        )
        self.memory_state = foget_gate * self.memory_state + input_gate * candidate_memory
        self.hidden_state = output_gate * np.tanh(self.memory_state)
        y = softmax(
            np.dot(Wy, self.hidden_state) + by
        )
        output_vector.append(y)
    return np.array(output_vector)

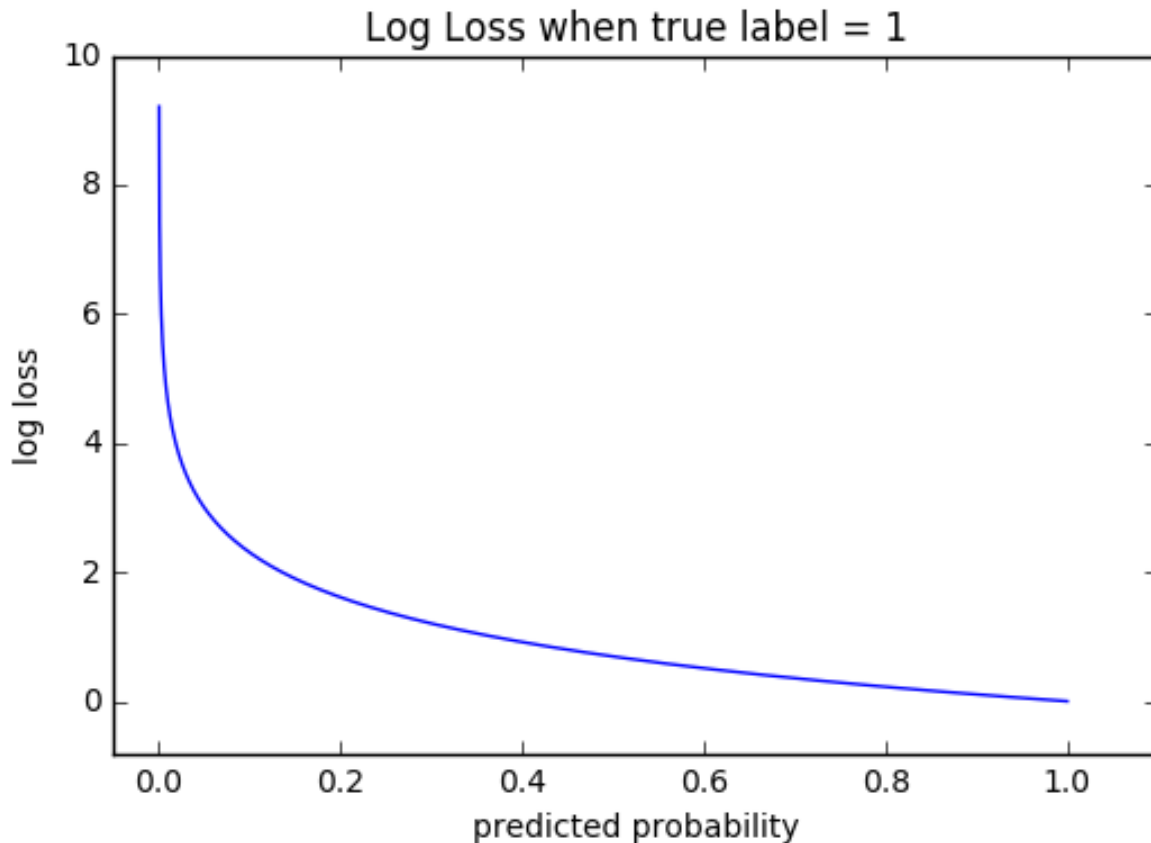
```

References

- *Cross-Entropy*
- *Hinge*
- *Huber*
- *Kullback-Leibler*
- *RMSE*
- *MAE (L1)*
- *MSE (L2)*

15.1 Cross-Entropy

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.



The graph above shows the range of possible loss values given a true observation (isDog = 1). As the predicted probability approaches 1, log loss slowly decreases. As the predicted probability decreases, however, the log loss increases rapidly. Log loss penalizes both types of errors, but especially those predictions that are confident and wrong!

Cross-entropy and log loss are slightly different depending on context, but in machine learning when calculating error rates between 0 and 1 they resolve to the same thing.

Code

Math

In binary classification, where the number of classes M equals 2, cross-entropy can be calculated as:

$$-(y \log(p) + (1 - y) \log(1 - p))$$

If $M > 2$ (i.e. multiclass classification), we calculate a separate loss for each class label per observation and sum the result.

$$-\sum_{c=1}^M y_{o,c} \log(p_{o,c})$$

Note:

- M - number of classes (dog, cat, fish)

- log - the natural log
 - y - binary indicator (0 or 1) if class label c is the correct classification for observation o
 - p - predicted probability observation o is of class c
-

15.2 Hinge

Used for classification.

Code

15.3 Huber

Typically used for regression. It's less sensitive to outliers than the MSE as it treats error as square only inside an interval.

$$L_{\delta} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |(y - \hat{y})| < \delta \\ \delta((y - \hat{y}) - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Code

Further information can be found at [Huber Loss in Wikipedia](#).

15.4 Kullback-Leibler

Code

15.5 RMSE

Root Mean Square Error

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})^2}$$

RMSE - root mean square error

m - number of samples

$x^{(i)}$ - i-th sample from dataset

$h(x^{(i)})$ - prediction for i-th sample (thesis)

$y^{(i)}$ - ground truth label for i-th sample

Code**15.6 MAE (L1)**

Mean Absolute Error, or L1 loss. Excellent overview below [6] and [10].

$$MAE = \frac{1}{m} \sum_{i=1}^m |h(x^{(i)}) - y^{(i)}|$$

MAE - mean absolute error

m - number of samples

$x^{(i)}$ - i-th sample from dataset

$h(x^{(i)})$ - prediction for i-th sample (thesis)

$y^{(i)}$ - ground truth label for i-th sample

Code**15.7 MSE (L2)**

Mean Squared Error, or L2 loss. Excellent overview below [6] and [10].

$$MSE = \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$$

MSE - mean square error

m - number of samples

$y^{(i)}$ - ground truth label for i-th sample

$\hat{y}^{(i)}$ - predicted label for i-th sample

References

What is Optimizer ?

It is very important to tweak the weights of the model during the training process, to make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?

Best answer to all above question is *optimizers*. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

Below are list of example optimizers

- *Adagrad*
- *Adadelta*
- *Adam*
- *Conjugate Gradients*
- *BFGS*
- *Momentum*
- *Nesterov Momentum*
- *Newton's Method*
- *RMSProp*
- *SGD*

Image Credit: CS231n

16.1 Adagrad

Adagrad (short for adaptive gradient) adaptively sets the learning rate according to a parameter.

- Parameters that have higher gradients or frequent updates should have slower learning rate so that we do not overshoot the minimum value.
- Parameters that have low gradients or infrequent updates should faster learning rate so that they get trained quickly.
- It divides the learning rate by the sum of squares of all previous gradients of the parameter.
- When the sum of the squared past gradients has a high value, it basically divides the learning rate by a high value, so the learning rate will become less.
- Similarly, if the sum of the squared past gradients has a low value, it divides the learning rate by a lower value, so the learning rate value will become high.
- This implies that the learning rate is inversely proportional to the sum of the squares of all the previous gradients of the parameter.

$$g_t^i = \frac{\partial \mathcal{J}(w_t^i)}{\partial W}$$
$$W = W - \alpha \frac{\partial \mathcal{J}(w_t^i)}{\sqrt{\sum_{r=1}^t (g_r^i)^2 + \epsilon}}$$

Note:

- g_t^i - the gradient of a parameter, θ at an iteration t.
 - α - the learning rate
 - ϵ - very small value to avoid dividing by zero
-

16.2 Adadelata

AdaDelta belongs to the family of stochastic gradient descent algorithms, that provide adaptive techniques for hyper-parameter tuning. Adadelata is probably short for 'adaptive delta', where delta here refers to the difference between the current weight and the newly updated weight.

The main disadvantage in Adagrad is its accumulation of the squared gradients. During the training process, the accumulated sum keeps growing. From the above formula we can see that, As the accumulated sum increases learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

Adadelata is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelata continues learning even when many updates have been done.

With Adadelata, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

Implementation is something like this,

$$\begin{aligned}
 v_t &= \rho v_{t-1} + (1 - \rho) \nabla_{\theta}^2 J(\theta) \\
 \Delta \theta &= \frac{\sqrt{w_t + \epsilon}}{\sqrt{v_t + \epsilon}} \nabla_{\theta} J(\theta) \\
 \theta &= \theta - \eta \Delta \theta \\
 w_t &= \rho w_{t-1} + (1 - \rho) \Delta \theta^2
 \end{aligned}$$

16.3 Adam

Adaptive Moment Estimation (Adam) combines ideas from both RMSProp and Momentum. It computes adaptive learning rates for each parameter and works as follows.

- First, it computes the exponentially weighted average of past gradients (v_{dW}).
- Second, it computes the exponentially weighted average of the squares of past gradients (s_{dW}).
- Third, these averages have a bias towards zero and to counteract this a bias correction is applied ($v_{dW}^{corrected}$, $s_{dW}^{corrected}$).
- Lastly, the parameters are updated using the information from the calculated averages.

$$\begin{aligned}
 v_{dW} &= \beta_1 v_{dW} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W} \\
 s_{dW} &= \beta_2 s_{dW} + (1 - \beta_2) \left(\frac{\partial \mathcal{J}}{\partial W} \right)^2 \\
 v_{dW}^{corrected} &= \frac{v_{dW}}{1 - (\beta_1)^t} \\
 s_{dW}^{corrected} &= \frac{s_{dW}}{1 - (\beta_2)^t} \\
 W &= W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected} + \epsilon}}
 \end{aligned}$$

Note:

- v_{dW} - the exponentially weighted average of past gradients
 - s_{dW} - the exponentially weighted average of past squares of gradients
 - β_1 - hyperparameter to be tuned
 - β_2 - hyperparameter to be tuned
 - $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer
 - W - the weight matrix (parameter to be updated)
 - α - the learning rate
 - ϵ - very small value to avoid dividing by zero
-

16.4 Conjugate Gradients

Be the first to [contribute!](#)

16.5 BFGS

Be the first to [contribute!](#)

16.6 Momentum

Used in conjunction Stochastic Gradient Descent (sgd) or Mini-Batch Gradient Descent, Momentum takes into account past gradients to smooth out the update. This is seen in variable v which is an exponentially weighted average of the gradient on previous steps. This results in minimizing oscillations and faster convergence.

$$v_{dW} = \beta v_{dW} + (1 - \beta) \frac{\partial \mathcal{J}}{\partial W}$$
$$W = W - \alpha v_{dW}$$

Note:

- v - the exponentially weighted average of past gradients
 - $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer weight tensor
 - W - weight tensor
 - β - hyperparameter to be tuned
 - α - the learning rate
-

16.7 Nesterov Momentum

Be the first to [contribute!](#)

16.8 Newton's Method

Be the first to [contribute!](#)

16.9 RMSProp

Another adaptive learning rate optimization algorithm, Root Mean Square Prop (RMSProp) works by keeping an exponentially weighted average of the squares of past gradients. RMSProp then divides the learning rate by this average to speed up convergence.

$$s_{dW} = \beta s_{dW} + (1 - \beta) \left(\frac{\partial \mathcal{J}}{\partial W} \right)^2$$
$$W = W - \alpha \frac{\frac{\partial \mathcal{J}}{\partial W}}{\sqrt{s_{dW}^{corrected} + \epsilon}}$$

Note:

- s - the exponentially weighted average of past squares of gradients
 - $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer weight tensor
 - W - weight tensor
 - β - hyperparameter to be tuned
 - α - the learning rate
 - ϵ - very small value to avoid dividing by zero
-

16.10 SGD

SGD stands for Stochastic Gradient Descent. In Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called “batch” which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get really huge.

This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

Since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with significantly shorter training time.

References

- *Data Augmentation*
- *Dropout*
- *Early Stopping*
- *Ensembling*
- *Injecting Noise*
- *L1 Regularization*
- *L2 Regularization*

What is overfitting?

From Wikipedia [overfitting](#) is,

The production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably

What is Regularization?

It is a Techniques for combating overfitting and improving training.

17.1 Data Augmentation

Having more data is the surest way to get better consistent estimators (ML model). Unfortunately, in the real world getting a large volume of useful data for training a model is cumbersome and labelling is an extremely tedious (or expensive) task.

‘Gold standard’ labelling requires more manual annotation. For example, in order to develop a better image classifier we can use Mturk and involve more man power to generate dataset, or we could crowdsource by posting on social media and asking people to contribute. The above process can yield good datasets; however, those are difficult to carry and expensive. On the other hand, having a small dataset will lead to the well-known problem of overfitting.

Data Augmentation is one interesting regularization technique to resolve the above problem. The concept is very simple, this technique generates new training data from given original dataset. Dataset Augmentation provides a cheap and easy way to increase the volume of training data.

This technique can be used for both NLP and CV.

In CV we can use the techniques like Jitter, PCA and Flipping. Similarly in NLP we can use the techniques like Synonym Replacement, Random Insertion, Random Deletion and Word Embeddings.

Many software libraries contain tools for data augmentation. For example, Keras provides the ImageDataGenerator for augmenting image datasets.

Sample code for random deletion

```
def random_deletion(words, p):
    """
    Randomly delete words from the sentence with probability p
    """

    #obviously, if there's only one word, don't delete it
    if len(words) == 1:
        return words

    #randomly delete words with probability p
    new_words = []
    for word in words:
        r = random.uniform(0, 1)
        if r > p:
            new_words.append(word)

    #if you end up deleting all words, just return a random word
    if len(new_words) == 0:
        rand_int = random.randint(0, len(words)-1)
        return [words[rand_int]]

    return new_words
```

Furthermore, when comparing two machine learning algorithms, it is important to train both with either augmented or non-augmented dataset. Otherwise, no subjective decision can be made on which algorithm performed better

Further reading

- NLP Data Augmentation
- CV Data Augmentation
- Regularization

17.2 Dropout

What is Dropout?

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data.

Dropout is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

Simply put, It is the process of ignoring some of the neurons in particular forward or backward pass.

Dropout can be easily implemented by randomly selecting nodes to be dropped-out with a given probability (e.g. .1%) each weight update cycle.

Most importantly Dropout is only used during the training of a model and is not used when evaluating the model.

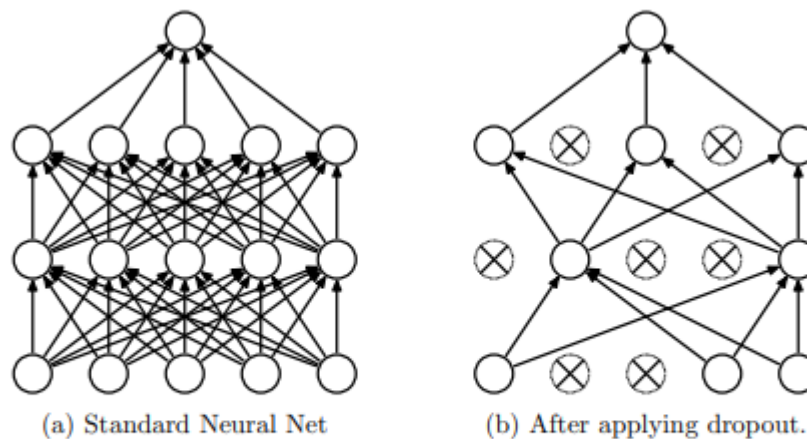


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

image from <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>

```
import numpy as np
A = np.arange(20).reshape((5,4))

print("Given input: ")
print(A)

def dropout(X, drop_probability):
    keep_probability = 1 - drop_probability
    mask = np.random.uniform(0, 1.0, X.shape) < keep_probability
    if keep_probability > 0.0:
        scale = (1/keep_probability)
    else:
        scale = 0.0
    return mask * X * scale

print("\n After Dropout: ")
print(dropout(A,0.5))
```

output from above code

```
Given input:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]

After Dropout:
[[ 0.  2.  0.  0.]
 [ 8.  0.  0. 14.]
 [16. 18.  0. 22.]
 [24.  0.  0.  0.]
 [32. 34. 36.  0.]]
```

Further reading

- Dropout <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>

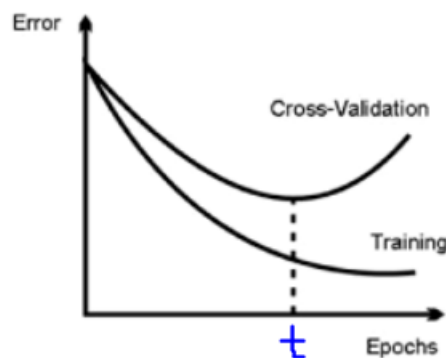
17.3 Early Stopping

One of the biggest problem in training neural network is how long to train the model.

Training too little will lead to underfit in train and test sets. Training too much will have the overfit in training set and poor result in test sets.

Here the challenge is to train the network long enough that it is capable of learning the mapping from inputs to outputs, but not training the model so long that it overfits the training data.

One possible solution to solve this problem is to treat the number of training epochs as a hyperparameter and train the model multiple times with different values, then select the number of epochs that result in the best accuracy on the train or a holdout test dataset, But the problem is it requires multiple models to be trained and discarded.



Clearly, after 't' epochs, the model starts overfitting. This is clear by the increasing gap between the train and the validation error in the above plot.

One alternative technique to prevent overfitting is use validation error to decide when to stop. This approach is called Early Stopping.

While building the model, it is evaluated on the holdout validation dataset after each epoch. If the accuracy of the model on the validation dataset starts to degrade (e.g. loss begins to increase or accuracy begins to decrease), then the training process is stopped. This process is called Early stopping.

Python implementation for Early stopping,

```
def early_stopping(theta0, (x_train, y_train), (x_valid, y_valid), n = 1, p = 100):
    """ The early stopping meta-algorithm for determining the best amount of time to
    ↪train.
        REF: Algorithm 7.1 in deep learning book.

        Parameters:
        n: int; Number of steps between evaluations.
        p: int; "patience", the number of evaluations to observe worsening validation
    ↪set.
        theta0: Network; initial network.
        x_train: iterable; The training input set.
        y_train: iterable; The training output set.
        x_valid: iterable; The validation input set.
        y_valid: iterable; The validation output set.

        Returns:
        theta_prime: Network object; The output network.
        i_prime: int; The number of iterations for the output network.
        v: float; The validation error for the output network.
    """
    # Initialize variables
    theta = theta0.clone()          # The active network
    i = 0                          # The number of training steps taken
    j = 0                          # The number of evaluations steps since last update of
    ↪theta_prime
    v = np.inf                     # The best evaluation error observed thusfar
    theta_prime = theta.clone()     # The best network found thusfar
    i_prime = i                    # The index of theta_prime

    while j < p:
        # Update theta by running the training algorithm for n steps
        for _ in range(n):
            theta.train(x_train, y_train)

        # Update Values
        i += n
        v_new = theta.error(x_valid, y_valid)

        # If better validation error, then reset waiting time, save the network, and
    ↪update the best error value
        if v_new < v:
            j = 0
            theta_prime = theta.clone()
            i_prime = i
            v = v_new

        # Otherwise, update the waiting time
        else:
            j += 1

    return theta_prime, i_prime, v
```

Further reading

- Regularization

17.4 Ensembling

Ensemble methods combine several machine learning techniques into one predictive model. There are a few different methods for ensembling, but the two most common are:

Bagging

- Bagging stands for bootstrap aggregation. One way to reduce the variance of an estimate is to average together multiple estimates.
- It trains a large number of “strong” learners in parallel.
- A strong learner is a model that’s relatively unconstrained.
- Bagging then combines all the strong learners together in order to “smooth out” their predictions.

Boosting

- Boosting refers to a family of algorithms that are able to convert weak learners to strong learners.
- Each one in the sequence focuses on learning from the mistakes of the one before it.
- Boosting then combines all the weak learners into a single strong learner.

Bagging uses complex base models and tries to “smooth out” their predictions, while boosting uses simple base models and tries to “boost” their aggregate complexity.

17.5 Injecting Noise

Noise is often introduced to the inputs as a dataset augmentation strategy. When we have a small dataset the network may effectively memorize the training dataset. Instead of learning a general mapping from inputs to outputs, the model may learn the specific input examples and their associated outputs. One approach for improving generalization error and improving the structure of the mapping problem is to add random noise.

Adding noise means that the network is less able to memorize training samples because they are changing all of the time, resulting in smaller network weights and a more robust network that has lower generalization error.

Noise is only added during training. No noise is added during the evaluation of the model or when the model is used to make predictions on new data.

Random noise can be added to other parts of the network during training. Some examples include:

Noise Injection on Weights

- Noise added to weights can be interpreted as a more traditional form of regularization.
- In other words, it pushes the model to be relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions.

Noise Injection on Outputs

- In the real world dataset, We can expect some amount of mistakes in the output labels. One way to remedy this is to explicitly model the noise on labels.

- An example for Noise Injection on Outputs is **label smoothing**

Further reading

- [Regularization](#)

17.6 L1 Regularization

A regression model that uses L1 regularization technique is called *Lasso Regression*.

Mathematical formula for L1 Regularization.

Let's define a model to see how L1 Regularization works. For simplicity, We define a simple linear regression model Y with one independent variable.

In this model, W represent Weight, b represent Bias.

$$W = w_1, w_2 \dots w_n$$

$$X = x_1, x_2 \dots x_n$$

and the predicted result is \hat{Y}

$$\hat{Y} = w_1 x_1 + w_2 x_2 + \dots w_n x_n + b$$

Following formula calculates the error without Regularization function

$$Loss = Error(Y, \hat{Y})$$

Following formula calculates the error With L1 Regularization function

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{i=1}^n |w_i|$$

Note: Here, If the value of lambda is Zero then above Loss function becomes Ordinary Least Square whereas very large value makes the coefficients (weights) zero hence it under-fits.

One thing to note is that $|w|$ is differentiable when $w \neq 0$ as shown below,

$$\frac{d|w|}{dw} = \begin{cases} 1 & w > 0 \\ -1 & w < 0 \end{cases}$$

To understand the Note above,

Let's substitute the formula in finding new weights using Gradient Descent optimizer.

$$w_{new} = w - \eta \frac{\partial L1}{\partial w}$$

When we apply the L1 in above formula it becomes,

$$\begin{aligned} w_{new} &= w - \eta \cdot (Error(Y, \hat{Y}) + \lambda \frac{d|w|}{dw}) \\ &= \begin{cases} w - \eta \cdot (Error(Y, \hat{Y}) + \lambda) & w > 0 \\ w - \eta \cdot (Error(Y, \hat{Y}) - \lambda) & w < 0 \end{cases} \end{aligned}$$

From the above formula,

- If w is positive, the regularization parameter $\lambda > 0$ will push w to be less positive, by subtracting λ from w .
- If w is negative, the regularization parameter $\lambda < 0$ will push w to be less negative, by adding λ to w . hence this has the effect of pushing w towards 0.

Simple python implementation

```
def update_weights_with_l1_regularization(features, targets, weights, lr, lambda):  
    '''  
    Features: (200, 3)  
    Targets: (200, 1)  
    Weights: (3, 1)  
    '''  
  
    predictions = predict(features, weights)  
  
    #Extract our features  
    x1 = features[:,0]  
    x2 = features[:,1]  
    x3 = features[:,2]  
  
    # Use matrix cross product (*) to simultaneously  
    # calculate the derivative for each weight  
    d_w1 = -x1*(targets - predictions)  
    d_w2 = -x2*(targets - predictions)  
    d_w3 = -x3*(targets - predictions)  
  
    # Multiply the mean derivative by the learning rate  
    # and subtract from our weights (remember gradient points in direction of ↪  
    ↪steepest ASCENT)  
  
    weights[0][0] = (weights[0][0] - lr * np.mean(d_w1) - lambda) if weights[0][0] > ↪  
    ↪0 else (weights[0][0] - lr * np.mean(d_w1) + lambda)  
    weights[1][0] = (weights[1][0] - lr * np.mean(d_w2) - lambda) if weights[1][0] > ↪  
    ↪0 else (weights[1][0] - lr * np.mean(d_w2) + lambda)  
    weights[2][0] = (weights[2][0] - lr * np.mean(d_w3) - lambda) if weights[2][0] > ↪  
    ↪0 else (weights[2][0] - lr * np.mean(d_w3) + lambda)  
  
    return weights
```

Use Case

L1 Regularization (or variant of this concept) is a model of choice when the number of features are high, Since it provides sparse solutions. We can get computational advantage as the features with zero coefficients can simply be ignored.

Further reading

- [Linear Regression](#)

17.7 L2 Regularization

A regression model that uses L2 regularization technique is called *Ridge Regression*. Main difference between L1 and L2 regularization is, L2 regularization uses “squared magnitude” of coefficient as penalty term to the loss function.

Mathematical formula for L2 Regularization.

Let's define a model to see how L2 Regularization works. For simplicity, We define a simple linear regression model Y with one independent variable.

In this model, W represent Weight, b represent Bias.

$$W = w_1, w_2 \dots w_n$$

$$X = x_1, x_2 \dots x_n$$

and the predicted result is \hat{Y}

$$\hat{Y} = w_1 x_1 + w_2 x_2 + \dots w_n x_n + b$$

Following formula calculates the error without Regularization function

$$Loss = Error(Y, \hat{Y})$$

Following formula calculates the error With L2 Regularization function

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_1^n w_i^2$$

Note: Here, if lambda is zero then you can imagine we get back OLS. However, if lambda is very large then it will add too much weight and it leads to under-fitting.

To understand the Note above,

Let's substitute the formula in finding new weights using Gradient Descent optimizer.

$$w_{new} = w - \eta \frac{\partial L2}{\partial w}$$

When we apply the L2 in above formula it becomes,

$$w_{new} = w - \eta \cdot (Error(Y, \hat{Y}) + \lambda \frac{\partial L2}{\partial w})$$

$$= w - \eta \cdot (Error(Y, \hat{Y}) + 2\lambda w)$$

Simple python implementation

```
def update_weights_with_l2_regularization(features, targets, weights, lr, lambda):
    """
    Features: (200, 3)
    Targets: (200, 1)
    Weights: (3, 1)
    """
    predictions = predict(features, weights)

    # Extract our features
    x1 = features[:,0]
    x2 = features[:,1]
    x3 = features[:,2]

    # Use matrix cross product (*) to simultaneously
    # calculate the derivative for each weight
```

(continues on next page)

(continued from previous page)

```
d_w1 = -x1*(targets - predictions)
d_w2 = -x2*(targets - predictions)
d_w3 = -x3*(targets - predictions)

# Multiply the mean derivative by the learning rate
# and subtract from our weights (remember gradient points in direction of ↘
↪steepest ASCENT)

weights[0][0] = weights[0][0] - lr * np.mean(d_w1) - 2 * lambda * weights[0][0]
weights[1][0] = weights[1][0] - lr * np.mean(d_w2) - 2 * lambda * weights[1][0]
weights[2][0] = weights[2][0] - lr * np.mean(d_w3) - 2 * lambda * weights[2][0]

return weights
```

Use Case

L2 regularization can address the multicollinearity problem by constraining the coefficient norm and keeping all the variables. L2 regression can be used to estimate the predictor importance and penalize predictors that are not important. One issue with co-linearity is that the variance of the parameter estimate is huge. In cases where the number of features are greater than the number of observations, the matrix used in the OLS may not be invertible but Ridge Regression enables this matrix to be inverted.

Further reading

- [Ridge Regression](#)

References

- *Autoencoder*
- *CNN*
- *GAN*
- *MLP*
- *RNN*
- *VAE*

18.1 Autoencoder

An autoencoder is a type of feedforward neural network that attempts to copy its input to its output. Internally, it has a hidden layer, **h**, that describes a **code**, used to represent the input. The network consists of two parts:

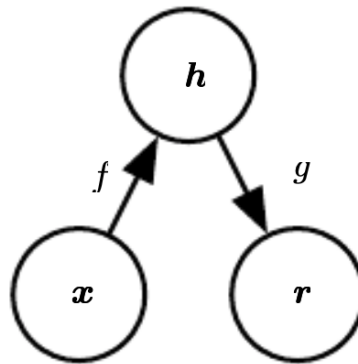
- An *encoder* function: $h = f(x)$.
- A *decoder* function, that produces a reconstruction: $r = g(h)$.

The figure below shows the presented architecture.

The autoencoder compresses the input into a lower-dimensional code, and then it reconstructs the output from this representation. The code is a compact “summary”, or “compression”, of the input, and it is also called the *latent-space representation*.

If an autoencoder simply learned to set $g(f(x)) = x$ everywhere, then it would not be very useful; instead, autoencoders are designed to be unable to learn to copy perfectly. They are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input to copy, it learns useful properties of the data.

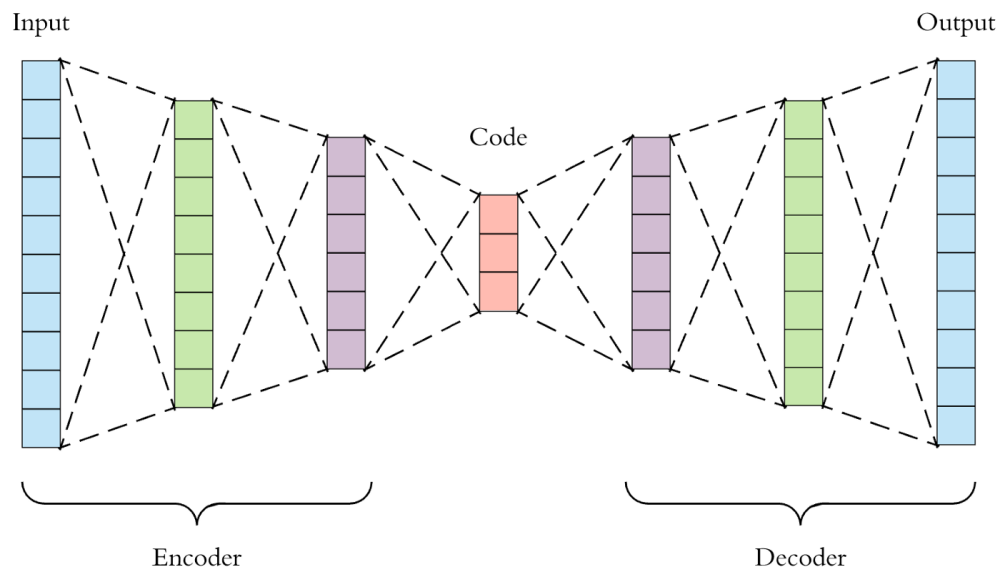
⁶ *Deep Learning Book - Autoencoders <<https://www.deeplearningbook>

Fig. 1: Source⁶

In order to build an autoencoder, three things are needed: an encoding method, a decoding method, and a loss function to compare the output with the target.

Both the encoder and the decoder are fully-connected feedforward neural networks. The code is a single layer of an artificial neural network, with the dimensionality of our choice. The number of nodes in the code layer (the *code size*) is a *hyperparameter* to be set before training the autoencoder.

The figure below shows the autoencoder architecture. First, the input passes through the encoder, which is a fully-connected neural network, in order to produce the code. The decoder, which has the similar neural network structure, then produces the output by using the code only. The aim is to get an output identical to the input.

Fig. 2: Source⁵

Traditionally, autoencoders were used for dimensionality reduction or feature learning. More recently, theoretical connections between autoencoders and latent variable models have brought autoencoders to the forefront of generative modeling. As a compression method, autoencoders do not perform better than their alternatives. And the fact that autoencoders are data-specific makes them impractical as a general technique.

⁵ • Applied Deep Learning - Part 3: Autoencoders

In general, autoencoders have three common use cases:

- **Data denoising:** It should be noted that denoising autoencoders are not meant to automatically denoise an image, instead they were invented to help the hidden layers of the autoencoder learn more robust filters, and reduce the risk of overfitting.
- **Dimensionality reduction:** Visualizing high-dimensional data is challenging. t-SNE⁷ is the most commonly used method, but struggles with large number of dimensions (typically above 32). Therefore, autoencoders can be used as a preprocessing step to reduce the dimensionality, and this compressed representation is used by t-SNE to visualize the data in 2D space.
- **Variational Autoencoders (VAE):** this is a more modern and complex use-case of autoencoders. VAE learns the parameters of the probability distribution modeling the input data, instead of learning an arbitrary function in the case of vanilla autoencoders. By sampling points from this distribution we can also use the VAE as a generative model⁸.

Model

An example implementation in PyTorch.

```
class Autoencoder(nn.Module):
    def __init__(self, in_shape):
        super().__init__()
        c,h,w = in_shape
        self.encoder = nn.Sequential(
            nn.Linear(c*h*w, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 12),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(12, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, c*h*w),
            nn.Sigmoid()
        )

    def forward(self, x):
        bs,c,h,w = x.size()
        x = x.view(bs, -1)
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.view(bs, c, h, w)
        return x
```

Training

```
def train(net, loader, loss_func, optimizer):
    net.train()
```

(continues on next page)

⁷ t-SNE

⁸ VAE

(continued from previous page)

```

for inputs, _ in loader:
    inputs = Variable(inputs)

    output = net(inputs)
    loss = loss_func(output, inputs)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

Further reading

- [Convolutional Autoencoders](#)
- [Deep Learning Book](#)

18.2 CNN

The *convolutional neural network*, or *CNN*, is a feed-forward neural network which has at least one convolutional layer. This type of deep neural network is used for processing structured arrays of data. It is distinguished from other neural networks by its superior performance with speech, audio, and especially, image data. For the latter data type, CNNs are commonly employed in computer vision tasks, like image classification, since they are especially good at finding out patterns from the input images, such as lines, circles, or more complex objects, e.g., human faces.

Convolutional neural networks comprise many convolutional layers, stacked one on top of the other, in a sequence. The sequential architecture of CNNs allows them to learn hierarchical features. Every layer can recognize shapes, and the deeper the network goes, the more complex are the shapes which can be recognized. The design of convolutional layers in a CNN reflects the structure of the human visual cortex. In fact, our visual cortex is similarly made of different layers, which process an image in our sight by sequentially identifying more and more complex features.

The CNN architecture is made up of three main distinct layers:

1. Convolutional layer
2. Pooling layer
3. Fully-connected (FC) layer

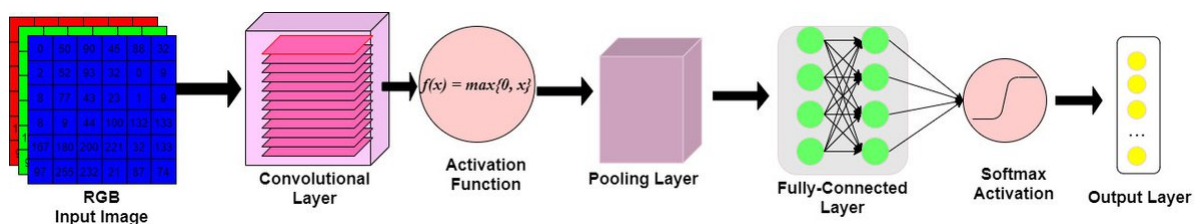


Fig. 3: **Overview of CNN architecture.** The architecture of CNNs follows this structure, but with a greater number of layers for each layer's type. The convolutional and pooling layers are layers peculiar to CNNs, while the fully-connected layer, activation function and output layer, are also present in regular feed-forward neural networks. Source: [2]

When working with image data, the CNN architecture accepts as input a 3D volume, or a 1D vector depending if the image data is in RGB format, for the first case, or in grayscale format, for the latter. Then it transforms the input through different equations, and it outputs a class. The convolutional layer is the first layer of the convolutional neural

network. While this first layer can be followed by more convolutional layers, or pooling layers, the fully-connected layer remains the last layer of the network, which outputs the result. At every subsequent convolutional layer, the CNN increases its complexity, and it can identify greater portions in the image. In the first layers, the algorithm can recognize simpler features such as color or edges. Deeper in the network, it becomes able to identify both larger objects in the image and more complex ones. In the last layers, before the image reaches the final FC layer, the CNN identifies the full object in the image.

Model

An example implementation of a CNN in PyTorch.

Training

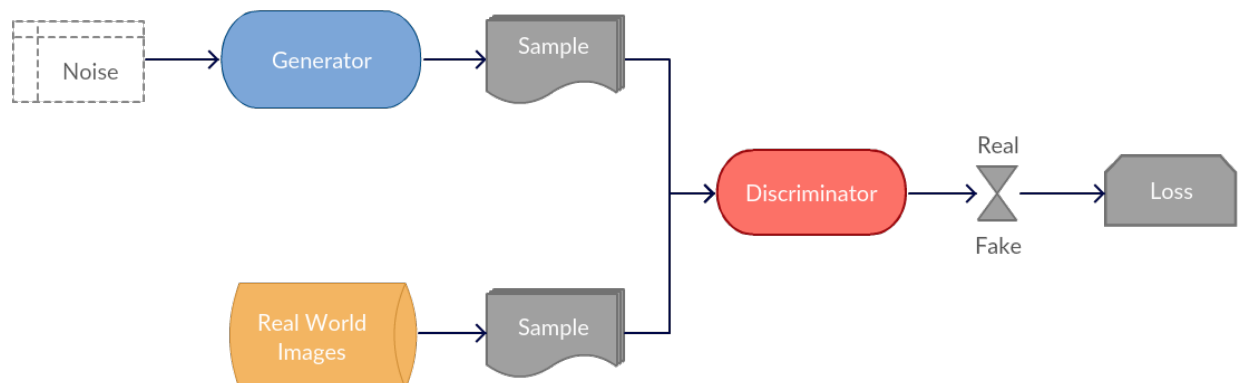
Further reading

- [CS231 Convolutional Networks](#)
- [Deep Learning Book](#)

18.3 GAN

A Generative Adversarial Network (GAN) is a type of network which creates novel tensors (often images, voices, etc.). The generative portion of the architecture competes with the discriminator part of the architecture in a zero-sum game. The goal of the generative network is to create novel tensors which the adversarial network attempts to classify as real or fake. The goal of the generative network is generate tensors where the discriminator network determines that the tensor has a 50% chance of being fake and a 50% chance of being real.

Figure from [3].



Model

An example implementation in PyTorch.

Generator

```

class Generator(nn.Module):
    def __init__(self):
        super()
        self.net = nn.Sequential(
            nn.ConvTranspose2d( 200, 32 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(32 * 8),
            nn.ReLU(),
            nn.ConvTranspose2d(32 * 8, 32 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32 * 4),
            nn.ReLU(),
            nn.ConvTranspose2d( 32 * 4, 32 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32 * 2),
            nn.ReLU(),
            nn.ConvTranspose2d( 32 * 2, 32, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.ConvTranspose2d( 32, 1, 4, 2, 1, bias=False),
            nn.Tanh()
        )
    def forward(self, tens):
        return self.net(tens)

```

Discriminator

```

class Discriminator(nn.Module):
    def __init__(self):
        super()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),
            nn.Conv2d(32, 32 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32 * 2),
            nn.LeakyReLU(0.2),
            nn.Conv2d(32 * 2, 32 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32 * 4),
            nn.LeakyReLU(0.2),
            # state size. (32*4) x 8 x 8
            nn.Conv2d(32 * 4, 32 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32 * 8),
            nn.LeakyReLU(0.2),
            # state size. (32*8) x 4 x 4
            nn.Conv2d(32 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
    def forward(self, tens):
        return self.net(tens)

```

Training

```

def train(netD, netG, loader, loss_func, optimizerD, optimizerG, num_epochs):
    netD.train()
    netG.train()
    device = "cuda:0" if torch.cuda.is_available() else "cpu"

```

(continues on next page)

(continued from previous page)

```

for epoch in range(num_epochs):
    for i, data in enumerate(loader, 0):
        netD.zero_grad()
        realtens = data[0].to(device)
        b_size = realtens.size(0)
        label = torch.full((b_size,), 1, dtype=torch.float, device=device) # gen_
↪ labels
        output = netD(realtens)
        errD_real = loss_func(output, label)
        errD_real.backward() # backprop discriminator fake and real based on label
        noise = torch.randn(b_size, 200, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(0)
        output = netD(fake.detach()).view(-1)
        errD_fake = loss_func(output, label)
        errD_fake.backward() # backprop discriminator fake and real based on label
        errD = errD_real + errD_fake # discriminator error
        optimizerD.step()
        netG.zero_grad()
        label.fill_(1)
        output = netD(fake).view(-1)
        errG = loss_func(output, label) # generator error
        errG.backward()
        optimizerG.step()

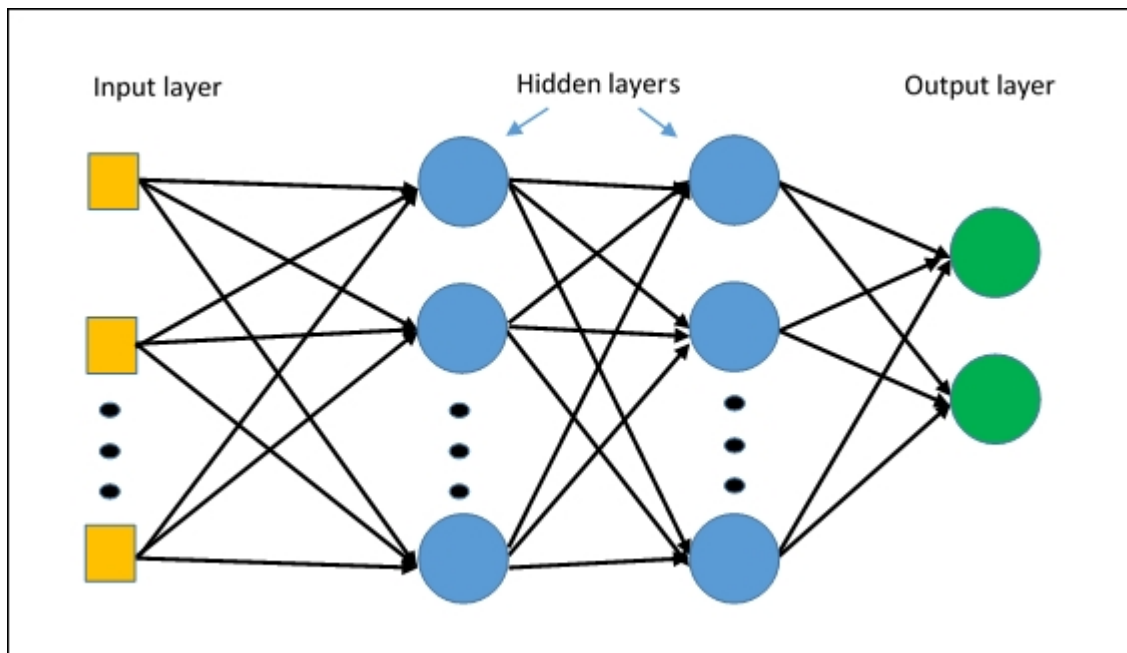
```

Further reading

- [Generative Adversarial Networks](#)
- [Deep Learning Book](#)
- [PyTorch DCGAN Example](#)
- [Original Paper](#)

18.4 MLP

A Multi Layer Perceptron (MLP) is a neural network with only fully connected layers. Figure from [5].



Model

An example implementation on FMNIST dataset in PyTorch. [Full Code](#)

1. The input to the network is a vector of size 28×28 i.e. (image from FashionMNIST dataset of dimension 28×28 pixels flattened to single dimension vector).
2. 2 fully connected hidden layers.
3. Output layer with 10 outputs. (10 classes)

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # define layers
        self.fc1 = nn.Linear(in_features=28*28, out_features=500)
        self.fc2 = nn.Linear(in_features=500, out_features=200)
        self.fc3 = nn.Linear(in_features=200, out_features=100)
        self.out = nn.Linear(in_features=100, out_features=10)

    def forward(self, t):
        # fc1 make input 1 dimensional
        t = t.view(-1, 28*28)
        t = self.fc1(t)
        t = F.relu(t)
        # fc2
        t = self.fc2(t)
        t = F.relu(t)
        # fc3
        t = self.fc3(t)
        t = F.relu(t)
        # output
        t = self.out(t)
        return t
```

Training

```
def train(net, loader, loss_func, optimizer):
    net.train()
    n_batches = len(loader)
    for inputs, targets in loader:
        inputs = Variable(inputs)
        targets = Variable(targets)

        output = net(inputs)
        loss = loss_func(output, targets)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # print statistics
    running_loss = loss.item()
    print('Training loss: %.3f' %( running_loss))
```

Evaluating

```
def main():
    train_set = torchvision.datasets.FashionMNIST(
        root = './FMNIST',
        train = True,
        download = False,
        transform = transforms.Compose([
            transforms.ToTensor()
        ])
    )
    mlp = MLP()
    loader = torch.utils.data.DataLoader(train_set, batch_size = 1000)
    optimizer = optim.Adam(mlp.parameters(), lr=0.01)
    loss_func=nn.CrossEntropyLoss()
    for i in range(0,15):
        train(mlp,loader,loss_func,optimizer)
    print("Finished Training")
    torch.save(mlp.state_dict(), "./mlpmodel.pt")
    test_set = torchvision.datasets.FashionMNIST(
        root = './FMNIST',
        train = False,
        download = False,
        transform = transforms.Compose([
            transforms.ToTensor()
        ])
    )
    testloader = torch.utils.data.DataLoader(test_set, batch_size=4)
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = mlp(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

(continues on next page)

(continued from previous page)

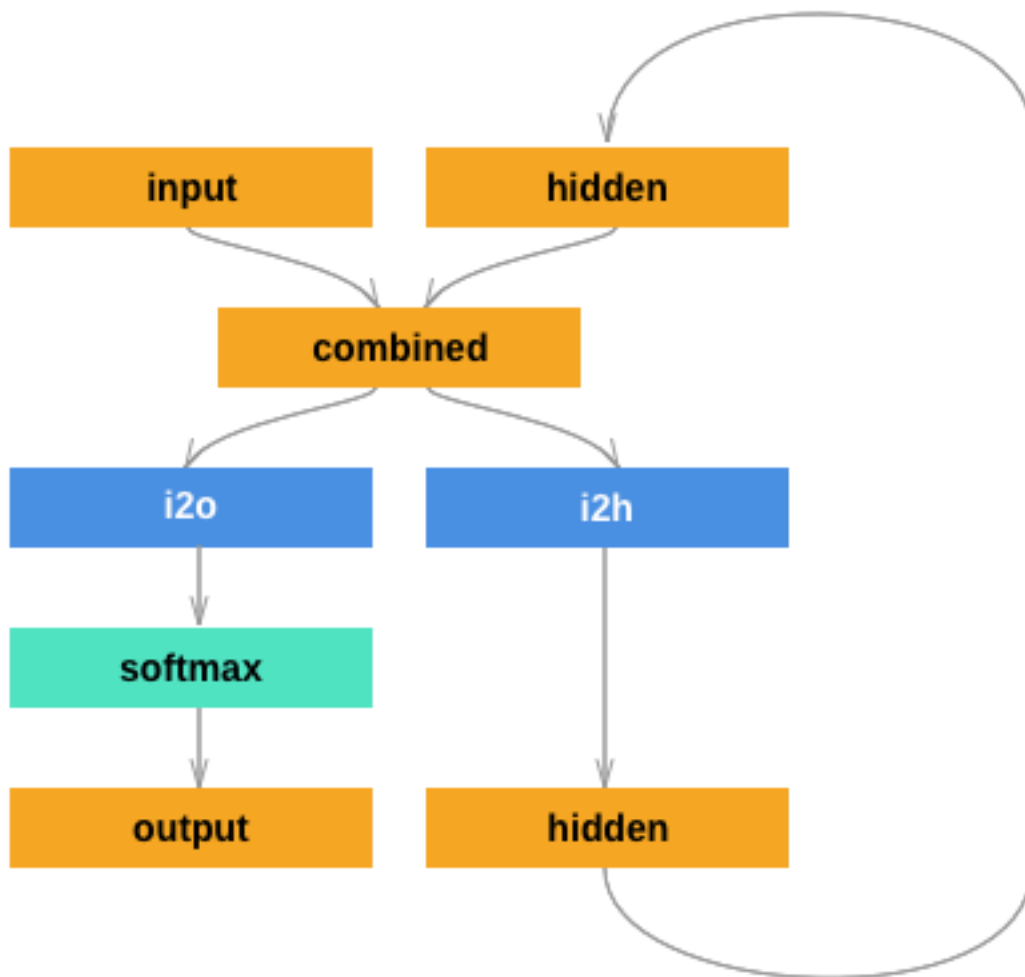
```
print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

Further reading

TODO

18.5 RNN

Description of RNN use case and basic architecture.

**Model**

```
class RNN(nn.Module):
    def __init__(self, n_classes):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```

self.hid_fc = nn.Linear(185, 128)
self.out_fc = nn.Linear(185, n_classes)
self.softmax = nn.LogSoftmax()

def forward(self, inputs, hidden):
    inputs = inputs.view(1, -1)
    combined = torch.cat([inputs, hidden], dim=1)
    hid_out = self.hid_fc(combined)
    out = self.out_fc(combined)
    out = self.softmax(out)
    return out, hid_out

```

Training

In this example, our input is a list of last names, where each name is a variable length array of one-hot encoded characters. Our target is a list of indices representing the class (language) of the name.

1. For each input name..
2. Initialize the hidden vector
3. Loop through the characters and predict the class
4. Pass the final character's prediction to the loss function
5. Backprop and update the weights

```

def train(model, inputs, targets):
    for i in range(len(inputs)):
        target = Variable(targets[i])
        name = inputs[i]
        hidden = Variable(torch.zeros(1, 128))
        model.zero_grad()

        for char in name:
            input_ = Variable(torch.FloatTensor(char))
            pred, hidden = model(input_, hidden)

        loss = criterion(pred, target)
        loss.backward()

        for p in model.parameters():
            p.data.add_(-.001, p.grad.data)

```

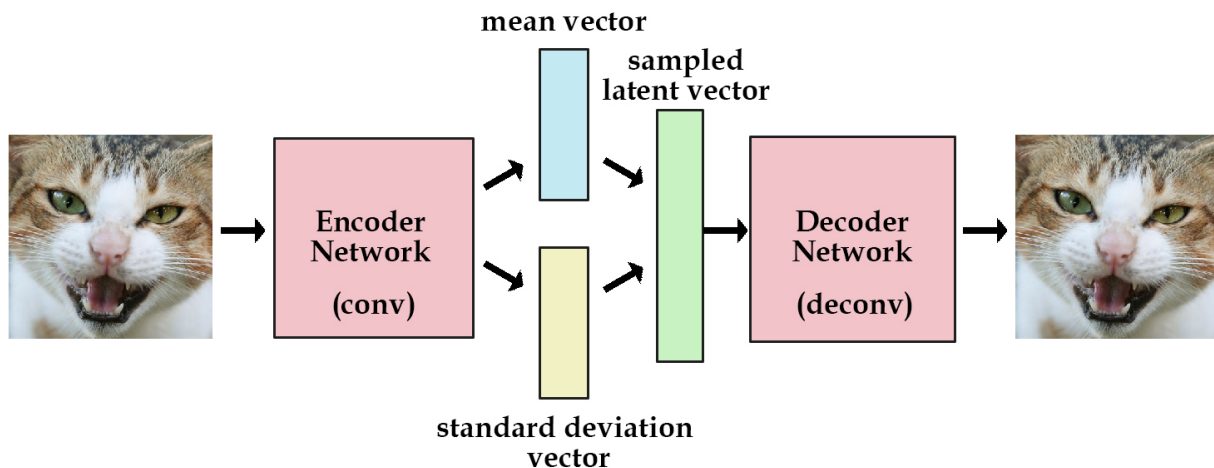
Further reading

- [Jupyter notebook](#)
- [Deep Learning Book](#)

18.6 VAE

Autoencoders can encode an input image to a latent vector and decode it, but they can't generate novel images. Variational Autoencoders (VAE) solve this problem by adding a constraint: the latent vector representation should

model a unit gaussian distribution. The Encoder returns the mean and variance of the learned gaussian. To generate a new image, we pass a new mean and variance to the Decoder. In other words, we “sample a latent vector” from the gaussian and pass it to the Decoder. It also improves network generalization and avoids memorization. Figure from [4].



Loss Function

The VAE loss function combines reconstruction loss (e.g. Cross Entropy, MSE) with KL divergence.

```
def vae_loss(output, input, mean, logvar, loss_func):
    recon_loss = loss_func(output, input)
    kl_loss = torch.mean(0.5 * torch.sum(
        torch.exp(logvar) + mean**2 - 1. - logvar, 1))
    return recon_loss + kl_loss
```

Model

An example implementation in PyTorch of a Convolutional Variational Autoencoder.

```
class VAE(nn.Module):
    def __init__(self, in_shape, n_latent):
        super().__init__()
        self.in_shape = in_shape
        self.n_latent = n_latent
        c, h, w = in_shape
        self.z_dim = h//2**2 # receptive field downsampled 2 times
        self.encoder = nn.Sequential(
            nn.BatchNorm2d(c),
            nn.Conv2d(c, 32, kernel_size=4, stride=2, padding=1), # 32, 16, 16
            nn.BatchNorm2d(32),
            nn.LeakyReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1), # 32, 8, 8
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
```

(continues on next page)

(continued from previous page)

```

    )
    self.z_mean = nn.Linear(64 * self.z_dim**2, n_latent)
    self.z_var = nn.Linear(64 * self.z_dim**2, n_latent)
    self.z_develop = nn.Linear(n_latent, 64 * self.z_dim**2)
    self.decoder = nn.Sequential(
        nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=0),
        nn.BatchNorm2d(32),
        nn.ReLU(),
        nn.ConvTranspose2d(32, 1, kernel_size=3, stride=2, padding=1),
        CenterCrop(h,w),
        nn.Sigmoid()
    )

    def sample_z(self, mean, logvar):
        stddev = torch.exp(0.5 * logvar)
        noise = Variable(torch.randn(stddev.size()))
        return (noise * stddev) + mean

    def encode(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)
        mean = self.z_mean(x)
        var = self.z_var(x)
        return mean, var

    def decode(self, z):
        out = self.z_develop(z)
        out = out.view(z.size(0), 64, self.z_dim, self.z_dim)
        out = self.decoder(out)
        return out

    def forward(self, x):
        mean, logvar = self.encode(x)
        z = self.sample_z(mean, logvar)
        out = self.decode(z)
        return out, mean, logvar

```

Training

```

def train(model, loader, loss_func, optimizer):
    model.train()
    for inputs, _ in loader:
        inputs = Variable(inputs)

        output, mean, logvar = model(inputs)
        loss = vae_loss(output, inputs, mean, logvar, loss_func)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

Further reading

- [Original Paper](#)

- VAE Explained
- Deep Learning Book

References

<<https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798/>>‘__
.org/contents/autoencoders.html/>‘__

Classification Algorithms

Classification problems is when our output Y is always in categories like positive vs negative in terms of sentiment analysis, dog vs cat in terms of image classification and disease vs no disease in terms of medical diagnosis.

19.1 Bayesian

Overlaps..

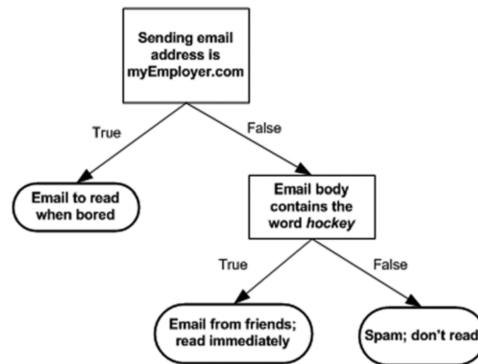
19.2 Decision Trees

Intuitions

Decision tree works by successively splitting the dataset into small segments until the target variable are the same or until the dataset can no longer be split. It's a greedy algorithm which make the best decision at the given time without concern for the global optimality².

The concept behind decision tree is straightforward. The following flowchart show a simple email classification system based on decision tree. If the address is "myEmployer.com", it will classify it to "Email to read when bored". Then if the email contains the word "hockey", this email will be classified as "Email from friends". Otherwise, it will be identified as "Spam: don't read". Image source².

² Machine Learning in Action by Peter Harrington



Algorithm Explained

There are various kinds of decision tree algorithms such as ID3 (Iterative Dichotomiser 3), C4.5 and CART (Classification and Regression Trees). The constructions of decision tree are similar⁵:

1. Assign all training instances to the root of the tree. Set current node to root node.
2. Find the split feature and split value based on the split criterion such as information gain, information gain ratio or gini coefficient.
3. Partition all data instances at the node based on the split feature and threshold value.
4. Denote each partition as a child node of the current node.
5. **For each child node:**
 1. If the child node is “pure” (has instances from only one class), tag it as a leaf and return.
 2. Else, set the child node as the current node and recurse to step 2.

ID3 creates a multiway tree. For each node, it tries to find the categorical feature that will yield the largest information gain for the target variable.

C4.5 is the successor of ID3 and remove the restriction that the feature must be categorical by dynamically define a discrete attribute that partitions the continuous attribute in the discrete set of intervals.

CART is similar to C4.5. But it differs in that it constructs binary tree and support regression problem³.

The main differences are shown in the following table:

Dimensions	ID3	C4.5	CART
Split Criterion	Information gain	Information gain ratio (Normalized information gain)	Gini coefficient for classification problems
Types of Features	Categorical feature	Categorical & numerical features	Categorical & numerical features
Type of Problem	Classification	Classification	Classification & regression
Type of Tree	Mltiway tree	Mltiway tree	Binary tree

⁵ Decision Trees

³ Scikit-learn Documentations: Tree algorithms: ID3, C4.5, C5.0 and CART

Code Implementation

We used object-oriented patterns to create the code for [ID3](#), [C4.5](#) and [CART](#). We will first introduce the base class for these three algorithms, then we explain the code of CART in details.

First, we create the base class [TreeNode](#) class and [DecisionTree](#)

```
class TreeNode:
    def __init__(self, data_idx, depth, child_lst=[]):
        self.data_idx = data_idx
        self.depth = depth
        self.child = child_lst
        self.label = None
        self.split_col = None
        self.child_cate_order = None

    def set_attribute(self, split_col, child_cate_order=None):
        self.split_col = split_col
        self.child_cate_order = child_cate_order

    def set_label(self, label):
        self.label = label


class DecisionTree()
    def fit(self, X, y):
        """
        X: train data, dimension [num_sample, num_feature]
        y: label, dimension [num_sample, ]
        """
        self.data = X
        self.labels = y
        num_sample, num_feature = X.shape
        self.feature_num = num_feature
        data_idx = list(range(num_sample))
        # Set the root of the tree
        self.root = TreeNode(data_idx=data_idx, depth=0, child_lst=[])
        queue = [self.root]
        while queue:
            node = queue.pop(0)
            # Check if the terminate criterion has been met
            if node.depth > self.max_depth or len(node.data_idx) == 1:
                # Set the label for the leaf node
                self.set_label(node)
            else:
                # Split the node
                child_nodes = self.split_node(node)
                if not child_nodes:
                    self.set_label(node)
                else:
                    queue.extend(child_nodes)
```

The CART algorithm, when constructing the binary tree, will try searching for the feature and threshold that will yield the largest gain or the least impurity. The split criterion is a combination of the child nodes' impurity. For the child nodes' impurity, gini coefficient or information gain are adopted in classification. For regression problem, mean-square-error or mean-absolute-error are used. Example codes are showed below. For more details about the formulas, please refer to [Mathematical formulation for decision tree in scikit-learn documentation](#)

```

class CART(DecisionTree):

    def get_split_criterion(self, node, child_node_lst):
        total = len(node.data_idx)
        split_criterion = 0
        for child_node in child_node_lst:
            impurity = self.get_impurity(child_node.data_idx)
            split_criterion += len(child_node.data_idx) / float(total) * impurity
        return split_criterion

    def get_impurity(self, data_ids):
        target_y = self.labels[data_ids]
        total = len(target_y)
        if self.tree_type == "regression":
            res = 0
            mean_y = np.mean(target_y)
            for y in target_y:
                res += (y - mean_y) ** 2 / total
        elif self.tree_type == "classification":
            if self.split_criterion == "gini":
                res = 1
                unique_y = np.unique(target_y)
                for y in unique_y:
                    num = len(np.where(target_y==y)[0])
                    res -= (num/float(total))**2
            elif self.split_criterion == "entropy":
                unique, count = np.unique(target_y, return_counts=True)
                res = 0
                for c in count:
                    p = float(c) / total
                    res -= p * np.log(p)

        return res

```

19.3 K-Nearest Neighbor

Introduction

K-Nearest Neighbor is a supervised learning algorithm both for classification and regression. The principle is to find the predefined number of training samples closest to the new point, and predict the label from these training samples¹.

For example, when a new point comes, the algorithm will follow these steps:

1. Calculate the Euclidean distance between the new point and all training data
2. Pick the top-K closest training data
3. For regression problem, take the average of the labels as the result; for classification problem, take the most common label of these labels as the result.

Code

Below is the Numpy implementation of K-Nearest Neighbor function. Refer to [code example](#) for details.

¹ <https://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-classification>

```

def KNN(training_data, target, k, func):
    """
    training_data: all training data point
    target: new point
    k: user-defined constant, number of closest training data
    func: functions used to get the the target label
    """
    # Step one: calculate the Euclidean distance between the new point and all_
    ↪ training data
    neighbors= []
    for index, data in enumerate(training_data):
        # distance between the target data and the current example from the data.
        distance = euclidean_distance(data[:-1], target)
        neighbors.append((distance, index))

    # Step two: pick the top-K closest training data
    sorted_neighbors = sorted(neighbors)
    k_nearest = sorted_neighbors[:k]
    k_nearest_labels = [training_data[i][1] for distance, i in k_nearest]

    # Step three: For regression problem, take the average of the labels as the_
    ↪ result;
    #               for classification problem, take the most common label of these_
    ↪ labels as the result.
    return k_nearest, func(k_nearest_labels)

```

19.4 Logistic Regression

Be the first to [contribute!](#)

19.5 Random Forests

Random Forest Classifier using ID3 Tree: [code example](#)

19.6 Boosting

Be the first to [contribute!](#)

19.7 Support Vector Machine

Support Vector Machine, or *SVM*, is one of the most popular supervised learning algorithms, and it can be used both for classification as well as regression problems. However, in machine learning, it is primarily used for classification problems. In the SVM algorithm, each data item is plotted as a point in *n-dimensional* space, where *n* is the number of features we have at hand, and the value of each feature is the value of a particular coordinate.

The goal of the SVM algorithm is to create the best line, or decision boundary, that can segregate the *n-dimensional* space into distinct classes, so that we can easily put any new data point in the correct category, in the future. This best decision boundary is called a hyperplane. The best separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class. Indeed, there are many hyperplanes that might classify the data.

As a reasonable choice for the best hyperplane is the one that represents the largest separation, or margin, between the two classes.

The SVM algorithm chooses the extreme points that help in creating the hyperplane. These extreme cases are called support vectors, while the SVM classifier is the frontier, or hyperplane, that best segregates the distinct classes.

The diagram below shows two distinct classes, denoted respectively with blue and green points. The *maximum-margin hyperplane* is the distance between the two parallel hyperplanes: *positive hyperplane* and *negative hyperplane*, shown by dashed lines. The maximum-margin hyperplane is chosen in a way that the distance between the two classes is maximised.

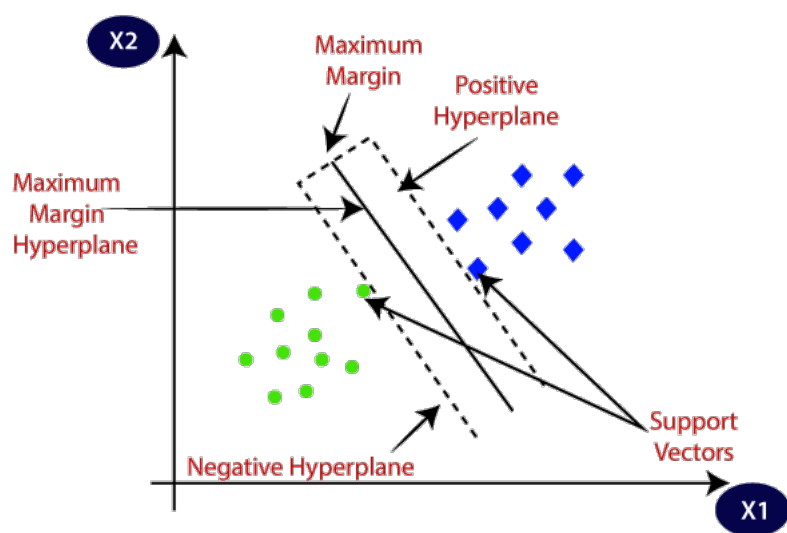


Fig. 1: **Support Vector Machine:** Two different categories classified using a decision boundary, or hyperplane. Source⁶

Support Vector Machine can be of two types:

- **Linear SVM:** A linear SVM is used for linearly separable data, which is the case of a dataset that can be classified into two distinct classes by using a single straight line.
- **Non-linear SVM:** A non-linear SVM is used for non-linearly separated data, which means that a dataset cannot be classified by using a straight line.

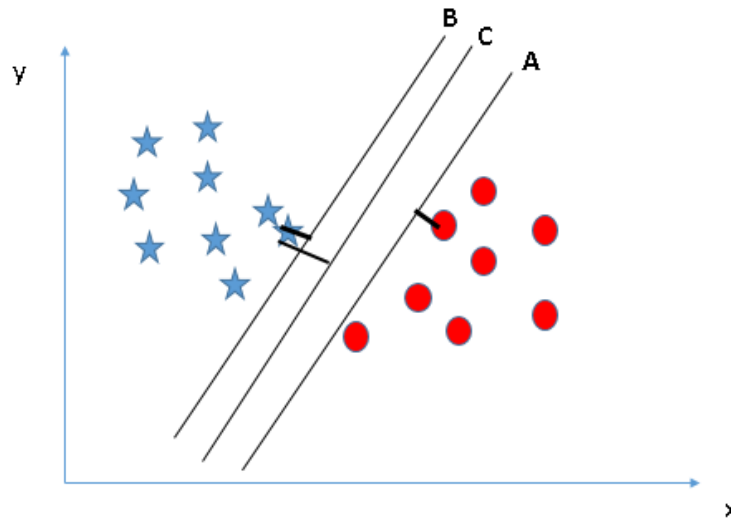
Linear SVM

Let's suppose we have a dataset that has two classes, stars and circles. The dataset has two features, $x1$ and $x2$. We want a classifier that can classify the pair $(x1, x2)$ of coordinates in either stars or circles. Consider the figure below.

Since it is a 2-dimensional space, we can separate these two classes by using a straight line. The figure shows that we have three hyperplanes, A, B, and C, which are all segregating the classes well. How can we identify the right hyperplane? The SVM algorithm finds the closest point of the lines from both of the classes. These points are called support vectors. The distance between the support vectors and the hyperplane is referred as the *margin*. The goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane. From the figure above, we see that the margin for hyperplane C is higher when compared to both A and B. Therefore, we name C as the (right) hyperplane.

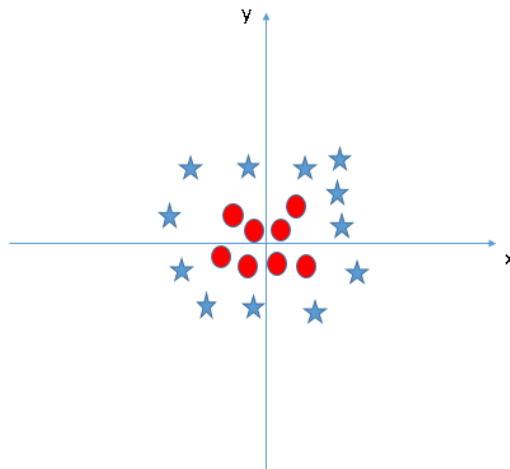
⁶ Support Vector Machine

⁷ Support Vector Machine

Fig. 2: Source⁷

Non-linear SVM

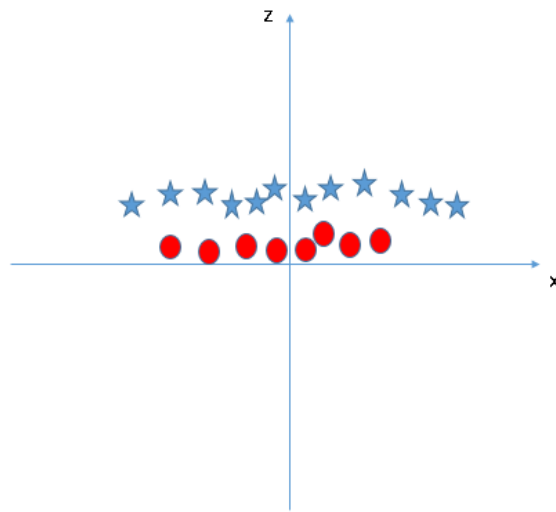
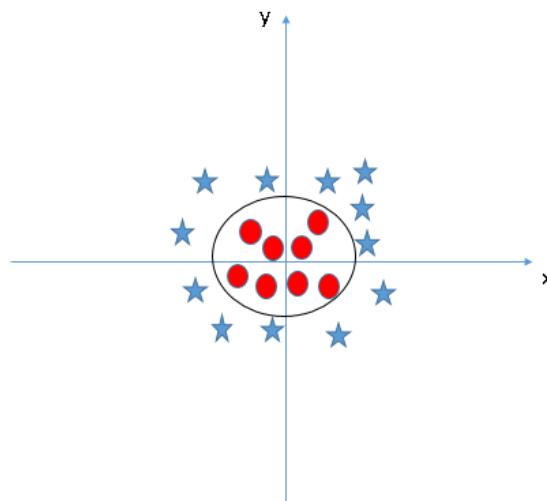
When the data is linearly arranged, we can separate it by using a straight line. However, for non-linear data, we cannot draw a single straight line. Let's consider the figure below.

Fig. 3: Source⁷

In order to separate the circles from the stars, we need to introduce an additional feature. In case of linear data, we would use the two features x and y . For this non-linear data, we will add a third dimension, z . z is defined as $z = x^2 + y^2$. By adding the third feature, our space will become as below image.

In the above figure, all values for z will always be positive, because z is the squared sum of x and y . Now, the SVM classifier will divide the dataset into two distinct classes by finding a *linear* hyperplane between these two classes.

Since now we are in a *3-dimensional* space, the hyperplane looks like a plane parallel to the x -axis. If we convert it in *2-dimensional* space with $z = 1$, then it will become as the figure below.

Fig. 4: Source⁷Fig. 5: Source⁷

(Hence, in case of non-linear data, we obtain a circumference of $radius = 1$)

In order to find the hyperplane with the SVM algorithm, we do not need to add this third dimension z manually: the SVM algorithm uses a technique called the “kernel trick”. The SVM kernel is a function which takes a low dimensional input, and it transforms it to a higher dimensional space, i.e., it converts non-linearly separable data to linearly separable data.

References

20.1 Centroid

Be the first to [contribute!](#)

20.2 Density

Be the first to [contribute!](#)

20.3 Distribution

Be the first to [contribute!](#)

20.4 Hierarchical

Be the first to [contribute!](#)

20.5 K-Means

Be the first to [contribute!](#)

20.6 Mean shift

Be the first to [contribute!](#)

References

- *Ordinary Least Squares*
- *Polynomial*
- *Lasso*
- *Ridge*
- *Stepwise*

21.1 Ordinary Least Squares

OLS is the method with which linear regression is performed. The square of the difference from the mean is taken for every data point, and the summed loss function is to be minimized.

$$l = \sum_{i=1}^n (y_i - \bar{y})^2$$

21.2 Polynomial

Polynomial regression is a modification of linear regression where the existing features are mapped to a polynomial form. The problem is still a linear regression problem, but the input vector is now mapped to a higher dimensional vector which serves as a pseudo-input vector of sorts.

$$\mathbf{x} = (x_0, x_1) \rightarrow \mathbf{x}' = (x_0, x_0^2, x_1, x_1^2, x_0x_1)$$

21.3 Lasso

Lasso Regression tries to reduce the ordinary least squares error similar to vanilla regression, but adds an extra term. The sum of the L_1 norm for every data point multiplied by a hyperparameter α is used. This reduces model complexity and prevents overfitting.

$$l = \sum_{i=1}^n (y_i - \tilde{y})^2 + \alpha \sum_{j=1}^p |w_j|$$

21.4 Ridge

Ridge regression is similar to lasso regression, but the regularization term uses the L_2 norm instead.

$$l = \sum_{i=1}^n (y_i - \tilde{y})^2 + \alpha \sum_{j=1}^p w_j^2$$

21.5 Stepwise

Stepwise regression or spline regression helps us fit a piece wise function to the data. It is usually used with linear models, but it can be generalized to higher degrees as well. The regression equation takes the form of

$$y = ax + b(x - \bar{x})H_\alpha + c$$

where H_α is the shifted Heaviside step function, having its discontinuity at α .

References

Reinforcement Learning

In machine learning, supervised is sometimes contrasted with unsupervised learning. This is a useful distinction, but there are some problem domains that have share characteristics with each without fitting exactly in either category. In cases where the algorithm does not have explicit labels but does receive a form of feedback, we are dealing with a third and distinct paradigm of machine learning - reinforcement learning.

Programmatic and a theoretical introduction to reinforcement learning:<https://spinningup.openai.com/>

There are different problem types and algorithms, but all reinforcement learning problems have the following aspects in common:

- an **agent** - the algorithm or “AI” responsible for making decisions
- an **environment**, consisting of different **states** in which the agent may find itself
- a **reward** signal which is returned by the environment as a function of the current state
- **actions**, each of which takes the agent from one state to another
- a **policy**, i.e. a mapping from states to actions that defines the agent’s behavior

The goal of reinforcement learning is to learn the optimal policy, that is the policy that maximizes expected (discounted) cumulative reward.

Many RL algorithms will include a value function or a Q-function. A value function gives the expected cumulative reward for each state under the current policy In other words, it answers the question, “If I begin in state i and follow my policy, what will be my expected reward?”

In most algorithms, expected cumulative reward is discounted by some factor $\gamma \in (0, 1)$; a typical value for γ is 0.9. In addition to more accurately modeling the behavior of humans and other animals, $\gamma < 1$ helps to ensure that algorithms converge even when there is no terminal state or when the terminal state is never found (because otherwise expected cumulative reward may also become infinite).

22.1 Note on Terminology

For mostly historical reasons, engineering and operations research use different words to talk about the same concepts. For example, the general field of reinforcement learning itself is sometimes referred to as optimal control, approximate

dynamic programming, or neuro-dynamic programming.¹

22.2 Exploraton vs. Exploitation

One dilemma inherent to the RL problem setting is the tension between the desire to choose the best known option and the need to try something new in order to discover other options that may be even better. Choosing the best known action is known as exploitation, while choosing a different action is known as exploration.

Typically, this is solved by adding to the policy a small probability of exploration. For example, the policy could be to choose the optimal action (optimal with regard to what is known) with probability 0.95, and exploring by randomly choosing some other action with probability 0.5 (if uniform across all remaining actions: probability $0.5/(n-1)$ where n is the number of states).

22.3 MDPs and Tabular methods

Many problems can be effectively modeled as Markov Decision Processes (MDPs), and usually as *Partially Observable Markov Decision Processes* (POMDPs) <https://en.wikipedia.org/wiki/Partially_observable_Markov_decision_process>. That is, we have

- a set of states S
- a set of actions A
- a set of conditional state transition probabilities T
- a reward function $R : S \times A \rightarrow \mathbb{R}$
- a set of observations Ω
- a set of condition observation probabilities O
- a discount factor $\gamma \in [0]$

Given these things, the goal is to choose the action at each time step which will maximize $E [\sum_{t=0}^{\infty} \gamma^t r_t]$, the expected discounted reward.

22.4 Monte Carlo methods

One possible approach is to run a large number of simulations to learn p^* . This is good for cases where we know the environment and can run many simulations reasonably quickly. For example, it is fairly trivial to compute an optimal policy for the card game 21 (*blackjack*) <[https://en.wikipedia.org/wiki/Twenty-One_\(card_game\)](https://en.wikipedia.org/wiki/Twenty-One_(card_game))> by running many simulations, and the same is true for most simple games.

22.5 Temporal-Difference Learning

TODO

22.6 Planning

TODO

22.7 On-Policy vs. Off-Policy Learning

TODO

22.8 Model-Free vs. Model-Based Approaches

TODO

22.9 Imitation Learning

TODO

22.10 Q-Learning

TODO

22.11 Deep Q-Learning

Deep Q-learning pursues the same general methods as Q-learning. Its innovation is to add a neural network, which makes it possible to learn a very complex Q-function. This makes it very powerful, especially because it makes a large body of well-developed theory and tools for deep learning useful to reinforcement learning problems.

22.12 Examples of Applications

TODO

22.13 Links

- [Practical Applications of Reinforcement Learning \(tTowards Data Science\)](#)
- [Reinforcement learning \(GeeksforGeeks\)](#)
- [Reinforcement Learning Algorithms: An Intuitive Overview \(SmartLabAI\)](#)

References

Public datasets in vision, nlp and more forked from caesar0301's [awesome datasets](#) wiki.

- *Agriculture*
- *Art*
- *Biology*
- *Chemistry/Materials Science*
- *Climate/Weather*
- *Complex Networks*
- *Computer Networks*
- *Data Challenges*
- *Earth Science*
- *Economics*
- *Education*
- *Energy*
- *Finance*
- *GIS*
- *Government*
- *Healthcare*
- *Image Processing*
- *Machine Learning*
- *Museums*

- *Music*
- *Natural Language*
- *Neuroscience*
- *Physics*
- *Psychology/Cognition*
- *Public Domains*
- *Search Engines*
- *Social Networks*
- *Social Sciences*
- *Software*
- *Sports*
- *Time Series*
- *Transportation*

23.1 Agriculture

- U.S. Department of Agriculture's PLANTS Database
- U.S. Department of Agriculture's Nutrient Database

23.2 Art

- Google's Quick Draw Sketch Dataset

23.3 Biology

- 1000 Genomes
- American Gut (Microbiome Project)
- Broad Bioimage Benchmark Collection (BBBC)
- Broad Cancer Cell Line Encyclopedia (CCLE)
- Cell Image Library
- Complete Genomics Public Data
- EBI ArrayExpress
- EBI Protein Data Bank in Europe
- Electron Microscopy Pilot Image Archive (EMPIAR)
- ENCODE project
- Ensembl Genomes

- [Gene Expression Omnibus \(GEO\)](#)
- [Gene Ontology \(GO\)](#)
- [Global Biotic Interactions \(GloBI\)](#)
- [Harvard Medical School \(HMS\) LINCS Project](#)
- [Human Genome Diversity Project](#)
- [Human Microbiome Project \(HMP\)](#)
- [ICOS PSP Benchmark](#)
- [International HapMap Project](#)
- [Journal of Cell Biology DataViewer](#)
- [MIT Cancer Genomics Data](#)
- [NCBI Proteins](#)
- [NCBI Taxonomy](#)
- [NCI Genomic Data Commons](#)
- [NIH Microarray data or FTP \(see FTP link on RAW\)](#)
- [OpenSNP genotypes data](#)
- [Pathguid - Protein-Protein Interactions Catalog](#)
- [Protein Data Bank](#)
- [Psychiatric Genomics Consortium](#)
- [PubChem Project](#)
- [PubGene \(now Coremine Medical\)](#)
- [Sanger Catalogue of Somatic Mutations in Cancer \(COSMIC\)](#)
- [Sanger Genomics of Drug Sensitivity in Cancer Project \(GDSC\)](#)
- [Sequence Read Archive\(SRA\)](#)
- [Stanford Microarray Data](#)
- [Stowers Institute Original Data Repository](#)
- [Systems Science of Biological Dynamics \(SSBD\) Database](#)
- [The Cancer Genome Atlas \(TCGA\), available via Broad GDAC](#)
- [The Catalogue of Life](#)
- [The Personal Genome Project or PGP](#)
- [UCSC Public Data](#)
- [UniGene](#)
- [Universal Protein Resource \(UnitProt\)](#)

23.4 Chemistry/Materials Science

- NIST Computational Chemistry Comparison and Benchmark Database - SRD 101
- Open Quantum Materials Database
- Citrination Public Datasets
- Khazana Project

23.5 Climate/Weather

- Actuaries Climate Index
- Australian Weather
- Aviation Weather Center - Consistent, timely and accurate weather information for the world airspace system
- Brazilian Weather - Historical data (In Portuguese)
- Canadian Meteorological Centre
- Climate Data from UEA (updated monthly)
- European Climate Assessment & Dataset
- Global Climate Data Since 1929
- NASA Global Imagery Browse Services
- NOAA Bering Sea Climate
- NOAA Climate Datasets
- NOAA Realtime Weather Models
- NOAA SURFRAD Meteorology and Radiation Datasets
- The World Bank Open Data Resources for Climate Change
- UEA Climatic Research Unit
- WorldClim - Global Climate Data
- WU Historical Weather Worldwide

23.6 Complex Networks

- AMiner Citation Network Dataset
- CrossRef DOI URLs
- DBLP Citation dataset
- DIMACS Road Networks Collection
- NBER Patent Citations
- Network Repository with Interactive Exploratory Analysis Tools
- NIST complex networks data collection
- Protein-protein interaction network

- PyPI and Maven Dependency Network
- Scopus Citation Database
- Small Network Data
- Stanford GraphBase (Steven Skiena)
- Stanford Large Network Dataset Collection
- Stanford Longitudinal Network Data Sources
- The Koblenz Network Collection
- The Laboratory for Web Algorithmics (UNIMI)
- The Nexus Network Repository
- UCI Network Data Repository
- UFL sparse matrix collection
- WSU Graph Database

23.7 Computer Networks

- 3.5B Web Pages from CommonCrawl 2012
- 53.5B Web clicks of 100K users in Indiana Univ.
- CAIDA Internet Datasets
- ClueWeb09 - 1B web pages
- ClueWeb12 - 733M web pages
- CommonCrawl Web Data over 7 years
- CRAWDAD Wireless datasets from Dartmouth Univ.
- Criteo click-through data
- OONI: Open Observatory of Network Interference - Internet censorship data
- Open Mobile Data by MobiPerf
- Rapid7 Sonar Internet Scans
- UCSD Network Telescope, IPv4 /8 net

23.8 Data Challenges

- Bruteforce Database
- Challenges in Machine Learning
- CrowdANALYTIX dataX
- D4D Challenge of Orange
- DrivenData Competitions for Social Good
- ICWSM Data Challenge (since 2009)
- Kaggle Competition Data

- KDD Cup by Tencent 2012
- Localytics Data Visualization Challenge
- Netflix Prize
- Space Apps Challenge
- Telecom Italia Big Data Challenge
- TravisTorrent Dataset - MSR'2017 Mining Challenge
- Yelp Dataset Challenge

23.9 Earth Science

- AQUASTAT - Global water resources and uses
- BODC - marine data of ~22K vars
- Earth Models
- EOSDIS - NASA's earth observing system data
- Integrated Marine Observing System (IMOS) - roughly 30TB of ocean measurements or on S3
- Marinexplore - Open Oceanographic Data
- Smithsonian Institution Global Volcano and Eruption Database
- USGS Earthquake Archives

23.10 Economics

- American Economic Association (AEA)
- EconData from UMD
- Economic Freedom of the World Data
- Historical Macroeconomic Statistics
- International Economics Database and various data tools
- International Trade Statistics
- Internet Product Code Database
- Joint External Debt Data Hub
- Jon Haveman International Trade Data Links
- OpenCorporates Database of Companies in the World
- Our World in Data
- SciencesPo World Trade Gravity Datasets
- The Atlas of Economic Complexity
- The Center for International Data
- The Observatory of Economic Complexity
- UN Commodity Trade Statistics

- [UN Human Development Reports](#)

23.11 Education

- [College Scorecard Data](#)
- [Student Data from Free Code Camp](#)

23.12 Energy

- [AMPds](#)
- [BLUEd](#)
- [COMBED](#)
- [Dataport](#)
- [DRED](#)
- [ECO](#)
- [EIA](#)
- [HES - Household Electricity Study, UK](#)
- [HFED](#)
- [iAWE](#)
- [PLAID - the Plug Load Appliance Identification Dataset](#)
- [REDD](#)
- [Tracebase](#)
- [UK-DALE - UK Domestic Appliance-Level Electricity](#)
- [WHITED](#)

23.13 Finance

- [CBOE Futures Exchange](#)
- [Google Finance](#)
- [Google Trends](#)
- [NASDAQ](#)
- [NYSE Market Data \(see FTP link on RAW\)](#)
- [OANDA](#)
- [OSU Financial data](#)
- [Quandl](#)
- [St Louis Federal](#)
- [Yahoo Finance](#)

23.14 GIS

- ArcGIS Open Data portal
- Cambridge, MA, US, GIS data on GitHub
- Factual Global Location Data
- Geo Spatial Data from ASU
- Geo Wiki Project - Citizen-driven Environmental Monitoring
- GeoFabrik - OSM data extracted to a variety of formats and areas
- GeoNames Worldwide
- Global Administrative Areas Database (GADM)
- Homeland Infrastructure Foundation-Level Data
- Landsat 8 on AWS
- List of all countries in all languages
- National Weather Service GIS Data Portal
- Natural Earth - vectors and rasters of the world
- OpenAddresses
- OpenStreetMap (OSM)
- Pleiades - Gazetteer and graph of ancient places
- Reverse Geocoder using OSM data & additional high-resolution data files
- TIGER/Line - U.S. boundaries and roads
- TwoFishes - Foursquare's coarse geocoder
- TZ Timezones shapfiles
- UN Environmental Data
- World boundaries from the U.S. Department of State
- World countries in multiple formats

23.15 Government

- A list of cities and countries contributed by community
- Open Data for Africa
- OpenDataSoft's list of 1,600 open data

23.16 Healthcare

- EHDP Large Health Data Sets
- Gapminder World demographic databases
- Medicare Coverage Database (MCD), U.S.

- Medicare Data Engine of medicare.gov Data
- Medicare Data File
- MeSH, the vocabulary thesaurus used for indexing articles for PubMed
- Number of Ebola Cases and Deaths in Affected Countries (2014)
- Open-ODS (structure of the UK NHS)
- OpenPaymentsData, Healthcare financial relationship data
- The Cancer Genome Atlas project (TCGA) and BigQuery table
- World Health Organization Global Health Observatory

23.17 Image Processing

- 10k US Adult Faces Database
- 2GB of Photos of Cats or Archive version
- Adience Unfiltered faces for gender and age classification
- Affective Image Classification
- Animals with attributes
- Caltech Pedestrian Detection Benchmark
- Chars74K dataset, Character Recognition in Natural Images (both English and Kannada are available)
- Face Recognition Benchmark
- GDxray: X-ray images for X-ray testing and Computer Vision
- ImageNet (in WordNet hierarchy)
- Indoor Scene Recognition
- International Affective Picture System, UFL
- Massive Visual Memory Stimuli, MIT
- MNIST database of handwritten digits, near 1 million examples
- Several Shape-from-Silhouette Datasets
- Stanford Dogs Dataset
- SUN database, MIT
- The Action Similarity Labeling (ASLAN) Challenge
- The Oxford-IIIT Pet Dataset
- Violent-Flows - Crowd Violence Non-violence Database and benchmark
- Visual genome
- YouTube Faces Database

23.18 Machine Learning

- Context-aware data sets from five domains
- Delve Datasets for classification and regression (Univ. of Toronto)
- Discogs Monthly Data
- eBay Online Auctions (2012)
- IMDb Database
- Keel Repository for classification, regression and time series
- Labeled Faces in the Wild (LFW)
- Lending Club Loan Data
- Machine Learning Data Set Repository
- Million Song Dataset
- More Song Datasets
- MovieLens Data Sets
- New Yorker caption contest ratings
- RDataMining - “R and Data Mining” ebook data
- Registered Meteorites on Earth
- Restaurants Health Score Data in San Francisco
- UCI Machine Learning Repository
- Yahoo! Ratings and Classification Data
- Youtube 8m

23.19 Museums

- Canada Science and Technology Museums Corporation’s Open Data
- Cooper-Hewitt’s Collection Database
- Minneapolis Institute of Arts metadata
- Natural History Museum (London) Data Portal
- Rijksmuseum Historical Art Collection
- Tate Collection metadata
- The Getty vocabularies

23.20 Music

- Nottingham Folk Songs
- Bach 10

23.21 Natural Language

- Automatic Keyphrase Extraction
- Blogger Corpus
- CLiPS Stylometry Investigation Corpus
- ClueWeb09 FACC
- ClueWeb12 FACC
- DBpedia - 4.58M things with 583M facts
- Flickr Personal Taxonomies
- Freebase.com of people, places, and things
- Google Books Ngrams (2.2TB)
- Google MC-AFP, generated based on the public available Gigaword dataset using Paragraph Vectors
- Google Web 5gram (1TB, 2006)
- Gutenberg eBooks List
- Hansards text chunks of Canadian Parliament
- Machine Comprehension Test (MCTest) of text from Microsoft Research
- Machine Translation of European languages
- Microsoft MACHine Reading COMprehension Dataset (or MS MARCO)
- Multi-Domain Sentiment Dataset (version 2.0)
- Open Multilingual Wordnet
- Personae Corpus
- SaudiNewsNet Collection of Saudi Newspaper Articles (Arabic, 30K articles)
- SMS Spam Collection in English
- Universal Dependencies
- USENET postings corpus of 2005~2011
- Webhose - News/Blogs in multiple languages
- Wikidata - Wikipedia databases
- Wikipedia Links data - 40 Million Entities in Context
- WordNet databases and tools

23.22 Neuroscience

- Allen Institute Datasets
- Brain Catalogue
- Brainomics
- CodeNeuro Datasets
- Collaborative Research in Computational Neuroscience (CRCNS)

- FCP-INDI
- Human Connectome Project
- NDAR
- NeuroData
- Neuroelectro
- NIMH Data Archive
- OASIS
- OpenfMRI
- Study Forrest

23.23 Physics

- CERN Open Data Portal
- Crystallography Open Database
- NASA Exoplanet Archive
- NSSDC (NASA) data of 550 space spacecraft
- Sloan Digital Sky Survey (SDSS) - Mapping the Universe

23.24 Psychology/Cognition

- OSU Cognitive Modeling Repository Datasets

23.25 Public Domains

- Amazon
- Archive-it from Internet Archive
- Archive.org Datasets
- CMU JASA data archive
- CMU StatLab collections
- Data.World
- Data360
- Datamob.org
- Google
- Infochimps
- KDNuggets Data Collections
- Microsoft Azure Data Market Free DataSets
- Microsoft Data Science for Research

- Numbray
- Open Library Data Dumps
- Reddit Datasets
- RevolutionAnalytics Collection
- Sample R data sets
- Stats4Stem R data sets
- StatSci.org
- The Washington Post List
- UCLA SOCR data collection
- UFO Reports
- Wikileaks 911 pager intercepts
- Yahoo Webscope

23.26 Search Engines

- Academic Torrents of data sharing from UMB
- Datahub.io
- DataMarket (Qlik)
- Harvard Dataverse Network of scientific data
- ICPSR (UMICH)
- Institute of Education Sciences
- National Technical Reports Library
- Open Data Certificates (beta)
- OpenDataNetwork - A search engine of all Socrata powered data portals
- Statista.com - statistics and Studies
- Zenodo - An open dependable home for the long-tail of science

23.27 Social Networks

- 72 hours #gamergate Twitter Scrape
- Ancestry.com Forum Dataset over 10 years
- Cheng-Caverlee-Lee September 2009 - January 2010 Twitter Scrape
- CMU Enron Email of 150 users
- EDRM Enron EMail of 151 users, hosted on S3
- Facebook Data Scrape (2005)
- Facebook Social Networks from LAW (since 2007)
- Foursquare from UMN/Sarwat (2013)

- GitHub Collaboration Archive
- Google Scholar citation relations
- High-Resolution Contact Networks from Wearable Sensors
- Mobile Social Networks from UMASS
- Network Twitter Data
- Reddit Comments
- Skytrax' Air Travel Reviews Dataset
- Social Twitter Data
- SourceForge.net Research Data
- Twitter Data for Online Reputation Management
- Twitter Data for Sentiment Analysis
- Twitter Graph of entire Twitter site
- Twitter Scrape Calufa May 2011
- UNIMI/LAW Social Network Datasets
- Yahoo! Graph and Social Data
- Youtube Video Social Graph in 2007,2008

23.28 Social Sciences

- ACLED (Armed Conflict Location & Event Data Project)
- Canadian Legal Information Institute
- Center for Systemic Peace Datasets - Conflict Trends, Politics, State Fragility, etc
- Correlates of War Project
- Cryptome Conspiracy Theory Items
- Datacards
- European Social Survey
- FBI Hate Crime 2013 - aggregated data
- Fragile States Index
- GDELT Global Events Database
- General Social Survey (GSS) since 1972
- German Social Survey
- Global Religious Futures Project
- Humanitarian Data Exchange
- INFORM Index for Risk Management
- Institute for Demographic Studies
- International Networks Archive

- International Social Survey Program ISSP
- International Studies Compendium Project
- James McGuire Cross National Data
- MacroData Guide by Norsk samfunnsvitenskapelig datatjeneste
- Minnesota Population Center
- MIT Reality Mining Dataset
- Notre Dame Global Adaptation Index (NG-DAIN)
- Open Crime and Policing Data in England, Wales and Northern Ireland
- Paul Hensel General International Data Page
- PewResearch Internet Survey Project
- PewResearch Society Data Collection
- Political Polarity Data
- StackExchange Data Explorer
- Terrorism Research and Analysis Consortium
- Texas Inmates Executed Since 1984
- Titanic Survival Data Set [or](#) on Kaggle
- UCB's Archive of Social Science Data (D-Lab)
- UCLA Social Sciences Data Archive
- UN Civil Society Database
- Universities Worldwide
- UPJOHN for Labor Employment Research
- Uppsala Conflict Data Program
- World Bank Open Data
- WorldPop project - Worldwide human population distributions

23.29 Software

- FLOSSmole data about free, libre, and open source software development

23.30 Sports

- Basketball (NBA/NCAA/Euro) Player Database and Statistics
- Betfair Historical Exchange Data
- Cricsheet Matches (cricket)
- Ergast Formula 1, from 1950 up to date (API)
- Football/Soccer resources (data and APIs)
- Lahman's Baseball Database

- Pinhooker: Thoroughbred Bloodstock Sale Data
- Retrosheet Baseball Statistics
- Tennis database of rankings, results, and stats for ATP, WTA, Grand Slams and Match Charting Project

23.31 Time Series

- Databanks International Cross National Time Series Data Archive
- Hard Drive Failure Rates
- Heart Rate Time Series from MIT
- Time Series Data Library (TSDL) from MU
- UC Riverside Time Series Dataset

23.32 Transportation

- Airlines OD Data 1987-2008
- Bay Area Bike Share Data
- Bike Share Systems (BSS) collection
- GeoLife GPS Trajectory from Microsoft Research
- German train system by Deutsche Bahn
- Hubway Million Rides in MA
- Marine Traffic - ship tracks, port calls and more
- Montreal BIXI Bike Share
- NYC Taxi Trip Data 2009-
- NYC Taxi Trip Data 2013 (FOIA/FOILed)
- NYC Uber trip data April 2014 to September 2014
- Open Traffic collection
- OpenFlights - airport, airline and route data
- Philadelphia Bike Share Stations (JSON)
- Plane Crash Database, since 1920
- RITA Airline On-Time Performance data
- RITA/BTS transport data collection (TranStat)
- Toronto Bike Share Stations (XML file)
- Transport for London (TFL)
- Travel Tracker Survey (TTS) for Chicago
- U.S. Bureau of Transportation Statistics (BTS)
- U.S. Domestic Flights 1990 to 2009
- U.S. Freight Analysis Framework since 2007

Libraries

Machine learning libraries and frameworks forked from [josephmisti's awesome machine learning](#).

- *APL*
- *C*
- *C++*
- *Common Lisp*
- *Clojure*
- *Elixir*
- *Erlang*
- *Go*
- *Haskell*
- *Java*
- *Javascript*
- *Julia*
- *Lua*
- *Matlab*
- *.NET*
- *Objective C*
- *OCaml*
- *PHP*
- *Python*

- *Ruby*
- *Rust*
- *R*
- *SAS*
- *Scala*
- *Swift*

24.1 APL

General-Purpose Machine Learning

- *naive-apl* - Naive Bayesian Classifier implementation in APL

24.2 C

General-Purpose Machine Learning

- *Darknet* - Darknet is an open source neural network framework written in C and CUDA. It is fast, easy to install, and supports CPU and GPU computation.
- *Recommender* - A C library for product recommendations/suggestions using collaborative filtering (CF).
- *Hybrid Recommender System* - A hybrid recomender system based upon scikit-learn algorithms.

Computer Vision

- *CCV* - C-based/Cached/Core Computer Vision Library, A Modern Computer Vision Library
- *VLFeat* - VLFeat is an open and portable library of computer vision algorithms, which has Matlab toolbox

Speech Recognition

- *HTK* -The Hidden Markov Model Toolkit. HTK is a portable toolkit for building and manipulating hidden Markov models.

24.3 C++

Computer Vision

- *DLib* - DLib has C++ and Python interfaces for face detection and training general object detectors.
- *EbLearn* - Eblearn is an object-oriented C++ library that implements various machine learning models
- *OpenCV* - OpenCV has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS.

- **VIGRA** - VIGRA is a generic cross-platform C++ computer vision and machine learning library for volumes of arbitrary dimensionality with Python bindings.

General-Purpose Machine Learning

- **BanditLib** - A simple Multi-armed Bandit library.
- **Caffe** - A deep learning framework developed with cleanliness, readability, and speed in mind. [DEEP LEARNING]
- **CNTK** by Microsoft Research, is a unified deep-learning toolkit that describes neural networks as a series of computational steps via a directed graph.
- **CUDA** - This is a fast C++/CUDA implementation of convolutional [DEEP LEARNING]
- **CXXNET** - Yet another deep learning framework with less than 1000 lines core code [DEEP LEARNING]
- **DeepDetect** - A machine learning API and server written in C++11. It makes state of the art machine learning easy to work with and integrate into existing applications.
- **Distributed Machine learning Tool Kit (DMTK)** Word Embedding.
- **DLib** - A suite of ML tools designed to be easy to imbed in other applications
- **DSSTNE** - A software library created by Amazon for training and deploying deep neural networks using GPUs which emphasizes speed and scale over experimental flexibility.
- **DyNet** - A dynamic neural network library working well with networks that have dynamic structures that change for every training instance. Written in C++ with bindings in Python.
- **encog-cpp**
- **Fido** - A highly-modular C++ machine learning library for embedded electronics and robotics.
- **igraph** - General purpose graph library
- **Intel(R) DAAL** - A high performance software library developed by Intel and optimized for Intel's architectures. Library provides algorithmic building blocks for all stages of data analytics and allows to process data in batch, online and distributed modes.
- **LightGBM** framework based on decision tree algorithms, used for ranking, classification and many other machine learning tasks.
- **MLDB** - The Machine Learning Database is a database designed for machine learning. Send it commands over a RESTful API to store data, explore it using SQL, then train machine learning models and expose them as APIs.
- **mlpack** - A scalable C++ machine learning library
- **ROOT** - A modular scientific software framework. It provides all the functionalities needed to deal with big data processing, statistical analysis, visualization and storage.
- **shark** - A fast, modular, feature-rich open-source C++ machine learning library.
- **Shogun** - The Shogun Machine Learning Toolbox
- **sofia-ml** - Suite of fast incremental algorithms.
- **Stan** - A probabilistic programming language implementing full Bayesian statistical inference with Hamiltonian Monte Carlo sampling
- **Timbl** - A software package/C++ library implementing several memory-based learning algorithms, among which IB1-IG, an implementation of k-nearest neighbor classification, and IGTREE, a decision-tree approximation of IB1-IG. Commonly used for NLP.

- [Vowpal Wabbit \(VW\)](#) - A fast out-of-core learning system.
- [Warp-CTC](#), on both CPU and GPU.
- [XGBoost](#) - A parallelized optimized general purpose gradient boosting library.

Natural Language Processing

- [BLLIP Parser](#)
- [colibri-core](#) - C++ library, command line tools, and Python binding for extracting and working with basic linguistic constructions such as n-grams and skipgrams in a quick and memory-efficient way.
- [CRF++](#) for segmenting/labeling sequential data & other Natural Language Processing tasks.
- [CRFsuite](#) for labeling sequential data.
- [frog](#) - Memory-based NLP suite developed for Dutch: PoS tagger, lemmatiser, dependency parser, NER, shallow parser, morphological analyzer.
- [libfolia](https://github.com/LanguageMachines/libfolia)(<https://github.com/LanguageMachines/libfolia>) - C++ library for the [FoLiA format
- [MeTA](https://github.com/meta-toolkit/meta)](<https://github.com/meta-toolkit/meta>) - [MeTA : ModERn Text Analysis is a C++ Data Sciences Toolkit that facilitates mining big text data.
- [MIT Information Extraction Toolkit](#) - C, C++, and Python tools for named entity recognition and relation extraction
- [ucto](#) - Unicode-aware regular-expression based tokenizer for various languages. Tool and C++ library. Supports FoLiA format.

Speech Recognition

- [Kaldi](#) - Kaldi is a toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. Kaldi is intended for use by speech recognition researchers.

Sequence Analysis

- [ToPS](#) - This is an objected-oriented framework that facilitates the integration of probabilistic models for sequences over a user defined alphabet.

Gesture Detection

- [grt](#) - The Gesture Recognition Toolkit. GRT is a cross-platform, open-source, C++ machine learning library designed for real-time gesture recognition.

24.4 Common Lisp

General-Purpose Machine Learning

- [mgl](#), Gaussian Processes
- [mgl-gpr](#) - Evolutionary algorithms
- [cl-libsvm](#) - Wrapper for the libsvm support vector machine library

24.5 Clojure

Natural Language Processing

- [Clojure-openNLP](#) - Natural Language Processing in Clojure (opennlp)
- [Infections-clj](#) - Rails-like inflection library for Clojure and ClojureScript

General-Purpose Machine Learning

- [Touchstone](#) - Clojure A/B testing library
- [Clojush](#) - The Push programming language and the PushGP genetic programming system implemented in Clojure
- [Infer](#) - Inference and machine learning in clojure
- [Clj-ML](#) - A machine learning library for Clojure built on top of Weka and friends
- [DL4CLJ](#) - Clojure wrapper for Deeplearning4j
- [Encog](#)
- [Fungp](#) - A genetic programming library for Clojure
- [Statistiker](#) - Basic Machine Learning algorithms in Clojure.
- [clortex](#) - General Machine Learning library using Numenta's Cortical Learning Algorithm
- [comportex](#) - Functionally composable Machine Learning library using Numenta's Cortical Learning Algorithm
- [cortex](#) - Neural networks, regression and feature learning in Clojure.
- [lambda-ml](#) - Simple, concise implementations of machine learning techniques and utilities in Clojure.

Data Analysis / Data Visualization

- [Incanter](#) - Incanter is a Clojure-based, R-like platform for statistical computing and graphics.
- [PigPen](#) - Map-Reduce for Clojure.
- [Envision](#) - Clojure Data Visualisation library, based on Statistiker and D3

24.6 Elixir

General-Purpose Machine Learning

- [Simple Bayes](#) - A Simple Bayes / Naive Bayes implementation in Elixir.

Natural Language Processing

- [Stemmer](#) stemming implementation in Elixir.

24.7 Erlang

General-Purpose Machine Learning

- [Disco](#) - Map Reduce in Erlang

24.8 Go

Natural Language Processing

- [go-porterstemmer](#) - A native Go clean room implementation of the Porter Stemming algorithm.
- [paicehusk](#) - Golang implementation of the Paice/Husk Stemming Algorithm.
- [snowball](#) - Snowball Stemmer for Go.
- [go-ngram](#) - In-memory n-gram index with compression.

General-Purpose Machine Learning

- [gago](#) - Multi-population, flexible, parallel genetic algorithm.
- [Go Learn](#) - Machine Learning for Go
- [go-pr](#) - Pattern recognition package in Go lang.
- [go-ml](#) - Linear / Logistic regression, Neural Networks, Collaborative Filtering and Gaussian Multivariate Distribution
- [bayesian](#) - Naive Bayesian Classification for Golang.
- [go-galib](#) - Genetic Algorithms library written in Go / golang
- [Cloudforest](#) - Ensembles of decision trees in go/golang.
- [gobrain](#) - Neural Networks written in go
- [GoNN](#) - GoNN is an implementation of Neural Network in Go Language, which includes BPNN, RBF, PCN
- [MXNet](#) - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Go, Javascript and more.
- [go-mxnet-predictor](#) - Go binding for MXNet c_predict_api to do inference with pre-trained model

Data Analysis / Data Visualization

- [go-graph](#) - Graph library for Go/golang language.
- [SVGo](#) - The Go Language library for SVG generation
- [RF](#) - Random forests implementation in Go

24.9 Haskell

General-Purpose Machine Learning

- [haskell-ml](#) - Haskell implementations of various ML algorithms.
- [HLearn](#) - a suite of libraries for interpreting machine learning models according to their algebraic structure.
- [hnn](#) - Haskell Neural Network library.
- [hopfield-networks](#) - Hopfield Networks for unsupervised learning in Haskell.
- [caffegraph](#) - A DSL for deep neural networks
- [LambdaNet](#) - Configurable Neural Networks in Haskell

24.10 Java

Natural Language Processing

- [Cortical.io](#) as quickly and intuitively as the brain.
- [CoreNLP](#) - Stanford CoreNLP provides a set of natural language analysis tools which can take raw English language text input and give the base forms of words
- [Stanford Parser](#) - A natural language parser is a program that works out the grammatical structure of sentences
- [Stanford POS Tagger](#) - A Part-Of-Speech Tagger (POS Tagger)
- [Stanford Name Entity Recognizer](#) - Stanford NER is a Java implementation of a Named Entity Recognizer.
- [Stanford Word Segmenter](#) - Tokenization of raw text is a standard pre-processing step for many NLP tasks.
- [Tregex](#), [Tsurgeon](#) and [Semgrex](#).
- [Stanford Phrasal: A Phrase-Based Translation System](#)
- [Stanford English Tokenizer](#) - Stanford Phrasal is a state-of-the-art statistical phrase-based machine translation system, written in Java.
- [Stanford Tokens Regex](#) - A tokenizer divides text into a sequence of tokens, which roughly correspond to “words”
- [Stanford Temporal Tagger](#) - SUTime is a library for recognizing and normalizing time expressions.
- [Stanford SPIED](#) - Learning entities from unlabeled text starting with seed sets using patterns in an iterative fashion
- [Stanford Topic Modeling Toolbox](#) - Topic modeling tools to social scientists and others who wish to perform analysis on datasets
- [Twitter Text Java](#) - A Java implementation of Twitter’s text processing library
- [MALLET](#) - A Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text.
- [OpenNLP](#) - a machine learning based toolkit for the processing of natural language text.
- [LingPipe](#) - A tool kit for processing text using computational linguistics.
- [ClearTK](#) components in Java and is built on top of Apache UIMA.

- [Apache cTAKES](#) is an open-source natural language processing system for information extraction from electronic medical record clinical free-text.
- [ClearNLP](#) - The ClearNLP project provides software and resources for natural language processing. The project started at the Center for Computational Language and Education Research, and is currently developed by the Center for Language and Information Research at Emory University. This project is under the Apache 2 license.
- [CogcompNLP](#) developed in the University of Illinois' Cognitive Computation Group, for example *illinois-core-utilities* which provides a set of NLP-friendly data structures and a number of NLP-related utilities that support writing NLP applications, running experiments, etc, *illinois-edison* a library for feature extraction from *illinois-core-utilities* data structures and many other packages.

General-Purpose Machine Learning

- [aerosolve](#) - A machine learning library by Airbnb designed from the ground up to be human friendly.
- [Datumbox](#) - Machine Learning framework for rapid development of Machine Learning and Statistical applications
- [ELKI](#)
- [Encog](#) - An advanced neural network and machine learning framework. Encog contains classes to create a wide variety of networks, as well as support classes to normalize and process data for these neural networks. Encog trains using multithreaded resilient propagation. Encog can also make use of a GPU to further speed processing time. A GUI based workbench is also provided to help model and train neural networks.
- [FlinkML in Apache Flink](#) - Distributed machine learning library in Flink
- [H2O](#) - ML engine that supports distributed learning on Hadoop, Spark or your laptop via APIs in R, Python, Scala, REST/JSON.
- [htm.java](#) - General Machine Learning library using Numenta's Cortical Learning Algorithm
- [java-deeplearning](#) - Distributed Deep Learning Platform for Java, Clojure, Scala
- [Mahout](#) - Distributed machine learning
- [Meka](#).
- [MLlib in Apache Spark](#) - Distributed machine learning library in Spark
- [Hydrosphere Mist](#) - a service for deployment Apache Spark MLLib machine learning models as realtime, batch or reactive web services.
- [Neuroph](#) - Neuroph is lightweight Java neural network framework
- [ORYX](#) - Lambda Architecture Framework using Apache Spark and Apache Kafka with a specialization for real-time large-scale machine learning.
- [Samoa](#) SAMOA is a framework that includes distributed machine learning for data streams with an interface to plug-in different stream processing platforms.
- [RankLib](#) - RankLib is a library of learning to rank algorithms
- [rapaio](#) - statistics, data mining and machine learning toolbox in Java
- [RapidMiner](#) - RapidMiner integration into Java code
- [Stanford Classifier](#) - A classifier is a machine learning tool that will take data items and place them into one of k classes.
- [SmileMiner](#) - Statistical Machine Intelligence & Learning Engine
- [SystemML](#) language.

- [WalnutiQ](#) - object oriented model of the human brain
- [Weka](#) - Weka is a collection of machine learning algorithms for data mining tasks
- [LBJava](#) - Learning Based Java is a modeling language for the rapid development of software systems, offers a convenient, declarative syntax for classifier and constraint definition directly in terms of the objects in the programmer's application.

Speech Recognition

- [CMU Sphinx](#) - Open Source Toolkit For Speech Recognition purely based on Java speech recognition library.

Data Analysis / Data Visualization

- [Flink](#) - Open source platform for distributed stream and batch data processing.
- [Hadoop](#) - Hadoop/HDFS
- [Spark](#) - Spark is a fast and general engine for large-scale data processing.
- [Storm](#) - Storm is a distributed realtime computation system.
- [Impala](#) - Real-time Query for Hadoop
- [DataMelt](#) - Mathematics software for numeric computation, statistics, symbolic calculations, data analysis and data visualization.
- [Dr. Michael Thomas Flanagan's Java Scientific Library](#)

Deep Learning

- [Deeplearning4j](#) - Scalable deep learning for industry with parallel GPUs

24.11 Javascript

Natural Language Processing

- [Twitter-text](#) - A JavaScript implementation of Twitter's text processing library
- [NLP.js](#) - NLP utilities in javascript and coffeescript
- [natural](#) - General natural language facilities for node
- [Knwl.js](#) - A Natural Language Processor in JS
- [Retext](#) - Extensible system for analyzing and manipulating natural language
- [TextProcessing](#) - Sentiment analysis, stemming and lemmatization, part-of-speech tagging and chunking, phrase extraction and named entity recognition.
- [NLP Compromise](#) - Natural Language processing in the browser

Data Analysis / Data Visualization

- [D3.js](#)
- [High Charts](#)
- [NVD3.js](#)
- [dc.js](#)
- [chartjs](#)
- [dimple](#)
- [amCharts](#)
- [D3xter](#) - Straight forward plotting built on D3
- [statkit](#) - Statistics kit for JavaScript
- [datakit](#) - A lightweight framework for data analysis in JavaScript
- [science.js](#) - Scientific and statistical computing in JavaScript.
- [Z3d](#) - Easily make interactive 3d plots built on Three.js
- [Sigma.js](#) - JavaScript library dedicated to graph drawing.
- [C3.js](#)- customizable library based on D3.js for easy chart drawing.
- [Datamaps](#)- Customizable SVG map/geo visualizations using D3.js.
- [ZingChart](#)- library written on Vanilla JS for big data visualization.
- [cheminfo](#) - Platform for data visualization and analysis, using the [visualizer](#) project.

General-Purpose Machine Learning

- [Convnet.js](#) - ConvNetJS is a Javascript library for training Deep Learning models[DEEP LEARNING]
- [Clusterfck](#) - Agglomerative hierarchical clustering implemented in Javascript for Node.js and the browser
- [Clustering.js](#) - Clustering algorithms implemented in Javascript for Node.js and the browser
- [Decision Trees](#) - NodeJS Implementation of Decision Tree using ID3 Algorithm
- [DN2A](#) - Digital Neural Networks Architecture
- [figue](#) - K-means, fuzzy c-means and agglomerative clustering
- [Node-fann](#) bindings for Node.js
- [Kmeans.js](#) - Simple Javascript implementation of the k-means algorithm, for node.js and the browser
- [LDA.js](#) - LDA topic modeling for node.js
- [Learning.js](#) - Javascript implementation of logistic regression/c4.5 decision tree
- [Machine Learning](#) - Machine learning library for Node.js
- [machineJS](#) - Automated machine learning, data formatting, ensembling, and hyperparameter optimization for competitions and exploration- just give it a .csv file!
- [mil-tokyo](#) - List of several machine learning libraries
- [Node-SVM](#) - Support Vector Machine for nodejs
- [Brain](#) - Neural networks in JavaScript **[Deprecated]**

- [Bayesian-Bandit](#) - Bayesian bandit implementation for Node and the browser.
- [Synaptic](#) - Architecture-free neural network library for node.js and the browser
- [kNear](#) - JavaScript implementation of the k nearest neighbors algorithm for supervised learning
- [NeuralN](#) - C++ Neural Network library for Node.js. It has advantage on large dataset and multi-threaded training.
- [kalman](#) - Kalman filter for Javascript.
- [shaman](#) - node.js library with support for both simple and multiple linear regression.
- [ml.js](#) - Machine learning and numerical analysis tools for Node.js and the Browser!
- [Pavlov.js](#) - Reinforcement learning using Markov Decision Processes
- [MXNet](#) - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Go, Javascript and more.

Misc

- [sylvester](#) - Vector and Matrix math for JavaScript.
- [simple-statistics](#) as well as in node.js.
- [regression-js](#) - A javascript library containing a collection of least squares fitting methods for finding a trend in a set of data.
- [Lyric](#) - Linear Regression library.
- [GreatCircle](#) - Library for calculating great circle distance.

24.12 Julia

General-Purpose Machine Learning

- [MachineLearning](#) - Julia Machine Learning library
- [MLBase](#) - A set of functions to support the development of machine learning algorithms
- [PGM](#) - A Julia framework for probabilistic graphical models.
- [DA](#) - Julia package for Regularized Discriminant Analysis
- [Regression](#)
- [Local Regression](#) - Local regression, so smooooth!
- [Naive Bayes](#) - Simple Naive Bayes implementation in Julia
- [Mixed Models](#) mixed-effects models
- [Simple MCMC](#) - basic mcmc sampler implemented in Julia
- [Distance](#) - Julia module for Distance evaluation
- [Decision Tree](#) - Decision Tree Classifier and Regressor
- [Neural](#) - A neural network in Julia
- [MCMC](#) - MCMC tools for Julia
- [Mamba](#) for Bayesian analysis in Julia

- [GLM](#) - Generalized linear models in Julia
- [Online Learning](#)
- [GLMNet](#) - Julia wrapper for fitting Lasso/ElasticNet GLM models using glmnet
- [Clustering](#) - Basic functions for clustering data: k-means, dp-means, etc.
- [SVM](#) - SVM's for Julia
- [Kernal Density](#) - Kernel density estimators for julia
- [Dimensionality Reduction](#) - Methods for dimensionality reduction
- [NMF](#) - A Julia package for non-negative matrix factorization
- [ANN](#) - Julia artificial neural networks
- [Mocha](#) - Deep Learning framework for Julia inspired by Caffe
- [XGBoost](#) - eXtreme Gradient Boosting Package in Julia
- [ManifoldLearning](#) - A Julia package for manifold learning and nonlinear dimensionality reduction
- [MXNet](#) - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Go, Javascript and more.
- [Merlin](#) - Flexible Deep Learning Framework in Julia
- [ROCAalysis](#) - Receiver Operating Characteristics and functions for evaluation probabilistic binary classifiers
- [GaussianMixtures](#) - Large scale Gaussian Mixture Models
- [ScikitLearn](#) - Julia implementation of the scikit-learn API
- [Knet](#) - Koç University Deep Learning Framework

Natural Language Processing

- [Topic Models](#) - TopicModels for Julia
- [Text Analysis](#) - Julia package for text analysis

Data Analysis / Data Visualization

- [Graph Layout](#) - Graph layout algorithms in pure Julia
- [Data Frames Meta](#) - Metaprogramming tools for DataFrames
- [Julia Data](#) - library for working with tabular data in Julia
- [Data Read](#) - Read files from Stata, SAS, and SPSS
- [Hypothesis Tests](#) - Hypothesis tests for Julia
- [Gadfly](#) - Crafty statistical graphics for Julia.
- [Stats](#) - Statistical tests for Julia
- [RDataSets](#) - Julia package for loading many of the data sets available in R
- [DataFrames](#) - library for working with tabular data in Julia
- [Distributions](#) - A Julia package for probability distributions and associated functions.
- [Data Arrays](#) - Data structures that allow missing values

- [Time Series](#) - Time series toolkit for Julia
- [Sampling](#) - Basic sampling algorithms for Julia

Misc Stuff / Presentations

- [DSP](#).
- [JuliaCon Presentations](#) - Presentations for JuliaCon
- [SignalProcessing](#) - Signal Processing tools for Julia
- [Images](#) - An image library for Julia

24.13 Lua

General-Purpose Machine Learning

- [Torch7](#)
- [cephes](#) - Cephes mathematical functions library, wrapped for Torch. Provides and wraps the 180+ special mathematical functions from the Cephes mathematical library, developed by Stephen L. Moshier. It is used, among many other places, at the heart of SciPy.
- [autograd](#) - Autograd automatically differentiates native Torch code. Inspired by the original Python version.
- [graph](#) - Graph package for Torch
- [randomkit](#) - Numpy's randomkit, wrapped for Torch
- [signal](#) - A signal processing toolbox for Torch-7. FFT, DCT, Hilbert, cepstrums, stft
- [nn](#) - Neural Network package for Torch
- [torchnet](#) - framework for torch which provides a set of abstractions aiming at encouraging code re-use as well as encouraging modular programming
- [nngraph](#) - This package provides graphical computation for nn library in Torch7.
- [nnx](#) - A completely unstable and experimental package that extends Torch's builtin nn library
- [rnn](#) - A Recurrent Neural Network library that extends Torch's nn. RNNs, LSTMs, GRUs, BRNNs, BLSTMs, etc.
- [dpnn](#) - Many useful features that aren't part of the main nn package.
- [dp](#) - A deep learning library designed for streamlining research and development using the Torch7 distribution. It emphasizes flexibility through the elegant use of object-oriented design patterns.
- [optim](#) - An optimization library for Torch. SGD, Adagrad, Conjugate-Gradient, LBFGS, RProp and more.
- [unsup](#).
- [manifold](#) - A package to manipulate manifolds
- [svm](#) - Torch-SVM library
- [lbfgs](#) - FFI Wrapper for liblbfgs
- [vowpalwabbit](#) - An old vowpalwabbit interface to torch.
- [OpenGM](#) - OpenGM is a C++ library for graphical modeling, and inference. The Lua bindings provide a simple way of describing graphs, from Lua, and then optimizing them with OpenGM.

- [sphagetti](#) module for torch7 by @MichaelMathieu
- [LuaSHKit](#) - A lua wrapper around the Locality sensitive hashing library SHKit
- [kernel smoothing](#) - KNN, kernel-weighted average, local linear regression smoothers
- [cutorch](#) - Torch CUDA Implementation
- [cunn](#) - Torch CUDA Neural Network Implementation
- [imgraph](#) - An image/graph library for Torch. This package provides routines to construct graphs on images, segment them, build trees out of them, and convert them back to images.
- [videograph](#) - A video/graph library for Torch. This package provides routines to construct graphs on videos, segment them, build trees out of them, and convert them back to videos.
- [saliency](#) - code and tools around integral images. A library for finding interest points based on fast integral histograms.
- [stitch](#) - allows us to use hugin to stitch images and apply same stitching to a video sequence
- [sfm](#) - A bundle adjustment/structure from motion package
- [fex](#) - A package for feature extraction in Torch. Provides SIFT and dSIFT modules.
- [OverFeat](#) - A state-of-the-art generic dense feature extractor
- [Numeric Lua](#)
- [Lunatic Python](#)
- [SciLua](#)
- [Lua - Numerical Algorithms](#)
- [Lunum](#)

Demos and Scripts

- [Core torch7 demos repository](#). * linear-regression, logistic-regression * face detector (training and detection as separate demos) * mst-based-segmenter * train-a-digit-classifier * train-autoencoder * optical flow demo * train-on-housenumbers * train-on-cifar * tracking with deep nets * kinect demo * filter-bank visualization * saliency-networks
- [Training a Convnet for the Galaxy-Zoo Kaggle challenge\(CUDA demo\)](#)
- [Music Tagging](#) - Music Tagging scripts for torch7
- [torch-datasets](#) - Scripts to load several popular datasets including: * BSR 500 * CIFAR-10 * COIL * Street View House Numbers * MNIST * NORB
- [Atari2600](#) - Scripts to generate a dataset with static frames from the Arcade Learning Environment

24.14 Matlab

Computer Vision

- [Contourlets](#) - MATLAB source code that implements the contourlet transform and its utility functions.
- [Shearlets](#) - MATLAB code for shearlet transform

- [Curvelets](#) - The Curvelet transform is a higher dimensional generalization of the Wavelet transform designed to represent images at different scales and different angles.
- [Bandlets](#) - MATLAB code for bandlet transform
- [mexopencv](#) - Collection and a development kit of MATLAB mex functions for OpenCV library

Natural Language Processing

- [NLP](#) - An NLP library for Matlab

General-Purpose Machine Learning

- [Training a deep autoencoder or a classifier on MNIST](#)
- [Convolutional-Recursive Deep Learning for 3D Object Classification](#) - Convolutional-Recursive Deep Learning for 3D Object Classification[DEEP LEARNING]
- [t-Distributed Stochastic Neighbor Embedding](#) technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets.
- [Spider](#) - The spider is intended to be a complete object orientated environment for machine learning in Matlab.
- [LibSVM](#) - A Library for Support Vector Machines
- [LibLinear](#) - A Library for Large Linear Classification
- [Machine Learning Module](#) - Class on machine w/ PDF,lectures,code
- [Caffe](#) - A deep learning framework developed with cleanliness, readability, and speed in mind.
- [Pattern Recognition Toolbox](#) - A complete object-oriented environment for machine learning in Matlab.
- [Pattern Recognition and Machine Learning](#) - This package contains the matlab implementation of the algorithms described in the book Pattern Recognition and Machine Learning by C. Bishop.
- [Optunity](#) - A library dedicated to automated hyperparameter optimization with a simple, lightweight API to facilitate drop-in replacement of grid search. Optunity is written in Python but interfaces seamlessly with MATLAB.

Data Analysis / Data Visualization

- [matlab_gbl](#) - MatlabBGL is a Matlab package for working with graphs.
- [gamic](#) - Efficient pure-Matlab implementations of graph algorithms to complement MatlabBGL's mex functions.

24.15 .NET

Computer Vision

- [OpenCVDotNet](#) - A wrapper for the OpenCV project to be used with .NET applications.
- [Emgu CV](#) - Cross platform wrapper of OpenCV which can be compiled in Mono to e run on Windows, Linus, Mac OS X, iOS, and Android.
- [AForge.NET](#) - Open source C# framework for developers and researchers in the fields of Computer Vision and Artificial Intelligence. Development has now shifted to GitHub.

- [Accord.NET](#) - Together with AForge.NET, this library can provide image processing and computer vision algorithms to Windows, Windows RT and Windows Phone. Some components are also available for Java and Android.

Natural Language Processing

- [Stanford.NLP for .NET](#) - A full port of Stanford NLP packages to .NET and also available precompiled as a NuGet package.

General-Purpose Machine Learning

- [Accord-Framework](#) - The Accord.NET Framework is a complete framework for building machine learning, computer vision, computer audition, signal processing and statistical applications.
- [Accord.MachineLearning](#) - Support Vector Machines, Decision Trees, Naive Bayesian models, K-means, Gaussian Mixture models and general algorithms such as Ransac, Cross-validation and Grid-Search for machine-learning applications. This package is part of the Accord.NET Framework.
- [DiffSharp](#) for machine learning and optimization applications. Operations can be nested to any level, meaning that you can compute exact higher-order derivatives and differentiate functions that are internally making use of differentiation, for applications such as hyperparameter optimization.
- [Vulpes](#) - Deep belief and deep learning implementation written in F# and leverages CUDA GPU execution with Alea.cuBase.
- [Encog](#) - An advanced neural network and machine learning framework. Encog contains classes to create a wide variety of networks, as well as support classes to normalize and process data for these neural networks. Encog trains using multithreaded resilient propagation. Encog can also make use of a GPU to further speed processing time. A GUI based workbench is also provided to help model and train neural networks.
- [Neural Network Designer](#) - DBMS management system and designer for neural networks. The designer application is developed using WPF, and is a user interface which allows you to design your neural network, query the network, create and configure chat bots that are capable of asking questions and learning from your feed back. The chat bots can even scrape the internet for information to return in their output as well as to use for learning.
- [Infer.NET](#) - Infer.NET is a framework for running Bayesian inference in graphical models. One can use Infer.NET to solve many different kinds of machine learning problems, from standard problems like classification, recommendation or clustering through to customised solutions to domain-specific problems. Infer.NET has been used in a wide variety of domains including information retrieval, bioinformatics, epidemiology, vision, and many others.

Data Analysis / Data Visualization

- [numl](#) - numl is a machine learning library intended to ease the use of using standard modeling techniques for both prediction and clustering.
- [Math.NET Numerics](#) - Numerical foundation of the Math.NET project, aiming to provide methods and algorithms for numerical computations in science, engineering and every day use. Supports .Net 4.0, .Net 3.5 and Mono on Windows, Linux and Mac; Silverlight 5, WindowsPhone/SL 8, WindowsPhone 8.1 and Windows 8 with PCL Portable Profiles 47 and 344; Android/iOS with Xamarin.
- [Sho](#) to enable fast and flexible prototyping. The environment includes powerful and efficient libraries for linear algebra as well as data visualization that can be used from any .NET language, as well as a feature-rich interactive shell for rapid development.

24.16 Objective C

General-Purpose Machine Learning

- [YCML](#).
- [MLPNeuralNet](#) - Fast multilayer perceptron neural network library for iOS and Mac OS X. MLPNeuralNet predicts new examples by trained neural network. It is built on top of the Apple's Accelerate Framework, using vectorized operations and hardware acceleration if available.
- [MACHINELearning](#) - An Objective-C multilayer perceptron library, with full support for training through back-propagation. Implemented using vDSP and vecLib, it's 20 times faster than its Java equivalent. Includes sample code for use from Swift.
- [BPN-NeuralNetwork](#). This network can be used in products recommendation, user behavior analysis, data mining and data analysis.
- [Multi-Perceptron-NeuralNetwork](#) and designed unlimited-hidden-layers.
- [KRHebbian-Algorithm](#) in neural network of Machine Learning.
- [KRRKmeans-Algorithm](#) - It implemented K-Means the clustering and classification algorithm. It could be used in data mining and image compression.
- [KRFuzzyCMeans-Algorithm](#) the fuzzy clustering / classification algorithm on Machine Learning. It could be used in data mining and image compression.

24.17 OCaml

General-Purpose Machine Learning

- [Oml](#) - A general statistics and machine learning library.
- [GPR](#) - Efficient Gaussian Process Regression in OCaml.
- [Libra-Tk](#) - Algorithms for learning and inference with discrete probabilistic models.
- [TensorFlow](#) - OCaml bindings for TensorFlow.

24.18 PHP

Natural Language Processing

- [jieba-php](#) - Chinese Words Segmentation Utilities.

General-Purpose Machine Learning

- [PHP-ML](#) - Machine Learning library for PHP. Algorithms, Cross Validation, Neural Network, Preprocessing, Feature Extraction and much more in one library.
- [PredictionBuilder](#) - A library for machine learning that builds predictions using a linear regression.
- [Rubix ML](#) - A high-level machine learning and deep learning library for the PHP language.

24.19 Python

Computer Vision

- **Scikit-Image** - A collection of algorithms for image processing in Python.
- **SimpleCV** - An open source computer vision framework that gives access to several high-powered computer vision libraries, such as OpenCV. Written on Python and runs on Mac, Windows, and Ubuntu Linux.
- **Vigranumpy** - Python bindings for the VIGRA C++ computer vision library.
- **OpenFace** - Free and open source face recognition with deep neural networks.
- **PCV** - Open source Python module for computer vision

Natural Language Processing

- **NLTK** - A leading platform for building Python programs to work with human language data.
- **Pattern** - A web mining module for the Python programming language. It has tools for natural language processing, machine learning, among others.
- **Quepy** - A python framework to transform natural language questions to queries in a database query language
- **TextBlob** tasks. Stands on the giant shoulders of NLTK and Pattern, and plays nicely with both.
- **YAlign** - A sentence aligner, a friendly tool for extracting parallel sentences from comparable corpora.
- **jieba** - Chinese Words Segmentation Utilities.
- **SnowNLP** - A library for processing Chinese text.
- **spammy** - A library for email Spam filtering built on top of nltk
- **loso** - Another Chinese segmentation library.
- **genius** - A Chinese segment base on Conditional Random Field.
- **KoNLPy** - A Python package for Korean natural language processing.
- **nut** - Natural language Understanding Toolkit
- **Rosetta**
- **BLLIP Parser**
- **PyNLPI**[(<https://github.com/proycon/pynlpl>)] - Python Natural Language Processing Library. General purpose NLP library for Python. Also contains some specific modules for parsing common NLP formats, most notably for [FoLiA, but also ARPA language models, Moses phrasatables, GIZA++ alignments.
- **python-ucto**
- **python-frog**
- **python-zpar**[(<https://github.com/EducationalTestingService/python-zpar>)] - Python bindings for [ZPar, a statistical part-of-speech-tagger, constituency parser, and dependency parser for English.
- **colibri-core** - Python binding to C++ library for extracting and working with with basic linguistic constructions such as n-grams and skipgrams in a quick and memory-efficient way.
- **spaCy** - Industrial strength NLP with Python and Cython.
- **PyStanfordDependencies** - Python interface for converting Penn Treebank trees to Stanford Dependencies.
- **Distance** - Levenshtein and Hamming distance computation

- [Fuzzy Wuzzy](#) - Fuzzy String Matching in Python
- [jellyfish](#) - a python library for doing approximate and phonetic matching of strings.
- [editdistance](#) - fast implementation of edit distance
- [textacy](#) - higher-level NLP built on Spacy
- [stanford-corenlp-python](https://github.com/dasmith/stanford-corenlp-python)(<https://github.com/dasmith/stanford-corenlp-python>) - Python wrapper for [Stanford CoreNLP]

General-Purpose Machine Learning

- [auto_ml](#) - Automated machine learning for production and analytics. Lets you focus on the fun parts of ML, while outputting production-ready code, and detailed analytics of your dataset and results. Includes support for NLP, XGBoost, LightGBM, and soon, deep learning.
- [machine_learning](https://github.com/jefflevesque/machine-learning)(<https://github.com/jefflevesque/machine-learning>) - automated build consisting of a [web-interface](<https://github.com/jefflevesque/machine-learning#web-interface>), and set of [programmatic-interface], are stored into a NoSQL datastore.
- [XGBoost Library](#)
- [Bayesian Methods for Hackers](#) - Book/iPython notebooks on Probabilistic Programming in Python
- [Featureforge](#) A set of tools for creating and testing machine learning features, with a scikit-learn compatible API
- [MLlib in Apache Spark](#) - Distributed machine learning library in Spark
- [Hydrosphere Mist](#) - a service for deployment Apache Spark MLLib machine learning models as realtime, batch or reactive web services.
- [scikit-learn](#) - A Python module for machine learning built on top of SciPy.
- [metric-learn](#) - A Python module for metric learning.
- [SimpleAI](#) Python implementation of many of the artificial intelligence algorithms described on the book “Artificial Intelligence, a Modern Approach”. It focuses on providing an easy to use, well documented and tested library.
- [astroML](#) - Machine Learning and Data Mining for Astronomy.
- [graphlab-create](#) implemented on top of a disk-backed DataFrame.
- [BigML](#) - A library that contacts external servers.
- [pattern](#) - Web mining module for Python.
- [NuPIC](#) - Numenta Platform for Intelligent Computing.
- [Pylearn2](https://github.com/lisa-lab/pylearn2)(<https://github.com/lisa-lab/pylearn2>) - A Machine Learning library based on [Theano].
- [keras](https://github.com/fchollet/keras)(<https://github.com/fchollet/keras>) - Modular neural network library based on [Theano].
- [Lasagne](#) - Lightweight library to build and train neural networks in Theano.
- [hebel](#) - GPU-Accelerated Deep Learning Library in Python.
- [Chainer](#) - Flexible neural network framework
- [prophet](#) - Fast and automated time series forecasting framework by Facebook.
- [gensim](#) - Topic Modelling for Humans.
- [topik](#) - Topic modelling toolkit

- [PyBrain](#) - Another Python Machine Learning Library.
- [Brainstorm](#) - Fast, flexible and fun neural networks. This is the successor of PyBrain.
- [Crab](#) - A exible, fast recommender engine.
- [python-recsys](#) - A Python library for implementing a Recommender System.
- [thinking bayes](#) - Book on Bayesian Analysis
- [Image-to-Image Translation with Conditional Adversarial Networks](#)(<https://github.com/williamFalcon/pix2pix-keras>) - Implementation of image to image (pix2pix) translation from the paper by [isola et al.[DEEP LEARNING]
- [Restricted Boltzmann Machines](#) -Restricted Boltzmann Machines in Python. [DEEP LEARNING]
- [Bolt](#) - Bolt Online Learning Toolbox
- [CoverTree](#) - Python implementation of cover trees, near-drop-in replacement for `scipy.spatial.kdtree`
- [nilearn](#) - Machine learning for NeuroImaging in Python
- [imbalanced-learn](#) - Python module to perform under sampling and over sampling with various techniques.
- [Shogun](#) - The Shogun Machine Learning Toolbox
- [Pyevolve](#) - Genetic algorithm framework.
- [Caffe](#) - A deep learning framework developed with cleanliness, readability, and speed in mind.
- [breze](#) - Theano based library for deep and recurrent neural networks
- [pyhsmm](#), focusing on the Bayesian Nonparametric extensions, the HDP-HMM and HDP-HSMM, mostly with weak-limit approximations.
- [mrjob](#) - A library to let Python program run on Hadoop.
- [SKLL](#) - A wrapper around scikit-learn that makes it simpler to conduct experiments.
- [neurolab](#) - <https://github.com/zueve/neurolab>
- [Spearmint](#) - Spearmint is a package to perform Bayesian optimization according to the algorithms outlined in the paper: Practical Bayesian Optimization of Machine Learning Algorithms. Jasper Snoek, Hugo Larochelle and Ryan P. Adams. Advances in Neural Information Processing Systems, 2012.
- [Pebl](#) - Python Environment for Bayesian Learning
- [Theano](#) - Optimizing GPU-meta-programming code generating array oriented optimizing math compiler in Python
- [TensorFlow](#) - Open source software library for numerical computation using data flow graphs
- [yahmm](#) - Hidden Markov Models for Python, implemented in Cython for speed and efficiency.
- [python-timbl](#) - A Python extension module wrapping the full TiMBL C++ programming interface. Timbl is an elaborate k-Nearest Neighbours machine learning toolkit.
- [deap](#) - Evolutionary algorithm framework.
- [pydeep](#) - Deep Learning In Python
- [mlxtend](#) - A library consisting of useful tools for data science and machine learning tasks.
- [neon](#)(<https://github.com/NervanaSystems/neon>) - Nervana's [high-performance Python-based Deep Learning framework [DEEP LEARNING]
- [Optunity](#) - A library dedicated to automated hyperparameter optimization with a simple, lightweight API to facilitate drop-in replacement of grid search.

- [Neural Networks and Deep Learning](#) - Code samples for my book “Neural Networks and Deep Learning” [DEEP LEARNING]
- [Annoy](#) - Approximate nearest neighbours implementation
- [skflow](#) - Simplified interface for TensorFlow, mimicking Scikit Learn.
- [TPOT](#) - Tool that automatically creates and optimizes machine learning pipelines using genetic programming. Consider it your personal data science assistant, automating a tedious part of machine learning.
- [pgmpy](#) A python library for working with Probabilistic Graphical Models.
- [DIGITS](#) is a web application for training deep learning models.
- [Orange](#) - Open source data visualization and data analysis for novices and experts.
- [MXNet](#) - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Go, Javascript and more.
- [milk](#) - Machine learning toolkit focused on supervised classification.
- [TFLearn](#) - Deep learning library featuring a higher-level API for TensorFlow.
- [REP](#) - an IPython-based environment for conducting data-driven research in a consistent and reproducible way. REP is not trying to substitute scikit-learn, but extends it and provides better user experience.
- [rgf_python](#) Library.
- [gym](#) - OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms.
- [skbayes](#) - Python package for Bayesian Machine Learning with scikit-learn API
- [fuku-ml](#) - Simple machine learning library, including Perceptron, Regression, Support Vector Machine, Decision Tree and more, it's easy to use and easy to learn for beginners.

Data Analysis / Data Visualization

- [SciPy](#) - A Python-based ecosystem of open-source software for mathematics, science, and engineering.
- [NumPy](#) - A fundamental package for scientific computing with Python.
- [Numba](#) compiler to LLVM aimed at scientific Python by the developers of Cython and NumPy.
- [NetworkX](#) - A high-productivity software for complex networks.
- [igraph](#) - binding to igraph library - General purpose graph library
- [Pandas](#) - A library providing high-performance, easy-to-use data structures and data analysis tools.
- [Open Mining](#)
- [PyMC](#) - Markov Chain Monte Carlo sampling toolkit.
- [zipline](#) - A Pythonic algorithmic trading library.
- [PyDy](#) - Short for Python Dynamics, used to assist with workflow in the modeling of dynamic motion based around NumPy, SciPy, IPython, and matplotlib.
- [SymPy](#) - A Python library for symbolic mathematics.
- [statsmodels](#) - Statistical modeling and econometrics in Python.
- [astropy](#) - A community Python library for Astronomy.
- [matplotlib](#) - A Python 2D plotting library.
- [bokeh](#) - Interactive Web Plotting for Python.

- [plotly](#) - Collaborative web plotting for Python and matplotlib.
- [vincent](#) - A Python to Vega translator.
- [d3py](#)(<https://github.com/mikedewar/d3py>) - A plotting library for Python, based on [D3.js.
- [PyDexter](#) - Simple plotting for Python. Wrapper for D3xterjs; easily render charts in-browser.
- [ggplot](#) - Same API as ggplot2 for R.
- [ggfortify](#) - Unified interface to ggplot2 popular R packages.
- [Kartograph.py](#) - Rendering beautiful SVG maps in Python.
- [pygal](#) - A Python SVG Charts Creator.
- [PyQtGraph](#) - A pure-python graphics and GUI library built on PyQt4 / PySide and NumPy.
- [pycascading](#)
- [Petrel](#) - Tools for writing, submitting, debugging, and monitoring Storm topologies in pure Python.
- [Blaze](#) - NumPy and Pandas interface to Big Data.
- [emcee](#) - The Python ensemble sampling toolkit for affine-invariant MCMC.
- [windML](#) - A Python Framework for Wind Energy Analysis and Prediction
- [vispy](#) - GPU-based high-performance interactive OpenGL 2D/3D data visualization library
- [cerebro2](#) A web-based visualization and debugging platform for NuPIC.
- [NuPIC Studio](#) An all-in-one NuPIC Hierarchical Temporal Memory visualization and debugging super-tool!
- [SparklingPandas](#)
- [Seaborn](#) - A python visualization library based on matplotlib
- [bqplot](#)
- [pastalog](#) - Simple, realtime visualization of neural network training performance.
- [caravel](#) - A data exploration platform designed to be visual, intuitive, and interactive.
- [Dora](#) - Tools for exploratory data analysis in Python.
- [Ruffus](#) - Computation Pipeline library for python.
- [SOMPY](#).
- [somoclu](#) Massively parallel self-organizing maps: accelerate training on multicore CPUs, GPUs, and clusters, has python API.
- [HDBScan](#) - implementation of the hdbscan algorithm in Python - used for clustering
- [visualize_ML](#) - A python package for data exploration and data analysis.
- [scikit-plot](#) - A visualization library for quick and easy generation of common plots in data analysis and machine learning.

Neural networks

- [Neural networks](#) - NeuralTalk is a Python+numpy project for learning Multimodal Recurrent Neural Networks that describe images with sentences.
- [Neuron](#) neural networks learned with Gradient descent or Levenberg–Marquardt algorithm.

- [Data Driven Code](#) - Very simple implementation of neural networks for dummies in python without using any libraries, with detailed comments.

24.20 Ruby

Natural Language Processing

- [Treat](#) - Text REtrieval and Annotation Toolkit, definitely the most comprehensive toolkit I've encountered so far for Ruby
- [Ruby Linguistics](#) - Linguistics is a framework for building linguistic utilities for Ruby objects in any language. It includes a generic language-independent front end, a module for mapping language codes into language names, and a module which contains various English-language utilities.
- [Stemmer](#) - Expose libstemmer_c to Ruby
- [Ruby Wordnet](#) - This library is a Ruby interface to WordNet
- [Raspel](#) - raspel is an interface binding for ruby
- [UEA Stemmer](#) - Ruby port of UEALite Stemmer - a conservative stemmer for search and indexing
- [Twitter-text-rb](#) - A library that does auto linking and extraction of usernames, lists and hashtags in tweets

General-Purpose Machine Learning

- [Ruby Machine Learning](#) - Some Machine Learning algorithms, implemented in Ruby
- [Machine Learning Ruby](#)
- [jRuby Mahout](#) - JRuby Mahout is a gem that unleashes the power of Apache Mahout in the world of JRuby.
- [CardMagic-Classifer](#) - A general classifier module to allow Bayesian and other types of classifications.
- [rb-libsvm](#) - Ruby language bindings for LIBSVM which is a Library for Support Vector Machines
- [Random Forester](#) - Creates Random Forest classifiers from PMML files

Data Analysis / Data Visualization

- [rsruby](#) - Ruby - R bridge
- [data-visualization-ruby](#) - Source code and supporting content for my Ruby Manor presentation on Data Visualisation with Ruby
- [ruby-plot](#) - gnuplot wrapper for ruby, especially for plotting roc curves into svg files
- [plot-rb](#) - A plotting library in Ruby built on top of Vega and D3.
- [scruffy](#) - A beautiful graphing toolkit for Ruby
- [SciRuby](#)
- [Glean](#) - A data management tool for humans
- [Bioruby](#)
- [Arel](#)

Misc

- [Big Data For Chimps](#)
- [Listof\]\(https://github.com/kevincobain2000/listof\)](https://github.com/kevincobain2000/listof) - Community based data collection, packed in gem. Get list of pretty much anything (stop words, countries, non words) in txt, json or hash. [Demo/Search for a list

24.21 Rust

General-Purpose Machine Learning

- [deeplearn-rs](#) - deeplearn-rs provides simple networks that use matrix multiplication, addition, and ReLU under the MIT license.
- [rustlearn](#) - a machine learning framework featuring logistic regression, support vector machines, decision trees and random forests.
- [rusty-machine](#) - a pure-rust machine learning library.
- [leaf\]\(https://github.com/autumnai/leaf\)](https://github.com/autumnai/leaf) - open source framework for machine intelligence, sharing concepts from TensorFlow and Caffe. Available under the MIT license. **[**[Deprecated]**]**
- [RustNN](#) - RustNN is a feedforward neural network library.

24.22 R

General-Purpose Machine Learning

- [ahaz](#) - ahaz: Regularization for semiparametric additive hazards regression
- [arules](#) - arules: Mining Association Rules and Frequent Itemsets
- [biglasso](#) - biglasso: Extending Lasso Model Fitting to Big Data in R
- [bigrf](#) - bigrf: Big Random Forests: Classification and Regression Forests for Large Data Sets
- **[‘bigRR <http://cran.r-project.org/web/packages/bigRR/index.html>](http://cran.r-project.org/web/packages/bigRR/index.html) - **bigRR: Generalized Ridge Regression (with special advantage for $p \gg n$ cases)**’[__](#)**
- [bmrm](#) - bmrm: Bundle Methods for Regularized Risk Minimization Package
- [Boruta](#) - Boruta: A wrapper algorithm for all-relevant feature selection
- [bst](#) - bst: Gradient Boosting
- [C50](#) - C50: C5.0 Decision Trees and Rule-Based Models
- [caret](#) - Classification and Regression Training: Unified interface to ~150 ML algorithms in R.
- [caretEnsemble](#) - caretEnsemble: Framework for fitting multiple caret models as well as creating ensembles of such models.
- [Clever Algorithms For Machine Learning](#)
- [CORElearn](#) - CORElearn: Classification, regression, feature evaluation and ordinal evaluation
- [CoxBoost](#) - CoxBoost: Cox models by likelihood based boosting for a single survival endpoint or competing risks
- [Cubist](#) - Cubist: Rule- and Instance-Based Regression Modeling

- [e1071](#), TU Wien
- [earth](#) - earth: Multivariate Adaptive Regression Spline Models
- [elasticnet](#) - elasticnet: Elastic-Net for Sparse Estimation and Sparse PCA
- [ElemStatLearn](#) - ElemStatLearn: Data sets, functions and examples from the book: “The Elements of Statistical Learning, Data Mining, Inference, and Prediction” by Trevor Hastie, Robert Tibshirani and Jerome Friedman Prediction” by Trevor Hastie, Robert Tibshirani and Jerome Friedman
- [evtree](#) - evtree: Evolutionary Learning of Globally Optimal Trees
- [forecast](#) - forecast: Timeseries forecasting using ARIMA, ETS, STLM, TBATS, and neural network models
- [forecastHybrid](#) - forecastHybrid: Automatic ensemble and cross validation of ARIMA, ETS, STLM, TBATS, and neural network models from the “forecast” package
- [fpc](#) - fpc: Flexible procedures for clustering
- [frbs](#) - frbs: Fuzzy Rule-based Systems for Classification and Regression Tasks
- [GAMBoost](#) - GAMBoost: Generalized linear and additive models by likelihood based boosting
- [gamboostLSS](#) - gamboostLSS: Boosting Methods for GAMLSS
- [gbm](#) - gbm: Generalized Boosted Regression Models
- [glmnet](#) - glmnet: Lasso and elastic-net regularized generalized linear models
- [glmpath](#) - glmpath: L1 Regularization Path for Generalized Linear Models and Cox Proportional Hazards Model
- [GMMBoost](#) - GMMBoost: Likelihood-based Boosting for Generalized mixed models
- [grplasso](#) - grplasso: Fitting user specified models with Group Lasso penalty
- [grpreg](#) - grpreg: Regularization paths for regression models with grouped covariates
- [h2o](#) - A framework for fast, parallel, and distributed machine learning algorithms at scale – Deeplearning, Random forests, GBM, KMeans, PCA, GLM
- [hda](#) - hda: Heteroscedastic Discriminant Analysis
- [Introduction to Statistical Learning](#)
- [ipred](#) - ipred: Improved Predictors
- [kernlab](#) - kernlab: Kernel-based Machine Learning Lab
- [klaR](#) - klaR: Classification and visualization
- [lars](#) - lars: Least Angle Regression, Lasso and Forward Stagewise
- [lasso2](#) - lasso2: L1 constrained estimation aka ‘lasso’
- [LiblineaR](#) - LiblineaR: Linear Predictive Models Based On The Liblinear C/C++ Library
- [LogicReg](#) - LogicReg: Logic Regression
- [Machine Learning For Hackers](#)
- [maptree](#) - maptree: Mapping, pruning, and graphing tree models
- [mboost](#) - mboost: Model-Based Boosting
- [medley](#) - medley: Blending regression models, using a greedy stepwise approach
- [mlr](#) - mlr: Machine Learning in R
- [mvpart](#) - mvpart: Multivariate partitioning

- `ncvreg` - `ncvreg`: Regularization paths for SCAD- and MCP-penalized regression models
- `nnet` - `nnet`: Feed-forward Neural Networks and Multinomial Log-Linear Models
- `oblique.tree` - `oblique.tree`: Oblique Trees for Classification Data
- `pamr` - `pamr`: Pam: prediction analysis for microarrays
- `party` - `party`: A Laboratory for Recursive Partytioning
- `partykit` - `partykit`: A Toolkit for Recursive Partytioning
- `penalized` - `penalized`: penalized estimation in GLMs and in the Cox model
- `penalizedLDA` - `penalizedLDA`: Penalized classification using Fisher's linear discriminant
- `penalizedSVM` - `penalizedSVM`: Feature Selection SVM using penalty functions
- `quantregForest` - `quantregForest`: Quantile Regression Forests
- `randomForest` - `randomForest`: Breiman and Cutler's random forests for classification and regression
- `randomForestSRC`
- `rattle` - `rattle`: Graphical user interface for data mining in R
- `rda` - `rda`: Shrunk Centroids Regularized Discriminant Analysis
- `rdetools` in Feature Spaces
- `REEMtree` Data
- `relaxo` - `relaxo`: Relaxed Lasso
- `rgenoud` - `rgenoud`: R version of GENetic Optimization Using Derivatives
- `rgp` - `rgp`: R genetic programming framework
- `Rmalschains` in R
- `rminer` in classification and regression
- `ROCR` - `ROCR`: Visualizing the performance of scoring classifiers
- `RoughSets` - `RoughSets`: Data Analysis Using Rough Set and Fuzzy Rough Set Theories
- `rpart` - `rpart`: Recursive Partitioning and Regression Trees
- `RPMM` - `RPMM`: Recursively Partitioned Mixture Model
- `RSNNS`
- `RWeka` - `RWeka`: R/Weka interface
- `RXshrink` - `RXshrink`: Maximum Likelihood Shrinkage via Generalized Ridge or Least Angle Regression
- `sda` - `sda`: Shrinkage Discriminant Analysis and CAT Score Variable Selection
- `SDDA` - `SDDA`: Stepwise Diagonal Discriminant Analysis
- `SuperLearner` [<https://github.com/ecpolley/SuperLearner>] and `subsemble` - Multi-algorithm ensemble learning packages.
- `svmpath` - `svmpath`: `svmpath`: the SVM Path algorithm
- `tgp` - `tgp`: Bayesian treed Gaussian process models
- `tree` - `tree`: Classification and regression trees
- `varSelRF` - `varSelRF`: Variable selection using random forests

- [XGBoost.R](#) Library
- [Optunity](#) - A library dedicated to automated hyperparameter optimization with a simple, lightweight API to facilitate drop-in replacement of grid search. Optunity is written in Python but interfaces seamlessly to R.
- [igraph](#) - binding to igraph library - General purpose graph library
- [MXNet](#) - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Go, Javascript and more.
- [TDSP-Utilities](#).

Data Analysis / Data Visualization

- [ggplot2](#) - A data visualization package based on the grammar of graphics.

24.23 SAS

General-Purpose Machine Learning

- [Enterprise Miner](#) - Data mining and machine learning that creates deployable models using a GUI or code.
- [Factory Miner](#) - Automatically creates deployable machine learning models across numerous market or customer segments using a GUI.

Data Analysis / Data Visualization

- [SAS/STAT](#) - For conducting advanced statistical analysis.
- [University Edition](#) - FREE! Includes all SAS packages necessary for data analysis and visualization, and includes online SAS courses.

High Performance Machine Learning

- [High Performance Data Mining](#) - Data mining and machine learning that creates deployable models using a GUI or code in an MPP environment, including Hadoop.
- [High Performance Text Mining](#) - Text mining using a GUI or code in an MPP environment, including Hadoop.

Natural Language Processing

- [Contextual Analysis](#) - Add structure to unstructured text using a GUI.
- [Sentiment Analysis](#) - Extract sentiment from text using a GUI.
- [Text Miner](#) - Text mining using a GUI or code.

Demos and Scripts

- [ML_Tables](#) - Concise cheat sheets containing machine learning best practices.
- [enlighten-apply](#) - Example code and materials that illustrate applications of SAS machine learning techniques.

- [enlighten-integration](#) - Example code and materials that illustrate techniques for integrating SAS with other analytics technologies in Java, PMML, Python and R.
- [enlighten-deep](#) - Example code and materials that illustrate using neural networks with several hidden layers in SAS.
- [dm-flow](#) - Library of SAS Enterprise Miner process flow diagrams to help you learn by example about specific data mining topics.

24.24 Scala

Natural Language Processing

- [ScalaNLP](#) - ScalaNLP is a suite of machine learning and numerical computing libraries.
- [Breeze](#) - Breeze is a numerical processing library for Scala.
- [Chalk](#) - Chalk is a natural language processing library.
- [FACTORIE](#) - FACTORIE is a toolkit for deployable probabilistic modeling, implemented as a software library in Scala. It provides its users with a succinct language for creating relational factor graphs, estimating parameters and performing inference.

Data Analysis / Data Visualization

- [MLlib in Apache Spark](#) - Distributed machine learning library in Spark
- [Hydrosphere Mist](#) - a service for deployment Apache Spark MLLib machine learning models as realtime, batch or reactive web services.
- [Scalding](#) - A Scala API for Cascading
- [Summing Bird](#) - Streaming MapReduce with Scalding and Storm
- [Algebird](#) - Abstract Algebra for Scala
- [xerial](#) - Data management utilities for Scala
- [simmer](#) - Reduce your data. A unix filter for algebird-powered aggregation.
- [PredictionIO](#) - PredictionIO, a machine learning server for software developers and data engineers.
- [BIDMat](#) - CPU and GPU-accelerated matrix library intended to support large-scale exploratory data analysis.
- [Wolfe Declarative Machine Learning](#)
- [Flink](#) - Open source platform for distributed stream and batch data processing.
- [Spark Notebook](#) - Interactive and Reactive Data Science using Scala and Spark.

General-Purpose Machine Learning

- [Conjecture](#) - Scalable Machine Learning in Scalding
- [brushfire](#) - Distributed decision tree ensemble learning in Scala
- [ganitha](#) - scalding powered machine learning
- [adam](#) - A genomics processing engine and specialized file format built using Apache Avro, Apache Spark and Parquet. Apache 2 licensed.

- [bioscala](#) - Bioinformatics for the Scala programming language
- [BIDMach](#) - CPU and GPU-accelerated Machine Learning Library.
- [Figaro](#) - a Scala library for constructing probabilistic models.
- [H2O Sparkling Water](#) - H2O and Spark interoperability.
- [FlinkML in Apache Flink](#) - Distributed machine learning library in Flink
- [DynaML](#) - Scala Library/REPL for Machine Learning Research
- [Saul](#) - Flexible Declarative Learning-Based Programming.
- [SwiftLearner](#) - Simply written algorithms to help study ML or write your own implementations.

24.25 Swift

General-Purpose Machine Learning

- [Swift AI](#) - Highly optimized artificial intelligence and machine learning library written in Swift.
- [BrainCore](#) - The iOS and OS X neural network framework
- [swix](#) - A bare bones library that includes a general matrix language and wraps some OpenCV for iOS development.
- [DeepLearningKit](#) an Open Source Deep Learning Framework for Apple's iOS, OS X and tvOS. It currently allows using deep convolutional neural network models trained in Caffe on Apple operating systems.
- [AIToolbox](#) - A toolbox framework of AI modules written in Swift: Graphs/Trees, Linear Regression, Support Vector Machines, Neural Networks, PCA, KMeans, Genetic Algorithms, MDP, Mixture of Gaussians.
- [MLKit](#) - A simple Machine Learning Framework written in Swift. Currently features Simple Linear Regression, Polynomial Regression, and Ridge Regression.
- [Swift Brain](#) - The first neural network / machine learning library written in Swift. This is a project for AI algorithms in Swift for iOS and OS X development. This project includes algorithms focused on Bayes theorem, neural networks, SVMs, Matrices, etc..

- *Machine Learning*
- *Deep Learning*
 - *Understanding*
 - *Optimization / Training Techniques*
 - *Unsupervised / Generative Models*
 - *Image Segmentation / Object Detection*
 - *Image / Video*
 - *Natural Language Processing*
 - *Speech / Other*
 - *Reinforcement Learning*
 - *New papers*
 - *Classic Papers*

25.1 Machine Learning

Be the first to contribute!

25.2 Deep Learning

Forked from terryum's awesome deep learning papers.

25.2.1 Understanding

- Distilling the knowledge in a neural network (2015), G. Hinton et al. [\[pdf\]](#)
- Deep neural networks are easily fooled: High confidence predictions for unrecognizable images (2015), A. Nguyen et al. [\[pdf\]](#)
- How transferable are features in deep neural networks? (2014), J. Yosinski et al. [\[pdf\]](#)
- CNN features off-the-Shelf: An astounding baseline for recognition (2014), A. Razavian et al. [\[pdf\]](#)
- Learning and transferring mid-Level image representations using convolutional neural networks (2014), M. Oquab et al. [\[pdf\]](#)
- Visualizing and understanding convolutional networks (2014), M. Zeiler and R. Fergus [\[pdf\]](#)
- Decaf: A deep convolutional activation feature for generic visual recognition (2014), J. Donahue et al. [\[pdf\]](#)

25.2.2 Optimization / Training Techniques

- Batch normalization: Accelerating deep network training by reducing internal covariate shift (2015), S. Loffe and C. Szegedy [\[pdf\]](#)
- Delving deep into rectifiers: Surpassing human-level performance on imagenet classification (2015), K. He et al. [\[pdf\]](#)
- Dropout: A simple way to prevent neural networks from overfitting (2014), N. Srivastava et al. [\[pdf\]](#)
- Adam: A method for stochastic optimization (2014), D. Kingma and J. Ba [\[pdf\]](#)
- Improving neural networks by preventing co-adaptation of feature detectors (2012), G. Hinton et al. [\[pdf\]](#)
- Random search for hyper-parameter optimization (2012) J. Bergstra and Y. Bengio [\[pdf\]](#)

25.2.3 Unsupervised / Generative Models

- Pixel recurrent neural networks (2016), A. Oord et al. [\[pdf\]](#)
- Improved techniques for training GANs (2016), T. Salimans et al. [\[pdf\]](#)
- Unsupervised representation learning with deep convolutional generative adversarial networks (2015), A. Radford et al. [\[pdf\]](#)
- DRAW: A recurrent neural network for image generation (2015), K. Gregor et al. [\[pdf\]](#)
- Generative adversarial nets (2014), I. Goodfellow et al. [\[pdf\]](#)
- Auto-encoding variational Bayes (2013), D. Kingma and M. Welling [\[pdf\]](#)
- Building high-level features using large scale unsupervised learning (2013), Q. Le et al. [\[pdf\]](#)

25.2.4 Image Segmentation / Object Detection

- You only look once: Unified, real-time object detection (2016), J. Redmon et al. [\[pdf\]](#)
- Fully convolutional networks for semantic segmentation (2015), J. Long et al. [\[pdf\]](#)
- Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks (2015), S. Ren et al. [\[pdf\]](#)
- Fast R-CNN (2015), R. Girshick [\[pdf\]](#)
- Rich feature hierarchies for accurate object detection and semantic segmentation (2014), R. Girshick et al. [\[pdf\]](#)

- Semantic image segmentation with deep convolutional nets and fully connected CRFs, L. Chen et al. [\[pdf\]](#)
- Learning hierarchical features for scene labeling (2013), C. Farabet et al. [\[pdf\]](#)

25.2.5 Image / Video

- Image Super-Resolution Using Deep Convolutional Networks (2016), C. Dong et al. [\[pdf\]](#)
- A neural algorithm of artistic style (2015), L. Gatys et al. [\[pdf\]](#)
- Deep visual-semantic alignments for generating image descriptions (2015), A. Karpathy and L. Fei-Fei [\[pdf\]](#)
- Show, attend and tell: Neural image caption generation with visual attention (2015), K. Xu et al. [\[pdf\]](#)
- Show and tell: A neural image caption generator (2015), O. Vinyals et al. [\[pdf\]](#)
- Long-term recurrent convolutional networks for visual recognition and description (2015), J. Donahue et al. [\[pdf\]](#)
- VQA: Visual question answering (2015), S. Antol et al. [\[pdf\]](#)
- DeepFace: Closing the gap to human-level performance in face verification (2014), Y. Taigman et al. [\[pdf\]](#):
- Large-scale video classification with convolutional neural networks (2014), A. Karpathy et al. [\[pdf\]](#)
- DeepPose: Human pose estimation via deep neural networks (2014), A. Toshev and C. Szegedy [\[pdf\]](#)
- Two-stream convolutional networks for action recognition in videos (2014), K. Simonyan et al. [\[pdf\]](#)
- 3D convolutional neural networks for human action recognition (2013), S. Ji et al. [\[pdf\]](#)

25.2.6 Natural Language Processing

- Neural Architectures for Named Entity Recognition (2016), G. Lample et al. [\[pdf\]](#)
- Exploring the limits of language modeling (2016), R. Jozefowicz et al. [\[pdf\]](#)
- Teaching machines to read and comprehend (2015), K. Hermann et al. [\[pdf\]](#)
- Effective approaches to attention-based neural machine translation (2015), M. Luong et al. [\[pdf\]](#)
- Conditional random fields as recurrent neural networks (2015), S. Zheng and S. Jayasumana. [\[pdf\]](#)
- Memory networks (2014), J. Weston et al. [\[pdf\]](#)
- Neural turing machines (2014), A. Graves et al. [\[pdf\]](#)
- Neural machine translation by jointly learning to align and translate (2014), D. Bahdanau et al. [\[pdf\]](#)
- Sequence to sequence learning with neural networks (2014), I. Sutskever et al. [\[pdf\]](#)
- Learning phrase representations using RNN encoder-decoder for statistical machine translation (2014), K. Cho et al. [\[pdf\]](#)
- A convolutional neural network for modeling sentences (2014), N. Kalchbrenner et al. [\[pdf\]](#)
- Convolutional neural networks for sentence classification (2014), Y. Kim [\[pdf\]](#)
- Glove: Global vectors for word representation (2014), J. Pennington et al. [\[pdf\]](#)
- Distributed representations of sentences and documents (2014), Q. Le and T. Mikolov [\[pdf\]](#)
- Distributed representations of words and phrases and their compositionality (2013), T. Mikolov et al. [\[pdf\]](#)
- Efficient estimation of word representations in vector space (2013), T. Mikolov et al. [\[pdf\]](#)

- Recursive deep models for semantic compositionality over a sentiment treebank (2013), R. Socher et al. [\[pdf\]](#)
- Generating sequences with recurrent neural networks (2013), A. Graves. [\[pdf\]](#)

25.2.7 Speech / Other

- End-to-end attention-based large vocabulary speech recognition (2016), D. Bahdanau et al. [\[pdf\]](#)
- Deep speech 2: End-to-end speech recognition in English and Mandarin (2015), D. Amodei et al. [\[pdf\]](#)
- Speech recognition with deep recurrent neural networks (2013), A. Graves [\[pdf\]](#)
- Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups (2012), G. Hinton et al. [\[pdf\]](#)
- Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition (2012) G. Dahl et al. [\[pdf\]](#)
- Acoustic modeling using deep belief networks (2012), A. Mohamed et al. [\[pdf\]](#)

25.2.8 Reinforcement Learning

- End-to-end training of deep visuomotor policies (2016), S. Levine et al. [\[pdf\]](#)
- Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection (2016), S. Levine et al. [\[pdf\]](#)
- Asynchronous methods for deep reinforcement learning (2016), V. Mnih et al. [\[pdf\]](#)
- Deep Reinforcement Learning with Double Q-Learning (2016), H. Hasselt et al. [\[pdf\]](#)
- Mastering the game of Go with deep neural networks and tree search (2016), D. Silver et al. [\[pdf\]](#)
- Continuous control with deep reinforcement learning (2015), T. Lillicrap et al. [\[pdf\]](#)
- Human-level control through deep reinforcement learning (2015), V. Mnih et al. [\[pdf\]](#)
- Deep learning for detecting robotic grasps (2015), I. Lenz et al. [\[pdf\]](#)
- Playing atari with deep reinforcement learning (2013), V. Mnih et al. [\[pdf\]](#)

25.2.9 New papers

- Deep Photo Style Transfer (2017), F. Luan et al. [\[pdf\]](#)
- Evolution Strategies as a Scalable Alternative to Reinforcement Learning (2017), T. Salimans et al. [\[pdf\]](#)
- Deformable Convolutional Networks (2017), J. Dai et al. [\[pdf\]](#)
- Mask R-CNN (2017), K. He et al. [\[pdf\]](#)
- Learning to discover cross-domain relations with generative adversarial networks (2017), T. Kim et al. [\[pdf\]](#)
- Deep voice: Real-time neural text-to-speech (2017), S. Arik et al., [\[pdf\]](#)
- PixelNet: Representation of the pixels, by the pixels, and for the pixels (2017), A. Bansal et al. [\[pdf\]](#)
- Batch renormalization: Towards reducing minibatch dependence in batch-normalized models (2017), S. Ioffe. [\[pdf\]](#)
- Wasserstein GAN (2017), M. Arjovsky et al. [\[pdf\]](#)
- Understanding deep learning requires rethinking generalization (2017), C. Zhang et al. [\[pdf\]](#)

- Least squares generative adversarial networks (2016), X. Mao et al. [\[pdf\]](#)

25.2.10 Classic Papers

- An analysis of single-layer networks in unsupervised feature learning (2011), A. Coates et al. [\[pdf\]](#)
- Deep sparse rectifier neural networks (2011), X. Glorot et al. [\[pdf\]](#)
- Natural language processing (almost) from scratch (2011), R. Collobert et al. [\[pdf\]](#)
- Recurrent neural network based language model (2010), T. Mikolov et al. [\[pdf\]](#)
- Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion (2010), P. Vincent et al. [\[pdf\]](#)
- Learning mid-level features for recognition (2010), Y. Boureau [\[pdf\]](#)
- A practical guide to training restricted boltzmann machines (2010), G. Hinton [\[pdf\]](#)
- Understanding the difficulty of training deep feedforward neural networks (2010), X. Glorot and Y. Bengio [\[pdf\]](#)
- Why does unsupervised pre-training help deep learning (2010), D. Erhan et al. [\[pdf\]](#)
- Learning deep architectures for AI (2009), Y. Bengio. [\[pdf\]](#)
- Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations (2009), H. Lee et al. [\[pdf\]](#)
- Greedy layer-wise training of deep networks (2007), Y. Bengio et al. [\[pdf\]](#)
- A fast learning algorithm for deep belief nets (2006), G. Hinton et al. [\[pdf\]](#)
- Gradient-based learning applied to document recognition (1998), Y. LeCun et al. [\[pdf\]](#)
- Long short-term memory (1997), S. Hochreiter and J. Schmidhuber. [\[pdf\]](#)

CHAPTER 26

Other Content

Books, blogs, courses and more forked from josephmisiti's [awesome machine learning](#)

- *Blogs*
 - *Data Science*
 - *Machine learning*
 - *Math*
- *Books*
 - *Machine learning*
 - *Deep learning*
 - *Probability & Statistics*
 - *Linear Algebra*
- *Courses*
- *Podcasts*
- *Tutorials*

26.1 Blogs

26.1.1 Data Science

- <https://jeremykun.com/>
- <http://iamtrask.github.io/>
- <http://blog.explainmydata.com/>

- <http://andrewgelman.com/>
- <http://simplystatistics.org/>
- <http://www.evanmiller.org/>
- <http://jakevdp.github.io/>
- <http://blog.yhat.com/>
- <http://wesmckinney.com>
- <http://www.overkillanalytics.net/>
- <http://newton.cx/~peter/>
- http://mbakker7.github.io/exploratory_computing_with_python/
- <https://sebastianraschka.com/blog/index.html>
- <http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/>
- <http://colah.github.io/>
- <http://www.thomasdimson.com/>
- <http://blog.smellthedata.com/>
- <https://sebastianraschka.com/>
- <http://dogdogfish.com/>
- <http://www.johnmyleswhite.com/>
- <http://drewconway.com/zia/>
- <http://bugra.github.io/>
- <http://opendata.cern.ch/>
- <https://alexanderetz.com/>
- <http://www.sumsar.net/>
- <https://www.countbayesie.com>
- <http://blog.kaggle.com/>
- <http://www.danvk.org/>
- <http://hunch.net/>
- <http://www.randalolson.com/blog/>
- https://www.johndcook.com/blog/r_language_for_programmers/
- <http://www.dataschool.io/>

26.1.2 Machine learning

- OpenAI
- Distill
- Andrej Karpathy Blog
- Colah's Blog
- WildML

- [FastML](#)
- [TheMorningPaper](#)

26.1.3 Math

- <http://www.sumsar.net/>
- <http://allendowney.blogspot.ca/>
- <https://healthyalgorithms.com/>
- <https://petewarden.com/>
- <http://mrtz.org/blog/>

26.2 Books

26.2.1 Machine learning

- [Real World Machine Learning \[Free Chapters\]](#)
- [An Introduction To Statistical Learning - Book + R Code](#)
- [Elements of Statistical Learning - Book](#)
- [Probabilistic Programming & Bayesian Methods for Hackers - Book + IPython Notebooks](#)
- [Think Bayes - Book + Python Code](#)
- [Information Theory, Inference, and Learning Algorithms](#)
- [Gaussian Processes for Machine Learning](#)
- [Data Intensive Text Processing w/ MapReduce](#)
- [Reinforcement Learning: - An Introduction](#)
- [Mining Massive Datasets](#)
- [A First Encounter with Machine Learning](#)
- [Pattern Recognition and Machine Learning](#)
- [Machine Learning & Bayesian Reasoning](#)
- [Introduction to Machine Learning - Alex Smola and S.V.N. Vishwanathan](#)
- [A Probabilistic Theory of Pattern Recognition](#)
- [Introduction to Information Retrieval](#)
- [Forecasting: principles and practice](#)
- [Practical Artificial Intelligence Programming in Java](#)
- [Introduction to Machine Learning - Amnon Shashua](#)
- [Reinforcement Learning](#)
- [Machine Learning](#)
- [A Quest for AI](#)
- [Introduction to Applied Bayesian Statistics and Estimation for Social Scientists - Scott M. Lynch](#)

- Bayesian Modeling, Inference and Prediction
- A Course in Machine Learning
- Machine Learning, Neural and Statistical Classification
- Bayesian Reasoning and Machine Learning Book+MatlabToolBox
- R Programming for Data Science
- Data Mining - Practical Machine Learning Tools and Techniques Book

26.2.2 Deep learning

- Deep Learning - An MIT Press book
- Coursera Course Book on NLP
- NLTK
- NLP w/ Python
- Foundations of Statistical Natural Language Processing
- An Introduction to Information Retrieval
- A Brief Introduction to Neural Networks
- Neural Networks and Deep Learning

26.2.3 Probability & Statistics

- Think Stats - Book + Python Code
- From Algorithms to Z-Scores - Book
- The Art of R Programming
- Introduction to statistical thought
- Basic Probability Theory
- Introduction to probability - By Dartmouth College
- Principle of Uncertainty
- Probability & Statistics Cookbook
- Advanced Data Analysis From An Elementary Point of View
- Introduction to Probability - Book and course by MIT
- The Elements of Statistical Learning: Data Mining, Inference, and Prediction. -Book
- An Introduction to Statistical Learning with Applications in R - Book
- Learning Statistics Using R
- Introduction to Probability and Statistics Using R - Book
- Advanced R Programming - Book
- Practical Regression and Anova using R - Book
- R practicals - Book
- The R Inferno - Book

26.2.4 Linear Algebra

- Linear Algebra Done Wrong
- Linear Algebra, Theory, and Applications
- Convex Optimization
- Applied Numerical Computing
- Applied Numerical Linear Algebra

26.3 Courses

- CS231n, Convolutional Neural Networks for Visual Recognition, Stanford University
- CS224d, Deep Learning for Natural Language Processing, Stanford University
- Oxford Deep NLP 2017, Deep Learning for Natural Language Processing, University of Oxford
- Artificial Intelligence (Columbia University) - free
- Machine Learning (Columbia University) - free
- Machine Learning (Stanford University) - free
- Neural Networks for Machine Learning (University of Toronto) - free
- Machine Learning Specialization (University of Washington) - Courses: Machine Learning Foundations: A Case Study Approach, Machine Learning: Regression, Machine Learning: Classification, Machine Learning: Clustering & Retrieval, Machine Learning: Recommender Systems & Dimensionality Reduction, Machine Learning Capstone: An Intelligent Application with Deep Learning; free
- Machine Learning Course (2014-15 session) (by Nando de Freitas, University of Oxford) - Lecture slides and video recordings.
- Learning from Data (by Yaser S. Abu-Mostafa, Caltech) - Lecture videos available

26.4 Podcasts

- The O'Reilly Data Show
- Partially Derivative
- The Talking Machines
- The Data Skeptic
- Linear Digressions
- Data Stories
- Learning Machines 101
- Not So Standard Deviations
- TWIMLAI

- <http://ocdevel.com/mlg> -

26.5 Tutorials

Be the first to [contribute!](#)

CHAPTER 27

Contribute

Become a contributor! Check out our [github](#) for more information.