Machine Learning

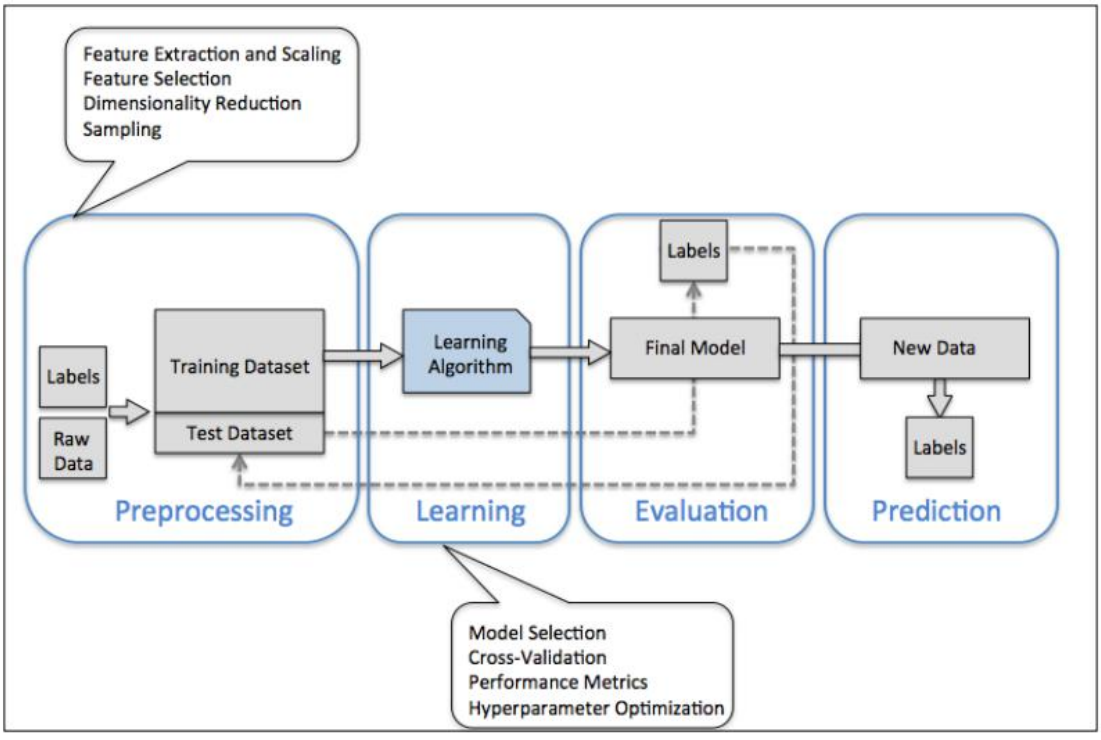
This project is to build machine learning models based on the concepts. Our goal is to go through machine learning, comprehend the concepts, and employ them to solve problems. That is to learn mathematics behind these algorithms. We will understand the basic concepts as well as implement the algorithms in ***Python/Pycharm*** which is an Integrated Development Environment. The following will demonstrate how each algorithm works.

There are three types of machine learning
Unsupervised learning
Supervised learning
Reinforcement Learning

Each type of machine learning can solve different problems. Based on this logic, we categorize the learning algorithms into sub-learning algorithms

Unsupervised Learning	***Supervised Learning***	***Reinforcement Learning***
Classification	Clustering	Value Iteration
Regression	Dimensionality Reduction	Policy Iteration

Systematic Machine Learning Roadmap



In order to visualize the matrix and sometimes, we will denote feature vector and weights as follows.

$$x = \begin{bmatrix} x_1 & \cdots & x_d \end{bmatrix}$$

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

Most of the algorithms are to find the optimal weights that best fit the data. Hence, the general algorithm:

1. Initialize the weights;
2. Compute the predicted value
3. Check stop rules
4. If satisfied, output
5. Update the weights
6. Go back to step 2

The Perceptron

How it works

Prediction

$$z = xw$$

$$\hat{y} = \phi(z) = \begin{cases} 1 \; if \;\; z > 0 \\ -1 \; otherwise \end{cases}$$

Update

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

$$w_j = w_j + \Delta w_j$$

$$where \; \eta = learning \;\; rate$$

The biggest disadvantage is that it never converges if the classes are not perfectly linearly separable.

Implementation

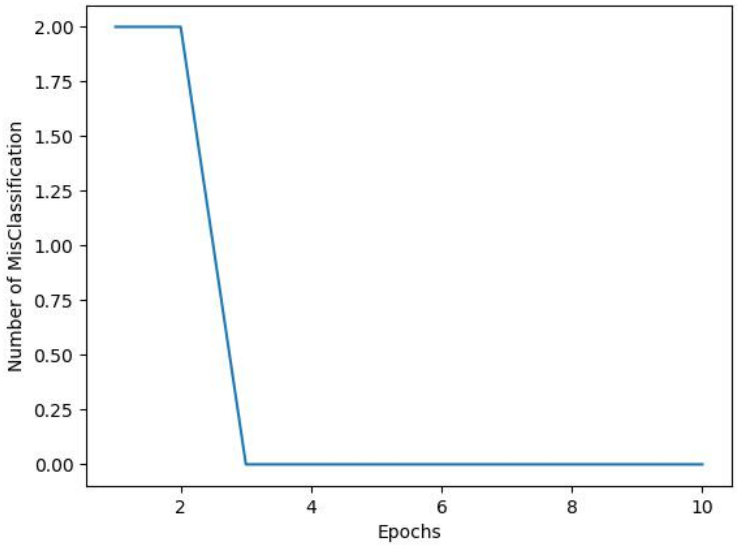
We are going to implement the algorithm using python. This is an example to show how to write a perceptron.

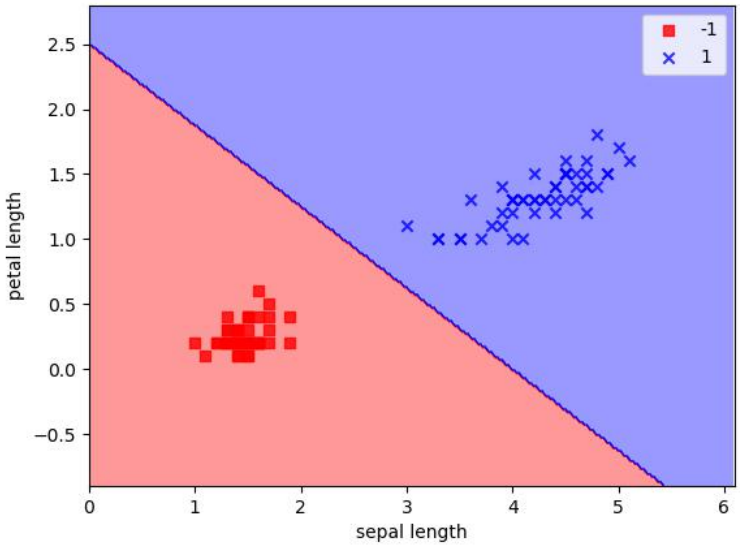
```
class Perceptron(object):

    def __init__(self, learning_rate=0.01, max_iter=None):
        self.learning_rate = learning_rate
        self.max_iter = max_iter

    def fit(self, X, y):
        self.weight = np.zeros(X.shape[1] + 1)
        self.errors = []

        for _ in range(self.max_iter):
            errors = 0

            for xi, target in zip(X, y):
```

```
            update = self.learning_rate * (target - self.predict(xi))
            self.weight[1:] += update * xi
            self.weight[0] += update

            errors += int(update != 0.0)
        self.errors.append(errors)
    return self

def net_input(self, X):
    return np.dot(X, self.weight[1:]) + self.weight[0]

def predict(self, X):
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

We test our model in main function. We plot the errors with epochs increasing and decision regions.

```
ppn = Perceptron(learning_rate=0.01, max_iter=10)
ppn.fit(x, y)
plt.plot(range(1, len(ppn.errors) + 1), ppn.errors)
plt.xlabel('Epochs')
plt.ylabel('Number of MisClassification')
plt.show()

plot_decision_regions(x, y, classifer=ppn)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.legend()
plt.show()
```





Adaptive Linear Neurons

Most of algorithms are optimization problems. We can use gradient descent to solve the problems.

How it works

Prediction

$$\phi(z) = z = \hat{y}$$

$$J(w) = \frac{1}{2}\sum_i \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Update

$$\Delta w = \eta \nabla J(w)$$

$$w = w + \Delta w$$

Derivation

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2}\sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

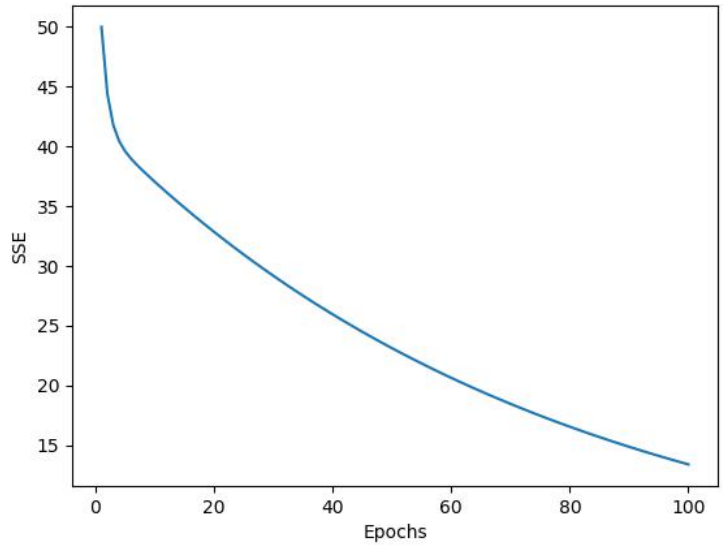
Implementation

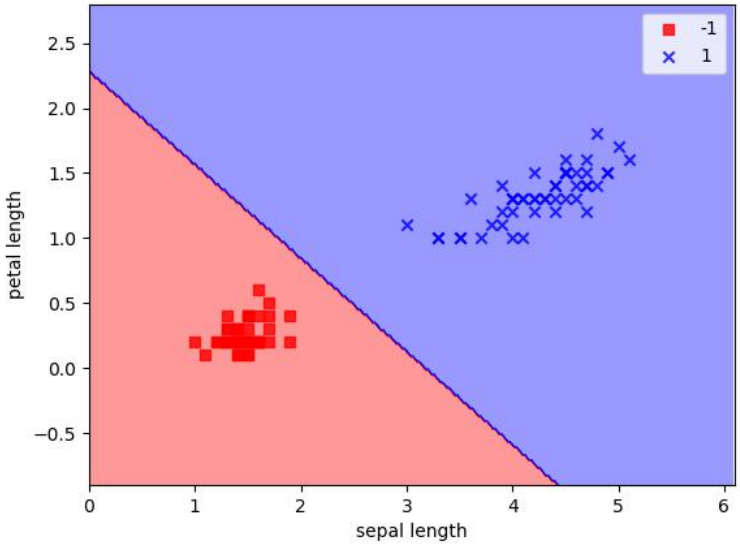
```
class AdalineGD(object):


    def __init__(self, learning_rate=0.01, max_iter=10):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
```

```
    def fit(self, X, y):
        self.weight = np.zeros(X.shape[1] + 1)
        self.cost_ = []

        for i in range(self.max_iter):
            output = self.activation(X)
            errors = y - output

            self.weight[1:] = self.learning_rate * np.dot(X.T, errors)
            self.weight[0] += self.learning_rate * errors.sum()

            cost = (errors ** 2).sum() / 2.0
            self.cost_.append(cost)

        return self

    def net_input(self, X):
        return np.dot(X, self.weight[1:]) + self.weight[0]

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Test it in the main function

```
ada = AdalineGD(learning_rate=0.0003, max_iter=100)
ada.fit(x, y)
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_)
plt.xlabel('Epochs')
plt.ylabel('SSE')
plt.show()
plot_decision_regions(x, y, classifer=ada)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.legend()
plt.show()
```





Stochastic Gradient

Update the weights incrementally for each training sample.

How it works

$$\frac{\partial J}{\partial w_j} = \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

$$w_j = w_j + \eta \frac{\partial J}{\partial w_j}$$

Advantage:

Reach convergence much faster

Escape shallow local minima more readily.

The learning rate can be an adaptive learning rate.

$$\frac{c_1}{[number\ of\ iterations]+c_2}$$

Online learning

We can modify the code to implement SGD.

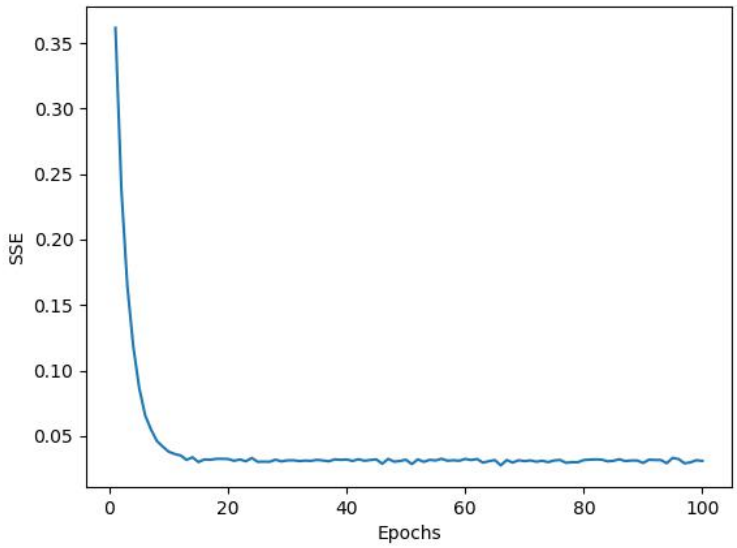
Implementation

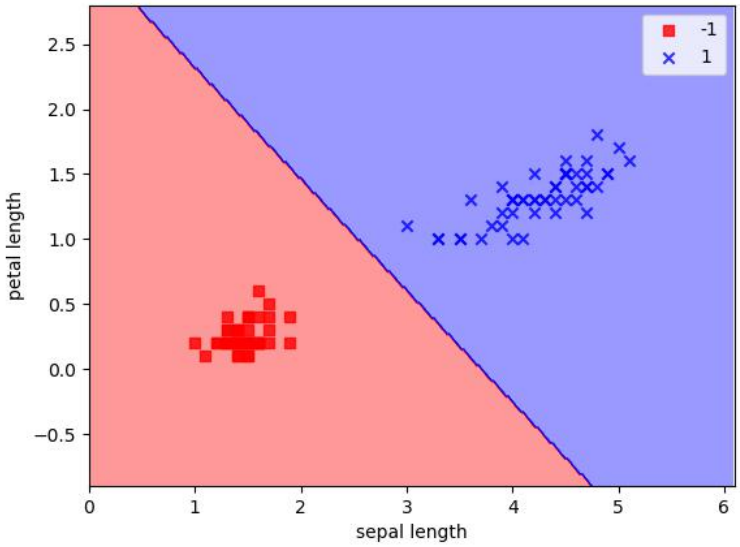
```
class AdalineSGD(object):

    def __init__(self, learning_rate=0.01, max_iter=10,
                 shuffle=True, random_state=None):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.w_initialized = False
        self.shuffle = shuffle

        if random_state:
            seed(random_state)

    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        self.cost_ = []

        for i in range(self.max_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []

            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)

        return self

    def _shuffle(self, X, y):
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, n):
        self.weight = np.zeros(n + 1)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        output = self.net_input(xi)
```

```
        error = target - output
        self.weight[1:] += self.learning_rate * np.dot(xi, error)
        self.weight[0] += self.learning_rate * error
        cost = (error ** 2) / 2.0

        return cost

    def net_input(self, X):
        return np.dot(X, self.weight[1:]) + self.weight[0]

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Test it in the main function

```
ada = AdalineSGD(learning_rate=0.01, max_iter=100, random_state=1)
ada.fit(x, y)
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_)
plt.xlabel('Epochs')
plt.ylabel('SSE')
plt.show()

plot_decision_regions(x, y, classifer=ada)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.legend()
plt.show()
```





Classification

The five main steps in training machine learning algorithm

1. Selection of features
2. Choosing a performance metric
3. Choosing a classifier and optimization algorithm
4. Evaluating the performance of the model
5. Tuning the algorithm

Logistic Regression

Logit function

$$\log it(p) = \log \frac{p}{1-p}$$

Sigmoid function

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

How it works

prediction

$$L(w) = p(y \mid x; w) = \prod_{i=1}^{n} p\left(y^{(i)} \mid x^{(i)}; w\right)$$

$$= \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

Update

$$\Delta w_j = -\eta \frac{\partial}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j$$

$$w = w + \Delta w = w - \eta \nabla J(w)$$

derivation

In practice, we try to maximize the log of cost function

$$l(w) = \max \log L(w) = \max \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

Find the gradient of log-cost function

$$\frac{\partial}{\partial w_j} l(w) = \sum_{i=1}^{n} \left(y^{(i)} \frac{1}{\phi\left(z^{(i)}\right)} - \left(1 - y^{(i)}\right) \left(\frac{1}{1 - \phi\left(z^{(i)}\right)} \right) \right) \frac{\partial}{\partial w_j} \phi\left(z^{(i)}\right)$$

$$\frac{\partial}{\partial z^{(i)}} \phi\left(z^{(i)}\right) = \phi\left(z^{(i)}\right)\left(1 - \phi\left(z^{(i)}\right)\right)$$

$$\frac{\partial z^{(i)}}{\partial w_j} = x_j^{(i)}$$

$$\frac{\partial}{\partial w_j} l(w) = \sum_{i=1}^{n} \left(y^{(i)} \frac{1}{\phi\left(z^{(i)}\right)} - \left(1 - y^{(i)}\right) \left(\frac{1}{1 - \phi\left(z^{(i)}\right)} \right) \right) \phi\left(z^{(i)}\right)\left(1 - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

$$= \sum_{i=1}^{n} \left(y^{(i)} \left(1 - \phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

$$= \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

$$\frac{\partial}{\partial w_j} l(w) = \sum_{i=1}^{n} \left(y^{(i)} \frac{1}{\phi\left(z^{(i)}\right)} - \left(1 - y^{(i)}\right) \left(\frac{1}{1 - \phi\left(z^{(i)}\right)} \right) \right) \phi\left(z^{(i)}\right)\left(1 - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

$$= \sum_{i=1}^{n} \left(y^{(i)} \left(1 - \phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

$$= \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Regularization L2

$$\frac{\lambda}{2} \|w\|^2 = \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2$$

The cost function with regularization

$$J(\phi(z), y; w) = C \left[\sum_{i=1}^{n} \left(- y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right) \right] + \frac{1}{2} \|w\|^2$$

$$C = \frac{1}{\lambda}$$

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)} + w_j$$

Maximum margin Classification SVM

$$w_0 + w^T x_{pos} = 1$$

$$w_0 + w^T x_{neg} = -1$$

$$w^T \left(x_{pos} - x_{neg} \right) = 2$$

Normalization

$$\frac{w^T \left(x_{pos} - x_{neg} \right)}{\|w\|} = \frac{2}{\|w\|}$$

Constraints

$$\begin{cases} w_0 + w^T x^{(i)} \geq 1 \quad if \quad y^{(i)} = 1 \\ w_0 + w^T x^{(i)} \leq -1 \; if \quad y^{(i)} = -1 \end{cases}$$

Hence,

$$y^{(i)} \left(w_0 + w^T x^{(i)} \right) \geq 1$$

$$L(w) = \min \frac{1}{2} \|w\|^2$$

To solve this optimization problem, we transform it into a dual problem. Here, we consider Lagrange Multiplier.

$$L(w, w_0, a) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^{n} a_i \left[y^{(i)} \left(w_0 + w^T x^{(i)} \right) - 1 \right]$$

Find the derivatives with respect to the parameters.

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} a_i y^{(i)} x^{(i)} = 0$$

$$w = \sum_{i=1}^{n} a_i y^{(i)} x^{(i)}$$

$$\frac{\partial L}{\partial w_0} = \sum_{i=1}^{n} a_i y^{(i)} = 0$$

$$L(a) = \frac{1}{2} \left(\sum_{i=1}^{n} a_i y^{(i)} x^{(i)} \right)^2 - \sum_{j=1}^{n} a_j y^{(j)} \left(\sum_{i=1}^{n} a_i y^{(i)} x^{(i)} \right)^T x^{(j)} + \sum_{i=1}^{n} a_i$$

$$= \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y^{(i)} y^{(j)} \left(x^{(i)} \right)^T x^{(j)}$$

Hence, the problem can be

$$L(a) = \max \left(\sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y^{(i)} y^{(j)} \left(x^{(i)} \right)^T x^{(j)} \right)$$

$$s.t. \sum_{i=1}^{n} a_i y^{(i)} = 0, a_i > 0$$

If we use kernel function, then the problem can be

$$L(a) = \max\left(\sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y^{(i)} y^{(j)} \left(\phi\left(x^{(i)}\right)\right)^T \phi\left(x^{(j)}\right) \right)$$

$$s.t. \sum_{i=1}^{n} a_i y^{(i)} = 0, a_i > 0$$

Kernel Function

As I can see, kernel function is a mapping from original feature space to high dimensional feature space.

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = \left(x_1, x_2, x_1^2 + x_2^2\right)$$

$$k\left(x^{(i)}, x^{(j)}\right) = \phi\left(x^{(i)}\right)^T \phi\left(x^{(j)}\right)$$

Radial Basis Function kernel

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\frac{\left\| x^{(i)} - x^{(j)} \right\|^2}{2\sigma^2} \right)$$

Which can be simplified to

$$k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\gamma \left\| x^{(i)} - x^{(j)} \right\|^2 \right)$$

Similarity function

Decision Tree

Based on the features in our training set, the decision tree model learns a series of questions to infer the class labels of the samples.

Information Gain

Prune the tree and limit the maximal depth of the tree

$$IG\left(D_p, f\right) = I\left(D_p\right) - \sum_{j=1}^{m} \frac{N_j}{N_p} I\left(D_j\right)$$

$$f = feature\ to\ perform\ the\ split$$

Binary Search tree

$$IG\left(D_p, f\right) = I\left(D_p\right) - \frac{N_{left}}{N_p} I\left(D_{left}\right) - \frac{N_{right}}{N_p} I\left(D_{right}\right)$$

Entropy

$$I_H(t) = -\sum_{i=1}^{c} p(i \mid t) \log_2 p(i \mid t)$$

The Gini

$$I_G(t) = \sum_{i=1}^{c} p(i \mid t)\left(1 - p(i \mid t)\right) = 1 - \sum_{i=1}^{c} p(i \mid t)^2$$

Random forest

1. Draw a random bootstrap sample of size n (randomly choose n samples from the training set with replacement).
2. Grow a decision tree from the bootstrap sample. At each node.
    1. Randomly select d features without replacement.
    2. Split the node using the feature that provides the best split according to the objective function, for instance, by maximizing the information gain.
3. Repeat the steps 1 to 2 k times.
4. Aggregate the prediction by each tree to assign the class label by majority vote.

K-nearest neighbors

$$d\left(x^{(i)}, x^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

1. Choose the number of k and a distance metric.
2. Find the k nearest neighbors of the sample that we want to classify.
3. Assign the class label by majority vote.

These predictive models are employed to solve the Classification problems. In order to select the best model each time, we write our computational model as a system. Our framework includes Lib.py, Data_preprocessing.py, Classifier.py, Optimization.py, Plot.py, main.py.

<Lib>

```
import pandas as pd
from pandas import DataFrame, Series
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from matplotlib.colors import ListedColormap
from Data_preprocessing import load_split_standardize_data
from Plot import plot
from Classifier import optimize_classifier, train
from Optimization import optimization_and_recommendation
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import Perceptron
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
```

<Data_preprocessing>

```
def load_split_standardize_data(filename=None, test_size=0.3, random_state=None):

    if filename != None:
        # load data
        df = pd.read_csv(filename)
        x, y = df.iloc[:, 1:].values, df.iloc[:, 0].values
    else:
        iris = datasets.load_iris()
        x, y = iris.data[:, [2, 3]], iris.target

    # split data
    x_train, x_test, y_train, y_test = train_test_split(
        x, y, test_size=test_size, random_state=random_state)

    # standardize data
    sc = StandardScaler()
    x_train_std = sc.fit_transform(x_train)
    x_test_std = sc.transform(x_test)

    return x_train_std, x_test_std, y_train, y_test
```

<Classifier>

```
def train(x, y, clf):
    clf.fit(x, y)


def calculate_accuracy(x_train, x_test, y_train, y_test, clf):

    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    return accuracy_score(y_test, y_pred)


def optimize_classifier(option, x_train, x_test, y_train, y_test):

    accuracy = []

    if option == 'Perceptron':
        eta = [0.00001, 0.0001, 0.001, 0.01, 0.1,  0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
0.9]
        for eta0 in eta:
            ppn = Perceptron(n_iter=40, eta0=eta0, random_state=0)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, ppn)
```

```
            accuracy.append(acc)
        return eta, accuracy

    elif option == 'Logistic_Regression':
        C = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
        for c in C:
            lr = LogisticRegression(C=c, random_state=0)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, lr)
            accuracy.append(acc)
        return C, accuracy

    elif option == 'SVM_Kernel_Linear':
        C = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
        for c in C:
            svm = SVC(kernel='linear', C=c, random_state=0)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, svm)
            accuracy.append(acc)
        return C, accuracy

    elif option == 'SVM_Kernel_rbf':
        C = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
        for c in C:
            svm = SVC(kernel='rbf', C=c, random_state=0)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, svm)
            accuracy.append(acc)
        return C, accuracy

    elif option == 'Decision Tree':
        depth = np.arange(1, 15)
        for max_depth in depth:
            tree = DecisionTreeClassifier(criterion='entropy', max_depth=max_depth,
random_state=0)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, tree)
            accuracy.append(acc)
        return depth, accuracy

    elif option == 'Random Forest':
        n_estimator = np.arange(1, 20)
        for n in n_estimator:
            forest = RandomForestClassifier(criterion='entropy', n_estimators=n,
random_state=1, n_jobs=2)
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, forest)
            accuracy.append(acc)
        return n_estimator, accuracy
```

```
    elif option == 'K-Neighbors':
        K = np.arange(1, 20)
        for k in K:
            knn = KNeighborsClassifier(n_neighbors=k, p=2, metric='minkowski')
            acc = calculate_accuracy(x_train, x_test, y_train, y_test, knn)
            accuracy.append(acc)
        return K, accuracy
```

<Optimization>

```
from Lib import *


def optimization_and_recommendation(mode=None, filename=None):

    # load data
    x_train, x_test, y_train, y_test = load_split_standardize_data(
        filename=filename, test_size=0.3, random_state=0)

    if mode == 'Recommendation':
        x_combined = np.vstack((x_train, x_test))
        y_combined = np.hstack((y_train, y_test))

    # model name
    options = ['Perceptron', 'Logistic_Regression', 'SVM_Kernel_Linear',
               'SVM_Kernel_rbf', 'Decision Tree', 'Random Forest', 'K-Neighbors']

    # store the best parameters for each predictive model
    model_param_n_accuracy = {}

    # visualize the relationship between parameter and accuracy for each model
    if mode == 'Optimization':
        fig = plt.figure()
        cols = 4
        rows = np.ceil(len(options) / cols)
        ith_fig = 0

        print('Optimizing...')

    for option in options:
        params, acc = optimize_classifier(option, x_train, x_test, y_train, y_test)

        if mode == 'Optimization':
            ith_fig += 1
```

```
        ax = fig.add_subplot(rows, cols, ith_fig)
        ax.plot(params, acc, marker='o')
        plt.title(option)
        plt.ylabel('Accuracy')

    model_param_n_accuracy[option] = [params[acc.index(max(acc))], max(acc)]

print('Finished!')

if mode == 'Optimization':
    plt.show()

if mode == 'Recommendation':

    ppn = Perceptron(n_iter=40,
                     eta0=model_param_n_accuracy['Perceptron'][0],
                     random_state=0)
    lr = LogisticRegression(C=model_param_n_accuracy['Logistic_Regression'][0],
                            random_state=0)
    svm_linear = SVC(kernel='linear',
                     C=model_param_n_accuracy['SVM_Kernel_Linear'][0],
                     random_state=0)
    svm_rbf = SVC(kernel='rbf',
                  C=model_param_n_accuracy['SVM_Kernel_rbf'][0],
                  random_state=0)
    tree = DecisionTreeClassifier(criterion='entropy',
                                  max_depth=model_param_n_accuracy['Decision Tree'][0],
                                  random_state=0)
    forest = RandomForestClassifier(criterion='entropy',
                                    n_estimators=model_param_n_accuracy['Random Forest'][0],
                                    random_state=1,
                                    n_jobs=2)
    knn = KNeighborsClassifier(n_neighbors=model_param_n_accuracy['K-Neighbors'][0],
                               p=2,
                               metric='minkowski')

    clfs = [ppn, lr, svm_linear, svm_rbf, tree, forest, knn]

    fig = plt.figure()
    cols = 4
    rows = np.ceil(len(options) / cols)
    ith_fig = 0

    for option, clf in zip(options, clfs):
```

```
            train(x_train, y_train, clf)

            # visualize the results
            plot_labels = {'x': 'petal length',
                           'y': 'petal width'}
            plot_title = option

            ith_fig += 1
            _ = fig.add_subplot(rows, cols, ith_fig)
            plot(x_combined, y_combined, classifer=clf,
                test_idx=range(105, 150), label=plot_labels, title=plot_title)

        plt.show()

    df = DataFrame(model_param_n_accuracy, index=['Parameter', 'Accuracy'])
    print(df.T)
```

<Plot>

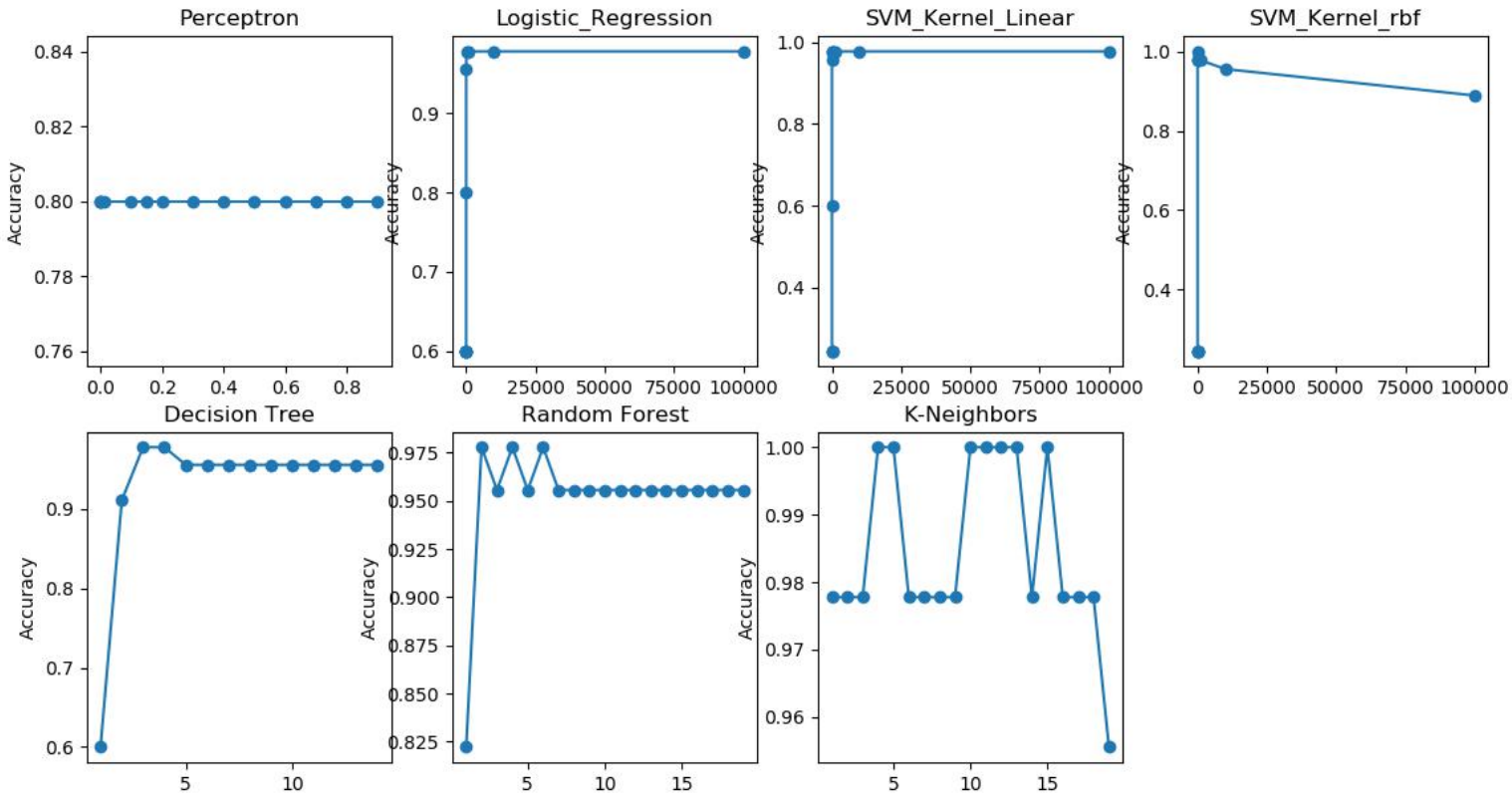
```
def plot_decision_regions(X, y, classifer, resolution=0.02, test_idx=None):

    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')

    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))

    zz = classifer.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    zz = zz.reshape(xx1.shape)

    plt.contourf(xx1, xx2, zz, alpha=0.4, cmap=cmap)
    plt.xlim(x1_min, x1_max)
    plt.ylim(x2_min, x2_max)

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(X[y == cl, 0], X[y == cl, 1],
                    alpha=0.8, color=cmap(idx),
                    marker=markers[idx], label=cl)

    if test_idx:
        x_test, y_test = X[test_idx, :], y[test_idx]
```

```
        plt.scatter(x_test[:, 0], x_test[:, 1], c='gray',
                    alpha=0.2, linewidths=1, marker='^',
                    s=55, label='test set')


def plot(x, y, classifer, test_idx, label, title):
    plot_decision_regions(x, y, classifer, resolution=0.02, test_idx=test_idx)
    plt.xlabel(label['x'])
    plt.ylabel(label['y'])
    plt.legend()
    plt.title(title)
```

<main>

```
def run(mode=None, filename=None):
    optimization_and_recommendation(mode, filename)


def app(mode=None, Name=None):
    print('This app is to demonstrate how each predictive model works.')
    print('We use Sklearn to train model and use accuracy as an assessment.')
    print('There are some predictive models.')
    print('Perceptron, Logistic Regression, SVM_Kernel_Linear, '
          'SVM_Kernel_rbf, Decision Tree, Random Forest, K-Neighbors')
    print('It will output the best model.')
    run(mode, Name)


if __name__ == '__main__':

    # Optimization / Recommendation
    option = 'Optimization'
    data_name = None
    app(option, data_name)
```

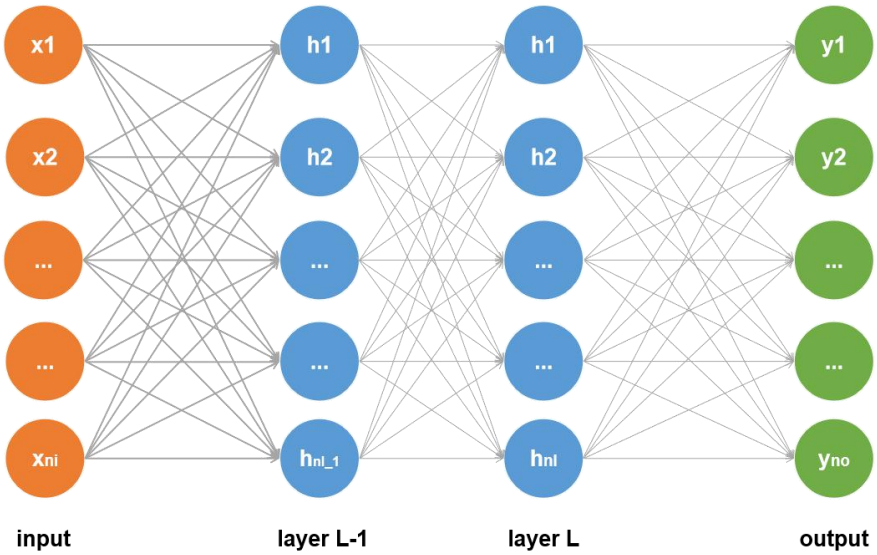
	Parameter	Accuracy
Perceptron	0.00001	0.800000
Logistic_Regression	100.00000	0.977778
SVM_Kernel_Linear	1.00000	0.977778
SVM_Kernel_rbf	100.00000	1.000000
Decision Tree	3.00000	0.977778
Random Forest	2.00000	0.977778
K-Neighbors	4.00000	1.000000


Perceptron
Logistic_Regression
SVM_Kernel_Linear
SVM_Kernel_rbf
Decision Tree
Random Forest
K-Neighbors
Accuracy

Classifier	***Mathematics***	***(Dis-)Advantages***
Perceptron	$z = xw$ $\hat{y} = \phi(z) = \begin{cases} 1 \; if \;\; z > 0 \\ -1 \; otherwise \end{cases}$	Easy to implement; Never converges if not linearly separable.
Logistic Regression	$l(w) = \max \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$ $\phi(z) = \frac{1}{1 + e^{-z}}$	Prone to outliers Easily implement and update
SVM_linear	$L(w) = \min \frac{1}{2} \|w\|^2$	Care about decision boundary
SVM_rbf	$L(w) = \min \frac{1}{2} \|w\|^2$ $k\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\gamma \left\| x^{(i)} - x^{(j)} \right\|^2 \right)$	Handle nonlinearly separable
Decision Tree	$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$ $I_H(t) = -\sum_{i=1}^{c} p(i \mid t) \log_2 p(i \mid t)$ $I_G(t) = \sum_{i=1}^{c} p(i \mid t)(1 - p(i \mid t)) = 1 - \sum_{i=1}^{c} p(i \mid t)^2$	Easily result in overfitting
Random Forest		Combine weak learners to build a robust model Don't care about hyperparameters # of feature $d = \sqrt{m}$
K-Nearest Neighbors	$d\left(x^{(i)}, x^{(j)}\right) = \sqrt[p]{\sum_{k} \left\| x_k^{(i)} - x_k^{(j)} \right\|^p}$	Curse of dimensionality

Neural Network		
Simple NN		
AutoEncoder		
Convolutional NN		
Boltzmann Machine		
Generative Adversial NN		
Recurrent Neural Network		

Neural Network is to try to solve the problems by approximation using weights networks. This approach is quite useful. It almost approximate every function by different layered-networks and activation function. All the deep neural network are based on the simple neural network. let's take a close look at neural network. We use a picture to show the neural network.



Forward Neural Network

input

$$\begin{bmatrix} x_1 & x_2 & \cdots & \cdots & x_d \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,ni} \\ \vdots & \ddots & \vdots \\ w_{d1} & \cdots & w_{d,ni} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{ni} \end{bmatrix} = \begin{bmatrix} a_1^1 & \cdots & a_{ni}^1 \end{bmatrix}$$

$$\begin{bmatrix} h_1^1 & \cdots & h_{ni}^1 \end{bmatrix} = g\left(\begin{bmatrix} a_1^1 & \cdots & a_{ni}^1 \end{bmatrix}\right)$$

hidden

$$\begin{bmatrix} h_1^{L-1} & \cdots & h_{n_{L-1}}^{L-1} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,n_L} \\ \vdots & \ddots & \vdots \\ w_{n_{L-1},1} & \cdots & w_{n_{L-1},n_L} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{n_L} \end{bmatrix} = \begin{bmatrix} a_1^L & \cdots & a_{n_L}^L \end{bmatrix}$$

$$\begin{bmatrix} h_1^L & \cdots & h_{n_L}^L \end{bmatrix} = g\left(\begin{bmatrix} a_1^L & \cdots & a_{n_L}^L \end{bmatrix}\right)$$

output

$$\begin{bmatrix} h_1^L & \cdots & h_{n_L}^L \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,n_o} \\ \vdots & \ddots & \vdots \\ w_{n_L,1} & \cdots & w_{n_L,n_o} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{no} \end{bmatrix} = \begin{bmatrix} a_1^o & \cdots & a_{n_o}^o \end{bmatrix}$$

$$\begin{bmatrix} y_1^o & \cdots & y_{n_o}^o \end{bmatrix} = g\left(\begin{bmatrix} a_1^o & \cdots & a_{n_o}^o \end{bmatrix}\right)$$

We can define our cost function based on the computational model accordingly. Here, we use cross-entropy as cost function.

$$L = y * \log\left(soft\max\left(y^o\right)\right)$$

Backward Propagation to update weights

output layer

$$\frac{\partial L}{\partial y^o} = \begin{pmatrix} \frac{\partial L}{\partial y_1^0} & \cdots & \frac{\partial L}{\partial y_{n_0}^0} \end{pmatrix}$$

$$\frac{\partial y^o}{\partial a^o} = g_o^{'}\left(a^o\right)$$

$$\frac{\partial a^o}{\partial w^o} = \begin{pmatrix} \left(h^L\right)_1^T & \cdots & \left(h^L\right)_{n_o}^T \end{pmatrix}, \frac{\partial a^o}{\partial b^o} = \begin{pmatrix} 1_1 & \cdots & 1_{n_o} \end{pmatrix}$$

$$\frac{\partial a^o}{\partial h^L} = w^o$$

hidden layer

$$\frac{\partial h^L}{\partial a^L} = g_L^{'}\left(a^L\right)$$

$$\frac{\partial a^L}{\partial w^L} = \begin{pmatrix} \left(h^{L-1}\right)_1^T & \cdots & \left(h^{L-1}\right)_{n_L}^T \end{pmatrix}, \frac{\partial a^L}{\partial b^L} = \begin{pmatrix} 1_1 & \cdots & 1_{n_L} \end{pmatrix}$$

$$\frac{\partial a^L}{\partial h^{L-1}} = w^L$$

We use a two-hidden layer neural network to demonstrate how it works.

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
import tensorflow as tf



# Parameters
learning_rate = 0.01
num_steps = 5000
batch_size = 128
display_step = 100
```

```
# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 64 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}

biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}


# Create model
def neural_net(x):

    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']

    return out_layer


# Construct model
logits = neural_net(X)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model
```

```
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})

            print("Step " + str(step) + ", Minibatch Loss= " +
                  "{:.4f}".format(loss) + ", Training Accuracy= " +
                  "{:.3f}".format(acc))

    print("Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:",
          sess.run(accuracy, feed_dict={X: mnist.test.images,
                                        Y: mnist.test.labels}))
```

Step 4800, Minibatch Loss= 1.8571, Training Accuracy= 0.891

Step 4900, Minibatch Loss= 1.0696, Training Accuracy= 0.914

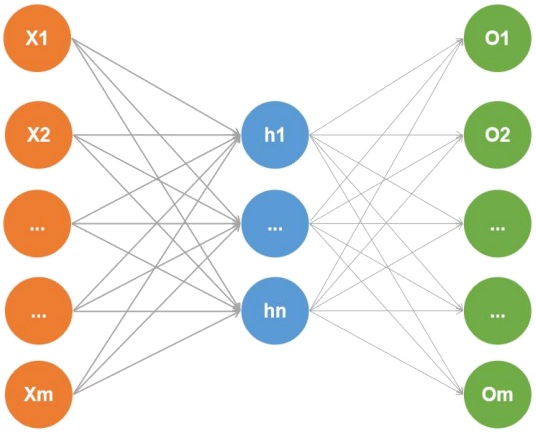
Step 5000, Minibatch Loss= 2.1295, Training Accuracy= 0.773

Finished!

Testing Accuracy: 0.8614

AutoEncoder

AutoEncoder is neural network that tries to reconstruct inputs. The number of AutoEncoder is quite small which means we use less nodes to capture as more features as possible.




```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Training Parameters
learning_rate = 0.04
num_steps = 30000
batch_size = 256
display_step = 1000
examples_to_show = 10

# Network Parameters
# 1st layer num features
num_hidden_1 = 256
# 2nd layer num features (the latent dim)
num_hidden_2 = 128
# MNIST data input (img shape: 28*28)
num_input = 784

# tf Graph input (only pictures)
X = tf.placeholder("float", [None, num_input])

weights = {
    'encoder_h1': tf.Variable(tf.random_normal([num_input, num_hidden_1])),
    'encoder_h2': tf.Variable(tf.random_normal([num_hidden_1, num_hidden_2])),
    'decoder_h1': tf.Variable(tf.random_normal([num_hidden_2, num_hidden_1])),
    'decoder_h2': tf.Variable(tf.random_normal([num_hidden_1, num_input])),
}
```

```
biases = {
    'encoder_b1': tf.Variable(tf.random_normal([num_hidden_1])),
    'encoder_b2': tf.Variable(tf.random_normal([num_hidden_2])),
    'decoder_b1': tf.Variable(tf.random_normal([num_hidden_1])),
    'decoder_b2': tf.Variable(tf.random_normal([num_input])),
}


# Building the encoder
def encoder(x):
    # Encoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']),
                                   biases['encoder_b1']))
    # Encoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']),
                                   biases['encoder_b2']))
    return layer_2


# Building the decoder
def decoder(x):
    # Decoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_h1']),
                                   biases['decoder_b1']))
    # Decoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']),
                                   biases['decoder_b2']))

    return layer_2


# Construct model
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)

# Prediction
y_pred = decoder_op
# Targets (Labels) are the input data.
y_true = X

# Define loss and optimizer, minimize the squared error
loss = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)
```

```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start Training
# Start a new TF session
with tf.Session() as sess:
    # Run the initializer
    sess.run(init)
    # Training
    for i in range(1, num_steps+1):
        # Prepare Data
        # Get the next batch of MNIST data (only images are needed, not labels)
        batch_x, _ = mnist.train.next_batch(batch_size)

        # Run optimization op (backprop) and cost op (to get loss value)
        _, L = sess.run([optimizer, loss], feed_dict={X: batch_x})

        # Display logs per step
        if i % display_step == 0 or i == 1:
            print('Step %i: Minibatch Loss: %f' % (i, L))

    # Testing
    # Encode and decode images from test set and visualize their reconstruction.
    n = 4
    canvas_orig = np.empty((28 * n, 28 * n))
    canvas_recon = np.empty((28 * n, 28 * n))

    for i in range(n):
        # MNIST test set
        batch_x, _ = mnist.test.next_batch(n)
        # Encode and decode the digit image
        g = sess.run(decoder_op, feed_dict={X: batch_x})

        # Display original images
        for j in range(n):
            # Draw the original digits
            canvas_orig[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] = batch_x[j].reshape([28,
28])

        # Display reconstructed images
        for j in range(n):
            # Draw the reconstructed digits
            canvas_recon[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] = g[j].reshape([28, 28])
```

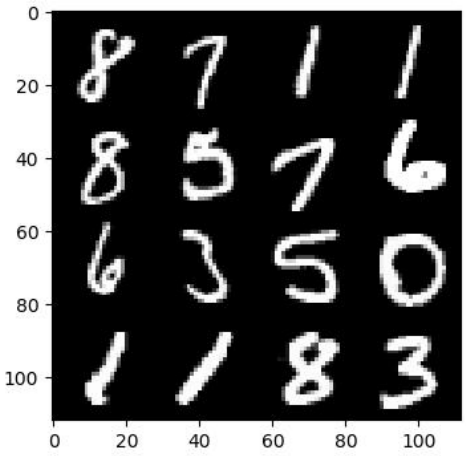
```
print("Original Images")
plt.figure(figsize=(n, n))
plt.imshow(canvas_orig, origin="upper", cmap="gray")
# plt.show()

print("Reconstructed Images")
plt.figure(figsize=(n, n))
plt.imshow(canvas_recon, origin="upper", cmap="gray")
plt.show()
```

Step 28000: Minibatch Loss: 0.020715

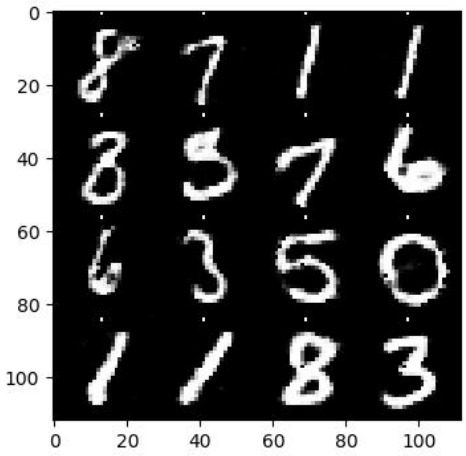
Step 29000: Minibatch Loss: 0.020383

Step 30000: Minibatch Loss: 0.020150

Original Images

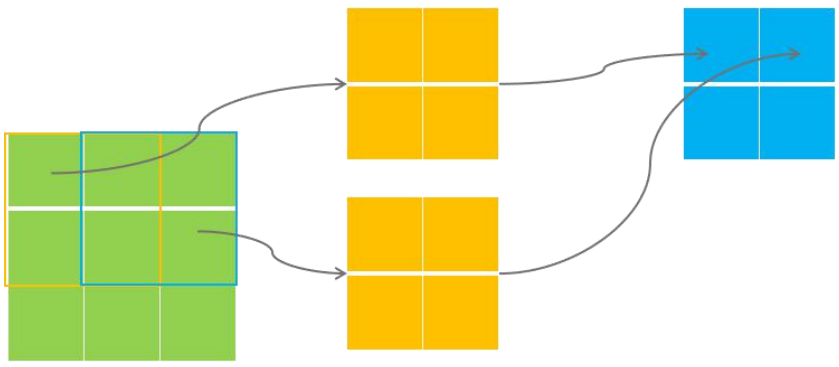


Reconstructed Images



Convolutional Neural Networks

It consists of Convolution and Pooling operations. These basic operations are the elements of deep convolutional neural network. The key idea is to design convolutional layers. In order to visualize the computational model, we use matrix multiplication. Here, we one-channel image as an example.



$$\begin{bmatrix} x_{11} & \cdots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mk} \end{bmatrix}$$

To better illustrate the operation, we use very simple matrix. For Convolutional operation,

filter

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \rightarrow \begin{bmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{bmatrix}$$

input

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \rightarrow \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix}$$

Then we can find the product of the two matrices.

$$\begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{bmatrix} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix}$$

$$\begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix} \rightarrow \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

The gradient descent can be calculated.

$$\frac{\partial h}{\partial w} = \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Then we have

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

Through reshape the matrix, we can do the matrix operations easily. We can find the gradient and update weights easily.

For pooling operation,

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \rightarrow \begin{bmatrix} c_{11} & c_{12} & c_{21} & c_{22} \\ c_{12} & c_{13} & c_{22} & c_{23} \\ c_{21} & c_{22} & c_{31} & c_{32} \\ c_{22} & c_{23} & c_{32} & c_{33} \end{bmatrix}$$

Find the maximum in each row

$$\begin{bmatrix} c_{11} & c_{12} & c_{21} & c_{22} \\ c_{12} & c_{13} & c_{22} & c_{23} \\ c_{21} & c_{22} & c_{31} & c_{32} \\ c_{22} & c_{23} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} p_{11} \\ p_{12} \\ p_{21} \\ p_{22} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

Then, we get

$$pooling\left(\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}\right) = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

For a three-channel image, we can use the same method to do matrix multiplication.

$$filter \begin{cases} \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \end{bmatrix} \\ \begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \end{bmatrix} \\ \begin{bmatrix} w_{11}^3 & w_{12}^3 \\ w_{21}^3 & w_{22}^3 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix} \end{cases} \rightarrow \begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \\ w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \\ w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix}$$

$$
\begin{cases}
\begin{bmatrix} x_{11}^1 & x_{12}^1 & x_{13}^1 \\ x_{21}^1 & x_{22}^1 & x_{23}^1 \\ x_{31}^1 & x_{32}^1 & x_{33}^1 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 \\ x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 \\ x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 \\ x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 \end{bmatrix} \\
\begin{bmatrix} x_{11}^2 & x_{12}^2 & x_{13}^2 \\ x_{21}^2 & x_{22}^2 & x_{23}^2 \\ x_{31}^2 & x_{32}^2 & x_{33}^2 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 \\ x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 \\ x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 \\ x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 \end{bmatrix} \\
\begin{bmatrix} x_{11}^3 & x_{12}^3 & x_{13}^3 \\ x_{21}^3 & x_{22}^3 & x_{23}^3 \\ x_{31}^3 & x_{32}^3 & x_{33}^3 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^3 & x_{12}^3 & x_{21}^3 & x_{22}^3 \\ x_{12}^3 & x_{13}^3 & x_{22}^3 & x_{23}^3 \\ x_{21}^3 & x_{22}^3 & x_{31}^3 & x_{32}^3 \\ x_{22}^3 & x_{23}^3 & x_{32}^3 & x_{33}^3 \end{bmatrix}
\end{cases}
\rightarrow
\begin{bmatrix}
x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 & x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 & x_{11}^3 & x_{12}^3 & x_{21}^3 & x_{22}^3 \\
x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 & x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 & x_{12}^3 & x_{13}^3 & x_{22}^3 & x_{23}^3 \\
x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 & x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 & x_{21}^3 & x_{22}^3 & x_{31}^3 & x_{32}^3 \\
x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 & x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 & x_{22}^3 & x_{23}^3 & x_{32}^3 & x_{33}^3
\end{bmatrix}
$$

$$
\begin{bmatrix}
x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 & x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 & x_{11}^3 & x_{12}^3 & x_{21}^3 & x_{22}^3 \\
x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 & x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 & x_{12}^3 & x_{13}^3 & x_{22}^3 & x_{23}^3 \\
x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 & x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 & x_{21}^3 & x_{22}^3 & x_{31}^3 & x_{32}^3 \\
x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 & x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 & x_{22}^3 & x_{23}^3 & x_{32}^3 & x_{33}^3
\end{bmatrix}
\begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \\ w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \\ w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix}
=
\begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix}
$$

```python
import tensorflow as tf
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)


# Training Parameters
learning_rate = 0.01
num_steps = 1000
batch_size = 128
display_step = 10

# Network Parameters
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
```

```
dropout = 0.75 # Dropout, probability to keep units

# tf Graph input
X = tf.placeholder(tf.float32, [None, num_input])
Y = tf.placeholder(tf.float32, [None, num_classes])
keep_prob = tf.placeholder(tf.float32) # dropout (keep probability)


# Create some wrappers for simplicity
def conv2d(x, W, b, strides=1):

    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)

    return tf.nn.relu(x)


def maxpool2d(x, k=2):

    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                          padding='SAME')


# Create model
def conv_net(x, weights, biases, dropout):
    # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])

    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)

    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
```

```
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)

    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])

    return out


# Store layers weight & bias
weights = {
    # 5x5 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    # 5x5 conv, 32 inputs, 64 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    # fully connected, 7*7*64 inputs, 1024 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    # 1024 inputs, 10 outputs (class prediction)
    'out': tf.Variable(tf.random_normal([1024, num_classes]))
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}


# Construct model
logits = conv_net(X, weights, biases, keep_prob)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model
```

```
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:
    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y, keep_prob: 0.8})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y,
                                                                 keep_prob: 1.0})

            print("Step " + str(step) + ", Minibatch Loss= " +
                  "{:.4f}".format(loss) + ", Training Accuracy= " +
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 256 MNIST test images
    print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: mnist.test.images[:256],
                                                             Y: mnist.test.labels[:256],
                                                             keep_prob: 1.0}))
```

Step 980, Minibatch Loss= 10.0609, Training Accuracy= 0.984

Step 990, Minibatch Loss= 0.0000, Training Accuracy= 1.000

Step 1000, Minibatch Loss= 2.1988, Training Accuracy= 0.984

Optimization Finished!

Testing Accuracy: 0.97265625

In the optimization process, the learning rate has an impact on the accuracy, which means it influences the convergence. In practice, the learning rate should be quite small to avoid fluctuation. The filters required need to be designed carefully to achieve the high accuracy results.