Neural Network and Machine Learning

Midterm Report for Summer of Science - 2020

Name: Sahasra Ranjan
Mentor: Amitrajit Bhattacharjee
Roll no.: 190050102

Contents

Introduction to Machine Learning

Machine Learning is undeniably one of the most influential and powerful technologies in today's world. It is a tool for turning information into knowledge. In the past 50 years, there has been an explosion of data. This mass of data is useless unless we analyse it and find the patterns hidden within. Machine learning techniques are used to automatically find the valuable underlying patterns within complex data that we would otherwise struggle to discover. The hidden patterns and knowledge about a problem can be used to predict future events and perform all kinds of complex decision making.

There are multiple forms of Machine Learning; supervised, unsupervised , semi-supervised and reinforcement learning. Each form of Machine Learning has differing approaches, but they all follow the same underlying process and theory. I'll cover supervised and unsupervised learning for my report.

1.1 Supervised and Unsupervised Learning

The most basic thing to remember is that we already know what out correct output should look like in Supervised Learning. But, we have little or no idea about what out results should look like.

Supervised Learning:

- Classification: Spam/Not-spam.
- Regression: Predicting age.

Unsupervised Learning:

- Clustering: Grouping based on different variables.
- Non Clustering: Finding structue in chaotic environment.

Linear Regression

2.1 Linear Regression with one variable

Regression being a part of Supervised Learning is used for estimatng data (Real-valued output).

2.1.1 Cost Function

This function measures the performance of a Machine Learning model for given data.

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost Function:

$$J(\theta_0, \theta_1) = 1/2m \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.1}$$

Goal: Minimize cost function with θ_0, θ_1 as parameters.

2.1.2 Gradient Descent

Basic idea:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we end up at minima.

Algorithm: repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \tag{2.2}$$

(for $j = 0, 1$,here).

Intution: If α is too small, descent can be slow and if too large, descent may fail to converge or even diverge. Gradient descent can converge to a local minimum, even with

fixed learning rate α. As we approach local mimimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.

2.1.3 Gradient Descent for linear regression

Combining gradient descent algorithm with linear regression model, we get:

$$j = 0 : \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = 1/2 \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \tag{2.3}$$

$$j = 1 : \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = 1/2 \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x^{(i)} \tag{2.4}$$

Now, we can repeat 2.3 and 2.4 until convergence to obtain the minima.

"Batch" gradient descent: Each step of gradient descent uses all the training examples. For eq. "m" batches in equation 2.1.

2.2 Multivariate Linear Regression

Linear regression involving more than one variable. For eq., Predicting price of a house based on parameters "Plot Area", "No. of Floors", "Connectivity with markets", etc.

2.2.1 Multiple Features

The multivariable form of the hypothesis is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n. \tag{2.5}$$

This hypothesis funtion can be concisely represented as:

$$h_\theta(x) = \theta^T x \tag{2.6}$$

where, θ^T is a 1xn matrix consisting of $\theta_0, \theta_1, \theta_2 ... \theta_n$.

2.2.2 Gradient Descent for Multiple Variables

Gradient descent formula for Multiple variable will be similar to that of single variable.

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)} \tag{2.7}$$

Repeating this equation until convergence will give the minima. [1]

[1] $x_0 = 1$ in equation 2.7

Feature Scaling

Feature Scaling is used to reduce the number of iterations in Gradient Descent. Basic idea of feature scaling is to bring all the features on the same scale. (in general we try to approximate every feature in the range $-1 < x_i < 1$)

Reducing the number of iteration doesn't mean making computation of each step easier. And also it does not effect comtational efficiency of Normal Equation.

Mean Normalisation

Mean Normalisation makes features to have approximately zero mean.

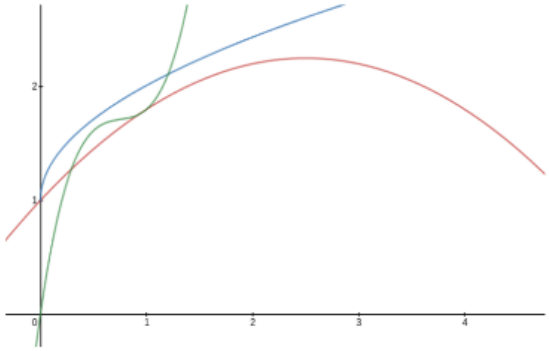
Learning Rate

If α is too small: slow convergence.

if α is too large: $J(\theta)$ may not decrease on every iteration, or may not converge.

Polynomial Regression

Selecting proper polynomial for fitting data is very important.



Red: Quadratic

Blue: Square root funtion $\theta_0 + \theta_1 x + \theta_2 \sqrt{x}$

Green: Cubic function

2.3 Normal equation

Normal Equation is a method to solve for θ_T analytically, by creating a $m \times (n+1)$ matrix X and another $m \times 1$ matrix Y.[2]

[2]Every element of first column of matrix X is 1 and other are the feature's coefficient

Mathematically θ is given as:

$$\theta = (X^T X)^{-1} X^t y \tag{2.8}$$

Gradient Descent	Normal Equation
Need to choose α	No need to choose α
Needs many iteration	Don't need to iterate
Works well with large n	Slow for large n

Reasons for non-invertiblity of $X^T X$

- Redundant features (linear dependence) [3]
- Too many features (m <= n)

```
function J = computeCost(X, y, theta)
  % COMPUTECOST Compute cost for linear regression
  %   J = COMPUTECOST(X, y, theta) computes the cost of using theta as the
  %   parameter for linear regression to fit the data points in X and y

  % Initialize some useful values
  m = length(y); % number of training examples

  J = ((X*theta-y)'*(X*theta-y))/(2*m);
end



function [theta, J_history] = gradientDescent(X, y, theta, alpha,
    num_iters)
  % GRADIENTDESCENT Performs gradient descent to learn theta
  %   theta = GRADIENTDESCENT(X, y, theta, alpha, num_iters) updates theta
    by
  %   taking num_iters gradient steps with learning rate alpha

  % Initialize some useful values
  m = length(y); % number of training examples
  J_history = zeros(num_iters, 1);

  for iter = 1:num_iters,
    theta = theta - alpha*(X'*(X*theta-y))/m;
      % Save the cost J in every iteration
      J_history(iter) = computeCost(X, y, theta);

  end
end
```

[3] Eg. Using both m^2 & $(feet)^2$ features

Logistic Regression

3.1 Classification and Represention

3.1.1 Classification

The classification problem is just like the regression problem, except that the values we now want to predict take on onle a small number of discrete values. For now, we'll discuss binary classification problem.

3.1.2 Hypothesis Representation

We may use out old regression algorithm by classifying data on the basis of a threshold. But it will have very poor performance.

We will introduce "Sigmoid Function", also called the "Logistic Function":

$$h_\theta(x) = g(\theta^T x) \tag{3.1}$$

$$z = \theta^T x \tag{3.2}$$

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3.3}$$

This is how the Sigmoid Function looks like:

3.1.3 Decesion Boundary

The decesion boundary is the line that separates the area where y=0 and where y=1.
It is similar to the decesion boundary for linear regression, the only difference is distibution of values (linear and sigmoid)

3.2 Logistic Regression Model

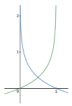
3.2.1 Cost Function

Cost funtion for logistic regression looks like:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \tag{3.4}$$

$$Cost(h_\theta(x), y) = -\log\left(h_\theta(x)\right) \qquad \text{if } y = 1$$

$$Cost(h_\theta(x), y) = -\log\left(1 - h_\theta(x)\right) \qquad \text{if } y = 0$$



Siplified Cost Funtion

This cost funtion can be compressed into a single funtion:

$$Cost(h_\theta(x), y) = -y\log\left(h_\theta(x)\right) - (1-y)\log\left(1 - h_\theta(x)\right) \tag{3.5}$$

A vectorised implementation is:

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

Vectorised implementation for Gradient Descent:

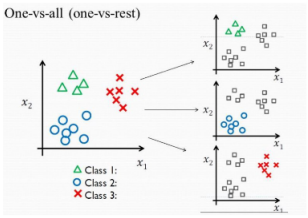
$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - y)$

3.3 Multiclass Classification

3.3.1 One-vs-all

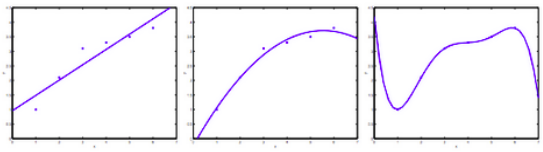
This approach is when data has more than two categories.We divide our problem into n[1] binary classification problems, in each one, we predict the probability considering one of the category to be $+$ve and all other to be $-$ve. Repeating this for all other categories will finally give us all the decesion boundaries.

[1] n = no of categories in dataset



3.4 The Problem of Overfitting

Consider the problem of predicting y from x $\in$ R. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2 , and fit $t = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th} order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**.

How to address this issue?

1. Reduce the number of features:
    - Manually select which features to keep.
    - Use a model selection algorithm.[2]
2. Regularisation:
    - Keep all the features, but reduce the magnitude of parameters θ_j.
    - Regularization works well when we have a lot of slightly useful features.

[2] we'll cover it later

3.4.1 Regularized Cost Function

To solve this problem of overfitting, we can eleminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our cost function:

$$J_\theta = min\ of \left[\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{2} \theta_j^2 \right] \tag{3.6}$$

These extra terms will inflate the cost of extra parameters.

The λ is called the **regularisation parameter**/ It determines how much the costs of out theta parameters are inflated.

3.4.2 Regularized Linear Regression

Repeat {

$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$

$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \qquad j \in \{1, 2 \ldots n\}$

}

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

The first term in the above equation, $\alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of $j\theta_j$ by some amount on every update.

Normal Eequation

This will be the non-iterative approach for regularazation.

To add in regularization, we'll just add another term:

$$\theta = X^T X + \lambda . L^{-1} X^T y \tag{3.7}$$

where, L is $(n+1) \times (n+1)$ matrix with 0 at the top lest, 1's down the diagonal and all other element '0'.

```
function g = sigmoid(z)
  %SIGMOID Compute sigmoid function
  %   g = SIGMOID(z) computes the sigmoid of z.

```

```
  % You need to return the following variables correctly
  g = ones(size(z));
  g = g ./ (1+exp(-z));
end

function [J, grad] = costFunction(theta, X, y)
  %COSTFUNCTION Compute cost and gradient for logistic regression
  %   J = COSTFUNCTION(theta, X, y) computes the cost of using theta as the
  %   parameter for logistic regression and the gradient of the cost
  %   w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples
  h = sigmoid(X*theta);

  % You need to return the following variables correctly
  J = 1/m * (-y'*log(h)-(1-y)'*log(1-h));

  grad = 1/m * (h-y)'*X;

end

function [J, grad] = costFunctionReg(theta, X, y, lambda)
  %COSTFUNCTIONREG Compute cost and gradient for logistic regression with regularization
  %   J = COSTFUNCTIONREG(theta, X, y, lambda) computes the cost of using
  %   theta as the parameter for regularized logistic regression and the
  %   gradient of the cost w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  [c,g] = costFunction(theta, X, y);

  temp = lambda/(2*m) * theta'*theta;
  temp = temp - lambda/(2*m)*theta(1,1)^2;
  J = c+temp;


  grad = g + lambda/m * theta';
  grad(1,1) = g(1,1);

end

function out = mapFeature(X1, X2)
  % MAPFEATURE Feature mapping function to polynomial features
  %
  %   MAPFEATURE(X1, X2) maps the two input features
  %   to quadratic features used in the regularization exercise.
  %
  %   Returns a new feature array with more features, comprising of
  %   X1, X2, X1.^2, X2.^2, X1*X2, X1*X2.^2, etc..

```

```
  degree = 6;
  out = ones(size(X1(:,1)));
  for i = 1:degree
      for j = 0:i
          out(:, end+1) = (X1.^(i-j)).*(X2.^j);
      end
  end
end

function plotDecisionBoundary(theta, X, y)
  %PLOTDECISIONBOUNDARY Plots the data points X and y into a new figure
   with
  %the decision boundary defined by theta
  %   PLOTDECISIONBOUNDARY(theta, X,y) plots the data points with + for
   the
  %   positive examples and o for the negative examples. X is assumed to
   be
  %   a either
  %   1) Mx3 matrix, where the first column is an all-ones column for the
  %      intercept.
  %   2) MxN, N>3 matrix, where the first column is all-ones

  % Plot Data
  plotData(X(:,2:3), y);
  hold on

  if size(X, 2) <= 3
      % Only need 2 points to define a line, so choose two endpoints
      plot_x = [min(X(:,2))-2,  max(X(:,2))+2];

      % Calculate the decision boundary line
      plot_y = (-1./theta(3)).*(theta(2).*plot_x + theta(1));

      % Plot, and adjust axes for better viewing
      plot(plot_x, plot_y)

      % Legend, specific for the exercise
      legend('Admitted', 'Not admitted', 'Decision Boundary')
      axis([30, 100, 30, 100])
  else
      % Here is the grid range
      u = linspace(-1, 1.5, 50);
      v = linspace(-1, 1.5, 50);

      z = zeros(length(u), length(v));
      % Evaluate z = theta*x over the grid
      for i = 1:length(u)
          for j = 1:length(v)
              z(i,j) = mapFeature(u(i), v(j))*theta;
          end
      end
      z = z'; % important to transpose z before calling contour

      % Plot z = 0
```

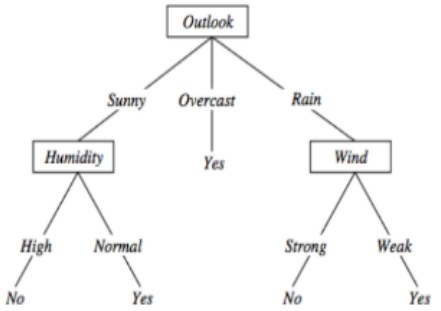
```
      % Notice you need to specify the range [0, 0]
      contour(u, v, z, [0, 0], 'LineWidth', 2)
  end
  hold off

end

function p = predict(theta, X)
  %PREDICT Predict whether the label is 0 or 1 using learned logistic
  %regression parameters theta
  %   p = PREDICT(theta, X) computes the predictions for X using a
  %   threshold at 0.5 (i.e., if sigmoid(theta'*x) >= 0.5, predict 1)

  m = size(X, 1); % Number of training examples

  % You need to return the following variables correctly
  p = zeros(m, 1);

  p = sigmoid(X*theta);

  for i = [1:m],
    if p(i,1) >= 0.5,
      p(i,1) = 1;
     else
      p(i,1) = 0;
    end
  end
end
```

Decision Tree

A decision tree is a map of the possible outcomes of a series of a related choices. It allows to weigh possible actions against one another based of various factors.

It uses a tree-like model of decision. It typically starts with a single node, which branches into possible outcomes. Each of those outcomes leads to additional nodes, which branch off into other possiblities.

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



⟨*Outlook=Sunny, Temp=Hot, Humidity=High, Wind=Strong*⟩ No

4.0.1 Advantages and Disadvantages of Decision Trees

Advantages:

- Performs classification without requiring much computation.
- Provides clear indication of important fields for prediction of classification.

Disadvantages:

- Less appropriate for predicting continious attributes.
- Computationally expensive to train

4.0.2 Creating a Decision Tree

For every node, we have to create subtrees with all the possiblities. and then further repeat for other features.

For eg., In the tennis match problem, for the first node let's check outlook, since having three possiblity (viz. Sunny, Overcast, Rainy), we created three subtrees and then further we keep asking for other features like Humidity & wind to get the final tree.

4.0.3 Greedy Approach for creating decision tree

Greedy approach is implemented by making an optimal local choice at each node. By making these local optimal choices, we reach the approximate optimal solution globaly.

The algorithm can be summarized as:

1. At each stage (node), pick out the best feature as the test conditiion.
2. Now split the node into possibel outcomes (internal nodes)
3. Repeat the above steps till all the test conditions have been exhausted into leaf nodes.

4.0.4 Continuous Features

There might be some feature which are not categorical, for these we need to create possibilities on the basis of aprropriate ranges. One such tree is shown below:

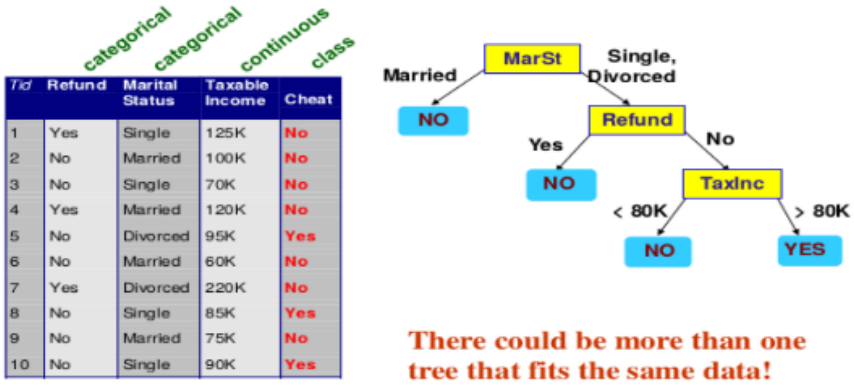
4.0.5 Entropy

In the most layman terms, Entropy is nothing but the **The measure of disorder**. Why is it important to study entropy for machine learning?

Entropy is a measure of disorder or uncertainity and the goal of machine learning models and Data Scientists in general is to reduce uncertainity.

The Mathematical formula for entropy is -

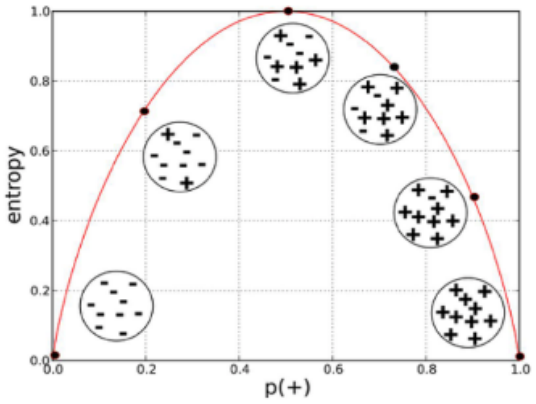
$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{4.1}$$



Where p_i is the frequentist probability of an element/class i in out data.

Let's say we have only two classes, a positive and a negative class. Out of 100 data, suppose that 70 belongs to $-$ve class and 30 to $+$ve. Then, P$+$ will be 0.3 and P$-$ will be 0.7. Entropy E will be given by:

$$E = -\frac{3}{10} \times log_2 \frac{3}{10} - \frac{7}{10} \times log_2 \frac{7}{10} \approx \mathbf{0.88} \tag{4.2}$$



4.0.6 Information Gain

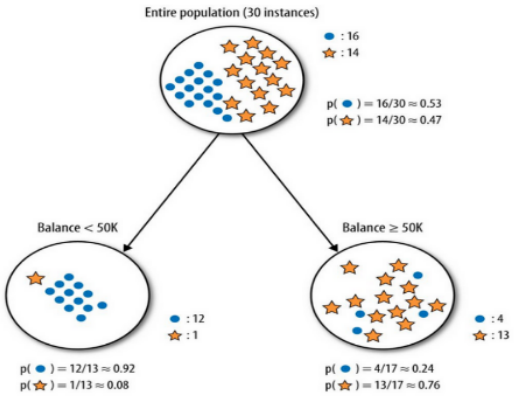
Information gain is basically how much Entropy is removed after training a decision tree.
Higher information gain = more entropy removed.
In technical terms, Information Gain from X on Y is defined as:

$$IG(Y, X) = E(Y) - E(Y|X) \tag{4.3}$$

Basics of information gain is well explained here: A Simple Explanation of Information Gain and Entropy

4.0.7 Example: Decision Tree

Consider an example where we are building a decision tree to predict whether a loan given to a person would result in a write-off or not. Our entire population consists of 30 instances. 16 belong to the write-off class and the other 14 belong to the non-write-off class. We have two features, namely "Balance" that can take on two values: "$<$ 50K" or "$>$ 50K" and "Residence" that can take on three values: "OWN", "RENT" or "OTHER". I'm going to show you how a decision tree algorithm would decide what attribute to split on first and what feature provides more information, or reduces more uncertainty about our target variable out of the two using the concepts of Entropy and Information Gain. The dots are the data points with class right-off and the stars are the non-write-offs.

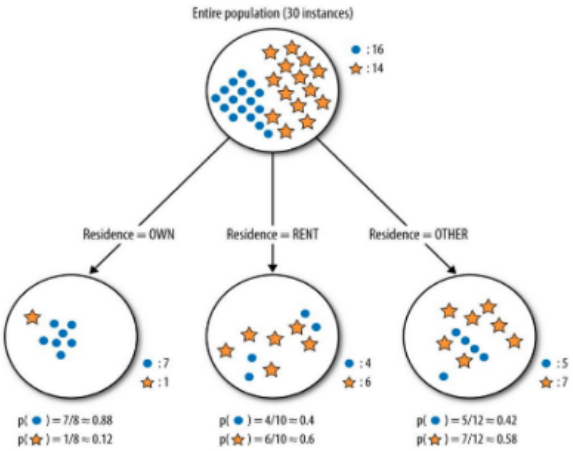


Splitting the parent node on attribute balance gives us 2 child nodes. The left node gets 13 of the total observations with 12/13 (0.92 probability) observations from the write-off class and only 1/13(0.08 probability) observations from the non-write of class. The right node gets 17 of the total observation with 13/17(0.76 probability) observations from the non-write-off class and 4/17 (0.24 probability) from the write-off class.

Let's calculate[1] the entropy for the parent node and see how much uncertainty the tree can reduce by splitting on Balance. Splitting on feature ,"Balance" leads to an information gain of 0.37 on our target variable. Let's do the same thing for feature, "Residence" to see how it compares.

Splitting the tree on Residence gives us 3 child nodes. The left child node gets 8 of the total observations with 7/8 (0.88 probability) observations from the write-off class and only 1/8 (0.12 probability) observations from the non-write-off class. The middle child nodes gets 10 of the total observations with 4/10 (0.4 probability) observations of the write-off class and 6/10(0.6 probability) observations from the non-write-off class. The right child node gets 12 of the total observations with 5/12 (0.42 probability) observations from the write-off class and 7/12 (0.58) observations from the non-write-off class. We already know the entropy for the parent node. We simply need to calculate the

[1]See this for calculations and further refrences: Entropy: How Decision Trees Make Decisions



entropy after the split to compute the information gain from "Residence"

The information gain from feature, Balance is almost 3 times more than the information gain from Residence! If you go back and take a look at the graphs you can see that the child nodes from splitting on Balance do seem purer than those of Residence. However the left most node for residence is also very pure but this is where the weighted averages come in play. Even though that node is very pure, it has the least amount of the total observations and a result contributes a small portion of it's purity when we calculate the total entropy from splitting on Residence. This is important because we're looking for overall informative power of a feature and we don't want our results to be skewed by a rare value in a feature.

By itself the feature, Balance provides more information about our target variable than Residence. It reduces more disorder in our target variable. A decision tree algorithm would use this result to make the first split on our data using Balance. From here on, the decision tree algorithm would use this process at every split to decide what feature it is going to split on next. In a real world scenario , with more than two features the first split is made on the most informative feature and then at every split the information gain for each additional feature needs to be recomputed because it would not be the same as the information gain from each feature by itself. The entropy and information gain would have to be calculated after one or more splits have already been made which would change the results. A decision tree would repeat this process as it grows deeper and deeper till either it reaches a pre-defined depth or no additional split can result in a higher information gain beyond a certain threshold which can also usually be specified as a hyper-parameter!

4.0.8 Worked-out Implementation

```
1 import pandas as pd
```

```
import numpy as np

eps = np.finfo(float).eps

from numpy import log2 as log
dataset = {'Taste':['Salty','Spicy','Spicy','Spicy','Spicy','Sweet','Salty','Sweet','Spicy','Salty'],
       'Temperature':['Hot','Hot','Hot','Cold','Hot','Cold','Cold','Hot','Cold','Hot'],
       'Texture':['Soft','Soft','Hard','Hard','Hard','Soft','Soft','Soft','Soft','Hard'],
       'Eat':['No','No','Yes','No','Yes','Yes','No','Yes','Yes','Yes']}

df = pd.DataFrame(dataset, columns=['Taste','Temperature','Texture','Eat'])

def find_entropy(df):
    '''
    Function to calculate the entropy of the label i.e. Eat.
    '''
    Class = df.keys()[-1]
    entropy = 0
    values = df[Class].unique()
    for value in values:
        fraction = df[Class].value_counts()[value]/len(df[Class])
        entropy += -fraction*np.log2(fraction)
    return entropy


def find_entropy_attribute(df,attribute):
    '''
    Funtion to calculate the entropy of all the features.
    '''
    Class = df.keys()[-1]
    target_variables = df[Class].unique()
    variables = df[attribute].unique()
    entropy2 = 0

    for variable in variables:
        entropy = 0
        for target_variable in target_variables:
            num = len(df[attribute][df[attribute]==variable][df[Class] == target_variable])
            den = len(df[attribute][df[attribute]==variable])
            fraction = num/(den+eps)
            entropy += -fraction*log(fraction+eps)
        fraction2 = den/len(df)
        entropy2 += -fraction2*entropy
    return abs(entropy2)


def find_winner(df):
    '''
    Function to find the feature with the highest information gain
```

```
    '''
    Entropy_att = []
    IG = []
    for key in df.keys()[:-1]:
        IG.append(find_entropy(df) - find_entropy_attribute(df,key))

    return df.keys()[:-1][np.argmax(IG)]

def get_subtable(df, node, value):
    '''
    Function to get a subtable of met conditions.

    node: Column name
    value: Unique value of the column
    '''
    return df[df[node] == value].reset_index(drop=True)

def buildTree(df,tree=None):
    '''
    Function to build the ID3 Decision Tree.
    '''
    Class = df.keys()[-1]
    #Here we build our decision tree

    #Get attribute with maximum information gain
    node = find_winner(df)

    #Get distinct value of that attribute e.g Salary is node and Low,Med
   and High are values
    attValue = np.unique(df[node])

    #Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}

   #We make loop to construct a tree by calling this function recursively.
    #In this we check if the subset is pure and stops if it is pure.

    for value in attValue:

        subtable = get_subtable(df,node,value)
        clValue,counts = np.unique(subtable['Eat'],return_counts=True)

        if len(counts)==1:#Checking purity of subset
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subtable) #Calling the function
   recursively

    return tree


tree = buildTree(df)
```

```
103
104
105 def predict(inst,tree):
106     '''
107     Function to predict for any input variable.
108     '''
109     #Recursively we go through the tree that we built earlier
110
111     for nodes in tree.keys():
112
113         value = inst[nodes]
114         tree = tree[nodes][value]
115         prediction = 0
116
117         if type(tree) is dict:
118             prediction = predict(inst, tree)
119         else:
120             prediction = tree
121             break;
122
123     return prediction
124
125 data = {'Taste':'Salty','Temperature':'Cold','Texture':'Hard'}
126
127 inst = pd.Series(data)
128
129 prediction = predict(inst,tree)
130
131 print(prediction)
132
133 # Prints "Yes"
```

Naive Bayes

5.1 Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.

What actually Bayes' theorem is?

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{5.1}$$

$P(A|B)$ is the probability of **A** happening, given that **B** has occured.

Why "Naive"? Because the presence of one particular feature does not affect the other.

Without going to deep, let's see an example:

5.1.1 The Golf Match Problem

Consider the problem of playing golf, dataset for the same:

Bayes theorem for this example can be rewritten as:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)} \tag{5.2}$$

The variable **y** is the class variable (play golf), which represents if it is suitable to play golf or not given the conditions. Variable **X** is a matrix representing the parameters/features.

$\mathbf{X} = (x_1, x_2, x_3, ..., x_n)$ $x_1, x_2, ...x_n$ represent the features[1].

[1] temperature, humidity and windy (here)

	OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY GOLF
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

$$P(y|x_1,...,x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)} \tag{5.3}$$

These values can be obtained by looking at the dataset and substituting them into the equation will give us the result.

In our case, the the class variable(**y**) has only two outcomes, yes or no. Therefore, we need to find class **y** with maximum probability.

$$y = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y) \tag{5.4}$$

5.1.2 Types of Naive Bayes classifier:

Miltinomial Naive Bayes:

This is mostly used for the document classification problem, i.e whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of the words present in the document.

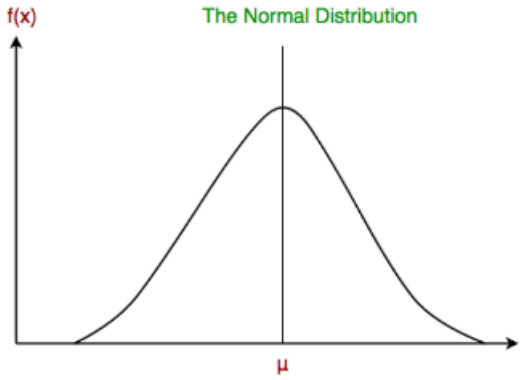
Bernoulli Naive Bayes

This is similar to the multinomial Naive Bayes but the predictors are boolean variables. The parameters that we use to predict the class variable take up only values yes or no, for example, if a

word occurs in the text or not.

Gaussian Naive Bayes

When the predictors take up a continuous value and are not discrete, we assume that these values are sampled from a gaussian distribution.



The formula for conditional probability changes to:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}} \tag{5.5}$$

5.1.3 Laplace Smoothing

It is problematic when a frequency-based probability is zero because it will wipe out all the information in the other probabilities.

A solution would be **Laplace smoothing**, which is a technique for smoothing categorical data. A small-sample correction, or **pseudo-count**, will be incorporated in every probability estimate. Consequently, no probability will be zero. this is a way of regularizing Naive Bayes, and when the pseudo-count is zero, it is called Laplace smoothing. While in the general case it is often called **Lidstone smoothing.**

$$P_{i,\ \alpha-smoothed} = \frac{x_i + \alpha}{N + \alpha d} \tag{5.6}$$

where, $\alpha > 0$ the "pseudocount" is a smoothing parameter. And, $1/d$ is the Uniform Probability.

Note

In practice, we use logs to represent probabilities:

$$\log\left(P(x_1|y)P(x_2|y)...P(x_n|y)\right) = \log P(x_1|y) + \log P(x_2|y) + ... + \log P(x_n|y) \tag{5.7}$$

5.1.4 Applications

Naive Bayes algorithms are mostly used in sentiment analysis, spam filtering, recommendation systems etc. They are fast and easy to implement but their biggest disadvantage is that the requirement of predictors to be independent. In most of the real-life cases, the predictors are dependent, this hinders the performance of the classifier.

Worked-out Implementation[2]

```
import numpy as np
import pandas as pd

mush = pd.read_csv("./mushrooms.csv")
# This data consist of missing data with '?' as value. All the missing
    values are from single column.

# To remove it.
mush.replace('?',np.nan,inplace=True)
print(len(mush.columns),"columns, after dropping NA,",len(mush.dropna(axis
    =1).columns))

mush.dropna(axis=1,inplace=True)

target = 'class'
features = mush.columns[mush.columns != target]
classes = mush[target].unique()

test = mush.sample(frac=0.3)
mush = mush.drop(test.index)

# Probabilties Calculation
probs = {}
probcl = {}
for x in classes:
    mushcl = mush[mush[target]==x][features]
    clsp = {}
    tot = len(mushcl)
    for col in mushcl.columns:
        colp = {}
        for val,cnt in mushcl[col].value_counts().iteritems():
            pr = cnt/tot
```

[2]**Output:**
23 columns, after dropping NA, 22
Test 1
5671 correct of 5687
Accuracy: 0.997186565851943
Test 2
2433 correct of 2437
Accuracy: 0.9983586376692655

This implementation of Naive Bayes Algorithm has an accuracy of approximately 99.8% which is good in the first go.

```
            colp[val] = pr
        clsp[col] = colp
    probs[x] = clsp
    probcl[x] = len(mushcl)/len(mush)


def probabs(x):
    #X - pandas Series with index as feature
    if not isinstance(x,pd.Series):
        raise IOError("Arg must of type Series")
    probab = {}
    for cl in classes:
        pr = probcl[cl]
        for col,val in x.iteritems():
            try:
                pr *= probs[cl][col][val]
            except KeyError:
                pr = 0
        probab[cl] = pr
    return probab


def classify(x):
    probab = probabs(x)
    mx = 0
    mxcl = ''
    for cl,pr in probab.items():
        if pr > mx:
            mx = pr
            mxcl = cl
    return mxcl

#Train data
b = []
for i in mush.index:
    b.append(classify(mush.loc[i,features]) == mush.loc[i,target])
print("Test 1")
print(sum(b),"correct of",len(mush))
print("Accuracy:", sum(b)/len(mush))


#Test data
b = []
for i in test.index:
    b.append(classify(test.loc[i,features]) == test.loc[i,target])
print("Test 2")
print(sum(b),"correct of",len(test))
print("Accuracy:",sum(b)/len(test))
```

k-Nearest Neighbours

K-Nearest Neighbours is one of the most basic yet essential algorithms in Machine Learning. It has intense application in pattern recognition, data mining and intrusion detection.

6.0.1 The Algorithm

1. Load the data
2. Initialize K
3. For each example in data
   - Calculate the distance between the query example and the current example from the data
   - Add the distance and the index of the example to an ordered collection
4. Sort the collection in ascending order.
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. If regression, return the mean else if classification, return the mode of the K labels

6.0.2 Chosing the right value for K

To select the K that's right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

Here are some things to keep in mind:

1. As we decrease the value of K to 1, our predictions become less stable.
2. Inversely, as we increase the value of K, our predictions become more stable due to majority voting (up to a certain point). Eventually, increasing K will increase the error.
3. In cases where we are taking majority vote, we usually make "K" an odd number to have a tiebreaker.

Advantages

- Simple and easy to implement.
- No need to build a model, tune several parameters, or make additional assumptions.
- Versatile, can be used for classification, regresion and search as well.

Disadvantages

- Slow with increase in number of examples and/or predictors.

6.0.3 Applications

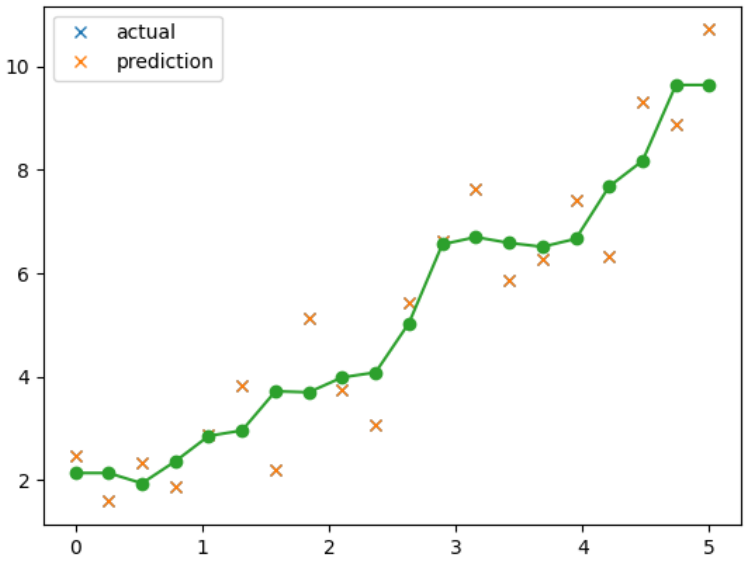
kNN's main disadvantage of becoming significantly slower as the volume of data increases makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.

However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, KNN can still be useful in solving problems that have solutions that depend on identifying similar objects. An example of this is using the KNN algorithm in recommender systems, an application of KNN-search.

6.0.4 Worked-out Implementation

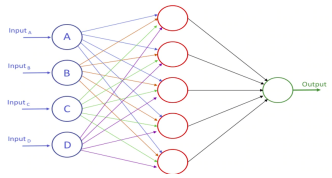
```
import numpy as np
from matplotlib import pyplot as plt

class KNearestNeighbor(object):
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        # Store the original points

        self.X = X
        self.y = y

        return self

    def predict(self, X, y=None):

        # Initialize a zero distance matrix
        dists = np.zeros((X.shape[0], self.X.shape[0]))

        # Loop through all possible pairs and compute their distances
        for i in range(dists.shape[0]):
            for j in range(dists.shape[1]):
                dists[i, j] = self.distance(X[i], self.X[j])


```

```
        # Sort the distance array row-wise, and select the top k indexes for each row
        indexes = np.argsort(dists, axis=1)[:,:self.k]

        # Compute the mean of the values
        mean = np.mean(self.y[indexes], axis=1)

        return mean

    def distance(self, x, y):
        return np.sqrt(np.dot(x - y, x - y))

x = np.linspace(0, 5, 20)

m = 1.5
c = 1
y = m * x + c + np.random.normal(size=(20,))

plt.plot(x, y, 'x')

model = KNearestNeighbor(k=3)

model.fit(x,y)

predicted = model.predict(x.reshape(-1, 1))

plt.plot(
    x, y, "x",
    x, model.predict(x), "-o"
)
plt.legend(["actual", "prediction"])

plt.show()
```


actual
prediction
10
8
6
4
2
0
1
2
3
4
5

Neural Networks

At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1 ... x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1.



Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer". We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

These "hidden layer" nodes are called as "activation units". The values for each activation nodes are represented as:

$$\begin{matrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{matrix} \quad \begin{matrix} a_1^{(2)} \\ \rightarrow a_2^{(2)} \rightarrow h_\theta(x) \\ a_3^{(2)} \end{matrix}$$

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3) \tag{7.1}$$

$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) \tag{7.2}$$

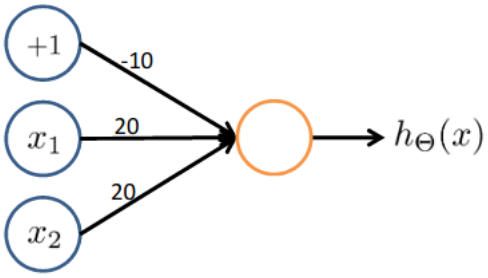
$$a_3^{(2)} = g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) \tag{7.3}$$

$$h_\theta(x) = a_1^{(3)]} = a_1^{(2)} = g(\theta_{10}^{(2)} x_0 + \theta_{11}^{(2)} x_1 + \theta_{12}^{(2)} x_2 + \theta_{13}^{(2)} x_3)5 \tag{7.4}$$

From these equations, we can conclude that we will get a matrix for each layer to calculate the weight for the second layer.

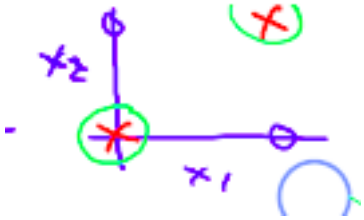
7.1 Intutions

7.1.1 OR Function



x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

7.1.2 Important Note



For any prediction which involve a straight line as decision boundary, we can represent it with a neural network without any hidden layer but otherwise we'll have to include few hidden layers. An important point to note is that we can represent almost any distribution with cenrain arrangement of neural network.

7.2 Multiclass Classification

To classify data into multiple classes, we'll have to define out set of resulting classes as y:

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix},$$

7.3 Cost Function

Let's first define a few variables that we'll need to use:

- L = total number of layers in the network
- numbers of units (non counting bias unit) in layer 1
- K = numbet of output unit/classes

Cost function for neural networks will be slightly more complicated as it involves few other factors of 'K' and 'L' defined earlier.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log((h_\theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{j,i}^{(L)})^2 \quad (7.5)$$

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer.
- the triple sum simply adds up the squares of all the individual θs in the entire network.
- the i in the triple sum does not refer to training example i.

7.4 Backpropagation Algorithm

Given training set $(x^(1), y^(1))...(x^(m), y^(m))$

- Set $\Delta_{i,j}^{(l)} := 0$ for all (l,i,j), (hence you end up having a matrix full of zeros)

For training example t=1 to m:

1. Set $a^{(1)} := x^{(t)}$
2. Perform forward propagation to compute $a^{(l)}$ for l = 2,3,...,L
   Gradient Computation
3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(L)}$
4. Compute $\delta^{(l)}$
5. $\Delta^{(L)} := \Delta^{(L)} + \delta^{(l+1)}(a^{(l)})^T$

Hence we update our new Δ matrix.

- $D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)} + \lambda\,\Theta_{i,j}^{(l)}$, If j≠0.
- $D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)}$ If j=0

$$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta+\epsilon) - J(\Theta-\epsilon)}{2\epsilon}$$

7.5 Gradient Checking

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

And for multiple features,

$$\frac{\partial}{\partial\Theta_j}J(\Theta) \approx \frac{J(\Theta_1,...,\Theta_j+\epsilon,...,\Theta_n) - J(\Theta_1,...,\Theta_j-\epsilon,...,\Theta_n)}{2\epsilon}$$

7.6 Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights[3]
2. Implement forward propagation to get $h_\theta(x(i))$ for any $x^{(i)}$
3. Implement the cost function

[3] A good choice for $e_init = \frac{\sqrt{6}}{\sqrt{L_in + L_out}}$

4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

```
function [J, grad] = lrCostFunction(theta, X, y, lambda)
  %LRCOSTFUNCTION Compute cost and gradient for logistic regression with
  %regularization
  %   J = LRCOSTFUNCTION(theta, X, y, lambda) computes the cost of using
  %   theta as the parameter for regularized logistic regression and the
  %   gradient of the cost w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;
  grad = zeros(size(theta));

  h = sigmoid(X*theta);

  temp = theta;
  temp(1) = 0;

  J = 1/m * (-log(h)'*y-log(1-h)'*(1-y)) + lambda/(2*m) * temp'*temp;

  grad = 1/m * X'*(h-y) + lambda/m * temp;

  grad = grad(:);
end
% =============================================================

function [all_theta] = oneVsAll(X, y, num_labels, lambda)
  %ONEVSALL trains multiple logistic regression classifiers and returns
   all
  %the classifiers in a matrix all_theta, where the i-th row of all_theta
  %corresponds to the classifier for label i
  %   [all_theta] = ONEVSALL(X, y, num_labels, lambda) trains num_labels
  %   logistic regression classifiers and returns each of these
   classifiers
  %   in a matrix all_theta, where the i-th row of all_theta corresponds
  %   to the classifier for label i

  % Some useful variables
  m = size(X, 1);
  n = size(X, 2);

  all_theta = zeros(num_labels, n + 1);

  % Add ones to the X data matrix
  X = [ones(m, 1) X];
```

```
45
46    for c = (1:num_labels),
47      initial_theta = zeros(n+1,1);
48      options = optimset('GradObj', 'on', 'MaxIter', 50);
49      [theta] = ...
50          fmincg (@(t)(lrCostFunction(t, X, (y==c), lambda)), initial_theta,
      options);
51
52      all_theta(c,:) += theta';
53    end
54  end
55  % =========================================================================
56
57
58  function p = predictOneVsAll(all_theta, X)
59    %PREDICT Predict the label for a trained one-vs-all classifier. The
       labels
60    %are in the range 1..K, where K = size(all_theta, 1).
61    %  p = PREDICTONEVSALL(all_theta, X) will return a vector of predictions
62    %  for each example in the matrix X. Note that X contains the examples
       in
63    %  rows. all_theta is a matrix where the i-th row is a trained logistic
64    %  regression theta vector for the i-th class. You should set p to a
       vector
65    %  of values from 1..K (e.g., p = [1; 3; 1; 2] predicts classes 1, 3, 1,
        2
66    %  for 4 examples)
67
68    m = size(X, 1);
69    num_labels = size(all_theta, 1);
70
71    p = zeros(size(X, 1), 1);
72
73    % Add ones to the X data matrix
74    X = [ones(m, 1) X];
75
76    temp = sigmoid(X*all_theta');
77    [~,p] = max(temp, [], 2);
78
79  end
80  % =========================================================================
81
82
83  function p = predict(Theta1, Theta2, X)
84    %PREDICT Predict the label of an input given a trained neural network
85    %   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X
       given the
86    %   trained weights of a neural network (Theta1, Theta2)
87
88    % Useful values
89    m = size(X, 1);
90    num_labels = size(Theta2, 1);
91
92    p = zeros(size(X, 1), 1);
```

```
93
94    X = [ones(m, 1), X];
95
96    z2 = sigmoid(X*Theta1');
97    z2 = [ones(m, 1), z2];
98
99    z3 = sigmoid(z2*Theta2');
100
101   [~, p] = max(z3, [], 2);
102
103 end
104 %
      ==========================================================================
```

Improving Neural Networks

What to try next for improving our neural networks?

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing or decreasing λ

8.1 Evaulating a Hypothesis

A hypothesis may have a low error for training examples but still be inaccurate (because of overfitting). Thus, to evaulate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a training set and a test set. Typically, the training set consists of 70% of your data and the test set is the remaining 30%.

The test set error

1. For linear regression: $J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$
2. For classification ~ Misclassification error (aka 0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1 & \text{if } h_\Theta(x) \geq 0.5 \text{ and } y = 0 \text{ or } h_\Theta(x) < 0.5 \text{ and } y = 1 \\ 0 & otherwise \end{cases}$$

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

$$\text{Test Error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}), y_{test}^{(i)})$$

This gives us the proportion of the test data that was misclassified.

8.1.1 Model Selection

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

We usually break down out dataset into three sets:

- Training set: 60%
- Cross validation set: 20%
- Test set: 20%

Now to improve our training:

1. Optimize the parameters in θ using training set.
2. Find the polynomial degree d with the least error usign the cross validation set.
3. Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$ (d = θ from polynomial with lower error)

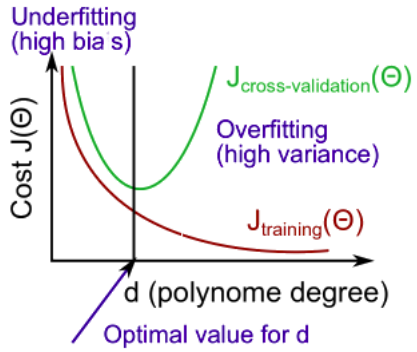
```
function checkNNGradients(lambda)
  %CHECKNNGRADIENTS Creates a small neural network to check the
  %backpropagation gradients
  %   CHECKNNGRADIENTS(lambda) Creates a small neural network to check the
  %   backpropagation gradients, it will output the analytical gradients
  %   produced by your backprop code and the numerical gradients (computed
  %   using computeNumericalGradient). These two gradient computations
    should
  %   result in very similar values.
  %

  if ~exist('lambda', 'var') || isempty(lambda)
      lambda = 0;
  end

  input_layer_size = 3;
  hidden_layer_size = 5;
  num_labels = 3;
  m = 5;

  % We generate some 'random' test data
  Theta1 = debugInitializeWeights(hidden_layer_size, input_layer_size);
  Theta2 = debugInitializeWeights(num_labels, hidden_layer_size);
  % Reusing debugInitializeWeights to generate X
  X  = debugInitializeWeights(m, input_layer_size - 1);
  y  = 1 + mod(1:m, num_labels)';

  % Unroll parameters
  nn_params = [Theta1(:) ; Theta2(:)];

  % Short hand for cost function
  costFunc = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size,
    ...
                                 num_labels, X, y, lambda);

  [cost, grad] = costFunc(nn_params);
  numgrad = computeNumericalGradient(costFunc, nn_params);

```

```
  % Visually examine the two gradient computations.  The two columns
  % you get should be very similar.
  disp([numgrad grad]);
  fprintf(['The above two columns you get should be very similar.\n' ...
           '(Left-Your Numerical Gradient, Right-Analytical Gradient)\n\n']);

  % Evaluate the norm of the difference between two solutions.
  % If you have a correct implementation, and assuming you used EPSILON = 0.0001
  % in computeNumericalGradient.m, then diff below should be less than 1e-9
  diff = norm(numgrad-grad)/norm(numgrad+grad);

  fprintf(['If your backpropagation implementation is correct, then \n' ...
           'the relative difference will be small (less than 1e-9). \n' ...
           '\nRelative Difference: %g\n'], diff);
end
%% ================================================================================



function W = randInitializeWeights(L_in, L_out)
%RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with L_in
%incoming connections and L_out outgoing connections
%   W = RANDINITIALIZEWEIGHTS(L_in, L_out) randomly initializes the weights
%   of a layer with L_in incoming connections and L_out outgoing
%   connections.
%
%   Note that W should be set to a matrix of size(L_out, 1 + L_in) as
%   the first column of W handles the "bias" terms
%

W = zeros(L_out, 1 + L_in);

epsilon_init = 0.12;
W = rand(L_out, 1+L_in) * 2 * epsilon_init - epsilon_init;

end
%% ================================================================================
```

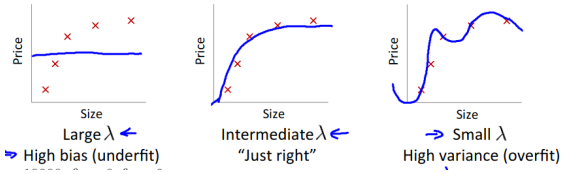
8.2 Bias vs. Variance

High bias (underfitting): both $J_train(\theta)$ and $J_{CV}(\theta)$ will be high and also similar.

High variance (overfitting): $J_train(\theta)$ will be low but $J_{CV}(\theta)$.



8.2.1 Regularization

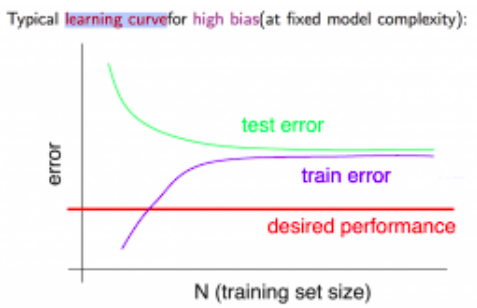


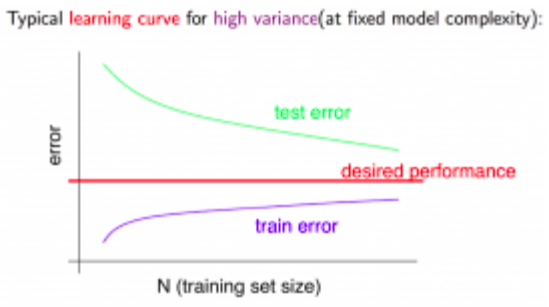
In the figure above, we see that as λ increases, our fit becomes more rigid. On the other hand, as λ approaches 0, we tend to over overfit the data. So how do we choose our parameter λ to get it 'just right' ?

1. Create a list of lambdas.
2. Iterate through the λs and for each, go throught all the models to learn some θ
3. Compute the cross validation error.
4. Select the best combo of $\theta \& \lambda$.

8.2.2 Learning Curves

```
function [J, grad] = linearRegCostFunction(X, y, theta, lambda)
  %LINEARREGCOSTFUNCTION Compute cost and gradient for regularized linear
  %regression with multiple variables
  %   [J, grad] = LINEARREGCOSTFUNCTION(X, y, theta, lambda) computes the
  %   cost of using theta as the parameter for linear regression to fit
    the
  %   data points in X and y. Returns the cost in J and the gradient in
    grad
```






```
7
8   % Initialize some useful values
9   m = length(y); % number of training examples
10
11  % You need to return the following variables correctly
12  J = 0;
13  grad = zeros(size(theta));
14
15  h = X*theta;
16
17  J = (h-y)'*(h-y);
18
19  theta(1)=0;
20
21  J += lambda * theta'*theta;
22  J /= 2*m;
23
24
25  grad = X'*(h-y);
26  grad += lambda*theta;
27  grad /= m;
28
29
30
31  grad = grad(:);
32
33 end
34
35 % =========================================================
36
37 function [error_train, error_val] = ...
38       learningCurve(X, y, Xval, yval, lambda)
39   %LEARNINGCURVE Generates the train and cross validation set errors
      needed
40   %to plot a learning curve
41   %   [error_train, error_val] = ...
42   %       LEARNINGCURVE(X, y, Xval, yval, lambda) returns the train and
43   %       cross validation set errors for a learning curve. In particular,
44   %       it returns two vectors of the same length - error_train and
45   %       error_val. Then, error_train(i) contains the training error for
46   %       i examples (and similarly for error_val(i)).
47   %
```

```
  %   In this function, you will compute the train and test errors for
  %   dataset sizes from 1 up to m. In practice, when working with larger
  %   datasets, you might want to do this in larger intervals.
  %

  % Number of training examples
  m = size(X, 1);

  % You need to return these values correctly
  error_train = zeros(m, 1);
  error_val   = zeros(m, 1);


  for i = 1:m
     Xtemp = X(1:i, :);
     ytemp = y(1:i);

     theta = trainLinearReg(Xtemp, ytemp, lambda);


    error_train(i) = linearRegCostFunction(Xtemp, ytemp, theta, 0);
    error_val(i) = linearRegCostFunction(Xval, yval, theta, 0);
  end

end

% =========================================

function [X_poly] = polyFeatures(X, p)
  %POLYFEATURES Maps X (1D vector) into the p-th power
  %   [X_poly] = POLYFEATURES(X, p) takes a data matrix X (size m x 1) and
  %   maps each example into its polynomial features where
  %   X_poly(i, :) = [X(i) X(i).^2 X(i).^3 ...  X(i).^p];
  %


  % You need to return the following variables correctly.
  X_poly = zeros(numel(X), p);

  X_poly(:,1) = X;

  for i = 2:p,
    X_poly(:,i) = X_poly(:,i-1) .* X;
  end

end
% ================================================

function [lambda_vec, error_train, error_val] = ...
      validationCurve(X, y, Xval, yval)
  %VALIDATIONCURVE Generate the train and validation errors needed to
  %plot a validation curve that we can use to select lambda
  %   [lambda_vec, error_train, error_val] = ...
  %       VALIDATIONCURVE(X, y, Xval, yval) returns the train
```

```
  %       and validation errors (in error_train, error_val)
  %       for different values of lambda. You are given the training set (
    X,
  %       y) and validation set (Xval, yval).
  %

  % Selected values of lambda (you should not change this)
  lambda_vec = [0 0.001 0.003 0.01 0.03 0.1 0.3 1 3 10]';

  % You need to return these variables correctly.
  error_train = zeros(length(lambda_vec), 1);
  error_val = zeros(length(lambda_vec), 1);

  for i = 1:length(lambda_vec)
     lambda = lambda_vec(i);

     % Compute train / val errors when training linear
     % regression with regularization parameter lambda
     % You should store the result in error_train(i)
     % and error_val(i)
     ....
     theta = trainLinearReg(X, y, lambda);
     error_train(i) = linearRegCostFunction(X, y, theta, 0);
     error_val(i) = linearRegCostFunction(Xval, yval, theta, 0);

  end
end
% =========================================================
```

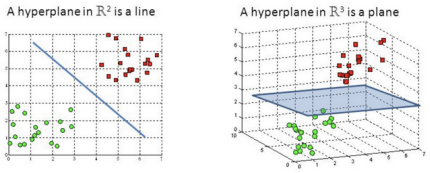
Support Vector Machines

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.

To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

9.1 Hyperplanes and Support Vectors

Hyperplanes are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.

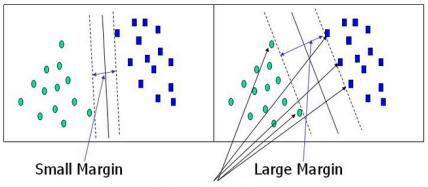


Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

9.2 Large Margin Intuition

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value(0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1,

we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values([-1,1]) which acts as margin.




```
function [C, sigma] = dataset3Params(X, y, Xval, yval)
  %EX6PARAMS returns your choice of C and sigma for Part 3 of the exercise
  %where you select the optimal (C, sigma) learning parameters to use for
   SVM
  %with RBF kernel
  %   [C, sigma] = EX6PARAMS(X, y, Xval, yval) returns your choice of C
   and
  %   sigma. You should complete this function to return the optimal C and
  %   sigma based on a cross-validation set.
  %

  % You need to return the following variables correctly.
  C = 1;
  sigma = 0.3;

  results = eye(64,3);
  errorRow = 0;

  for C_test = [0.01 0.03 0.1 0.3 1, 3, 10 30]
      for sigma_test = [0.01 0.03 0.1 0.3 1, 3, 10 30]
          errorRow = errorRow + 1;
          model = svmTrain(X, y, C_test, @(x1, x2) gaussianKernel(x1, x2,
   sigma_test));
          predictions = svmPredict(model, Xval);
          prediction_error = mean(double(predictions ~= yval));

          results(errorRow,:) = [C_test, sigma_test, prediction_error];
      end
  end

  sorted_results = sortrows(results, 3); % sort matrix by column #3, the
   error, ascending

  C = sorted_results(1,1);
  sigma = sorted_results(1,2);

end
% =========================================================


function sim = gaussianKernel(x1, x2, sigma)
  %RBFKERNEL returns a radial basis function kernel between x1 and x2
  %   sim = gaussianKernel(x1, x2) returns a gaussian kernel between x1
   and x2
```

```
  %   and returns the value in sim

  % Ensure that x1 and x2 are column vectors
  x1 = x1(:); x2 = x2(:);

  sim = 0;

  xdif = x1-x2;

  sim = exp(-(xdif'*xdif)/(2*sigma*sigma));
end
% ================================================================
```

Unsupervised Learning

Unsupervised learning is a type of machine learning that looks for previously undetected patterns in a data set with no pre-existing labels and with a minimum of human supervision. In contrast to supervised learning that usually makes use of human-labeled data, unsupervised learning, also known as self-organization allows for modeling of probability densities over inputs. It forms one of the three main categories of machine learning, along with supervised and reinforcement learning. Semi-supervised learning, a related variant, makes use of supervised and unsupervised techniques.

10.1 K-Means Clustering Algorithm

Kmeans algorithm is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

The way kmeans algorithm works is as follows:

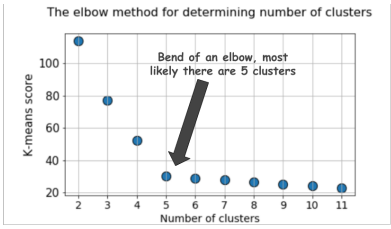
1. Specify number of clusters K.
2. Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement.
3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing:
    - Compute the sum of the squared distance between data points and all centroids.
    - Assign each data point to the closest cluster (centroid).
    - Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

10.1.1 The Elbow Method

For the k-means clustering method, the most common approach for choosing the number of clusters is the so-called **elbow method**. It involves running the algorithm multiple times over a loop, with an increasing number of cluster choice and then plotting a clustering score as a function of the number of clusters.

What is the score or metric which is being plotted for the elbow method? Why is it called the 'elbow' method?

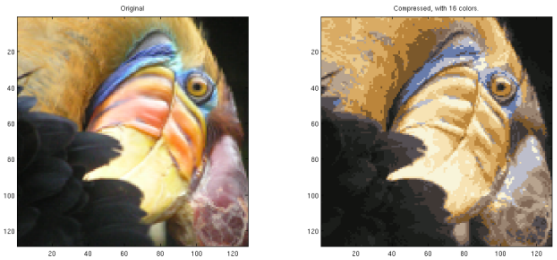
A typical plot looks like following,



The score is, in general, a measure of the input data on the k-means objective function i.e. **some form of intra-cluster distance relative to inner-cluster distance**.

10.1.2 Image compression with K-means

In a straightforward 24-bit color representation of an image, each pixel is repre- sented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green and blue intensity values. This encoding is often refered to as the RGB encoding. Our image contains thousands of colors, and in this part of the exercise, you will reduce the number of colors to 16 colors. By making this reduction, it is possible to represent (compress) the photo in an efficient way. Specifically, you only need to store the RGB values of the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits are necessary to represent 16 possibilities). In this exercise, you will use the K-means algorithm to select the 16 colors that will be used to represent the compressed image. Concretely, you will treat every pixel in the original image as a data example and use the K-means algorithm to find the 16 colors that best group (cluster) the pixels in the 3- dimensional RGB space. Once you have computed the cluster centroids on the image, you will then use the 16 colors to replace the pixels in the original image.



10.2 Anamoly Detection

Anomaly detection is the process of identifying unexpected items or events in data sets, which differ from the norm. And anomaly detection is often applied on unlabeled data which is known as unsupervised anomaly detection. Anomaly detection has two basic assumptions:

- Anomalies only occur very rarely in the data.
- Their features differ from the normal instances significantly.

```
function [mu sigma2] = estimateGaussian(X)
  %ESTIMATEGAUSSIAN This function estimates the parameters of a
  %Gaussian distribution using the data in X
  %   [mu sigma2] = estimateGaussian(X),
  %   The input X is the dataset with each n-dimensional data point in one
    row
  %   The output is an n-dimensional vector mu, the mean of the data set
  %   and the variances sigma^2, an n x 1 vector
  %

  % Useful variables
  [m, n] = size(X);

  % You should return these values correctly
  mu = zeros(n, 1);
  sigma2 = zeros(n, 1);

  mu = sum(X)'/m;

  mu_repeated = repmat(mu', m, 1);
  temp = X-mu_repeated;

  sigma2 = sum(temp.^2)'/m;
end

% =============================================

function [bestEpsilon bestF1] = selectThreshold(yval, pval)
  %SELECTTHRESHOLD Find the best threshold (epsilon) to use for selecting
  %outliers
  %   [bestEpsilon bestF1] = SELECTTHRESHOLD(yval, pval) finds the best
  %   threshold to use for selecting outliers based on the results from a
  %   validation set (pval) and the ground truth (yval).
  %

  bestEpsilon = 0;
  bestF1 = 0;
  F1 = 0;

  stepsize = (max(pval) - min(pval)) / 1000;
  for epsilon = min(pval):stepsize:max(pval)

      predictions = (pval < epsilon);
```

```
43
44      fp = sum((predictions==1) & (yval==0));
45      fn = sum((predictions==0) & (yval==1));
46      tp = sum((predictions==1) & (yval==1));
47      tn = sum((predictions==0) & (yval==0));
48
49      prec = tp / (tp + fp);
50      rec  = tp / (tp + fn);
51
52      F1 = 2*prec*rec/(prec+rec);
53
54
55      if F1 > bestF1
56         bestF1 = F1;
57         bestEpsilon = epsilon;
58      end
59    end
60
61  end
62  % =================================================
63
64  function [J, grad] = cofiCostFunc(params, Y, R, num_users, num_movies, ...
65                                    num_features, lambda)
66    %COFICOSTFUNC Collaborative filtering cost function
67    %   [J, grad] = COFICOSTFUNC(params, Y, R, num_users, num_movies, ...
68    %   num_features, lambda) returns the cost and gradient for the
69    %   collaborative filtering problem.
70    %
71
72    % Unfold the U and W matrices from params
73    X = reshape(params(1:num_movies*num_features), num_movies, num_features)
       ;
74    Theta = reshape(params(num_movies*num_features+1:end), ...
75                    num_users, num_features);
76
77
78    % You need to return the following values correctly
79    J = 0;
80    X_grad = zeros(size(X));
81    Theta_grad = zeros(size(Theta));
82
83    temp = X*Theta' - Y;
84
85
86    J = sum(sum(R.*(temp.^2))) + lambda*(sum(sum(X)) + sum(sum(Theta)));
87    J /= 2;
88
89    X_grad = ((R.*temp)*Theta);
90    Theta_grad = ((R.*temp)'*X);
91
92    R
93    X
94
95    % =============================================
```

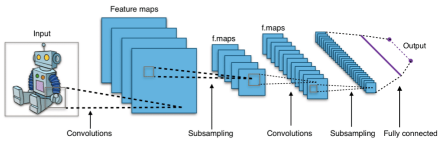
```
96
97    grad = [X_grad(:); Theta_grad(:)];
98
99  end
```

Convolutional neural network

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

The Convolutional Neural Networks enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing, etc.



11.1 CNNs vs Feed-forward Neural Nets!

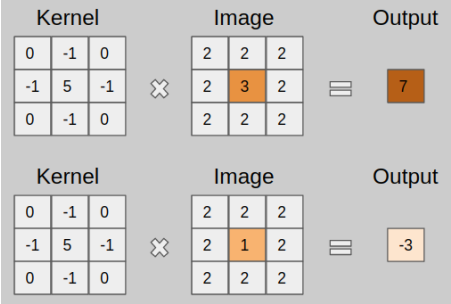
An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes?

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

11.2 Convolution Layer — The Kernel

Each convolutional layer contains a series of filters known as convolutional kernels. The filter is a matrix of integers that are used on a subset of the input pixel values, the same size as the kernel. Each pixel is multiplied by the corresponding value in the kernel, then the result is summed up for a single value for simplicity representing a grid cell, like a pixel, in the output channel/feature map. These are linear transformations, each convolution is a type of affine function.



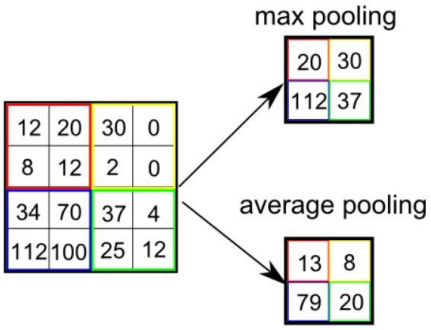
In computer vision the input is often a 3 channel RGB image. For simplicity, if we take a greyscale image that has one channel (a two dimensional matrix) and a 3x3 convolutional kernel (a two dimensional matrix). The kernel strides over the input matrix of numbers moving horizontally column by column, sliding/scanning over the first rows in the matrix containing the images pixel values. Then the kernel strides down vertically to subsequent rows. Note, the filter may stride over one or several pixels at a time, this is detailed further below. In other non-vision applications, a one dimensional convolution may slide vertically over an input matrix.

11.3 Pooling Layer

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average



Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

After going through the above process, we have successfully enabled the model to understand the features. Moving on, we are going to flatten the final output and feed it to a regular Neural Network for classification purposes.

PyTorch Implementations

12.1 Perceptrons

-

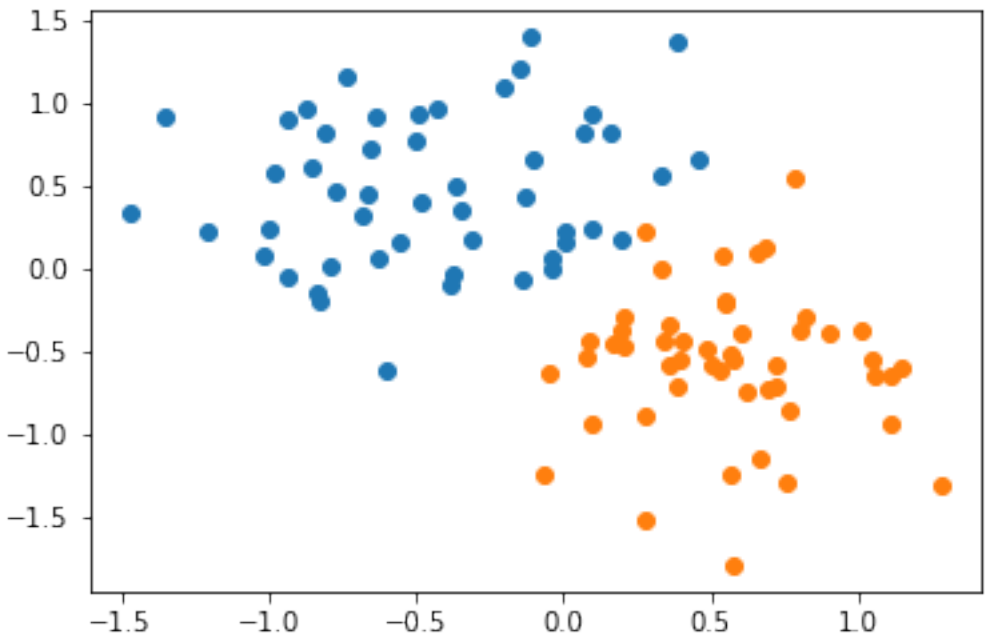
[1]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
from sklearn import datasets
```

[2]:
```
  n_pts = 100
  centers = [[-0.5, 0.5], [0.5, -0.5]]
  X, y = datasets.make_blobs(n_samples=n_pts, random_state=123, centers=centers,␣
 ↪cluster_std=0.4)

  x_data = torch.Tensor(X)
  y_data = torch.Tensor(y.reshape(100, 1))
  print(y.shape)
```

```
(100,)
```

[3]:
```
def scatter_plot():
    plt.scatter(X[y==0, 0], X[y==0, 1]) # Blue
    plt.scatter(X[y==1, 0], X[y==1, 1]) # Orange
```

[4]:
```
scatter_plot()
```



[5]:
```
class Model(nn.Module):
    def __init__(self, input_size, output_size):
      super().__init__()
      self.linear = nn.Linear(input_size, output_size)
    def forward(self, x):
      pred = torch.sigmoid(self.linear(x))
      return pred
    def predict(self, x):
      pred = self.forward(x)
      if pred >= 0.5:
        return 1
      else:
        return 0
```

[6]:
```
torch.manual_seed(2)
model = Model(2, 1)
print(list(model.parameters()))
```

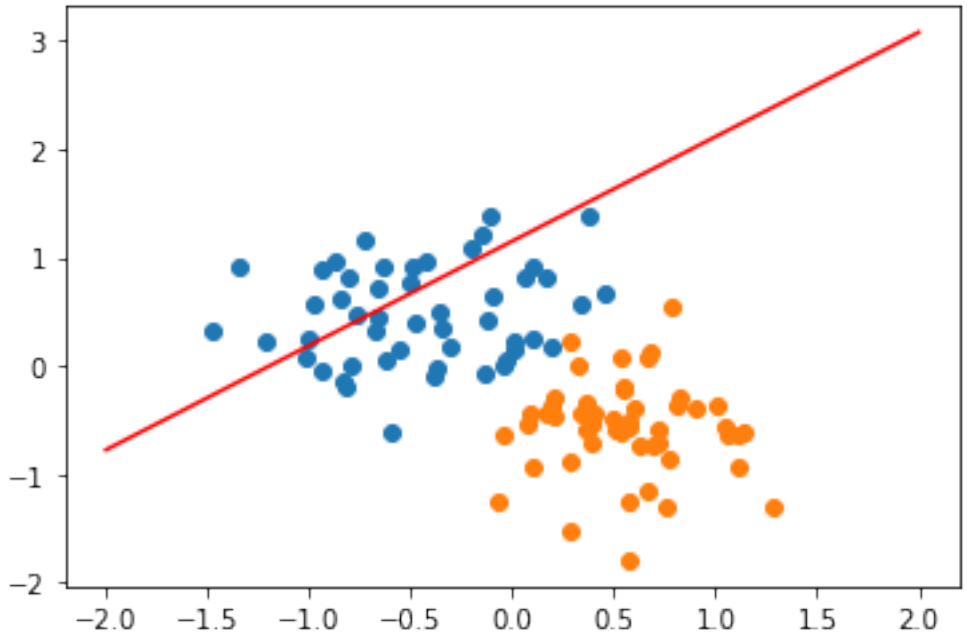
```
[Parameter containing:
tensor([[ 0.1622, -0.1683]], requires_grad=True), Parameter containing:
tensor([0.1939], requires_grad=True)]
```

[7]:
```
[w, b] = model.parameters()
w1, w2 = w.view(2)
def get_params():
```

```
  return (w1.item(), w2.item(), b[0].item())
```

[8]:
```
def plot_fit(title):
    plt.title = title
    w1, w2, b1 = get_params()
    x1 = np.array([-2.0, 2.0])
    x2 = -1*(w1*x1 + b1)/w2
    scatter_plot()
    plt.plot(x1, x2, 'r')
    plt.show()
```

[9]:
```
plot_fit('Initial Model')
```



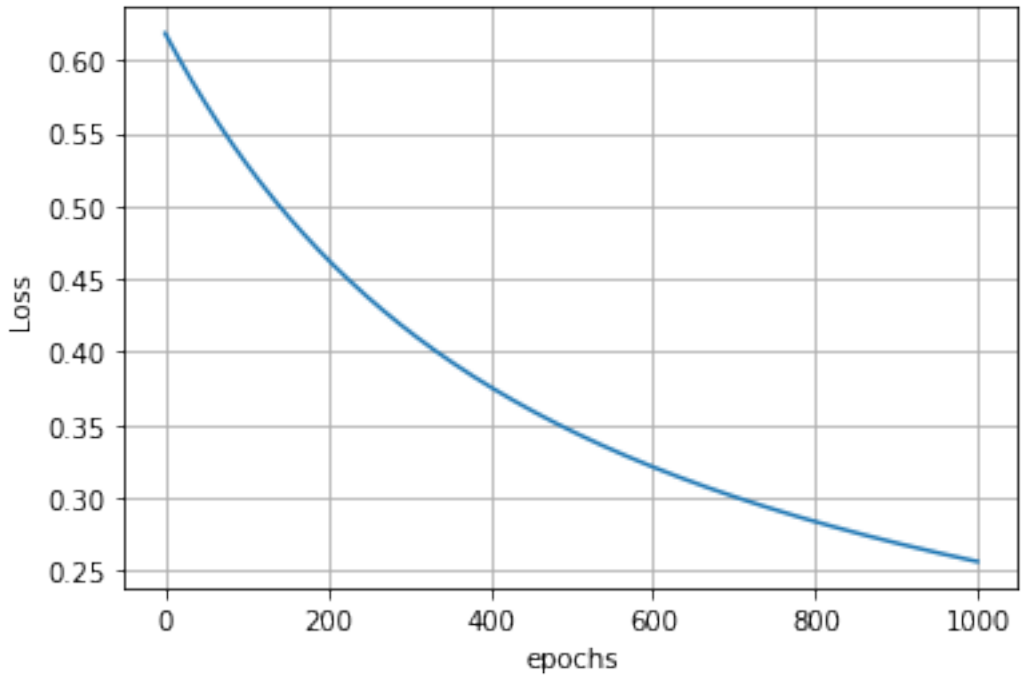
[10]:
```
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[11]:
```
epochs = 1000
losses = []

for i in range(epochs):
  y_pred = model.forward(x_data)
  loss = criterion(y_pred, y_data)
  print("epoch:", i, "loss:", loss.item())
  losses.append(loss.item())
  optimizer.zero_grad()
```

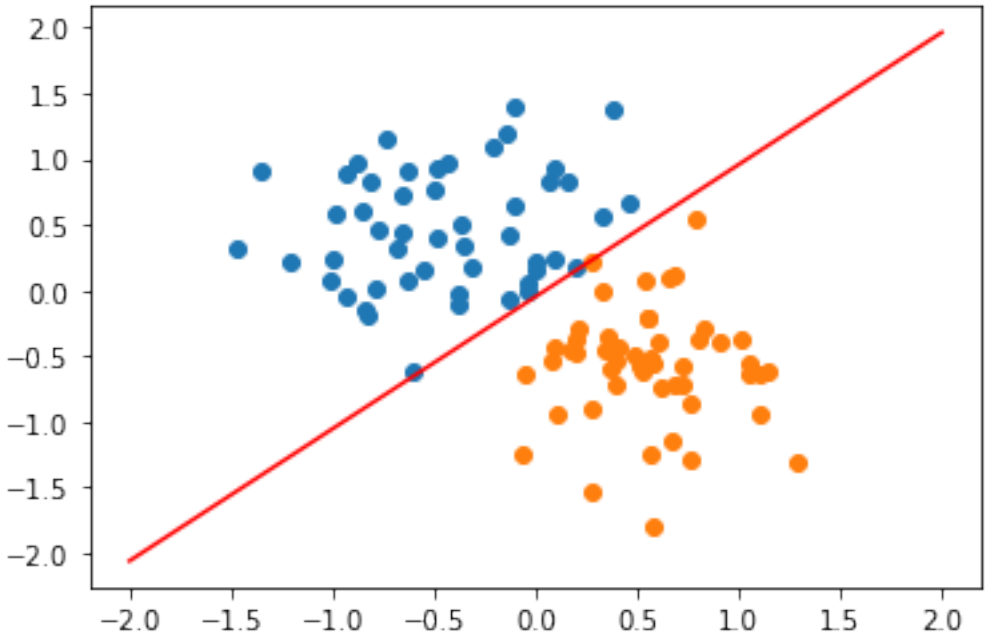
```
  loss.backward()
  optimizer.step()
```

```
epoch: 0 loss: 0.6185115575790405
epoch: 1 loss: 0.617464005947113
epoch: 2 loss: 0.6164200305938721
.
.
.
epoch: 999 loss: 0.2561521828174591
```

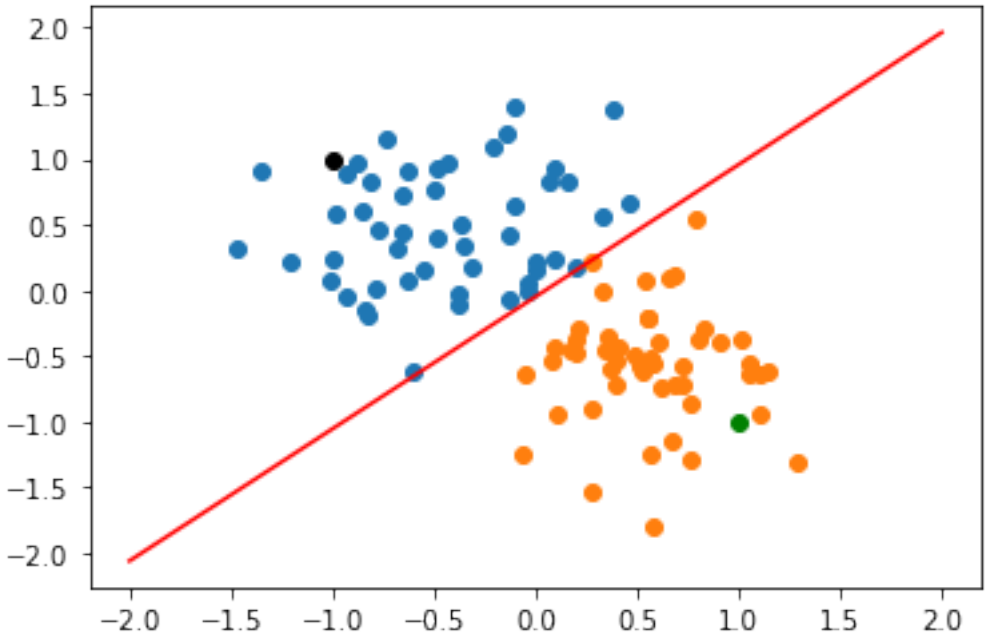
[12]:
```
plt.plot(range(epochs), losses)
plt.ylabel('Loss')
plt.xlabel('epochs')
plt.grid()
```



[13]:
```
plot_fit("Trained Model")
```



[17]:
```
point1 = torch.tensor([1.0, -1.0])
point2 = torch.tensor([-1.0, 1.0])
plt.plot(point1.numpy()[0], point1.numpy()[1], 'go')
plt.plot(point2.numpy()[0], point2.numpy()[1], 'ko')
plot_fit("Trained Model")

print("Red point positive probability = {}".format(model.forward(point1).item()))
print("Black point positive probability = {}".format(model.forward(point2).
 ↪item()))
print("Red point belongs in class {}".format(model.predict(point1)))
print("Black point belongs in class = {}".format(model.predict(point2)))
```



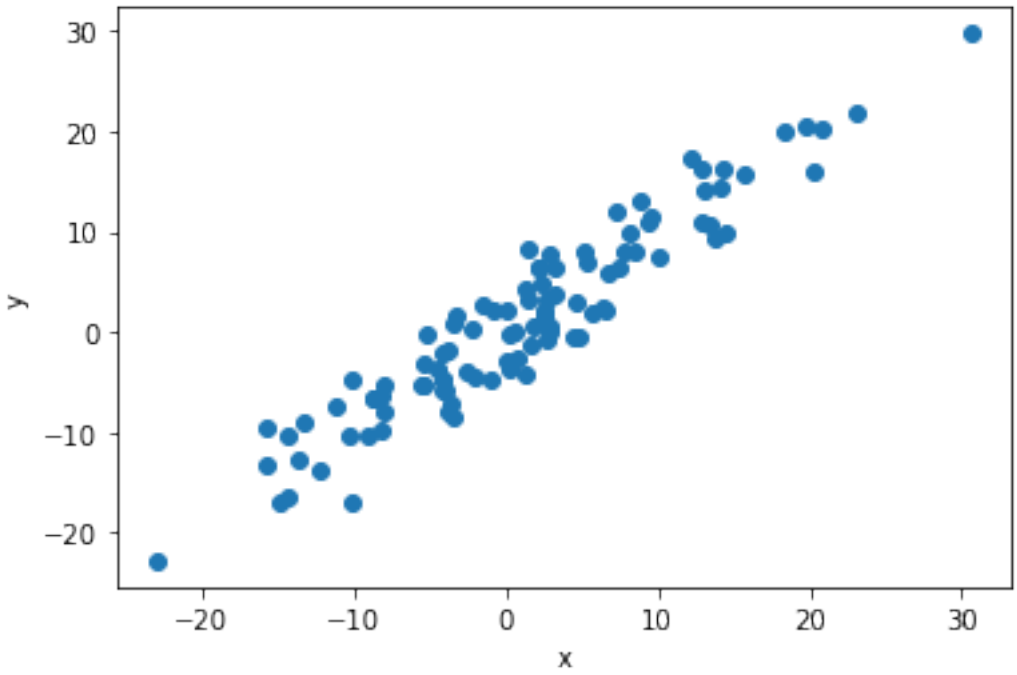

```
Red point positive probability = 0.9424158334732056
Black point positive probability = 0.05055497586727142
Red point belongs in class 1
Black point belongs in class = 0
```

12.2 Linear Regression

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

[2]:
```
X = torch.randn(100, 1)*10
y = X + 3*torch.randn(100, 1)
plt.plot(X.numpy(), y.numpy(), 'o')
plt.ylabel('y')
plt.xlabel('x')
```

[2]:
```
Text(0.5, 0, 'x')
```



[3]:
```
class LR(nn.Module):
  def __init__(self, input_size, output_size):
    super().__init__()
    self.linear = nn.Linear(input_size, output_size)
  def forward(self, x):
    pred = self.linear(x)
    return pred
```

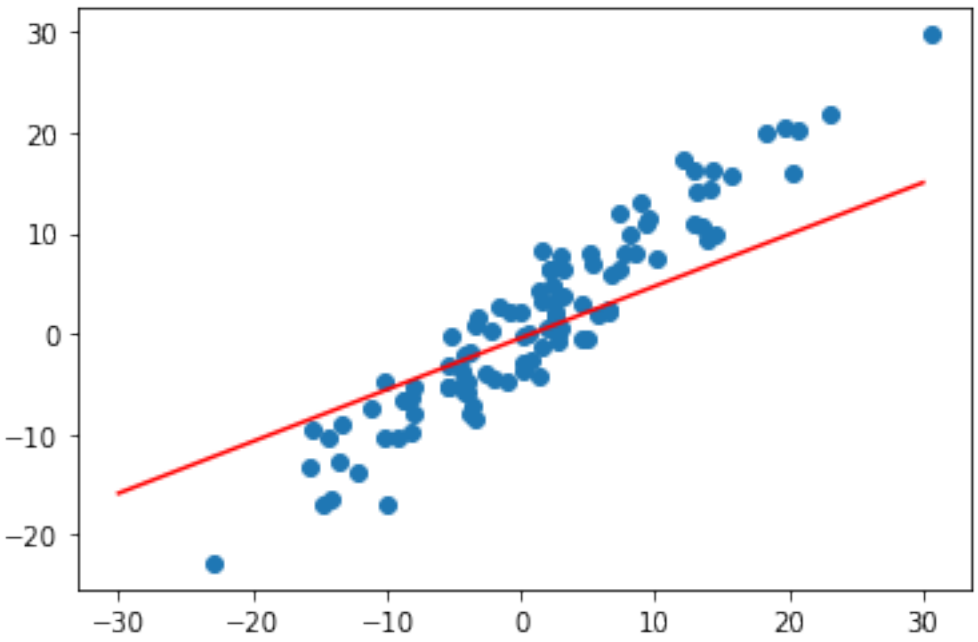
[4]:
```
torch.manual_seed(1)
model = LR(1, 1)
print(model)
```

```
LR(
  (linear): Linear(in_features=1, out_features=1, bias=True)
)
```

[5]:
```
[w,b] = model.parameters()

def get_params():
    return (w[0][0].item(), b[0].item())
```

[6]:
```
def plot_fit(title):
  plt.title = title
  w1, b1 = get_params()
  x1 = np.array([-30, 30])
  y1 = w1*x1 + b1
  plt.plot(x1, y1, 'r')
  plt.scatter(X, y)
  plt.show()
```

[7]:
```
plot_fit('Initial Model')
```



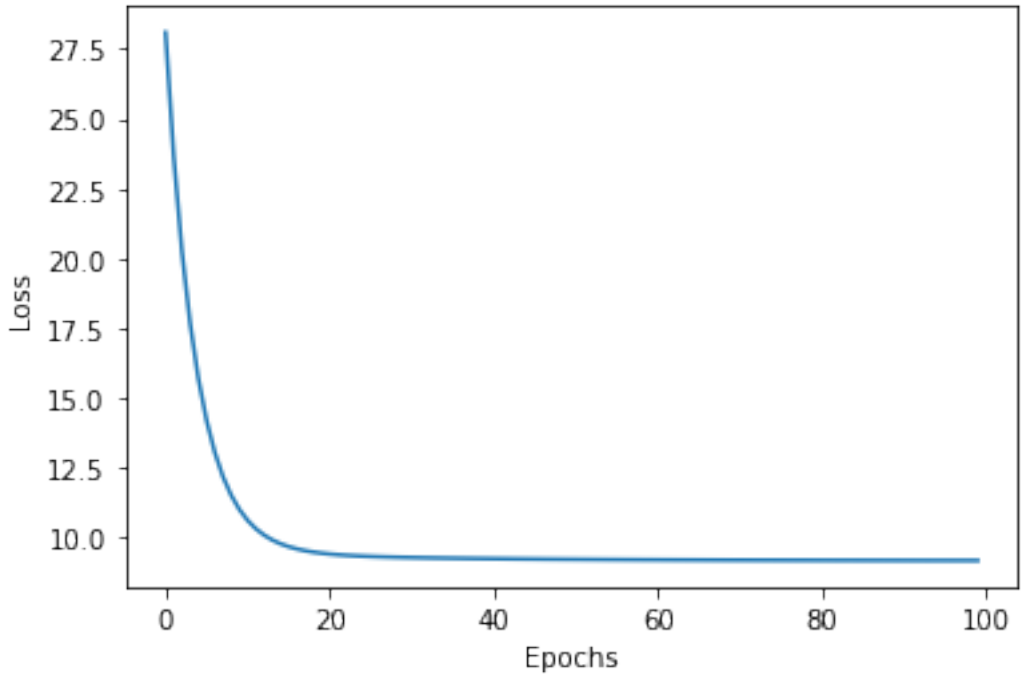
[8]:
```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
```

[9]:
```
epochs = 100
losses = []
for i in range(epochs):
  y_pred = model.forward(X)
  loss = criterion(y_pred, y)
  print("epoch:", i, "loss:", loss.item())

  losses.append(loss)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

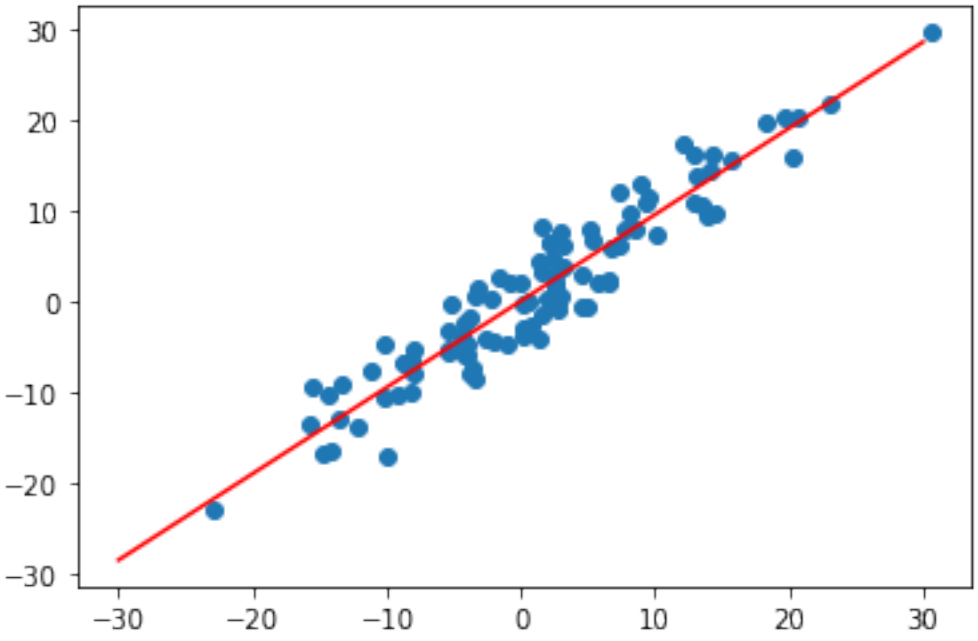
```
epoch: 0 loss: 28.098552703857422
epoch: 1 loss: 23.643329620361328
epoch: 2 loss: 20.250816345214844
.
.
.
epoch: 99 loss: 9.178099632263184
```

[10]:
```
plt.plot(range(epochs), losses)
plt.ylabel('Loss')
plt.xlabel('Epochs')
```

[10]:
```
Text(0.5, 0, 'Epochs')
```



[11]:
```
plot_fit("Trained Model")
```



12.3 Titanic Survival

Problem:

The sinking of the Titanic is one of the most infamous shipwrecks in history.

On April 15, 1912, during her maiden voyage, the widely considered "unsinkable" RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren't enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew.

While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others.

In this challenge, we ask you to build a predictive model that answers the question: "what sorts of people were more likely to survive?" using passenger data (ie name, age, gender, socio-economic class, etc).

[1]:
```
import numpy as np
import torch
import torch.nn as nn
import pandas as dd
import matplotlib.pyplot as plt
```

[2]:
```
device = torch.device("cuda:0")
```

[3]:
```
X = dd.read_csv('train.csv', encoding="UTF-8")
X = X.drop(['PassengerId', 'Survived', 'Ticket', 'Name', 'Cabin'], axis=1)

y = dd.read_csv('train.csv', usecols=['Survived'])
```

[4]:
```
X['Sex'] = X['Sex'].replace(['male', 'female'],[1,2])
X['Embarked'] = X['Embarked'].replace(['S', 'C', 'Q'],[1,2,3])
# X['Pclass'] = X['Pclass'].replace([2,3],[-1,0])
```

[5]:
```
X.isnull().any() # Tell whether any row contains NaNs or not
```

[5]:
```
Pclass      False
Sex         False
Age          True
SibSp       False
Parch       False
Fare        False
Embarked     True
dtype: bool
```

[6]:
```
X['Age'].fillna(X['Age'].mode()[0], inplace=True)  # replace NaN with average␣
↪age
X['Embarked'].fillna(X['Embarked'].mode()[0], inplace=True)
```

```
X.isnull().any()
```

[6]:
```
Pclass      False
Sex         False
Age         False
SibSp       False
Parch       False
Fare        False
Embarked    False
dtype: bool
```

[7]:
```
X
```

[7]:

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	3	1	22.0	1	0	7.2500	1.0
1	1	2	38.0	1	0	71.2833	2.0
2	3	2	26.0	0	0	7.9250	1.0
3	1	2	35.0	1	0	53.1000	1.0
4	3	1	35.0	0	0	8.0500	1.0
..	...	...	...	...	...	...	...
886	2	1	27.0	0	0	13.0000	1.0
887	1	2	19.0	0	0	30.0000	1.0
888	3	2	24.0	1	2	23.4500	1.0
889	1	1	26.0	0	0	30.0000	2.0
890	3	1	32.0	0	0	7.7500	3.0

[891 rows x 7 columns]

[8]:
```
# X['Age'] = (X['Age']-40)/40
# X['Fare'] = X['Fare']/512
# X
```

[9]:
```
y = torch.Tensor(y.values)
X = torch.Tensor(X.values)
```

[10]:
```
class Model(nn.Module):
    def __init__(self, input_size, output_size):
      super().__init__()
      self.linear = nn.Linear(input_size, output_size)
    def forward(self, x):
      pred = torch.sigmoid(self.linear(x))
      return pred
    def predict(self, x):
      pred = self.forward(x)
      if pred >= 0.5:
        return 1
      else:
```

```
        return 0
```

[11]:
```
model = Model(7, 1)

criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

[12]:
```
torch.manual_seed(2)
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[ 0.1296, -0.1197,  0.3220,  0.2237,  0.0826,  0.0652,  0.0791]],
       requires_grad=True), Parameter containing:
tensor([-0.0733], requires_grad=True)]
```

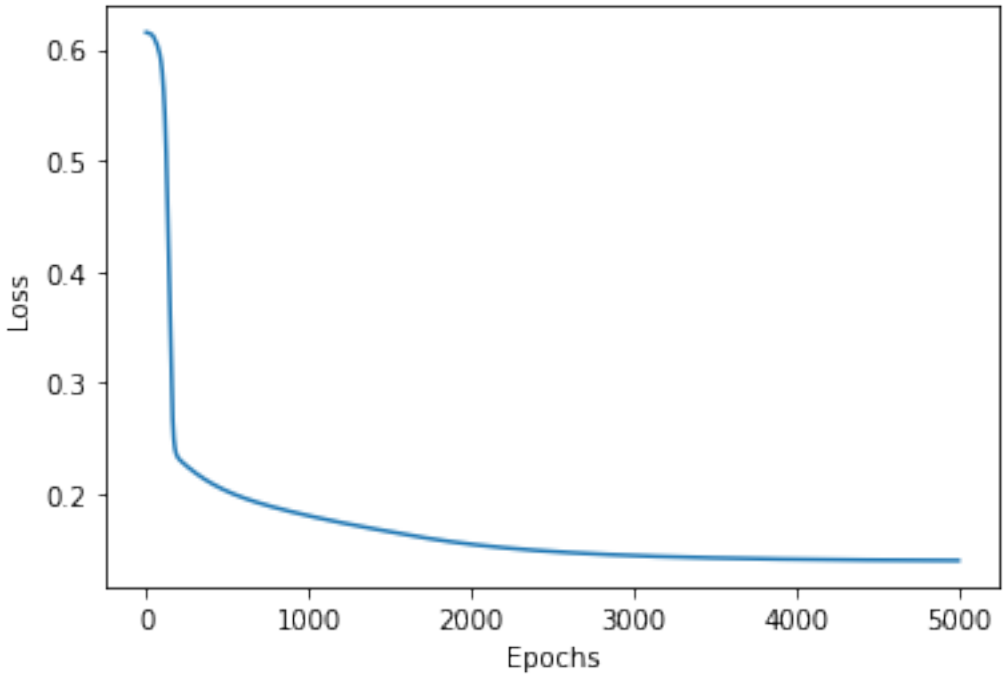
[13]:
```
epochs = 5000
losses = []
loss = 0

for i in range(epochs):
  y_pred = model.forward(X)
  loss = criterion(y_pred, y)
  print("epoch:", i, "loss:", loss.item())

  losses.append(loss)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

```
epoch: 0 loss: 0.6152137517929077
epoch: 1 loss: 0.6151852607727051
epoch: 2 loss: 0.615155816078186
epoch: 3 loss: 0.6151252388954163
epoch: 4 loss: 0.6150935292243958
epoch: 5 loss: 0.6150606274604797
.
.
.
epoch: 4998 loss: 0.13981899619102478
epoch: 4999 loss: 0.13981841504573822
```

[14]:
```
plt.plot(range(epochs), losses)
plt.ylabel('Loss')
plt.xlabel('Epochs')
```

```
    # print("Loss: {}".format(int(loss)))
```

[14]:
```
Text(0.5, 0, 'Epochs')
```



12.3.1 Testing

[15]:
```
X_test = dd.read_csv('test.csv', encoding="UTF-8")
X_test = X_test.drop(['PassengerId', 'Ticket', 'Name', 'Cabin'], axis=1)

pId = dd.read_csv('test.csv', encoding="UTF-8", usecols=['PassengerId'])

X_test['Sex'] = X_test['Sex'].replace(['male', 'female'],[1,2])
X_test['Embarked'] = X_test['Embarked'].replace(['S', 'C', 'Q'],[1,2,3])
```

[16]:
```
X_test['Age'].fillna(X_test['Age'].mode()[0], inplace=True)  # replace NaN␣
↪with average age
X_test['Embarked'].fillna(X_test['Embarked'].mode()[0], inplace=True)
X_test['Fare'].fillna(X_test['Fare'].mode()[0], inplace=True)

X_test.isnull().any()
```

[16]:
```
Pclass      False
Sex         False
```

```
Age         False
SibSp       False
Parch       False
Fare        False
Embarked    False
dtype: bool
```

[17]:
```
X_test.head()
# len(X_test)
```

[17]:

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	3	1	34.5	0	0	7.8292	3
1	3	2	47.0	1	0	7.0000	1
2	2	1	62.0	0	0	9.6875	3
3	3	1	27.0	0	0	8.6625	1
4	3	2	22.0	1	1	12.2875	1

[18]:
```
X_test = torch.Tensor(X_test.values)
pId = torch.Tensor(pId.values)
```

[19]:
```
predictions = [['PassengerId', 'Survived']]

i = 0
for x in X_test:
#     print(x)
    pred = model.predict(x)
    predictions.append([int(pId[i]), pred])
    i += 1

predictions
```

[19]:
```
[['PassengerId', 'Survived'],
 [892, 0],
 [893, 0],
 [894, 0],
 .
 .
 .
 [1309, 0]]
```

[20]:
```
import csv

with open('submission.csv', 'w') as submissionFile:
    writer = csv.writer(submissionFile)
    writer.writerows(predictions)
```

12.4 Deep Neural Network

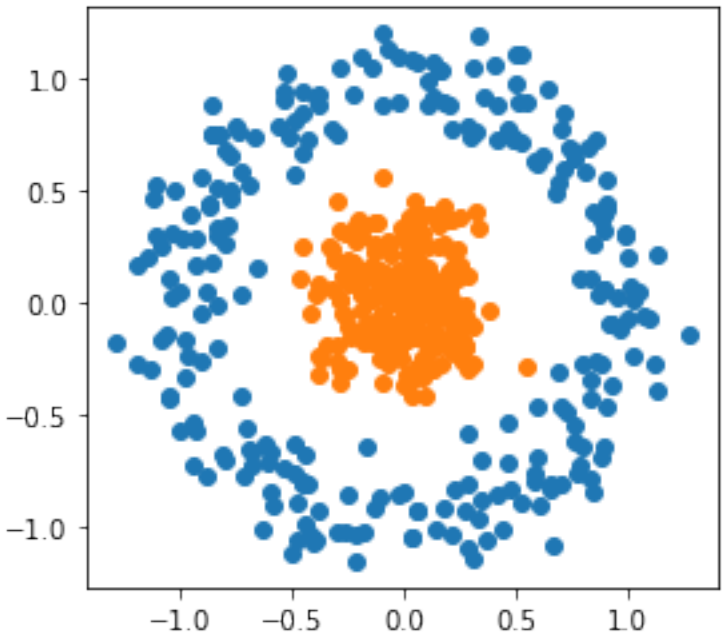
[1]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
from sklearn import datasets
```

[35]:
```
n_pts = 500
X, y = datasets.make_circles(n_samples=n_pts, random_state=20, noise=0.13,␣
 ↪factor=0.2)
print(X.size, y.size)
x_data = torch.Tensor(X)
y_data = torch.Tensor(y.reshape(500, 1))
```

```
1000 500
```

[36]:
```
def scatter_plot():
    plt.scatter(X[y==0, 0], X[y==0, 1]) # Blue
    plt.scatter(X[y==1, 0], X[y==1, 1]) # Orange
    plt.axis('scaled')
```

[37]:
```
scatter_plot()
```



[23]:
```
class Model(nn.Module):
  def __init__(self, input_size, H1, H2, H3, output_size):
    super().__init__()
    self.linear = nn.Linear(input_size, H1)
    self.linear2 = nn.Linear(H1, H2)
    self.linear3 = nn.Linear(H2, H3)
    self.linear4 = nn.Linear(H3, output_size)
  def forward(self, x):
    x = torch.sigmoid(self.linear(x))
    x = torch.sigmoid(self.linear2(x))
    x = torch.sigmoid(self.linear3(x))
    x = torch.sigmoid(self.linear4(x))
    return x
  def predict(self, x):
    pred = self.forward(x)
    if pred >= 0.5:
      return 1
    else:
      return 0
```

[24]:
```
torch.manual_seed(2)
model = Model(2, 4, 4, 3, 1)
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[ 0.1622, -0.1683],
        [ 0.1939, -0.0361],
        [ 0.3021,  0.1683],
        [-0.0813, -0.5717]], requires_grad=True), Parameter containing:
tensor([ 0.1614, -0.6260,  0.0929,  0.0470], requires_grad=True), Parameter
containing:
tensor([[-0.1099,  0.4088,  0.0334,  0.2073],
        [ 0.2116, -0.2950, -0.1922,  0.4809],
        [-0.4897, -0.0340, -0.0396,  0.3547],
        [-0.0475,  0.1317, -0.0240, -0.2800]], requires_grad=True), Parameter
containing:
tensor([-0.2834, -0.2429, -0.4542, -0.3245], requires_grad=True), Parameter
containing:
tensor([[ 0.1177,  0.3291,  0.0246, -0.2292],
        [ 0.2197, -0.1919, -0.1108, -0.2741],
        [-0.1570, -0.4633,  0.2133,  0.1944]], requires_grad=True), Parameter
containing:
tensor([0.0993, 0.2455, 0.2119], requires_grad=True), Parameter containing:
tensor([[0.0255, 0.0611, 0.0441]], requires_grad=True), Parameter containing:
tensor([0.3081], requires_grad=True)]
```

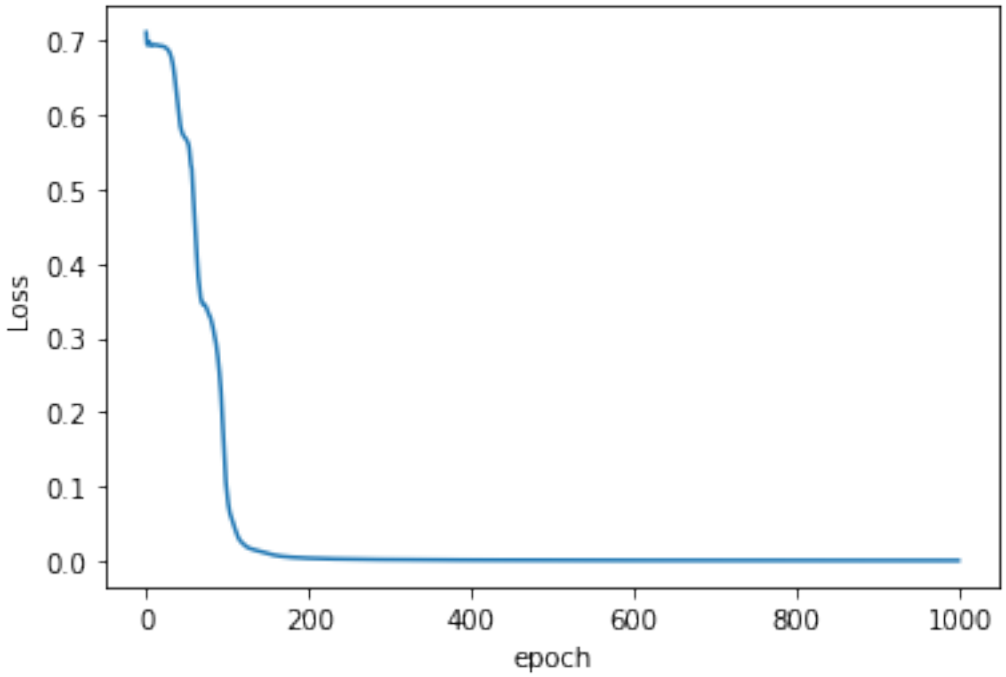
[25]:
```
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

[26]:
```
epochs = 1000
losses = []
for i in range(epochs):
  y_pred = model.forward(x_data)
  print(y_pred.size(), y_data.size())
  loss = criterion(y_pred, y_data)
  print("epoch:", i, "loss", loss.item())
  losses.append(loss.item())
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

```
torch.Size([500, 1]) torch.Size([500, 1])
epoch: 0 loss 0.7108473777770996
torch.Size([500, 1]) torch.Size([500, 1])
epoch: 1 loss 0.6952362060546875
torch.Size([500, 1]) torch.Size([500, 1])
.
.
.
torch.Size([500, 1]) torch.Size([500, 1])
epoch: 999 loss 0.00017163113807328045
```

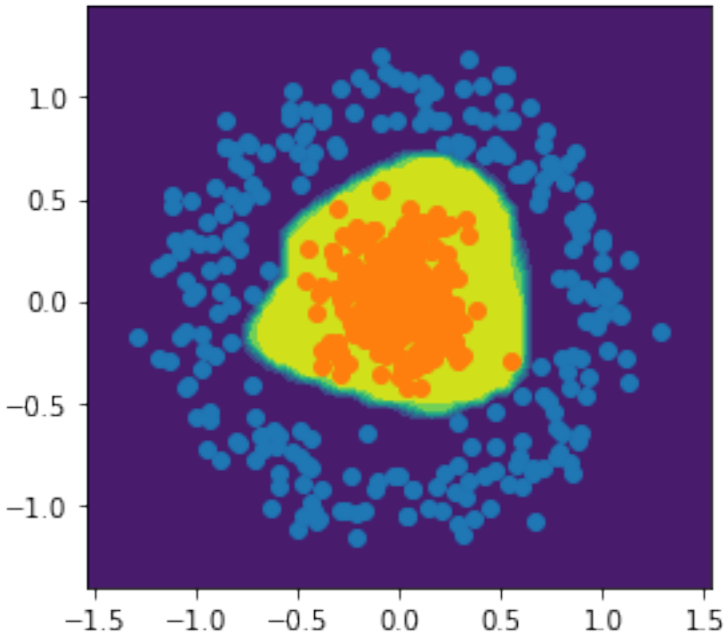
[27]:
```
plt.plot(range(epochs), losses)
plt.ylabel('Loss')
plt.xlabel('epoch')
```

[27]:
```
Text(0.5, 0, 'epoch')
```



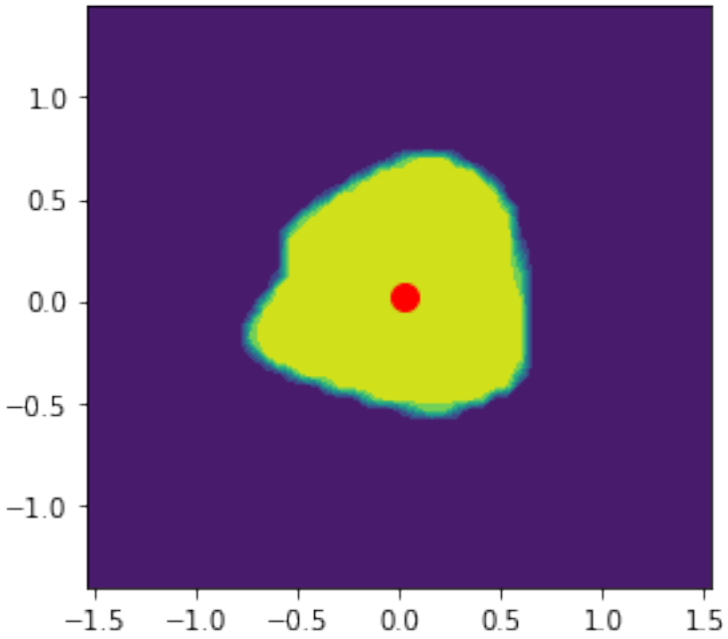
[28]:
```
def plot_decision_boundary(X, y):
    x_span = np.linspace(min(X[:,0])-0.25, max(X[:,0])+0.25) # Created equally␣
↪spaced elements in x_span (will use for plotting)
    y_span = np.linspace(min(X[:,1])-0.25, max(X[:,1])+0.25)
    # Created grid (returns two grid): https://www.geeksforgeeks.org/
↪numpy-meshgrid-function/
    xx, yy = np.meshgrid(x_span, y_span)
    grid = torch.Tensor(np.c_[xx.ravel(), yy.ravel()])
    pred_func = model.forward(grid)
    z = pred_func.view(xx.shape).detach().numpy()
    plt.contourf(xx, yy, z)
    plt.axis('scaled')
```

[29]:
```
plot_decision_boundary(X, y)
scatter_plot()
```



[30]:
```
x = 0.025
y = 0.025
point = torch.Tensor([x, y])
prediction = model.predict(point)
plt.plot([x], [y], marker='o', markersize=10, color="red")
print("Prediction is", prediction)
plot_decision_boundary(X, y)
```

```
Prediction is 1
```


1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

12.5 Convolutional Neural Network

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

12.5.1 Convolutional kernels

Each convolutional layer contains a series of filters known as convolutional kernels. The filter is a matrix of integers that are used on a subset of the input pixel values, the same size as the kernel. Each pixel is multiplied by the corresponding value in the kernel, then the result is summed up for a single value for simplicity representing a grid cell, like a pixel, in the output channel/feature map.

These are linear transformations, each convolution is a type of affine function.

In computer vision the input is often a 3 channel RGB image. For simplicity, if we take a greyscale image that has one channel (a two dimensional matrix) and a 3x3 convolutional kernel (a two dimensional matrix). The kernel strides over the input matrix of numbers moving horizontally column by column, sliding/scanning over the first rows in the matrix containing the images pixel values. Then the kernel strides down vertically to subsequent rows. Note, the filter may stride over one or several pixels at a time, this is detailed further below.

In other non-vision applications, a one dimensional convolution may slide vertically over an input matrix.

[1]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F
from torch import nn
from torchvision import datasets, transforms
```

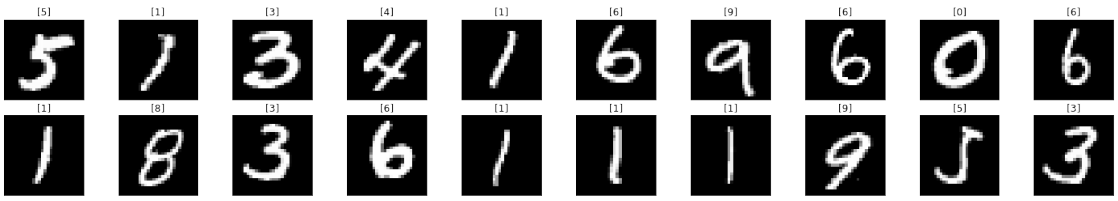
[2]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[3]:
```
transform = transforms.Compose([transforms.Resize((28,28)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))
                                ])
```

```
training_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
validation_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)

training_loader = torch.utils.data.DataLoader(training_dataset, batch_size=100,␣
 ↪shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size =␣
 ↪100, shuffle=False)
```

[4]:
```
def im_convert(tensor):
  image = tensor.clone().detach().numpy()
  image = image.transpose(1, 2, 0)
  image = image * np.array((0.5, 0.5, 0.5)) + np.array((0.5, 0.5, 0.5))
  image = image.clip(0, 1)
  return image
```

[5]:
```
dataiter = iter(training_loader)
images, labels = dataiter.next()
fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
  ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
  plt.imshow(im_convert(images[idx]))
  ax.set_title([labels[idx].item()])
```



[6]:
```
class LeNet(nn.Module):
    def __init__(self):
      super().__init__()
      self.conv1 = nn.Conv2d(1, 20, 5, 1)
      self.conv2 = nn.Conv2d(20, 50, 5, 1)
      self.fc1 = nn.Linear(4*4*50, 500)
      self.dropout1 = nn.Dropout(0.5)
      self.fc2 = nn.Linear(500, 10)
    def forward(self, x):
      x = F.relu(self.conv1(x))
      x = F.max_pool2d(x, 2, 2)
```

```
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = self.fc2(x)
        return x
```

[7]:
```
model = LeNet().to(device)
model
```

[7]:
```
LeNet(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=800, out_features=500, bias=True)
  (dropout1): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=500, out_features=10, bias=True)
)
```

[8]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.0001)
```

[9]:
```
epochs = 15
running_loss_history = []
running_corrects_history = []
val_running_loss_history = []
val_running_corrects_history = []

for e in range(epochs):

  running_loss = 0.0
  running_corrects = 0.0
  val_running_loss = 0.0
  val_running_corrects = 0.0

  for inputs, labels in training_loader:
    inputs = inputs.to(device)
    labels = labels.to(device)
    outputs = model(inputs)
    loss = criterion(outputs, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    _, preds = torch.max(outputs, 1)
    running_loss += loss.item()
```

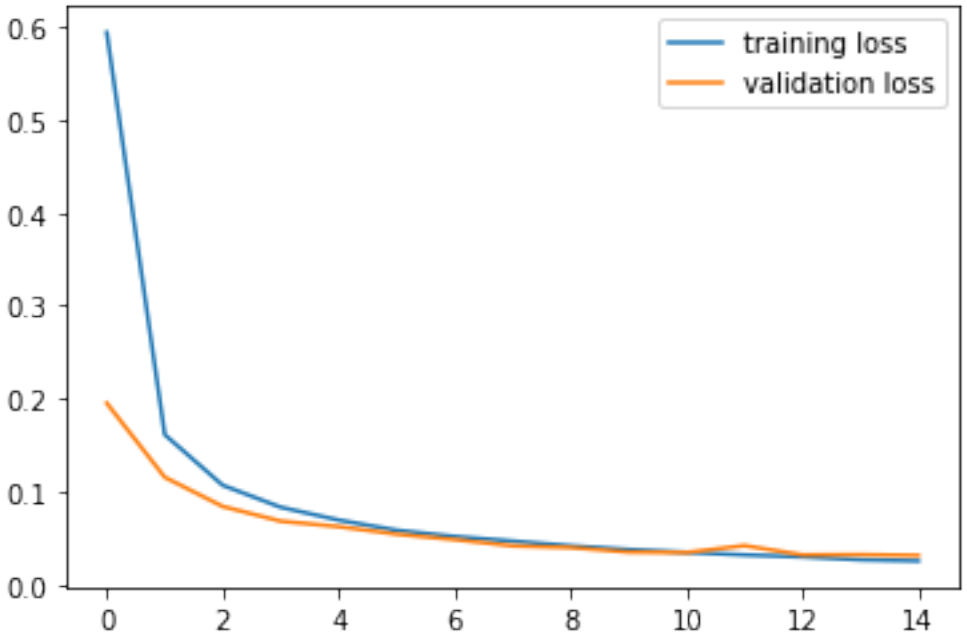
```
    running_corrects += torch.sum(preds == labels.data)

  else:
    with torch.no_grad():
      for val_inputs, val_labels in validation_loader:
        val_inputs = val_inputs.to(device)
        val_labels = val_labels.to(device)
        val_outputs = model(val_inputs)
        val_loss = criterion(val_outputs, val_labels)

        _, val_preds = torch.max(val_outputs, 1)
        val_running_loss += val_loss.item()
        val_running_corrects += torch.sum(val_preds == val_labels.data)

    epoch_loss = running_loss/len(training_loader)
    epoch_acc = running_corrects.float()/ len(training_loader)
    running_loss_history.append(epoch_loss)
    running_corrects_history.append(epoch_acc)

    val_epoch_loss = val_running_loss/len(validation_loader)
    val_epoch_acc = val_running_corrects.float()/ len(validation_loader)
    val_running_loss_history.append(val_epoch_loss)
    val_running_corrects_history.append(val_epoch_acc)
    print('epoch :', (e+1))
    print('training loss: {:.4f}, acc {:.4f} '.format(epoch_loss, epoch_acc.
↪item()))
    print('validation loss: {:.4f}, validation acc {:.4f} '.
↪format(val_epoch_loss, val_epoch_acc.item()))
```

```
epoch : 1
training loss: 0.5943, acc 84.0750
validation loss: 0.1951, validation acc 94.3600
epoch : 2
training loss: 0.1612, acc 95.1967
validation loss: 0.1153, validation acc 96.5300
epoch : 3
training loss: 0.1067, acc 96.8667
validation loss: 0.0839, validation acc 97.4500
epoch : 4
training loss: 0.0830, acc 97.4717
validation loss: 0.0680, validation acc 98.0100
epoch : 5
training loss: 0.0692, acc 97.9200
validation loss: 0.0622, validation acc 98.0100
epoch : 6
training loss: 0.0583, acc 98.2800
validation loss: 0.0541, validation acc 98.3700
```

```
epoch : 7
training loss: 0.0515, acc 98.4550
validation loss: 0.0482, validation acc 98.5200
epoch : 8
training loss: 0.0466, acc 98.5433
validation loss: 0.0415, validation acc 98.6800
epoch : 9
training loss: 0.0415, acc 98.7550
validation loss: 0.0396, validation acc 98.6600
epoch : 10
training loss: 0.0375, acc 98.8550
validation loss: 0.0348, validation acc 98.8900
epoch : 11
training loss: 0.0345, acc 98.9500
validation loss: 0.0343, validation acc 98.9000
epoch : 12
training loss: 0.0317, acc 99.0300
validation loss: 0.0417, validation acc 98.8100
epoch : 13
training loss: 0.0296, acc 99.1017
validation loss: 0.0317, validation acc 99.0000
epoch : 14
training loss: 0.0265, acc 99.1817
validation loss: 0.0320, validation acc 99.0600
epoch : 15
training loss: 0.0253, acc 99.2200
validation loss: 0.0311, validation acc 99.0500
```

[10]:
```
plt.plot(running_loss_history, label = 'training loss')
plt.plot(val_running_loss_history, label = 'validation loss')
plt.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7ffaa4a62430>
```



[11]:
```
plt.plot(running_corrects_history, label = 'training accuracy')
plt.plot(val_running_corrects_history, label = 'validation accuracy')
plt.legend()
```

[11]: <matplotlib.legend.Legend at 0x7ffaa560fbe0>

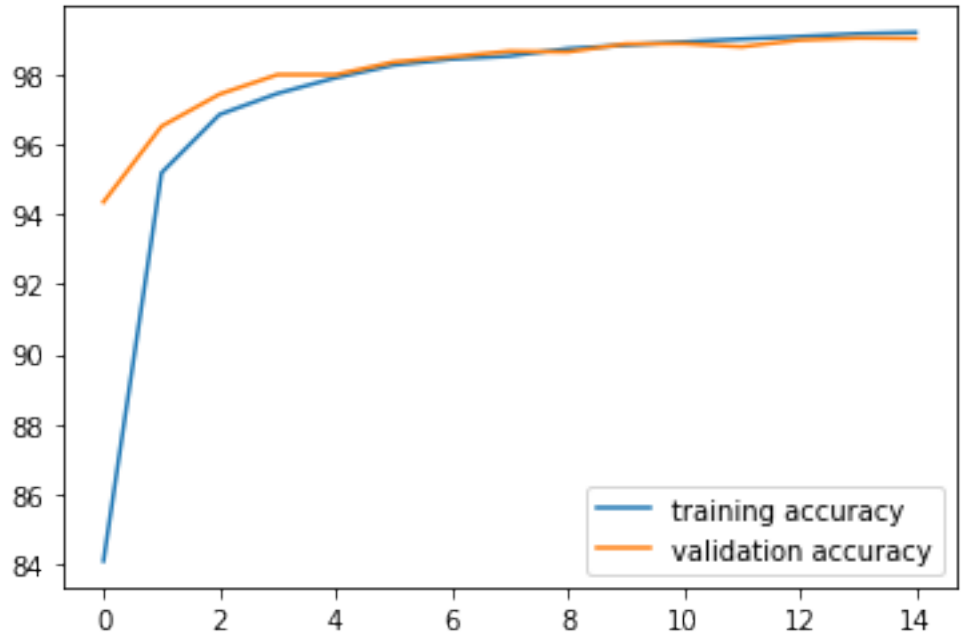
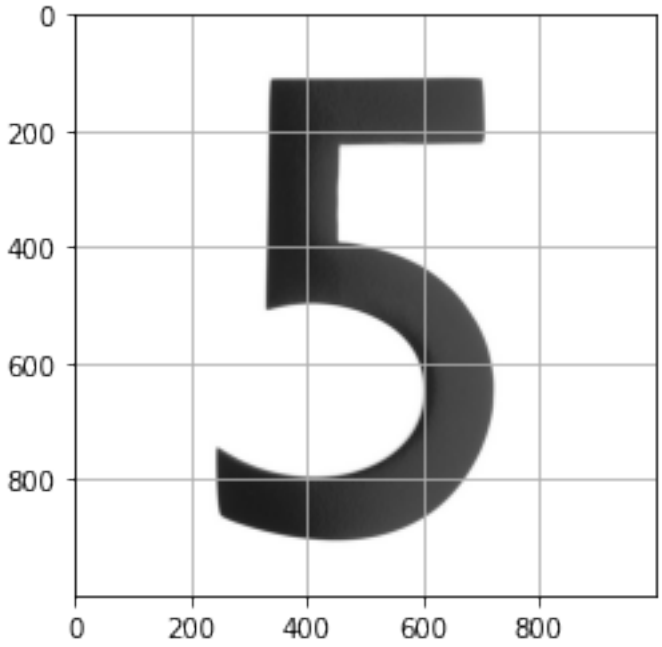
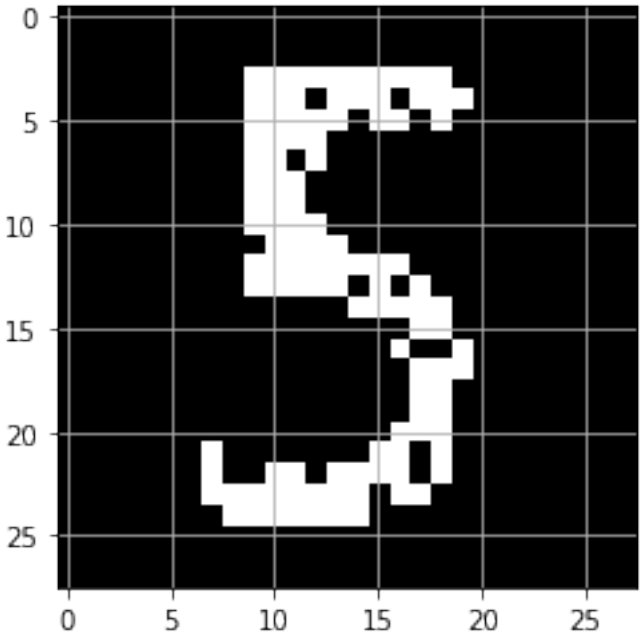


Image Link - These images will be used for further demonstations

[16]:
```
import requests
import PIL
from PIL import Image

url = 'https://images.homedepot-static.com/productImages/
 ↪007164ea-d47e-4f66-8d8c-fd9f621984a2/svn/
 ↪architectural-mailboxes-house-letters-numbers-3585b-5-64_1000.jpg'
response = requests.get(url, stream = True)
img = Image.open(response.raw)
plt.imshow(img)
plt.grid()
```



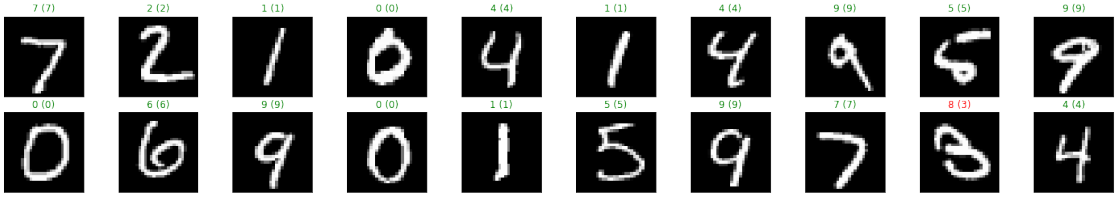
[17]:
```
img = PIL.ImageOps.invert(img)
img = img.convert('1')
img = transform(img)
plt.imshow(im_convert(img))
plt.grid()
```



[18]:
```
images = img.to(device)
image = images[0].unsqueeze(0).unsqueeze(0)
output = model(image)
_, pred = torch.max(output, 1)
print(pred.item())
```

```
5
```

[19]:
```
dataiter = iter(validation_loader)
images, labels = dataiter.next()
images = images.to(device)
labels = labels.to(device)
output = model(images)
_, preds = torch.max(output, 1)

fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
  ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
  plt.imshow(im_convert(images[idx]))
  ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].
 ↪item())), color=("green" if preds[idx]==labels[idx] else "red"))
```


7 (7)
2 (2)
1 (1)
0 (0)
4 (4)
1 (1)
4 (4)
9 (9)
5 (5)
9 (9)
0 (0)
6 (6)
9 (9)
0 (0)
1 (1)
5 (5)
9 (9)
7 (7)
8 (3)
4 (4)

12.6 Style Transfer

[0]:
```
!pip install torch torchvision
!pip install Pillow==4.0.0
```

[1]:
```
%matplotlib inline
import torch
import torch.optim as optim
from torchvision import transforms, models
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
vgg = models.vgg19(pretrained=True).features

for params in vgg.parameters():
    params.requires_grad_(False)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to
/home/sahasra/.cache/torch/checkpoints/vgg19-dcbb9e9d.pth

HBox(children=(FloatProgress(value=0.0, max=574673361.0), HTML(value='')))
```

[3]:
```
device = torch.device('CUDA' if torch.cuda.is_available() else 'cpu')
vgg.to(device)
```

[3]:
```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
```

```
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace=True)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace=True)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace=True)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace=True)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
```

[4]:
```python
def load_image(img_path, max_size=400, shape=None):

    image = Image.open(img_path).convert('RGB')
    if max(image.size) > max_size:
        size = max_size
    else:
        size = max(image.size)

    if shape is not None:
        size = shape

    in_transform = transforms.Compose([
                        transforms.Resize(size),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5, 0.5, 0.5),
                                             (0.5, 0.5, 0.5))])

    image = in_transform(image).unsqueeze(0)
```

```
    return image
```

[6]:
```
content = load_image('./data/styleTransfer-1/Images/City.jpg').to(device)
style = load_image('./data/styleTransfer-1/Images/StarryNight.jpg',␣
 ↪shape=content.shape[-2:]).to(device)
```

[7]:
```
def im_convert(tensor):
    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.5, 0.5, 0.5)) + np.array((0.5, 0.5, 0.5))
    image = image.clip(0, 1)

    return image
```

[8]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.imshow(im_convert(content))
ax1.axis('off')
ax2.imshow(im_convert(style))
ax2.axis('off')
```

[8]:
```
(-0.5, 599.5, 399.5, -0.5)
```

[9]:
```
def get_features(image, model):

    layers = {'0': 'conv1_1',
              '5': 'conv2_1',
              '10': 'conv3_1',
              '19': 'conv4_1',
              '21': 'conv4_2',  # Content Extraction
              '28': 'conv5_1'}

    features = {}
```

```
    for name, layer in model._modules.items():
        image = layer(image)
        if name in layers:
            features[layers[name]] = image

    return features
```

[10]:
```
content_features = get_features(content, vgg)
style_features = get_features(style, vgg)
```

[11]:
```
def gram_matrix(tensor):
  _, d, h, w = tensor.size()
  tensor = tensor.view(d, h * w)
  gram = torch.mm(tensor, tensor.t())
  return gram
```

[12]:
```
style_grams = {layer: gram_matrix(style_features[layer]) for layer in␣
 ↪style_features}
```

[13]:
```
style_weights = {'conv1_1': 1.,
                 'conv2_1': 0.75,
                 'conv3_1': 0.2,
                 'conv4_1': 0.2,
                 'conv5_1': 0.2}

content_weight = 1  # alpha
style_weight = 1e6  # beta
```

[14]:
```
target = content.clone().requires_grad_(True).to(device)
```

[15]:
```
show_every = 300
optimizer = optim.Adam([target], lr=0.003)
steps = 2100

height, width, channels = im_convert(target).shape
image_array = np.empty(shape=(300, height, width, channels))
capture_frame = steps/300
counter = 0
```

[17]:
```
for ii in range(1, steps+1):
  target_features = get_features(target, vgg)
  content_loss = torch.mean((target_features['conv4_2'] -␣
 ↪content_features['conv4_2'])**2)
  style_loss = 0

  for layer in style_weights:
```

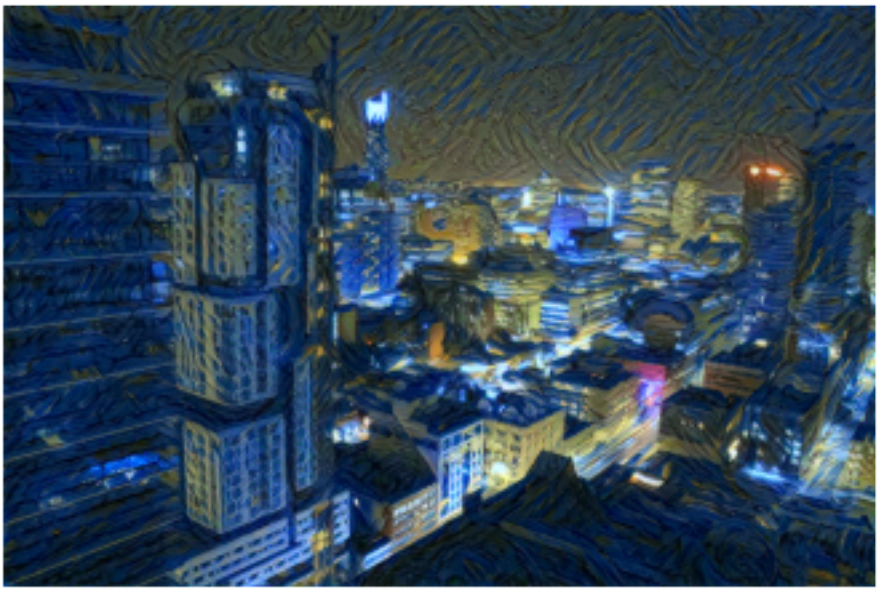
```
    target_feature = target_features[layer]
    target_gram = gram_matrix(target_feature)
    style_gram = style_grams[layer]
    layer_style_loss = style_weights[layer] * torch.mean((target_gram -␣
↪style_gram)**2)
    _, d, h, w = target_feature.shape
    style_loss += layer_style_loss / (d*h*w)

  total_loss = content_weight*content_loss + style_weight*style_loss

  optimizer.zero_grad()
  total_loss.backward()
  optimizer.step()

  if ii % show_every == 0:
    print('Total loss: ', total_loss.item())
    print('Iteration: ', ii)
    plt.imshow(im_convert(target))
    plt.axis("off")
    plt.show()

  if ii % capture_frame == 0:
    image_array[counter] = im_convert(target)
    counter = counter + 1
```

```
Total loss:  932833.5
Iteration:  300
```

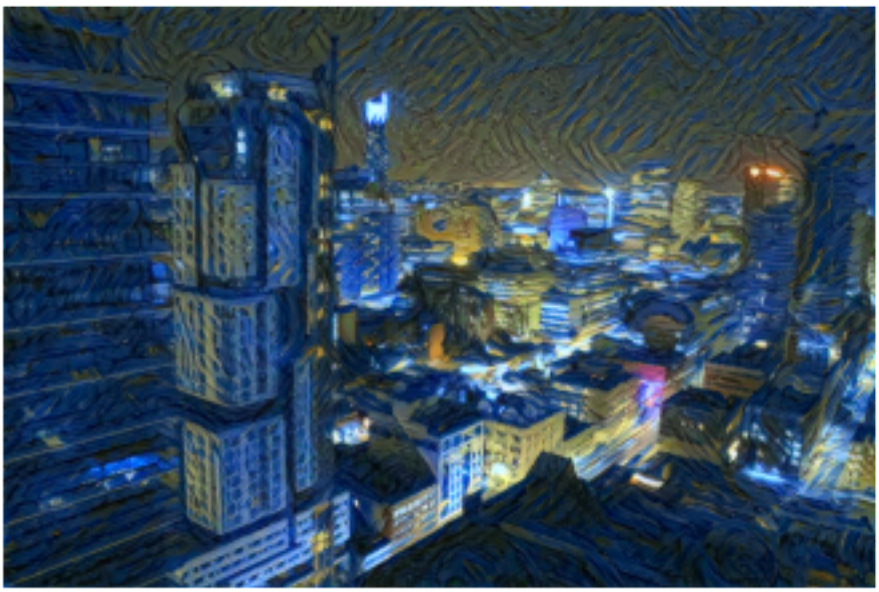
```
Total loss:  651102.25
Iteration:  600
```

```
Total loss:  504280.59375
Iteration:  900
```

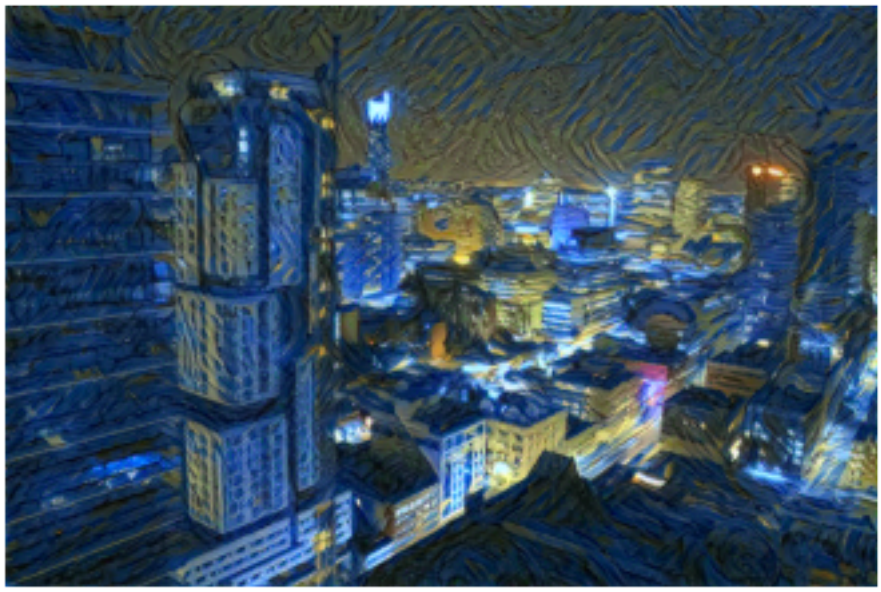
```
Total loss:  395064.53125
Iteration:  1200
```



```
Total loss:  309852.34375
Iteration:  1500
```



```
Total loss:  242368.46875
Iteration:  1800
```



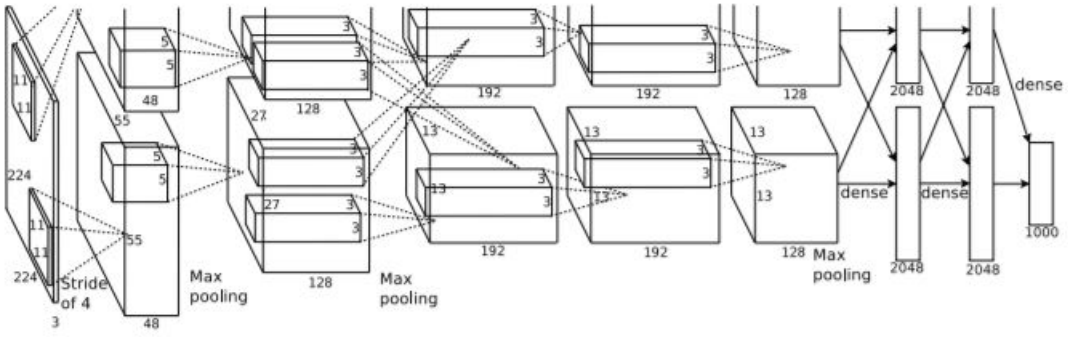
[18]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 10))
ax1.imshow(im_convert(content))
ax1.axis('off')
ax2.imshow(im_convert(style))
ax2.axis('off')
ax3.imshow(im_convert(target))
ax3.axis('off')
```

[18]: (-0.5, 599.5, 399.5, -0.5)

12.7 Transfer Learning: AlexNet

AlexNet is the name of a convolutional neural network (CNN), designed by Alex Krizhevsky, and published with Ilya Sutskever and Krizhevsky's doctoral advisor Geoffrey Hinton.

AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge on September 30, 2012. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up. The original paper's primary result was that the depth of the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training.



[1]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F
from torch import nn
from torchvision import datasets, transforms, models
```

[2]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

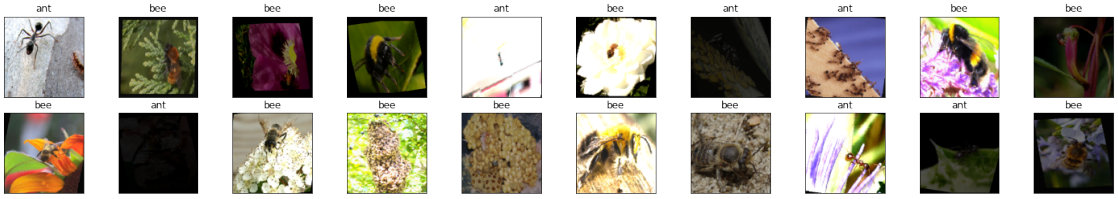
[3]:
```
transform_train = transforms.Compose([transforms.Resize((224,224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(10),
                                      transforms.RandomAffine(0, shear=10,␣
 ↪scale=(0.8,1.2)),
                                      transforms.ColorJitter(brightness=1,␣
 ↪contrast=0.2, saturation=0.2),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.
 ↪5, 0.5, 0.5))
                                     ])


transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor(),
```

```
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,␣
 ↪0.5))
                                ])
training_dataset = datasets.ImageFolder('data/ants_and_bees/train',␣
 ↪transform=transform_train)
validation_dataset = datasets.ImageFolder('data/ants_and_bees/val',␣
 ↪transform=transform)

training_loader = torch.utils.data.DataLoader(training_dataset, batch_size=100,␣
 ↪shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size =␣
 ↪100, shuffle=False)
```

[4]:
```
def im_convert(tensor):
  image = tensor.clone().detach().numpy()
  image = image.transpose(1, 2, 0)
  image = image * np.array((0.5, 0.5, 0.5)) + np.array((0.5, 0.5, 0.5))
  image = image.clip(0, 1)
  return image
```

[5]:
```
classes = ('ant', 'bee')
```

[6]:
```
dataiter = iter(training_loader)
images, labels = dataiter.next()
fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
  ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
  plt.imshow(im_convert(images[idx]))
  ax.set_title(classes[labels[idx].item()])
```



[7]:
```
model = models.alexnet(pretrained=True) # try vgg16 and reduce no. of epoch to 5
```

[8]:
```
print(model)
```

```
AlexNet(
  (features): Sequential(
```

```
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

[9]:
```
for param in model.features.parameters():
    param.require_grad = False
```

[10]:
```
import torch.nn as nn
```

[11]:
```
n_inputs = model.classifier[6].in_features
last_layer = nn.Linear(n_inputs, len(classes))
model.classifier[6] = last_layer
model.to(device)
```

[11]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
```

```
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
)
```

[12]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.0001)
```

[13]:
```
epochs = 10
running_loss_history = []
running_corrects_history = []
val_running_loss_history = []
val_running_corrects_history = []

for e in range(epochs):

  running_loss = 0.0
  running_corrects = 0.0
  val_running_loss = 0.0
  val_running_corrects = 0.0

  for inputs, labels in training_loader:
    inputs = inputs.to(device)
    labels = labels.to(device)
    outputs = model(inputs)
    loss = criterion(outputs, labels)

    optimizer.zero_grad()
    loss.backward()
```

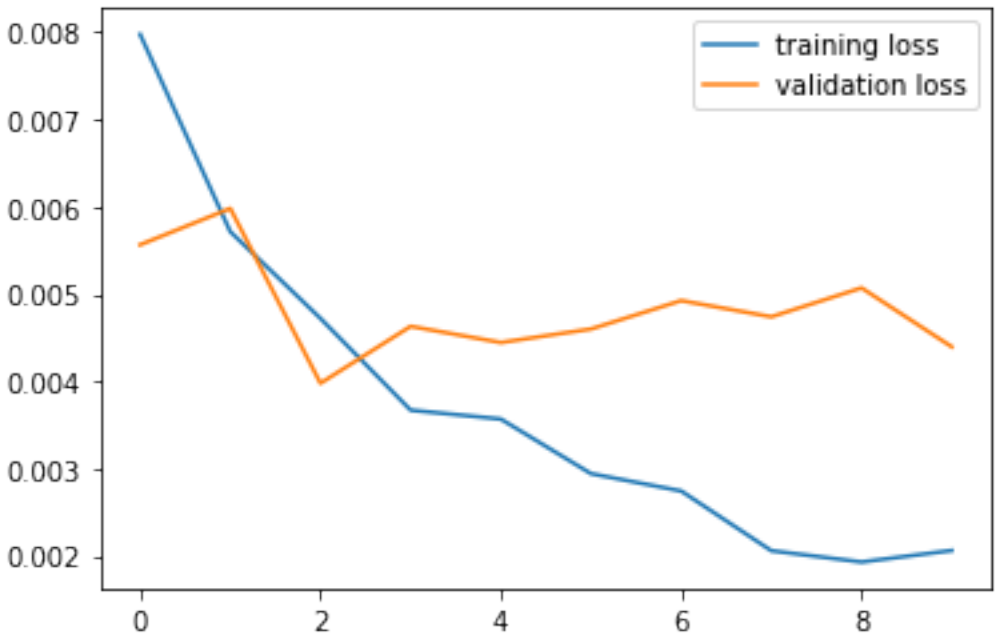
```
    optimizer.step()

    _, preds = torch.max(outputs, 1)
    running_loss += loss.item()
    running_corrects += torch.sum(preds == labels.data)

  else:
    with torch.no_grad():
      for val_inputs, val_labels in validation_loader:
        val_inputs = val_inputs.to(device)
        val_labels = val_labels.to(device)
        val_outputs = model(val_inputs)
        val_loss = criterion(val_outputs, val_labels)

        _, val_preds = torch.max(val_outputs, 1)
        val_running_loss += val_loss.item()
        val_running_corrects += torch.sum(val_preds == val_labels.data)

    epoch_loss = running_loss/len(training_loader.dataset)
    epoch_acc = running_corrects.float()/ len(training_loader.dataset)
    running_loss_history.append(epoch_loss)
    running_corrects_history.append(epoch_acc)

    val_epoch_loss = val_running_loss/len(validation_loader.dataset)
    val_epoch_acc = val_running_corrects.float()/ len(validation_loader.dataset)
    val_running_loss_history.append(val_epoch_loss)
    val_running_corrects_history.append(val_epoch_acc)
    print('epoch :', (e+1))
    print('training loss: {:.4f}, acc {:.4f} '.format(epoch_loss, epoch_acc.
↪item()))
    print('validation loss: {:.4f}, validation acc {:.4f} '.
↪format(val_epoch_loss, val_epoch_acc.item()))
```

```
epoch : 1
training loss: 0.0080, acc 0.5574
validation loss: 0.0056, validation acc 0.7451
epoch : 2
training loss: 0.0057, acc 0.7705
validation loss: 0.0060, validation acc 0.8366
epoch : 3
training loss: 0.0047, acc 0.8156
validation loss: 0.0040, validation acc 0.8758
epoch : 4
training loss: 0.0037, acc 0.8484
validation loss: 0.0046, validation acc 0.8627
epoch : 5
training loss: 0.0036, acc 0.8566
```

```
validation loss: 0.0044, validation acc 0.8954
epoch : 6
training loss: 0.0029, acc 0.8934
validation loss: 0.0046, validation acc 0.8889
epoch : 7
training loss: 0.0027, acc 0.9303
validation loss: 0.0049, validation acc 0.8758
epoch : 8
training loss: 0.0021, acc 0.9016
validation loss: 0.0047, validation acc 0.8954
epoch : 9
training loss: 0.0019, acc 0.9467
validation loss: 0.0051, validation acc 0.8824
epoch : 10
training loss: 0.0021, acc 0.9180
validation loss: 0.0044, validation acc 0.8954
```

[14]:
```
plt.plot(running_loss_history, label = 'training loss')
plt.plot(val_running_loss_history, label = 'validation loss')
plt.legend()
```

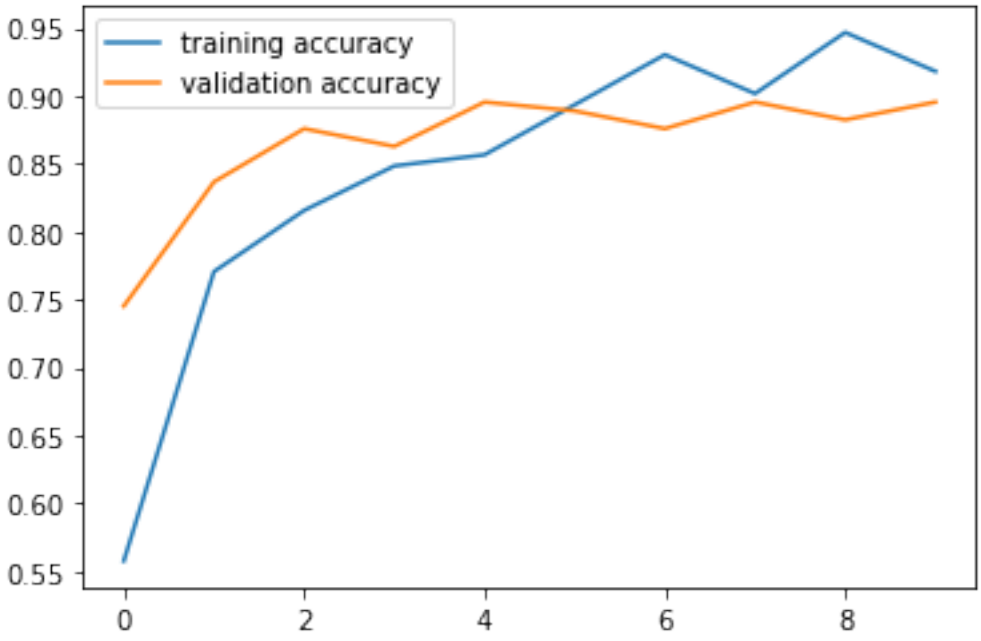
[14]: <matplotlib.legend.Legend at 0x7f677e887af0>



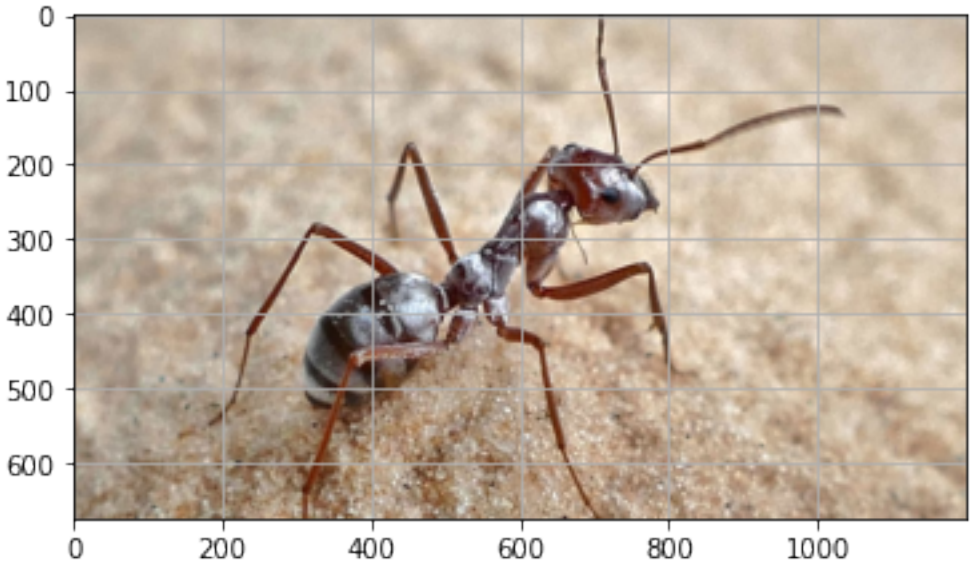
[15]:
```
plt.plot(running_corrects_history, label = 'training accuracy')
plt.plot(val_running_corrects_history, label = 'validation accuracy')
```

```
plt.legend()
```

[15]: <matplotlib.legend.Legend at 0x7f67925d0250>

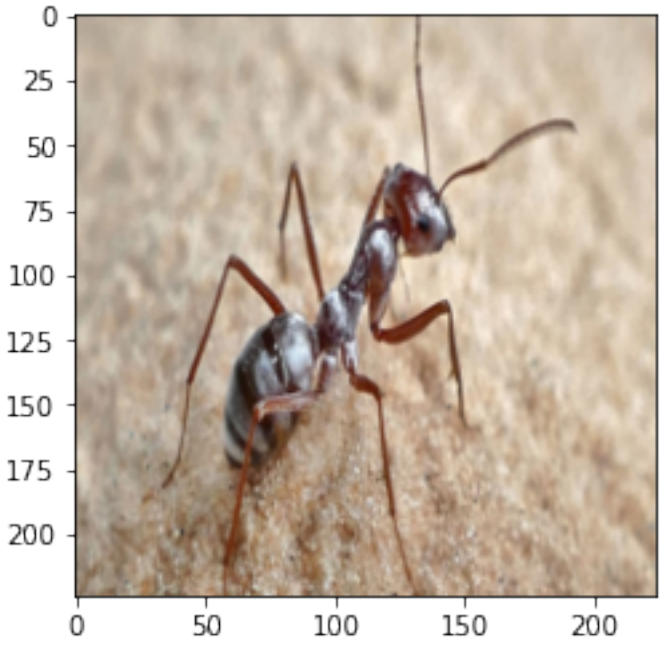


[16]:
```
import requests
import PIL
from PIL import Image

url = 'https://dynaimage.cdn.cnn.com/cnn/c_fill,g_auto,w_1200,h_675,ar_16:9/
 ↪https%3A%2F%2Fcdn.cnn.
 ↪com%2Fcnnnext%2Fdam%2Fassets%2F191016111719-03-record-breaking-ants.jpg'
response = requests.get(url, stream = True)
img = Image.open(response.raw)
plt.imshow(img)
plt.grid()
```



[17]:
```
img = transform(img)
plt.imshow(im_convert(img))
```

[17]: <matplotlib.image.AxesImage at 0x7f677e56f0a0>



[18]:
```
image = img.to(device).unsqueeze(0)
output = model(image)
_, pred = torch.max(output, 1)
print(classes[pred.item()])
```

```
ant
```

[19]:
```
dataiter = iter(validation_loader)
images, labels = dataiter.next()
images = images.to(device)
labels = labels.to(device)
output = model(images)
_, preds = torch.max(output, 1)

fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
  ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
  plt.imshow(im_convert(images[idx]))
  ax.set_title("{} ({})".format(str(classes[preds[idx].item()]),␣
 ↪str(classes[labels[idx].item()])), color=("green" if preds[idx]==labels[idx]␣
 ↪else "red"))
```