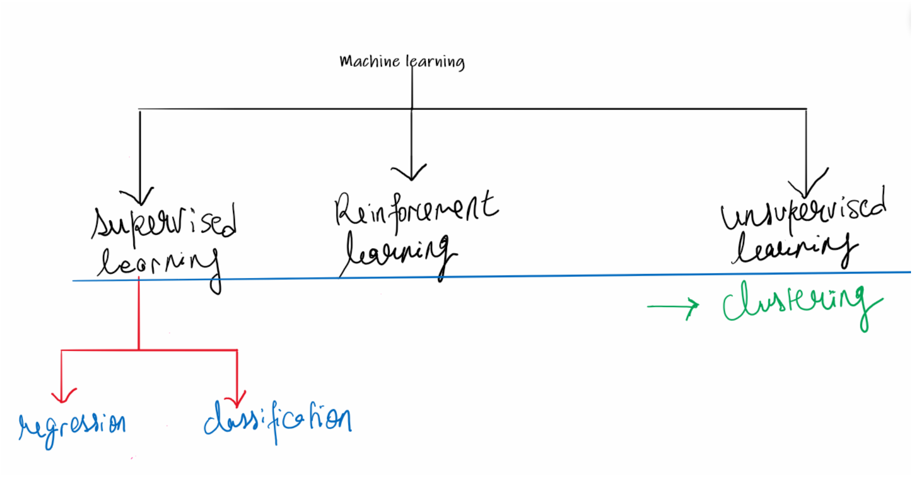
Neural Networks

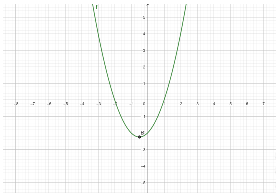
Pranat Sharma, Aaryamitra Pateriya

ML Layout


Machine learning
Supervised learning
Reinforcement learning
unsupervised learning
regression
classification
clustering

Supervised Learning

In the previous session we were able to establish what's known as a loss function and we said that we needed to minimize it for some regression coefficients which we'll refer to as weights now. So, we need to find the minimum of that $L(w)$ function.How do we do that? Well, there are lots of methods but today we'll be talking about only one method in specific which which is known as Gradient Descent method. Let's start with the 1-D case where $w \in \mathbb{R}$. Suppose the loss function is given by $L(w) = w^2 + w - 2$.
So, now to minimize it we follow what we learnt in our 12th grade textbooks.
$\frac{dL(w)}{dw}|_{w*} = 0 => 2w * +1 = 0$
Thus, $w* = -0.5$. One can check that this indeed is the minimum as $L''(-0.5) = 2 > 0$. Thus we achieved the minimum of the loss function. You can imagine this process as tweaking w such that it minimizes $L(w)$. It's like a ball rolling down the hill. Now, what if $w \in \mathbb{R}^2$ i.e. is a two-dimensional vector which can be represented as a column vector.



$$w = \begin{bmatrix} x \\ y \end{bmatrix}$$

The generalization of this is called as a gradient. The gradient of a function gives us it's steepest ascent, hence if we multiply it by -1 we'll get the direction of steepest descent(which we're interested in). Thus

$$\nabla_w L(w)|_{w*} = 0$$

$$=> \nabla_w L(w) = \begin{bmatrix} \frac{\partial L(w)}{\partial w_1} \\ \frac{\partial L(w)}{\partial w_2} \end{bmatrix}$$

Some common used properties gradient:

1)$\nabla_w(af(w) + bg(w)) = a\nabla_w f(w) + b\nabla_w g(w)$

2)$\nabla_w(A^T A w) = (A + A^T)w$

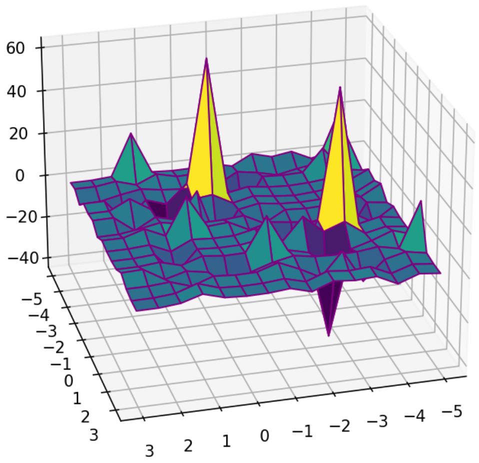
3)$\nabla_w(b^T w) = b$

Now, we know that
$L(w) = (xw - y)^T(xw - y) = w^T x^T xw - 2y^T xw + y^T y$ and now this an exercise for you guys to use the properties of gradients prove the result we arrived in the previous session that $w* = (x^T x)^{-1} x^T y$
(This is bonus exercise)
In general a gradient of a multivariable function $f(x, y)$ is given as:
$\vec{\nabla} f = \frac{\partial f}{\partial x}\hat{i} + \frac{\partial f}{\partial y}\hat{j}$
Gradient is a vector and gives the rise/run or slope of a graph.

Bit On Stochastic Gradient Descent(SGD)

The word "stochastic" here means at random. To reduce the computation power, we pick data points(known as mini-batch) at random and use our gradient decent algorithm till we get zero. Mini-batch is used to strike a balance the usefulness of gradient descent and the efficiency of SGD.
Here we try to minimize
$L(w) = \frac{1}{n}\Sigma_{i=1}^{n} L(w(x_i), y_i) = f(w) = \frac{1}{n}\Sigma_{i=1}^{n} f_i(w)$
and we wanna minimize this new loss function now.
Our new update rule is:
(we'll come back to it bit later) $w_{j+1} = w_j - \eta \nabla_{i_t'} f_i(w)$
Here we're sampling from random index i_t' from $\{1, ..., n, \}$ and then updating.

Gradient Descent



Linear Model

Some common terminologies:

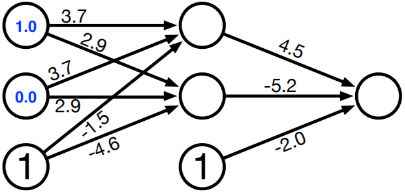
- Features: These are the elements of input vector. The no. of features is equal to the no. of neurons in input layer of the neural network.
- Weights: They represent the "strength" of connection between neurons.
- Biases: These are constants and will be discussed in detail later.
  We compute what's known as a score and it's basically the linear combination of features and weights.
  $score(w_i, d_i) = \Sigma_j w_j h_j(d_i)$
  Score is the weighted linear combination of feature values and weights for example d_i.

Limits of linearity:

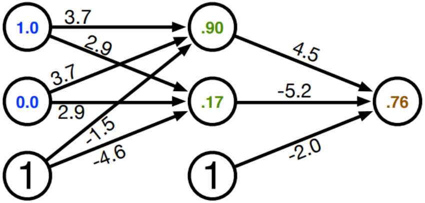
- We can give each feature a weight.
- But not more complex relationships:
  for example: any value in the range $[0, 5]$ is equally good, over 8 are bad and higher than 10 are not worse.

Non-Linearity

we add an intermidiate layer for processing called hidden or dense layer.
Each arrow is a weight.
Now, we apply a non-linearity to our score and we do that by giving it as an input to a non linear function, let's say $f(x)$
$score(\lambda, d_i) = f(\Sigma_j w_j h_j)$
There are many choices for f such as sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and $tanh(x)$
more layers = deep learning.


1.0
0.0
1
1
3.7
2.9
3.7
2.9
-1.5
-4.6
4.5
-5.2
-2.0

1.2


1.0
0.0
1
.90
.17
1
.76
3.7
2.9
3.7
2.9
-1.5
-4.6
4.5
-5.2
-2.0

1.3

Frame Title

Simple Neural Networks(given above)
Innovation: bias unit(no inputs, just 1) Let's do an example, let's chose some arbitrary input values for a neural network given by. Let's perform some hidden unit calculations:
$\sigma(1.0 \times 3.7 + 0.0(3.7) + 1(-1.5)) = \sigma(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$
$\sigma(1.0 \times 2.9 + 0.0(2.9) + 1(-4.5)) = \sigma(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$
sly, for the output layer:
$\sigma(1.17) = 0.76$

Back-Propagation Training

Let's suppose the wanted output from our neural networks should've been 1.0 but what we got 0.76.So, now we've to adjust our weights accordingly but how do we do that. To do that we need to understand some concepts before:

Gradient Decent:

- error obtained is a function of weights.
- Ofc, we wanna reduce the error.
- use gradient decent:move towards the direction minimum error.
- adjust weights such that we move towards minimum error.

Back-Propogation:

- First adjust the weights of the last layer.
- Propagate error back to each previous layer.
- Then adjust the weights.

The derivative of sigmoid function function is given by $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Prove it yourself by using the quotient rule.
Last Layer update:

- score is given by $s = \Sigma_j w_j h_j$
- activation function(or non-linearity) $y = \sigma(x)$
- Error, L2 normalized $E = \frac{1}{2}(t - y)^2$
- derivative w.r.t. to a single weight w_k.
  $\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$

Now, we focus on individual derivative term:
$\frac{dE}{dy} = \frac{d}{dy}\frac{1}{2}(t - y)^2 = -(t - y)$
$\frac{dy}{ds} = \frac{d}{ds}\sigma(s) = \sigma(s)(1 - \sigma(s))$
$\frac{ds}{dw_j} = \frac{d}{dw_j}\Sigma_j w_j h_j = h_j$
Thus, we finally get we always wanted

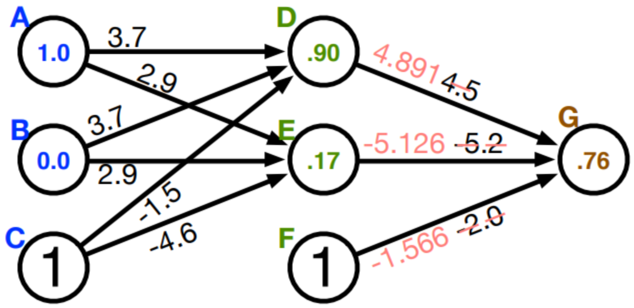
$\frac{dE}{dw_j} = -(t-y)y(1-y)h_j = -(t-y)y'h_j$
Thus we can adjust the weights by a fixed constant called the learning rate(η)

$$\delta w_j = \eta(t-y)y'h_j \tag{1}$$

Multiple Output NN:
For the multiple output NN, we compute error(E) for each output neuron:
$E = \frac{1}{2}\Sigma_k(t_k - y_k)^2$
weights are adjusted according to the nodes the point to $j \to k$
$\delta w_{k\to j} = \eta(t_k - y_k)y'kh_j$

Hidden Layer Update

In dense layers, we don't have a specific output value.
Thus here we compute how much each node, contributed to error.
Error term in each node:
$\Delta_j = (t_k - y_k)y'_k$
Back-propagation of error term:
$\Delta_i = (\Sigma_j w_{i \to j} \delta_j)y'_i$
Arriving at a universal update formula:
$\delta w_{j \to k} = \eta \Delta_k h_j$

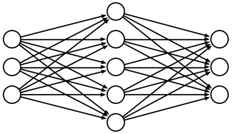


1.4

Initializing Weights

- One way is you can just simply randomly initialize between the closed interval $[-0.01, 0.01]$
- But, the suggested initialization for Shallow Neural Networks is given by:
  $[\frac{-1}{\sqrt{k}}, \frac{1}{\sqrt{k}}]$
  k is the size of the previous layer.
- For Deep Neural Networks:
  $[\frac{-\sqrt{6}}{\sqrt{k_j+k_{j+1}}}, \frac{\sqrt{6}}{\sqrt{k_j+k_{j+1}}}]$
  k_j size of the previous layer and k_{j+1} size of the following layer.

Classification NNs

- 
  We have something known as predict class i.e. one output node for each class.We obtain the output in the form of "one-hot key", e.g. $y = [1, 0, 0]^T$Prediction:
  - predicted class has output node(y_i) with the highest vale.
  - We obtain a posterior probability distribution through soft-max: $sf(y_i) = \frac{e^{y_i}}{\Sigma_j e^{y_j}}$

Unsupervised Learning

K-Nearest Neighbour:

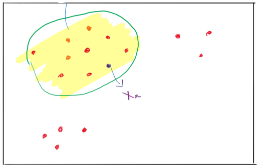
- When we've a new point to classify, we find it's K nearest neighbhours from the training data set.
- The distance is calculated by one of the following measures:
  - Eucledian distance
  - Minkowski distance

KNN Algorithm

Given a query instance to be classified

- given an instance x_a to be classified:
  - Let $x_1, x_2, .., x_k$ be the k instances from our example training data set, that our nearest to x_a.
  - Output the class which represents the maximum of the k instances.

Accuracy(A) = (no. of correctly classified examples/total no. of examples)$\times$100
Eucledian distance $d(x_i, x_j) = \sqrt{(x_{i,a} - x_{j,a})^2}$ for all attributes a.



Weighted K nearest Neighbour

Backward Elimination

- Delete that attribute.
- For all attributes
  - For each example x_i in the training data set:
    - Find K nearest neighbhour in the training data set based on eucledian distance.
    - Predict the class by finding the maximum class represented in the k nearest neighbhours.
    - calculate the accuracy
- If the accuracy has reduced, restore the deleted attribute.

Weighted KNN with Backward Elimination

- Read the training data.
- Read the testing data.
- Set K to some value.
- Normalize the attribute in somewhere $[0, 1]$:
  $Value(v) = \frac{v}{1+v}$
- Apply backward elimination.
  Follow the steps given above.