

# Advanced Machine Learning, Deep Learning, NLP & Graphical Models

Ryan Vaz

Hunter College

**Abstract.** This PDF is a collection of lecture notes from the Statistics 72556 class at Hunter College with some brief simplifications and details regarding how to approach Machine Learning from a Statistical POV. A firm grasp of Python Programming, Statistics, and Matrix/Linear Algebra is assumed. I'll try to minimize the usage of Calculus however some understanding will help certain topics.

## Table of Contents

Advanced Machine Learning, Deep Learning, NLP & Graphical Models .....	1
<i>Ryan Vaz</i>	
1 Lecture 1 - ANN .....	1
1.1 Machine Learning Problems .....	1
1.2 Basic Neural Networks .....	1
1.3 Deep Neural Networks .....	3
1.4 Error Backpropagation Learning Algorithm .....	3
1.5 Gradient Descent(GD) .....	4
2 Lecture 2 - MLP Architecture .....	5
2.1 Math .....	5
2.2 Hyperparameters .....	6
2.3 Methodology .....	6
3 Lecture 3 - DNN Fitting .....	7
3.1 Gradient Problems .....	7
3.2 The forefront of Deep Learning .....	8
4 Lecture 4 - CNN .....	10
4.1 Convolutional Neural Networks .....	10
*	

## 1 Lecture 1 - ANN

### 1.1 Machine Learning Problems

**Unsupervised Learning** A problem where only the features are observed.

**Supervised Learning** A problem in which both the features and the target are observed. We can further classify supervised learning into two categories...

**Regression** A case of supervised learning where the variable we are studying is continuous/numerical

**Classification** A case of supervised learning where the variable we are studying is categorical

### 1.2 Basic Neural Networks

**Artificial Neural Networks(ANNs)** A neural network is a computing system that is inspired by biological neural networks that make up the brain. They consist of multiple connected units called artificial neurons or linear threshold units.

**Table 1.** Examples of different machine learning problems and classifications

Problem Type	Example
Unsupervised	Market segmentation where we divide customers into groups based on their characteristics
Regression	Predicting the value of the DOW in 6 months Predicting the value of a given house based on various inputs
Classification	Will the DOW be up (T) or down (F) in 6 months? Is this email a spam(T) or not(F)?

**Linear Threshold Unit(LTU)** A single artificial neuron that calculates the net output from a series of inputs

It uses inputs( $\mathbf{x}$ ) and weights( $\mathbf{w}$ ) to calculate a weighted sum

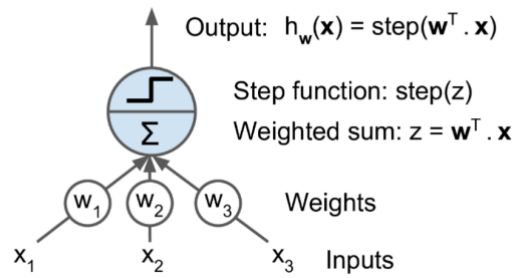
That value is then applied to a binary step function which turns the result into a Boolean value.

**Theorem 1 (Weighted Sum).**

$$z = w_1x_1 + w_2x_2 + \dots + w_kx_k = \mathbf{x}^T \mathbf{w}$$

**Theorem 2 (Step Function).**

$$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



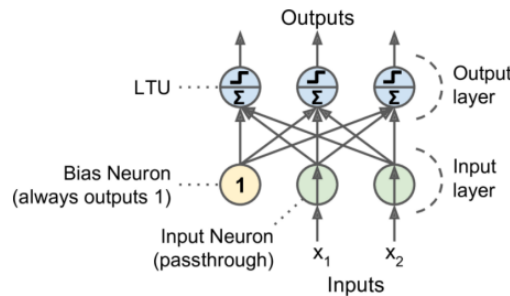
**Fig. 1.** Example of a LTU.

To summarize a single LTU: It takes in data from input layer and calculates output by multiplying it with the corresponding weight then summing it together and sending it through a step function(Binary Step Function in this case).

**Perceptron** A neural network formed from a single layer of LTUs

Each LTU is connected to all the input neurons along with an extra input neuron that is always 1(bias neuron). This bonds the Input Layer to the Output Layer.

These Perceptrons can output several values(Multioutput Classifier).



**Fig. 2.** Example of a Dense Perceptron.

If all the neurons in a layer are connected to every neuron in the previous layer, its a dense layer

**Theorem 3 (Output of a Dense Layer).**

$$h_{W,b}(X) = \phi(XW + b)$$

$\phi$  - activation function

X - inputs, W - weights, b - bias vector

### 1.3 Deep Neural Networks

**Perceptron Training** It's commonly believed that when biological neurons trigger others, it reinforces the connection between these two, the same logic is what's used for perceptrons.

To train perceptrons, they're given a training instance and it makes a prediction.

When it makes a WRONG prediction, this reinforces the connection between the inputs that would have contributed to a correct prediction.

This can be expressed mathematically as follows:

**Theorem 4 (Weights Update).**

$$w_{i,j}^{next\ step} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

$w_{i,j}$  is the connection weight between the i-th input neuron and the j-th output neuron

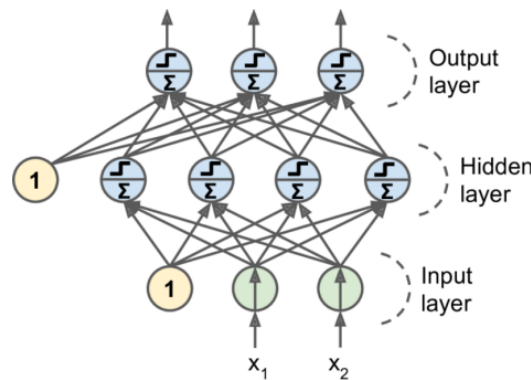
$x_i$  is the i-th input value of the current training instance

$\hat{y}_j$  is the output of the j-th output neuron for the current training instance

$y_j$  is the target output of the j-th output neuron for the current training instance

$\eta$  is the learning rate

**Deep Neural Networks(DNNs)** When a perceptron has multiple layers of LTUs(or hidden layers) that are fully connected to the next layer with a bias on each level, we call that a Multilayer Perceptron(MLP) or Deep Neural Network(DNN).



**Fig. 3.** Example of a DNN.

### 1.4 Error Backpropagation Learning Algorithm

The next set of topics is going to be significantly harder, so if you want to get the spark notes and skip to the coding section, they go as follows: in order to train DNN's a backpropagation training algorithm is used, which is basically gradient descent using reverse-mode autodifferentiation.

Note: This section is going to be different in that we will simulate each step of the algorithm

**Initialization** The weights of the input layer are initialized with random small values.

**Forward pass** The input layer is provided and the neural network begins processing the data. The difference between expected and actual value is provided directly to the backwards pass step.

**Reverse Pass** This step computes how much each neuron in the last hidden layer contributed to each output neuron's error and it tweaks the weights according to a gradient descent(coming soon!).

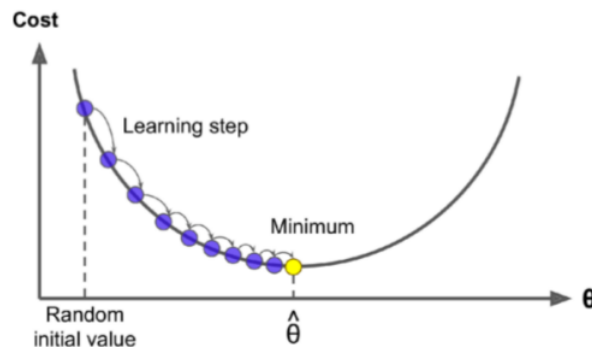
It then repeats the initialization, forward pass and backwards pass which is why this whole process is called backpropagation, or backprop for short as it propagates the error back to the weights of the neurons that contributed to the error.

Now the question remains, how do we minimize this error function?

## 1.5 Gradient Descent(GD)

Yes, Gradient Descent gets it's whole own section, mad? Stay mad.

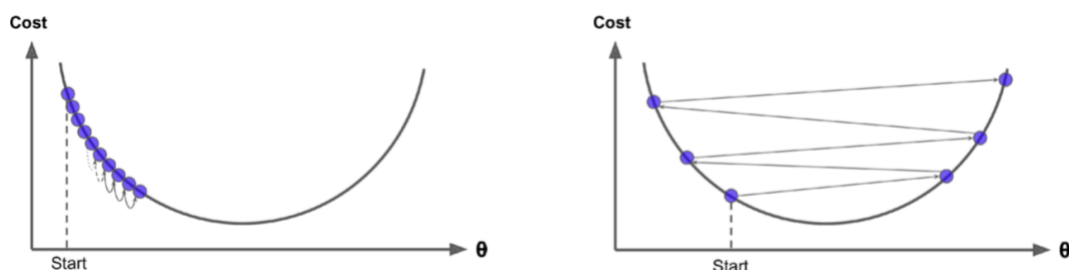
**Definition** GD is a optimization algorithm that starts with a random value on a graph and will try tweaking the parameters in order to minimize a cost/error function  
By using GD, we can find the minimum amount of error to best use with our DNN.



**Fig. 4.** Example of Gradient Descent.

**Learning Rate( $\eta$ )** Learning rate will determine the size of the steps. If it's too small, it won't be able to get a solution in a reasonable amount of iterations. If it's too big, it may skip the final result entirely and diverge!

Learning rate is so important can make or break our code, thus it is a hyper-parameter. Our code has other hyper-parameters that we will address later on.



**Fig. 5.** Example of bad learning rates.

**Math** Problem: Minimize  $f(w)$  - weight/error function

iterative solution:  $w_{k+1} = w_k - \gamma_k \nabla f(w_k)$

$w_{k+1}$  = next iteration

$w_k$  = current iteration

$\gamma_k$  = step size

$\nabla f(w_k)$  = gradient of f

One such example is our linear regression problem, let's switch w with  $\theta$

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n = h_\theta(x) = \theta^T \mathbf{w}$$

We need to find  $\hat{\theta}$  which is the minimizes the criterion(loss)

$$f(\lambda_k) = MSE(X, h_\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T x^i - y^i)^2, \nabla f = \frac{2}{m} X^T (X\theta - y)$$

Gradient Descent Step Equation

$$\theta^{nextstep} = \theta - \eta \nabla_\theta MSE(\theta)$$

where

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y)$$

Implementation in code:

```
eta = 0.1 #learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)

for iter in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta -= (eta * gradients)
```

## 2 Lecture 2 - MLP Architecture

### 2.1 Math

**MLP as a Composite Function** Imagine each layer of the Neural Network is a function, then we can say that function n maps to function k with the invertable linear map  $f : R^n \rightarrow R^k$  where f is expressed as the following

**Theorem 5.**  $f(x) = g \circ f_N \circ f_{N-1} \dots \circ f_2 \circ f_1(x)$ ,  $f_i : R^{i-1} \rightarrow R^i$

where  $f_i(x) = \sigma(w_i x + b_i)$ ,  $i = 1, 2, \dots, N$

$$\implies f(x) = g(\sigma(\dots(\sigma(w_2 \sigma(w_1 x + b_1) + b_2) + \dots + b_N))$$

Where  $\sigma$  is a non-linear activation function and  $g(x)$  is the linear decision function.

**Autodiff** s.p.z. we have a function that maps n dimensional values to a scalar output, and we have  $x_1, x_2, \dots, x_n$  as inputs, and  $x_{n+1}, \dots, x_{N-1}$  as intermediate values, and  $x_N$  to be the final value.

Example problem: Compute  $f(x)$ ,  $L(f(x), y)$ , and  $dL/dw_i$

1. Input  $\mathbf{x}$
2. Set  $v_{1..n} \leftarrow x$
3. For  $i = n+1, n+1, \dots, N$ :

$$\begin{aligned} v_i &\leftarrow \sigma_i(w_i \cdot v_{Pa(i)}) \\ v'_i &\leftarrow \sigma'_i(w_i \cdot v_{Pa(i)}) \end{aligned}$$

4. Set  $L \leftarrow L(f, y)$
5. Compute  $\frac{dL}{df}$
6. Initialize  $\frac{dL}{dv_i} \leftarrow 0$
7. For  $m = 1, 2, \dots, M$ :

$$\frac{dL}{dv_{N-M+m}} \leftarrow \frac{dL}{df_m}$$

8. For  $i = N, N-1, \dots, n+1$ :

$$\begin{aligned} \frac{dL}{dw_i} &\leftarrow \frac{dL}{dv_i} v'_{Pa(i)} \\ \frac{dL}{dv_{Pa(i)}} &\leftarrow \frac{dL}{dv_{Pa(i)}} + \frac{dL}{dv_i} v'_i w_i \end{aligned}$$

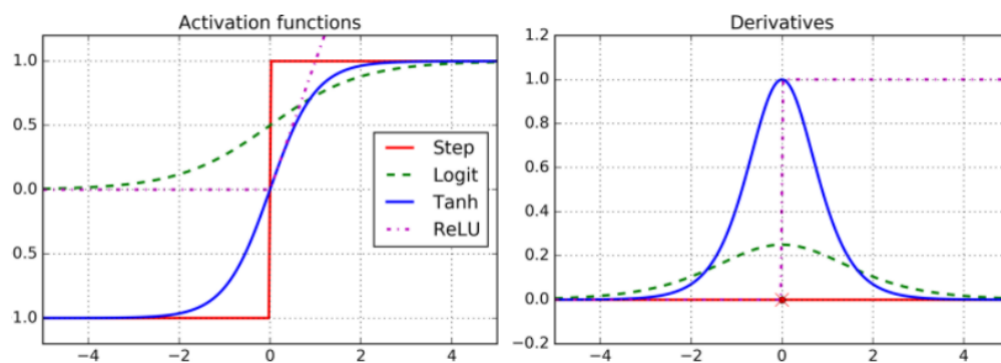
## 2.2 Hyperparameters

### Hyperparameter Examples

1. # of hidden layers
2. # of neurons per hidden layer
3. Learning rate( $\eta$ )
4. Optimizer
5. Batch size
6. Activation functions

### Activation Functions

1. ReLU (most used)
2. Hyperbolic tangent function (tanh)
3. Logit function
4. Sigmoid (not as good as ReLU, but biologically more accurate)



**Fig. 6.** Comparison of activation functions.

## 2.3 Methodology

**Regression MLP Architecture** When Using Regression, we do not need an activation function, and furthermore the loss function is normally Mean Square Error(MSE). Due to no activation function, it's easy to think of Regression MLP's as a MLP with the identity activation function.

**Table 2.** Examples of standard values for Hyperparameters

Hyperparameter	Typical Value
# input neurons	One per input feature
Regression	Predicting the value of the DOW in 6 months Predicting the value of a given house based on various inputs
Classification	Will the DOW be up (T) or down (F) in 6 months? Is this email a spam(T) or not(F)?

**Classification MLP Architecture** Classification is best used when outputs can be grouped catagorically, there are 3 different classifications that we will focus on.

**Table 3.** Types of Classification MLP

Classification Model	Use Case
# Binary	A probability of it being the assumed output class
Multilabel Binary	Similar to binary, but using a 2 output classes
Multiclass	Also similar to binary, but using > 2 output classes

**Softmax** The softmax function makes it so the sum of the estimated probabilities will be between 0 and 1, and furthermore will ensure that the sum of them all will equal 1.

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^k \exp(s_j(x))}$$

## Backpropagation

Step 0: Initialize weights and biases

Choose the weights and biases, typically at random

Step 1: Pick one of the data points at random

Input the vector  $x$  into the left side of the network and calculate the output vector  $y$

Step 2: Calculate contribution to the loss function

Adjust the loss function accordingly

Step 3: Starting at the right, calculate all the  $\delta$ 's

$$\delta^L = \frac{d\phi^L}{dz} \big|_{z^L} (\hat{y} - y)$$

As you move to the left, connect the nodes with the biases.

$$\sum_j \delta_j^l w_{i,j}^l$$

Step 4: Update the weights and biases using (stochastic) GD

Return to step 1

## 3 Lecture 3 - DNN Fitting

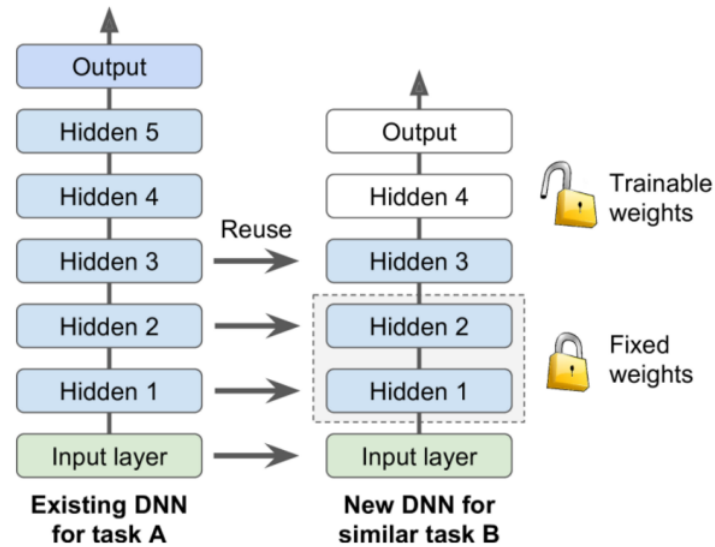
### 3.1 Gradient Problems

**Vanishing/Exploding gradients** Gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is Known as the Vanishing Gradients Problem

Similarly the gradient can also grow bigger, then diverge. This is the Exploding Gradient Problem. We'll address some possible solutions that you can implement to solve this

1. One approach is replacing the normal distribution (mean 0, deviation 1) random initialization of the weights with a normal distribution with mean 0 but a custom deviation to try to equal the variance of the inputs and outputs
2. Another approach is to replace the logistic sigmoid activation function(due to the fact that it saturates) with ReLu, Leaky ReLu or ELU.  
In general ELU > leaky ReLU > ReLU > Tanh > Logistic
3. Adding Batch Normalization to normalize the inputs then scale and shift the results, but at the cost of run time

**Transfer Learning** Ever heard the saying it's not your code, it's our code? That's exactly what we're doing here.



**Fig. 7.** Example of Transfer Learning.

The main use case for this is when we find a similar problem that would require a similar model to solve, we can "yoink" them from model zoos to accelerate our learning.

### 3.2 The forefront of Deep Learning

This section is purely optional, but if you stick with me, we'll learn some very cool topics that are at the forefront of Deep Learning, and can be incorporated into your projects. We'll not go into too much detail, so just enjoy the theory.

**Momentum Optimization** Ever seen a ball roll down a hill, ever wondered if we can model gradient descent the same way. Well I guess I'll take a 50%, but we can do that!

$$m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta)$$

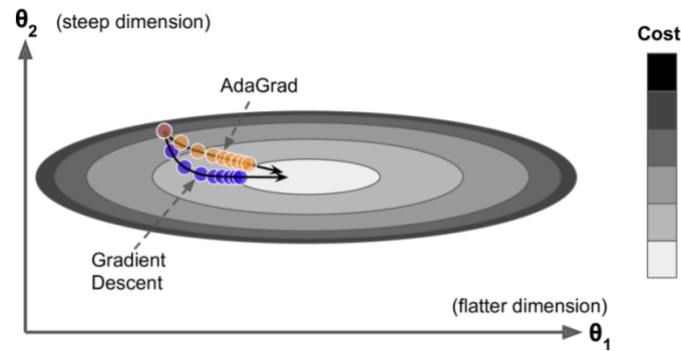
$$\theta \leftarrow \theta + m$$

Where the terminal velocity equals  $1/(1 - \beta)$ .

**Nesterov Accelerated Gradient (NAG)** Similar to Momentum Optimization however we measure the gradient at a point slightly ahead of it's momentum.

**AdaGrad** By making the learning rate slower, but allowing it to accelerate for steep dimensions, we gain an adaptive gradient which can push towards the minimum more effectively.





**Fig. 8.** Adagrad vs GD.

**RMSProp** Similar to AdaGrad, but it doesn't slow down that much, because it only accumulates the gradients from the most recent iterations

**Adam (adaptive momentum estimation)** combination of the Momentum and the RMSProp  
 Uses Momentum optimization to keep track of an exponentially decaying average of past gradients  
 Also keeps track of average of past squared gradients (like RMSProp)

**Table 4.** Optimizer Comparison

Class	Speed	Quality
SGD	*	***
SGD(Momentum = ...)	**	***
SGD(Momentum = ..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

**Regularization** Due to the structure of the data, it is easy to overfit it, thus we should stop it once it gets close enough otherwise it'll attempt to mimic the patterns viewed in this dataset, and won't be able to replicate it in other datasets, thus ruining the point of analyzing it.

**Early Stopping** Interrupt the training when it's performance on the validation data sets starts to decline. This method works, but it relatively barebones.

**Dropout Regularization** Method: at every training step, every neuron has a probability  $p$  of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step. This prevents neurons from becoming co-dependant on each other.

**Data Augmentation** Generating new training instances from the already existing ones, which will increase the training set. This core idea will be readdressed on GANs.

**DNN Summary** Thank you for sticking with me for the DNN Section of this PDF, the last thing I want to provide you is a state of the art DNN Configuration that you can almost always rely on

**Table 5.** Default DNN configuration

Hyperparameter	Default Value
Kernel Initializer	He Initialization
Activation Function	ELU
Normalization	None if Shallow; Batch Norm if Deep
Regularization	Early Stopping
Optimizer	Momentum Optimization(or RMSProp or Nadam)
Learning Rate Schedule	1 Cycle

## 4 Lecture 4 - CNN

### 4.1 Convolutional Neural Networks

**Intro** Unlike ANN's and DNN's which focus on analyzing each and every pixel of an image and connecting each nerve to each nerve. CNN's are modeled after the brain's way of understanding