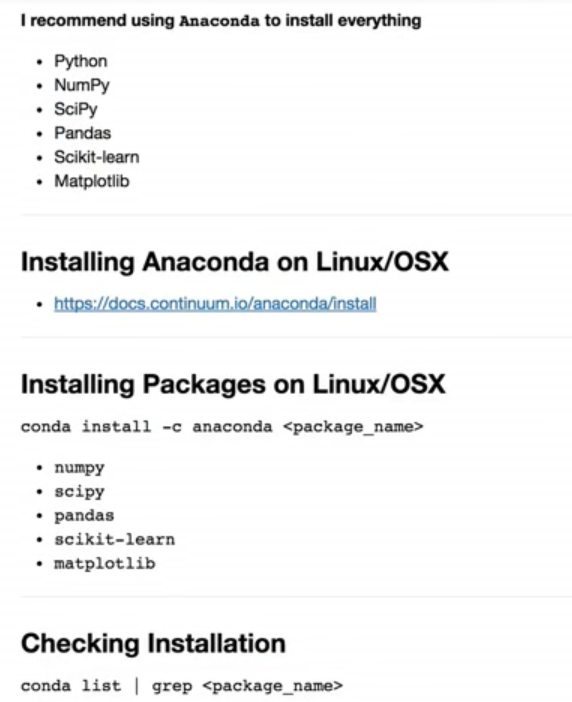
# Requirements:



# Instalation on Windows

1. **Download and InstallAnaconda (Python 3.6)**

<https://www.continuum.io/downloads>

1. **Download and install Kivy graphic library**

<https://www.dropbox.com/s/wso12fmfd55ysus/Kivy-1.10.1.dev0-cp36-cp36m-win_amd64.whl?dl=0>

Anaconda Prompt -> right click -> run as administrator -> commands:

conda install -c anaconda python=3.6.1

cd C:\Users\seby\Downloads **(local onde foi baixado kivy)**

pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew

pip install kivy.deps.gstreamer

pip install Kivy-1.10.1.dev0-cp36-cp36m-win\_amd64.whl

1. **remove any packages of torch if existes:**

conda uninstall cuda80

conda uninstall cuda90

1. **Install pytorch**

conda install -c peterjc123 pytorch

1. **Check instalation**

conda list

# Matrizes

**- Matriz nomral**: arr = [[1, 2], [3,4]]

**- Converter matriz normal para pytorch:** torch.Tensor(arr)

**- Matriz 2x2 torch com valores 1**: torch.ones((2,2))

**- Matriz 2x2 torch com valores 0**: torch.zeros((2,2))

**- Matriz 2x2 torch com valores randomicos 0 até 1**: torch.rand((2,2))

# Seed Numeros Randômicos

- O seed é utilizado para gravar uma escolha de numero randômicos (para qualquer matriz) onde pode ser utilizada varias vezes com o mesmo valor:

torch.manual\_seed(0) #0 é o index da sequencia gravada, onde podem ser acessados novamente (pode ser qualquer para gravar a sequencia randômica)

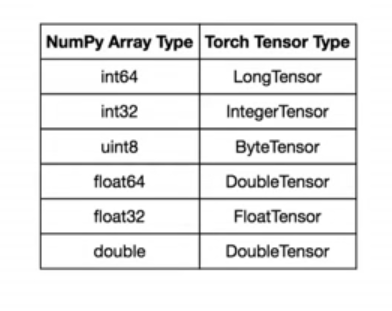
torch.rand(2,2) #Se este método estiver executado antes da linha torch.manual\_seed(x) então vai obter a sequência gravada, se não vai gerar um matriz com valor qualquer de 2x2

# Converter Matriz numpy para Torch

Ele só consegue converter matrizes do tipo: double, flaot, int64, int32, uint8

np\_array = np.ones((2,2),dtype=np.int64) ou np\_array = np.ondes((2,2))

torch.from\_numpy(np\_array) **#Converter**



# Converter Matriz Torch para Numpy

torch.ones(2,2)

numpy\_matrix = torch\_matrix.numpy();

# Converter Matrix Torch CPU para GPU

tensorMatrix\_CPU = torch.ones(2,2)  **#CPU Matrix (Default)**

If torch.cuda.is\_available():

tensorMatrix\_GPU = tensorMatrix\_CPU.cuda()  **#Converter**

# Converter Matrix Torch GPU para CPU

tensorMatrix\_CPU = torch.cuda.ones(2,2)  **#GPU Matrix**  tensorMatrix\_GPU = tensorMatrix\_GPU.cpu() **#Converter**

# Tensor Matrix convert to array (Utilizado para NN)

- Pode ser utilizado em matrix de qualquer dimensão para converter em array 1D

x = torch.ones(2,3)

1 1 1

1 1 1

x.view(6) **#(6 = 2x3 obrigatorio ser a multiplicação linhaxcoluna para caber todos os elementos)**

1

1

1

1

1

1

# Torch Matrix Math Opertaions

## Soma

c= a + b / c= torch.add (a,b)

c.add \_(a) **#\_ para indicar inplace que é para colocar a resposta no c**

## Subtração

c = a - b / c= torch.sub(a,b) #sem \_ a resposta é somente retornada

c.sub\_(a) **#\_ para indicar inplace que é para colocar a resposta no c**

## Multiplicação

c = a \* b / c = torch.mul(a,b)

c = torch.mul\_(a,b) **#\_ para indicar inplace que é para colocar a resposta no c**

## Divisão

c = a / b / c = torch.div(a,b)

c = torch.div\_(a,b) **#\_ para indicar inplace que é para colocar a resposta no c**

# Tensor Funtions

## Create a tensor = torch matrix

x\_tensor = torch.tensor([1,2,3,4,5]) **#tem que passar como lista para criar Tensor**

## Size of tensor

x\_tensor.size()

## Media(mean) de uma dimensão do Tensor

x.mean(dim=0) **#dim=0: 1º dimensão que é a primeira linha do tensor**

x.mean(dim=1) **#dim=1: 1º & 2º dimensão que vai retornar lista da media da primeira e segunda dimensão**

## Calcular desvio Padrão de uma dimensão

x.std(0)

# Variable

Variable guarda um tensor + gradiente

Variable pode fazer operações matematicas

a = Variable(torch.ones(2,2), requires\_grad=True)

## Operações Matematicas + - \* /

- As operações matematicas são efeituadas para o tensor do variavel

a+b / a.add(b)

a-b / a.sub(b)

a\*b/ a.mul(b)

a/b / a.div(b)

# Gradiente

É utilizado para obter o maior taxa de crescimento/decrescimento da função em um determinado ponto. O gradiente é a direção que tem o maior taxa de variação(maior esforço)

É um vetor que indica o sentido e a direção do deslocamento. E calculado por taxa de variação do valor medio(maior crescimento):

O Gradiente é utilizado para fazer o processo de backpropagation & processo de aprendizagem(Optimizer) da rede neural. Onde o gradiente é calculado via a taxa de variação(dervidada) do erro medio quadrado(MSE). O gradiente é multiplicado por o indice de aprendizagem e depois é subtraido do parametro atual(Seção do Otimizador).

Matematicamente o gradiente pode ser positivo(se alterar a direção para frente) ou negativo (se a direção foi alterada para atrás)

## Calculo do Gradiente via Erro MSE:

Calcular Gradiente = Calculas taxa de variação do Erro(previsto - desejado):

(Derivada do Erro)

(Seção do Calculo de Erro)

## Calculo do Gradiente via Likelihood Criterion

Esse tipo de calculo do gradiente é utilizado para redes neurais para

Código:

**#Metodo para sabe o quão errada está a resposta**

lossCriterion = nn.MSELoss()

**#Calculo do Erro: loss(x,y)=1/n∑|xi−yi|^2 (x = matriz de saida, y = matriz dos valores desejados)**

loss\_error = lossCriterion(output\_values, labels\_desejados)

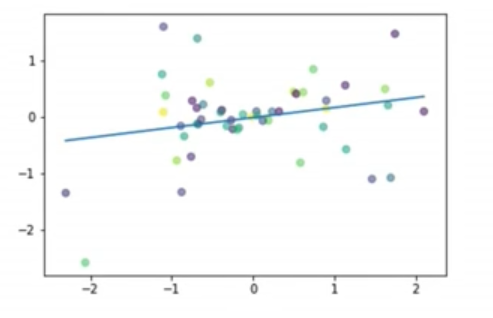
**#- obtendo os gradients (Calculo do gradiente apartir da perda)**

loss\_error.backward()

# - Linear Regression

A função de regressão linear de primeiro grau tem objetivo de criar uma reta que abora o maior numero de pontos para serem interligados, assim diminuindo o erro o máximo possível, onde a letra ‘a’ equivale o coeficiente angular que determina o angulo de inclinação da reta. A letra ‘’ é equivale o coeficiente linear, que determina a altura da reta no eixo Y.

A função ‘y’ depende do ‘x’, mas o ‘x’ não depende do ‘y’. Por isso o ‘y’ é chamado de termo dependente.



- Adotando função de ativação y = 2x+1. Criamos um matrix de valores 0 -> 10

x = [i for i in range(11)]  **# Criar array de uma sequencia**

x\_train = np.array(x, dtype=np.float32)  **# Colocar o array em numpy**

x\_train.shape  **# Mostrar o tamanho do array em cada dimensão**

x\_train = x\_train.reshape(-1,1) **# Converter cada linha em coluna (-1 é a ultima dimensão) para depois implementar o gradiente**

y = (2 \* i + 1 in i range x) **#Apartir da função y = 2x + 1 criar o matrix Ydo X**

y\_train = np.array(y. dtype=np.float32) **# Colocar o array em numpy**

y\_train = y\_train.reshape(1,-1) **# Converter cada linha em coluna (-1 é a ultima dimensão) (formato para processamento)**

# Calculo de Erro (MSE)

O Calculo do erro é utilizado para calcular a diferencia entre um ponto previsto com o valor do ponto real. **É utilizado para ajustar os pesos das conexões da rede neural para aproximar o valor previsto do valor real (backpropagation).**

MSE Loss = Mean Squared Error (Erro Medio Quadrado)

= Predicted

Y = true value

Código:

PyTorch: lossCriterion = nn.MSELoss()

# Otimizador (Melhorador das Conexões Synapses)

O otimizador é um processo de aprendizagem onde tem o indice multiplicativo que modifica os Variables () para ter o menor erro MSE e os valores previstos sejam mais próximos dos valores reais.

Utilizando a função de ativação Linear , os parametros alfa e beta são alterados no processo da otimização(aprendizagem) para obter o melhor resultado com menor erro.

Parameters\_New = Parameters – Learning\_rate \* parameters\_gradient

: Parameters ()

: indice de aprendizagem (velocidade de alteração).

Se = 1 (Sempre subsituir a memoria antiga por a memoria nova)

Se = 0 (Sem aprendizagem, descartar a memoria nova)

: Parameters Gradient

Exemplo: Matrix desejado [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

No processo de aprendizagem o valor de tende a ser 2 e o valor tende a ser 1 para Criar uma reta(Função Linear) que aborda todos os pontos do matriz. Será formado a função y = 2x + 1. O processo de aprendizagem vai tentar jogar o indice(0,1,2,3) no ‘X’ e ajustar o valor de para obter o valor desejado no ‘Y’

Código:

Learning\_rate = 0.01

Optim = torch.optim.SGD(model.parameters(), lr = Learning\_rate) **#SGD é o processo/algoritimo de otimização (existem varios outros)**

# Processo de treinamento

1 epoch = percorrer todos os valores do matrix de X 1 vez

1. converter os matrizes numpy para tensor depois para variable (formato para processamento).
2. Limpar o Buffer dos gradiente para remover os gradientes antigos. (porque os gradientes antigo já foram incluidos nos pesos)
3. Executar o metodo para processar forward e obter os valores da saidas
4. Calculo do Erro: loss(x,y)=1/n∑|xi−yi|^2 (x = matriz de saida, y = matriz dos valores desejados
5. obtendo os gradients dos parametros alfa e beta(Calcular o gradiente = derivada do Erro MSE) (Gradiente = Taxa de variação do erro que atualiza os pesos/parametros alfa e beta)
6. Atualizar os pesos/parametros alfa e beta: Parameters\_New = Parameters – Learning\_rate \* parameters\_gradient

# Salvar o Carregar Parametros (alfa e beta)

import os #Biblioteca para salvar e load um arquivo

def save():

torch.save(model.state\_dict, 'linear\_model.pkl') #salvar somente os parametros (alpha e beta)

def load():

if os.path.isfile('linear\_model.pkl'): #Verificar se o arquivo existe,path já leva

model.load\_state\_dict(torch.load('linear\_model.pkl')) #load somente os parametros (alpha e beta)

# Plotagem de Gráfico Demonstrative

import numpy as np **#Biblioteca Responsable for criar arrays, matrizes e funções em Python (Usada para criar matrix numpy )**

import matplotlib.pyplot as plt **#(Opcional e demonstrativo)Biblioteca de plotagem de graficos**

plt.clf()  **#limpar o plotter**

plt.plot(x\_train,y\_valores\_desejados,'go',label='Valores desejados',alpha=0.5) **#plot o grafico com pontos e coeficiente angular 0.5 (45º)**

plt.plot(x\_train,output\_values.data.numpy(),'--',label='Valores Previstos',alpha=0.5)**#plot o grafico traçado e coeficiente angular 0.5 (45º)**

plt.legend(loc='best') **#Mostrar legenda no gráfico**

plt.show() **#Mostrar o gráfico**

# Processamento GPU (Não é suportado por Windows)

Para melhor processamento é utilizado o GPU para processamento do gráfico e da dos torch.Variables. **As objetos recomendaveis estar na GPU é modelo e Variables(training)**

If torch.cuda.is\_available() **#module**

model.cuda()

if torch.cuda.is\_available():  **#Training (se GPU não está available vai roda em CPU)**

inputs = Variable(torch.from\_numpy(x\_train).cuda())

if torch.cuda.is\_available():

labels = Variable(torch.from\_numpy(y\_train).cuda())

# - Logistic Regression

A função logisitc regression é utilizado em redes neurais onde tem objetivo de obter a probabilidade de uma ação, categorizar ações, imagens...etc

# Linear vs Logistic Regression

O Linear Regression tem saída de numeros qualquer y = αx+β, impossibilitando utilizar para categorizar ações, Isso porque a reta depende de todos os pontos e isso mudar a reta e não convergir com os valores.

O logistic regression ao invés de prever a saída com valor exato, ele fornece entre (probabilidade) entre 0 e 1 que pertence uma classe (ação).

**Exemplo de reconhecimento de padrões de numeros Utilizando filtros**

# Neurons de Entrada 784 = imagem 28x28

# Neurons da Saida 10 = numero 0 a 9

# MNIST Dataset é um dataset que contem imgens filtradas de numeros manuescritos, o dataset contém tuples de imagens e label para indica qual numero que a imagem pertence

# O MNIST Dataset é utilizado para treinar os redes neurais para reconhecer padrões de numeros (tem 60000 imagens de numero de 0 a 9 feitas de formas diferentes)

# O MNIST Dataset tem 60000 posições(digitos aleatóris) dataset[59999] cada um do tipo tuple[][] (1º imagem 28x28 - 2º numero)

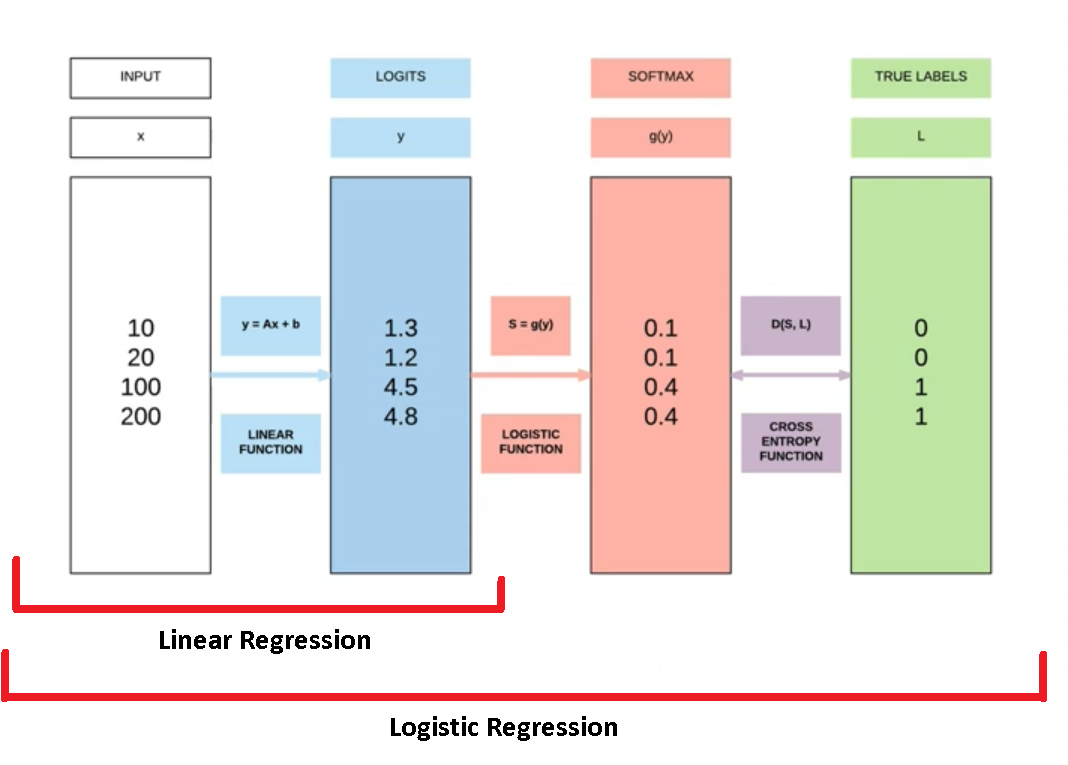
# Cada posição (digito) do dataset contém um tensor 1x28x28 (imagen 28x28) + um label que indica qual numero que é

# É necessário testar a precisão e o indice de erro para não ficar aprendendo até chegar até o melhor ponto e depois perder a precisão (overfitting)

# Etapas do Logistic Regression

O logistic regression contem 4 etapas: as primeira 2 etapas são Linear Regression

Depois tem a função logaritima da função linear para o calculo da probabilidade e depois tem o calculo do erro.



1. Linear Regression(função/reta que aborda mais pontos):

**alfa é um matrix de numeroDeSaidas x numeroDeEntradas**

**beta é um array de numero de saidas x 1**

X é um matrix 784 \* 1 (784 = 28(Entradas) \* 28(Saídas))

**Alfa \* X é um produto de matrizes**

**(Alfa \* X) + Beta é um soma de matrizes**

1. Função Logatima da função linear:
2. Calcular a probabilidade de y=1 apartir para cada valor de entrada (X)
3. Softmax: Utiliza as funções logaritimas para classificar cada classe(ação)
4. Calculo do erro(seção abaixo) & calculo dos valores do softmax
5. Processo de aprendizagem com o otimizador

Parameters\_New = Parameters – Learning\_rate \* parameters\_gradient

## Calculo do erro (Cross Entropy)

Calculo do erro via o calculo da distância entre os valores logaritimos do Softmax e os valores desejados:

D: distancia entre saida dos valores logaritimos do Softmax e os valores desejados

S: saida dos valores logaritimos do Softmax

L: valores desejados (0 ou 1)

-Quando o L(valor desejado) = 0:

Se o valor do S = L = 0, então D(S,L) tem valor positivo menor da distância de erro = menor o erro

Se o valor do S =1 e L = 0, então D(S,L) tem valor mais positivo da distância de erro = maior o erro

-Quando o L(valor desejado) = 1:

Se o valor do S = L = 1, então D(S,L) tem valor menos negativo da distância de erro = menor o erro

Se o valor do S = L = 1, então D(S,L) tem valor mais negativo da distância de erro = maior o erro

**Calculo do Erro Cross Entropy Loss L**:

Cross Entropy Loss L = É a somatória De: a distância entre os valores na saida do softmax e valores desejados, divide por o numero de valores do matriz de saída(achar a média do erro).

# - Feedward Neural Network

## Neural Network vs Logistic Regression

Logistic regression analisa e processa bem as funções lineares, mas não tem um bom indice de analise para o processamento de funções não lineares(Equação de segundo grau, ou mais graus)

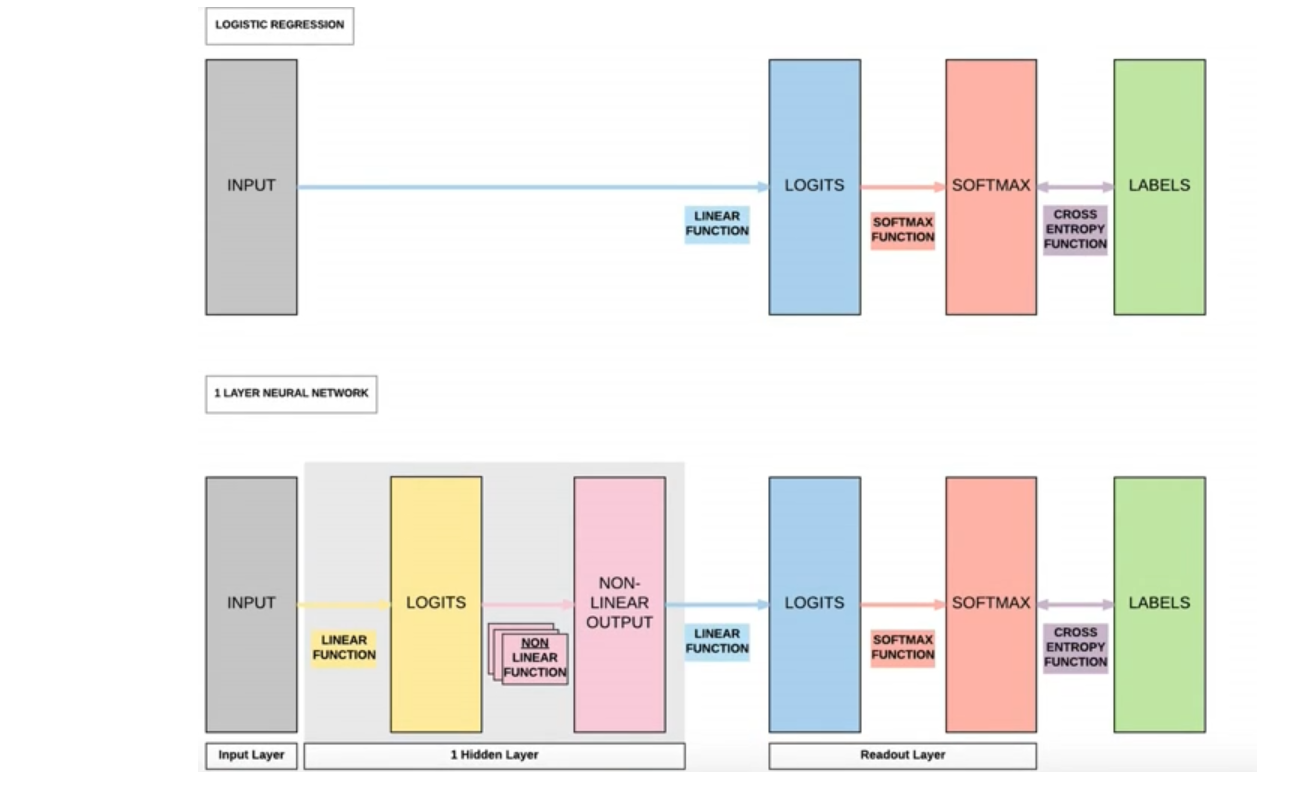
Model: Linear Ativation Function(Entrada - Oculta) -> Non-Linear Ativation Function(Camada Oculta - Camada logaritima) -> Linear Ativation Function (Camada Logartima - Saida final) -> Entropy Loss e Softmax

Quanto mais neurons na camada Oculta ele aprendará mais, mas precisará mais dados e mais treinamentos. Cada Neuron na camada oculta tem uma função não-linear/linear

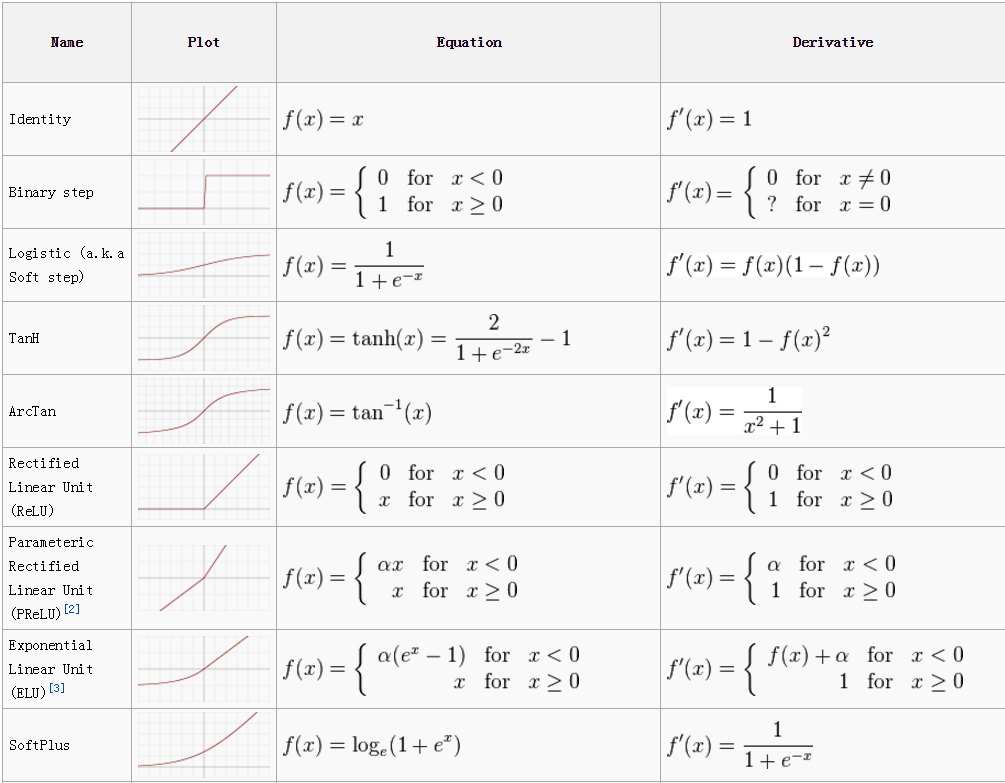
**IMPORTANTE: NA BIBLIOTECA TODA VEZ QUE PRECISA ALIMENTAR A REDE NEURAL TEM QUE SER NO FORMATO [1DADO X 10 INPUTS\_NEURONS]**

*O Neural Network é mais potente onde para processamento de imagens com 3000 iterações o logistic regression chegou em 81%, o neural network Sigmoid chegou em 91%, Tanh 95%, ReLU 96%, 2 layer ReLU 97%*

*Nem sempre quanto mais hidden layer melhor, por que quando mais layer mais treinamento e mais dados são necessários.*



## Activation Non-Linear Functions



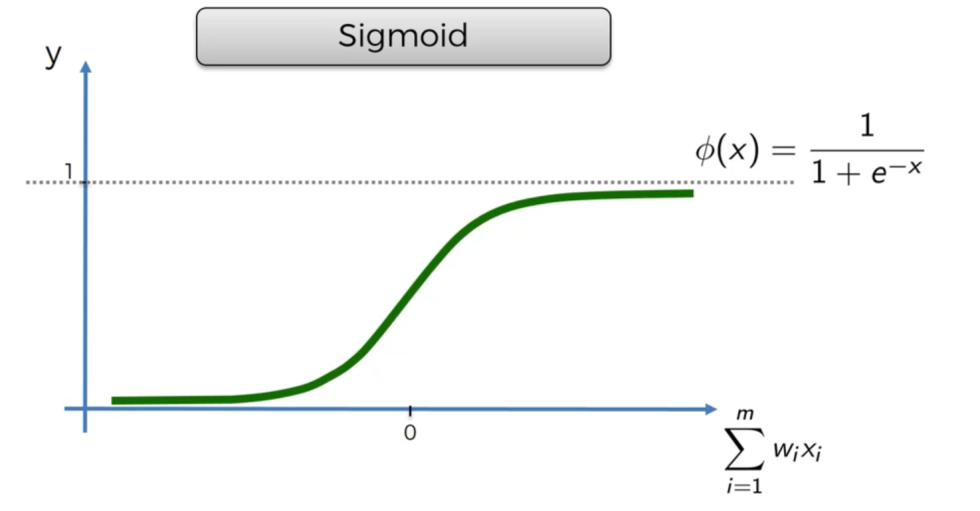
### - Threshold Function

Uma função de ativação com resposta binaria, onde tem de grau para determinar se enviar um sinal ou não dependendo do valor do peso das ligações(X) calculado pelo o neuron

### - Sigmoid Function:

Função logaritima tem a saida com progressão gradual. Dependendo do valor do peso das ligações(X) é calculado pelo o neuron a saída analogica e relativa. Quando a entrada menor de zero, a saída tende a ser zero. Quando a entrada é maior do que zero, a saída tende a ser perto de 1. **Esta função também é utilizada para prever probabilidades para a resposta ser 1**.

**Problema: Os pesos dos Synapses tem que ser inicializados para evitar a saturação da função de ativação em 0 ou 1, onde não terá mais aprendizagem.**



### -Tanh Function

É a versão modificada do sigmoid onde o 0 em Y é o centro referencial

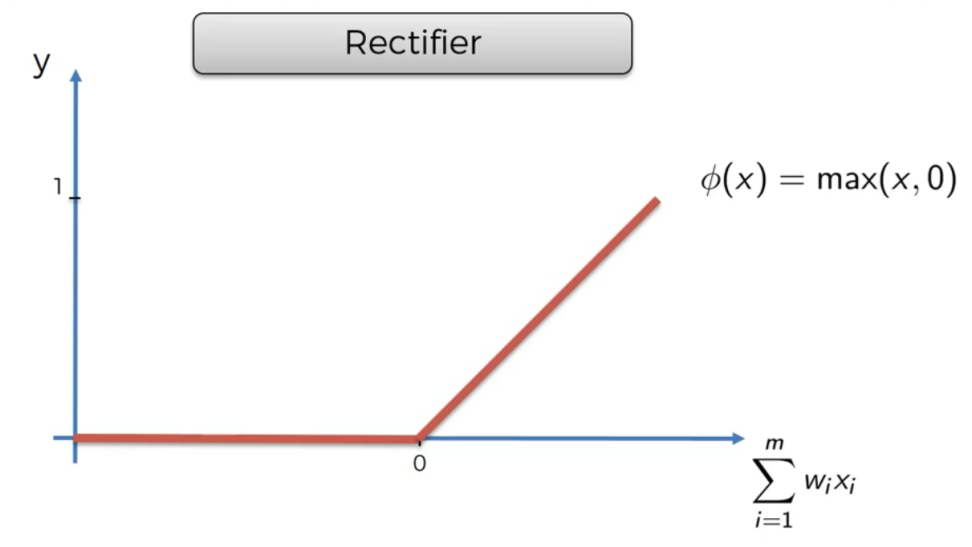
Onde é a função Sigmoid

**Problema: Os pesos dos Synapses tem que ser inicializados para evitar a saturação da função de ativação em 0 ou 1, onde não terá mais aprendizagem.**

### - Rectifier Function(mais utilizada):

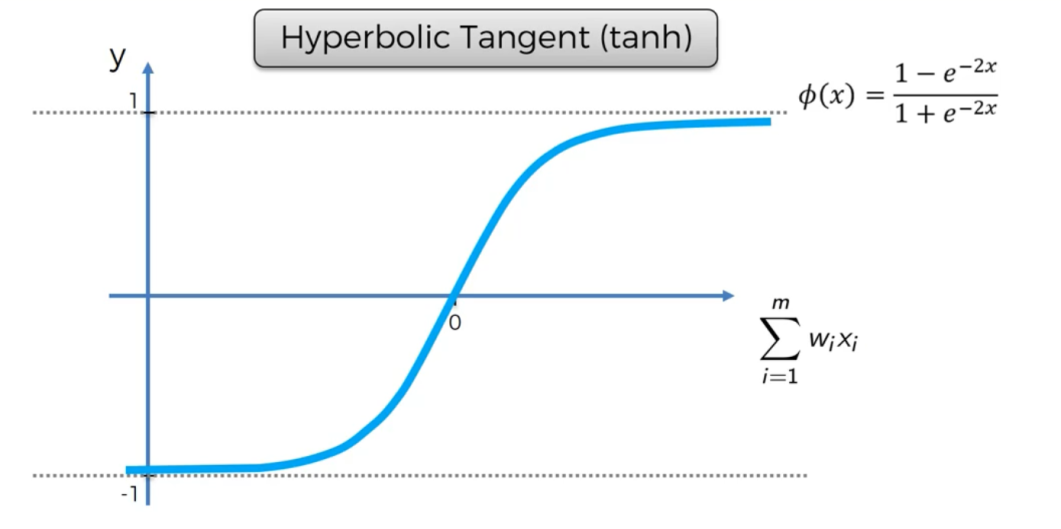
Função de ativação Retificadora é a função mais utilizada. Quando X menor ou igual a zero, a saída é zero. Qunado X é maior do que zero a saida é o valor X

**Nesta função tem que escolher o indice de aprendizagem com cuidado para não zerar o gradiente muito rápido, se isso acontece a conexão synapse morre.**



### - Hyperbolic Function:

Função similiar da sigmoid, mas tem o componente negativo e positivo.



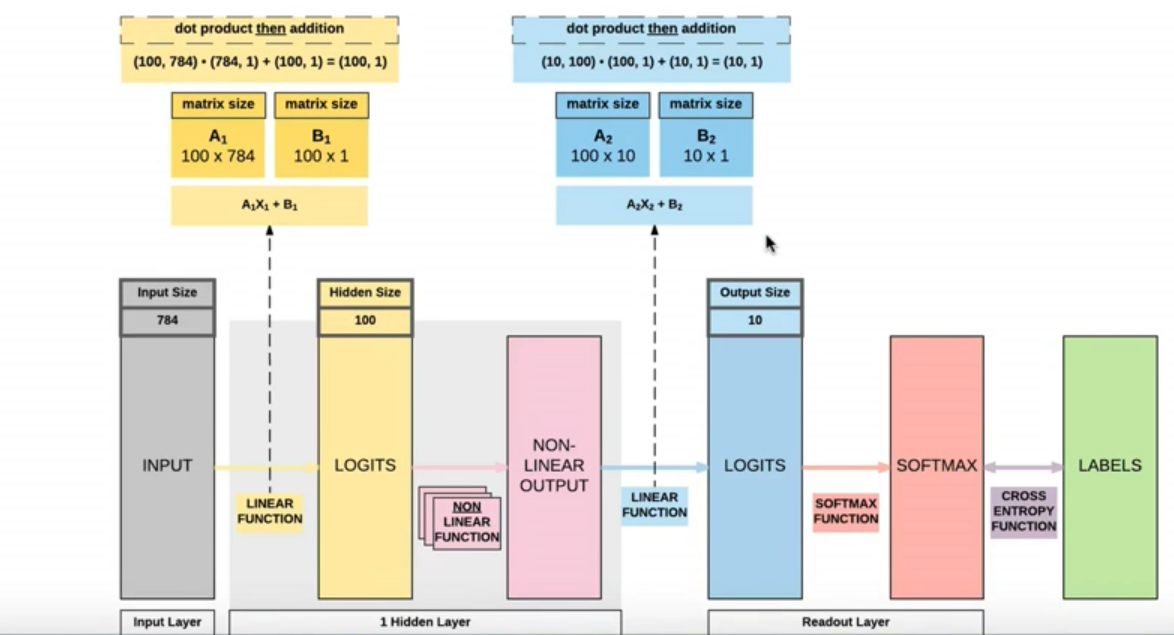
# Neural Network Parameters

**Para cada neuron na camada oculta 1 tem um função Linear predefinida , Os parametros São alfa e beta. O parametro alfa é um matrix de (numeroDeSaidas x numeroDeEntradas), beta é um matrix de (numero de saidas x 1)**.

- Na Função Linear 1: alfa é um matrix (100images x 784 pixels), beta(100 imagens \* 1)

X é entrada (784pixels x 1imagem), saida depois da operação (100x1).

- Na Função Linear 2: X(100x1)(saida da função Linear 1 com saida da função não linear na entrada da função linear 2), Alfa(10numeros x 100entradas), beta()



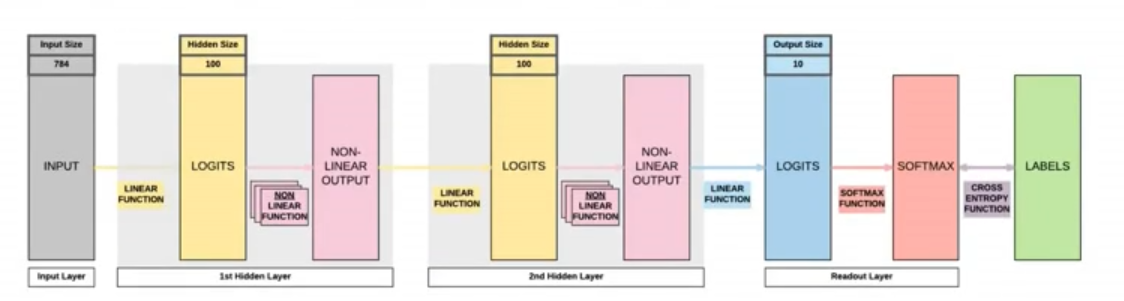
# Multiple Hidden Layers

Feedward Neural Network with 2 ReLu Layers (Um dos melhores que tem)

**Modelo:** Linear Ativation Function(Entrada - Camada Oculta1) -> Non-Linear Ativation Function(Camada Oculta1 - Relu1)

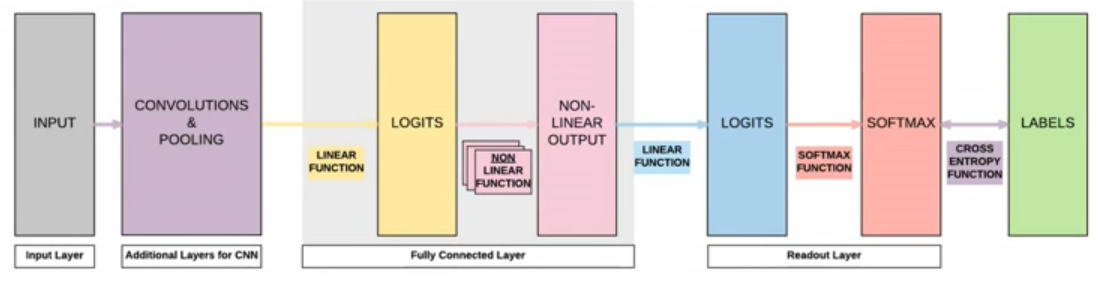
->Linear Ativation Function(Relu1 - Camada Oculta2) -> Non-Linear Ativation Function(Camada Oculta2 - Relu2)

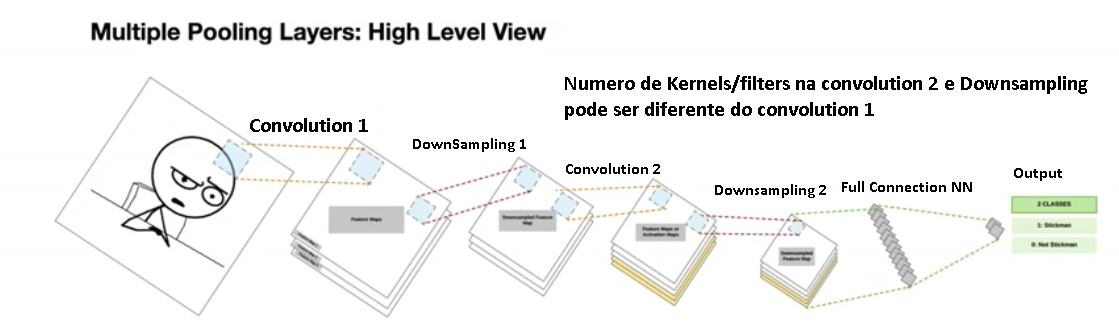
-> Linear Ativation Function (Relu2 - Saida final) -> Entropy Loss e Softmax

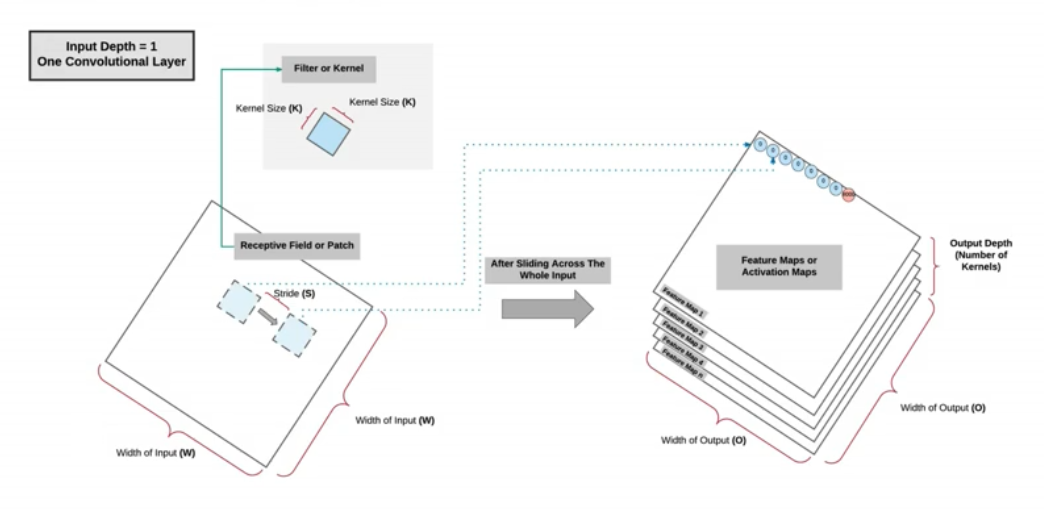


# Convolutional Neural Network

O convolutional neural network é identica rede neural convencional mas tem uma etapa de convolução e pooling que são utilizados para filtrar, detectar caracterisitca e reduzir o tamanho da imagem.



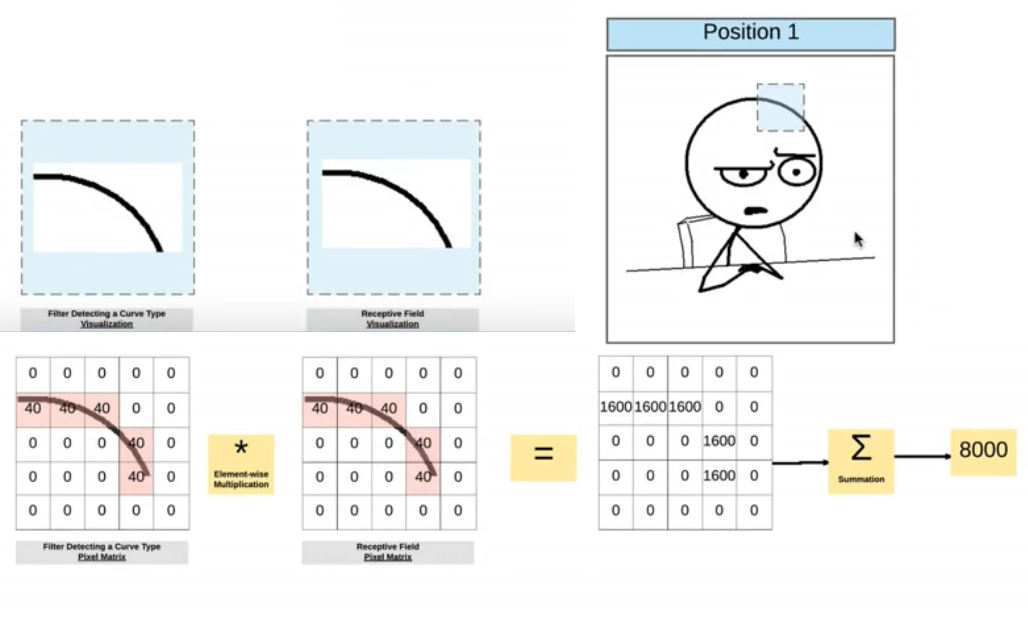




## Convolution process

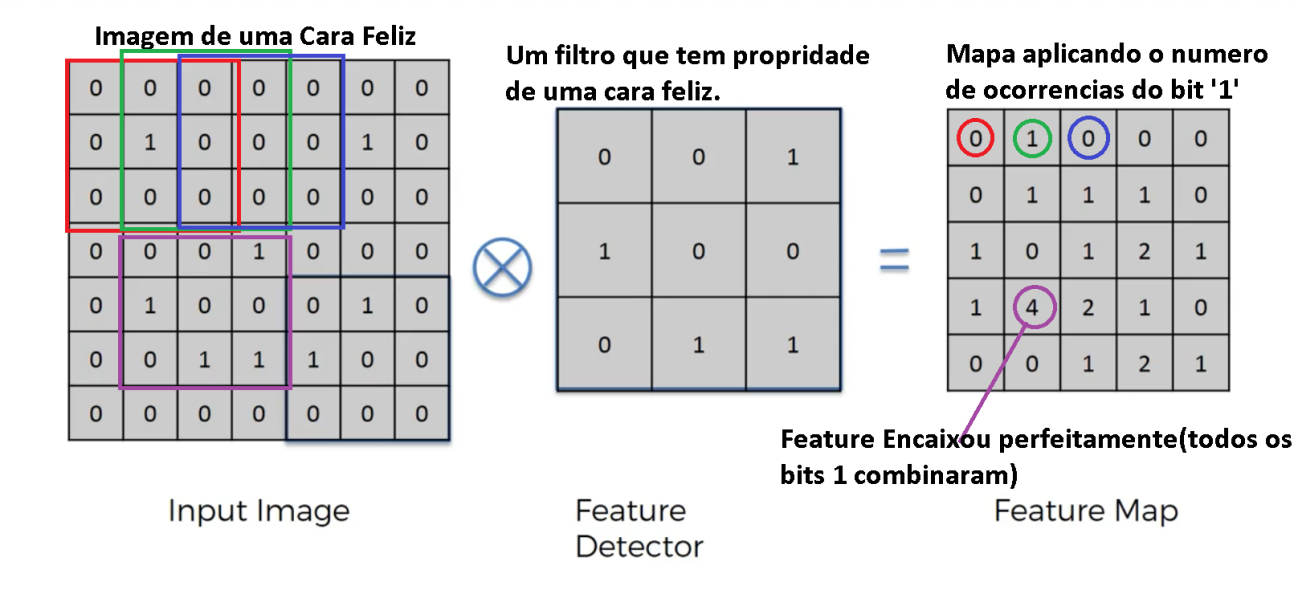
É o processo de utilização de um ‘Filter Detector/Kernel/Feature Detector’ que contém uma caracteristica só da imagem e depois convolucionar na imagem(passar na imagem toda multiplicando o filtro com o pedaço da imagem e depois fazendo somatoria do matrix criado). Este filtro detector serve para detectar caracteristicas como traços ou curvas para idêntificação da imagem. Estes filtros são construidos no processo de aprendizagem de convolutional neural network (CNN)

Toda vez que um filtro é passado na imagem toda, é criado um mapa de caracteristica “Feature Map/Activation Map” que contém o número de combinação de bits do filtro com o pedaço da imagem. Quanto mais filtros detectores mais mapas de caracteristicas são criadas e mais precisão para detecção da imagem tem.

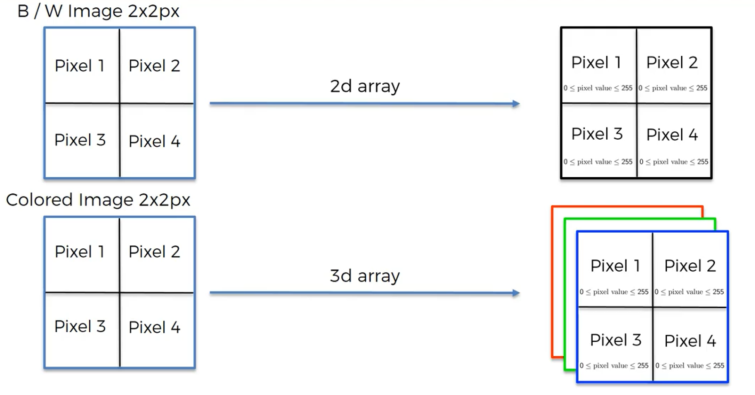


Neste processo a imagem é reduzida de tamanho, onde terá um processamento mais fácil e rápido. O Feature map é utiliza um matriz 3x3. Mesmo com a redução da imagem mas a perda é pequena. (Stride é quantas casas vai avançar a cada combinação, no caso abaixo Stride = 1)

**Depois de cada camada de convolução, é necessário passar por processo de não linearidade**



A imagem é um array 2D de pixels. Se a imagem for preto e branco tem somente 1 filtro detetivo de pixels (matriz com valores de o a 255) (input\_depth = 1) para contruir 1 Feature ma. Se a imagem for colorida, **Então terá 3 filtros Red, Green e blue para cada caracteristica** (input\_depth = 3) depois somados os feature maps dos 3 camadas RGB para o feature map principal.





**Calculo da dimensão do matriz de saida**

Onde:

O = Dimensão do matriz de saida

W: Dimensão do matriz de Entrada, se (4x4) W = 4

K = tamamho do Filtro (Kernel), se (3x3) K = 3

P = Tamaho do Padding

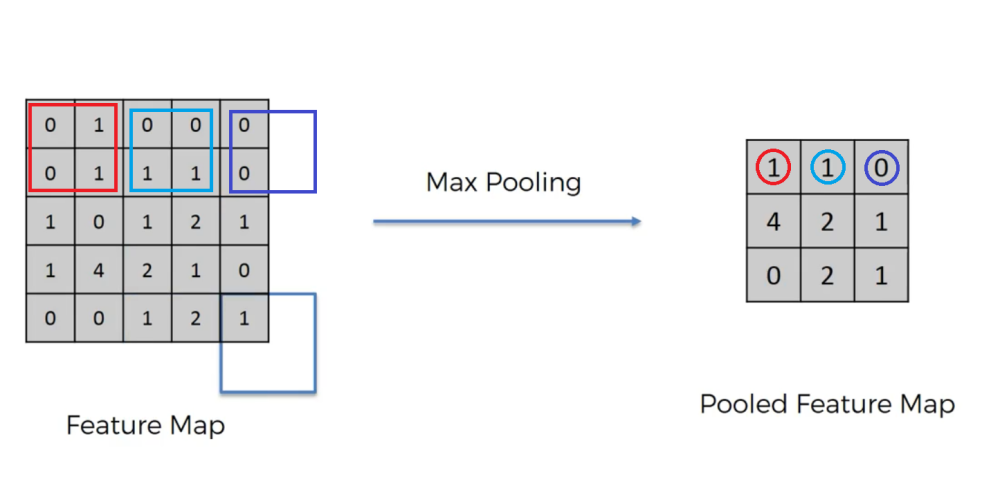
S = casas que o filtro avança a cada combinação (Default Stride = 1)

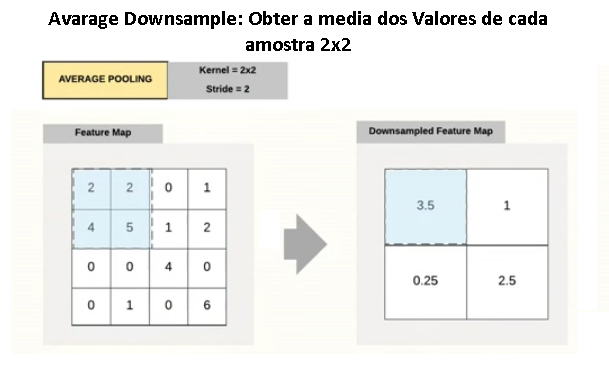
## Max Pooling/ Down Sampling

Esta técnica é utilizada para detectar um caracteristica na imagem mesmo se está redimensionada, com outros planos de fundo, rotacionada, etc... Também serve para diminuir o tamanho do matrix e facilitar o processamento

Para isso é colocado uma caixa 2x2 ou mais no ‘Feature Map’, agrupamento de 2x2 ou mais e obtendo somente o maior numero de combinações de bits do feature map.

Existe outros tipos de pooling como avg pooling/subsampling para calcular o valor medio de cada caixa 2x2 ou mais





**Calculo da dimensão do matriz de saida depois do down sampling**

Onde:

O = Dimensão do matriz de saida

W: Dimensão do matriz de Entrada, se (4x4) W = 4

K = tamamho do Filtro (Kernel), se (3x3) K = 3

## Padding

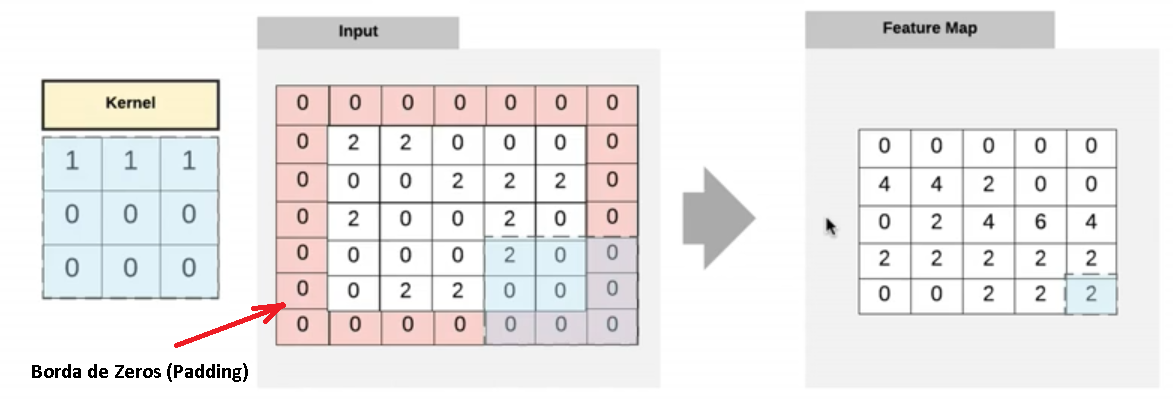
Toda vez que não utiliza padding na imagem (zero padding) no processo de convolução/pooling a imagem fica menor. Quando deseja efeituar convolução/pooling sem diminuir o tamanho da imagem, então é colocado bordas de numero 0 chamado de Padding.

**O Padding normalmente é utilizado na etapa de convolution para não diminuir o tamanho do matriz da imagem processada, para não perder muita informação, A redução de tamanho e informação normalmente é feito na etapa de pooling/Downsampling.**

**Max padding é a técnica mais eficiente do que avg padding ou outros tipos de padding.**

Valid padding (Zero Padding): Output < Input

Same padding: Output = Input



**Calculo de quantas Bordas Padding São necessárias para ter Input = Output**

Onde:

P = Tamaho do Padding

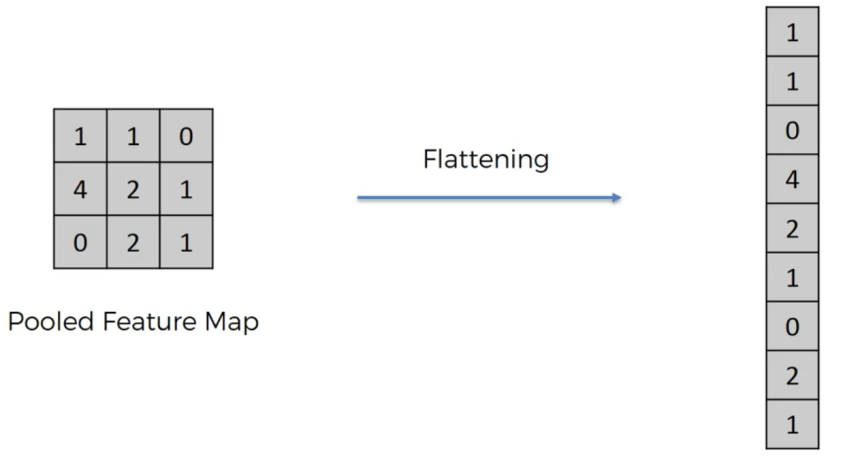
K = tamamho do Filtro (Kernel), se (3x3) K = 3

## Flatting

Colocar o matrix como uma linha para colocar na rede neural. Este processo feito no metodo forward() out = self.maxPool2(out, -1). O processo de flatting converte o matriz (100, 32, 7, 7) para matriz (100,32 \* 7 \* 7)

**IMPORTANTE: Então nao precisa tornar o matrix 100x1x28x28 para 100x784 na etapa de treinamento, porque o formato é utilizado no CNN e depois é convertido no processo de flatting antes de passar para nn full connection.**

**IMPORTANTE: NA BIBLIOTECA TODA VEZ QUE PRECISA ALIMENTAR A REDE NEURAL TEM QUE SER NO FORMATO [1DADO X 10 INPUTS\_NEURONS]**

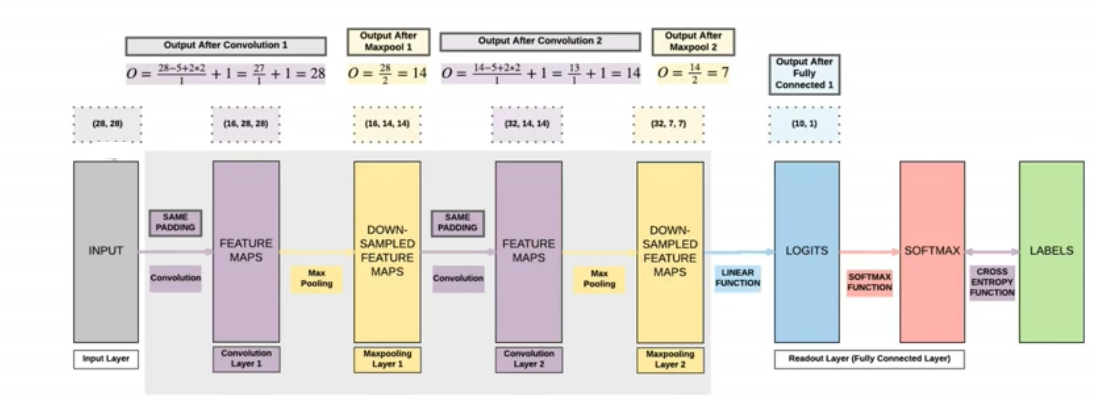


## Full Connection

Neste processo é colocado a linha na rede neural e aplicado forward propagation e no backward propagation o peso dos synapses e Feature Detector é atualizado para aprender mais para detectar a imagem. No processo de aprendizagem é determinado os neurons com maior somatória para um objeto especifico, Estes neurons são definidos para o neuron especifico para determinar o objeto desejado.



# Convolutional Neural Network para processamento de Imagem



Calculo do matriz depois da Convolução 1

O1 = (W-K+2P)/S+1 = [(28 - 5 + 2 \* 2)/1] + 1 = 28

O = Dimensão do matriz de saida

W = Dimensão do matriz de Entrada, Neste Caso(28x28) W = 28

K = tamamho do Filtro (Kernel), Neste Caso (5x5) K = 5

P = Tamaho do Padding P= (K-1)/2 = (5-1)/2 = 2

S = casas que o filtro avança a cada combinação (Default Stride = 1)

Calculo do Matrix depois da Pooling/Downsampling 1

O = W/K = 28/2 = 14

O = Dimensão do matriz de saida

W = Dimensão do matriz de Entrada, Neste Caso 28

K = tamamho do Filtro (Kernel), Neste Caso (2x2) K = 2

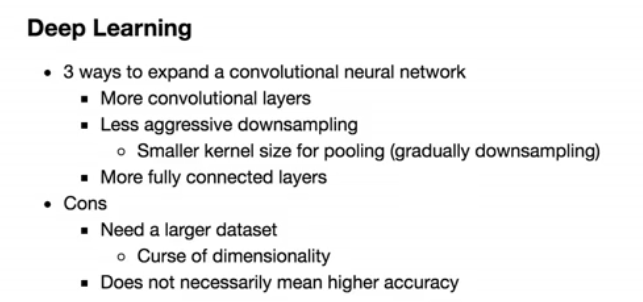
Calculo do matriz depois da Convolução 2

O2 = (W-K+2P)/S+1 = [(14 - 5 + 2 \* 2)/1] + 1 = 14

Calculo do Matrix depois da Pooling/Downsampling 2

O = W/K = 14/2 = 7

Modelo: Entrada (Imagem 28x28) -> Convolution 1 (16(nº kernal escolhido)x28x28) -> Downsampling1 (16(nº kernal escolhido)14x14) -> Convolution 2 (32(nº kernal escolhido)14x14) -> Downsampling 2 (32(nº kernal escolhido)7x7) -> Flatting -> Full Connection 2(32(nº kernal escolhido)7x7)



# Exemplo: DoomCorridor-v0 (by [@ppaquette](https://github.com/ppaquette))

This map is designed to improve your navigation. There is a vest at the end of the corridor, with 6 enemies (3 groups of 2). Your goal is to get to the vest as soon as possible, without being killed.

Goal: 1,000 points

* Reach the vest (or get very close to it)

Rewards:

* Plus distance for getting closer to the vest
* Minus distance for getting further from the vest
* Minus 100 pts for getting killed

Ends when:

* Player touches vest
* Player is dead
* Timeout (1 minutes - 2,100 frames)

Allowed actions:

* ATTACK
* MOVE\_RIGHT
* MOVE\_LEFT
* MOVE\_FORWARD
* TURN\_RIGHT
* TURN\_LEFT

-Neste jogo o Bot inicialmente vai tomar decisões aleadoras processar e vai obter uma sequencia de imagens e recompensas,

-Então vai processar a sequencia e vai aprender a melhor maneira para ganha

- Mais camadas convolution, mais feature detector terá, e mais precisão terá também. (mais treino precisa).

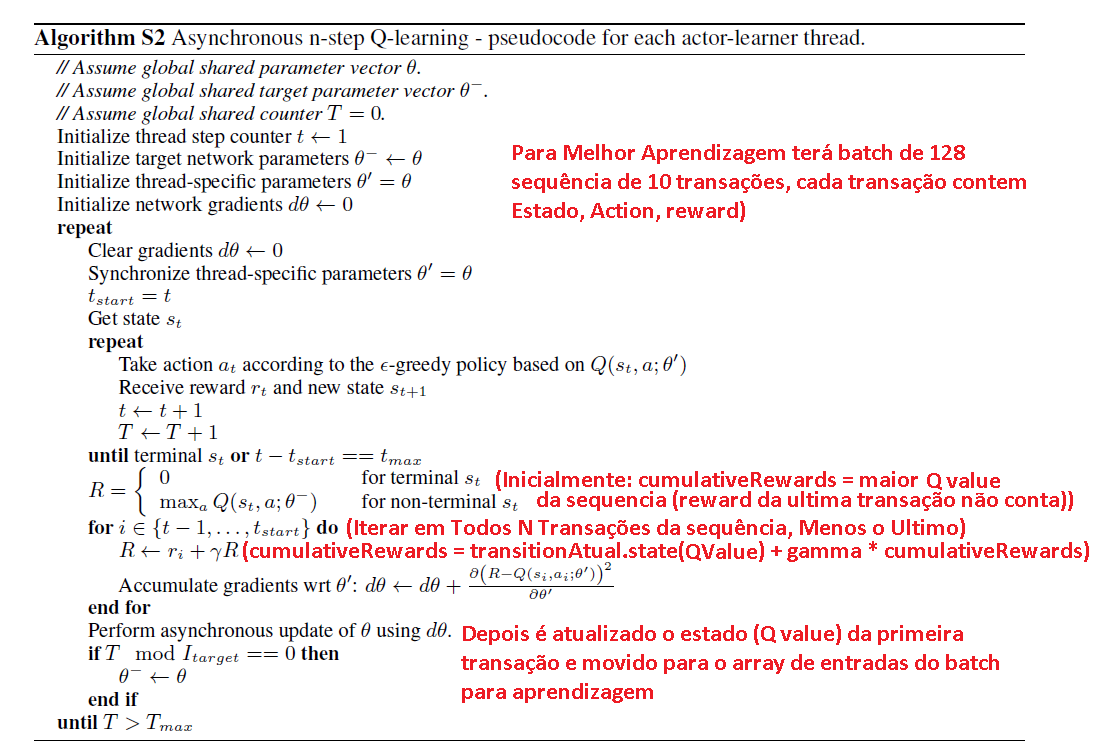
-O Bot terá olhos (images preprocessing) - brain (IA module) - body (Actions)

# N Steps Eligibility Trace:

É uma técnica de acumular os rewards de varias etapas (N Steps) e depois o valor Q é aprendido deste acumulo de rewards (diferente da técnica normal que aprende depois de cada transição). Essa técnica é muito eficiente .

Implementando Algoritimo de Eligibility Trace Q Learning(Processar a cada N Steps(Transações) e aprender do primeiro movimento de cada sequencia)

#ele aprenderá somente do primeiro movimento de cada sequência de 10 movimento do batch. Para o primeiro movimento é calculado o Q-value do Q learning equtaion dos 10 movimentos da sequência (Mapeamento do Enviroment). Então opara o primeiro movimento de cada sequência tem o "Distino calculado" via o Q Learning



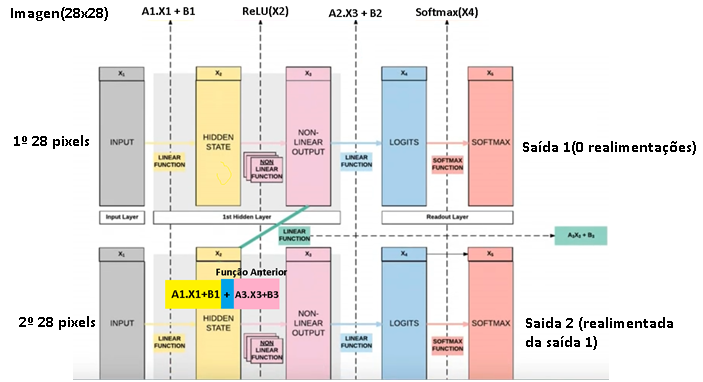
# - Recurrent Neural Network

Recurrent neural network (RNN) basicamente é uma rede neural (NN), onde a primeira entrada é obtida dos neurons de entrada, mas depois **cada Função linear** vai depender da função logaritima do ultimo resultado(T-1), onde a função logaritima anterior vai somar com a função linear atual

**Entrada: cada sequencia RNN insere 28 pixels da imagem(28x28), #Cada etapa das 28 etapas é inserido somente 28 pixels da imagem(28x28) e realimentado a função linear da função não linear da etapa anterior.**

**Então na saida tem [100 imagens, 28 etapas, 100 dados da camada oculta], o que importa e a ultima etapa que contem todas as realimentações anteriores para cada imagem então é filtado o matrix para obter [100 imagens x 100 (dados da camada oculta da ultima etapa)]**

**No exemplo de processamento de imagem vai passar 28 vezes Linear -> Non-Linear -> Linear ->Softmax -> Linear +NonLinear (do anterior)) -> Non Linear -> Linear -> Softmax....**



Outro exemplo para 2 camadas RNN

