**Deep Learning**
Using neural network architectures with multiple hidden layers of neurons to build generative as well as predictive models.

# SUMMARY : DEEP LEARNING USING Flux.jl

By : Madhav Sharma/PseudoCodeNerd

**Flux.jl**
A Machine Learning Library in Julia that doesn't tensor ! **Install :**

```
] add Flux
```

## Introduction

### 1. to Flux.jl

To use **Flux** in your code, start by running

```
using Flux
```

Automatically create neurons using **Dense** (2I, IO)

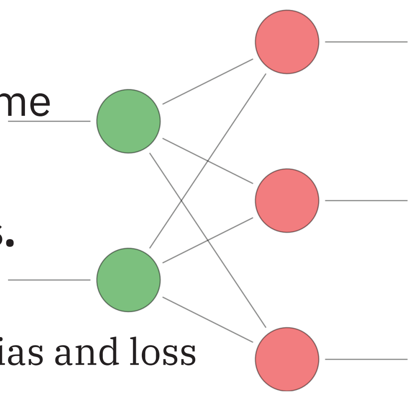
```
model = Dense(2, 1, σ)
```

Flux contains many helpful built-in funcions like :
- σ : which is the sigmoid activation function
- `Flux.mse` : the mean squared error (loss) f()
- `Flux.train!`: to train the model on the data

Defining model --> defining loss funtion --> setting optimiser to params of Grad. Desc. --> train --> predict!

### 2. to Neural Networks

When two neurons are connected in such a way that their outputs are used to predict the end result it is a **Neural Network.**
We build our model the same way even when we have **multiple inputs & outputs.**



We just define our weights, bias and loss in the form of matrices and vectors and perfrom **Matrix Multipication.**

## Continued.

$$\sigma(x; w, b) = \begin{bmatrix} \sigma^{(1)} \\ \sigma^{(2)} \\ \vdots \\ \sigma^{(n)} \end{bmatrix} = \frac{1}{1+\exp(-Wx+b)}$$

Here's how we can write our equation collectively
Here **Wx** is the dot product between our weights & x
We use `Flux's` **`onehot`** function to encode categorical integer features to boolean vectors in our data.

### 3. Deep Neural Networks

When two neurons are chained in such a way that **one neuron's output is other's input**, it is a **Deep Neural Network.**

At every layer, a non-linear function is applied to the data and it is this **function** which helps us to **deal with non-linearly seperable data** and **make accurate predictions.**

Flux provides **`Chain`** to connect Dense layers.

**The Core Algorithm (Not Deep).**

```
model = Chain(Dense(2,4,σ),Dense(4,3,σ))
L(x,y) = Flux.mse(model(x),y) #loss fun.
opt  = SGD(params(model))
Flux.train!(L, zip(xs, ys), opt)
# xs & ys are the training data (!IMP)
```

## Continued.

We can also use **batching to improve the efficiency** of our model.

```
databatch = (Flux.batch(xs), Flux.batch(ys))
```

Flux provides **`batch`** to **combat cumbersome matrix multiplications.**

The softmax function is often used in the final layer of a neural network to **normalise output** into a probability distribution.

```
model = Chain(Dense(2,4,σ),
              Dense(4,3,identity),
              softmax)
L(x,y) = Flux.crossentropymodel(x),y)
opt  = SGD(params(model))
```

### 3. Neural Network on MNIST

(L) Fig : Sample MNIST images.

1. Getting data from `Flux.Data.MNIST (0:53)`
2. Create a vector of feature vectors `(imgs+labels)`
3. Setting up our Neural Network : `(7:04)`
   a) Input : Vectors $\mathbf{x}^{(i)}$ so it has ***n*** nodes,
   b) Output : Size 10 (0->9 nums) | One-hot vector encoding digit from 0 to 9 using softmax.

```
model = Chain(Dense(n_inputs,n_outputs,identity),softmax)
```

4. Training `(Flux.train!) (8:26)`
5. Testing `(17:20)` Note : Try addding addnl. layers :-)

**Footnotes :**  Thanks to Julia Academy and Matt Bauman for the Course and shoutout to Logan !