





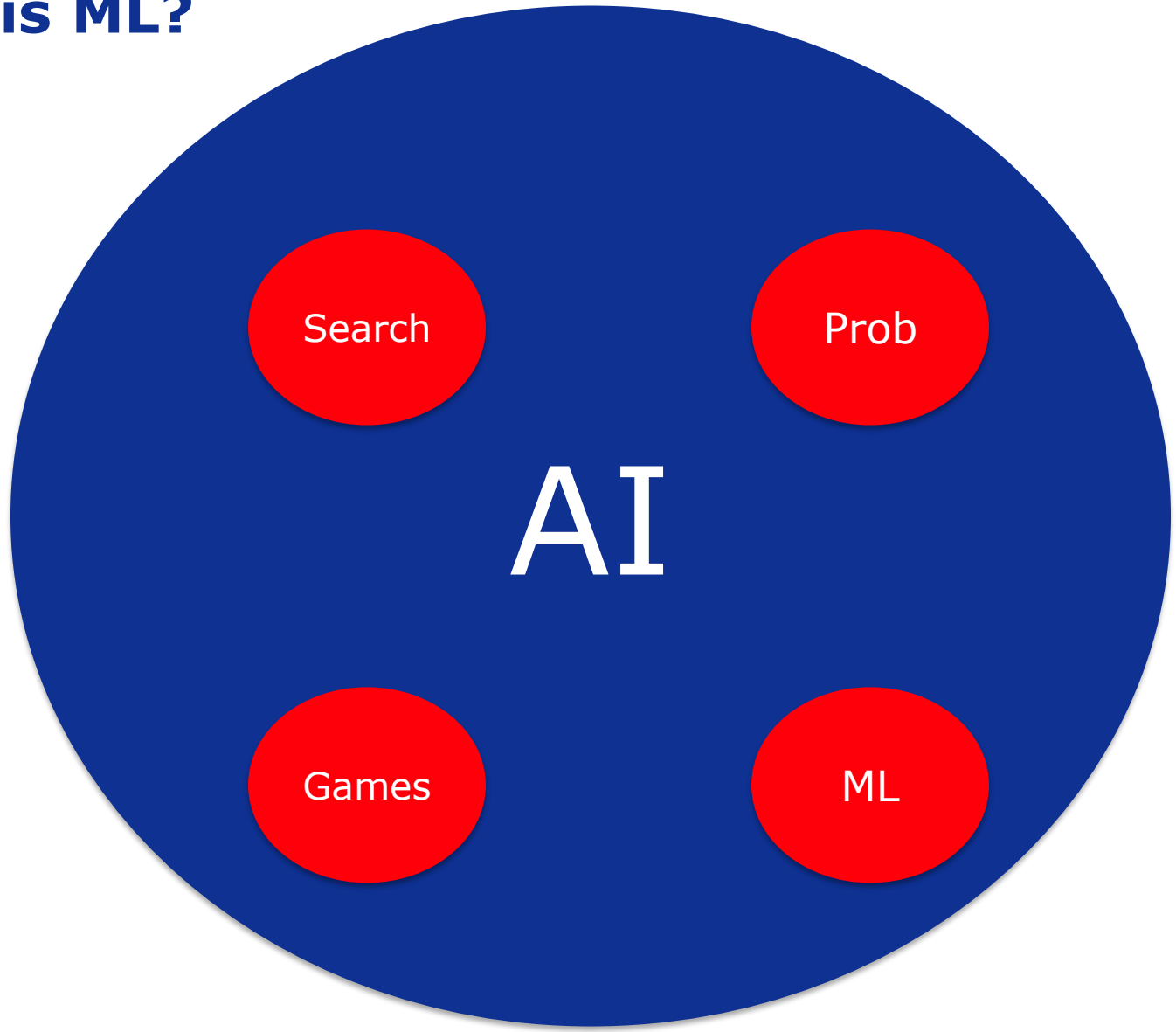

# Artificial Intelligence - Module 2 - Machine Learning

*Georges Sakr - ESIB*

# Day 1: Digit Recognition

**Georges Sakr**
ESIB

# What is ML?



# Teaching the machine by examples

# Trash Problem


Paper

Metal

Plastic

# Done at ESIB

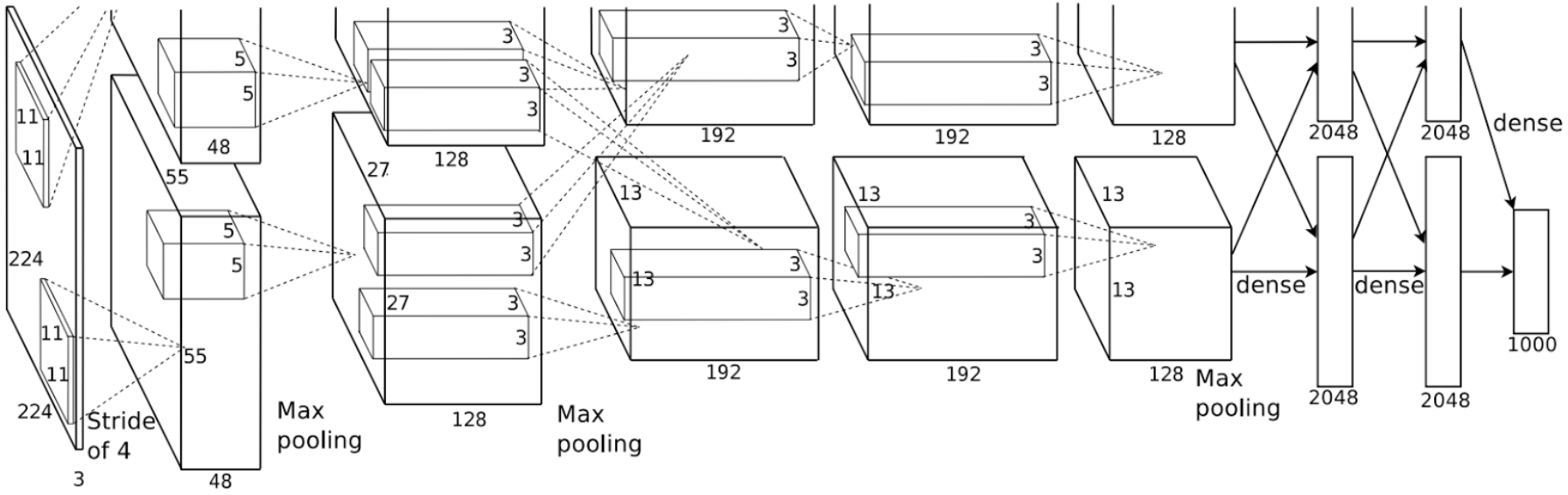
- Use AI to recognize the type of waste only from its picture.
- Implement the algorithm on a raspberry pi
- The model wi stic, Paper and Metal
- Achieve a higl
- Achieve fast p tion time < 1s)

# Data Collection

The waste pictures were taken by a Pi Camera connected to a Pi 3.

- 256 x 256 lossless .png images
- The mechanical system contains a small chamber in which the waste is initially placed.
- The chamber is lit 10 LEDS (1watt each)
- When the chamber door opens, the LEDS turn on for 5 seconds, during which the picture is taken as soon as the door closes.
- A total of 2000 pictures were collected

# Convolution Neural Networks



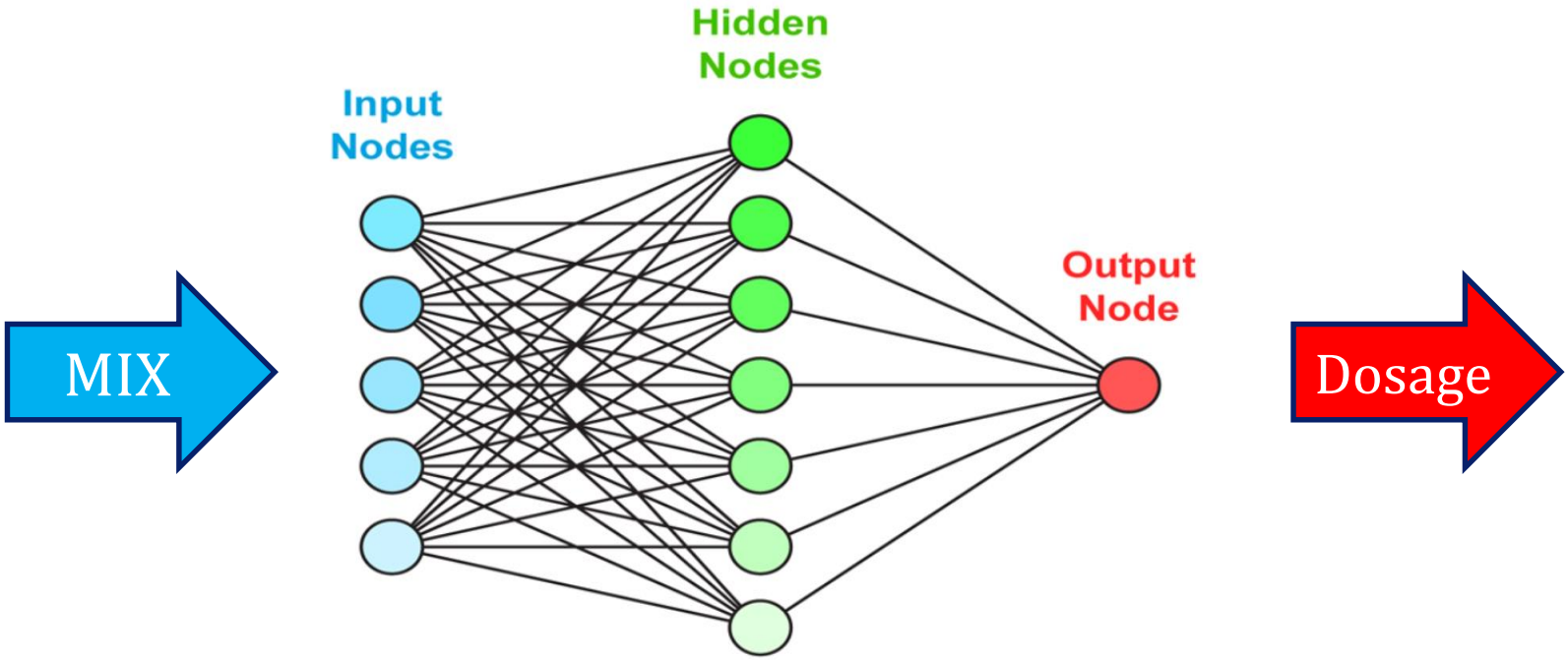

**AlexNet**

# Blood Thinning Drug Dosage

- **Problem**
  - The unpredictability of Acenocoumarol dose needed to achieve target blood thinning level to patients who have undergone heart bypass surgery

- **Solution**
  - Use demographic and physiological features (age, gender, BMI, BSA…)
  - Add to it some lifestyle habits (Alcohol, Smoking)
  - Mix them with genotyping results for CYP2C9 and VKORC1

# Solution – Artificial Neural Networks



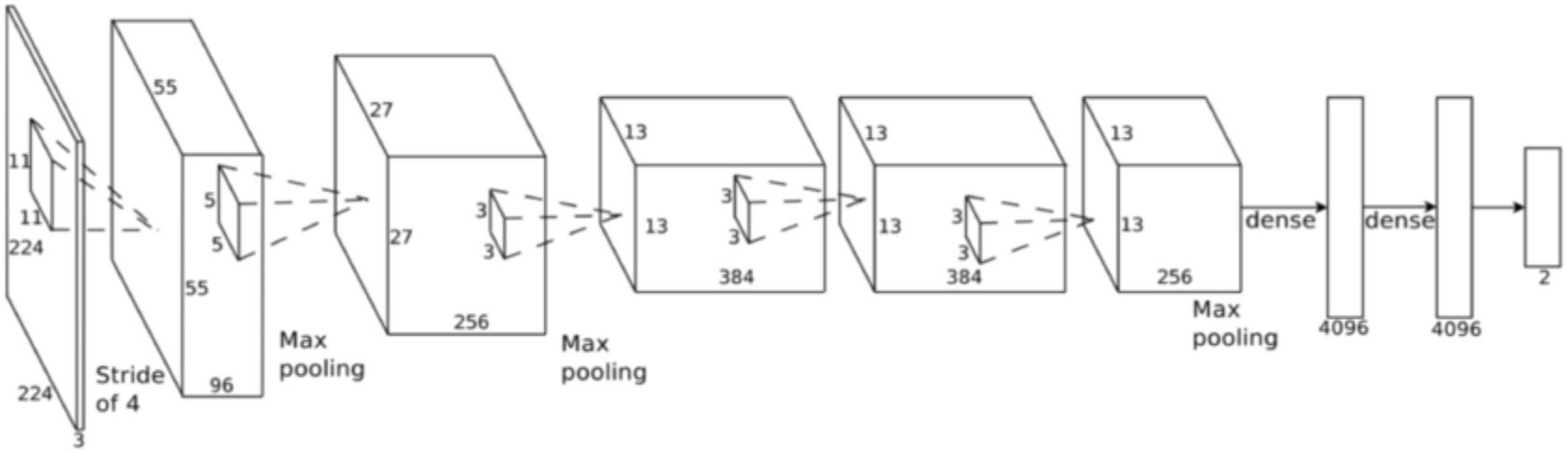
# Prediction Model Results (174 patients)

- Error of 7mg/week

# Arabic Font Recognition




# Deep Network CNN



# Accuracy

- Accuracy of 80% was achieved.

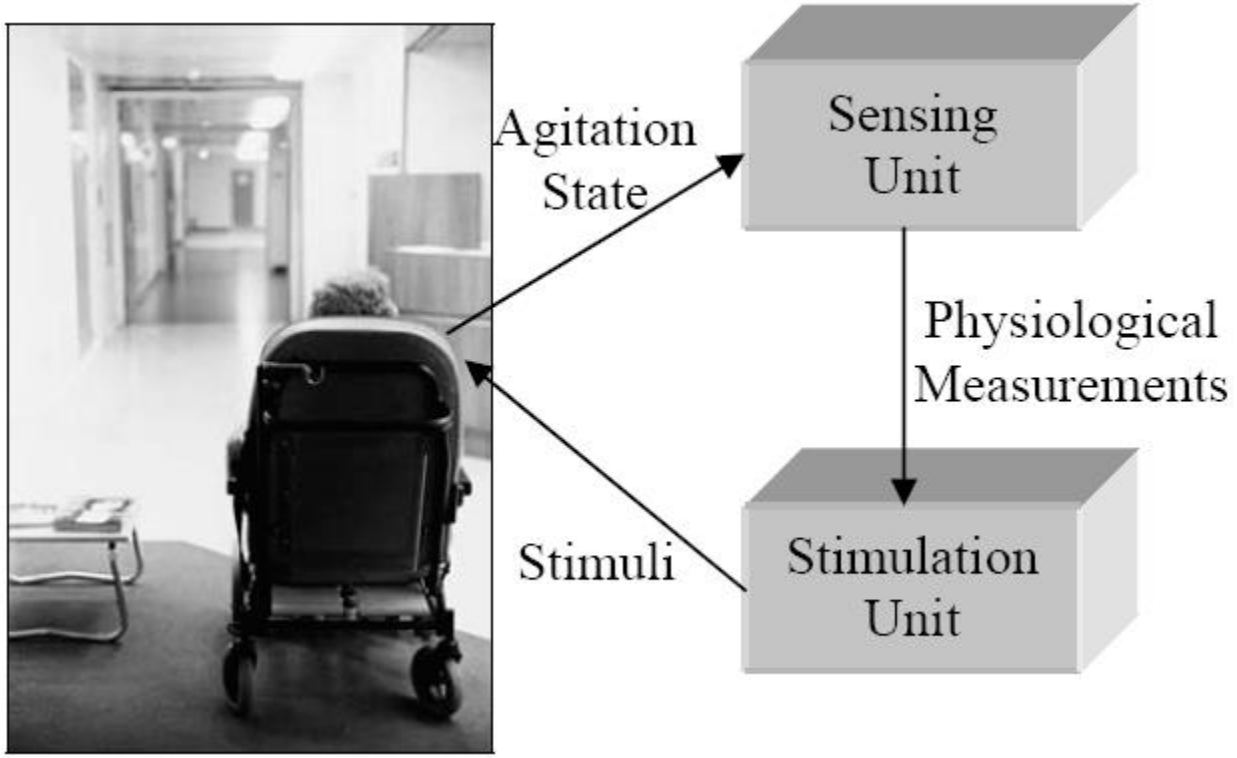
- Problem with 2 fonts that were very similar and the network did not learn to differentiate between the,

# Agitation in dementia Patients

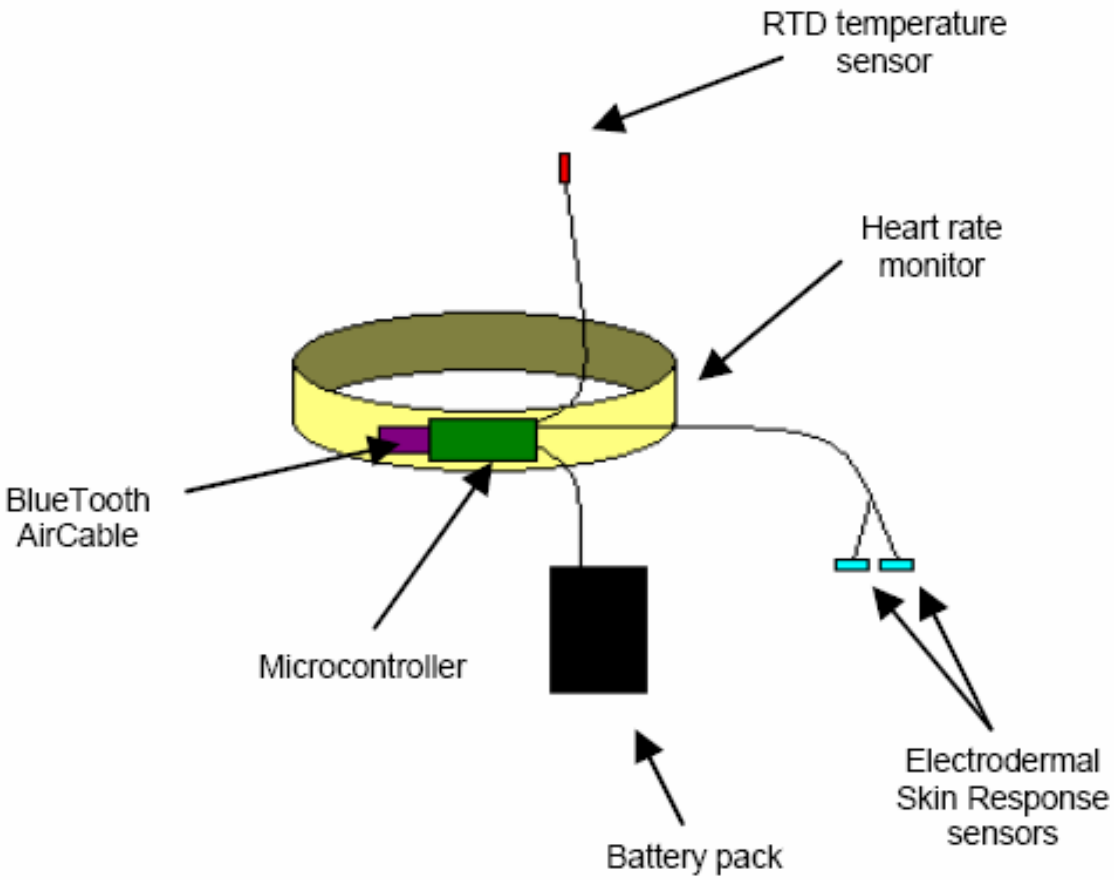

- **Problem**
  - People suffering from dementia (Alzheimer's) get agitated for no apparent reason.
  - Medical studies have shown that multi-sensory stimulation help patients to calm down.

- **Solution**
  - Use skin temperature, skin conductivity and heart rate variability
  - Train a support vector machine model to detect agitation.
  - Control the ambient environment to help patients to calm down.

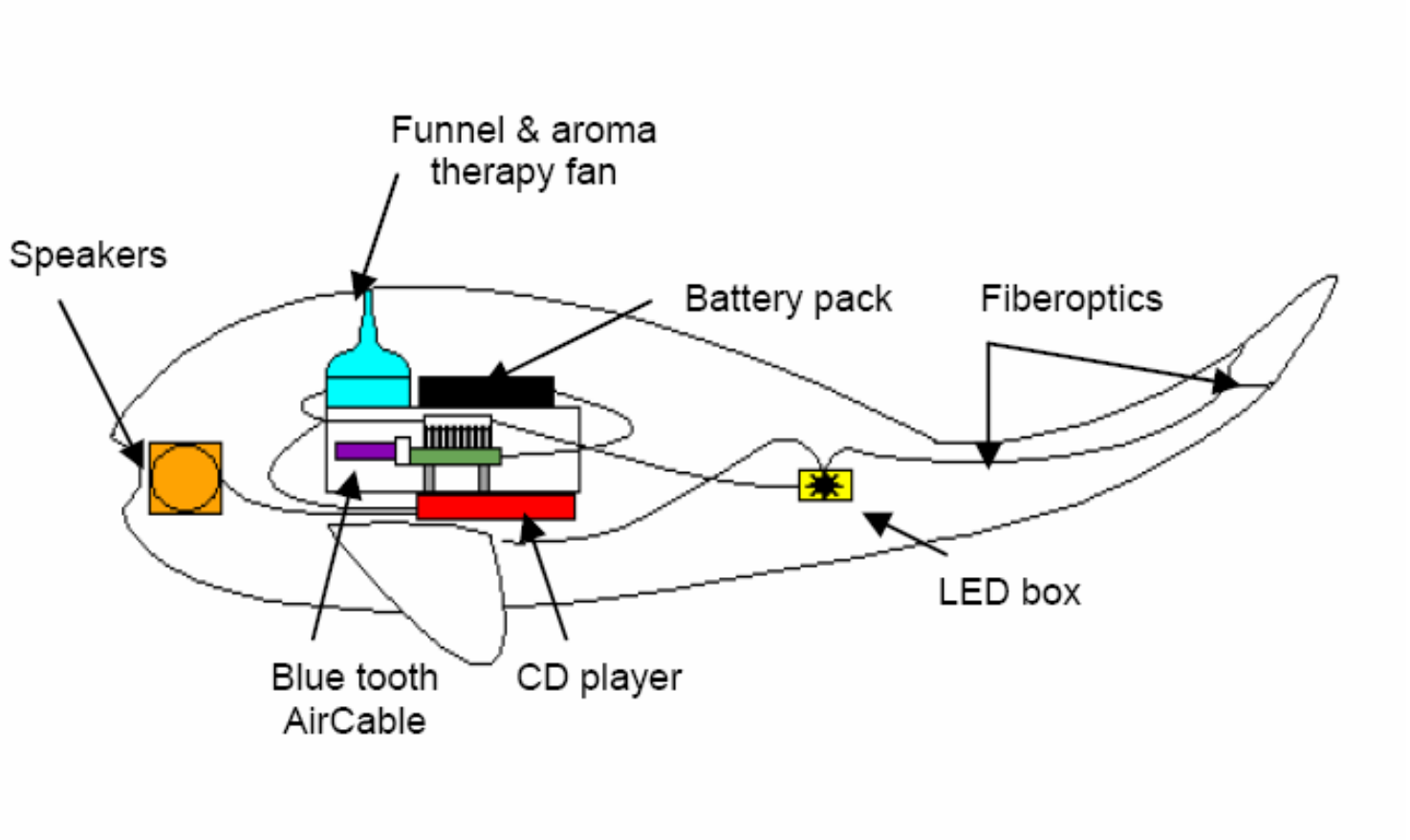
# An Autonomous Multi-sensory Intervention Device for Persons with Dementia

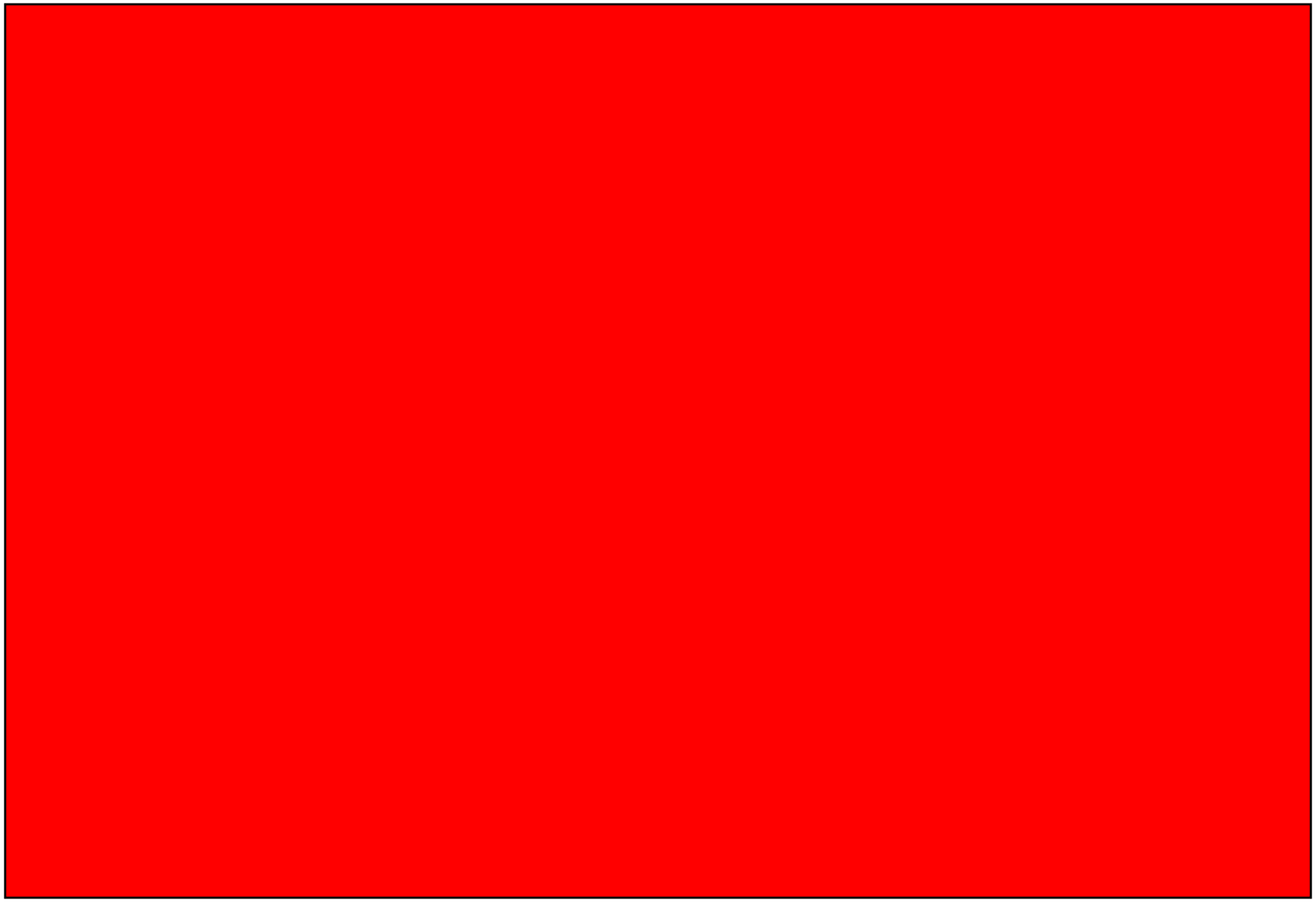


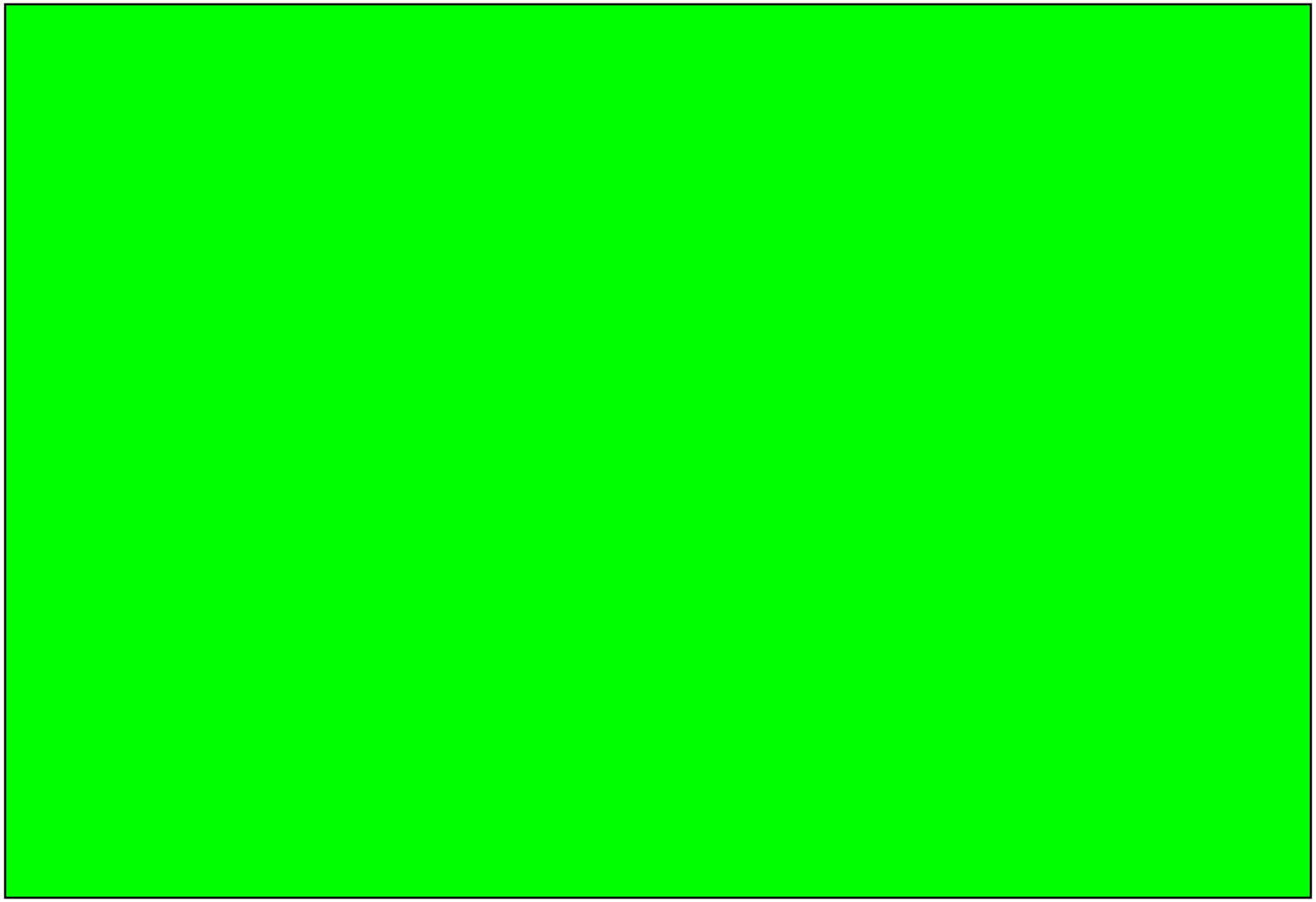
# Sensing Unit

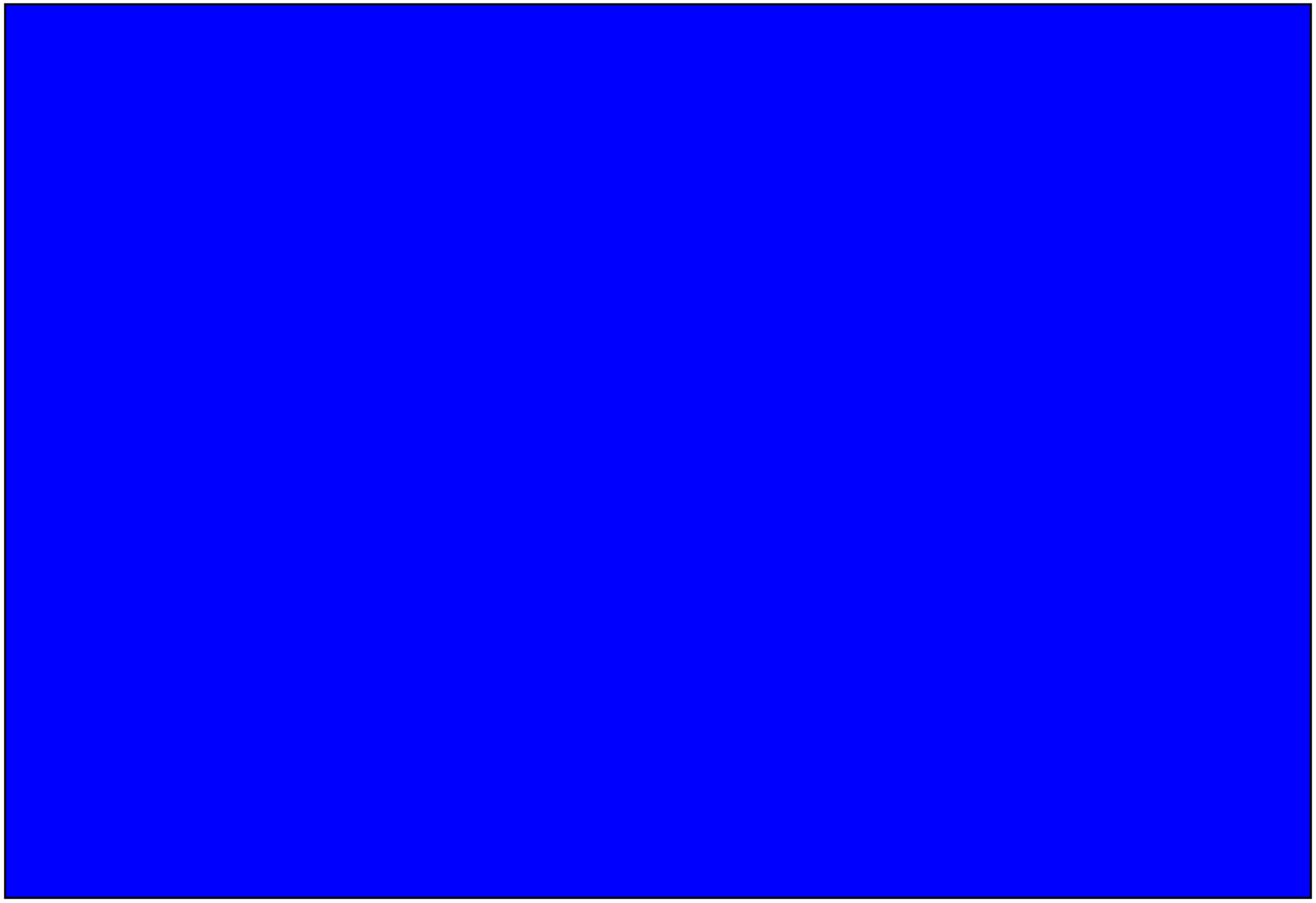


# Multi-Sensory Stimulation Unit









RED

GREEN

PINK

RED

# GREEN

PINK

# Data

- Three sets of PowerPoint slides were used to conduct the STROOP test
- The test included 60 randomly set color blocks (1min) where subjects are to name out loud the color they see
- 60 randomly set congruent word slides (1min) where subjects are to read out loud the word they see
- 120 randomly set incongruent word slides (2min) where subjects are to name the color of the word they see (not read the word)

# Results

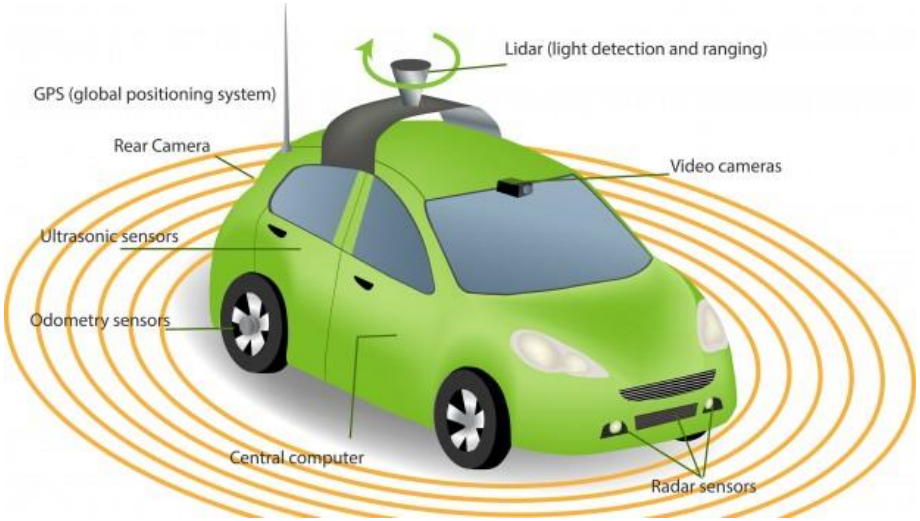
**Average accuracy of 95.1% achieved**

# Hypertension-SALT

- **Problem**

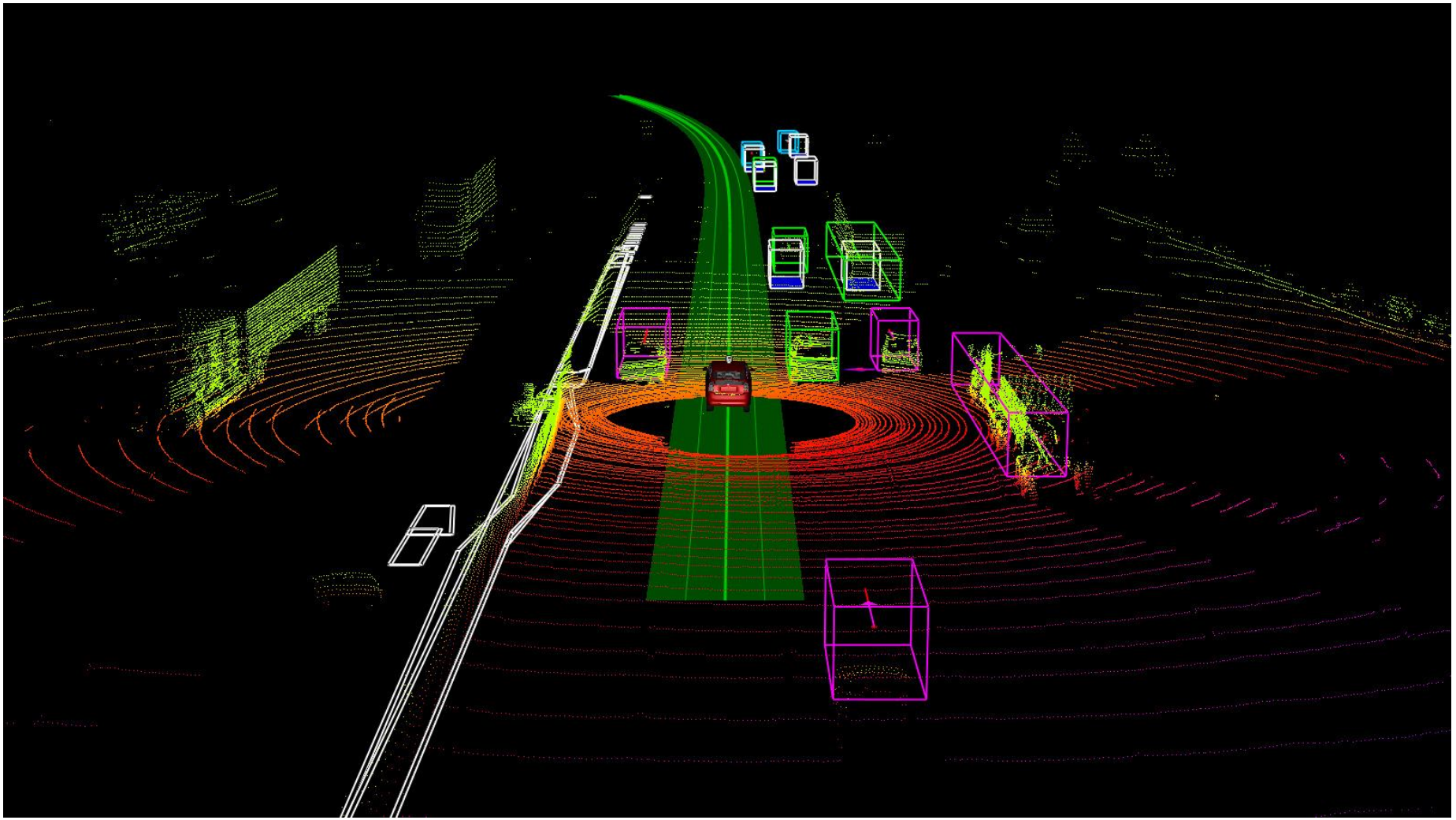
  High dietary salt intake is directly linked to hypertension and cardiovascular diseases. Predicting behaviors regarding salt intake habits is vital to guide interventions and increase their effectiveness.

# Road Asset Detection in self-driving cars




- Radar: Radio Detection And Ranging
  - To detect moving objects using doppler effect.
- LiDAR
  - To detect static objects using amount of energy returned by road assets
- In this research we focus on static objects classification using LiDAR data.

# Lidar



# Setting up the environement

```
conda create --name ML19 tensorflow-gpu

activate ML19

pip install keras
```

# Workshop Outline

- Neural Networks Foundations, teaches the basics of neural networks.

- Deep Learning with ConvNets introduces the concept of convolutional networks. It is a fundamental innovation in deep learning that has been used with success in multiple domains, from text to video to speech, going well beyond the initial image processing domain where it was originally conceived.

- Generative Adversarial Networks introduces generative adversarial networks used to reproduce synthetic data that looks like data generated by humans. And we will present WaveNet, a deep neural network used for reproducing human voice and musical instruments with high quality.

# Workshop Outline

- Word Embeddings, discusses word embeddings, a set of deep learning methodologies for detecting relationships between words and grouping together similar words.

- Recurrent Neural Networks – RNN, covers recurrent neural networks, a class of network optimized for handling sequence data such as text.

- AI Game Playing teaches you deep reinforcement learning and how it can be used to build deep learning networks with Keras that learn how to play arcade games based on reward feedback

# Outline

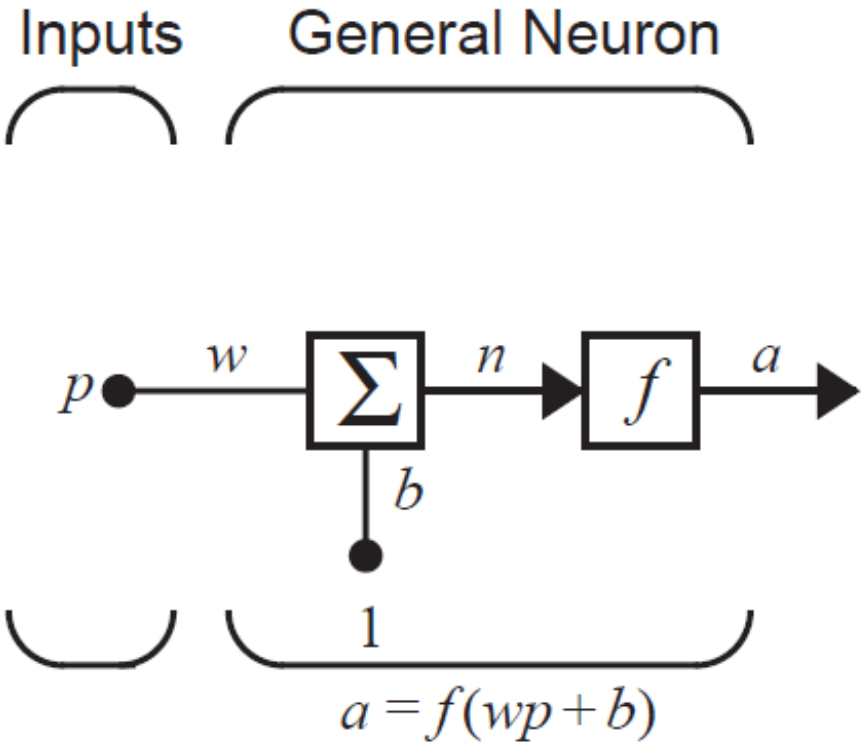
- Perceptron
- Multilayer perceptron
- Activation functions
- Gradient descent
- Stochastic gradient descent
- Digit Recognition

# Perceptron

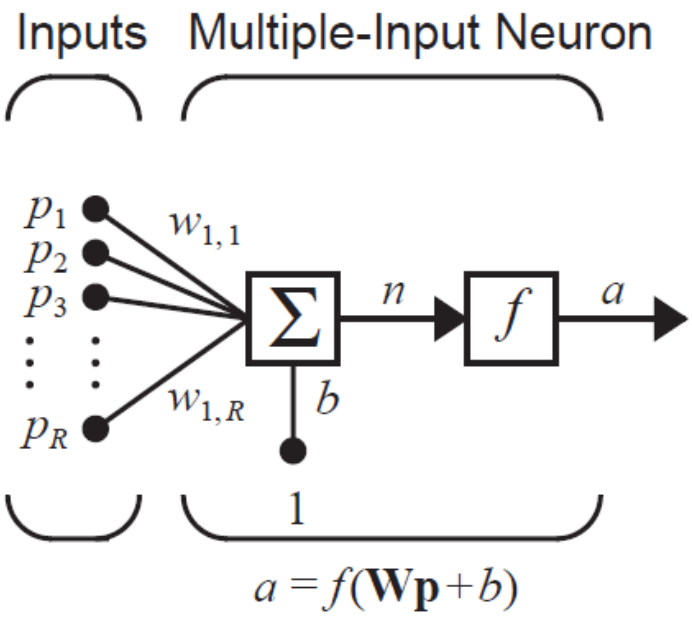
- The perceptron is a simple algorithm which, given an input vector *x* of *m* values (*x1*, *x2*, ..., *xn*) often called input features or simply features, outputs either *1* (yes) or *0* (no). Mathematically, we define a function:

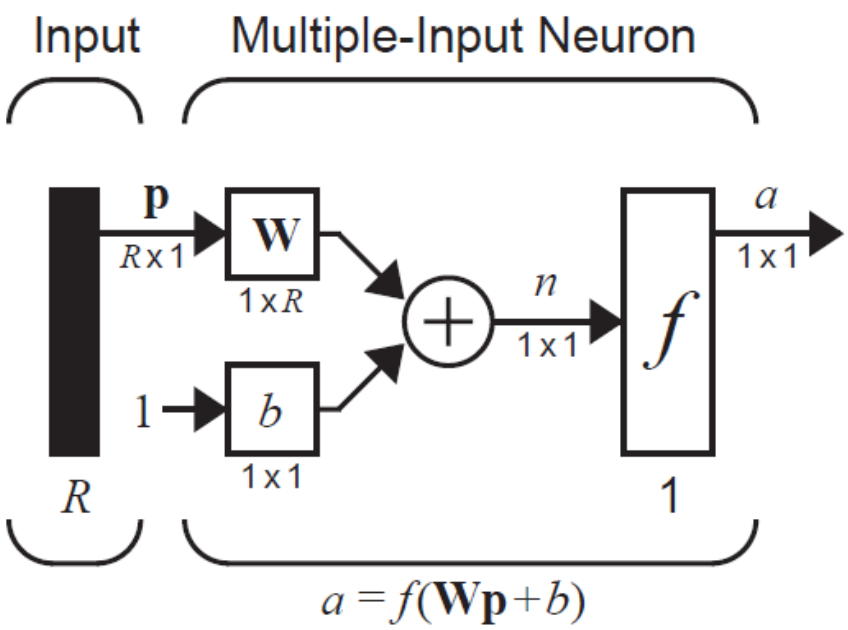
$$f(x) = \begin{cases} 1 & wx + b > 0 \\ 0 & otherwise \end{cases}$$

# 1 input 1 neuron



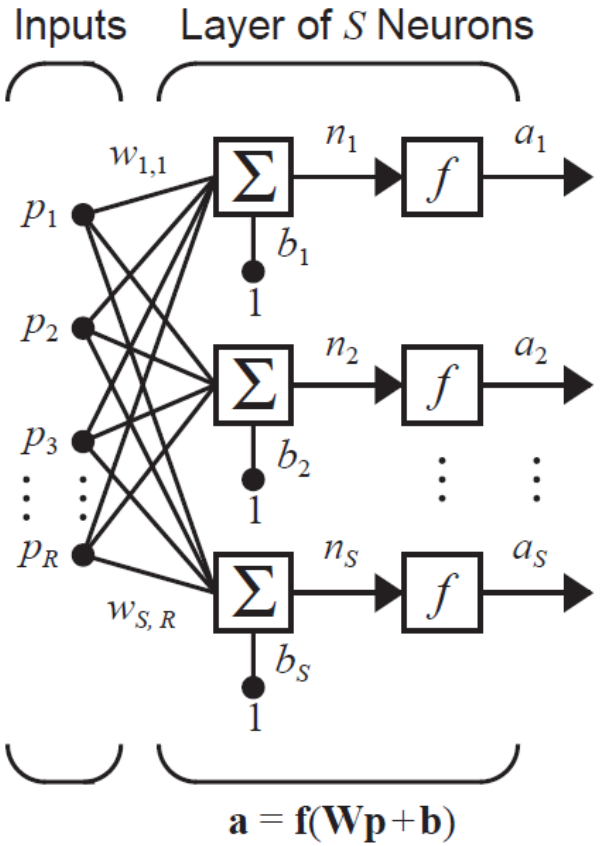
# Multiple input – 1 neuron



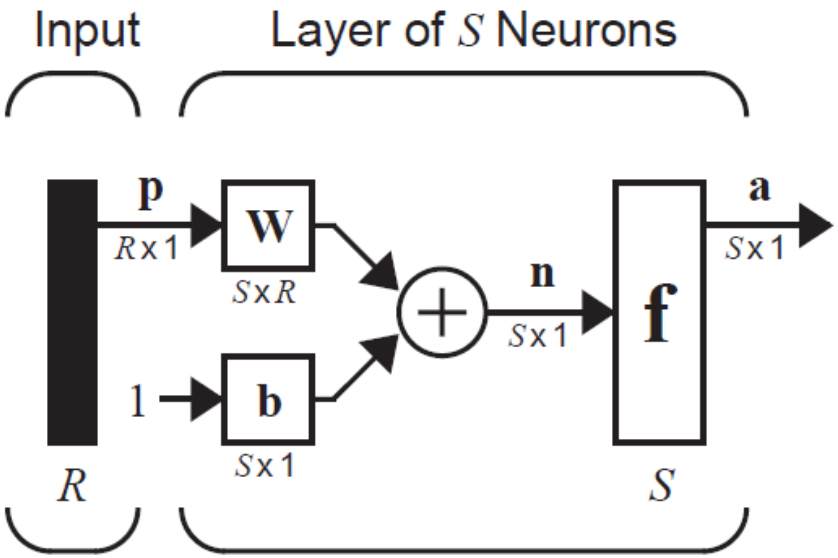




Abreviated Notation

# Layer of neurons



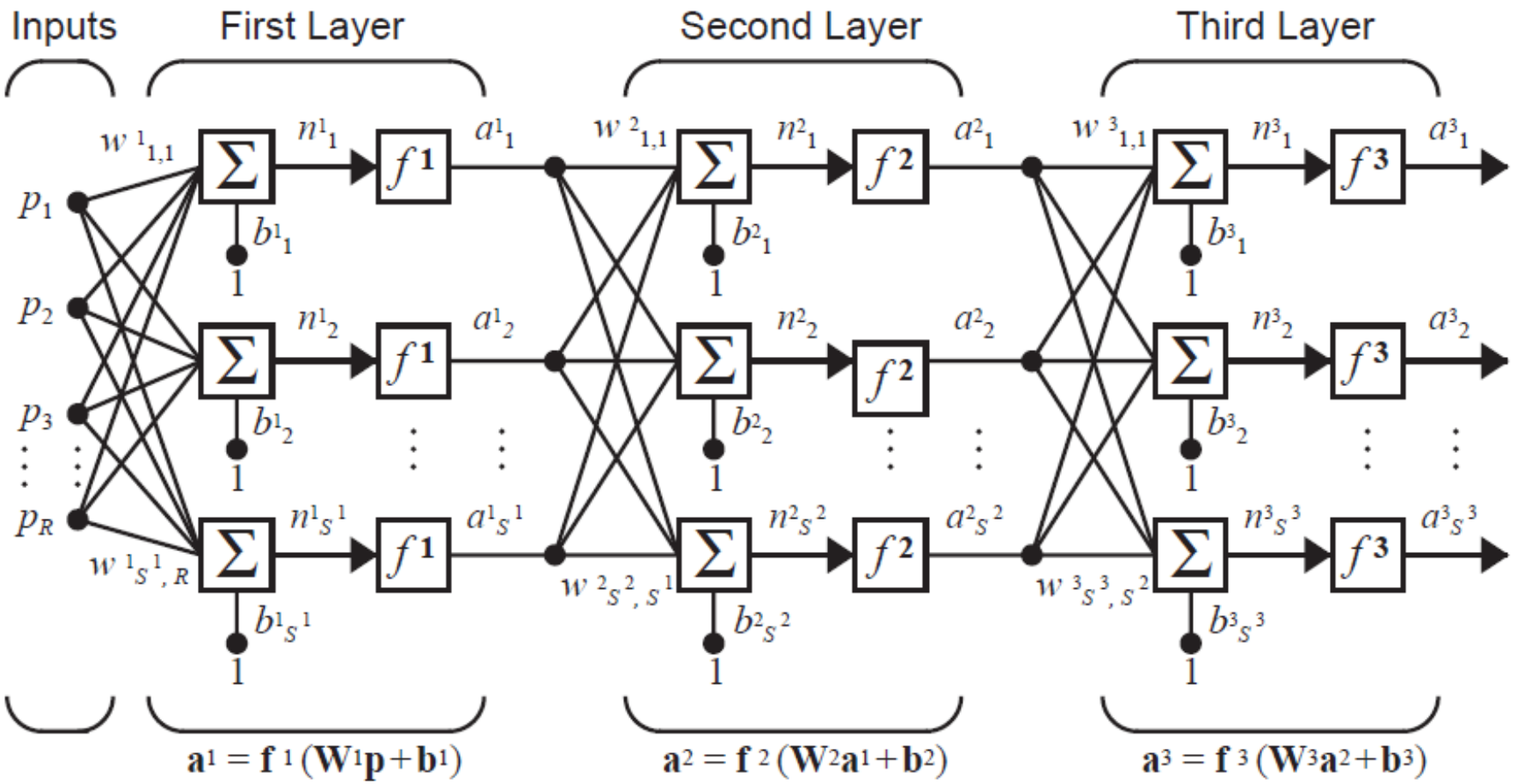
# Abbreviated Notation




$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

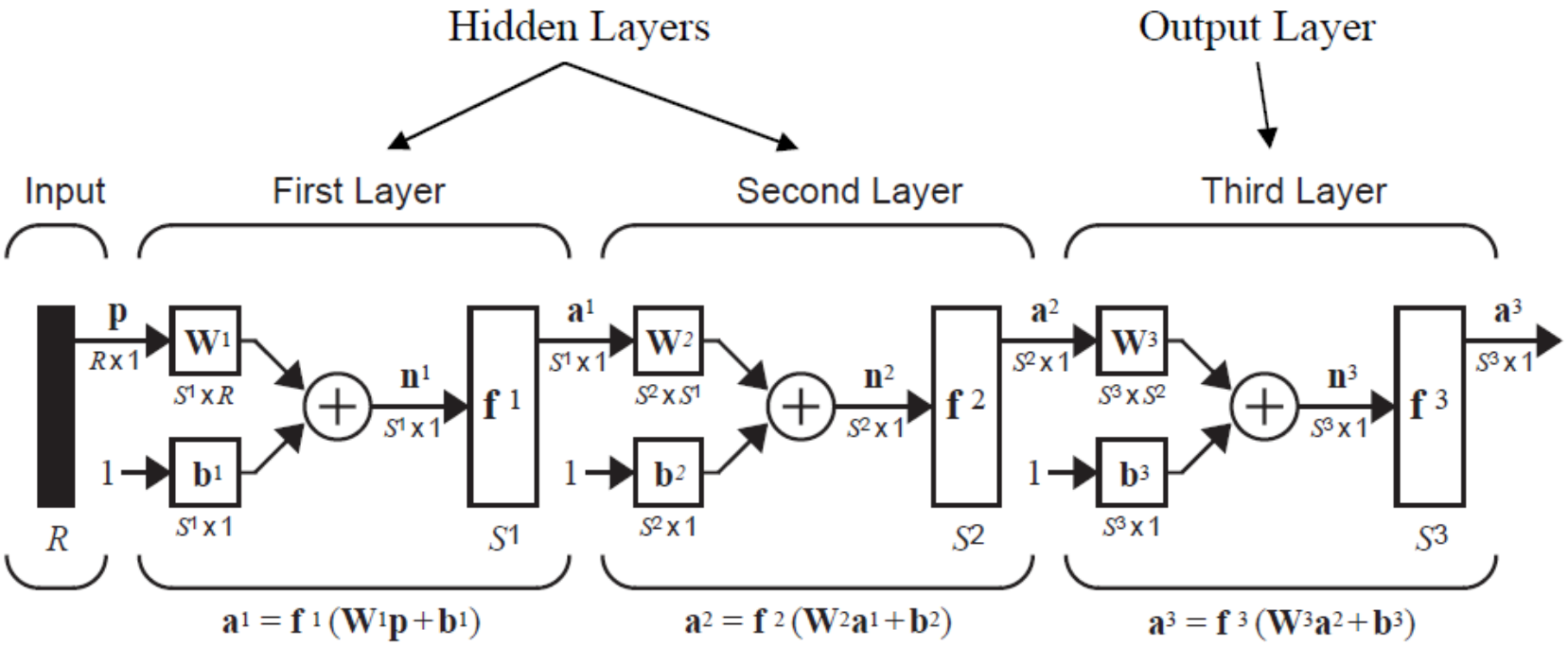
$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

# Multilayer Perceptron Network

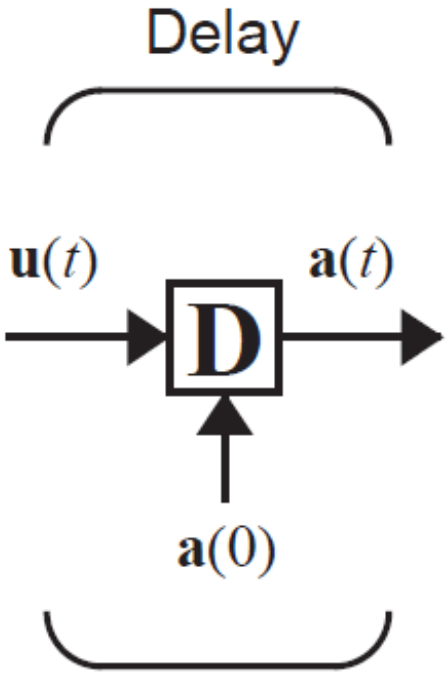



$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

# Abbreviated Notation



# Delay




$$\mathbf{a}(t) = \mathbf{u}(t - 1)$$

# Single Layer in Keras

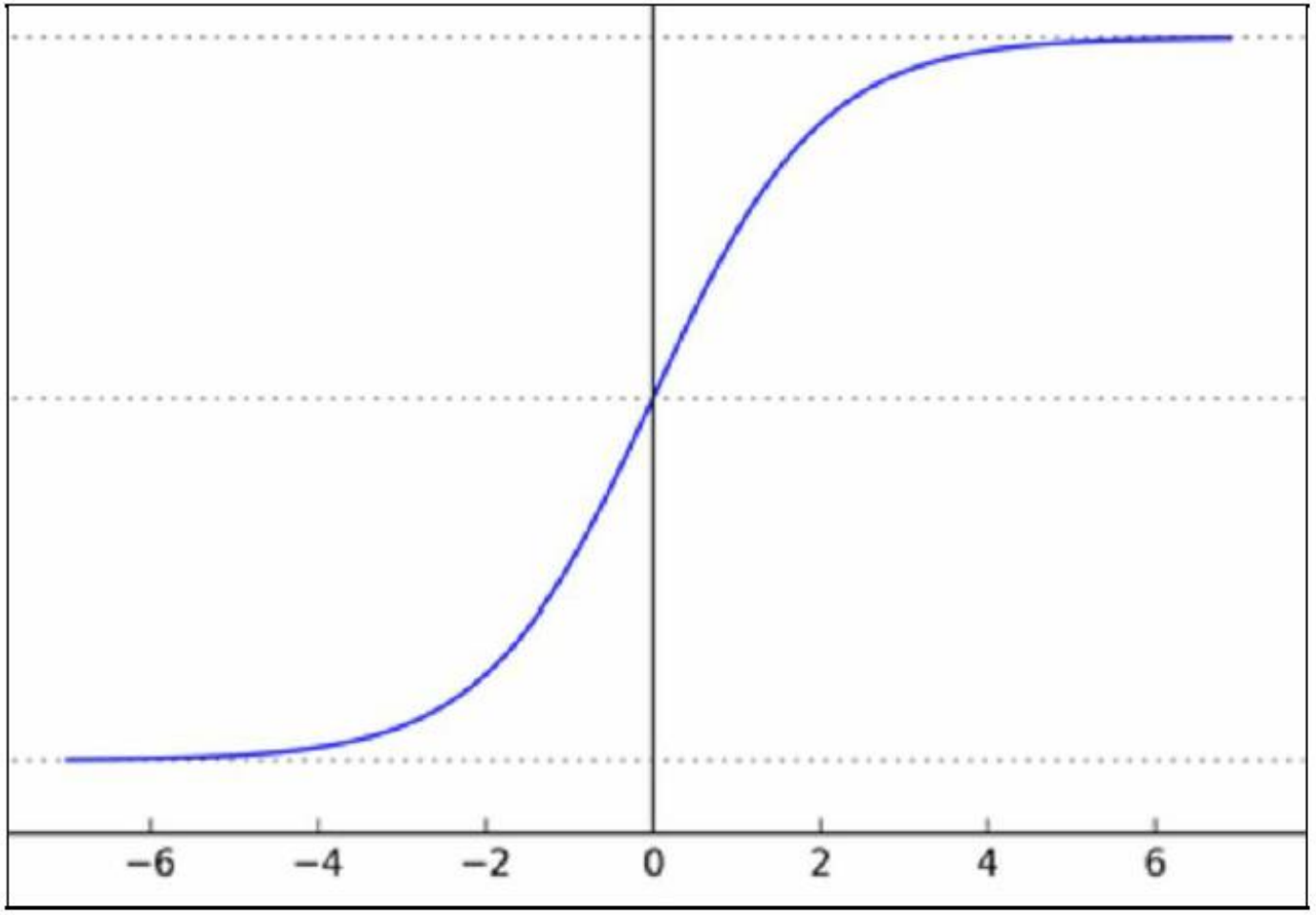
- >>> from keras.models import Sequential
- >>> from keras.layers import Dense
- >>> model = Sequential()
- >>> model.add(Dense(12,input_dim=8,kernel_initializer='random_uniform')

# Training the MLP

- Let's consider a single neuron; what are the best choices for the weight *w* and the bias *b*?
- We would like to provide a set of training examples and let the computer adjust the weight and the bias in such a way that the errors produced in the output are minimized.
- Suppose we have a set of images of cats and another separate set of images not containing cats.
- We would like our neuron to adjust its weights and bias so that we have fewer and fewer images wrongly recognized as non-cats. This requires that a small change in weights (and/or bias) causes only a small change in outputs.
- After all, kids learn little by little.

# Examples and Perceptron Rule

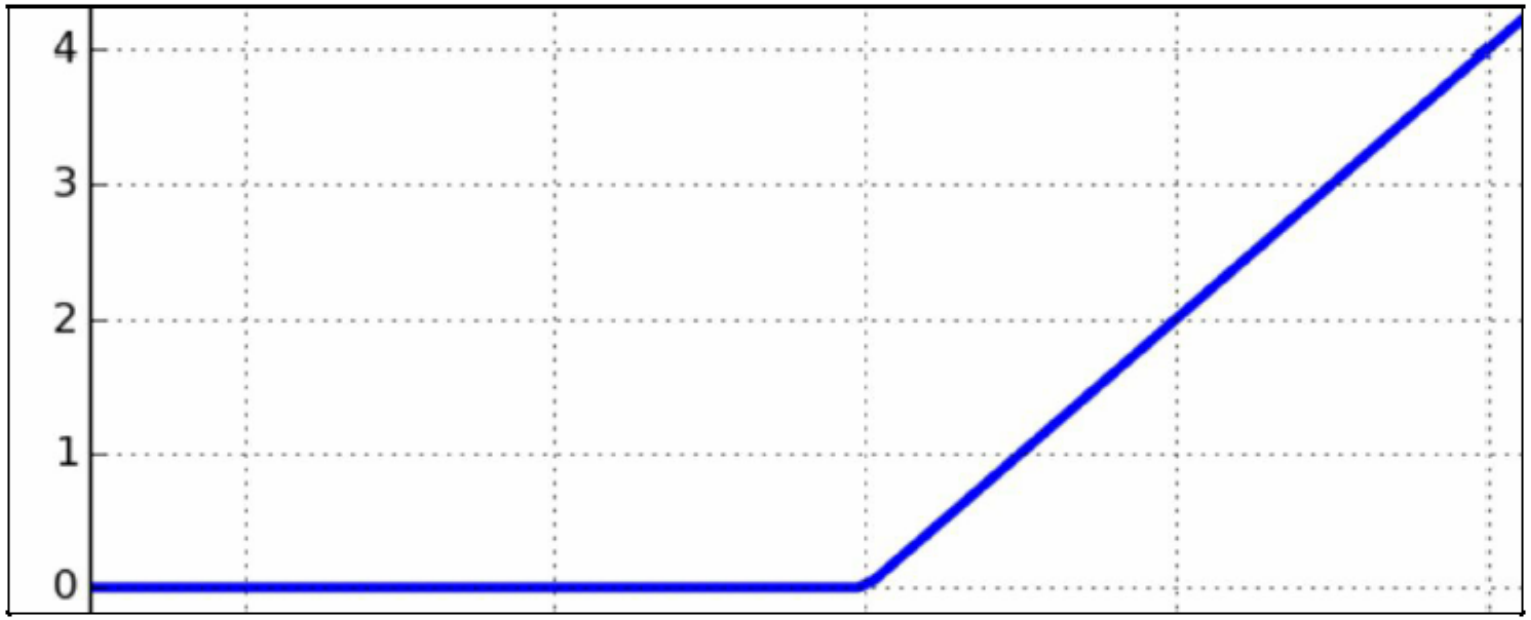
# Sigmoid



# Sigmoid

- A neuron with sigmoid activation has a behavior similar to the perceptron
- But the changes are gradual and output values, such as *0.5539* or *0.123191*, are perfectly legitimate.
- In this sense, a sigmoid neuron can answer *maybe*.

# Relu

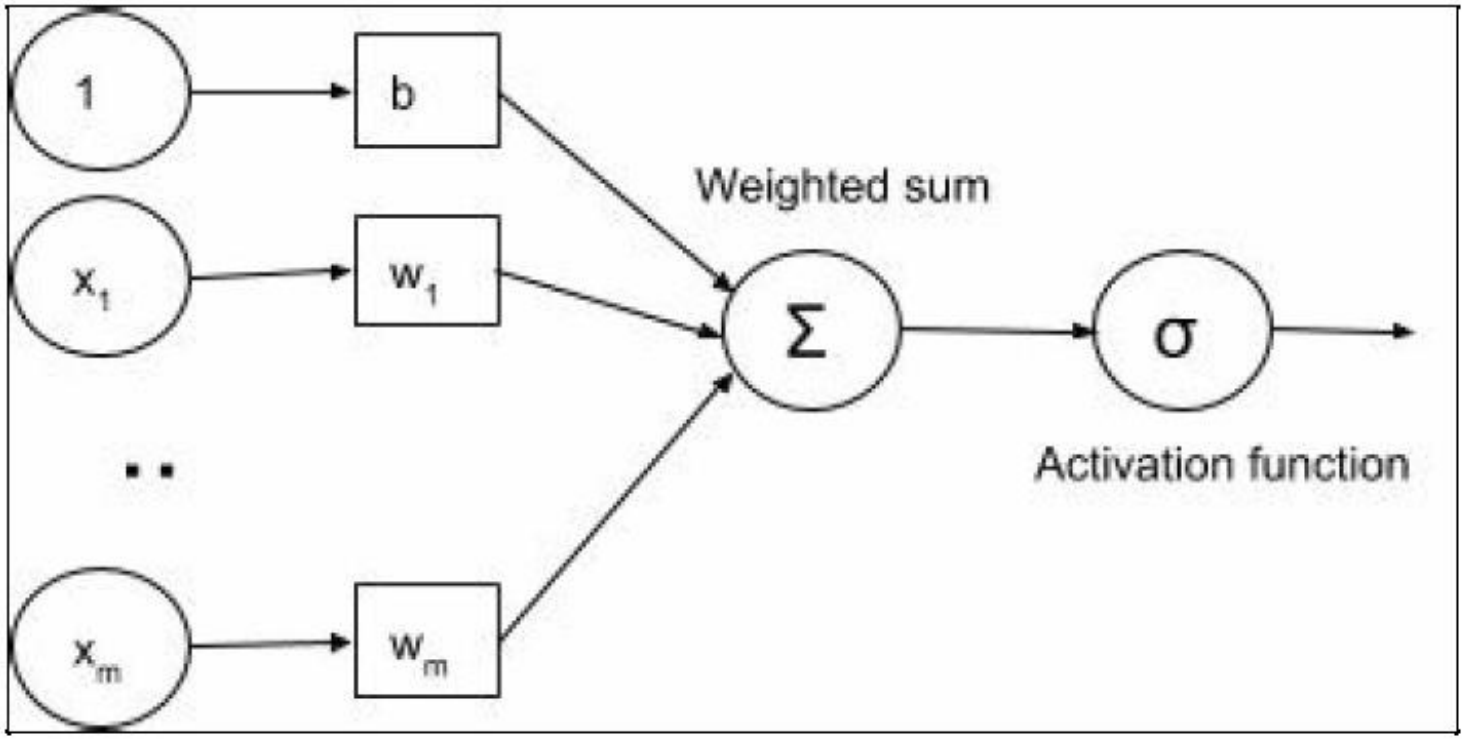


# Relu

- Recently, a very simple function called **rectified linear unit** (**ReLU**) became very popular because it generates very good experimental results.

- A ReLU is simply defined as f(x)=max(0,x)

# Activation function at Work



# Handwritten Digit Recognition

- In this section, we will build a network that can recognize handwritten numbers.
- For achieving this goal, we use MNIST (for more information, refer to http://yann.lecun.com/exdb/mnist/)
- It is a database of handwritten digits made up of a training set of 60,000 examples and a test set of 10,000 examples.
- The training examples are annotated by humans with the correct answer. For instance, if the handwritten digit is the number three, then three is simply the label associated with that example.

# Supervised Learning

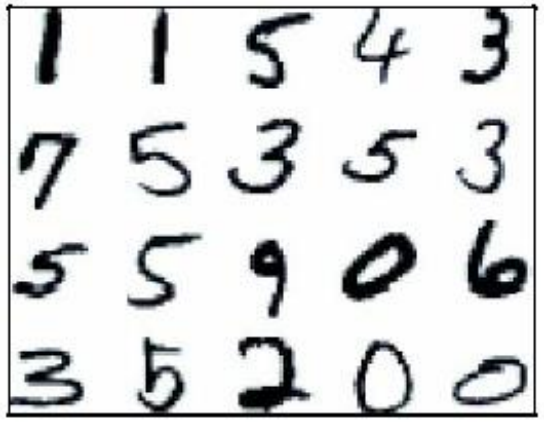
- In machine learning, when a dataset with correct answers is available, we say that we can perform a form of *supervised learning*. In this case, we can use training examples for tuning up our net.

- Testing examples also have the correct answer associated with each digit. In this case, however, the idea is to pretend that the label is unknown, let the network do the prediction, and then later on, reconsider the label to evaluate how well our neural network has learned to recognize digits.

- So, not unsurprisingly, testing examples are just used to test our net.

# MNIST Dataset

- Each MNIST image is in gray scale, and it consists of 28 x 28 pixels. A subset of these numbers is represented in the following diagram:



# One-hot Encoding - OHE

- In many applications, it is convenient to transform categorical (non-numerical) features into numerical variables.

- For instance, the categorical feature digit with the value *d* in *[0-9]* can be encoded into a binary vector with *10* positions, which always has *0* value, except the *d*-th position where a *1* is present.

- This type of representation is called **one-hot encoding** (**OHE**) and is very common in data mining when the learning algorithm is specialized for dealing with numerical functions.

# Defining a simple neural net in Keras

- Here, we use Keras to define a network that recognizes MNIST handwritten digits.
- We start with a very simple neural network and then progressively improve it.
- https://github.com/INCIBMB/ML-Day1

# Dataset

- Keras provides suitable libraries to load the dataset and split it into training sets X_train, used for fine-tuning our net, and tests set *X_test,* used for assessing the performance.

- Data is converted into float32 for supporting GPU computation and normalized to *[0, 1]*.

- In addition, we load the true labels into Y_train and Y_test respectively and perform a one-hot encoding on them.

# Code

```
# 10 outputs
# final stage is softmax
model = Sequential()
model.add(Dense(NB_CLASSES,
input_shape=(RESHAPED,)))
model.add(Activation('softmax'))
model.summary()
```

# Compile

Once we define the model, we have to compile it so that it can be executed by the Keras backend (either Theano or TensorFlow). There are a few choices to be made during compilation:

- We need to select the *optimizer* that is the specific algorithm used to update weights while we train our model

- We need to select the *objective function* that is used by the optimizer to navigate the space of weights (frequently, objective functions are called *loss function*, and the process of optimization is defined as a process of loss *minimization*)

- We need to evaluate the trained model

# Loss Function

- **MSE**: This is the mean squared error between the predictions and the true values. Mathematically, if P is a vector of *n* predictions, and *Y* is the vector of *n* observed values, then they satisfy the following equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(p_i - Y_i)^2$$

- *This objective functions average all the mistakes made for each prediction, and if the prediction is far from the true value, then this distance is made more evident by the squaring operation.*

# Loss Function 2

- **Categorical cross-entropy**: This is the multiclass logarithmic loss. If the target is *ti,j* and the prediction is *pi,j*, then the categorical cross-entropy is this:

$$L_i = \sum_{1}^{C} t_{i,j} \log(p_{i,j})$$

- *This objective function is suitable for multiclass labels predictions. It is also the default choice in association with softmax activation*

# Metrics

Some common choices for metrics (a complete list of Keras metrics is at https://keras.io/metrics/) are as follows:

- **Accuracy**: This is the proportion of correct predictions with respect to the targets

# Metrics

$$Precision = \frac{tp}{tp + fp} = Sensitivity$$

$$Recall = \frac{tp}{tp + fn}$$

$$Specificity = \frac{tn}{tn + fp}$$

| | | True condition | |
|---|---|---|---|
| | Total population | Condition positive | Condition negative |
| **Predicted condition** | Predicted condition positive | **True positive** | **False positive**, Type I error |
| | Predicted condition negative | **False negative**, Type II error | **True negative** |

# F1 Score

- The traditional F-measure or balanced F-score (F1 score) is the harmonic mean of precision and sensitivity:

$$F_1 = 2\frac{sens * spec}{sens + spec}$$

- Metrics are similar to objective functions, with the only difference that they are not used for training a model but only for evaluating a model. Compiling a model in Keras is easy:

# Training the model

- Once the model is compiled, it can be then trained with the fit() function, which specifies a few parameters:

- *epochs*: This is the number of times the model is exposed to the training set. At each iteration, the optimizer tries to adjust the weights so that the objective function is minimized.

- *batch_size*: This is the number of training instances observed before the optimizer performs a weight update.

# Training - 2

Training a model in Keras is very simple. Suppose we want to iterate for NB_EPOCH steps:

- history = model.fit(X_train, Y_train, batch_size=BATCH_SIZE, epochs=NB_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)

# Testing

- Once the model is trained, we can evaluate it on the test set that contains new unseen examples. In this way, we can get the minimal value reached by the objective function and best value reached by the evaluation metric.

- Note that the training set and the test set are, of course, rigorously separated. There is no point in evaluating a model on an example that has already been used for training.

- Learning is essentially a process intended to generalize unseen observations and not to memorize what is already known:

# Testing Code

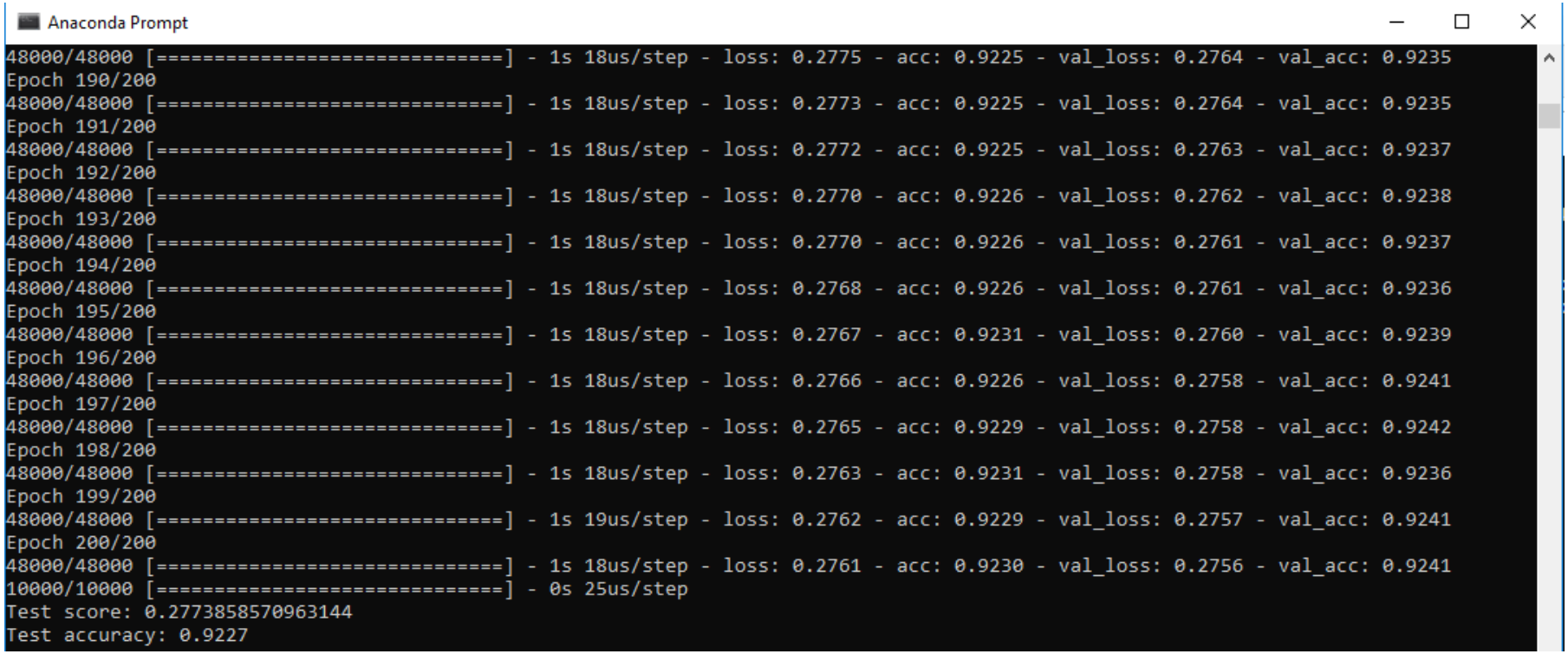
- score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
- print("Test score:", score[0])
- print('Test accuracy:', score[1])

# Congrats

So, congratulations, you have just defined your first neural network in Keras. A few lines of code, and your computer is able to recognize handwritten numbers. Let's run the code and see what the performance is.

# Results

The network is trained on 48,000 samples, and 12,000 are reserved for validation. Once the neural model is built, it is then tested on 10,000 samples. We can notice that the program runs for 200 iterations, and each time, the accuracy improves. When the training ends, we test our model on the test set and achieve about 92.36% accuracy on training, 92.27% on validation, and 92.22% on the test.

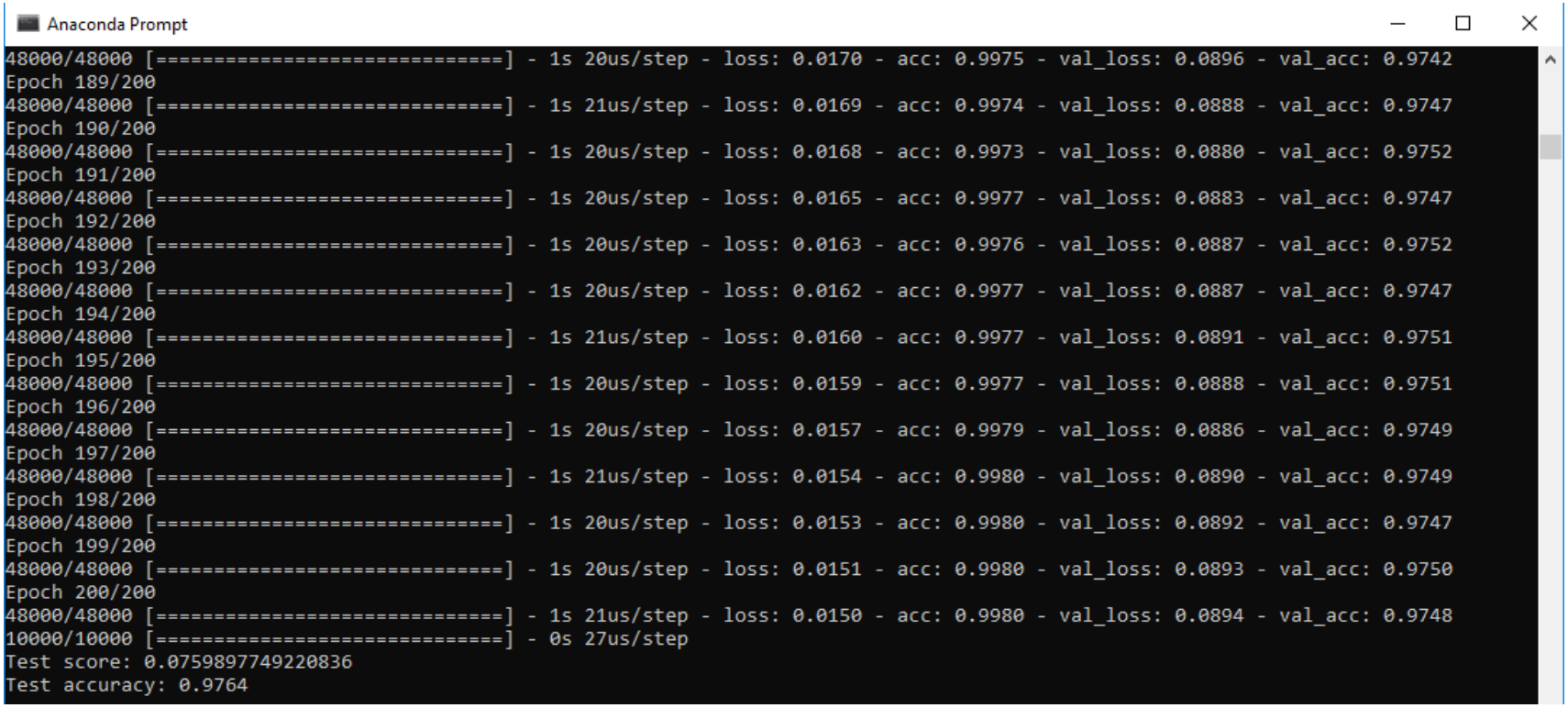


# Improving Simple-Net

- We have a baseline accuracy of 92.36% on training, 92.27% on validation, and 92.22% on the test. We can certainly improve it. Let's see how.
- A first improvement is to **add additional layers** to our network. So, after the input layer, we have a first dense layer with the N_HIDDEN neurons and an activation function relu.
- This additional layer is considered *hidden* because it is not directly connected to either the input or the output. After the first hidden layer, we have a second hidden layer, again with the N_HIDDEN neurons, followed by an output layer with 10 neurons, each of which will fire when the relative digit is recognized.
- The following code defines this new network:

# Results Stop at 20 epochs

Train: 99.8%, Validation: 97.48%, Testing: 97.64%

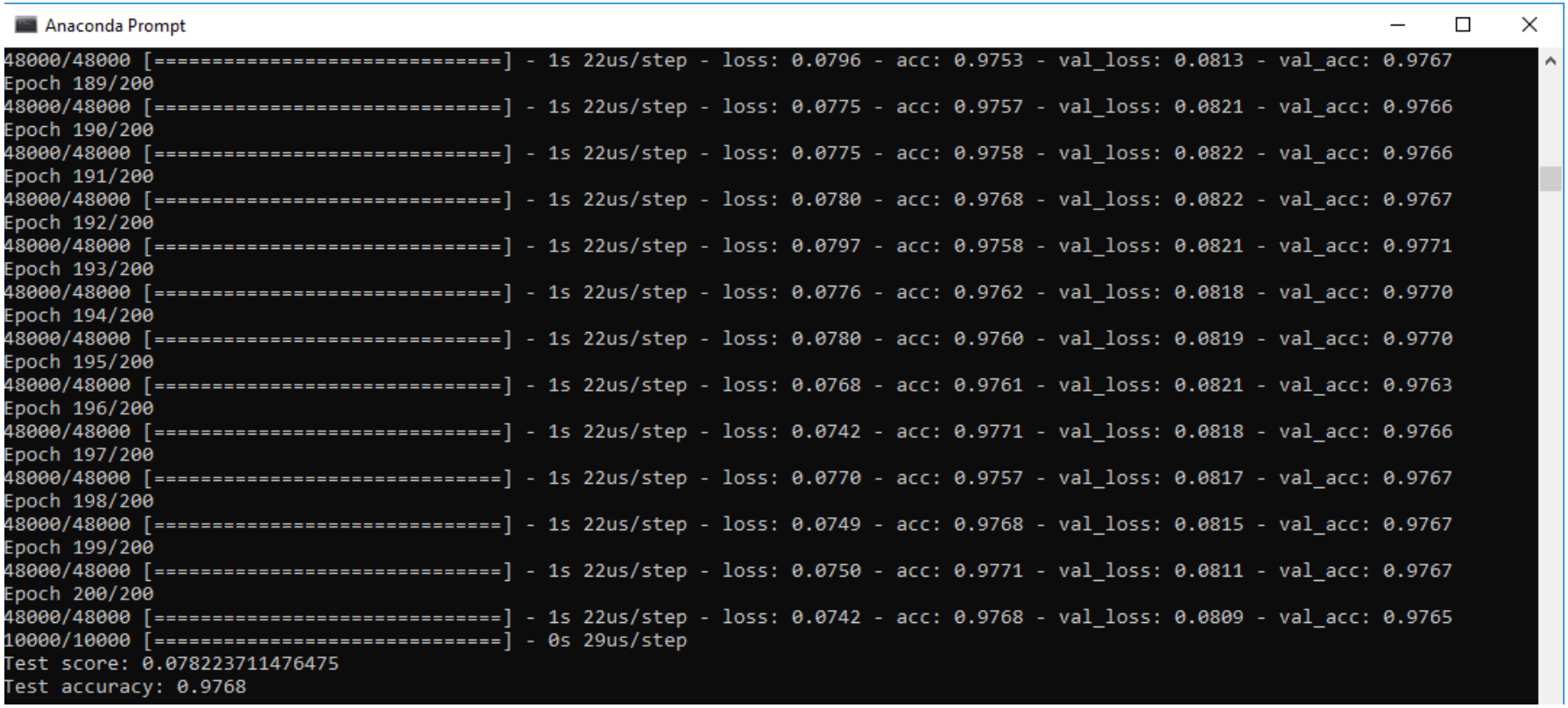


# Further improvement with Dropouts

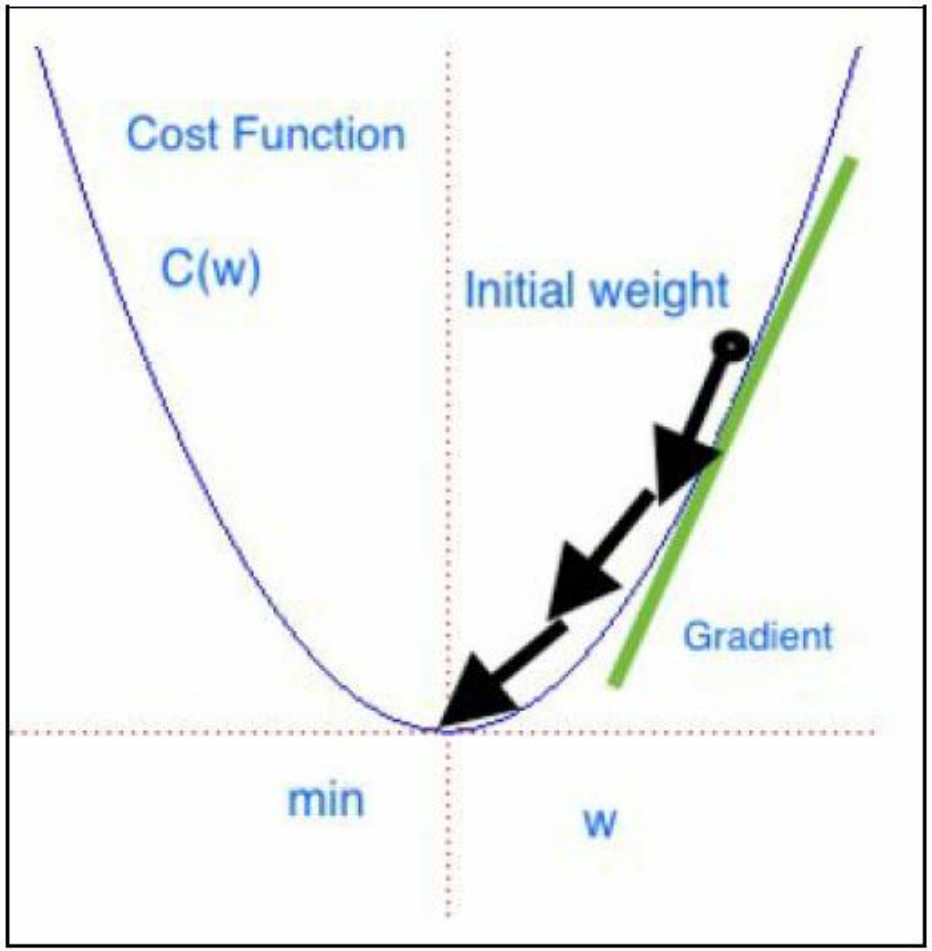
- Now our baseline is 94.50% on the training set, 94.63% on validation, and 94.41% on the test.
- A second improvement is very simple. We decide to randomly **drop with the dropout probability** some of the values propagated inside our internal dense network of hidden layers.
- In machine learning, this is a well-known form of regularization.
- Surprisingly enough, this idea of randomly dropping a few values can improve our performance:

# Results

Train: 97.68%, Validation: 97.65%, Testing: 97.68%



# Optimizers (Gradient Descent)




The gradient descent can be seen as a hiker who aims at climbing down a mountain into a valley.

The mountain represents the function *C*, while the valley represents the minimum *Cmin*.

The hiker has a starting point *w0*. The hiker moves little by little. At each step *r*, the gradient is the direction of maximum increase.

# Optimizer – Learning Rate

- At each step, the hiker can decide what the leg length is before the next step. This is the *learning rate η.*

- Note that if $\eta$ is too small, then the hiker will move slowly. However, if $\eta$ is too high, then the hiker will possibly miss the valley.

- Keras uses its backend (either TensorFlow or Theano) for computing the derivative on our behalf so we don't need to worry about implementing or computing it.

- We just choose the activation function, and Keras computes its derivative on our behalf.

# Other Optimizers - Test

- Two more advanced optimization techniques known as **RMSprop** and **Adam** can also be used.

- RMSprop and Adam include the concept of momentum (a velocity component) in addition to the acceleration component that SGD has.

- This allows faster convergence at the cost of more computation.

| Optimizer | Test Accuracy |
|---|---|
| RMSprop | 97.82% |
| Adam | 97.79% |

# Learning Rate - Test

- There is another attempt we can make, which is changing the learning parameter for our optimizer.

- As you can see in the following graph, the optimal value is somewhere close to *0.001*, which is the default learning rate for the optimizer. Good! Adam works well out of the box:

| LR | Test Accuracy |
|---|---|
| 0.1 | <20% |
| 0.01 | 96% |
| 0.001 | 97% |
| 0.0001 | 96% |

# Increasing the number of neurons

- We can make yet another attempt, that is, changing the number of internal hidden neurons.

- The training time increases more and more.
- However, the gains that we are getting by increasing the size of the network decrease more and more

| NHIDDEN | Test Acc | # Params | Exec Time |
|---|---|---|---|
| 32 | 95% | 26506 | 50-60 |
| 64 | 97% | 55050 | 56-60 |
| 128 | 97% | 118282 | 55-60 |
| 256 | | | |
| 512 | | | |
| 1024 | 96% | 1863690 | 70 |

# Increasing the size of batch computation

- Gradient descent tries to minimize the cost function on all the examples provided in the training sets

- Stochastic gradient descent is a much less expensive variant, which considers only BATCH_SIZE examples.

| BATCH_SIZE | Test Accuracy |
|---|---|
| 64 | |
| 128 | |
| 256 | |
| 512 | |

# Saving your model

- pip install h5py

- from keras.models import model_from_json

- model_json = model.to_json()
- with open("model.json", "w") as json_file:
- json_file.write(model_json)
- # serialize weights to HDF5
- model.save_weights("model.h5")
- print("Saved model to disk")

# Saving / Loading Model and Architecture

- model.save("model.h5")
- print("Saved model to disk")

- # load model
- model = load_model('model.h5')
- # summarize model.
- model.summary()

- pip install opencv-python

# Regularizers

- In order to solve the overfitting problem, we need a way to capture the complexity of a model
- The complexity of a model can be conveniently represented as the number of nonzero weights.
- In other words, if we have two models, *M1* and *M2*, achieving pretty much the same performance in terms of loss function, then we should choose the simplest model that has the minimum number of nonzero weights.

# 3 types of Regularizers

- **L1 regularization** (also known as **lasso**): The complexity of the model is expressed as the sum of the absolute values of the weights
- **L2 regularization** (also known as **ridge**): The complexity of the model is expressed as the sum of the squares of the weights
- **Elastic net regularization**: The complexity of the model is captured by a combination of the two preceding techniques

# Keras supports L1 and L2

- from keras import regularizers

- model.add(Dense(64, input_dim=64, kernel_regularizer=regularizers.l2(0.01)))