Neural Networks for Epidemic Modelling

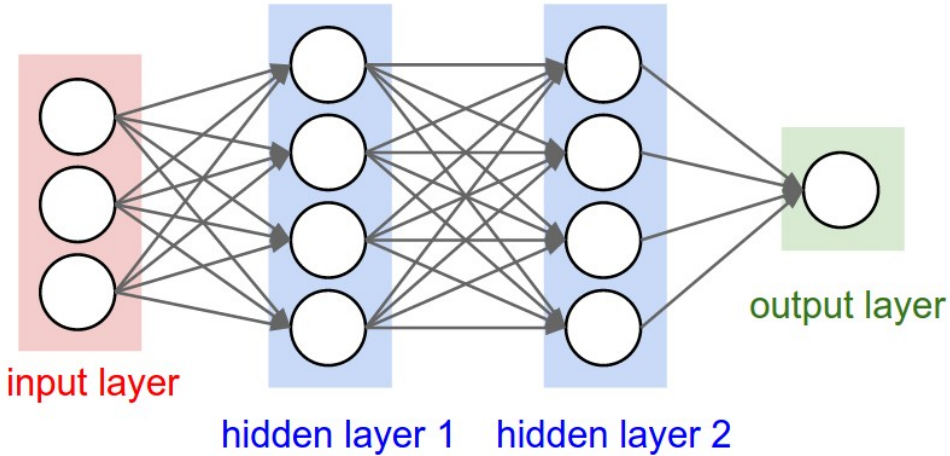
Alan Reyes-Figueroa
Modelación Epidemiológica, CIMAT

17.noviembre.2020

Table of Contents

Review of Session 1

Fully connected networks



A fully connected neural networks consist of a sequence of Dense layers.

Fully connected networks

Implementation in Keras

```
def FC_model(input_shape, output_shape):
   I = Input(input_shape, name='input'
   X = Dense(128, activation='relu', name='dense1')(I)
   X = Dense(128, activation='relu', name='dense2')(X)
   X = Dense(64, activation='relu', name='dense3')(X)
   X = Dense(32, activation='relu', name='dense4')(X)
   X = Dense(output_shape, activation=None, name='output')(X)

   model = Model(I, X, name='FC_model')
   return model
```

Geometry of fully connected networks

There is a geometric interpretation of fully connected, and in general, for all neural networks.

Geometry of fully connected networks

There is a geometric interpretation of fully connected, and in general, for all neural networks.

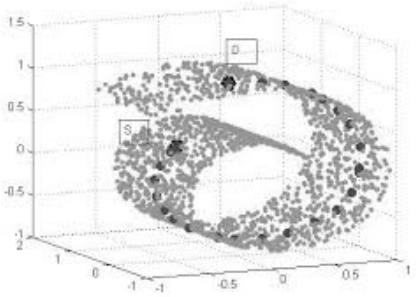
We work under the *manifold hypothesis*: We have data (X, Y), and we assume this data lives in an smooth (or almost smooth) low dimensional manifold $\mathcal{S}$, but S is represented immersed in a high-dimensional space $\mathcal{S} \subset \mathbb{R}^p$.

Geometry of fully connected networks

There is a geometric interpretation of fully connected, and in general, for all neural networks.

We work under the *manifold hypothesis*: We have data (X, Y), and we assume this data lives in an smooth (or almost smooth) low dimensional manifold $\mathcal{S}$, but S is represented immersed in a high-dimensional space $\mathcal{S} \subset \mathbb{R}^p$.

The neural networks will tray to estimate the shape and find this manifold $\mathcal{S}$. To do this, each dense layer folds and twists its subjacent space $\mathbb{R}^m$. This sequence of folds will approximate the shape of the manifold S.



A fully connected neural networks consist of a sequence of Dense layers.

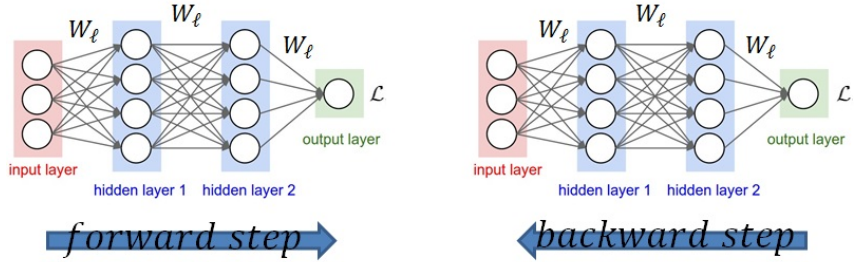
Loss function

- Regression: In a regression problem, common ways to measure the difference between the estimation $\widehat{y}_i$ and the desired ground-truth y_i are the MSE

$$\mathcal{L}(\mathbf{x}_i, y_i) = MSE = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \widehat{y}_i||_2^2,$$

and MAE errors

$$\mathcal{L}(\mathbf{x}_i, y_i) = MAE = \frac{1}{n}\sum_{i=1}^{n} ||y_i - \widehat{y}_i||_1.$$



Backpropagation

- Forward step: , given actual weights $w_{ij\ell}$ the data is passed through the network and loss function $\mathcal{L}$ is computed.

- Backward step: compute the derivatives $\nabla_{w_{ij\ell}}\mathcal{L}$ and $\nabla_{b_{j\ell}}\mathcal{L}$, and update weights $w_{ij\ell}$ and biases $b_{i\ell}$.

The re-calculation process is done by using variants of the *stochastic gradient descent* optimization algorithm, using mini-batchs

$$\begin{aligned} w_{ij\ell}^{(k+1)} &= w_{ij\ell}^{(k)} - \alpha\nabla_{w_{ij\ell}^{(k)}}\mathcal{L}(\mathbf{x}, y), \\ b_{i\ell}^{(k+1)} &= b_{i\ell}^{(k)} - \alpha\nabla_{b_{i\ell}^{(k)}}\mathcal{L}(\mathbf{x}, y), \end{aligned}$$

Here, $\alpha > 0$ is the step size of *learning rate*.

Hyperparameters

Parameters: (learnable by gradient descent)

- weights $w_{ij\ell}$ and biases $b_{i\ell}$, for $\ell = 1, \ldots, L$.

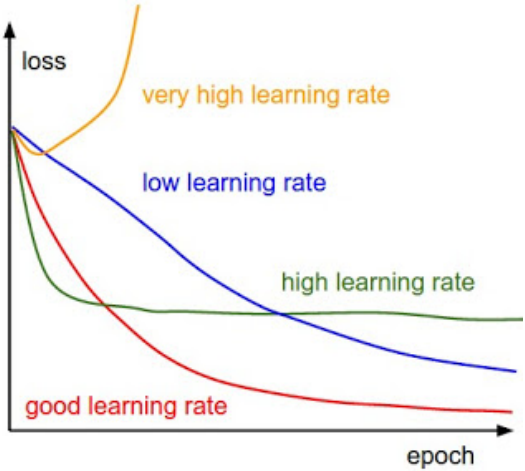
Hyper-parameters: (user-defined, non-learnable by gradient descent)

- Number of layers, and the type of each layer.
- Size of each layer (number of neurons in each layer).
- Activation function of each layer.
- Connections between the layers.

- Loss function (and other metrics).
- Optimization algorithms (GD, SGD, Nesterov, Adam, Adagrad, Adamax, RMSProp, ...), and parameters of those algorithms.
- learning rate or step-size α.
- Size ob the batch.
- Number of epochs.

- Training / Validation / Test partition size ...

Learning curves



Training/validation learning curves. The ideal case is the red curve.

Develop a neural network model

Prepare the data:

- Obtain data, select important variables, clean
- Inputation process
- Split in train / validation / test

Training:

- Propose / design an specific architecture
- Selection of hyper-parameters
- Training and validation
- Usually this process is done several times).

Testing:

- Test with new data
- Variations of the model
- (Prunning, re-ordering, ablation-study)

Dropout

Dropout

Dropout is a regularization technique, introduced in 2014 by Srivastava *et al.* in *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. It is a very practical form of regularization.

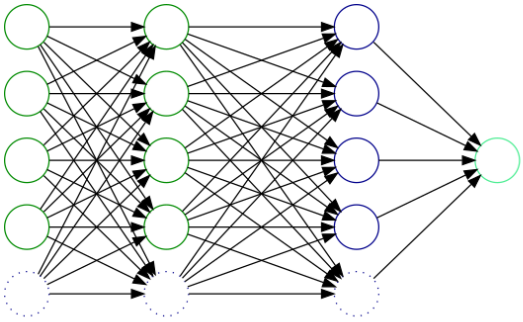
Dropout

Dropout is a regularization technique, introduced in 2014 by Srivastava *et al.* in *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. It is a very practical form of regularization.

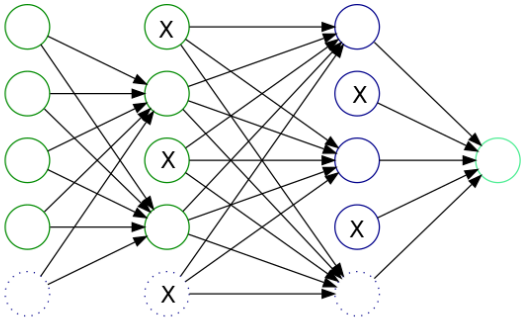
How it works?

- Each time a batch of data passes through the network, it chooses a random percentage of neurons and "kills" them.
  More specifically, for each Dense layer, one chooses a probability parameter $0 \leq p < 1$. Then the Dropout selects neurons with probability p, and set all weights associated with the selected neurons as $w_{ij} = 0$. This is equivalent to say the these neurons don't perform any operation to the data (the input signal just avoid them). Basically, it applies a Bernoulli percolator with probability p at each Dense layer.
- This method prevents to concentrate the weights in a reduced set of neurons, and distributes the weights in a better way across all components.
- This stochastic percolator is changed with every batch.

Dropout



Without dropout



With dropout

- Advantages: avoids fast overfitting, produce more robust models, stable under missing data.

Dropout

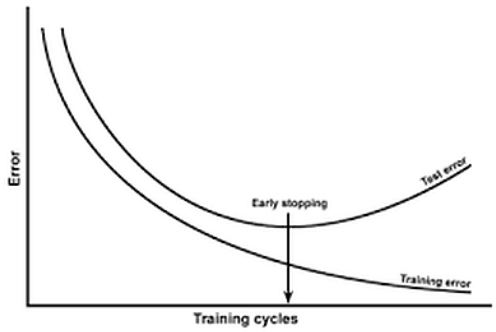
Implementation in Keras

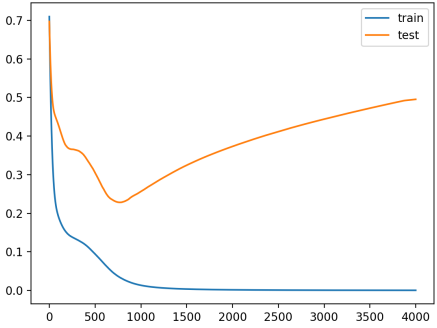
```
def Dropout_model(input_shape, output_shape):
    I = Input(input_shape, name='input'
    X = Dense(128, activation='relu', name='dense1')(I)
    X = Dropout(0.40, name='drop1')(X)
    X = Dense(128, activation='relu', name='dense2')(X)
    X = Dropout(0.25, name='drop2')(X)
    X = Dense(64, activation='relu', name='dense3')(X)
    X = Dropout(0.15, name='drop3')(X)
    X = Dense(32, activation='relu', name='dense4')(X)
    X = Dropout(0.15, name='drop4')(X)
    X = Dense(output_shape, activation=None, name='output')(X)

    model = Model(I, X, name='FC_Dropout_model')
    return model
```

Early Stopping

Overfitting

Recall when we talk about finding the optimal number of epochs to training the networks, before it starts overfitting.





Early stopping

Keras has an automatic mode to find this optimal number of epoch in the training process. This method is called *Early Stopping*.

In the Early Stopping, we will check a predefined evaluation metric. One checks this metric epoch by epoch, to see that if in the actual iteration, the metric is lower that the best record in all previous iterations.
If the actual metric is lower, we continue the training process. If the evaluation metric doesn't reduce during the last k number of epochs, then Keras automatically stops the training process. This number k is another user-defined hyper-parameter called the *patience*.

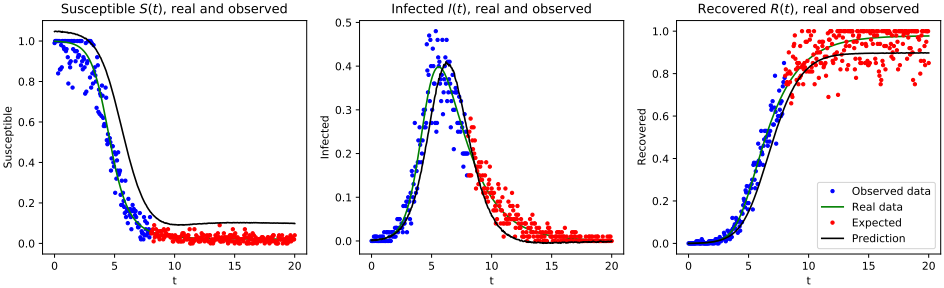
Keras also has an automatic form to save the best model found during the the training. Basically, it stores the model with lowest chosen evaluation metric.

- In practice, one usually set the evaluation metric as the validation loss.

Neural Networks for SIR

SIR model

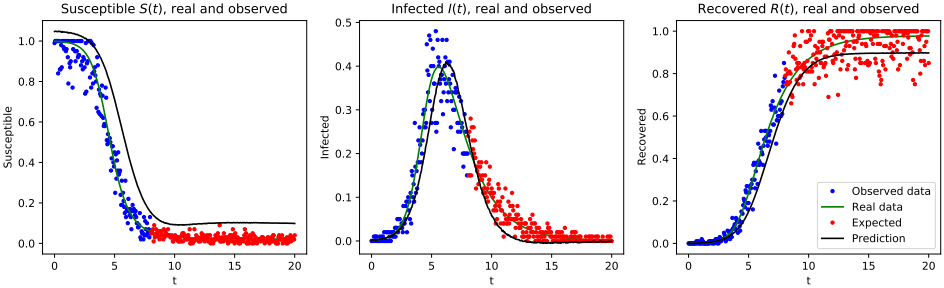
Recall the SIR model.



SIR model outputs: blue = training data, red = test data, black = prediction, greeen = real.

SIR model

Recall the SIR model.



SIR model outputs: blue = training data, red = test data, black = prediction, greeen = real.

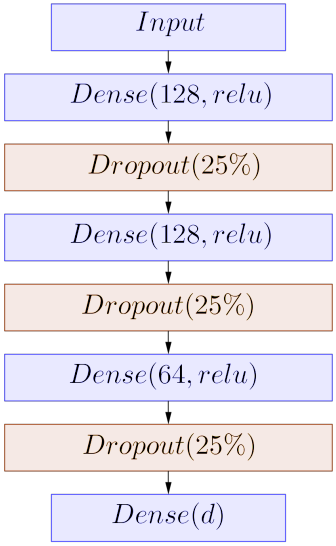
- Input data: usually we will have a sequence of $I(t_k)$ data for some finite interval $k = 0, 1, \ldots, T$.
- Output data: we usually want the estimation $I(t)$. Other times, we want the estimations for $S(t), I(t)$ and $R(t)$ (same or different interval). Sometimes we want parameter estimators for β and γ.

SIR model: Example 1

Example of a neural model for estimate $I(t)$ using only I data.

Suppose we have $I(t_k)$, $k = 0, 1, 2 \ldots, T$ (1 series of data is a vector in $\mathbb{R}^{T+1}$).

- Model input: series vectors of dimension $p = T + 1$.
- Model Output: series vector of dimension d, where d is the length of the desired prediction interval.

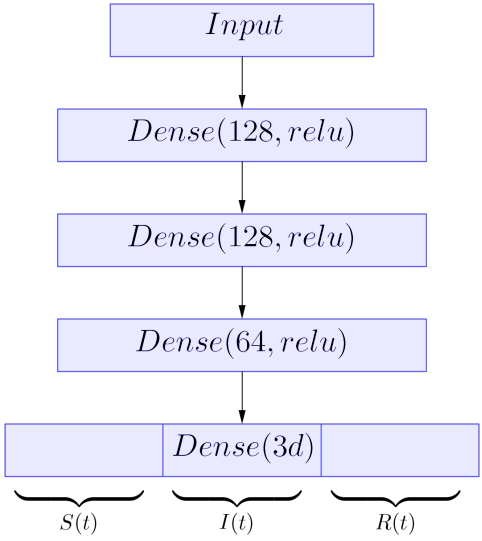


SIR model: Example 2

Example of a neural model for estimate $S(t), I(t)$ and $R(t)$ using only I data.

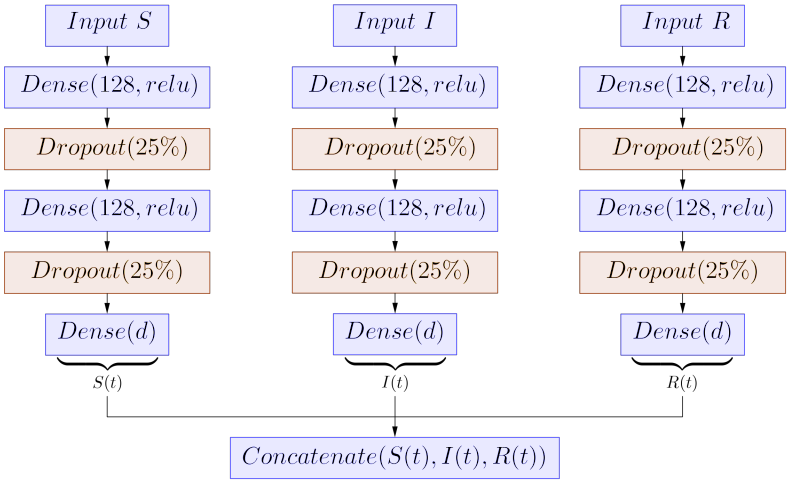
Suppose we have $I(t_k)$, $k = 0, 1, 2 \ldots, T$ (1 series of data is a vector in $\mathbb{R}^{T+1}$).

- Model input: series vectors of dimension $p = T + 1$.
- Model Output: series vector of dimension $3d$, where d is the length of the desired prediction interval.



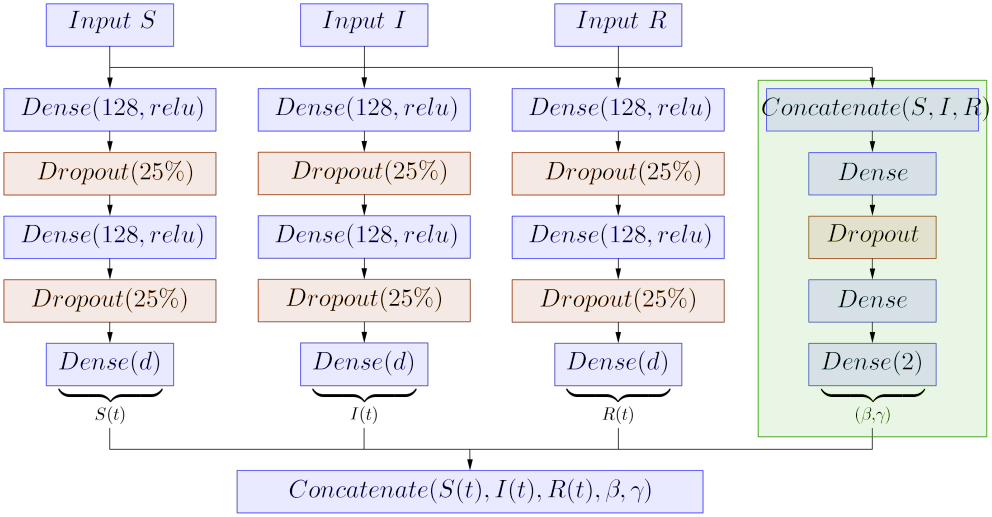
SIR model: Example 3

More complex architectures. A Parallel neural model for estimating $S(t), I(t)$ and $R(t)$ using all S, I, R data.



SIR model: Example 4

Another complex architecture. This time we also estimate the parameters β and γ.



Objectives for Practical Session 2

- Learn how to implement Dropout layers in Keras.
- Implement a (sequencial) fully-connected + dropout neural network, for estimate $I(t)$ in some interval of time, similar to Model 1.
- Learn how to implement custom loss functions in Keras.
- Implement a (sequencial) fully-connected + dropout neural network, for estimate all data $S(t), I(t), R(t)$ in some interval of time, similar to Model 2.

- Learn how to create non-sequencial Keras models. In particular, how to split the input in several branches and concatenate those branches.
- Implement a non-sequencial neural network, for estimate all data $S(t), I(t), R(t)$, similar to Model 3.
- Implement a non-sequencial neural network, for estimate all data $S(t), I(t), R(t)$ and the parameters β, γ , similar to Model 4.
- In all cases, we will se examples of how to generate and prepare the data.