ISMT S-136 Time Series Analysis with Python

Harvard Summer School

Dmitry Kurochkin

Summer 2021
Lecture 10

Contents

Contents

Unit Root Testing: Dickey-Fuller Test

Consider a univariate process of type

$$x_t = \phi x_{t-1} + w_t,$$

where w_t is Gaussian white noise.

How can one test whether the process is causal, i.e.

$$\text{H}_0\text{: } \phi = 1 \quad \text{versus} \quad \text{H}_1\text{: } |\phi| < 1?$$

Let's rewrite the difference equation as follows:

$$\underbrace{x_t - x_{t-1}}_{\nabla x_t} = \underbrace{(\phi - 1)}_{\rho} x_{t-1} + w_t$$

and test

$$\text{H}_0\text{: } \rho = 0 \quad \text{versus} \quad \text{H}_1\text{: } \rho \neq 0$$

by regressing ∇x_t on x_{t-1}
- namely *Dickey-Fuller* test.

Contents

Unit Root Testing: Augmented Dickey-Fuller Test

Consider a univariate process of type

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \ldots + \phi_p x_{t-p} + w_t,$$

where w_t is Gaussian white noise.

Let's rewrite the difference equation as follows:

$$\underbrace{x_t - x_{t-1}}_{\nabla x_t} = \underbrace{\left(\sum_{j=1}^{p} \phi_j - 1 \right)}_{\rho} x_{t-1} + \sum_{j=1}^{p-1} \underbrace{\left(- \sum_{i=j+1}^{p} \phi_i \right)}_{\psi_j} \nabla x_{t-j} + w_t$$

and test

$$\text{H}_0\text{: } \rho = 0 \quad \text{versus} \quad \text{H}_1\text{: } \rho \neq 0$$

by regressing ∇x_t on $x_{t-1}, \nabla x_{t-1}, \ldots, \nabla x_{t-p+1}$
- namely *augmented Dickey-Fuller* test.

Note: Unit root $z = 1$, i.e. $\phi(1) = 0$, means $\phi(1) = 1 - \sum_{j=1}^{p} \phi_p = 0$, i.e. $\rho = 0$.

Model Selection

Given observations $x_1, x_2, \ldots, x_n$, the criteria for model selection are:

1. Akaike information criterion (AIC):

$$\text{AIC} = -2 \ln \hat{L} + \frac{2k}{n},$$

where $\hat{L}$ is the estimated maximum value of the Likelihood and k is the number of parameters of the model (for example, ARMA(p, q) has $k = p + q$).

2. Bayesian information criterion (BIC):

$$\text{BIC} = -2 \ln \hat{L} + \frac{\ln k}{n},$$

where $\hat{L}$ and k are as above.

3. Use the Mean Squared Error (MSE) computed for newly predicted observations, that is:
for some $1 < m < n$, train on $x_1, \ldots, x_m$ and then test on $x_{m+1}, \ldots, x_n$ as follows:

$$\text{MSE} = \frac{1}{n-m} \sum_{i=m+1}^{n} (x_i - \hat{x}_i)^2,$$

where $\hat{x}_i$ denotes one-step ahead forecast of x_i.

Time Series Forecasting as Supervised Learning

Let's recall how we define a time series model:

Def.

Time series model is a specification of the joint distribution of $x_1, x_2, \ldots, x_n$.

For example, AR(m)-GARCH(p, q) medel defines the joint distribution as follows:

$$\phi(B)x_t = r_t$$

$$r_t = \sigma_t \varepsilon_t, \quad \text{where} \quad \varepsilon_t \overset{\text{iid}}{\sim} \mathcal{N}(0,1),$$

$$\sigma_t^2 = \alpha_0 + \sum_{j=1}^{p} \alpha_j r_{t-j}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2.$$

Given a time series model, one then can estimate the parameters (by maximizing Likelihood, for example) and forecast new observations of x_{n+1} as follows:

$$x_{n+1}^n = \mathrm{E}\big[x_{n+1} \,|\, x_n, x_{n-1}, \ldots, x_{n-m}\big]$$

Time Series Forecasting as Supervised Learning

Can we model the dependence

$$x_{n+1}^{n} = \mathrm{E}\big[x_{n+1} \,|\, x_n, x_{n-1}, \ldots, x_{n-m}\big],$$

which is a function of $x_n, x_{n-1}, \ldots, x_{n-m}$, directly without an explicit time series model?

Answer: Supervised Learning algorithms.

Time Series Forecasting as Supervised Learning

The goal is to model the dependence

$$\hat{x}_{t+1} = \mathrm{E}\big[x_{t+1} \,\big|\, x_t, x_{t-1}, \ldots, x_{t-m}\big]$$

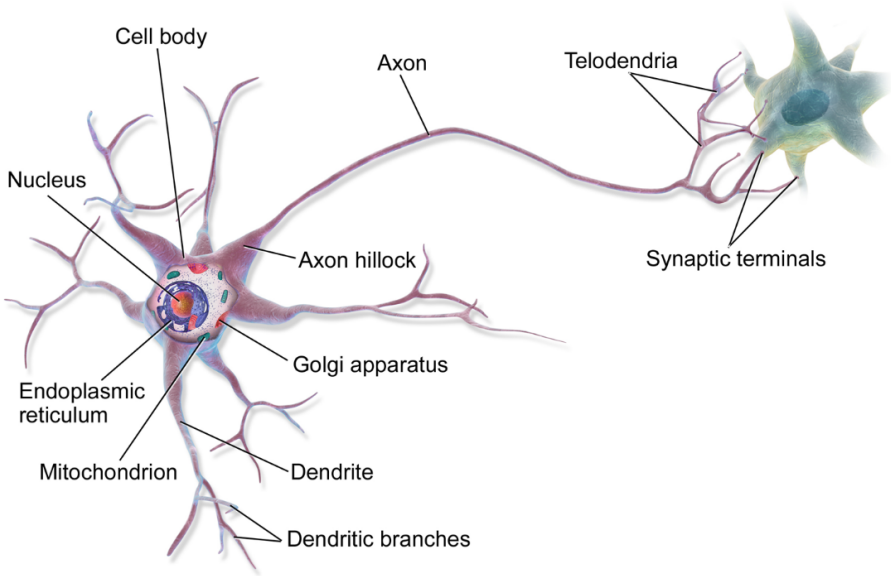
using data

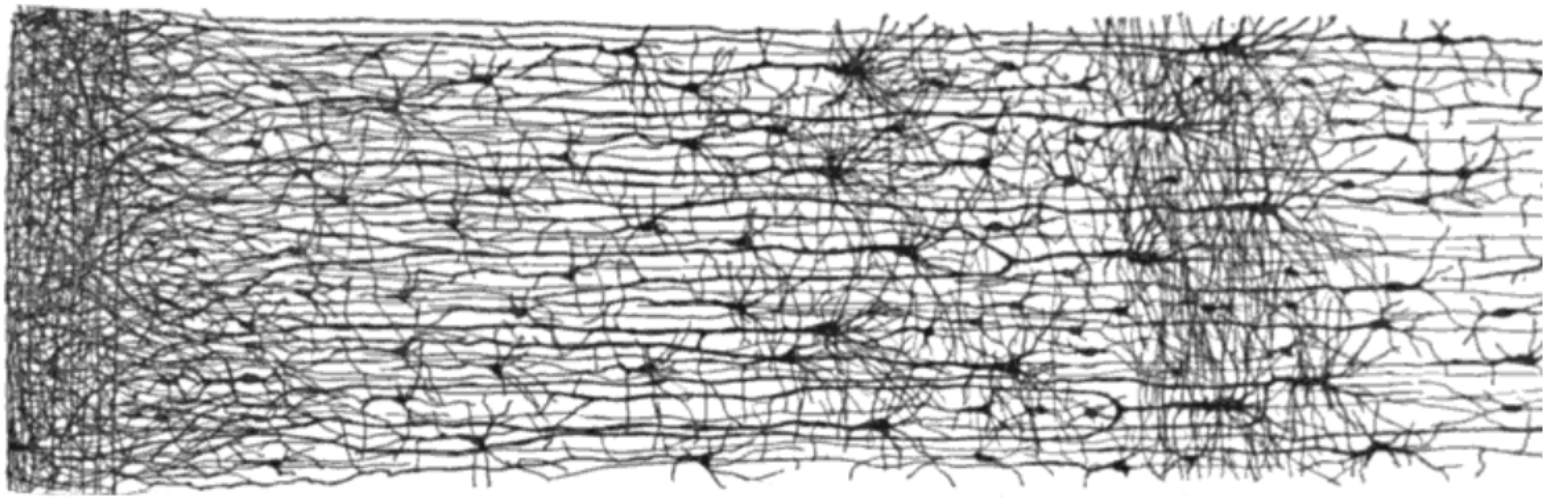
inputs $\boldsymbol{x}$						output y
x_1	x_2	x_3	x_4	$\ldots$	x_m	x_{m+1}
x_2	x_3	x_4	x_5		x_{m+1}	x_{m+2}
x_3	x_4	x_5	x_6		x_{m+2}	x_{m+3}
x_4	x_5	x_6	x_7		x_{m+3}	x_{m+4}
x_5	x_6	x_7	x_8		x_{m+4}	x_{m+5}
x_6	x_7	x_8	x_9		x_{m+5}	x_{m+6}
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$

Notice, however, that the assumptions of linear regression are likely violated.

Contents

Biological Neurons





Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Example: Artificial NN with 2 inputs and 1 output

Let's consider a *Deep Neural Network* which has two real-valued inputs (denoted by x_1 and x_2), one hidden layer that consists of two neurons (u_1 and u_2) with ReLU activation functions, and one output $\hat{y}$ with the ReLU activation function.

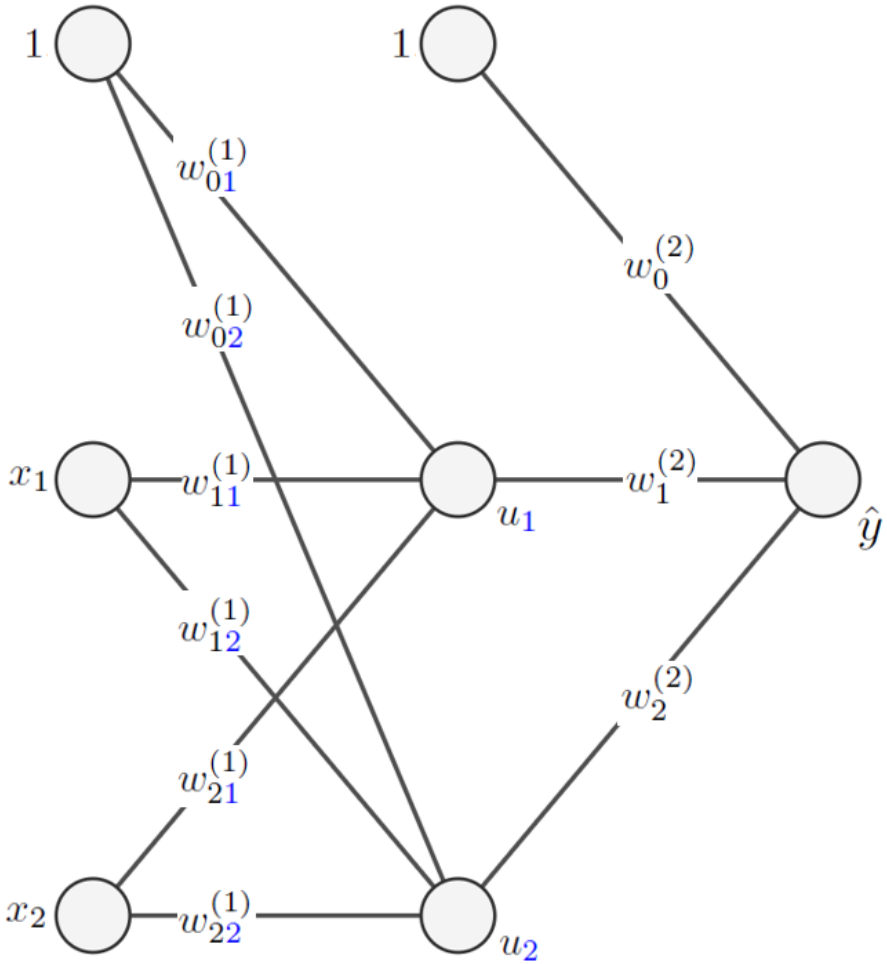
The explicit representation of this network is

input layer	hidden layer	output layer
x_1 x_2	$u_1 = f(w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2)$ $u_2 = f(w_{02}^{(1)} + w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2)$	$\hat{y} = f(w_0^{(2)} + w_1^{(2)} u_1 + w_2^{(2)} u_2)$

Here, $f(x)$ denotes the rectified linear unit (ReLU) defined as follows:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0. \end{cases}$$

Example: Artificial NN with 2 inputs and 1 output (cont.)



Contents

Artificial Neural Network

Neural Network (NN) with

n inputs,
M outputs and
1 hidden layer with H neurons is defined as:

$$\hat{\boldsymbol{y}} = f^{(2)}(f^{(1)}(\boldsymbol{x})),$$

where
$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ are inputs,
$\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M)^T \in \mathbb{R}^M$ are outputs,

$f^{(1)} : \mathbb{R}^n \mapsto \mathbb{R}^H$ and $f^{(2)} : \mathbb{R}^H \mapsto \mathbb{R}^M$,
where H is the number of *neurons* in the hidden layer.

Regression with NNs

NN

$$\hat{\boldsymbol{y}} = f^{(2)}(\underbrace{f^{(1)}(\boldsymbol{x})}_{\doteq \boldsymbol{u}})$$

with

- $f^{(2)} : \mathbb{R}^H \mapsto \mathbb{R}$, i.e. $M = 1$,

  is used for *regression*.

Regression with NNs

Keras:

```
import keras
from keras import models
from keras import layers

# number of inputs
n = 900

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(n,)))
model.add(layers.Dense(1, activation='relu'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 16)                14416
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 17
=================================================================
Total params: 14,433
Trainable params: 14,433
Non-trainable params: 0
_________________________________________________________________
```

Classification with NNs

NN

$$\hat{\boldsymbol{y}} = f^{(2)}(\underbrace{f^{(1)}(\boldsymbol{x})}_{\doteq \boldsymbol{u}})$$

with

- $f_m^{(2)}(\boldsymbol{u}) \geq 0$ for all $m \in \{1, 2, \ldots, M\}$ and $\boldsymbol{u} \in \mathbb{R}^H$ and $\sum_{m=1}^{M} f_m^{(2)}(\boldsymbol{u}) = 1$ for all $\boldsymbol{u} \in \mathbb{R}^H$

  is used for *classification*.

Classification with NNs

Keras:

```
import keras
from keras import models
from keras import layers

# number of inputs
n = 900

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(n,)))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 16)                14416
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 34
=================================================================
Total params: 14,450
Trainable params: 14,450
Non-trainable params: 0
_________________________________________________________________
```

Activation Functions

NN

$$\hat{\boldsymbol{y}} = f^{(2)}(\underbrace{f^{(1)}(\boldsymbol{x})}_{\doteq \boldsymbol{u}}).$$

What $f^{(1)}$ and $f^{(2)}$ should use?

Activation Functions

NN

$$\hat{\boldsymbol{y}} = f^{(2)}(\underbrace{f^{(1)}(\boldsymbol{x})}_{\doteq \boldsymbol{u}}).$$

What $f^{(1)}$ and $f^{(2)}$ should use?

Let

- $u_h \doteq f_h^{(1)}(\boldsymbol{x}) = \sigma_h^{(1)}\left(\sum_{j=0}^{n} w_{jh}^{(1)} x_j\right)$, where we define $x_0 \doteq 1$.
- $\hat{y}_m \doteq f_m^{(2)}(\boldsymbol{u}) = \sigma_m^{(2)}\left(\sum_{h=0}^{H} w_{hm}^{(2)} u_h\right)$, where we define $u_0 \doteq 1$.

Activation Functions

The NN

$$\hat{\boldsymbol{y}} = f^{(2)}(\underbrace{f^{(1)}(\boldsymbol{x})}_{\doteq \boldsymbol{u}})$$

becomes

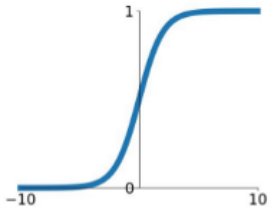
$$\hat{y}_m = \sigma_m^{(2)}\left(\sum_{h=0}^{H} w_{hm}^{(2)} \underbrace{\sigma_h^{(1)}\left(\sum_{j=0}^{n} w_{jh}^{(1)} x_j\right)}_{\doteq u_h}\right).$$

Activation Functions

What $\sigma_h^{(1)}$ and $\sigma_m^{(2)}$ should use?

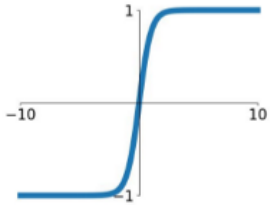
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



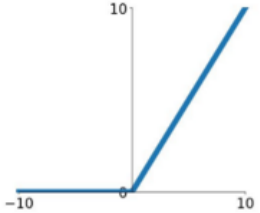
tanh

$$\tanh(x)$$



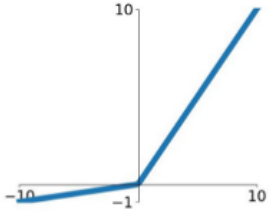
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

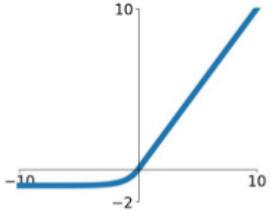


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Contents

Regularization Penalty

If we wish to discourage overfitting we can add a *regularization penalty*, $R(\boldsymbol{w})$, to the loss function:

$$J_{\text{reg}}(\boldsymbol{w}) = J(\boldsymbol{w}) + \lambda R(\boldsymbol{w}).$$

Regularization Penalty

If we wish to discourage overfitting we can add a *regularization penalty*, $R(\boldsymbol{w})$, to the loss function:

$$J_{\text{reg}}(\boldsymbol{w}) = J(\boldsymbol{w}) + \lambda R(\boldsymbol{w}).$$

The most common regularization terms are:

- L1-norm (similar to Lasso regression):

$$R(\boldsymbol{w}) \doteq \sum_{\ell} \sum_{i,j} |w_{ij}^{(\ell)}|$$

- L2-norm (similar to Ridge regression):

$$R(\boldsymbol{w}) \doteq \sum_{\ell} \sum_{i,j} \left(w_{ij}^{(\ell)}\right)^2$$

Regularization Penalty

Keras:

L1 Regularization

```
from keras import regularizers
lam = 0.0001

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu'))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu'))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

Regularization Penalty

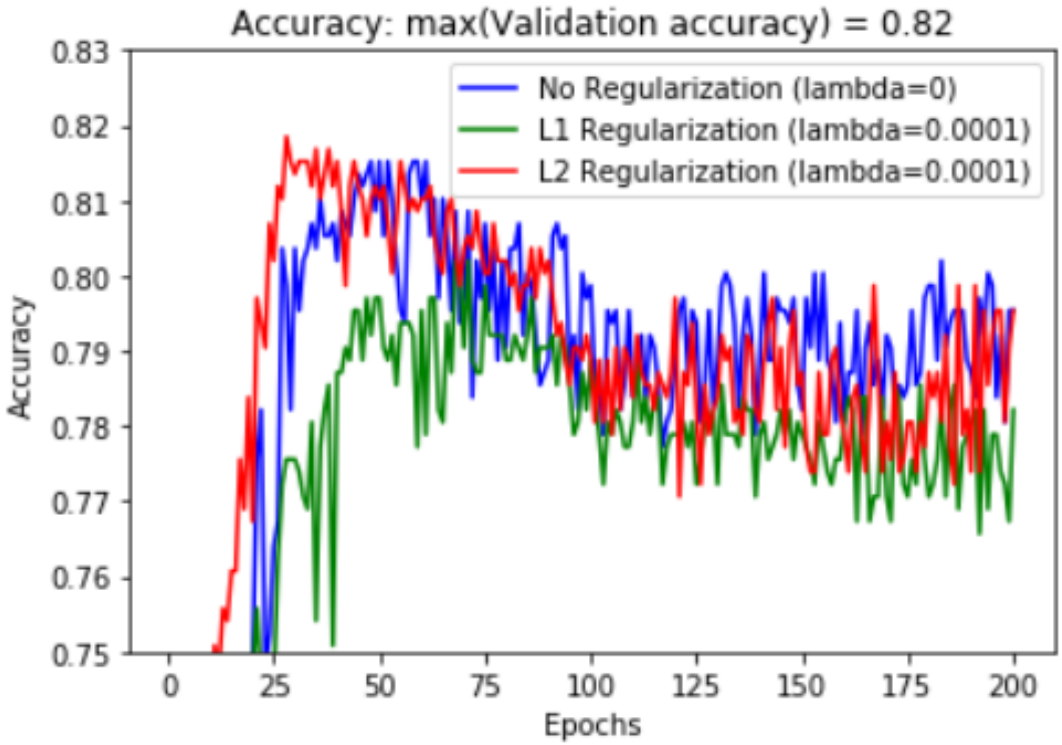
Keras:

L2 Regularization

```
from keras import regularizers
lam = 0.0001
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu'))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu'))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2,
                       activation='softmax'))

model.summary()
```

Regularization Penalty

Keras:

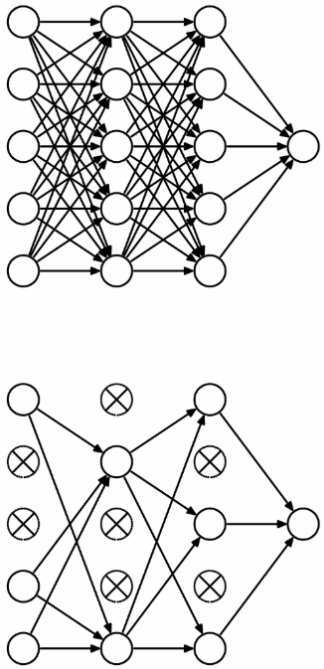


Contents

Dropouts

Dropout is another regularization tool which literally means "dropping out" a random number of neurons during every training step. After training, each neuron's input connection weight needs to be adjusted by a factor of (1-dropout rate).

Effectively, all we need to do is to set to zero a random number of outputs from a given layer:



Dropouts

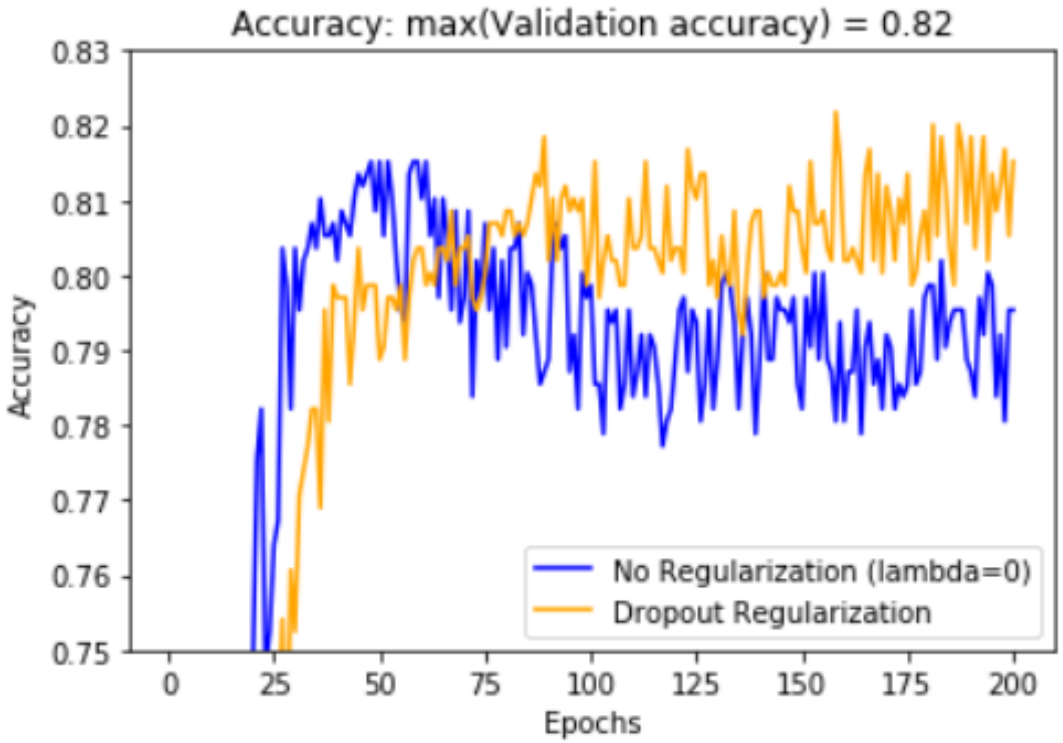
Keras:

Dropout Regularization

```
dropout_rate = 0.3

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dropout(dropout_rate))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(dropout_rate))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(dropout_rate))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

Dropouts

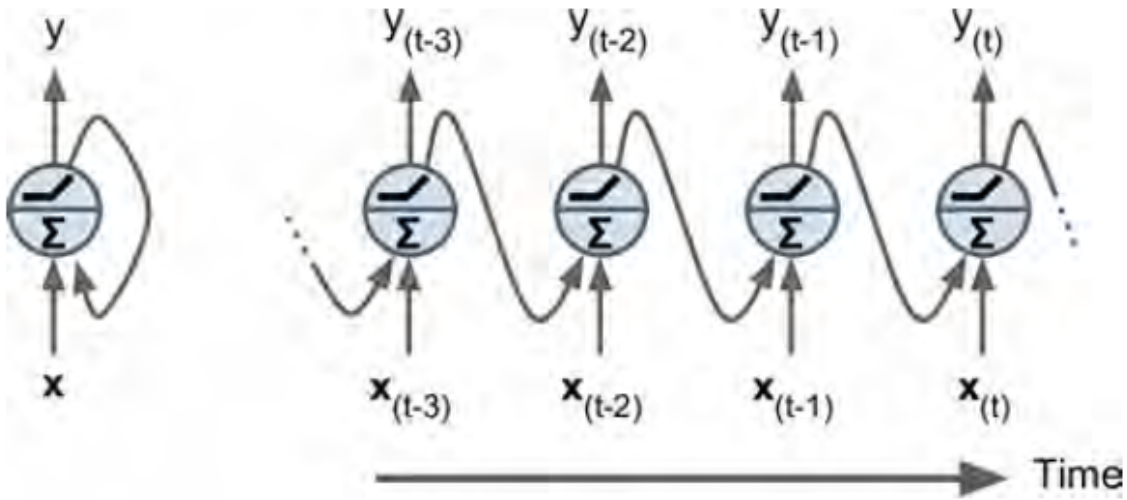
Keras:



Contents

RNN: Recurrent Neuron

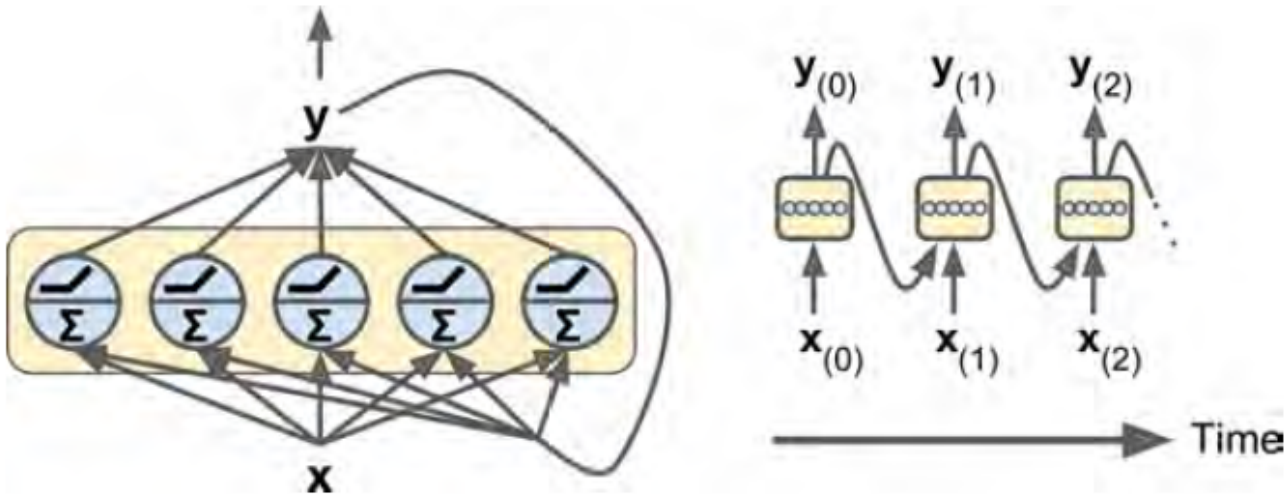
Recurrent Neuron:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

RNN: Layer of Recurrent Neurons

Layer of Recurrent Neurons:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Layer of Recurrent Neurons: Keras

Simple RNN in Keras:

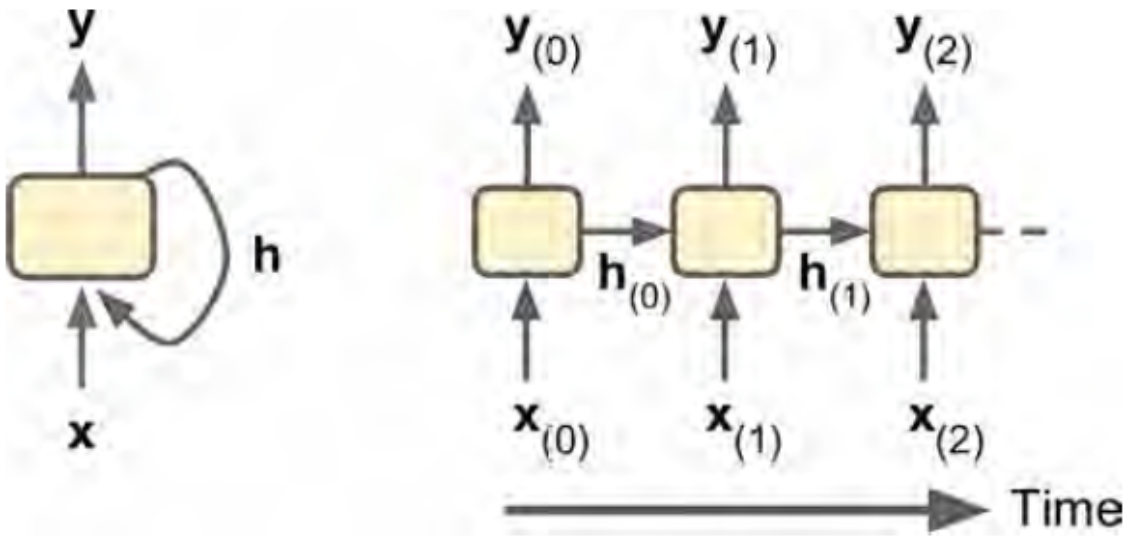
```
n_features = 2
n_timesteps = 200

model = models.Sequential()
model.add(layers.SimpleRNN(3, activation='relu', input_shape=(n_timesteps,n_features)))
model.add(layers.Dense(1, activation='linear'))

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_6 (SimpleRNN)     (None, 3)                 18
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 4
=================================================================
Total params: 22
Trainable params: 22
Non-trainable params: 0
```

Contents

RNN: Memory Cells

Layer of Recurrent Neurons:

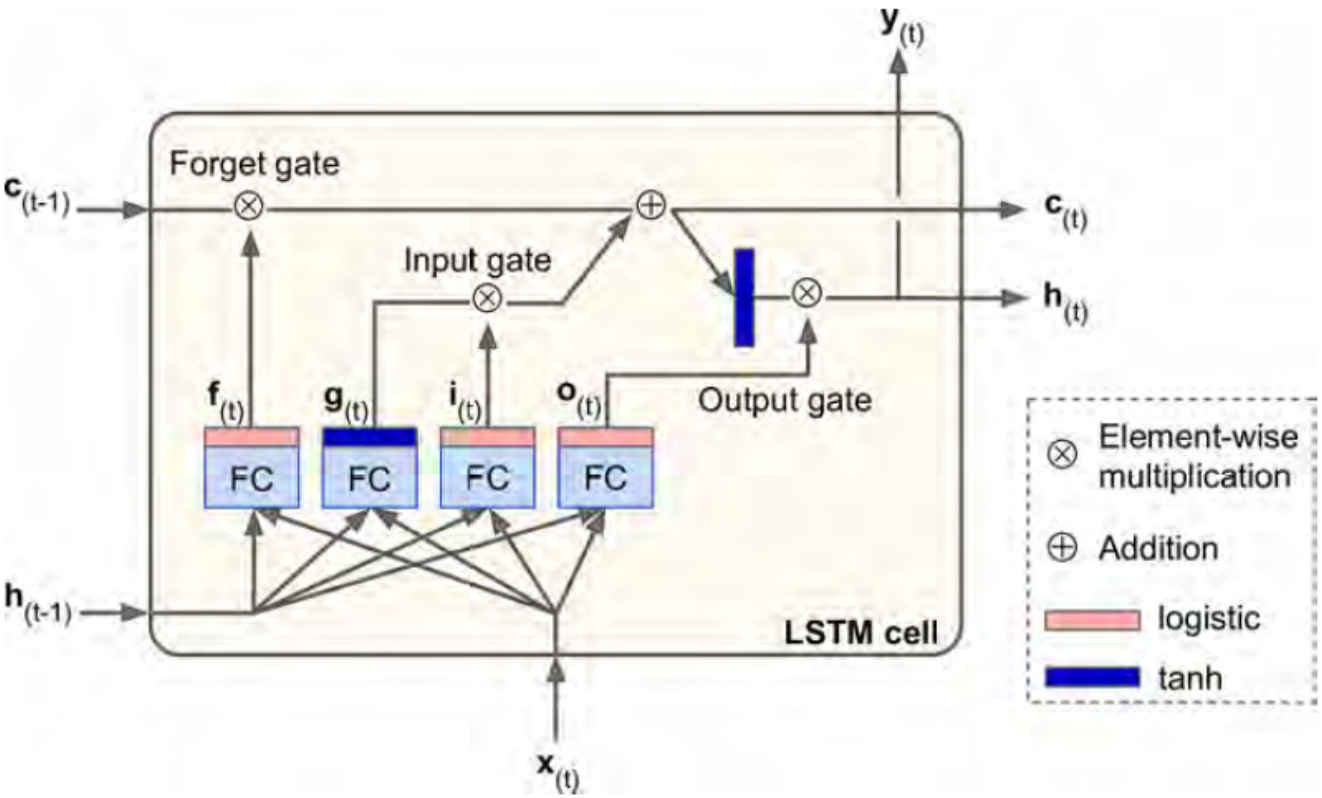


Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Long Short-Term Memory (LSTM) Cell

LSTM Cell:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Long Short-Term Memory (LSTM) Cell

LSTM Cell Model:

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_f$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Applications of LSTM Cells

- greatly improved speech recognition on over 4 billion Android phones (since mid 2015)
- greatly improved machine translation through Google Translate (since Nov 2016)
- greatly improved machine translation through Facebook (over 4 billion LSTMbased translations per day as of 2017)
- Siri and Quicktype on almost 2 billion iPhones (since 2016)
- generating answers by Amazon's Alexa and numerous other similar applications.

Long Short-Term Memory (LSTM) Cell: Keras

LSTM in Keras:

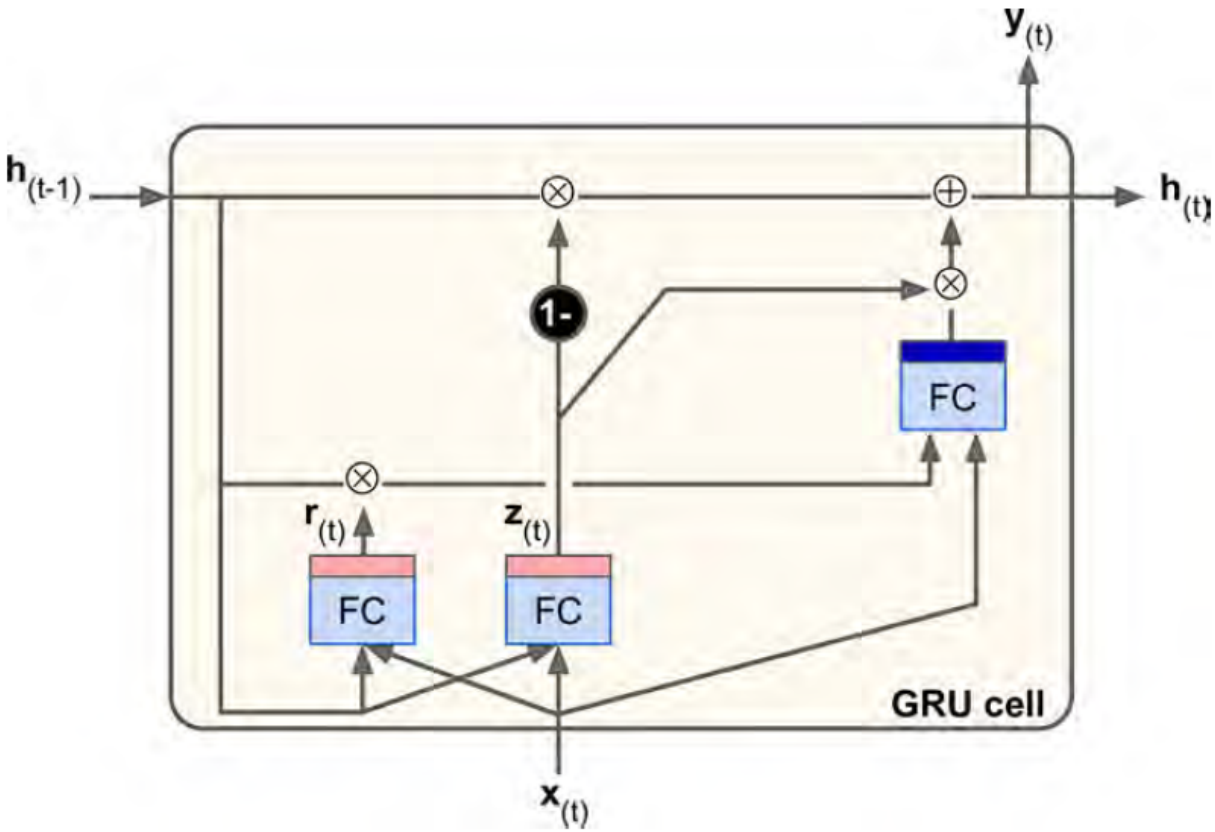
```
n_features = 2
n_timesteps = 200

model = models.Sequential()
model.add(LSTM(16, activation='relu', input_shape=(n_timesteps,n_features)))
model.add(layers.Dense(1, activation='linear'))

model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 16)                1216
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,233
Trainable params: 1,233
Non-trainable params: 0
```

Contents

Gated Recurrent Unit (GRU) Cell

GRU Cell:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Gated Recurrent Unit (GRU) Cell

GRU Cell Model:

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)\right) \\
\mathbf{h}_{(t)} &= \left(1 - \mathbf{z}_{(t)}\right) \otimes \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{t}\right)
\end{aligned}
$$

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron