

Głębokie uczenie

prof. UAM dr hab. Tomasz Górecki

tomasz.gorecki@amu.edu.pl

Uniwersytet im. Adama Mickiewicza w Poznaniu
Wydział Matematyki i Informatyki





Bengio, Y., Courville, A., Goodfellow, I. (2018). Deep Learning. Systemy uczące się. PWN.



Chollet, F. (2019). Deep Learning. Praca z językiem Python i biblioteką Keras. Helion.



Chollet, F., Allaire, J.J. (2019). Deep Learning. Praca z językiem R i biblioteką Keras. Helion.



Gibson, A., Patterson, J. (2018). Deep Learning. Praktyczne wprowadzenie. Helion.



Krzyśko, M., Wołyński, W., Górecki, T., Skorzybut, M. (2008). Systemy uczące się – rozpoznawanie wzorców, analiza skupień i redukcja wymiarowości. WNT.



Sejnowski, T.J. (2019). Deep learning Głęboka rewolucja. Poltext.



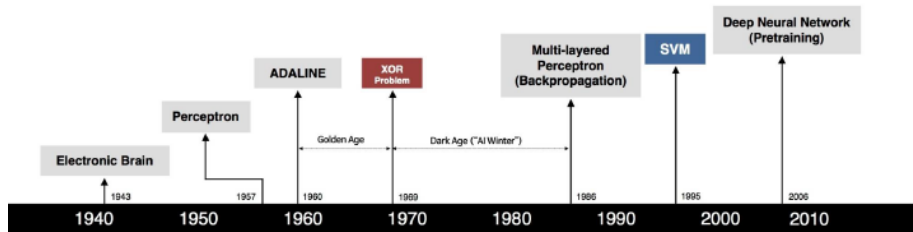
Trask, A. (2019). Zrozumieć głębokie uczenie. PWN.



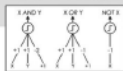
Zocca, V., Spacagna, G., Slater, D., Roelants, P. (2018). Deep Learning. Uczenie głębokie z językiem Python. Sztuczna inteligencja i sieci neuronowe. Helion.

- 1 <https://youtu.be/bfmFfD2RIcg> – Co to jest sieć neuronowa i jak działa [ENG]
- 2 https://youtu.be/gJ-1SD_tslk – Wprowadzenie do sztucznych sieci neuronowych [ENG]
- 3 https://youtu.be/_E7af0xwTkA – Sieci neuronowe [PL]
- 4 https://youtu.be/Wa_9S20SkKw – Sieci neuronowe nieco bardziej matematycznie [PL]
- 5 <https://youtu.be/sWP1LQgwxG8> – Sieci neuronowe w pełni matematycznie [PL]
- 6 <https://youtu.be/gSo1rv2k9Uc> – Głębokie sieci neuronowe w 5 minut. Playground Tensorflow [PL]
- 7 <https://youtu.be/1A56EDeiuM8> – Głębokie sieci neuronowe w 5 minut. Jak działa neuron. [PL]
- 8 <https://youtu.be/RykJGQLvFSM> – Głębokie sieci neuronowe – typy i szczegóły działania. [PL]

Początki sztucznych sieci neuronowych



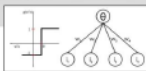
S. McCulloch - W. Pitts



- Adjustable Weights
- Weights are not Learned



F. Rosenblatt



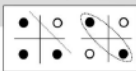
- Learnable Weights and Threshold



B. Widrow - M. Hoff



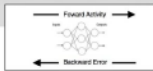
M. Minsky - S. Papert



- XOR Problem



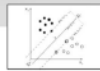
D. Rumelhart - G. Hinton - R. Williams



- Solution to nonlinearly separable problems
- Big computation, local optima and overfitting



V. Vapnik - C. Cortes



- Limitations of learning prior knowledge
- Kernel function: Human Intervention



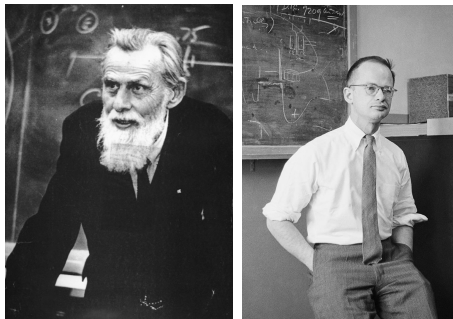
G. Hinton - S. Ruslan



- Hierarchical Feature Learning

Początki sztucznych sieci neuronowych

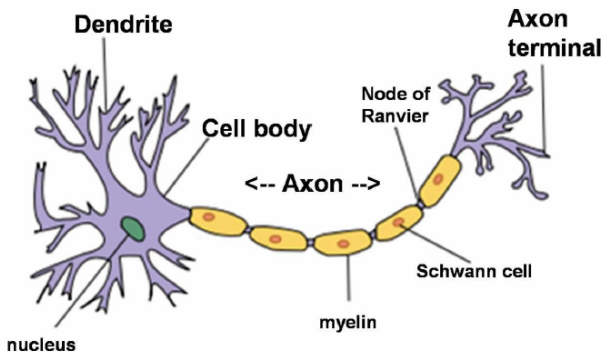
Za początek sztucznej sieci neuronowej określa się rok 1943. W tym roku dwóch naukowców, WARREN McCULLOCH – neurofizjolog amerykańskiego pochodzenia, oraz WALTER PITTS – logik, stworzyło model sztucznego neuronu w oparciu o biologiczny mechanizm działania ludzkiej komórki nerwowej. Od nazwisk autorów został on nazwany **neuronem McCULLOCHA-PITTSA** (model MCP).



Warren Sturgis McCulloch (1898-1969) & Walter Harry Pitts (1923-1969)

Początki sztucznych sieci neuronowych

Model ten zakłada, że sygnały wejściowe odbierane są przez dendryty, a następnie kierowane do jądra komórki nerwowej, gdzie są akumulowane. Gdy skumulowana wartość zgromadzonej informacji przekroczy pewien próg graniczny, wysyłany jest sygnał wzdłuż aksonu, odpowiedzialnego za transmisję sygnałów do zakończeń nerwowych – synaps. Model MCP nie zakładał jednak możliwości uczenia.

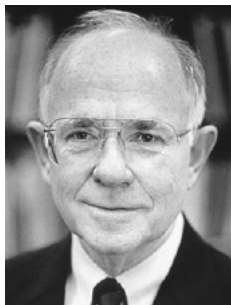


Kilka lat później, w 1949 r., kanadyjski psycholog DONALD HEBB przedstawił teorię uczenia neuronu. Jednak dopiero w 1957 roku FRANK ROSENBLATT – amerykański psycholog, w oparciu o model neuronu MCCULLOCHA-PITTSA przedstawił koncepcję **uczenia nadzorowanego**. Model **perceptronu ROSENBLATTA** był udoskonaloną wersją modelu MCP.



Donald Olding Hebb (1904-1985) & Frank Rosenblatt (1928-1971)

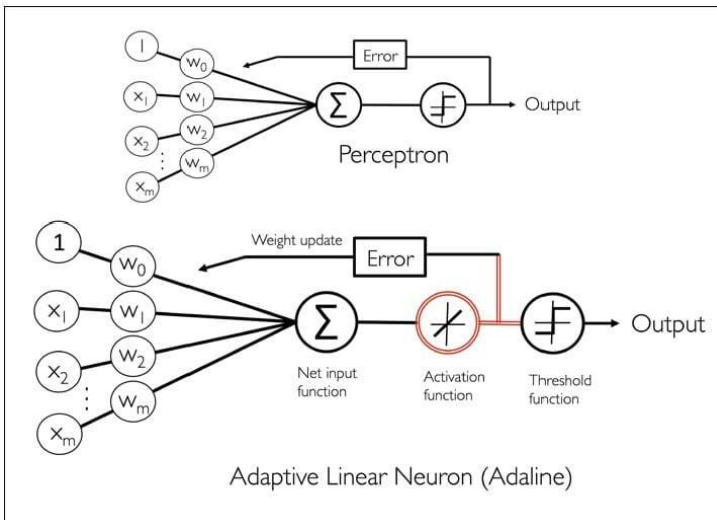
Ten prosty schemat uczenia sztucznego neuronu nie zatrzymał rozważań nad sztuczną inteligencją i pracy nad rozwojem teorii. W 1960 r. na Uniwersytecie Stanforda dwóch naukowców, BERNARD WIDROW oraz MARCIAN HOFF, opracowało model nazwany **ADALINE** (ang. **Adaptive Linear Neuron**).



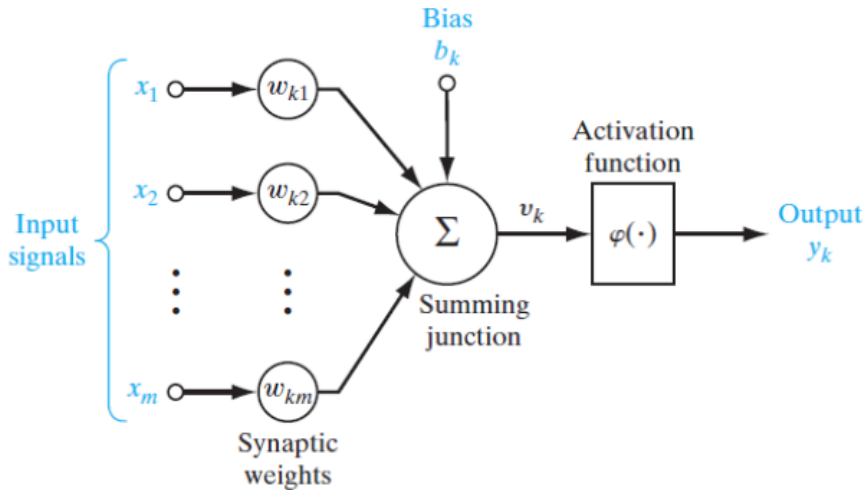
Bernard Widrow (1929-) & Marcian Edward „Ted” Hoff (1937-)

Model ADALINE jest udoskonaloną wersją perceptronu ROSENBLATTA. Przewaga algorytmu WIDROWA i HOFFA znajduje się w podejściu do wag, dla ADALINE są one dopasowywane na podstawie funkcji aktywacji, natomiast w przypadku perceptronu jest to funkcja skoku jednostkowego. Różnica ta daje przewagę adaptacyjnemu neuronowi liniowemu ponieważ wprowadzone zostało pojęcie minimalizacji funkcji kosztu, które znalazło zastosowanie w bardziej złożonych algorytmach uczenia maszynowego.

Początki sztucznych sieci neuronowych



Pojedynczy neuron

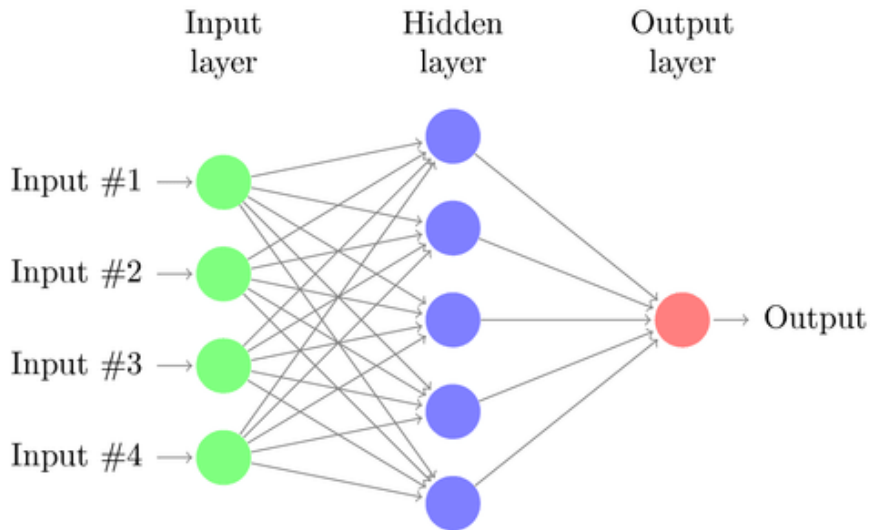


Tworzenie sztucznych sieci neuronowych jest oparte o mechanizm działania ludzkiego mózgu. Nieustannie trwają próby ulepszenia algorytmów mogących zbliżyć się użytecznością do tej, jaką pełni ludzki układ nerwowy. Znaczący wzrost efektywności w porównaniu do pojedynczego neuronu można zauważyć grupując kilka tych podstawowych jednostek w większą strukturę – **sieć**. To jak taka sieć będzie działać jest zależne od jej architektury, na którą składa się liczba neuronów, liczba warstw oraz w jaki sposób są one między sobą połączone. Najprostszym przykładem **sztucznej sieci neuronowej (SSN)** jest **jednokierunkowa sieć**, w której oprócz warstwy wejściowej występuje również **warstwa ukryta**. Można również rozbudować architekturę SSN o kolejne warstwy ukryte. Każda z warstw może składać się z innej liczby neuronów. Warstwy te są wzajemnie między sobą połączone.

Wektorem danych wejściowych dla warstwy pierwszej są surowe dane, natomiast dla kolejnych warstw (warstwy ukryte) na wejściu są przyjmowane aktywacje warstw poprzednich. W najbardziej podstawowej wersji SSN występują kolejno warstwy:

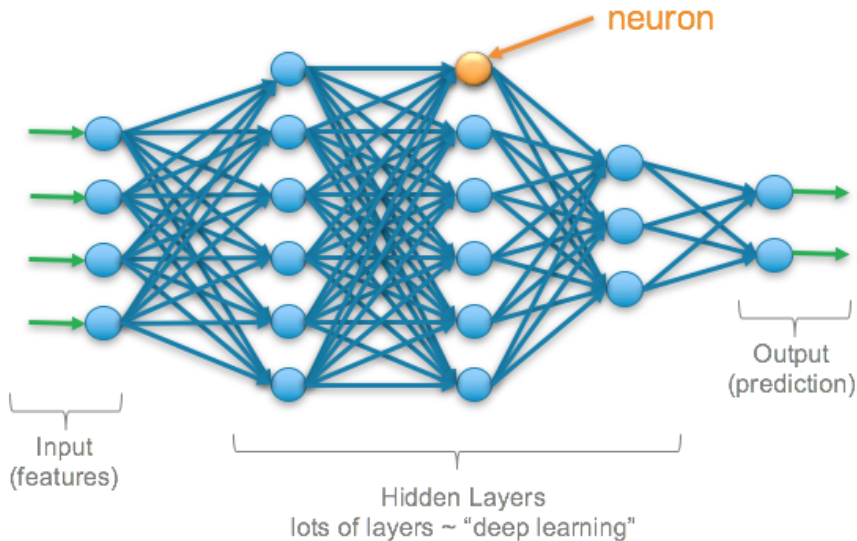
- 1 Warstwa wejściowa – jest odpowiedzialna za przyjęcie wektora danych do sieci. Liczba neuronów odpowiada zazwyczaj liczbie cech. Najczęściej jest w pełni połączona z pierwszą warstwą ukrytą.
- 2 Warstwa ukryta – na wejściu przyjmują aktywację warstwy wejściowej. Ma duże znaczenie przy rozwiązywaniu problemów nieliniowych, co jest niemożliwe w przypadku jednowarstwowego modelu.
- 3 Warstwa wyjściowa – tutaj uzyskuje się wynik działania całej sieci. Otrzymany wynik zależy głównie od funkcji aktywacji zastosowanej w SSN. Wynikiem działania może być wektor prawdopodobieństw przynależności (klasyfikacja) lub pojedyncza liczba (regresja).

Schemat sieci neuronowej



Założeniem głębokiego uczenia jest bardziej efektywne wykorzystanie i zastosowanie wcześniej wspomnianych sieci neuronowych. Tworzone są modele składające się z wielu warstw, aby można było uczyć je na danych o wielu poziomach abstrakcji. Większa liczba neuronów to jedna z kilku zasadniczych różnic w porównaniu do prostej sieci neuronowej. W głębokich sieciach neuronowych połączenia między neuronami są bardziej skomplikowane, a co za tym idzie, wymagana jest znacznie większa moc obliczeniowa aby taki model wytrenować.

Głębokie sieci neuronowe – idea



- „Płytkie” modele wymagają o wiele (wykładniczo) więcej neuronów
- „Płytkie” modele łatwiej się przetrenowują, nie są w stanie odszukać lepszego rozwiązania.
- Liczba neuronów podwaja się co 2.5 roku.

1 Uczenie nadzorowane

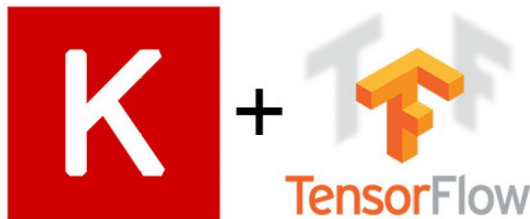
- **Klasyczne sieci głębokie (DNN)** – są to zwykłe sieci wielowarstwowe z dużą liczbą warstw (często zwane głębokie MLP).
- **Sieci konwolucyjne (CNN)** – wariant MLP inspirowany biologicznie, gdzie mnożenie macierzy wag i sygnału wejściowego zastąpione jest operacją splotu. Świetnie działają w przypadku obrazów, video itp.
- **Sieci rekurencyjne (RNN)** – posiadają połączenie zwrotne do poprzednich warstw. Świetnie nadają się do modelowania sygnałów: audio, tekst, ruch obiektów. Szczególnym przypadkiem są sieci **LSTM** (ang. Long Short-Term Memory).

2 Uczenie nienadzorowane

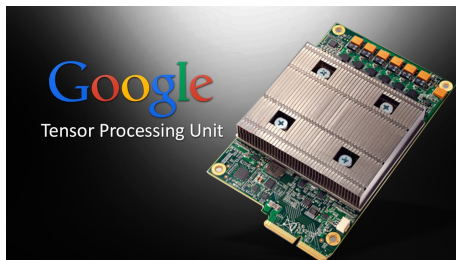
- **Autoenkodery (AE)** – sieci jednokierunkowe, których celem jest rekonstrukcja sygnału, kodowanie sygnału, usuwanie szumu.
- **Restricted Boltzman Machines (RBM)** – sieci modelujące rozkłady prawdopodobieństwa wejść. Głęboki wariant zazwyczaj jest nazywany **Deep Belief Networks (DBN)**.

3 Generative Adversarial Network (GAN) – dwie sieci (generująca i oceniająca) rywalizujące ze sobą w grze o sumie zerowej. Sieć oceniająca (dyskryminująca) stara się odróżnić prawdziwy sygnał od wygenerowanego przez sieć generującą. Sieć generująca) tworzy sygnał z pewnego rozkładu starając się „oszukać” sieć oceniającą, dąży do maksymalizacji błędu dyskryminacji. Sieci takie są stosowane do generowania realistycznych zdjęć.

Keras jest popularną biblioteką Open Source do tworzenia modeli sztucznych sieci neuronowych, ich uczenia i oceny skuteczności. Zasadniczo Keras jest wrapperem do biblioteki **TensorFlow** (bardzo rozbudowana biblioteka do uczenia maszynowego od Google), która jest silnikiem obliczeniowym dla głębokich sieci neuronowych. Jej bezpośrednie używanie jest jednak zdecydowanie trudniejsze. Keras napisany jest w języku Python, aczkolwiek można go również używać w R.



Siła TensorFlow drzemie w integracji z wszystkimi głównymi platformami chmurowymi, wliczając w to chmury Microsoft Azure, Amazon'a oraz Google. Poza tym można również liczyć na wbudowane wsparcie sprzętowe dla technologii wspomagających wielowątkowe przetwarzanie danych przez CPU/GPU oraz TPU! TPU czyli **Tensor Processing Units** to specjalne dedykowane chipy Google'a, zbudowane w jednym celu – wspomagać uczenie oraz ewaluację sieci neuronowych. Poza tym jest również narzędzie **TensorBoard**, które umożliwia nam wyświetlanie i analizowanie danych, które są wykorzystywane podczas uczenia



W ostatnim czasie pewną alternatywą dla bibliotek Keras/TensoFlow stała się biblioteka **PyTorch**. Jest to również biblioteka Open Source przeznaczona do uczenia maszynowego, stworzona przez oddział sztucznej inteligencji Facebooka. Można jej używać w językach Python oraz C++.

The PyTorch logo features the word "PYTORCH" in a bold, black, sans-serif font. The letter "O" is replaced by a stylized orange flame icon with a small purple flame at the top. The entire logo is centered on a white background.

PYTORCH

Deep Learning with Keras : : CHEAT SHEET



Intro

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. It supports multiple backends, including TensorFlow, CNTK and Theano.

TensorFlow is a lower level mathematical library for building deep neural network architectures. The Keras R package makes it easy to use Keras and TensorFlow in R.



<https://keras.rstudio.com>

<https://www.manning.com/books/deep-learning-with->

The "Hello, World!" of deep learning

INSTALLATION

The Keras R package uses the Python Keras library. You can install all the prerequisites directly from R.

https://keras.rstudio.com/reference/install_keras.html

```
library(keras)
install_keras()
```

See [install_keras](#) for GPU instructions

This installs the required libraries in an Anaconda environment or virtual environment 'r-tensorflow'.

Working with keras models

DEFINE A MODEL

keras_model() Keras Model

keras_model_sequential() Keras Model composed of a linear stack of layers

multi_gpu_model() Replicates a model on different GPUs

COMPILE A MODEL

compile(object, optimizer, loss, metrics = NULL) Configure a Keras model for training

FIT A MODEL

fit(object, x = NULL, y = NULL, batch_size = NULL, epochs = 10, verbose = 1, callbacks = NULL, ...) Train a Keras model for a fixed number of epochs (iterations)

fit_generator() Fits the model on data yielded batch-by-batch by a generator

train_on_batch() test_on_batch() Single gradient update or model evaluation over one batch of samples

EVALUATE A MODEL

evaluate(object, x = NULL, y = NULL, batch_size = NULL) Evaluate a Keras model

evaluate_generator() Evaluates the model on a data generator

PREDICT

predict() Generate predictions from a Keras model

predict_proba() and **predict_classes()** Generates probability or class probability predictions for the input samples

predict_on_batch() Returns predictions for a single batch of samples

predict_generator() Generates predictions for the input samples from a data generator

OTHER MODEL OPERATIONS

summary() Print a summary of a Keras model

export_savedmodel() Export a saved model

get_layer() Retrieves a layer based on either its name (unique) or index

pop_layer() Remove the last layer in a model

save_model_hdf5(); load_model_hdf5() Save/Load models using HDF5 files

serialize_model(); unserialize_model() Serialize a model to an R object

clone_model() Clone a model instance

freeze_weights(); unfreeze_weights() Freeze and unfreeze weights

CORE LAYERS

layer_input() Input layer

layer_dense() Add a densely-connected NN layer to an output

layer_activation() Apply an activation function to an output

layer_dropout() Applies Dropout to the input

layer_reshape() Reshapes an output to a certain shape

layer_permute() Permute the dimensions of an input according to a given pattern

layer_repeat_vector() Repeats the input n times

layer_lambda(object, f) Wraps arbitrary expression as a layer

layer_activity_regularization() Layer that applies an update to the cost function based input activity

layer_masking() Masks a sequence by using a mask value to skip timesteps

layer_flatten() Flattens an input

input layer: use MNIST images

```
mnist <- dataset_mnist()
x_train <- mnist$train$; y_train <- mnist$train$y
x_test <- mnist$test$; y_test <- mnist$test$y
```

3047

reshape and rescale

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
x_train <- x_train / 255; x_test <- x_test / 255
```

```
y_train <- to_categorical_train(10)
y_test <- to_categorical_test(10)
```

defining the model and layers

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu',
    input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
```

compile (define loss and optimizer)

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy'))
```

train (fit)

```
model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2)

















model %>% evaluate(x_test, y_test)
model %>% predict_classes(x_test)
```















RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • <https://rstudio.com> • 844-448-1212 • info@rstudio.com • Learn more at keras.rstudio.com • Keras 2.1.2 • Updated: 2017-12

More layers






CONVOLUTIONAL LAYERS

-  **layer_conv_1d()** 1D, e.g. temporal convolution
Apply an activation function to an output
-  **layer_conv_2d_transpose()**
Transposed 2D (deconvolution)
-  **layer_conv_2d()** 2D, e.g. spatial convolution over images
-  **layer_conv_3d_transpose()**
Transposed 3D (deconvolution)
-  **layer_conv_3d()** 3D, e.g. spatial convolution over volumes
-  **layer_conv_lstm_2d()**
Convolutional LSTM
-  **layer_separable_conv_2d()**
Depthwise separable 2D
-  **layer_upsampling_1d()**
Upsampling layer
-  **layer_upsampling_2d()**
Upsampling layer
-  **layer_upsampling_3d()**
Upsampling layer
-  **layer_zero_padding_1d()**
Zero-padding layer
-  **layer_zero_padding_2d()**
Zero-padding layer
-  **layer_zero_padding_3d()**
Zero-padding layer
-  **layer_cropping_1d()**
Cropping layer
-  **layer_cropping_2d()**
Cropping layer
-  **layer_cropping_3d()**
Cropping layer





POOLING LAYERS

-  **layer_max_pooling_1d()**
Maximum pooling for 1D to 3D
-  **layer_max_pooling_2d()**
Maximum pooling for 1D to 3D
-  **layer_max_pooling_3d()**
Maximum pooling for 1D to 3D
-  **layer_average_pooling_1d()**
Average pooling for 1D to 3D
-  **layer_average_pooling_2d()**
Average pooling for 1D to 3D
-  **layer_average_pooling_3d()**
Average pooling for 1D to 3D
-  **layer_global_max_pooling_1d()**
Global maximum pooling
-  **layer_global_max_pooling_2d()**
Global maximum pooling
-  **layer_global_max_pooling_3d()**
Global maximum pooling
-  **layer_global_average_pooling_1d()**
Global average pooling
-  **layer_global_average_pooling_2d()**
Global average pooling
-  **layer_global_average_pooling_3d()**
Global average pooling


ACTIVATION LAYERS

-  **layer_activation(object, activation)**
Apply an activation function to an output
-  **layer_activation_leaky_relu()**
Leaky version of a rectified linear unit
-  **layer_activation_parametric_relu()**
Parametric rectified linear unit
-  **layer_activation_thresholded_relu()**
Thresholded rectified linear unit
-  **layer_activation_elu()**
Exponential linear unit

DROPOUT LAYERS

-  **layer_dropout()**
Applies dropout to the input
-  **layer_spatial_dropout_1d()**
Spatial 1D to 3D version of dropout
-  **layer_spatial_dropout_2d()**
Spatial 1D to 3D version of dropout
-  **layer_spatial_dropout_3d()**
Spatial 1D to 3D version of dropout

RECURRENT LAYERS

-  **layer_simple_rnn()**
Fully-connected RNN where the output is to be fed back to input
-  **layer_gru()**
Gated recurrent unit - Cho et al
-  **layer_cudnn_gru()**
Fast GRU implementation backed by CuDNN
-  **layer_lstm()**
Long-Short Term Memory unit - Hochreiter 1997
-  **layer_cudnn_lstm()**
Fast LSTM implementation backed by CuDNN

LOCALLY CONNECTED LAYERS

-  **layer_locally_connected_1d()**
Similar to convolution, but weights are not shared, i.e. different filters for each patch
-  **layer_locally_connected_2d()**
Similar to convolution, but weights are not shared, i.e. different filters for each patch

Preprocessing

SEQUENCE PREPROCESSING

- pad_sequences()**
Pads each sequence to the same length (length of the longest sequence)
- skipgrams()**
Generates skipgram word pairs
- make_sampling_table()**
Generates word rank-based probabilistic sampling table

TEXT PREPROCESSING

- text_tokenizer()** Text tokenization utility
- fit_text_tokenizer()** Update tokenizer internal vocabulary
- save_text_tokenizer(); load_text_tokenizer()**
Save a text tokenizer to an external file
- texts_to_sequences(); texts_to_sequences_generator()**
Transforms each text in texts to sequence of integers
- texts_to_matrix(); sequences_to_matrix()**
Convert a list of sequences into a matrix
- text_one_hot()** One-hot encode text to word indices
- text_hashing_trick()**
Converts a text to a sequence of indexes in a fixed-size hashing space
- text_to_word_sequence()**
Convert text to a sequence of words (or tokens)

IMAGE PREPROCESSING

- image_load()** Loads an image into PIL format.
- flow_images_from_data()**
flow_images_from_directory()
Generates batches of augmented/normalized data from images and labels, or a directory
- image_data_generator()** Generate minibatches of image data with real-time data augmentation.
- fit_image_data_generator()** Fit image data generator internal statistics to some sample data
- generator_next()** Retrieve the next item
- image_to_array(); image_array_resize()**
image_array_save() 3D array representation



Pre-trained models

Keras applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

- application_xception()**
xception_preprocess_input()
Xception v1 model
- application_inception_v3()**
inception_v3_preprocess_input()
Inception v3 model, with weights pre-trained on ImageNet
- application_inception_resnet_v2()**
inception_resnet_v2_preprocess_input()
Inception-ResNet v2 model, with weights trained on ImageNet
- application_vgg16(); application_vgg19()**
VGG16 and VGG19 models
- application_resnet50()** ResNet50 model
- application_mobilenet()**
mobilenet_preprocess_input()
mobilenet_decode_predictions()
mobilenet_load_model_hdf5()
MobileNet model architecture

IMAGENET

imageNet is a large database of images with labels, extensively used for deep learning

- imagenet_preprocess_input()**
imagenet_decode_predictions()
Preprocesses a tensor encoding a batch of images for ImageNet, and decodes predictions

Callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.

- callback_early_stopping()** Stop training when a monitored quantity has stopped improving
- callback_learning_rate_scheduler()** Learning rate scheduler
- callback_tensorboard()** TensorBoard basic visualizations



RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • <https://www.rstudio.com> • 944-448-1212 • info@rstudio.com • Learn more at [rstudio.com](https://www.rstudio.com) • keras 2.1.2 • Updated: 2017-12