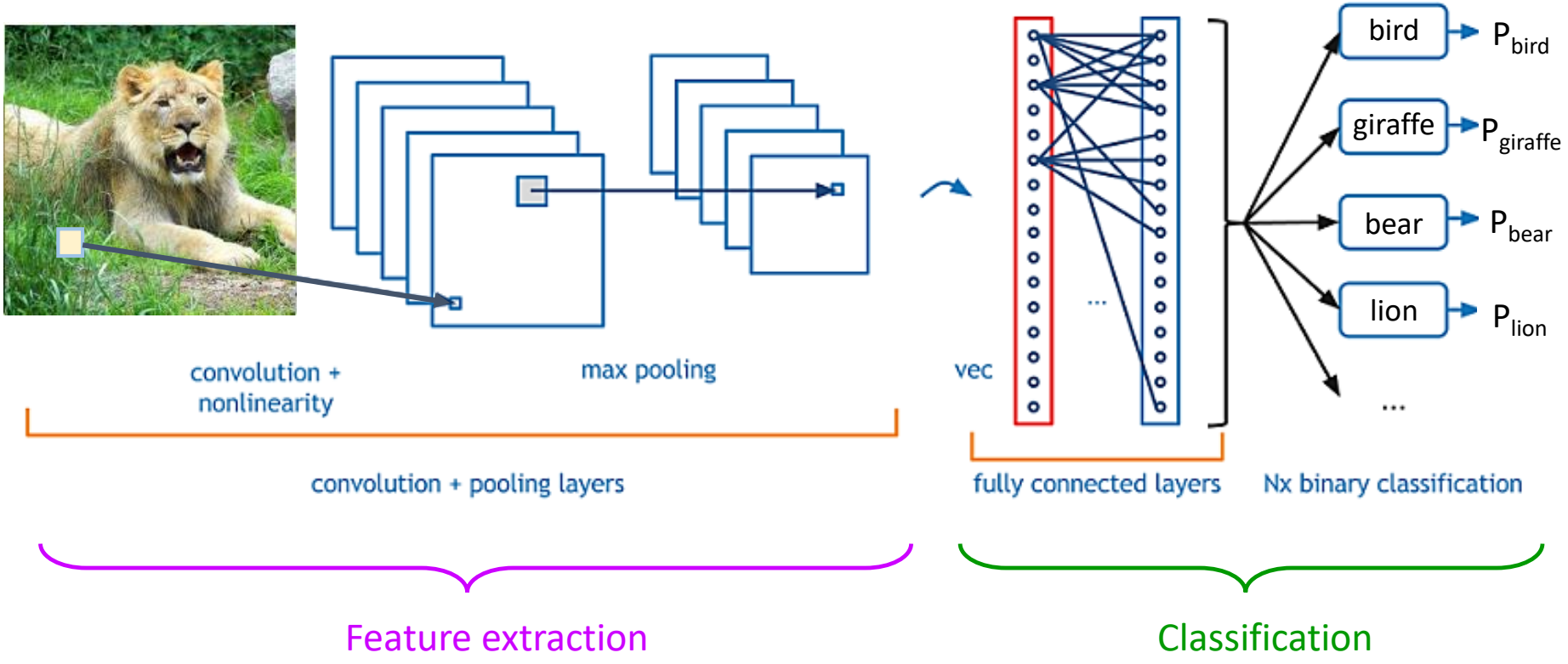
# CNN Implementation

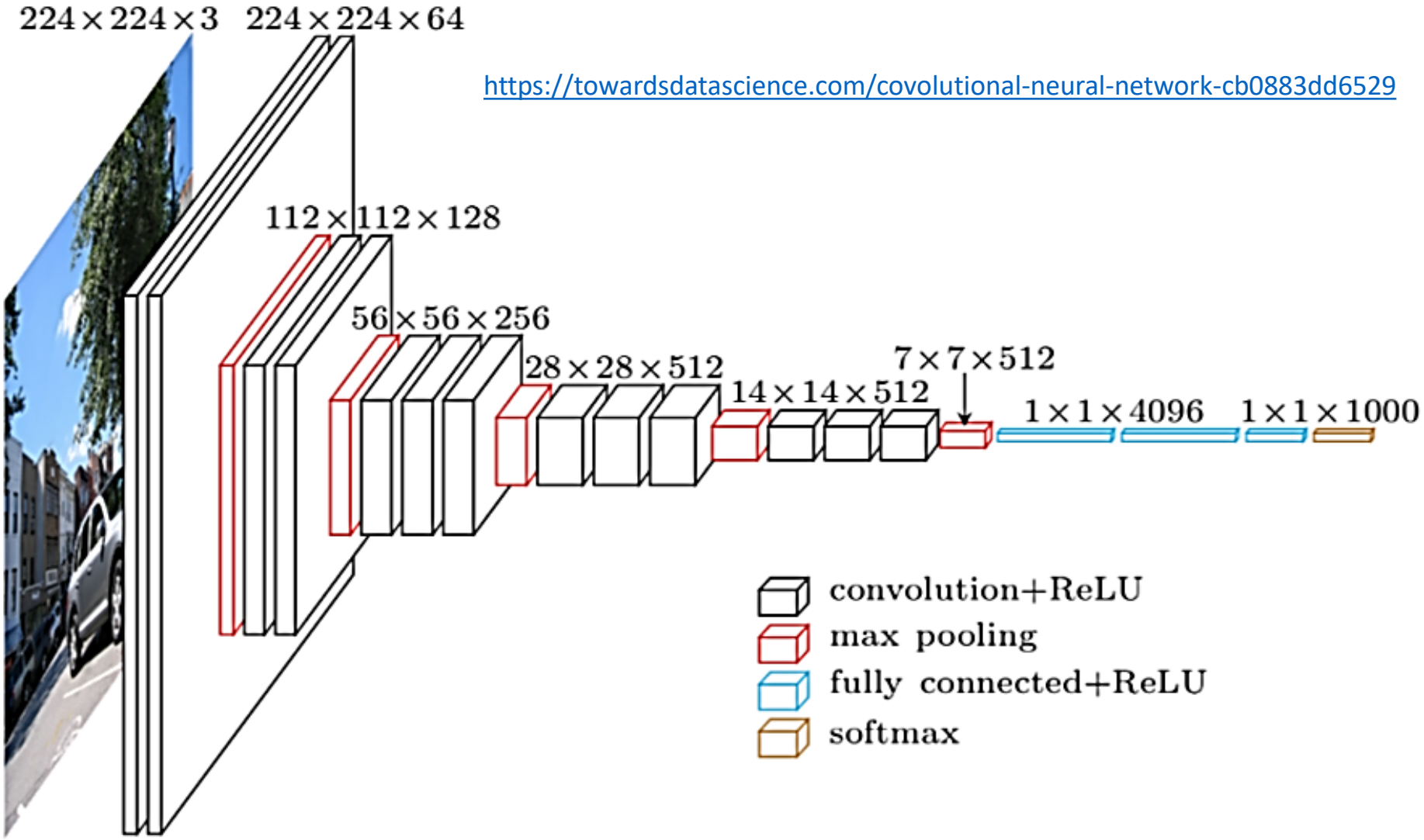
## Python, PyTorch Colaboratory

# Convolutional Neural Network illustration




Adaptation after: https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529

# Downsampling illustration




https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529

**The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers, and normalization layers**

# **Case study:** CIFAR-10 dataset

https://www.cs.toronto.edu/~kriz/cifar.html

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class.

There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images.

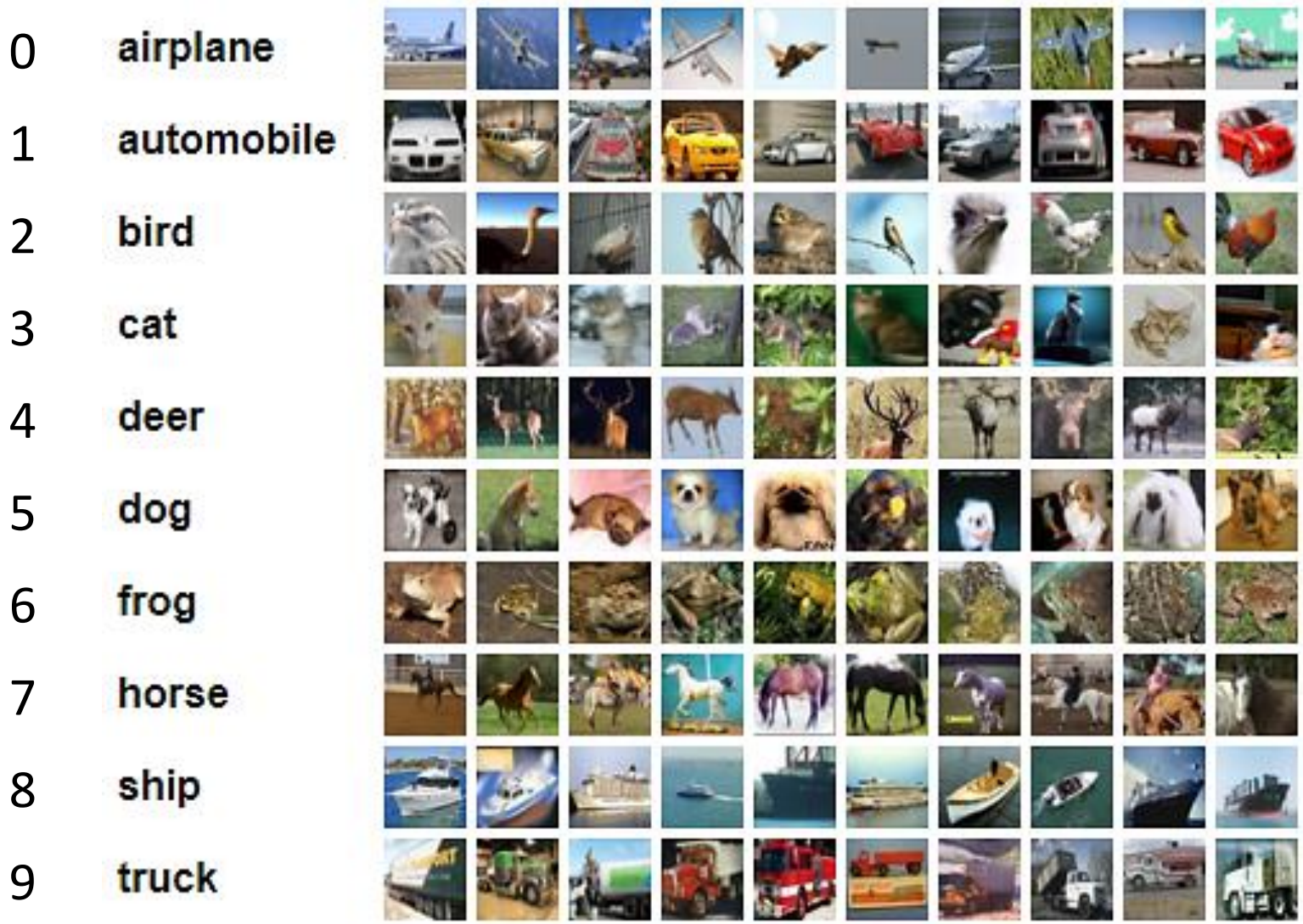
The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another.

Between them, the training batches contain exactly 5000 images from each class.

The test batch contains exactly 1000 randomly-selected images from each class.

# **Case study:** CIFAR-10 dataset

https://www.cs.toronto.edu/~kriz/cifar.html



# Development environment
# Google Colab

Welcome to Colaboratory!

https://colab.research.google.com/notebooks/welcome.ipynb

**Colaboratory** is a free **Jupyter notebook environment** that requires **no setup** and runs entirely in the **cloud**.

With Colaboratory we can write and execute code, save and share our analyses, and access powerful computing resources, all **for free from our browser**.

# Development environment
# Google Colab

Google Colab is an executable document that lets us write, run, and share code within Google Drive.

A notebook document is composed of cells, each of which can contain code, text, images, and more.

Colab connects the notebook to a cloud-based runtime, meaning we can execute Python code without any required setup on our own machine.

Additional code cells are executed using the same runtime, resulting in an interactive coding experience in which we can use any of the functionality that Python offers.

# Deep Learning Frameworks

## GOOGLE'S TENSORFLOW

TensorFlow is open source deep learning framework created by developers at Google and released in 2015. The official research is published in the paper "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems."
TensorFlow is now widely used by companies, startups, and business firms to automate things and develop new systems. It draws its reputation from its distributed training support, scalable production and deployment options, and support for various devices like Android.

## FACEBOOK'S PYTORCH

PyTorch is one of the latest deep learning frameworks and was developed by the team at Facebook and open sourced on GitHub in 2017. You can read more about its development in the research paper "Automatic Differentiation in PyTorch."
PyTorch is gaining popularity for its simplicity, ease of use, dynamic computational graph and efficient memory usage.

https://builtin.com/data-science/pytorch-vs-tensorflow November 6, 2019

# Deep Learning Frameworks

## TENSORFLOW VS PYTORCH: RECOMMENDATION

TensorFlow is a very powerful and mature deep learning library with strong visualization capabilities and several options to use for high-level model development. It has production-ready deployment options and support for mobile platforms.

PyTorch, on the other hand, is still a young framework with stronger community movement and it's more Python friendly.

If you want to make things faster and build AI-related products, TensorFlow is a good choice.

PyTorch is mostly recommended for research-oriented developers as it supports fast and dynamic training.

https://builtin.com/data-science/pytorch-vs-tensorflow November 6, 2019

# Application flowchart

Uses PyTorch

CNN implementation for CIFAR10

- Import modules
- Mount the gdrive for data files
- Import dataset
- Explore dataset
- Prepare the dataset for training
  - Define dataset loaders
- Define our CNN model (CNN arhitecture)
  - Initializing the device
- **Train our CNN model - main part of the program**
  - Testing the model (inference)
  - Plot confusion matrix

# CNN model

The input of a Pytorch Neural Network is:

[BATCH_SIZE] * [CHANNEL_NUMBER] * [HEIGHT] * [WIDTH]

49 * 3 * 32 *32 in our implementation

The resulting size of the images after a convolution layer is

$$W = \frac{W - F + 2P}{S} + 1 \qquad H = \frac{H - F + 2P}{S} + 1$$

$W$ - WIDTH
$H$ - HEIGHT
$F$ - FILTER SIZE
$P$ - PADDING
$S$ - STRIDE

# Initial CIFAR10 images for pytorch: 3 x 32 x 32

channel = 3; W = H = 32; batch_size = 49

```
import torch.nn as nn
import torch.nn.functional as F
```

nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')

F.max_pool2d(input, self.kernel_size, self.stride, self.padding, self.dilation, self.ceil_mode, self.return_indices)

```
# 1st convolution + batch normalization + rectification + max pooling

    x = self.convol1(x)
    x = self.bnorm1(x)
    x = F.relu(x)
    x = F.max_pool2d(x, kernel_size=2, stride=2)
```

Initial CIFAR10 images for pytorch: 3 x 32 x 32

channel = 3; W = H = 32; batch_size = 49

```
import torch.nn as nn
import torch.nn.functional as F
```

nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')

F.max_pool2d(input, self.kernel_size, self.stride, self.padding, self.dilation, self.ceil_mode, self.return_indices)

**1st convolution + pooling layer**

```
nn.Conv2d(3, 16, 7, stride = 1, padding=3, bias=True)
```

Filter size, F = 7; Stride, S = 1; Padding, P = 3

$$W = H = \frac{32 - 7 + 2 * 3}{1} + 1 = 32$$

Feature maps (output channels) = 16

```
F.max_pool2d(x, kernel_size=2, stride=2)
```

$$W = H = \frac{32}{2} = 16$$

**Output: 16 x 16 x 16**

## 2nd convolution + pooling layer

```
nn.Conv2d(16,32,5, padding=2, bias=True)
```

W = H = 16; F = 5; S = 1 (default); P = 2

$$W = H = \frac{16 - 5 + 2 * 2}{1} + 1 = 16$$

Feature maps (output channels) = 32

```
F.max_pool2d(x, kernel_size=2, stride=2)
```

$$W = H = \frac{16}{2} = 8$$

**Output: 32 x 8 x 8**

## 3rd convolution + pooling layer

```
nn.Conv2d(32,64,3, padding=1, bias=True)
```

W = H = 8; F = 3; S = 1 (default); P = 1

$$W = H = \frac{8 - 3 + 2 * 1}{1} + 1 = 8$$

Feature maps (output channels) = 64

```
F.max_pool2d(x, kernel_size=2, stride=2)
```

$$W = H = \frac{8}{2} = 4$$

**Output: 64 x 4 x 4**

# Fully connected layers

**Flattening –** reshape the tensor for the fully connected layers;
the total number of elements needs to remain the same

```
x_fc = x.view(-1, 4*4*64)
```

x – the output data for the last convolution + pooling layer
49 x 64 x 4 x 4

x_fc – data for the input of the first hidden layer

-1 : number of rows to be automatically computed

4*4*64 – number of columns (extracted features for one image)

For this example with batch_size = 49

x_fc has 49 rows and 1024 columns

## 1st fully connected layer

```
nn.Linear(4*4*64, 100, bias=True)
```

100 neurons (100 outputs) that should be connected to the input with 4*4*64 = 1024 outputs from the previous layer

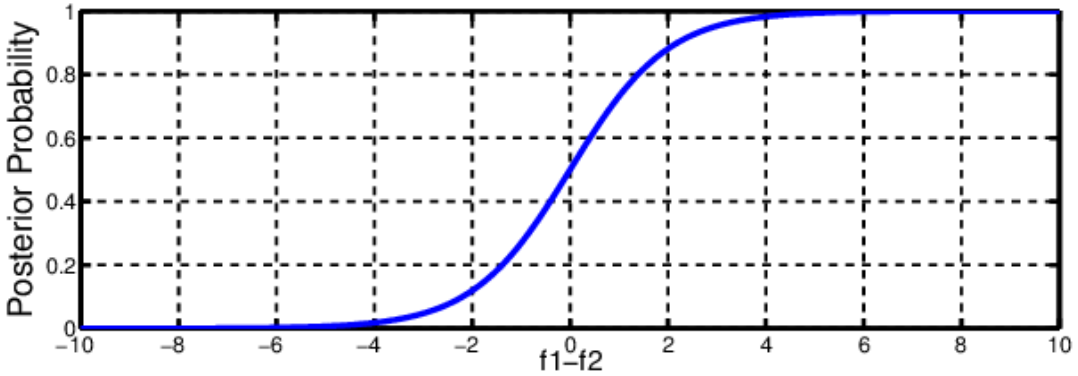
## 2nd fully connected layer (the last fc layer, output neural network layer)

```
nn.Linear(100, 10, bias=True)
```

10 neurons (10 outputs) that should be connected to the input with 100 outputs from the previous layer

The number of neurons = number of classes

# Softmax activation function




**Softmax** (normalized exponential function) takes as input a vector of *K* real numbers, and normalizes it into a **probability distribution** consisting of *K* probabilities proportional to the exponentials of the input numbers.

- Prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1;
- After applying softmax, each component will be in the interval (0, 1), and the components will add up to 1, so that they can be interpreted as probabilities. Also, this additional constraint helps training converge more quickly than it otherwise would.
- The larger input components will correspond to larger probabilities.

Softmax assumes that each example is a member of exactly one class

## Softmax layer

Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0.

torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range [0, 1] and sum to 1

$$Softmax(s_i) = y_i = \frac{e^{s_i}}{\sum_i e^{s_i}} \qquad \sum_i y_i = 1$$

$s_i$ - the output of the last fully connected layer

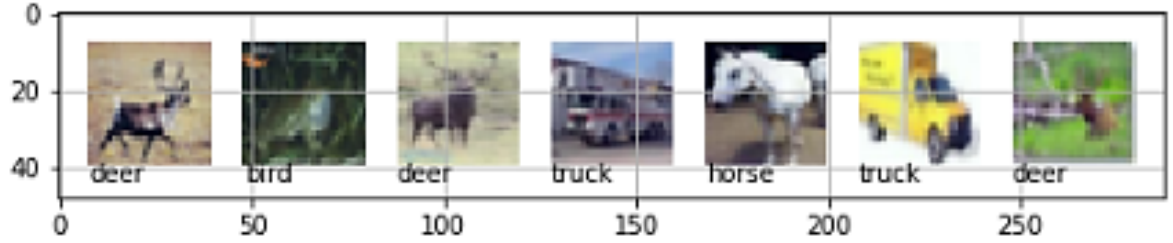
$y_i$ - the output of the last softmax layer

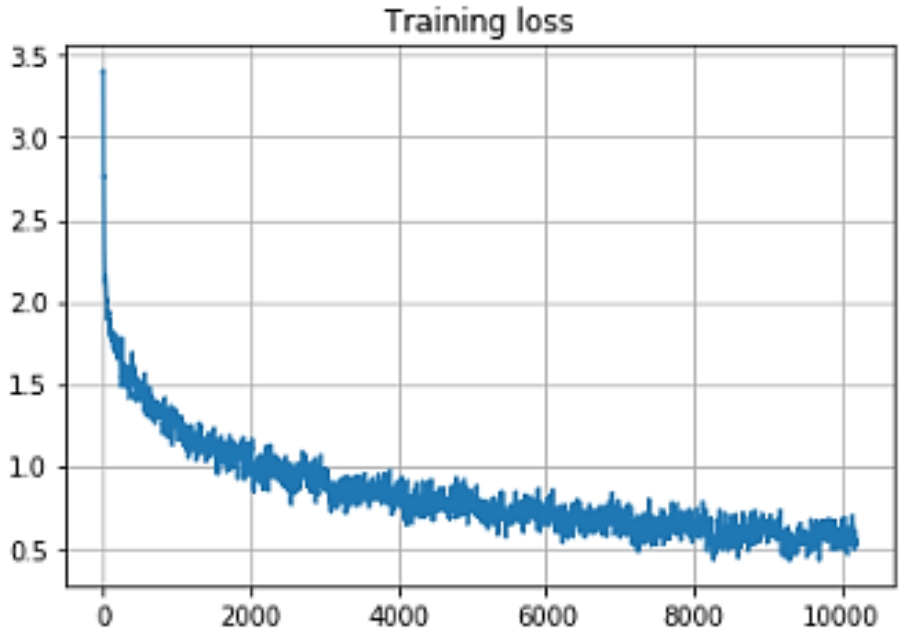
## Log Softmax

torch.nn.functional.log_softmax(input, dim=None, _stacklevel=3, dtype=None)

$$LogSoftmax(s_i) = y_i = \log\left(\frac{e^{s_i}}{\sum_i e^{s_i}}\right) = log(e^{s_i}) - log\left(\sum_i e^{s_i}\right)$$

```
Train Epoch: 10 [49000/50000 (98%)]     Loss: 0.663238
```



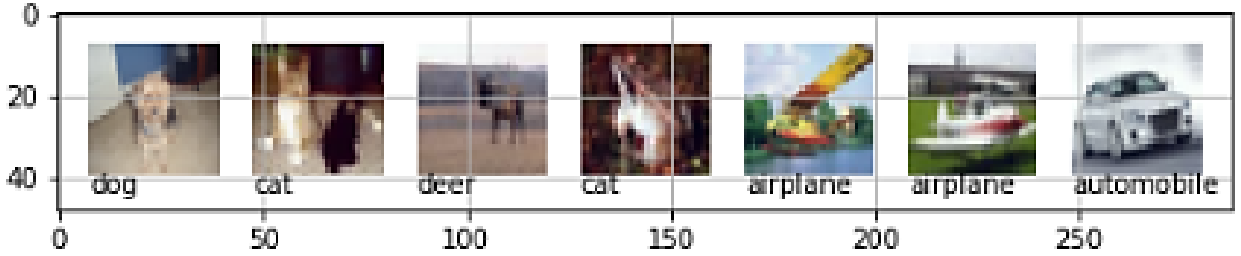

# Results

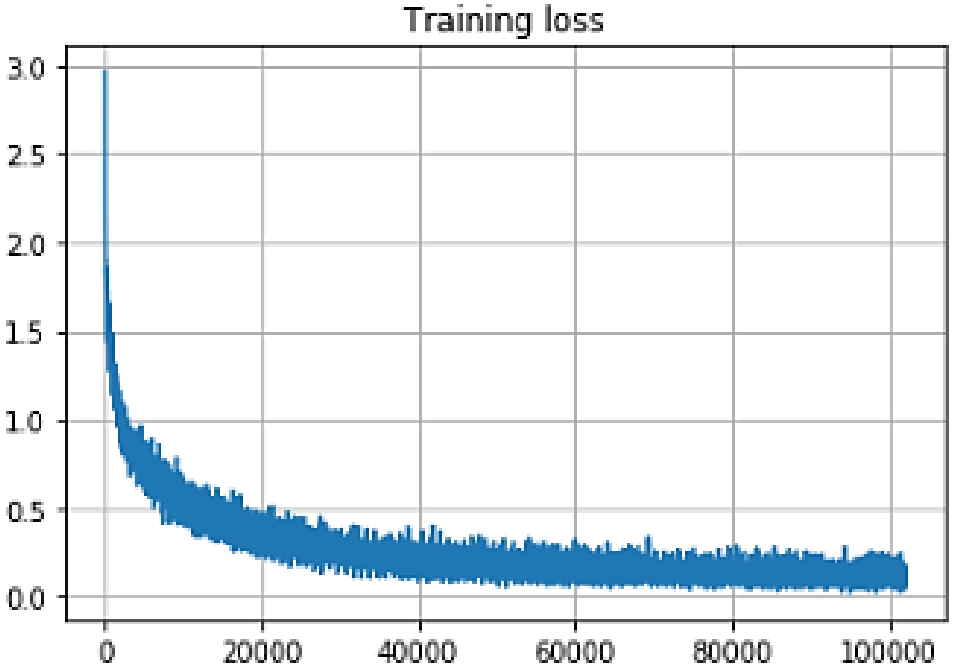
- 10 training epochs
- 4 convolutional layers

```
Lenght of the losses array is: 10201
Losses:  [3.39832821 3.41186922 2.98404746 ... 0.51017985 0.50905155 0.52241483]
```

```
Test set: Average loss: 2.3917, Accuracy: 7285/10000 (73%)
```

```
Train Epoch: 100 [49000/50000 (98%)]    Loss: 0.068332
```




# Results

- 100 training epochs
- 4 convolutional layers

```
Lenght of the losses array is: 102091
Losses:  [2.96476409 2.95061927 2.67629166 ... 0.10392949 0.10487079 0.11074336]
```

```
Test set: Average loss: 2.3917, Accuracy: 7285/10000 (73%)
```

# Confusion matrix

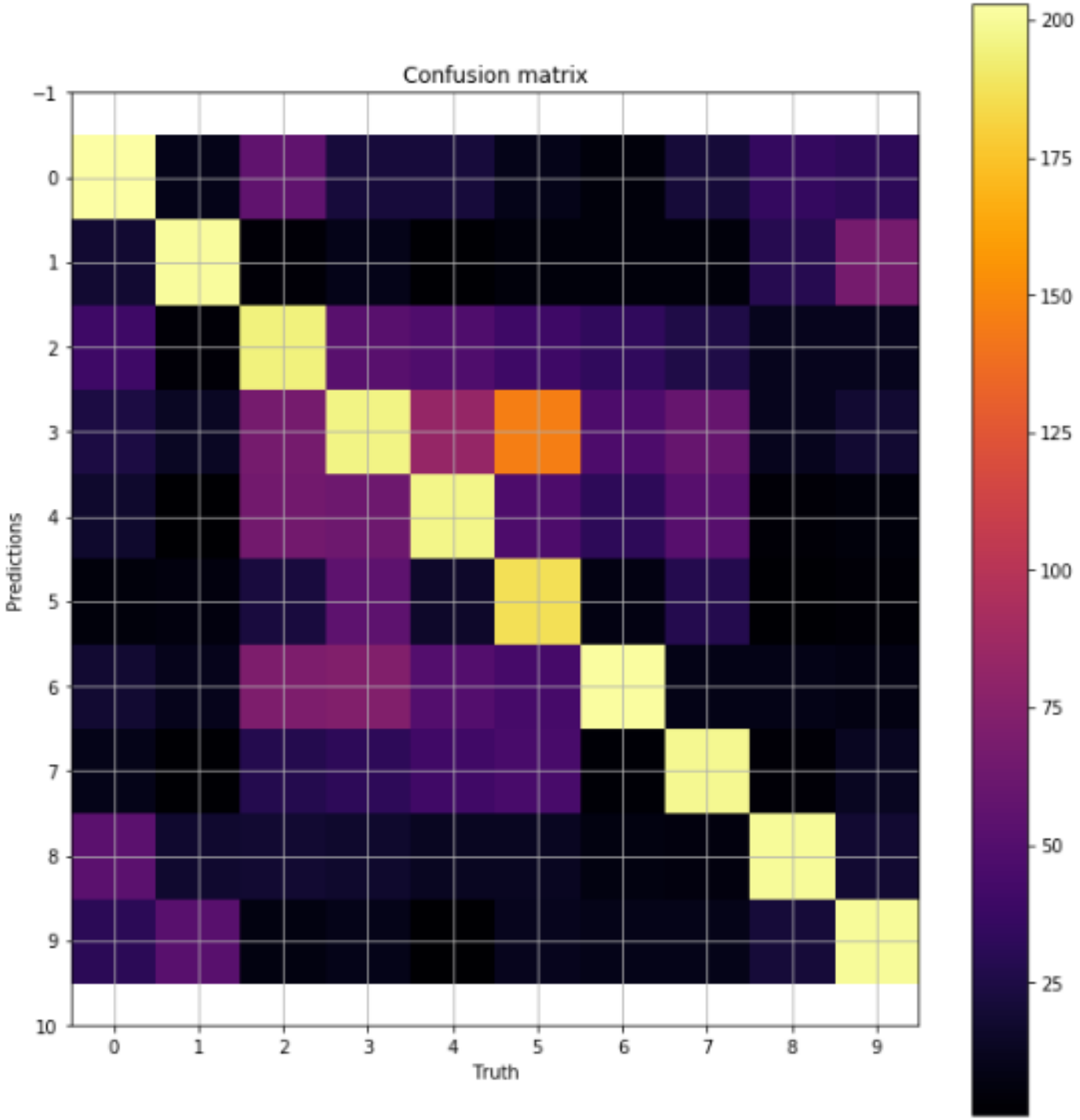
- 100 training epochs
- 4 convolutional layers

| PREDICTED | airplane | automobile | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| airplane | 203 | 10 | 57 | 22 | 22 | 10 | 5 | 21 | 35 | 32 |
| automobile | 19 | 201 | 4 | 10 | 2 | 5 | 5 | 5 | 29 | 67 |
| bird | 40 | 4 | 195 | 53 | 48 | 40 | 34 | 26 | 12 | 12 |
| cat | 24 | 14 | 67 | 196 | 83 | 146 | 47 | 60 | 12 | 19 |
| deer | 17 | 1 | 65 | 63 | 197 | 47 | 32 | 52 | 4 | 5 |
| dog | 5 | 6 | 23 | 55 | 16 | 187 | 8 | 28 | 2 | 4 |
| frog | 19 | 11 | 71 | 73 | 50 | 44 | 202 | 9 | 9 | 8 |
| horse | 10 | 2 | 28 | 32 | 41 | 45 | 3 | 199 | 4 | 13 |
| ship | 54 | 18 | 19 | 17 | 13 | 13 | 7 | 6 | 200 | 19 |
| truck | 31 | 53 | 7 | 10 | 1 | 12 | 10 | 10 | 21 | 200 |

TRUTH

# Confusion matrix

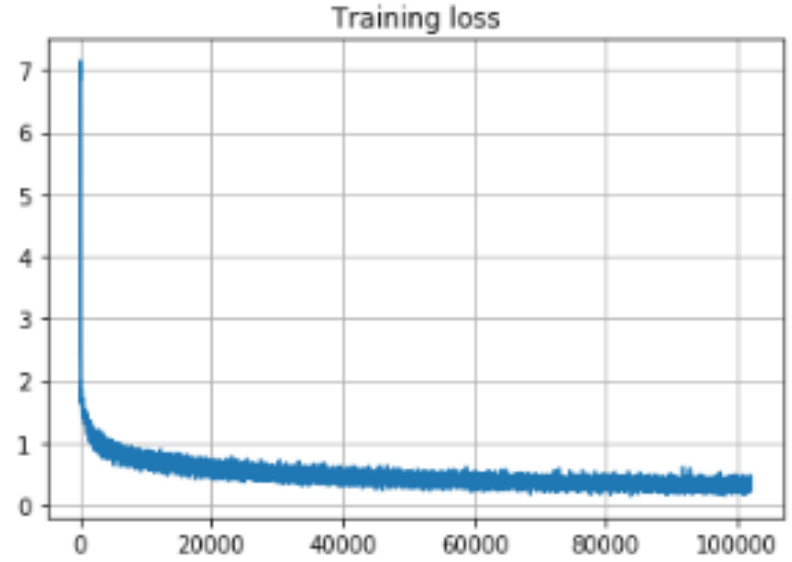
- 100 training epochs
- 4 convolutional layers



```
Train Epoch: 100 [0/50000 (0%)] Loss: 0.272444

Train Epoch: 100 [24500/50000 (49%)]    Loss: 0.264038

Train Epoch: 100 [49000/50000 (98%)]    Loss: 0.188240
```




# Results

- 100 training epochs
- 3 convolutional layers

```
Lenght of the losses array is: 102091
Losses:  [6.85915077 7.14504895 6.5125247  ... 0.23784495 0.24462851 0.23556667]
```

```
Test set: Average loss: 1.0609, Accuracy: 6285/10000 (63%)
```

**Results for the last 4 images in the testset**

10000 images;  batch_size = 49;

10000 / 49 = 204.082;  204 full batches;

10000 – 204*49 = 10000 – 9996 = 4

**Output  (log softmax)**

```
[[-3.2353e+01, -4.7827e+01, -2.4556e+01, -4.3907e-03, -1.4375e+01,
-5.4307e+00, -1.3595e+01, -2.8390e+01, -4.4619e+01, -3.2801e+01],

[-2.5058e+01, -4.2708e+01, -2.1051e+01, -7.3828e+00, -2.2477e+01,
-6.2180e-04, -2.0211e+01, -1.9865e+01, -3.3112e+01, -3.6092e+01],

[-1.9073e-06, -1.9862e+01, -2.9568e+01, -1.2988e+01, -1.6823e+01,
-2.4511e+01, -1.4282e+01, -1.7671e+01, -2.1944e+01, -3.3232e+01],

[-5.4293e+01, -4.9853e+01, -2.4284e+01, -3.3939e+01, -1.5435e+01,
-2.5975e+01, -4.0547e+01, 0.0000e+00, -5.2675e+01, -4.8895e+01]]

Predicted labels for classes
[[3],
 [5],
 [0],
 [7]]
```

**Output (log softmax)**

```
[[-3.2353e+01, -4.7827e+01, -2.4556e+01, -4.3907e-03, -1.4375e+01,
-5.4307e+00, -1.3595e+01, -2.8390e+01, -4.4619e+01, -3.2801e+01],

[-2.5058e+01, -4.2708e+01, -2.1051e+01, -7.3828e+00, -2.2477e+01,
-6.2180e-04, -2.0211e+01, -1.9865e+01, -3.3112e+01, -3.6092e+01],

[-1.9073e-06, -1.9862e+01, -2.9568e+01, -1.2988e+01, -1.6823e+01,
-2.4511e+01, -1.4282e+01, -1.7671e+01, -2.1944e+01, -3.3232e+01],

[-5.4293e+01, -4.9853e+01, -2.4284e+01, -3.3939e+01, -1.5435e+01,
-2.5975e+01, -4.0547e+01, 0.0000e+00, -5.2675e+01, -4.8895e+01]]
```

```
Predicted labels for classes
[[3],
 [5],
 [0],
 [7]]
```


| | |
|---|---|
| 0 | airplane |
| 1 | automobile |
| 2 | bird |
| 3 | cat |
| 4 | deer |
| 5 | dog |
| 6 | frog |
| 7 | horse |
| 8 | ship |
| 9 | truck |

Analyze the output (log softmax value) for each image in respect with every class

# Using the Notebook file

This is a link to a read-only version of the application notebook:

CNN_ICSDC_CIFAR10.ipynb

https://colab.research.google.com/drive/191AIF7hliCrO-wqTMCW0Z39XRch6ZswR

You may access this link using your google account and open it in playground mode.

In playground mode you can execute or edit the notebook.

You can save a copy of the notebook (in your Google drive) after opening it in playground mode.

# Converting a Jupyter Notebook file in a Python script

jupyter and nbconvert should be installed

> jupyter nbconvert

The application is used to convert notebook files (*.ipynb) to various other formats.

The user can specify the export format with `--to`.

Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook', 'pdf', '**python**', 'rst', 'script', 'slides'].

> jupyter nbconvert --to python mynotebook.ipynb

Generates the mynotebook.py file