Deep Learning with Keras in R

Michał Maj

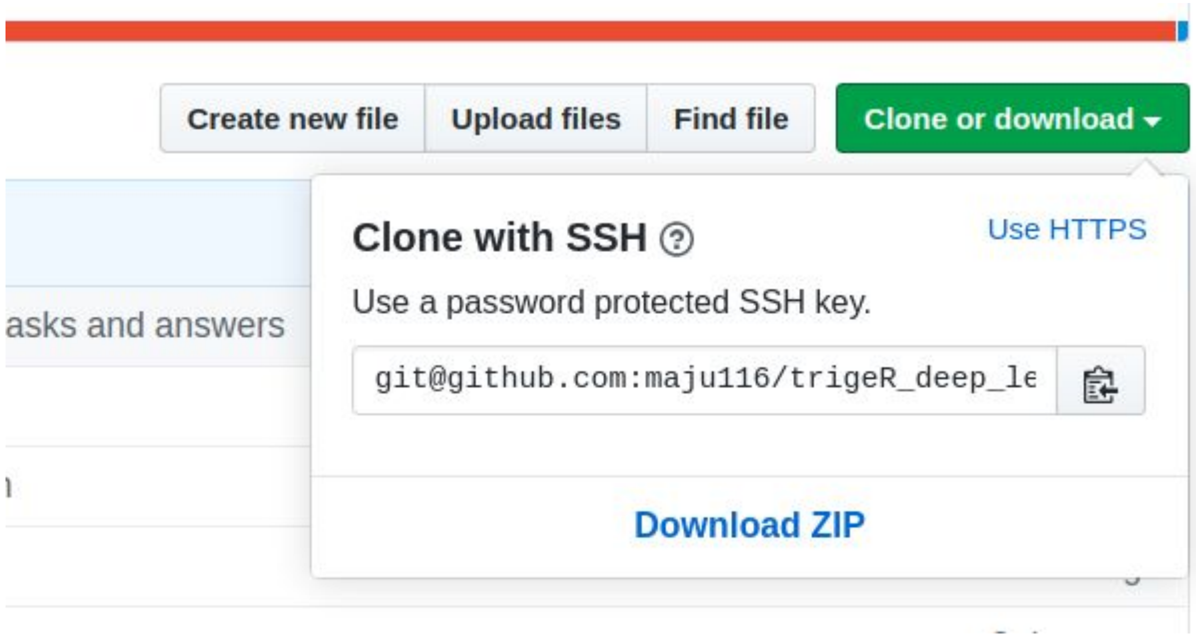
14.01.2020

Requirements

- your own laptop with up to date R version and RStudio installed
- tensorflow and keras installed: install_keras() function
- basic R language knowledge: objects and functions
- basic ML knowledge: linear/logistic regression, MSE, Accuracy
- scripts and data

Scripts and data

https://github.com/maju116/trigeR_deep_learning_woth_keras_in_R



Clone with SSH

Use a password protected SSH key.

```
git@github.com:maju116/trigeR_deep_le
```

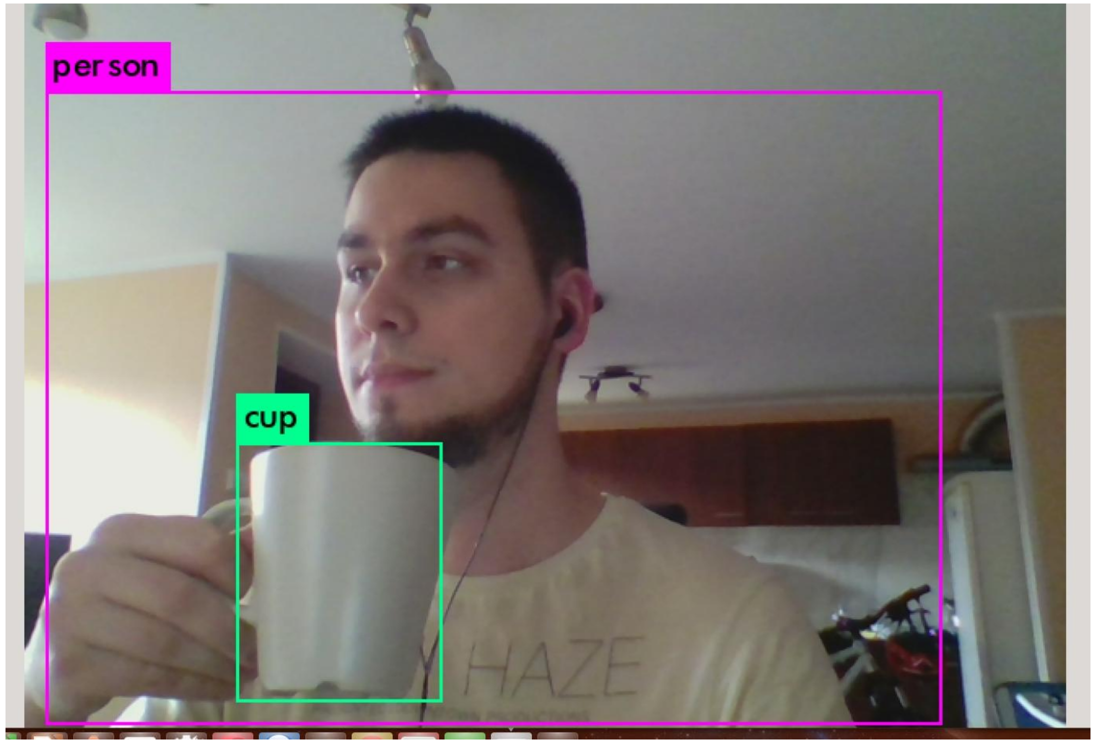
Unfortunately there's no internet connection for now!

Who am I

My name is Michał Maj:

- Linkedin https://www.linkedin.com/in/michal-maj116/
- Twitter @MichalMaj116

I am a Data Scientist at Billenium.



I am interested in machine/deep learning and statistics. I love new challenges and I'm always ready to help solving data science problems. I'm a big R language enthusiast and a co-organizer R Enthusiasts meetups in Gdańsk (https://www.meetup.com/Trojmiejska-Grupa-Entuzjastow-R/). Currently trying to become a deep learning expert!

What will you learn ?

- What is **Tensorflow**, **Keras** and other supporting tools in R ?
- What is **MLP** (Multilayer perceptron) ?
- What is **CNN** (Convolutional Neural Network) ?
- What is **RNN** (Recurrent Neural Network ?
- What is **fine-tuning** and how to use it ?
- What is **data augmentation** ?
- How **SGD** (Stochastic Gradient Descent) works ?
- How to use MLP, CNN, and RNN for different tasks like, image segmentation, object detection, text generation, NLP, and many more…
- … (we will see during the course)

Agenda

15.01.2020:

- What is **Tensorflow**, **Keras** and other supporting tools in R ?
- What are **tensors** and how they **flow** ?
- Different types of models in Keras ?
- What is **MLP** ?
- How to **build**, **compile**, **fit** and **evaluate** model in Keras ?
- What is **dropout** and how to use it ?
- How to use Keras **callbacks** for model checkpoint and early stopping
- How to tune hyperparameters

What is Tensorflow ?

TensorFlow:

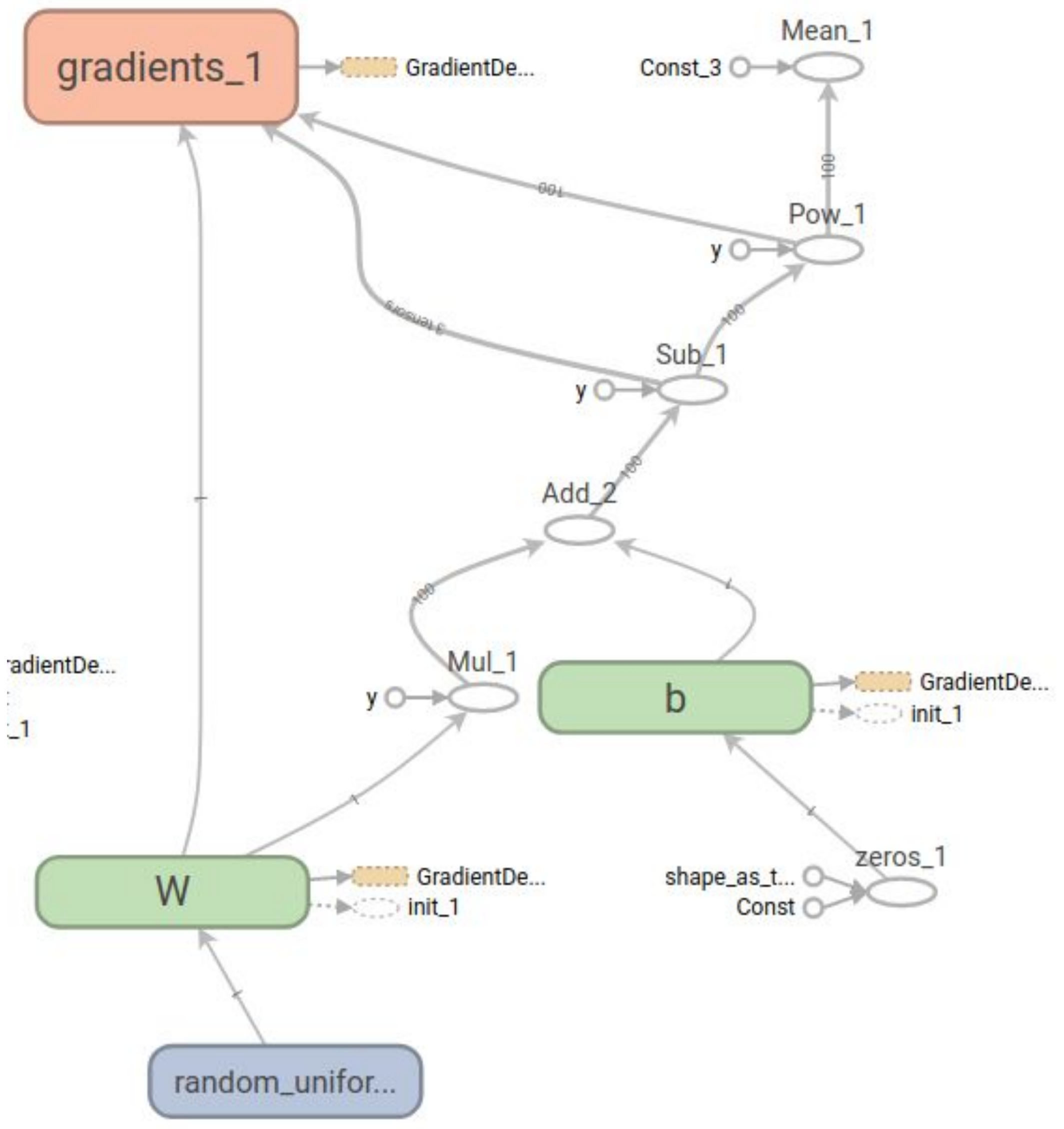
- is **general purpose** numerical computing library (not only Deep Learning!).
- is an **open-source** software.
- allows you to deploy computation to **multiple CPUs, GPUs and TPUs**.
- was developed by researchers and engineers working on the **Google Brain Team.**

What “flows” ?

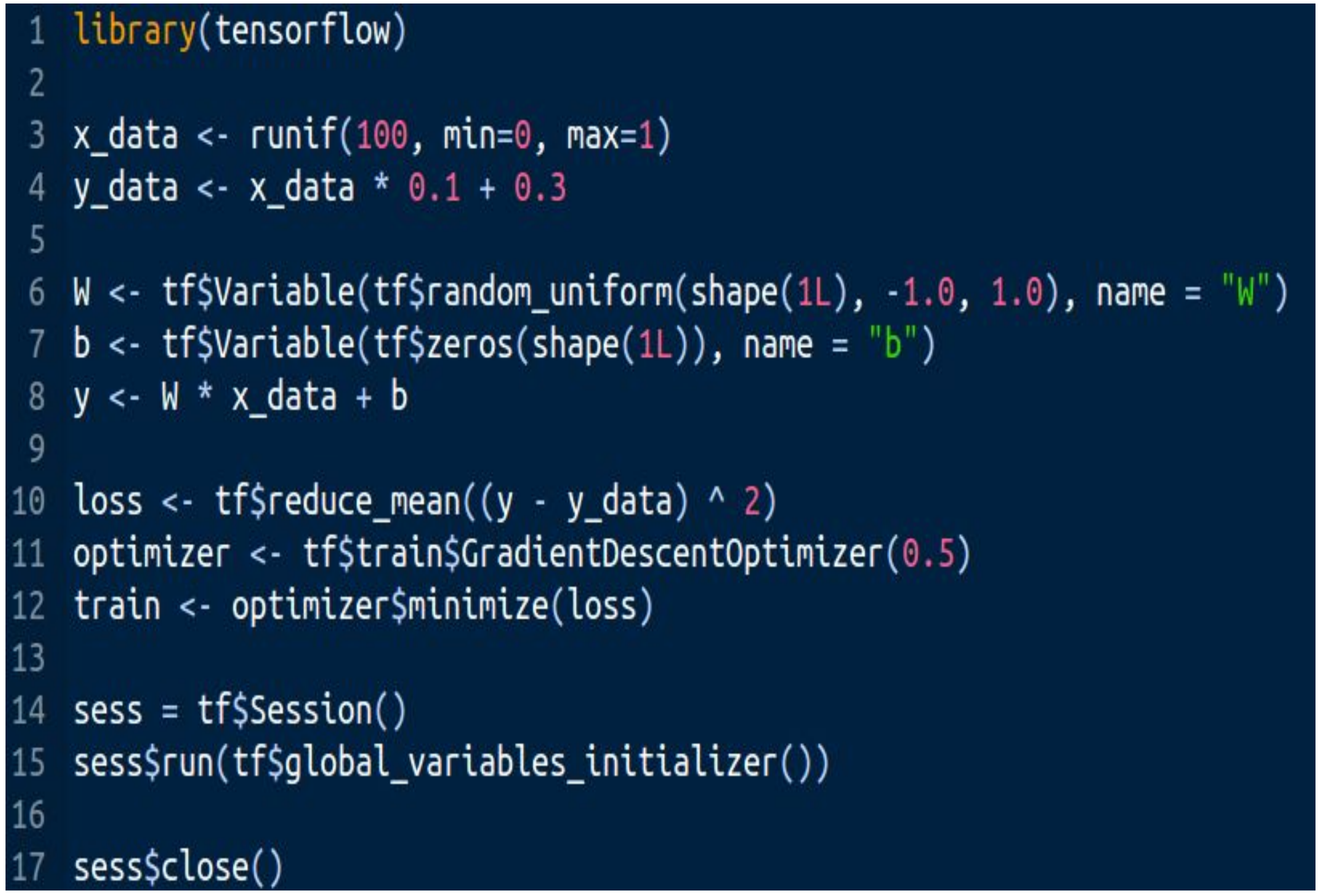
You can think of a **tensor** as of a multidimensional array:

Data	Tensor
Vector data	2D tensor: (samples, features)
Timeseries data	3D tensor: (samples, timestep, features)
Image	4D tensor: (samples, height, width, channels)
Video	5D tensor: (samples, frames, height, width, channels)

The “flow”

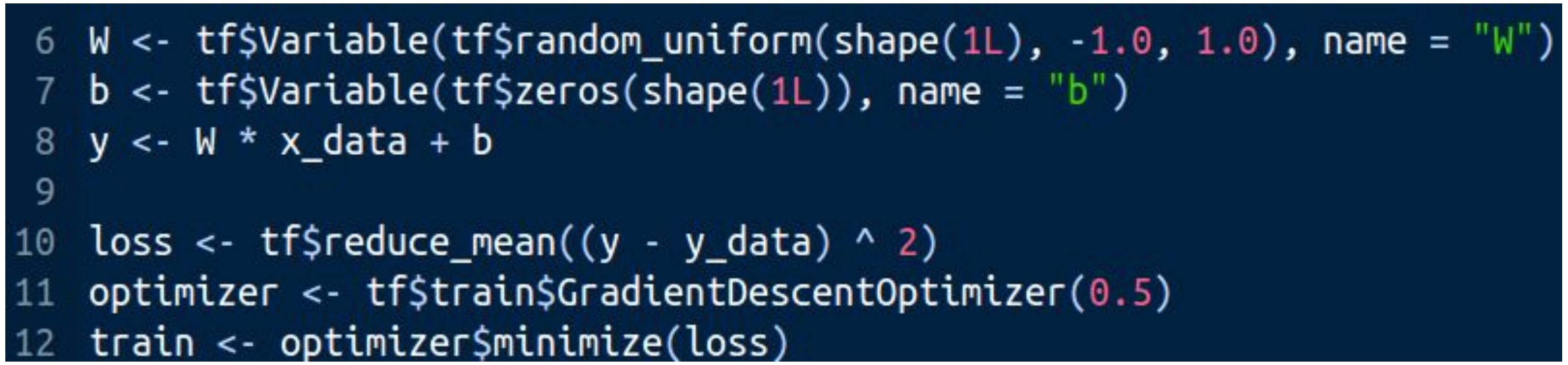


A TensorFlow **graph** is a description of **computations**. Tensors **flow** between **nodes**. Node takes zero or more Tensors, performs some computation, and produces zero or more Tensors.


```r
library(tensorflow)

x_data <- runif(100, min=0, max=1)
y_data <- x_data * 0.1 + 0.3

W <- tf$Variable(tf$random_uniform(shape(1L), -1.0, 1.0), name = "W")
b <- tf$Variable(tf$zeros(shape(1L)), name = "b")
y <- W * x_data + b

loss <- tf$reduce_mean((y - y_data) ^ 2)
optimizer <- tf$train$GradientDescentOptimizer(0.5)
train <- optimizer$minimize(loss)

sess = tf$Session()
sess$run(tf$global_variables_initializer())

sess$close()
```

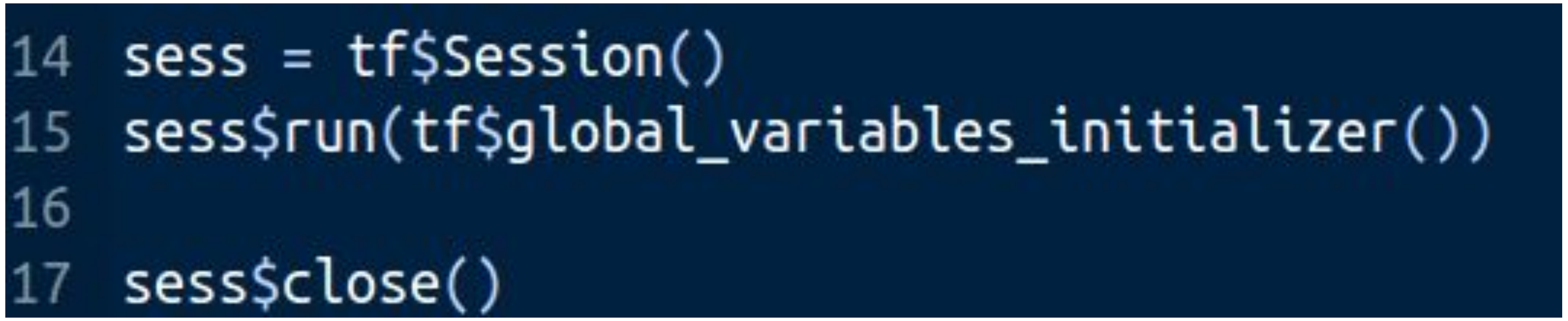
What is happening ?

1. Create a graph - create constants, variables, operations between them (tensors and nodes)



```r
W <- tf$Variable(tf$random_uniform(shape(1L), -1.0, 1.0), name = "W")
b <- tf$Variable(tf$zeros(shape(1L)), name = "b")
y <- W * x_data + b

loss <- tf$reduce_mean((y - y_data) ^ 2)
optimizer <- tf$train$GradientDescentOptimizer(0.5)
train <- optimizer$minimize(loss)
```

2. Run the graph in a session



```r
sess = tf$Session()
sess$run(tf$global_variables_initializer())

sess$close()
```

What is Keras ?

Keras:

Keras is a high-level neural networks API capable of running on top of multiple back-ends including: **TensorFlow**, **CNTK**, or **Theano**. One of its biggest advantages is its "user friendliness". With Keras you can easily build advanced models like **convolutional** or **recurrent neural networks**.

Keras and Tensorflow in R

Keras API

The Keras API for TensorFlow provides a high-level interface for neural networks, with a focus on enabling fast experimentation.

Estimator API

The Estimator API for TensorFlow provides high-level implementations of common model types such as regressors and classifiers.

Core API

The Core TensorFlow API is a lower-level interface that provides full access to the TensorFlow computational graph.

Support tools

- tfruns—Track, visualize, and manage TensorFlow training runs and experiments. Tune hyperparameters.
- tfdeploy—Tools designed to make exporting and serving TensorFlow models straightforward.
- cloudml—R interface to Google Cloud Machine Learning Engine.

Building models in Keras

In Keras you can build models in 3 different ways:

1. Using a **sequential model** (layer by layer):
   - MLP, ConvNet, RNN

2. Using **functional API** (multiple inputs/outputs, shared layers, systems of networks):
   - Object detection (f.e. Faster R-CNN)
   - Generative adversarial networks (GANs)

3. Using **pre-trained models** (applications)

Building models in Keras

Steps to build model in Keras:

1. **Define model architecture** - choose model type, add layers, define inputs and outputs, add regularization,...

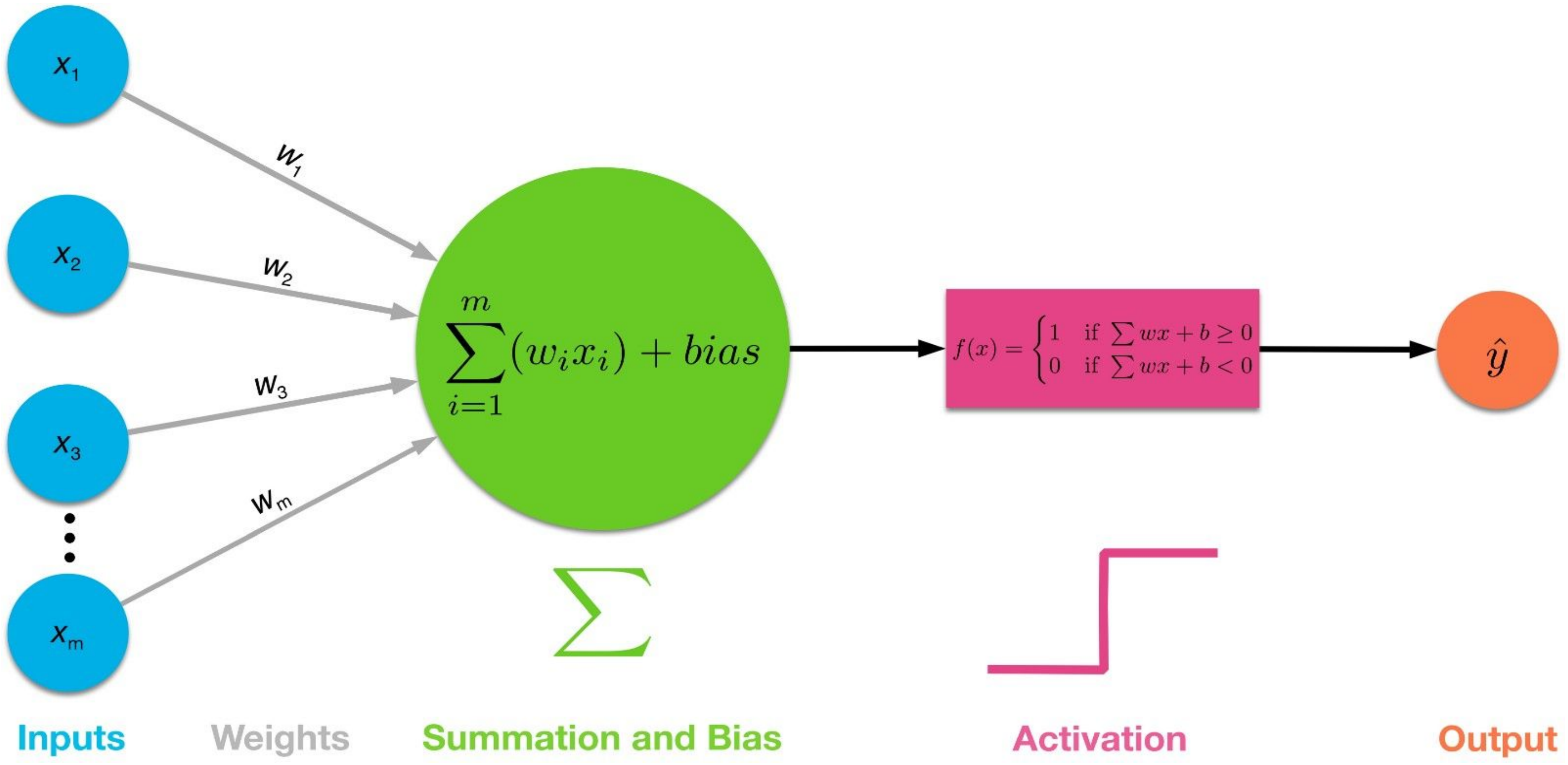
2. **Compile** - choose loss function, optimizer and metrics

3. **Fit** - define train/validation sets, callbacks

4. **Evaluate / Predict** - evaluate on test set

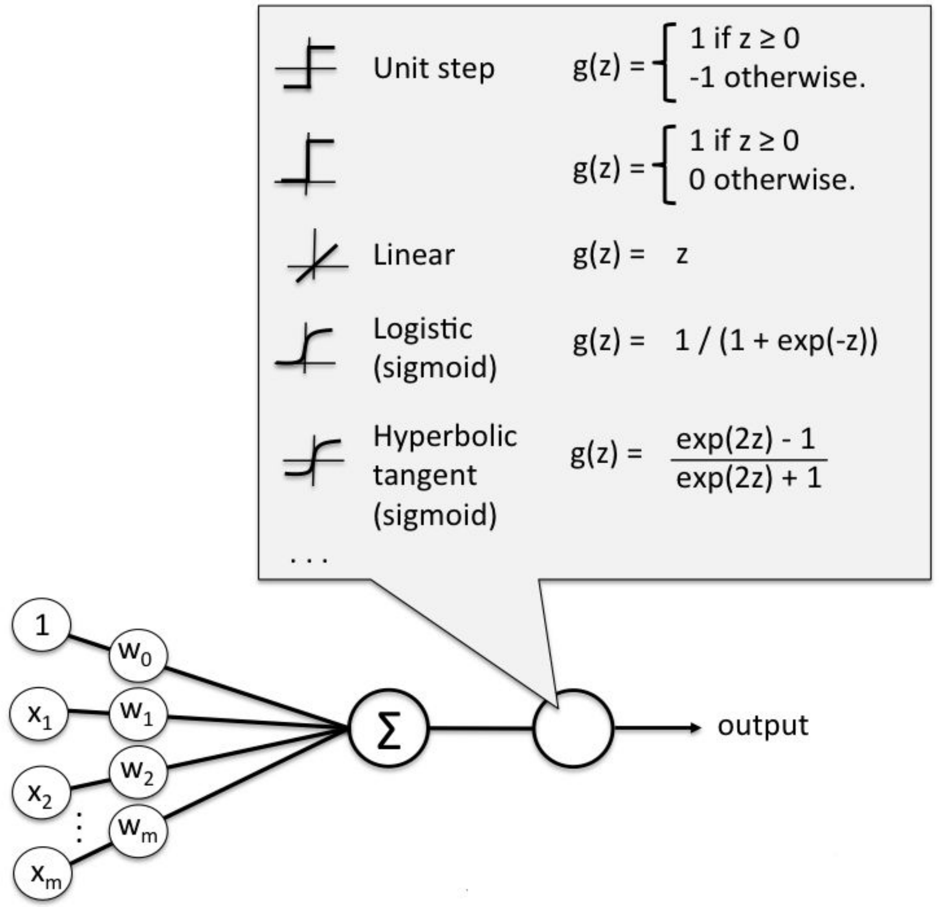
MLP

In a single **neuron**, output is a **linear combination** of **inputs** (and sometimes **bias**) transformed using (usually) nonlinear **activation function**.


x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

MLP

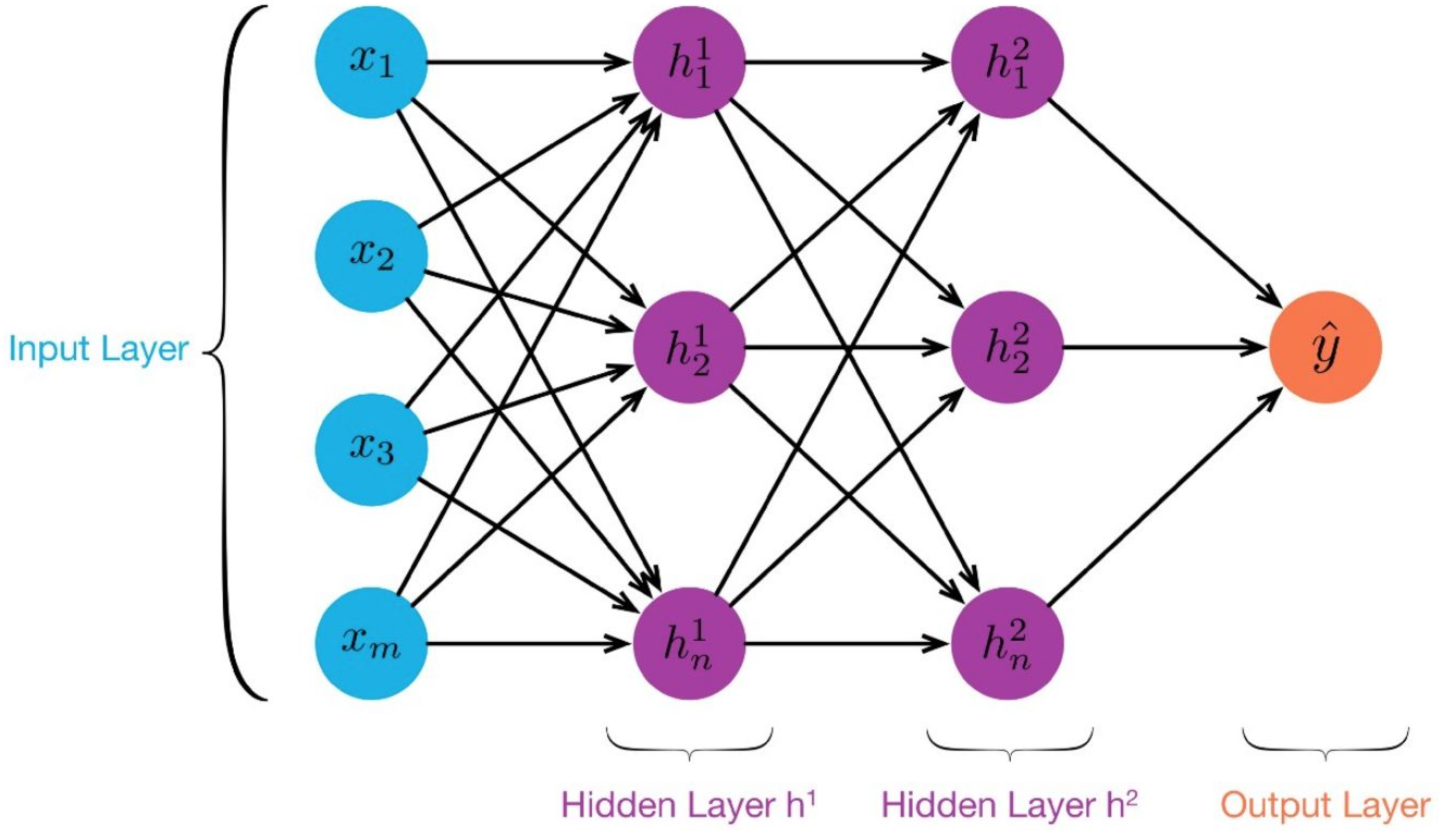
Choosing correct activation function could be crucial.



Unit step
$g(z) = \begin{cases} 1 \text{ if } z \geq 0 \\ -1 \text{ otherwise.} \end{cases}$
$g(z) = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ otherwise.} \end{cases}$
Linear
$g(z) = z$
Logistic (sigmoid)
$g(z) = 1 / (1 + \exp(-z))$
Hyperbolic tangent (sigmoid)
$g(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1}$
. . .
1
x_1
x_2
x_m
w_0
w_1
w_2
w_m
Σ
output

MLP

An MLP consists of at least three **layers** of neurons (nodes, units). We can think of a neurons in **hidden layers** as a “new variables” created from initial inputs.



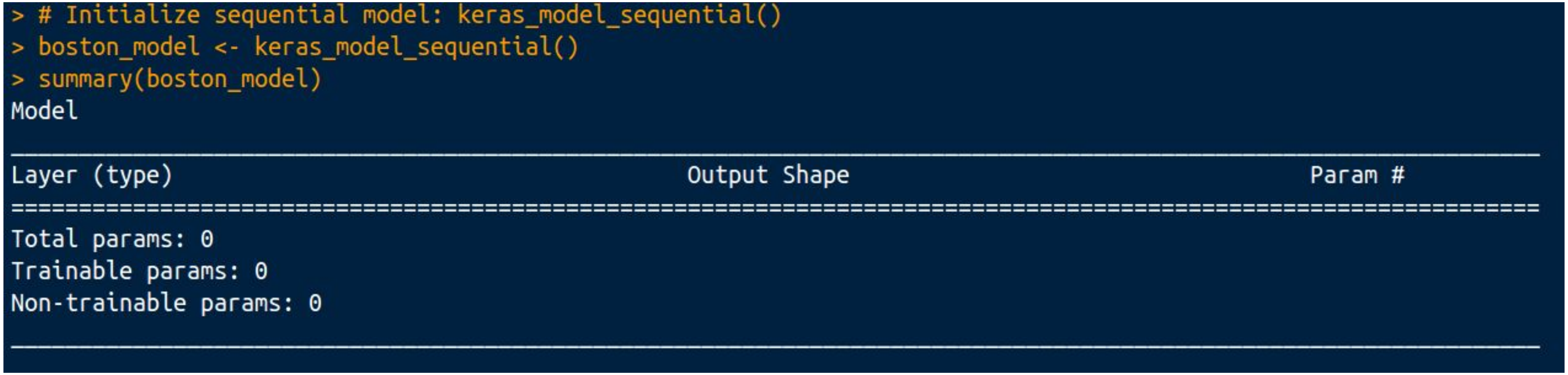
Example

Regression example using **Boston housing** dataset.

13 **inputs** and one output (median value of owner-occupied homes in $1000s.)

Define model architecture

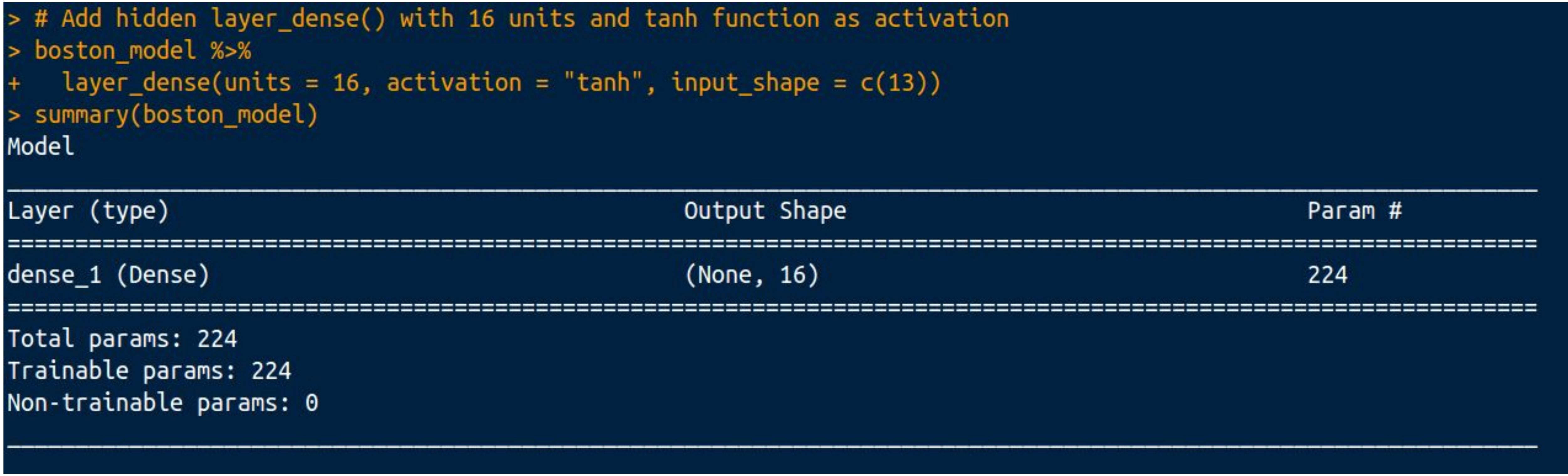
To initialize a sequential model use **keras_model_sequential()** function.



```
> # Initialize sequential model: keras_model_sequential()
> boston_model <- keras_model_sequential()
> summary(boston_model)
Model
________________________________________________________________________________
Layer (type)                        Output Shape                      Param #
================================================================================
Total params: 0
Trainable params: 0
Non-trainable params: 0
________________________________________________________________________________
```

Define model architecture

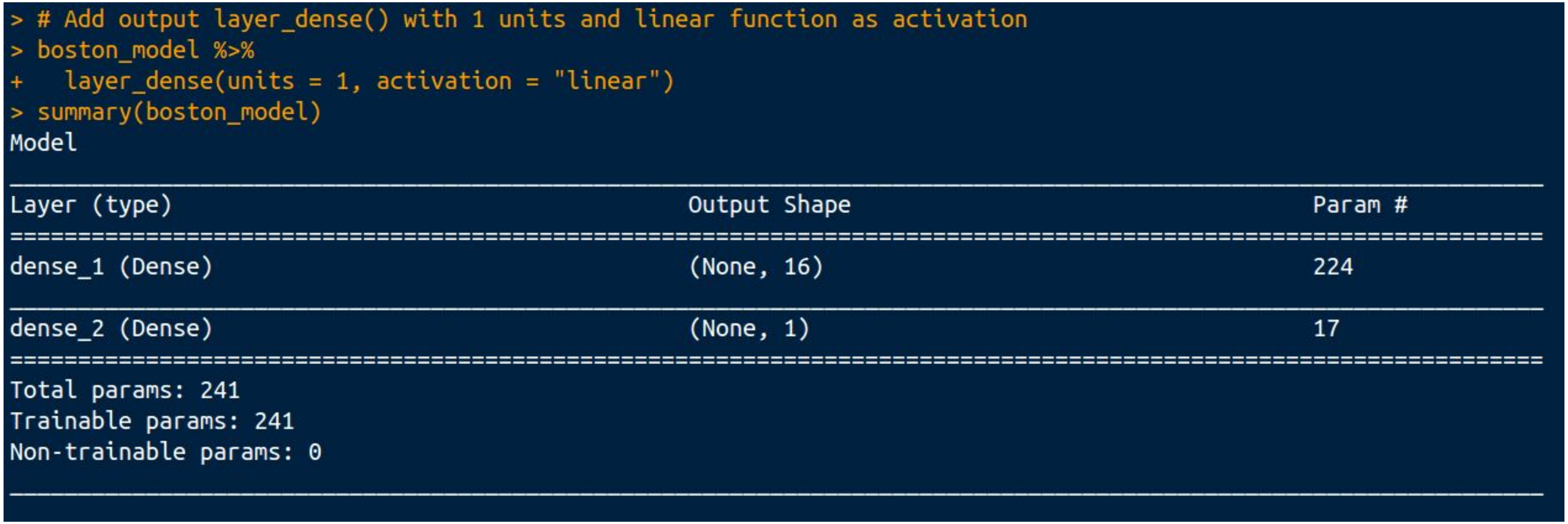
We can add new layers using **%>%** operator. Note that keras models are changed **in-place** and there’s no need for second assignment. **layer_dense()** adds fully connected layer (here with 16 **neurons** and **tanh** as activation). First layer should always specify **input shape**.



```
> # Add hidden layer_dense() with 16 units and tanh function as activation
> boston_model %>%
+   layer_dense(units = 16, activation = "tanh", input_shape = c(13))
> summary(boston_model)
Model

________________________________________________________________________________
Layer (type)                        Output Shape                    Param #
================================================================================
dense_1 (Dense)                     (None, 16)                      224
================================================================================
Total params: 224
Trainable params: 224
Non-trainable params: 0

________________________________________________________________________________
```

Define model architecture

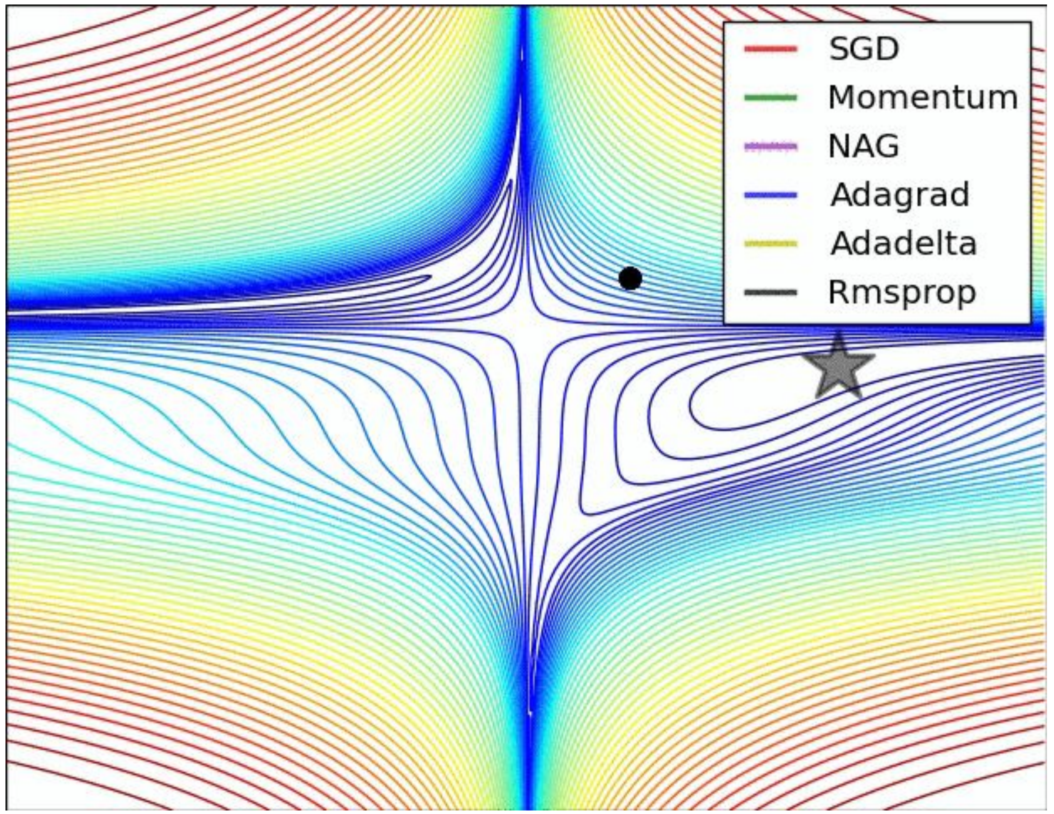
In similar way we will add output layer. We will use only one neuron with linear activation - regression task.

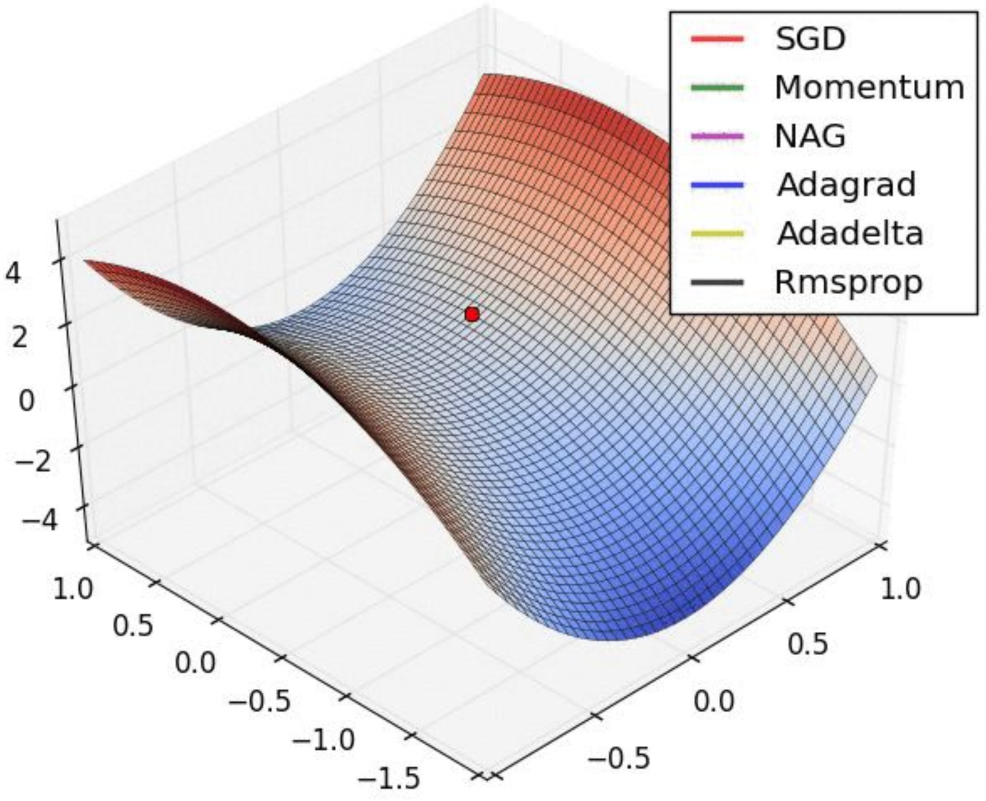


```
> # Add output layer_dense() with 1 units and linear function as activation
> boston_model %>%
+   layer_dense(units = 1, activation = "linear")
> summary(boston_model)
Model
________________________________________________________________________________
Layer (type)                        Output Shape                       Param #
================================================================================
dense_1 (Dense)                     (None, 16)                         224
________________________________________________________________________________
dense_2 (Dense)                     (None, 1)                          17
================================================================================
Total params: 241
Trainable params: 241
Non-trainable params: 0
________________________________________________________________________________
```

Loss function and optimization

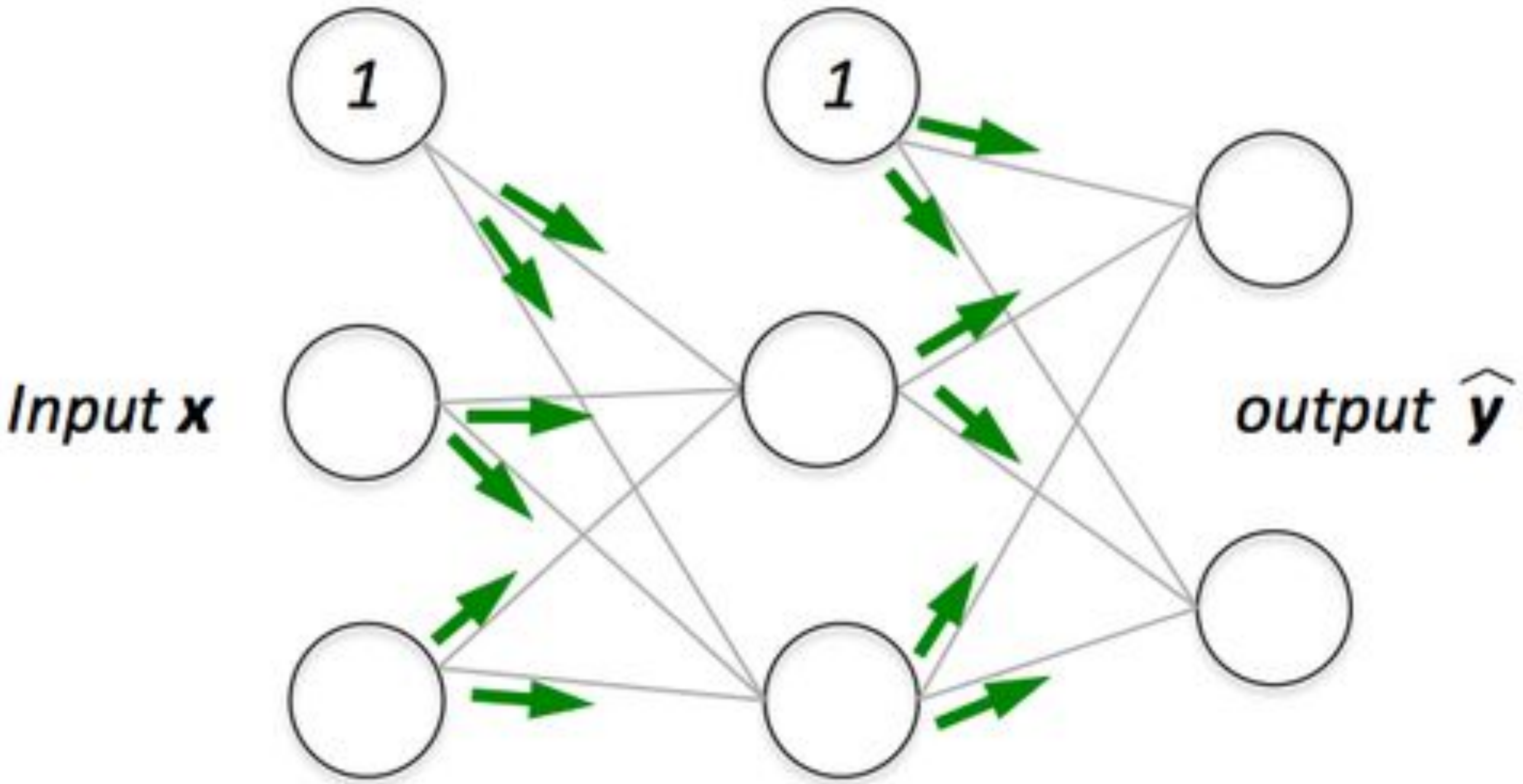
Neural network weights are updated in a process called **backpropagation**.
Choosing good optimization algorithm is crucial.



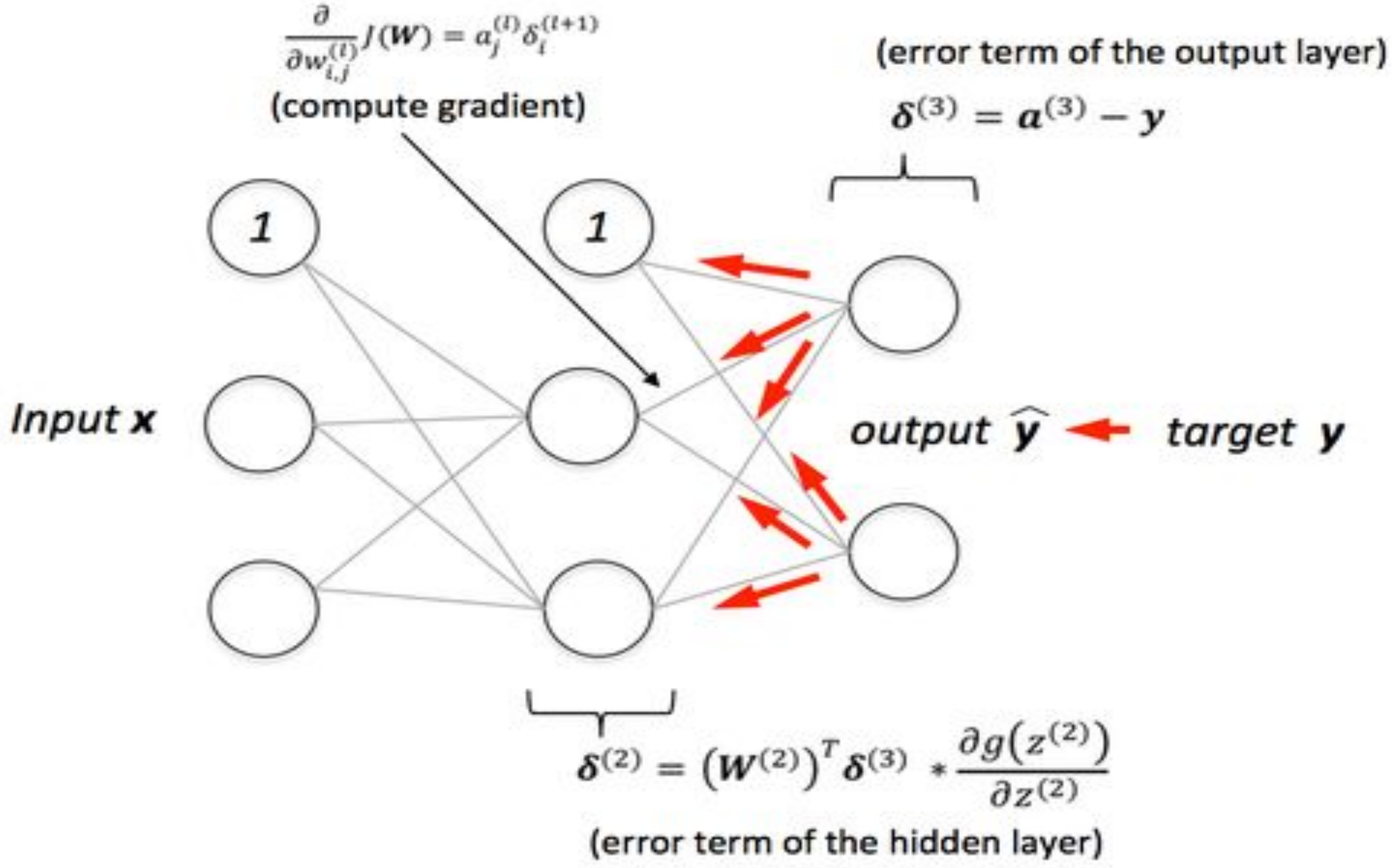


Source: http://ruder.io/optimizing-gradient-descent/

Forward pass



Backward pass


$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

Gradient descent

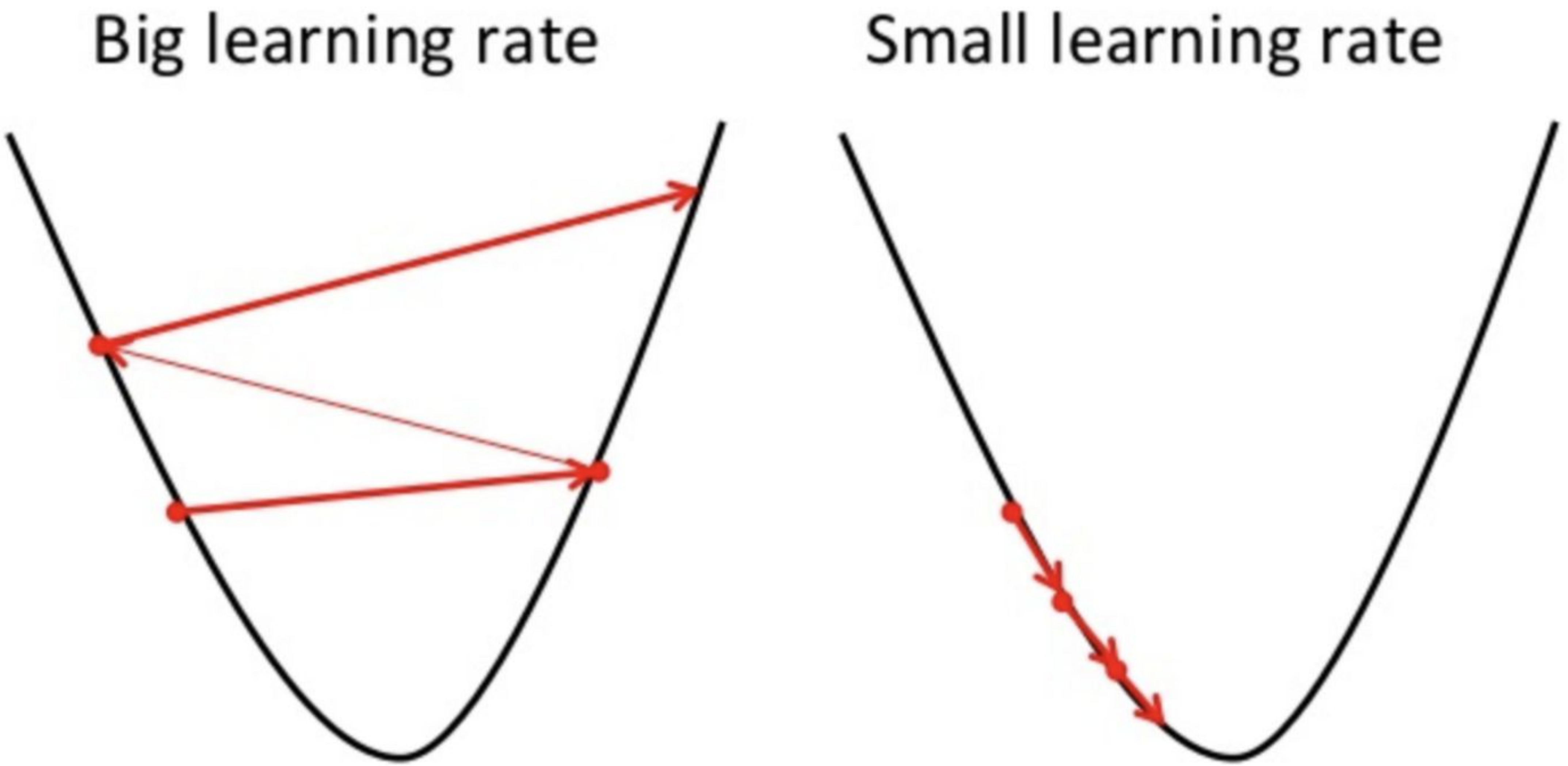
$$f(x_i) = f_{W,b}(x_i) = b + \sum_{j=1}^{p} W_j x_{ij} \tag{1}$$

$$L(W,b) = \frac{1}{n} \sum_{i=1}^{n} (f(x_i) - y_i)^2 \tag{2}$$

$$\frac{\partial L}{\partial W} = \frac{2}{n} \sum_{i=1}^{n} (f(x_i) - y_i) x_i \qquad \frac{\partial L}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (f(x_i) - y_i) \tag{3}$$

$$\begin{aligned} W &\leftarrow W - \alpha \frac{\partial L}{\partial W} \\ b &\leftarrow b - \alpha \frac{\partial L}{\partial b} \end{aligned} \tag{4}$$

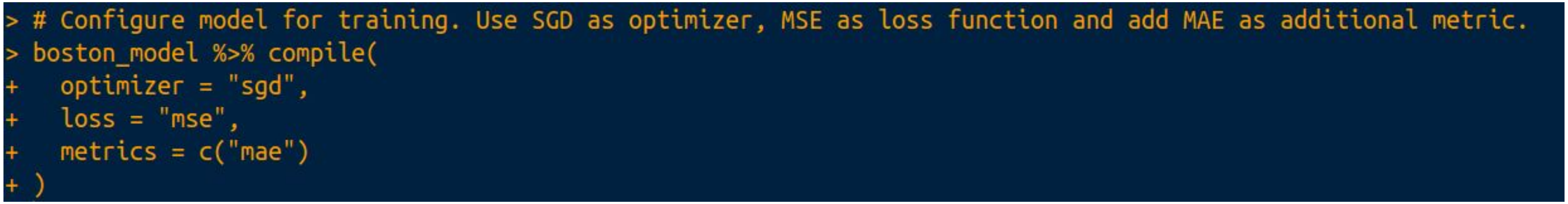
Gradient descent



Compile model

After defining architecture of our model we have to configure it for training, we can do this with **compile()** function. We have to define:

- **loss** - loss function to minimize
- **optimizer** - algorithm that will minimize loss
- **metrics** (optional) - additional metrics to print



```
> # Configure model for training. Use SGD as optimizer, MSE as loss function and add MAE as additional metric.
> boston_model %>% compile(
+   optimizer = "sgd",
+   loss = "mse",
+   metrics = c("mae")
+ )
```

Loss functions in keras

Some important loss functions:

Loss	Task
“mse”, “mae”	regression
“binary_crossentropy”	binary classification
“categorical_crossentropy”	multiclass classification
“hinge”	classification

You can always create your own loss function.

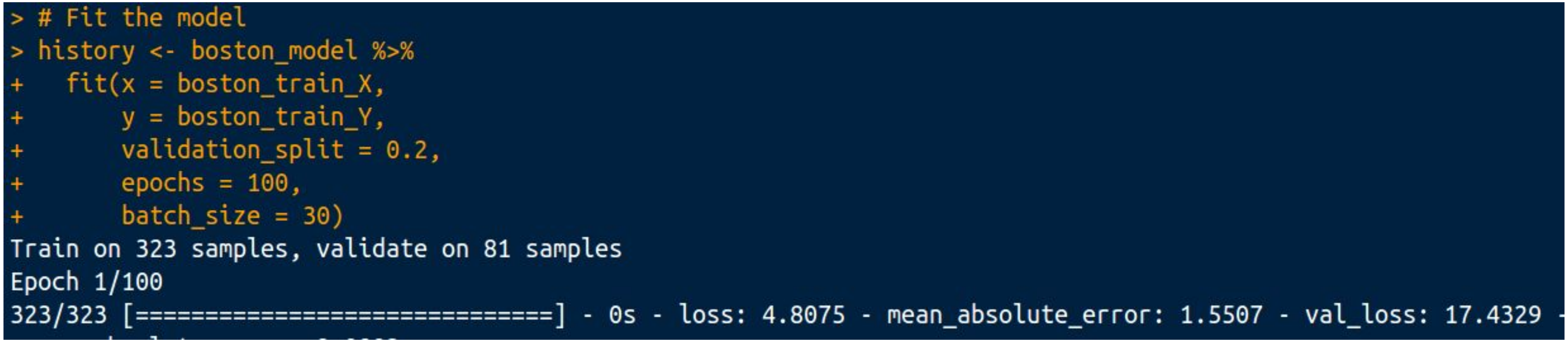
Optimizers in Keras

Some important optimizers:

Loss	Function
“sgd”	optimizer_sgd()
“rmsprop”	optimizer_rmsprop()
“adam”	optimizer_adam()
“adadelta”	optimizer_adadelta()

Fitting the model in Keras

To fit the model we can use **fit()** function. Computing derivatives on a whole dataset can take time. We can divide the dataset into **batches** and update weights looking only at one batch. **Batch size** defines number of samples that going to be propagated through the network. Number of **epochs** defines how many times we should propagate through all dataset. F.e. if I’ve got 9000 samples in train set and I want to create 3 batches, each batch will have 3000 samples. 1 epoch = 3 batches.



```
> # Fit the model
> history <- boston_model %>%
+   fit(x = boston_train_X,
+       y = boston_train_Y,
+       validation_split = 0.2,
+       epochs = 100,
+       batch_size = 30)
Train on 323 samples, validate on 81 samples
Epoch 1/100
323/323 [==============================] - 0s - loss: 4.8075 - mean_absolute_error: 1.5507 - val_loss: 17.4329 -
```

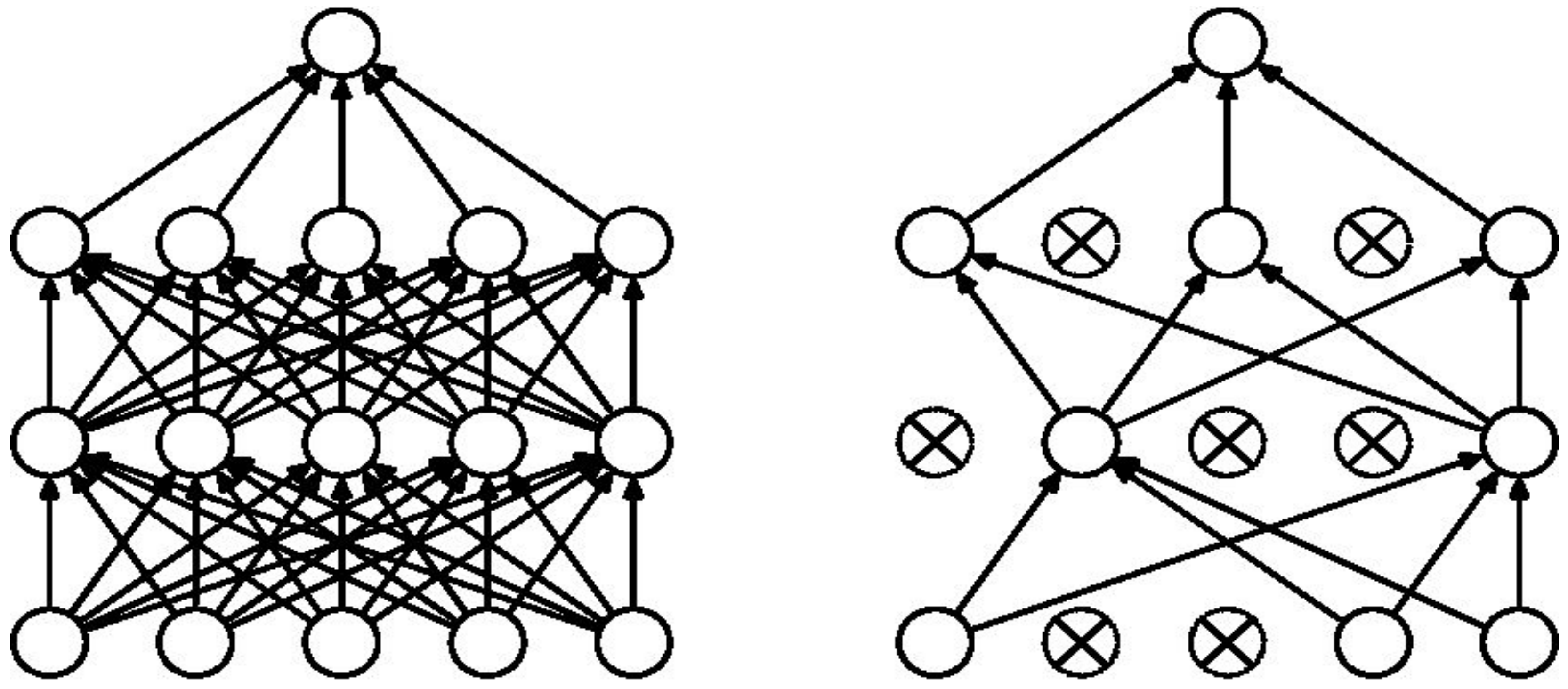
Ex. 1 - Fashion MNIST

Each example is a 28x28 grayscale (values from 0 to 255) image (**784 variables**), associated with a label from 10 classes:

T-shirt/top, Trouser ,Pullover
Dress, Coat, Sandal, Shirt
Sneaker, Bag, Ankle boot

Dropout

At each training stage, individual nodes are either dropped out of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left.

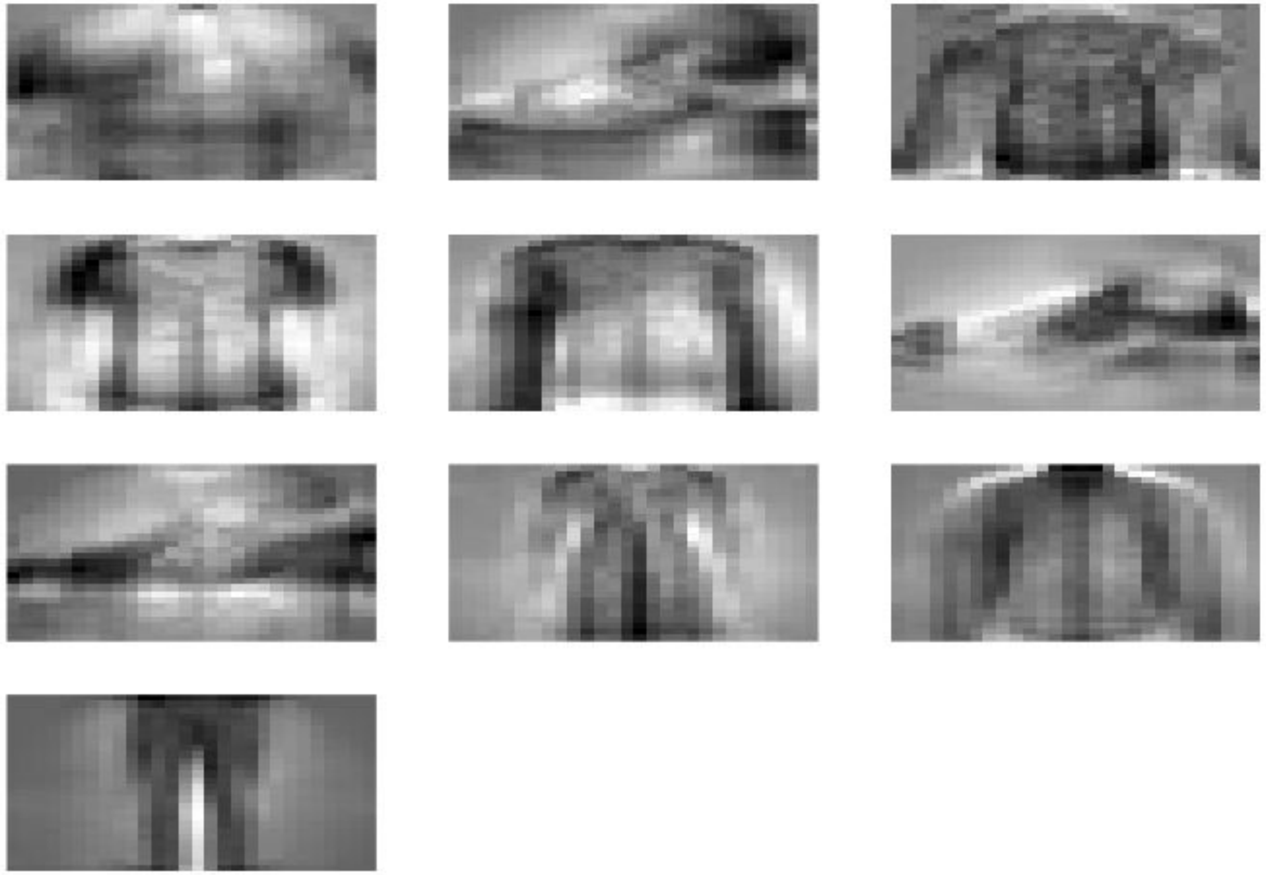


(a) Standard Neural Net (b) After applying dropout.

Image classification using MLP

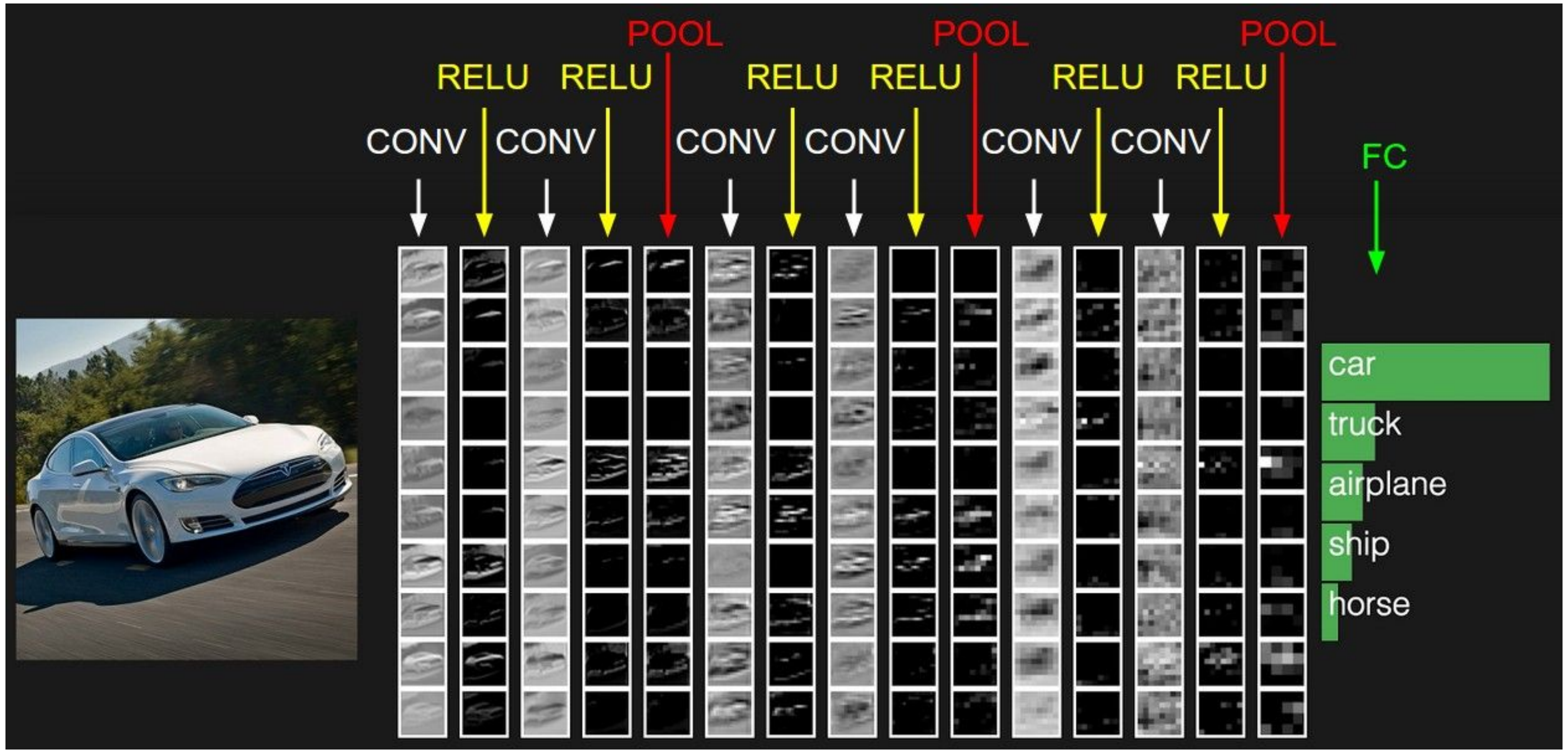
In our example, "new variables" (neurons in first hidden layer) are dependent on every pixel in an image (every pixel will be associated with a weight), so we're looking at **global patterns** in our data.

Can we create "new variables" in a better way? How can we search for **local patterns**?



What is ConvNet ?

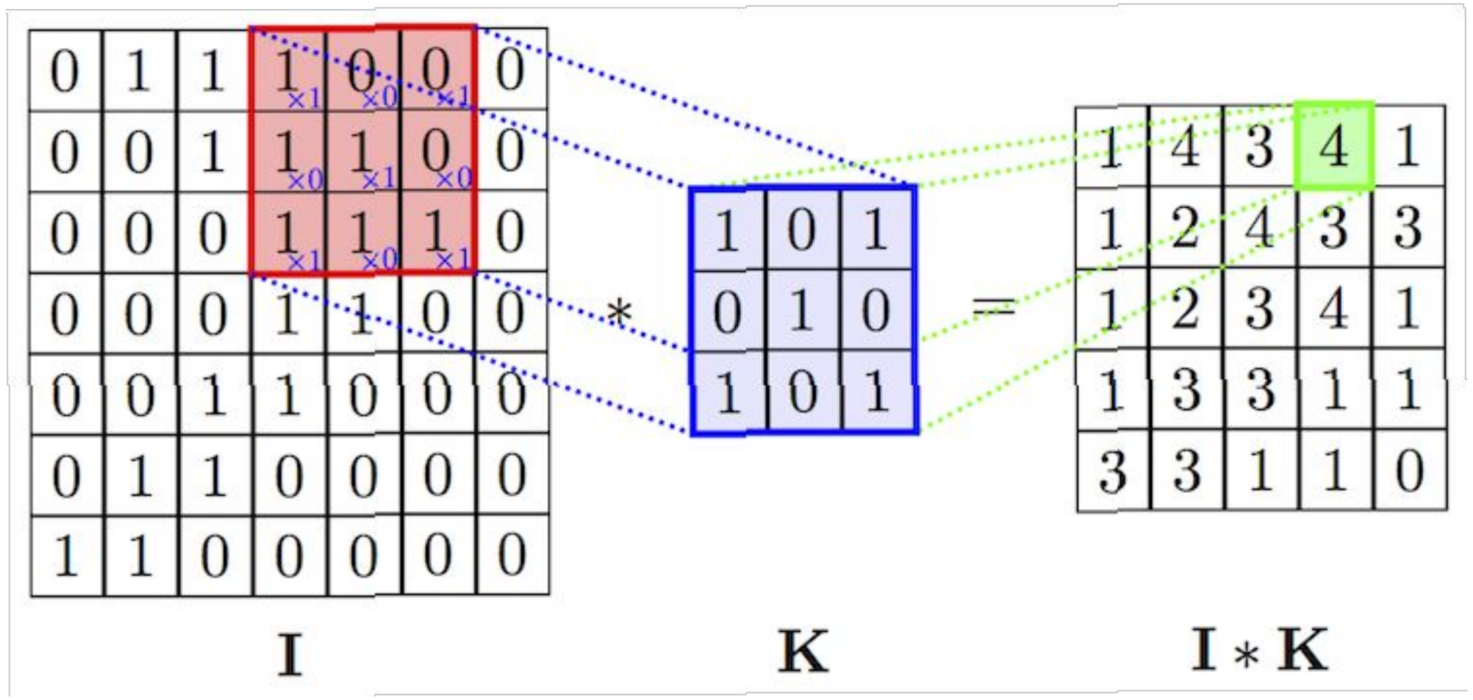
Convolutional neural networks (CNN) or **ConvNets** for short are a class of deep, feed-forward artificial neural networks designed for solving problems like image/video/audio recognition, object detection etc. Architecture of ConvNets differs depending on the issue, but there are some basic common parts.



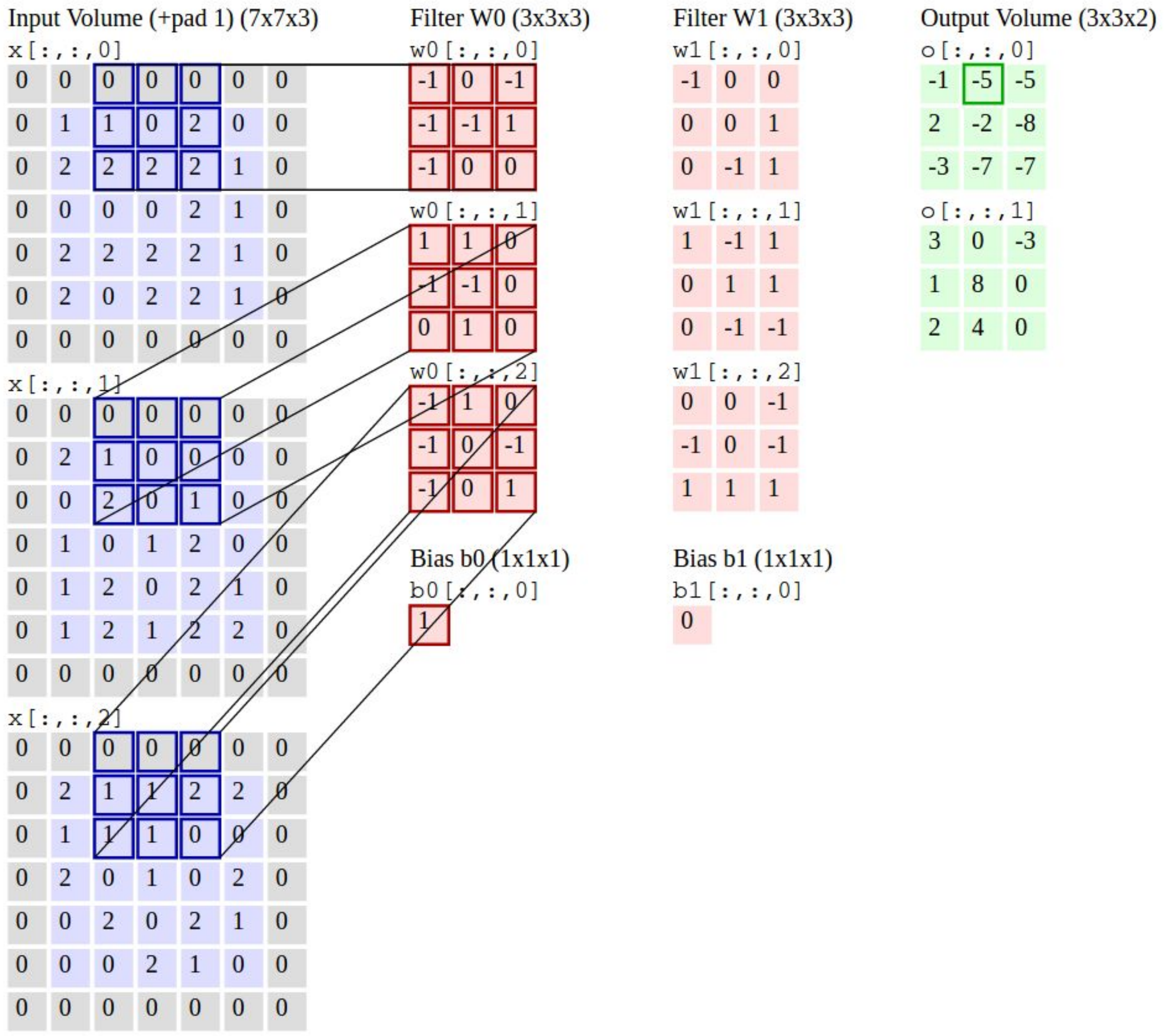
Convolutional layer

The first type of layers in CNN is a **convolutional layer** and it is a core building block of a ConvNets. Simply speaking in convolutional layers we take a set of small **filters** (also called **kernels**) and we're placing them on a part of our original image to get the dot product between kernel and corresponding image part. Next we are moving our filter to the next place and we're repeating this action. Numbers of pixels to move filter are called **strides**. After getting dot products for whole image we get so called **activation map**.

$$O = \frac{(W - K + 2P)}{S} + 1$$



Convolutional layer



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	1	1	0	2	0	0
0	2	2	2	2	1	0
0	0	0	0	2	1	0
0	2	2	2	2	1	0
0	2	0	2	2	1	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	2	1	0	0	0	0
0	0	2	0	1	0	0
0	1	0	1	2	0	0
0	1	2	0	2	1	0
0	1	2	1	2	2	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	1	1	2	2	0
0	1	1	1	0	0	0
0	2	0	1	0	2	0
0	0	2	0	2	1	0
0	0	0	2	1	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	-1
-1	-1	1
-1	0	0

w0[:,:,1]

1	1	0
-1	-1	0
0	1	0

w0[:,:,2]

-1	1	0
-1	0	-1
-1	0	1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

-1	0	0
0	0	1
0	-1	1

w1[:,:,1]

1	-1	1
0	1	1
0	-1	-1

w1[:,:,2]

0	0	-1
-1	0	-1
1	1	1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

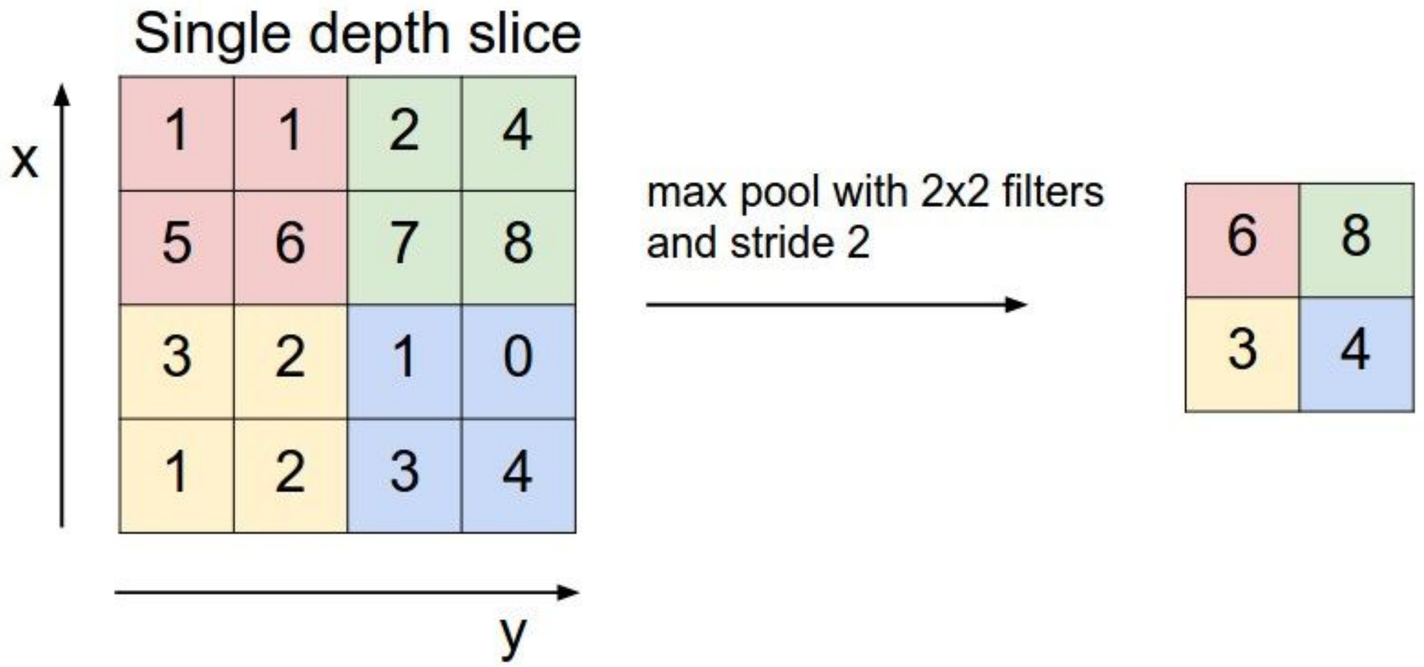
-1	-5	-5
2	-2	-8
-3	-7	-7

o[:,:,1]

3	0	-3
1	8	0
2	4	0

Pooling layer

The second type of layers in CNN is **pooling layer**. This layer is responsible for dimensionality reduction of activation maps. There are several types of pooling, but **max pooling** is the most common one. As it was in the case of convolutional layers we have some filter (**pool**) and **strides**. After placing the filter on an image part we are taking maximum value from that part and then moving to next place by number of pixels specified by strides.

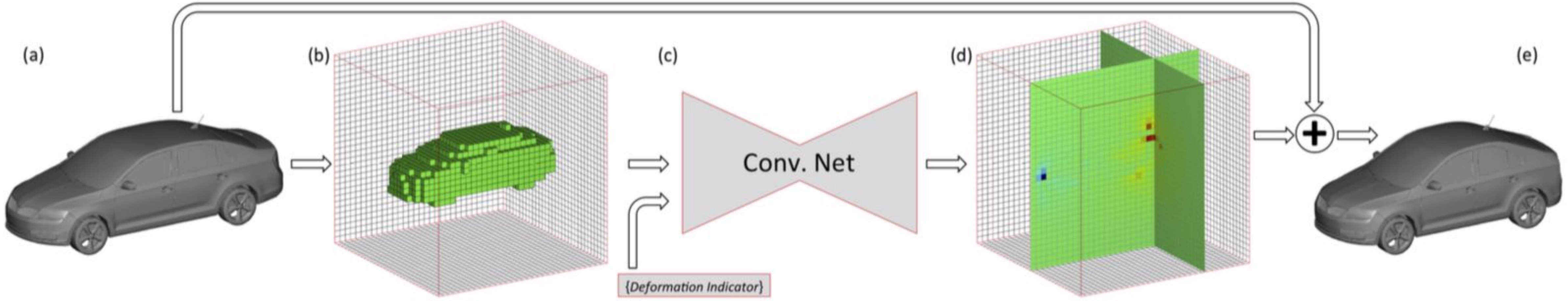


Ex - Fashion MNIST

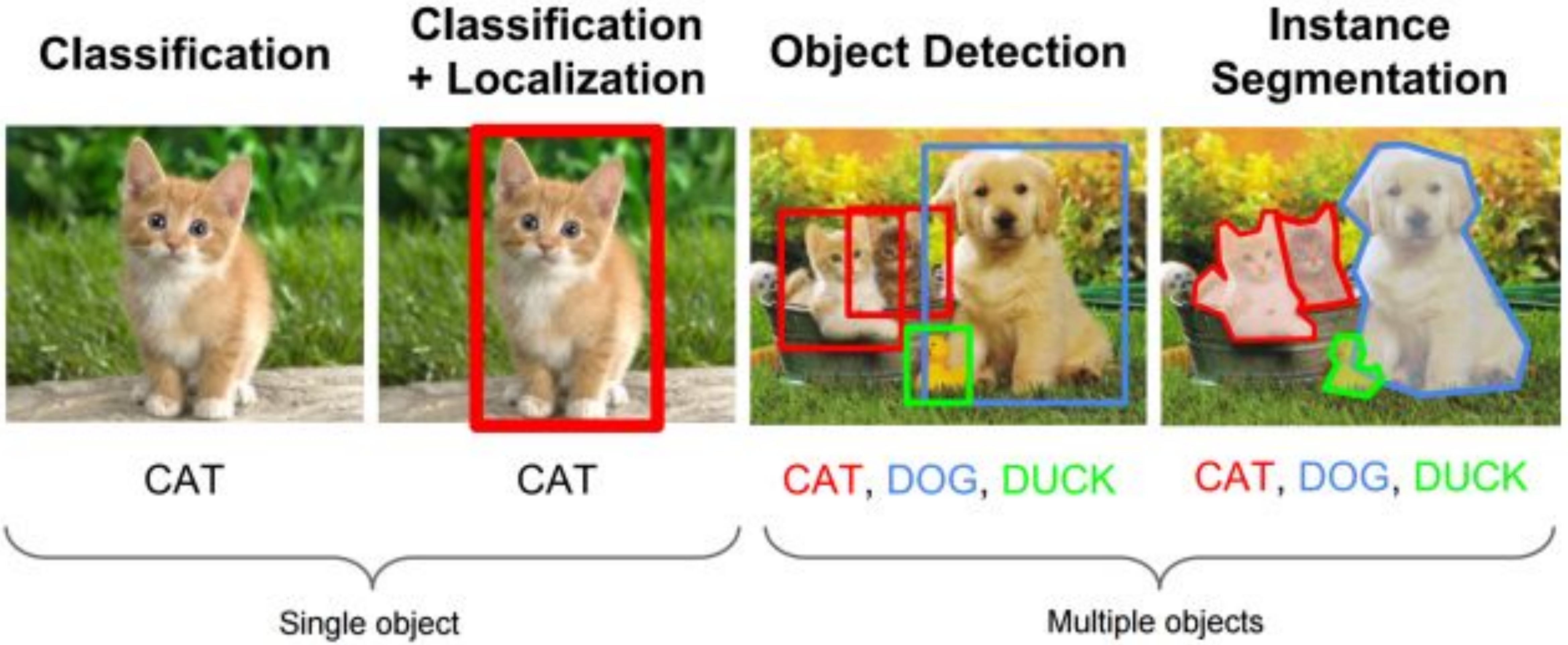
Each example is a 28x28 grayscale (values from 0 to 255) image (**784 variables**), associated with a label from 10 classes:

T-shirt/top, Trouser ,Pullover
Dress, Coat, Sandal, Shirt
Sneaker, Bag, Ankle boot

1D and 3D convolution



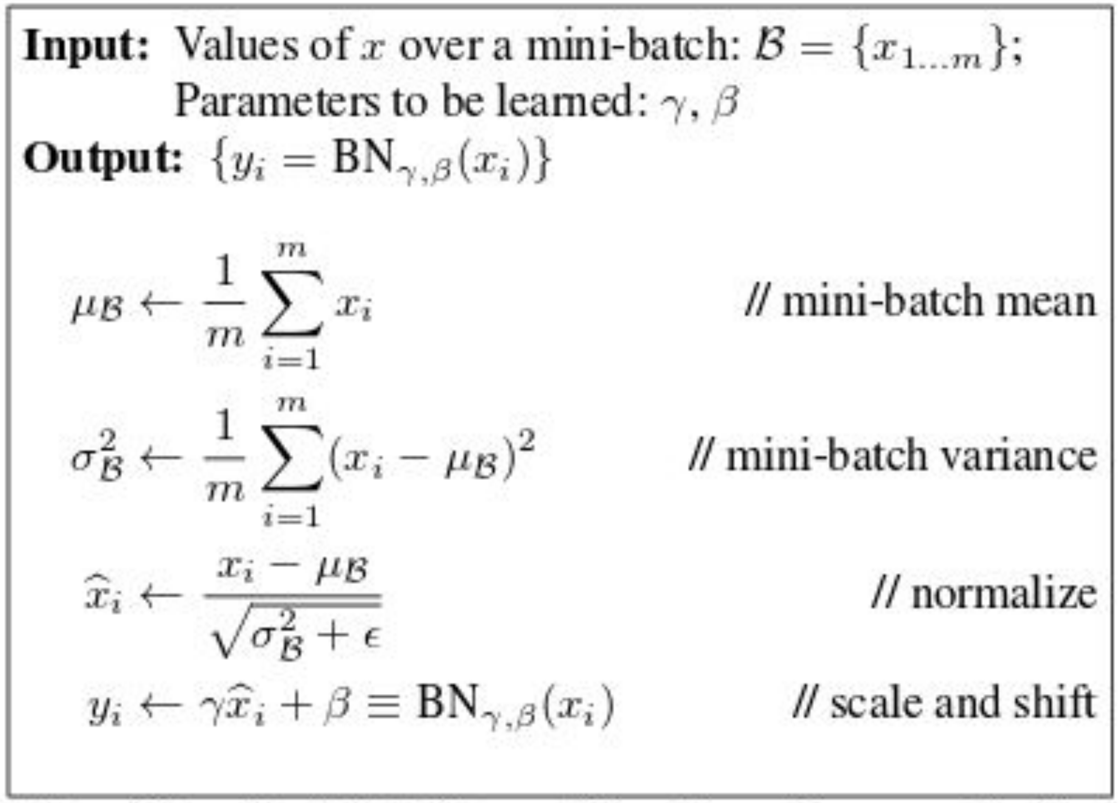
What’s next ?


Classification
Classification + Localization
Object Detection
Instance Segmentation
CAT
CAT
CAT, DOG, DUCK
CAT, DOG, DUCK
Single object
Multiple objects

Batch normalization and callbacks

Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift).

Batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a "**standard deviation**" parameter (gamma) and add a "**mean**" parameter (beta)



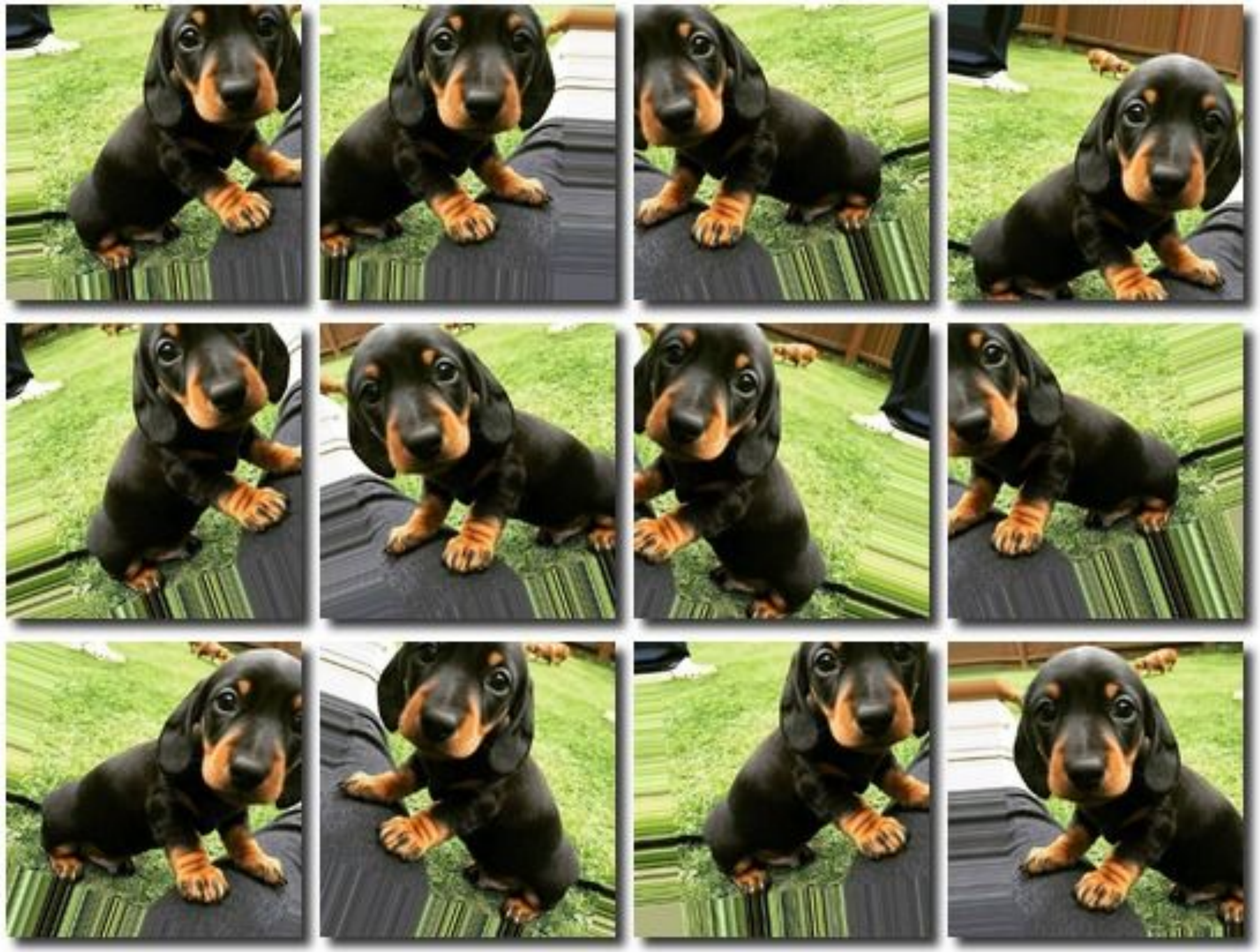
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                  // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Generators in Keras

Small Datasets	Larger Datasets
fit()	fit_generator()
predict()	predict_generator()

Data augmentation



Fine-tuning

Fine-tuning is a process to take a network model that has already been trained for a given task, and make it perform a second similar task.

Steps:

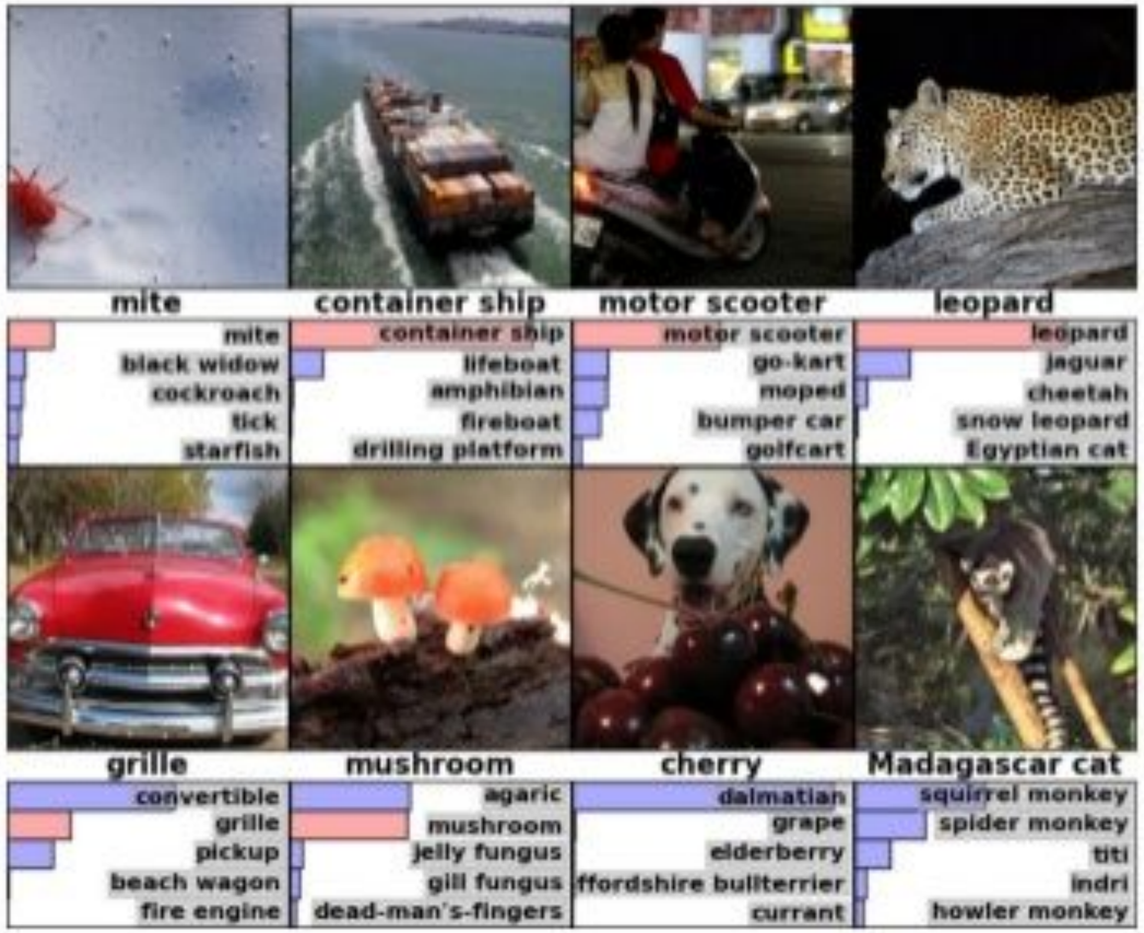
1. Get pre-trained network using **application_NET**() function.
2. Remove dense layers (we want only **convolutional base**).
3. Freeze few last layers in convolutional base.
4. Add randomly initialized dense layers.
5. Train the model on new data.

Imagenet

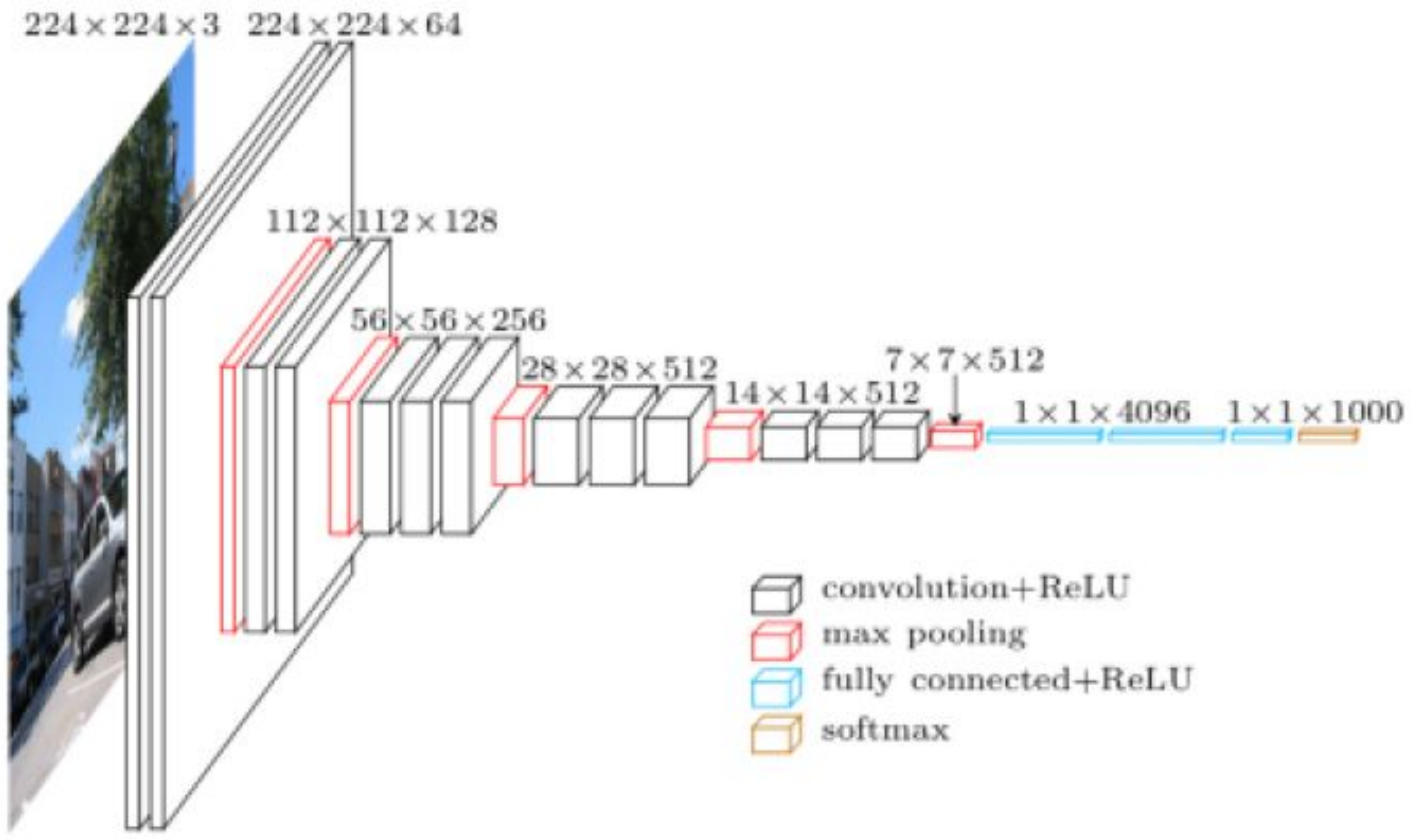
ImageNet Challenge

IMAGENET

- 1,000 object classes (categories).
- Images:
  - 1.2 M train
  - 100k test.

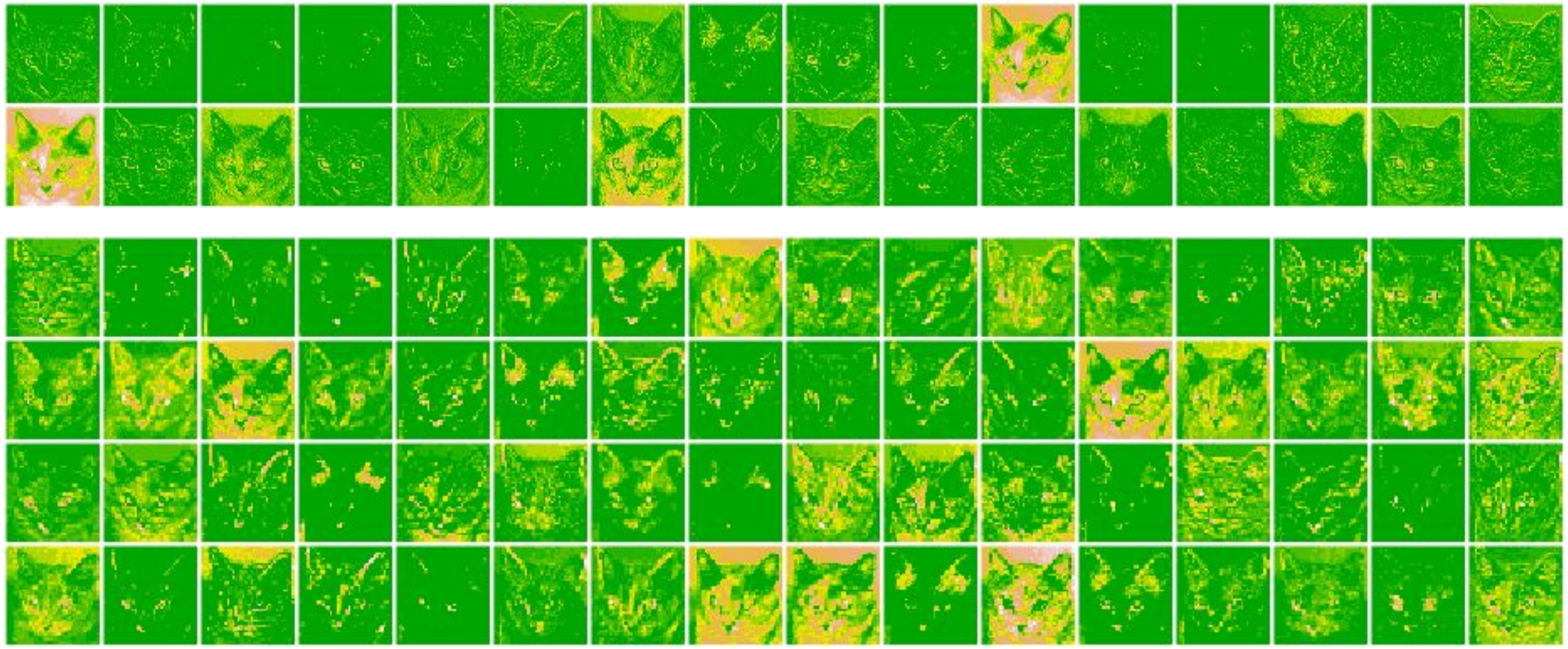


Fine-tuning VGG16



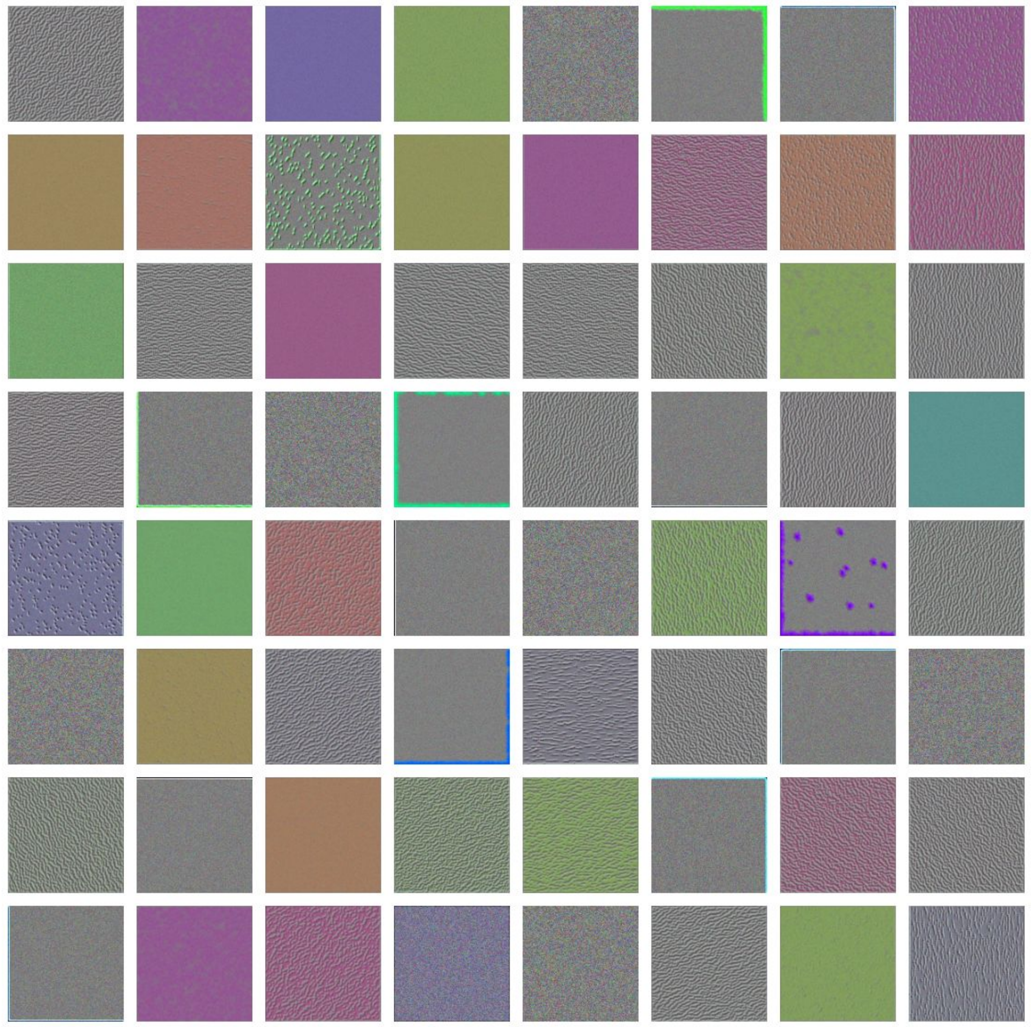
What ConvNet learns ?

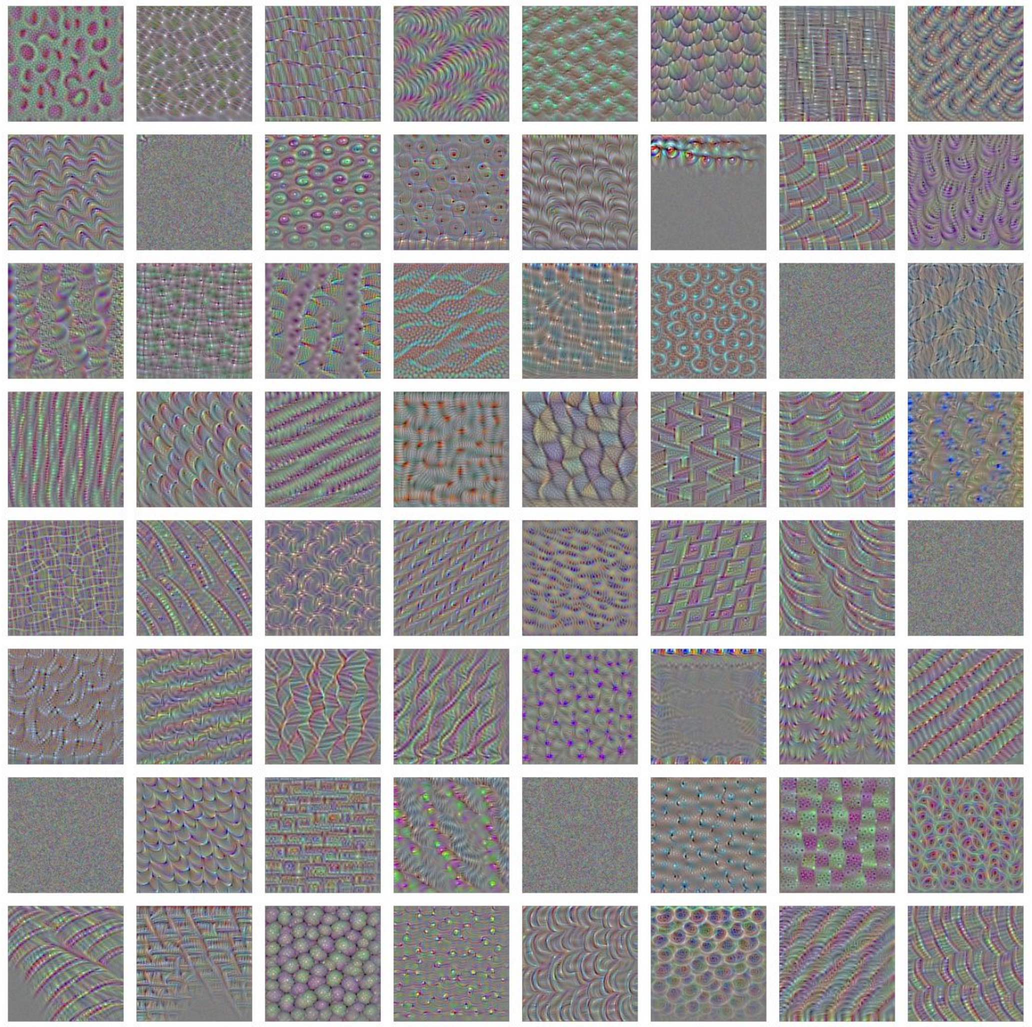
Activation map visualization:



What ConvNet learns ?

Filters (kernels) visualization:





Next meeting proposition

??.??.2020:

-

Questions ?