CIS 8392
Topics in Big Data Analytics

#Intro to Deep Learning

Yu-Kai Lin

Agenda

- What deep learning is
- What it can achieve
- How it works
- Deep learning frameworks
- A concrete example

[**Acknowledgements**] The materials in the following slides are based on the source(s) below:

- Deep Learning with R by Francois Chollet and J.J. Allaire
- R interface to Keras

Prerequisites

If you don't have a conda environment on your machine, please install Anaconda for Python 3.x: https://www.anaconda.com/distribution/

Click here for my R script to setup Keras or simply use the commands below.

```
install.packages(c("keras", "reticulate"))

library(keras)
library(reticulate)

install_keras() # will take a while; make sure no error message

# install with GPU version of TensorFlow (if you have an NVIDIA GPU)
# install_keras(tensorflow = "gpu")
```

If you use Windows and see error message like "CondaHTTPError: HTTP 000 Connection Failed ..." when you run `install_keras()`, follow the instruction below to install OpenSSL and run `install_keras()` again:

https://github.com/conda/conda/issues/8046#issuecomment-450492945

Verify keras has been successfully installed

```
library(keras)
library(reticulate)
library(tidyverse)
mnist <- keras::dataset_mnist()
str(mnist)
```

```
## List of 2
##  $ train:List of 2
##   ..$ x: int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ y: int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
##  $ test :List of 2
##   ..$ x: int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ y: int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...
```

Your keras installation was successful if you see such output.

If you are using a Mac laptop and see error message about **SSL Certificate**, your keras installation was successful but keras has problem connect googleapis.com to download the mnist dataset. In that case, download data from my Dropbox and run:

- https://www.dropbox.com/s/wun98d29bldsj10/mnist.rds?dl=0

```
mnist <- readRDS("mnist.rds") # put mnist.rds in your working directory
```

Troubleshooting keras installation issues

It is extremely difficult to troubleshoot errors in keras installation because it involves layers of software:

- R and the keras package in R
- Python/Conda and numerous packages in Python/Conda
- Operating system

If you are not able to setup keras on your laptop, please just use the VM provided to you, which has been tested successfully.

Nevertheless, below I provide a few diagnostic tests for you to identify (and possibly resolve) the installation problem.

Make sure you have python and R can recognize it.

You should see `[...]\r-reticulate\python.exe` in the output below:

```
library(reticulate)
py_config()
```

```
## python:         C:/Users/yklin/anaconda3/envs/r-reticulate/python.exe
## libpython:      C:/Users/yklin/anaconda3/envs/r-reticulate/python36.dll
## pythonhome:     C:/Users/yklin/anaconda3/envs/r-reticulate
## version:        3.6.12 |Anaconda, Inc.| (default, Sep  9 2020, 00:29:25) [MSC v.1916 64 bit
## Architecture:   64bit
## numpy:          C:/Users/yklin/anaconda3/envs/r-reticulate/Lib/site-packages/numpy
## numpy_version:  1.19.5
## tensorflow:     C:\Users\yklin\ANACON~1\envs\R-RETI~1\lib\site-packages\tensorflow\__init__
##
## python versions found:
##  C:/Users/yklin/anaconda3/envs/r-reticulate/python.exe
##  C:/Users/yklin/anaconda3/python.exe
##  C:/Users/yklin/anaconda3/envs/bert_env/python.exe
```

Make sure you have conda environment and R can recognize it.

You should see `r-reticulate` in the output below:

```
conda_list()
```

```
##           name                                                      python
## 1     bert_env     C:\\Users\\yklin\\anaconda3\\envs\\bert_env\\python.exe
## 2 r-reticulate C:\\Users\\yklin\\anaconda3\\envs\\r-reticulate\\python.exe
```

Make sure you have tensorflow installed (part of `install_keras()`).

You should see `[...]\r-reticulate\python.exe` and `tensorflow: [...]\lib\site-packages\tensorflow\` in the output below:

```
py_discover_config("tensorflow")
```

```
## python:         C:/Users/yklin/anaconda3/envs/r-reticulate/python.exe
## libpython:      C:/Users/yklin/anaconda3/envs/r-reticulate/python36.dll
## pythonhome:     C:/Users/yklin/anaconda3/envs/r-reticulate
## version:        3.6.12 |Anaconda, Inc.| (default, Sep  9 2020, 00:29:25) [MSC v.1916 64 bit
## Architecture:   64bit
## numpy:          C:/Users/yklin/anaconda3/envs/r-reticulate/Lib/site-packages/numpy
## numpy_version:  1.19.5
## tensorflow:     C:\Users\yklin\ANACON~1\envs\R-RETI~1\lib\site-packages\tensorflow\__init__
##
## python versions found:
##  C:/Users/yklin/anaconda3/envs/r-reticulate/python.exe
##  C:/Users/yklin/anaconda3/python.exe
##  C:/Users/yklin/anaconda3/envs/bert_env/python.exe
```

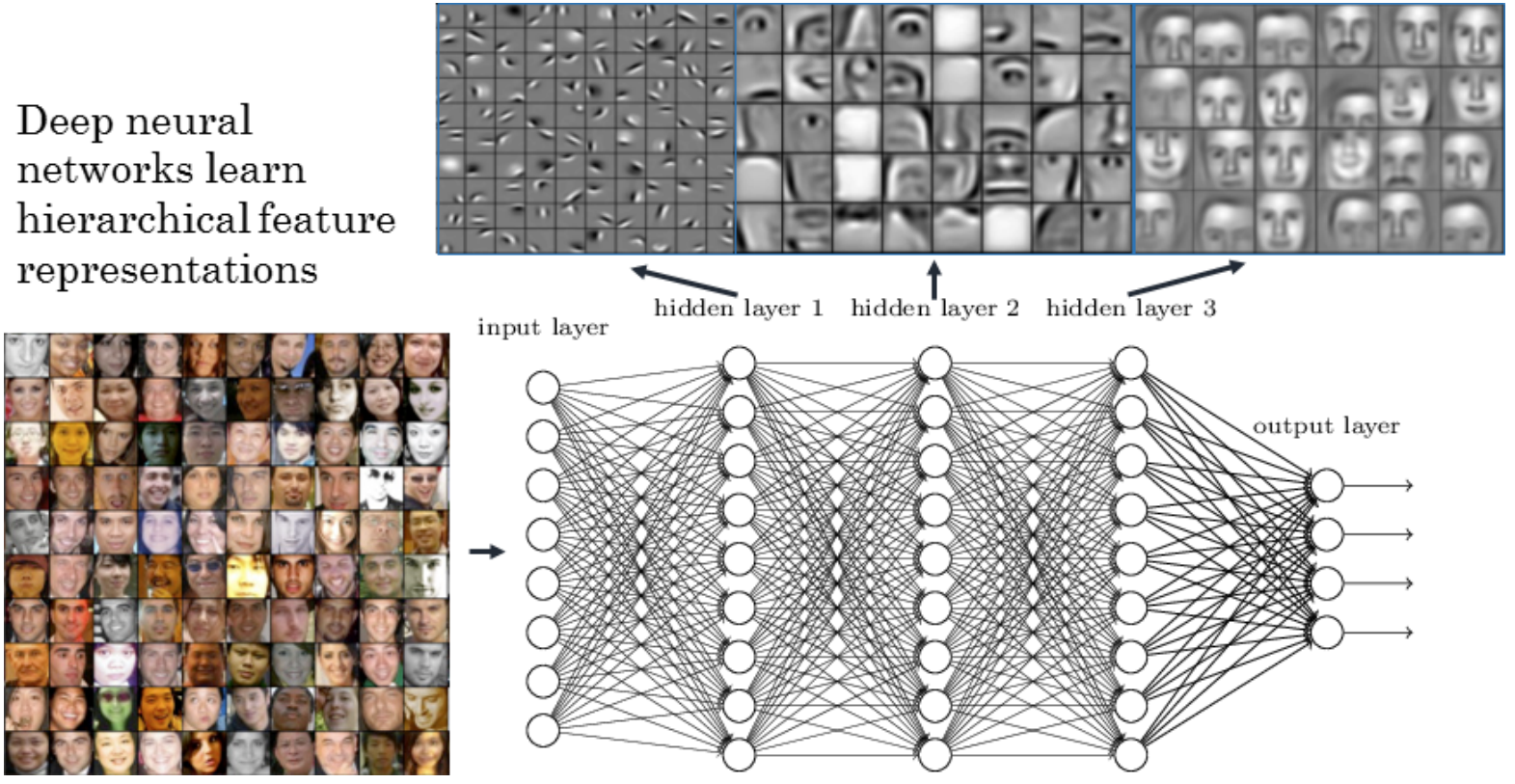
Specify the correct conda environment to use.

Once the three previous steps are all correct, ask R to use the `r-reticulate` conda environment:

```
use_condaenv("r-reticulate", required = T)
```

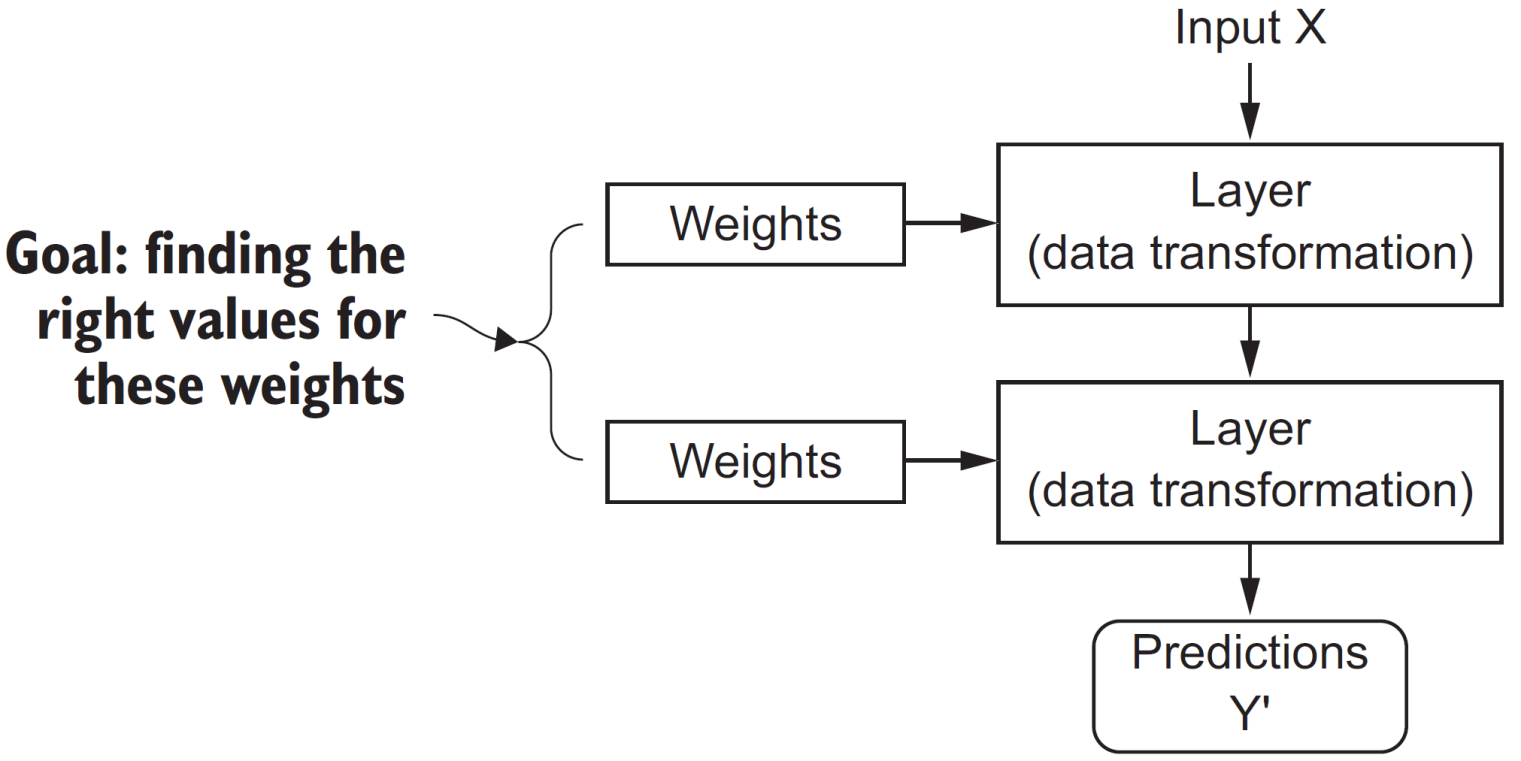
Neural networks (NNs)

In deep learning, these layered representations are learned via models called *neural networks*, structured in literal layers stacked on top of each other.



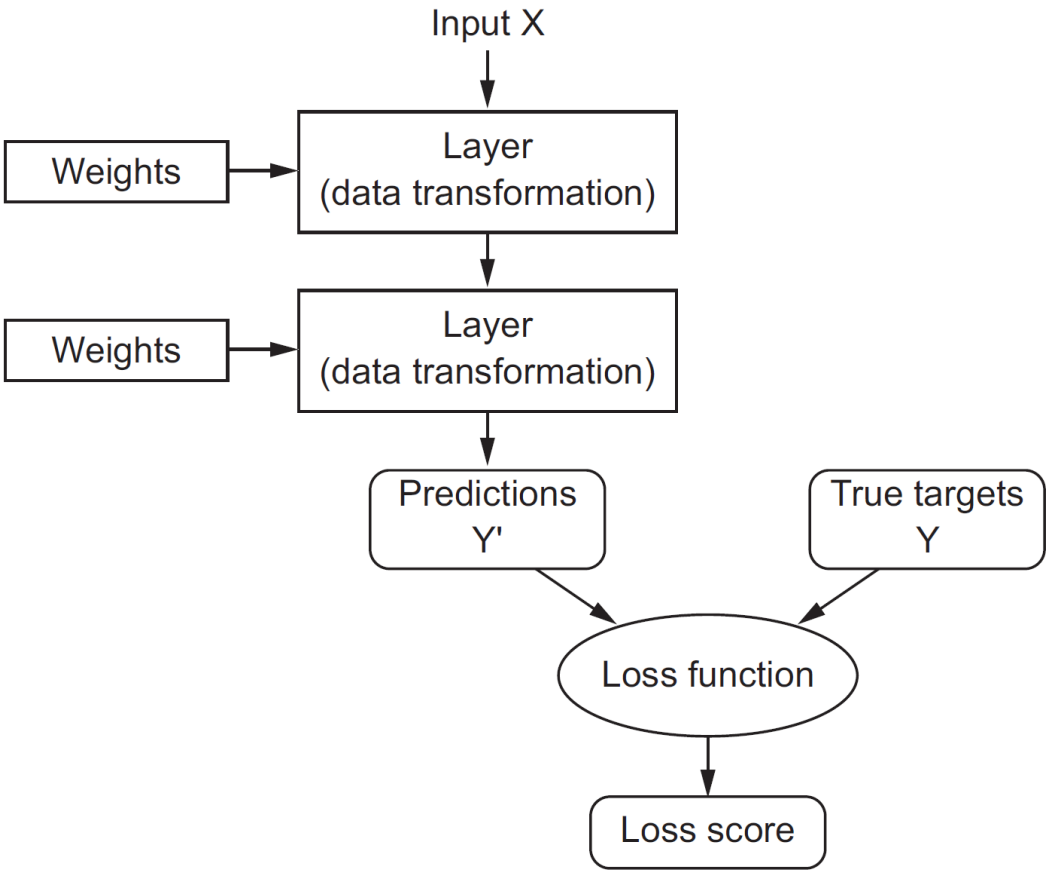
Principles in building NNs (1)

1. A neural network is parameterized by its weights



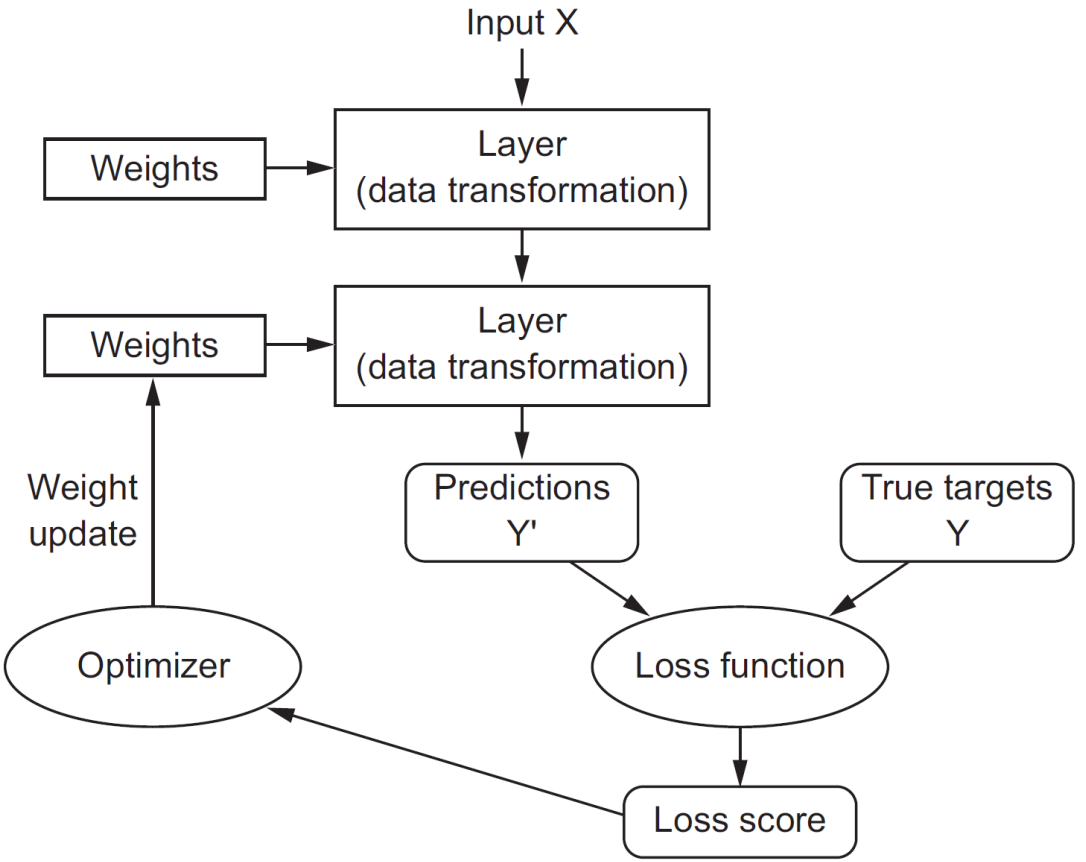
Principles in building NNs (2)

2. A loss function measures the quality of the network's output



Principles in building NNs (3)

3. The loss score is used as a feedback signal to adjust the weights (through backpropagation)



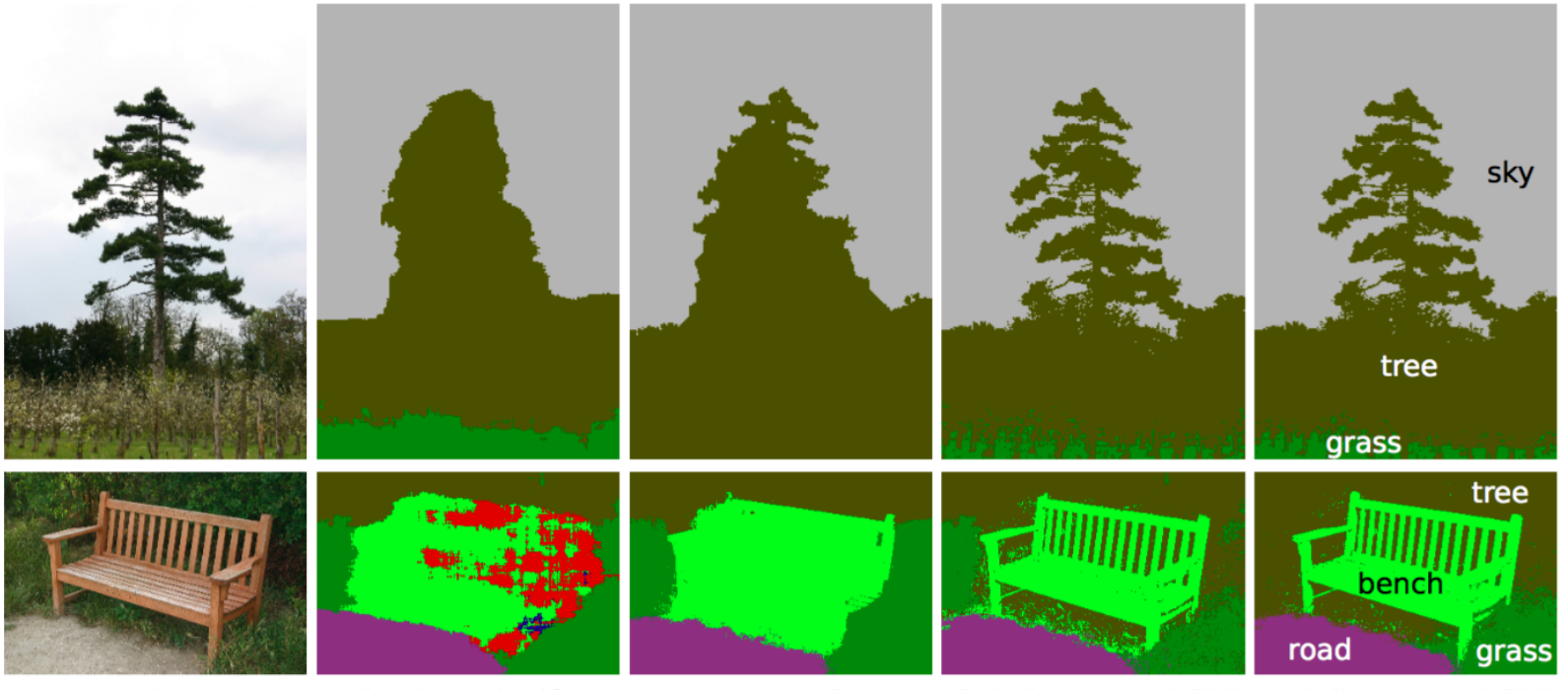
Why DL is suddenly becoming so popular?

- NNs were already well understood in the 1980s
- Kernel methods (e.g., support vector machines) and tree-based methods (e.g., random forests) were preferred techniques in ML in 1990s and 2000s
- We return to NNs around 2010 because of four technical advances:
  - Hardware: GPU, TPU, CUDA, BLAS
  - Datasets and benchmarks: ImageNet (1.4 million images; 1,000 image categories), Common Crawl (Internet archive)
  - Algorithmic advance:
    - Better activation functions for neural layers
    - Better weight-initialization schemes
    - Better optimization schemes
  - High level **DL frameworks** for Python and R developers

Self-driving Cars

Source: YouTube - Waymo's fully self-driving cars are here

Semantic Segmentation

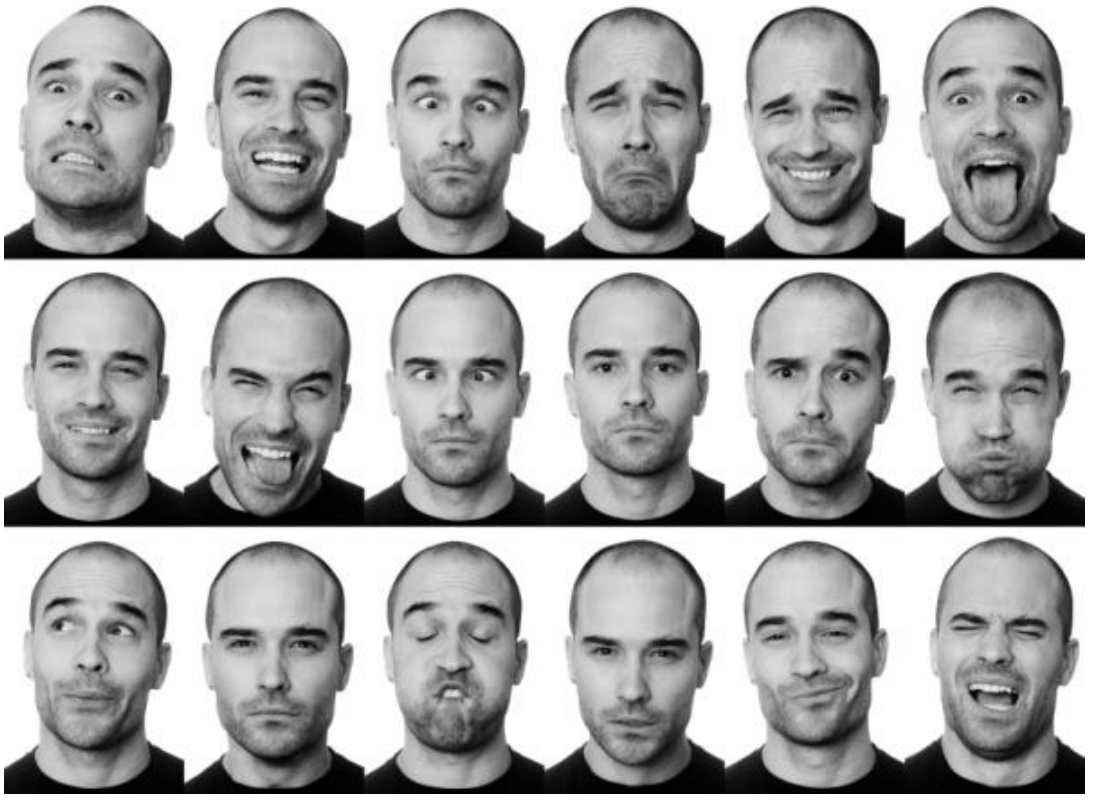


(a) Image (b) Unary classifiers (c) Robust P^n CRF (d) Fully connected CRF, MCMC inference, 36 hrs (e) Fully connected CRF, our approach, 0.2 seconds

---

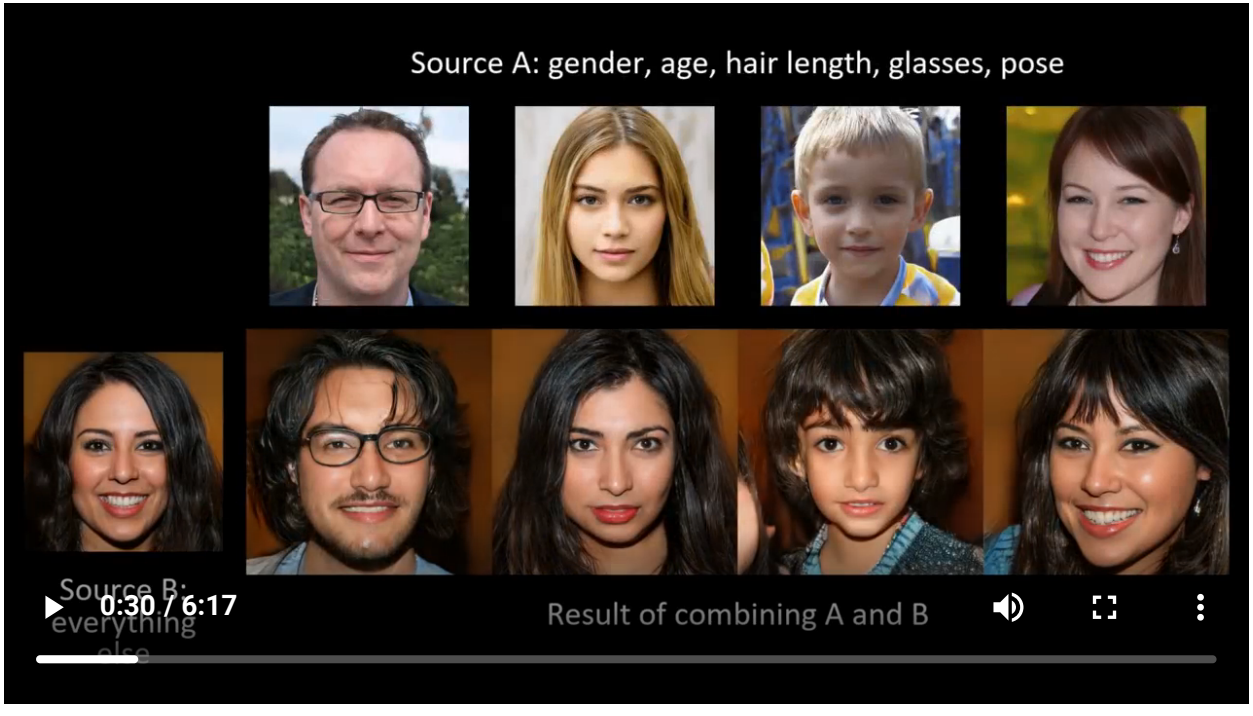
- Source: http://blog.qure.ai/notes/semantic-segmentation-deep-learning-review
- Extra: https://towardsdatascience.com/background-removal-with-deep-learning-c4f2104b3157

Affect Detection



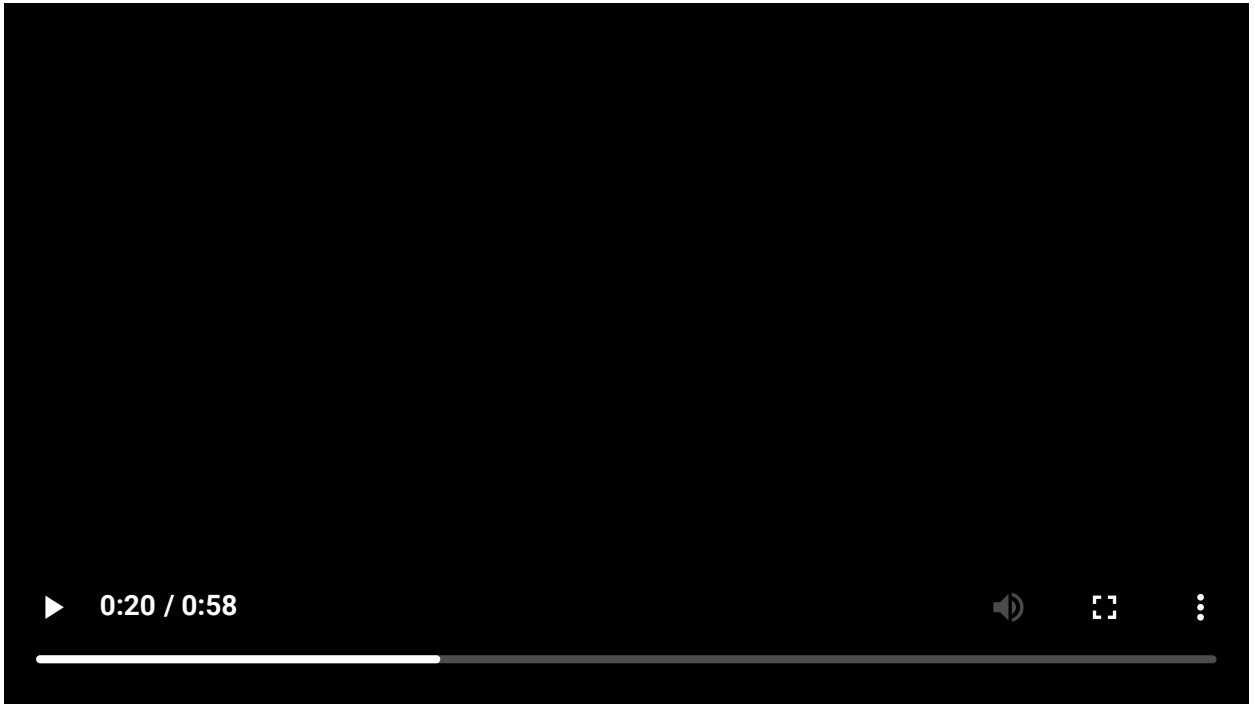
- Source: http://www.personal.psu.edu/afr3/blogs/siowfa12/2012/10/facial-expressions.html
- Extra: https://www.affectiva.com/ and https://www.kairos.com/

Face Generator



- Source: YouTube - NVIDIA's Hyperrealistic Face Generator
- Extra: https://www.thispersondoesnotexist.com/

Find Waldo



- Source: YouTube - There's Waldo is a robot that finds Waldo

Basic math concepts in neural networks

Tensors are a generalization of vectors and matrices to an arbitrary number of dimensions

- Scalars (0D tensors)

```
x <- 10
```

- Vectors (1D tensors)

```
x <- c(12, 3, 6, 14, 10)
str(x)
```

```
##  num [1:5] 12 3 6 14 10
```

```
dim(as.array(x))
```

```
## [1] 5
```

That's a five-dimensional vector. Don't confuse a 5D vector with a 5D tensor!

- Matrices (2D tensors)

```
x <- matrix(rep(0, 3*5), nrow=3, ncol=5)
dim(x)
```

```
## [1] 3 5
```

- 3D and higher-dimensional tensors

```
x <- array(rep(0,2*3*2), dim=c(2,3,2))
dim(x)
```

```
## [1] 2 3 2
```

By packing 3D tensors in an array, you can create a 4D tensor, and so on. In DL, you'll generally manipulate tensors that are 0D to 4D, although you may go up to 5D if you process video data.

Key attributes

A tensor is defined by three key attributes:

- **Number of axes (rank)**—For instance, a 3D tensor has three axes, and a matrix has two axes.
- **Shape**—This is an integer vector that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape `(3, 5)`, and the 3D tensor example has shape `(2, 3, 2)`.
- **Data type**—This is the type of the data contained in the tensor; for instance, a tensor's type could be integer or double.

To make this more concrete, let's look at an example:

MNIST database (Modified National Institute of Standards and Technology database)

- A large database of handwritten digits that is commonly used for training various image processing systems
- Contains 60,000 training images and 10,000 testing images
- 28 x 28 gray-scale images of handwritten digits like these
    - An 8-bit integer giving a range of possible values from 0 to 255
    - Typically zero is taken to be black, and 255 is taken to be white (see here)

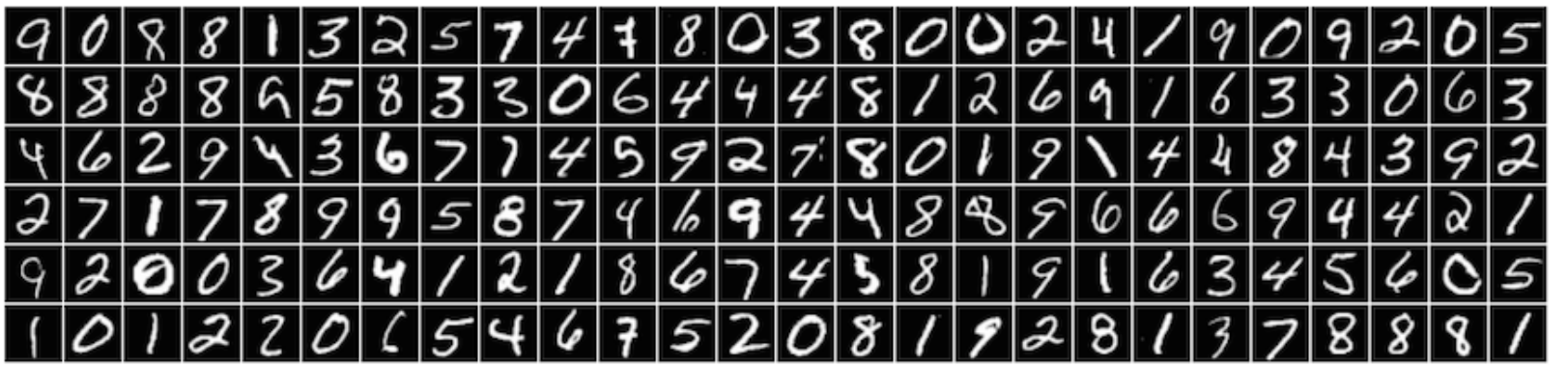


Image source: https://ml4a.github.io/ml4a/neural_networks/

```
mnist <- dataset_mnist() # get the data from the internet; ~200MB

x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

Next, we display the number of axes of the tensor `x_train`:

```
length(dim(x_train))
```

```
## [1] 3
```

Here's its shape:

```
dim(x_train)
```

```
## [1] 60000    28    28
```

And this is its data type:

```
typeof(x_train)
```

```
## [1] "integer"
```

So what we have here is a 3D tensor of integers. More precisely, it's an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a gray-scale image, with coefficients between 0 and 255. Let's plot the fifth digit in this 3D tensor:

```
digit <- x_train[5,,]
plot(as.raster(digit, max = 255))
```

Tips

Cache the dataset locally

Reading data from the Internet is slow. For frequently used data, we may want to save the data to our working directory so that we can read it back later.

```
write_rds(mnist, "mnist.rds")
mnist <- read_rds("mnist.rds")
```

Using the multi-assignment (`%<-%`) operator

The multi-assignment version is preferable because it's more compact. The datasets built into Keras are all nested lists of training and test data. Here, we use the multi-assignment operator (`%<-%`) from the `zeallot` package to unpack the list into a set of distinct variables.

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% mnist
```

Manipulating tensors in R

In the previous example, we *selected* a specific digit alongside the first axis using the syntax `train_images[i,,]`. Selecting specific elements in a tensor is called **tensor slicing**.

The following example selects digits #10 to #99 and puts them in an array of shape `(90, 28, 28)`:

```
my_slice <- x_train[10:99,,]
dim(my_slice)
```

```
## [1] 90 28 28
```

Equivalently,

```
my_slice <- x_train[10:99,1:28,1:28]
dim(my_slice)
```

```
## [1] 90 28 28
```

The notion of data batches

In general, the first axis in all data tensors you'll come across in deep learning will be the *samples axis* (sometimes called the *samples dimension*). In the MNIST example, samples are images of digits.

Deep-learning models typically don't process an entire dataset at once; rather, they break the data into small batches. Concretely, here's one batch of our MNIST digits, with batch size of 128:

```
batch <- x_train[1:128,,]
```

And here's the next batch:

```
batch <- x_train[129:256,,]
```

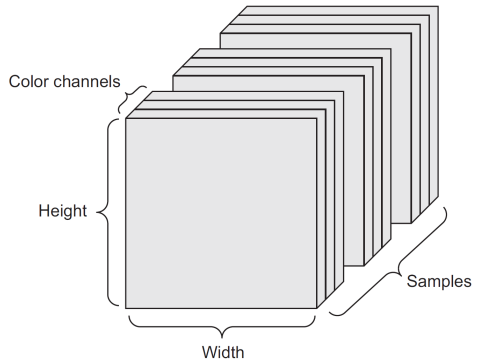
When considering such a batch tensor, the first axis is called the *batch axis* or *batch dimension*. This is a term you'll frequently encounter when using deep-learning libraries.

Real-world examples of data tensors

Vector data—2D tensors of shape (samples, features)

Time series data or sequence data—3D tensors of shape (samples, timesteps, features)

Images—4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)



- Some DL libraries use the *channels-last* convention (TensorFlow) while others use the *channels-first* convention (Theano)
  - A batch of 128 gray-scale images of size 256 × 256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`, and a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)`

Video—5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Your turn

Suppose we have a 5 minutes HD (1920×1080) YouTube video clip which runs 30 frames per second. How many values are in this video data tensor?

- How many sample?
- How many frames?
- How many channels?
- Height?
- Width?

Total?

Deep learning frameworks

You typically won't need to build deep learning applications from scratch. Most people use an existing deep learning framework/toolkit to help them build their applications.

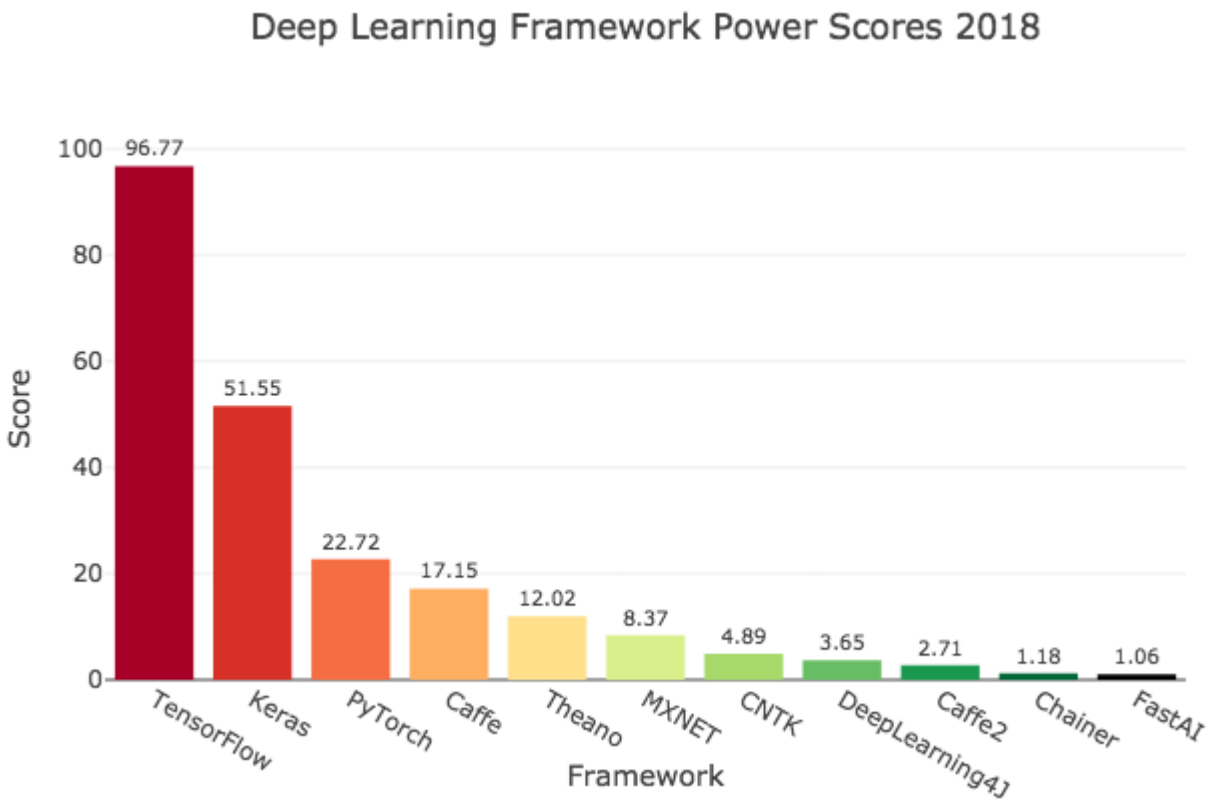
Popular deep learning frameworks:

- TensorFlow: Powerful, sophisticated, but difficult to learn. Backed by Google.
- Keras: Runs on top of TensorFlow, Theano, or CNTK; not as sophisticated as TensorFlow but still powerful; beginner friendly
- PyTorch: Powerful and flexible for customization and fast prototyping; backed by Facebook.
- ...

Although TensorFlow and Keras are written in Python, there are R interfaces that allow you to use these frameworks in R (see here and here).

Jeff Hale made a very interesting summary and comparison for these frameworks based on 7 evaluation categories:

- Online Job Listings; KDnuggets Usage Survey; Google Search Volume; Medium Articles; Amazon Books; ArXiv Articles; GitHub Activity



Source: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

Keras

We will learn how to use the R interface to Keras for deep learning

Developing with Keras: a quick overview

1. Prepare the data

```
str(mnist)
```

```
## List of 2
##  $ train:List of 2
##   ..$ x: int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ y: int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
##  $ test :List of 2
##   ..$ x: int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ y: int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...
```

```
## The x data is a 3-d array (images,width,height) of of grayscale values
# View(mnist$train$x[1,,]) # Let's look at the first training example

c(c(x_train, y_train), c(x_test, y_test)) %<-% mnist

str(x_train)
```

```
##  int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
```

```
## Reshape x from 3d to 2d (by default, row-major)
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))

str(x_train) # View(x_train[1,])
```

```
##  num [1:60000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
```

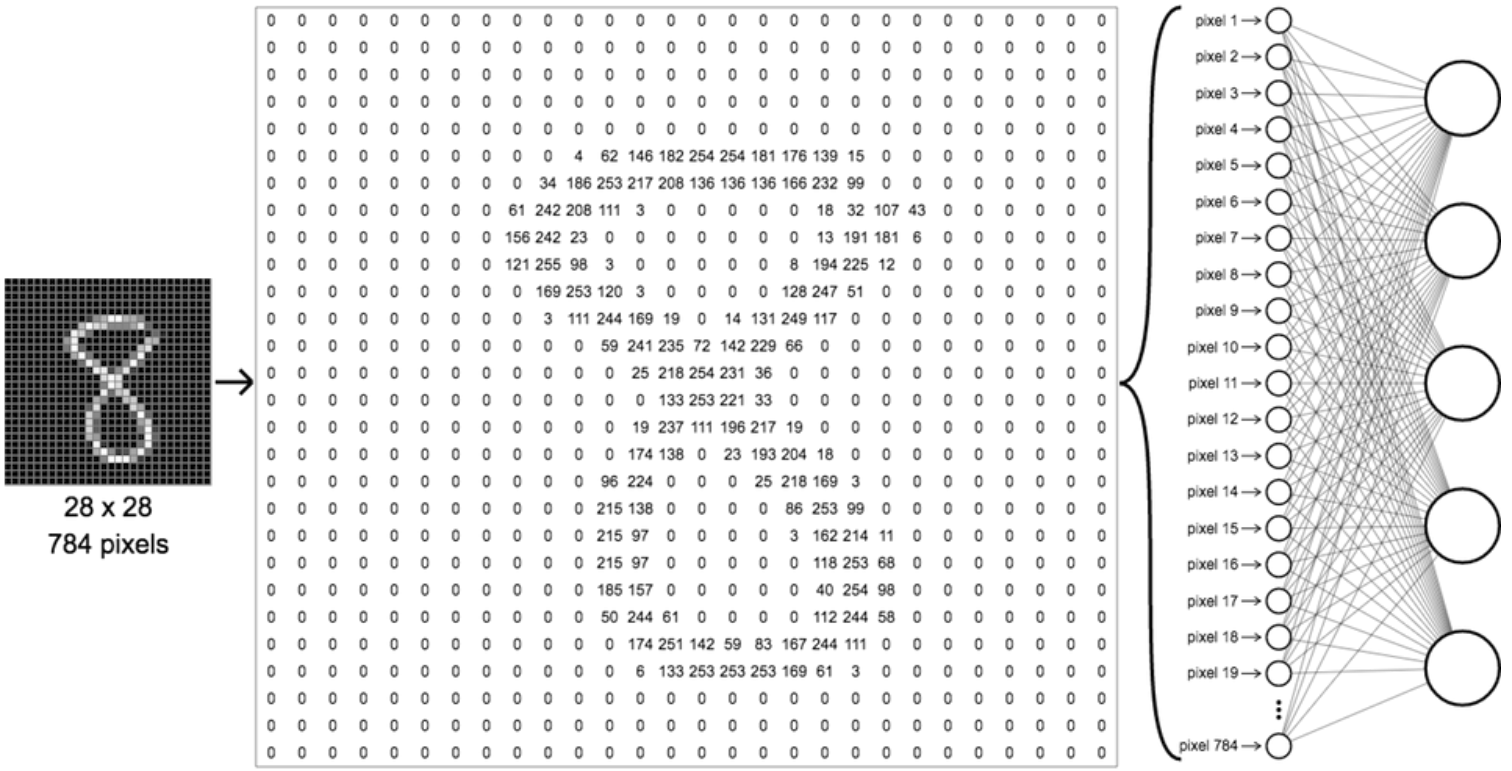


Image source: https://ml4a.github.io/ml4a/neural_networks/

```
## Rescale values so that they are between 0 and 1
x_train <- x_train / 255
x_test <- x_test / 255

# View(x_train[1,])

## Prepare the y data
## The y data is an integer vector with values ranging from 0 to 9
str(y_train) # View(y_train)
```

```
##  int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
```

```
## Deep learning models prefer binary values rather than integers
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
str(y_train) # View(y_train)
```

```
##  num [1:60000, 1:10] 0 1 0 0 0 0 0 0 0 0 ...
```

2. Defining the Model

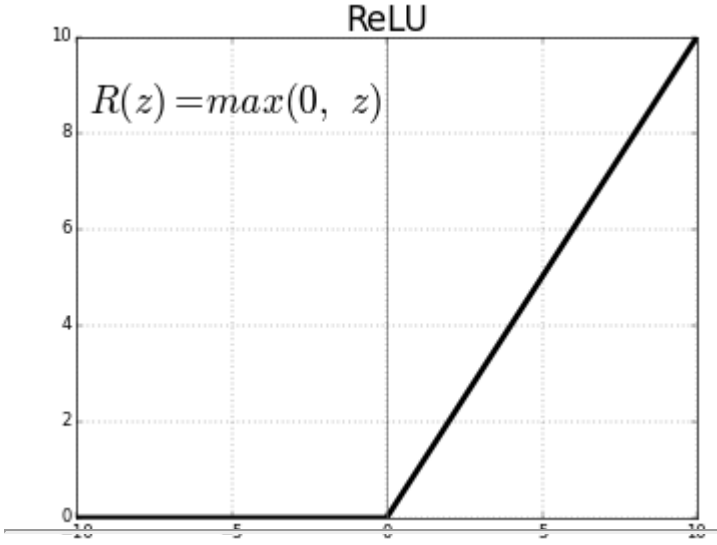
```
## Sequential model is simply a linear stack of layers
model <- keras_model_sequential()

## Define the structure of the neural net
model %>% # A dense layer is a fully connected layer
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>% # randomly set 40% of weights to 0
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>% # this helps prevent overfitting
  layer_dense(units = 10, activation = 'softmax') # probability of each class

summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =========================================================================
## dense_2 (Dense)                      (None, 256)                      200960
## _________________________________________________________________________
## dropout_1 (Dropout)                  (None, 256)                      0
## _________________________________________________________________________
## dense_1 (Dense)                      (None, 128)                      32896
## _________________________________________________________________________
## dropout (Dropout)                    (None, 128)                      0
## _________________________________________________________________________
## dense (Dense)                        (None, 10)                       1290
```

ReLU: rectified linear unit

$$R(x) = max(0, x)$$

ReLU activations are the simplest non-linear activation function you can use, resulting in much faster training for large networks.



Softmax: normalized exponential function

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for all } j = 1, \ldots, K$$

Softmax squashes values in a vector to be between 0 and 1. So the outputs represent the probability of each class category.

Image sources:

1. https://ml4a.github.io/ml4a/neural_networks/
2. https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions

Choosing the right **last-layer activation and loss function**:

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

```
## Compile the model with appropriate loss function, optimizer, and metrics
model %>% compile(
  optimizer = optimizer_rmsprop(),      # see next slide
  loss = 'categorical_crossentropy',    # since we have 10 categoreis
  metrics = c('accuracy')               # for classification
)
```

Before training a model, you need to **configure the learning process**, which is done via the `compile` method. It receives three arguments:

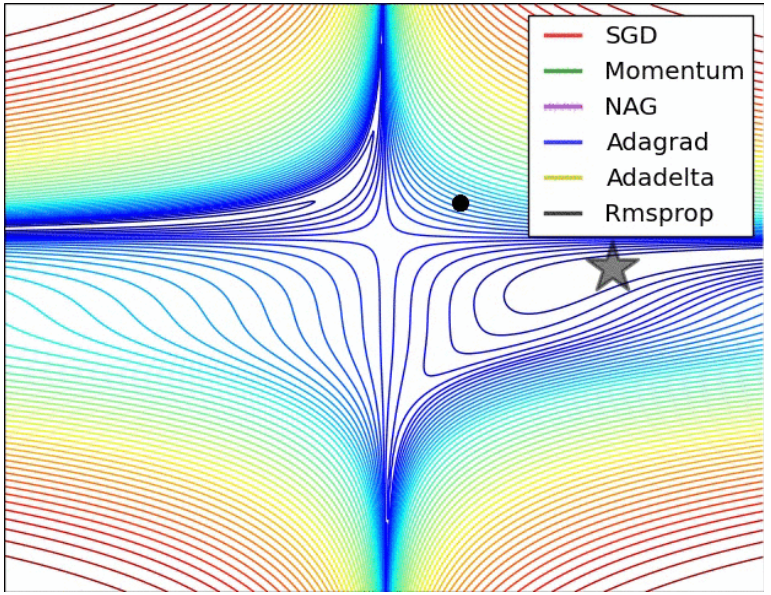
- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the Optimizer class. List of existing optimizer
- A loss function. This is the objective that the optimizer will try to minimize. It can be the string identifier of an existing loss function, or it can be a customized objective function. List of existing loss functions
- A list of metrics. A metric is used to judge the performance of your model. This is only for you to look at and has nothing to do with the optimization process. A metric could be the string identifier of an existing metric or a custom metric function. List of existing metrics

Each of the optimizers in Keras is a specific variant of **stochastic gradient descent (SGD)**.

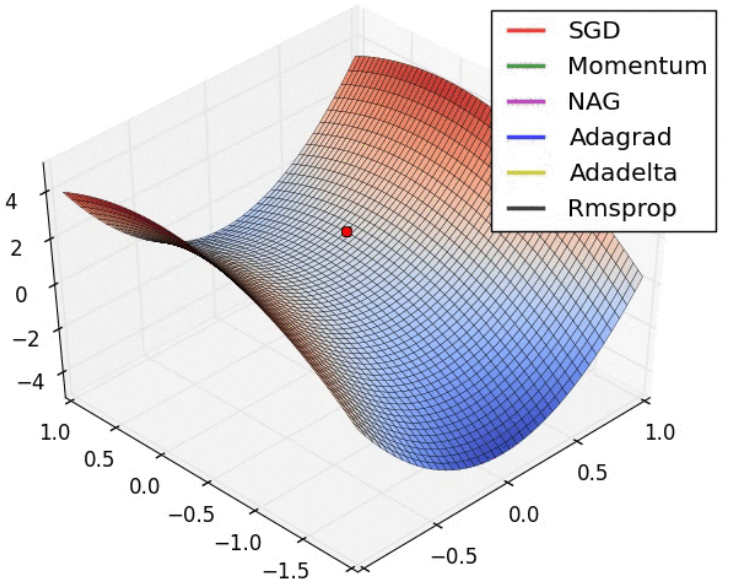
SGD optimizers explore the loss surface based on the slope (gradient) of the current state. To get to the top as fast as they can, they take big steps in the steepest direction and do smaller and smaller steps as they come further to the top to avoid overshoot it.

The Bird Box challenge

The learning process dynamics of different optimizers:



Contours of a loss surface and time evolution of different optimization algorithms



A saddle point in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down)

Image source: http://cs231n.github.io/neural-networks-3/

3. Training and Evaluation

```
## Use x_train and y_train for training
history <- model %>% fit(
  x_train, y_train,
  batch_size = 128,      # a set of 128 samples
  epochs = 30,           # let's go through x_train 30 times
  validation_split = 0.2 # use the last 20% of train data for validation
)
plot(history)
```

```
## Use x_test and y_test for evaluation
model %>% evaluate(x_test, y_test)
```

```
##      loss  accuracy
## 0.1170008 0.9810000
```

```
model %>% predict(x_test) %>% k_argmax()
```

```
## tf.Tensor([7 2 1 ... 4 5 6], shape=(10000,), dtype=int64)
```

4. Save and Reuse

```
# save the model to your current working directory
model %>% save_model_hdf5("mnist_model.h5")
```

The following components of the model are saved:

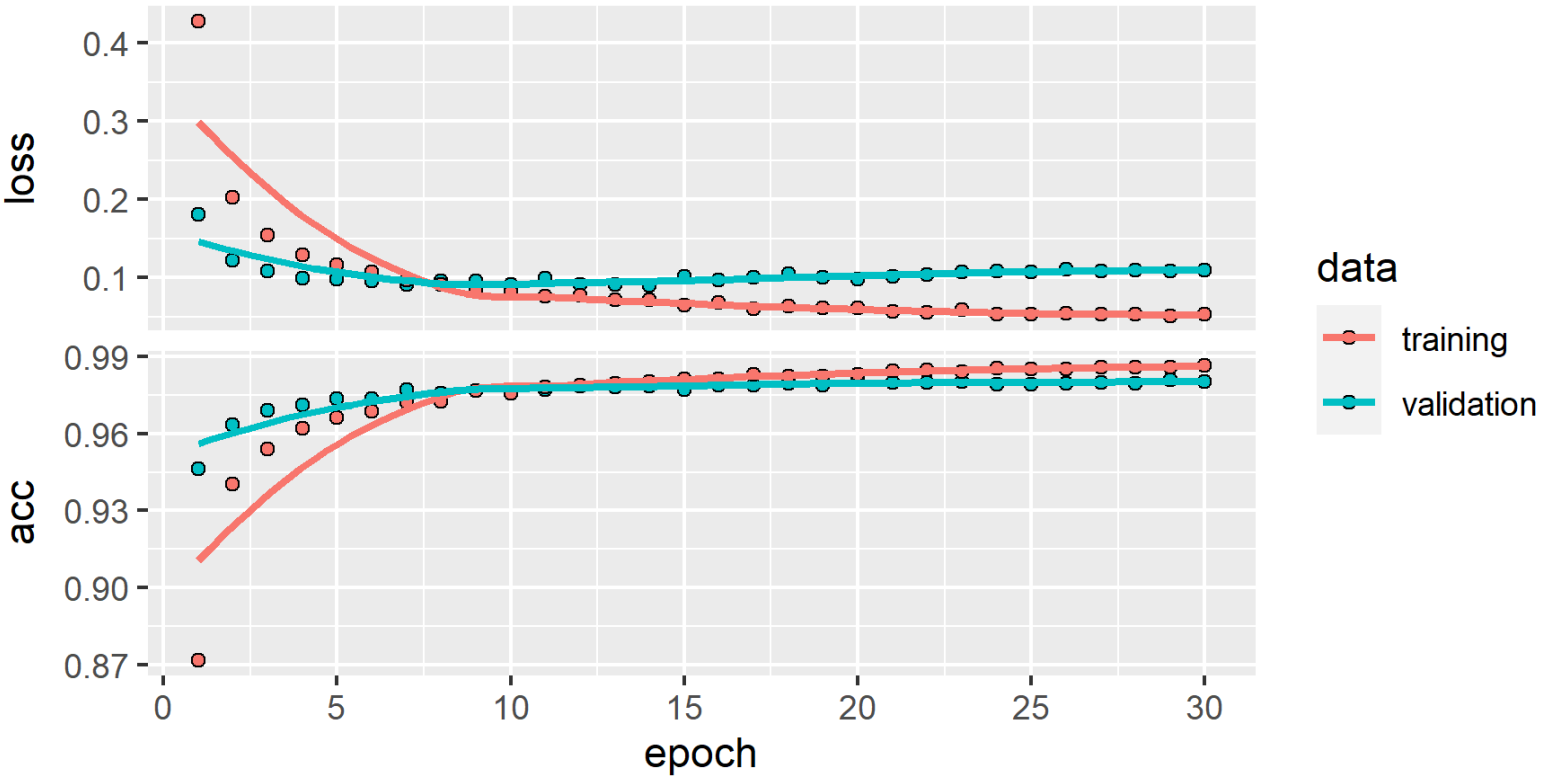
- The model architecture, allowing to re-instantiate the model.
- The model weights.
- The state of the optimizer, allowing to resume training exactly where you left off. This allows you to save the entirety of the state of a model in a single file.

Saved models can be re-instantiated via `load_model_hdf5()`. The model returned by `load_model_hdf5()` is a compiled model ready to be used.

```
# read from your current working directory
# make sure you have the file in that folder
new_model <- load_model_hdf5("mnist_model.h5")
```

You can also save the training history:

```
write_rds(history, "mnist_fit_history.rds")
history_new = read_rds("mnist_fit_history.rds")
plot(history_new)
```



Your turn

Based on the mnist example in the previous slides, try to experiment with different settings and see how the accuracy for test data changes under these settings.

- Number of layers
- Number of nodes in the layers
- Dropout rate
- Optimizer
- Batch size
- Epochs