



# Deep Learning

*Hands-on*

Elisa Ricci

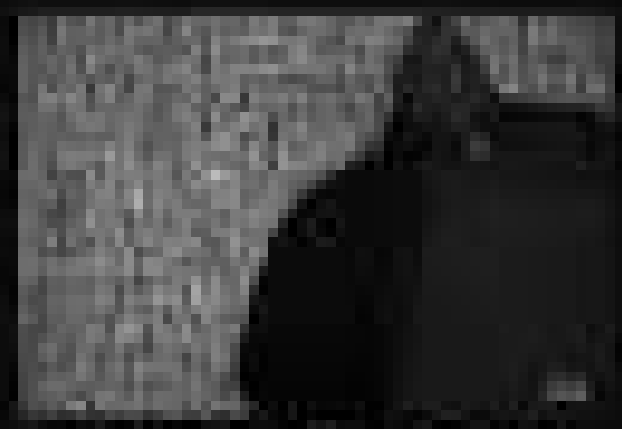
# Deep Learning



What society thinks I do



What my friends think I do



What other computer scientists think I do



What mathematicians think I do




What I think I do

From [Elemental Impact](#) [1]

What I actually do

# Outline

- Deep Learning Frameworks
- Introduction to TensorFlow
  - Examples (linear regression, MNIST)
- Introduction to Keras
  - Examples (MNIST MLP & CNN)



# Deep Learning Frameworks

# Deep Learning Frameworks

- Many different frameworks over the past few years...



# Deep Learning Frameworks

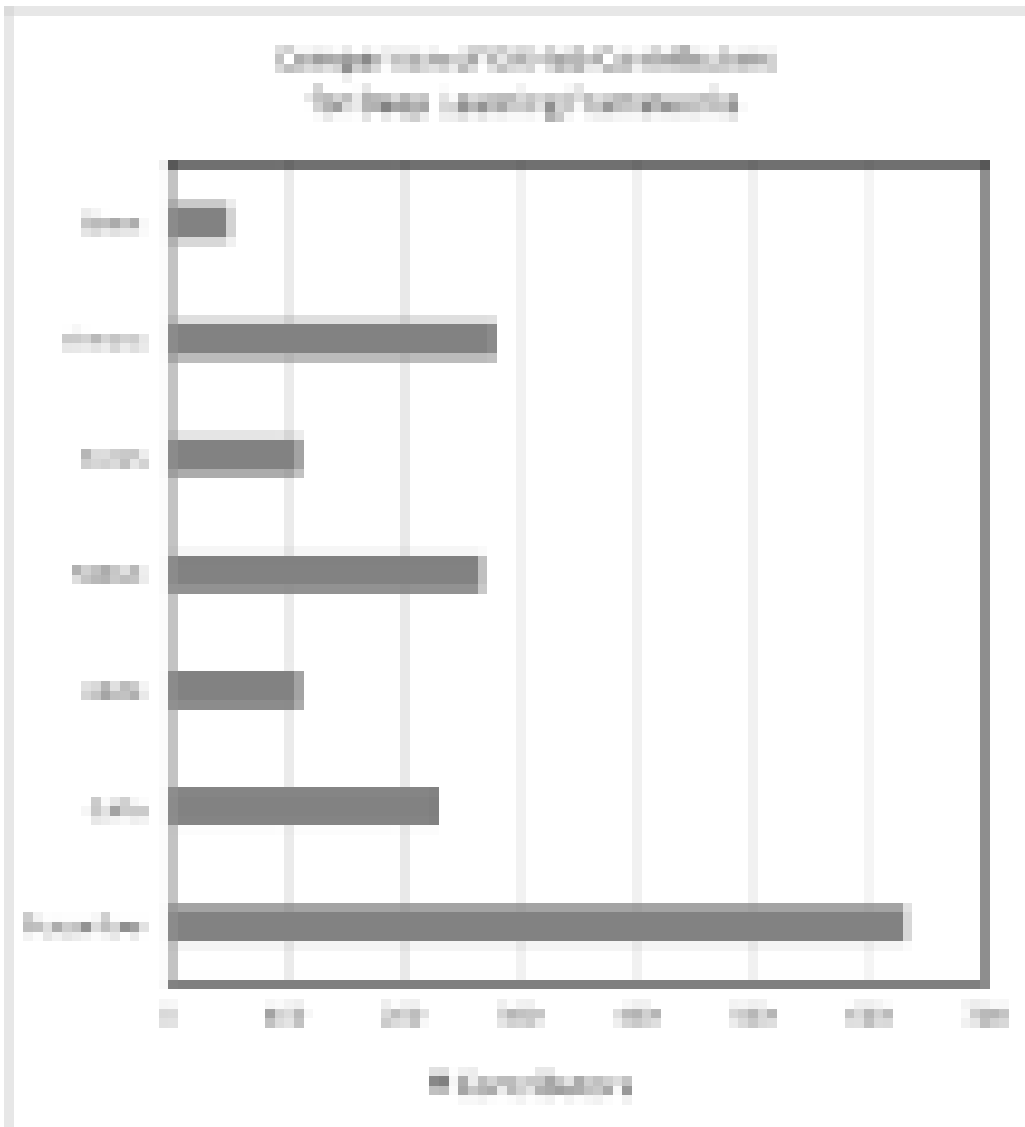
TensorFlow	Google Brain, 2015 (nowadays Discontinued)
Theano	University of Montreal, 2008
Keras	François Chollet, 2015 (now at Google)
Travis	Facebook AI Research, Twitter, Google DeepMind
Caffe	Berkeley Vision and Learning Center (BVLC), 2010

# Deep Learning Frameworks

- Which framework to choose? Look at GitHub...



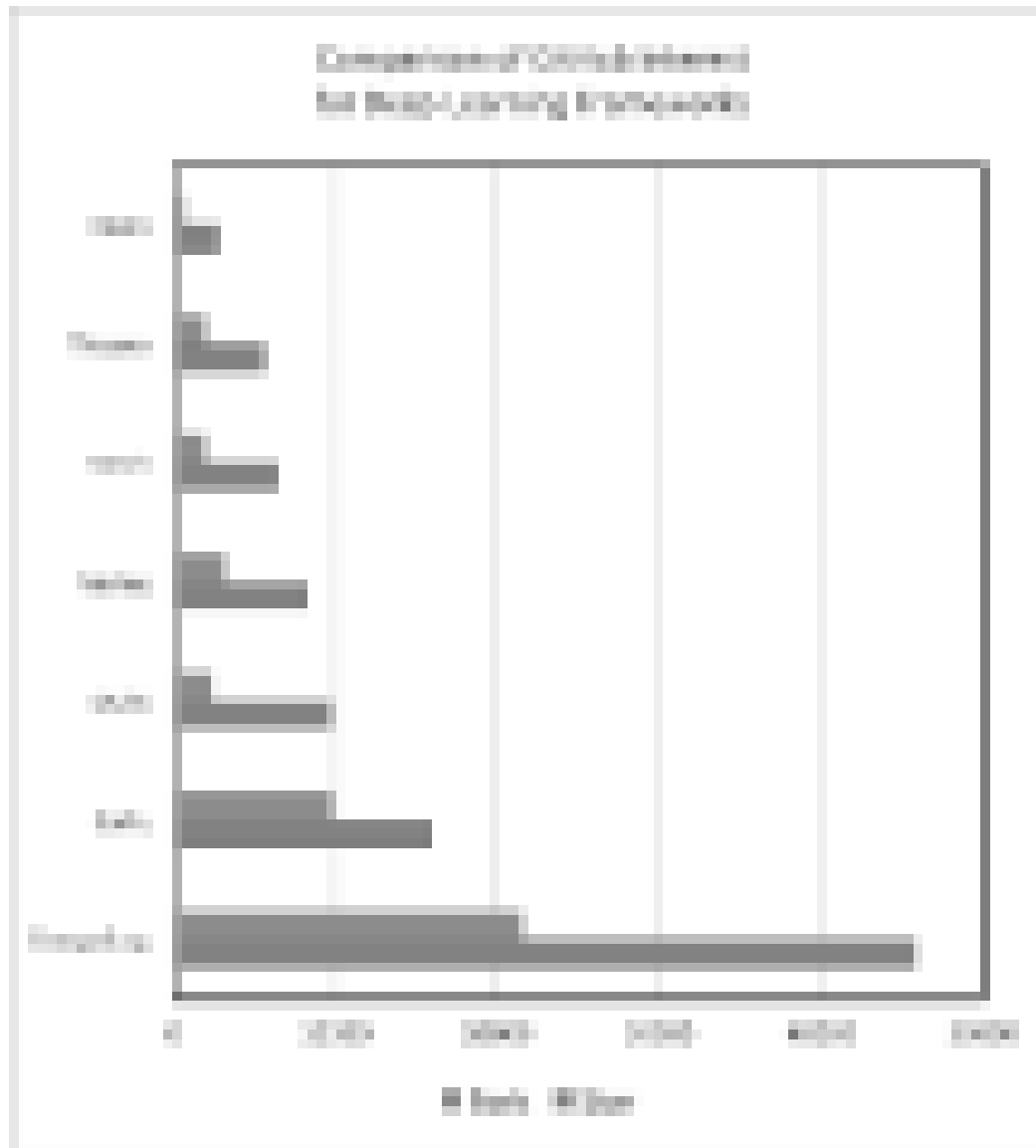
# Deep Learning Frameworks



[Rubashkin]

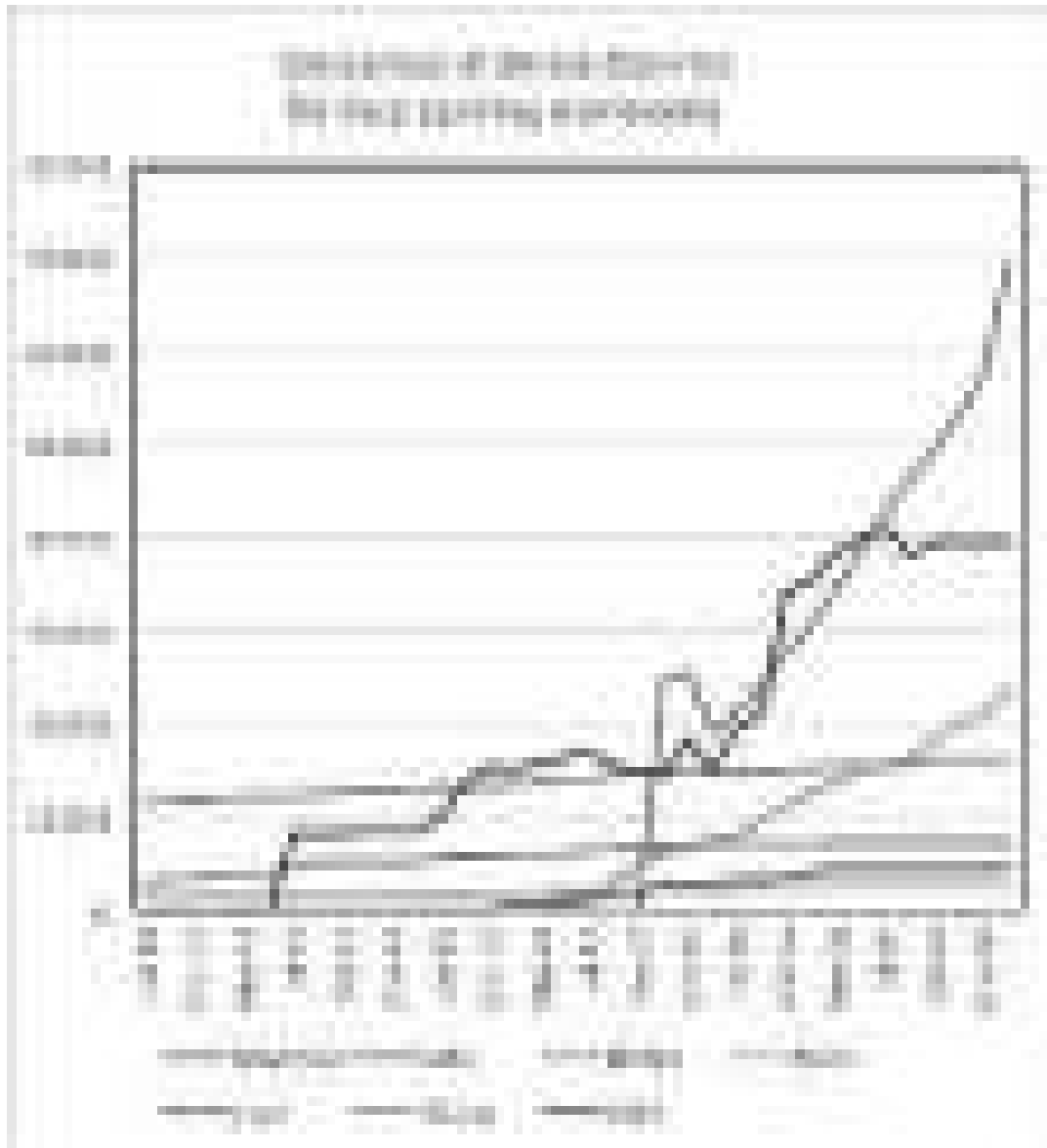


# Deep Learning Frameworks



[Rubashkin]

# Deep Learning Frameworks



[Rubashkin]

# Community and Resources

- (Github, groups, discussions...)
  - For CNNs Caffe has the largest community
  - TensorFlow's is already large and growing
  - Keras' community is growing
  - Theano's and Lasagne's community are declining

# Theano

theano

- Maintained by Montréal University group
- Pioneered the use of a computational graph
- General machine learning tool
- Symbolic differentiation
- Use of Lasagne and Keras
- Very popular in the research community, but not much elsewhere.  
Falling behind

# Torch



- Mixed language:
  - C/CUDA backend built on common backend libraries
  - Lua frontend
- Flexibility: existing building blocks from the community can be easily integrated
- Automatic differentiation
- Modularity
- Speed
- *(People hate Lua) → very recently PyTorch*

# Caffe

The word "Caffe" is displayed in a pixelated, monospace-style font.

- Pros:
  - Especially good for CNN and Computer Vision
  - Extremely easy to code
  - Easy to use pretrained models
  - Matlab and Python interface
  - Easy to include different libraries
  - Layer as building block and many layers already implemented online

# Caffe

# Caffe

- Cons:
  - No auto-differentiation
  - Need to write C++/CUDA for new GPU layers
  - Not good for RNN
  - Cumbersome for big networks (ResNet)

# Caffe

The word "Caffe" is displayed in a pixelated, monospace-style font.

- Main steps:
  - creation of the training network for learning and test network(s) for evaluation
  - iterative optimization by calling forward/backward and parameter updating
  - (periodical) evaluation of the test networks
  - snapshotting of the model and solver state throughout the optimization





# Caffe

# Caffe

- Solver:

```
base_lr: 0.01
```

```
lr_policy: "step"
```

```
gamma: 0.1
```

```
stepsize: 20000
```

```
max_iter: 20000
```

```
# Apply training at a learning rate of 0.01 = 1%.
```

```
# Learning rate policy, decreases learning rate as follows:
```

```
# 1) by a factor of gamma every stepsize iterations.
```

```
# 2) drop the learning rate by a factor of 0.1
```

```
# (i.e.) multiply it by a factor of gamma-0.1.
```

```
# Drop the learning rate every 2000 iterations.
```

```
# Train for 20000 iterations total.
```

# Which framework to chose

Framework	Integration	System and External Interactions	Data Interchange	API Interchange	Architecture (API, External, Internal)	Usage	Integration with Existing	Security Interchange
Spring	High	++	++	++	+	++	+	+
Spring Boot	High	+++	+++	++	+++	++	++	+
Quarkus	High, but not overall	+	+++	++	+++	+++	+++	
Vert.x	High	+	++		+	+	+	

[Rubashkin]

# Which framework to chose

- You work in industry:
  - TensorFlow, Caffe
- You want to work “seriously” on new models (research-oriented):
  - TensorFlow, Theano, (Torch)
- You don't have time and you are just curious about deep learning:
  - Keras, Caffe
- You want to use deep learning for educational purposes:
  - Keras, Caffe



TensorFlow

# TensorFlow

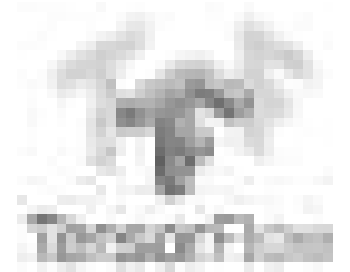
- An open-source software library for *Machine Intelligence*
- Especially useful for Deep Learning
- For research & industry



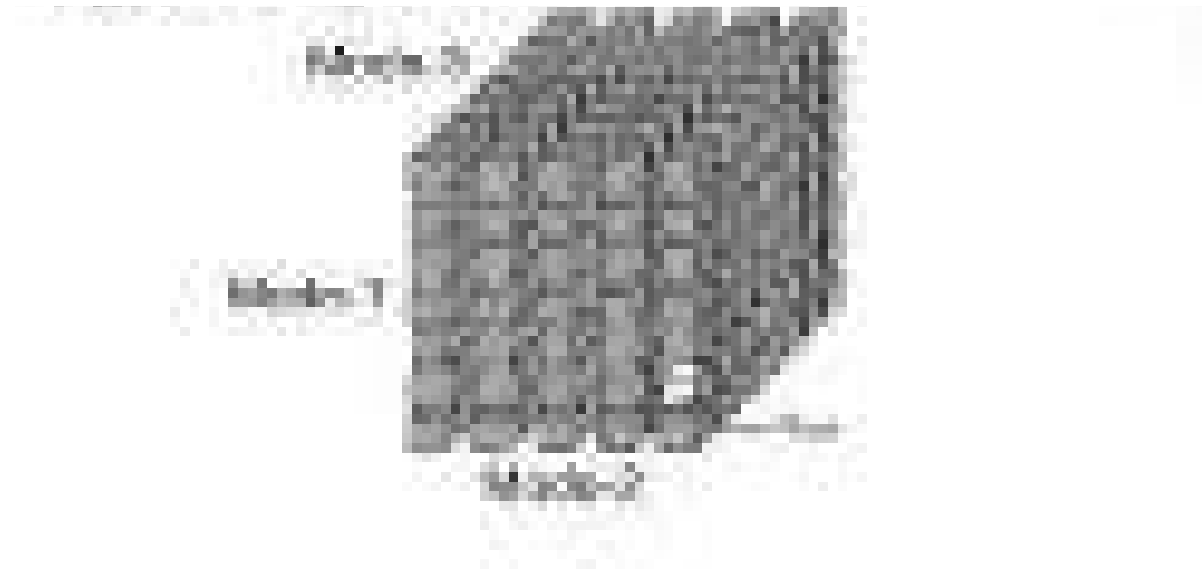
# TensorFlow



# TensorFlow

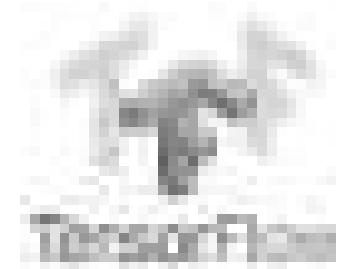


Tensors: multidimensional arrays





# TensorFlow

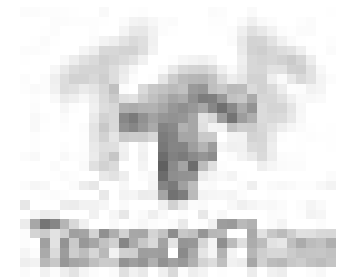


Tensors: multidimensional arrays

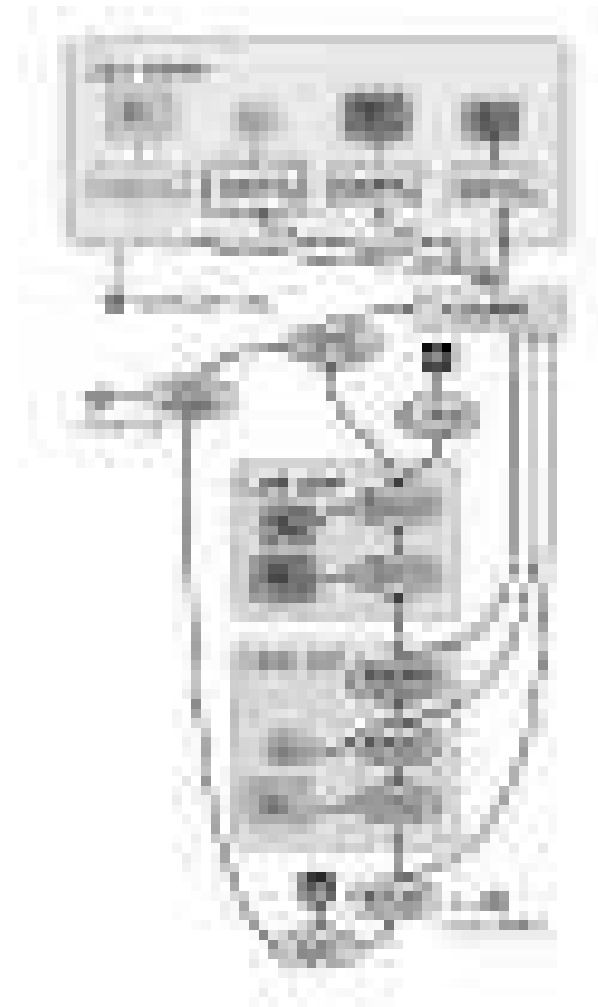
The central unit of computation in TensorFlow is the tensor. A tensor consists of a set of numerical values arranged into one or more dimensions. Tensors can be used to represent data, such as images, audio, and text, and they are used to perform operations on that data.

```
import tensorflow as tf
# Create a 1D tensor with values [1, 2, 3]
t1 = tf.constant([1, 2, 3])
# Create a 2D tensor with values [[1, 2], [3, 4]]
t2 = tf.constant([[1, 2], [3, 4]])
# Create a 3D tensor with values [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
t3 = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

# TensorFlow



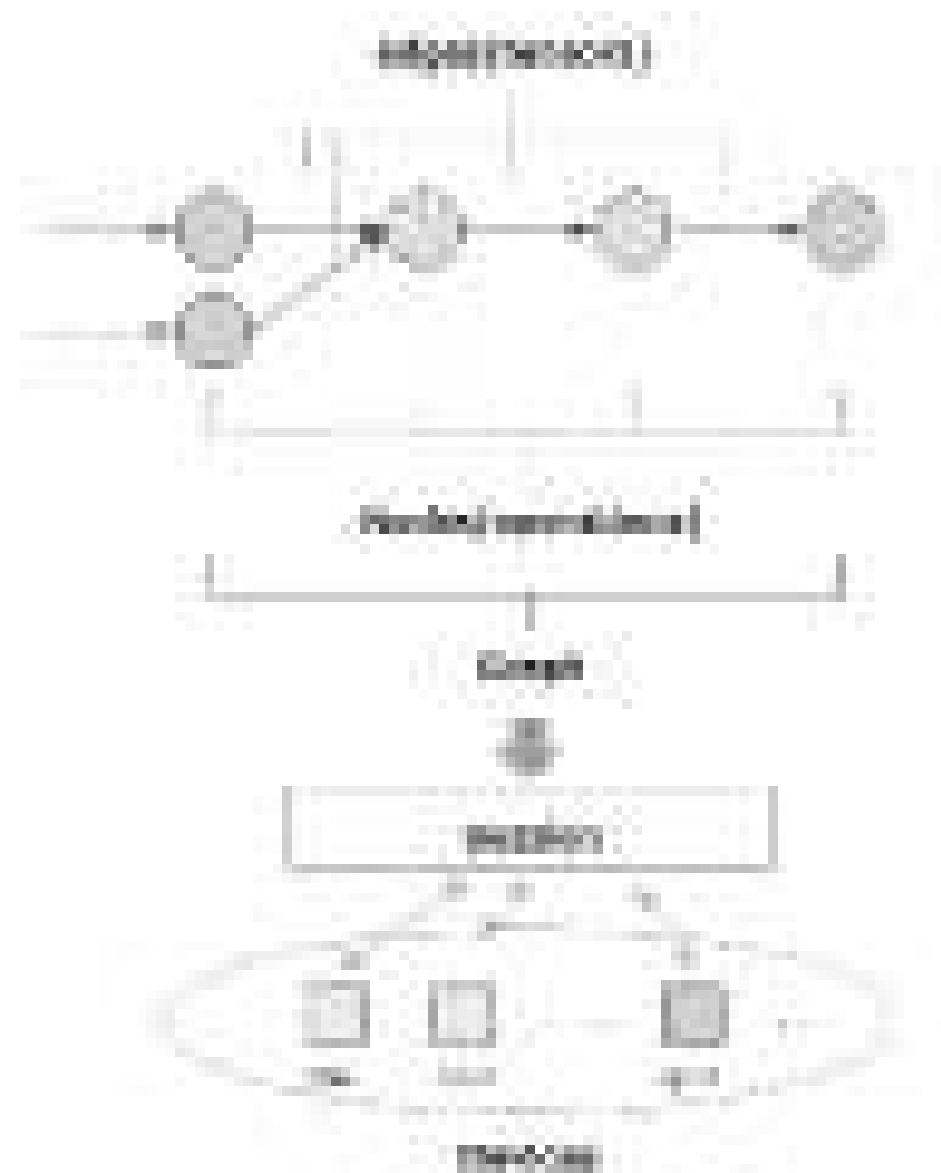
Flow: Graph describing operations



# DataFlow Graph

- Computation is defined as a directed acyclic graph (DAG) to optimize an objective function
- Graph is defined in high-level language (Python, C++)
- Graph is compiled and optimized
- Graph is executed (in parts or fully) on available low level devices (CPU, GPU, Android)
- Data (tensors) flow through the graph

# TensorFlow Idea



# Automatic differentiation

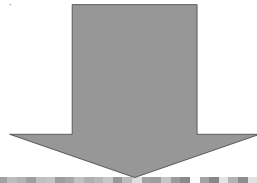
- TensorFlow can compute gradients automatically
  - Reverse automatic differentiation
  - In a nutshell:
    - When you define an operator (op), you also define together how its derivatives are computed (of course most of the common ops are already provided).
    - After you write a function by stacking a series of ops, the program can figure out by itself how should the corresponding derivatives be computed (usually by keeping some computation graphs and using the chain rule).
    - The benefit is obvious as it saves us from working out the math, writing the code, verifying the derivatives numerically...

# Main Components

- The main components of Tensorflow:
  - **Variables:** Retain values between sessions, use for weights/bias
  - **Nodes:** The operations
  - **Tensors:** Signals that pass from/to nodes
  - **Placeholders:** used to send data between your program and the tensorflow graph
  - **Session:** Place when graph is executed.

# What we do

- Create a graph using code C++ or Python and ask TensorFlow to execute this graph.



# What we do

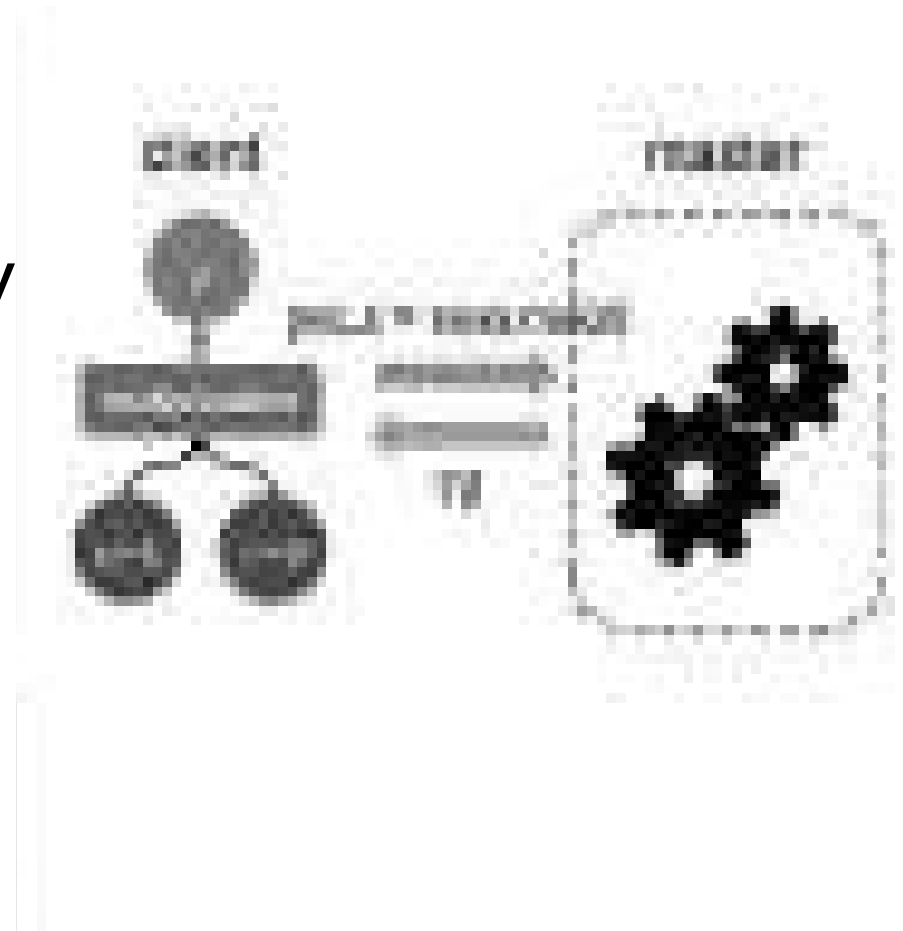
- Execution





# Hello world

- Multiply two numbers
- Main phases:
  - Import TensorFlow library
  - Build the graph
  - Create a session
  - Run the session



# Hello world

- Multiply two numbers

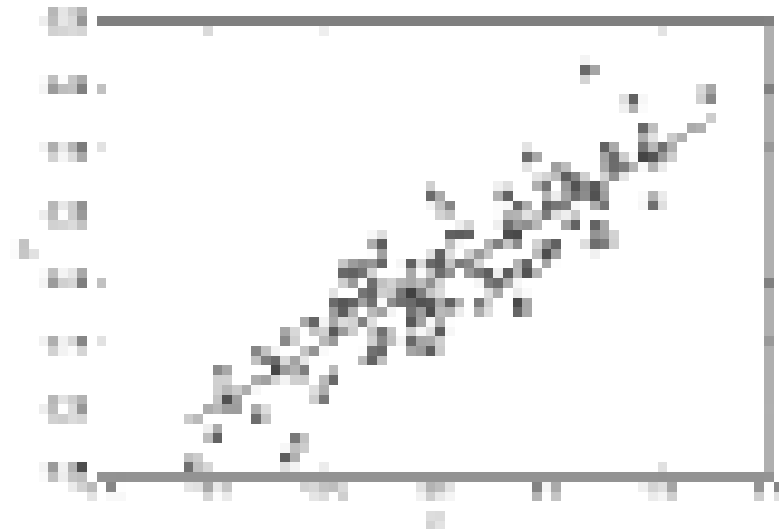
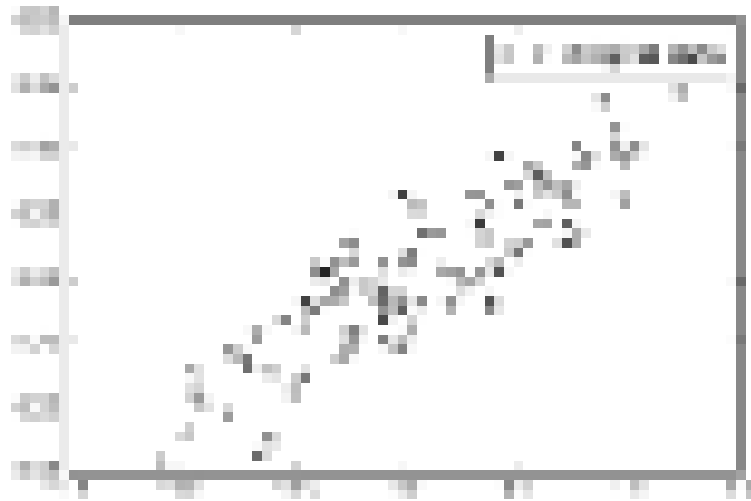
```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 my $a = 10;
7 my $b = 20;
8
9 my $result = $a * $b;
10
11 print "The result of $a * $b is: $result\n";
```

# Placeholders

- Allow exchanging data with your graph variables through "placeholders".
- They can be assigned when we ask the session to run

```
1 import tensorflow as tf
2
3 # Create a constant placeholder
4 a = tf.placeholder(tf.float32)
5 b = tf.placeholder(tf.float32)
6
7 # Create a constant placeholder
8 c = tf.constant(1.0)
9
10 # Create a constant placeholder
11 session = tf.Session()
12 # Run the graph with the placeholders
13 print session.run(a + b + c)
```

# Linear Regression



# Linear Regression

```
1 import numpy as np
2 import tensorflow as tf
3
4 # float parameters
5 w = tf.Variable(-1), tf.float32
6 b = tf.Variable(-3), tf.float32
7
8 # float input and output
9 x = tf.placeholder(tf.float32)
10 linear_model = w * x + b
11 y = tf.placeholder(tf.float32)
12
13 # loss
14 loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
```

# Linear Regression

```
1 # optimizer
2 optimizer = tf.train.GradientDescentOptimizer(0.01)
3 train = optimizer.minimize(loss)
4
5 # training loop
6 x_train = [1,2,3,4]
7 y_train = [0.1,1,2,3]
8
9 # training loop
10 init = tf.global_variables_initializer()
11 sess = tf.Session()
12 sess.run(init) # reset values to zeros
13 for i in range(1000):
14     sess.run(train, {x: x_train, y: y_train})
15     # calculate current accuracy
16     curr_A, curr_B, curr_loss = sess.run([A, B, loss], {x: x_train, y: y_train})
17     print('iter %d: acc: %f loss: %f" % (i, curr_A, curr_B, curr_loss))
```

# MNIST

- Classification of hand-written digits (0-9) from 28x28 pixel greyscale images (MNIST data set).
- Full data set of 70k examples: <http://yann.lecun.com/exdb/mnist>



# MNIST

- As common in machine learning, the MNIST data is split into three parts:
  - Training: 55,000 images
  - Test: 10,000 images
  - Validation: 5,000 images.
  - Dataset contains pair of images and labels.
  - Useful to test hyper parameters and generalization performance

The diagram illustrates the data split for MNIST. It shows a large rectangle representing the total dataset, which is divided into three horizontal sections. The top section is labeled 'Training' and contains 55,000 images. The middle section is labeled 'Test' and contains 10,000 images. The bottom section is labeled 'Validation' and contains 5,000 images. The total number of images is 70,000.

Category	Count
Training	55,000
Test	10,000
Validation	5,000



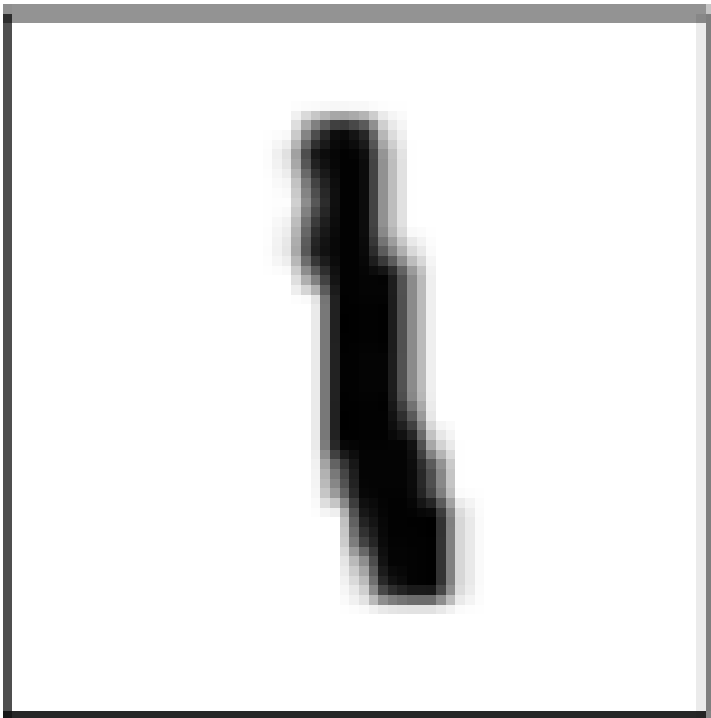
# MNIST

- As common in machine learning, the MNIST data is split into three parts:
  - Training: 55,000 images
  - Test: 10,000 images
  - Validation: 5,000 images.
  - Dataset contains pair of images and labels.
  - Useful to test hyper parameters and generalization performance

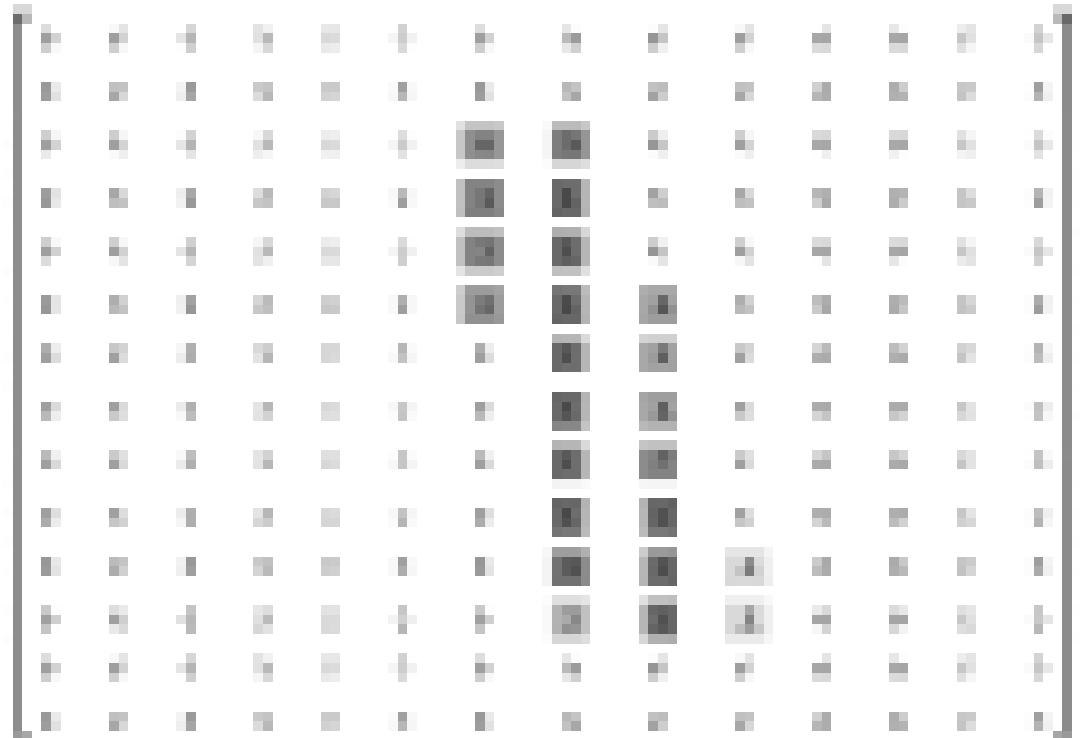


# MNIST

- Each image is 28 pixels by 28 pixels.
  - We can flatten this array into a vector of  $28 \times 28 = 784$  numbers.
  - Vector representation but losing structure.



→



# Import data

- Download and read the data automatically:

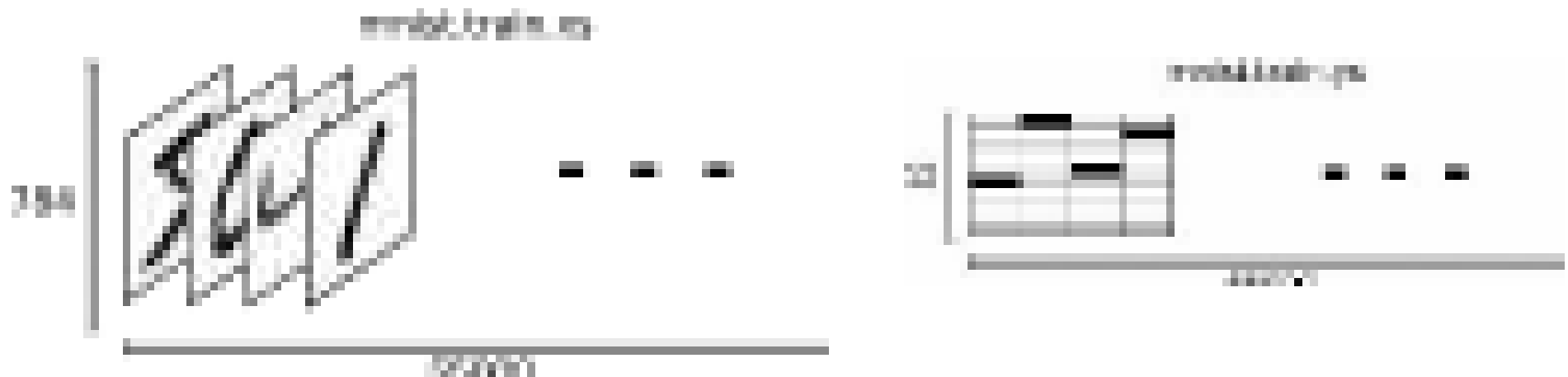


# Import data

- We get:

`mnist.train.images`: tensor with a shape of [55000, 784]

`mnist.train.labels`: a [55000, 10] array of floats - vector notation for class labels.

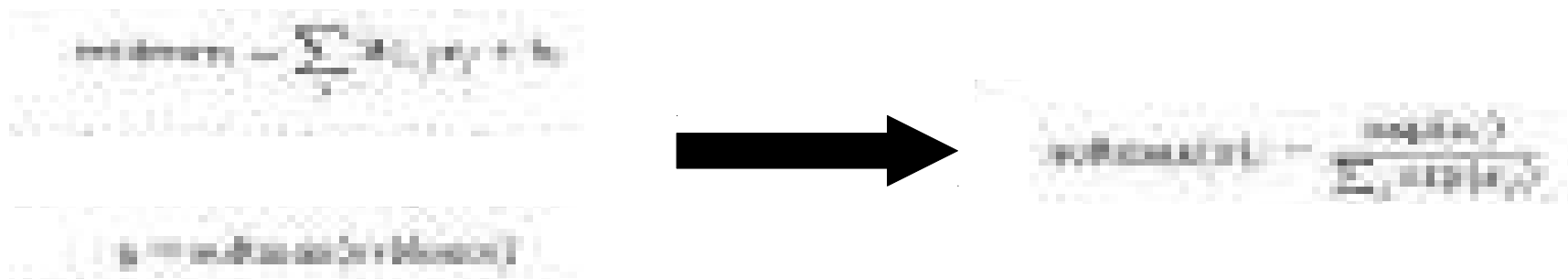


# NN training

- Several things to decide (data, hyperparameters):
  - Training data
    - Representation (vectors, images, text).
    - Normalization
  - Architecture
    - Layers: type, shape, number.
    - Activation functions
    - Output type (according to task, e.g. classification/regression) and loss function.
  - Learning algorithm
    - Initialization.
    - Update scheme.
    - Learning rate.
    - Momentum.
    - Regularization (weight decay, dropout).
    - Batch normalization
    - Stopping criteria

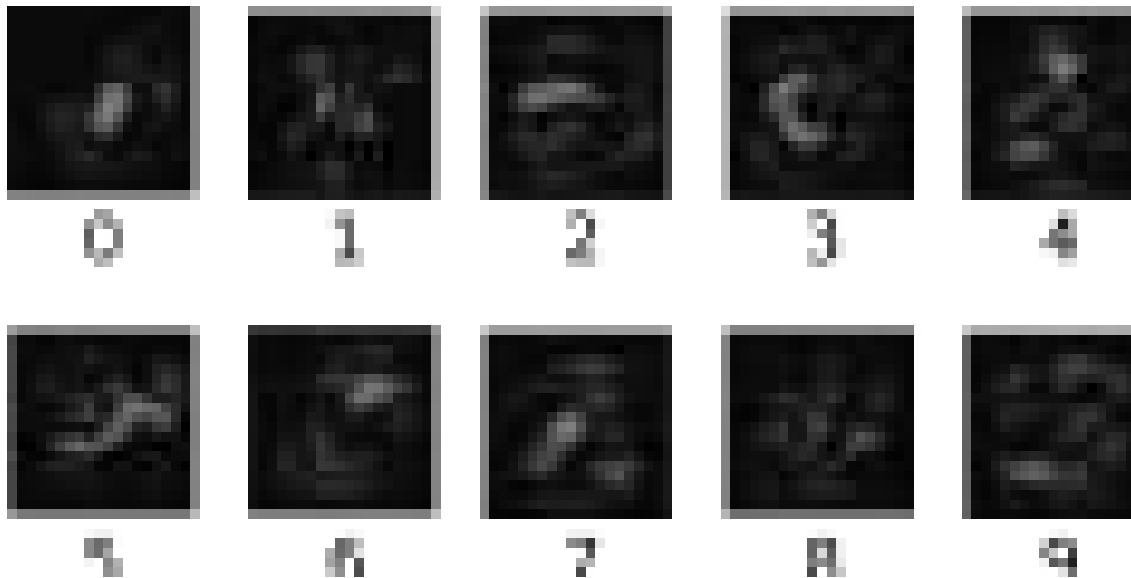
# Softmax regression

- Recap: softmax regression to output probabilities
- Two steps: add up the evidence of our input being in certain classes and then convert evidences into probabilities.



# Softmax regression

- Output: As we do a weighted sum of the pixel intensities we can inspect them.
- Red: negative weights.
- Blue: positive weights.



# Softmax regression

- Matrix Notation

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} = \text{softmax} \left( \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$



# MNIST

- We use variables and placeholders to create the model:
  - Look at the dimensionality
  - What is missing?

```
import tensorflow as tf
import numpy as np
import random

# Load MNIST data
mnist = tf.keras.datasets.mnist

# Split into training and testing data
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Reshape data to match the model's input shape (28x28x1)
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))


# Convert data to float32
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255

# Create the model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=10,
         validation_data=(test_images, test_labels))
```

# MNIST

- Model training:
  - Use cross-entropy 
  - Optimize with gradient descent with a learning rate 0.5.
  - Many other optimizers ([link](#))

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, y_train, x_test, y_test = x_train/255.0, y_train, x_test/255.0, y_test

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

# MNIST

- Run the session
  - Training considering mini-batches
  - Evaluate performance (are they good?)

```
import tensorflow as tf
import numpy as np
import random
import sys
import time

# Load MNIST data
mnist = tf.keras.datasets.mnist

# Split into training and testing data
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess the data
train_images = train_images / 255.0
test_images = test_images / 255.0

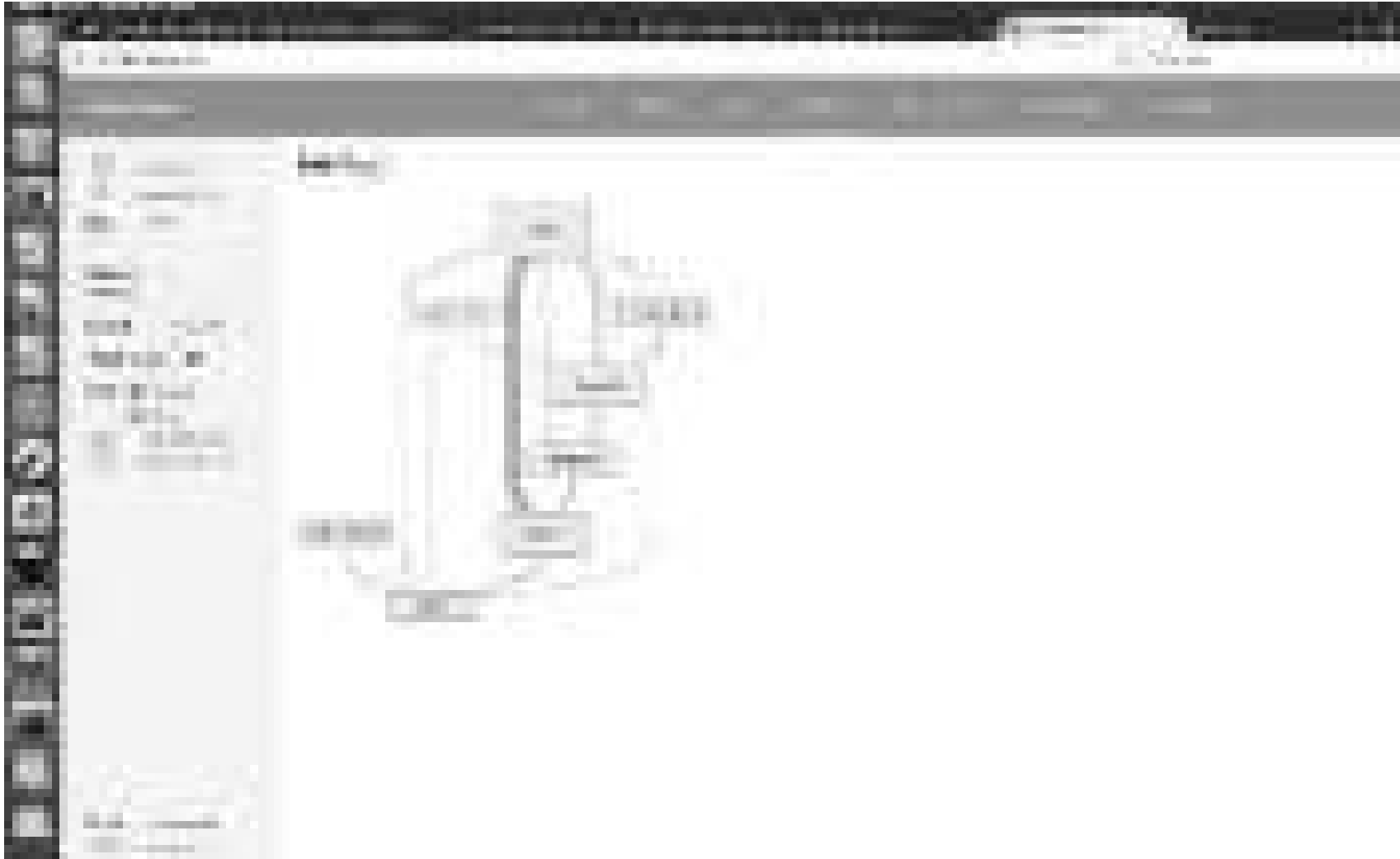
# Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=10,
         validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
print(f'Test loss: {test_loss}')
```

# TensorBoard



# TensorBoard

- Training a massive deep neural network can be complex and confusing.
- TensorBoard: visualization tools to facilitate models understanding and debug.
- Visualize graph, plot quantitative metrics about the execution of the graph, show additional data like images used, visualize statistics.

# TensorBoard

- Modify code to generate summary data.
  - (1) Create graph and decide which nodes you would like to collect summary data.

Example MNIST:

- Monitor learning rate and loss.
- Use `tf.summary.scalar` for to the nodes that output the learning rate and loss respectively.

# TensorBoard

- Modify code to generate summary data.
  - (1) Create graph and decide which nodes you would like to collect summary data.

Example MNIST:

- Visualize the distributions of activations coming off a particular layer, or the distribution of gradients or weights.
- Use `tf.summary.histogram`.

# TensorBoard

- Modify code to generate summary data.
  - (1) Create graph and decide which nodes you would like to collect summary data.

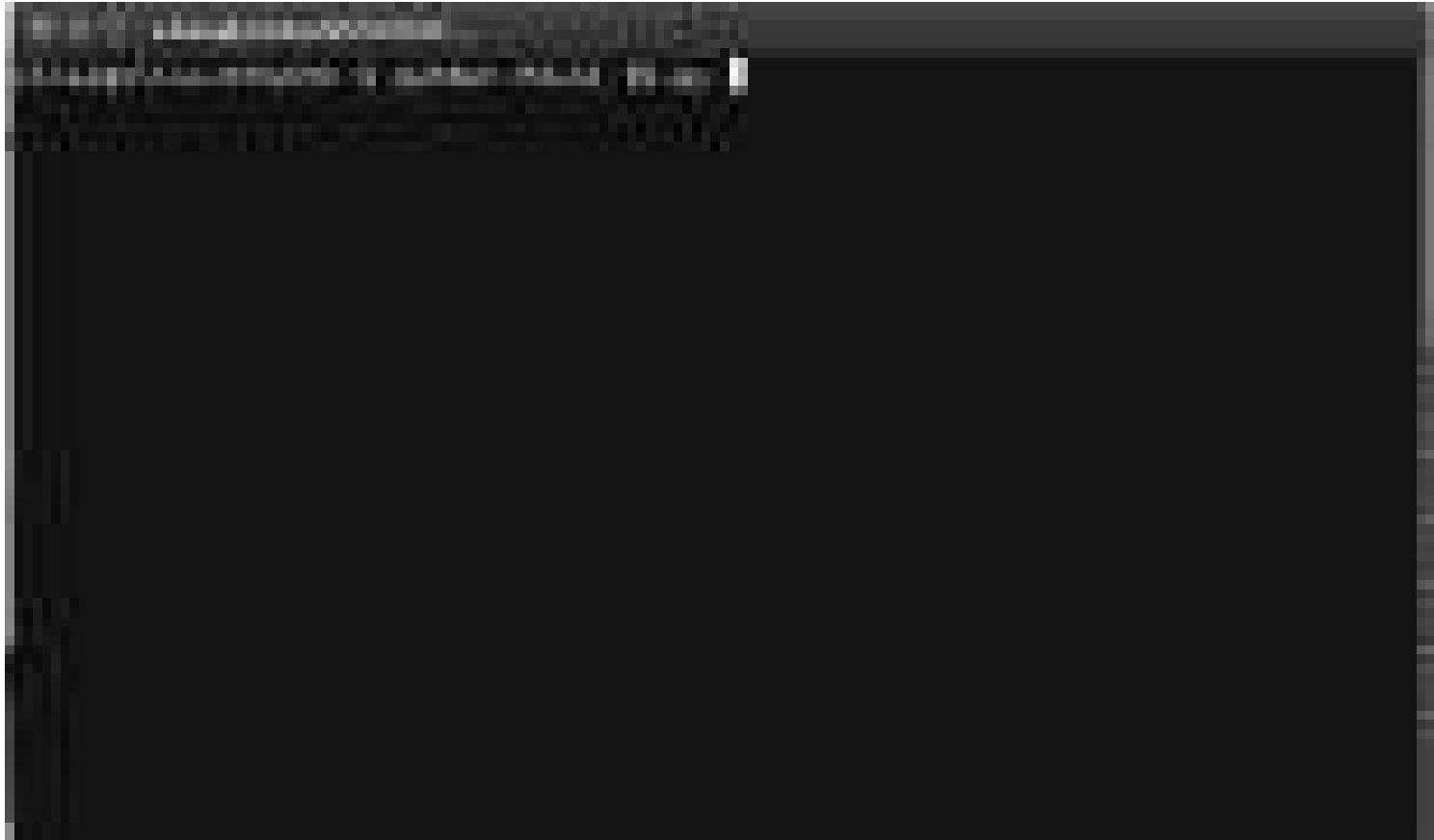
The summary nodes are peripheral nodes added to the graph: none of the ops we are currently running depend on them.



# TensorBoard

- Modify code to generate summary data.
  - (2) To generate summaries, run all of the summary nodes.
    - (2a) Use `tf.summary.merge_all` to combine them.
    - (2b) Run the merged summary op, which will generate a serialized Summary protobuf object with all of your summary data at a given step.
  - (5) Write summary data to disk, pass the summary protobuf to a `tf.summary.FileWriter`.

# TensorBoard

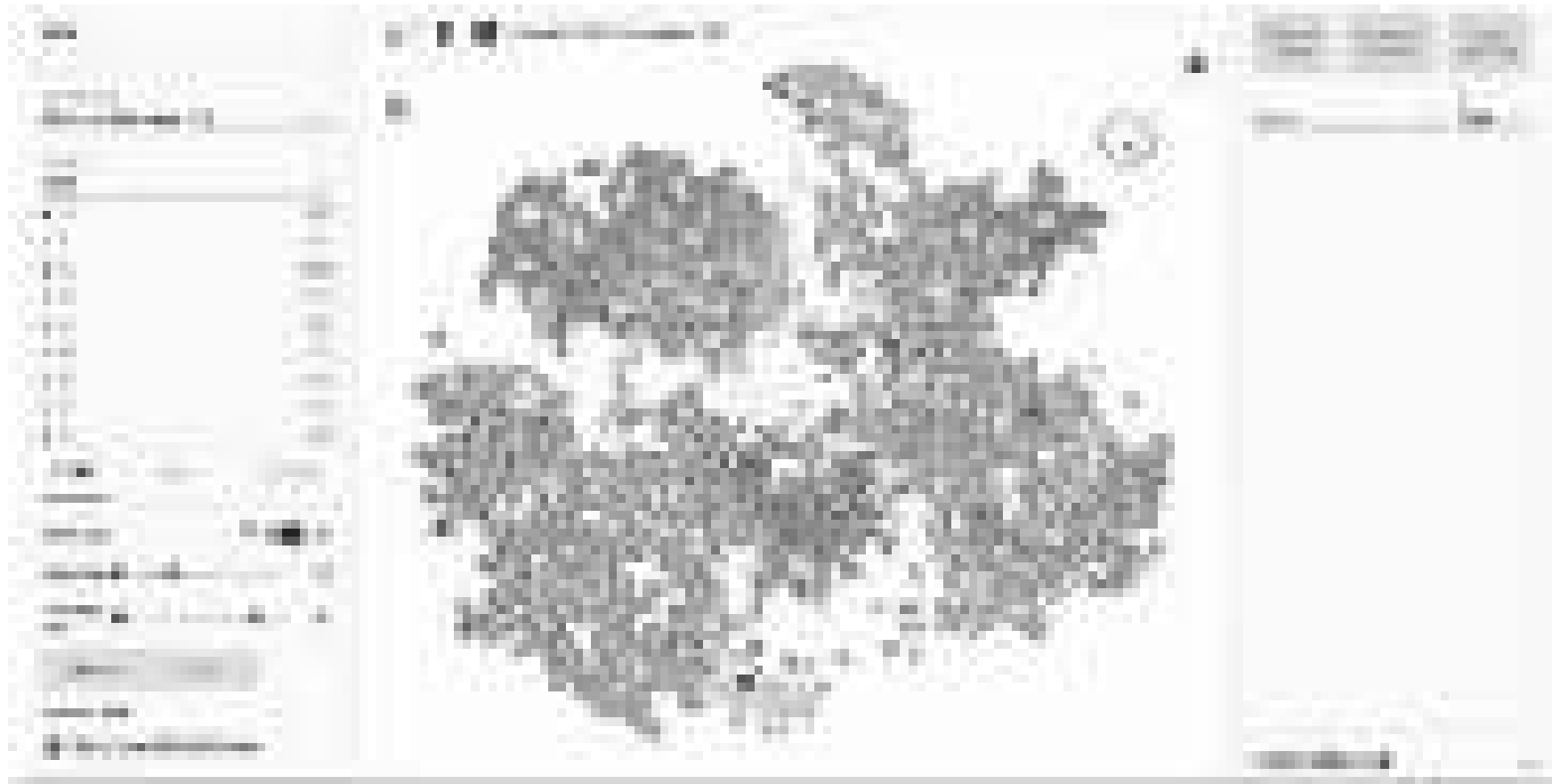


# TensorBoard



# TensorBoard

- Other features: Embedding visualization.





**Keras**

# Keras

- Keras (κέρας) means *horn* in Greek.
- In the *Odyssey* it is mentioned that dream spirits are divided between:
  - those who deceive men with false visions, who arrive to Earth through a gate of ivory
  - those who announce a future that will come to pass, who arrive through a gate of horn.



# Keras

- Easy-to-use Python library
- Why Python? Easy to learn, powerful libraries (scikit-learn, matplotlib...)
- It wraps Theano and TensorFlow (it benefits from the advantages of both)
- Guiding principles: modularity, minimalism, extensibility.

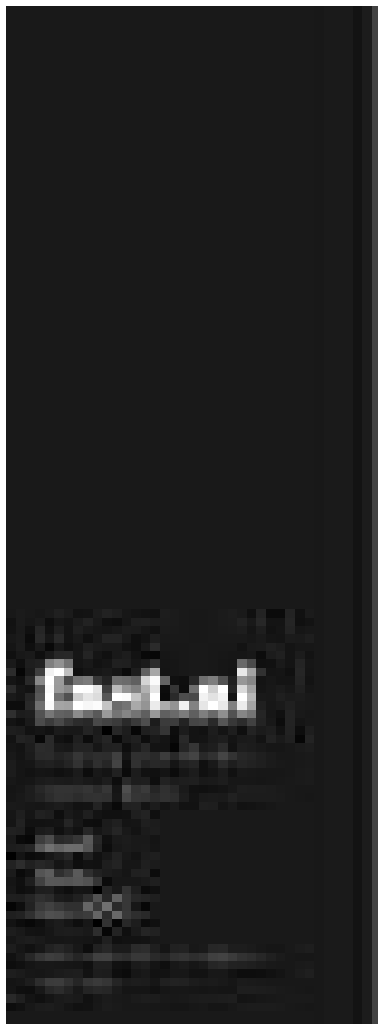
# Keras

- Use both GPU and CPUs
- Easy to use both convolutional networks and recurrent networks and combinations of the two.
- Supports arbitrary connectivity schemes (including multi-input and multi-output training)
- Many easy-to-use tools: real-time data augmentation, callbacks (Tensorboard visualization)



# Keras

- Keras gained official Google support



The deep learning framework Keras has been adopted as the official deep learning framework for TensorFlow and Microsoft's Cognitive Toolkit (CNTK).

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed by François Fleuret, Armin Schuster, and others at the EPFL.

Keras is designed to be simple and easy to use, while still being powerful enough to handle complex tasks. It provides a clean, minimalist API that makes it easy to experiment with different architectures and hyperparameters.

Keras is also designed to be modular and extensible, allowing users to easily integrate their own custom layers and models. This makes it a great choice for researchers and practitioners alike.

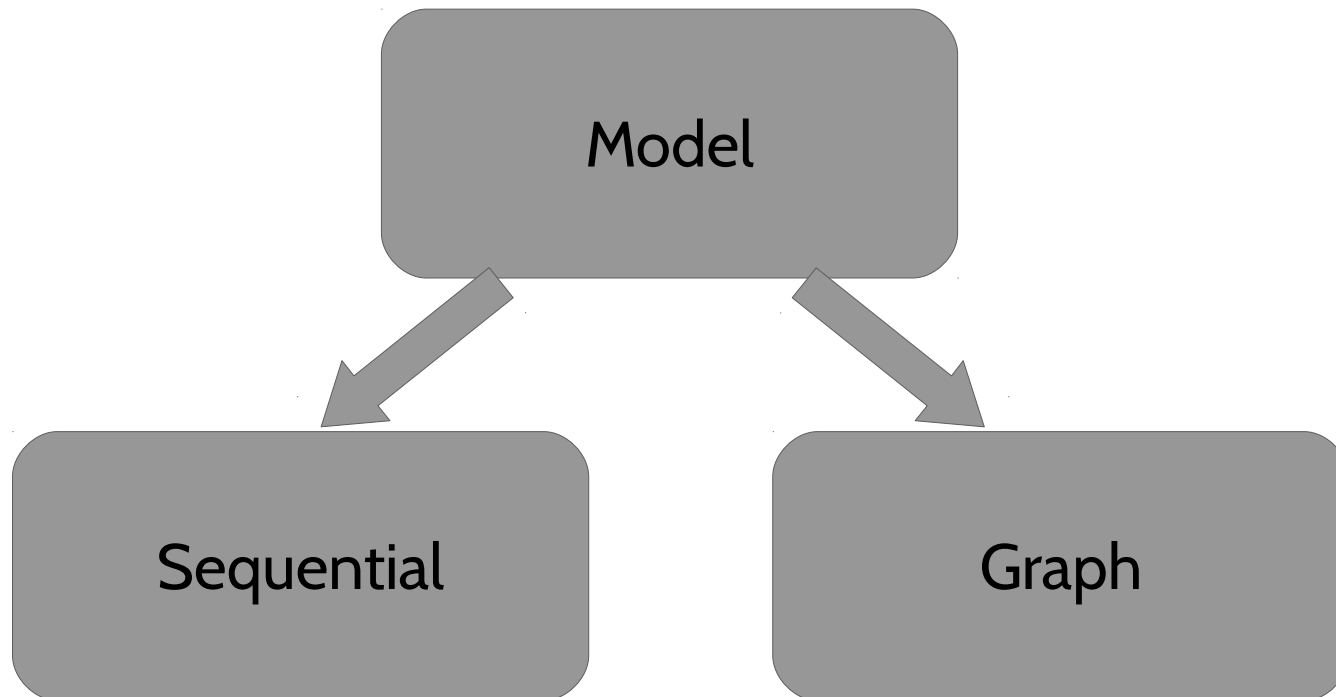
Keras is now officially supported by Google and Microsoft, which is a significant milestone for the framework. This support will help to ensure that Keras remains a leading choice for deep learning applications.

# Keras

- Weaknesses:
  - Less flexible
  - Some stuff not there yet (no RBM for example)
  - Less projects available online (e.g. with respect to Caffe)

# Model

- A model is a *sequence* or a *graph* of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible.



# Modularity

- A model is a *sequence* or a *graph* of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible.
- Modules:
  - neural layers
  - cost functions
  - optimizers
  - initialization schemes
  - activation functions
  - regularization schemes
  - *your own module*

# Keras

- Extensibility: modules are easy to add.
- Simplicity: modules should be made extremely simple.

TensorFlow:

```
graph TD
    subgraph Input
        direction TB
        I1[Input 1]
        I2[Input 2]
    end
    I1 --> L1[Layer 1]
    I2 --> L1
    L1 --> L2[Layer 2]
    L2 --> L3[Layer 3]
    L3 --> O[Output]
```

Keras:

```
graph TD
    subgraph Input
        direction TB
        I1[Input 1]
        I2[Input 2]
    end
    I1 --> L1[Layer 1]
    I2 --> L1
    L1 --> L2[Layer 2]
    L2 --> L3[Layer 3]
    L3 --> O[Output]
```

# Install Keras

- Extremely easy:

```
>> source tensorflow/bin/activate
```

```
>> python
```

```
>> pip install keras
```

```
>> import keras as k
```



# Graph model

- Useful to create two or more independent networks to diverge or merge
- Useful to create multiple separate inputs or outputs
- Different merging layers (sum or concatenate)





# Let's run MNIST again

- Homepage

<https://keras.io/>

<https://keras.io/getting-started/sequential-model-guide/#getting-started-with-the-keras-sequential-model>

- Examples:

<https://github.com/fchollet/keras/tree/master/examples>

- Let's compare a MLP and a CNN...

# Questions?

