MASTER OF SCIENCE IN ENGINEERING

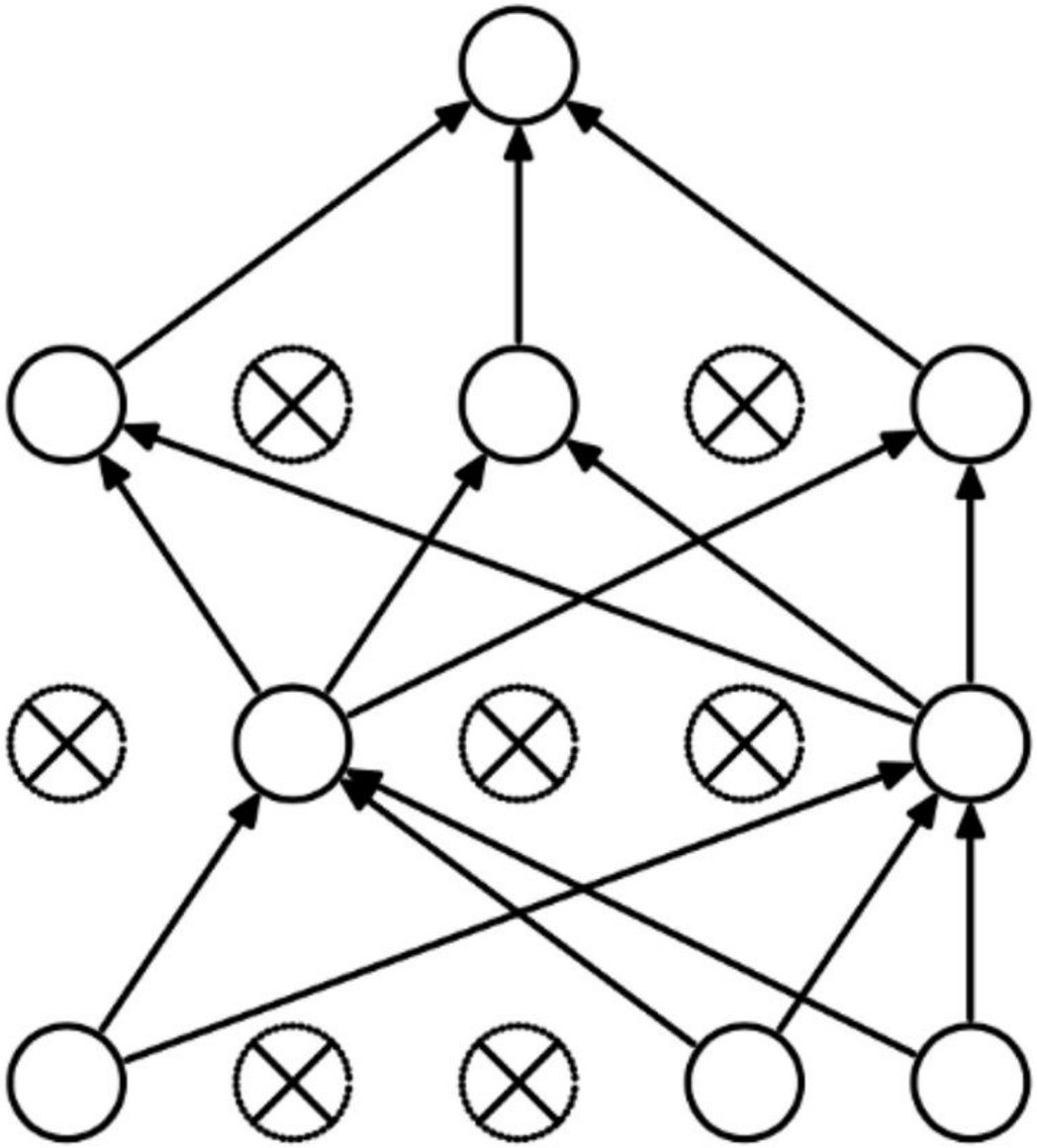
Regularisation and DL Frameworks

TSM_DeLearn

Andreas Fischer
Klaus Zahn

We are grateful to J. Hennebert and M. Melchior, that they provided their slides

Regularisation



Regularisation as a Means to Avoid Overfitting

- Problem:
  - DNNs have a large number of parameters: ~10k - 1M.
  - Risk of overfitting!

- Regularisation comes to the rescue!

- Goal: Learn the most popular regularisation techniques.

Regularisation Methods (Overview)

- **Weight Penalty**
  Constraints on parameters (e.g. length of parameter vector, number of parameters) to give preference to simple models.

- **Dropout**
  Randomly drop (neutralise) neurons during training steps to make the solution less dependent on individual neurons.

- **Early Stopping**
  Stop training at the minimum of the cost function on the validation set.

- **Data Augmentation**
  Generate more training data with additional characteristics (e.g. symmetries) the solution should have.

covered with CNNs part

Weight Penalty

The loss function is modified to give preference to smaller or fewer weights — by adding a suitable penalty term.

$$\mathbf{J} = \mathbf{J}_0 + \lambda \cdot \Omega(\mathbf{W}) \quad (\lambda \geq 0)$$

Original loss function (e.g. RMS loss)

regularisation parameter

Penalty term that favours models with smaller (fewer) weights.
Not a function of the activations of the network.

Two forms of penalties are popular:

$$L_1\text{-Regularisation: } \Omega(\mathbf{W}) = \|\mathbf{W}\|_1 = \sum_{l,k,j} \left| W_{kj}^{[l]} \right|$$

$$L_2\text{-Regularisation: } \Omega(\mathbf{W}) = \|\mathbf{W}\|_2^2 = \sum_{l,k,j} \left| W_{kj}^{[l]} \right|^2$$

For optimisation with gradient descent, the derivatives of the loss are modified by the penalty term in a direction to make the loss and the penalty term smaller.

Gradient Descent with L^2-Regularisation

Gradient for the regularised loss function:

$$\nabla\left(J_0(\mathbf{W}) + \frac{\lambda}{2}\|\mathbf{W}\|^2\right) = \nabla J_0(\mathbf{W}) + \lambda\mathbf{W}$$

Update rule for gradient descent with L^2 regularised loss:

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} - \alpha\nabla J(\mathbf{W}) \\ &\leftarrow \mathbf{W} - \alpha\left(\nabla J_0(\mathbf{W}) + \lambda\mathbf{W}\right) \\ &\leftarrow \underbrace{(1-\alpha\lambda)}_{\text{reduction in length}}\mathbf{W} - \alpha\underbrace{\nabla J_0(\mathbf{W})}_{\text{original loss term}}\end{aligned}$$

Learning rule modified to multiplicatively shrink the weight vector by a constant factor before performing the usual gradient update.

Gradient Descent with L^1-Regularisation

Gradient for the regularised loss function:

$$\nabla \left(J_0(\mathbf{W}) + \lambda \|\mathbf{W}\|_1 \right) = \nabla J_0(\mathbf{W}) + \lambda \ \mathrm{sign}(\mathbf{W})$$

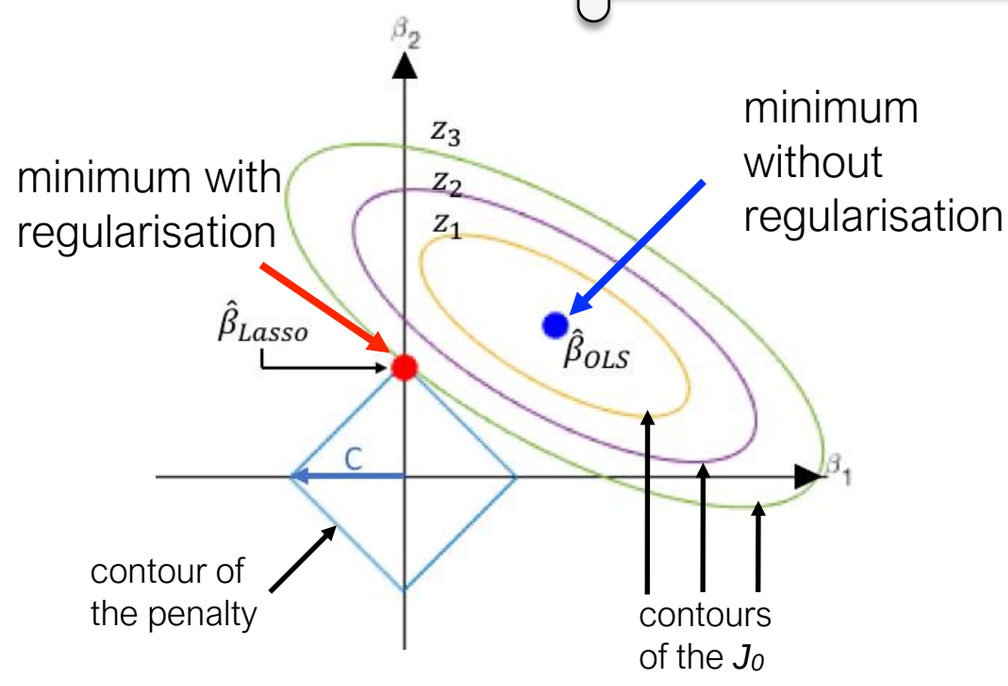
element-wise application of sign-function

Effect on the update rule as easy to see as for L^2 regularisation.

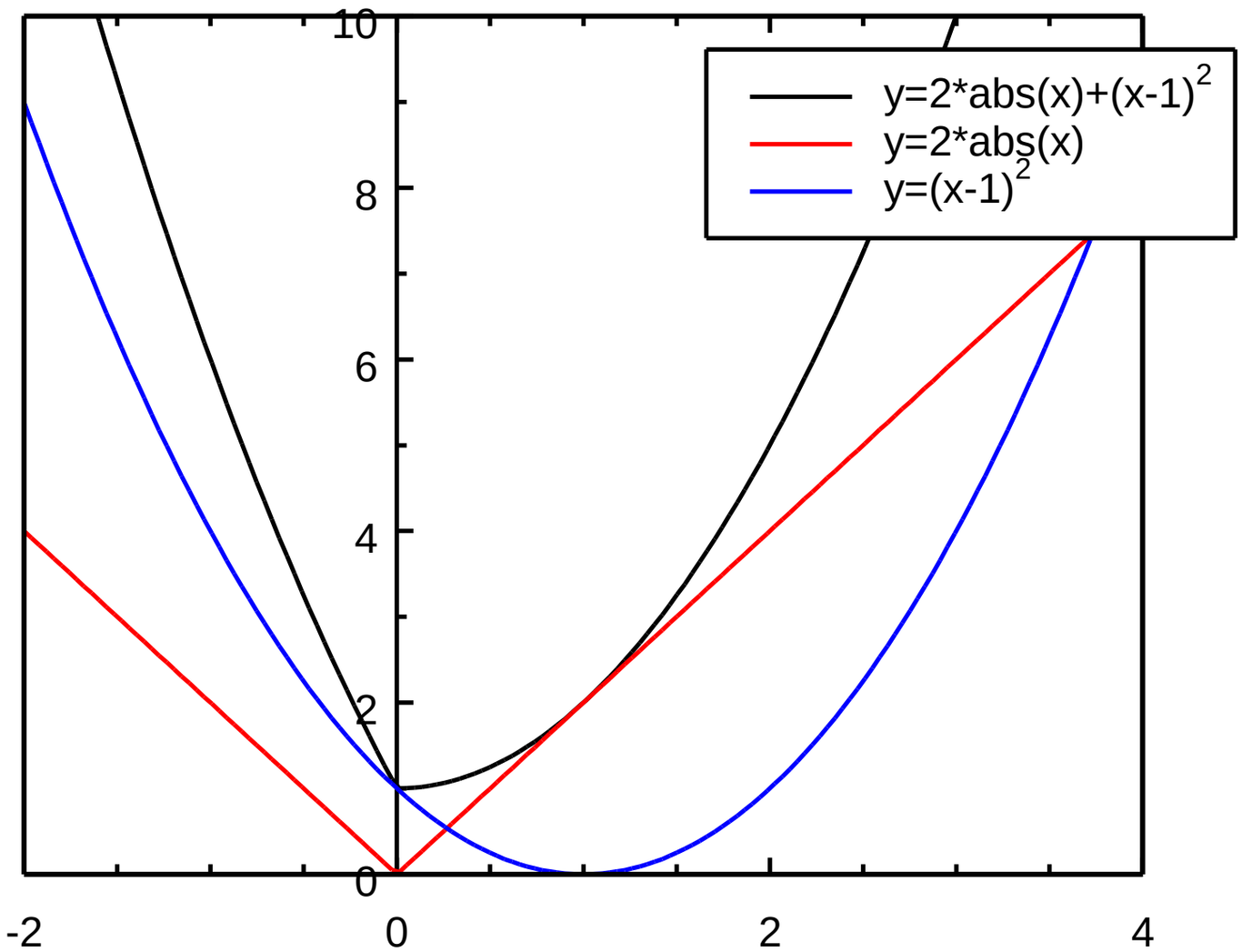
L^1-regularisation leads to sparser solutions than L^2 regularisation.

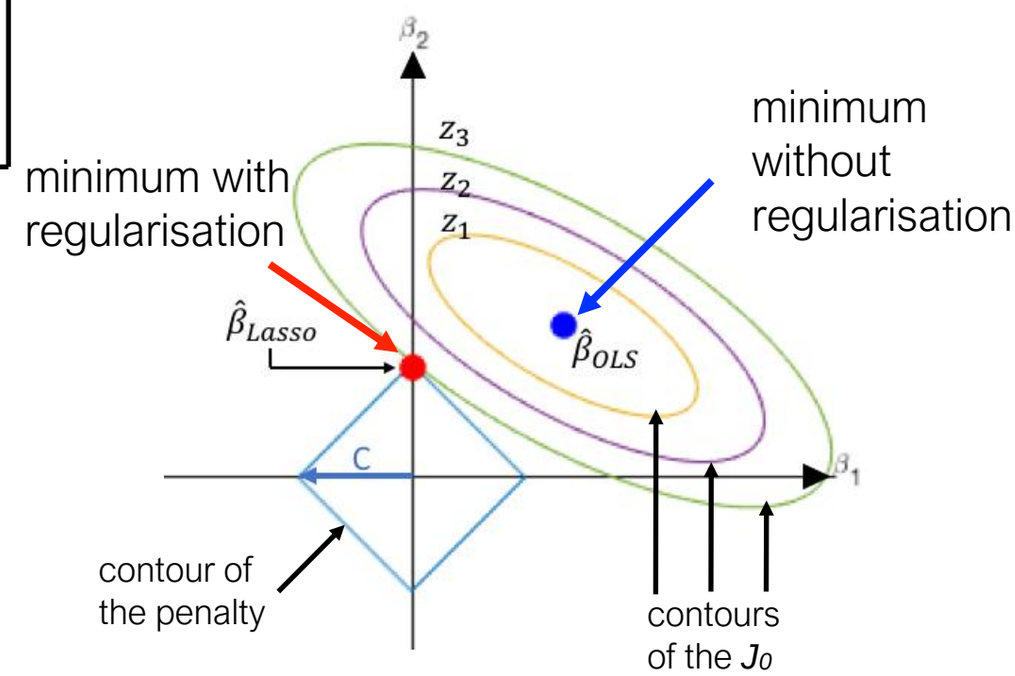
Sparsity means that at the optimum some parameters are zeroed out.

For linear regression problems the addition of the L^1 penalty term is also referred to as LASSO regression.



Gradient Descent with L^1-Regularisation

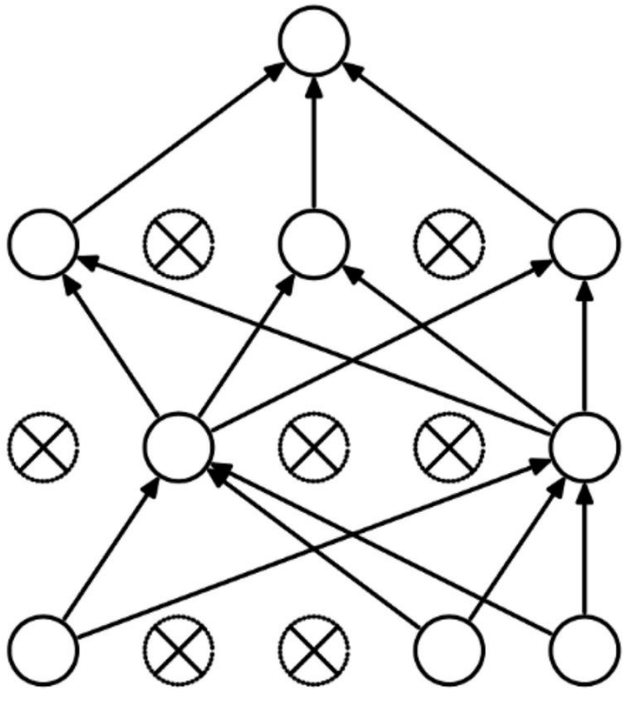




Dropout

- Most popular regularisation technique for deep neural networks.
- Highly successful: Even state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout.
- Proposed by G. E. Hinton in 2012 and further detailed in a paper by Nitish Srivastava et al.
- Algorithm:
  - At each training step, each neuron (incl. input neurons, excl. output neurons) has a probability ***p*** (“dropout rate”) of being ignored during this step —> activations masked.
  - Dropout rate typically set to 50% for hidden, and 20% for input units.
  - For testing or in production, neurons don’t get dropped, but weights or outputs are corrected (since weights trained including the dropout rate).

(1) G. Hinton et al., “Improving neural networks by preventing co-adaptation of feature detectors,”(2012)
(2) N. Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” (2014).

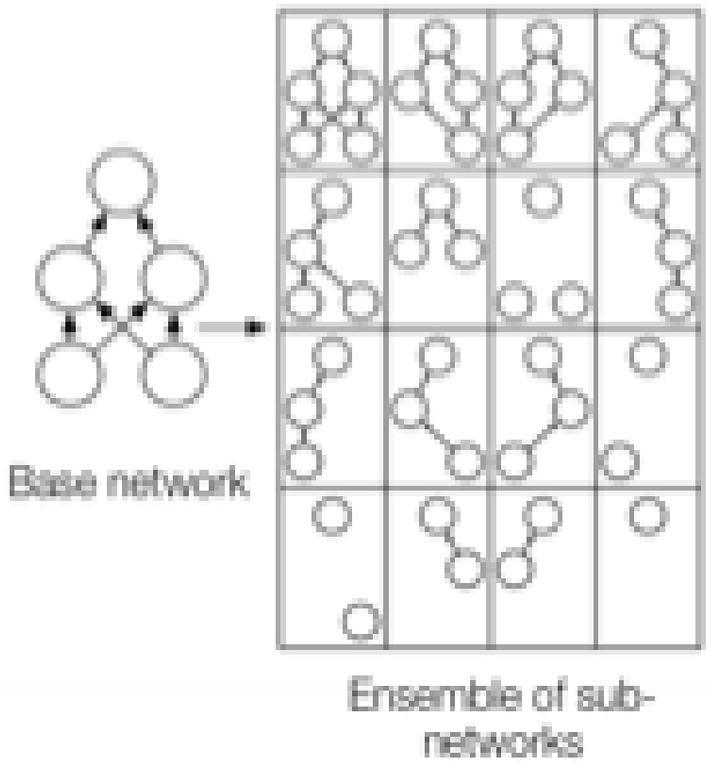


Intuition on Why Dropout Works

- Implication of using dropout:
  - Simpler models with less units are considered during training.
  - Models learn to be less dependent on single units and units cannot co-adapt with their neighbouring units; they have to be as useful as possible on their own.
  - A more robust network is obtained that generalises better.

From Geron, “Hands-On Machine Learning with Scikit-Learn and TensorFlow”:
“Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn’t make much of a difference. “

- Related to ensemble methods:
  Trains an ensemble of sub-networks derived from a base network by removing (non-output) units.



Implementation of Dropout

- Computationally very cheap:
  - Requires only drawing for each unit a random binary number: O(n) computations per update step (mini-batch)
  - O (n) additional memory to store these binary numbers for the backprop.
  - No additional cost at test time or in production.

- Very versatile:
  - Does not significantly limit the type of model or training procedure that can be used. (applicable e.g. to MLP, CNN or RNNs or other optimisers).
  - Can be combined with other regularisation techniques.

- Reduced representational capacity:
  - As a regularisation technique, dropout reduces the effective capacity of a model. Possibly, capacity of model needs to be increased. Training of these larger models have an impact on performance.
  - For very large datasets, regularisation implies little reduction in generalisation error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularisation.

Early Stopping

- Stop training when validation error begins to increase while training error still decreases.

Typically, can only be observed when training large models with sufficient representational capacity so that overfitting is possible.

- Algorithm
  - Run optimisation algorithm to train the model - simultaneously compute the validation set error.
  - Store a copy of the model parameters as long as the validation set error improves.
  - Iterate until validation set error stops improving (e.g. has not improved for k steps).
  - Return the parameters where the smallest validation set error is observed.

Geoffrey Hinton called it a "**beautiful free lunch**."

Why is Early Stopping a Regularisation Method?

- Training time (number of training steps/epochs) can be considered as hyper-parameter.

- It controls the *effective capacity* of the model by determining the number of steps it can take to fit the training set.
  - Restricts the volume in parameter space to be searched - it can only move a limited number of steps T away from the initial parameters. .
  - Equivalent to L^2-regularisation in the case of a linear model with a quadratic error function.(*)

- Efficient, non-intrusive search for this hyper-parameter:
  - The cost relates to repeated calculations of the validation set error and keeping a copy of the recent model parameters. The first can easily be parallelised.
  - Almost no change to the code needed.

- Easy to combine with other regularisation techniques.
  - Typically, the best generalisation does not occur at a local minimum of the training objective.

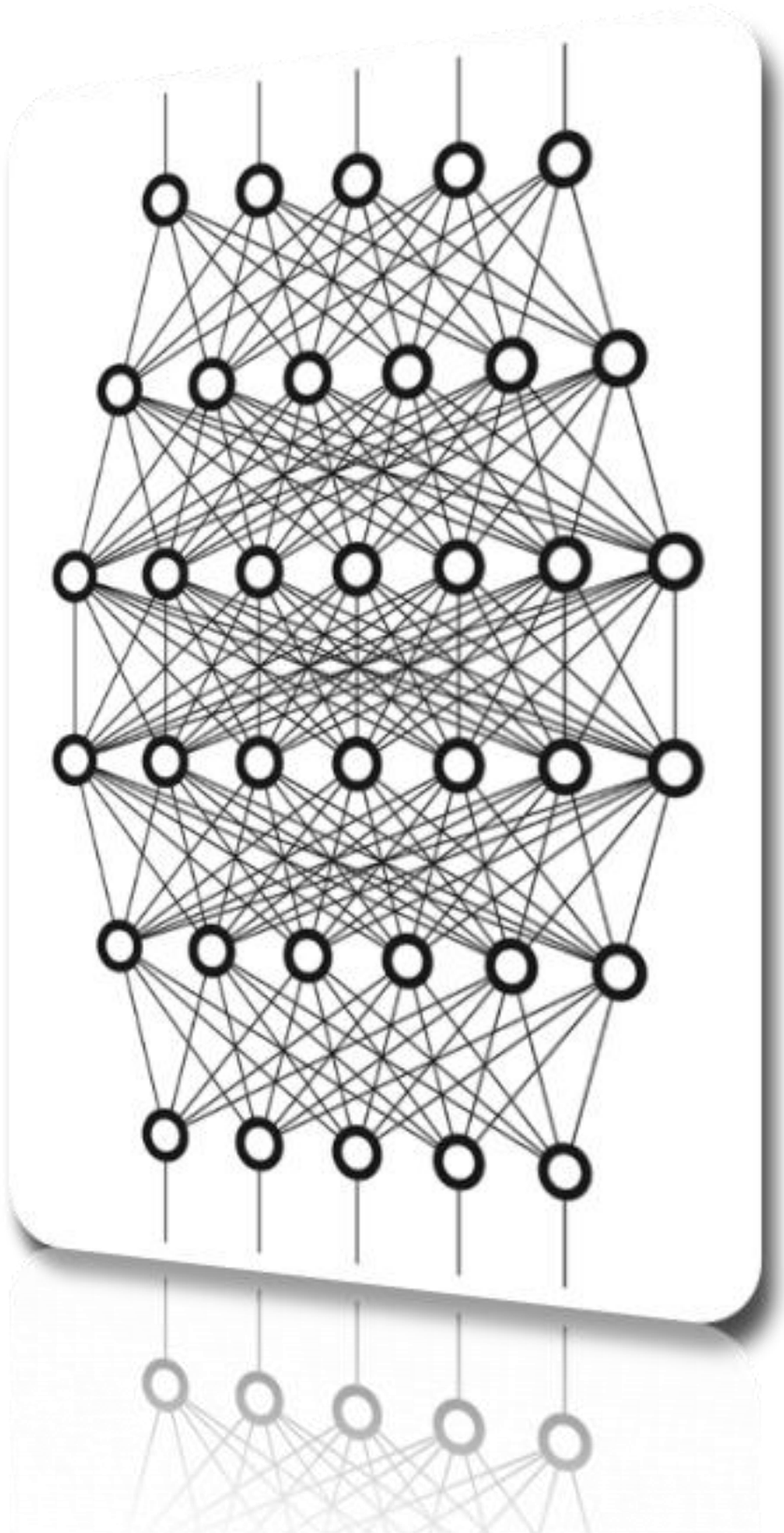
(*) See Goodfellow et al.,“Deep Learning”.

Deep Learning Frameworks

Plan

1. Recap Computational graph
2. Computational graphs implementation
3. Deep Learning Frameworks
    1. Overview of the “zoo”
    2. Tensorflow overview
4. Keras Sequential API

Computational Graph Implementations



Recaps

Example

Implementation strategies

This Section inspired from CS231 Stanford class
https://youtu.be/d14TUNcbn1k

Recap 1 - Gradient descent

- Update any parameters of your model in the opposite direction of the gradient of the loss w.r.t. weights

$$param \leftarrow param - \alpha \frac{\partial J}{\partial param}$$

In practice we have stopping criteria, e.g. epoch < max_epochs or loss_gain < ε

1 sample = stochastic gradient descent
B samples = mini-batch gradient descent
N samples = (full) batch gradient descent

Current weighs

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

The loss function usually includes a **performance** term and a **regularisation** term. Regularisation will be seen this week.

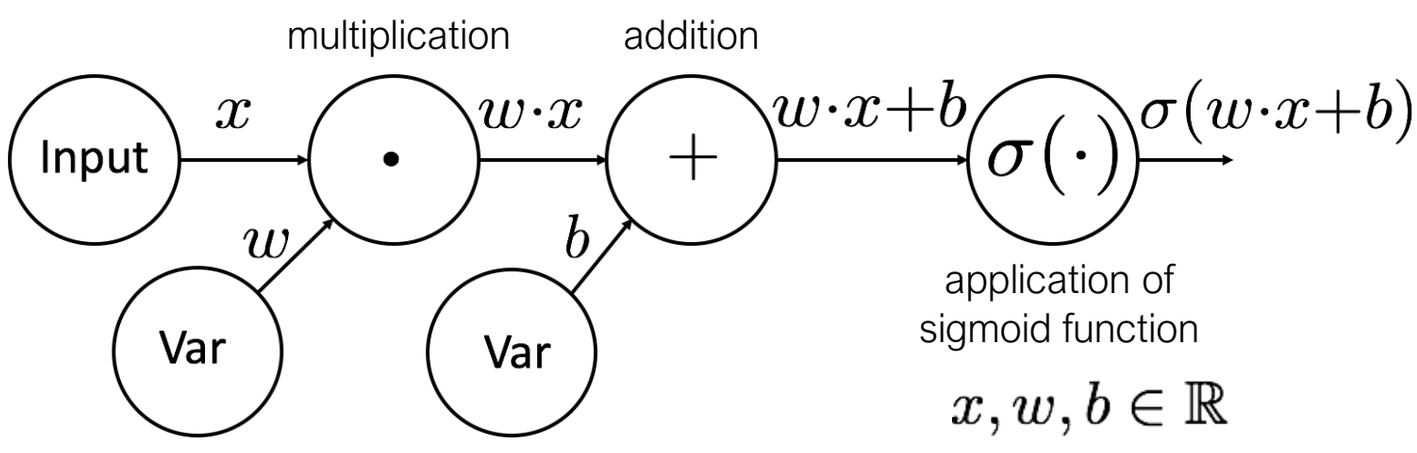
Recap 2 - Computational Graphs

A computational graph is a directed graph where

- *nodes* correspond to operations or input variables
- *edges* correspond to inputs of an operation which can originate from input variables or outputs of other operations.

Two types of input variables: Input data and model parameters.

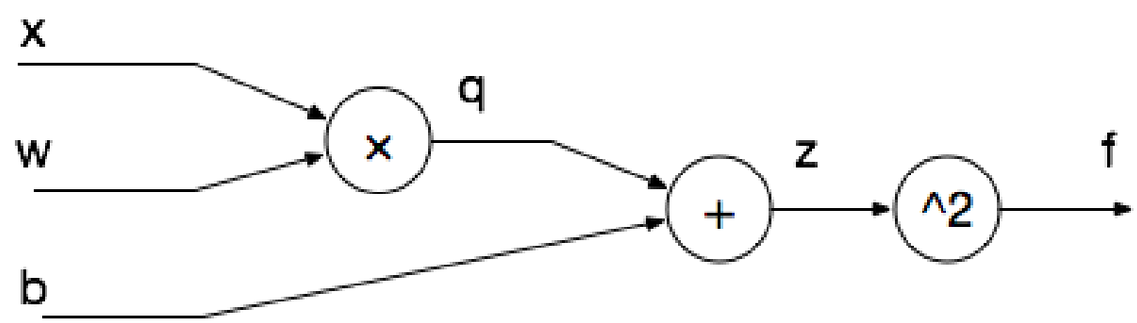
Example $$g(x; w, b) = \sigma(w \cdot x + b)$$



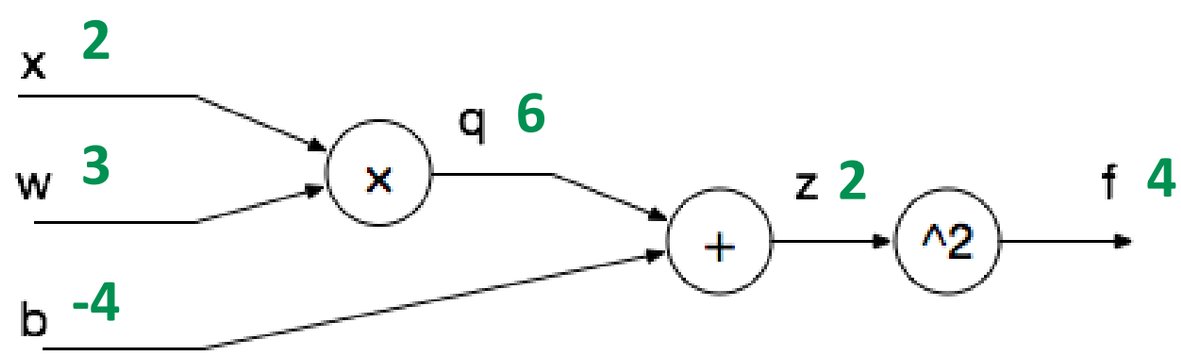
Complexity of operations defined for the nodes not precisely defined and allowing for a wide range of possibilities: Simple addition, sigmoid function, MLP layer, etc.

Recap 3 - A simple example

$$f = (wx + b)^2$$



- The input can be propagated ***forward*** through the graph
- E.g. x=2, w=3, b=-4



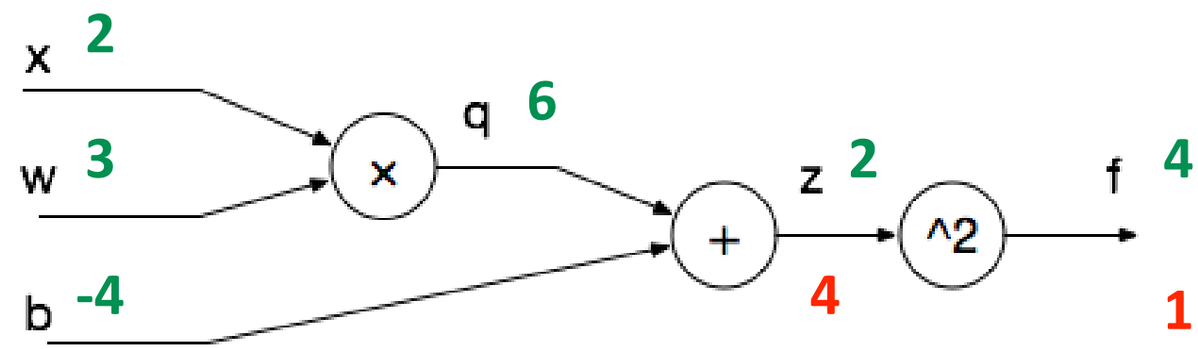
Recap 3 - A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



4×1

$\frac{\partial f}{\partial z} \frac{\partial f}{\partial f}$

$\frac{\partial f}{\partial f}$

- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

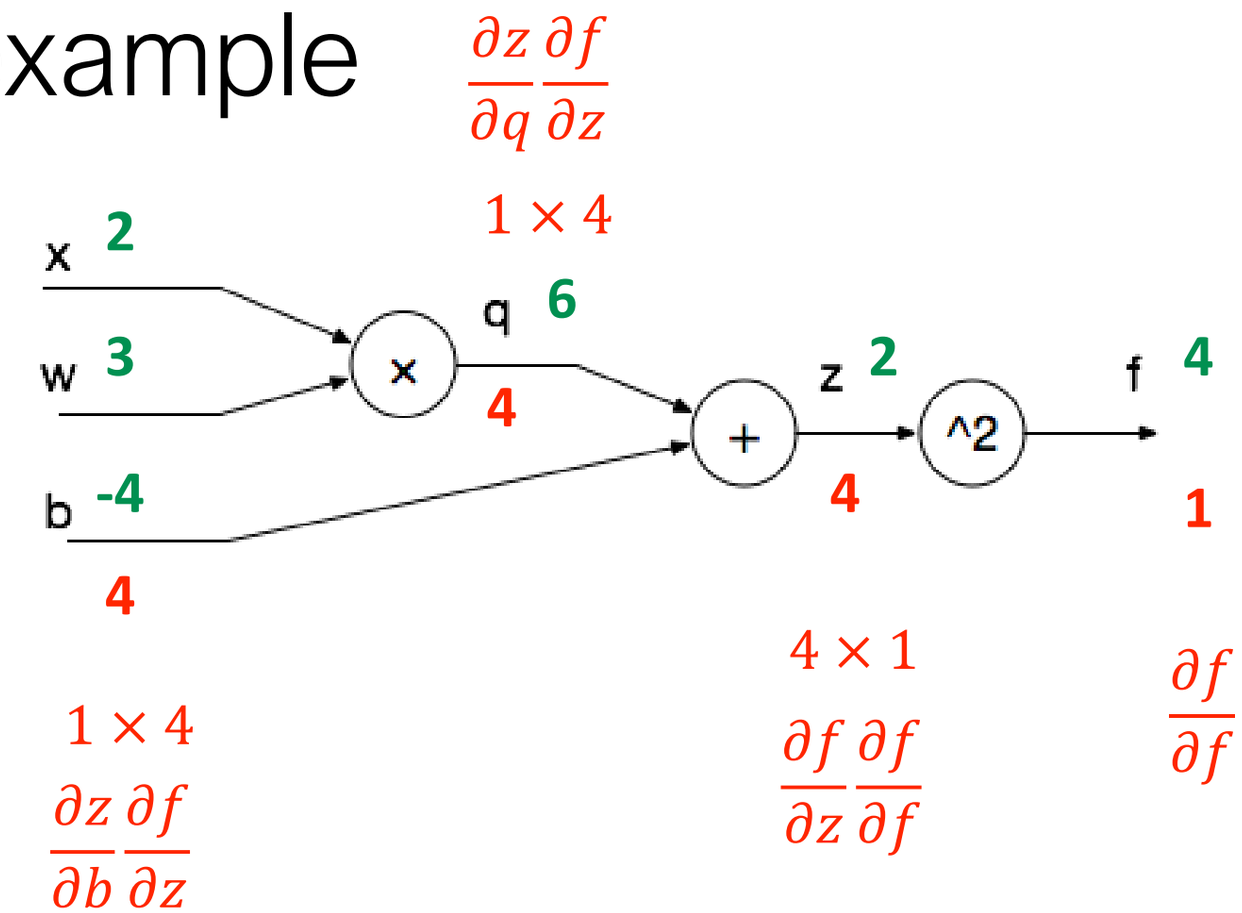
Recap 3 - A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

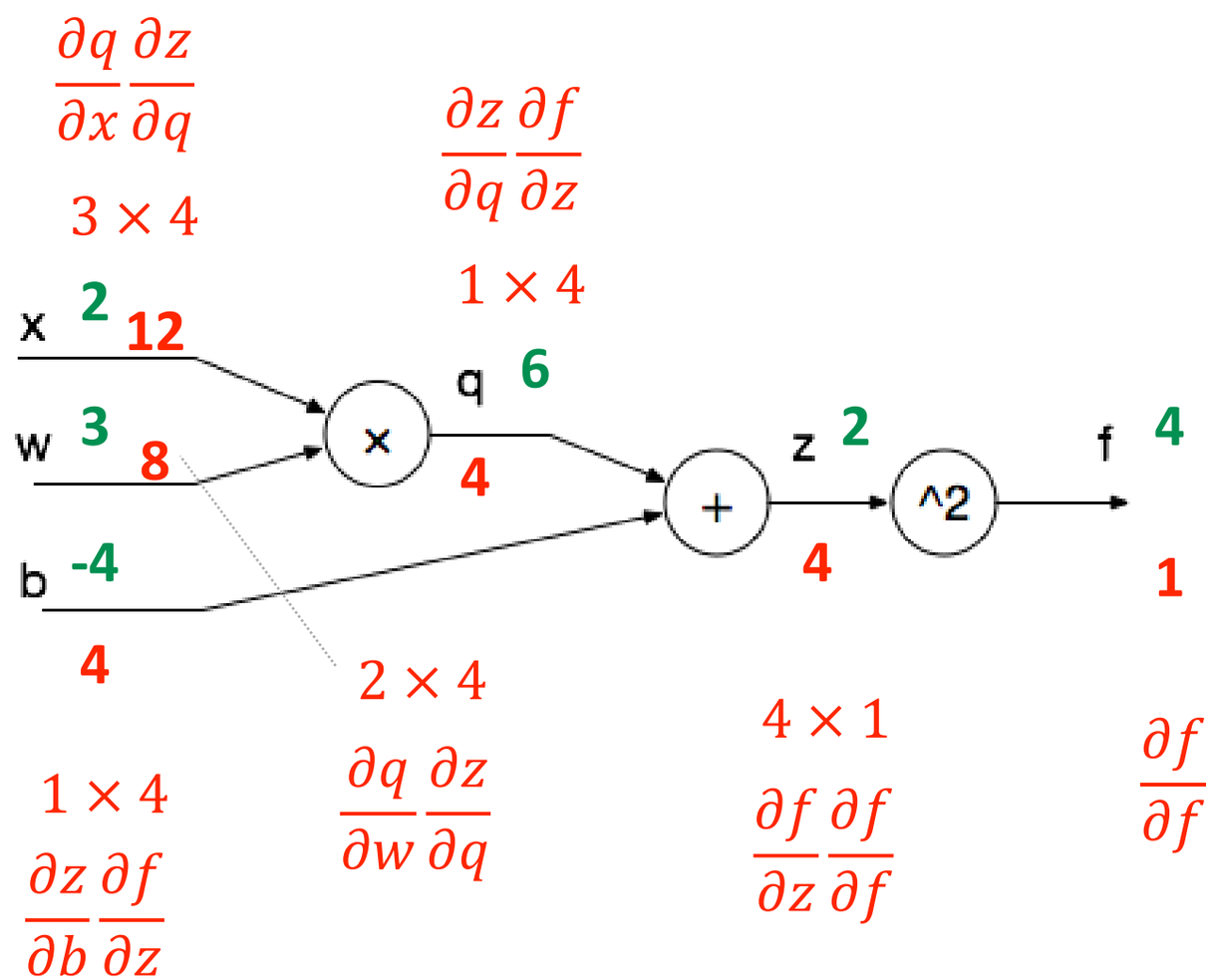
Recap 3 - A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

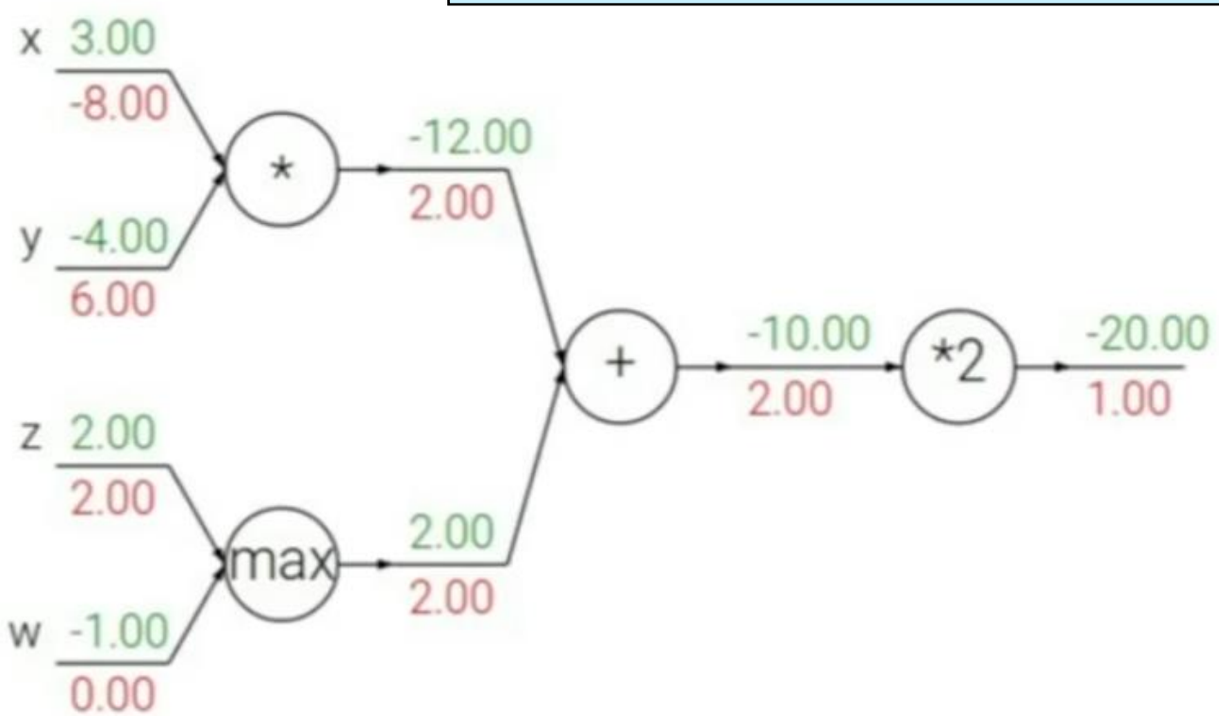
Patterns in backward flow

Advantage 1: Intuitive interpretation of gradient backpropagation

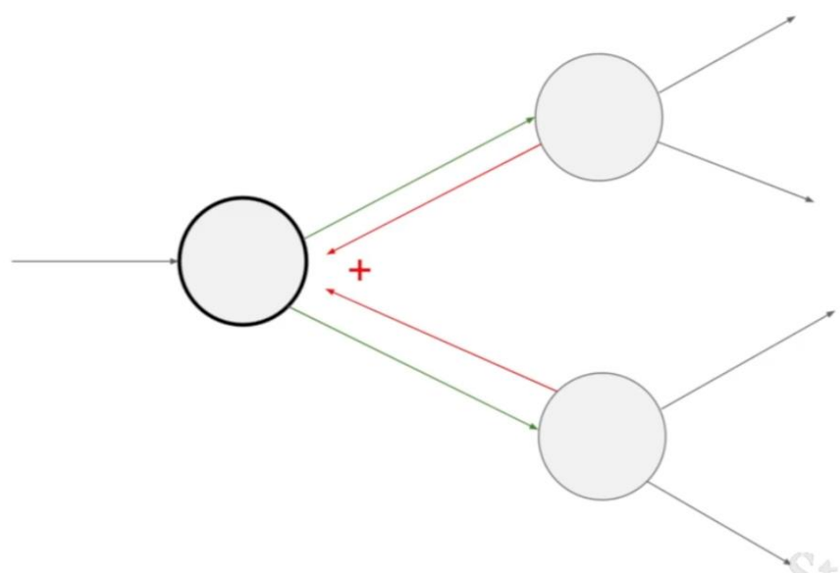
add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



gradients add at branches

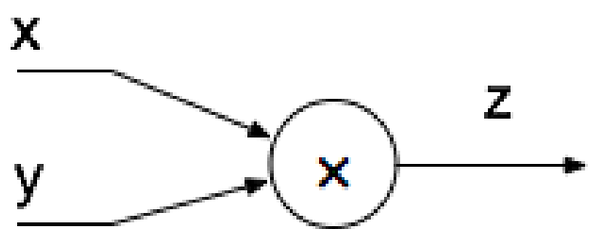


From Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

Modularized implementation

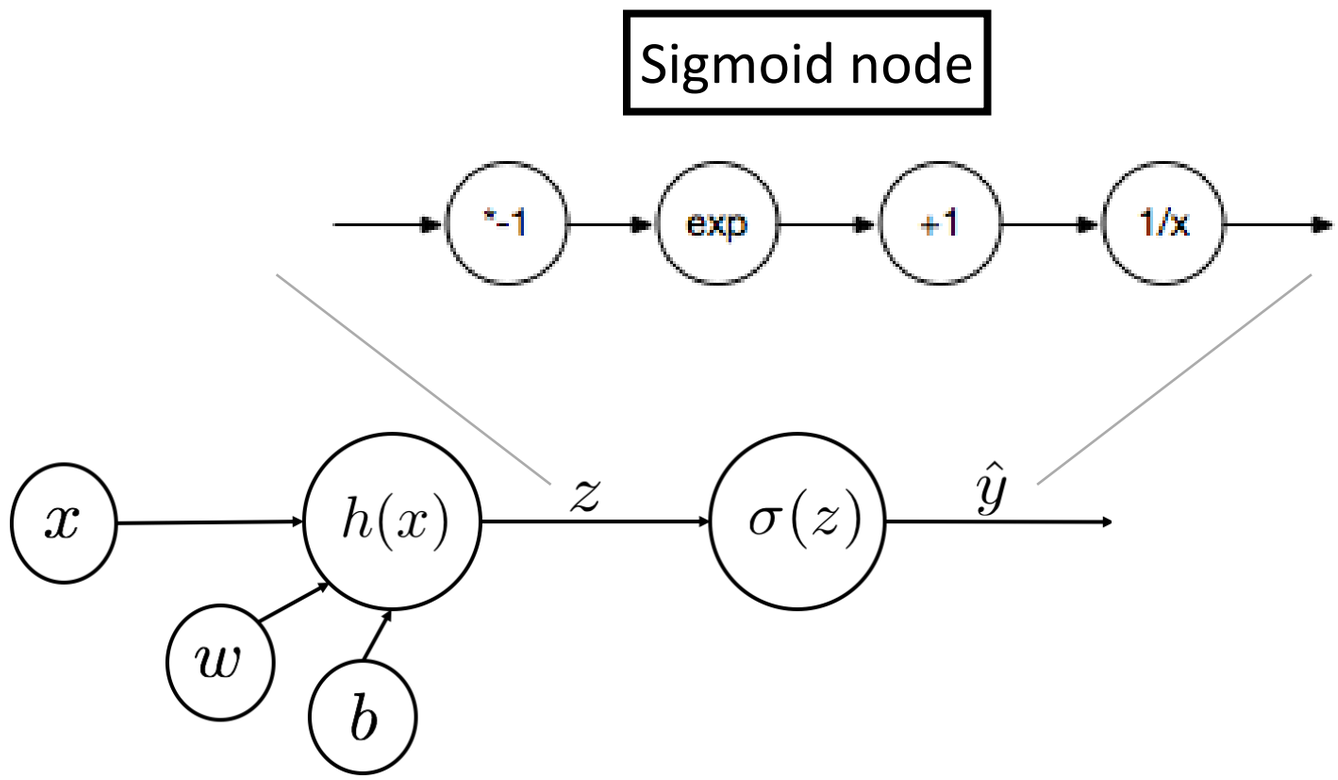
Advantage 2: We can easily define new nodes using the forward/backward pattern

- All nodes will follow the same design pattern




```
class MultiplyNode(object):

    def forward(x, y):
        self.x = x   # must be kept for when backward is called
        self.y = y
        z = x * y
        return z

    def backward(grad_z):
        grad_x = grad_z * self.y   # dL/dz * dz/dx
        grad_y = grad_z * self.x   # dL/dz * dz/dy
        return [grad_x, grad_y]
```

Nodes composition or factorisation



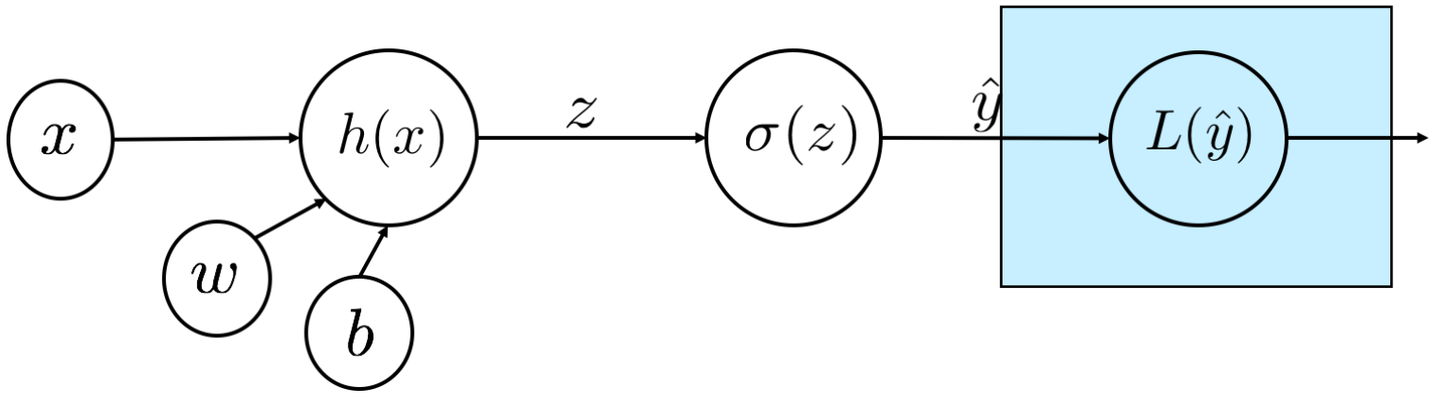
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Meta-nodes can be composed of nodes
- Or equivalently, a sub-graph composed of nodes can be re-implemented in a single node if we can compute an analytic form of the gradients

$$\frac{\partial \sigma}{\partial z} = (1 - \sigma(z))\sigma(z)$$

Advantages 3 & 4: Node composition or factorisation: any complex learning architecture can be composed from atomic nodes. No need to compute complex global gradient.

The loss is also composed from nodes



- The loss computation is “plugged” at the end of the graph and constitutes a sub-graph composed of nodes
  - The loss value is computed in the forward pass
  - The gradient is back-propagated from the loss

Advantage 5: The loss functions can actually be seen as extra nodes in the graph (update rules too).

Summary of advantages for computational graphs

Advantage 1: Intuitive interpretation of gradient backpropagation

Advantage 2: We can easily define new nodes using the forward/backward pattern

Advantages 3 & 4: Node composition or factorisation: any complex learning architecture can be composed from atomic nodes. No need to compute complex global gradient.

Advantage 5: The loss functions can actually be seen as extra nodes in the graph (update rules too).

- We will observe advantages 3 to 5 in industrial computational graph implementations.

Deep Learning Frameworks

A zoo of frameworks

Our choice

Tensorflow tutorial

A look at the code in other frameworks

A zoo of frameworks for deep learning

Academia

Theano - U Montreal
Torch - IDIAP/NYU/…
Caffe - UC Berkeley
MXNet - CMU, MIT, U Wash, HK UST
…

Companies

TensorFlow - Google
Pytorch - Facebook
Caffe2 - Facebook
MXNet - Amazon, Intel
DeepLearning4J -SkyMind
CNTK - Microsoft
PaddlePaddle - Baidu
…

See more on https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

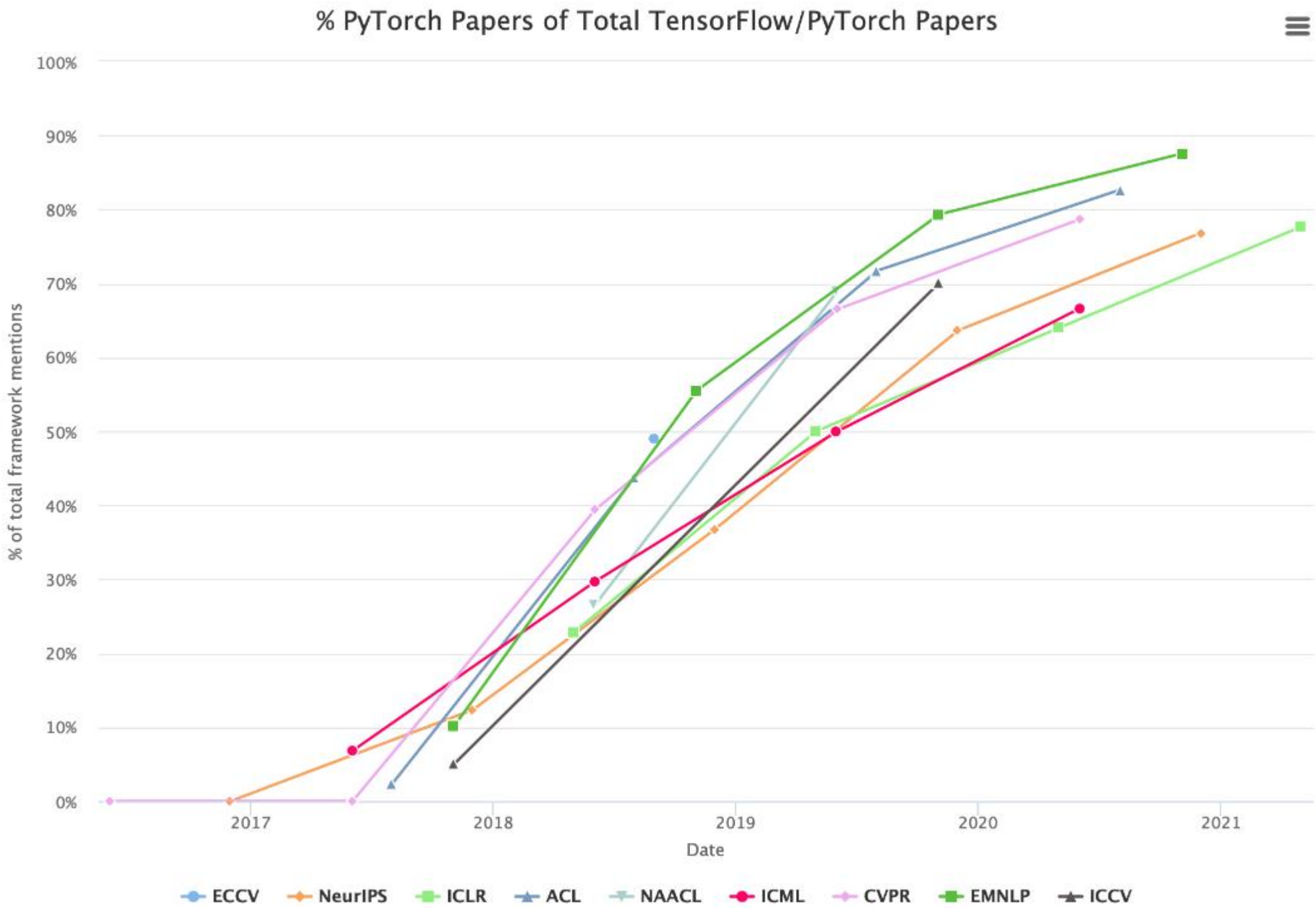
TensorFlow vs. Pytorch

- From Google
- Based on Theano
- Initially based on a **static** computational graph strategy
  - e.g. faster on CNN, more cumbersome on variable input length such as in RNN
  - From version 2.0: based on an “eager” compilation of the graph (a bit as in Pytorch), also more “Pythonic"
- Steeper learning curve, people usually use high level APIs (Keras)
- Bigger community
- Seems more accepted to go to production in the industry

- From Facebook
- Based on Torch
- **Dynamic** computational graph strategy
  - e.g. faster on RNN and slower on static architectures
  - Can handle variable sequence length in RNN
- Closer to python code, more “pythonic”, feel more “Python native”
- Younger, smaller community
- Better to do rapid prototyping, seems to be well accepted in the research community

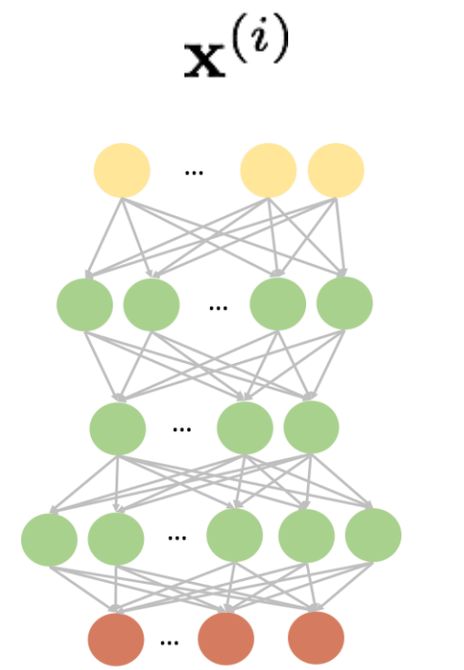
For this year’s DL class we will chose Tensorflow + Keras

TensorFlow vs. Pytorch

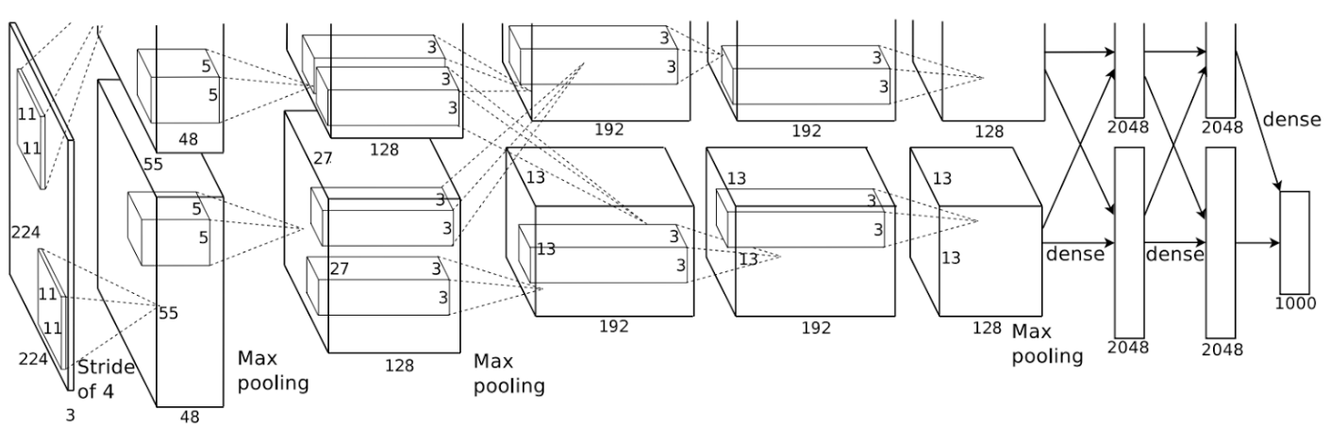


4 main reasons to use DL frameworks

1. Easy to build big computational graphs
2. Easy to compute losses $\mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$ and gradients in computational graphs for update rules $param \leftarrow param - \alpha \frac{\partial L}{\partial param}$
3. Have at hand all the state-of-the-art strategies for regularisations and optimisations
4. Switch easily from cpu to gpu when needed



$$\hat{\mathbf{y}}^{(i)} = h_\theta(\mathbf{x}^{(i)})$$



Tensorflow 2.0 installation

TensorFlow

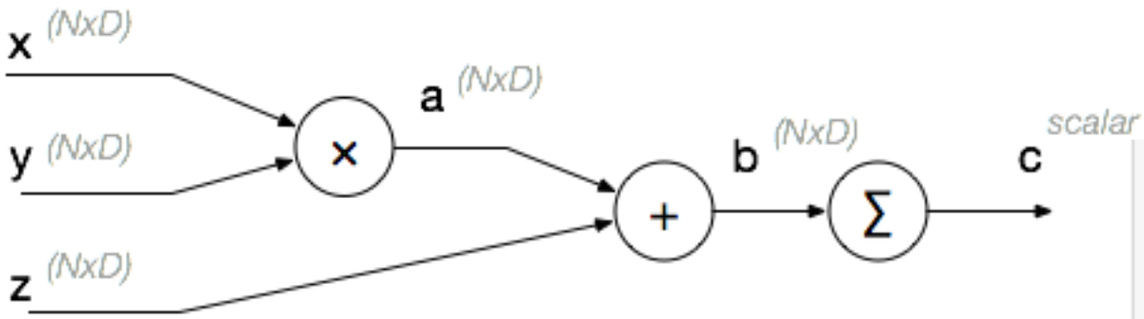
- See https://www.tensorflow.org/install/pip
- We recommend using a virtual environment
  - https://virtualenv.pypa.io/en/latest/user_guide.html
  - The packages are evolving rather fast
  - "Isolate" your packages in a directory so that
    - you can reproduce all your experiments with the right versions
    - you can "try out" new release safely without corrupting your system
  - Install virtualenv
    - pip install virtualenv OR conda install -c anaconda virtualenv (if anaconda install)
  - Create your virtualenv
    - virtualenv -p python3 ./venv OR python -m venv venv
  - Activate virtualenv (you need to do this each time you use you environment
    - source venv/bin/activate # Mac OS
    - .\venv\Scripts\activate # Windows if trouble with execution policies 'Set-ExecutionPolicy Unrestricted -Force'
  - Install Tensorflow
    - pip install --upgrade tensorflow
  - Install Jupiter, matplotlib
    - pip install jupyter
    - pip install matplotlib
  - Use it with
    - jupyter notebook
    - OR create a kernel : ipython kernel install —user —name=WHATEVER (from within the env)
  - Deactivate the environment
    - deactivate

Tensorflow 1.x versus 2.0

TensorFlow

- TF moved from 1.x to 2.0 this year with significant changes in the syntax and behaviour.
- We provide code in the following examples using both syntax.
- Pay attention to the version:
  - 1.x
  - 2.0

TensorFlow 1 - simple example



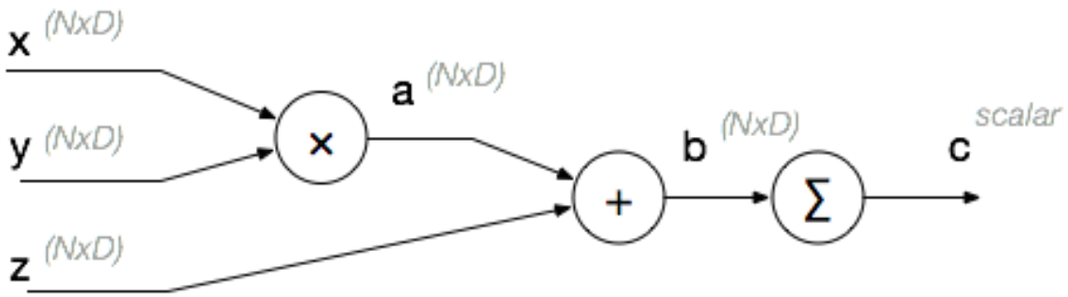

```python
# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N,D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

We compute ourselves the gradients! Automatic in TensorFlow

```python
import numpy as np
import tensorflow.compat.v1 as tf

tf.disable_v2_behavior() # to use former syntax of
                         # tensorflow 1.X, eg 1.14
# simple computational graph with tensorflow
N, D = 3, 4
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=(N, D))

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D)
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Example inspired from Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

TensorFlow 1 - simple example

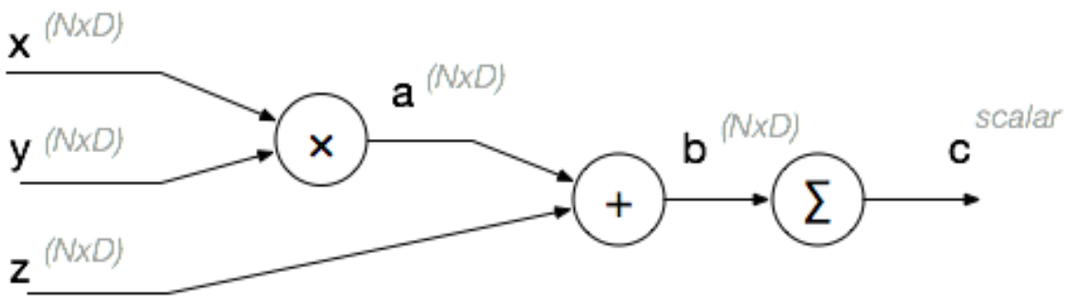



```python
# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N,D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Just one line to run the graph on gpu

```python
# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

    grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D)
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Tensorflow 1 - simple example



First define the computational graph

```
# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

    grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])
```

Then run the graph many times

```
with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D)
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

This is the “static” graph strategy of TensorFlow. The graph structure is defined and then sits on the gpu. Quite efficient for graph where the structure is not dynamic, e.g. CNN. Optimisations can also be applied on the graph if it is static.

TensorFlow 1 - simple example

Numpy

```python
import numpy as np

# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N,D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

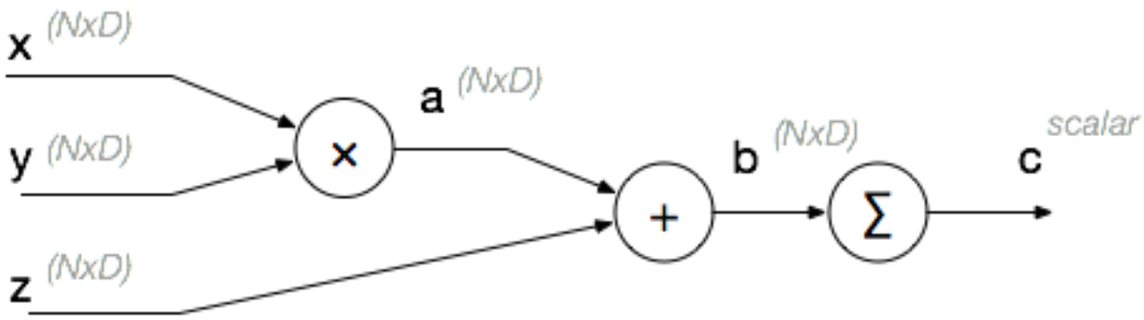
Tensorflow

```python
import numpy as np
import tensorflow as tf

# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

    grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D)
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Pytorch

```python
import torch
from torch.autograd import Variable

# simple computational graph - torch on gpu
N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

TensorFlow 1 versus 2 - simple example



Graph definitions.
In TF1: through placeholders.
In TF2: through functions with a *@tf.function* decorator.

1

```
import numpy as np
import tensorflow.compat.v1 as tf

tf.disable_v2_behavior() # to use former syntax of
                         # tensorflow 1.X, eg 1.14
# simple computational graph with tensorflow
N, D = 3, 4
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=(N, D))

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])
```

```
with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D)
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

2

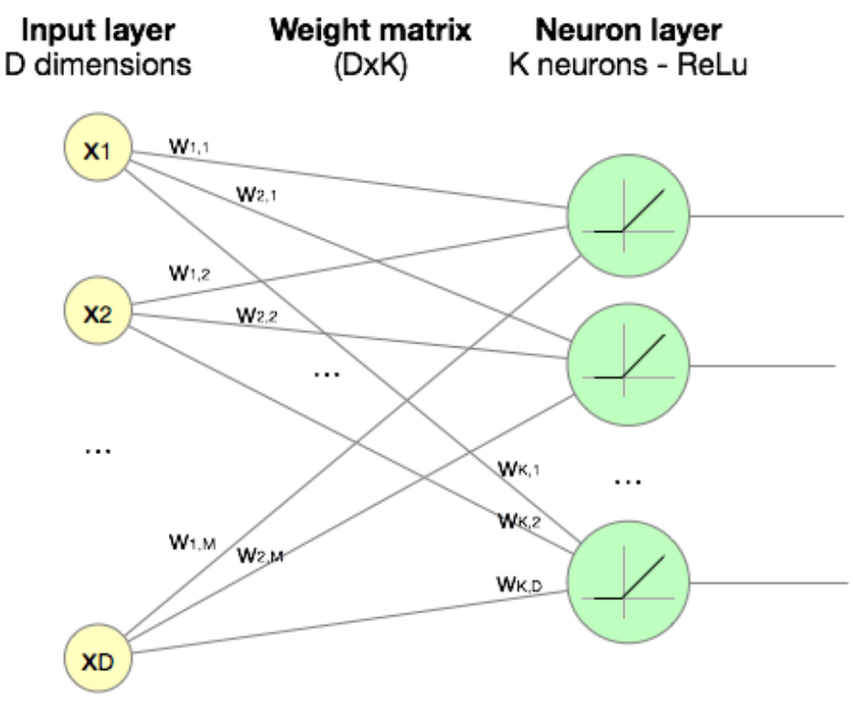
```
#TENSORFLOW 2.0
import numpy as np
import tensorflow as tf

# simple computational graph with tensorflow

@tf.function # this decorator tells tf that a graph is defined
def simple_graph(x, y, z) :
    a = x * y
    b = a + z
    c = tf.reduce_sum(input_tensor=b)
    grad_x, grad_y, grad_z = tf.gradients(ys=c, xs=[x, y, z])
    return c, grad_x, grad_y, grad_z

N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

c_val, grad_x_val, grad_y_val, grad_z_val = simple_graph(x, y, z)

print(c_val)
print(grad_x_val)
print(grad_y_val)
print(grad_z_val)
```

Run the graph. In TF1: through a tf.Session(). In TF2: calling the function.

Starting from this slides,
all examples are with TensorFlow 2.0

TensorFlow - a more complex example

Assuming a classification problem with training set in x_train and target variables 1-hot in y_train.



Training with MSE loss

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

```
import numpy as np
import tensorflow as tf

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

N = x_train.shape[0]        # number of samples
D = x_train.shape[1]        # dimension of input sample
n_classes = y_train.shape[1] # output dim

np.random.seed(0)
w1 = np.random.randn(D, n_classes)

with tf.device('/CPU:0'):  # change to /GPU:0 to move to GPU
    out = simple_ann_train(x_train, w1, y_train)

y_pred, loss_val, grad = out
grad_w1 = grad[0] #grad is a list
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

First define the computational graph. No computation, the graph is *eagerly* built when the function is called.

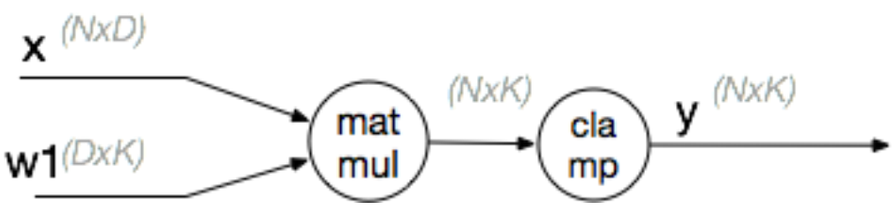
Then run the graph on the selected device

```
import numpy as np
import tensorflow as tf

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

N = x_train.shape[0]        # number of samples
D = x_train.shape[1]        # dimension of input sample
n_classes = y_train.shape[1] # output dim

np.random.seed(0)
w1 = np.random.randn(D, n_classes)

with tf.device('/CPU:0'):  # change to /GPU:0 to move to GPU
    out = simple_ann_train(x_train, w1, y_train)

y_pred, loss_val, grad = out
grad_w1 = grad[0] #grad is a list
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

Computational graph creation: forward pass

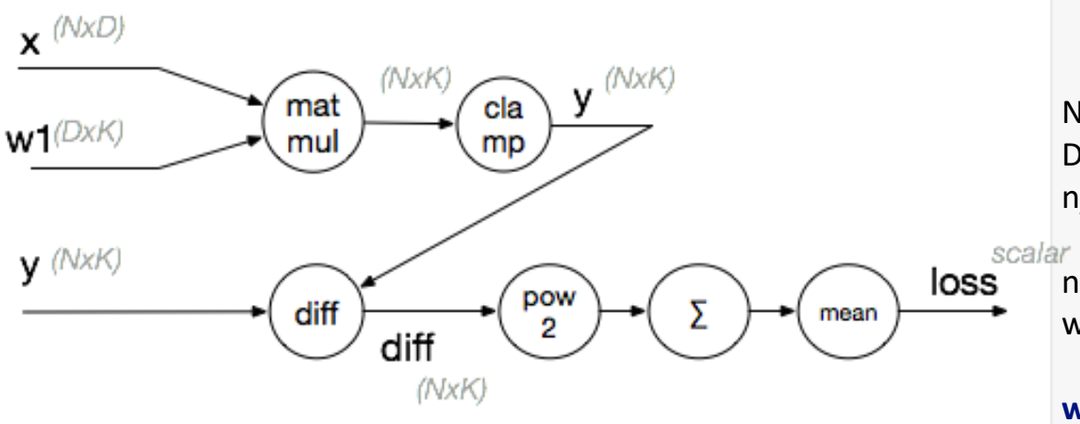



```
import numpy as np
import tensorflow as tf

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

N = x_train.shape[0]        # number of samples
D = x_train.shape[1]        # dimension of input sample
n_classes = y_train.shape[1] # output dim

np.random.seed(0)
w1 = np.random.randn(D, n_classes)

with tf.device('/CPU:0'):  # change to /GPU:0 to move to GPU
    out = simple_ann_train(x_train, w1, y_train)

y_pred, loss_val, grad = out
grad_w1 = grad[0] #grad is a list
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

Computational graph creation: forward pass and loss




```
import numpy as np
import tensorflow as tf

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

N = x_train.shape[0]        # number of samples
D = x_train.shape[1]        # dimension of input sample
n_classes = y_train.shape[1] # output dim

np.random.seed(0)
w1 = np.random.randn(D, n_classes)

with tf.device('/CPU:0'):  # change to /GPU:0 to move to GPU
    out = simple_ann_train(x_train, w1, y_train)

y_pred, loss_val, grad = out
grad_w1 = grad[0] #grad is a list
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

```
import numpy as np
import tensorflow as tf

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

N = x_train.shape[0]        # number of samples
D = x_train.shape[1]        # dimension of input sample
n_classes = y_train.shape[1] # output dim

np.random.seed(0)
w1 = np.random.randn(D, n_classes)

with tf.device('/CPU:0'):  # change to /GPU:0 to move to GPU
    out = simple_ann_train(x_train, w1, y_train)

y_pred, loss_val, grad = out
grad_w1 = grad[0] #grad is a list
```

Now we enter a session to run the graph. Feed the graph with inputs x, weights w1 and targets y

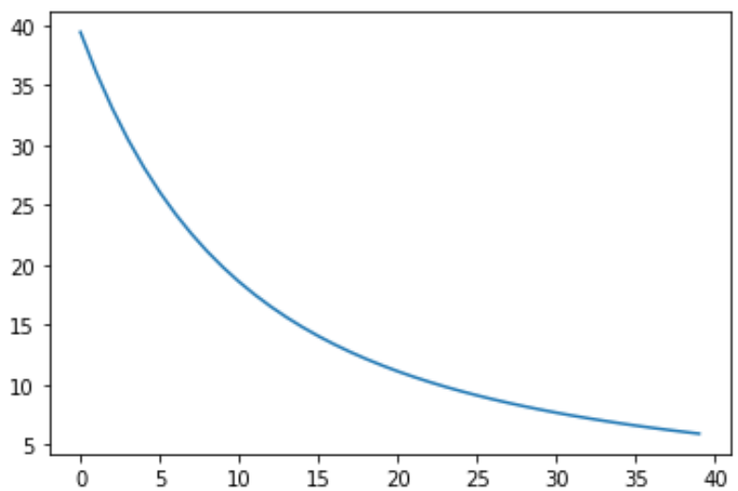
Graph is run and returns the gotten outputs y_pred, the loss value loss_val and the gradients grad_w1

TensorFlow - a more complex example

Now we include a for loop with 40 epochs of updating the weights. The gradients are computed on the full training set:"*full batch*" mode.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, w1, y):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    return y_pred, loss, grad

np.random.seed(0)
w1 = np.random.randn(D, n_classes)
alpha = 1e-2
J = []
for epoch in range(40):
    with tf.device('/GPU:0'):  # change to /GPU:0 to move it to GPU
        out = simple_ann_train(x_train, w1, y_train)
    y_pred, loss_val, grad = out
    grad_w1 = grad[0] # grad is a list of gradients
    w1 -= alpha * grad_w1.numpy()
    J.append(loss_val)
    print("epoch = {}, loss = {}".format(epoch, loss_val))

plt.plot(J)
```

Now train for 20 epochs upgrading the weights.



Problem: weights are copied between cpu and gpu at each step.

TensorFlow - a more complex example

Now we avoid the problem of transferring the weights back and forth between cpu and gpu.

Change w1 from **placeholder inputs** (fed on each call) to **Variable** (persist in the graph between calls).

The values are initialised once when the variable w1 is declared.

Add **assign** operation to update w1 as part of the graph.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, y, alpha):
    y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
    diff = y_pred - y
    loss = tf.reduce_mean(tf.pow(diff, 2))
    grad = tf.gradients(ys=loss, xs=[w1])
    # tf.gradients returns a list of sum(dy/dx) for each x in xs
    grad_w1 = grad[0]
    w1.assign(w1 - alpha * grad_w1)
    return y_pred, loss

np.random.seed(0)
alpha = 1e-2
J = []
w1 = tf.Variable(tf.random.normal((D, n_classes), dtype='float64'))
for epoch in range(40):
    with tf.device('/CPU:0'):  # change to /GPU:0 to move it to GPU
        out = simple_ann_train(x_train, y_train, alpha)
    y_pred, loss_val = out
    J.append(loss_val)
    print("epoch = {}, loss = {}".format(epoch,loss_val))

plt.plot(J)
```

TensorFlow - a more complex example

Instead of redefining loss computation or weight updates ourselves, we can rely on the ones provided in TF. This code should be equivalent to the previous one.

Use pre-defined loss functions

Use pre-defined optimizers

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

@tf.function # this decorator tells tf that a graph is defined
def simple_ann_train(x, y, alpha):
    y_pred = tf.nn.relu(tf.matmul(x, w1))  # ReLU on logit
    mse = tf.keras.losses.MeanSquaredError()
    loss = mse(y, y_pred)
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(1e-2)
    updates = optimizer.minimize(loss, var_list=w1)
    return y_pred, loss

np.random.seed(0)
alpha = 1e-2
J = []
w1 = tf.Variable(tf.random.normal((D, n_classes), dtype='float64'))
for epoch in range(50):
    with tf.device('/CPU:0'):  # change to /GPU:0 to move it to GPU
        out = simple_ann_train(x_train, y_train, alpha)
    y_pred, loss_val = out
    J.append(loss_val)
    print("epoch = {}, loss = {}".format(epoch, loss_val))

plt.plot(J)
```

Keras

Overview

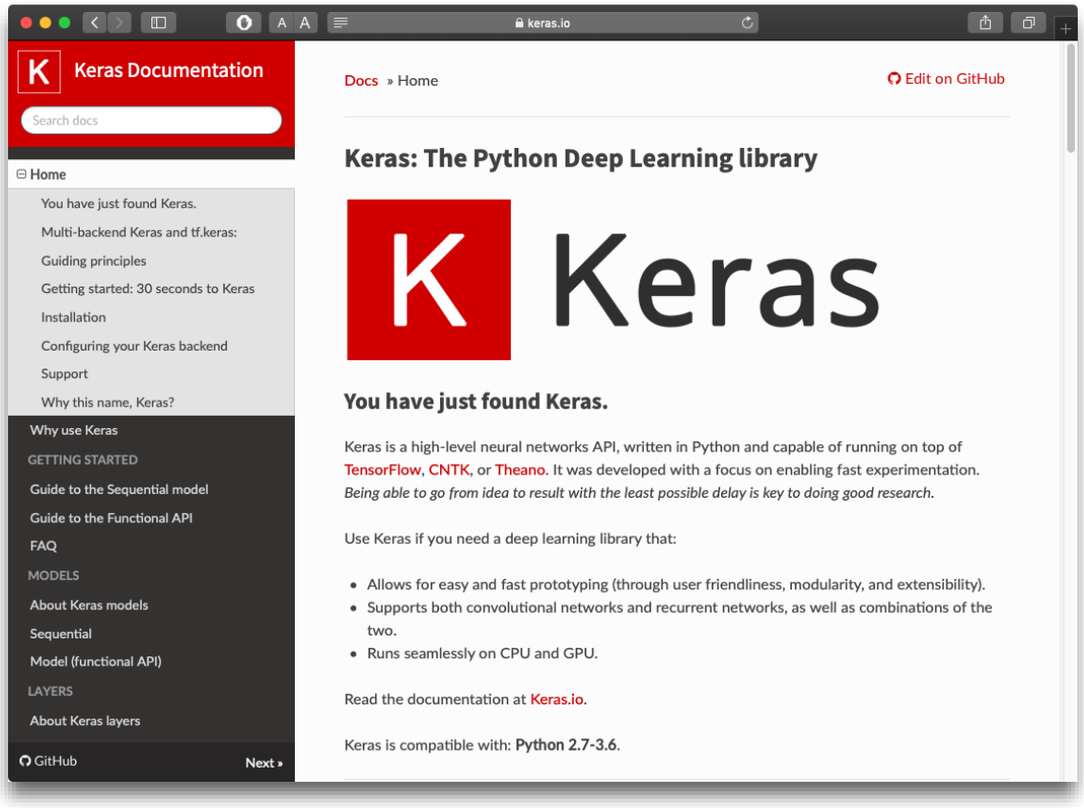
Cheat Sheet

Example

Keras - what is it?

Keras is a high-level open-source neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

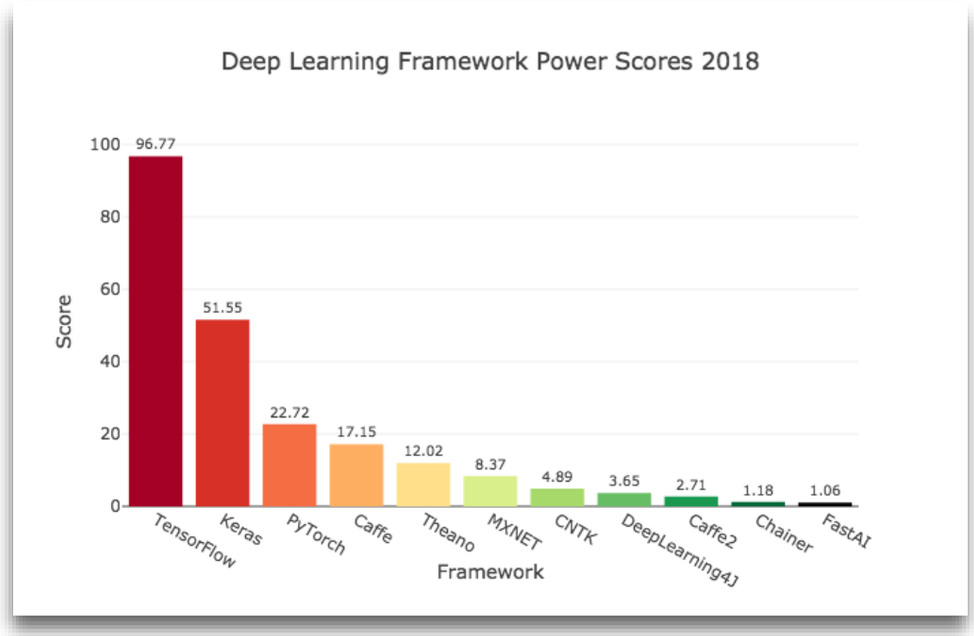
- Designed to be minimalistic & straight forward yet extensive
- "Deep" enough to build serious models
  - Support Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), combination of both.
- Wrapper on top of TF / CNTK / Theano backend
  - But not only a wrapper - see next slides

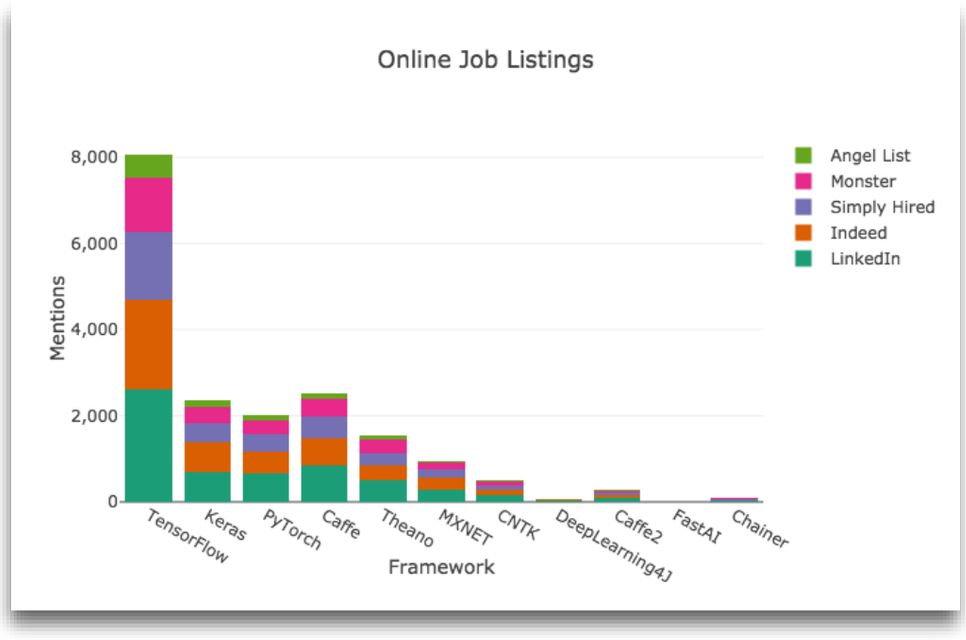


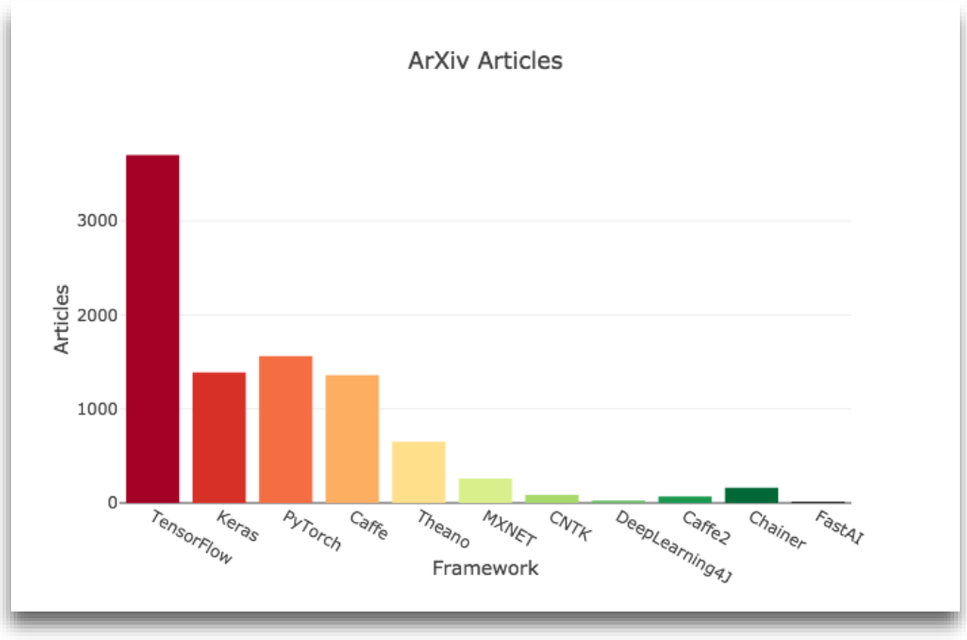
Source: https://keras.io

Keras - why using?

- Simple to use, well done documentation
- Not only wrapper, it “integrates” with TensorFlow
  - K can call low-level TF methods
  - TF can call high-level Keras methods
- Not only TF, also CNTK (from Microsoft) and Theano
  - Code and model portability
    - For ex train on TF and test on CNTK
  - MXNet coming soon (officially not supported yet)





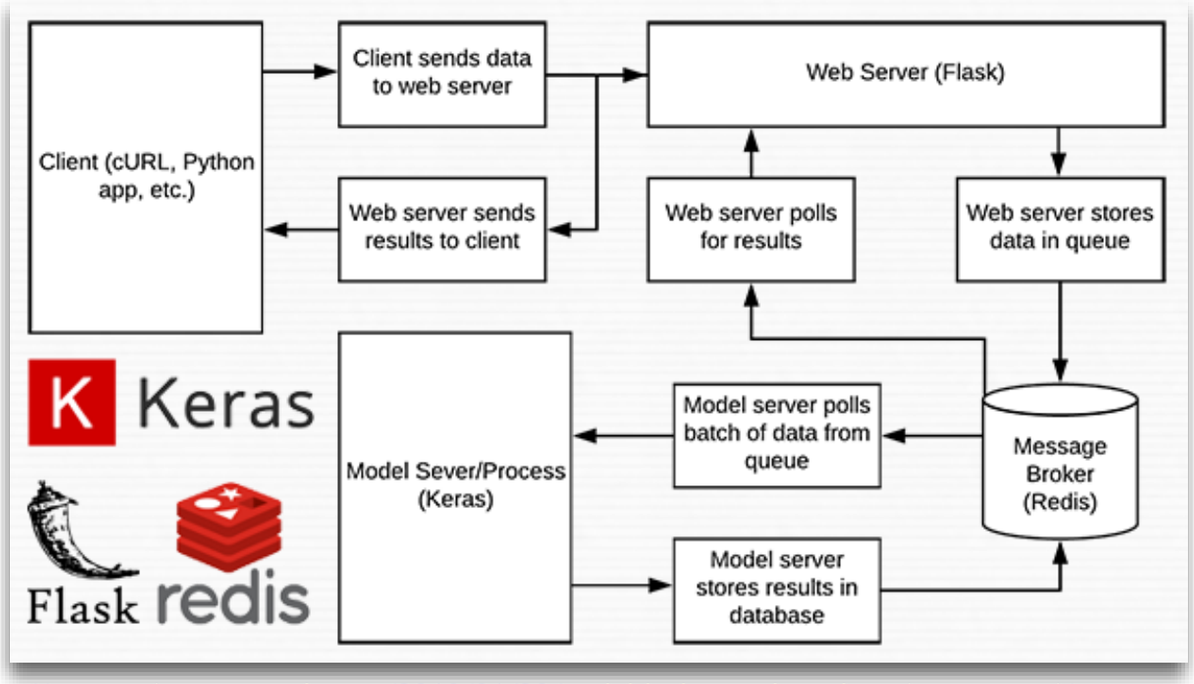


Source: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

Keras - why using?

- Not only for research, in production at Netflix, Uber, Yelp, Instacart, Zocdoc, Square, and many others.
- Going into production:
  - On iOS, via Apple's CoreML
  - On Android, via TensorFlow Android runtime.
    - Example: Not Hotdog app. See the hilarious Silicon Valley episode https://youtu.be/mrk95jFVKqY
  - In the browser, via GPU-accelerated JavaScript runtimes such as Keras.js and WebDNN
  - On Google Cloud, via TensorFlow-Serving
  - In an R or Python webapp backend (such as a Shiny or Flask app)

Source: https://keras.rstudio.com/articles/why_use_keras.html



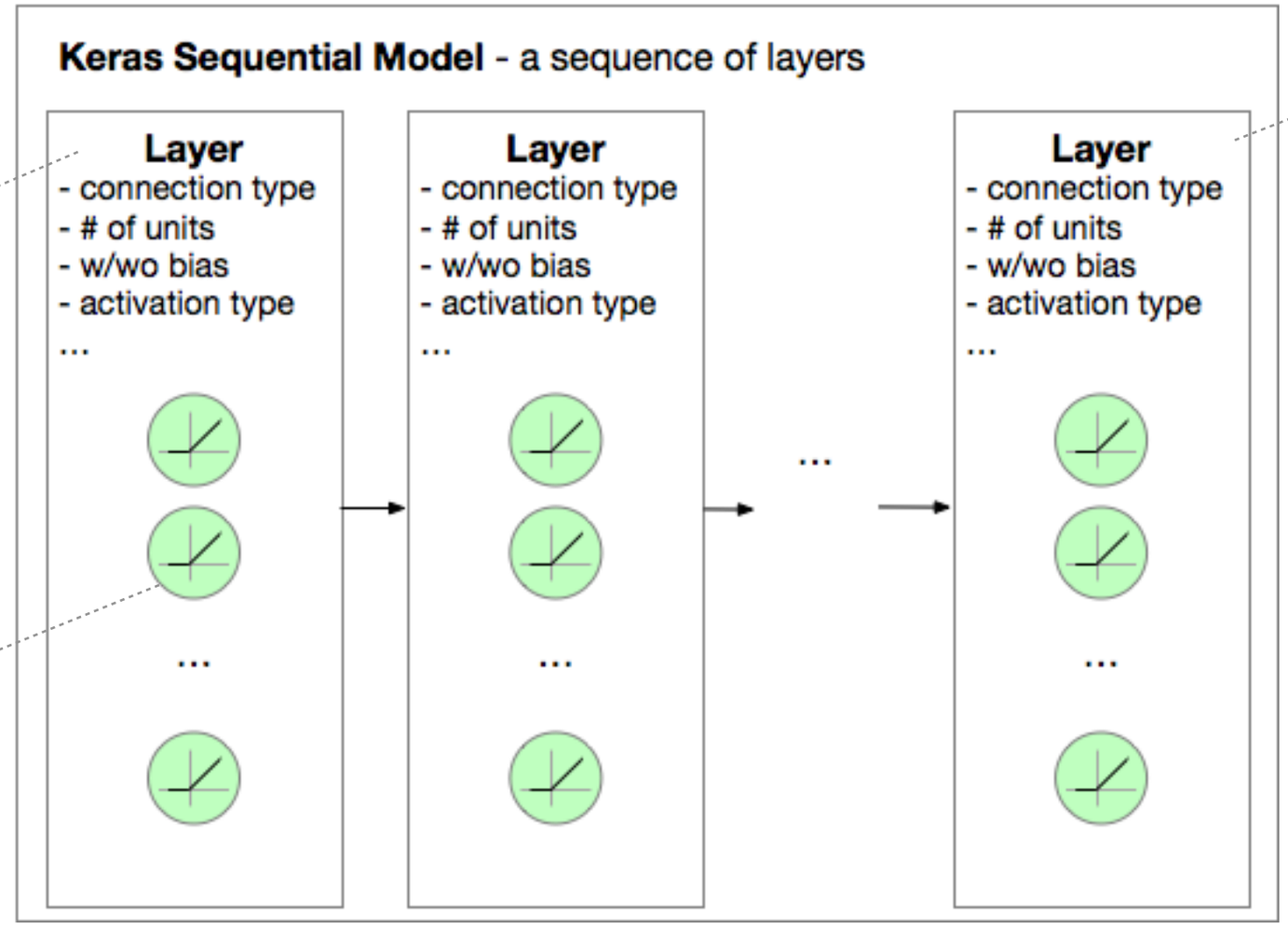
https://www.pyimagesearch.com/2018/01/29/scalable-keras-deep-learning-rest-api/

Keras Models

- A **model** in Keras is the way to organise the layers of neurons: sequential or functional
- The **sequential** model corresponds to a regular stack of layers
  - 1 layer = 1 object that feeds to the next

Once the model initialised, we “add()” layers

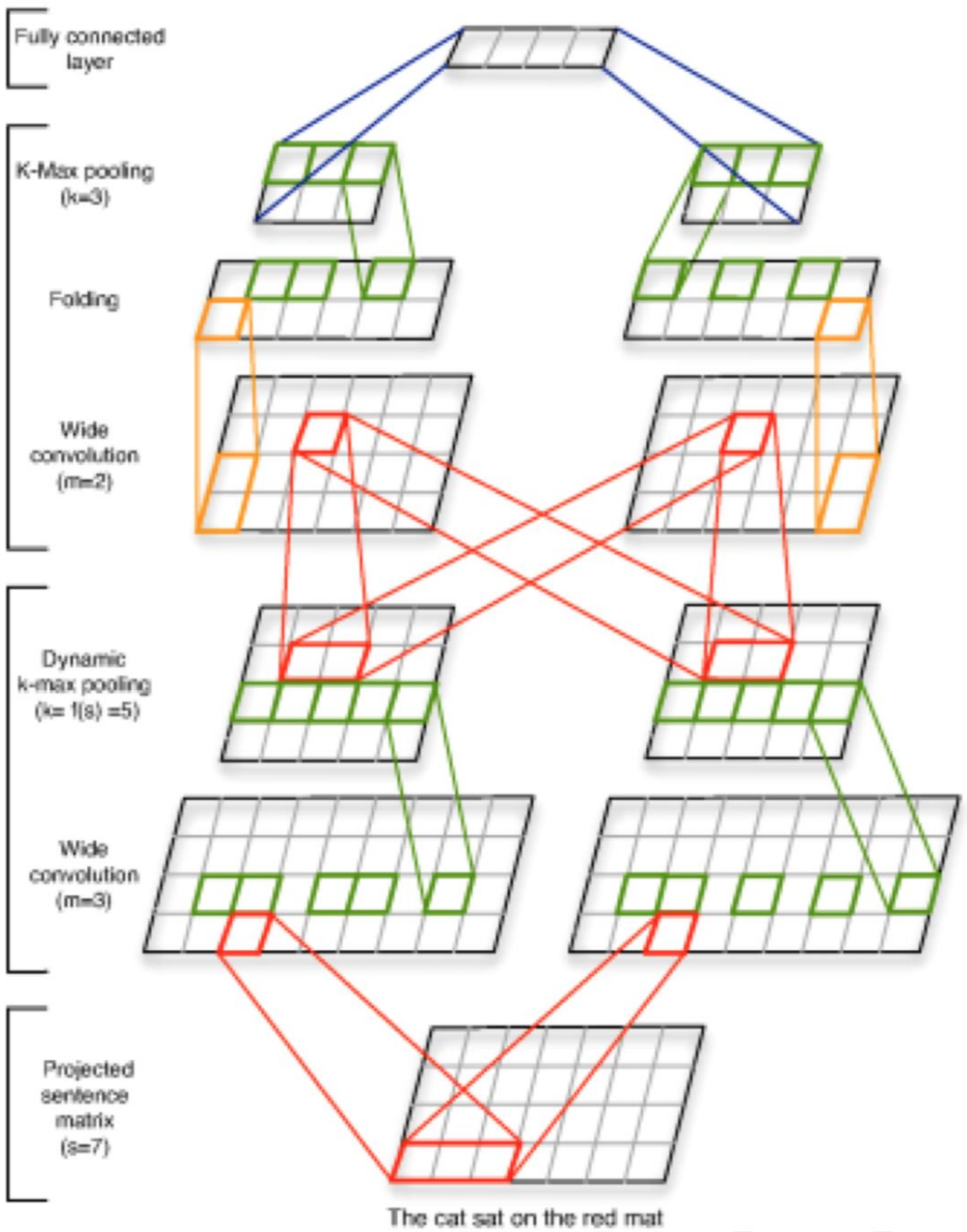
The term “unit” is used to represent a neuron



A layer is defined by the type of connections to the previous layer, e.g. Dense, by the number of neurons, by the activation type, e.g. ReLu, sigmoid, … by the use or not of bias terms, etc

Keras Models

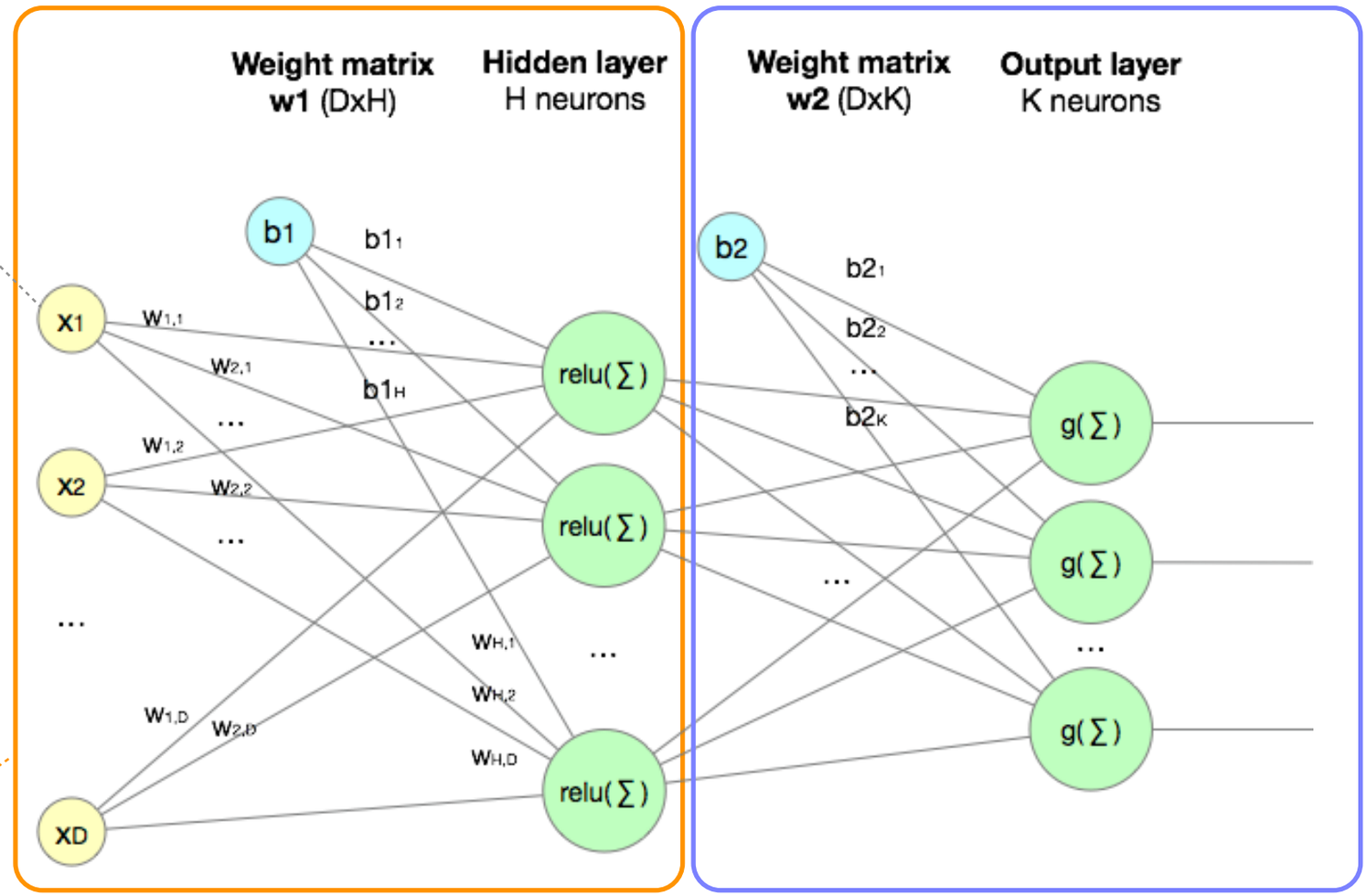
- The functional API allows to define **graphs** of layers and is used for non sequential architectures
  - Typically allowing for independent networks to diverge or merge



Layers in Keras - Dense

- Dense layers represent fully connected layers of neurons

Note: the "input layer" usually defined in books is actually not a really a layer.

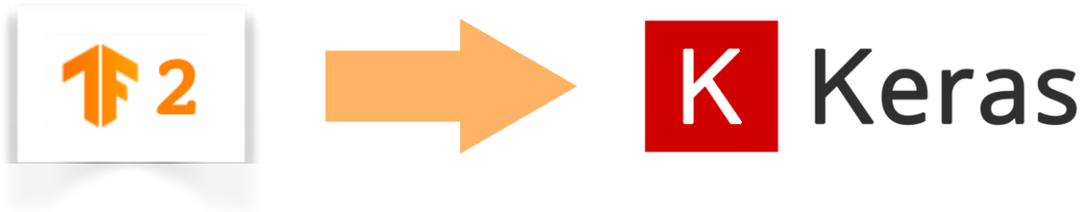


model.add(Dense(H, input_shape=(D,), activation='relu'))

model.add(Dense(n_classes, activation='sigmoid'))

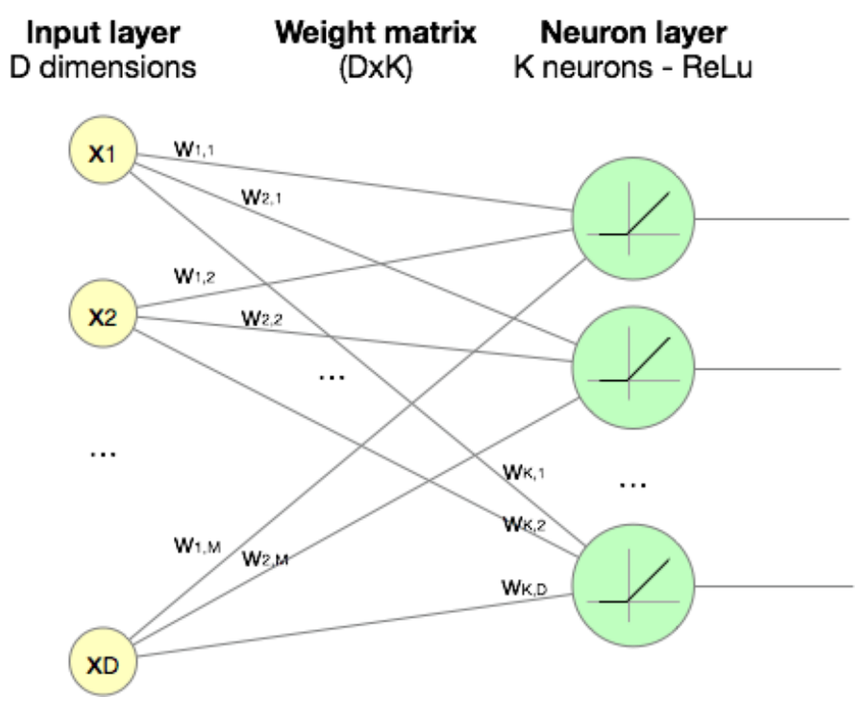
Keras and TensorFlow

- From TF 2.0, Keras has been fully integrated in TF
- For users of TF 2.0:
  - It is recommended to use the package `tf.keras` integrated in TF



- For users of TF 1.x:
  - You can use directly Keras that, in turns, is calling TF

TensorFlow + Keras - a more complex example



We use here the Sequential model of Keras to build a computational graph equivalent to the one of slide 42.

```
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, input_shape=(D,),
                                use_bias=False, activation='relu'))
model.summary()

sgd = tf.keras.optimizers.SGD(learning_rate=0.5)

model.compile(optimizer=sgd, loss='mse', metrics=['accuracy'])

history1 = model.fit(x_train, y_train, batch_size=N, epochs=40)

model.evaluate(x_test,  y_test, verbose=2)
```

Assuming a classification problem with training set in x_train and target variables 1-hot in y_train. Training with MSE loss

TensorFlow + Keras - a more complex example

We use here the Sequential model of Keras to build a computational graph equivalent to the one of slide 42

We define a Keras Sequential model. We add a Dense layer with 10 neurons fully connected to the inputs (of size D). Bias are not used and activation function is a Rectified Linear Unit (ReLU).

Use pre-defined Stochastic Gradient Descent optimizer.

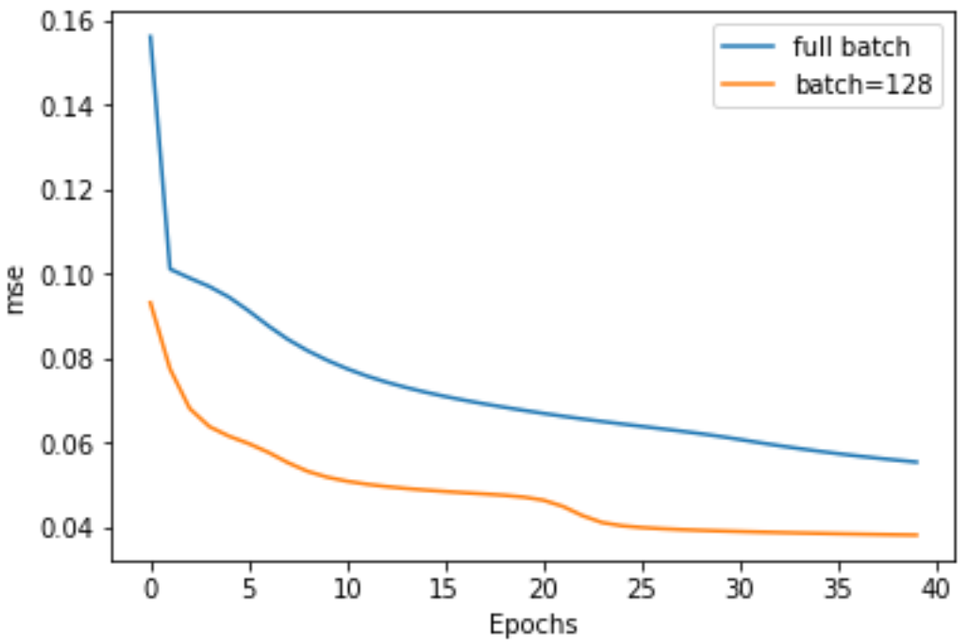
The model is "compiled", i.e. the graph is composed. We define here the loss function and the metrics to observe from epoch to epoch.

The model is trained with the *fit* function with args including train data, targets, batch size and number of epochs.

```
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, input_shape=(D,),
                                use_bias=False, activation='relu'))
model.summary()

sgd = tf.keras.optimizers.SGD(learning_rate=0.5)

model.compile(optimizer=sgd, loss='mse', metrics=['accuracy'])

history1 = model.fit(x_train, y_train, batch_size=N, epochs=40)

model.evaluate(x_test,  y_test, verbose=2)
```

TensorFlow + Keras - a more complex example

Full batch versus batched SGD.




```
Loss and Accuracy on MNIST test set
[0.05339845517277718, 0.6011]
```

```
Loss and Accuracy on MNIST test set
[0.03688519067168236, 0.733]
```

```
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, input_shape=(D,),
                                use_bias=False, activation='relu'))
model.summary()

sgd = tf.keras.optimizers.SGD(learning_rate=0.5)

model.compile(optimizer=sgd, loss='mse', metrics=['accuracy'])

history1 = model.fit(x_train, y_train, batch_size=N, epochs=40)

model.evaluate(x_test,  y_test, verbose=2)
```

```
import tensorflow as tf


model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, input_shape=(D,),
                                use_bias=False, activation='relu'))
model.summary()

sgd = tf.keras.optimizers.SGD(learning_rate=0.01)

model.compile(optimizer=sgd, loss='mse', metrics=['accuracy'])

history2 = model.fit(x_train, y_train, batch_size=128, epochs=40)

model.evaluate(x_test,  y_test, verbose=2)
```

Wrap-up

- **Computational graph** implementations are important in deep learning:
  - Intuitive interpretation of gradient back-propagation
  - Easy to define new nodes using the forward/backward pattern
  - Node composition or factorisation
    - any complex architecture can be composed from atomic nodes
    - atomic nodes can be factorised into less atomic nodes, e.g. sigmoid or layers
  - No need to compute manually complex global gradient.
  - The loss functions can actually be seen as extra nodes in the graph (update rules too).
- **Deep learning frameworks**
  - It is a zoo! **Very** fast moving landscape.
  - Two big players of 2019 are TensorFlow and Pytorch
- **Keras** is a high level API used to ease the creation of large neural network architectures.