Course: Computer Vision

Unit 4: Object Recognition

Introduction to Deep Learning with Keras

Luis Baumela
2018
Universidad Politécnica de Madrid

Introduction to Deep Learning with Keras

1. Brief review
   - Machine learning life cicle
   - Deep learning frameworks

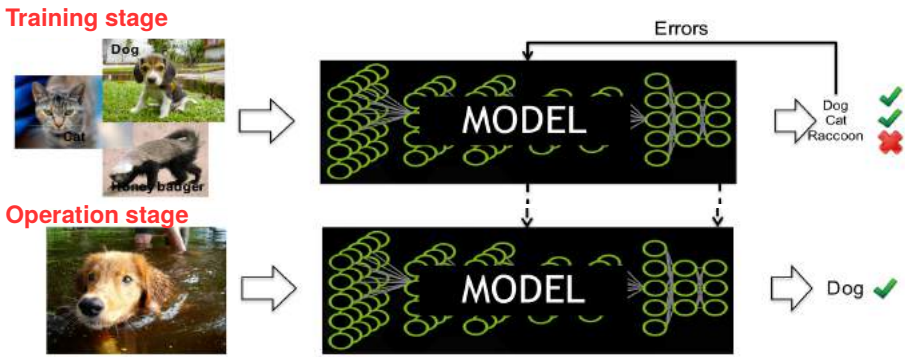
2. Deep Learning with Keras
   - Overview
   - Main components and training steps
   - Convert and load data
   - Network definition
   - Solver definition
   - Training

3. Homework

Deep Learning with Caffe

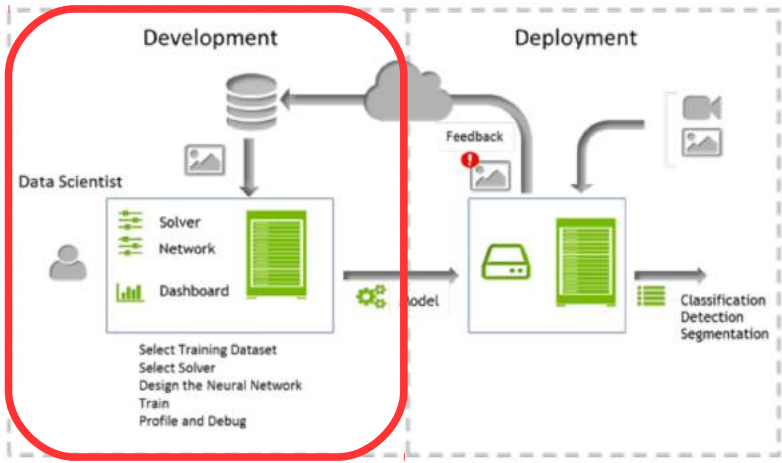
- The machine learning approach to computer vision

  Make machines that learn to “see”.



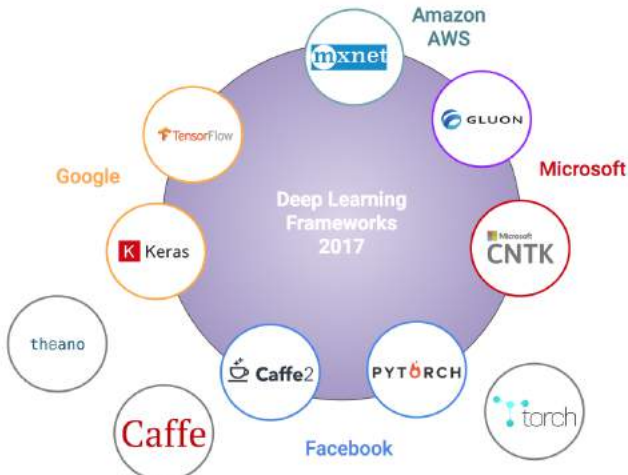
Deep Learning with Caffe

- The deep learning life cicle



Deep Learning with Caffe

- Some deep learning frameworks



https://towardsdatascience.com/battle-of-the-deep-learning-frameworks-part-i-cff0e3841750

Deep Learning with Keras

- Main component
  - **model**. Organized in layers. Stores weights and derivatives.

In [1]:

Slide Type Sub-Slide

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten

model = Sequential()
model.add(Flatten(input_shape=(32, 32, 3)))
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10))
model.add(Activation('softmax'))
```

  - **core layers**.

```
keras.layers.Flatten(data_format=None)
```

http://https://keras.io

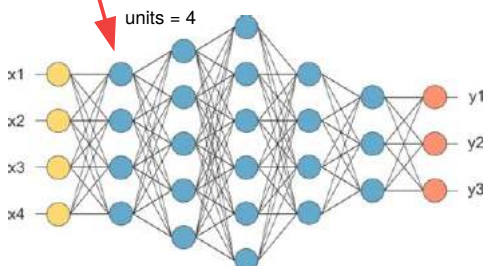
Deep Learning with Keras

- Main component
  - **model**. Organized in layers. Stores data and derivatives.
  - **core layers**.

```
keras.layers.Dense(units, activation=None, use_bias=True,
```

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).



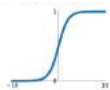
http://https://keras.io

Deep Learning with Keras

- **Hidden layers activation functions**
  - Non-linear component in a multi-layer NN required to learn arbitrarily complex non-linear functions.
  - **Saturating**

Sigmoid

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



tanh

$\tanh(x)$

- Used by initial NN models
- Vanising gradients
- Poor convergence

  - **Non-saturating**

ReLU

$\max(0, x)$

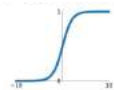
- New DL insignts
- Do not saturate for x > 0
- *Dying ReLUs* problem

Deep Learning with Keras

- **Hidden layers activation functions**
  - Non-linear component in a multi-layer NN required to learn arbitrarily comples non-linear functions.
  - **Saturating**

Sigmoid

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



tanh

$$\tanh(x)$$

- Used by initial NN models
- Vanising gradients
- Poor convergence

  - **Non-saturating**
    - Leaky ReLUs never die
      - parameter fixed, random or learned
    - Exponential Linear Unit (ELU)
      - everywhere smooth
      - nonzero gradient for x < 0
      - average output closer to 0
      - computationally expensive

Leaky ReLU

$$\max(0.1x, x)$$

Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

http://https://keras.io

Deep Learning with Keras

- **Output layers activation functions**
  - **Softmax**
    - NN output modeling a multinomial logistic
    - Generalization of the sigmoid (logistic) to c classes

$$\mathbf{S}(\mathbf{z}_i) = \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^{c} e^{z_i^j}} =$$

such that

$$\mathbf{S}(z) : \mathbb{R}^K \longmapsto \left\{ \mathbf{S} \in \mathbb{R}^K | S^i > 0, \sum_i S^i = 1 \right\}$$

Deep Learning with Keras

- **Weight initialization**
  - **Vanishing/exploding gradients**

    The output activation of a NN with l layers is given by

    $$\mathbf{z}^l = \mathtt{W}^l\mathbf{R}(\ldots\mathtt{W}^3\mathbf{R}(\mathtt{W}^2\mathbf{R}(\mathtt{W}^1\mathbf{x})))$$

    if, for simplification, we assume $\mathbf{R}(\mathbf{z}) = \mathbf{z}$ then

    $$||\mathbf{z}^l|| \approx ||\mathtt{W}^l|| \ldots ||\mathtt{W}^1|| \cdot ||\mathbf{x}|| \approx ||\mathtt{W}||^l \cdot ||\mathbf{x}||$$

    so, for large enough l,

    if $||\mathtt{W}|| > 1$ then $||\mathbf{z}^l|| \longrightarrow \infty$

    if $||\mathtt{W}|| < 1$ then $||\mathbf{z}^l|| \longrightarrow 0$.

    The same happens to the gradients, so learning may diverge or become stalled.

A partial solution is to carefully initialize the network weights

Deep Learning with Keras

- **Weight initialization**

  For a proper behaviour we need that layers do not change the input data variance.

  - Sigmoid (Xavier or Glorot initialization)

$$\mathtt{W} \sim \mathcal{N}\left(0, \sigma^2 = \frac{2}{n_{inputs} + n_{outputs}}\right)$$

  - ReLU, and its variants (He initialization)

$$\mathtt{W} \sim \mathcal{N}\left(0, \sigma^2 = \frac{4}{n_{inputs} + n_{outputs}}\right)$$

Deep Learning with Keras

- Main component
  - **model.** Organized in layers. Stores data and derivatives.
  - **learning**. Configure the learning process.

Slide Type Sub-Slide

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

  - **Loss.** We train the net by minimizing a loss
  - **Optimizer.** Algorithm used to minimize the loss.
  - **Metrics.** Information about the learning process displayed.

Deep Learning with Keras

- **Loss functions**

  The loss optimized during the net training process.

  - Classification problems.

    {categorical, binary}_crossentropy

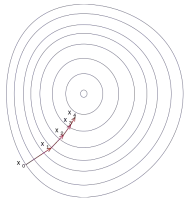
$$\mathcal{L}(\mathbf{g}, \mathcal{D}) = -\frac{1}{n}\sum_{i=1}^{n} \log(\mathbf{x}^l)^{y_i}$$

  - Regression problems.

    mean_{squared, absolute}_error

$$\mathcal{L}(\mathbf{g}, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} [y_i - x_i]^2$$

Deep Learning with Keras

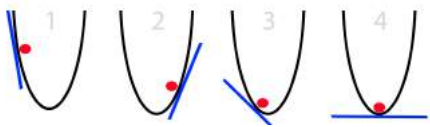


- **Optimizers**
  - **Stochastic gradient descent**

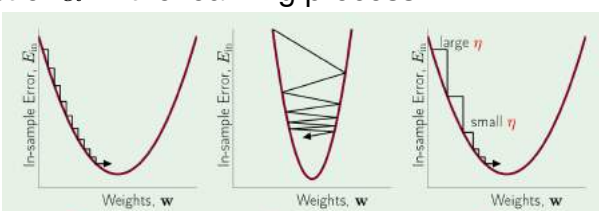
    Minimize the loss by

    $$W_{t+1} = W_t - \alpha \nabla_W \mathcal{L}(W, \mathcal{D})$$

    where α is the **learning rate**.



→ Effect of α in the learning process



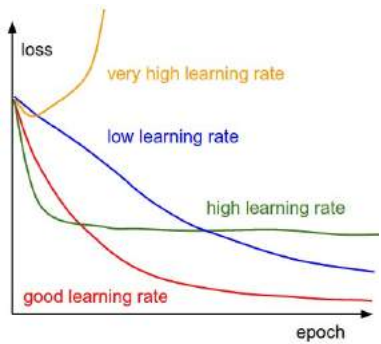
too small too large variable

Deep Learning with Keras

- **Optimizers**
  - **Stochastic gradient descent**

    Tuning the **learning rate**

    1. Start with a fixed value (e.g. 0.01)
       - If error increases or has large oscilations → decrease
       - If error decreases too slowly → increase



Deep Learning with Keras

- **Optimizers**
  - **Stochastic gradient descent**

    Tuning the **learning rate**

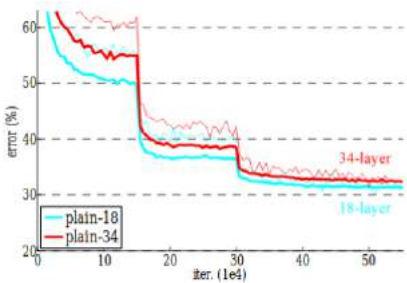
    1. Start with a fixed value (e.g. 0.01)

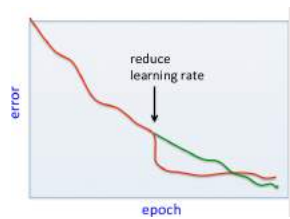
       If error increases or has large oscilations → decrease

       If error decreases too slowly → increase

    2. During traning

       Whenever error stabilizes → decrease





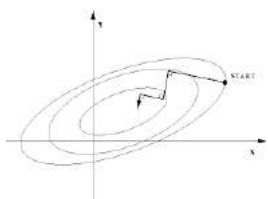
Do not turn down the learning rate too soon!

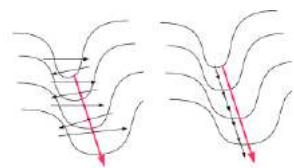
Deep Learning with Keras

- **Optimizers**
  - **Stochastic gradient descent**

    Sometimes the steepest descent direction might not be the best choice.

    Descending along narrow valleys is tough … the narrower, the tougher.





Deep Learning with Keras

- **Optimizers**
  - **Gradient descent with momentum**

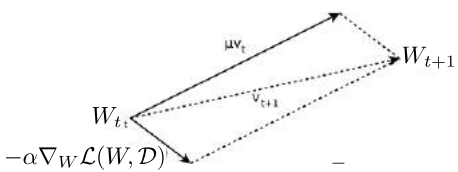
    Weights are updated using a “velocity” term, V

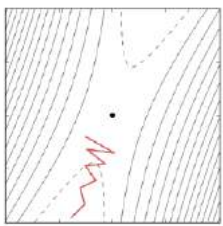
$$W_{t+1} = W_t + V_{t+1}$$

$$V_{t+1} = \mu V_t - \alpha \nabla_W \mathcal{L}(W_t, \mathcal{D})$$

    where, α is the learning rate and $\mu < 1.0$ the **momentum**.

    Now gradient influences velocity, *V*, that has an effect on the position in weight space, *W*.





Deep Learning with Keras

- **Optimizers**
  - **Gradient descent with momentum**

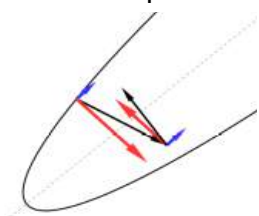
    Weights are updated using a “velocity” term, V

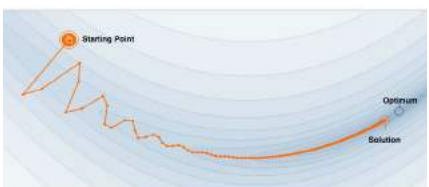
$$W_{t+1} = W_t + V_{t+1}$$

$$V_{t+1} = \mu V_t - \alpha \nabla_W \mathcal{L}(W_t, \mathcal{D})$$

    where, α is the learning rate and $\mu < 1.0$ the **momentum**.

    The momentum averages out opposing gradient components and builds up velocity in directions with consistent gradient





Deep Learning with Keras

- **Optimizers**
  - **Gradient descent with momentum**

    Weights are updated using a “velocity” term, V

$$W_{t+1} = W_t + V_{t+1}$$
$$V_{t+1} = \mu V_t - \alpha \nabla_W \mathcal{L}(W_t, \mathcal{D})$$

    Tunning μ

    1. Start with a small value (e.g. 0.5).
    2. After a few iterations, when the optimization is caught in a valley, slowly increase it to a constant value (0.9 or 0.99).
    3. Beware that α and μ interact. At $t = \infty$

$$V = -\frac{\alpha}{1-\mu} \nabla_W \mathcal{L}(W, \mathcal{D})$$

    so, if we increase μ we also may decrease α accordingly

Deep Learning with Keras

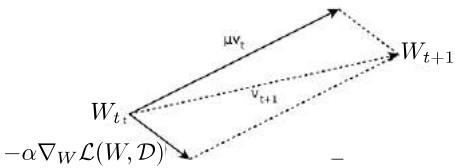
- **Optimizers**
  - **Gradient descent with Nesterov momentum**

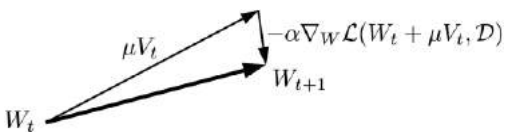
    Weights are updated using a “velocity” term, V

$$W_{t+1} = W_t + V_{t+1}$$

$$V_{t+1} = \mu V_t - \alpha \nabla_W \mathcal{L}(W_t + \mu V_t, \mathcal{D})$$

    computes the gradient after the long “jump” in V_t direction





Deep Learning with Keras

- **Optimizers**
  - **Adaptive learning rate algorithms**

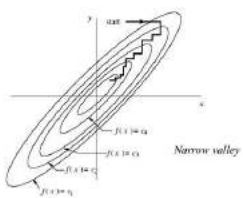
    Scale down the gradient along the steepest direction.
    - **AdaGrad**

$$W_{t+1} = W_t - \alpha \nabla_W \mathcal{L}(W, \mathcal{D}) / \sqrt{s + \epsilon}$$
$$s_{t+1} = s_t + \nabla_W \mathcal{L}(W_t, \mathcal{D}) \cdot \nabla_W \mathcal{L}(W_t, \mathcal{D})$$

    - **RMSProp** idem W, but

$$s_{t+1} = \beta s_t + (1 - \beta) \nabla_W \mathcal{L}(W_t, \mathcal{D}) \otimes \nabla_W \mathcal{L}(W_t, \mathcal{D})$$

    - **Adam** (RMSProp + momentum)



Deep Learning with Keras

- **Optimizers**
  - **Adaptive learning rate algorithms**

    Discussion

    These algoritms
    - Adaptively scale the learning rate of each W_i so that it decays faster for steep directions.
    - They require less tuning of the α parameter
    - **Adam** is the preferred algorithm.

      It is typically used with the default learning paramters.
    - In some problems they may lead to solutions worse than a carefully tuned SGD with Nesterov momentum.

Deep Learning with Keras

- **Optimizers**
  - **Setting optimizer parameters**

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

You can either instantiate an optimizer before passing it to `model.compile()` , as in the above example, or you can call it by its name. In the latter case, the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```