An Introduction to NNs using Keras

Michela Paganini
michela.paganini@cern.ch

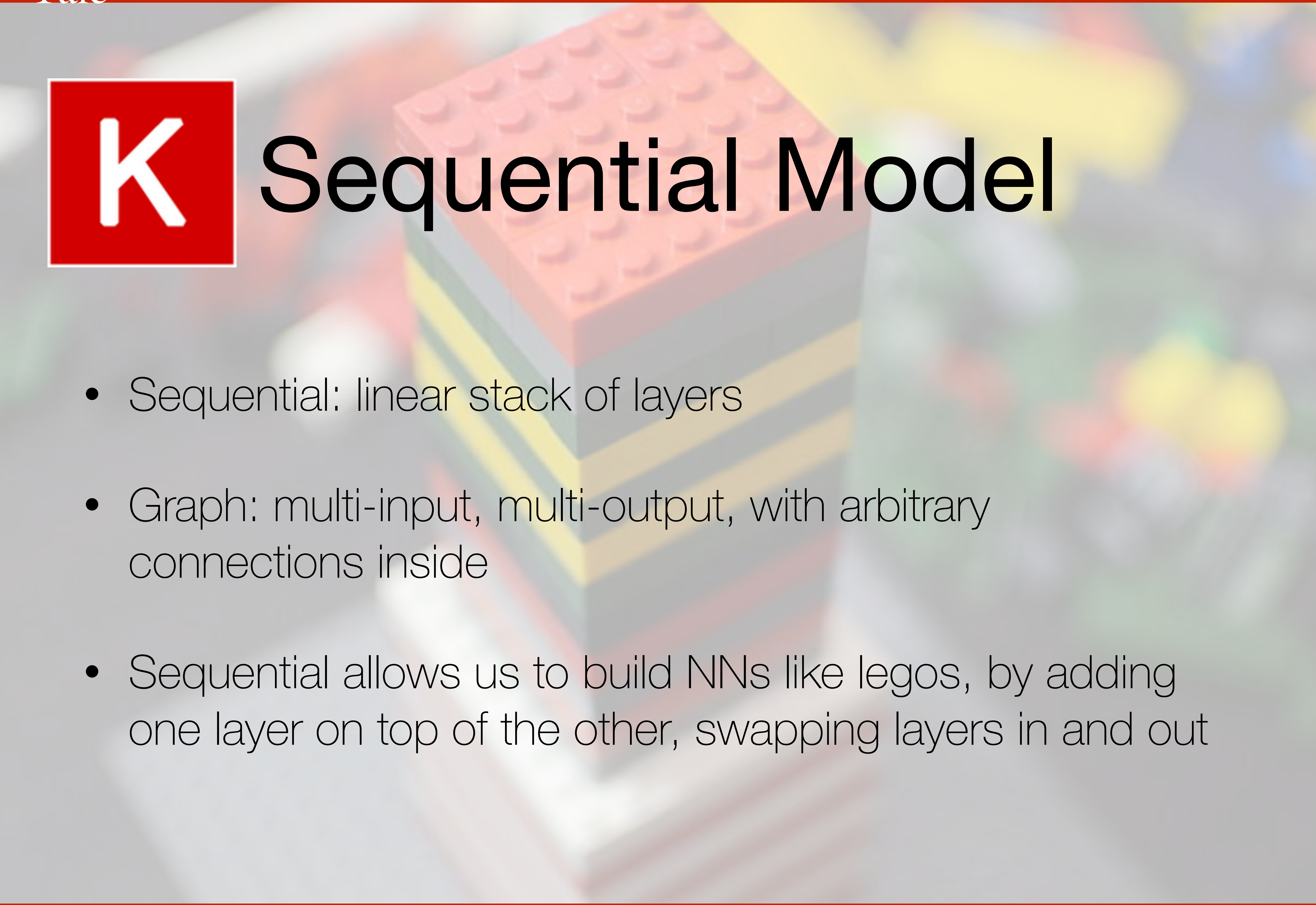
Yale University

Keras

- Modular, powerful and intuitive Deep Learning python library built on Theano and TensorFlow
- Minimalist, user-friendly interface
- CPUs and GPUs
- Open-source, developed and maintained by a community of contributors, and publicly hosted on github
- Extremely well documented, lots of working examples
- Very shallow learning curve —> it is by far one of the best tools for both beginners and experts

Neural Networks

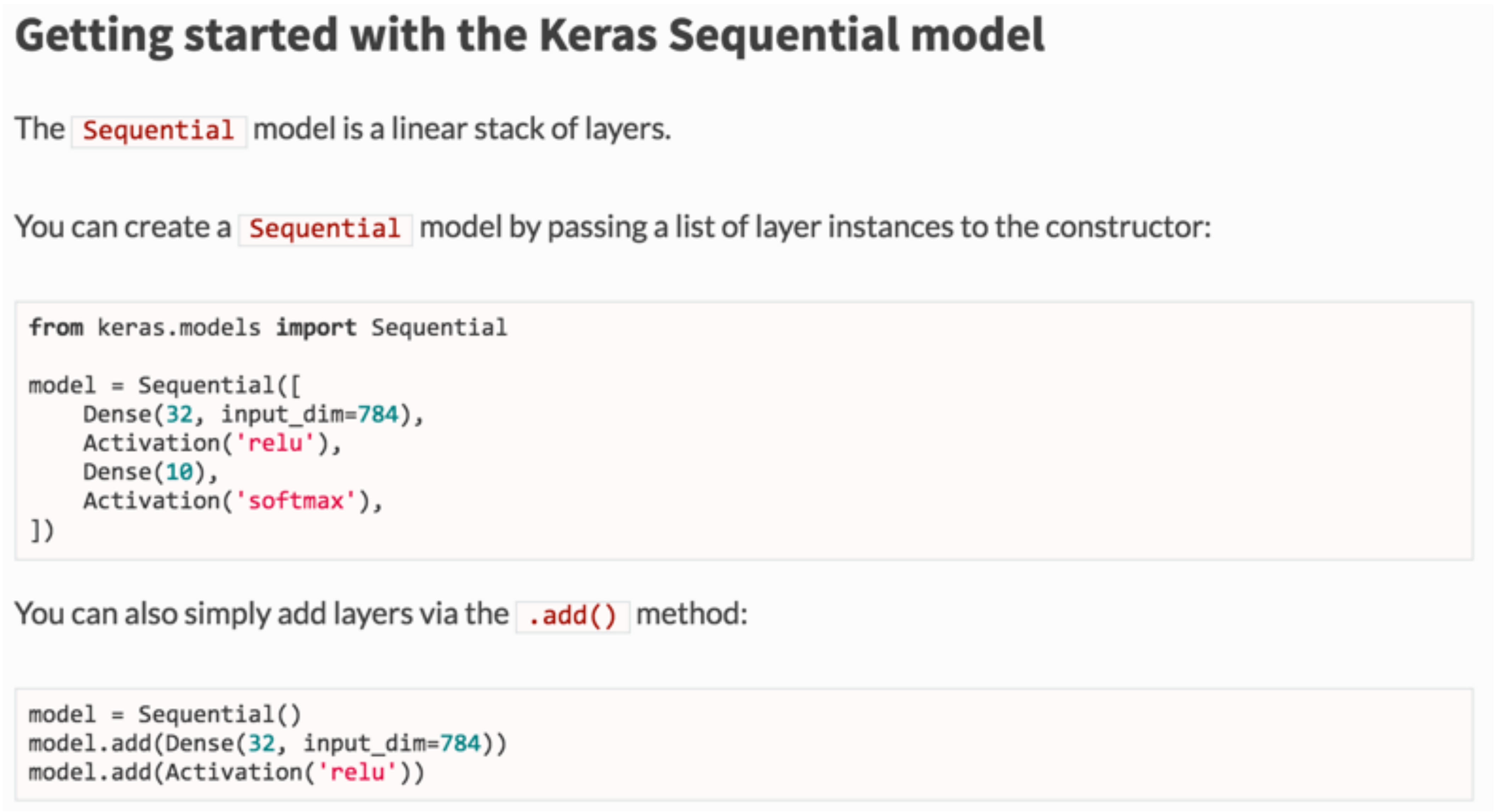
- A stack of tensor operators
- A series of linear and non-linear transformations with the goal of finding the optimal parameters to transform inputs and approximate targets
- For classification and regression
- Deep nets > shallow nets
- Great for raw inputs instead of highly-engineered variables



Sequential Model

- Sequential: linear stack of layers
- Graph: multi-input, multi-output, with arbitrary connections inside
- Sequential allows us to build NNs like legos, by adding one layer on top of the other, swapping layers in and out

Sequential Model



Getting started with the Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential

model = Sequential([
    Dense(32, input_dim=784),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Dense

- Core unit of a Multi-Layer Perceptron
- Linear transformation of the input vector $x \in \mathbb{R}^n$, which can be expressed using the matrix $W \in \mathbb{R}^{m \times n}$ as:

$$u = Wx + b$$

where $b \in \mathbb{R}^m$ is the bias unit

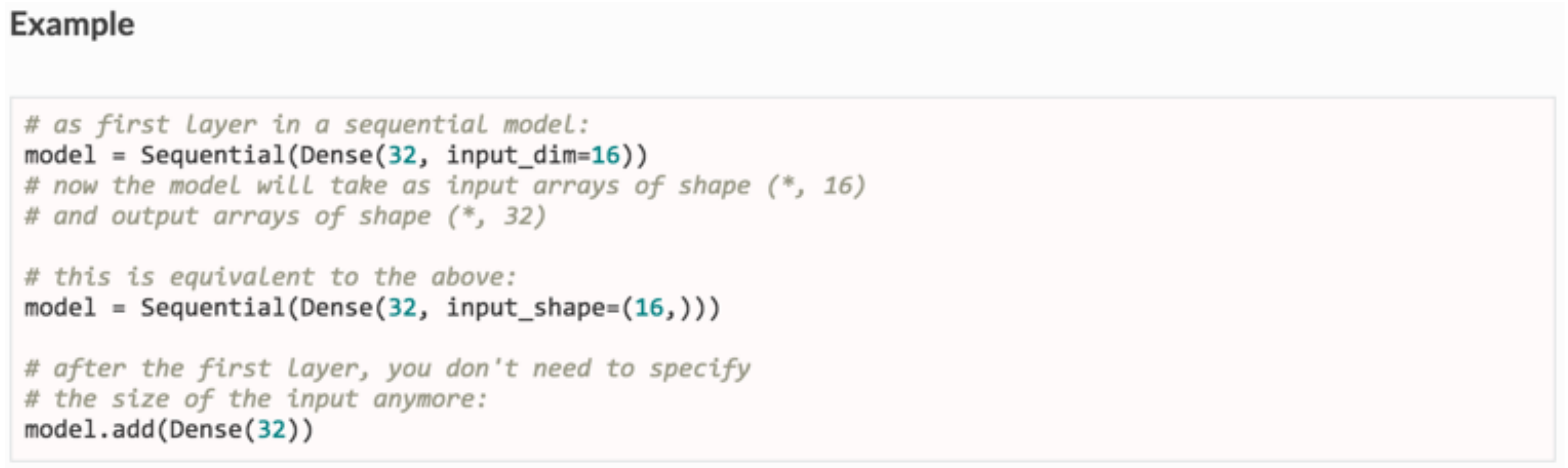
- All entries in both *W* and *b* are trainable

Dense

- In Keras:

```
keras.layers.core.Dense(
        output_dim,
        init='glorot_uniform',
        activation='linear',
        weights=None,
        W_regularizer=None,
        b_regularizer=None,
        activity_regularizer=None,
        W_constraint=None,
        b_constraint=None,
        input_dim=None)
```

- `input_dim` (or `input_shape`) are necessary arguments for the 1st layer of the net



Example

```
# as first layer in a sequential model:
model = Sequential(Dense(32, input_dim=16))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# this is equivalent to the above:
model = Sequential(Dense(32, input_shape=(16,)))

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

Activation

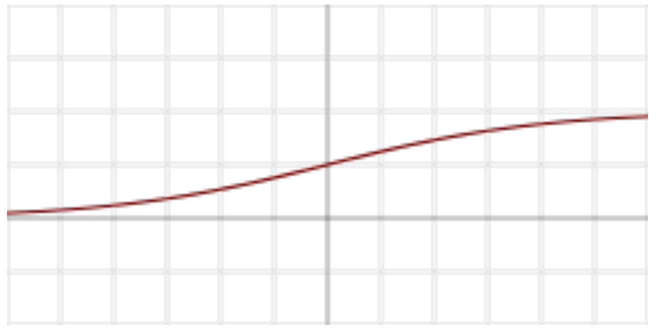
- Mathematical way of quantifying the activation state of a node —> whether it's firing or not
- Non-linear activation functions are the key to Deep Learning
- Allow NNs to learn complex, non-linear transformations of the inputs

Activation

- Some popular choices:

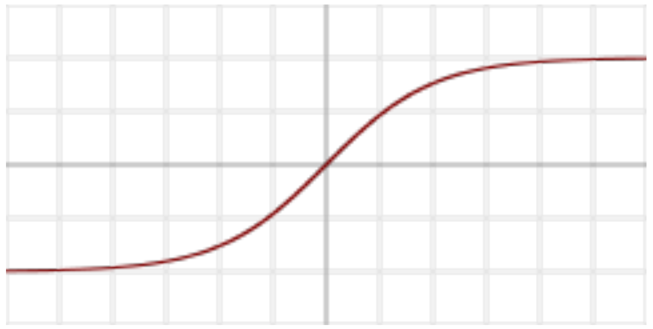
Sigmoid

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



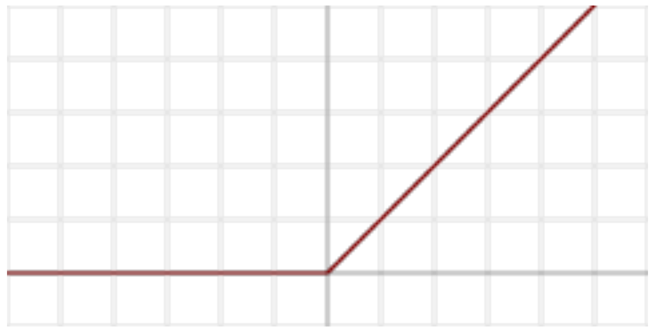
Tanh

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$



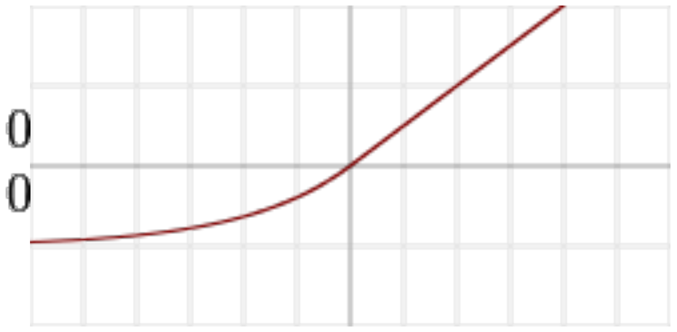
ReLu

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

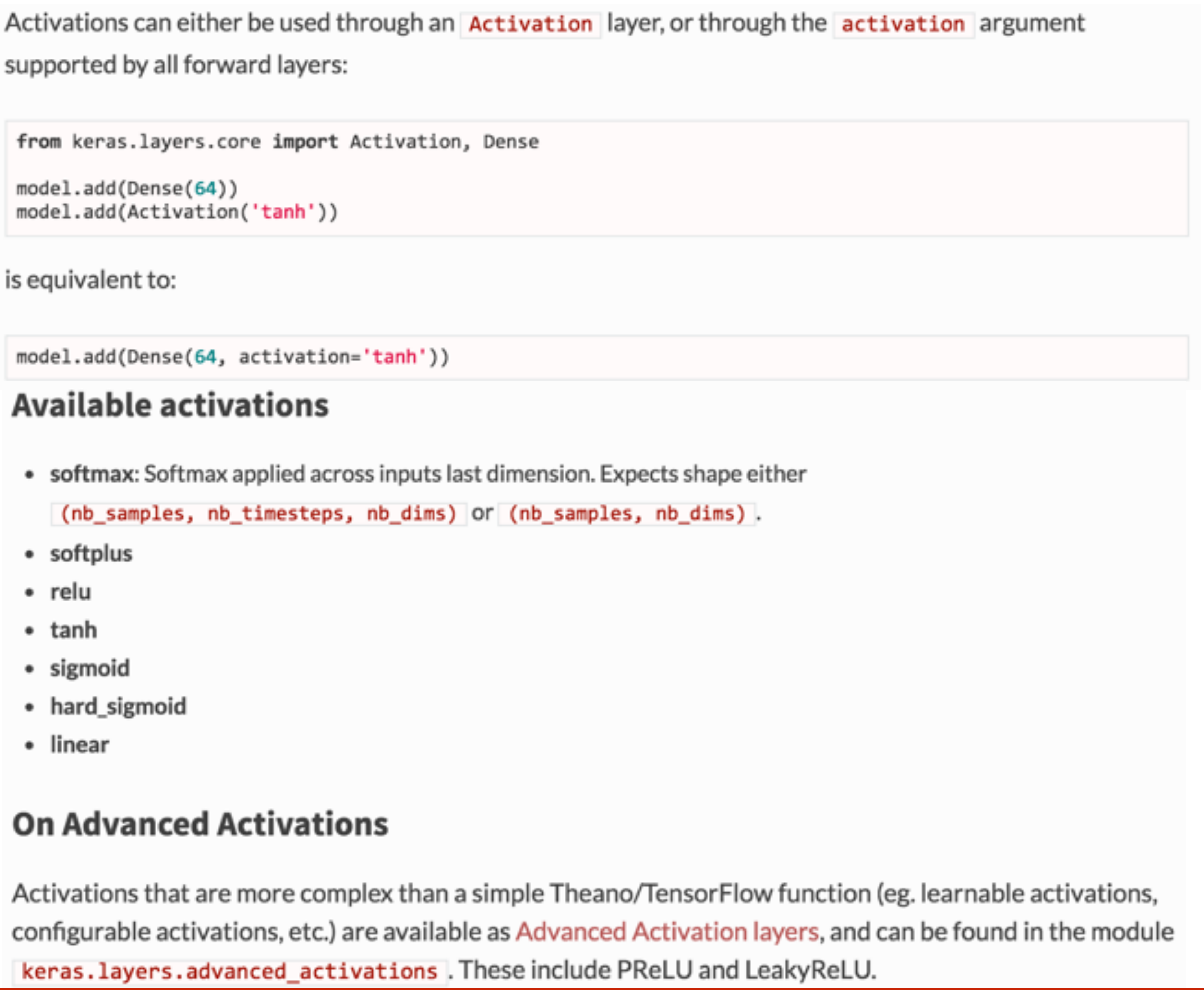


ELu

$$f(x) = \begin{cases} \alpha(e^{x} - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$



Activation



Activations can either be used through an `Activation` layer, or through the `activation` argument supported by all forward layers:

```
from keras.layers.core import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

is equivalent to:

```
model.add(Dense(64, activation='tanh'))
```

Available activations

- **softmax**: Softmax applied across inputs last dimension. Expects shape either `(nb_samples, nb_timesteps, nb_dims)` or `(nb_samples, nb_dims)`.
- **softplus**
- **relu**
- **tanh**
- **sigmoid**
- **hard_sigmoid**
- **linear**

On Advanced Activations

Activations that are more complex than a simple Theano/TensorFlow function (eg. learnable activations, configurable activations, etc.) are available as Advanced Activation layers, and can be found in the module `keras.layers.advanced_activations`. These include PReLU and LeakyReLU.

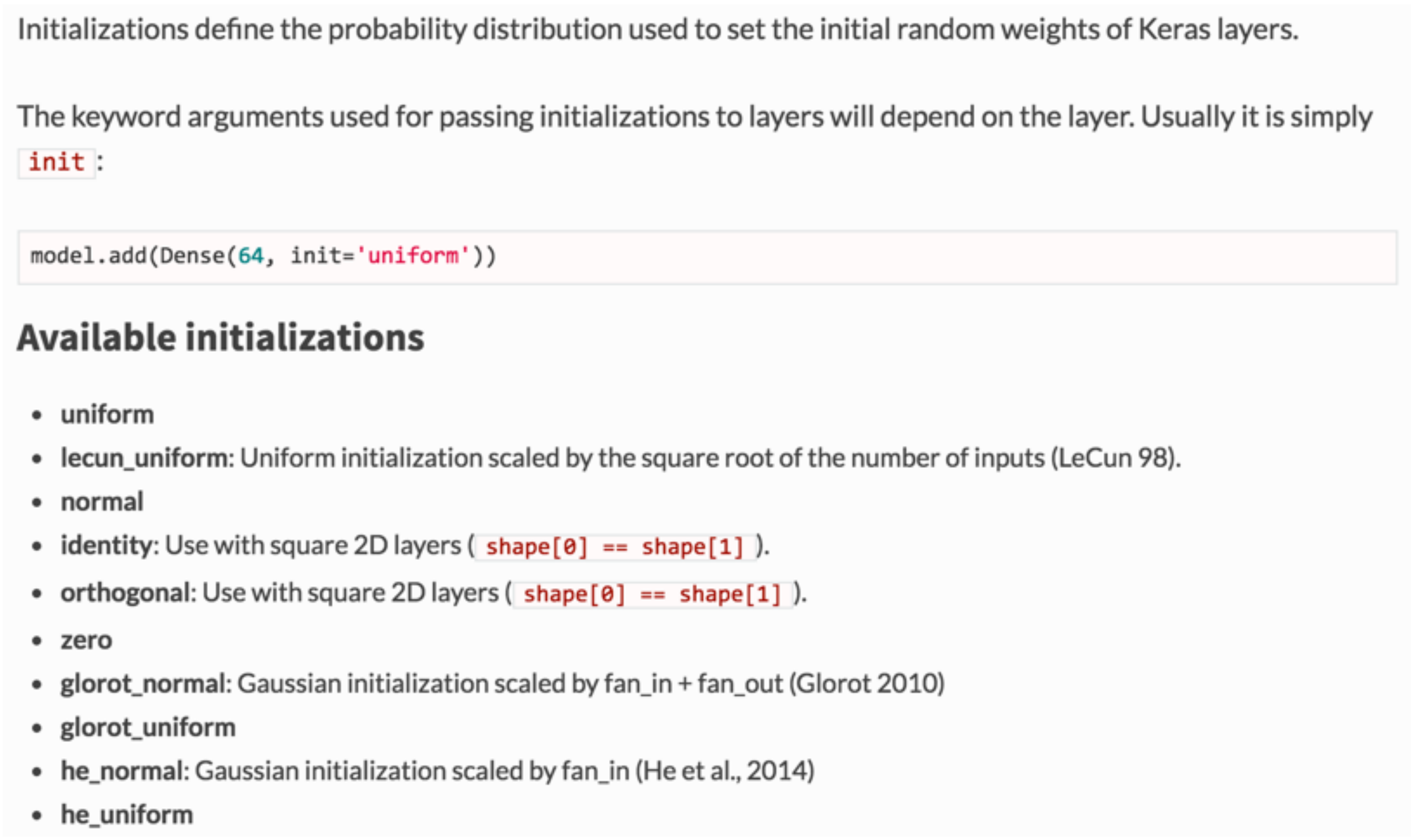
Weights Initialization

- Before training, NN weights need to be initialized to some values
- Initial values must be suitable for the optimization problem to converge as quickly as possible
- Lots of local minima in non-convex optimization problem:

  poor choice of initial weights may lead to convergence to sub-optimal minimum

- Cannot initialize all weights in a layer to a constant
- Big risk = saturation —> very slow learning
- Variance of initialization distribution should be a function of one or both the input and output dimensions —> done automatically by Keras

Weights Initialization



Initializations define the probability distribution used to set the initial random weights of Keras layers.

The keyword arguments used for passing initializations to layers will depend on the layer. Usually it is simply `init`:

```
model.add(Dense(64, init='uniform'))
```

Available initializations

- **uniform**
- **lecun_uniform**: Uniform initialization scaled by the square root of the number of inputs (LeCun 98).
- **normal**
- **identity**: Use with square 2D layers (`shape[0] == shape[1]`).
- **orthogonal**: Use with square 2D layers (`shape[0] == shape[1]`).
- **zero**
- **glorot_normal**: Gaussian initialization scaled by fan_in + fan_out (Glorot 2010)
- **glorot_uniform**
- **he_normal**: Gaussian initialization scaled by fan_in (He et al., 2014)
- **he_uniform**

Forward Propagation

- Sequential series of operations
- Transforms input vector of features *x* through the layers on the NN to obtain the final output $\hat{y}$
- $\hat{y}$ depends not only on the input vector *x* but also on the current values of the weights *W* and *b* in each layer
- A sequential model with L layers is simply computing:

$$\hat{y} = f^{(L)}(z_{L-1})$$

$$z_k = f^{(k)}(z_{k-1})$$

$$z_0 = x$$

- Each *f* is simply a non-linear tensor map
- Result of forward propagation of the same input through the NN will be different during each update
- Output $\hat{y}$ represents the net's current attempt at reproducing the target *y*
- For a FFNN, this step consists of just traversing a linear graph where each node is a tensor op

Loss Function

- Mathematical way of quantifying how much $\hat{y}$ deviates from y
- Dictates how strongly we penalize certain types of mistakes
- Cost of inaccurately classifying an event (“cost function”)
- Used by the optimizer to evaluate performance of NN
- Core of the training: it's the objective of the optimization process, the value that we're trying to minimize (“objective function”)

The goal of optimization is **not** to maximize the area under the ROC curve. That can be used as a metric to evaluate performance and determine the best weight configuration, but it can't be directly maximized during training by your optimization algorithm, because it's a non-differentiable quantity.

Loss Function

- Common loss functions included in Keras:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{Y}_i - Y_i|$$

$$\mathrm{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{Y}_i - Y_i}{Y_i}\right|$$

$$\mathrm{MSLE} = \frac{1}{n}\sum_{i=1}^{n}(\log(\hat{Y}_i + 1) - \log(Y_i + 1))^2$$

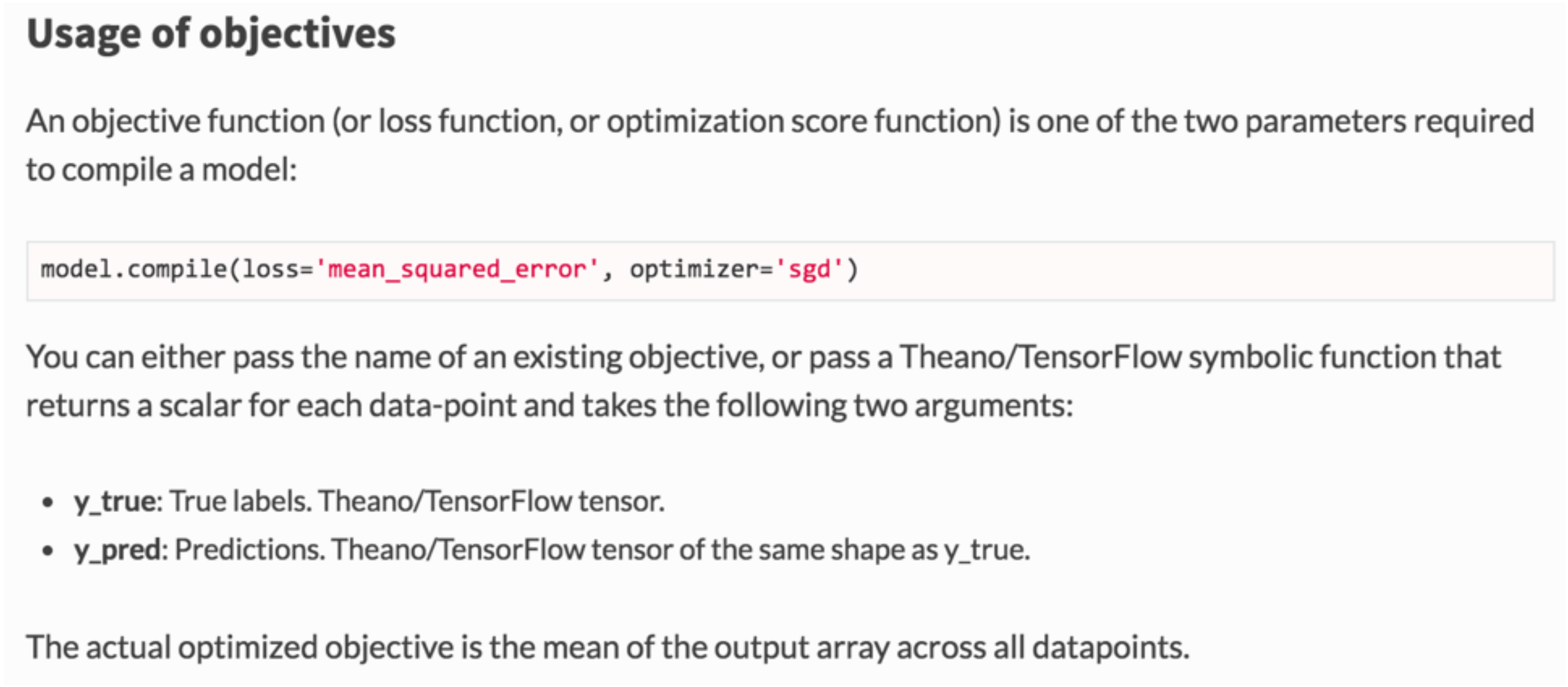
$$\mathrm{Hinge} = \frac{1}{n}\sum_{i=1}^{n}\max(1 - Y_i\hat{Y}_i, 0)$$

$$\mathrm{SH} = \frac{1}{n}\sum_{i=1}^{n}\left(\max(1 - Y_i\hat{Y}_i, 0)\right)^2$$

$$\mathrm{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[Y_i \log \hat{Y}_i + (1 - Y_i)\log\left(1 - \hat{Y}_i\right)\right]$$

- The choice of loss simply resides in understanding what types of errors are or aren't acceptable in the specific problem under consideration

Loss Function



Usage of objectives

An objective function (or loss function, or optimization score function) is one of the two parameters required to compile a model:

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

You can either pass the name of an existing objective, or pass a Theano/TensorFlow symbolic function that returns a scalar for each data-point and takes the following two arguments:

- **y_true**: True labels. Theano/TensorFlow tensor.
- **y_pred**: Predictions. Theano/TensorFlow tensor of the same shape as y_true.

The actual optimized objective is the mean of the output array across all datapoints.

Optimizers

- The learning process is just a global optimization problem, where the weights will take on values such that the loss function is minimized

- Optimizers = methods to quickly take steps in parameter space that are going to be good for minimizing the global loss function

Optimizers

Stochastic Gradient Descent (SGD)

$$\Delta\theta_{(k)} = -\alpha \sum_{i=1}^{n} \nabla L\left(x_i | \theta_{(k)}\right) + \mu\Delta\theta_{(k-1)}$$

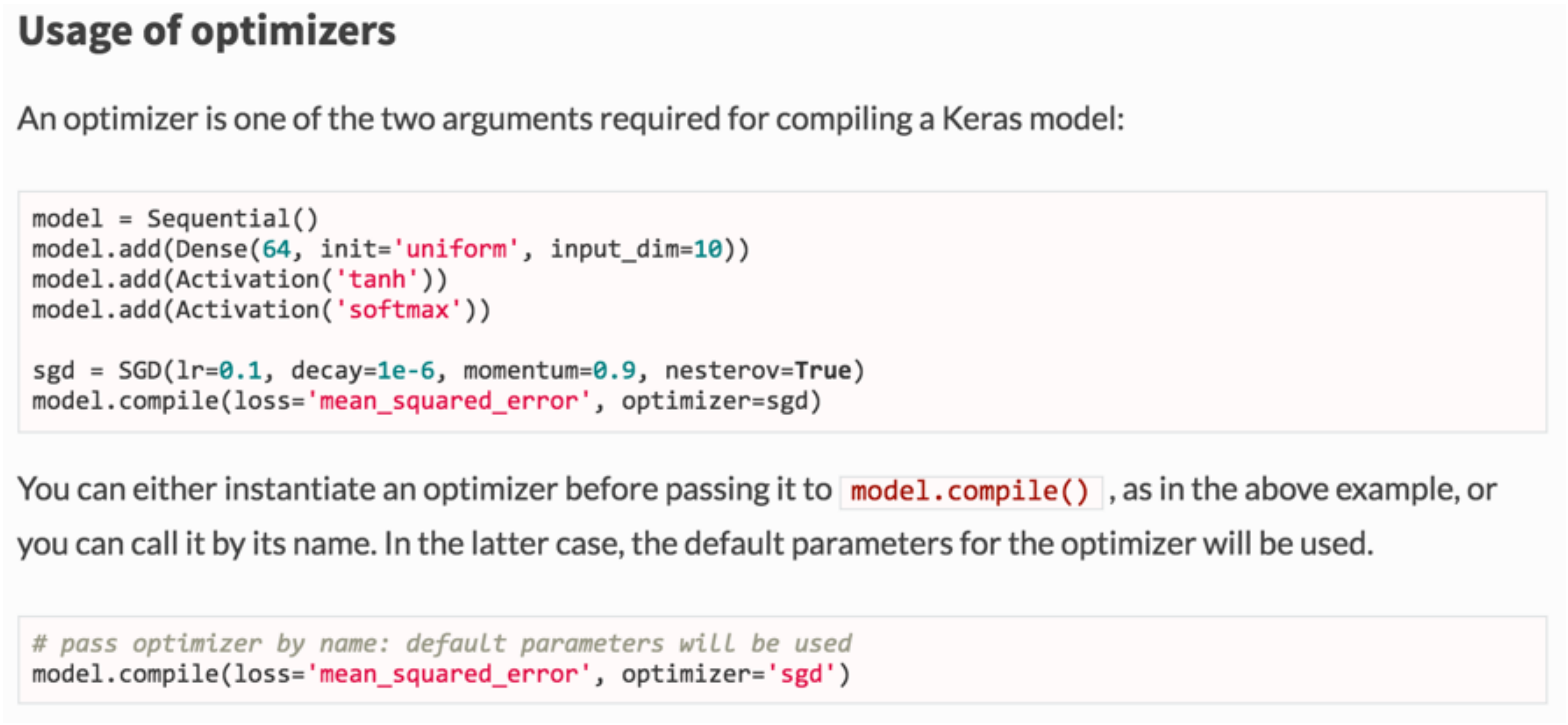
- Learning rate α: how large a step to take
- Momentum μ: how important previous update is in calculating current update
- Decay: exponential rate of change of the learning rate as a function of the number of iteration

  at each iteration: $\alpha_{(k)} = \alpha_{(k-1)}/(1 + \text{decay} * k)$

Adam, Adagrad, Adamax, Adadelta, …

- Smoothing between steps
- Infer 2nd order information about optimization problem, like curvature
- Adaptive optimization algorithms adapt to the landscape and vary the parameters accordingly, performing parameterized scheduling with no human involvement

Optimizers



Usage of optimizers

An optimizer is one of the two arguments required for compiling a Keras model:

```
model = Sequential()
model.add(Dense(64, init='uniform', input_dim=10))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, 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')
```

Back Propagation

- Way of taking derivatives through graphically representable systems
- Loss must be differentiable with respect to any parameter (end-to-end differentiable)
- Modern DL libraries, like Keras, use tensor math libraries such as Theano and TF to do automatic differentiation of symbolically expressed DAGs, simplify operations, and compile logic into the graph
    - The hardest part about deep learning is completely solved!
      You don't have to do anything at all, no matrix derivatives or any ugly stuff like that
- Theano compiles CUDA code directly on the GPU (or machine instructions on CPU, that are specific to the matrix math library installed). If you have both, specify `THEANO_FLAGS=device=gpu` or `cpu`, or change it in your `~/.theanorc`

Regularization

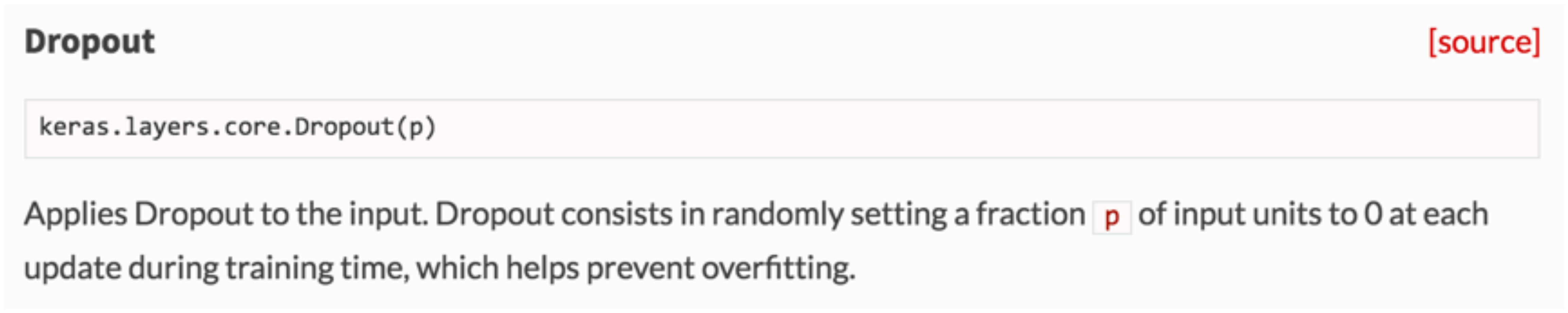
- Series of methods to avoid overfitting
- Mathematical encouragement towards simpler models
- Explicitly penalize weights that get too large
- Two main categories:

  1. norm-based:
     generally uses loss functions of the form $L(x, y, \theta) = L_{pred}(x, y) + \lambda f(\theta)$
     where *f(θ)* is some function of the parameters, and *λ* is the regularization parameter. Common examples of *f* are:
     - $||W||_F$ the Frobenius norm — encourages small weights
     - $||W||_1$ the 1-norm — encourages parameter sparsity
  2. stochastic:
     Main example = Dropout —> randomly sets some percentage of output nodes in a layer equal to zero
     The more commonly used form of regularization in deep nets in the modern era —> fast and lead to simpler, sparser models

Regularization

In Keras:

- Dropout is added in as a layer
  It masks the outputs of the previous layer such that some of them will randomly become inactive and will not contribute to information propagation



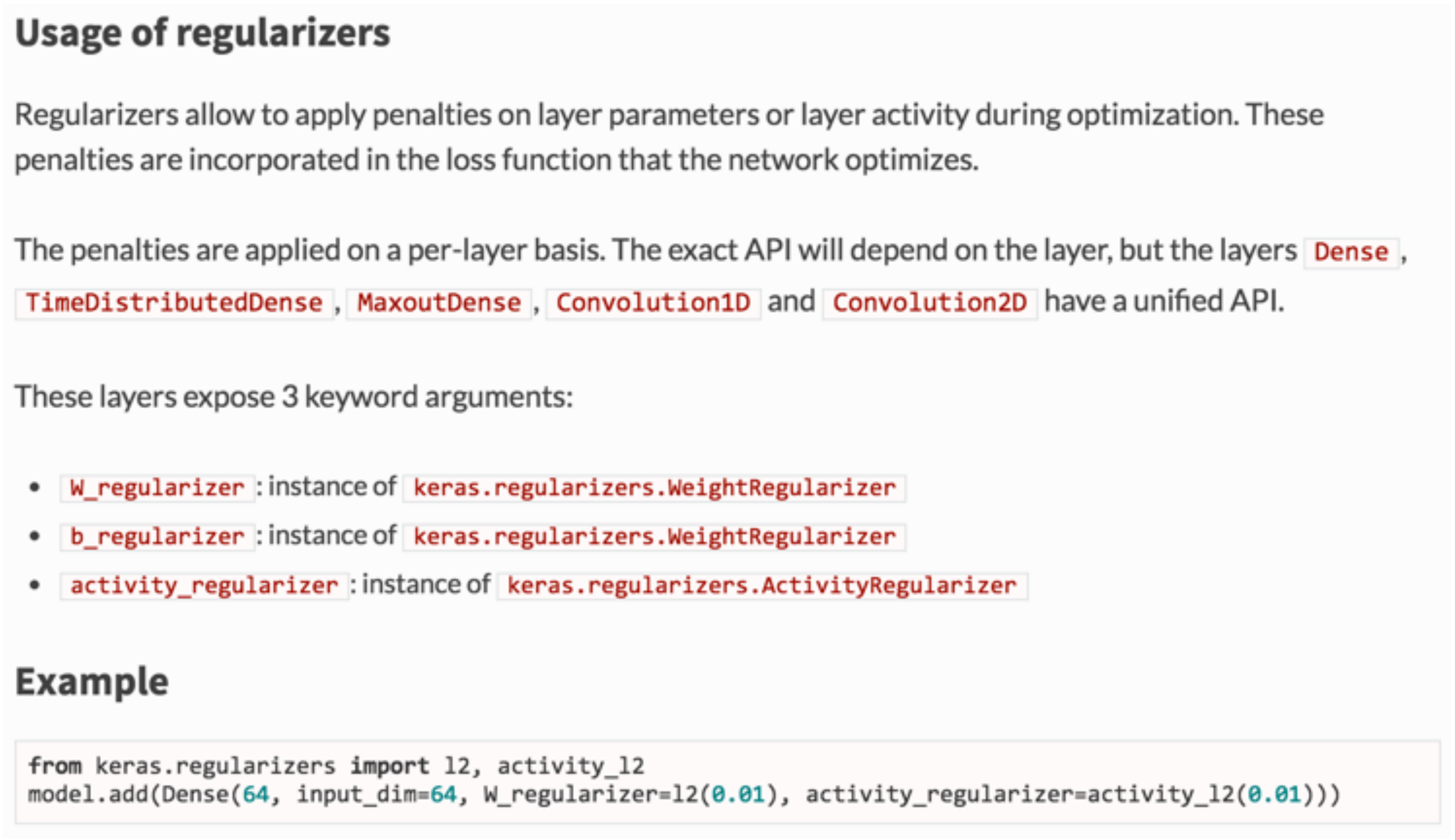
Dropout [source]

```
keras.layers.core.Dropout(p)
```

Applies Dropout to the input. Dropout consists in randomly setting a fraction `p` of input units to 0 at each update during training time, which helps prevent overfitting.

Regularization

- Norm-based regularization is specified per layer
  It represents an added cost associated with the weights of that specific layer being too large in magnitude
  Usually set to zero by default, but can be modified in the definition of the layer



Usage of regularizers

Regularizers allow to apply penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes.

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `TimeDistributedDense`, `MaxoutDense`, `Convolution1D` and `Convolution2D` have a unified API.

These layers expose 3 keyword arguments:

- `W_regularizer`: instance of `keras.regularizers.WeightRegularizer`
- `b_regularizer`: instance of `keras.regularizers.WeightRegularizer`
- `activity_regularizer`: instance of `keras.regularizers.ActivityRegularizer`

Example

```
from keras.regularizers import l2, activity_l2
model.add(Dense(64, input_dim=64, W_regularizer=l2(0.01), activity_regularizer=activity_l2(0.01)))
```

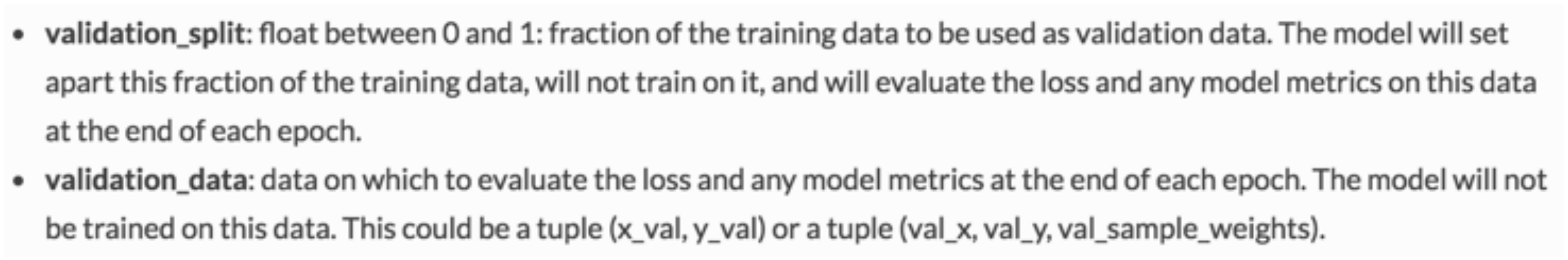
Validation

- NN need to be able to generalize well to new examples
- It should not simply memorize the training set
- Minimizing training loss is not enough —> check validation loss to stop the training!
- But, this way we indirectly fit to the validation set —> overestimate real performance
- Need final, unbiased test on another separate sample (test set) to get accurate estimate of error

Validation

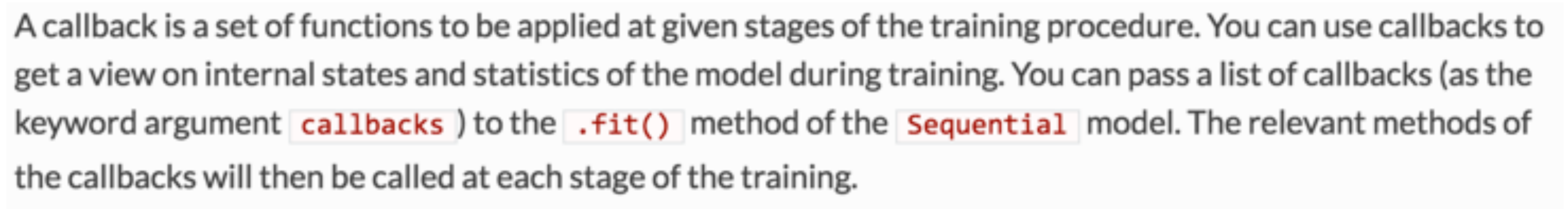
```
# — training command
model.fit( x, y,
           batch_size=32, nb_epoch=10, verbose=1,
           callbacks=[], validation_split=0.0, validation_data=None,
           shuffle=True, class_weight=None, sample_weight=None )
```

where:

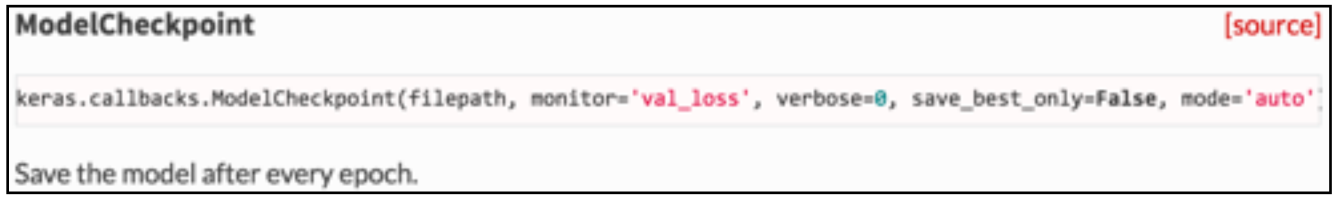


- **validation_split**: float between 0 and 1: fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.
- **validation_data**: data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. This could be a tuple (x_val, y_val) or a tuple (val_x, val_y, val_sample_weights).

and:



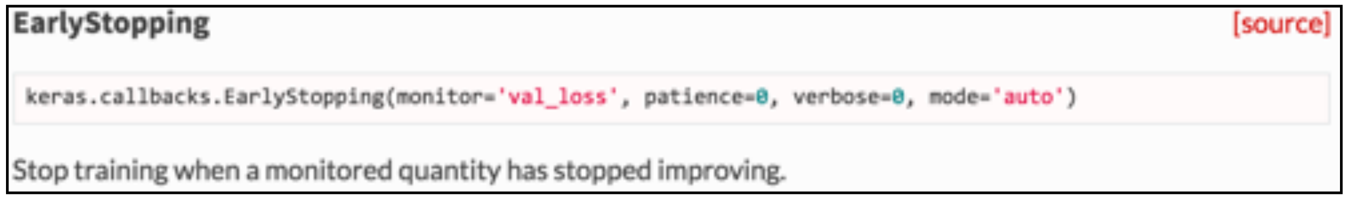
A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training. You can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of the `Sequential` model. The relevant methods of the callbacks will then be called at each stage of the training.



ModelCheckpoint [source]

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, mode='auto'
```

Save the model after every epoch.



EarlyStopping [source]

```
keras.callbacks.EarlyStopping(monitor='val_loss', patience=0, verbose=0, mode='auto')
```

Stop training when a monitored quantity has stopped improving.

Training a Simple NN

```
# -- build net
model = Sequential()
model.add(Dense(16, input_shape= (10,)))
model.add(Activation('relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# -- compile net, specifying optimizer and loss function
model.compile('adam', ‘binary_crossentropy')

# -- train!
model.fit(X_train, y_train)

# -- test
yhat = model.predict(X_test, verbose = True, batch_size = 516)
```

$$\begin{tikzcd}
\boxed{\text{Input (10)}} \arrow[r] & \boxed{\text{Dense(16)}} \arrow[r] & \boxed{\text{ReLu}} \arrow[r] & \boxed{\text{Dense(8)}} \arrow[r] & \boxed{\text{ReLu}} \arrow[r] & \boxed{\text{Dense(4)}} \arrow[r] & \boxed{\text{ReLu}} \arrow[r] & \boxed{\text{Dense(1)}} \arrow[r] & \boxed{\text{Sigmoid}}
\end{tikzcd}$$

Training a Simple NN

```
# -- build net
model = Sequential()
model.add(Dense(16, input_shape= (10,)))
model.add(Activation('relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# -- compile net, specifying optimizer and loss function
model.compile('adam', 'binary_crossentropy')

try:
    # -- train!
    model.fit(X_train, y_train, batch_size=16,
              callbacks = [
                          EarlyStopping(verbose=True, patience=20, monitor='val_loss'),
                          ModelCheckpoint('TestModel-progress', monitor='val_loss', verbose=True, save_best_only=True)],
              nb_epoch=100,
              validation_split = 0.2,
              show_accuracy=True)

except KeyboardInterrupt:
    print 'Training ended early.'

# -- load in best network
model.load_weights('TestModel-progress')
# -- test
yhat = model.predict(X_test, verbose = True, batch_size = 516)
```

stops training after 20 epochs if the validation loss does not improve

saves net if the validation loss improves

manually interrupt training

Congratulations!

- You can now use and understand the coolest visualization tool ever:

  http://playground.tensorflow.org/