Introduction to Keras

Nina Poerner, Dr. Benjamin Roth

CIS LMU München

Keras

- Python-based Neural Network library with three backends:
  - tensorflow, CNTK, Theano
- Very high-level $\approx$ does much of the hard work for you
- ... but powerful enough to implement interesting architectures
- Little redundancy: Architectural details are inferred when possible
- Reasonable defaults (e.g. weight matrix initialization).
- Pre-implements many important layers, loss functions and optimizers
- Easy to extend by defining custom layers, loss functions, etc.
- Documentation: `https://keras.io/`

Keras vs. PyTorch

	Keras	PyTorch
graph definition	static	dynamic
defining simple NNs	🙂	😐
defining complex NNs	😐	😐
training and evaluation	🙂	😐
convenience (callbacks, ...)	🙂	☹️*
debugging + printing	☹️	🙂

*The `ignite` package contains PyTorch-compatible callbacks

Installation

```
conda install keras
# or
pip3 install keras
# or
git clone https://github.com/keras-team/keras
cd keras
python3 setup.py install
```

Choosing a backend

- In most cases, your code should work with any of the three backends
- Recommended: tensorflow
- To change the backend temporarily, set environment variable before executing any script:

  ```
  KERAS_BACKEND=tensorflow
  ```

- To change the backend permanently, edit ~/.keras/keras.json

  ```
  {
      "floatx": "float32",
      "image_dim_ordering": "tf",
      "epsilon": 1e-07,
      "backend": "tensorflow"
  }
  ```

The Sequential Model

- Sequential: A model where every layer has exactly one input tensor and one output tensor. (The name has nothing to do with RNNs!)
- Example: Multi-layer perceptron with input size 10, hidden size 20, output size 1

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
hidden_layer = Dense(units = 20, input_shape = (10,), activation = "relu")
model.add(hidden_layer)
# first layer needs an input_shape

output_layer = Dense(units = 1, activation = "sigmoid")
model.add(output_layer)
# other layers can infer their input shape (why?)

print([w.shape for w in model.get_weights()])
[(10, 20), (20,), (20, 1), (1,)]
print(model.predict(np.random.random(size = (2,10))))
[[0.4927521 ]
 [0.45954984]]
```

Defining a topic classifier in under 10 lines of code

```
from keras.layers import LSTM, Dense, Embedding
from keras.models import Sequential

VOCAB_SIZE, EMB_SIZE, HIDDEN_SIZE, NUM_TOPICS = 1000, 100, 200, 50
x = np.random.randint(size = (4, 80), low = 0, high = VOCAB_SIZE))

model = Sequential()
embedding_layer = Embedding(input_dim = VOCAB_SIZE, output_dim = EMB_SIZE)
model.add(embedding_layer)
print(model.predict(x).shape)
(4, 80, 100)

lstm_layer = LSTM(units = HIDDEN_SIZE)
model.add(lstm_layer)
print(model.predict(x).shape)
(4, 200)

output_layer = Dense(units = NUM_TOPICS, activation = "softmax"))
model.add(output_layer)
print(model.predict(x).shape)
(4, 50)
```

Other useful layers

- `Conv1D`: 1D Convolution (for text)
- `Conv2D`: 2D Convolution (for pictures)
- `Bidirectional` wrapper: Applies RNNs bidirectionally:
  ```
  layer = Bidirectional(GRU(units = HIDDEN_DIM))
  ```
- `TimeDistributed` wrapper: Applies the same layer to all time steps in parallel (e.g., for POS tagging)
  ```
  layer = TimeDistributed(Dense(units = NUM_CLASSES, activation = "softmax"))
  ```
- `Dropout`: Randomly sets n% of neurons to zero (a form of regularization)
- ...

Compilation

- `compile` adds loss function and optimizer to Neural Network
- `compile` must be called before training

```
model.compile(loss = "categorical_crossentropy",
        optimizer = "sgd",
        metrics = ["accuracy"])
```

- metric: a “loss function” that is not used for training
  - all losses can be metrics, but not all metrics can be losses (e.g., accuracy)

Available loss functions & metrics

- Mean squared error, mean absolute error
- binary crossentropy (for sigmoid, expects vectors of zeros and ones)
  - e.g., `Y`$= [[0, 1, 0], [1, 1, 1]]$
- categorical crossentropy (for softmax, expects one-hot vectors)
  - e.g., `Y`$=[[0, 0, 1], [1, 0, 0]]$ (one-hot)
- sparse categorical crossentropy (for softmax, expects indices)
  - e.g., `Y`$= [[2], [0]]$ (sparse)
- cosine proximity
- KL divergence
- accuracy (as metric only)
- ...

DIY losses & metrics

```
def myloss(y_true, y_pred):
        loss = # do something with y_true, y_pred
        return loss

model.compile(loss=myloss, optimizer = "sgd")

# as metric:
model.compile(loss="mean_squared_error", optimizer = "sgd", metrics = [myloss])
```

Optimizers

- Available optimizers: SGD, Adam, RMSProp...

```
model.compile(loss = "categorical_crossentropy", optimizer = "sgd")
model.compile(loss = "categorical_crossentropy", optimizer = "adam")

# or customize your optimizer:
from keras.optimizers import SGD
customsgd = SGD(lr = 0.006, momentum = True)
model.compile(loss = "categorical_crossentropy", optimizer = customsgd)
```

Training

- `fit` receives numpy tensors `X` and `Y`
- Their shape must match expected input and output shapes
- `fit` returns history object with losses/metrics over epochs
- By default, `fit` shuffles the training data

```
print(model.input_shape)
(None, None) # (batchsize, timesteps). None means that any size > 0 is okay.
print(model.output_shape)
(None, 50) # (batchsize, timesteps, output_dim)
X, Y = # load_training_data()
print(X.shape)
(20, 30)
print(Y.shape)
(20, 50)
history = model.fit(X, Y, epochs = 5, shuffle = True)
print(history.history["loss"])
[0.317502856254577637, 0.26498502135276794, ...]
```

Evaluation

```
X, Y = # load_dev_data()
results = model.evaluate(X, Y)
for name, number in zip(model.metrics_names, results):
        print(name, number)
```

```
loss 0.29085057973861694
acc 0.7510684013366699
```

Validation during training

```
X_train, Y_train = # load_train_data()
X_dev, Y_dev = # load_dev_data()
history = model.fit(X_train, Y_train, epochs = 5,
        validation_data = (X_dev, Y_dev))
# validation loss history in history.history["val_loss"]

#or use 10% of X_train, Y_train as validation set
history = model.fit(X_train, Y_train, epochs = 5, validation_split = 0.1)
```

Callbacks

- `EarlyStopping`: Stop training when a loss/metric stops improving
- `ModelCheckpoint`: Save model at regular intervals
- `ReduceLROnPlateau`: Reduce learning rate when loss stops improving
- ...

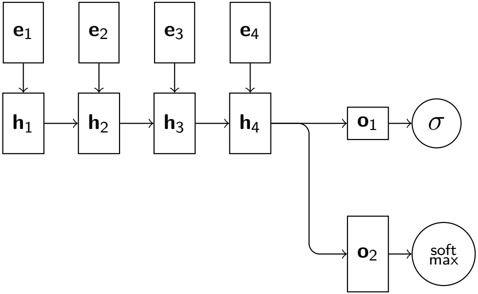
```
from keras.callbacks import EarlyStopping, ModelCheckpoint

earlystop = EarlyStopping(monitor = "val_acc", patience = 5)
# stop training if validation accuracy did not improve for 5 epochs

checkpoint = ModelCheckpoint("./mymodel.h5",
        save_best_only = True, monitor = "val_acc")
# save model after epochs with improved validation accuracy improves

model.fit(X, Y, validation_split = 0.1, epochs = 100000,
        callbacks = [earlystop, checkpoint])
```

Keras: functional API

- Layers in Sequential can have only one input, one output
- Functional API generalizes to multiple in-/outputs
- Class `Input`: placeholder for input data, needs to know its own shape
- "Split" information flow by passing one tensor to multiple layers
- "Merge" information flow with merge functions:
  - `concatenate`, `add`, `dot`...

Keras: functional API – examples

- Example: Two outputs
  - Use a single LSTM with two different loss functions
  - e.g., to predict topic (1-of-N) and sentiment polarity (binary)

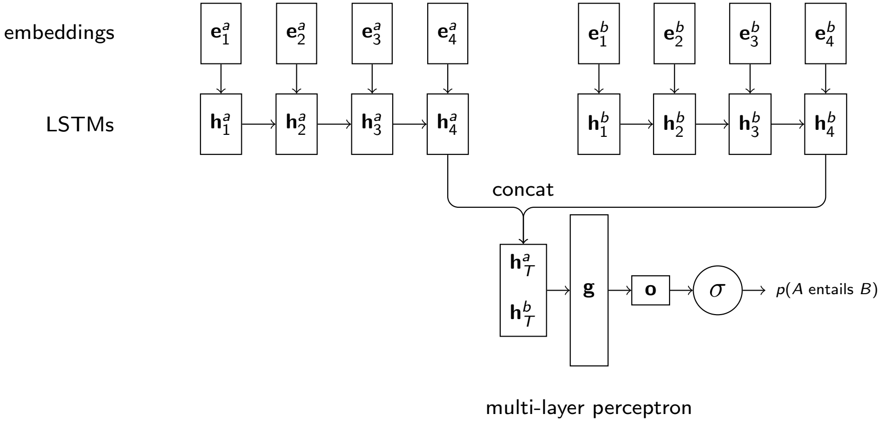


Keras: functional API – example

```
from keras.layers import Input, Embedding, LSTM, Dense, concatenate
from keras.models import Model

embedding_layer, lstm_layer = # Embedding(...), LSTM(...)
i_sentence = Input(input_shape = (None,))
# None means: Any sentence length > 0 is fine

embedded = embedding_layer(i_sentence) # call layer like a function
h_T = lstm_layer(embedded)

softmax_layer = Dense(units = 20, activation = "softmax")
sigmoid_layer = Dense(units = 1, activation = "sigmoid")

o_polarity = sigmoid_layer(h_T)
o_topic = softmax_layer(h_T)

model = Model(inputs = [i_sentence], outputs = [o_polarity, o_topic])
model.compile(optimizer = "sgd",
        loss = ["binary_crossentropy", "categorical_crossentropy"])
# order in model outputs and loss must match!

X, Y_sigmoid, Y_softmax = # load_training_data()
model.fit(x=X, y=[Y_sigmoid, Y_softmax])
```

Keras: functional API – example

- Two inputs:
  - Encode sentences A and B with LSTMs, then predict if A entails B



Keras: functional API – example

```
from keras.layers import Input, Embedding, LSTM, Dense, concatenate
from keras.models import Model

embedding_layer_for_A, lstm_layer_for_A = # Embedding(...), LSTM(...)
embedding_layer_for_B, lstm_layer_for_B = # Embedding(...), LSTM(...)

i_A = Input(input_shape = (None,)) # None: Sentence length can vary
i_B = Input(input_shape = (None,))

h_a_T = lstm_layer_for_A(embedding_layer_for_A(i_A))
h_b_T = lstm_layer_for_B(embedding_layer_for_B(i_B))
concat = concatenate([h_a_T, h_b_T]) # merge by concatenation

hidden_layer = Dense(units = 200, activation = "relu")
output_layer = Dense(units = 1, activation = "sigmoid")

o_entailment = output_layer(hidden_layer(concat))
model = Model(inputs = [i_A, i_B], outputs = [o_entailment])

model.compile(loss = "binary_crossentropy", optimizer = "sgd")
X_A, X_B, Y = # load_training_data()
model.fit(x=[X_A, X_B], y=Y)
# order in x, and model inputs must match!
```

DIY layers

- If the layer has no weights and does something simple: `Lambda` layer
- Implement custom function using the backend
- If output size $\neq$ input size, implement a shape change function
- Backend documentation: https://keras.io/backend/

```
import keras.backend as K
from keras.layers import Lambda

def myfunction(x):
        """A function that returns x concatenated with x**2"""
        x_squared = K.square(x)
        return K.concatenate([x, x_squared], axis = -1)

def myfunction_shape(shape):
        return shape[:-1] + (shape[-1]*2,) # last dimension doubles in size

layer = Lambda(myfunction, output_shape = myfunction_shape)
```

DIY layers

- Custom layers must inherit from `Layer`
- Reimplement `call()` and `build()`
- If the output has a different shape from the input, reimplement `compute_output_shape`

```
from keras.layers import Layer
from keras.initializers import Ones

class ScalarLayer(Layer):
        """A layer that scales input by a trainable scalar"""
        def build(self, input_shape):
                self.scalar = self.add_weight(shape = (1,),
                        name = "scalar",
                        initializer = Ones())
        def call(self, inputs):
                return inputs * self.scalar[0]

layer = ScalarLayer(input_shape = (None,))
```

Why masking?

- Frequent problem: sentences of varying length
- To combine many sentences into a matrix, we must trim long sentences or pad short ones
- `pad_sequences` preprocessing function: pad shorter sentences with zeros at the beginning or end
- Problem 1: RNN may forget information while reading long sequence of zeros
- Problem 2: Label padding is trivial to predict – overestimated performance!

Masking

- Mask: Tensor of ones and zeros that accompanies an input
- 1: time step is valid
- 0: time step is masked

this	is	a	long	sentence
1	1	1	1	1
short	sentence	pad	pad	pad
1	1	0	0	0

- RNN states skip masked inputs
- Masked outputs are ignored by loss/metric

Masking

```
embedding_layer = Embedding(input_dim = VOCAB_SIZE,
        output_dim = EMBEDDING_SIZE, mask_zero = True)
# output of embedding_layer is masked for timesteps where x=0

# or:
masking_layer = Masking(9)
# output of masking_layer is masked for timesteps where x=9
```

Why generators?

- Previously:
  - Collect all training data in variables `X`, `Y`
  - Call `model.fit(X, Y)`
- What if `X`, `Y` don't fit into memory?

```
def data_generator():
        while True:
                x_batch, y_batch = # read data batch from disk
                yield(x_batch, y_batch)

steps = # calculate number of batches to expect per epoch
model.fit_generator(data_generator(), steps_per_epoch = steps)
```

- Similar: `evaluate_generator`, `predict_generator`
- Validation data can be generator or tuple of arrays
- Caveat: Shuffling must be done by generator

Why sample weights?

- Previously: $L(\mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_n L(\mathbf{x}_n, y_n)$
- ... but not all training samples are created equal.
- Possible scenarios:
  - We trust some annotators more than others
  - We have a mix of manually annotated data and data derived by some heuristic ("distant supervision")
  - We have a mix of out-of-domain and in-domain data
- Sample weighting: $L(\mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_n w_n L(\mathbf{x}_n, y_n)$
- w_n is a scalar chosen by the developer (not learned!)

```
X, Y = # training data; shape = (num_samples, ...)
W = # vector of sample weights; length =  num_samples

model.fit(X, Y, sample_weight = W)

# with generator:

def generator():
        while True:
                x_batch, y_batch, w_batch = # get batch
                yield(x_batch, y_batch, w_batch)
```

Why class weights?

- Scenario: We have a skewed class distribution and don't want the model to focus on the over-represented classes
- Class weighting: $L(\mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_n c(y_n) L(\mathbf{x}_n, y_n)$
- c is a function that assigns a scalar weight to every class. c is chosen by developer (not learned!). Possible choice: inverse class frequency.

```
X, Y = # training data

# 10 class 0 samples for every class 1 sample
model.fit(X, Y, class_weight = {0: 0.1, 1: 1})
```