Introduction to Keras

Nina Poerner, Dr. Benjamin Roth

CIS LMU München

Outline

1. Introduction
2. The Sequential Model
3. Compiling
4. Training, Evaluation, Validation

Outline

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"
}
```

Outline

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
```

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?)
```

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)
```

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)
```

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)
- ...

Outline

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)
```

Outline

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"])
[0.31750285625457763 7, 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])
```