SDU Summer School

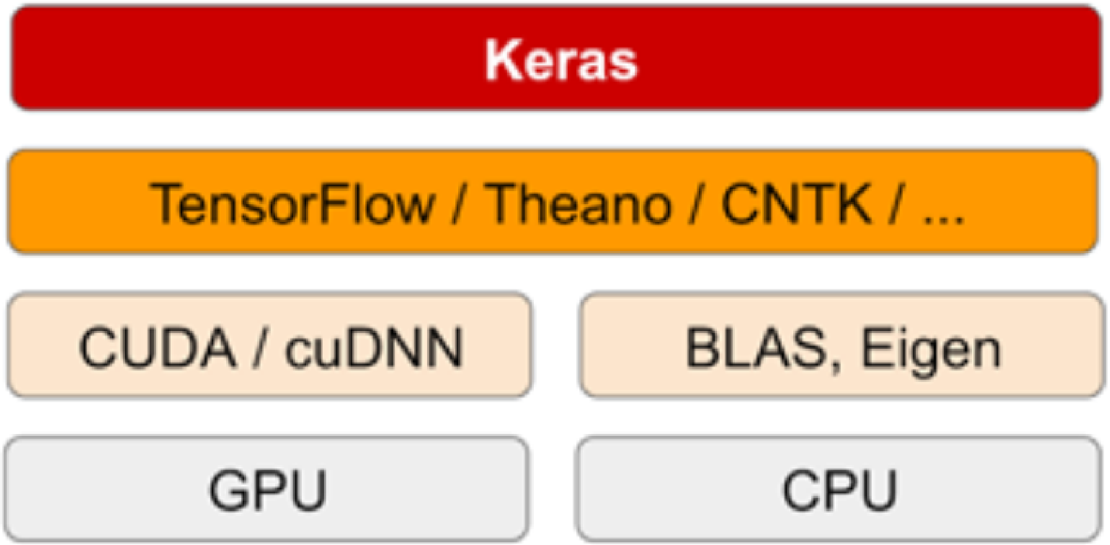
Deep Learning

Summer 2018

Introduction to KERAS

UNIVERSITY OF SOUTHERN DENMARK.DK

What is Keras?



- Keras is a high-level API providing easy to use elements for deep learning
- Can work with several backends
- Programs can easily deployed on CPUs, GPUs without changing the code

Who makes Keras? Contributors and backers

633 contributors

Google

Microsoft

NVIDIA

aws

The Keras user experience

- Keras API is easy to understand.
  - Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- Easy to learn.
  - As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.
- This ease of use does not come at the cost of reduced flexibility:
  - Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows.

Multi-Backend, Multi-Platform

- Develop in Python, R
  - On Unix, Windows, OSX
- Run the same code with…
  - TensorFlow
  - CNTK
  - Theano
  - MXNet
  - PlaidML
  - ??
- Run on CPU, NVIDIA GPU, AMD GPU, TPU...

How to use Keras: An introduction

Three API Styles

- **The Sequential Model**
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases
- **The functional API**
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases
- **Model subclassing**
  - Maximum flexibility
  - Larger potential error surface

The Sequential API

```
import keras
from keras import layers

model = keras.Sequential()
model.add(layers.Dense(20, activation='relu', input_shape=(10,)))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.fit(x, y, epochs=10, batch_size=32)
```

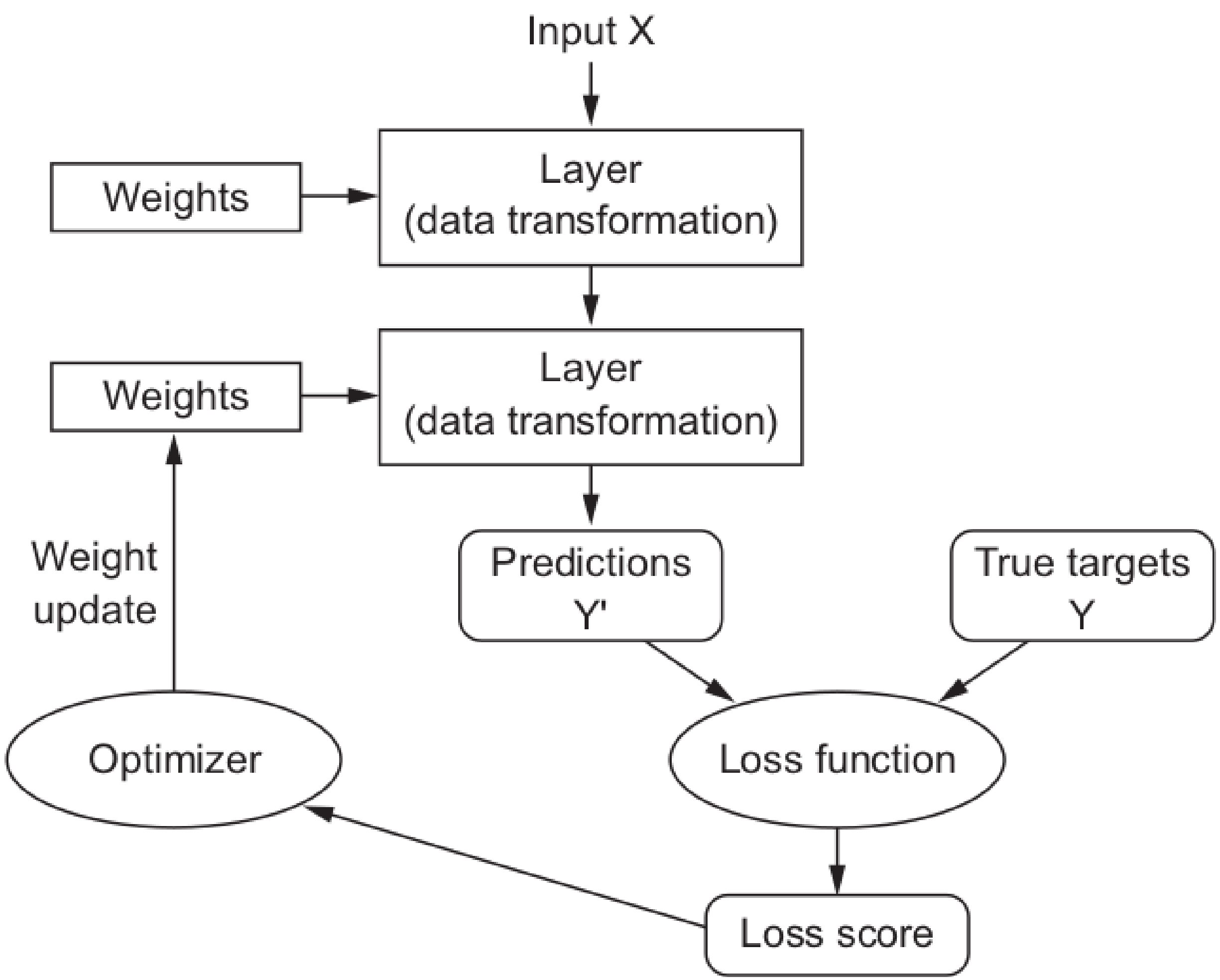
The Functional API

```
import keras
from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
model.fit(x, y, epochs=10, batch_size=32)
```

Model subclassing

```
import keras
from keras import layers

class MyModel(keras.Model):

    def __init__(self):
        super(MyModel, self).__init__()
        self.dense1 = layers.Dense(20, activation='relu')
        self.dense2 = layers.Dense(20, activation='relu')
        self.dense3 = layers.Dense(10, activation='softmax')

    def call(self, inputs):
        x = self.dense1(x)
        x = self.dense2(x)
        return self.dense3(x)

model = MyModel()
model.fit(x, y, epochs=10, batch_size=32)
```

Alrighty, let's put it together



Defining the Model

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

- The same model defined using the functional API:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Options for Layers

- Core Layers
- Convolutional Layers
- Pooling Layers
- Locally-connected Layers
- Recurrent Layers
- Embedding Layers
- Merge Layers
- Normalization Layers
- Noise layers

Options for Layers

- The core layers perform the most basic operations
- They are enough to built FFN networks
- **Core Layers**
  - Dense Layers
  - Activation Layer
  - Dropout Layer
  - Reshape
  - Permute

Dense Layer

```
keras.layers.Dense(units,
    activation=None, #Standard: use linear output
    use_bias=True, #Add a bias vector
    kernel_initializer='glorot_uniform’, #How to initialize the
                                          weigths
    bias_initializer='zeros’, #How the biases
    kernel_regularizer=None, #For example, apply L2 regularization
    bias_regularizer=None, #For example, apply L2 regularization
    activity_regularizer=None, #For example, apply L2 regularization
    kernel_constraint=None, #For example, non-negative constraint
    bias_constraint=None #For example, non-negative constraint
)
```

Activation Function

```
model.add(Dense(64))
model.add(Activation('tanh’))
#This is equivalent to:
model.add(Dense(64, activation='tanh'))
```

- Available Activations:
  - softmax
  - elu: (Exponential linear unit.)
    - x if $x > 0$ and $\alpha * (\exp(x) - 1)$ if $x < 0$.
  - selu: Scaled Exponential Linear Unit
  - softplus
    - $\log(\exp(x) + 1)$
  - relu
    - relu(x, alpha=0.0, max_value=None)
  - sigmoid
  - tanh

Compiling the Model

```
from keras import optimizers

model.compile(
    optimizer=optimizers.RMSprop(lr=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

- Before training a model, you need to configure the learning process, which is done via the compile method, defining
  - **An optimizer**. This could be the string identifier of an existing optimizer (such as rmsprop or adagrad), or an instance of the Optimizer class
  - **A loss function**. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function, or it can be an objective function.
  - **A list of metrics.** A metric could be the string identifier of an existing metric or a custom metric function.

Examples: Compiling Models

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

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

What does compile do?

- Compile defines the loss function, the optimizer and the metrics. That's all.
  - It has nothing to do with the weights and you can compile a model as many times as you want without causing any problem to pretrained weights.
  - You need a compiled model to train (because training uses the loss function and the optimizer). But it's not necessary to compile a model for predicting.
  - Do you need to use compile more than once? Only if:
    - You want to change one of these:
      - Loss function
      - Optimizer / Learning rate
      - Metrics
    - You loaded (or created) a model that is not compiled yet. Or your load/save method didn't consider the previous compilation.
- Consequences of compiling again:
  - If you compile a model again, you will lose the optimizer states.

Loss Functions

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- …

Metrics

- Can be any of the loss functions
- Some standard metrics like
  - F1
  - Precision
  - Recall
- have been removed in Keras 2.0
  - “These are all global metrics that were approximated batch-wise, which is more misleading than helpful.”

Train the Model

```
fit(x=None, y=None, # Input and desired outcome
    batch_size=None, # Number of samples per gradient update. If
                          none, it defaults to 32
    epochs=1, # Number of runs over the complete x and y
    verbose=1, # Verbosity mode. 0 = silent, 1 = progress bar, 2 =
                 one line per epoch.
    callbacks=None, # List of functions to call during training
    validation_split=0.0, # Part of dataset set aside of test
    validation_data=None, # Validation dataset, tupel (x_val,y_val)
    shuffle=True, # shuffle the training data before each epoch
    class_weight=None, # Give some classes more/less weight
    sample_weight=None, # Give some samples more/less weight
    ...
)
```

In Context

```
model.compile(optimizer='rmsprop’,
    loss='binary_crossentropy’,
    metrics=['acc’])

history = model.fit(partial_x_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(x_val, y_val))
```

The History Object

- Note that the call to model.fit() returns a History object. This object has a member history, which is a dictionary containing data about everything that happened during training.

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

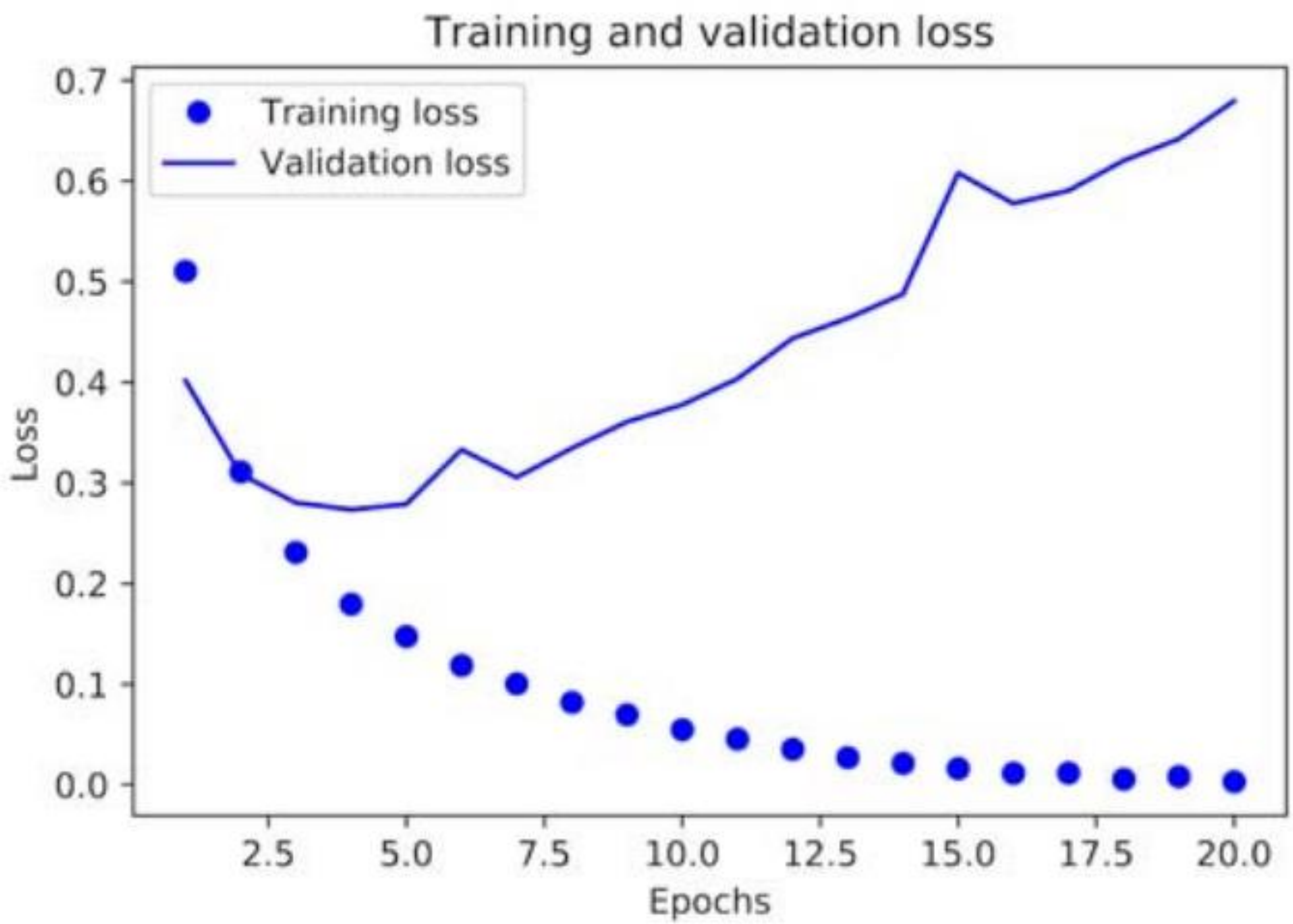
- The dictionary contains four entries: one per metric that was being monitored during training and during validation.

- You can now plot these to get Information about your performance

Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss’]

epochs = range(1, len(loss_values) + 1)

#’bo’ is for blue dot, ‘b’ is for solid blue line
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Plotting the training and validation loss



Example: Breast Cancer Revisited

- See demo

Larger Example with Cross-Validation

- Boston Housing Price data
  - You'll attempt to predict the median price of homes in a given Boston suburb in the mid-1970s
  - Given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.
  - It has relatively few data points: only 506, split between 404 training samples and 102 test samples.
  - Each feature in the input data (for example, the crime rate) has a different scale.

- We have to solve a regression problem

- See demo

Practical Recommendations

- The fewer data, you should train smaller and shallower networks in order to prevent overfitting
- Preprocessing
  - Take small values - Typically, most values should be in the 0–1 range.
  - Be homogenous- That is, all features should take values in roughly the same range.

Regularization

- Adding weight regularization

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16,
    kernel_regularizer=regularizers.l2(0.001),
    activation='relu', input_shape=(10000,)))

model.add(layers.Dense(1, activation='sigmoid'))
```

  - l2(0.001) means every coefficient in the weight matrix of the layer will add 0.001 * weight_coefficient_value to the total loss of the network.
  - Note that because this penalty is only added at training time, the loss for this network will be much higher at training than at test time.

Dropout

```
model = models.Sequential()

model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))

model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))

model.add(layers.Dense(1, activation='sigmoid'))
```

Load and Save models

- You save a Keras model into a single HDF5 file which will contain:
  - the architecture of the model, allowing to re-create the model
  - the weights of the model
  - the training configuration (loss, optimizer)
  - the state of the optimizer, allowing to resume training exactly where you left off.

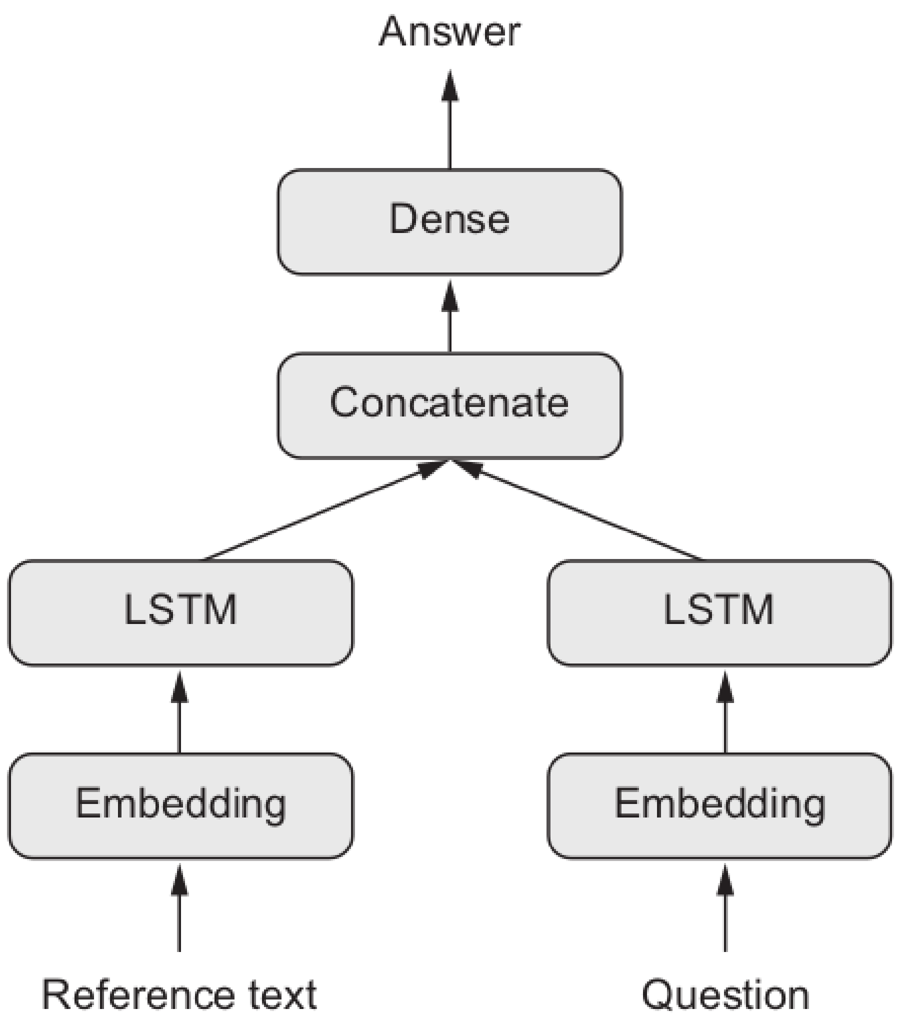
```
from keras.models import load_model

model.save('my_model.h5') # creates a HDF5 file 'my_model.h5'
del model # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

Implementing your own Callback

- Callbacks are implemented by sub-classing the class `keras.callbacks.Callback`.

- You can implement the following functions:
  - on_epoch_begin
  - on_epoch_end
  - on_batch_begin
  - on_batch_end
  - on_train_begin
  - on_train_end

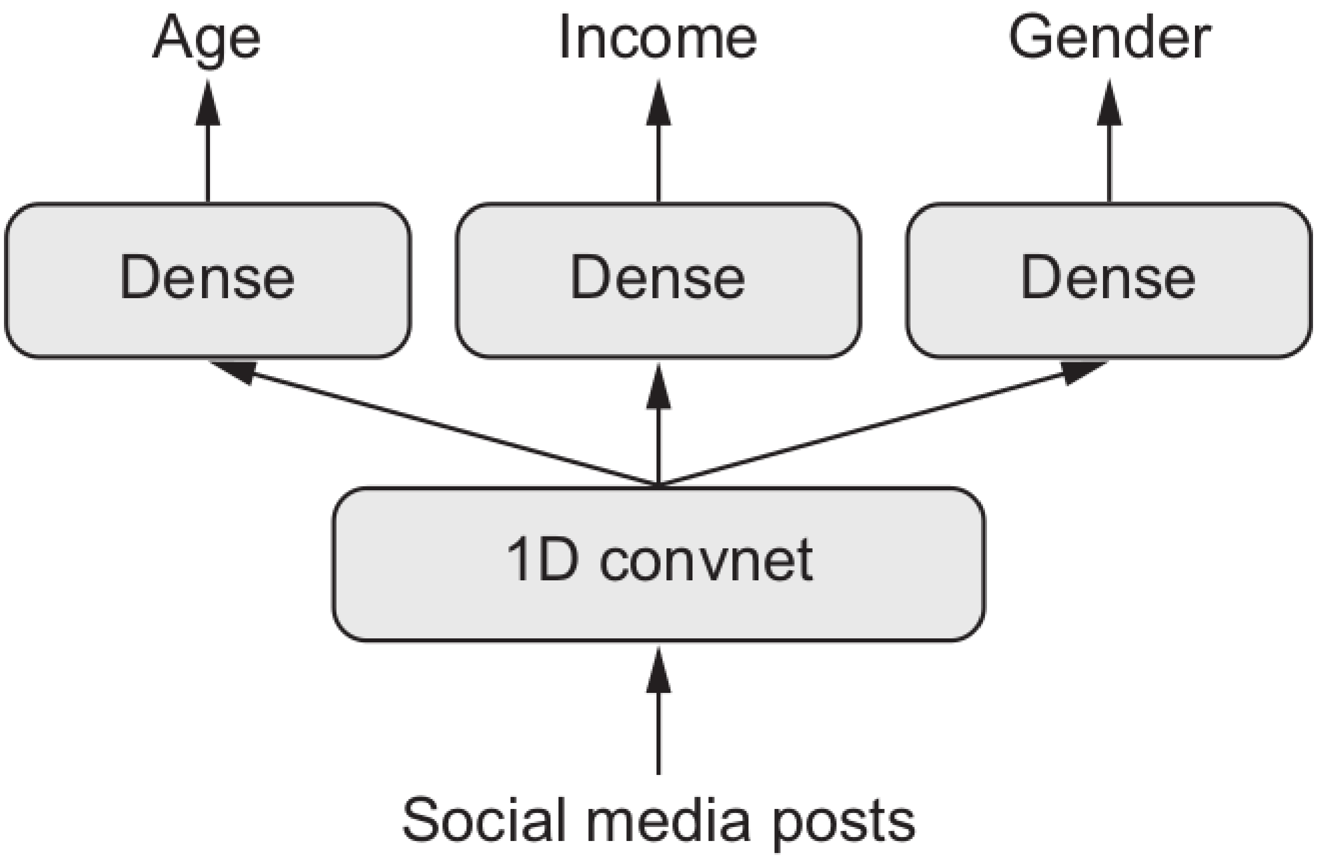
Multi Input Models



Multi Input Model using the functional API

```
text_input = Input(shape=(None,), dtype='int32', name='text')
embedded_text = layers.Embedding(64, text_vocabulary_size)(text_input)
encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,), dtype='int32', name='question')
embedded_question = layers.Embedding(32,
     question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
     axis=-1)

answer = layers.Dense(answer_vocabulary_size,
     activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
     metrics=['acc'])
```

Multi-Output Models



Multi-Output Models

```
posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
... # Construct the network how you like it
x = layers.Dense(128, activation='relu')(x)


age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups, activation='softmax’,
     name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input,[age_prediction, income_prediction,
     gender_prediction])
```

Multi-Output Models

```
model.compile(optimizer='rmsprop’,
     loss=['mse', 'categorical_crossentropy', 'binary_crossentropy’])


model.compile(optimizer='rmsprop’,
     loss={'age': 'mse’,
          'income': 'categorical_crossentropy’,
          'gender': 'binary_crossentropy'})
```

- Fitting, etc, of the model remains the same as with a normal network.

Callbacks

- When training a model, there are many things you cannot predict
- Sometime it would be helpful to intervene when something goes wrong
- Keras provides callbacks:
  - Model checkpointing: Saving the current weights of the model at different points during training.
  - Early stopping: Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
  - Dynamically adjusting the value of certain parameters during training: Such as the learning rate of the optimizer.
  - Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated: The Keras progress bar is a callback!

Early Stopping

```
import keras

callbacks_list = [
     keras.callbacks.EarlyStopping(
          monitor='acc’,
          patience=1
     ),
     keras.callbacks.ModelCheckpoint(
          filepath='my_model.h5',
          monitor='val_loss’,
          save_best_only=True
     )
]
model.compile(optimizer='rmsprop’,
     loss='binary_crossentropy’,
     metrics=['acc’])

model.fit(x, y, epochs=10, batch_size=32, callbacks=callbacks_list,
     validation_data=(x_val, y_val))
```

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. **Defining the problem and assembling a dataset**
   - What will your input data be?
   - What type of problem are you facing? Classification? Regression? …

2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. **Choosing a measure of success**
   - How do you measure if the model is successful
   - Not to be confused with the loss function which is often only a surrogate for what you actually want achieve
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. **Decide on the evaluation protocol**
   - hold-out validation
   - Cross-validation

4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. **Preparing your data**
   - Clean data
   - Normalize data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. **Define a model better than base-line**
   - Last-layer activation
   - Loss function
   - Optimization Algorithm
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. **Scaling up: Make the model overfit**
   - Add layers.
   - Make the layers bigger.
   - Train for more epochs.
7. Regularizing your model and tuning your hyperparameters
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. **Regularizing your model and tuning your hyperparameters**
   - This will take the most time
   - Add dropout.
   - Try different architectures: add or remove layers.
   - Add L1 and/or L2 regularization
8. Finalize your final model

Deep Learning Workflow

1. Defining the problem and assembling a dataset
2. Choosing a measure of success
3. Decide on the evaluation protocol
4. Preparing your data
5. Define a model better than base-line
6. Scaling up: Make the model overfit
7. Regularizing your model and tuning your hyperparameters
8. **Finalize your final model**
   - Save and distribute the model