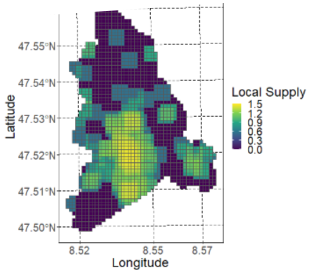
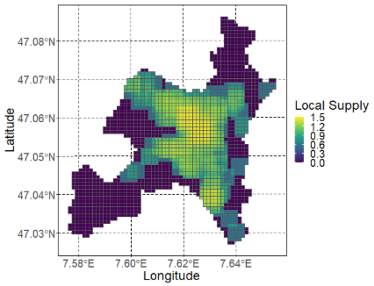
Fachhochschule Graubünden
University of Applied Sciences

Multi-Input and Multi-Output Models

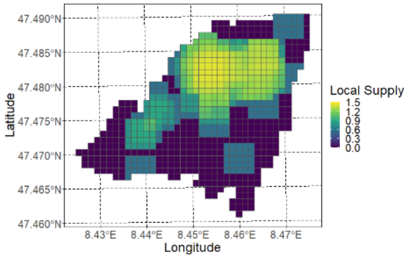
Dr. Yves Staudt



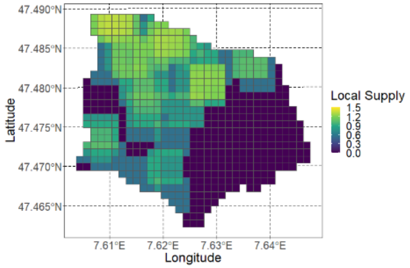
(a)
Grenchen



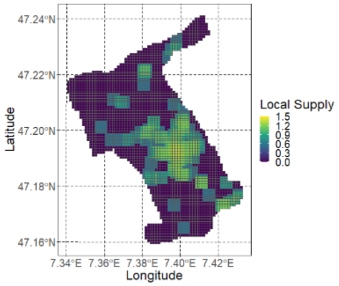
(b)
Oberglatt



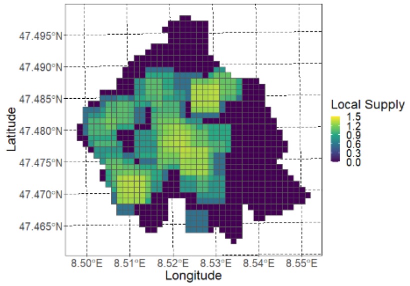
(c)
Reinach



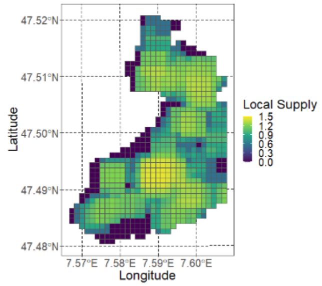
(d)
Rubigen



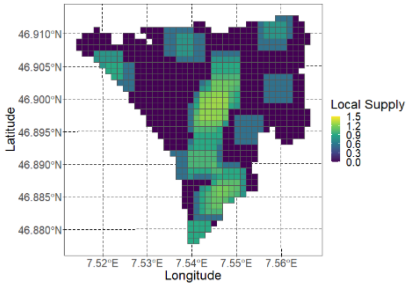
(e)
Buelach



(f)
Burgdorf



(g)
Dielsdorf



(h)
Dornach

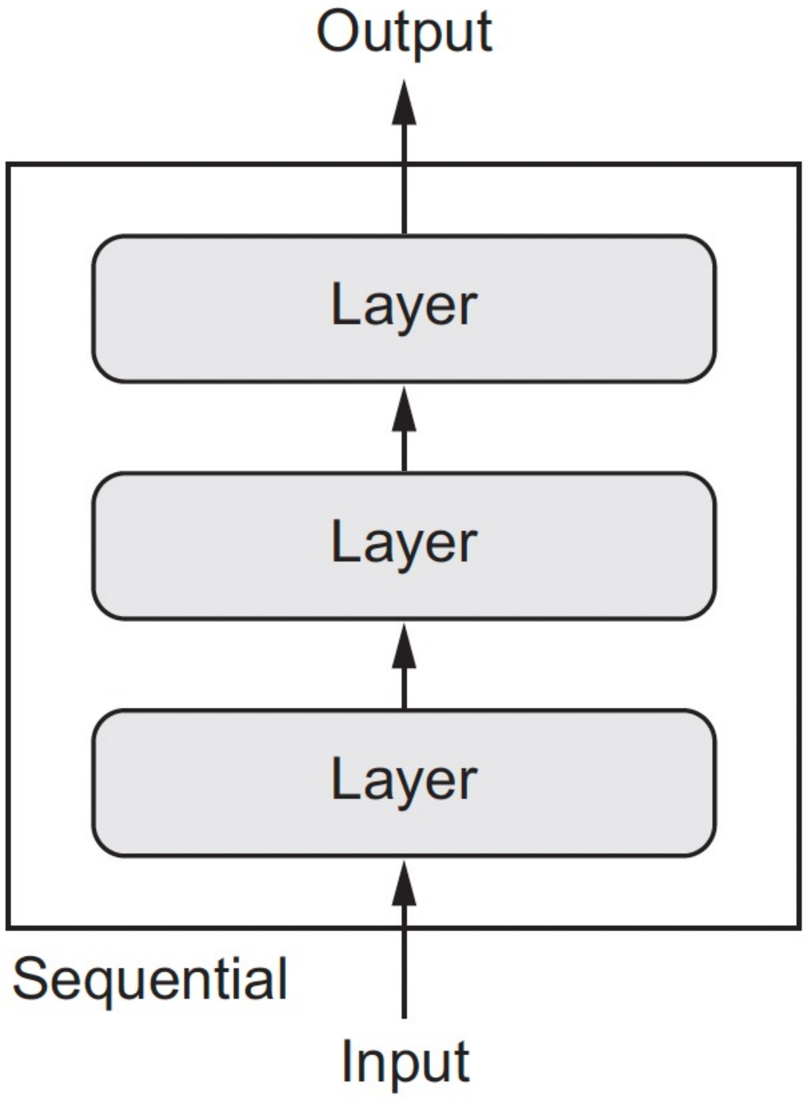
Lernziel

Die Studierende sind in der Lage

- Layers und Input Tensor als Tensoren zu verwenden.
- Layers als Funktionen anzuwenden.
- komplexere Modelle mit mehreren Inputs oder Outputs zu beschreiben und anzuwenden.

Sequential Model

- Sequential Models make the assumption that the network has exactly one input and exactly one output
- This set of assumptions is to inflexible in a number of cases, e.g. in the case several inputs or multiple outputs

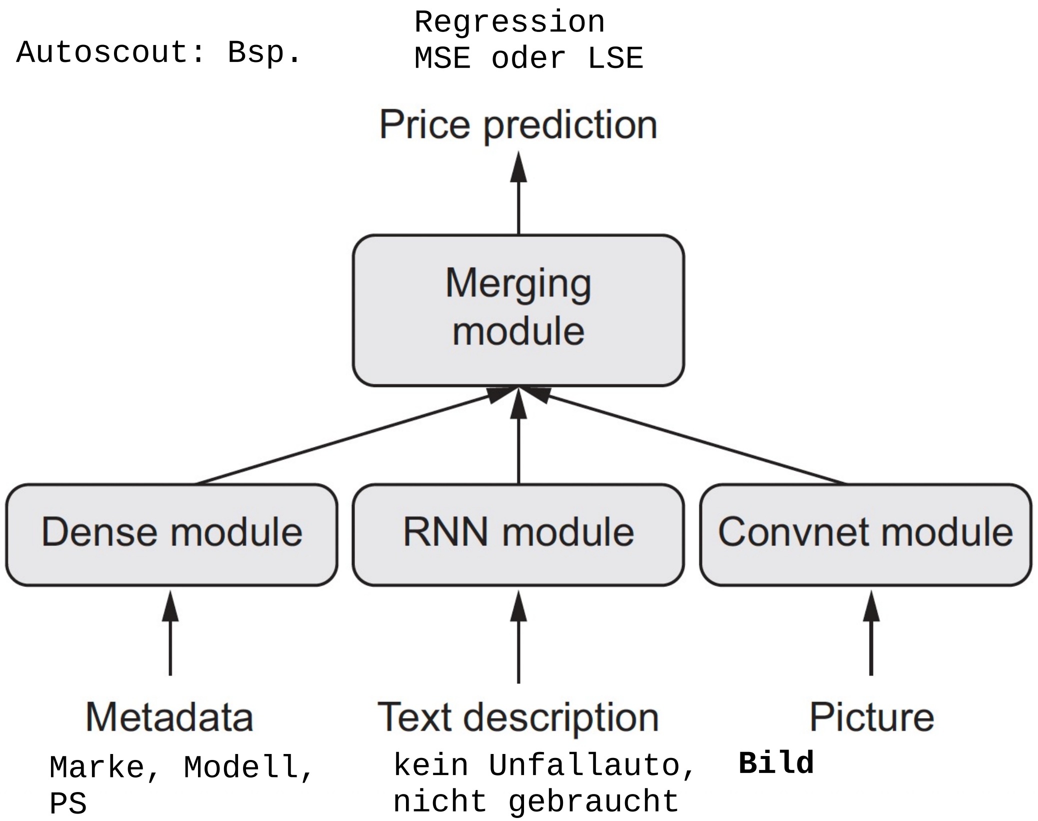


Visualization of a Sequential Model: a Linear Stack of Layers

FH GR

Multimodal Inputs

- A Multimodal Input Model merge data coming from different input sources
- A multimodal input model process each source of data using a different kind of neural network
- Example: Deep Learning Model for predicting the market price of a second hand piece of clothing using the following inputs:
  - ❖ user-provided metadata (item's brand, age and so on)
  - ❖ user-provided text description
  - ❖ picture

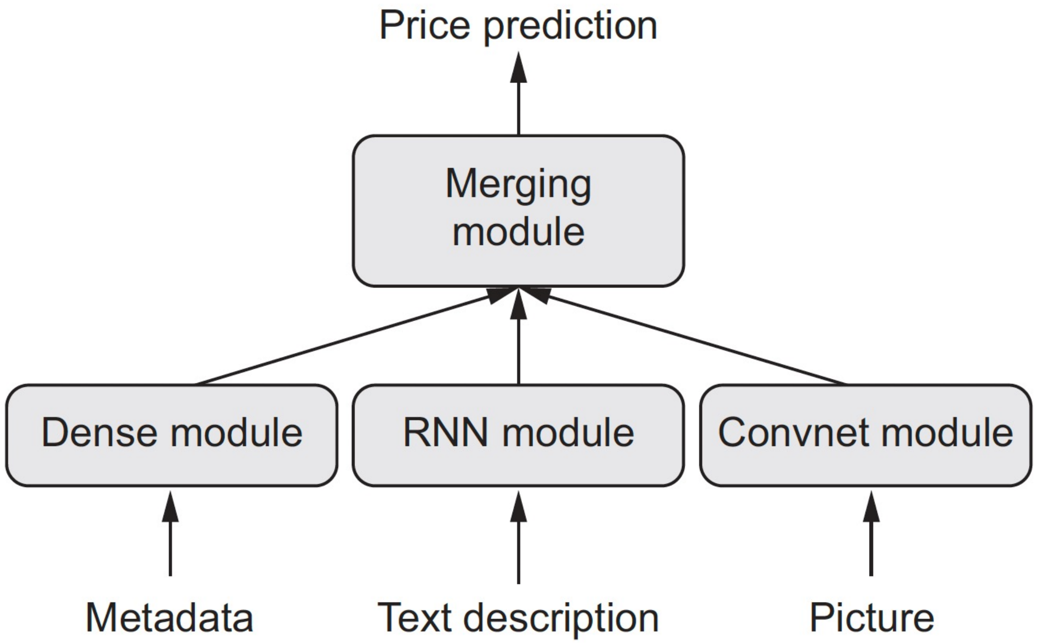


Visualization of an example of a multimodal input model for the prediction of the market price of a second-hand piece

Multimodal Inputs

- In the case of only **one data source**, the deep learning models would look like:
  - ❖ user-provided metadata – **Densely connected network** on one-hot encoded variables
  - ❖ user-provided text description – **RNN** or **1D CNN**
  - ❖ Picture – **2D CNN**

- How to combine all three at the same time?



Visualization of an example of a multimodal input model for the prediction of the market price of a second-hand piece

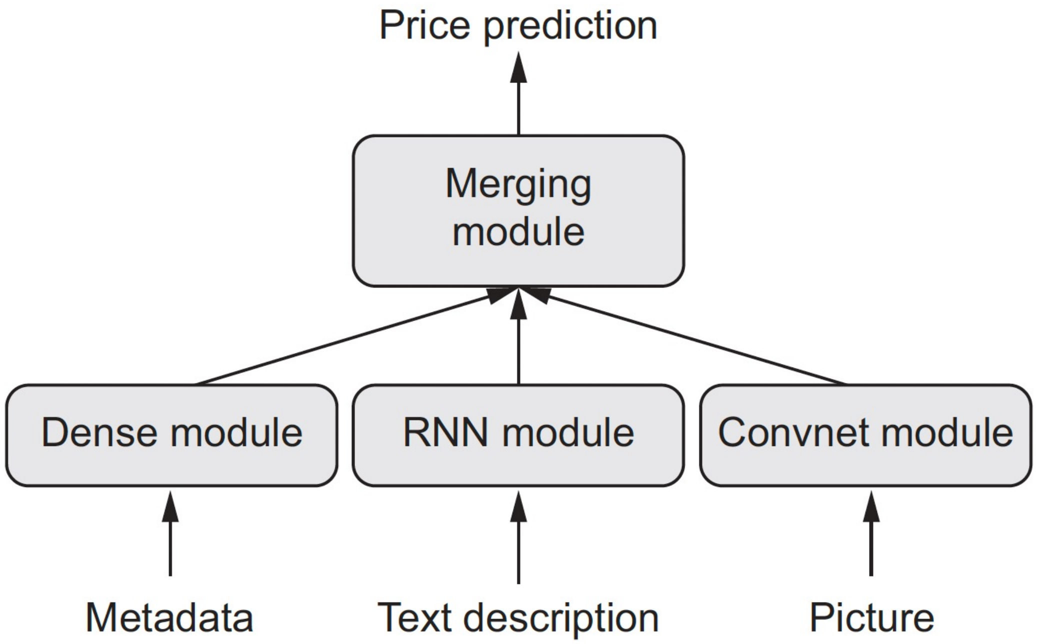
FH GR

Multimodal Inputs

- Naïve Approach:
  1. To train all three **separate** models
  2. Do a **weighted average** of their predictions

- This solution is **suboptimal**, because the information extracted by the models may be **redundant**.

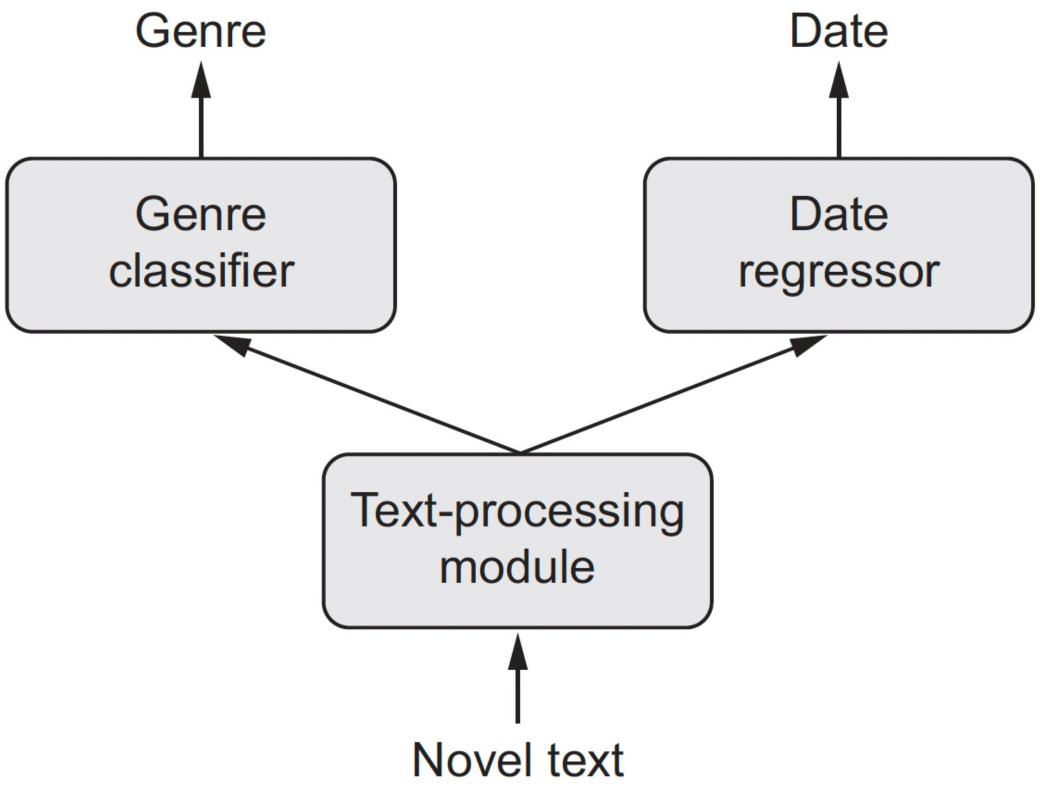
- Better way: Jointly learn a more accurate model of the data by using a model that can see **all available input** modalities **simultaneously**



Visualization of an example of a multimodal input model for the prediction of the market price of a second-hand piece

Multiple Targets

- Multiple target models consist of a model which has a task to predict **multiple target attributes** of input data
- **Example**: Given the text of a novel or short story, goal to
  - Classify the text by genre
  - Predict the data the text was written
- One could train **two separate** models:
  - One for the genre, and
  - One for the date
- Attributes are **not statistically independent – correlations** between genre and date
- Could build better models by **learning jointly** the prediction of genre and date at the same time
- Joint models have multiple targets

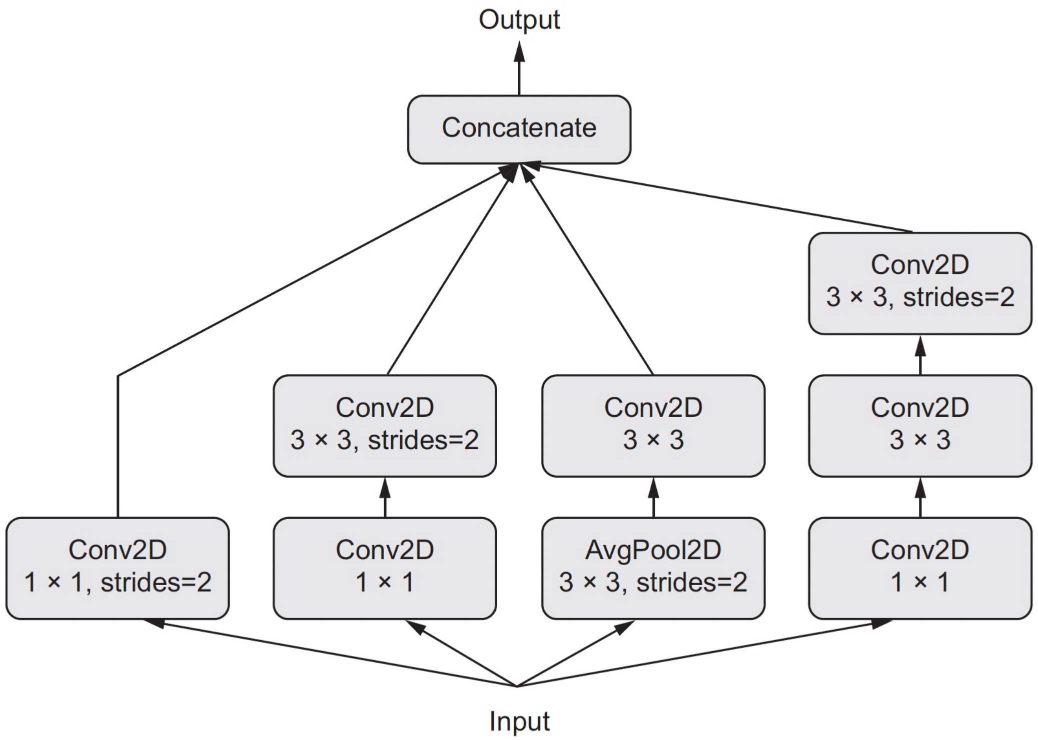


Visualization of an example of a multiple target model for classifying a text by genre and simultaneously predict the date the text was written

FH GR

Nonlinear Network Topology – Acyclic Graphs

- Example of nonlinear network topology: **inception family** (developed by Szegedy et al. at Google)
- The Inception family relies on **Inception modules**
- In Inception modules, the input is processed by several **parallel convolutional branches** whose outputs are then merged back into a single tensor



Visualization of an inception module: a subgraph of layers with several parallel convolutional branches

FH GR

Nonlinear Network Topology – Residual Connection

- **Residual connection** consist of **reinjecting previous** representations into the **downstream** flow of data by adding a past output tensor to a later output tensor

- Residual connection helps to **prevent information loss** along the **data-processing flow**

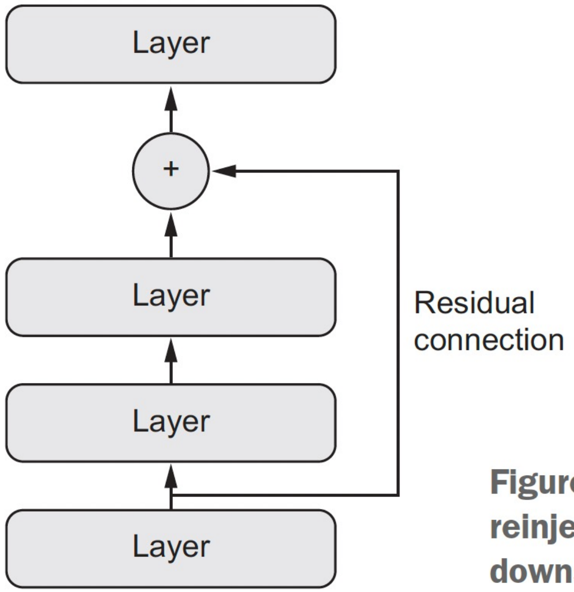


Figure 7.5 A residual connection: reinjection of prior information downstream via feature-map addition

Visualization of a residual connection example

Sequential vs Functional API

```
from keras.models import Sequential, Model
from keras import layers
from keras import Input

seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Sequential model, which you already know about

Representation of the Sequential API in Keras

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = Model(input_tensor, output_tensor)

model.summary()
```

Its functional equivalent

The Model class turns an input tensor and output tensor into a model.

Let's look at it!

Representation of the Functional API in Keras

FH GR

Functional API

- In the functional API:
  - ❖ we **directly manipulate tensors** and
  - ❖ we use **layers as functions** that take tensors and return tensors

- New part is instantiating a **Model object** using only an input and an output tensor
- Behind the scenes, the Model object **retrieves every layer** involved in going **from** input tensor **to** output tensor
- Model object brings **all layer together** into a graph like structure
- The output tensor is obtained by repeatedly transforming input tensors
- **RuntimeError** stands for the case, we want to train a model, that cannot relate inputs and outputs

Multi-input models

- Functional API can be used to build models that have **multiple inputs**
- In multiple-input models the **different branches** must be **merged** at some point
- A **layer** is used to combine several tensors
- In keras the merge operations are done with
  - ❖ Keras.layers.add
  - ❖ Keras.layers.concatentate
  - ❖ ….

FH
GR

Example Multi-Input Model

Typical Example: Question-Answering Model:

- **Two inputs**: a natural-language question and a text snippet
- Output: Answer

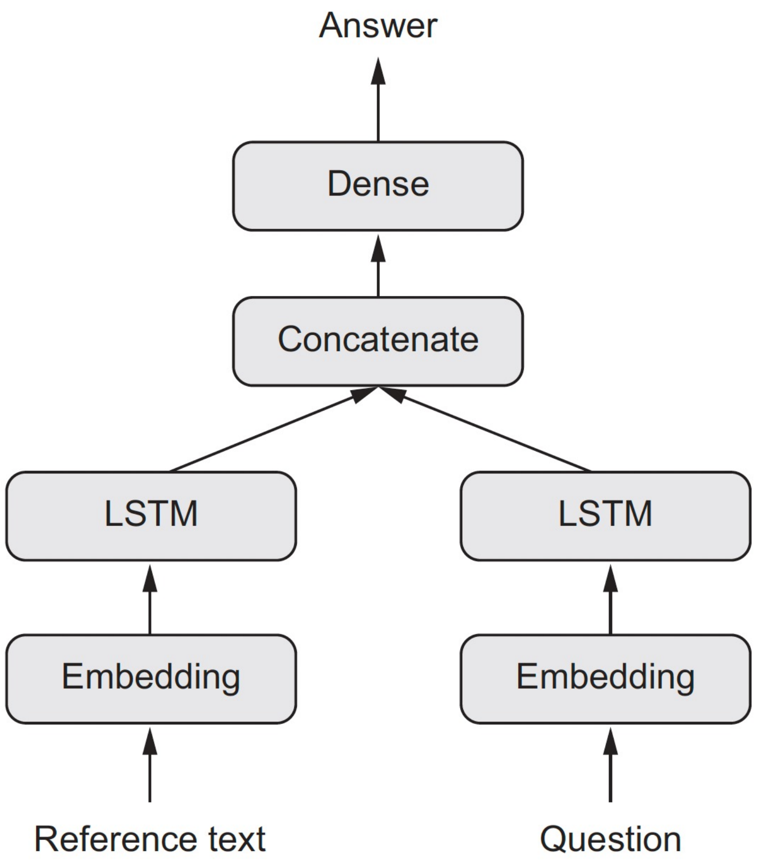
The two inputs **provide information** to be used for answering the question

Model must produce an answer

Simplest answer can be obtained via softmax over som predfined vocabulary

Deep Learning Setup:

1. Set up two independent branches encoding the two inputs
2. Concatenate these vectors
3. Softmax classifier



Visualization of an answer and question model

FH
GR

Multi-Output Models

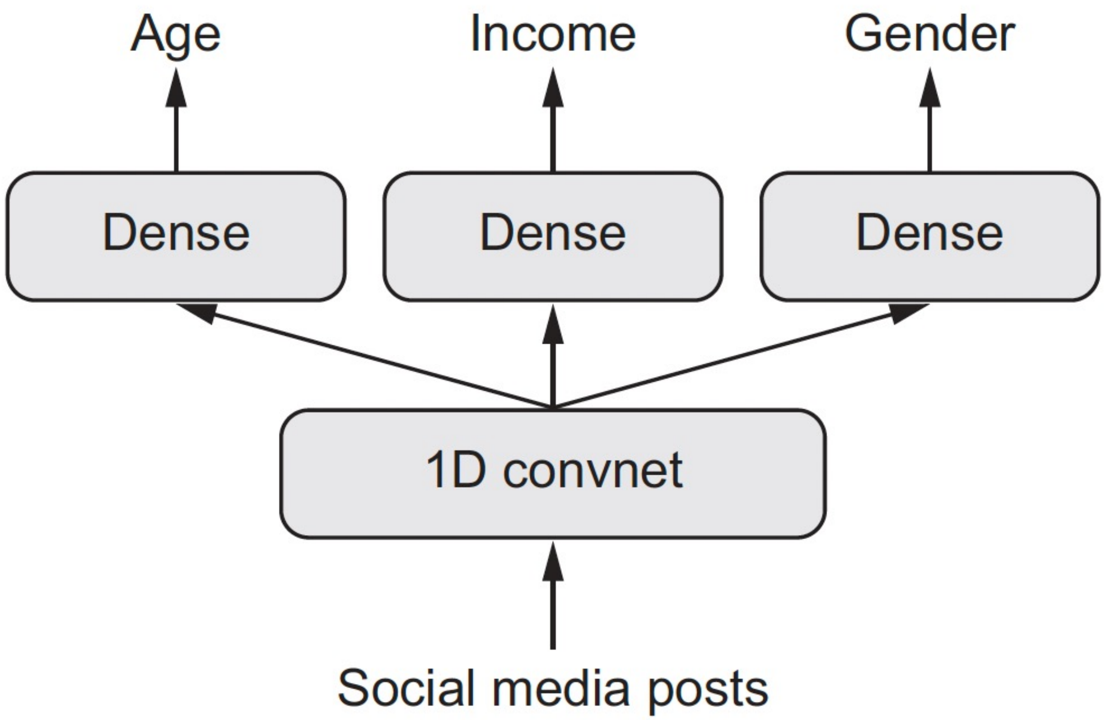
Modell kann nur eine Verlustfunktion minimieren

unterschiedliche Grössenordnungnen (Alter 10er vs
Income 1'000er)
Skalierbarkeit wird schwieriger

- We can use the **functional API** the same way to build modles with multiple outputs

- Network that attempts to **simultaneously** predict different properties of the data

- Example:
  - ❖ Input: Social Media Post single anonymous person
  - ❖ Ouput: Age, Gender and Income Level



Visualization of multi head model, where age, income and gender are predicted from social media posts

FH
GR

Multi-Output Models

Important:

- Training such a model requires the ability to **specify different loss functions** for different heads (outputs) of the network

gemeinsame Verlustfunktion erstellen

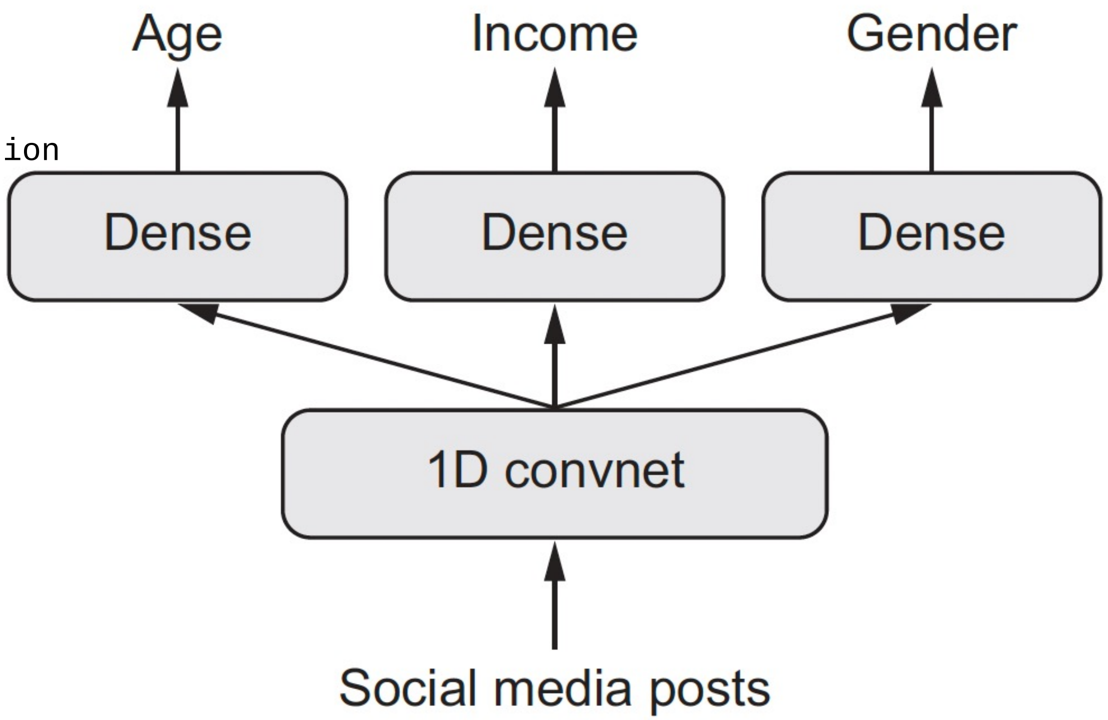
Example:

- Age prediction – Scalar Regression
- Income prediction – Scalar Regression
- Gender prediction – Binary Classification

Gradient Descent requires you to minimize a **scalar**

Gradient Descent requires we **combine** the **losses** to a single value in order to train the model

Simplest Combination: **Sum** of the different losses



Visualization of multi head model, where age, income and gender are predicted from social media posts

FH GR

Loss Function in Multi-Output Model

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

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

Equivalent (possible only if you give names to the output layers)

- **Gradient Descent** requires you to minimize a **scalar**
- Gradient Descent requires we **combine** the **losses** to a single value in order to train the model
- Simplest Combination: **Sum** of the different losses

- In **Keras**: the losses will specify in a list or a dictionary of losses
- Loss values are summed into a global loss

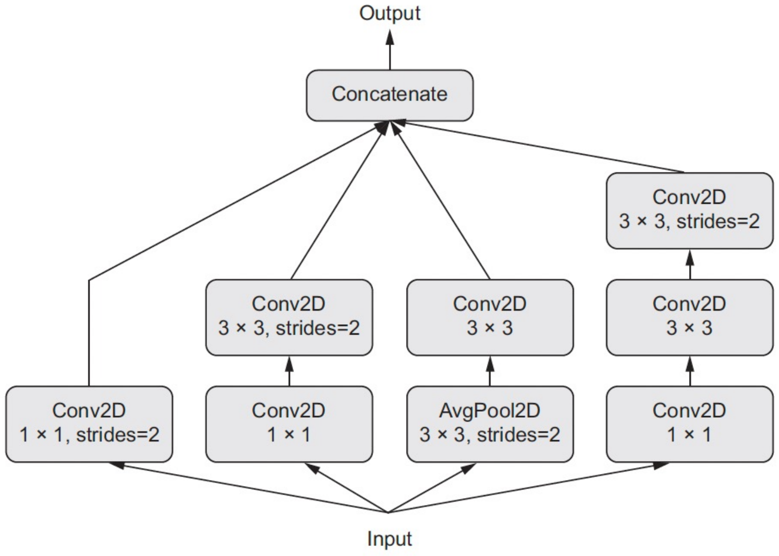
- Very **imbalanced loss** contributions will cause the model representation to be optimized preferentially for the task with the **largest** individual loss
- Optimization of the loss will be done at the **expense** of the other tasks
- **Solution**: Assign **different levels** of **importance** to the loss values in their contribution to the final loss
- Useful solution for the case where the losses' values use different scales

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
              loss_weights=[0.25, 1., 10.])
```

FH GR

Directed Acyclic Graphs of Layers

- With the functional API more complex models as inception modules or residual connections can be built
- For further information we refer to the book of Chollet (2018)



Visualization of an inception module

Every branch has the same stride value (2), which is necessary to keep all branch outputs the same size so you can concatenate them.

In this branch, the striding occurs in the spatial convolution layer.

```
from keras import layers

branch_a = layers.Conv2D(128, 1,
                         activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate(
    [branch_a, branch_b, branch_c, branch_d], axis=-1)
```

Concatenates the branch outputs to obtain the module output

In this branch, the striding occurs in the average pooling layer.

Visualization of the code an inception module

Layer weight sharing

- Important feature of the functional API: ability to **reuse a layer instance** several times

- When we call a layer instance **twice**, instead of instantiating a new layer for each call, you reuse the **same weight** with every call

`Reihenfolge der Layers A / B ist irrelevant`

- Reuse of a layer instance allow to build models with **shared branches** – several branches that **share** the **same knowledge** and perform the same operations

- Reuse of a layer instance share the **same representations** and learn these **representations simultaneously** for different sets of inputs

Example Layer Weight Sharing

- Consider a model that attempts to assess the **semantic similarity** between two sentences
- Model has **two inputs** - the two sentences to compare
- **Output**: Score between 0 and 1, where 0 means unrelated and 1 means sentences that are either identical or reformulations of each other
- The two input sentences are interchangeable – semantic similarity is a **symmetrical relationship**
- Similarity A to B is identical to the similarity of B to A
- The symmetrical relationship explains why it **would not** make sense to learn **two independent models**
- More Sense to process both Inputs with a single LSTM layer
- The representations of this LSTM (its weights) are learned based on both inputs **simultaneously**
- In this case we speak from a **Siamese LSTM** model or **shared** LSTM

```
from keras import layers
from keras import Input
from keras.models import Model

lstm = layers.LSTM(32)

left_input = Input(shape=(None, 128))
left_output = lstm(left_input)

right_input = Input(shape=(None, 128))
right_output = lstm(right_input)

merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)

model = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)
```

Instantiates a single LSTM layer, once

Building the left branch of the model: inputs are variable-length sequences of vectors of size 128.

Building the right branch of the model: when you call an existing layer instance, you reuse its weights.

Builds the classifier on top

Instantiating and training the model: when you train such a model, the weights of the LSTM layer are updated based on both inputs.

Visualization of a code example of layer weight sharing

FH GR

Models as Layers

- In the functional API, **models** can be used as you would use **layers**
- Models can be thought as a "**bigger layer**"
- When you call a model instance, you are **reusing** the **weights** of the model – exactly like what happens when you call a layer instance
- Calling an instance, whether it is a layer instance or a model instance, will always **reuse the existing learned representations** of the instance

```
from keras import layers
from keras import applications
from keras import Input

xception_base = applications.Xception(weights=None,
                                      include_top=False)

left_input = Input(shape=(250, 250, 3))
right_input = Input(shape=(250, 250, 3))

left_features = xception_base(left_input)
right_input = xception_base(right_input)

merged_features = layers.concatenate(
    [left_features, right_input], axis=-1)
```

The base image-processing model is the Xception network (convolutional base only).

The inputs are 250 × 250 RGB images.

Calls the same vision model twice

The merged features contain information from the right visual feed and the left visual feed.

- **Practical example** of model reuse: **Vision model** that uses a **dual camera** as its input: two parallel cameras, a few centimeters (one inch) apart
- Such an example allows to perceive the depth
- We **do not** need two **independent** models to extract visual features from the left camera and the right camera before merging the two feeds
- Such low-level processing can be **shared** across the **two inputs** – the sharing is done via **layers** that use the **same weights** and thus share the same representations

Visualization of a code example of model sharing

Multiple Inputs and Multiple Outputs

In the case of multiple inputs and mulitple output tensors, the model should be calles with a list of tensors

For example: $y_1, y_2 = model([x_1, x_2])$

FH
GR

Conclusion

- With the functional API you can produce the same results as with Sequential API, however functional API allows you to produce more **complex** models

- With the functional API you can build **models** with several inputs, outputs and complex internal network topology

- With the functional API you can **reuse** the weights of a layer or model across different processing branches

FH
GR

Referenzen

Chollet, François – Deep Learning with Python (2017)

FH GR

Fragen

Darstellung eines Fragesymbol aufgerufen von der Webseite https://www.qnigge.de/news/detail/modul-v/#images am 12.07.2021.

Fachhochschule Graubünden
Pulvermühlestrasse 57
7000 Chur
T +41 81 286 24 24
info@fhgr.ch

Vielen Dank für Ihre Aufmerksamkeit.
Grazia fitg per l'attenziun.
Grazie per l'attenzione.

Fachhochschule Graubünden
Scola auta spezialisada dal Grischun
Scuola universitaria professionale dei Grigioni
University of Applied Sciences of the Grisons