Texte analysieren mit NLP

Nutzung von BERT und Transfer Learning

Dr. Christian Winkler, datanizing GmbH

Text-Analyse Workshop, 01.12.2021

Zwei Tickermeldungen: Naive Vektorisierung vs. Sprachmodell

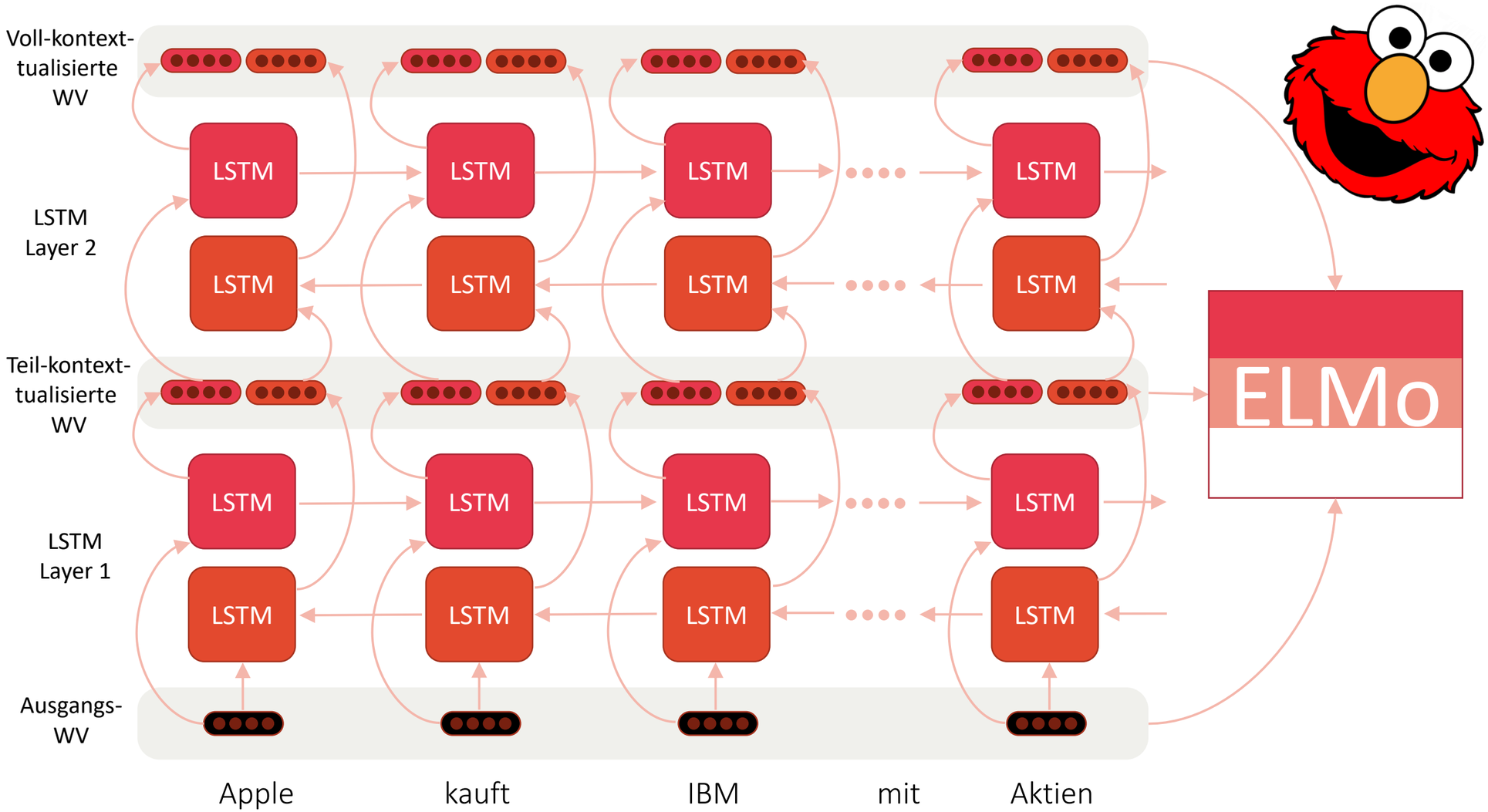
Tickermeldung	Linguistische Analyse (Lemmas)	Bag-of-Words	Kommentare
Analyse: Apple- oder IBM-Aktien kaufen?	Analyse Apple oder IBM Aktien kaufen		2
Analyse: Apple kauft IBM mit Aktien	Analyse Apple kaufen IBM mit Aktien		18.232

Sprache ist ein extrem stark kontextualisiertes Kommunikationsmedium (anders als Bilder)

Training führt häufig nur zum Lernen immer wieder verwendeter Konstrukte

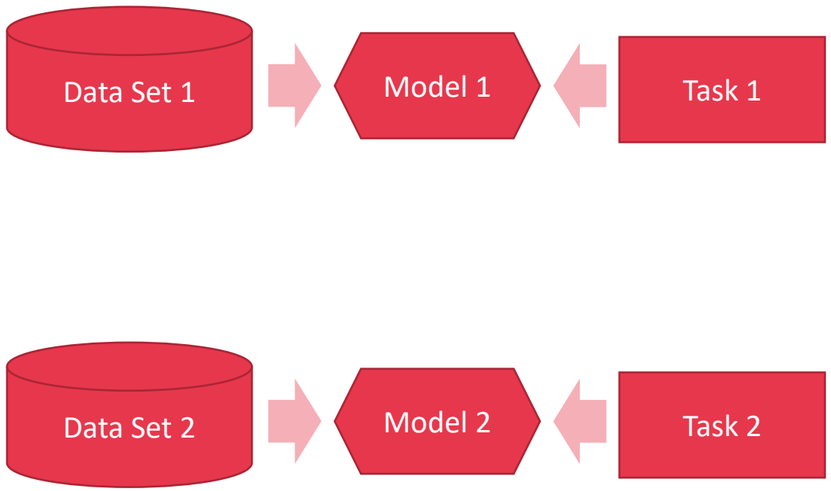
➔ Sprachmodell

Konstruktion eines Sprachmodells mit ELMO



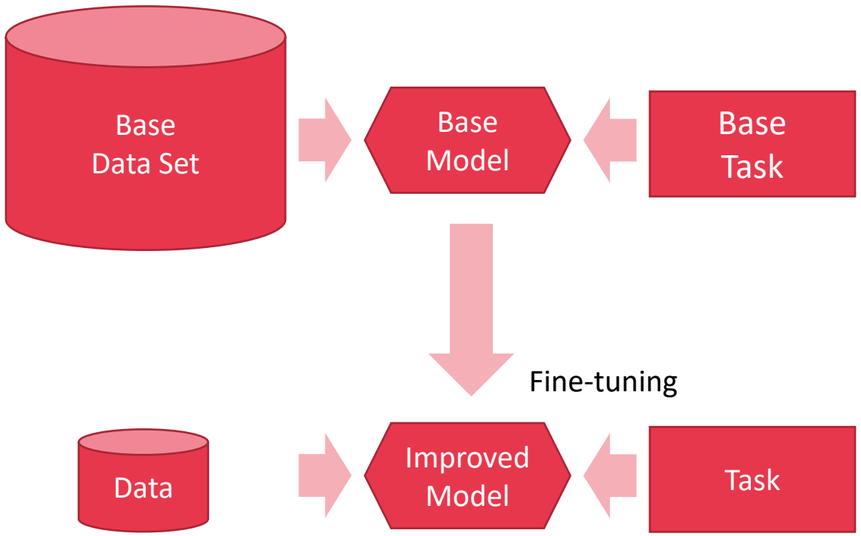
Funktionsweise von Transfer Learning

Klassisches ML



Ein Modell wird für genau eine Aufgabe
bei Null beginnend überwacht trainiert.
Es werden immer viele Trainingsdaten benötigt.

Transfer Learning



Ein Basis-Modell, dass auf einem großen Datensatz
(unüberwacht!) trainiert wurde, wird mit wenig Daten auf eine
spezifische Aufgabe angepasst.

Grundfunktionsweise BERT: Transformer-Modell mit Attention

BERT basiert auf einem sog. Transformer-Modell

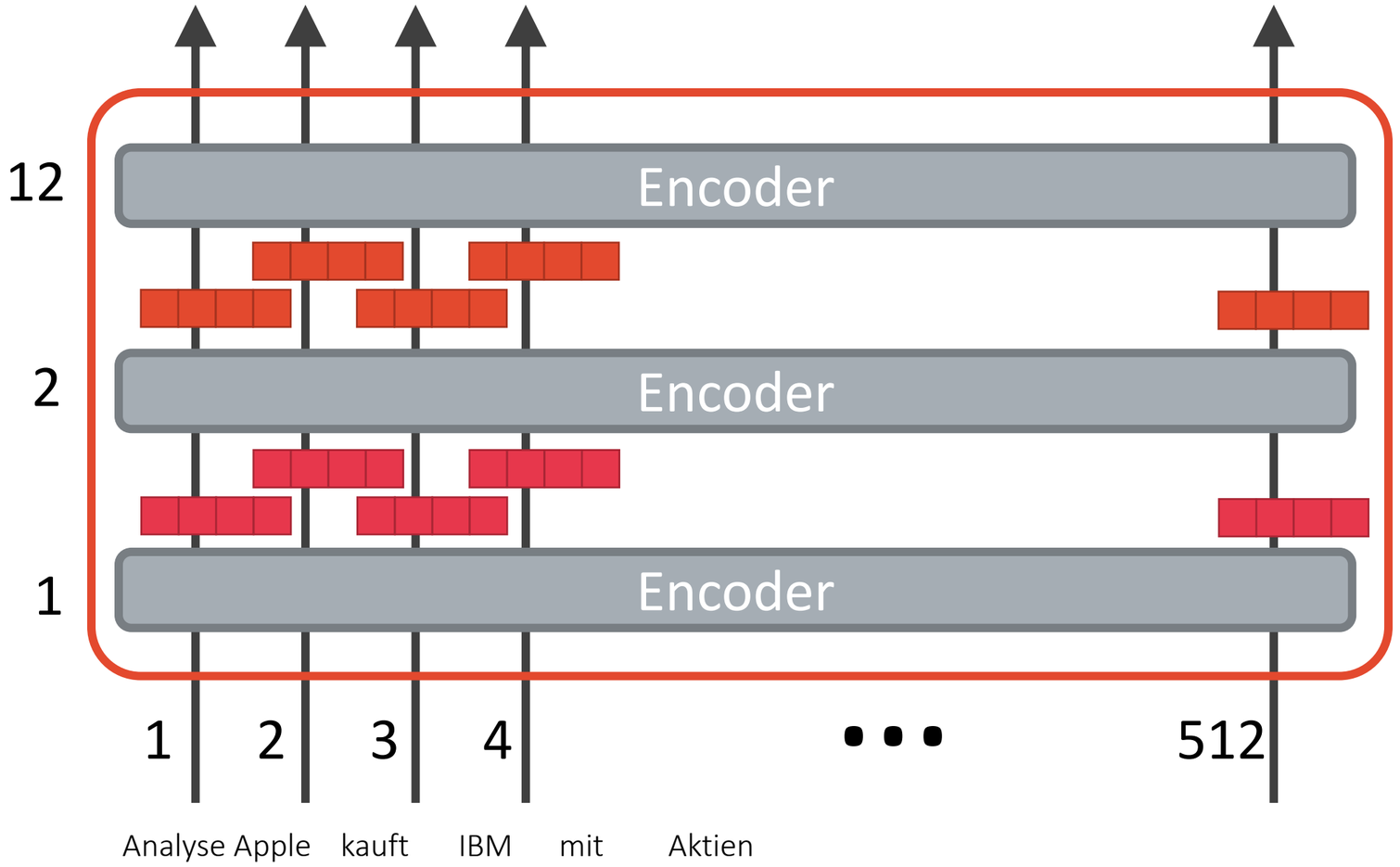
- Vorher verwendet für Übersetzungen
- Sog seq2seq-Verfahren

BERT nutzt nur den Encoder

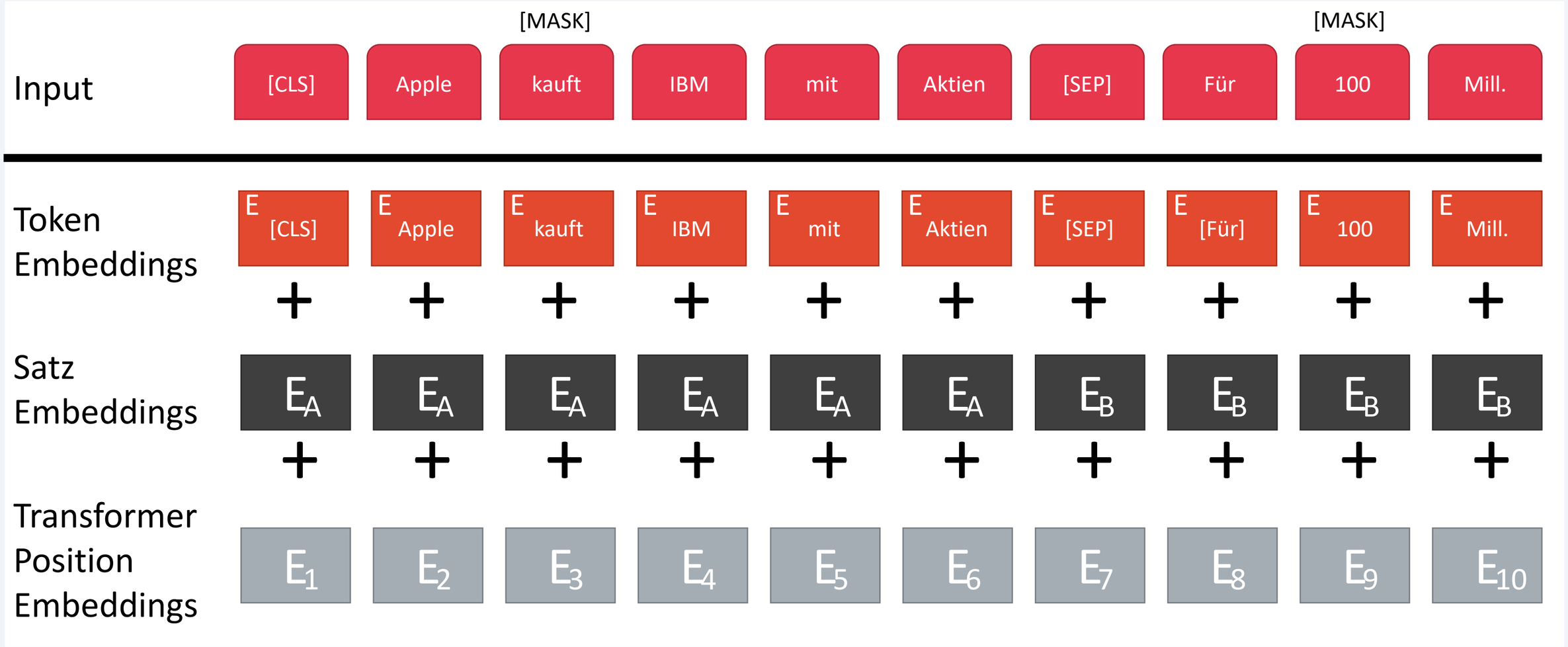
- Encoder wird kaskadiert
- Unterschiedliche Layer modellieren die Kontextualisierung

BERT ist ein sehr komplexes Modell

- Base: 12 Layer, 12 Attention Heads, 110 Millionen Parameter
- Large: 24 Layer, 16 Attention Heads, 340 Millionen Parameter
- Sehr, sehr aufwändiges Training
  ➔ Transfer Learning



Training auf Vorhersage des nächsten Satzes



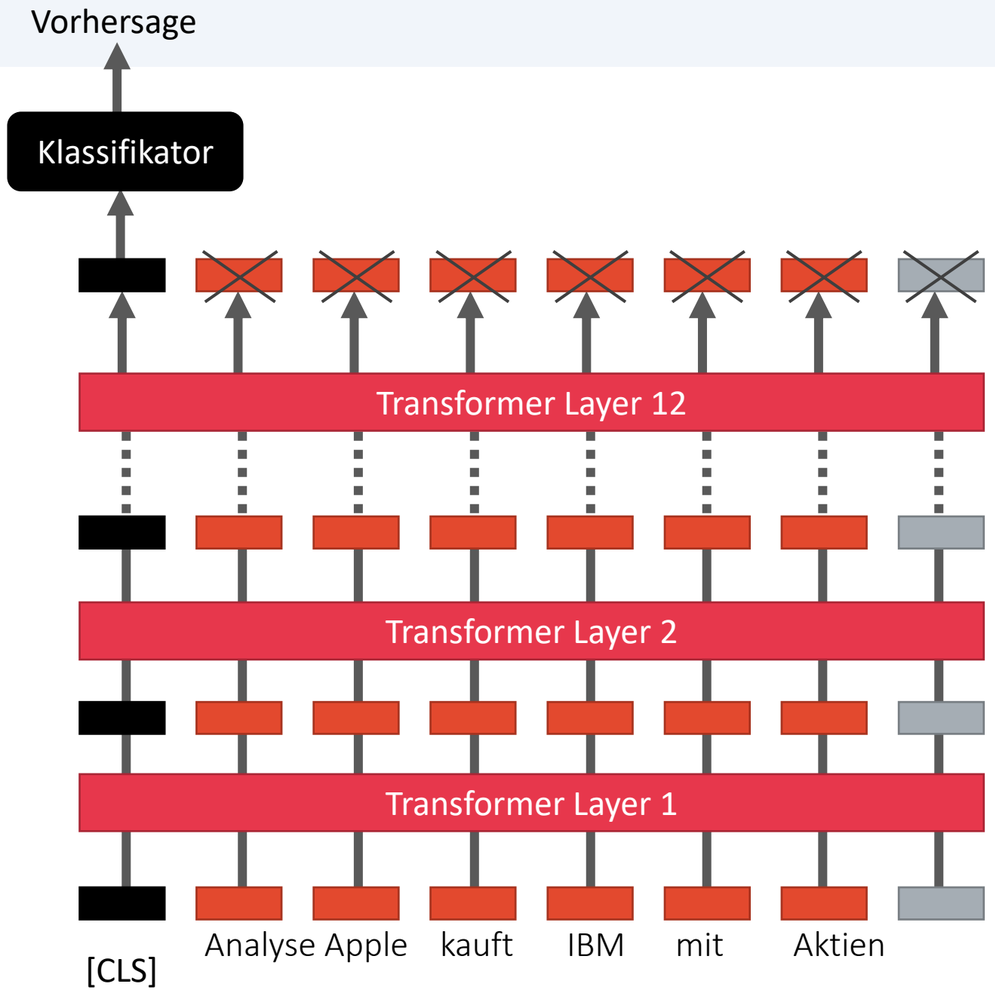
Finetuning für Klassifikationsaufgaben

Grober Ablauf des Trainingsprozesses:

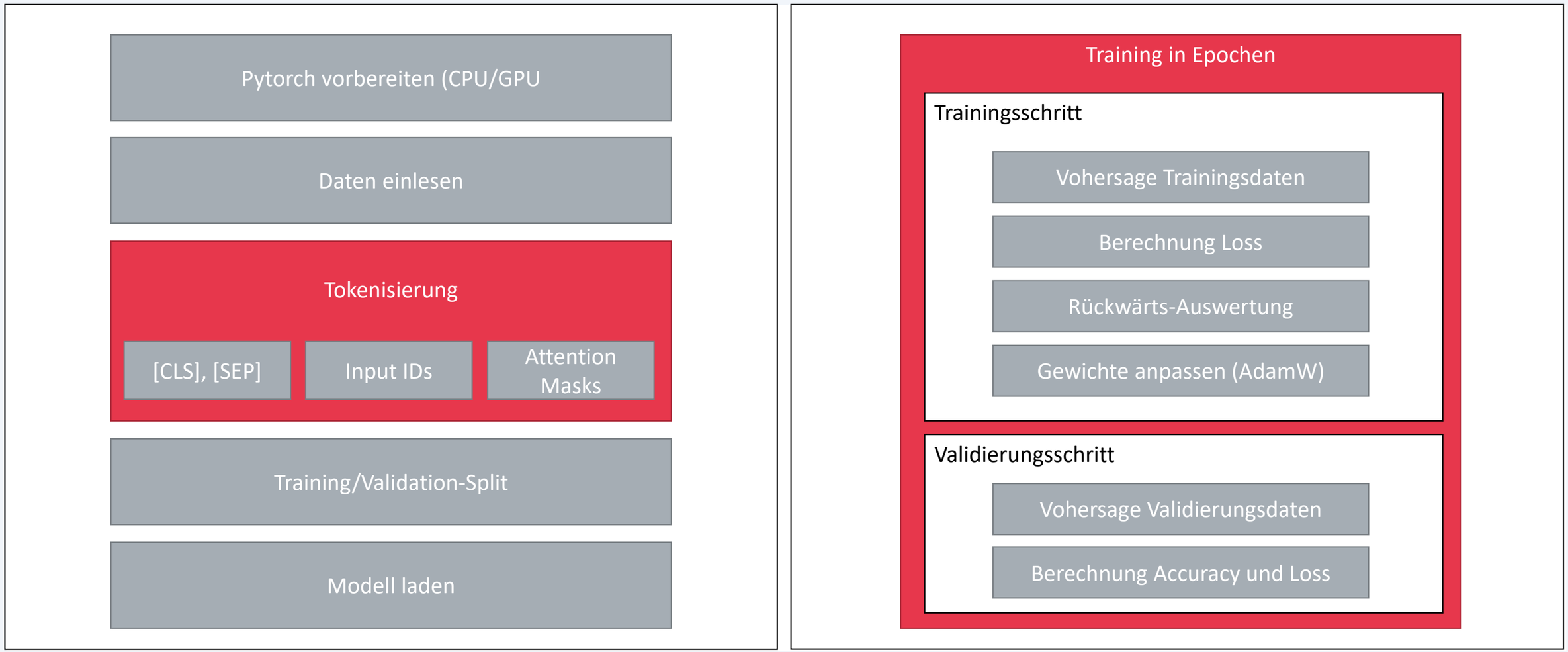
- Embedding der gesamten Sätze berechnen
- Berechnete Embeddings sind hoch kontextualisiert
- Sehr viel (alle?) Information steckt daher im kontextualisierten Embedding des ersten Tokens (der gar nicht zum Satz gehört)
- Embedding-Vektor von [CLS] wird durch Classifier verarbeitet
- Iteration mit Anpassung der Gewichte im letzten Layer

Klassifikationsprozess:

- Kontextualisierung des gesamten Satzes
- Klassifikation nur mit (kontextualisiertem) [CLS]



Ablauf: Finetuning von BERT



Beispiel: Erfolgsvorhersage von Newsticker-Meldungen

Schritt 1: Tokenisierung

```python
from transformers import BertTokenizer

# Wir nutzen den DBMDZ-Tokenizer der Bayerischen Staatsbibliothek
tokenizer = BertTokenizer.from_pretrained('dbmdz/bert-base-german-uncased', do_lower_case=True)
```

```python
# Jetzt alle Sätze tokenisieren und IDs merken
input_ids = []
attention_masks = []

for t in text:
    encoded_dict = tokenizer.encode_plus(
                        t,
                        add_special_tokens = True,    # '[CLS]' und '[SEP]'
                        max_length = 64,
                        pad_to_max_length = True,
                        return_attention_mask = True,  # Attention-Masks erzeugen
                        return_tensors = 'pt',         # pytorch-Tensoren als Ergebnis
                   )
    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])
```

```python
# Headline, Tokenisierung und IDs anzeigen
print(text[0])
print(tokenizer.tokenize(text[0]))
print(input_ids[0])
```

```
Apple TV+: Start in der Türkei unklar
['apple', 'tv', '+', ':', 'start', 'in', 'der', 'turk', '##ei', 'unklar']
tensor([  102, 12045,  3845,  1376,   847,  2629,   142,   127, 29538,   109,
         9718,   103,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0])
```

Beispiel: Erfolgsvorhersage von Newsticker-Meldungen

Schritt 2: Training

```
# Modell in Trainingsmodus stellen
model.train()

# For each batch of training data...
for step, batch in enumerate(tqdm(train_dataloader, desc="Training")):
    # Daten entpacken und in device-Format wandeln
    b_input_ids = batch[0].to(device)
    b_input_mask = batch[1].to(device)
    b_labels = batch[2].to(device)

    # Gradienten löschen
    model.zero_grad()

    # Vorwärts-Auswertung (Trainingsdaten vorhersagen)
    loss, logits = model(b_input_ids,
                         token_type_ids=None,
                         attention_mask=b_input_mask,
                         labels=b_labels)

    # Loss berechnen und akkumulieren
    total_train_loss += loss.item()

    # Rückwärts-Auswertung, um Gradienten zu bestimmen
    loss.backward()

    # Gradient beschränken wegen Exploding Gradient
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    # Parameter und Lernrate aktualisieren
    optimizer.step()
    scheduler.step()

# Calculate the average loss over all of the batches.
avg_train_loss = total_train_loss / len(train_dataloader)
```

```
# Evaluate data for one epoch
for batch in tqdm(validation_dataloader, desc="Validierung"):
    # jetzt die Validierungs-Daten entpacken
    b_input_ids = batch[0].to(device)
    b_input_mask = batch[1].to(device)
    b_labels = batch[2].to(device)

    # Rückwärts-Auswertung wird nicht benötigt, daher auch kein Gradient
    with torch.no_grad():
        # Vorhersage durchführen
        (loss, logits) = model(b_input_ids,
                               token_type_ids=None,
                               attention_mask=b_input_mask,
                               labels=b_labels)

    # Loss akkumulieren
    total_eval_loss += loss.item()

    # Vorhersagedaten in CPU-Format wandeln, um Accuracy berechnen zu können
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()
    total_eval_accuracy += flat_accuracy(logits, label_ids)


# Report the final accuracy for this validation run.
avg_val_accuracy = total_eval_accuracy / len(validation_dataloader)
tqdm.write("Accuracy: %f" % avg_val_accuracy)
```