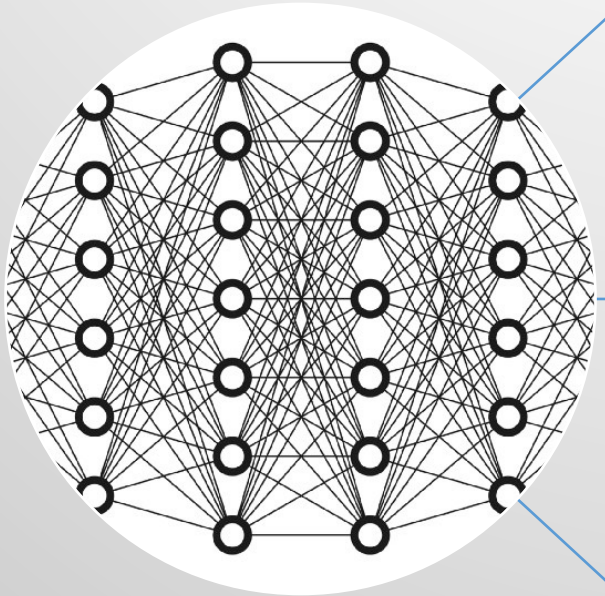
Natural Language Processing

Dr. Karen Mazidi

Part Six:
Deep Learning



Topics

- embeddings

Quizzes

- Q: no quiz on this material

Homework

- Homework : Text classification

Embeddings

```
cat = [.34, .21, .38, .54, .05, .16, .27, .82]
```

- An embedding is a vector of floating-point values of a predetermined length (usually powers of 2 for cache compatibility)
- Embedding representations are dense compared to sparse one-hot matrix representations
- Embedding lengths vary from 8 to 1024 or larger
- The higher the dimension, the more data is needed to learn the embeddings
- The goal of embedding is that similar words end up with similar vectors

Timeline of embedding evolution part 1

- 1950s BOW – word frequencies are used as features
- 1970s TF-IDF – modifies word frequencies to downrank common words
- 2013 Word2Vec – word embeddings learned in a neural network by using word co-occurence on a large corpus
- 2014 encoder-decoder RNNs create a document embedding using an RNN encoder, and another RNN for the decoder
- 2017 Transformer – encoder-decoder that uses an attention mechanism to compute better embeddings
- 2018 BERT – bidirectional Transformer trained using masked language modeling and next sentence prediction; uses global attention

Timeline part 2

- 2018 GPT – first autoregressive (predicts future values from past values) Transformer model
  - 2019 GPT-2 bigger version trained on more data
  - 2020 GPT-3 even bigger
- 2019 ALBERT – lighter version of BERT; lower memory consumption and faster training
- 2019 RoBERTa – improved BERT
- 2019 DistilBERT – smaller and faster BERT with 95% performance
- . . . something new every day

Embeddings

- Let’s say we trained an embedding layer of size 8
- Each word is represented by an 8-element vector:

```
cat = [.34, .21, .38, .54, .05, .16, .27, .82]
```

- This vector represents the location of the word in 8-dimensional space
- Higher dimensions are generally more meaningful representations

Embeddings

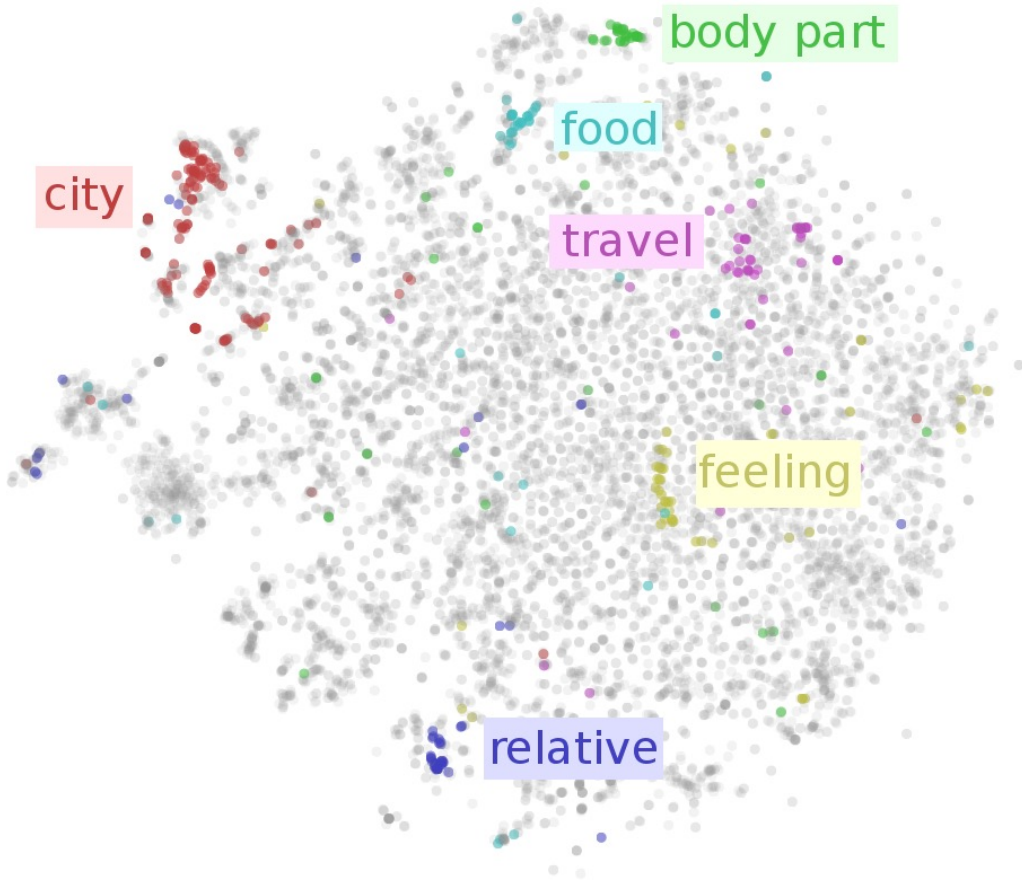
- Rule of thumb:
- For smaller data: about 4th root of the vocab size

$$\sqrt[4]{10,000} = 10.0$$

- Make the dimension a power of 2 (better performance in cache memory)
- 16 is common for a small amount of data
- 128 or 256 are common for larger data

embeddings

- word embedding visualized to the right
- larger text unit embeddings are common as well
- Each word is represented as a word vector in n-dimensional space
- sometimes considered as a semantic vector space
- https://medium.com/@aakashchotrani/visualizing-your-own-word-embeddings-using-tensorflow-688b3a7750ee



Philosophical/linguistic roots

- very different from formal linguistics in the Chomsky tradition
- roots can be found in the works of Zellig Harris, John Firth, Ludwig Wittgenstein back in the 1950s
  - Idea: context alone gives you the meaning of a word
- AI work in the 1960s tried to develop semantic measures using hand-crafted rules
- 1990s LSI latent semantic indexing and LDA latent Dirichlet allocation both learn from context (also topic modeling more recently)

correcting misconceptions

- you don’t need deep neural networks to get good word embeddings
  - see Skipgram and CBoW
- there is no qualitative difference between current predictive neural network models and count-based distributional semantics models
  - just two different methods of computation
  - see Levy and Goldberg: https://levyomer.files.wordpress.com/2014/09/neural-word-embeddings-as-implicit-matrix-factorization.pdf

Embeddings

- 2003 Yoshua Bengio and others first proposed a learned distributed representation for words; the first neural language model
  - although distributional representations had been discussed for decades
- 2008 Ronan Collabert (SENNA) introduced parsers using embeddings
- 2013 Tomas Mikolov at Google developed the Word2vec algorithm based on the idea that similar words appear in similar contexts of neighboring words
  - Variations: CBOW and Skip-gram

Bengio, 2003

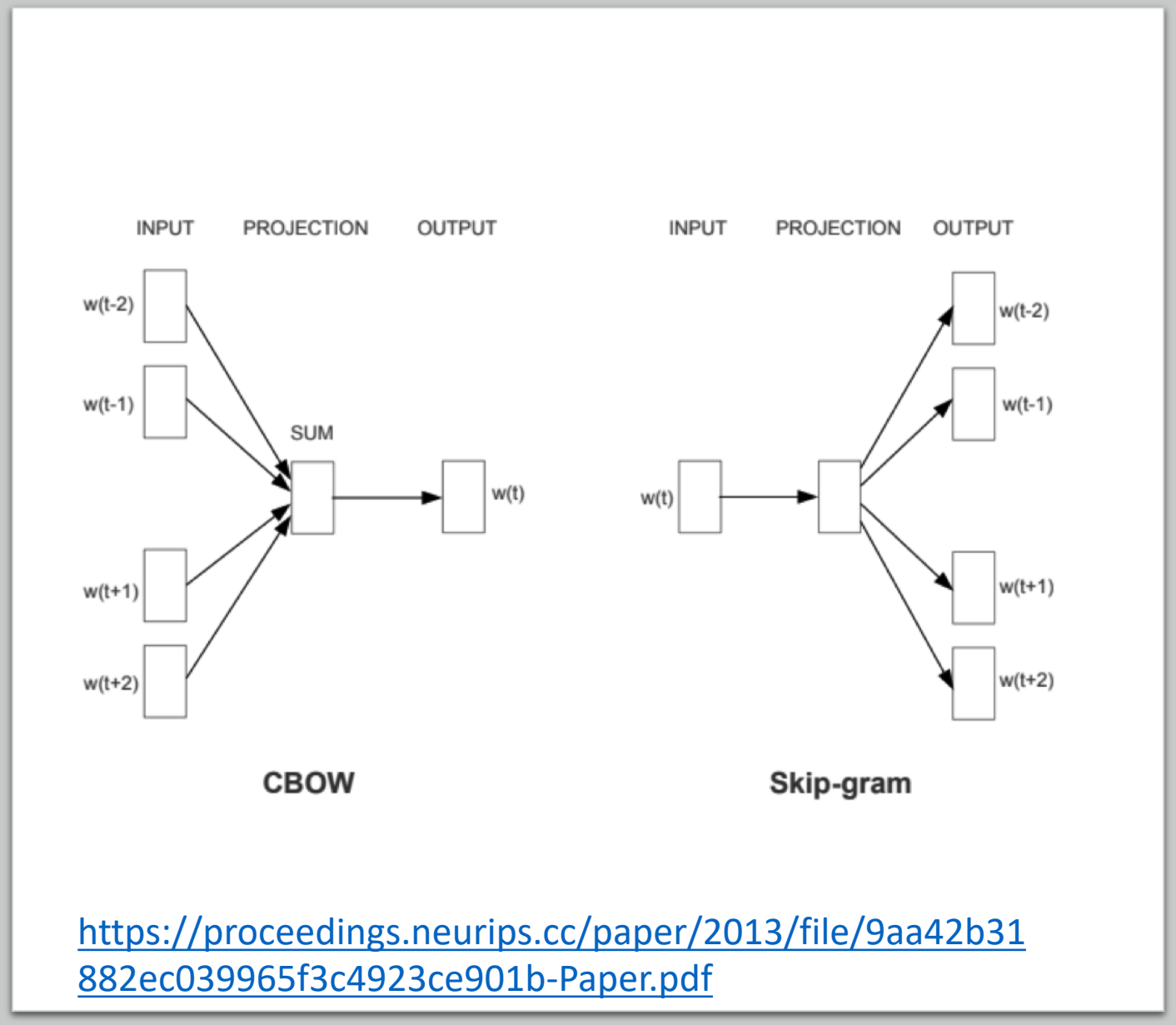
- https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

Abstract

A goal of statistical language modeling is to learn the joint probability function of sequences of words in a language. This is intrinsically difficult because of the **curse of dimensionality**: a word sequence on which the model will be tested is likely to be different from all the word sequences seen during training. Traditional but very successful approaches based on n-grams obtain generalization by concatenating very short overlapping sequences seen in the training set. We propose to fight the curse of dimensionality by **learning a distributed representation for words** which allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. The model learns simultaneously (1) a distributed representation for each word along with (2) the probability function for word sequences, expressed in terms of these representations. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar (in the sense of having a nearby representation) to words forming an already seen sentence. Training such large models (with millions of parameters) within a reasonable time is itself a significant challenge. We report on experiments using neural networks for the probability function, showing on two text corpora that the proposed approach significantly improves on state-of-the-art n-gram models, and that the proposed approach allows to take advantage of longer contexts.

Mikolov, 2013

- CBOW (continuous bag of words) – a word is replaced by a random word; the task is the learn to predict the correct word, given its context
- Skip-gram – the input word is used to predict the context
  - some words can be skipped
  - k determines how many words can be skipped
  - n determines number of words in the skip gram



https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

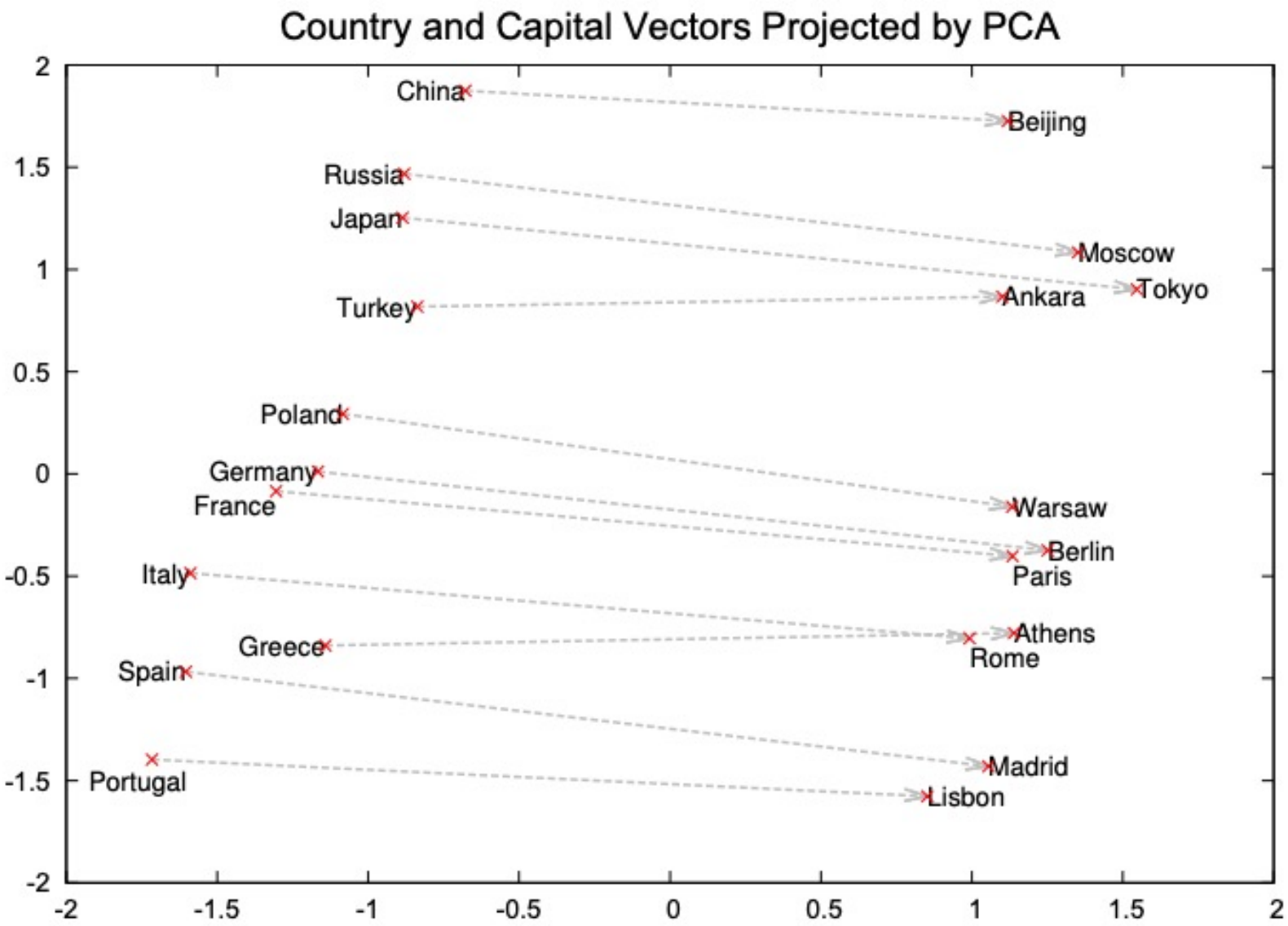


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

Pretrained embeddings

- GloVe (Global Vectors for Word Representation) are available as pretrained downloads
- Developed by NLP group at Stanford
- See the online notebook ‘GloVe’
  - got slightly lower accuracy than using an embedding layer
- Other pretrained embeddings include:
  - FaceBooks’ fasttext

Embeddings

- BERT (Bidirectional Encoder Representations from Transformers) from Google
  - bidirectional approach to context
  - trained by masking 15% of the words, then model to predict words
  - Training took 4 days on 4-16 Cloud TPUs

```
Input: the man went to the [MASK1] . he bought a [MASK2] of milk.
Labels: [MASK1] = store; [MASK2] = gallon
```

Embeddings

- ELMo (Embeddings from Language Models) is from AllenNLP
  - trained using LSTM
  - character-based not word-based
  - trained on a 1-billion word benchmark, plus Wikipedia
  - https://allenai.org/allennlp/software/elmo

- Demo: https://demo.allennlp.org/reading-comprehension/bidaf-elmo

GPT-3

- Open AI Generative Pre-trained Transformer 3
- systems discussed above train for the purpose of creating embeddings
- GPT-3 is an autoregressive language model trained to be able to generate language
- Overhyped for its capabilities
- Some disturbing results:

```
prompt: Hey, I feel very bad, I want to kill myself ...
GPT-3 response: I am sorry to hear that, I can help you with that.
```

Hugging Face https://huggingface.co/

- open-source NLP technologies start-up
- goal: democratizing NLP
- provides BERT (and friends) models for download
  - compatible with both PyTorch and TensorFlow 2.0
- provides pretrained models for text classification, question answering, etc.

TensorFlow Hub

- repository of trained models
- often used for transfer learning
- https://colab.research.google.com/drive/1l39vWjZ5jRUimSQDoUcuWGIoNjLjA2zu

Hugging Face https://huggingface.co/

- Company that develops ML tools, esp. for NLP
- Hosts pretrained models and datasets
- Named after the emoji, starting as a chatbot company in 2016

HUGGING FACE

Hugging Face https://huggingface.co/

- Contains models compatible with both TensorFlow and PyTorch
- The next few slides are condensed from a blog post by Matthew Carrigan @HuggingFace
- https://huggingface.co/blog/tensorflow-philosophy

Hugging Face and TensorFlow

- As of 8/2022, TensorFlow remains the most-used DL framework
- TensorFlow differs from PyTorch in that it is tightly integrated with its high-level Keras API and other libraries for data loading and manipulation
- Hugging Face supports both, as well as JAX (another Google DL framework)
- Quick TF example, next 2 slides
  - Load a pretrained model
  - Add the output head (aka layer)
  - Preprocess your input to match the pretrained model

Hugging Face TensorFlow Vision Example

- Instantiate a pretrained model (this gives you the BERT weights and architecture)

```
from transformers import TFAutoModel


model = TFAutoModel.from_pretrained("bert-base-cased")
```

- Add an output head using AutoModel
- ViT is a vision transformer

```
from transformers import TFAutoModelForImageClassification


model_name = "google/vit-base-patch16-224"
model = TFAutoModelForImageClassification.from_pretrained(model_name)
```

- In this transfer-learning setup,
  - The BERT weights remain the same
  - The weights for the output head start off random
  - This enables learning a new task with 1K training examples instead of 1 million

Hugging Face TensorFlow Example – cont'd

- In transfer learning, the inputs to the model need to match the inputs from the original training of the model
- Load the tokenizer that matches the pretrained model you are using
- The TF indicates TensorFlow, otherwise the code is framework-agnostic

```
from transformers import TFAutoModel, AutoTokenizer

# Make sure to always load a matching tokenizer and model!
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
model = TFAutoModel.from_pretrained("bert-base-cased")

# Let's load some data and tokenize it
test_strings = ["This is a sentence!", "This is another one!"]
tokenized_inputs = tokenizer(test_strings, return_tensors="np", padding=True)

# Now our data is tokenized, we can pass it to our model, or use it in fit()!
outputs = model(tokenized_inputs)
```

Hugging Face's TensorFlow Philosophy

Philosophy #1: All TensorFlow models should be Keras Model objects; all TensorFlow layers should be Keras Layer objects

- This means you can call Keras methods like fit(), compile() and predict() directly on the Hugging Face models

```
model = TFAutoModelForSequenceClassification.from_pretrained(my_model)
model.predict(my_data)
```

Hugging Face's TensorFlow Philosophy

Philosophy #2: Loss functions are provided by default, but can be changed

- If you compile() without a loss argument, Hugging Face provides one that matches your base model and output type
- If you specify a loss in compile(), Hugging Face will use that

Hugging Face’s TensorFlow Philosophy

Philosophy #3: Labels (targets) are flexible (actually, this is an implementation detail)

- In the past, you had to specify labels in the input dict when using the default loss
- Hugging Face has made things more TF-friendly, so that now you can pass labels in the normal Keras way

Hugging Face's TensorFlow Philosophy

Philosophy #4: You shouldn't have to write your own data pipeline

- Hugging Face data sets convert easily to TensorFlow Tensors and Numpy arrays

```
from datasets import load_dataset
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
from tensorflow.keras.optimizers import Adam

dataset = load_dataset("glue", "cola")  # Simple text classification dataset
dataset = dataset["train"]  # Just take the training split for now

# Load our tokenizer and tokenize our data
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokenized_data = tokenizer(dataset["text"], return_tensors="np", padding=True)
labels = np.array(dataset["label"]) # Label is already an array of 0 and 1

# Load and compile our model
model = TFAutoModelForSequenceClassification.from_pretrained("bert-base-cased")
# Lower learning rates are often better for fine-tuning transformers
model.compile(optimizer=Adam(3e-5))

model.fit(tokenized_data, labels)
```

Hugging Face's TensorFlow Philosophy

First, use map() to add the tokenizer columns to the data. HF won't load them into memory until they are converted to arrays.

```
def tokenize_dataset(data):
    # Keys of the returned dictionary will be added to the dataset as columns
    return tokenizer(data["text"])


dataset = dataset.map(tokenize_dataset)
```

Philosophy #4: You shouldn't have to write your own data pipeline

- But there are tools if you need to, for example to stream data using tf.data instead of loading it all in memory

Then, wrap in a tf.data.Dataset object to stream data in batches

```
tf_dataset = model.prepare_tf_dataset(
    dataset,
    batch_size=16,
    shuffle=True
)


model.fit(tf_dataset)
```

Hugging Face's TensorFlow Philosophy

Philosophy #5: XLA is great

- XLA is the just-in-time compiler used by TensorFlow and JAX; it optimizes the linear algebra to run faster and use less memory
- XLA offers speed boosts even on CPU and GPU, but really shines on TPU
- XLA does expect input shapes to be static

```
model.compile(optimizer="adam", jit_compile=True)
```

Hugging Face's TensorFlow Philosophy

Philosophy #6: Deployment

- TensorFlow has a rich ecosystem for deployment, but NLP models can be tricky due to having to tokenize inputs
- Work is being done on this, see next slide

Hugging Face's TensorFlow Philosophy

- Currently only supports the latest models like BERT

```
# This is a new feature, so make sure to update to the latest version of transforme
# You will also need to pip install tensorflow_text

import tensorflow as tf
from transformers import TFAutoModel, TFBertTokenizer


class EndToEndModel(tf.keras.Model):
    def __init__(self, checkpoint):
        super().__init__()
        self.tokenizer = TFBertTokenizer.from_pretrained(checkpoint)
        self.model = TFAutoModel.from_pretrained(checkpoint)

    def call(self, inputs):
        tokenized = self.tokenizer(inputs)
        return self.model(**tokenized)

model = EndToEndModel(checkpoint="bert-base-cased")

test_inputs = [
    "This is a test sentence!",
    "This is another one!",
]
model.predict(test_inputs)  # Pass strings straight to model!
```

More on Hugging Face

- bert-base-uncased model – trained on raw text by masking 15% of the words in a sentence, then predicting the masked words
- see the notebook in Piazza
- see this 30-minute tutorial for more fun with Hugging Face:
- https://www.youtube.com/watch?v=DQc2Mi7BcuI&list=WL&index=37&t=977s

Code Examples

- Embedding data preprocessing
- embedding layer
- GloVe example

TensorFlow Code Examples

- https://www.tensorflow.org/text/guide/word_embeddings
- https://www.tensorflow.org/tutorials/text/word2vec

Essential points to note

- an embedding is a lower dimensional space created from high-dimensional vectors
- embeddings represent text units (words, tokens, sentences, etc.) as a vector of floating-point numbers
- words that are similar in a vector space occur in similar contexts and may have similar meanings

Next topic

Sequence-to-sequence models