Deep Learning

Jeff Prosise

@jprosise

Deep Learning

Deep learning is a **superpower**. With it, you can make a computer **see**, synthesize **art**, translate **languages**, render a medical **diagnosis**, or build pieces of a car that can **drive itself**. If that isn't a superpower, I don't know what is.

— **Andrew Ng**, Stanford University

The Big Picture

1950s – Researchers begin pondering whether computers can think like humans and devise **Symbolic AI** – for example, chess programs

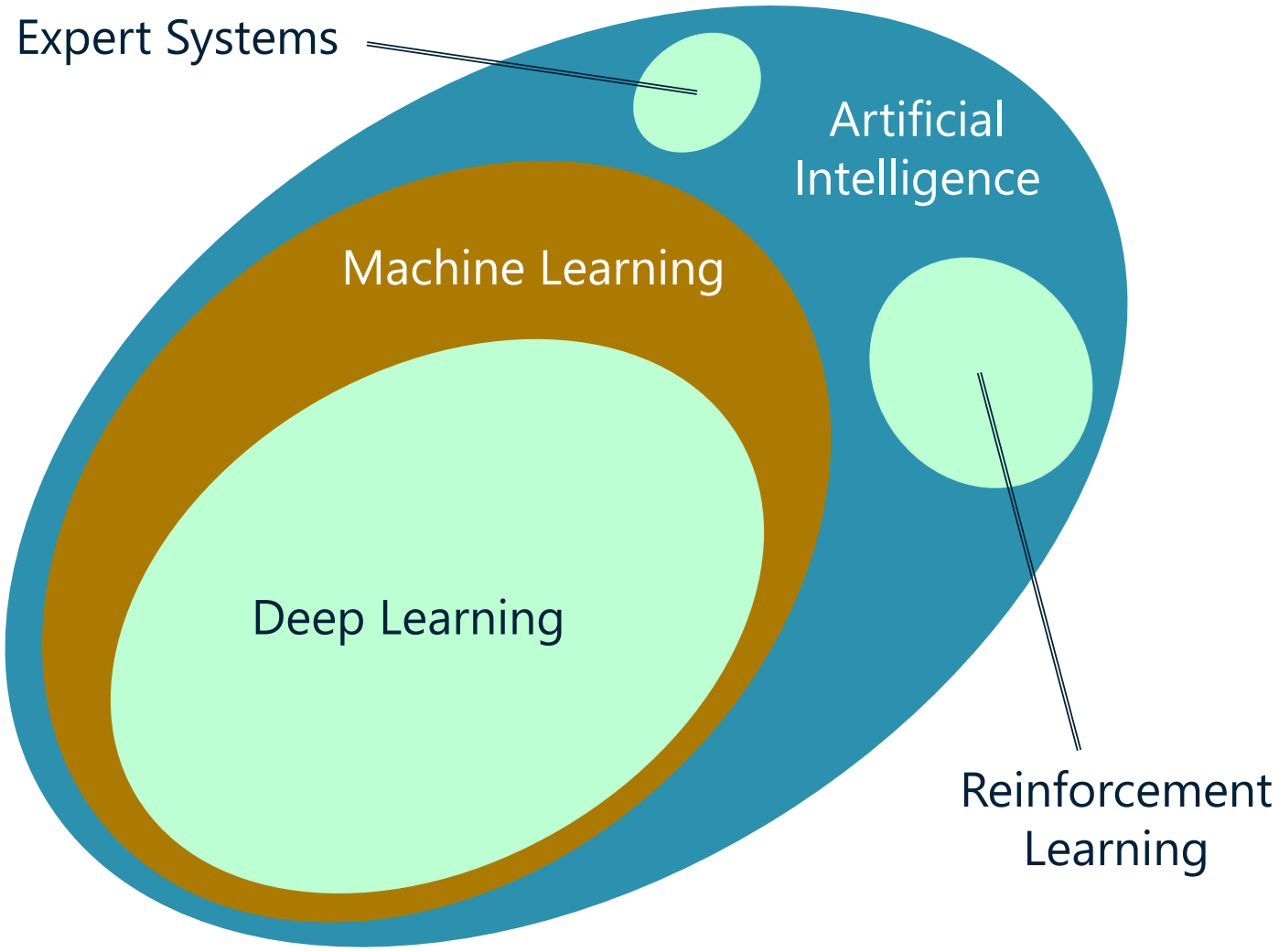
1960s – Early **neural networks** are devised, but training is impractical due to lack of computing power and access to compute resources

1980s – Researchers improve **backpropagation algorithms** for training neural networks and apply them to convolutional neural networks

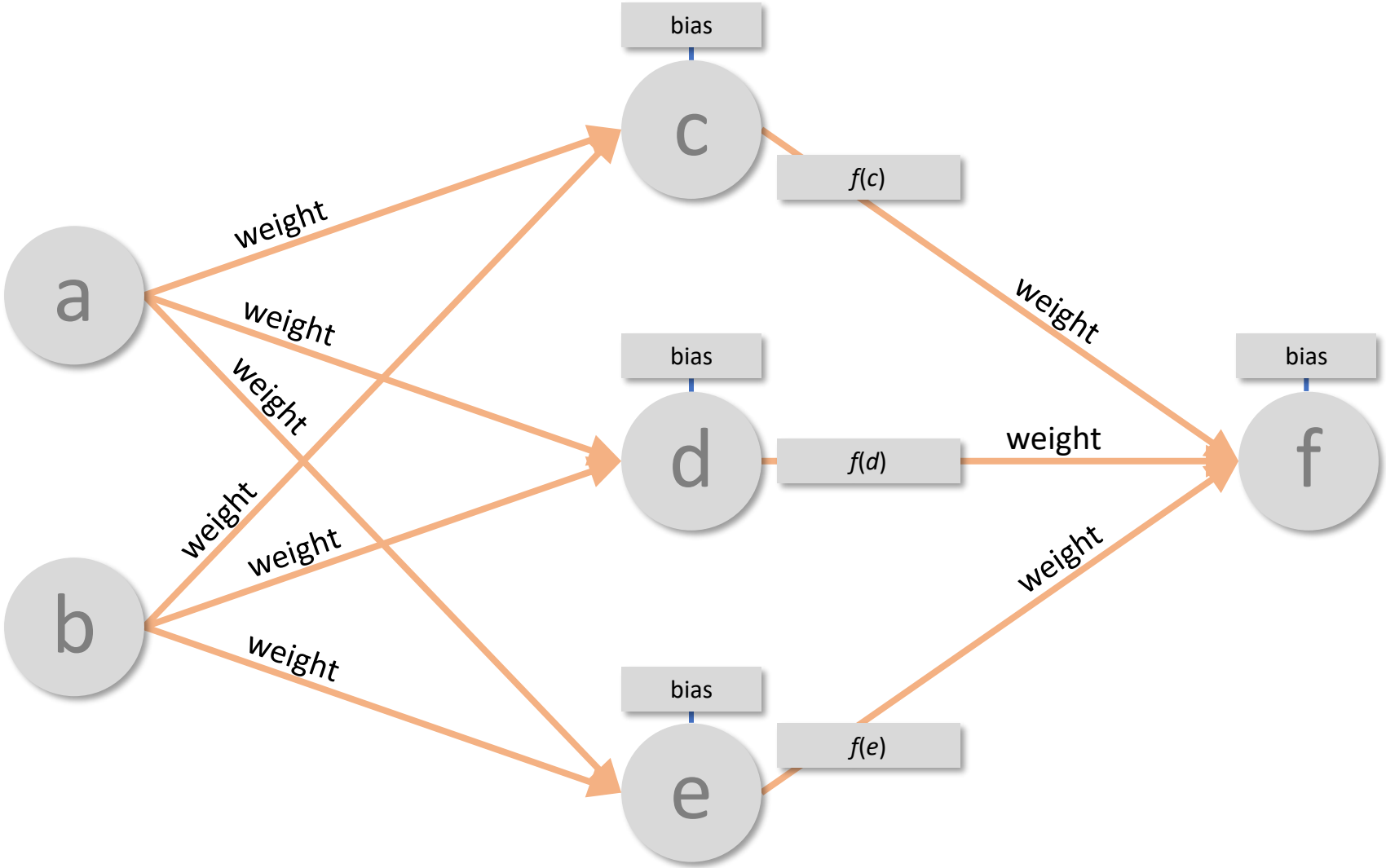
1990s – Researchers at Bell Labs invent the modern **Support Vector Machine** (SVM) algorithm for discovering decision boundaries

2000s – Algorithmic advances in machine learning give rise to **decision trees, random forests**, **gradient-boosting machines**, and more

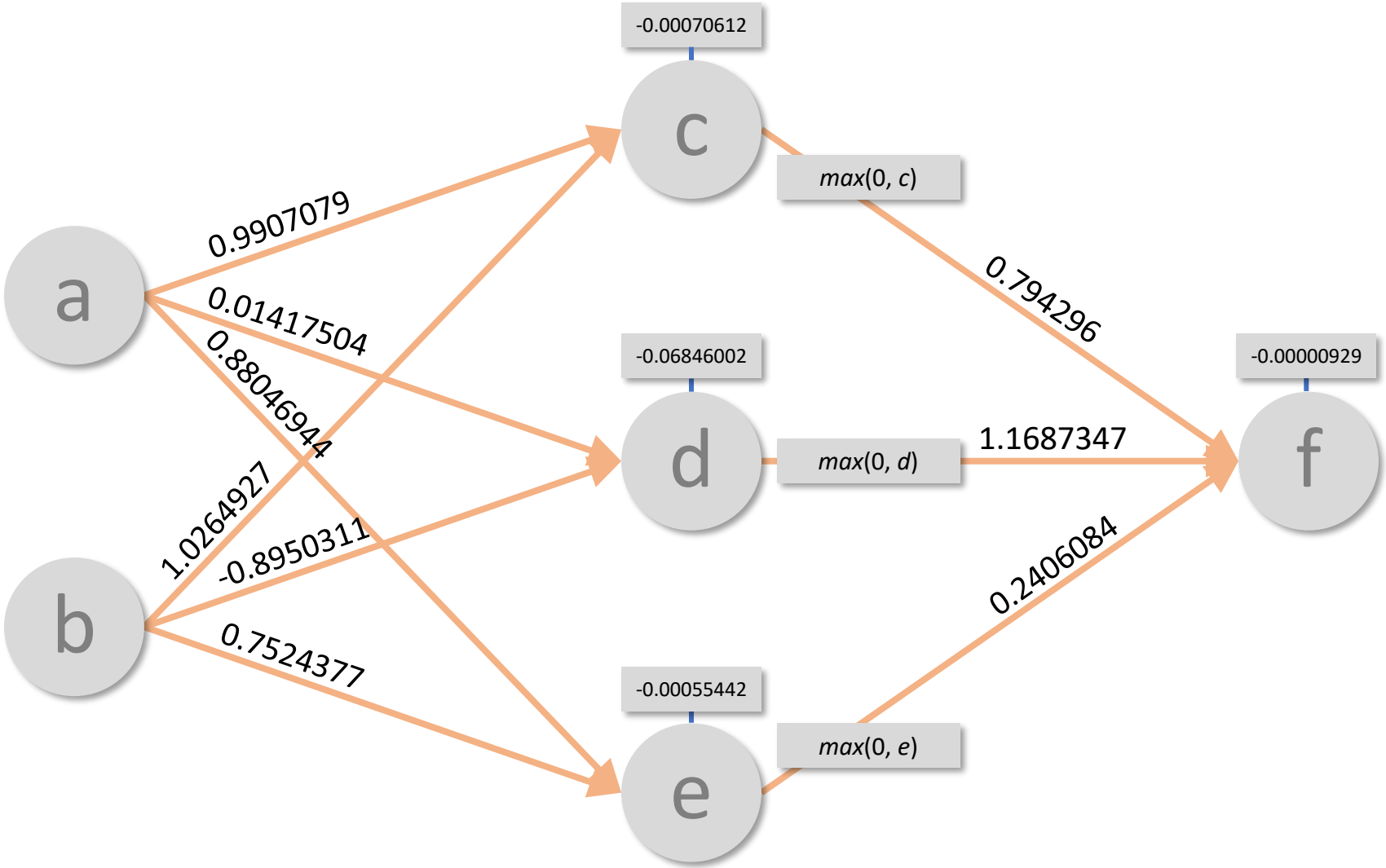
2010s – Deep learning explodes due to faster computing hardware (**GPUs**), expanded availability of data, and increased research funding



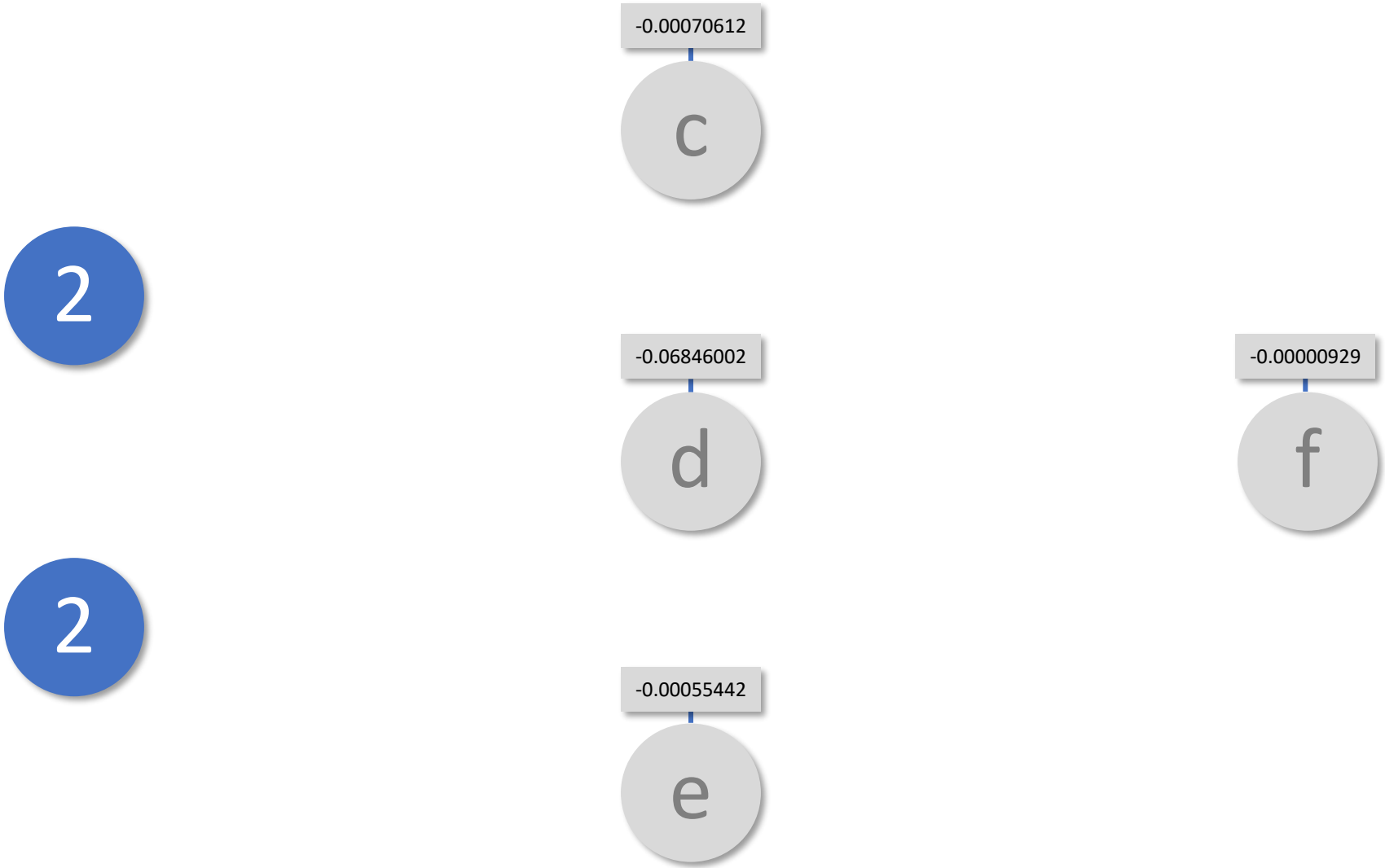
How Neural Networks Work



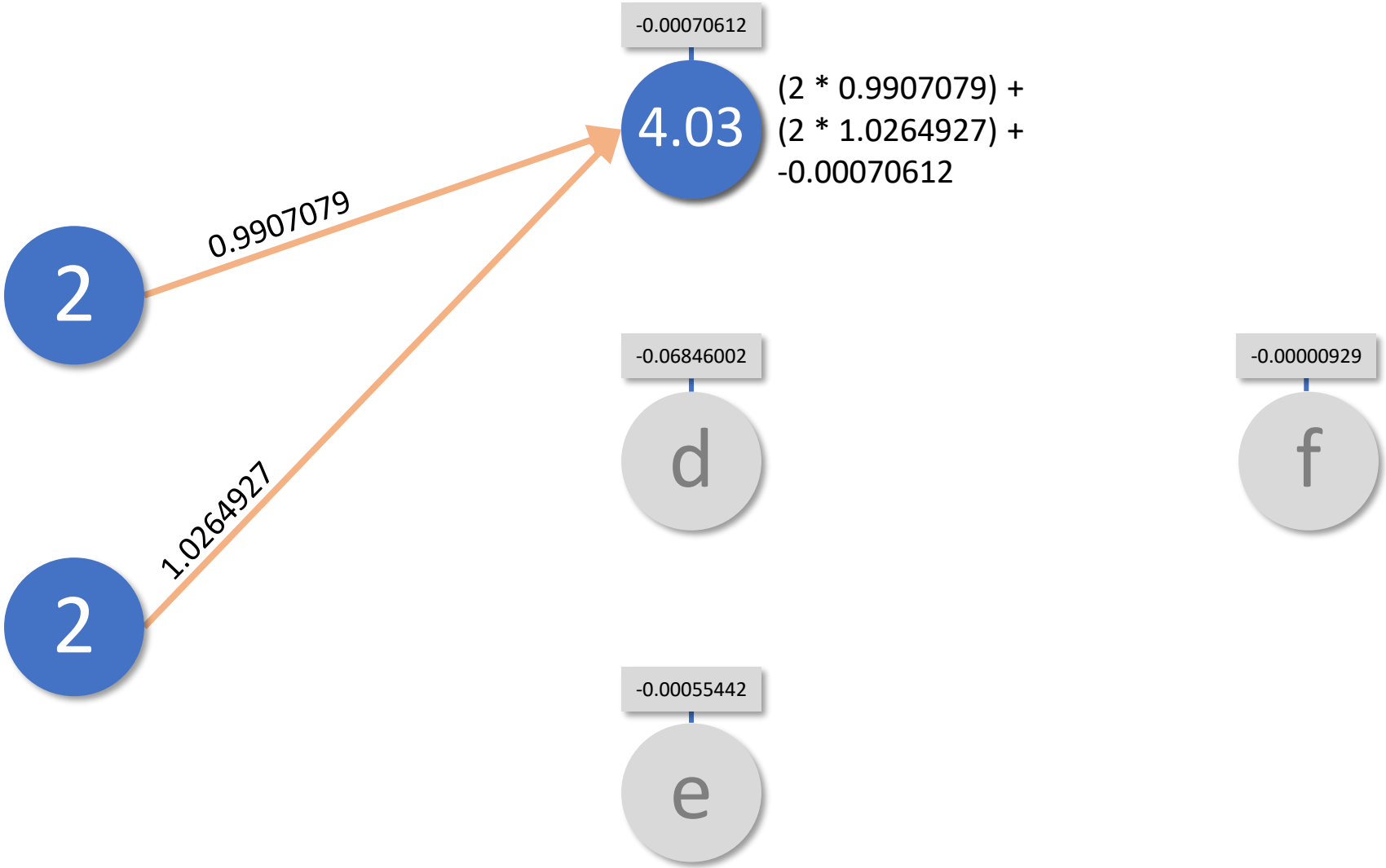
How Neural Networks Work, Cont.



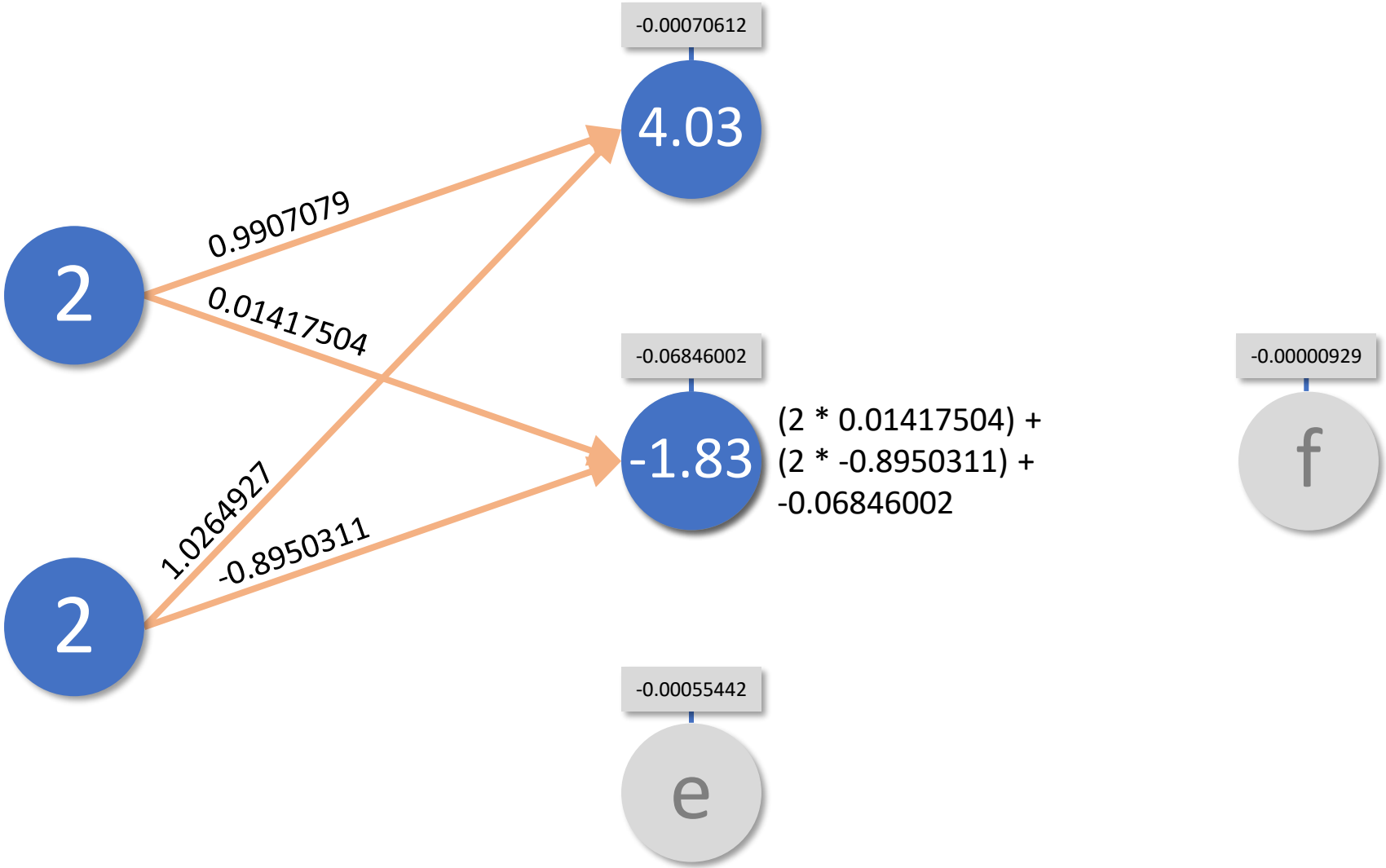
How Neural Networks Work, Cont.



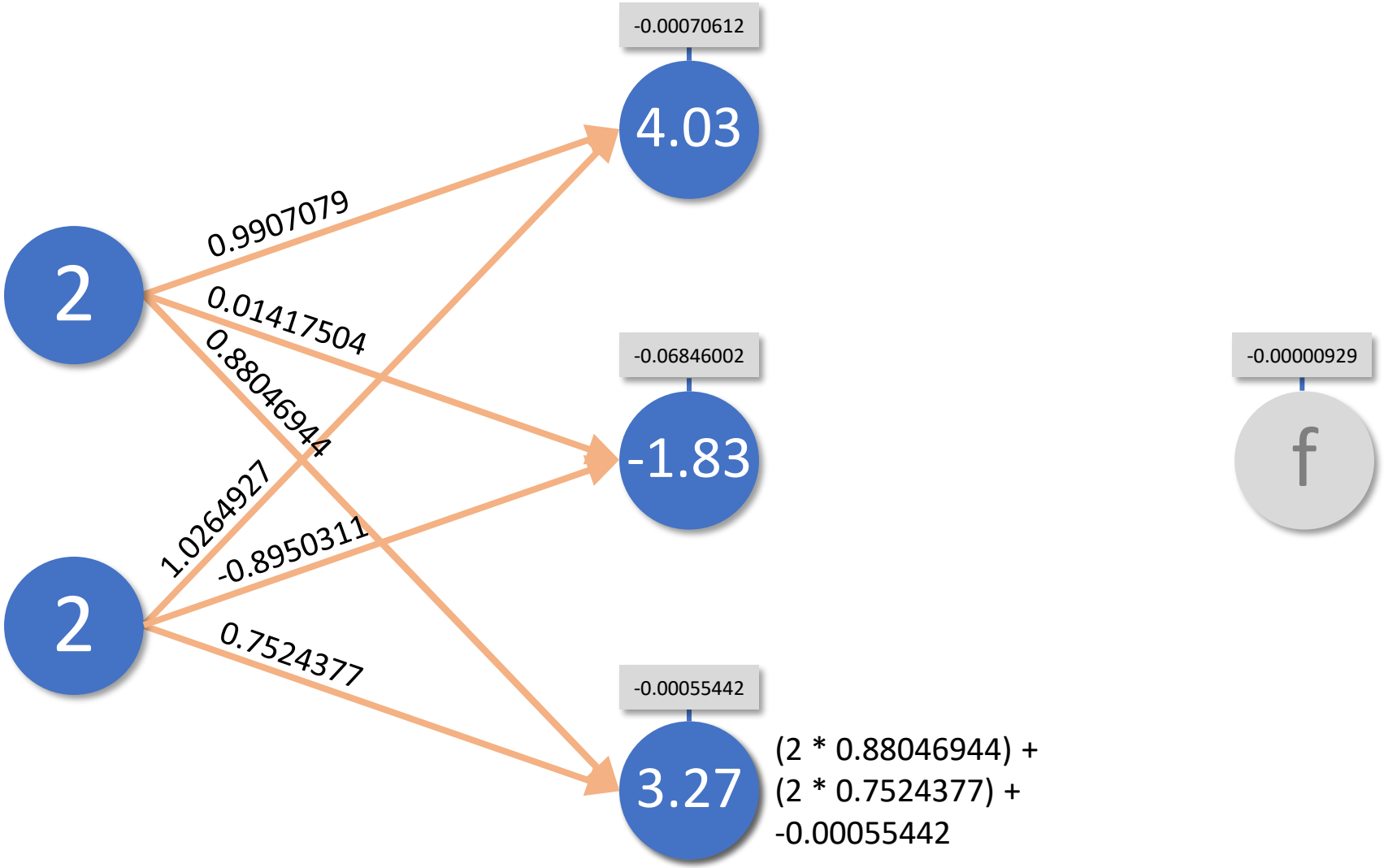
How Neural Networks Work, Cont.



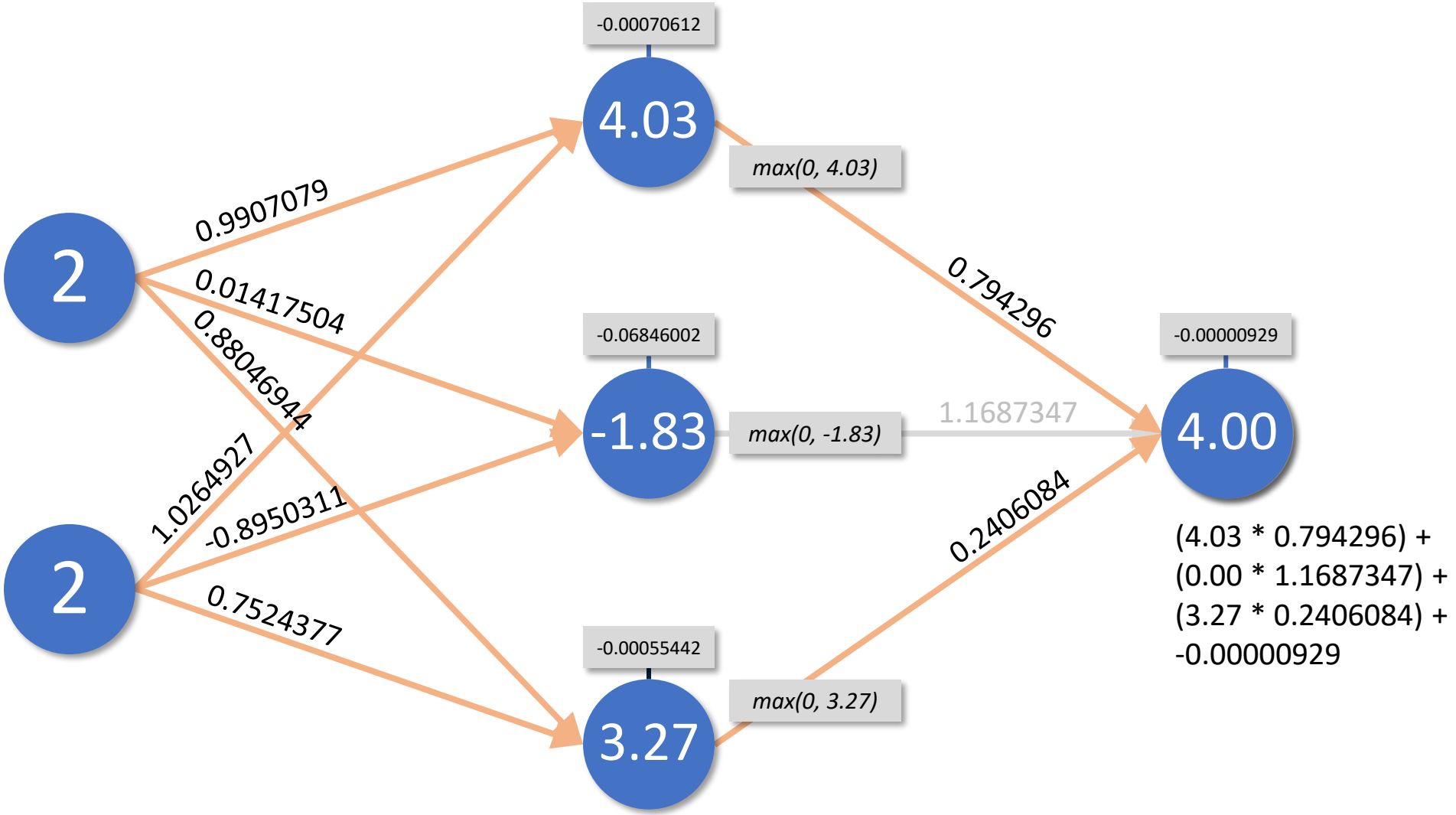
How Neural Networks Work, Cont.



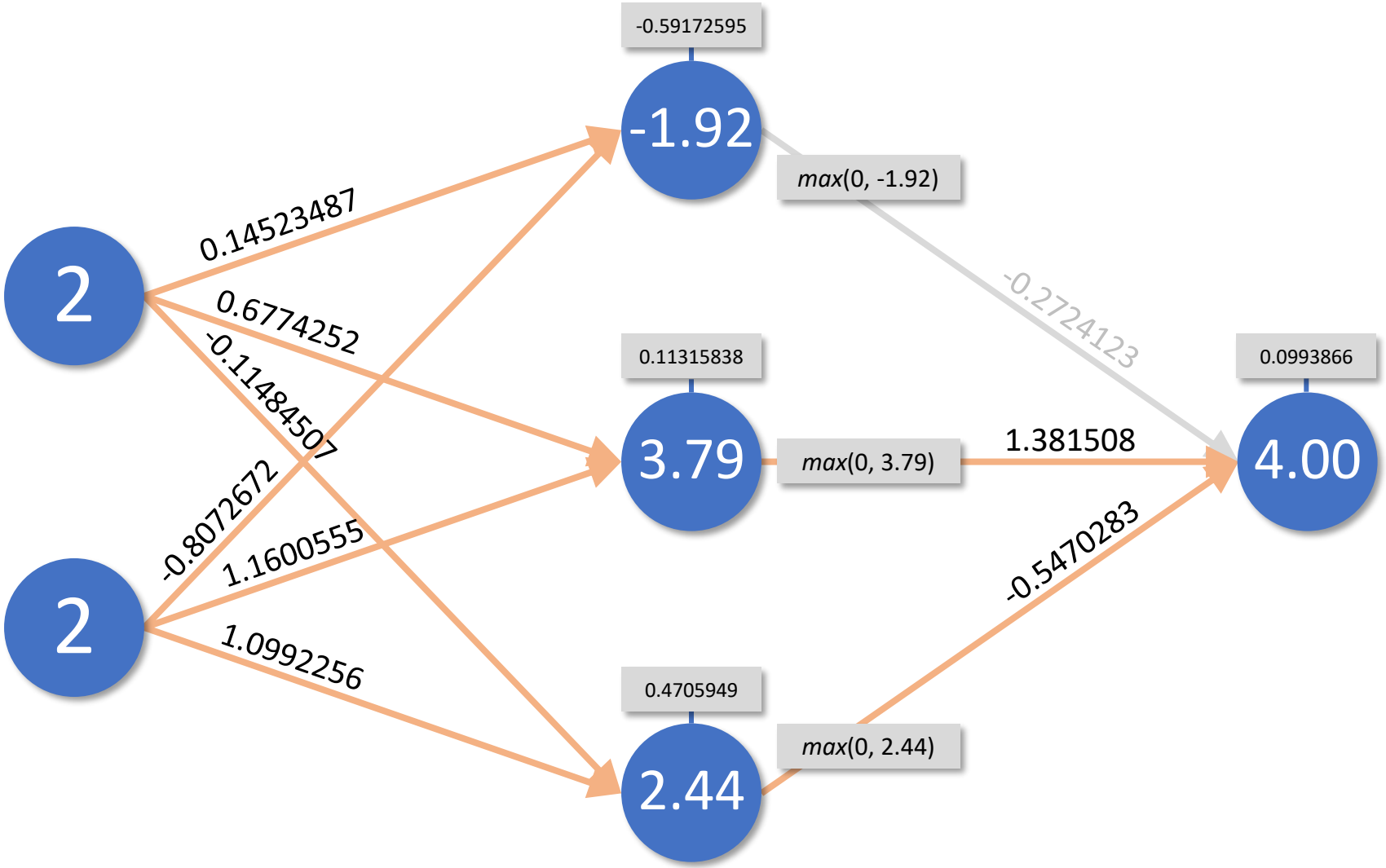
How Neural Networks Work, Cont.



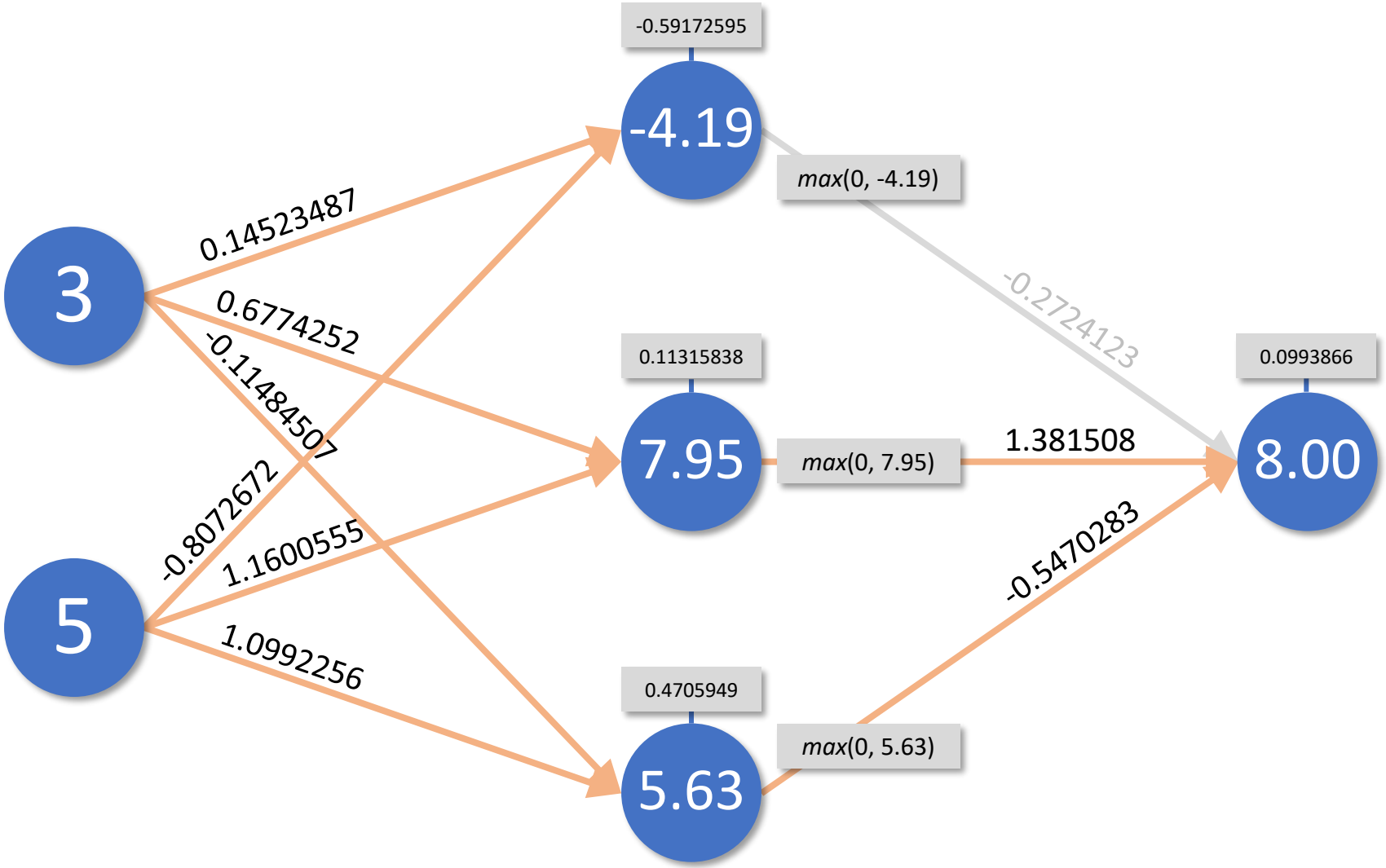
How Neural Networks Work, Cont.



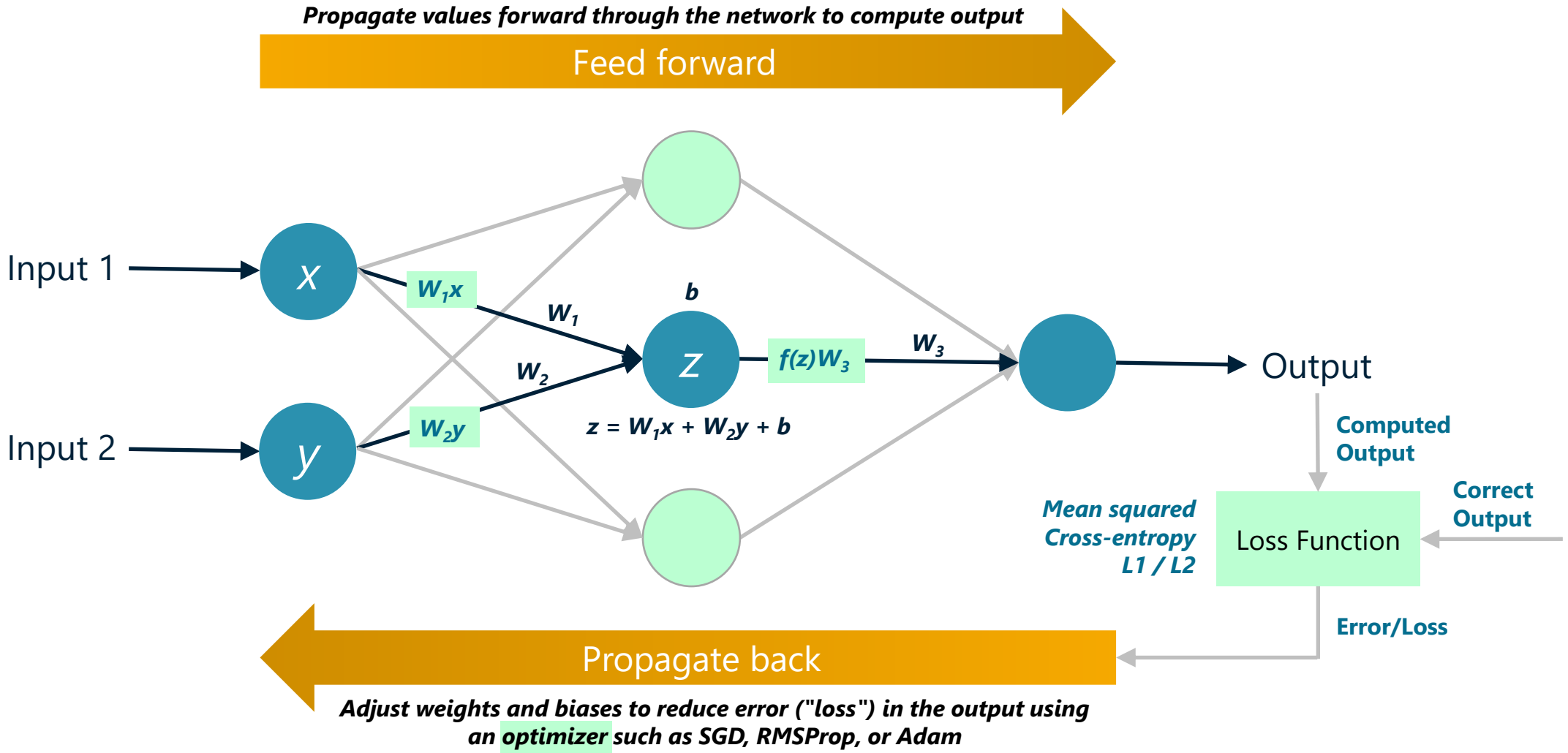
Different Weights and Biases, Same Result



Changing the Inputs

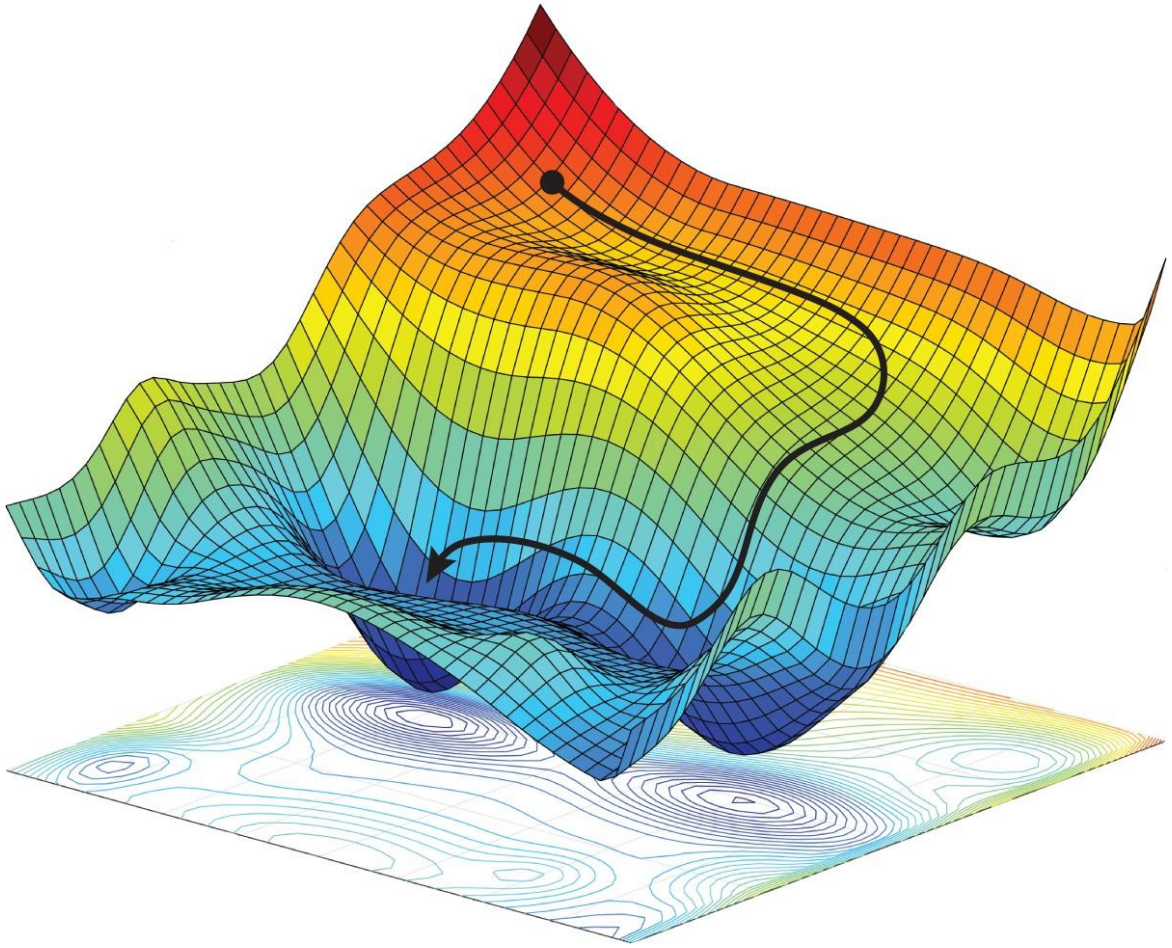


Backpropagation



Gradient Descent

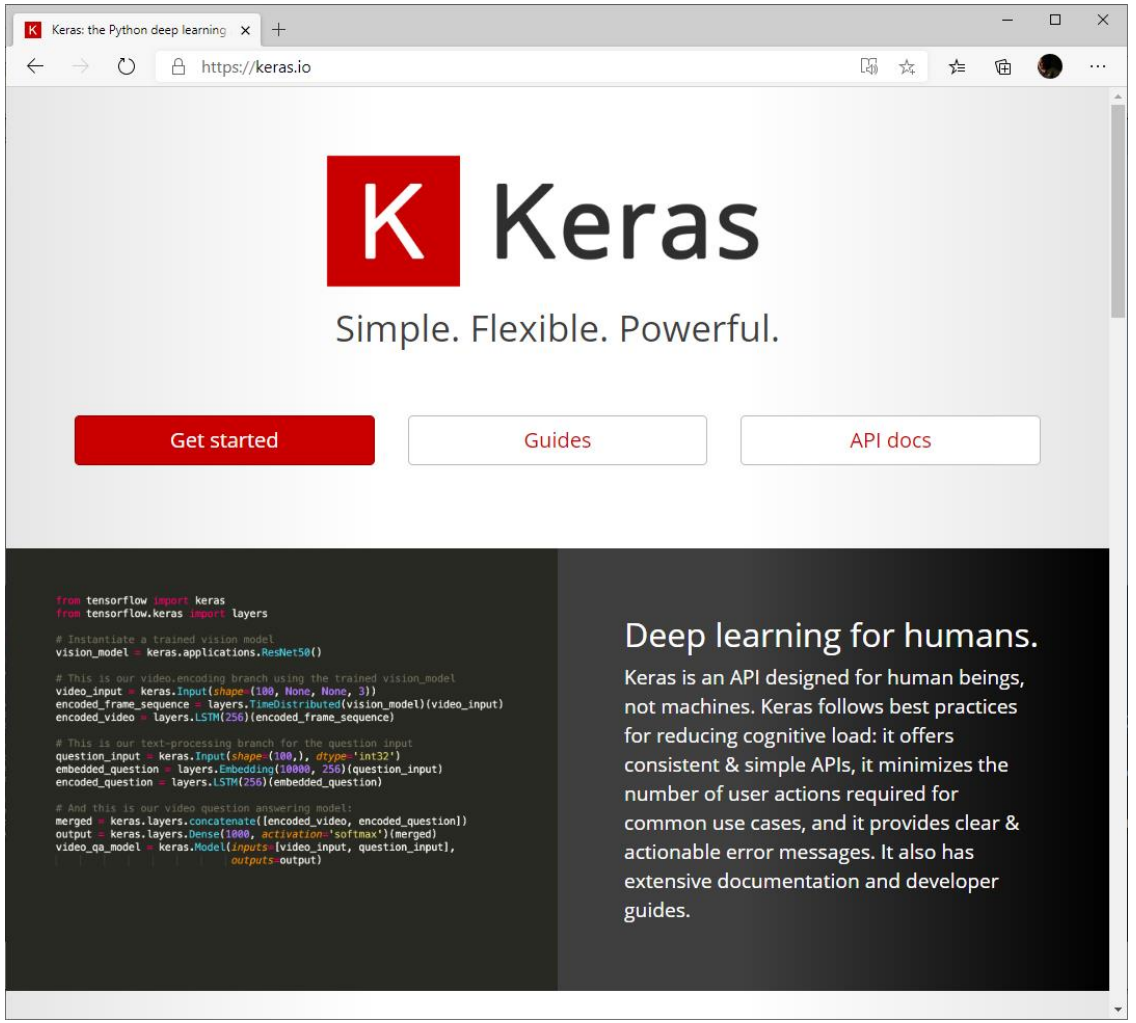
- Training involves navigating multi-dimensional loss landscape searching for global minimum
- Weights and biases are adjusted using gradients computed from partial derivatives
  - Partial derivatives of loss function with respect to individual weights
- Gradients are multiplied by learning rate to make steps incremental



Source: https://arxiv.org/pdf/1805.04829 ("Spatial Uncertainty Sampling for End-to-End Control")

Keras

- "Scikit for neural networks"
- Free, open-source, and with support for GPUs and TPUs
- Provides high-level APIs over TensorFlow, Theano, and CNTK
  - Sequential API for most models
  - Functional API for advanced models (e.g., shared layers)
- https://keras.io/



Building a Neural Network with Keras

```
from keras.layers import Dense
from keras.models import Sequential

model = Sequential()
model.add(Dense(16, activation='relu', input_dim=2)) # Each input contains 2 values
model.add(Dense(16, activation='relu'))
model.add(Dense(1)) # Single numeric output
model.compile(loss='mae', optimizer='adam', metrics=['mae'])
```

Component responsible for adjusting weights and biases during training

Training a Neural Network

```
# Train without validation
model.fit(x, y, epochs=10, batch_size=50)

# Train with 80% of the input data and validate with 20%, with Keras doing the split
model.fit(x, y, validation_split=0.2, epochs=10, batch_size=50)

# Train using explicit datasets for training and validation
model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10, batch_size=50)

# Evaluate the model's accuracy separately
scores = model.evaluate(x_test, y_test, verbose=0)
```

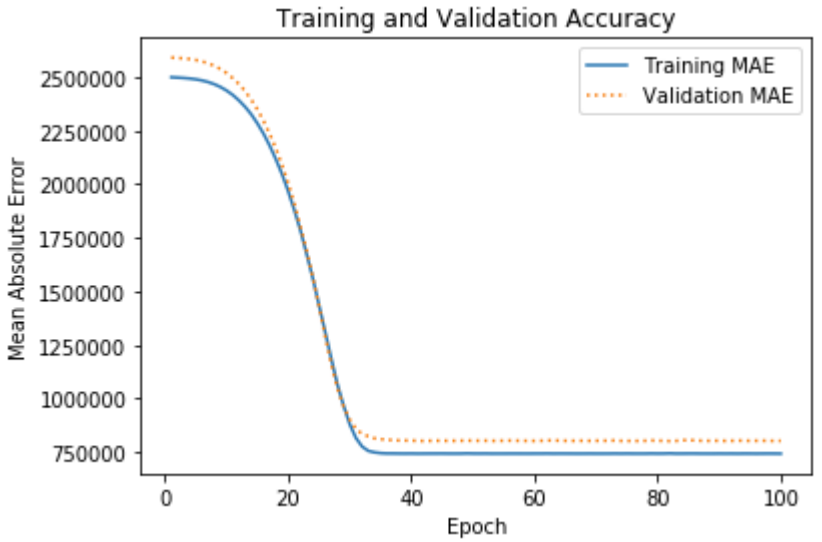
Batch Size

- How often should weights and biases be adjusted during training?
  - After every training sample
  - After every *n* training samples
  - After every epoch
- Answer: After every *n* training samples, where *n* is the *batch size*
  - Small batch sizes increase training time but may increase accuracy, too
  - Large batch sizes decrease training time but may decrease accuracy
- Experiment to find the right balance

Plotting Accuracy and Validation Accuracy

```
# Determine the right number of epochs and check for overfitting
hist = model.fit(x, y, validation_split=0.2, epochs=100, batch_size=50)
err = hist.history['mae']
val_err = hist.history['val_mae']
epochs = range(1, len(err) + 1)

plt.plot(epochs, err, '-', label='Training MAE')
plt.plot(epochs, val_err, ':', label='Validation MAE')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Mean Absolute Error')
plt.legend(loc='upper right')
plt.plot()
```



Making a Prediction

```
# Assumes network expects two input values
model.predict(np.array([[1.0, 2.0]]))
```

Demo

Regression

Binary Classification

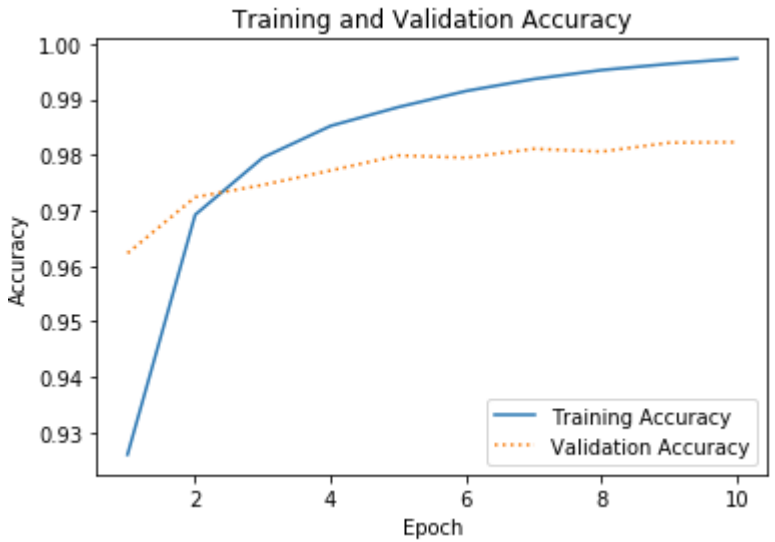
- Include one neuron in the output layer with **sigmoid** activation
  - Output is probability that the input belongs to the positive class (0.0 to 1.0)
- Use **binary_crossentropy** as the loss function
- Use 0s for the negative class and 1s for the positive class

```
model = Sequential()
model.add(Dense(16, activation='relu', input_dim=2))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid')) # Probability from 0.0 to 1.0
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(x, y, validation_split=0.2, epochs=10, batch_size=50)
```

Plotting Accuracy for a Classifier

```
hist = model.fit(x, y, validation_split=0.2, epochs=10, batch_size=50)
acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, '-', label='Training Accuracy')
plt.plot(epochs, val_acc, ':', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.plot()
```



Making a Prediction

```
# Get the probability that the input belongs to the positive class
model.predict(np.array([[1.0, 2.0]]))

# Get the predicted class
model.predict_classes(np.array([[1.0, 2.0]]))
(model.predict(np.array([[1.0, 2.0]])) > 0.5).astype('int32')
```

Demo

Binary Classification

Multiclass Classification

- Use one neuron per class in the output layer with **softmax** activation
  - Output is an array of per-class probabilities that add up to 1.0
- Use **sparse_categorical_crossentropy** as the loss function

```
model = Sequential()
model.add(Dense(16, activation='relu', input_dim=2))
model.add(Dense(16, activation='relu'))
model.add(Dense(10, activation='softmax')) # 10 possible classes
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
model.fit(x, y, validation_split=0.2, epochs=10, batch_size=50)
```

Making a Prediction

```
# Get the probabilities for each class
model.predict(np.array([[1.0, 2.0]]))

# Get the predicted class
model.predict_classes(np.array([[1.0, 2.0]]))
np.argmax(model.predict(np.array([[1.0, 2.0]])), axis=-1)
```

Demo

Multiclass Classification

Natural Language Processing (NLP)

- Using deep-learning models to process human language

- Capabilities have grown exponentially in recent years thanks to new neural architectures, including transformer encoder-decoders
- Key concepts: Word embeddings, neural attention, and self-attention

Tokenizing Text

```
lines = ['Quick brown fox', # Stop words removed
         'Jumps over lazy lazy brown dog']

tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
vectors = tokenizer.texts_to_matrix(lines)
```

Vocabulary

brown	1
lazy	2
quick	3
fox	4
jumps	5
over	6
dog	7

	brown	lazy	quick	fox	jumps	over	dog
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	1

Turning Text into Sequences

```
lines = ['Quick brown fox', # Stop words removed
         'Jumps over lazy lazy brown dog']

tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
padded_sequences = pad_sequences(sequences, 5)
```

Vocabulary

brown	1
lazy	2
quick	3
fox	4
jumps	5
over	6
dog	7

0	0	3	1	4
6	2	2	1	7

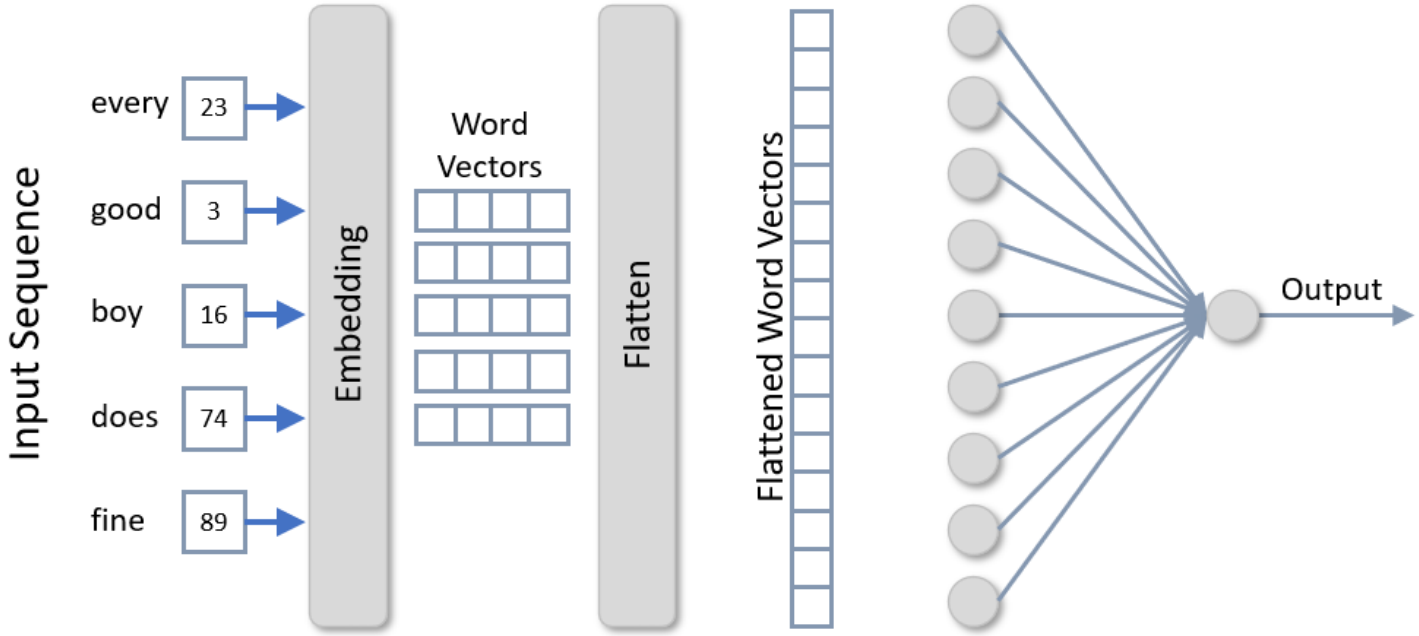
Padded Sequences

5

Word Embeddings

- Embedding layers turn sequences of word indexes into arrays of *word vectors*, which encode information about relationships between words
- Keras makes this easy with its **Emdedding** class

Lines of text are input as **sequences**, which are arrays of integers representing individual words (e.g., indices into a dictionary). An **embedding layer** transforms integers representing words into **word vectors**, or arrays of floating-point numbers. Word vectors encode information about **relationships between words**, such as the fact that both "excellent" and "amazing" express positive sentiment.



Using an Embedding Layer

```
model = Sequential()
model.add(Embedding(10000, 32, input_length=500))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(x, y, validation_split=0.2, epochs=10, batch_size=50)
```

Vocabulary length = 10,000
Output dimension = 32
Length of each sequence = 500

Making Predictions

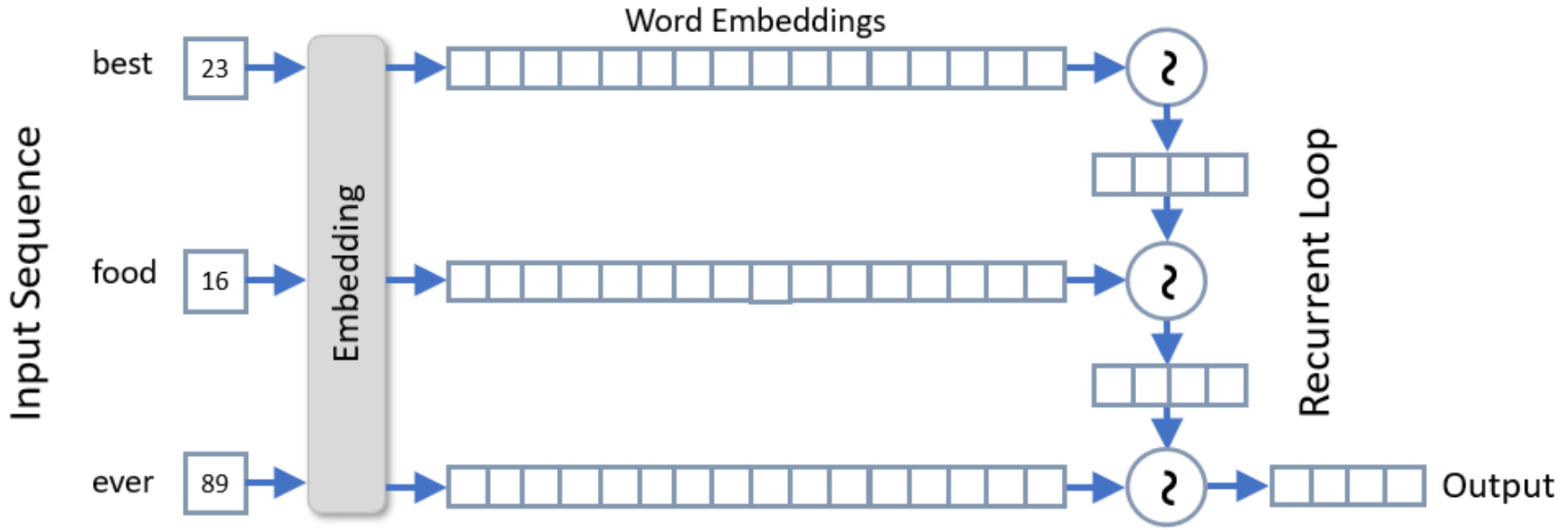
```
sequence = tokenizer.texts_to_sequences(['Can you attend a code review on Tuesday?'])
padded_sequence = pad_sequences(sequence, maxlen=500)
model.predict(padded_sequence)
```

Demo

Spam Filtering

Recurrent Neural Networks

- Carry information (context) forward as a sequence is processed



Tokenized phrase is input to the embedding layer as a **sequence of word indexes**

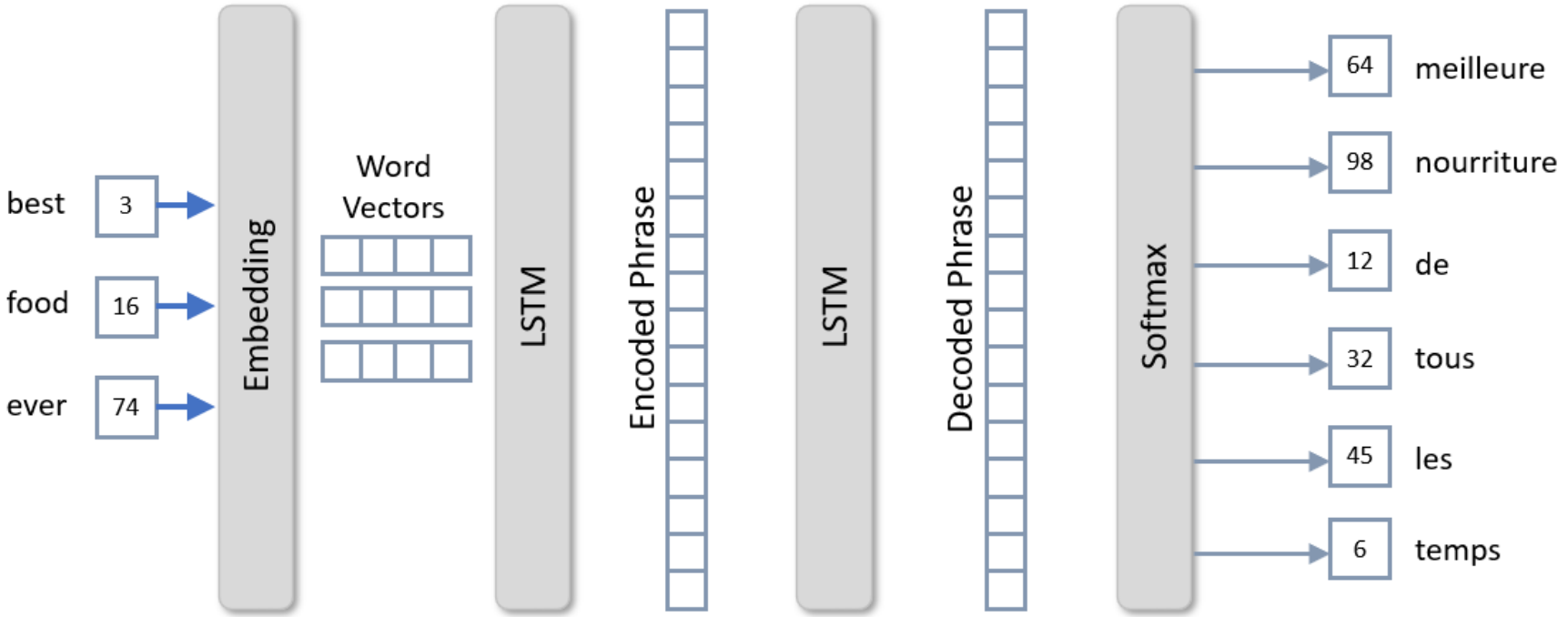
Each token is converted into a **word embedding**, which is an array (vector) of floating-point values modeling each word's **relationship to other words**

A recurrent layer **loops** through the word embeddings in the sequence, computing a value for each that **factors in the output from the previous iteration**

Neural Machine Translation (NMT)

- NMT models translate text from one language to another
  - Superior to rules-based machine translation (RBMT)
  - Superior to statistical machine translation (SMT)
- Accomplished today with either (or a hybrid) of two architectures
  - LSTM encoder-decoders (prevalent 2012 to 2017)
  - Transformer encoder-decoders (2017 to present)
- RNNs are sequence-to-sequence models that accept a tokenized sequence as input and generate a tokenized sequence as output
  - For example, tokenized English sentence -> tokenized French sentence

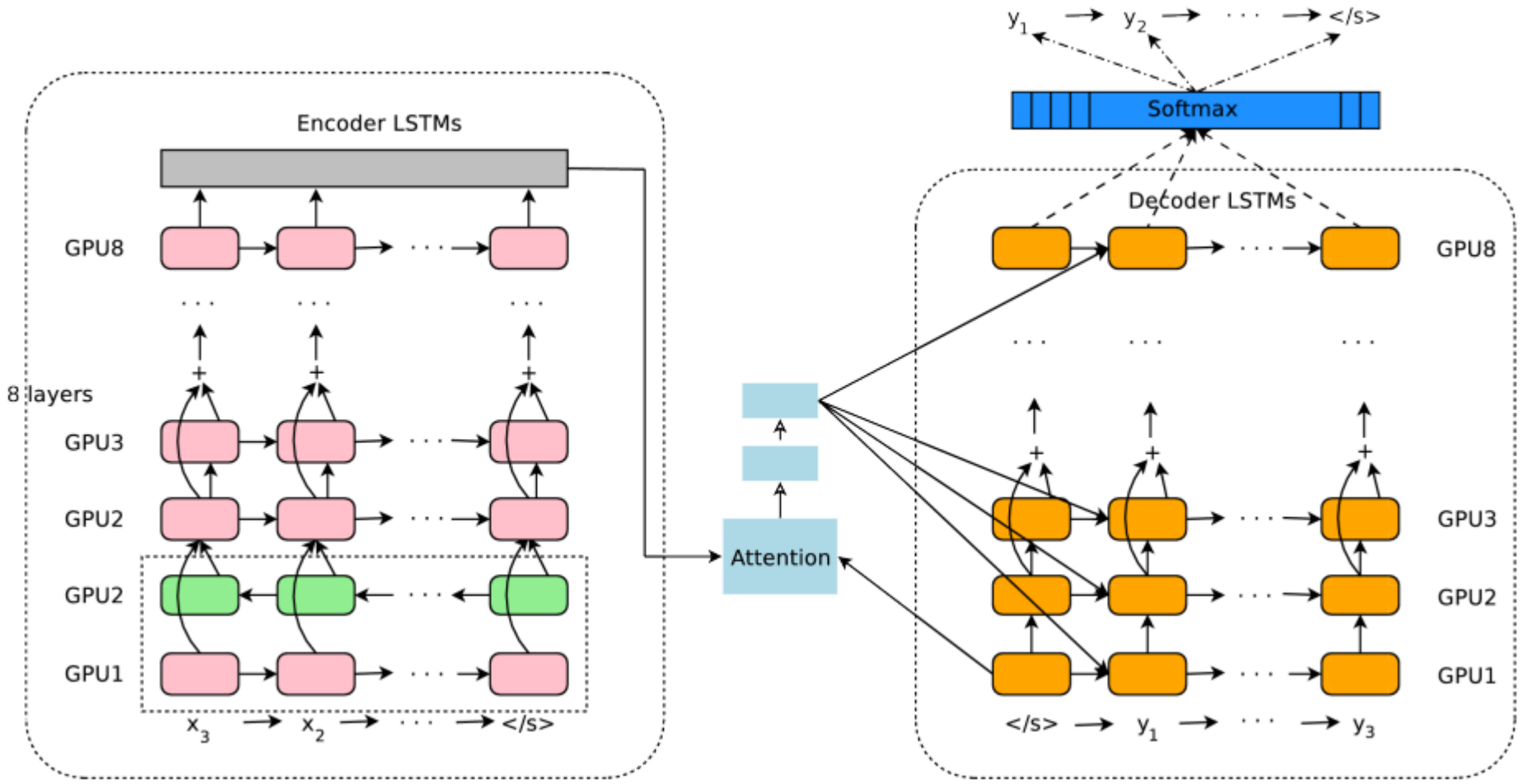
LSTM Encoder-Decoder Architecture



Tokenized input is **transformed to word vectors** by an embedding layer. Word vectors are input to an **LSTM encoder** that produces a dense vector representation of the input phrase.

An **LSTM decoder** transforms the vector generated by the encoder into a dense vector representing the translated phrase. A **softmax output layer** translates the vector into a **set of probabilities**. The word assigned to each position in the output sequence is the word in the vocabulary **assigned the highest probability**.

Google Translate circa 2016



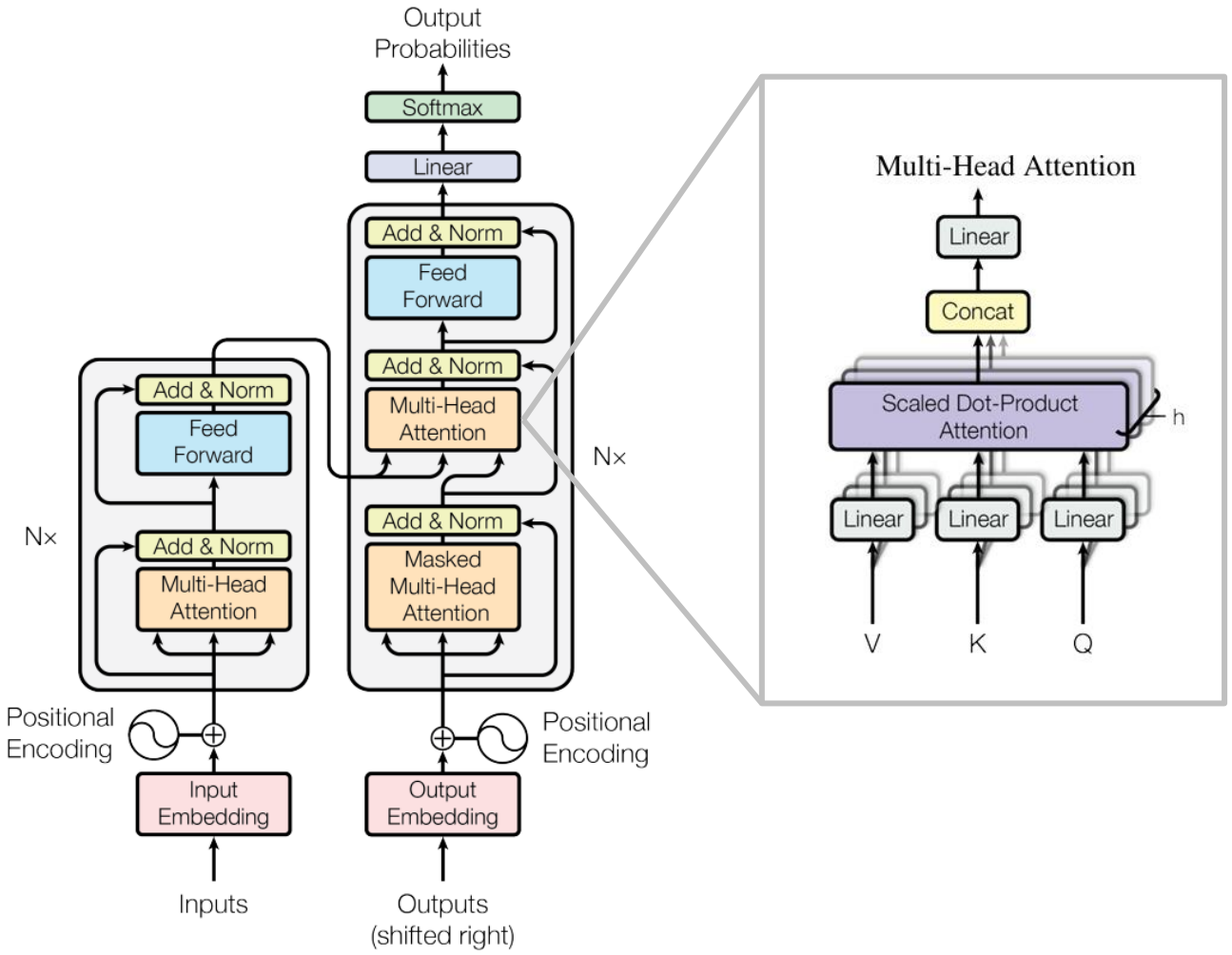
"Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation" (https://arxiv.org/abs/1609.08144)

Demo

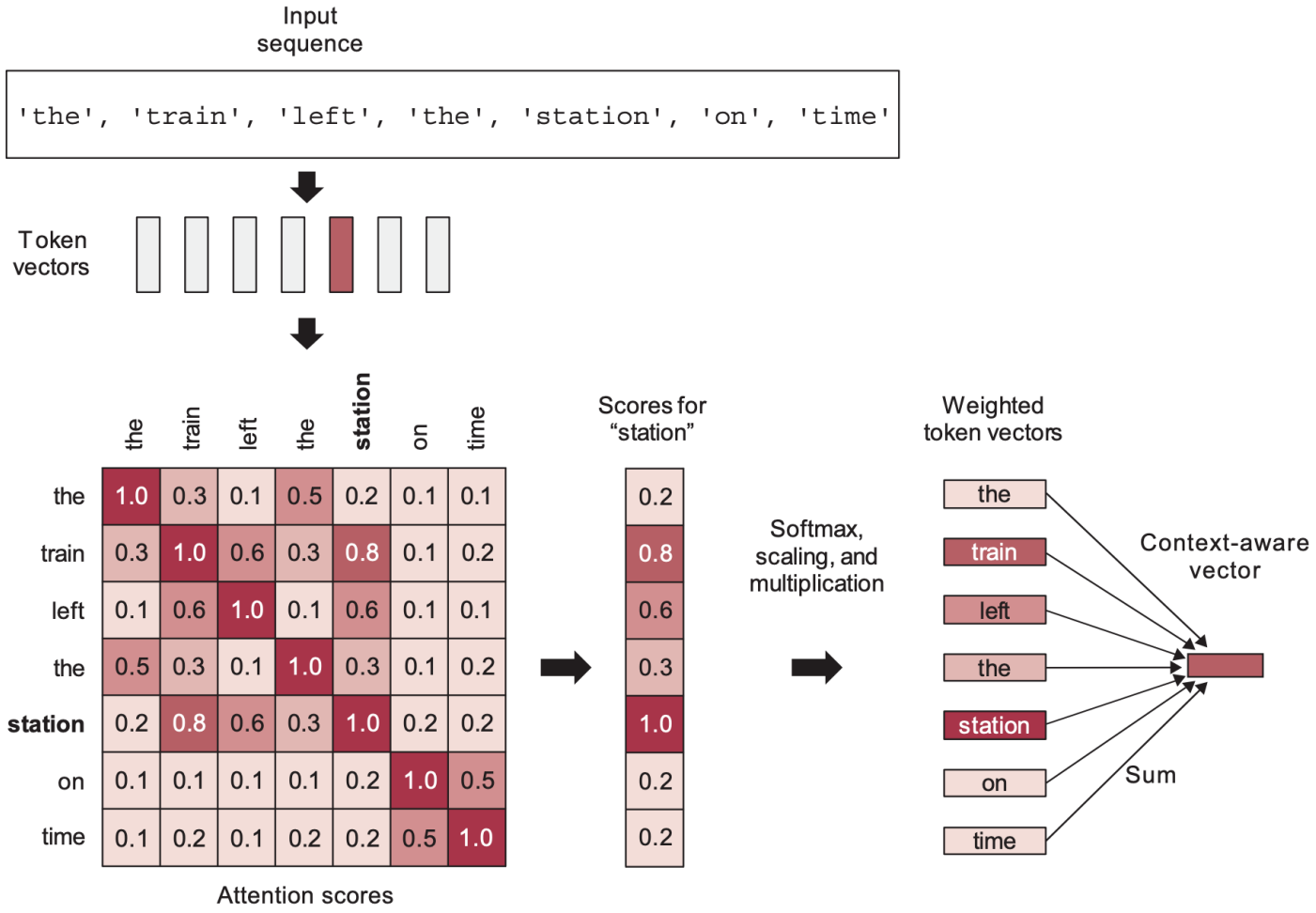
NMT with LSTM Encoder-Decoders

Transformers

- Introduced in 2017 paper "Attention is All You Need"
- Multi-head attention layers extract meaning from text
  - Discern between different meanings of the same word (polysemy)
  - Connect pronouns to subjects
- Train in parallel and connect words independent of distance

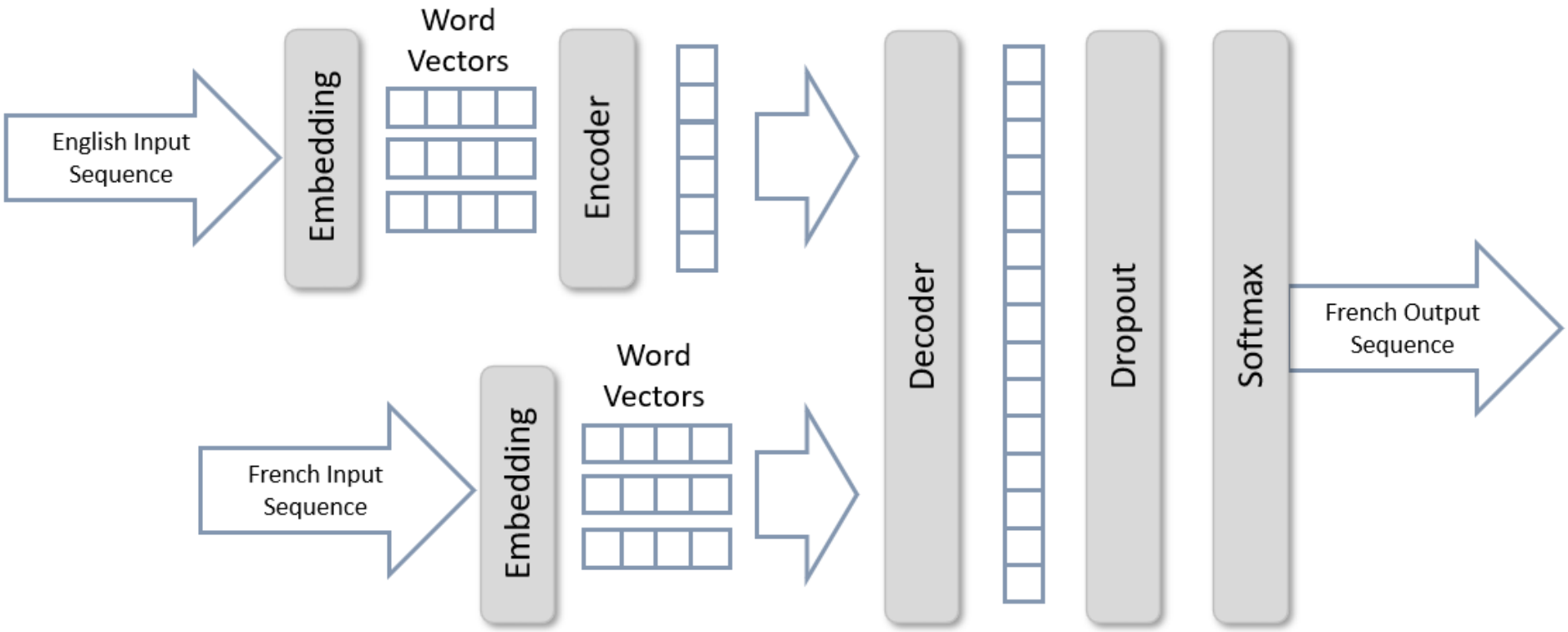


Self-Attention



Source: "Deep Learning with Python" by Francois Chollet (2021, Manning Publications)

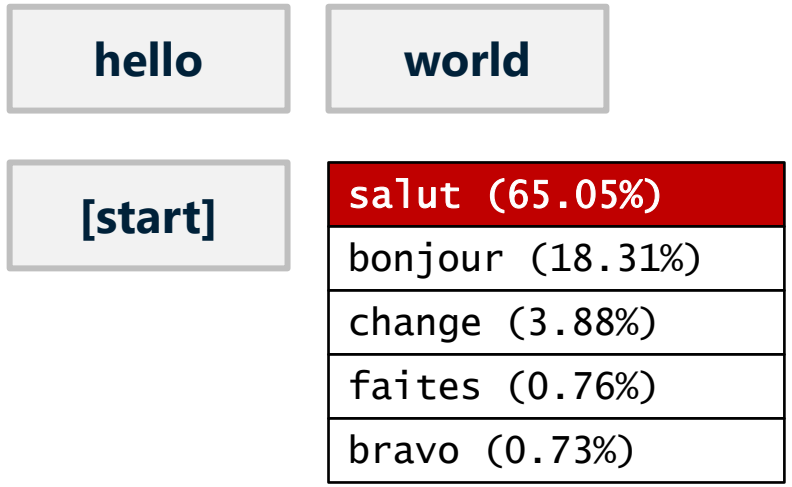
NMT with Transformer Encoder-Decoder



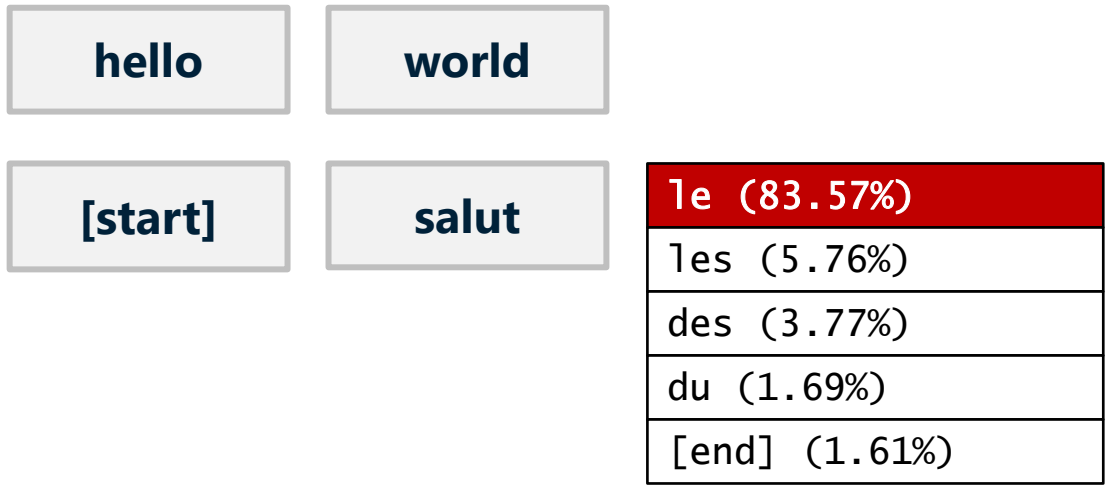
The model has **two inputs**: one for **tokenized English input**, and another for **tokenized French input**. Embedding layers translate each into word vectors.

The decoder translates the inputs into an output sequence, and a softmax output layer predicts the **next word in the French sequence**. The next word is appended to the text predicted thus far and **fed back into the model** to predict the **next word**. The cycle repeats until the **entire English phrase** has been translated.

Translating Text



Translating Text, Cont.

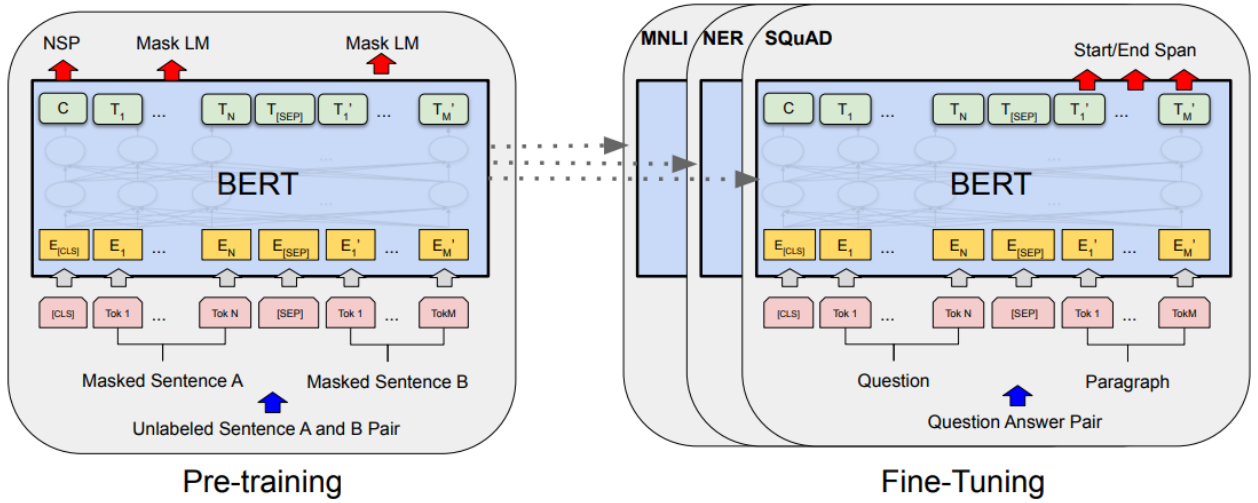


Translating Text, Cont.

Translating Text, Cont.

BERT

- Bidirectional Encoder Representations from Transformers (BERT)
- Built by Google and instilled with language understanding by pretraining with billions of words and phrases
- Can be fine-tuned to perform a variety of NLP tasks



Masked Language Modeling (MLM)

- Turns unlabeled text into training ground for language structure
- During training, a specified percentage (usually 15%) of the tokens are randomly masked (dropped) from the training sequences, and the model is trained to predict the missing words

?	and	seven	years	**?**	our
fathers	**?**	forth	on	this	continent
a	new	**?**	conceived	in	liberty
and	dedicated	**?**	the	proposition	**?**

KerasNLP

- Contains classes for building transformer-based deep-learning models
  - **TokenAndPositionEmbedding** class implements positional embedding layers
  - **TransformerEncoder** class implements transformer encoders that include multi-head attention modules for self-attention
  - **TransformerDecoder** class implements transformer decoders that include multi-head attention modules for self-attention
  - **WordPieceTokenizer** class tokenized input for BERT models
- Free, open-source, and by the same team that brought you Keras

https://keras.io/api/keras_nlp/