ML for SS: Neural Networks and Transfer Learning

Session 11

Dr. Richard M. Crowley
rcrowley@smu.edu.sg
http://rmc.link/

Overview

Papers

- Liu, Lee and Srinivasan (2019)
  - Examines the impact of user reviews on e-commerce purchasing behavior
  - Uses a Convolutional Neural Network to accomplish this
- Huang, Wang, and Yang (2020)
  - Proposes a BERT-based model of sentiment for finance and financial accounting usage
  - Uses transfer learning to accomplish this

Technical Discussion

> Focus on Neural Networks for numeric and text data

Python

- Using Keras with Tensorflow for numeric inputs
- Using premade models for text analytics
- Using premade models for transfer learning

R

- You can use Keras from R through RStudio's package

> Python's support is a lot better here

Frameworks for Neural networks

TensorFlow

- It can run almost ANY ML/AI/NN algorithm
- It has APIs for easier access like Keras
- Comparatively easy GPU setup
- It can deploy anywhere
  - Python & C/C++ built in
  - Swift, R Haskell, and Rust bindings
  - TensorFlow light for mobile deployment
  - TensorFlow.js for web deployment

TensorFlow Lite

TensorFlow.js

magenta

TensorFlow Hub

TensorFlow resources

- It has strong support from Google and others
  - TensorFlow Hub – Premade algorithms for text, image, and video
  - tensorflow/models – Premade code examples
    - The research folder contains an amazing set of resources
  - trax – AI research models from Google Brain

TensorFlow Lite

TensorFlow.js

magenta

TensorFlow Hub

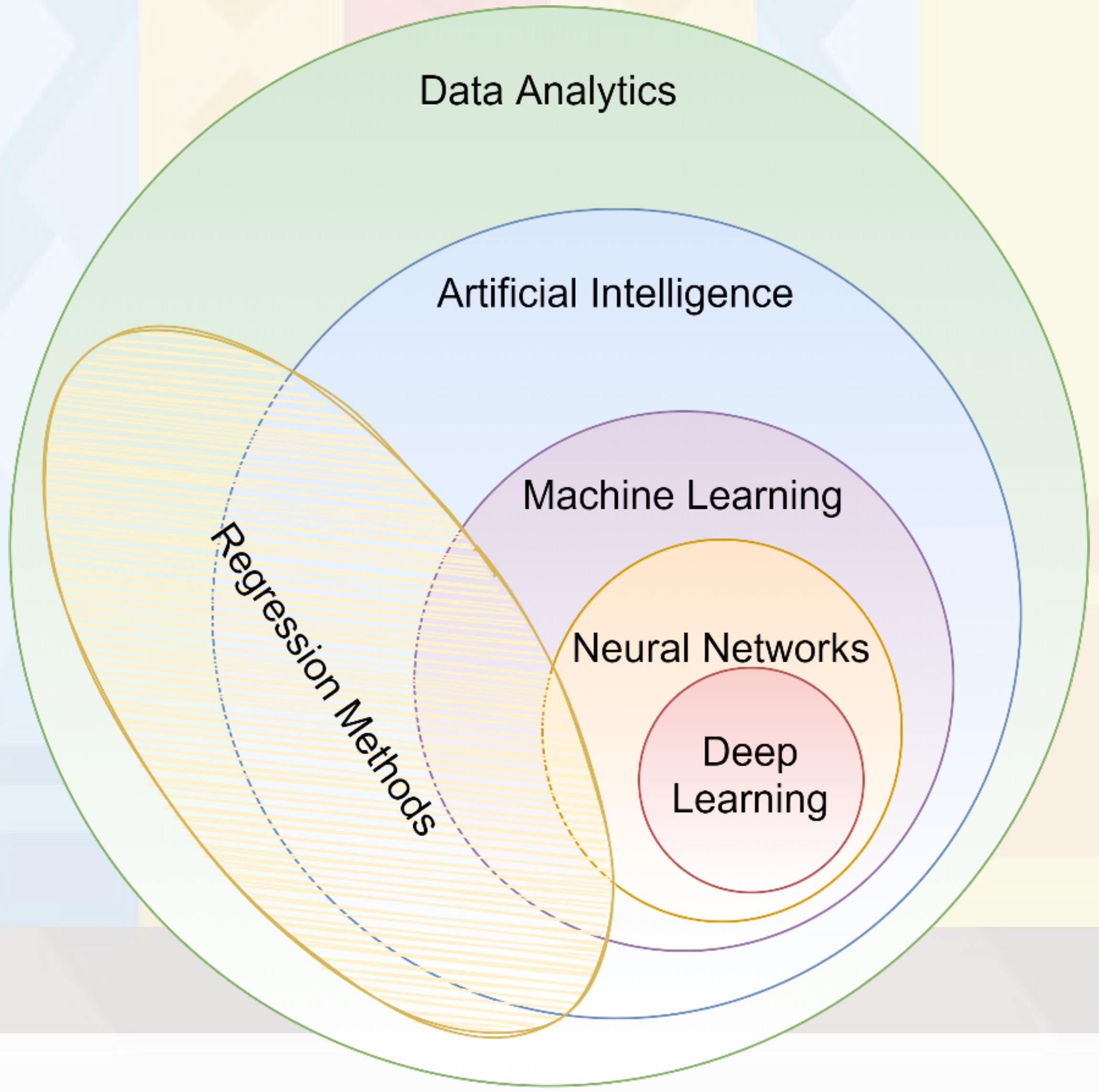
Other notable frameworks

- Caffe
  - Python, C/C++, Matlab
  - Good for image processing
- Caffe2
  - C++ and Python
  - Still largely image oriented
- Microsoft Cognitive Toolkit
  - Python, C++
  - Scales well, good for NLP
- Torch and Pytorch
  - For Lua and python
  - fast.ai, ELF, and AllenNLP
- H20
  - Python based
  - Integration with R, Scala…

Neural Networks

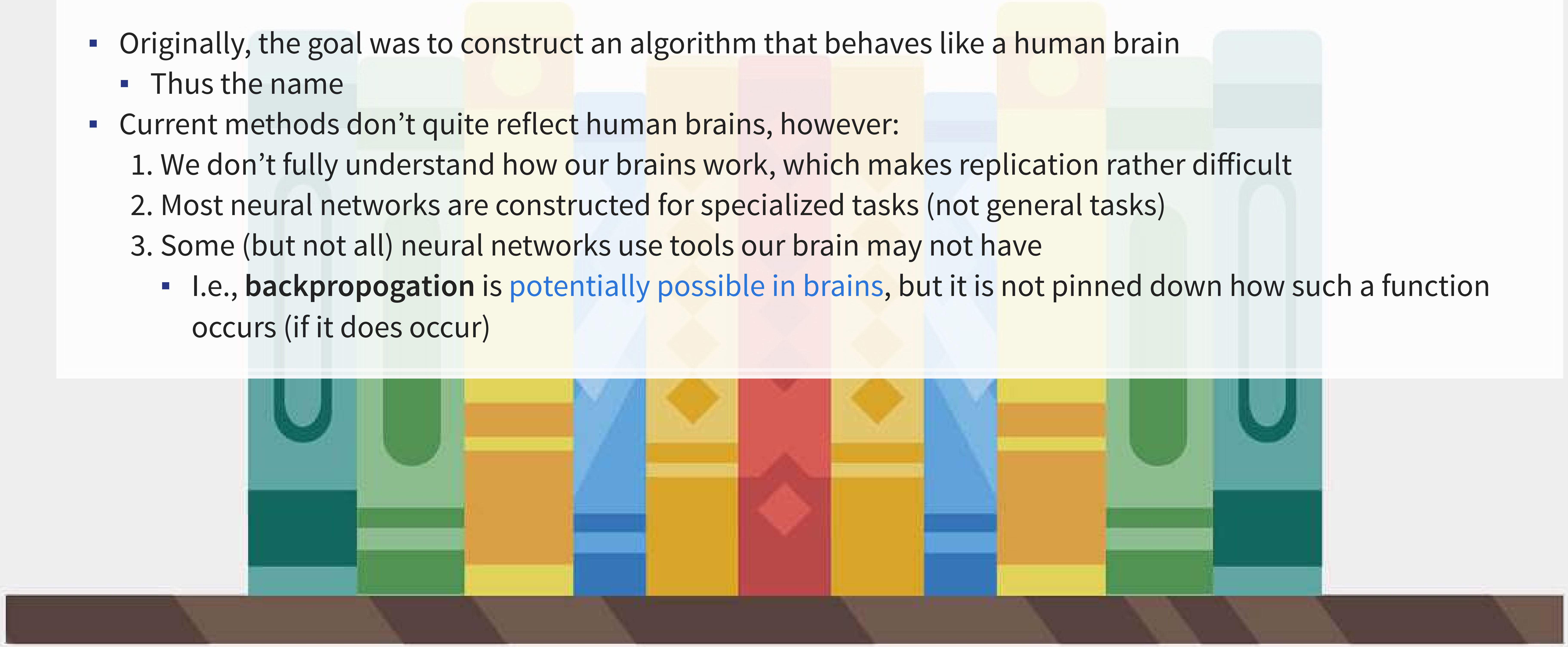
What are neural networks?

- The phrase *neural network* is thrown around almost like a buzz word
- *Neural networks* are actually a specific type class algorithms
  - There are many implementations with different primary uses



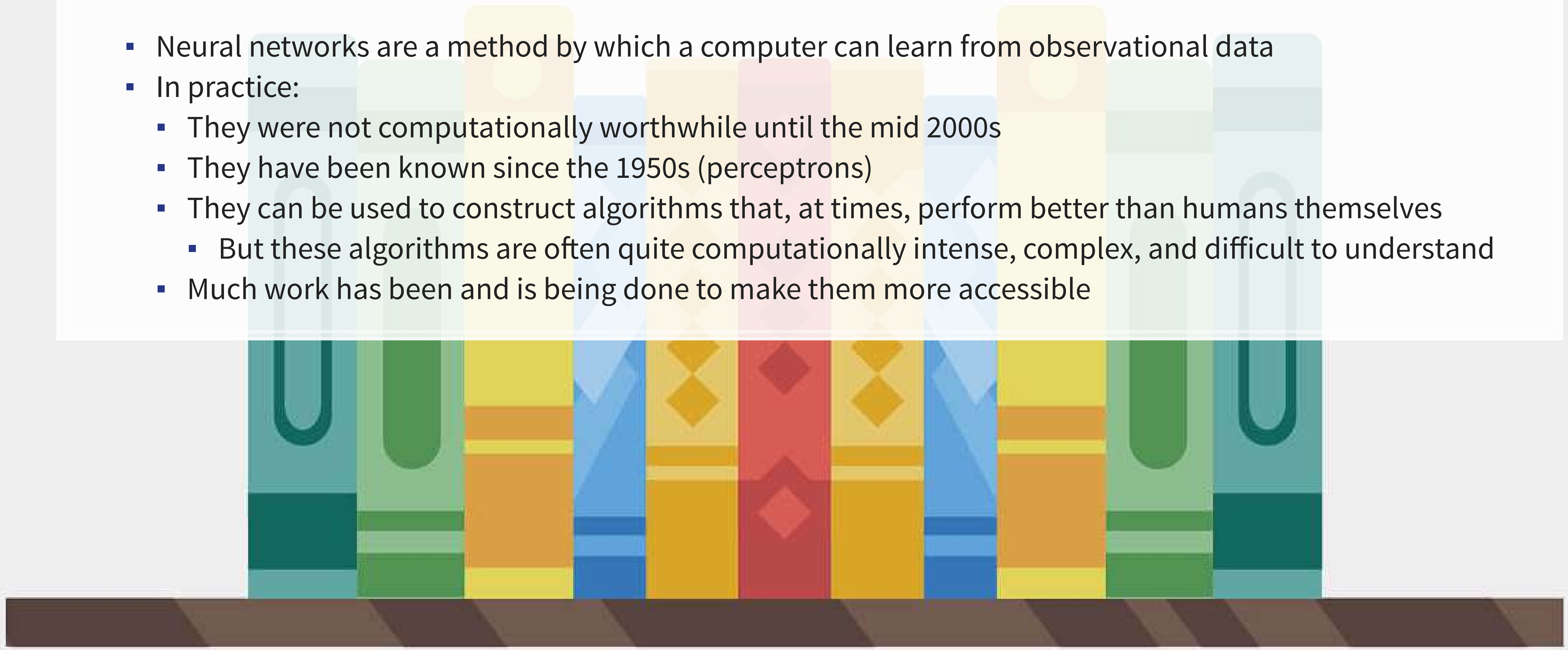
What are neural networks?

- Originally, the goal was to construct an algorithm that behaves like a human brain
  - Thus the name
- Current methods don’t quite reflect human brains, however:
  1. We don’t fully understand how our brains work, which makes replication rather difficult
  2. Most neural networks are constructed for specialized tasks (not general tasks)
  3. Some (but not all) neural networks use tools our brain may not have
     - I.e., **backpropogation** is potentially possible in brains, but it is not pinned down how such a function occurs (if it does occur)



What are neural networks?

- Neural networks are a method by which a computer can learn from observational data
- In practice:
  - They were not computationally worthwhile until the mid 2000s
  - They have been known since the 1950s (perceptrons)
  - They can be used to construct algorithms that, at times, perform better than humans themselves
    - But these algorithms are often quite computationally intense, complex, and difficult to understand
  - Much work has been and is being done to make them more accessible

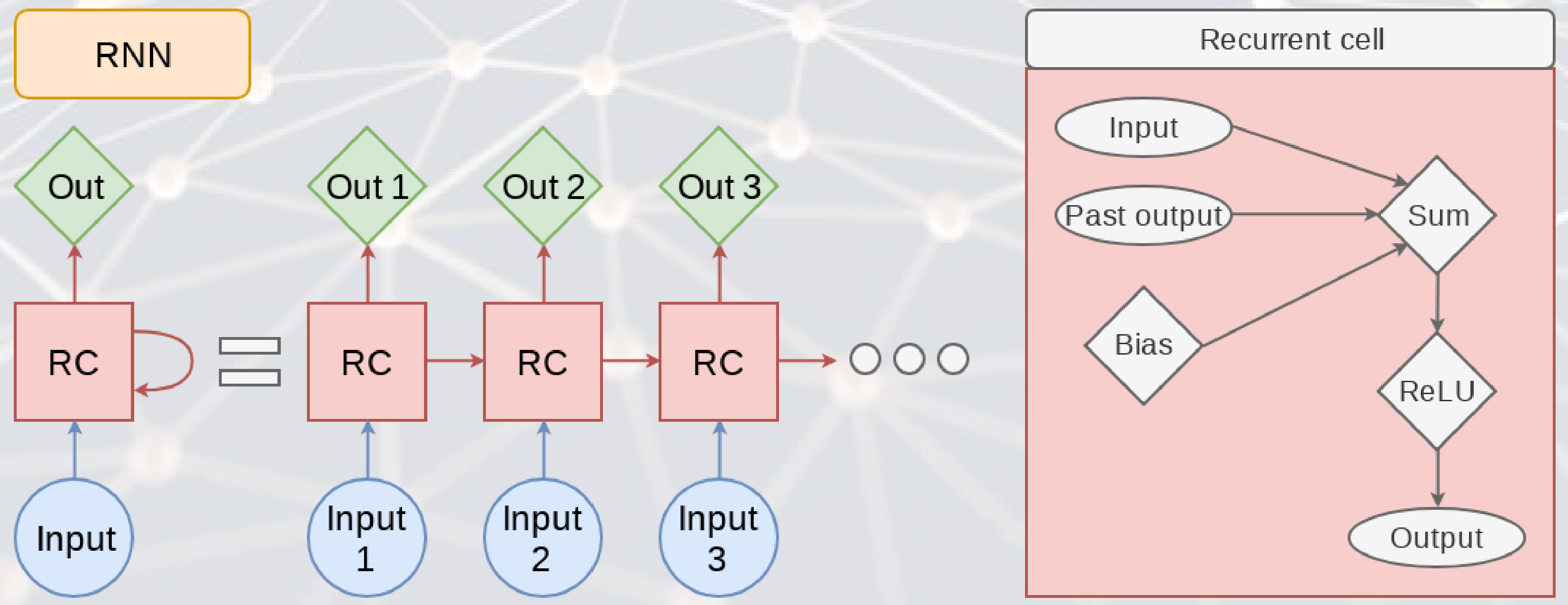


Types of neural networks

- There are *a lot* of neural network types
  - See The “Neural Network Zoo”
- Some of the more interesting ones which we will see or have seen:
  - RNN: Recurrent Neural Network
  - LSTM: Long/Short Term Memory
  - CNN: Convolutional Neural Network
  - DAN: Deep Averaging Network
  - GAN: Generative Adversarial Network
- Others worth noting
  - VAE (Variational Autoencoder): Generating *new* data from datasets
- Not in the Zoo, but of note:
  - Transformer: Networks with “attention”
    - From Attention is All You Need

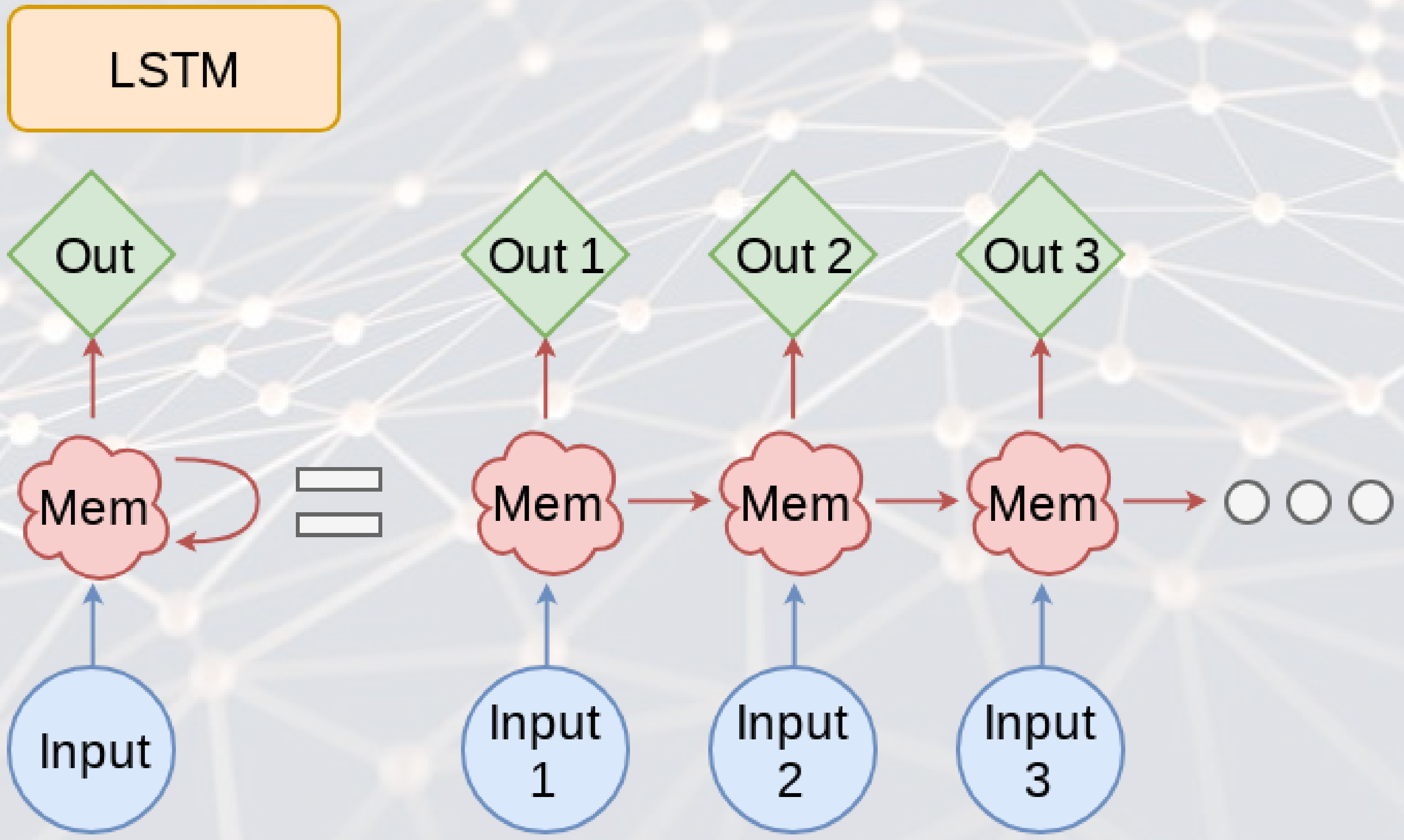
RNN: Recurrent NN

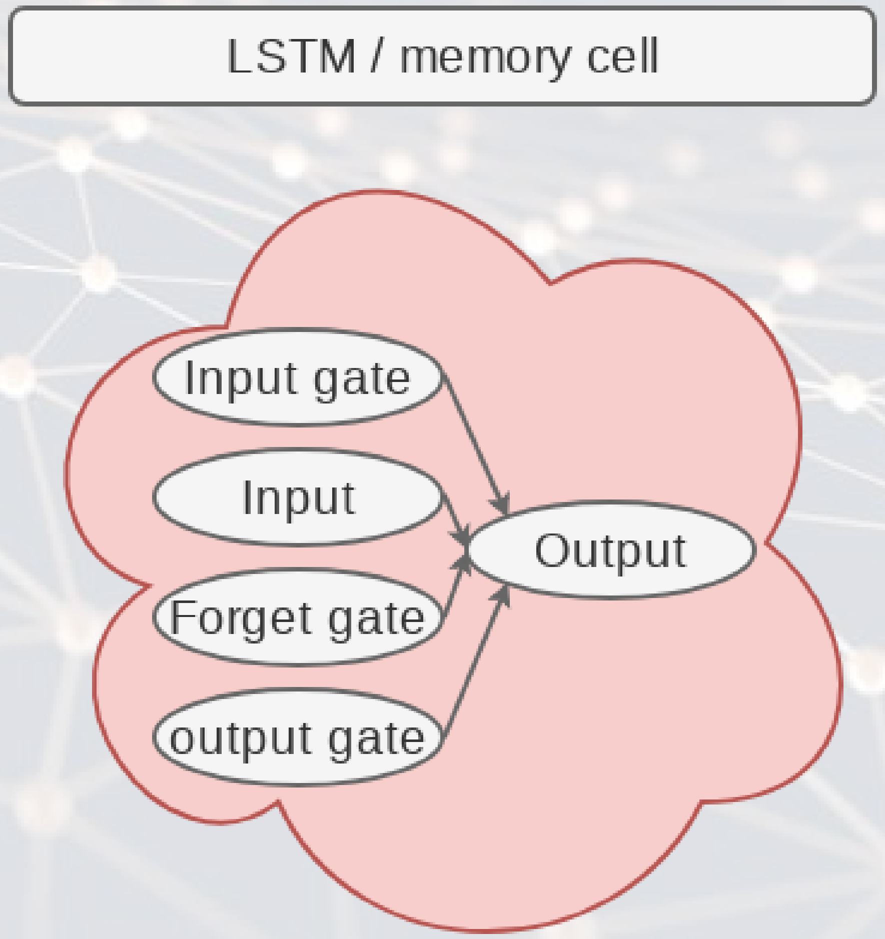
- Recurrent neural networks embed a history of information in the network
  - The previous computation affects the next one
  - Leads to a *short term memory*
- Used for speech recognition, image captioning, anomaly detection, and many others
  - Also the foundation of LSTM
  - SketchRNN (live demo)



LSTM: Long Short Term Memory

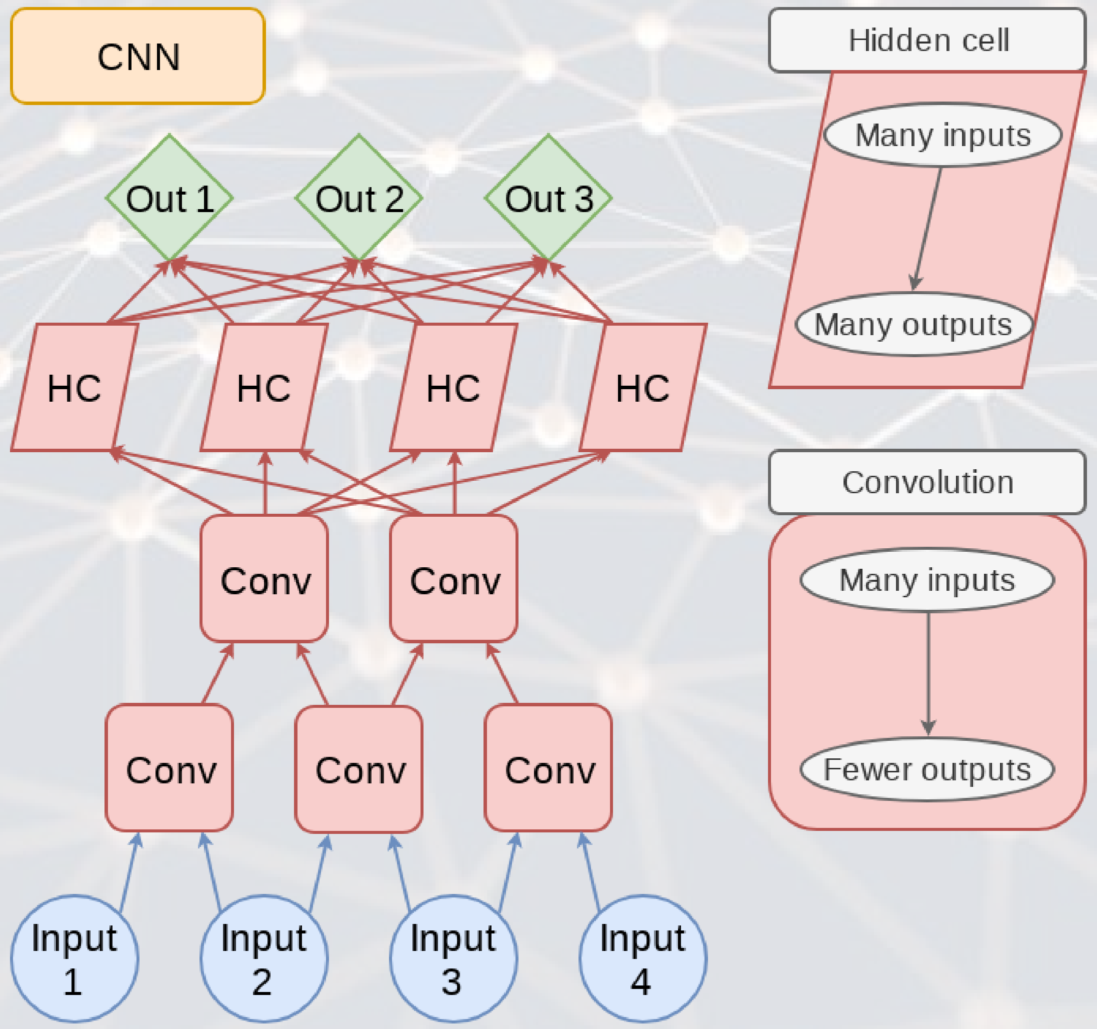
- LSTM improves the *long term memory* of the network while explicitly modeling a *short term memory*
- Used wherever RNNs are used, and then some
  - Ex.: Seq2seq (machine translation)





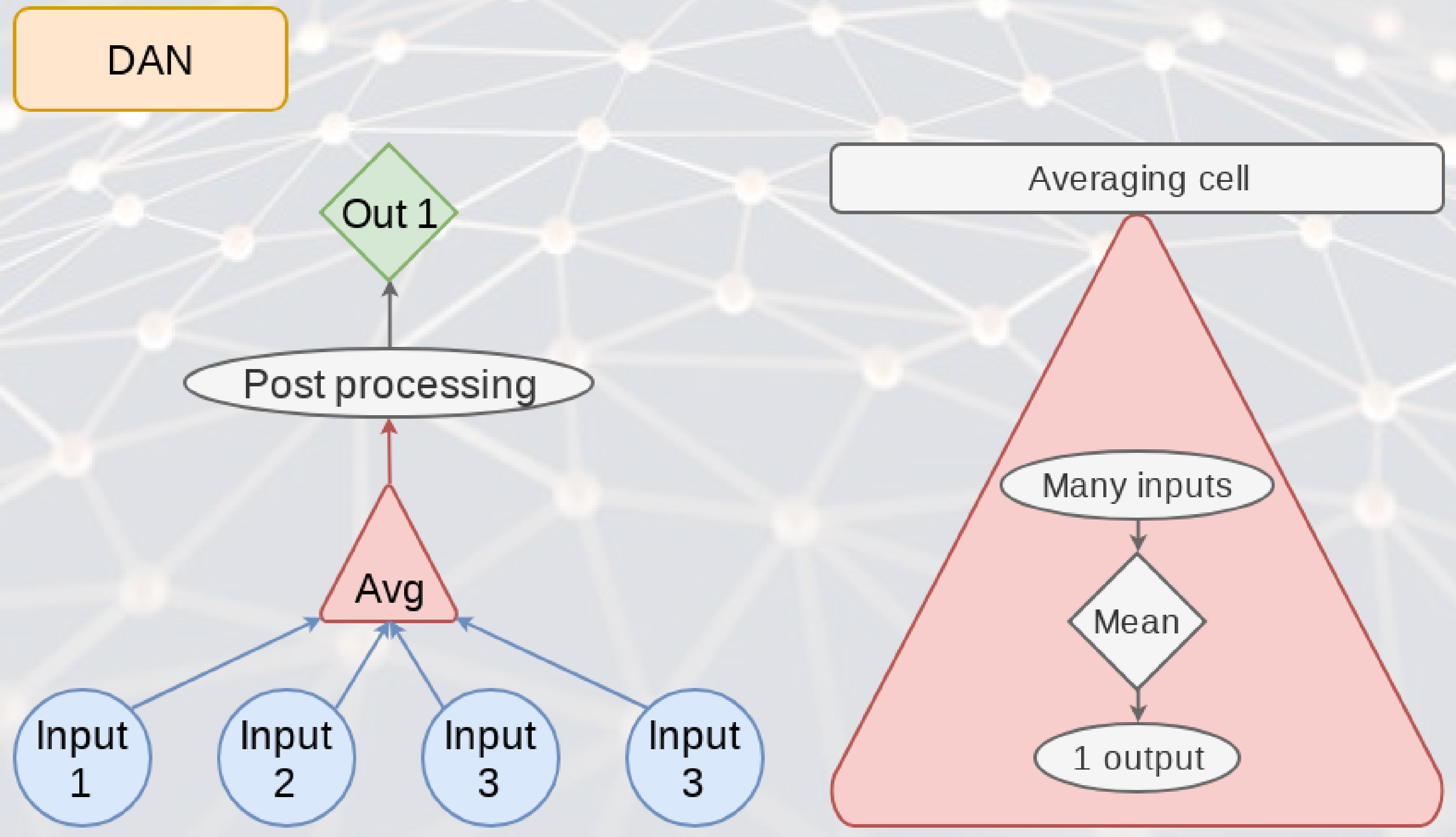
CNN: Convolutional NN

- Networks that excel at object detection (in images)
- Can be applied to other data as well
- Ex.: Inception-v3



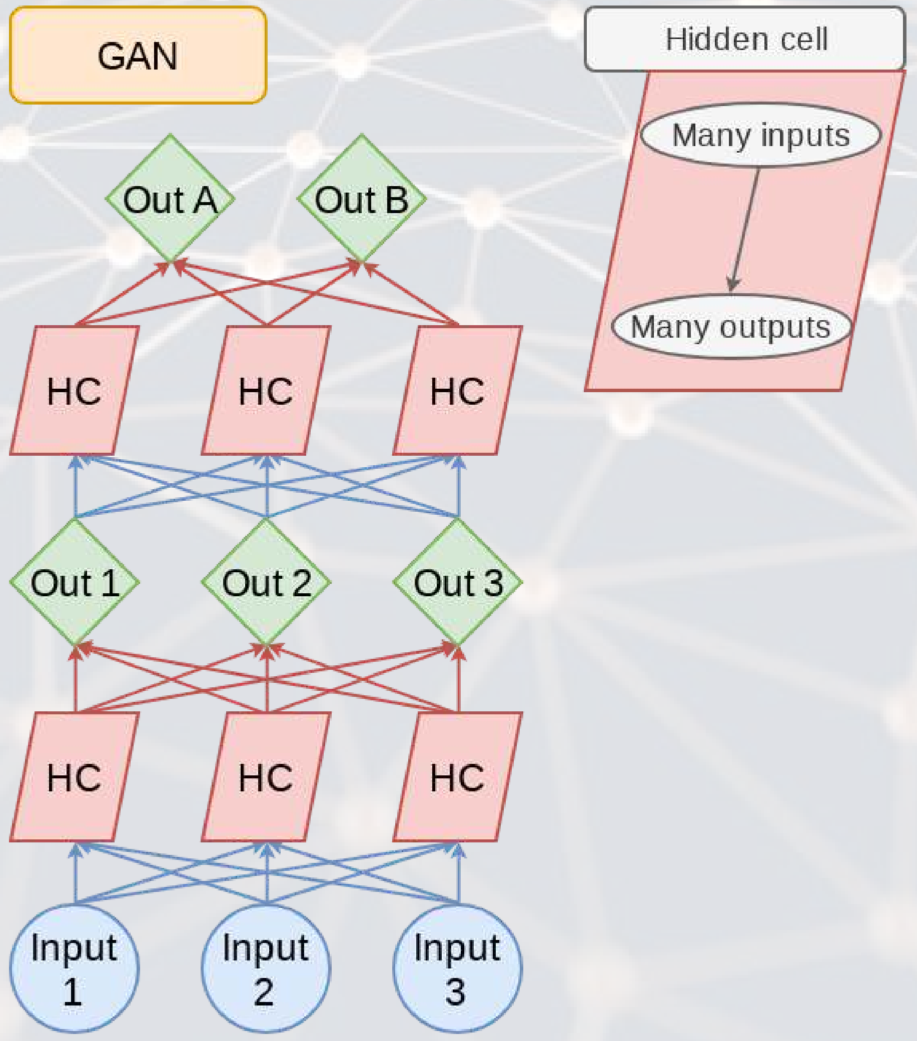
DAN: Deep Averaging Network

- DANs are simple networks that simply average their inputs
- Averaged inputs are then processed a few times
- These networks have found a home in NLP
    - Ex.: Universal Sentence Encoder



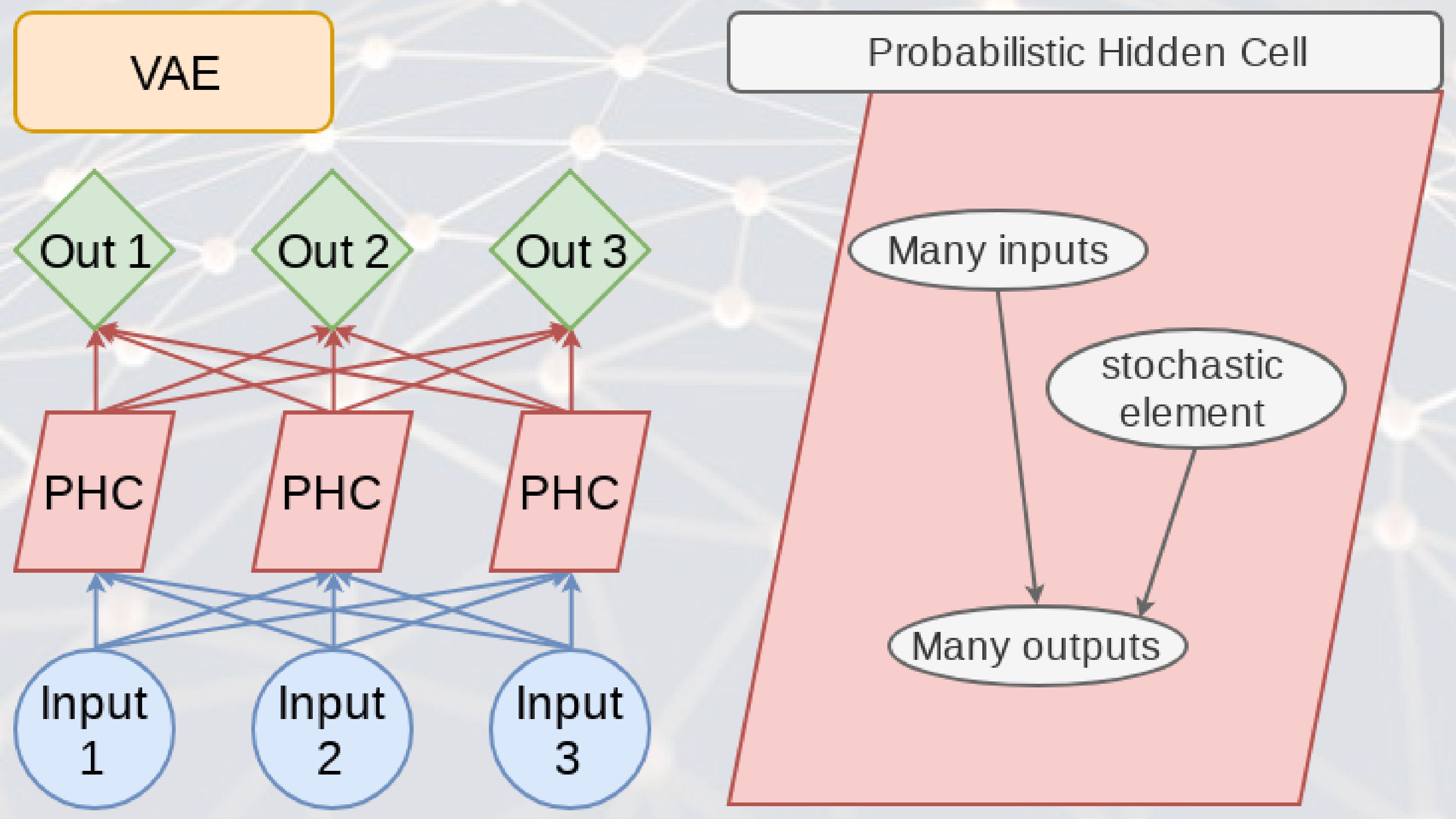
GAN: Generative Adversarial Network

- Feature two networks working against each other
- Many novel uses
  - Ex.: Anonymizing clinical trial data by simulating an attack on the dataset
  - Ex.: Aging images



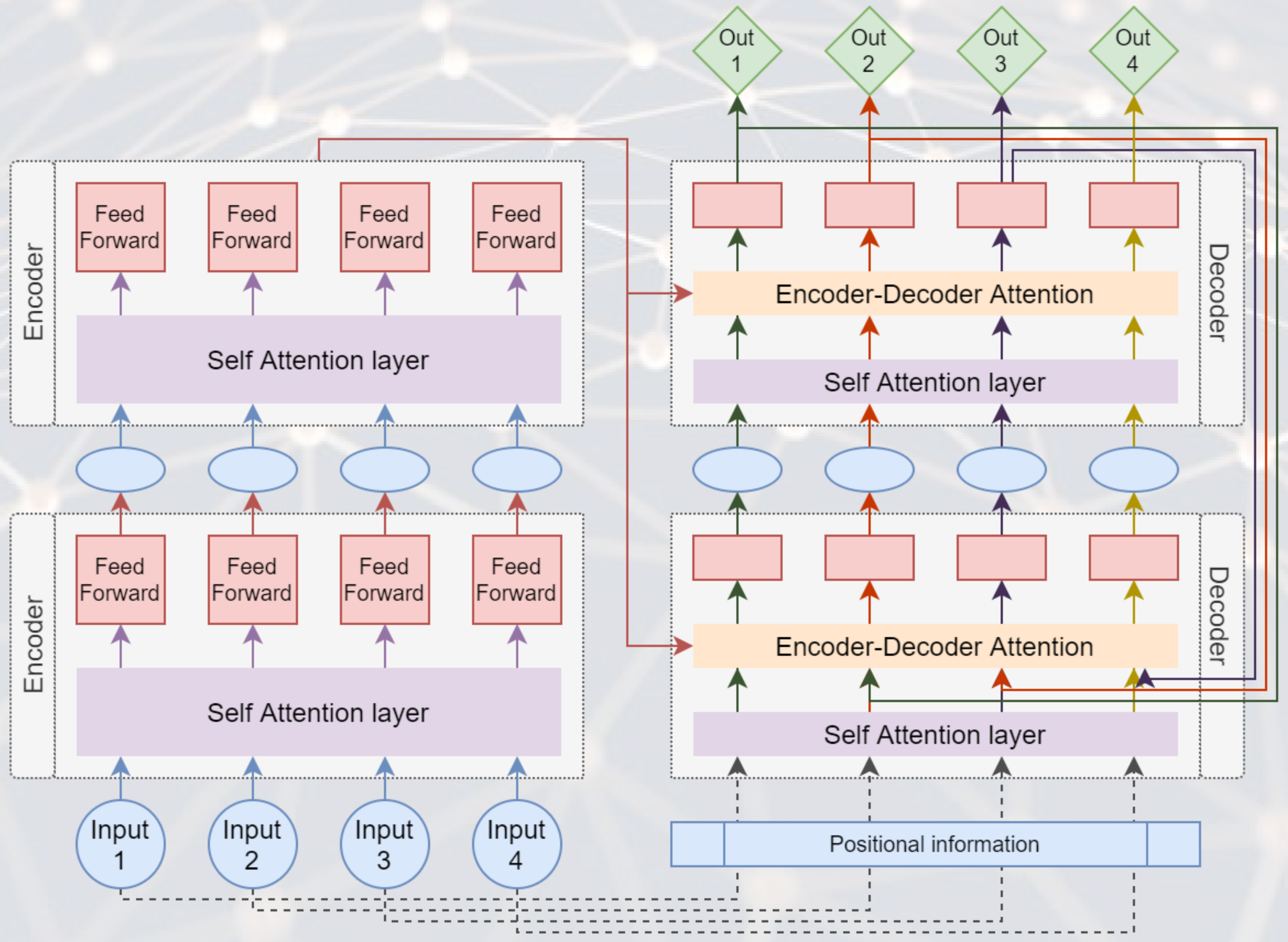
VAE: Variational Autoencoder

- An autoencoder (AE) is an algorithm that can recreate input data
- Variational means this type of AE can vary other aspects to generate completely new output
  - Good for creating fake data
- Like a simpler, noisier GAN



Transformer

- Shares some similarities with RNN and LSTM: Focuses on attention
- Currently being applied to solve many types of problems
- Examples: BERT, GPT-3, XLNEt



A simple example: MNIST

MNIST

- MINST is a set of handwritten numbers with annotations
  - It has prespecified training and testing samples
    - Ensures comparability
    - 60,000 for training, 10,000 for testing
- It’s available in `tensorflow`, so we will import from there

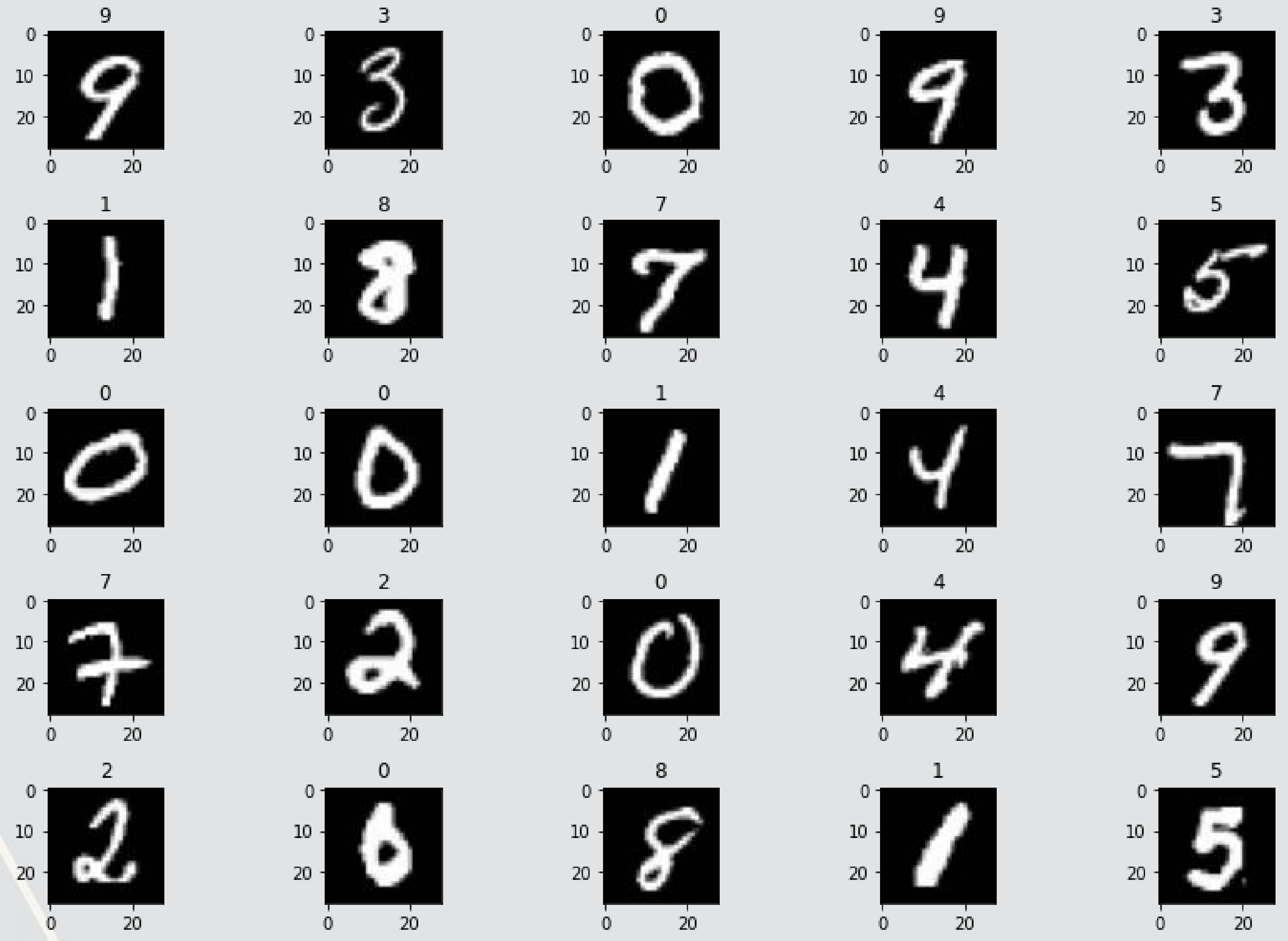
```python
(train_X, train_Y), (test_X, test_Y) = keras.datasets.mnist.load_data()

print('Train, X:%s, Y:%s' % (train_X.shape, train_Y.shape))
print('Test, X:%s, Y:%s' % (test_X.shape, test_Y.shape))
```

```
## Train, X:(60000, 28, 28), Y:(60000,)
## Test, X:(10000, 28, 28), Y:(10000,)
```

A look at the MNIST data

```python
images = np.random.randint(0, train_X.shape[0], size=25)
for i in range(0, 25):
    # define subplot
    image = images[i]
    plt.subplot(5, 5, i+1)
    # plot raw pixel data
    plt.imshow(train_X[image], cmap=plt.get_cmap('gray'))
    plt.title(train_Y[image])
plt.tight_layout()
```



Thinking about images as data

- Images **are** data, but they are very unstructured
  - No instructions to say what is in them
  - No common grammar across images
  - Many, many possible subjects, objects, styles, etc.
- From a computer’s perspective, images are just 3-dimensional matrices
  - Rows (pixels)
  - Columns (pixels)
  - Color channels (usually Red, Green, and Blue)
- We can think of the MNIST data as a set of 28x28x1 3D matrices
  - If we ignore spacial aspects, we can just think of each image as a 784-dim vector

Simple neural network

- We will ignore the 2D nature of the image – instead, we will treat it as a vector of values between 0 and 1
- To do this, we need to…
  1. Scale by 255 (the max value in the data/
  2. Reshape our data into vectors

```python
# Scale data
train_X = train_X.astype("float32") / 255
test_X  = test_X.astype("float32") / 255

# convert to vectors
rows = train_X.shape[0]
dim1 = train_X.shape[1]
dim2 = train_X.shape[2]
train_X = train_X.reshape((rows, dim1 * dim2))
rows = test_X.shape[0]
test_X = test_X.reshape((rows, dim1 * dim2))

print('Train, X:%s, Y:%s' % (train_X.shape, train_Y.shape))
print('Test, X:%s, Y:%s' % (test_X.shape, test_Y.shape))
```

```
## Train, X:(60000, 784), Y:(60000,)
## Test, X:(10000, 784), Y:(10000,)
```

Dealing with categorical DVs

- We need to take special care that the Y values are interpreted as categories
  - Otherwise, the default behavior would be to treat them as a continuous numeric measure
- We can use `keras.utils.to_categorical` to convert our data into the right format

```python
train_Y = keras.utils.to_categorical(train_Y, 10)
test_Y = keras.utils.to_categorical(test_Y, 10)

print('Train, X:%s, Y:%s' % (train_X.shape, train_Y.shape))
print('Test, X:%s, Y:%s' % (test_X.shape, test_Y.shape))
```

```
## Train, X:(60000, 784), Y:(60000, 10)
## Test, X:(10000, 784), Y:(10000, 10)
```

Note that Y is now 10-dimensional – it is one hot encoded now

Constructing a simple neural network

- This model is a very simplistic algorithm
- The data streams in as 784-dim vectors (`InputLayer`)
- The data is compressed by 10 fully-connected neurons all in the same layer (`Dense`)
  - Each neuron will take on one category to try to pick up
- The highest probability neuron will be the category guess (`softmax`)

```python
# Parameters for the model
num_classes = 10
input_shape = (784)

model_dense = keras.Sequential(
    [
        keras.layers.InputLayer(input_shape=input_shape),
        keras.layers.Dense(num_classes, activation="softmax")
    ]
)

model_dense.summary()
```

```
## Model: "sequential_3"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #   
## =================================================================
## dense_4 (Dense)              (None, 10)                7850      
## =================================================================
## Total params: 7,850
## Trainable params: 7,850
## Non-trainable params: 0
## _________________________________________________________________
```

Run the neural network

- There are 2 steps to running a neural network:
  1. Compile the model: We previously described the network shape, but didn’t build the network itself
  2. Fit the model to our data
- The `loss` function tells the model what to optimize in training
  - `categorical_crossentropy` corresponds to multiclass classification accuracy
- The `optimizer` is the function used for training the model – `adam` is a good default
- Metrics are what you want it to track and report back to you
- Within the fit command, note that `epochs` is the number of rounds to train the model
  - Higher is often better, but not always
- The model itself runs quickly

```python
batch_size = 128
epochs = 10

model_dense.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model_dense.fit(train_X, train_Y, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

Model performance

- The model we compiled is 92.45% accurate in-sample, with 93.78% accuracy on validation data
- However, what matters most is the accuracy on the testing data
  - `model.evaluate()` will test this for us

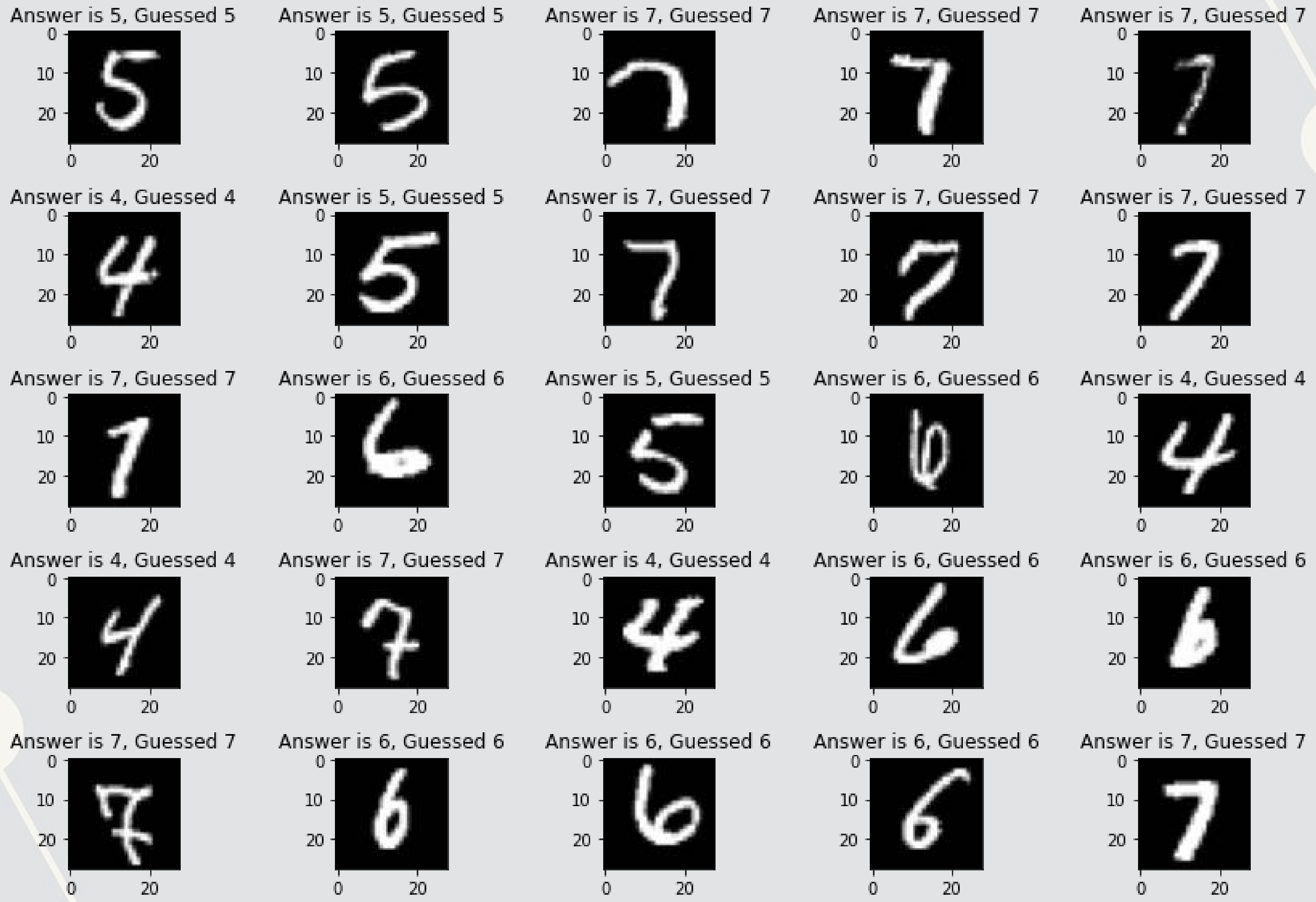
```python
score = model_dense.evaluate(test_X, test_Y, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
## Test loss: 0.267339140176773O7
## Test accuracy: 0.9259999990463257
```

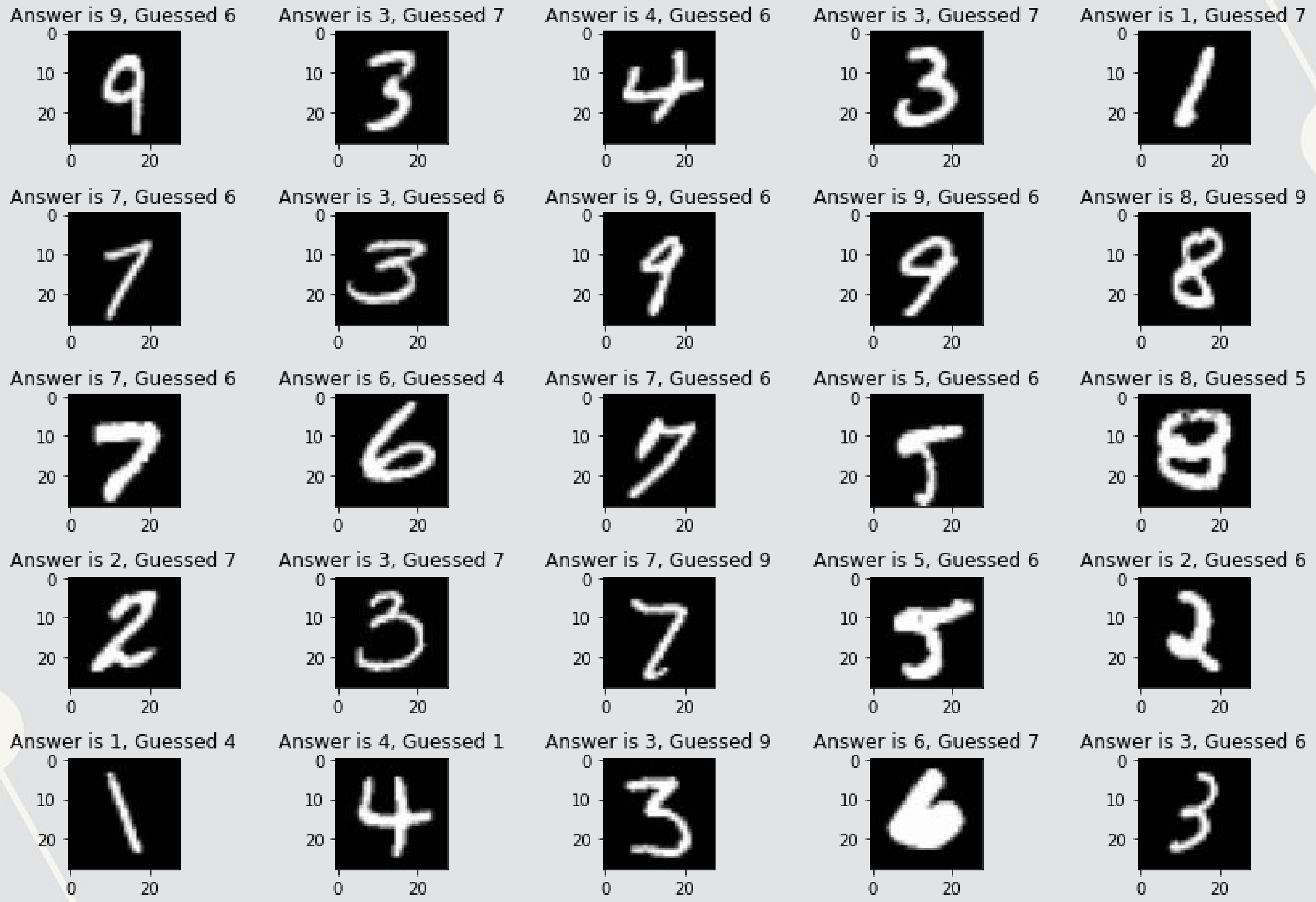
- We will also make lists of what it got right and wrong

```python
correct = np.where(np.argmax(model_dense.predict(test_X), axis=-1) == np.argmax(test_Y, axis=-1))[0]
incorrect = np.where(np.argmax(model_dense.predict(test_X), axis=-1) != np.argmax(test_Y, axis=-1))[0]
```

What does the model get right?



What does the model get wrong?



Addendum: Using R

By R Studio: details here

- There is a port of `keras` for R made by the RStudio team
  - It calls TensorFlow in python, however
- Install with: `devtools::install_github("rstudio/keras")`
- Finish the install in one of two ways:

For those using Conda

- CPU Based, works on *any* computer

```r
library(keras)
install_keras()
```

- Nvidia GPU based
  - Install the Software requirements first

```r
library(keras)
install_keras(tensorflow = "gpu")
```

Using your own python setup

- Follow Google's install instructions for TensorFlow
- Install keras from a terminal with `pip install keras`
- R Studio's keras package will automatically find it
  - May require a reboot to work on Windows

Transfer Learning

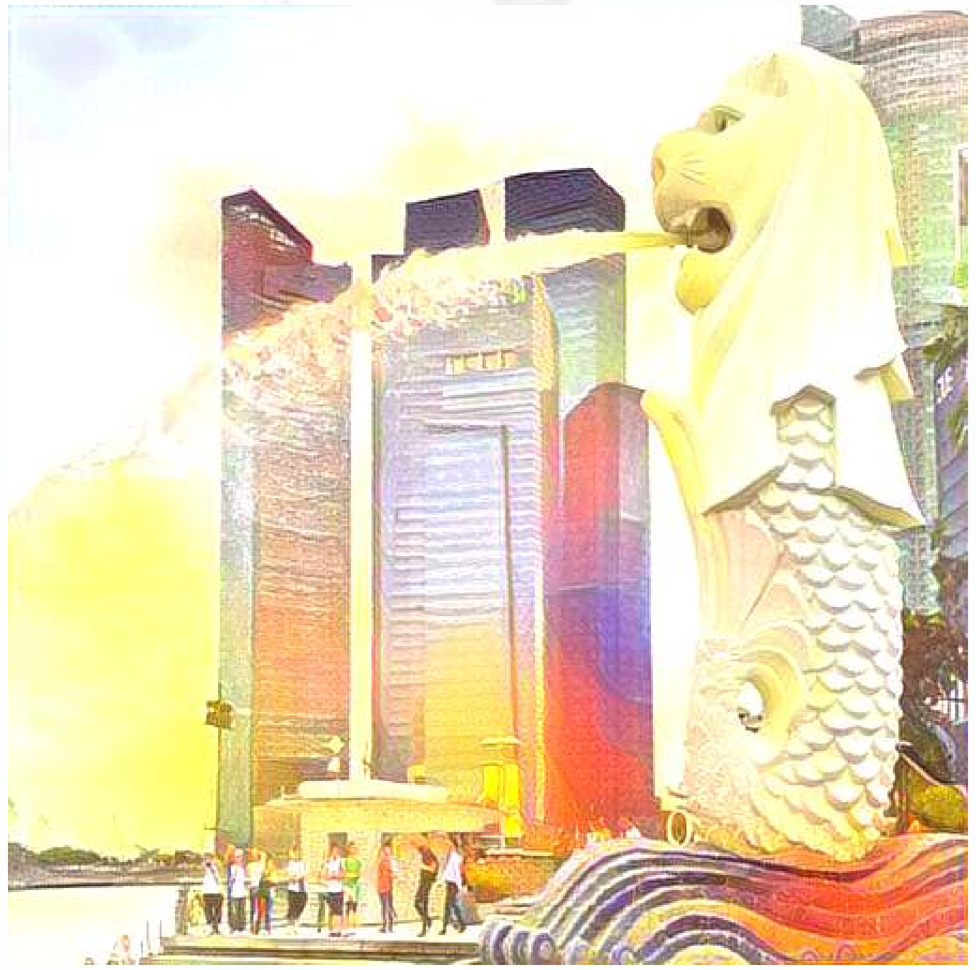
What is transfer learning?

- It is a method of training an algorithm on one domain and then applying the algorithm on another domain
- It is useful when…
  - You don’t have enough data for your primary task
    - And you have enough for a related task
  - You want to augment a model with even more

Try it out!

- Colab file available at this link
  - Largely based off of dsgiitr/Neural-Style-Transfer
  - It just took a few tweaks to get it working in a Google Colaboratory environment properly

Inputs:



How can transfer learning be useful?

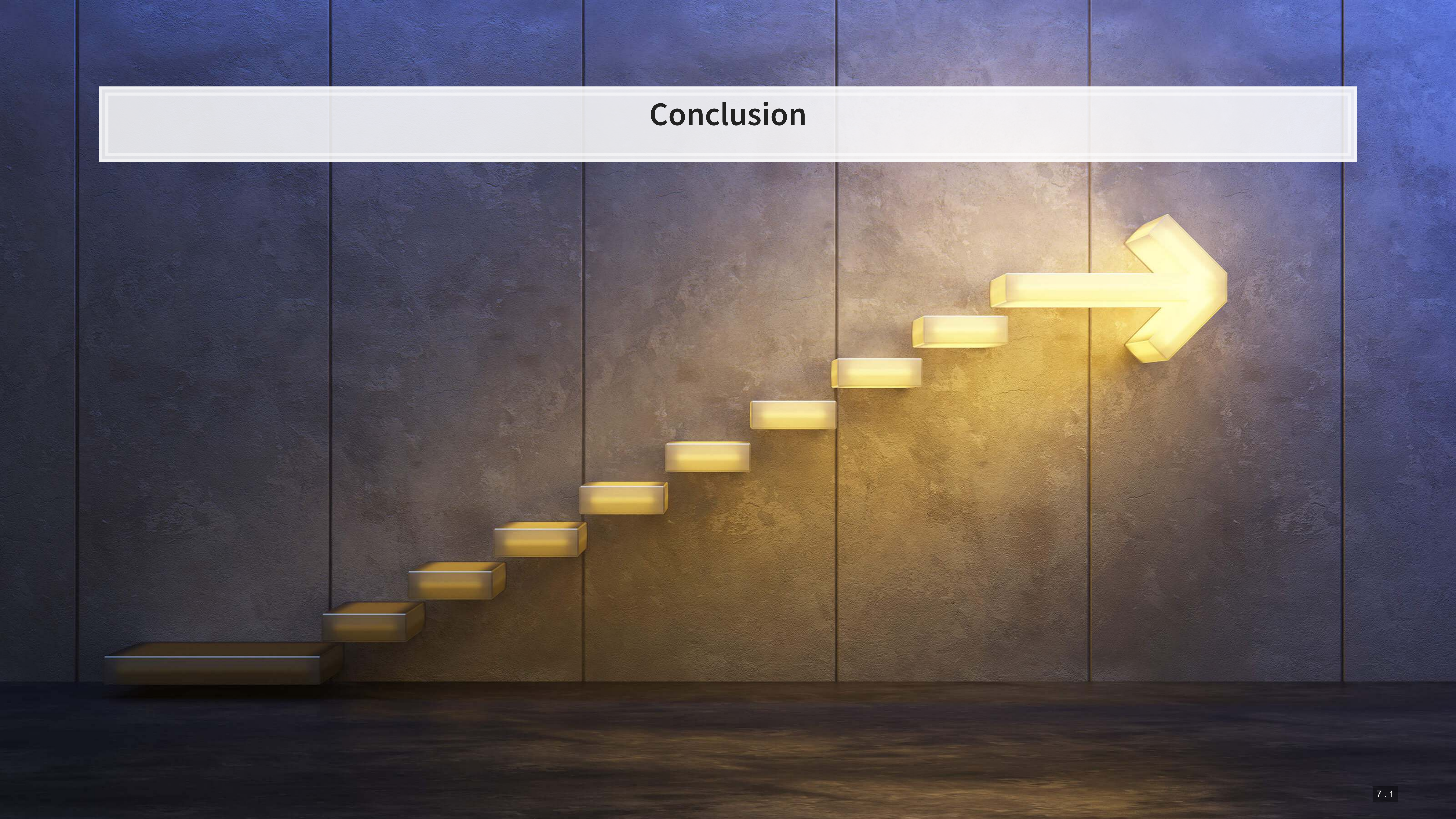
- Simple neural networks are easy enough to train yourself using standard hardware
- Some more complex models, such as BERT models, are too complex to train fully on consumer hardware
  - E.g., the FinBERT paper is trained on an DGX-1 (128 GB of Video RAM)

> As researchers, most of us won't have sufficient hardware to train these models

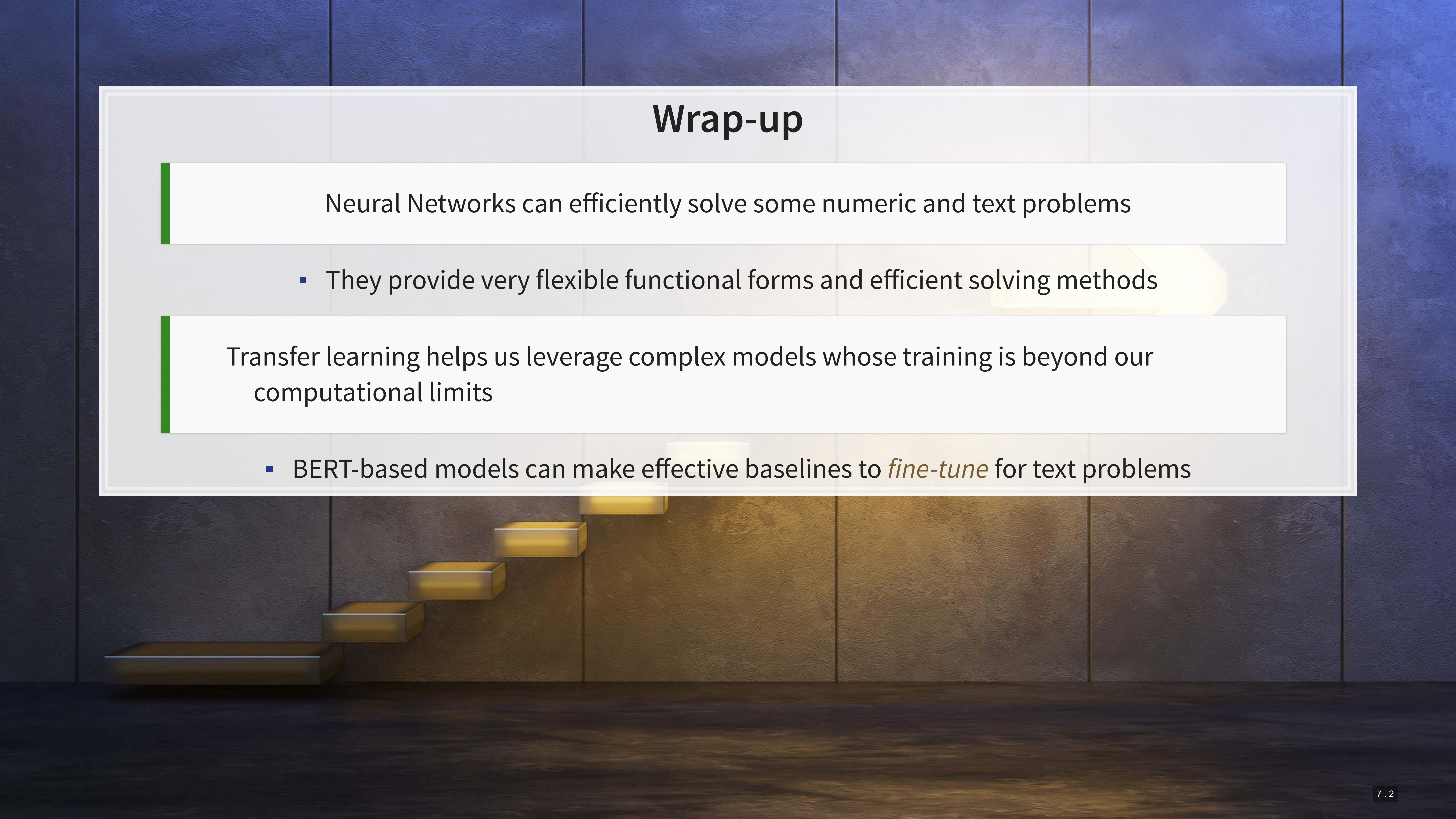
- Transfer learning represents an efficient middle ground
  - Take a pretrained model and retrain it for a specific domain
    - Neural networks often need much less data to retrain than to train initially
    - This retraining is called *fine-tuning*

Using FinBERT

- FinBERT is available in three forms:
  1. A pretrained `pytorch` model
  2. A pretrained huggingface model via the `transformers` package
  3. A fine-tunable `pytorch` model
- Demos of each are available via my colab shares:
  - pretrained pytorch model
  - pretrained huggingface model
  - Fine-tuning pytorch model
    - Note: Colab lacks the computational power needed for this task



Conclusion



Wrap-up

> Neural Networks can efficiently solve some numeric and text problems

- They provide very flexible functional forms and efficient solving methods

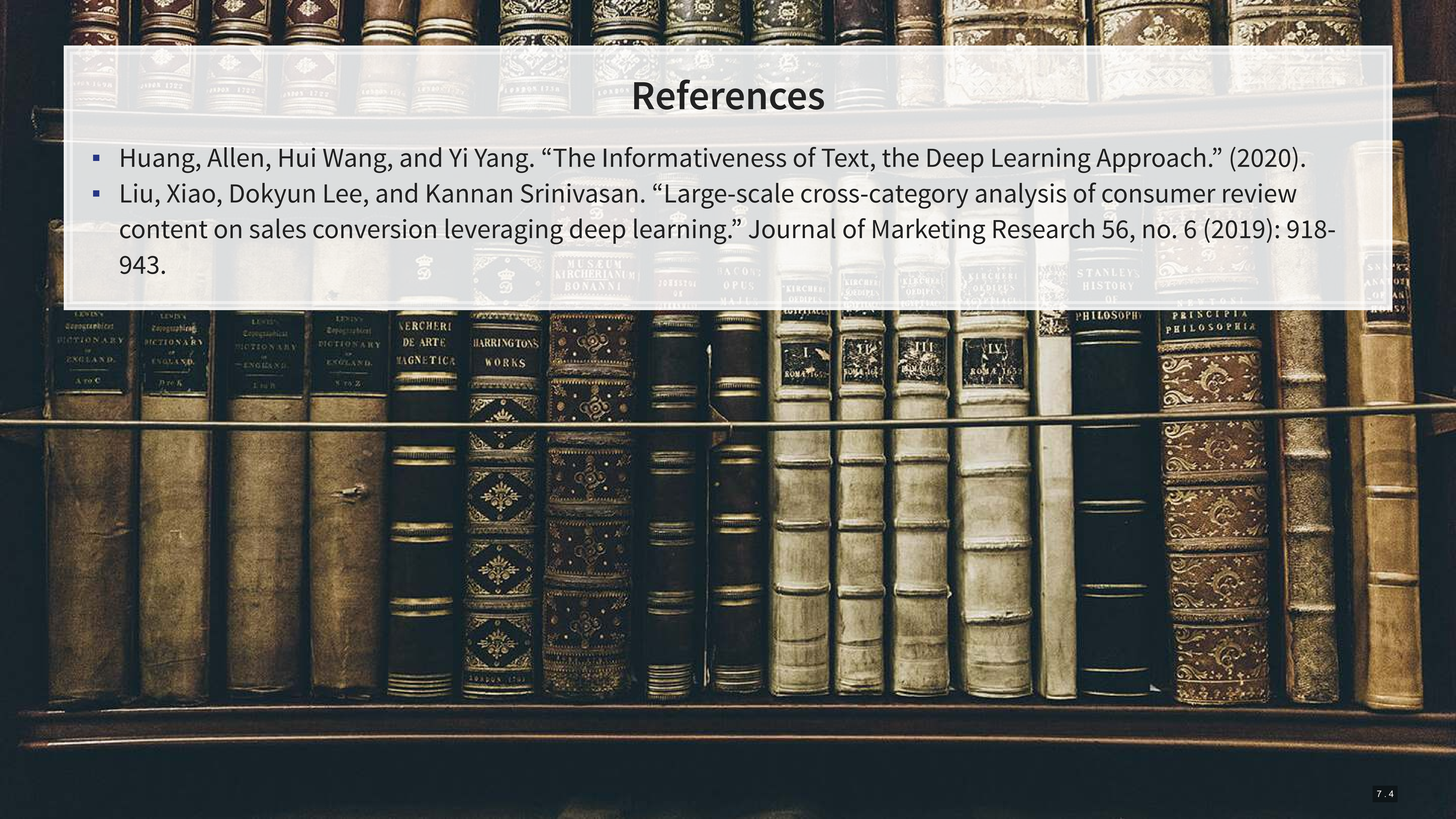
> Transfer learning helps us leverage complex models whose training is beyond our computational limits

- BERT-based models can make effective baselines to *fine-tune* for text problems

Packages used for these slides

Python

- numpy
- pandas
- PIL
- pytorch_pretrained_bert
- torch
- tensorflow
- transformers



References

- Huang, Allen, Hui Wang, and Yi Yang. “The Informativeness of Text, the Deep Learning Approach.” (2020).
- Liu, Xiao, Dokyun Lee, and Kannan Srinivasan. “Large-scale cross-category analysis of consumer review content on sales conversion leveraging deep learning.” Journal of Marketing Research 56, no. 6 (2019): 918-943.