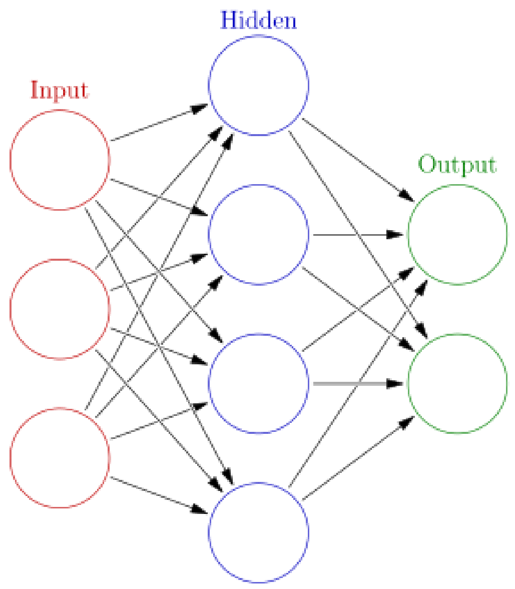
CSCI 303

Introduction to Data Science

Neural Networks



This Lecture

- **History**
- **Applications (what are NNs good for?)**
- **Network architectures**
- **Network construction and training with TensorFlow**

Supplementary material: 3Blue1Brown videos

These are some very well produced videos that explain some of the key NN concepts. **Highly recommended!**

- What is a Neural Network? (https://www.youtube.com/watch?v=aircAruvnKk)
- Gradient descent (https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3&t=0s)
- Backpropagation (https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4&t=0s)
- Backpropagation calculus (https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=5&t=0s)

Biological Inspiration

The mammalian brain is a powerful learning machine!

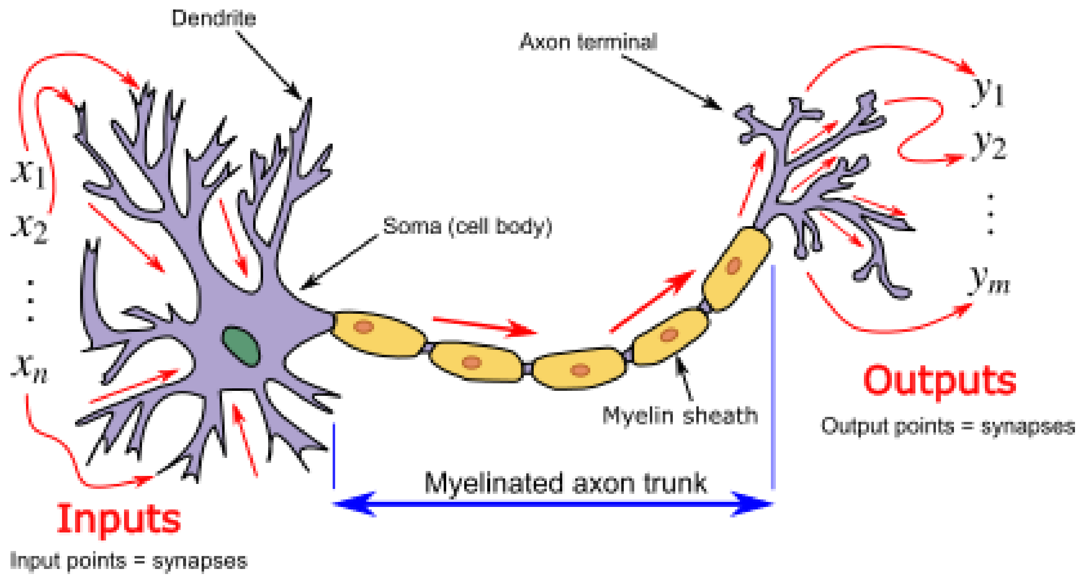
Can we mimic the components and learning algorithms?

- **Components**: Somewhat, yes. The **neurons**.
- **Learning algorithms**: To date, not so much (for **practical** machine learning algorithms).

The biological neuron:

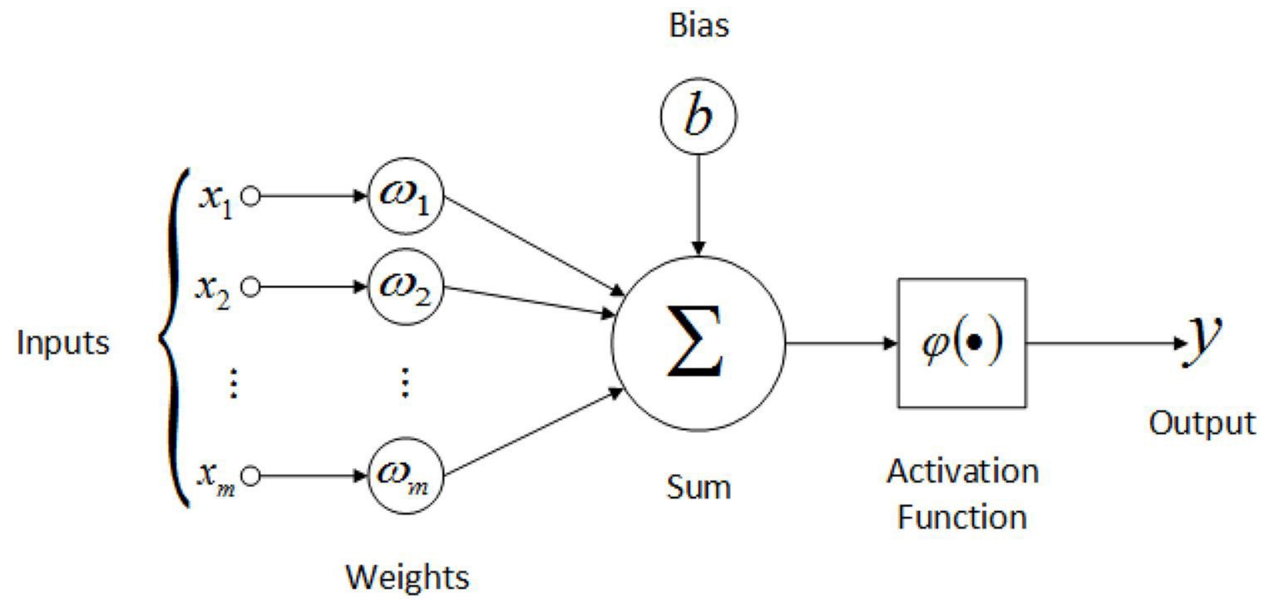
A neuron takes input from many neurons (thousands) and sends output to many neurons (thousands) via connections called **synapses**.

- In biological brains, there are
    - approximately **86 billion neurons**.
    - more than **100 trillion synapses**.
    - many *types* of neurons.
- Biological **learning**
    - **New synapses** are created (especially during first few years of life)
    - **Existing synapses** are "strengthened" or "weakened"
- Biological **signals**
    - **Spikes** are bursts of electrical energy that propagate from one neuron to the next.
    - **Excitatory neurons** increase spiking of neurons to which they send input.
        - More excitatory input spikes typically leads to more output spikes, but with a **non-linear relationship**.
            - Neuron won't create any output spikes if there aren't enough input spikes.
            - Output spike rate can't go above physiological limit.
    - **Inhibitory neurons** decrease spiking of neurons to which they send input.



The Artifical Neuron

An artificial neuron *weights* its inputs, adds a *bias* term, and then applies a non-linearity (*activation* function).



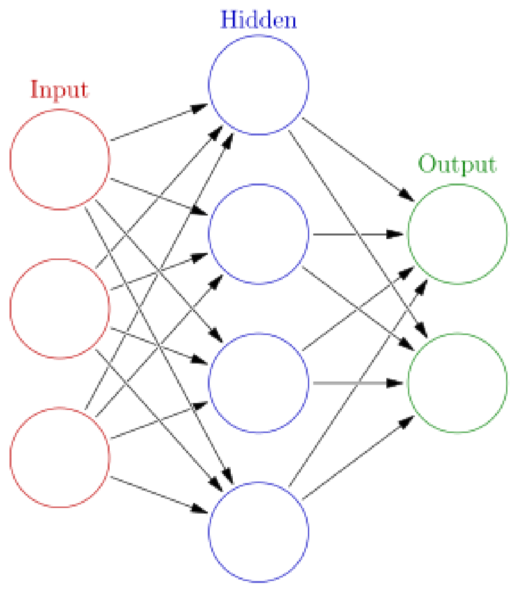
Classic neural networks (circa 1980s)

Composed of input, output, and "hidden" layers

A universal function approximator (https://en.wikipedia.org/wiki/Universal_approximation_theorem) that can model any continuous-valued function with asymptotic precision, if there are enough neurons.

Almost always trained by gradient descent (supervised learning), using a specialized implementation called **backpropagation**, which allows for efficient training of **layered** networks like the one in this figure.

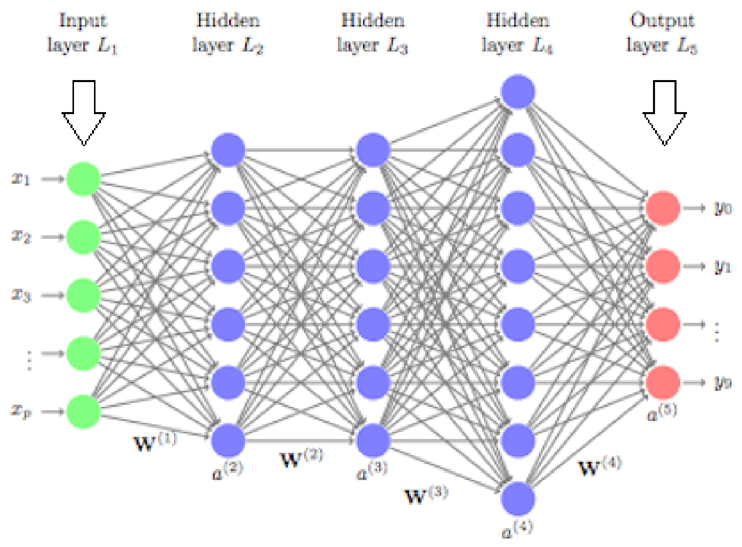
The "input **layer**" is a poor name. It is not composed of a layer of artificial neurons. It is just the sample input values that are sent to the hidden layer. Hidden and output layers are composed of neurons.



Some aspects of these neural networks originated as far back as the 1940s, but it wasn't until the 1980s that all the pieces came to together and the networks were well understood.

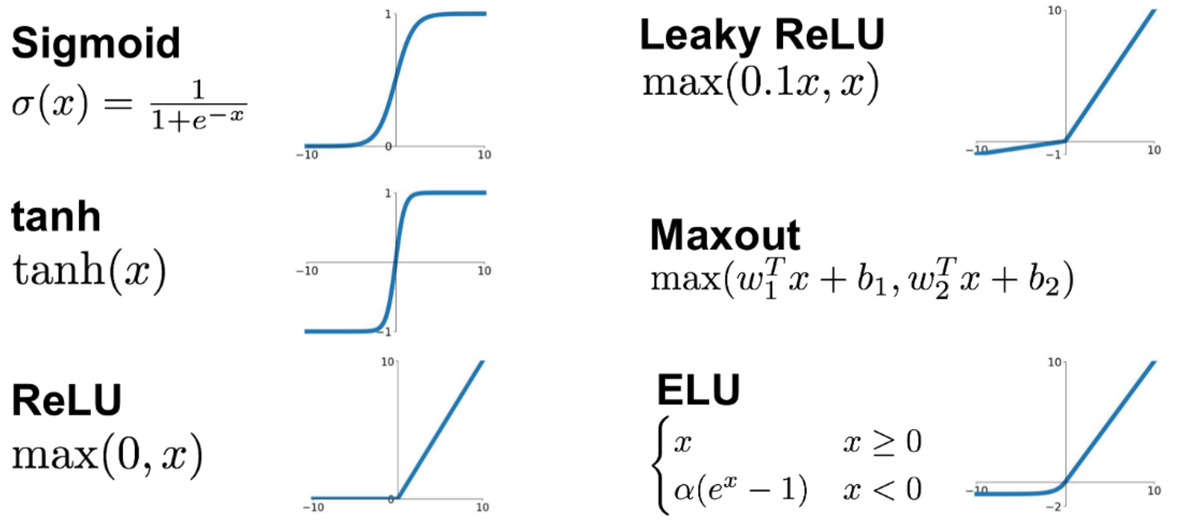
Deep neural networks (modern era)

- Composed of many **more layers** (100s sometimes!), neurons, and synapses (weights) than classic networks
- Use **modern activation functions**, especially rectified linear units (ReLU), rather than sigmoidal
- Training uses "tricks" to **accelerate training speed** (gradient descent convergence)
- More layers and neurons means many **more parameters** and thus:
  - Need for much **larger data sets** for training (typically)
  - Need for lots of **hardware resources** (memory, processing speed, etc.)
- Variety of network architectures
  - Layers of **"fully-connected" neurons**, as in figure below
  - **Spatially or temporally structured** (further ahead in this lecture)



Activation functions

- **Sigmoidal** is simple model of activation function of biological neurons
- **Rectified Linear (ReLU)** is most common activiation function for artificial neurons



What tasks are deep NNs good for? Tasks that use *unstructured data*!

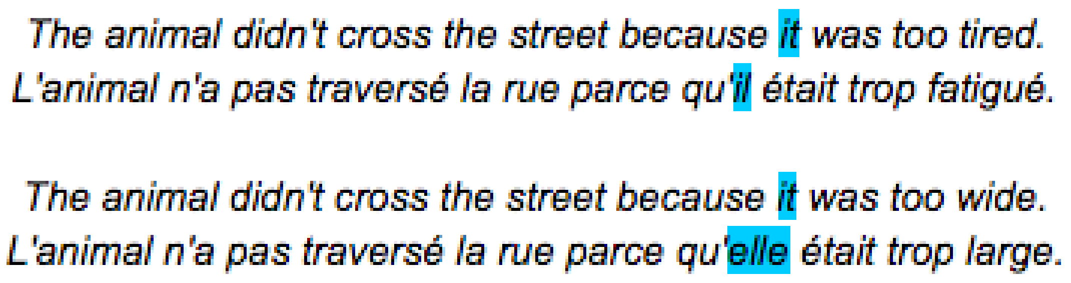
Models applied to such tasks are often dubbed **Artificial Intelligence (AI)** rather than Machine Learning or Data Science

Object (image) recognition, on photographic images

Natural Language Processing

Speech recognition

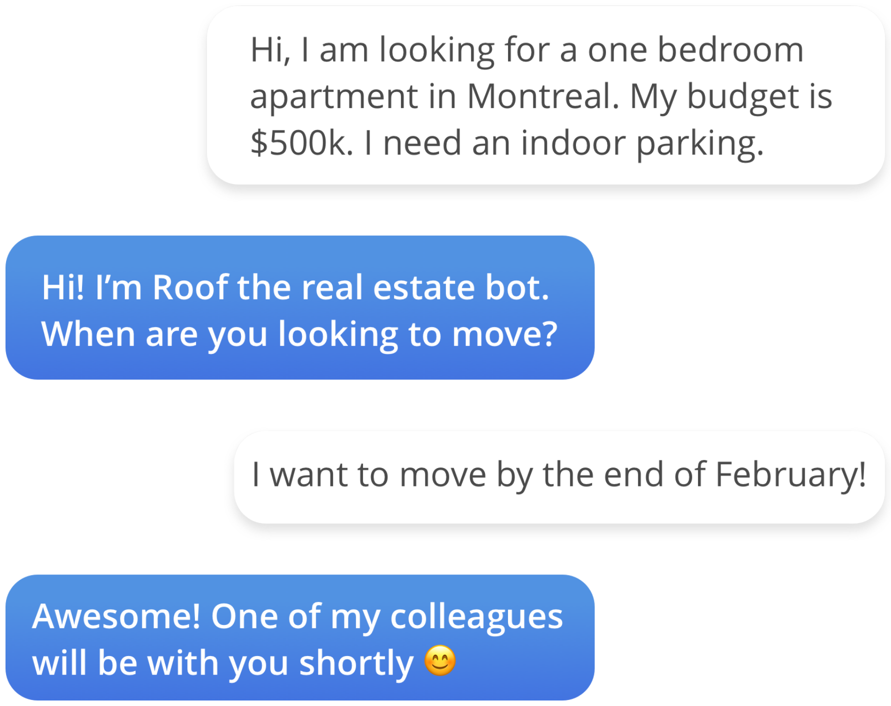
Text language translation



The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

Chatbots


Hi, I am looking for a one bedroom apartment in Montreal. My budget is $500k. I need an indoor parking.

Hi! I’m Roof the real estate bot. When are you looking to move?

I want to move by the end of February!

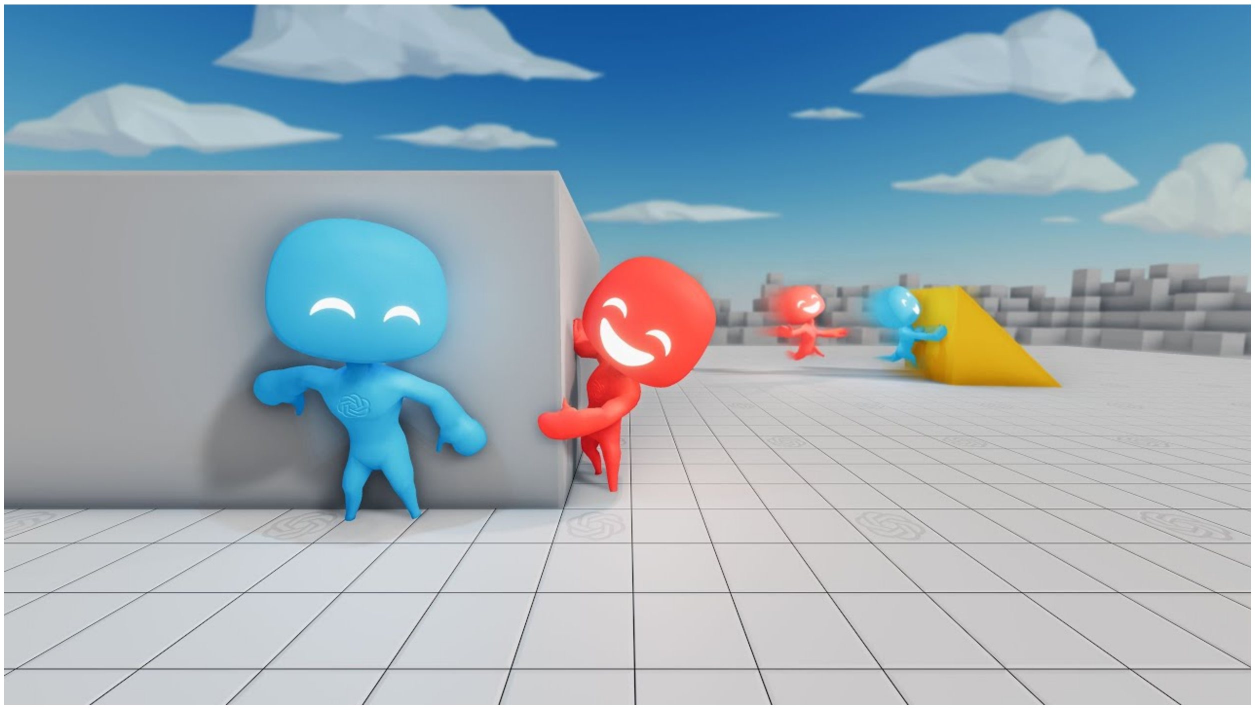
Awesome! One of my colleagues will be with you shortly 😊

Competitive AI "Agents"

Trained with **Reinforcement Learning (RL)** that leaverages Deep NNs.

Agent learning via self-play competition

OpenAI: "... agents build a series of six distinct strategies and counterstrategies, some of which we did not know our environment supported."



Self-play YouTube video link: [https://www.youtube.com/watch?v=kopoLzvh5jY (https://www.youtube.com/watch?v=kopoLzvh5jY)](https://www.youtube.com/watch?v=kopoLzvh5jY)

Physical modeling and sensor processing

Titles of some papers and press articles

- Accelerating Physics-Based Simulations Using Neural Network Proxies: An Application in Oil Reservoir Modeling link (https://arxiv.org/abs/1906.01510)
- Deep Learning Accelerates Scientific Simulations up to Two Billion Times link (https://www.infoq.com/news/2020/03/deep-learning-simulation/)
- Practical Processing of Mobile Sensor Data for Continual Deep Learning Predictions link (https://arxiv.org/abs/1705.06224)

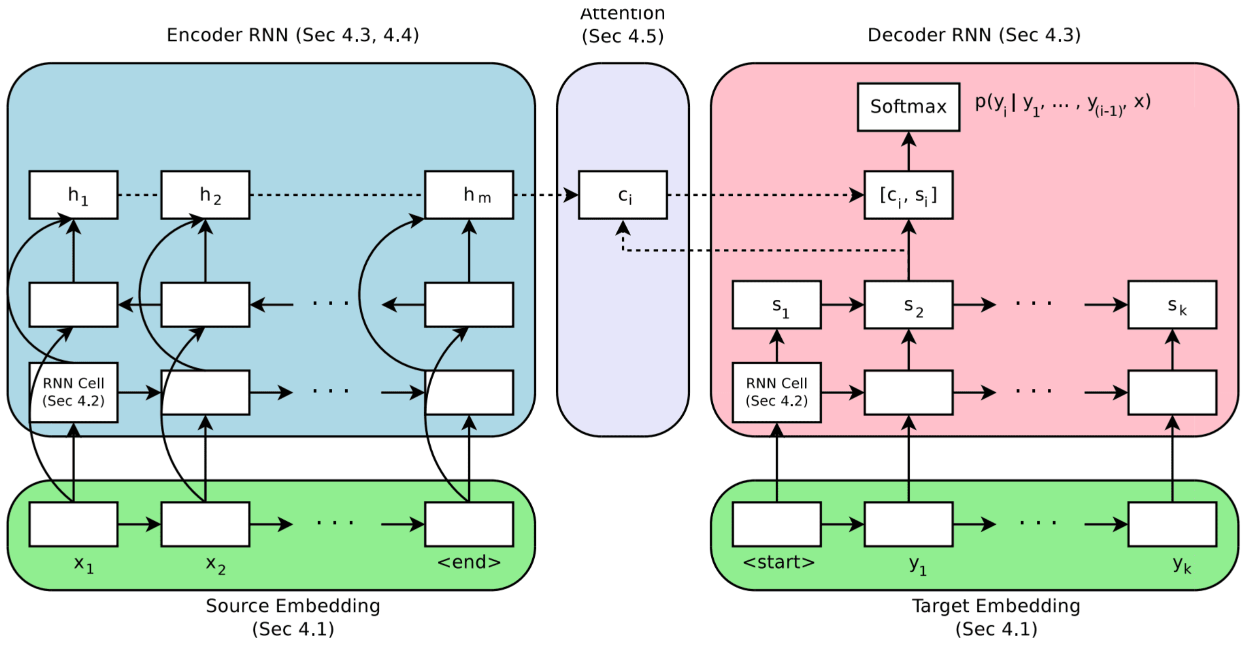
But what about structured (tabulated) data?

- **Structured data** is the type of data we've worked with to date. I.e., data that can be represented in a **table or matrix**, with rows of samples and columns of features.
- Deep NNs can perform well on structured data (despite wide belief in the mid-2010s that they underperformed traditional methods). However, **many traditional methods perform about as well, and often with lower computational costs**.

Neural Network Architectures

Automated discovery of good neural architectures is a very active research area

- Network architectures generally relate to the **type of task** they are used for.
  - **Feed-forward**
    - Layered (our examples above)
    - Convolution (models mammalian vision--we'll use one in the next lecture)
  - **Recurrent**
    - Dynamic input (e.g., streaming sensor data)
    - Short-term memory requirements (language translation)
  - **Feed-back**
    - Reinforcement learning
    - Models of biological networks
    - "Attention" networks
- Larger networks (more neurons and synapses/connections) are often better, but harder to train

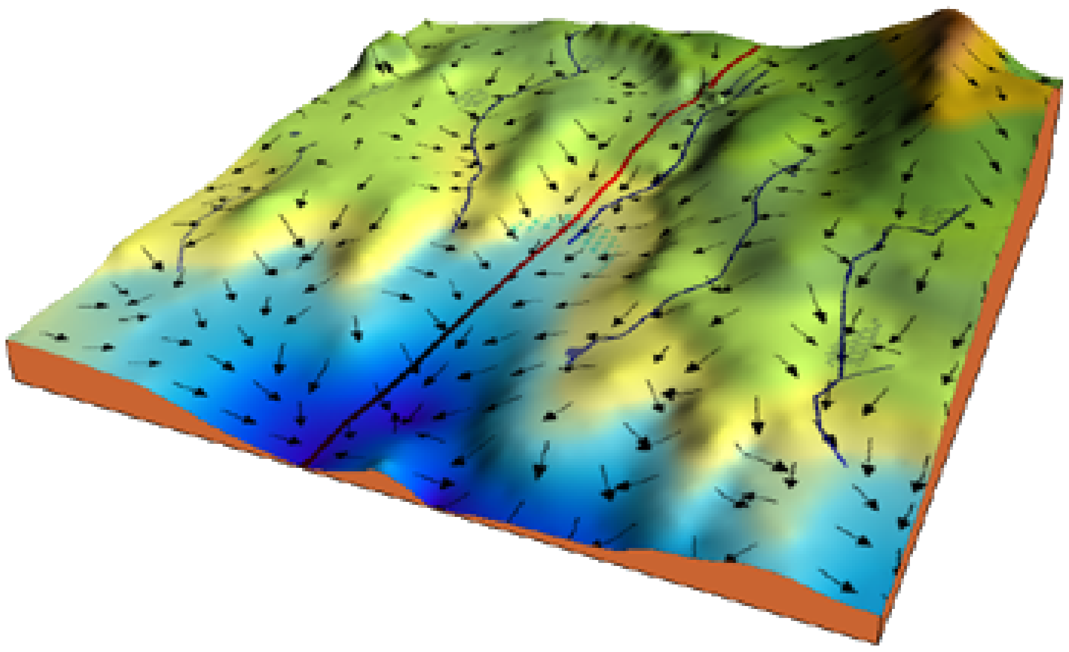


Training: Gradient Descent and Backpropagation

Gradient Descent

- Used for **supervised learning**
- The algorithm takes **iterative steps**, trying to **find the miminum of an error (cost, loss) function**
- **Steps**:

  1. For iteration n, compute difference (error) between network outputs and targets
  2. Compute derivative of error function, with respet to model parameters
  3. Adjust model parameters (add/subtract) by the derivative (negative of the gradient), scaled by a step size.
  4. Repeat for iteration n+1

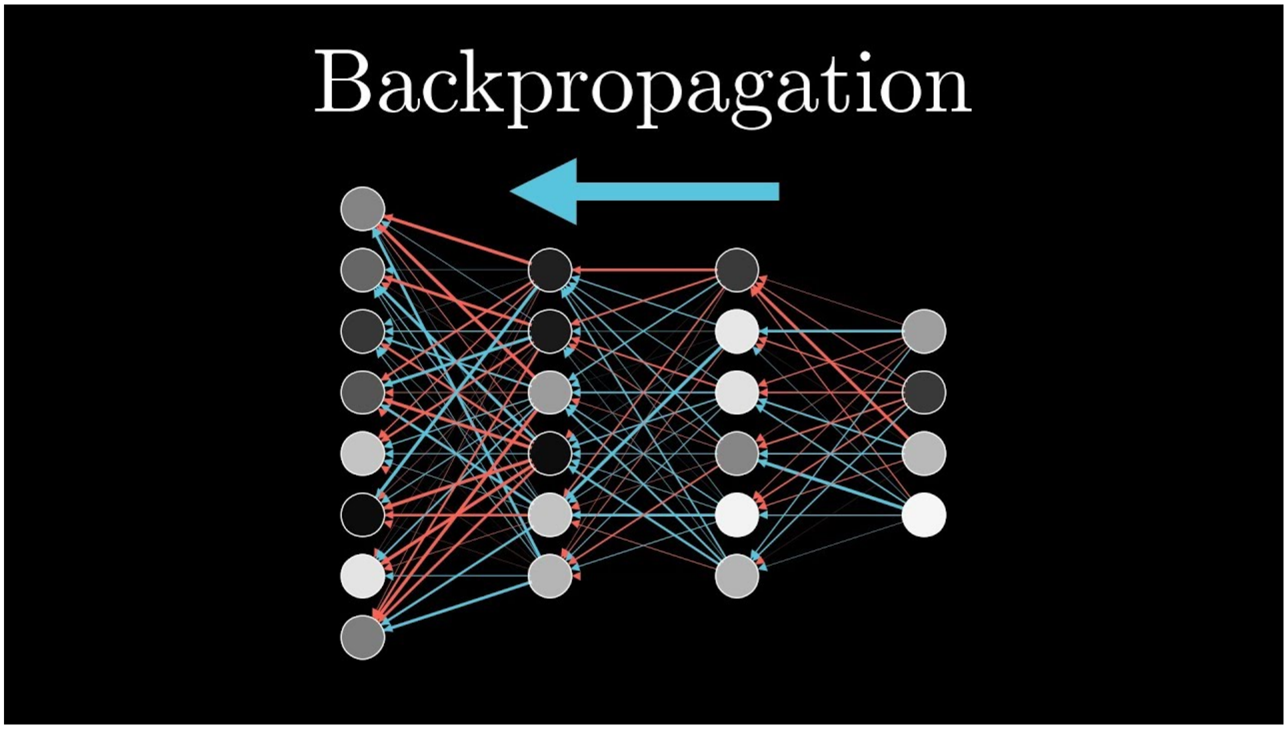


Mini-batch Gradient Descent

- Used to help prevent getting stuck in a local minimum
- For each iteration, we **compute the error (and derivative) only for a subset of the training samples**
- Thus, **the error surface is slightly different for each iteration**
- Imagine a rumbling of the surface in the image above. A ball stuck in a local minimum might be "popped" out of that location, and then start rolling toward a lower minimum.

Backpropagation

- Backpropagation is an efficient way to compute gradients in layered NNs.
- It is derived by using the **Chain Rule** from calculus (see the **3Blue1Brown video**).
- Steps (vaguely):
  1. Run your training samples through the network (the "forward pass").
  2. Compute the error from the network output predictions and the targets.
  3. **Send that error backwards through the network**. The backwards activations of neurons in layer L, and weights between layer L and L+1, can be used to easily compute the gradient for those weights.



Let's code!

TensorFlow and PyTorch are the two most popular software tools for working with deep neural networks. We'll use TensorFlow.

We'll create layered network with "fully-connected" or "dense" connections between layers, and apply it to the Iris dataset.

In [1]:
```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(0)
tf.random.set_seed(0)
```

In [2]:
```python
# Import the iris data and create train/test sets
iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

# We'll normalize our data, which often results in better models
transformer = Normalizer().fit(X_train)
X_train = transformer.transform(X_train)
X_test = transformer.transform(X_test)

# Get the number of target classes (which will be 3), and samples (150)
num_classes = len(np.unique(y))
num_samples = len(y)
```

We'll be using the "Keras" API for TensorFlow. Keras has become a de facto API for creating deep neural networks with TensorFlow, PyTorch, and other NN frameworks.

https://www.tensorflow.org/guide/keras/overview (https://www.tensorflow.org/guide/keras/overview)

This is a small dataset, with few features (4) and few samples (150) so we'll use a very small NN model.

In [3]:
```
## First we construct our network architecture...

num_hidden = 10 # number of hidden neurons we'll use, in each hidden layer

# We specify the use of a sequential model--that is, a layered, feed-forward model
model = tf.keras.Sequential()

# Start with a layer of num_hidden neurons, that take in the input features
model.add(layers.Dense(num_hidden, activation='relu'))

# Add a second layer of num_hidden neurons, fully connected to the previous layer
model.add(layers.Dense(num_hidden, activation='relu'))

# Add an output layer, with number of output units equal to our number of classes
# Again, these neurons are fully connected to those of the previous layers
model.add(layers.Dense(num_classes))


## Now we 'compile' the model. This will make it execute more quickly during train
#
# - The *optimizer* specifies what our step size and "momentum" factors are, for g
# - The *loss* function is "cross-entropy", rather than RMSE or other losses we've
#   in the past. It's very effective when training classification models.
# - The *metrics* argument just lets TensorFlow that we want to record/monitor the
#   scores during training (we might have specified something else, like F1-score)
#
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
              metrics=['accuracy'])


## Finally, we fit the model to the data. For each gradient descent iteration we'l
# a (mini-) batch size of 30. Commonly used batch sizes range from 8 to 32.
# An "epoch" is the number of iterations that is needed such that all the training
# have been used one time in training--thus, 150/30==5, in this case. We train for
# epochs, to see if the model gets to a good fit, and then starts to overfit.
epochs = 300
batch_size = num_samples//5
history = model.fit(X_train, y_train, epochs=epochs,
                    batch_size=batch_size, validation_data=(X_test, y_test))
```

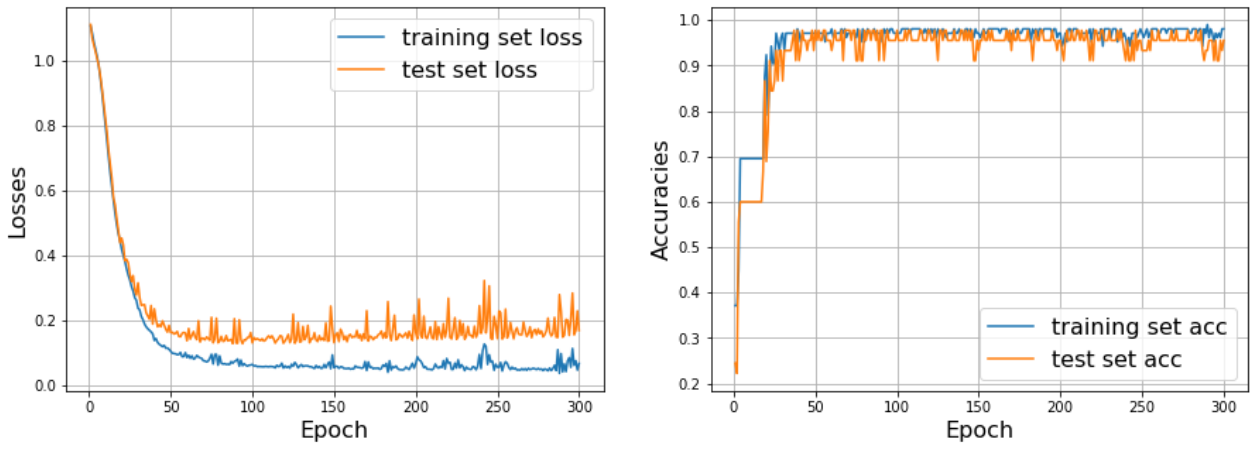
```
Epoch 1/300
4/4 [==============================] - 1s 85ms/step - loss: 1.1065 - accuracy:
0.3714 - val_loss: 1.1111 - val_accuracy: 0.2444
Epoch 2/300
4/4 [==============================] - 0s 12ms/step - loss: 1.0835 - accuracy:
0.3714 - val_loss: 1.0682 - val_accuracy: 0.2222
Epoch 3/300
4/4 [==============================] - 0s 12ms/step - loss: 1.0567 - accuracy:
0.4476 - val_loss: 1.0477 - val_accuracy: 0.5556
Epoch 4/300
4/4 [==============================] - 0s 17ms/step - loss: 1.0355 - accuracy:
0.6952 - val_loss: 1.0291 - val_accuracy: 0.6000
Epoch 5/300
4/4 [==============================] - 0s 10ms/step - loss: 1.0119 - accuracy:
0.6952 - val_loss: 1.0063 - val_accuracy: 0.6000
Epoch 6/300
4/4 [==============================] - 0s 12ms/step - loss: 0.9822 - accuracy:
0.6952 - val_loss: 0.9837 - val_accuracy: 0.6000
Epoch 7/300
```

In [5]:
```python
## We can plot the accuracy scores (for both training and test sets) that were rec
#  as well as the cross-entropy losses...

plt.figure(figsize=(15, 5))
fs = 16 # fontsize

# plot loss information for training and testing
plt.subplot(1,2,1)
plt.plot(np.arange(1,epochs+1), history.history['loss'], label='training set loss'
plt.plot(np.arange(1,epochs+1), history.history['val_loss'], label='test set loss'
plt.legend(fontsize=fs)
plt.xlabel('Epoch', fontsize=fs)
plt.ylabel('Losses', fontsize=fs)
plt.grid(True)

# plot accuracy information for training and testing
plt.subplot(1,2,2)
plt.plot(np.arange(1,epochs+1), history.history['accuracy'], label='training set ac
plt.plot(np.arange(1,epochs+1), history.history['val_accuracy'], label='test set ac
plt.legend(fontsize=fs)
plt.xlabel('Epoch', fontsize=fs)
plt.ylabel('Accuracies', fontsize=fs)
plt.grid(True)
```



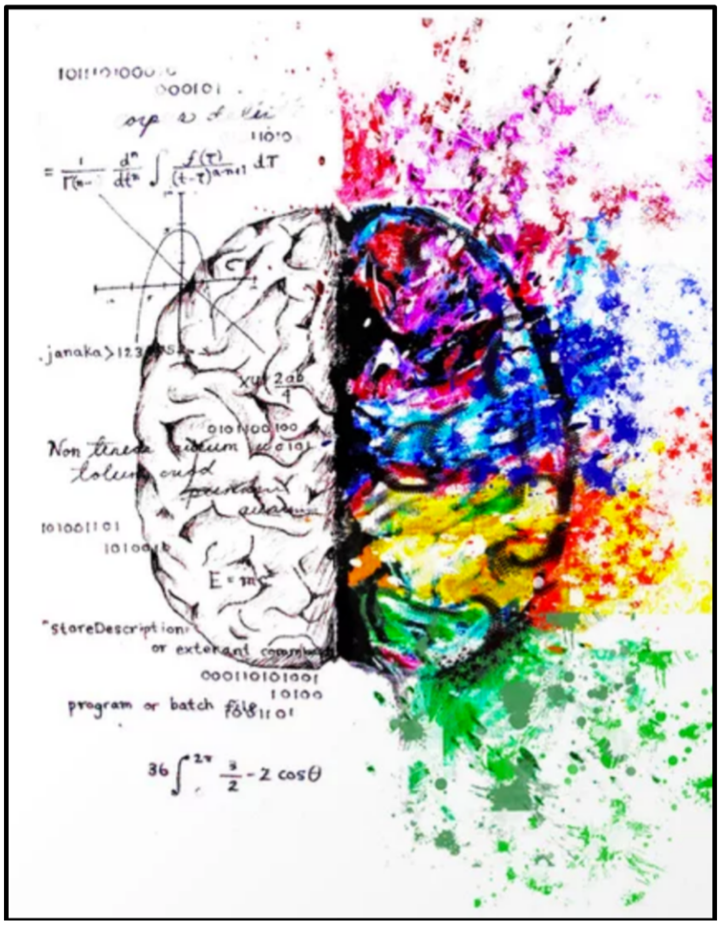
Results

- The training and test set **accuracies** did not diverge, even after many epochs of training.
- Thus, our model does not suffer from (much) over-fitting.
- This is likely due to the low number of features, and the compactness of the target clusters in the feature space.
- The test set **loss** does diverge slightly, however. Suggesting there is some overfitting, but not enough to notably impact the model's classification predictions.

Instructor self-promotion!

If you have further interest in the similarities between biological neurons and artificial neurons, you might enjoy this 2019 blog post from Matt Roos.

[Deep Learning Neurons versus Biological Neurons (https://towardsdatascience.com/deep-learning-versus-biological-neurons-floating-point-numbers-spikes-and-neurotransmitters-6eebfa3390e9)](https://towardsdatascience.com/deep-learning-versus-biological-neurons-floating-point-numbers-spikes-and-neurotransmitters-6eebfa3390e9)



Next time

- Convolutional neural networks (CNNs): NNs that model the visual system, and perform well on images tasks.