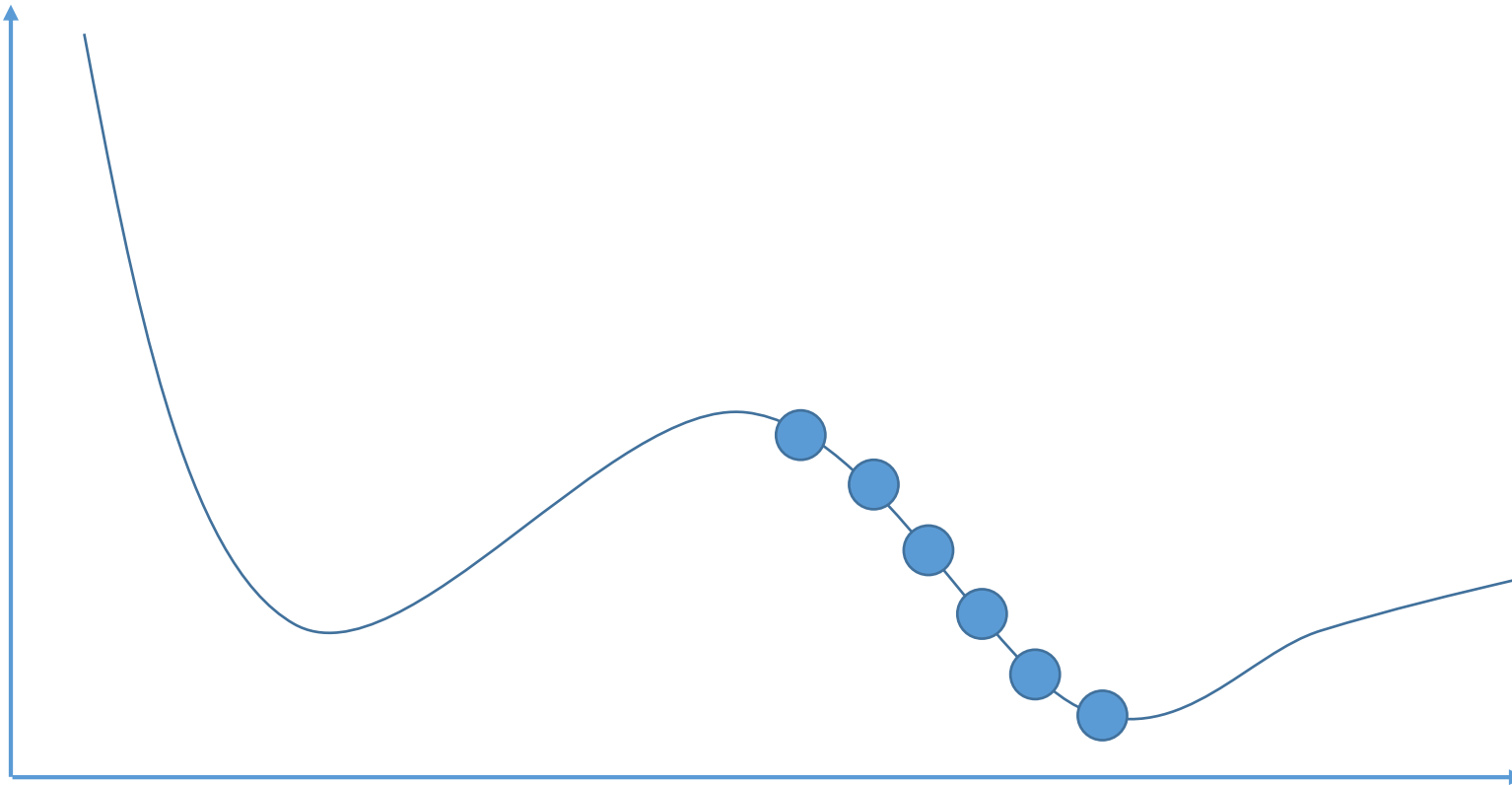


Deep Learning: Pre-Requisites

Understanding gradient descent, autodiff, and softmax

Gradient Descent



autodiff

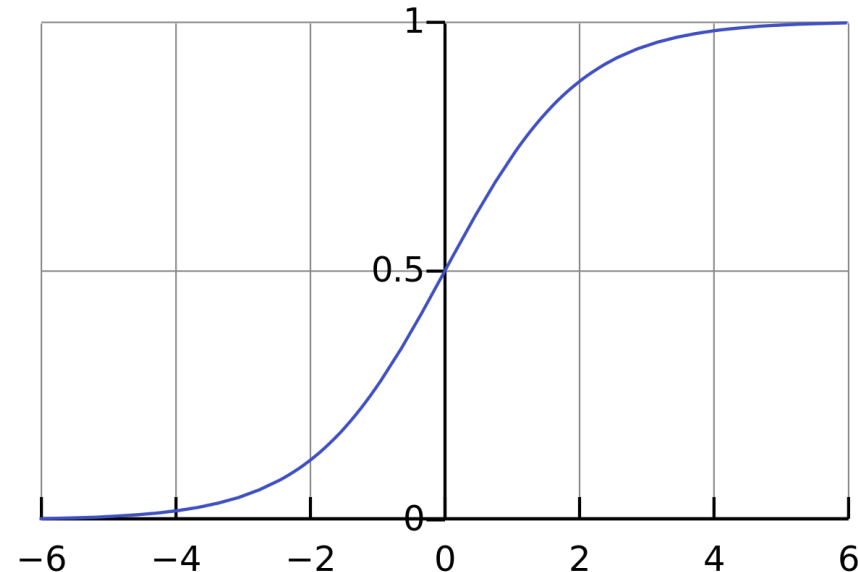
- Gradient descent requires knowledge of, well, the gradient from your cost function (MSE)
- Mathematically we need the first partial derivatives of all the inputs
 - This is hard and inefficient if you just throw calculus at the problem
- Reverse-mode autodiff to the rescue!
 - Optimized for many inputs + few outputs (like a neuron)
 - Computes all partial derivatives in # of outputs + 1 graph traversals
 - Still fundamentally a calculus trick – it's complicated but it works
 - This is what Tensorflow uses

softmax

- Used for classification
 - Given a score for each class
 - It produces a probability of each class
 - The class with the highest probability is the “answer” you get

$$h_{\theta}(x) = \frac{1}{1 + \exp(-\theta^T x)},$$

x is a vector of input values
theta is a vector of weights



In review:

- Gradient descent is an algorithm for minimizing error over multiple steps
- Autodiff is a calculus trick for finding the gradients in gradient descent
- Softmax is a function for choosing the most probable classification given several input values



Introducing Artificial Neural Networks

Evolving beyond nature

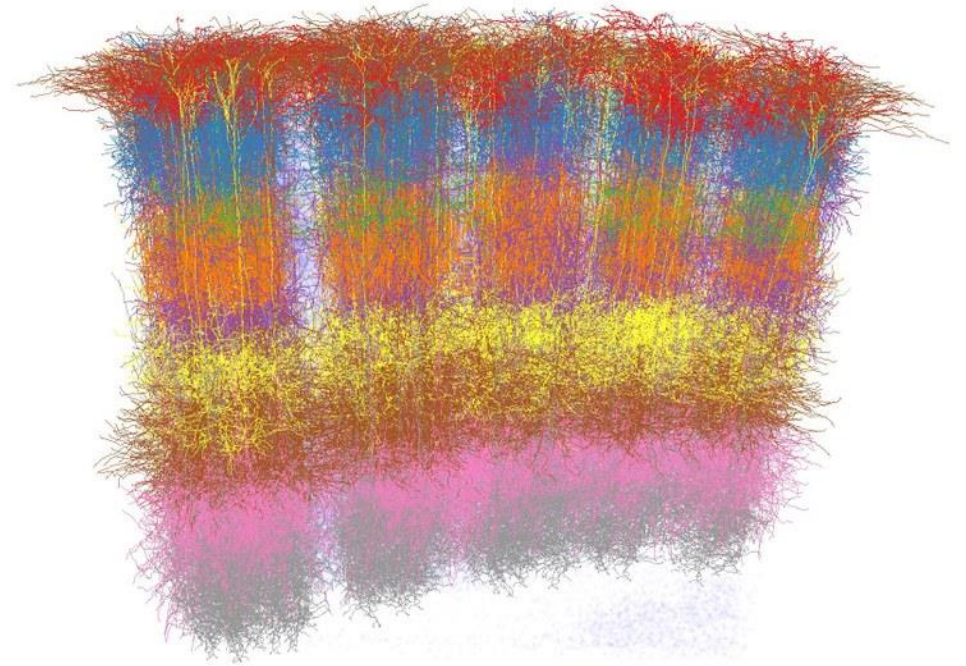
The biological inspiration

- Neurons in your cerebral cortex are connected via axons
- A neuron “fires” to the neurons it’s connected to, when enough of its input signals are activated.
- Very simple at the individual neuron level – but layers of neurons connected in this way can yield learning behavior.
- Billions of neurons, each with thousands of connections, yields a mind



Cortical columns

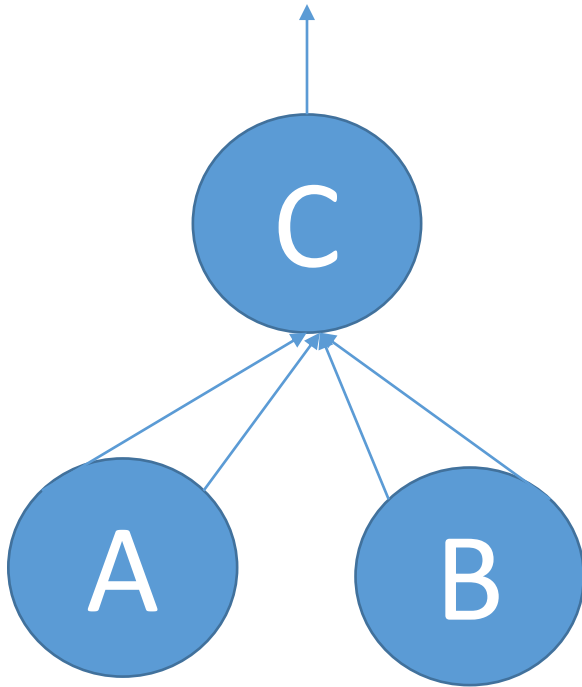
- Neurons in your cortex seem to be arranged into many stacks, or “columns” that process information in parallel
- “mini-columns” of around 100 neurons are organized into larger “hyper-columns”. There are 100 million mini-columns in your cortex
- This is coincidentally similar to how GPU’s work...



(credit: Marcel Oberlaender et al.)

The first artificial neurons

- 1943!!



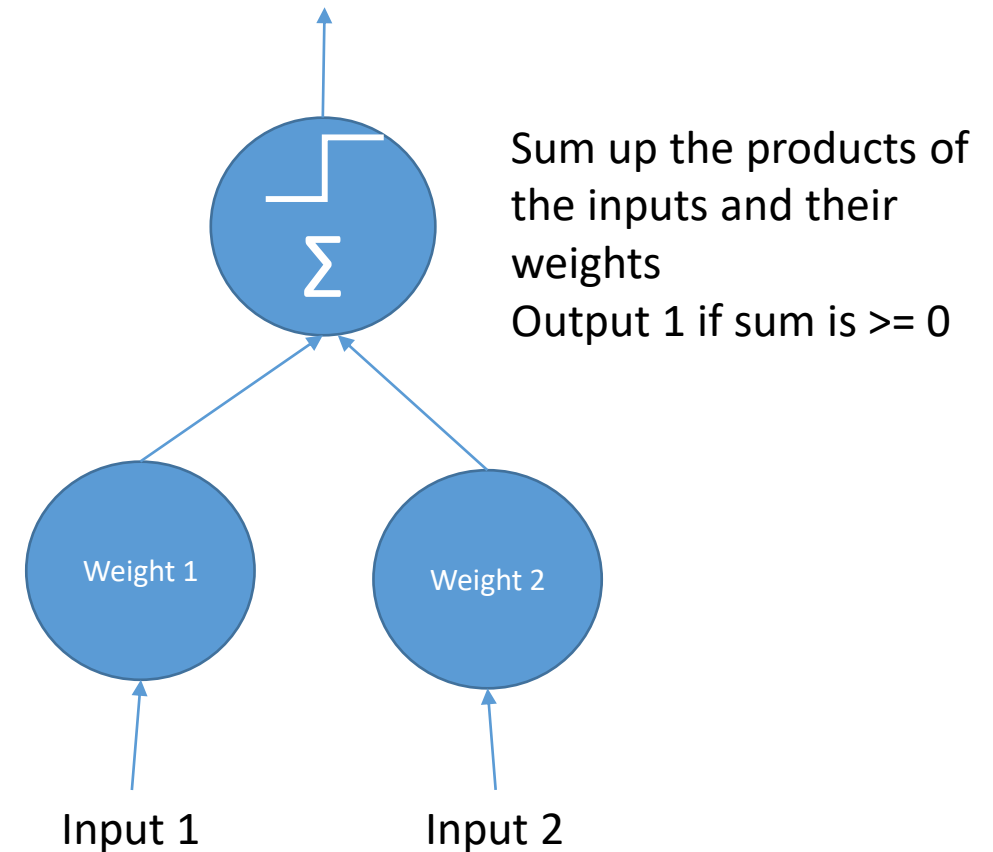
An artificial neuron “fires” if more than N input connections are active.

Depending on the number of connections from each input neuron, and whether a connection activates or suppresses a neuron, you can construct AND, OR, and NOT logical constructs this way.

This example would implement $C = A \text{ OR } B$ if the threshold is 2 inputs being active.

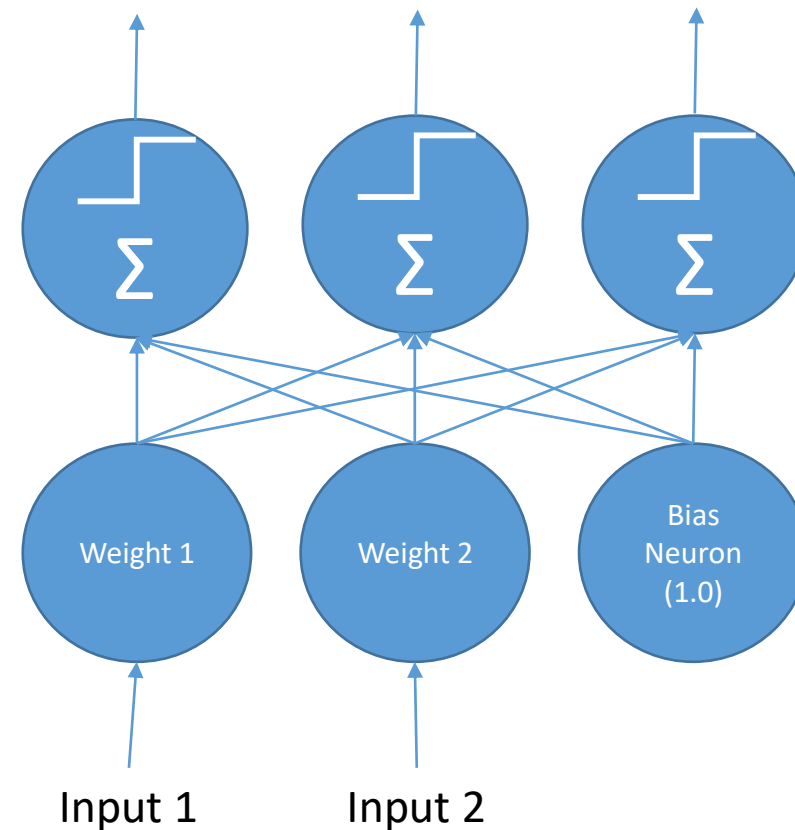
The Linear Threshold Unit (LTU)

- 1957!
- Adds weights to the inputs; output is given by a step function



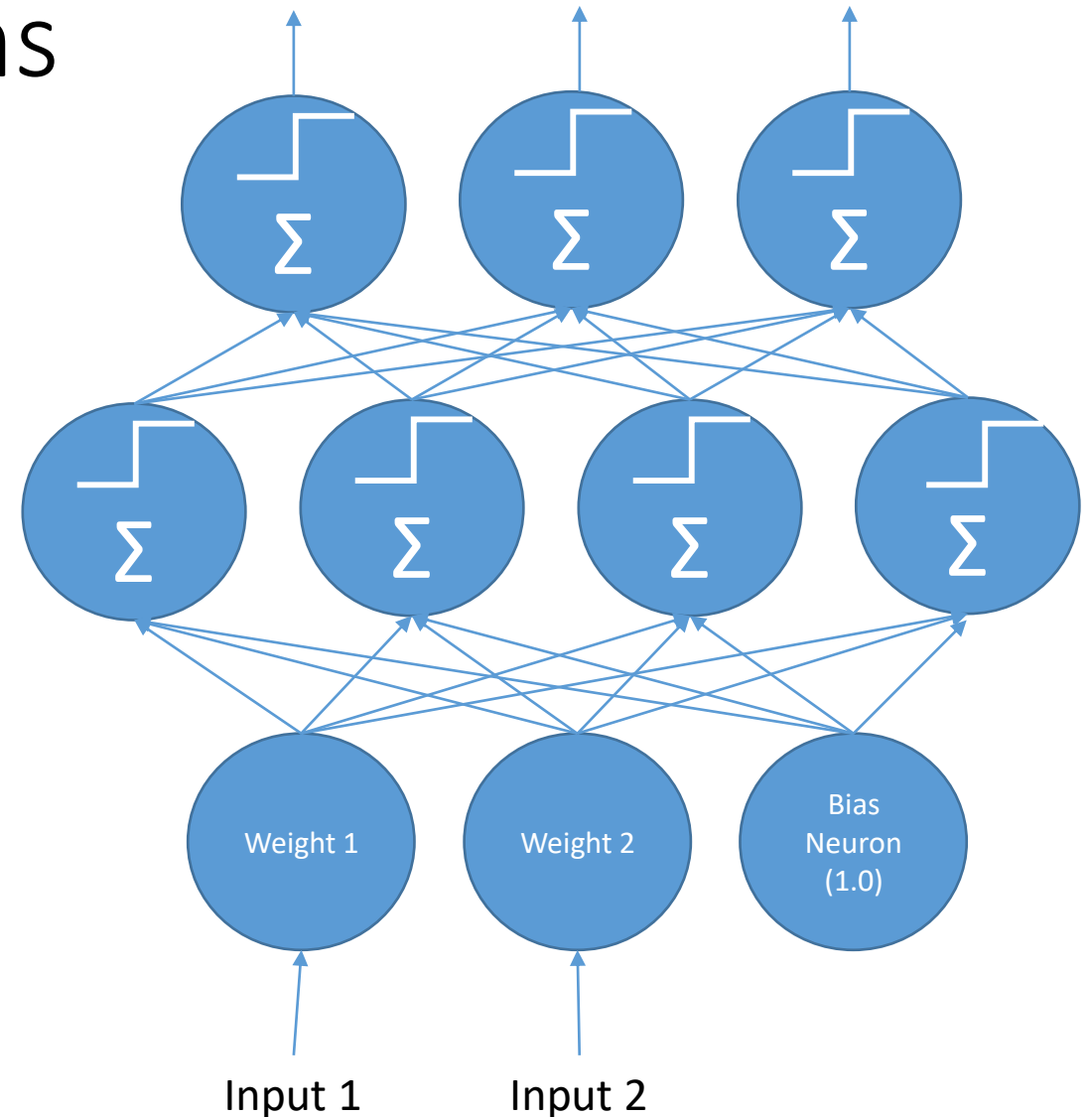
The Perceptron

- A layer of LTU's
- A perceptron can learn by reinforcing weights that lead to correct behavior during training
- This too has a biological basis (“cells that fire together, wire together”)



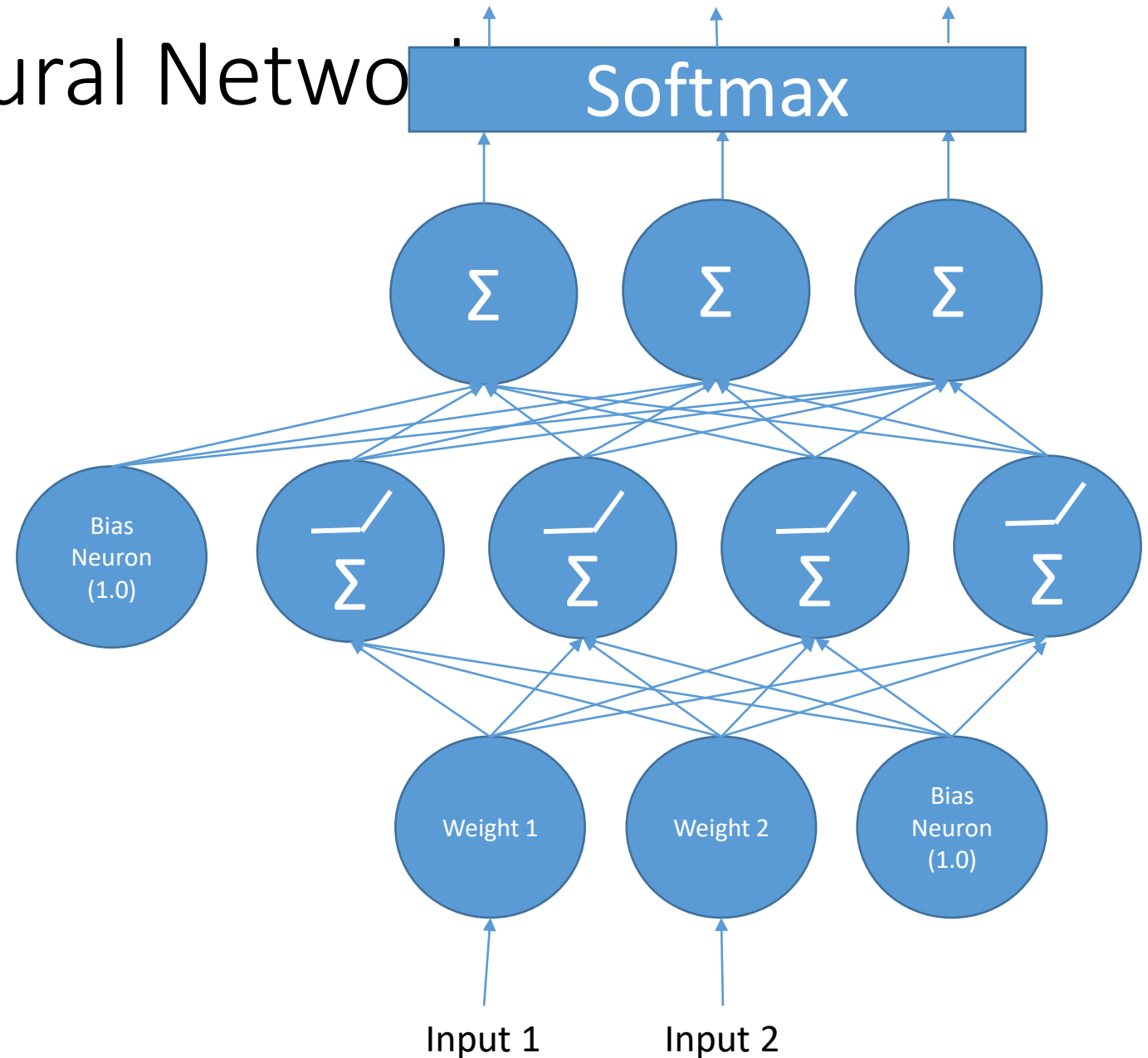
Multi-Layer Perceptrons

- Addition of “hidden layers”
- This is a Deep Neural Network
- Training them is trickier – but we’ll talk about that.



A Modern Deep Neural Network

- Replace step activation function with something better
- Apply softmax to the output
- Training using gradient descent



Let's play

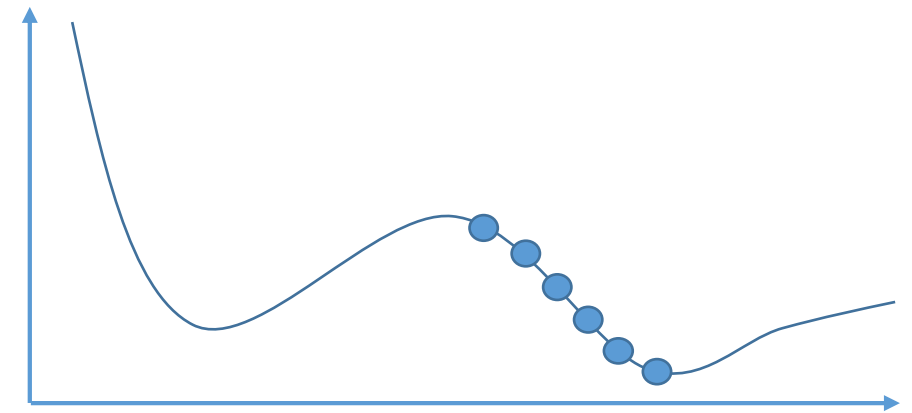
playground.tensorflow.org

Deep Learning

Constructing, training, and tuning multi-layer perceptrons

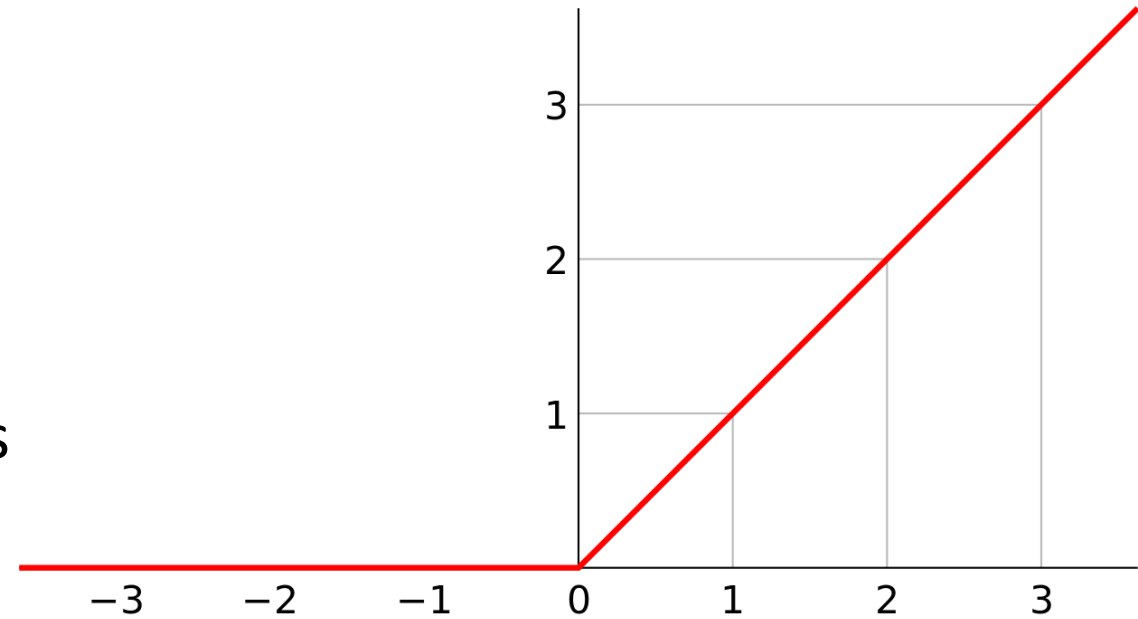
Backpropagation

- How do you train a MLP's weights? How does it **learn**?
- Backpropagation... or more specifically: Gradient Descent using reverse-mode autodiff!
- For each training step:
 - Compute the output error
 - Compute how much each neuron in the previous hidden layer contributed
 - Back-propagate that error in a reverse pass
 - Tweak weights to reduce the error using gradient descent



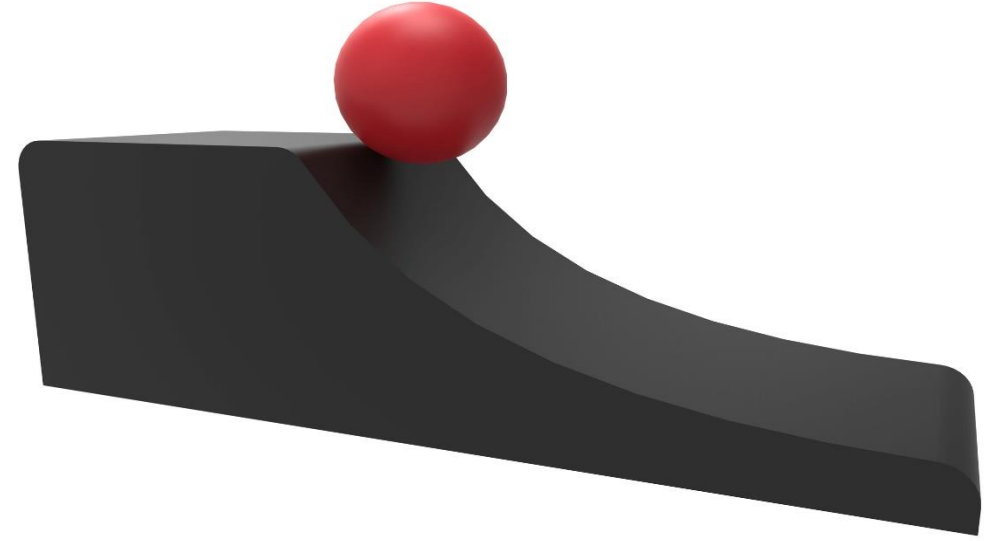
Activation functions (aka rectifier)

- Step functions don't work with gradient descent – there is no gradient!
 - Mathematically, they have no useful derivative.
- Alternatives:
 - Logistic function
 - Hyperbolic tangent function
 - Exponential linear unit (ELU)
 - ReLU function (Rectified Linear Unit)
- ReLU is common. Fast to compute and works well.
 - Also: “Leaky ReLU”, “Noisy ReLU”
 - ELU can sometimes lead to faster learning though.



ReLU function

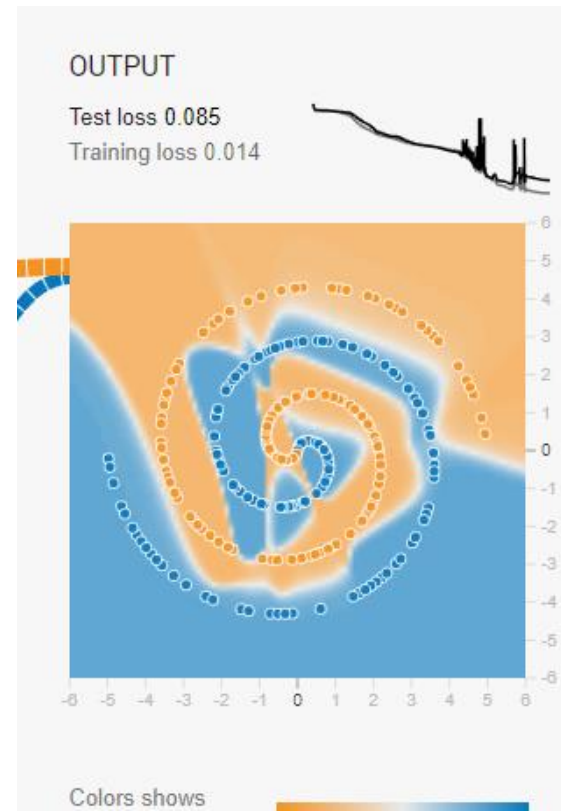
Optimization functions



- There are faster (as in faster learning) optimizers than gradient descent
 - Momentum Optimization
 - Introduces a momentum term to the descent, so it slows down as things start to flatten and speeds up as the slope is steep
 - Nesterov Accelerated Gradient
 - A small tweak on momentum optimization – computes momentum based on the gradient slightly ahead of you, not where you are
 - RMSProp
 - Adaptive learning rate to help point toward the minimum
 - Adam
 - Adaptive moment estimation – momentum + RMSProp combined
 - Popular choice today, easy to use

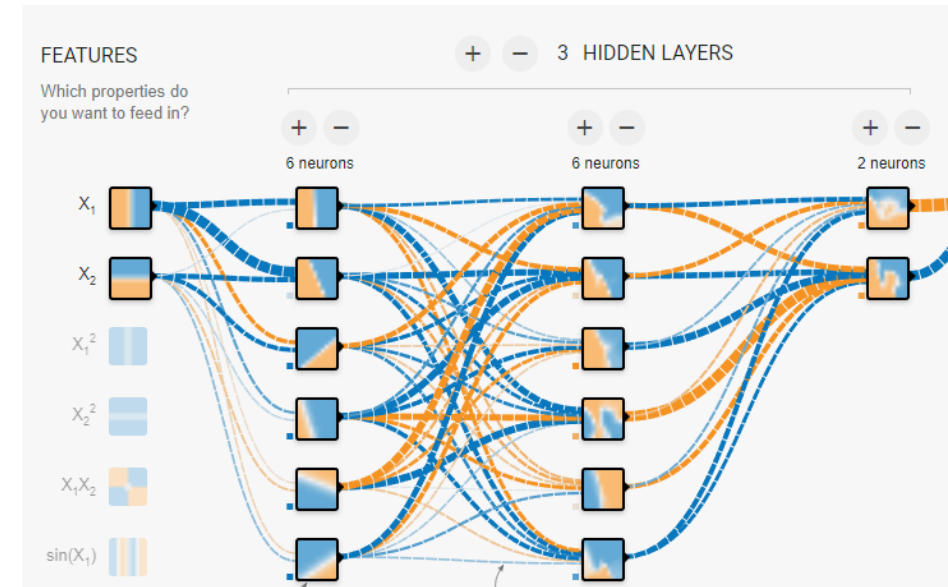
Avoiding Overfitting

- With thousands of weights to tune, overfitting is a problem
- Early stopping (when performance starts dropping)
- Regularization terms added to cost function during training
- Dropout – ignore say 50% of all neurons randomly at each training step
 - Works surprisingly well!
 - Forces your model to spread out its learning



Tuning your topology

- Trial & error is one way
 - Evaluate a smaller network with less neurons in the hidden layers
 - Evaluate a larger network with more layers
 - Try reducing the size of each layer as you progress – form a funnel
- More layers can yield faster learning
- Or just use more layers and neurons than you need, and don't care because you use early stopping.
- Use “model zoos”



Tensorflow

Why Tensorflow?

- It's not specifically for neural networks– it's more generally an architecture for executing a graph of numerical operations
- Tensorflow can optimize the processing of that graph, and distribute its processing across a network
 - Sounds a lot like Apache Spark, eh?
- It can also distribute work across GPU's!
 - Can handle massive scale – it was made by Google
- Runs on about anything
- Highly efficient C++ code with easy to use Python API's

Tensorflow basics

- Install with `pip install tensorflow` or `pip install tensorflow-gpu`
- A tensor is just a fancy name for an array or matrix of values
- To use Tensorflow, you:
 - Construct a graph to compute your tensors
 - Initialize your variables
 - Execute that graph – nothing actually happens until then

World's simplest Tensorflow app:

```
import tensorflow as tf
```

```
a = tf.Variable(1, name="a")
```

```
b = tf.Variable(2, name="b")
```

```
f = a + b
```

```
init = tf.global_variables_initializer()
```

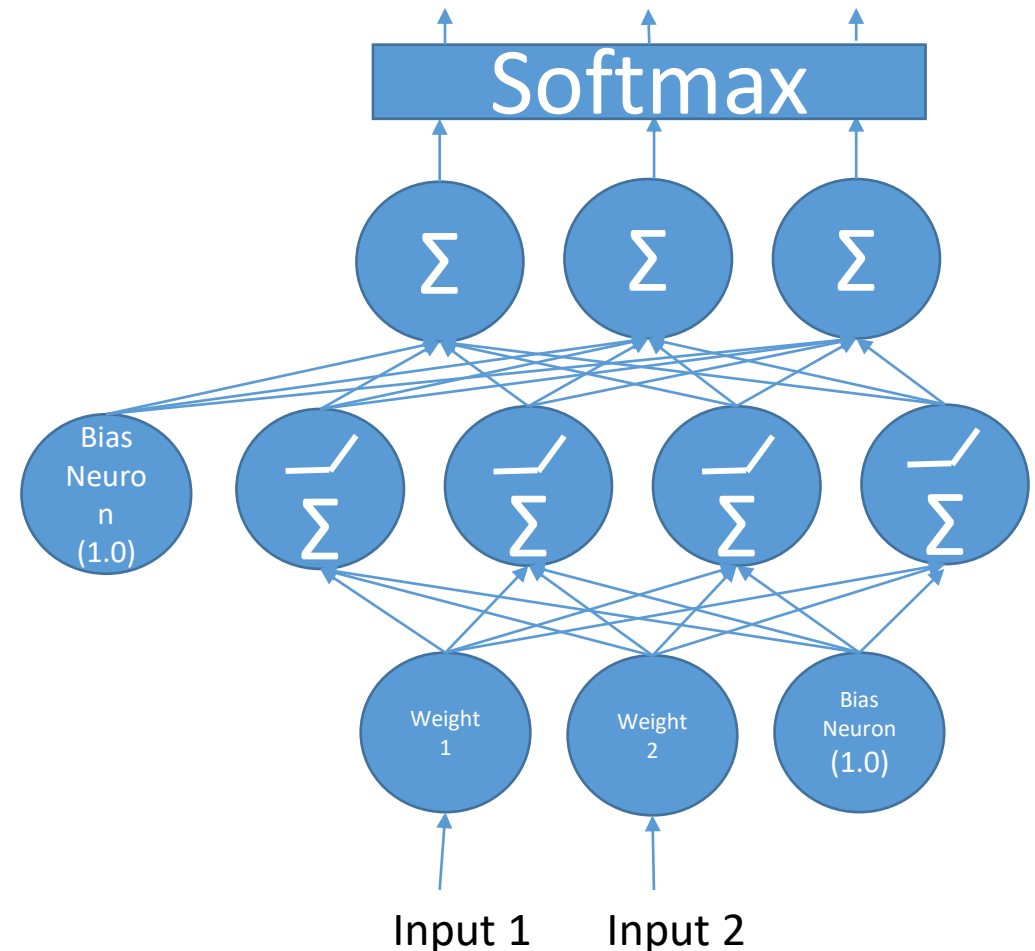
```
with tf.Session() as s:
```

```
    init.run()
```

```
    print( f.eval() )
```

Creating a neural network with Tensorflow

- Mathematical insights:
 - All those interconnected arrows multiplying weights can be thought of as a big matrix multiplication
 - The bias term can just be added onto the result of that matrix multiplication
- So in Tensorflow, we can define a layer of a neural network as:
$$\text{output} = \text{tf.matmul}(\text{previous_layer}, \text{layer_weights}) + \text{layer_biases}$$
- By using Tensorflow directly we're kinda doing this the "hard way."



Creating a neural network with Tensorflow

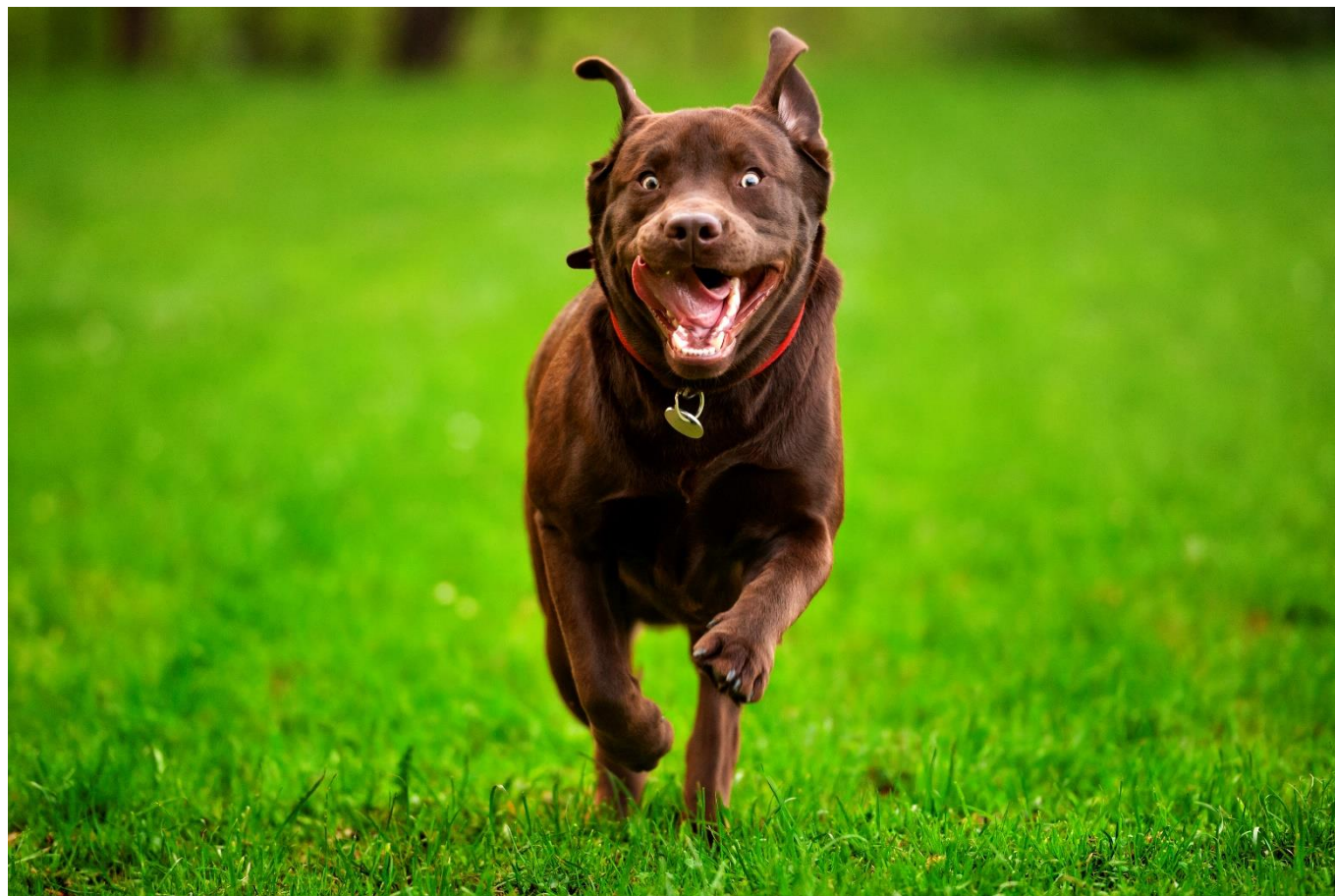
- Load up our training and testing data
- Construct a graph describing our neural network
 - Use **placeholders** for the input data and target labels
 - This way we can use the same graph for training and testing!
 - Use **variables** for the learned weights for each connection and learned biases for each neuron
 - Variables are preserved across runs within a Tensorflow session
- Associate an optimizer (ie gradient descent) to the network
- Run the optimizer with your training data
- Evaluate your trained network with your testing data



Make sure your features are normalized

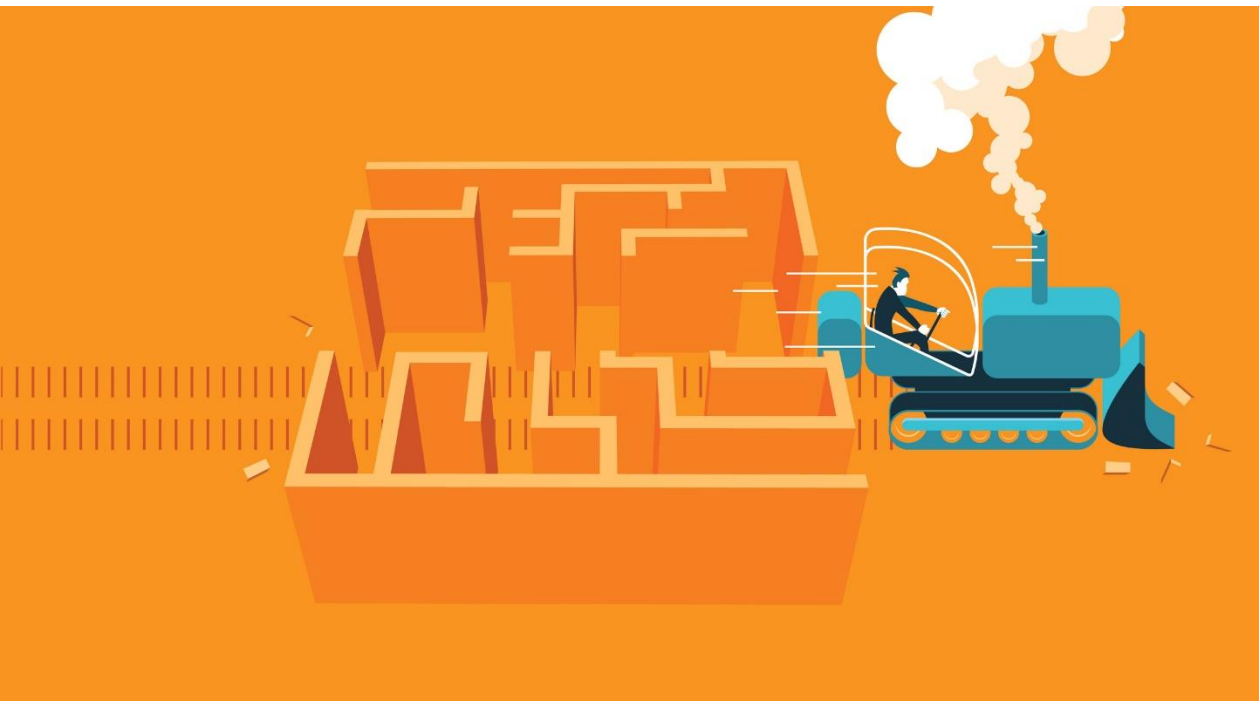
- Neural networks usually work best if your input data is normalized.
 - That is, 0 mean and unit variance
 - The real goal is that every input feature is comparable in terms of magnitude
- scikit_learn's StandardScaler can do this for you
- Many data sets are normalized to begin with – such as the one we're about to use.

Let's try it out



Keras

Why Keras?



- Easy and fast prototyping
 - Available as a higher-level API within Tensorflow 1.9+
 - scikit_learn integration
 - Less to think about – which often yields better results without even trying
 - This is really important! The faster you can experiment, the better your results.

Let's dive in: MNIST with Keras



Example: multi-class classification

- MNIST is an example of multi-class classification.

```
model = Sequential()

model.add(Dense(64, activation='relu', input_dim=28))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9,
          nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd, metrics=['accuracy'])
```

Example: binary classification

```
model = Sequential()  
model.add(Dense(64, input_dim=20, activation='relu'))  
model.add(Dropout(0.5))  
model.add(Dense(64, activation='relu'))  
model.add(Dropout(0.5))  
model.add(Dense(1, activation='sigmoid'))  
model.compile(loss='binary_crossentropy', optimizer='rmsprop',  
              metrics=['accuracy'])
```

Integrating Keras with scikit_learn

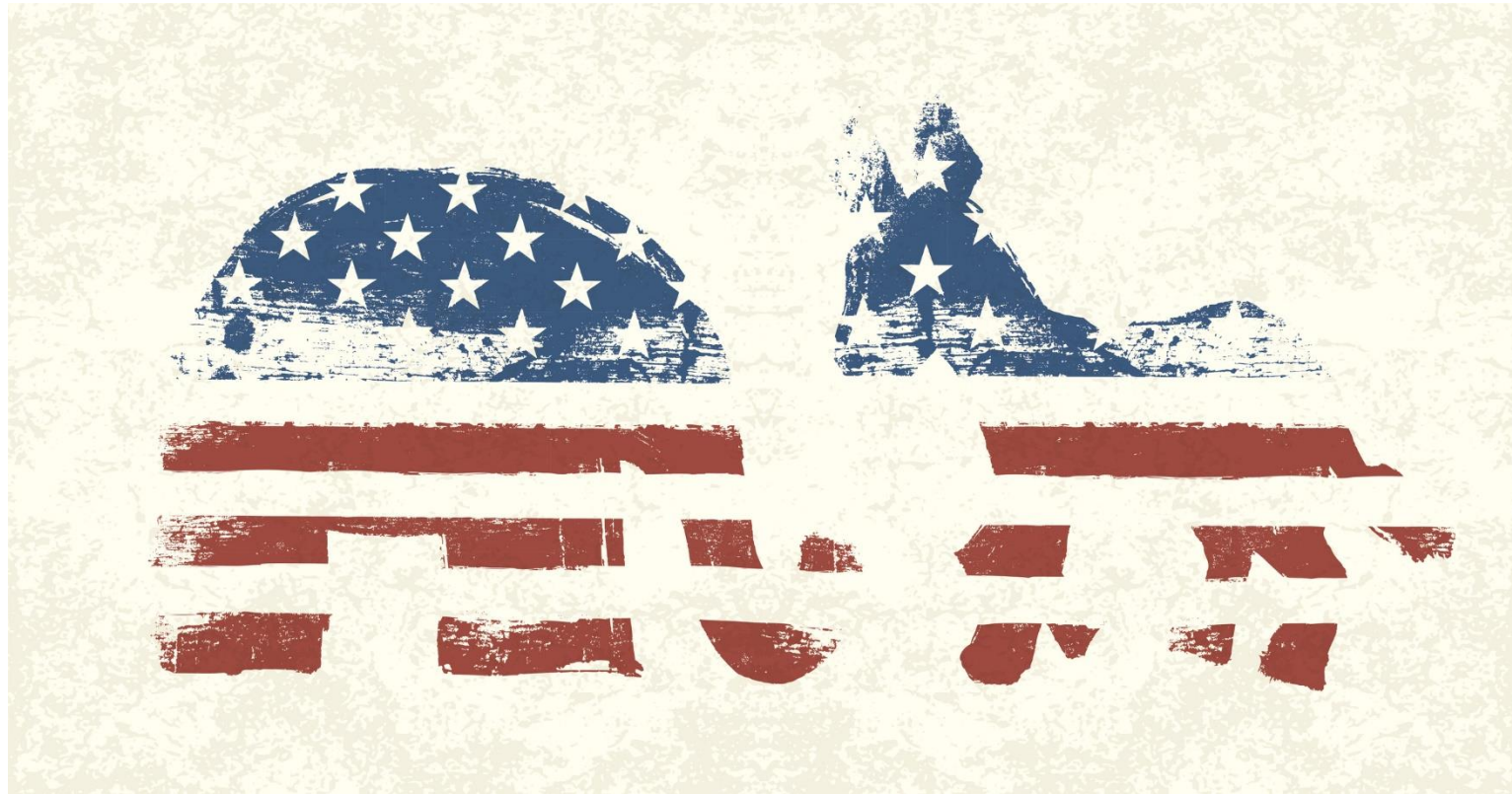
```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

def create_model():
    model = Sequential()
    model.add(Dense(6, input_dim=4, kernel_initializer='normal', activation='relu'))
    model.add(Dense(4, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimator = KerasClassifier(build_fn=create_model, epochs=100, verbose=0)

cv_scores = cross_val_score(estimator, features, labels, cv=10)
print(cv_scores.mean())
```

Let's try it out: predict political parties with Keras



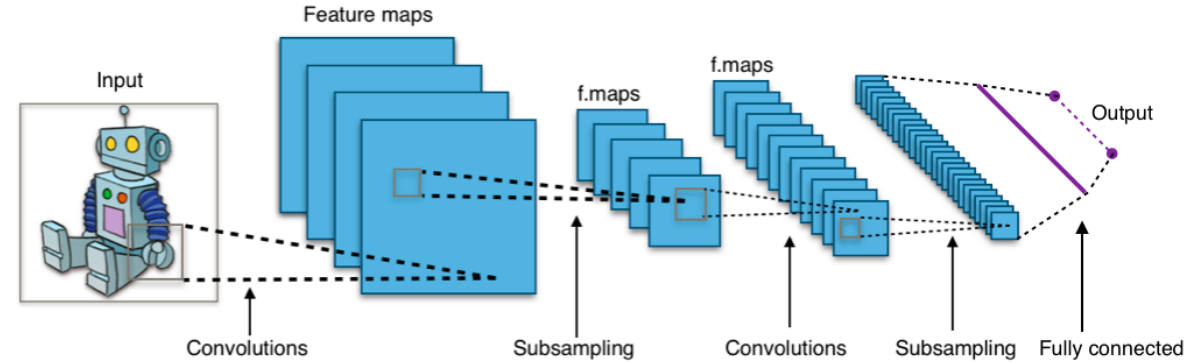
Convolutional Neural Networks

CNN's: what are they for?

- When you have data that doesn't neatly align into columns
 - Images that you want to find features within
 - Machine translation
 - Sentence classification
 - Sentiment analysis
- They can find features that aren't in a specific spot
 - Like a stop sign in a picture
 - Or words within a sentence
- They are “feature-location invariant”



CNN's: how do they wor



- Inspired by the biology of the visual cortex
 - Local receptive fields are groups of neurons that only respond to a part of what your eyes see (subsampling)
 - They overlap each other to cover the entire visual field (convolutions)
 - They feed into higher layers that identify increasingly complex images
 - Some receptive fields identify horizontal lines, lines at different angles, etc. (filters)
 - These would feed into a layer that identifies shapes
 - Which might feed into a layer that identifies objects
 - For color images, extra layers for red, green, and blue

How do we “know” that’s a stop sign?

- Individual local receptive fields scan the image looking for edges, and pick up the edges of the stop sign in a layer
- Those edges in turn get picked up by a higher level convolution that identifies the stop sign’s shape (and letters, too)
- This shape then gets matched against your pattern of what a stop sign looks like, also using the strong red signal coming from your red layers
- That information keeps getting processed upward until your foot hits the brake!
- A CNN works the same way



CNN's with Keras

- Source data must be of appropriate dimensions
 - ie width x length x color channels
- Conv2D layer type does the actual convolution on a 2D image
 - Conv1D and Conv3D also available – doesn't have to be image data
- MaxPooling2D layers can be used to reduce a 2D layer down by taking the maximum value in a given block
- Flatten layers will convert the 2D layer to a 1D layer for passing into a flat hidden layer of neurons
- Typical usage:
 - Conv2D -> MaxPooling2D -> Dropout -> Flatten -> Dense -> Dropout -> Softmax

CNN's are hard

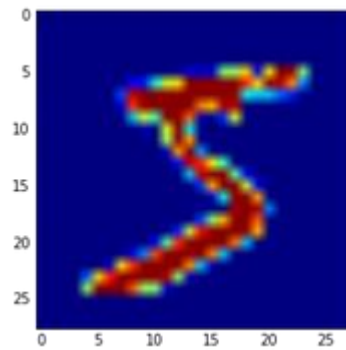
- Very resource-intensive (CPU, GPU, and RAM)
- Lots of hyperparameters
 - Kernel sizes, many layers with different numbers of units, amount of pooling... in addition to the usual stuff like number of layers, choice of optimizer
- Getting the training data is often the hardest part! (As well as storing and accessing it)



Specialized CNN architectures

- Defines specific arrangement of layers, padding, and hyperparameters
- LeNet-5
 - Good for handwriting recognition
- AlexNet
 - Image classification, deeper than LeNet
- GoogLeNet
 - Even deeper, but with better performance
 - Introduces *inception modules* (groups of convolution layers)
- ResNet (Residual Network)
 - Even deeper – maintains performance via *skip connections*.

Let's try it out



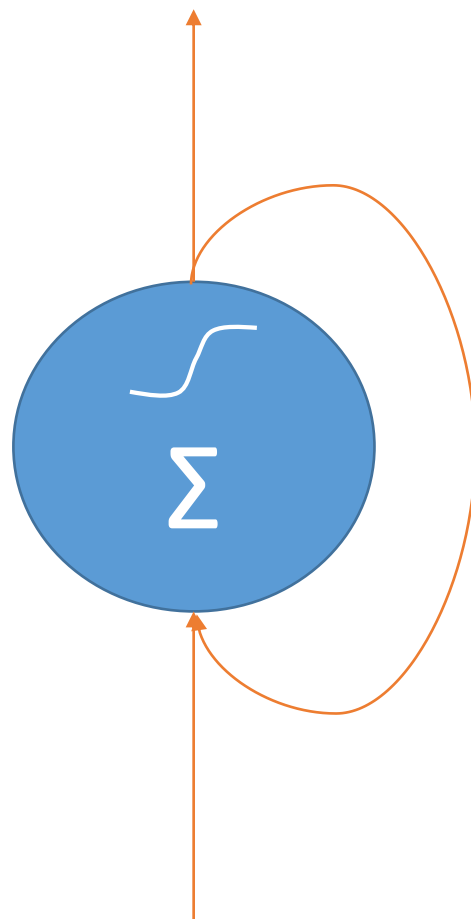
Recurrent Neural Networks

RNN's: what are they for?

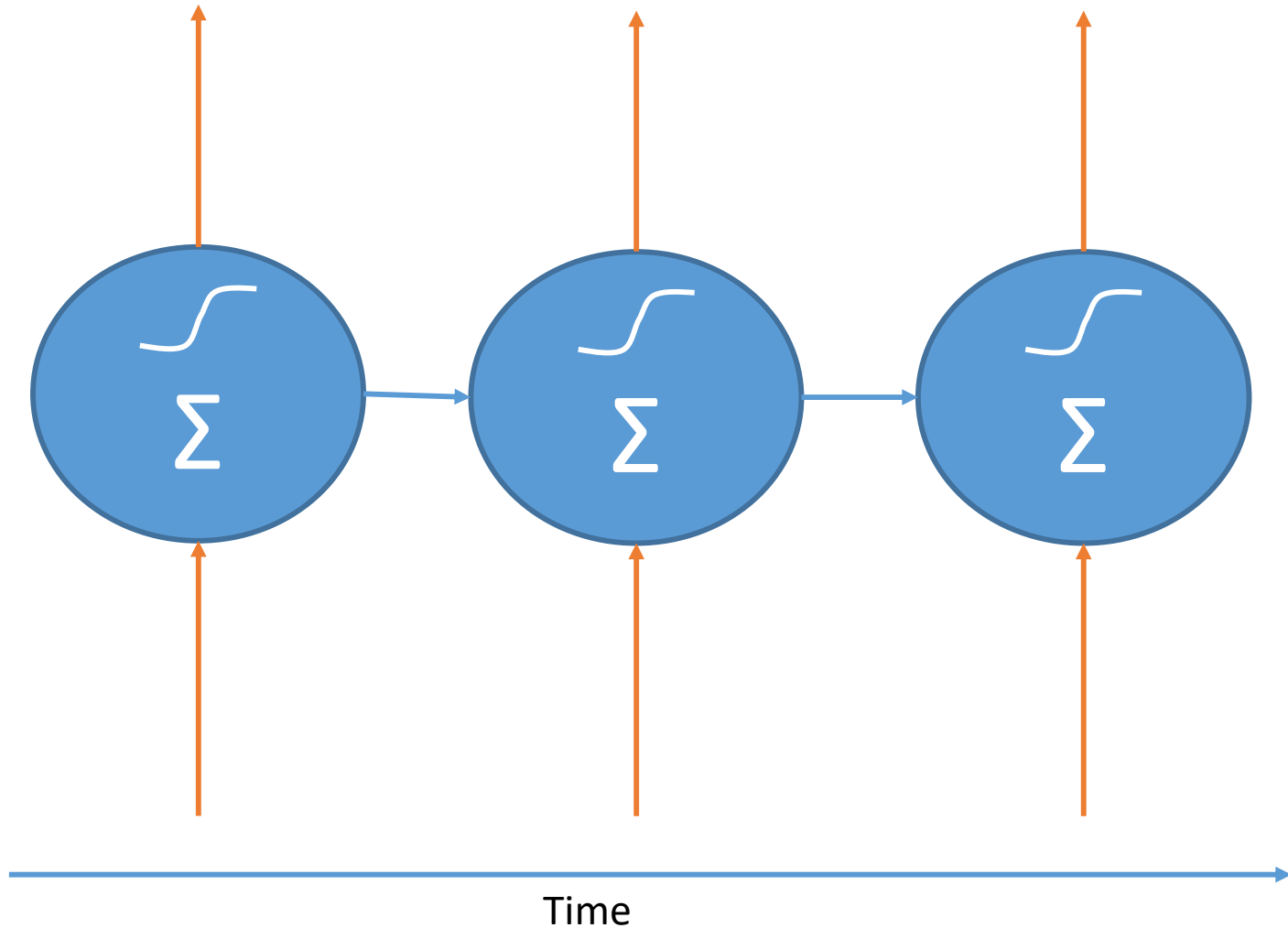
- Time-series data
 - When you want to predict future behavior based on past behavior
 - Web logs, sensor logs, stock trades
 - Where to drive your self-driving car based on past trajectories
- Data that consists of sequences of arbitrary length
 - Machine translation
 - Image captions
 - Machine-generated music



A Recurrent Neuron

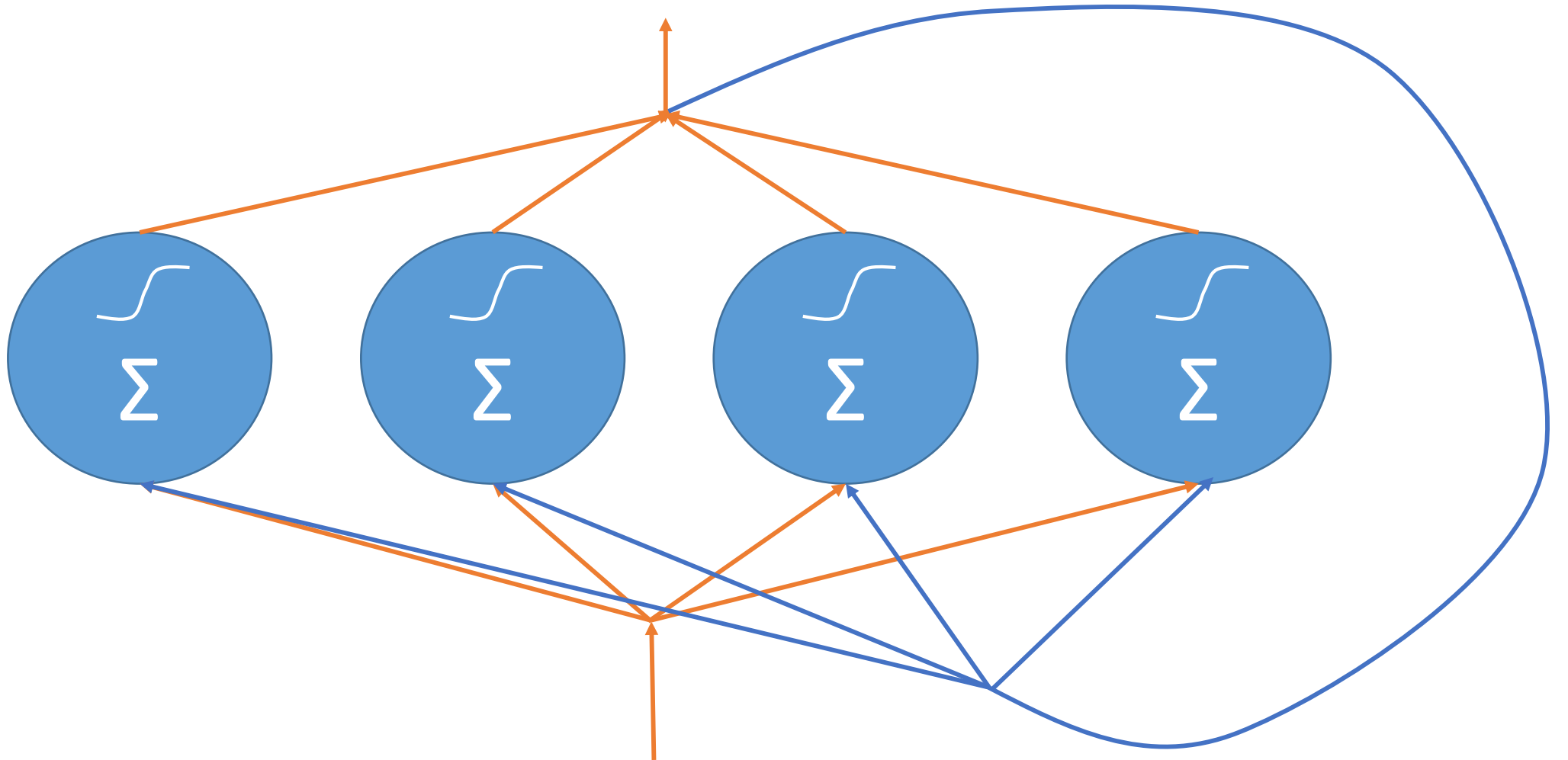


Another Way to Look At It



A "Memory Cell"

A Layer of Recurrent Neurons



RNN Topologies

- Sequence to sequence
 - i.e., predict stock prices based on series of historical data
- Sequence to vector
 - i.e., words in a sentence to sentiment
- Vector to sequence
 - i.e., create captions from an image
- Encoder -> Decoder
 - Sequence -> vector -> sequence
 - i.e., machine translation

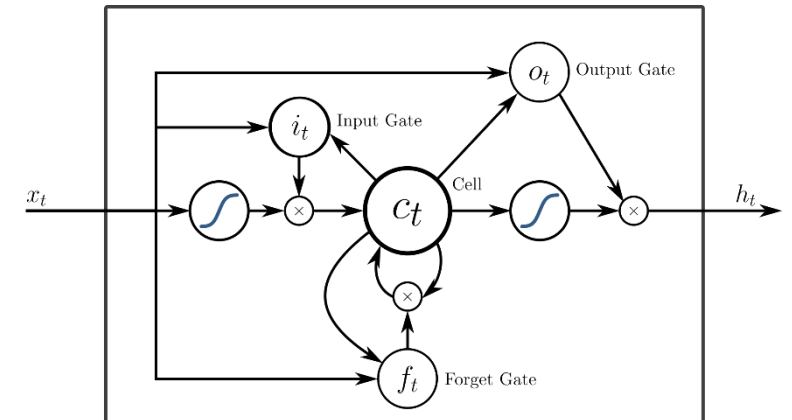


Training RNN's

- Backpropagation through time
 - Just like backpropagation on MLP's, but applied to each time step.
- All those time steps add up fast
 - Ends up looking like a really, really deep neural network.
 - Can limit backpropagation to a limited number of time steps (truncated backpropagation through time)

Training RNN's

- State from earlier time steps get diluted over time
 - This can be a problem, for example when learning sentence structures
- LSTM Cell
 - Long Short-Term Memory Cell
 - Maintains separate short-term and long-term states
- GRU Cell
 - Gated Recurrent Unit
 - Simplified LSTM Cell that performs about as well



Training RNN's

- It's really hard
 - Very sensitive to topologies, choice of hyperparameters
 - Very resource intensive
 - A wrong choice can lead to a RNN that doesn't converge at all.



Let's run an example.



The Ethics of Deep Learning

Types of errors

- Accuracy doesn't tell the whole story
- Type 1: False positive
 - Unnecessary surgery
 - Slam on the brakes for no reason
- Type 2: False negative
 - Untreated conditions
 - You crash into the car in front of you
- Think about the ramifications of different types of errors from your model, tune it accordingly



Hidden biases

- Just because your model isn't human doesn't mean it's inherently fair
- Example: train a model on what sort of job applicants get hired, use it to screen resumes
 - Past biases toward gender / age / race will be reflected in your model, because it was reflected in the data you trained the model with.



Is it really better than a human?

- Don't oversell the capabilities of an algorithm in your excitement
- Example: medical diagnostics that are almost, but not quite, as good as a human doctor
- Another example: self-driving cars that can kill people



Unintended applications of your research

- Gather 'round the fire while Uncle Frank tells you a story.



Learning More about Deep Learning