Tutorial on Neural Networks *COMP 550*

Kian Kenyon-Dean, M. Sc. Computer Science
https://kiankd.github.io
kian.kenyon-dean@mail.mcgill.ca

Objectives of this Tutorial

After attending & reviewing your notes for this tutorial, you should know:

1. **Why** we use neural networks
2. The differences between the following **fundamental components** of neural networks: *learning rate*, *feeding-forward*, *backpropagation*, *gradient descent*, *loss function*, *input layer*, *hidden layer*, *output layer*, *activation functions*, *softmax*
3. Know the different purposes of & basic differences between: *feed-forward neural nets (FFNNs)*, *recurrent neural nets (RNNs)*, *long short-term memory neural nets (LSTMs)*

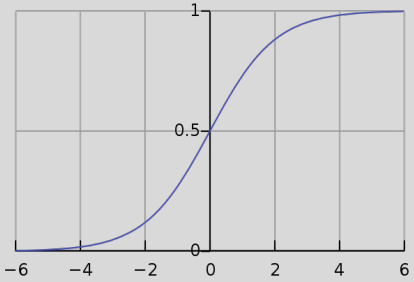
Preliminary: the Sigmoid (logistic) Function

- **What is the primary purpose of using the sigmoid function?**
- **How is it used in classification?**

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^{x}}{e^{x} + 1}$$

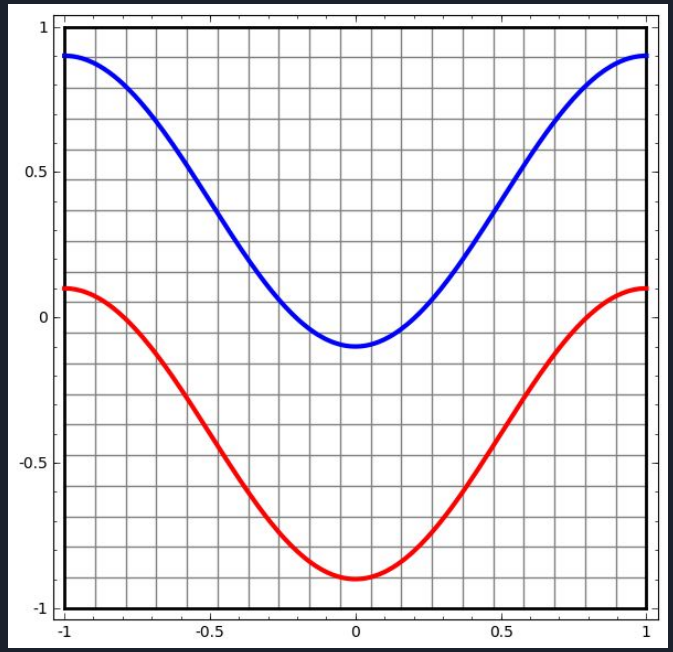
Preliminary: the Sigmoid (logistic) Function

- **What is the primary purpose of using the sigmoid function?**
  - It bounds its inputs between 0 and 1
- **How is it used in classification?**
  - For binary classification, we now can say $\sigma(f(x)) = 0 \Rightarrow$ class 0, $\sigma(f(x)) = 1 \Rightarrow$ class 1
    - Where we learn parameters of the function $f(x) = w^T x + b$ using gradient descent (this is logistic regression)



$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

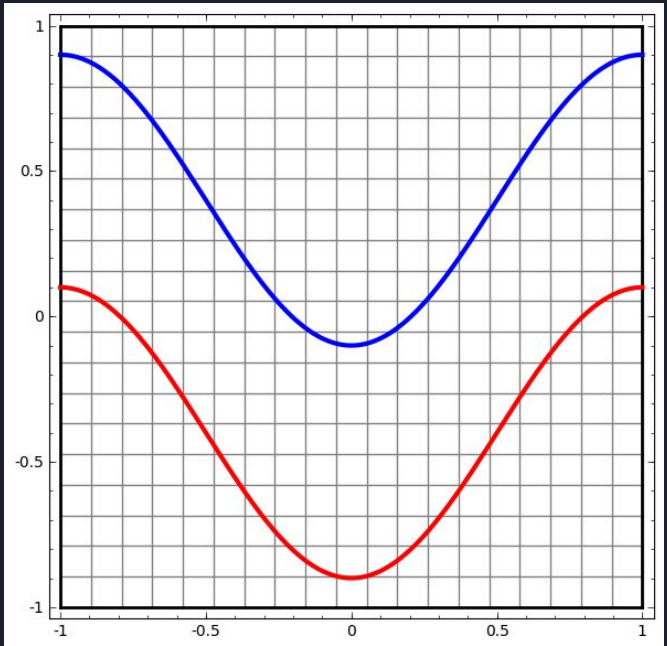
Reasons for using Neural Networks: *A Motivating Example*



Are data points sampled along the red and blue lines ***linearly separable****?*

E.g., can we learn a weight matrix **W** *for a function* f *that can predict the color of a point,* $x = (x_1, x_2)$*?*

Reasons for using Neural Networks: *A Motivating Example*

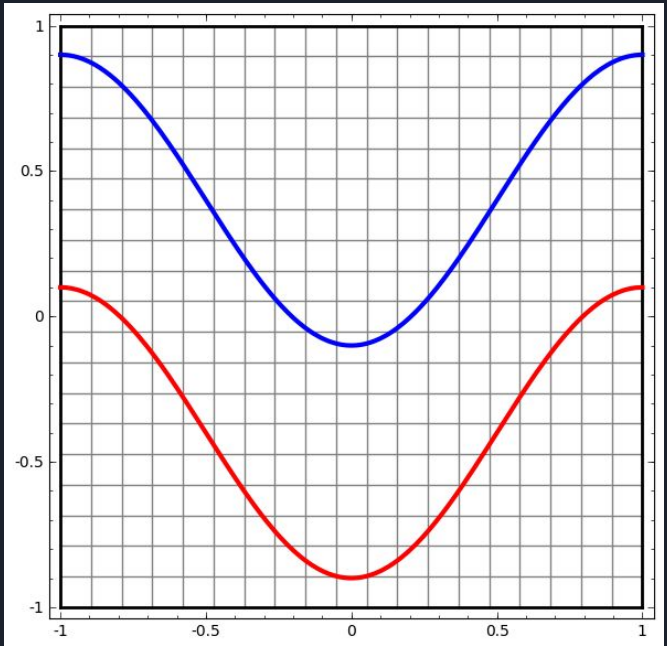


Are data points sampled along the red and blue lines ***linearly separable?***

*E.g., can we learn a linear function **f** in order to predict the color of a point, x = (x_1, x_2)?*

$$f(x) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$
$$= \sigma(w_1x_1 + w_2x_2 + b)$$
$$= \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + b)}}$$

Reasons for using Neural Networks: *A Motivating Example*

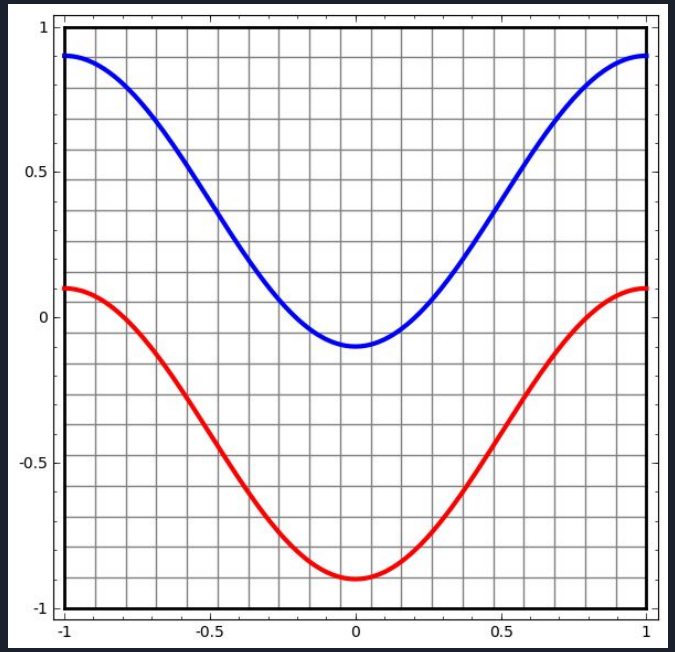


Are data points sampled along the red and blue lines ***linearly separable?***

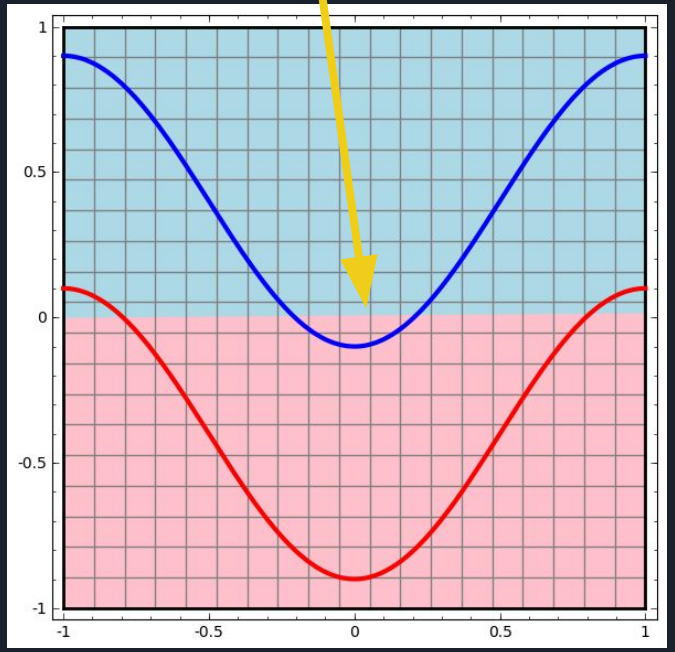
E.g., can we learn a linear function ***f*** *in order to predict the color of a point,* $x = (x_1, x_2)$*?*

Answer: no!

Reasons for using Neural Networks:
A Motivating Example



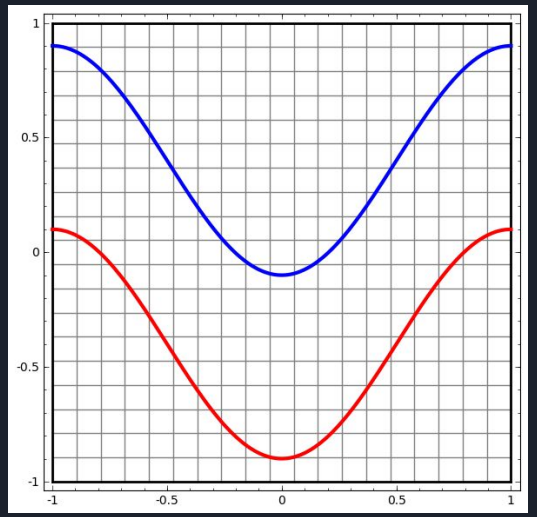
The best line that can be drawn to separate the data! Not good enough!

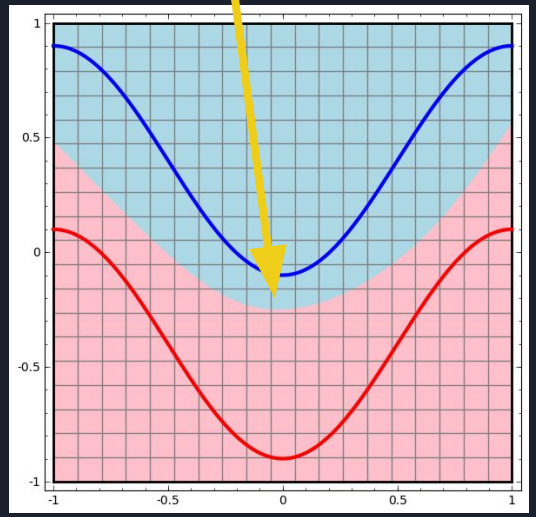


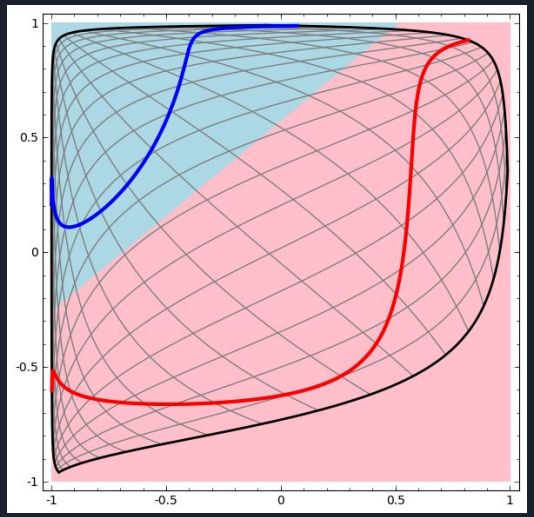
Reasons for using Neural Networks: *A Motivating Example*

Neural networks perform *non-linear* separation in *the input feature space*

With hidden layers, learns to project to a new, linearly separable space!







Source: Christopher Olah's blog, check it out! Lots of great stuff. *http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/*

To brush up on linear algebra, check out 3Blue1Brown's series!

Basics of Neural Networks: Overall Problem

Given:

- Input samples with feature vectors **x**
- Training samples with labels **y**
- Goal: train a classifier. *(Note that we can also use NNs for regression!)*

The model is a function **F**(**x**) such that, given *K* classes (1, …, K), our model predicts what class the sample **x** belongs to.

Running example:

We have a very simple problem, given a word with some feature vector **x**, predict if it is a **noun** or not!

Objective: If **x** is a **noun**, predict **1**, else predict **0.**

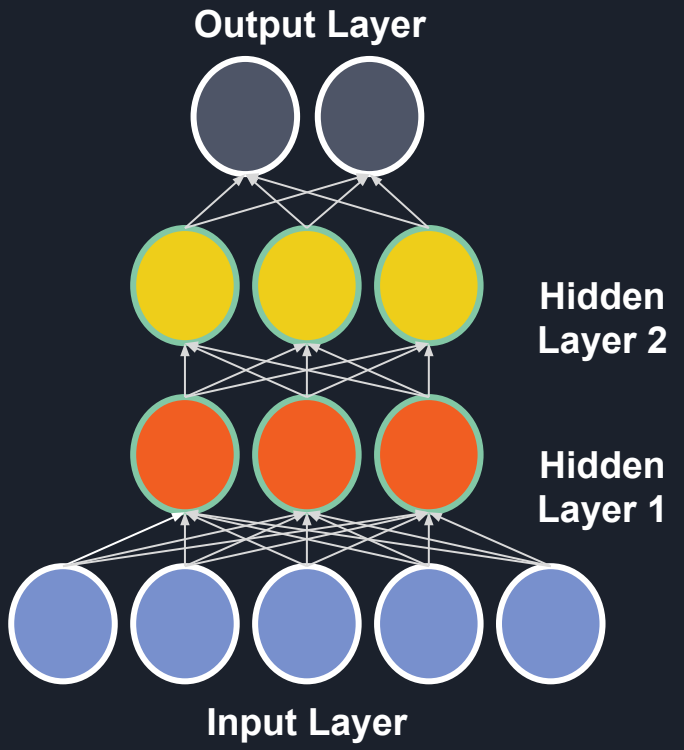
Say, our training set is X = [‘run’, ‘frog’]

- If **F**(‘run’) = 0, and **F**(‘frog’) = 1, we’re done; our model is great! Otherwise, we need to learn!

Basics of Neural Networks: Hidden Layers

We are given some input samples with feature vectors **x**, and labels **y**, so that we can train a classifier: a **feed-forward neural network**.

- *Hidden parameters: activation function, number of hidden layers, dimensionalities.*
- Hidden layer *parameters* are a **design choice**, determined with cross-validation.
- They allow you to do **non-linear** transformations, they perform **abstraction** over the inputs.

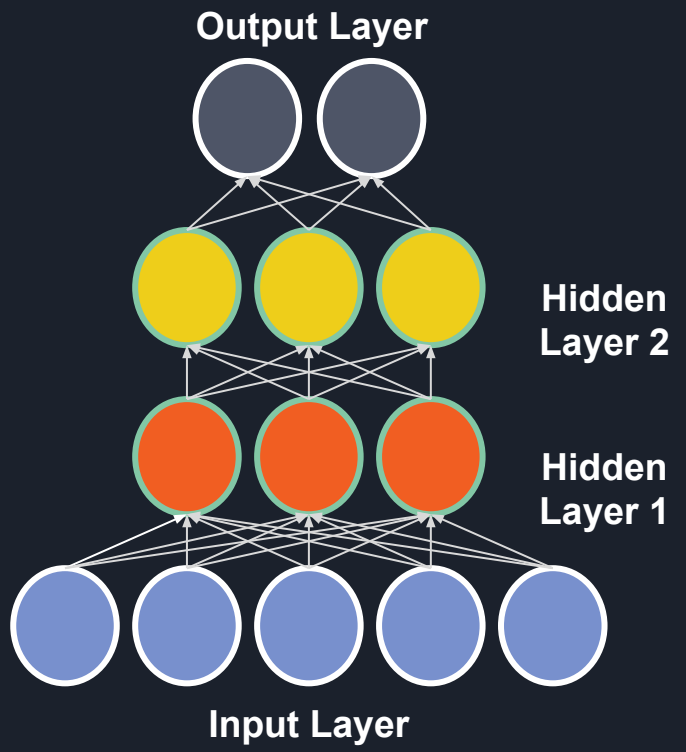


Basics of Neural Networks: Hidden Layers

We are given some input samples with feature vectors **x**, and labels **y**, so that we can train a classifier: a **feed-forward neural network**.

The neural model is defined as:

$$
\begin{aligned}
F(x) &= \text{softmax}(h_2(h_1(\mathbf{x}))) \\
h_2(h) &= a(\mathbf{h}\mathbf{W}_2 + \mathbf{b}_2) \\
h_1(x) &= a(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)
\end{aligned}
$$

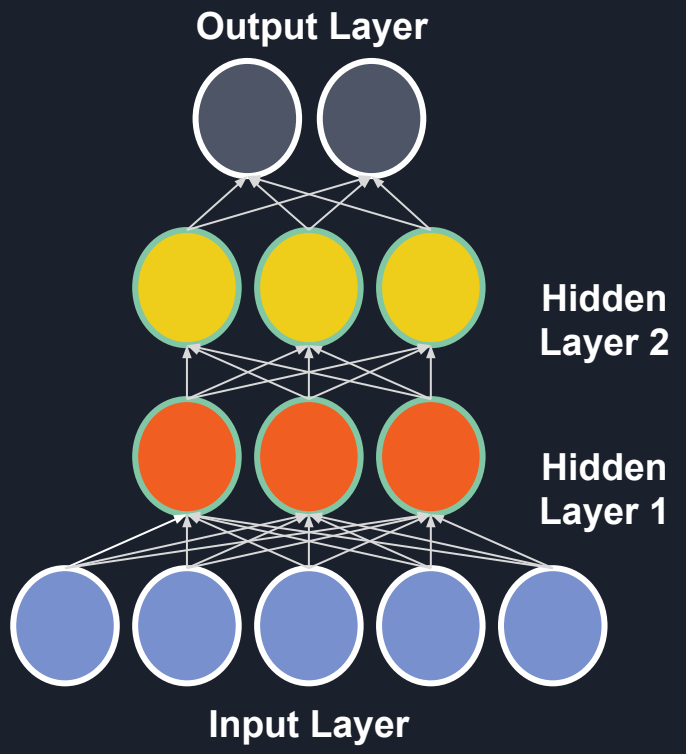


Basics of Neural Networks: Hidden Layers

Design choices:

- Choose number of hidden layers (two are presented here as an example)
- In theory, two should be able to approximate any real-world function, but sometimes deeper models are better (but don't get crazy!)

$$
\begin{aligned}
F(x) &= \text{softmax}(h_2(h_1(\mathbf{x}))) \\
h_2(h) &= a(\mathbf{h}\mathbf{W}_2 + \mathbf{b}_2) \\
h_1(x) &= a(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)
\end{aligned}
$$



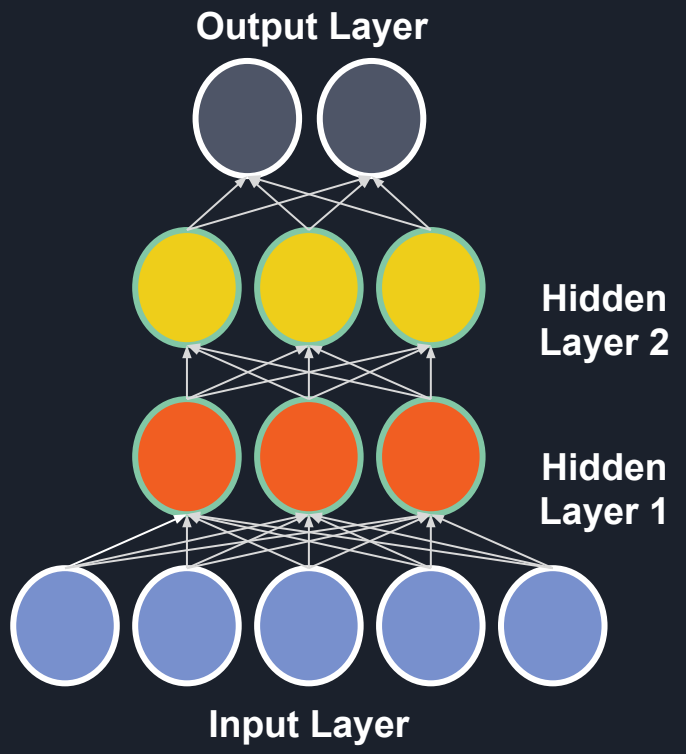
Basics of Neural Networks: Hidden Layers

Design choices:

- Dimensionality of hidden layers. We have fixed length feature vectors ***x*** of size *m*, so $\mathbf{W}_1$ is a *m* by h_1 matrix - you decide h_1!
- $\mathbf{W}_2$ is a h_1 by h_2 matrix - you decide h_2!
- Softmax also has a linear transformation, h_2 by *K* (for *K-class* classification problem)

Basically, this is logistic regression in the bent hidden space! Finding the linear separation in the nonlinear space.

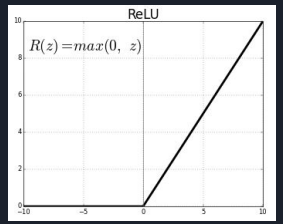
$$F(x) = \text{softmax}(h_2(h_1(\mathbf{x})))$$
$$h_2(h) = a(\mathbf{h}\mathbf{W}_2 + \mathbf{b}_2)$$
$$h_1(x) = a(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)$$



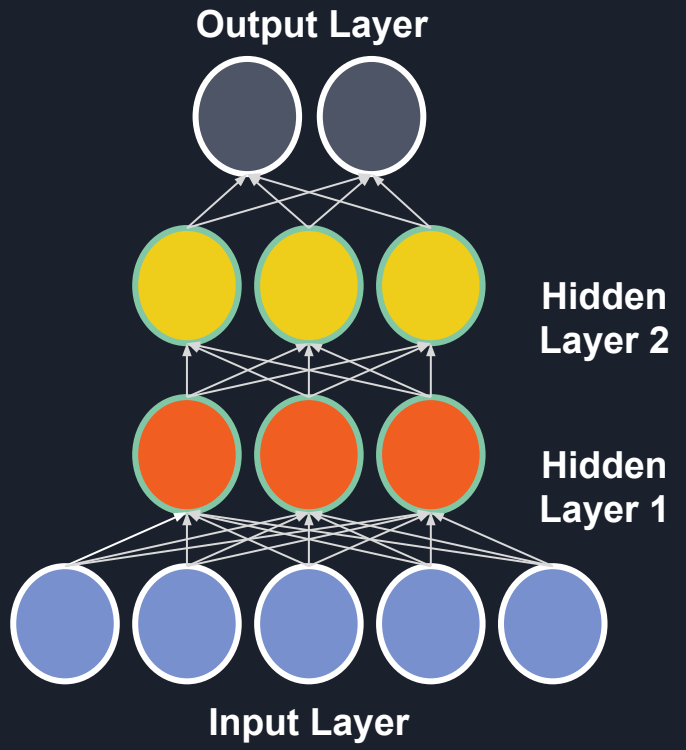
Basics of Neural Networks: Hidden Layers

Design choices:

- The activation function *a* - what **nonlinear** function should it be?
  - Sigmoid, tanh, ReLU, LeakyReLU, or one of many others??
  - Most of the time, ReLU!
    - *a(z) = max(0, z)*



$$F(x) = \text{softmax}(h_2(h_1(\mathbf{x})))$$
$$h_2(h) = a(\mathbf{h}\mathbf{W}_2 + \mathbf{b}_2)$$
$$h_1(x) = a(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)$$



Basics of Neural Networks: Hidden Layers

Are you paying attention?

- Let's use 300-dimensional "word embeddings" as features in our problem.
- How many learnable parameters will the following feed-forward neural net have?
  - 300-dimensional input features
  - Two hidden layers, each with 64 neurons (e.g., 64-dimensions)
  - 2-class classification problem
  - ReLU activations

Running example:

We have a very simple problem, given a word with some feature vector **x**, predict if it is a **noun** or not!

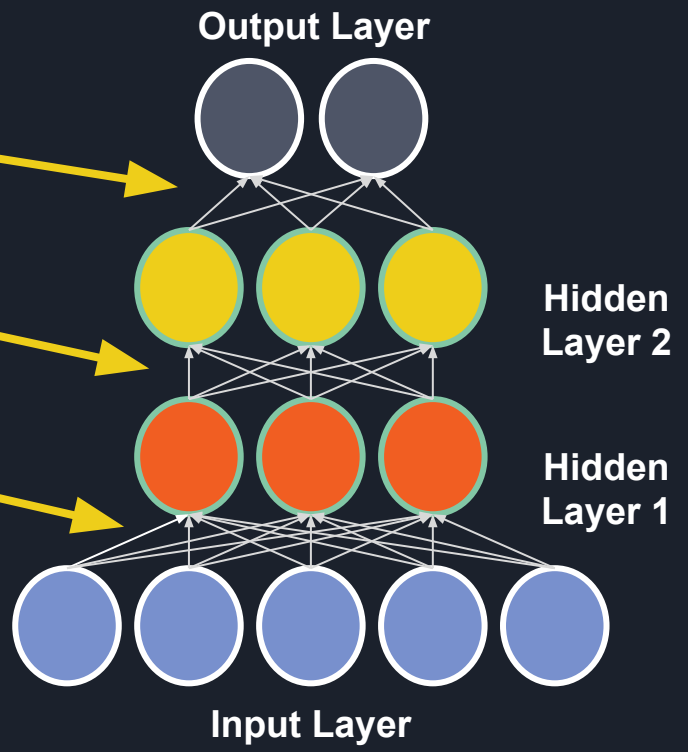
Objective: If **x** is a **noun**, predict **1**, else predict **0.**

Basics of Neural Networks: Hidden Layers

1. 64 hidden layer 2, 2 output classes ⇒ Weight matrix $\mathbf{W}_3$ that is 64 x 2, and a bias vector $\mathbf{b}_3$ that is 1 x 2.
2. 64 hidden layer 1, 64 hidden layer 2 ⇒ Weight matrix $\mathbf{W}_2$ that is 64 x 64, and a bias vector $\mathbf{b}_2$ that is 1 x 64.
3. 300 input, 64 hidden layer 1 ⇒ Weight matrix $\mathbf{W}_1$ that is 300 x 64, and a bias vector $\mathbf{b}_1$ that is 1 x 64.

Total: 23,554 weights to be learned!

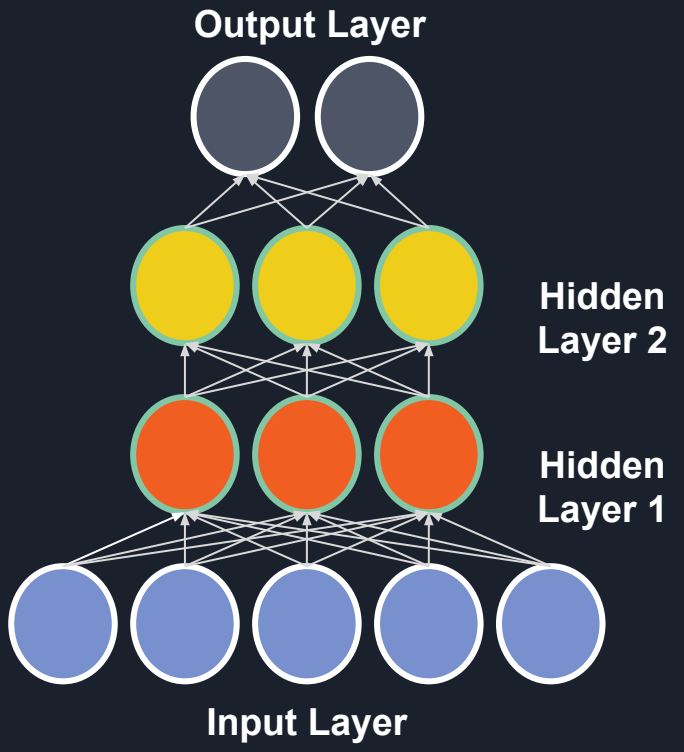
Meanwhile, logistic regression would have 301!



Basics of Neural Networks: **Learning**

How do you learn these weights? You need:

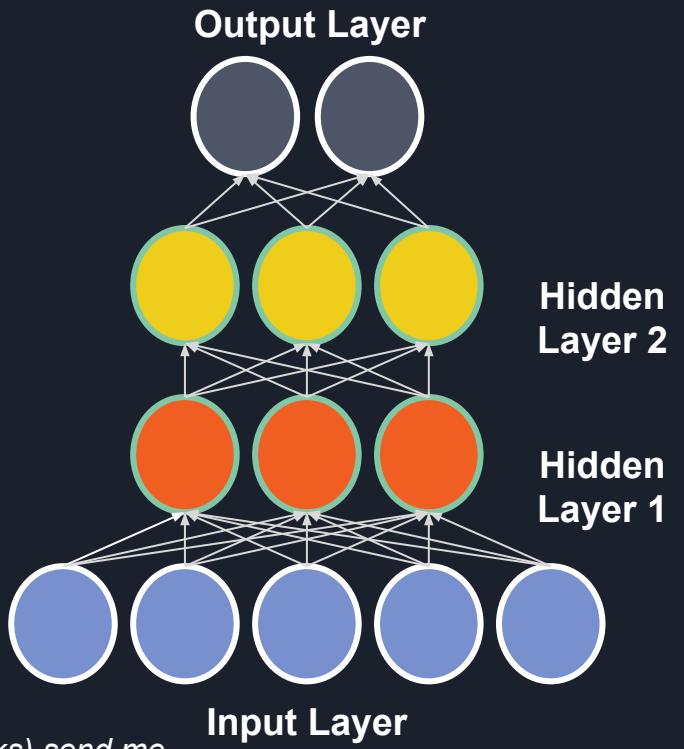
- **Loss function**: “how much error did I make?”
- **Gradient descent optimizer**: “let’s get a smaller error next time!”
- Then, **backpropagation**! “update weights: if they mostly caused the error then change them more, otherwise change them less!”



Basics of Neural Networks: **Learning**

Loss function: categorical cross-entropy for classification. Tell the model how much error was made, given it's predictions for some samples.

- Maximizes the **log likelihood** of the data
  - E.g., if we have to predict if a word is a **noun** or not, we have the objective of making every **noun** get a prediction of **1**, and every other word get a prediction of **0**.
  - If ***x*** is a **noun**, we want *f(**x**) = 1*, otherwise we want *f(**x**) = 0*



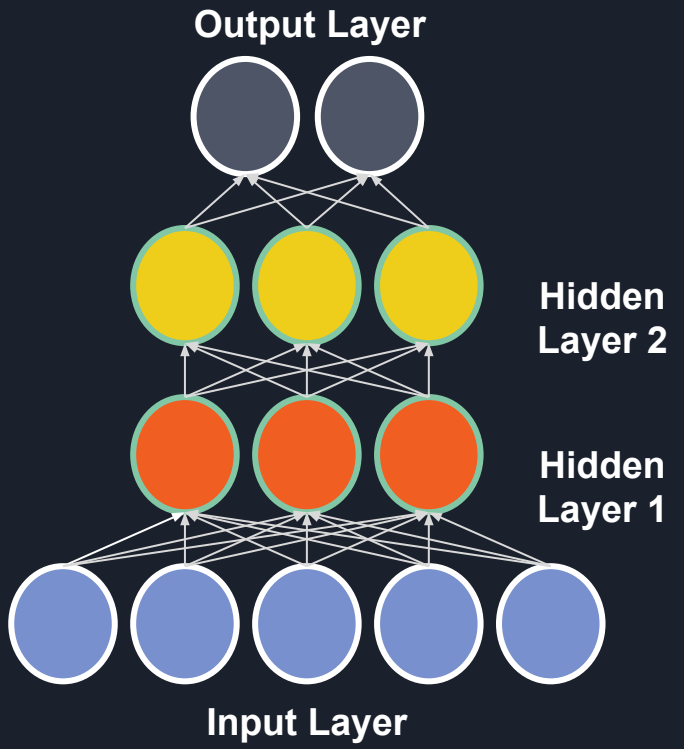
If you're interested in alternative loss functions for classification (or other tasks) send me an email and ask about my research. kian.kenyon-dean@mail.mcgill.ca

Basics of Neural Networks: **Learning**

Gradient Descent Optimizer: travel down the loss function to minimize it as rapidly as possible.

(a few important) Algorithms:

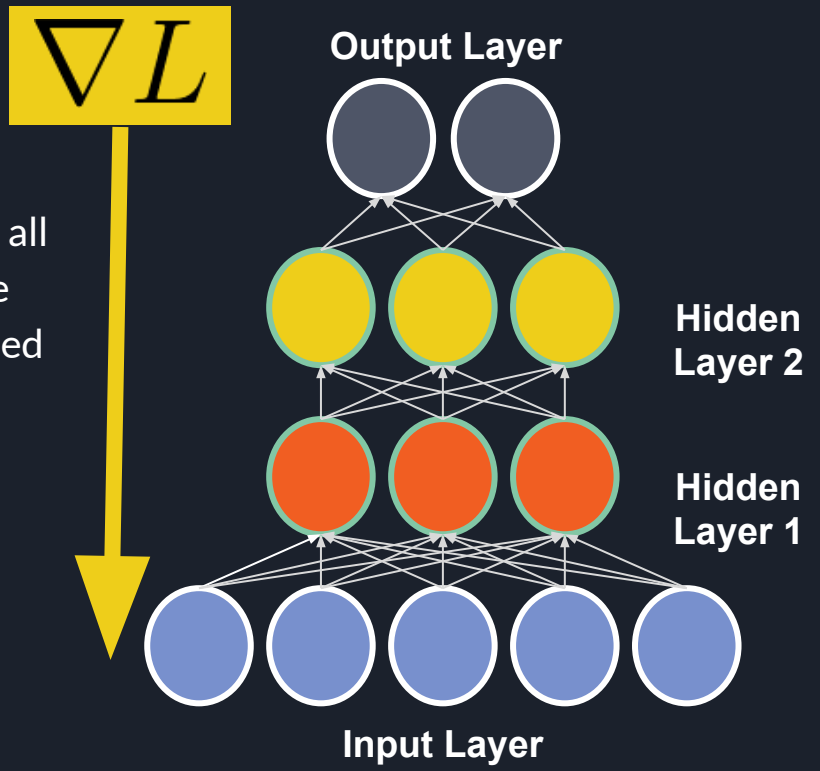
- Stochastic gradient descent [vanilla] (**SGD**) - *you should always use **momentum** with it*
- **LBFGS** - *objectively the best, but is impractical due to being highly memory intensive (as it has to approximate the Hessian matrix of the massive gradient)*
- **Adam** - *used all the time, very successful (should **always default to this one** unless you have a strong background in stochastic optimziation)*



See PyTorch's reference documentation for many more.

Basics of Neural Networks: **Learning**

Backpropagation: the algorithm that updates all of the weights in the network according to the error (computed by the loss function) attributed to each weight by the gradient descent optimizer



See PyTorch's reference documentation for many more.

Different types of Networks - *Motivation*

Suppose we want to label a sequence of things, like POS-tags for words:

Input: “The cat is walking to the sink”

- **The cat is walking to the sink .**
- ***DET NOUN VERB VERB ADP DET NOUN PUNCT***

Different types of Networks - *Motivation*

Suppose we want to label a sequence of things, like POS-tags for words:

Input: "The cat is walking to the sink"

- **The cat is walking to the sink .**
- ***DET NOUN VERB VERB ADP DET NOUN PUNCT***

Problem formulation (*Sequence Labelling*):

- Input: arbitrary-length sequence of words *(word embeddings)*
  - Note: each word is not **i.i.d.**, they are (partially) determined by the past (*and future!*)
- Output: a sequence of **discrete** labels for each word

See this interactive text analysis website for this (and many other examples of NLP solutions) in action!

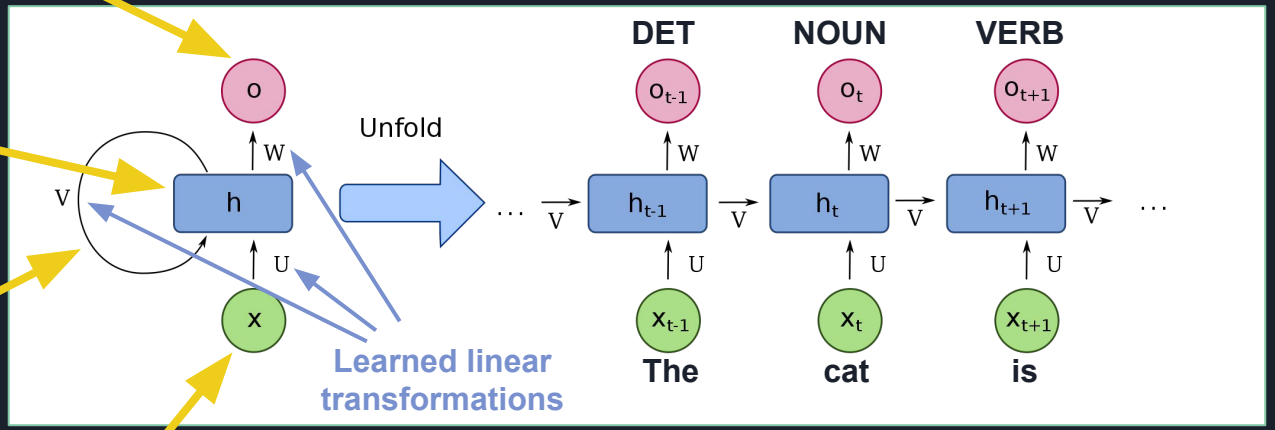
Recurrent Neural Network *(RNN)* for Sequence Labelling

Output label
(Probability dist.)

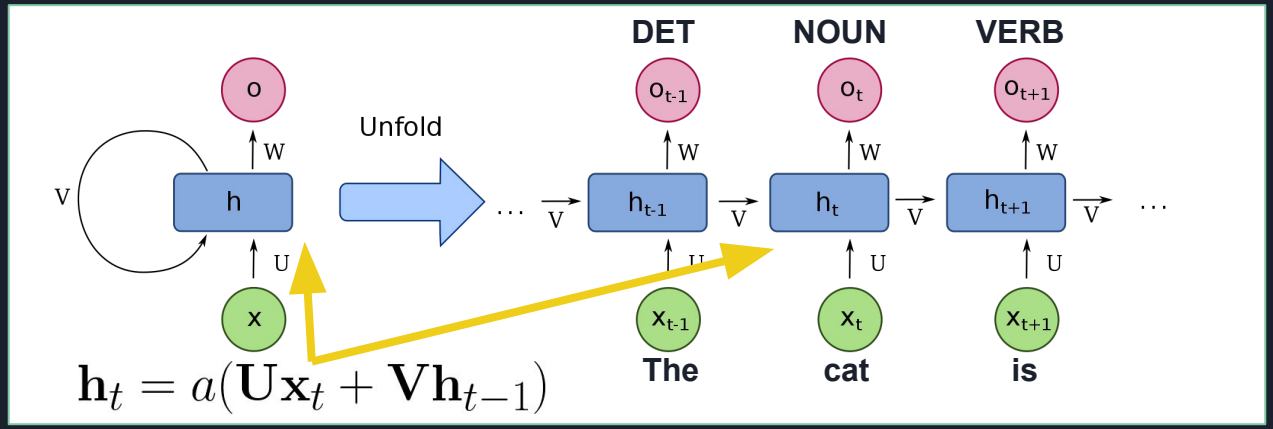
Hidden state
(can be multiple layers of depth)

Recurrence
(allows an arbitrary length sequence)

Input sample features
(e.g., word embedding)



Recurrent Neural Network *(RNN)* for Sequence Labelling



Learned Parameters

1. Input-to-hidden linear transformation, **U** [same as in FFNNs!]
2. *Hidden-to-hidden* linear transformation, **V** [**new part**, the recurrence!]
3. Hidden to output linear transformation, **W** [same as in FFNNs!]

Problems with **Recurrent Neural Networks**

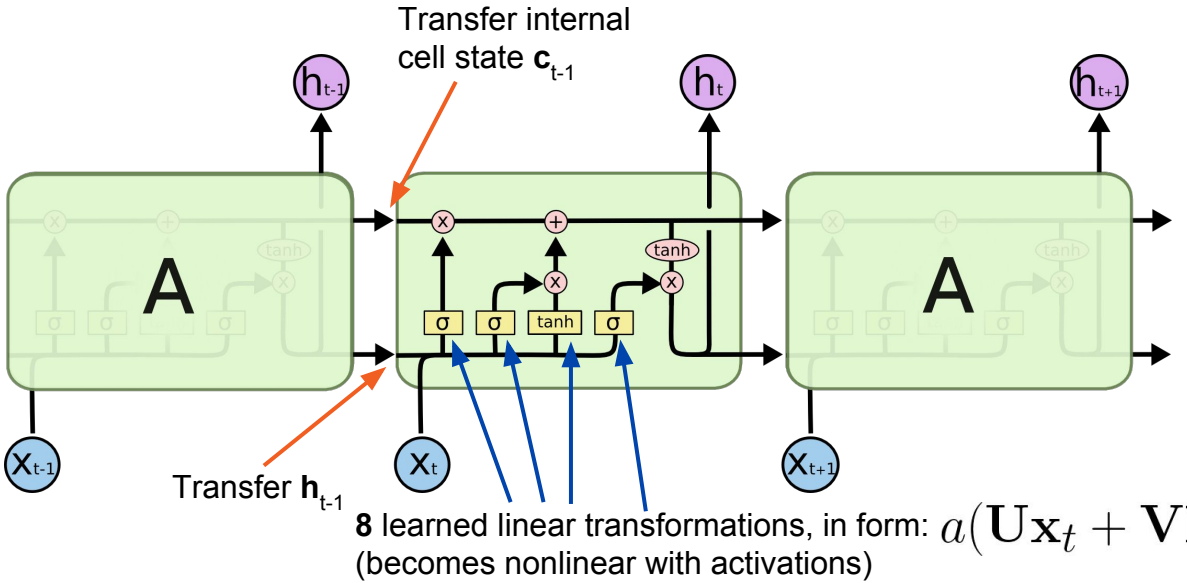
- On the surface, they seem great!
  - Take a series of inputs & hope that the model learns to use the **hidden state** in order to hold and sustain information over time.
- **Main difficulty**: they basically don't work!
  - Subject to the **vanishing/exploding gradient problem**
    - Due to the long sequence of recurrences, during backpropagation the gradient can either get very small (if norm < 1) or very large (if norm > 1) due to a huge sequence of multiplications.
  - Doesn't actually capture **long-term dependencies** (even though they should be able to, in theory)
    - The hidden state is not expressive enough to retain information from very early in the sequence - it will be lost over time

This would be very important for correct automatic translation!

Solution to the Problems of RNNs: Add **Long Short-Term Memory (LSTM)**

- Classic paper by Hochreiter & Schmidhuber in 1997 solved the previous problems of RNNs.
- They learn to **hold long-term dependencies** easily and "quickly" (compared to RNNs), and **do not suffer from the vanishing/exploding** gradient problem.
- Basic features:
  - Introduce **memory gates** & cell states within the hidden layer
  - Instead of learning *two* linear transformations (input→hidden, hidden→hidden, like in RNNs), they have to learn **8!**
    - LSTMs must learn 4 times as many parameters as an RNN!

Solution to the Problems of RNNs: Add **Long Short-Term Memory (LSTM)**



See Colah's blog post on LSTMs for a comprehensive understanding!

Summary of RNNs

- RNNs are used in order to *model sequential data*, where each element in the sequence is not i.i.d. (identically & independently drawn)
- RNNs can be used for **sequence labelling** (like the example with POS-tagging), but *can also be used for* **sequence classification**; these RNNs only have an output node *at the last time step*. E.g., predict the sentiment of the Tweet "I hate Doug Ford #hesbasicallyanazi" → **Negative** (predicted at *end*!)
- **Vanilla RNNs** suffer from the **vanishing/exploding gradient problem** & are **not** (typically) **used in practice**
- **LSTMS** are special types of RNNs that require 4 times as many parameters as a vanilla RNN, but **capture long-term dependencies** and can be **trained easily**

How to use Neural Nets in Practice

- Run using a **GPU** (graphics processor), which massively parallelize the many many matrix multiplications required to run one
- Use a **standard NN library**, makes life easy - I highly recommend PyTorch!
  - Supports using either GPU or CPU
  - Massive library on top of Python (2 or 3)
  - Use *Adam* for stochastic optimization
  - Use *ReLU* (or *LeakyReLU*) for hidden layer activation functions
  - Use *smaller numbers of neurons* (256, maybe 512) in hidden layers
- Count the number of parameters in your model and see if it seems crazy big!
- For **sequence classification** you should use a FFNN as a baseline (where the input is, for example, an average of all the word embeddings in the sequence) before jumping to an LSTM.
- When using RNNs, always use an LSTM (or a GRU). Also, look into *bi-directional* RNNs!