14. Language models and neural networks

Learning outcomes

- Define and give examples of n-grams
- Describe the difficulties associated with using standard linguistic theories in NLP and ML
- Describe neural networks
  - As basis function learners
  - As more complicated function learners
- Run and modify code that trains neural networks
- Describe the universal approximation theorem and how width and depth affect neural networks
- Discuss the semantics of neural networks

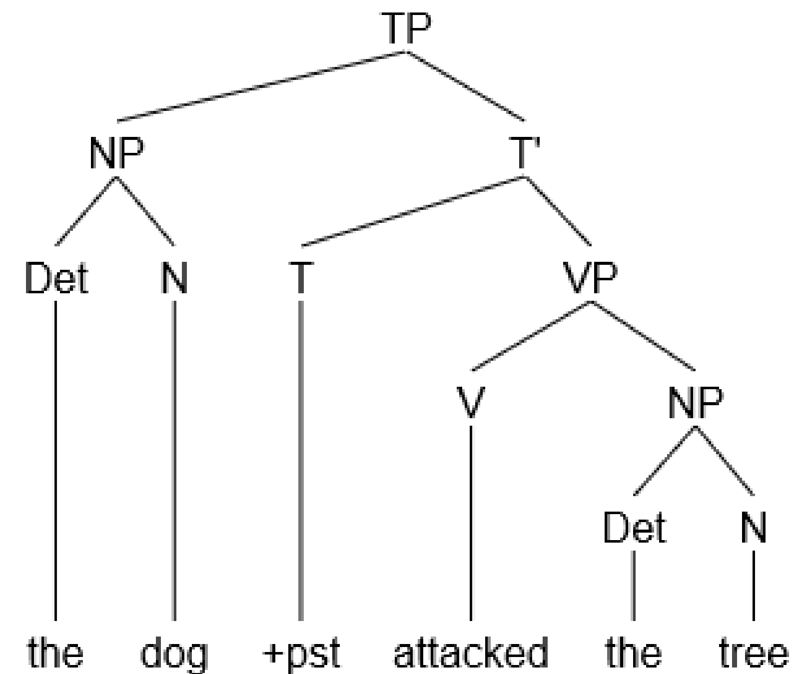
Caveats

- We are at a point now where we won't be introducing more math or statistics
  - Too complicated to summarize in a few slides
  - May still tweak some notation
- Neural networks often take a long time to run
- And, neural network design can take a lot of fiddling to even get running
- So, you won't be asked to write new networks
  - You just need to step through the code I have provided you

Language models

Traditional language models

- Linguistics has many language models
  - Phonology, syntax, semantics, etc.
- Some of these models are opposed to each other
  - E.g., generative grammar and construction grammar
- They are sophisticated and require expert knowledge



Formalizing traditional models

- A language model (grammar) must be **formalized** to use it on a computer
  - All assumptions must be laid out
  - There must be a mechanism to process data according to the grammar
- Not all models are equally well-formalized

Chomskyan phrase structure rules for basic English sentences:

TP: NP T VP

NP → (Det)* (AP)* N (PP)*

VP → (Adv) V (NP) (NP) (PP)*

AP → (Deg) A (PP)

PP → (Deg) P (NP)

Goldbergian transitive construction:

[TRANSITIVE] – [SBJ TRVERB OBJ]

Difficulties with formalization of linguistic theory

- Formalization requires collaboration between linguistics and someone working in computation (computer scientist, mathematician, machine learning expert, etc.)
  - Or someone who has expertise in both linguistics and computation
- Not all linguists want their theories formalized
- Not all linguists care about computational implementations of their theories
- Not all theories work equally well (or very well at all) for processing natural language computationally
  - "Every time I fire a linguist, the performance of the speech recognizer goes up" – Frederick Jelinek (1988)

What works in practice?

- If linguistic theories are hard to adapt or don't work, what is used instead?
- Statistical models of language use
- Cheap model: n-grams
  - A series of "n" words
  - Rarely go past 3 for many purposes
  - (Sometimes use 5 though...)

Sentence: "I ate some beans today"

1-gram (unigram):	"I"
2-gram (bigram):	"I ate"
3-gram (trigram):	"I ate some"
4-gram:	"I ate some beans"
etc.	

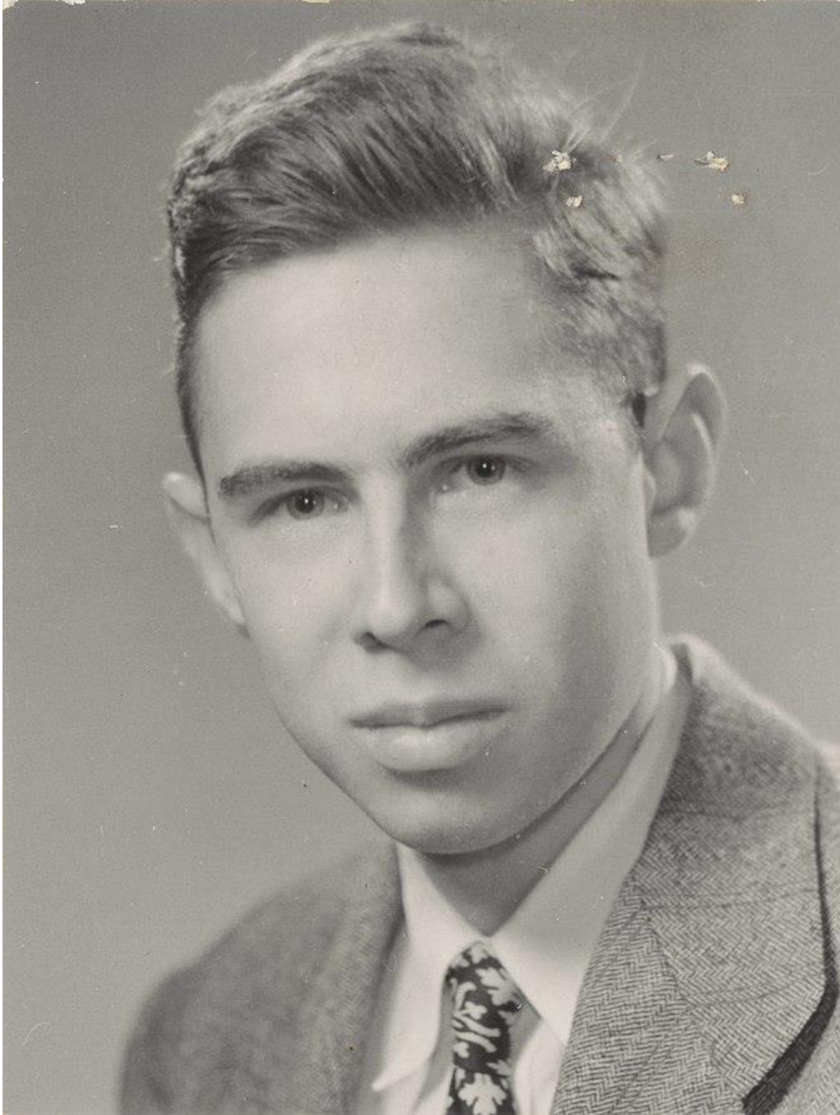
Relative frequency comes from n-grams

- As we have seen before, relative frequencies can be obtained from n-grams since n-grams only require counting
- N-grams can also be manipulated to get conditional frequency
  - Remember our lecture on p(other | to the)?
- As n gets larger, the probabilities get lower because there are fewer instances of an n-gram than its constituent parts
  - E.g., "the dog" is less frequent than "the" and "dog", which can occur in more contexts than just "the dog"
- But, larger n also means better models of the language

Are we stuck using just n-grams?

- N-grams do provide good performance for how simple they are
- We can see that a word's probability usually only depends on a small handful of prior words
  - Long-term grammatical dependencies are rare (in English…)
- However… A more modern language model is a **neural language model**
- Neural language models are based on **artificial neural networks**
  - We will get to the neural language model on Thursday
  - We must first go over neural nets in general

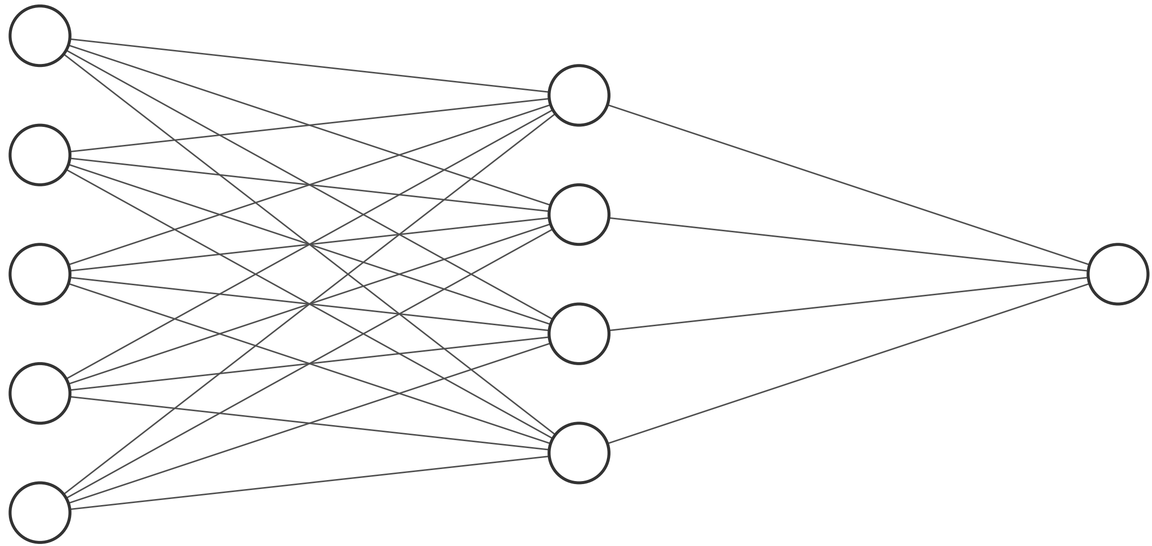
(Artificial) neural networks



History of artificial neural networks

- Artificial neural networks were inspired by how neurons fire in the brain
  - Dates back to Rosenblatt's work on modeling neurons in the 1950s
- We know now that real neurons work differently than neural nets
  - But ANNs are enormously useful and powerful

Image of Frank Rosenblatt from WeeKee under CC BY-SA 4.0



How do neural nets work?

- In a broad sense, they can be thought of passing information from one neuron to another
- Each successive layer contains more "abstract" features
- Ultimately, a neural network is a **function** that takes an input and produces an output

The implementation of a (not deep) neural net

- The modern neural network has a lot of parts
  - But we have seen all of them before in linear regression and logistic regression!
- Generally, there is
  - An input layer
  - Two sets of weights (matrices)
  - An output layer
  - Two sets of activation functions
- Between the two input and output there is a **"hidden" layer**

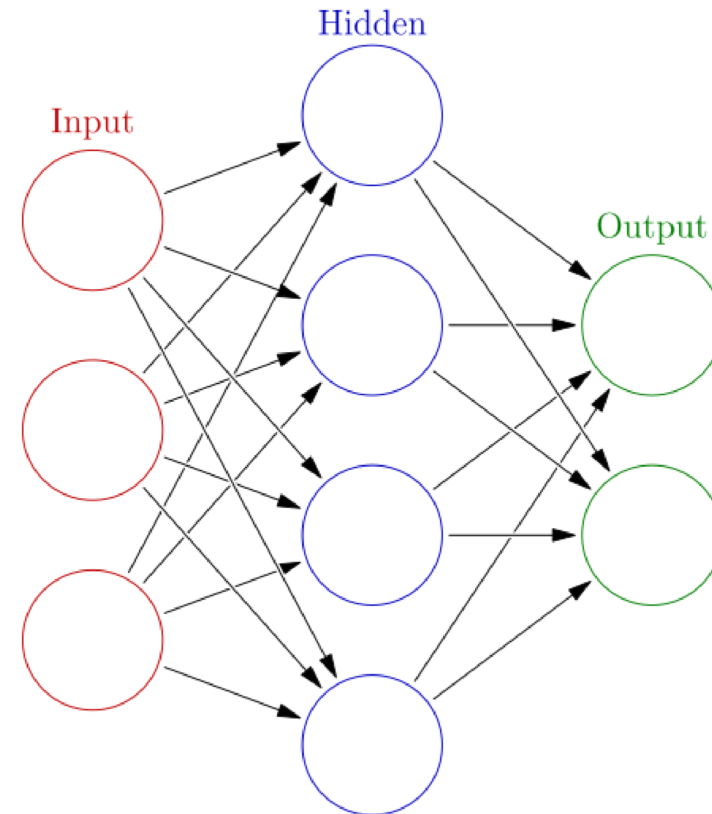


Image from Glosser.ca under CC BY-SA 3.0

Let's back up for a moment

- I want us to be able to start by thinking about neural networks in terms of what we covered last week
  - Linear regression, basis expansion, and logistic regression
- Let's assume that
  - We want to model a nonlinear function
  - But we don't know what the basis expansion should be to model it
- How can we determine the basis function for our regression?

Possible strategies for discovering the basis expansion

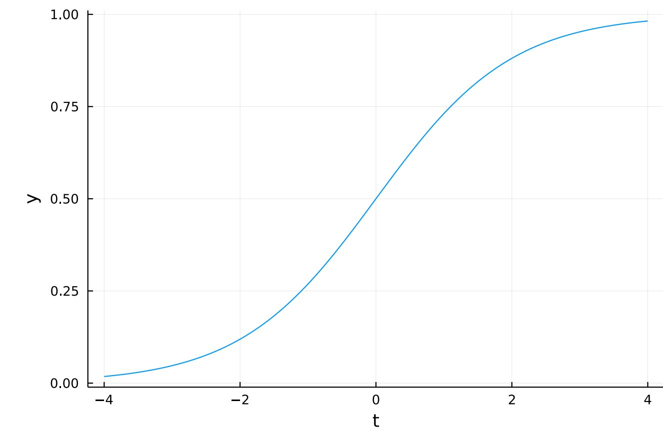
- We could randomly guess functions
  - This might take too long
- We could overspecify the expansion and just use 1000s of polynomial terms
  - We will likely overfit
- Or… We could **learn** the/a basis expansion that works optimally for our data

$$x \rightarrow \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \text{(call this } x'\text{)}$$

We then want to determine some coefficients b that solve the linear least squares problem for ${x'}^T b \approx y$

Choosing the expansion type

- We still need to choose some sort of function we will learn weights/coefficients for
  - Polynomials, exponentials, sinusoids?
- For the sake of familiarity, let's assume we want to add a lot of logistic sigmoid functions
  - Assumption: if you have "enough" sigmoids in your expansion, they will add "enough" wiggliness to match/approximate the function we want to model



Some notation work

- We need to set up our model
  - First we need to talk about some notation
- Logistic regression looked like $\sigma(\beta_1\chi_1 + \beta_2\chi_2)$
- Let's make a notational note that for some vector x,

$$\sigma(x) = \begin{bmatrix} \sigma(\chi_1) \\ \sigma(\chi_2) \\ \vdots \end{bmatrix}$$

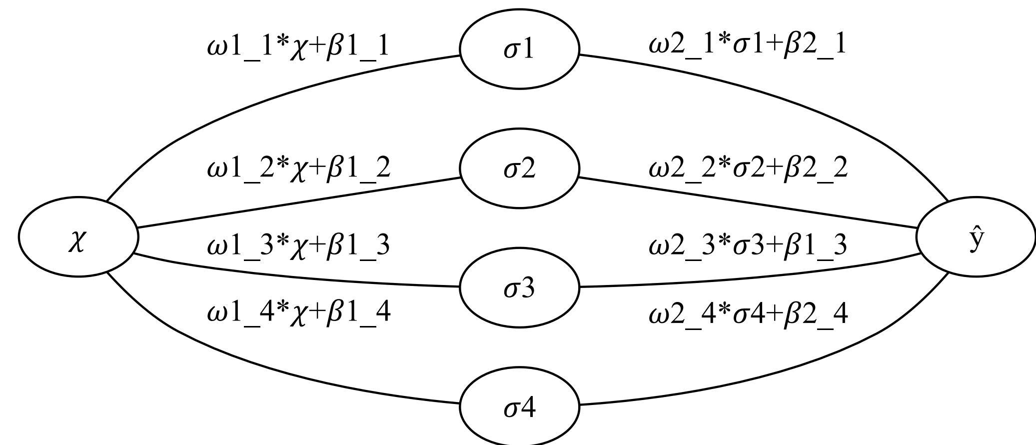
- Let's put our line slopes into a weight matrix W and our intercepts into a bias vector b
  - Our cast into the sigmoid bases can now be expressed $\sigma(Wx + b)$

Setting up our problem

- We have two overall steps we are working on
  1. Cast our input into the sigmoid bases
  2. Take those sigmoid bases and perform regression on them to our output
- So, we can say that our model is
  - $y \approx W_2\sigma(W_1x + b_1) + b_2$ or
  - $\hat{y} = W_2\sigma(W_1x + b_1) + b_2$, where $\hat{y}$ is our best approximation of y

Visualizing our network

- We can make different columns in a graph
- Each connection represents some part of $Wx + b$ type of operation
- This type of visualization and connection is why these tools are called networks

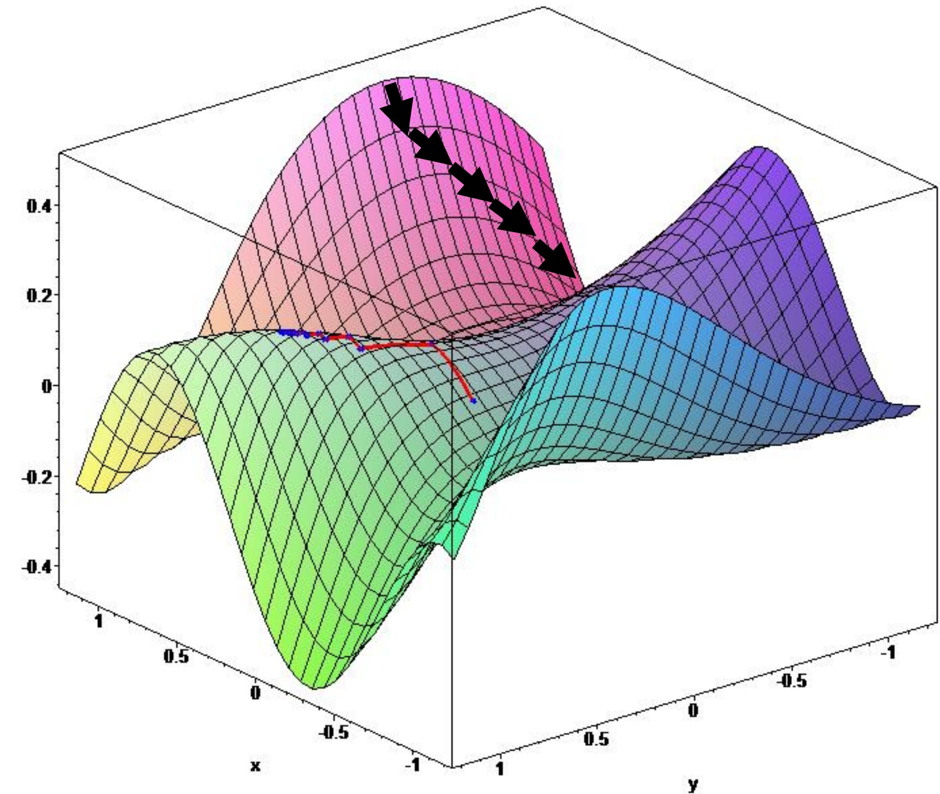


Learning W_1, b_1, W_2, b_2

- We haven't said what number are inside of our weights and biases
  - So we still can't actually compute our basis expansion!
- We *could* randomly guess values until we have low loss
  - But this is unlikely to work
- Instead we start with random values and then perform **numerical optimization** to minimize our prediction error (make $\hat{y}$ as close to y as possible)

Numerical optimization: Gradient descent

- The optimization technique we use is called **gradient descent**
- It is a physics and calculus simulation
- We determine how our performance loss changes for different weight and bias values
  - This is called the **gradient**
- We can tune the weights and biases a bit toward minimizing the loss
  - This is the **descent**
- It is like rolling a ball down a hill!



Training the network

- For each data point in our data set:
  1. Calculate our predicted value from current weights and biases
  2. Determine how far away the prediction was from the true value (compute the loss function)
  3. Determine how to decrease the loss and adjust the weights and biases accordingly
  4. Repeat for next data point
- Each run through the entire data set is called an **epoch**
- Train for as many epochs as needed until the loss is minimized

When to stop training: Theoretical advice

- Stop when the loss is minimized!
  - How do you know when the loss is minimized?
- When the weights and biases converge!
  - When do the weights and biases converge?
- They converge when the adjustments to them are small?
  - How small is small?
- !!!??? (This process can go on and on and on)

When to stop training: Practical advice

- Establish a validation subset of your training data that won't be trained on directly but will be evaluated at the end of every epoch
- Stop training when your validation loss has not decreased for some amount of time
- Or, train for many epochs and choose the best result on your validation set

5 minute break

Training neural networks: Keras and TensorFlow

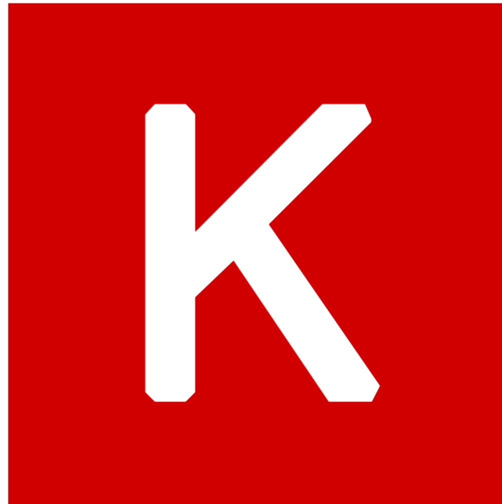
Neural network (and general ML) packages

- There is too much math involved with neural nets (and ML) for one person to write everything *and* have it run fast
- Dedicated teams from companies like Facebook and Google write these packages for us
- Many of these packages involve writing weird looking code
  - Often feels like writing another programming language in Python

```
# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Screen shot from https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py



Keras and TensorFlow

- TensorFlow is Google's machine learning platform
  - Using it directly feels very arcane
- Keras is an abstraction that allows for *much* easier use of TensorFlow for neural networks
  - Used to support lots of other backends until Google got stricter with it
  - Developed by François Chollet, a Google engineer
  - Eventually integrated into TensorFlow itself

Versions of Keras

- There are two separate "versions" of Keras
- Keras (standalone name) is the original package
  - It is still actively developed, but newer versions only support TensorFlow
- tf.keras is Keras bundled into TensorFlow
  - Conveniently installed when you install TensorFlow
- They are basically equivalent, but we will stick to tf.keras so you only have to install one package

A toy example

- Let's train a basic neural network that performs the basis expansion we discussed previously
  - Let's use the function $f(x) = 3x^4 + 2x^2 + x + 4$
  - Pretend we don't know this is actually the function, though…
- Let's generate data to train on

```
from tensorflow import keras

import numpy as np

import matplotlib.pyplot as plt

def f(x):

    return 3*(x-4)**3 + 2*x**2 + x + 4

x_in = np.linspace(0, 10, num=5000)

x = x_in.reshape(-1, 1)

y = f(x_in)

plt.plot(x_in, y)
```

Setting up the model

- `keras.Sequential()` creates a model that you can sequentially add layers to
- If we use `keras.layers.Dense`, we can create dense/fully-connected layers
  - All inputs connected to the layer, like with a matrix
- We will use 64 sigmoid units in the layer

```
m = keras.Sequential()

m.add(keras.layers.Dense(64,
activation='sigmoid', input_dim=1))

m.add(keras.layers.Dense(1,
activation='linear'))
```

Fit the model

- We first tell Keras to build the model with the compile function
  - Here is where we tell Keras what loss function to optimize and how to optimize it
  - adam is an efficient gradient descent algorithm
- When fitting, verbose level 0 has no output, verbose level 1 has progress bars in each epoch, and verbose level 2 summarizes each epoch

```
m.compile(optimizer='adam',
loss='mse')

h = m.fit(x, y, epochs=100,
verbose=2)
```

Plot the results

- I know we haven't gone over plotting yet, but these functions will help us understand our results
- The first plot is how our loss function decreases with each epoch
- The second plot is how our approximation looks after all of our training

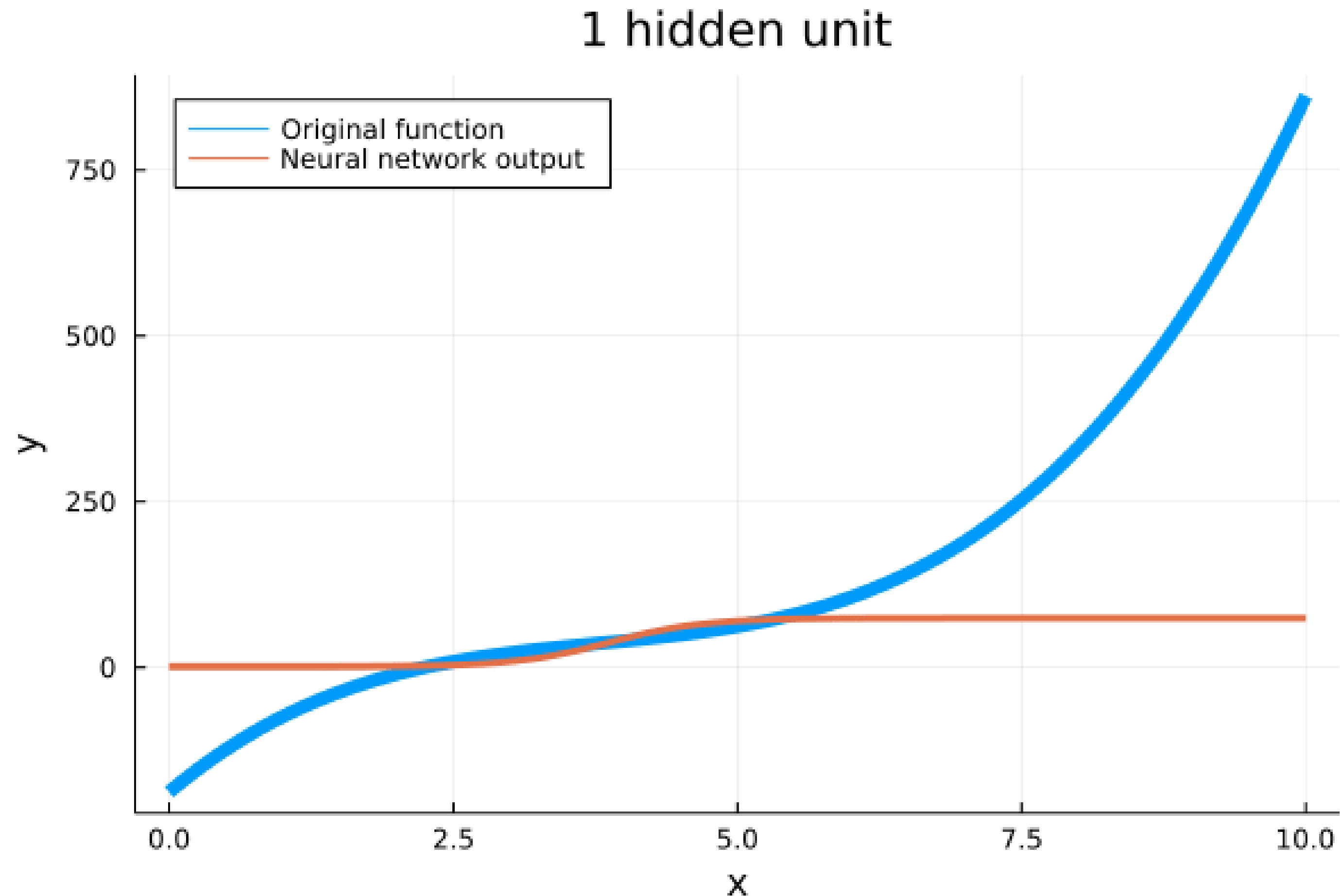
```
# %% Plot loss over number of epochs of training
plt.plot(h.history['loss'])
plt.xlabel('Epoch number')
plt.ylabel('MSE training loss')

# %% Plot predicted values
plt.plot(x, y, label='Original function')
plt.plot(x, m.predict(x), label='Fitted function')
plt.xlabel('x')
```

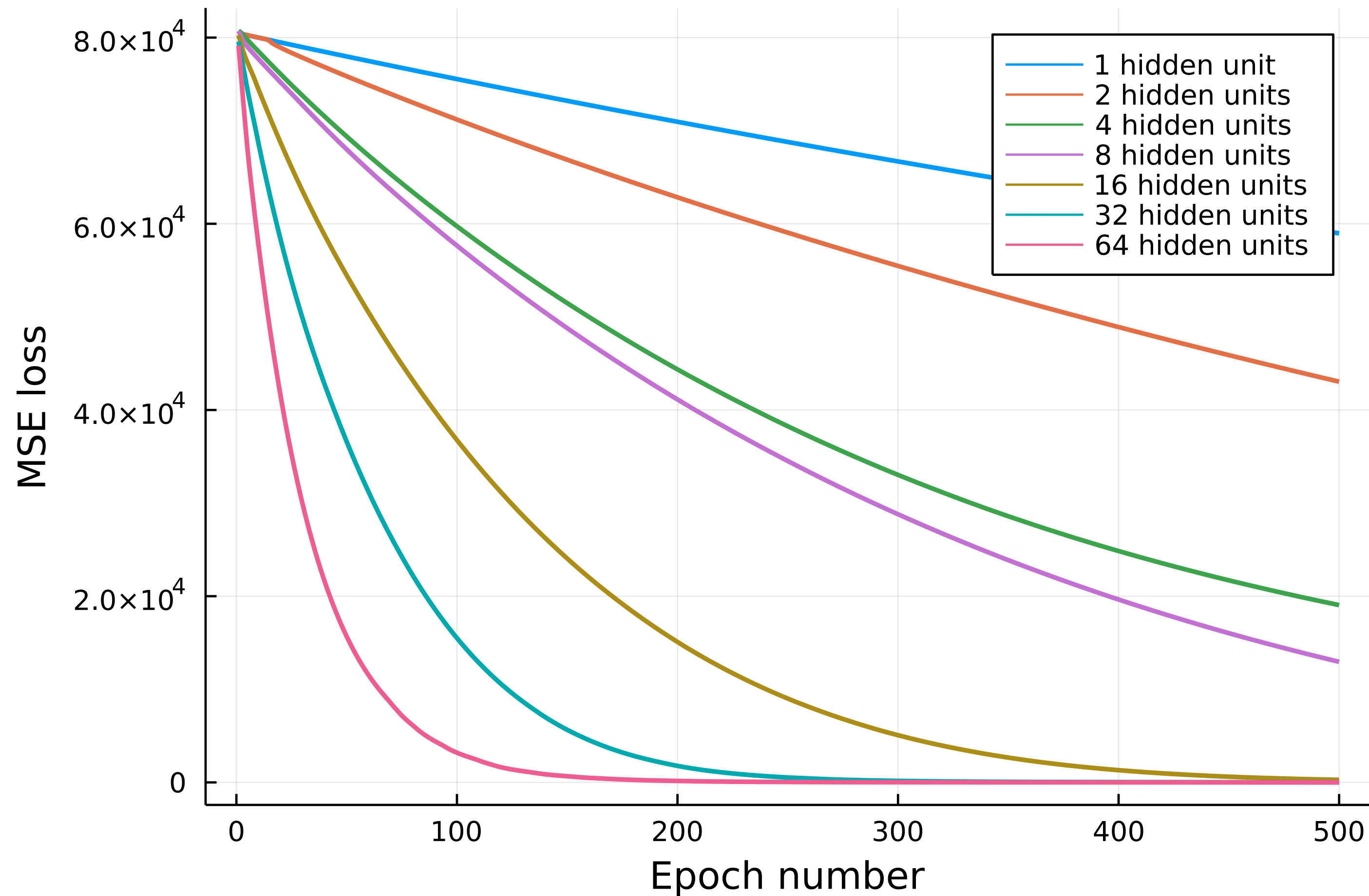
Programming activity

- Download the demo from GitHub and add it to VSCode
- If needed, install the tensorflow, numpy, and matplotlib packages
- Run the code
  - Observe what the program does
  - Change the number of hidden units and see how the results change
  - Change the number of epochs and see how the results change

500 epochs; variable number of hidden units



Loss over 500 epochs; variable number of hidden units



Some neural network theory

Let's reflect on what we just did

- We performed a basis expansion to model a nonlinear function
- We did not tell the computer what features to use for the bases
- Instead, we told the computer to learn features that minimized our mean square error function
  - The basis features we learned were all just logistic regressions
  - The mapping from our sigmoids to $\hat{y}$ was a linear regression
  - Minimizing mean square error is equivalent to minimizing the sum of squared errors

Why does this work?

- It has been proven that given "enough" sigmoid units like we have used, a neural network can **approximate any smooth function**
- This is called the **universal approximation theorem**
- But, what does "enough" mean?
  - It often means "a lot"!

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

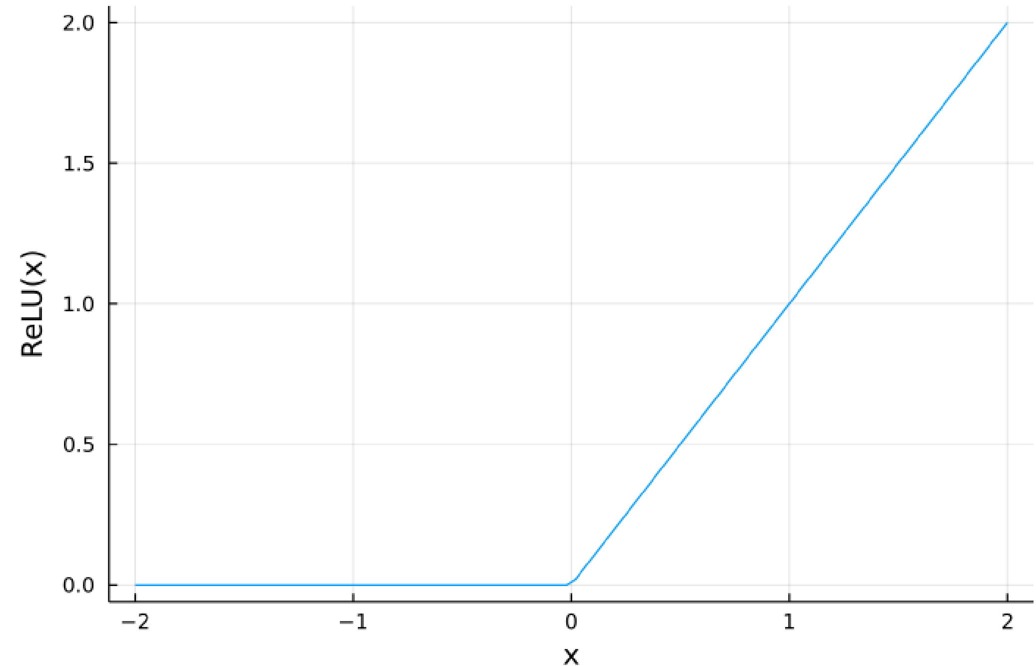
G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of *n* real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

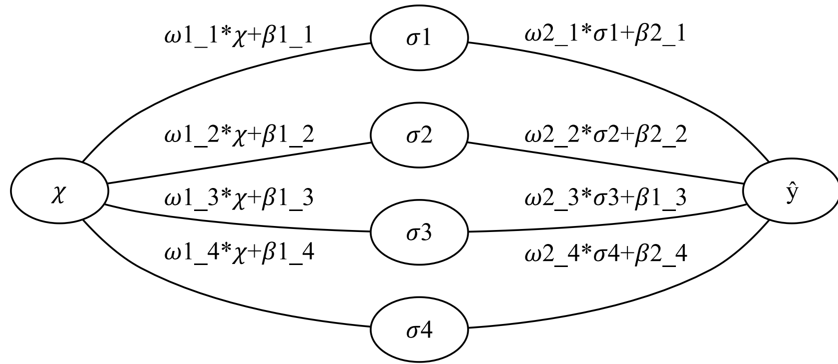
Replacing sigmoids

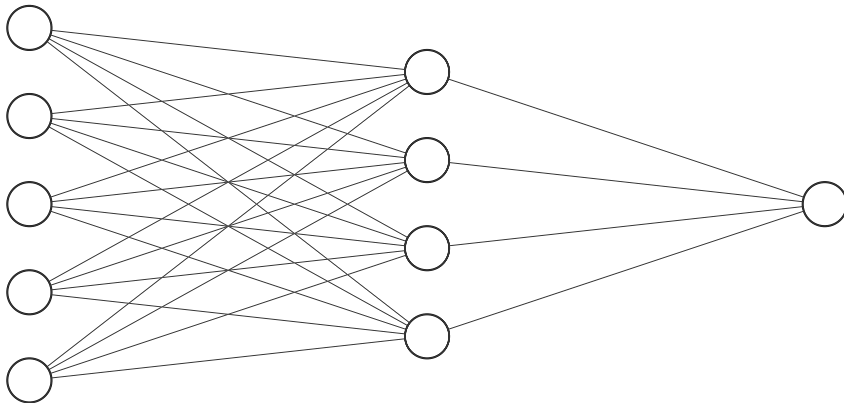
- Sigmoid functions were very popular for a long time
- Sigmoids make some problems harder to optimize because only part of them is very sensitive
- Today, the default **activation function** is the rectified linear unit or "ReLU"
- $\mathrm{ReLU}(x) = \max(x, 0)$



Activation functions and nonlinearity

- The sigmoid function itself was not particularly crucial to neural networks' functionality
- Instead, what mattered is that a nonlinear function was applied to the result of the first $Wx + b$ operation (and to any other layers before the output layer)
- These functions that are applied over the results of $Wx + b$ are called activation functions, often symbolized $a(\cdot)$
  - So, we would have $a(Wx + b)$ for our first layer in our network



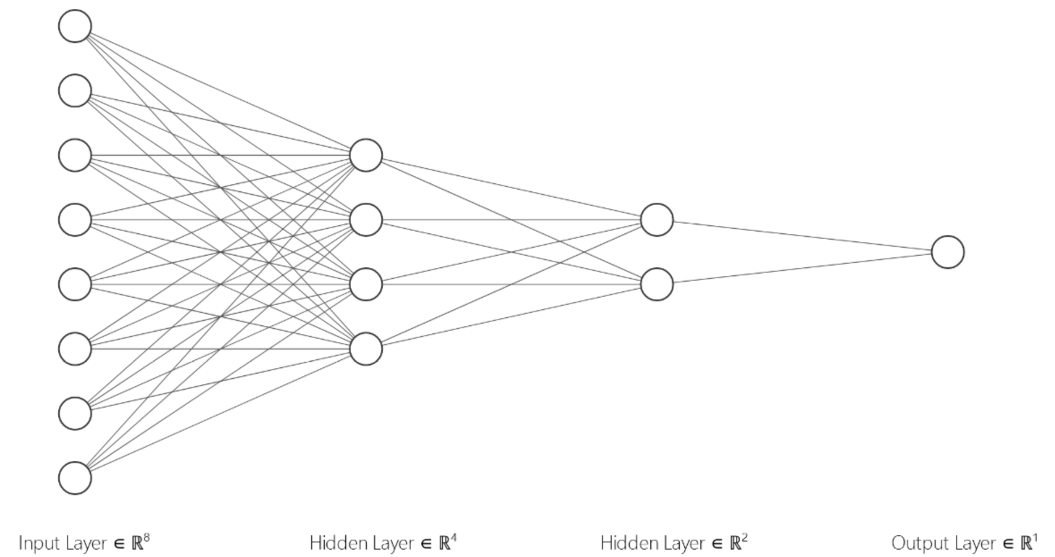


Input variables

- We looked at just one input variable, but you can have as many inputs as you want
- It just increases the number of weights and biases you learn
- Not exactly a traditional basis expansion at this point either since each feature depends on multiple variables now

Depth and width

- You can have additional hidden layers in your network
- Usually, adding layers (**depth**) increases your network's capacity faster than increasing the number of hidden units in a layer (**width**)
- Once you have more than one hidden layer, you have a **deep** neural network
- More on this Tuesday…



Differentiable programming

- Neural networks are an entire computing paradigm
  - One of the core components of **differentiable programming**
- Certain kinds of networks have been proven to be capable of computing every possible computable function (like Python)
- They are, effectively, learning some function that maps from an input to an output
  - The math that performs the optimization is understood fairly well

The (un)interpretability of neural networks

- Neural networks are often considered black boxes meaning that we can't easily understand what their features mean
  - E.g., what do each of the sigmoid features we learned mean? Is one of them equivalent to x^2?
- Contrast this with the slope and intercept in a linear regression that are easily interpretable
  - We can state explicitly what each number represents and how it affects the outcome

The semantics of neural networks

- My analysis: Neural networks are **semantically noncompositional**
  - We deal with this concept all the time in linguistics
  - E.g., *John kicked the bucket* is not made up of the sum of *John, kicked, the*, and *bucket*
- That is, their meaning is not made up of the sum of their parts
- Or, their meaning cannot be predicted by the meaning of their parts
  - We can say precisely what matrix multiplications, vector additions, activation functions, etc. to perform to generate an output
  - But these functions do not tell us why these operations are performed
- In my opinion, assigning meaning to networks' operations requires probing and experimentation, similar to social and behavioral sciences

Reminders

- Assignment 4 due 5/26
- Presentations in two weeks