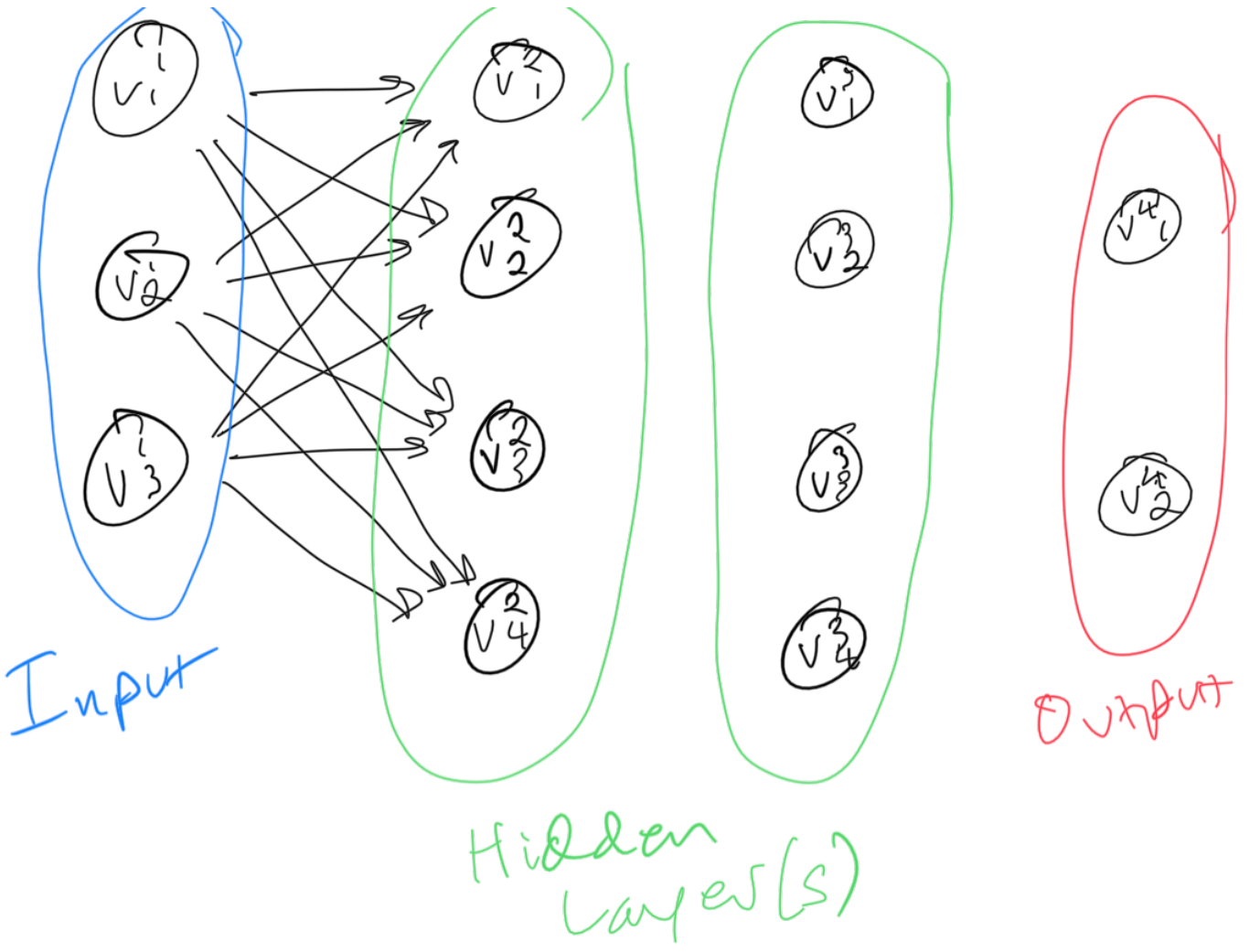
Neural Network Notes

- Working with *huge* amounts of data
  - Parameters
    - Linear Regression has relatively few parameters
      - Say, one coefficient per column; maybe some interactions too
    - NNs can have *billions* of parameters!
  - Performance
    - If the data has a good linear fit, we'll find it with a relatively small training set
      - Diminishing returns from additional data
    - If the data doesn't have a good linear fit, weren't we kinda wasting our time trying to force one?
  - Interpretability
    - A common reason to try for linear fits even if we don't think the data has a good linear fit
  - Recent Developments
    - NNs popular in the 80s but then fell out of favor
    - Recently (past ≈5 years) something magical happened
      - Data volume increased?
      - Computational power crossed some threshold?
      - We now have the ability to make use of enormous amounts of training data
      - This can outperform methods that are perhaps better suited to take advantage of data as a scarce resource
      - Software (TensorFlow) and hardware (specialized processors, GPUs) are being actively developed to speed up training
- How is working with NNs different?
  - Feature Engineering
    - Generally this is something the model does for us
    - Each layer in a NN accepts input from previous layer, giving progressively higher-level features as we move up the network towards the output
      - Hierarchy of understanding from the raw input to final result
      - Higher layers can be useful for other tasks outside of the particular network itself
- Why are they so good?
  - Who knows?
  - Indications that the whole NN structure isn't necessary, and with more research we may learn a more interpretable model
  - Maybe it's overfitting? Using millions of examples to fit millions of parameters, after all
    - But they don't seem to overfit too badly (and we have ways to move along

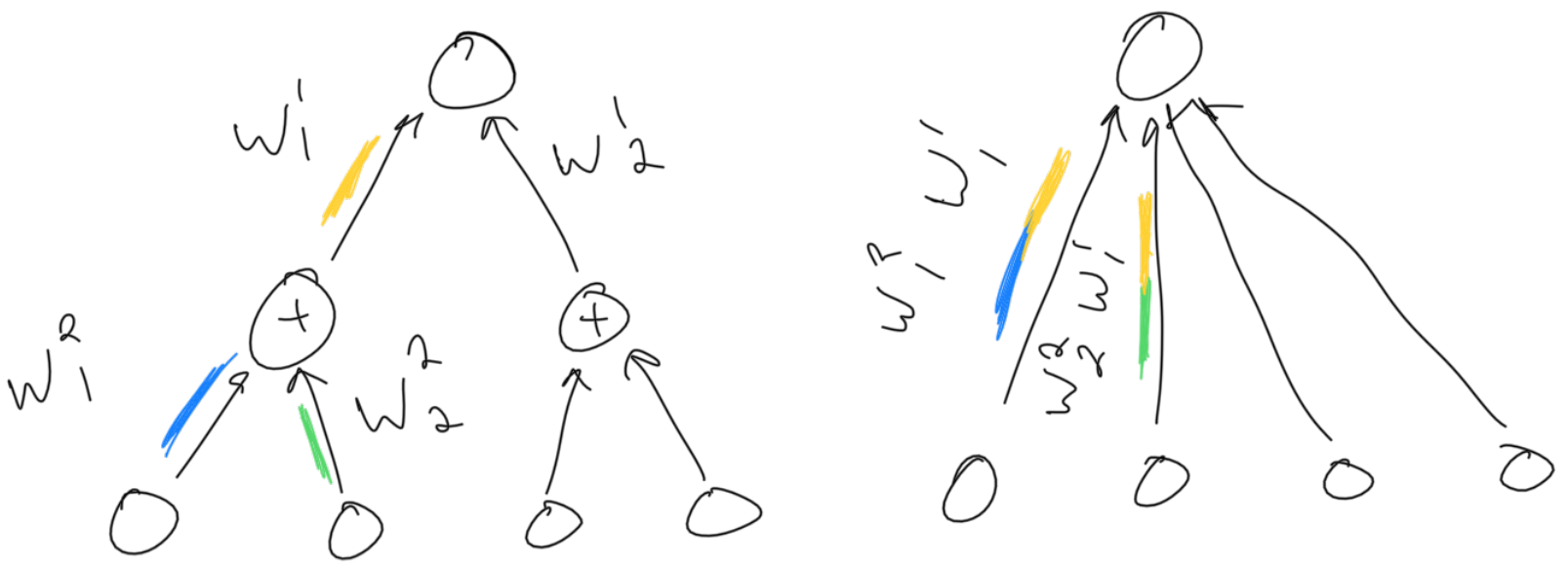
the spectrum)
    - Perhaps it's that while low level features might be overfit by such a ratio of data points::parameters, the high level features are more flexible and generalizable?
- Structure



    - Diagram:
        - It's a graph! It has vertices/nodes and edges
            - Consider how we store a graph in a computer: adjacency matrix, adjacency list, edge list... there are many ways to represent the problem, but they allow us to think at the conceptual level of a graph
            - For NNs, we'll similarly think at the conceptual level of a graph, a network, but when it comes to implementation it'll be lots of dot products and vectors of weights
        - Nodes
            - Each node represents a computational unit, it computes some simple function f(x) over all inputs to it. For example, sigma(x)
        - Edges
            - Each edge (they're directed edges) connects the output of one node to the input of another node higher in the network
        - Weights
            - Each edge has a weight, a multiplier coefficient
            - So, each vertex receives a set of inputs and weights to those inputs (the weights of the edges along which the inputs arrived), and computes a weighted function of its inputs
        - Layers
            - The input layer is a set of variables (they didn't have incoming weights,

        - really, except maybe normalization/standardization), the values of which change when we need the network to make a prediction. It's our input, x
        - The next level(s) up until the last are "hidden layers"; it's not magic, "hidden" just means "not the input or output layer"
        - The last level is the output, which has one or more variables that represent the output of the computation
- Prediction
    - Given a network with weights set and an input, it's fairly straightforward to predict
    - Compute the values of the lowest level in the network, propagate them forward, and repeat until you hit the output layer
- Learning
    - Learning the network is the process of finding the right weights (the coefficient parameters)
    - The more edges there are, the more parameters we have to learn
        - If we add more layers, we're adding lots more edges
    - We learn by analyzing a training corpus of input,output (x,y) pairs, somehow adjusting the weights of the edge parameters so that the output nodes generate something close to y when fed the input x
- Depth
    - In a rough sense, depth should correspond to the conceptual hierarchy associated with the objects being modeled
    - The input is being transformed, distilled, filtered, and shaped as we move up the network
    - In general, the number of nodes per layer should decrease as we move up the network
    - Levels of Abstraction
        - Image: pixels, neighborhood patches, edges, textures, regions, simple objects, compound objects, scenes. Boom: 8 layers sounds reasonable
        - Text: characters, words, phrases, sentences, paragraphs, sections, documents. Boom: 7 layers sounds reasonable
        - The higher levels can be useful representations in other tasks
            - Imagenet is a NN for object recognition in images
            - A high-level layer (not the output) of 1000 nodes measures confidence that the image contains objects of each of 1000 different types. We can apply the already-trained network to a different task, like image similarity, by asking which images share similar values when they are propagated through Imagenet and compared at the 1000 node layer
        - It is not we who assert what each level should represent!

        - Perhaps analogous to how in more traditional ML it's not we who assert what value coefficients should have, we simply come up with the features
    - We simply create a network with a structure that seems like it could potentially recognize the levels of complexity in our data
        - Maybe think of it as "meta feature engineering"; we're engineering what the broadly-speaking "shape" of features might look like, and letting the learning process fill out that skeleton
- Deeper networks become harder to train
    - Recognition performance generally increases with more layers
        - With diminishing returns; adding one hidden layer does a lot, allowing for more power than just recognizing linearly separable classes (which we can do with no hidden layers), but more hidden layers do less per layer (generally)
    - Each new layer adds new parameters to learn, and increases potential to overfit
    - Interpreting the effect of edge weights (which one contributed how much to error?) becomes more difficult, because there are more intervening layers between that edge and the output
    - Real-world size
        - On the higher end (though changing rapidly), tens of layers and billions of parameters
    - Predictions become more expensive with more layers, but it's not that bad and can be done in parallel (each node on a level can be evaluated independently on a different core, for example)
    - Training time is really a bottleneck to training super deep networks
- Nonlinearity
    - Why do more layers let us do more?
    - Linear activation functions don't really benefit from depth



    - Non-linear functions cannot be composed in the same way that, say, addition can be composed to yield addition

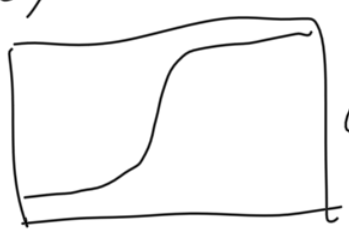
- To benefit from depth, we need a non-linear activation function that operates on a weighted sum of inputs (plus perhaps a bias term that defines activation in absence of other inputs)

$$V_i = \beta + \sum_i W_i X_i$$

↑ input to activation function

Get activation (V_i):

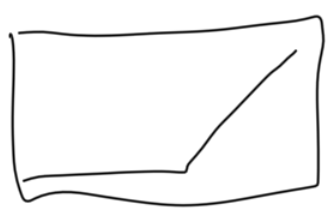
$$net = \frac{1}{1 + e^{-V_i}}$$



σ

OR

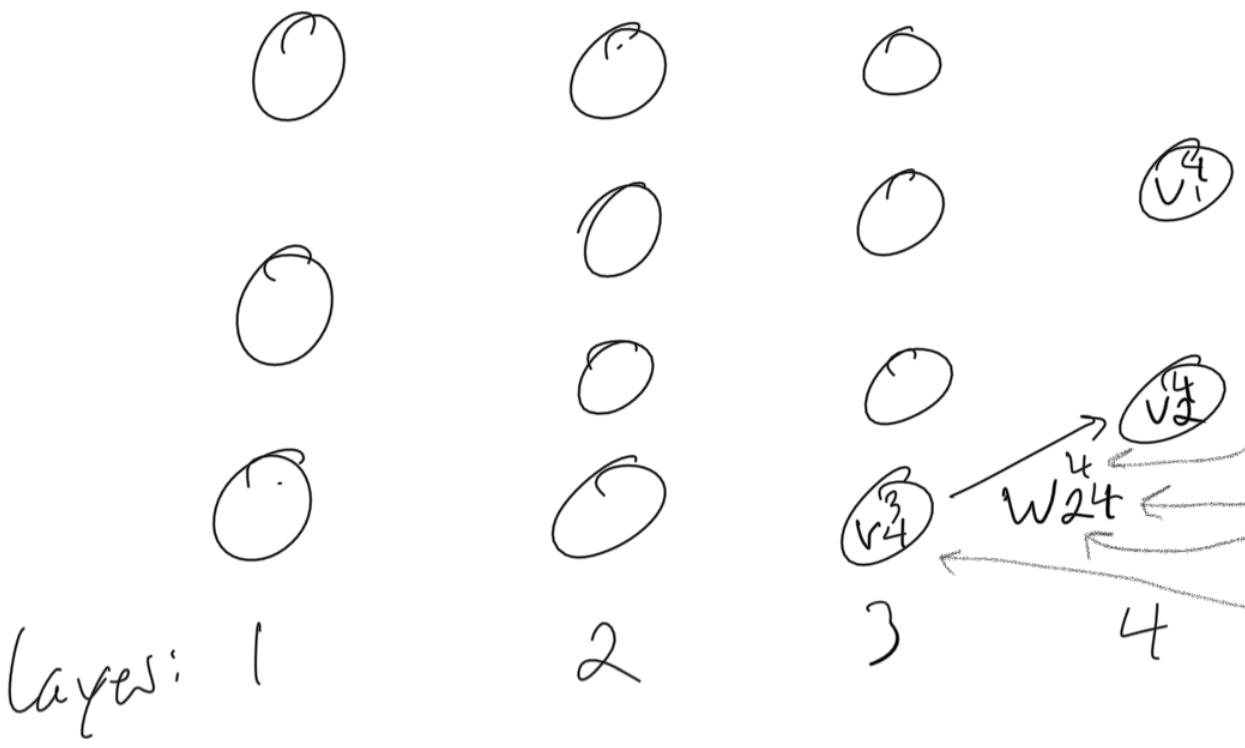
$$net = \max(V_i, 0)$$



ReLU

- More on Prediction
  - It's nice! It's just a single matrix multiplication per level:
    - Get your |V| * |V-1| weight matrix for all the edge weights going from level V to level V-1
    - Get your |V-1| * 1 output vector from the V-1 level
    - Multiply to get a |V| * 1 vector
    - Throw each element of that vector through your activation function (sigma, ReLU, tanh, whatever) and Boom: you have your outputs for this layer to pass on to the next
    - Keep going through all the layers till you hit output
    - Which activation function?
      - Mostly empirical, try a few
- [Break]
  - Training NNs
    - The Humble Perceptron
      - Weights and biases
      - 0 and 1 inputs
      - 0 and 1 outputs
    - Structure
      - NAND
      - Universal Computation
      - Learning?

- Backpropagation
  - It's how we train NNs
  - It goes backwards!
  - Very much like Stochastic Gradient Descent
  - High level
    - We initialize a network with a structure we like and random weights, and we have a training set of data
    - We can calculate the error of the network at a given layer; let's do it for the last layer
    - We can get the derivative of the error (on a mini-batch of data points, that way we don't have to go through all our data points, this is the part that makes it "stochastic" like SGD), our gradient
    - We move in the negative gradient direction some amount, determined by a learning rate
    - This improves the error on the layer we just adjusted; then we move on to the next layer in (going from output->input or top->bottom this time, rather than what we did when we were predicting)



$$E_l = \|y_i - v^4\|^2 = \sum \left(\phi\left(\beta + \sum W_{ij}^{L} V_{ij}^{L-1}\right) - y_{ij}\right)^2$$

    - ^ What an Error metric might look like for a given layer l
- Word Embeddings
  - Consider the best way to represent a phrase for later use as input to some prediction
    - Bag of words
    - LDA, NMF, another Matrix Factorization technique...
    - NN embedding
      - Let's say we created a network where the input accepts the embeddings of 5 words (this third word is embedded in four others around it, two on either side)

    - If the network's task was to predict the middle word W3 from the embeddings of the other four words, we could run backprop to adjust the weights of the nodes in the network to improve on this example; we can continue the backprop past the lowest level, so that we modify the actual input parameters themselves! These parameters represented the embeddings for the words in the given phrase, so improving them improves the embedding for the prediction task; if we do this across enough samples, we get a meaningful embedding for the entire vocabulary
    - See word2vec
        - Main parameter: d, the dimensionality
            - If d is too small, embedding can't fully capture the meaning of the symbol (word)
            - If d is too large, the representation is unwieldy and overfit
            - Good numbers for d empirically are ≈50-300
            -

- Backpropagation
-

Weights:
W_{jk}^L is weight from k^{th} neuron in the $(L-1)^{th}$ layer to the j^{th} neuron in the L^{th} layer.

Biases:
b_j^L is the bias of the j^{th} neuron in the L^{th} layer

Activations:
a_j^L is the activation of the j^{th} neuron in the L^{th} layer

$$a_j^L = \sigma\left(\sum_k W_{jk}^L a_k^{L-1} + b_j^L\right)$$

$\sum_k$: sum over all neurons k in the $(L-1)^{th}$ layer

As a matrix:

W^L : Weights connecting to L^{th} layer of neurons

W^L: (to) j × k (from)

b^L : b_j^L for each j in the L^{th} layer

a^L : a_j^L "

$$a^L = \sigma(W^L a^{L-1} + b^L)$$

(3) (1) (2)

"apply the weight matrix to the activations"

"add the bias"

"apply the σ function (vectorized)"

* if we had W_{kj}^L instead of W_{jk}^L, we'd need $(W^L)^T$ here

z^L: "weighted input to neurons in layer L"

$a^L = \sigma(z^L)$

$z^L = W^L a^{L-1} + b^L$

or

$z_j^L = \sum_k W_{jk}^L a_k^{L-1} + b_j^L$

↑ weighted input to the activation function for neuron j in layer L

For backprop we need to get $\frac{dC}{dw}$ and $\frac{dC}{db}$ of the cost function C w.r.t any weight w or bias b in the network.

Example cost function:

$$C = \frac{1}{2n} \sum_x \| y(x) - a^L(x) \|^2$$

- n: total # training examples
- $\sum_x$: sum over individual training examples
- $y(x)$: desired output
- L: # layers in net
- $a^L(x)$: activations output when x is input

Think of y here as a fixed parameter. This function gives us cost as a function of activations and for any given training example y will be fixed so is not a parameter of our model we can change/learn.

So, think of C as a function of just activations a^L and some other constants.

Hadamard ($\odot$)

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 3 \\ 2 \times 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

We want: $\frac{dC}{dw^L_{jk}}$ and $\frac{dC}{db^L_j}$.

Let's introduce an intermediate quantity:

$$\delta^L_j : \frac{dC}{dz^L_j}$$

(z^L_j: weighted input to activation function)

δ^L_j: error of neuron j in layer L

Why is this a measure of the error of a neuron?

Demon story: demon comes in and tries to help. changes the weighted input to the neuron by Δz^L_j so instead of outputting $\sigma(z^L_j)$ it outputs $\sigma(z^L_j + \Delta z^L_j)$

$\sigma'(z_j^l)$ the neuron activation $\sigma(z_j^l)$ is
Which down the line causes
the cost to change by $\frac{\partial C}{\partial z_j^l} \cdot \Delta z_j^l$

if $\frac{\partial C}{\partial z_j^l}$ is large, the neuron can
flip the sign to help us out, by setting
Δz_j^l's sign to the opposite.

if $\frac{\partial C}{\partial z_j^l}$ is small, not much
leverage to help us out neuron
is pretty near local optima.
So, it's a good measure of the
error of a neuron.

δ^l: vector of errors associated
with layer l

Backprop lets us calculate δ^l
for every layer, and then relate
them to the $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ we want.

Backprop Algorithm:

1) Input: Set the activation for the
input layer: a^1

2) Feedforward: for each layer in l:
Compute $z^l = w^l a^{l-1} + b^l$
and store them. $a^l = \sigma(z^l)$

3) Output error: $\delta^L = \nabla_a C \odot \sigma'(z^L)$
$= (a^L - y) \odot \sigma'(z^L)$

4) Back-propagate the error: for each layer in l:
$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

5) Output: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

$\frac{\partial C}{\partial b_j^l} = \delta_j^l$

6) Update: $w \leftarrow w - \lambda \frac{\partial C}{\partial w}$
$b \leftarrow b - \lambda \frac{\partial C}{\partial b}$

Or, if in mini-batch:
$w^l \leftarrow w^l - \frac{\lambda}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
$b^l \leftarrow b^l - \frac{\lambda}{m} \sum_x \delta^{x,l}$