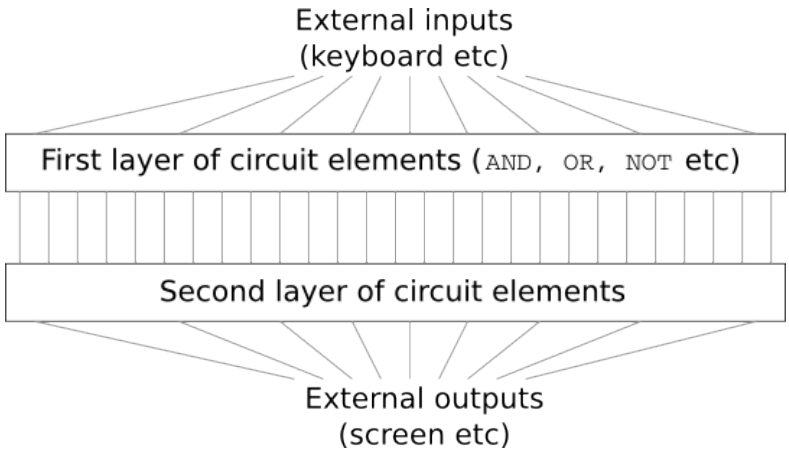
Why are Deep Neural Networks so hard to train?

- Your boss asks you to design a computer from scratch using just two layers of OR, AND, and NAND gates:



  - Turns out that this is possible! (but probably not a good idea)
  - ***Why not possible with just 1 layer though?***

- Deep circuits make the design process much easier
  - But it's not just for ease– math shows that shallow circuits require exponentially more elements than deeper circuits
    - ***Show an easy example on the board how 2 layers of gates make it way easier than when you just have 1***
  - Up till now, the book has just used networks with a single layer
    - And we could still classify stuff to 98% accuracy!
  - But it's hard to train networks as they get deeper

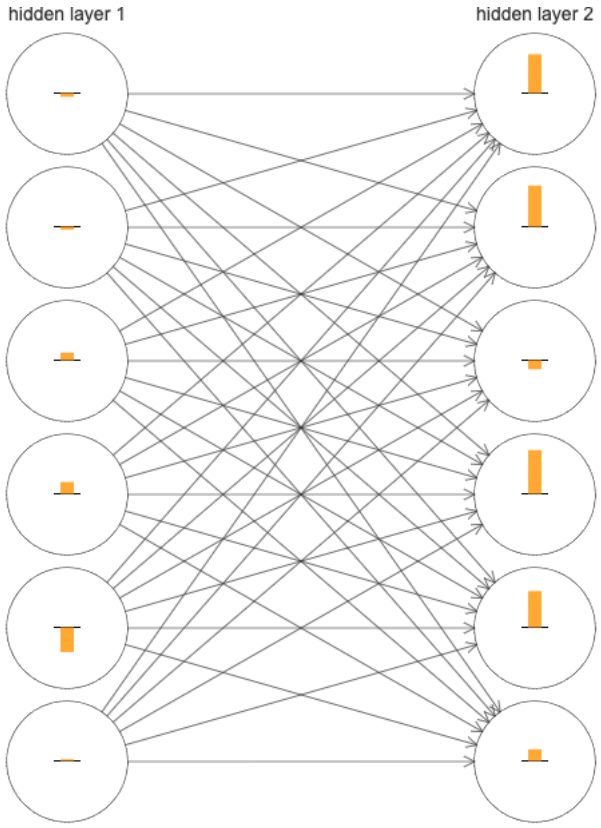
- ***Different layers learn at vastly different speeds.***
  - When later layers in the network are learning well, early layers often get stuck during training
  - But the opposite can also occur– the early layers can learn very well while the later layers become "stuck"

The vanishing gradient problem

- Training MNIST
  - Neural network with 1 hidden layer of 30 neurons
    - **96.48%**
  - 2 hidden layers, each of 30 neurons
    - **96.90%**
  - 3 layers, each of 30 neurons

        - Accuracy DROPS to **96.57**%?!

- Why didn't more hidden layers help, or at least not hurt?
    - It must be that our algorithm isn't learning the right weights and biases (in deep learning, it's more often to refer to the weights and biases together as the ***parameters*** of a model)



- This is the gradient of each neuron (I guess just on one batch?) right at the beginning of training
    - ***What is the difference between a batch and a mini-batch?***
    - This is a very rough measure of the speed at which layers are learning
    - But it's pretty easy to tell: generally, the neurons in the second hidden layer are really learning much more quickly than the neurons in the first hidden layer

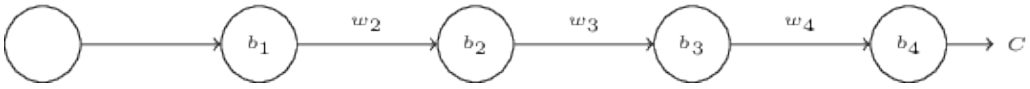
    - And what happens if we add more hidden layers?
        - Same thing—early layers learn more slowly than later layers
- Average speed over many epochs
    - ***Insert the graph with 2 lines here***
    - Shows that they start out learning at very different speeds (which we already know) (also, note that it's a log axis)
        - After that, the second layer learns much more quickly

        - Makes sense in a way – if the second layer can *react* to fit each datapoint pretty closely, the first hidden layer doesn't have much to do
    - **What about with three hidden layers?**
    - **And four?**
    - Still, early hidden layers learn much more slowly than earlier ones
        - First layer learns 100 times more slowly!
    - Basic rule: ***The gradient is getting much smaller as we move backward during the hidden layers***.
- This is known as the **vanishing gradient problem**.
- Sometimes the opposite happens, and the gradient gets much much larger in early layers
    - This it the **exploding gradient problem**
- How do we deal with this during training?
    - Fundamental problem: **the gradient is unstable**—it tends to vanish or explode in early layers
    - This is a fundamental problem for training DNNs
    - How can we take steps to address this?
- One response—is this really a problem?
    - After all, our network pretty much did fine, even with many hidden layers
    - Counter: the weights in the early layers never change much from their random initializations
    - What are the odds that the random initializations were correct? Pretty low!
    - ***With a random initialization, the first layer throws away lots of information about the input image***

What's causing the vanishing gradient problem? Unstable gradients in deep neural networks

- **Vanishing gradient problem**
    - Let's get insight by thinking about the simplest DNN: one with just one hidden neuron in each layer

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

- 

    - Look at the gradient for the bias of the neuron in the first "layer"
    - Make a small change to the bias b1—that change will cascade through the network
    - Let's track the effect of each step in this "cascade"

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2 \sigma'(z_2)\, w_3 \sigma'(z_3)\, w_4 \sigma'(z_4)\, \frac{\partial C}{\partial a_4}.$$

    - The derivative of sigma reaches a maximum at sigma(0) = ¼

        - Generally, weights will be < 1, since they're on a Gaussian with mean 0 and stdev 1
        - So generally, we can see how with more sigma' and more weights (more layers back from the output), the gradient with respect to the bias will be smaller and smaller

- **Exploding gradient problem**
    - He fixes the network to large weights and certain biases that make sigma'(zj) small
        - "With these choices we get an exploding gradient"

- **Unstable** gradient problem
    - Fundamental problem—gradient in early layers is the product of the terms from all the later layers
    - When there are many layers, that's an intrinsically unstable situation
    - With standard gradient-based learning, different layers *have* to learn at wildly different speeds

Unstable gradients in more complex networks

- What about real networks, with more neurons and more layers?
- The same behavior occurs (but has a much more complicated derivation)

Other obstacles to deep learning

- 2010 Glorot and **Y. Bengio** "Understanding the difficulty of training deep feedforward neural networks"
    - There are fundamental problems with the sigmoid activation function
    - They cause the final hidden layer to saturate near 0 early on, substantially slowing down learning
    - Use alternative activation functions and initialization schemes

- 2013 **Sutskever**, Martens, Dahl and **Hinton** "On the importance of initialization and momentum in deep learning"
    - Good choices about
      (1) random weight initialization and
      (2) momentum schedule (for SGD) make a huge difference

- Lots more recent work has done stuff

Summary

- What makes deep networks hard to train? Different things play a role. Important ones:
    - 1. Activation function
    - 2. Weight initialization scheme
    - 3. Details of how gradient descent is implemented (learning rate schedule, optimizer choice)
    - 4. Network architecture
    - 5. Hyperparameters

- ***Deep learning is an *engineering* science!***

Karpathy: A recipe for training neural networks

http://karpathy.github.io/2019/04/25/recipe/

Notes

1. **Neural network training is a leaky abstraction**
   If you really want to get results, "off-the-shelf" libraries will make it harder. The second you deviate from pretrained ImageNet, you're on your own.

2. **Neural network training fails silently**
   Off-by-one bug? Forget to convert an integer to a string? Network still trains—sometimes works—and it will take a long time to figure out (a) something is wrong (b) what went wrong and how to fix it.

   *A "fast and furious" approach to training neural networks does not work! (It takes many iterations to understand how to best fit your dataset and get optimal results. That's why you're never *really* done, as Michael Nielsen said in Chapter 4.)*

The Recipe

1. **Become one with the data**
   First step: Don't touch any neural network code at all. Thoroughly expect and understand your data.

   Spend time scrolling through data. Look for imbalances and biases. Understand the distribution. Your brain is good at this.

   "The neural net is effectively a compressed/compiled version of your dataset." If you understand the data, you'll understand why it's hard to compress. This will make it easier to figure out why your network is underperforming or mis-predicting and how to

improve.

Piece of advice: **write some simple code to search and categorize patterns in your data** and see how well it works. Outliers almost always exist and uncover bugs in data quality or preprocessing.

2. **Set up the end-to-end training/evaluation skeleton and get "dumb" baselines**

   Sub Tips:

   a. Use a fixed random seed
   b. Simplify unnecessary fanciness– delete code you don't user and don't understand!
   c. Use a human baseline
   d. Verify training loss decreases
   e. Print the network to the console and inspect it
   f. Visualize training with TensorBoard or equivalent

3. **Overfit!**

   Now that this works, (1) choose a model large enough to overfit the data and (2) regularize it appropriately.

   In other words: (1) maximize training loss and (2) induce regularization to hurt training loss but help validation loss.

   Sub tips:

   a. Adam optimizer with learning rate 3e-4 is good and pretty forgiving to hyperparameters (even a bad learning rate).
   b. Don't trust learning rate decay defaults.

4. **Regularize**

   Ways to do this:

   a. Get more data
      i. Find more real data to train with
      ii. Augment data to increase training set size

iii. Create synthetic data (simulation doesn't always work!)

b. Shrink input dimensionality—remove features that may not help, or may contain spurious signal
c. Decrease model size (one way—add max pooling)
d. Decrease batch size
    i. Since batch norm normalizes within a batch, smaller batch sizes correspond to stronger regularization
e. Add dropout
f. Increase the penalty for weight decay
g. Use early stopping

5. **Tune**

Now we're "in the loop" with our network and dataset—let's maximize performance by finding best hyperparameters!

a. Use a **random search** over **grid search**
b. Use some toolboxes for optimizing individual hyperparameters, apparently they work

6. **"Squeeze out the juice"**

Found the best hyperparameters? We can still do more!

a. **Use ensembles** "**Model ensembles are a pretty much guaranteed way to gain 2% of accuracy on anything.**"

b. **Train your network for longer** if you have the resources. Try out leaving it training for *way* longer and see if performance improves (your "early stopping" may have stopped too early!).

"One time I accidentally left a model training during the winter break and when I got back in January it was SOTA ("state of the art")."