Unit 2:

Supervised Learning

How do we adjust a neural network so that it does what we want it to do?
How do we get the network to *learn*?

By the end of this unit, you will be able to...

- Formulate the problem of supervised learning as an optimization problem.
- List and explain some of the most common loss/cost functions.
- Describe a perceptron, and its limitations.
- Use gradient descent to optimize connection weights.
- Derive and implement the error backpropagation algorithm.
- Use labelled data wisely to train models that are generalizable.
- Explain the problem of vanishing/exploding gradients.
- Employ some methods to improve the convergence of our learning method.

Neural Learning

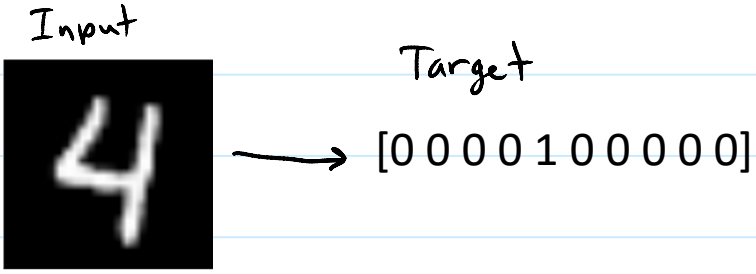
Goal: To formulate the problem of supervised learning as an optimization problem.

Getting a neural network to do what you want usually means finding a set of connection weights that yield the desired behaviour. That is, neural learning is all about adjusting connection weights.

There are three basic categories of learning problems:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

In **supervised learning**, the desired output is known so we can compute the error and use that error to adjust our network.
Example: Given an image of a digit, identify which digit it is.



In **unsupervised learning**, the output is not known (or not supplied), so cannot be used to generate an error signal. Instead, this form of learning is all about finding efficient representations for the statistical structure in the input.
Example: Given spoken English words, transform them into a more efficient representation such as phonemes, and then syllables.

In **reinforcement learning**, feedback is given, but usually less often, and the error signal is usually less specific.
Example: When playing a game of chess, a person knows their play

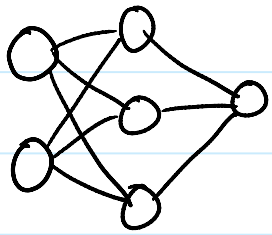
was good if they win the game. They can try to learn from the moves they made.

In this course, we will mostly focus on supervised learning. But we will look at some examples of unsupervised learning, and possibly some reinforcement learning.

Supervised Learning

Our neural network performs some mapping from an input space to an output space.

Eg:



We are given training data, with many MANY examples of input/target pairs. This data is (presumably) the result of some consistent mapping process. For example, handwritten digits map to numbers. Or,

A	B	XOR(A,B)
1	1	
1	0	
0	1	
0	0	

Input

Output/Target

We are given inputs and their corresponding targets, one pair (or a few pairs) at a time. Our task is to alter the connection weights in our network so that our network mimics this mapping.

Our goal is to bring the output as close as possible to the target. But what, exactly, do we mean by "close"? For now, we will use the scalar function as an error function, which returns a smaller value as our outputs are closer to the target.

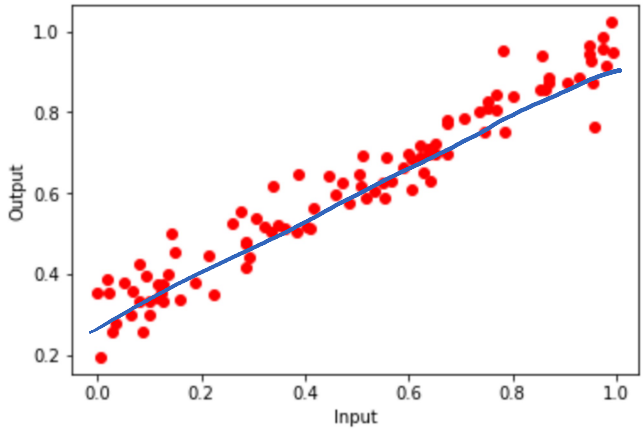
Two common types of mappings encountered in supervised learning are **regression** and **classification**.

Regression

Output values are a continuous-valued function of the inputs. The outputs can take on a range of values.

Example:

Linear Regression



Classification

Outputs fall into a number of distinct categories.

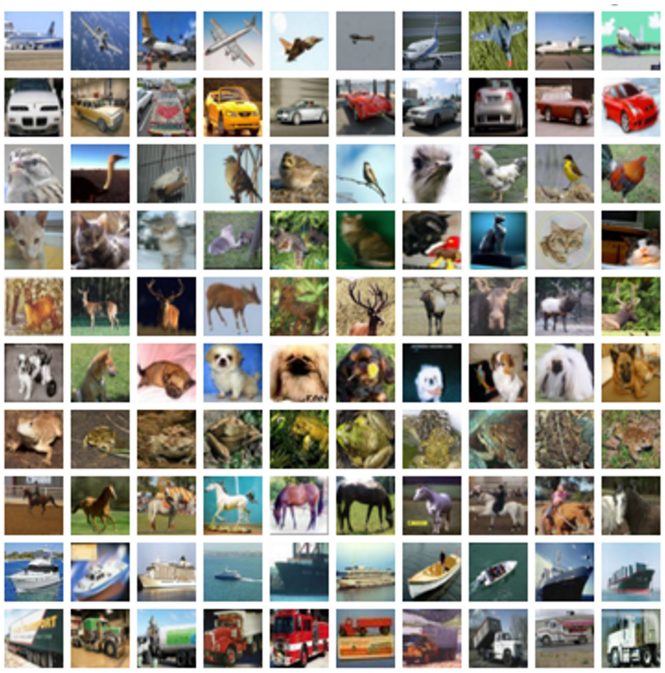
Example:

MNIST

Inputs	Outputs	Inputs	Outputs
	[0 0 0 0 0 0 0 1 0 0]		[0 0 0 0 0 1 0 0 0 0]
	[1 0 0 0 0 0 0 0 0 0]		[0 0 0 0 1 0 0 0 0 0]
	[0 0 0 0 0 0 1 0 0 0]		[0 0 0 0 0 0 0 0 0 1]

CIFAR-10

Inputs



Outputs

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

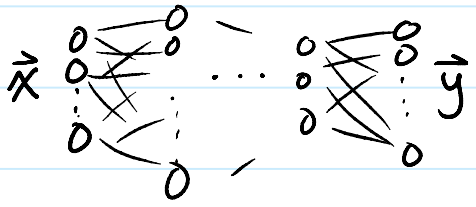
Optimization

Once we have a cost function, our neural-network learning problem can be formulated as an optimization problem.

Let our network be represented by the mapping f, so that

$$\vec{y} = f(\vec{x}; \theta)$$

where θ represents all the weights and biases.



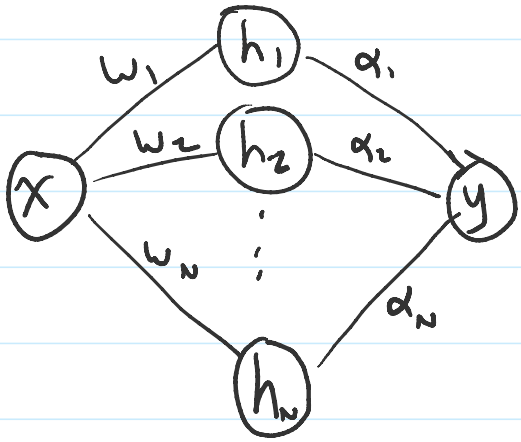
In other words, find the weights and biases that minimize the expected cost between the outputs and the targets.

Universal Approximation Theorem

Question: Can we approximate **any** function using a neural network?

Given a function $f(x)$, can we find the weights ω_j, α_j, and biases θ_j, $j=1,\dots,N$ such that

$$f(x) \approx \sum_{j=1}^{N} \alpha_j \sigma(\omega_j x + \theta_j)$$



to arbitrary precision?

Universal Approximation Theorem:

> **Theorem 2.** *Let σ be any continuous sigmoidal function. Then finite sums of the form*
>
> $$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(\omega_j x + \theta_j)$$
>
> *are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which*
>
> $$|G(x) - f(x)| < \varepsilon \qquad \text{for all} \quad x \in I_n.$$

Cybenko G, "Approximation by Superpositions of a Sigmoidal Function", *Math. Control Signals Systems*, 2:303-314, 1989.

A function σ is "sigmoidal" if $\sigma(x) =$

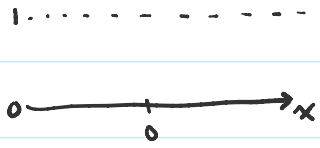
The theorem states that

and for such that

Proof:

S'pose we let for $j = 1, \ldots, N$

Then $\sigma(w_j x)$



Or $\sigma(w_j(x - b_j))$

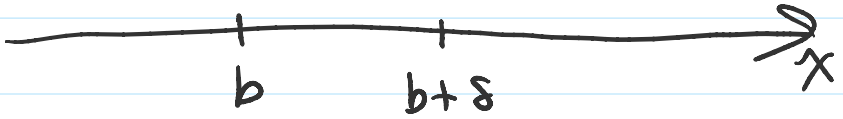
This is the same as the Heaviside step function,

$$H(x) =$$

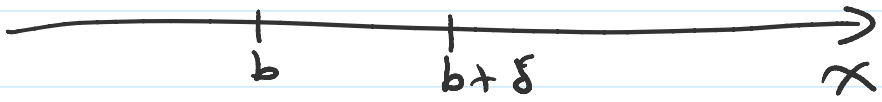
Define: $H(x; b) =$

We can use two such functions to create a piece,

$$P(x; b, \delta) =$$

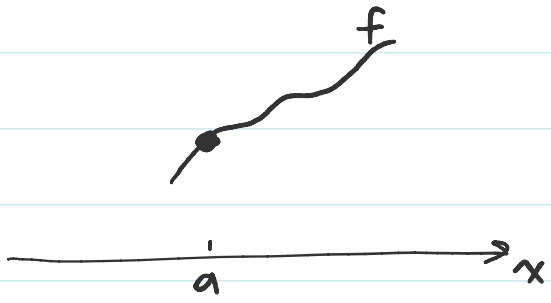


$P(x; b, \delta) =$



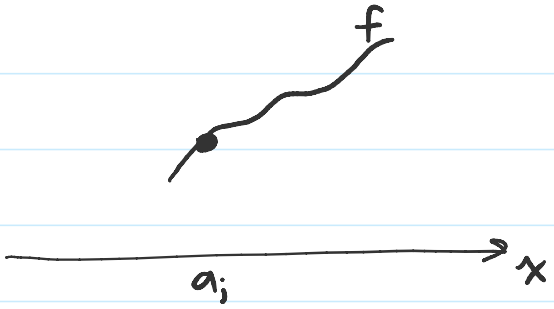
Since $f(x)$ is continuous, $\lim_{x \to a} f(x) = f(a)$. $\forall a \in I_n$

$\therefore \exists$ an interval, such that



Choose b_j , δ_j , and α_j

$\therefore$



Repeat this process for $x = a_{j+1} = a_j + \delta_j$

Construct

$G(x) =$

So, why would we ever need a neural network with more than one hidden layer?

Answer:

Loss Functions

Goal: To become familiar with some of the most common ways to measure error.

We have to choose a way to quantify how close our output is to the target. For this, we use a "cost function", also known as an "objective function". There are many choices, but here are two commonly-used ones.

For input $\vec{x}$, our target is $\vec{t}(\vec{x})$, and the output of our network is $\vec{y}(\vec{x})$.

Mean Squared Error (MSE)

$E(\vec{y}, \vec{t}) =$

$=$

The use of MSE as a cost function is often associated with linear activation functions, or ReLU. This is because these activation functions afford a larger output range.

Cross Entropy

Consider a function (or network) with a single output that is either 0 or 1. The task of mapping inputs to the correct output (0 or 1) is a classification problem.

Suppose we are given a training set,

where the true class is expressed in the target, t_i

If we suppose that y_i is the probability that $x_i \to 1$,

$$y_i = P(x_i \to 1 \mid \theta)$$

then we can treat it as a Bernoulli distribution.

i.e. $P(\quad \mid \theta) = y_i$

Thus $P(\quad \mid \theta) =$

Or

$P(\quad \mid \theta) =$

Based on those probabilities, the likelihood of observing our training dataset is

$P(x_1, \ldots, x_N, t_1, \ldots, t_N) =$

$=$

As is common practice in probability, it is more convenient to look at the

$-\frac{1}{N} \ln P(x_1, \ldots, t_N) =$

$=$

$=$

This log-likelihood formula is the basis of the cost function.

Cross entropy assumes that the output values are in the range [0, 1].

Hence, it works nicely with the logistic activation function.

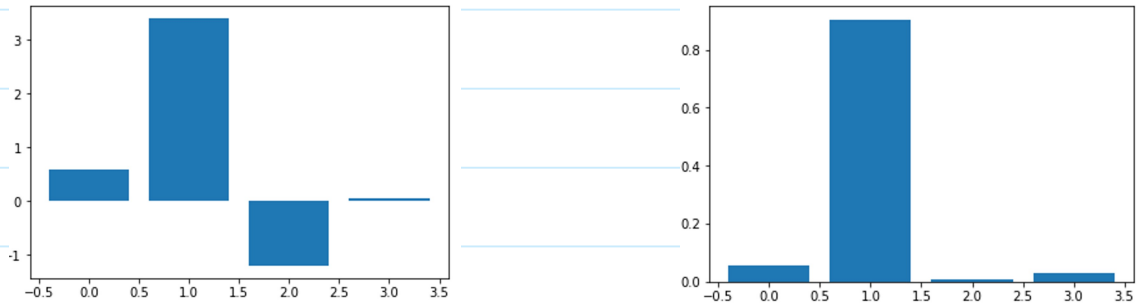
SoftMax

SoftMax is like a probability distribution (or probability vector), so its elements add to 1. If z is the drive (input) to the output layer, then

Then, by definition,

Example:

$z = [0.6, 3.4, -1.2, 0.05] \xrightarrow{\text{softmax}} y =$



One-Hot

One-Hot is the extreme of the softmax, where only the largest element remains nonzero, while the others are set to zero.

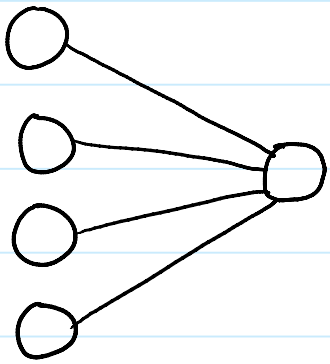
$z = [0.6, 3.4, -1.2, 0.05]$

Perceptrons

Goal: To see a simple neural learning algorithm, and understand its limitations.

Let's look at a simple neural network that aims to recognize certain input patterns.

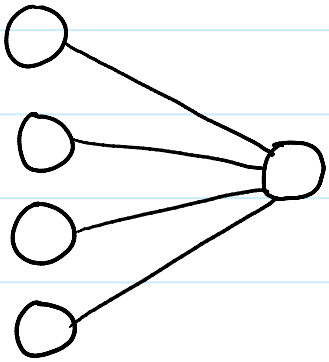
For example, let the output node be a simple threshold neuron, and suppose we want the output node to be 1 when the input is [1, 1, 0, 1], and zero otherwise.



Notice that if we set the weights to [1, 1, 0, 1] (matching the input), then we maximize the input to the output node.

But what about other inputs?

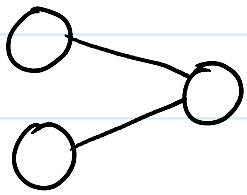
We need the un-matching input to give us a negative value so that the output node returns zero.
Solution: a negative bias



Can we find the weights and bias automatically so that our

perceptron produces the correct output for a variety of inputs?

To see an approach, let's look at a 2-D case.



Suppose the 4 different inputs are:

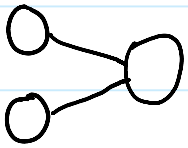
[0, 0], [0, 1], [1, 0], and [1, 1]

And their corresponding outputs are

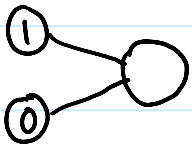
0, 1, 1, 1 (an "OR" gate)

Also, we will use the L1 error:

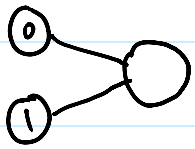
Start with random weights

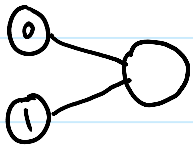


Input [1, 0] target 1

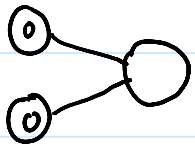


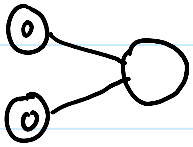
Input [0, 1] target 1



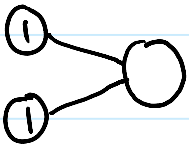


Input [0, 0] target 0

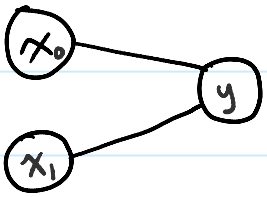




Input [1, 1] target 1

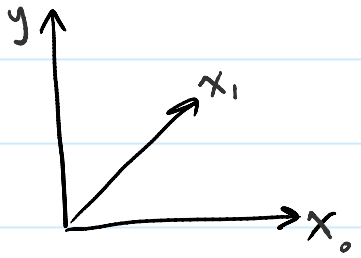


Graphical Interpretation

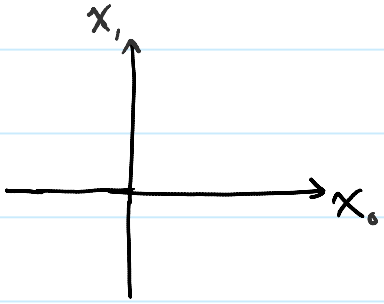


$y =$

this is a linear equation... the equation of a plane in 3-D.



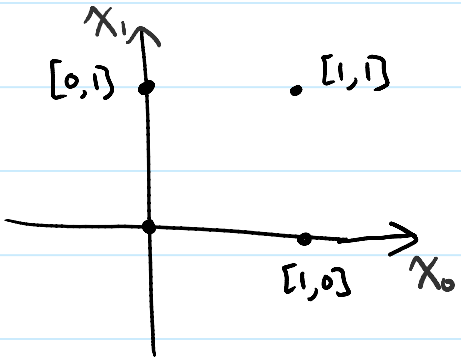
Looking down on the x_0-x_1 plane,

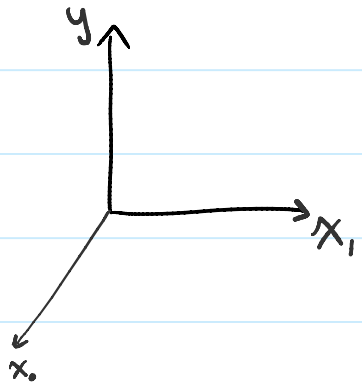


Finding the weights and bias is the same as finding a linear classifier… a linear function that returns a positive value for the inputs that should yield a 1, and a negative value for the inputs that should yield a 0.

Another example:

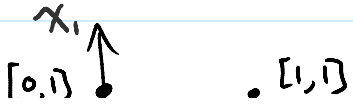
[0, 0] -> 0 [0, 1] -> 1 [1, 0] -> 0 [1, 1] -> 1

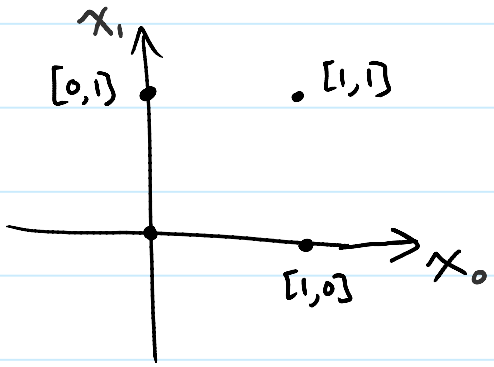




A final example:

[0, 0] -> 0 [0, 1] -> 1 [1, 0] -> 1 [1, 1] -> 0





Perceptrons are simple, two-layer neural networks, so only work for

If you want to handle non-linearly-separable data, your neural network is going to need more layers.

But they give us our first glimpse at a learning algorithm.

(demo perceptron)

Gradient Descent Learning

Goal: To see how we can use a simple optimization method to tune our network weights.

The operation of our network can be written

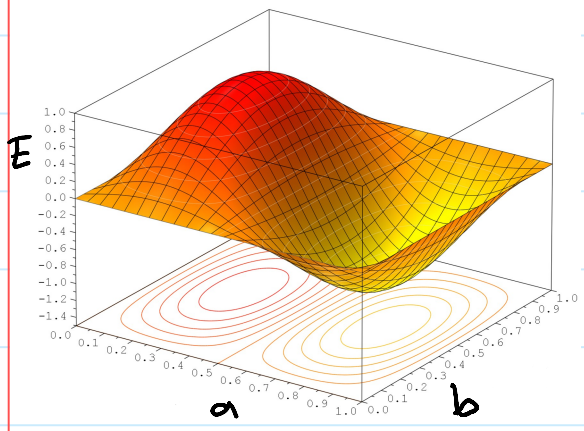
$$\vec{y} = f(\vec{x}; \theta)$$

So, if our cost function is $E(\vec{y}, \vec{t})$, where $\vec{t}$ is the target, then neural learning becomes the optimization problem

We can apply gradient descent to E, using the gradient

Gradient-Based Optimization

If you want to find a local maximum of a function, you can simply start somewhere, and keep walking uphill.
For example, suppose you have a function with two inputs, $E(a, b)$.
You wish to find a and b to maximize E.



https://commons.wikimedia.org/wiki/File:2D_Wavefunction_(2,1)_Surface_Plot.png

We are trying to find the parameters $(\bar{a}, \bar{b})$ that yield the maximum value of E.

i.e.

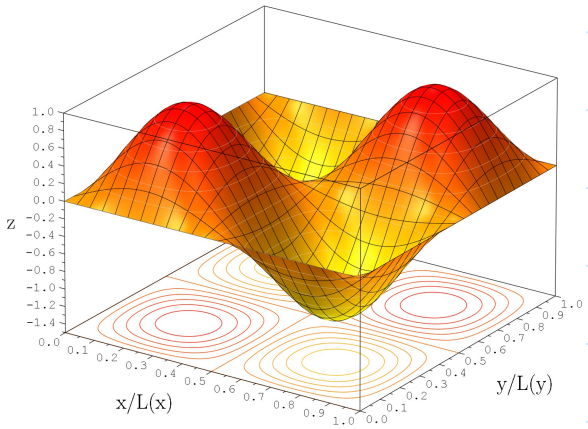
No matter where you are, "uphill" is in the direction of the gradient vector,

Gradient ascent is an optimization method where you step in the direction of your gradient vector.
If your current position is , then

where is your step multiplier.

Gradient **DESCENT** aims to **minimize** your objective function. So, you walk downhill, stepping in the direction **opposite** the gradient vector.

Note that there is no guarantee that you will actually find the global optimum. In general, you will find a local optimum that may or may not be the global optimum.



Approximating the Gradient Numerically
We can estimate the partial derivatives in the gradient using finite-differencing.

Finite-Difference Approximation
For a function $f(\theta)$, we can approximate $\frac{df}{d\theta}$ using

$$\frac{df}{d\theta} \approx$$

As an example, consider this network:

It's a neural network, with connection weights and biases shown.
We can model the action of the entire network using

$$\vec{y} = f(\vec{x};\theta)$$

And we can formulate the problem as

Or, more compactly,

Consider θ_1 on its own. With θ_1 = -0.01, our network output is

This gives

What if we perturb θ_1, so that θ_1 = -0.01 **+ 0.5** = 0.49.
Then our output is

This yields

If, instead, we perturb θ_1 so that θ_1 = -0.01 **- 0.5** = -0.51 ,
then our output is

which gives

In summary,

Parameters	MSE
$\theta_1+\Delta\theta$	
θ_1	
$\theta_1-\Delta\theta$	

We can estimate $\frac{\partial \bar{E}}{\partial \theta_1}$ using

$\frac{\partial \bar{E}}{\partial \theta_1} \approx$

Obviously, seems to be the right thing to do.

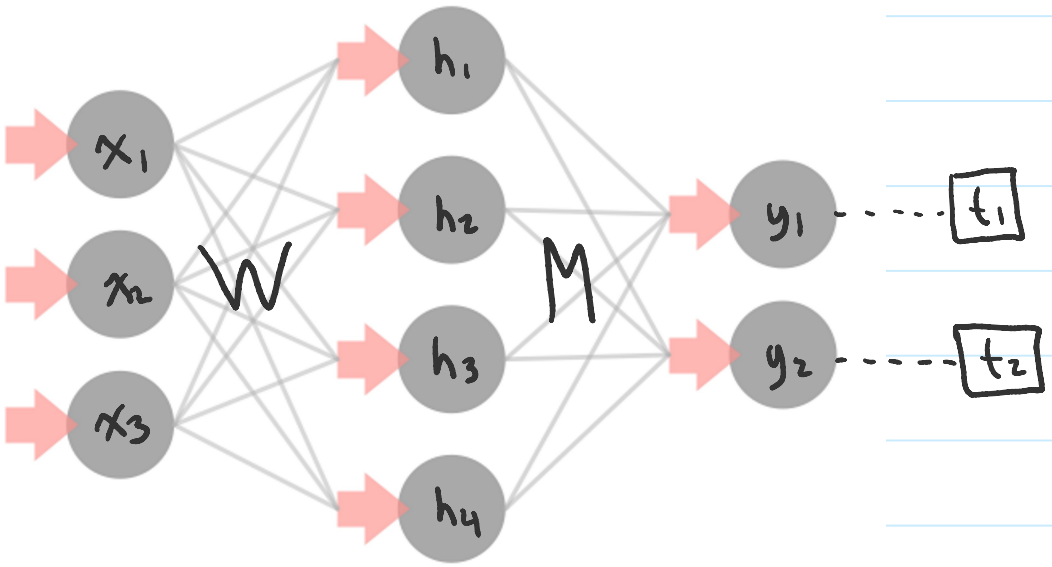
(demo: XOR example)

Error Backpropagation

Goal: To find an efficient method to compute the gradients for gradient-descent optimization.

We can apply gradient descent on a multi-layer network, again using chain rule to calculate the gradients of the error with respect to deeper connection weights and biases.

Consider the network



α_i is the input current to hidden node i.

β_j is the input current to output node j.

For our cost (loss) function, we will use

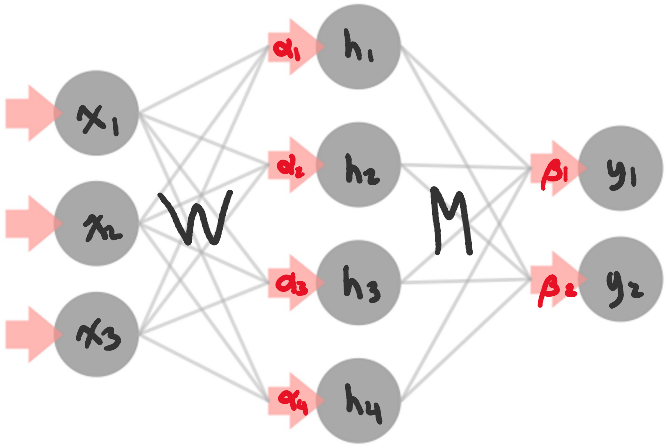
For learning, suppose we want to know, $\frac{\partial E}{\partial M_{14}}$

eg. M_{14}

Recall,

Thus,

OK, that works for the connection weights between the top two layers. What about the connection weights between layers deeper in the network?



$$\frac{\partial E}{\partial w_{12}} =$$

$$=$$

$$=$$

$$=$$

$$=$$

More generally, $\vec{x} \in \mathbb{R}^X$, $\vec{h} \in \mathbb{R}^H$, $\vec{y}, \vec{t} \in \mathbb{R}^Y$

$$\frac{\partial E}{\partial \alpha_i} =$$

$=$

For more elements

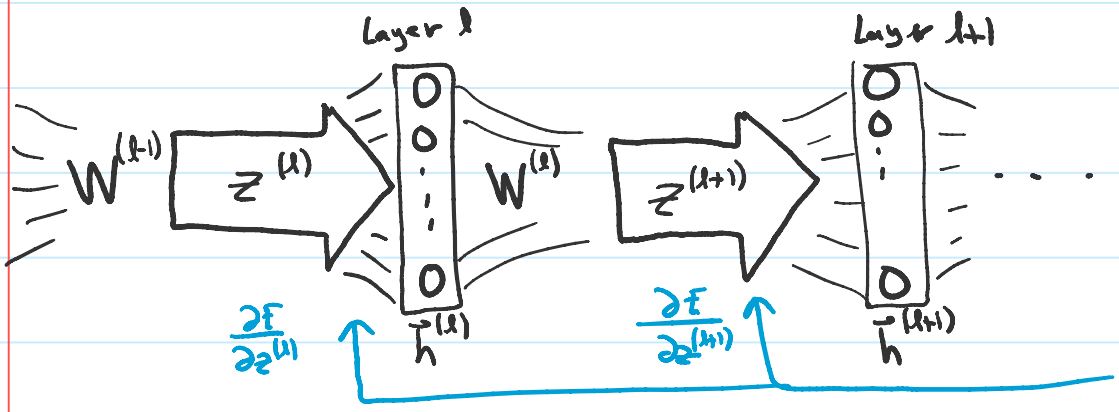
$$\begin{bmatrix} \frac{\partial E}{\partial a_i} \\ \frac{\partial E}{\partial a_n} \end{bmatrix} =$$

For all elements,

$$\begin{bmatrix} \frac{\partial E}{\partial a_1} \\ \vdots \\ \frac{\partial E}{\partial a_n} \end{bmatrix} =$$

$=$

The most general, in going down a layer



Suppose we have $\frac{\partial E}{\partial \vec{z}^{(l+1)}} = \nabla_{l+1} E$

Let $\vec{h}^{(l+1)} = \sigma\left(\vec{z}^{(l+1)}\right) = \sigma\left(W^{(l)}\vec{h}^{(l)} + b^{(l+1)}\right)$

$$\frac{\partial E}{\partial \vec{z}^{(l)}} =$$

Then, to compute $\frac{\partial E}{\partial W_{ij}^{(l)}}$ ...

$$\frac{\partial E}{\partial W_{ij}^{(l)}} =$$

$$=$$

$$\frac{\partial E}{\partial W^{(l)}} =$$

Training and Testing

Goal: Develop a process to use our labelled data to generate models that can predict future, unseen samples.

We have seen how to adjust a neural network to get it to learn our training data. Of course, the purpose of training a model is so that we can use it on other samples that aren't in our training set. For this reason, we usually break our data into two pieces:

1. Training set: Use most of your labeled data to train your model.
2. Test set: Once your model is trained, use the remaining labeled samples to evaluate your model.

Why? Suppose after some trial-and-error, adjusting the hyperparameters:

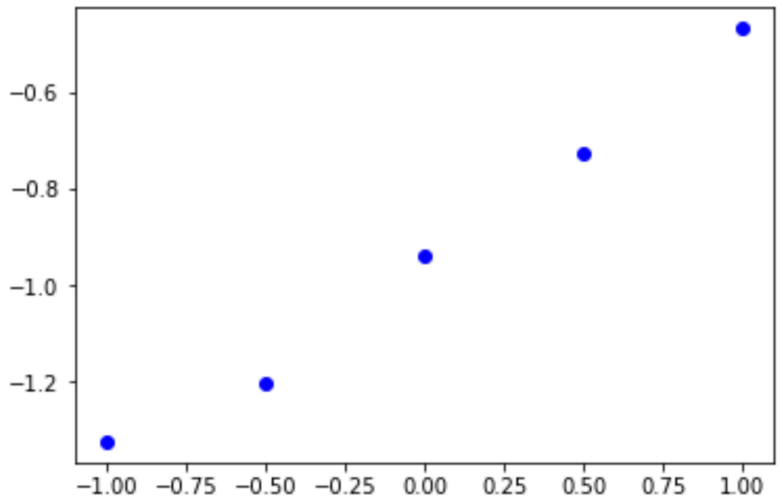
- number of neurons in each layer
- learning rate
- number of epochs
- initial weights

you finally get a low error on the training data. Does that accomplish what you want? Consider an example.

As an experiment, consider noisy samples coming from the ideal mapping,

$$y = 0.4x - 0.9$$

We can only get noisy samples from that mapping.



Our training dataset has 5 samples. And since this is a regression problem, we will use a activation function on the output, and as a loss function.

Training usually entails going through the training data repeatedly, updating the network weights as we go. Each pass through the data is called an

Let's create a neural network to learn this mapping.

```
net = Network([1, 1000, 1])
```

Before training...

```
Training MSE = 0.955758517256
```

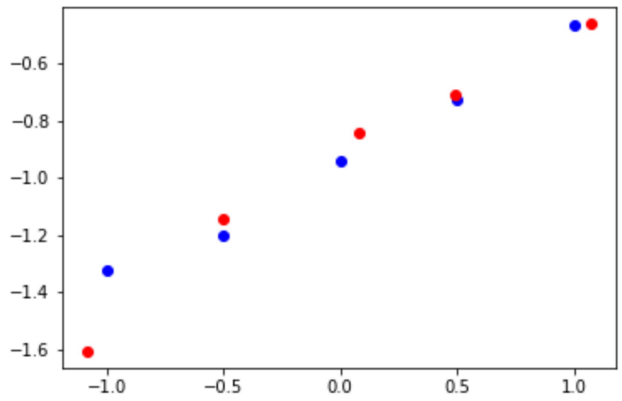
Call the learning function for many epochs.

```
net.Learn([training_input, training_output], epochs=40000, lrate=0.01)
```

After going through the dataset 500 times, our average loss is

```
Training MSE = 0.000692658082161
```

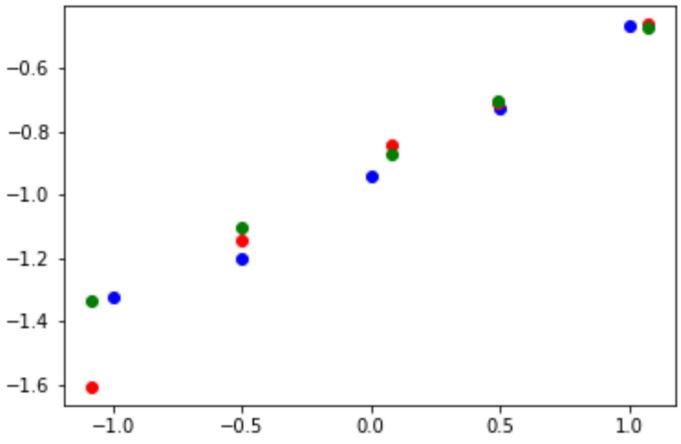
Let's bask in the glory of our brilliance, and demonstrate how great we are on another sampling of data.



Our average loss on this new set of samples is

```
Test MSE = 0.0156456136501
```

In fact, what happens when we give it a perfect dataset, without noise?



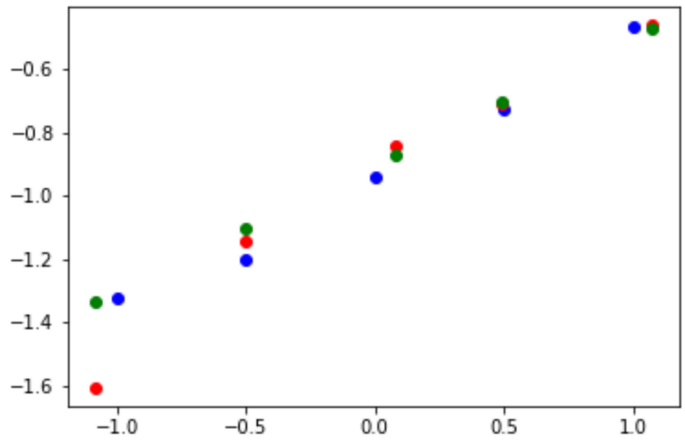

```
Perfect MSE = 0.00161413205859
```

Even perfect data has a higher error than the test data. ???

Recall that our sole purpose was to create a model to predict the output for samples it hasn't seen. How can we

avoid overfitting?

The false sense of success we get from the results on our training dataset is known as



The model starts to fit the noise specific to the training set, rather than just to the underlying model.

Validation

If we want to estimate how well our model will generalize to samples it hasn't trained on, we can withhold part of the training set and try our model on that "validation set". Once our model does reasonably well on the validation set, then we have more confidence that it will perform reasonably well on the test set.

It's common to use a random subset of the training set as a validation set.

Overfitting

Goal: See some tricks for how to mitigate against overtraining.

We saw that if a model has enough degrees of freedom, it can become hyper-adapted to the training set, and start to fit the noise in the dataset.

This is a problem because the model does not generalize well to new samples.

There are some strategies to try to stop our network from trying to fit the noise.

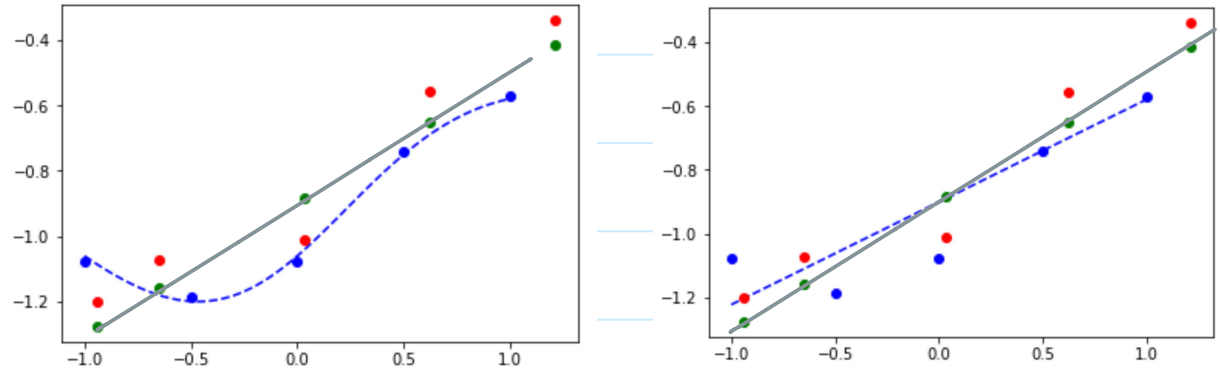
Regularization

Weight Decay

We can limit overfitting by creating a preference for solutions with smaller weights, achieved by adding a term to the loss function that penalizes for the magnitude of the weights.

How does this change our gradients, and thus our update rule?

How does this change our gradients, and thus our update rule?




```
Training MSE = 0.00596061335994
    Test MSE = 0.00574788521961
 Perfect MSE = 0.00178339752734
```

λ controls the weight of the regularization term.

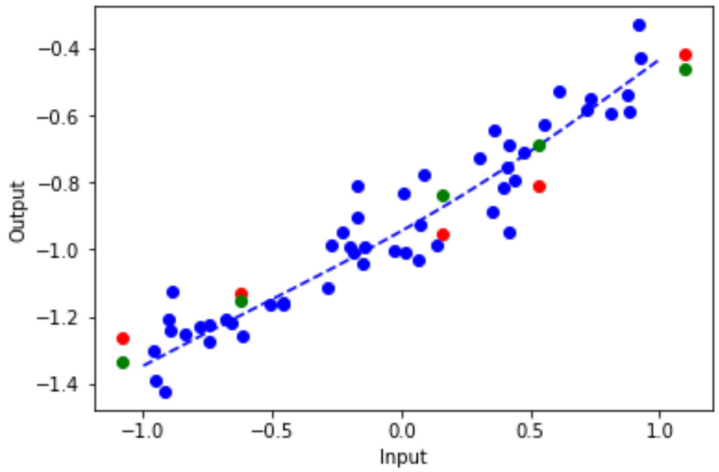
One can also use different norms. For example, it is common to use the L1 norm,

The L1 norm tends to favour sparcity (most weights are close to zero, with only a small number of non-zero weights).

Data Augmentation

Another approach is to include a wider variety of samples in your training set, so that the model is less likely to focus its efforts on

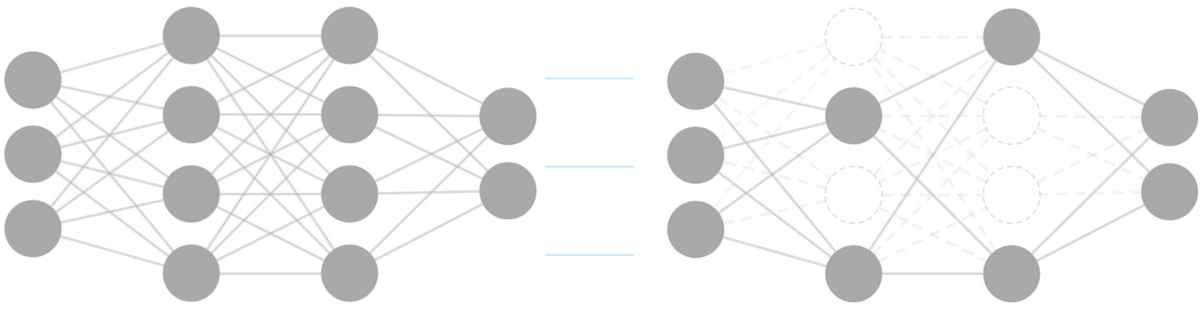
the noise of a few.




```
Training MSE = 0.00694267279424
    Test MSE = 0.00685380408996
 Perfect MSE = 0.00627912229049
```

Where does this extra data come from?
For image-recognition datasets, one can generate more samples by or the images. Those transformations presumably do not change the labelling.

Dropout
The last method we will talk about is the most bizarre.
While training using the dropout method, you systematically ignore a large fraction (typically half) of the hidden nodes for each sample. That is, you randomly choose about half of your hidden nodes to be temporarily taken off-line and set to zero.



Do both a feedforward and backprop pass with this diminished network.

Then apply mask

We need to double the remaining current to

Let

Likewise, we get

During backprop, suppose we have

Then,

And

Recall,

Thus,

as long as

However, you still have to calculate , or additionally ensuring that

Often, you can compute without , using only

This is true for the logistic function

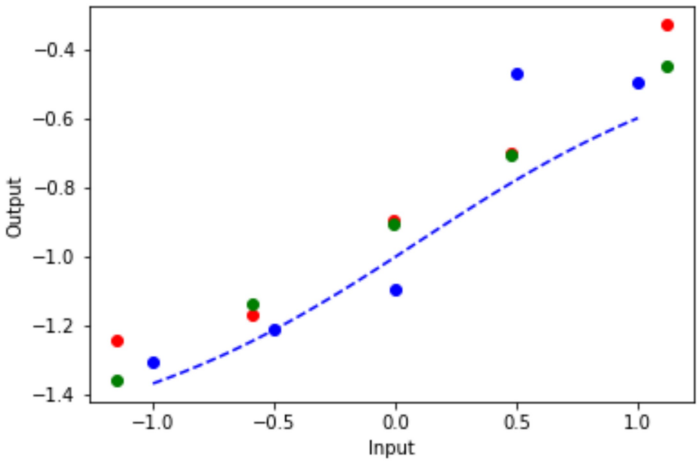
It even has the property that

This is convenient, since we can get from

So

Important caveat:

You have to the connection weights projecting from a diminished layer in order to give inputs to the next layer.




```
Training MSE = 0.0234801932225
    Test MSE = 0.0212572143563
 Perfect MSE = 0.00874342134987
```

Why does dropout work?

- It's akin to training a bunch of different networks and combining their answers. Each diminished network is like a contributor to this consensus strategy.
- Dropout disallows sensitivity to particular combinations of nodes. Instead, the network has to seek a solution

that is robust to loss of nodes.

Deep Neural Networks

Goal: To see the advantages and disadvantages of deep neural networks: representational power vs. vanishing or exploding gradients.

How many layers should our neural network have?

Recall the Universal Approximation Theorem:

Theorem 2. *Let σ be any continuous sigmoidal function. Then finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(\omega_j x + \theta_j)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \varepsilon \qquad \text{for all} \quad x \in I_n.$$

Cybenko G, "Approximation by Superpositions of a Sigmoidal Function", *Math. Control Signals Systems*, 2:303-314, 1989.

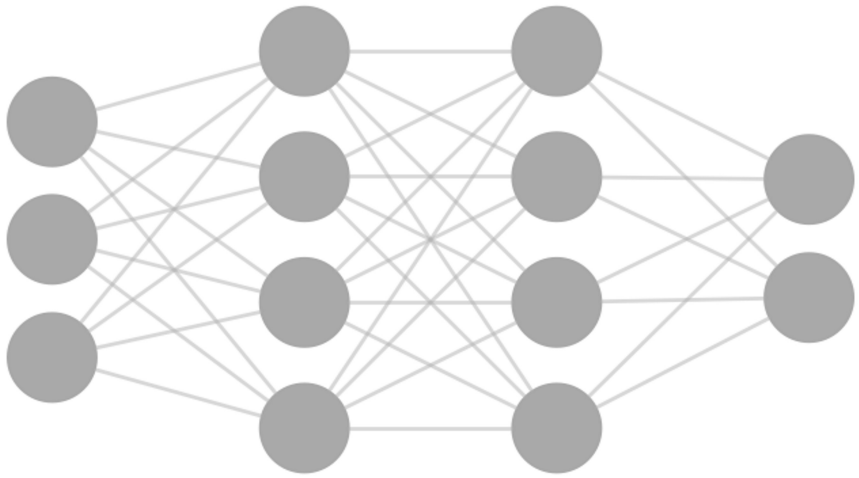
Thus, we really only ever need one hidden layer. But is that the best approach, either in number of nodes, or learning efficiency? No, it can be shown that such a shallow network would require an exponentially large number of nodes (ie. A really big N) to work.

So, a deeper network is preferred in many cases. (see Python example)

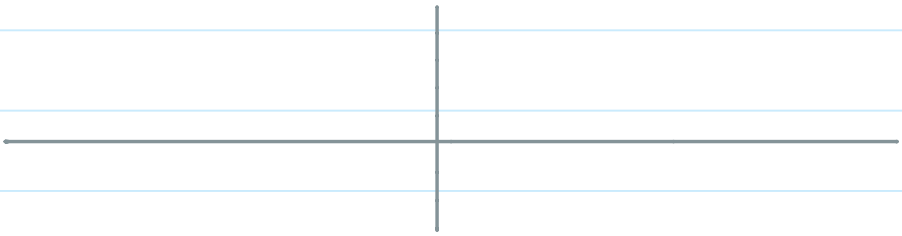
So, why don't we always use really deep networks?

Vanishing Gradients

Suppose the initial weights and biases were large enough that the input current to many of the nodes was not too close to zero.
As an example, consider one of the output nodes.



Compare that to if the input current was 0.1.



Hence, the updates to the weights will be smaller when the input currents are large in magnitude.

What about the next layer down?

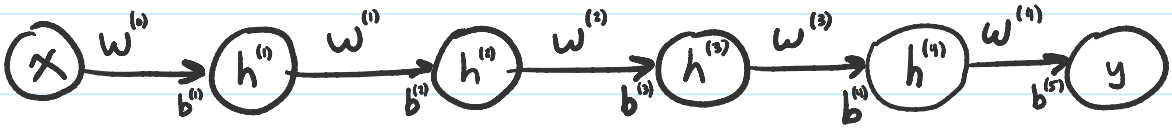
What if the inputs to the penultimate layer were around 4 in

magnitude?
Then the corresponding slopes of their sigmoid functions will also be small.

Recall that

And it gets smaller and smaller as you go deeper.
When this happens, learning comes to a halt, especially in the deep layers. This is often called the

Another way to look at it. Consider this simple, but deep network.



Start with the loss on the output side:
The gradient w.r.t. the input current of the output node is

$$\frac{\partial E}{\partial z^{(5)}} =$$

Then, using backprop, we can compute a single formula for

$$\frac{\partial E}{\partial z^{(4)}} =$$

$$\frac{\partial E}{\partial z^{(i)}} =$$

Going deeper...

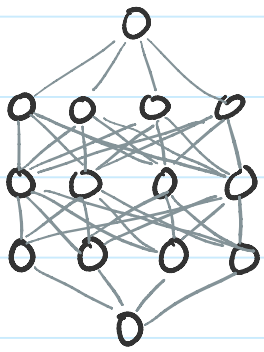
$$\frac{\partial E}{\partial z^{(i)}} =$$

What is the steepest slope that $\sigma(z)$ attains?

$$\sigma(z) =$$

All else being equal, the gradient goes down by a factor of at least 4 each layer.
We can see this if we look at the norm of the gradients at each layer.

i.e. $\left\| \frac{\partial E}{\partial z^{(i)}} \right\|^2 =$



5 layers
layer 4 norm=0.7598 [0.76]

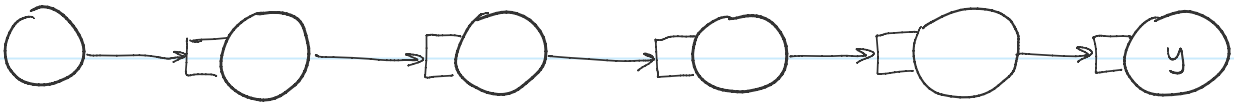
layer 3 norm=0.2266 [-0.038 -0.217 -0.039 -0.039]

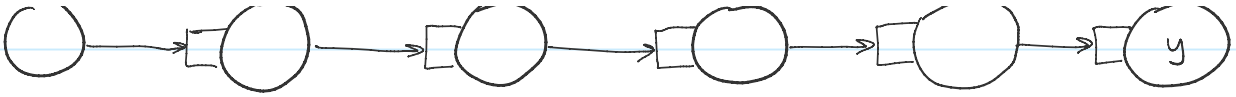
layer 2 norm=0.0316 [0.018 -0.018 -0.007 0.018]

layer 1 norm=0.0094 [-0.004 0.002 0.001 0.008]

Exploding Gradients

A similar, though less frequent phenomenon can result in very large gradients.





This situation is more rare since it only occurs when the weights are high and the biases compensate so that the input current lands in the sweet spot of the logistic curve.

Enhancing Optimization

Goal: To learn some methods that help learning go faster.

Suppose our training set is

where $\vec{x} \in \mathbb{R}^X$, $\vec{t} \in \mathbb{R}^Y$

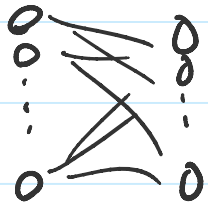
Notice that we can put all of our inputs into a single matrix

$X =$

As well as our targets

$T =$

Does this help us? Yes!
Consider the 1st hidden layer...



$\vec{z}^{(1)} = W\vec{x} + \vec{b}$

Then $\vec{h}^{(1)} = \sigma(\vec{z}^{(1)})$

But we can process all P inputs at once!

$Z^{(1)} =$

Then, $H^{(1)} =$

At the top layer, we get Y.

$E(Y, T) =$

Now, working our way back down,

$$\frac{\partial E}{\partial z^{(k+1)}} =$$

And going down one layer

$$\frac{\partial E}{\partial z^{(k)}} =$$

Then,

$$\frac{\partial E}{\partial W^{(k)}} =$$

Hence, we can use the same formulas to process a whole batch of samples.

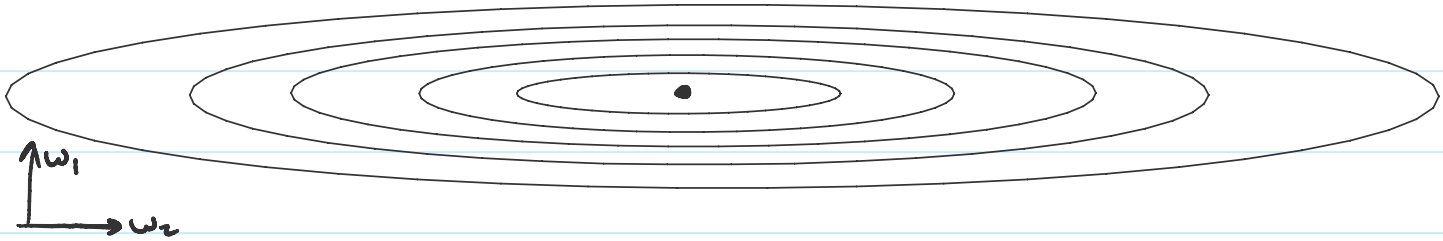
One problem: processing the entire dataset for a single update to the weights can be slow. Instead, there is an intermediate approach.

Stochastic Gradient Descent

Rather than processing the entire dataset, or just one sample, we can process chunks of our samples, randomly chosen, to determine our weight updates. We call these chunks "mini-batches". This approach is more stable than single-sample updates, but faster than full-dataset updates.

Momentum

Consider gradient descent optimization in this situation...



Instead, we can smooth out our trajectory using
Recall from physics,

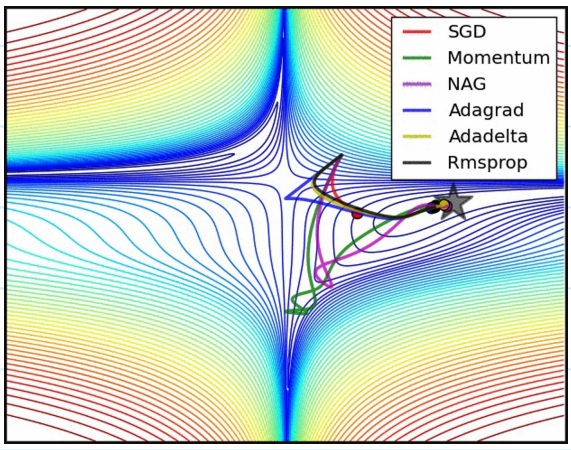
In our previous optimization method, we used our error gradients like , and represented our weights. Then we updated our weights using

But we can instead treat our error gradients as , and integrate to get an accumulated weight update, akin to
In fact, let's call it

Or, as is commonly used,

Then, update our weights using

Not only does this smooth out oscillations, but can also help to avoid getting stuck in local minima.



http://ruder.io/optimizing-gradient-descent/