Introduction to neural networks

- Analogy to neuroscience
- Caution when comparing to neuroscience
- Feedforward neural network architectures
- The importance of nonlinearity
- Example: XOR
- Activation functions
- Output units

Neuroscience

Neural networks, as their name suggests, take their inspiration from neurons. However, the connection is very loose. There are many known aspects of neural signaling that are not incorporated into artificial neural networks. That said, it is accurate to state that the basic computing principle for artificial neural networks is inspired by neuroscience.

(A plug: if you're interested in learning more about signaling in the brain, and how we can analyze and decode neural activity, consider taking EE C143A/C243A, "Neural signal processing and machine learning," which I teach in the Spring quarter. The two course numbers are for undergraduate/graduate. There, we'll do a more proper treatment of signaling in the nervous system.)

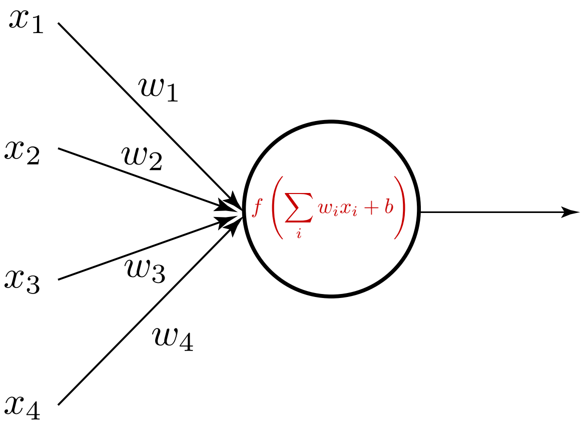
A 10,000 foot view of how neurons work

At a very high level, neurons in the brain communicate via discrete impulses called *action potentials.*

- Action potentials are the fundamental currency of information in the brain.
- Action potentials are "all-or-nothing" signals that either occur or do not occur.
- Action potentials are "fired" when the neuron has received sufficient input.
- The neuron then transmits this action potential to other neurons it is connected to.

Information may be encoded in the *rate* or *timing* of action potentials.

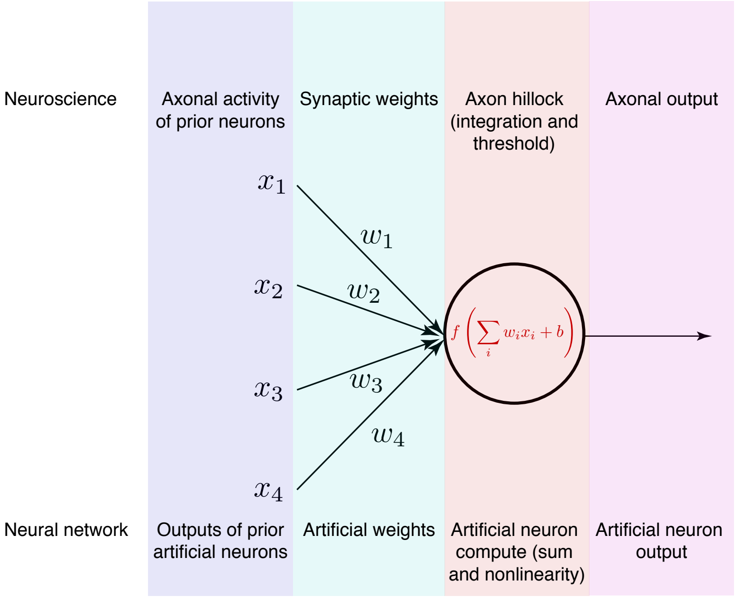
The artificial neuron



The artificial neuron (cont.)

- The incoming signals, a vector $\mathbf{x} \in \mathbb{R}^N$, reflects the output of N neurons that are connected to the current artificial neuron.
- The incoming signals, $\mathbf{x}$, are pointwise multiplied by a vector, $\mathbf{w} \in \mathbb{R}^N$. That is, we calculate $w_i x_i$ for $i = 1, \ldots, N$. This computation reflects dendritic processing.
- The "dendritic-processed" signals are then summed, i.e., we calculate $\sum_i w_i x_i + b$. This computation reflects integration at the axon hillock (the first "Node of Ranvier") where action potentials are generated if the integrated signal is large enough.
- The output of the artificial neuron is then a nonlinearly transformation of the integrated signal, i.e., $f(\sum_i w_i x_i + b)$. Rather than reflecting whether an action potential was generated or not (which is a noisy process), this nonlinear output is typically treated as the *rate* of the neuron. The higher the rate, the more likely the neuron is to fire action potentials.

The artificial neuron (cont.)



Neuroscience	Axonal activity of prior neurons	Synaptic weights	Axon hillock (integration and threshold)	Axonal output
Neural network	Outputs of prior artificial neurons	Artificial weights	Artificial neuron compute (sum and nonlinearity)	Artificial neuron output

Caution when comparing to biology

These computing analogies are not precise, with large approximations. Limitations in the analogy include:

- Synaptic transmission is probabilistic, nonlinear, and dynamic.
- Dendritic integration is probabilistic and may be nonlinear.
- Dendritic computation has associated spatiotemporal decay.
- Integration is subject to biological constraints; for example, ion channels (which change the voltage of the cell) undergo refractory periods when they do not open until hyperpolarization.
- Different neurons may have different action potential thresholds depending on the density of sodium-gated ion channels.
- Feedforward and convolutional neural networks have no recurrent connections.
- Many different cell types.
- Neurons have specific dynamics that can be modulated by e.g., calcium concentration.
- And so many more...

Caution when comparing to biology

On the prior list, several of these bullet points constitute entire research areas. E.g., several labs work specifically on studying the details of synaptic transmission.

Big picture: though neural networks are inspired by biology, they approximate biological computation at a fairly crude level. These networks ought not be thought of as models of the brain, although recent work (including my research group's work) has used them as a means to propose mechanistic insight into neurophysiological computation.

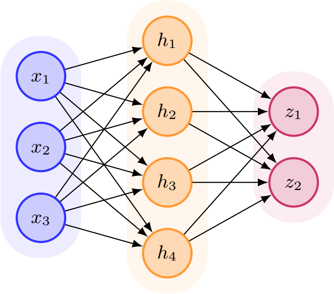
Nomenclature

Some naming conventions.

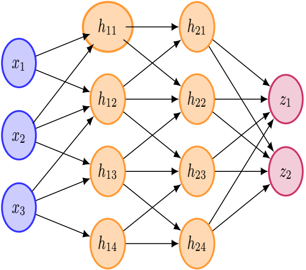
- We call the first layer of a neural network the "input layer." We typically represent this with the variable $\mathbf{x}$.
- We call the last layer the "output layer." We typically represent this with the variable $\mathbf{z}$. (Note: why not $\mathbf{y}$ to match our prior nomenclature for the supervised outputs? Because the output of the network may be a processed version of $\mathbf{z}$, e.g., $\mathrm{softmax}(\mathbf{z})$.)
- We call the intermediate layers the "hidden layers." We typically represent this with the variable $\mathbf{h}$.
- When we specify that a network has N layers, this does not include the input layer.

Neural network architecture

An example 2-layer network is shown below.



Here, the three dimensional inputs ($\mathbf{x} \in \mathbb{R}^3$) are processed into a four dimensional intermediate representation ($\mathbf{h} \in \mathbb{R}^4$), which are then transformed into the two dimensional outputs ($\mathbf{z} \in \mathbb{R}^2$).


x_1
x_2
x_3
h_{11}
h_{12}
h_{13}
h_{14}
h_{21}
h_{22}
h_{23}
h_{24}
z_1
z_2

Neural networks

The above figure suggests the following equation for a neural network.

- Layer 1: $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$
- Layer 2: $\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

Any composition of linear functions can be reduced to a single linear function. Here, $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where

$$\mathbf{W} = \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{W}_1$$

and

$$\mathbf{b} = \mathbf{b}_N + \mathbf{W}_N\mathbf{b}_{N-1} + \cdots + \mathbf{W}_N \cdots \mathbf{W}_3\mathbf{b}_2 + \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{b}_1$$

- This may be useful in some contexts. For example, when $\dim(\mathbf{h}) \ll \dim(\mathbf{x})$, this corresponds to finding a low-rank representation of the inputs.
- However, a system with greater complexity may require a higher capacity model.

Introducing nonlinearity

To increase the network capacity, we can make it nonlinear. We do this by introducing a nonlinearity, $f(\cdot)$, at the output of each artificial neuron.

- Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

A few notes:

- These equations describe a *feedforward neural network*, also called a *multilayer perceptron*.
- $f(\cdot)$ is typically called an *activation function* and is applied elementwise on its input.
- The activation function does not typically act on the output layer, $\mathbf{z}$, as these are meant to be interpreted as scores. Instead, separate "output activations" are used to process $\mathbf{z}$. While these output activations may be the same as the activation function, they are typically different. For example, it may comprise a softmax or SVM classifier.

Example: XOR

Consider a system that produces training data that follows the $\mathrm{xor}(\cdot)$ function. The xor function accepts a 2-dimensional vector $\mathbf{x}$ with components x_1 and x_2 and returns 1 if $x_1 \neq x_2$. Concretely,

x_1	x_2	$\mathrm{xor}(\mathbf{x})$
0	0	0
0	1	1
1	0	1
1	1	0

Our goal is to mimic this function by minimizing the mean-square error of a predicted output, $g(\mathbf{x})$ to the true output, $\mathrm{xor}(\mathbf{x})$. (Here, $\mathbf{x}$ is a vector with two elements containing the two input values to xor.) If the model has parameters θ, the loss function is:

$$J(\theta) = \frac{1}{2} \sum_{\mathbf{x}} (g(\mathbf{x}) - y(\mathbf{x}))^2$$

(Note, we wouldn't know $\mathrm{xor}(\mathbf{x})$, but we would have samples of corresponding inputs and outputs from training data. Hence, it may be better to simply replace $\mathrm{xor}(\mathbf{x})$ with $y(\mathbf{x})$ representing training examples.)

Example: XOR

Consider first a linear approximation of xor, via $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. Then,

$$\begin{aligned}\frac{\partial J(\mathbf{w}, b)}{\partial \mathbf{w}} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\mathbf{x} \\ \frac{\partial J(\mathbf{w}, b)}{\partial b} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\end{aligned}$$

Equating these to 0, we arrive at:

$$(w_1 + b - 1)\begin{bmatrix} 1 \\ 0 \end{bmatrix} + (w_2 + b - 1)\begin{bmatrix} 0 \\ 1 \end{bmatrix} + (w_1 + w_2 + b)\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

These two equations can be simplified as:

$$\begin{aligned}(w_1 + b - 1) + (w_1 + w_2 + b) &= 0 \\ (w_2 + b - 1) + (w_1 + w_2 + b) &= 0\end{aligned}$$

These equations are symmetric, implying $w_1 = w_2 = w$. This means:

$$3w + 2b - 1 = 0 \implies b = \frac{1 - 3w}{2}$$

Example: XOR

We also have our equation from setting the derivative w.r.t. b equal to 0, and substituting $w_1 = w_2 = w$, we have

$$4w + 4b - 2 = 0$$

Solving these equations, we arrive at $w_1 = w_2 = 0$ and $b = 1/2$.

Hence, the optimal linear mapping will always output $g(\mathbf{x}) = 1/2$, which is not a great solution. This is because a linear classifier cannot perform the $\mathrm{xor}(\cdot)$ operation.

Example: XOR

Now let's consider using a two-layer neural network, with the following equation:

$$g(\mathbf{x}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

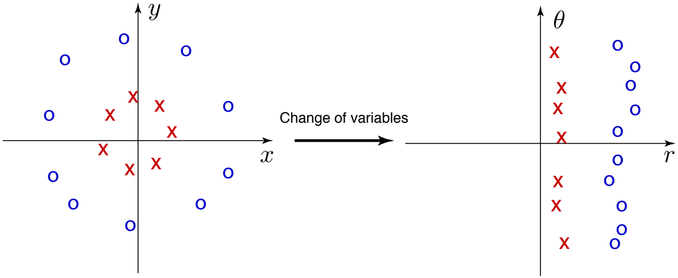
We haven't yet discussed how to optimize these parameters, but the point here is to show that by introducing a simple nonlinearity like $f(x) = \max(0, x)$, we can now solve the $\mathrm{xor}(\cdot)$ problem. Consider the solution:

$$\begin{aligned} \mathbf{W} &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ \mathbf{c} &= [0, -1]^T \\ \mathbf{w} &= [1, -2]^T \end{aligned}$$

One can check that this solution correctly replicates the behavior of $\mathrm{xor}(\cdot)$.

A perspective on feature learning

One area of machine learning is very interested in finding *features* of the data that are then good for use as the input data to a classifier (like a SVM). Why might this be important?



The intermediate layers of the neural network (i.e., $\mathbf{h}_1, \mathbf{h}_2$, etc.) are features that the later layers then use for decoding. If the performance of the neural network is well, these features are good features.

Importantly, these features don't have to be handcrafted.

The name "deep learning"

The term "deep learning" is related to the depth of the model (i.e., how many layers it has). There isn't an agreed upon definition of when a network goes from "shallow" to "deep," although Schmidhuber suggests that 10 layers is "very deep."

Activation functions

Common activation functions include (put in figures for each of these):

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $\tanh(x)$, or equivalently $2\sigma(2\mathbf{x}) - 1$.
- Rectified linear unit (aka ReLU): $\mathrm{ReLU}(x) = \max(0, x)$.
- Variations on the ReLU: $\max(0, x) + \alpha_i \min(0, x)$.
  - When $\alpha_i = -1$, this is the absolute value rectifier.
  - When α_i is small, this is the leaky ReLU.
  - When α_i is a hyperparameter, this is called parametric ReLU (or PReLU).
- Maxout: $f(\mathbf{x})_i = \max_j(\mathbf{w}_j^T x + b_j)$ for $j \in \mathcal{G}^{(i)}$
  - Here, $\mathcal{G}^{(i)}$ refers to the set of indices to which i is a member. All indices in this group are assigned the same value.
  - The maxout generalizes the ReLU unit.

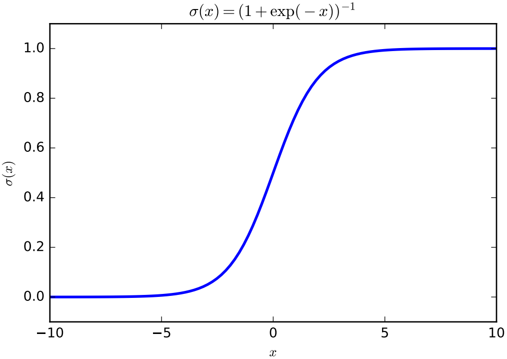
Other activation functions exist, but these are more commonly encountered.

Which activation function should I use?

Activation function choice is an active area of research. Some activation functions are more commonly used than others, although the particular application may influence the activation function chosen. We'll briefly discuss pros and cons of some of these functions. However, some takehome points are:

- It is difficult to determine what activation functions is best to use in a given scenario. The design process consists largely of trial and error.
- A common choice is to use the ReLU; this enjoys popular use in convolutional neural networks.
- It is rare to ever use the sigmoid nonlinearity; the hyperbolic tangent is preferred.

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$



Its derivative is:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.

Cons:

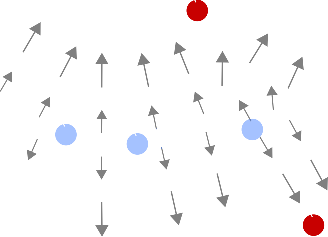
- At extremes, the unit *saturates* and thus has zero gradient. This results in no learning with gradient descent.
- The sigmoid unit is not zero-centered; rather its outputs are centered at 0.5.

Consider $f(\mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$. Defining $z = \mathbf{w}^T\mathbf{x} + b$, the derivative with respect to $\mathbf{w}$, the parameters, is:

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \sigma(z)(1 - \sigma(z))\mathbf{x}$$

If $\mathbf{x} \geq 0$ (e.g., if the input units all had a sigmoidal output), then the gradient has all positive entries. Let's say we had some gradient, $\frac{\partial \mathcal{L}}{\partial f}$, which can be positive or negative. Then $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ will have all positive or negative entries.

This can result in zig-zagging during gradient descent.



Hyperbolic tangent, $\tanh(x) = 2\sigma(x) - 1$

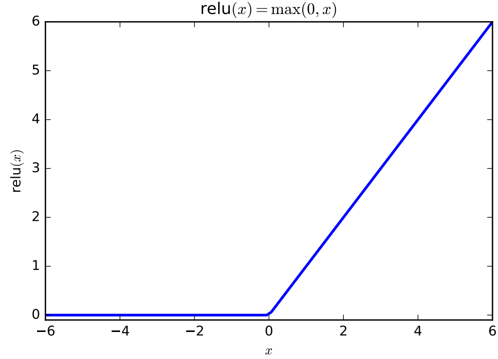
Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.
- It is zero-centered.

Cons:

- Like the sigmoid unit, when a unit saturates, i.e., its values grow larger or smaller, the unit saturates and no additional learning occurs.
- The gradient is subject to second order effects; as $|x|$ grows, the gradient decreases to zero. This makes the gradient smaller at larger $|x|$, which may distort the gradient.

Rectified linear unit, $\mathrm{ReLU}(x) = \max(0, x)$



Its derivative is:

$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

This function is not differentiable at $x = 0$. However, we can define its subgradient by setting the derivative to be between $[0, 1]$ at $x = 0$.

Rectified linear unit, $\mathbf{ReLU}(x) = \max(0, x)$

Pros:

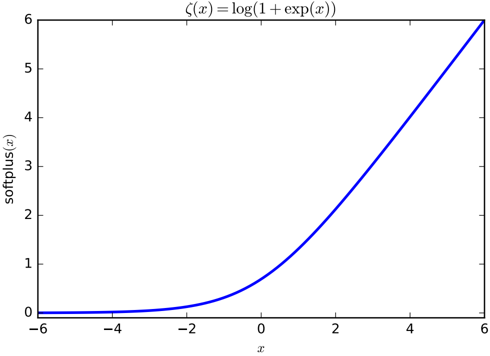
- In practice, learning with the ReLU unit converges faster than sigmoid and tanh.
- When a unit is active, it behaves as a linear unit.
- The derivative at all points, except $x = 0$, is 0 or 1. When $x > 0$, the gradients are large, and not scaled by second order effects.
- There is no saturation if $x > 0$.

Cons:

- $\mathrm{ReLU}(x)$, like sigmoid, is not zero-centered.
- $\mathrm{ReLU}(x)$ is not differentiable at $x = 0$. However, in practice, this is not a large issue. A heuristic when evaluating $\left.\frac{d\mathrm{ReLU}(x)}{dx}\right|_{x=0}$ is to return the left derivative (0) or the right derivative (1); this is reasonable given digital computation is subject to numerical error.
- Learning does not happen for examples that have zero activation. This can be fixed by e.g., using a leaky ReLU or maxout unit. For this reason, it is also typical to initialize the biases to be slightly positive, so that all neurons start off active.

Softplus unit, $\zeta(x) = \log(1 + \exp(x))$

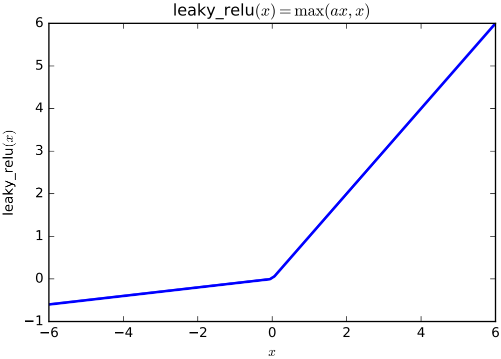
One may consider using the softplus function, $\zeta(x) = \log(1 + e^x)$, in place of $\mathrm{ReLU}(x)$. Intuitively, this ought to work well as it resembles $\mathrm{ReLU}(x)$ and is differentiable everywhere. However, empirically, it performs worse than $\mathrm{ReLU}(x)$.



Its derivative is:

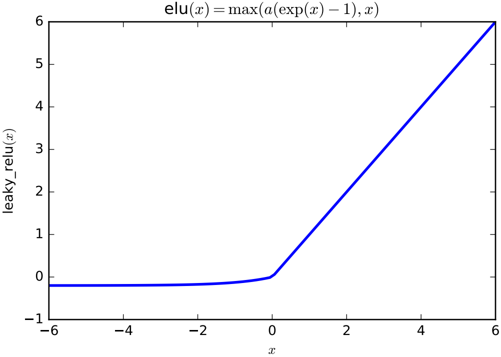
$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

Leaky rectified linear unit, $f(x) = \max(\alpha x, x)$



The leaky ReLU avoids the stopping of learning when $x < 0$. α may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it is typically called the "PReLU" for parametrized rectified linear unit.

Exponential linear unit, $f(x) = \max(\alpha(\exp(x) - 1), x)$



The exponential linear unit avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.

Maxout unit

A generalization of the ReLU and PReLU units is the maxout unit, where:

$$\mathrm{maxout}(\mathbf{x}) = \max(\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2)$$

This can be generalized to more than two components. If $\mathbf{w}_1 = \mathbf{0}$ and $b_1 = 0$, this is the rectified linear unit.

- This unit does not saturate.
- The unit is a composition of linear functions.
- However, it at least doubles the number of parameters in the model.

In practice...

In practice...

- The ReLU unit is very popular.
- The sigmoid unit is almost never used; tanh is preferred.
- It may be worth trying out leaky ReLU / PReLU / ELU / maxout for your application.
- This is an active area of research.

What outputs and cost functions?

There are several options to process the output scores, $\mathbf{z}$, to ultimately arrive at a cost function. The choice of output units interacts with what cost function to use.

Example: Consider a neural network that produces a single score, z, for binary classification. As the output unit, we choose the sigmoid nonlinearity, so that $\hat{y} = \sigma(z)$. On a given example, $y^{(i)}$ is either 0 or 1, and $\hat{y}^{(i)} = \sigma(z^{(i)})$ can be interpreted as the algorithm's probability the output is in class 1. Is it better to use mean-square error or cross-entropy (i.e., corresponding to maximum-likelihood estimation) as the cost function? For n examples:

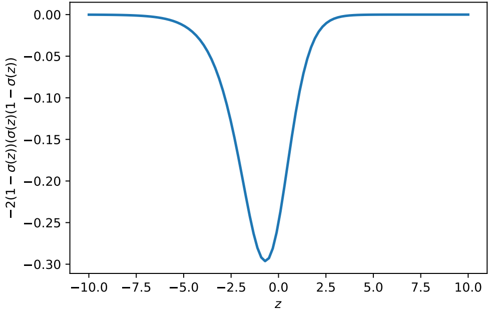
$$
\begin{aligned}
\text{MSE} &= \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2 \\
\text{CE} &= -\sum_{i=1}^{n}\left[y^{(i)}\log\sigma(z^{(i)}) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right]
\end{aligned}
$$

What outputs and cost functions? (cont.)

Example (cont): Consider just one example, where $y^{(i)} = 1$. For this example,

$$\frac{\partial \text{MSE}}{\partial z^{(i)}} = -2(y^{(i)} - \sigma(z^{(i)}))(\sigma(z^{(i)})(1 - \sigma(z^{(i)})))$$

This derivative looks like the following:



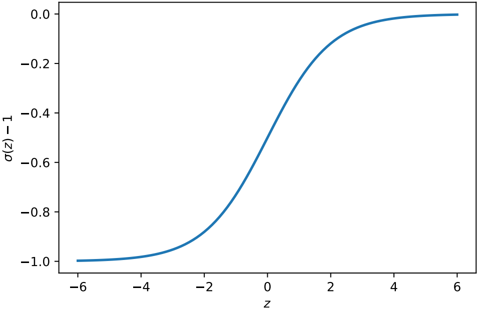
When z is very negative, indicating a large MSE, the gradient saturates to zero, and no learning occurs.

What outputs and cost functions? (cont.)

Example (cont): Now let's consider the cross-entropy. For one example, where $y^{(i)} = 1$,

$$\frac{\partial \mathrm{CE}}{\partial z^{(i)}} = \sigma(z^{(i)}) - 1$$

This derivative looks like the following:



Notice that when z is very negative, learning will occur, and it will only "stall" when z gets close to the right answer.

Other outputs

Other potential output functions include:

- Linear output units: $\hat{\mathbf{y}} = \mathbf{z}$.

  These output units typically specify the conditional mean of a Gaussian distribution, i.e.,

  $$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{z}, \mathbf{I})$$

  and in this case, MLE estimation is equivalent to minimizing squared error.

- Sigmoid outputs: $\hat{\mathbf{y}} = \sigma(\mathbf{z})$.

  These outputs are typically used in binary classification to approximate a Bernoulli distribution.

  Question: Why not use the following output?

  $$\Pr(y = 1|\mathbf{x}) = \max\left(0, \min\left(1, \mathbf{z}\right)\right)$$

  (Any z_i outside of the interval $[0, 1]$ has zero gradient, meaning no learning.)

Other outputs

- Softmax output: $\hat{\mathbf{y}}_i = \mathrm{softmax}_i(\mathbf{z})$.

  The softmax is the generalization of the sigmoid output to multiple classes. One can then implement a softmax classifier at the output.

- It is also possible to use an SVM at the output, with the features $\mathbf{z}$ being the input to the SVM.

Universal function approximators

Feedforward neural networks are universal function approximators, meaning they can approximate any function arbitrarily closely. More formally, given a function $f(\mathbf{x})$ and some $\epsilon > 0$, there is a one layer neural network $g(\mathbf{x})$ that achieves $|f(\mathbf{x}) - g(\mathbf{x}) < \epsilon|$. The proof of this is beyond the scope of this class, and isn't relevant for practical use. However, it's a handy fact to know.