ML

2018
NS

Artificial Neural Networks

Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks
Artificial Neural Networks

Dogs and cats are both furry animals with four legs and many other shared traits. Why, then, is it easy to distinguish between them?

As young people, we're told which animals we observe are dogs and which are cats. Fairly quickly, we stop needing new examples. Our learning is powerful enough to classify a new animal as a dog or a cat, even when it doesn't look particularly similar to one we've seen before. It turns out that computers can learn similarly.

A **supervised learning** algorithm attempts to model a function to relate inputs to outputs. It uses known examples to learn this relationship.

When building a supervised learning model to distinguish whether an image is of a dog or a cat, what should the inputs for the examples be?

- The fur and eye colors of dogs and cats
- The lengths and weights of dogs and cats
- Numerical data representing images of dogs and cats

A **supervised learning** algorithm attempts to model a function to relate inputs to outputs. It uses known examples to learn this relationship.

When building a supervised learning model to distinguish whether an image is of a dog or a cat, what should the inputs for the examples be?

- ○ The fur and eye colors of dogs and cats
- ○ The lengths and weights of dogs and cats
- ○ Numerical data representing images of dogs and cats

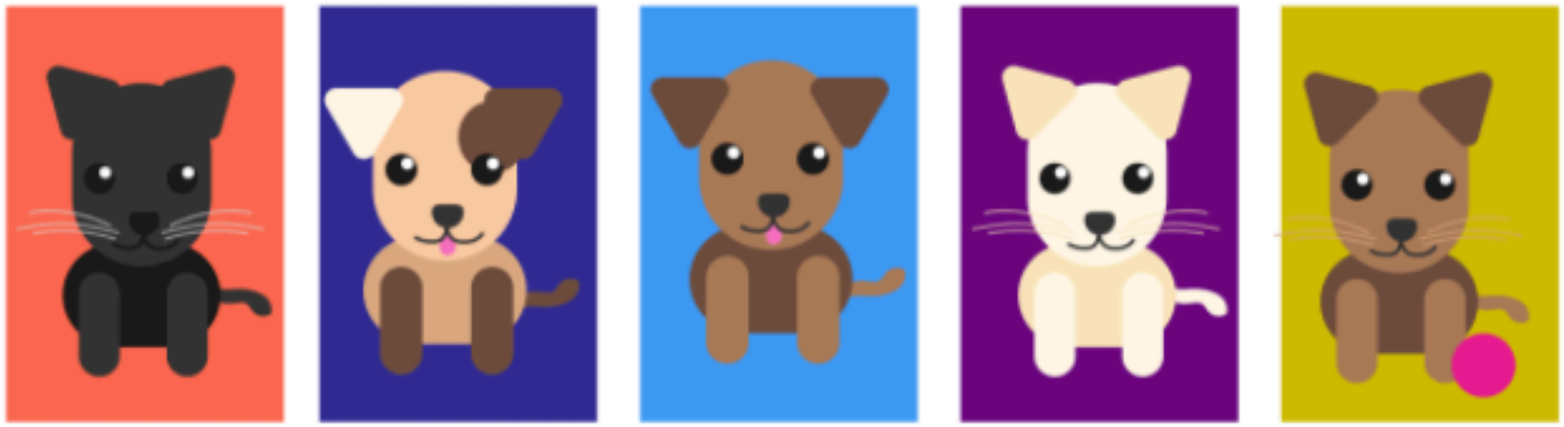
Explanation

Correct answer: **Numerical data representing images of dogs and cats**

If you want to identify if an animal is a dog or a cat, it's only natural that your input would be images of dogs and cats. Typically, this data is represented by a matrix of numerical values that correspond to each pixel in an image.

Furthermore, this data will be the most useful for training the model because it will allow a computer to learn similarly to humans, by "seeing" the images and finding patterns between them. Of course, a computer will need more examples than a human to be able to successfully distinguish between most dogs and cats.

LEARNING PROBLEMS FOR NEURAL NETWORKS

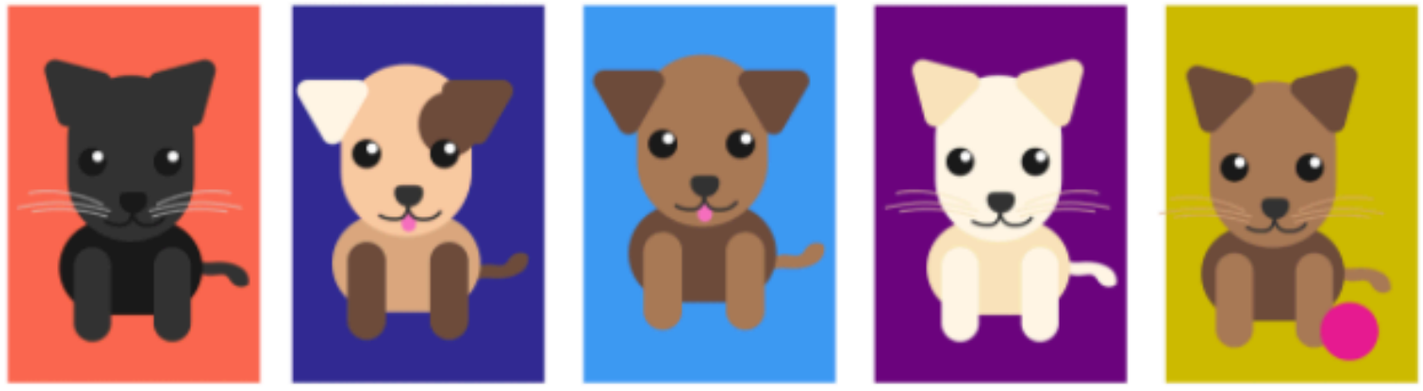


Suppose you have access to 100,000 images of dogs and cats that you can use to build a supervised learning model that distinguishes between dogs and cats.
After using images as examples to **train** (or teach) the model, you'll want to use images to **test** the model; that is, to determine if the model is actually successful at identifying if the image is of a dog or a cat.

What would be a reasonable way to select your images for training and testing?

- ○ Use the dog pictures for training and the cat pictures for testing
- ○ Use the cat pictures for training and the dog pictures for testing
- ○ Split the images randomly into two sets: one for training, one for testing
- ○ Use all of the images in both training and testing

LEARNING PROBLEMS FOR NEURAL NETWORKS



Suppose you have access to 100,000 images of dogs and cats that you can use to build a supervised learning model that distinguishes between dogs and cats.
After using images as examples to **train** (or teach) the model, you'll want to use images to **test** the model; that is, to determine if the model is actually successful at identifying if the image is of a dog or a cat.

What would be a reasonable way to select your images for training and testing?

- Use the dog pictures for training and the cat pictures for testing
- Use the cat pictures for training and the dog pictures for testing
- ✓ Split the images randomly into two sets: one for training, one for testing
- Use all of the images in both training and testing

Explanation

Correct answer: **Split the images randomly into two sets: one for training, one for testing**

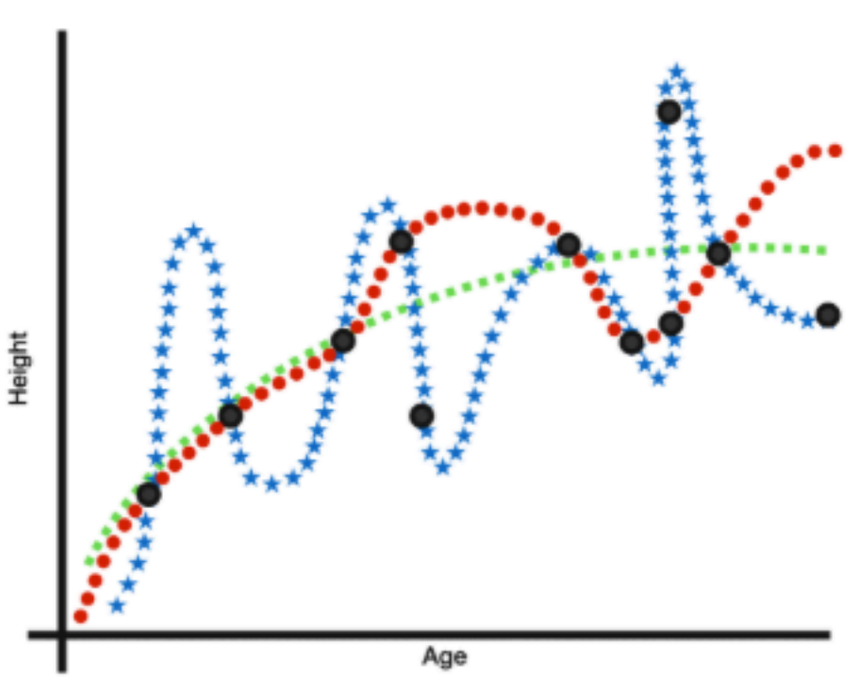
The end goal of supervised learning is to find a function that models the relationship between the inputs and outputs, which means that it should be able to predict an output given a new input which it has never seen before. This is known as **generalization**.

Thus, we need to evaluate how well the model and learning algorithm generalizes the observed data to the unobserved data. One simple way to do this is to split the input data into two sets, one set to train the model on and another to evaluate how well it generalizes. The first set is known as the training data, since it is what the model uses to train. The second set is known as the test data, since it is used to evaluate the model and learning algorithm's ability to generalize.

How to actually split this data (i.e., what proportion of the data to save for testing) depends on the problem at hand and the data available.

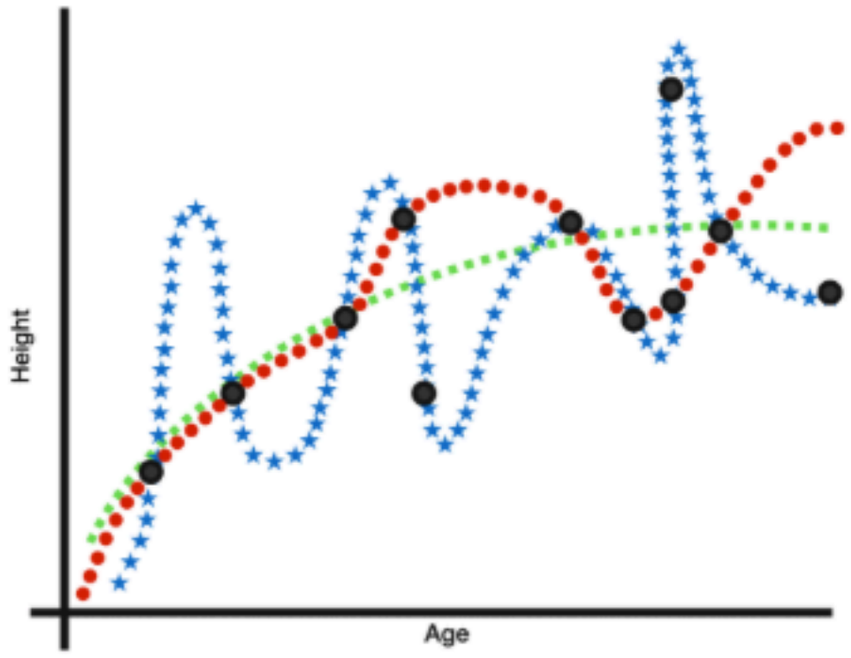
A good supervised learning model predicts the outputs of unobserved inputs using knowledge of the outputs of observed inputs. The ability to make successful predictions on unobserved inputs from observed data is called **generalization**.

For any observed data, there are an infinite number of functions that pass through all input-output pairs. The "best" function is *not* necessarily one which fits all of the observed data, but instead is one that generalizes well.



- The blue model (stars)
- The green model (dashes)
- The red model (circles)

You are training a height-prediction model using observed inputs of children's ages and outputs of their heights, shown as points in the graph above. Which of the three functions drawn is likely to be the best model?



- ○ The blue model (stars)
- ○ The green model (dashes)
- ○ The red model (circles)

You are training a height-prediction model using observed inputs of children's ages and outputs of their heights, shown as points in the graph above. Which of the three functions drawn is likely to be the best model?

Correct answer: **The green model (dashes)**

Given just the points, we would be more likely to draw a curve resembling the green one than the blue or red ones--especially given our intuition about how age and height should relate to each other. In particular, the sharp changes in predicted height for similar age values in the blue and red models do not make sense.

Typically, a simpler function is more likely to avoid **overfitting** (making a model that is needlessly complex) and therefore better generalize.

In the task of distinguishing between dogs and cats, we wanted to classify an image into discrete categories with no numerical relationship; i.e., we cannot say dog is 2 times cat. This type of problem is called a **classification** problem.

On the other hand, in the previous question, we found a function to relate an input to a numerical output (height). These outputs have a clear numerical relationship; e.g., the output of 6 feet is twice the output of 3 feet. This type of problem is known as a **regression** problem.

Artificial neural networks (ANNs) are flexible enough to be used in both classification and regression problems.

Of the following three learning problems, **how many** should be treated as **regression** problems?

- Identifying which zip-code digits have been written on an envelope
- Predicting the total number of points scored by two teams in a basketball game
- Determining the adult height of a child given his/her age and current height

- None
- 1
- 2
- 3

Of the following three learning problems, **how many** should be treated as **regression** problems?

- Identifying which zip-code digits have been written on an envelope
- Predicting the total number of points scored by two teams in a basketball game
- Determining the adult height of a child given his/her age and current height

- None
- 1
- 2
- 3

Correct answer: **2**

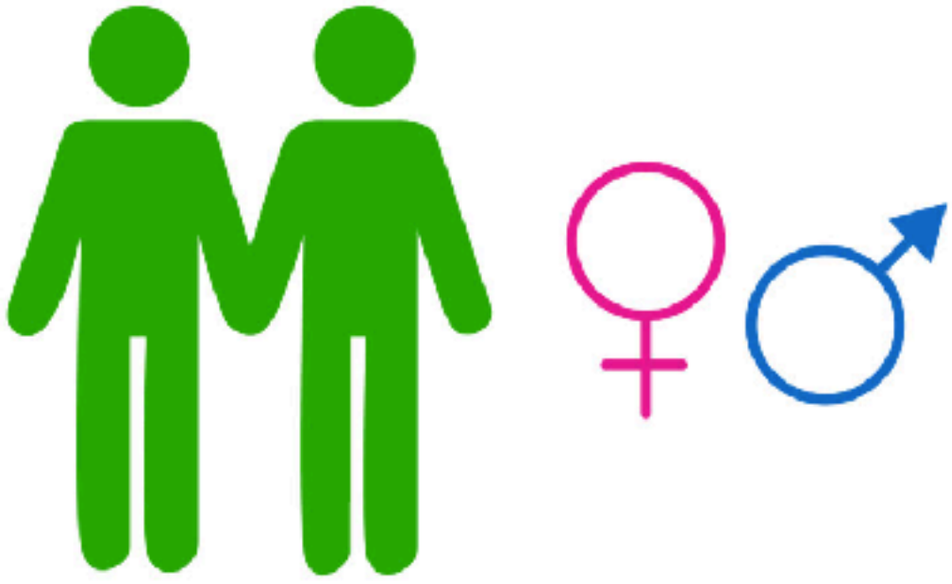
While zip-code digits are numbers, the outputs are not actually related numerically - they are simply symbolic characters. This is therefore a classification problem with many possible outputs and not a regression problem.

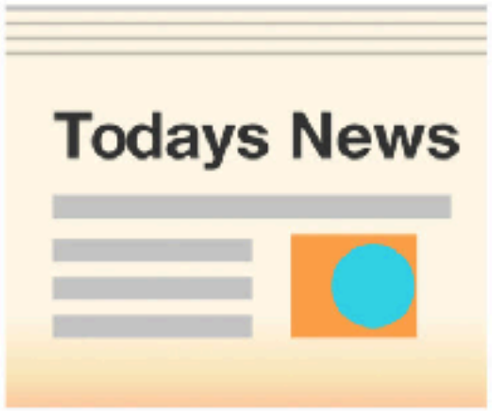
The other two problems should be treated as regression problems. Height is a continuous variable (even though we sometimes approximate it with discrete values). While the number of points scored in a basketball game is a positive integer, a good algorithm would treat this as a regression problem since the outputs are numerically related (e.g., 120 points is 30 fewer than 150 points, and is twice 60 points).

LEARNING PROBLEMS FOR NEURAL NETWORKS

This quiz has focused on supervised learning problems, as many of the basic applications of ANNs involve supervised learning. However, there is another type of learning: **unsupervised learning**.

As the name might imply, unsupervised learning attempts to determine relationships between inputs without using any example outputs (such as “dog” or “cat” in our earlier example). Which of the following problems would **not** be a good fit for unsupervised learning?





- Determining possible friendship matches from interest profiles
- Predicting someone’s gender from their name
- Grouping news articles about similar topics

LEARNING PROBLEMS FOR NEURAL NETWORKS

This quiz has focused on supervised learning problems, as many of the basic applications of ANNs involve supervised learning. However, there is another type of learning: **unsupervised learning**.

As the name might imply, unsupervised learning attempts to determine relationships between inputs without using any example outputs (such as “dog” or “cat” in our earlier example). Which of the following problems would **not** be a good fit for unsupervised learning?

- ○ Determining possible friendship matches from interest profiles
- ○ Predicting someone’s gender from their name
- ○ Grouping news articles about similar topics

Correct answer: **Predicting someone’s gender from their name**

Finding friendship matches and grouping news articles are both types of an unsupervised learning problem known as **clustering**. Similar inputs (e.g., interest information or words in news articles) can be grouped together to solve these problems without any output information.

On the other hand, it would be hard to predict someone’s gender from their name with unsupervised learning. Even if you instructed an unsupervised learning algorithm to make two groups of “similar” names, it would have a very hard time doing so--it might try to group by length, which would not be useful. Instead, using a supervised learning algorithm by tagging common names as male or female could allow a model to learn some structural similarities between male and female names.

We now have a big-picture sense of what learning problems are all about. There are two main types of learning: unsupervised and supervised, and problems within supervised learning can be categorized as classification or regression problems. This course illustrates how to construct ANNs, a computational model that performs well in a wide variety of learning problems such as the ones discussed in this quiz.

In the next quiz, we'll explore the basics of how the human brain works and how these ideas inspire the mechanisms within ANNs.

When training a learning model, there are two main processes that can be used with respect to how the training data is handled: batch learning and online learning.

In **batch learning**, the model learns from batches of data - often from the entire training set at once. In **online learning**, the model learns from data processed sequentially over time, as it becomes available.

Which type of learning does the human brain mainly use?

Correct answer: **Online learning**

The human brain is able to incrementally improve over time given the data currently available to it. This explains why adults are better at most tasks than children; they've had more time to train than children. This is indicative of online learning. If humans learned by batch learning, we would need to collect huge sets of input-output pairs before learning anything useful.

ANNs can actually use either (or both) methods of learning - there are some tradeoffs which will be discussed later in this course. But, unsurprisingly, online learning has been shown to work better in many cases, such as with backpropagation (a technique discussed later on).

Online learning requires less storage than batch learning because each data point is learned (and then "thrown away") once it is acquired. Batch learning, on the other hand, must store all of its training data (or large chunks) before it can begin learning, so it has higher memory requirements.

Online learning also allows for gradual improvement over time. This is in contrast to batch learning, which shows sudden improvements once enough data is collected and learning can begin. This is especially useful for learning time series data (data arriving sequentially in time).

However, online learning does **not** typically make performance evaluation of the learning model easier. In particular, we often cannot designate some data to be "test data" to test how the model performs after learning on the training data, because the data is unlikely to be independent over time. Also, it's unclear how to weight more recent performance against past performance.

As a real-world analogy, if a child has poor techniques for learning language, it usually would not be noticed until much later; e.g., maybe when he/she first has to write full sentences. At that point, you would have a hard time diagnosing where the learning process went wrong and how to fix it. In batch learning, these issues are less present, since you can easily designate some data to be "test data" and regularly alter the model and simply rerun on your batched data.

For both practical and theoretical reasons, ANNs can actually use either (or both) of online learning and batch learning. Some of these tradeoffs will be explored later in this course.

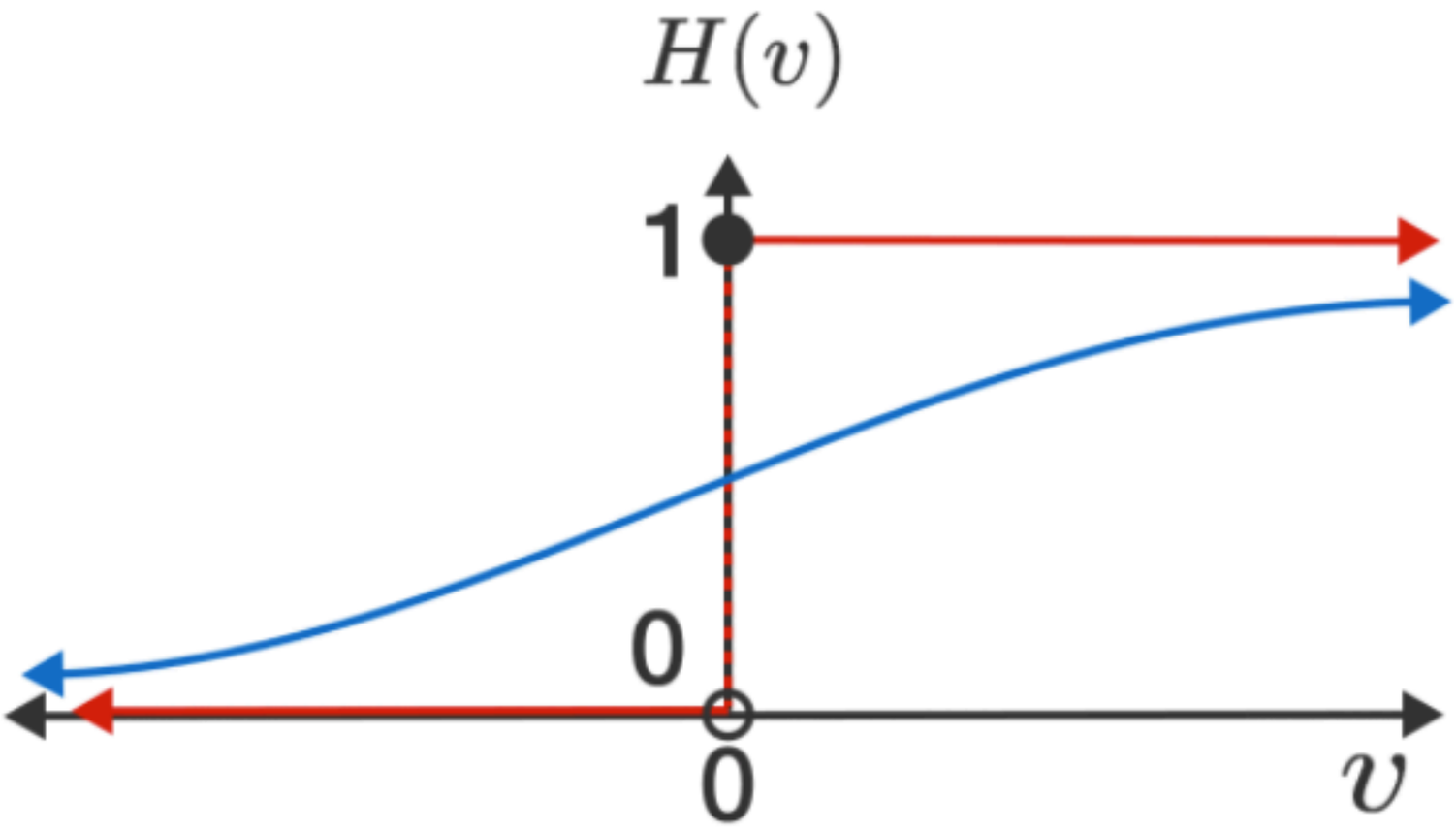
Now that we've discussed how the brain (and ANNs) might import data, what does it actually do with this data to learn? This isn't a neuroanatomy course, so straight to the point: **neurons** are the simplest units of computation in the human brain, and their interactions facilitate brain functions such as learning.

Neurons each compute a simple function. While the actual dynamics of a neuron's computation are complex, a simplified view of them is that they **integrate** and **fire**. That is, a neuron performs a computation with its inputs, and then fires if that computation passes a certain threshold.

COMPUTATIONALLY MODELING THE BRAIN

The basic idea of the neuron model in ANNs is that the inputs to a neuron are combined (as a weighted sum) into a single value, v. Then, an activation function, $H(v)$, is applied to determine whether or not the neuron fires.

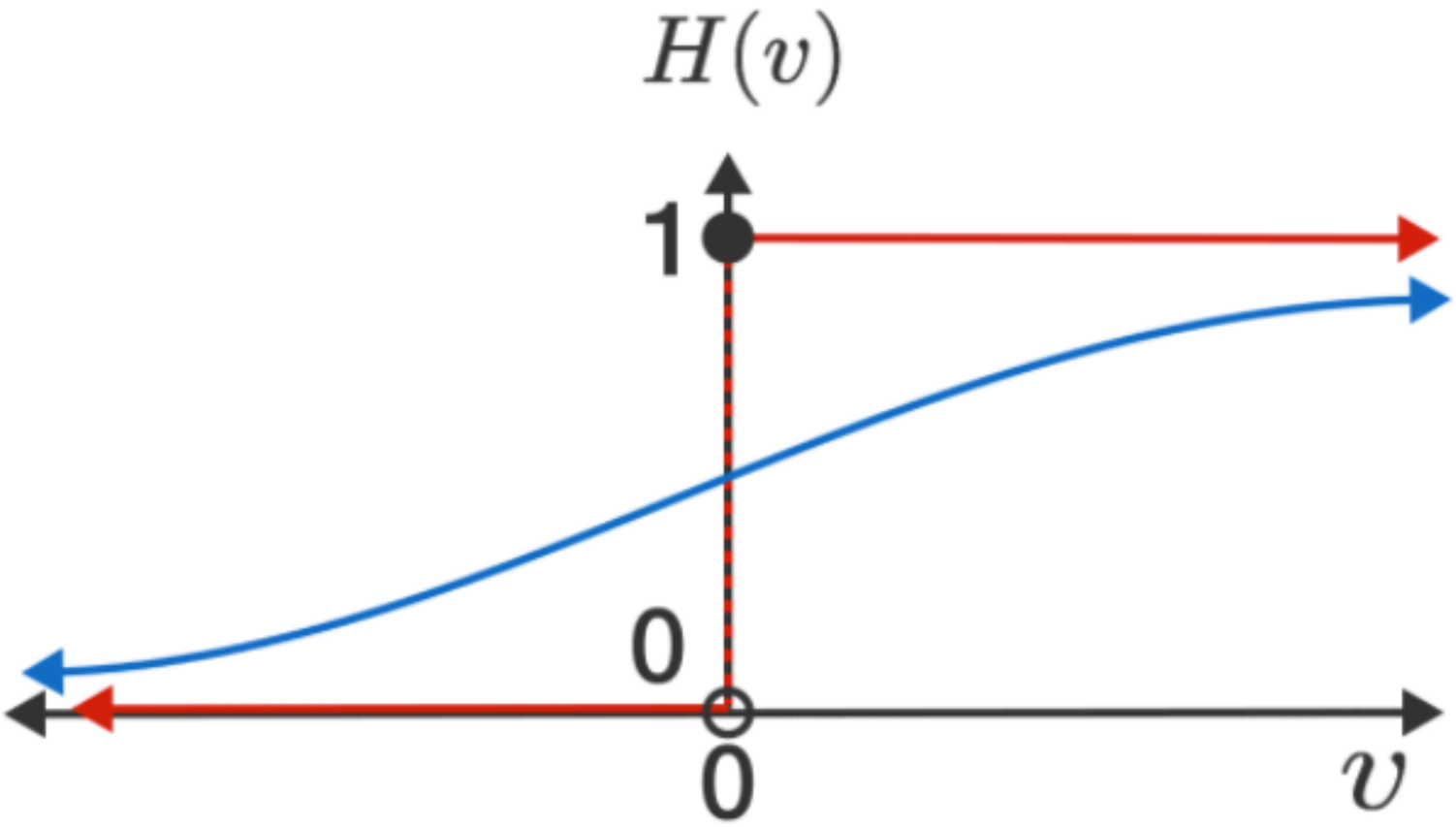
For a physical neuron, the firing is "all-or-nothing"; that is, it fires or it doesn't. Which activation function below would best model this?



- The blue one
- The red one

The basic idea of the neuron model in ANNs is that the inputs to a neuron are combined (as a weighted sum) into a single value, v. Then, an activation function, $H(v)$, is applied to determine whether or not the neuron fires.

For a physical neuron, the firing is "all-or-nothing"; that is, it fires or it doesn't. Which activation function below would best model this?



- The blue one
- The red one

Correct answer: **The red one**

All-or-nothing firing requires a sharp jump in the output for two similar inputs. This is because all-or-nothing firing usually results from a threshold, which determines whether a neuron fires (all) or not (nothing). Thus, right below the threshold, the output will be close to 0 while above the threshold the output will not be close to 0 - here, it is exactly 1 in the red function.

The red function is known as the Heaviside step function.

COMPUTATIONALLY MODELING THE BRAIN

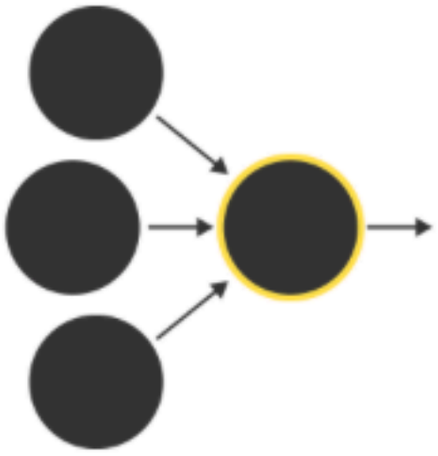
Connections (or synapses) between neurons are used to pass information from the outputs of some neurons to the inputs of other neurons. Not every neuron is connected to every other neuron, and certain neurons have stronger connections to some than others.

Estimates for the number of neurons and synapses in a human brain vary widely, but there are approximately 10^{11} neurons in the human brain and between 10^{14} and 10^{15} synapses. Which of these values is a reasonable estimate for the average number of connections per neuron?

Correct answer: **1,000**

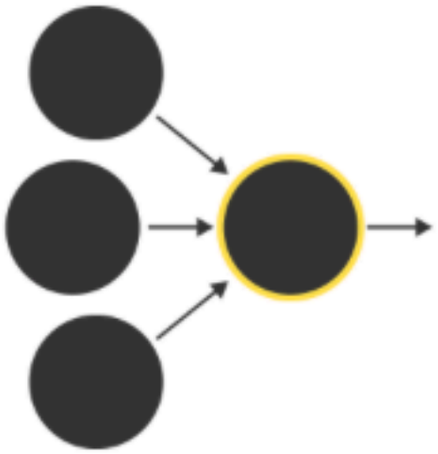
The number of synapses divided by the number of neurons is between $10^3 = 1000$ and $10^4 = 10000$. Whether or not you consider the synapses to be bidirectional, the most reasonable estimate given is 1,000. That's a lot of connections (for an even larger number of neurons), and far beyond what we'll model with ANNs.

By adjusting which connections exist and how strong they are, the human brain is able to learn a huge variety of complex functions. Thus, a computational model influenced by the human brain should include simple computational units (like neurons) which are connected to one another (as with synapses). If the model can learn to adjust the strengths of those connections appropriately, it may be able to approach the power of the human brain.



A neuron has many inputs but only one output, so it must "integrate" its inputs into one output (a single number). Recall that the inputs to a neuron are generally outputs from other neurons. What is the most natural way to represent the set of these inputs to a single neuron in an ANN?

By adjusting which connections exist and how strong they are, the human brain is able to learn a huge variety of complex functions. Thus, a computational model influenced by the human brain should include simple computational units (like neurons) which are connected to one another (as with synapses). If the model can learn to adjust the strengths of those connections appropriately, it may be able to approach the power of the human brain.



A neuron has many inputs but only one output, so it must "integrate" its inputs into one output (a single number). Recall that the inputs to a neuron are generally outputs from other neurons. What is the most natural way to represent the set of these inputs to a single neuron in an ANN?

Correct answer: **A vector**

The inputs to a neuron are the outputs of other neurons, and the output of a neuron is a single number. Thus, a vector (as a column of numbers) makes the most sense. For example, if the outputs of the other neurons were $\sqrt{2}$, 3, and $-\pi$, we could write the input as

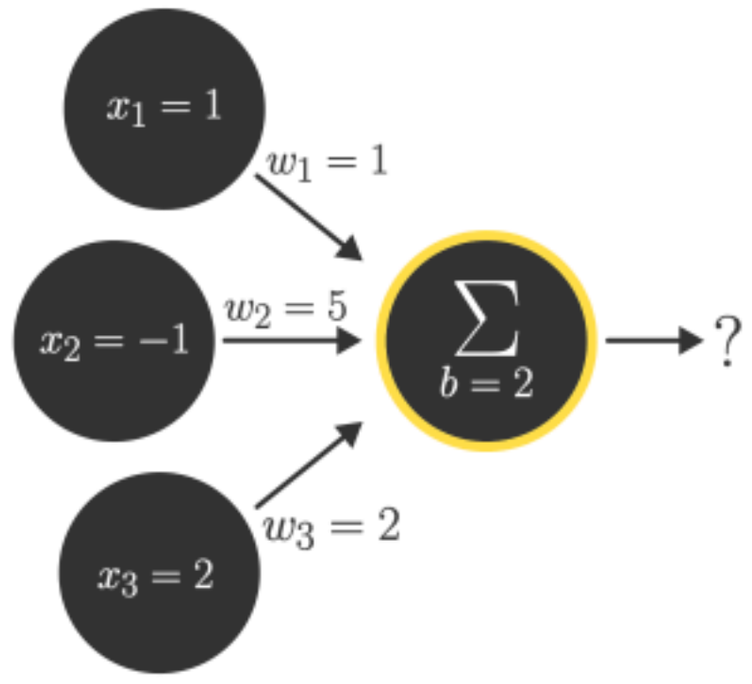
$$\vec{x} = \begin{pmatrix} \sqrt{2} \\ 3 \\ -\pi \end{pmatrix}.$$

In our computational model of a neuron, the inputs defined by the vector $\vec{x}$ are "integrated" by taking the **bias** b plus the dot product of the **inputs** $\vec{x}$ and **weights** $\vec{w}$.

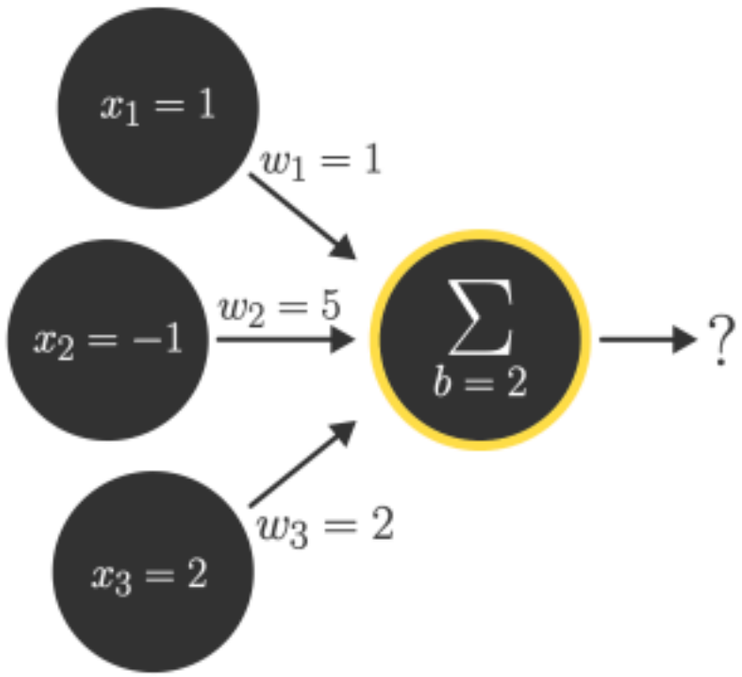
$$\vec{w} \cdot \vec{x} + b$$

The dot product represents a "weighted sum" because it multiplies each input by a weight.

A biological interpretation is that the inputs defining $\vec{x}$ are the outputs of other neurons, the weights defining $\vec{w}$ are the strengths of the connections to those neurons, and the bias b impacts the threshold the computing neuron must surpass in order to fire.



Given the inputs, weights, and bias shown above, what is the integration of these inputs according to the weighted sum $\vec{w} \cdot \vec{x} + b$?



Given the inputs, weights, and bias shown above, what is the integration of these inputs according to the weighted sum $\vec{w} \cdot \vec{x} + b$?

The integration is

$$\vec{w} \cdot \vec{x} + b = \begin{pmatrix} 1 \\ 5 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix} + 2 = 1 \cdot 1 + 5 \cdot (-1) + 2 \cdot 2 + 2 = 2.$$

An activation function, $H(v)$, is used to transform the integration (weighted sum) into a single output which determines whether or not the neuron would fire. For example, we might have $H(v)$ as the Heaviside step function; that is,

$$H(v) = \begin{cases} 1 & \text{if } v \geq 0, \\ 0 & \text{if } v < 0. \end{cases}$$

Considering $H(\vec{w} \cdot \vec{x} + b)$, how does *increasing* the bias b affect the likelihood of the neuron firing (all else equal), assuming that a 1 corresponds to firing?

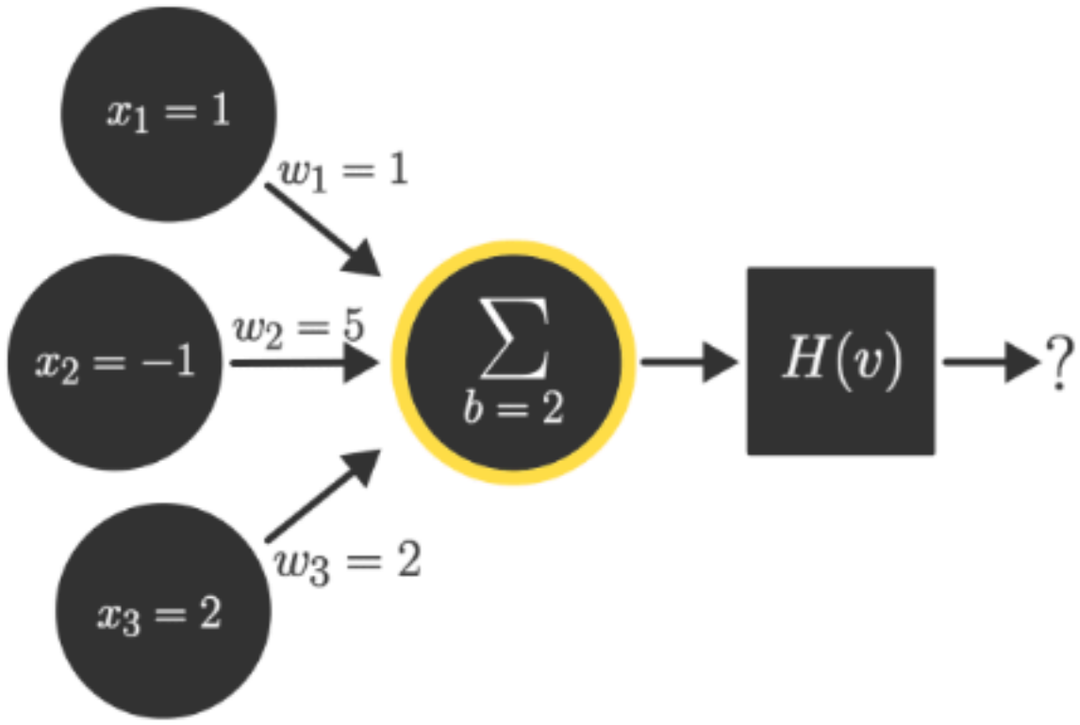
Correct answer: **It increases it**

When the bias is increased, $\vec{w} \cdot \vec{x} + b$ is increased. Thus, increasing the bias makes $\vec{w} \cdot \vec{x} + b$ more likely to pass the firing threshold of 0, as implied by $H(v)$ being the Heaviside step function.

When $H(v)$ is the Heaviside step function, the neuron modeled by $H(\vec{w} \cdot \vec{x} + b)$ fires when $\vec{w} \cdot \vec{x} + b \geq 0$.

The hypersurface $\vec{w} \cdot \vec{x} + b = 0$ is called the **decision boundary**, since it divides the input vector space into two parts based on whether the input would cause the neuron to fire. This model is known as a linear classifier because this boundary is based on a linear combination of the inputs.

NEURON MODELS

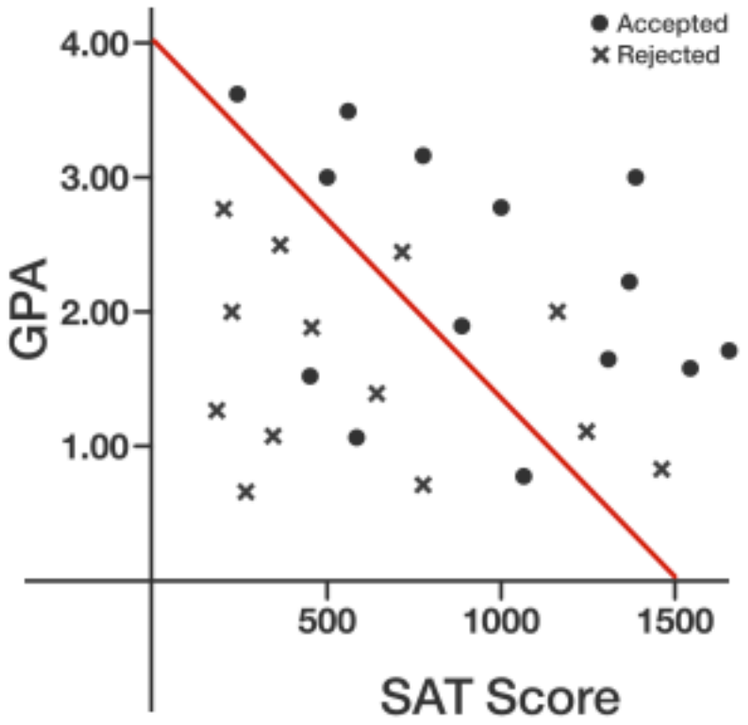


Consider the activation function $H(v) = \dfrac{1}{1+e^{-v}}$, which is known as the sigmoid function. Given the inputs, weights, and bias shown above (which are the same as in an earlier question), what is the approximate output (to the nearest thousandth) from this neuron after the integrated value of the inputs is evaluated by the activation function?

Correct answer: **0.881**

In the earlier question, we found that the integration for this neuron is 2. Thus, the output is $H(2) = \dfrac{1}{1+e^{-2}} \approx 0.88$. We could interpret this as follows: the neuron is "mostly firing" or that it fires with a 0.88 probability.

Note that the choice of activation function is usually fixed for neurons, prior to learning, in ANNs. Since each neuron is parameterized by a weight vector $\vec{w}$ and bias b, the model can learn by changing the values of $\vec{w}$ and b. This can usually be done successfully without changing the activation function itself.



The model above shows a decision boundary for predicting college admission based on the input

$$\vec{x} = \begin{pmatrix} \text{SAT score} \\ \text{GPA} \end{pmatrix}$$

and the activation function $H(\vec{w} \cdot \vec{x} + b)$, where $H(v)$ is the Heaviside step function. Which of the following is a possible value for the weight vector, $\vec{w}$?

- $\begin{pmatrix} 1 \\ 375 \end{pmatrix}$
- $\begin{pmatrix} 375 \\ 1 \end{pmatrix}$
- $\begin{pmatrix} -1 \\ 375 \end{pmatrix}$
- $\begin{pmatrix} -375 \\ 1 \end{pmatrix}$

Explanation

Correct answer: $\begin{pmatrix} 1 \\ 375 \end{pmatrix}$

The slope of the decision boundary is $-\frac{4}{1500} = -\frac{1}{375}$, so in point-slope form our decision boundary will be

$$(\text{GPA}) = -\frac{1}{375} \cdot (\text{SAT score}) + b.$$

Under some algebraic manipulations, this equation has the form

$$1 \cdot (\text{SAT score}) + 375 \cdot (\text{GPA}) - 375b = 0.$$

We can then convert this to vector notation to get,

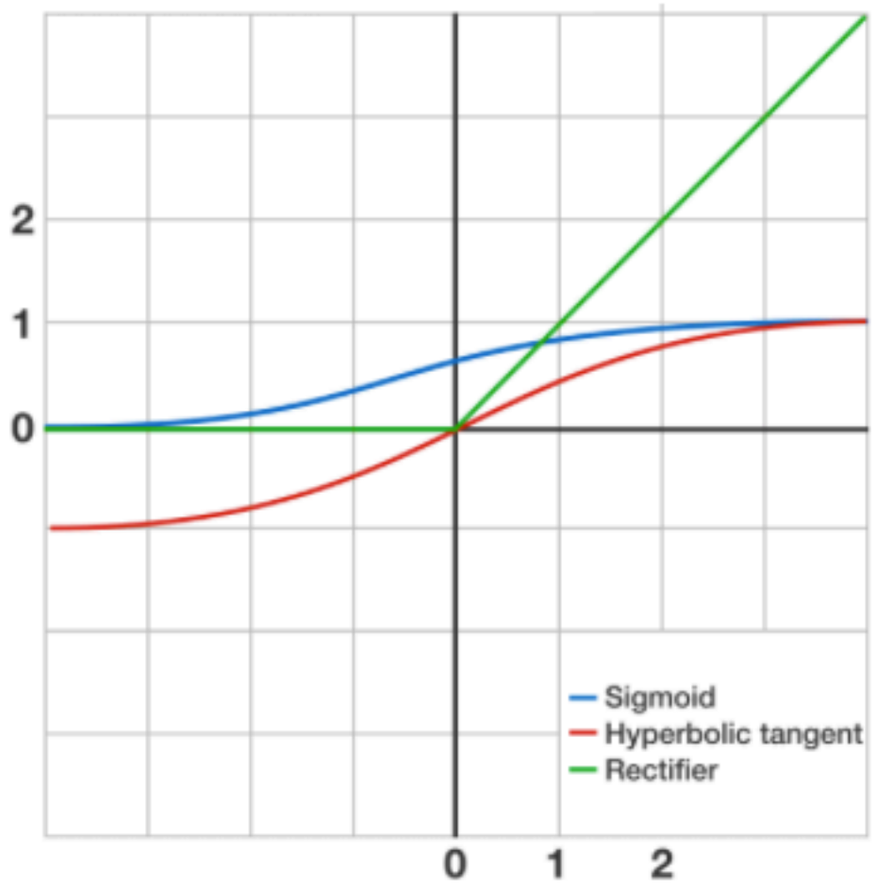
$$\begin{pmatrix} 1 \\ 375 \end{pmatrix} \cdot \vec{x} - 375b = 0$$

Thus, a possible value for the weight vector is

$$\vec{w} = \begin{pmatrix} 1 \\ 375 \end{pmatrix}.$$

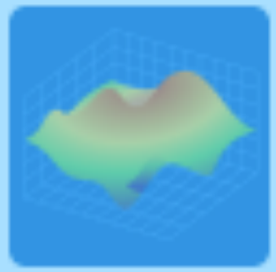
This is the only option for which an equation of the form $\vec{w} \cdot \vec{x} = b$ is capable of producing the line shown. Note that the actual value could be any scalar multiple of this vector.

So far, we've considered an activation function $H(v)$ with binary outputs, as inspired by a physical neuron. However, in ANNs, we don't need to restrict ourselves to a binary function. Functions like the ones below avoid counterintuitive jumps and can model continuous values (e.g., a probability).



The power of ANNs is illustrated by the **universal approximation theorem**: ANNs using activation functions like these can model *any* continuous function, given some general requirements about the size and layout of the ANN.

Math for Neural Networks



3/3

$\vec{w} \cdot \vec{x} + b$

Vectors

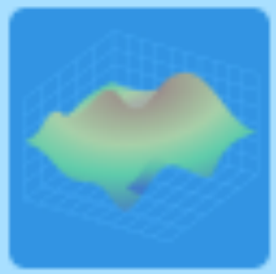
$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

Matrices

$\frac{dH}{dv} = 0$

Optimization

Math for Neural Networks



3/3

The integration of inputs, $\vec{x}$, into our model of the neuron is given by

$$\vec{w} \cdot \vec{x} + b,$$

where $\vec{w}$ is a vector of weights and b is a constant term known as bias. Recall that the dot product is the sum of the element-wise products between two vectors. Which of the following Python implementations correctly computes this integration (assuming valid inputs)?

Python

```python
def integration(x,w,b):
    weighted_sum = sum(x[k] * w[k] for k in xrange(0,len(x)))
    return weighted_sum + b
```

The dot product computes the sum of the element-wise products between two vectors. In other words,

$$\vec{w} \cdot \vec{x} = \sum_{k=1}^{n} w_k x_k.$$

Only B achieves this; A actually finds the sum of the product of *all* pairs of terms between the two vectors.

From a computational perspective, it's often best to keep as much data as possible in vectors, so if possible we should include the bias term in our vectors instead of adding it to the dot product directly.

Suppose that you have changed input vectors $\vec{x}$ to the updated $\overrightarrow{x_u}$ by prepending a new first element of 1. How can you modify the weight vector so that the integration $\vec{w} \cdot \vec{x} + b$ can be transformed into a single dot product that gives equivalent results, $\overrightarrow{w_u} \cdot \overrightarrow{x_u}$?

- ◯ Prepend a new first element of -1
- ◯ Prepend a new first element of b
- ◯ Prepend a new first element of 1
- ◯ This is impossible

Correct answer: **Prepend a new first element of b**

Adding a new first element of b to the weight vector and then taking the dot product gives an extra term in the sum of $1 \cdot b = b$.

For instance, suppose you have

$$\overrightarrow{w_1} \cdot \overrightarrow{x_1} + b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} + b = (1 \cdot 4) + (2 \cdot 5) + (3 \cdot 6) + b$$

We can form a new $\overrightarrow{w_2} \cdot \overrightarrow{x_2} = \overrightarrow{w_1} \cdot \overrightarrow{x_1} + b$ with the vectors:

$$\overrightarrow{w_2} \cdot \overrightarrow{x_2} = \begin{pmatrix} b \\ 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 4 \\ 5 \\ 6 \end{pmatrix} = (1 \cdot b) + (1 \cdot 4) + (2 \cdot 5) + (3 \cdot 6)$$

Which is equal to $\overrightarrow{w_1} \cdot \overrightarrow{x_1} + b$ above.

This technique (of modifying the weight and input vectors to directly account for the bias term) is widely used in ANNs, especially because many languages have built-in vector operations.

In ANNs, the magnitude of various vectors can impact the convergence times when training a model. The magnitude of a vector with elements $x_1, x_2, \ldots, x_n$ is

$$\sqrt{\sum_{k=1}^{n} x_k^2}.$$

Which of the following correctly computes this magnitude given an input vector as a list?

A :

```python
def magnitude(x):
    return sum(k**2 for k in x)**0.5
```

B :

```python
def magnitude(x):
    return sum(x[k]**2 for k in x)**0.5
```

Explanation

Correct answer: A

For every element k in the list, we want to square that element and find the sum of these squares. Then, we take the square root of that result. This is precisely what the code in A does.

The code in B makes the mistake of treating k as an index. This would only be true if the `for` condition were `for k in xrange(0,len(x))`.

To make our algorithms efficient, we'll want to do some *normalization* of vectors. This often requires finding a unit vector (a vector with magnitude 1) in the same direction as a given vector. What vector is the normalization of

$$\begin{pmatrix} 3 \\ 4 \\ 12 \end{pmatrix}?$$

The magnitude of this vector is

$$\sqrt{3^2 + 4^2 + 12^2} = \sqrt{169} = 13.$$

To normalize the vector to a unit vector, we can divide each term by this magnitude to get

$$\begin{pmatrix} \frac{3}{13} \\ \frac{4}{13} \\ \frac{12}{13} \end{pmatrix}.$$

Matrices are a collection of numbers placed into an $m \times n$ grid. More intuitively for ANNs, they are a way to represent a collection of vectors; i.e. n column vectors of length m can be combined into an $m \times n$ matrix (or similarly using row vectors).

The decision to combine many vectors into a matrix in ANNs isn't arbitrary. Mathematically, many steps in our algorithms will be expressed more simply with a matrix. Computationally, there are already implemented ways to perform highly optimized matrix computations.

The multiplication of two matrices gives a matrix such that the element in the i^{th} row and j^{th} column is equal to the dot product of the vectors that form the i^{th} row of the first matrix and j^{th} column of the second matrix.

Suppose A is an $a_1 \times a_2$ matrix and B is a $b_1 \times b_2$ matrix. If the product AB exists, what are its dimensions?

Correct answer: $a_1 \times b_2$

The elements of AB are given by taking the dot product of rows from A and columns from B. Thus, AB will have as many rows as A and as many columns as B. If this is an unfamiliar concept, we'd recommend brushing up on matrix multiplication before moving on.

What is

$$\begin{pmatrix} -2 & 3 \\ 3 & -4 \end{pmatrix}\begin{pmatrix} 4 & 3 \\ 3 & 2 \end{pmatrix}?$$

Explanation

Correct answer: $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

The product is

$$\begin{pmatrix} (-2)\cdot 4 + 3\cdot 3 & (-2)\cdot 3 + 3\cdot 2 \\ 3\cdot 4 + (-4)\cdot 3 & 3\cdot 3 + (-4)\cdot 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

As a side note, this product is known as the identity matrix, usually denoted I. It's the identity element for matrix multiplication. Thus, in this example the two matrices are inverses of each other since their product is the 2×2 identity matrix.

Let A and B be matrices, and I be the identity matrix. I is the identity element for matrix multiplication, and the inverse of a matrix M is a matrix M^{-1} such that $MM^{-1} = M^{-1}M = I$.

If A is an invertible matrix such that $AB = I$, is it necessarily true that $BA = I$?

Hint: Left-multiply the first equation by the inverse of A.

Explanation

Correct answer: **Yes**

Left-multiplying by A^{-1} gives

$$A^{-1}AB = A^{-1}I.$$

By definition, $A^{-1}A = I$, and since I is the identity element for matrix multiplication, $A^{-1}I = A^{-1}$. Thus,

$$B = A^{-1},$$

so $BA = A^{-1}A = I$.

In the previous question, we found a case where $AB = I$ implies that $BA = I$. However, is matrix multiplication commutative? That is, is it **always** true that $AB = BA$?

Explanation

Correct answer: **No**

As a simple counterexample, let A be a 2×3 matrix and B be a 3×2 matrix. Then, AB is a 2×2 matrix but BA is a 3×3 matrix, so $AB \neq BA$.

Even for square matrices, this typically does not hold. Be careful with your orders!

MATRICES FOR NEURAL NETWORKS

The sum S of two matrices A, B *of the same size* satisfies the relation

$$S_{i,j} = A_{i,j} + B_{i,j}$$

for all i, j within the size of the matrices.

The product P of a constant c and matrix A satisfies the relation

$$P_{i,j} = c \cdot A_{i,j}$$

for all i, j within the size of the matrices.

The product P of an $m \times n$ matrix A and an $n \times p$ matrix B satisfies

$$P_{i,j} = A_{i,*} \cdot B_{*,j}$$

for all i, j within the size of the matrices.

Here $A_{i,*}$ denotes the i^{th} row of A, which is a vector, and $B_{*,j}$ denotes the j^{th} column of B, which is also a vector. Thus, the dot $(\cdot)$ in this sense refers to multiplying vectors, defined by the dot product. Note that i and j are defined on $1 \leq i \leq m$ and $1 \leq j \leq p$, so the product P will be a $m \times p$ matrix.

The determinant of an $n \times n$ matrix A is

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+1} a_{1,i} \det(A_{1i}) = a_{1,1} \det A_{11} - a_{1,2} \det A_{12} + \cdots.$$

So, why introduce the complexity of matrices? Graphics processing units (GPUs) are highly optimized to perform parallel matrix operations (since matrices are used to represent many geometrical transforms). Although this requires converting data, models, and algorithms into matrix form, the time taken to do so yields worthwhile practical returns in model training time.

If the idea of parallelization is unfamiliar, consider the example of matrix addition. Summing two $m \times n$ matrices can be parallelized into mn concurrent tasks (single additions), which can then be recombined (summed) to solve the original task. Thus, properly parallelized matrix addition can be completed in $O(1)$ time versus $O(mn)$ time for a serial implementation.

The sigmoid function - often used as an activation function in ANNs - is given by

$$H(v) = \frac{1}{1+e^{-v}}\,.$$

Using the chain rule, compute $\frac{dH}{dv}$.

Correct answer: $\frac{e^{-v}}{(1+e^{-v})^2}$

For the sigmoid function $H(v) = \frac{1}{1+e^{-v}}$ we can define $H(x)$ as $H(x) = \frac{1}{x}$ and $x(v) = 1 + e^{-v}$. Then to implement the chain rule, we'll need to find $H'(x)$ and $x'(v)$.

Via the power rule, the derivative of $\frac{1}{x}$ is $-\frac{1}{x^2}$. Via the chain rule, the derivative of $1 + e^{-v}$ is $-e^{-v}$. Thus, by the chain rule,

$$\frac{dH}{dv} = -\frac{1}{(1+e^{-v})^2} \cdot (-e^{-v}) = \frac{e^{-v}}{(1+e^{-v})^2}\,.$$

Chain Rule

$$\frac{d}{dx} f(g(x)) \equiv f'(g(x)) \cdot g'(x)$$

The alternative statement of the chain in terms of function composition is

$$(f \circ g)' = (f' \circ g) \cdot g'.$$

Another way of stating this which makes it intuitive (almost trivial) would be

$$\frac{df \circ g}{dx} = \frac{df \circ g}{dg}\,\frac{dg}{dx}\,.$$

In our algorithms, we'll want to understand how some function is changing as a function of *all* of its variables. The gradient is, simply, a row vector of a function's partial derivatives.

What is the gradient ∇f of the function $f(a, b, c) = ab^2 + 2c^3$, where the variables, in order, are a, b, and c?

Explanation

Correct answer: $\begin{pmatrix} b^2 & 2ab & 6c^2 \end{pmatrix}$

The gradient is

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial a} & \frac{\partial f}{\partial b} & \frac{\partial f}{\partial c} \end{pmatrix} = \begin{pmatrix} b^2 & 2ab & 6c^2 \end{pmatrix}.$$

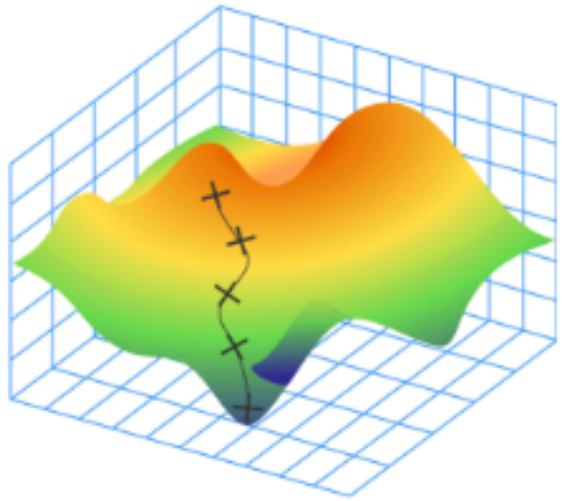
The tricky thing here is to keep track of which "constants" are multiplicative (so stay there as multiplicative constants) and which are additive (so go away as 0).

Training an ANN involves finding weights and biases such that our model performs well. This is an **optimization** problem. In basic calculus, with a simple algebraic function such as a polynomial, a standard optimization process is:

1. Take the derivative of the function
2. Set the derivative equal to 0
3. Solve for the parameters (inputs) that satisfy this equation

Unfortunately, the functions (and their derivatives) involved in ANNs are often very complicated. Accordingly, step 3 in the optimization process above is not possible -- typical analytical approaches for finding where the derivative is 0 will not work. Thus, **heuristic methods are often used**, which we'll explore next.

Gradient descent is a heuristic method that starts at a random point and iteratively moves in the direction (hence "gradient") that decreases (hence "descent") the function that we want to minimize. With enough of these steps in the decreasing direction, a local minimum can theoretically be reached. Colloquially, think of it as playing a game of “hot” and “cold”, until the improvement becomes negligible.



Given a function `gradient` that has computed the gradient for a given function (and the ability to do vector addition), pseudocode for gradient descent would look like this:

Python

```python
def gradient_descent(point, step_size, threshold):
    value = f(point)
    new_point = point - step_size * gradient(point)
    new_value = f(new_point)
    if abs(new_value - value) < threshold:
        return value
    return gradient_descent(new_point, step_size, threshold)
```

Suppose gradient descent is used to try to find the minimum of the function $f(x, y) = 1 + x^2 + y^2$ starting at the point $(x, y) = (1, 1)$.

For each step, the new point will be generated by moving in the negative direction of the gradient using a multiplicative **step size** of 0.5. In other words, it will move in the direction opposite of the vector that is one-half the gradient at $(1, 1)$.

What will the sum of the x and y coordinates be after the first step?

Our first step is to calculate the gradient by calculating the partial derivatives separately.

$$\frac{\partial}{\partial x}\ x^2 + y^2 + 1 = 2x$$
$$\frac{\partial}{\partial y}\ x^2 + y^2 + 1 = 2y$$

This gives us the gradient, $\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{pmatrix} = (2x\ \ 2y)$.

From this, we can see that the gradient at $(1, 1)$ is $(2\ \ 2)$. Earlier, we were told to move in the direction of the negative gradient, scaled by 0.5, so to find our answer we can simply multiply the gradient by -0.5 and add it to our current position.

Since,

$$(1, 1) - 0.5(2, 2) = (1, 1) - (1, 1) = (0, 0),$$

a single step will move us to the point $(0, 0)$. This is actually the global minimum!

In the previous question, the step size (which is also known as the **learning rate**) was chosen such that gradient descent stumbled upon the global minima in just a single step.

However, the term "learning rate" is a bit of a misnomer, because it is possible for the learning rate to be too small or too large. That is, larger learning rates don't always yield faster learning. In fact, large learning rates often yield worse models, since they cause gradient descent to overshoot minima. Similarly, small learning rates learn too slow and may fail to converge in a reasonable amount of time.

Suppose gradient descent is used on the function $f(x) = x^2 - 1$. If the gradient descent begins at $x = -1$ and uses a step size (or learning rate) of 1.0, will it ever converge to the global minimum at $x = 0$?

Correct answer: **No**

Gradient descent moves in the direction of the negative gradient weighted by the learning rate for each iteration. Since the gradient of the function $y = x^2 - 1$ is $\nabla y = (2x)$, the negative gradient is $(-2x)$. Weighted by the learning rate $\alpha = 1$, each iteration of gradient descent moves by

$$\alpha \cdot -\nabla y = -2x$$

if the current iteration of gradient descent is at x.

Denoting $x_0 = -1$ as the starting point for gradient descent, by the equations above, the next step of gradient descent will be at

$$x_1 = x_0 + (-2) \cdot x_0 = (-1) + (-2) \cdot (-1) = 1.$$

Similarly, the next step of gradient descent will be at

$$x_2 = x_1 + (-2) \cdot x_1 = 1 + (-2) \cdot 1 = -1.$$

Gradient descent will oscillate endlessly between the values of $x = -1$ and $x = 1$, never reaching the minimum at $x = 0$.

This example shows that gradient descent doesn't always find a minimum. Thus, it is important to find the right learning rate for the problem at hand. This often takes the form of running a learning algorithm with many different learning rates and choosing the one that offers the best performance on average.

ANNs are complicated models that should have a large number of local minima, making gradient descent ineffective as a training mechanism (since gradient descent will find any local minimum near its starting point). However, highly performant ANNs are often trained using gradient descent, so scientists have long wondered how to explain this seeming contradiction.

Recently, empirical evidence has indicated that linear ANNs have most of their minima near the global minimum, meaning that **gradient descent often performs "well enough"**. There are other, more advanced optimization methods that are employed with ANNs, but this course will focus on using gradient descent due to its simplicity and intuitive interpretation.

Now that we've got the math down, **let's dive back into the models**.

SVM

Support vector machines are a supervised learning method used to perform binary classification on data. They are motivated by the principle of *optimal separation*, the idea that a good classifier finds the largest gap possible between data points of different classes.
For example, an algorithm learning to separate the United States from Europe on a map could correctly learn a boundary 100 miles off the eastern shore of the United States, but a much better boundary would be the one running down the middle of the Atlantic Ocean. Intuitively, this is because the latter boundary maximizes the distance to both the United States and Europe.

The original support vector machines (**SVMs**) were invented by Vladimir Vapnik in 1963. They were designed to address a longstanding problem with logistic regression, another machine learning technique used to classify data.
Linear regression is a probabilistic binary linear classifier, meaning it calculates the probability that a data point belongs to one of two classes. Linear regression attempts to maximize the probability of the classes of known data points according to the model, and so, may place the classification boundary arbitrarily close to a particular data point. This violates the commonsense notion that a good classifier should not place a boundary near a known data point, since data points that are close to each other should be of the same class.

Maximum Margin Classifiers

Edit

Support vector machines, on the other hand, are **non-probabilistic**, so they assign a data point to a class with 100% certainty (though a bad SVM may still assign a data point to the wrong class). This means that two SVMs giving the same class assignment to a set of data points have the same classification accuracy. How then should we determine which of the two is better?
The answer lies in the size of the gap between data points of different classes. If two SVMs give the same class assignment of data points, we would like to choose the model whose closest data point is furthest away from its classification boundary. Ideally, the **classification boundary** will be a curve that goes right down the middle of the gap between classes, because this would be the classficiation boundary with the largest distance to the closest data point.
In the case of two linearly separable classes in the plane, this boundary would be a line that passes through the middle of the two closest data points from different classes. Passing through the midpoint of the line connecting two data points maximizes the distance to each data point. In more than two dimensions, this boundary is known as a **hyperplane**. This reasoning, which says that the best linear classifier is the one that maximizes the distance from the boundary to data points from each class, gives what are known as **maximum margin classifiers**. The classification boundary of a maximum margin classifier is known as a **maximum margin hyperplane**. Support vector machines are one such example of maximum margin classifiers.

Definition

The distance from the SVM's classification boundary to the nearest data point is known as the **margin**. The data points from each class that lie closest to the classification boundary are known as **support vectors**. If an SVM is given a data point closer to the classification boundary than the support vectors, the SVM declares that data point to be too close for accurate classification. This defines a **'no-man's land'** for all points within the margin of the classification boundary. Since the support vectors are the data points closest to this 'no-man's land' without being in it, intuitively they are also the points most likely to be misclassified.

Thus, SVMs can be defined as linear classifiers under the following two assumptions:

1. The margin should be as large as possible.
2. The support vectors are the most useful data points because they are the ones most likely to be incorrectly classified.

The second assumption leads to a desirable property of SVMs. After training, the SVM can throw away all other data points, and just perform classification using the support vectors. This means that once classification is done, an SVM can predict a data point's class very efficiently, since it only needs to use a handful of support vectors, instead of the entire dataset. This means that the primary goal of training SVMs is to find support vectors in the dataset that both separate the data and find the maximum margin between classes.

Hard-Margin SVMs

Edit

Training an SVM is easiest to visualize in two dimensions when the classes are linearly separable. Suppose we are given a dataset where for inputs in class 0 and for inputs in class 1. Recalling the vector equation for a line in two dimensions, the classification boundary is defined as , where and are two dimensional vectors. Furthermore, define the negative support vector to be the input vector from class 0 and the positive support vector to be the input vector from class 1.

Again recalling the vector equation for a line, define the negative classification boundary to be and the positive classification boundary to be . Then, the distance between the negative and positive classification boundaries is . Thus, the size of the margin is .

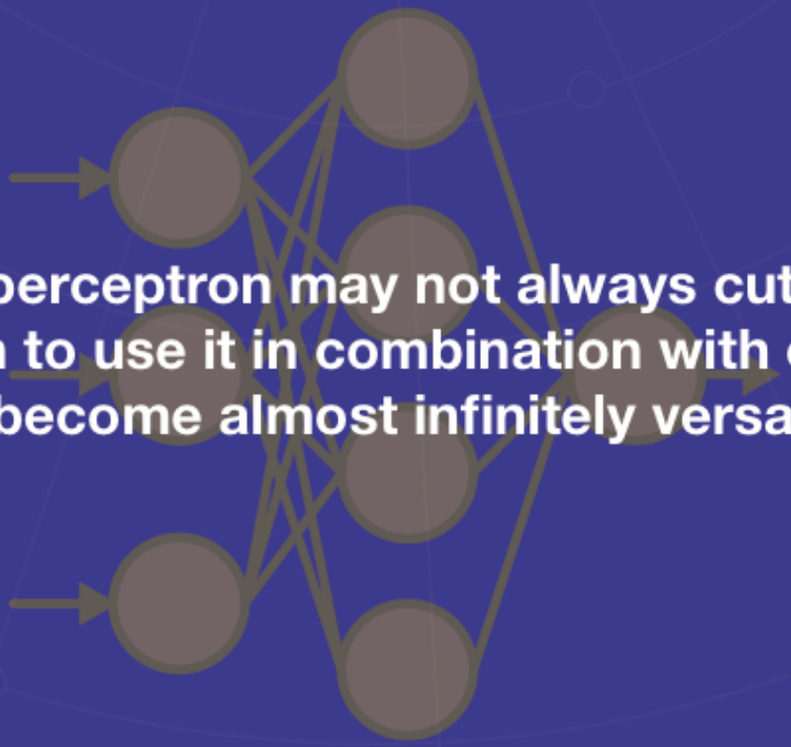
Since we want to maximize , we have to minimize . However, we also want to make sure that no points fall in the 'no-man's land', so we also need to introduce the constraints that for all in class 0 and for all in class 1. This leads to following optimization problem:

Minimize subject to for

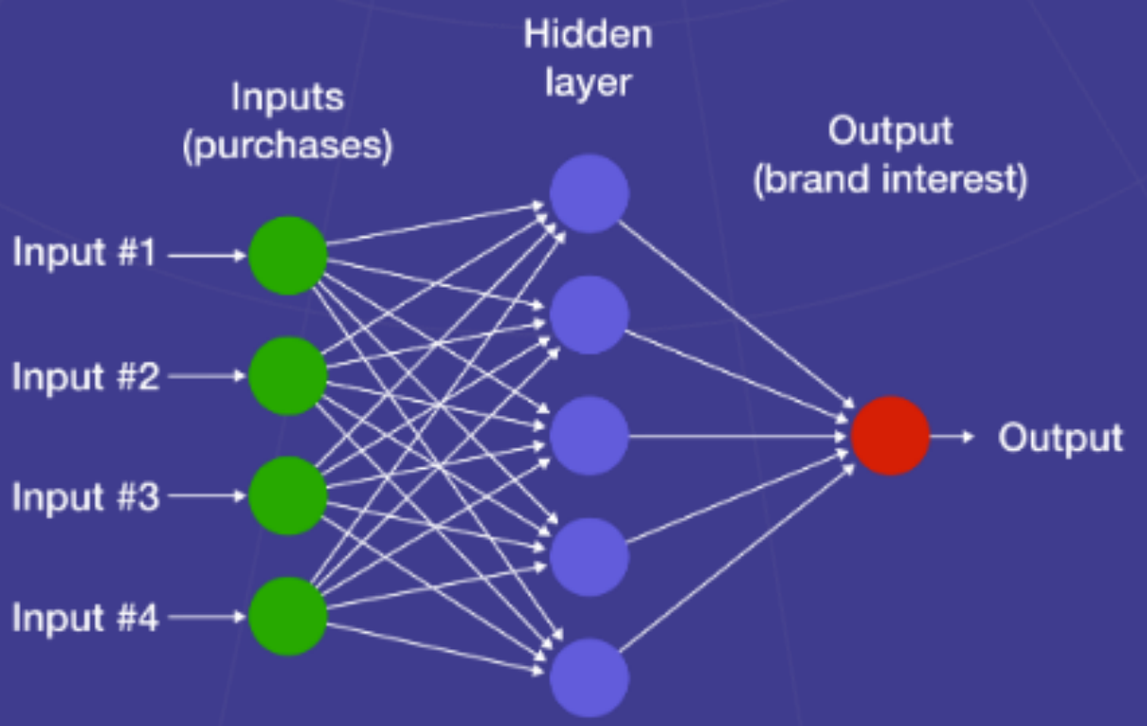
This optimization problem can be solved using linear programming techniques.

Artificial Neural Networks

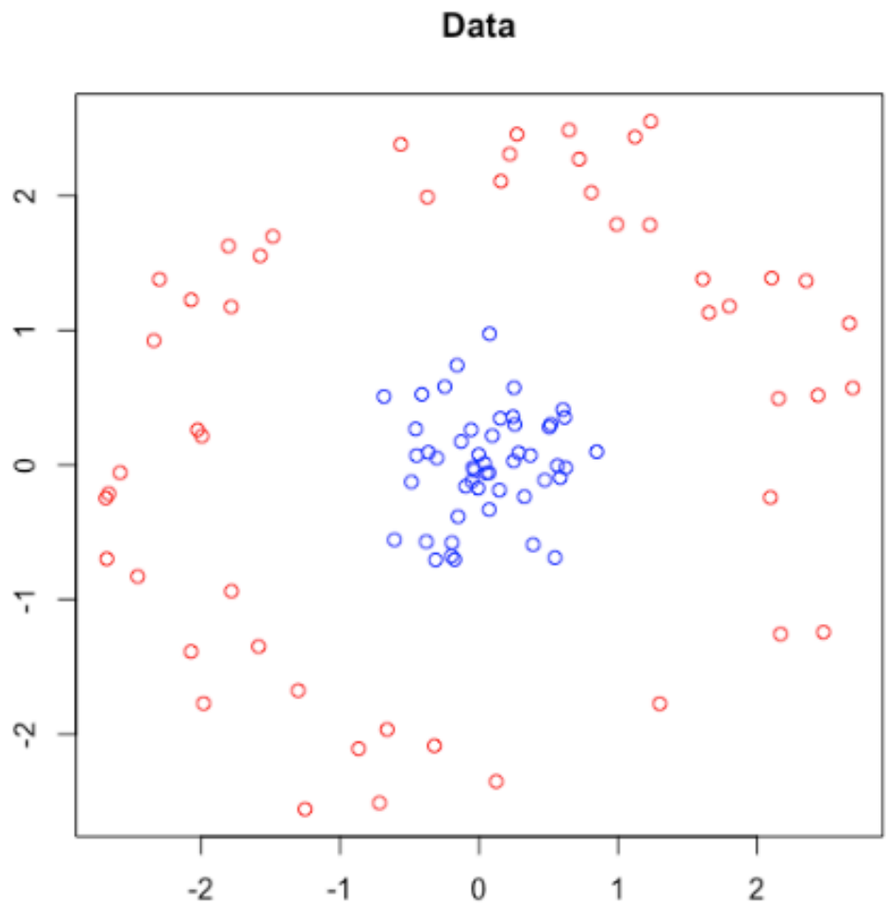
Multilayer Perceptrons



A single perceptron may not always cut it, but if you learn to use it in combination with others it can become almost infinitely versatile.



Will the perceptron algorithm we saw previously successfully classify the following data?



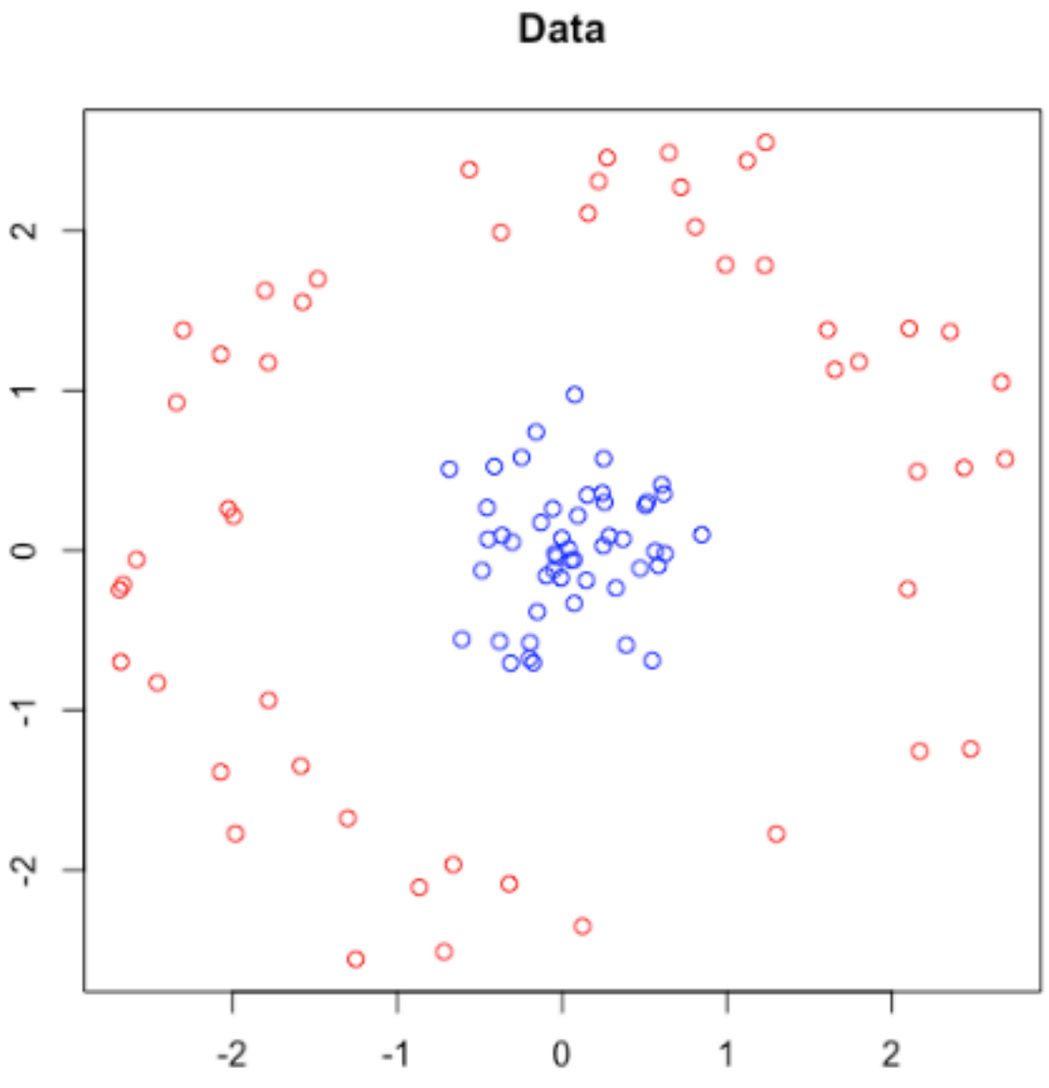
Note: We cannot modify the perceptron algorithm for the purposes of this problem.

Correct answer: **No**

The perceptron algorithm gives us a way to draw a line between two groups of points. In order to be able to converge and successfully separate the groups, this requires that there exists a line that successfully separates them. We say that such a line exists if two groups of points are linearly separable. However, the blue and the red points are not linearly separable because any line that does not have red points on both sides of it has the blue points on the same side of the line as the red points.

In the last problem, we saw it was impossible to linearly separate the points. However, when two groups of points are not linearly separable, there may be a way to transform the coordinates under which we get linearly separable points. These transformations are called **basis functions**, and can be anything. For instance, suppose you have points (x, y), then you could map these to the one-dimensional point $\phi\big((x, y)\big) = y - x$.

If we encounter the following data pattern,



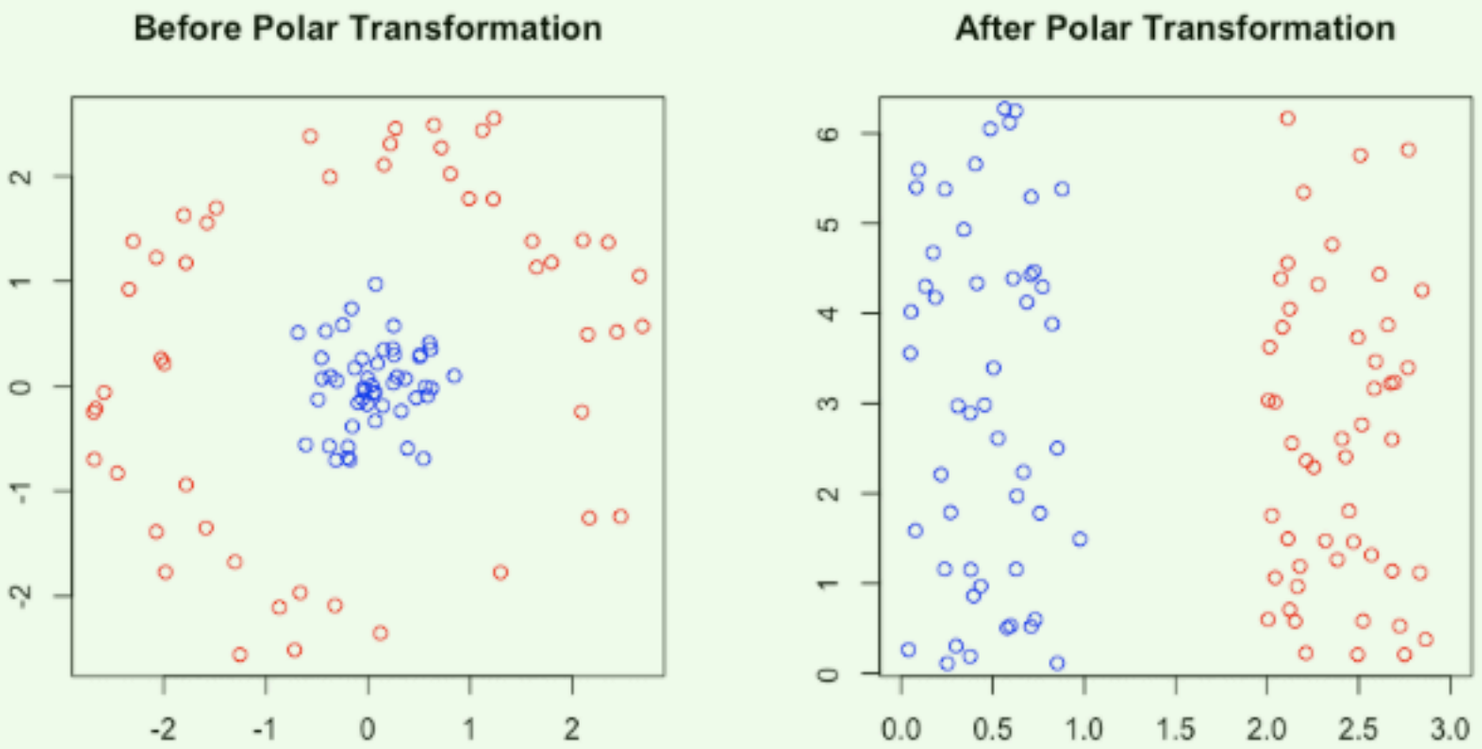
which transformation can we apply to the data so as to make the points linearly separable?

Correct answer: **Polar transformation**

If we solve the following equations for (r, θ):

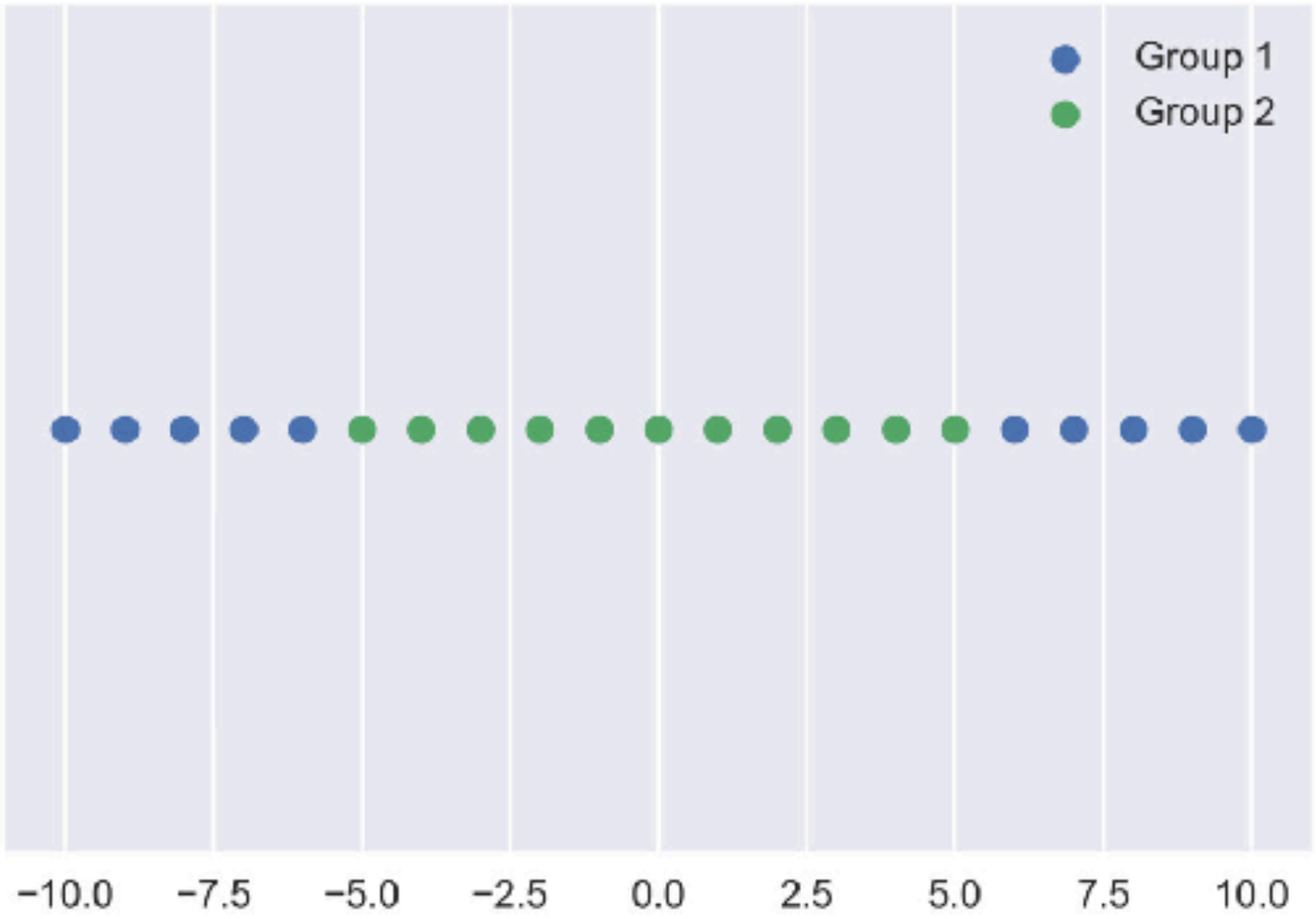
$$x = r\cos\theta$$
$$y = r\sin\theta,\ \theta \in [0, 2\pi),$$

and plot the results on an (r, θ) plane, we obtain the following:



Note that this is now (obviously) linearly separable. Rotation and translation are **linear** transformations themselves and cannot make the data linearly separable. (Quick proof: suppose the data is linearly separable after rotation, then if we rotate back, the line that separates should still be a line, contradiction). Fourier transformation is not well-defined on points in $\mathbf{R}^2$.

Note that this data is not linearly separable. Which of the following is the best transformation to apply so as to make this data linearly separable?



x->x^2
x->x^3
x->e^x

Correct Answer: x -> x^2

In the last two problems, we relied on our mathematical intuition to find basis functions under which the data became linearly separable. In general, though, we'd love to have a way to automatically generate basis functions for any dataset.

The **universal approximation theorem** gives us an inspiration for such functions. Let $f(x)$ be a continuous function where x may be a multidimensional vector, and let σ be the logistic function. Then, the universal approximation theorem tells us that there exists an N for which we can write

$$F(x) = \sum_{i=1}^{N} c_i \sigma(w_i^\top x + b_i)$$

such that $|F(x) - f(x)| < \epsilon$ for all x. In other words, we can approximate an arbitrarily complex function (which may be a decision boundary) arbitrarily well using some number of these logistic functions. This sets the stage for the multi-layer perceptron.

You are in charge of advertising for your company that sells athletic sportswear for adult men. You want to launch an advertising campaign, and your boss gave you a budget as well as a list of consumers and things they've bought in the past. You need to predict which consumers are most likely to be interested in your product, to show them ads.

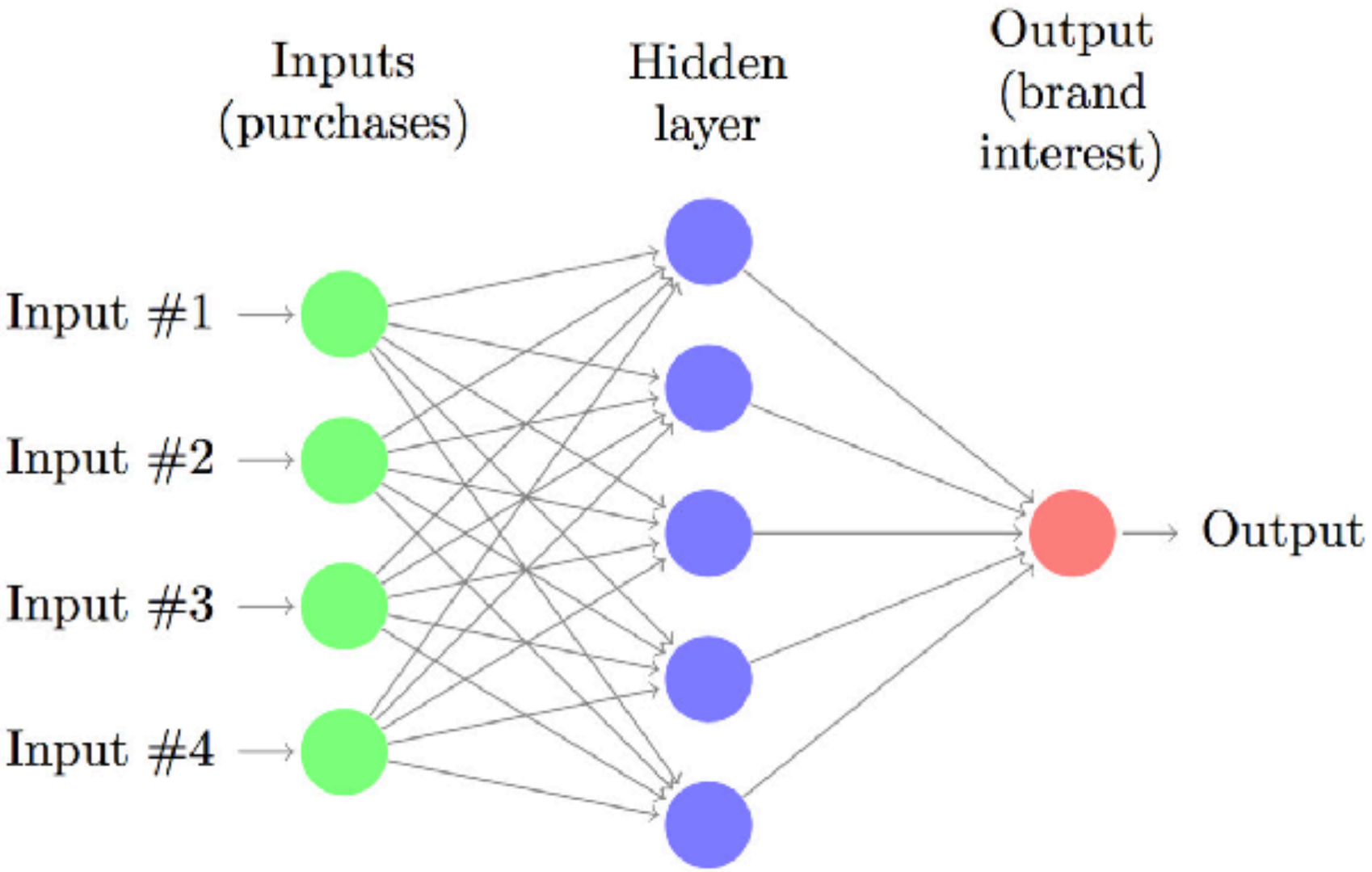
Suppose you see two consumers: the first whose only purchase you can see is men's deodorant, and the second for whom you have no recorded purchases. If you could only send ads to one of them, which one would it be?

Explanation

Correct answer: **The first**

There are a couple different plausible intuitions why the first person would be a better target consumer. First of all, the purchase of any product that does not give us information that a consumer is averse to men's sportswear is an indication of an increased propensity to buy. Therefore, all else equal, you'd prefer to advertise to a consumer who is more likely to buy in general. More specifically to this problem, though, the purchase of men's deodarant suggests that the consumer is a man, which would increase propensity to buy men's clothing.

A multi-layer perceptron works by using the raw features (in this example, purchases) in order to predict intermediate variables. The intermediate variables are then used to predict the final variable of interest (in this example, affinity to this marketer's product).



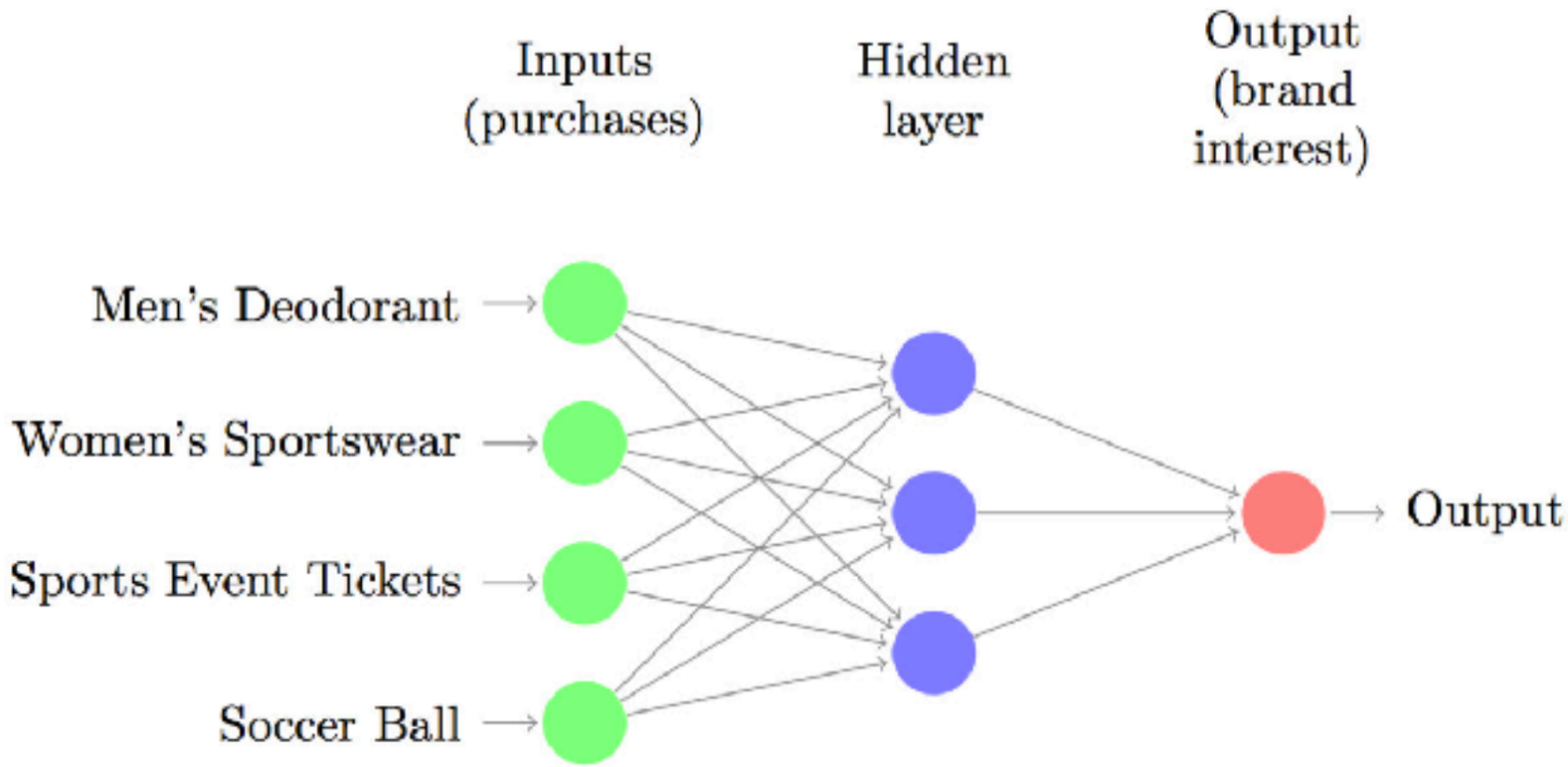
In this context, each of the variable corresponds with a neuron, and the intermediate ones are collectively called the hidden layer (because we never observe them directly). For this example, which of the following would be the least useful hidden-layer neuron to have?

A. Being a man B. Being into art C. Being into sports D. Propensity to consume

Correct answer: **Being into art**

In assessing the usefulness of a given hidden-layer variable, we'd like to think about how predictive it is in determining whether someone would be interested in buying men's sportswear. The "being a man" neuron captures the main intuition we used in determing whether a purchase of men's deodarant was predictive of brand interest. Similarly, a propensity to consume identifies those consumers who are more likely to buy something, and interest in sports should also be predictive of interest in sportswear. Of these, interest in art is the least related to predicting interest in the product because knowing whether someone is interested in art doesn't tell us much about whether they might also be interested in men's sportwear.

Suppose that we now have a multi-layer perceptron that looks like



where the values of our three hidden neurons capture gender, sports interest, and general propensity to consume. Why do we not just use the purchases to directly predict the output?

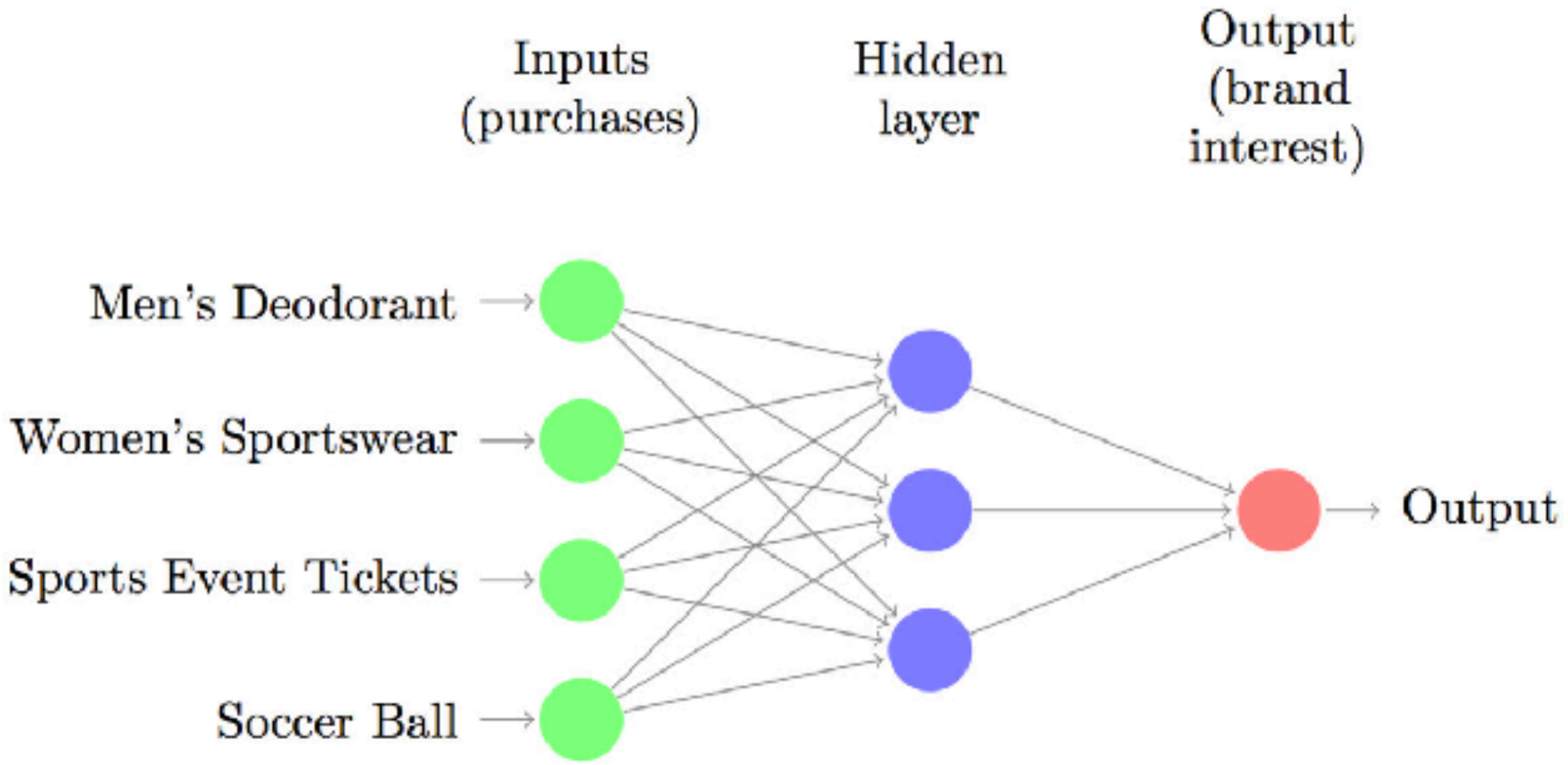
A. The hidden layer can capture non-linearities

B. B. The model is more intuitive

Correct answer: **The hidden layer can capture non-linearities**

Adding the hidden layer makes the model more complex because we're now making four predictions instead of only one. This in turn makes this model more complex computationally than a one-layer perceptron. While, for this particular example, the hidden layer is interpretable in the way that we have defined it, in general, it is actually quite difficult to find interpretations for the hidden layer. In fact, the more hidden layers and neurons we add to any model, the harder it becomes to interpret what the coefficients or the neurons mean. However, adding the hidden layer makes the model better able to classify points even when they aren't linearly separable becaue the hidden neurons act as basis functions.

As before, we have a multi-layer perceptron that looks like

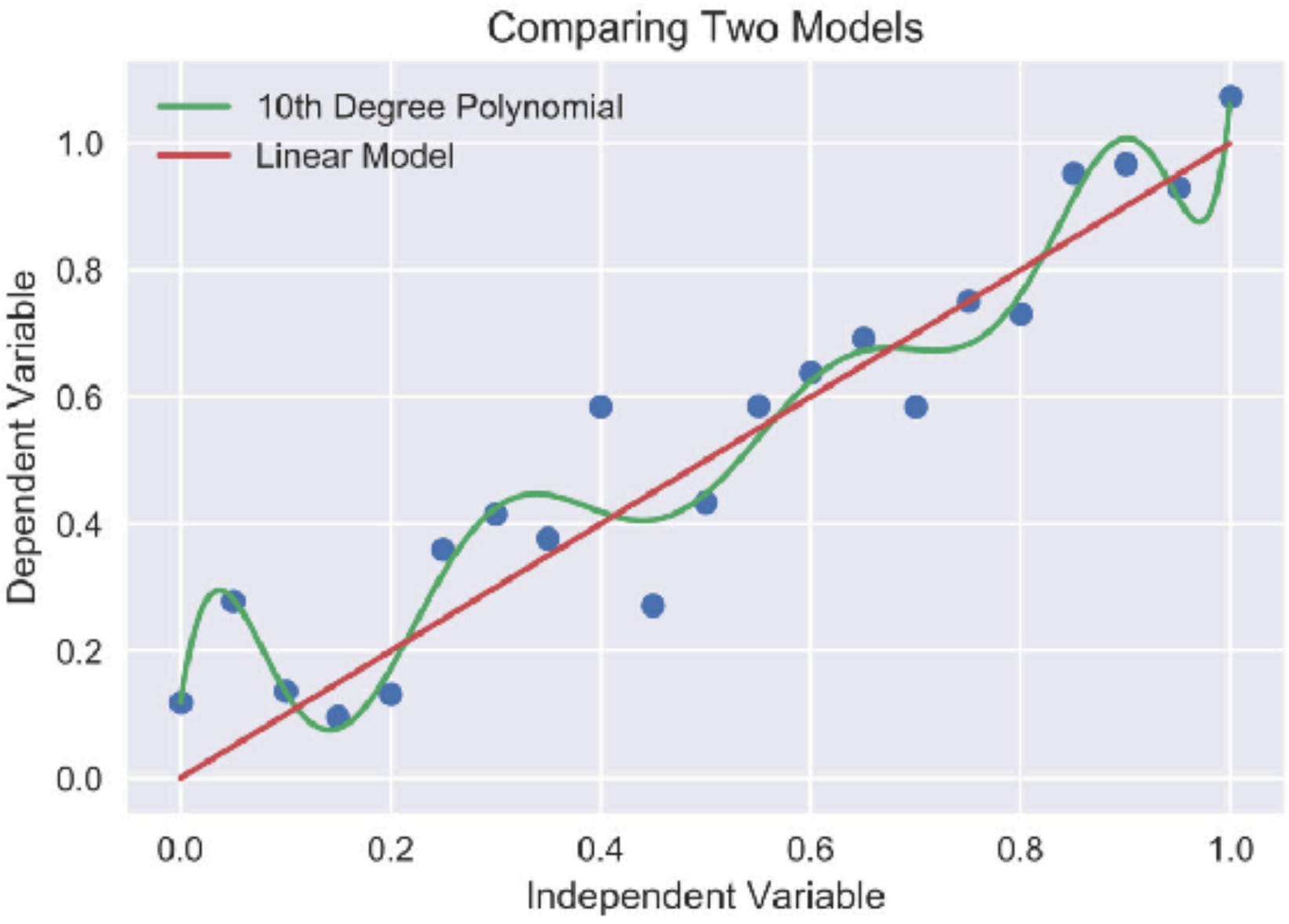


where the values of our three hidden neurons capture being male, sports interest, and general propensity to consume, all of which are positively predictive of affinity with the brand. All the inputs but one have an unambiguous effect on the output (i.e. they either only affect one hidden neuron, or they affect more than one but with the same direction of effect on output). Which is the exception?

Correct answer: **Women's Sportswear**

All purchases presumably affect propensity to consume positively (because we have seen evidence that the consumer is willing to buy something). A purchase of men's deodorant further shows that the consumer is more likely to be male, and thus more prone to purchase men's sportswear. Purchases of the sports events ticket and a soccer ball both demonstrate interest in athletics and thus increase affinity with the brand product. However, women's sportswear demonstrates interest in athletics as well as decreased likelihood of being male. These effects will be at odds with each other when combined in the output.

Model complexity is an important thing to keep in mind. If a model is too simple, it may not capture all the relevant features of the data and be a poor predictor. However, if it's too complex, it may overfit to noise in the data you use to train it and be a poor predictor of new data. A linear model and a 10^{th} degree polynomial are fit to the following data:



Which is less likely to overfit the data?

Correct answer: **The line**

The polynomial is the more complex of the two models. If we look at the main pattern of the data, we see that there is an upward trend with some noise around that trend. The 10^{th} degree polynomial is closely following that noise, creating a zig-zag-esque pattern around the line of best fit. This suggests that the 10^{th} degree polynomial is overfitting to what is actually just a linear relationship. Indeed, the data was generated as random **normal** noise around the line $y = x$, making the linear model a better choice.

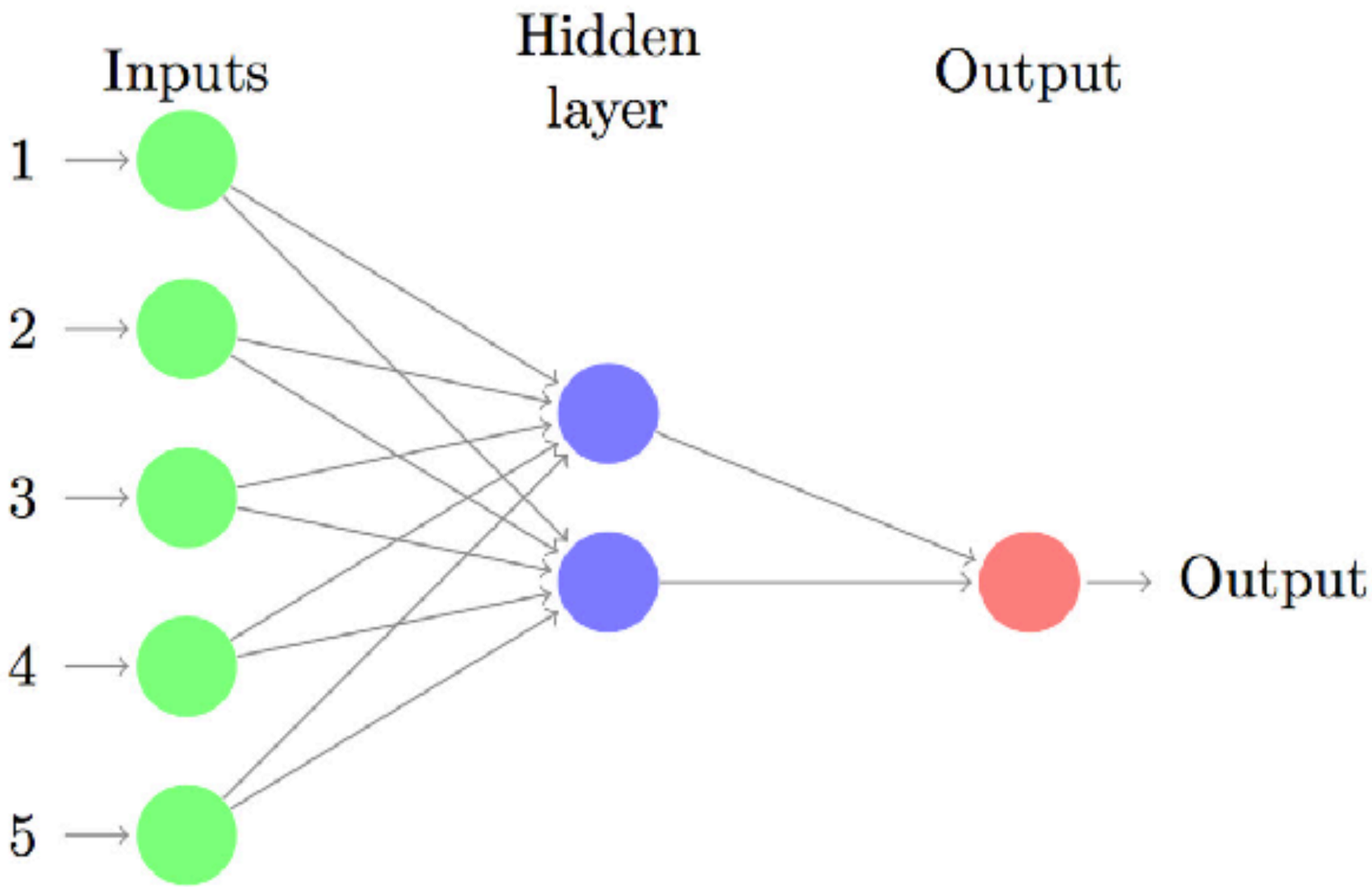
One very important way to measure model complexity is by counting the number of parameters that it has. For example, the linear model from the previous problem has 2 parameters (a slope and a y-intercept) whereas the 10^{th} degree polynomial has 11 parameters $\big($one coefficient for each x^k term for $k \in \{0, 1, \ldots, 10\}\big)$.

How many parameters does a single-layer perceptron with 5 inputs and one output have?

Correct answer: **6**

The single-layer perceptron will have one coefficient for each input, as well as a bias term, amounting to 6 parameters.

We decide that the single-layer perceptron was not complex enough for our purposes. We still have 5 inputs and 1 output, but now we add a hidden layer with two neurons. Our neural network (multi-layer perceptron) looks like



Given that input nodes require no parameters, and each neuron is fed into by all neurons from the previous layer, how many parameters does this model have? Here, the parameters of a neuron correspond with the parameters of a single layer perceptron with the same inputs.

Correct answer: **15**

Each of the hidden neurons has 6 parameters, one for each input as well as a bias term. Then, the output neuron has 3 parameters, one for each hidden neuron that feeds into it and one for bias. Adding these all together gives us

$$2(6) + 3 = 12 + 3 = 15.$$

The model with the hidden layer worked a lot better! You now add another hidden layer with two neurons, so now your neural network has 5 inputs, 2 hidden layers with 2 neurons each, and 1 output. How many parameters do you have now?

Correct answer: **21**

Each of the nodes in the first hidden layer will have 6 parameters, one for each input and one for bias. Each node in the second hidden layer will have 3 parameters, one for each neuron in the first hidden layer, and one for bias. Finally, the output will have three parameters, one for each neuron in the second hidden layer, and one for bias. Summing these up gives

$$2(6) + 3(3) = 12 + 9 = 21.$$

While your model with two hidden layers fit the training data quite well, it performed poorly on data that it had not seen before. What conclusion might you draw from this?

Correct answer: **The model was too complex**

As with our 10^{th} degree polynomial, we can identify the tell-tale signs of a model that is too complex because it fits the training data much better than it fits data that it's never seen before. A model that is too simplistic wouldn't fit the training data well.

The practice of adding many hidden layers to a neural network and using it for predictions is called deep learning. Because of the sizable number of parameters that need to be learned in deep learning, deep learning has been most successful in domains where there are lots of data, so that it doesn't overfit. The task this neural network was being used for probably does not qualify.

Overfitting

Sometimes, neural networks (and even deep neural networks) give us a good model, but we also really like the model not to be *overfitting*.

The problem of overfitting can be solved by adding more sensors for redundancy. To see why this would be useful, consider the following example, in which we only have one test:

Suppose there is a disease which infects one in a ten thousand people. Suppose a test procedure determines whether you have the disease with 99% accuracy--that is, if you have the disease there is a 1% chance of a false negative, and if you don't have the disease there is a 1% chance of a false positive.

You took the test and the test result is positive. Which is closest to the probability that you have the disease?

Hint: Bayes' Rule is essential for this problem.

- 0.98%
- 1.03%
- 10%
- 99.00%

Correct answer: **0.98%**

For any events A and B, Bayes' Rule states that

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|not\ A)P(not\ A)}\ .$$

Here, we will say that A is the event of actually being ill, while B is the event of testing positive for the illness.

From the problem statement, we know that $P(A) = \frac{1}{10000}$ and correspondingly $P(not\ A) = \frac{9999}{10000}$. Additionally, we know that there is 1% chance of a positive test result when you don't have the disease. This translates to the statement, $P(B|not\ A) = 0.01$.

Finally, since there is a 99% chance of a correct result when you have the disease, $P(B|A) = 0.99$.

With all of this, we can calculate the overall probability by plugging everything into the formula.

$$\frac{0.99 \cdot \frac{1}{10000}}{0.99 \cdot \frac{1}{10000} + 0.01 \cdot \frac{9999}{10000}} \approx 0.00980392 \approx 0.98\%.$$

Now suppose you took two independent tests, each with the same properties as the one from our last problem, and both of them turned positive. What is the probability that you have the disease?

Correct answer: **49.50%**

If we take the answer to the last question as our new Bayesian prior, we have

$$\frac{0.99 \cdot 0.0098}{0.99 \cdot 0.0098 + 0.01 \cdot (1 - 0.0098)} \approx 0.495.$$

We can implement this intuition in neural networks by randomly "dropping out" neurons at each training stage. This forces the neurons around the dropped neurons to pick up what the dropped neurons know. And just as two independent tests vastly improve certainty over one test, we now have multiple neurons checking the same result, making the predictions more robust.

Recall Bayes' rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^c)P(A^c)}.$$

The term $P(A)$ is called the Bayesian prior, and the process of conditioning on the evidence B is called Bayesian updating. We now see how different initial priors influence the results of updating.

Suppose you have a coin. Your Bayesian prior is p that the coin is fair, and $1-p$ that the coin is heads on both sides. You flip the coin, and it lands heads. Let $q(p)$ be the posterior probability that the coin is fair after Bayesian updating, given that the prior is p, then what is $q(0.5) - q(0.1)$?

By Bayes' rule,

$$q(p) = \frac{\frac{1}{2} \cdot p}{\frac{1}{2} \cdot p + 1 \cdot (1-p)} = \frac{p}{2-p}.$$

So

$$q(0.5) - q(0.1) \approx 0.28.$$

Frequentist statistics takes a different approach to inference. Unlike Bayesians, frequentists do not rely on updating and priors. Instead, frequentists assume a null hypothesis and check how implausible the evidence is under the null hypothesis. Frequentists reject the null hypothesis if

$$p\text{-value} := P(\text{evidence}|\text{null hypothesis}) < \alpha \in (0, 1),$$

where α is known as the significance level, for which 0.05 is a popular choice. You may have heard of expressions like "the statistic is significant at the 5% level."

However, if we are testing 20 hypotheses, using $\alpha = 0.05$ for each would be silly. If the hypotheses are independent, then we would expect to reject one hypothesis even if all of them are true, just out of random chance. This is where Bonferroni correction comes in. The correction says we should decrease the significance level when we have more hypotheses. In particular, Bonferroni says that we should adjust by dividing the number of hypotheses. The following exercise illustrates why.

Suppose John holds 20 lottery tickets, each of which has the same probability p of winning. However, the lottery tickets are not necessarily independent.

Additionally, we are told that John has at most a 5% chance of winning anything at all. In other words, John has at least a 95% chance of losing all on all 20 tickets. Intuitively, a bound on the overall winning rate should provide a bound on p.

What is the maximum p such that this upper bound on John's chances of winning will hold regardless of the interdependencies between tickets.

Hint: $P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B) \leq P(A) + P(B)$.

Correct answer: **0.25%**

Following the hint, note that the "worst case scenario" is that winning each ticket is a disjoint event from winning other tickets. In this case, the overall winning rate is maximized to be $20p$. So we have

$$20p \leq 0.05 \implies p \leq 0.250\%.$$

Note that if we assume all lottery tickets are independent, the bound on p is

$$1 - (1-p)^{20} < 0.05 \implies p \leq 0.256\%,$$

which is a looser bound than 0.250. This bound would not work if the lotteries are indeed disjoint.

The intuition of the last exercise means that if you have 20 hypotheses and you're aiming for an overall significance level of 0.05, you should test each hypothesis with a significance level of $\frac{0.05}{20} = 0.0025$ to avoid excessive false positives. This is the intuition behind the Bonferroni correction.

Regularization is another way to avoid overfitting. Regularization means that you penalize large parameters in order to bring them closer to 0. This minimizes the number of parameters that are spuriously large, and corresponds to a Bayesian prior that relationships do not exist.

Suppose we are minimizing a loss function L with respect to b_1, b_2, but we are worried that b_2 may be spuriously large in magnitude, so we want to penalize large b_2. What is a reasonable function form for a regularized loss function that accomplishes this?

(The positive number λ is known as the regularization parameter. It controls how much we trade off between minimizing loss and penalizing large parameters.)

- ◯ $L + \lambda|b_2|^2, \lambda > 0$
- ◯ $L - \lambda|b_2|^2, \lambda > 0$
- ◯ $L + \lambda b_2, \lambda > 0$
- ◯ $L + \lambda b_1, \lambda > 0$

Correct answer: $L + \lambda|b_2|^2, \lambda > 0$

We want to penalize large b_2, so we need to add the size of b_2 to the loss function. We also want to penalize large magnitudes of b_2, not just b_2 that are positively large (and somehow reward b_2 that are negative). This rules out other answers.

Artificial Neural Networks

Backpropagation

Go from basic knowledge of the chain rule to the algorithm which makes gradient descent possible.

WHAT YOU'LL ACCOMPLISH

$$\frac{d}{dx} f(g(x)) = g'(x)\, f'(g(x))$$

Apply the chain rule to derive the equations of backpropagation.

Gradient Decent

In machine learning, we often need to perform optimization of the following form

$$\min_{\mathbf{w}} Q(\mathbf{w}),$$

where $\mathbf{w}$ is a vector of parameters we are optimizing over. Minimizing loss, for example, is one scenario where problems like this show up.

Recall the gradient descent method is as follows:
We start from some initial guess $\mathbf{w}_0$, and we update according to

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \nu \nabla Q(\mathbf{w}_j),$$

where ν is known as the learning parameter.

The graphical intuition of this is very clear. Suppose you are an ant crawling in the inside of a bowl, and you want to find the lowest point. One straightforward, "greedy" algorithm is simply that every step, you move in the direction of the steepest descent. ∇Q gives you the direction of the steepest ascent, and $-\nabla Q$ points to the direction of the steepest descent, and you move with a step size controlled by ν. Once you get to the bottom of the bowl, $\nabla Q = 0$, and the algorithm terminates.

Choosing ν, however, is tricky business. The next questions illustrate that point.

Gradient Decent

Let $Q(w_1, w_2) = \frac{1}{2}\left(w_1^2 + w_2^2\right)$. Suppose $\mathbf{w}_0 = (1, 0)$ and $\nu = 2$. What is $\mathbf{w}_2$?

Correct answer: $(1, 0)$

Note that $\nabla Q(1, 0) = (1, 0)$. So $\mathbf{w}_1 = \mathbf{w}_0 - 2 \cdot (1, 0) = (-1, 0)$. Now $\nabla Q(-1, 0) = (-1, 0)$. So $\mathbf{w}_2 = \mathbf{w}_1 - 2 \cdot (-1, 0) = (1, 0)$.

Recall that in the past problem $\mathbf{w}_0 = \mathbf{w}_2 = \mathbf{w}_4 = \cdots$. So the algorithm never terminates, even though clearly Q attains a minimum at $\mathbf{w} = (0, 0)$. The problem is that we chose a ν too large, so the algorithm "swings around" and never finds the optimum.

Gradient Decent

Now, suppose $Q(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} Q_i(\mathbf{w})$. If N is extremely large, computing $\nabla Q = \frac{1}{N}\sum_{i=1}^{N} \nabla Q_i$ and evaluating all N functions at $\mathbf{w}$ may be very time consuming.

One way to get around this problem is by performing a stochastic gradient descent. Instead of updating using ∇Q every time, we randomly shuffle $i = 1, \ldots, N$ and update using ∇Q_i instead. This way, instead of computing N functions every time, we only compute 1 and randomize over all N.

Stochastic gradient descent usually requires at least as many steps as deterministic gradient descent, converges at similar probabilities, and is similarly difficult to implement computationally. Its great advantage is that it is much less computationally costly.

In fact, we've already seen an example of the **stochastic gradient descent** algorithm in practice! When we update our weight vector in the single-layer perceptron $w_{\text{new}} := w + y \cdot x$ for misclassified examples, we are using the fact that the perceptron loss function has a slope of 0 for correctly classified examples, and a slope of 1 for incorrectly classified examples. Next, we'll see how we can build on this idea to train the more complex multi-layer perceptrons.

Back Propagation - Updating Parameters

We will now revisit the example where we are predicting consumers' affinity with men's sportswear based on what they have and have not purchased in the past. For now, consider only a single-layer perceptron. Suppose that we observe a consumer who has bought a certain magazine, and that consumer buys men's sportswear from our brand. If the weight on having purchased that magazine was 0 before seeing that consumer, what will the weight be when the perceptron has learned from that example after making an incorrect prediction?

- ○ Still 0
- ○ Negative
- ○ Positive
- ○ Ambiguous

Correct answer: **Positive**

The example we see suggests that buying the magazine is positively associated with purchasing men's sportswear. A weight of 0 means that the one-layer perceptron was unaware of any relationship between buying that magazine and buying men's sportswear. Thus, the update will increase the weight to a positive amount to reflect the new information we have.

Back Propagation - Updating Parameters

The previous question illustrates how we can update our weights in a single-layer perceptron. However, the added difficulty of updating weights for a multi-layer perceptron is that we *don't observe* the values of the hidden neurons.

Nevertheless, we will be able to make sensible updates that properly train our parameters by the idea of backpropagation. We'll see the intuition of this technique in the next couple of problems, and we'll approach it mathematically in the next quiz.

Suppose we are still predicting affinity to men's sportswear on the basis of purchases. We have one hidden layer with two neurons which correspond to the likelihood of being male and to the likelihood of being interested in sports, both of which we presuppose are positively associated with the target output of interest in men's sportswear.

We see a consumer who purchased item X. Then we find out that the consumer purchased men's sportswear. After we find out of this purchase, how confident are we that the consumer is a male?

- More confident than before
- Less confident than before
- Equally confident as before

Back Propagation - Updating Parameters

Correct answer: **More confident than before**

This question captures the intuition to the name of "backpropagation." That is, because the information is moving backward through the neural network. More concretely, when we find out that the consumer actually made a men's sportswear purchase, because we believe that men are more likely to buy men's sportswear, we increase our confidence that this consumer is male.

As we'll see in the next question, we can use this change in confidence to update our beliefs about the predictive power of a purchase.

Back Propagation - Updating Parameters

This question again regards our seeing a consumer who purchased item X, and purchased men's sportswear. After seeing this data point, what should our neural network do to the weight of "purchasing X" in predicting whether a consumer is male if the weight is presently 0?

Correct answer: **Increase it**

This question combines the first question we did on the update of the one-layer perceptron with the backpropagation intuition we say in the previous question. Since when we see that the consumer purchased men's sportswear we conclude that the consumer is more likely to be male, since the consume purchased X, we now have reason to believe that purchasing X is a positive signal in predicting whether a consumer is a male.

For this example, it's possible to intuitively grasp what direction the updates should be in because we have given interpretations to the hidden-layer neurons. Note, however, that when the hidden-layer neurons don't have interpretable meaning (as is the case most of the time), we will need to rely on the mathematical procedure of backpropagation to properly tell us how to adjust the weights when we see new data.

Back Propagation

To effectively use gradient descent or stochastic gradient descent, we need an efficient way of computing the gradient of an error function. Backpropagation provides such a way.

Define for each neuron j in layer l the output $o_j^{(l)}$, such that

$$o_j^{(l)} = \phi(a_j^{(l)}) = \phi\left(\sum_{k=1}^{n} w_{kj} o_k^{(l-1)}\right),$$

where $o_k^{(l-1)}$ are neuron outputs from the previous layer, w_{kj} is the weight on synapse from k to j, and ϕ is a sigmoid. Here, bias is omitted.

Suppose we want to find the partial derivative of the error function with respect to w_{ij}.

Back Propagation

To find how any error function E changes with respect to a weight w_{ij}, we apply the chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}} .$$

By referencing the definition on the first slide, we can see that $a_j^{(l)} = \sum_{k=1}^{n} w_{kj} o_k^{(l-1)}$. This implies that $\frac{\partial a_j^{(l)}}{\partial w_{ij}} = o_i^{(l-1)}$. Additionally, we define $\delta_j^{(l)} = \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial a_j^{(l)}}$. Then

$$\frac{\partial E}{\partial w_{ij}} = \delta_j^{(l)} o_i^{(l-1)}.$$

It turns out that there is a recursive identity for $\delta_j^{(l)}$, called the backpropagation formula

$$\delta_j^{(l)} = \phi'(a_j^{(l)}) \sum_m w_{jm} \delta_m^{(l+1)},$$

where the summation is over the neurons that neuron j from the current layer sends signals to, and ϕ' is the derivative of ϕ.

Back Propagation

We can derive this identity by using the chain rule to write $\frac{\partial E}{\partial o_j^{(l)}}$ in terms of $\frac{\partial E}{\partial a_m^{(l+1)}}$ and $\frac{\partial a_m^{(l+1)}}{\partial o_j^{(l)}}$.

$$\begin{aligned}
\delta_j^{(l)} &= \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial a_j^{(l)}} \\
&= \frac{\partial o_j^{(l)}}{\partial a_j^{(l)}} \left(\frac{\partial E}{\partial o_j^{(l)}} \right) \\
&= \phi'(a_j^{(l)}) \sum_m \frac{\partial E}{\partial o_m^{(l+1)}} \frac{\partial o_m^{(l+1)}}{\partial a_m^{(l+1)}} \frac{\partial a_m^{(l+1)}}{\partial o_j^{(l)}} \\
&= \phi'(a_j^{(l)}) \sum_m \delta_m^{(l+1)} w_{jm}
\end{aligned}$$

Note that this allows us to compute previous layers of δ_j by later layers recursively--this is where backpropagation comes from. We can compute δ_j directly, if j is an output layer, so this process eventually terminates.

If we double the number of layers in a neural network, how much more time (approximately) do we expect the backpropagation algorithm to take?

Correct answer: **Twice**

It should take twice as long, as now you have twice as many derivatives in the gradient. Each derivative is not taking longer since it only depends on the number of neurons of the previous layer, which is not increasing.

If we double the number of neurons in each layer of a neural network, how much more time (approximately) do we expect the backpropagation algorithm to take?

Correct answer: **Four times**

Note that for each layer, we now have twice the number of neurons to compute partial derivatives for. Computing each partial derivative depends on the number of neurons of the next layer, which is also doubled. So computing each partial derivative takes twice as long and there are twice as many to compute. Overall it should take four times as long.

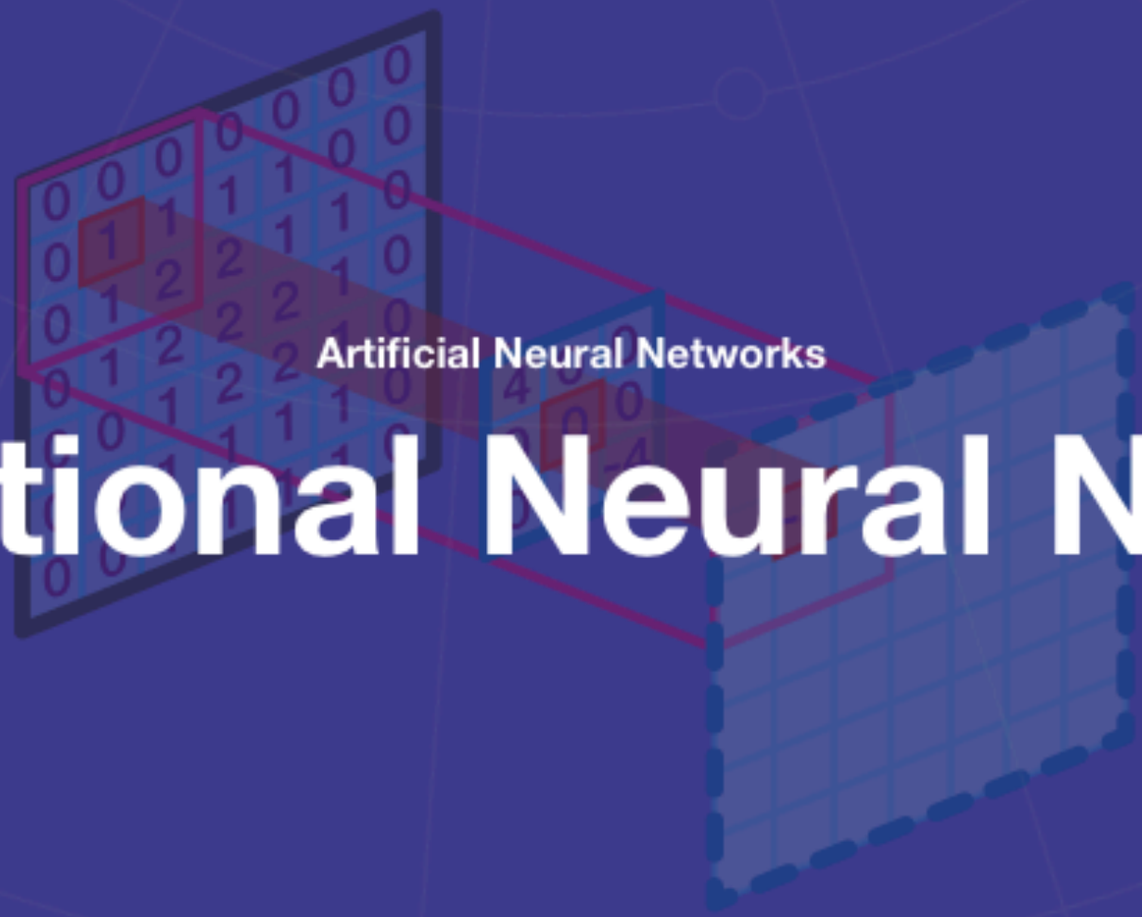
Back Propagation

A good way to think about backpropagation is that its algorithmic complexity is $O(mn)$, where m is the number of neurons in this layer and n is the number of neurons in the next layer. One could also show that this is equivalent to $O(W)$, where W is the number of synapses in the network.

We have seen the basic intuition of backpropagation--computing the gradient of prediction errors by recursively computing terms like δ_j starting from the output layer and into the input layer. We have also seen that the time complexity of this algorithm is linear in the number of synapses of the network.

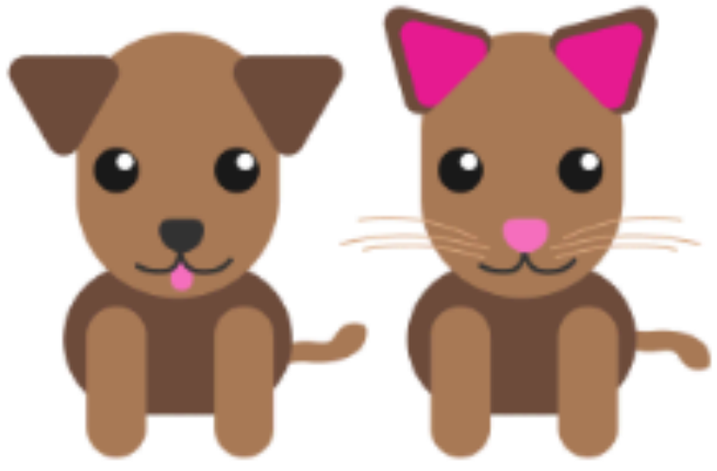
Artificial Neural Networks

Convolutional Neural Networks



If you want to analyze images in particular, convolutional neural networks are an essential part of your education.

Let's return to the problem with which we opened this course. Given images of dogs and cats, how can we train a computer to differentiate between the two?



It is possible to feed the pixels of the images directly into a multilayer perception neural net, but this is an unintuitive approach—it's not at all how our brain processes images. Can we take inspiration from biology to improve our approach?

In this chapter, we will explore **convolutional neural nets (CNNs)**, which have been used to great success in image classification problems, even achieving super-human performance!

In 1959, neuroscientists David Hubel and Torsten Wiesel performed an experiment to gain insight into the brain's visual processing mechanisms. They attached a microelectrode to an anesthetized cat's visual cortex, and projected images in front of the cat to determine what images would cause these neurons to activate. Rather than finding "square" or "circle" neurons which activated when those shapes were projected, they instead observed neurons which were only activated when lines at specific angles or certain patterns of light and dark were displayed.

For example, suppose that there are four neurons, which activate for the following features:

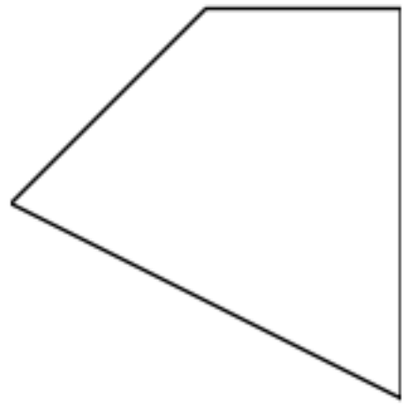
1)

2)

3)

4)

Which neuron(s) will activate if they see this picture?



Correct answer: **1, 2, and 3**

By inspection, we can see that 1, 2, and 3 all match the orientation of at least one of the lines in the image.

CONVOLUTIONAL NEURAL NETWORKS - OVERVIEW

Building on this, suppose that there are four neurons, which activate for the following features:

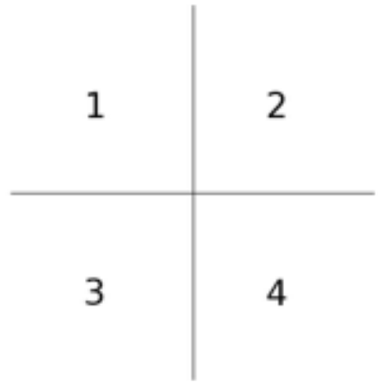
1)

2)

3)

4)

Suppose that we apply these filters to the four quadrants of an image:

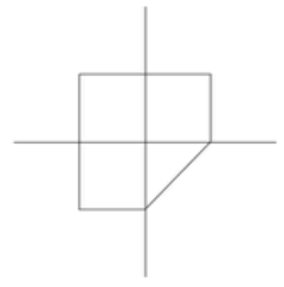


Suppose that the following neurons are observed to activate in each of the following quadrants:

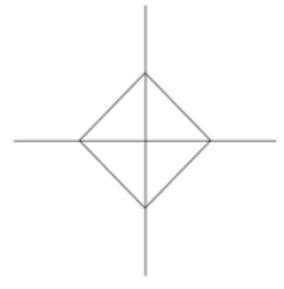
Quadrant	Filters Activated
1	1, 2
2	1, 2
3	1, 2
4	3

Which of the below images could this have been applied to, given these activations?

1)



2)



Correct answer: **1)**

By inspection, for image 1), we can see that, in quadrant 1, we have activations for filters 1 and 2. This is also the case for quadrants 2 and 3, while only filter 3 is activated in quadrant 4. This is the only image for which this is the case, so our answer is image 1).

This is the central idea behind convolutional nets: we train filters to recognize low-level features in an image. We then take the activations of these filters across the image, which gives us a **feature map** for each filter. This is a single convolutional step. Performing more convolutional steps allows us to recognize more complex shapes within images, until we are able to correctly identify objects from thousands of categories in an image.

In the next section, we will discuss more formally how this paradigm is applied in neural nets.

In this quiz, we discuss **convolutions**, **padding**, and **striding**, which mathematically define feature maps when a filter is applied to an image.

The convolution operation for CNNs is quite simple. Suppose we have $N \times N$ matrices A and B.

Let $M_{i,j}$ denote the entry in the i^{th} row and j^{th} column of matrix M.

Then, we define $A \circ B$ as follows, where $\circ$ is the convolutional operator (formally, the Hadamard product): $(A \circ B)_{i,j} = A_{i,j} B_{i,j}$.

For example, let $A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}$.

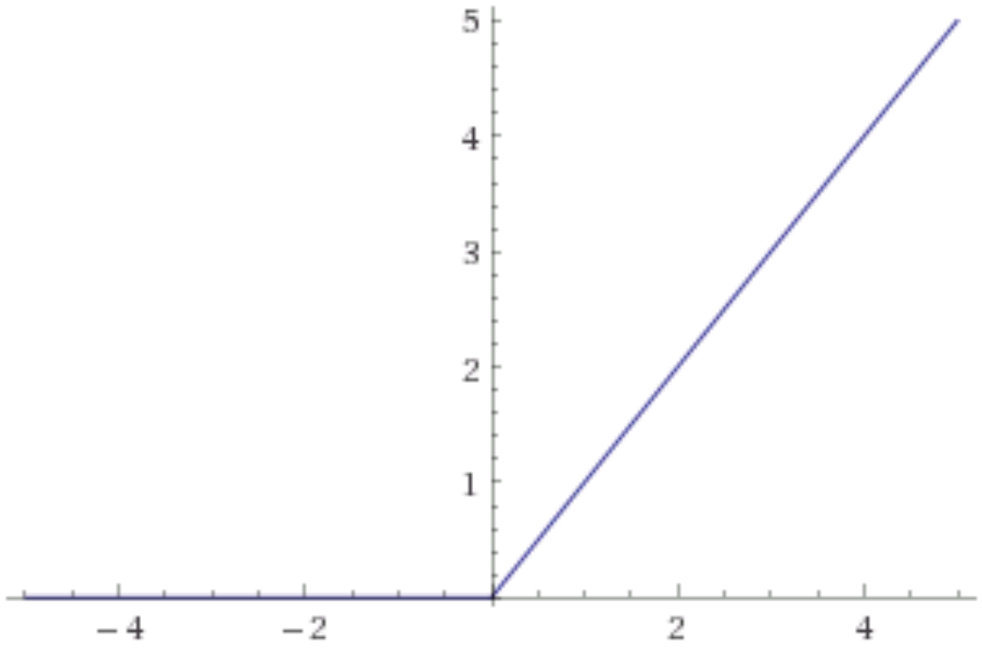
What is the sum of the entries in $A \circ B$?

$A \circ B = \begin{bmatrix} 0 & -1 \\ 2 & 0 \end{bmatrix}$, so the sum of the entries is 1.

Convolutions

The activation of a filter on an image of the same size is found by applying the convolutional operator to the filter and the image, then taking the sum of the entries in the resulting matrix, adding the filter's bias, and applying an activation function to it.

In practice, it is standard to use the ReLU activation function: $f(x) = \max(0, x)$.



What is the activation of the filter $\begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}$ upon the image $\begin{bmatrix} 2 & 3 \\ 3 & 1 \end{bmatrix}$ if the filter's bias is 1 and we use ReLU as our activation function?

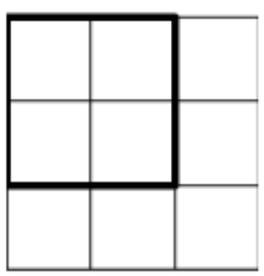
Correct answer: **3**

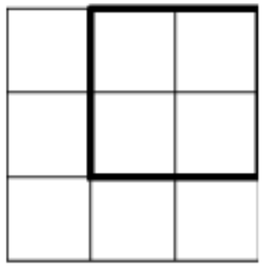
After applying the convolutional operator to the filter and the image, we get $\begin{bmatrix} 2 & -3 \\ 3 & 0 \end{bmatrix}$, so the sum of the entries is 2. Adding the bias gives 3, and $\max(0, 3) = 3$.

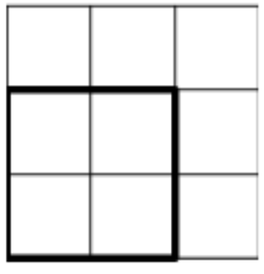
Striding

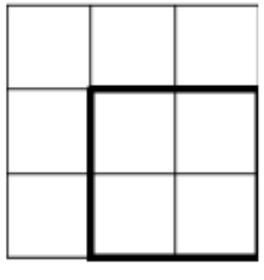
Next, we introduce **striding**. If we have a filter, we can apply it to a larger matrix at a bunch of possible positions.

For example, we can apply the 2×2 filter F to a 3×3 image M at 4 positions:









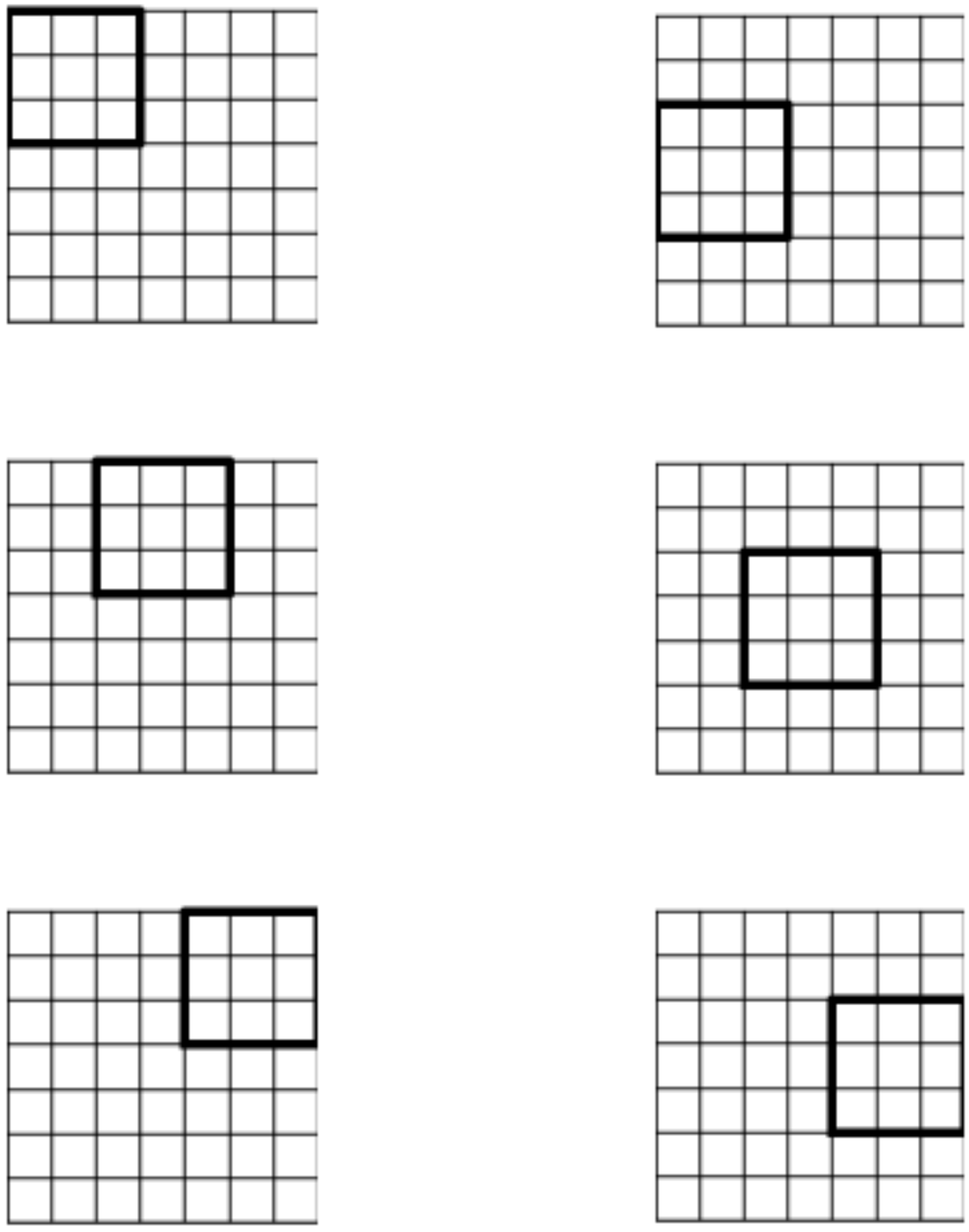
If we have an $N \times N$ filter and an $M \times M$ image, in how many positions can we apply the filter?

Correct answer: $(M - N + 1)^2$

There are $M - N + 1$ places to apply the filter along the horizontal axis, and $M - N + 1$ places to apply the filter along the vertical axis. Therefore our answer is $(M - N + 1)^2$.

Striding

We can also skip spaces when striding by specifying the number of squares to move when shifting our filter. For example, if we had a 3×3 filter F and a 7×7 image M, a stride $S = 2$ would be applied as follows:



Striding

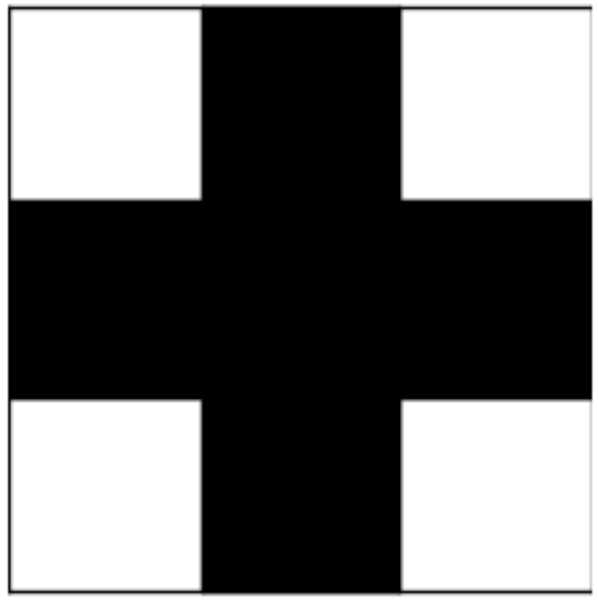
If we have a 20×20 filter and a 250×150 image, and we want to use strides of size 10, in how many positions will we apply the filter?

Correct answer: **336**

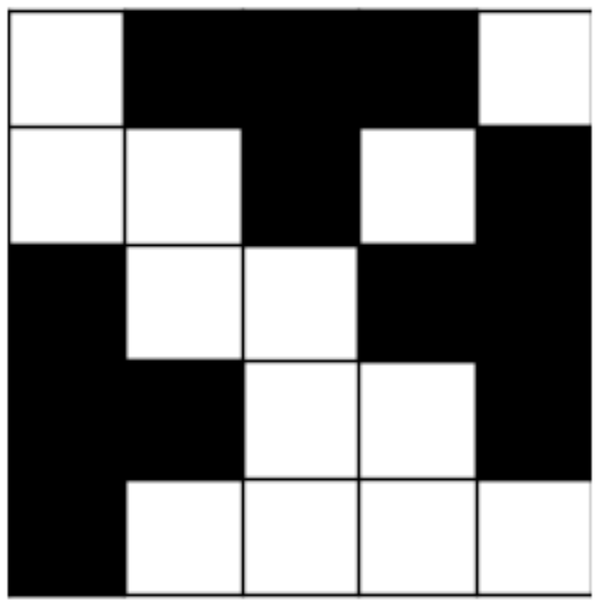
In general, if we have an $N \times N$ filter, a $W \times H$ image, and stride S, there are $\frac{W-N}{S} + 1$ places to apply the filter along the horizontal axis, and $\frac{H-N}{S} + 1$ to apply the filter along the vertical axis. So the total number of positions for our example is $\left(\frac{250-20}{10} + 1\right)\left(\frac{150-20}{10} + 1\right) = 336.$

Padding

But what if we have an object which is only partially in the image? For example, suppose that we have the following filter for a cross:

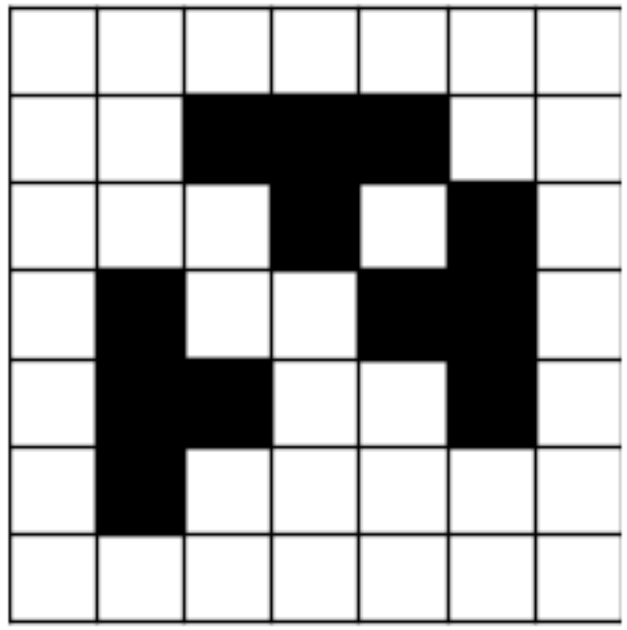


If we apply this filter to this image with our current method:



Padding

then we will miss all of the partial crosses on the sides of the image. So, we need to pad the image at the sides, so that we can catch these crosses as well.
If we use a pad side of $P = 1$, we pad our picture to look like this:



so now we will capture all the partial crosses.
Putting this all together, what will be the dimensions of the feature map for a filter with size $N \times N$ applied to an image with size $W \times H$ with padding P and stride length S?

Correct answer: $\left(\frac{W-N+2P}{S}+1\right) \times \left(\frac{H-N+2P}{S}+1\right)$

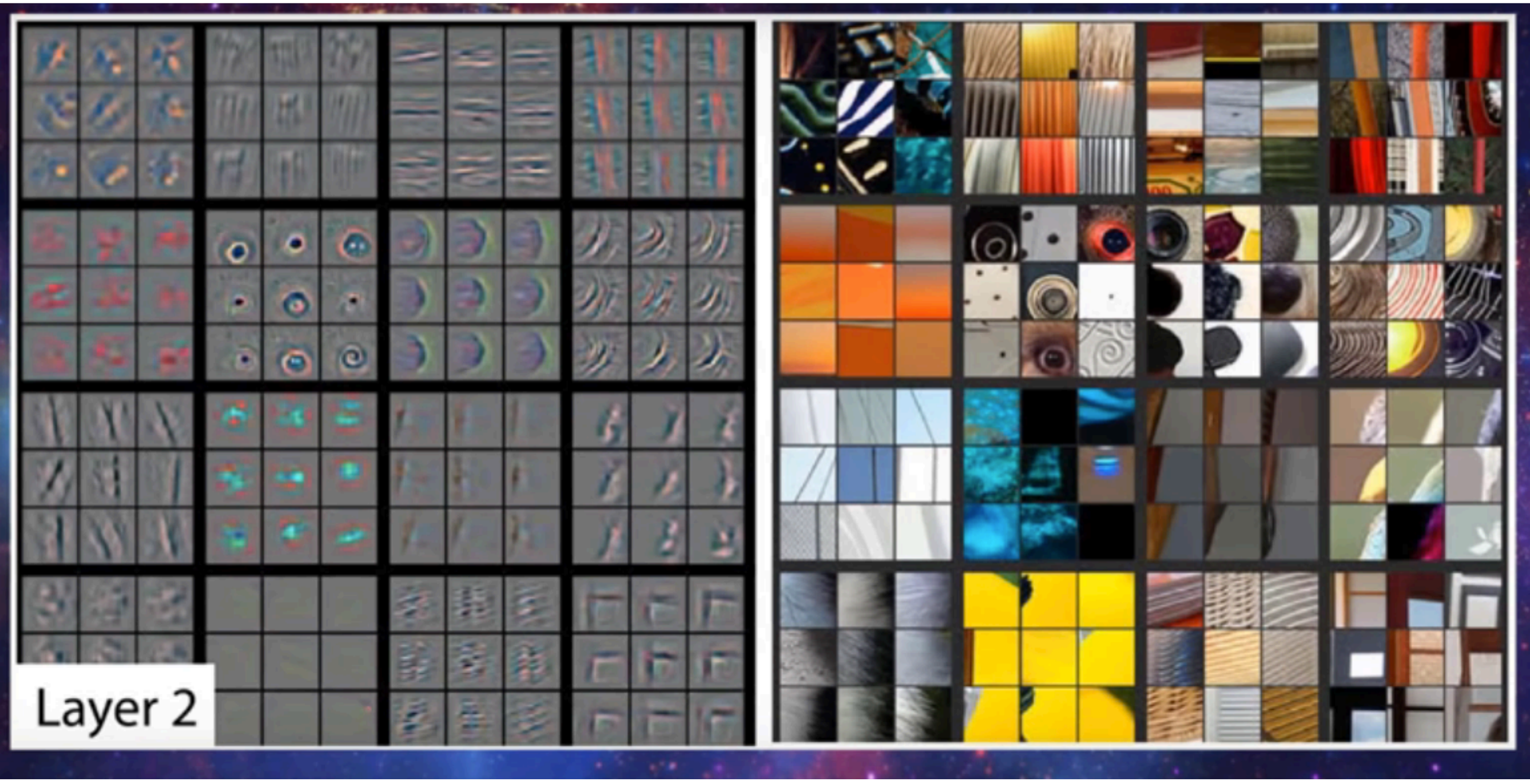
This is the same as the last question, except that the image has been padded along the horizontal and vertical axes by $2P$ pixels $(P$ on each side$)$. So our answer is

$$\left(\frac{W-N+2P}{S}+1\right) \times \left(\frac{H-N+2P}{S}+1\right).$$

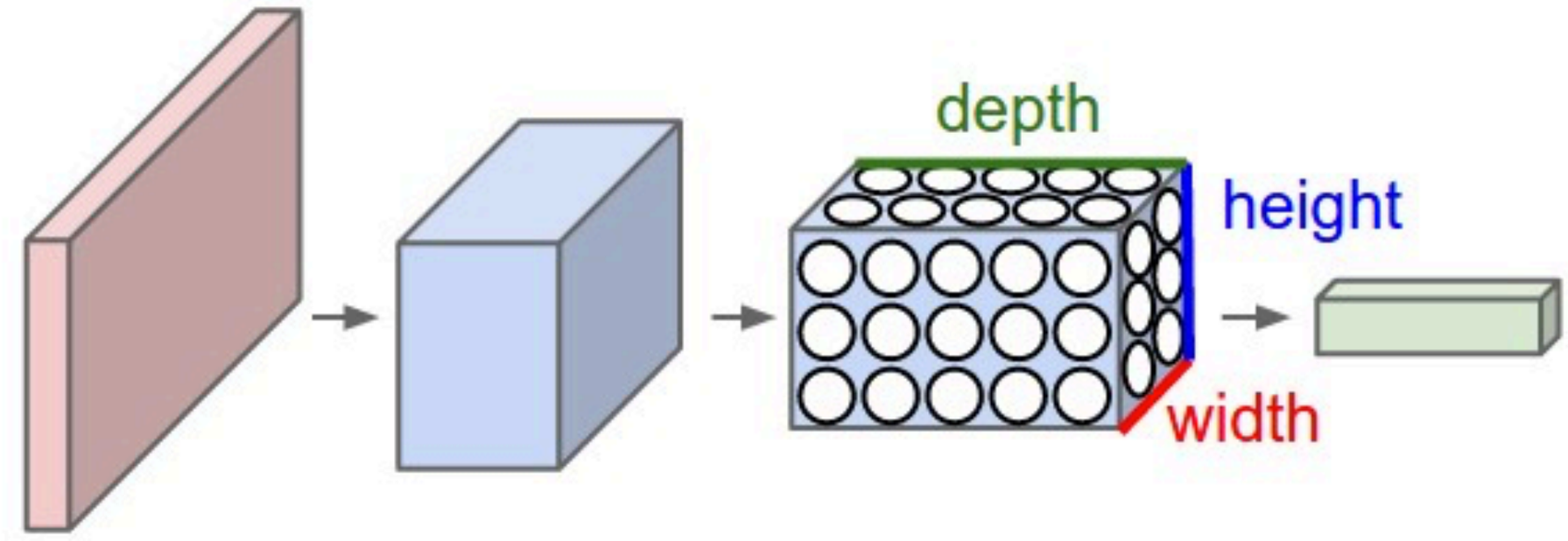
Layer 1 Simple fliters

Suppose we have 4 3x3 filters . If we convolve the image with each filter individually we see the output.

Layer 2 more advanced filters



When actually creating a CNN, we will want to have multiple filters to recognize different features at each convolutional step. This means that our output matrix has a **volume**: if we have K $N \times N$ filters, and apply them to a $W \times H$ image with P padding and S stride length, our output matrix will have dimensions $\left(\frac{W-N+2P}{S}+1\right) \times \left(\frac{H-N+2P}{S}+1\right) \times K$.



Suppose that we have applied three 5×5 filters to a 99×99 image with a stride of 1 and padding of 3, and gotten a matrix M_1. Suppose that we want to apply another convolution to M_1 with a stride of 2 and padding of 5, and we want the dimensions of this output matrix M_2 to be $50 \times 50 \times 6$. How many filters do we need, and what should their dimensions be?

Correct answer: **6 filters, dimensions** $13 \times 13 \times 3$

We need 6 filters, since the depth of M_2 is 6. Then, we need the first two dimensions of the filters to be 13×13 to get the first two dimensions of M_2 to be 50×50 (see previous section). Lastly, the depth of the filters must be 3, since they are being applied to the matrix M_1, which has depth 3.

Recall that the convolutional paradigm aims to combine simple features into increasingly more complex features. In practice, this means that we must stack many layers to recognize sufficiently rich objects. Therefore, we would like to **downsample** the image, rather than use a convolutional layer, if possible, to save computation time. This has the added benefit of working against overfitting by making our model *less* powerful--we now have fewer parameters to work with, so they must be individually more meaningful.

In **pooling**, we downsample the image to make it smaller. This is accomplished by going through each layer of an image, and replacing the $N \times N$ blocks inside it with $M \times M$ blocks, where $M < N$. After this process, the smaller block is designed to contain the most important information from the larger block, while taking up less memory.

We can apply the concept of striding from convolutions to max pooling as well. When downsampling across an image, we move the $N \times N$ square used to take samples in intervals of the stride length, just as we do with filters when taking convolutions. For this reason, the formula for number of datapoints after applying an $N \times N$ filter to an image also gives the number of $N \times N$ squares sampled in the downsampling process. This then corresponds with the number of $M \times M$ squares in the output.

Suppose that $N = 3$, $M = 1$, and our stride is $S = 2$. If we have a 45×25 input image, what will be the number of datapoints in the downsampled image after applying this pooling operation? Keep in mind that we are storing color and each pixel has a depth of three.

Correct answer: **792**

As with an $N \times N$ filter, if we are taking $N \times N$ samples from a $W \times H$ image with stride S, there are $\frac{W-N}{S} + 1$ places to apply the filter along the horizontal axis and $\frac{H-N}{S} + 1$ to apply the filter along the vertical axis.

So, using the formulas from our study of convolutions, we have $\left(\frac{45-3}{2} + 1\right)\left(\frac{25-3}{2} + 1\right) = 264$ samples taken on every layer of the image. Since $M = 1$, each sample simply corresponds with a 1×1 square in the output, which is a single datapoint. So, each layer will produce 264 data points.

Finally, pooling will separately take samples from each of the three layers, so our answer is $3 \times 264 = 792$.

We now introduce two different methods for performing this downsampling.

In **average pooling**, an $N \times N$ area is mapped to a single pixel, with value equal to the average of the original pixels.

In **max pooling**, an $N \times N$ area is mapped to a single pixel, with value equal to the maximum of the original pixels.

Max pooling is generally used over average pooling, since it has performed better in experiments.

In practice, N is chosen to be 2 or 3, M is always 1, and $S = 2$.

Why don't we choose N to be greater than 2 or 3?

Correct answer: **We lose too much information by having larger N**
If we try and reduce the size of the image by too much, then we are likely to lose features relevant to image recognition after we downsample.

As part of the ImageNet challenge, computers are tasked with classifying the object pictured in an image from one of 1000 categories. They are allowed to output their 5 most likely guesses for what the object is, and are successful if one of those matches correctly.

Suppose that our architecture currently contains a bunch of convolutional and pooling layers.

How should we modify our architecture to suit the ImageNet challenge?

A. Add a fully connected network at the end to predict what the object is based on the high-level filter activations.

B. Add fully connected networks to introduce information about what the object is between each convolutional layer.

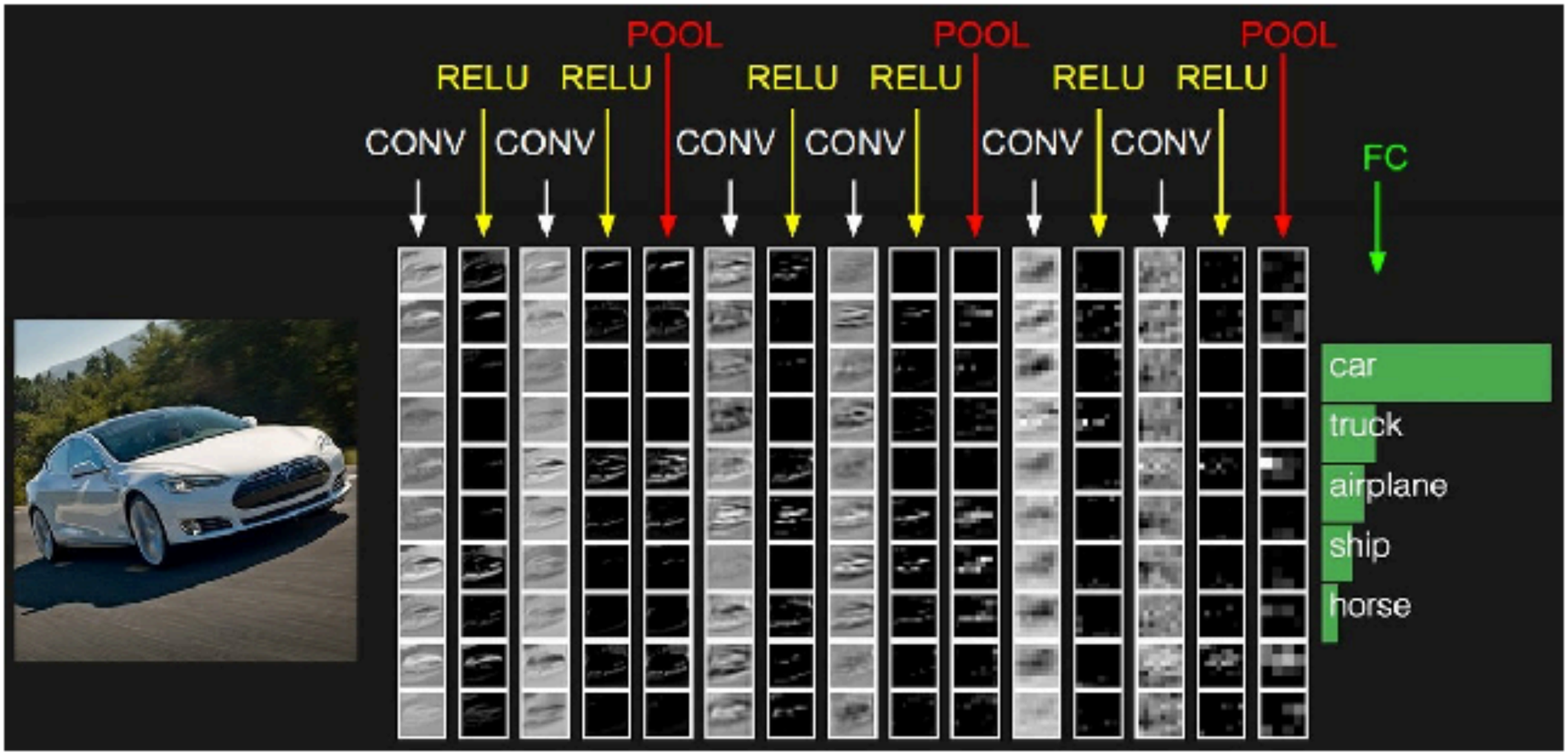
C. Add a fully connected network at the end to predict *probabilities* for what the object is based on the high-level filter activations.

D. Replace the ConvNet with a fully connected neural network.

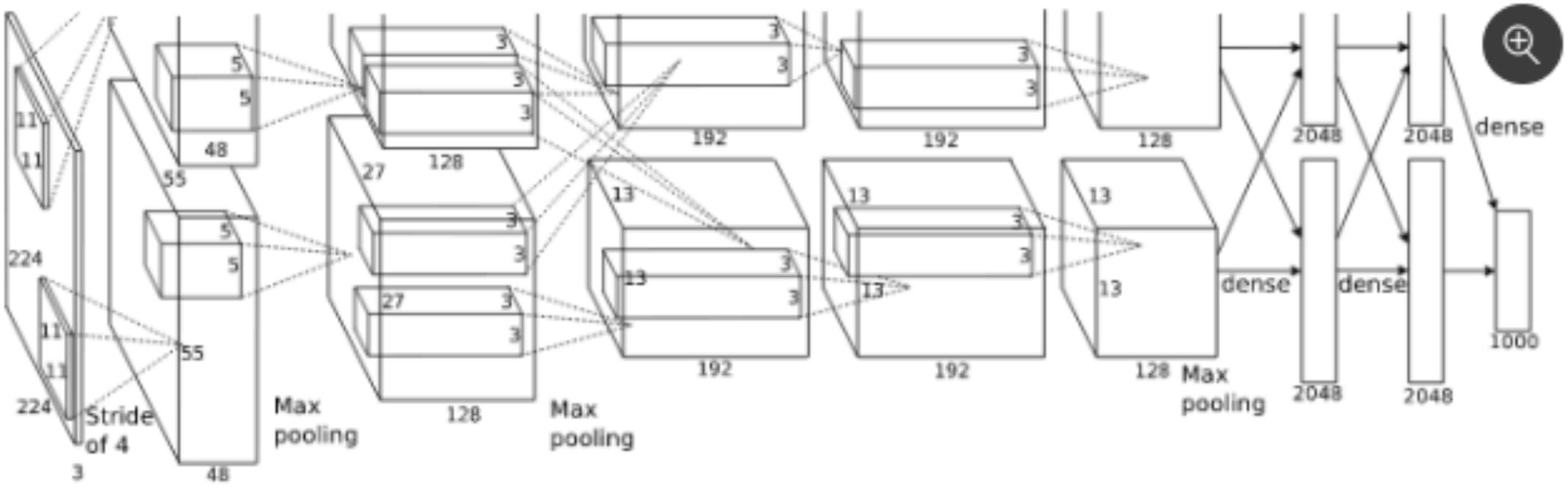
Correct answer: **C**

Once we have used our convolutional net to do image processing, it will output the activations from our high-level filters. We want to then use the activations to predict what the object is, so we tack an FCNN onto the end. We use this FCNN to output the probabilities, since we get 5 chances at predicting what the object is. Then, for our final answer, we'll output the objects with the highest probabilities of being in the image.

After putting all of our layers together, our CNN might look something like this:



AlexNet is a convolutional neural net architecture designed by Alex Krizhevsky, which won the ImageNet challenge in 2012 and launched CNNs into the cutting edge of the field of computer vision.



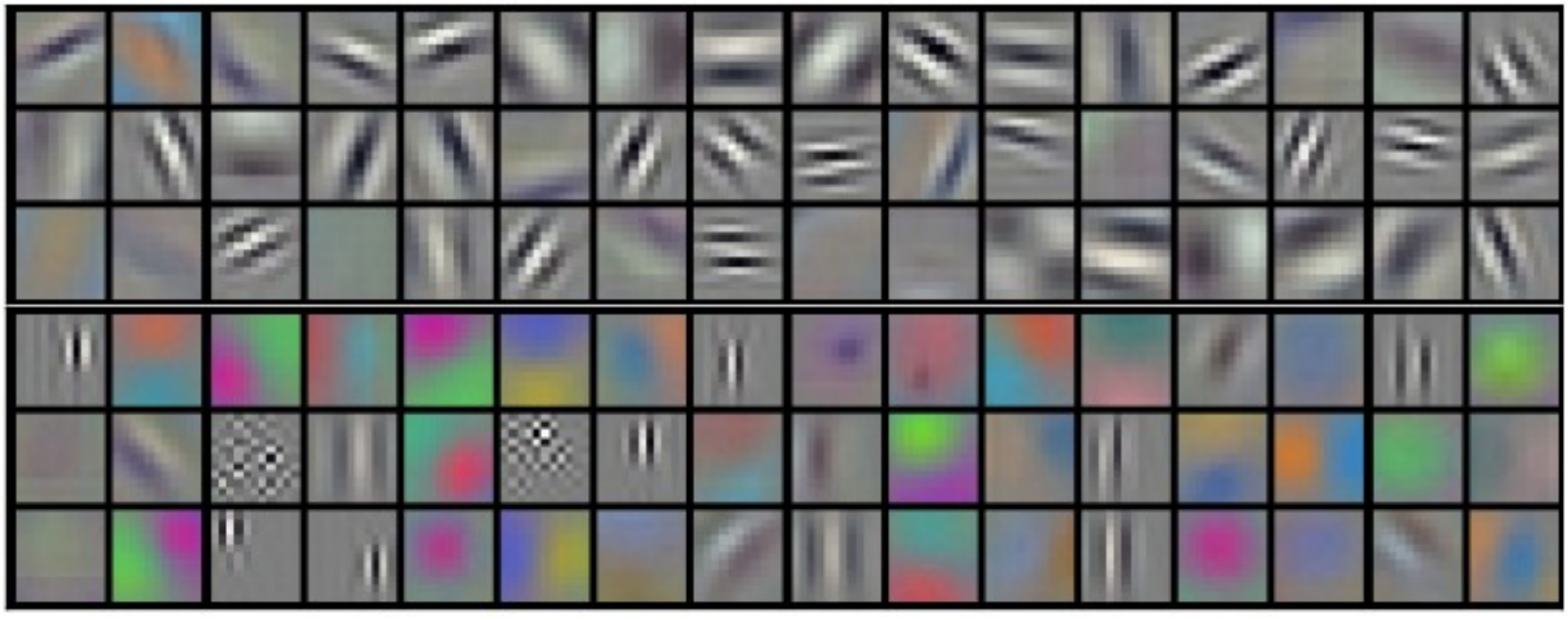
Above is a diagram of the architecture they used. It diverges into two halves since they trained on two GPUs separately. How many convolutional layers did they use?

Correct answer: **5**

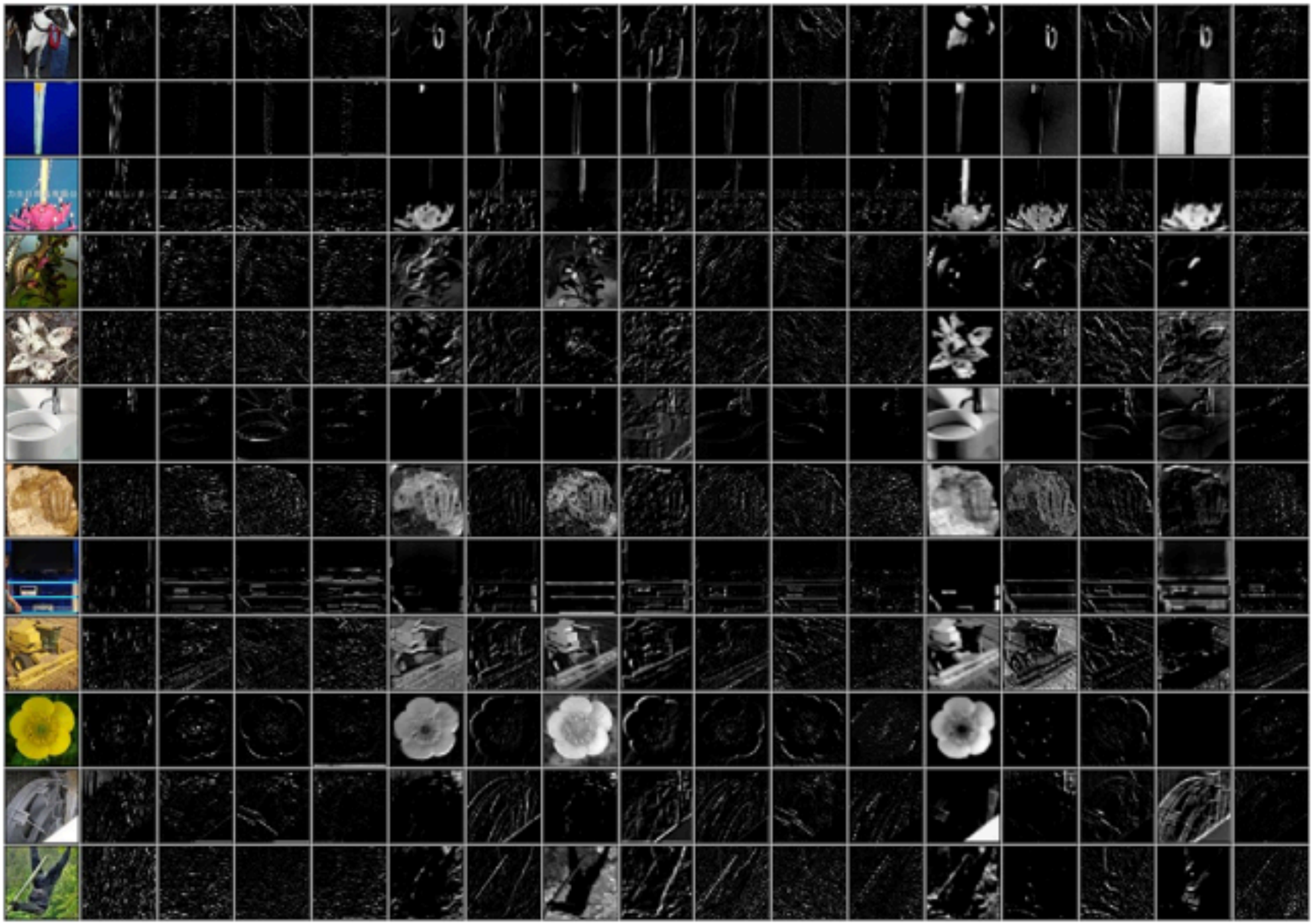
There are 5 convolutional layers pictured in the diagram. As an example, note that the first has 48 filters of size $11 \times 11 \times 3$ and a stride of 4, and is followed by a max pooling layer. Information about the remaining layers can be read off in a similar manner.

Krizhevsky's team used 5 convolutional layers, 3 pooling layers, and two FC layers. AlexNet was able to achieve an error rate of 17% on the ImageNet challenge, which was a massive improvement over the previous best of 28.2%. Since then, deeper and more refined CNNs have brought the error rate down to under 6.7%, better than the average person!

This picture shows the filters learned by the first layer of the CNNs in AlexNet:



This picture shows some of the feature maps of the images after the first layer:



CNNs have also been applied to other problems outside of object recognition, such as artistic style transfer, creating art, and even text-to-speech!

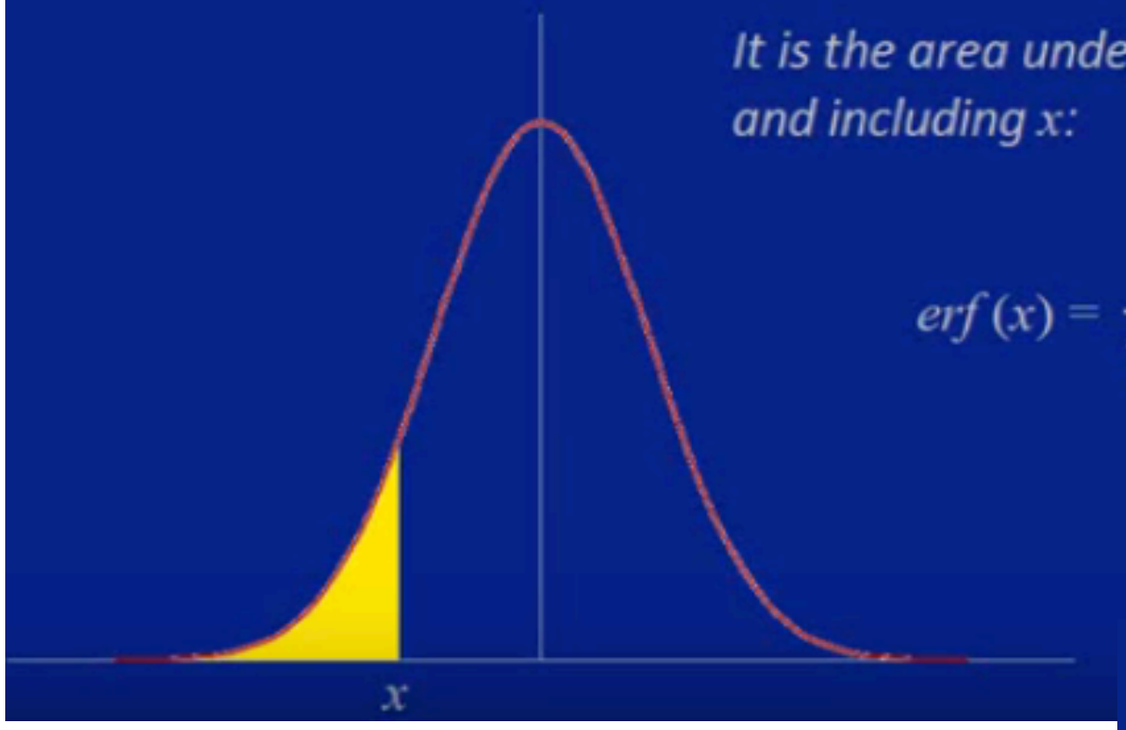
USEFUL

Error Function

The Error Function erf (x) is the probability that a number drawn at random from the Standard Normal Distribution (mean = 0, standard deviation = 1) will be no greater than x.

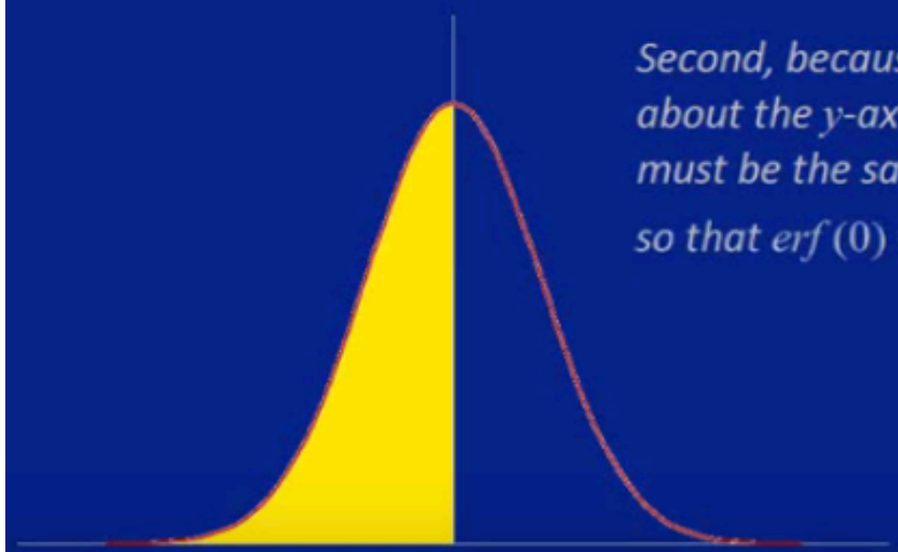
It is the area under the curve up to and including x:

$$erf(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{u^2}{2}} du$$



First recognize that because this is a probability function, the total area under the curve must be 1. This is just saying that a number randomly drawn from the distribution must lie somewhere along the number line.

Second, because the curve is symmetric about the y-axis, the area for all $x > 0$ must be the same for all $x < 0$, namely $\frac{1}{2}$; so that $erf(0) = \frac{1}{2}$.



You have a million grams of a radioactive element with a half-life of one minute. This means that every minute, the mass of the element halves. Of the following options, which is the shortest time after which you have less than a gram?

Explanation

Correct answer: **30 minutes**

After 15 minutes, the mass has halved 15 times, which means that we have

$$\frac{10^6}{2^{15}} > \frac{\left(2^3\right)^6}{2^{15}} = 2^3 > 1,$$

so we still have more than one gram at this time. However, after 30 minutes, we have

$$\frac{10^6}{2^{30}} < \frac{\left(2^4\right)^6}{2^{30}} = 2^{-6} < 1,$$

so 30 minutes is the right answer.

Upon hearing about this exploding/vanishing gradient problem, Bobby concludes that he could avoid this problem using $y = x$ as his activation function in a deep neural network. Indeed, using the chain rule for any composition of $y = x$ would not lead to a gradient that blows up or vanishes. Does Bobby's idea have a flaw?

Explanation

Correct answer: **Yes, the model would become a lot less expressive**

If we made the hidden neurons simply linear functions of the inputs, then we would have the composition of linear functions in computing our output. Since a linear function composed with a linear function is still a linear function, the hidden neurons would serve no purpose under this activation function, because we would be unable to get anything more complex than what linear regression would give us. Because this activation function amounts to performing linear regression, it would actually be much easier from a computational standpoint, but wouldn't be nearly as expressive as a deep neural network could be because the basis functions we're using can't capture complexity nearly as well.

Convolutional Operator

Hadamard Product

Definition [edit]

For two matrices A, B of the same dimension $m \times n$, the Hadamard product $A \circ B$ is a matrix of the same dimension as the operands, with elements given by

$$(A \circ B)_{i,j} = (A)_{i,j}(B)_{i,j}.$$

For matrices of different dimensions ($m \times n$ and $p \times q$, where $m \neq p$ or $n \neq q$ or both) the Hadamard product is undefined.

Example [edit]

For example, the Hadamard product for a 3×3 matrix A with a 3×3 matrix B is

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{bmatrix}.$$

Matrix Multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2