Machine Learning

Collected From Deeplizard

By:Mohamed Yossri Elwaly

Machine Learning Vs. Traditional Programming

Example: Analyzing the sentiment of a popular media outlet and classifying that sentiment as positive or negative.

Traditional Programming Approach

The algorithm may first look for particular words associated with a negative or positive sentiment.

With conditional statements, the algorithm would classify articles as positive or negative based on the words that it knows are positive or negative.

```
// pseudocode
let positive = [
    "happy",
    "thankful",
    "amazing"
];

let negative = [
    "can't",
    "won't",
    "sorry",
    "unfortunately"
];
```

These are arbitrarily chosen by the programmer. Once we have the list of positive and negative examples, one simple algorithm is to simply count up the occurrences of each type of word in a given article. Then, the article can be classified as positive or negative based on which word count is higher, the positive examples or the negative examples.

Machine Learning Approach

The algorithm analyzes given media data and learns the features that classify what a negative article looks like versus a positive article.With what it has learned, the algorithm can then classify new articles as positive or negative. As a machine learning programmer in this case, you won't be explicitly specifying the words for the algorithm to recognize. Instead, the algorithm will “learn” that certain words are positive or negative based on labels given to each article it examines

```
// pseudocode
let articles = [
    {
        label: "positive",
        data: "The lizard movie was great! I really liked..."
    },
    {
        label: "positive",
        data: "Awesome lizards! The color green is my fav..."
    },
    {
        label: "negative",
        data: "Total disaster! I never liked..."
    },
    {
        label: "negative",
        data: "Worst movie of all time!..."
    }
];
```

Deep learning

Deep learning is a sub-field of machine learning that uses algorithms inspired by the structure and function of the brain's neural networks.
we call them *artificial* neural networks (ANNs).

We often also use other terms to refer to ANNs. In the field of deep learning, the term *artificial neural network* (ANN) is used interchangeably with the following:

- net
- neural net
- model

What Does *Deep* Mean In Deep Learning?

To understand the term *deep* in deep learning, we need to first understand how ANNs are structured. Once we know this, we will be able to see that deep learning uses a specific type of ANN that we call a deep net or deep artificial neural network

Artificial neural network

An artificial neural network is a computing system that is comprised of a collection of connected units called neurons that are organized into what we call layers.

The connected neural units form the so-called network. Each connection between neurons transmits a signal from one neuron to the other. The receiving neuron processes the signal and signals to downstream neurons connected to it within the network. Note that neurons are also commonly referred to as ***nodes*** : **Nodes** are organized into what we call layers. At the highest level, there are three types of layers in every ANN:

1. Input layer
2. Hidden layers
3. Output layer

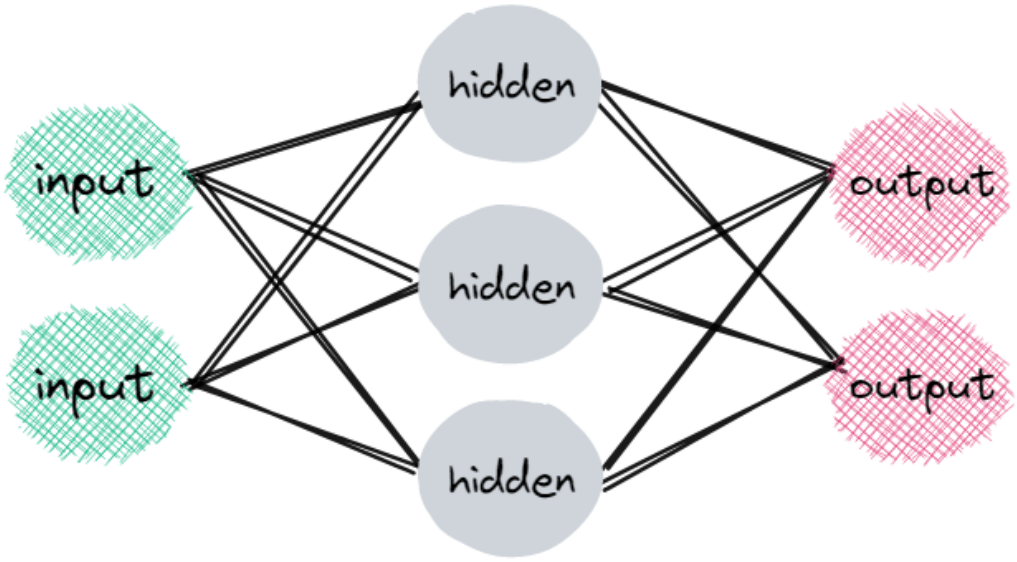
Different layers perform different kinds of transformations on their inputs. Data flows through the network starting at the input layer and moving through the hidden layers until the output layer is reached. This is known as a forward pass through the network. Layers positioned between the input and output layers are known as hidden layers.

Let's consider the number of nodes contained in each type of layer:

1. **Input layer** - One node for each component of the input data.
2. **Hidden layers** - Arbitrarily chosen number of nodes for each hidden layer.

3. **Output layer** - One node for each of the possible desired outputs.

Visualizing An Artificial Neural Network



Since this network has two nodes in the input layer, this tells us that each input to this network must have two dimensions, like for example *height* and *weight*.

Since this network has two nodes in the output layer, this tells us that there are two possible outputs for every input that is passed forward (left to right) through the network. For example, *overweight* or *underweight* could be the two output classes. Note that the output classes are also known as the prediction classes.

Keras Sequential Model

In Keras, we can build what is called a sequential model. Keras defines a sequential model as a sequential stack of linear layers. This is what we might expect as we have just learned that neurons are organized into layers.

This sequential model is Keras' implementation of an artificial neural network. Let's see now how a very simple sequential model is built using Keras.

First we import the required Keras classes.

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Then, we create a variable called `model`, and we set it equal to an instance of a `Sequential` object.

```
model = Sequential(layers)
```

To the constructor, we pass an array of `Dense` objects. Each of these objects called `Dense` are actually layers.

```
layers = [
    Dense(units=3, input_shape=(2,), activation='relu'),
    Dense(units=2, activation='softmax')
]
```

The word ***dense*** indicates that these layers are of type `Dense`. Dense is one particular type of layer, but there are many other types that we will see as we continue our deep learning journey.

For now, just understand that **dense** is the most basic kind of layer in an ANN and that each output of a dense layer is computed using every input to the layer.

types of layers

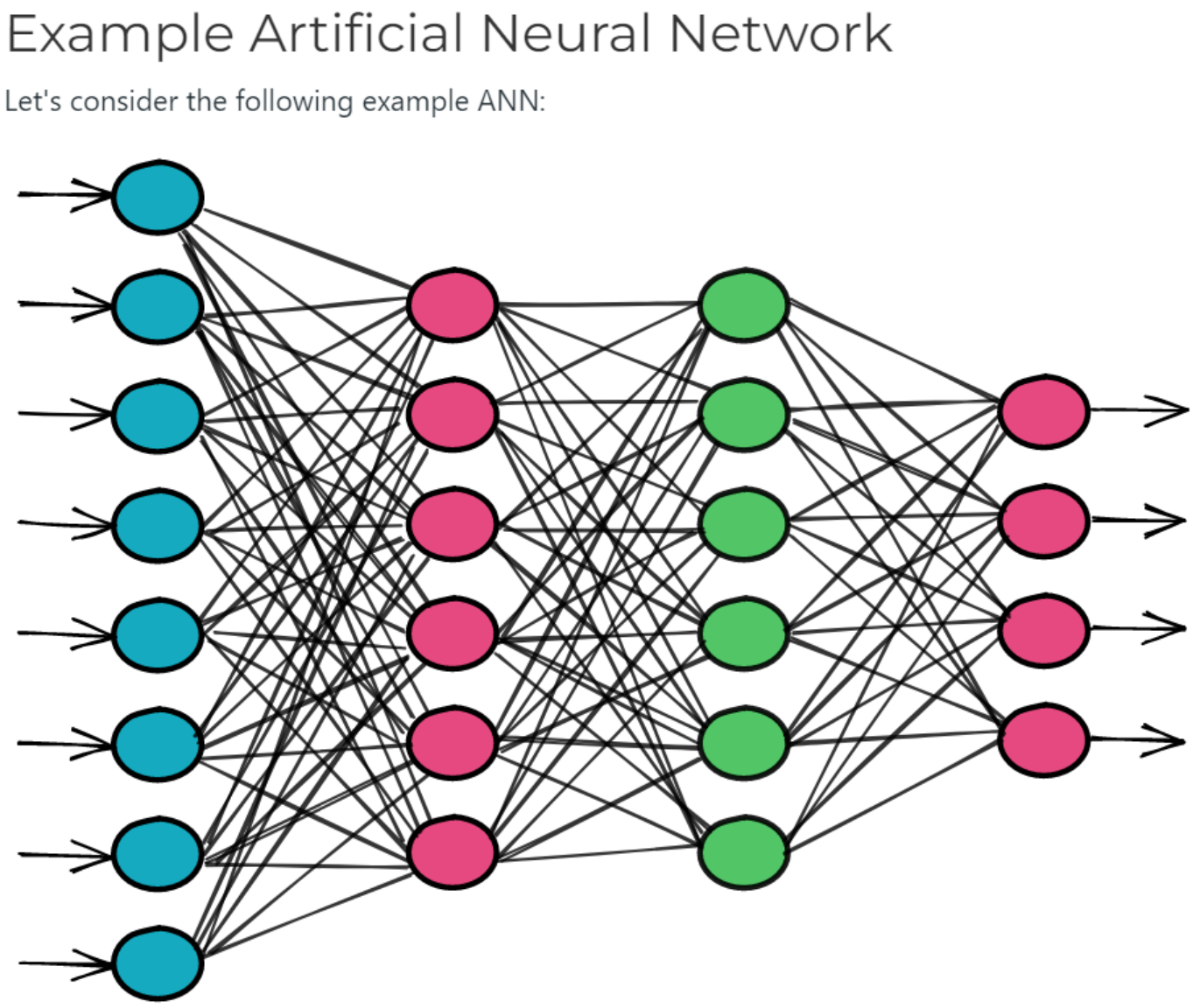
- Dense (or fully connected) layers
- Convolutional layers
- Pooling layers

- Recurrent layers
- Normalization layers

Why Have Different Types Of Layers?

Different layers perform different transformations on their inputs, and some layers are better suited for some tasks than others.

For example, a ***convolutional*** layer is usually used in models that are doing work with image data. ***Recurrent*** layers are used in models that are doing work with time series data, and fully connected layers, as the name suggests, fully connects each input to each output within its layer.



We can see that the first layer, the input layer, consists of eight nodes. Each of the eight nodes in this layer represents an individual feature from a given sample in our dataset. This tells us

that a single sample from our dataset consists of eight dimensions. When we choose a sample from our dataset and pass this sample to the model, each of the eight values contained in the sample will be provided to a corresponding node in the input layer.

Layer Weights

Each connection between two nodes has an associated weight, which is just a number.

Each weight represents the strength of the connection between the two nodes. When the network receives an input at a given node in the input layer, this input is passed to the next node via a connection, and the input will be multiplied by the weight assigned to that connection.

For each node in the second layer, a weighted sum is then computed with each of the incoming connections. This sum is then passed to an activation function, which performs some type of transformation on the given sum. For example, an activation function may transform the sum to be a number between zero and one. The actual transformation will vary depending on which activation function is used

```
node output = activation(weighted sum of inputs)
```

Forward Pass Through A Neural Network

Once we obtain the output for a given node, the obtained output is the value that is passed as input to the nodes in the next layer.

This process continues until the output layer is reached. The number of nodes in the output layer depends on the number of possible output or prediction classes we have. In our example, we have four possible prediction classes.

Suppose our model was tasked with classifying four types of animals. Each node in the output layer would represent one of four possibilities. For example, we could have cat, dog, llama or lizard. The categories or classes depend on how many classes are in our dataset.

For a given sample from the dataset, the entire process from input layer to output layer is called a forward pass through the network.

Finding The Optimal Weights

As the model learns, the weights at all connections are updated and optimized so that the input data point maps to the correct output prediction class

Defining The Neural Network In Code With Keras

Will start out by defining an list of `Dense` objects, our layers. This list will then be passed to the constructor of the sequential model.

```
layers = [
    Dense(units=6, input_shape=(8,), activation='relu'),
    Dense(units=6, activation='relu'),
    Dense(units=4, activation='softmax')
]
```

Notice how the first `Dense` object specified in the list is not the input layer. The first `Dense` object is the first hidden layer. The input layer is specified as a parameter to the first `Dense` object's constructor.

Our input shape is eight. This is why our input shape is specified as `input_shape=(8,)`. Our first hidden layer has six nodes as does our second hidden layer, and our output layer has four nodes.

For now, just note that we are using an activation function called relu `activation='relu'` for both of our hidden layers and an activation function called softmax `activation='softmax'` for our output layer

Our final product looks like this:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

layers = [
    Dense(units=6, input_shape=(8,), activation='relu'),
    Dense(units=6, activation='relu'),
    Dense(units=4, activation='softmax')
]

model = Sequential(layers)
```

What Is An Activation Function?

> In an artificial neural network, an activation function is a function that maps a inputs to its corresponding output.

This makes sense given the illustration we saw in the previous chapter . We took the weighted sum of each incoming connection for each node in the layer, and passed that weighted sum to an activation function.

```
node output = activation(weighted sum of inputs)
```

The activation function does some type of operation to transform the sum to a number that is often times between some lower limit and some upper limit. This transformation is often a non-linear transformation. Keep this in mind because this will come up again

What Do Activation Functions Do?

What's up with this activation function transformation? What's the intuition? To explain this, let's first look at some example activation functions.

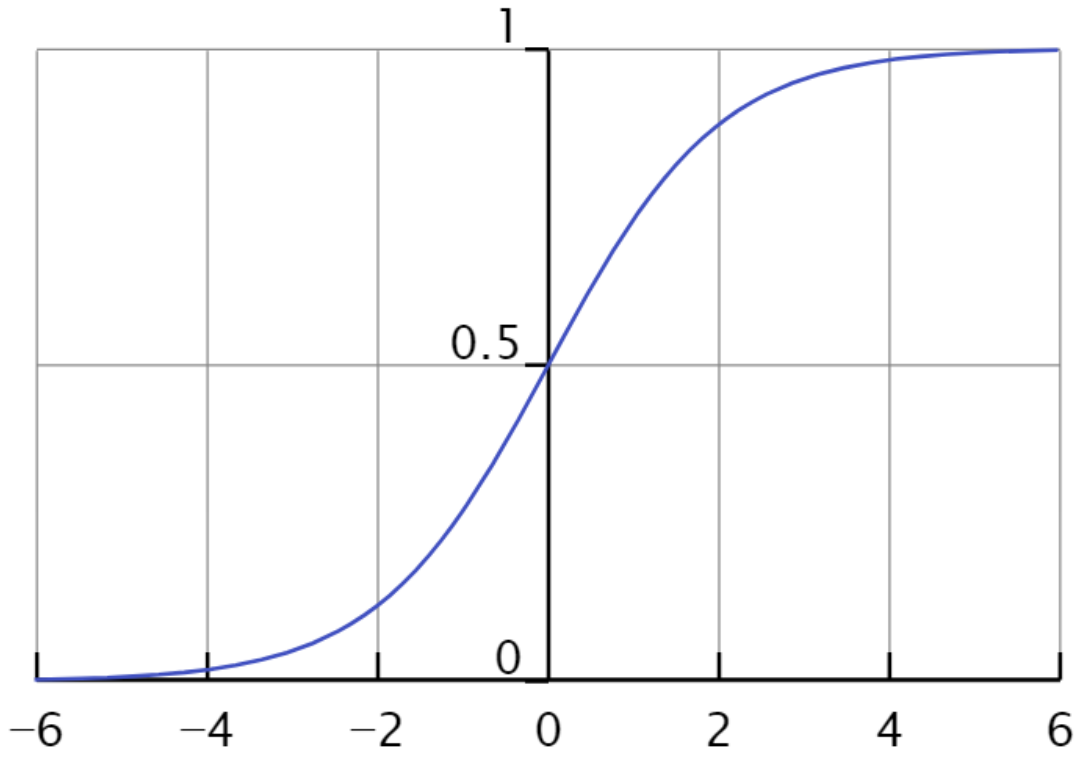
Sigmoid Activation Function

Sigmoid takes in an input and does the following:

- For most negative inputs, sigmoid will transform the input to a number very close to 0.
- For most positive inputs, sigmoid will transform the input into a number very close to 1.
- For inputs relatively close to 0, sigmoid will transform the input into some number between 0 and 1.

Mathematically, we write

$$sigmoid(x) = \frac{e^x}{e^x + 1}$$



So, for sigmoid, 0 is the lower limit, and 1 is the upper limit

Activation Function Intuition

if you smell something pleasant, like freshly baked cookies, certain neurons in your brain will fire and become activated. If you smell something unpleasant, like spoiled milk, this will cause other neurons in your brain to fire.

Deep within the folds of our brains, certain neurons are either firing or they're not. This can be represented by a 0 for not firing or a 1 for firing.

```
// pseudocode
if (smell.isPleasant()) {
    neuron.fire();
}
```

With the ***Sigmoid activation function*** in an artificial neural network, we have seen that the neuron can be between 0 and 1, and the closer to 1, the more activated that neuron is while the closer to 0 the less activated that neuron is.

Relu Activation Function

Now, it's not always the case that our activation function is going to do a transformation on an input to be between 0 and 1.

In fact, one of the most widely used activation functions today called *ReLU* doesn't do this. ReLU, which is short for *rectified linear unit*, transforms the input to the maximum of either 0 or the input itself.

$$relu(x) = max(0, x)$$

So if the input is less than or equal to 0, then relu will output 0. If the input is greater than 0, relu will then just output the given input.

```
// pseudocode
function relu(x) {
    if (x <= 0) {
        return 0;
    } else {
        return x;
    }
}
```

Why Do We Use Activation Functions?

To understand why we use activation functions, we need to first understand linear functions.

Suppose that f is a function on a set X.
Suppose that a and b are in X.
Suppose that x is a real number.

The function f is said to be a linear function if and only if:

$$f(a+b) = f(a) + f(b) \quad \textbf{AND} \quad f(xa) = xf(a)$$

An important feature of linear functions is that the composition of two linear functions is also a linear function. This means that, even in very deep neural networks, if we only had linear transformations of our data values during a forward pass, the learned mapping in our network from input to output would also be linear.

Typically, the types of mappings that we are aiming to learn with our deep neural networks are more complex than simple linear mappings.

*This is where **activation functions** come in*. Most activation functions are non-linear, and they are chosen in this way on purpose. Having non-linear activation functions allows our neural networks to compute arbitrarily complex functions.

Proof That Relu Is Non-Linear

For every real number x, we define a function f to be

$$f(x) = relu(x)$$

Suppose that a is a real number and that a<0.

Using the fact that a<0, we can see that

$$f(-1a) = max(0, -1a) > 0$$

$$(-1)f(a) = (-1)max(0, a) = 0$$

This allows us to conclude that

$$f(-1a) \neq (-1)f(a)$$

Therefore, we have shown that the function f fails to be linear.

Activation Functions In Code With Keras

Let's take a look at how to specify an activation function in a Keras Sequential model.

There are two basic ways to achieve this. First, we'll import our classes.

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Now, the first way to specify an activation function is in the constructor of the layer like so:

```
model = Sequential([
    Dense(units=5, input_shape=(3,), activation='relu')
])
```

In this case, we have a `Dense` layer and we are specifying relu as our activation function `activation='relu'` .

The second way is to add the layers and activation functions to our model after the model has been instantiated like so:

```
model = Sequential()
model.add(Dense(units=5, input_shape=(3,)))
model.add(Activation('relu'))
```

Remember that **node output = activation(weighted sum of inputs)**

For our example, this means that each output from the nodes in our Dense layer will be equal to the relu result of the weighted sums like

node output = relu(weighted sum of inputs)

What Is Training?

When we train a model, we're basically trying to solve an optimization problem. We're trying to optimize the weights within the model. Our task is to find the weights that most accurately map our input data to the correct output class. This mapping is what the network must *learn*.

Recall, we touched on this idea in our Chapter about layers. There, we showed how each connection between nodes has an arbitrary weight assigned to it. During training, these weights are iteratively updated and moved towards their optimal values.

Optimization Algorithm

The weights are optimized using what we call an optimization algorithm. The optimization process depends on the chosen optimization algorithm. We also use the term ***optimizer*** to refer to the chosen algorithm. The most widely known optimizer is called ***stochastic gradient descent***, or more simply, **SGD.**

When we have any optimization problem, we must have an optimization objective, so now let's consider what SGD's objective is in optimizing the model's weights.

The objective of SGD is to minimize some given function that we call a loss function. So, SGD updates the model's weights in such a way as to make this loss function as close to its minimum value as possible.

Loss Function

One common loss function is *mean squared error* (MSE), but there are several loss functions that we could use in its place. As deep learning practitioners, it's our job to decide which loss function to use. For now, let's just think of general loss functions, and later we'll look at specific loss functions in more detail.

Alright, but what *is* the actual loss we're talking about? Well, during training, we supply our model with data and the corresponding labels to that data.

For example, suppose we have a model that we want to train to classify whether images are either images of cats or images of dogs. We will supply our model with

images of cats and dogs along with the labels for these images that state whether each image is of a cat or of a dog.

Suppose we give one image of a cat to our model. Once the forward pass is complete and the cat image data has flowed through the network, the model is going to provide an output at the end. This will consist of what the model thinks the image is, either a cat or a dog.

In a literal sense, the output will consist of probabilities for cat or dog. For example, it may assign a 75% probability to the image being a cat, and a 25% probability to it being a dog. In this case, the model is assigning a higher likelihood to the image being of a cat than of a dog.

- 75% chance it's a cat
- 25% chance it's a dog

If we stop and think about it for a moment, this is very similar to how humans make decisions. Everything is a prediction!

The loss is the error or difference between what the network is predicting for the image versus the true label of the image, and SGD will to try to minimize this error to make our model as accurate as possible in its predictions.

After passing all of our data through our model, we're going to continue passing the same data over and over again. This process of repeatedly sending the same data through the network is considered *training*. During this training process is when the model will actually *learn*. More about learning in the next chapter. So, through this process that's occurring with SGD iteratively, the model is able to learn from the data.

What Does It Mean To Learn?

So what exactly does it mean for the model to *learn*?

Well, remember, when the model is initialized, the network weights are set to arbitrary values. We have also seen that, at the end of the network, the model will provide the output for a given input.

Once the output is obtained, the loss (or the error) can be computed for that specific output by looking at what the model predicted versus the true label. The loss computation depends on the chosen loss function

Gradient Of The Loss Function

After the loss is calculated, the gradient of this loss function is computed with respect to each of the weights within the network. Note, *gradient* is just a word for the derivative of a function of several variables.

Continuing with this explanation, let's focus in on only one of the weights in the model.

At this point, we've calculated the loss of a single output, and we calculate the gradient of that loss with respect to our single chosen weight. This calculation is done using a technique called ***backpropagation***, which is covered in full detail in the next chapter

Once we have the value for the gradient of the loss function, we can use this value to update the model's weight. The gradient tells us which direction will move the loss towards the minimum, and our task is to move in a direction that lowers the loss and steps closer to this minimum value.

Learning Rate

We then multiply the gradient value by something called a *learning rate*. A learning rate is a small number usually ranging between 0.01 and 0.0001, but the actual value can vary.

The learning rate tells us how large of a step we should take in the direction of the minimum.
Just keep this in mind for now, and we'll look more closely at learning rates soon

Updating The Weights

Alright, so we multiply the gradient with the learning rate, and we subtract this product from the weight, which will give us the new updated value for this weight.

$$new\ weight = old\ weight - (learning\ rate * gradient)$$

In this discussion, we just focused on one single weight to explain the concept, but this same process is going to happen with each of the weights in the model each time data passes through it.

The only difference is that when the gradient of the loss function is computed, the value for the gradient is going to be different for each weight because the gradient is being calculated with respect to each weight.

So now imagine all these weights being iteratively updated with each epoch. The weights are going to be incrementally getting closer and closer to their optimized values while SGD works to minimize the loss function.

The Model Is Learning

This updating of the weights is essentially what we mean when we say that the model is learning. It's learning what values to assign to each weight based on how those incremental changes are affecting the loss function. As the weights change, the network is getting smarter in terms of accurately mapping inputs to the correct output.

Training In Code With Keras

Let's begin by importing the required classes:

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy
```

Next, we define our model:

```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='sigmoid')
])
```

Before we can train our model, we must compile it like so:

```
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

To the `compile()` function, we are passing the optimizer, the loss function, and the metrics that we would like to see. Notice that the optimizer we have specified is called *Adam*. `Adam` is just a variant of SGD. Inside the `Adam` constructor is where we specify the learning rate, and in this case `Adam(learning_rate=.0001)`, we have chosen `0.0001`.

Finally, we fit our model to the data. Fitting the model to the data means to train the model on the data. We do this with the following code:

```
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

`scaled_train_samples` is a numpy array consisting of the training samples.

`train_labels` is a numpy array consisting of the corresponding labels for the training samples.

`batch_size=10` specifies how many training samples should be sent to the model at once.

`epochs=20` means that the complete training set (all of the samples) will be passed to the model a total of 20 times.

`shuffle=True` indicates that the data should first be shuffled before being passed to the model.

`verbose=2` indicates how much logging we will see as the model trains.

Running this code gives us the following output:

```
Epoch 1/20 0s - loss: 0.6400 - acc: 0.5576
Epoch 2/20 0s - loss: 0.6061 - acc: 0.6310
Epoch 3/20 0s - loss: 0.5748 - acc: 0.7010
Epoch 4/20 0s - loss: 0.5401 - acc: 0.7633
Epoch 5/20 0s - loss: 0.5050 - acc: 0.7990
Epoch 6/20 0s - loss: 0.4702 - acc: 0.8300
Epoch 7/20 0s - loss: 0.4366 - acc: 0.8495
Epoch 8/20 0s - loss: 0.4066 - acc: 0.8767
Epoch 9/20 0s - loss: 0.3808 - acc: 0.8814
Epoch 10/20 0s - loss: 0.3596 - acc: 0.8962
Epoch 11/20 0s - loss: 0.3420 - acc: 0.9043
Epoch 12/20 0s - loss: 0.3282 - acc: 0.9090
Epoch 13/20 0s - loss: 0.3170 - acc: 0.9129
Epoch 14/20 0s - loss: 0.3081 - acc: 0.9210
Epoch 15/20 0s - loss: 0.3014 - acc: 0.9190
Epoch 16/20 0s - loss: 0.2959 - acc: 0.9205
Epoch 17/20 0s - loss: 0.2916 - acc: 0.9238
Epoch 18/20 0s - loss: 0.2879 - acc: 0.9267
Epoch 19/20 0s - loss: 0.2848 - acc: 0.9252
Epoch 20/20 0s - loss: 0.2824 - acc: 0.9286
```

Loss Functions In Neural Networks

The loss function is what SGD is attempting to minimize by iteratively updating the weights in the network.

At the end of each epoch during the training process, the loss will be calculated using the network's output predictions and the true labels for the respective input.

Suppose our model is classifying images of cats and dogs, and assume that the label for cat is 0 and the label for dog is 1.

- cat: 0
- dog: 1

Now suppose we pass an image of a cat to the model, and the provided output is 0.25. In this case, the difference between the model's prediction and the true label is $0.25 - 0.00 = 0.25$. This difference is also called the *error*.

$$\text{error} = 0.25 - 0.00 = 0.25$$

This process is performed for every output. For each epoch, the error is accumulated across all the individual outputs.

Let's look at a loss function that is commonly used in practice called the *mean squared error* (MSE).

Mean Squared Error (MSE)

For a single sample, with MSE, we first calculate the difference (the error) between the provided output prediction and the label. We then square this error. For a single input, this is all we do.

$$\text{MSE}(\text{input}) = (\text{output} - \text{label})(\text{output} - \text{label})$$

If we passed multiple samples to the model at once (a batch of samples), then we would take the mean of the squared errors over all of these samples.

This was just illustrating the math behind how one loss function, MSE, works. There are several different loss functions that we could work with though.

The general idea that we just showed for calculating the error of individual samples will hold true for all of the different types of loss functions. The implementation of what we actually *do* with each of the errors will be dependent upon the algorithm of the given loss function we're using.

For example, we averaged the squared errors to calculate MSE, but other loss functions will use other algorithms to determine the value of the loss.

If we passed our entire training set to the model at once, then the process we just went over for calculating the loss will occur at the end of each epoch during training.

If we split our training set into batches, and passed batches one at a time to our model, then the loss would be calculated on each batch.

With either method, since the loss depends on the weights, we expect to see the value of the loss change each time the weights are updated. Given that the objective of SGD is to minimize the loss, we want to see our loss decrease as we run more epochs.

Loss Functions In Code With Keras

```
model = Sequential([
    Dense(16, input_shape=(1,), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='sigmoid')
])
```

Once we have our model, we can compile it like so:

```
model.compile(
    Adam(learning_rate=.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Looking at the second parameter of the call to `compile()`, we can see the specified loss function `loss='sparse_categorical_crossentropy'`.

In this example, we're using a loss function called *sparse categorical crossentropy*, but there are several others that we could choose, like MSE, for instance.

The currently available loss functions for Keras are as follows:

- **mean_squared_error**
- **mean_absolute_error**
- **mean_absolute_percentage_error**
- **mean_squared_logarithmic_error**
- **squared_hinge**
- **hinge**
- **categorical_hinge**
- **logcosh**
- **categorical_crossentropy**
- **sparse_categorical_crossentropy**
- **binary_crossentropy**
- **kullback_leibler_divergence**
- **poisson**
- **cosine_proximity**

Introducing The Learning Rate

We know that the objective during training is for SGD to minimize the loss between the actual output and the predicted output from our training samples. The path towards this minimized loss is occurring over several steps.

Recall that we start the training process with arbitrarily set weights, and then we incrementally update these weights as we move closer and closer to the minimized loss.

Now, the size of these steps we're taking to reach our minimized loss is going to depend on the learning rate. Conceptually, we can think of the learning rate of our model as the *step size*.

Before going further, let's first pause for a quick refresher. We know that during training, after the loss is calculated for our inputs, the gradient of that loss is then calculated with respect to each of the weights in our model.

Once we have the value of these gradients, this is where the idea of our learning rate comes in. The gradients will then get multiplied by the learning rate.

This learning rate is a small number usually ranging ***between*** 0.01 ***and*** 0.0001, but the actual value can vary, and any value we get for the gradient is going to become pretty small once we multiply it by the learning rate.

Updating The Network's Weights

Alright, so we get the value of this product for each gradient multiplied by the learning rate, and we then take each of these values and update the respective weights by subtracting this value from them.

*new weight = old weight - (learning rate * gradient)*

We ditch the previous weights that were set on each connection and update them with these new values.

The value we choose for the learning rate is going to require some testing. The learning rate is another one of those *hyperparameters* that we have to test and tune with each model before we know exactly where we want to set it, but as mentioned earlier, a typical guideline is to set it somewhere between 0.01 and 0.0001.

When setting the learning rate to a number on the higher side of this range, we risk the possibility of overshooting. This occurs when we take a step that's too large in the direction of the minimized loss function and shoot past this minimum and miss it.

To avoid this, we can set the learning rate to a number on the lower side of this range. With this option, since our steps will be really small, it will take us a lot longer to reach the point of minimized loss.

Overall, the act of choosing between a higher learning rate and a lower learning rate leaves us with this kind of trade-off idea.

Alright, so now we should have an idea about what the learning rate is and how it fits into the overall process of training.

Learning Rates In Keras

```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    Dense(units=2, activation='sigmoid')
])

model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

With the line where we're compiling our model, we can see that the first parameter we're specifying is our optimizer. In this case, we're using `Adam` as the optimizer for this model.
Now to our optimizer, we can optionally pass our learning rate by specifying the `learning_rate` parameter. We can see that here we're specifying `0.0001` as the learning rate.

We mentioned that this `learning_rate` parameter is optional. If we don't explicitly set it, then the default learning rate that Keras has assigned to this particular optimizer will be set. To see what this default learning rate is, you'll need to check the Keras documentation for the optimizer you're specifying.

There's also another way we can specify the learning rate. After compiling our model, we can set the learning rate by setting `model.optimizer.learning_rate` to our designated value.

```
model.optimizer.learning_rate = 0.01
```

Here we can see that we're setting it to `0.01`. Now, if we print the value of our learning rate, we can see it has now changed from `.0001` to `.01`.

```
> model.optimizer.learning_rate
0.01
```

Datasets For Deep Learning

For training and testing purposes for our model, we should have our data broken down into three distinct datasets. These datasets will consist of the following:

- Training set
- Validation set
- Test set

Training Set

The training set is what it sounds like. It's the set of data used to train the model. During each epoch, our model will be trained over and over again on this same data in our training set, and it will continue to learn about the features of this data.

The hope with this is that later we can deploy our model and have it accurately predict on new data that it's never seen before. It will be making these predictions based on what it's learned about the training data. Ok, now let's discuss the validation set.

Validation Set

The validation set is a set of data, separate from the training set, that is used to validate our model during training. This validation process helps give information that may assist us with adjusting our hyperparameters.

Recall how we just mentioned that with each epoch during training, the model will be trained on the data in the training set. Well, it will also simultaneously be validated on the data in the validation set.

We know from our pervious chapters that during the training process, the model will be classifying the output for each input in the training set. After this classification occurs, the loss will then be calculated, and the weights in the model will be adjusted. Then, during the next epoch, it will classify the same input again.

Now, also during training, the model will be classifying each input from the validation set as well. It will be doing this classification based only on what it's learned about the data it's being trained on in the training set. The weights will not be updated in the model based on the loss calculated from our validation data.

Remember, the data in the validation set is separate from the data in the training set. So when the model is validating on this data, this data does not consist of samples that the model already is familiar with from training.

One of the major reasons we need a validation set is to ensure that our model is not overfitting to the data in the training set. We'll discuss overfitting and underfitting in detail at a later time. But the idea of overfitting is that our model becomes really good at being able to classify data in the training set, but it's unable to generalize and make accurate classifications on data that it wasn't trained on.

During training, if we're also validating the model on the validation set and see that the results it's giving for the validation data are just as good as the results it's giving for the training data, then we can be more confident that our model is not overfitting.

The validation set allows us to see how well the model is generalizing during training.

On the other hand, if the results on the training data are really good, but the results on the validation data are lagging behind, then our model is overfitting. Now let's move on to the test set.

Test Set

The test set is a set of data that is used to test the model after the model has already been trained. The test set is separate from both the training set and validation set.

After our model has been trained and validated using our training and validation sets, we will then use our model to predict the output of the unlabeled data in the test set.

One major difference between the test set and the two other sets is that *the test set should not be labeled*. The training set and validation set have to be labeled so that we can see the metrics given during training, like the loss and the accuracy from each epoch.

When the model is predicting on unlabeled data in our test set, this would be the same type of process that would be used if we were to deploy our model out into the field.

The test set provides a final check that the model is generalizing well before deploying the model to production.

For example, if we're using a model to classify data without knowing what the labels of the data are beforehand, or with never have being shown the exact data it's going to be classifying, then of course we wouldn't be giving our model labeled data to do this.

The entire goal of having a model be able to classify is to do it without knowing what the data is beforehand.

The ultimate goal of machine learning and deep learning is to build models that are able to generalize well.

Deep Learning Datasets In Summary

Dataset	Updates Weights	Description
Training set	Yes	Used to train the model. The goal of training is to fit the model to the training set while still generalizing to unseen data.
Validation set	No	Used during training to check how well the model is generalizing.
Test set	No	Used to test the model's final ability to generalize before deploying to production.

Predicting With A Neural Network

Passing Samples With No Labels

For predicting, essentially what we're doing is passing our unlabeled test data to the model and having the model predict on what it thinks about each sample in our test data. These predictions are occurring based on what the model learned during training.

Predictions are based on what the model learned during training.

For example, suppose we trained a model to classify different breeds of dogs based on dog images. For each sample image, the model outputs which breed it thinks is most likely.

Now, suppose our test set contains images of dogs our model hasn't seen before. We pass these samples to our model, and ask it to predict the output for each image. Remember, the model does not have access to the labels for these images.

This process will tell us how well our model performs on data it hasn't seen before based on how well its predictions match the true labels for the data.

This process will also help give us some insight on what our model has or hasn't learned. For example, suppose we trained our model only on images of large dogs, but our test set has some images of small dogs. When we pass a small dog to our model, it likely isn't going to do well at predicting what breed the dog is, since it's not been trained very well on smaller dogs in general.

This means that we need to make sure that our training and validation sets are representative of the actual data we want our model to be predicting on.

Note that it is possible to pass unlabeled data to our model for predictions prior to training, however, we would expect the model to perform poorly with these predictions since it has not yet learned how to classify the data in the training set.

Deploying The Model In The Real World (Production)

if we deployed this neural network for classifying dog breeds to a website that anyone could visit and upload an image of their dog, then we'd want to be predicting the breed of the dog based on the image.

This image would likely not have been one that was included in our training, validation, or test sets, so this prediction would be occurring with true data from out in the field.

Using A Keras Model To Get A Prediction

```
predictions = model.predict(
    x=scaled_test_samples,
    batch_size=10,
    verbose=0
)
```

The first item we have here is a variable we've called `predictions`. We're assuming that we already have our model built and trained. Our model in this example is the object called `model`. We're setting `predictions` equal to `model.predict()`.

This `predict()` function is what we call to actually have the model make predictions. To the `predict()` function, we're passing the variable called `scaled_test_samples`. This is the variable that's holding our test data.

We set our `batch_size` here arbitrarily to `10`. We set the verbosity, which is how much we want to see printed to the screen when we run these predictions, to `0` here to show nothing.

Ok, so we ran our predictions. Now let's look at our output.

```
for p in predictions:
    print(p)

[ 0.7410683  0.2589317]
[ 0.14958295  0.85041702]
...
[ 0.87152088  0.12847912]
[ 0.04943148  0.95056852]
```

For this sample model, we have two output categories, and we're just printing each prediction from each sample in our test set, which is stored in our `predictions` variable.

We see we have two columns here. These represent the two output categories, and are showing us probabilities for each category. These are the actual predictions. Let's call the categories `0` and `1` for simplicity.

For example, for the first sample in our test set, the model is assigning a `74%` probability that the sample falls into category `0` and only a `26%` probability that it falls into category `1`.

Overfitting In A Neural Network

Overfitting occurs when our model becomes really good at being able to classify or predict on data that was included in the training set, but is not as good at classifying data that it wasn't trained on. So essentially, the model has overfit the data in the training set.

How To Spot Overfitting

We can tell if the model is overfitting based on the metrics that are given for our training data and validation data during training. We previously saw that when we specify a validation set during training, we get metrics for the validation accuracy and loss, as well as the training accuracy and loss.

If the validation metrics are considerably worse than the training metrics, then that is indication that our model is overfitting.

We can also get an idea that our model is overfitting if during training, the model's metrics were good, but when we use the model to predict on test data, it doesn't accurately classify the data in the test set.

The concept of overfitting boils down to the fact that the model is unable to generalize well. It has learned the features of the training set extremely well, but if we give the model any data that slightly deviates from the exact data used during training, it's unable to generalize and accurately predict the output.

Underfitting In A Neural Network

Underfitting is on the opposite end of the spectrum. A model is said to be underfitting when it's not even able to classify the data it was trained on, let alone data it hasn't seen before.

A model is said to be underfitting when it's not able to classify the data it was trained on.

Ways of Reducing Overfitting

Overfitting is an incredibly common issue. Let's look at some techniques to reduce it

1-Adding More Data To The Training Set

The easiest thing we can do, as long as we have access to it, is to add more data. The more data we can train our model on, the more it will be able to learn from the training set. Also, with more data, we're hoping to be adding more diversity to the training set as well.

2-Data Augmentation

Another technique we can deploy to reduce overfitting is to use data augmentation. This is the process of creating additional augmented data by reasonably modifying the data in our training set. For image data, for example, we can do these modifications by:

- Horizontal flip Vertical flip
- Zoom in Zoom out
- Cropping Rotation
- Color variations

The general idea of data augmentation allows us to add more data to our training set that is similar to the data that we already have, but is just reasonably modified to some degree so that it's not the exact same, For example, if most of our dog images were dogs facing to the left, then it would be a reasonable modification to add augmented flipped images so that our training set would also have dogs that faced to the right.

3-Reduce The Complexity Of The Model

Something else we can do to reduce overfitting is to reduce the complexity of our model. We could reduce complexity by making simple changes, like removing some layers from the model, or reducing the number of neurons in the layers. This may help our model generalize better to data it hasn't seen before.

4-Dropout

The general idea it will randomly ignore some subset of nodes in a given layer during training, i.e., it *drops out* the nodes from the layer. Hence, the name *dropout*. This will prevent these dropped out nodes from participating in producing a prediction on the data ,This technique may also help our model to generalize better to data it hasn't seen before

Ways of Reducing Underfitting

1-Increase The Complexity Of The Model

This is the exact opposite of a technique we gave to reduce overfitting. If our data is more complex, and we have a relatively simple model, then the model may not be sophisticated enough to be able to accurately classify or predict on our complex data.

We can increase the complexity of our model by doing things such as:

- Increasing the number of layers in the model.
- Increasing the number of neurons in each layer.
- Changing what type of layers we're using and where.

2-Add More Features To The Input Samples

add more features to the input samples in our training set if we can. These additional features may help our model classify the data better.
For example, say we have a model that is attempting to predict the price of a stock based on the last three closing prices of this stock. So our input would consist of three features:

- day 1 close day 2 close day 3 close

If we added additional features to this data, like, maybe the opening prices for these days, or the volume of the stock for these days, then perhaps this may help our model learn more about the data and improve its accuracy.

3-Reduce Dropout

Again, this is exactly opposite of a technique we gave for reducing overfitting.
dropout, which we'll cover in more detail at a later time, is a *regularization* technique that randomly ignores a subset of nodes in a given layer. It essentially prevents these dropped out nodes from participating in producing a prediction on the data.
When using dropout, we can specify a percentage of the nodes we want to drop. So if we're using a 50% dropout rate, and we see that our model is underfitting, then we can decrease our amount of dropout by reducing the dropout percentage to something lower than 50 and see what types of metrics we get when we attempt to train again.
These nodes are only dropped out for purposes of training and not during validation. So, if we see that our model is fitting better to our validation data than it is to our training data, then this is a good indicator to reduce the amount of dropout that we're using.

Supervised Learning For Machine Learning

Up to this point in this series, each time we've mentioned the process of training a model or the learning process that the model goes through, we've actually been implicitly talking about supervised learning.

Labeled Data

Supervised learning occurs when the data in our training set is labeled.

Labels are used to supervise or guide the learning process.

Recall from our post on training, validation, and testing sets, we explained that both the training data and validation data are labeled when passed to the model. This is the case for supervised learning.

With supervised learning, each piece of data passed to the model during training is a pair that consists of the input object, or sample, along with the corresponding label or output value.

Essentially, with supervised learning, the model is learning how to create a mapping from given inputs to particular outputs based on what it's learning from the labeled training data.

For example, say we're training a model to classify different types of reptiles based on images of reptiles. Now during training, we pass in an image of a *lizard*.

For example, say we're training a model to classify different types of reptiles based on images of reptiles. Now during training, we pass in an image of a *lizard*.

Since we're doing supervised learning, we'll also be supplying our model with the label for this image, which in this case is simply just *lizard*. we know that the model will then classify the output of this image, and then determine the error for that image by looking at the difference between the value it predicted and the actual label for the image.

Labels Are Numeric

To do this, the labels need to be encoded into something numeric. In this case, the label of *lizard* may be encoded as `0`, whereas the label of *turtle* may be encoded as `1`.

After this, we go through this process of determining the error or loss for all of the data in our training set for as many epochs as we specify. Remember, during this training, the objective of the model is to minimize the loss, so when we deploy our model and use it to predict on data it wasn't trained on, it will be making these predictions based on the labeled data that it did see during training.

If we didn't supply our labels to the model, though, then what's the alternative? Well, as opposed to ***supervised learning,*** we could instead use something called ***unsupervised learning***. We could also use another technique called ***semi-supervised learning***. We'll be covering each of these topics in the future.

Working With Labeled Data In Keras

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
import numpy as np
```

```
model = Sequential([
    Dense(units=16, input_shape=(2,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='sigmoid')
])
```

```
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

```
# weight, height
train_samples = np.array([
    [150, 67],
    [130, 60],
    [200, 65],
    [125, 52],
    [230, 72],
    [181, 70]
])
```

The actual training data is stored in the `train_samples` variable. Here, we have a list of pairs, and each of these pairs is an individual sample, and a sample is the weight and height of a person.

The first element in each pair is the weight measured in *pounds*, and the second element is the height measured in *inches*.
Next, we have our labels stored in this `train_labels` variable. Here, a `0` represents a male, and a `1` represents a female.

```
# 0: male
# 1: female
train_labels = np.array([1, 1, 0, 1, 0, 0])
```

The position of each of these labels corresponds to the positions of each sample in our `train_samples` variable. For example, this first `1` here, which represents a female, is the label for the first element in the `train_samples` array. This second `1` in `train_labels` corresponds to the second sample in `train_samples`, and so on.

```
model.fit(
    x=train_samples,
    y=train_labels,
    batch_size=3,
    epochs=10,
    shuffle=True,
    verbose=2
)
```

Now, when we go to train our model, we call `model.fit()` as we've discussed in previous posts, and the first parameter here specified by `x` is going to be our `train_samples` variable, and the second parameter, specified by `y`, is going to be the corresponding `train_labels`.

Unsupervised Learning In Machine Learning

Unsupervised learning occurs with unlabeled data.

Unlabeled Data

In contrast to supervised learning, unsupervised learning occurs when the data in our training set is *not labeled*.

With unsupervised learning, each piece of data passed to our model during training is solely an unlabeled input object, or sample. There is no corresponding label that's paired with the sample.

Hm... but if the data isn't labeled, then how is the model learning? How is it evaluating itself to understand if it's performing well or not?

Well, first, let's go ahead and touch on the fact that, with unsupervised learning, since the model is unaware of the labels for the training data, there is no way to measure accuracy. Accuracy is not typically a metric that we use to analyze an unsupervised learning process.

Essentially, with unsupervised learning, the model is going to be given an unlabeled dataset, and it's going to attempt to learn some type of *structure* from the data and will extract the useful information or features from this data.

It's going to be learning how to create a mapping from given inputs to particular outputs based on what it's learning about the structure of this data without any labels.

Unsupervised Learning Examples

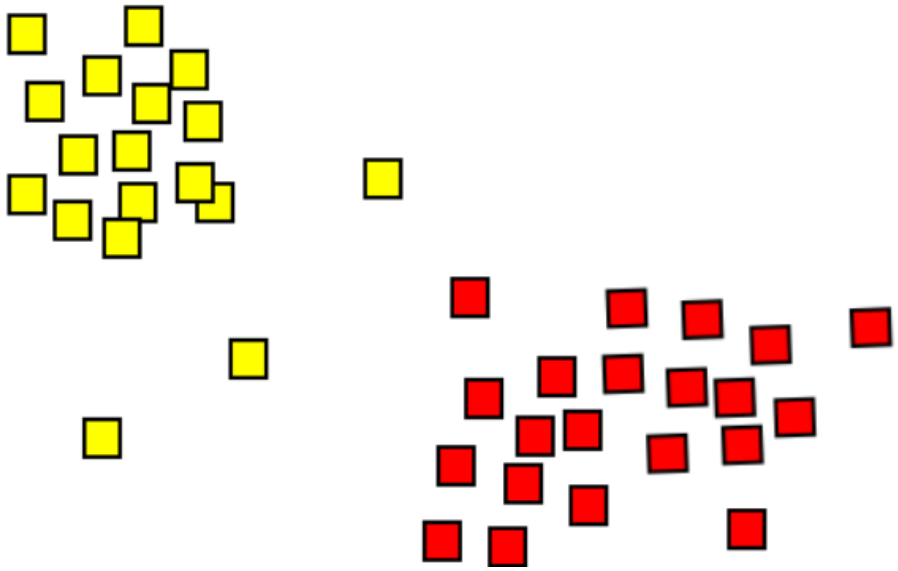
1-Clustering Algorithms

One of the most popular applications of unsupervised learning is through the use of *clustering algorithms*. Sticking with our example from our previous post on supervised learning, let's suppose we have the *height* and *weight* data for a particular age group of *males* and *females*.

This time, we don't have the labels for this data, so any given sample from this data set would just be a pair consisting of one person's height and weight. There is no associated label telling us whether this person was a male or female.

Now, a clustering algorithm could analyze this data and start to learn the structure of it even though it's not labeled. Through learning the structure, it can start to cluster the data into groups.

We could imagine that if we were to plot this height and weight data on a chart, then maybe it would look something like this with weight on the x-axis and height on the y-axis.



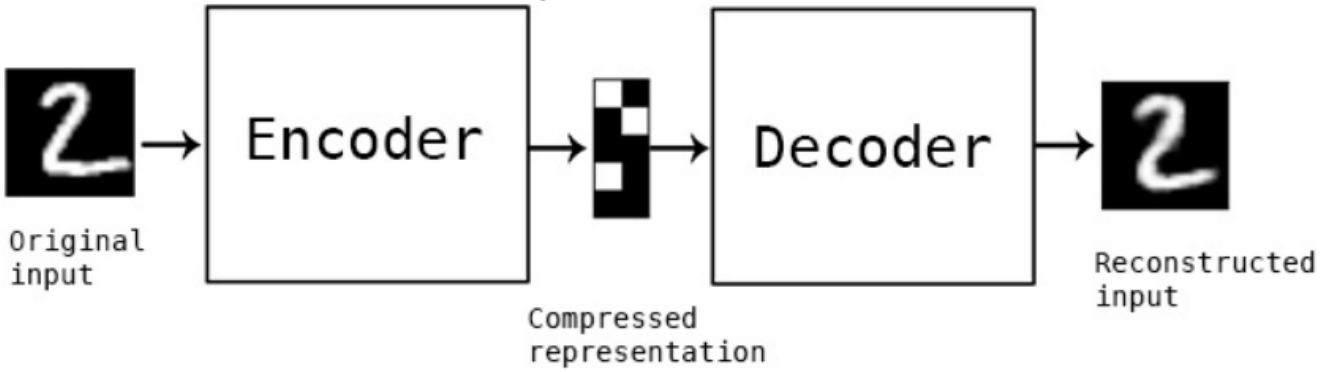
There's nothing explicitly telling us the labels for this data, but we can see that there are two pretty distinct clusters here, and so we could infer that perhaps this clustering is occurring based on whether these individuals are male or female.

One of these clusters may be made up predominantly of females, while the other is predominantly male, so clustering is one area that makes use of unsupervised learning. Let's look at another.

2-Autoencoders

In the most basic terms, an autoencoders is an artificial neural network that takes in input, and then outputs a reconstruction of this input.

Based on everything we've learned so far on neural networks, this seems pretty strange, but let's explain this idea further using an example.



The example we'll use is written about in a blog by François Chollet, the author of Keras

Suppose we have a set of images of handwritten digits, and we want to pass them through an autoencoder. Remember, an autoencoder is just a neural network.

This neural network will take in this image of a digit, and it will then *encode* the image. Then, at the end of the network, it will *decode* the image and output the decoded reconstructed version of the original image.
The goal here is for the reconstructed image to be as close as possible to the original image.

A question we might ask about this process is: How can we even measure how well this autoencoder is doing at reconstructing the original image without visually inspecting it?

Well, we can think of the loss function for this autoencoder as measuring how similar the reconstructed version of the image is to the original version. The more similar the reconstructed image is to the original image, the lower the loss.

Since this is an artificial neural network after all, we'll still be using some variation of SGD during training, and so we'll still have the same objective of minimizing our loss function.

During training, our model is incentivized to make the reconstructed images closer and closer to the originals.

Applications Of Autoencoders

what would be an application for doing this? Why would we just want to reconstruct input?

Well, one application for this could be to denoise images. Once the model has been trained, then it can accept other similar images that may have a lot of noise surrounding them, and it will be able to extract the underlying meaningful features and reconstruct the image without the noise.

Semi-Supervised Learning For Machine Learning

Semi-supervised learning kind of takes a middle ground between supervised learning and unsupervised learning.

Semi-supervised learning uses a combination of supervised and unsupervised learning techniques, and that's because, in a scenario where we'd make use of semi-supervised learning, we would have a combination of both *labeled* and *unlabeled* data.

Large Unlabeled Dataset

Suppose we have access to a large unlabeled dataset that we'd like to train a model on and that manually labeling all of this data ourselves is just not practical.

Well, we could go through and manually label some portion of this large data set ourselves and use that portion to train our model.

This is fine. In fact, this is how a lot of data used for neural networks becomes labeled. However, if we have access to large amounts of data, and we've only labeled some small portion of this data, then what a waste it would be to just leave all the other unlabeled data on the table.

I mean, after all, we know the more data we have to train a model, the better and more robust our model will be. What can we do to make use of the remaining unlabeled data in our data set?

Well, one thing we can do is implement a technique that falls under the category of semi-supervised learning called *pseudo-labeling*.

Pseudo-Labeling

This is how pseudo-labeling works. As just mentioned, we've already labeled some portion of our data set. Now, we're going to use this labeled data as the training set for our model. We're then going to train our model, just as we would with any other labeled data set.

Just through the regular training process, we get our model performing pretty well, and so everything we've done up to this point has been regular old supervised learning in practice.

Now here's where the unsupervised learning piece comes into play. After we've trained our model on the labeled portion of the data set, we then use our model to predict on the remaining unlabeled portion of data, and we then take these predictions and label each piece of unlabeled data with the individual outputs that were predicted for them.

This process of labeling the unlabeled data with the output that was predicted by our neural network is the very essence of pseudo-labeling.

After labeling the unlabeled data through this pseudo-labeling process, we train our model on the full dataset, which is now comprised of both the data that was actually truly labeled along with the data that was pseudo labeled.

As we can imagine, sometimes the cost of acquiring or generating a fully labeled data set is just too high, or the pure act of generating all the labels itself is just not feasible.

Through this process, we can see how this approach makes use of both supervised learning, with the labeled data, and unsupervised learning, with the unlabeled data, which together give us the practice of semi-supervised learning.

One-Hot Encodings For Machine Learning

Labels are not what you think

We know that when we're training a neural network via supervised learning, we pass labeled input to our model, and the model gives us a predicted output.

If our model is an image classifier, for example, we may be passing labeled images of animals as input. When we do this, the model is usually not interpreting these labels as words, like *dog* or *cat*. Additionally, the output that our model gives us in regards to its predictions aren't typically words like *dog* or *cat* either. Instead, most of the time our labels become encoded, so they can take on the form of an integer or of a vector of integers.

Hot And Cold Values

One type of *encoding* that is widely used for encoding categorical data with numerical values is called *one-hot encoding*.

One-hot encodings transform our categorical labels into vectors of `0`s and `1`s. The length of these vectors is the number of classes or categories that our model is expected to classify.

Value	Interpretation
0	Cold
1	Hot

Vectors Of 0s And 1s

If we were classifying whether images were either of a dog or of a cat, then our one-hot encoded vectors that corresponded to these classes would each be of length `2` reflecting the two categories.

If we added another category, like lizard, so that we could then classify whether images were of dogs, cats, or lizards, then our corresponding one-hot encoded vectors would each be of length `3` since we now have three categories.

Alright, so we know the labels are transformed or *encoded* into vectors. We know that each of these vectors has a length that is equal to the number of output categories, and we briefly mentioned that the vectors contain `0`s and `1`s. Let's go into further detail on this last piece.

One-Hot Encodings For Multiple Categories

Let's stick with the example of classifying images as being either of a *cat*, *dog*, or *lizard*. With each of the corresponding vectors for these categories being of length 3, we can think of each index or each element within the vector corresponding to one of the three categories.

Let's say for this example that the cat label corresponds to the first element, dog corresponds to the second element, and lizard corresponds to the third element.

With each of these categories having their own *place* in the corresponding vectors, we can now discuss the intuition behind the name *one-hot*.

With each one-hot encoded vector, every element will be a zero EXCEPT for the element that corresponds to the actual category of the given input. This element will be a *hot one*.

One of the indices of the vector is hot!

Sticking with our same example, recall we said that a cat corresponded to the first element, dog to the second, and lizard to the third, so the corresponding one-hot encoded vectors for each of these categories would look like this.

Label	Index-0	Index-1	Index-2
Cat	1	0	0
Dog	0	1	0
Lizard	0	0	1

For cat, we see that the first element is a one and the next two elements are zeros. This is because each element within the vector is a zero except for the element that corresponds to the actual category, and we said that the cat category corresponded to the first element.

One Vector For Each Category

Similarly, for dog, we see that the second element is a one, while the first and third elements are zeros. Lastly, for lizard, the third element is a one, while the first and second elements are zeros.

We can see that each time the model receives input that is a cat, it's not interpreting the label as the word *cat*, but instead is interpreting the label as this vector [1,0,0].

For images labeled as dog, the model is interpreting the dog label as the vector [0,1,0], and for images labeled as lizard, the model is interpreting the label as the vector [0,0,1]

Label	Vector
Cat	[1,0,0]
Dog	[0,1,0]
Lizard	[0,0,1]

Just for clarity purposes, say we add another category, llama, to the mix. Now, we have four categories total, and so this will cause each one-hot encoded vector corresponding to each of these categories to be of length 4 now.

The vectors will now look like this.

Label	Vector
Cat	[1,0,0,0]
Dog	[0,1,0,0]
Lizard	[0,0,1,0]
Llama	[0,0,0,1]

Finally, the new one-hot encoded vector for the llama category is all zeros except for the fourth element, which is a one, since the fourth element corresponds to the llama category.

Note that we just arbitrarily said that cat corresponded to the first element, dog to the second, lizard to the third, and llama to the fourth, but this could very well be in a different order. This just depends on how the underlying code or library is doing the one-hot encoding.

Convolutional Neural Networks

A convolutional neural network, also known as a *CNN* or *ConvNet*, is an artificial neural network that has so far been most popularly used for analyzing images for computer vision tasks. A convolutional neural network, also known as a *CNN* or *ConvNet*, is an artificial neural network that has so far been most popularly used for analyzing images for computer vision tasks.

What Is A CNN?

Most generally, we can think of a CNN as an artificial neural network that has some type of specialization for being able to pick out or detect patterns. This pattern detection is what makes CNNs so useful for image analysis.

If a CNN is just an artificial neural network, though, then what differentiates it from a standard multilayer perceptron or MLP?

CNNs have hidden layers called *convolutional* layers, and these layers are what make a CNN, well... a CNN!

CNNs can, and usually do, have other, non-convolutional layers as well, but the basis of a CNN is the convolutional layers.

Convolutional Layers

Just like any other layer, a convolutional layer receives input, transforms the input in some way, and then outputs the transformed input to the next layer. The inputs to convolutional layers are called input channels, and the outputs are called ***output channels***.

With a convolutional layer, the transformation that occurs is called a *convolution operation*. This is the term that's used by the deep learning community anyway. Mathematically, the convolution operations performed by convolutional layers are actually called ***cross-correlations.***

let's look at a high level idea of what convolutional layers are doing.

Filters And Convolution Operations

As mentioned earlier, convolutional neural networks are able to detect patterns in images.

With each convolutional layer, we need to specify the number of *filters* the layer should have. These filters are actually what detect the patterns.

Patterns

Let's expand on precisely what we mean When we say that the filters are able to *detect patterns*. Think about how much may be going on in any single image. Multiple edges, shapes, textures, objects, etc. These are what we mean by *patterns*.

- edges
- shapes
- textures
- curves
- objects
- colors

One type of pattern that a filter can detect in an image is edges, so this filter would be called an *edge detector*.

Aside from edges, some filters may detect corners. Some may detect circles. Others, squares. Now these simple, and kind of geometric, filters are what we'd see at the start of a convolutional neural network.

The deeper the network goes, the more sophisticated the filters become. In later layers, rather than edges and simple shapes, our filters may be able to detect specific objects like eyes, ears, hair or fur, feathers, scales, and beaks.

In even deeper layers, the filters are able to detect even more sophisticated objects like full dogs, cats, lizards, and birds.

To understand what's actually happening here with these convolutional layers and their respective filters, let's look at an example.

Filters (Pattern Detectors)

Suppose we have a convolutional neural network that is accepting images of handwritten digits (like from the MNIST data set) and our network is classifying them into their respective categories of whether the image is of a 1, 2, 3, etc.

Let's now assume that the first hidden layer in our model is a convolutional layer. As mentioned earlier, when adding a convolutional layer to a model, we also have to specify how many filters we want the layer to have.

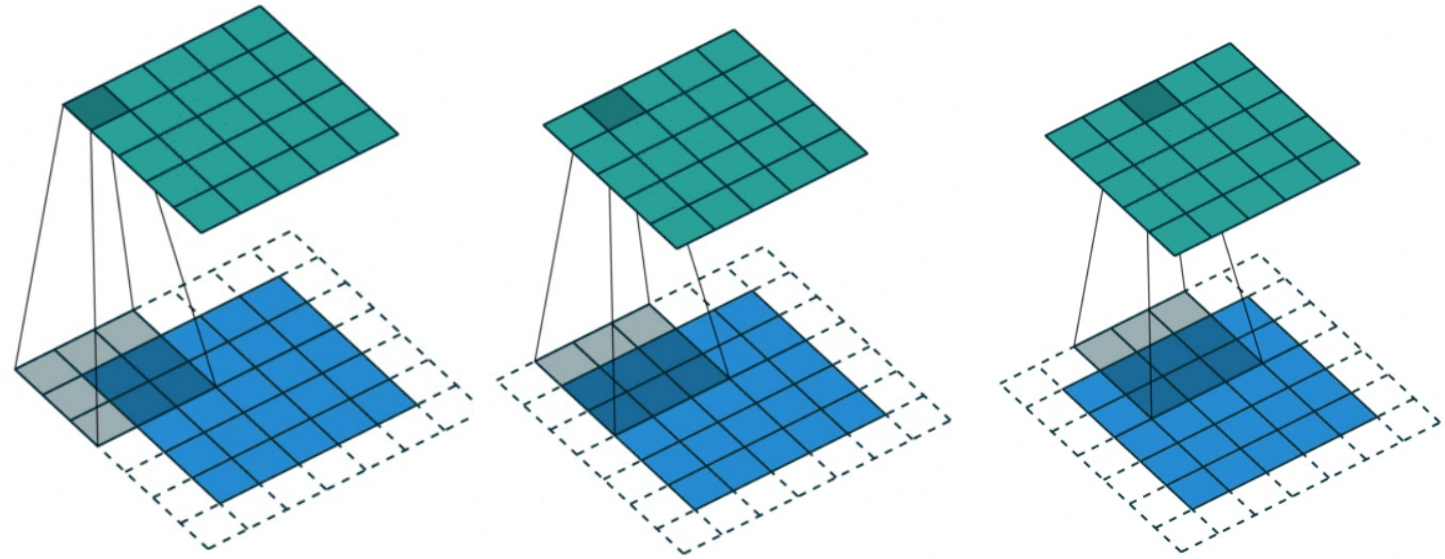
The number of filters determines the number of output channels.

A filter can technically just be thought of as a relatively small matrix (tensor), for which, we decide the number of rows and columns this matrix has, and the values within this matrix are initialized with random numbers.

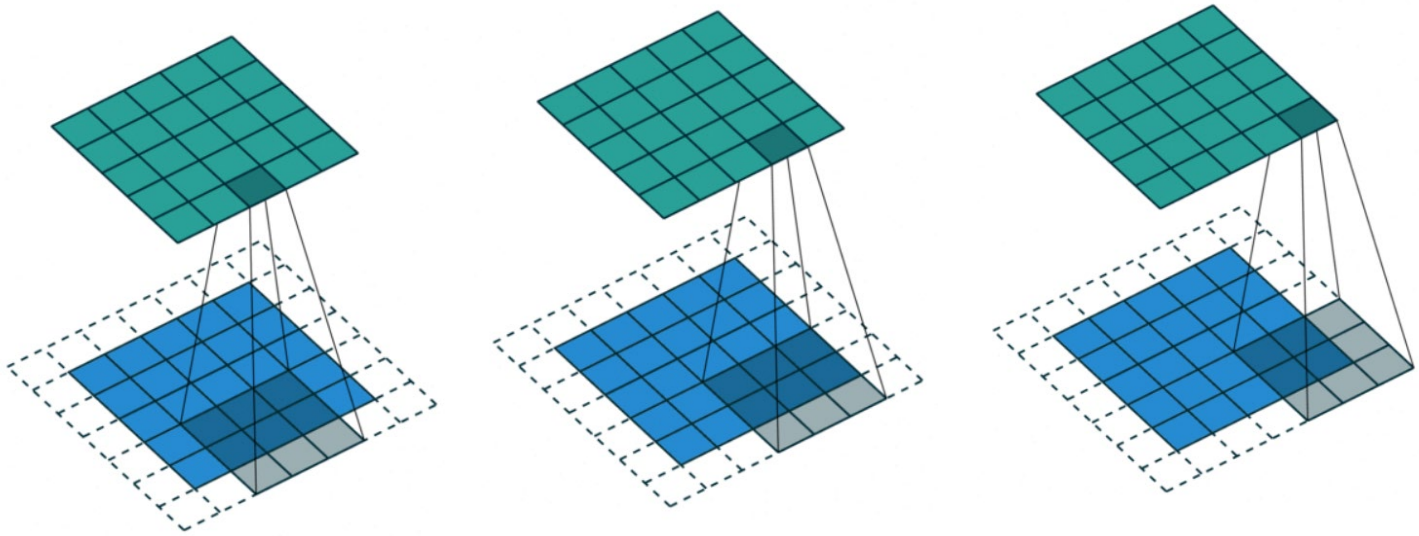
For this first convolutional layer of ours, we're going to specify that we want the layer to contain one filter of size 3 x 3.

Convolutional Layer

Let's look at an example animation of the convolution operation:



To the end



This animation showcases the convolution process without numbers. We have an input channel in blue on the bottom. A convolutional filter shaded on the bottom that is sliding across the input channel, and a green output channel:

- Blue (bottom) - Input channel
- Shaded (on top of blue) - 3×3 convolutional filter
- Green (top) - Output channel

For each position on the blue input channel, the 3×3 filter does a computation that maps the shaded part of the blue input channel to the corresponding shaded part of the green output channel.

This convolutional layer receives an input channel, and the filter will slide over each 3×3 set of pixels of the input itself until it's slid over every 3×3 block of pixels from the entire image.

Convolution Operation

This sliding is referred to as *convolving*, so really, we should say that this filter is going to *convolve* across each 3×3 block of pixels from the input.

The blue input channel is a matrix representation of an image from the MNIST dataset. The values in this matrix are the individual pixels from the image. These images are grayscale images, and so we only have a single input channel.

- Grayscale images have a single color channel
- RGB images have three color channels

This input will be passed to a convolutional layer. As just discussed, we've specified the first convolutional layer to only have one filter, and this filter is going to convolve across each 3×3 block of pixels from the input. When the filter lands on its first 3×3 block of pixels, the dot product of the filter itself with the 3×3 block of pixels from the input will be computed and stored. This will occur for each 3×3 block of pixels that the filter convolves.

For example, we take the dot product of the filter with the first 3×3 block of pixels, and then that result is stored in the output channel. Then, the filter slides to the next 3×3 block, computes the dot product, and stores the value as the next pixel in the output channel. After this filter has convolved the entire input, we'll be left with a new representation of our input, which is now stored in the output channel. This output channel is called a feature map.

This green output channel becomes the input channel to the next layer as input, and then this process that we just went through with the filter will happen to this new output channel with the next layer's filters.

This was just a very simple illustration, but as mentioned earlier, we can think of these filters as pattern detectors.

A Note About The Usage Of The "Dot Product"

We are loosely using the term "dot product" to discuss the operation done above, however, technically what we're actually doing is summing the element-wise products of each pair of elements in the two matrices.

For example, suppose we have two $3 \text{ x } 3$ matrices A and B as follows

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \qquad B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Then, we sum the pairwise products like this: $a_{1,1}b_{1,1} + a_{1,2}b_{1,2} + \cdots + a_{3,3}b_{3,3}$

So, technically this operation is the *summation of the element-wise products*. Even so, you may still encounter the term "dot product" used loosely to refer to this operation. The reason for this is due to the fact that the operation shown here is an *inner product*, which is a generalization of the *dot product*. For this reason, you may also see this operation referred to as the *Frobenius inner product* or the *summation of the Hadamard product* as well.

Input And Output Channels

Suppose that this grayscale image (single color channel) of a seven from the MNIST data set is our input:

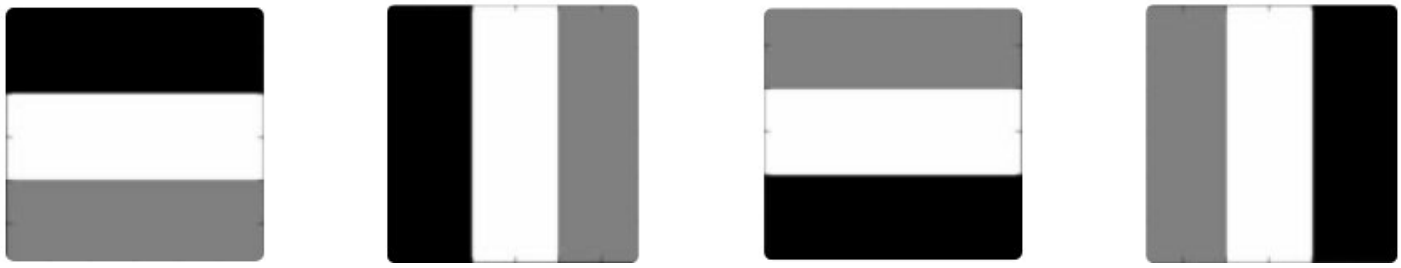
Let's suppose that we have four `3 x 3` filters for our first convolutional layer, and these filters are filled with the values you see below. These values can be represented visually by having `-1`s correspond to black, `1`s correspond to white, and `0`s correspond to grey.

-1	-1	-1
1	1	1
0	0	0

-1	1	0
-1	1	0
-1	1	0

0	0	0
1	1	1
-1	-1	-1

0	1	-1
0	1	-1
0	1	-1



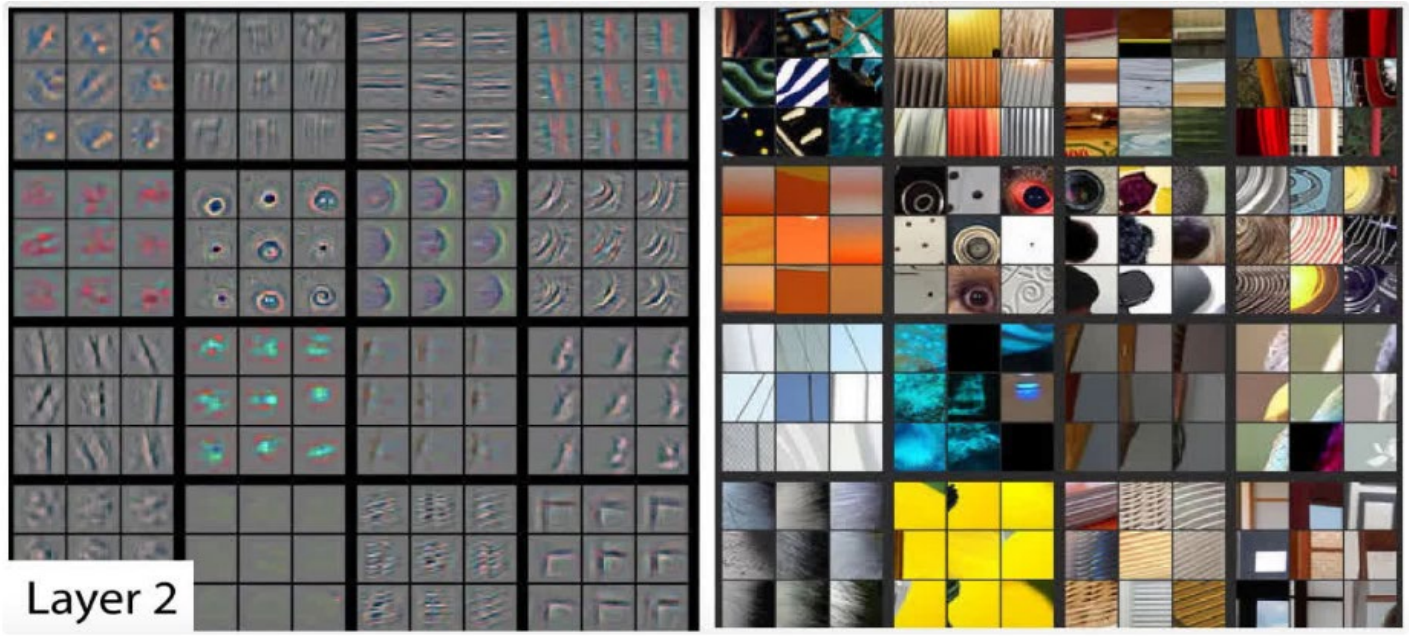
(Convolutional Layer with 4 filters)

If we convolve our original image of a seven with each of these four filters individually, this is what the output would look like for each filter:

(Output channels from the Convolutional Layer)

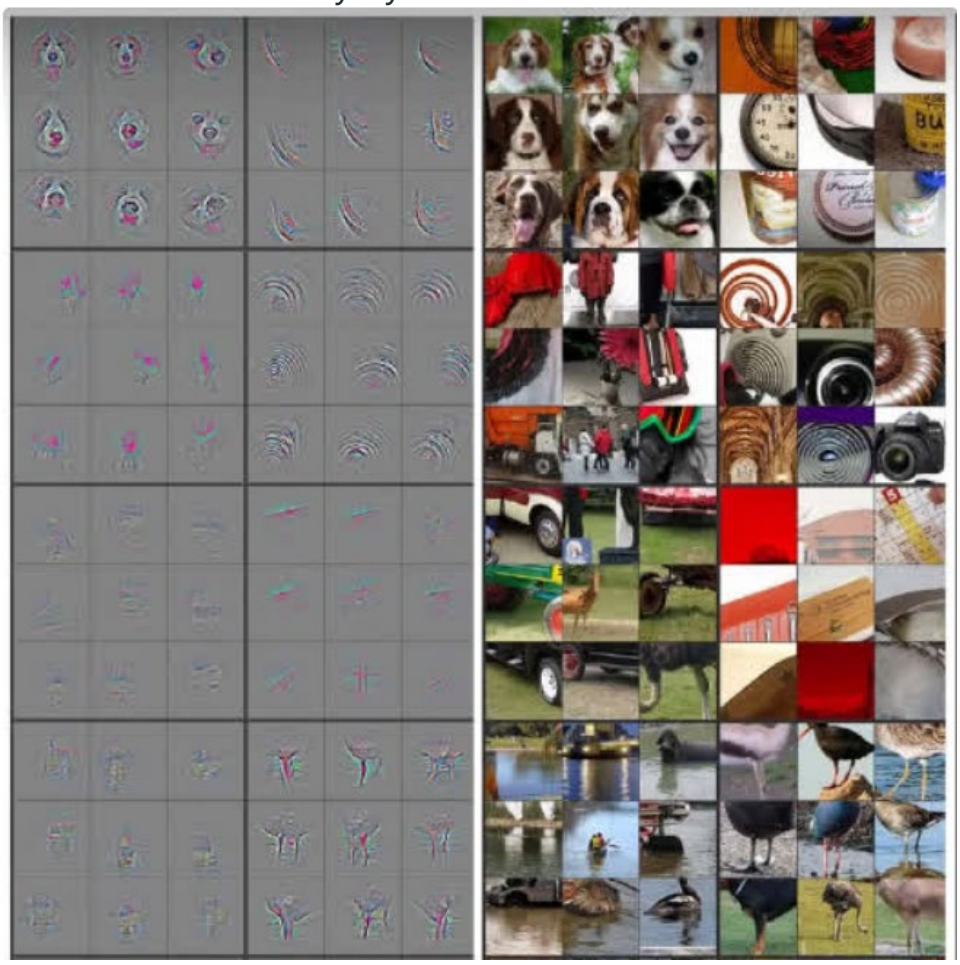
We can see that all four of these filters are detecting edges. In the output channels, the brightest pixels can be interpreted as what the filter has detected. In the first one, we can see detects top horizontal edges of the seven, and that's indicated by the brightest pixels (white).

The second detects left vertical edges, again being displayed with the brightest pixels. The third detects bottom horizontal edges, and the fourth detects right vertical edges. These filters, as we mentioned before, are really basic and just detect edges. These are filters we may see towards the start of a convolutional neural network. More complex filters would be located deeper in the network and would gradually be able to detect more sophisticated patterns like the ones shown here:



We can see the shapes that the filters on the left detected from the images on the right. We can see circles, curves and corners. As we go further into our layers, the filters are able to detect much more complex patterns like dog faces or bird legs shown here:

(about the image downward)The amazing thing is that the pattern detectors are derived automatically by the network. The filter values start out with random values,



and the values change as the network learns during training. The pattern detecting capability of the filters emerges automatically. ***Pattern detectors emerge as the network learns.***

In the past, computer vision experts would develop filters (pattern detectors) manually. One example of this is the Sobel filter, an edge detector. However, with deep learning, we can learn these filters automatically using neural networks!

Visualizing Convolutional Filters

how to visualize the convolutional filters from a convolutional neural network so that we can better understand how these networks learn.

To do this, we're going to build on some ideas and concepts that we covered in our previous chapter on convolutional neural networks.

In that chapter, we discussed how each convolutional layer has some set number of filters and that these filters are what actually detect patterns in the given input. We explained technically how this works, and then at the end of the chapter, we looked at some filters from a CNN and observed what they were able to detect from real world images.

Keras And The Code

We're going to be using Keras, a neural network API, to visualize the filters of the convolutional layers from the VGG16 network. We've talked about VGG16 previously in the Keras series, but in short, VGG16 is a CNN that won the ImageNet competition in 2014. This is a competition where teams build algorithms to compete on visual recognition tasks.

Most of the code we'll be using to visualize the filters comes from the blog, How convolutional neural networks see the world, by the creator of Keras, François Chollet.

Rather than going over the code line-by-line, we're going to instead give a high-level overview of what the code is doing, and then we'll get to the visualization piece. This github link contains the original code from the blog so you can check it out or run it yourself.

The first step is to import the pre-trained VGG16 model.

```
# build the VGG16 network with ImageNet weights
model = vgg16.VGG16(weights='imagenet', include_top=False)
```

Then we define a loss function that has an objective to maximize the activation of a given filter within a given layer. We then calculate gradient ascent with regard to our filter's activation loss.

```
# we build a loss function that maximizes the activation
# of the nth filter of the layer considered
layer_output = layer_dict[layer_name].output
if K.image_data_format() == 'channels_first':
    loss = K.mean(layer_output[:, filter_index, :, :])
else:
    loss = K.mean(layer_output[:, :, :, filter_index])

# we compute the gradient of the input picture wrt this loss
grads = K.gradients(loss, input_img)[0]
```

Note that gradient *ascent* is the same thing as gradient *descent*, except for rather than trying to minimize our loss, we're trying to maximize it.

We can think of the purpose of maximizing our loss here as basically trying to activate the filter as much as possible in order for us to be able to visually inspect what types of patterns the filter is detecting.

We then pass the network a plain gray image with some random noise as input.

```
# we start from a gray image with some random noise
if K.image_data_format() == 'channels_first':
    input_img_data = np.random.random((1, 3, img_width, img_height))
else:
    input_img_data = np.random.random((1, img_width, img_height, 3))
input_img_data = (input_img_data - 0.5) * 20 + 128
```

After we maximize the loss, we're then able to obtain a visual representation of what sort of input maximizes the activation for each filter in each layer.

```
# save the result to disk
save_img('stitched_filters_%dx%d.png' % (n, n), stitched_filters)
```

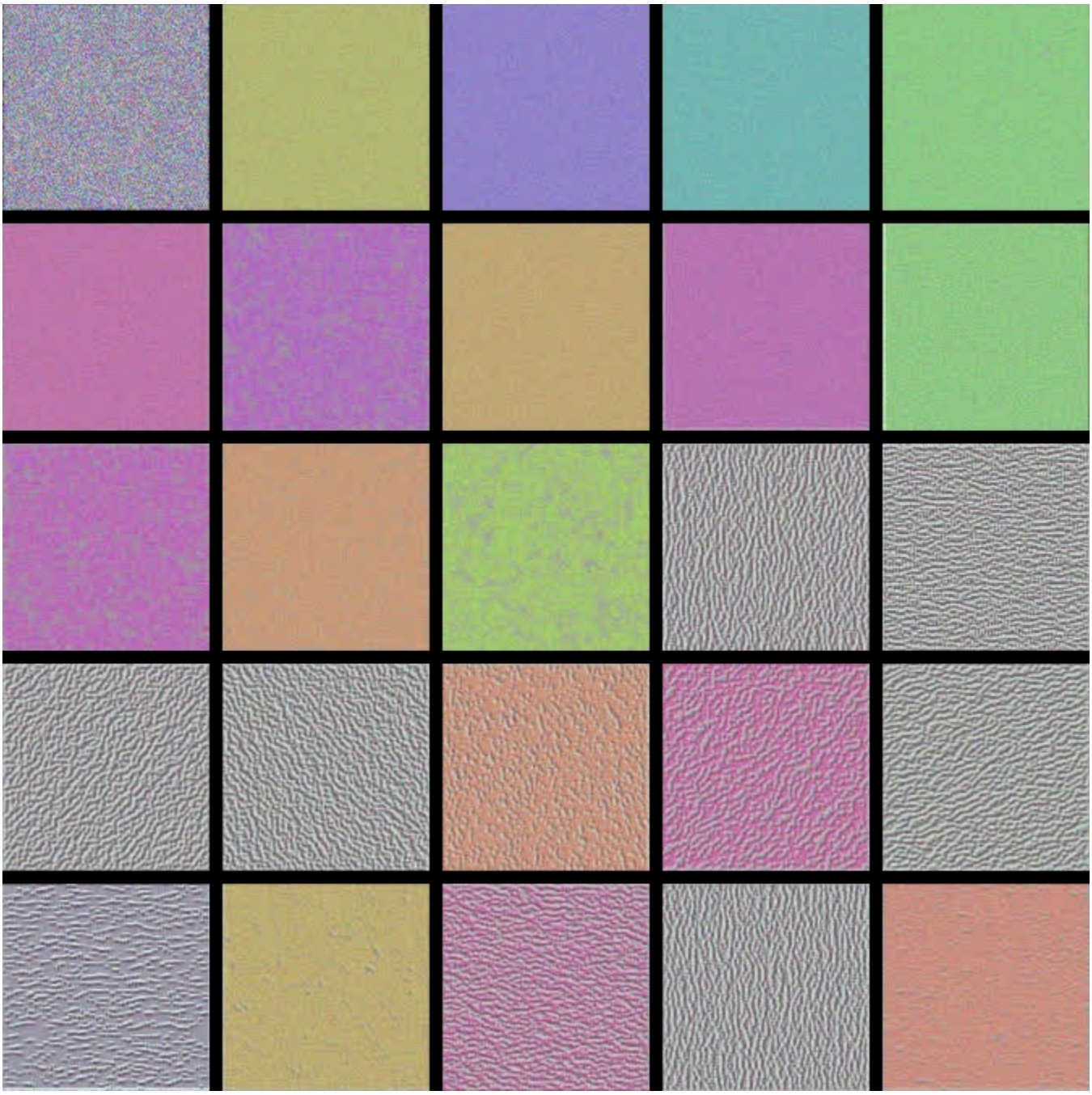
This is generated from the original gray image that we supplied the network.

To run this code, it did take a bit of time running on a CPU. Maybe about an hour to generate all of the visualizations.

That's a summary of what our code is actually doing. Now, let's get to the cool part and step through some of these generated visualizations from each convolutional layer.

Generated CNN Layer Visualizations

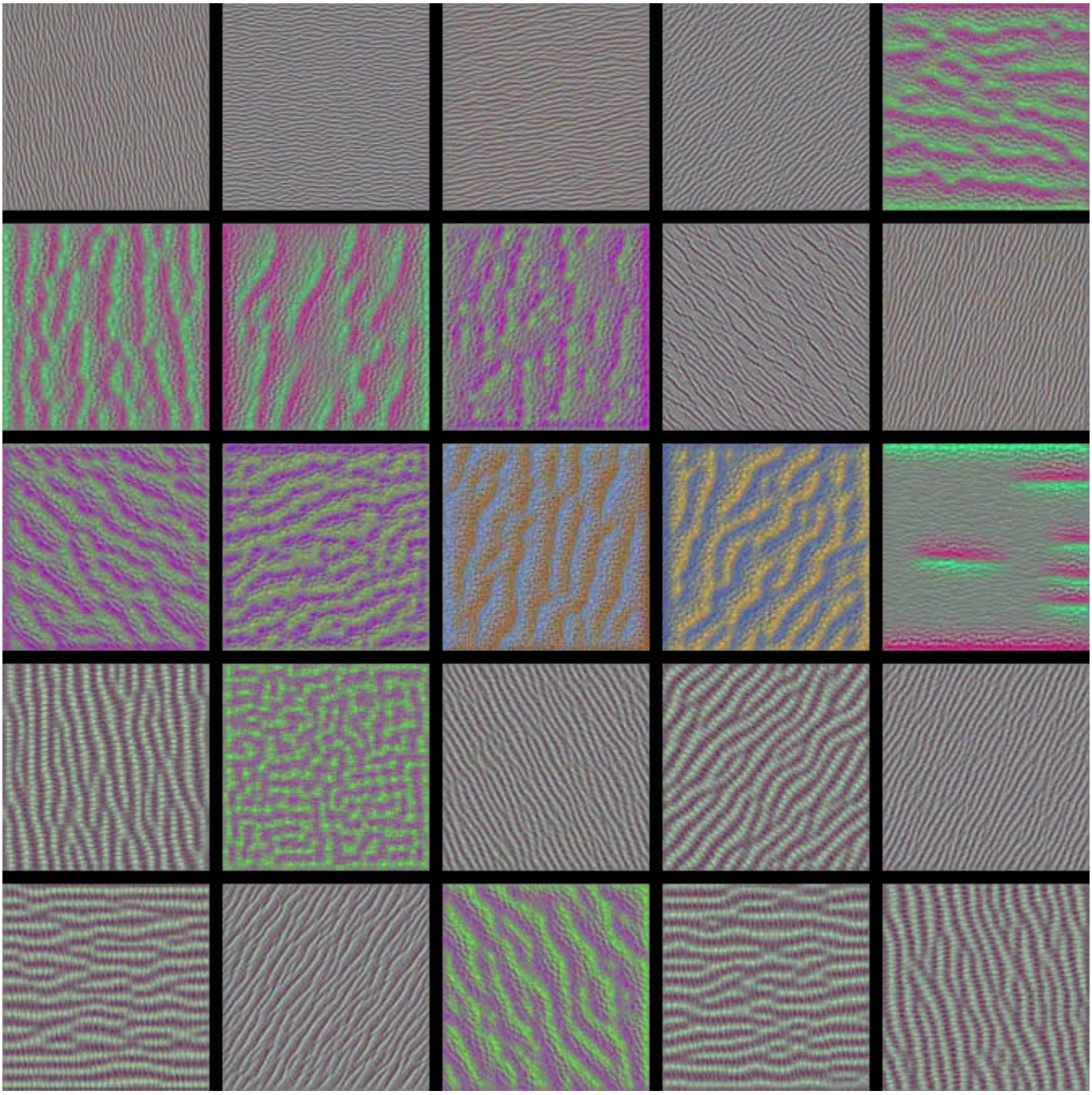
Here, we're looking at 25 filters from the first convolutional layer in the first convolutional block of the network. It looks like most of these have encoded some type of direction or color.



We can see some that indicate the vertical patterns and others that that indicate left and right diagonal patterns.

Let's skip to another deeper convolutional layer. We'll choose the second conv layer from the second conv block.

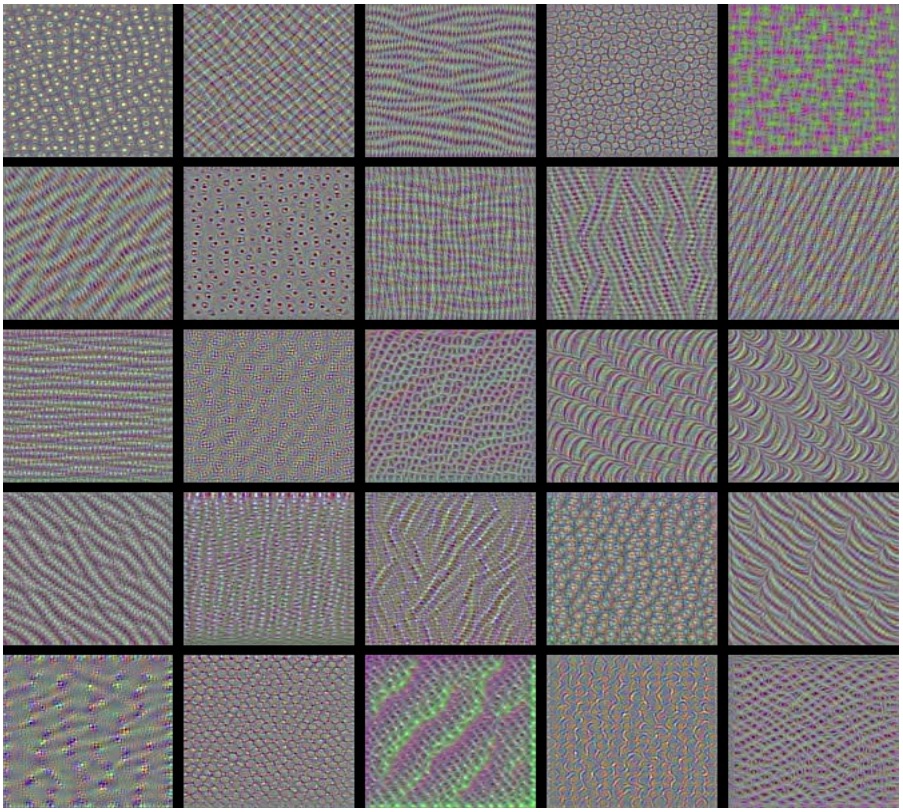
2nd Conv Layer From The 2nd Conv Block



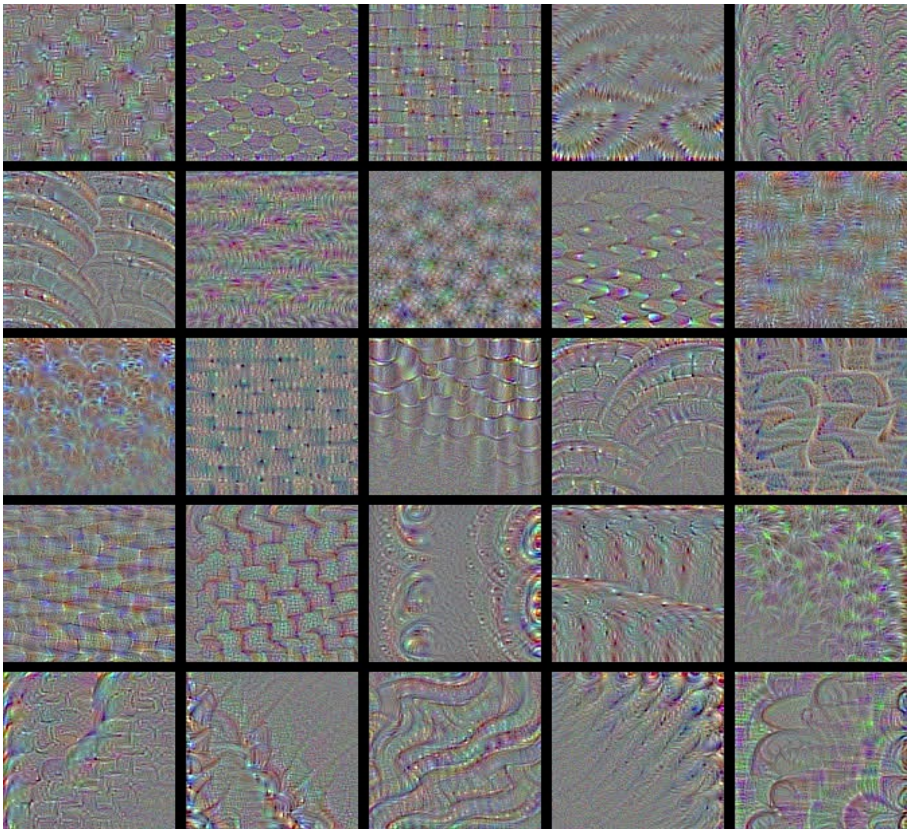
Here, these visualizations have become more complex and a little more interesting in regards to what types of patterns some of the filters have encoded.

Let's check out some even deeper layers.

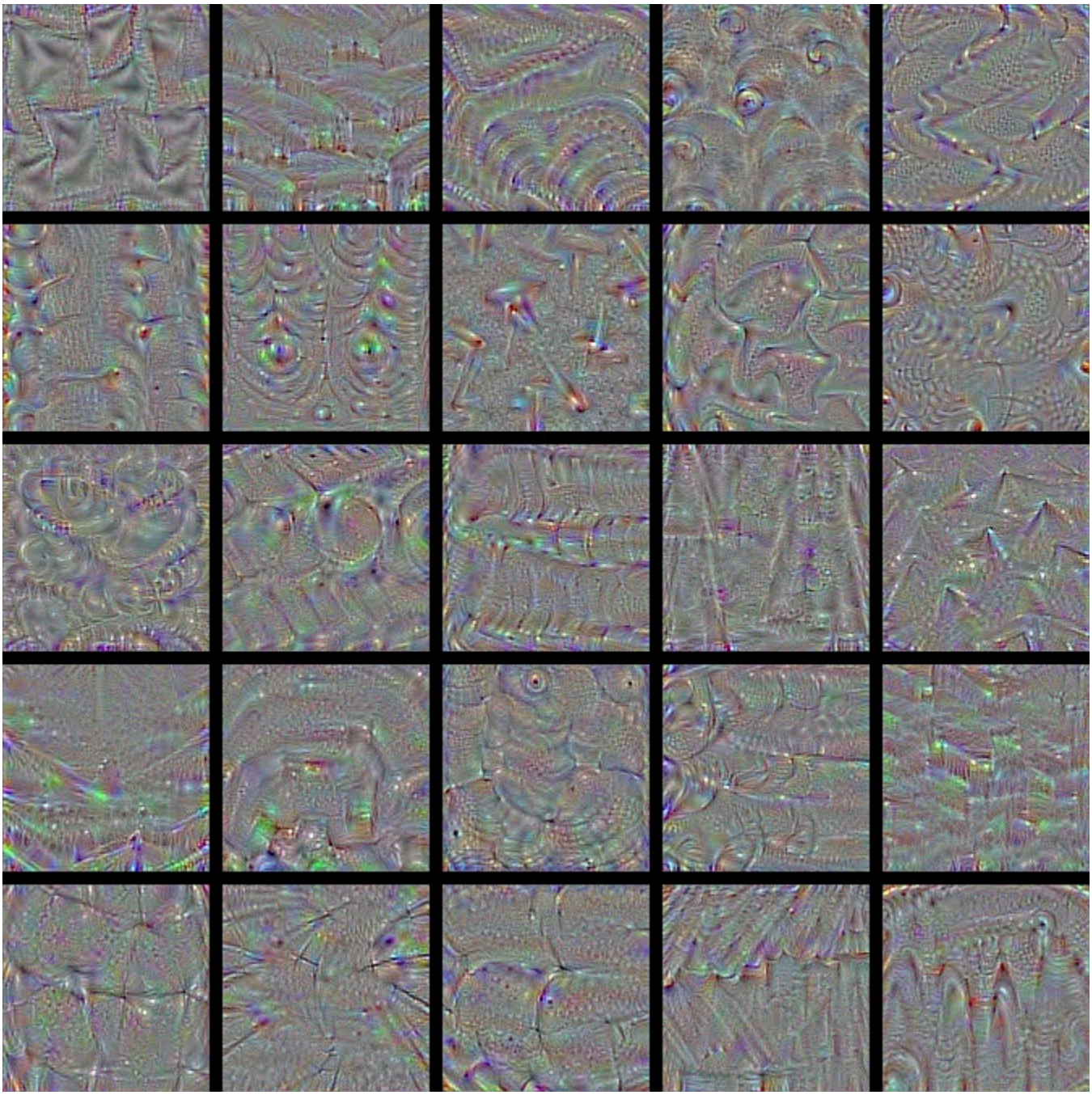
2nd Conv Layer From The 3rd Conv Block



3rd Conv Layer From 4th Conv Block

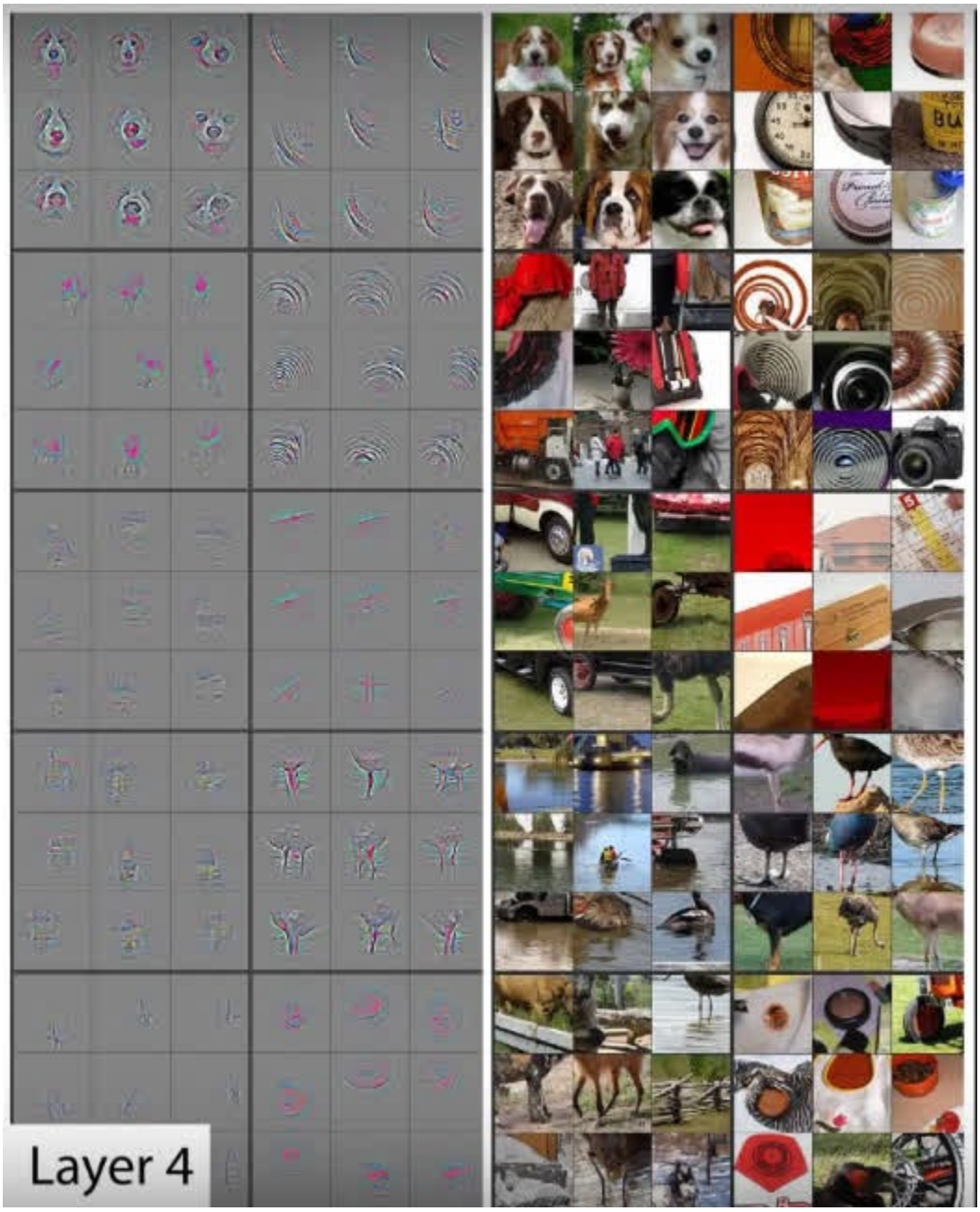


2nd Conv Layer From 5th Conv Block



Notice how with each deeper convolutional layer, we're getting more complex and more interesting visualizations. This whole visualization process was pretty fascinating for me when I first observed it, so I hope you think it's just as cool!

Recall, in the chapter, we showed the visualization of these filters on the left relative to the input images on the right.



Let's focus on the one of dog faces at the top. Recall that none of the filter visualizations we just observed gave us anything that looked remotely like an actual real world object. Instead, we just saw those cool patterns.Why is this? Why didn't we see things like dog faces? Well, recall, what we were previously observing was visual representation of what sort of input would maximize the activation for any given filter. Here, what we're looking at is the patterns that a given filter was able to detect on specific image input for which the filter was highly activated. I just wanted to touch on the differences between those two illustrations.

Zero Padding In Convolutional Neural Networks

We're going to start out by explaining the motivation for ***zero padding***, and then we'll get into the details about what zero padding actually is. We'll then talk about the types of issues we may run into if we don't use zero padding, and then we'll see how we can implement zero padding in code using Keras.

We're going to be building on some of the ideas that we discussed in our post on convolutional neural networks, so if you haven't seen that yet, go ahead and check it out, and then come back to to this one once you've finished up there.

Convolutions Reduce Channel Dimensions

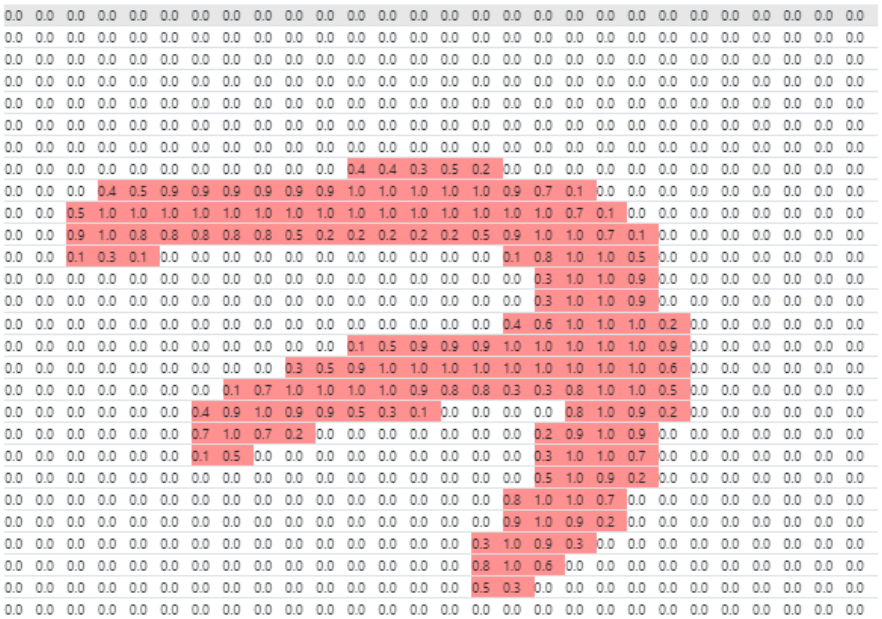
We've seen in our post on CNNs that each convolutional layer has some number of filters that we define, and we also define the dimension of these filters as well. We also showed how these filters convolve image input.

When a filter convolves a given input channel, it gives us an output channel. This output channel is a matrix of pixels with the values that were computed during the convolutions that occurred on the input channel.

When this happens, ***the dimensions of our image are reduced.***

Let's check this out using the same image of a seven that we used in our previous post on CNNs. Recall, we have a 28×28 matrix of the pixel values from an image of a 7 from the MNIST data set. We'll use a 3×3 filter. This gives us the following the items:

A 28×28 single input channel (grayscale image):



3 x 3 Filter

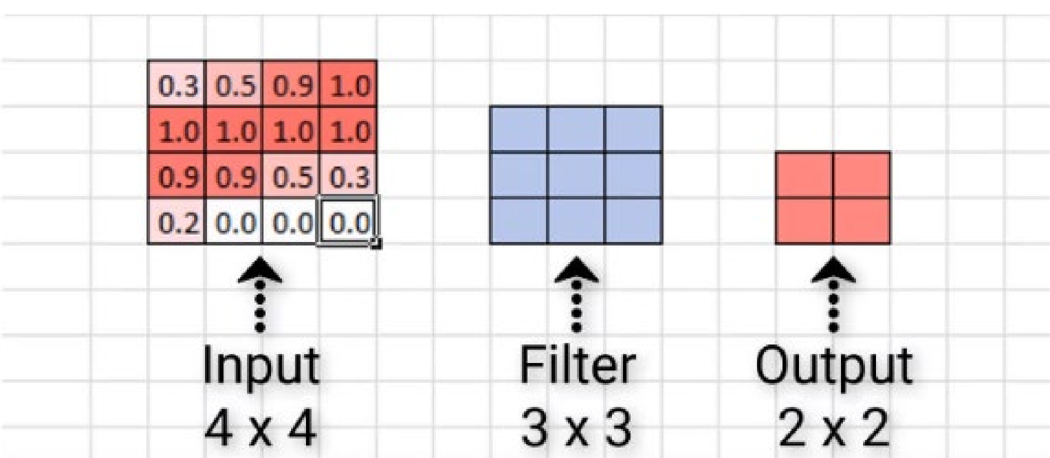
0.98	0.28	0.94
0.71	0.05	0.56
0.6	0.33	0.85

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.4	0.6	0.7	0.5	0.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.3	0.6	1.2	1.4	1.6	1.6	1.6	1.6	1.9	1.9	2.2	2.3	2.1	2.0	1.7	0.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.5	1.2	1.8	2.6	2.7	3.0	3.0	3.0	3.0	3.4	3.5	3.8	4.0	3.7	3.6	3.2	2.3	1.5	0.5	0.1	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.1	3.2	4.2	4.4	4.7	4.7	4.5	4.2	4.0	3.8	3.9	3.9	4.1	4.5	4.7	4.1	3.1	1.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.0	3.1	3.6	3.3	3.2	3.2	3.1	2.9	2.7	2.5	2.5	2.5	2.7	3.0	3.9	4.4	4.1	2.9	1.4	0.3	0.0	0.0	0.0	0.0	0.0
0.9	1.4	2.1	2.2	1.8	1.7	1.7	1.5	1.1	0.8	0.5	0.5	0.5	0.8	1.3	2.4	3.7	4.5	4.0	2.4	1.0	0.0	0.0	0.0	0.0	0.0
0.1	0.3	0.3	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	1.3	2.8	4.2	4.7	2.8	1.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.2	2.9	3.9	5.1	3.1	2.2	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.4	1.0	1.3	1.6	1.9	2.4	3.7	4.4	5.2	3.8	2.5	0.7	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.5	1.1	1.7	2.3	2.7	3.0	3.4	3.7	4.6	4.9	5.2	4.1	2.5	1.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	0.7	1.3	1.9	2.6	3.2	4.0	4.4	4.8	4.4	4.2	4.5	4.8	5.2	4.5	2.7	1.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.4	1.0	1.8	2.6	3.3	3.8	3.9	3.8	3.6	3.4	3.0	2.9	3.6	4.1	5.0	3.8	2.5	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.8	1.7	3.0	3.5	3.7	3.3	3.0	2.5	2.2	1.9	1.3	1.3	2.4	3.3	4.8	3.4	2.3	0.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	2.0	2.7	3.2	2.6	1.8	1.3	0.7	0.4	0.1	0.0	0.4	2.2	3.3	4.6	3.0	2.0	0.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.7	1.4	1.6	1.7	0.7	0.2	0.0	0.0	0.0	0.0	0.0	0.8	2.5	3.7	4.2	2.6	1.5	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	0.5	0.2	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.7	1.7	3.3	4.0	3.6	2.2	0.8	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.3	2.3	4.0	3.9	2.8	1.6	0.2	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.3	3.1	4.5	3.4	2.0	0.8	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	2.6	3.4	3.8	2.5	1.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	2.0	2.8	2.4	1.5	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.3	2.0	1.3	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We can see that the output is actually not the same size as the original input. The output size is 26 x 26. Our original input channel was 28 x 28, and now we have an output channel that has shrank in size to 26 x 26 after convolving the image. Why is that?

With our 28 x 28 image, our 3 x 3 filter can only fit into 26 x 26 possible positions, not all 28 x 28. Given this, we get the resulting 26 x 26 output. This is due to what happens when we convolve the edges of our image.

For ease of visualizing this, let's look at a smaller scale example. Here we have an input of size 4 x 4 and then a 3 x 3 filter. Let's look at how many times we can convolve our input with this filter, and what the resulting output size will be.



This means that when this 3×3 filter finishes convolving this 4×4 input, it will give us an output of size 2×2.

We see that the resulting output is 2×2, while our input was 4×4, and so again, just like in our larger example with the image of a seven, we see that our output is indeed smaller than our input in terms of dimensions.

We can know ahead of time by how much our dimensions are going to shrink. In general, if our image is of size $n \times n$, and we convolve it with an $f \times f$ filter, then the size of the resulting output is

(n–f+1) x (n–f+1).

Let's see if this holds up with our example here.

Our input was size 4×4, so 4 would be our n, and our filter was 3×3, so 3 would be our f. Substituting these values in our formula, we have:

$$(n-f+1)=(4-3)+1=2$$

Indeed, this gives us a 2×2 output channel, which is exactly what we saw a

moment ago. This holds up for the example with the larger input of the seven as well, so check that for yourself to confirm that the formula does indeed give us the same result of an output of size 26×26 that we saw when we visually inspected it.

Issues With Reducing The Dimensions

Consider the resulting output of the image of a seven again. It doesn't really appear to be a big deal that this output is a little smaller than the input, right?

We didn't lose that much data or anything because most of the important pieces of this input are kind of situated in the middle. But we can imagine that this would be a bigger deal if we did have meaningful data around the edges of the image.

Additionally, we only convolved this image with one filter. What happens as this original input passes through the network and gets convolved by more filters as it moves deeper and deeper?

Well, what's going to happen is that the resulting output is going to continue to become smaller and smaller. This is a problem.

If we start out with a 4×4 image, for example, then just after a convolutional layer or two, the resulting output may become almost meaningless with how small it becomes. Another issue is that we're losing valuable data by completely throwing away the information around the edges of the input.

What can we do here? Queue the super hero music because this is where zero padding comes into play.

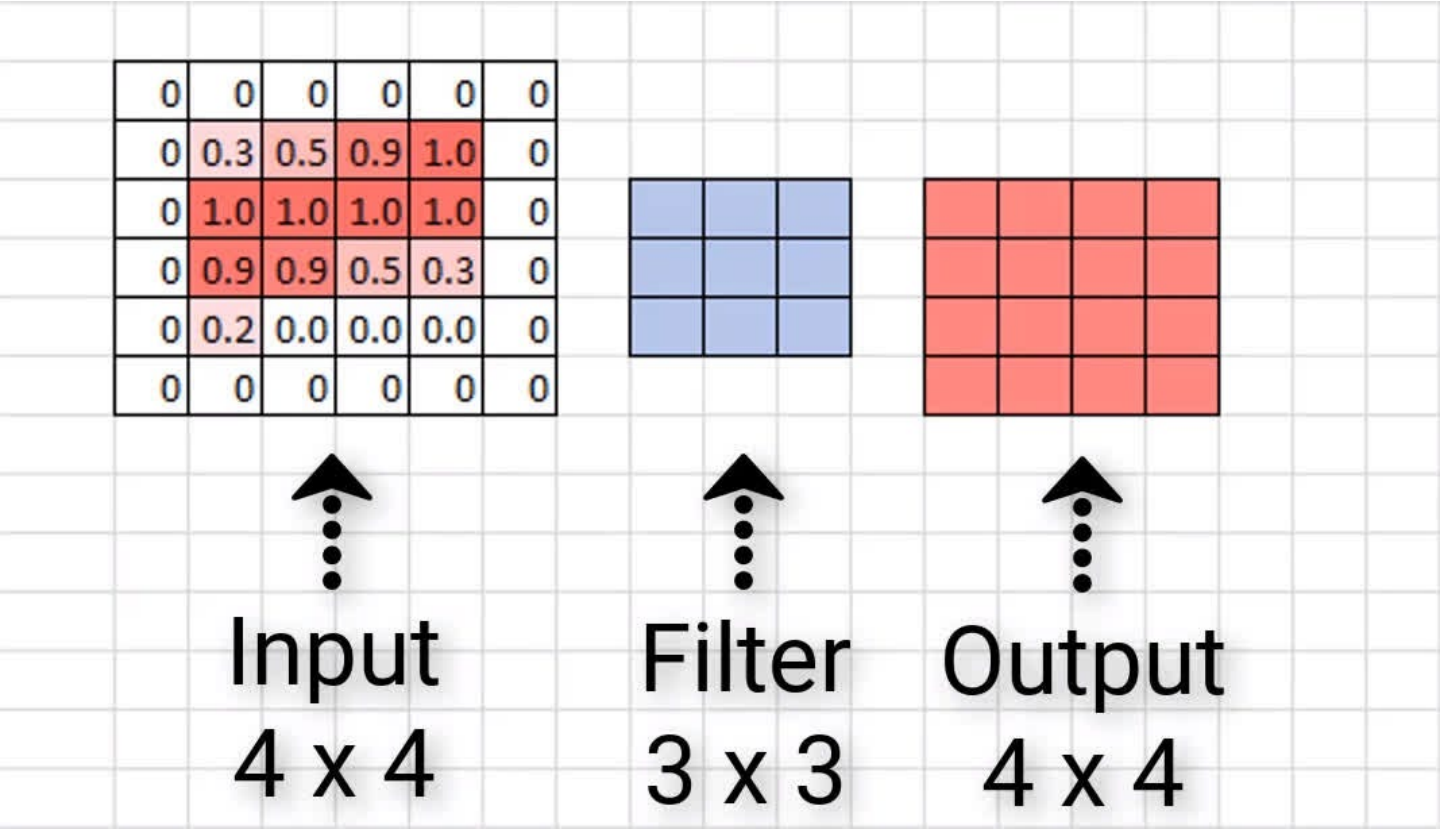
Zero Padding To The Rescue

Zero padding is a technique that allows us to preserve the original input size. This is something that we specify on a per-convolutional layer basis. With each convolutional layer, just as we define how many filters to have and the size of the filters, we can also specify whether or not to use padding.

What Is Zero Padding?

We now know what issues zero padding combats against, but what actually is it?

Zero padding occurs when we add a border of pixels all with value zero around the edges of the input images. This adds kind of a *padding* of zeros around the outside of the image, hence the name *zero padding*. Going back to our small example from earlier, if we pad our input with a border of zero valued pixels, let's see what the resulting output size will be after convolving our input.



We see that our output size is indeed 4×4, maintaining the original input size. Now, sometimes we may need to add more than a border that's only a single pixel thick. Sometimes we may need to add something like a double border or triple border of zeros to maintain the original size of the input. This is just going to depend on the size of the input and the size of the filters.

The good thing is that most neural network APIs figure the size of the border out for us. All we have to do is just specify whether or not we actually want to use padding in our convolutional layers.

Valid And Same Padding

There are two categories of padding. One is referred to by the name *valid*. This just means *no padding*. If we specify valid padding, that means our convolutional layer is not going to pad at all, and our input size won't be maintained.

The other type of padding is called *same*. This means that we want to pad the original input before we convolve it so that the output size is the *same size* as the input size.

Padding Type	Description	Impact
Valid	No padding	Dimensions reduce
Same	Zeros around the edges	Dimensions stay the same

Working With Code In Keras

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import *
```

Now, we'll create a completely arbitrary CNN.

```
model_valid = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='valid'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='valid'),
    Conv2D(128, kernel_size=(7,7), activation='relu', padding='valid'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

It has a dense layer, then `3` convolutional layers followed by a dense output layer.

We've specified that the input size of the images that are coming into this CNN is `20 x 20`, and our first convolutional layer has a filter size of `3 x 3`, which is specified in Keras with the `kernel_size` parameter. Then, the second conv layer specifies size `5 x 5`, and the third, `7 x 7`.

With this model, we're specifying the parameter called `padding` for each convolutional layer. We're setting this parameter equal to the string `'valid'`. Remember from earlier that, valid padding means no padding.

This is actually the default for convolutional layers in Keras, so if we don't specify this parameter, it's going to default to valid padding. Since we're using valid padding here, we expect the dimension of our output from each of these convolutional layers to decrease.

Let's check. Here is the summary of this model.

```
> model_valid.summary()


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 18, 18, 32)        4640
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         401536
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 16386
=================================================================
Total params: 473,890
Trainable params: 473,890
Non-trainable params: 0
```

We can see the output shape of each layer in the second column. The first two integers specify the dimension of the output in height and width. Starting with our first layer, we see our output size is the original size of our input, $20 \text{ x } 20$.

Once we get to the output of our first convolutional layer, the dimensions decrease to $18 \text{ x } 18$, and again at the next layer, it decreases to $14 \text{ x } 14$, and finally, at the last convolutional layer, it decreases to $8 \text{ x } 8$.

So, we start with $20 \text{ x } 20$ and end up with $8 \text{ x } 8$ when it's all done and over with.

On the contrary, now, we can create a second model.

```
model_same = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='same'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='same'),
    Conv2D(128, kernel_size=(7,7), activation='relu', padding='same'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

This one is an exact replica of the first, except that we've specified `same` padding for each of the convolutional layers. Recall from earlier that same padding means we want to pad the original input before we convolve it so that the output size is the same size as the input size.

Let's look at the summary of this model.

```
> model_same.summary()


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 20, 20, 64)        51264
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 20, 20, 128)       401536
_________________________________________________________________
flatten_3 (Flatten)          (None, 51200)             0
_________________________________________________________________
dense_7 (Dense)              (None, 2)                 102402
=================================================================
Total params: 559,906
Trainable params: 559,906
Non-trainable params: 0
_________________________________________________________________
```

We can see again that we're starting out with our input size of $20 \text{ x } 20$, and if we look at the output shape for each of the convolutional layers, we see that the layers do indeed maintain the original input size now.

This is why we call this type of padding same padding. Same padding keeps the input dimensions the same.

Max Pooling In Convolutional Neural Networks

Max pooling is a type of operation that is typically added to CNNs following individual convolutional layers.

When added to a model, max pooling reduces the dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer.

Let's go ahead and check out a couple of examples to see what exactly max pooling is doing operation-wise, and then we'll come back to discuss why we may want to use max pooling.

Example Using A Sample From The MNIST Dataset

We've seen in our chapter on CNNs that each convolutional layer has some number of filters that we define with a specified dimension and that these filters convolve our image input channels.

When a filter convolves a given input, it then gives us an output. This output is a matrix of pixels with the values that were computed during the convolutions that occurred on our image. We call these *output channels*.

We're going to be using the same image of a seven that we used in our previous post on CNNs. Recall, we have a matrix of the pixel values from an image of a `7` from the MNIST data set.

We used a `3 x 3` filter to produce the output channel below:

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.4	0.6	0.7	0.5	0.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.3	0.6	1.2	1.4	1.6	1.6	1.6	1.6	1.9	1.9	2.2	2.3	2.1	2.0	1.7	0.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.5	1.2	1.8	2.6	2.7	3.0	3.0	3.0	3.0	3.4	3.5	3.8	4.0	3.7	3.6	3.2	2.3	1.5	0.5	0.1	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.1	3.2	4.2	4.4	4.7	4.7	4.5	4.2	4.0	3.8	3.9	3.9	4.1	4.5	4.7	4.1	3.1	1.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.0	3.1	3.6	3.3	3.2	3.2	3.1	2.9	2.7	2.5	2.5	2.5	2.7	3.0	3.9	4.4	4.1	2.9	1.4	0.3	0.0	0.0	0.0	0.0	0.0
0.9	1.4	2.1	2.2	1.8	1.7	1.7	1.5	1.1	0.8	0.5	0.5	0.5	0.8	1.3	2.4	3.7	4.5	4.0	2.4	1.0	0.0	0.0	0.0	0.0	0.0
0.1	0.3	0.3	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	1.3	2.8	4.2	4.7	2.8	1.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.2	2.9	3.9	5.1	3.1	2.2	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.4	1.0	1.3	1.6	1.9	2.4	3.7	4.4	5.2	3.8	2.5	0.7	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.5	1.1	1.7	2.3	2.7	3.0	3.4	3.7	4.6	4.9	5.2	4.1	2.5	1.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	0.7	1.3	1.9	2.6	3.2	4.0	4.4	4.8	4.4	4.2	4.5	4.8	5.2	4.5	2.7	1.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.4	1.0	1.8	2.6	3.3	3.8	3.9	3.8	3.6	3.4	3.0	2.9	3.6	4.1	5.0	3.8	2.5	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.8	1.7	3.0	3.5	3.7	3.3	3.0	2.5	2.2	1.9	1.3	1.3	2.4	3.3	4.8	3.4	2.3	0.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	2.0	2.7	3.2	2.6	1.8	1.3	0.7	0.4	0.1	0.0	0.4	2.2	3.3	4.6	3.0	2.0	0.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.7	1.4	1.6	1.7	0.7	0.2	0.0	0.0	0.0	0.0	0.0	0.8	2.5	3.7	4.2	2.6	1.5	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	0.5	0.2	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.7	1.7	3.3	4.0	3.6	2.2	0.8	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.3	2.3	4.0	3.9	2.8	1.6	0.2	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.3	3.1	4.5	3.4	2.0	0.8	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	2.6	3.4	3.8	2.5	1.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	2.0	2.8	2.4	1.5	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.3	2.0	1.3	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

26 x 26 output channel

As mentioned earlier, max pooling is added after a convolutional layer. This is the output from the convolution operation and is the input to the max pooling operation.

After the max pooling operation, we have the following output channel:

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.3	0.6	0.7	0.4	0.0	0.0	0.0	0.0	0.0
1.2	2.6	3.0	3.0	3.4	3.8	4.0	3.6	2.3	0.5	0.0	0.0	0.0
2.1	4.2	4.7	4.7	4.2	3.9	4.1	4.7	4.4	2.9	0.3	0.0	0.0
1.4	2.2	1.8	1.7	1.1	0.5	0.8	2.4	4.5	4.7	1.6	0.0	0.0
0.0	0.0	0.0	0.0	0.1	1.0	1.6	2.4	4.4	5.2	2.5	0.0	0.0
0.0	0.0	0.1	1.3	2.6	4.0	4.8	4.4	4.9	5.2	2.7	0.0	0.0
0.0	0.0	1.7	3.5	3.8	3.9	3.6	3.0	4.1	5.0	2.5	0.0	0.0
0.0	0.0	2.0	3.2	2.6	1.3	0.4	0.8	3.7	4.6	2.0	0.0	0.0
0.0	0.0	0.5	0.5	0.0	0.0	0.0	2.3	4.0	3.6	0.8	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.9	3.4	4.5	2.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.2	2.8	2.4	0.3	0.0	0.0	0.0

13 x 13 output channel

Max pooling works like this. We define some $n \times n$ region as a corresponding filter for the max pooling operation. We're going to use 2×2 in this example.
We define a stride, which determines how many pixels we want our filter to move as it slides across the image.

Stride determines how many units the filter slides.

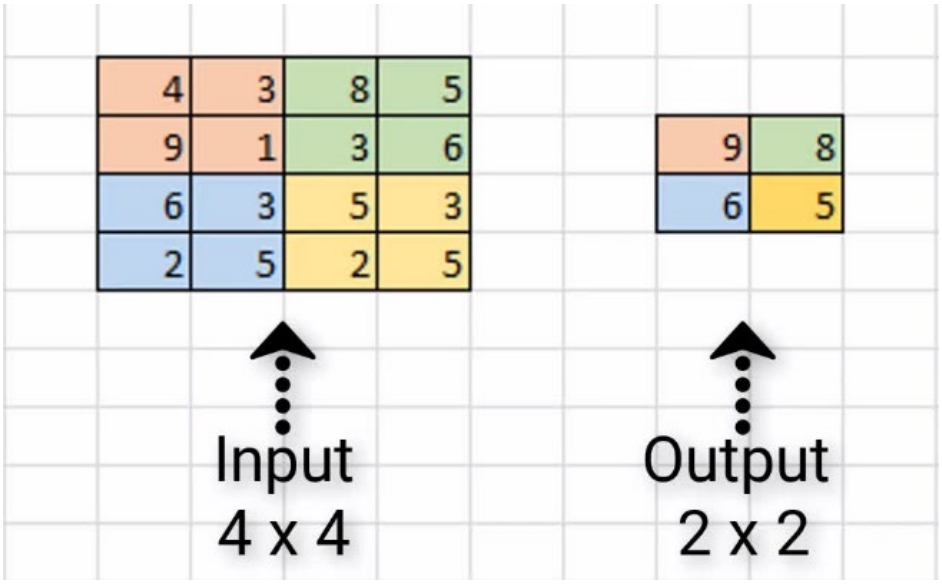
On the convolutional output, and we take the first 2×2 region and calculate the max value from each value in the 2×2 block. This value is stored in the output channel, which makes up the full output from this max pooling operation.

We move over by the number of pixels that we defined our stride size to be. We're using 2 here, so we just slide over by 2, then do the same thing. We calculate the max value in the next 2×2 block, store it in the output, and then, go on our way sliding over by 2 again.

Once we reach the edge over on the far right, we then move down by 2 (because that's our stride size), and then we do the same exact thing of calculating the max value for the 2×2 blocks in this row.
We can think of these 2×2 blocks as *pools* of numbers, and since we're taking the max value from each pool, we can see where the name *max pooling* came from.
This process is carried out for the entire image, and when we're finished, we get the new representation of the image, the output channel.
In this example, our convolution operation output is 26×26 in size. After performing max pooling, we can see the dimension of this image was reduced by a factor of 2 and is now 13×13.

Just to make sure we fully understand this operation, we're going to quickly look at a scaled down example that may be more simple to visualize.

Scaled Down Example



We have some sample input of size 4×4, and we're assuming that we have a 2×2 filter size with a stride of 2 to do max pooling on this input channel.

Our first 2×2 region is in orange, and we can see the max value of this region is 9, and so we store that over in the output channel.

Next, we slide over by 2 pixels, and we see the max value in the green region is 8. As a result, we store the value over in the output channel.

Since we've reached the edge, we now move back over to the far left, and go down by 2 pixels. Here, the max value in the blue region is 6, and we store that here in our output channel.

Finally, we move to the right by 2, and see the max value of the yellow region is 5. We store this value in our output channel.

This completes the process of max pooling on this sample 4×4 input channel, and the resulting output channel is this 2×2 block. As a result, we can see that our input dimensions were again reduced by a factor of two.

Alright, we know what max pooling is and how it works, so let's discuss why would we want to add this to our network?

Why Use Max Pooling?

There are a couple of reasons why adding max pooling to our network may be helpful.

Reducing Computational Load

Since max pooling is reducing the resolution of the given output of a convolutional layer, the network will be looking at larger areas of the image at a time going forward, which reduces the amount of parameters in the network and consequently reduces computational load.

Reducing Overfitting

Additionally, max pooling may also help to reduce overfitting. The intuition for why max pooling works is that, for a particular image, our network will be looking to extract some particular features.

Maybe, it's trying to identify numbers from the MNIST dataset, and so it's looking for edges, and curves, and circles, and such. From the output of the convolutional layer, we can think of the higher valued pixels as being the ones that are the most activated.

With max pooling, as we're going over each region from the convolutional output, we're able to pick out the most activated pixels and preserve these high values going forward while discarding the lower valued pixels that are not as activated.

Just to mention quickly before going forward, there are other types of pooling that follow the exact same process we've just gone through, except for that it does some other operation on the regions rather than finding the max value.

Average Pooling

For example, average pooling is another type of pooling, and that's where you take the average value from each region rather than the max.

Currently max pooling is used vastly more than average pooling, but I did just want to mention that point.

Alright, now let's jump over to Keras and see how this is done in code.

Working With Code In Keras

We'll start with some imports:

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import *
from keras.layers.pooling import *
```

Here, we have a completely arbitrary CNN.

```
model_valid = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='same'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

It has an input layer that accepts input of `20 x 20 x 3` dimensions, then a dense layer followed by a convolutional layer followed by a max pooling layer, and then one more convolutional layer, which is finally followed by an output layer.

Following the first convolutional layer, we specify max pooling. Since the convolutional layers are `2d` here, We're using the `MaxPooling2D` layer from Keras, but Keras also has `1d` and `3d` max pooling layers as well.

The first parameter we're specifying is the `pool_size`. This is the size of what we were calling a filter before, and in our example, we used a `2 x 2` filter.

The next parameter is `strides`. Again, in our earlier examples, we used `2` as well, so that's what we've specified here. The last parameter that we have specified is the `padding` parameter.

we discussed how *valid padding* means to use *no padding*, that's what we've specified here, and actually I don't think it's a common practice at all to use padding on max pooling layers.

But while we're on the subject of padding, I wanted to point something else out, which is that for the two convolutional layers, we've specified same padding so that the input is padded such that the output of the convolutional layers will be the same size as the input.

If we go ahead and look at a summary of our model, we can see that the dimensions from the output of our first layer are $20 \text{ x } 20$, which matches the original input size. The dimensions of the output from our first convolutional layer maintain the same $20 \text{ x } 20$ values because we're using same padding on that layer.

```
> model_valid.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 64)        51264
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 12802
=================================================================
Total params: 68,770
Trainable params: 68,770
Non-trainable params: 0
_________________________________________________________________
```

Once we go down to the max pooling layer, we see the value of the dimensions has been cut in half to become $10 \text{ x } 10$. This is because, as we saw with our earlier examples, a filter of size $2 \text{ x } 2$ along with a stride of 2 for our max pooling layer will reduce the dimensions of our input by a factor of two, so that's exactly what we see here.

Lastly, this max pooling layer is followed by one last convolutional layer that is using same padding, so we can see that the output shape for this last layer maintains the $10 \text{ x } 10$ dimensions from the previous max pooling layer.

Backpropagation In Neural Networks

we're going to discuss *backpropagation* and what its role is in the training process of a neural network.

Stochastic Gradient Descent (SGD) Review

We'll start out by first going over a quick recap of some of the points about stochastic gradient descent we learned in those posts. Then, we're going to talk about where backpropagation comes into the picture, and we'll spend the majority of our time discussing the intuition behind what backpropagation is actually doing.

In the previous posts we referenced, we discussed how, during training, stochastic gradient descent, or SGD, works to minimize the loss function by updating the weights with each epoch.

We mentioned how this updating occurs by calculating the gradient, or taking the derivative, of the loss function with respect to the weights in the model, but we didn't really elaborate on this point.

That's what we're going to discuss now. This act of calculating the gradients in order to update the weights actually occurs through a process called *backpropagation*.

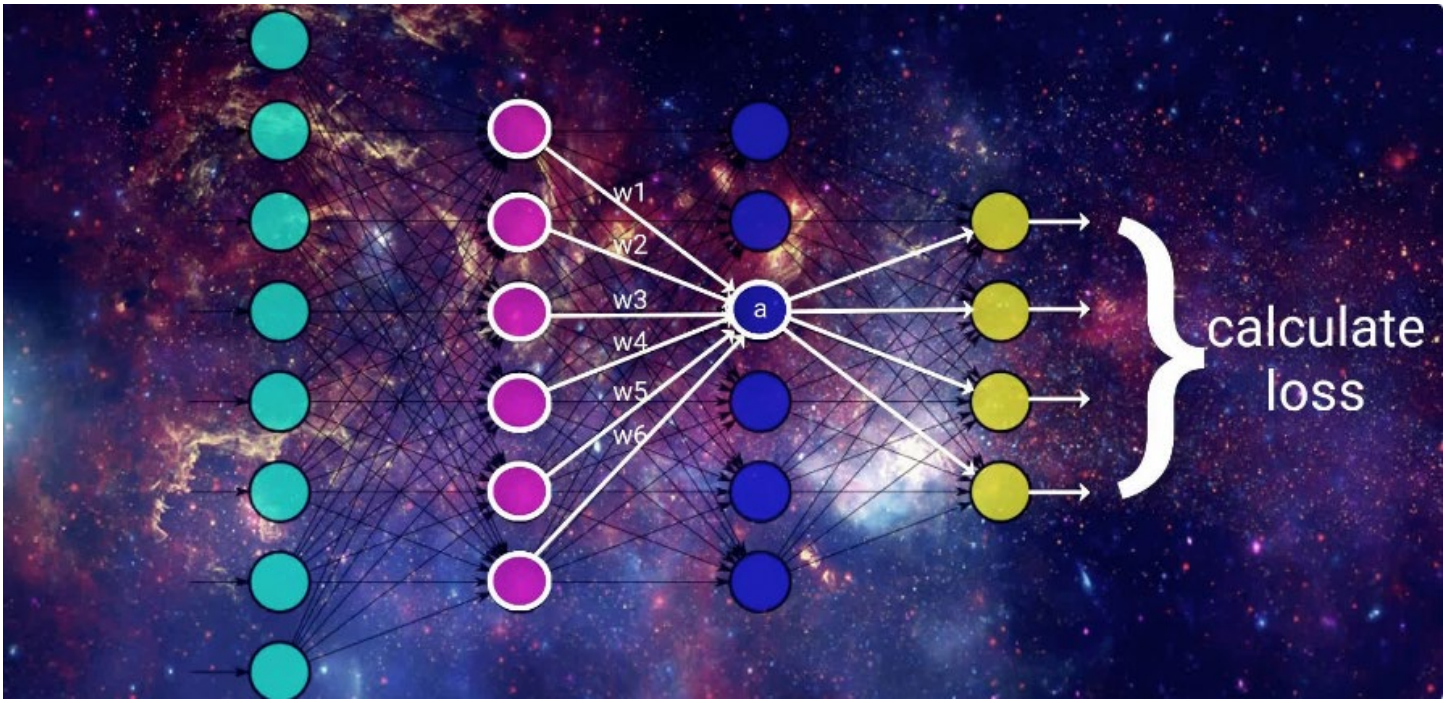
Forward Propagation

We have a sample arbitrary network here with two hidden layers. For simplicity, going forward with our explanation, we're going to be dealing with a single sample of input being supplied to our model, rather than a batch of input.

Now, as a quick refresher on the training process, remember that whenever we pass data to the model, we've seen that this data propagates forward through the network until it reaches the output layer.

Recall that each node in our model receives its input from the previous layer, and that this input is a weighted sum of the weights at each of the connections multiplied by the previous layer's output.

We pass this weighted sum to an activation function, and the result from this activation function is the output for a particular node and is then passed as part of the input for the nodes in the next layer. This happens for each layer in the network until we reach the output layer, and this process is called *forward propagation*.



Once we reach the output layer, we obtain the resulting output from the model for the given input. If we're working to classify images of animals, for example, then each of the output nodes would correspond to a different type of animal, and the output node with the highest activation would be the output that the model thinks is the best match for the corresponding input.

Calculating The Loss

Given the output results, we then calculate the loss on this result. The way the loss is calculated is going to depend on the particular loss function we're using, but for simplicity, let's just think of it for now as being how far off the model is on classifying the given input.
We can think of it as the difference between what the model predicted for a given input and what the given input actually is. We have a post on the loss function if you want to check that out further.

Alright, then, we've discussed how gradient descent's objective is to minimize this loss function. This is done by taking the derivative, that is, *the gradient*, of the loss function with respect to the weights in the model.

This is where backpropagation comes in. $\frac{d\,(loss)}{d\,(weight)}$

Backpropagation is the tool that gradient descent uses to calculate the gradient of the loss function.

As we mentioned, the process of moving the data forward through the network is called forward propagation. Given that the process we're about to cover is called backpropagation, you'd be correct in if you're thinking that this means we're somehow going to be working backwards through the network.

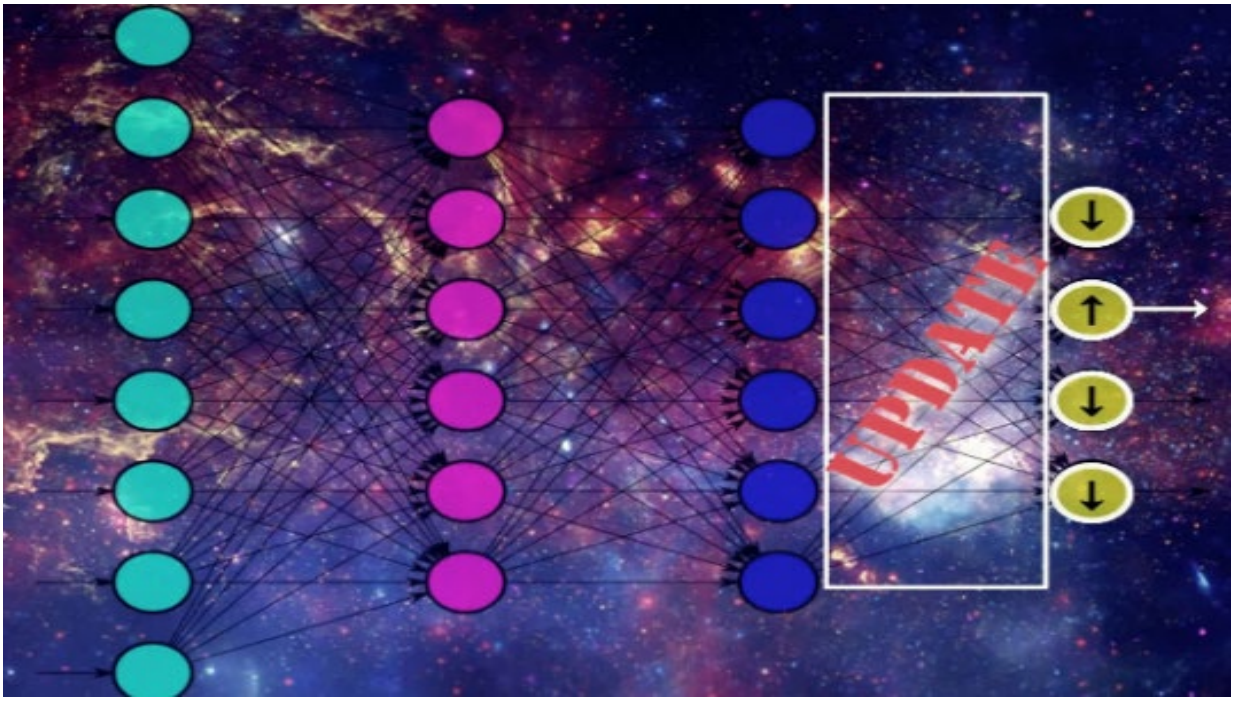
We have the output that was generated for our given input, the loss then gets calculated for that output, and now gradient descent starts updating our weights, using backpropagation, in order to minimize the loss function.

We're going to now focus on what backpropagation is doing by focusing on the intuition behind it.

Backpropagation Intuition

To update the weights, gradient descent is going to start by looking at the activation outputs from our output nodes.

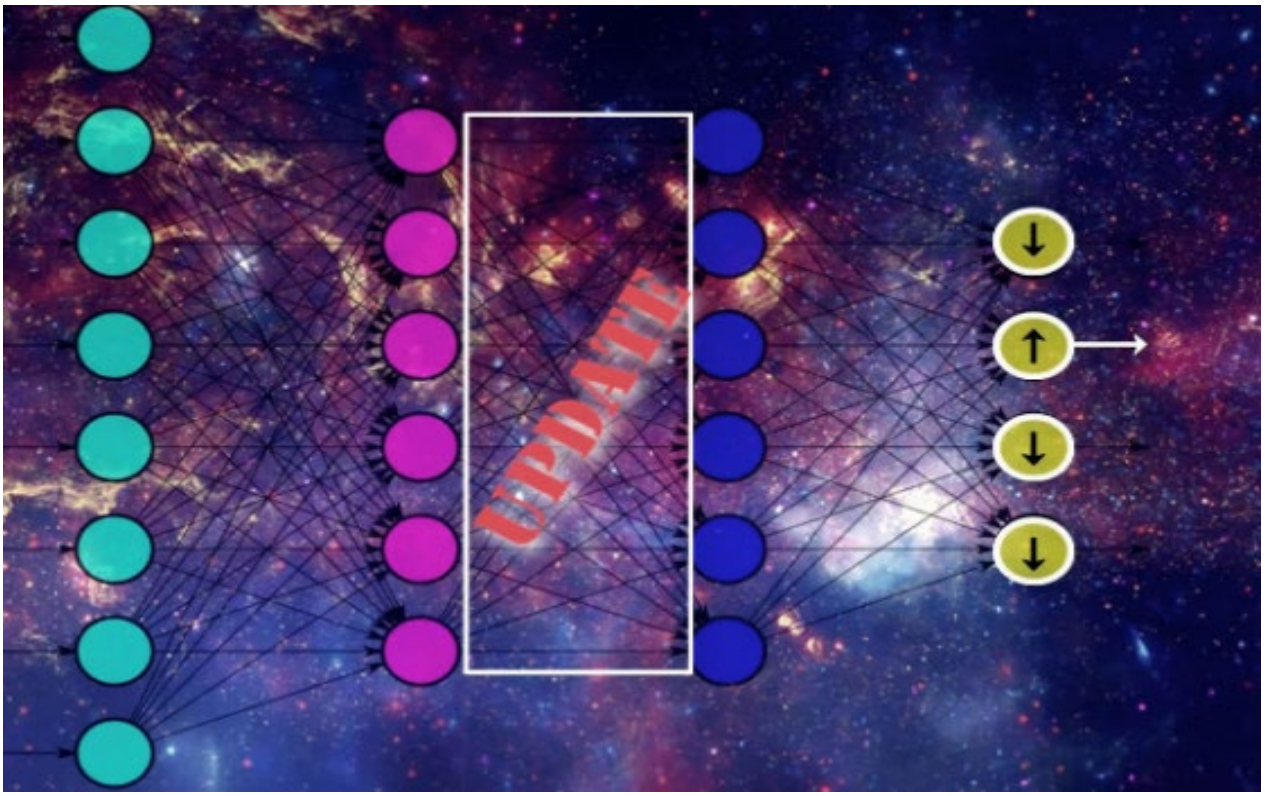
Suppose that this output node here with the up arrow pictured below maps to the output that our given input actually corresponds to. If that's the case, then gradient descent understands that the value of this output should increase, and the values from all the other output nodes should decrease. Doing this will help SGD lower the loss for this input.



We know that the values of these output nodes come from the weighted sum of the weights for the connections in the output layer here being multiplied by the output from the previous layer and then passing this weighted sum to the output layer's activation function.

Therefore, if we want to update the values for the output nodes in the way we just discussed, one way to do this is by updating the weights for these connections that are connected to the output layer. Another way of doing this is by changing the activation output from the previous layer.

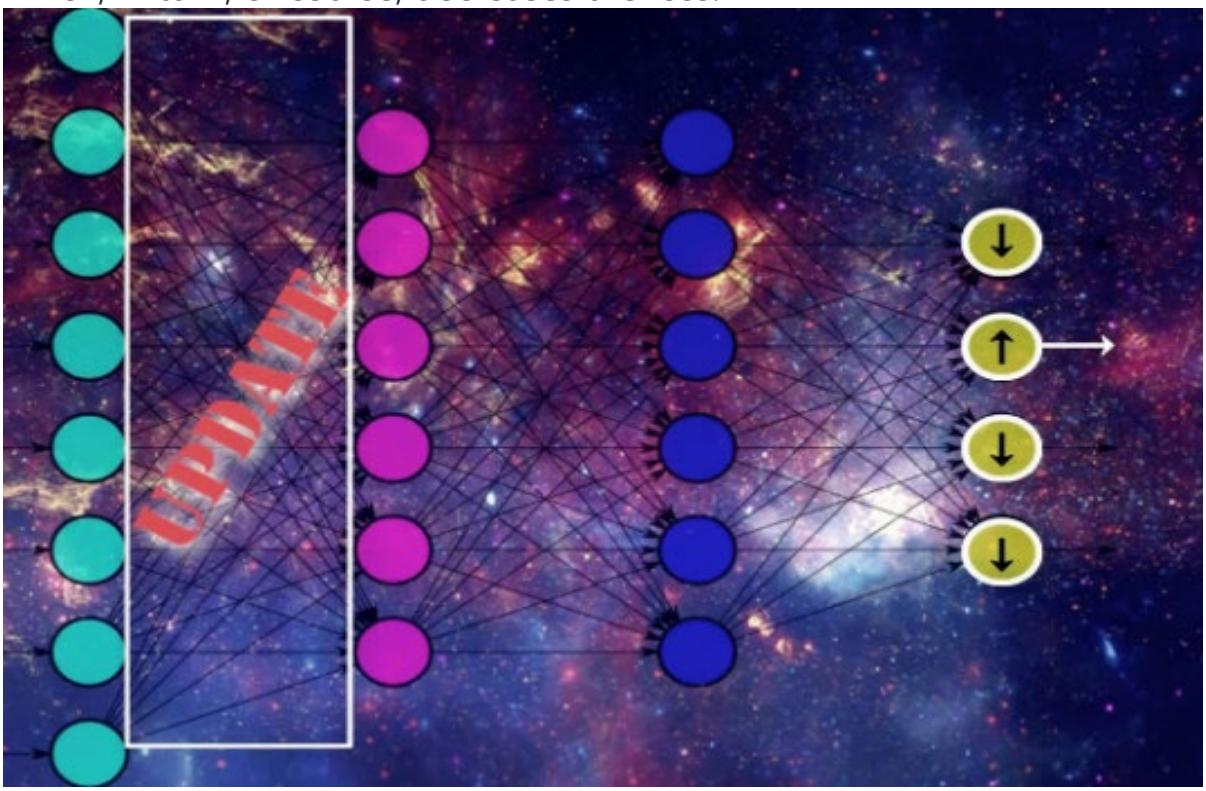
We can't actually directly change the activation output because it's a calculation based on the weights and the previous layer's output. But, we can *indirectly* influence a change in this layer's activation output by jumping backwards, and again, updating the weights here in the same way we just discussed for the output layer.



We continue this process until we reach the input layer. We don't want to change any of the values from the nodes in our input layer since this contains our actual input data.

As we can see, we're moving backwards through our network, updating the weights from right to left in order to slightly move the values from our output nodes in the direction that they should be going in order to help lower the loss.

This means that, for an individual sample, SGD is trying to increase the output value for the correct output node and decrease the output value for the incorrect output nodes, which, in turn, of course, decreases the loss.



It's also important to note, that in addition to updating weights to move in the desired direction i.e. positive or negative, backpropagation is also working to *efficiently* update the weights so that the updates are being done in a manner that helps to reduce the loss function most efficiently.

The proportion in which some weights are updated relative to others may be higher or lower, depending on how much affect the update is going to have on the network as a whole to lower the loss.

After calculating the derivatives, the weights are proportionally updated to their new values using the derivatives we obtain. The technical explanation for this update is shown in an earlier chapter.

We went through this example for a single input, but this exact same process will occur for all the input for each batch we provide to our network, and the resulting updates to the weights in the network are going to be the average updates that are calculated for each individual input.
These averaged results for each weight are indeed the corresponding gradient of our loss function with respect to each weight.

Summary Of This Process

Alright, we've done a lot, so let's give a quick summary of it all. When training an artificial neural network, we pass data into our model. The way this data flows through the model is via forward propagation where we're repeatedly calculating the weighted sum of the previous layers activation output with the corresponding weights, and then passing this sum to the next layer's activation function.

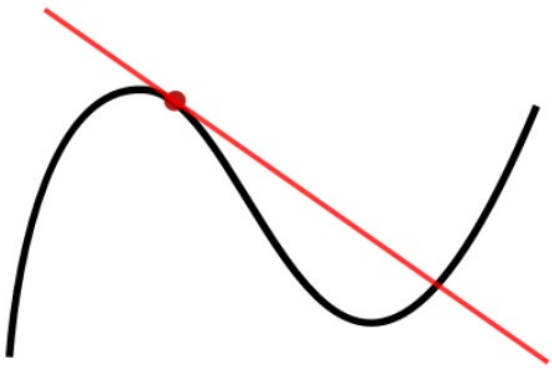
We do this until we reach the output layer. At this point, we calculate the loss on our output, and gradient descent then works to minimize this loss.

Gradient descent does this minimization process by first calculating the gradient of the loss function and then updating the weights in the network accordingly. To do the actual calculation of the gradient, gradient descent uses backpropagation.

Ok, so this covers the intuition behind what backpropagation is doing, but of course, this is all done with math behind the scenes.

Calculus Behind The Scenes

The backpropagation process we just went through uses calculus. Recall, that backpropagation is working to calculate the derivative of the loss with respect to each weight.



To do this calculation, backprop is using the chain rule to calculate the gradient of the loss function. If you've taken a calculus course, then you may be familiar with the chain rule as being a method for calculating the derivative of the composition of two or more functions. We'll start covering the mathematics of this process in the next chapter.

Recapping Backpropagation

We know that after we forward propagate our data through our network, the network gives an output for that data. The loss is then calculated for that predicted output based on what the true value of the original data is.

Stochastic gradient descent, or SGD, has the objective to minimize this loss. To do this, it calculates the derivative of the loss with respect to each of the weights in the network. It then uses this derivative to update the weights.

It does this process over and over again until it's found a minimized loss. We covered how this update is actually done using the learning rate in our previous chapter that covers how a neural network learns .

When SGD calculates the derivative, it's doing this using backpropagation. Essentially, SGD is using backprop as a tool to calculate the derivative, or the gradient, of the loss function.

Going forward, this is going to be our focus. All the math that we'll be covering in the next few posts will be for the sole purpose of seeing how backpropagation calculates the gradient of the loss function with respect to the weights.

Ok, we've now got our refresher of backprop out of the way, so let's jump over to the math!

Backpropagation Mathematical Notation

As discussed, we're going to start out by going over the definitions and notation that we'll be using going forward to do our calculations.

This table describes the notation we'll be using throughout this process.

Symbol	Definition
L	Number of layers in the network
l	Layer index
j	Node index for layer l
k	Node index for layer $l-1$
y_j	The value of node j in the output layer L for a single training sample
C_0	Loss function of the network for a single training sample
$w_j^{(l)}$	The vector of weights connecting all nodes in layer $l-1$ to node j in layer l
$w_{jk}^{(l)}$	The weight that connects node k in layer $l-1$ to node j in layer l
$z_j^{(l)}$	The input for node j in layer l
$g^{(l)}$	The activation function used for layer l
$a_j^{(l)}$	The activation output of node j in layer l

Let's narrow in and discuss the indices used in these definitions a bit further.

Importance Of Indices

Recall at the top of the table, we covered the notation that we'd be using to index the layers and nodes within our network. All further definitions then depended on these indices.

Symbol	Definition
l	Layer index
j	Node index for layer l
k	Node index for layer $l-1$

We saw that, for each of the terms we introduced, we have either a subscript or a superscript, or both. Sometimes, our subscript even had two terms, as we saw when we defined the weight between two nodes.

$$w_{jk}^{(l)}$$

These indices we're using everywhere may make the terms look a little intimidating and overly bulky. That's why I want to focus on this topic further here.

It turns out that if we use these indices properly and we understand their purpose, it's going to make our lives a lot easier going forward when working with these terms and will reduce any ambiguity or confusion, rather than induce it.

In code, when we run loops, like a for loop or a while loop that, the data that the loop is iterating over is an indexed sequence of data.

```
// pseudocode (java)
for (int i = 0; i < data.length; i++) {
    #do stuff
}
```

Indexed data allows the code to understand where to start, where to end, and where it is, at any given point in time, within the loop itself.

This idea of keeping track of where we are during an iteration over a sequence is precisely why keeping track of which layer, which node, which weight, or really, which anything that we introduced here, is important.

In the math in the upcoming post, we'll be seeing a lot of iteration, particularly via summation, where summation is simply the addition of a sequence of numbers. A summation is just the process of iterating over a sequence of values and summing them.

Math example:

Suppose that (a_n) is a sequence of numbers. The sum of this sequence is given $\sum_{j=1}^{n} a_j$

Code example:

Suppose that $a = [1,2,3,4]$ is a sequence of numbers. The sum is given by:

```
int sum = 0;
while (j < a.length) {
    sum = sum + a[j];
}
```

Aside from iteration, any time we choose a specific item to work with, like a particular layer, node, or weight, the indexing that we introduced here is what will allow us to properly reference this particular item that we've chosen to focus on.

As it turns out, backpropagation itself is an iterative process, iterating backwards through each layer, calculating the derivative of the loss function with respect to each weight for each layer.

Given this, it should be clear why these indices are required in order to make sense of the math going forward. Hopefully, rather than causing confusion within our notation, these indices can instead become intuition for when we think about doing anything iterative over our network.

Mathematical Observations For Backpropagation

The Path Forward

In our last chapter, we focused on the mathematical notation and definitions that we would be using going forward to show how backpropagation mathematically works to calculate the gradient of the loss function.

Well, now's the time that we'll start making use of them, so it's crucial that you have a full understanding of everything we covered in that chapter first.

Here, we're going to be making some mathematical observations about the training process of a neural network. The observations we'll be making are actually facts that we already know conceptually, but we'll now just be expressing them mathematically.

We'll be making these observations because the math for backprop that comes next, particularly, the differentiation of the loss function with respect to the weights, is going to make use of these observations.

We're first going to start out by making an observation regarding how we can mathematically express the loss function. We're then going to make observations around how we express the input and the output for any given node mathematically.

Lastly, we'll observe what method we'll be using to differentiate the loss function via backpropagation. Alright, let's begin.

Loss C_0

Observe that the expression $\left(a_j^{(L)} - y_j\right)^2$

is the squared difference of the activation output and the desired output for node j in the output layer L. This can be interpreted as the loss for node j in layer L.

Therefore, to calculate the total loss, we should sum this squared difference for each node j in the output layer L.

This is expressed as

$$C_0 = \sum_{j=0}^{n-1} \left(a_j^{(L)} - y_j\right)^2.$$

Input $z_j^{(l)}$

We know that the input for node j in layer l is the weighted sum of the activation outputs from the previous layer l−1.

An individual term from the sum looks like this: $w_{jk}^{(l)} a_k^{(l-1)}$

So, the input for a given node j in layer l is expressed as $$z_j^{(l)} = \sum_{k=0}^{n-1} w_{jk}^{(l)} a_k^{(l-1)}.$$

Activation Output $a_j^{(l)}$

We know that the activation output of a given node j in layer l is the result of passing the input, $z_j^{(l)}$, to whatever activation function we choose to use $g^{(l)}$

Therefore, the activation output of node j in layer l is expressed as $$a_j^{(l)} = g^{(l)}\left(z_j^{(l)}\right).$$

C_0 as a Composition Of Functions

Recall the definition of C_0,

$$C_0 = \sum_{j=0}^{n-1} \left(a_j^{(L)} - y_j\right)^2.$$

So the loss of a single node j in the output layer L can be expressed as

$$C_{0_j} = \left(a_j^{(L)} - y_j\right)^2.$$

We see that C_{0_j} is a function of the activation output of node j in layer L, and so we can express C_{0_j} as a function of $a_j^{(L)}$ as

$$C_{0_j}\left(a_j^{(L)}\right).$$

Observe from the definition of C_{0_j} that C_{0_j}also depends on y_j. Since $y_j$is a constant, we only observe C_{0_j}as a function of $a_j^{(L)}$, and $y_j$as a parameter that helps define this function.

The activation output of node j in the output layer L is a function of the input for node j. From an earlier observation, we know we can express this as

$$a_j^{(L)} = g^{(L)}\left(z_j^{(L)}\right).$$

The input for node j is a function of all the weights connected to node j. We can express $z_j^{(L)}$as a function of $w_j^{(L)}$as

$$z_j^{(L)}\left(w_j^{(L)}\right).$$

Therefore,

$$C_{0_j} = C_{0_j}\left(a_j^{(L)}\left(z_j^{(L)}\left(w_j^{(L)}\right)\right)\right).$$

Given this, we can see that C_0 is a composition of functions We know $C_0 = \sum_{j=0}^{n-1} C_{0_j}$,

and so using the same logic, we observe that the total loss of the network for a single input is also a composition of functions. This is useful in order to understand how to differentiate C_0.

Backpropagation Explained | Calculating The Gradient

Hey, what's going on everyone? In this episode, we're finally going to see how backpropagation calculates the gradient of the loss function with respect to the weights in a neural network.

Our Task

We're now on episode number four in our journey through understanding backpropagation. In the last episode, we focused on how we can mathematically express certain facts about the training process.

Now we're going to be using these expressions to help us differentiate the loss of the neural network with respect to the weights.

Recall from the episode that covered the intuition for backpropagation that for stochastic gradient descent to update the weights of the network, it first needs to calculate the gradient of the loss with respect to these weights.

Calculating this gradient is exactly what we'll be focusing on in this episode.

We're first going to start out by checking out the equation that backprop uses to differentiate the loss with respect to weights in the network.

Then, we'll see that this equation is made up of multiple terms. This will allow us to break down and focus on each of these terms individually.

Lastly, we'll take the results from each term and combine them to obtain the final result, which will be the gradient of the loss function.

Alright, let's begin.

Derivative Of The Loss Function With Respect To The Weights

Let's look at a single weight that connects node 2 in layer L−1 to node 1 in layer L.

This weight is denoted as $w_{12}^{(L)}$.

The derivative of the loss C_0 with respect to this particular weight $w_{12}^{(L)}$ is denoted as

$$\frac{\partial C_0}{\partial w_{12}^{(L)}}.$$

Since C_0 depends on $a_1^{(L)}$, and $a_1^{(L)}$, depends on $z_1^{(L)}$, and $z_1^{(L)}$, depends on $w_{12}^{(L)}$, the chain rule tells us that to differentiate C_0 with respect to $w_{12}^{(L)}$, we take the product of the derivatives of the composed function.

This is expressed as $$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right).$$

Let's break down each term from the expression on the right hand side of the above equation.

The First Term: $\frac{\partial C_0}{\partial a_1^{(L)}}$

We know that $C_0 = \sum_{j=0}^{n-1}\left(a_j^{(L)} - y_j\right)^2$. Therefore, $\frac{\partial C_0}{\partial a_1^{(L)}} = \frac{\partial}{\partial a_1^{(L)}}\left(\sum_{j=0}^{n-1}\left(a_j^{(L)} - y_j\right)^2\right)$.

Expanding the sum, we see

$$\frac{\partial}{\partial a_1^{(L)}}\left(\sum_{j=0}^{n-1}\left(a_j^{(L)} - y_j\right)^2\right) =$$

$$\begin{aligned}
&= \frac{\partial}{\partial a_1^{(L)}}\left(\left(a_0^{(L)} - y_0\right)^2 + \left(a_1^{(L)} - y_1\right)^2 + \left(a_2^{(L)} - y_2\right)^2 + \left(a_3^{(L)} - y_3\right)^2\right) \\
&= \frac{\partial}{\partial a_1^{(L)}}\left(\left(a_0^{(L)} - y_0\right)^2\right) + \frac{\partial}{\partial a_1^{(L)}}\left(\left(a_1^{(L)} - y_1\right)^2\right) + \frac{\partial}{\partial a_1^{(L)}}\left(\left(a_2^{(L)} - y_2\right)^2\right) + \frac{\partial}{\partial a_1^{(L)}}\left(\left(a_3^{(L)} - y_3\right)^2\right) \\
&= 2\left(a_1^{(L)} - y_1\right).
\end{aligned}$$

Observe that the loss from the network for a single input sample will respond to a small change in the activation output from node 1 in layer L by an amount equal to two times the difference of the activation output a_1 for node 1 and the desired output y_1 for node 1.

The Second Term: $\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}$

We know that for each node j in the output layer L, $a_j^{(L)} = g^{(L)}\left(z_j^{(L)}\right)$, we have

and since j=1, we have $a_1^{(L)} = g^{(L)}\left(z_1^{(L)}\right)$.

Therefore,
$$\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} = \frac{\partial}{\partial z_1^{(L)}}\left(g^{(L)}\left(z_1^{(L)}\right)\right)$$
$$= g'^{(L)}\left(z_1^{(L)}\right).$$

Therefore, this is just the direct derivative of $a_1^{(L)}$ since $a_1^{(L)}$ is a direct function of $z_1^{(L)}$

The Third Term: $\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}$

We know that, for each node j in the output layer L, we have $z_j^{(L)} = \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$.

Since j=1, we have $z_1^{(L)} = \sum_{k=0}^{n-1} w_{1k}^{(L)} a_k^{(L-1)}$.

Therefore, $\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} = \frac{\partial}{\partial w_{12}^{(L)}}\left(\sum_{k=0}^{n-1} w_{1k}^{(L)} a_k^{(L-1)}\right)$.

Expanding the sum, we see that $\frac{\partial}{\partial w_{12}^{(L)}}\left(\sum_{k=0}^{n-1} w_{1k}^{(L)} a_k^{(L-1)}\right)$

$$= \frac{\partial}{\partial w_{12}^{(L)}}\left(w_{10}^{(L)} a_0^{(L-1)} + w_{11}^{(L)} a_1^{(L-1)} + w_{12}^{(L)} a_2^{(L-1)} + \cdots + w_{15}^{(L)} a_5^{(L-1)}\right)$$
$$= \frac{\partial}{\partial w_{12}^{(L)}} w_{10}^{(L)} a_0^{(L-1)} + \frac{\partial}{\partial w_{12}^{(L)}} w_{11}^{(L)} a_1^{(L-1)} + \frac{\partial}{\partial w_{12}^{(L)}} w_{12}^{(L)} a_2^{(L-1)} + \cdots + \frac{\partial}{\partial w_{12}^{(L)}} w_{15}^{(L)} a_5^{(L-1)}$$
$$= a_2^{(L-1)}$$

The input for node 1 in layer L will respond to a change in the weight w12(L) by an amount equal to the activation output for node 2 in the previous layer, L−1.

Combining terms

Combining all terms, we have
$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right)$$
$$= 2\left(a_1^{(L)} - y_1\right)\left(g'^{(L)}\left(z_1^{(L)}\right)\right)\left(a_2^{(L-1)}\right)$$

We Conclude

We've seen how to calculate the derivative of the loss with respect to one individual weight for one individual training sample.

To calculate the derivative of the loss with respect to this same particular weight, w_{12}, for all n training samples, we calculate the average derivative of the loss function over all n training samples.

This can be expressed as

$$\frac{\partial C}{\partial w_{12}^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial C_i}{\partial w_{12}^{(L)}}.$$

We would then do this same process for each weight in the network to calculate the derivative of C with respect to each weight.

Backpropagation | What Puts The "Back" In Backprop?

Now, we're going to build on the knowledge that we've already developed to understand what exactly puts the back in backpropagation.

The explanation we'll give for this will be math-based, so we're first going to start out by exploring the motivation needed for us to understand the calculations we'll be working through.

We'll then jump right into the calculations, which we'll see, are actually quite similar to ones we've worked through in the previous episode.

After we've got the math down, we'll then bring everything together to achieve the mind-blowing realization for how these calculations are mathematically done in a backwards fashion.

Alright, let's begin.

Motivation

We left off from our last episode by seeing how we can calculate the gradient of the loss function with respect to any weight in the network. When we went through the process for showing how that was calculated, recall that, we worked with this single weight in the output layer of the network. $w_{12}^{(L)}$

Then, we generalized the result we obtained by saying this same process could be applied for all the other weights in the network.

For this particular weight, we saw that the derivative of the loss with respect to this weight was equal to this

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right).$$

Now, what would happen if we chose to work with a weight that is not in the output layer, like this weight here? $w_{22}^{(L-1)}$

Well, using the formula we obtained for calculating the gradient of the loss, we see that the gradient of the loss with respect to this particular weight is equal to this

$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right).$$

Alright, check it out. This equation looks just like the equation we used for the previous weight we were working with.

The only difference is that the superscripts are different because now we're working with a weight in the third layer, which we're denoting as L−1, and then the subscripts are different as well because we're working with the weight that connects the second node in the second layer to the second node in the third layer.

Given this is the same formula, then we should just be able to calculate it in the exact same way we did for the previous weight we worked with in the last episode, right?

Well, not so fast.

So yes, this is the same formula, and in fact, the second and third terms on the right hand side will be calculated using the same exact approach as we used before.

The first term on the right hand side of the equation is the derivative of the loss with respect to this one activation output, and for this one, there's actually a different approach required for us to calculate it. $\left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)$

Let's think about why.

When we calculated the derivative of the loss with respect to a weight in the output layer, we saw that the first term is the derivative of the loss with respect to the activation output for a node in the output layer. $\left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)$

Well, as we've talked about before, the loss is a direct function of the activation output of all the nodes in the output layer. You know, because the loss is the sum of the squared errors between the actual labels of the data and the activation output of the nodes in the output layer.

Ok, so, now when we calculate the derivative of the loss with respect a weight in layer L−1, for example, the first term is the derivative of the loss with respect to the activation output for node two, not in the output layer, L, but in layer L−1.

And, unlike the activation output for the nodes in the output layer, the loss is not a direct function of this output.

See, because consider where this activation output is within the network, and then consider where the loss is calculated at the end of the network. We can see that the output is not being passed directly to the loss.

What we need now is to understand how to calculate this first term then. That's going to be our focus here. $\left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)$

If needed, go back and watch the previous episode where we calculated the first term in the first equation to see the approach we took.

Then, use that information to compare with the approach we're going to use to calculate the first term in second equation.

Which	Equation
First	$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right)$$
Second	$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right)$$

Now, because the second and third terms on the right hand side of the second equation are calculated in the exact same manner as we've seen before, we're not going to cover those here.

We're just going to focus on how to calculate the first term on the right hand side of the second equation, and then we'll combine the results from all terms to see the final result.

Alright, at this point, go ahead and admit, you're thinking to yourself:

"I'm here to see how backprop works backwards. What the heck does any of this so far have to do with the backwards movement of backpropagation?"

I hear you. We're getting there, so stick with me. We have to go through the math first and see what it's doing, and then once we see that, we'll be able to clearly see the whole point of the backwards movement.

So let's go ahead and jump in to the calculations.

Calculations

Alright, time to get set up.

We're going to show how we can calculate the derivative of the loss function with respect to the activation output for any node that is not in the output layer. We're going to work with a single activation output to illustrate this.

Particularly, we'll be working with the activation output for node 2 in layer L−1.

This is denoted as $\boldsymbol{a_2^{(L-1)}}$,

and the partial derivative of the loss with respect to this activation output is denoted as this $\frac{\partial C_0}{\partial a_2^{(L-1)}}$.

Observe that, for each node j in L, the loss C_0 depends on on $a_j^{(L)}$, and $a_j^{(L)}$ depends on $z_j^{(L)}$ The node $z_j^{(L)}$ depends on all of the weights connected to node j from the previous layer, L−1, as well as all the activation outputs from L−1.

This means that the node, $z_j^{(L)}$ depends on $a_2^{(L-1)}$

Ok, now the activation output for each of these nodes depends on the input to each of these nodes.

In turn, the input to each of these nodes depends on the weights connected to each of these nodes from the previous layer, L−1, as well as the activation outputs from the previous layer.

Given this, we can see how the input to each node in the output layer is dependent on the activation output that we've chosen to work with, the activation output for node 2 in layer L−1.

Using similar logic to what we used in the previous episode, we can see from these dependencies that the loss function is actually a composition of functions, and so, to calculate the derivative of the loss with respect to the activation output we're working with, we'll need to use the chain rule.

The chain rule tells us that to differentiate C_0 with respect to $a_2^{(L-1)}$ we take the product of the derivatives of the composed function. This derivative can be expressed as

$$\frac{\partial C_0}{\partial a_2^{(L-1)}} = \sum_{j=0}^{n-1} \left(\left(\frac{\partial C_0}{\partial a_j^{(L)}} \right) \left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \right) \left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} \right) \right).$$

This tells us that the derivative of the loss with respect to the activation output for node 2 in layer L−1 is equal to the expression on the right hand side of the above equation.

This is the sum for each node j in the output layer, L, of the derivative of the loss with respect to the activation output for node j, times the derivative of the activation output for node j with respect to the input for node j, times the input for node j with respect to the activation output for node 2 in layer L−1.

Now, actually, this equation looks almost identical to the equation we obtained in the last episode for the derivative of the loss with respect to a given weight. Recall that this previous derivative with respect to a given weight that we worked with was expressed as

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}} \right) \left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} \right) \left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} \right).$$

Just eye-balling the general likeness between these two equations, we see that the only differences are one, the presence of the summation operation in our new equation, and two, the last term on the right hand side differs.

The reason for the summation is due to the fact that a change in one activation output in the previous layer is going to affect the input for each node j in the following layer L, so we need to sum up these effects.

Now, we can see that the first and second terms on the right hand side of the equation are the same as the first and second terms in the last equation with regards to $w_{12}^{(L)}$ in the output layer when j=1.

Since we've already gone through the work to find how to calculate these two derivatives in the last episode, we won't do it again here.

We're only going to focus on breaking down the third term, and then we'll combine all terms to see the final result.

The Third Term

Alright, so let's jump in to how to calculate the third term from the equation we just looked at.
The third term is the derivative of the input to any node j in the output layer L with respect to the activation output for node 2 in layer L−1. $\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}$

We know for each node j in layer L that $z_j^{(L)} = \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$.

Therefore, we can substitute this expression in for $z_j^{(L)}$ in our derivative.

$$\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} = \frac{\partial}{\partial a_2^{(L-1)}} \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}.$$

Expanding the sum, we have $\frac{\partial}{\partial a_2^{(L-1)}} \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)} =$

$$= \frac{\partial}{\partial a_2^{(L-1)}} \left(w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + \cdots + w_{j5}^{(L)} a_5^{(L-1)} \right)$$
$$= \frac{\partial}{\partial a_2^{(L-1)}} w_{j0}^{(L)} a_0^{(L-1)} + \frac{\partial}{\partial a_2^{(L-1)}} w_{j1}^{(L)} a_1^{(L-1)} + \frac{\partial}{\partial a_2^{(L-1)}} w_{j2}^{(L)} a_2^{(L-1)} + \cdots + \frac{\partial}{\partial a_2^{(L-1)}} w_{j5}^{(L)} a_5^{(L-1)}$$
$$= 0 + 0 + \frac{\partial}{\partial a_2^{(L-1)}} w_{j2}^{(L)} a_2^{(L-1)} + \cdots 0$$
$$= \frac{\partial}{\partial a_2^{(L-1)}} w_{j2}^{(L)} a_2^{(L-1)}$$
$$= w_{j2}^{(L)}$$

Due to the linearity of the summation operation, we can pull the derivative operator through to each term since the derivative of a sum is equal to the sum of the derivatives.

This means we're taking the derivatives of each of these terms with respect to $a_2^{(L-1)}$, but actually we can see that only one of these terms contain $a_2^{(L-1)}$

This means that when we take the derivative of the other terms that don't contain $a_2^{(L-1)}$ these terms will evaluate to zero.

Now taking the derivative of this one term that does contain $a_2^{(L-1)}$ we apply the power rule, to obtain the result.

This result says that the input for any node j in layer L will respond to a change in the activation output for node 2 in layer L−1 by an amount equal to the weight connecting node 2 in layer L−1 to node j in layer L.

Alright, let's now take this result and combine it with our other terms to see what we get as the total result for the derivative of the loss with respect to this activation output.

Combining The Terms

Alright, so we have our original equation here for the derivative of the loss with respect to the activation output we've chosen to work with.

$$\begin{aligned}\frac{\partial C_0}{\partial a_2^{(L-1)}} &= \sum_{j=0}^{n-1}\left(\left(\frac{\partial C_0}{\partial a_j^{(L)}}\right)\left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\right)\left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}\right)\right) \\ &= \sum_{j=0}^{n-1}\left(2\left(a_j^{(L)}-y_j\right)\left(g'^{(L)}\left(z_j^{(L)}\right)\right)\left(w_{j2}^{(L)}\right)\right)\end{aligned}$$

From the previous episode, we already know what these first two terms evaluate to. So I've gone ahead and plugged in those results, and since we have just seen what the result of the third term is, we plug it in as well.

Ok, so we've got this full result. Now what was it that we wanted to do with it again?

Oh yeah, now we can use this result to calculate the gradient of the loss with respect to any weight connected to node 2 in layer L−1, like we saw for $w_{22}^{(L-1)}$ for example, with the following equation

$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right)$$

The result we just obtained for the derivative of the loss with respect to the activation output for node 2 in layer L−1 can then be substituted for the first term in this equation.

As mentioned earlier, the second and third terms are calculated using the exact same approach we took for those terms in the previous episode.

Notice that we've used the chain rule twice now. With one of those times being nested inside the other. We first used the chain rule to obtain the result for this entire derivative for the loss with respect to the given weight.

Then, we used it again to calculate the first term within this derivative, which itself was the derivative of the loss with respect to this activation output.

The results from each of these derivatives using the chain rule depended on derivatives with respect to components that reside later in the network.

Essentially, we're needing to calculate derivatives that depend on components later in the network first and then use these derivatives in our calculations of the gradient of the loss with respect to weights that come earlier in the network.

We achieve this by repeatedly applying the chain rule in a backwards fashion.

Average Derivative Of The Loss Function

Note, to find the derivative of the loss function with respect to this same particular activation output, a2(L−1), for all n training samples, we calculate the average derivative of the loss function over all n training samples. This can be expressed as

$$\frac{\partial C}{\partial a_2^{(L-1)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial C_i}{\partial a_2^{(L-1)}}.$$

Setting Things Up for vanishing gradient problem

What do we already know about gradients as it pertains to neural networks?

Well, for one, when we use the word *gradient* by itself, we're typically referring to the gradient of the loss function with respect to the weights in the network.

We also know how this gradient is calculated, using backpropagation, which we covered in our earlier episodes dedicated solely to backprop.

Finally, as we saw in the episode that demonstrates how a neural network learns, we know what to do with this gradient after it's calculated. We update our weights with it!

Well, we don't per se, but stochastic gradient descent does, with the goal in mind to find the most optimal weight for each connection that will minimize the total loss of the network.

With this understanding, we're now going to talk about the *vanishing gradient problem*.

We're first going to answer, well, what the heck is the vanishing gradient problem anyway?

Here, we'll cover the idea conceptually. Then, we'll move our discussion to talking about how this problem occurs, and with the understanding that we'll have developed up to this point, we'll discuss the problem of exploding gradients.

We'll see that the exploding gradient problem is actually very similar to the vanishing gradient problem, and so we'll be able to take what we learned about that problem and apply it to this new one.

What Is The Vanishing Gradient Problem?

What is the vanishing gradient problem anyway?
In general, the vanishing gradient problem is a problem that causes major difficulty when training a neural network. More specifically, this is a problem that involves weights in earlier layers of the network.
Recall that, during training, stochastic gradient descent (or SGD) works to calculate the gradient of the loss with respect to weights in the network.
Now, sometimes the gradient with respect to weights in earlier layers of the network becomes really small, like vanishingly small. Hence, vanishing gradient.
Ok, what's the big deal with a small gradient?

Small Gradients

Well, once SGD calculates the gradient with respect to a particular weight, it uses this value to update that weight, and the weight gets updated in some way that is proportional to the gradient. If the gradient is vanishingly small, then this update is, in turn, going to be vanishingly small as well.

Therefore, if this newly updated value of the weight has just barely moved from its original value, then it's not really doing much for the network. This change is not going to carry through the network very well to help to reduce the loss because it has barely changed at all from where it was before the update occurred.

As a result, this weight becomes kind of stuck, never really updating enough to even get close to its optimal value which has implications for the remainder of the network to the right of this one weight and impairs the ability of the network to learn well.

This is the problem, and now that we know what this problem is, how exactly does this problem occur?

Well, we know from what we learned about backpropagation that the gradient of the loss with respect to any given weight is going to be the product of some derivatives that depend on components that reside later in the network.

Given this, we can deduce that the earlier in the network a weight lives, the more terms will be needed in the product we just mentioned to get the gradient of the loss with respect to this weight.

The key now is to understand what happens if the terms in this product, or at least some of them, are small? And by small, we mean less than one, small.

Well, the product of a bunch of numbers less than one is going to give us an even smaller number, right? Ok cool

As we mentioned earlier, we now take this result, the small number, and update our weight with it. Recall that we do this update by first multiplying this number by our learning rate, which it itself is a small number, usually ranging between $.01$ and $.0001$.

Now the result of this product is even a smaller number. After this smaller number is obtained, we subtract the number from the weight, and the final result of this difference is going to be the value of the updated weight.

Stuck Weights

Now, we can think about if the gradient that we obtain with respect to this weight is already really small, i.e., vanishing, then by the time we multiply it by the learning rate, the product is going to be even smaller, and so when we subtract this teeny tiny number from the weight, it's just barely going to move the weight at all.

Essentially, the weight gets into this kind of stuck state. Not moving, not *learning*, and therefore not really helping to meet the overall objective of minimizing the loss of the network.

We can see why earlier weights are subject to this problem. Because, as we said, the earlier in the network the weight resides, the more terms are going to be included in the product to calculate the gradient.

The more terms we're multiplying together that are less than one, the quicker the gradient is going to vanish.

Exploding Gradient

Now let's talk about this problem in the opposite direction. Not a gradient that vanishes, but rather, a gradient that explodes.

Think about the conversation we just had about how the vanishing gradient problem occurs with weights early in the network due to a product of, at least some, relatively small values.

Now think about calculating the gradient with respect to the same weight, but instead of really small terms, what if they were large? And by large, we mean greater than one.

Well, if we multiply a bunch of terms together that are all greater than one, we're going to get something greater than one, and perhaps even a lot greater than one.
The same argument holds here that we discussed about the vanishing gradient, where, the earlier in the network a weight lives, the more terms will be needed in the product we just mentioned.
As a result, we can see that the more of these larger valued terms we have being multiplied together, the larger the gradient is going to be, thus essentially exploding in size.
With this gradient, we go through the same process to proportionally update our weight with it that we talked about earlier.
However, this time, instead of barely moving our weight with this update, we're going to greatly move it, So much so, that the optimal value for this weight won't be achieved because the proportion to which the weight becomes updated with each epoch is just too large and continues to move further and further away from its optimal value.

Conclusion

A main-takeaway that we should be able to gain from this discussion is that the problem of vanishing gradients and exploding gradients is actually a more general problem of unstable gradients.

Weight Initialization ExplainedA (Way To Reduce The Vanishing Gradient Problem)

In an artificial neural network, we know that weights are what connect the nodes between layers. To kick off our discussion on weight initialization, we're first going to discuss how these weights are initialized, and how these initialized values might negatively affect the training process.

With this in mind, we'll then explore what we can do to influence how this initialization occurs. Then, we'll see how we can specify how the weights for a given model are initialized in code using Keras.

How Are Weights Initialized?

So, how are weights even initialized in the first place? We briefly touched on this concept in our chapter on backpropagation. Recall there, we mentioned that weights are *randomly initialized*.

To elaborate, whenever we build and compile a network, the values for the weights will be set with random numbers. One random number per weight. Typically, these random numbers will be normally distributed such that the distribution of these numbers has a mean of 0 and standard deviation of 1.

So, how does this random initialization impact training? To see this, let's consider the following example.

Random Initialization Example

Suppose that our neural network's input layer has 250 nodes, and for simplicity, suppose that the value of each of these 250 nodes is 1.

Now, let's focus only on the weights that connect the input layer to a single node in the first hidden layer. In total, there will be 250 weights connecting this node in our first hidden layer to all the nodes in the input layer. (See the corresponding video for an illustration.)

Now, each of these weights were randomly generated and normally distributed with a mean of 0 and a standard deviation of 1. So, what does this mean for the weighted sum, z, that this node accepts as input?

Note, in our case, all the input nodes have a value of 1, so each weight in z will be multiplied by a 1, so z becomes just a sum of the weights.

So back to how this random initialization affects z, more specifically we want to know what this means for the variance of z. (Stick with me for a sec, we'll see why we care about this in a minute.)

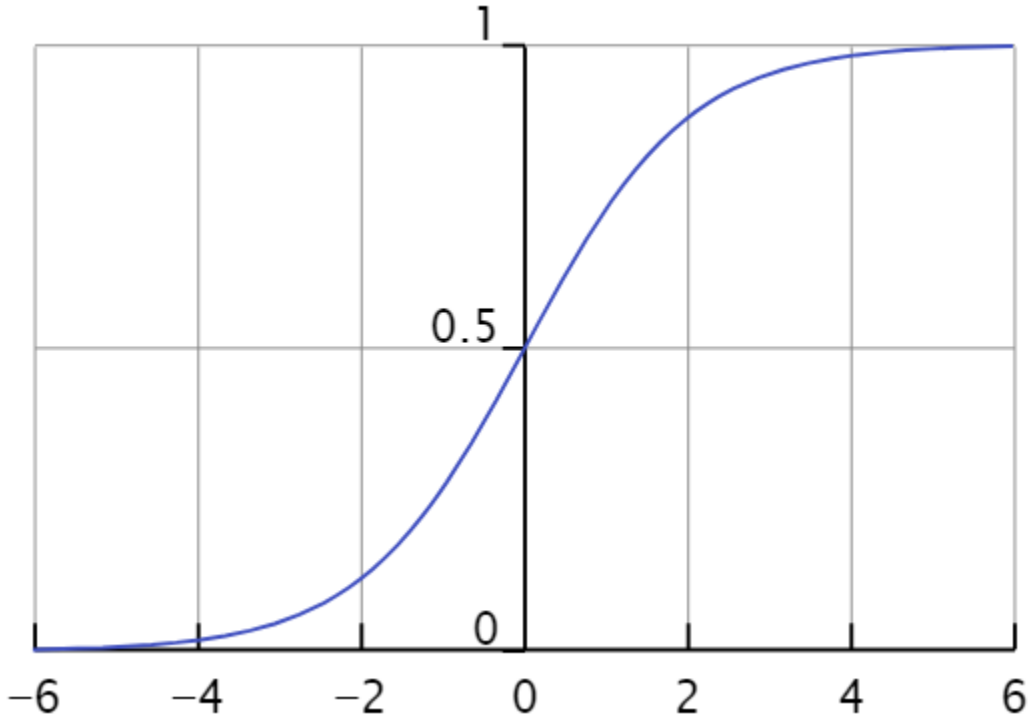
Well, z, as a sum of normally distributed numbers with a mean of 0, will also be normally distributed around 0, but its variance, and similarly its derived standard deviation, will be larger than 1.

That's because the variance of a sum of independent random numbers is the sum of the variances of each of these numbers. So, since the variance for each of our random numbers is 1, that means the variance of z, which is the sum of these 250 numbers, is 250.

Taking the square root of this value, we see that z has a standard deviation of 15.811.

So looking at the normal distribution of z, we see that it's quite broader than a normal distribution with a standard deviation of 1.

With this larger standard deviation, the value of z is more likely to take on a number that is significantly larger or smaller than 1. When we pass this value to our activation function, then, if we're using sigmoid, for example, we know that most positive inputs, especially these that we're saying will be significantly larger than 1, will be mapped to the value 1. Similarly, most negative inputs will be mapped to 0.



Problems With Random Initialization

If the desired output from our activation function is on the opposite side from where it saturated, then during training, when SGD updates the weights in attempts to influence the activation output, it will only make very small changes in the value of this activation output, barely even incrementally moving it in the right direction.

Thus, the network's ability to learn becomes hindered, and training is stuck running in this slow and inefficient state.

These problems that we've discussed so far with weight initialization also contribute to the vanishing and exploding gradient problem.

Given this random weight initialization causes issues and instabilities with training, can we do anything to help ourselves out here? Can we change how weights are initialized?

I'm glad you asked. Yes, yes we can.

Xavier Initialization

In hindsight, we should be able to look back at the problems we've discussed and trace them back to being caused by the weighted sum taking on a variance that is decently larger, or smaller, than 1. So to tackle this problem, what we can do is force this variance to be smaller.

How do we do this?
Well, since the variance of the input for a given node is determined by the variance of the weights connected to this node from the previous layer, we need to shrink the variance of these weights, which will shrink the variance of the weighted sum.

Some researchers identified a value for the variance of the weights that seems to work pretty well to mitigate the earlier problems we discussed. The value for the variance of the weights connected to a given node is $1/n$, where n is the number of weights connected to this node from the previous layer.

So, rather than the distribution of these weights be centered around 0 with a variance of 1, which is what we had earlier, they are now still centered around 0, but with a significantly smaller variance, $1/n$.

It turns out that, to get these weights to have this variance of $1/n$, what we do is, after randomly generating the weights centered around 0 with variance 1, we multiply each of them by $\sqrt{1/n}$. Doing this causes the variance of these weights to shift from 1 to 1/n. This type of initialization is referred to as *Xavier initialization* and also *Glorot initialization*.

It's important to note that actually, if we're using relu as our activation function, which is highly likely, then this ideal value for the variance is 2/n rather than 1/n. Besides that, everything else stated so far for this solution is the same. This value just happens to be what works better for relu.

Also, note that, given how we defined n as being the number of weights connected to a given node from the previous layer, we can see that this weight initialization process occurs on a per-layer basis.

Another thing also worth noting that when this Xavier initialization was originally announced, it was suggested to use $2/n_{in} + n_{out}$ as the variance where $n_{in}$is defined as the number of weights coming into this neuron, and $n_{out}$is the number of weights coming out of this neuron. You may still still see this value referenced in some places.

Now, we've talked a lot about Xavier initialization. Aside from this one, there are other initialization techniques that you can explore, but this Xavier is currently one of the most popular and has an aim to reduce the vanishing and exploding gradient problem.

Weight Initialization In Keras

Let's see now how we can specify a weight initializer for our own model in code using Keras.

We're going to use this arbitrary model that has two hidden `Dense` layers and an output layer with two nodes.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(16, input_shape=(1,5), activation='relu'),
    Dense(32, activation='relu', kernel_initializer='glorot_uniform'),
    Dense(2, activation='softmax')
])
```

Now, almost everything shown in this model has been covered in previous episodes of this series, so we won't touch on these items individually. Instead, we'll focus on the one single thing we haven't yet seen before, which is the `kernel_initializer` parameter in the second hidden layer.

This is the parameter we use to specify what type of weight initialization we want to use for a given layer. Here, I've set the value equal to `glorot_uniform`. This is the Xavier initialization using a uniform distribution. You can also use `glorot_normal` for Xavier initialization using a normal distribution.

Now actually, if you specify nothing at all, by default Keras initializes the weights in each layer with the `glorot_uniform` initialization. This is true for other layer types as well, not just `Dense`. Convolutional layers, for example, also use the `glorot_uniform` initializer by default as well.

There are several other initializers that Keras supports besides the two we just mentioned, and they're all documented in the Keras docs.

So, for each layer, we can just choose to leave the initializer as this default one, or, if we don't want to use `glorot_uniform`, we can explicitly state the value for the `kernel_initializer` parameter that we want to use, like `glorot_normal`, for example.

So hey, lucky us! Keras has been initializing these weights for us using Xavier initialization this whole time without us even knowing.

Wrapping Up

What we can draw from this entire discussion is that weight initialization plays a key role in how well and how quickly we can train our networks. It all links back to the idea of the inputs to our neurons having large variance when we just randomly generate a normally distributed set of weights.

So, to combat this, we can just shrink this variance, and we saw that doing this wasn't actually that bad. In Keras, it's not bad at all because it's already done for us!

Bias In An Artificial Neural Network Explained || How Bias Impacts Training

Background

When reading up on artificial neural networks, you may have come across the term *bias*. It's sometimes just referred to as bias. Other times you may see it referenced as bias nodes, bias neurons, or bias units within a neural network.

We're going to break this bias down and see what it's all about.

We'll first start out by discussing the most obvious question of, well, what is bias in an artificial neural network?

We'll then see, within a network, how bias is implemented.

Then, to hit the point home, we'll explore a simple example to illustrate the impact that bias has when introduced to a neural network.

Understanding Bias Inside Neural Networks

Let's get started by working to understand what exactly bias is inside neural networks.

What Is Bias?

So, what is bias in an artificial neural network?

Well, first, when we talk about bias, we're talking about it on a per-neuron basis. We can think of each neuron as having its own bias term, and so the entire network will be made up of multiple biases.

Now, the values assigned to these biases are learnable, just like the weights. Just how stochastic gradient descent learns and updates the weights via backpropagation during training, SGD is also learning and updating the biases as well.

Now, conceptually, we can think of the bias at each neuron as having a role similar to that of a threshold. This is because the bias value is what's going to determine whether or not the activation output from a neuron is going to be propagated forward through the network.

In other words, the bias is determining whether or not, or by how much, a neuron will fire. It's letting us know when a neuron is meaningfully activated. As we'll see in a few moments, the addition of these biases ends up increasing the flexibility of a model to fit the given data.

Where Bias Fits In

Alright, so we have an idea of what bias is now, but where exactly does it fit into the scheme of things? As we've discussed in past episodes, we know how each neuron receives a weighted sum of input from the previous layer, and then that weighted sum gets passed to an activation function.

Well, the bias for a neuron is going to fit right in here within this process. What we do is, rather than pass the weighted sum directly to the activation function, we instead pass the weighted sum plus the bias term to the activation function.

Ok, well, what good is that?

Let's look at a simple example that illustrates the role of this bias term in action.

Example That Shows Bias In Action

Suppose we have a neural network that has an input layer with just two nodes. Suppose the first node has a value of 1, and the second node has a value of 2.

Now we're going to focus our attention on a single neuron within the first hidden layer that directly follows the input layer.

The activation function we will use for this first hidden layer is relu, and we're going to assign some randomly generated weights to our connections.

Now, let's see what the output of this node would be without introducing any bias.

The weighted sum that this node receives is given by
$(1 \times -0.55) + (2 \times 0.10) = -0.35$.
We pass this result to relu. We know that the value of relu at any given input will be the maximum of either zero or the input itself, and in our case, we have

$\text{relu}(-0.35) = 0$.

With an activation output of zero, the neuron is considered to not be activated, or not firing. In fact, with relu, any neuron with a weighted sum of input is less than or equal to zero will not be firing, and so no information from these non-activated neurons will be passed forward through to the rest of the network.

Essentially, zero is the threshold here for the weighted sum in determining whether a neuron is firing or not.

Well, what if we want to shift our threshold? What if instead of zero, we determined that a neuron should fire if its input is greater than or equal to −1?

This is where bias comes into play.

Remember, we earlier said that the bias gets added to the weighted sum before being passed to the activation function. The value we assign to our bias is the opposite of this so called threshold value.

Continuing with our example, we want the threshold to move from 0 to −1, right? The bias will then be the opposite of −1, which is just 1.

The weighted sum of −0.35 plus our bias of 1 equals 0.65.

$(1\times-0.55)+(2\times0.10)+1=-0.35+1=0.65$.

Passing this value to relu, we can see that

$\text{relu}(0.65)=0.65$.

The neuron is now considered to be firing.

The model now has a bit of increased flexibility in fitting the data since it now has a broader range in what values it considers as being activated or not.

We could also do the same process in the opposite direction to narrow what output values from neurons that we consider as being activated. For example, if we determined that a neuron should be considered activated when its output is greater than or equal to five, then our bias would be minus five.

Conclusion

Now, we explicitly choose and set our bias in our example. In practice, this isn't the case. Just as we don't explicitly choose and control the weights in a network, we don't explicitly choose and control the biases either.

Remember, the biases are learnable parameters within the network, just like weights. After the biases are initialized, with random numbers, or zeros, or really any other value, they will be updated during training, allowing our model to learn when to activate and when not to activate each neuron.

Learnable Parameters

What Are Learnable Parameters?

Let's go into deep thought and see if we could possibly infer what a learnable parameter is solely based on its name.

Whew, that was tough. If you happened to just reach enlightenment, then you now know that a learnable parameter is, well, a parameter that is learned by the network during training.

Ok, all jokes aside, really, a learnable parameter is just that.

During the training process, we've discussed how stochastic gradient descent, or SGD, works to learn and optimize the weights and biases in a neural network. These weights and biases are indeed learnable parameters.

In fact, any parameters within our model which are learned during training via SGD are considered learnable parameters.

It's useful to note that these parameters are also referred to as trainable parameters, since they're optimized during the training process.

Calculating The Number Of Learnable Parameters

Alright, so we know what learnable parameters are. How can we calculate the number of these parameters within each layer, or even within the entire network?

To find this result, essentially we just count the number of parameters within each layer and then sum them up to get the total number of parameters within the full network.

What we need in order to calculate the number of parameters within an individual layer is:

- The number of inputs to that layer.
- The number of outputs from that layer.
- Whether or not the layer contains biases.

Note that we're talking about a fully connected network made up of standard dense layers. In another episode, we'll focus on how this is done for other networks, like CNNs.

Now, once we have the needed information, we multiply the input to a layer by the number of outputs from the layer. Another way to think about the outputs is by simply thinking about the number of nodes within the layer. The number of nodes is equal to the number of outputs.

> Number of nodes is equal to the number of outputs.

Now, multiplying the inputs by the outputs is going to give us the number of weights coming in to that layer.

Then, we just need to understand whether or not the layer contains biases for each node. If it does, then we simply add to the weights we just calculated, the number of biases. The number of biases will be equal to the number of nodes in the layer.

This will give us the number of learnable parameters within a given dense layer. We then do this same calculation for the remaining layers in the network and then sum all the results together to get the total number of learnable parameters within the entire network.

Learnable Parameters Example Calculation

Suppose we have a fully connected network with three layers:

- Input layer
- Hidden layer
- Output layer

We'll assume the following network architecture:

Layer	Number of Nodes
Input	2
Hidden	3
Output	2

Additionally, we're assuming our network contains biases. This means that there are bias terms within our hidden layer and our output layer.

Now, let's calculate the number of learnable parameters within each layer.

First things first, the input layer has no learnable parameters since the input layer is just made up of the input data, and the output from the layer is actually just going to be considered as input to the next layer.

Moving on, let's calculate the number of learnable parameters within the hidden layer.

Alright, we discussed earlier we first need the number of inputs to the layer. We have two inputs, which are the outputs from the two nodes in the input layer. Next, we need the number of outputs from this layer.

The number of outputs is the number of nodes. This means that we have three outputs. We multiply these two numbers together, which gives us six total weights.

Next, we add in our biases. The hidden layer has three nodes, which means it has three bias terms. Adding three to six, we see that this layer has nine total learnable parameters.

Moving on to the output layer, we do the same. How many inputs are coming in to the output layer? We have three coming from our three nodes in the hidden layer.

How many outputs are coming from the output layer? We have two, since that's the amount of nodes this layer has. How many biases? We have two, again since that's how many nodes we have in the layer.

Multiplying our input by our output, we have three times two, so that's six weights, plus two bias terms. That's eight learnable parameters for our output layer.

Adding eight to the nine parameters from our hidden layer, we see that the entire network contains seventeen total learnable parameters. During training, SGD will be learning and optimizing all seventeen of these weights.

Learnable Parameters In A CNN

What Are The Learnable Parameters In A CNN?

Alright, what are the learnable parameters in a CNN? Well, it turns out, that generally, they're the same parameters we saw in a standard fully connected network. That is, the weights and biases. But, we have to consider how, architecturally, the two types of networks are different, and how that's going to affect our calculation. Let's explore that now.

How The Number Of Learnable Parameters Is Calculated

So, just as with a standard network, with a CNN, we'll calculate the number of parameters per layer, and then we'll sum up the parameters in each layer to get the total amount of learnable parameters in the entire network.

```
// pseudocode
let sum = 0;
network.layers.forEach(function(layer) {
    sum += layer.getLearnableParameters().length;
}
```

For a dense layer, this is what we determined would tell us the number of learnable parameters:

```
inputs * outputs + biases
```

Now, let's consider what a convolutional layer has that a dense layer doesn't.

A convolutional layer has filters, also known as kernels. As the architects of our network, we determine how many filters are in a convolutional layer as well as how large these filters are, and we need to consider these things in our calculation.

With this in mind, we'll modify our formula for determining the number of learnable parameters in a convolutional layer.

So, what is the input going to be for a given convolutional layer? Well that's going to depend on what type of layer the previous layer was.

- If the previous layer was a dense layer, the input to the conv layer is just the number of nodes in the previous dense layer.
- If the previous layer was a convolutional layer, the input will be the number of filters from that previous convolutional layer.

Now, what's the output of a convolutional layer?

- With a dense layer, it was just the number of nodes.
- With a convolutional layer, the output will be the number of filters times the size of the filters.

We'll see this illustrated in just a sec. Finally, the number of biases, well that'll just be equal to the number of filters in the layer.

So overall, we have the same general setup for the number of learnable parameters in the layer being calculated as the number of inputs times the number of outputs plus the number of biases.

```
inputs * outputs + biases
```

Just with a convolutional layer, the inputs and outputs themselves are considering the number of filters and the size of the filters. Let's check ourselves by seeing this calculation in action with a simple CNN.

Calculating The Number Of Learnable Parameters In A CNN

Suppose we have a CNN made up of an input layer, two hidden convolutional layers, and a dense output layer.

- input layer
- hidden convolutional layer
- hidden convolutional layer
- dense output layer

Our input layer is made up of input data from images of size $20x20x3$,
where $20x20$ specifies the width and height of the images, and 3 specifies the number

of channels. The three channels indicate that our images are in RGB color scale, and these three channels will represent the input features in this layer.

Our first convolutional layer is made up of `2` filters of size `3x3`. Our second convolutional layer is made up of `3` filters of size `3x3`. And our output layer is a dense layer with `2` nodes.

We'll assume that the network contains bias terms and that we're using zero padding throughout the network to maintain the dimensions of the images. Check the zero padding chapter if you're unfamiliar with this concept.

- input layer - images of size `20x20x3`
- hidden convolutional layer - `2` filters of size `3x3`
- hidden convolutional layer - `3` filters of size `3x3`
- dense output layer - `2` nodes

Input Layer

Now, the same rule applies here for the input layer that we talked about last time. The input layer has no learnable parameters since it just contains the input data.

Conv Layer 1

Moving on to the first hidden convolutional layer, how many inputs do we have coming into this layer? We have `3` from our input layer. How many outputs? Well, let's see. Remember, the number of outputs is the number of filters times the filter size. So we have two filters, each of size `3x3`. So `2*3*3 = 18`. Multiplying our three inputs by our `18` outputs, we have `54` weights. Now how many biases? Just two, since the number of biases is equal to the number of filters. So that gives us `56` total learnable parameters in this layer.

Conv Layer 2

Now let's move to our next convolutional layer. How many inputs are coming in to this layer? We have two from the number of filters in the previous layer. How many outputs? Well, we have three filters, again of size `3x3`. So that's `3*3*3 = 27` outputs. Multiplying our two inputs by the `27` outputs, we have `54` weights in this layer. Adding three bias terms from the three filters, we have `57` learnable parameters in this layer .

Output Layer

Onto the output layer. How many inputs? We may think just three, right, since that's the number of filters in the last convolutional layer? But, that's not quite right. If you've followed the Keras series, you know that before passing output from a convolutional layer to a dense layer, that we have to flatten the output by multiplying the dimensions of the data from the conv layer by the number of filters in that layer. In our case, the data is image data.

Since we're assuming that this network uses zero padding, the dimensions of our images of size $20x20$ haven't changed by the time we get to this layer. So multiplying $20x20$ by the three filters gives us a total of 1200 inputs coming in to our output layer.

Now, since this output layer is a dense layer, the number of outputs is just equal to the number of nodes in this layer, so we have two outputs. Multiplying $1200*2$ gives us 2400 weights. Adding in our two biases from this layer, we have 2402 learnable parameters in this layer.

The Result

Summing up the parameters from all the layers gives us a total of 2515 learnable parameters within the entire network.

So we can see that the process for determining the number of learnable parameters in a convolutional network is generally the same as a standard fully connected network, but we have to do a little extra work by considering some extras, like the number of channels being used in image data, the number of filters, the filter sizes, and flattening convolutional output.

Regularization In A Neural Network

In our Previous Chapter on overfitting, we briefly introduced dropout and stated that it is a regularization technique. We hadn't yet discussed what regularization is, so let's do that now.

In general, *regularization* is a technique that helps reduce overfitting or reduce variance in our network by penalizing for complexity. The idea is that certain complexities in our model may make our model unlikely to generalize well, even though the model fits the training data.

> *Regularization* is a technique that helps reduce overfitting or reduce variance in our network by penalizing for complexity.

Given this, if we add regularization to our model, we're essentially trading in some of the ability of our model to fit the training data well for the ability to have the model generalize better to data it hasn't seen before.

To implement regularization is to simply add a term to our loss function that penalizes for large weights. We'll expand on this idea in just a moment.

L2 Regularization

The most common regularization technique is called *L2 regularization*. We know that regularization basically involves adding a term to our loss function that penalizes for large weights.

L2 Regularization Term

With L2 regularization, the term we're adding to the loss is the sum of the squared norms of the weight matrices

$$\sum_{j=1}^{n} \left\| w^{[j]} \right\|^2,$$

multiplied by a small constant $\frac{\lambda}{2m}$.

Norms Are Positive

If you're not familiar with norms in general, understand that a norm is just a function that assigns a strictly positive length or size for each vector in a vector space. The vector space we're working with here depends on the sizes of our weight matrices.

Rather than going on a linear algebra tangent about norms in this moment, we'll continue on with the general idea about regularization. Given that norms are a fundamental concept of linear algebra, there is a lot of information available on the web that explains norms in detail if you need to get a better grasp.

To over simplify, know for now that the norm of each of our weight matrices is just going to be a positive number.

Suppose that v is a vector in a vector space. The norm of v is denoted as ||v||, and it is required that

$$\|\boldsymbol{v}\| \geq \mathbf{0}.$$

Adding The Term To The Loss

Let's look at what L2 regularization looks like. We have $loss + \left(\sum_{j=1}^{n} \left\|w^{[j]}\right\|^2\right) \frac{\lambda}{2m}.$

The table below gives the definition for each variable in the expression above.

Variable	Definition
n	Number of layers
$w^{[j]}$	Weight matrix for the j^{th} layer
m	Number of inputs
λ	Regularization parameter

The term λ is called the regularization parameter, and this is another hyperparameter that we'll have to choose and then test and tune in order to choose the correct number for our specific model.

To summarize, we now know that regularization is just a technique that penalizes for relatively large weights in our model, and behind the scenes, the implementation of regularization is just the addition of a term to our existing loss function.

Impact Of Regularization

So why does regularization help?

Well, using L2 regularization as an example, if we were to set λ to be large, then it would incentivize the model to set the weights close to zero because the objective of SGD is to minimize the loss function. Remember our original loss function is now being summed with the sum of the squared matrix norms, $\sum_{j=1}^{n} \left\| w^{[j]} \right\|^2$,

which is multiplied by $\frac{\lambda}{2m}$.

If λ is large, then this term, $\frac{\lambda}{2m}$ will continue to stay relatively large, and if we're multiplying that by the sum of the squared norms, then the product may be relatively large depending on how large our weights are. This means that our model is incentivized to make the weights small so that the value of this entire function stays relatively small in order to minimize loss.

Intuitively, we could think that maybe this technique will set the weights so close to zero, that it could basically zero-out or reduce the impact of some of our layers. If that's the case, then it would conceptually simplify our model, making our model less complex, which may in turn reduce variance and overfitting.

Batch Size In Artificial Neural Networks

Introducing Batch Size

Put simply, the *batch size* is the number of samples that will be passed through to the network at one time. Note that a batch is also commonly referred to as a mini-batch.

> The *batch size* is the number of samples that are passed to the network at once.

Now, recall that an *epoch* is one single pass over the entire training set to the network. The batch size and an epoch are not the same thing. Let's illustrate this with an example.

Batches In An Epoch

Let's say we have 1000 images of dogs that we want to train our network on in order to identify different breeds of dogs. Now, let's say we specify our batch size to be 10. This means that 10 images of dogs will be passed as a group, or as a batch, at one time to the network.

Given that a single epoch is one single pass of all the data through the network, it will take 100 batches to make up full epoch. We have 1000 images divided by a batch size of 10, which equals 100 total batches.

> batches in epoch = training set size / batch_size

Ok, we have the idea of batch size down now, but what's the point? Why not just pass each data element one-by-one to our model rather than grouping the data in batches?

Why Use Batches?

Well, for one, generally the larger the batch size, the quicker our model will complete each epoch during training. This is because, depending on our computational resources, our machine may be able to process much more than one single sample at a time.

The trade-off, however, is that even if our machine can handle very large batches, the quality of the model may degrade as we set our batch larger and may ultimately cause the model to be unable to generalize well on data it hasn't seen before.

In general, the batch size is another one of the *hyperparameters* that we must test and tune based on how our specific model is performing during training. This parameter will also have to be tested in regards to how our machine is performing in terms of its resource utilization when using different batch sizes.

For example, if we were to set our batch size to a relatively high number, say `100`, then our machine may not have enough computational power to process all `100` images in parallel, and this would suggest that we need to lower our batch size.

Mini-Batch Gradient Descent

Additionally, note if using *mini-batch gradient descent*, which is normally the type of gradient descent algorithm used by most neural network APIs like Keras by default, the gradient update will occur on a per-batch basis. The size of these batches is determined by the batch size.

This is in contrast to *stochastic gradient descent*, which implements gradient updates per sample, and *batch gradient descent*, which implements gradient updates per epoch.

Alright, we should now have a general idea about what batch size is. Let's see how we specify this parameter in code now using Keras.

Working With Batch Size In Keras

We'll be working with the same model we've used in the last several posts. This is just an arbitrary `Sequential` model.

```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    Dense(units=2, activation='sigmoid')
])
```

Let's focus our attention on where we call `model.fit()`. We know this is the function we call to train our model, and we saw this in action in our previous post on how an artificial neural network learns.

```
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    validation_data=valid_set,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

This `fit()` function accepts a parameter called `batch_size`. This is where we specify our `batch_size` for training. In this example, we've just arbitrarily set the value to `10`.

Now, during the training of this model, we'll be passing in `10` samples at a time until we eventually pass in all the training data to complete one single epoch. Then, we'll start the same process over again to complete the next epoch.

That's really all there is to it for specifying the batch size for training a model in Keras!

Fine-Tuning Neural Networks

Introducing Fine-Tuning And Transfer Learning

Fine-tuning is very closely linked with the term *transfer learning*. Transfer learning occurs when we use knowledge that was gained from solving one problem and apply it to a new but related problem.

Transfer learning occurs when we use knowledge that was gained from solving one problem and apply it to a new but related problem.

For example, knowledge gained from learning to recognize cars could be applied in a problem of recognizing trucks.

Fine-tuning is a way of applying or utilizing transfer learning. Specifically, fine-tuning is a process that takes a model that has already been trained for one given task and then tunes or tweaks the model to make it perform a second similar task.

Why Use Fine-Tuning?

Assuming the original task is similar to the new task, using an artificial neural network that has already been designed and trained allows us to take advantage of what the model has already learned without having to develop it from scratch.

When building a model from scratch, we usually must try many approaches through trial-and-error.

For example, we have to choose how many layers we're using, what types of layers we're using, what order to put the layers in, how many nodes to include in each layer, decide how much regularization to use, what to set our learning rate as, etc.

- Number of layers
- Types of layers
- Order of layers
- Number of nodes in each layer
- How much regularization to use
- Learning rate

Building and validating our model can be a huge task in its own right, depending on what data we're training it on.

This is what makes the fine-tuning approach so attractive. If we can find a trained model that already does one task well, and that task is similar to ours in at least some remote way, then we can take advantage of everything the model has already learned and apply it to our specific task.

Now, of course, if the two tasks are different, then there will be some information that the model has learned that may not apply to our new task, or there may be new information that the model needs to learn from the data regarding the new task that wasn't learned from the previous task.

For example, a model trained on cars is not going to have ever seen a truck bed, so this feature is something new the model would have to learn about. However, think about everything our model for recognizing trucks could use from the model that was originally trained on cars.

This already trained model has learned to understand edges and shapes and textures and more objectively, head lights, door handles, windshields, tires, etc. All of these learned features are definitely things we could benefit from in our new model for classifying trucks.

So this sounds fantastic, right, but how do we actually technically implement this?

How To Fine-Tune

Going back to the example we just mentioned, if we have a model that has already been trained to recognize cars and we want to fine-tune this model to recognize trucks, we can first import our original model that was used on the cars problem.

For simplicity purposes, let's say we remove the last layer of this model. The last layer would have previously been classifying whether an image was a car or not. After removing this, we want to add a new layer back that's purpose is to classify whether an image is a truck or not.

In some problems, we may want to remove more than just the last single layer, and we may want to add more than just one layer. This will depend on how similar the task is for each of the models.

Layers at the end of our model may have learned features that are very specific to the original task, where as layers at the start of the model usually learn more general features like edges, shapes, and textures.

After we've modified the structure of the existing model, we then want to freeze the layers in our new model that came from the original model.

Freezing Weights

By *freezing*, we mean that we don't want the weights for these layers to update whenever we train the model on our new data for our new task. We want to keep all of these weights the same as they were after being trained on the original task. We only want the weights in our new or modified layers to be updating.

After we do this, all that's left is just to train the model on our new data. Again, during this training process, the weights from all the layers we kept from our original model will stay the same, and only the weights in our new layers will be updating.

Batch Normalization ("Batch Norm")

Normalization Techniques

Before getting to the details about batch normalization, let's quickly first discuss regular normalization techniques.

$$z = \frac{x - mean}{std}$$

Generally speaking, when training a neural network, we want to normalize or standardize our data in some way ahead of time as part of the pre-processing step. This is the step where we prepare our data to get it ready for training.

Normalization and standardization have the same objective of transforming the data to put all the data points on the same scale.

A typical normalization process consists of scaling numerical data down to be on a scale from zero to one, and a typical standardization process consists of subtracting the mean of the dataset from each data point, and then dividing that difference by the data set's standard deviation.

This forces the standardized data to take on a mean of zero and a standard deviation of one. In practice, this standardization process is often just referred to as normalization as well.

Why Use Normalization Techniques?

In general, this all boils down to putting our data on some type of known or standard scale. Why do we do this?

Well, if we didn't normalize our data in some way, we can imagine that we may have some numerical data points in our data set that might be very high, and other that might be very low.

For example, suppose we have data on the number of miles individuals have driven a car over the last 5 years. We may have someone who has driven $100{,}000$ miles total, and we may have someone else who's only driven 1000 miles total. This data has a relatively wide range and isn't necessarily on the same scale.

Additionally, each one of the features for each of our samples could vary widely as well. If we have one feature which corresponds to an individual's age and the other feature corresponds to the number of miles that individual has driven a car over the last five years, then, again, we can see that these two pieces of data, age and miles driven, will not be on the same scale.

The larger data points in these non-normalized data sets can cause instability in neural networks because the relatively large inputs can cascade down through the layers in the network, which may cause imbalanced gradients, which may therefore cause the famous exploding gradient problem. This topic is covered in it’s chapter.

For now, understand that this imbalanced, non-normalized data may cause problems with our network that make it drastically harder to train. Additionally, non-normalized data can significantly decrease our training speed.

When we normalize our inputs, however, we put all of our data on the same scale, in attempts to increase training speed as well as avoid the problem we just discussed because we won't have this relatively wide range between data points.

This is good, but there is another problem that can arise even with normalized data. From our Previous chapter on how a neural network learns, we know how the weights in our model become updated over each epoch during training via the process of stochastic gradient descent.

Weights That Tip The Scale

What if, during training, one of the weights ends up becoming drastically larger than the other weights?

Well, this large weight will then cause the output from its corresponding neuron to be extremely large, and this imbalance will, again, continue to cascade through the network, causing instability. This is where batch normalization comes into play.

Applying Batch Norm To A Layer

Batch norm is applied to layers that we choose within our network.

When applying batch norm to a layer, the first thing batch norm does is normalize the output from the activation function. Recall from our chapter on activation functions that the output from a layer is passed to an activation function, which transforms the output in some way depending on the function itself, before being passed to the next layer as input.

After normalizing the output from the activation function, batch norm multiplies this normalized output by some arbitrary parameter and then adds another arbitrary parameter to this resulting product.

Step	Expression	Description
1	$z = \frac{x - mean}{std}$	Normalize output x from activation function.
2	$z * g$	Multiply normalized output z by arbitrary parameter g.
3	$(z * g) + b$	Add arbitrary parameter b to resulting product $(z * g)$.

Trainable Parameters

This calculation with the two arbitrary parameters sets a new standard deviation and mean for the data. The two arbitrarily set parameters, g and b are trainable, meaning that they will be become learned and optimized during the training process.

Parameter	Trainable
g	Yes
b	Yes

This process makes it so that the weights within the network don't become imbalanced with extremely high or low values since the normalization is included in the gradient process.

This addition of batch norm to our model can greatly increase the speed in which training occurs and reduce the ability of outlying large weights to over-influence the training process.

When we spoke about normalizing our input data in the pre-processing step before training occurs, we understand that this normalization happens to the data before being passed to the input layer.

With batch norm, we can normalize the output data from the activation functions for individual layers within our model as well. This means we have normalized data coming in, and we also have normalized data within the model.

Normalizing Per Batch

Everything we just mentioned about the batch normalization process occurs on a per-batch basis, hence the name *batch norm*.

These batches are determined by the batch size we set when we train our model. If we're not yet familiar with training batches or batch size, check out this post on the topic.

Now that we have an understanding of batch norm, let's look at how we can add batch norm to a model in code using Keras.

Working With Code In Keras

Here, we've just copied the code for a model that we've built in a previous chapter.

```
model = Sequential([
    Dense(units=16, input_shape=(1,5), activation='relu'),
    Dense(units=32, activation='relu'),
    BatchNormalization(axis=1),
    Dense(units=2, activation='softmax')
])
```

We have a model with two hidden layers with `16` and `32` nodes respectively, both using `relu()` as their activation functions, and an output layer with two output categories using the `softmax()` activation function.

The only difference here is the line between the last hidden layer and the output layer.

```
BatchNormalization(axis=1)
```

This is how we specify batch normalization in Keras. Following the layer for which we want the activation output normalized, we specify a `BatchNormalization` object. To do this, we need to import `BatchNormalization` from Keras, as shown below.

```
from keras.models import Sequential
from keras.layers import Dense, Activation, BatchNormalization
```

The only parameter that we're specifying for `BatchNormalization` is the `axis` parameter, and that is just to specify the axis from the data that should be normalized, which is typically the features axis.

There are several other parameters that we can optionally specify, including two called `beta_initializer` and `gamma_initializer`. These are the initializers for the arbitrarily set parameters that we mentioned when we were describing how batch norm works.

These are set by default to `0` and `1` by Keras, but we can optionally change these, along with several other optionally specified parameters.