Elizabeta Ozretić

0011162671

0955267806

Table of Contents

[WHAT IS DEEP LEARNING? 1](#_Toc64022500)

[WHAT DOES *DEEP* MEAN IN DEEP LEARNING? 2](#_Toc64022501)

[WHAT IS AN ARTIFICIAL NEURAL NETWORK? 3](#_Toc64022502)

[VISUALIZING AN ARTIFICIAL NEURAL NETWORK 4](#_Toc64022503)

[WHY HAVE DIFFERENT TYPES OF LAYERS 5](#_Toc64022504)

[EXAMPLE ARTIFICIAL NEURAL NETWORK 5](#_Toc64022505)

[LAYER WEIGHTS 6](#_Toc64022506)

[FORWARD PASS THROUGH A NEURAL NETWORK 6](#_Toc64022507)

[FINDING THE OPTIMAL WEIGHTS 7](#_Toc64022508)

[ACTIVATION FUNCTIONS IN A NEURAL NETWORK 7](#_Toc64022509)

[WHAT IS AN ACTIVATION FUNCTION? 7](#_Toc64022510)

[WHAT DO ACTIVATION FUNCTIONS DO? 7](#_Toc64022511)

[TRAINING AN ARTIFICIAL NEURAL NETWORK 10](#_Toc64022512)

[WHAT IS TRAINING? 10](#_Toc64022513)

[HOW A NEURAL NETWORK LEARNS EXPLAINED 11](#_Toc64022514)

[LEARNING IN ARTIFICIAL NEURAL NETWORKS 11](#_Toc64022515)

[WHAT DOES IT MEAN TO LEARN? 12](#_Toc64022516)

[LOSS IN A NEURAL NETWORK EXPLAINED 13](#_Toc64022517)

[LOSS FUNCTIONS IN NEURAL NETWORKS 13](#_Toc64022518)

[MEAN SQUARED ERROR (MSE) 14](#_Toc64022519)

[LEARNING RATE IN A NEURAL NETWORK EXPLAINED 15](#_Toc64022520)

[LEARNING RATES AND NEURAL NETWORKS 15](#_Toc64022521)

[INTRODUCING THE LEARNING RATE 15](#_Toc64022522)

[UPDATING THE NETWORKS WEIGHTS 16](#_Toc64022523)

[TRAIN, TEST, AND VALIDATION SETS EXPLAINED 17](#_Toc64022524)

[DATASETS FOR DEEP LEARNING 17](#_Toc64022525)

[DEEP LEARNING DATASETS IN SUMMARY 19](#_Toc64022526)

[PREDICTING WITH A NEURAL NETWORK EXPLAINED 19](#_Toc64022527)

[PREDICTING WITH A NEURAL NETWORK 19](#_Toc64022528)

[PASSING SAMPLES WITH NO LABELS 19](#_Toc64022529)

[OVERFITTING IN A NEURAL NETWORK 20](#_Toc64022530)

[HOW TO SPOT OVERFITTING 20](#_Toc64022531)

[REDUCING OVERFITTING 21](#_Toc64022532)

[UNDERFITTING IN A NEURAL NETWORK 22](#_Toc64022533)

[REDUCING OVERFITTING 22](#_Toc64022534)

[SUPERVISED LEARNING FOR MACHINE LEARNING 23](#_Toc64022535)

[DATA AUGMENTATION FRO MACHINE LEARNING 24](#_Toc64022536)

[WHY USE DATA AUGMENTATION? 24](#_Toc64022537)

[ONE-HOT ENCODINGS FOR MACHINE LEARNING 25](#_Toc64022538)

[LABELS 25](#_Toc64022539)

[HOT AND COLD VALUES 25](#_Toc64022540)

[ONE-HOT ENCODINGS FOR MULTIPLE CATEGORIES 26](#_Toc64022541)

[DEEP LEARNING WITH CONVOLUTIONAL NEURAL NETWORKS 27](#_Toc64022542)

[WHAT IS A CNN? 27](#_Toc64022543)

[FILTERS AND CONVOLUTION OPERATIONS 27](#_Toc64022544)

[INPUT AND OUTPUT CHANNELS 31](#_Toc64022545)

[VISUALIZING CONVOLUTIONAL FILTERS 35](#_Toc64022546)

[KERAS AND THE CODE 35](#_Toc64022547)

[GENERATED CNN LAYER VISUALIZATIONS 36](#_Toc64022548)

[ZERO PADDING IN CONVOLUTIONAL NEURAL NETWORKS 38](#_Toc64022549)

[CONVOLUTIONS REDUCE CHANNEL DIMENSIONS 38](#_Toc64022550)

[ISSUES WITH REDUCING THE DIMENSIONS 38](#_Toc64022551)

[ZERO PADDING 39](#_Toc64022552)

[MAX POOLING IN CONVOLUTIONAL NEURAL NETWORKS 40](#_Toc64022553)

[WHY USE MAXPOOLING 43](#_Toc64022554)

[BACKPROPAGATION IN NEURAL NETWORKS 44](#_Toc64022555)

[STOCHASTIC GRADIENT DESCENT (SGD) REVIEW 44](#_Toc64022556)

[FORWARD PROPAGATION 45](#_Toc64022557)

[BACKPROPAGATION INTUITION 47](#_Toc64022558)

[VANISHING AND EXPLODING GRADIENT 49](#_Toc64022559)

[EXPLODING GRADIENT 51](#_Toc64022560)

[WEIGHT INITIALIZATION EXPLAINED 52](#_Toc64022561)

[HOW ARE WEIGHTS INITIALIZED? 52](#_Toc64022562)

[XAVIER INITIALIZATION 54](#_Toc64022563)

[BIAS IN AN ARTIFICIAL NETWORK 55](#_Toc64022564)

[UNDERSTANDING BIAS INSIDE NEURAL NETWORKS 55](#_Toc64022565)

[LEARNABLE PARAMETERS IN A NEURAL NETWORK 57](#_Toc64022566)

[LEARNABLE PARAMETERS 57](#_Toc64022567)

[LEARNABLE PARAMTERS IN A CNN 60](#_Toc64022568)

[WHAT ARE LEARNABLE PARAMETERS IN A CNN 60](#_Toc64022569)

[HOW THE NUMBER OF LEARNABLE PARAMETERS IS CALCULATED 60](#_Toc64022570)

[CALCULATING THE NUMBER OF LEARNABLE PARAMETERS IN A CNN 61](#_Toc64022571)

[REGULARIZATION IN A NEURAL NETWORK 63](#_Toc64022572)

[L2 REGULARIZATION 64](#_Toc64022573)

[IMPACT OF REGULARIZATION 65](#_Toc64022574)

[BATCH SIZE IN ARTIFICIAL NEURAL NETWORK 65](#_Toc64022575)

[INTRODUCING BATCH SIZE 65](#_Toc64022576)

[FINE TUNING NEURAL NETWORKS 67](#_Toc64022577)

[HOW TO FINE-TUNE 68](#_Toc64022578)

[BATCH NORMALIZATION 69](#_Toc64022579)

[NORMALIZATION TECHNIQUES 69](#_Toc64022580)

[USE OF NORMALIZATION TECHNIQUES 69](#_Toc64022581)

[APPLYING BATCH NORM TO A LAYER 71](#_Toc64022582)

[elizabetaozretic@gmail.com](mailto:elizabetaozretic@gmail.com)

utorak 8, 12, 16

srijeda 16

četvrtak 8, 12, 16

Veronika Ozretić

00231152535

0912530796

vozret00@fesb.hr

DEEPLIZARD

## WHAT IS DEEP LEARNING?

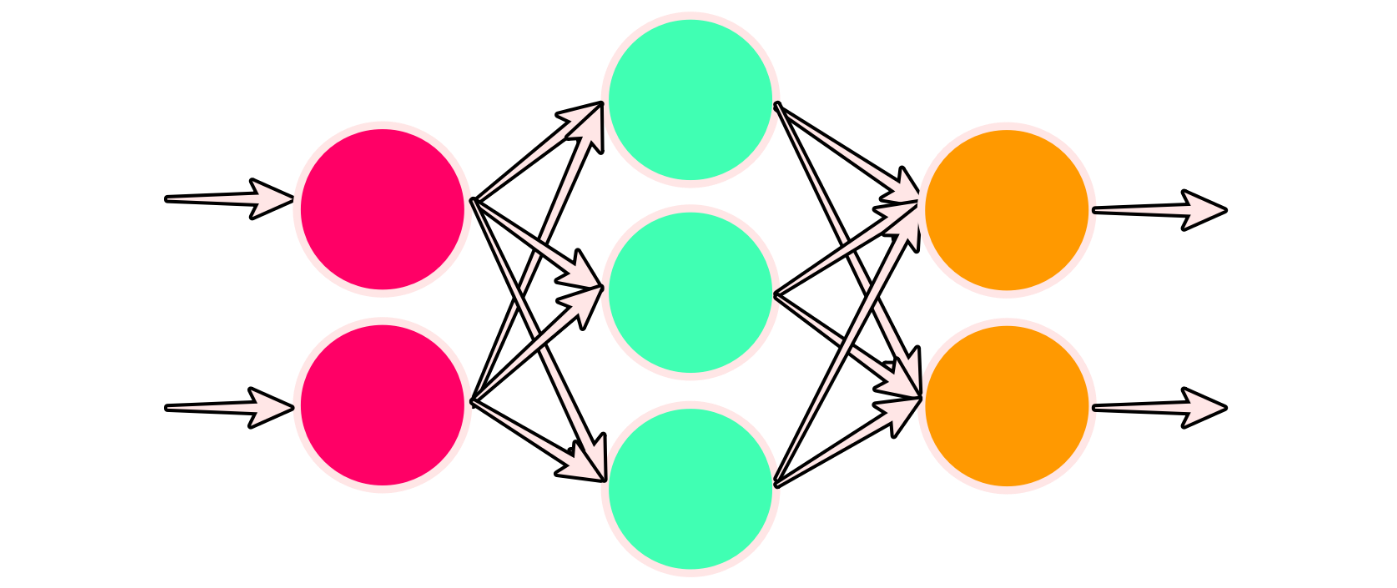
Deep learning is a sub-field of machine learning that uses algorithms inspired by the structure and function of the brain's neural networks.

With deep learning, we're still talking about algorithms that *learn* from data and the algorithms or models that do this learning are based loosely on the structure and function of the brain's neural networks.

The neural networks that we use in deep learning aren't actual biological neural networks though. They simply share some characteristics with biological neural networks, and for this reason, 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

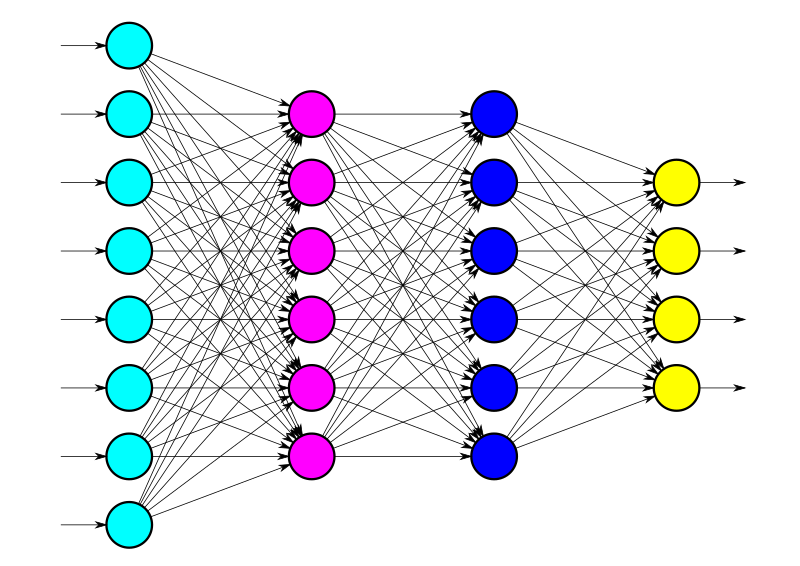


### WHAT DOES *DEEP* MEAN IN DEEP LEARNING?

Deep learning uses a specific type of ANN that we call a deep net or deep artificial neural network.

For now, here is what you need to know:

1. ANNs are built using what we call neurons.
2. Neurons in an ANN are organized into what we call layers.
3. Layers *within* an ANN (all but the input and output layers) are called hidden layers.
4. If an ANN has more than one hidden layer, the ANN is said to be a deep ANN.



In summary, deep learning uses ANNs that have multiple hidden layers.

## WHAT IS AN ARTIFICIAL NEURAL NETWORK?

We defined deep learning as a sub-field of machine learning that uses algorithms inspired by the structure and function of the brain's neural networks.

For this reason, the models used in deep learning are called artificial neural networks (ANNs).

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:

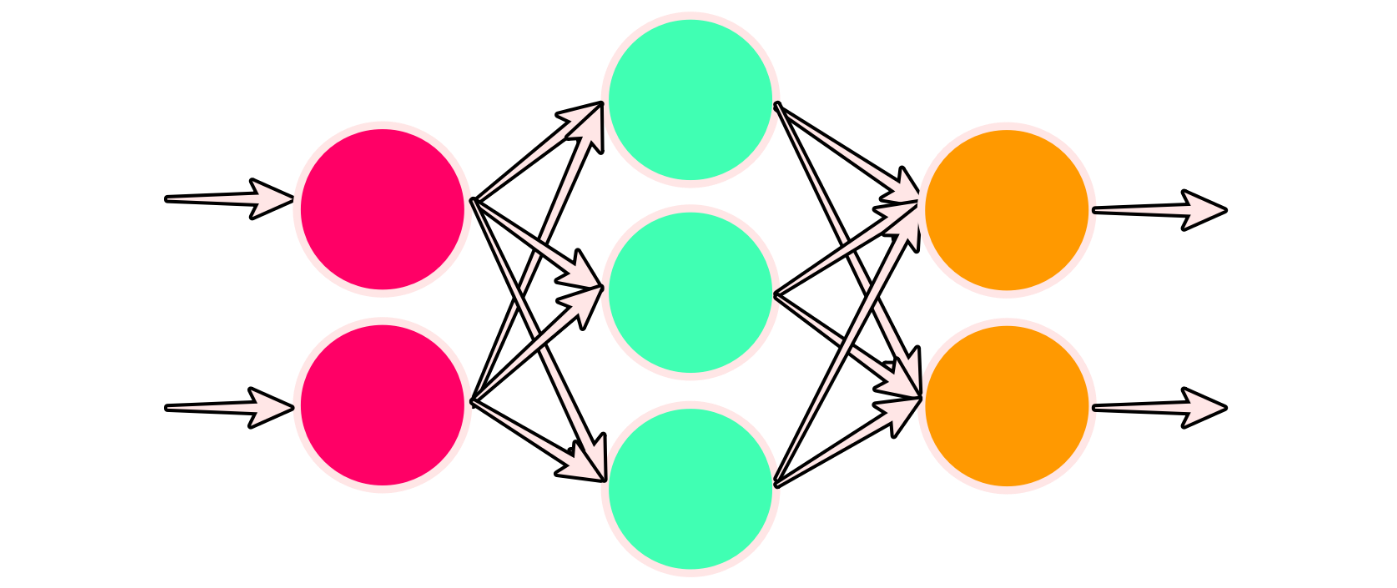
1. Input layer
2. Hidden layers
3. Output layer

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



This ANN has three layers total. The layer on the left is the input layer. The layer on the right is the output layer, and the layer in the middle is the hidden layer. Remember that each layer is comprised of neurons or nodes. Here, the nodes are depicted with the circles, so let’s consider how many nodes are in each layer of this network.

Number of nodes in each layer:

1. Input layer (left): 2 nodes
2. Hidden layer (middle): 3 nodes
3. Output layer (right): 2 nodes

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.

In the [last post](https://deeplizard.com/learn/video/hfK_dvC-avg), we saw how the neurons in an ANN are organized into layers. The examples we looked at showed the use of dense layers, which are also known as fully connected layers.

There are, however, different types of layers. Some examples include:

* Dense (or fully connected) layers
* Convolutional layers
* Pooling layers
* Recurrent layers
* Normalization layers

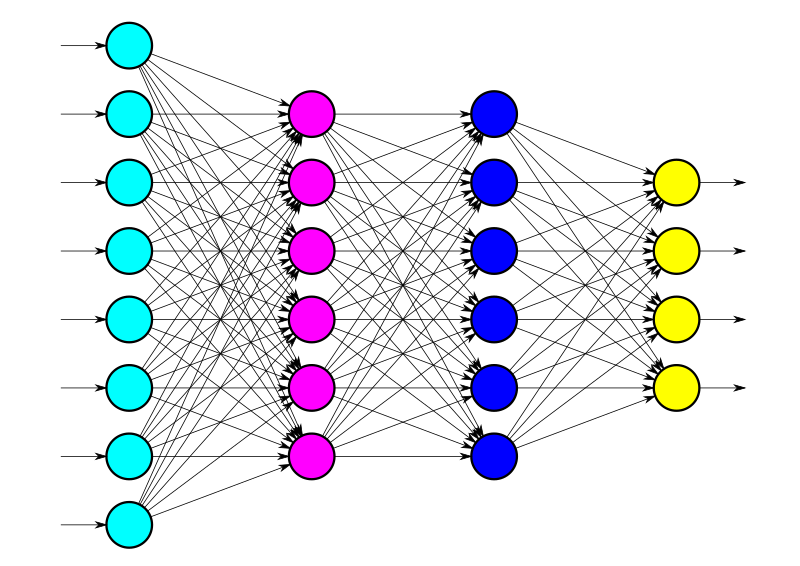
### WHY HAVE DIFFERENT TYPES OF LAYERS

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.

### EXAMPLE ARTIFICIAL NEURAL NETWORK

Let’s consider the following example ANN:



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.

We can see that each of the eight input nodes are connected to every node in the next layer.

Each connection between the first and second layers transfers the output from the previous node to the input of the receiving node (left to right). The two layers in the middle that have six nodes each are hidden layers simply because they are positioned between the input and output layers.

### 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.

## ACTIVATION FUNCTIONS IN A NEURAL NETWORK

In this post, we’ll be discussing what an activation function is and how we use these functions in neural networks.

### WHAT IS AN ACTIVATION FUNCTION?

Let's give a definition for an activation function:

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

This makes sense given the illustration we saw in the previous [post on layers](https://deeplizard.com/learn/video/FK77zZxaBoI). 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.

### WHAT DO ACTIVATION FUNCTIONS DO?

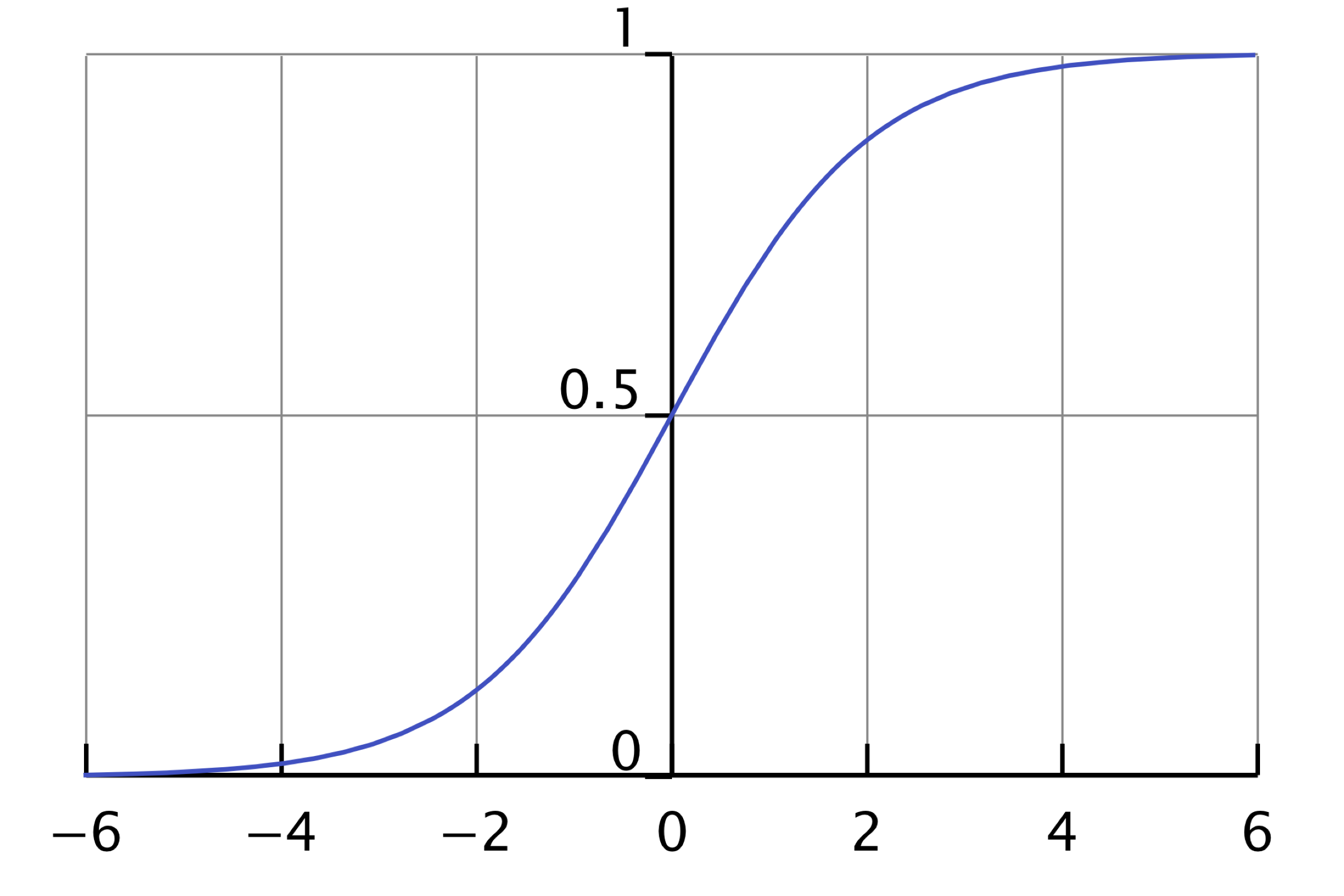
To explain this, let’s first look at some example activation functions.

**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



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

**ACTIVATION FUNCTION INTUITION**

Well, an activation function is biologically inspired by activity in our brains where different neurons fire (or are activated) by different stimuli.

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.

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.

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.

The idea here is, the more positive the neuron is, the more activated it is. Now, we’ve only talked about two activation functions here, sigmoid and relu, but there are other types of activation functions that do different types of transformations to their inputs.

**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:

and

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**

Let’s prove that the relu activation function is non-linear. To do this, we will show that relu fails to be linear.

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

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

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

and that

This allows us to conclude that

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

## TRAINING AN ARTIFICIAL NEURAL NETWORK

In this post, we’ll discuss what it means to train an artificial neural network. In a [previous post](https://deeplizard.com/learn/video/FK77zZxaBoI), we went over the basic architecture of a general artificial neural network. Now, after configuring the architecture of the model, the next step is to train it.

### 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 [post about layers](https://deeplizard.com/learn/video/FK77zZxaBoI). 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.

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*. So, through this process that’s occurring with SGD iteratively, the model is able to learn from the data.

## HOW A NEURAL NETWORK LEARNS EXPLAINED

### LEARNING IN ARTIFICIAL NEURAL NETWORKS

In this post, we’ll investigate what it means for an artificial neural network to learn.

In a [previous post](https://deeplizard.com/learn/video/sZAlS3_dnk0), we learned about the training process and saw that each data point used for training is passed through the network. This pass through the network from input to output is called a *forward pass*, and the resulting output depends on the weights at each connection inside the network.

Once all of the data points in our dataset have been passed through the network, we say that an epoch is complete.

An epoch refers to a single pass of the entire dataset to the network during training.

Note that many epochs occur throughout the training process as the model learns.

### 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.*

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.

**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.

## LOSS IN A NEURAL NETWORK EXPLAINED

### LOSS FUNCTIONS IN NEURAL NETWORKS

In this post, we’ll be discussing what a loss function is and how it’s used in an artificial neural network.

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*.

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.

MSE(input) = (output - label)(output - 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 (batch\_size=1), 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.

The currently available [loss functions for Keras](https://keras.io/losses/) 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

## LEARNING RATE IN A NEURAL NETWORK EXPLAINED

### LEARNING RATES AND NEURAL NETWORKS

In this post, we’ll be discussing the learning rate, and we’ll see how it’s used when we train a neural network.

In our [previous post](https://deeplizard.com/learn/video/_N5kpSMDf4o) on what it means for an artificial neural network to learn, we briefly mentioned the learning rate as a number that we multiply our resulting gradient by.

### INTRODUCING THE LEARNING RATE

We know that the objective during [training](https://deeplizard.com/learn/video/sZAlS3_dnk0) 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.

gradients \* 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 NETWORKS 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.

## TRAIN, TEST, AND VALIDATION SETS EXPLAINED

### DATASETS FOR DEEP LEARNING

In this post, we’ll discuss the different datasets used during training and testing a neural network.

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.

**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.

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. 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.

**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

| Deep Learning Datasets | | |
| --- | --- | --- |
| **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. |

The main reason for having three separate datasets is to ensure that the model is able to generalize by predicting accurately on unseen data. When the model is failing to generalize, we are usually in a situation of overfitting or underfitting.

## PREDICTING WITH A NEURAL NETWORK EXPLAINED

### PREDICTING WITH A NEURAL NETWORK

In an [earlier post](https://deeplizard.com/learn/video/sZAlS3_dnk0), we discussed what it means to train a neural network. After this training has completed, if we’re happy with the metrics that the model gave us for our training and validation data, then the next step would be to have our model predict on the data in our test set.

Unlike the train and validation data that get passed to the model with their respective labels, when we pass our test data to the model, we do *not* pass the corresponding labels. So, the model is not aware of the labels for the test set at all.

### 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.

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.

## 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.

### REDUCING OVERFITTING

**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.

**DATA AUGMENTATION**

Another technique we can deploy to reduce overfitting is to use [data augmentation](https://deeplizard.com/learn/video/rfM4DaLTkMs). 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:

* Cropping
* Rotating
* Flipping
* Zooming

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.

**REDUCING 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.

**DROPOUT**

The general idea behind dropout is that, if you add it to a model, 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.

## UNDERFITTING IN A NEURAL NETWORK

In this post, we’ll discuss what it means when a model is said to be underfitting. We’ll also cover some techniques we can use to try to reduce or avoid underfitting when it happens.

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.

We can tell that a model is underfitting when the metrics given for the training data are poor, meaning that the training accuracy of the model is low and/or the training loss is high.

If the model is unable to classify data it was trained on, it’s likely not going to do well at predicting on data that it hasn’t seen before.

### REDUCING OVERFITTING

**INCREASE THE COMPLEXITY OF THE MODEL**

One thing we can do is increase the complexity of our 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.

**ADD MORE FEATURES TO THE INPUT SAMPLES**

Another technique we can use to reduce underfitting is to 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](https://deeplizard.com/learn/video/DoJiau6mPFc), 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.

**REDUCE DROPOUT**

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

**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](https://deeplizard.com/learn/video/Zi-0rlM4RDs) 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.

Based on what we saw in [our post on training](https://deeplizard.com/learn/video/sZAlS3_dnk0), we know that the model will then classify the output of this input, and then determine the error for that input by looking at the difference between the value it predicted and the actual label for the input.

**LABELS ARE NUMERIC**

To do this, the labels need to be encoded into something [numeric](https://deeplizard.com/learn/video/YDDqe60omL0). (0, 1, 2…)

After this, we go through this process of determining the error or [loss](https://deeplizard.com/learn/video/Skc8nqJirJg) 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](https://deeplizard.com/learn/video/lEfrr0Yr684). We could also use another technique called [semi-supervised learning](https://deeplizard.com/learn/video/b-yhKUINb7o).

## DATA AUGMENTATION FRO MACHINE LEARNING

In this post, we’ll be discussing *data augmentation* and under what circumstances we may want to use it.

Data augmentation occurs when we create new data based on modifications of our existing data. Essentially, we’re creating new, augmented data by making reasonable modifications to data in our training set.

For example, we could augment image data by flipping the images, either horizontally or vertically. We could rotate the images, zoom in or out, crop, or even vary the color of the images. All of these are common data augmentation techniques.

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

## WHY USE DATA AUGMENTATION?

Well, we may just want or need to add more data to our training set. For example, say we have a relatively small amount of samples to include in our training set, and it’s difficult to get more. Then we could create new data from our existing data set using data augmentation to create more samples.

**REDUCE OVERFITTING**

Additionally, we may want to use data augmentation to reduce overfitting.

If our model is overfitting, one technique to reduce it to add more data to the training set. Given the first point we just made a moment ago, we can easily create more data using data augmentation if we don’t have access to additional data.

## ONE-HOT ENCODINGS FOR MACHINE LEARNING

In this post, we’re going to discuss *one-hot encoding*, and how we make use of it in machine learning.

### LABELS

We know that when we’re training a neural network via [supervised learning](https://deeplizard.com/learn/video/Quh6x4kG6VY), 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](https://deeplizard.com/learn/video/M8CbBfx5228), 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](https://deeplizard.com/learn/video/YDDqe60omL0) values is called *one-hot encoding*.

One-hot encodings transform our categorical labels into vectors of 0s and 1s. 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.

### 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].

|  |  |
| --- | --- |
| **Label** | **Vector** |
| Cat | [1,0,0] |
| Dog | [0,1,0] |
| Lizard | [0,0,1] |

## DEEP LEARNING WITH 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.

Although image analysis has been the most wide spread use of CNNS, they can also be used for other data analysis or classification as well.

### WHAT IS A CNN?

Most generally, we can think of a CNN as an [artificial neural network](https://deeplizard.com/learn/video/hfK_dvC-avg) 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!

**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](https://deeplizard.com/learn/video/k6ZF1TSniYk).

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](https://en.wikipedia.org/wiki/Cross-correlation).

### 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*.

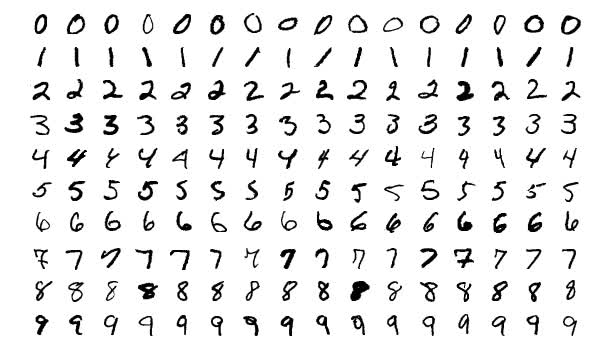
Aside from edges, some filters may detect corners. Some may detect circles. Others, squares. Now these simple, and kind of geometric, filters are what we’d see at the start of a convolutional neural network.

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.

**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.



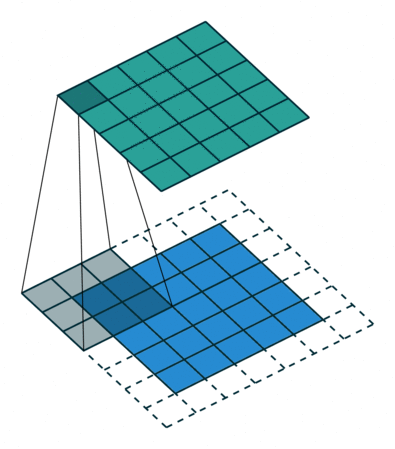
Let’s now assume that the first hidden layer in our model is a convolutional layer. As mentioned earlier, when adding a convolutional layer to a model, we also have to specify how many filters we want the layer to have.

The number of filters determines the number of output channels.

A filter can technically just be thought of as a relatively small matrix ( [tensor](https://deeplizard.com/learn/video/Csa5R12jYRg)), 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**



This image showcases the [convolution](https://deeplizard.com/resource/pavq7noze2) 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 x 3 convolutional filter
* Green (top) - Output channel

For each position on the blue input channel, the 3 x 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 x 3 set of pixels of the input itself until it’s slid over every 3 x 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 x 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 x 3 block of pixels from the input. When the filter lands on its first 3 x 3 block of pixels, the dot product of the filter itself with the 3 x 3 block of pixels from the input will be computed and stored. This will occur for each 3 x 3 block of pixels that the filter convolves.

For example, we take the dot product of the filter with the first 3 x 3 block of pixels, and then that result is stored in the output channel. Then, the filter slides to the next 3 x 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](https://deeplizard.com/learn/video/k6ZF1TSniYk).

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. 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 x 3 matrices A and B as follows.

Then, we sum the pairwise products like this:

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:

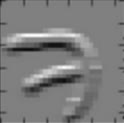
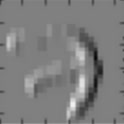
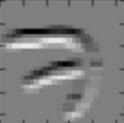
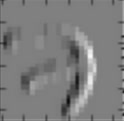


Let’s suppose that we have four 3 x 3 filters for our first convolutional layer, and these filters are filled with the values you see below. These values can be represented visually by having -1s correspond to black, 1s correspond to white, and 0s correspond to grey.

|  |  |  |  |
| --- | --- | --- | --- |
| 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:

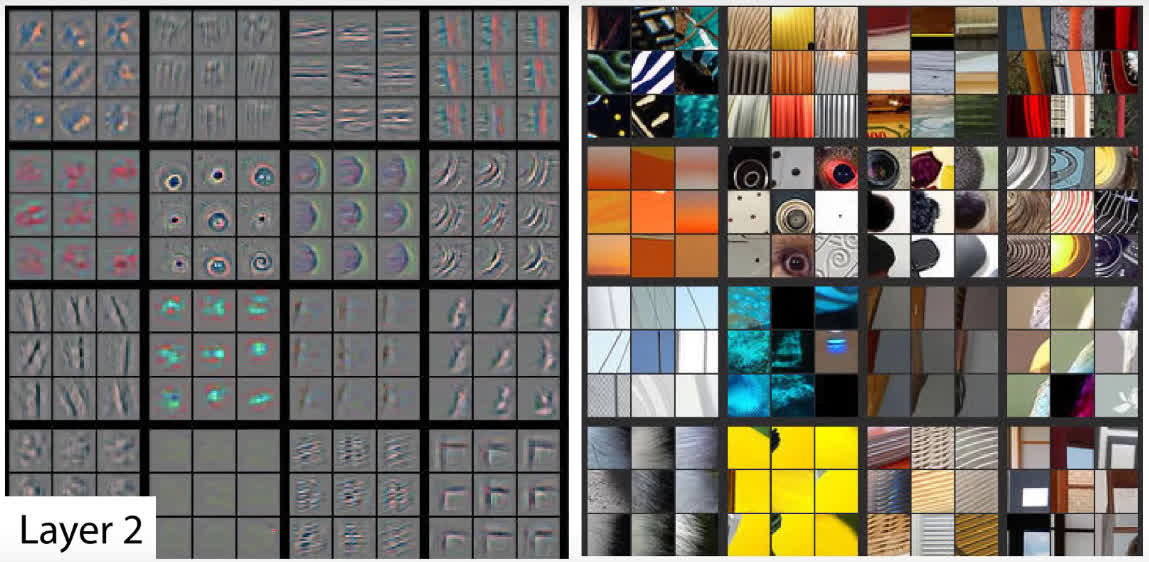
   

|  |
| --- |
| Output channels from the Convolutional Layer |
|  |

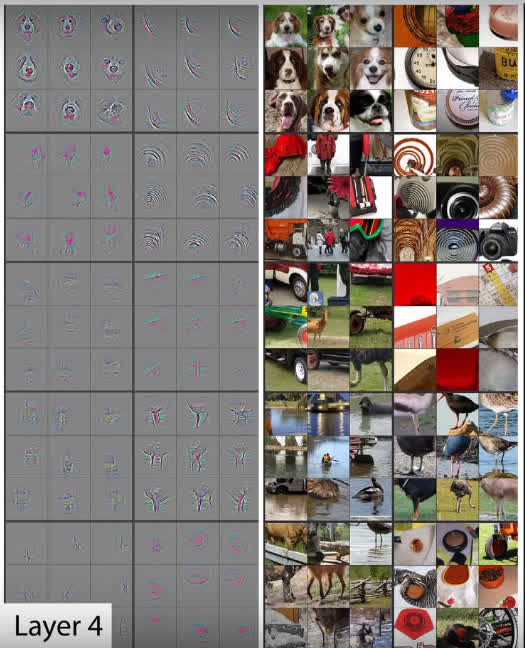
We can see that all four of these filters are detecting edges. In the output channels, the brightest pixels can be interpreted as what the filter has detected. In the first one, we can see detects top horizontal edges of the seven, and that’s indicated by the brightest pixels (white).

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:



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](https://en.wikipedia.org/wiki/Sobel_operator), an edge detector.

## VISUALIZING CONVOLUTIONAL FILTERS

TEHNIČKO POGLAVLJE, NIJE PREVEDENO ZA TEORIJU

In this post, we’re going to discuss 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 [post](https://deeplizard.com/learn/video/YRhxdVk_sIs) on convolutional neural networks.

In that [post](https://deeplizard.com/learn/video/YRhxdVk_sIs), 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 [post](https://deeplizard.com/learn/video/YRhxdVk_sIs), we looked at some filters from a CNN and observed what they were able to detect from real world images.

### KERAS AND THE CODE

Most of the code we’ll be using to visualize the filters comes from the blog, [How convolutional neural networks see the world](https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html), by the creator of Keras, [François Chollet](https://twitter.com/fchollet).

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

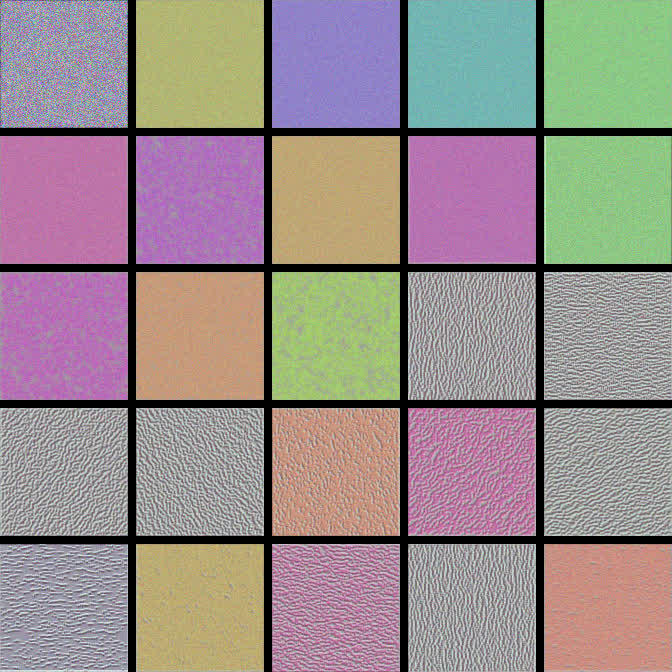
save\_img('stitched\_filters\_%dx%d.png' % (n, n), stitched\_filters)

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

### 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.

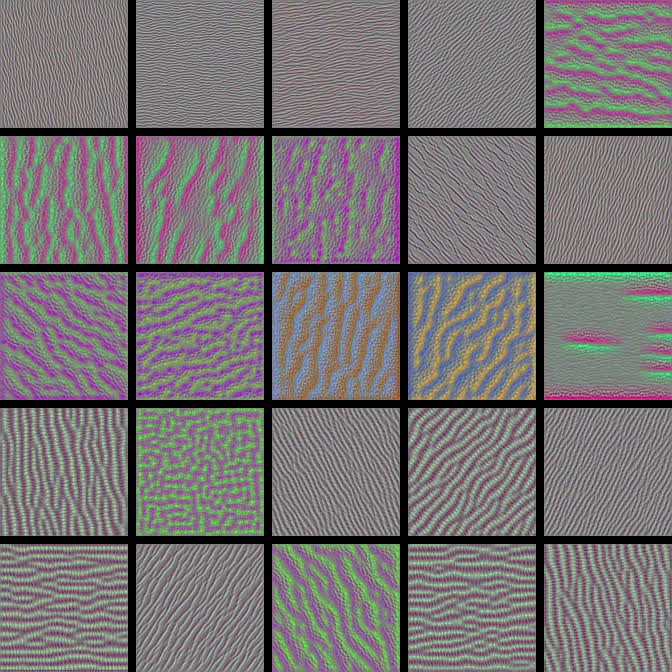
**1st CONV LAYER FROM THE 1st CONV BLOCK**

****

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

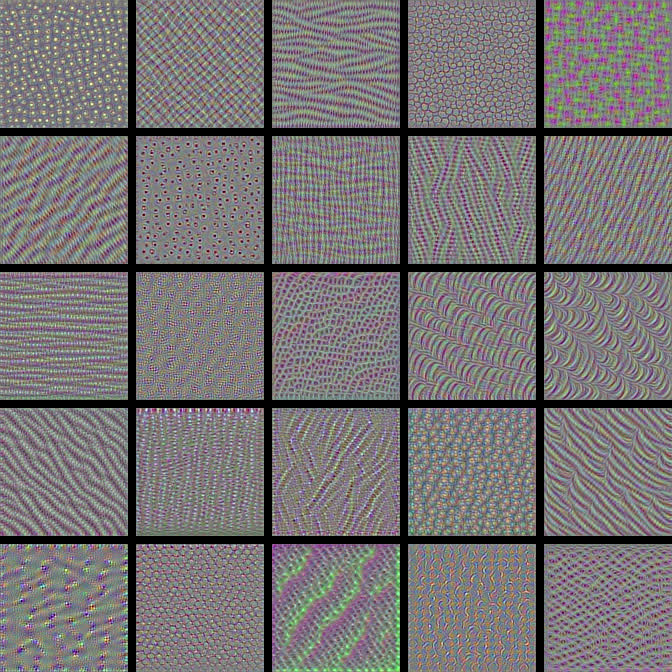
Let’s skip to another deeper convolutional layer. We’ll choose the second conv layer from the second conv block.

**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.

**2nd CONV LAYER FROM THE 3rd CONV BLOCK**

****

With each deeper convolutional layer, we’re getting more complex and more interesting visualizations.

* Gradient ascent differs from gradient descent by trying to maximize los sin order to emphasize pattenr detection of the filter

## 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.

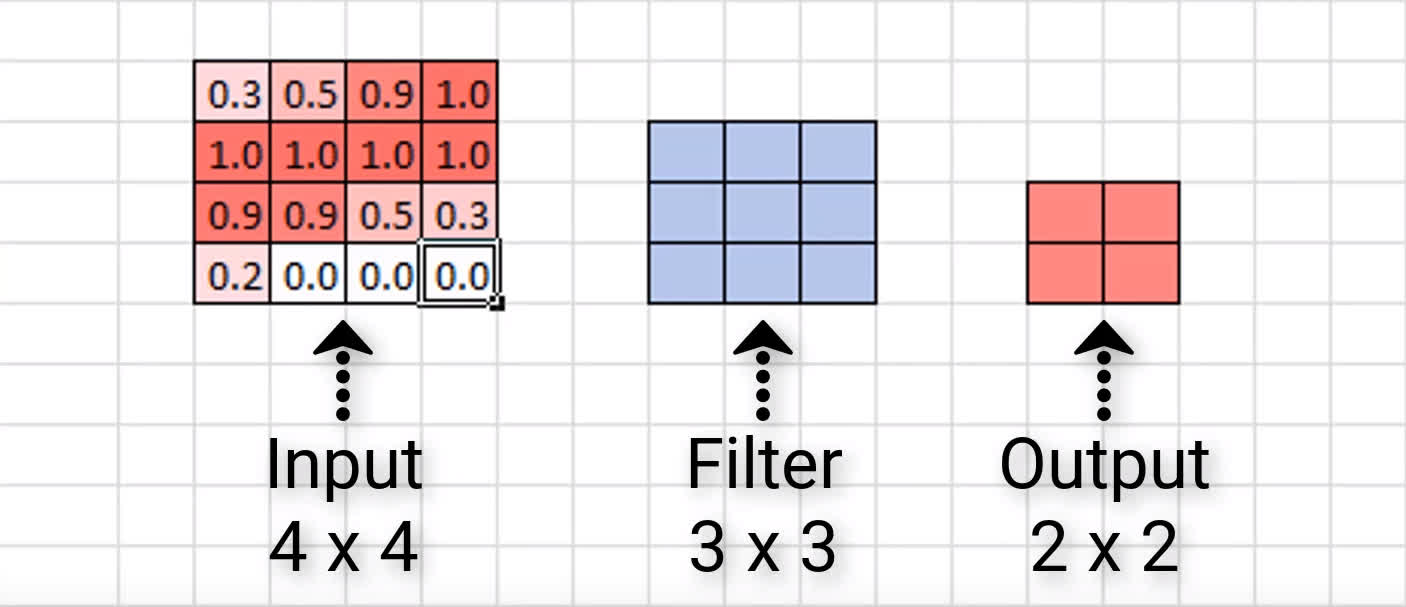
### CONVOLUTIONS REDUCE CHANNEL DIMENSIONS

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.

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.



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

In general, if our image is of size n x n, and we convolve it with an f x f filter, then the size of the resulting output is (n–f+1) x (n–f+1).

### ISSUES WITH REDUCING THE DIMENSIONS

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?

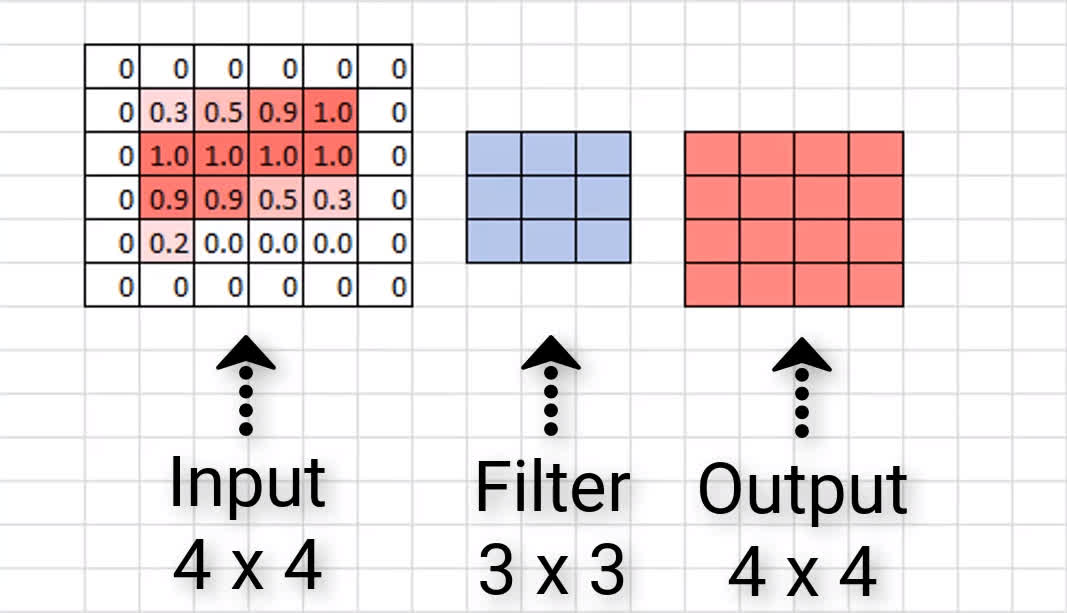
Well, what’s going to happen is that the resulting output is going to continue to become smaller and smaller. This is a problem.

If we start out with a 4 x 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.

### ZERO PADDING

*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.

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 x 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.

**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 |

## 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.

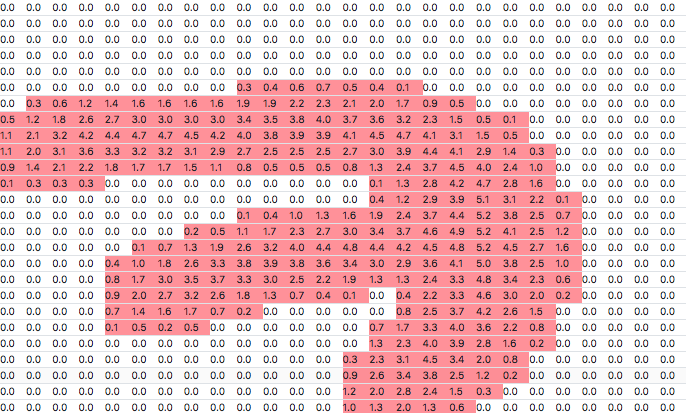
**EXAMPLE USING A SAMPLE FROM THE MNIST DATASET**

We’ve seen in our [post on CNNs](https://deeplizard.com/learn/video/YRhxdVk_sIs) 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.

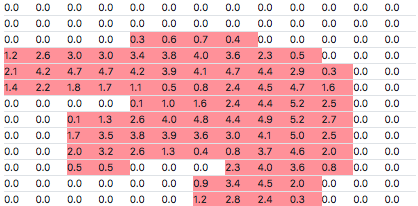
We used a 3 x 3 filter to produce the output channel below:



|  |
| --- |
| 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:



|  |
| --- |
| 13 x 13 output channel |
|  |

Max pooling works like this. We define some n x n region as a corresponding filter for the max pooling operation. We’re going to use 2 x 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.

On the convolutional output, and we take the first 2 x 2 region and calculate the max value from each value in the 2 x 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 x 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 x 2 blocks in this row.

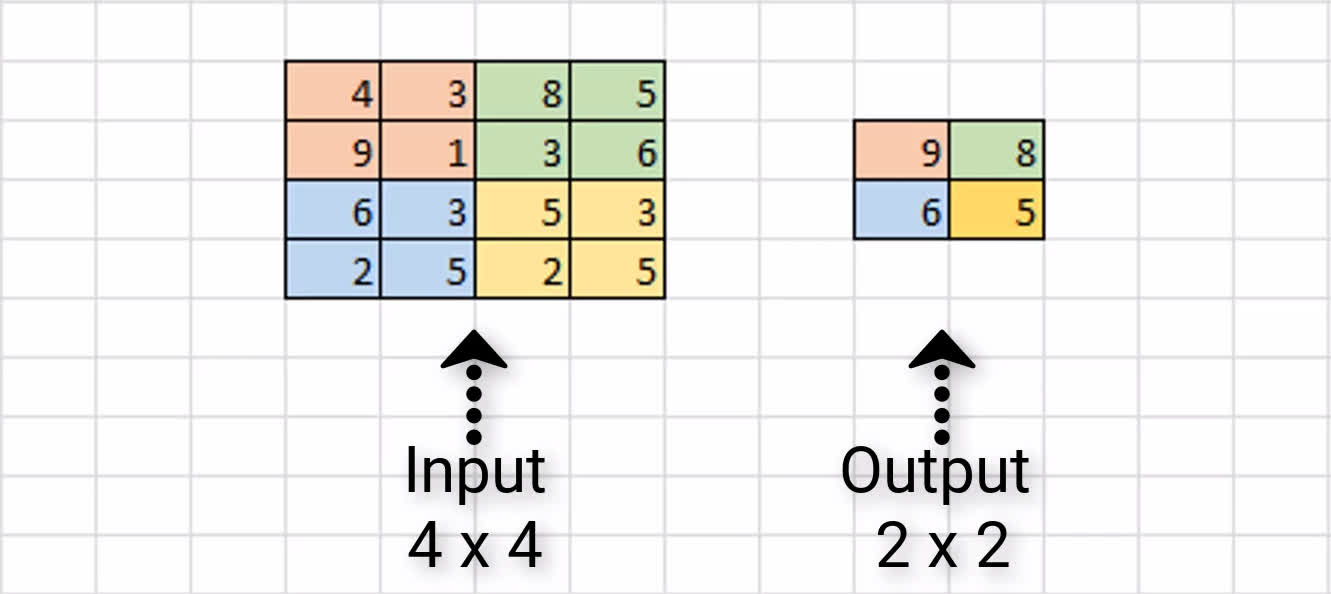
We can think of these 2 x 2 blocks as *pools* of numbers, and since we’re taking the max value from each pool, we can see where the name *max pooling* came from.

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](https://deeplizard.com/resource/pavq7noze2) operation output is 26 x 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 x 13.

**SCALED DOWN EXAMPLE**

Suppose we have the following:



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

Our first 2 x 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 x 4 input channel, and the resulting output channel is this 2 x 2 block. As a result, we can see that our input dimensions were again reduced by a factor of two.

### WHY USE MAXPOOLING

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.

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.

## BACKPROPAGATION IN NEURAL NETWORKS

Hey, what’s going on everyone? In this post, we’re going to discuss *backpropagation*and what its role is in the [training process](https://deeplizard.com/learn/video/sZAlS3_dnk0) 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.

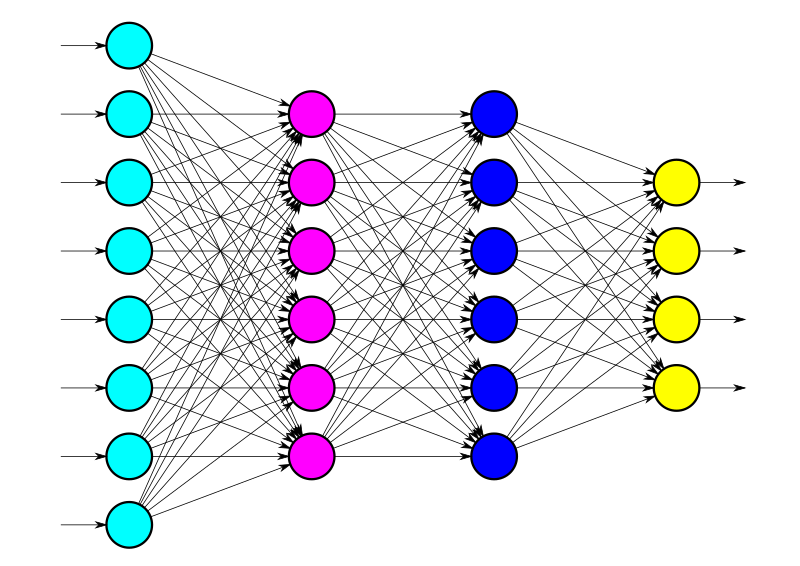
In the previous posts we referenced, we discussed how, during [training](https://deeplizard.com/learn/video/sZAlS3_dnk0), stochastic gradient descent, or SGD, works to minimize the [loss function](https://deeplizard.com/learn/video/Skc8nqJirJg)by updating the weights with each epoch.

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.

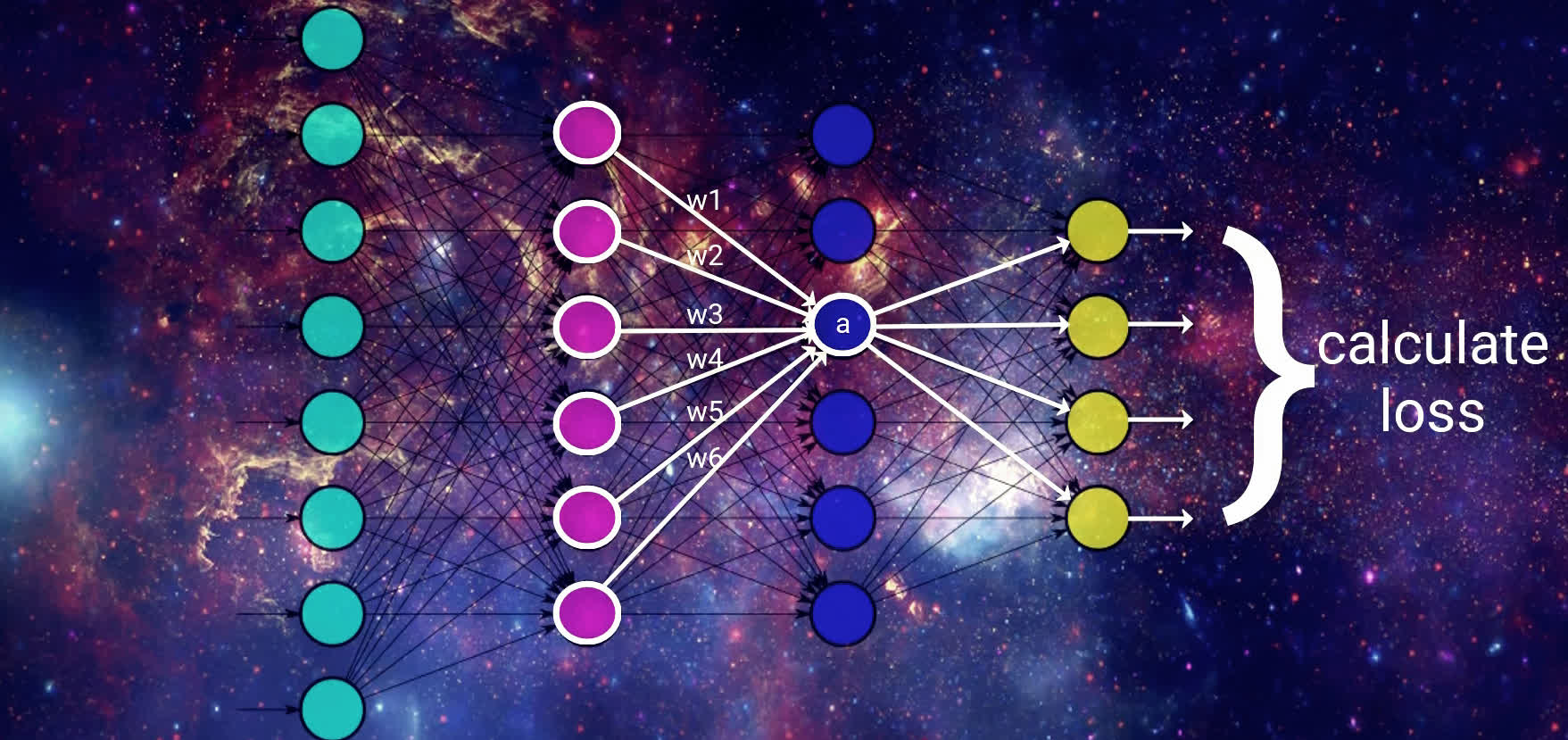
Recall that each node in our model receives its input from the previous layer, and that this input is a weighted sum of the weights at each of the connections multiplied by the previous layer’s output.

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](https://deeplizard.com/learn/video/Skc8nqJirJg) 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.

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.

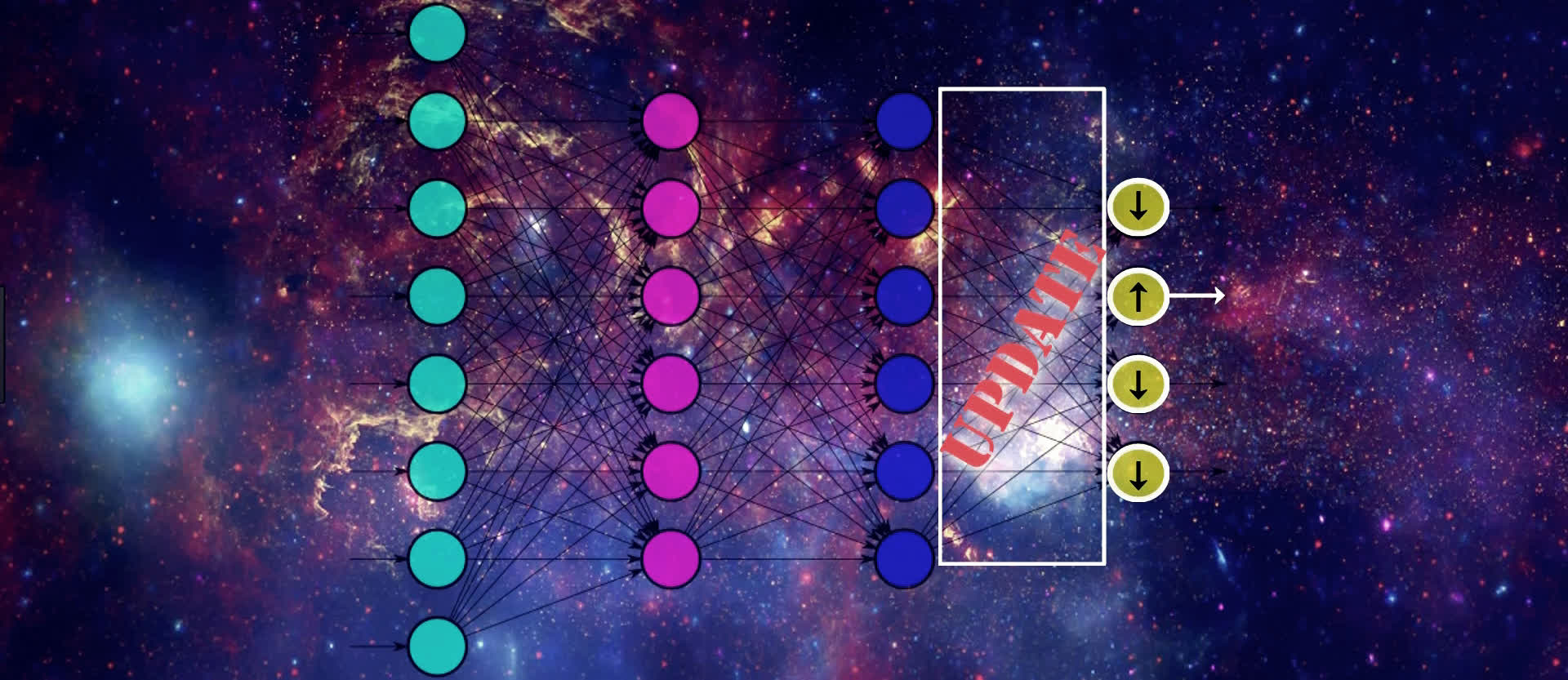
Backpropagation is the tool that gradient descent uses to calculate the gradient of the loss function.

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.

### BACKPROPAGATION INTUITION

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

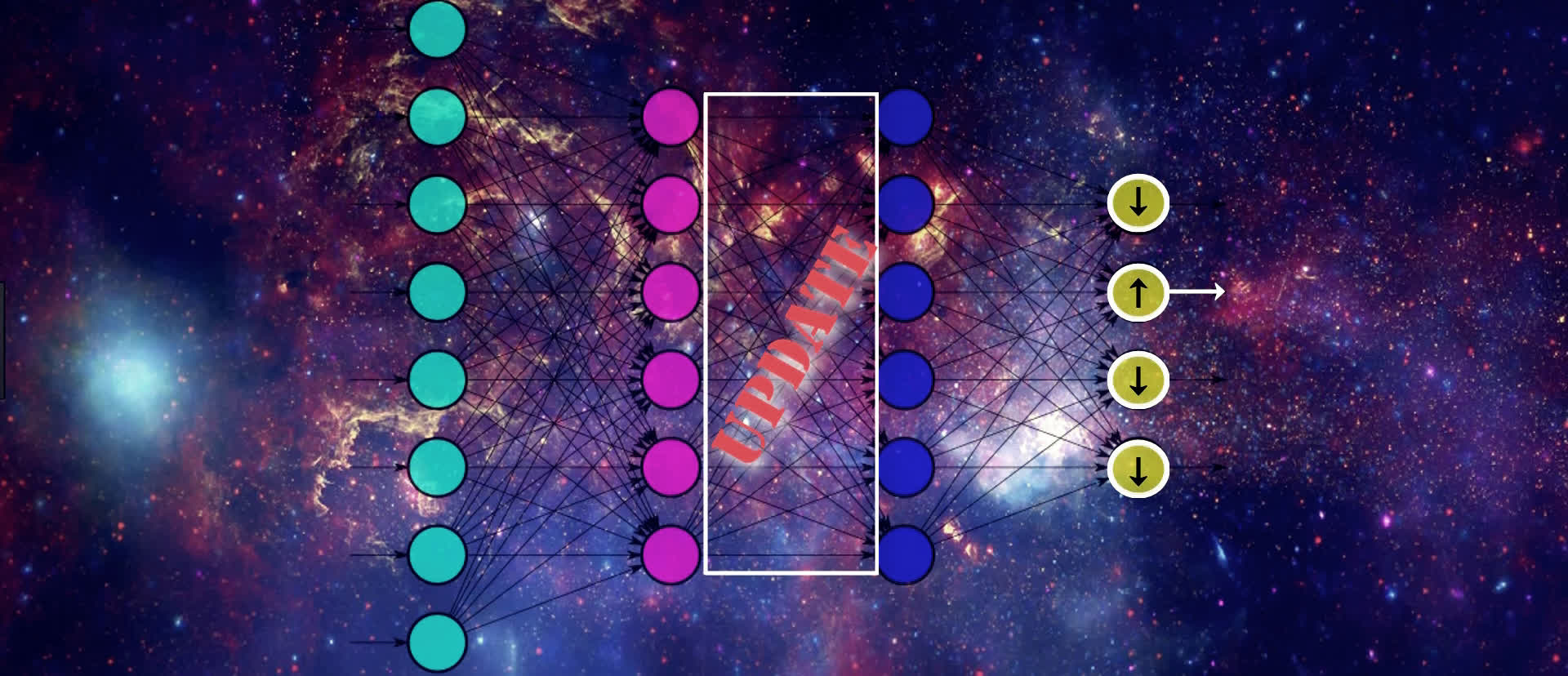
Suppose that this output node here with the up arrow pictured below maps to the output that our given input actually corresponds to. If that’s the case, then gradient descent understands that the value of this output should increase, and the values from all the other output nodes should decrease. Doing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

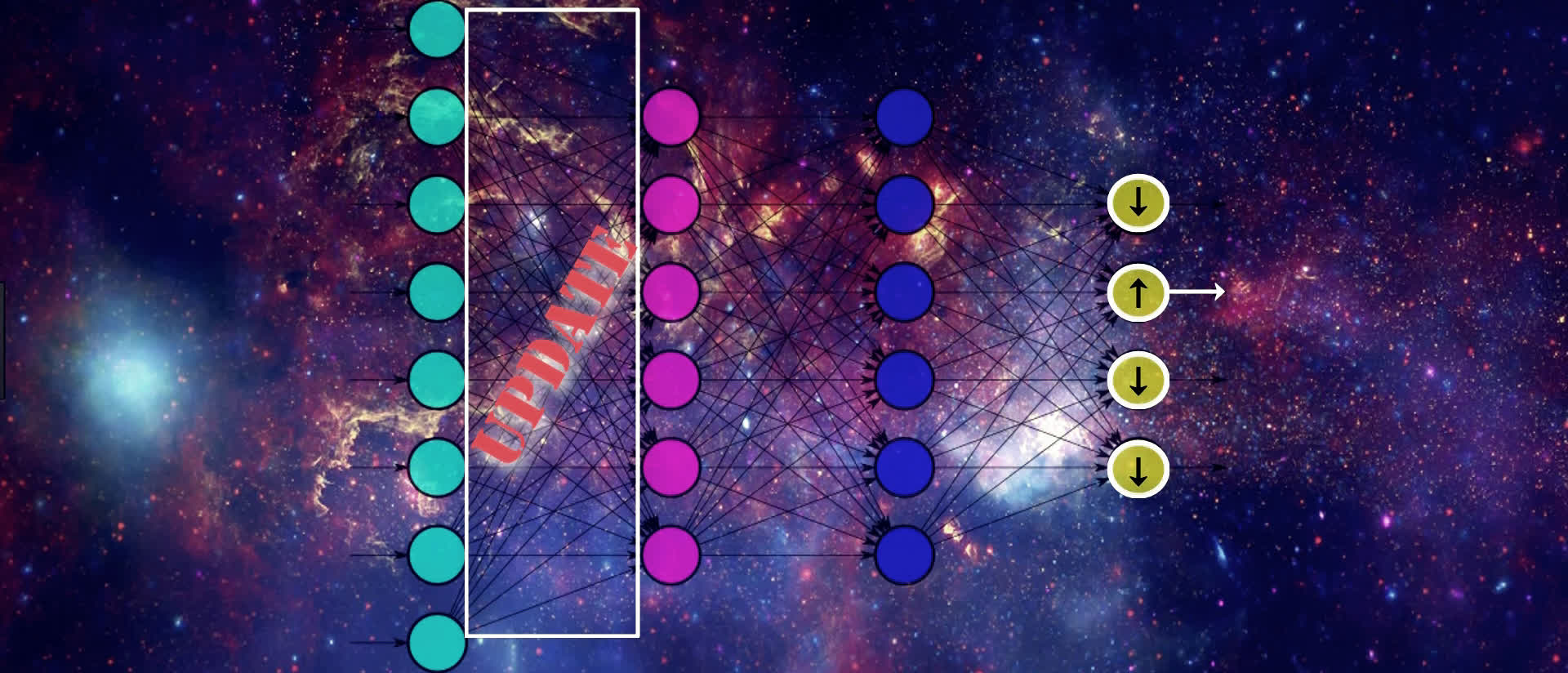
We can’t actually directly change the activation output because it’s a calculation based on the weights and the previous layer’s output. But, we can *indirectly* influence a change in this layer’s activation output by jumping backwards, and again, updating the weights here in the same way we just discussed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

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**

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.

### VANISHING AND EXPLODING GRADIENT

In this episode, we’re going to discuss a problem that creeps up time and time again during the training process of an artificial neural network.

This is the problem of unstable gradients and is most popularly referred to as the vanishing gradient problem.

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.

**SMALL GRADIENTS**

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.

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.

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

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.

## WEIGHT INITIALIZATION EXPLAINED

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.

### HOW ARE WEIGHTS INITIALIZED?

We briefly touched on this concept in our [episodes on backpropagation](https://deeplizard.com/learn/video/XE3krf3CQls). 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.

**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.

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.

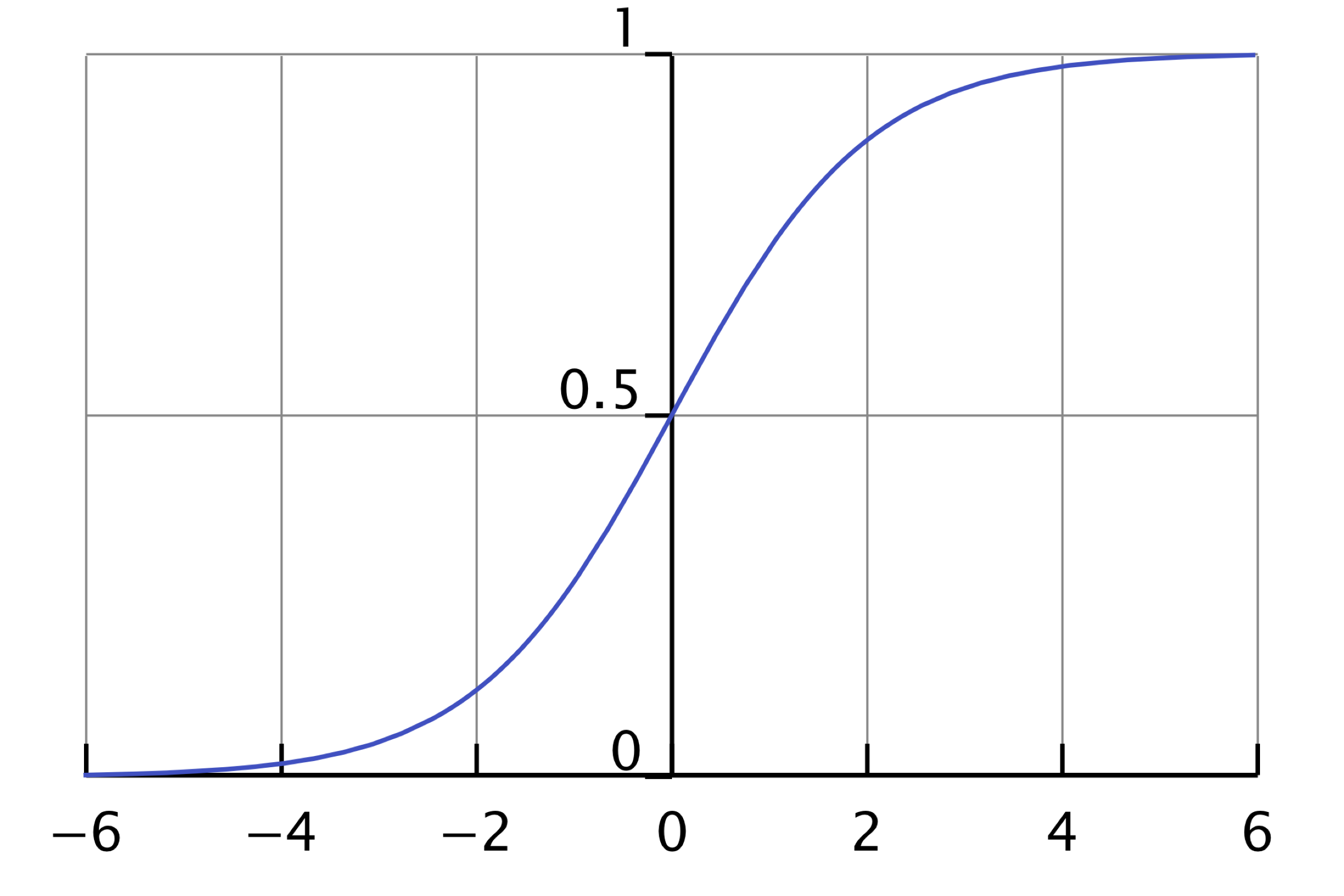
Well, z, as a sum of normally distributed numbers with a mean of 0, will also be normally distributed around 0, but its variance, and similarly its derived standard deviation, will be larger than 1.

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](https://deeplizard.com/learn/video/m0pIlLfpXWE), 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](https://deeplizard.com/learn/video/qO_NLVjD6zE).

### XAVIER INITIALIZATION

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 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/nin+nout as the variance where nin is defined as the number of weights coming into this neuron, and nout is the number of weights coming out of this neuron.

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.

## BIAS IN AN ARTIFICIAL NETWORK

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’ll first start out by discussing 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.

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.

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 THE BIAS FITS IN**

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.

**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

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

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.

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. 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.

Passing this value to relu, we can see that

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.

Now, we explicitly choose and set our bias in our example. In practice, this isn’t the case.

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 IN A NEURAL NETWORK

We’ve already talked a lot about learnable parameters in a neural network, but we haven’t necessarily given the general topic a formal introduction.

In this episode, we’ll start out by defining what a learnable parameter within a neural network is.

Then, we will see how the total number of learnable parameters within a network is calculated.

After we see how this is done, we’ll illustrate the calculation using a simple neural network.

### LEARNABLE PARAMETERS

A learnable parameter is a parameter that is learned by the network during training.

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**

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.

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.

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.

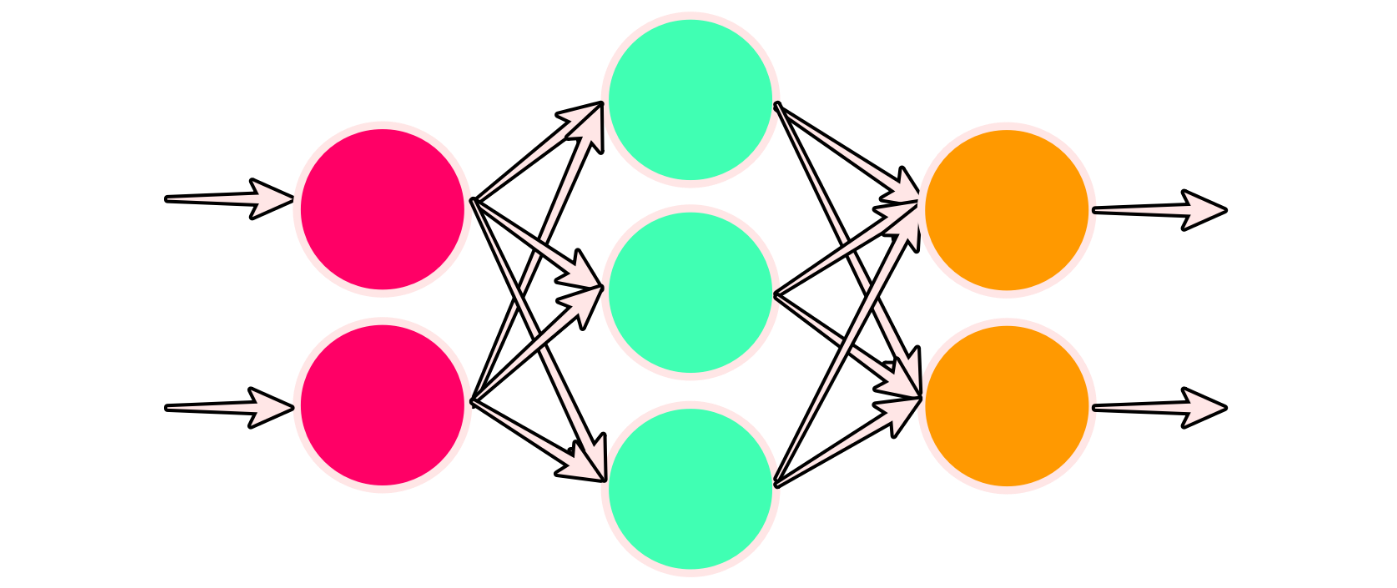
**LEARNABLE PARAMETERS EXAMPLE CALCULATION**

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.

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.

Let’s calculate the number of learnable parameters within the hidden layer.

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. We have three coming from our three nodes in the hidden layer.

We have two outputs coming from the output layer, since that’s the amount of nodes this layer has. Also, we have two biases, again since that’s how many nodes we have in the layer.

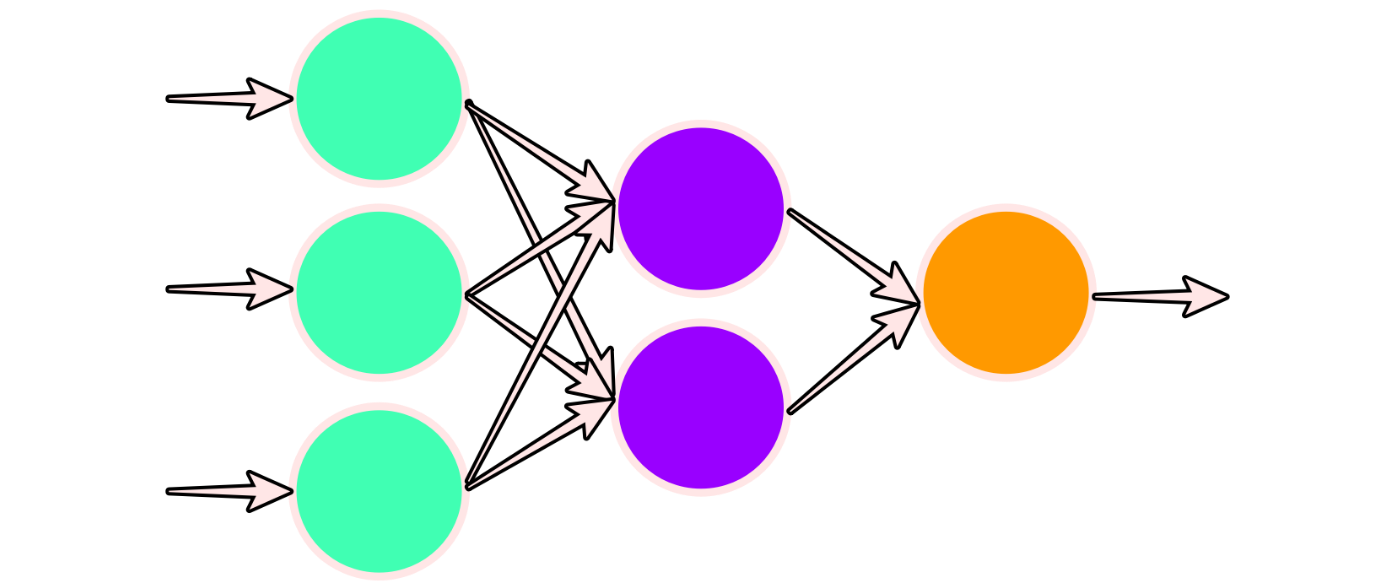
Multiplying our input by our output, we have three times two, so that’s six weights, plus two bias terms. That’s eight learnable parameters for our output layer.

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 parameters.

## LEARNABLE PARAMTERS IN A CNN

Last time, we learned about learnable parameters in a fully connected network of dense layers. Now, we’re going to talk about these parameters in the scenario when our network is a [convolutional neural network](https://deeplizard.com/learn/video/YRhxdVk_sIs).

We’ll first start out by discussing what the learnable parameters within a convolutional neural network are, and then see how the total number of learnable parameters within a CNN is calculated. And after we see how this is done, we’ll illustrate the calculation using a simple convolutional neural network.



### WHAT ARE LEARNABLE PARAMETERS IN A CNN

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.

### 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.

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, the output of a convolutional layer is:

* 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.

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.

Just with a convolutional layer, the inputs and outputs themselves are considering the number of filters and the size of the filters.

### 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 20x20specifies 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.

* 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**

The input layer has no learnable parameters since it just contains the input data.

**CONV LAYER 1**

We have 3 inputts from our input layer. 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. We have just two biases, 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**

We have two inputs from the number of filters in the previous layer. Calculating the outputs, 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**

We may think that the output layer has just three inputs since that’s the number of filters in the last convolutional layer. But, before passing output from a convolutional layer to a dense layer using Keras, 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 this post, we’ll discuss what regularization is, and when and why it may be helpful to add it to our model.

In our [previous post](https://deeplizard.com/learn/video/DEMmkFC6IGM) on overfitting, we briefly introduced dropout and stated that it is a regularization technique.

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.

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.

### 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.

With L2 regularization, the term we’re adding to the loss is the sum of the squared norms of the weight matrices

multiplied by a small constant

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. 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

**ADDING THE TERM TO THE LOSS**

We have

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 jth 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

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](https://deeplizard.com/learn/video/Skc8nqJirJg). Our original loss function is now being summed with the sum of the squared matrix norms,

Which is multiplied by

If λ is large, then this term, λ2m, 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 NETWORK

In this post, we’ll discuss what it means to specify a batch size as it pertains to training an artificial neural network.

### 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.

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.

**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

**WHY USE BATCHES?**

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**

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.

## FINE TUNING NEURAL NETWORKS

In this post, we’ll discuss what fine-tuning is and how we can take advantage of it when building and training our own artificial neural networks.

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.

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.

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.

## 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.

**FREEZIN 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](https://deeplizard.com/learn/video/sZAlS3_dnk0), 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

In this post, we’ll be discussing *batch normalization*, otherwise known as *batch norm*, and how it applies to training artificial neural networks.

### NORMALIZATION TECHNIQUES

In this post, we’ll be discussing *batch normalization*, otherwise known as *batch norm*, and how it applies to training artificial neural networks.

Before getting to the details about batch normalization, let’s quickly first discuss regular normalization techniques.

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](https://en.wikipedia.org/wiki/Normalization_(statistics)) 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](https://deeplizard.com/learn/video/YDDqe60omL0) 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.

### USE OF NORMALIZATION TECHNIQUES

In general, this all boils down to putting our data on some type of known or standard scale.

If we didn’t normalize our data in some way, we can imagine that we may have some [numerical data](https://deeplizard.com/learn/video/YDDqe60omL0) 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](https://deeplizard.com/learn/video/qO_NLVjD6zE).

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 post](https://deeplizard.com/learn/video/_N5kpSMDf4o) 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**

If, during training, one of the weights ends up becoming drastically larger than the other weights, 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](https://deeplizard.com/learn/video/FK77zZxaBoI), the first thing batch norm does is normalize the output from the activation function. Recall from our [post on activation functions](https://deeplizard.com/learn/video/m0pIlLfpXWE)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 |  | Normalize output x from activation function. |
| 2 |  | Multiply normalized output z by arbitrary parameter g. |
| 3 |  | Add arbitrary parameter b to resulting product (z∗g). |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Batch Normalization Process  **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](https://deeplizard.com/learn/video/pg3hJpSopHQ), meaning that they will be become learned and optimized during the training process.   |  |  | | --- | --- | | Hyperparameters | | | **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.