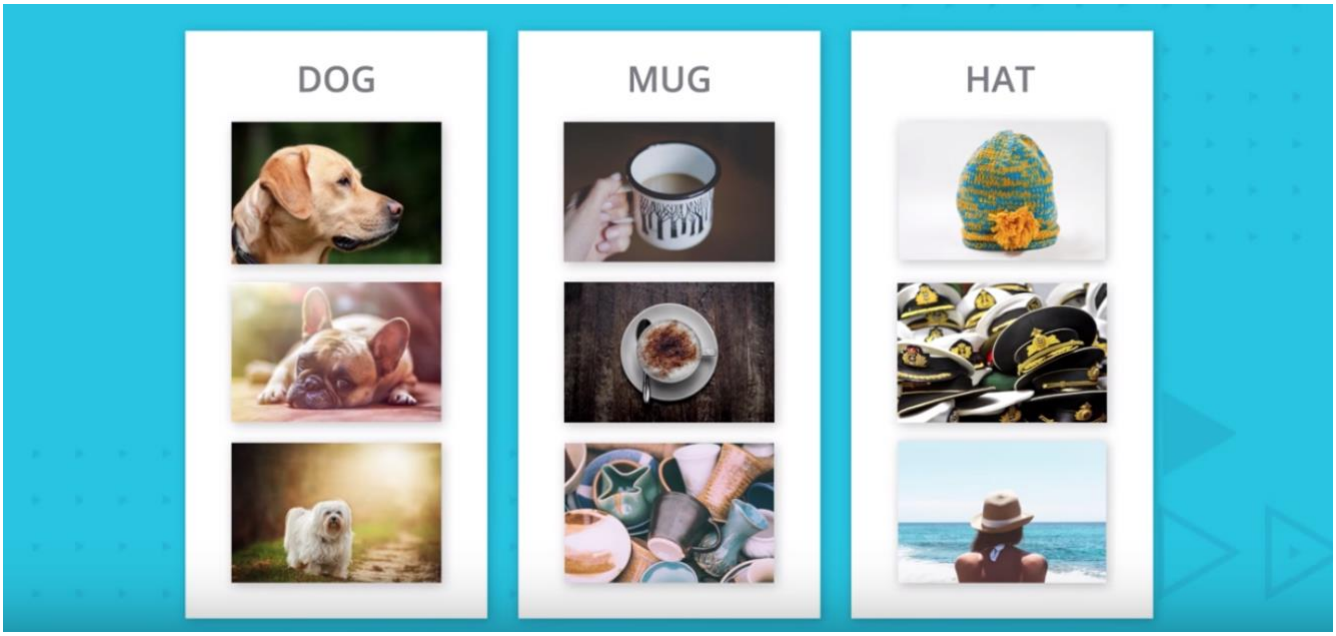
Already_read_begin

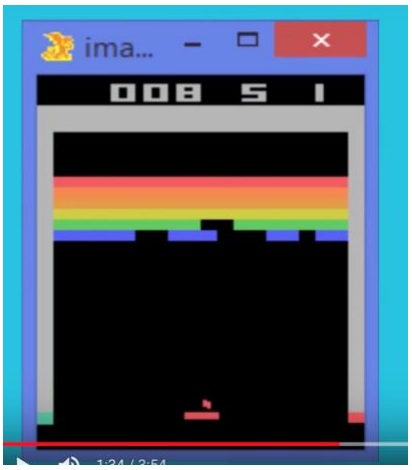
Lesson 5: Convolutional Neural Networks

(No picture)
93. Now that you have a solid foundation in Neural Networks, we are excited to teach you about convolutional neural networks. I'm happy to introduce you to Alexis Cook, an expert in the subject. Hi everyone. I'm Alexis. Over the next several videos, I'll help you learn more about Convolutional Neural Networks, which we'll also refer to as CNNs. For now, let's investigate some of their fascinating applications.

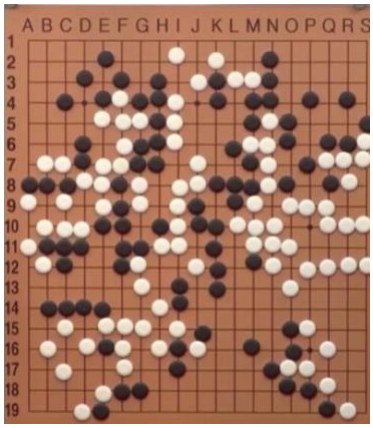
(No picture)
94. CNNs achieve state of the art results in a variety of problem areas including Voice User Interfaces, Natural Language Processing, and computer vision. In the field of Voice User Interfaces, Google made use of CNNs in its recently released WaveNet model which we've linked to below. WaveNet takes any piece of text as input and does an excellent job of returning computer-generated audio of a human reading the text. What's really cool is that if you supply the algorithm with enough samples of your voice, it's possible to train it to sound just like you. As for the field of Natural Language Processing, you'll see later that Recurrent Neural Networks are used more frequently than Convolutional Neural Networks. CNNs, however, can be used in this area too. We provide a link below that shows you how CNNs are used to extract information from sentences. This information can be used to classify sentiment. For example, is the writer happy or sad? If they're talking about a movie, did they like or dislike it?



In this program, we'll focus on applications and computer vision and specifically work towards applying CNNs to image classification tasks. Given an image, your CNN will assign a corresponding label which you believe summarizes the content of the image. This is a core problem in computer vision and has applications in a wide range of problem areas.

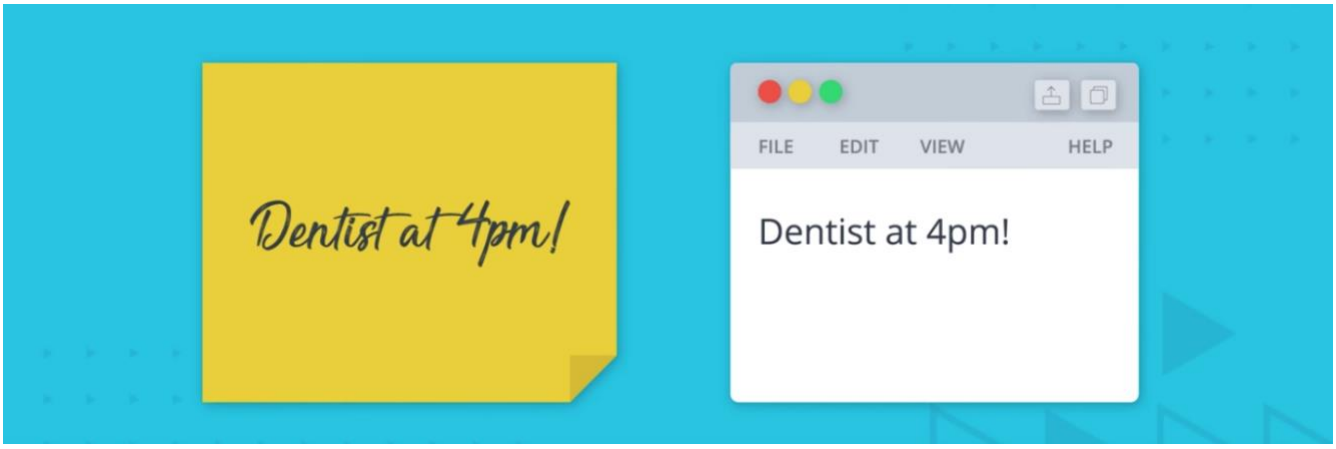


For instance, CNNs are used to teach artificially intelligent agents to play video games such as Atari Breakout. The CNN-based models are able to learn to play games without being given any prior knowledge of what a ball is. And without even being told precisely what the controls do, the agent only sees the screen and its score but it does have access to all of the controls that you'd give a human user. With this limited knowledge, CNNs can extract crucial information that allows them to develop a useful strategy. CNNs have even been trained to play Pictionary. There's currently an app called Quickdraw that guesses what you're drawing based on your finger-drawn picture. The applications of CNNs are almost limitless.



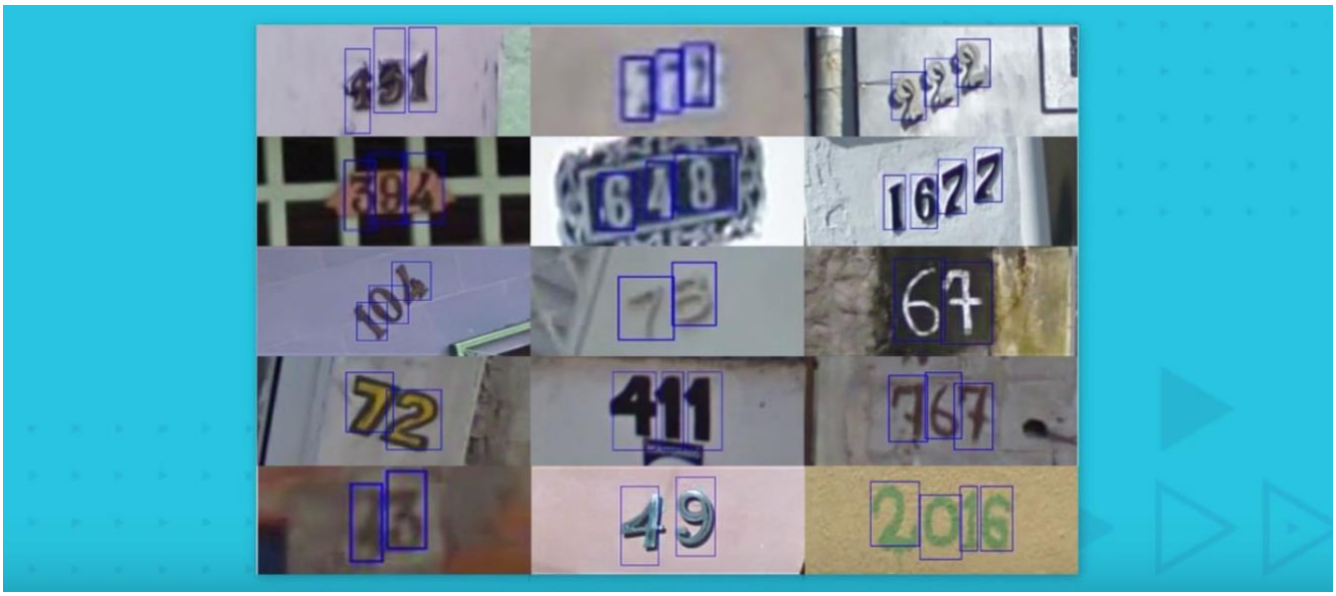
Go, for example, is an ancient Chinese board game considered one of the most complex games in existence. It said that there are more configurations in the game than there are atoms in the universe. Recently, researchers from Google's DeepMind use CNNs to train an artificially intelligent agent to beat human professional Go players.

CNNs also allowed drones (無人駕駛飛機) to navigate unfamiliar territory. Drones are now used to deliver medical supplies to remote areas. And CNNs give the drone the ability to see or to determine what's happening in streaming video data. But for now we'll consider algorithms that can decode images of text.



Maybe you'd like to digitize the historical book or your handwritten notes, then you could start with developing an algorithm that can identify images of letters or numbers or punctuation.

Similarly, you could develop an algorithm to aid self-driving cars with reading road signs.



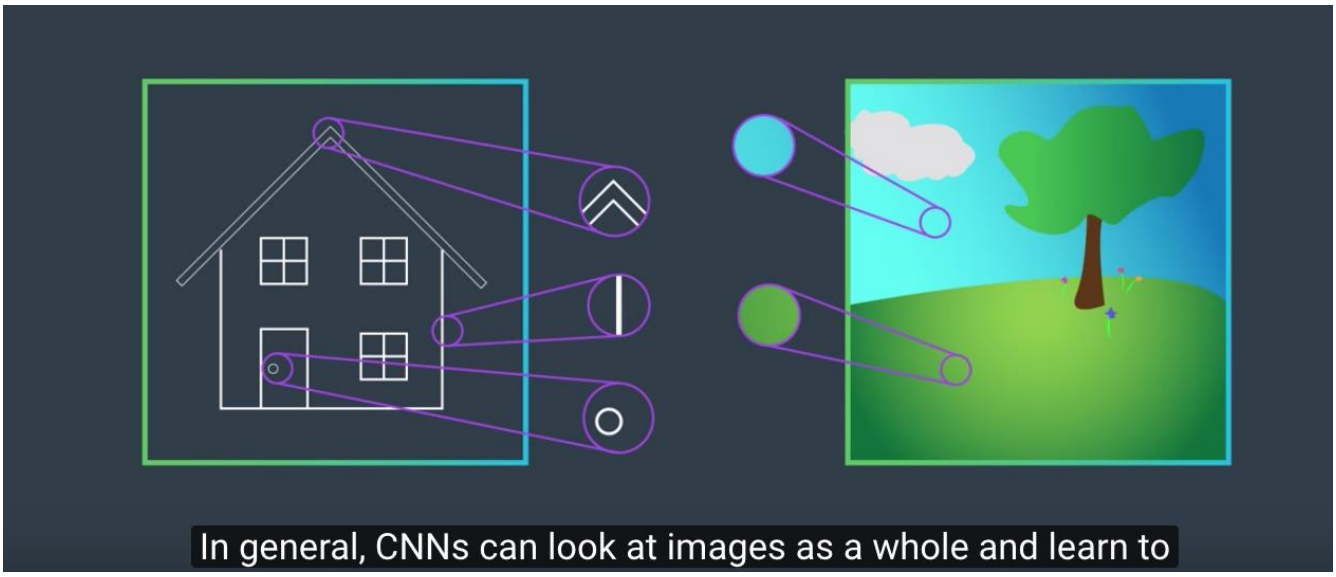
Likewise, Google has built a better more accurate street maps of the world by training an algorithm that can read house numbers signs from street view images. CNNs achieved state of the art performance in all of these problem domains. We mentioned that while it's possible to use the neural network from the previous section for image classification, you can almost always, if not always, get better results with the CNN. In the next few videos, we will explore why this is the case.

Hi, I'm Cezanne Camacho, I'll be teaching this lesson in tandem with Alexis, and later on, show you how to implement CNNs in PyTorchand use them in a variety of ways.

What is a feature?
I've found that a helpful way to think about what a feature is, is to think about what we are visually drawn to when we first see an object and when we identify different objects. For example, what do we look at to distinguish a cat and a dog? The shape of the eyes, the size, and how they move are just a couple of examples of visual features.

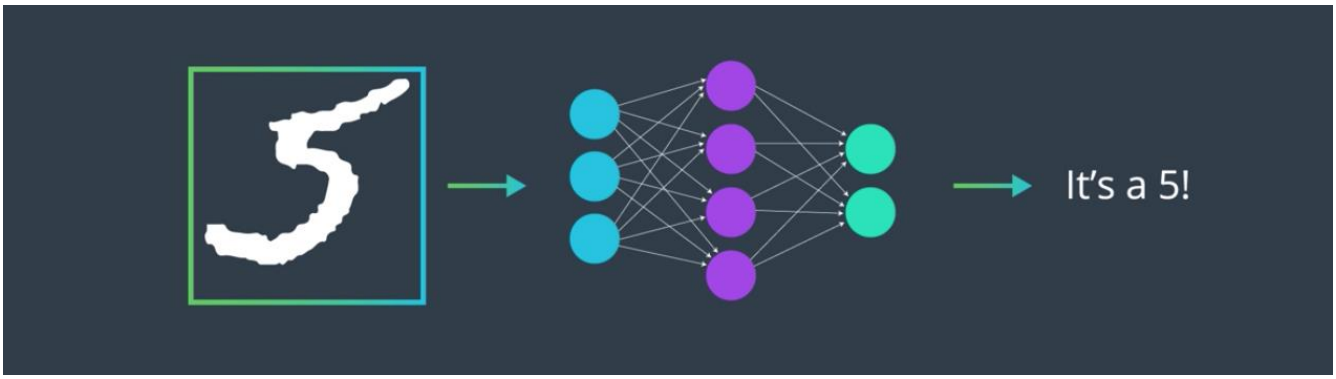
As another example, say we see a person walking toward us and we want to see if it's someone we know; we may look at their face, and even further their general shape, eyes (and even color of their eyes). The distinct shape of a person and their eye color a great examples of distinguishing features!

Next, we'll see that features like these can be measured, and represented as numerical data, by a machine.

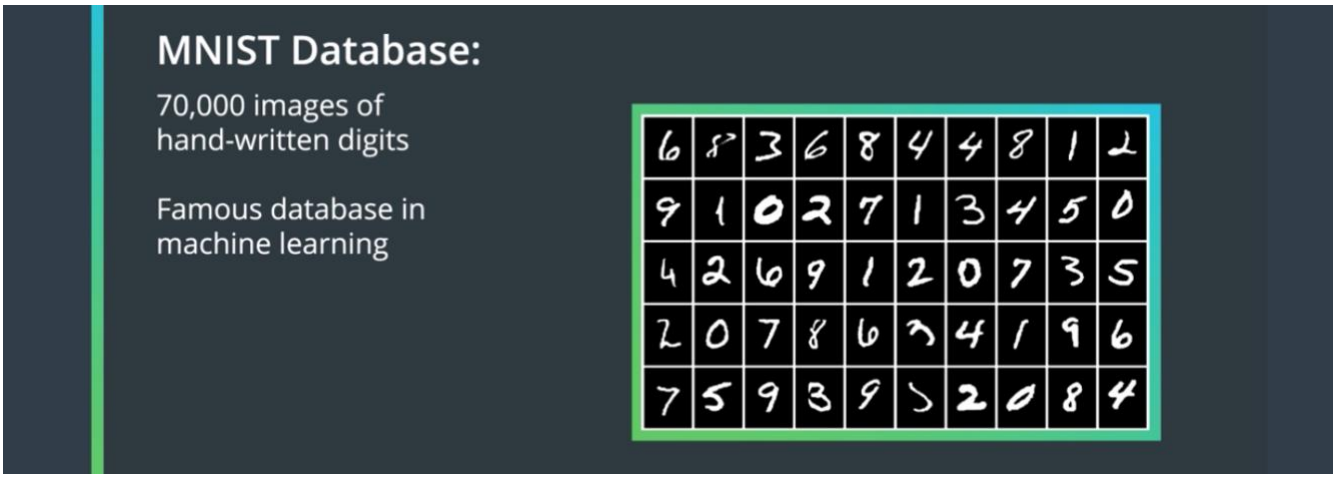


95. I'll be teaching this lesson in tandem with Alexis. And as she already mentioned this lesson is all about convolutional neural networks and how they improve our ability to classify images. In general, CNNs can look at images as a whole and learn to identify spatial patterns such as prominent colors and shapes, or whether a texture is fuzzy or smooth and so on. The shapes and colors that define any image and any object in an image are often called features. I'll cover how a CNN can learn to identify these features and how a CNN can be used for image classification. And Alexis will talk about the various layers that make up a complete convolutional neural network.

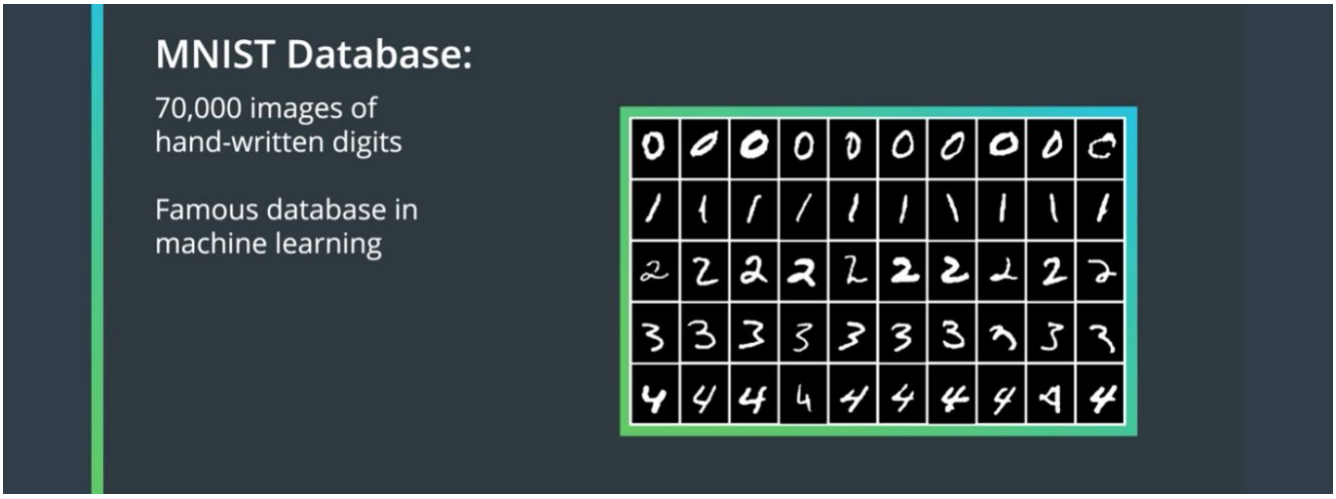
By the end of this lesson, you should have all the skills you need to define and train an image classifier of your own.



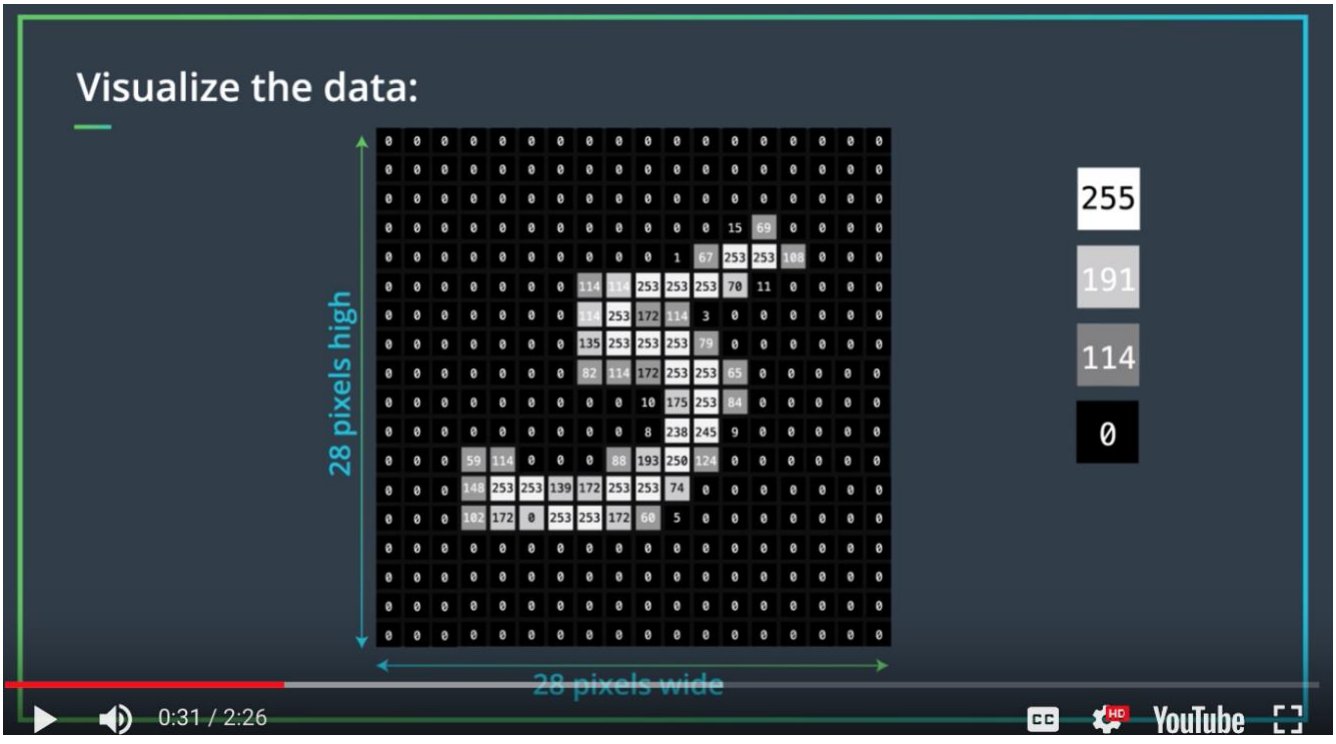
96. Let's begin by investigating how deep learning can be used to recognize a single object in an image. In this example, we want to design an image classifier that takes in an image of a hand-written number, and produces a predicted class for that number. This class ideally, correctly identifies the given image.



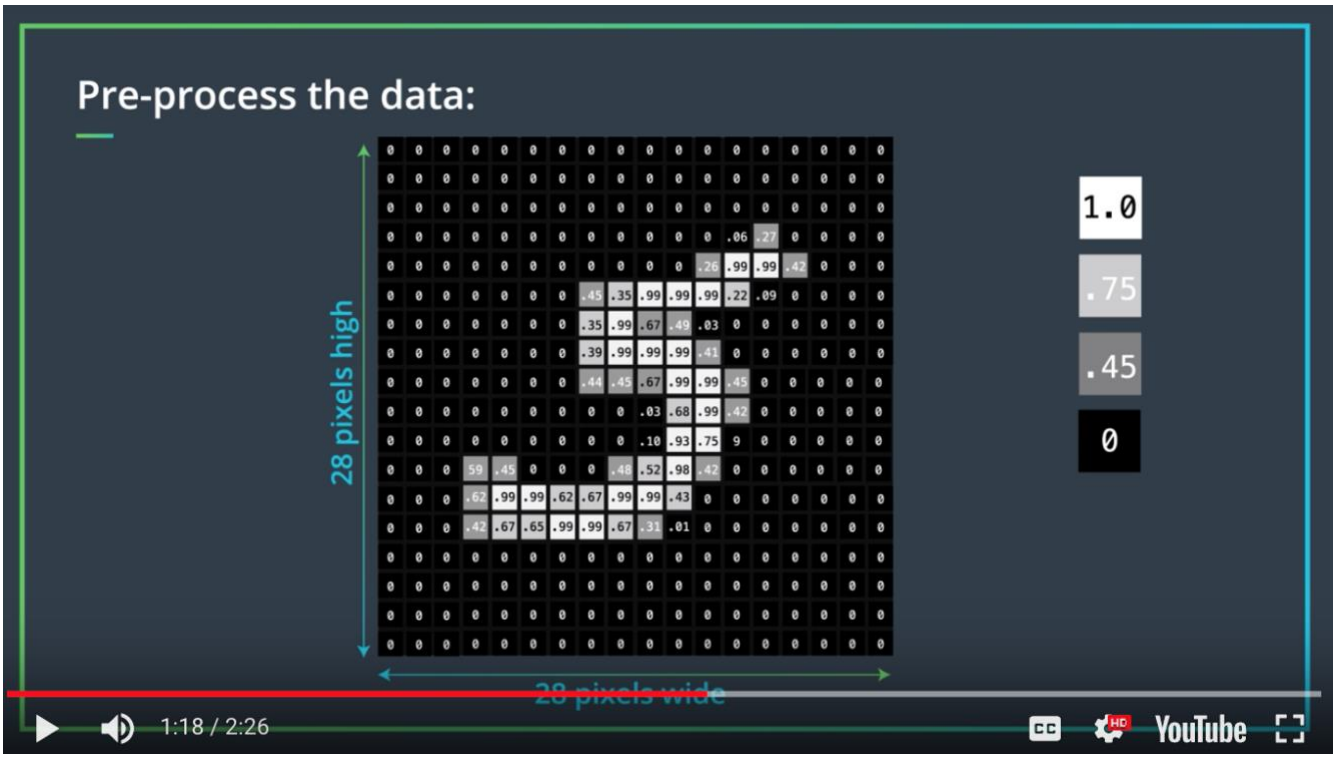
To build this, we'll use the MNIST database, which contains thousands of small gray scale images of hand-written digits. Each image depicts one of the numbers zero through nine. This database is perhaps one of the most famous databases in the field of machine and deep learning. It was one of the first databases used to prove the usefulness of neural networks and has continued to inform the development of new architectures overtime. If you look over this small sample of data, you might notice that some digits are more legible than others. For instance, if you squint just right, this three could pass for an eight. You can imagine that the dataset contains some fours that could pass for nines, or ones and sevens. Even with these subtleties, for us the task of identifying these numbers is pretty easy. We also want to design an appropriate model such that it too can accurately distinguish between each of these numbers. Using deep learning, we can take a data-driven approach to training an algorithm that can examine these images and discover patterns that distinguish one number from another.



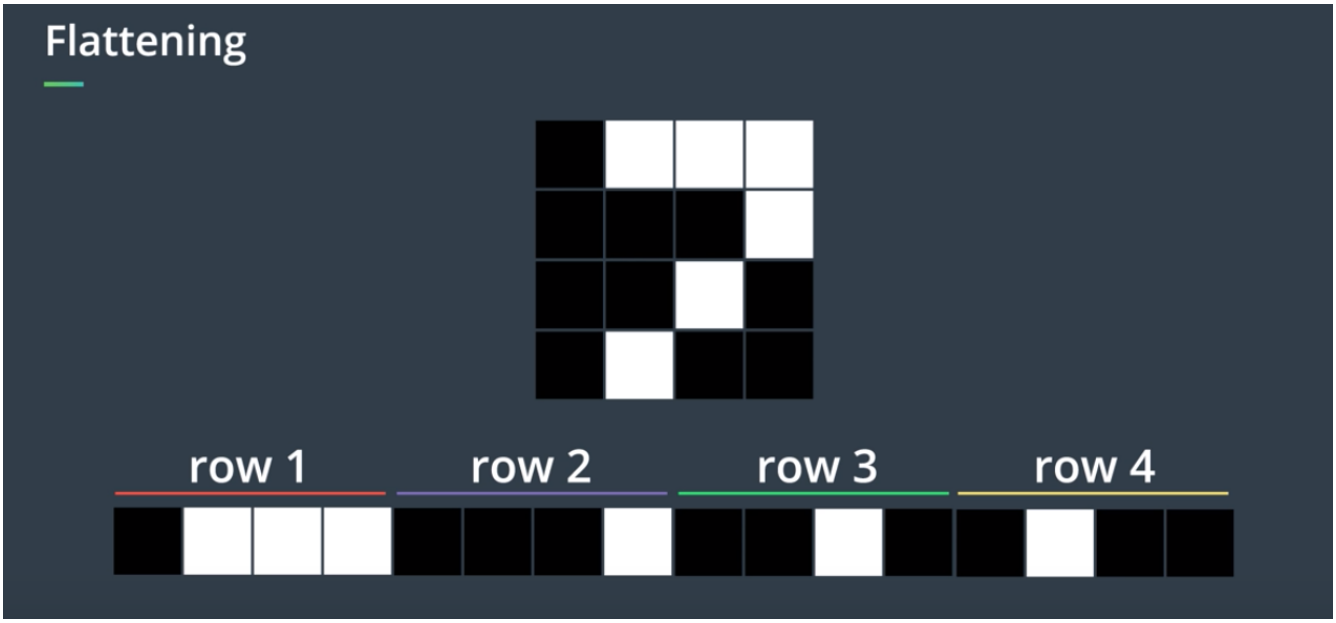
Our algorithm will need to attain some level of understanding of what makes a hand-drawn one look like a one, and how images of ones differ from images of say twos or threes. The first step in recognizing patterns in images is learning how images are seen by computers. So, before we start to design an algorithm, let's first visualize this data and take a closer look at these images.



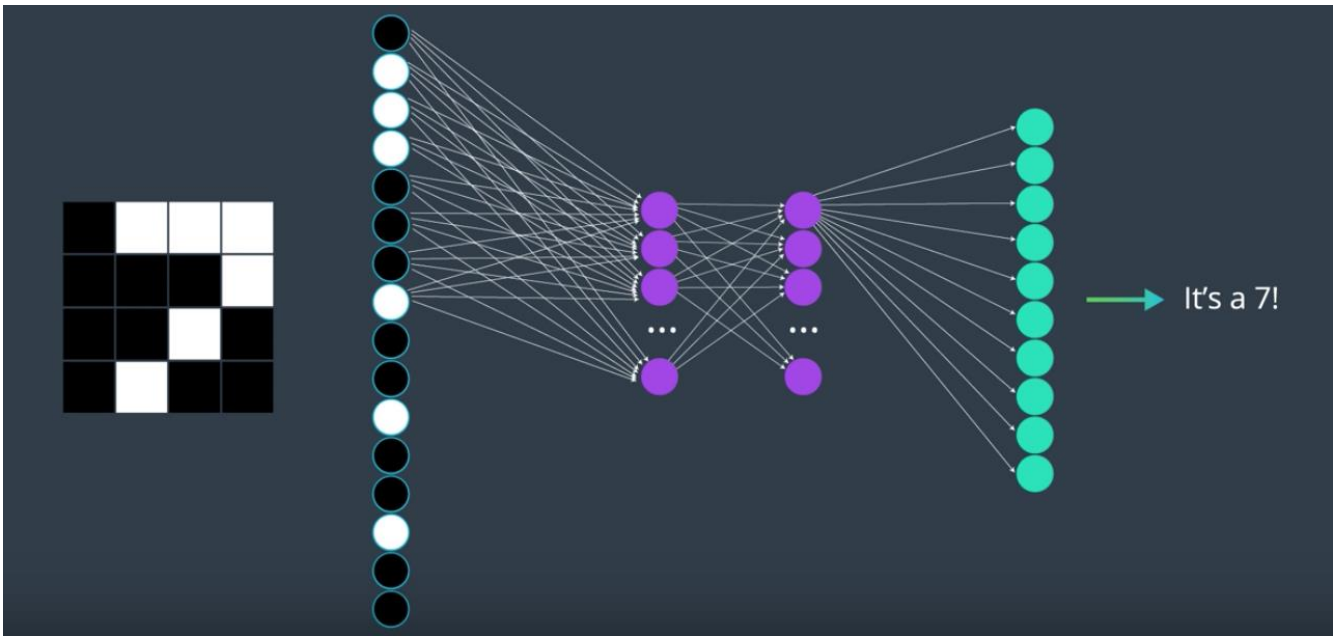
97. Any gray scale image is interpreted by a computer as an array. A grid of values for each grid cell is called a pixel, and each pixel has a numerical value. Each image in the MNIST database is 28 pixels high and wide. And so, it's understood by a computer as a 28 by 28 array. In a typical (not MNIST, tao) gray scale image, white pixels are encoded as the value 255 (picture is above, not below, tao), and black pixels are encoded as zero. Gray pixels fall somewhere in between, with light-gray being closer to 255. We'll soon see that color images have similar numerical representations for each pixel color.



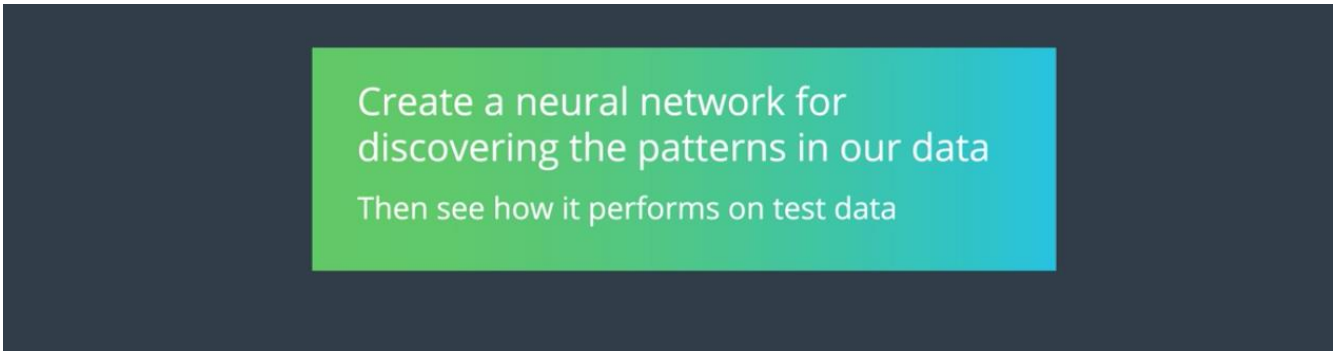
These MNIST images have actually gone through a quick pre-processing step. They've been re-scaled so that each image has pixel values in a range from zero to one, as opposed to from 0-255. To go from a range of 0-255 to zero to one, you just have to divide every pixel value by 255. This step is called normalization, and it's common practice in many deep learning techniques. Normalization will help our algorithm to train better. The reason we typically want normalized pixel values is because neural networks rely on gradient calculations. These networks are trying to learn how important or how weighty a certain pixel should be in determining the class of an image. Normalizing the pixel values helps these gradient calculations stay consistent, and not get so large that they slow down or prevent a network from training. So, now we have a normalized data, how might we approach the task of classifying these images? Well, you already learned one method for classification, using a multi-layer perceptron. How might we input this image data into an MLP? Recall that MLPs only take vectors as input. So, in order to use an MLP with images, we have to first convert any image array into a vector. This process is so common that it has a name, flattening.



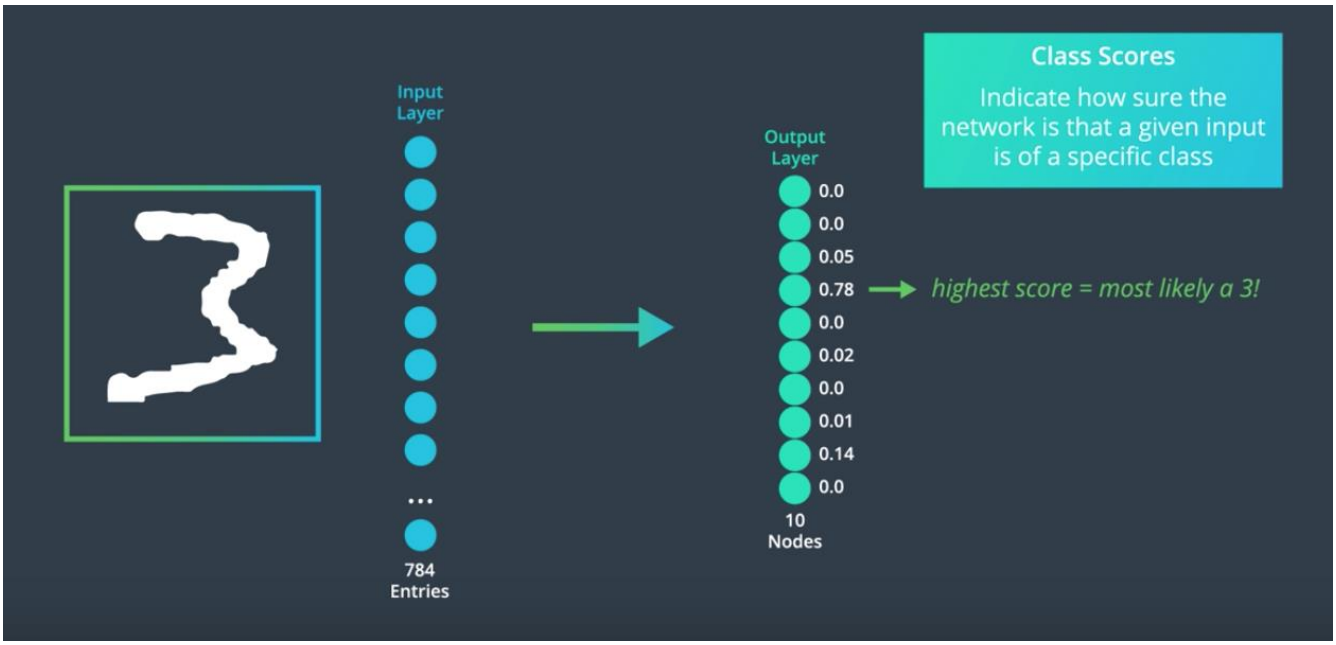
I'll illustrate this flattening conversion process on a small example here. In the case of a four-by-four image, we have a matrix with 16 pixel values. Instead of representing this as a four-by-four matrix, we can construct a vector with 16 entries, where the first first four entries of our vector correspond to the first wheel of our old array. The second four entries correspond to the second wheel and so on.



After converting our images into vectors, they can then be fed into the input layer of an MLP. We'll see how this works and see how to produce a class prediction for any given image in the next few videos.

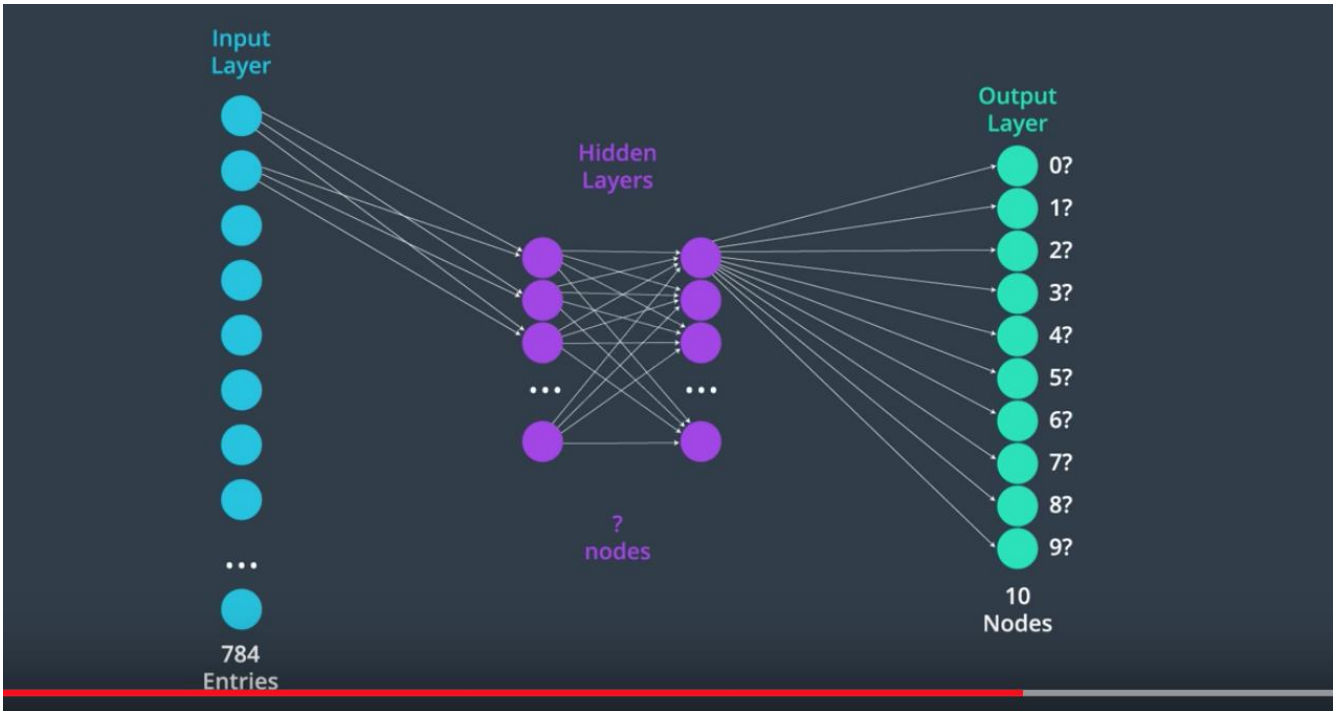


98. After looking at and normalizing our image data, we'll then create a neural network for discovering the patterns in our training data. After training, our network should be able to look at totally new images that it hasn't trained on, and classify the digits contained in those images. This previously unseen data is often called test data.

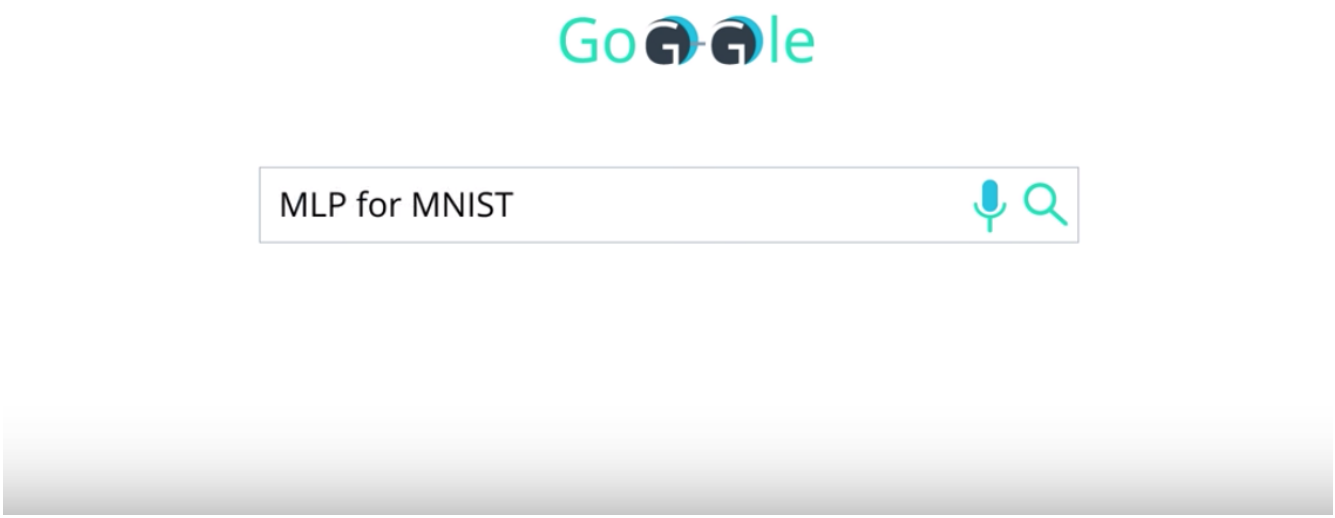


At this point, our images have been converted into vectors with 784 entries. So, the first input layer in our MLP should have 784 nodes. We also know that we want the output layer to distinguish between 10 different digit types, zero through nine. So, we'll want the last layer to have 10 nodes. So, our model will take in a flattened image and produce 10 output values, one for each possible class, zero through nine. These output values are often called class scores. A high class score indicates that a network is very certain that a given input image falls into a certain class. You can imagine that the class score for a 3, for example, will have a high score for the class three and a low score for the classes zero, one, and so on. But it may also have a small score for an eight or any other class that looks kind of similar in shape to a 3.

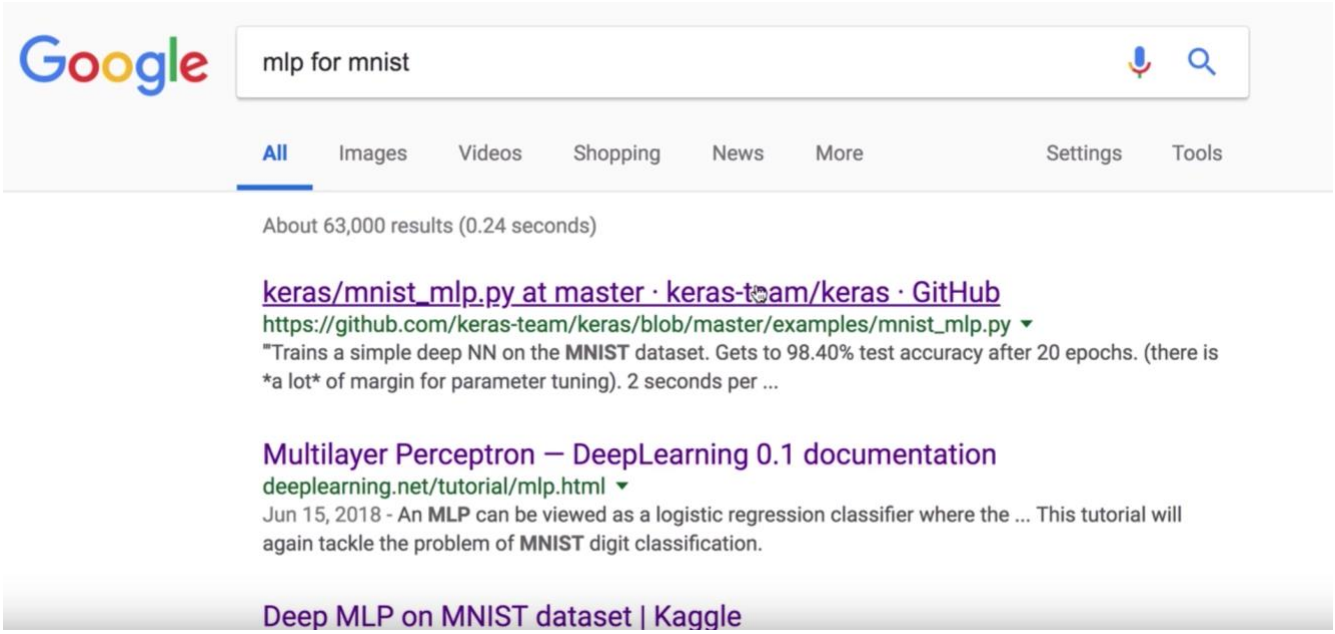
The class scores are often represented as a vector of values or even as a bar graph indicating the relative strengths of the scores.



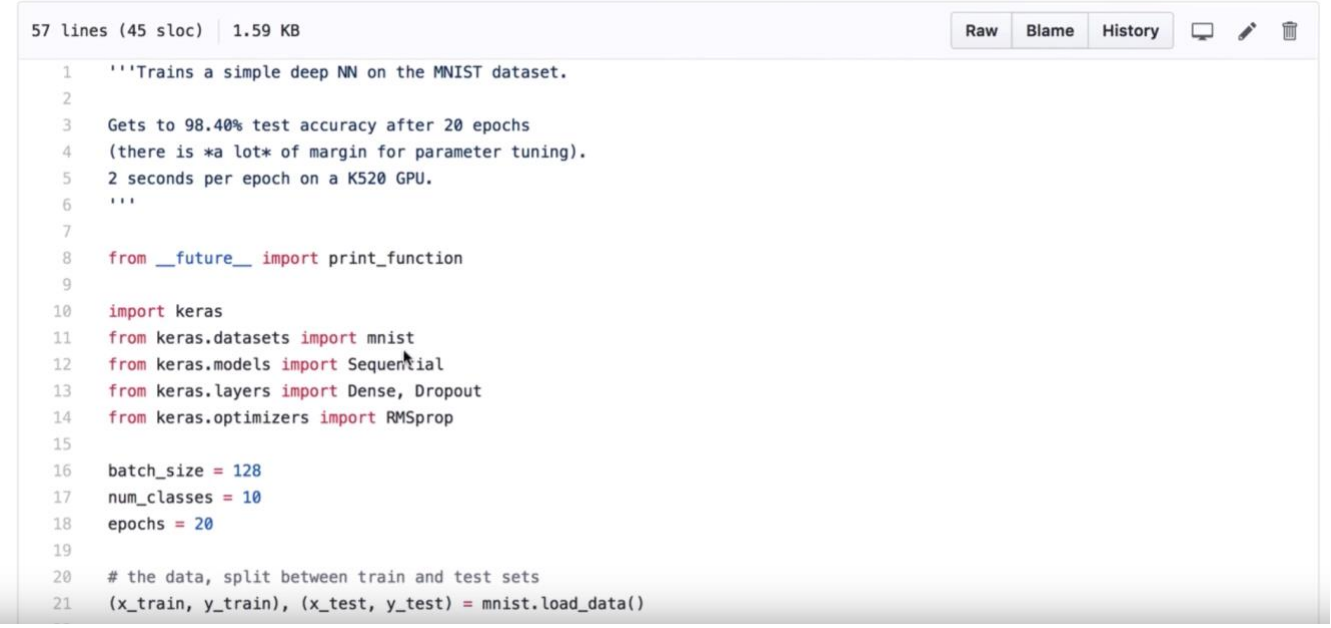
Now, the part of this MLP architecture that's up to you to define is really in between the input and output layers. How many hidden layers do you want to include and how many nodes should be in each one? This is a question you'll come across a lot as you define neural networks to approach a variety of tasks. I usually start by looking at any papers or related work I can find that may act as a good guide.



In this case, I would search for MLP for MNIST, or even more generally, MLP for classifying small greyscale images. Next, I'm going to ask you to perform a search like this and see if you can find a good place to start when it comes to defining the hidden layers of an MLP for image classification.



99. When I type in mlp for mnist into Google, a few things pop up, including a couple of code implementations.

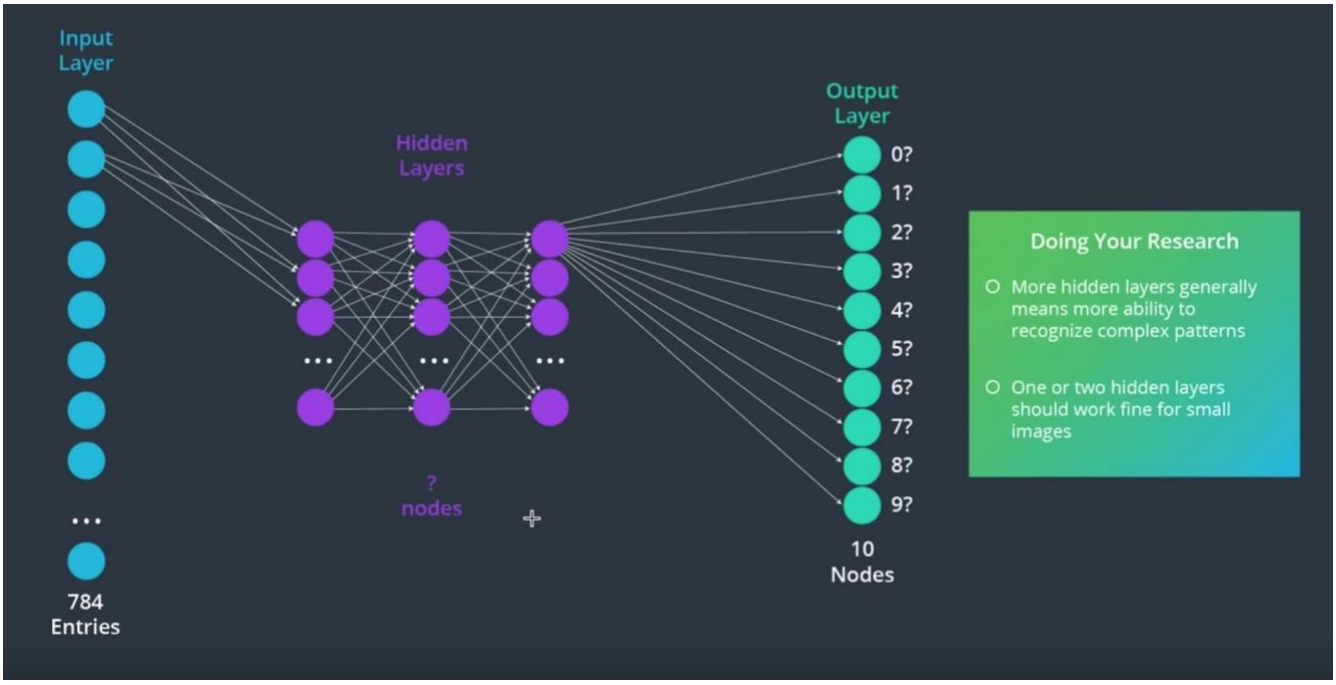


This first one from the official keras GitHub repository looks like a reputable source.

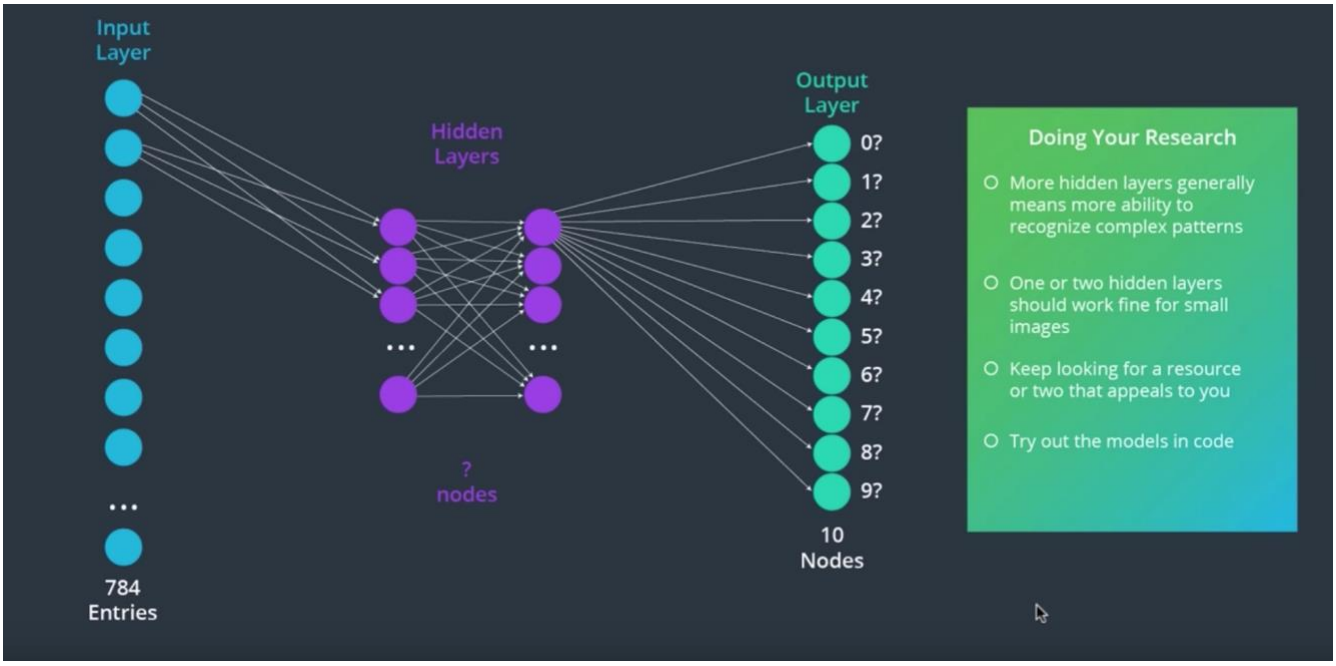
Scrolling through this code, I can see that they've imported the MNIST dataset and that they flattened each input image into a vector of size 784. Here, it looks like they're using one hidden layer to convert that input to be 512 nodes. We can also see an activation function being applied, a relu function, and we have one more hidden layer also with 512 nodes. We can also see some

Dropout layers in between. This 0.2 means they have a 20 percent probability of Dropout or of a node being turned off during a training cycle. As you've learned, Dropout layers are used to avoid overfitting data.

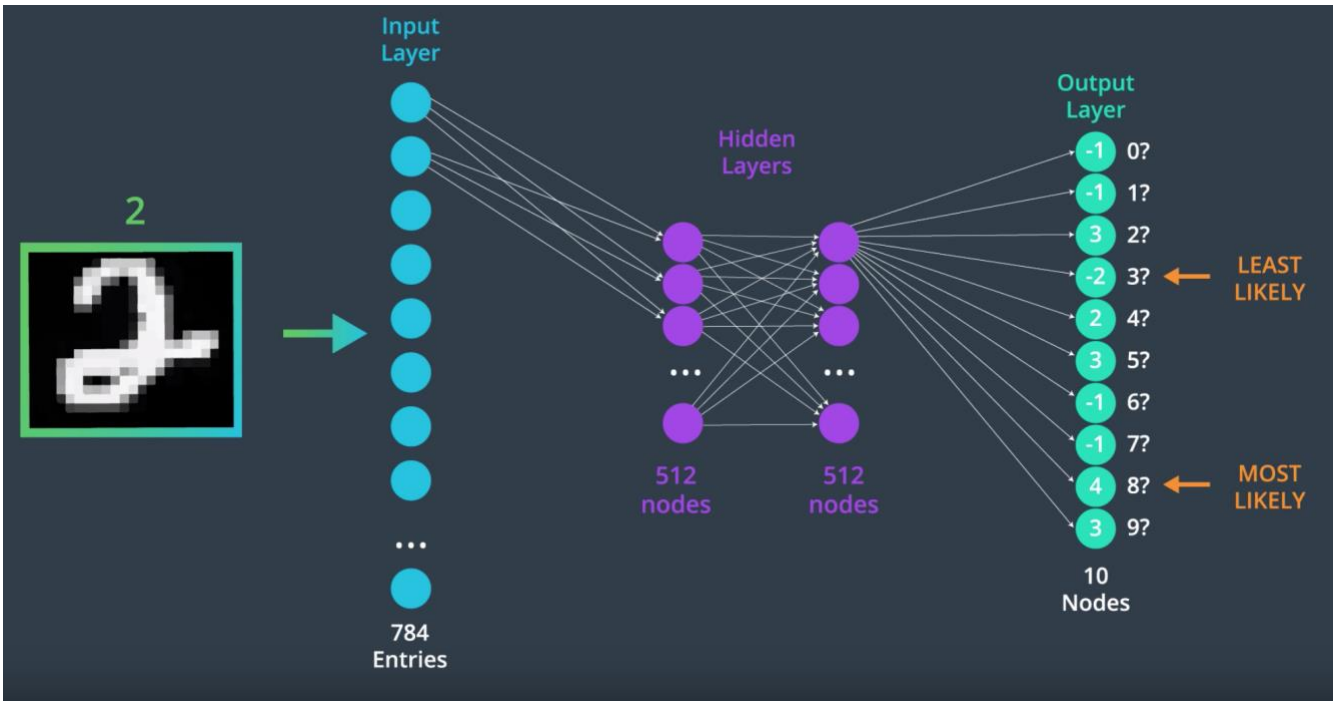
Lastly, I see one more fully connected layer that's producing an output vector of length 10 for a number of classes. When I look at a source like this, I first try to think about whether it makes sense.



I know that the more hidden layers I include in the network, the more complex patterns this network will be able to detect, but I don't want to add unnecessary complexity either.



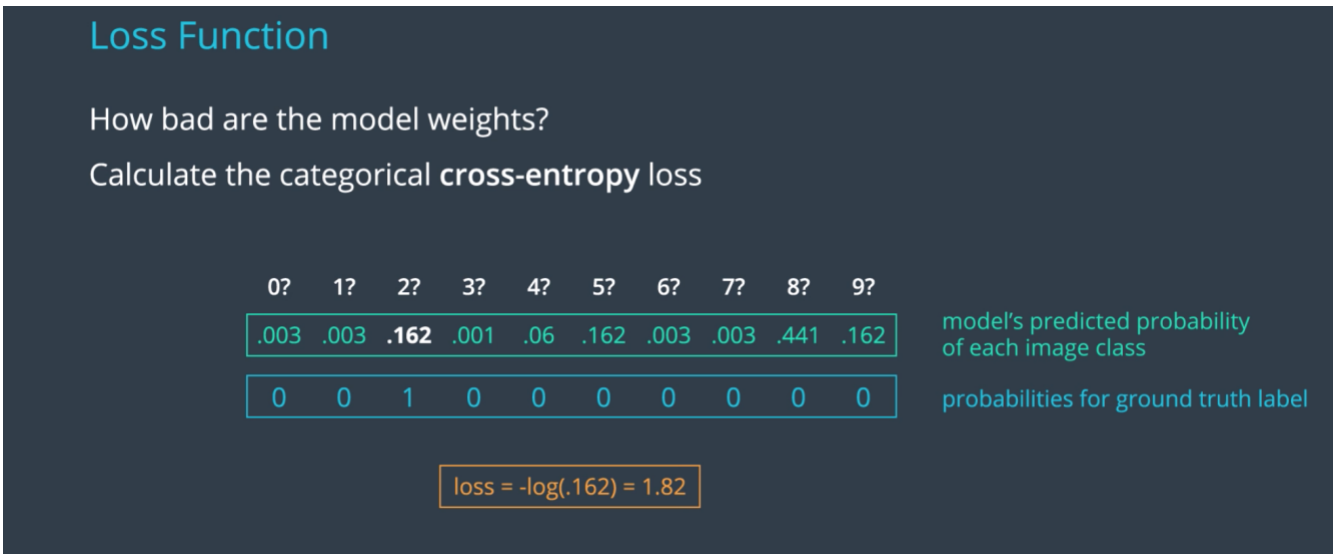
My intuition is telling me that for small images, two hidden layers sounds very reasonable. Now, this is just one solution to the task of handwritten digit classification. The next step in approaching a problem like this is to either, one, keep looking and see if you can find another structure that appeals to you; then two, when you do find a model or two that look interesting, try them out in code and see how well they perform. This model that I found is good enough for me, so I think I'm going to proceed with defining an MLP with two hidden layers based on this structure.



100. Now that we've decided on the structure of our MLP, let's talk a bit about how this entire thing will actually learn from the MS data. What happens when it actually sees an input image. Take this input image of a two for instance. Say we feed this image into the network and when we do so, we get these ten class scores for my output layer. Again, a higher score means that the network is more certain that the input image is of that particular class. Four is the largest value here and negative two is the smallest. So, the network believes that the image input is least likely to be a -2 and most likely to be an 8, but this is incorrect. We can see that the correct label is two. So, what we can do is tell our network to learn from this mistake.

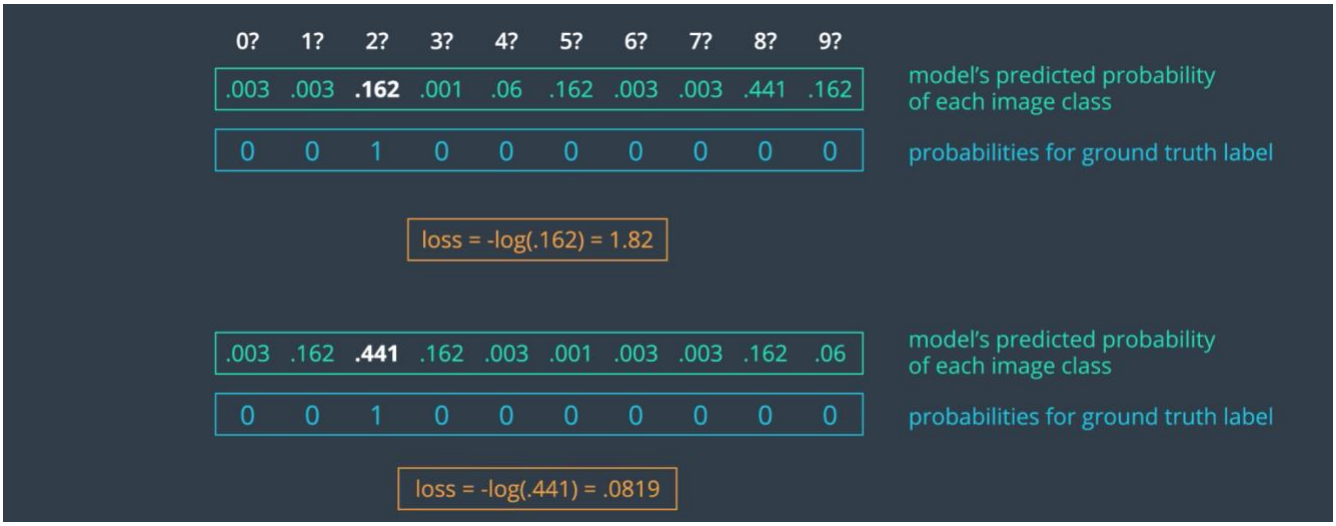
As a network trains, we measure any mistakes that it makes using a loss function, whose job is to measure the difference between the predicted and true class labels. Then using back propagation, we can compute the gradient of the loss with respect to the models' weights. In this way, we quantify how bad a particular weight is and find out which weights in the network are responsible for any errors. Finally, using that calculation, we can choose an optimization function like gradient descent to give us a way to calculate a better weight value.

Towards this goal, the first thing we'll need to do is make this output layer a bit more interpretable. What's commonly done is to apply a softtmax activation function to convert these scores into probabilities. To apply a softmax function to this output layer, we begin by evaluating the exponential function at each of the scores, then we add up all of the values. Let's denote this sum with a capital "S". Then we divide each of these values by the sum. When you plug in all of the math, you get these 10 values.

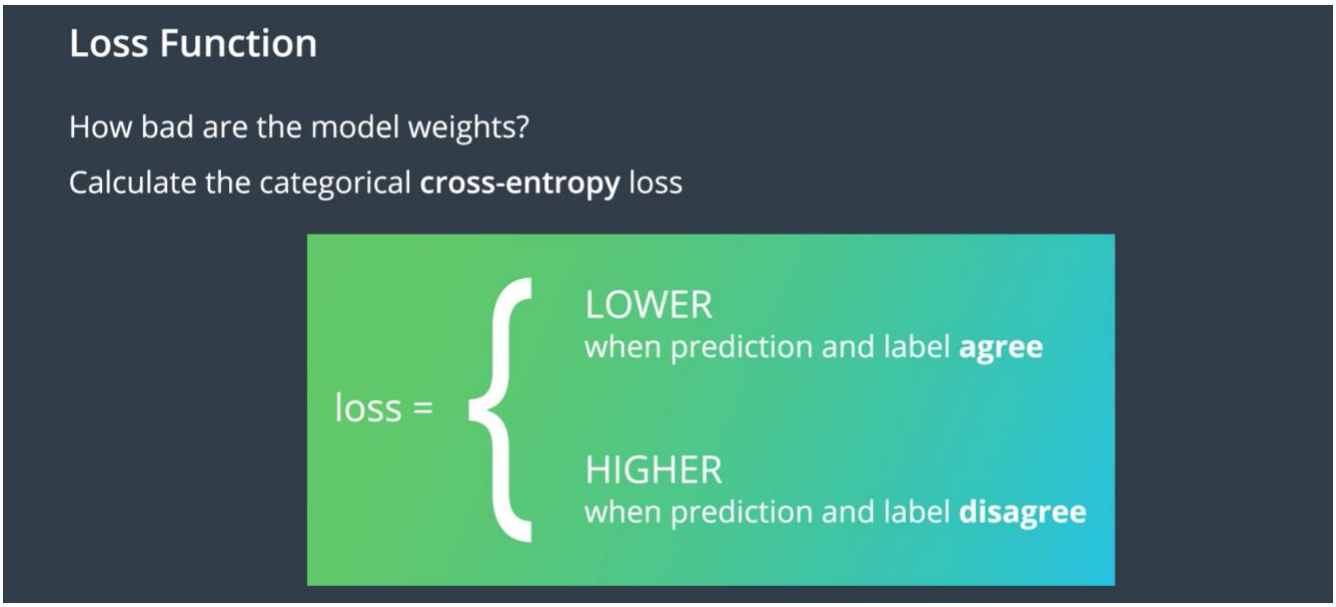


Now each value yields the probability that the image depicts its corresponding image class. For instance, the network believes that the image shows an eight with 44.1% probability. Remember

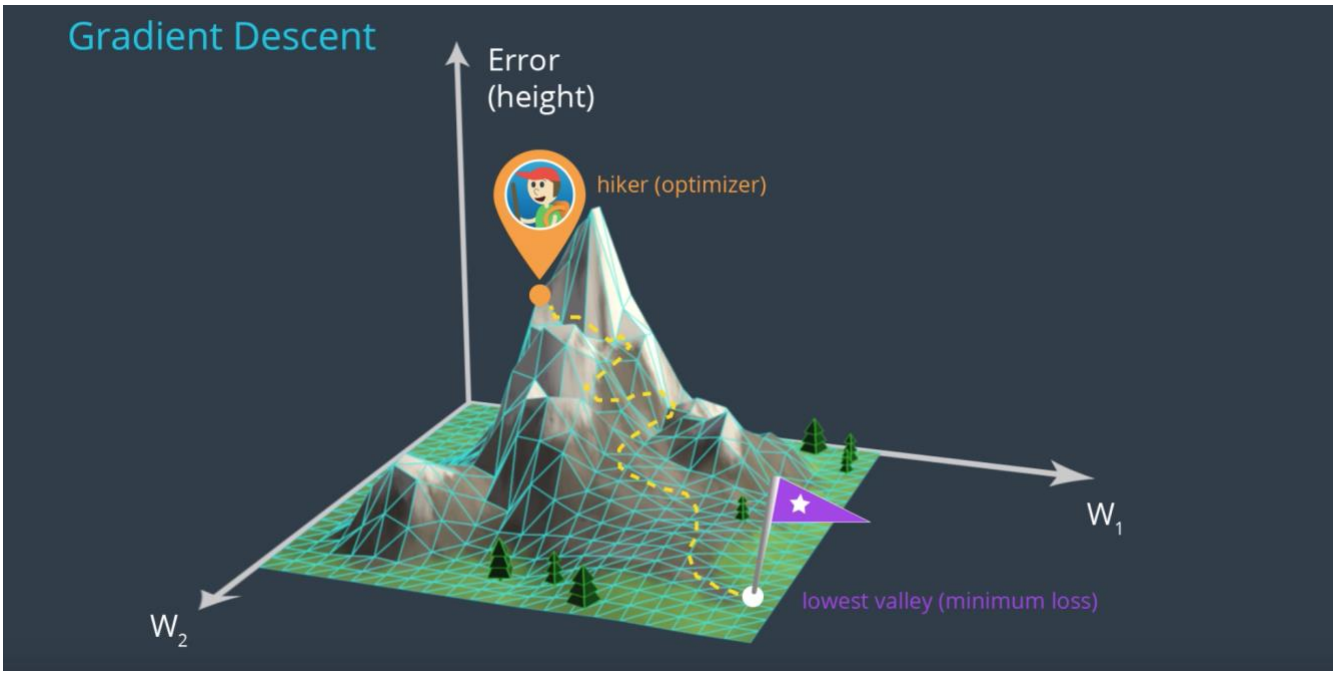
that the input to the network was an image of a handwritten 2 and yet the network incorrectly predicts that the image shows a two with only 16.2 percent probability. Now our goal is to update the weights of the network in response to this mistake, so that next time it sees this image, it predicts that 2 is the most likely label. In a perfect world, the network would predict that the image is 100 percent likely to be the true class. In order to get the model's prediction closer to the ground truth, we'll need to define some measure of exactly how far off the model currently is from perfection. We can use a loss function to find any errors between the true image classes and our predicted classes, then backpropagation will find out which model parameters are responsible for those errors. Since we're constructing a multi-class classifier, we'll use categorical cross entropy loss. To calculate the loss in this example, we begin by looking at the model's predicted probability of the true class which is 16.2 percent. Cross entropy loss looks at this probability value which in decimal form is 0.162 and takes the negative log loss of that value. In this case, we get negative log 0.162 or 1.82.



Now for argument's sake, say instead that the weights of the network were slightly different. The model instead returned at these predicted probabilities. This prediction is much better than the one above and when we calculate the cross entropy loss, we get a much smaller value.

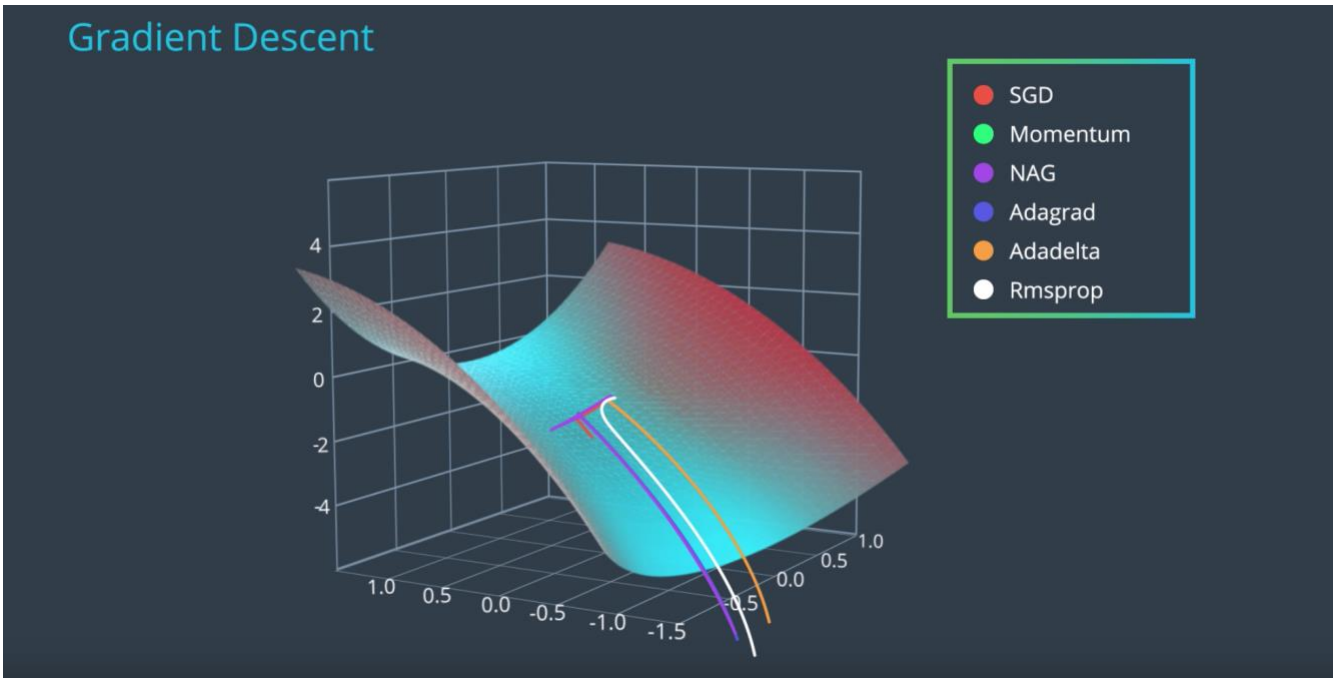


In general, it's possible to show that the categorical cross entropy loss is defined in such a way that the loss is lower when the model's prediction agrees more with the true class label, and it's higher when the prediction and the true class label disagree. As a model trains, its goal will be to find the weights that minimize this loss function and therefore give us the most accurate predictions. So, a loss function and backpropagation give us a way to quantify how bad a particular network weight is, based on how close a predicted and the true class label are from one another. Next, we need a way to calculate a better weight value.



In the previous lesson, you were encouraged to think of the loss function as a surface that resembles a mountain range. Then to minimize this function, we need only find a way to descend to the lowest

value. This is the role of an optimizer. The standard method for minimizing the loss and optimizing for the best weight values is called "Gradient Descent".



You've been introduced to a number of ways to perform gradient descent and each method has a corresponding optimizer. The surface depicted here is an example of a loss function and all of the optimizers are racing towards the minimum of the function. As you can see, some do better than others and you're encouraged to experiment with all of them in your code.

Not_read_begin

notebook-mnist_mlp_exercise-first-time-starts

101. You've learned a lot about how to approach the task of image classification. In this notebook, I'll go over how to load in image data, define a model, and train it. You may already be familiar with this process from the earlier lesson on deep learning with PyTorch, but I'd encourage you to stay tuned just so you can see another example in detail. I'll present this code as an exercise. Then you'll get a chance to change this code, define your own model, and try it out on your own in a Jupyter Notebook, much like this one. So, the first thing I've done in this notebook is to load in the necessary torch and NumPy libraries. Next, I'm going to import the torchvision datasets and transformation libraries. I'll use these to actually load in the MNIST dataset and transform it into a Tensor datatype. Here transforms.ToTensor is where I define that transformation. The tensor datatype is just a data type that's very similar to a NumPy array, only it can be moved onto a GPU for faster calculation, which you'll learn more about later. You'll also see that I've set some parameters for loading in the image data. I can define the batch size which is the number of training images that will be seen in one training iteration, where one training iteration means one time that a network make some mistakes and learn from them using back propagation. The number of workers is if you want to load data in parallel. For most cases, zero will work fine here. Now I'm

going to load in training and test data using datasets.MNIST. I'm going to download each set and I'm going to transform it into a tensor datatype which I defined here. I'll put the downloaded data into a directory named data. Finally, I'm going to create training and test loaders. These loaders taking our data that we defined above, our batch size and number of workers. The train and test loaders give us a way to iterate through this data one batch at a time. Downloading the data may take a minute or two. After I've downloaded my data, I do the first step in any image analysis task, I visualize the data. Here I'm grabbing one batch of images and their correct labels and I'm plotting 20 of them. Here you can see a variety of MNIST images and their labels. This step allows me to check and make sure that these images look how I expect them to look. I can even look at an image in more detail. Here I'm just looking at one image in our set and I'm displaying the gray scale values. You can see that these values are normalized and the brightest pixels are close to a value of one. The black pixels have a value of zero. Next comes the really interesting part which is defining the MLP model. We've talked about defining the input hidden and output layers and so I'll leave most of this section for you to fill out. I do want to point out a couple of things here. First the init function. To define any neural network in PyTorch you have to define and name any layers that have learned weight values in the Init function, in this case, any fully-connected linear layers that you define. I've defined a sample first input layer which I've named FC1 for you. This has 784 or 28 by 28 inputs and a number of hidden nodes. This is the number of outputs that this layer will produce. For now, I've left this as one and this will have to be changed to create a working solution. Next, you have to define the feedforward behavior of your network. This is just how an input X will be passed through various layers and transformed. I'm assuming the past in X will be a grayscale image like an MNIST image and I've provided some starting code in here for you. First, I've made sure to flatten the input image X by using the view function. View takes in a number of rows and columns and then squishes and input into that desired shape. In this case, the number of columns will be 28 by 28 or 784 and by putting a negative one here, this function will automatically fit all of the x values into this column shape. The end result is that this X will be a vector of 784 values. Then I'm passing this flattened vector to our first fully-connected layer defined up here. I just call this layer by name, pass in our input and apply a relu activation function. A relu should be applied generally to the output of every hidden layer so that those outputs are consistent positive values and finally I return the transformed X. Now to complete this model, you should add to the init end forward functions so that the final returned X is a list of class scores. Next, I'm going to describe how you might go about training your defined model and complete this task on your own.

102. After loading in data and defining a model, the next task is to define your loss and optimization functions. For a simple classification problem like this one, I recommend starting with cross-entropy loss and a method of gradient descent. But you may also find it useful to read the documentation on the various types of loss functions and optimizers. In fact, let's check out the loss documentation. You can see a few different types of loss here, and we can read about cross-entropy loss. The documentation says that the cross-entropy function is actually performing a softmax function on an output layer and then doing negative log loss. This means that you only need your model to produce class scores, and then this loss function will turn those into probabilities using a softmax function and calculate the loss, even details how these calculations are performed. You'll also notice that this loss is calculated as an average value over a batch of images. So, if a batch of 20 images is seen by our model during training, the return loss will be the average loss over those 20 images. Now back to our notebook, the loss and optimization functions will really define how the network updates its weights as it trains. Next, we'll get to the actual

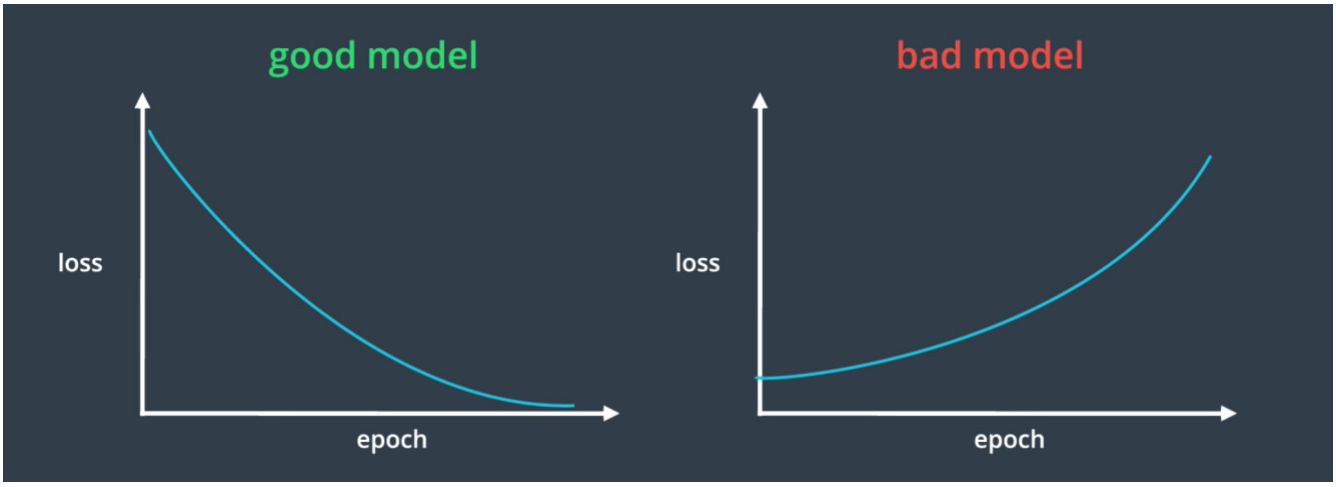
training loop, this as the number of epochs we will train for. Epochs are how many times we want the model to iterate through the entire training dataset. One epoch for example means it sees every training image just once. And I suggest starting with at least 20 epochs for this dataset. It's good practice to start small if you're testing out different model architectures and then increase the number of epochs when you're trying to train your final model. Then, we loop over each epoch and I'm going to keep track of the training loss as I go. Inside this epoch loop is the batch loop. Here, the train loader will load a batch of training data and we can look at the images and the true labels for every image in that batch. At the start of the batch loop, we clear out any gradient calculations that PyTorch has accumulated using optimizer.zero_grad. Then, we call in our model and perform a forward pass. Our model takes in the input images, the data from our train loader and then this returns the predicted class scores, which I'll call output. Next, we use our defined loss function to compare these predicted outputs and the true labels, the target. These were again gotten from our train loader. So, we compare a batch of outputs and target labels and calculate the cross-entropy loss. To complete the backpropagation steps, we then perform a backward pass to compute the gradient of the loss and we perform a single optimization step. This step is actually responsible for updating the values of the weights in our network. Finally, we compute a running training loss. This last item is a loss value that's averaged over the batch and in this case we actually want to record the accumulated loss over the batch, not the average quite yet. So, I'm going to multiply it by the batch size data.size(0). Then, after this loop has completed and we've gone through one whole epoch, I'll calculate the average loss over that epoch. To do this, I'm going to divide the accumulated loss by the total number of images in our training set. This returns an average training loss and I'll print it out after each epoch. I should note that this is just one way to calculate the average loss. And as you look at example code, you may see a different calculation. For example, you could calculate the average loss at any point in this batch loop, as long as you add up the loss for every input image and then divide the accumulated loss by the number of images seen by the network. Now, after you run this training loop for some time after a number of epochs, you'll be able to test your model on previously unseen test data. I've provided some code that iterates through the test data from our test loader, and applies our trained model to that data. This code will allow you to see how well your model performs on each class in our dataset. Next, you'll have a chance to take a look at this exercise code and a solution to creating an MLP for image classification. I encourage you to look at the solution after you've attempted to build a network of your own.

103. I'm going to quickly show you how I went about defining and training an MLP for image classification. So first, I just loaded in and looked at my normalized image data. Then, I started to define the classification model. I started with the fully connected layer FC1 that sees the 784 entry long vector that represents a flattened image as input. Then, I proceed it based on the resource that I found, which had two hidden layers with 512 inputs and outputs. I've actually stored those values here as two variables, hidden_1 and hidden_2. This is an extra step, but it makes it easy should I want to change these values later on for testing or something else. So, to complete my first fully connected layer, I put hidden_1 as the number of outputs I want this to produce. Then, I create a second fully connected layer, FC2, which takes in that number of outputs and produces 512 outputs again. I also want to be clear that the outputs of one layer feed into the inputs of the next. Then, I've defined the last fully connected layer FC3, such that it sees 512 inputs and produces 10 outputs. The 10 outputs correspond to the number of digit classes zero through nine. So, this layer will produce our class scores. Finally, still in the init function, I've also defined a dropout layer with a dropout probability of 0.2 or 205. Now that I've defined all the layers that I need to make up this

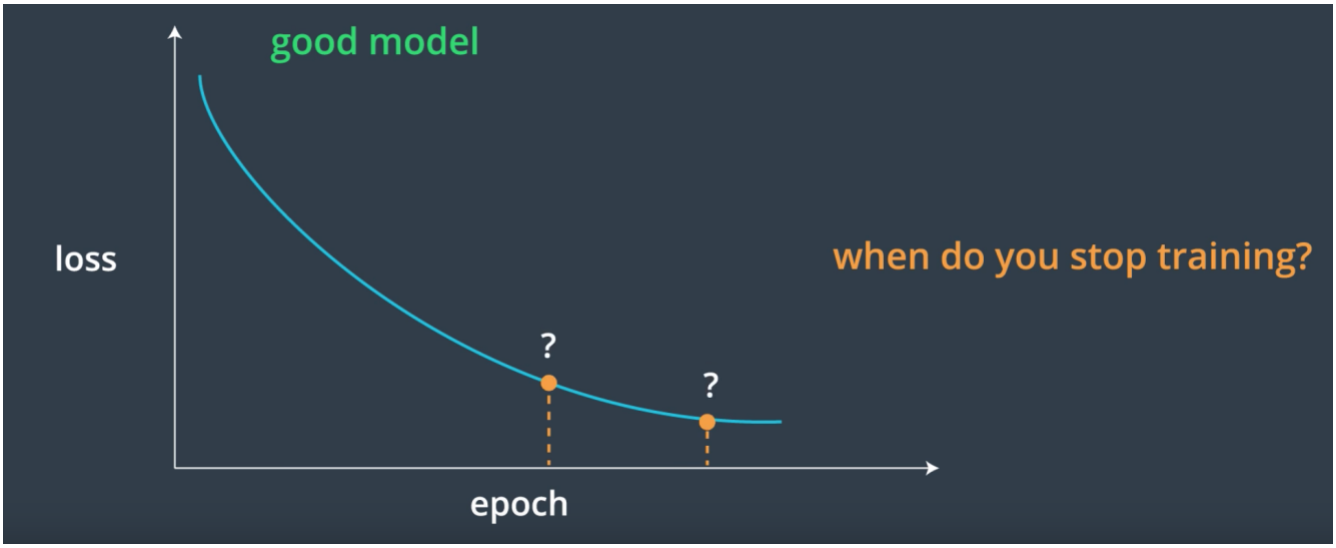
classification MLP, I have to explicitly define the feedforward behavior of the network. With the forward function, I basically want to answer how will an input vector proceed through these layers? So, the input x starts out as a 28 by 28 image tensor, and my first step is to flatten it into a 784-length vector. Once the input is flattened, I'm passing it through the first fully connected layer, FC1, and applying an activation function. Next, I'm going to do the exact same thing with our second fully connected layer, FC2. Only in-between these two layers, I'm actually adding a dropout layer which would help prevent over-fitting. After x passes through our second hidden layer, we're going to add one more dropout layer and reach our final fully connected layer. You'll notice that I'm not applying an activation function to this final layer. That's because later it will have a softmax activation function applied to it. So right now, I am leaving it as is and returning the final transformed x. Because FC3 produces 10 outputs, this x should represent our 10 class scores. Okay. Then, I'm instantiating and printing out the net to make sure it has the layers that I want. I should see our three linear layers and our dropout. The next step is defining the loss and optimization functions. Here, I'm defining the last criterion as cross-entropy loss. This is just a pretty standard loss for any classification task. Then, I'm going to use stochastic gradient descent, SGD, from the torch optimization library. This takes in our model parameters and a learning rate. I've set a learning rate of 0.01. If you find that your loss is decreasing too slowly or sporadically, you can change this value. Next, I spent some time training this model for 50 epochs. This took some time because I'm just using my CPU for now, and later I'll show you how to use a GPU for faster training. At the end of each of these epochs I printed out the training loss and watched how it decreased over time. You can see that it decreased fairly quickly at first and then later on slows down especially around the 40 epoch mark. But it is still decreasing up to epoch 50. Then, to actually see how well my trained model generalizes to new data, I tested on our test data that we loaded in at the start. Here, I've iterated through all the data in our test litter. I applied the model and recorded the test loss. Recall that our model is actually returning a list of class scores. To isolate our predicted class, I'm going to take the maximum value of the scores and return it as our prediction. Then, I compare this prediction to our target label. This creates a list whether a certain prediction was correct or not, then I actually separate these into the 10 classes and I print out the accuracy for each class. I have our overall test loss and our overall accuracy which is 97%. This is pretty good, and you can see among all the digit classes this value is pretty consistent. The model seems to do the worst on images of the number seven or eight. But an even distribution indicates that your model is trained pretty evenly on each type of data. I've also included a cell, where you can display test images and the predicted and true labels side-by-side. This makes it easy to see if you've gotten any specific image wrong. But going back up to the test accuracy overall, I actually wonder if I could do even better. If I could improve this model by adding, say, another layer to find more patterns in the data. I even wonder if I chose the right point to stop training this model. In fact, one thing to note is that I stopped training at 50 epochs based on only how I expected the loss to decrease over time. But it's more of an art than a science at this point. So next, I'll talk about one more concrete method of knowing when to stop training. A technique called model validation.

notebook-mnist_mlp_exercise-first-time-ends

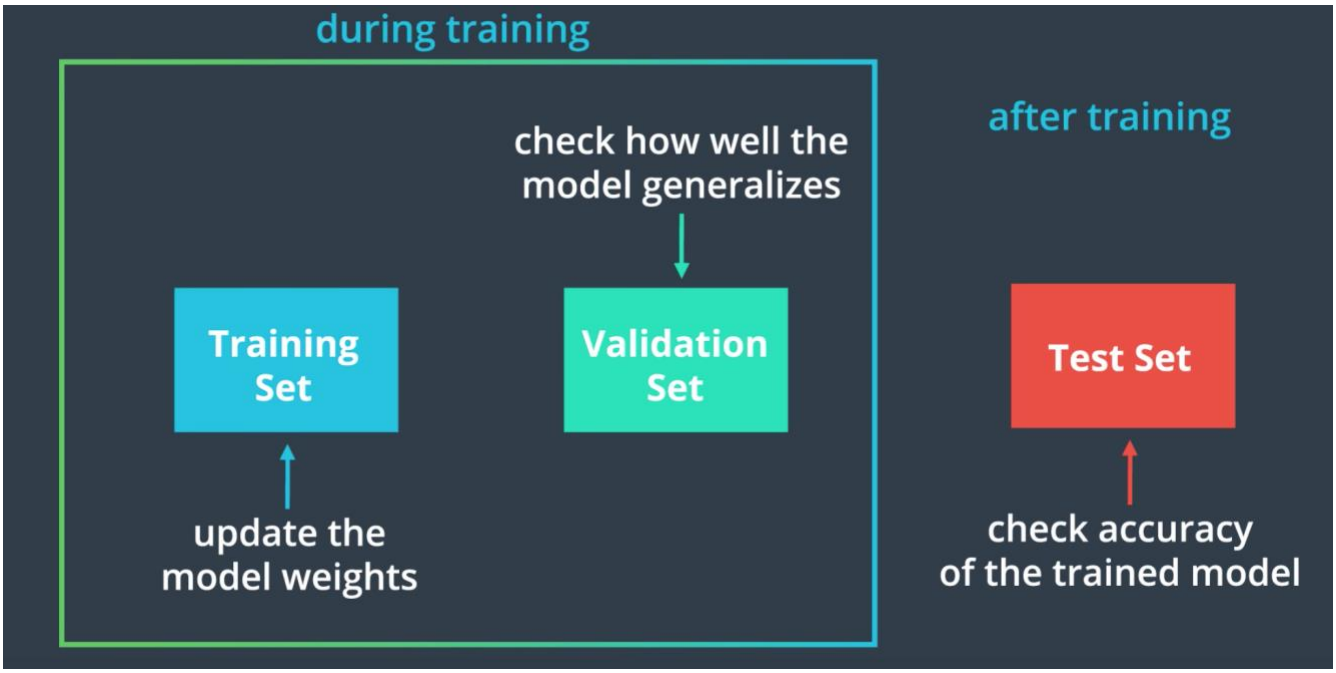
Not_read_end



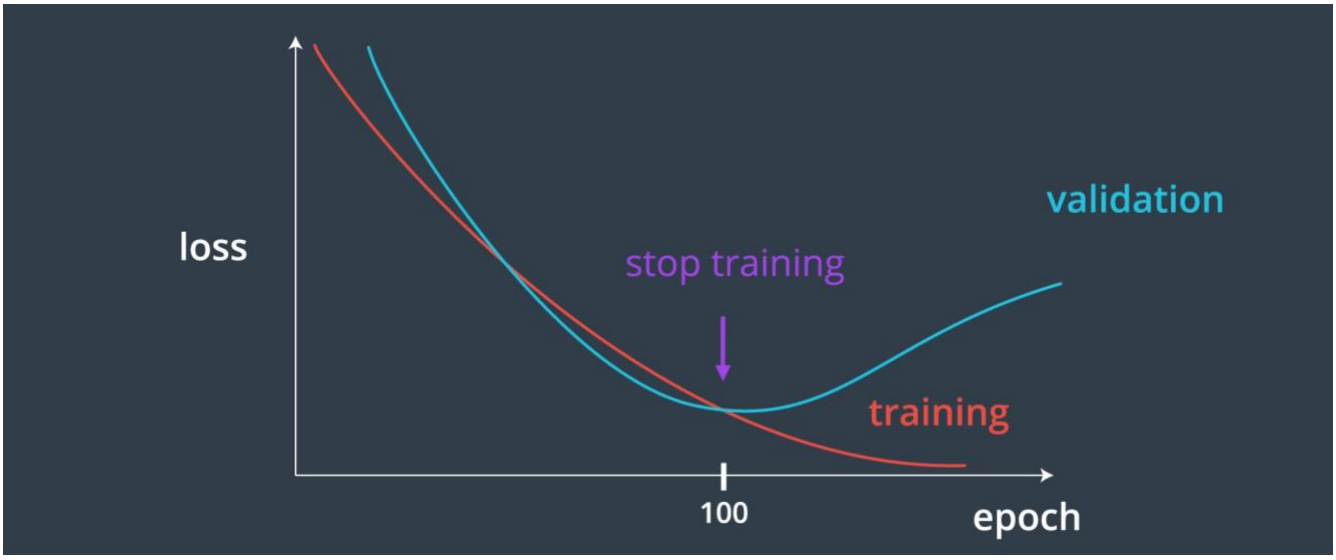
104. So far, we've checked how well our models are performing based on how the loss changes over each epoch.



But the exact number of epochs to train for is often hard to determine. How many epochs should you train for so that your network is accurate but it's not overfitting the training data?

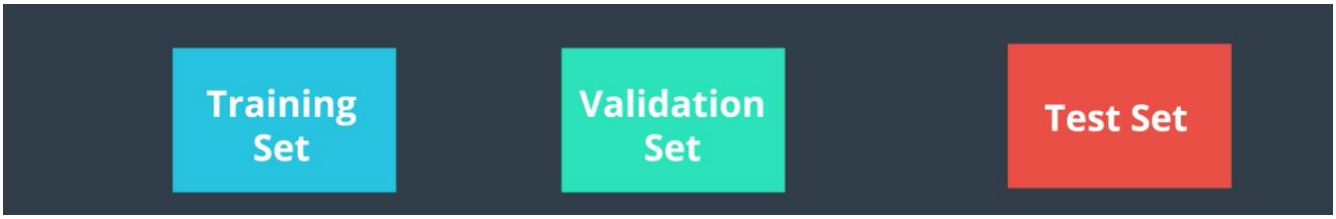


One method that's used in practice involves breaking the dataset into three sets called training validation and test sets. Each is treated separately by the model. The model looks only at the training set when it's actively training and deciding how to modify its weights. After every training epoch, we check how the model is doing by looking at the training loss and the loss on the validation set. But it's important to note that the model does not use any part of the validation set for the back propagation step. We use this training set to find all the patterns we can, and to update and determine the weights of our model. The validation set only tells us if that model is doing well on the validation set. In this way gives us an idea of how well the model generalizes to a set of data that is separate from the training set. The idea is, since the model doesn't use the validation set for deciding its weights, that it can tell us if we're overfitting the training set of data. Finally, the test set of data is saved for checking the accuracy of the model after it's trained.



This may be easiest to see in an example. Say it's at the model to train for 200 epochs, and around epoch 100, you notice that the training loss starts decreasing but the validation loss is actually starting to increase. This actually indicates that the model is overfitting the training data because it's not generalizing well enough to also perform well on the validation set. So, if you see this divide and how the training and validation losses decrease, you'll want to stop changing the weights of your network around epoch 100 and ignore or throw away the weights from later epochs where there's evidence of overfitting.

This process can also prove useful if we have multiple potential architectures to choose from. For example, if you're deciding on the number of layers to put in the model, then you'll want to save the weights from each potential architecture for later comparison, and can choose to pick the model that gets the lowest validation loss.



You might be wondering why must we create a third dataset. Couldn't we just use the test set for this purpose? Well, the idea is that when we go to test the model, it looks at data that it has truly never seen before. Even though the model doesn't use the validation set to update its weights, our model selection process is based on how the model performs on both the training and validation sets. So, in the end, the model is biased in favor of the validation set. Thus, we need a separate test set of data to truly see how our selected model generalizes and performs when given data it really has not seen before. In the next video, I'll show you how to use these ideas to fine tune your network architecture

Not_read_begin

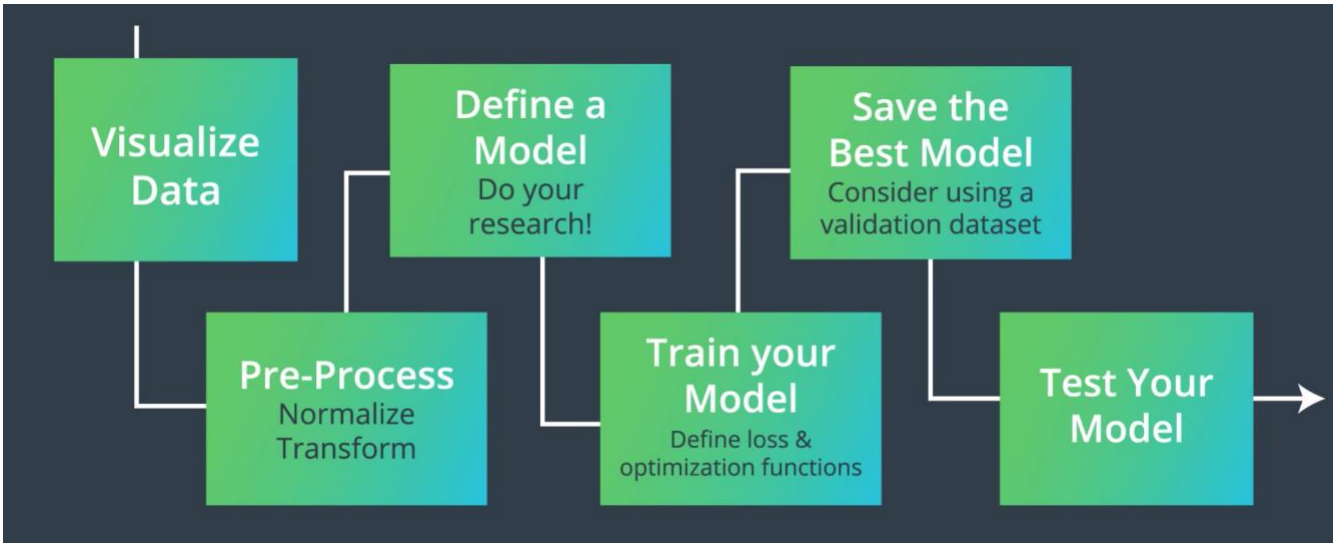
notebook-mnist_mlp_exercise-second-time-starts

105. So, in our last example, I got pretty good classification accuracy just by looking at how during training the cross entropy loss that measured the difference between predicted and true classes, got smaller and smaller. I estimated when a good time to stop training would be, when the training loss had plateaued it's decreasing. But as you just learned, there's a way to use a validation dataset to programmatically know when to stop training. So, I'm going to show you how to add that encode. So, the first thing I'm going to do, is to create a validation dataset much like we created training and test sets. In fact, I'm actually going to take a percentage of data from the training set. I'll set a variable valid size to take 20 percent of the training set data and turn it into validation data. There should be a good enough size since the MNIST dataset is very large. Then I'm going to use something called, subset random sampler to help me do the work of splitting the training data. First, I'll record how many training images there are, and I'll determine which indices in the training set I'll access to create both sets training and validation. I'll list out all the possible indices by grabbing the length of the entire training set. So, these indices are going to be the values that point to each of the 70,000 images in the training set. Then, I'm going to shuffle these indices so that any index I select out of this list, will reference a random piece of data. Then, I'm defining a split boundary, and this is just going to be the number of examples that I want to include in the validation set. So, it's going to be 20 percent of our training data. Then I'll use this to get an 80-20 split between training and validation data. Finally, I'm using subset random sampler to create data samplers for this training and validation data. This adds one more argument to our train loader and validation loaders. So, previously, I had only training and test data loaders, and now I've split the training data into two sets by essentially shuffling it and selecting 20 percent for validation set using a specific data sampler. So, now, we officially have a validation set loader and I'm going to scroll down to my training loop to actually use this validation set. Okay. Here's our training loop and this time in addition to keeping track of the training loss, I'm also going to track the validation loss. I'll actually want to start training whenever I reach an epic for which the training loss decreases but the validation loss does not. So, I'm going to track the change in the validation loss, and specifically, I'll track the minimum validation loss over time, so that I can compare it to the current validation loss and see if it's increased or decreased from the minimum over a given epic. So, within our epic loop, we have our usual training batch loop and we also have a validation batch loop. This is looping through all the data and labels in the validation set. It's applying our model to that data and recording the loss as usual. Notice, we're not performing the back propagation step here. That is reserved only for our training data. Then, I've added a little bit to my print statement and I'm printing out the average validation loss after each epic as well. Also at the end of each epic, I'm going to check the validation loss and see if it's less than the currently recorded minimum. If it is, I'm going to save the model because that means the validation loss has decreased, and I'll store that as the new minimum value. You may have noticed that I set the initial minimum to infinity. This high-value guarantees that this loss will update after the first epic. Also take note of this line which allows me to save the model and its current parameters by name model.pt. Okay. Then I've run this for 50 epics and I can see both the training loss and the validation loss after each epic. We can see that both the training and validation loss are decreasing for the first 30 or so epics. So, the model is being saved after each of these points where the validation loss decreases. We can also see this decrease slowing down right around here. In fact, our last model was saved after epic number 37. By saving the model at the point where the validation and training loss diverge am preventing my model from overfitting the training data, this is also an issue of efficiency. We should see that our validation loss stays the same for the last 10 to 15 epics here. So, the lack of decrease here is actually indicating to me that the best model is really reached even around epic

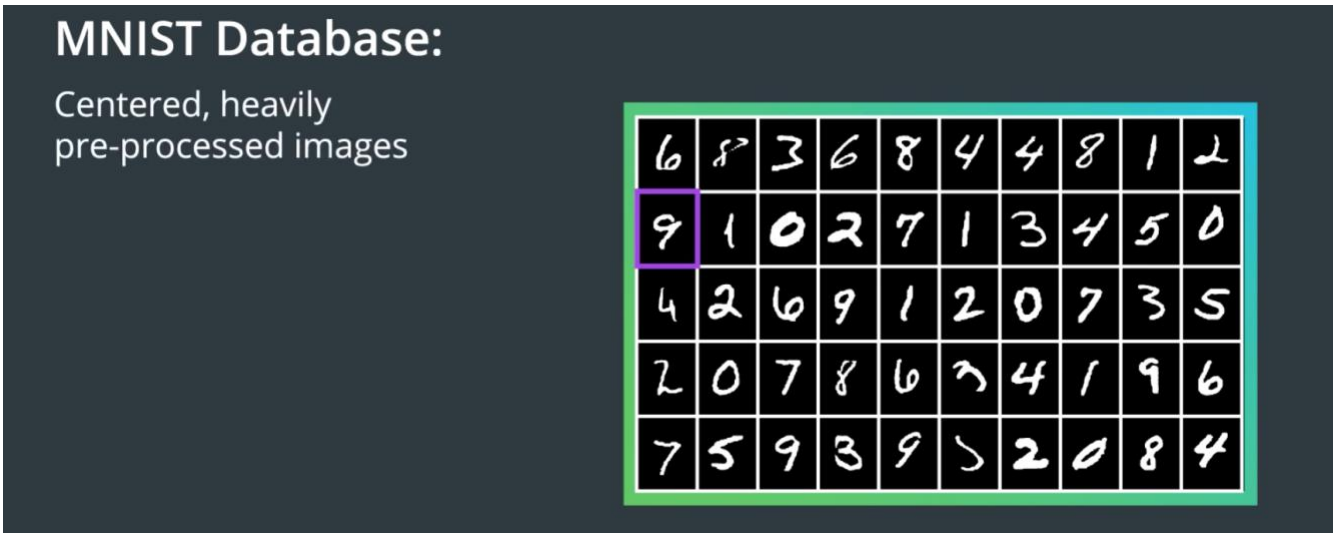
30, but definitely by epic 37. So, next step is to see how this model performs on our test data. Here, I'm actually loading the model that we saved earlier by name into our instantiated model, and I'm testing it as usual. Passing our test data into our model and recording the class accuracies. You can see we get 97 percent overall accuracy. This is really about the same as our last model, the one without validation. So, even though we train that model for 15 more epics, the results are about the same. This makes sense because this validation loss really isn't changing much. So, whether we save the model after epic 37 or 50, the model should be pretty similar. This behavior is also occurring because most of these images are very similar. The images are very processed and all of the digits look the same. So, in our non validation case, it didn't matter so much that our model trained for much longer, but in some cases, you will get overfitting and choosing a model based on validation loss will be even more important. So, model validation can be a really helpful way to select the best model and decide when to stop training.

notebook-mnist_mlp_exercise-second-time-end

Not_read_end

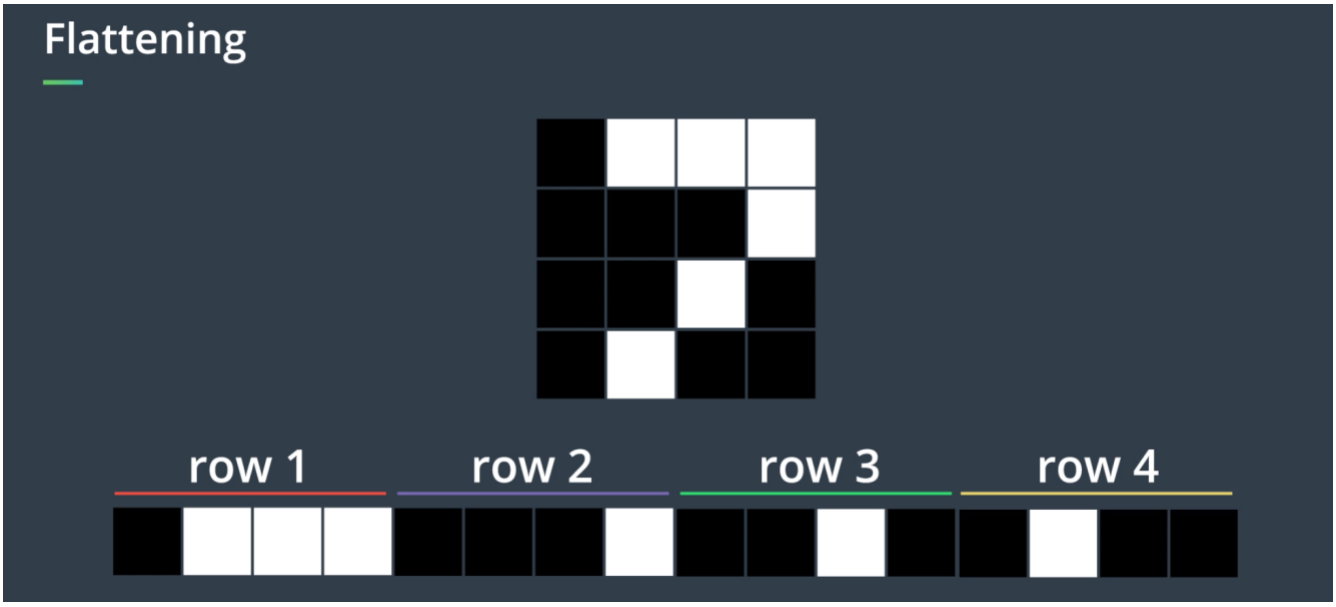


106. So far, you've learned a lot about approaching the task of image classification. In fact, now is a good time to review the pipeline we've developed for such a task. First, load and visualize the data you're working with, then pre-process that data by normalizing it and converting it into a tensor, so that it's prepped for further processing by the layers of a neural network. Then, do your research, has anyone approached this or similar task before? What kind of deep learning models have already been tried and tested? Use this research to help you decide on a model architecture and define it. Then, decide on loss and optimization functions and proceed with training your model. Here, you may choose to use a validation dataset to select and save the best model during training. Finally, test your model on previously unseen data. At this point, feel free to take a break from what you are learning, take time to absorb what you've learned so far. Then when you're ready, you can keep going. Next, I'll go over a new kind of architecture for image classification, a convolutional neural network, which can help us build even better image classification models.



107. So far, we've investigated how to train an MLP to classify handwritten digits in the MNIST dataset, and we got a really high test accuracy. MLPs are a nice solution in this case. Some other solutions for the MNIST classification task and their errors on the test dataset can be explored at the link provided below. You'll find that the best algorithms or the ones with the least test error are the approaches that use convolutional neural networks or CNNs. In fact, for most image classification tasks, CNNs and MLPs do not even compare, CNNs do much better. The MNIST database is an exception, and that it's very clean and pre-processed. All images of digits are roughly the same size and are centered in a 28 by 28 pixel grid.

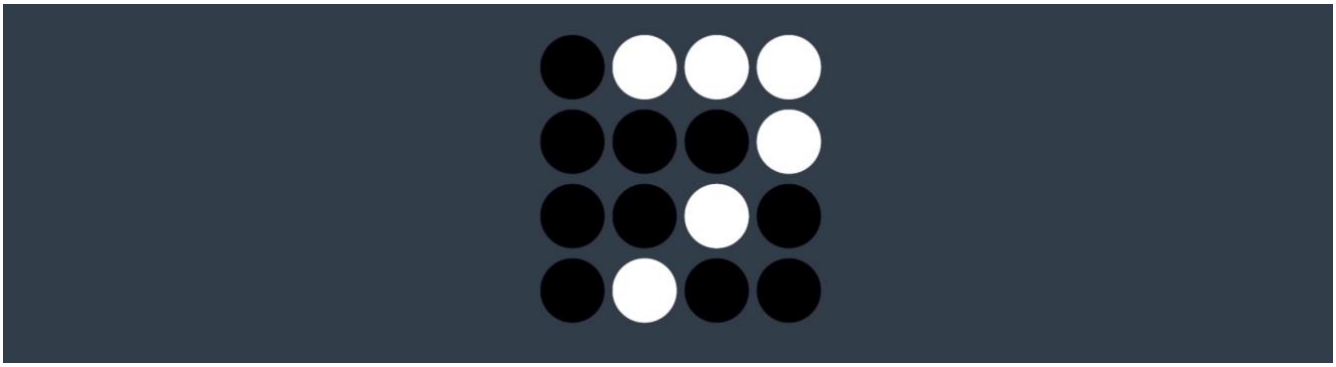
You can imagine that if instead of having to classify the digit within these very clean images, you had to work with images where the digit could appear anywhere within the grid, and sometimes appear quite small or quite large. It would be a more challenging task for an MLP. In the case of real-world messy image data, CNNs will truly shine over MLPs.



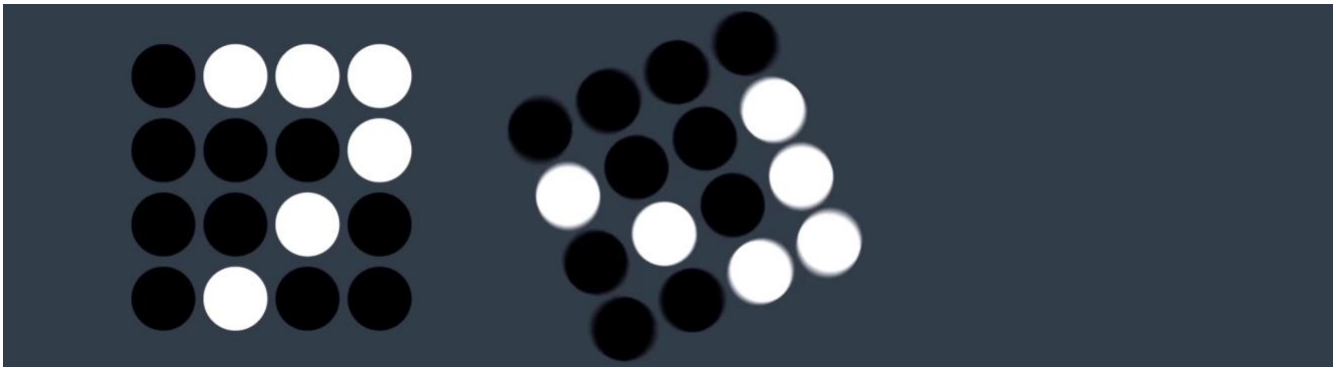
For some intuition for why this might be the case, we saw that in order to feed an image to an MLP, you must first convert the image to a vector. The MLP then treats this converted image as a simple vector of numbers with no special structure. It has no knowledge of the fact that these numbers were originally spatially arranged in a grid. CNNs, in contrast, are built for the exact purpose of working with or elucidating(闡明,說明) the patterns in multidimensional data. Unlike MLPs, CNNs understand the fact that image pixels that are closer in proximity to each other are more heavily related than pixels that are far apart. In this section, Alexis and I will discuss the math behind this kind of understanding. We'll present different types of layers that make up a CNN, and provide some intuition about each of their roles in processing an input and forming a complete neural network.

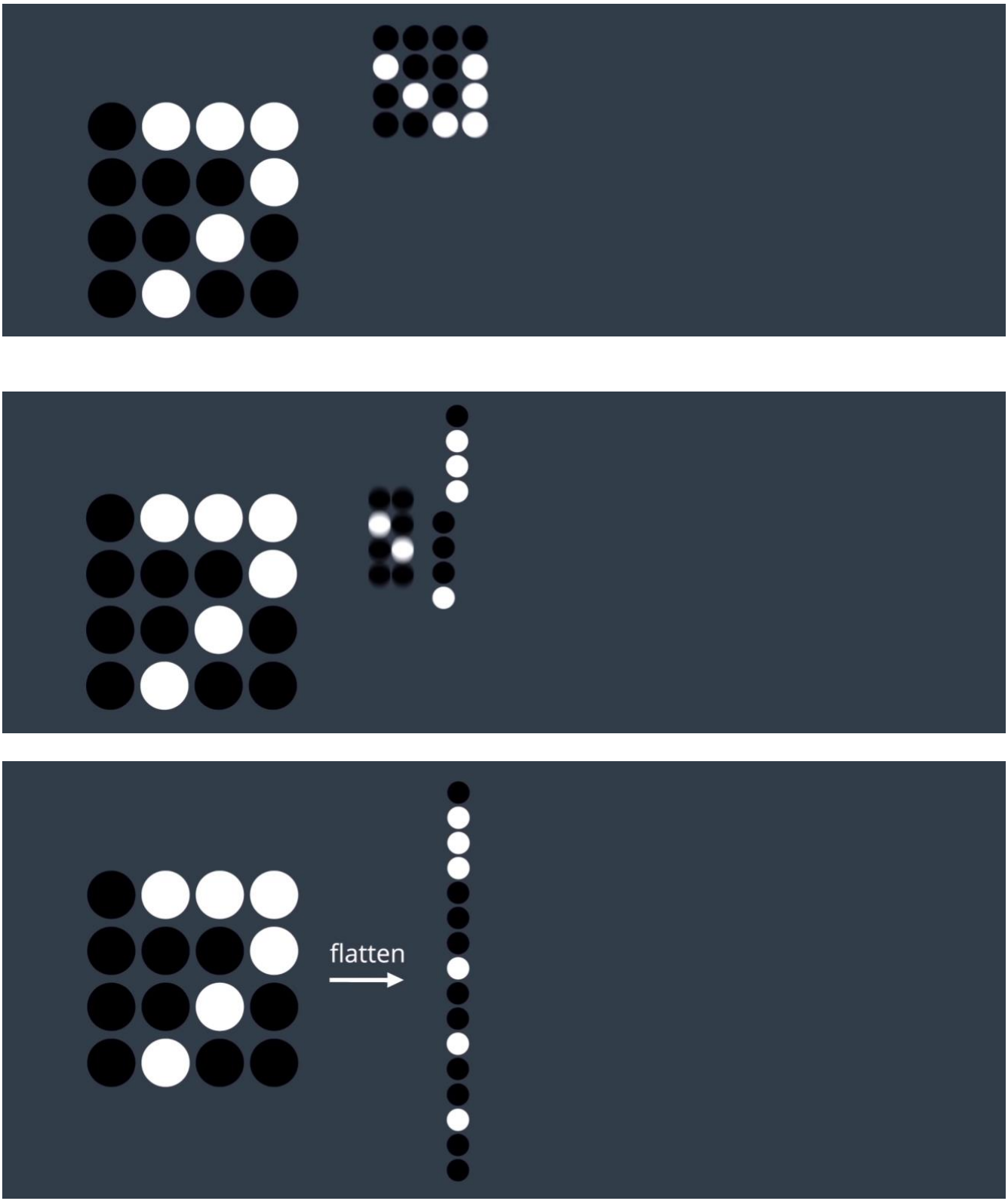
108. In the previous video, we used MLPs to decode images of handwritten numerical digits. Before feeding a gray scale image to an MLP, we had to first convert its matrix to a vector. This vector was then fed as input to an MLP with two hidden layers. We saw that it was a perfectly valid model for classifying images in the MNIST dataset. In fact, we got less than 2% error on our held out test images. But for other image classification problems, where we might need to analyze more sophisticated real world images with more complicated patterns, we'll need a different technique. In this video, towards motivating and defining CNNs, we specify a few improvements that

eliminate some of the drawbacks and limitations that we encounter when performing image classification with MLPs. Specifically, we'll adjust two important issues. First, we've seen that MLPs use a lot of parameters. The MLP from the previous video, for our very small 28 by 28 images, already contained over half a million parameters. You can imagine that the computational complexity for even moderately sized images could get out of control pretty fast. Another issue is that we throw away all of the 2-D information contained in an image when we flatten its matrix to a vector. This spatial information or knowledge of where the pixels are located in reference to each other is relevant to understanding the image and could aid significantly towards elucidating the patterns contained in the pixel values. This suggests that we need an entirely new way of processing image input, where the 2-D information is not entirely lost. CNNs will address these problems by using layers that are more sparsely connected. Where the connections between layers are informed by the 2-D structure of the image matrix. Furthermore, CNNs will accept our matrix as input.

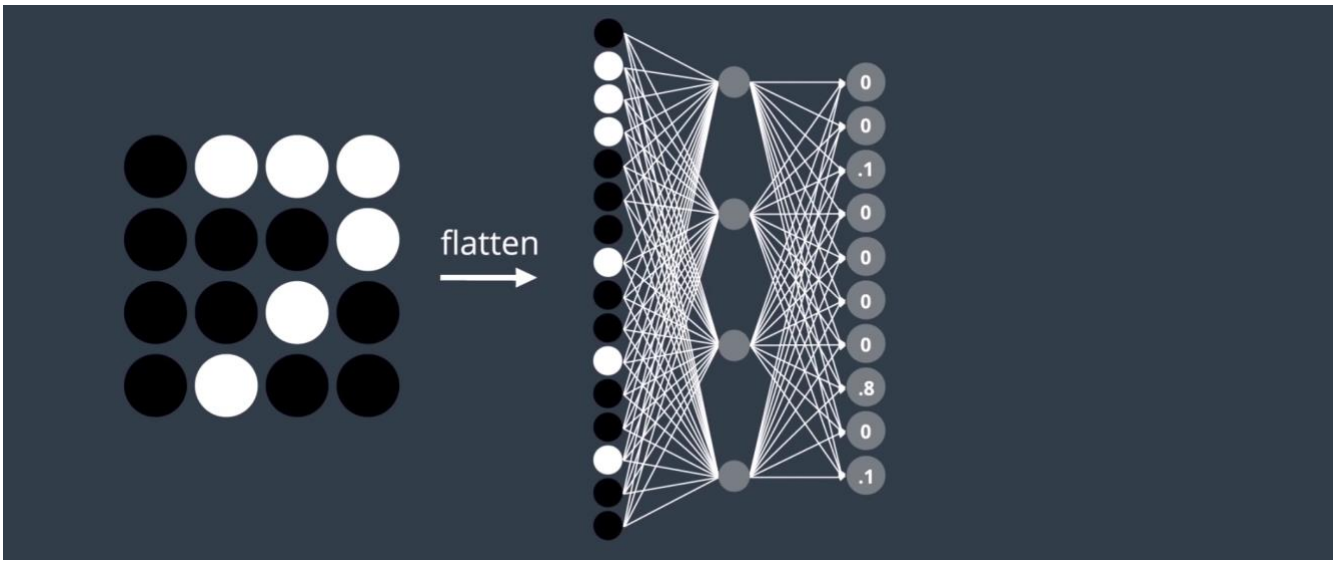


For illustration, consider a toy example with 4*4 images of handwritten digits. Our goal remains the same, we'd like to classify the digit that's depicted in the image.

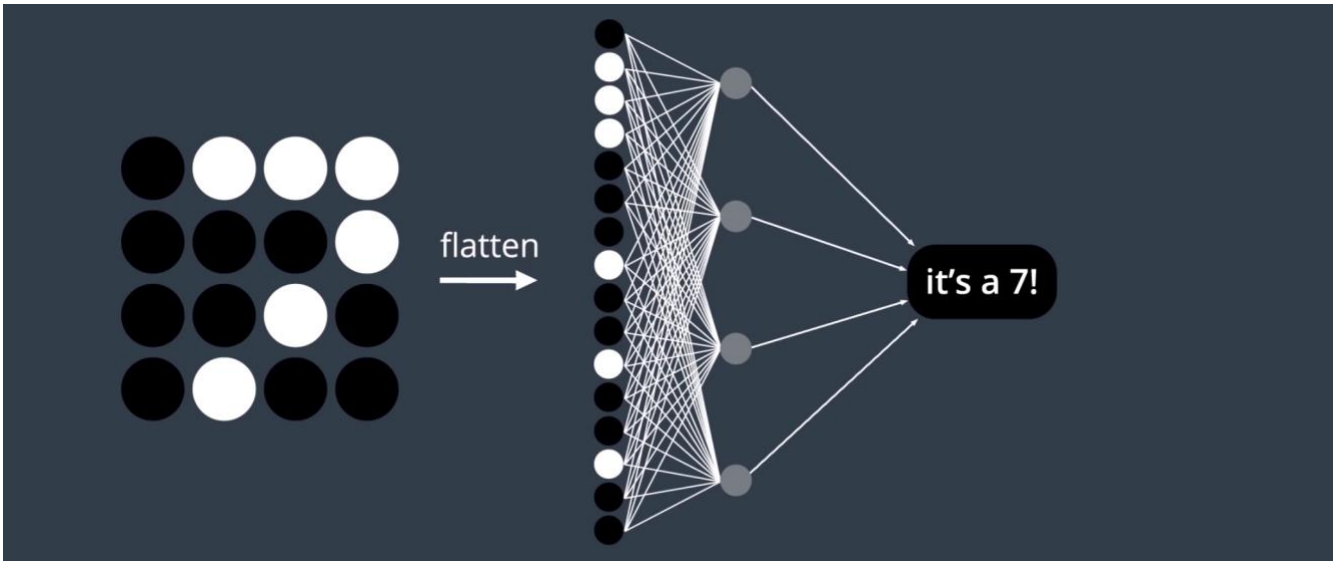




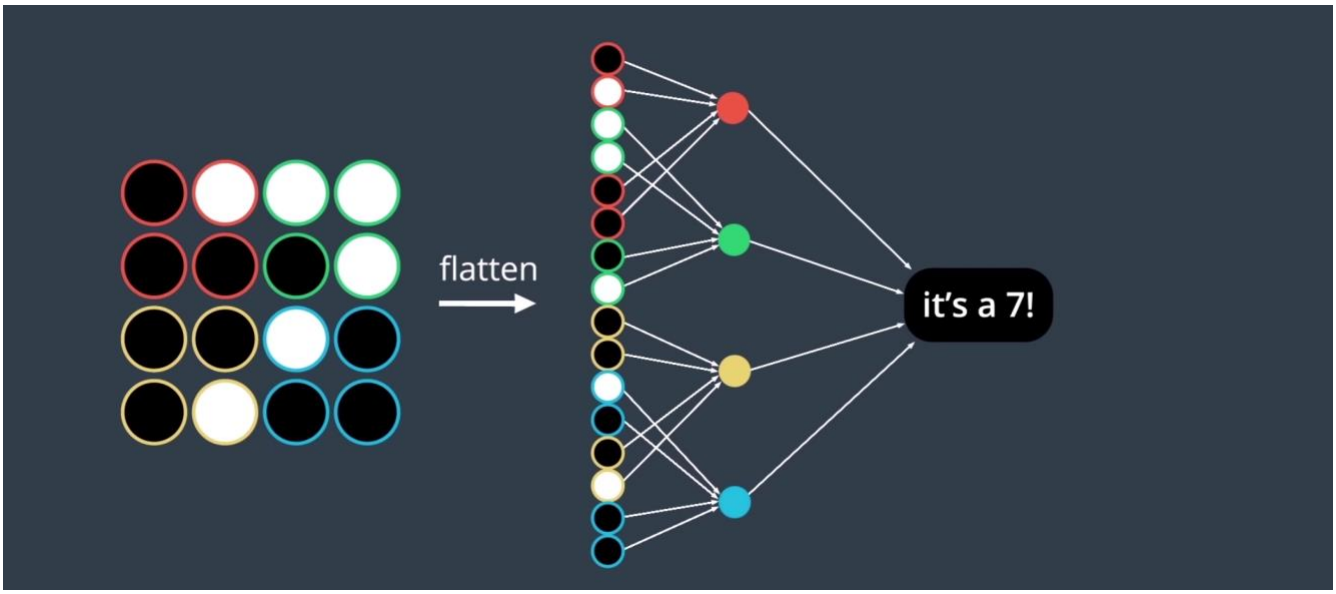
After converting the four by four matrix to a 16 dimensional vector, we can supply the vector as input to an MLP.



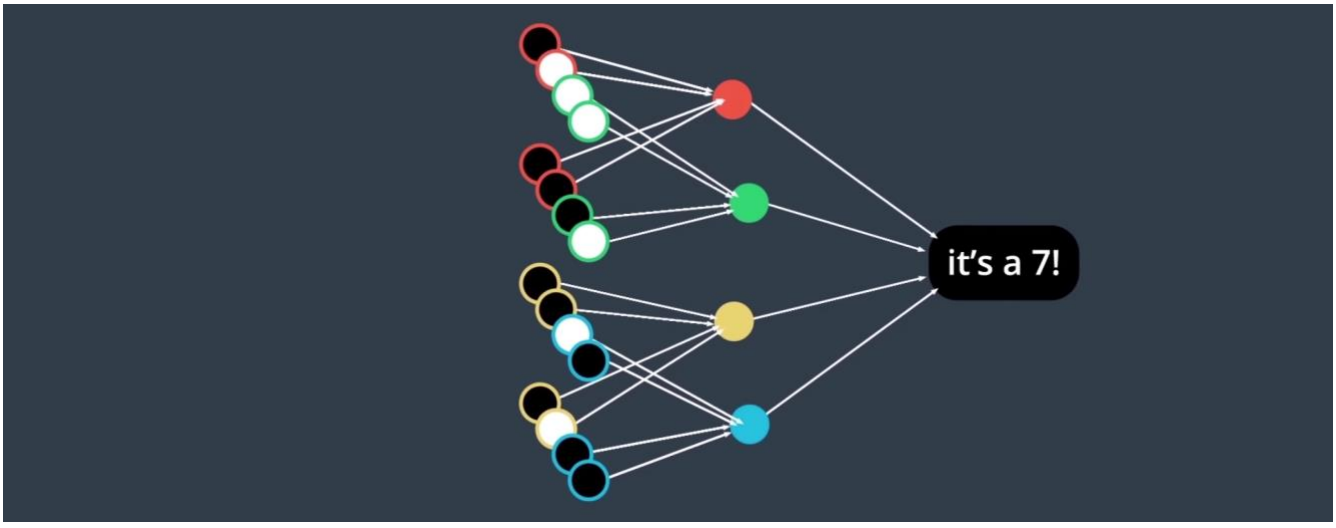
We construct an MLP with a single hidden layer with four nodes. The output layer has 10 nodes. As in the MLP in the previous video, the output has a softmax activation function and returns a 10-dimensional vector containing the probability that the image depicts each of the possible digits zero through nine. If we've trained our model well, the vector will predict that a seven is depicted in the image with high probability. But let's clean up this figure by replacing this detailed depiction with the suggested output layer.

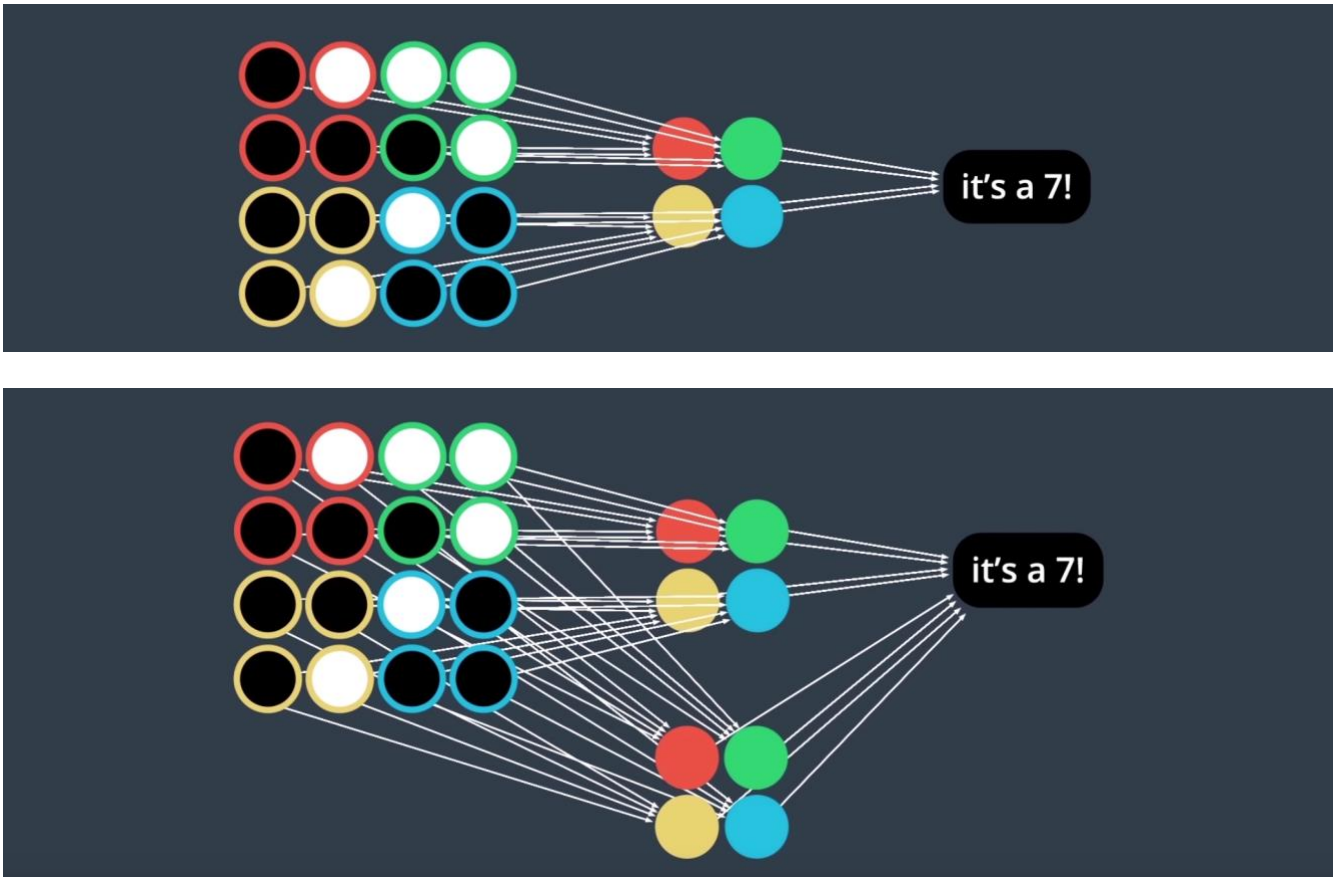


Here, the box annotated with “it's a seven”, is just shorthand notation for an output layer which predicts a seven. Just looking at this MLP, it becomes apparent that there may be some redundancy. Does every hidden node need to be connected to every pixel in the original image?



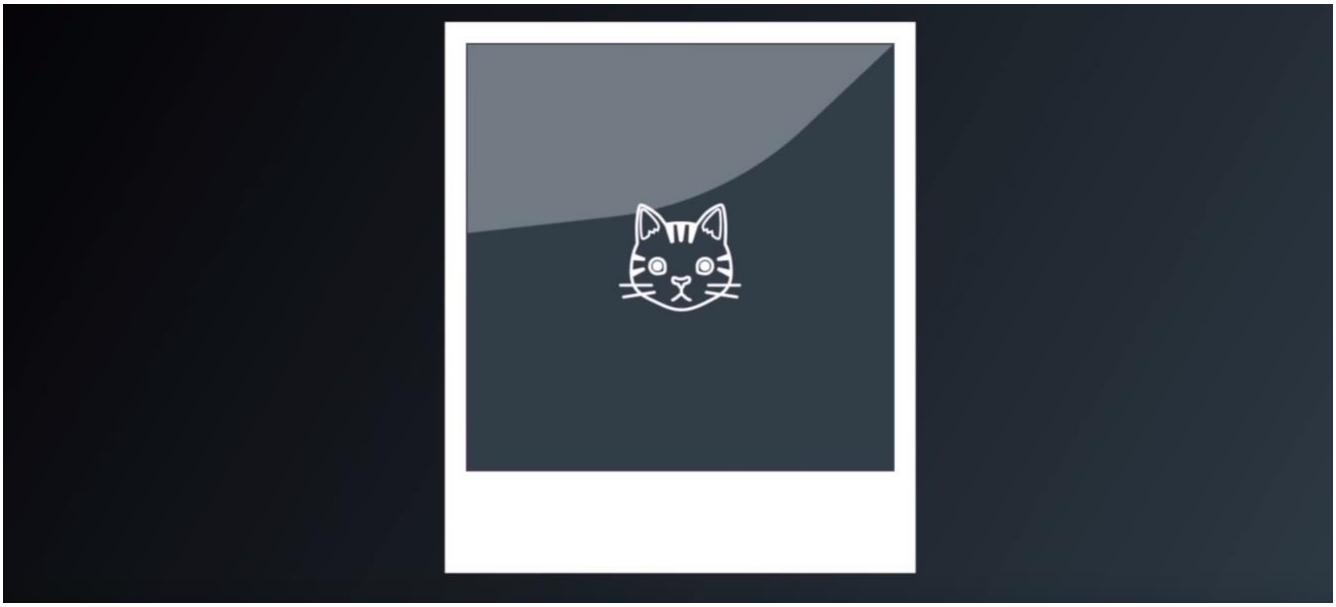
Perhaps not, consider breaking the image into four regions. Here, color coded as red, green, yellow, and blue. Then, each hidden node could be connected to only the pixels in one of these four regions. Here, each hidden node sees only a quarter of the original image. In the case of the previous fully connected layer, every hidden node was responsible for gaining an understanding of the entire image all at once. With this new regional breakdown and the assignment of small local groups of pixels to different hidden nodes, every hidden node finds patterns in only one of the four regions in the image. Then, each hidden node still reports to the output layer where the output layer combines the findings for the discovered patterns learned separately in each region. This so called locally connected layer uses far fewer parameters than a densely connected layer.



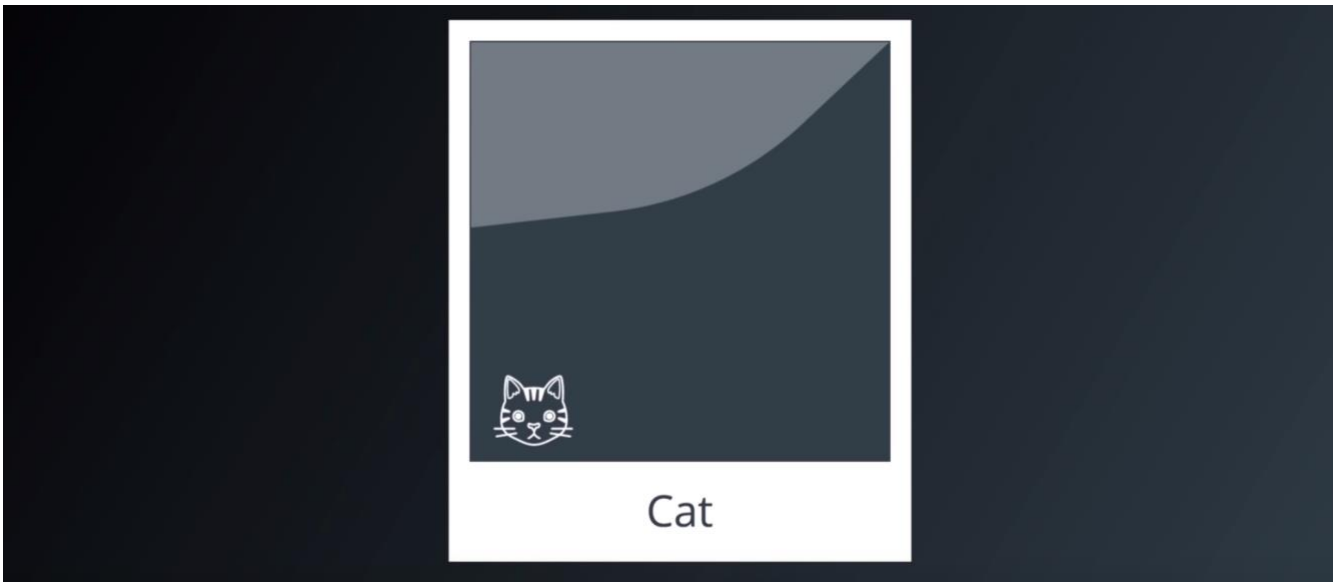


It's less prone to overfitting and truly understands how to tease out the patterns contained in image data. We can rearrange each of these vectors as a matrix. Where now the relationships between the nodes in each layer are more obvious. We could expand the number of patterns that we're able to detect while still making use of the 2-D structure to selectively and conservatively add weights to the model by introducing more hidden nodes. Where each is still confined to analyzing a single small region within the image. The red nodes in the hidden layer are still only connected to the red nodes in the input layer. With the same color coding for all other colors. After all we saw in the previous videos on neural networks that by expanding the number of nodes in the hidden layer we can discover more complex patterns in our data. We now have two collections of hidden nodes where each collection contains nodes responsible for examining a different region of the image.

It will prove useful to have each of the hidden nodes within a collection share a common group of weights. The idea being that different regions within the image can share the same kind of information. In other words, every pattern that's relevant towards understanding the image could appear anywhere within the image. Perhaps the simplest way to see how this parameter sharing will help our neural network classify objects is through a higher resolution image example.

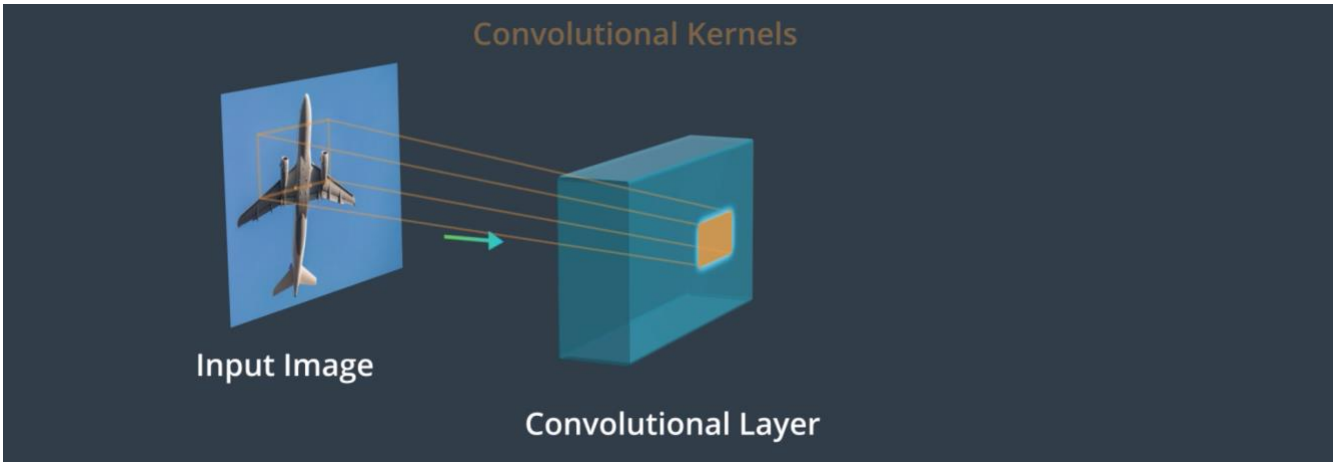


Say you have an image, and you want your network to say it's an image containing a cat.

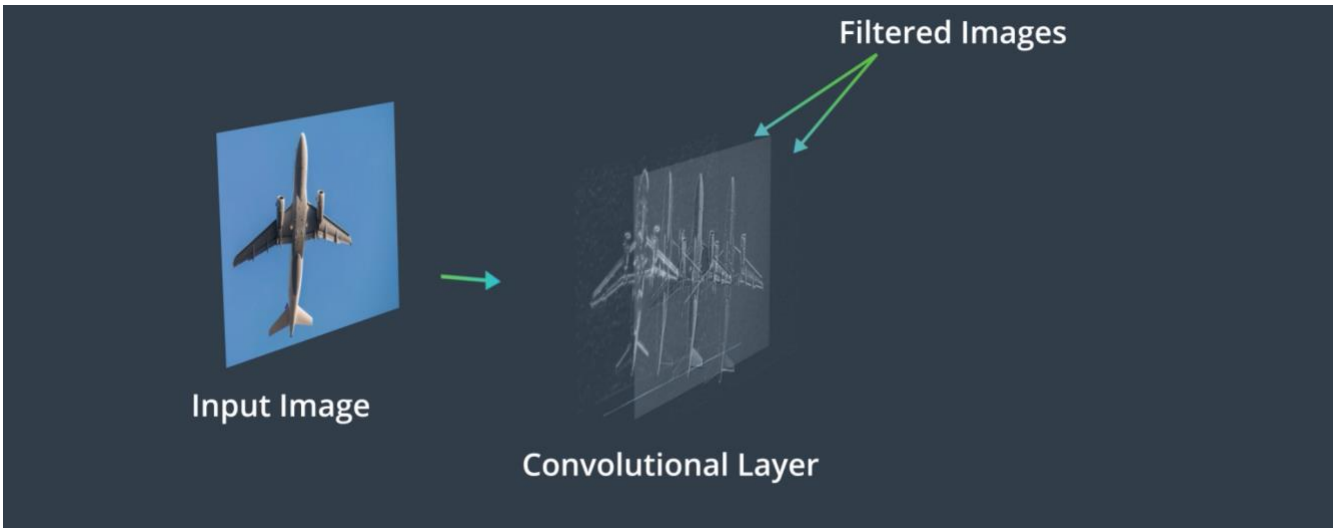


It doesn't matter where the cat is. It's still an image with the cat.

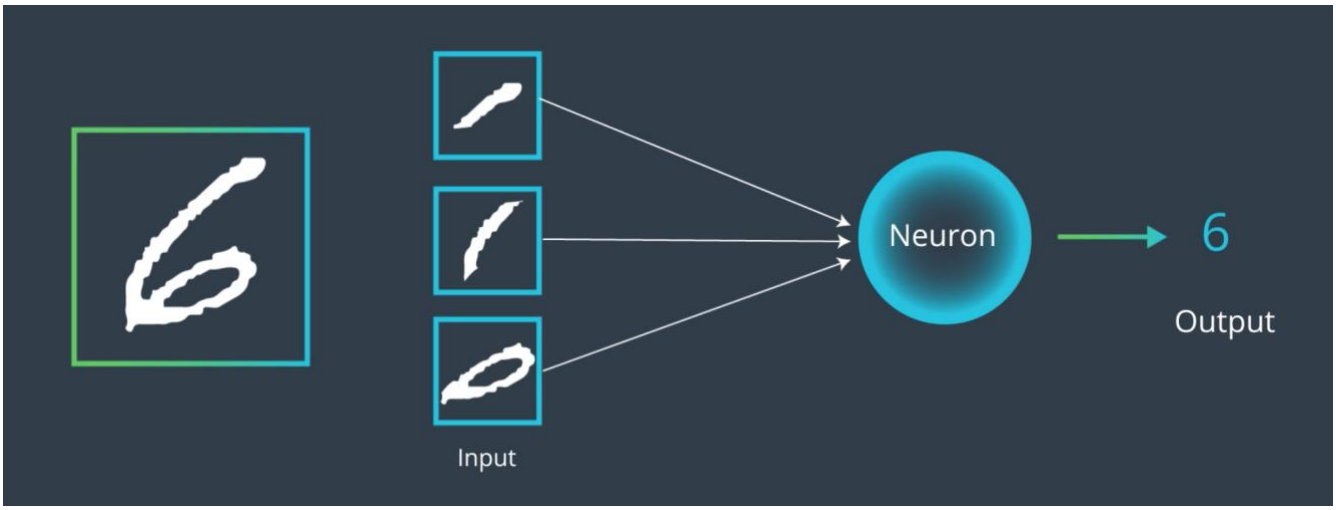
If your network has to learn about kittens in the left corner and kittens in the right corner independently. That's a lot of work that it has to do. Instead, we tell the network explicitly that objects and images are largely the same whether they're on the left or the right of the picture. And this is partially accomplished through weight sharing. All of these ideas will motivate so-called convolutional layers, which we present in the next video.



109. A convolutional neural network is a special kind of neural network in that it can remember spatial information. The neural networks that you've seen so far only look at individual inputs. But convolutional neural networks, can look at an image as a whole, or in patches and analyze groups of pixels at a time. The key to preserving the spatial information is something called the convolutional layer. A convolutional layer applies a series of different image filters also known as convolutional kernels to an input image.



The resulting filtered images have different appearances. The filters may have extracted features like the edges of objects in that image, or the colors that distinguish the different classes of images.



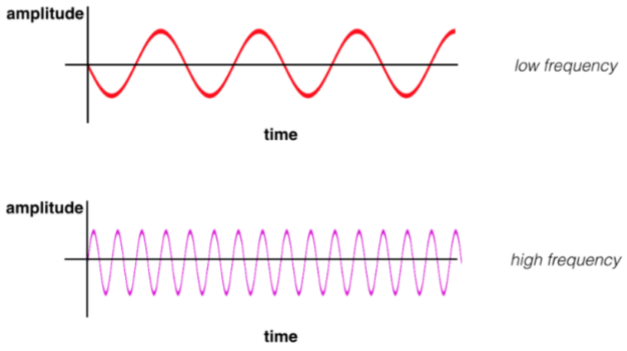
In the case of classifying digits for example, CNN should learn to identify spatial patterns like the curves and lines that make up the number six as distinct from another digit. Later layers in a model will learn how to combine different color and spatial features to produce an output like a class label. Before we get into defining and training a complete CNN in the next couple videos, you'll learn a bit more about the filters that define a convolutional layer. How they're defined, and how they can transform an input image? This will be important foundational knowledge that will come up again as you get to training as CNN and even visualizing the inner workings of a trained CNN.

(No picture)
110. When we talk about spatial patterns in an image, we're often talking about one of two things, color or shape, and I want to focus on shape for a moment. Shape can also be thought of as patterns of intensity in an image. Intensity is a measure of light and dark, similar to brightness, and we can often use this knowledge to detect the shape of objects in an image. For example, say, you're trying to distinguish a person from a background in an image. You can look at the contrast that occurs where the person ends and the background begins to define a shape boundary that separates the two. You can often identify the edges of an object by looking at abrupt changes in intensity, which happen when an image changes from a very dark to light area. In the next few videos, you'll be using and creating specific image filters that look at groups of pixels and detect big changes in intensity in an image. These filters produce an output that shows various edges and shapes. So, let's take a closer look at these filters and see what role they play in a convolutional neural network.

Frequency in images

We have an intuition of what frequency means when it comes to sound. High-frequency is a high pitched noise, like a bird chirp or violin. And low frequency sounds are low pitch, like a deep voice or a bass drum. For sound, frequency actually refers to how fast a sound wave is oscillating; oscillations are usually measured in cycles/s (Hz), and high pitches and made by high-frequency

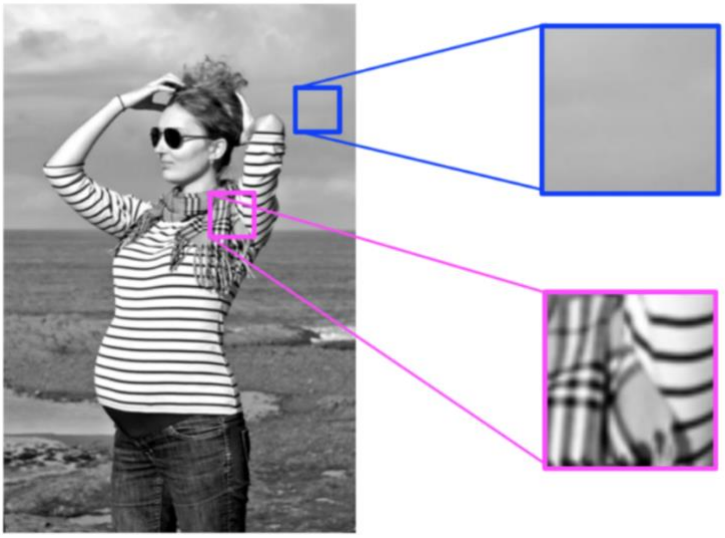
waves. Examples of low and high-frequency sound waves are pictured below. On the y-axis is amplitude, which is a measure of sound pressure that corresponds to the perceived loudness of a sound, and on the x-axis is time.



(Top image) a low frequency sound wave (bottom) a high frequency sound wave.

High and low frequency

Similarly, frequency in images is a rate of change. But, what does it means for an image to change? Well, images change in space, and a high frequency image is one where the intensity changes a lot. And the level of brightness changes quickly from one pixel to the next. A low frequency image may be one that is relatively uniform in brightness or changes very slowly. This is easiest to see in an example.

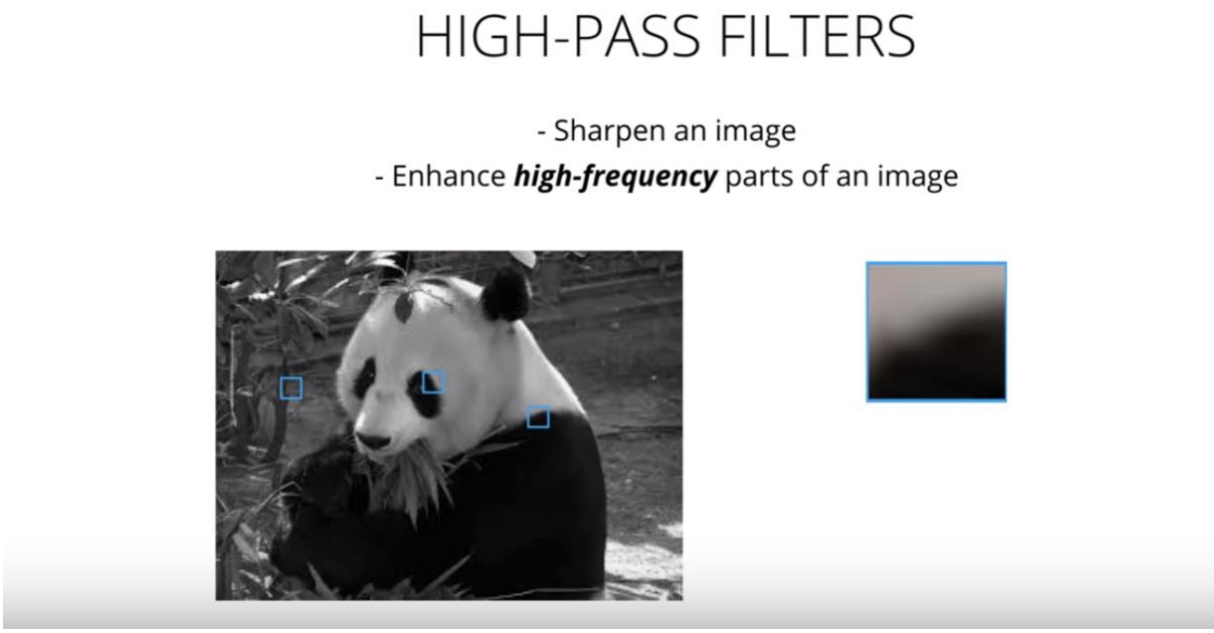


High and low frequency image patterns.

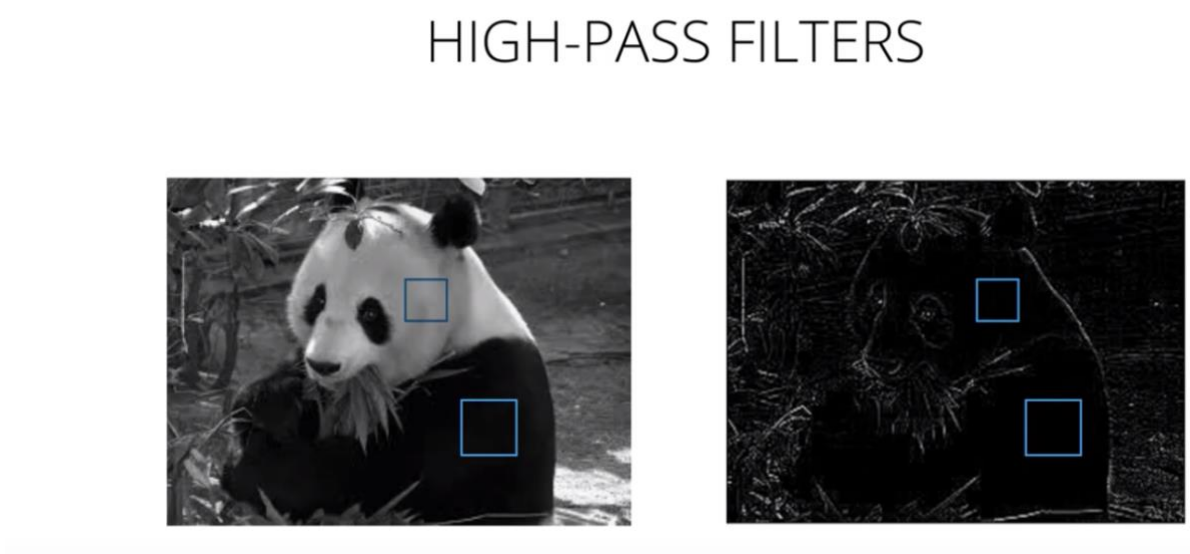
Most images have both high-frequency and low-frequency components. In the image above, on the scarf and striped shirt, we have a high-frequency image pattern; this part changes very rapidly from one brightness to another. Higher up in this same image, we see parts of the sky and background that change very gradually, which is considered a smooth, low-frequency pattern.

High-frequency components also correspond to the edges of objects in images, which can help us classify those objects.

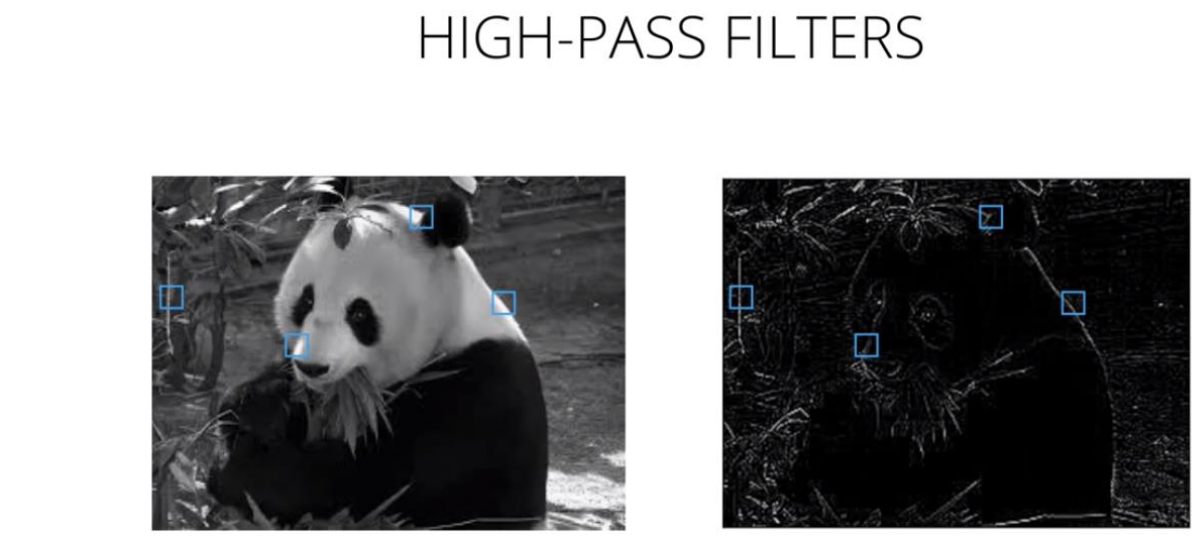
111. In image processing, filters are used to filter out unwanted or irrelevant information in an image or to amplify features like object boundaries or other distinguishing traits.



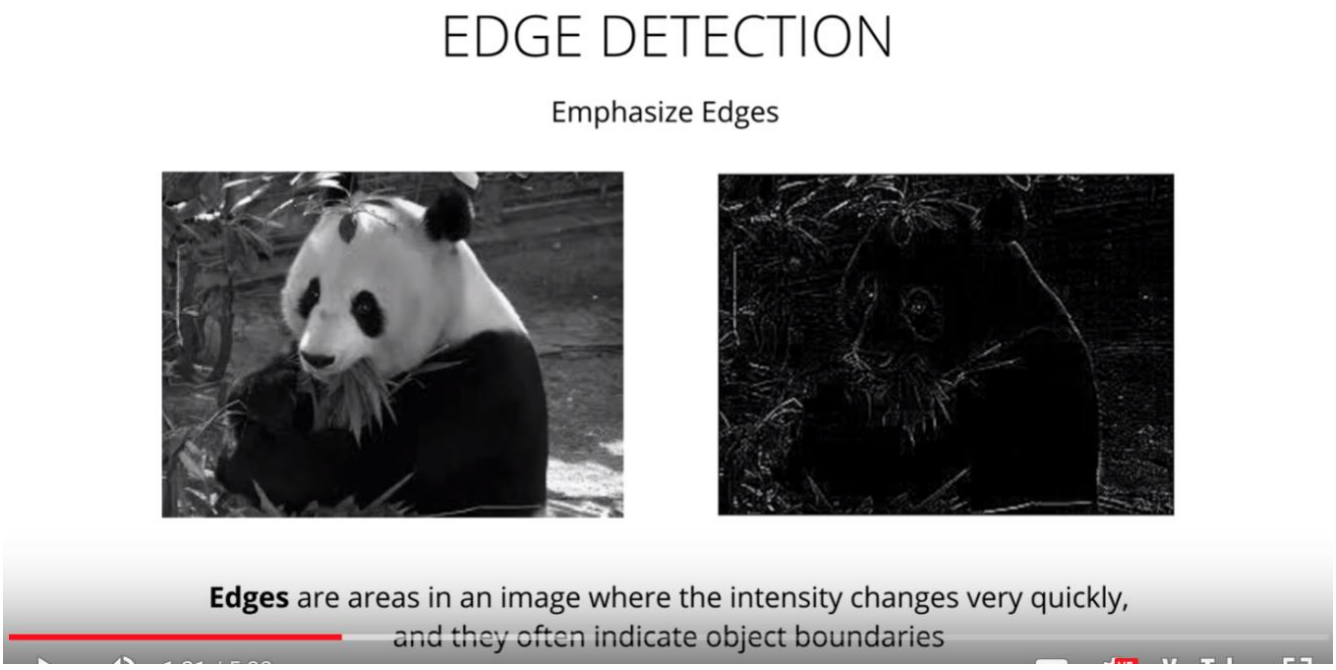
High-pass filters are used to make an image appear sharper and enhance high-frequency parts of an image, which are areas where the levels of intensity in neighboring pixels rapidly change like from very dark to very light pixels. Since we're looking at patterns of intensity, the filters we'll be working with will be operating on grayscale images that represent this information and display patterns of lightness and darkness in a simple format.



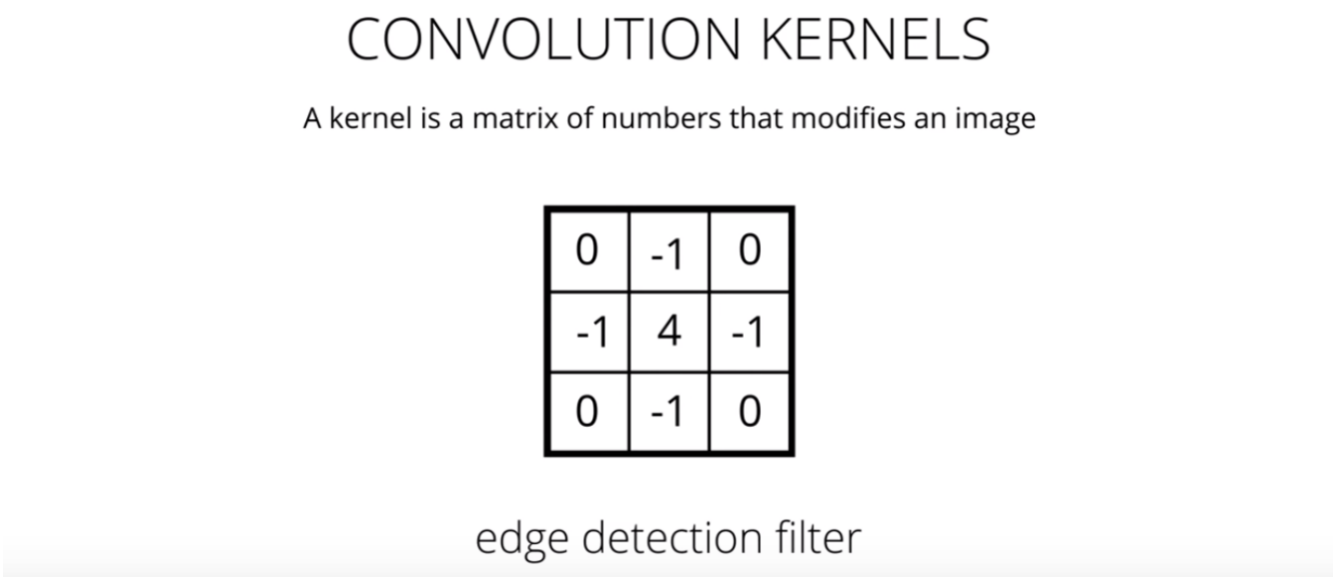
Let's take a closer look at this panda image as an example. What do you think will happen if we apply a high-pass filter? Well, where there is no change or a little change in intensity in the original picture, such as in these large areas of dark and light, a high-pass filter will black these areas out and turn the pixels black,



but in these areas where a pixel is way brighter than its immediate neighbors, the high-pass filter will enhance that change and create a line.



You can see that this has the effect of emphasizing edges. Edges or just areas in an image where the intensity changes very quickly and these edges often indicate object boundaries.



Now, let's see how exactly a filter like this works. The filters I'll be talking about are in the form of matrices often called convolution kernels, which are just grids of numbers that modify an image.

Here's an example of a high-pass filter that does edge detection. It's a three by three kernel whose elements all sum to zero.

CONVOLUTION KERNELS

0	-1	0
-1	4	-1
0	-1	0

$$0 + -1 + 0 + -1 + 4 + -1 + 0 + -1 + 0 = \mathbf{0}$$

It's important that for edge detection all of the elements sum to zero because this filter is computing the difference or change between neighboring pixels. Differences are calculated by subtracting pixel values from one another.

CONVOLUTION KERNELS

0	-1	0
-1	**8**	-1
0	-1	0

$$0 + -1 + 0 + -1 + \mathbf{8} + -1 + 0 + -1 + 0 = \mathbf{4}$$

CONVOLUTION KERNELS

0	-1	0
-1	4	**-2**
0	-1	0

0 + -1 + 0 + -1 + 4 + **-2** + 0 + -1 + 0 = **-1**

In this case, subtracting the value of the pixels that surround a center pixel, and if these kernel values did not add up to zero, that would mean that this calculated difference will be either positively or negatively weighted, which will have the effect of brightening or darkening the entire filtered image respectively.

CONVOLUTION

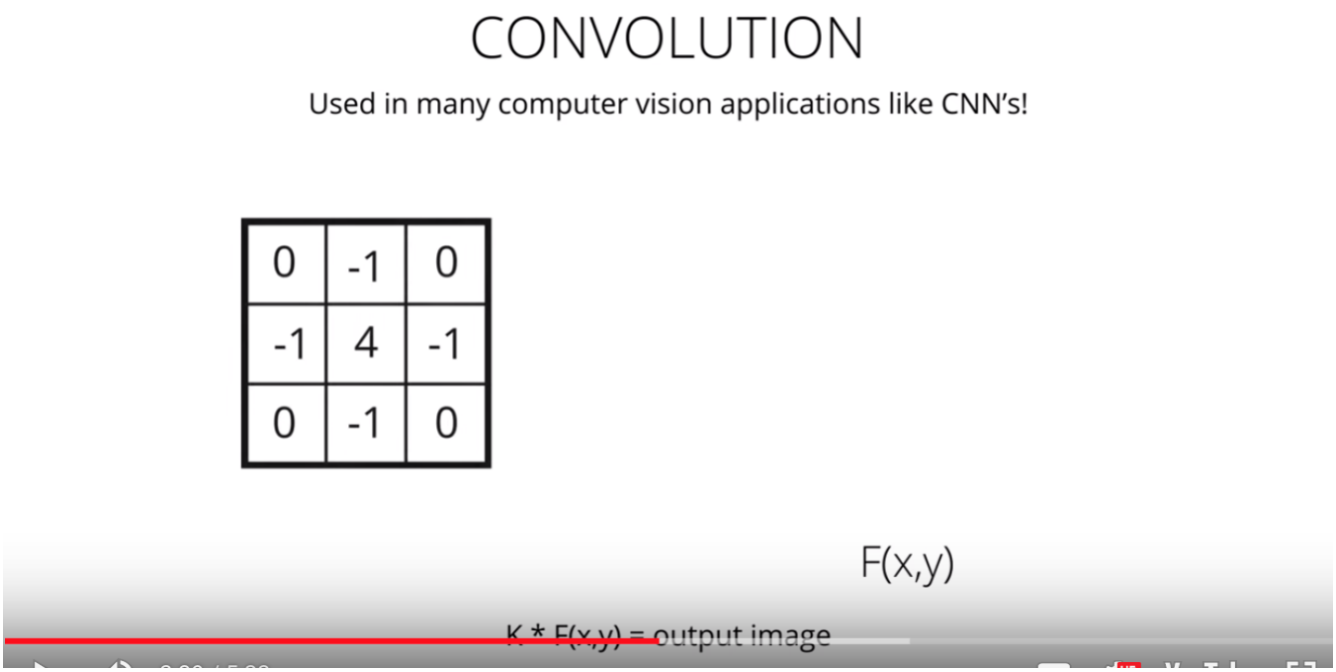
0	-1	0
-1	4	-1
0	-1	0

*

K F(x,y)

K * F(x,y) = output image

To apply this filter, an input image F(xy) is convolved with this kernel, which I'll call k. This is called kernel convolution and convolution is represented by an asterisk, not to be mistaken for a multiplication.



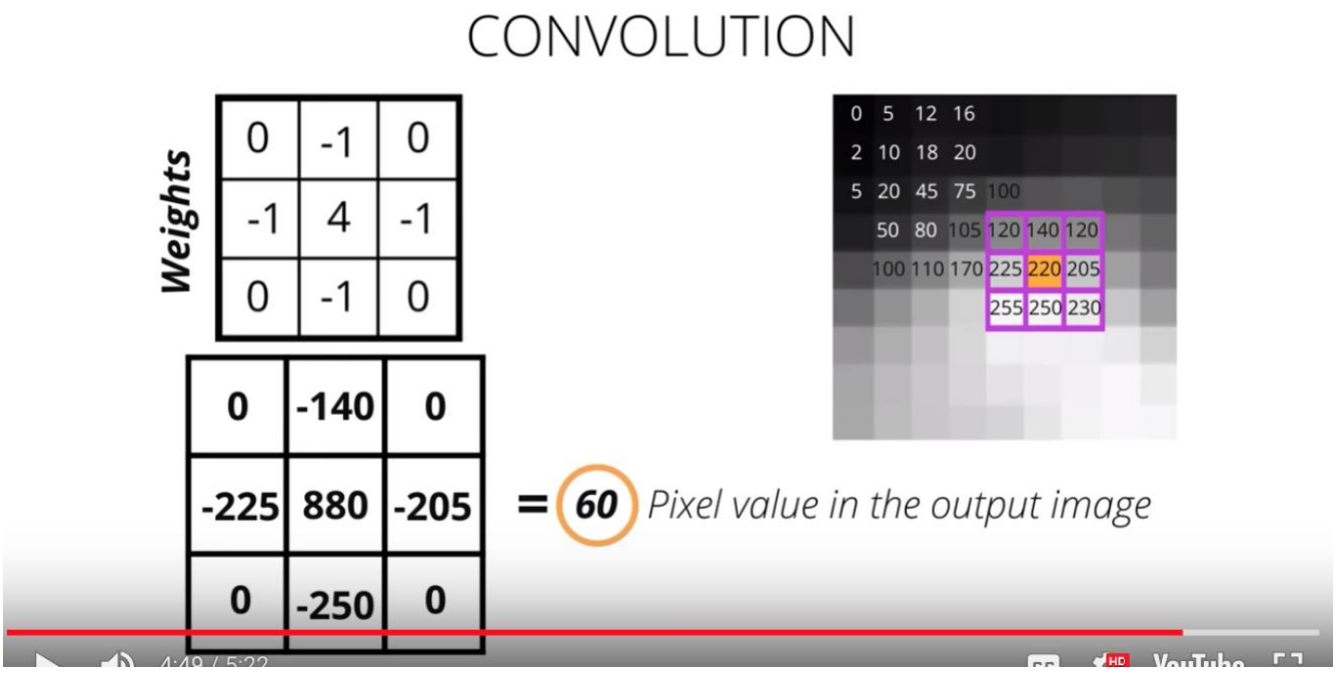
Kernel convolution is an important operation in computer vision applications and it's the basis for convolutional neural networks.

It involves taking a kernel, which is our small grid of numbers, and passing it over an image pixel by pixel transforming it based on what these numbers are and we'll see that by changing these numbers, we can create many different effects from edge detection to blurring an image.

CONVOLUTION

0	-1	0
-1	4	-1
0	-1	0

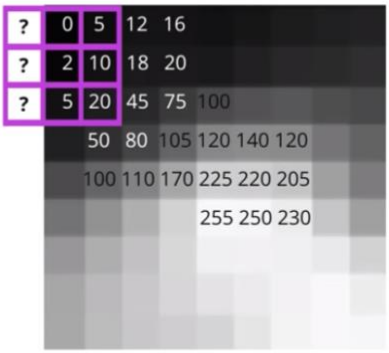
I'll walk through an example using this three by three edge detection filter.



To better see the pixel operations, I'll zoom in on this panda right by its ear to see the grayscale pixel values. First, for every pixel in this greyscale image, we put our kernel over it so that the pixel is in the center of the kernel, and I'm just choosing this pixel as an example. Then we look at

the three by three grid of pixels centered around that one pixel. We then take the numbers in our kernel and multiply them with their corresponding pixel in pairs. So this pixel in the top left corner, 120, is multiplied by the kernel corner 0 and next to that, we multiply the value 140 by -1, and the next, another 120 by 0. We do that for all nine pixel kernel value pairs. Notice that the center pixel with a value of 220 will be multiplied by 4, the center kernel value. Finally, these values are all summed up to get a new pixel value, 60. This value means a very small edge has been detected, which we can see by looking at this three by three area in the image. It changes from light at the bottom to a little darker on top, but it changes very gradually. These multipliers in our kernel are often called weights because they determine how important or how weighty a pixel is in forming a new output image. In this case, for edge detection, the center pixel is the most important followed by its closest pixels on the top and bottom and to its left and right, which are negative weights that increase the contrast in the image. The corners are the farthest away from the center pixel and in this example, we don't give them any weight. So this weighted sum becomes the value for the corresponding pixel at the same location XY in the output image,

0	-1	0
-1	4	-1
0	-1	0



and you do this for every pixel position in the original image until you have a complete output image that's about the same size as the input image with new filtered pixel values. The only thing you need to consider, other than this weighted sum, is what to do at the edges and corners of your image since the kernel cannot be nicely laid over three by three pixel values everywhere (tao: did not answer). Next, let's get a little more practice with these kinds of high-pass filters then get into coding our own.

Tao: see question under the picture:

Kernel convolution

Now that you know the basics of high-pass filters, let's see if you can choose the *best* one for a given task.

a)

-1	-1	-1
-1	8	-1
-1	-1	-1

b)

-1	0	1
-2	0	2
-1	0	1

c)

0	-1	0
-2	6	-2
0	-1	0

d)

-1	-2	-1
0	0	0
1	2	1

Four different kernels

Of the four kernels pictured above, which would be best for finding and enhancing horizontal edges and lines in an image?

OpenCV

Before we jump into coding our own convolutional kernels/filters, I'll introduce you to a new library that will be useful to use when dealing with computer vision tasks, such as image classification: OpenCV.

OpenCV logo

OpenCV is a computer vision and machine learning software library that includes many common image analysis algorithms that will help us build custom, intelligent computer vision applications. To start with, this includes tools that help us process images and select areas of interest! The library is widely used in academic and industrial applications; from their site, OpenCV includes an impressive list of users: “Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many startups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV.”

So, note, how we import cv2 in the next notebook and use it to create and apply image filters!

Notebook: Custom Filters

The next notebook is called custom_filters.ipynb. <- Tao: no subtitle (text) for this notebook.

To open the notebook, you have two options:

Go to the next page in the classroom (recommended).
Clone the repo from Github and open the notebook custom_filters.ipynb in the convolutional-neural-networks > conv-visualization folder. You can either download the repository with git clone https://github.com/udacity/deep-learning-v2-pytorch.git, or download it as an archive file from this link.

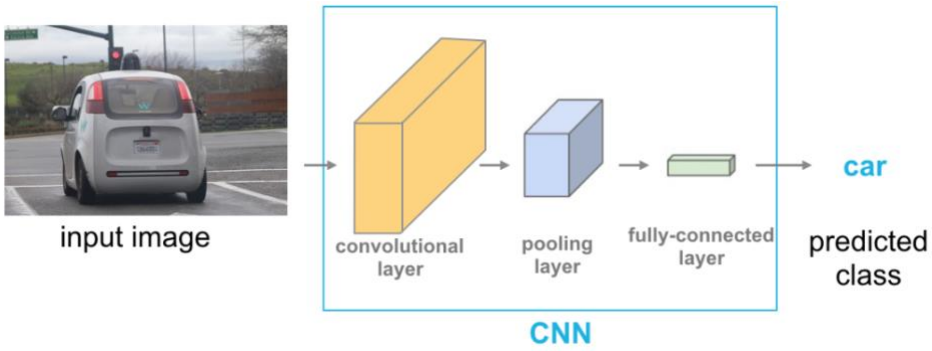
Instructions

- Define your own convolutional filters and apply them to an image of a road
- See if you can define filters that detect horizontal or vertical edges

This notebook is meant to be a playground where you can try out different filter sizes and weights and see the resulting, filtered output image!

The Importance of Filters

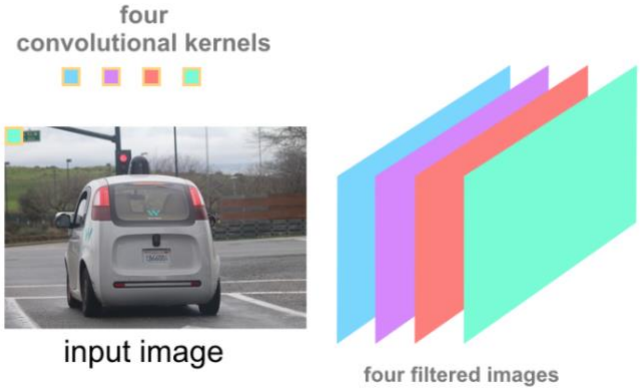
What you've just learned about different types of filters will be really important as you progress through this course, especially when you get to Convolutional Neural Networks (CNNs). CNNs are a kind of deep learning model that can learn to do things like image classification and object recognition. They keep track of spatial information and learn to extract features like the edges of objects in something called a convolutional layer. Below you'll see an simple CNN structure, made of multiple layers, below, including this "convolutional layer".



Layers in a CNN.

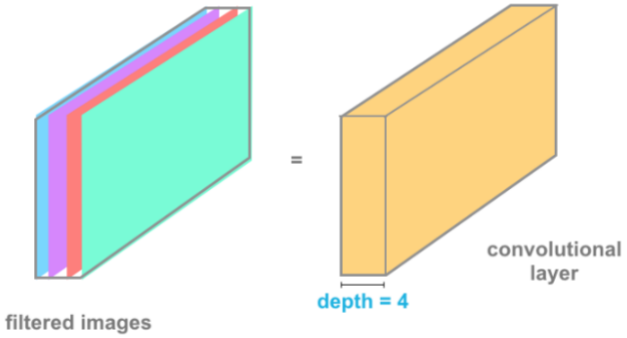
Convolutional Layer

The convolutional layer is produced by applying a series of many different image filters, also known as convolutional kernels, to an input image.



4 kernels = 4 filtered images.

In the example shown, 4 different filters produce 4 differently filtered output images. When we stack these images, we form a complete convolutional layer with a depth of 4.



A convolutional layer.

Learning

In the code you've been working with, you've been setting the values of filter weights explicitly, but neural networks will actually learn the best filter weights as they train on a set of image data. You'll learn all about this type of neural network later in this section, but know that high-pass and low-pass filters are what define the behavior of a network like this, and you know how to code those from scratch!

In practice, you'll also find that many neural networks learn to detect the edges of images because the edges of object contain valuable information about the shape of an object.