Hewlett Packard Enterprise

Using HPE Cray AI Development Environment

Learner Guide

Rev. 22.11

Hewlett Packard
Enterprise

Machine Learning and Deep Learning Fundamentals

Module 1

Objectives

Artificial intelligence (AI) has spread into applications across many different industries. You likely encounter AI every day in the form of chat bots as you browse the Internet, text prediction as you type out messages on your smartphone, spam filters as you check your email, and recommendations as you decide which movie to stream. The list goes on.

In this module, you will learn about machine learning and deep learning, the technologies that make AI attainable. By the time that you have completed this module, you should be able to:

- Have a conversation with customers about machine learning (ML) and deep learning (DL)
- Understand the challenges customers face in training DL models

Introduction to artificial intelligence (AI)

You will begin with a brief overview of AI, ML, and DL.

What Is Machine Learning (ML)?

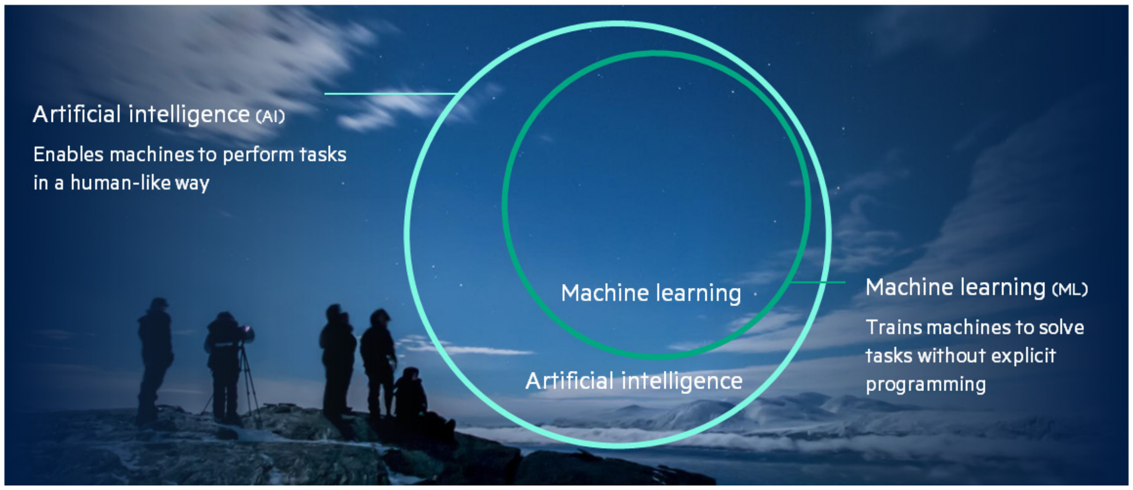

Artificial intelligence (AI)
Enables machines to perform tasks in a human-like way
Machine learning
Artificial intelligence
Machine learning (ML)
Trains machines to solve tasks without explicit programming

Figure 1-1: What Is Machine Learning (ML)?

Before diving deeper into ML, pause and review some terminology.

ML is an example of AI. AI encompasses any machine or application that is capable of imitating human behavior. For example, assistants such as Siri are programmed to respond to questions and perform tasks that you would expect from a human assistant, rather than a computer program. Unlike with a typical computer program, you do not have to execute precisely the same commands to get Siri to perform a specific task. Instead Siri “figures out” what you want and then does it. Of course, Siri and other examples of AI are not perfect. It is relatively easy to tell the difference between Siri and a real human. However, such applications are getting better and better at imitating human behavior.

AI is a description of how an application behaves. It is agnostic as to how the application is programmed to do so. ML is a *type* of AI, which represents one of the most successful *methods* for developing AI applications. Rather than a human programming the application precisely how to behave, an ML application is based on an algorithm that trains itself how to perform a task. For example, many route guidance applications use ML to train themselves how to find the best route from point A to point B.

Common ML tasks

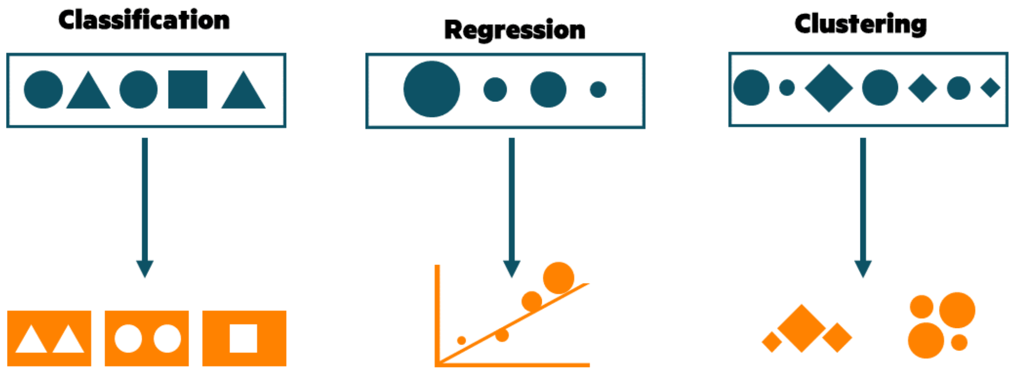


Figure 1-2: Common ML tasks

While companies can apply ML in nearly limitless ways, ML often involves making predictions. Given a set of inputs, what is the correct output? Tasks performed by ML often fall in several main categories:

- **Classification**—Classification seems simple; the application sorts data into pre-defined categories. But some of the most complex ML applications involve classification. Think about image recognition. The application recognizes whether a picture is of a cat, a dog, or a snake. Or perhaps an application finds the location of a face within an image. Computer vision, in which a machine processes camera input in real time to "see" like a human does, fundamentally relies on image classification.
- **Regression**—In mathematics, regression seeks to find the relationship between several independent variables (inputs) and another dependent variable (the output). Regression aids in predictions. For example, a regression could predict the price of a house based on factors such as area, number of bedrooms, number of bathrooms, and location. The price will fall along a scale. The graph of the relationship between the inputs and output might look like a line, for a linear regression, or take another shape, such as an s curve for a logistic regression.

  Regression can actually underlie many ML tasks, including classification. For example, a spam filter might use regression; based on a variety of factors, the spam filter assigns a particular email a "spam" score. If that score passes a threshold, the filter classifies the email as spam.
- **Clustering**—A clustering algorithm seeks to group similar data together. A recommender, such as one used by streaming sites, might use clustering to group together users with similar profiles and histories. The recommender can then recommend options to each user based on what other users in the same cluster have enjoyed.

ML applications might combine the features described above. Natural language processing (NLP) of audio input involves classification. Each word is a class, and the app assigns bursts of sound to the correct class (word). But then the NLP application might use clustering to determine what type of verbal request is being made.

What is an ML algorithm?

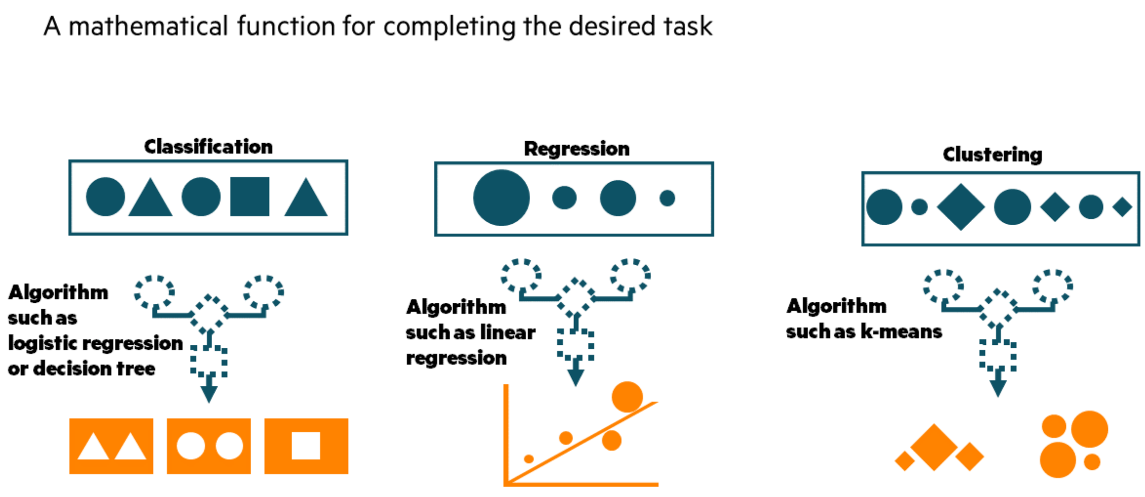

A mathematical function for completing the desired task
Classification
Algorithm such as logistic regression or decision tree
Regression
Algorithm such as linear regression
Clustering
Algorithm such as k-means

Figure 1-3: What is an ML algorithm?

An ML application uses a particular algorithm to complete the task. An algorithm is simply a mathematical function. Return to the example of housing price prediction, which is a regression task. The algorithm for completing this task could be a linear regression that looks a bit like this:

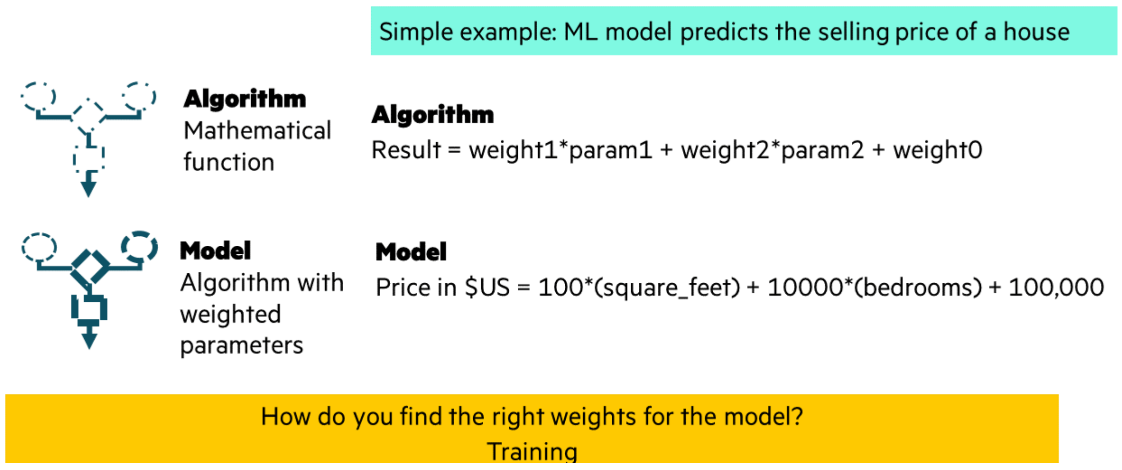
$y = w_1x_1 + w_2x_2 + w_3x_3 + w_0$

In that algorithm, where y is the price, each x refers to an input parameter (such as house area), and each w refers to how heavily to weigh that parameter in calculating the answer. For the purposes of this course, you do not need to understand the mathematics of the algorithms, but hopefully this simple example gives you an idea of what an algorithm can look like.

As mentioned before, classification might use regression algorithms as well. The algorithm receives inputs and produces an output, which is often a probability that data belongs to a particular class. Based on those probabilities, the application assigns the most likely class. Or an ML application might use a decision tree algorithm to sort data into different classes.

Other algorithms are best suited for clustering tasks. ML engineers often use a k-means algorithm, which plots data across a "space" and then partitions that space into groups. Data records within the same partition are part of the same cluster.

ML algorithms and models



Simple example: ML model predicts the selling price of a house

Algorithm
Mathematical function

Algorithm
Result = weight1*param1 + weight2*param2 + weight0

Model
Algorithm with weighted parameters

Model
Price in $US = 100*(square_feet) + 10000*(bedrooms) + 100,000

How do you find the right weights for the model?
Training

Figure 1-4: ML algorithms and models

In ML, “model” refers to an algorithm with specific weights for each parameter in the algorithm. Return to the example of the housing price predictor to clarify the relationship.

Suppose that you have selected a linear regression algorithm for this application. That algorithm takes this format:

$y = w_1x_1 + w_nx_n + w_0$

As you start to develop the model, you choose the parameters. In the real world, you would choose many parameters, but for this simple example, you select just two (area in square feet and number of bedrooms):

Price in \$US = w_1*(area)* + w_2*(bedrooms)*

You can also add a “bias,” here depicted as w_0, which pushes the price higher or lower:

Price in \$US = w_1*(area)* + w_2*(bedrooms)* + w_0

A specific model assigns particular weights to each parameter and the bias. For example, this is one model:

Price in $US = 100*(area in square feet) + 10,000*(bedrooms) + 100,000

And here is another model:

Price in $US = 125*(area in square feet) + 8,000*(bedrooms) + 120,000

As you can tell, even an artificially simplified algorithm like this one has myriads of potential models. How do you find out the correct weights for the model that produces the most accurate results? You must train the model. The training process teaches the application to adjust the weights until the model performs very accurately.

Simplified example of the ML lifecycle

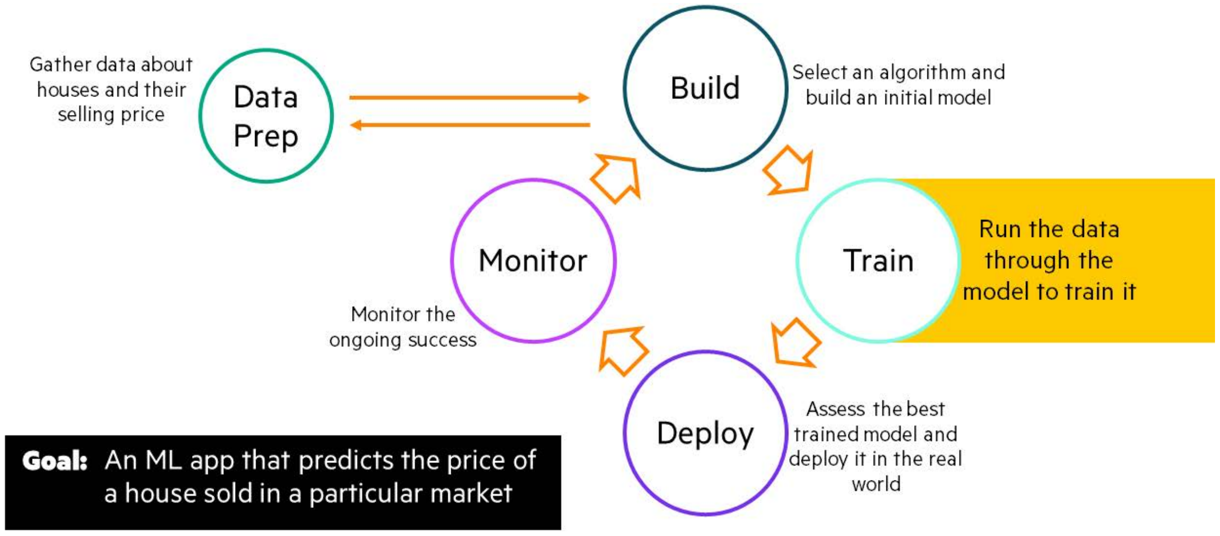

Gather data about houses and their selling price
Data Prep
Build
Select an algorithm and build an initial model
Train
Run the data through the model to train it
Monitor
Monitor the ongoing success
Deploy
Assess the best trained model and deploy it in the real world
Goal: An ML app that predicts the price of a house sold in a particular market

Figure 1-5: Simplified example of the ML lifecycle

The figure above shows a simplified view of the lifecycle of developing an ML application. Consider this lifecycle for our simple housing price predictor. The sections below describe the types of tasks ML engineers perform at each stage of the lifecycle. (Note that, depending on the application, the users developing the ML application might have many titles: data scientist, researcher, or others. This course will refer to these users generically as “ML engineers.”)

Data prep

ML engineers begin by gathering the data that they will use to train the model. For this example, they collect a broad range of selling prices for houses across the market in question.

Carefully selecting data plays a key role in creating a powerful and accurate ML application. You will hear the phrase “garbage in, garbage out” frequently here. In other words, if you train your ML app on “garbage”—too little data or data that does not accurately reflect the range of information that the app will confront in the real world—the app will fail to perform well. Those collecting the data need to take particular care to avoid building prejudice into the data set. For example, to train a face recognition app, you must include faces from people of many races, genders, ages, and body types.

At this stage, ML engineers might need to clean the data, fixing any errors or removing records with insufficient information. They also often need to place the data in a format that can be used by the training process. In this example, they might create a spreadsheet in which each row is a record: one house. For each record, the ML engineers input the values for the selected parameters, one parameter per column. The record also includes a column for a “label,” the actual selling price for that house.

Depending on the application, companies might want to update the data set with new data. Companies might implement extract, transform, load (ETL) processes for taking the data from the environment in which it is first stored and copying it to the environment in which the ML applications use it. ETL operates on batches of data; for example, a company might run ETL once a week or once a day. As an alternative to ETL, some companies are moving to streaming, in which they continuously stream data into the location used by ML applications. Streaming typically requires sophisticated data management pipelines.

Build

Next ML engineers select an appropriate algorithm for the application and start to build the initial model using a coding language such as Python and open source libraries such as NumPy or Scikit. ML engineers might also look in repositories such as Github for (untrained) ML models that have already been defined for a use case similar to theirs.

As they build the model, ML engineers might select initial weights for the parameters, or the training process might select those randomly. In either case, those initial weights probably do not produce great results. The model needs to be trained.

Train

Training feeds data to the model, allowing the model to produce results. Training further assesses how well the model is performing. The training process can then adjust the model weights so that the model can perform better. HPE Cray AI Development Environment, the software on which this course focuses, primarily addresses the training stage. Therefore, you will look at this stage in a lot more detail over the next pages.

Deploy

When the training is complete, the ML engineers have a model that is performing well. Perhaps they have even trained several models. They validate how well the models do with new data and decide which one performs best. For certain applications, companies might also run their trained models through “bias detection” tools. These tools attempt to detect and remove biases that have crept into the model based on issues such as biased data or parameter selection.

Once the ML engineers have identified the best model, they can deploy, or “serve,” it.

Serving involves exposing the model to clients, often through an application programming interface (API). Clients input new data, and the model outputs the results. This process is sometimes called inferencing. In our housing price example, you submit information about a house to the ML app, and the app returns a prediction of its eventual selling price.

Monitor

It is critical that companies continue to monitor ML applications in deployment. They need to verify that the algorithm is working as expected in the real world. One characteristic of ML apps is that they might continue to “learn” and adjust as they function in the deployment environment. So companies also need to be aware if the application is learning correctly and continuing to perform as expected.

Various forms of “drift” can occur. Drift refers to a deployed model becoming less accurate over time. You might see drift if the data changes—maybe the training data was not actually representative of the real world.

Drift in “concept” can also occur. A “concept” drift means that the correlation between input parameters and outputs change. For example, housing prices can be subject to a great deal of drift. Perhaps during the COVID-19 pandemic, people preferred larger houses, but, after the pandemic, other features might start to become more important than size.

Monitoring helps companies to detect drift so that they can determine whether they need to build, train, and deploy a new model.

Supervised training

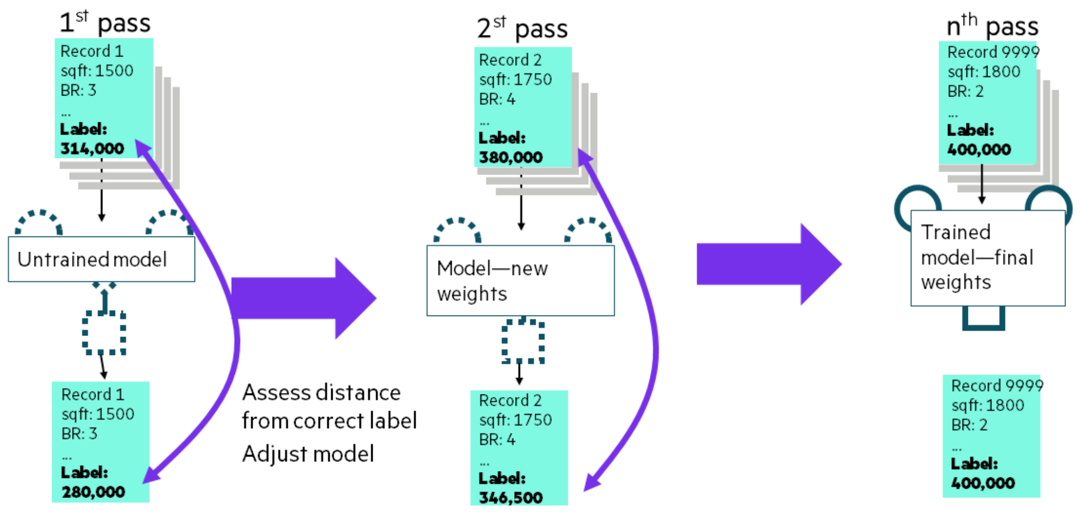


Figure 1-6: Supervised training

You will now dive into more detail on training, which falls into two broad categories: supervised and unsupervised.

With supervised training, the data records used to train the model include labels, which answer the question the model is intended to answer. In this example, the label is the actual selling price for the house. Supervised training often works well for classification and regression tasks. The answers for training data are known. You know the actual selling price for the houses in your training data set; you know that a particular image shows a cat.

Training consists of a series of "passes" of data through the model. After each pass, the training process assesses the model performance. Based on that performance, it adjusts the model and then tries again.

Consider an example first pass. Record 1 is a house of 1500 square feet with three bedrooms. The training process submits that data to the untrained model, which might be:

Price in $US = 100*(area in square feet) + 10,000*(bedrooms) + 100,000

The output is $280,000.

Because you are using supervised training, the record included a label: $314,000. The training process assesses the difference between the output label and the original label. It then adjusts the model weights in an attempt to do better. It might decide to weigh area and bedrooms more heavily and to increase the bias—as just one example:

Price in $US = 110*(area in square feet) + 11,000*(bedrooms) + 110,000

After adjusting the model, the training process conducts a second pass on a *new* record. Again, it assesses the result and its difference from the accurate label. It makes a further adjustment to the model weights.

The process continues for pass after pass until the model has been trained on the desired amount of data. The model is now considered "trained," and it should be performing very well.

This example shows the model analyzing one record per-pass. But often ML engineers set up training such that the model processes a "batch" of several records at each pass.

Unsupervised training

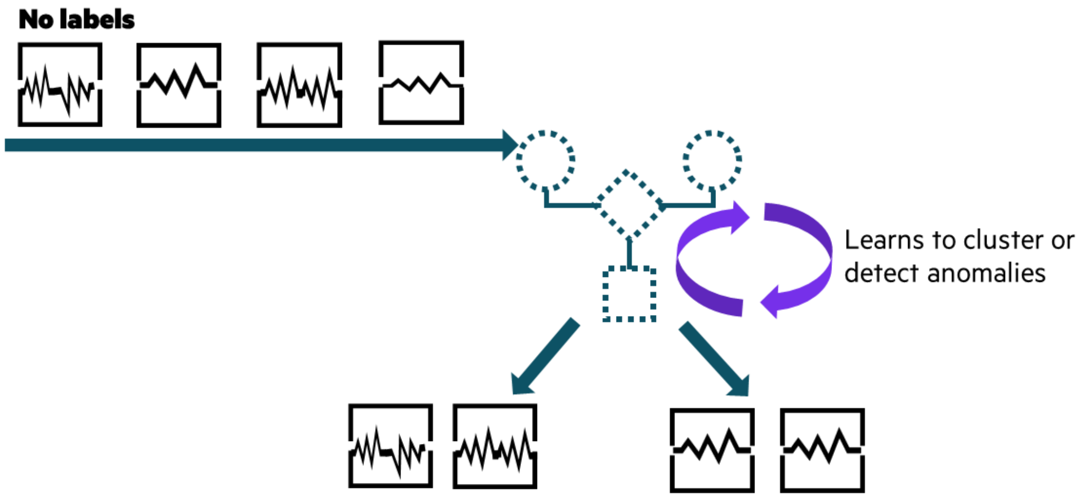


Figure 1-7: Unsupervised training

ML can also use unsupervised training, in which the model is trained with unlabeled data. In this case, the model attempts to find relationships between pieces of data without being told anything more about the data. Unsupervised training can work well for “clustering” tasks in which the application needs to group similar data together. It can also work well for applications that need to detect normal versus anomalous behavior, such as monitoring or security applications. Unsupervised training usually requires a vast amount of data to work well.

What is deep learning (DL)?

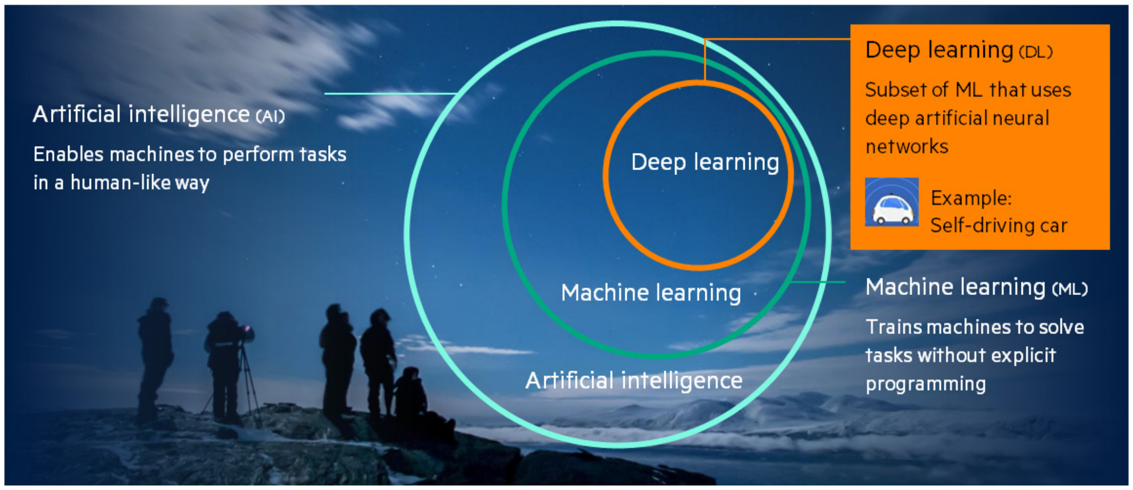

Artificial intelligence (AI)
Enables machines to perform tasks in a human-like way
Deep learning
Machine learning
Artificial intelligence
Deep learning (DL)
Subset of ML that uses deep artificial neural networks
Example: Self-driving car
Machine learning (ML)
Trains machines to solve tasks without explicit programming

Figure 1-8: What is deep learning (DL)?

Deep learning is a further subset of ML. Rather than train a model based on a single algorithm, DL uses multiple layers of interacting algorithms. These algorithms establish a “neural network,” which can execute highly complex tasks. A self-driving car is one example of a DL application.

Artificial neural networks (ANNs) for DL

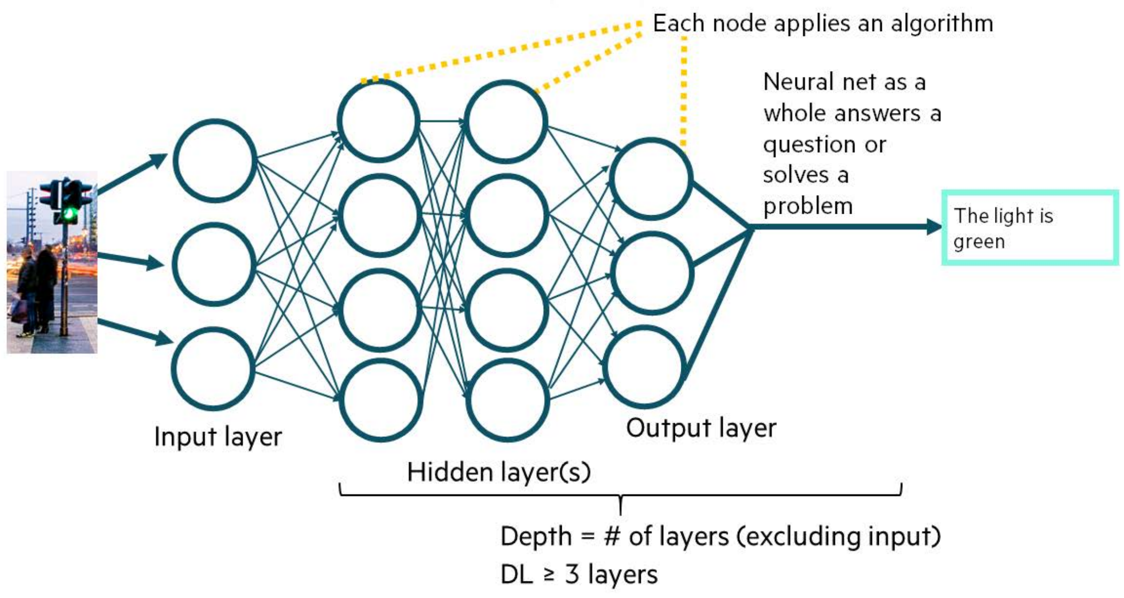

Each node applies an algorithm
Neural net as a whole answers a question or solves a problem
The light is green
Input layer
Hidden layer(s)
Output layer
Depth = # of layers (excluding input)
DL ≥ 3 layers

Figure 1-9: Artificial neural networks (ANNs) for DL

DL uses artificial neural networks (ANNs), sometimes also called deep neural networks (DNNs). An ANN consists of several layers of nodes (sometimes called “neurons”). Each node is actually an algorithm. The algorithm’s inputs come from the output of nodes in the previous layer. The algorithm then creates an output, which it passes to nodes in the next layer.

You will look at this process in more detail in a moment. For now, simply understand that all of the nodes work together to fulfill a task. And that task is typically much more complex than the task performed by ML using a single algorithm. For example, DL can be used for computer vision. The DL application “sees” and recognizes an object such as a green traffic light.

ANNs come in many sizes, but they always consist of:

- An input layer, which brings data into the ANN
- Hidden layers, which apply algorithms to data
- An output layer, which applies further algorithms and outputs the result, such as making a prediction or applying a label to categorize data

The depth of the ANN equals the number of layers, excluding the input layer. DL specifically refers to ANNs that have three or more layers, or, in other words, at least two hidden layers.

Common DL use cases

- Image recognition and computer vision
  - Examples: autonomous vehicles, healthcare
- Natural Language Processing (NLP)
  - Examples: digital assistants, transcription

Figure 1-10: Common DL use cases

Image recognition and computer vision are two common use cases for DL. Within these use cases lie many applications, of which these are just some examples:

- Seeing traffic and current conditions for autonomous driving vehicles
- Identifying areas of concern in medical images
- Automatic image labelling for social media platforms or for data archival
- Face identification for camera applications
- Facial recognition for identification purposes
- Seeing objects for manufacturing processes

Much as DL can help machines "see" and recognize objects, DL can also help machines "hear" and recognize words using natural language processing (NLP). NLP can refer to processing and interpreting audio language or textual language. And it typically also enables the application to formulate its own appropriate response. Think, for example, of a digital assistant that hears you say, "Create an appointment for tomorrow at 10 am," adds that appointment to your calendar, and then tells you what it did.

Again, NLP enables many different types of applications, including:

- Audio-enabled digital assistants
- Chat bots
- Auto-captioning and transcription

A deeper look at DL training

Because HPE Cray AI Development solutions are specifically designed to accelerate DL use cases, you will now examine ANNs and DL training in more detail.

The goal of this section is *not* to make you an expert in DL, as that is impossible in the length of a brief module. However, you will learn enough to have a conversation with customers about their ML/DL deployments and understand their general challenges.

As a final note, in this section, and throughout this course, you should take references to “ML engineers” to refer to engineers developing any type of ML models, including DL ones. For some companies, “data scientists” take a similar role.

What the ANN input layer does

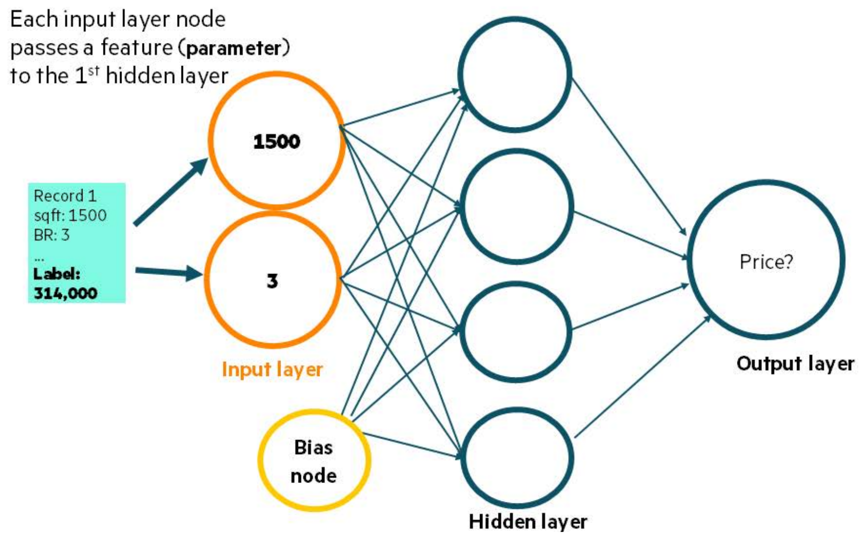


Figure 1-11: What the ANN input layer does

You will begin by examining an ANN in more detail and learning the function of each layer. So that you can better understand the *broad* concepts, you will continue to examine the same simplified use case from earlier in this module: a housing price predictor. Be aware, however, that DL would be overkill for this particular use case, as DL typically applies to more complex image recognition, computer vision, and NLP use cases.

The input layer is responsible for extracting the features, or parameters, from the data record that the model is processing. The input layer has one node for each parameter in question, such as area and number of bedrooms in this example. Each input layer node then passes its parameter's value for this data record on to each node in the first hidden layer.

Often a "bias" node also connects to the first hidden layer. This bias node helps to "push" outputs in the next layer toward one direction or another. The ANN will gradually learn how heavily to weigh the bias just as it learns how heavily to weigh various incoming parameters.

Note that bias here does *not* refer to the type of the *undesirable* bias about which you learned earlier, in which issues such as data and parameter selection cause a model to include prejudices based on race, gender, or other factors.

What ANN hidden layers do

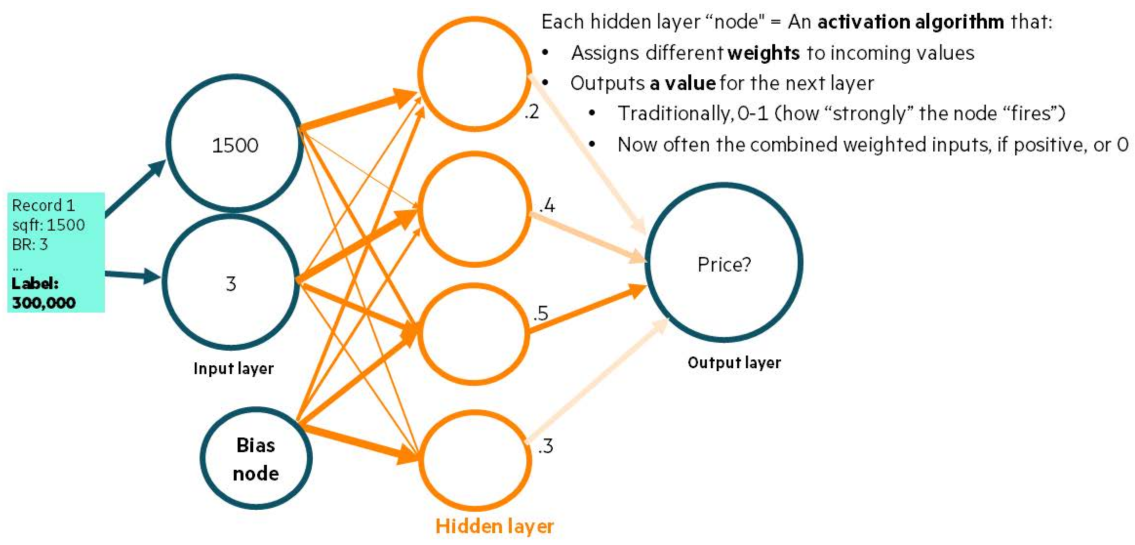

Each hidden layer "node" = An **activation algorithm** that:
• Assigns different **weights** to incoming values
• Outputs **a value** for the next layer
• Traditionally, 0-1 (how "strongly" the node "fires")
• Now often the combined weighted inputs, if positive, or 0
Record 1
sqft: 1500
BR: 3
...
Label: 300,000
1500
3
Input layer
Bias node
.2
.4
.5
.3
Hidden layer
Price?
Output layer

Figure 1-12: What ANN hidden layers do

Each hidden layer node applies an "activation" algorithm using values input from the previous layer and the model weights. Traditionally, the activation algorithm was usually a sigmoid function, which uses the weighted inputs to output a value between zero and one. You can think of that output as telling the "neuron" how strongly to "fire" with 0 meaning do not fire, .1 meaning to fire weakly, .9 meaning to fire strongly, and so on. Other activation algorithms, such as rectified linear unit (relu), have become popular. The relu algorithm combines the weighted inputs and outputs that value, if it is positive; otherwise, it outputs zero. The choice of activation algorithm can affect the learning process and ultimate accuracy of the model, but a detailed examination is beyond the scope of this course.

To clarify these concepts, consider our simplified example. Every node in the first hidden layer receives all of the data record's parameter values from the input layer. However, each node has its own weights that it applies to those parameters. Remember: weights are the model's "secret sauce," or, in other words, what make one model work better than another model. In this example, the first hidden layer node might weigh the house's area highly, as indicated by the thick line in the figure. The next hidden layer node might weight the number of bedrooms more highly, but area less so. Of course, in a model with thousands or even millions of parameters, the model becomes much more complicated, but you can understand the general concept.

Using these weights and the values from the input layer, each hidden layer node applies its activation function. As you see, some nodes might "fire" more strongly—essentially passing on the signal that the house price will be higher—while others might fire more weakly.

What is the benefit of passing the parameters through the hidden layer before inputting them into the output layer? In this example, the model as a whole is using linear regression to predict housing prices based on a variety of factors. The hidden nodes, by applying activation functions, introduce non-linearity into the model. While the benefits of non-linearity might be minimal for the highly simplified use case shown here, it is essential for more complex use cases like image recognition.

What the ANN output layer does

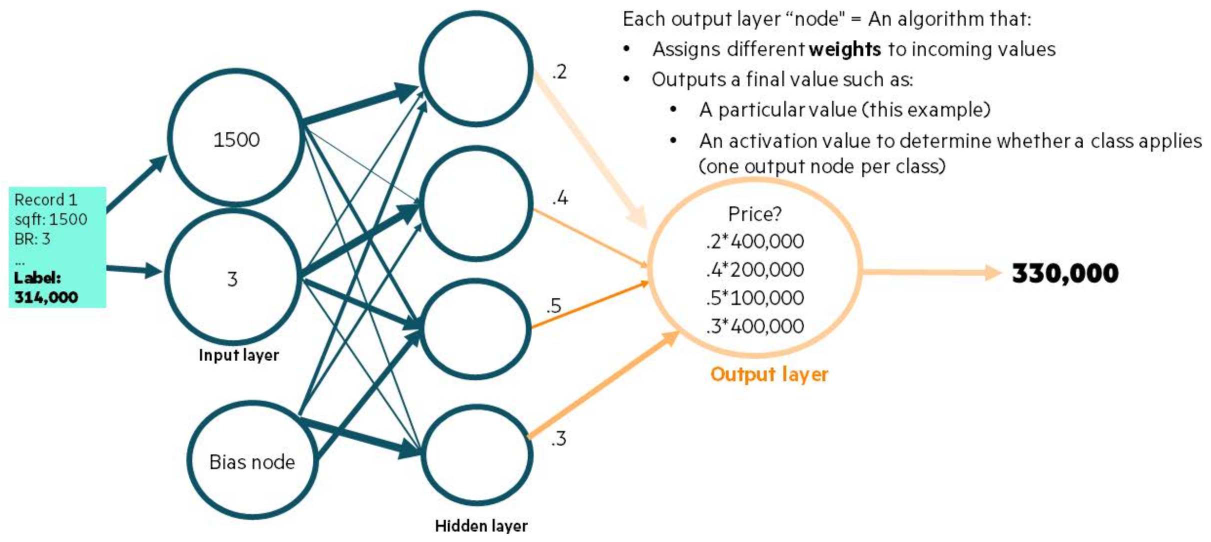

Each output layer "node" = An algorithm that:
• Assigns different **weights** to incoming values
• Outputs a final value such as:
• A particular value (this example)
• An activation value to determine whether a class applies (one output node per class)
Record 1
sqft: 1500
BR: 3
...
Label: 314,000
1500
3
Input layer
Bias node
Hidden layer
.2
.4
.5
.3
Price?
.2*400,000
.4*200,000
.5*100,000
.3*400,000
Output layer
330,000

Figure 1-13: What the ANN output layer does

The output layer consists of nodes that apply algorithms much like hidden layer nodes. However, the output layer nodes are responsible for the ANN's final decisions. Each output node receives inputs from the final hidden layer. It applies varying weights to the incoming values in an algorithm that produces a final result.

In this example, the ANN has a single output node, and its algorithm is a linear function for calculating housing price. The output node essentially multiples the value from each hidden layer node by a weight and adds them up for its final answer.

For a classification use case, though, the output layer has a node for each potential class (label). Each output node determines whether its class applies or not. For example, consider an ANN that classifies images of cats and dogs. It would have a "cat" output node and a "dog" output node. The "cat" output node would learn to weigh strongly the inputs from hidden nodes that fire strongly for "cat-like" parameters. It would learn to weigh weakly the inputs from hidden nodes that fire strongly for "dog-like" parameters. Each output node applies an activation algorithm (such as sigmoid or relu) and outputs a value. The ANN generally then applies the class of the node outputting the strongest signal (highest value).

Training the ANN (forward pass)

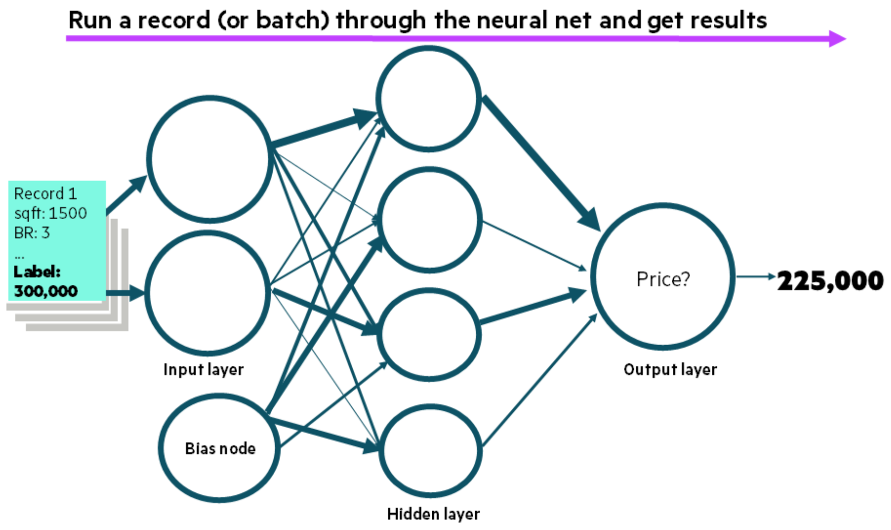


Figure 1-14: Training the ANN (forward pass)

Now that you understand a bit about how the ANN processes data, you can explore how the ANN is trained.

The model begins with randomized weights for each node's connection to a node on the preceding layer. Of course, these randomized weights are unlikely to produce good results at first. The model needs to be trained on data so that it can adjust its weights and perform more successfully.

With ANNs it can be useful to break this process down in a bit more detail. Each cycle starts with a “forward pass.” That is, the training process runs a record, or often an entire batch of records, through the ANN, and the ANN outputs results.

Training the ANN (backward pass)

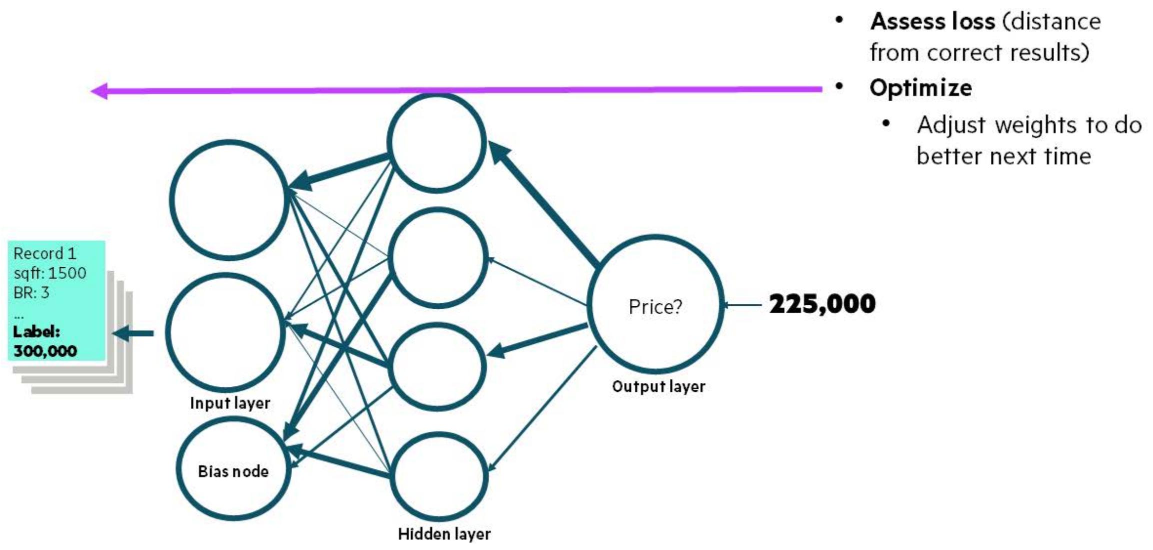


Figure 1-15: Training the ANN (backward pass)

Now the training process assesses loss—or in other words, distance between the ANN model's output and the correct answers. The loss assessment proceeds from the output layer back through the hidden layers in reverse order. Therefore, this part of the training process is often called a “backward pass.”

The backward pass also involves optimizing. As you recall, every node applies a specific weight to the input sent by each node to which it is connected in the previous layer. In the figure, these weights are represented by relatively thick (high weight) or thin (low weight) lines. The optimizer function calculates how to adjust all these weights to minimize loss. You can think of optimization as strengthening some node-to-node connections and weakening others. The end result is that the ANN can better receive inputs and output an accurate answer.

Optimization typically involves some form of “gradient descent” algorithm. If you would like to learn more about these algorithms, refer to the appendix at the end of this module.

End result of ANN training

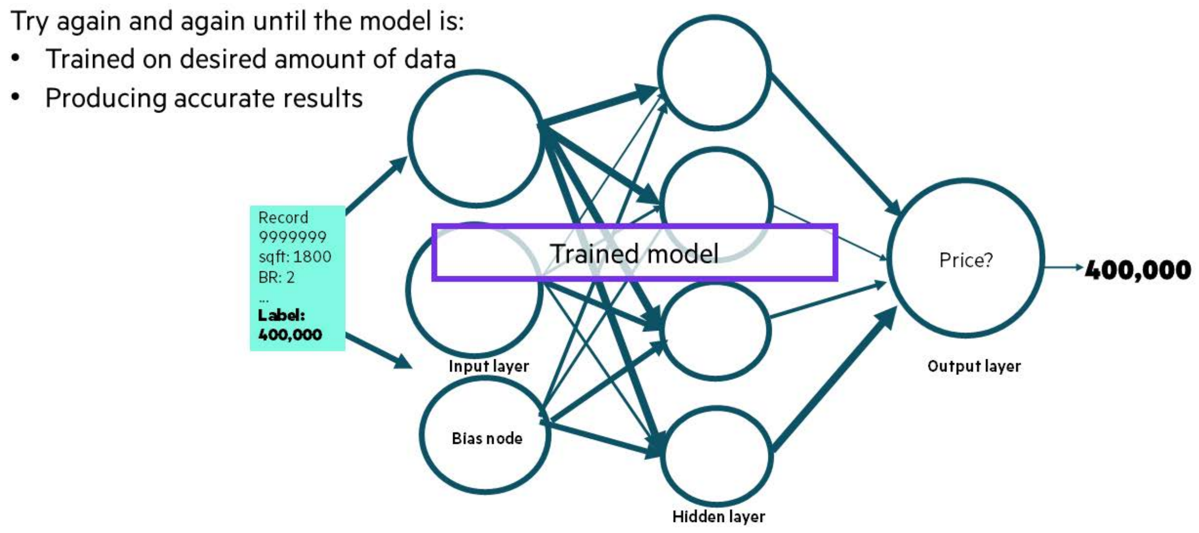


Figure 1-16: End result of ANN training

The cycle of forward and backward passes continues until the ANN has processed the desired amount of data. At that point the model is trained, and it should be performing accurately. Few models achieve one hundred percent accuracy, but, of course, ML engineers attempt to maximize accuracy. (And, as you will explore in more detail in a moment, training accuracy is not always the same as accuracy in the real world.)

ANNs and image recognition

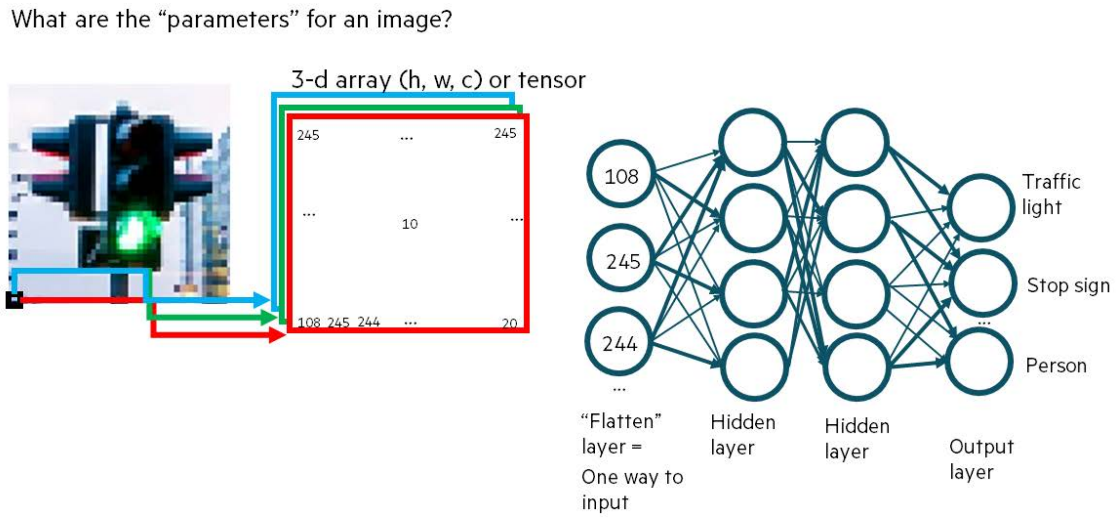


Figure 1-17: ANNs and image recognition

Up to now you have been examining how ANNs can address a relatively simple use case. You can easily understand how factors like area and number of rooms affect a house's selling price. But ML engineers more often use ANNs for complex use cases such as image recognition. What are parameters for an image?

Typically, these parameters are a three-dimensional array, or tensor, that describes every pixel in the image.

A tensor is simply a multi-dimensional array. You can think of a two-dimensional (2-d) tensor as organizing data in a rectangle of columns and rows, while a three-dimensional (3-d) tensor organizes data in a rectangular prism. And tensors can feature even more dimensions.

In the computer vision example, a 2-d image consists of a 3-d tensor. Two dimensions are based on the image height and width. For example, if an image is 64 pixels by 64 pixels, the tensor consists of multiple 2-d arrays that are 64 by 64. The third-dimension, often called "channel," describes the depth of the tensor. If the image is grayscale, it has a depth of one. A color image, though, is a 3-d array with a depth of three: it has a "red," "green," and "blue" channel, and each channel consists of the same 2-d shape as the image.

As you see in the figure above, each pixel is mapped onto the tensor based on its location within the image. The value depends on the pixel's color. The lower left pixel's red value is placed in the lower left of the 2-d array for the red channel, its green value in the lower left of the 2-d array for the green channel, and its blue value in the lower left of the 2-d array for the blue channel. The next pixel to the left has its red, blue, and green values placed in the next space to the left in the red, blue, and green arrays. The example array in the figure above does not map out every value, but you can imagine how they would look when fully mapped out. Together these three 2-d arrays create a 3-d array.

(A grayscale pixel has a single value describing how dark it is. Again, a grayscale image is a 3-d array with a depth of one.)

The 3-d tensor, describing the image, passes into the ANN. Often, the tensor is actually 4-d; the fourth dimension lets the ANN process multiple images at once.

As you recall, the input layer is responsible for receiving parameters. The figure above shows one way that the ANN might receive these parameters. The input layer "flattens" the tensor into a 1-d array, which strings together all the values held within the multi-dimensional shape into one long line. The figure above, of course, does not have space to show all the inputs. A color image of only 64 by 64 pixels would input 12,288 parameters (64x64x3).

The ANN then continues to function as you learned before. The first hidden layer of the ANN receives all of those values, weighing some more highly than others. Each node performs an activation function and decides how strongly to fire, passing on its activation value to the next hidden layer. This process continues until data passes through the output layer nodes. In this example, the strongest firing node determines which class is assigned to the image. Is the image a traffic light, a stop sign, a person, or any number of other classes defined in this model?

But how does weighing particular pixels more or less highly help the ANN figure out the class? It can seem rather opaque. The next pages explain more.

Simplified look at convolutional neural networks (CNNs)

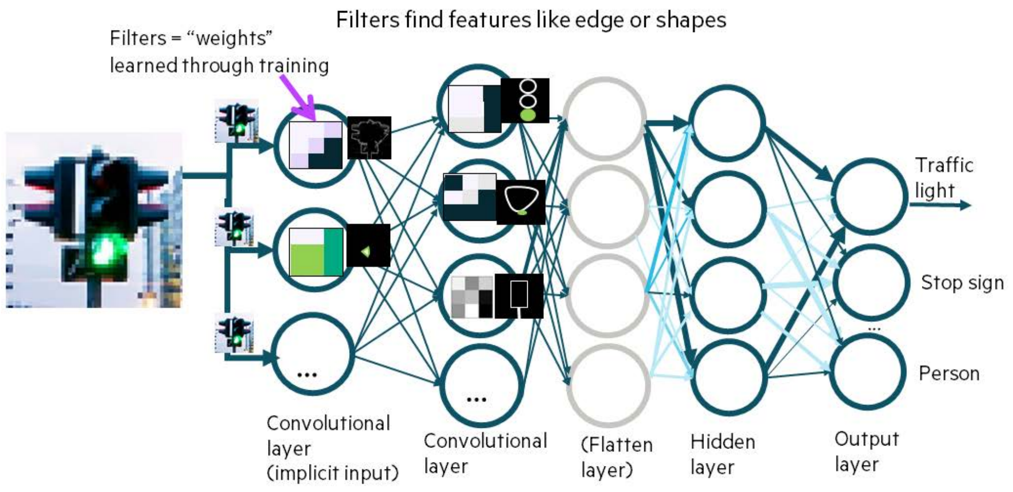

Filters find features like edge or shapes
Filters = "weights" learned through training
Traffic light
Stop sign
Person
Convolutional layer (implicit input)
Convolutional layer
(Flatten layer)
Hidden layer
Output layer

Figure 1-18: Simplified look at convolutional neural networks (CNNs)

The most successful models for image recognition and NLP are often convolutional neural network (CNNs). A CNN is an ANN that uses convolutional layers with *filters* to recognize an image, sound, or other complex input.

The figure shows a highly simplified example of a CNN used for computer vision. The CNN receives a camera image as a 3-d tensor, much as you learned on the previous page (or multiple images as a 4-d tensor). But the first CNN hidden layer is a "convolutional" layer, which operates on the image in order to extract the most relevant features from the image. (In the figure above, the convolutional layer is shown as the first layer, but there is also actually an implicit input layer.)

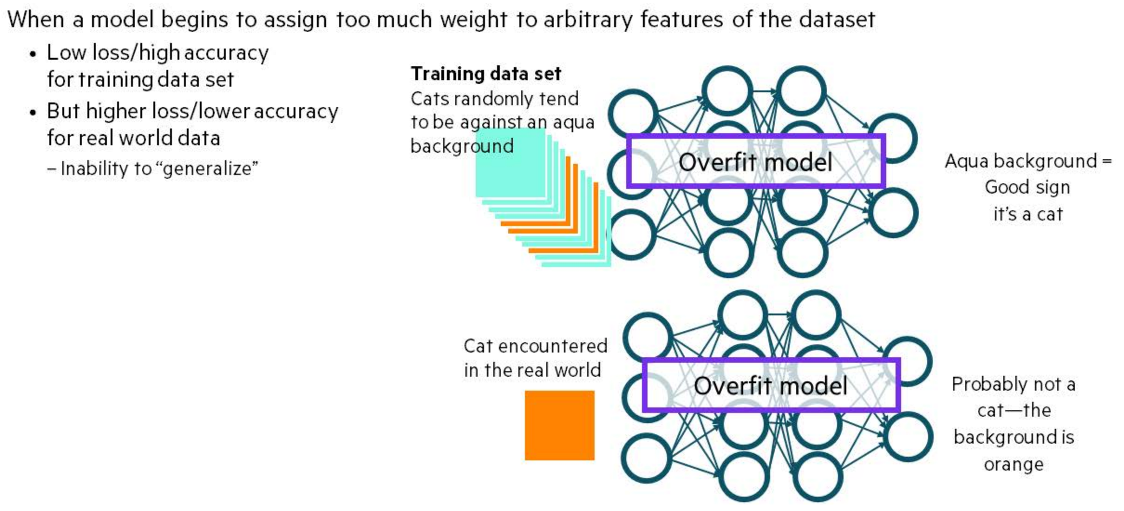
The convolutional layer extracts the relevant features by applying "filters." As an analogy, think of a small transparent film, which you slide over the image. The example above shows a 3x3 pixel filter with a gradient from light to dark. If the filter matches up with the image below, you are likely to have found an edge between a light and dark object—such as the sky and traffic light in the example image. By sliding the filter across the complete image, you can map out the edges around the traffic light. Then you can pass the filtered image to the next hidden layer. The convolutional layer consists of multiple nodes, each of which applies a different filter. In this way, it can extract multiple features.

Often the first convolutional layer finds edges. In this example, they simplify the image, rendering it more like a line drawing. The next hidden layer might be another convolutional layer that applies more filters to further home in on the relevant features for the model's task. Some filters might find particular shapes that indicate a traffic light, for example, while other filters might find shapes that indicate a stop sign.

In a simple black and white image, finding edges would be relatively easy. But real-world images are complex. A CNN starts out with random filters, which it then trains as part of the training process—essentially teaching itself how to see edges and other features. Each node in the convolutional layer acts as one filter, and the filter values are the weights that are learned at that layer.

Although this example shows just two convolutional layers, a computer vision CNN might consist of many convolutional layers. In the end, these layers typically pass what they have learned to a hidden layer that uses activation functions, such as the ones you have examined earlier. These layers are often called fully-connected or dense layers. (Perhaps the model also flattens the image into a 1-d shape on the way.) From there, the CNN works much as you have seen before. Some final hidden layer nodes will learn to weigh most strongly the input from nodes in earlier layers that find " traffic-light-like" features. The traffic light output node will learn, in turn, to weigh *those* hidden layer nodes input most strongly. And the same holds true for other nodes and "stop-sign-like" features or "person-like" features. The strongest firing output node assigns the label to the image. In this example, the CNN determines that the image belongs to the "traffic light" class.

Overfitting


When a model begins to assign too much weight to arbitrary features of the dataset

- Low loss/high accuracy for training data set
- But higher loss/lower accuracy for real world data
  – Inability to "generalize"

Training data set
Cats randomly tend to be against an aqua background
Overfit model
Aqua background = Good sign it's a cat
Cat encountered in the real world
Overfit model
Probably not a cat—the background is orange

Figure 1-19: Overfitting

Successful DL models need to be trained on vast amounts of data, hundreds of thousands or even millions of records. Sometimes ML engineers will train the model on the full data set, and the model is still not performing up to the required accuracy. The ML engineers can then continue to train the model on multiple iterations of the full data set. Although the model is technically not receiving new information, the model is getting more chances to adjust its weights and decrease loss. You should be aware, though, that training a model on 100 iterations of a data set with 1,000 records will not produce as good of results as training a model on 1 iteration of a data set with 100,000 different records.

A primary reason for that fact is that training a model on the same data over and over can cause "overfitting." Overfitting can also occur for other reasons, such as issues with how the model is trained.

An overfit model has learned arbitrary features from the data set as if those features are meaningful. Consider a set of images of animals used to train a DL image recognition app. In this data set, the images of cats simply happened to have an aqua background slightly more often than images of dogs or other animals. An overfit model begins to assign too much weight to that fact. When deployed in the real world, the overfit model might mislabel cat images on an orange background or mislabel dogs as cats simply because the dog images have an aqua background.

In more precise terms, an overfit model has reached very low *training* loss or, conversely, very high *training* accuracy; however, it has higher loss, or lower accuracy, when processing new data. In other words, it does not do well at generalizing what it has learned when it starts operating in the real world.

Validating a model

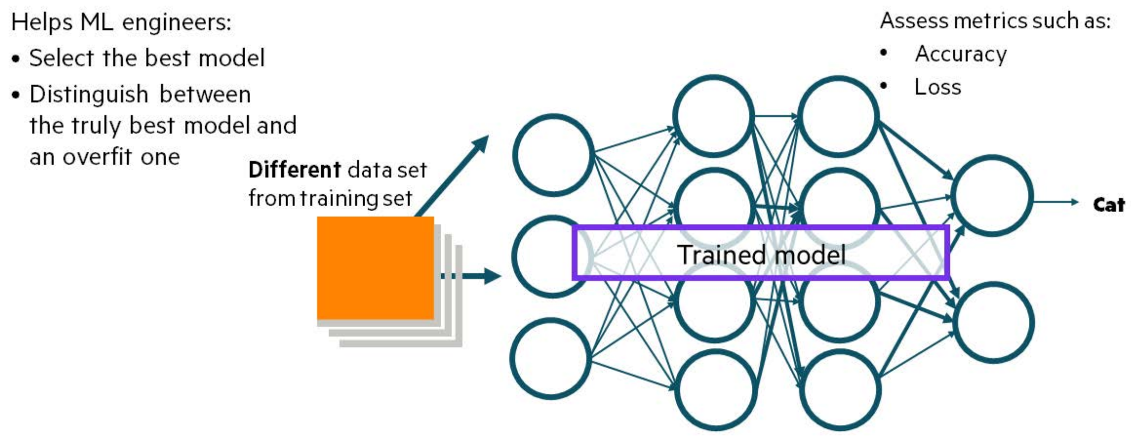

Helps ML engineers:
• Select the best model
• Distinguish between the truly best model and an overfit one
Different data set from training set
Trained model
Assess metrics such as:
• Accuracy
• Loss
Cat

Figure 1-20: Validating a model

In their efforts to create a model that performs as well as possible, ML engineers might experiment with multiple training processes for the same untrained model. They might even train a model for a period, pause and save the current model, continue the training, pause and save again. As a result, the ML engineers end up with multiple models. How can the ML engineers select the best one? Perhaps they could simply select the model that has reached the lowest loss during the training process. With that approach, they would sometimes select the best model. But sometimes they would select an overfit model, as opposed to a model that is performing a bit less well on training data, but might actually perform better in the real world.

To ensure that they select the actual best model, ML engineers “validate” their candidate models. The validation process uses a *different* data set from the training data set. Then each model has the chance to show how well it can apply what it learned in the training process when processing new data. After the model has processed records from the validation data set, the model’s loss and accuracy is assessed. ML engineers can then select the model with the lowest *validation* loss or, if they prefer, the highest *validation* accuracy.

What are ANNs: Another perspective

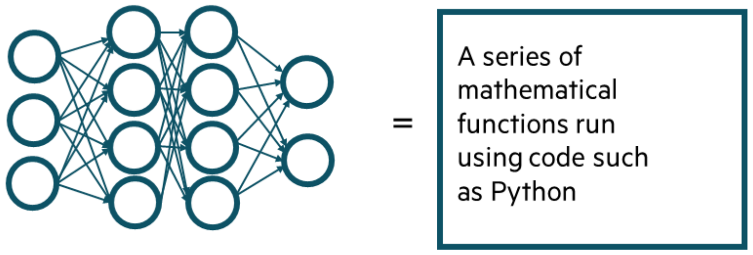
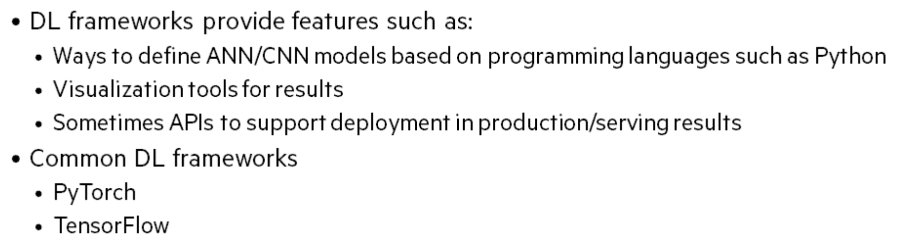

=
A series of mathematical functions run using code such as Python

Figure 1-21: What are ANNs: Another perspective

You have spent quite a bit of time looking at ANNs (including CNNs, as a subset of ANNs) from a conceptual level, as a system of interconnecting nodes. Now consider ANNs from a lower level perspective. You can also think of an ANN as a set of mathematical functions that are executed using code such as Python. The code defines each function (ANN node) and its inputs. The inputs for one function are variables, set by the outputs from other functions, establishing the node-to-node connections.

Code for training a model would make the weights in functions variables too. The code would then also include loss assessment and optimizer functions; the optimization results would set the variable weights for the next iteration of the ANN. And the code would be set up to loop through the process again and again to train the model.

ML/DL frameworks



- DL frameworks provide features such as:
  - Ways to define ANN/CNN models based on programming languages such as Python
  - Visualization tools for results
  - Sometimes APIs to support deployment in production/serving results
- Common DL frameworks
  - PyTorch
  - TensorFlow

Figure 1-22: ML/DL frameworks

In theory, ML engineers could program an ANN from scratch in Python or another coding language. They could also program the code for training the ANN and validating it—all from scratch. But in practice scientists naturally prefer to work within a framework that includes pre-defined building blocks for these tasks.

A DL framework builds on common coding languages such as Python and Java. It defines standard classes for components such as ANNs, as well as standard functions for processes such as passing data through the ANN (forward pass) or optimizing the ANN (backward pass). ML engineers can then easily define a model and training process by invoking those predefined components.

PyTorch and TensorFlow are two very commonly used DL frameworks.

PyTorch

PyTorch is an open source library for building ML and DL models. It specializes in tasks such as language and visual processing. It primarily uses Python coding, but also supports C++ and Java.

TensorFlow

TensorFlow, developed by Google Brain, is an open source library for ML and DL. Developers can use TensorFlow libraries for building code, training their models, and even deploying the models. TensorFlow libraries exist for several popular programming languages, including Python and Java. TensorFlow is often used for GPU-accelerated algorithms and establishing neural networks for DL.

TensorFlow can use Keras as the API for interacting with Python code. Keras attempts to make coding neural networks more approachable to more people. It also helps with hyperparameter optimization (HPO) about which you will learn more later in this module. TensorFlow can alternatively use Estimator; however, Estimator is currently less emphasized.

TensorBoard

TensorFlow and PyTorch also integrate with additional tools, which help to support the training process. For example, both TensorFlow and PyTorch support TensorBoard, a visualization tool that helps ML engineers view and analyze the results of a training process.

ML/DL ecosystem

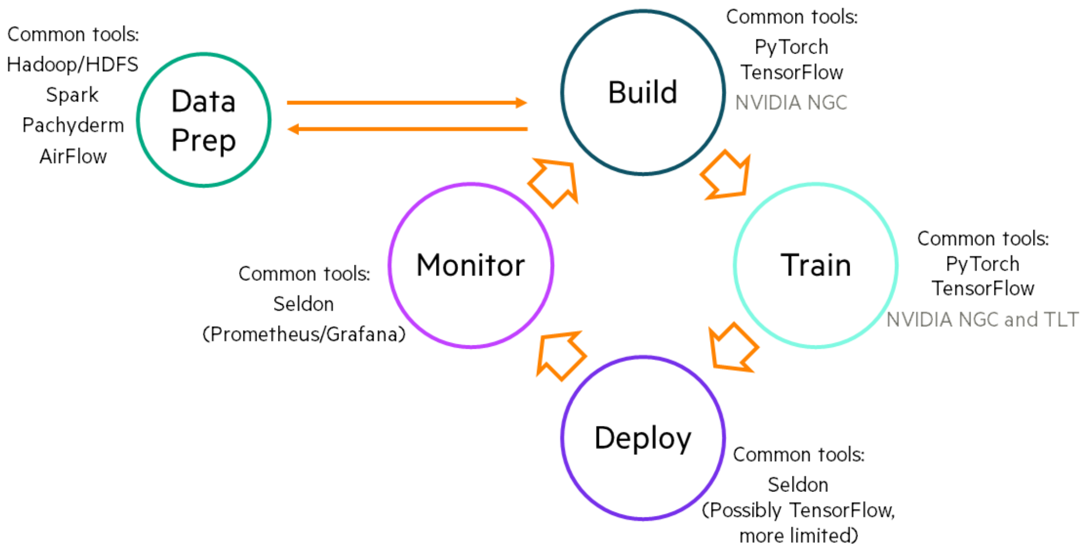


Figure 1-23: ML/DL ecosystem

Remember that building and deploying ML/DL applications involves a complete lifecycle. ML engineers typically use PyTorch and TensorFlow at the build and train stages of that lifecycle. (For non-DL machine learning, though, they might use Spark.)

To manage the complete lifecycle, ML engineers need additional tools and applications. The figure shows some common ones for each stage; however, other tools exist in the ever-expanding ML/DL ecosystem. The sections below provide a bit of background information about each of the listed tools and applications.

Apache Hadoop/HDFS

Apache Hadoop provides a framework for running big data analytic applications on data stored in Hadoop Distributed File System (HDFS). Some companies use data they have already stored in HDFS for their ML/DL use cases.

Apache Spark

Apache Spark is a framework for analyzing and processing data with in-memory computing. Companies might already have data in a cluster using Spark. ML engineers might use ML libraries for Spark to take that data and start using it to train their models.

Pachyderm

Pachyderm provides data management, specifically designed for ML and DL use cases. It gives ML engineering teams unified data repositories with features such as data versioning. Data pipelining helps to automate the incorporation of new data into the ML/DL lifecycle.

Apache AirFlow

Apache AirFlow provides a platform for workflow management. In addition to managing workflows across the ML/DL lifecycle, AirFlow can help companies load and prepare data for ML/DL applications.

Seldon

Seldon provides solutions for deploying, or serving, ML/DL applications at scale. Seldon solutions help clients or other applications communicate with the trained model, submit data to it, and receive results.

(Specifically Seldon offers several solutions, including Seldon Core and Seldon Alibi. Seldon Deploy integrates Seldon Core, Seldon Alibi, and KFserving.)

Seldon solutions can detect drift in the deployed models. (As mentioned earlier, drift refers to changes over time, which cause the deployed model to perform less accurately.)

Prometheus/Grafana

Prometheus is an open-source monitoring solution for a broad range of applications. It receives metrics and stores them in a time-series database. It then helps users visualize that data with graphs and charts. Prometheus often uses Grafana as the visualization dashboard in which it presents data visualizations. Prometheus can also provide alerting for when a particular metric passes a defined threshold.

Seldon can expose metrics to Prometheus/Grafana so that companies can monitor their deployed ML/DL applications. These metrics can include performance related metrics, such as requests per second. Customers can also send custom metrics such as model accuracy so that they can detect issues with falling accuracy.

NVIDIA NGC and TLT

NVIDIA has assembled a catalog of components for NVIDIA GPU-optimized software related to HPC and AI. These components include container images, software development kits (SDKs), and even pretrained models for use cases such as NLP or computer vision. Many customers like to begin with NGC models; they can then use NVIDIA Transfer Learning Toolkit (TLT) to continue training those models on their own data for their own particular applications.

ML/DL pipeline applications

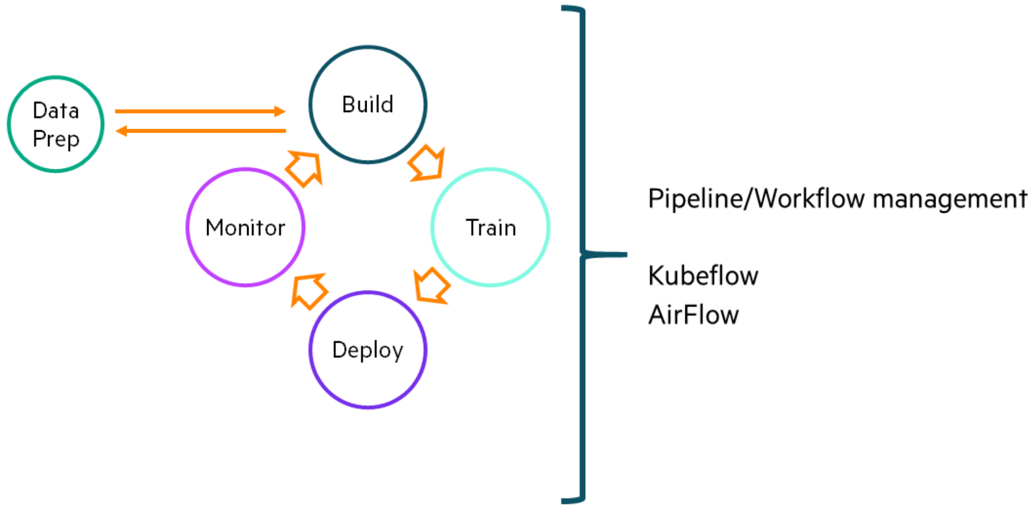


Figure 1-24: ML/DL pipeline applications

Customers need tools that extend across the ML/DL lifecycle, helping to manage the movement of an application from one stage to another. These tools establish pipelines, which are sometimes called workflows. A pipeline automates the lifecycle. Users could run a single pipeline, which prepares data, trains the defined model, validates the model, and deploys the model. A complex pipeline might feature branches in which different next steps are taken, based on the results of the previous step.

Pipeline applications, such as Kubeflow and AirFlow, make it easier for companies to create these pipelines and manage the ML/DL application workflow.

Kubeflow

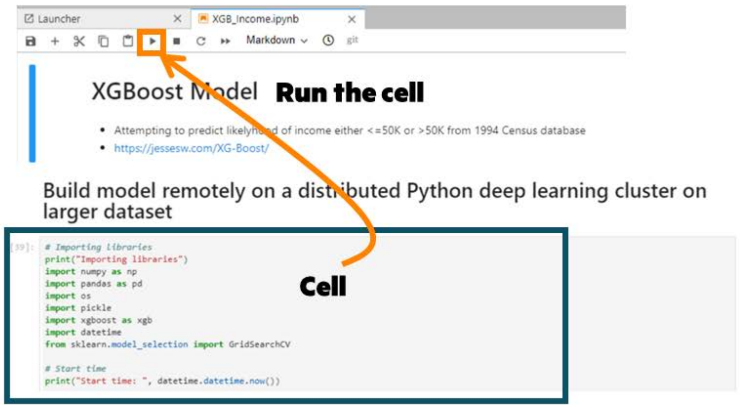
Kubernetes provides orchestration for a cluster of servers that host applications in containers (usually Docker containers). Kubeflow brings together an open source toolkit for building and training ML/DL applications in Kubernetes environments. It integrates frameworks such as Spark, PyTorch, and TensorFlow, enabling users to run these applications on the Kubernetes cluster. Kubeflow also provides pipelines for managing the ML/DL lifecycle.

AirFlow

AirFlow is an open-source platform for workflow management. It helps ML engineers use familiar Python to easily define pipelines that work in a variety of environments. AirFlow further features a UI for scheduling and managing pipelines, as well as checking on the status of tasks executed by the pipelines.

A quick look at Jupyter Notebook

- Jupyter Notebooks:
  - Consist of cells of comments (markdown) and code
  - Are commonly used to hold code for ML/DL models
- To execute code, users run each cell in turn



XGBoost Model

- Attempting to predict likelyhood of income either <=50K or >50K from 1994 Census database
- https://jessesw.com/XG-Boost/

Build model remotely on a distributed Python deep learning cluster on larger dataset

```
[39]: # Importing libraries
      print("Importing libraries")
      import numpy as np
      import pandas as pd
      import os
      import pickle
      import xgboost as xgb
      import datetime
      from sklearn.model_selection import GridSearchCV

      # Start time
      print("Start time: ", datetime.datetime.now())
```

Figure 1-25: A quick look at Jupyter Notebook

You should also be aware of an additional application, which ML engineers often use during the building or even training stage of the ML/DL lifecycle: Jupyter Notebook.

Jupyter Notebook is an open source solution for interactive documents. As you see, a Notebook consists of cells. A cell can be a markdown cell (which contains comments) or a code cell. In this way, users can mix words and code within the same document. ML engineers like to use Notebook because they can embed data and data visualizations directly into the document. They can share their code with each other, together with comments on the code. They can then execute code directly within the Notebook by playing (running) the cell.

JupyterLab is an interface for creating, opening, and running Notebooks.

Module 1—Lab 1

You will now explore some of these concepts using an HPE Cray AI Development Environment cluster already deployed in your lab environment. You are sharing this cluster with your classmates, so be very careful to only execute the steps outlined in the lab.

You will begin by launching a JupyterLab environment. You will then load a Jupyter Notebook provided to you and start to run it. This Notebook will show you an example ANN created with TensorFlow Keras. This particular ANN is intended to classify images, but, as you will see, the untrained ANN model does not yet do well at this classification.

You will then view a completed “experiment,” which refers to the process of training a DL model in Cray AI Development Environment. This experiment happened to train the same model that you saw defined in the Jupyter Notebook. You will see the metrics collected during the training and how loss decreased and accuracy increased.

After exploring this metrics, you will return to the Notebook. You will continue to run it to load the model that was trained during the experiment. You will then see how the trained model can more accurately classify images.

Next you will explore an example CNN created with TensorFlow Keras. You will view a completed experiment that trained that same CNN. Finally, you will see how well the trained CNN performs, as well as explore the CNN's filters.

Module 1—Lab 1 debrief

Discuss these topics with your classmates:

- The HPE Cray AI Development Environment Web UI showed records of “experiments” that trained an image recognition model. What were you able to see about the training process in the **Overview** tab?
- When you viewed the graph in the **Overview** tab, what difference did you notice between the training accuracy and the validation accuracy?
- Compare the performance between the untrained model, the model trained on an ANN with one hidden layer, and the model trained on a CNN with multiple hidden layers.

Challenges of accelerating training

You will next look at some of the challenges that companies face in training DL models, beginning with the challenge of decreasing the time required for that training.

DL training, tensors, and GPUs

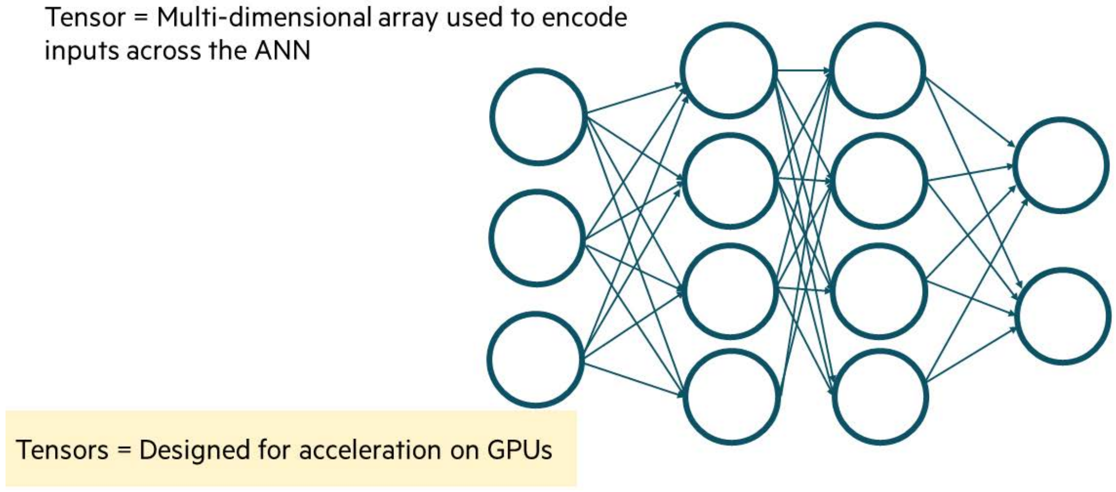

Tensor = Multi-dimensional array used to encode inputs across the ANN
Tensors = Designed for acceleration on GPUs

Figure 1-26: DL training, tensors, and GPUs

You should have some sense now of the complexity of the computations involved in training a DL model. The model consists of multiple nodes, each executing a complex computation with perhaps thousands or millions of parameters. And to become trained, the model must execute those computations, plus optimizers for adjusting the model, hundreds of thousands, or even millions, of times.

How can you speed up those computations? First you can run them on hardware that is optimized for them.

As you learned earlier, DL models typically use tensors, or multi-dimensional arrays, to encode inputs and outputs across the ANN. The hidden layer and output layer nodes perform mathematical operations on the tensor.

Tensors are specifically designed for fast processing on graphical processing units (GPUs). Thus companies find the best success by training DL models on GPUs.

The increasing computational demands of AI

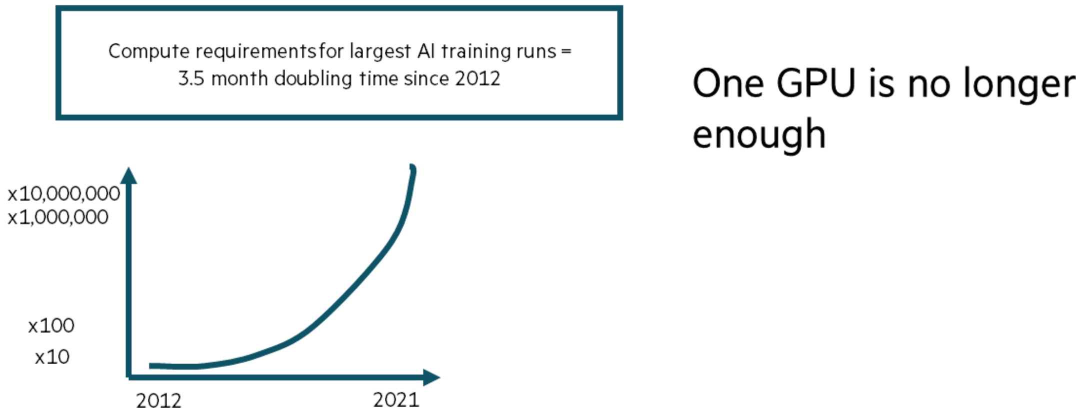


Figure 1-27: The increasing computational demands of AI

As DL models become more complex and powerful, their computational demands continue to increase. Research shows that compute used in the largest AI training runs has been increasing exponentially since 2012 with a 3.5 month doubling time (analysis by OpenAI, a non-profit AI research company, https://openai.com/blog/ai-and-compute/).

Digital Catapult further found that an approximate minimum computation requirement for training an ANN on a data set of 1.28 million images would be an exaflop. In short, one GPU is often no longer enough. With one GPU, training that ANN on 1.28 million images could take days or even months. (https://assets.ctfassets.net/nubxhjiwc091/6qDT7u9pzUsq8uukwCAayw/acc7e59350faa88fc504fc990c17deb7/MIG_MachinesforMachineIntelligence_Report_DigitalCatapult-1.pdf).

AI and HPC convergence

- To further accelerate training, companies can:
  - Use a cluster of servers that each contribute GPUs
  - Distribute computations over the cluster
- HPC clusters = "Ready-made" environment for exascale computing
- But distributing DL training comes with unique challenges

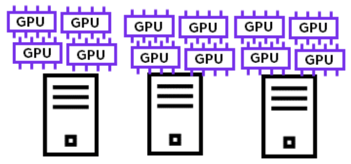


Figure 1-28: AI and HPC convergence

To further accelerate training, companies can distribute training across multiple GPUs. Because there is a limit on the number of GPUs one server can feasibly support, the company can further distribute training across a cluster of servers, each of which contributes GPUs.

As companies seek an environment with a cluster of powerful servers, they naturally start to consider high performance computing (HPC) clusters. An HPC cluster is already designed to run computational workloads that reach into exaflops. The environment also often already includes software for managing the cluster and scheduling workloads on it.

However, the HPC scheduling methods were designed for HPC workloads, and DL training workloads have their own particular requirements. Correctly distributing training across multiple GPUs, and possibly even multiple servers, requires taking into account what pieces of the training can be parallelized (executed on different GPUs at the same time) and what pieces need to be synchronized. It further requires establishing processes for that synchronization.

Data parallelization

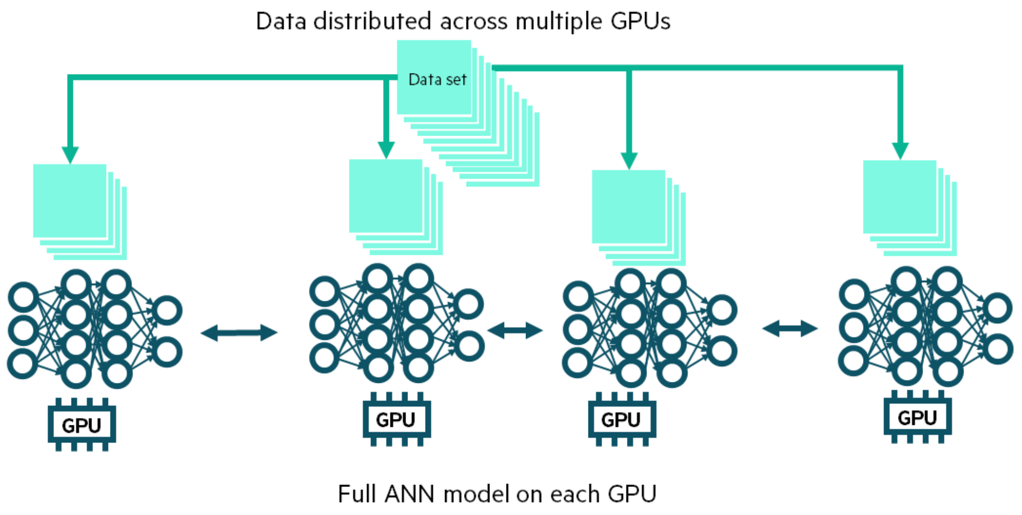

Data distributed across multiple GPUs
Data set
GPU
GPU
GPU
GPU
Full ANN model on each GPU

Figure 1-29: Data parallelization

Companies have several choices for distributing training across multiple GPUs.

Data parallelization provides the most common, and often the most powerful, approach to distributing DL training. The data set is divided into multiple pieces and distributed across the GPUs. Each GPU has the full ANN model and trains the model on its portion of the data. (More precisely, a process on the machine uses the GPU to train the model. For simplicity, though, this discussion will treat the GPU as the actor.)

Keep in mind, though, that the training process involves adjusting the model after each backward pass. You cannot let each GPU train the model on its full portion of the data without checking in with the other GPUs; otherwise the models would get out of sync.

Data parallelization process

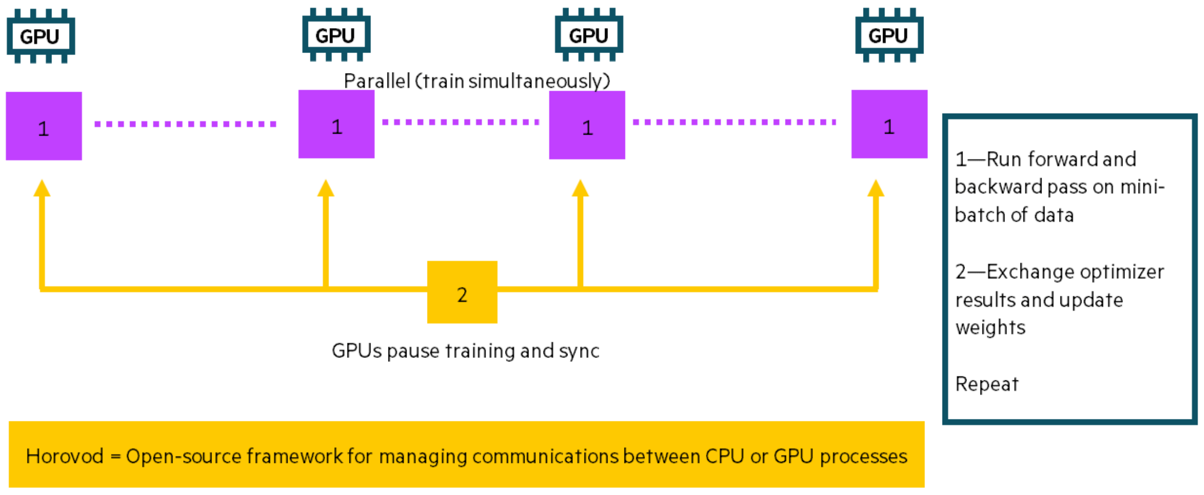

GPU
GPU
GPU
GPU
Parallel (train simultaneously)
1
1
1
1
2
GPUs pause training and sync
1—Run forward and backward pass on mini-batch of data
2—Exchange optimizer results and update weights
Repeat
Horovod = Open-source framework for managing communications between CPU or GPU processes

Figure 1-30: Data parallelization process

Overview of the process

Here you see a high level overview of how data parallelization keeps the model in sync across the GPUs.

Each GPU begins by training the model on a mini-batch of data. For example, it might train the model on 32 or 64 records. The GPU runs a forward pass, and it runs a backward pass to start figuring out how to optimize the model weights for better performance next time.

At this point, the GPUs must pause and exchange updates. They might communicate directly or through a central node. In any case, the results from all GPUs are combined, and new weights for the model are calculated. All GPUs then have the same updated weights, keeping the model in sync.

This process repeats over and over again until the model is fully trained. While the process adds some communications overhead, the most compute-intensive portions of the training are highly parallelized. ML engineers can train the DL model on a large amount of data much more quickly than they could with just one GPU.

Horovod

The training processes require a mechanism for communicating with each other, whether they are running on the same machine or on different machines.

Horovod provides a framework for distributing DL training for TensorFlow, Keras, PyTorch, or Apache MXNet. To enable communications between CPU processes, it uses MPI, a library long used for process-to-process communications in HPC clusters. Horovod made it easier for non-MPI experts to establish communications specifically for distributed DL training. (Horovod can also use other libraries.)

To reduce latency when training runs on GPUs, Horovod uses NCCL. NCCL supports GPUDirect—direct communications between GPU processes.

Horovod defines several ways for structuring the exchange of updates, including allreduce, allgather, and broadcast. You will learn more about allreduce, which is the mechanism used by HPE Cray AI Development Environment, in a later module.

Model parallelization

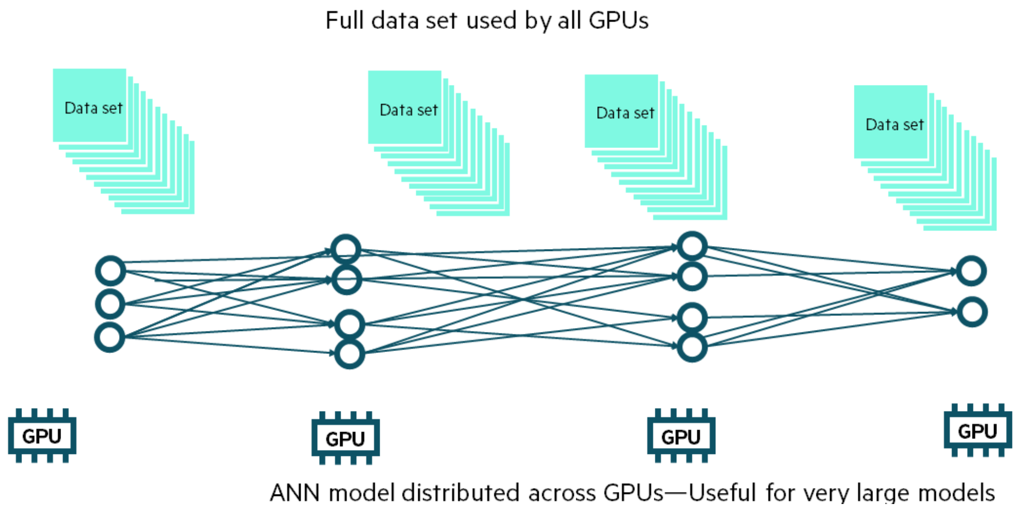

Full data set used by all GPUs
Data set
Data set
Data set
Data set
GPU
GPU
GPU
GPU
ANN model distributed across GPUs—Useful for very large models

Figure 1-31: Model parallelization

Model parallelization is another approach for distributing the training across multiple GPUs. It distributes pieces of the ANN model across multiple GPUs, making it ideal for very large models. Usually model parallelization puts different model layers on different GPUs. But sometimes it might divide up nodes in a single layer across multiple GPUs. Each GPU will eventually train its portion of the model on the full data set.

Model parallelization process

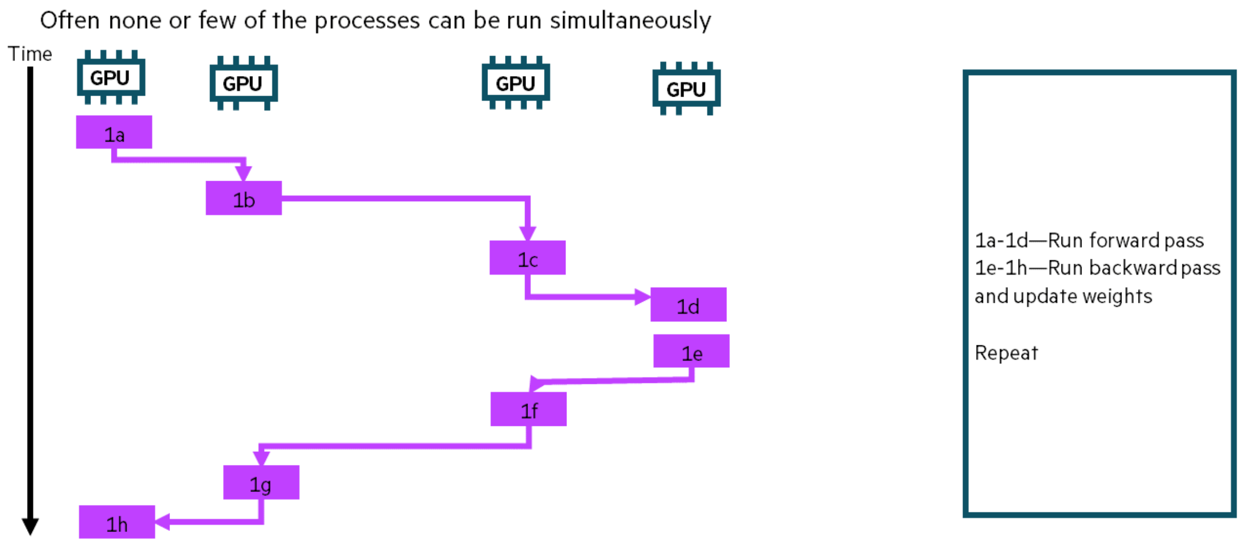

Often none or few of the processes can be run simultaneously
Time
GPU
GPU
GPU
GPU
1a
1b
1c
1d
1e
1f
1g
1h
1a-1d—Run forward pass
1e-1h—Run backward pass and update weights
Repeat

Figure 1-32: Model parallelization process

Despite its name, model parallelization often does not permit a high degree of parallelization. Data, after all, must pass through the ANN in a particular sequence. The GPU with hidden layer 1, for example, must pass its outputs to the GPU with hidden layer 2, before that second GPU can start to process data. Some parallelization might be possible, depending on how the model is split up, but much less so than data parallelization.

The same holds true for the backward pass. The GPU with the output layer must run the backward pass before the GPU with the deepest hidden layer can start, and so on.

Because model parallelization features less parallelization than data parallelization, it is less commonly used. However, it is useful for very large models that cannot fit on a single GPU. Speech recognition models, for example, can have millions of parameters and might require model parallelization.

Pipeline parallelization

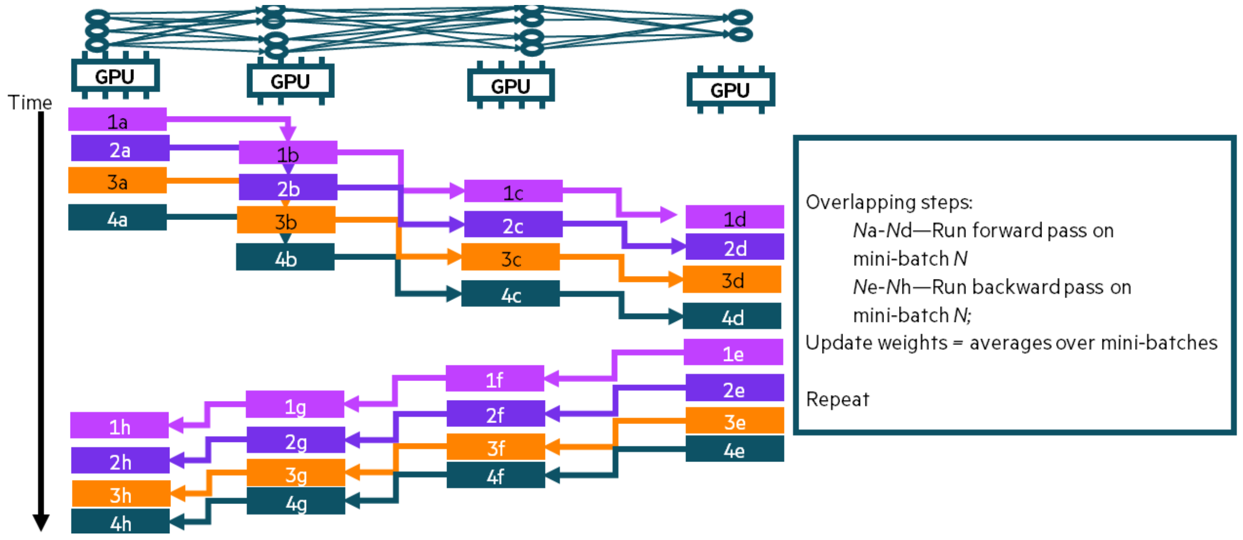

GPU
GPU
GPU
GPU
Time
1a
2a
3a
4a
1b
2b
3b
4b
1c
2c
3c
4c
1d
2d
3d
4d
1e
2e
3e
4e
1f
2f
3f
4f
1g
2g
3g
4g
1h
2h
3h
4h
Overlapping steps:
Na-Nd—Run forward pass on mini-batch N
Ne-Nh—Run backward pass on mini-batch N;
Update weights = averages over mini-batches
Repeat

Figure 1-33: Pipeline parallelization

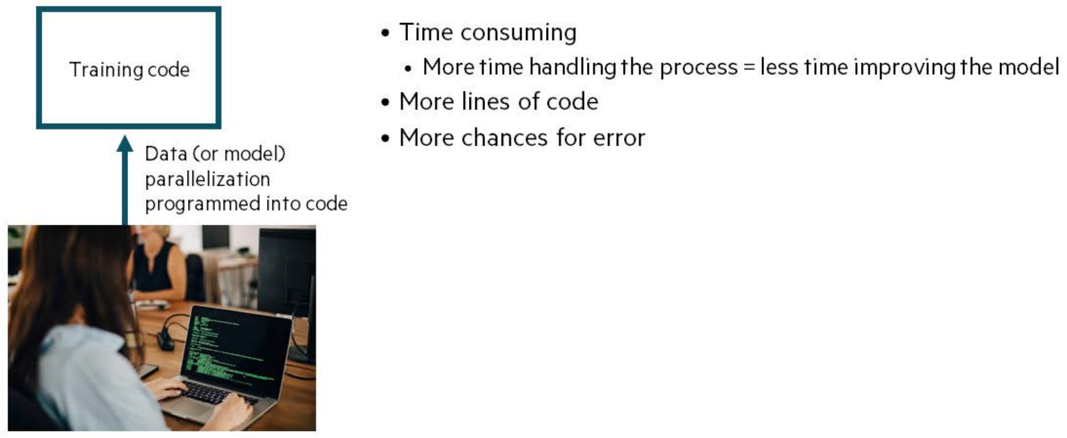
Pipeline parallelization helps to increase the efficiency and training speeds for very large models. Like model parallelization, it divides the model across multiple GPUs. However, rather than have GPUs process a single batch of data at a time—leading to long periods in which some GPUs sit idle—it has GPUs process multiple mini-batches at once.

A pipeline schedules the workloads for each GPU, keeping track of which operations can proceed in parallel. For example, a model might have four layers, each of which is held on a different GPU. GPU 1 can pass its outputs for mini-batch 1 to GPU 2. Then GPU 1 can start mini-batch 2 while GPU 2 processes mini-batch 1. Similarly, GPU 1 can start on mini-batch 3 while GPU 2 processes mini-batch 2 and GPU 3 begins on mini-batch 1. Thus multiple GPUs can run computations on different mini-batches in parallel. Similarly backward pass computations can be partially parallelized.

After a GPU has completed its backward pass for a given number of mini-batches, it combines what it has learned across all of those backward passes. It then updates the model weights for its layer.

When all GPUs have updated their weights, the process then repeats again and again until the model is trained.

Challenges of parallelization for ML engineers



- Time consuming
  - More time handling the process = less time improving the model
- More lines of code
- More chances for error

Figure 1-34: Challenges of parallelization for ML engineers

ML engineers are often forced to script the training parallelization process directly within the training code. That takes time and effort. The engineers find themselves spending more time on scripting the supporting processes than on improving the models themselves. The code can become long and unwieldy, difficult to understand and troubleshoot.

ML engineers need solutions that handle the training process—even a complex, distributed process over multiple GPUs and machines—so that they can focus on their main job.

A deeper look at DL training optimization

In addition to accelerating the DL training process, companies also need to optimize that process. They need to ensure that their process promotes the best final results within an attainable period of time.

How to build a better training

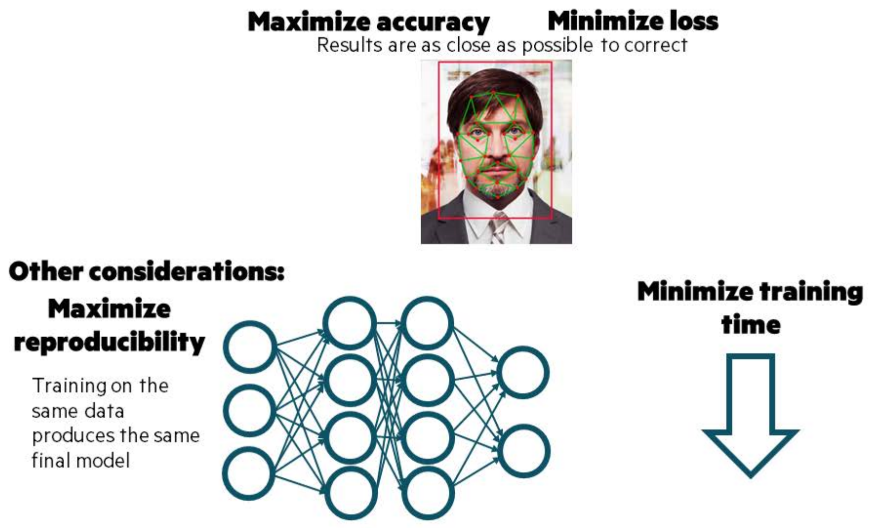

Maximize accuracy
Minimize loss
Results are as close as possible to correct
Other considerations:
Maximize reproducibility
Training on the same data produces the same final model
Minimize training time

Figure 1-35: How to build a better training

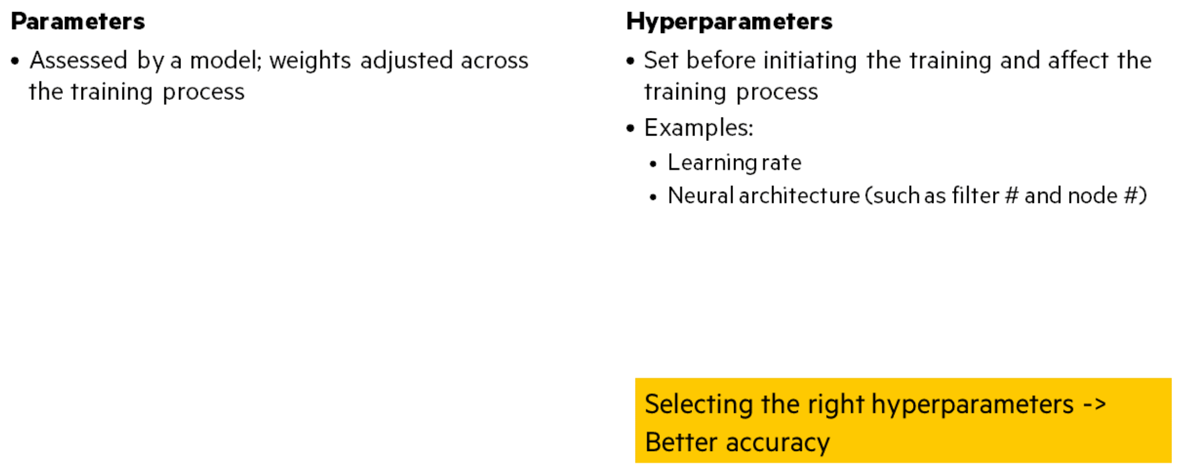
Fundamentally, all ML engineers have a similar end goal. They want to produce a trained model that outputs results as close to correct as possible. They might think of this as maximizing accuracy or, conversely, as minimizing loss. (Accuracy and loss measure slightly different things. Think, for example, of an image classification ANN. The ANN's accuracy refers to the percentage of correct classifications that it makes. Its loss refers to the distance between the predicted outputs and the actual correct outputs. But, in general, the lower the loss, the higher the accuracy.)

As discussed earlier, here ML engineers are specifically considering the *validation* accuracy and loss as opposed to those metrics calculated during the training. They want a model that performs well on any new data from the real world.

The ML engineers might keep some additional considerations in mind too. Often, they want to maximize reproducibility. In other words, they want consistency in their model. If they train an ANN on the same data twice in the same way, the two final models should match.

And, as discussed in the previous section, companies want to achieve the best results within a reasonable amount of time. They cannot dedicate months to a single training process.

The importance of hyperparameters



Parameters

- Assessed by a model; weights adjusted across the training process

Hyperparameters

- Set before initiating the training and affect the training process
- Examples:
  - Learning rate
  - Neural architecture (such as filter # and node #)

Selecting the right hyperparameters -> Better accuracy

Figure 1-36: The importance of hyperparameters

You have learned about how models have parameters. In a simple ML model for predicting housing price, house area might be a parameter. DL complicates matters because it consists of multiple interconnecting algorithms. For example, an ANN might have 24 nodes in the input layer, to pull in 24 record parameters, and a bias node. All these nodes connect to every node in the first hidden layer, which might have 100 nodes. The first hidden layer has 2500 parameters because it receives 2500 inputs ((24+1)*100 = 2500) to which it applies weights. The next hidden layer introduces more parameters. But whether you are considering a simple ML model or a complex DL model, the training process is all about adjusting the parameter weights.

But the training process also involves “hyperparameters,” which are set before the process begins and determine how the process itself executes. One hyperparameter is learning rate. The learning rate defines how fast weights can change after one backward pass. A fast learning rate lets the model progress towards better results more quickly, but also increases the chances of the model leaping off in the wrong direction. A slow learning rate is more conservative, but might mean that the model does not reach high enough accuracy before the training ends. You might define the same initial model and train it on the same data, but with two different learning rates; the result would be two different trained models. By achieving the right balance for the learning rate, you can obtain a model that performs better in the end.

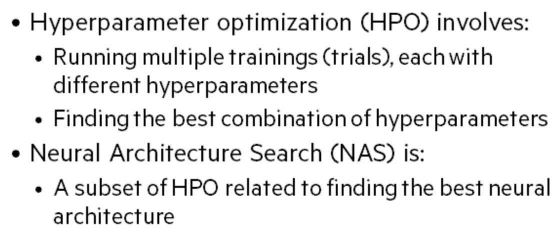
Other examples of hyperparameters include:

- **Neural architecture**—Neural architecture hyperparameters include the number of nodes in each layer or the number of filters in a convolutional layer.
- **Optimizer algorithm**—Different algorithms exist for calculating the weights for the next pass, and each has its advantages and disadvantages.
- **Learning rate decay**—Some optimizers allow the learning rate to change as the training progresses. The learning rate decay defines how quickly the learning rate changes.
- **Node dropout**—The training process can randomly select nodes to ignore for one pass, which helps the model as a whole perform better in general, rather than coming to over-rely on certain nodes. This hyperparameter controls the probability that a node’s output is retained or ignored for that pass.

The algorithms used by the DL model might also have their own hyperparameters. For example, a decision tree algorithm can use trees with different numbers of splits; the number of splits is a hyperparameter.

The key point for you to understand is this: all of these hyperparameters together affect the ultimate accuracy of the trained model. By choosing the best combination of hyperparameters ML engineers can obtain better end results.

HPO and NAS



- Hyperparameter optimization (HPO) involves:
  - Running multiple trainings (trials), each with different hyperparameters
  - Finding the best combination of hyperparameters
- Neural Architecture Search (NAS) is:
  - A subset of HPO related to finding the best neural architecture

Figure 1-37: HPO and NAS

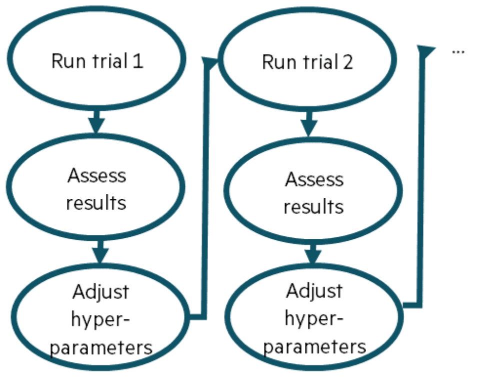
However, choosing the optimal hyperparameters is not a trivial task, and what works well for one model might not work well for another. Hyperparameter optimization (HPO) involves finding the right hyperparameters for your particular model. You run multiple instances of the training process, each with a different set of hyperparameters. The instance of the training process is often called a “trial,” while the set of hyperparameters is called a “config.” By performing multiple trials, you can find the trial and config that create the best model.

Neural Architecture Search (NAS) is a subset of HPO; it refers to finding the best hyperparameters related to neural architecture.

Manual versus automatic HPO

Manual

- Requires ongoing monitoring and tuning
- Can be time-consuming
- Requires a great deal of expertise



Automatic

- Fast, scalable way to optimize
- Uses a "searcher" method
- Might allow running multiple trials in parallel
- Requires less ongoing monitoring and fine-tuning from experts

Figure 1-38: Manual versus automatic HPO

ML engineers can perform manual HPO. They can run one trial and assess the results. They can then adjust the hyperparameters, run a second trial, and assess the results again. Based on the differences between the results, they can try to figure out a better way to adjust the hyperparameters yet again. They can then run a third trial, and a fourth, and so on. But this process is tedious, even if ML engineers have pipelines to help manage the process. The engineers also require a great deal of expertise in interpreting results and selecting hyperparameters for the next trial.

Automatic HPO provides a fast, scalable way to optimize. Automatic HPO relies on a searcher method—you will explore some examples in a moment—to run multiple trials. The searcher might run some of these trials in parallel, while also intelligently using the results from ongoing trials to guide the process.

The end result: ML engineers can optimize hyperparameters, and obtain a better final model, with less ongoing monitoring and fine-tuning.

High level view of automatic HPO

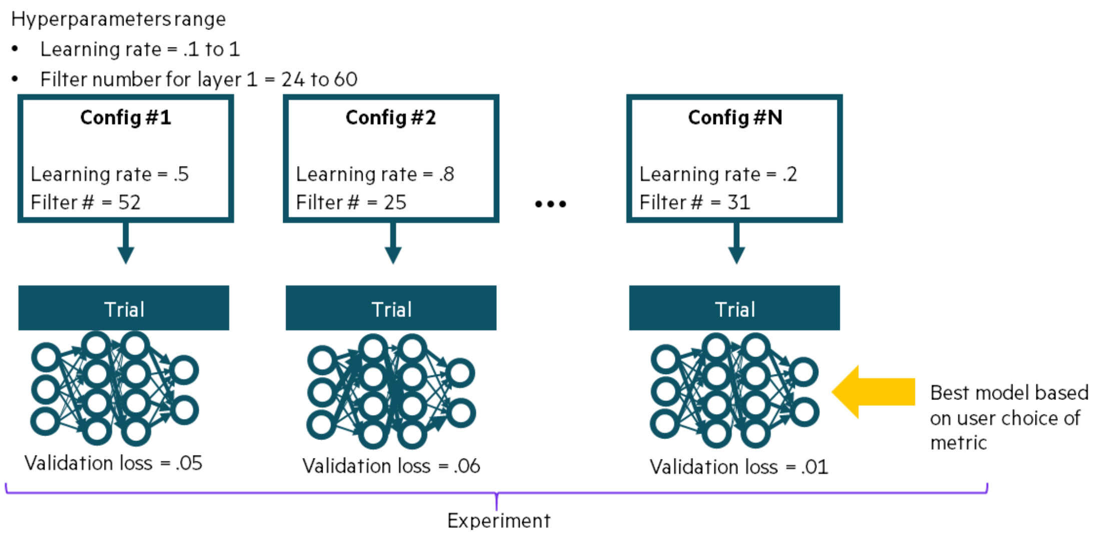


Figure 1-39: High level view of automatic HPO

Each HPO searcher has its own particular way of operating, but at a high-level they involve some common processes.

The searcher selects a number of configs (sets of values for hyperparameters) from a range specified by the ML engineers. For example, ML engineers might specify that the learning rate can be between .1 and 1, while the number of filters for convolutional hidden layer 1 can be between 24 and 60. From these ranges, the searcher creates configs, which use different combinations of values for the hyperparameters. The figure shows three example configs. A true HPO search would use many more.

The search is called on experiment. For each config, the searcher starts a trial. Each trial proceeds like any DL model training. The model processes data and adjusts itself to minimize loss.

Depending on the searcher type, the searcher might simply let the trials proceed. Or it might periodically pause the trials and have them process validation data to show how they are doing. Based on the results, the searcher might stop poorly performing trials and possibly create new trials. (Throughout this discussion, “poor performance” refers to worse validation metrics, such as high validation loss or low validation accuracy. “Good performance” refers to better validation metrics.)

In the end, the searcher, and ML engineers, can select the model with the best validation performance.

Grid and random searcher methods

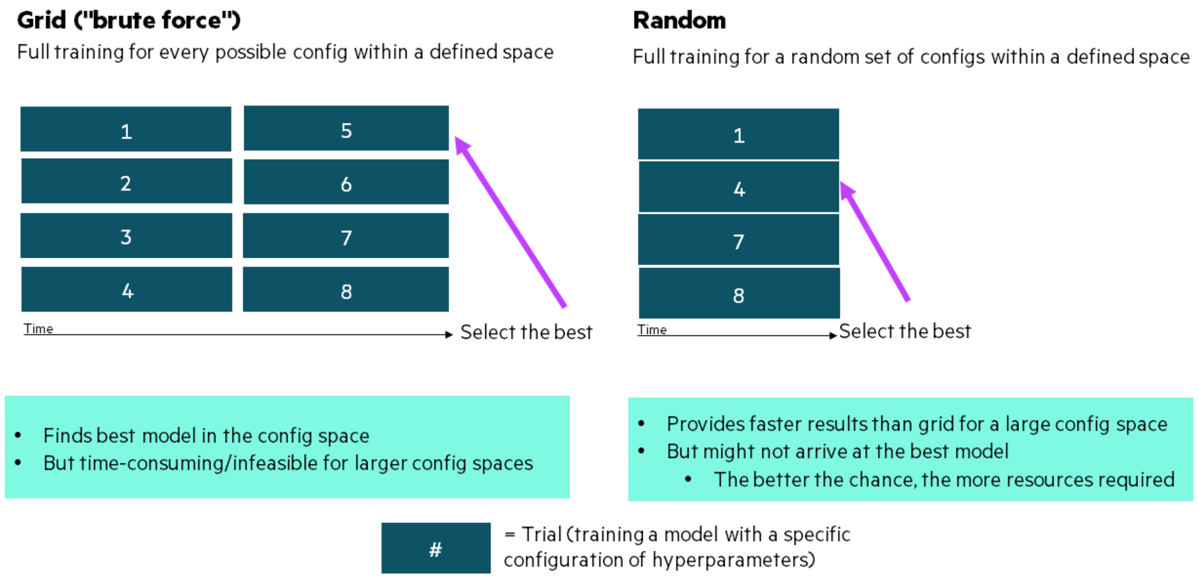

Grid ("brute force")
Full training for every possible config within a defined space
1
2
3
4
5
6
7
8
Time
Select the best
Random
Full training for a random set of configs within a defined space
1
4
7
8
Time
Select the best
Finds best model in the config space
But time-consuming/infeasible for larger config spaces
Provides faster results than grid for a large config space
But might not arrive at the best model
The better the chance, the more resources required
#
= Trial (training a model with a specific configuration of hyperparameters)

Figure 1-40: Grid and random searcher methods

You will now examine particular searcher methods in more detail, learning some of the advantages and disadvantages of each.

Grid

You can think of a grid search as a “brute force” search. The searcher tries out every combination of hyperparameters within the “config space” defined by ML engineers.

A config space defines all the possible combinations for hyperparameters. For example, the ML engineers might specify four options for the learning rate. And they might specify two options for node dropout. This simple config space has eight possible configs with different combinations of hyperparameters.

The grid search creates one trial for each possible config, and it runs that trial to completion. In other words, each trial trains its model on the full number of records desired by ML engineers.

As one advantage, a grid searcher ensures you find the best config within the defined space. And because the grid searcher simply starts the trials and runs them to completion, how one trial runs does not affect how another trial runs. The trials can run in parallel over as many GPUs as you have available. In this example, the company has four GPUs, so the grid searcher assigns one trial to each GPU and runs those trials simultaneously. As those trials end, it can start the remaining trials on the available GPUs.

However, while grid searchers guarantee you find the best config within your defined space, that space itself, in practice, cannot be large. What if you have four different hyperparameters that you want to explore? And what if each of those hyperparameters could have one of eight values? With thousands of possible combinations, it becomes infeasible to test them all. And in many cases, the config space could be much larger than that.

Random

A random searcher selects a certain number of configs randomly from the defined config space. It then creates one trial for each of those configs and lets all trials run to completion. This searcher allows the ML engineers to define a larger config space, but cap the number of trials to a reasonable number. For example, with a config space that has 100 possible configs, they might choose to try just 32 random configs (or for the highly simplified example shown in the figure, they might choose to try just four out of eight possible configs).

Like grid searchers, random searchers can run multiple trials at the same time, as permitted by the resources available. If you have four GPUs available, for example, you might run four trials at a time. But again, since the searcher will run all trials to completion on a large amount of data, you will probably want to cap the total number of trials at a relatively low number. As the ratio between number of trials and number of possible configs decreases, the chances of finding the best, or close to best, final model also decrease.

Population-Based Training (PBT)

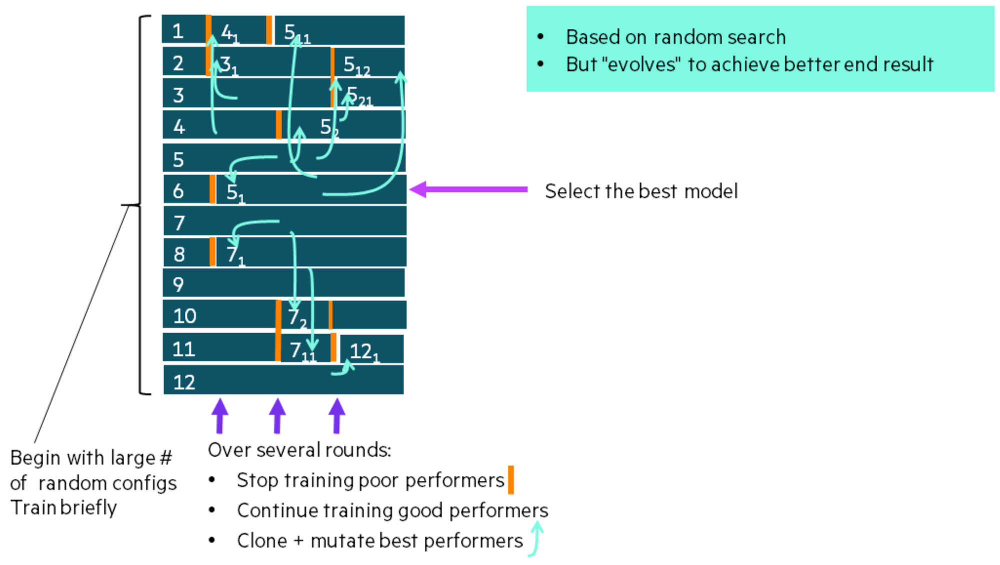

1
2
3
4
5
6
7
8
9
10
11
12
4₁
5₁₁
3₁
5₁₂
5₂₁
5₂
5₁
7₁
7₂
7₁₁
12₁
Based on random search
But "evolves" to achieve better end result
Select the best model
Begin with large # of random configs Train briefly
Over several rounds:
Stop training poor performers
Continue training good performers
Clone + mutate best performers

Figure 1-41: Population-Based Training (PBT)

Next you will look at some searcher methods that permit experimenting with many different trials in a reasonable amount of time. The searchers achieve this goal by monitoring trials on an ongoing basis and stopping ones that fail to perform well, opening up more compute resources for other trials.

First you will examine the Population-Based Training (PBT) searcher.

PBT starts much like a random searcher. It selects a specific number of random configs from the config space; this is the population size. In this example, the population size is 12, so PBT selects 12 configs and starts 12 trials. The company also has 12 GPUs available in this example, so the trials can run at the same time. (For a real HPO search, though, the population size will often be larger.)

Unlike a random searcher, though, PBT does not necessarily let every trial train the model on the full data set. Instead it implements a series of rounds. In each round, trials train their model on a smaller portion of data. At the end of a round, PBT validates the trials and determines which are performing worst and which are performing best. PBT stops the worst performers—no need to continue tying up GPU resources for those trials since they are unlikely to achieve the best results even after full training. You can choose the fraction of trials to stop. In this example, that fraction is one-third.

For the next round, PBT will allow other trials, which are performing well, to continue training their models. And because resources are available for additional trials, PBT fills out the population with several new trials with new configs. PBT creates those configs intelligently by “mutating” the configs of the best performers (the best third in this example). In other words, it uses the same hyperparameters with a few adjustments. In the example shown in the figure, trials 3, 4, 5, and 7 are performing best after the first round, so PBT creates new trials with similar configs.

PBT continues this process over several rounds. At the end of each round, it follows the same process. It stops the trials that are performing worst. And it replaces those trials with new ones, based on mutating the best performers.

By the end of the process, you have tried more configs than you could have with a random searcher in the same time period. And you have devoted more compute resources to the best trials, hopefully achieving a better final result.

Note that this example shows a population size that exactly matches the resources available; all trials run at once. However, in the real world, the population size could be larger than the number of trials that can run at once on the company’s resources. Then PBT would make trials take turns. It would run some trials on the amount of data for the first round; it would then pause those trials while other trials train their models on the required amount of data. After all trials in the current population have completed the round, PBT would assess the results and continue to the next round; in each round, trials would continue to take turns using the resources available.

Successive Halving Algorithm (SHA) and Asynchronous SHA (ASHA)

- Test many configs to increase the chances of finding better ones
- But reduce the resource requirements with early stopping
  - Train most trials on a small amount of data
  - Train best performers on more and more data

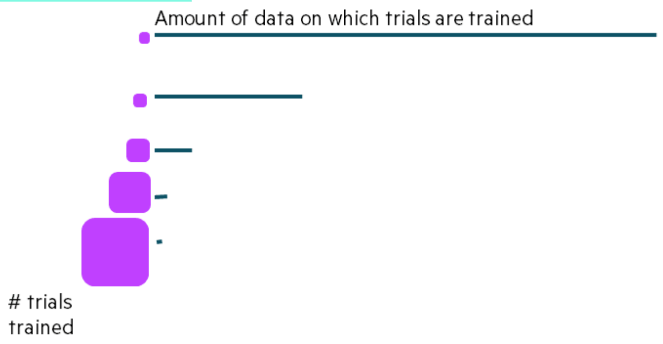


Figure 1-42: Successive Halving Algorithm (SHA) and Asynchronous SHA (ASHA)

SHA and ASHA are two other searcher methods that attempt to test many configs in order to increase the chances of finding one of the best in a large config space. However, to make that search feasible, SHA and ASHA devote the most compute resources to training only the best performing trials. They also use the principle that, if a trial does not perform well at first (in other words, exhibits poor validation metrics), it is unlikely to result in the best model even after extensive training.

At a high level, both SHA and ASHA work as follows:

- They create a very large number of trials with configs selected randomly from a defined config space.
- They run most of those trials on quite a small amount of data.
- They “promote” the trials that have exhibited the best performance. They allow the promoted trials to train their models on more data.
- They continue to select and promote the best performers until just a few trials, which exhibit the absolute best performance, have trained their models on the complete data set.

A closer look at SHA

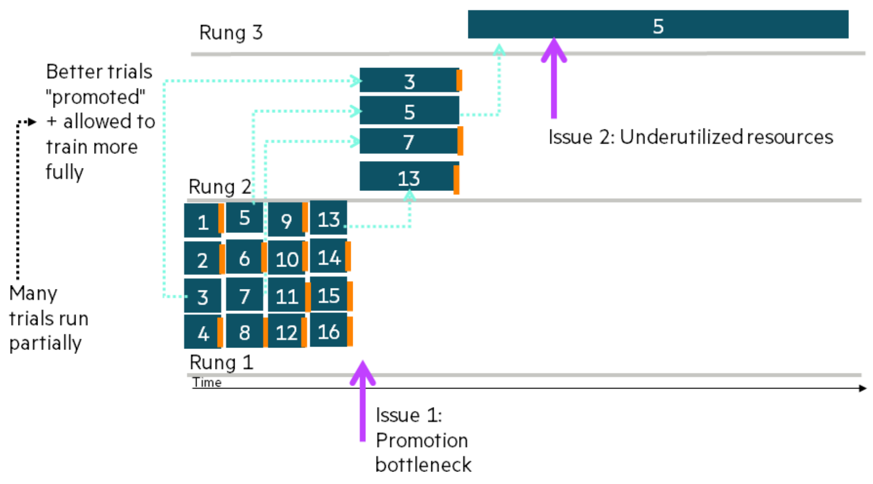

Rung 3
5
Better trials "promoted" + allowed to train more fully
3
5
7
13
Issue 2: Underutilized resources
Rung 2
1
5
9
13
2
6
10
14
3
7
11
15
4
8
12
16
Many trials run partially
Rung 1
Time
Issue 1: Promotion bottleneck

Figure 1-43: A closer look at SHA

You will now look at how SHA works in a bit more detail.

SHA establishes multiple rungs. This simple example shows three rungs; however, often SHA uses four or five rungs. At each successive rung, a smaller number of trials train models on a larger amount of data.

SHA begins by creating the maximum number of trials desired by the ML engineers defining the process. It randomly selects a different hyperparameter config for each trial from the defined config space. The trials all begin at rung 1. At this rung, each trial trains its model on just a small amount of data. For example, if scientists want to eventually train the best model on 160,000 records, the trials might be trained on just 10,000 records—or even fewer.

In the example shown here, SHA starts 16 trials; in the real world, the number would more likely be in the hundreds. This example further shows how the company has four GPU slots available, so four trials can run at once.

After each trial has trained its model on this small amount of data, SHA validates all of the trials' models and chooses to promote the best ones—in other words, the ones with the lowest validation loss or highest validation accuracy. The fraction of trials promoted depends on SHA's divisor. This example shows a divisor of four. Out of 16 trials, SHA permits only four trials to continue.

SHA promotes those trials to rung 2. The trials train their models on more data. The models should now be improving quite a bit, showing how well they really can perform. After the trials have completed processing the desired amount of data, SHA once again pauses and validates them. Again it chooses the ones with the best accuracy. Because the divisor in this example is four, and only four trials remain, SHA choose the best single trial.

SHA promotes that trial and permits it to train its model on the rest of the data set.

By aggressively stopping poorly performing trials, SHA enables ML engineers to explore a large config space in a relatively short amount of time. SHA homes in on the best config and uses that config to train a model on the full data set.

However, SHA does exhibit some inefficiencies. First, note how bottlenecks can occur at promotion time. In this example, the company has four GPUs and can run four trials at a time. However, SHA is trying to assess many more trials than that. SHA must wait until all trials have a chance to run before it can assess good performers and promote them. Second, SHA tends to underutilize resources as it progresses to the higher rungs. In this example, by the time SHA has selected the best trial in rung 3, it is using only one of the four GPUs available. And suppose that the company had actually had 16 GPUs available for the beginning of the process. As trials are weeded out and the number running reduces, the company would need fewer and fewer GPUs, leaving the others idle.

A closer look at ASHA

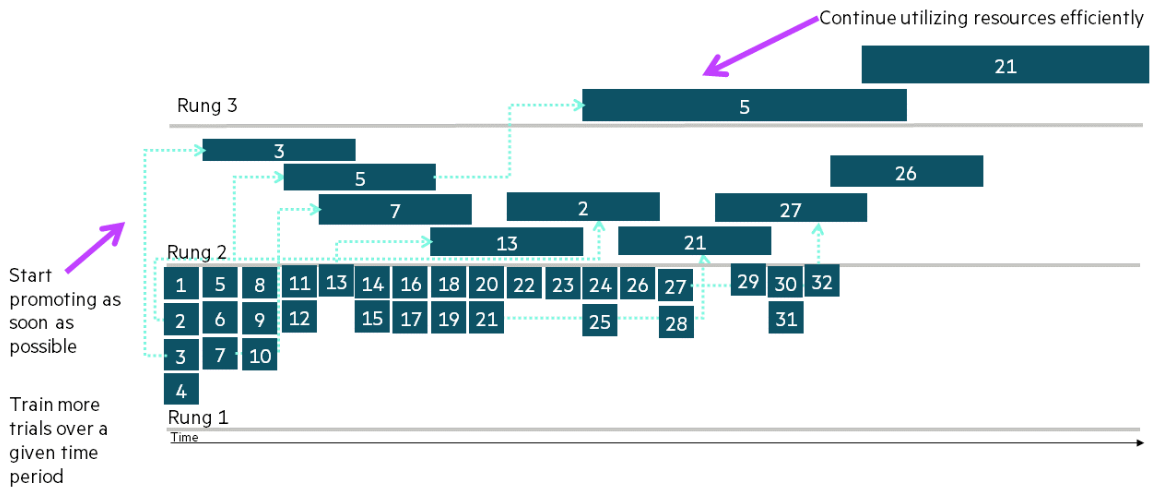


Figure 1-44: A closer look at ASHA

ASHA operates similarly to SHA, but addresses those inefficiencies.

As you see, like SHA, ASHA:

- Creates many trials with different hyperparameter configs, randomly selected from the config space
- Initially trains each trial's model on just a small amount of data
- Promotes a fraction of the best performing trials to the next rung, where the trials train their models on more data

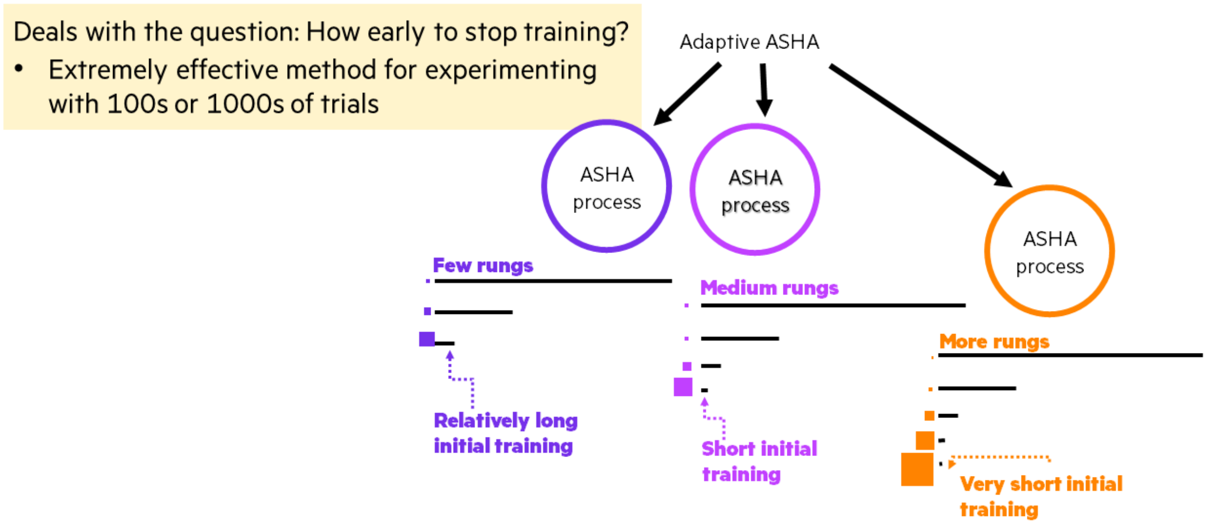
However, ASHA does not wait until all trials have been trained on the minimum amount of data to start promoting. Instead, it starts promoting as soon as it can. In this example, the divisor is four. So as soon as four trials have been trained on the rung 1 amount of data and validated, ASHA can select the best one-fourth (one trial) and promote that trial. That good performer can continue training its model without waiting for many other trials to start and run. At the same time ASHA is also starting more trials at rung 1 and letting those trials show what they can do on a small amount of data. That is why ASHA is asynchronous; it lets trials run on multiple rungs at the same time.

By starting trials 5, 6, 7, and then 8, ASHA broadens the base. Now it can select another best performer to meet the 1/4 divisor rule—in this example, trial 5.

ASHA continues in this way, promoting best performers as soon as it can, while also expanding the number of total trials at lower rungs. That expanded base, in turn, means that ASHA can promote more trials.

The end result is that ASHA continues to utilize resources efficiently throughout its complete course. It can be using GPUs for just a couple trials in the top rung while using other GPUs to continue training trials at bottom and middle rungs. In less than twice the time it would take than SHA to run 16 trials, ASHA can run 32.

Adaptive ASHA


Deals with the question: How early to stop training?

- Extremely effective method for experimenting with 100s or 1000s of trials

Figure 1-45: Adaptive ASHA

ASHA works very well for HPO, but even it requires tuning. ML engineers need to consider questions such as how early to stop trials at rung 1. If they stop the trials too early, they might not give potentially good performers a chance to show what they can do. But if they let trials run too long initially, they lose some of the advantages of using ASHA. Making these decisions can be difficult, and ML engineers are not always sure that they have defined the best processes.

HPE Cray AI Development Environment introduces Adaptive ASHA, which works extremely well for experimenting with hundreds or even thousands of trials. Adaptive ASHA creates multiple ASHA processes, each with its own settings. For example, it might create three processes. One process provides relatively long initial training and uses just three rungs. Another process stops initial training very quickly and drops out trials aggressively over five rungs. A third process uses settings in between these two extremes. In this way, ML engineers can balance the benefits of various approaches, explore a very large config space, and arrive at a highly accurate final model.

The challenges of collaboration

Companies that are serious about ML/DL require multi-member teams working on projects. They need tools that help to promote seamless collaboration, which in turn leads to consistent and reproducible results.

Common collaboration challenges

- Sharing data and code across fragmented environments
- Obtaining visibility into training and reasons for model differences
- Reproducing results
- Sharing compute resources

Figure 1-46: Common collaboration challenges

To work together effectively, an ML team needs a shared repository for model code, including trained models. If multiple members are working on building and training models for the same application, they need to know that they are working with the right version of code. And they need to be able to find each other's trained models to compare and evaluate them. Because data is so important for model training and validation, shared storage for data sets is also critical. ML engineers need to know that they are working with consistent data sets, even as new data is extracted from other sources and brought into the ML/DL ecosystem. A lack of consistent data can compromise scientists' efforts to refine and improve models. Without data consistency, the team cannot be sure if changes to model code or to an HPO search are what produced different results, or if the differences derive from the training data.

The team members also need visibility into each other's work and results. They need easy-to-see metrics that show which models are truly performing most accurately. And to have confidence in their trained models, different members of the team need to be able to reproduce each other's results. In fact, the company might build reproducibility requirements into the process, stopping the project from moving on until multiple members can reproduce the model. However, reproducing a model can be surprisingly difficult even with totally consistent data. Think about all the factors that can affect the final model from the selected hyperparameters to the random weights with which the model was initialized. Without processes and tools that help team members track these variables, they can struggle for a long time to reproduce each other's models.

Many companies start out in an ad hoc way with each team member building and training models on their own machines. However, this approach creates multiple pitfalls as companies scale their efforts. Using a shared training environment can help the team members work together. It can also furnish the high levels of GPU resources needed for accelerated, distributed training and automatic HPO.

As companies move to a shared compute environment, however, they need ways to handle the sharing. Suppose that the shared environment provides 16 GPUs. ML engineers need applications that help them figure out which of those GPUs are currently available for running their training workloads. If no GPUs are available, the application should help them schedule the workload and then later execute it without the engineer having to remember to come back and restart the process. Ideally the application should also help team members prioritize the right training, as well as save and preserve models if a training process needs to pause temporarily. ML engineers naturally do not want to take time away from their main job to handle these processes. Instead they look for software that can handle these issues for them.

Module 1—Lab 2

You will once again access the HPE Cray AI Development Environment cluster already deployed in your lab environment. You will view a completed “experiment” (a training process with multiple trials) that used the Adaptive ASHA HPO mechanism.

Module 1—Lab 2 debrief

Discuss these questions with your classmates:

- The HPE Cray AI Development Environment UI showed the record of an experiment that used Adaptive ASHA. Discuss what you observed, such as:
  - In the **Visualization** tab, why were there many dots and lines at the left of the graph and then fewer at the right?
  - Which tabs would help ML engineers consider the combined effect of two hyperparameters on validation accuracy?
- What validation accuracy did the best model produced by the experiment achieve?

Summary

You have learned enough about ML and DL to understand the many challenges companies face as they seek to accelerate the training of highly complex DL models. You also understand the critical role that HPO/NAS play in optimizing the training process to produce the best performing model. You also learned about some of the challenges that ML engineering teams face in collaborating.

In the next module, you will learn how HPE Cray AI Development solutions helps customers surmount these challenges.

Learning checks

1. What does the optimization step do during ML/DL training?
   a. Selects new hyperparameters for the next training pass
   b. Adjusts the model weights to help the model perform better
   c. Changes the number of layers in the neural architecture
   d. Tests the model performance on different data from training data
2. What correctly describes an artificial neural network (ANN)?
   a. It is a group of ML algorithms that were trained in different ways.
   b. It consists of multiple interconnected layers of algorithms.
   c. It is any ML algorithm that trains itself using a data set.
3. What is a common way to speed up training by distributing the process over multiple GPUs?
   a. Use data parallelization, in which different GPUs train the model with different data and then sync the weights.
   b. Use model parallelization, in which different GPUs train the model with different data and then compare the results.
   c. Use hardware parallelization, in which different GPUs perform different computational operations within a single training pass.
   d. Use lifecyle parallelization, in which different GPUs perform different pieces of the building, training, and validating portions of the lifecycle.
4. What is a difference between a random searcher for HPO and an ASHA searcher?
   a. The random searcher selects configs randomly, while the ASHA searcher selects configs based on configs that worked well in a previous training run.
   b. The random searcher selects configs randomly, while the ASHA searcher lets researchers weight certain options for more frequent selection.
   c. The random searcher only trains most trials partially, while the ASHA searcher trains all trials on the complete data set.
   d. The random searcher trains all trials on the complete data set, while the ASHA searcher trains just the most successful trials on the complete data set.

Appendix

This appendix provides a few more details on ML/DL concepts. You do not need to know these concepts to pass the exam associated with this course. However, if you are interested, understanding a bit more about these concepts might give you a chance to deepen your customer conversations.

Optimizer functions

- Calculate how to adjust weights to minimize loss
- Typically based on gradient descent algorithms

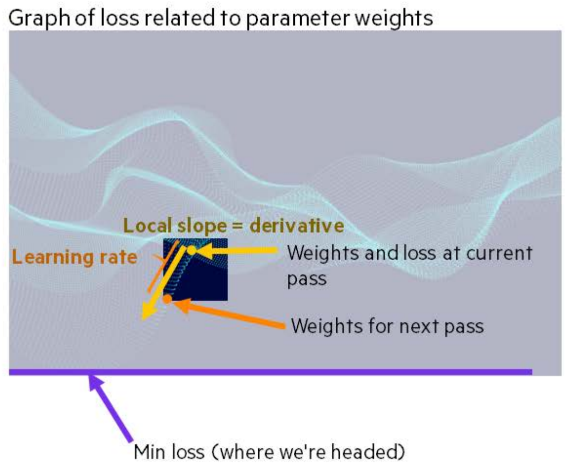

Graph of loss related to parameter weights
Local slope = derivative
Learning rate
Weights and loss at current pass
Weights for next pass
Min loss (where we're headed)

Figure 1-47: Optimizer functions

You have learned about “backward passes” in the training process for ANNs. During the backward pass, an optimizer function calculates how to adjust the weight for every parameter to make the model perform better on the next forward pass. This optimizer function is typically based on a gradient descent algorithm.

To understand how that algorithm works, take a moment to consider the goal of the optimizer algorithm conceptually.

In ML/DL, training loss measures the distance between actual results and accurate results. Your goal is zero loss—no distance between the two at all. You can graph the relationship between each parameter weight and the loss. If an ML/DL model featured just one parameter, that graph would be a simple line with weight on the x-axis and loss on the y-axis. That line charts how the loss goes up and down as the weight changes. This line might follow all sorts of shapes. Maybe it’s a simple slope, but more likely it’s a parabola, an “s” curve, or some other shape. In any case, your goal is to find where that line crosses zero on the y-axis—the weight where loss is zero.

Now think about a DL model that features two parameters. The graph resembles a 3D topological map such as that shown in the figure. The x-axis tracks the value for one parameter’s weight, the z-axis tracks the value for the other parameter’s weight, and the y-axis tracks the loss. You want to find the combination of weights that takes you to zero loss, the “bottom” of the topological map. As shown in the figure, the relationship between the weights and the loss can be quite complex. Increasing both weights might take you closer to the bottom for a while, then you might need to increase one weight while decreasing the other.

And, of course, DL models feature thousands or millions of parameters—making it hard to visualize the relationship between all those parameters’ current weights and the loss. Fortunately, you do not have to. The gradient descent algorithm concept remains the same for one, two, or many more parameters.

If you knew how the complete graph looks, of course, training the ML/DL model would not be necessary. You would already know the right weights. So the gradient descent algorithm helps you move toward the minimum point with a *limited* view of the graph. You can think of this algorithm as a person who is trying to get down a mountain in the dark with a flashlight. The person uses the limited pool of light from the flashlight to find where the ground slopes most steeply. The person then follows that slope down a little way, stops, reassesses, and repeats the process. At a high level, this algorithm performs the following at each backward pass:

- It looks at the current weights and loss and applies a derivative to find the local slope or “gradient.”
- It moves in the direction of the local slope to select new weights. The learning rate defines how “far” down the local slope the optimizer moves at each pass. You can edge down slowly with a small

learning rate. Or you can take a big step with a large learning rate. As you will explore on the next page, both small and large steps—slow and fast learning rates—come with risks.

Some common optimizer functions

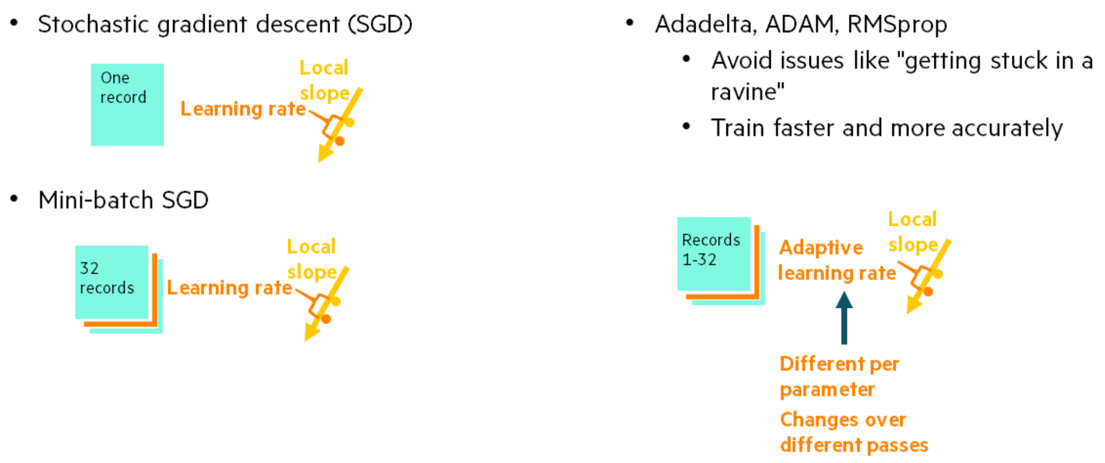

- Stochastic gradient descent (SGD)
- Mini-batch SGD
- Adadelta, ADAM, RMSprop
  - Avoid issues like "getting stuck in a ravine"
  - Train faster and more accurately

Figure 1-48: Some common optimizer functions

The stochastic gradient descent (SGD) function operates as described on the previous page. With SGD, the model completes a forward pass on just one record and then runs the optimizer. Mini-batch SGD is similar to SGD, but it sends a batch of records through the model. Then it assesses the loss for all those records and runs the optimizer.

Consider the complex topographical map that you can use to visualize the relationship between ANN parameter weights and loss. Often this map features multiple valleys and hills. A simple SGD or mini-batch SGD function could get “stuck” in a “ravine,” which is a local minimum for the loss that is nonetheless still above the real minimum. In other words, the optimizer follows the slope down to the bottom of the ravine. Its next adjustments to the model weights take it slightly “up” the ravine side. The optimizer then adjusts the weights to go back towards the bottom of the ravine. It moves the model back and forth over nearly the same weights. It is stuck. A faster learning rate could push the model out of the ravine and moving on again toward the true minimum.

Thus the ideal learning rate can differ based on “where” the model is on the graph of weights versus loss. A fast learning rate might help the model get out of a ravine. But at other times a slower learning rate could keep the model from shooting in the wrong direction.

Several optimizer functions seek to handle these issues using adaptive learning rates. The optimizers can adjust the learning rate as the training progresses. The optimizers can also use a different learning rate per parameter, which makes sense because the current slope for one parameter’s weight versus loss might differ from another parameter’s. Examples of such adaptive optimizer functions include Adadelta, ADAM, and RMSprop.

A conceptual view on why Adaptive ASHA is superior

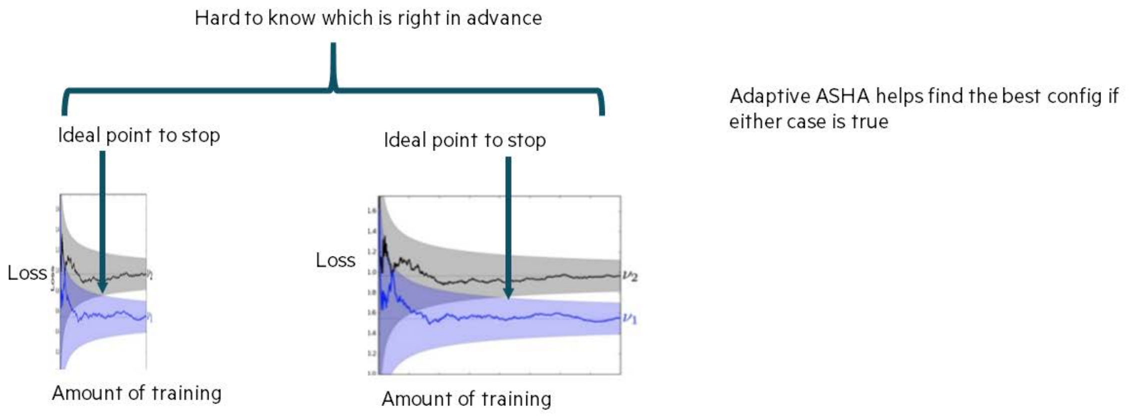


Figure 1-49: A conceptual view on why Adaptive ASHA is superior

The next part of this appendix considers Adaptive ASHA in more detail.

Adaptive ASHA is based on Hyperband, which originally adapted SHA and then later incorporated principles of ASHA.

SHA takes the viewpoint of people at a sports game who slip out earlier when one team is clearly ahead—why stick around when the conclusion is forgone? Why devote resources to training a model with a particular hyperparameter config when you can already see that the config does not work well? And yet others might say: “It’s not over until it’s over.” How can you balance these principles and stop the training at the *right* point?

Think about it like this. If you train a model on the same amount of data multiple times with every possible combination of hyperparameters in your config space, you’ll end up with a lot of different trained models. One of those models will “win” by virtue of exhibiting the lowest validation loss—call that the winner’s terminal validation loss or v_w. Every other model will have its own terminal validation loss somewhere above v_w—call that v_i.

Next think about the model’s journey to its particular terminal validation loss. The model experiences many intermediate validation losses on the way. Loss tends to drop rapidly at first and then slope more gradually downward. An optimal HPO searcher would stop each trial as soon as it sees that the model is headed to a v_i rather than a v_w. But models can have ups and downs on the way, making it harder to distinguish where the model is headed.

An “envelope function” can help; it describes the maximum deviation of the intermediate validation from the terminal validation. Think of it as an “envelope” surrounding the curve. Initially, the envelopes for a model heading to v_i and a model heading to v_w will overlap. You can’t tell for sure where the model is heading. But after some training, the envelops stop overlapping. That’s the moment for our optimal searcher to stop the more poorly performing trial.

SHA implements early stopping. But it does not stop every individual trial at the optimal point. Instead it stops multiple trials together at certain set points. That means that it could stop some trials too early or too late. If you want to be reasonably confident that SHA finds the true best config of the lot, you would tip it towards stopping too late, which means SHA requires more resources than the optimal searcher. However, researchers have shown that, *if given access to the envelope function*, you can implement SHA such that it can find the best config of its population with resources only a constant factor above the optimal searcher.

But—and this is a big “but”—realistically, you do not know the envelope function. That makes it hard to know the best way to implement SHA. If envelopes stop overlapping quickly (fast training), you should optimize your resources by running a large number of trials with a short initial training time. But if training is slower, you should optimize by running a smaller number of trials for a longer initial training time. Without a clear knowledge of the envelope function, how do you choose the right approach?

Hyperband is optimized for *either* case. It calls multiple SHA processes, some with a larger number of trials and a shorter initial training time and others with smaller numbers of trials and a longer initial training time. In the end, Hyperband provides higher confidence that the correct winner was selected. At what price—in terms of increased resource demands—does that confidence come? Hyperband's authors have proven that Hyperband's required resource budget is only a log factor larger than SHA. In a field in which accuracy is so highly prized, this is often a resource cost well worth paying—particularly since it helps you avoid running HPO experiments again and again in search of better results.

After the original publication of the paper on Hyperband, ASHA became more prominent. As you have learned, ASHA functions similarly to SHA, but uses resources more optimally by promoting good performers as soon as possible. Adaptive ASHA incorporates the principles of Hyperband for ASHA processes. It is the most accurate implementation of Hyperband. It does feature some differences, making it more flexible and simpler for users to implement. For example, users can choose whether to implement the conservative mode (closer to Hyperband), standard mode, or aggressive mode (closer to pure ASHA).

If you would like to read more about Hyperband, including the mathematics behind how it works and proof points of how it stacks up against competitors, refer to this paper: https://arxiv.org/abs/1603.06560