Machine Learning Interviews

Machine Learning Systems Design

Chip Huyen
huyenchip.com
@chipro

Table of Contents

Introduction

This part contains 27 open-ended questions that test your ability to put together what you've learned to design systems to solve practical problems. Interviewers give you a problem, possibly related to their products, and ask you to design a machine learning system to solve it. This type of question has become so popular that it's almost guaranteed that you'll be asked at least one during your interview process. In an hour-long interview, you might have time to go over only one or two questions.

These questions don't have single correct answers, though there are answers that are considered correct. There are many ways to solve a problem, and there are many follow-up questions the interviewer can ask to evaluate the candidate's knowledge, implementation ability, and critical thinking skills. Interviewers generally agree that even if you can't get to a working solution, as long as you communicate your thinking process to show that you understand different constraints, trade-offs, and concerns of your system, it's good enough.

These are the kind of questions candidates often both love and hate. Candidates love these questions because they are fun, practical, flexible, and require the least amount of memoization. Candidates hate these questions for several reasons.

First, they lack evaluation guidelines. It's frustrating for candidates when the interviewer asks an open-ended question but expects only one right answer -- the answer that the interviewer is familiar with. It's hard to come up with a perfect solution on the spot and candidates might need help overcoming obstacles. However, many interviewers are quick to dismiss candidates' half-formed solutions because they don't see where the solutions are headed.

Second, these questions are ambiguous. There's no typical structure for these interviews. Each interview starts with a purposefully vague task: design X. It's your job as the candidate to ask for clarification and narrow down the problem. You drive the interview and choose what to focus on. What you choose to focus on speaks volumes about your interest, your experience, and your understanding of the problem.

Many candidates don't even know what a good answer looks like. It's not taught in school. If you've never deployed a machine learning system to users, you might not even know what you need to worry about when designing a system.

When I asked on Twitter what interviewers look for with this type of question, I got varying answers. Dmitry Kislyuk, an engineering manager for Computer Vision at Pinterest, is more interested in the non-modeling parts:

"*Most candidates know the model classes (linear, decision trees, LSTM, convolutional neural networks) and memorize the relevant information, so for me the interesting bits in machine learning systems interviews are data cleaning, data preparation, logging, evaluation metrics, scalable inference, feature stores (recommenders/rankers).*"

Ravi Ganti, a data scientist at WalmartLabs, looks for the ability to divide and conquer the problem:

"*When I ask such questions, what I am looking for is the following. 1. Can the candidate break down the open ended problem into simple components (building blocks) 2. Can the candidate identify which blocks require machine learning and which do not.*"

Similarly, Illia Polosukhin, a co-founder of the blockchain startup NEAR Protocol and who was previously at Google and MemSQL, looks for the fundamental problem-solving skills:

"*I think this [the machine learning systems design] is the most important question. Can a person define the problem, identify relevant metrics, ideate on data sources and possible important features, understands deeply what machine learning can do. Machine learning methods change every year, solving problems stays the same.*"

This book doesn't attempt to give perfect answers -- they don't exist. Instead, it aims to provide a framework for approaching those questions.

Research vs production

To approach these questions, let's first examine the fundamental differences between machine learning in an academic setting and machine learning in production.

In academic settings, people care more about training whereas in production, people care more about serving. Candidates who have only learned about machine learning but haven't deployed a system in the real world often make the mistake of focusing entirely on training: getting the model to do well on some benchmark task without thinking of how it would be used.

Performance requirements

In machine learning research, there's an obsession with achieving state-of-the-art (SOTA) results on benchmarking tasks. To edge out a small increase in performance, researchers often resort to techniques that make models too complex to be useful.

A technique often used by the winners of machine learning competitions, including the famed $1M Netflix Prize and many Kaggle competitions, is ensembling: combining "*multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.*" While it can give you a few percentage point increase in performance, ensembling makes your system more complex, requires much more time to develop and train, and costs more.

A few percentage points might be a big deal on a leaderboard, but might not even be noticeable for users. From a user's point of view, an app with a 95% accuracy is not that different from an app with a 96% accuracy.

Note

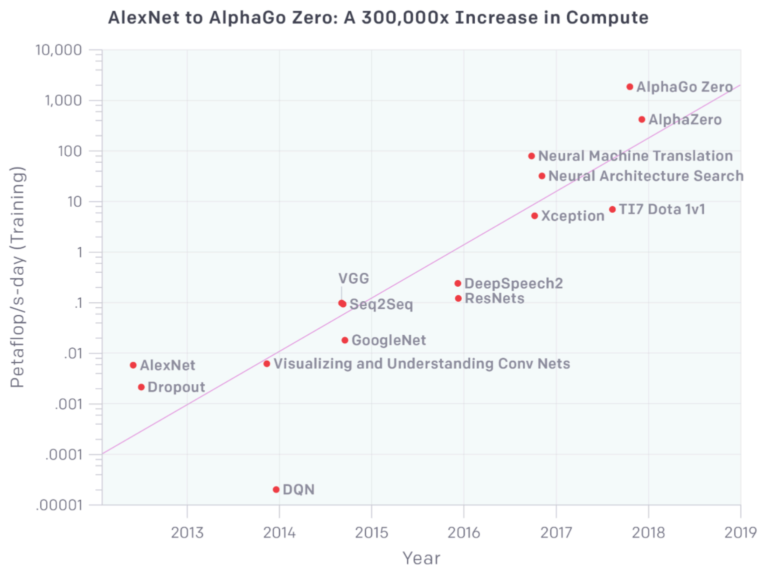
There have been many arguments that leaderboard-styled competitions, especially Kaggle, aren't machine learning. An obvious argument is that Kaggle already does a lot of the steps needed for machine learning for you (Machine learning isn't Kaggle competitions, Julia Evans).

A less obvious, but fascinating, argument is that due to the multiple hypothesis testing scenario that happens when you have multiple teams testing on the same hold-out test set, a model can do better than the rest just by chance (AI competitions don't produce useful models, Luke Oakden-Rayner, 2019).

Compute requirements

In the last decade, machine learning systems have become exponentially larger, requiring exponentially more compute power and exponentially more data to train. According to OpenAI, "*the amount of compute used in the largest AI training runs has doubled every 3.5 months.*"

From AlexNet in 2012 to AlphaGo Zero in 2018, the compute power required increased 300,000 times. The architectural search that resulted in AmoebaNets by the Google AutoML team required 450 K40 GPUs for 7 days (Regularized Evolution for Image Classifier Architecture Search, Real et al., 2018). If done on one GPU, it'd have taken 9 years.



These massive models make ideal headlines, not ideal products. They are too expensive to train, too big to fit onto consumer devices, and too slow to be useful to users. When I talk to companies that want to use machine learning in production, many tell me that they want to do what leading research labs are doing, and I have to explain to them that they don't.

There's undeniably a lot of value in fundamental research. These big models might eventually be useful as the community figures out a way to make them smaller and faster, or can be used as pretrained models on top of which consumer products are developed. However, the goals of research are very different from the goals of production. When asked by engineers to develop systems to be used in production, you need to keep the production goals in mind.

Design a machine learning system

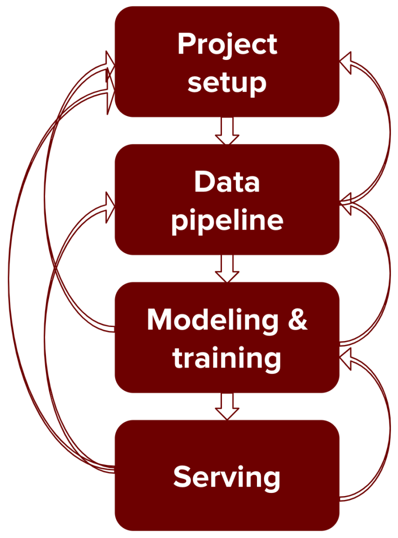
Designing a machine learning system is an iterative process. There are generally four main components of the process: project setup, data pipeline, modeling (selecting, training, and debugging your model), and serving (testing, deploying, maintaining).

The output from one step might be used to update the previous steps. Some scenarios:

- After examining the available data, you realize it's impossible to get the data needed to solve the problem you previously defined, so you have to frame the problem differently.
- After training, you realize that you need more data or need to re-label your data.
- After serving your model to the initial users, you realize that the way they use your product is very different from the assumptions you made when training the model, so you have to update your model.

When asked to design a machine learning system, you need to consider all of these components.

Machine learning project flow



Project setup

Before you even say neural network, you should first figure out as much detail about the problem as possible.

- **Goals**: What do you want to achieve with this problem? For example, if you're asked to create a system to rank what activities to show first in one's newsfeed on Facebook, some of the possible goals are: to minimize the spread of misinformation, to maximize

revenue from sponsored content, or to maximize users' engagement.
- **User experience**: Ask your interviewer for a step by step walkthrough of how end users are supposed to use the system. If you're asked to predict what app a phone user wants to use next, you might want to know when and how the predictions are used. Do you only show predictions only when a user unlocks their phone or during the entire time they're on their phone?
- **Performance constraints**: How fast/good does the prediction have to be? What's more important: precision or recall? What's more costly: false negative or false positive? For example, if you build a system to predict whether someone is vulnerable to certain medical problems, your system must not have false negatives. However, if you build a system to predict what word a user will type next on their phone, it doesn't need to be perfect to provide value to users.
- **Evaluation**: How would you evaluate the performance of your system, during both training and inferencing? During inferencing, a system's performance might be inferred from users' reactions, e.g. how many times they choose the system's suggestions. If this metric isn't differentiable, you need another metric to use during training, e.g. the loss function to optimize. Evaluation can be very difficult for generative models. For example, if you're asked to build a dialogue system, how do you evaluate your system's responses?
- **Personalization**: How personalized does your model have to be? Do you need one model for all the users, for a group of users, or for each user individually? If you need multiple models, is it possible to train a base model on all the data and finetune it for each group or each user?
- **Project constraints**: These are the constraints that you have to worry about in the real world but less so during interviews: how much time you have until deployment, how much compute power is available, what kind of talents work on the project, what available systems can be used, etc.

Resources

- Choosing the Right Metric for Evaluating Machine Learning Models by Alvira Swalin, USF-Data Science, 2018. Part I. Part II.

Data pipeline

In school, you work with available, clean datasets and can spend most of your time on building and training machine learning models. In industry, you probably spend most of your time collecting, annotating, and cleaning data. When teaching, I noticed that many students shied away from data wrangling as they considered it uncool, the way a backend engineer sometimes considers frontend uncool, but the reality is that employers value highly both frontend and data wrangling abilities.

As machine learning is driven more by data than by algorithms, for every formulation of the problem that you propose, you should also tell your interviewer what kind of data and how much data you need: both for training and for evaluating your systems.

You need to specify the input and output of your system. There are many different ways to frame a problem. Consider the app prediction problem above. A naive setup would be to have a user profile (age, gender, ethnicity, occupation, income, technical savviness, etc.) and environment profile (time, location, previous apps used, etc.) as input and output a probability distribution for every single app available. This is a bad approach because there are too many apps and when a new app is added, you have to retrain your model. A better approach is to have the user profile, the environment, and the app profile as input, and output a binary classification whether it's a match or not.

Some of the questions you should ask your interviewer:

- **Data availability and collection**: What kind of data is available? How much data do you already have? Is it annotated and if so, how good is the annotation? How expensive is it to get the data annotated? How many annotators do you need for each sample? How to resolve annotators' disagreements? What's their data budget? Can you utilize any of the weakly supervised or unsupervised methods to automatically create new annotated data from a small amount of humanly annotated data?
- **User data**: What data do you need from users? How do you collect it? How do you get users' feedback on the system, and if you want to use that feedback to improve the system online or periodically?
- **Storage**: Where is the data currently stored: on the cloud, local, or on the users' devices? How big is each sample? Does a sample fit into memory? What data structures are you planning on using for the data and what are their tradeoffs? How often does the new data come in?
- **Data preprocessing & representation**: How do you process the raw data into a form useful for your models? Will you have to do any featuring engineering or feature extraction? Does it need normalization? What to do with missing data? If there's class imbalance in the data, how do you plan on handling it? How to evaluate whether your train set and test set come from the same distribution, and what to do if they don't? If you have data of different types, say both texts, numbers, and images, how are you planning on combining them?
- **Challenges**: Handling user data requires extra care, as any of the many companies that have got into trouble for user data mishandling can tell you.
- **Privacy**: What privacy concerns do users have about their data? What anonymizing methods do you want to use on their data? Can you store users' data back to your servers or can only access their data on their devices?
- **Biases**: What biases might represent in the data? How would you correct the biases? Are your data and your annotation inclusive? Will your data reinforce current societal biases?

Resources

- More data usually beats better algorithms by Anand Rajaraman, Datawocky, 2008.

Modeling

Modeling, including model selection, training, and debugging, is what's often covered in most machine learning courses. However, it's only a small component of the entire process. Some might even argue that it's the easiest component.

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

Source: xkcd

Model selection

Most problems can be framed as one of the common machine learning tasks, so familiarity with common machine learning tasks and the typical approaches to solve them will be very useful. You should first figure out the category of the problem. Is it supervised or unsupervised? Is it regression or classification? Does it require generation or only prediction? If generation, your models will have to learn the latent space of your data, which is a much harder task than just prediction.

Note that these "or" aren't mutually exclusive. An income prediction task can be regression if we output raw numbers, but if we quantize the income into different brackets and predict the bracket, it becomes a classification problem. Similarly, you can use unsupervised learning to learn labels for your data, then use those labels for supervised learning.

Then you can frame the question as a specific task: object recognition, text classification, time series analysis, recommender systems, dimensionality reduction, etc. Keep in mind that there are many ways to frame a problem, and you might not know which way works better until you've tried to train some models.

When searching for a solution, your goal isn't to show off your knowledge of the latest buzzwords but to use the simplest solution that can do the job. Simplicity serves two purposes. First, gradually adding more complex components makes it easier to debug step by step. Second, the simplest model serves as a baseline to which you can compare your more complex models.

Setting up an appropriate baseline is an important step that many candidates forget. There are three different baselines that you should think about:

- *Random baseline*: if your model just predicts everything at random, what's the expected performance?
- *Human baseline*: how well would humans perform on this task?
- *Simple heuristic*: for example, for the task of recommending the app to use next on your phone, the simplest model would be to recommend your most frequently used app. If this simple heuristic can predict the next app accurately 70% of the time, any model you build has to outperform it significantly to justify the added complexity.

Your first step to approaching any problem is to find its effective heuristics. Martin Zinkevich, a research scientist at Google, explained in his handbook *Rules of Machine Learning: Best Practices for ML Engineering* that "*if you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there.*" However, resist the trap of increasingly complex heuristics. If your system has more than 100 nested if-else, it's time to switch to machine learning.

When considering machine learning models, don't forget that non-deep learning models exist. Deep learning models are often expensive to train and hard to explain. Most of the time, in production, they are only useful if their performance is unquestionably superior. For example, for the task of classification, before using a transformer-based model with 300 million parameters, see if a decision tree works. For fraud detection, before wielding complex neural networks, try one of the many popular non-neural network approaches such as k-nearest neighbor classifier.

Most real world problems might not even need deep learning. Deep learning needs data, and to gather data, you might first need users. To avoid the catch-22, you might want to launch your product without deep learning to gather user data to train your system.

Resources

- Machine Learning Algorithms: Which One to Choose for Your Problem by Daniil Korbut, Stats and Bots, 2017.

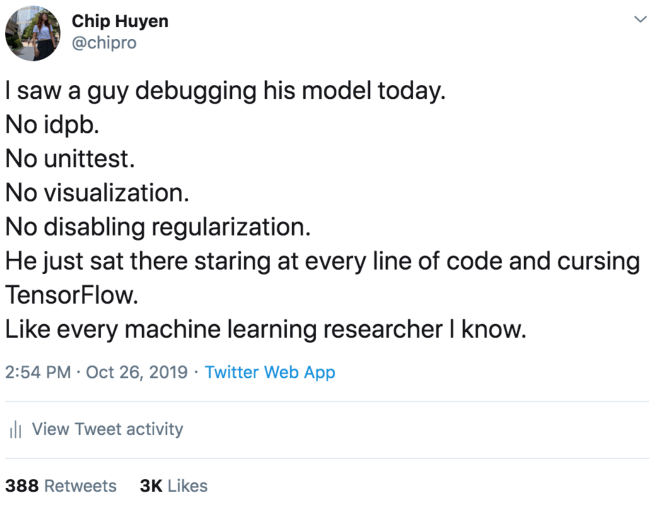
Training

You should be able to anticipate what problems might arise during training and address them. Some of the common problems include: the training loss doesn't decrease, overfitting, underfitting, fluctuating weight values, dead neurons, etc. These problems are covered in the Regularization and training techniques, Optimization, and Activations sections in Chapter 9:

Deep Learning.

Debugging

Have you ever experienced the euphoria of having your model work flawlessly on the first run? Neither have I. Debugging a machine learning model is hard, so hard that poking fun at how incompetent we are at debugging machine learning models has become a sport.



Chip Huyen
@chipro

I saw a guy debugging his model today.
No idpb.
No unittest.
No visualization.
No disabling regularization.
He just sat there staring at every line of code and cursing TensorFlow.
Like every machine learning researcher I know.

2:54 PM · Oct 26, 2019 · Twitter Web App

Typo: idpb is supposed to be ipdb, python interactive debugger tool

There are many reasons that can cause a model to perform poorly:

- **Theoretical constraints**: e.g. wrong assumptions, poor model/data fit.
- **Poor model implementation**: the more components a model has, the more things that can go wrong, and the harder it is to figure out which goes wrong.
- **Snobby training techniques**: e.g. call `model.train()` instead of `model.eval()` during evaluation.
- **Poor choice of hyperparameters**: with the same implementation, a set of hyperparameters can give you the state-of-the-art result but another set of hyperparameters might never converge.
- **Data problems**: mismatched inputs/labels, over-preprocessed data, noisy data, etc.

Most of the bugs in deep learning are invisible. Your code compiles, the loss decreases, but your model doesn't learn anything or might never reach the performance it's supposed to. Having a procedure for debugging and having the discipline to follow that principle are crucial in developing, implementing, and deploying machine learning models.

During interviews, the interviewer might test your debugging skills by either giving you a piece of buggy code and ask you to fix it, or ask you about steps you'd take to minimize the opportunities for bugs to proliferate. There is, unfortunately, still no scientific approach to debugging in machine learning. However, there have been a number tried-and-true debugging techniques published by experienced machine learning engineers and researchers. Here are some of the steps you can take to ensure the correctness of your model.

1. Start simple and gradually add more components

   Start with the simplest model and then slowly add more components to see if it helps or hurts the performance. For example, if you want to build a recurrent neural network (RNN), start with just one level of RNN cell before stacking multiple together, or adding more regularization. If you want to use a BERT-like model (Devlin et al., 2018) which uses both masked language model (MLM) and next sentence prediction loss (NSP), you might want to use only the MLM loss before adding NSP loss.

   Currently, many people start out by cloning an open-source implementation of a state-of-the-art model and plugging in their own data. On the off-chance that it works, it's great. But if it doesn't, it's very hard to debug the system because the problem could have been caused by any of the many components in the model.

2. Overfit a single batch

   After you have a simple implementation of your model, try to overfit a small amount of training data and run evaluation on the same data to make sure that it gets to the smallest possible loss. If it's for image recognition, overfit on 10 images and see if you can get to the accuracy to be 100%, or if it's for machine translation, overfit on 100 sentence pairs and see if you can get to the BLEU score of near 100. If it can't overfit a small amount of data, there's something wrong with your implementation.

3. Set a random seed

   There are so many factors that contribute to the randomness of your model: weight initialization, dropout, data shuffling, etc. Randomness makes it hard to compare results across different experiments -- you have no idea if the change in performance is due to a change in the model or a different random seed. Setting a random seed ensures consistency between different runs. It also allows you to reproduce errors and other people to reproduce your results.

Resources

- Troubleshooting Deep Neural Networks: A Field Guide to Fixing Your Model. Josh Tobin, 2018.
- Things I wish we had known before we started our first Machine Learning project. Aseem Bansal, towards-infinity, 2018.
- How to unit test machine learning code by Chase Roberts, 2017
- A Recipe for Training Neural Networks. Andrej Karpathy, 2019.
- Practical Advice for Building Deep Neural Networks. Matt Holt and Daniel Ricks, BYU's Perception, Control and Cognition Laboratory, 2017.
- Top 6 errors novice machine learning engineers make. Christopher Dossman, AI³ | Theory, Practice, Business, 2017.

Hyperparameter tuning

With different sets of hyperparameters, the same model can give drastically different performance on the same dataset. Melis et al. showed in their 2018 paper *On the State of the Art of Evaluation in Neural Language Models* that weaker models with well-tuned hyperparameters can outperform stronger, more recent models.

Despite knowing its importance, people without real-world experience often ignore systematic approaches to hyperparameter tuning in favor of manual, gut-feeling approach. The most popular method is arguably Graduate Student Descent (GSD), a technique in which a graduate student plays around with the hyperparameters until the model works (GSD is a well-documented technique, see here, here, here, and here).

There have been a lot of research done on hyperparameter search algorithms, as well as tools to help you automatically search for a good set of hyperparameters. You might want to check out some of the popular methods for hyperparameter tuning including random search, grid search, Bayesian optimization. The book *AutoML: Methods, Systems, Challenges* by the AutoML group at the University of Freiburg dedicates its first chapter to hyperparameter optimization, which you can read online for free here.

The performance of each set of hyperparameters is evaluated on the validation set. Keep in mind that not all hyperparameters are created equal. A model's performance might be more sensitive to the change in one hyperparameter, and there have also been research done on accessing the importance of different hyperparameters.

Scaling

As models are getting bigger and more resource-intensive, companies care a lot more about training at scale. It's usually not listed as requirements since expertise in scalability is hard to acquire without regular access to massive compute resources. For machine learning engineering roles, you'll get huge bonus points if you're familiar with common scalability challenges and solutions. Scalability is an elaborate topic that merits its own book. This section covers some common issues, but scratches only the surface.

It's not uncommon to train a model with a dataset that can't be fit into the main memory. This is especially common when dealing with medical data such as CT scans or genome sequences. If you run into a situation like this, you should know how to preprocess (e.g. zero-centering, normalizing, whitening), shuffle, and batch your data when it doesn't fit into memory. When each sample of your data is too large, your model can handle a very small batch size, which can lead to instability for stochastic gradient descent based optimization.

On a very rare case, each sample is so large a single sample can't even fit into the memory, you will have to use techniques such as gradient checkpointing, a technique that leverages the memory footprint/computation tradeoff to make your system do more computation but require less memory. You can use an open-source package `gradient-checkpointing` developed by by Tim Salimans and Yaroslav Bulatov. According to the authors of the package, "*for feed-forward model, we were able to fit more than 10x larger models onto our GPU, at only a 20%*

increase in computation time."

It's almost the norm now for machine learning engineers and researchers to train their models on multiple machines (CPUs, GPUs, TPUs). Modern machine learning frameworks make it easy to do distributed training. The most common parallelization method with multiple workers is data parallelism: you split your data on multiple machines, train your model on all of them, and accumulate gradients. This gives rise to a couple of issues.

The most challenging problem is how to accurately and effectively accumulate gradients from different machines. As each machine produces its own gradient, if your model waits for all of them to finish a run -- this technique is called Synchronous stochastic gradient descent (SSGD) -- stragglers will cause the entire model to slow down.

However, if your model updates the weight using gradient from each machine separately -- this is called Asynchronous SGD (ASGD) -- it will cause gradient staleness because the gradients from one machine has caused the weights to change before the gradients from another machine has come in. How to mitigate gradient staleness is an active area of research.

Second, spreading your model on multiple machines can cause your batch size to be very big. If a machine processes a batch of size 128, then 128 machines processes a batch of size 16,384. If training an epoch on a machine takes 100k steps, training on 128 machines takes under 800 steps. An intuitive approach is to scale learning rate on multiple machines to account for so much more learning at each step, but we also can't make the learning rate too big as it will lead to unstable convergence.

Last but not least, with the same model setup, the master worker will use a lot more resources than other workers. To make the most use out of all machines, you need to figure out a way to balance out the workload among them. The easiest way, but not the most effective way, is to use a smaller batch size on the master worker and a larger batch size on other workers.

With data parallelism, each worker has its own copy of the model and does all the computation necessary for the model. Model parallelism is when different components of your model can be evaluated on different machines. For example, machine 0 handles the computation for the first two layers while machine 1 handles the next two layers, or some machines can handle the forward pass while several others handle the backward pass. In theory, nothing stops you from using both data parallelism and model parallelism. However, in practice, it can pose a massive engineering challenge.

A scaling approach that has gained increasing popularity is to reduce the precision during training. Instead of using a full 32 bits to represent a floating point number, you can use less bits for each number while maintaining a model's predictive power. The paper *Mixed Precision Training* by Paulius Micikevicius et al. at NVIDIA showed that by alternating between full floating point precision (32 bits) and half floating point precision (16 bits), we can reduce the memory footprint of a model by half, which allows us to double our batch size. Less precision also speeds up computation.

Most modern hardwares for deep learning take advantage of mixed and/or reduced precision training. Newer NVIDIA GPUs, such as Volta and Turing architecture, feature Tensor Cores, processing units that support mixed precision training. Compared to standard FP32 on P100, Tensor Cores provide up to 12x higher peak TFLOPS during training, and up to 6x during inferencing. Google TPUs also support training with Bfloat16 (16-bit Brain Floating Point Format), which the company dubbed as "*the secret to high performance on Cloud TPUs.*"

Resources

- Training Neural Nets on Larger Batches: Practical Tips for 1-GPU, Multi-GPU & Distributed setups. Thomas Wolf. 2018.
- Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. Tal Ben-Nun and Torsten Hoefler. 2018.
- A Guide to Scaling Machine Learning Models in Production. Hamza Harkous, Hackernoon. 2017.
- Scaling SGD Batch Size to 32K for ImageNet Training. Yang You, Igor Gitman, and Boris Ginsburg. Berkeley EECS. 2018.

Serving

Before serving your trained models to users, you need to think of experiments you need to run to make sure that your models meet all the constraints outlined in the problem setup. You need to think of what feedback you'd like to get from your users, whether to allow users to suggest better predictions, and from user reactions, how to defer whether your model does a good job.

Training and serving aren't two isolated processes. Your model will continuously improve as you get more user feedback. Do you want to train your model online with each new data point? Do you need to personalize your model to each user? How often should you update your machine learning model?

Some changes to your model require more effort than others. If you want to add more training samples, you can continue training your existing model on the new samples. However, if you want to add a new label class to a neural classification model, you're likely need to retrain the entire system.

If it's a prediction model, you might want to measure your model's confidence with each prediction so that you can show only predictions that your model is confident about. You might also want to think about what to do in case of low confidence -- e.g. would you refer your user to a human specialist or collect more data from them?

You should also think about how to run inferencing: on the user device or on the server and the tradeoffs between them. Inferencing on the user phone consumes the phone's memory and battery, and makes it harder for you to collect user feedback. Inferencing on the cloud increases the product latency, requires you to set up a server to process all user requests, and might scare away privacy-conscious users.

And there's the question of interpretability. If your model predicts that someone shouldn't get a loan, that person deserves to know the reason why. You need to consider the performance/

interpretability tradeoffs. Making a model more complex might increase its performance but make the results harder to interpret.

For complex models with many different components, it's especially important to conduct ablation studies -- removing each component while keeping the rest -- to determine the efficiency of each component. You might find components whose removals don't significantly reduce the model's performance but significantly reduce its complexity.

You also need to think about the potential biases and misuses of your model. Does it propagate any gender and racial biases from the data, and if so, how will you fix it? What happens if someone with malicious intent has access to your system?

On the engineering side, there are many challenges involved in deploying a machine learning model. However, most companies likely have their own deployment teams who know a lot about deployment and less about machine learning.

Note: The assumptions your model is making

The statistician George Box said in 1976 that "*all models are wrong, but some are useful.*" The real world is intractably complex, and models can only approximate using assumptions. Every single model comes with its own assumptions. It's important to think about what assumptions your model makes and whether our data satisfies those assumptions.

Below are some of the common assumptions. It's not meant to be an exhaustive list, but just a demonstration.

- Prediction assumption: every model that aims to predict an output Y from an input X makes the assumption that it's possible to predict Y based on X.
- IID: Neural networks assumes that the data points are independent and identically distributed.
- Smoothness: Every supervised machine learning method assumes that there's a set of functions that can transform inputs into outputs such that similar inputs are transformed into similar outputs. If an input X produces an output Y, then an input close to X would produce an output proportionally close to Y.
- Tractability: Let X be the input and Z be the latent representation of X. Every generative model makes the assumption that it's tractable to compute the probability P(Z I X).
- Boundaries: A linear classifier assumes that decision boundaries are linear.
- Conditional independence: A Naive Bayes classifier assumes that the attribute values are independent of each other given the class. Normally distributed: many statistical methods assume that data is normally distributed.

Note: Tips on preparing

The list of steps above is long and intimidating. Think of a project you did in the past and try to answer the following questions.

- How did you collect the data? How did you process your data?
- How did you decide what models to use? What models did eventually try? What models did better? Why? Any surprise?
- How did you evaluate your models?
- If you did the project again, what would you do differently?

Resources

- Rules of Machine Learning: Best Practices for ML Engineering. Martin Zinkevich, 2017.
- How to build scalable Machine Learning systems - Part II: Architecting a Machine Learning Pipeline. Semi Koen, Towards Data Science, 2017.
- A Brief History of Machine Learning Models Explainability. Zelros AI, 2018.
- The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation. Miles Brundage et al., 2018.
- Fairness in Machine Learning Engineering crash course. Google.

Case studies

To learn to design machine learning systems, it's helpful to read case studies to see how actual teams deal with different deployment requirements and constraints. Many companies, such as Airbnb, Lyft, Uber, Netflix, run excellent tech blogs where they share their experience using machine learning to improve their products and/or processes. If you're interested in a company, you should visit their tech blogs to see what they've been working on -- it might come up during your interviews! Below are some of these great case studies.

1. Using Machine Learning to Predict Value of Homes On Airbnb (Robert Chang, Airbnb Engineering & Data Science, 2017)

   In this detailed and well-written blog post, Chang described how Airbnb used machine learning to predict an important business metric: the value of homes on Airbnb. It walks you through the entire workflow: feature engineering, model selection, prototyping, moving prototypes to production. It's completed with lessons learned, tools used, and code snippets too.

2. Using Machine Learning to Improve Streaming Quality at Netflix (Chaitanya Ekanadham, Netflix Technology Blog, 2018)

   As of 2018, Netflix streams to over 117M members worldwide, half of those living outside the US. This blog post describes some of their technical challenges and how they use machine learning to overcome these challenges, including to predict the network quality, detect device anomaly, and allocate resources for predictive caching.

3. 150 Successful Machine Learning Models: 6 Lessons Learned at Booking.com (Bernardi et al., KDD, 2019).

   As of 2019, Booking.com has around 150 machine learning models in production. These models solve a wide range of prediction (e.g. predicting users' travel preferences and how many people they travel with) and optimization (e.g.optimizing the background images and reviews to show for each user). Adrian Colyer gave a good summary of the six lessons learned here:

   - Machine learned models deliver strong business value.
   - Model performance is not the same as business performance.
   - Be clear about the problem you're trying to solve.
   - Prediction serving latency matters.
   - Get early feedback on model quality.
   - Test the business impact of your models using randomized controlled trials.

4. How we grew from 0 to 4 million women on our fashion app, with a vertical machine learning approach (Gabriel Aldamiz, HackerNoon, 2018)

   To offer automated outfit advice, Chicisimo tried to qualify people's fashion taste using machine learning. Due to the ambiguous nature of the task, the biggest challenges are framing the problem and collecting the data for it, both challenges are addressed by the article. It also covers the problem that every consumer app struggles with: user retention.

5. Machine Learning-Powered Search Ranking of Airbnb Experiences (Mihajlo Grbovic, Airbnb Engineering & Data Science, 2019)

   This article walks you step by step through a canonical example of the ranking and recommendation problem. Four main steps are system design, personalization, online scoring, and business aspect. The article explains which features to use, how to collect data and label it, why they chose Gradient Boosted Decision Tree, which testing metrics to use, what heuristics to take into account while ranking results, how to do A/B testing during deployment. Another wonderful thing about this post is that it also covers personalization to rank results differently for different users.

6. From shallow to deep learning in fraud (Hao Yi Ong, Lyft Engineering, 2018)

   Fraud detection is one of the earliest use cases of machine learning in industry. This article explores the evolution of fraud detection algorithms used at Lyft. At first, an algorithm as simple as logistic regression with engineered features was enough to catch most fraud cases. Its simplicity allowed the team to understand the importance of different features. Later, when fraud techniques have become too sophisticated, more complex models are required. This article explores the tradeoff between complexity and interpretability, performance and ease of deployment.

7. Space, Time and Groceries (Jeremy Stanley, Tech at Instacart, 2017)

   Instacart uses machine learning to solve the task of path optimization: how to most efficiently assign tasks for multiple shoppers and find the optimal paths for them. The article explains the entire process of system design, from framing the problem, collecting data, algorithm and metric selection, topped with tutorial for beautiful visualization.

8. Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering, 2018)

   With massive data comes massive engineering requirement. Relying heavily on data for decision making, "from forecasting rider demand during high traffic events to identifying and addressing bottlenecks in our driver-partner sign-up process", Uber has collected "over 100 petabytes of data that needs to be cleaned, stored, and served with minimum latency." This article focuses on the evolution of analytical data warehouse at Uber, from Vertica to Hadoop to their own Spark library Hudi, each with their limitations analyzed and addressed.

9. Creating a Modern OCR Pipeline Using Computer Vision and Deep Learning (Brad Neuberg, Dropbox Engineering, 2017)

An application as simple as a document scanner has two distinct components: optical character recognition and word detector. Each requires their own production pipeline, and the end-to-end system requires additional steps for training and tuning. This article also goes into detail the team's effort to collect data, which includes building their own data annotation platform.

10. Scaling Machine Learning at Uber with Michelangelo (Jeremy Hermann and Mike Del Balso, Uber Engineering, 2019)

    Uber uses extensive machine learning in their production, and this article gives an impressive overview of their end-to-end workflow, where machine learning is being applied at Uber, and how their teams are organized.

Exercises

The answers for these questions will be published in the book **Machine Learning Interviews**. You can look at and contribute to community answers to these questions on GitHub here. You can read more about the book and sign up for the book's mailing list here.

Note: many questions are ambiguous on purpose. It's your job, as a candidate, to ask for clarification and narrow down the scope of the problem.

1. Duolingo is a platform for language learning. When a student is learning a new language, Duolingo wants to recommend increasingly difficult stories to read.
    - How would you measure the difficulty level of a story?
    - Given a story, how would you edit it to make it easier or more difficult?
2. Given a dataset of credit card purchases information, each record is labelled as fraudulent or safe, how would you build a fraud detection algorithm?
3. You run an e-commerce website. Sometimes, users want to buy an item that is no longer available. Build a recommendation system to suggest replacement items.
4. For any user on Twitter, how would you suggest who they should follow? What do you do when that user is new? What are some of the limitations of data-driven recommender systems?
5. When you enter a search query on Google, you're shown a list of related searches. How would you generate a list of related searches for each query?
6. Build a system that return images associated with a query like in Google Images.
7. How would you build a system to suggest trending hashtags on Twitter?
8. Each question on Quora often gets many different answers. How do you create a model that ranks all these answers? How computationally intensive is this model?
9. How to you build a system to display top 10 results when a user searches for rental listings in a certain location on Airbnb?
10. Autocompletion: how would you build an algorithm to finish your sentence when you text?
11. When you type a question on StackOverflow, you're shown a list of similar questions to make sure that your question hasn't been asked before. How do you build such a system?
12. How would you design an algorithm to match pool riders for Lyft or Uber?
13. On social networks like Facebook, users can choose to list their high schools. Can you estimate what percentage of high schools listed on Facebook are real? How do we find out, and deploy at scale, a way of finding invalid schools?
14. How would you build a trigger word detection algorithm to spot the word "activate" in a 10 second long audio clip?
15. If you were to build a Netflix clone, how would you build a system that predicts when a user stops watching a TV show, whether they are tired of that show or they're just taking a break?
16. Facebook would like to develop a way to estimate the month and day of people's birthdays, regardless of whether people give us that information directly. What methods would you propose, and data would you use, to help with that task?
17. Build a system to predict the language a text is written in.

18. Predict the house price for a property listed on Zillow. Use that system to predict whether we invest on buying more properties in a certain city.
19. Imagine you were working on iPhone. Everytime users open their phones, you want to suggest one app they are most likely to open first with 90% accuracy. How would you do that?
20. How do you map nicknames (Pete, Andy, Nick, Rob, etc) to real names?
21. An e-commerce company is trying to minimize the time it takes customers to purchase their selected items. As a machine learning engineer, what can you do to help them?
22. Build a chatbot to help people book hotels.
23. How would you design a question answering system that can extract an answer from a large collection of documents given a user query?
24. How would you train a model to predict whether the word "jaguar" in a sentence refers to the animal or the car?
25. Suppose you're building a software to manage the stock portfolio of your clients. You manage X amount of money. Imagine that you've converted all that amount into stocks, and find a stock that you definitely must buy. How do you decide which of your currently owned stocks to drop so that you can buy this new stock?
26. How would you create a model to recognize whether an image is a triangle, a circle, or a square?
27. Given only CIFAR-10 dataset, how to build a model to recognize if an image is in the 10 classes of CIFAR-10 or not?