Chapter 5. Model Development

Note: *This note is a work-in-progress, created for the course CS 329S: Machine Learning Systems Design (Stanford, 2022). For the fully developed text, see the book Designing Machine Learning Systems (Chip Huyen, O'Reilly 2022).*

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

Model Development and Training

In this section, we'll discuss necessary aspects to help you develop and train your model, including how to evaluate different ML models for your problem, creating ensembles of models to solve your problem, experiment tracking and versioning, and distributed training, which is necessary for the scale at which models today are usually trained at. We'll end this section with the more advanced topic of AutoML — using ML to automatically choose a model best for your problem.

Evaluating ML Models

Developing a model starts with selecting the right model for your problem. There are many possible solutions to any given problem, both ML solutions and non-ML solutions. Should you start with the good old logistic regression? You've heard of a fancy new model that is supposed to be the new state-of-the-art for your problem, should you spend two weeks learning that process then three months implementing it? Should you try an ensemble of various decision trees?

If you had unlimited time and compute power, the rational thing to do would be to try all possible solutions and see what is best for you. However, time and compute power are limited resources, and you have to be strategic about what models we select.

When thinking about ML algorithms, many people think of classical ML algorithms versus neural networks. There are a lot of interests in neural networks, especially in deep learning, which is understandable given that most of the AI progress in the last decade is due to neural networks getting bigger and deeper.

Many newcomers to the field that I've talked to think that deep learning is replacing classical ML algorithms. However, even though deep learning is finding more use cases in production, classical ML algorithms are not going away. Many recommendation systems still rely on collaborative filtering and matrix factorization. Tree-based algorithms, including gradient-boosted trees, still power many classification tasks with strict latency requirements.

Even in applications where neural networks are deployed, classic ML algorithms are still being used in tandem, either in an ensemble or to help extract features to feed into neural networks.

When selecting a model for your problem, you don't choose from every possible model out there, but usually focus on a set of models suitable for your problem. For example, if your boss tells you to build a system to detect toxic tweets, you know that this is a text classification problem — given a piece of text, classify whether it's toxic or not — and common models for

text classification include Naive Bayes, Logistic Regression, recurrent neural networks, Transformer-based models such as BERT, GPT, and their variants.

If your client wants you to build a system to detect fraudulent transactions, you know that this is the classic abnormality detection problem — fraudulent transactions are abnormalities that you want to detect — and common algorithms for this problem are many, including k-nearest neighbors, isolation forest, clustering, and neural networks.

Knowledge of common ML tasks and the typical approaches to solve them is essential in this process.

Different types of algorithms require different amounts of labels as well as different amounts of compute power. Some take longer to train than others, while some take longer to make predictions. Non-neural network algorithms tend to be much easier to explain (for example, what features in the email that made the model classify it as spam) than neural networks.

When considering what model to use, it's important to consider not only the model's performance, measured by metrics such as accuracy, F1 score, log loss, but also its other properties such as how much data it needs to run, how much compute and time it needs to both train and do inference, and interpretability. For example, a simple logistic regression model might have lower accuracy than a complex neural network, but it requires less labeled data to start, it's much faster to train, it's much easier to deploy, and it's also much easier to explain why it's making certain predictions.

Comparing ML algorithms is out of the scope for this book. No matter how good a comparison is, it will be outdated as soon as new algorithms come out. Back in 2016, LSTM-RNNs were all the rage and the backbone of the architecture seq2seq (Sequence-to-Sequence) that powered many NLP tasks from machine translation to text summarization to text classification. However, just two years later, recurrent architectures were largely replaced by Transformer architectures for NLP tasks.

To understand different algorithms, the best way is to equip yourself with basic ML knowledge and run experiments with the algorithms you're interested in. To keep up-to-date with so many new ML techniques and models, I find it helpful to monitor trends at major ML conferences such as NeurIPS, ICLR, and ICML as well as following researchers whose work has a high signal-to-noise ratio on Twitter.

Six tips for model selection

Without getting into specifics of different algorithms, here are six tips that might help you decide what ML algorithms to work on next.

1. Avoid the state-of-the-art trap

While helping companies as well as recent graduates get started in ML, I usually have to spend a non-trivial amount of time steering them away from jumping straight into state-of-the-art (SOTA) models. I can see why people want SOTA models. Many believe that these models would be the best solutions for their problems — why try an old solution if you believe that a newer and superior solution exists? Many business leaders also want to use SOTA models because they want to use them to make their businesses appear cutting-edge. Developers might also be more excited getting their hands on new models than getting stuck into the same old things over and over again.

Researchers often only evaluate models in academic settings, which means that a model being SOTA often only means that it performs better than existing models on some static datasets. It doesn't mean that this model will be fast enough or cheap enough for you to implement in your case. It doesn't even mean that this model will perform better than other models on **your** data.

While it's essential to stay up-to-date to new technologies and beneficial to evaluate them for your businesses, the most important thing to do when solving a problem is finding solutions that can solve that problem. If there's a solution that can solve your problem that is much cheaper and simpler than SOTA models, use the simpler solution.

2. Start with the simplest models

Zen of Python states that "simple is better than complex", and this principle is applicable to ML as well. Simplicity serves three purposes. First, simpler models are easier to deploy, and deploying your model early allows you to validate that your prediction pipeline is consistent with your training pile. Second, starting with something simple and adding more complex components step-by-step makes it easier to understand your model and debug it. Third, the simplest model serves as a baseline to which you can compare your more complex models.

Simplest models are not always the same as models with the least effort. For example, pretrained BERT models are complex, but they require little effort to get started with, especially if you use a ready-made implementation like the one in HuggingFace's Transformer. In this case, it's not a bad idea to use the complex solution, given that the community around this solution is well-developed enough to help you get through any problems you might encounter. However, you might still want to experiment with simpler solutions, if you haven't already, to make sure that pretrained BERT is indeed better than those simpler solutions for your problem.

3. Avoid human biases in selecting models

Imagine an engineer on your team is assigned the task of evaluating which model is better for your problem: a gradient boosted tree or a pretrained BERT model. After two weeks, this

engineer announced that the best BERT model outperforms the best gradient boosted tree by 5%. Your team decides to go with the pretrained BERT model.

A few months later, however, a seasoned engineer joins your team. She decides to look into gradient boosted trees again and finds out that this time, the best gradient boosted tree outperforms the pretrained BERT model you currently have in production. What happened?

There are a lot of human biases in evaluating models. Part of the process of evaluating an ML architecture is to experiment with different features and different sets of hyperparameters to find the best model of that architecture. If an engineer is more excited about an architecture, she will likely spend a lot more time experimenting with it, which might result in better performing models for that architecture.

When comparing different architectures, it's important to compare them under comparable setups. If you run 100 experiments for an architecture, it's not fair to only run a couple of experiments for the architecture you're evaluating it against. You also want to run 100 experiments for the other architectures too.

Because the performance of a model architecture depends heavily on the context it's evaluated in — e.g. the task, the training data, the test data, the hyperparameters, etc. — it's extremely difficult to make claims that a model architecture is better than another architecture. The claim might be true in a context, but unlikely true for all possible contexts.

4. Evaluate good performance now vs. good performance later

The best model now doesn't always mean the best model two months from now on. For example, a tree-based model might work better now because you don't have a ton of data yet, but two months from now, you might be able to double your amount of training data, and your neural network might perform much better[1].

A simple way to estimate how your model's performance might change with more data is to use learning curves. A learning curve of a model is a plot of its performance — e.g. training loss, training accuracy, validation accuracy — against the number of training samples it uses, as shown in Figure 5-4. The learning curve won't help you estimate exactly how much performance gain you can get from having more training data, but it can give you a sense of whether you can expect any performance gain at all from more training data.

[1] Andrew Ng has a great lecture where he explains that if a learning algorithm suffers from high bias, getting more training data by itself won't help much. Whereas, if a learning algorithm suffers from high variance, getting more training data is likely to help.

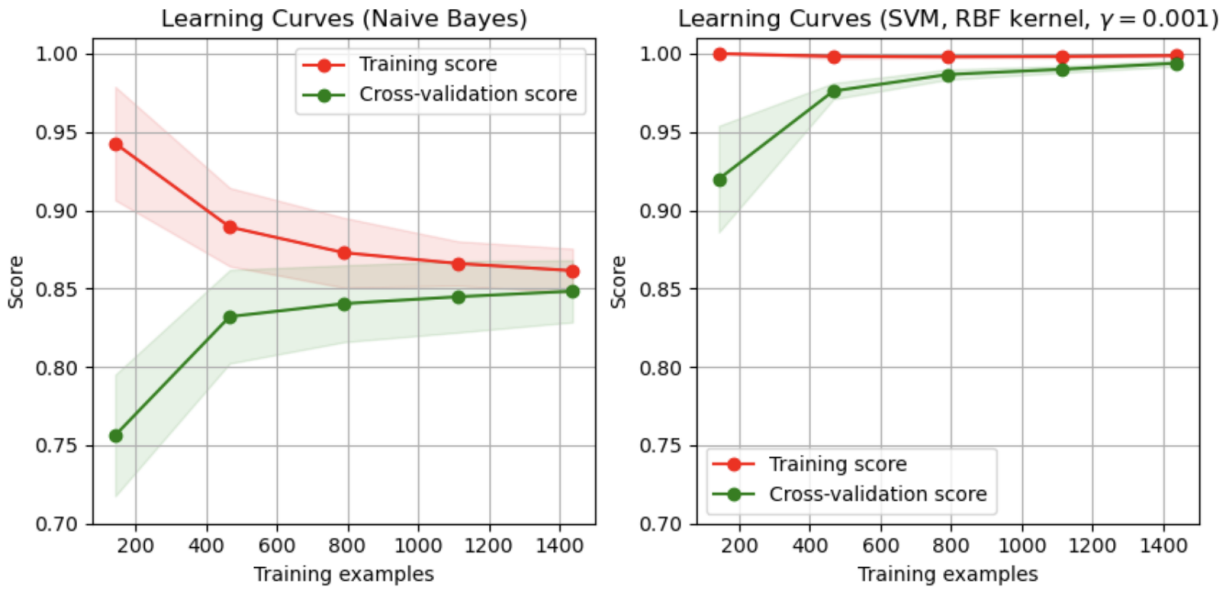

Figure 5-4: The learning curves of a Naive Bayes model and an SVM model. Example from scikit-learn.

A situation that I've encountered is when a team evaluates a simple neural network against a collaborative filtering model for making recommendations. When evaluating both models offline, the collaborative filtering model outperformed. However, the simple neural network can update itself with each incoming example whereas the collaborative filtering has to look at all the data to update its underlying matrix. The team decided to deploy both the collaborative filtering model and the simple neural network. They used the collaborative filtering model to make predictions for users, and continually trained the simple neural network in production with new, incoming data. After two weeks, the simple neural network was able to outperform the collaborative filtering model[2].

While evaluating models, you might want to take into account their potential for improvements in the near future, and how easy/difficult it is to achieve those improvements.

5. Evaluate trade-offs

There are many tradeoffs you have to make when selecting models. Understanding what's more important in the performance of your ML system will help you choose the most suitable model.

One classic example of tradeoff is the false positives and false negatives tradeoff. Reducing the number of false positives might increase the number of false negatives, and vice versa. In a task where false positives are more dangerous than false negatives, such as fingerprint unlocking (unauthorized people shouldn't be classified as authorized and given access), you might prefer a model that makes less false positives. Similarly, in a task where false negatives are more

[2] Continual learning is almost magical. We'll cover it in detail in Chapter 8.

dangerous than false positives, such as covid screening (patients with covid shouldn't be classified as no covid), you might prefer a model that makes less false negatives.

Another example of tradeoff is compute requirement and accuracy — a more complex model might deliver higher accuracy but might require a more powerful machine, such as a GPU instead of a CPU, to generate predictions with acceptable inference latency. Many people also care about the interpretability and performance tradeoff. A more complex model can give a better performance but its results are less interpretable.

6. Understand your model's assumptions

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. Understanding what assumptions a model makes and whether our data satisfies those assumptions can help you evaluate which model works best for your use case.

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 assume that the examples are independent and identically distributed, which means that all the examples are independently drawn from the same joint distribution.
- **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|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.

Ensembles

When considering an ML solution to your problem, you might want to start with a system that contains just one model, and the process of selecting one model for your problem is discussed above. After you've deployed your system, you might think about how to continue improving its performance. One method that has consistently given your system a performance boost is to use

an ensemble of multiple models instead of just an individual model to make predictions. Each model in the ensemble is called a **base learner**. For example, for the task of predicting whether an email is SPAM or NOT SPAM, you might have 3 different models. The final prediction for each email is the majority vote of all these three models. So if at least two base learners output SPAM, the email will be classified as SPAM.

20 out of 22 winning solutions on Kaggle competitions in 2021, as of August 2021, use ensembles[3]. An example of an ensemble used for a Kaggle competition is shown in Figure 5-1. Ensembling methods are less favored in production because ensembles are more complex to deploy and harder to maintain. However, they are still common for tasks where a small performance boost can lead to a huge financial gain such as predicting click-through rate (CTR) for ads.

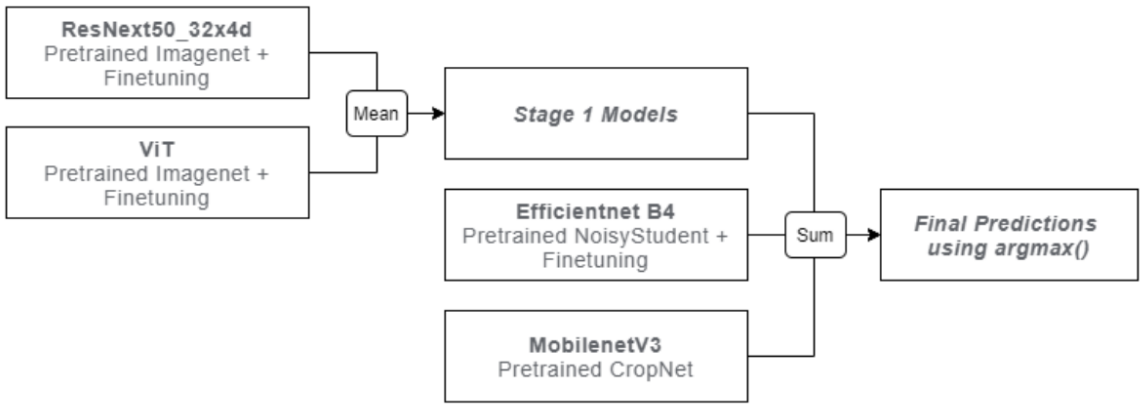

Figure 5-1: The ensemble used in the top solution for the Cassava Leaf Disease Classification competition on Kaggle. Image by Jannis Hanke.
[TODO: The image is not clear so it's probably better to redraw it.]

We'll go over an example to give you the intuition of why ensembling works. Imagine you have three email spam classifiers, each with an accuracy of 70%. Assuming that each classifier has an equal probability of making a correct prediction for each email, and that these three classifiers are not correlated, we'll show that by taking the majority vote of these three classifiers, we can get an accuracy of 78.4%.

For each email, each classifier has a 70% chance of being correct. The ensemble will be correct if at least 2 classifiers are correct. Table 5-1 shows the probabilities of different possible outcomes of the ensemble given an email. This ensemble will have an accuracy of 0.343 + 0.441 = 0.784, or 78.4%.

[3] I went through the winning solutions listed at https://farid.one/kaggle-solutions/. One solution used 33 models

Outputs of 3 models	Probability	Ensemble's output
All 3 are correct	0.7 * 0.7 * 0.7 = 0.343	Correct
Only 2 are correct	(0.7 * 0.7 * 0.3) * 3 = 0.441	Correct
Only 1 is correct	(0.3 * 0.3 * 0.7) * 3 = 0.189	Wrong
None is correct	0.3 * 0.3 * 0.3 = 0.027	Wrong

Table 5-1: Possible outcomes of the ensemble that takes the majority vote from three classifiers

This calculation only holds if the classifiers in an ensemble are uncorrelated. If all classifiers are perfectly correlated — all three of them make the same prediction for every email — the ensemble will have the same accuracy as each individual classifier. When creating an ensemble, the less correlation there is among base learners, the better the ensemble will be. Therefore, it's common to choose very different types of models for an ensemble. For example, you might create an ensemble that consists of one transformer model, one recurrent neural network, and one gradient boosted tree.

There are three ways to create an ensemble: bagging to reduce variance, boosting to reduce bias, and stacking to help with generalization. Other than to help boost performance, according to several survey papers, ensemble methods such as boosting and bagging, together with resampling, have shown to help with imbalanced datasets[4], [5]. We'll go over each of these three methods, starting with bagging.

Bagging

Bagging, shortened from **b**ootstrap **agg**regat**ing**, is designed to improve both the training stability[6] and accuracy of ML algorithms. It reduces variance and helps to avoid overfitting.

Given a dataset, instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a classification or regression model on each of these bootstraps. Sampling with replacement ensures that each bootstrap is independent from its peers. Figure 5-2 shows an illustration of bagging.

[4] A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches (Galar et al., 2011)

[5] Solving class imbalance problem using bagging, boosting techniques, with and without using noise filtering method (Rekha et al., 2019)

[6] Training stability here means less fluctuation in the training loss.

If the problem is classification, the final prediction is decided by the majority vote of all models. For example, if 10 classifiers vote SPAM and 6 models vote NOT SPAM, the final prediction is SPAM.

If the problem is regression, the final prediction is the average of all models' predictions.

Bagging generally improves unstable methods, such as neural networks, classification and regression trees, and subset selection in linear regression. However, it can mildly degrade the performance of stable methods such as k-nearest neighbors[7].

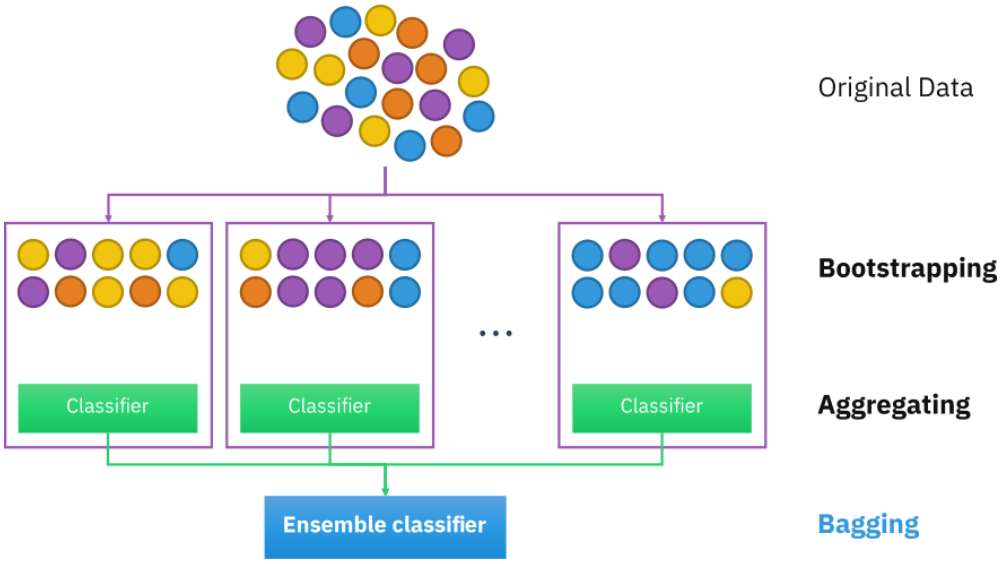

Figure 5-2: Bagging illustration by Sirakorn

A random forest is an example of bagging. A random forest is a collection of decision trees constructed by both bagging and feature randomness, where each tree can pick only from a random subset of features to use.

Boosting

Boosting is a family of iterative ensemble algorithms that convert weak learners to strong ones. Each learner in this ensemble is trained on the same set of samples but the samples are weighted differently among iterations. As a result, future weak learners focus more on the examples that previous weak learners misclassified. Figure 5-3 shows an illustration of boosting.

1. You start by training the first weak classifier on the original dataset.

[7] Bagging Predictors (Leo Breiman, 1996)

2. Samples are reweighted based on how well the first classifier classifies them, e.g. misclassified samples are given higher weight.
3. Train the second classifier on this reweighted dataset. Your ensemble now consists of the first and the second classifiers.
4. Samples are weighted based on how well the ensemble classifies them.
5. Train the third classifier on this reweighted dataset. Add the third classifier to the ensemble.
6. Repeat for as many iterations as needed.
7. Form the final strong classifier as a weighted combination of the existing classifiers — classifiers with smaller training errors have higher weights.

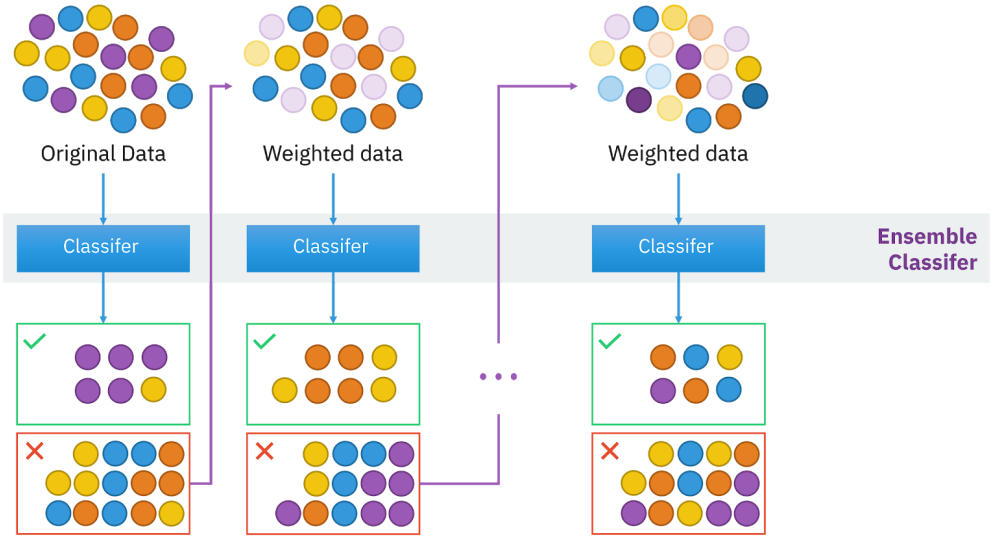

Figure 5-3: Boosting illustration by Sirakorn

An example of a boosting algorithm is Gradient Boosting Machine which produces a prediction model typically from weak decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

XGBoost, a variant of GBM, used to be the algorithm of choice for many winning teams of machine learning competitions. It's been used in a wide range of tasks from classification, ranking, to the discovery of the Higgs Boson[8]. However, many teams have been opting for LightGBM, a distributed gradient boosting framework that allows parallel learning which generally allows faster training on large datasets.

[8] Higgs Boson Discovery with Boosted Trees (Tianqi Chen and Tong He, 2015)

Stacking

Stacking means that you train base learners from the training data then create a meta-learner that combines the outputs of the base learners to output final predictions, as shown in Figure 5-4. The meta-learner can be as simple as a heuristic: you take the majority vote (for classification tasks) or the average vote (for regression tasks) from all base learners. It can be another model, such as a logistic regression model or a linear regression model.

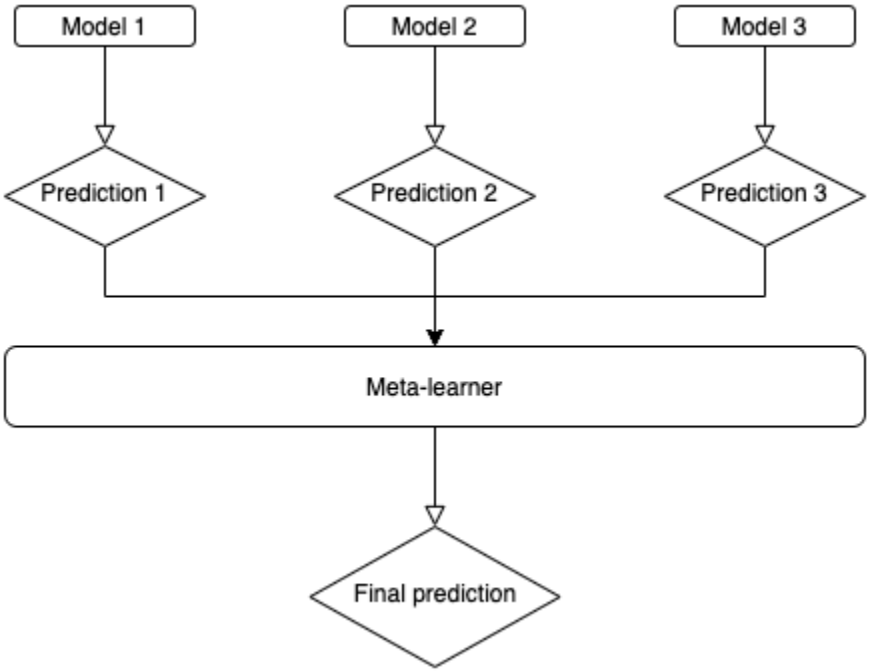

Figure 5-4: A visualization of a stacked ensemble from 3 base learners

For more great advice on how to create an ensemble, refer to this awesome ensemble guide by one of Kaggle's legendary team MLWave.

Experiment Tracking and Versioning

During the model development process, you often have to experiment with many architectures and many different models to choose the best one for your problem. Some models might seem similar to each other and differ in only one hyperparameter — such as one model uses the learning rate of 0.003 while the other model uses the learning rate of 0.002 — and yet their performances are dramatically different. It's important to keep track of all the definitions needed to recreate an experiment and its relevant artifacts. An artifact is a file generated during an experiment — examples of artifacts can be files that show the loss curve, evaluation loss graph, logs, or intermediate results of a model throughout a training process. This enables you to compare different experiments and choose the one best suited for your needs. Comparing

different experiments can also help you understand how small changes affect your model's performance, which, in turn, gives you more visibility into how your model works.

The process of tracking the progress and results of an experiment is called experiment tracking. The process of logging all the details of an experiment for the purpose of possibly recreating it later or comparing it with other experiments is called versioning. These two go hand-in-hand with each other. Many tools originally set out to be experiment tracking tools, such as Weights & Biases, have grown to incorporate versioning. Many tools originally set out to be versioning tools, such as DVC, have also incorporated experiment tracking.

Experiment tracking

A large part of training an ML model is babysitting the learning processes. Many problems can arise during the training process, including loss not decreasing, overfitting, underfitting, fluctuating weight values, dead neurons, and running out of memory. It's important to track what's going on during training not only to detect and address these issues but also to evaluate whether your model is learning anything useful.

When I just started getting into ML, all I was told to track was loss and speed. Fast forward several years, people are tracking so many things that their experiment tracking boards look both beautiful and terrifying at the same time. Below is just a short list of things you might want to consider tracking for each experiment during its training process.

- The **loss curve** corresponding to the train split and each of the eval splits.
- The **model performance metrics** that you care about on all non-test splits, such as accuracy, F1, perplexity.
- The **speed** of your model, evaluated by the number of steps per second or, if your data is text, the number of tokens processed per second.
- **System performance metrics** such as memory usage and CPU/GPU utilization. They're important to identify bottlenecks and avoid wasting system resources.
- The values over time of any **parameter and hyperparameter** whose changes can affect your model's performance, such as the learning rate if you use a learning rate schedule, gradient norms (both globally and per layer) if you're clipping your gradient norms, weight norm especially if you're doing weight decay.

In theory, it's not a bad idea to track everything you can. Most of the time, you probably don't need to look at most of them. But when something does happen, one or more of them might give you clues to understand and/or debug your model. However, in practice, due to limitations of tooling today, it can be overwhelming to track too many things, and tracking less important things can distract you from tracking really important things.

Experiment tracking enables comparison across experiments. By observing how a certain change in a component affects the model's performance, you gain some understanding into what that component does.

A simple way to track your experiments is to automatically make copies of all the code files needed for an experiment and log all outputs with their timestamps[9]. However, using third-party experiment tracking tools can give you nice dashboards and allow you to share your experiments with your coworkers.

Versioning

Imagine this scenario. You and your team spent the last few weeks tweaking your model and one of the runs finally showed promising results. You wanted to use it for more extensive tests so you tried to replicate it using the set of hyperparameters you'd noted down somewhere, only to find out that the results weren't quite the same. You remembered that you'd made some changes to the code between that run and the next, so you tried your best to undo the changes from memory because your reckless past self had decided that the change was too minimal to be committed. But you still couldn't replicate the promising result because there are just too many possible ways to make changes.

This problem could have been avoided if you versioned your ML experiments. ML systems are part code, part data so you need to not only version your code but your data as well. Code versioning has more or less become a standard in the industry. However, at this point, data versioning is like floss. Everyone agrees it's a good thing to do but few do it.

There are a few reasons why data versioning is challenging. One reason is that because data is often much larger than code, we can't use the same strategy that people usually use to version code to version data.

For example, code versioning is done by keeping track of all the changes made to a codebase. A change is known as a diff, short for difference. Each change is measured by line-by-line comparison. A line of code is usually short enough for line-by-line comparison to make sense. However, a line of your data, especially if it's stored in a binary format, can be indefinitely long. Saying that this line of 1,000,000 characters is different from the other line of 1,000,000 characters isn't going to be that helpful.

To allow users to revert to a previous version of the codebase, code versioning tools do that by keeping copies of all the old files. However, a dataset used might be so large that duplicating it multiple times might be unfeasible.

[9] I'm still waiting for an experiment tracking tool that integrates with git commits and data version control commits.

To allow for multiple people to work on the same code base at the same time, code versioning tools duplicate the code base on each person's local machine. However, a dataset might not fit into a local machine.

Second, there's still confusion in what exactly constitutes a diff when we version data. Would diffs mean changes in the content of any file in your data repository, only when a file is removed or added, or when the checksum of the whole repository has changed?

As of 2021, data versioning tools like DVC only register a diff if the checksum of the total directory has changed and if a file is removed or added.

Another confusion is in how to resolve merge conflicts: if developer 1 uses data version X to train model A and developer 2 uses data version Y to train model B, it doesn't make sense to merge data versions X and Y to create Z, since there's no model corresponding with Z.

Third, if you use user data to train your model, regulations like GDPR might make versioning this data complicated. For example, regulations might mandate that you delete user data if requested, making it legally impossible to recover older versions of your data.

Aggressive experiment tracking and versioning helps with reproducibility, but doesn't ensure reproducibility. The frameworks and hardware you use might introduce non-determinism to your experiment results[10], making it impossible to replicate the result of an experiment without knowing everything about the environment your experiment runs in.

The way we have to run so many experiments right now to find the best possible model is the result of us treating ML as a blackbox. Because we can't predict which configuration will work best, we have to experiment with multiple configurations. However, I hope that as the field progresses, we'll gain more understanding into different models and can reason about what model will work best instead of running hundreds or thousands of experiments.

[SIDEBAR]
Debugging ML Models

Debugging is an inherent part of developing any piece of software. ML models aren't an exception. Debugging is never fun, and debugging ML models can be especially frustrating for the following three reasons.

[10] Notable examples include atomic operations in CUDA where non-deterministic orders of operations lead to different floating point rounding errors between runs.

First, ML models fail silently. When traditional software fails, you might get some warnings such as crashes, runtime errors, 404. However, AI applications can fail silently. The code compiles. The loss decreases as it should. The correct functions are called. The predictions are made, but the predictions are wrong. The developers don't notice the errors. And worse, users don't either and use the predictions as if the application was functioning as it should.

Second, even when you think you've found the bug, it can be frustratingly slow to validate whether the bug has been fixed. When debugging a traditional software program, you might be able to make changes to the buggy code and see immediately the result. However, when making changes to an ML model, you might have to retrain the model and wait until it converges to see whether the bug is fixed, which can take hours. In some cases, you can't even be sure whether the bugs are fixed until the application is deployed to the users.

Third, debugging ML models is hard because of their cross-functional complexity. There are many components in an ML system: data, labels, features, machine learning algorithms, code, infrastructure, ... These different components might be owned by different teams. For example, data is managed by the data science team, labels by subject matter experts, ML algorithms by the ML engineers, and infrastructure by the DevOps engineers. When an error occurs, it could be because of any of these components or a combination of them, making it hard to know where to look or who should be looking into it.

Here are some of the things that might cause an ML model to fail.

- **Theoretical constraints**: As discussed above, each model comes with its own assumptions about the data and the features it uses. A model might fail because the data it learns from doesn't conform to its assumptions. For example, you use a linear model for the data whose decision boundaries aren't linear.
- **Poor implementation of model**: The model might be a good fit for the data, but the bugs are in the implementation of the model. For example, if you use PyTorch, you might have forgotten to stop gradient updates during evaluation when you should. The more components a model has, the more things that can go wrong, and the harder it is to figure out which goes wrong. However, with models being increasingly commoditized and more and more companies using off-the-shelf models, this is becoming less of a problem.
- **Poor choice of hyperparameters**: With the same model, a set of hyperparameters can give you the state-of-the-art result but another set of hyperparameters might cause the model to never converge. The model is a great fit for your data, and its implementation is correct, but a poor set of hyperparameters might render your model useless.
- **Data problems**: Many things that could go wrong in data collection and preprocessing that might cause your models to perform poorly, such as data samples and labels being incorrectly matched, label inaccuracies, features normalized using outdated statistics, and more.

- **Poor choice of features**: There might be many possible features for your models to learn from. Too many features might cause your models to overfit to the training data. Too few features might lack predictive power to allow your models to make good predictions.

Debugging should be both preventive and curative. You should have healthy practices to minimize the opportunities for bugs to proliferate as well as a procedure for detecting, locating, and fixing bugs. Having the discipline to follow both the best practices and the debugging procedure is crucial in developing, implementing, and deploying ML models.

There is, unfortunately, still no scientific approach to debugging in ML. However, there have been a number of tried-and-true debugging techniques published by experienced ML engineers and researchers. Here are three of them. Readers interested in learning more might want to check out Andrej Karpathy's awesome post A Recipe for Training Neural Networks.

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 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.

[/SIDEBAR]

AutoML

There's a joke that a good ML researcher is someone who will automate themselves out of job, designing an AI algorithm intelligent enough to design itself. It was funny until the TensorFlow DevSummit 2018, where Jeff Dean took the stage and declared that Google intended on replacing ML expertise with 100 times more computational power, introducing AutoML to the excitement and horror of the community. Instead of paying a group of 100 ML researchers/engineers to fiddle with various models and eventually select a sub-optimal one, why not use that money on compute to search for the optimal model? A screenshot from the recording of the event is shown in Figure 5-5.

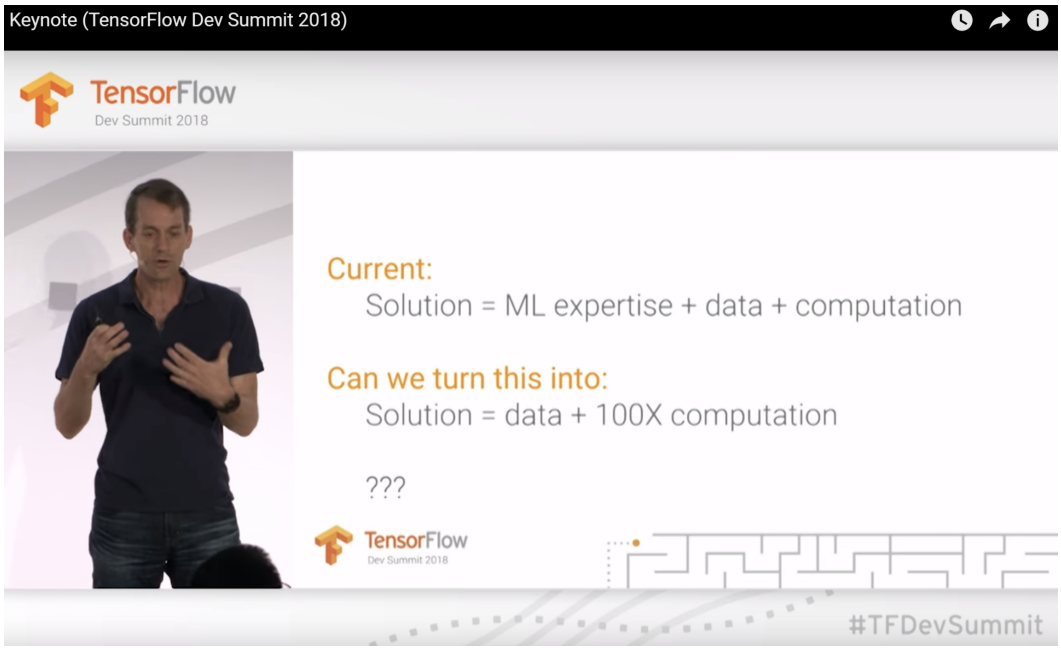

Figure 5-5: Jeff Dean unveiling Google's AutoML at TensorFlow Dev Summit 2018

Soft AutoML: Hyperparameter Tuning

AutoML refers to automating the process of finding ML algorithms to solve real-world problems. One mild form, and the most popular form, of AutoML in production is hyperparameter tuning. A hyperparameter is a parameter supplied by users whose value is used to control the learning process, e.g. learning rate, batch size, number of hidden layers, number of

hidden units, dropout probability, β_1 and β_2 in Adam optimizer, etc. Even quantization level — e.g. mixed-precision, fixed-point — can be considered a hyperparameter to tune.

With different sets of hyperparameters, the same model can give drastically different performances 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, fancier models. The goal of hyperparameter tuning is to find the optimal set of hyperparameters for a given model within a search space — the performance of each set evaluated on the validation set.

Despite knowing its importance, many still ignore systematic approaches to hyperparameter tuning in favor of a manual, gut-feeling approach. The most popular is arguably Graduate Student Descent (GSD), a technique in which a graduate student fiddles around with the hyperparameters until the model works[11].

However, more and more people are adopting hyperparameter tuning as part of their standard pipelines. Popular ML frameworks either come with built-in utilities or have third-party utilities for hyperparameter tuning, e.g. scikit-learn with auto-sklearn[12], TensorFlow with Keras Tuner, Ray with Tune. Popular methods for hyperparameter tuning including random search[13], 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.

When tuning hyperparameters, keep in mind that a model's performance might be more sensitive to the change in one hyperparameter than another, and therefore sensitive hyperparameters should be more carefully tuned.

[WARNING]
One important thing is to never use your test split to tune hyperparameters. Choose the best set of hyperparameters for a model based on its performance on a validation split, then report the model's final performance on the test split. If you use your test split to tune hyperparameters, you risk overfitting your model to the test split.
[/WARNING]

[11] GSD is a well-documented technique, see here, here, here, and here.
[12] auto-sklearn 2.0 also provides basic model selection capacity.
[13] Our team at NVIDIA developed Milano, a framework-agnostic tool for automatic hyperparameter tuning using random search. See the code at https://github.com/NVIDIA/Milano.

Hard AutoML: Architecture search and learned optimizer

Some teams take hyperparameter tuning to the next level: what if we treat other components of a model or the entire model as hyperparameters. The size of a convolution layer or whether or not to have a skip layer can be considered a hyperparameter. Instead of manually putting a pooling layer after a convolutional layer or ReLu after linear, you give your algorithm these building blocks and let it figure out how to combine them. This area of research is known as architectural search, or neural architecture search (NAS) for neural networks, as it searches for the optimal model architecture.

A NAS setup consists of three components:

- a search space that defines possible neural networks, e.g. building blocks to choose from and constraints on how they can be combined.
- a performance estimation strategy to evaluate the performance of a candidate architecture. Even though the final architecture resulting from the research might need retraining, estimating a candidate architecture should be done without having to re-construct or re-train this candidate model from scratch.
- a search strategy to explore the search space. A simple approach is random search — randomly choosing from all possible configurations — which is unpopular because it's prohibitively expensive even for NAS. Common approaches include reinforcement learning[14] (rewarding the choices that improve the performance estimation) and evolution[15] (adding mutations to an architecture, choosing the best-performing ones, adding mutations to them, and so on).

For NAS, the search space is discrete — the final architecture uses only one of the available options for each layer/operation[16], and you have to provide the set of building blocks. The common building blocks are various convolutions of different sizes, linear, various activations, pooling, identity, zero, etc.. The set of building blocks varies based on the base architecture, e.g. convolutional neural networks or recurrent neural networks.

In a typical ML training process, you have a model and then a learning algorithm, a set of functions that specifies how to update the weights of the model. Learning algorithms are also called optimizers, and popular optimizers are, as you probably already know, Adam, Momentum, SGD, etc. In theory, you can include existing learning algorithms as building blocks in NAS and search for one that works best. In practice, this is tricky since optimizers are sensitive to the setting of their hyperparameters, and the default hyperparameters don't often work well across architectures.

---

[14] Neural architecture search with reinforcement learning, Zoph et Le. 2016.
[15] Regularized Evolution for Image Classifier Architecture Search, Real et al., 2018.
[16] You can make the search space continuous to allow differentiation, but the resulting architecture has to be converted into a discrete architecture. See DARTS: Differentiable Architecture Search, Liu et al., 2018.

This leads to an exciting research direction: what if we replace the functions that specify the learning rule with a neural network? How much to update the model's weights will be calculated by this neural network. This approach results in learned optimizers, as opposed to designed optimizers.

Since learned optimizers are neural networks, they need to be trained. You can train your learned optimizer on the same dataset you're training the rest of your neural network on, but this requires you to train an optimizer every time you have a task.

Another approach is to train a learned optimizer once on a set of existing tasks — using aggregated loss on those tasks as the loss function and existing designed optimizers as the learning rule — and use it for every new task after that. For example, Metz et al. constructed a set of thousands of tasks to train learned optimizers. Their learned optimizer was able to generalize to both new datasets and domains as well as new architectures[17]. And the beauty of this approach is that the learned optimizer can then be used to train a better-learned optimizer, an algorithm that improves on itself.

Whether it's architecture search or meta-learning learning rules, the upfront training cost is expensive enough that only a handful of companies in the world can afford to pursue them. However, it's important for people interested in ML in production to be aware of the progress in AutoML for two reasons. First, the resulting architectures and learned optimizers can allow ML algorithms to work off-the-shelf on multiple real-world tasks, saving production time and cost, during both training and inferencing. For example, EfficientNets, a family of models produced by Google's AutoML team, surpass state-of-the-art accuracy with up to 10x better efficiency[18]. Second, they might be able to solve many real-world tasks previously impossible with existing architectures and optimizers.

[SIDEBAR]

Four Phases of ML Model Development

Before we transition to model training, let's take a look at the four phases of ML model development. Once you've decided to explore ML, your strategy depends on which phase of ML adoption you are in. There are four phases of adopting ML. The solutions from a phase can be used as baselines to evaluate the solutions from the next phase.

Phase 1. Before machine learning

If this is your first time trying to make this type of prediction from this type of data, start with non-ML solutions. Your first stab at the problem can be the simplest heuristics. For example, to

[17] [2009.11243] Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves (Metz et al. 2020)

[18] EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling (Tan et Le, 2019)

predict what letter users are going to type next in English, you can show the top three most common English letters, “e”, “t”, and “a”, which is correct 30% of the time.

Facebook newsfeed was introduced in 2006 without any intelligent algorithms — posts were shown in chronological order. It wasn’t until 2011 that Facebook started displaying news updates you were most interested in at the top of the feed, as shown in Figure 5-6[19].

Figure 5-6: Facebook newsfeed circa 2006

According to Martin Zinkevich in his magnificent *Rules of Machine Learning: Best Practices for ML Engineering*:

> *“If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there.[20]”*

[19] The Evolution of Facebook News Feed (Samantha Murphy, Mashable 2013)
[20] Rules of Machine Learning: Best Practices for ML Engineering (Martin Zinkevich, Google 2019)

You might find out that non-ML solutions work just fine and you don't need ML yet.

Phase 2. Simplest machine learning models
For your first ML model, you want to start with a simple algorithm, something that gives you visibility into its working to allow you to validate the usefulness of your problem framing and your data. Logistic regression XGBoost, K-nearest neighbors can be great for that.

They are also easier to implement and deploy which allows you to quickly build out a framework from data management to development to deployment that you can test and trust.

Phase 3. Optimizing simple models
Once you've had your ML framework in place, you can focus on optimizing the simple ML models with different objective functions, hyperparameter search, feature engineering, more data, and ensembles.

This phase will allow you to answer questions such as how quickly your model decays in production and update your infrastructure accordingly.

Phase 4. Complex systems
Once you've reached the limit of your simple models and your use case demands significant model improvement, experiment with more complex models.

[/SIDEBAR]