The universal workflow of machine learning

Outline

- Steps for framing a machine learning problem
- Steps for developing a working model
- Steps for deploying your model in production and maintaining it

Universal Workflow of Machine Learning

The universal workflow of machine learning is broadly structured in three parts:

- Define the task
- Develop a model
- Deploy the model

Universal Workflow of Machine Learning

- Define the task
  - Understand the problem domain and business logic underlying what the customer asked for.
  - Collect a dataset, understand what the data represents.
  - Choose how you will measure success on the task.

Universal Workflow of Machine Learning

- Develop a model
  - Prepare your data so that it can be processed by a machine learning model.
  - Select a model evaluation protocol and a simple baseline to beat.
  - Train a first model that has generalization power and that can overfit, and then regularize and tune your model until you achieve the best possible generalization performance.

Universal Workflow of Machine Learning

- Deploy the model
  - Present your work to stakeholders.
  - Ship the model to a web server, a mobile app, a web page, or an embedded device.
  - Monitor the model’s performance in the wild.

Define the Task

Define the task

You can’t do good work without a deep understanding of the context of what you’re doing.

- Why is your customer trying to solve this particular problem?
- What value will they derive from the solution?
  - how will your model be used?
  - how will it fit into your customer’s business processes?
- What kind of data is available, or could be collected?
- What kind of machine learning task can be mapped to the business problem?

Frame the problem

The questions that should be on the top of your mind:

- What will your input data be? What are you trying to predict?
  - You can only learn to predict something if you have training data available.
  - In many cases, you will have to resort to collecting and annotating new datasets yourself

Frame the problem

- What type of machine learning task are you facing?
    - Is it binary classification? Multiclass classification? Multiclass, multilabel classification?
    - Scalar regression? Vector regression?
    - Image segmentation?
    - Ranking?
    - Something else, like clustering, generation, or reinforcement learning?
- In some cases, it may be that machine learning isn’t even the best way to make sense of the data, such as plain old-school statistical analysis.

Frame the problem…

- Make sure you understand what systems are already in place and how they work.
    - Perhaps your customer already has a handcrafted algorithm that handles spam filtering or credit card fraud detection, with lots of nested if statements.

Frame the problem…

- .Are there particular constraints you will need to deal with?
  - Perhaps the cookie-filtering model needs to consider the latency constraints to it run on an embedded device rather than on a remote server.

Frame the problem…

- Now, you should know
  - what your inputs will be
  - what your targets will be
  - what broad type of machine learning task the problem maps to

Frame the problem: Examples

- The photo search engine project is a multiclass, multilabel classification task.
- The spam detection project is a binary classification task. If you set “offensive content” as a separate class, it’s a three-way classification task.
- The music recommendation engine turns out to be better handled not via deep learning, but via matrix factorization (collaborative filtering).

Frame the problem: Examples

- The credit card fraud detection project is a binary classification task.
- The click-through-rate prediction project is a scalar regression task.
- The project for finding new archeological sites from satellite images is an image-similarity ranking task: you need to retrieve new images that look the most like known archeological sites.

Collect a dataset

- Once you know what your inputs and targets are going to be, it's time for data collection—the most arduous, time-consuming, and costly part of most machine learning projects.
  - For the chat app's spam detection project, you need to gain access to a dataset of tens of thousands of unfiltered social media posts, and manually tag them as spam, offensive, or acceptable.
  - for the click-through-rate prediction project- you need an extensive record of click- through rate for your past ads, going back years.

Collect a dataset

- If you get an extra 50 hours to spend on a project, the most effective way to allocate them is to collect more data rather than search for incremental modeling improvements.
- Data matters more than algorithms- was most famously made in a 2009 paper by Google researchers titled "The Unreasonable Effectiveness of Data"

Investing in Data Annotation Infrastructure

- Your data annotation process will determine the quality of your targets and the quality of your model. Available options-
  - Should you annotate the data yourself?
  - Should you use a crowdsourcing platform?
  - Should you use the services of a specialized data-labeling company?
- Outsourcing can potentially save time and money, but it takes away control.
- Crowdsourcing is likely to be inexpensive and to scale well, but your annotations may end up being quite noisy.

Investing in Data Annotation Infrastructure…

To pick the best option, consider the constraints you’re working with:

- Do the data labelers need to be subject matter experts, or could anyone annotate the data?
    - The labels for a cat-versus-dog image classification problem can be selected by anyone, but annotating CT scans of bone fractures pretty much requires a medical degree.
- If annotating the data requires specialized knowledge, can you train people to do it? If not, how can you get access to relevant experts?
- If you decide to label your data in-house, ask yourself what software you will use to record annotations. Productive data annotation software will save you a lot of time, so it’s worth investing in it early in a project.

Beware of Non-representative Data

- Deep learning models can only make sense of inputs that are similar to what they've seen before.
- As such, it's critical that the data used for training should be representative of the production data.

Beware of Non-representative Data

- Suppose you're developing an app where users can take pictures of a plate of food to find out the name of the dish.
- You train a model using pictures from an image-sharing social network that's popular with foodies. Come deployment time, feedback from angry users starts rolling in: your app gets the answer wrong 8 times out of 10.
- What's going on? Your accuracy on the **test set was well over 90%!**
- A quick look at user-uploaded data reveals that mobile picture uploads of random dishes from random restaurants taken with random smartphones look nothing like the professional-quality, well-lit, appetizing pictures you trained the model on: your training data wasn't representative of the production data.
- That's a cardinal sin—welcome to machine learning hell.

Understand your data

- It’s pretty **bad practice** to treat a dataset as a black box.
- Before you start training models, you should explore and visualize your data to gain insights about what makes it predictive, which will inform feature engineering and screen for potential issues.

Understand your data

- If your data includes images or natural language text, take a look at a few samples (and their labels) directly.
- If your data contains numerical features, it’s a good idea to plot the histogram of feature values to get a feel for the range of values taken and the frequency of different values.
- If your data includes location information, plot it on a map. Do any clear patterns emerge?
- Are some samples missing values for some features? If so, you’ll need to deal with this when you prepare the data.

Understand your data

- If your task is a classification problem, print the number of instances of each class in your data. Are the classes roughly equally represented? If not, you will need to account for this imbalance.
- Check for target leaking: the presence of features in your data that provide information about the targets and which may not be available in production.
  - If you’re training a model on medical records to predict whether someone will be treated for cancer in the future, and the records include the feature “this person has been diagnosed with cancer,” then your targets are being artificially leaked into your data. Always ask yourself, is every feature in your data something that will be available in the same form in production?

Choose a measure of success

- To control something, you need to be able to observe it.
- To achieve success on a project, you must first define what you mean by success.
    - Accuracy?
    - Precision and recall?
    - Customer retention rate?

Choose a measure of success

- For balanced classification problems, where every class is equally likely,
    - accuracy and the area under a receiver operating characteristic (ROC) curve, abbreviated as ROC AUC, are common metrics.
- For class-imbalanced problems, ranking problems, or multilabel classification,
    - you can use precision and recall, as well as a weighted form of accuracy or ROC AUC.

Develop a Model

Develop a model

- The hardest things in machine learning are framing problems and collecting, annotating, and cleaning data.
- Model development comes next and will be easy in comparison!

Develop a model

- Steps required to develop a model:
  - Prepare the data
  - Choose an evaluation protocol
  - Beat a baseline
  - Scale up: Develop a model that overfits
  - Regularize and tune your model

Prepare the data

- Data preprocessing aims at making the raw data at hand more amenable to neural networks. That includes
    - Vectorization
    - Value normalization
    - Handling missing values

Vectorization

- All inputs and targets in a neural network must typically be tensors of floating-point data (or, in specific cases, tensors of integers or strings).
- Whatever data you need to process—sound, images, text—you must first turn into tensors, a step called data vectorization.
    - For instance, in the previous text-classification example, we started with text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of float32 data.
    - In the example of classifying digits and predicting house prices, the data came in vectorized form, so we were able to skip this step.

Value Normalization

- It isn’t safe to feed into a neural network data that takes relatively large values (e.g., multi-digit integers, which are much larger than the initial values taken by the weights of a network).
- Data that is heterogeneous (e.g., data where one feature is in the range 0– 1 and another is in the range 100–200).
- Doing so can trigger large gradient updates that will prevent the network from converging.

Value Normalization

- To make learning easier for your network, your data should have the following characteristics:
  - Take small values—Typically, most values should be in the 0–1 range.
  - Be homogenous—All features should take values in roughly the same range.
- It is also common to
  - Normalize each feature independently to have a mean of 0.
  - Normalize each feature independently to have a standard deviation of 1.

Handling Missing Values

- What if a feature isn’t available for all samples?
- You’d then have missing values in the training or test data.

Handling Missing Values

- If the feature is categorical,
    - it’s safe to create a new category that means “the value is missing.” The model will automatically learn what this implies with respect to the targets.

Handling Missing Values

- If the feature is numerical,
    - Avoid inputting an arbitrary value like "0", because it may create a discontinuity in the latent space formed by your features, making it harder for a model trained on it to generalize.
    - Consider replacing the missing value with the average or median value for the feature in the dataset.
    - You could also train a model to predict the feature value given the values of other features.

Choose an evaluation protocol

- The purpose of a model is to achieve generalization.
- Every modeling decision made throughout the model development process will be guided by validation metrics that seek to measure generalization performance.

Choose an evaluation protocol

Three common evaluation protocols:

- Maintaining a holdout validation set— when you have plenty of data.
- Doing K-fold cross-validation— when you have too few samples for holdout validation to be reliable.
- Doing iterated K-fold validation— for performing highly accurate model evaluation when little data is available.
- In most cases, the first will work well enough.

- Also, be careful not to have redundant samples between your training set and your validation set.

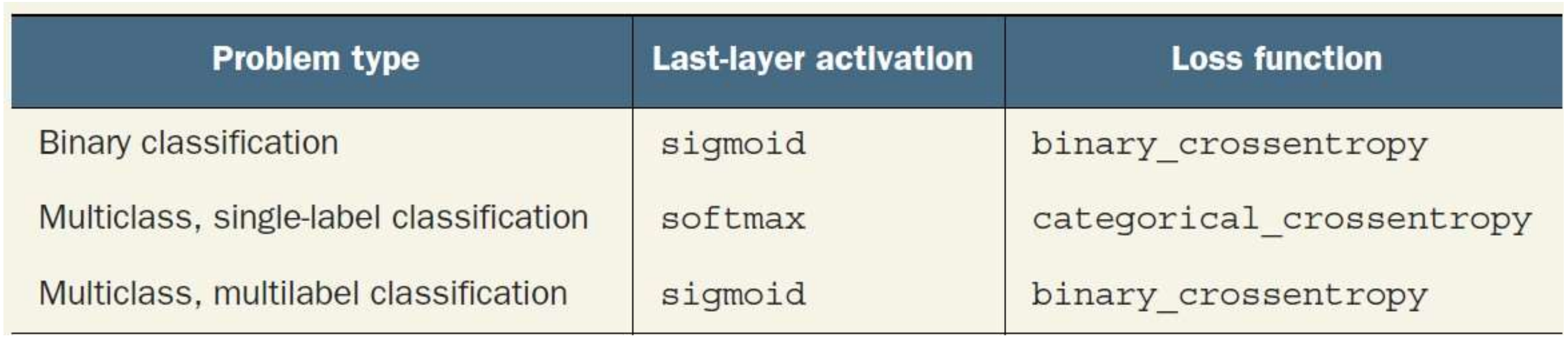
Beat a baseline

At this stage, the three most important things you should focus on:

- *Feature engineering*—Filter out uninformative features (feature selection) and use your knowledge of the problem to develop new features that are likely to be useful.
- *Selecting the correct architecture priors*—What type of model architecture will you use? A densely connected network, a convnet, a recurrent neural network, a Transformer? Is deep learning even a good approach for the task, or should you use something else?
- *Selecting a good-enough training configuration*—What loss function should you use? What batch size and learning rate?

Beat a baseline…

- The following table can help you choose a last-layer activation and a loss function for a few common problem types.



Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`

- If you can’t beat a simple baseline after trying multiple reasonable architectures, it may be that the answer to the question you’re asking isn’t present in the input data.

Scale up: Develop a model that overfits

- The ideal model is one that stands right at the border between underfitting and overfitting.
- To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy:
    - Add layers.
    - Make the layers bigger.
    - Train for more epochs.
- Always monitor the training loss and validation loss,
- When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.

Regularize and tune your model

- Once you’ve achieved statistical power and you’re able to overfit, your goal becomes to maximize generalization performance.

Regularize and tune your model

- Here are some things you should try:
    - Try different architectures; add or remove layers.
    - Add dropout.
    - Or, if your model is small, add L1 or L2 regularization.
    - Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
    - Optionally, iterate on data curation or feature engineering: collect and annotate more data, develop better features, or remove features that don’t seem to be informative.
    - It’s possible to automate a large chunk of this work by using automated hyperparameter tuning software, such as KerasTuner (Ch. 13).

Deploy the Model

Deploy the model

- Your model has successfully cleared its final evaluation on the test set—it's ready to be deployed and to begin its productive life.
- Points that are associated in model deployment:
  - Explain your work to stakeholders and set expectations
  - Ship an inference model
  - Monitor your model in the wild
  - Maintain your model

Explain work to stakeholders and set expectations

- The client might expect human-level performance, especially for processes that were previously handled by people.
- Most machine learning models, because they are (imperfectly) trained to approximate human-generated labels, do not nearly get there.

Explain work to stakeholders and set expectations

- You should clearly convey model performance expectations. Avoid using abstract statements like “The model has 98% accuracy” (which most people mentally round up to 100%).
    - For instance, you could say, “With these settings, the fraud detection model would have a 5% false negative rate and a 2.5% false positive rate.
    - Every day, an average of 200 valid transactions would be flagged as fraudulent and sent for manual review, and an average of 14 fraudulent transactions would be missed.
    - An average of 266 fraudulent transactions would be correctly caught.” Clearly relate the model’s performance metrics to business goals.

Ship an inference model

- You rarely put in production the exact same Python model object that you manipulated during training.

Ship an inference model

- First, you may want to export your model to something other than Python:
    - Your production environment may not support Python at all—for instance, if it’s a mobile app or an embedded system.
- If the rest of the app isn’t in Python (it could be in JavaScript, C++, etc.), the use of Python to serve a model may induce significant overhead.

Ship an inference model

- Second, since your production model will only be used to output predictions (a phase called *inference*), rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint.
- Different model deployment options are:

Ship an inference model

- There are three different model deployment options:
    - Deploying a model as a REST API.
    - Deploying a model on a device.
    - Deploying a model in the browser.

Deploying A Model As A Rest API

- This is the common way to turn a model into a product:
    - Install TensorFlow on a server or cloud instance, and query the model’s predictions via a REST API.
    - You could build your own serving app using something like Flask, or use TensorFlow’s own library for shipping models as APIs, called *TensorFlow Serving* (www.tensorflow.org/tfx/guide/serving). With Tensor-Flow Serving, you can deploy a Keras model in minutes.

Deploying A Model As A Rest API

- An important question when deploying a model as a REST API is whether you want to host the code on your own, or whether you want to use a fully managed third party cloud service.
    - For instance, Cloud AI Platform, a Google product, lets you simply upload your TensorFlow model to Google Cloud Storage (GCS), and it gives you an API endpoint to query it.

Deploying A Model as a Rest API…

- You should use this deployment setup when
  - The application that will consume the model's prediction will have reliable access to the internet (obviously). For instance, if your application is a mobile app, serving predictions from a remote API means that the application won't be usable in airplane mode or in a low-connectivity environment.
  - The application does not have strict latency requirements: the request, inference, and answer round trip will typically take around 500 ms.
  - The input data sent for inference is not highly sensitive: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model.

Deploying A Model on A Device

- Sometimes, you may need your model to live on the same device that runs the application that uses it—maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device.

Deploying A Model on A Device

- You should use this setup when
  - Your model has strict latency constraints or needs to run in a low-connectivity environment. If you’re building an immersive augmented reality application, querying a remote server is not a viable option.
  - Your model can be made sufficiently small that it can run under the memory and power constraints of the target device. You can use the TensorFlow Model Optimization Toolkit to help with this (www.tensorflow.org/model_optimization).

Deploying A Model on A Device

- Getting the highest possible accuracy isn’t mission critical for your task. There is always a trade-off between runtime efficiency and accuracy, so memory and power constraints often require you to ship a model that isn’t quite as good as the best model you could run on a large GPU.
- The input data is strictly sensitive and thus shouldn’t be decryptable on a remote server.

Deploying A Model on A Device…

- Our spam detection model will need to run on the end user’s smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model.
- Likewise, the bad-cookie detection model has strict latency constraints and will need to run at the factory. Thankfully, in this case, we don’t have any power or space constraints, so we can actually run the model on a GPU.
- To deploy a Keras model on a smartphone or embedded device, your go-to solution is TensorFlow Lite (www.tensorflow.org/lite).
    - It’s a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers.
    - It includes a converter that can straightforwardly turn your Keras model into the TensorFlow Lite format.

Deploying a Model in the Browser

- While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in having the model run directly in the browser, on the user’s computer (utilizing GPU resources if they’re available).

Deploying a Model in the Browser

- Use this setup when
  - You want to offload compute to the end user, which can dramatically reduce server costs.
  - The input data needs to stay on the end user’s computer or phone. For instance, in our spam detection project, the web version and the desktop version of the chat app (as a cross-platform app written in Java-Script) should use a locally run model.

Deploying a Model in the Browser

- Use this setup when
  - Your application has strict latency constraints. While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 ms of network round trip.
  - You need your app to keep working without connectivity, after the model has been downloaded and cached.
  - You should only go with this option if your model is small enough that it won't hog the CPU, GPU, or RAM of your user's laptop or smartphone.

Inference Model Optimization

- Optimizing the model for inference is important when deploying in an environment with strict constraints on available power and memory (smartphones and embedded devices) or for applications with low latency requirements.

Inference Model Optimization

- There are two popular optimization techniques you can apply:
  - *Weight pruning*— It’s possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. This reduces the memory and compute footprint of your model, at a small cost in performance metrics. By deciding how much pruning you want to apply, you are in control of the trade-off between size and accuracy.
  - *Weight quantization*—Deep learning models are trained with single-precision floating- point (float32) weights. However, it’s possible to *quantize* weights to 8-bit signed integers (int8) to get an inference-only model that’s a quarter the size but remains near the accuracy of the original model.

Inference Model Optimization

- The TensorFlow ecosystem includes a weight pruning and quantization toolkit (www.tensorflow.org/model_optimization) that is deeply integrated with the Keras API.

Monitor your model in the wild

- Once you’ve deployed a model, you need to keep monitoring its behavior, its performance on new data, and its eventual impact on business metrics.
    - Is user engagement in your online radio up or down after deploying the new music recommender system? Has the average ad click-through rate increased after switching to the new click-through-rate prediction model?
    - If possible, do a regular manual audit of the model’s predictions on production data.
        - send some fraction of the production data to be manually annotated, and compare the model’s predictions to the new annotations.
    - When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system).

Maintain your model

- Lastly, no model lasts forever.
- Over time, the characteristics of your production data will change (*concept drift*), gradually degrading the performance and relevance of your model.
    - The lifespan of your music recommender system will be counted in weeks.
    - For the credit card fraud detection systems, it will be days.
    - A couple of years in the best case for the image search engine.

Maintain your model

- As soon as your model has launched, you should be getting ready to train the next generation that will replace it. As such,
  - Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
  - Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, pay special attention to collecting samples that seem to be difficult for the current model to classify—such samples are the most likely to help improve performance.

Summary

- When you take on a new machine learning project, first define the problem at hand:
  - Understand the broader context of what you're setting out to do—what's the end goal and what are the constraints?
  - Collect and annotate a dataset; make sure you understand your data in depth.
  - Choose how you'll measure success for your problem—what metrics will you monitor on your validation data?

Summary

- Once you understand the problem and you have an appropriate dataset, develop a model:
  - Prepare your data.
  - Pick your evaluation protocol: holdout validation? K-fold validation? Which portion of the data should you use for validation?
  - Achieve statistical power: beat a simple baseline.
  - Scale up: develop a model that can overfit.
  - Regularize your model and tune its hyperparameters, based on performance on the validation data.
- A lot of machine learning research tends to focus only on this step, but keep the big picture in mind.

Summary…

- When your model is ready and yields good performance on the test data, it’s time for deployment:
    - First, make sure you set appropriate expectations with stakeholders.
    - Optimize a final model for inference.
    - Ship the model to the deployment environment of choice—web server, mobile, browser, embedded device, etc.
    - Monitor your model’s performance in production, and keep collecting data so you can develop the next generation of the model.