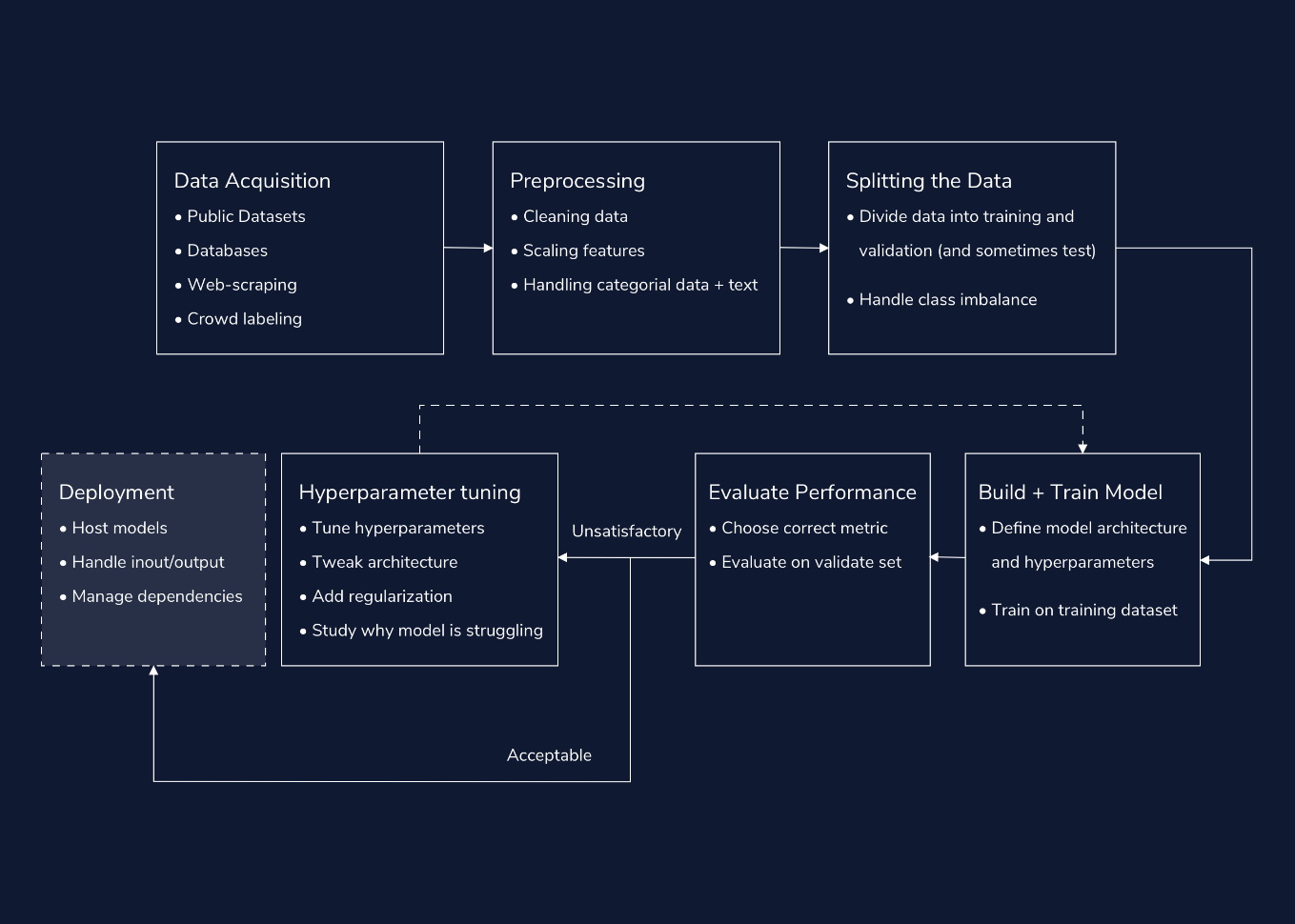
**Deep Learning Workflow**

**In this article, we cover the workflow for a deep learning project.**

**Introduction**

Successfully using deep learning requires more than just knowing how to build neural networks; we also need to know the steps required to apply them in real-world settings effectively.



In this article, we cover the workflow for a deep learning project: how we build out deep learning solutions to tackle real-world tasks. The deep learning workflow has seven primary components:

1. Acquiring data
2. Preprocessing
3. Splitting and balancing the dataset
4. Building and training the model
5. Evaluation
6. Hyperparameter tuning
7. Deploying our solution (For Industry)

**Part 1: Acquiring Data**

In a deep learning project, the most pressing concern is almost always: “Can we get enough labeled data?” The more labeled data we have, the better our model can be. Our ability to acquire data can make or break our solution. Not only is getting data usually the most important part of a deep learning project, but it’s also often the hardest.

Luckily, there are many potential data sources:

**Publicly Available Datasets**

The best source of data is often publicly available datasets. Sites like [Kaggle](https://www.kaggle.com/datasets" \t "_blank) host thousands of large, labeled data sources. Working with these curated datasets helps reduce the overhead of starting a deep learning project.

**Existing Databases**

In some cases, our organization may have a large dataset on hand. Often, these datasets are stored in a Relational Database Management System (RDMS). In this case, we can build our specific dataset using SQL queries.

**Web scraping/APIs**

Online news, social media posts, and search results represent rich streams of data, which we can leverage for our deep learning projects. We do this via Web scraping: the extraction of data from websites. While scraping and collecting data, we should keep in mind ethical considerations, including privacy and consent issues. There are many tools to web scrape in Python, including [BeautifulSoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/" \t "_blank). Many sites, like [Reddit](https://praw.readthedocs.io/en/latest/index.html" \t "_blank) and [Twitter](https://developer.twitter.com/en/docs/twitter-api), have Python *Application Programming Interfaces* (APIs). We can use APIs to gather data from different applications. While some APIs are free, others are paid services.

Depending on the size of the dataset, we may be able to directly write our scraped data to raw data files (e.g., .txt or .csv). However, for larger datasets, we sometimes need to store the resulting data in our own databases.

**Crowd-sourced Labeling**

For many tasks, it’s much easier to acquire data than it is to find labeled data. For example, it is much easier to scrape the raw text from an entire Reddit subreddit than to correctly label the contents of each Reddit post. When automated labeling tools aren’t available, we require human labels. One possibility would be for us to go through our own data, and annotate each datapoint ourselves. However, sometimes that just isn’t feasible, no matter how much coffee we have on hand. An alternative is crowd-sourcing sites like [Amazon Mechanical Turk](https://www.mturk.com/). We can utilize these sites to pay “gig” workers for thousands of human annotations.

**Part 2: Preprocessing**

Once we have built our dataset, we need to preprocess it into useful features for our deep learning models. At a high-level, we have three primary goals when preprocessing data for neural networks: we want to 1) clean our data, 2) handle categorical features and text and 3) scale our real-valued features using normalization or standardization techniques. Preprocessing is also a fantastic opportunity to become more familiar with our data.

**Cleaning data**

Often, our datasets contain noisy examples, extra features, missing data and outliers. It is good practice to test for and remove outliers, remove unnecessary features, fill-in missing data, and filter out noisy examples.

**Scaling features**

Because we initialize neural networks with small weights to stabilize training, our models will struggle when faced with input features that have large values. As a result, we often scale real-valued features in two ways: We can *normalize* features so that they are between 0 and 1, and *standardize* them so they have a mean of zero and a variance of one.

**Handling categorical data and text**

Neural networks expect numbers as their inputs. This means we need to convert all categorical data and text to real-valued numbers. - We usually handle categorical variables by assigning each option its own unique integer or converting them to one-hot encodings. - When working with strings of raw text, we need to handle a few extra processing steps before encoding our words as integers. These steps include tokenizing our data (splitting our text into individual words/tokens), and *padding* our data (adding padding tokens to make all of our examples the same length).

**Part 3: Splitting and Balancing the Dataset**

Once we have processed our data, it’s time to split our dataset. Generally, we split our data into two datasets: training and validation. In certain cases, we also create a third holdout dataset, referred to as the test set. When we don’t do this, we often use the terms “validation” and “test” sets interchangeably.

We train our model on the training dataset and we evaluate it on the validation dataset. If we have defined a third holdout test set, we test our model on this dataset after we have finished selecting our model and tuning our hyperparameters. This third step helps us avoid choosing a set of hyperparameters that only happen to work well on the data we chose for our validation set.

When splitting our dataset, there are two major considerations: the size of our splits, and whether we will stratify our data. After we split our data, we need to address imbalances in our training set.

**Splitting Our Data**

We usually save 10-30% of our data for validation and testing. When we have a smaller corpus, it is more important to assign a larger proportion of data to the validation set. This helps ensure that our validation dataset better represents the true distribution of our data.

Scikit-learn [provides](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) the train\_test\_split function, which splits our data into training and validation datasets and specifies the size of our validation data.

**Stratified Train-Test Splits**

We have to be extra careful when splitting a very imbalanced dataset for classification; it’s very possible that more instances of our minority classes end up in either the training or the validation set. In the first case, our validation metrics will not accurately capture our model’s ability to classify the minority class. In the second case, the model will overestimate the probability of the majority class.

The solution is to use a stratified split: a split that ensures the training and validation sets have the same proportion of examples from each class.

If we set the train\_test\_split function’s stratify parameter to our array of labels, the function will compute the proportion of each class, and ensure that this ratio is the same in our training and validation data.

**Handling Imbalanced Data**

Imbalanced data, where some classes appear much more than others, pose a challenge for deep learning models. If we train neural networks on imbalanced data, our resulting model will be heavily biased towards predicting those majority classes. This is especially problematic because usually we care much more about identifying instances of the minority classes (like rare cases of disease or credit fraud).

There are two main approaches to dealing with imbalanced training data: undersampling and oversampling. These two approaches should be taken-up with utmost caution, and it’s best to have a domain expert on hand to weigh in.

* In undersampling, we balance our data by throwing out examples from our majority class.
* In oversampling, we duplicate instances of our minority class so that they occur more often. A popular alternative to traditional oversampling is called *Synthetic Minority Oversampling TEchnique (SMOTE)*. The SMOTE algorithm creates synthetic examples that are similar to those in our minority class, and adds them to our dataset.

Almost always, we only correct the imbalance in our training data, and leave the validation data as is. In order to only augment our training data, we need to correct for imbalance only after our train-test split.

We never oversample our data before we split it. If we do, copies of our testing data can sneak into our training data. This is called *information leak*.

**Part 4: Building and Training the Model**

Once we have split our dataset, it’s time to choose our loss function, and our layers.

For each layer, we also need to select a reasonable number of hidden units. There is no absolute science to choosing the right size for each layer, nor the number of layers — it all depends on your specific data and architecture.

* It’s good practice to start with a few layers (2-6).
* Usually, we create each layer with between 32 and 512 hidden units.
* We also tend to decrease the size of hidden layers as we move upwards - through the model.
* We usually try SGD and Adam optimizers first.
* When setting an initial learning rate, a common practice is to default to 0.01.

**Part 5: Evaluating Performance**

Each time we train the model, we evaluate its performance on our validation set. When we provide a validation set at training time, Keras handles this automatically. Our performance on the validation set gives us a sense for how our model will perform on new, unseen data.

When considering performance, it’s important to choose the correct metric. If our data set is heavily imbalanced, accuracy (and even AUC) will be less meaningful. In this case, we likely want to consider metrics like precision and recall. [F1-score](https://en.wikipedia.org/wiki/F1_score) is another useful metric that combines both precision and recall. A confusion matrix can help visualize what data-points are misclassified and what aren’t.

**Part 6: Tuning Hyperparameters**

We will almost always need to iterate upon our initial hyperparameters. When training and evaluating our model, we explore different learning rates, batch sizes, architectures, and regularization techniques.

As we tune our parameters, we should watch our loss and metrics, and be on the lookout for clues as to why our model is struggling.:

* Unstable learning means that we likely need to reduce our learning rate and or/increase our batch size.
* A disparity between performance on the training and evaluation sets means we are overfitting, and should reduce the size of our model, or add regularization (like dropout).
* Poor performance on both the training and the test set means that we are underfitting, and may need a larger model or a different learning rate.

A common practice is to start with a smaller model and scale up our hyperparameters until we do see training and validation performance diverge, which means we have overfit to our data.

Critically, because neural network weights are randomly initialized, your scores will fluctuate, regardless of hyperparameters. One way to make accurate judgments is to run the same hyperparameter configuration multiple times, with different random seeds.

Once our results are satisfactory, we are ready to use our model!

If we made a holdout test set, separate from our validation data, now is when we use it to test out our model. The holdout test set provides a final guarantee of our model’s performance on unseen data.

**Part 7: Deployment (For Industry)**

Once we have trained a model, we may want to deploy it into the real world. This is especially true in industry settings, when our networks will be used by our coworkers and customers, or working behind the scenes in our products and internal tools.

When deploying a neural network there are three big considerations…

**How will we handle the compute requirements for running our models?**

It takes a significant amount of computation to evaluate a single input using a neural network, let alone manage traffic from many different users. As a result, when deploying a neural network model in a Docker container, it’s important to host the container where it can access powerful computing resources. Cloud platforms like AWS, GCP and Azure are great places to start. These platforms provide flexible hosting services for applications that can scale up to meet changing demand.

**How will we pass inputs into the model?**

A common approach for interfacing with our model over the web is [Flask](https://flask.palletsprojects.com/en/1.1.x/), a Python-based web framework. Flask can [handle requests and pass inputs](https://towardsdatascience.com/deploying-keras-deep-learning-models-with-flask-5da4181436a2) to our model.

**How will we run the code and manage dependencies, wherever we host our application? (Optional)**

This can depend on where we host our model. However, a popular general-purpose solution to this last question is [Docker Containers](https://www.docker.com/why-docker" \t "_blank). Docker containers are a way to package up our code and its dependencies (e.g. the correct version of TensorFlow), in such a way that our application can run quickly in any computing environment.

**Conclusion**

In this article, we covered the general workflow for a deep learning project. We covered a lot of material, so don’t sweat every detail. Our goal is to provide a sense for the overarching flow of a deep learning project, from data acquisition and preprocessing to evaluation and hyperparameter tuning.

These guidelines are not completely ironclad rules. Rather, in a successful deep learning project, we often pivot back and forth between different steps, continuously tweaking and debugging our scripts, data, and architectures on our quest for the best performing model.