Technical Strategy for AI Engineers (based on the book *Machine Learning Yearning* by Andrew Ng)

Created by Aldo Zaimi

March 3, 2019

1. **Scales drive machine learning progress**

   - The two biggest drivers of the recent ML progress: (i) **data availability** and (ii) **computational scale**.
   - The best performances are usually obtained by: (i) training a **very large neural network** and (ii) having a **huge amount of data**.

2. **Setting up development and test sets**

   - Data should be divided into 3 sets: (i) **training set**, (ii) **development set** (dev set, also called hold-out cross validation set) and (iii) **test set**.
   - Dev and test sets should reflect data you expect to get in the future and want to do well on.
   - It is recommended to choose dev and test sets that are drawn from the same distribution: easier to figure out what is and isn't working.
   - **Dev set**: should be large enough to detect even smaller performance improvements.
   - **Test set**: (i) should be large enough to give high confidence on the overall performance of the system, (ii) can be around 30% if smaller dataset, (iii) no need to be excessively large.
   - Use a **single-number evaluation metric** to optimize: (i) makes it easier to compare algorithms, (ii) **combine metrics** into a single-number metric (e.g. F1 score for both precision and recall), (iii) use average or weighted average to combine metrics, (iv) define **satisficing metrics** that only need to meet a certain threshold (i.e. use only one evaluation metric and (N-1) satisficing metrics).
   - **Iterations** in ML applications: (i) **idea**, (ii) **code**, (iii) **experiment** on your dev set using only one performance metric.
   - Change dev/test sets and/or metrics if: (i) the dev/test set distribution is not representative of the actual distribution you want to do well on, (ii) your algorithm is overfitting (i.e. dev set performance much better than test set performance), (iii) the metric is measuring something not relevant to the task.

3. **Basic error analysis**

   - Build your first system quickly, then **iterate**: (i) a basic system is enough to identify the most promising directions.

- Look at the **misclassified dev/test set samples**: (i) evaluate the frequency of each type of misclassification to help **prioritize projects** and inspire new directions, (ii) use a spreadsheet to evaluate multiple misclassifications in parallel during error analysis, (iii) clean up mislabelled dev/test set samples if significant amount of mislabelled samples, (iv) apply the same procedures on dev and test sets.

4. **Bias and variance**

   - **Bias**: (i) the algorithm's error rate on the training set, (ii) high bias when high training error and similar error on the dev/test set, (iii) **underfitting**, (iv) can be divided into **unavoidable bias** (the optimal error rate that the algorithm can achieve, also called Bayes error rate) and **avoidable bias**.
   - **Variance**: (i) how much worse the algorithm does on the dev and/or test set than the training set, (ii) high variance when low training error but does not generalize well on the dev/test set, (iii) **overfitting**.
   - Total dev/test set error = Bias + Variance = (Unavoidable bias + Avoidable bias) + Variance.
   - **Addressing bias**: (i) the typical solution is to **increase the size of the model** (i.e. larger neural network with more layers/neurons), (ii) **modify input features** based on insights from error analysis, (iii) **reduce or eliminate regularization** (i.e. L2, L1, dropout), (iv) **modify model architecture** (use an architecture that is more suitable for your problem).
   - **Addressing variance**: (i) the typical solution is to **increase training set size**, (ii) **add regularization** (L2, L1, dropout), (iii) add **early stopping**, (iv) perform **feature selection** to decrease number/type of input features (very useful when your training set is small), (v) decrease the model size (only use if concerned about the computational cost and regularization has already been tested), (vi) **modify input features** based on insights from error analysis, (vii) **modify model architecture** (use an architecture that is more suitable for your problem).

5. **Learning curves**

   - What to plot: (i) **dev set error** vs the training set size, (ii) **training set error** vs the training set size.
   - Increasing the training set size usually: (i) decreases the dev set error, (ii) increases the training set error.
   - You should usually try to: (i) keep the training error curve below the desired performance error (low bias) and (ii) keep the dev error curve close to the training error curve (low variance).
   - Typical **high (avoidable) bias curves**: at the largest training set size (i) there is a large gap between the training error and the desired performance and (ii) there is a small gap between the training and dev curves.
   - Typical **high variance curves**: at the largest training size (i) the training error curve is relatively low and (ii) the dev error curve is much higher than the training error curve.
   - Typical **high bias and variance curves**: (i) training error curve much higher than the desired level of performance and (ii) dev error curve much larger than the training error curve.

- What to do if small training set or noisy training curves: (i) plot the average training and dev set curves after doing multiple trainings on different randomly selected subsets of the training set and/or (ii) use balanced subsets of the training set if there is class imbalance in the training set.

6. **Comparing to human-level performance**

   - How to identify tasks that are easier to solve with a ML system: (i) you can easily obtain data from human labelers (high accuracy labels), (ii) human intuition can help error analysis, (iii) you can use human-level performance to estimate the optimal error rate and/or set a desired error rate.

7. **Training and testing on different distributions**

   - Using training and dev/test sets that come from different distributions offers some special challenges.
   - Using all the **data from different distributions** (but similar properties) for the training: (i) the groups of data share some similarities so it gives the neural network more examples, (ii) but it forces the network to use some of its capacity to learn about properties that are specific to one of the groups, meaning less capacity to learn properties on the distribution for which you really care about (could lower algorithm performance).
   - How to decide whether to include **inconsistent data**: (i) if the different distributions of data add a new feature (i.e. the type of distribution), adding both in your training will hurt the performance of your algorithm, but (ii) if the task is consistent in both distributions, then including all the data could help.
   - **Weighting data from different distributions**: (i) useful when you do not have huge computational resource, (ii) you can give one distribution a much lower weight in the optimization objective (by adding a weight factor) to compensate for the different amount of samples coming from the distributions.
   - **Data mismatch** issue: (i) it generalizes well to new data drawn from the same distribution as the training set, but (ii) it does not perform well on data drawn from the dev/test set distribution.
   - The three main issues: (i) **high bias** (does not do well on the training set), (ii) **high variance** (does well on the training set but does not generalize well on unseen data drawn from the same distribution as the training set), (iii) **data mismatch** (generalizes well on new data from the same distribution as the training set, but not on data drawn from the dev/test set distribution).
   - How to identify **bias/variance/mismatch issues**: (i) split data into **4 subsets of data**: (a) **training set** (the data you will train on, can be drawn from different distributions), (b) **training dev set** (drawn from the same distribution as the training set, usually smaller), (c) **dev set** (should reflect the distribution of data that you want your algorithm to do well on) and (d) **test set** (same as dev set, but different samples), (ii) **evaluate training error** using the training set, (iii) **evaluate performance on new data drawn from the training set distribution** using the training dev set, (iv) **evaluate performance on the task** you care about on the dev and/or test sets.
   - When you have **high variance**: (i) low error on the training set (e.g. 1%), (ii) significantly higher error on both the training dev and dev sets (e.g. both 5%).

- When you have **high (avoidable) bias**: (i) high error on the training set (e.g. 10%), (ii) similar errors on the training dev and dev sets (e.g. 11% and 12% respectively).
- When you have **data mismatch**: (i) low error on the training set (e.g. 1%), (ii) similar error on the training dev set (e.g. 2%), (iii) significantly higher error on the dev set (e.g. 7%).
- An algorithm can suffer from any subset of high bias, high variance and data mismatch.
- How to **resolve data mismatch**: (i) try to understand what properties of the data differ between the distributions of your data, (ii) try to find more training data that better matches the dev/test set examples.
- **Artificial data synthesis**: (i) can help create a huge dataset that matches the dev set, but (ii) it is challenging and can decrease performance if not done well.

8. **End-to-end deep learning**

- The rise of **end-to-end learning systems**: (i) very used in neural networks, (ii) very successful in problems where data is abundant, (iii) not recommended when the training set is small (use pipelines with hand-engineered components instead).
- Consider using a **system with pipeline components** if: (i) there is a lot of data available for training **intermediate modules** of the pipeline and (ii) you can divide it into **pipeline components** that are individually easy to build and/or learn (i.e. task simplicity).
- **Error analysis by parts**: (i) attribute each mistake the algorithm makes to one of the pipeline components, (ii) helps deciding which part of the pipeline should you work on improving.
- Strategy to **attribute the error to one component** in a pipeline with 3 components A, B and C: for each mistake that the system makes: (i) modify the output of A to be a perfect output, run the rest of the pipeline and attribute the error to A if the final output is correct, (ii) modify the output of B to be a perfect output, run the rest of the pipeline and attribute the error to B if the final output is correct, (iii) attribute the error to C if not attributed to previous components.