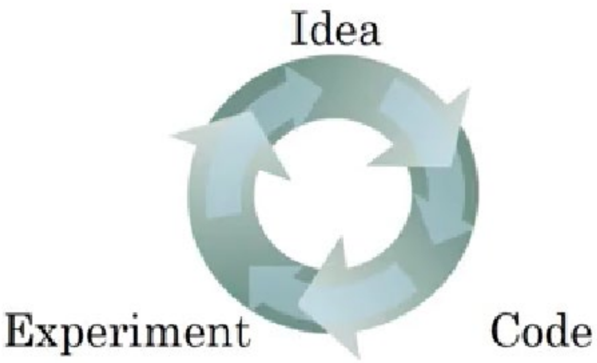
STRUCTURING MACHINE LEARNING PROJECTS

By: Mohamed Aziz Tousli

ORTHOGONALIZATION



<u>Orthogonalization</u>: Tune the knobs of the system in a way to modify each element <u>separately</u>

<u>Chain of assumption</u>:

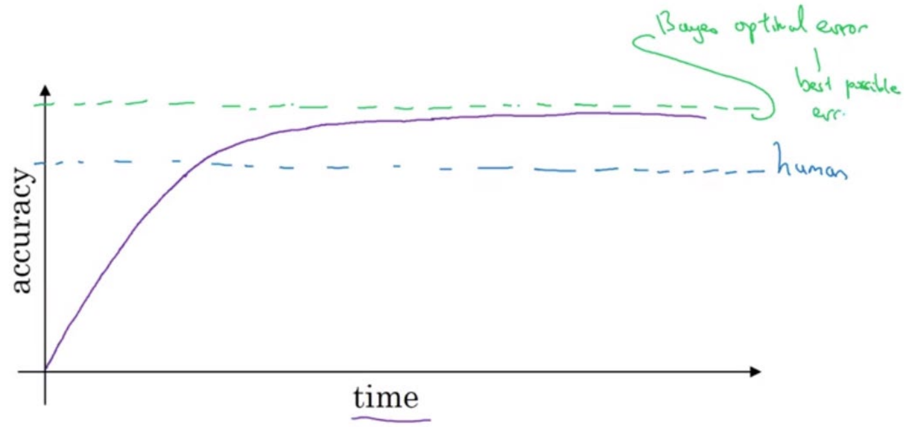
- Fit training set well on cost function
- Fit dev set well on cost function (**tuning**)
- Fit test set well on cost function
- Performs well in the real world

PS: Dev set and test set must come from the same distribution

- ❑ Old way of splitting data → train 60% test set 20% dev set 20%
- ❑ New way of splitting day → train 98% test set 1% dev set 1%

PS: If doing well on your metric + dev/test does not correspond to doing well on your application, change your metric and/or dev/test set

HUMAN-LEVEL PERFORMANCE



Avoidable bias: Human-level error – training error

- Train bigger model
- Train longer/better optimization algorithm (RMSprop, Momentum, Adam)
- NN architecture/hyperparameters search (RNN, CNN)

Variance: Dev error – training error

- More data
- Regularization (L_2, Dropout, Data augmentation)
- NN architecture/hyperparameter search

Evaluation metrics:

- Harmonic mean: F Score: $\frac{2}{\frac{1}{P}+\frac{1}{R}}$
- Average
- With bigger weights for 'big mistakes': $\text{Error} = \frac{1}{\sum_i w^{(i)}} \sum_i w^{(i)} cost_i \,;\, w^{(i)} = \begin{cases} 1, if\ x^{(i)} is\ normal \\ Big\ weight, if\ x^{(i)} is\ 'big\ mistake' \end{cases}$

Human-level error = Lowest human error (Closer to bias error)

PS: If training error < human-level error → We can't predict bias error

PS: Humans are good in natural perception cases: NLP, speech recognition, image recognition…

- ❑ Max metric → **Optimization** metric
- ❑ Good value corresponding to a threshold → **Satisfying** metric

ERROR ANALYSIS

<u>Error analysis</u>: Examine mistakes **manually**

- Get ~100 mislabeled dev set examples
- Count up how many for each category (add a category for incorrectly labeled examples)
  - <u>Ceiling in performance</u>: How well would working on a sub problem help you?

<u>Incorrectly labeled examples</u>:

- **Random errors**: DL algorithms are quite robust to them
- **Systematic errors**: They are <u>dangerous</u>

	General data	Specified data
Human-level	Human-level	-
Error on examples trained on	Training error	-
Error on examples not trained on	Training-dev error	Dev/Test error

Avoidable bias

Variance

Data mismatch

<u>Guideline</u>:

1. Set up dev/test set and metric
2. Build initial system quickly
3. Use bias/variance analysis and error analysis to prioritize next steps

<u>If training set and dev/test set from ≠ distributions</u>:

- **Option 1**: Add new data to whole date → Shuffle → Divide data on train, dev and test
- **Option 2**: Add part of new data to training-dev set and the rest to dev and test
  - Training-dev set: Same distribution as training set, but not used for training

<u>PS</u>: Test error – Dev error = Degree of overfitting to the dev set

OTHER NOTIONS

<u>Addressing data mismatch</u>:

- Carry out manual error analysis to try to understand difference between training and dev/test sets
- Make training data more similar, or collect more data similar dev/test sets

<u>Artificial data synthesis</u>: Normal data set + Artificial effect = Synthesized data set → Create more data

PS: Be careful of overfitting to the 'artificial effect'

<u>Transfer learning</u>: Learned task A → Learn task B by modifying output layers

- Task A and task B must have the same input x
- Task A have more data than tastk B
- Low level features from A could be helpful for learning B (similar features)

Eg. Image recognition → Radiology diagnosis, Speech recognition → Wakeword detection

<u>Multi-task learning</u>: Learn multiple tasks at the same time with the same input layers

- Training on a set of tasks could benefit from having shared lower-level features
- Amount of data you have for each task is quite similar usually
- Can train a big enough neural network to do well on all the tasks

<u>End-to-end learning</u>: Learn the function **mapping** from input to output **directly** (passing the intermediate steps)

- **Pros**: Let the data speak + Less hand-designing of components needed
- **Cons**: May need large amount of data + Excludes potentially useful hand-designed components

<u>Traditional pipe line</u>: Learn the ≠ intermediate functions from input to output