2025-08-26T04-13-38
Tags: DLS

DLS C3 week 2

Carrying out error analysis

When:

- get a learning algorithm to do a task that humans can do, and
- learning algorithm is not yet at human-level performance

Error analysis examines mistakes that the algorithm makes to gain insight on what to do next.

Given a cat classifier:

- 90% accuracy and 10% error on the dev set.

It is observed that the algorithm misclassifies dogs as cats—should one start a project focused on the dog problem?

Error analysis procedure for this problem:

1. Get about ~ 100 mislabeled dev set examples.
2. Count how many of them are dogs.

Case 1: 5% of the 100 are dogs.

- Not worth the effort.
- Error could drop from $10\% \to 9.5\%$.

Case 2: 50% of the 100 are dogs.

- Effort is justifiable.
- Error could drop from $10\% \to 5\%$.

Evaluating multiple ideas in parallel:

1. Get a set of mislabeled examples from the dev set.
2. Categorize errors (e.g., dog, great cats, blurry image).
3. Count fraction of errors per category using tables or spreadsheets.

Images	Dog	Great cat	Blurry image	Comment
1	✓			Pitbull
2			✓	
3		✓	✓	Lion at rainy day
⋮	⋮	⋮	⋮	⋮
% of total	8%	43%	61%	

Doing this won't give a rigid formula. But, it shows a promising direction and displays ceilings of each error.

Cleaning up incorrectly labeled data

Mislabeled examples	• Algorithm predicts incorrectly.	Incorrectly labeled examples	• The dataset label itself is wrong.

DL algorithms are quite robust to *random* errors in the train set. They are less robust to *systematic* errors.

- Random errors: e.g., accidental label due to human oversight.
- Systematic error: e.g., all white dogs labeled as cats.

Incorrect labels matter more in dev and test sets.

During error analysis, add an extra column to count the number of examples where the label Y was incorrect. Why?

- The classifier might predict correctly, but the dev set label is incorrect.

Three key numbers to decide if it's worthwhile in reducing incorrectly labeled examples:

- overall dev set error.
- errors due to incorrect labels.
- errors due to other causes.

If incorrect labels only represent a tiny fraction, it's not worthwhile.

Otherwise, fix the incorrect labels in the dev set.

- Why? If we're comparing two models A (2.1% error) and B (1.9% error) both with a 0.6% error due to incorrect labels. The fraction due to incorrect labels is significant.

Guidelines for fixing incorrect labels:

- Apply fixes to both dev and test sets to maintain same distribution.
- Check not only incorrect but also correct predictions—to avoid bias.
  - Not always done. It's difficult for high-accuracy models.
- Train set may come a slightly different distribution.
- Dev and test sets must come from the same distribution.

Build your first system quickly, then iterate

Recommended process:

1. Set up dev/test set and metrics.
2. Build the first system quickly.
3. Perform bias/variance analysis and error analysis.
4. Use insights to prioritize next steps.

Training and testing on different distributions

DL algorithms require a large amount of labeled training data.

Training data from a different distribution than the dev and test sets are now seen more often.

Example 1

Let's look at a hypothetical mobile app that detects images that contains cats with two data sources:

- target distribution — data from the mobile app (lower in quality).
- different distribution — data from web (high quality).

Given: 200,000 web images and 5,000 mobile app images.
Goal: the system must perform well on the target distribution.

Recommended split:

- train set — 200,000 web images and 5,000 mobile app images
- dev/test set — 5,000 mobile app images (2,500 each)

Advantage: dev/test set reflect target distribution, optimized for real use case.

Disadvantages: train set distribution is not equal to dev/test set distribution.

Example 2

A hypothetical speech-activated rearview mirror where users talk to it to get directions.

Different distribution: 500,000 utterances from the ff. data

- purchased data
- existing data from previous speech-related projects

Target distribution: 20,000 utterances

- speech activated rearview mirror data

Splitting strategies:

Option A	• train set — 500,000 from different • dev/test set — 20,000 from target (10,000 each)	Option B	• train set — 500,000 from different and 10,000 from target • dev/test set — 10,000 from target (5,000 each)

Advantages of Option B:

- larger train set.
- test/dev distribution relevance.

Bias and variance with mismatched data distributions

When train set comes from a different distribution than the dev/test sets the analysis of bias and variance changes.

A hypothetical cat classifier with the ff. error:

- Train error — 1%
- Dev error — 10%

Train set comes from a different distribution than the dev set.

If dev set came from the same distribution as train set: there's high variance.

But when it's not, there are two possibilities:

- the algo saw data in train set but not in dev set.
- the data distribution in the dev set is different.

Thus, it is difficult to know the source of the 9% increase in error.

- How much of it is because the algo didn't see data in dev set? (variance)
- How much of it is because the dev set data is just different?

To unravel the two effects: define a new category of data, the training-dev set.

- Training-dev set — same distribution as train set but excluded for training, only for evaluation.

New split:

- train set
- train-dev set
- dev set
- test set

Example 1

- train error: 1%
- train-dev error: 9%
- dev error: 10%

Interpretation: Large gap between train and training-dev means it's a variance problem.

Example 2

- train error: 1%
- train-dev error: 1.5%
- dev error: 10%

Interpreation:

- small gap between train and train-dev, means low variance.
- large gap between train-dev and dev, means data mismatch.

Example 3

- train error: 10%
- train-dev error: 11%
- dev error: 12%

Interpretation: high train error, means high bias problem (avoidable bias).

Example 4

- train error: 10%
- train-dev error: 11%
- dev error: 20%

Interpretation:

- high bias.
- data mismatch.

General analysis

Key quantities:

- human-level error (as a proxy to Bayes error).
- train error.
- train-dev error.
- dev error.
- test error.

Huge gap interpretations:

- human-level and train: avoidable bias
- train and trian-dev: high variance.
- train-dev and dev: data mismatch.
- dev and test: overfitting to dev set.

To fix overffiting to dev set: get a larger dev set.

Example 5

- human-level error: 4%
- train error: 7%
- train-dev error: 10%
- dev error: 6%
- test error: 6%

Interpretation: dev set is easier than train set.

More general formulation

Using example no. 5, we make use of a more general formulation using tables.

- x-axis: data sources.
- y-axis: error types.

Error type	General speech recognition	Rearview mirror speech data
Human-level	4%	6%
Trained examples	7% (train)	6%
Not trained examples	10% (train-dev)	6% (dev/test)

Interpretations:

- bias.
- variance.
- data mismatch.
- target distribution is harder for the rearview mirror speech data.

Addressing data mismatch

Unlike bias and variance, there is no systematic solution for fixing data mismatch.

What to do when there's data mismatch?

1. Carry out error analysis to find difference between train-dev and dev error.
2. Make or collect train data similar to dev set data.

Artificial data synthesis

Artificial data synthesis make train data closer to dev set data without collecting large amounts of real-world data.

Example 1

Synthesizing noisy audio:

1. Start with clean speech recording.
2. Record car background noise.
3. Mix speech with car noise to simulate in-car speech.

Be careful, artificial data synthesis makes data prone to overfitting.

- If there's 10,000 hours of clean speech and only 1 hour of car noise, repeating that 1 hour of car noise creates bias.
- Even if it sounds fine to humans, the neural network may overfit to that single noise.

What to do to prevent it?

- Collect diverse noise samples.
- Use 10,000 hours of unique noise for better generalization. It's possible but not always feasible.

Example 2

Synthesizing car images using computer graphics is a way to do artificial data synthesis.

If your dataset includes only 20 car models from a game, it may look fine to humans.

But the real world has more than 20 unique car designs. Therefore, only having 20 will probably overfit the data which leads to poor real world performance

Key point: when doing artificial data synthesis, ensure a wide variation of data.

Transfer learning

Transfer learning means taking knowledge (or a part of it) from a neural network trained on a task (Task A) and applying it to a seperate task (Task B).

For example:

- Train on image recognition of cats.
- Use that (or a part of it) to improve x-rays scans.

Transfer learning procedure:

- Train a neural network on (x, y) pairs on the original task (called pre-training).
- Delete the last output layer including its weights.
- Create a new output layer.
- Randomly initialized weights for the new output layer.
- Swap in a new dataset (x, y) for the new task.
- Retrain on new dataset.

Two strategies for retraining:

- Only small dataset avaiable — only retrain the weights of last layer $w^{[L]}, b^{[L]}$.
- Large dataset available — retrain all of the layers (fine-turning).

When does Transfer Learning makes sense:

- Much more data for the original task (Task A) than the new task (Task B).
- Both tasks share similar input types.

When does Transfer Learning makes DONT sense:

- Much less data for the original task (Task A) than the new task (Task B).

Key Conditions for Transfer Learning:

- Same input type (e.g., images to images, audio to audio).
- A lot more data for Task A than Task B.
- Low-level features from Task A are useful for Task B.

Multi-task learning

Multi-task learning is simultaneous process where one neural network learns several tasks at the same time.

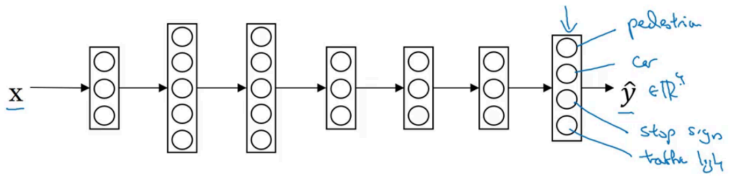
Say we're building an autonomous self-driving car may need to detect several objects.

Also, consider it's only four objects: pedestrians, stop signs, traffic lights, and other cars.

Instead of a single label $y^{(i)}$, we now have 4 labels. The output layer is now a vector of shape $(4, 1)$.

- For m examples Y is now a $4 \times m$ matrix.

$$Y = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$



Given a $\hat{y}^{(i)}$ of shape $(4, 1)$, the loss function is:

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

where $\mathcal{L}$ is the logistic loss:

$$\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)}) = -y_j^{(i)}\log(\hat{y}_j^{(i)}) - (1 - y_j^{(i)})\log(1 - \hat{y}_j^{(i)})$$

If you train a neural network to minimize this cost function, you are carrying out multi-task learning.

Softmax regression	• assigns one label to single example.	Multi-task learning	• one image can have multiple labels.

Why is it called multi-task learning: it assigns one image to multiple labels.

Using multi-task for this example, a single neural network solves 4 problems:

- detect pedestrian.
- detect car.
- detect stop sign.
- detect traffic light.

Advantage of multi-task learning to seperate learning:

- If earlier features in a neural network can be shared between different types of objet, training a single neural network to do m things results in better performance than training four completely seperate neural networks to do four tasks seperately.

The reality is not every image has all labels annotated. During image do the ff. to fix it:

- Sum over only available labels.

When does multi-task learning make sense?

- Training on a set of tasks that could benefit from having shared lower-level features.
- Amount of data for each task is quite similar (optional but usual).
- Can train a big enougn neural network to do well on all the tasks.

Let's say we have a 100 tasks with 1,000 examples each:

- At task 100, multi-task learning leverages the 99,000 examples from other tasks.

Multi-task learning is used less often than transfer learning except for computer vision.

- It is hard to find many related tasks to train simultaneously.

End-to-end deep learning

End-to-end deep learning replaces multiple processing stage in a system with a single neural network.

Traditional systems involve multiple stages of feature extraction and processing. On the other hand, end-to-end learning takes a raw input X and maps it directly to Y.

Challenges to end-to-end learning:

- End-to-end learning requires a very large dataset.

Which has the best performance for every dataset size:

- small dataset — traditional learning.
- medium dataset — hybrid approach (partial end-to-end).
- large dataset — end-to-end learning.

Example 1

Say we need a system that recognizes employees using a camera:

Naive approach	• map raw image X to directly to identity Y.	Multi-step approach	1. face detection. 2. face recognition.

The multi-step approach is better for this case:

- Each sub-task is simpler than a full end-to-end solution.
- Data is available for each sub-task but not for the end-to-end solution.

If a huge end-to-end dataset for this case exists, it might outperform the multi-step approach.

Example 2

Machine translation of a human language.

The traditional pipeline consists of text analysis, feature extraction, and translation rules.

While an end-to-end implementation is feasible due to large datasets of English-French sentence pairs. Therefore, a neural network can learn direct mapping from English to French.

Example 3

X-ray age estimation is used by Pediatricians to estimate if a child is developing normally.

The traditional approach consists of:

1. Segment and identify bone structures in X-ray.
2. Compare bone lengths to growth charts to estimate age.

End-to-end approach directly predicts age from image. But if there is not enough labeled X-ray images, it would lead to poor performance.

The advantage for the multi-step approach are:

- requires less data.
- hand-engineered components works well with small datasets.

When to use end-to-end deep learning

Benefits of end-to-end deep learning

- The data speaks for itself—free from human bias and perception.
- No need for manual creation of features.
- Simplified workflow and development.

Disadvantages of end-to-end deep learning:

- Requires a very large amount of data.
- Excludes potentially useful hand-designed components.

Hand-designing is a double-edged sword:

- Well-designed components may improve accuracy significantly on small datasets.
- If components impose poor assumptions, it can constrain performance.

If end-to-end learning is not feasible:

- use deep learning for specific components where data is abundant.
- use traditional algorithms for tasks with optimal solutions.

Don't default to end-to-end deep learning, consider the ff. first:

- data availability.
- complexity of the mapping.
- modular design advantages.

Which to use?

End-to-end approach	• you have a very large dataset. • mapping is extremely complex. • manual feature design is limiting.	Modular approach	• data is limited. • intermediate tasks have abundant labeled data.