1. Suppose you are asked to build a model to determine whether a satellite image of an area has low, medium, or high population density. Your labelled dataset contains 10M images with 3 highly unbalanced classes (low, medium, and high). How would you choose your train/dev/test split (or cross validation scheme)? What do you think is the best approach to handling the class imbalance problem?

Assuming we cannot collect more data of the underrepresented class, firstly, we have to *understand our data*. we should do some data exploration and then data selection. *Are the samples useful? Quality is better than quantity*. We should inspect the images of the overrepresented class and see if a lot of them are highly similar to one another (which is highly probable when collecting satellite images). NVidia's End-to-End Deep Learning for Self-Driving Cars paper said that they sampled video images at 10 FPS because a higher sampling rate would include highly similar images which won't provide much additional useful information. There are also some cases (especially if the images are automatically labeled) such that the image actually doesn't contain any useful features that could classify it correctly (e.g too blurry or taken at a rare occasion or basically outliers). We can do this by hand, or use anomaly detection algorithms to detect outliers. *We have to ask ourselves, what are the actual features on a satellite image that indicate a type of population density of an area?* Removing samples from a majority class is also called undersampling and one interesting family include distance-based and cluster-based undersampling methods, which attempts to systematically select the best representative samples from the majority class. We can check the current popular and effective methods to weed out unnecessary (and sometimes harmful) data.

In actuality, synthetic data oversampling (SMOTE/ ADASYN / CBOS based algorithms) seems to be the more popular way to handle the imbalance problem compared to undersampling. The idea behind synthetic data oversampling is to generate artificial samples based on the existing "real" minority class samples and add them to the training set. Synthetic data generation algorithms range from interpolating neighboring data points, preprocessing to add small randomness (data augmentations), and clustering (Pretty cool topic by itself!). A recent paper about CNN studied the imbalance problem on CNNs and concluded that many synthetic oversampling techniques is very good at addressing the problem and they don't lead to overfitting and that many undersampling methods actually performed poorly. This is relevant to us since it is state-of-the-art to use CNNs for image classification tasks.

Classically, there are also "easy" approaches that are said to work well in practice like adjusting the weight contribution to the loss value of an instance (more frequent the class the less the weight of the loss) or weighted random sampling when stochastically training in mini-batches.

In summary, I think we should inspect the data from the majority class and remove those that aren't helpful and also smartly generate synthetic data to add to the minority class so that the training set becomes more balanced before training. I would choose the size of the training data to that of "medium" size. 10%-25% of that would be for validation. The rest is for testing.

Some references:

[1] A systematic study of the class imbalance problem in convolutional neural networks, Mateusz Buda, 13 Oct 2018 1710.05381
[2] Clustering and Learning from Imbalanced Data, Naman D. Singh, Abhinav Dhall, 12 Nov 2018 1811.00972

2. Suppose you've chosen a neural network to classify the images. After training your system, you get the following accuracy scores: - Train set accuracy: 80% - Dev set accuracy: 40% What problems can you identify, and what would be your next steps?

There are many unknowns with this question and there are many possible reasons why this occurs. The first question that comes to mind is how many samples are there in our data set? Can we collect more of them? I would also check the precision and recall metrics to gain some insight. I think it is both kind of “semi-overfitting” and “semi-underfitting” in the vaguest sense of the words. I say this because the training accuracy isn’t as high as I want and the dev set accuracy is really horrible / unacceptable. It doesn’t capture the underlying trend of the data, that’s why it’s “semi-underfitting”. It’s “semi-overfitting” because the difference between the training accuracy and the dev set accuracy is really apparent.

When classifying images (those taken by a camera at least) we usually use state-of-the-art pretrained models and apply transfer learning (having untrainable parameters and only train a few layers). If we use transfer learning, then this means it’s probably not the neural architecture causing the problem. I’m assuming that we’re transfer learning with best practices, if not, then it’s the first thing we should do. If it’s still behaving this way then the problem is either the data or the training or both and I’m leaning towards the data.

The next thing is I would to inspect the data. As they say, *debug your data before you debug your model.* My first hunch is that the samples in the training set is dramatically different from the samples in the dev set. CNN learns patterns. You shouldn’t expect your classifier to correctly identify something when every other thing it has seen in that class is dramatically different from this newly seen data. What are the differences between the training samples and the dev samples? How many samples are in the training set compared to the dev set? Rule of thumb says given the training data set 10 - 25% should be used for validation. And the set should be balanced (almost equal representation for each class in both sets). If you trained 90% dogs and 10% cats, and your dev set is 100% cats then such results are expected. Is the data processing the same before feeding into the network (e.g same scale? Referring to the same color channels?)

If the samples in the are balanced and a reasonable size and also that the training set and devset are similar in a degree that we are comfortable with, then we look at the entire dataset. Depending on the inspection of the data and the size of the dataset, maybe we can use classical approaches like data augmentation (randomly rotating / adding noise / zooming).

If the dataset is fine, then we proceed to examining the training. We can also look at hyperparameters. Did we use dropouts and/or regularization when training? How about weights initialization? The optimizer algorithm did we use? Maybe we converged to a local minimum. We should use a random seed for weight initialization to be able to replicate the results and also experiment on optimizers. Also batch size, adaptive learning rate, and epochs if applicable. If manually tuning is not enough, I’d try grid search. I’d also plot the accuracy/loss charts per epoch for training and validation set and see if early stopping would help. Maybe try cross validation techniques as well.

Another thing I would do it perform layer visualization with libraries like KERAS VIS and KERAS GRADCAM to understand what the neural network is seeing or paying attention to.

In summary, I’m assuming I used transfer learning with existing state-of-the art-model, I’d inspect the datasets and modify the dataset if necessary as I see fit. If trained model is still bad I’d experiment on tweaking training parameters and do layer visualization to help improve the model.

3. Suppose the owner of a well-known restaurant asks you to create a model that can classify customer reviews as positive or negative. He has given you a sample of 1500 client reviews labelled accordingly (75% positive, 25% negative). What would be your step-by-step approach to solving this problem? Be specific!

The first thing that comes to mind is that the dataset given by the owner is really small and also imbalanced. Having a small dataset has some benefits as well, as it is easier to manually inspect the data, the number of total words might be even less than a book! For some perspective, last updated on Feb 2018, currently there are about 5.2M user reviews from about 170k businesses in the Yelp dataset Kaggle and Yelp confirms that Restaurants the category with the most total reviews (2018). An interesting insight from Yelp says that longer reviews (more words) are more likely to be negative as they tend to tell there specific unhappy story in detail. We can verify if this applies to our dataset later.

I am assuming that the reviews are in all English. I also have a hypothesis that restaurant reviews are "transferable" in the sense that given the (English) sentences in the review, the actual restaurant being reviewed isn't that relevant in determining the sentiment of the review. This is backed by the Yelp saying That the average rating wherever they're eating (state in America) is pretty consistent. Of course this hypothesis should also be tested.

In a high-level overview, I'll be doing three things, (1) Exploring the data (2) Reading previous related work code, and tools (3) Experimenting on different types of preprocessing, feature selection and training models

Exploring the data
As usual, the first thing we should do is to inspect the dataset. I'll separate the negative reviews from the positive ones (2 folders/ directory) And I'll put each review on a separate text file. I'll check if my hypothesis is correct that negative reviews tend to have more words than positive ones. I'll also check the distribution of the number of words per dataset. I'll generate a two word cloud to gain a glimpse of what kind of words appear on each type of reviews. In this stage, some preprocessing is necessary like making everything lowercase, removing words that don't add useful information like pronouns and "a, an, the", "correcting spelling" etcetera which can be done with open source libraries like Textblob. I'll check how many types of words there is on the dataset (After correcting the spelling and other relevant preprocessing). I'd also rank the "meaningful" words based on frequency. These are basic things to understand the kind of data that we're dealing with before building our model.

Reading previous related work, code and tools
The next thing I would do is to read previous work related to this problem. I have scanned a view research papers, code, and articles that attempt to solve this problem, and interestingly most of the models are "shallow" less than five layers after preprocessing and embedding layer, the next layers include convolutions and or RNNs/LSTMs. This makes sense because LSTMs/RNNs are very good at sequence modelling and reviews are a sequence of words. 1d-convolutions is also quite a good choice as it "slides" across phrases of words.

One of the first things I would do is use Textblob (Open-source Library) and Google's Cloud Sentence Sentiment API (which is sadly not open source) for each review. Essentially for Google Cloud Sentence API, you'd feed a raw text file (which I said I'd do in the first section) analyze each sentence (Give a score for each sentence) and give a score for example, taken from the documentation:

```
python sentiment-analysis.py reviews/bladerunner-pos.txt
Sentence 0 has a sentiment score of 0.8
Sentence 1 has a sentiment score of 0.9
</snip>
Sentence 9 has a sentiment score of 0.9
Overall Sentiment: score of 0.5 with magnitude of 5.5
The above example would indicate a review that was relatively positive (score of 0.5), and relatively
emotional (magnitude of 5.5).
python sentiment-analysis.py reviews/bladerunner-neg.txt
</snip>
Overall Sentiment: score of -0.6 with magnitude of 3.3
```

Having only 1500 reviews, we could actually gain more insight from this and check whether the analysis of Google Sentiment API actually supports the labels for each review. We can inspect the reviews that have contracting labels and see if they were mislabeled or misclassified and possible reasons why (Maybe it's too neutral?).. We can also check how many sentences are emotionally polarizing as opposed to sentences that are just neutral. Textblob also has the same functionality, and (it's opensource!), we can check if it performs almost as well or better than Google's API and actually check the source code of how it works.

Experimenting on different types of preprocessing, feature selection, and training models
After inspecting state-of-the-art sentiment analysis models and how it performs in our data set, reading papers and codes (See References below) We'll finally create our model. Let's start with the two most common models we see in the wild (LSTM and CNN based models) and also maybe actually copy-paste (after understanding) the relevant code used in Textblob's sentence sentiment analysis and start from there.
A recent useful project related to this is in the repository of cmasch (Dec 2018). As I've mentioned earlier, the standard practice for preprocessing is to make them all lowercase, remove 'stop words' (a, and, the etc), and pronouns, because those shouldn't matter, as well as to correct mispellings or 99% the same (IE new_york / new york, newyork, NY, NewYork), remove punctuations etc. But based on cmasch (Dec 2018) experiments s/he says that "*It seems that cleaning the text by removing stopwords, numerical values and punctuation remove important features too.*" So we'll keep that in mind. Embeddings is a compact way of representing words to be fed in our system, such that similar words should theoretically be closer to each other. Some open-source pretrained embedding layers [GloVe or Word2Vec] exist which we can use as a starting point. So we can start with that, and maybe retrain them if they don't work so well. This third step is essentially tweaking existing already well-experimented models that are related to the problem that we are trying to solve.

In summary, as I've said before I'll be doing three things, (1) Exploring the data (2) Reading previous related work code, and tools (3) Experimenting on different types of preprocessing, feature selection and training models.

References

[Paper] Deep Bidirectional LSTM model with Attention, based on the work of Baziotis et al., (2017)
[Paper] Yoon Kim: Convolutional Neural Networks for Sentence Classification (2014) arXiv:1408.5882

- Model variants: cmasch (Dec 2018) and Dennybritz (Code)
- [Paper] Zhe Yang: Effects of hyperparameters on CNN Layer of this model (2016) arXiv:1510.03820

Bentrevett: PyTorch Sentiment Analysis with RNN / LSTM (Code) (2019)

Written by MITHI SEVILLA, APRIL 19, 2019