COMS4705: Practical Tips for Final Projects Notes

Last updated on September 23, 2025

Adapted from Stanford CS224N Natural Language Processing with Deep Learning

Contents

1 Introduction

These notes discuss *Practical Tips for Final Projects*. In particular this document contains lots of links to useful resources. This document does *not* contain a detailed specification for the project deliverables (proposal, milestone, poster and report) – those specifications will be released separately.

Section 2 contains information on how to choose and formulate a project topic.

Section 3 contains resources and information useful for planning and carrying out your projects.

This document may be updated.

2 Choosing a Project Topic

Project Suitability You can choose any topic related to Deep Learning for NLP. That means your project should make substantive use of deep learning *and* substantive use of human language/code/math data.

Project types Here is a (non-exhaustive!) list of possible project types:

1. Applying an existing neural model to a new task
2. Implementing a complex neural architecture
3. Proposing a new neural model (or a new variation of an existing model)
4. Proposing a new training, optimization, or evaluation scheme
5. Experimental and/or theoretical analysis of neural models

Project ideas from Stanford researchers

2.1 Expectations

Your project should aim to provide some kind of scientific knowledge gain, similar to typical NLP research papers (see Section 2.2 on where to find them). A typical case is that your project will show that your proposed method provides good results on an NLP task you're dealing with.

Given that you only have a few weeks to work on your project, it is *not* necessary that your method beats the state-of-the-art performance, or works better than previous methods. But it should at least show performance broadly expected of the kinds of methods you're using.

In any case, your paper should try to provide reasoning explaining the behaviour of your model. You will need to provide some qualitative analysis, which will be useful for supporting or testing your explanations. This will be particularly important when your method is not working as well as expected.

Ultimately, your project will be graded *holistically*, taking into account many criteria: originality, performance of your methods, complexity of the techniques you used, thoroughness of your evaluation, amount of work put into the project, analysis quality, writeup quality, etc.

2.2 Finding existing research

Generally, it's much easier to define your project (see Section 3.1) if there is existing published research using the same or similar task, dataset, approaches, and/or evaluation metrics. Identifying existing relevant research (and even existing code) will ultimately save you time, as it will provide a blueprint of how you might sensibly approach the project. There are many ways to find relevant research papers:

- You could browse recent publications at any of the top venues where NLP and/or Deep Learning research is published: ACL, EMNLP, TACL, NAACL, EACL, NIPS, ICLR, ICML (Not exhaustive!)
- In particular, publications at many NLP venues are indexed at `http://www.aclweb.org/anthology/`
- Try a keyword search at:
  - `http://arxiv.org/`
  - `http://scholar.google.com`
  - `http://dl.acm.org/`
  - `http://aclasb.dfki.de/`
- Use Google Scholar's "Cited by" feature to surf the citation network of related papers.
- Look at publications from the Stanford NLP group `https://nlp.stanford.edu/pubs/`

2.3 Finding datasets and tasks

There are lots of publicly-available datasets on the web. Here are some useful resources to find datasets:

- HuggingFace has a large repository of public text datasets. `https://huggingface.co/datasets?modality=modality:text&sort=downloads`
- A repository to track progress in NLP, including listings for the datasets and the current state-of-the-art performance for the most common NLP tasks. `https://nlpprogress.com/`
- A small list of well-known standard datasets for common NLP tasks: `https://machinelearningmastery.com/datasets-natural-language-processing/`
- An alphabetical list of free or public domain text datasets: `https://github.com/niderhoff/nlp-datasets`

- Wikipedia has a list of machine learning text datasets, tabulated with useful information such as dataset size.
  `https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research#Text_data`
- Kaggle has many datasets, though some of them are too small for Deep Learning. Try searching for 'nlp'.
  `https://www.kaggle.com/datasets`
- Datahub has lots of datasets, though not all of it is Machine Learning focused.
  `https://datahub.io/collections`
- Microsoft Research has a collection of datasets (look under the 'Dataset directory' tab):
  `http://research.microsoft.com/en-US/projects/data-science-initiative/datasets.aspx`
- A script to search arXiv papers for a keyword, and extract important information such as performance metrics on a task.
  `https://huyenchip.com/2018/10/04/sotawhat.html`
- A collection of links to more collections of links to datasets!
  `http://kevinchai.net/datasets`
- A collection of papers with code on many NLP tasks.
  `https://paperswithcode.com/sota`
- Datasets for machine translation.
  `http://statmt.org`
- Syntactic corpora for many languages.
  `https://universaldependencies.org`

2.3.1 Obtaining access to datasets via Columbia

Columbia has access to select natural language data: The Linguistic Data Consortium (LDC) is a very large repository of data from many languages, including lots of annotated data. To inquire about access to data from the LDC, please reach out to the course staff.

2.3.2 Obtaining datasets from researchers

In some cases, researchers may not make their datasets publicly available, but allow use in specific circumstances. Typically, steps to obtain access are included somewhere in the paper or associated code repository. It can be difficult to determine how long it will take to access such data, so you should incorporate this time in your plan and search for suitable alternatives. Please contact the course staff if you need assistance in requesting data from researchers.

2.4 Collecting your own data

It is possible to collect your own data for your project. However, data collection is often a time-consuming and messy process that is more difficult than it appears. Given the limited timeframe, we generally don't recommend collecting your own data. If you really do want to collect your own data, make sure to budget the data collection time into your project. Remember, your project must have a substantial Deep Learning component, so if you spend all your time on data collection and none on building neural models, we can't give you a good grade.

2.5 Accessing pretrained models

Your project may benefit from access to pretrained models. The largest repository of pretrained models is on HuggingFace: `https://huggingface.co/models`. For instructions on how download, run and finetune pretrained models, see the HuggingFace documentation: `https://huggingface.co/docs/transformers/index`. Kaggle also has a repository of models here: `https://www.kaggle.com/models`.

3 Project Advice

3.1 Define your goals

At the very beginning of your project, it's important to clearly define your goals in your mind and make sure everyone in your team understands them. In particular:

- Clearly define the task. What's the input and what's the output? Can you give an example? If the task can't be framed as input and output, what exactly are you trying to achieve?

- What dataset(s) will you use? Is that dataset already organized into the input and output sections described above? If not, what's your plan to obtain the data in the format that you need?

- What is your evaluation metric (or metrics)? This needs to be a well-defined, numerical metric (e.g. ROUGE score), not a vague idea (e.g. 'summary quality'). See section 3.6 for more detail on how to evaluate your methods.

- What does success look like for your project? For your chosen evaluation metrics, what numbers represent expected performance, based on previous research? If you're doing an analysis or theoretical project, define your hypothesis and figure out how your experiments will confirm or negate your hypothesis.

3.2 Processing data

When you have downloaded or collected your data, it is often beneficial to manually examine samples to gain an intuitive understanding of the task. This may also reveal unforseen inconsistencies or other issues with your data that you may wish to mitigate when designing your experiments (e.g., typos, odd formatting). You may need to do (additional) processing of your data (e.g. tokenization, tagging, or parsing). Here are some tools that may be useful:

- StanfordNLP: a Python library providing tokenization, tagging, parsing, and other capabilities. Covers 53 languages.
  `https://stanfordnlp.github.io/stanfordnlp/`

- Software from the Stanford NLP group:
  `http://nlp.stanford.edu/software/index.shtml`

- NLTK, a lightweight Natural Language Toolkit package in Python:
  `http://nltk.org/`

- spaCy, another Python package that can do preprocessing, but also includes neural models (e.g. Language Models):
  `https://spacy.io/`

3.3 Data hygiene

At the beginning of your project, split your data set into **training data** (most of your data), **development data** (also known as *validation data*) and **test data**. A typical train/dev/test split might be 90/5/5 percent (assigned randomly). Many NLP datasets come with predefined splits, and if you want to compare against existing work on the same dataset, you should use the same split as used in that work. Here is how you should use these data splits in your project:

1. **Training data**: Use this (and only this data!) to optimize the parameters of your neural model.

2. **Development data**: This has two main uses. The first is to compare the performance of your different models (or versions of the same model) by computing the evaluation metric on the development data. This enables you to choose the best hyperparameters and/or architectural choices that should be evaluated on the test data. The second important usage of development data is to decide when to stop training your model. Two simple and common methods for deciding when to stop training are:

   (a) Every epoch (or every N training iterations, where N is predefined), record performance of the current model on the development set and store the current model as a checkpoint. If development performance is worse than on the last previous iteration (alternatively, if it fails to beat best performance M times in a row, where M is predefined), stop training and keep the best checkpoint.

   (b) Train for E epochs (where E is some predefined number) and, after each epoch, record performance of the current model on the development set and store the current model as a checkpoint. Once the E epochs are finished, stop training and keep the best checkpoint.

3. **Test data**: At the end of your project, evaluate your best trained model(s) on the test data to compute your final performance metric. To be scientifically honest, you should *only* use the training and development data to select which models to evaluate on the test set.

The reason we use data splits is to avoid *overfitting*. If you simply selected the model that does best on your training set, then you wouldn't know how well your model would perform on *new* samples of data – you'd be overfitting to the training set. In NLP, powerful neural models are particularly prone to overfitting to their training data, so this is especially important.

Similarly, if you look at the test set before you've chosen your final architecture and hyperparameters, that might impact your decisions and lead your project to overfit to the test data. Thus, in the interest of science, it is extremely important that you don't touch the test set until the very end of your project. This will ensure that the quantitative performance that you report will be an honest unbiased estimate of how your method will do on new samples of data.

It's even possible to overfit to the development set. If we train many different model variants, and only keep the hyperparameters and architectures that perform best on the development set, then we may be overfitting to the development set. To fix this, you can even use two separate development sets (one of them called the **tuning set**, the other one the **development set**). The tuning set is used for optimizing hyperparameters, the development set for measuring overall progress. If you optimize your hyperparameters a lot and do many iterations, you may even want to create multiple distinct development sets (dev, dev2, ...) to avoid overfitting.

3.4 Build strong baselines

A baseline is a simpler method to compare your more complex neural system against. Baselines are important so that we can understand the performance of our systems in context.

For example, suppose you're building a multilayer LSTM-based network with attention to do binary sentiment analysis (classification of sentences as positive or negative). The simplest baseline is the guessing baseline, which would achieve 50% accuracy (assuming the dataset is 50% positive and 50% negative). A more complex baseline would be a simple non-neural Machine Learning algorithm, such as a Naive Bayes classifier. You could also have simple neural baselines – for example, encoding the sentence using an average of word embeddings. Lastly, you should compare against simpler versions of your full model, such as a vanilla RNN version, a single-layer version, or a version without attention.

These last few options are also called *ablation experiments*. An ablation experiment removes some part of the full model and measures the performance – this is useful to quantify how much different parts of the network help performance. Ablation experiments are an excellent way analyze your model.

Building strong baselines is very important. Too often, researchers and practitioners fall into the trap of making baselines that are too weak, or failing to define any baselines at all. In this case, we cannot tell whether our complex neural systems are adding any value at all. Sometimes, strong baselines perform much better than you expected, and this is important to know.

3.5 Training and debugging neural models

Unfortunately, neural networks are notoriously difficult to debug. However, here are some tips:

- Because training neural models often involves some element of randomness (e.g., data ordering and sampling), you should fix the seed for common libraries (i.e., `transformers`, `numpy`, `random`). This ensures that when you are able to replicate any problems that you encounter when you rerun experiments, but also that someone else is able to replicate you final results. Note that you should not change your seed to achieve different results, and sometimes it is appropriate to select multiple random seeds.

- To debug a neural model, train on a small toy dataset (e.g., small fraction of training data, or a hand-created toy dataset) to sanity-check and diagnose bugs. For example, if your model is unable to overfit (e.g. achieve near-zero training loss) on this toy dataset, then you probably have a bug in your implementation.
- Hyperparameters (e.g. learning rate, number of layers, dropout rate, etc.) often impact results significantly. Use performance on the development set to tune these parameters (see Section 3.3). Though you probably won't have time for a very exhaustive hyperparameter search, try to identify the most sensitive/important hyperparameters and tune those.
- Due to their power, neural networks overfit easily. Regularization (dropout, weight decay) and stopping criteria based on the development set (e.g. early stopping, see Section 3.3) are extremely important for ensuring your model will do well on new unseen data samples.
- A more complicated neural model can 'fail silently': It may get decent performance by relying on the simple parts, when the really interesting components (e.g. a cool attention mechanism) fail due to a bug. Use ablation experiments (see Section 3.4) to diagnose whether different parts of the model are adding value.
- During training, randomize the order of training samples, or at least remove obvious ordering (e.g., don't train your Seq2Seq system on data by going through the corpus in the order of sentence length – instead, make sure the lengths of the sentences in subsequent minibatches are uncorrelated). SGD relies on the assumption that the samples come in random order.
- There are many online resources containing practical advice for building neural network models. Note, most of this type of advice is based more on personal experience than rigorous theory, and these ideas evolve over time – so take it with a grain of salt! After all, your project could open new perspectives by disproving some commonly-held belief.
  - A Twitter thread on most common neural net mistakes (June 2018): `https://twitter.com/karpathy/status/1013244313327681536`
  - Deep Learning for NLP Best Practices blog post (July 2017): `http://ruder.io/deep-learning-nlp-best-practices/`
  - Practical Advice for Building Deep Neural Networks (October 2017): `https://pcc.cs.byu.edu/2017/10/02/practical-advice-for-building-deep-neural-networks/`

3.6 Evaluation

In your project, carrying out *meaningful evaluation* is as important as designing and building your neural models. Meaningful evaluation means that you should carefully compare the performance of your methods using appropriate evaluation metrics.

Choosing evaluation metrics You must have *at least one evaluation metric* (which should be a numerical metric that can be automatically computed) to measure the performance of your methods. If there is existing published work on the same dataset and/or task, you should use the same metric(s) as that work (though you can evaluate on additional metrics if you think it's useful).

Human evaluation Human evaluation is often necessary in research areas that lack good, comprehensive automatic evaluation metrics (e.g. some natural language generation tasks). If you want to use human judgment as an evaluation metric, you are welcome to do so (though you may find it difficult to find the time and/or funding to collect many human evaluations). Collecting a small number of human judgments could be a valuable addition to your project, but *you must have at least one automatic evaluation metric* – even if it is an imperfect metric.

What to compare You should use your evaluation metric(s) to (a) compare your model against previous work, (b) compare your model against your baselines (see Section 3.4), and (c) compare different versions of your model. When comparing against previous work, make sure to get the details right – for example, did the previous work compute the BLEU metric in a case-sensitive or case-insensitive way? If you calculate your evaluation metric differently to previous work, the numbers are not comparable!

Statistical significance You are not expected to conduct hypothesis tests or check for statistical significance, but you should ensure that your experiments are designed to sufficiently support the claims you make. In particular, you should ensure that data sets you are using for comparisons (e.g., your test set or any specialized subsets) are large enough to support the conclusions you make. For example, 10-50 test samples may not sufficiently support an argument about differences in performance if the gap is small and performance is highly variable. Additionally, randomization is an important consideration for ensuring that your data and results are representative of the broader use-cases you are interested in. At minimum, unless already divided by an earlier work, data splits (i.e., train/test/development) should be random. If feasible, you might also, for example, consider averaging results across multiple random seeds/initializations of a model, multiple paraphrases of a prompt, or multiple sampled model generations depending on your particular aims.

Statistical tests can help you better understand and support your findings. For example, in comparing the mean performance of two models, you may run a two sample t-test or for continuous output variables, you may assess the significance of the correlation between two variables. Alternatively, it may be appropriate to include confidence intervals (90%/95%/99% are typical) in bar and line plots that you generate.

If you choose to do so, many python libraries are available to facilitate randomization in your experiments and conducting statistical tests like those mentioned above:

- `scikit-learn`: `https://scikit-learn.org/stable/`
- `scipy`: `https://scipy.org/`
- `statsmodels`: `https://www.statsmodels.org/stable/index.html`

Qualitative evaluation So far this section has discussed *quantitative evaluation* – numerical performance measures. *Qualitative evaluation*, or *analysis*, seeks to understand your system (how it works, when it succeeds and when it fails) by measuring or inspecting key characteristics or outputs of your model. You will be expected to include some qualitative evaluation in your final report. Here are some types of qualitative evaluation:

- A simple kind of qualitative evaluation is to include some examples (e.g. input and model output) in your report. However, don't just provide random examples without comment – find interesting examples that support your paper's overall arguments, and comment on them.[1]
- Error analysis (as seen in some of the assignments) is another important type of qualitative evaluation. Try to identify categories of errors.
- Break down the performance metric by some criteria. For example, if you think a translation model is especially bad at translating long sentences, show that by plotting the BLEU score as a function of source sentence length.
- Compare the performance of two systems beyond the single evaluation metric number. For example, what examples does your model get right that the baseline gets wrong, and vice versa? Can these examples be characterized by some quality? If so, substantiate that claim by measuring or plotting the quality.
- If your model uses attention, you can create a plot or visualization of the attention distribution to see what the model attended to on particular examples.

If your method is successful, qualitative evaluation is important to understand the reason behind the numbers, and identify areas for improvement. If your method is unsuccessful, qualitative evaluation is even more important to understand what went wrong.

[1]It can be useful to provide a true random sample of model outputs, especially for e.g. natural language generation papers. This gives readers a true, non-cherry-picked qualitative overview of your model's output. If you wish to do this, make it clear that the selection is random, and comment on it.