Not Safe for Work (NSFW) media Classifier for Wikimedia Commons

Project proposal | Slavina Stefanova

Project overview

Wikimedia Commons is an online repository of free-use images, sounds, other media, and JSON files. Anyone can upload media to the Commons portal. The uploads are moderated by members and volunteers of the foundation manually. This project aims to build a classifier that can flag NSFW images/media for review.

Upon successful completion of this internship, the intern would have designed, implemented and tested a machine learning model that would be able to classify image and video media as SFW or NSFW with a high accuracy. They would also be given the chance to deploy the model to WIkimedia test and production servers. Further, they would build a data processing pipeline and an API for the model.

Internship tasks

The project is composed of three tasks, out of which the first one is the main and most important one:

1. **Model Development for NSFW Classifier**
2. API for NSFW Classifier
3. Video Processing Module for NSFW Classifier

Proposed implementation

Model Development for NSFW Classifier

This section contains the following parts:

1. Development environment & frameworks
2. Datasets
3. Model Architecture
4. Performance metrics
5. Developing the model

6. Further improvements

1. Development environment & frameworks

- Local dev environment on my personal Macbook Pro
- git/GitHub for version control
- Jupyter Notebooks for showcasing/sharing work
- Google Colab for model training on GPUs
- Google Drive to host datasets (easy integration with Colab)
- VS Code for more rigorous testing of Python code, including linting. VS Code can also be conveniently set up for remote access to Colab.
- Pytorch + FastAI for fast prototyping
- Tensorflow with Keras for production model development

A word on reproducibility

One issue I have run into while training experimental models on GPU in Google Colab during the contribution period is the difficulty of obtaining **reproducible results** despite setting all recommended random seeds. Judging by how often this is discussed in Stackoverflow and Kaggle forums, this seems to be a common problem across multiple frameworks (PyTorch, Tensorflow, Keras…).

Google Colab's different GPUs can't be chosen by the users but are allocated depending on availability. This may be part of the problem as **precision in floating point operations** will depend on the hardware acceleration library compilation options and specific system architecture details.

While I haven't had the time to look into this further for now, I think it would be important to make sure the development environment behaves in a way that is **deterministic** before embarking on large-scale experimentation in order to be able to test out different hypotheses in a rigorous way. If this turns out to be impossible, the next best thing would be to report results accordingly by averaging over several trial runs, etc.

2. Datasets

During the contribution period, I created a dataset containing roughly 6000 images, 3000 for each category, by scraping the web. While my experimental models performed quite well on this dataset (92-96% accuracy), sifting through it image-by-image has made me aware of some **major flaws**:

- Male nudity, certain age groups, larger body sizes, and non-caucasian ethnicities are all underrepresented
- Most images are user-generated content posted on reddit and tumblr

- Sex scenes, as opposed to people posing alone, are underrepresented

My first priority would therefore be to **increase the diversity** of image sources, as well as address the problem of underrepresented demographic groups. Creating a better dataset in a more deliberate way, I would however have to be careful not to insert my own biases!

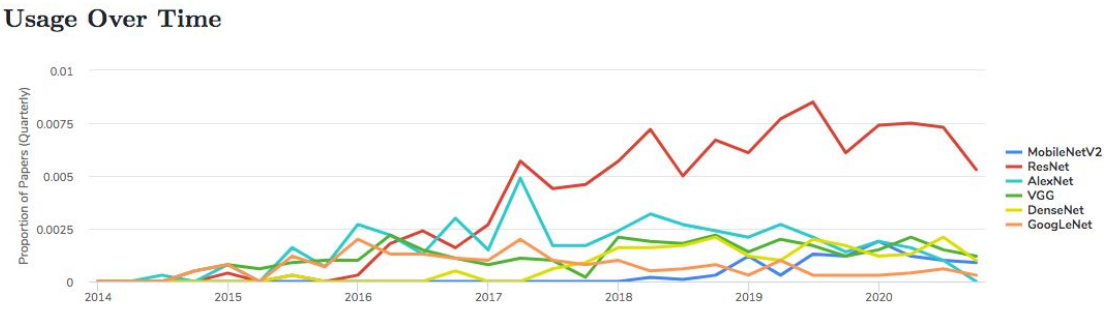
Additionally, I'd try to obtain the **NPDI dataset** commonly used for benchmarking in the scientific literature, as discussed in the Datasets report.

In terms of **dataset size**, I would aim for 10-20k total images as that seems doable both in terms of training time on GPU, Google Drive storage space, and the time it takes to clean the data image-by-image. For reference, with my current setup, the **training time** for 6000 images has been around 2 min per epoch, which is more than adequate for rapid iteration. In terms of data cleaning, using Adobe Bridge to quickly look through thumbnails, I'm able to manually check approximately **5000 images/hour**.

So far, I have been using a 80/20 train/validate split in all experiments. For the time being, I haven't needed an additional **test set** as the need to generalize to unseen data hasn't been addressed yet. This is however something that should be added. Ideally, the test set would be constructed by someone else than myself to avoid bias.

3. Model Architecture

Of the different possibilities discussed in the Classifier report, my first choice would be to train a **ResNet50** model by using **transfer learning**. Convolutional neural networks have become the *de facto* standard for visual classification tasks in recent years, and ResNet50 has proven to be one of the more robust and widely used architectures, obtaining SOTA results across many different computer vision tasks. Furthermore, ResNet50 offers a good trade-off between accuracy and model size, and even though it has already been around for a while, it remains one of the most popular choices:



(source: paperswithcode.com)

In recent years, several **light-weight architectures** such as MobileNet and EfficientNet designed for use on edge devices have achieved very good performance without sacrificing too much accuracy.

While I do have some concerns regarding these architectures (see my Classifier comparison report), I think it would be worth trying them out and comparing them to the ResNet50 baseline.

4. **Performance metrics**

As the train/dev/test datasets used in the model development stage can be designed to be perfectly balanced, I'd suggest **accuracy** as the performance metric to optimize for during this stage. However, as the majority of content uploaded to Wikicommons is at the moment SFW, **precision and recall** (including derived metrics such as the F1-score) would be a better choice on the account of the classes being imbalanced.

To evaluate the model's real-world performance, it would be good if the implementation included a **feedback loop**: once the model has made its predictions and a human moderator has reviewed them, there would ideally be a convenient way to retrieve and store the images, or at least a representative sample, with their corresponding true and predicted labels. This data could then be used to both evaluate the model's production performance, and serve as a basis for continuous improvement.

Other relevant metrics would be **speed/memory usage** at inference time. These could be used as thresholding metrics, i.e. not optimized for as long as they stay below whatever max values are deemed fit.

5. **Developing the model**

During the contribution period, I developed a baseline model that achieved 92.5% using a dataset containing roughly 5000 images, 2500 for each class.

After cleaning and extending the dataset to 6000 images, and adding data augmentation, the model achieved **95.9% accuracy on ResNet50** and was less prone to overfitting. The methodology, code and results of these experiments can be found in this repository on my GitHub.

In order to evaluate performance on a smaller architecture, I also fine tuned two **Squeezenet 1_0 and 1_1** models pre-trained on Imagenet, using the exact same dataset, data augmentation techniques, and hyperparameters as for ResNet50. The notebook can be found here. The reason for choosing this specific architecture was mainly one of convenience, as Squeezenet comes out-of-the-box with the FastAI API, while MobileNet and EfficientNet don't.

Compared to ResNet50, Squeezenet has roughly **25x fewer parameters** (~1.2M vs ~25M). It is also **3-5x smaller** than both MobileNet v2 and EfficientNetB0 (~3.5M and ~5.3M parameters, respectively), so I was quite curious about how it was going to perform.

Surprisingly, Squeezenet obtained comparable results to ResNet50 in terms of accuracy, in the range of **94-95%**. There was almost no difference in training time per epoch (~ 2 min), however it needed around 20 epochs to converge, vs. 10 for ResNet50. Most notably, SqueezeNet was **more well-behaved in respect to overfitting**.

Intuitively, both slower convergence and reduced overfitting make sense, as the model's representational capacity is lower on account of having considerably fewer parameters. However, this being a quick experiment, there is of course no way to know for certain that this is the actual reason.

Based on the tentative success of this smaller model, I would go ahead and evaluate MobileNet and EfficientNet as possible candidates alongside ResNet50.

My goal going forward would be to further improve the models in terms of accuracy, while making sure it generalizes well to new data and performs well in production. Some ideas for further improvement:

a) **Improving the dataset**, as discussed in the "datasets" section above.

b) **Regularization.** While performance is good on the training data, with training loss converging smoothly and nearing the Bayes error toward the end of training, the validation loss stalls and diverges quite early on.

   Getting more data could help with overfitting. Realistically, there's a limit to the max size of the dataset I could use in practice due to space and training time constraints, and even at this max size, the model's representational capacity would still be too big in comparison, resulting in low bias (good) and high variance (bad), as evidenced by overfitting.

   The solution then could be to decrease model capacity by using techniques such as L2 and/or drop-out regularization, possibly with the addition of early stopping.

c) Hyperparameter tuning. So far, all experiments have been conducted using Adam as the optimizer, a constant learning rate of 0.003, and mini-batches of size 64.

These, as well as other hyperparameters, can all be fine tuned. My approach here would be to use random search, rather than grid search, and use Fastai/PyTorch for prototyping to increase iteration speed.

d) **Model architecture.** I believe that adequate accuracy and performance can be If time allows, I would like to try out some methods I've come across in the scientific literature during the research phase of this project that have shown promising results:

- Substituting the last pooling layer with a Spatial Pyramid Pooling layer (SPP) to allow for variable input sizes, thus avoiding cropping/warping the images to fit the fixed input size required by CNN architectures.
- Using an SVM instead of dense layers + Softmax as the output layer.
- Adding an Attention module to the architecture.
- Adding a pre-processing module to make the model robust against adversarial attacks.

API for NSFW Classifier

Deploying the classifier would be the domain of Wikimedia's Machine Learning Platform team so any implementational details would have to be worked out with them prior to developing an API.

That said, I believe **FastAPI**, one of the proposed solutions in the corresponding microtask, would be a good choice of framework for the following reasons:

- Faster than Flask
- Lightweight
- Supports asynchronous function handlers
- Has built-in support for data validation, JSON serialization, automatic API documentation, and more

Video Processing Module for NSFW Classifier

Compared to images, video has the distinct property that it contains temporal information (context) in addition to spatial information (images).

For video classification, different approaches can be used over single or multi frame analysis. According to a Google research paper, Large-scale Video Classification with Convolutional Neural Networks, spatio-temporal networks display only a surprisingly modest improvement compared to single-frame models (59.3% vs. 60.9%). We can

thus hypothesize that since video is a combination of single frames, the correct classification of single images should translate to a correct classification of a video stream with reasonably high accuracy.

Proposed implementation

As it only takes one NSFW frame to make a whole video inappropriate, I suggest using the **max NSFW-score** attributed to the individual frames as criterion for classifying the whole video. While this may seem extreme, an example of this would be a malicious troll inserting a few seconds of inappropriate content into a childrens' movie.

Classifying methodology

1. Extract frames from the video.
2. Feed each frame to the NSFW classifier in a loop.
3. Retrieve the NSFW score after each pass.
4. If the NSFW-score is higher than a predetermined threshold: break loop, classify video as NSFW.
5. Else: continue feeding frames to the classifier.
6. If the end of the video is reached, return the max NSFW-score of the individual frames.

Extracting video frames

I suggest **OpenCV** as the library of choice for this subtask, for the following reasons:

- It's **open source** and **free for commercial use**.
- It's a **highly optimized** library with focus on real-time applications.
- It's **cross-platform**, with C++, Python and Java interfaces that support Linux, MacOS, Windows, iOS, and Android.
- It's been around for 20 years, is **well-documented**, and has a thriving community.

Another alternative worth looking into would be **Decord**, a video slicing library specifically designed for deep learning applications that is reportedly faster than OpenCV. https://github.com/dmlc/decord

Maybe both solutions could be implemented and tested for robustness and speed, if time allows.

Video processing time

An important question to consider would be the **processing time** for a video, and how it could be reduced. Assuming that inference will be done on CPU to account for a worst-case scenario, images would be processed sequentially and not in batches. The

total processing time would thus be a function of the total number of frames extracted, plus classification time per frame.

For experimental purposes, I created a Python script for frame extraction based on OpenCV and downloaded a 32-min mp4 video of medium-low resolution (480p).

The first question is, **at what time intervals should frames be extracted** to reliably classify a video? For the test video, I found that extracting frames at 1-second intervals yielded too many redundant images, and that 5-second intervals seemed like a reasonable choice. However, for very short formats such as gifs, 5 seconds may be insufficient. I would therefore propose to let the time interval be **a function of the total video length**, ranging between 1 and 5 seconds. So for instance, a 10-second gif would yield 10 frames, while a 30-min video would yield 360.

Extracting the frames at 1-second time-intervals from the abovementioned 32-min video:

```
(nsfw-env) [img_test]$ python vid2frame.py black.mp4 ./extracted_frames 1000
Extracting frames...
Extracted 1942 frames
Elapsed time: 166.3064 seconds
(nsfw-env) [img_test]$
```

This operation yielded 1942 images of size 854 x 480 px, for a total of 140 MB. 1942 frames / 166 seconds = 11.6 frames/second (= ~**700 frames/min**)

In my Classifier comparison report, I discuss inference speed of ResNet50-based NSFW classifiers in the literature. In the most comprehensive study, the average inference time on CPU was **720 images/min** (vs. 4500 on GPU). In the worst case scenario, where all the frames of a video would have to be passed through the classifier, the **total classification speed** would be approximately:

Video length	Frame extraction time interval	Size of resulting image folder*	Total number of frames extracted	Frame extraction time	Inference time	Total time
30 min	5 sec	26MB	360	30 sec	30 sec	60 sec
10 min	4 sec	11MB	150	13 sec	13 sec	26 sec
2 min	2 sec	4.5MB	60	5 sec	5 sec	10 sec
30 sec	1 sec	2MB	30	2.5 sec	2.5 sec	5 sec

*** Assuming 480p video quality**

The above is based on my current setup and non-optimized methodology, but it does give a rough estimate of the order of magnitude.

Project timeline

This week-by-week timeline provides a rough outline of the major subtasks, milestones, and deliverables.

Week	Dates	Tasks
1	Dec 1-4	Community bonding period. Define project specs & requirements, set up local dev environment & dependencies, refine timeline for project milestones and deliverables. Write **blog post #1 (Dec 1)**.
2	Dec 7-11	Curate/clean **datasets** & upload to training environment.
	Dec 11	**Initial feedback due**
3	Dec 14-18	Develop an **MVP version of classifier**, explore options for deployment with Machine Learning Platform Team. Write **blog post #2 (Dec 15)**
4	Dec 20-24	Outline strategy for **model improvement** with concrete experiments to undertake. Create detailed report w/ links to research papers.
5	Dec 26-31	Run experiments to iteratively improve model performance. **Research and plan API** development. Write **blog post #3 (Dec 29)**
6	Jan 4-8	**Develop API**. Make sure all work up to this point is correctly organized and well-documented.
	Jan 11	**Mid-point feedback due**
7	Jan 12-15	**Research and plan video pre-processing module**. Outline strategy for continued model improvement. Write **blog post #4 (Jan 12)**
8	Jan 18-22	**Develop video pre-processing module** & research how to integrate it into the production environment.
9	Jan 25-29	**Test & integrate video pre-processing module** into production version. If the model has been deployed, evaluate real-world performance and outline strategy for continued improvement. Write **blog post #5 (Jan 26)**
10	Feb 1-5	**Review** work done so far, improve & refactor if needed. Make sure everything is correctly organized and well-documented.
11	Feb 8-12	Final cycle of **model improvement**. Write **blog post #6 (Feb 9)**
12	Feb 16-19	Catch up with parts of the project that may have fallen behind or need improvement in view of wrapping up the project next week.
13	Feb 22-27	Wrap up project, repos, docs, etc. Write **blog post #7 (Feb 23)**
	Mar 2	**Final feedback due**