Assignment 2

Introduction

In the second assignment the goal is to build a deep learning model that given an image can as accurately as possibly predict the tags that the uploader has assigned to it. The framework that was used is *fasta.ai* by Jeremy Howard and Rachel Thomas, which basically is a layer on top of pytorch that wraps up pytorch's low level code and functionality so that we could focus more on the problem itself.

Reforming (slightly) the task

The dataset at our disposal consists of more than 11696 images scraped from Instagram, that have the hashtag *#recipe* and each of these images is expected to have at least one tag related to its content which is not always the case since 3822 images have no tags assigned to them reducing the total count of useful images to 7874. The total unique values of labels that were found in the dataset was 97, the distribution of which can be seen in Figure , but heavy preprocessing was applied on them as an attempt to reduce that number to at least half its value. The raw labels were, at the first phase of the preprocessing, lemmatized so that words like *cake* and *cakes* were grouped together that led to the count of distinct values of the labels to be significantly reduced to a total of 62 labels. As a side effect of the lemmatization process, tags like *vegan* and *vegetarian* that are semantically different were also grouped together. Taking into account the ratio of the total count of distinct labels and the total number of images it felt like we should reduce that number even further; this was actually confirmed when we run a baseline model, after doing some minor image-augmentation of course, in order to get an idea of what to expect as default accuracy and the results where a lot worse than what was initially expected. Other than the problem itself being extremely difficult, if solvable at all, the target feature, the tag label, is as noisy as it could be, since it is highly dependent on the uploader's perception or likings or personality in general, the overall trend that takes place at the specific place and time that the image was uploaded etc. Simply put, there is no guarantee that certain that the tags that the uploader chooses are the 'correct' ones or that there is an underlying pattern associated with these tags that the deep learning model will be able to uncover. Further grouping the labels in a way that is as balanced as possible, not only by lemmatizing them, but also grouping them together based on their meaning, even though sometimes they used to be quite different, and eventually throwing away the labels with low count, that were considered to add noisy to an already rather noisy dataset, we ended up working with 24 labels, the distribution of which is shown in *Figure 2.*

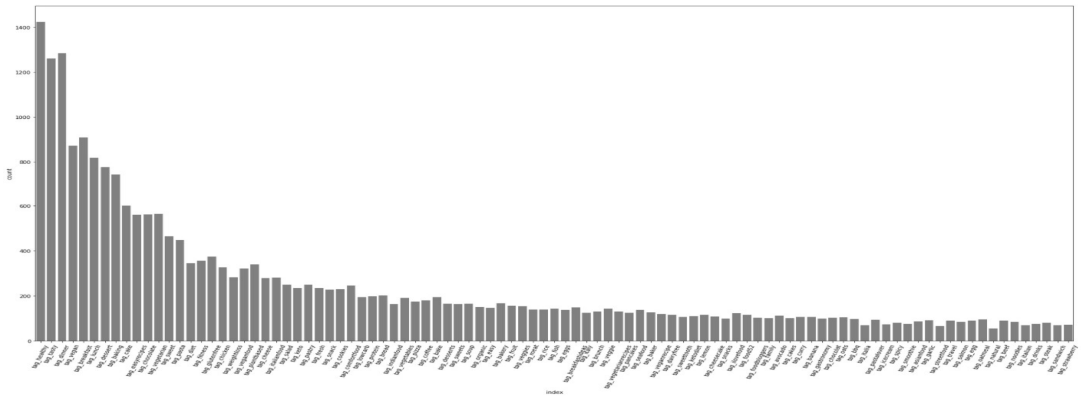

Figure 1. Distribution of distinct labels in the dataset

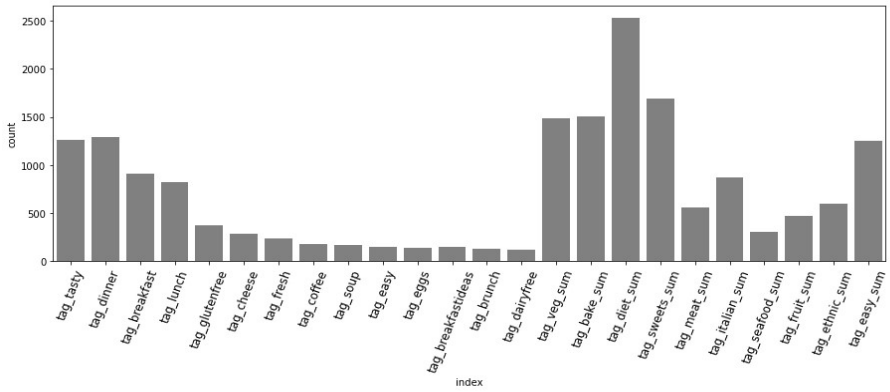

Figure 2.Distribution of distinct labels after lemmatizing and groupping tags together. Tags having the suffix 'sum' indicates that they were the product of the groupping procedure.

Data preprocessing

There are a few preprocessing steps that need to be done in order get our data ready for the *fastai* model. *Fastai* offers a convenient class (or *Container* as the documentation states) that wraps up all the steps needed for this, the *DataBlock.* Other than trivial procedures like properly loading up the dataset and splitting it to train and validation set, the *DataBlock* make the distinction between two transformations that are being applied on the dataset, the *item_tfms* which is applied on an item basis and the *batch_tfms,* with which we can define the augmentations to be applied on the dataset at a batch level.

In short, he dimensionality of the input space has to be implicitly defined, meaning that all images that are fed into the fastai model need to have the same total number of features, which is equal to the number of pixels an image has. However, it is expected that not all of the pictures have the same pixel count which is taken care of with *DataBlock* 's parameter *item_tfms.* Instead of just stretching or squeezing the data so that it fits some specified height and width pixel count, we used the *RandomResizedCrop* function that picks a random scaled crop of an image and resizes it to the size specified, 64 in our case. Such transformations are shown in *Figure 3.*

Figure 3. Three output images of RandomResizedCrop for the image at the top left.

As for the *batch_tfms,* it was clear from the start that heavy augmentations would have to be applied given the small size and the complexity of the data set. In a way it can be thought of artificially expanding the size of our training dataset, which as already mentioned above is quite limited, and thus can improve generalization. The augmentations chosen for the project are:

- Randomly flip an image with $p = 0.5$ probability
- Rotate the image and fill in with black pixels the unallocated parts of the rotated image
- Randomly adjust the overall lighting of the image at a different level.

The model

Training a model from scratch for this project would be infeasible, so we chose the 'transfer learning' path and trained on a couple of the most popular pretrained models for computer vision tasks, out of which the *resnet18* seemed to fit our needs. What initially made the *resnet* stand out from the rest of the models that existed at the time period that it came out, I the way that it dealt with the vanishing gradients; it introduced the idea of the so-called "identity shortcut connection" that skipped on or even more layers. It was claimed that going deeper with this architecture would not lead to degrading of the net as compared to swallower ones of the same architecture, since was possible to just stack identity mappings that would essentially make it at least equivalent to them. This means that, theoretically speaking at least, a *resnet50* architecture should perform at least equally with the *resnet18* that we chose as the final model of this project, given that there is enough computational power to train the deeper architecture. Using the *fastai* framework on could access the pretrained model's architecture with

```
model.summary()
```

and parameter weights with

```
[layer for layer in model.named_parameters()]
```

More interesting than looking at raw tensors would be to visualize what the feature maps and network activations corresponding to the first convolutional layer of the network, shown in the following figure.

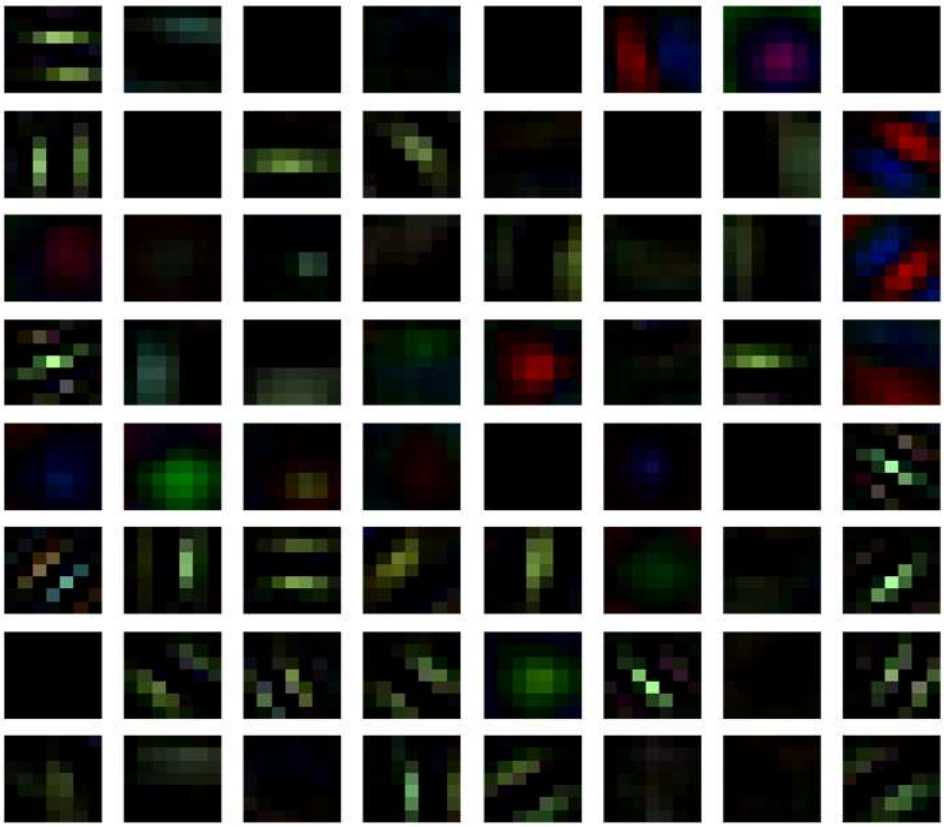

Figure 4. Feature maps / Filters of the 'conv1' layer of the pretrained resnet18. Takes in 3 channels and outputs 64 to the next layer, with a kernel size of 7, stride 2 and padding 3.

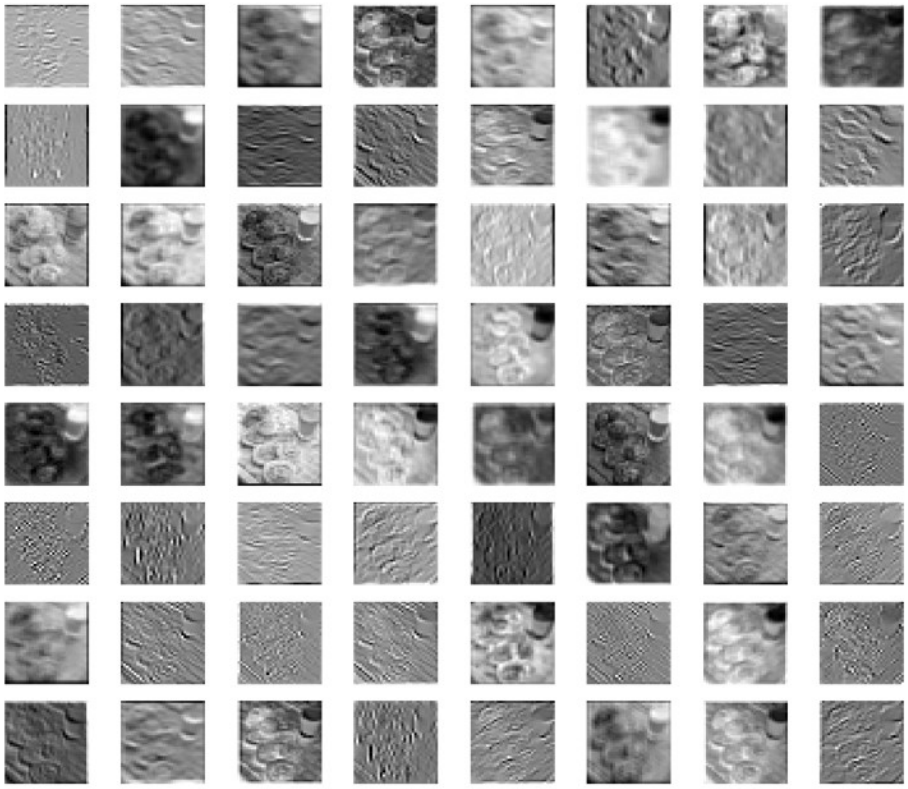

Figure 5. Outputs from the 'conv1' layer of the pretrained resnet18. Takes in 3 channels and outputs 64 to the next layer, with a kernel size of 7, stride 2 and padding 3. The same image as that of Figure 3 was fed through the layer.

Training

The essence of "transfer learning" lies in the fact that a model, having been trained for quite a long time on millions of images that it has been fed with, it has "learned" meaningful patterns and thus, it can exploit the knowledge gained from a previous task to improve generalization for another. In fact, only the very last few fully connected layers are needed to be trained, with the rest of the network being "frozen". As a sidenote, these last few layers are implicitly transformed so that the output of the last layer coincides with the count of classes, when we define the model as

```
model = cnn_learner(dls, resnet18, pretrained=True)
```

by taking into account the target variable that was previously defined and is now stored in the *dls* parameter. In other words, although our model might have been trained in a dataset, let's say cifar-10 that consists of 10 classes, we could as well use it to on completely different dataset and make predictions over a different number of classes; only thing that is needed is that the last layer (typically with softmax activation) is redefined so that the output is what it needs to be and the pretrained model gets fine_*tuned.* This is exactly the strategy we followed for this project with the pretrained *resnet18* architecture.

Possibly the most important hyper-parameter to be tuned for a neural net is the *learning rate* (*LR*); Picking extreme values would lead to a net that is either trained too slowly for a low value of *LR* or make the loss fluctuate around the optimal value, without ever getting closer to it or even causing outright divergence. *Fastai* offers a quite convenient function, *model.lr_find( ),* which seeks for a well-performing candidate for the *LR* parameter, based on the paper by *Leslie N. Smith - Cyclical Learning Rates for Training Neural Networks.* The output of the function is shown in *Figure 4.* It should be noted that once a pretrained model is loaded in, fastai automatically freezes all the pretrained layers, which means that the $LR \approx 0.02$ that was found previously (one of the two that are reported back by the *lr_funtcion( ) that corresponds to the steepest loss*) refers to the model when all the pretrained layers except for the classification-specific ones at the very end are frozen. So, we start training the model, at the current frozen state, with the suggested *LR,* so that only the fully connected layers at the end, that would otherwise be too difficult to train, are tuned for our task. The results for 15 epochs, that were requested are shown in *Figure 5.*

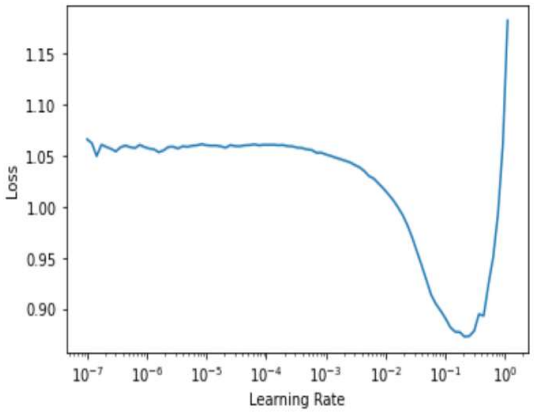

Figure 6. model.lr_find( ) visual output

epoch	train_loss	valid_loss	time
0	0.781344	0.499366	02:08
1	0.485502	0.321449	02:12
2	0.374997	0.311537	02:06
3	0.332219	0.300511	02:06
4	0.313249	0.316015	02:05
5	0.303413	0.296638	01:57
6	0.294805	0.277004	02:04
7	0.285983	0.274117	01:55
8	0.280332	0.272165	01:55
9	0.278921	0.274620	02:01
10	0.279094	0.272484	01:55
11	0.280458	0.281181	01:59
12	0.280973	0.277768	02:00
13	0.282907	0.278735	02:01
14	0.283030	0.279708	02:02

Figure 7. Fastai output while at the 'frozen' state

epoch	train_loss	valid_loss	time
0	0.277627	0.271630	02:04
1	0.275142	0.270805	02:06
2	0.275052	0.273454	02:01
3	0.276973	0.272790	02:01
4	0.277415	0.289770	02:01
5	0.276961	0.283857	01:55
6	0.277038	0.272810	01:59
7	0.276347	0.273734	01:58
8	0.273940	0.272182	02:00
9	0.272250	0.272015	01:57
10	0.270138	0.270633	01:58
11	0.268946	0.268720	01:58
12	0.267597	0.266964	01:58
13	0.267129	0.266682	01:58
14	0.265529	0.266809	01:57

Figure 8. Fastai output while at the 'unfrozen' state

As expected, the *train_loss* constantly keeps decreasing till a point, around the 9th epoch, where it reaches a plateau, while the *valid_loss* keeps decreasing till the 15th epoch, where it shows some uprising trend (although it is not shown here). That is an indication that the model has started learning the training images' specific features and its ability of generalizing to unseen images starts decreasing, meaning that it has started overfitting the data. At that point we stop the training procedure, "unfreeze" the model, find a new *LR* that corresponds to this unfrozen state and retrain every single parameter of the *resnet18.* It was requested that the model would be trained for 15 epochs as well and the results of this second training part are shown in *Figure 6.*

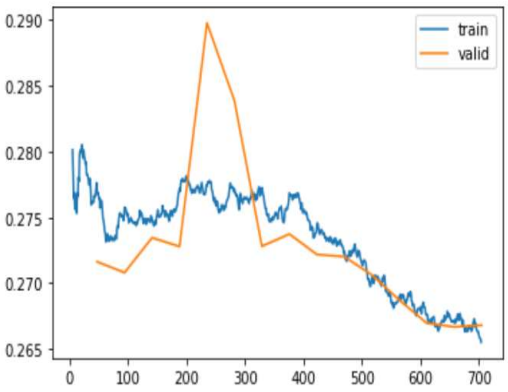

Figure 9. Loss vs images processed for the train and validation set.

The training process for the "unfrozen" state of the model seemed to plateau at around the 15th epoch for both the training and the validation loss, which could be considered as an indication that a larger *LR* should be used , but that led to no improvement in our case. A handy function offered by *fastai* that wraps up the steps of *i)* training the last layers of the frozen model for a specified number of epochs, *ii)* unfreezing the model and *iii)* training the rest of the network for a specified number of epochs with the use of *discriminative LR* is the

```
model.fine_tune(epochs=15, base_lr=lr_[0], freeze_epochs=15),
```

that was used to produce the outputs of *Figure 7, 8* and *9.*

The verdict

The model that can be found in this LINK was somewhat able to capture the underlying patterns of the data, given the nature of the problem, that even a human would have a hard time with and the hugely imbalanced and small-sized dataset. For demonstration purposes two images were fed into the model and the predicted tags are shown in the following Figures. For the prediction part of the project, we implemented a function that could easily be configured on the fly, although we could have used *fastai*'s implementation with appropriately tuned thresholds for classification. Moreover, the multilabel metrics offered by the *fastai,* like *fbeta_score*, *f1* and *recall*, did not seem to fit the needs of the task so we implemented and used one that has similarities with a *multilabel-Recall* kind of metric, but focuses on the dominant labels, completely ignoring the noise introduced by less important labels.

```
# Visualize the img_test (raw/unprocessed)
img_test.resize((500,400))
```

And our model's predictions:

```
my_predict_func(preds, request_number=2)
```

```
['tag_sweets_sum', 'tag_bake_sum']
```

```
# Visualize the img_test (raw/unprocessed)
img_test.resize((500,400))
```

And our model's predictions:

```
my_predict_func(preds, request_number=2)
```

```
['tag_diet_sum', 'tag_veg_sum']
```

Figure 10. Testing the final model on random images found on instagram.