Documentation

Table of contents

File-Structure.md

The list below contains all the information you need to understand all of the file structure in our project's `code` folder. For more information on how to run these files and/or reproduce our results, please see the How-to page of our wiki. Aside from the `code` folder, our project also has an `images` folder, which is described in the Images page of this Wiki.

Some remarks on notation:

- Deprecated scrips are denoted with *(deprecated)*; these files are no longer used in the final version of our model but were left in for completeness.
- Scripts that you can run from command line are denoted with **(runnable)**; the How-to page of this Wiki explains how to run them, and in what order.
- Notebooks can be recognized by the file extension `.ipynb`; the notebooks themselves contain further information regarding what they do.

File structure of the `code` folder

- **analysis**: Notebooks we used for analysis; both for exploratory data analysis (EDA) and analysis of the performance of the trained models on the validation set ("Failure Analysis").
    - `EDA_Histograms_Notebook_Creator.ipynb`
        - A notebook which can be used to create other notebooks filled with scripts that are responsible for creating histograms. You have to specify the location of the data, the slices you would like to use, whether you would like to use normal and/or abnormal patient data and the name of the new notebook. When these parameters are specified you can run the cells and a new notebook will be created in the folder. If you open and run this newly created notebook you can inspect the histograms. The goal was to create an easy way to create and inspect histograms for each group member.
    - `EDA_histograms_overlapping_patients.ipynb`
        - A notebook used to inspect the differences in average pixel intesities of slices of normal and abnormal brains. The notebook with output can be found on this Google Drive. The "unrun" cells can be found in this repository.
    - `EDA_nonoverlapping_histograms_abnormal_patients.ipynb`
        - A notebook used to create histograms based on the average pixel intensities of slices or abnormal brains.
    - `EDA_nonoverlapping_histograms_normal_patients.ipynb`
        - A notebook used to create histograms based on the average pixel intensities of slices or normal brains.
    - `EDA_RandomPlotter.ipynb`
        - A notebook used for visual inspection of the slices. In this notebook brains and slices of patients are randomly plotted and compared. Afterwards the most interesting patients are investigated more thoroughly by plitting all slices for each patient. The notebook with output can be found on this Google Drive. The "unrun" cells can be found in this repository.
    - `Failure_Analysis_CombinedNet.ipynb`
        - A notebook used to perform failure analysis on the full combined network finetuning the combined network. The analysis consists of a confusion matrix, classification metrics, ROC-curve and AUC, and visualisations of misclassified patients.
    - `Failure_Analysis_LSTM.ipynb`
        - A notebook identical used to `Failure_Analysis_CombinedNet.ipynb` but instead used to analyse the combined network after training the LSTM, and thus before fine tuning the combined network.
    - `Failure_Analysis_ResNet.ipynb`
        - A notebook used to perform failure analysis on the ResNet part of our network. Analysis consists of the same steps as in `Failure_Analysis_CombinedNet.ipynb` and `Failure_Analysis_LSTM.ipynb`, as well as visualisation of probability predictions for each slice of patients labeled as normal or abnormal.
    - `Preproc_Augment_Visualisation.ipynb`
        - A notebook with example visualisations of the original acquisitions of a random slice, the preprocessed version of this slice, and

the augmented (flipped, rotated, flipped+rotated) versions of this slice, as a test and for clarification purposes.

- **dataset**: All of the code for extracting slices from the data, splitting the training data into a training set and a validation set for training, and dataset structures for slices and patients that we can feed to data loaders.
  - `create_data_split.py` **(runnable)**
    - Runnable script containing functions for randomly splitting the data into a training and a validation set, making sure that there is no overlap between patients in the training and validation set (i.e. all slices of a patient are either part of train or part of validation). Running this file results in a train/validation split which is saved for use during training (so that all steps of our training procedure use the same validation set).
  - `create_slices.py` **(runnable)**
    - Runnable script that extracts all slices for all participants from the data and stores them in separate files, while also storing a CSV file that keeps track of which slice belongs to which patient and the label (normal/abnormal) belonging to that patient. This CSV is used as a reference by the slice and patient datasets, so that they know which files belong to which patient/label.
  - `patient_dataset.py`
    - File containing a class for the dataset structure for patients, which is compatible with data loaders. The getter function for this dataset can return a single slice or a set of slices for a given patient; it also applies our preprocessing (and possibly augmentation) steps to the slices before returning them. This dataset structure is used to trainthe LSTM part of our network, as well as to fine-tune the combined network.
  - `preprocessing.py`
    - Script containing functions for preprocessing and augmenting slices. The `preprocess` function preprocesses slices by standardizing and center cropping them (leaving images of 426 by 426 pixels). The `augment` function augments a slice by flipping it horizontally (i.e. swapping left and right) with a certain probability and rotating it with a random small angle (between -10 and 10) with a certain probability. The `get_dataset_mean_std` function was used to find the mean and standard deviation per acquisition over the training set; the outputs of this function are now hardcoded into the standardisation.
  - `slice_dataset.py`
    - Script containing the class for the dataset structure for slices, which is compatible with data loaders. The getter function for this dataset structure applies our preprocessing (and possibly augmentation) steps to the slices before returning them. This dataset structure is used to train the ResNet component of our network, which predicts whether a slice came from a "normal" or an "abnormal" patient.
- **models**: Scripts containing specifications for the various components of our model.
  - `combined_net.py`
    - The implementation of our complete network, which combines a ResNet (see `omnipotent_resnet.py`) and an LSTM (see `lstm.py`). The combined net takes a ResNet that has been trained to predict whether a slice came from a "normal" or an "abnormal" patient, removes the final layer in order to make the ResNet produce "feature vectors", and feeds these feature vectors to the LSTM. The `CombinedNet` class also contains a function that allows us to "freeze" the weights of the ResNet component, so that they will not be updated during training. This is used when we train only the LSTM part of the network.
  - `feature_vector_model.py` *(deprecated)*
    - An implementation of a simple feature vector model, consisting of several linear layers. This network was used as a placeholder for the ResNet part of our network; it is not used anymore, since we now use the ResNet.
  - `lstm.py`
    - Script containing the implementation of the LSTM component of our network; the user can specify the number of features that are fed into the LSTM as input, the number of hidden units the network contains, and whether the LSTM is bidirectional or not.
  - `omnipotent_resnet.py`
    - Script containing code for our redefined ResNet architecture, which takes a ResNet as implemented in `torchvision.models`, removes the final layer, and replaces it with two fully connected layers in order for the model to be able to predict whether a slice came from a "normal" or an "abnormal" patient. The user can specify the number of hidden units in the first fully connected

layer, as this corresponds to the length of the "feature vectors" produced by the ResNet once the final layer is removed.

  - `resnet.py` *(deprecated)*
    - Basic implementation of our redefined ResNet model for ResNet-34; this implementation was made redundant by `omnipotent_resnet.py`, which is much more flexible.

- **test**: Code for using a trained (ResNet + LSTM) model to create a valid submission file.
  - `submission.py` **(runnable)**
    - Script that takes several parameters (such as the location of a trained model's parameter file and various specifics of the model to test) and uses them to recreate a trained model. This model is then used to obtain probabilities and labels (i.e. "normal"/"abnormal") for every patient in the test set, which are finally written to a submission file that can be submitted to Grand Challenge.

- **train**: All of the code we used to train our model: a general training loop and specific calls to that loop for different stages in our model training.
  - `train.py`
    - Script containing the `Trainer` class, that can be instantiated using several training parameters (e.g. model, criterion, optimizer, data loaders, number of epochs) and can then be used to train a model. It has some functions for training and validation and some helper functions, as well as the `train_and_validate` function we use to train (and validate) all of our models and save model parameters. The trainer logs training and validation losses and accuracies to Weights and Biases.
  - `train-combinednet.py` **(runnable)**
    - Script that takes several parameters in order to create a `Trainer` instance that is used to fine-tune the combined network (ResNet + LSTM) on the `PatientDataset` created from the data split obtained by running `create_data_split.py`, updating parameters of both the ResNet and the LSTM with a smaller learning rate.
  - `train-lstm.py` **(runnable)**
    - Script that takes several parameters in order to create a `Trainer` instance that is used to train the LSTM part of our network on the `PatientDataset` created from the data split obtained by running `create_data_split.py`, freezing the trained ResNet parameters and only updating parameters of the LSTM. The network is trained so that it should return values close to 1 for "abnormal" patients, and values close to 0 for "normal" patients.
  - `train-resnet.py` **(runnable)**
    - Script that takes several parameters in order to create a `Trainer` instance that is used to train the ResNet part of our network on the `SliceDataset` created from the data split obtained by running `create_data_split.py`; it trains the ResNet so that it should return values close to 1 if slices belong to an "abnormal" patients, and values close to 0 if they belong to "normal" patients.

- **visualisation**: The code we used to create visualisations of the data, i.e., to display slices of brain scans, as well as to create several kind of plots used for our analysis.
  - `analysis_plotter.py`
    - Code for creating plots used for "failure analysis" (see `Failure_Analysis_Full.ipynb` & `Failure_Analysis_ResNet.ipynb`). `plot_roc_curve` plots the ROC-curve and shows the AUC score. `plot_confusion_matrix` creates a confusion matrix given a list of targets and predictions.
  - `blob_plotter.py`
    - This module contains methods that can be used to plot the slices with red circles around the brightest spots. It is especially usefull for slices with the T2-FLAIR acquisition.
  - `intensity_histogram.py`
    - This module contains methods that are used to create histograms based on the average pixel intensities of slices and all normal and or abnormal patients. The average pixel intensities can be calculated over all slices, groups of slices and over all patients or only normal / abnormal patients. The histograms for both normal and abnormal patients are plotted in the same figure to make analysing more user-friendly.

    - `slice_plotter.py`
        - Code for plotting slices of patients. `plot_patient_slices` plots all 32 T2-Flair images of a patient, given all patient image data. The `plot_slices` function shows the T2-Flair acquisition of each slice in a range of slices for a given patient, whereas the `plot_slices_by_acquisition` function plots all three acquisition types (T1, T2, T2-Flair) for a range of slices of a given patient, and `plot_slice_by_acquisition` plots all three acquisitions for a single slice.
- `full-pipeline.ipynb`
    - Notebook that contains the full pipeline of our project. This notebook can be used to reproduce our results (see How-to). It contains parameters for all of the components of the pipeline, and runs them all in order: generating slice data, creating the train/validation split, training ResNet, training the LSTM, fine-tuning the CombinedNet, and predicting test data. In order to keep track of the best model, it creates a temporary folder in which it stores some strings; this folder is cleaned up (i.e. removed) afterwards.
- `utils.py`
    - File containing basic utility functions used across the code. It contains is `set_seed`, which makes our pipeline more deterministic by setting seeds for random processes. It also contains the function `clean_up`, which removes a temporary directory that is created when running the full pipeline for our project (see `full-pipeline.ipynb`).

Home.md

Welcome to our project's Wiki!

In this Wiki, we will elaborate on some aspects of our code.

The pages of this Wiki will explain the file structure and training process of our project. The descriptions correspond to the code in the `master` branch of this repository.

Aside from the `master` branch, our GitHub repository contains two other, older branches. The `old_code` branch contains an older version of our project, in which we used an incorrect split of training and validation data (this was kept in the repository since we are planning on discussing it in our presentation). The `cross_validation` branch contains code for cross-validation on this older version, but after moving to the correct split of training and validation data we had to drop cross validation since training on a single train/validation split already takes 4 to 5 days.

This Wiki consists of three pages:

- File Structure: An explanation of all files in the `code` folder and what they do.
- How-to: An explanation of which code files need to be run in order to reproduce our results, how to run them, and a more detailed explanation of what these files do.
- Images: An explanation of all the images in the `images` folder.

How-to.md

This page contains a detailed explanation of how our code can be used to train and test a model that consists of a ResNet and an LSTM; all files that are denoted as **(runnable)** in the File Structure page will be elaborated upon. Note that, because scheduling and running our code on Cartesius would take a long time, we opted to train our network on Google Cloud Platform; the default paths in each of these files point to where we stored the data on the Cloud Platform, not to where they are stored on Cartesius.

Our `code` folder contains a single, easy-to-run notebook: `full-pipeline.ipynb`. This notebook calls all of the runnable files in the correct order, to perform all of the steps required to train and test our network. This notebook does not contain a lot of code; instead, to keep things modular and to allow us to easily run the various components of our code separately on cloud platforms, we opted to use separate scripts all containing different parts of our code, which the `full-pipeline.ipynb` notebook calls one by one. We did not actually use this notebook to produce our result -- we ran these files from a command prompt -- but the notebook works in exactly the same way, simply running these commands from a notebook instead of a command prompt. below, we will explain how these separate scripts work, and how to run them (separately, via command prompt) to reproduce our results.

There are a total of 6 files that can be run from command line:

1. `create_slices.py`
2. `create_data_split.py`
3. `train_resnet.py`
4. `train_lstm.py`
5. `train_combinednet.py`
6. `submission.py`

In order to reproduce our results, these files need to be run in the order specified above. How to do this is explained below.

1. `create_slices.py`

The first file that needs to be executed is `create_slices.py`, which takes the data (the training and testing files we were given for the project) and splits it into separate slice files, one per slice, using naming scheme `<patient_nr>_<slice_id>.pt`, where `<patient_nr>` is the identifying number of the patient the slice belongs to and `<slice_id>` is the index of the slice (integer between 0 and 31). Splitting the data into separate slice files is required in order to use our dataset structures (see `patient_dataset.py` and `slice_dataset.py` in File Structure). The script also creates a file, `labels_slices.csv`, which stores (patient number, slice number, class (normal/abnormal)) combinations.

The command required to run this script from the command prompt has the following shape:

```
python create_slices.py -d <data path> -o <output path> --train --test
```

Note that `create_slices.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `create_slices.py` script (e.g. `dataset/create_slices.py`).

This file has four parameters:

- `-d <data path>` The path to the original data; `<data path>` should be replaced with the path where the original data is stored. By default this is `../../../data`, which is where we stored the data on Google Cloud Platform, but when running this on Cartesius, this should be replaced with `/projects/0/ismi2018/BrainTriage`.
- `-o <output path>` The output path; `<output path>` should be replaced by the path to the location (either relative to the current folder or absolute) where the created slices should be stored (**WARNING**: this can get very big; the slices from the train data alone take up approximately 94 GB)
- `--train` This parameter specifies whether the slices from the training data should be extracted (leaving it out will result in the training data not being extracted; can be useful if you already have the training data and wish to extract the test data only).
- `--test` This parameter specifies whether the slices from the test data should be extracted (leaving it out will result in the test data not being extracted; can be useful if you already have the test data and wish to extract the training data only).

2. `create_data_split.py`

After splitting the data into slices, the next step is to split the training data into a training set and a validation set (by default, this is a 90/10 split). This is done by running `create_data_split.py`. This file uses the `labels_slices.csv` file created by `create_slices.py` and splits it into train

and validation parts, which it saves to CSV files so that every step in our training process can use the same patients for training and validation. Note that this file did not exist in the old version of our code (see Home).

The command required to run this script from a command prompt has the following form:

```
python create_data_split.py -k <k> -d <data path> -ds <data split path>
```

Note that `create_data_split.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `create_data_split.py` script (e.g. `dataset/create_data_split.py`).

This file has three parameters:

- `-k <k>` The number of "folds" the data should be split into (replace `<k>` with some positive integer). The first fold (by default, 100 patients, evenly split between "normal" and "abnormal") is used as a validation set, whereas the remaining k-1 folds are used as a training set. Note that, due to the need to create k evenly-sized splits, not all values of k work (but e.g. 2, 5, 10, 20 all do work).
- `-d <data path>` The path to the sliced data (`<data path>` should be replaced with whatever was used in the `-o` parameter of `create_slices.py`).
- `-ds <data split path>` The path to where the CSV files that store the train/validation split are stored.

3. `train_resnet.py`

Once the training data has been split into train and validation sets, the next step is training the ResNet component of our network, which is done by `train_resnet.py`. Using the train/validation split created by `create_data_slices.py`, this script trains a residual neural network (ResNet). The ResNet is either ResNet-18, ResNet-34 or ResNet-50, as predefined in `torchvision.models` but with slight alterations (removing the final layer and replacing it with two fully connected layers) in order to enable the network to predict from which class of patient ("normal" or "abnormal") a given slice came. After every epoch, model weights are stored a specified location.

The command required to run this script from the command prompt has the following shape:

```
python train-resnet.py <name> <resnet type> -s <seed> -d <data path> -ds <data split path> -lr <learning rate> -e <epochs> -b <batch size> -m <model path> -f <nr of features> -ts <target slices> -afp <flip probability> -arp <rotate probability> --tuple --pretrained
```

Note that `train-resnet.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `train-resnet.py` script (e.g. `train/train-resnet.py`).

This file has 15 parameters:

- `<name>` The name of the model (can be any string), for tracking purposes.
- `<resnet type>` The type of ResNet to use (resnet18, resnet34, resnet50).
- `-s <seed>` The seed that the random components of the training procedure should use.
- `-d <data path>` The path to where the separate slices are stored (identical to the `-o` parameter in `create_slices.py`).
- `-ds <data split path>` The path to where the training/validation data split files are stored (identical to `-ds` in `create_data_split.py`).
- `-lr <learning rate>` The learning rate to be used during training (default = 0.0001).
- `-e <epochs>` The number of epochs the model should be trained for (default = 30. **WARNING**: each epoch takes approximately 1.5 hours, depending on hardware. If you do not want to wait for days for this model to stop training, consider lowering the number of epochs).
- `-b <batch size>` Number of slices in one batch. Default = 16. **WARNING**: batch sizes larger than 16 might result in out-of-memory errors, depending on hardware).
- `-m <model path>` Path to where the intermediate model parameters should be saved after each epoch.
- `-f <nr of features>` The size of the feature vector that the network should produce once the final layer is removed in the combined (ResNet + LSTM) network, i.e. the number of output features of the last fully connected layer. Default = 128.
- `-ts <target slices>` Which slice indices of each patient to use; this can be a single slice index (in which case only the slices at that position in the list of slices of each patient will be used), a list of slice indices (in which case all slices at the specified indices will be used) or a tuple (in which case all slices starting from the first element up to but not including the last element of the tuple will be used). Note that values should always be between 0 and 31 (except for tuples, in which case ranges ending in 32 are possible). Default = (0, 32).

- `-afp <flip probability>` Probability that training slices will be flipped left to right (part of our data augmentation). Default = 0.5.
- `-arp <rotate probability>` Probability that training slices will be randomly slightly rotated (part of our data augmentation). Default = 0.5.
- `--tuple` Whether or not the `-ts` parameter is a tuple. If this parameter is added, the `-ts` parameters will be treated as a tuple; if not, it will be treated as a list.
- `--pretrained` Whether to use the pre-trained variant of the ResNet (as provided by `torchvision.models`). If this parameter is added, the pre-trained version is used; if not, the model will be trained from scratch.

4. `train_lstm.py`

After training the ResNet component of the network, we can start training the LSTM component, which is done by `train_lstm.py`. This script takes the ResNet model that was trained in `train_resnet.py`, removes the final layer so that it outputs feature vectors of a predefined shape (`-f` in `train_resnet.py`). The feature vectors obtained for all slices corresponding to a patient are then used as input for the LSTM component, which uses these feature vectors to determine whether the patient belongs to the "normal" or "abnormal" class. During the training of the LSTM component, the weights of the ResNet component are "frozen" so that only the weights of the LSTM part are updated. After every epoch, model weights of the total network (ResNet + LSTM) are stored at a specified location.

The command required to run this script from the command prompt has the following shape:

```
python train-lstm.py <name> <resnet type> -s <seed> -d <data path> -ds <data split path> -c <cnn path> -lr <learning rate> -e <epochs> -b <batch size> -m <model path> -f <nr of features> -ts <target slices> -afp <flip probability> -arp <rotate probability> --tuple
```

Note that `train-lstm.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `train-lstm.py` script (e.g. `train/train-lstm.py`).

This file has 15 parameters, most of which are identical to those used in `train-resnet.py`. The only differences are:

- `-c <cnn path>` The path to where the ResNet (which is a type of Convolutional Neural Network) weights are stored (`-m` in `train-resnet.py`, but with the addition of the actual file name of the model file).
- This file does not use the `--pretrained` parameter, since the ResNet component should never use the pre-trained weights provided by `torchvision.models` (we always load the weights obtained from `train-resnet.py`).
- The default batch size is 2 (as opposed to 16 in `train-resnet.py`).
- `<resnet type>` can only be "resnet18" or "resnet34", since we never used ResNet-50.

5. `train_combinednet.py`

After training the ResNet and LSTM components of our network, we attempt to do some fine-tuning by training both components together for a few more epochs. This process is identical to how the LSTM component is trained, except that the weights of the ResNet component are no longer frozen.

The command required to run this script from the command prompt has the following shape:

```
python train-combinednet.py <name> <resnet type> -s <seed> -d <data path> -ds <data split path> -l <lstm path> -lr <learning rate> -e <epochs> -b <batch size> -m <model path> -f <nr of features> -ts <target slices> -afp <flip probability> -arp <rotate probability> --tuple
```

Note that `train-combinednet.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `train-combinednet.py` script (e.g. `train/train-combinednet.py`).

This file has 15 parameters, which are all identical to those used by `train-lstm.py`, with one exception:

- `-l <lstm path>` The path to where the weights obtained from LSTM training are stored (i.e. the weights of the trained ResNet component + the trained LSTM component). This parameter works similarly to `-c` in `train-lstm.py`, but it uses the model files created by `train-lstm.py`, which include weights for both the ResNet and the LSTM component.

6. `submission.py`

Finally, we created a script that creates a valid submission file that can be submitted to Grand Challenge, given a trained model. Using the model trained in the previous files, it predicts the labels ("normal"/"abnormal") for all patients in the final test set, which it then writes to a CSV submission file.

The command required to run this script from the command prompt has the following shape:

`python submission.py <name> <resnet type> <model file name> -d <test data path> -m <model path> -sd <submission path> -b <batch size> -s <target slices> -f <nr of features>`

Note that `submission.py` should be replaced with the path leading to the file if you are not currently in the directory that contains the `submission.py` script (e.g. `test/submission.py`).

This file has 9 parameters:

- `<name>` The name of the model (can be any string), for tracking purposes.
- `<resnet type>` The type of ResNet to use (resnet18 or resnet34).
- `<model file name>` File name of the final trained model's parameter file (excluding path).
- `-d <test data path>` Path to where the slice files for the test data are stored (as created by `create_slices.py` with the `--test` parameter).
- `-m <model path>` Path to where model parameters are stored, i.e. where the final trained model's parameter file is located (excluding file name).
- `-sd <submission path>` Path to where submissions should be stored (excluding file name).
- `-b <batch size>` Batch size to use during testing. Default = 2 (**WARNING**: increasing batch size might lead to out-of-memory errors, depending on hardware).
- `-s <target slices>` Which slice indices of each patient to use (identical to `-ts` in `train-resnet.py`).
- `-f <nr of features>` The size of the feature vectors produced by the ResNet component. Default = 128 (needs to be identical to `-f` in `train_resnet.py`.

Images.md

Aside from the `code` folder, the contents of which are explained in the File Structure page of this Wiki, we have also included an `images` folder, which contains several images that are relevant to our project (information regarding slice acquisitions and loss/accuracy plots as produced by Weights and Biases). Below, we will list these images and what they show.

- `combinednet.png`
  - ???
- `differences.png`
  - Image showing how every acquisition (T1, T2, T2-Flair) displays particular parts of brain scans (e.g. CSF, gray/white matter, inflammations).
- `lstm.png`
  - A plot of training and validation accuracy and loss for our LSTM training procedure as generated by Weights and Biases, using ResNet-18 with the weights resulting from training the ResNet for 9 epochs. This network was trained for 30 epochs (the x-axis shows steps due to ease of implementation, each epoch (training + testing) consisted of 500 steps). The LSTM weights that were stored after the 19th epoch were used in the Combined Net fine-tuning, since this resulted in the highest validation accuracy (0.66).
- `resnet18_augmentation.png`
  - A plot of training and validation accuracy and loss for our ResNet-18 training procedure as generated by Weights and Biases, using data augmentation in the training set. This network was trained for 30 epochs (the x-axis shows steps due to ease of implementation, each epoch (training + testing) consisted of 2000 steps). The ResNet-18 weights that were stored after the 9th epoch were used for the ResNet to train the LSTM, since this resulted in the highest validation accuracy (0.6222) for ResNet-18, which was slightly lower than the best ResNet-34 validation accuracy (0.6297) but ResNet18 is much less complex and performed similarly during ResNet failure analysis.
- `resnet34_augmentation.png`
  - A plot of training and validation accuracy and loss for our ResNet-34 training procedure as generated by Weights and Biases, using data augmentation in the training set. This network was trained for 30 epochs (the x-axis shows steps due to ease of implementation, each epoch (training + testing) consisted of 2000 steps). As can be seen in this image, the network overfits much less than without data augmentation, but performance is very similar to ResNet-18 (and since ResNet-18 is simpler, we opted for ResNet18 in the combined network).
- `resnet34_no_augmentation.png`
  - A plot of training and validation accuracy and loss for our ResNet-34 training procedure as generated by Weights and Biases, without the use of data augmentation in the training set. This network was trained for 15 epochs (the x-axis shows steps due to ease of implementation, each epoch (training + testing) consisted of 2000 steps). As can be seen in this image, the network severely overfits: validation loss gets as high as 1.7. Hence we chose to re-train ResNet-34, as well as train the simpler ResNet-18, using data augmentation, which had a big effect on the amount of overfitting.