Capstone Project
Machine Learning Engineer Nanodegree

Mark Rode
August 2019

Definition

Project Overview

Inspired by Stephen Wolfram's "Personal Analytics" and Cal Newport's book "Deep Work"' I set out to build a simple desktop application that would help people improve focus whilst keeping better track of their screen time. With ever increasing technology enabled distractions, I wanted to build something that utilizes technology to help people use their time more effectively.

As I often catch myself unknowingly wandering to Youtube or looking at one of the many apps that have habituated me to regularly check-in on them I thought it would be helpful to build a tool that can tell me how I'm spending my time on the computer and possibly warn me if I don't catch myself wasting it.

Problem Statement

The first version of the proposed application will be solving a multi-label classification problem with the following approach:

1. Create an evenly balanced dataset that consists of screenshots of the following 14 desktop activities:

Class index	Class description	Class code
0	writing code in an IDE	code
1	navigating the desktop	desktop
2	using Gmail in the browser	email
3	browsing Facebook	facebook
4	browsing Github	github
5	Google searches	google
6	using a jupyter notebook	jupyter
7	using Kaggle	kaggle
8	reading Medium articles	medium
9	Netflix	netflix
10	taking notes	notes
11	reading pdf documents	pdf
12	using the terminal	terminal
13	watching/navigating Youtube	youtube

2. Use transfer-learning to teach a pre-trained image classifier to differentiate between the 14 classes.
3. Create a script that takes screenshots at regular time intervals and feeds images to the classifier.
4. Have the app notify the user via Mac's built in notification functionality and make the following notifications:

- A summary of the last 30 minutes of screen activity
- A warning notification (with sound) if the user is spending more than n seconds of the 30 minute time period doing an 'unproductive' activity e.g. Facebook, Netflix or Youtube

Evaluation Metric

To keep things simple and because we have an even number of samples for each class in the dataset, I've chosen to use accuracy to measure the performance of the classifier, as any form of incorrect classification will reduce the utility of the tool.

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

Another factor to be taken into account will be the inference time needed to actually classify an individual image and provide feedback to the user.

Analysis

Data Exploration

The dataset has the following properties:

Number of classes	Total number of images	Training set size	Validation set size	Test set size
14	1728	1120	279	329

Each image has 3 color channels and a resolution of 244 x 244 pixels.

The training data is in a folder called "data", with the classes separated by folder. The test data is in a folder called "data_test" and uses a csv file called 'test.csv' to map the images to their true classes.

Exploratory Visualization

Figure 1. shows the class distribution of the combined training and validation set, each class has 100 images with the exception of the netflix class, which contains 99 images.

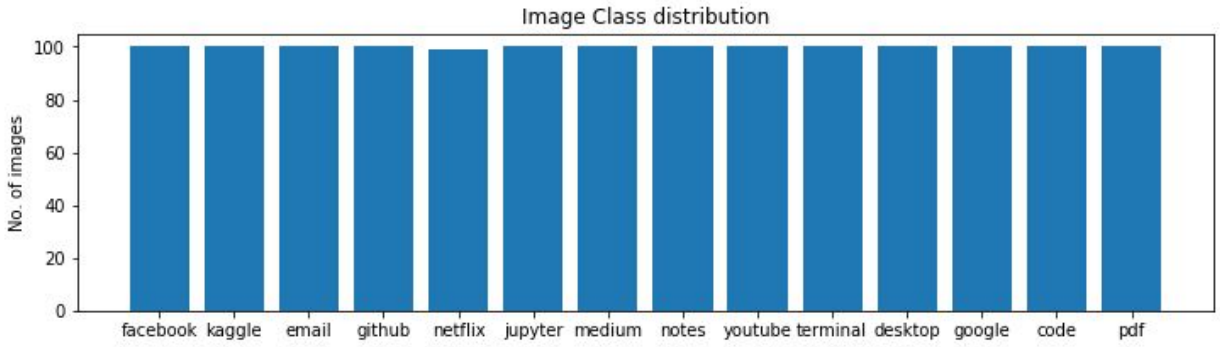
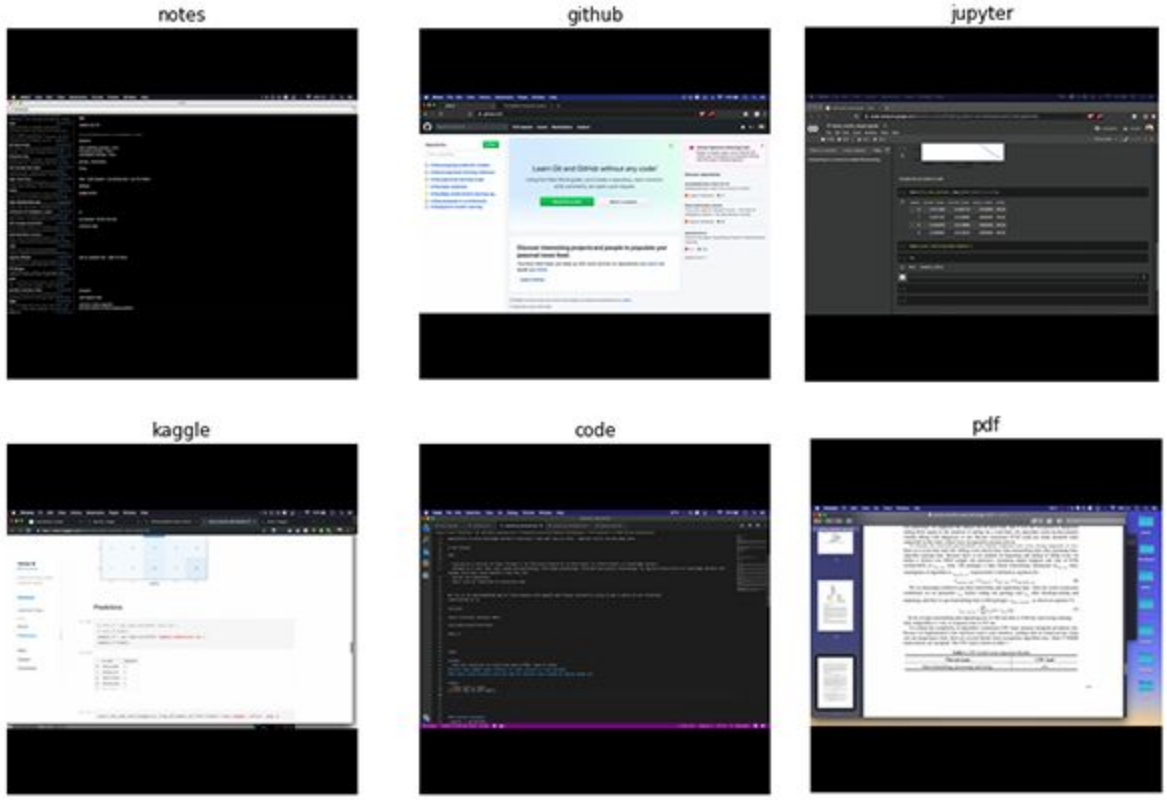
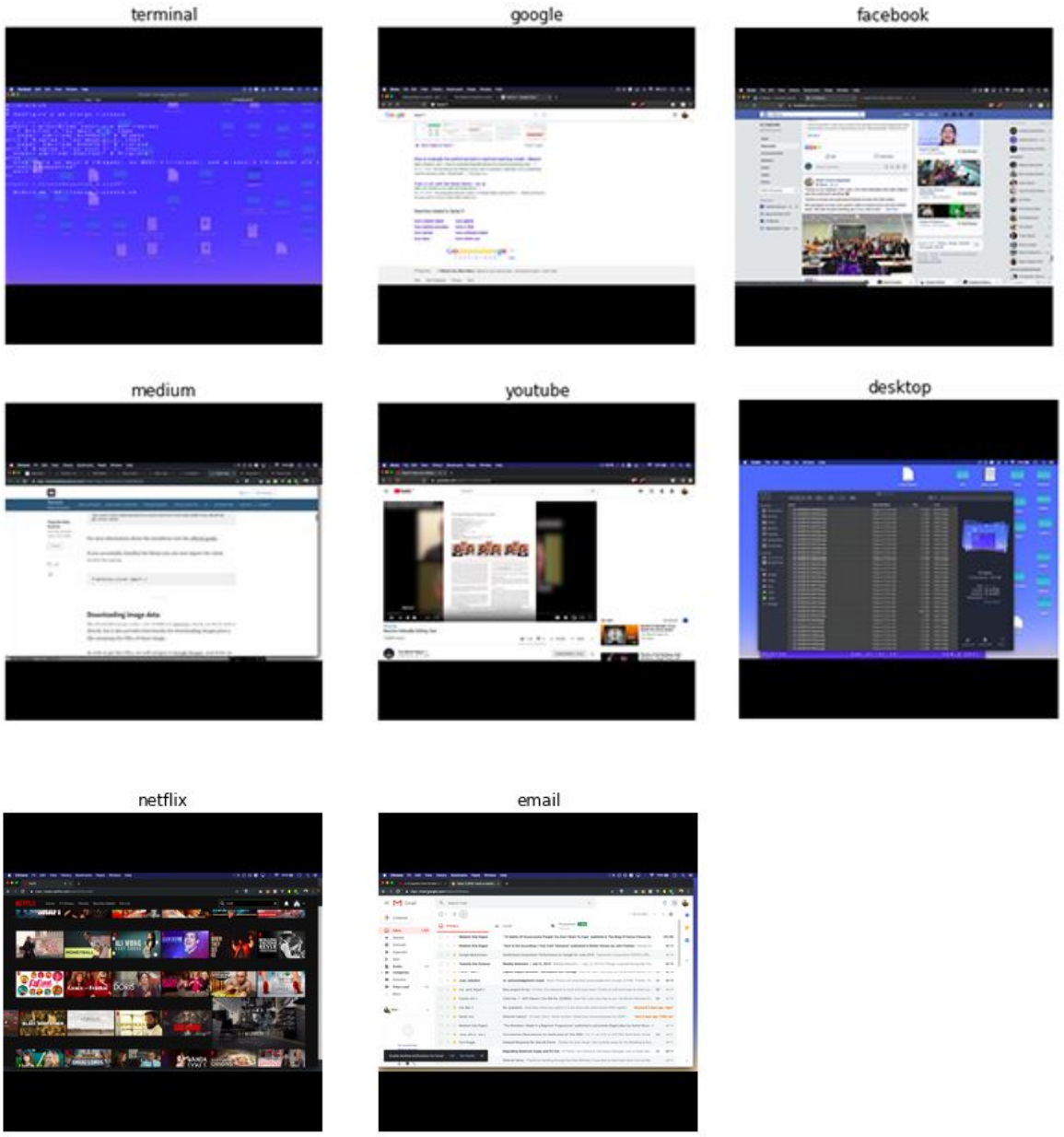


Figure 1.

After learning about findings from a study by Buda et al. which suggests that datasets with unbalanced classes have a negative effect on model performance [1], I opted to create a balanced dataset, given the relative ease of creating more data.

The figures below show samples of each dataset class.





Data class samples

Algorithms and Techniques

Transfer Learning

One shortcoming of using supervised learning to train neural networks for computer vision tasks is that they need vast amounts of training data to reach an acceptable performance.

Luckily there's a way to drastically reduce the required amount of training data from thousands, if not millions, of examples to only needing around a hundred training examples per class.

This technique, called transfer learning, takes advantage of pre-trained models that have already been trained on large datasets to learn a given task and re-purposes them to learn a new related task.

For this project and other image-classification tasks, the base model is commonly a CNN (Convolutional Neural Network) architecture, trained on the ImageNet dataset which has already learned to distinguish between a 1000 different classes.

CNNs are effective at computer vision, because they act as a layered feature detector that combines primitive detectors for e.g. lines and edges to create more complex detectors, for example by combining a horizontal line detector with a vertical line detector to create a corner [2]. Combining many layers of feature detectors eventually allows us to create CNN models that rivals humans at certain image recognition tasks. Figure 1. illustrates how simple feature detectors in the earlier layers of the model combine to detect more complex shapes as you add more and more layers.

Feature detectors in deep neural networks in CV have been observed to transition from general feature detectors in the earlier layers to task-specific detectors in the last layers [3].

Figure 2. illustrates the layers increasing in complexity as the network deepens.

Transfer learning works by taking the final layer of our pre-trained ImageNet model, which would have 1000 outputs (one output for each of the 1000 possible classes) and swapping it out with an output layer with 14 outputs, each representing our target classes and training ONLY the weights connecting to the final layer to distinguish between our target classes, when given a screenshot.

The idea is that the model has already learnt the primitive feature detectors for basic shapes in the earlier layers, with only the last "specialized" layer requiring tuning to detect the features specific to the target recognition task.

Differential Learning Rates

We can take this method a step further, by unfreezing all the network layers and "fine-tuning" the earlier layer groups in the model by training them with lower learning rates (relative the the learning rates used in later layers), thereby improving the earlier detectors to specialize on our given task as well[4]. Figure 3 shows how different layer groups are batched by learning rate.

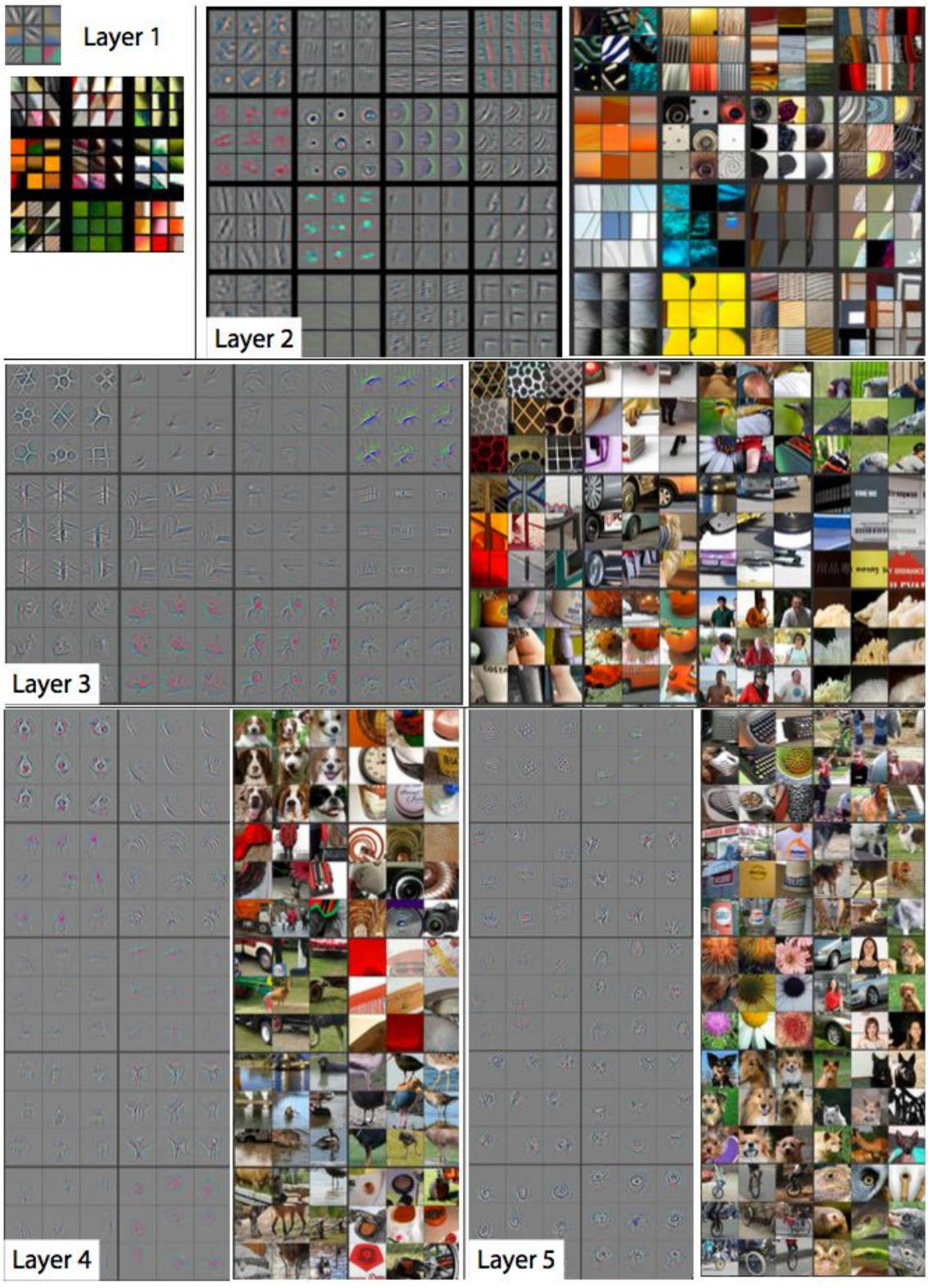


Figure 2. CNN layer visualization

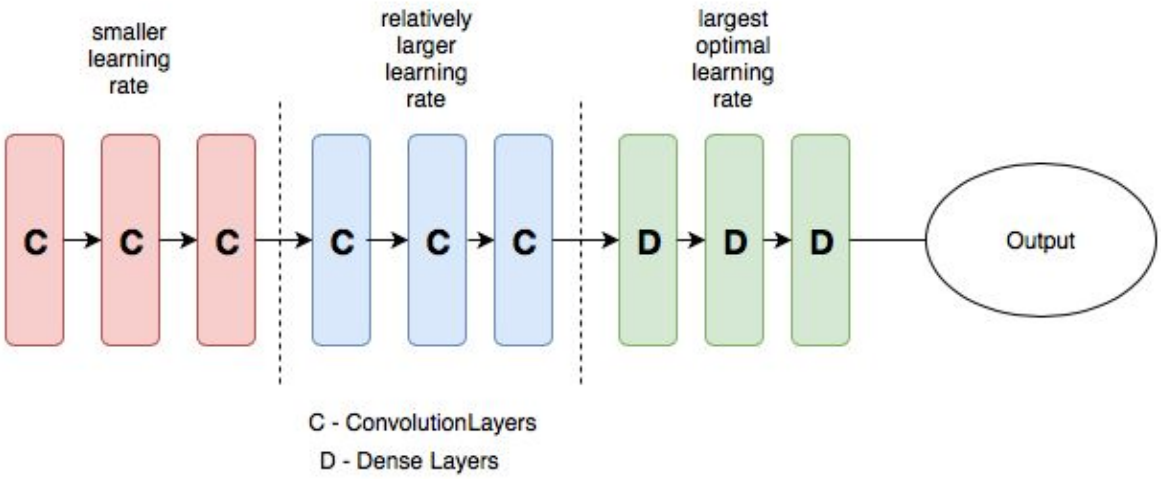


Figure 3. Differential Learning Rates

Model Architecture

After testing different model architectures and sizes like Inception, VGG and ResNet, I opted to use a ResNet34 architecture. The model was selected because it performed better than the larger ResNet50 and trained faster than the other models.

ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015 (ILSVRC2015) and secured 1st place at the 2015 MS COCO challenges for both detection and segmentation.

The architecture surpassed its predecessors Alexnet, GoogLeNet and VGG by taking advantage of so-called skip-connections to enable the training of deeper models, which was difficult if not impossible prior to the discovery of said skip-connections because of the vanishing gradient problem [5].

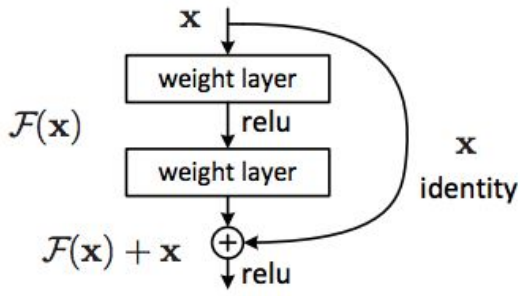


Figure 2. Residual learning: a building block.

Figure 4. ResNet block with skip connection

The ResNet architecture is made from stacked convolutional blocks [6], where each convolutional layer, has a skip connection that sends the inputs around the convolutional transformations at every other layer.

The idea behind ResNet is that increasing the network depth beyond a certain point no longer improves accuracy gains with regular 'vanilla' network architectures.

The reason being that back-propagating the gradients through deep networks becomes increasingly difficult when you add more layers, because repeated multiplication of gradients causes them to become smaller and smaller and eventually disappear [5].

Experiments from the original ResNet paper, as shown in Figure 5. , show that deeper “vanilla” networks with 56 layers perform worse than their 20 layer counterparts, returning a worse error after the same number of iterations for both the training and test sets.

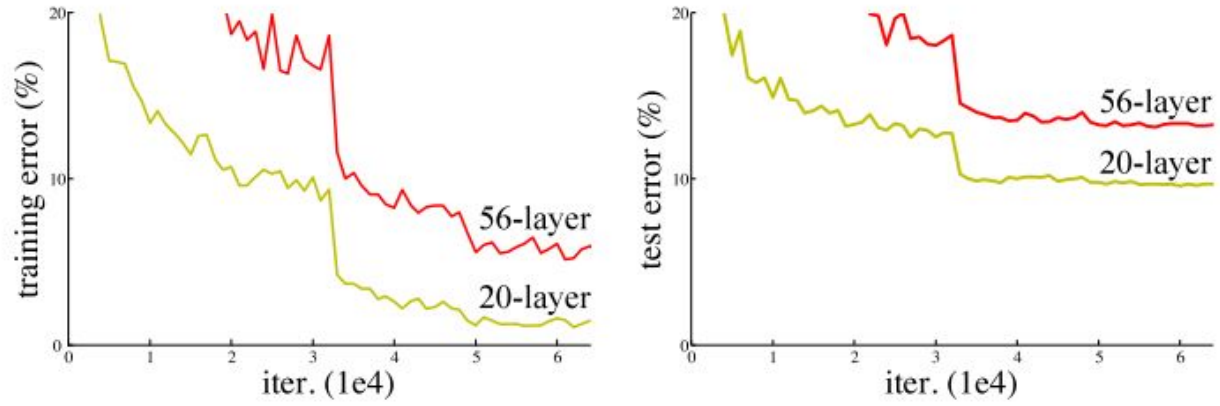


Figure 5. Comparison of 20 layer and 50 layer networks without skip-connections

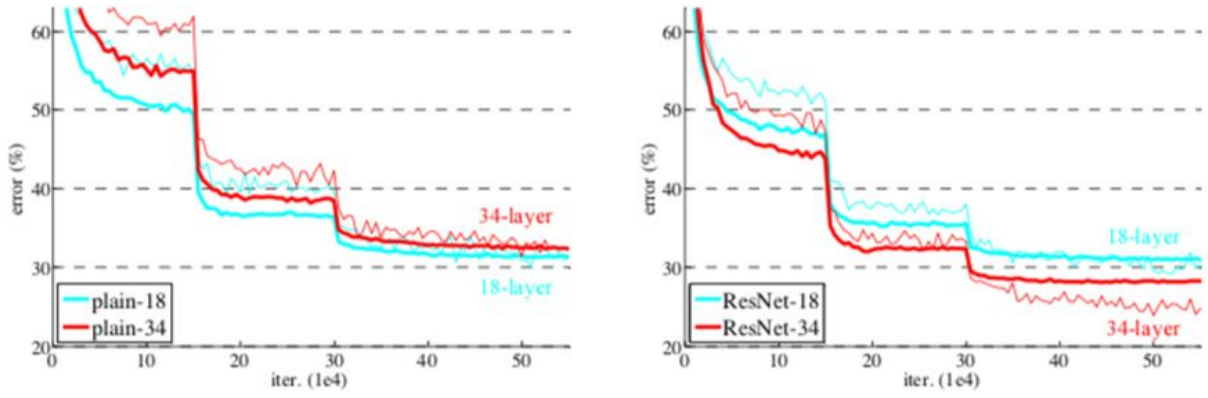


Figure 6. Vanilla networks (left) vs. ResNets (right)

Figure 6 shows results from the same paper with ImagNet training comparisons. The thin curves denote the training error and bold curves denote the validation error of the center crops.

On the left: vanilla networks of 18 and 34 layers, on the right: ResNets with skip connections of 18 and 34 layers. The graphs show that the deeper 34 layer ResNets with skip-connections return a lower(better) error than the 18 layer model, which is not the case for the “vanilla” networks [6].

Data Augmentation

Data augmentation is a method to improve model performance by artificially creating more data for the model to learn from.

This is done by making additional copies of the original images and transforming them by flipping, rotating, zooming or adjusting their brightness. This way a single image can be turned into multiple instances of the original, as long as the context of the image is maintained or the copy could just as well be an original.

When applied to pictures of everyday objects, people or animals many of these transformations make sense like slight rotations and horizontal flips, thus creating instances of the originals from different angles, yet flipping them upside down would destroy the context and be an example of what not to do.

Unfortunately, the nature of the project's dataset limits the possible data augmentation techniques that we can use. Training a model with rotated and flipped versions of the original screen data would teach the model examples of things it will never come across in production. The only data augmentation techniques that we can take advantage of, is to create minimally altered variants of the original data by changing the brightness and zooming in on the screens, to simulate different screen settings and zoomed windows.

Further techniques and methods to improve model performance will be discussed in the implementation and refinement section of this report.

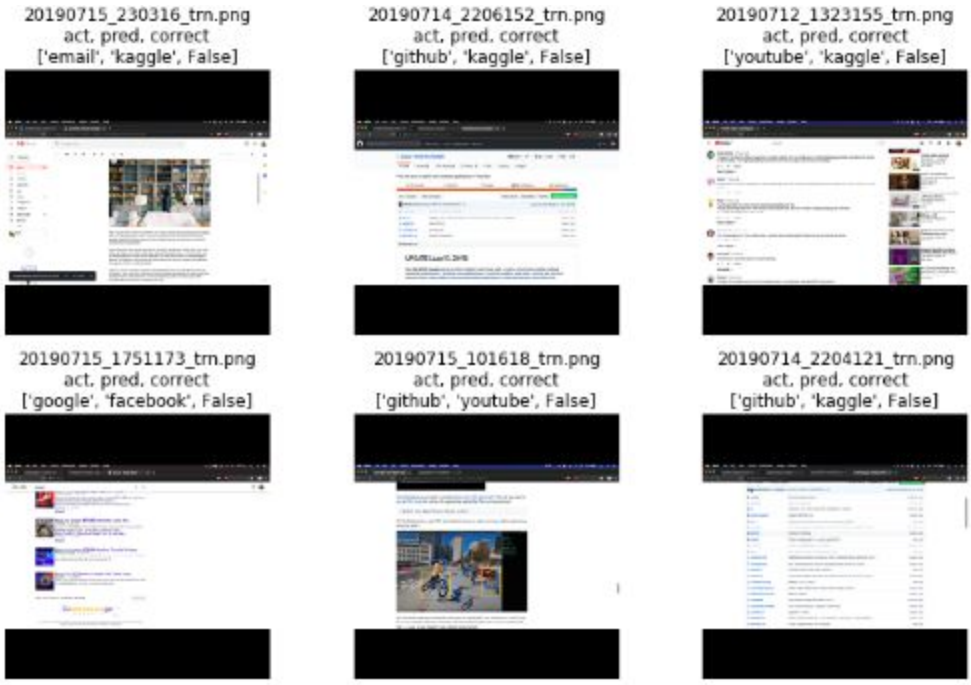
Benchmark

I have chosen to use a pre-trained Alexnet (trained on ImageNet) as the benchmark model for this project. The model which was trained for 4 epochs with a learning rate of 2e-2 and displayed an accuracy of 91.72% on the test set, misclassifying 27 out of the 326 test images.

epoch	train_loss	valid_loss	accuracy	time
0	1.825224	0.880822	0.899642	00:07
1	1.393214	1.140091	0.888889	00:05
2	1.012710	0.475178	0.946237	00:05
3	1.327065	0.741221	0.928315	00:05

The validation loss remains below the training loss, which is odd but could be due to the small training/validation set sizes and a short training cycle.

The image below shows a sample of the errors made by the benchmark model.



Benchmark model test set error examples

Methodology

Data Preprocessing

Prior to training, the images were randomized and segmented into the respective training, validation and test sets.

Aside from shrinking the input images from the original screen-resolution of 1440 x 900 to square 244 x 244 pixel images with 3 color channels and adding a black border for padding on the top and bottom of the image, no further transformations are made with the images.

These images are then normalized according to the imagenet mean and standard deviation, since the pre-trained model used for transfer-learning was trained with these stats.

Implementation

The implementation process was split into two phases:

1. Training a classifier using transfer learning
2. Building an application to make inferences with the classifier and provide user feedback

Phase 1: Classifier training

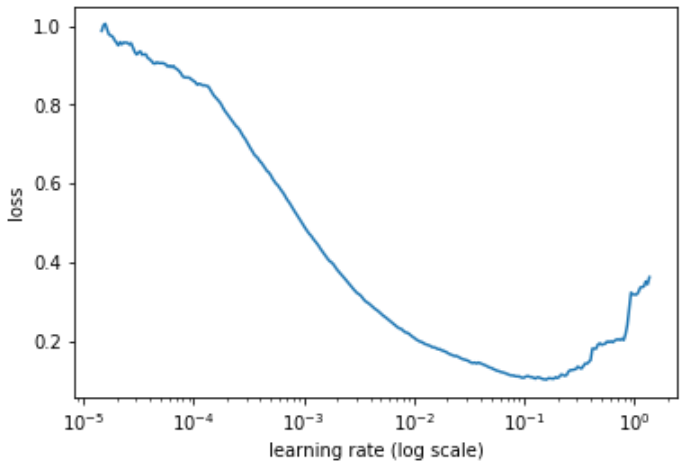
To improve on the initial Alexnet benchmark model we used two techniques that resulted from Leslie Smith's research called:

1. learning_rate finder
2. One cycle policy

These methods help us train the model with a higher than regular learning rate, which surprisingly acts as a regularization method and lead to better overall performance in less time [7].

Learning rate finder

The learning rate finder works by training the model over the duration of 1 epoch, starting at a very low learning rate e.g. 1e-8 and gradually increasing the learning rate over time, multiplying it by a given factor at each mini-batch, until it reaches a value of 1, whilst recording the loss at each timestep. We then plot a smoothed version of the loss (in this case an exponentially weighted moving average) against the learning rate, which is illustrated below.



Smoothed learning rate vs. loss over 1 epoch

One can observe the loss gradually decreasing until it hits a minimum and then rapidly increasing.

The method in the paper proposes that the ideal learning rate is about a factor of 10 lower than the minimum, as the values near the minimum may be too close to the point at which the model loss increases [8].

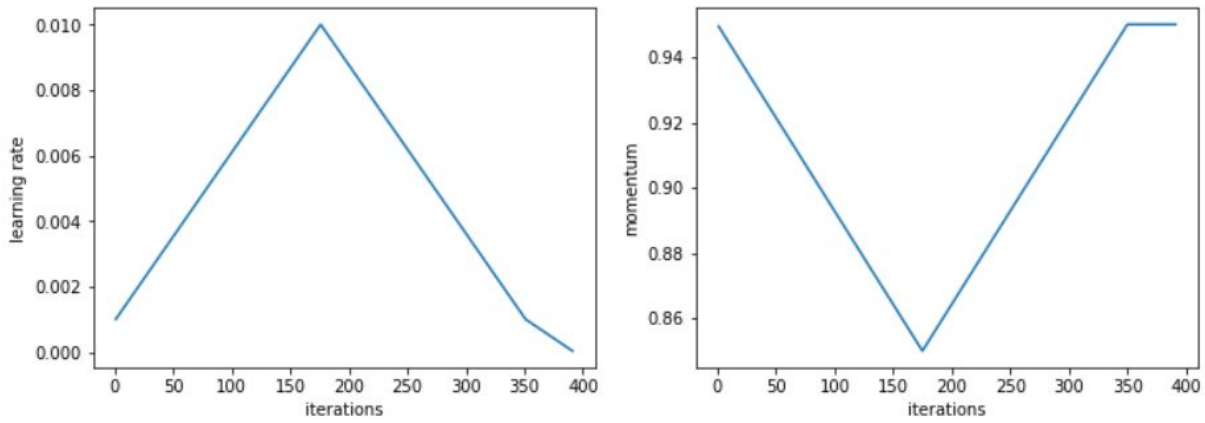
One cycle policy

The second technique refers to how we adjust the learning rate during a training cycle.

The idea is to start with a small learning rate at the beginning of the cycle and to gradually increase the value for half the training cycle, until it reaches a maximum, (which we set to the learning rate discovered by the learning rate finder) and then to gradually decrease the rate from the maximum to a rate even lower than the starting rate (to let the model hone in on the multi-dimensional minimum with a very, very small learning rate at towards the end of the cycle).

In addition to increasing the learning rate for half the cycle and decreasing it for the other half, we adjust the momentum i.e. the rate at which the learning rate increases or decreases in the opposite manner, meaning that we start with a high momentum at the start of the training session, then gradually reduce it so that it reaches its minimum after half the period and then increase it until it reaches it's starting rate for the remainder of the period. The paper suggests a range between 0.95 and 0.85.

The figure on the bottom illustrates graphs of the learning rate *left* and the momentum *right* over the training period.



Learning rate and momentum for One cycle policy

Training the classifier

The training methods used in this project leverage the, Pytorch powered, fastai library which offers a suite of very helpful tools to train, interpret and deploy deep learning and machine learning models.

Figure 7. Shows the graph returned by the learning rate finder mentioned earlier, which suggests that we should use a learning rate with a magnitude of 1e-2.

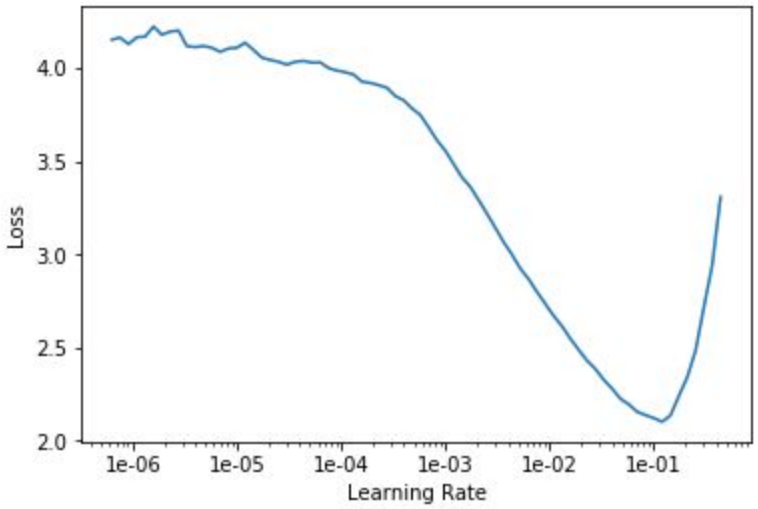


Figure 7. Learning rate finder output

The first iteration of the model was trained with the One cycle policy, a max learning rate of 1e-2, a batch size of 6 and the library's default regularization techniques: using dropout value of 0.5 and no weight decay.

After training for 5 epochs it reached a training accuracy of 94.26% on the test set and correctly classified 307 images out of the 326 images (8 errors less than the Alexnet benchmark).

Refinement

The model showed an improvement on the Alexnet Benchmark and was improved further by tweaking the learner's hyperparameters and regularization methods, specifically dropout and weight decay.

Weight decay penalizes model complexity by adding a scaled (squared) sum of the weights to our loss function, thereby penalizing the model if the weights are large if they don't need to be.

Dropout stops the model from overfitting by "forcing" model to learn real patterns from the data classes as opposed to traits specific to the training data. This is done by randomly turning off (hence the name) a percentage of the model's units and their connections during training [9].

After testing different values batch size, learning rate, dropout and weight decay, the model with the following parameters showed the best performance on the test set:

Epochs	Batch size	Weight Decay	Dropout
5	21	3e-7	0.25

The base model was then improved further by unfreezing all the model layers and training the layer groups with a max. learning rate of 1e-7.

Results

Final Model performance

These results reflect some findings from Leslie Smith's paper on Superconvergence, which suggests that training with a very high learning rate has a regularization effect on the learner in and of itself. To balance this effect other regularization methods need to be reduced to improve performance [7].

The final model returned an accuracy of 97.24%, only misclassifying 9 of the 326 test set images, making only a third of the initial benchmark's 27 errors (91.72% accuracy).

The weights of this model were then downloaded for use by the application.

Phase 2: Application development

The second phase of the project involved creating a script that can be run by entering the command: 'python timeNet.py' into the terminal after navigating to the timeNet folder.

The application was built using the help of 3 Python libraries:

- Fastai, for inference with our trained classifier
- PyAutoGui for taking screenshots
- Pync, which takes the model output and feeds it to Mac's notification center to display the results in small semi-transparent box of the screen's upper right-hand corner.

The application takes a screenshot every 10 seconds over a time period of 30 minutes.

After every screenshot is taken it is fed to the classifier, which in turn returns a string with the predicted class and adds the class to a list.

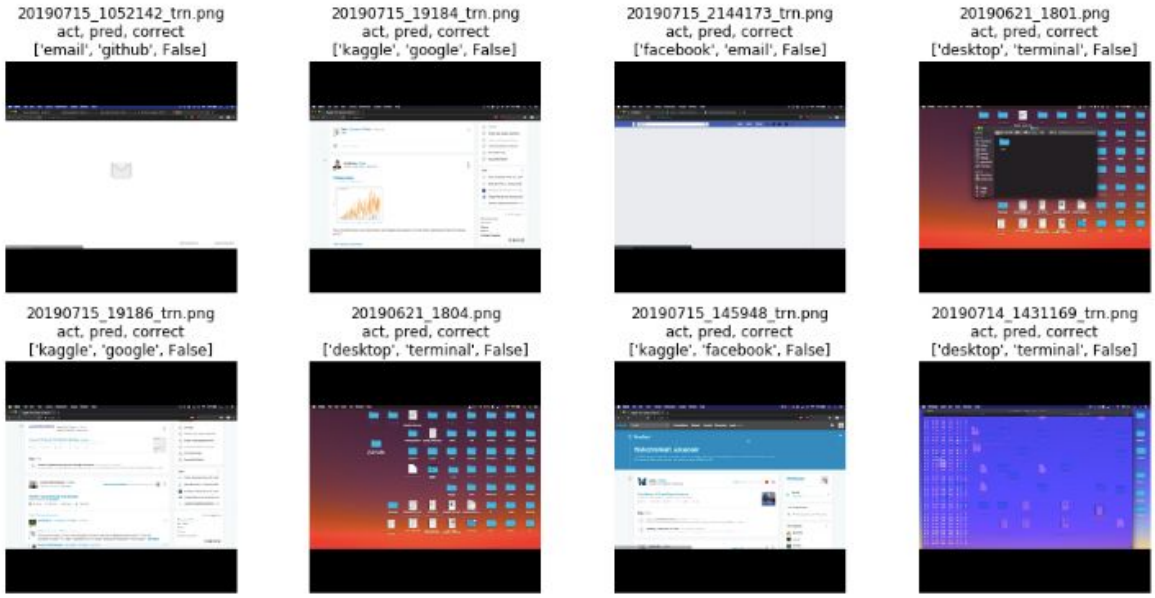
After the recording period the application returns a percentage breakdown of the user's screen activity using Mac's built in notification functionality and Python library called pync.

The application also makes a warning notification (with sound) if the model returns more than 3 classes that belong to “distracted” activities, in this case if they are labelled either Youtube, Facebook or Netflix.

The code can be found in the folder “timeNet” folder.

Conclusion

Free-Form Visualization



Final Model Error Samples

The image above shows examples of the final errors made by the model, which tends to confuse Google with Facebook and Kaggle, possibly due to the similarities in their user-interfaces.

The model also confuses desktop with terminal activities, likely because the terminal screens used to train the model have a semi-transparent window, causing desktop and terminal images to look very similar.

It appears like the application will need larger quantities of data and more varied data.

Reflection

Reflecting upon the process of building this project, it was a great learning experience and practice that forced me to dive deeper into concepts, techniques and ideas related to deep learning and computer vision.

I was surprised to learn about the extent to which batch-size influenced the performance of the model, when compared to changing other hyperparameters. My initial thinking was that a small batch size would be more suitable given the small dataset sizes, yet higher batch-sizes ultimately performed better.

I also recognized that the application, in its current form, would not be viable for use by others, because the training data is very specific to my own behaviors. For personal use over the last few days it's been somewhat helpful but still has a few iterations to go before I'd be happy with it.

Improvements

Over the next couple of iterations I intend to implement the following:

1.) Create a function that provides a summary on time spent over longer time periods, for the day, the week, or a customizable time period etc.
2.) Re-train the model on more data and more varied data, I imagine it would be possible to implement a form of data-augmentation using the PyAutoGui library,(used for taking the screenshots in this project), and creating a script that automates e.g. by taking pictures of desktops with different background pictures and colors, and simulating the use of other popular applications
3.) A way to have the model return and save the prediction and image for uncertain predictions, so these can be used to iteratively train the model to improve on shaky predictions
4.) I would change my initial method for collecting the initial training data using an autoencoder to find and label similar images (something similar to the one in this article: https://towardsdatascience.com/building-a-similar-images-finder-without-any-training-f69c0db900b5)
5.) Lastly it would be nice to find a way to enable users to train their own activity trackers using some form of semi-supervised learning.

References

[1] Mateusz Buda, Atsuto Maki, and Maciej A Mazurowski. A systematic study of the class imbalance problem in convolutional neural networks.

[2] Matthew D Zeiler, Rob Fergus: Visualizing and Understanding Convolutional Networks

[3] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014 : How transferable are features in deep neural networks? In Advances in neural information processing systems. pages 3320–3328

[4] Manikanta Yadunanda: Transfer Learning using differential learning rates

[5] X.Glorot and Y. Bengio: Understanding the difficulty of training deep feedforward neural networks. In AISTATS, 2010.

[6] He et al: Deep Residual Learning for Image Recognition

[7] L. Smith and Nicholay Topin: Super-convergence: Very Fast Training of Neural Networks Using Large Learning Rates

[8] Leslie Smith: A Disciplined Approach to Neural Network Hyper-parameters: Part 1 - Learning Rate, Batch Size and Weight Decay

[9] Hinton et al: Dropout: A Simple Way to Prevent Neural Networks from Overfitting