Where am I? Predicting Montreal Neighbourhoods from Google Street View Images

Code available at: `https://github.com/slflmm/Miniproject-4`

COMP 598 – Miniproject 4

Stephanie Laflamme
260376691

Benedicte Leonard-Cannon
260377592

Sherry Ruan
0000

ABSTRACT

1. INTRODUCTION

To a local resident, neighborhoods of the island of Montreal bear distinctive traits. For instance, Old Montreal is characterised by its historic buildings, Downtown Montreal by its skyscrapers and Westmount by its opulent houses. To the experienced human eye, it is usually possible to pinpoint the district at which a given picture was taken, granted high enough resolution. Can we say the same of machine learning algorithms?

Automatic classification of images by provenance would have cultural and social value; tourists could easily identify the source of their photos, animation studios might use these classifiers to reconstruct more vivid and evocative scenery [4], and, optimally, could help in investigations where photographs of unidentified locations might prove crucial.

We attempt to determine whether machine learning algorithms can be trained to differentiate between nine visually distinctive neighbourhoods on the island of Montreal based on Google Street View images. We tackle this classification problem with three models: a perceptron (as a baseline algorithm), a stacked denoising autoencoder (for neural network pretraining) and a convolutional neural network. We place the performance of these classifiers in context by comparing them with human results.

A classification task that is similar to our own was performed by researchers at Carnegie Mellon University [4]. The purpose of their study was to assess how cities are uniquely defined by their visual appearance. Their approach focused on extracting unique architectural characteristics from major cities in the United States, Europe and Japan, while paying particular attention to Paris. Given labeled images acquired from Google Street View, they developed a method for automatically extracting features that are recurrent and geographically distinctive within each city. Their approach relied on discriminative clustering to extract visual cues. The researchers observed that small architectural elements such as window shapes, street signs and lamp posts tend to differentiate cities more significantly than their landmarks, such as the Eiffel tower. It is worth mentioning that this method was also applied at a smaller scale: in a vein more similar to our own prediction task, an attempt was made to distinguish between districts of Paris on the basis of architectural features.

Another similar study by Knopp et al. tackled the problem of place recognition using images of Paris acquired from Google Street View and Panoramio, a photo-sharing website [7]. Their objective was to automatically extract from a database a set of images representing the same location as that of a query image, regardless of the viewpoint and lighting conditions. Their place recognition algorithm uses a visual bag-of-words approach combined with the filtering of confusing features such as trees or cars.

2. DATASET

2.1 Description

We consider nine of Montreal's most distinctive neighbourhoods as classes, listed in Table 1. The dataset consists of 72,000 greyscale images of size 100×100, with 8000 images per neighbourhood. The data is subdivided into a training set of 64,800 images and a test set with the remaining 7,200 images.

Table 1: Class labels and corresponding borough

Label	Name
0	Downtown
1	Old Montreal
2	Chinatown
3	Gay Village
4	Plateau
5	Outremont
6	Westmount
7	Hochelaga
8	Montreal-Nord

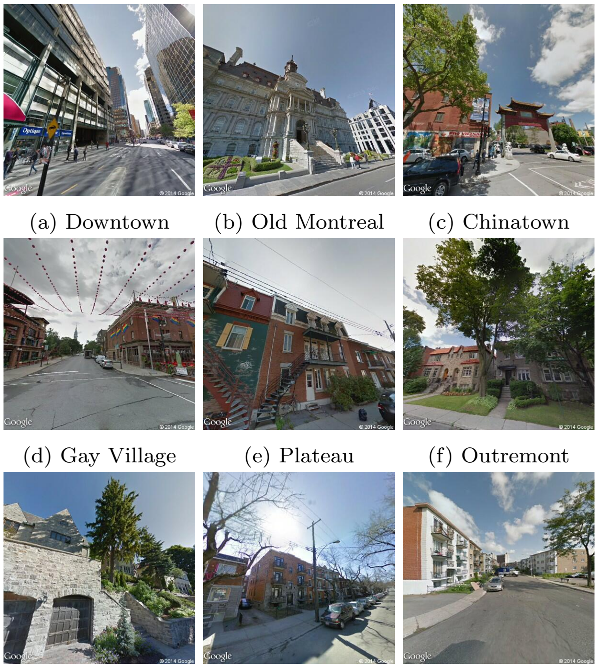
2.2 Acquisition

We use the Google Street View API [1] to collect images from the neighbourhoods of interest. The API allows users to download an image based on its geographical coordinates, and to define 'heading' (the camera's orientation in the x-y plane), 'pitch' (the camera's orientation in the z plane, relative to the ground), as well as the field of view and size of collected images.

Data from the City of Montreal [2] was used to identify the delimiting streets of our nine neighbourhoods. We then match their intersections to geographical coordinates (latitude and longitude) to obtain the vertices, in geographical coordinates, of polygons surrounding the neighbourhoods. Since the API enables us to poll for images based on geographical coordinates and that such coordinates are known for each neighborhood, labeling the images becomes a trivial concern.

We collect 400×400 images with a field of view of 100. For each image, the heading is set to a random number between 0 and 360, and the yaw is randomly selected between 0 and 35 degrees. Figure 1 shows a sample image from each neighbourhood. When there is no Street View panorama available at a geographical coordinate, the API returns a blank image; we identify and discard them online, as images are collected.

Once collected, images are downsampled from their original size to 100×100 pixels and converted to grayscale; larger datasets did not fit into GPU memory.



(a) Downtown (b) Old Montreal (c) Chinatown

(d) Gay Village (e) Plateau (f) Outremont

(g) Westmount (h) Hochelaga (i) Montreal-Nord

Figure 1: Sample images collected

2.3 Preprocessing

As a preprocessing step, images are contrast-normalized. Given a 100 × 100 image x from the dataset, we let

$$x = \frac{(x - x_{min})}{x_{max} - x_{min}},$$

where x_{max} and x_{min} are the maximum and minimum values in x's pixels, respectively. This step ensures all images have the same contrast range, with pixel values ranging between 0 and 1.

3. CLASSIFIERS

3.1 Human experiment

To obtain a human performance baseline in comparison to machine learning methods, we conduct a small scale experiment with human participants. Five people who are familiar with the Montreal area are asked to label 50 test images of the same format as those used by our classifiers (100 × 100 pixels, grayscale and contrast normalized). Their predictions are combined to get human labels for 250 images.

3.2 Perceptron

The perceptron arose from attempts at creating a machine that could recognize images.[2] Inspired from neuroscience, the perceptron is made up of a single layer of 'neurons'; the output of a neuron is the weighted sum of all inputs, passed through an activation function which ensures neurons have binary outputs. As a result, perceptron can only learn linearly separable patterns; it cannot, for instance, encode the XOR function, as this requires one more layer of neurons. As such, we implement a multiclass perceptron as a baseline classifier.

Given n examples with m features each, and given k classes, the perceptron learns a matrix W of $(m + 1) \times k$ weights (one weight for each feature-class combination, and a bias vector) by gradient descent on the error

$$Err(x) = (y - f(W^T x)),$$

where f is the Heaviside step function, x is an example, and y is its class.

3.3 Stacked denoising autoencoder

We evaluate the performance of a Theano-based stacked denoising autoencoder (SDA) implementation on our dataset [3]. Stacked denoising autoencoder have shown to provide significant gains in accuracy at certain image classification tasks over other models such as SVMs and DBNs [12].

An autoencoder is a feedforward neural network that is given a specific objective function; this model is trained to faithfully reproduce the input $\mathbf{x}$ at the output layer [9]. An autoencoder comprises three layers: an input layer, a hidden layer and an output layer. The objective is to find the parameter weights that can represent $\mathbf{x}$ at the hidden layer and reconstruct it at the output layer. We are interested in the contents of the hidden layer once the network has been trained. The contents of the hidden layer vary depending on the amount of neurons it contains compared to the input layer; with fewer neurons (undercomplete representation), the hidden layer compresses the input and with more

[1] https://developers.google.com/maps/documentation/streetview/

[2] http://ville.montreal.qc.ca/portal/page?_pageid=5798,85813661&_dad=portal&_schema=PORTAL

[3] http://deeplearning.net/tutorial/SdA.html

neurons (overcomplete representation), it extracts relevant features.

To find the hidden representation, $\mathbf{h}(\mathbf{x})$, and the output $\hat{\mathbf{x}}$, we perform forward propagation much like in a standard neural network. Forward propagation can be divided into two components: an encoding step,

$$h(\mathbf{x}) = sigmoid(\mathbf{b} + W\mathbf{x})$$

where we transform the input into its hidden representation and a decoding step,

$$\hat{\mathbf{x}} = sigmoid(\mathbf{c} + W^{*}\mathbf{h}(\mathbf{x}))$$

where we restore the input from this same latent representation [9]. We often rely on tied weights, meaning that the same weight matrix is used at both layers: W^{*} is in fact W^{T}.

The training objective of the autoencoder is to minimize the difference between the input $\mathbf{x}$ and the output $\hat{\mathbf{x}}$. In order to do so, an option is to minimize the cross entropy loss between the two,

$$l(\hat{\mathbf{x}}) = -\sum_{k} x_k \log \hat{x}_k + (1 - x_k) \log (1 - \hat{x}_k)$$

To optimize the values for $\mathbf{W}$, $\mathbf{c}$ and $\mathbf{b}$ we compute their gradients and backpropagate them by gradient descent.

In our implementation, we use a denoising autoencoder. Simply put, the denoising process consists of setting some of the elements of each input $\mathbf{x}$ to 0 with probability p [11]. We can achieve this by applying a Bernoulli mask to the input data. Then, we perform forward and backward propagation normally. However, we must keep in mind that the output $\hat{\mathbf{x}}$ must be optimized according to the noiseless input $\mathbf{x}$. Interestingly, denoising autoencoders tend to generate a better latent representation since they are more robust to “partial destruction of the input” [11].

Classification cannot be performed with a denoising autoencoder alone. A common technique consists of stacking pretrained hidden layers of denoising autoencoders in order to build a traditional neural network. More precisely, we train an autoencoder, then connect its hidden layer to the input of the next one and so on. We train each autoencoder sequentially from the bottom up, while keeping the parameters of the previous ones fixed. Once pretraining is complete, we add an output layer to the model and we finetune this newly formed neural network using ‘standard’ gradient descent. By pretraining a neural network in such a way, we improve its capability to generalize since the parameters are initialized to values that are closer to the global optimum than randomly initialized ones [12].

3.4 Convolutional neural network

Although fully-connected stacked auto-encoders tend to perform very well at machine learning tasks, their ability to classify images is sensitive to position, size, and distortions–they can be poor at identifying invariant patterns. In the early ‘80s, Fukushima introduced convolution layers to remedy that problem by handling geometrical similarities regardless of position [5]. A convolution applies a filter to an image; a convolution layer contains many such filters, whose values are learned as the neural network is trained. This serves as an implicit, learned feature extraction at the start of the neural network, typically followed by hidden layers.

We then outline a backpropagation update of 2-dimensional convolutional neural networks [3], following the same notations we used before (where n for the number of examples, m for the number of features, and k for the number of classes). In addition, we use $t_{i,j}$ to represent the jth output layer unit corresponding to the ith input example and $y_{i,j}$ to represent the actual jth output layer corresponding to the ith example.The squared-error loss function E is given as

$$E = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{k} (t_{i,j} - y_{i,y})^2$$

. We can then easily derive the derivative of E with respect to the weights W with the help of the backpropagation.

$$\frac{\partial E}{\partial W_l} = x_{l-1}(\delta_l)^T$$

where the subscript denotes the lth layer. Now, with the involvement of convolutional layers and subsampling layers, we get the following recursive equation: $\delta_l = c_l(f(u_l) \circ up(\delta_{l+1}))$ where c is a constant. f can be chosen as a sigmoid function and $u_l = W_l x_{l-1} + b_l$.

As convolutional neural networks tend to outperform other methods at image classification, we implement our own convolutional neural network and train it using minibatch gradient descent. Our implementation is optimized for time-efficiency and for maximizing accuracy and reducing overfitting.

Given the vast amount of time required to train a convolutional neural network, we implement ours to run on a GPU. This reduces the training time by approximately a factor of 10. We use Theano[1], a symbolic Python library that compiles computation-heavy functions in C and can execute them on a GPU. To further improve performance, we use the cuda-convnet implementation of convolutions [8], which offers a 3-fold speed improvement relative to Theano’s convolution function.

Our convolutional neural network uses rectifier linear units (ReLUs) as activations, written

$$f(x) = \max(0.0, x).$$

This activation function is known to perform better than bounded activations (e.g. tanh) in deep neural nets for image-related tasks, presumably because they preserve intensity information about incoming information [10].

We also handle the overfitting that tends to happen in neural networks with many layers; there may be many complicated relationships between examples and their labels, and enough hidden units to model these relationships in multiple ways. Hinton et al. [6] introduced the concept of dropout to alleviate this issue. During training, hidden units are randomly omitted with probability p, which helps prevent complex co-adaptations on training data. We apply dropout to our convolutional neural network as Krizhevsky et al. [8] did–that is, the fully-connected hidden layers are subject to dropout.

4. HYPERPARAMETER SELECTION

4.1 Perceptron

Using a cross-validation procedure with the training set, we perform gridsearch over two parameters; the learning rate and the number of training iterations. We consider values between 0.0001 and 0.1 for the learning rate α, and values between 10 and 35 for the number of iterations. We record the validation and training errors for each hyperparameter combination. We then use our best parameter settings according to the validation error to train the perceptron on the whole training set, and measure its error on the test set.

4.2 Stacked denoising autoencoder

We greedily search over several parameters: the pretraining learning rate, the number of pretraining epochs, the fine-tuning learning rate, the hidden layer count and sizes, the level of noise in each layer, and the L1 and L2 regularization coefficients. For each parameter setting, the network is trained for 120 epochs. We record the best validation error obtained, as well as the training error achieved at the same epoch. Finally, we save the predictions of our best model and compute its error against the test set.

4.3 Convolutional network

As we do for the stacked autoencoder, we perform greedy search over the space of hyperparameters. We consider as parameters the learning rate, the number of convolutional layers, the sizes of filters in each layer, the amount of downsampling applied after each layer, the number of hidden layers, the number of hidden units per hidden layer, and the number of training epochs. For each hyperparameter setting, we keep track of the best validation error and its corresponding training error. Once again, we save the test set predictions of our best model to calculate the testing error.

5. RESULTS

We report results for our three classifiers and human labeling. In general, convolutional neural networks outperform the stacked autoencoders, which in turn exceeds the baseline results from the perceptron.

5.1 Human experiment

Human labelers had error rates ranging from 66% to 80%, with a mean of 71.6% error. Figure 2 shows the confusion matrix of their combined predictions. Generally, humans are good at identifying images from the Downtown neighbourhood, as well as those from Westmount. Moreover, they succeed at separating residential areas from commercial areas; they tend to confuse the residential neighbourhoods with each other (Westmount, Outremont, Montreal-Nord, Hochelaga), and predict images from more touristy areas (Downtown, Chinatown, Gay Village, Old Montreal) as being taken in the Downtown area.

5.2 Perceptron

The cross-validation procedure for the perceptron yielded validation accuracies hovering between 12-14%. Figure 3 shows the effects of varying the learning rate α and number of training iterations on cross-validation accuracy; the optimal configuration achieved 13.99% accuracy, with $\alpha = 0.001$ over 35 iterations.

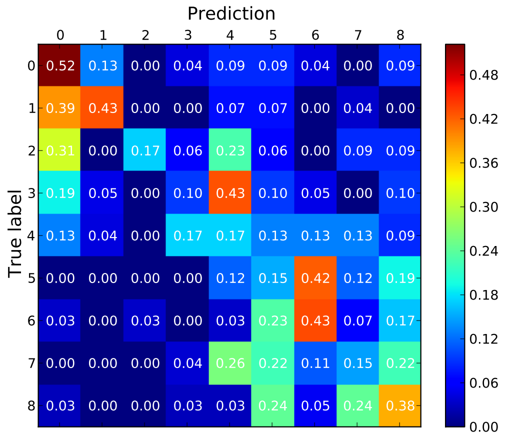


Figure 2: Confusion matrix for human labelers

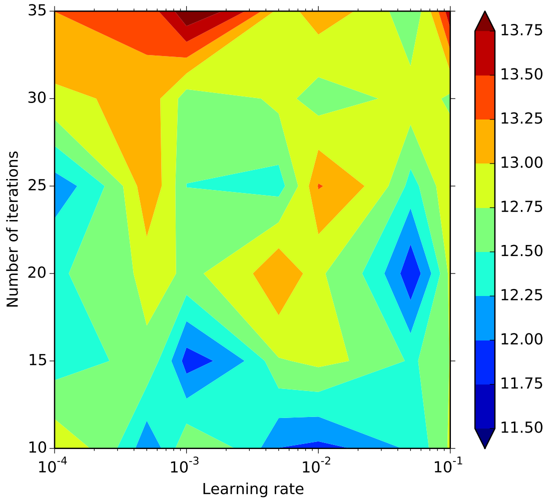


Figure 3: Cross-validation results for the perceptron

Training with these optimal hyperparameters led to an accuracy of 13.88% on the test set, a result which is only slightly better than a random classifier (which has an accuracy of $1/9 = 11.\bar{1}1$). Figure 4 shows the confusion matrix over the test set. Interestingly, it seems that the perceptron obtained slightly better-than-random results by simply over-predicting three classes (Downtown, Outremont, and Montreal-Nord).

5.3 Stacked denoising autoencoder

We performed validation experiments over a set of hyperparameters outlined in Table 2 in the Appendix. Figure 5 empirically illustrates the results of these experiments. In general, the validation error decreases steadily until epoch 40, at which point it oscillates between values of 0.75 and 0.65. Notably, the validation error during experiment 9 fluctuates more than for other experiments; this may be due to its much larger number of hidden units per layer. Indeed, while other classifiers had 500 hidden units per layer, this

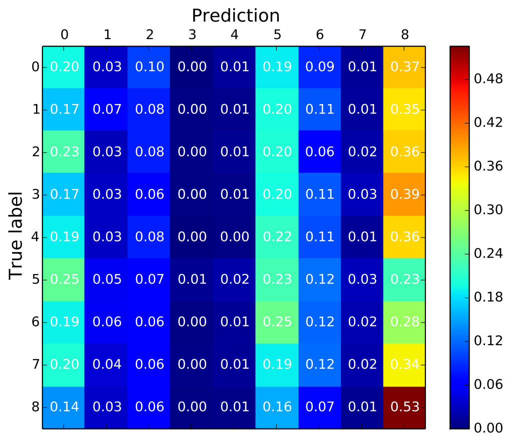


Figure 4: Confusion matrix for the perceptron

one boasted 1500.

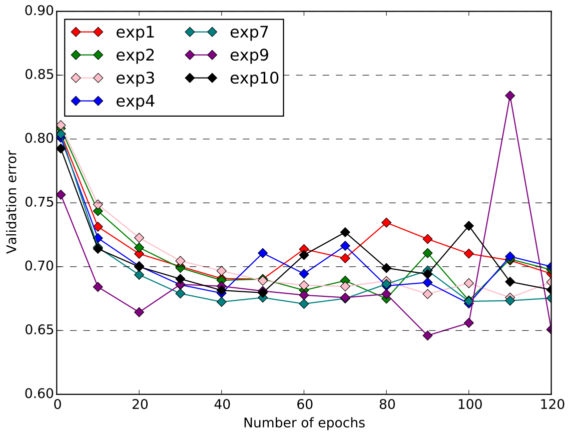


Figure 5: Validation error of stacked autoencoder experiments

Nonetheless, the hyperparameters used in experiment 9 produced our best validation results for the stacked denoising autoencoder, with a validation error of 64.61%. Figure 6 shows a comparison of its training error and validation error over the first 120 epochs. Note that while the validation error plateaus at a high value, the training error continues to decrease steadily toward 0; it is clear that this classifier overfits.

Our best stacked denoising autoencoder gave 66.09% error on the test set. As can be seen from the confusion matrix in Figure 7, the autoencoder does moderately well at identifying images from Chinatown, Old Montreal, Gay Village, Outremont, and Montreal-Nord, but is very poor at classifying images from the other neighbourhoods, especially those from the Plateau.

5.4 Convolutional neural network

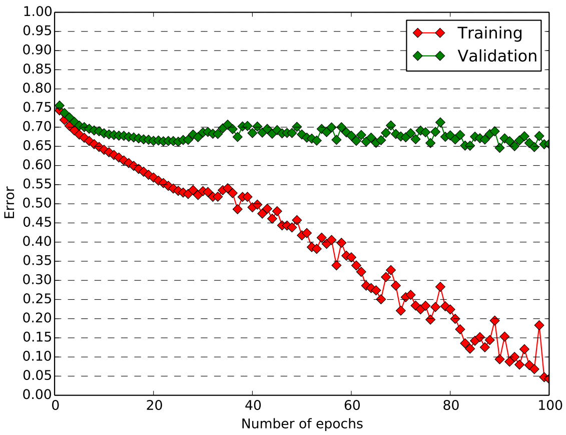


Figure 6: Training error vs validation error for the autoencoder

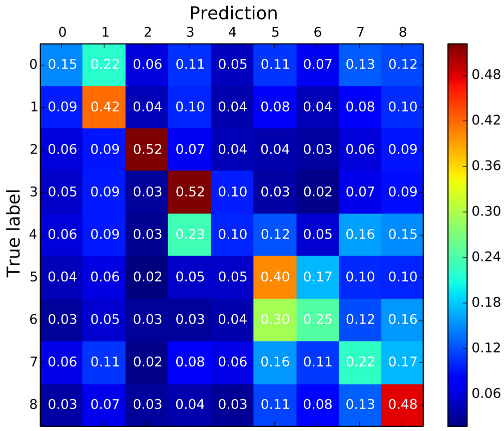


Figure 7: Confusion matrix for stacked denoising autoencoder

Table 3 in the Appendix lists the validation experiments performed with a convolutional neural network, with their hyperparameter settings and validation errors. Figure 8 shows results from a subset of those experiments; observe that experiments seem to reach the beginning of a plateau by epoch 80.

Our best validation results were obtained in experiment 4, with a validation error of 42.82 after 200 epochs. Figure 10 shows a comparison of the validation error vs the training error over time. One can see that the validation error reaches a plateau while the training error continues to decrease; the convolutional neural network overfits somewhat, but not so dramatically as the stacked denoising autoencoder.

Our best convolutional neural network obtained 42.61% testing error. The confusion matrix, shown in **??** demonstrates that the classifier is best at identifying images from Chinatown and the Gay Village. On the other hand, it completely fails to correctly classify images taken in the Plateau, mis-

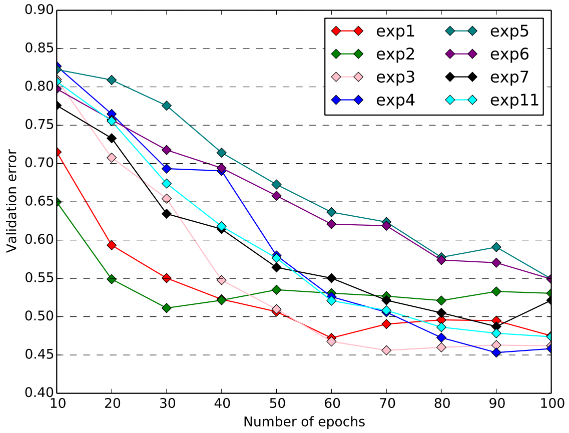


Figure 8: Validation errors of convolutional neural network experiments

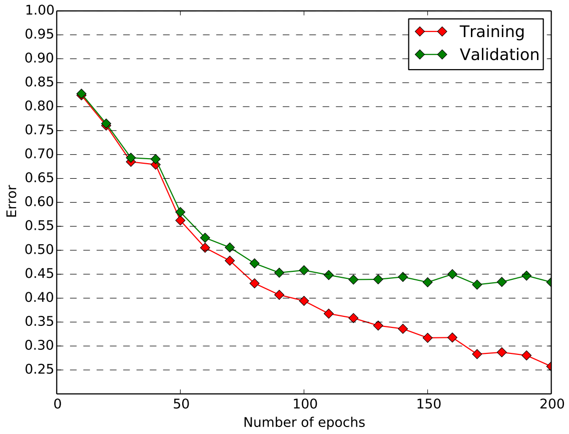


Figure 9: Training error vs validation error for the convolutional neural network

classifying them as taken in the Gay Village. The classifier shows a moderately good performance for the other neighbourhoods.

6. DISCUSSION

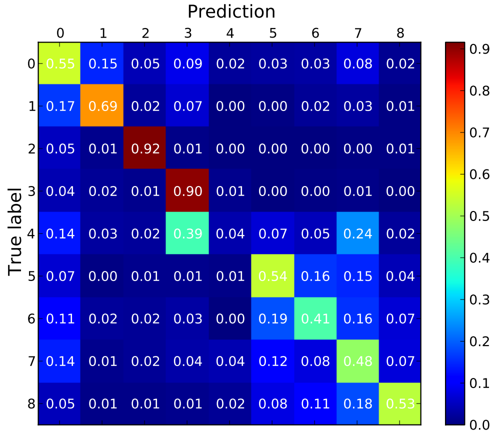
We hereby state that all the work presented in this report is that of the authors.

7. REFERENCES

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[2] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[3] J. Bouvrie. Notes on convolutional neural networks. 2006.



	0	1	2	3	4	5	6	7	8
0	0.55	0.15	0.05	0.09	0.02	0.03	0.03	0.08	0.02
1	0.17	0.69	0.02	0.07	0.00	0.00	0.02	0.03	0.01
2	0.05	0.01	0.92	0.01	0.00	0.00	0.00	0.00	0.01
3	0.04	0.02	0.01	0.90	0.01	0.00	0.00	0.01	0.00
4	0.14	0.03	0.02	0.39	0.04	0.07	0.05	0.24	0.02
5	0.07	0.00	0.01	0.01	0.01	0.54	0.16	0.15	0.04
6	0.11	0.02	0.02	0.03	0.00	0.19	0.41	0.16	0.07
7	0.14	0.01	0.02	0.04	0.04	0.12	0.08	0.48	0.07
8	0.05	0.01	0.01	0.01	0.02	0.08	0.11	0.18	0.53

Figure 10: Confusion matrix for the convolutional neural network

[4] C. Doersch, S. Singh, A. Gupta, J. Sivic, and A. A. Efros. What makes paris look like paris? In *ACM Transactions on Graphics (SIGGRAPH)*, volume 31, page 101. ACM, 2012.

[5] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[6] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[7] J. Knopp, J. Sivic, and T. Pajdla. Avoiding confusing features in place recognition. In *Computer Vision–ECCV 2010*, pages 748–761. Springer Berlin Heidelberg, 2010.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] H. Larochelle. Contenu du cours autoencodeurs. `http://info.usherbrooke.ca/hlarochelle/cours/ift725_A2014/contenu.html`, 2013. Accessed: 2014-11-30.

[10] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[11] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[12] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

APPENDIX

Table 2: Average training and validation errors based on hyperparameter configuration for the stacked denoising autoencoder

Experiment	Pretrain lr	# Pretrain epochs	Finetune lr	# Train epochs	# Hidden layers	Layer noise	L1 factor	L2 factor	Error (train)	Error (valid.)
1	0.001	20	0.1	120	500, 500	0.1, 0.2	0	0	0.59727431	0.68889509
2	0.001	20	0.1	120	500, 500, 500	0.1, 0.2, 0.3	0	0	0.51444444	0.66782924
3	0.001	20	0.1	120	500, 500, 500	0.2, 0.3, 0.4	0	0	0.48682292	0.67327009
4	0.001	20	0.2	120	500, 500, 500	0.1, 0.2, 0.3	0	0	0.33623264	0.67117746
5	0.001	20	0.2	120	500, 500, 500	0.3, 0.4, 0.5	0	0	0.28029514	0.68317522
6	0.001	20	0.2	120	250, 250, 250	0.1, 0.2, 0.3	0	0	0.58552083	0.69963728
7	0.005	20	0.2	120	500, 500, 500	0.1, 0.2, 0.3	0	0	0.44685764	0.66266741
8	0.005	20	0.2	120	500, 500, 500	0.25, 0.25, 0.25	0	0	0.51513889	0.66503906
9	0.005	20	0.2	120	1500, 1500	0.25, 0.25	0	0	0.09411458	0.64606585
10	0.005	20	0.2	120	500, 500	0.25, 0.25	1e-05	0	0.37560764	0.67424665
11	0.005	20	0.2	120	500, 500	0.25, 0.25	1e-05	1e-05	0.48104167	0.67368862
12	0.005	5	0.2	120	500, 500	0.25, 0.25	0	0	0.23010417	0.67606027
13	0.005	20	0.2	120	500, 500	0.35, 0.35	0	0	0.54907986	0.67926897
14	0.005	20	0.2	120	500, 500	0.35, 0.35	0	1e-04	0.56454861	0.67103795
15	0.005	20	0.05	120	500, 500	0.35, 0.35	1e-04	1e-04	0.68671875	0.70438058
16	0.001	5	0.05	120	500, 500	0.35, 0.35	1e-04	1e-04	0.72546875	0.74372210
17	0.005	20	0.2	120	1500, 1500	0.25, 0.25	1e-04	1e-04	0.70560764	0.72154018

Table 3: Average training and validation errors based on hyperparameter configuration for the convolutional neural network

	# Epochs	Learning rate	Filter sizes	# Filters	Downsampling	Hidden layers	Error (train)	Error (valid.)
1	100	0.2	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500)	0.353837	0.472238
2	100	0.2	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.456059	0.511300
3	100	0.1	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.381302	0.456055
4	100	0.05	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.406875	0.453125
5	100	0.025	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.430278	0.476283
6	100	0.05	(9, 6, 5, 5)	(32, 64, 80, 80)	(2, 2, 2, 2)	(500, 500)	0.467639	0.549107
7	100	0.05	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 1, 1, 1)	(500, 500)	0.370069	0.487165
8	100	0.05	(9, 6, 4, 3, 3)	(32, 64, 80, 80, 80)	(4, 2, 1, 1, 1)	(500, 500)	0.380729	0.452148
9	200	0.05	(9, 6, 4, 3, 3)	(32, 64, 80, 80, 80)	(4, 2, 1, 1, 1)	(500, 500)	0.246354	0.433873
10	100	0.05	(9, 6, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.283090	0.428153
11	100	0.05	(9, 8, 5, 5)	(32, 64, 80, 80)	(4, 2, 1, 1)	(500, 500)	0.392865	0.473772