CS EE WORLD
32/34
A
MAY 2019

Investigating The Effect of Customizeable Configurations of Computer Vision Algorithms on Performance

To what extent is the performance of computer vision algorithms in image analysis influenced by kernel count and subsampling stride?

A Computer Science Extended Essay

3972 Words

Table of Contents

1. Introduction

Computer vision (the ability for computers to learn to recognize and/or make meaning from visual data (Kaiser)) is a groundbreaking field with a lot of potential and a growing community behind it. It is being used in a variety of disciplines, including medicine & diagnosis, security, automated cars, video games and CAPTCHA (Peregud; Dodge and Karam 2). There is also constant research into present and future computer vision algorithms; large firms like Google and Facebook are big players in this field of research (Peregud; Basu, Kirit, and Josh Patterson).

For computers utilizing computer vision algorithms to become accurate enough for use in the real world, they must analyze large amounts of images. There are a number of configurable properties in computer vision algorithms that affect the time it takes them to analyze these sets of data and how accurate the resulting algorithm will be in understanding/recognizing similar images afterwards.

This paper seeks to investigate the extent at which two particularly popular configurable properties in modern computer vision algorithms (i.e. kernel count and subsampling stride) affect the performance of computer vision algorithms, specifically their data analysis speeds and their resultant accuracy in image recognition.

This research could prove especially useful in the medical field. Over the past few years, more attention has been given to the medical applications of computer vision, and software are being created to help diagnose diseases like Alzheimer's disease and detect excessive blood loss during surgeries, among other things (Peregud; Awate, Gururaj, et al.). However, during the development of such software, computer vision algorithms must often analyze large amounts of data repeatedly as the developers improve and optimize their configurations iteratively (Chen, Mu-Yen, et al)—this can take days or weeks. (Basu, Kirit, and Josh Patterson). A better understanding of how the configurable properties of these algorithms affect their performance would reduce the duration of this iterative process and help them make faster CNNs, resulting in sooner and better releases of medical computer vision software. This would save thousands of lives and billions of dollars in hospitals around the world (Peregud).

To investigate this relationship, two similar computer vision algorithms were programmed and were made to repeatedly analyze images from public datasets. Their kernel count and subsampling stride were altered after every rerun, and patterns in their performance were analyzed. Logical and mathematical explanations for the results obtained were discussed.

2. Background Information

2.1 Machine Learning and Its Types

Computer vision is based on machine learning, which is the study of getting computers to perform tasks they haven't been directly programmed to do (Ng). It involves providing computers with sample data for them to analyze and learn from, after which they will use their acquired knowledge to perform complex tasks.

There are several types of machine learning algorithms, but computer vision usually makes use of classification, which involves getting computers to learn the mappings between data and their labels or classifications (Salian; Peregud). For example, a computer could be given a collection of songs grouped by genre and the classification algorithm would analyze the properties of these songs and attempt to relate certain audio characteristics to certain genres. This process is called training (Salian). If successful, the computer will eventually be able to identify the genres of songs it has never encountered before because it found the correct relationships between the properties of a song and its genre.

Computer vision algorithms are trained in a similar fashion with thousands of labeled images.

2.2 Convolutional Neural Networks

Today, a classification algorithm called Convolutional Neural Networks (abbreviated CNN) is the main algorithm used in computer vision (Peregud). Due to their popularity, they will be the algorithm discussed in this paper. The inner workings of CNNs are discussed below.

The Tensor Representation of Images

When a CNN is given an image, it understands it as a stack of matrices called a *tensor* ("Convolutional Neural Network (CNN)"). Each matrix in this stack contains the pixel values that the image has in one of the three digital primary colors — red, green or blue ("Convolutional Neural Network (CNN)").

This means if a CNN is given an image of resolution 3x3 composed of only two of the primary colors—these colors are called *channels* ("Convolutional Neural Network (CNN)")—then the tensor would have two 3x3 layers, one for each channel.

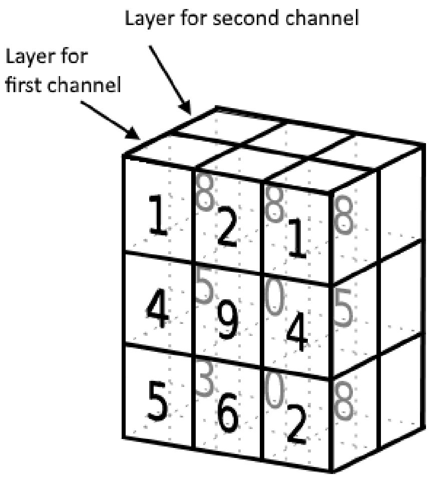


Figure 1: A graphical representation of what the aforementioned example tensor would look like. Adapted from (Santos)

Convolution

A CNN *could* try and analyze the image's tensor straight away, but this could prove both unnecessarily expensive and slow (Dodge and Karam 2). The number of pixels composing an image of resolution $n \times n$ scales with a Big-O of $O(n^2)$, so even moderately sized images could produce massive input tensors and significantly lengthen training times. Additionally, CNNs provided with excessive input data can undergo a process called overfitting, which reduces their predictive capabilities (Karpathy; Strand). Overfitting is when a neural network learns rules that are too specific to its training data, making it very inaccurate and/or inconsistent when interpreting input that it hasn't explicitly trained on before (Koehrsen).

To mitigate this, the CNN looks for certain patterns or visual structures in the image, and analyzes those patterns instead — it gathers these patterns by passing "filters" called *kernels* over the image's tensor, where each filter is designed to recognize a particular pattern("Convolutional Neural Network (CNN")

The image below illustrates how kernels work across the channels of an image:

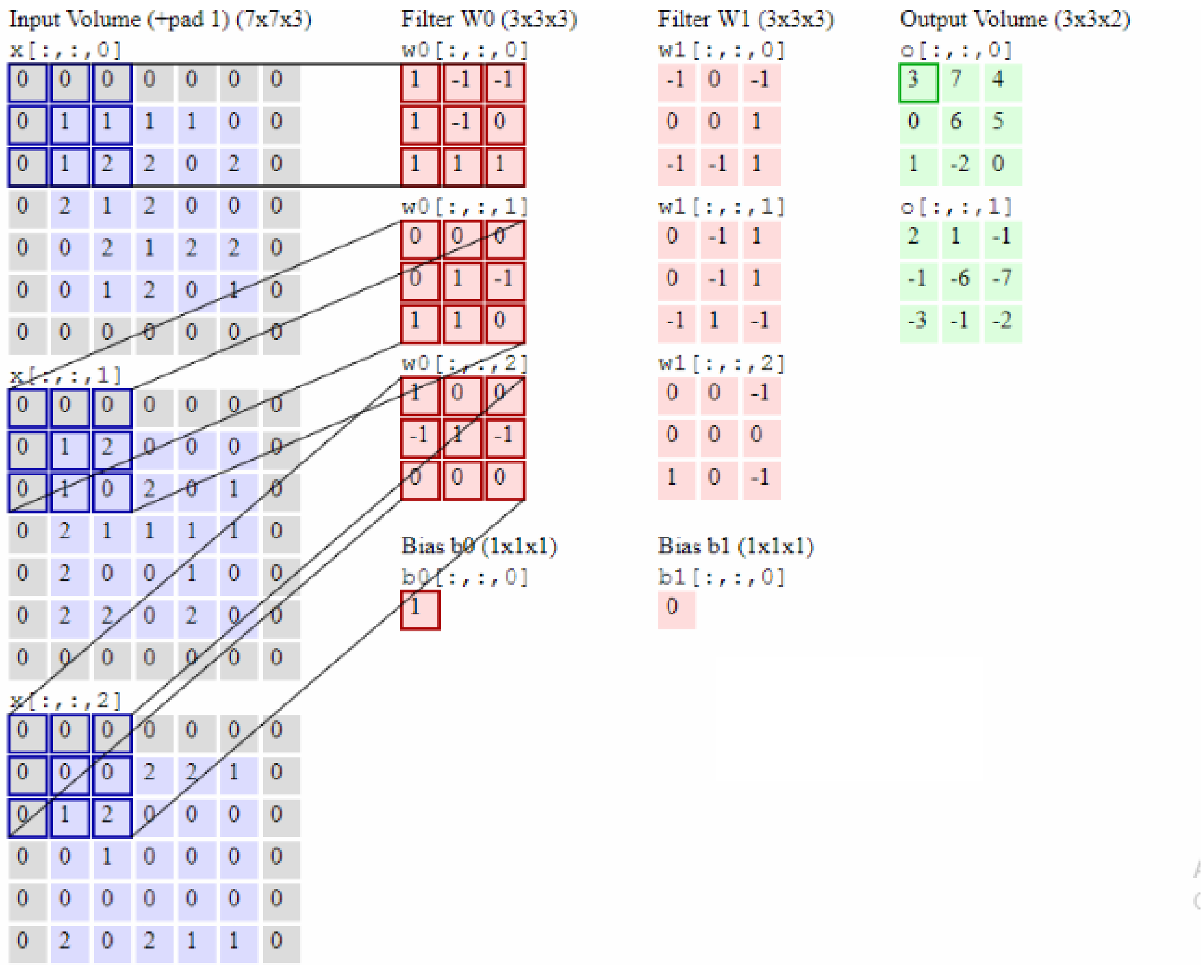


Figure 2 Visualization of Convolution (Karpathy)

The three different input volumes on the extreme left represent the matrices for the three channels of the input image. Each column of red matrices represents a full kernel ("Convolutional Neural Network (CNN)"). Each kernel has different matrices for each channel — this makes them color-sensitive, and makes the CNN more powerful. These kernels in particular have a *stride* of 2x2, meaning they will glide over the input channels by moving to the right by 2 pixels each time, and when there is not enough space on the right, they will move downwards by 2 pixels and then starting again from the extreme left ("Convolutional Neural Network (CNN)"). Its movement is similar to that of a cursor in a word processor.

Every time the kernel rests its matrices on the channels, it multiplies the values in its matrices by the values in the parts of the channels being analyzed, sums the products, adds an arbitrary number called a *bias* (indicated at the bottom under their respective kernels), and stores the resulting value in another matrix called an *activation map* ("Convolutional Neural Network (CNN)"). The activation maps are indicated in green.

Each kernel produces a full activation map; these maps are indications of how dense a kernel's pattern is in different parts of the input image ("Convolutional Neural Network (CNN)").

The part of a CNN that performs convolution is called a *convolution layer (*Chen, Gururaj, et al), and they initialize their kernels randomly using a number called the *seed* (Gibson).

This pattern recognition process (called convolution) increases the CNNs robustness because it is not merely looking at a collection of raw values during image analysis, but rather looking at the relative strength and location of patterns ("Convolutional Neural Network (CNN)")..

Reducing the number of kernels in a CNN can act as a form of optimization; it would reduce the number of activation maps that the CNN would have to process per image, and therefore reduce training time.

Subsampling

Activation maps from a convolution layer become the input for the rest of the CNN. CNNs often further analyze activation maps with secondary or even tertiary convolution layers.

This can result in a lot of data being passed down the network, some of which is redundant. So, CNNs practice something called *subsampling* ("Convolutional Neural Network (CNN)") to condense the activation maps before entering them into another layer.

The image below illustrates what a subsampling layer in a CNN does:

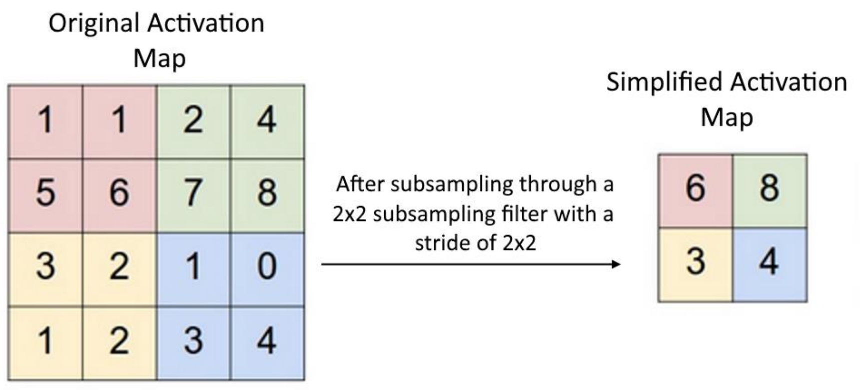

Original Activation Map
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
After subsampling through a 2x2 subsampling filter with a stride of 2x2
Simplified Activation Map
6 8
3 4

Figure 3 Visualization of Subsampling. Adapted from (Karpathy)

The matrix on the left represents an activation map that was inputted into a subsampling layer, and the matrix on the right represents its condensed counterpart.

Subsampling layers also use filters, but instead of looking for patterns, subsampling filters aim at condensing activation maps by isolating the maps’ largest values.

Increasing the subsampling stride of a subsampling layer is a method of CNN optimization because it reduces the size of the activation maps to be processed, which simplifies and speeds up the image analysis.

Neural Network Image Analysis

After convolution and subsampling have been performed, the resultant activation maps are fed into the CNN's neural network for the actual classification.

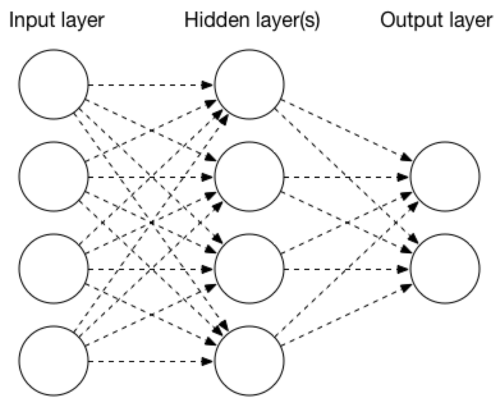


Figure 4 A neural network (Kerimbaev)

Neural networks are networks of digital units called neurons. The only purpose of a neuron is to receive a value, perform a calculation on it, and transfer the result to the next neuron (Sanderson). Input is represented as of a collection of values in the input layer neurons, and these values are calculated upon and passed down to each consecutive neuron layer until they reach the output layer, which represents the network's output (Sanderson).

Each link between two neurons is assigned a multiplier called a *weight* (Sanderson). These weights will multiply the value (or *activation* (Sanderson)) of the previous neuron, then pass it on to the next neuron. Consider two consecutive layers of *y* number of neurons, layer *j* and layer *k*. The formula below shows what the value for the *nth* neuron in layer *k* will be, based off all its connections to layer *j*'s neurons:

$$Ak_n = \theta\left(\sum_{i=1}^{y}(w_{k_n}^{j_i} \cdot Aj_i) + b\right)$$

Equation 1: Mathematical expression for the activation of a neuron given two layers, j and k

Ak_n represents the activation of the neuron in question, $\sum_{i=1}^{y}(w_{k_n}^{j_i} \cdot Aj_i)$ is the sum of the products of the activations of layer j's neurons and their weights (respective to neuron k_n), and *b* refers to a random value that every neuron has called a *bias* (Sanderson). The weights and biases are initialized randomly using a seed during the network's initialization. Lastly, θ refers to a function called an *activation function*, which, among other things, limits the activation of any neuron to a restricted range (usually 0 to 1, for performance and uniformity reasons) (Sanderson; Bupe).

In reality, the whole neural network is representing one huge function where a large amount of inputs and parameters produce a certain number of outputs (Sanderson).

Output is produced once the flow in information reaches the output layer. Each output layer neuron outputs a probability that the answer that neuron represents is the correct one (Sanderson). The network returns the answer with the highest probability, and compares it with the right answer to calculate a value called *cost* based off how close it was to the real answer (higher cost is worse). Since neural networks represent functions, their gradient relative to this cost can be calculated. Adjustments are made to all the weights and biases to move *down* this gradient and minimize cost (Sanderson).

Over time, as cost is minimized, the network becomes better at predicting the right answers.

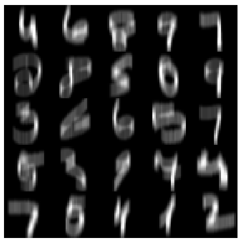
3. Experiment Methodology

Primary experimental data is the main source of data in this paper. Two CNNs were programmed (code in appendix, heavily adapted from (Black, Alex, and Fvaleri)) and fed images from public datasets, and their training duration and resultant accuracy were recorded with different kernel counts and subsampling strides. An experimental methodology was chosen because there was limited secondary data to answer this paper's research question, and this methodology provides ample freedom to manipulate the independent variables. However, this methodology is subject to technical limitations; the scope of the experiment was limited because only monochromatic images could be analyzed with the hardware available. Time constraints also prevented the utilization of more CNNs with varying structures.

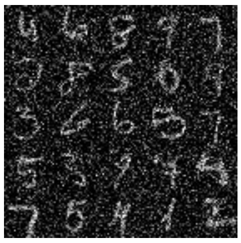
3.1 The Datasets Used

MNIST is a public dataset of thousands of 28x28 labelled images of handwritten digits ranging from 0-9 by LeCun, Yann, et al. For this experiment, 4 variations of the MNIST dataset were used. Since there are 10 different data classifications in MNIST, it's said to have 10 *classes* ("Glossary"). Different variations were used to determine whether any patterns observed would be specific to particular types of images. The images on the next page are samples of the variations of MNIST used:

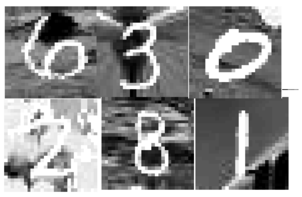
MNIST with random noise (i.e. random variations added to the value of reach pixel) added (Basu, et al.). Abbreviated nMNIST in this paper



MNIST with uniform motion blur added (Basu, et al.). Abbreviated as mMNIST.



MNIST with random noise and reduced contrast (Basu, et al.). Abbreviated as cnMNIST.



MNIST with random backgrounds added ("MnistVariations"). Abbreviated as bMNIST.

Figure 5: The different MNIST variations used. The nMNSIT, cnMNIST and mMNIST datasets are from a paper by Nemani, et al.

These datasets contain 60,000 images to train a CNN, and 10,000 to evaluate its accuracy after training. bMNIST is the only exception with 50,000 training images and 12,000 evaluation images. All these datasets have 1 channel.

3.2 Processing the datasets for use

These datasets were only available as .mat files, which are incompatible with the Java platform used to program the experimental CNNs, according to one of the creators of the platform, Gibson. The name of the Java platform used is DeepLearning4J, abbreviated as dl4j.

The datasets were converted from the .mat format to the more compatible .csv format by rearranging their data using the MATLAB command line, combining the labels with the image data, and using the `csvwrite` MATLAB command.

3.3 The Dependent Variables

The variables being measured in this paper are accuracy and training time.

Time

The time measured was the time it took the CNNs to fully train themselves on a dataset's training data; evaluation times with evaluation data were not considered. Java's `System.nanoTime()` method was used to obtain this time.

Accuracy

The accuracy recorded was the accuracy of the CNNs during post-training evaluations with evaluation datasets. The accuracy for any class in an evaluation dataset is the number of correct predictions (for that class) divided the number of total predictions. A prediction is correct relative to a class if the CNN could successfully tell something was or wasn't an instance of the class ("Classification: Accuracy."). The accuracy recorded was the average of the accuracies for all the 10 classes.

3.4 The Convolutional Neural Networks Programmed

The two CNNs programmed were identical in structure; the only difference was their seed for kernel, bias and weight initialization. One used a seed of `1234` (nicknamed CNN1234), the other `3732` (nicknamed CNN3732). Both seeds were arbitrarily chosen, and two seeds were used to ensure that the patterns observed were not seed-specific. The diagram below illustrates the structure of the CNNs programmed:

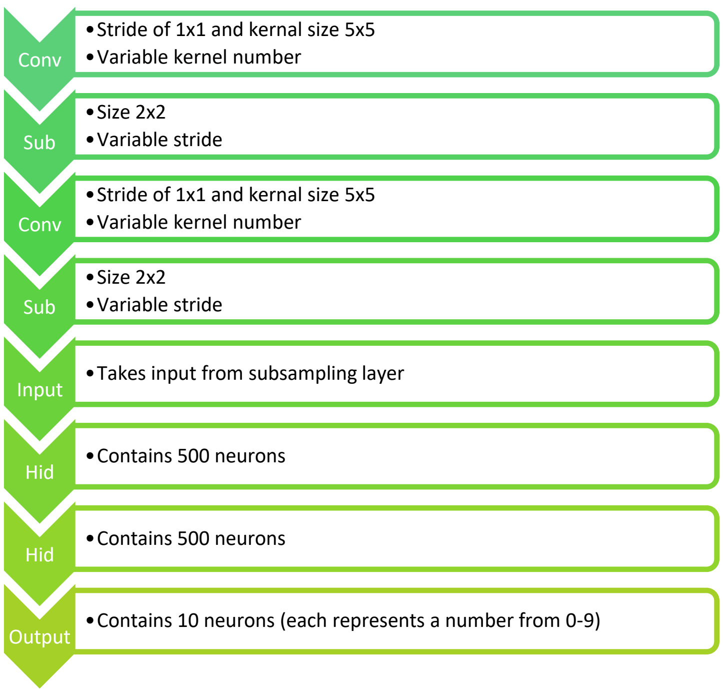


Figure 6 The structure of the CNNs used. Conv stands for a Convoluation layer, Sub for a Subsampling Layer, Input for an Input layer, Hid for a hidden layer and Output for an output layer. Drawn by Author

Some insight needed to draw figure 6 was gotten from Nicholas et al.'s articles "The Deeplearing4j Quick Reference" and the "Creating Deep-learning Networks" guide.

3.5 The Experimental Procedure

Each CNN was initially configured with 20 kernels in their first convolutional layer, 50 kernels in their second convolutional layer and a stride of 1x1 for both subsampling layers. Their training time and accuracy were recorded for each of the MNIST datasets. Their subsampling strides were then increased to 2x2, then 3x3, then4x4, and with each increase, their performance for each MINST dataset was recorded again.

This procedure was then repeated with different kernel counts for each CNN:

1. 75% of original kernel count (15 kernels for first convolutional layer, 38 for the second)
2. 50% of original kernel count (10 kernels for first layer, 25 for second)
3. 0.03% of original kernel count (1 kernel for each layer)

4. The Experimental Results

4.1 Tabular Data Presentation

Table of CNN1234 Results

The table below shows the experimental results of CNN1234. A large number of decimal places were used to maintain high accuracy in the recorded results, as some of the differences between some of the values are quite small.

		Accuracy (out of 1)				Training Duration in seconds			
		Kernel Concentration				Kernel Concentration			
Dataset	Subsampling Stride	100%	75%	50%	0.03%	100%	75%	50%	0.03%
nMINST	1x1	0.9572	0.9498	0.9494	0.9188	1073	857	546	91
	2x2	0.9514	0.946	0.9453	0.703	216	188	128	71
	3x3	0.8384	0.8042	0.7443	0.2699	114	98	86	65
	4x4	0.8683	0.7389	0.8791	0.303	97	85	79	67
cnMNIST	1x1	0.9393	0.9012	0.9279	0.8805	1068	815	536	89
	2x2	0.9211	0.9193	0.921	0.8215	218	167	130	68
	3x3	0.7387	0.7744	0.7745	0.2658	114	94	82	66
	4x4	0.818	0.777	0.8237	0.2592	94	81	77	64
mMNIST	1x1	0.9602	0.9591	0.9601	0.9352	1048	818	526	90
	2x2	0.9573	0.9561	0.9549	0.9114	216	167	126	68
	3x3	0.7928	0.7958	0.7508	0.3542	111	98	83	70
	4x4	0.8672	0.816	0.8002	0.3109	95	88	75	64
bMNIST	1x1	0.9194	0.92	0.9197	0.832	861	677	447	76
	2x2	0.9061	0.8992	0.8962	0.7131	177	138	110	59
	3x3	0.8335	0.8252	0.8103	0.227	93	81	73	57
	4x4	0.8361	0.824	0.8094	0.2203	82	76	70	54

Table 1: CNN1234 Results

Table of CNN3732 Results

The table below shows the experimental results of CNN3732.

		Accuracy (out of 1)				Training Duration in seconds			
		Kernel Concentration				Kernel Concentration			
Dataset	Subsampling Stride	100%	75%	50%	0.03%	100%	75%	50%	0.03%
nMINST	1x1	0.9426	0.9391	0.9514	0.8845	1017	788	517	91
	2x2	0.9325	0.9408	0.9451	0.8101	204	164	122	68
	3x3	0.845	0.8481	0.8176	0.2543	111	106	90	66
	4x4	0.7895	0.8545	0.7973	0.2883	89	81	75	65
cnMNIST	1x1	0.9193	0.9258	0.9006	0.884	991	765	515	88
	2x2	0.935	0.9115	0.8969	0.8294	168	161	122	66
	3x3	0.782	0.7436	0.7535	0.2795	107	98	82	64
	4x4	0.847	0.8025	0.7512	0.2779	90	80	74	63
mMNIST	1x1	0.9582	0.9569	0.9519	0.9314	993	772	511	87
	2x2	0.958	0.9713	0.948	0.8807	199	160	120	66
	3x3	0.7605	0.7018	0.7246	0.3393	106	95	75	64
	4x4	0.8837	0.8046	0.9024	0.3392	86	72	71	56
bMNIST	1x1	0.9182	0.9165	0.9129	0.8336	786	597	405	72
	2x2	0.9018	0.9019	0.8916	0.7083	156	125	97	56
	3x3	0.8323	0.8206	0.8082	0.2271	87	75	64	60
	4x4	0.8372	0.8229	0.8048	0.2254	69	74	66	56

Table 2: CNN3732 Results

4.2 Graphical Presentation of Data

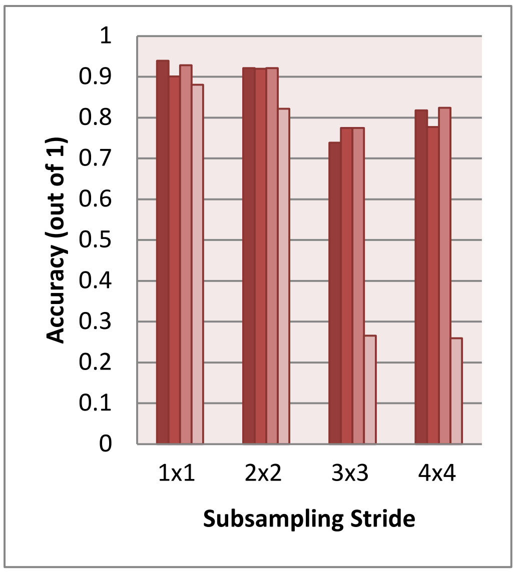
To better understand the trends in performance, the data has been represented as clustered bar charts below.

The first row of bar charts displays accuracy (no unit; maximum achievable accuracy is 1), and the second displays training duration (in seconds). The bars are arranged in clusters – each cluster represents the performance at a particular subsampling stride (indicated on x-axis). The first bar in each cluster represents 100% kernel concentration with its respective subsampling stride, the second 75%, the third 50% and the fourth 0.03%.

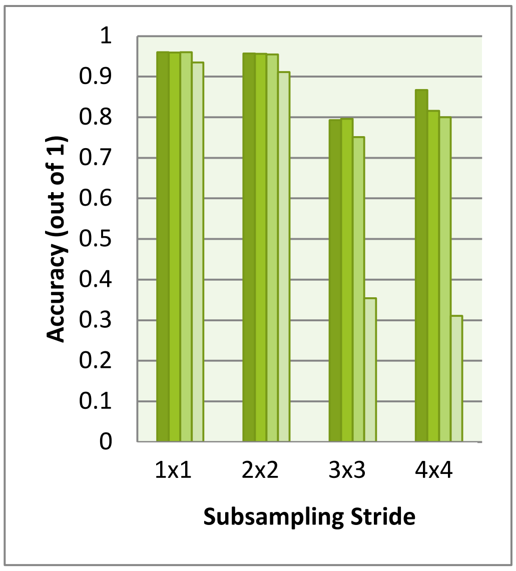
Graphical Presentation of CNN3732 cnMINST and mMNIST results

cnMNIST Results

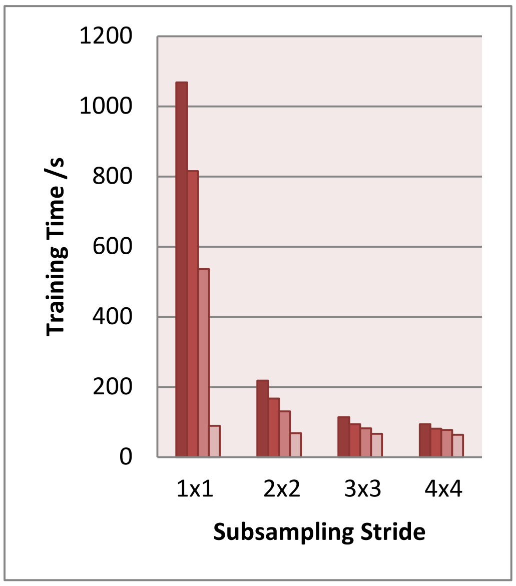
mMNIST Results



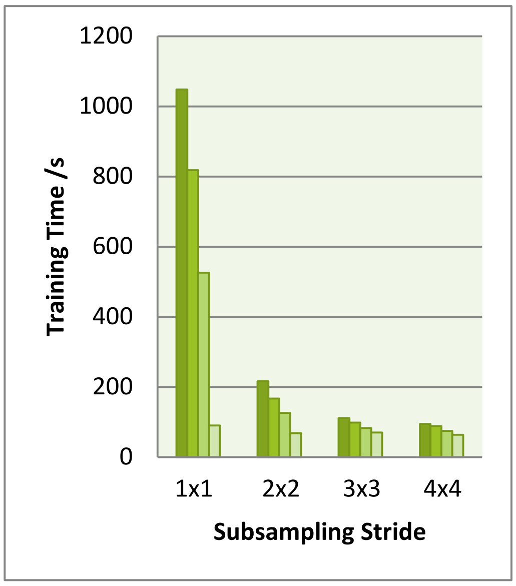
Graph 1: CNN3732 cnMNIST accuracy results



Graph 3: CNN3732 mMNIST accuracy results



Graph 1: CNN3732 cnMNIST training time results

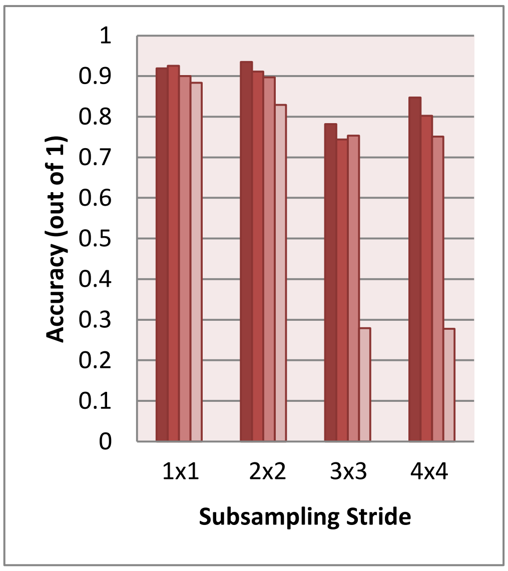


Graph 4: CNN3732 mMNIST training time results

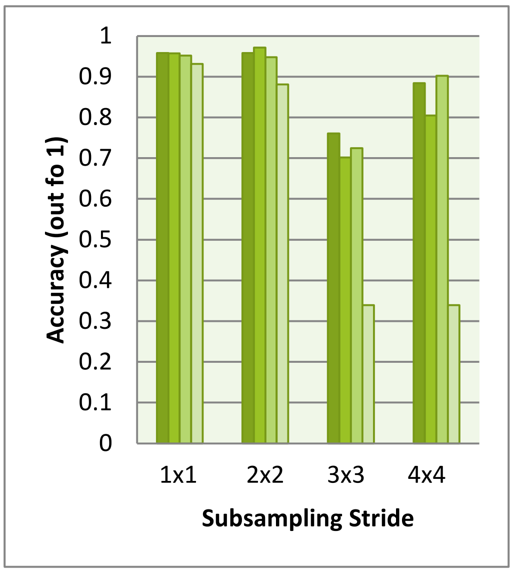
Graphical Presentation of CNN1234 cnMINST and mMNIST results

cnMNIST Results

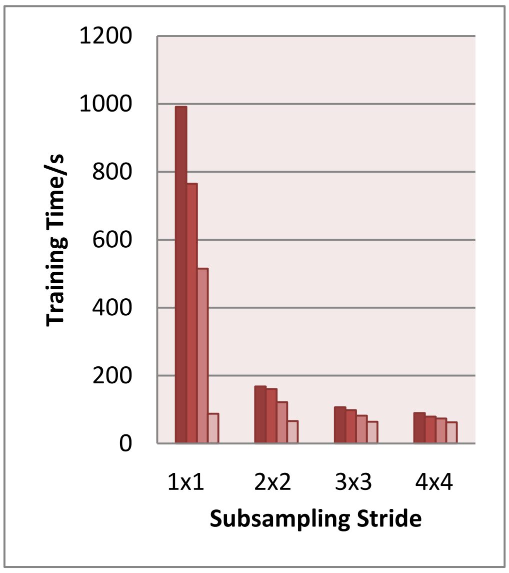
mMNIST Results



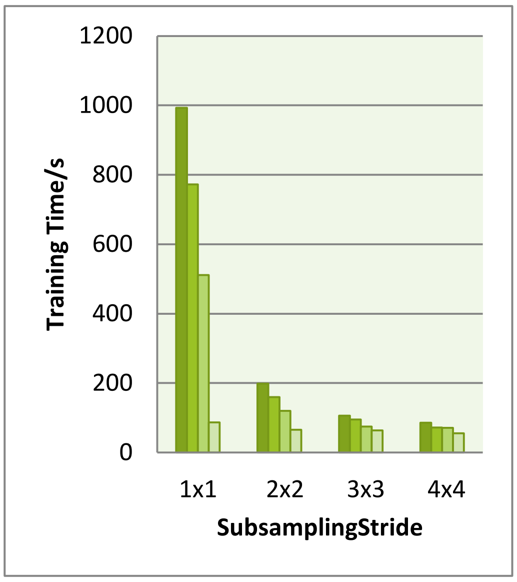
Graph 5: CNN1234 cnMNIST accuracy results



Graph 7: CNN1234 mMNIST accuracy results



Graph 6: CNN1234 cnMNIST training time results



Graph 8: CNN1234 mMNIST training time results

Only the results for mMNIST and cnMNIST have been graphically presented because these **results** are very representative of the results for all the datasets for both the CNNs.

4.3 Data Analysis

The patterns observed were consistent for all datasets across both CNNs, indicating that they are image and seed independent, and rather integral to CNNs.

Analysing Stride

These results indicate that increases in subsampling strides have a much larger effect on training speed than reductions in kernel amount. It can also be seen that both CNNs experienced diminishing speed improvements as the subsampling stride kept on being increased.

The results show a trade-off between speed and accuracy when the subsampling stride is increased. This only becomes significant with strides 3x3 and 4x4, meaning that it is possible for changes in subsampling stride to only affect speed, provided the stride is not already too large. Additionally, it can be seen that the more kernels a CNN has, the less its accuracy will reduce as subsampling stride increases.

Analysing Kernel Count

Though the effect is generally less prominent than that of subsampling stride, the results show that decreasing the kernel count does indeed increase training speed; this effect is more prominent when the subsampling stride was low.

The effect of kernel amount on accuracy is more interesting to analyse though, because there is no clear pattern. One can see there is somewhat of a general downward trend in accuracy as the kernel amount decreases, especially when one examines the 0.03% results. However, there are instances where a decrease in kernel amount lead to almost no change in accuracy, or even an *increase* in accuracy. This is rather counterintuitive.

Making sense of the Accuracy and Speed Results

Making sense of the counterintuitive relationship between kernel amount and accuracy requires one to remember that neural networks represent very large functions that improve their predictive power over time. The larger the number of kernels, the larger the number of parameters in the function that influence its input, meaning CNN3732 with 70 kernels isn’t the same function as CNN3732 with 35. Although the function with 70 kernels is more complex and has more parameters, it doesn't guarantee that it *will* be better; it only increases its potential to be better.

It is important to note (and the results show this) that neural networks do not necessarily *need* a lot of kernels to be effective because kernels only help infer data. The purpose of training is to get the network to learn and construct its own rules from what it sees. So if the number of kernels a network has access too is reduced, as long as the remaining kernels still provide *complete* data, it is possible for the network to stay accurate, because whatever information one kernel infers can still be inferred from a combination of other kernels if the network processes the their input properly enough.

This logic also explains why increasing the subsampling stride decreases accuracy *and* makes a network suffer larger accuracy drops upon reductions in kernel amount. Increasing the subsampling stride affects the *completeness* of the information being given to the network. If the information is less complete and hence less comprehensive (i.e. the activation maps are smaller), then the number of kernels becomes a more significant factor of accuracy because each individual kernel's output is less comprehensive and useful. It will take more kernels (hence more activation maps) to compensate for this absolute lack of information.

Explaining the diminishing speed benefits of increasing subsampling stride is a mathematical endeavour.

The activation maps outputted by the first convolutional layer in the experimental CNNs are of size 24x24. When these activation maps are passed into a subsampling layer, their dimensions are essentially being divided by a value to reduce their size. The length of their resultant activation maps can therefore be approximated mathematically as $y = \frac{24}{x}$ (the larger the subsampling stride is, the larger x is). If one were to look at a graph of $y = \frac{24}{x}$, they would notice with every increase in x, the decrease in y becomes less significant.

However, $\frac{24}{x}$ only expresses the length of the resultant map after it passes through a single subsampling layer. With the experimental CNNs, it will pass through another convolutional and subsampling layer afterwards. If we were to look at another approximated expression for the length of their resultant activation maps, but this time after going through all the CNN's layers, it would look like this:

$$\frac{24}{x} \div 5 \div x$$

Equation 2: A more accurate estimation for resultant map length given a dimension of the subsampling stride. The number 5 was used because my CNNs had kernels with dimensions 5x5

This equation's gradient also approaches zero over time, but at a faster rate than $\frac{24}{x}$. This goes to say that the multi-layered nature of the CNNs used intensifies the effect of the diminishing returns of increasing subsampling strides.

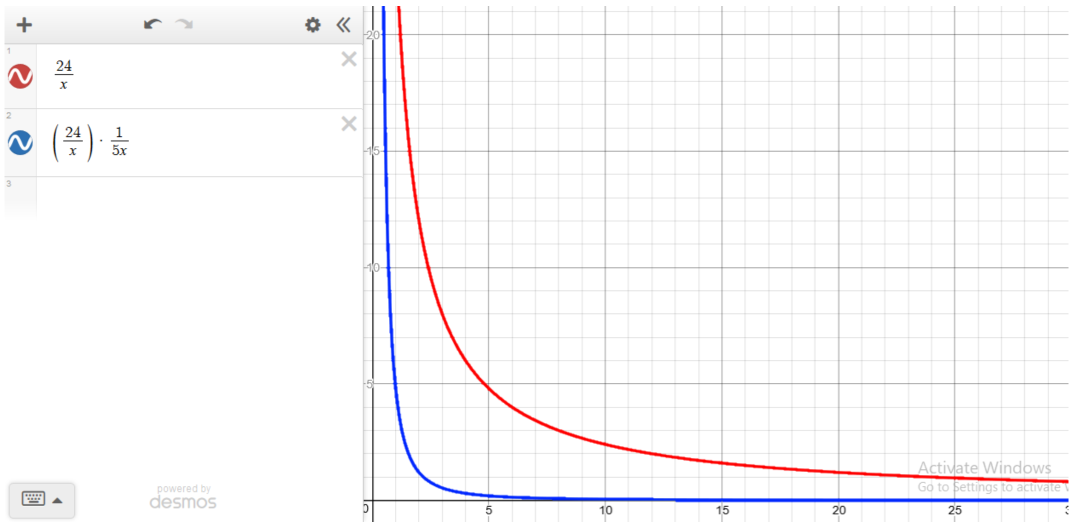


Figure 7: Graphs for 24/x (in red) and 24÷x÷5÷x (in blue). It can be seen that the blue line approaches 0 faster than the red line. (Desmos Graph)

Notice that these expressions are for the *length* of the resultant activation map. Once we square them to represent the number of input neurons required to input this activation map into a neural network, the diminishing effect intensifies further.

Each input neuron comes at a time cost, so these mathematical expressions effectively model the relationship between subsampling stride and training duration.

5. Further Research Opportunities

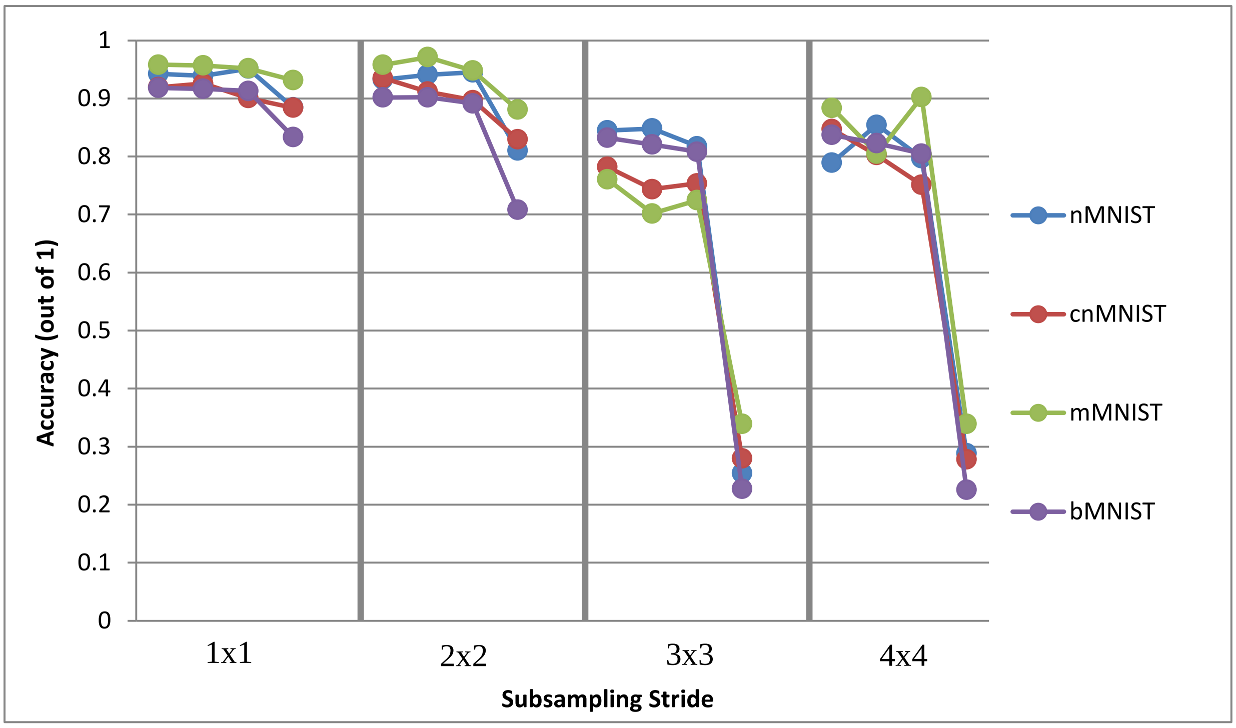
5.1 Investigating Datasets with Multiple Channels

It would be interesting to investigate how other CNNs perform in likewise experiments with datasets containing 3 channels, like the CIFAR10 dataset. Kernels could have a higher effect on accuracy in such scenarios, due to the higher input complexity and the presence of color.

5.2 Comparing the datasets themselves

During analysis, it was noticed that the CNNs had a certain level of preference for specific datasets. Though this isn't directly related to the research question, its intriguing nature and research implications merit it some attention.

Below is a graph showing all the accuracies of CNN1234 for all the 4 datasets. Once again, clusters of points represent subsampling strides, and each individual point within a cluster represents 100%, 75%, 50% and 0.03% kernel concentrations respectively.



Graph 9: CNN1234 accuracy results across all MINST variant datasets

It can be noticed that for strides 1x1 and 2x2, CNN1234 performed best with mMNIST consistently, followed by nMNIST. bMNIST was understandably consistently worst since it had 17% less training data than the other datasets.

Once the size of the subsampling stride started to hamper accuracy (strides 3x3 & 4x4), this pattern began to dissolve into randomness as CNN1234 struggled to make sense of its limited information. A very similar pattern is seen with CNN3732 as well.

It is surprising that the significant motion blur of mMNIST outperformed the noise of nMNIST for both CNNs. A paper by Zhou, et al. suggests that it should rather be the opposite (for MNIST-based datasets), and demonstrates this with a CNN of similar structure to the ones used here. More investigation into the relationship between the type of image distortion and the resultant CNN accuracy could prove promising.

6. Conclusion

In this paper, the effects of changing subsampling stride and the number of convolution kernels on CNNs' performance and speed were analyzed. Logical and mathematical explanations for the patterns observed were also provided.

The results show that these increasing subsuming stride and reducing kernel count generally have negative effects on accuracy, albeit they improve speed. The effects of these optimizations follow general trends, but the random nature of CNNs dilutes the consistency of said trends.

Subsampling stride has a relatively larger effect on speed and accuracy than kernel count. However, the speed benefits of increasing subsampling stride decrease as the stride increases—there are reducing marginal returns. At the same time, increasing subsampling stride will eventually come with significant accuracy trade-offs at some point, making them more disadvantageous than beneficial. The point at which this becomes a problem depends on how many kernels the CNNs has, the more the number of kernels, the higher the subsampling stride must be to experience significant accuracy trade-offs.

The effect of reducing kernel count on CNNs is less predictable. Though there is a general trend that more kernels leads to slower training speeds and higher accuracy, this trend has an element of randomness, so an increase in kernel count can sometimes decrease accuracy, if anything at all. More kernels isn't always better.

Hopefully this paper will prove useful to computer vision designers in guiding their choices as they iteratively improve on their CNNs, leading to more computer vision innovation in the field of medicine and beyond.

7. Works Cited

Most of these references were only used in the detailed background information section.

Awate, Gururaj, et al. "Detection of Alzheimers Disease from MRI using Convolutional Neural Network with Tensorflow." IEEE Xplore 2018, 26 June 2018, arXiv.org. arxiv.org/abs/1806.10170. Accessed 7 Nov. 2018.

Basu, Kirit, and Josh Patterson. "Solving Real-world Business Problems with Computer Vision." O'Reilly Media, 7 Sept. 2017, www.oreilly.com/ideas/solving-real-world-business-problems-with-computer-vision. Accessed 7 Nov. 2018.

Basu, Saikat, et al. "The N-MNIST Handwritten Digit Dataset." LSU Division of Computer Science and Engineering, csc.lsu.edu/~saikat/n-mnist/. Accessed 13 July 2018.

Black, Alex, and Fvaleri. "Deeplearning4j/dl4j-examples." GitHub, 27 Mar. 2018, github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/convolution/mnist/MnistClassifier.java. Accessed 1 June 2018.

Bupe, Chomba. "Why Do Neural Networks Need an Activation Function?" Quora, 14 Aug. 2016, www.quora.com/Why-do-neural-networks-need-an-activation-function. Accessed 26 July 2018.

Chen, Mu-Yen, et al. "Design of experiments on neural network's parameters optimization for time series forecasting in stock markets." Neural Network World, vol. 23, no. 4, 2013, pp. 369-393, ResearchGate. doi:10.14311/NNW.2013.23.023. Accessed 25 Oct. 2018.

"Classification: Accuracy." Google Developers, Google, 1 Oct. 2018, developers.google.com/machine-learning/crash-course/classification/accuracy. Accessed 11 Oct. 2018.

"Convolutional Neural Network (CNN)." Skymind, Skymind, 25 July 2018, skymind.ai/wiki/convolutional-network. Accessed 25 July 2018.

"Desmos Graph." Desmos Graphing Calculator, www.desmos.com/calculator. Accessed 14 Oct. 2018.

Dodge, Samuel, and Lina Karam. "Understanding how image quality affects deep neural networks." 2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX), no. 2, 21 Apr. 2016, arXiv. arxiv.org/abs/1604.04004v2. Accessed 25 July 2018.

Gibson, Adam. Internet Interview (Gitter.im). 10 July 2018.

---. "Creating Deep-learning Networks." Deeplearning4j, Skymind, deeplearning4j.org/multinetwork. Accessed 26 July 2018.
Link now broken due to the dl4j website being completely redone in Early August. Consider looking here: https://web.archive.org/web/20180807102456/https://deeplearning4j.org/multinetwork

Gibson, Adan. "Deeplearing4j with SVHN Dataset." Stack Overflow, Stack Exchange, 27 Dec. 2016, stackoverflow.com/a/41340781. Accessed 26 July 2018.

"Glossary." Skymind, skymind.ai/wiki/glossary. Accessed 21 Aug. 2018.

Kaiser, Adrien. "What is Computer Vision? | Hayo." Hayo | Virtual Controls for Real Life, 12 Jan. 2017, hayo.io/computer-vision/. Accessed 9 Nov. 2018.

Karpathy, Andrej. "Convolutional Neural Networks (CNNs / ConvNets)." CS231n Convolutional Neural Networks for Visual Recognition, cs231n.github.io/convolutional-networks/. Accessed 25 July 2018.

Kerimbaev, Bolot. "Neural Networks in IOS 10 and MacOS." Big Nerd Ranch, 29 July 2016, www.bignerdranch.com/blog/neural-networks-in-ios-10-and-macos/. Accessed 25 July 2018.

Koehrsen, William. "Overfitting Vs. Underfitting: A Conceptual Explanation." Towards Data Science, 28 Jan. 2018, towardsdatascience.com/overfitting-vs-underfitting-a-conceptual-explanation-d94ee20ca7f9. Accessed 21 Aug. 2018.

LeCun, Yann, et al. "MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges." Yann LeCun's Home Page, yann.lecun.com/exdb/mnist/. Accessed 25 Oct. 2018.

"MnistVariations." LISA Public web, 21 June 2007, www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/MnistVariations. Accessed 14 July 2018.

Nemani, Ramakrishna, et al. "Learning Sparse Feature Representations using Probabilistic Quadtrees and Deep Belief Nets." European Symposium on Artificial Neural Networks, 2015, arXiv. arxiv.org/abs/1509.03413. Accessed 26 July 2018.

---. "Deeplearning4j Quick Reference." Deeplearning4j, Skymind, deeplearning4j.org/quickref#layers-out. Accessed 26 July 2018.

Ng, Andrew. "Machine Learning." Coursera, Stanford University, www.coursera.org/learn/machine-learning. Accessed 8 Nov. 2018.

Peregud, Irina. "The Most Exciting Applications of Computer Vision Across Industries." InData Labs, 7 May 2018, indatalabs.com/blog/data-science/applications-computer-vision-across-industries. Accessed 25 July 2018.

Salian, Isha. "Supervised Vs. Unsupervised Learning." The Official NVIDIA Blog, 20 Sept. 2018, blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. Accessed 8 Nov. 2018.

Sanderson, Grant. "Neural Networks Playlist." YouTube, 1 Aug. 2017, www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 17 Feb. 2018.

Santos, Leonardo. "Linear Algebra · Artificial Inteligence." Leonardo Araujo Dos Santos, leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/linear_algebra.html. Accessed 25 July 2018.

Strand, Håkon H. "Can an Excessive Amount of Training Data Cause Overfitting in Neural Networks?" Quora, 27 Feb. 2018, www.quora.com/Can-an-excessive-amount-of-training-data-cause-overfitting-in-neural-networks. Accessed 21 Aug. 2018.

Zhou, Yiren, et al. "On classification of distorted images with deep convolutional neural networks." 2017 IEEE International Conference on Acoustics, Speech and Signal

Processing (ICASSP), no. 1, 8 Jan. 2017, arXiv. arxiv.org/abs/1701.01924. Accessed 28 July 2018.

8. Appendix

8.1 Code for CNN generation

The following code was used for my experiments. I used it to create my CNNs, feed them MNIST data in CSV format, and train & evaluate them using specific configurations, including subsampling stride and kernel size.

It was run using Deeplearning4j in IntelliJ IDEA Community Edition 2018 on an HP 15bs0xx laptop with an Intel i7-7500U processor.

The following VM Options, Program Arguments and Environment configurations respectively were used to increase performance:

- `-Xms1G -Xmx3584m`
- `-Dorg.bytedeco.javacpp.maxbytes=4G -Dorg.bytedeco.javacpp.maxphysicalbytes=4G`
- `OMP_NUM_THREADS=4`

```java
package org.deeplearning4j.examples.convolution;

import java.io.File;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

import org.datavec.api.io.labels.ParentPathLabelGenerator;
import org.datavec.api.records.reader.RecordReader;
import org.datavec.api.records.reader.impl.csv.CSVRecordReader;
import org.datavec.api.split.FileSplit;
import org.datavec.image.loader.NativeImageLoader;
import org.datavec.image.recordreader.ImageRecordReader;
import org.deeplearning4j.api.storage.StatsStorage;
import org.deeplearning4j.datasets.datavec.RecordReaderDataSetIterator;
import org.deeplearning4j.eval.Evaluation;
import org.deeplearning4j.evaluation.EvaluationTools;
import org.deeplearning4j.examples.utilities.DataUtilities;
import org.deeplearning4j.nn.conf.MultiLayerConfiguration;
import org.deeplearning4j.nn.conf.NeuralNetConfiguration;
import org.deeplearning4j.nn.conf.inputs.InputType;
import org.deeplearning4j.nn.conf.layers.*;
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork;
import org.deeplearning4j.nn.weights.WeightInit;
import org.deeplearning4j.optimize.listeners.ScoreIterationListener;
import org.deeplearning4j.ui.api.UIServer;
import org.deeplearning4j.ui.stats.StatsListener;
import org.deeplearning4j.ui.storage.InMemoryStatsStorage;
import org.deeplearning4j.util.ModelSerializer;
import org.nd4j.linalg.activations.Activation;
import org.nd4j.linalg.dataset.api.iterator.DataSetIterator;
import org.nd4j.linalg.dataset.api.preprocessor.DataNormalization;
import org.nd4j.linalg.dataset.api.preprocessor.ImagePreProcessingScaler;
import org.nd4j.linalg.learning.config.Nesterovs;
import org.nd4j.linalg.lossfunctions.LossFunctions;
import org.nd4j.linalg.schedule.MapSchedule;
import org.nd4j.linalg.schedule.ScheduleType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * This is a modified version of the MnistClassifier that reads modified MNIST
datasets (like n-MNIST)
 * that have been packaged as CSV files
 */
public class CSVMnistClassifier{

    private static final Logger log =
LoggerFactory.getLogger(CSVMnistClassifier.class);
    private static String trainFilePath = "";
    private static String evalFilePath = "";

    public static void main(String[] args) throws Exception {
        int height = 28;
        int width = 28;
        int channels = 1; // single channel for grayscale images
        int classCount = 10; // 10 digits classification
        int batchSize = 54;
        int nEpochs = 1;

        float trainingTime = 0;
        float testingTime = 0;

        int kernelSize = 5;
        int kernelStride = 1;
        int samplingSize = 2;
        int seed = 5732;

        //What I change for my research
        int firstKernelAmount = 20;
        int secondKernelAmount = 50;
        int samplingStride = 1;
        int MNISTType = 1; //0 = noise, 1 = low contrast and noise, 2 = motion
blur, 3 = background images


        setPaths(MNISTType);
        log.info("Data load and vectorization...");

        //Loading the training data
        RecordReader trainingRecordReader = new CSVRecordReader(0,",");
        trainingRecordReader.initialize(new FileSplit(new
File(trainFilePath)));
        //Initializing a dataSetIterator that understands that the 785th value
if on of 10 MNIST number classifications
        DataSetIterator trainIter = new
RecordReaderDataSetIterator(trainingRecordReader, batchSize, 784, classCount);
        // pixel values from 0-255 to 0-1 (min-max scaling)
        DataNormalization scaler = new ImagePreProcessingScaler(0, 1);
        scaler.fit(trainIter);
        if (MNISTType != 3) trainIter.setPreProcessor(scaler);

        // vectorization of evaluation data
        RecordReader evaluationRecordReader = new CSVRecordReader(0,",");
        evaluationRecordReader.initialize(new FileSplit(new
File(evalFilePath)));
        DataSetIterator evalIter = new
RecordReaderDataSetIterator(evaluationRecordReader, batchSize, 784,
classCount);
        if (MNISTType != 3) evalIter.setPreProcessor(scaler); // same
normalization for better results

        log.info("Network configuration and training...");
        Map<Integer, Double> lrSchedule = new HashMap<>();
        lrSchedule.put(0, 0.06); // iteration #, learning rate
        lrSchedule.put(200, 0.05);
        lrSchedule.put(600, 0.028);
        lrSchedule.put(800, 0.0060);
        lrSchedule.put(1000, 0.001);

        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(seed)
            .l2(0.0005)
            .weightInit(WeightInit.XAVIER)
            .updater(new Nesterovs(0.01, 0.9))
            .list()
            .layer(0, new ConvolutionLayer.Builder(kernelSize, kernelSize)
                //nIn and nOut specify depth. nIn here is the nChannels and
nOut is the number of filters to be applied
                .nIn(channels)
                .stride(kernelStride, kernelStride)
                .nOut(firstKernelAmount)
                .activation(Activation.IDENTITY)
                .build())
            .layer(1, new SubsamplingLayer.Builder(PoolingType.MAX)
                .kernelSize(samplingSize,samplingSize)
                .stride(samplingStride,samplingStride)
                .build())
            .layer(2, new ConvolutionLayer.Builder(kernelSize, kernelSize)
                //Note that nIn need not be specified in later layers
                .stride(kernelStride, kernelStride)
                .nOut(secondKernelAmount)
                .activation(Activation.IDENTITY)
                .build())
            .layer(3, new SubsamplingLayer.Builder(PoolingType.MAX)
                .kernelSize(samplingSize,samplingSize)
                .stride(samplingStride,samplingStride)
                .build())
            .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
            .layer(5, new
OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
                .nOut(classCount)
                .activation(Activation.SOFTMAX)
                .build())
            .setInputType(InputType.convolutionalFlat(28,28,1)) //See note
below
            .backprop(true).pretrain(false).build();

        MultiLayerNetwork net = new MultiLayerNetwork(conf);
        net.init();
        net.setListeners(new ScoreIterationListener(10));



        log.debug("Training model...\nTotal num of params: {}",
net.numParams());
        // evaluation while training (the score should go down)
        for( int i=0; i<nEpochs; i++ ) {

            long start = System.nanoTime();
            net.fit(trainIter);
            long end = System.nanoTime();
            trainingTime = ((end - start) / 1000000000);

            log.info("*** Completed epoch {} ***", i+1);

            log.info("Evaluate model....");
            start = System.nanoTime();
            Evaluation eval = net.evaluate(evalIter);
            end = System.nanoTime();
            testingTime = ((end - start) / 1000000000);
            log.info(eval.stats());
            evalIter.reset();
        }

        log.info("****************Example finished********************");
        log.info(String.format("Seed: %d\nTraining took: %f 
seconds\nEvaluation took: %f seconds", seed, trainingTime, testingTime));
        log.info(String.format("The convolution kernel size was %d and the 
stride was %d", kernelSize, kernelStride));
        log.info(String.format("The subsampling kernel size was %d and the 
stride was %d", samplingSize, samplingStride));
        log.info(String.format("The first and second convolutional layers used 
%d and %d kernels respectively", firstKernelAmount, secondKernelAmount));
    }

    public static void setPaths(int type){
        String basePath =  "C:\\Users\\addo_a\\Downloads\\MNIST 
Variants\\Finished CSVs\\";
        switch (type){
            case 0: //noise
                trainFilePath = basePath + "mnist-with-awgn-train.csv";
                evalFilePath = basePath + "mnist-with-awgn-eval.csv";
                break;
            case 1: //low contrast and noise
                trainFilePath = basePath + "both-train.csv";
                evalFilePath = basePath + "both-eval.csv";
                break;
            case 2: //motion blur
                trainFilePath = basePath + "mnist-with-motion-blur-train.csv";
                evalFilePath = basePath + "mnist-with-motion-blur-eval.csv";
                break;
            case 3: //background images
                trainFilePath = basePath + "background-image-train.csv";
                evalFilePath = basePath + "background-image-eval.csv";
                break;
        }
    }

}
```