Investigating the Effect of Convolutional Neural Network Architecture on Accuracy and Computational Efficiency of Image Classification

Research Question: To what extent do the amount of convolution layers and kernel size affect a convolutional neural network's speed and accuracy in image classification?

Extended Essay Subject: Computer Science

3937 words

Table of Contents

1. Introduction

In recent years, with advances in computer hardware and software, many artificial intelligence models have seen practical applications in a wide variety of fields; such as medicine, biometric authentication, language processing, and object detection (Kumar, 2021). A particular type of artificial intelligence model that has shown incredible promise and widespread adoption is the artificial neural network, often abbreviated as ANN. While there are many types of artificial neural networks, this paper will focus on convolutional neural networks; networks specifically designed to recognize learnable 'features' in input data and use said features to aid in classifying said data. As such, convolutional neural networks (CNNs) are commonly used for image processing (also known as 'computer vision,') although they are also capable of processing a wide variety of data types such as video, audio, and financial data (Kumar, 2021).

CNNs require incredibly large and varied datasets in order to effectively 'learn,' and thus are computationally expensive to train. The architecture of a network is imperative in determining its efficiency and accuracy; both metrics which researchers and engineers have been working to optimize since neural networks' inception. ANNs, and CNNs by extension, are considered to be 'deep learning' models as they contain many nonlinear data processing layers that effectively 'make decisions' about the information they receive (O'Shea, 2015).

This paper aims to explore the effect of a network's 'depth' on its performance; the amount of hidden layers, as well as the breadth of information that is passed through such layers. One significant factor that affects this breadth is kernel size. While much research is being conducted on new kinds of layers, activation functions, and optimization methods in neural networks; there is no definitive answer as to what the 'perfect structure' for a particular dataset will be. CNNs go through many design iterations before being deployed, and while it is

essentially impossible to conclude a ‘perfect’ model, this paper is intended to serve as a guide in foundational network design; attempting to conclude some ‘best practices.’

Given that CNNs are being used as assistive diagnosis tools in medicine, as they are able to analyze medical scans and search for characteristics of illness, and are used in self-driving vehicles to interpret images of their surroundings (Kumar, 2021), this paper’s design suggestions could prove incredibly useful in improving safety, accuracy, and quality of life.

This investigation was conducted by programming a simple CNN model in TensorFlow and making adjustments to depth and kernel size with each iteration; which was then trained multiple times per iteration on publicly available datasets. Visual representations of the results were created, and then analyzed to achieve logical and mathematical conclusions regarding the data.

2. Background Understanding

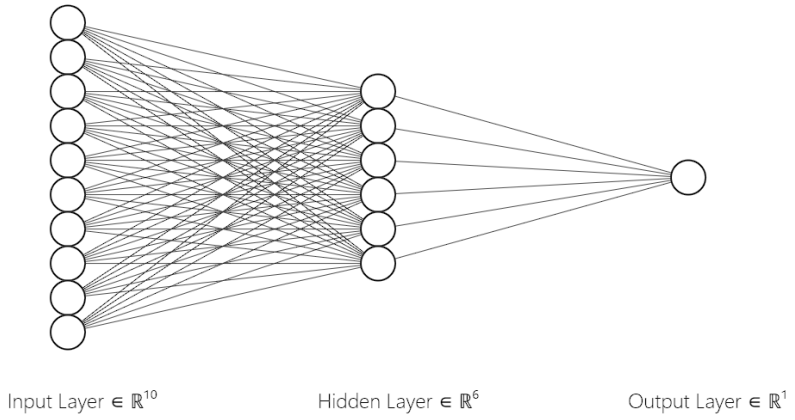
2.1 An Understanding of Machine Learning and Neural Networks

Machine learning (ML) describes the process of a computer program 'learning' to perform an action that it has not been directly programmed to do. Machine learning makes use of algorithms to allow computers to develop their own method of performing a task; algorithms that need to be 'trained' on data. ML algorithms can be either *probabilistic* or *deterministic*: where a probabilistic algorithm will not always 'think' the same way about the same piece of data, whereas a deterministic algorithm will always output the same result.

Convolutional neural networks are a subset of a category of machine learning algorithms called artificial neural networks. These neural networks are heavily inspired by the function of biological systems; particularly the human brain (O'Shea, 2015). Artificial neurons are constructed and form connections between one another; forming large webs of connections that function to mimic thought. These neurons are arranged linearly in layers, and in traditional layers ('dense layers,') each neuron is connected to every neuron in the layer before and after it.

Figure 1

A visual representation of a simple neural network; composed of input neurons, a hidden (dense) layer, and an output neuron.



Most machine learning models; particularly neural networks, perform *supervised learning*; with the 'supervision' being the presence of labels assigned to each item in a dataset (O'Shea, 2015). For example, each image in the MNIST dataset, which contains tens of thousands of images of handwritten digits from 0 to 9, has a label associated with it that was assigned by a human. Thus, a model trained on the MNIST dataset would be *classifying* input images; making a decision as to which number it is most 'confident' it is seeing, and then adjusting its 'thought process' according to whether or not its prediction matches up with the label. Convolutional neural networks are also trained in this way.

2.2 Understanding Convolutional Neural Networks

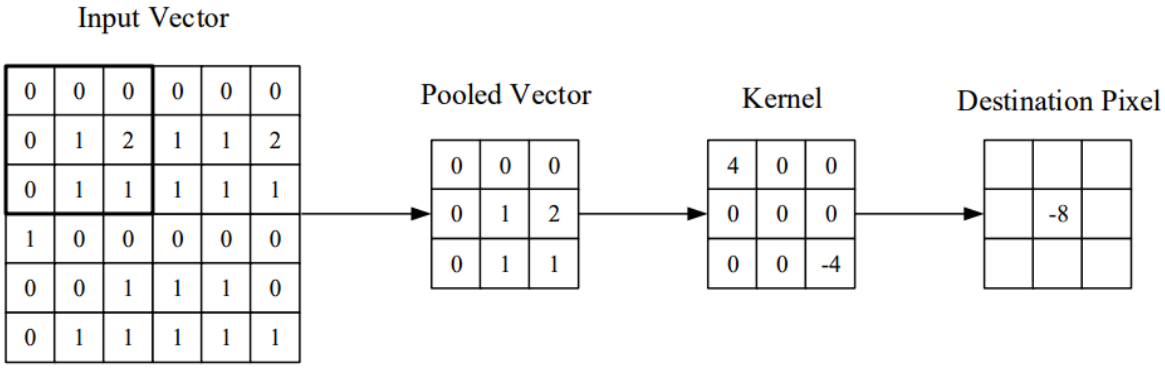
Convolutional neural networks behave similarly to traditional artificial neural networks, but the manner in which they handle input data and process it between layers is quite different. Inputs in neural networks are generally taken as *tensors*, which are essentially stacks of matrices (Görner, 2021). Traditional ANNs look at data in only one dimension; if a 28x28 image were to be processed, it would need to be converted into 784 input neurons. The opposite happens with CNNs, as a dimension, referred to as 'depth,' is added to the input, and thus the image's data could be contained within the matrix [28, 28, 1].

Convolutional layers are designated as such due to the mathematical operation of convolution that they perform, which is carried out by learnable sliding filters called kernels. Kernels are matrices that are moved over the input data, and assigned a particular dimension. They can be thought of as 'windows,' or activation maps, that 'fire' when they slide over a particular feature at a given position of the input, although features are not spatially dependent. The scalar product is calculated for each value in the kernel, and each kernel produces its own activation map; which is essentially a map that plots the likelihood of a certain feature being

present in a particular region of the input (O'Shea, 2015). There can be more than one kernel in each layer, in fact, the amount of kernels in a layer dictates its depth. Generally, the amount of kernels will increase from layer to layer in a CNN, as a smaller amount of 'simpler' features can then be extracted and evaluated to find more complex features.

Figure 2

A visual representation of a kernel operating within a convolution layer. The kernel's centre element is placed directly over the input pixel, and then replaced with a weighted sum of itself and nearby pixels (O'Shea, 2015, p. 6).



2.3 Neural Network Learning and Optimization

CNNs are much more computationally efficient than traditional ANNs due to many optimizations made in their structure that streamline input processing. In traditional ANNs, each neuron is given its own 'activation' function; a linear transformation is applied to its input. A nonlinearity function is then applied to the transformed input; this is necessary, as otherwise all layers could be mathematically simplified into one linear layer (Dodge, 2016). Each neuron is thus tasked with making its own 'decision,' which can be represented through the equation $f(x) = \phi(w^T x + b)$, where x is the input, w^T is the weight vector (a measure of how much the input should be 'weighed' in relation to the entire network,) b is a 'bias' term that is added or

subtracted, and ϕ is a nonlinearity function. In CNNs, weight vectors are shared between neurons in a layer, hence the same kernel being slid over many vectors of the input (Dodge, 2016).

After applying weights, biases, and activations, all neural networks converge in a 'fully-connected layer,' or multiple. The final fully-connected layer in a network is responsible for making the final decision and communicating a network's confidence in its classification of the input (Albawi, 2017). For each possible classification, a value (usually between 0 and 1) is provided, the network's guess as to what the probability of its classification being correct is. In order for output to be expressed in this way, an activation function called the softmax function is typically applied to the last layer. It works to normalize the vectors and create a set of values between 0 and 1 with sharp differences, by dividing an exponential form of a vector's weighted sum and bias by the sum of the vector's absolute values (Görner, 2021). The equation can be expressed as $softmax(L_n) = \frac{e^{L_n}}{||e^{L}||}$.

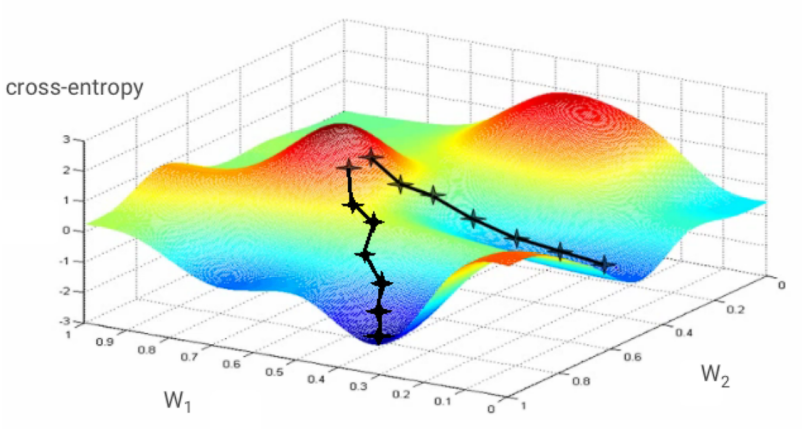
In order to learn effectively, regardless of whether or not it predicts the correct answer, an ANN makes use of a loss function to adjust randomly generated weights and biases so that its 'confidence' in the correct classification is increased, and its confidence in the incorrect classifications are decreased. The cross-entropy function, $-\ \Sigma Y'_i \bullet log(Y_i)$, is commonly used to achieve this, where Y_i are computed possibilities, and Y'_i is the actual probability expressed as a value of zero or 1, 'one-hot encoded.' (Görner, 2021). The partial derivatives of cross-entropy relative to all weights and biases are computed, and a 'gradient' is obtained; a gradient that is then minimized. Parameters are updated by a small fraction of the gradient known as the

network's learning rate, and an attempt is made to approach the global minimum; however it is highly difficult to find due to the presence of many local minima (Görner, 2021).

The process of gradient descent (minimizing the gradient) is often performed using a special technique called stochastic gradient descent, where the gradient is computed on a 'batch' of inputs. This better represents the constraints imposed by different example data, and is thus much more likely to converge towards the solution. An incredibly common algorithm that is used in stochastic gradient descent is that of Adam, a computationally efficient optimization algorithm that requires little tuning and is effective in analyzing complex networks (Kingma, 2014).

Figure 3

An illustration of gradient descent (Görner, 2021).



2.4 Convolutional Neural Network Hyperparameters

CNNs have various *hyperparameters* (not to be confused with learnable parameters) that are essential components of their design. Perhaps the most prevalent of these is depth, which can be varied through both the amount of kernels in a layer and the amount of layers present in the network, with increases in those hyperparameters increasing depth. Depth is crucial to a network's pattern recognition capabilities; as without enough of it, the network is unable to

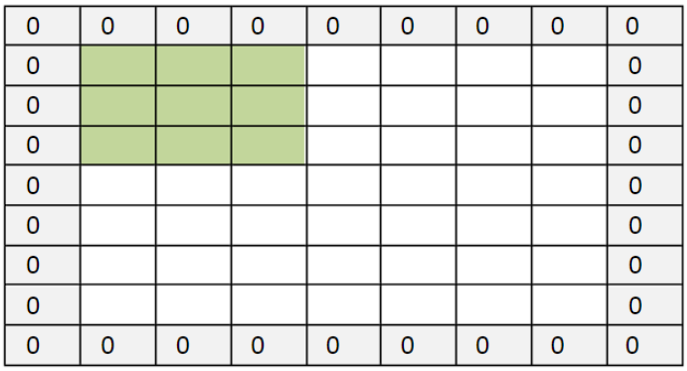
comprehend the full scope of the input data (O'Shea 2015). However, it is also necessary to monitor the effects of increasing depth, as a network that is overly complicated for its intended purpose may experience overfitting.

Subsampling stride is an incredibly important aspect of a convolution layer, as it dictates by how many pixels the kernel window will 'shift.' If stride is increased past 1, overlap in kernels' activations will decrease, and the output will have less spatial dimensions. (Albawi, 2017). Furthermore, kernel size impacts the complexity of feature analysis and the amount of information sent through the network. A larger kernel will be able to pick up on greater nuance, as there will be a greater amount of parameters for the network to learn; which should in theory increase performance. These two hyperparameters have a close relationship, and need to be chosen in accordance with one another.

Zero-padding is another essential technique implemented in CNN design. It is used to preserve output size between layers, and it achieves this by adding values of zero around pixel colour values at the edges of the input (Albawi, 2017). If kernel size is greater than 1x1, it is not possible to fit the same amount of kernels as pixels within the image's confines.

Figure 4

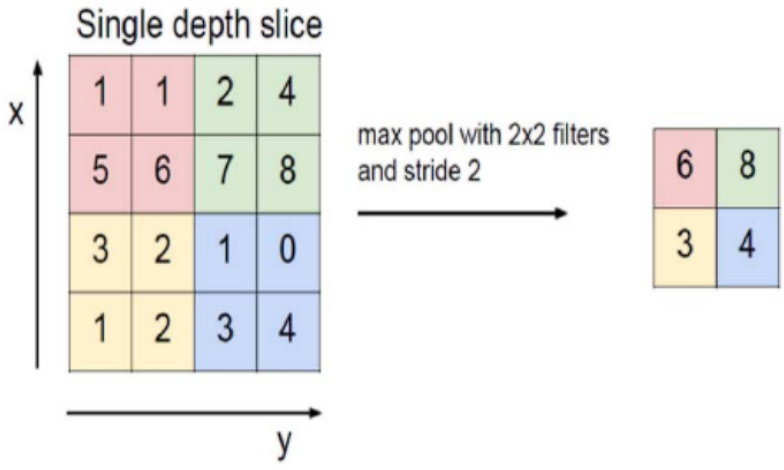
A visual representation of zero-padding (Albawi, 2017).



Although it was not used in the experiment, it is important to discuss the prominence of pooling layers in CNN design. Pooling layers aim to reduce the complexity of information processed by further layers through down-sampling; and can be considered as 'reducing an image's resolution' (Albawi, 2017). The most common method of pooling is max-pooling, where a 2x2 window with stride 2 picks out the largest value within the window and uses it to represent a feature's presence within said window.

Figure 5

An illustration of a pooling layer with window size 2x2 and stride 2 (Albawi, 2017).



2.5 Introducing Nonlinearity in CNNs

The most commonly used activation function in CNNs is the ReLU function (rectified linear unit,) defined as $ReLU(x) = max(0, x)$. Although it is not directly differentiable, its derivative can be expressed as $\frac{d}{dx}ReLU(x) = \{x > 0 = 1;\ x \leq 0 = 0\}$ (Albawi, 2017). ReLU has replaced saturated functions such as the sigmoid function, as they cause problems in backpropagation and lead to a 'vanishing gradient,' because their gradient is very close to zero almost anywhere but when x is close to 0 (Albawi, 2017, p. 4). Thus, ReLU was chosen for use in the experiment.

2.6 Dataset Splitting and Overfitting

Datasets are split into two portions; the training data, and validation data. There is typically a much greater amount of training data than validation data, but proportions vary by dataset. Training data can be thought of as a network's 'homework,' as the network is allowed to learn from each piece of this dataset and improve its understanding. Validation data acts as an 'examination,' where the network is tested on its capability to classify data it has never seen before; but it is not allowed to optimize the results of its classifications. Training data is typically fed through a network in batches, as discussed earlier, and a full iteration of the training dataset being passed through the network is referred to as an epoch. A CNN will learn from the entire training dataset through many iterations, or epochs, and it will typically see diminishing returns as it approaches its maximum capabilities.

When a network is exposed to a training dataset many times, it poses the risk of overfitting. Overfitting occurs when a network 'memorizes' a training dataset, and becomes less effective in classifying inputs in the validation dataset; and thus there is a greater discrepancy between validation and training accuracy.

3. Methodology of the Experiment

The aim of this experiment was to gain primary information that would allow for analysis of the relationship between kernel size and the amount of hidden layers on CNN performance (time and accuracy.) It was decided to perform this experiment as there was not a substantial amount of freely available information investigating this relationship despite its presence in the design of every CNN. A Keras model was created using TensorFlow and trained on four public datasets, with sixteen combinations of the independent variables per dataset. This experiment is limited by the small size and simplicity of the sample images; as hardware and time limitations prevented more complex models and datasets from being created and analyzed. Additionally, only one trial was performed for each variation of the variables, and due to the random initial parameters generated by Keras, fluctuations between seeds could have influenced data and research outcomes.

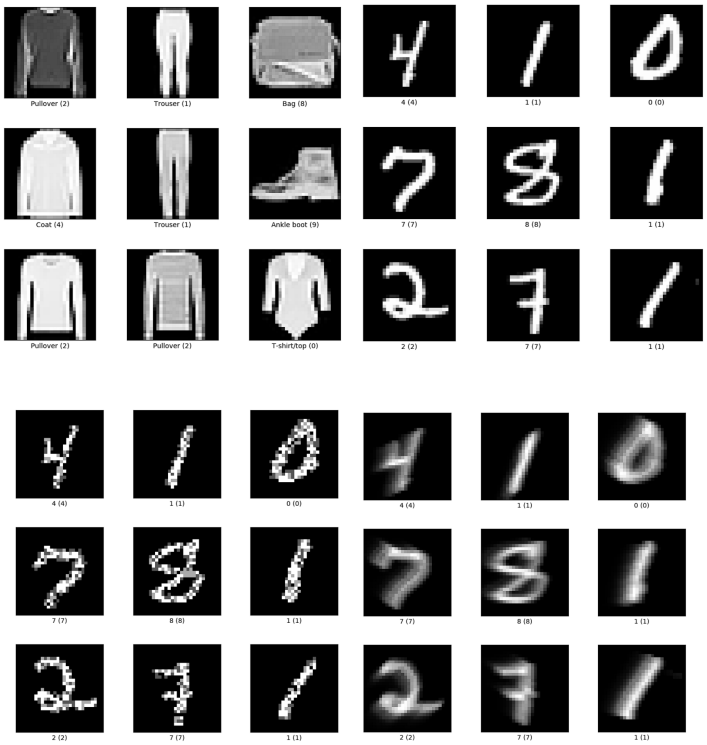
3.1 Chosen Datasets

The MNIST, Fashion-MNIST, and two variations of the MNIST-C datasets were chosen for this experiment due to their ease of access, relative simplicity, and status as a benchmark for CNN architectures. Each dataset is composed of 70,000 grayscale images with a 28x28 resolution; 60,000 training images and 10,000 test images (Deng, 2012.) The MNIST and MNIST-C datasets contain labelled images of handwritten digits from 0 to 9, the MNIST-C dataset containing various visual corruptions of the images (Mu, 2019.) The two chosen corruptions for this experiment were “shot noise” and “motion blur.” The Fashion-MNIST dataset contains the same number of labeled images of various clothing items, also in grayscale. Models generally find it more difficult to classify Fashion-MNIST images than the traditional

MNIST dataset (Xiao, 2017). The datasets were directly accessed through TensorFlow, imported, and reformatted for use in the Keras model.

Figure 6

A visualization of each of the MNIST datasets, in the order (from left to right,) Fashion-MNIST, MNIST, MNIST-C (Shot Noise), MNIST-C (Motion Blur.) Obtained from TensorFlow.



3.2 Dependent Variables

The two variables being measured are validation dataset accuracy and training time.

Time

The time taken to train the model (perform the model.fit() function) was measured using the python time library. The difference in system time was taken between when training was initiated and when it ended. The time difference was logged in a Google Sheet after each trial.

Accuracy

The network's validation dataset accuracy after the tenth and final epoch was stored in a dictionary and retrieved; as it best represented the model's accuracy after training. The validation dataset accuracy, "val_accuracy," was logged in a Google Sheet after each trial.

3.3 Structure of the Network

The network was programmed to have a variable structure, with an input layer of dimension (28, 28, 1), one convolution layer with a base kernel size of 3x3, a total of 12 filters, and stride 1. The rest of the layers were set as either pass-through layers (that return the same output as input,) or convolution layers with a base kernel size of 6x6 with a total of 24, 32, and 44 filters respectively, and stride of 2. The output of layer4 was then flattened for processing in two dense (traditional ANN) layers, with 200 and 10 neurons respectively. The ReLU activation function was used on all layers except for the last, which used the softmax function.

Figure 7

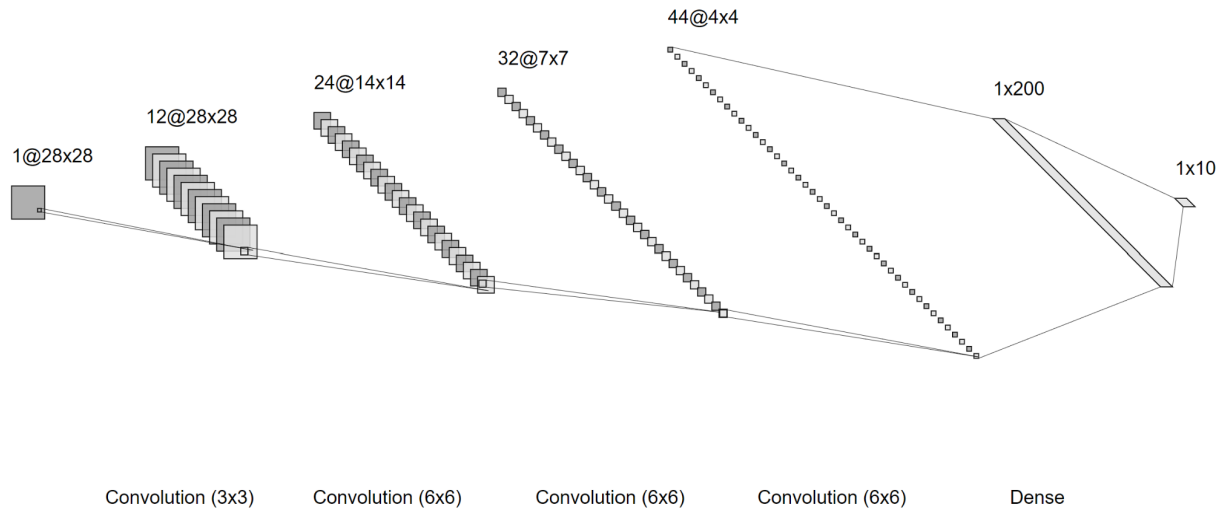
A snippet of the code that was used to create the model.

```
if z >= 2:
        layer2 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=24,
activation='relu', padding='same', strides=2)
      else:
        layer2 = tf.keras.layers.Layer()

      if z >= 3:
        layer3 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=32,
activation='relu', padding='same', strides=2)
      else:
        layer3 = tf.keras.layers.Layer()

      if z >= 4:
        layer4 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=44,
activation='relu', padding='same', strides=2)
      else:
        layer4 = tf.keras.layers.Layer()
```

```
model = tf.keras.Sequential([
          tf.keras.layers.Reshape(input_shape=(28, 28), target_shape=(28,
28, 1)),
          tf.keras.layers.Conv2D(kernel_size=(3 + y), filters=12,
activation='relu', padding='same'),
          layer2,
          layer3,
          layer4,
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(200, activation='relu'),
          tf.keras.layers.Dense(10, activation='softmax')
      ])
```

Figure 8

A visualization of the experimental CNN.



3.4 Procedure of the Experiment

A for loop was used to iterate through the modifications made to the hyperparameters. There were four variations of layer count; 1, 2, 3, and 4, and four variations of the base kernel size; subtracting 1, leaving it unchanged, adding 1, and adding 2.

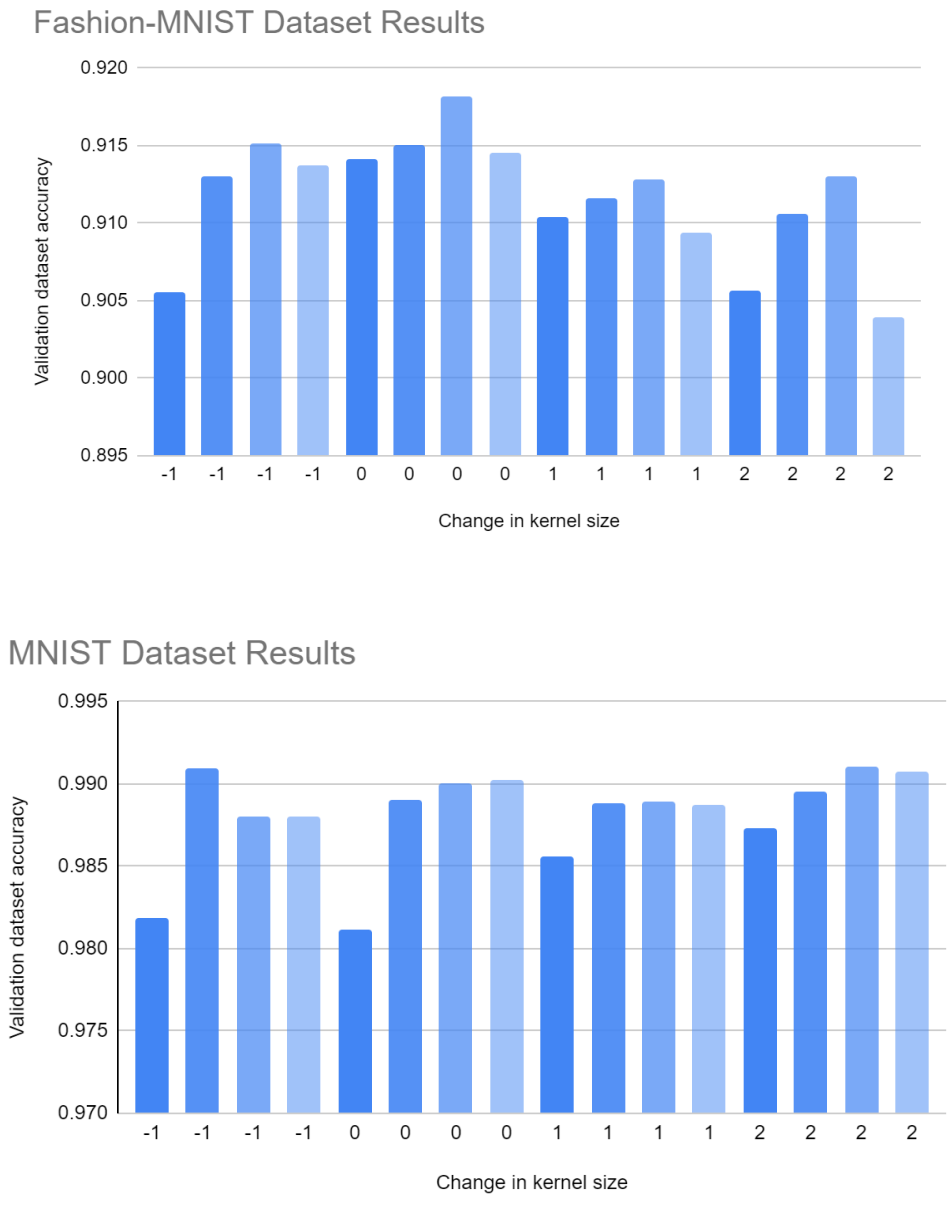
4. The Experimental Results

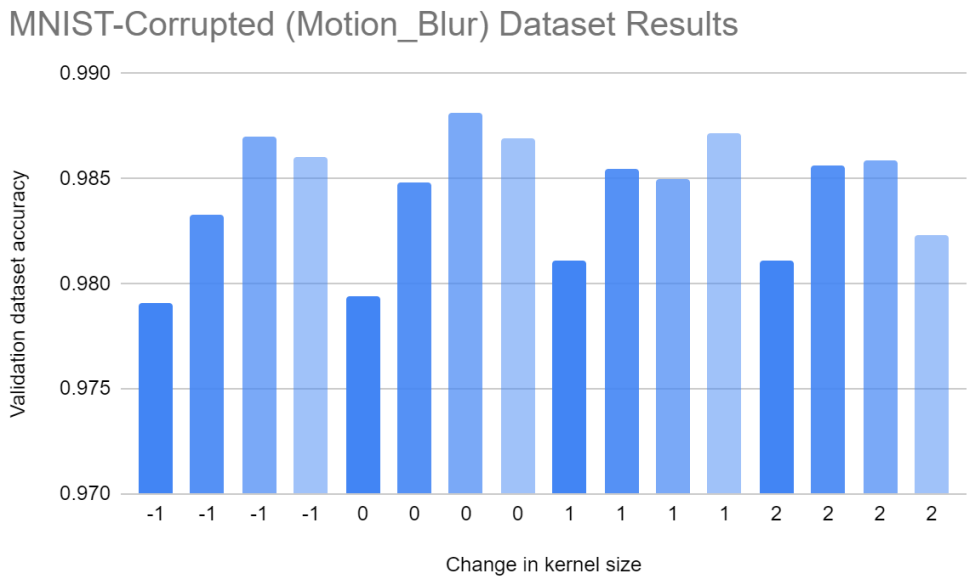
The graphs below display the results of the experiment. Time results were discussed but not represented graphically, as they were found to be less interesting and applicable.

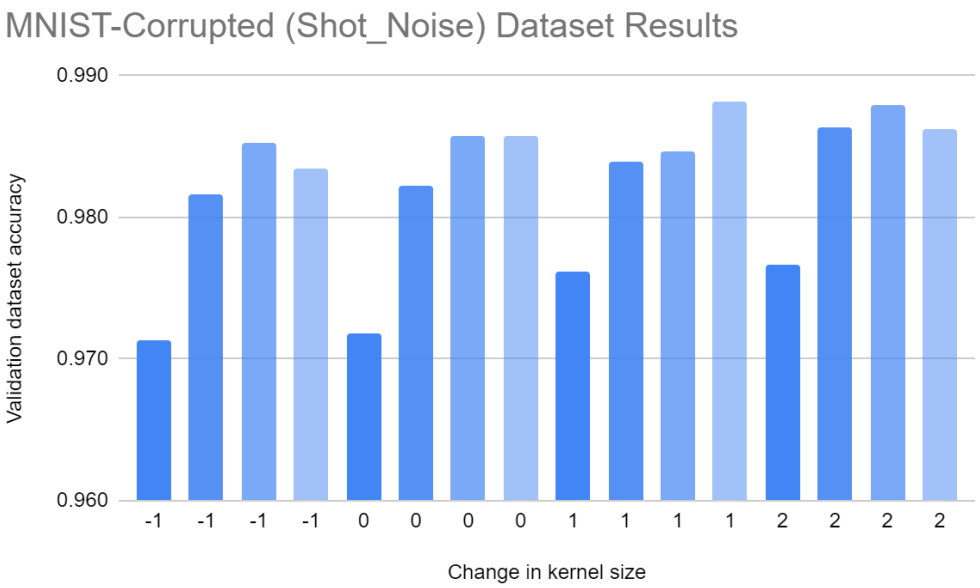
4.1 Graphical Representation

Figure 9

Graphical representation of results







4.2 Analyzing Data

Accuracy Results

The data was considerably more interesting than expected, as some of the observed trends contradicted suggestions from background research.

Firstly, it was observed that increasing the amount of hidden layers did not necessarily increase the model's accuracy; as diminishing returns in accuracy can be seen when going from dark blue to light blue (convolution layer count from 1 to 4.) After adding the second convolution layer, the discrepancy in accuracy was much smaller, and in many cases the model was more accurate with three convolution layers than four. The effect was different among the four datasets, however. The greatest discrepancy was seen in the MNIST-C shot noise dataset, as increasing the amount of hidden layers resulted in relatively consistent increases in accuracy. This is supported by a paper by S. Dodge et al, which suggests that depth protects networks from the adverse effects of image noise. It appears that this effect becomes less apparent as kernel size is increased.

It was surprising to see that the models performed very similarly on the corrupted versions of the MNIST dataset to the original MNIST. These three datasets saw a large improvement when increasing kernel size back to its base value, and the performance generally plateaued when increasing kernel size past it. Considering the relatively small size of the numbers, and the large amount of space surrounding them, it can be suggested that despite the kernels having access to more information, their accuracy barely seemed to improve due to the scope of the information with which they were presented being almost the entire image. Fashion-MNIST also experienced a similar effect, but its performance noticeably worsened as kernel size was increased past its base amount; suggesting that the convolution layers were receiving too much information. This could also have been the result of overfitting, and the effect of these parameters on the discrepancy between validation dataset and training dataset accuracy is something that may have potential to be explored.

Time Results

The time results were far less interesting, as training times typically ranged between 30 and 50 seconds. A similar trend to accuracy was seen as the independent variables were changed, but such variations in time proved rather inconsequential to the efficiency of training. This can be attributed to the relatively small variations in the amount of trainable parameters in the network, as increasing convolution layers decreases the size of the output to be processed by dense layers, which partially makes up for the gain in parameters that the convolution layers incur.

Trade-Offs

The base network configuration seemed to, in many cases, prove highly efficient and highly accurate. In terms of kernel size, this suggests that for image processing, it is often not worthwhile to increase the size of kernels so that they observe a significant portion of the image's contents. Rather, kernels should be kept relatively small and focused in order to better improve feature detection.

With layer count, it seems that less is sometimes more, as adding too much depth to the network can oversimplify the input data and fail to fully capture nuances of the image. Increasing layer count was observed to have a slightly more negative than positive effect, as it increased training time and sometimes decreased accuracy.

5. Conclusion

Overall, the experiment was conclusive across multiple datasets, in determining that kernel size in CNNs used for image classification should be significantly smaller than the image dimensions in order for nuance to be properly captured and for model accuracy to be optimized; while also requiring less time to compute. This results in a larger input to the dense layers at the end of a network, and thus greater decision-making capabilities. Convolution layer count also experienced diminishing returns in this simple application after being increased past 2; suggesting that a more even ratio of convolution layers and dense layers should be kept in mind when designing CNNs. This would optimize nuance being captured in an image

However, this relationship should be further explored. With greater computing power and more complex datasets, data could be obtained that would be more generally applicable to modern CNN designs. The difference in validation and training accuracy is also something that should be investigated, as it may be indicative of overfitting in a model. Use of pooling layers may also affect the nature of this relationship; as they present simplification of data. Subsampling stride is also heavily intertwined with kernel size, and including variations in stride may lead to richer conclusions and more applicable understanding.

This research may prove useful in creating image classification architectures in the fields of medicine, autonomous vehicles, and biometrics. The design suggestions presented by this paper will hopefully guide engineers and developers in the right direction when developing CNNs that positively impact human life.

6. References

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*. https://arxiv.org/pdf/1511.08458.pdf

Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2017, August). Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)* (pp. 1-6). Ieee. https://www.researchgate.net/publication/319253577_Understanding_of_a_Convolutional_Neural_Network

Dodge, S., & Karam, L. (2016, June). Understanding how image quality affects deep neural networks. In *2016 eighth international conference on quality of multimedia experience (QoMEX)* (pp. 1-6). IEEE. https://arxiv.org/pdf/1604.04004.pdf

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. https://arxiv.org/pdf/1412.6980.pdf

Görner, M. (2021, June 25). *TensorFlow, Keras and deep learning, without a PhD*. Google Codelabs. Retrieved from https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist#0

Arif, R. B., Siddique, M. A. B., Khan, M. M. R., & Oishe, M. R. (2018, September). Study and observation of the variations of accuracies for handwritten digits recognition with various hidden layers and epochs using convolutional neural network. In *2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT)* (pp. 112-117). IEEE. https://arxiv.org/pdf/1809.06187.pdf

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747. https://arxiv.org/pdf/1708.07747.pdf

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6), 141-142. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/MNIST-SPM2012.pdf

Mu, N., & Gilmer, J. (2019). Mnist-c: A robustness benchmark for computer vision. *arXiv preprint arXiv:1906.02337*. https://arxiv.org/pdf/1906.02337.pdf

Kumar, A. (2021, November 2). *Real-world Applications of Convolutional Neural Networks*. Data Analytics. Retrieved from https://vitalflux.com/real-world-applications-of-convolutional-neural-networks/

7. Appendix

Below is the code that was run for my experiment. It uses TensorFlow and Keras to import datasets, format them through an information pipeline, and train 64 different models with slight variations in layer structure and kernel size. The Google Sheets API was used to log experiment data to a digital spreadsheet; this required a .json file containing a unique access key that connects to my personal Google account.

The code was run through Google Colab, using a free Nvidia Tesla K80 GPU that was accessed through cloud computing.

```
import gspread
import tensorflow as tf
import tensorflow_datasets as tfds
import time
from oauth2client.service_account import ServiceAccountCredentials
scope =
["https://spreadsheets.google.com/feeds",'https://www.googleapis.com/auth/
spreadsheets',"https://www.googleapis.com/auth/drive.file","https://www.go
ogleapis.com/auth/drive"]
creds =
ServiceAccountCredentials.from_json_keyfile_name('/content/creds.json',
scope)
client = gspread.authorize(creds)
sheet = client.open("extended essay experiment results").sheet1
data = sheet.get_all_records()

nValues = [-1, 0, 1, 2]
datasets = ['mnist', 'fashion_mnist', 'mnist_corrupted/shot_noise',
'mnist_corrupted/motion_blur']
layerCount = [1, 2, 3, 4]
trialCount = 0

for x in datasets:
  for y in nValues:
    for z in layerCount:
      dataset = x
```

```
      (ds_train, ds_test), ds_info = tfds.load(
          dataset,
          split=['train', 'test'],
          shuffle_files=True,
          as_supervised=True,
          with_info=True,
      )
      if z >= 2:
        layer2 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=24,
activation='relu', padding='same', strides=2)
      else:
        layer2 = tf.keras.layers.Layer()

      if z >= 3:
        layer3 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=32,
activation='relu', padding='same', strides=2)
      else:
        layer3 = tf.keras.layers.Layer()

      if z >= 4:
        layer4 = tf.keras.layers.Conv2D(kernel_size=(6 + y), filters=44,
activation='relu', padding='same', strides=2)
      else:
        layer4 = tf.keras.layers.Layer()


      def normalize_img(image, label):
        """Normalizes images: `uint8` -> `float32`."""
        return tf.cast(image, tf.float32) / 255., label

      ds_train = ds_train.map(
          normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
      ds_train = ds_train.cache()
      ds_train = ds_train.shuffle(ds_info.splits['train'].num_examples)
      ds_train = ds_train.batch(128)
      ds_train = ds_train.prefetch(tf.data.AUTOTUNE)

      ds_test = ds_test.map(
          normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
      ds_test = ds_test.batch(128)
```

```
      ds_test = ds_test.cache()
      ds_test = ds_test.prefetch(tf.data.AUTOTUNE)

      model = tf.keras.Sequential([
          tf.keras.layers.Reshape(input_shape=(28, 28), target_shape=(28,
28, 1)),
          tf.keras.layers.Conv2D(kernel_size=(3 + y), filters=12,
activation='relu', padding='same'),
          layer2,
          layer3,
          layer4,
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(200, activation='relu'),
          tf.keras.layers.Dense(10, activation='softmax')
      ])

      model.compile(optimizer='adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

      model.summary()

      trainingBegin = time.time()

      history = model.fit(ds_train, epochs=10, validation_data=ds_test)

      trainingEnd = time.time()
      trainingTime = trainingEnd - trainingBegin
      trialCount += 1
      print("training time is", trainingTime)
      print("accuracy value is", (history.history['val_accuracy'])[9])
      print(x)
      print(y)
      print(z)
      print(trialCount)
      sheet.update_cell((trialCount+1), 1, trialCount)
      sheet.update_cell((trialCount+1), 2, y)
      sheet.update_cell((trialCount+1), 3, z)
      sheet.update_cell((trialCount+1), 4, x)
```

```
      sheet.update_cell((trialCount+1), 5,
(history.history['val_accuracy'])[9])
      sheet.update_cell((trialCount+1), 6, trainingTime)
```