Introduction

I. Summary

Background

Convolutional Neural Networks have been studied intensively within the space of image recognition tasks and computer vision. A popular, and well known implmentation regarding character recognition involves the famous MNIST dataset which is often considered as the 'hello world' version of machine learning. However, this problem becomes much more complex when we distort the images, and have to deal with occlusions, noise, distractors, font types and sizes, and other characteristics that exceed the constraints of a curated dataset for character recognition. When taking into account real world images, this becomes a difficult problem to solve without the help of deep learning networks. These networks allow us to employ algorithms that think in a multi-layered fashion, combining linear and nonlinear operations on the pixel values of the data set to develop the best results in predicting characters within a given image. A convolutional network can be thought of as an architecture that breaks down different segments of an image into different feature maps, which are then compiled to train the network on a set of classes. In this case, we have 10 classes: for numbers 0-9. These networks are a combination of affine and nonlinear transformations that allow computationally efficient ways of summarizing the activations (pooling layers) of adjacent neurons with a unique response (Fukushima, LeCun) that we can later use to predict the class it is most closely aligned to (softmax layers).

Components

There are different types of components that exist in a CNN as it relates to this project. These layers are the 'hidden' components to a CNN that allow the algorithm to make decisions towards weighting specific activations, aiding in the development of feature maps and positive results.

- Convolutional2D (First Layer is *Input Layer*)
- Fully Connected (Dense Layers)

- Fully Connected SoftMax Layer
- MaxPool

A **2D convolutional layer** is simply a layer that convolves the image with a number of filters that have a specific size (3x3 for example). This results in what we call 'feature extraction' and are essential in the formation of a neural network. Each layer results in an increase of the output of the array that is passed to the next layer, and it is these contiguous relationships that are analyzed in the production of weights for certain features in the training images. In these layers we perform 'RELU' activations, which stand for rectified linear units (nonlinear layer) which essential perform the a f(x) = max(0, x) function to the activations—which results in all negative values being clipped to 0. It is interesting to note that other nonlinear functions exist such as tanh, and sigmoid – but it has been proved that RELU activations are best for optimizing accuracy and efficiency.

A **fully connected layer is a dense layer**, and is named as such since every input is given a weight, and a corresponding output. These layers are usually set towards the end of a CNN architecture, since they take all the features and activations from the previous layers to develop consistent weights which are used in returning the correct class chosen for that input. This leads to the **last softmax layer** which is a dense layer with a specific number of classes (corresponding to the input passed), which uses a softmax activation to derive probabilities given the weights and inputs passed for each activation. This last layer is what provides us with the probability distribution of each class (0-9) for the given image(s).

A **MaxPool layer** is a layer that finds the maximum value found in a 2x2 square (size is arbitrary, but usually 2x2), and can be thought of as the maximum value of a moving window across a 2d input space. For a 2x2 kernel, it decreases the layer output by 75% and is important in reducing the number of parameters within the model (down-sampling). It also helps to summarize the results from a convolutional filter by extracting the maximum value found in the square window as it moves across the layer.

Another component of CNNs not mentioned above is the use of a **DropOut** function. This essentially tells the model to drop a percentage of the results, to ensure nonlinearity and a better accuracy overall. It can be thought of as shifting the focus of the model to rely less on adaptations to the training data. It is used primarily to regularize the data and prevent overfitting, while efficiently performing model averaging over each epoch.

Lastly the optimizer used for these networks were based on the **Adam** (adaptive moment estimation) **algorithm** which was initially proposed by Diederik Kingma and Jimmy Lei Ba, which implements adaptive estimates of lower-order moments in optimizing stochastic functions. This method is computationally efficient, and straightforward to implement. It is also invariant to diagonal rescaling of the gradients (Kingma et. al). What makes Adam different from stochastic gradient descent algorithms is that it only requires first-order gradients, and computes adaptive learning rates for different parameters for the 1st and 2nd order memoents of the gradients calculated. This means that it is memory efficient, and not restricted to one complete learning rate for the entire set of activations and their gradients during backpropagation. This method also works well with noisy datasets, and produced much better results than SGD during testing and implementation, and it is worthwhile to note that the authors describe Adam as a combination of the ‘advtanges of two recently popular optimization methods’, AdaGrad and RMSprop (Kingma et. Al, 10).

II. Methods

Network Architectures

Deep learning has enabled us to view these problems as a series of linear and nonlinear transformations to datasets that use complex algorithms to develop weights for every activation in each layer of the network. It has been well documented that the performance of this approach in character recognition increases as the depth of the network increases (Goodfellow et. al), and I took a similar approach in developing my own networks. **For brevity, Table 1 has omitted the MaxPool/Dropout Layers (.25, .25, .5 respectively). Note: All networks had a MaxPooling and Dropout after each ConvBlock.**

Network	13 Layers Network	VGG13 Adam	VGG10 Adam DG 20 256	VGG 16 Custom DNN	VGG16 Transfer Learning
Filename	vgg13_adam_datagen_30_128	vgg13_adam_ep20_32.h5	vgg10_adam_datagen_20_256	vgg16_custom_dnn.h5	pretrained_vgg16_datagen_2_ep20_256
Optimizer	Adam	Adam	Adam	Adam	Adam
Kernel Size	3x3	3x3	3x3	3x3	3x3
Batch Size	128	32	256	32	256
Epochs	30	20	20	20	20
ConvBlock 1	Filters: 32, Layers: 2	Filters: 32 Layers: 2	Filters: 32, Layers: 3	Filters: 3, Layers: 2	Pretrained Model
ConvBlock2	Filters: 32, Layers: 2	Filters: 32 Layers: 2	Filters: 64, Layers: 4	Filters: 6, Layers: 2	Pretrained Model
ConvBlock3	Filters: 64, Layers: 3	Filters: 64, Layers: 3	None	Filters: 12, Layers: 3	Pretrained Model
ConvBlock4	Filters: 64, Layers: 3	Filters: 64, Layers: 3	None	Filters: 24, Layers: 3	Pretrained Model
ConvBlock 5	None	None	None	Filters: 24, Layers: 3	Pretrained Model
FC-1	Input: 4096	Input: 4096	Input: 1024	Input: 300	Pretrained Model
FC-2	Input: 4096	Input: 4096	Input: 1024	Input: 300	Input: 4096
FC-SoftMax	Input: 10	Input: 10	Input: 10	Input: 10	Input: 10
Loss/Accuracy (Test)	Loss: .404, Accuracy: .884	Loss: 1.75, Accuracy: .44	Loss: .687, Accuracy: .766	Loss: 1.06, Accuracy: .698	Loss: nan Accuracy: .06

Table 1: An overview of the network architectures for this project. (Not including experimental models) Note that the MaxPool Layers and DropOut Layers have been ommitted for brevity. Also note that for the VGG inspired networks, the layers were reduced by half to shorten training time.

Data Augmentation

A common practice in the field of Machine Learning is the principle of data augmentation. Oftentimes, we are constrained by the amount of data we are given, and we still seek to find ways to train models on incomplete dataset, without the need for compiling more images. In this case we are using the Keras.applications.ImageDataGenerator() function to account for rotation invariance, scale invariance, reflectance invariance, and a variety of other image transformations to train our models. This enables the algorithm to learn to identify multiple representations of each image in the dataset, resulting in a more robust algorithm. This study compared the impacts of using data generators to provide robust profiles for the algorithms to use in extracting features and creating feature maps.

ImageDataGenerator Settings:

- Rotation Range = 180 degree changes
- Width shift range = .2
- Height shift range = .2
- Shear range = .2
- Zoom range= .2
- Horizontal flip = True

PreProcessing

In general there was minimal preprocessing. The mean of the complete dataset was subtracted from each image, and the axises were shifted to create a 'channels_last' data_format for the Keras pipeline. This channels_last format can be thought as:

Shape Example: (32, 32, 3, 73257) → (Shape[0], Shape[1], Depth, Num_Samples)

Network Settings

This project will focus on three main types of convolutional neural networks: a 13 layer network, VGG16 inspired model, and a VGG16 Transfer Learning Model. There are other models stored in the project files (under the directory /models) but I did not choose to focus on them for the scope of this project, and were merely used as experimental networks to train the SVHN dataset.

There are some global settings that each network shared which is listed below. The differences between the different network models came down to the number of filters, and the weights involved (pretrained vs. new).

Global Settings:

- **MaxPool**: Every Convolutional Block (of 2-3 layers) was followed by a MaxPool layer which had a 2x2 pool size, with a stride of 2. Inspired by VGG16 (Simonyan et. al)
- **Dropout:** After each MaxPool layer, a DropOut operation was performed to prevent overfitting. Each Dropout was set to .25, with the exception of the last droup out which was always .5. This was inspired by the VGG 16 Network Architecture (Simonyan et. al)
- **Epochs:** The default number of epochs were always set to 20 – though **please note** that an EarlyStop callback function was implemented *to ensure that the training stopped after the loss did not improve after 2 epochs.*

- **Optimizer:** An Adam Optimizer was used for all three networks. See Summary – Components above for a brief explanation of Adam Optimizer.

IV. Analysis

Results

The results of the different networks tested above showed some interesting generalities. Generally speaking, the deeper the network structure the longer it took to train—resulting in better and more stable accuracies. When looking at the history of the models, the deeper and more robust networks (larger filter sizes) increased more incrementally as opposed to larger up and downs in accuracies.

The best results were from the custom 13 layer network (Table 1, 13 Layers Network) which had anywhere from .88 - .92 test accuracies. This model consistently performed well when it came to the images in the test set, but did not exceed the state of the art.

Errors

Possible sources of errors are listed below:

- Epochs were too short – needed more complete training iterations as the accuracies did not converge quickly.
- Datagenerators elongated the needed training time, and developed lower accuracies overall with the test data set.
- Sub-optimal choice of hyper parameters for the Adam Method (Batch Size, Epochs, Steps Per Epoch)
- No Validation Accuracies (Validation Split) to more accurately analyze the effectiveness of a network at each training step – to ensure better overall accuracies and fine-tuning.

The potential sources of error listed above are often common errors in machine learning and convolutional networks – a lot of what is done in the state of the art involves testing the results of different hyper parameters using tools and resources like sklearn's (python package) meshgrid function. Using validation accuracies would have also helped understand and define the weak points of the network, as it would show a much more in-depth view of what the model was doing and learning.

The pretrained VGG Model had a lot of issues, when it came to transfer learning, and I believe it could have been optimized by fine tuning the previous weights in the unchanged layers of the model—instead of just freezing the previous weights. This would have led to a much more precise accuracy, as it would eventually learn to reduce 10000 classes to 1 (imagenet) if given enough time to learn. Possible reasons for why I obtained such a low accuracy may include a short training time, and a ineffecient organization of fully connected layers for training the network for the SVHN dataset.

Discussion/Analysis

In general, the networks developed in this study would have been better off if the models were trained over a longer period of time – with an augmented dataset. Even though this study used data generators, there were many more settings that I could have accounted for. Since the SVHN challenge proposes a much more challenging task than the classic MNIST database, I believe the results were aligned to my expectations for the project. I also believe that the study would have been much more efficient if I had used a meshgrid or program to test the outputs of multiple parameter choices, returning the most accurate model and it's settings afterwards.

V. Final Results

Results

The results were not optimal when it came to testing images that were outside the bounds of a 32x32 centered image as it is formatted for the 2nd datset from the Stanford SVHN website. On average it was able to pick 1-2 digits from each picture in the *graded_images/* directory, but it failed to extract useful features because of the loss in information from rescaling the images. The process, which is described in detail in the run_v3.py file, involves taking the set of images (with different angles and occlusions) and taking a sliding window approach to finding the digits in each image. The test images ranged from 300x400 pixels, and trained on a 32x32 centered image dataset—which is the size of the window I used to predict the digits using the models above.

On average I was able to obtain 77% accuracy – which consisted of mostly 1s and 2s. In my /*graded_images* directory, each picture had the numbers 812 in it. For each image I was able to extract the 1 and 2 fairly easily, but failed to extract the number 8. I believe this is due to the resizing issues from the pyramid (loss in information in reducing the sizes) and due to accuracy issues in the model (the way it was trained).

Model	Average Accuracy
vgg10_adam_datagen_20_256	77.77% (1 and 2 Partial Match Each Image)
pretrained_vgg16_datagen_2_ep20_256.h5	0%
vgg13_adam_ep20_32.h5	66.67 (2 Full Matches)
vgg13_adam_datagen_30_128	73.33% (1 Full Match)
vgg16_custom_dnn.h5	40% (Mostly 2s)

Discussion/Analysis

Compared to the state of the art, these networks did not perform well due to the fact that the models did not train for as long as they should have. Overall I was pleased to see that I was able to ascertain some full matches from the images I provided, but there were not comparable to the current state of the art. Augmenting my dataset further combined with better tuning of the parameters would have led to better results. Using the sliding window method for each image pyramid allowed me to maximize my ability to obtain the best possible data given my network structure.

References:

1. Simonyan, K & Zisserman, A 2015, ‘Very deep convolutional networks for large-scale image recognition’, paper presented to ICLR, 15 April 2015, viewed 26 November 2017

2. Goodfellow, I.J., Bulatov, Y., Ibarz, J., Arnoud, S. and Shet, V., 2013. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*.

3. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient based learning applied to document recognition. Proc. IEEE.

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

5. Kingma, D.P., and Lei, Jimmy, ‘ADAM: A Method for Stochastic Optimization’, paper presented to ICLR, 30 January 2017, viewed 26 November 2017