Brickwork Detection using Convolutional Neural Networks

Britton Chesley
University of Florida
bchesley97@ufl.edu

Christopher Crary
University of Florida
ccrary@ufl.edu

Wesley Piard
University of Florida
wespiard@ufl.edu

***Abstract*—Image recognition tasks have recently exploded in popularity due to the rise of deep learning, and, particularly, *convolutional neural networks* (*CNNs*). In brief, convolutional neural networks are a form of neural networks in which automatic feature extraction is performed mainly through neural network layers that are structured to represent convolution (filter) operations. Overall, convolutional neural networks have allowed for great strides in the fields of machine learning and AI, and it has been shown that they work extremely well for image classification and semantic segmentation tasks. In this paper, we introduce a system that accurately classifies images of four different types of brickwork masonry (and a fifth class representing all other possible things) via a convolutional neural network. Overall, for the relevant dataset, we found that classic machine learning approaches involving manual feature extraction were not as effective as a CNN. Overall, our final CNN model achieved 92.84% classification accuracy on a blind test set of 2,227 images.**

I. Introduction

This report will discuss the process we went through to attempt to develop a method of identifying four different types of brickwork masonry from 200 × 200 pixel RGB color images. In our previous work (Assignment #2) this was attempted using a *probabilistic generative classifier* with a dataset of only roughly 300 images. Now, however, there is a much larger dataset available with over 11,000 images to work with. In particular, such images are placed into one of five classes, with the distribution as follows:

Class 0 (17.1%): Not Brickwork
Class 1 (18.6%): Brickwork, Flemish Stretcher Bond
Class 2 (22.6%): Brickwork, English Bond
Class 3 (30.0%): Brickwork, Stretcher Bond
Class 4 (11.7%): Brickwork, All Other Types

The first obvious step for us was to apply the same feature extraction techniques that were used in Assignment #2 to the new full class dataset. Unfortunately, as illustrated in Figure 1, our features did not yield nicely-clustered classes like they did in Assignment #2.

Since these previous methods of feature extraction were not effective on the full class dataset, we considered a deep learning approach. Deep learning models automatically extract features from the raw data and can be very powerful in image classification tasks [1]. Preliminary results with deep CNNs provided better results than our previous traditional machine learning algorithms, and even better results were obtained by utilizing "wider" and "deeper" networks. However, our larger networks fell victim to overfitting, so implementing forms of regularization became crucial.

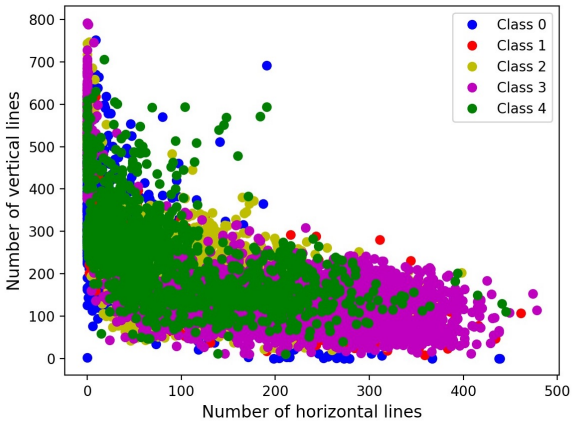

Fig. 1: Scatter plot of previous features on full dataset.

In our work, the regularization techniques dropout [2], early stopping [3], weight decay [4], and batch normalization [5], [6] were implemented in order to reduce overfitting on the training data and improve generalization on the validation set. These "regularizers" helped our models perform substantially better, with training and validation accuracies of over 98%.

II. Experiments

Our experiments relied largely on both (1) methodical trial and error and (2) results produced from methodical trial and error. In essence, an "alternating optimization" approach was conducted, where certain model parameters were optimized while most of the others (if not all) remained fixed. The remainder of this section should present meaningful conclusions that follow from fundamental experimental design principles.

A. Experimental Design Plan

The entire dataset available for training, validation, and testing consisted of a little more than 11,000 images. Although the dataset was somewhat large, it was still imperative that appropriate sizes be chosen for train, validation, and test sets, especially since the given data was not distributed evenly among the relevant five classes.

It was decided that 80% of the available dataset would be used for training/validation and that 20%

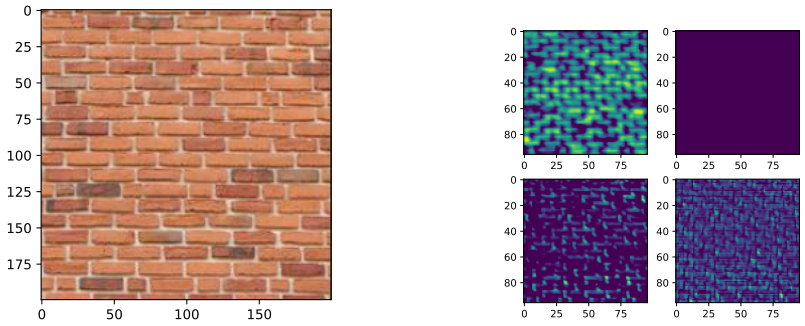

Fig. 2: Input image (left) and example feature maps (right) generated by CNN for given input.

would be used as a blind test set. Moreover, of the 80% that would be used for training/validation, stratified 5-fold cross-validation (where "stratification" would ensure that the five classes would be evenly represented within the folds) would be employed for the purposes of [1] deciding appropriate values for certain hyperparameters, [2] determining the most informative features, and [3] meaningfully validating any generated models. Overall, five folds were chosen for K-fold cross-validation so that there would still be a modest amount of images available for the validation set.

After creating training, validation, and test sets as described above, there were 7,284 images available for training, 1,822 images available for validation, and 2,277 images available for testing the final model.

B. Evolving the Model

Overall, it was decided that the evolution of our model (e.g., the network architecture, the values for certain hyperparameters, etc.) would rely almost entirely on the classification accuracy achieved after applying the validation set to the model.

Many model architectures were explored. Table I describes some of these architectures by providing the structure for the convolution and fully-connected layers within the network [1], the choice of loss function, and the final classification accuracies received by applying the architecture on the relevant training and validation set, respectively. Within the table, the series of base numbers given within the "Convolution Layers" and "Fully Connected Layers" columns (e.g., $3, 10, 5$ and $5, 5$ in the first data row) represent the number of convolution (filter) channels and number of neurons within each layer, respectively. In addition, for each of these base numbers, the subscript tuple of two elements represents the dropout and batch normalization regularization rates employed for the inputs of that layer, respectively. Note that the last fully connected layer for all models have 5 neurons, as we chose to use a one hot encoding of our class labels. Regularization strategies are discussed in the following subsection.

At first, smaller networks with only three convolutional layers and one fully connected layer were trained, but the best accuracies from these models were below 50%, e.g., architecture 1 within Table I. From this, much larger models began to be designed.

At the start, mean-squared error was used as the relevant loss function and ReLU as the relevant activation function. However, around the same time as larger models began to be created, it was decided that cross-entropy would replace mean-squared error, as cross entropy loss functions have been shown to work better with a one-hot encoding of the output labels [1], whereas mean squared error is typically used for regression tasks.

The larger models were successful, but achieving success with them was delicate. For example, overfitting was a pertinent issue that needed to be avoided via forms of regularization (Subsection II-C). All in all, it was discovered that wide and deep convolutional layers, along with wide and shallow fully connected layers led to the best results for the given problem. Utilizing progressively wider and deeper convolutional layers (e.g., architectures 3-15 in Table I) worked well seemingly because the additional layers developed feature maps on smaller and smaller images. In addition, utilizing wide but shallow fully connected layers (e.g., architectures 12-15 in Table I) worked well seemingly because there were more possible combinations of the features learned by the convolution layers. Figure **??** shows 4 feature maps learned in the first convolutional layer of our final model, where the features learned seemed to be the edges of the bricks or the pattern of the stucko.

The overall best model in terms of training and validation accuracy was architecture 15, listed in Table I, however architectures 12-14 performed very similarly, given the fact that they are very similar in architecture. The final model was chosen to be of Architecture 15, and, due to time constraints, the final model was trained on only one of the aforementioned five stratified K-folds, since this was the only even class distribution available.

Overall, after applying the blind test to the final

TABLE I: Description of Convolutional Neural Network Architectures.

CNN Architectures				
No.	Convolution Layers	Fully Connected Layers	Loss	Accuracy (%)
1	$3^{(3)}_{(0,0)}$, $10^{(3)}_{(0,0)}$, $5^{(3)}_{(0,0)}$	$5_{(0,0)}$, $5_{(0,0)}$	MSE	43.9, 38.5
2	$3^{(3)}_{(0,0)}$, $10^{(3)}_{(0,0)}$, $10^{(3)}_{(0,0)}$, $10^{(3)}_{(0,0)}$, $6^{(3)}_{(0,0)}$, $4^{(3)}_{(0,0)}$	$16_{(0,0)}$, $8_{(0,0)}$, $5_{(0,0)}$	MSE	86.6, 82.3
3	$3^{(3)}_{(0,0)}$, $256^{(3)}_{(0.4,0)}$, $256^{(3)}_{(0.4,0)}$, $128^{(3)}_{(0.4,0)}$, $50^{(3)}_{(0.4,0)}$, $10^{(3)}_{(0.4,0)}$	$180_{(0.4,0)}$, $100_{(0.5,0)}$, $20_{(0.5,0)}$, $5_{(0,0)}$	MSE	96.7, 70.0
4	$3^{(11)}_{(0,0)}$, $16^{(5)}_{(0,0)}$, $32^{(5)}_{(0,0)}$, $64^{(5)}_{(0,0)}$, $128^{(5)}_{(0,0)}$, $256^{(5)}_{(0,0)}$	$2{,}500_{(0.5,0)}$, $5_{(0,0)}$	CE	99.6, 87.1
5	$3^{(11)}_{(0,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$	$4{,}096_{(0.5,0)}$, $4{,}096_{(0.5,0)}$, $5_{(0.5,0)}$	CE	87.5, 82.3
6	$3^{(11)}_{(0.2,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$	$4{,}096_{(0.3,0)}$, $4{,}096_{(0.3,0)}$, $5_{(0.3,0)}$	CE	85.6, 79.3
7	$3^{(11)}_{(0.2,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$	$2{,}048_{(0.3,0)}$, $2{,}048_{(0.3,0)}$, $5_{(0.3,0)}$	CE	88.6, 80.1
8	$3^{(11)}_{(0.2,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$	$8{,}192_{(0.3,0)}$, $8{,}192_{(0.3,0)}$, $5_{(0.3,0)}$	CE	82.4, 79.7
9	$3^{(11)}_{(0.2,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$	$4{,}096_{(0.1,0)}$, $4{,}096_{(0.1,0)}$, $5_{(0.1,0)}$	CE	89.1, 84.6
10	$3^{(11)}_{(0,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$, $512^{(3)}_{(0.1,0)}$, $1{,}024^{(3)}_{(0.1,0)}$	$4{,}096_{(0.4,0)}$, $4{,}096_{(0.4,0)}$, $5_{(0.4,0)}$	CE	98.0, 80.8
11	$3^{(11)}_{(0,0)}$, $16^{(3)}_{(0.1,0)}$, $32^{(3)}_{(0.1,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$	$512_{(0.4,0)}$, $256_{(0.4,0)}$, $256_{(0.4,0)}$, $5_{(0.4,0)}$	CE	97.6, 88.7
12	$3^{(11)}_{(0,0)}$, $16^{(3)}_{(0.1,0)}$, $32^{(3)}_{(0.1,0)}$, $64^{(3)}_{(0.1,0)}$, $128^{(3)}_{(0.1,0)}$, $256^{(3)}_{(0.1,0)}$	$4{,}096_{(0.4,0)}$, $5_{(0,0)}$	CE	99.5, 89.7
13	$3^{(11)}_{(0,0)}$, $16^{(3)}_{(0.1,3)}$, $32^{(3)}_{(0.1,16)}$, $64^{(3)}_{(0.1,32)}$, $128^{(3)}_{(0.1,64)}$, $256^{(3)}_{(0.1,128)}$	$4{,}096_{(0.4,256)}$, $5_{(0,4,096)}$	CE	98.0, 95.0
14	$3^{(11)}_{(0,0)}$, $32^{(3)}_{(0.05,3)}$, $64^{(3)}_{(0.05,32)}$, $128^{(3)}_{(0.05,64)}$, $256^{(3)}_{(0.05,128)}$, $512^{(3)}_{(0.05,256)}$	$2{,}048_{(0.5,512)}$, $5_{(0,2,048)}$	CE	98.2, 98.0
15	$3^{(11)}_{(0,0)}$, $64^{(3)}_{(0.05,3)}$, $128^{(3)}_{(0.06,64)}$, $256^{(3)}_{(0.06,128)}$, $256^{(3)}_{(0.07,256)}$, $1{,}024^{(3)}_{(0.1,256)}$	$3{,}600_{(0.6,1,024)}$, $5_{(0,3,600)}$	CE	**98.2, 98.1**

model, a classification accuracy of 92.84% was achieved. A confusion matrix displaying the results of this test is shown in Fig. 4, and a learning curve for the training process of the final model is given in Fig. 3.

C. Regularization Strategies

Since it seemed that larger, more complex models were most appropriate for the problem at hand, much effort was put into exploring regularization techniques. Dropout was the first regularization technique implemented, and is inspired by ensemble learning methods, where neurons in a specific layer (or layers) are dropped with a user-defined probability, meaning they don't contribute to the weighted sum in the next layer. This has been shown to reduce overfitting since the architecture of the network is changing slightly every epoch. Different probabilities for the rate of dropout were tested, with smaller probabilities in the convolutional layers (> 0.2) and typical rates applied in the fully connected layer (~ 0.5). The convolutional layers required a smaller dropout rate so that the kernel weights could learn proper discriminative features. This helped improve model generalization significantly.

Early stopping was another regularization technique tried by our team. Early stopping involves training a neural network until the validation loss stops decreasing for a predefined number of iterations. So the model stops training before overfitting can occur. This increased model training time, however, but proved to be instrumental in preventing overfitting in our larger networks that contained around 100,000,000 parameters. An L2 regularizer was also added to the error function, but this led to poor performance so was discarded early on in the development process.

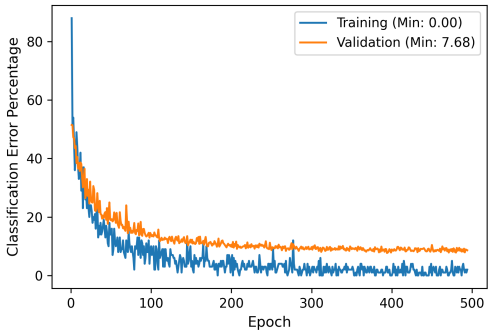

Fig. 3: Learning curves for final model.

Batch normalization is a relatively recent advancement, and normalizes a batch of input images to reduce internal covariate shift. This allows the model to be trained faster, and also acts as a regularizer, as the probability distributions of each layers inputs changes much less frequently.

At first, validation accuracy was tested on a variety of model size and architectures without performing random transformations on the input images. After this feature was implemented, model generalization improved substantially as the models were learning features invariant to the transforms that were being applied as described in the

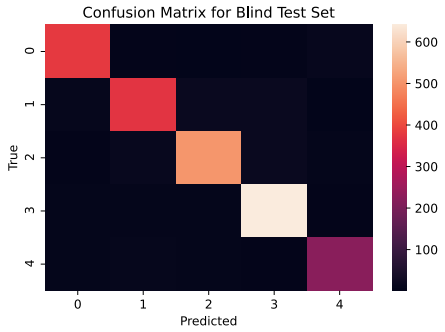

Fig. 4: Blind test set confusion matrix.

previous implementation section. The main changes applied to our model were in the error function, network depth and width, dropout rates, batch size, and learning rates. At first, batch size and learning rates were varied while the model architecture was held constant. The training loss and validation loss were printed every ten epochs of training, and the learning rate was chosen in most cases to be 0.001, as this reduced training time and provided stable results. Batch sizes from 1-256 were tested, where a batch size anywhere from 32-128 provided reasonable results (and allowed for faster training times due to increased parallelism in the GPU), so a batch size of 100 was chosen for most of the experiments. The "Adam" optimizer was used for all experiments. Adam uses adaptive learning rates for individual parameters leveraging the first and second moments of the gradient of the error function [7].

However, during early experiments with smaller networks, the networks became biased towards class 3 since this image is more frequent than the other images in the dataset. Thus, stratified K-folds cross validation was used to prevent our models from becoming biased towards a specific class.

III. Implementation

To develop our deep learning architectures with reasonable training times, we used *Google Colab Pro*. This allowed our team access to train models with much faster GPUs. All models were implemented in Python using *PyTorch* and *scikit-learn*, i.e., *sklearn*. The remainder of this section will overview how we managed the given training dataset, how we defined and trained our models, and how we validated our results.

A. Data Organization

First, the given dataset was loaded via the *numpy* package, and the labels were converted to a one-hot encoding. Then, the data was split into a training set and test set using sklearn's *train_test_split()* function, where 20% of the images were reserved for a final blind test set. The remaining 80% of the images were further split into five folds via stratified 5-fold cross-validation, as described in the previous section. A random state seed was utilized to keep splits consistent whenever separate training runs were performed.

B. Data Augmentation

For every fold, the training and validation sets were *separately* standardized/normalized with their respective means and variances. Sklearn transformations on the training data were also implemented to help the model learn features invariant to rotation, color skew, etc. These transformations included: 45 degree rotation, 90 degree rotation, perspective change, horizontal flip, or a random affine transformation. The transformations were applied randomly through PyTorch's *DataLoader* functionality.

C. CNN Architecture

The CNN was implemented using PyTorch's *nn* module. The architecture consists of five convolutional layers followed by two fully-connected linear layers. The following were applied to the output of every convolutional layers: the ReLU activation function, batch normalization, and max pooling (with a kernel size and stride of 2).

For the number of connections between each layer in the network architecture, refer to Table I, model number 15.

D. Training Process

The model was trained for a maximum of 500 epochs in batch sizes of 100. The learning rate was initialized to 0.001 with a maximum of 0.003 using the *Adam* optimizer [7] and a cross-entropy loss function.

For every epoch, batch by batch, a set of images was passed forward through the model and the outputs were used to compute loss, perform optimization, and perform back propagation.

To avoid stagnant models going without improvement all the way to 500 epochs, early stopping was implemented when training all networks. Validation accuracy was checked after every epoch and if there were 100 sequential epochs without improvement, the model was halted early to allow more models to be trained. Every successful model that ran to completion was then saved as a '.pt' file.

IV. Conclusion

For the majority of our team, this was the first application of deep learning. We thought it was a good introduction to the subject area and were constantly learning new ways to improve our validation accuracies by adding functionality over the course of over 100 training runs.

Overall, we are glad we made the decision to use a CNN architecture instead of manual feature extraction. It took a lot of manual hyperparameter tuning, but with an accuracy of almost 93% on our blind test set, we were content with the results.

REFERENCES

[1] S. Hijazi, R. Kumar, and C. Rowen, "Using convolutional neural networks for image recognition," *Cadence Design Systems Inc.: San Jose, CA, USA*, pp. 1–12, 2015.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[3] L. Prechelt, *Early Stopping - But When?* Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69.

[4] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[6] C. Garbin, X. Zhu, and O. Marques, "Dropout vs. batch normalization: an empirical study of their impact to deep learning," *Multimedia Tools and Applications*, pp. 1–39, 2020.

[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.