MLP Coursework 2: Exploring Convolutional Networks

s1879262

Abstract

The first part of the report explains the implementation of convolutional networks [Lecun et al.] using the MLP Framework. This involves the completion of classes `ConvolutionalLayer` for convolutional layers and `MaxPooling2DLayer` for performing max pooling on the layers. The second part of the report focuses on the construction, evaluation and comparison on a set of convolutional networks with different dimensionality reduction methodologies and hyperparameters applied on them. The amount of variables regarding the dimensionality reduction methodologies and hyperparameters results in a vast amount of combinations that can produce different models with different results. This can be proven cumbersome for scientists looking for the model that would provide the best results. So, the intention of the following experiments is to identify which of these dimensionality reduction methodologies are more efficient. The cause of this problem is also the reason why it is a difficult one to answer. While finding a single best solution is almost impossible, the insights that can be provided from the experimentation are still valuable.

1. Introduction

The first part of the coursework required the implementation of methods in two classes. Methods `fprop`, `bprop` and `grads_wrt_params` from class `ConvolutionalLayer` and methods `fprop` and `bprop` from class `MaxPooling2DLayer`. Class `ConvolutionalLayer` performs a convolution operation on the input images while class `MaxPooling2DLayer` implements pooling to combine neurons of a layer into a single neuron in a new reduced layer. In all methods of class `ConvolutionLayer`, the convolution function `scipy.signal.convolve2d` was used to compute the convolutions.

During the second part of the coursework a number of experiments were executed to test the different dimensionality reduction methodologies and their response to hyperparameter changes. The underlying questions that these experiments are trying to solve is which dimensionality reduction methodology performs better. The four dimensionality reduction methodologies used in the experiments are **max pooling**, **average pooling**, **dilated convolution** and **strided convolution**. To make the experiments clearer and because finding a single best methodology for every situation is not viable the experiments focused on comparing max with average pooling and dilated with strided convolution.

The initial experiments were performed with the predetermined hyperparameters on every dimensionality reduction methodology. The intial hyperparameter values were: batch size = 100, image height = 28, image width = 28, number of filters = 64, number of layers = 4 and weight decay coefficient = 0.00001. The number of epochs was set to 50 for all experiments in order to allow for the maximum amount of experiments in the allowed time frame and seed value was set to 7112018 for all experiments to allow reproducibility of the results.

For further experimentation only one hyperparameter was changed in order to allow for better comparison of its effect on the accuracy and efficiency of every model. The hyperparameters that were changed during the experiments were the **number of layers** and the **number of filters**. This process was performed for every dimensionality reduction methodology.

The dataset used for these experiments is the EMNIST Balanced dataset by Cohen et al. which is an extension of the original MNIST dataset. This also includes images of handwritten letters apart from the original images of handwritten digits. This dataset is split into training, validation and test sets based on the implementation of the `EMNISTDataProvider` class from the file `data_provides.py`. The sizes of the three sets are:

- Training: 100000 examples
- Validation: 15800 examples
- Testing: 15800 examples

Each example can be assigned to one of 47 classes (reduced from 62 due to inability to distinguish 15 lower case letters from their upper case counterparts after the data conversion process) by using a vector of 47 dimensions for 1 to 47 coded targets (assigning correct class with 1 and all other classes with 0 in the vector) as per method `to_one_of_k`.

2. Implementing convolutional networks

For the implementation of convolutional networks we built methods for two layer classes, `ConvolutionalLayer` for performing the convolution operation and `MaxPooling2DLayer` for performing the max pooling operation. Both classes required the completion of methods, `fprop` for forward propagation and `bprop` for back propagation while `ConvolutionalLayer` class needed

also the completion of the method `grads_wrt_params` for the calculation of the gradients wrt weights and biases.

2.1. Convolutional Networks

Convolutional neural networks are a kind of neural network that implements the mathematical operation of *convolution* instead of fully connected matrix multiplication. [Goodfellow et al.]. The convolution operation is performed by sliding a convolution kernel or weight layer over the input layer. This convolution kernel is usually smaller than the input layer and for each placement on the input layer the sum of the multiplication of each input element with its corresponding convolution kernel element is calculated. This represents a single output element in the output layer and its position on that layer corresponds to position of the current window on the input. Convolution is presented in figure 1 from Goodfellow et al.. The same operation is applied for forward and backward propagation as well as the calculation of the gradients wrt parameters by using the corresponding layers for each operation.

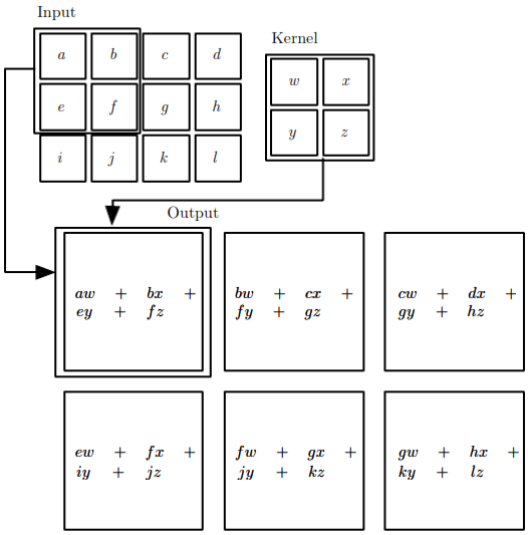

Figure 1. An example of 2D convolution. Figure copied from Goodfellow et al.

2.2. Max Pooling

Pooling is performed on a layer to reduce its size based on a specific function that could be either max pooling, *Lp* pooling, average pooling or sum pooling. The essential purpose of pooling is to reduce the information of the layer with the purpose of increasing its context and as a result reduce computation time and memory usage. Max pooling is applied by finding the maximum elements, firstly by creating a mask of a specific size and assigning each position on that mask with a 0 if the corresponding value of the projection of the mask on the input layer is not the max and 1 if it is and then by sliding the mask on the input layer and outputting those elements to form a pooled layer. This pooled layers contains the maximum elements of each sub region of the input layer. The operation with stride equal to 2 is presented in figure 2 from (Li et al.).

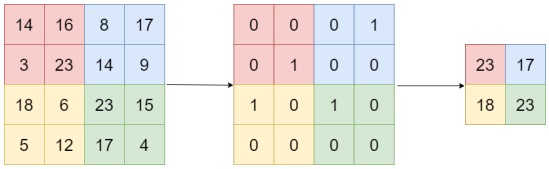

Figure 2. An example of max pooling. Figure generated with reference to Stanford University's course CS231n: Convolutional Neural Networks for Visual Recognition

2.3. Class ConvolutionLayer

Method fprop:

Initially, the batch size is extracted from the shape of the input layers. Then the output height and width are calculated with padding set to 0 and stride to 1. A check is performed to see if the size of the outputs is valid. A four-dimensional `numpy` array for outputs is created with sizes as follows: (batch size, number of output channels, output height, output width).

For the execution of forward propagation, three for loops are used as follows: on batches-> on output channels -> on input channels. The calculation equation for each output layer uses the `convolve2d` method to convolute the input layer of the current batch and input channel and the weight kernel of the current output and input channel. Then these convolutions are added and a final addition of the bias layer is performed to produce a single output layer. The process is repeated for all weight kernels and then for all batches of inputs.

Mean execution time and variance for 3 runs of 1000 loops is $134\mu s \pm 16.2\mu s$ per loop.

Method bprop:

Initially, the batch size is extracted from the shape of the input layers. A four-dimensional `numpy` array for the gradients with respect to (wrt) inputs is created with the same size as the input layers. The weight kernels have to be flipped as the convolution method will perform a flip on them and we want them to be in their original state.

For the execution of backward propagation, three for loops are used as follows: on batches -> on input channels -> on output channels. The calculation equation for each gradients wrt inputs layer uses the `convolve2d` method to convolute the gradients wrt outputs layer of the current batch and output channel and the flipped weight kernel of the current input and output channel. Then these convolutions are added to produce a single layer for the gradients wrt inputs. The process is repeated for all weight kernel layers of each weight kernel to produce the gradients for all the input channels and then for each batch.

Mean execution time and variance for 3 runs of 1000 loops is $111\mu s \pm 9.29\mu s$ per loop.

Method grads_wrt_params:

Initially, the batch size is extracted from the shape of the input layers. A four-dimensional `numpy` array for the gradients wrt kernels is created with sizes as follows: (batch size, number of input channels, kernel height, kernel width). Also, a one-dimensional numpy array is created for the gradients wrt biases with size equivalent to the output channels. As was the case in backward propagation, the inputs are flipped for the convolution method to produce correct results.

For the calculation of the gradients wrt parameters, three for loops are used as follows: on output channels -> on input channels -> on batches. The calculation equations for the gradients wrt parameters uses the `convolve2d` method to convolute the gradients wrt outputs layer of the current batch and output channel and the flipped input layer of the current batch and input channel. The convolutions of these input channels are added to produce a single kenrel layer and this process happens for every input channel and every output channel to produce the final gradients wrt kernels. Finally, the gradients wrt biases are simply the sum of the gradients wrt outputs for each output channel of every batch.

Mean execution time and variance for 3 runs of 1000 loops is $113\mu s \pm 278ns$ per loop.

2.4. Class MaxPooling2DLayer

Method fprop:

Initially, the batch size and number of input channels are extracted from the shape of the input layers. A check is performed to verify that the pooling stride and size can fit on the input data. Then the pooled input height and width are calculated with padding set to 0. A four-dimensional `numpy` array for the pooled inputs is created with sizes as follows: (batch size, number of input channels, pooled input height, pooled input width).

For the execution of forward propagation for pooling, four for loops are used as follows: on batches -> on input channels -> on input height -> on input width. Each channel of the input layers is scanned along its width and height and at each iteration the max value of each subsample of the input layer is stored in the pooled input layers.

Mean execution time and variance for 3 runs of 1000 loops is $493\mu s \pm 4.36\mu s$ per loop.

Method bprop:

Initially, the batch size and number of input channels are extracted from the shape of the input layers. Then the number of output channels, output height and output width are extracted from the output layers. Two four-dimensional `numpy` arrays for the input masks and the gradients wrt inputs are created with the same size as the input layers.

For the calculation of the input masks two for loops and two while loops are used as follows: on batches -> on input channels -> on the input height -> on the input width. All values of each subsample are compared to the max value to create each window mask.

For the calculation of the gradients wrt inputs four for loops are used as follows: on batches -> on output channels -> on output height -> on output width. At each iteration the gradients wrt outputs of the current subsample are multiplied by the corresponding window of the input mask to produce a gradient wrt inputs window that contains only zeros except for the position where the maximum input value was and where now the current gradient wrt outputs is. All these windows form the final gradient wrt outputs layers.

Mean execution time and variance for 3 runs of 1000 loops is $1.43ms \pm 8.9\mu s$ per loop.

3. Context in convolutional networks

The main aspect of the following experiments is the different subsampling methodologies applied to the convolutional networks in order to allow the model to get broader information. These are the following:

- Pooling layers
- Dilation
- Striding

3.1. Pooling layers

Pooling layers and more specifically max pooling were described in the previous section as part of the implementation for the first part of the coursework. While there are different pooling methods, in the following experiments we will compare max pooling to average pooling. While max pooling gets the max element of the subsection as the output, average pooling finds the average of all elements in the subsection.

3.2. Striding

Striding is the amount of steps the kernel makes between each scan of the image and it can easily be understood through figure 3 from (Deshpande).

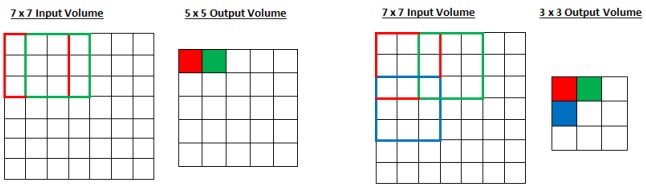

Figure 3. Striding of 1 *(left)*. Striding of 2 *(right)*. Source: (Deshpande)

3.3. Dilation

Dilation allows for increased context from each kernel without the need for dimensionality reduction. This happens

by increasing the size of the kernels by adding zero elements between them in order to increase their distance. The amount by which the distance is increased is the amount of dilation that has been applied on the kernel. Dilation is illustrated in figure 4 from (Antoniou et al., 2018).

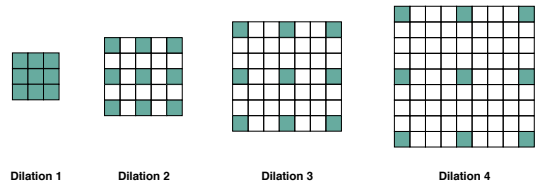

Figure 4. Different dilation distances from Antoniou et al.

3.4. Research questions

Two research questions were addressed in the following experiments involving the performance comparison between the subsampling methodologies described above. More specifically we compared max pooling to average pooling and dilation to striding by running convolutional networks using each method. The difference between each model in the comparisons was only the subsampling methodology used in order to allow for more accurate examination of the differences in performance.

4. Experiments

The following experiments were carried out using the provided Pytorch experiment MLP framework and they were executed remotely using the Google Compute Engine. Different convolution networks were trained and evaluated using the provided script `train_evaluate_emnist_classification_system.py`.

The baseline of the experiments across all four subsampling methodologies were: batch size 100, 64 filters, 4 layers and the given seed of 7112018. Some models were trained with seed set to 0 and 100 but the differences were insignificant and so those experiments were not furthers explored in this report. Epochs were set to 50 for all experiments. After running the baseline model for 100 epochs, no noticeable difference was noted between epoch 50 and 100.

The experiments that followed were designed in order to have only one varying hyperparameter from the baseline in order to allow for better recording of the changing effects. Two hyperparameters were chosen for experimentation. These were the number of layers and the number of filters. The number of layers defines how many times the convolution operation with the subsampling methodology will be performed to generate the final output while the number of filters is the amount of different kernels that are to be applied on the input layers. Each filter generates one output layer. The differences between the subsampling methodologies for each hyperparameter change were the core of the experiments that follow. For each question the reduction of the number of layers and filters was addressed. Specifically, models with 1 and 2 layers were trained as well as models with 16 and 32 filters apart from the baseline.

There has been research on the amount of hidden layers that are needed for optimal approximation but their results have a more theoretical than practical utility [Hornik et al.] and therefore there is no specific number of layers defined as the optimal for training convolutional neural networks on images. LeCun et al. used three hidden layers for their network architecture to perform feature extraction from handwritten zip codes to find that their network performed better than their previous system [Denker et al.]. As a result, it was decided that it was unnecessary to test more models with more than 4 layers. Instead, the number of layers for the follow up experiments was reduced first to 2 and then to 1 with the intention to find the fewest amount of layers needed for a good approximation.

The number of filters was the second hyperparameter that varied in the following experiments. The filters represent the kernels that are applied on the input layers through the convolution operation in order to produce an output layer that essentially is a transformation of the original image. Their main purpose is feature extraction and the number of them defines the amount of features that are going to be extracted at the end of the network. The choice of this number depends on the domain of the experiments and the type of the input images. In our case, where our images are digits and letters in grayscale the amount of varied features that can be extracted are limited so amounts larger than the baseline of 64 filters are not tested. On the other hand, amounts of 32 and 16 filters were tested with the intention to check the trade off between accuracy and computational efficiency.

4.1. Max vs Average Pooling

The first question addressed the performance differences between max and average pooling for different number of layers or filters.

The accuracies over the 4, 2 and 1 layer models trained are presented in figure 5 while the testing accuracies of the best models for each number of layers are presented in table 1. These results are inline with the theory regarding the amount of layers present in the network. It is evident that with the 4 layers of the base line model the testing accuracy is already very high at *0.878861*. When the layers are reduced to 2, the model is not significantly hindered as its test accuracy is reduced to *0.870886*. While there is some accuracy reduction this is offset by the increased computational efficiency as timings decreased by more than 3 minutes for both max and average pooling. The timings for the models with different number of layers are presented in table 2. The accuracy of the final model with 1 layer drops significantly from the other two models for both methods and while it is the most computationally efficient that does not constitute a worthy trade off.

The accuracies for models with 64, 32 and 11 filters trained for 50 epochs are presented in figure 6 and their testing ac-

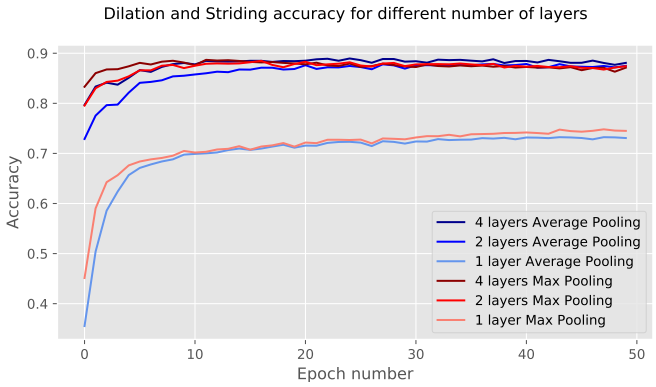

Figure 5. Accuracy for different number of layers on convolutional networks with max or average pooling.

Method	4 (base)	2	1
Max Pooling	0.878861	0.870886	0.737468
Average Pooling	0.87962	0.871266	0.725

Table 1. Testing accuracy for different number of layers.

Method	4 (base)	2	1
Max Pooling	14m23.510s	11m17.273s	5m20.456s
Average Pooling	14m55.701s	11m38.964s	5m41.626

Table 2. Timings for different number of layers.

curacies are in table 3. The accuracy is positively correlat with the amount of filters but unlike the number of laye decreasing the filters does not have as much as a negati impact on accuracy for both methods. In this case, the tra off between computational efficiency and accuracy is n debatable as the improvements in them can have an imp for even larger datasets. These timings are presented table 4.

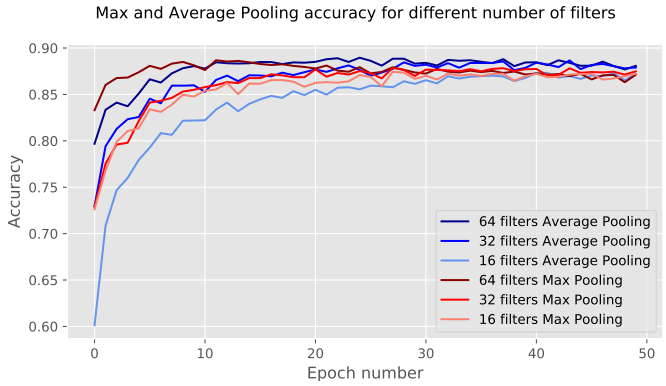

Figure 6. Accuracy for different number of filters on convolutional networks with max or average pooling.

Method	64 (base)	32	16
Max Pooling	0.878861	0.870823	0.867911
Average Pooling	0.87962	0.877975	0.863861

Table 3. Testing accuracy for different number of filters.

Method	64 (base)	32	16
Max Pooling	14m23.510s	7m25.024s	5m18.456s
Average Pooling	14m55.701s	7m33.666s	5m18.456s

Table 4. Timings for different number of filters.

4.2. Dilation vs Striding

The second question addressed had the same specifications of the first but this time dilation and striding were compared.

Unlike the first question models with 1 layer are performing almost as well as models with 2 and 4 layers for both dilation and striding. By checking figure 7 it is evident that during the first 10 epochs the 1 layer models underperform but by the end of 50 epochs they catch up to the other models. There testing accuracies (table 5) indicate so. Another thing to notice is that the 2 layer models perform slightly better than the baseline models with 4 layers under both dilation and striding.

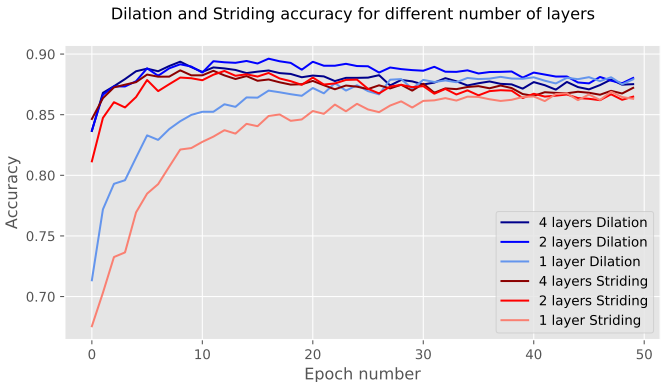

Figure 7. Accuracy for different number of layers on convolutional networks with dilatrion or striding.

Method	4 (base)	2	1
Dilation	0.881519	0.886266	0.873987
Striding	0.877152	0.878544	0.862025

Table 5. Testing accuracy for different number of layers.

The timings recorded (table 6) show the same pattern as the previous question, where fewer layers mean more computational efficiency. What should be noted though from the timings table is that dilation is taking significantly more time than all other methods due to the fact that it is not a dimensionality reduction methodology and thus the layer dimensions are not reduced over each layer.

Method	4 (base)	2	1
Dilation	53m48.102s	35m16.699	18m15.831s
Striding	17m4.367s	12m47.403s	8m38.888s

Table 6. Timings for different number of layers.

The accuracies over the 64, 32 and 16 filter models trained are presented in figure 8. The experiments on different number of filters have provided little insight on how they affect modelling under each method as their testing accuracies (table 7) are very similar. The only thing to notice is that dilation performs better than striding for all number of filters. The timings for the two methods are presented in table 8.

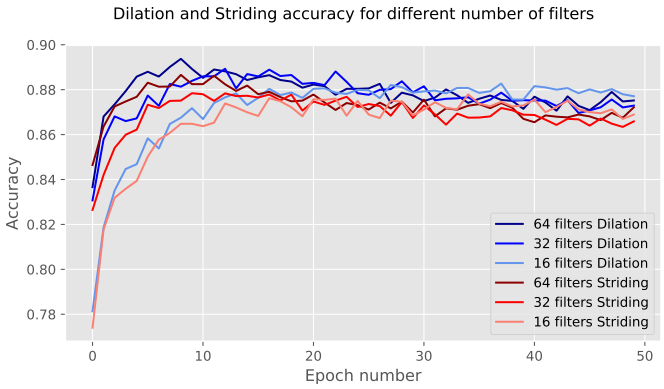

Figure 8. Accuracy for different number of filters on convolutional networks with dilation or striding.

Method	64 (base)	32	16
Dilation	0.881519	0.880506	0.878544
Striding	0.877152	0.865696	0.867848

Table 7. Testing accuracy for different number of filters.

Method	64 (base)	32	16
Dilation	53m48.102s	21m34.653s	14m59.328s
Striding	17m4.367s	9m18.314s	7m11.204s

Table 8. Timings for different number of filters.

5. Discussion

Choosing the number of layers and number of filters was considered a viable choice for the available time frame as these variables are two of the most important considering the modeling of convolutional networks.

Max and average pooling have very similar performances under both varying hyperparameters. To summarize, average pooling performs slightly better than max pooling under both hypeparameters except for their lower values of 1 layer and 16 filters where max pooling comes ahead. Both methods perform better for higher amount of layers and filters and both methods are very similar regarding computational efficiency.

Regarding the second question, dilation shows better performance under all cases of both variables. Unlike max and average pooling, dilation and striding show better performances for models with 2 layers rather than 4 or 1 layer. Dilation shows a similar trend on the number of filters as max and average pooling while striding performance on 16 filters is slightly better than on 32 filters.

After getting the above results the only sure way to move forward is to check how other hyperparameters affect the models. Under this framework, these could be the batch size and the weight decay coefficient. These experiments could also involve more epochs and also different kernel initializations.

6. Conclusions

The amount of variables that have an effect on model performance is large and research until now does not provide any optimal choices for methods or parameters. In general the search for optimal performance is highly dependable on the domain of the data. After getting these results from our experiments we can only assume that the search for the best performing model can only expand and it should always be adapted to the domain.

References

Antoniou, A., Słowik, A., Crowley, E. J., and Storkey, A. Dilated DenseNets for Relational Reasoning. *ArXiv e-prints*, November 2018.

Cohen, G., Afshar, S., Tapson, J., and van Schaik, A. Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017. URL http://arxiv.org/abs/1702.05373.

Denker, John S, Gardner, WR, Graf, Hans Peter, Henderson, Donnie, Howard, Richard E, Hubbard, W, Jackel, Lawrence D, Baird, Henry S, and Guyon, Isabelle. Neural network recognizer for hand-written zip code digits. In *Advances in neural information processing systems*, pp. 323–331, 1989.

Deshpande, A. A beginner's guide to understanding convolutional neural networks part 2. URL https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hornik, Kurt, Stinchcombe, Maxwell B., and White, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541. URL https://doi.org/10.1162/neco.1989.1.4.541.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

Li, F., Johnson, J., and Yeung, S. Cs231n convolution neural networks for visual recognition, university of stanford. URL http://cs231n.github.io/convolutional-networks/#pool.