A Appendix for Torres and Cantú, "Learning to See: Convolutional Neural Networks for the Analysis of Social Science Data"

Michelle Torres and Francisco Cantú

This Appendix provides additional analyses, diagnoses, and details regarding the description, application and plots presented in the main text. It is organized into 5 sections.

- **Glossary**: pp. 2-4
  - This section presents a glossary with terms regarding the elements, process, and implementation of CNNs.
- **Image Pre-processing** (A.1): p. 5
  - This section provides technical details, and extra information and plots regarding the data pre-processing steps such as zero-padding.
- **Feature extraction** (A.2): pp. 6-7
  - This section provides technical details and extended information regarding feature extraction, dimension reduction, and the different type of layers of a CNN.
- **Learning** (A.3): pp. 8-9
  - This section provides technical details and extra information regarding the training and learning process, such as an extended discussion of backpropagation, and alternative optimizing functions.
- **Application** (A.4): pp. 10-15
  - This section provides extra plots and analysis of the classification task presented as the illustrating example of the article: the coding of handwritten numbers in electoral tallies.

Glossary

activation function Function that allows to generate non-linear outputs. In the context of CNNs, these are mathematical rules or functions that transform the elements of a matrix. 8

backpropagation Long series of nested equations that have the objective of adjusting each weight in the network in proportion to how much it contributes to overall error. Backpropagation can be seen as an application the Chain rule to find the derivatives of a function with respect to any variable in the nested equation. 10

batch normalization Technique for improving the performance and stability of a neural network via a normalization step that fixes the means and variances of layer inputs. The normalization process occurs in "mini-batches" (e.g. subsets of the training dataset), to make the process more efficient. This is possible given that 1) the optimized loss over a mini-batch is an actual estimate of that in the full set whose quality improves as the size of the batch increases, and 2) takes advantage of parallel computation. For a layer with d-dimensional input $\mathbf{x} = (x^{(1)} \dots x^{(d)})$, we normalize each dimension with $\hat{x}^{(k)} = \frac{x^{(k)} - \mathbf{E}(x^{(k)})}{\sqrt{\text{Var}(x^{(k)})}}$, where the expectation and variance are computed over the training dataset. 17

batch size Number of images in a match, or subset of training images. 11

epochs A training iteration consisting on the single pass of the entire training database throughout the model. 11

feature map The matrix mapping the outputs from convolution of a given filter and the different regions of an image. 8

filter size Product of height and width, in pixels, of a matrix representing a filter. 7

filter stride Number of pixels that a filter slides through an image. 7

filters In a CNN, the filters represent the neurons of the network. These are matrixes of numbers representing patterns and combinations of pixels that permit the extraction of features of an image. The pixel combinations can represent edges, corners, blobs, color combinations, and textures. Filters are convolved with regions of the image to create feature maps that represent the prevalence of the patterns they represent in an image. 7

forward propagation The process in which the input data moves in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer . 10

generalization The ability of a model to perform well on previously unobserved inputs (Goodfellow, Bengio and Courville, 2016, p. 110). . 15

gradient descent Optimization algorithm used to to update the weights, or coefficients, of our model. Its objective is to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. The gradient is built with the partial derivatives of the function with respect to its different parameters. It is represented by $x^{'} = x - \epsilon \Delta x f(x)$. 10

hyperparameters Hyperparameters are the variables exogenous to the model that determine the network structure and how it will be trained. The values of the hyperparameters are set before the training begins and do not depend on the data. 5

L1 and L2 regularization L1 and L2 norms are regularization methods that add to the loss function an additional penalty term for the magnitude of the weights. The difference between these two norms lies on how they specify the penalty. In the case of L1, it adds the absolute value of the coefficient magnitude as a penalty term. Formally, the loss function in a L1 regression is specified by $L(x, y) = \sum_{i=1}^{n}(y_I - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$, where β is the size of the coefficient j, and λ is the penalty term. In contrast, l2 specifies the penalty as the sum of the squared magnitude of the weights. Formally, the cost function in the l2 norm is specified by: $L(x, y) = \sum_{i=1}^{n}(y_I - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta^2$. 16

layer depth Number of filters used in a layer. 7

learning rate A positive scalar that defines the magnitude of the steps in which the gradient descends. Formally, the learning rate is defined as the parameter ϵ in the gradient descent function (see gradient descent). 11

loss function Function that quantifies the differences between the predicted labels from the CNN, and the true labels of the input data. The loss over a dataset is the sum of the loss of its units. Formally, $L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$, where $f(x_i, W)$ is the a function of the weights of the filters in the CNN layers, and the pixel matrix representing the input image. 10

mini-batches Subsets of the images in the training dataset used in the batch-normalization process. 11

receptive field The area where a given filter, or neuron, is positioned to execute a convolution. 7

ReLU The name stands for REctified Linear Unit. It is the most commonly used activation function in CNNs formally defined as $y = max(0, x)$. It is computationally cheap due to its mathematical simplicity, converges faster due to the linearity for positive values and its sparsely activated given that it is zero for negative values. 9

Sigmoid Activation function defining a "S"-shaped curve, or sigmoid, formally defined as the inverse logit: $\frac{1}{1+e^{-x}}$. Useful when dealing with binary outcomes/labels. The function is differentiable and monotonic, and can cause a network to get stuck when training. 8

softmax layer A layer with a multinomial function embedded that transforms the output of the CNN layers up to that into probabilities that the input belongs to each of the potential labels. This is a fully-connected layer because its neurons are not independent and the output is based on this dependency (i.e. the probabilities summing to 1). 10

Tahn Also known as hyperbolic tangent, it is an activation function also with a sigmoidal shape but with a range between -1 and 1. Its formal definition is $\frac{22}{1+e^{-x}}$ implying that negative inputs are mapped strongly negative, and zero values would be near to that value when mapped. 8

transfer learning Exploiting a model trained in a particular setting to improve the generalization of the findings of a different setting. This is a valuable resource when the researcher considers that the factors that explain the variations of the original database are useful for the goal of the new database (Goodfellow, Bengio and Courville, 2016, p. 526-527). 18

weight The unknown parameter of the neural network that seek to improve the fit between the model and the data. 9

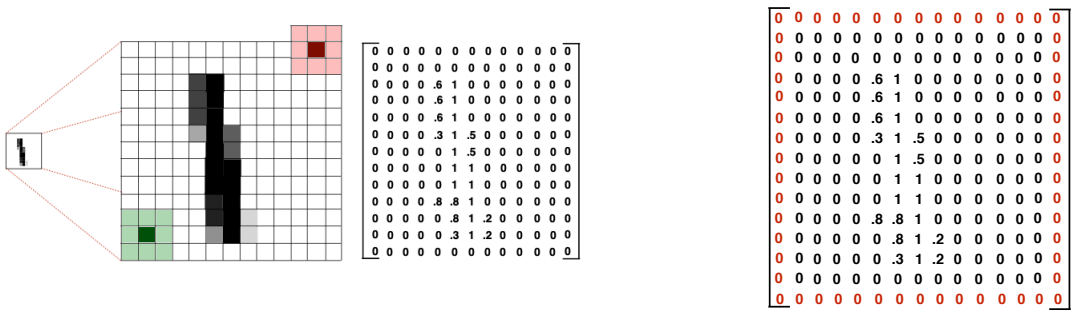
zero-padding A padding is a "frame" that we add to the border of an image to allow the convolution of the edges and corners of an image, and increase the information that is processed through the CNN. In this case, the zero-padding adds a vector of zeros with the length of the width of the image above and below it, and another vector of zeros with the length of the height of the image to the left and right of it. This is equivalent to adding a black frame of width 1 px to the image. 6

A.1 Image Pre-processing

A.1.1 Zero-padding

As we reviewed in the text, the convolution consists on overlaying a small filter, or kernel, with different areas of the images under analysis by placing the center of the filter on top of a pixel. However, this operation does not become possible for those pixels at the edge of the image under analysis. Consider Figure A.1(a) and the pixel highlighted in dark green. A filter of size 3×3 centered on it can perform the convolution seamlessly given that all the values surrounding it exist. However, the convolution is not possible when the filter is placed on the dark red pixel given that the kernel (in pink) exceeds the dimension of the image. Thus, we can "pad" the missing values beyond the edges with zeros as illustrated in Figure A.1(b) which shows a zero padding of $p = 1$, and thus increasing the size of the numerical array from 13×13 to 15×15. With this action, we also guarantee that the size of the output image will remain the same as the input. However, if the objective is to use a convolution layer also as a way of reducing the dimensionality of the image, then padding is not necessary.

Figure A.1: Image Pre-processing of a handwritten "1"



(a) Image Transformation

(b) Zero Padding

There exist other types of padding including the one with replicates of the pixels at the edges (instead of using "0"), or the "wrap around" which consists on examining the opposite side of the image to "pad" the edge. The latter is preferred in cases where aesthetics is a concern, while the former is best for overall efficiency. The size of the padding (e.g. the number of rows that you add to edge) depends on the size of the filter.

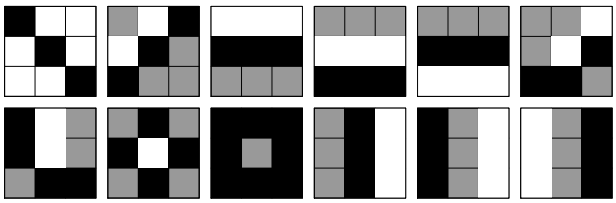
A.2 Feature extraction

A.2.1 Filters

The filters that are at the core of a convolution are simply matrices or *kernels* that we slide through an image of interest with the objective of identifying whether the feature that each filter represents is found in different parts of the image. The convolution achieves that by creating a map indicating how much a given area resembles the content of the filter. The features conveyed by the filters can be lines, corners, shapes or combinations of all of these (Figure A.2 shows a few examples).

For example, the features that distinguish handwritten digits are mainly lines with different orientations, curvatures and magnitudes: a vertical line is a crucial feature for the initial discrimination of a "1" versus a "0." However, the number and complexity of the features to distinguish a man from a woman will be much larger.

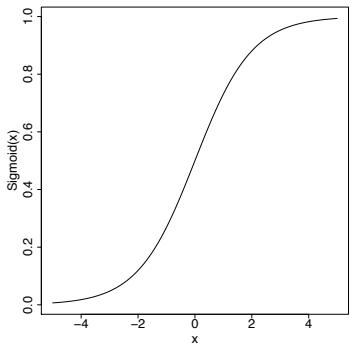
Figure A.2: Examples of filters



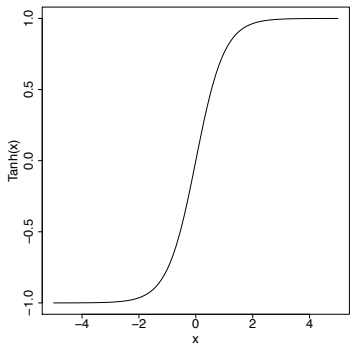
Note: This figure displays examples of filters of size 3 (height)$\times$ 3(width) = 9 that were randomly initialized in the first layer of a CNN.

It is important to remember that these filters and their features are learned throughout the backpropagation process described in the main text. However, as in any optimization process, the CNN needs a starting point. For the CNN used in this article, the filters in the first layer are initialized randomly using the Glorot uniform method. Also known as Xavier, this initializer draws samples from a uniform distribution: $W \sim \mathcal{U}\left(\frac{-6}{u_{in}+u_{out}}, \frac{6}{u_{in}+u_{out}}\right)$, where u_{in} is the number of input units in the weight tensor, and u_{out} is the number of output units in the weight tensor. In most canned architectures, it is not necessary to define the initialization of these filters since they contain default starting points.

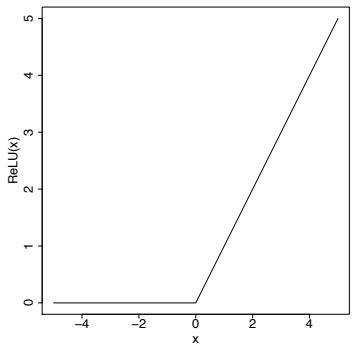
Figure A.3: Examples of Activation Functions



(a) $Sigmoid(x) = \frac{1}{1+e^{-x}}$



(b) $Tanh(x) = \frac{2}{1+e^{-2x}}$



(c)$ReLU(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{otherwise.} \end{cases}$

A.3 Learning

A.3.1 Backpropagation

Suppose that a neuron j in the last layer provides a classification outcome y_j.[1] To estimate the prediction error, the model compares such an outcome with the target label, t_j. In our digit recognition example, the prediction error of the neuron for the outcome "1" is the difference between the true outcome and the model's estimated probability for the image to belong to that category. After adding up the prediction error of all the neurons in the layer, $E = \frac{1}{2}\sum_{j\in 10}(t_j - y_j)^2$, we can estimate the error function derivative of the last layer:

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j) \tag{2}$$

Similarly, we can express the error derivatives in terms of the logit of the neuron, z_j:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j} \tag{3}$$

To minimize this error term, the network goes back to its prior layers and identifies those weights contributing the most to this error. In other words, it estimates how the neuron outcomes in layer i affect the outputs of layer j given the weighted connection between both layers, w_{ij} :

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial y_i} = \sum_j w_{ij}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}y_j(1-y_j)\frac{\partial E}{\partial y_j} \tag{4}$$

These partial derivatives allow us to estimate the contribution of a specific weight to the error term:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_j(1-y_j)\frac{\partial E}{\partial y_i} \tag{5}$$

The partial derivative in Equation 5 allows the model to gradually modify its weights after reviewing a set k of examples from the database K:

$$-\Delta w_{ij} = -\sum_{k\in K} y_i^{(k)} y_j^{(k)}(1-y_j^{(k)})\frac{\partial E^{(k)}}{\partial y_i^{(k)}} \tag{6}$$

[1] The explanation and notation of this example come from Buduma and Locascio (2017).

A.3.2 Transfer Learning

Table A.1: Popular CNN architectures that serve as base for transfer learning

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Number Params.	Depth	Trained on	Goal	Authors	Novelty
VGG16	528 MB	0.713	0.901	138357544	19	ImageNet: 1.2 million training images, 50K validation, and 150K testing.	Image classification into 22000 classes	Simonyan and Zisserman (2014)	Deeper network with smaller filters
GoogleLeNet	44 MB	0.64	0.933	6797700	22			Szegedy et al. (2015)	Building networks using dense modules/blocks. Instead of stacking convolutional layers, we stack modules or blocks, within which are convolutional layers.
Inception V3	92 MB	0.779	0.937	23851784	159			Szegedy et al. (2016)	Evolution of Inception V1 with added parameters
ResNet50	98 MB	0.749	0.921	25636712	-			He et al. (2016)	Popularised skip connections. Among the first to use batch normalisation.
Xception	88 MB	0.79	0.945	22910480	126			Chollet (2017)	Introduced CNN based entirely on depthwise separable convolution layers.
InceptionResNetV2	215 MB	0.803	0.953	55873736	572			Szegedy et al. (2017)	Converting Inception modules to Residual Inception blocks.
ResNetXt50	96 MB	0.777	0.938	25097128	-			Xie et al. (2017)	Scaling up the number of parallel towers ("cardinality") within a module
AlexNet	223 MB	0.625	0.830	60000000	8			Krizhevsky, Sutskever and Hinton (2012)	First to implement ReLu as ACT
LeNet		0.993	-	60000	5	MNIST: 60K training, 10K validation	OCR and character recognition	LeCun et al. (1998)	Iconic CNN with `CONV` and `POOL` stacking, followed by `FC`

Note: **Top-1 Accuracy** indicates the proportion of images correctly classified as the top-1 label manually attached to the image. **Top-5 Accuracy** indicates the proportion of images in the testing set correctly classified as one of the top-5 labels manually attached to the each image. **Depth** corresponds to the number of layers in the network. The **ImageNet** are images of several random objects and scenes taken from the Internet and manually labeled in Amazon MTurk. The **MNIST** dataset contains images of handwritten numbers.

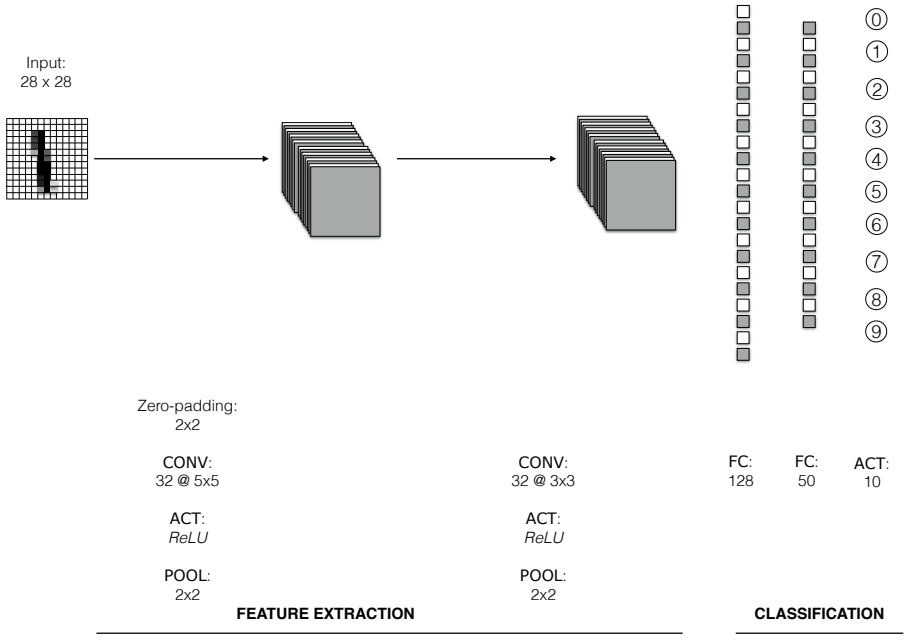
A.4 Application 1: Coding electoral results from tallies

A.4.1 Extracting digits from tallies

We decided to develop a function that identifies the coordinates of three focal points of the tally: the yellow banner at the top of the page, the bright pink rectangle at the bottom left of the tally, and the pink circle below the table. The coordinates of these elements, shown inside red rectangles in the first element of Figure 3, allow us to identify the bottom, top and left lines of the table containing the digits. The green dashed lines and yellow area in the second element of the diagram illustrates this process. Once we isolate the table, we divide it into 3 $\times$ *the number of parties/candidates in the district* cells. We then cut and save each cell under the assumption that it contains a digit.[2]

A.4.2 Network architectures for digit detection

Figure A.4: Network Architecture: base model



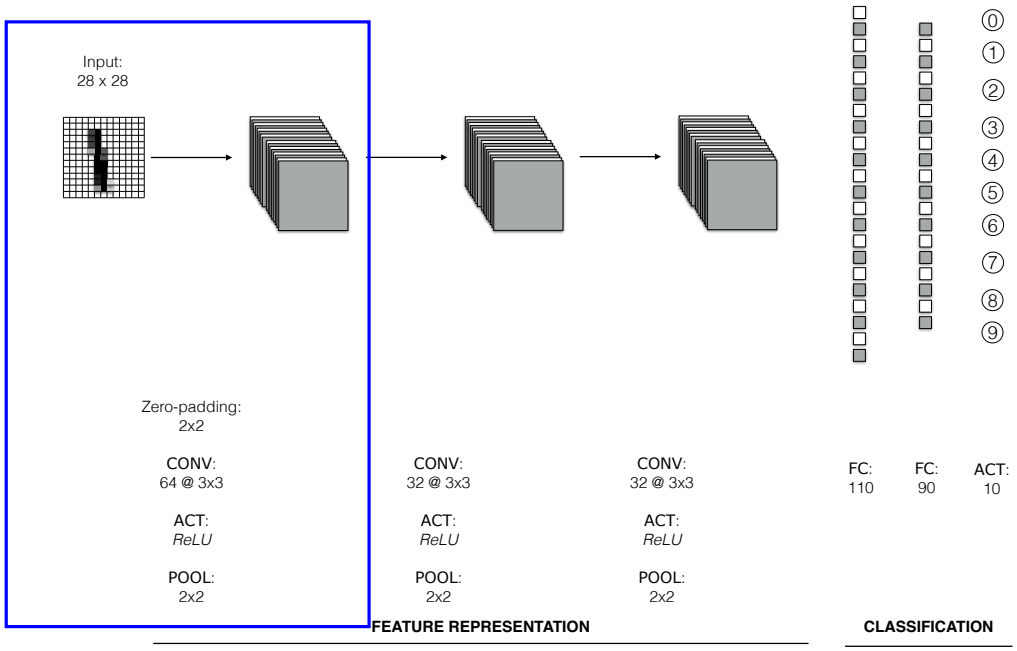
Notes: Figure A.4 illustrates the CNN structure applied to identify digit numbers. This is the base model whose inputs consis of numerical arrays of 28 (height) $\times$ 28 (width) pixel values. The network was trained in 60,000 digits from the MNIST data. Batch size is 200 and number of epochs is 15.

A.4.3 Hyperparameter grid search

To find the most "optimal" CNN, we ran different sets of model specifications. The training sample is composed of 60,000 digits MNIST, while our validation sample included 2,000 digits from

[2]This, however is not fulfilled in some cases. Although polling staff is supposed to fill all cells and use leading zeros for 1 and 2-digit numbers, or parties with no support, several ballots have empty cells.

Figure A.5: Network Architecture: revised model



Notes: Figure A.5 illustrates the CNN structure applied to identify digit numbers after conducting a hyperparameter search grid. The network was trained with 60,000 digits from MNIST and then tested on 2,000 digits from the validation set of the tallies. We froze the first convolutional layer (blue rectangle) of the architecture, and retrained the rest. Batch size is 200 and number of epochs is 20.

our tallies. We use the *Keras Tuner* (`keras-tuner` library in Python) to conduct the 180 trials. The space of potential values for the hyperparameters are the following:

Table A.2: Hyperparameter grid search set-up

Hyperparameter	Default	Min. value	Max. value	Interval	Selected
Number `CONV` blocks	2	1	3	1	3
Filters Layer 1	-	16	128	16	64
Filters Layer 2	-	16	128	16	96
Filters Layer 3	-	16	128	16	112
Dropout	0.5	0.1	0.5	0.1	0.1
Size `FC 1`	128	50	200	20	150
Size `FC 2`	50	10	100	10	70
Learning rate	0.0001	0.0001	0.01	-	≈0.0001
Epochs	-	-	45	-	20

Table A.3 below shows the results for a selected number of those trials:

A.4.4 Loss and accuracy history

The following plots show the loss and accuracy history of the *revised* model with transfer learning throughout the training process. The red line shows the loss/accuracy across epochs for the

Table A.3: Results from the hyperparameter search grid (selected)

Number CONV layers	Filters Layer 1	Filters Layer 2	Filters Layer 3	Dropout rate	Size FC 1	Size FC 2	Number Epochs	Validity Accuracy
3	64	32	32	0.200	110	20	45	0.332
3	96	32	112	0.300	70	50	5	0.335
3	112	80	80	0.200	70	50	45	0.338
3	80	64	96	0.200	50	20	5	0.340
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	96	112	48	0.300	130	50	45	0.352
3	80	112	112	0.500	170	30	15	0.350
3	32	32	112	0.200	110	30	5	0.346
3	96	32	112	0.300	70	50	15	0.343
3	32	32	112	0.200	110	30	15	0.342

Note: This is a selection of some models tested using the hyperparameter search grid. The specification of these models can be found in Table_A3.txt in the *CodeOcean* capsule. However, the column of "Validity accuracy" is included exclusively in the output file of such repository.

training data, whereas the blue ones show those indicators for the validation sample. While the training loss accuracy show a trend of classification improvement, the blue line started to appear stable, suggesting that the improvement of the model comes from learning features specific to the training set (overfitting). Thus, we can stop the training process at around 20 epochs.

A.4.5 Classification errors

There are substantial differences between the digits that belong to the MNIST data that are used to train and test most architectures for digit recognition, and digits taken from other contexts like those from the electoral tallies under analysis. The former are clearly defined, without stains, lines or blobs that do not belong to the digit, centered and placed in a black background. In contrast, the digits from the tallies contain certain irregularities such as stains from the scanning process, bounding boxes, and other lines and edges in ink or pencil. Below we present a few of the digits from each dataset to illustrate the differences and provide initial evidence of the potential challenges for classification that using a model trained entirely on the MNIST data pose.

A.4.6 Vote counts per party in District 15: predicted vs. observed

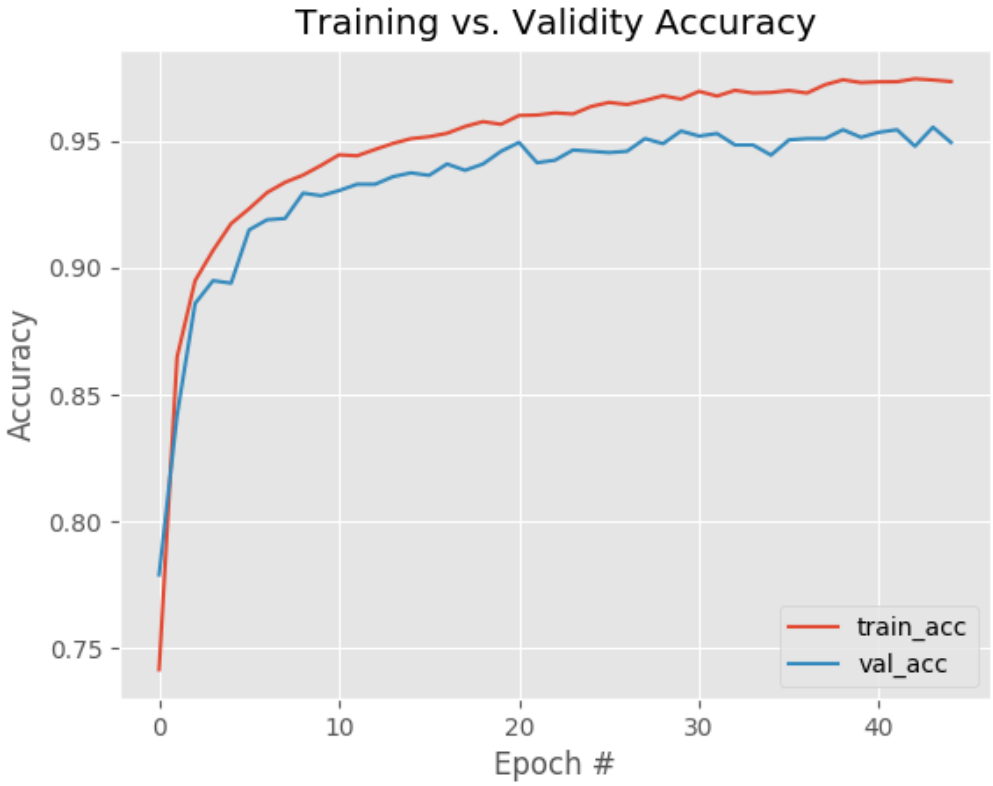
Figure A.9 presents the comparison of detected and real vote counts in the tallies of the district. Because of the right skewed distribution, we applied a logarithmic transformation to both the predicted and real vote counts.

Each point in the plot represents the comparison between the predicted vote counts of each of the parties (including null votes, non-registered candidates, and coalitions) and the actual votes. The size of the point indicates the frequency of each potential combination.

Notice that we also added to the plot information about the quality of the predictions of the digits. Recall that the last layer of the CNN, the *softmax* layer, outputs a list with the probabilities that each input digit has of belonging to each of the 10 possible outcomes (0-9). To classify the number, we take the category with the highest probability of the list. For most of these numbers, the maximum probabilities are pretty high (above 0.99). However, in cases where the number is ambiguous, or the model does not have enough information (e.g. the digits in the tally are not legible), the predictions that the CNN makes are less likely to be accurate. Therefore, we created

Figure A.6: Performance history of revised model for digit classification

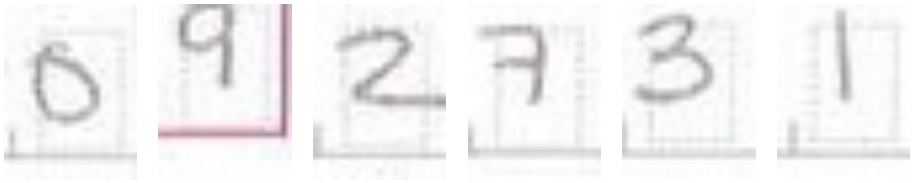
(a) Loss



(b) Accuracy

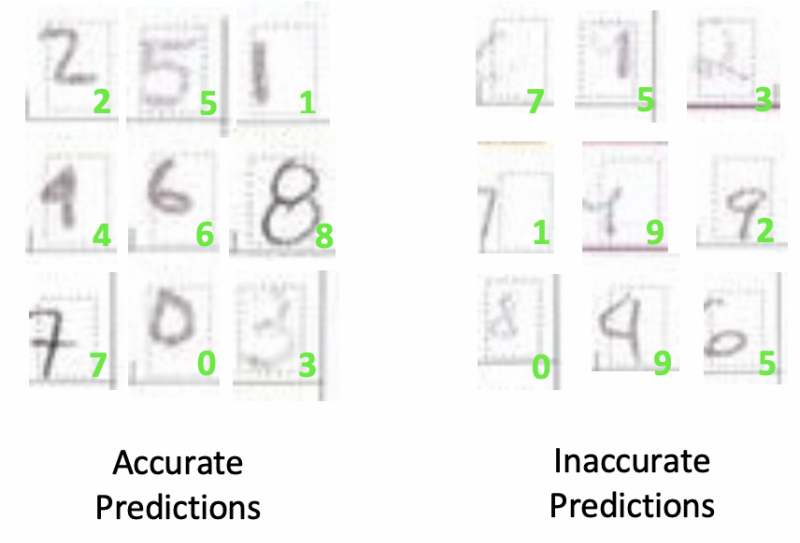
Figure A.7: Digits from MNIST vs. Electoral tallies

(a) *MNIST dataset*



(b) *Tallies*

Figure A.8: Examples of digit predictions



an indicator for each vote count registered in each tally that we then use to evaluate its overall quality. The triangles in Figure A.9 show the vote counts in the tallies identified as "moderate quality", whereas the blue circles show the "high quality" ones. If the CNN is yielding accurate predictions, we should see a high density of observations concentrated along the 45 degree dashed line indicating that the prediction and the official vote counts are equal. We indeed observe a dense distribution of a large number of observations along the red line. This is especially true for the high quality tallies: very few deviate from the line. The "moderate quality" observations show greater deviations, but these do not follow a pattern that would suggest a systematic bias.

Figure A.9: Number of votes registered in tallies: Official vs. Predicted