**Dataset**

# The data will come the RVL-CDIP Dataset (Ryerson Vision Lab Complex Document Information Processing). This dataset consists of 400,000 greyscale images of 16 classes of documents. The images are presplit into 320,000 training images, 40,000 validation images, and 40,000 testing images. All image dimensions are 1000 pixels or less, though this will be scaled down for training efficiency. The data consists of 16 document classes: letter, form, email, handwritten, advertisement, scientific report, scientific publication, specification, file folder, news article, budget, invoice, presentation, questionnaire, resume, and memo.

# With 320,000 training images and 16 classes, the dataset provides roughly 20,000 training images per class, far more than need to effectively train a neural network efficiently. For the sake of efficiency, random class subsets will be taken from the training, validation, and testing datasets to scale down the overall data size while maintaining class proportions.

# Convolutional Neural Networks

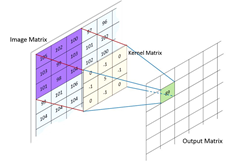
Convolutional neural networks are the primary neural architecture used for image analysis because they work with matrix inputs and have significantly fewer parameters that require training than other network designs. Where MLP would require each image to be flattened to a 1-dimensionl input array with a length of the square of the image dimension, CNNs work with the image in its original matrix form. MLP puts weight and bias parameters on each node (pixel) and connects that node to every other node in the next layer all the way down the network. For a 100 x 100 image, this means that the MLP would have 10,000 input neurons. A network with one hidden layer of the same size as the input and 16 output nodes would require over 1.6 billion weight parameters. The CNN drastically shrinks the number of parameters by running small sets of weight values over the input values in the form of kernels.

**Input Layer**

The input layer is a three-dimensional matrix with number of rows equal to the image height and number of columns equal to image width. For image analysis, the dimensions represent image height, width, and number of color channels, and the values represent pixel values within the range [0, 255]. Grayscale images, as used in this project, have one color channel representing grayscale value.

**Convolutional Layers**

The convolution layer consists of a set of learnable kernels, which are smaller matrices of randomly initialized weight values in a 3-dimensional matrix (number of color channels, height, width). In the feedforward process, the kernels slide (convolve) over each element of the input matrix and compute the sum of the element-wise multiplication between the kernel and the local image pixel values. This matrix multiplication results in a single value that maps to the feature map in the center position of the section of image matrix under the kernel.



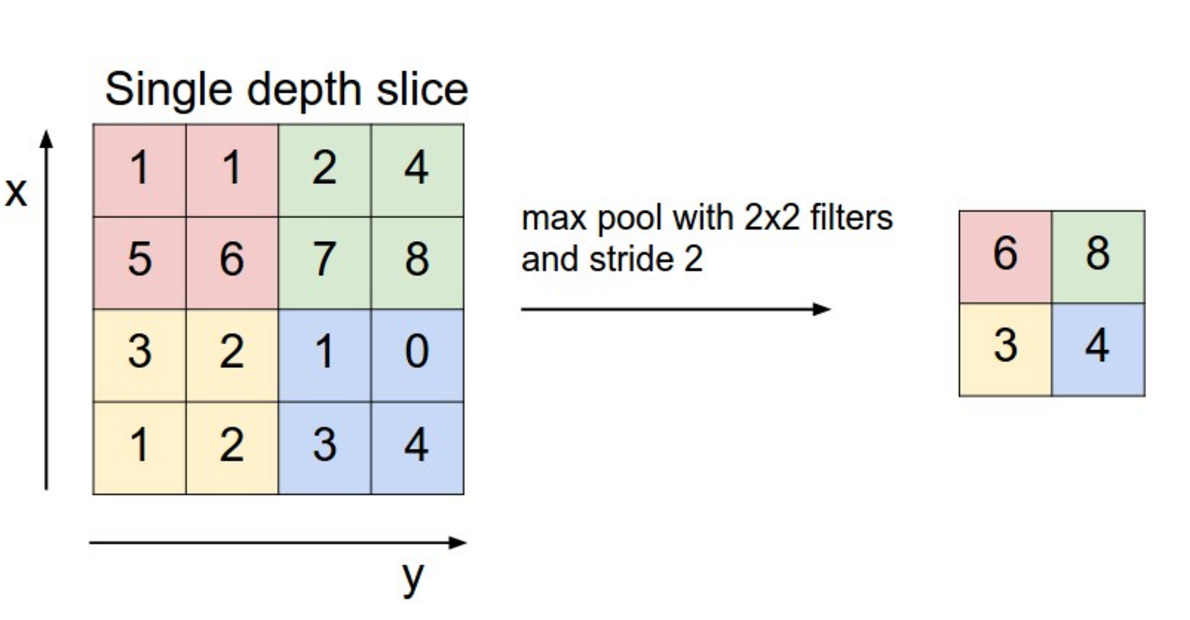
Each feature map is the resulting matrix from the computed values of a single kernel convolving over the whole of the input matrix. Aside from the number of kernels, convolutional layers have several adjustable parameters that control output size. Padding is a parameter that is used to maintain input size. As seen in the above figure, convolution results in a single value at the center of the kernel, which means that the edges of the input are cut offand the output is *n-1* (n representing the height and width of the kernel) dimensionssmaller than the input. To prevent this, layers of zeros or other values are added around outside of the input so that the first position of the kernel is centered on the upper-left pixel value. The number of padding layers is calculated by *(n-1)/2* with *n* representing kernel size. Stride is number of pixels that kernel jumps with each step of convolution. A stride of 1 means that the kernel moves one pixel row or column at a time and results in an output image that is the same size as the input image. Increasing stride to *n* decreases the output size *n-*fold (stride of 2 results in an output ½ the input size).

**Batch Normalization**

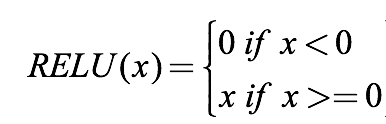
Batch normalization normalizes the output of the convolution by subtracting the batch mean and dividing by the batch standard deviation. Batch normalization lessens the likelihood of the gradients decaying (moving to 0) or exploding (moving toward infinity) as well as the chances of the model becoming dependent of a small number of features with high weight values due to exploding gradients. Higher learning rates can also be used because batch normalization ensures stabilization in activation output.

**ReLU Activation**

Following the batch normalization, a ReLU (Rectified Linear Units) activation layer is applied. The ReLU layer applies the function *f(x) = max(0, x)* to all of the values in the input, where *x* represents the input value. Essentially, the function sets all negative inputs to 0 and sets the output as the input for positive inputs. This function is used to calculate gradients for parameter updating and. ReLU results in a network that is computationally efficient and fast training due to the simplicity of the function derivation and the avoidance of vanishing gradient (gradient values decreasing exponentially) by maintaining a constant derivative.

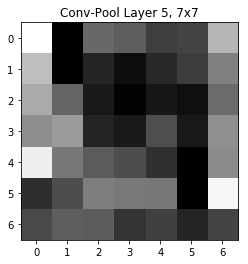
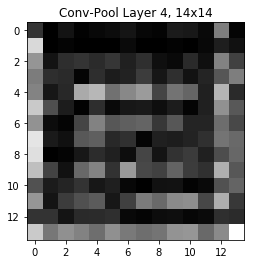
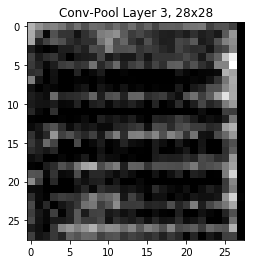
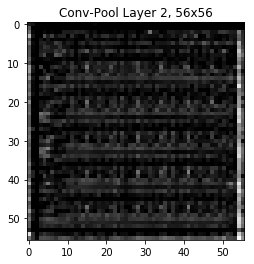
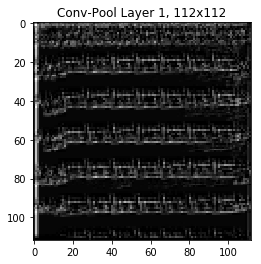
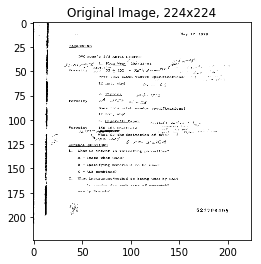


Link: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/



**Max Pooling**

A pooling layer is applied after the ReLU activation function to shrink the input to reduce the number of trainable parameters. Maximum pooling applies a filter of size *n x n* and stride *n* and calculates the maximum of that filter as the output. The output of this layer is *n2-*times smaller than the output. For example, as seen in Figure, a maximum pooling of 2x2 reduces the output a quarter the size of the input and the number of parameters by 75%.



*Convolutional Output Down 5 Layers*

**Flattening Layer**

For the purposes of classification, the final layer of the CNN is a 1-dimensional array of the class probabilities. This requires that the output of the convolutional layers also be converted to a 1-dimensional array. Following the final pooling layer, the output is flattened into a 1d array of a length equal to the number of feature maps multiplied by the output size of the final layer. For example, a final layer with a 6 x 6 output and 32 feature maps results in a 1 x 1152 array.

**Fully Connected Layers**

The fully connected layers act as a multilayer perceptron in which every neuron in the input layer is connected with weights and bias to every neuron in the output layer. These layers serve as the classification layers of the network by extracting important features related to each class. The final fully connected layer in the model is a *1 x n* array where *n* is the number of classes.

**Dropout**

Dropout is a technique used to prevent overfitting and extrapolation. During each feedforward pass in the training stage, all nodes in a certain layer are nodes are either dropped with probability 1-p or kept with probability p. Fully connected layer consist of the majority of parameters in the network and this cause neurons to develop co-dependency with each other during training which leads to the weakening of individual neuron power and leads to over-fitting of training data. Randomly turning off neurons allows for the neurons of true importance to be determined and weighted appropriately.



https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

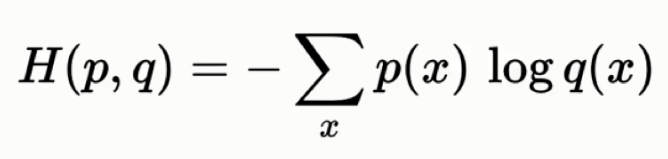
**Softmax Activation**

The softmax function normalizes the final output vector to values between 0 and 1 and divides each output value by the sum of the vector so that the sum of the vector equals 1. The result is a categorical probability distribution for the input class, with the index of the maximum value being the label of the most probable class.

**Loss**

Loss is a measure the quality of the parameters based how well the network classified the input, the difference between the output and the target labels. The entire purpose of training is to find parameters that minimize loss in the model. As this is a classification problem, cross entropy is utilized to find the minimum of the loss function. Cross entropy, or log loss, compares the probability of a correct prediction to the actual prediction and punishes both errors, meaning highly confident and wrong answers are scored worse than less confident and wrong answers.

Cross entropy is calculated:



**Optimization**

The purpose of optimization is to move towards weight and bias parameters that minimizes the loss function. Optimization calculates gradient descent and discovers the direction of steepest descent towards the minimum of the loss function along which to update the weight vector. The Adam optimizer was used for all networks.

**Experimental Setup**

# Preprocessing

# Histogram equalization was performed to increase the global contrast of each image and extract more faded text from each document. Histogram equalization is an image processing method that attempts to flatten the histogram of pixel values so bring out areas of lower local contrast by equally distributing high intensity values.

**Keras**

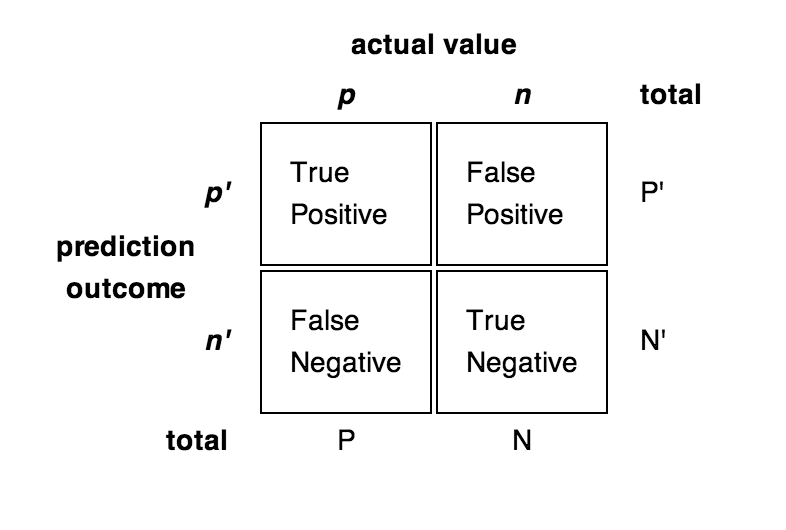
**PyTorch**

To load and prepare custom image data for analysis, PyTorch has an abstract Dataset class that representing a dataset. The custom dataset inherits Dataset functionality and overrides the “\_\_len\_\_” (size of the dataset) and “\_\_getitem\_\_” (extract and index data) methods. The provide the custom data loader with a simple method for importing the images, a reference csv is first created for both the training and testing datasets with the full image path and the numeric label. This csv is passed to the custom data loader. Within the \_\_getitem\_\_ method, each image path is read and the image imported as a numpy array of pixel values. Any image operations, including preprocessing, resizing, and sectioning are conducted here. Finally, the image and label are exported to PyTorch’s DataLoader to create the data in the format required.

The model is built as a custom CNN( ) class in a series of sequential layer blocks using the premade layers from PyTorch’s nn package. The convolution blocks consist of a convolution layer, batch normalization, ReLU activation, and a pooling layer. Each convolution layer takes as arguments the number of inputs, number of outputs (number of feature maps), number of padding layers, and size of the stride. After defining each layer block, fully connected layers (called Linear layers) are defined with the number input and output neurons. The input size of first linear layer is the size of the final pooling output (a *7x7* output with 32 feature maps results in a linear layer with 1568 neurons. Dropout layers are added between each fully connected layer as this is the location of the majority of the parameters and where overfitting is most likely to occur. The feedforward process is defined by passing the inputs through each defined layer in a sequential order.

PyTorch provides simple functions for conducting feed forward, loss calculation, optimizing, and gradient updating through backpropagation. The images are passed through the model through iteration of a user defined batch size, creating a 4-dimensional input of size *(batch size, # of color channels, image height, image width)*. The images and labels are converted to tensors and then PyTorch variables using the Variable class and run of the GPU. The images are fed to the CNN( ) class, loss is calculated, and gradients are updated with backpropagation with minimal code. The trained CNN is then tested on the testing dataset and the results converted back to numpy arrays for analysis.

Batch size is used to updating weight and bias parameters using the average output of n-number of inputs to limit the amount of parameter updating required by the network. Picking a suitable batch size is vital for producing an efficient model. Using all the training data (1 batch) to update parameters is highly computational expensive due to the large input size while updating after each input creates noise if the sample is not a good representation of the whole data. Mini-batches are smaller batches of a portion of the data used to compromise between efficiency and noise. In this project, mini-batches are used to run inputs through the model and mini-batch size is tested to ascertain the ideal size.



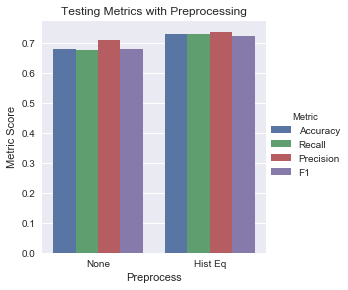
Model performance is tested on several criteria. Classification metrics used include accuracy, recall, precision, and F1 are used to analysis performance. Classification can be defined by four possible results: true positive (predicted as positive class, actual label is positive class), false positive (predicted positive class, actually negative class), false negative (predicted negative class, actually positive class) and true negative (predicted negative class, actually negative class). The classification metrics calculate the following:

* Accuracy represents how often the model correctly predicts class and is calculated: *(TP + FN) / (* *TP + FP + TN + FN)*
* Recall represents what percentage of the inputs of a class are identified as that class and is calculated: *TP / (TP + FN)*
* Precision represents what percentage of the inputs predicted as a class are actually of that class and is calculated: *TP / (TP + FP)*
* F1 is the harmonic mean of recall and precision and is calculated: *2 \* ((Precision \* Recall) / (Precision + Recall))*

The metrics are calculated as an average of individual class metrics using sci-kit learn. Loss will also be monitored through the input iterations and convergence on the minimum of the loss function will be analyzed for model comparison.

Training parameters are be examined in the PyTorch model by adjusting each one in turn while keeping other parameters constant and compared using the above metrics. Parameters tested included image size, mini-batch size, learning rate, kernel size, type of pooling, region testing, and preprocessing techniques.

**Results**



**Histogram Equalization**

Models were trained over 10 epochs with the original image pixel values and with histogram equalized pixel values. The model trained on the original images tested with an accuracy of 67% and similar recall, precision and F1. The model trained on the histogram equalized image, which are shown to have greater contrast than the original images, tested with an accuracy of 71% with similar recall, precision, and F1. As stated, this is most likely a result of the increase in contrast and the highlighting of lower local contrast through equal distribution of high intensity values.

**Image Size**

This initial training parameter tested is image size. Image size variations create a tradeoff between the number of parameters that require training and the number of features that are available for classification. Models with image sizes of *224x224, 100x100, and 50x50* are tested, with image sizes larger than *300x300* exceeding the memory capabilities of the GPU.

As expected, the larger image produced the best results, with 72.8% accuracy. Smaller images drastically worsen results, but greatly improve training time. This is due to the exponentially larger number of parameters that require training will the increased size of the input, with larger images having significantly more features to classify on but more parameters to update.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Image Size | Accuracy | Recall | Precision | F1 | Train Time |
| 50 | 0.398 | 0.407 | 0.52 | 0.376 | 1254 |
| 100 | 0.578 | 0.582 | 0.631 | 0.576 | 1842 |
| 224 | 0.728 | 0.729 | 0.736 | 0.7234 | 3039 |



**Batch Size**

Models with varying batch sizes were trained and compared. Batch sizes include 1, 10, 50, and 100 images per batch.

**Learning Rate**

Models were trained with several different learning rates, including 0.01, 0.001, 0.0001, and 0.00001. A higher learning rate will learn model parameters faster but result in a jumpy output, while a smaller learning rate will learn model parameters slower but provide a smoother convergence on the minimum of the loss function.

The below figures show the outputs of the different learning rates. A learning rate of 0.001 produces the best results, with an accuracy of 72.8% with similar recall, precision, and F1 scores. As the loss chart shows, this learning rate creates the fastest convergence on the minimum loss, with 0.0001 producing the next best results. A learning rate of 0.00001 causes a slower convergence but a more compact variance while a learning rate of 0.01 causes an initial spike of high loss and has a high loss variance. The learning rate of 0.001 produces the best compromise of convergence speed and variance.



**Kernel Size**

Models were trained with several difference kernel sizes in each convolution layer.

**Pooling Type**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pooling Type | Accuracy | Recall | Precision | F1 |
| Average | 0.304 | 0.308 | 0.469 | 0.279 |
| Max | 0.728 | 0.729 | 0.736 | 0.7234 |

Two types of pooling were considered: Max Pooling and Average Pooling. Average pooling acts similar to max pooling with the exception that the calculated value is the average of the kernel values rather than the maximum value. While max pooling is better at extract importance features, average pooling produces a smooth representation of the image. The results show that max pooling performs significantly between for these inputs. This is due to the limited number of features and majority white space of the input images. Max pooling extracts and enhances the limited number of features whereas average pooling reduces individual feature importance.

**Header and Center Cropping**

Manual examination of the documents led to the observation that most of the features of the document exist within the header or the center of the document. To test if specific regions of the document images

https://pytorch.org/tutorials/beginner/data\_loading\_tutorial.html