ML Kit: A Machine Learning Library for Rust

Owen Wetherbee (ocw6), Ethan Ma (em834), Sylvan Martin (slm338)

Keywords: Machine learning, Gradient Descent, PCA, Rust

Presentation: https://youtu.be/dLAc-HP8XBQ

Application Setting

Rust is a relatively new programming language with a dearth of machine learning infrastructure. We aimed to create a library for Rust programmers to make machine learning convenient, much like NumPy or TensorFlow in Python.

Project Description

Initially, we wanted to create a comprehensive machine learning library for Rust, which would use Rust's safety and speed to implement many algorithms used in data science. We had the original (lofty) goal of being able to run a diffusion model by the end of the semester, or be able to do anything that NumPy/SciKitLearn could do. As we began implementation we recognized that we lacked the time and resources to implement the sheer amount of algorithms we set out to. So, we shifted our focus to what we believe are some of the core algorithms in the field of Machine Learning.

In the end, we created the pure-Rust library `ml_kit`, which implements, from scratch, the following:

- Neural Network based learning, consisting of
  - the basic neural network model with user-defined network shape and activation functions for each layer
  - functionality for handling large datasets for training and testing
  - a stochastic gradient descent trainer, with whatever batch size and epochs a user may want

  - various "gradient update schedules," such as fixed learning rates, time-decay learning rates, AdaGrad, etc.
  - Convolutional Neural Networks
- Principle Component Analysis, consisting of
  - an implementation of Singular Value Decomposition (SVD),
  - using SVD to obtain a k-dimensional plane of best fit for a set of points in $\mathbb{R}^n$,
  - compressing images (or any data) using SVD by truncating low-variance dimensions

Neural Networks

Neural network based learning (i.e. deep learning) is a type of supervised learning that involves using black-box models which depend on a large number of tunable parameters to classify complicated inputs. The models are trained by iteratively evaluating their performance on training examples and updating their parameters accordingly, usually through gradient descent. We call these black-box models 'networks' because they are generally conceptualized as a series of composed functions, or 'layers', where each layer linearly maps some input vector (or, more generally, tensor) to a possibly differently-sized output vector, and then applies a simple element-wise non-linear 'activation function'. This sequence of highly inter-connected linear functions interspersed with element-wise non-linearities enables the model to identify complex features only using parameters that have relatively simple and computable gradients, ensuring efficient training.

Fully-connected neural networks

The traditional neutral network (NN) is a series of fully-connected layers, meaning each input element of a layer has a parameter through which it can linearly affect each output element. In particular, the function associated with each layer is of the form [1]

$$\vec{\mathbf{v}}_{\text{out}} = f(\vec{\mathbf{v}}_{\text{in}}) = \sigma\left(\mathbf{W}\vec{\mathbf{v}}_{\text{in}} + \vec{\mathbf{b}}\right) \ ,$$

where $\mathbf{W}$ is a matrix of 'weight' parameters, $\vec{\mathbf{b}}$ is a vector of 'bias' parameters, and σ is the non-linear activiation function. The derivative of the output vector with

1. Michael A. Nielsen, "Neural Networks and Deep Learning," 2015, http://neuralnetworksanddeeplearning.com.

respect to the input vector and these parameters can then be simply calculated as

$$\frac{\partial v_{\text{out},i}}{\partial W_{i,j}} = v_{\text{in},j} \cdot \sigma'(\vec{\mathbf{x}}) \ , \qquad \frac{\partial v_{\text{out},i}}{\partial b_i} = \sigma'(\vec{\mathbf{x}}) \ , \text{ and } \qquad \frac{\partial v_{\text{out},i}}{\partial v_{\text{in},j}} = W_{i,j} \cdot \sigma'(\vec{\mathbf{x}}) \ ,$$

where $\vec{\mathbf{x}} = \mathbf{W}\vec{\mathbf{v}}_{\text{in}} + \vec{\mathbf{b}}$ and all unspecified derivatives are zero. Thus, the output of the entire neural-network model can be simply computed as the composition of each layer function, and the gradient of the loss with respect to the model parameters (to use for parameter updates) can be determined by iteratively applying the chain rule to the above derivatives.

Implementation:

The NeuralNetwork implementation within ML Kit is straightforward and exncapsulates everything needed to build, inspect, and run a feed-forward network. Initially, a NeuralNet holds three parallel vectors:

1. Weight Matrices `Vec<Matrix<f64>>`, each matrix represents the weight between two layers.
2. Bias Vectors, one per non-input layer. Each is stored in a column sized vector of size $m \times 1$ where m is the number of neurons in said layer.
3. Activation Functions, a vector of activations indicate which activation is to be used upon each layer. Sigmoid, Relu, etc.

Our NeuralNet construction allows for two ways of instantiation. You can call NeuralNet::new(weights, biases, activations) to instantiate a network of user provided parameters. Otherwise, for ease of testing, you can call NeuralNet::from_shape(shape, activations) to create a zero initialized network and fill with NeuralNet::random_network(shape, activations).

Once instantiated, our compute_final_layer(input) function incorporates the forward pass logic. Starting with a column vector, for each layer l we multiply with weight matrix W_l and add bias b_l, and apply the element wise activation. The resulting column is then carried into the next layer until we return the output vector. We also incorporate functions such as compute_raw_layers() and compute_raw_and_full_layers().

We also implemented a paramter_count() method to tally all weights and biases. Furthermore, shape() reports the complete layer dimensions. classify(input) runs a forward pass and returns the neuron with max output. We also incorporated file I/O functionality through the write_to_file method. The function essentially encodes the number of layers, size, activations, etc.

Convolutional neural networks

Fully-connected neural networks are very general, but they do not take advantage of any potential structure of the input to minimize the number of required parameters or improve efficiency or performance. Enter, convolutional neural networks (CNNs). CNNs, primarily designed for image recognization and classification, use the spatial structure of an image to allow for shared weights and biases, reducing the number of required parameters. In particular, CNNs scan a series of filters across the image, each of which conceptually is intended to identify a certain local feature of the image (e.g. identify straight or curved lines). This scanning process is specifically done by convolving each filter with the image, hence the 'convolutional' in CNN. As described in [2], this convolution step can be implemented efficiently via matrix multiplication by first reshaping the input matrices and filters. Although we do not give the explicit formulas here (see [3]), the derivatives of the output of a convolution layer with respect to the inputs and filters can also be computed via a convolution.

CNNs often also contain pooling layers, which further reduce the dimensionality of the data. These layers scan a 'window' along the input matrix, combining all the elements in each window into a single output element. This combination process can be done in a number of ways, including taking the maximum, average, or sum of the window elements, which are referred to as max pooling, average pooling, and sum pooling, respectively. These pooling layers do not have any parameters, and their outputs are simply related to the elements of each window, so the derivatives of this layer are straightforward to compute.

A CNN usually involves a sequence of several of these convolution and pooling layers. However, once the dimensionality of the input data has been sufficiently reduced, the final classification is usually performed by a fully-connected neural network, which constitutues the last few layers of the CNN. As with the fully-connected NN, training proceeds by iteratively passing the training inputs through each layer to get the model output, and then moving back through the layers via the chain rule to compute the loss gradient.

Implementation:

Following the specifications above, the Convolutional Neural Network implementation within ML Kit implements a sequence of three kinds of layers. We define a type enum Layer to consist of the convolutional, pooling, and fully connected layers.

2. December 2019, https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/convolution_layer/making_faster.

3. Pavithra Solai, *Convolutions and Backpropagations*, April 2018, https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c.

We implement convolutional layers $ConvLayer$ that store a bank of multi-depth filters with a per filter bias. Furthermore, we included hyperparameters such as stride and padding. Pooling Layers implementex max, avg, or sum pooling and include the forward propagation abd backprop functionality in the feedforward and backprop methods respectively. We implement fully connected layeres $FullLayer$ as discussed above within the Fully Connected Neural Network section.

These layers are wrapped in our CNN datastructure $ConvNeuralNet$ which holds the subsequent methods. The compute_final_layaer() method sequentially calls the feed_forward method of each respective layer, returning final activations. The populate_gradients() method does a forward pass to record activations, computes. each intial gradient, and backtracks via calling each layers back_prop function, reshaping when needed. Grad descent step calls each layers update_params() method. sgd_batch_step() essentially does a forward and backward pass given the functions above and updates layers accordingly. train_sgd() loops over user defined epoch and applies sgd_batch_step().

Stochastic gradient descent

As referenced above, the training of these neural networks is often powered by stochastic gradient descent (SGD). SGD is a scheme for updating the paraemters of a network using the gradient of the loss for small sampled batches of training examples. In particular, to perform SGD, a set of training examples is provided, along with a specification of batch size and number of epochs. In each epoch, the whole set of training examples is sampled into different smaller sets specified by the batch size. The network parameters are then updated based on the loss incurred by each batch of training examples, rather than the entire training set. This greatly reduces the computational cost in each update step, while not changing the performance of the model in expectation. This is the primary type of gradient descent we implemented, although we do also have some support for AdaGrad.

Principle Component Analysis and SVD

Singular Value Decomposition (SVD) is a factorization of a matrix $A \in \mathbb{R}^{m \times n}$ into

$$A = U \Sigma V^\top$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices with columns $\vec{\mathbf{u}}_1, \ldots \vec{\mathbf{u}}_m$ and $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_n$ respectively, and Σ is a diagonal matrix of the form (assuming $n \leq m$)

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \\ & & & \end{bmatrix} \in \mathbb{R}^{m \times n}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ are the *singular values* of A. This allows us to write A as the linear combination

$$A = \sum_{i=1}^{n} \sigma_i \vec{\mathbf{u}}_i \vec{\mathbf{v}}_i^\top$$

Note that because the singular values are sorted in decreasing order, we can effectively "save data" in representing A by truncating all but the first $r \leq n$ singular vectors, as the last items in the sum do not contribute as much to the overall product. The immediate application is identifying the most significant axes of correlation in data, allowing dimensionality reduction of datasets by re-writing each item in the basis $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_r$. Commonly, the "data" of concern is written into the rows of A.

What we implemented

Our library implements the Golub-Kahan SVD algorithm (described in "Matrix Computations," by Golub and van Loan. [4]) which begins by bi-diagonalizing A, then performing SVD on the bidiagonalization, as the numerical stability and performance are better. Once we were able to compute the SVD of any matrix, we could use SVD as a subroutine in other useful techniques.

The (subjectively) coolest application of SVD we implemented is image compression. By taking an $m \times n$ image and writing it as a 4-tuple of matrices (R, G, B, A) representing the red, green, blue, and alpha channels of the pixels, we can perform SVD on each color channel, store only the SVD representation after truncating insignificant singular vectors, and then de-compress the image later by computing $U\Sigma V^\top$ for each color channel. In practice, one can discard roughly half the singular values and still obtain a recognizeable image. Examples of this and discussion of runtime and compression rates will be left to the evaluation section.

Another common goal in statistics is finding the "line of best fit" of a set of points, or in higher dimensions, a k-dimensional plane of best fit. For a cluster of data centered

4. Gene H. Golub and Charles F. van Loan, *Matrix Computations*, Fourth (JHU Press, 2013), ISBN: 1421407949 9781421407944, http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm.

at the origin, the first singular vector, $\vec{\mathbf{v}}_1$, is the line of best fit. This is because SVD is equivalently defined as an optimization proceedure where

$$\vec{\mathbf{v}}_1 = \underset{\|x\|=1}{\operatorname{argmax}} \|Ax\|$$

Since $\vec{\mathbf{v}}_1$ is maximizing the sum of squares of the inner products with all the data points, it is minimizing the sum of squared distances from each data point to the line spanned by $\vec{\mathbf{v}}_1$. More generally, taking the first $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_k$ vectors, we obtain a basis for a k-dimensional "plane of best fit."

We implement a proceedure which takes a data matrix $D \in \mathbb{R}^{n \times m}$ whose columns are each $\mathbb{R}^n$ data points, and returns $\vec{\mu} \in \mathbb{R}^n$ and $V \in \mathbb{R}^{n \times k}$ such that the plane defined by

$$\left\{ \vec{\mu} + \sum_{i=1}^{k} \alpha_i \vec{\mathbf{v}}_i \mid \alpha_i \in \mathbb{R} \right\}$$

is the plane of best fit for the data. This was a very minor part of our library and only consisted of a few extra lines of code, but since we had the functionality to easily implement it, we figured why not?

Relationship to Other Work

Linear algebra is the foundatin to machine learning, so we needed to use a good linear algebra library. Sylvan had previously spent winter break working on `matrix_kit`, which is a pure-Rust linear algebra package that implemented incredibly basic matrix-vector operations. We continued developing this library in parallel with `ml_kit` over the semester as we recognized more features that were needed from the linear algebra library. So, the sum of our work for the course can be thought of as the entirety of `ml_kit`, as well as significant improvement to the functionality of `matrix_kit`.

The `matrix_kit` library can be found on GitHub at https://github.com/SylvanM/matrix_kit.

Evaluation

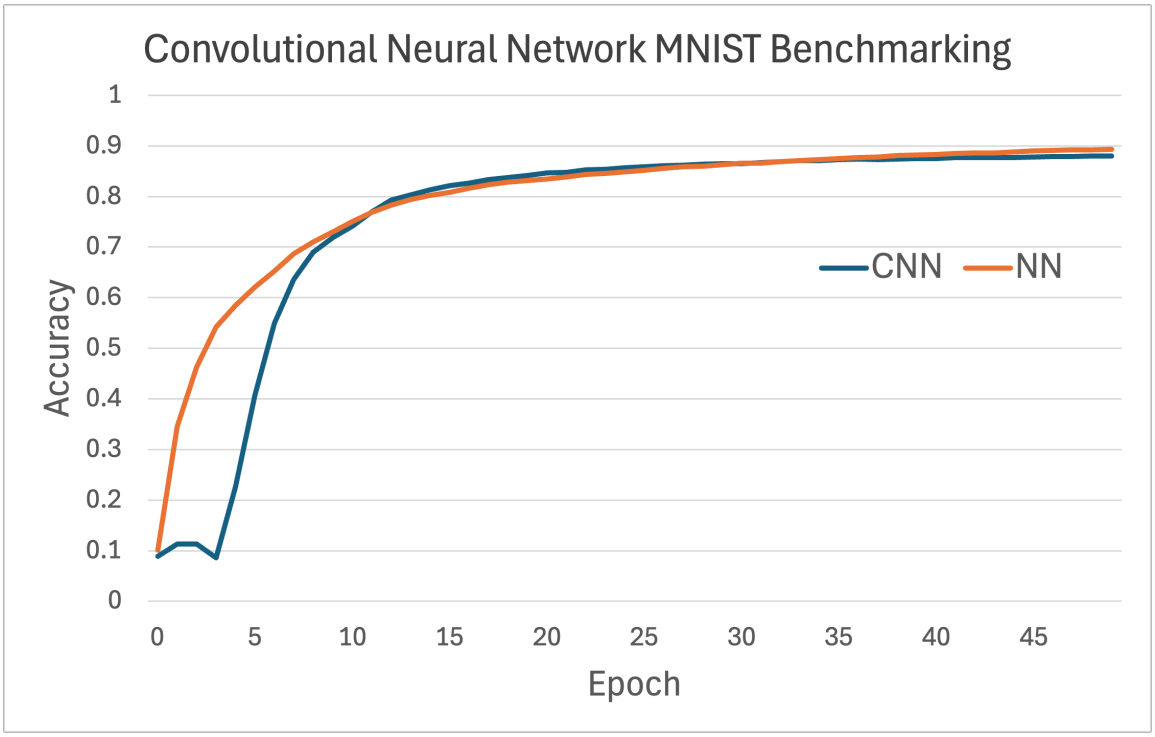
Neural Networks

We performed unit tests of various functions of our fully-connected neural network (NN), convolutional neural network (CNN), and stochastic gradient descent (SGD) implementations to ensure their correctness. Apart from this, the primary way we verified these implementations in a more integrated way was by testing toy network

models on the MNIST database of handwritten digits[5]. After parsing the MNIST data into rust and creating data items for easy integration with our SGD trainer methods, we constructed toy NN and CNN models to train on this data.

Each MNIST input consisted of a 28×28 pixel image, and each output consisted of one of the ten labels $\{0, 1, \ldots, 9\}$. We therefore constructed the fully-connected NN to have 3 layers, with input and output sizes of $(784, 16)$, $(16, 16)$, and $(16, 10)$, respectively, thus taking the $28 \times 28 = 784$ pixels to one of two possible label probabilities. For the CNN, we used the following layer structure: a convolutional layer with four 5×5 filters; a max pooling layer with a 2×2 window with stride 2; a convolutional layer with two $5 \times 5 \times 4$ filters; another max pooling layer with a 2×2 window with stride 2; and finally two fully-connected layers with input and output sizes of $(32, 16)$ and $(16, 10)$, respectively. Both convolutional layers had zero padding and stride 1, and all the layers used a sigmoid activation function.

We trained these two networks using the MNIST data and the SGD trainer methods, with the learning rate set to 0.05, a squared loss function, and a batch size of 32. The resulting accuracy after each epoch is shown in the plot below.



As can be seen, the two networks have a very similar performance curve, both achieving a classification accuracy of about 85% after 20 epochs, continuing more gradually

5. https://www.kaggle.com/datasets/hojjatk/mnist-dataset.

up to 90% by the 50-th epoch. Note that the CNN is specified by 1,004 total parameters while the fully-connected NN is specified by 13,002 total parameters. Despite this, the two networks achieve a similar performance, demonstrating the advantage of the CNN. However, it is important to note that the CNN took considerably longer to train for each epoch. This is likely due to the overhead in the convolutional layers, stemming from some inefficient implementations in both CNN and matrix kit.

Regardless, this MNIST benchmarking shows that our implementations of NN, CNN, and SGD indeed yield trainable networks which can achieve good classification accuracies. While not gauranteeing the correctness or efficiency of our implementations, this serves as a good integrated test to show that our implementations are behaving as intended.

Principle Component Analysis

In this section, we discuss the performance of our SVD-based algorithms in terms of their speed and correctness.

Singular Value Decomposition

Correctness

In numerical methods such as SVD, the accumulation of floating point error prevents us from measuring exact correctness. Instead, we define some small ε, and consider two matrices $A \approx_\varepsilon B$ to be "basically equal" if for all i, j, $|a_{ij} - b_{ij}| < \varepsilon$. When testing on large matrices, we chose $\varepsilon = 1 \times 10^{-5}$ to be our goal, though in practice on smaller matrices ($m, n \leq 50$) we attain correctness up to $\varepsilon = 1 \times 10^{-9}$.

To test correctness of our SVD implementation, we start with a randomly generated matrix $A \in \mathbb{R}^{m \times n}$ where each entry is sampled from the normal distribution $\mathcal{N}(0, 1)$, and compute its SVD into U, V, and Σ. We then check that U and V are orthogonal by testing $U^\top U \approx_\varepsilon I_m$ and $V^\top V \approx_\varepsilon I_n$. We then check that $A \approx_\varepsilon U \Sigma V^\top$.

Sometimes, our SVD implemention does not converge on a particular matrix, and so we abort the execution. The main loop of the Golub-Kahan algorithm should in theory take $\mathcal{O}(n)$ steps, so we introduced a check that if the main loop runs more than $30n$ times, we declare the instance a failure, and return empty 0×0 matrices.

With m and n sampled randomly in the range 2 through 20, we ran the above testing proceedure 10,000 times and found no correctness failures, though we observe a divergence rate of about 2.49%. That is, on small matrices, our algorithm failed to converge on 249 out of the 10,000 random matrices.

The issue of convergence seems to become better with larger matrices. When we sample m and n randomly from the range 2 through 200, we found that out of 300 trials, only 1 instance failed to converge, a rate of .33%.

For matrices with $200 \leq m, n \leq 300$, we found no matrices with failed to converge out of 30 trials.

Runtime Performance

Our SVD implementation is absurdly slow. It became impractical to test at a large scale for matrices with more than $500 \times 500 = 250,000$ entries. This is unfortunate, as most real-world applications would require much larger matrices. When we first tried image compression, we used a 4K image taken by an iPhone, and running SVD on each color channel took so long that even when let to run for 12 hours overnight, it didn't terminate. It is hard to know if this is because it genuinely just took so long, or if it fell into the small fraction of cases that caused divergence. Because of the decaying rate of divergence for larger dimensions, we believe it is the former. We also tried running SVD on matrices of comparable size (with m and n in the low thousands) and were unable to get it to terminate after several hours.

This is almost certainly due to the fact that at this stage of development we did *not* fully optimize our Golub-Kahan implementation. The first step, which relies on bi-diagonalizing the matrix, uses a sequence of Householder reflections, which if implemented efficiently can take linear time as they only affect two rows of a matrix. Further optimization would include only applying the reflection to a sub-matrix of A, and not the entire matrix. Instead, we applied the householder reflection matrix to the entire matrix A, taking $\mathcal{O}(n^3)$ time for each step. This easily balloons the runtime, since this is done inside an inner loop. At first, we tried implementing the fully optimized Golub-Kahan algorithm, but were unable to get it to work correctly. So, we focused on just trying to get a *correct* implementation instead of premature optimization. The natural next steps after this project would be to get this running faster!

Image Compression

As discussed above, we were unable to run our image compression on large images, so we started with low-resolution images. We ran our image compression algorithm on two images, one of some sheep,[6], and one of a cat.

When a matrix A is decomposed into U, Σ, V, and all but the first k singular values are ditched, we are left with three matrices of sizes $m \times k$, k and $n \times k$ (we need

6. The sheep are in the teaching barn by the Vet School. I highly recommend that you go visit and pet them! It's free and you can just walk right up to them!

only store the diagonal of Σ). So, the number of floating point values we must store as a fraction of the amount we would need to fully represent A is what we call the "compression ratio," computed as

$$r = \frac{r + rm + rn}{mn}$$

For small matrices and large values of r, this doesn't save much. However, this scales incredibly well for large matrices, as we can pick modestly small values of r and end up with impressive compression ratios while still being able to recognize the original image. A small caveat here is that typically a single pixel's channel for one color only uses one byte to represent it, whereas we are using 64-bit floating points, so one could argue that we need to include a factor of 8 in our compression ratio. However, our compression would work just as well if each color channel was represented as a floating point value between 0 and 1 (since this is what we convert the image to anyway when we begin compression), so we think it is fair to compare the sizes when thinking of the original channel as being a decimal between 0 and 1 as opposed to an integer between 0 and 255.

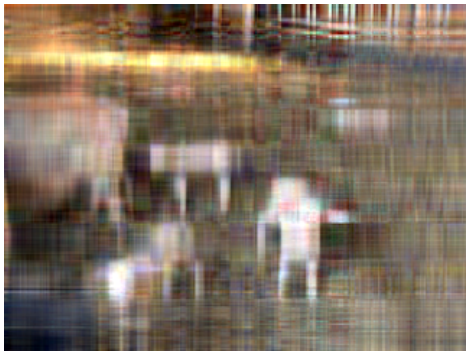
Sheep

We used a 240×320 pixel image of some sheep laying in a barn, and truncated the singular values up to the first $k \in \{100, 50, 10\}$. Running SVD to decompose the image took **1m 15s**.

Original

$k = 100$, 73.0% compression

$k = 50$, 36.5% compression



$k = 10$, 7.3% compression

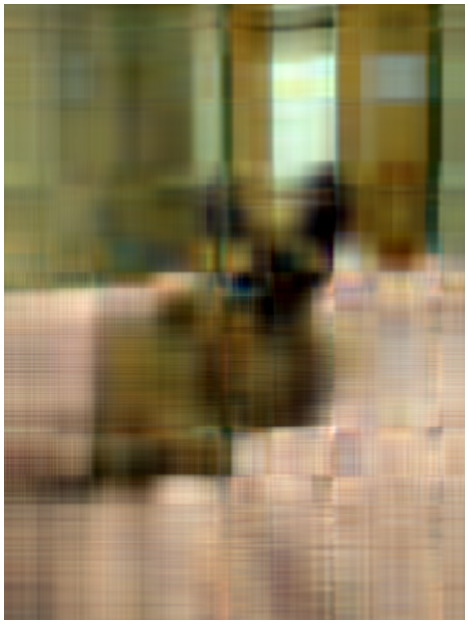
Panini (the cat)

On a 640×480 image of Panini, SVD took **20m 50s** to finish. Here are the results using the first $k \in \{150, 50, 7\}$ singular values.

Original

$k = 150$, 54.7% compression

$k = 50$, 18.2% compression



$k = 7$, 2.6% compression

References

, December 2019. https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/convolution_layer/making_faster.

. https://www.kaggle.com/datasets/hojjatk/mnist-dataset.

Golub, Gene H., and Charles F. van Loan. *Matrix Computations*. Fourth. JHU Press, 2013. ISBN: 1421407949 9781421407944. http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm.

Nielsen, Michael A. "Neural Networks and Deep Learning," 2015. http://neuralnetworksanddeeplearning.com.

Solai, Pavithra. *Convolutions and Backpropagations*, April 2018. https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c.