Image Classification Using an Artificial Neural Network

Frank Yang and Il Shan Ng, March 15 2017

1 Introduction

We implement an artificial neural network to tackle the problem of image classification, a routine task in the field of computer vision. Our goal in image classification is to find a function that best maps each image, represented in terms of its pixel values, to its ground truth label. Four characteristics of this problem make it particularly challenging. First, because each pixel is a feature for which a weight has to be estimated, classification of even small-sized images (28 by 28 pixels in our case) constitutes a high-dimensional problem. Additionally, because each monochrome pixel can take on a large number of discrete values (256 values for 8-bit images), the feature space is immense, and so there may be insufficient training instances for each combination of feature values to ensure that the machine is learning. Second, the large number of features means that data on images are likely to be non-linearly-separable. Third, because images with the same ground truth label can portray their subjects in a variety of styles, positions, and with possible occlusion, generalization of the learned weights is difficult. Lastly, the immense number of features and training instances means that convergence is likely to be slow, which cannot always be countered with a higher number of iterations due to limited computational resources.

We circumvent the first problem of high-dimensionality by using a proportionately large dataset to train our machine. This ensures that each combination of feature values occurs frequently enough for the algorithm to compute reliable weights. To address the second problem of linear non-separability, we choose an artificial neural network as our machine learning algorithm. Neural networks not only allow us to model non-linearities in the data, but also stand at the frontier of modern solutions to image classification problems in terms of accuracy rates and training times.[1] To alleviate the third problem of low-generalizability, we augment our basic implementation with two advanced components to reduce overfitting, and perform cross-validation to determine the extent of overfitting. For the last problem of slow convergence, we implement four more advanced components to speed up the rate of convergence, so that we may obtain reliable results without the need for too many iterations.

2 Data and Data Pre-processing

Although we used a variety of image datasets when learning how to implement our neural network, our final goal was to classify images of handwritten digits provided in the famous MNIST dataset. We obtained a variation of this dataset from Kaggle,[2] a popular platform that hosts machine learning competitions. The main advantage of doing so was that each image had been re-expressed in terms of its pixel values and stored conveniently in a csv file, removing the need for us to process the raw images manually.

The full dataset contained 42,000 instances of pixel values. We randomly sampled and reserved 12,000 of these as the test set. With the remaining 30,000 instances, we performed 5-fold cross-validation. That is, we first segmented the 30,000 instances into five subsets of 6,000 instances each, picked the first subset as our validation set, and pooled the remaining four to obtain our training set. We then trained our neural network on the training set of 24,000 instances, and used the trained weights to make predictions on the validation set to obtain a cross-validation accuracy. By choosing a different subset as the validation set each time, we perform five rounds of cross-validation to obtain five cross-validation accuracies. The average cross-validation accuracy serves as a reasonable measure of how well our neural network will perform on the test set.

Before training our neural network or testing it on the validation and test sets, we pre-processed the data by zero-centering and normalizing the feature values. That is, for each feature, we first computed the mean and standard deviation of the pixel values across all instances. We then subtracted the computed mean from each instance, and divided the result by the computed standard deviation. This pre-processing is an important step in ensuring that our implementation is numerically stable; by zero-centering and normalizing, we avoid

[1] The state-of-the-art solution to image classification problems is actually the Convolutional Neural Network. However, we found this too difficult to implement within a short time frame, and decided to focus on regular neural networks.

[2] https://www.kaggle.com/c/digit-recognizer/data

any extremely large or negative values that may lead to exponential over or underflows when executing the code.

3 Neural Network Structure

3.1 Basic Architecture

Figure 1 shows the basic architecture of our neural network. To keep our algorithm simple and computational needs low, we use only one hidden layer. The input layer has however many units as there are features F in the dataset, plus one unit x_0 that represents the bias for the input layer, and the size of the output layer is determined by the number of class labels C present in the dataset. The number of hidden layer units H, excluding the hidden layer bias a_0, is a hyper-parameter which allows us to add complexity to the network at the expense of training speed.

Because our dataset consists of 28 by 28 pixel images, $F = 784$ and the input layer for our network contains 784 units plus one bias unit. Because there are ten labels in the dataset, each corresponding to a unique digit 0 through 9, $C = 10$ and the output layer consists of 10 units. We set the number of hidden units H to 100, a number that we found gave us reasonably high accuracy rates compared to those achieved by the literature,[3] while at the same time preserved fast run-times to allow us to test our advanced components.

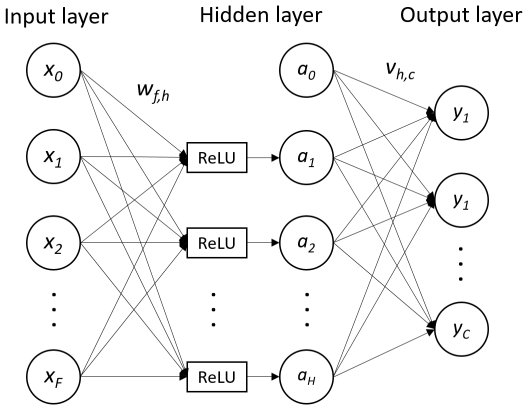


Figure 1: Neural Network Architecture

3.2 Forward Propagation

Although Figure 1 makes our network look simple, it shows how the features for only one instance are transformed into output class scores. In reality, because we have to compute class scores for as many as 24,000 instances at a time, to make our implementation efficient, we vectorize the forward propagation by defining matrices and and making use of matrix products. Let N be the number of instances whose class scores are being computed, and $\boldsymbol{X}$ be a $(F+1) \times N$ by matrix whose nth column corresponds to the feature values plus bias for instance n. With reference to Figure 1, let the weight to be multiplied by the fth input feature value and fed into the hth hidden unit, $h \neq 0$, be $w_{f,h}$. Define a $H \times (F+1)$ matrix $\boldsymbol{W}$ whose hth row and fth column stores the weight $w_{f,h}$. Then, we can compute the activation scores $\boldsymbol{S}$ for all instances entering the hidden layer using the matrix product

$$\boldsymbol{S} = \boldsymbol{W}\boldsymbol{X} = \begin{bmatrix} w_{0,1} & w_{1,1} & \cdots & w_{F,1} \\ w_{0,2} & w_{1,2} & \cdots & w_{F,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{0,H} & w_{1,H} & \cdots & w_{F,H} \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ x_F^{(1)} & x_F^{(2)} & \cdots & x_F^{(N)} \end{bmatrix}.$$

To verify that this product is correct, we check that the activation score for the nth instance that enters the hth hidden unit $s_h^{(n)} = \sum_{f=0}^{F} \mathrm{w}_{f,h}\, x_f^{(n)}$ is indeed a linear combination of the appropriate weights and features. A Rectified Linear Unit (ReLU) activation function $g(\boldsymbol{S}) = \max(0, \boldsymbol{S})$ is then applied element-wise to the matrix $\boldsymbol{S}$ to obtain the hidden unit values $\boldsymbol{A}$, thresholded at zero. We choose ReLU over the sigmoid function because the former's non-saturating form leads to faster convergence (Krizhevsky et al., 2012), and because the max operation is numerically more stable than the exponentiations required for the latter. To compute the output class scores, we define a second weights matrix $\boldsymbol{V}$, of dimension $C \times (H+1)$, whose cth row and hth column stores the weight $v_{h,c}$ to be multiplied by the hth hidden unit's value and entered into

[3] We compared our accuracy rates to a list of error rates achieved by various neural network setups found at http://yann.lecun.com/exdb/mnist/.

the cth output unit (Figure 1). By padding $\boldsymbol{A}$ with an initial row of 1's to store the hidden layer biases, the class scores $\boldsymbol{Y}$ for all instances can be simultaneously computed through the matrix product

$$\boldsymbol{Y} = \boldsymbol{V}\boldsymbol{A}' = \begin{bmatrix} v_{0,1} & v_{1,1} & \cdots & v_{H,1} \\ v_{0,2} & v_{1,2} & \cdots & v_{H,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{0,C} & v_{1,C} & \cdots & v_{H,C} \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ a_1^{(1)} & a_1^{(2)} & \cdots & a_1^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ a_H^{(1)} & a_H^{(2)} & \cdots & a_H^{(N)} \end{bmatrix}.$$

Each column in the $C \times N$ matrix $\boldsymbol{Y}$ therefore corresponds to the class scores for each of the N instances.

3.3 Computing the Loss and Accuracy

Given the ground truth labels in the dataset, we create a $C \times N$ matrix $\boldsymbol{Y}^*$ whose c, nth entry is 1 if class c corresponds to the true label for instance n, and 0 otherwise. Our basic implementation uses the squared loss function, which, with the matrix entries we have defined, takes on the form $L = \sum_{c=1}^{C} \sum_{n=1}^{N} \frac{1}{2}(y_{c,n} - y_{c,n}^*)^2$. Essentially, the algorithm accumulates loss for every training instance whose class scores do not approximately equal a basis vector. Accuracy, defined as the proportion of instances to which the network assigns a correct label, is easier to compute. By taking the argmax over all class scores for each instance in $\boldsymbol{Y}$, we collapse matrix $\boldsymbol{Y}$ into a vector of predicted class labels. We then compare the predicted labels to the ground truth labels, count the number of correct predictions, and average the count over the total number of instances. Mathematically, this can be written as $accuracy = \frac{1}{N} \sum_{n=1}^{N} I\left(\text{argmax}_c\, y_{c,n} = y_n^*\right)$, where I is the indicator function and y_n^* indexes the vector of true labels.

3.4 Backpropagation

For the backpropagation step, we use calculus to derive the gradients for both the input and hidden layer weights analytically (henceforth called the analytical gradient). We start with the loss function for a single class c and instance n, which can be expressed as $L_c^{(n)} = \frac{1}{2}(y_{c,n} - y_{c,n}^*)^2$. Taking the partial derivative of this expression w.r.t. $y_{c,n}$ gives $\frac{\partial L_c^{(n)}}{\partial y_{c,n}} = y_{c,n} - y_{c,n}^*$. Recalling that $y_{c,n} = \sum_{h=0}^{H} v_{h,c}\, a_h^{(n)}$, we find that $\frac{\partial y_{c,n}}{\partial v_{h,c}} = a_h^{(n)}$, and so using the chain rule, we derive the gradients for the hidden layer weights $\frac{\partial L_c^{(n)}}{\partial v_{h,c}} = \frac{\partial L_c^{(n)}}{\partial y_{c,n}} \frac{\partial y_{c,n}}{\partial v_{h,c}} = (y_{c,n} - y_{c,n}^*) a_h^{(n)}$. In matrix form, this is the product $\boldsymbol{dV} = (\boldsymbol{Y} - \boldsymbol{Y^*})(\boldsymbol{A}')^T$.

Obtaining the analytical gradients for the input layer weights requires more work. Recall from Figure 1 that the effect of any change in $w_{f,h}$ "flows" through the ReLU activation function, the hidden layer units a_h, and finally to the output layer units y_c. We calculate the gradients step by step in the reverse order. First, because $y_{c,n} = \sum_{h=0}^{H} v_{h,c}\, a_h^{(n)}$, for $h \neq 0$ to exclude the hidden layer bias, $\frac{\partial y_{c,n}}{\partial a_h^{(n)}} = v_{h,c}$. Next, because $a_h^{(n)} = g(s_h^{(n)})$, where g is the ReLU activation function, $\frac{\partial a_h^{(n)}}{\partial s_h^{(n)}} = g'(s_h^{(n)}) = 1$ if $s_h^{(n)} > 0$ and 0 otherwise. Finally, recalling that $s_h^{(n)} = \sum_{f=0}^{F} \mathrm{w}_{f,h}\, x_f^{(n)}$, we obtain the last required partial derivative $\frac{\partial s_h^{(n)}}{\partial w_{f,h}} = x_f^{(n)}$. Putting everything together using the chain rule, we have

$$\frac{\partial L_c^{(n)}}{\partial w_{f,h}} = \frac{\partial L_c^{(n)}}{\partial y_{c,n}} \frac{\partial y_{c,n}}{\partial a_h^{(n)}} \frac{\partial a_h^{(n)}}{\partial s_h^{(n)}} \frac{\partial s_h^{(n)}}{\partial w_{f,h}} = (y_{c,n} - y_{c,n}^*) v_{h,c} I(s_h^{(n)} > 0) x_f^{(n)},$$

where I is the indicator function. This is equivalent to the matrix product $\boldsymbol{dW} = \left\{ \left[\boldsymbol{V}^T (\boldsymbol{Y} - \boldsymbol{Y^*}) \right] \circ \boldsymbol{I}(\boldsymbol{S} > 0) \right\} \boldsymbol{X}^T$, where $\circ$ represents element-wise multiplication and $\boldsymbol{I}$ is the indicator matrix.

3.5 Stochastic Gradient Descent and Gradient Checking

Using all of the training data to compute the analytical gradients for a single round of parameter update is computationally wasteful. Instead, we implement stochastic gradient descent (SGD), which computes the analytical gradient using a random subset of the training data, whose size is controlled by a hyper-parameter

known as the SGD batch size. This technique relies on the inherent correlation among the instances in the training data; because the data are correlated, a representative sample of the training set should give a reasonable approximation of the analytical gradients. Using the standard batch size of 256, we compute $\boldsymbol{dW}$ and $\boldsymbol{dV}$ and perform the parameter updates $\boldsymbol{W}_{t+1} \leftarrow \boldsymbol{W}_t + \alpha\, \boldsymbol{dW}$ and $\boldsymbol{V}_{t+1} \leftarrow \boldsymbol{V}_t + \alpha\, \boldsymbol{dV}$, where α is the learning rate, another hyper-parameter.

Because deriving the analytical gradients using calculus is error-prone, we implement gradient checking. Specifically, we compute the numerical gradients using the centered finite difference approximation given by $\frac{\partial L}{\partial \theta_j} = \frac{1}{2\epsilon}[L(\theta_1, ..., \theta_j + \epsilon, ..., \theta_J) - L(\theta_1, ..., \theta_j - \epsilon, ..., \theta_J)]$, where L is the squared loss function, θ_j is an arbitrary weight in the neural network, and ϵ is the small change to the weight whose gradient is being approximated. Implementation-wise, we are looping through all of the weights in $\boldsymbol{W}$ and $\boldsymbol{V}$ and for each weight, introducing a small error, running forward propagation, and seeing how much the squared loss changes. We compare our numerical gradients to the analytical gradients to ensure that the latter has been computed correctly. Once we have ascertained this, we switch gradient checking off because it is computationally expensive and does not contribute any further to our algorithm.

4 Advanced Components

4.1 L2 Regularization

Because a neural network uses so many parameters, it has a tendency to overfit to the training data. We prevent this by augmenting our basic implementation with L2 regularization, which penalizes according to the squared magnitude of all weights in the network. Mathematically, we are adding $\frac{1}{2}\lambda \left(\sum_{h=1}^{H} \sum_{f=0}^{F} w_{h,f}^2 + \sum_{c=1}^{C} \sum_{h=0}^{H} v_{h,c}^2 \right)$ to the data loss, where λ is the regularization strength, another hyper-parameter. Our parameter updates must now also minimize the regularization loss, and take on the form $\boldsymbol{W}_{t+1} \leftarrow \boldsymbol{W}_t + \alpha(\boldsymbol{dW} + \lambda \boldsymbol{W})$ and $\boldsymbol{V}_{t+1} \leftarrow \boldsymbol{V}_t + \alpha(\boldsymbol{dV} + \lambda \boldsymbol{V})$.

4.2 Dropout

Dropout is another technique that complements L2 regularization in preventing overfitting. With dropout probability p, a hyper-parameter, we deactivate a hidden layer unit by setting its value to zero during forward propagation. This prevents the hidden layer units from adapting too much to similar input signals (Srivastava et al., 2014). We use dropout only during the training phase, and turn it off when computing cross-validation and test set accuracies.

4.3 Cross-entropy Loss Function

The squared loss function in our basic implementation may lead to slow convergence due to low convexity.[4] To allow our algorithm to achieve convergence in fewer iterations, we try the softmax cross-entropy loss function of the form $L = \frac{1}{N} \sum_{n=1}^{N} -\log p_n$ where p_n is the probability which the network predicts for the true class for each instance, and can be obtained by exponentiating the columns in matrix $\boldsymbol{Y}$ and normalizing. The analytic gradients computed during backpropagation also change but not by a lot. The partial derivative of the cross-entropy loss w.r.t. the output scores can be shown to be $\frac{\partial L^{(n)}}{\partial y_{c,n}} = p_{c,n} - I(y_n^* = c)$, where the indicator function I equals 1 when c is the true class for that instance. In matrix notation, this is equivalent to $\boldsymbol{P} - \boldsymbol{Y^*}$, where $\boldsymbol{P}$ is a matrix of probabilities obtained by exponentiating all the entries in $\boldsymbol{Y}$ and normalizing within each column. Computing the new analytical gradients is simply a matter of replacing $\boldsymbol{Y} - \boldsymbol{Y}^*$ in the squared loss case in Section 3.4 with $\boldsymbol{P} - \boldsymbol{Y^*}$.

4.4 Improved Initialization of Weights

Our basic implementation initializes all entries in $\boldsymbol{W}$ and $\boldsymbol{V}$ to small random numbers. One problem with this is that the variance of the output scores $y_{c,n}$ grows with the number of input and hidden units. Because

[4] We learned about this from a video of a talk by Andrew Ng at the Institute for Pure & Applied Mathematics. The claim appears at around 1:20 in the YouTube link https://www.youtube.com/watch?v=s9-MaMoR_Ww&t=807s.

our network contains 784 features and 100 hidden units, the computed output scores may be so large that exponentiating them to compute the cross-entropy loss leads to overflow when running the code. To correct for this, we follow the recommendation by He et al. (2015) and initialize our weights to small random numbers multiplied by 2 and divided by the square root of the number of input and hidden units. This not only makes our implementation more numerically stable, but also improves the rate of convergence.

4.5 Annealing the Learning Rate

A high initial learning rate makes the parameter updates larger and therefore helps the algorithm converge faster for the first few iterations. After many iterations, however, this may cause the weights to thrash around too much, preventing convergence. We anneal the learning rate by introducing a $1/t$ decay, where t is the iteration number, so that fine-tuning of the weights becomes possible for the later iterations.

4.6 Nesterov Momentum

The common metaphor for gradient descent is to imagine a ball being dropped onto an error surface and rolling down along the steepest slope to the local minimum. What gradient descent does is that it aligns the velocity of movement along the error surface in the direction of steepest descent. However, this metaphor is imprecise because we should be concerned not with velocity but with acceleration when dropping a ball. Based on this idea, we augment our gradient descent with a momentum update. Let $\frac{\partial L}{\partial \boldsymbol{W}}$ be the gradient of the loss w.r.t. to the weight matrix $\boldsymbol{W}$. We define a variable $\boldsymbol{w}$ that is a matrix that stores the velocity of $\boldsymbol{W}$. The classic momentum updates the velocity as $\boldsymbol{w}_{t+1} = m\boldsymbol{w}_t + \alpha\frac{\partial L}{\partial \boldsymbol{W}}$, where m is the "momentum" coefficient that serves as the role of friction in kinetics. Then, update the location as $\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \boldsymbol{w}_{t+1}$. There is an improved version of this momentum-based gradient descent, which was first proposed by Nesterov (1983). Nesterov suggests that we adjust the velocity based not on the gradient at the current position but on the one-step-ahead position where the velocity vector points to. To implement this, we use a simplified formulation given in Bengio et al (2013). We store the past velocity from the previous iteration and change the current position to the one-step-ahead position relative to the previous position.

5 Results

5.1 Effects of Advanced Components

We tested the effect of including each advanced component on the average 5-fold cross-validation (CV) accuracy. For each run, we used 10,000 iterations per round of cross-validation. For the advanced components that introduced hyper-parameters, we fine-tuned these hyper-parameters to produce the highest average CV accuracy.[5] Table 1 shows the results of our tests. The baseline model that uses the squared loss function performs surprisingly well, with an average CV accuracy of 95.96%.

Table 1: Effect of Advanced Components on Average Cross-validation Accuracy

Baseline	L2 Reg	Dropout	Cross-entropy	Weights	$1/t$ decay	Nesterov
95.96%	96.23%	95.99%	96.18%	96.29%	96.21%	96.22%

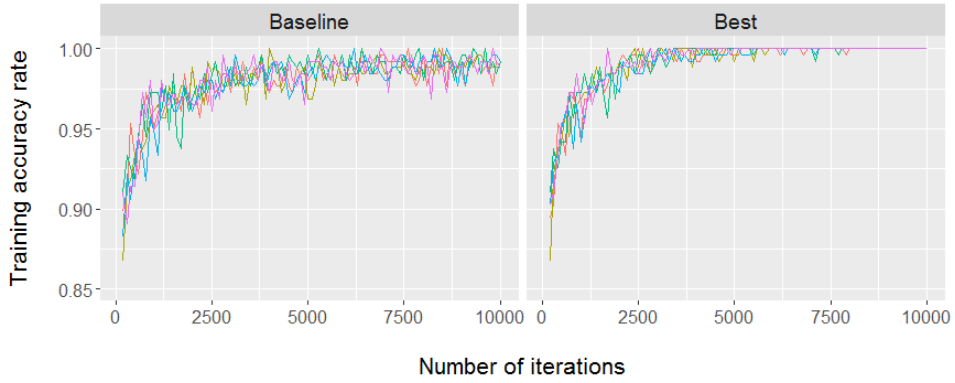
Based on our results, dropout leads to the least improvement — the small margin could in fact be the result of noise. We suspect that this is due to the small number of hidden units in our network, which masks the true potential of dropout. Indeed, the "drastic improvements" reported in Srivastava et al. (2014) occured in networks with two to four hidden layers that each houses 1024 to 2048 hidden units.

[5]The fine-tuned hyper-parameters are shown in the python script that accompanies this report.

5.2 Best Implementation

Although Table 1 suggests that the last four components do improve the rate of convergence, we make this more explicit by choosing our "best" implementation and comparing its performance to our baseline implementation. For our best implementation, we turn on L2 regularization, use the cross-entropy loss function, improve the initialization of weights, use the 1/t decay to anneal the learning rate, and incorporate Nesterov momentum. Figure 2 plots the training accuracy rate against the iteration number for both the baseline and best implementations. Different colors represent the different rounds of cross-validation. We see that the training accuracy rate thrashes around considerably for the baseline model, even in the later iterations. On the other hand, convergence is smoother in our best implementation.

Figure 2: Training Accuracies for our Baseline and Best Implementation



The average CV accuracy for our "best" implementation is 96.21%. This is lower than the CV accuracies achieved by some of the advanced components alone in Table 1. Apparently, the effects of individual advanced components are not additive when they are used together. This could be due to interactions among the individual components, or a lack of training instances to bring out the full potential of these components. Finally, we run our best implementation on the test data and achieve a test set accuracy of 95.93%, which is not too far below the average CV accuracy.

6 Discussion

6.1 Limitations

We have created a basic implementation of an artificial neural network that classifies handwritten digits with a CV accuracy of close to 96%. We then augmented our basic implementation with a few advanced components to reduce overfitting and improve the convergence rate, producing some favorable results. The main limitation of our analysis was our inability to test for interactions among the advanced components due to the short time frame for this project. It is possible, for example, that L2 regularization performs better when used in conjunction with the cross-entropy loss function, or that the optimal regularization strength changes with different loss functions. Additionally, the CV accuracy seems to be bound by the number of training instances. Using a larger training set may allow the advanced components to perform better.

6.2 Further Work

Future work should extend our analysis to networks with more hidden layers and hidden nodes. This not only allows dropout to perform better, but also better approximates how well the advanced components may perform in modern networks, which have millions of weights. Given more time, we would also try out different activation functions, such as the Random Rectified Linear Unit function, which thresholds the activation at zero at set probability. Finally, we would like to fine-tune our neural network to other benchmark data sets for neural network training, most notably the CIFAR-10 and CIFAR-100 data sets, which contain colored images with 10 and 100 labels respectively.

7 Bibliography

Bengio, Yoshua, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in optimizing recurrent networks." In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 8624-8628. IEEE, 2013.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In Proceedings of the IEEE international conference on computer vision, pp. 1026-1034. 2015.

Nesterov, Yurii. "A method for unconstrained convex minimization problem with the rate of convergence O (1/k2)." In Doklady an SSSR, vol. 269, no. 3, pp. 543-547. 1983.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012.

Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15, no. 1 (2014): 1929-1958.

Stanford University Computer Science Deparment. "CS231n Convolutional Neural Networks for Visual Recognition." Last accessed on March 15, 2017. http://cs231n.github.io/.