Deep Learning Assignment 1

by Furious Logic

Kelly Zhang kz918@nyu.edu
Vikram Mullachery mv333@nyu.edu
Nikita Nangia nn1119@nyu.edu

March 7, 2017

1 Backprop

1.1 Nonlinear activation functions

We know $\frac{\partial E}{\partial x_{in}} = \frac{\partial E}{\partial x_{out}} \frac{\partial x_{out}}{\partial x_{in}}$. So for each of these cases, we simply compute $\frac{\partial x_{out}}{\partial x_{in}}$, and then multiply the result by $\frac{\partial E}{\partial x_{out}}$

- **sigmoid**
  Note these:
  $$1 - x_{out} = 1 - \sigma(x_{in}) = \frac{e^{-x_{in}}}{1 + e^{-x_{in}}}$$
  $$\frac{\partial x_{out}}{\partial x_{in}} = (-1)(1 + e^{-x_{in}})^{-2}(-e^{-x_{in}})$$
  Simplifying the second partial differential application, we get:
  $$\frac{\partial x_{out}}{\partial x_{in}} = \frac{e^{-x_{in}}}{1 + e^{-x_{in}}} \frac{1}{1 + e^{-x_{in}}} = x_{out}(1 - x_{out})$$
  So we have:
  $$\frac{\partial E}{\partial x_{in}} = x_{out}(1 - x_{out}) \frac{\partial E}{\partial x_{out}}$$

- **tanh**
  Note these:
  $$1 + x_{out} = 1 + \tanh(x_{in}) = \frac{2e^{2x_{in}}}{e^{2x_{in}} + 1}$$
  $$\frac{\partial x_{out}}{\partial x_{in}} = \frac{1}{e^{2x_{in}} + 1} \frac{\partial (e^{2x_{in}} - 1)}{\partial x_{in}} + (e^{2x_{in}} - 1) \frac{\partial \frac{1}{e^{2x_{in}} + 1}}{\partial x_{in}}$$
  Simplifying the second line's partial differential application, we get:
  $$\frac{2e^{2x_{in}}}{e^{2x_{in}} + 1} + \frac{-2e^{2x_{in}}(e^{2x_{in}} - 1)}{e^{2x_{in}} + 1} \frac{1}{e^{2x_{in}} + 1} = (1 + x_{out}) - x_{out}(1 + x_{out})$$

So we have:
$$\frac{\partial E}{\partial x_{in}} = (1 + x_{out})(1 - x_{out})\frac{\partial E}{\partial x_{out}}$$

- **ReLU**
  We have:
$$\frac{\partial x_{out}}{\partial x_{in}} = \begin{cases} 1 & \text{when } x_{in} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$
  So we have:
$$\frac{\partial E}{\partial x_{in}} = \begin{cases} \frac{\partial E}{\partial x_{out}} & \text{when } x_{in} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

1.2 Softmax

Let $S = \sum_k e^{-\beta(X_{in})_k}$. So,
$$\frac{\partial(1/S)}{\partial(X_{in})_q} = \frac{\beta e^{-\beta(X_{in})_q}}{S^2}$$
Using this fact allows us to evaluate,
$$\frac{\partial(X_{out})_i}{\partial(X_{in})_j} = \frac{1}{S}\frac{\partial e^{-\beta(X_{in})_i}}{\partial(X_{in})_j} + e^{-\beta(X_{in})_i}\frac{\beta e^{-\beta(X_{in})_j}}{S^2}$$

- Case $i \neq j$
  The first term is zero, since $(X_{in})_i$ is a constant with respect to $(X_{in})_j$, so we have:
$$\frac{\partial(X_{out})_i}{\partial(X_{in})_j} = \beta(X_{out})_i(X_{out})_j$$
- Case $i = j$
  The partial derivative in the first term is now, $\frac{-\beta e^{-\beta(X_{in})_j}}{S} = -\beta(X_{out})_j$. So the entire expression reduces to:
$$\frac{\partial(X_{out})_i}{\partial(X_{in})_j} = \beta(X_{out})_j((X_{out})_j - 1)$$

2 Techniques

2.1 Optimization

Momentum is a technique to help speed up gradient descent optimization. Sometimes the error gradient calculated in SGD is too noisy and unstable from time-step to time-step for fast gradient descent. Momentum aims to stabilize the weight updates by keeping a history of gradients calculated from previous time-steps and basing the weight parameter update on both the current time-step's error gradient and the history of gradients.

- Classical Momentum:
  In Classical Momentum, the parameter update is based on a linear combination of the current time-step's error gradient and the velocity used to update the weight parameters in the previous time-step.

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

  In the above equations v represents velocity and θ represents the weight parameters of the function we are optimizing f; μ, the momentum coefficient, is the weight put on the "historical" velocity used to update weights in the previous time-step and ϵ, the learning rate, represents the weight put on the current time-step's gradient.

- Nesterov's Accelerated Gradient Method:
  Nesterov's Accelerated gradient method is another variant of the momentum method that also aims to stabilize gradient descent.

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\theta_t + \mu v_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

- Comparison:
  The only difference between this method and Classical Momentum is the method in which the current time-step's error gradient is calculated. While Classical Momentum calculates the error gradient straightforwardly, in the same way that normal SGD does, in Nesterov's method the error gradient used for the current time-step is calculated after updating the weights in the direction of the previous time-step's velocity vector. In other words, in Nesterov's method in each time-step of update, the algorithm first continues along the direction of the previous time-step's velocity vector (with momentum weight μ) *and then* calculates the error gradient, which is used as the current time-step's error gradient.

2.2 Reduce Over-fitting

1. Applying dropout is essentially equivalent to sampling a "thinned" network from our full network. For a network with n nodes, the are 2^n possible combinations of the nodes, therefore 2^n possible thinned networks to sample. The thinned networks share weights so the number if parameters doesn't increase. So using dropout is like training combinations of different thinned networks and then averaging them, like in ensembing, for the final model. Explicit averaging is prohibitive but an approximate averaging is used to get the "ensembled" network during test time.

2. During training time, each unit of the hidden layer has a probability p of being dropped. To achieve this dropout, a vector of independent Bernoulli random variables, $\mathbf{r}^{(l)}$, is sampled and multiplied element-wise with the output of that layer, $\mathbf{y}^{(l)}$. This gives us the thinned outputs, $\widetilde{\mathbf{y}}^{(l)}$. These thinned outputs are used as the input for the following layer. This input undergoes a linear transformation by being multiplied with the weight matrix of that layer and adding in a bias term: $\mathbf{W}^{(l+1)}\widetilde{\mathbf{y}}^{(l)} + \mathbf{b}^{(l+1)}$. This is then passed through a non-linear function to create the output.
   At test time though, instead of explicitly averaging all the thinned networks, the weights of the hidden units are scaled with probability p thus combining all the thinned networks into a single neural network. In this framework, the expected output of any hidden units under the distribution used to drop units during training is the same as the actual output during testing.

3. As illustrated in Simard et al. (2003), a number of data augmentation methods can be applied to the MNIST dataset to achieve an artificial expansion of the dataset. We can do the elementary step of doing many small rotations of the training data. Though on the whole, a rotated image is clearly still the same image, on a pixel-by-pixel it's a significant transformation. Additionally, we can translate and skew the training images. And finally, we can conduct "elastic distortions" which are meant to emulate the random oscillations of hand muscles.

2.3 Initialization

2.3.1 He et al. [4] vs. Xavier Initializations

The primary goals of a good initialization is to allow the information flow in both forward- and backward-propagation to be strong and not vanish. One way to help ensure this is to ensure that the the variance of both the activations forward and the gradients backwards to be constant from layer to layer. A poor choice of initializations can allow the gradient to vanish, as is the issue with recurrent neural networks, and prevent the first layers from getting a strong enough error signal to learn properly. In the papers by He et al. [4] and Glorot et al. [5], the authors discuss initialization techniques to mitigate against this issue.

The Xavier Initialization by Glorot et al. [5] was developed under the simplifying assumptions that the activation functions are linear. Their initialization compromises between maintaining constant variance of both the activations forward and the gradients backwards. The following equation represents the distribution from which Xavier initialized weights are drawn. n_l represents the number of units in layer l.

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}, \frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}\right]$$

He et al. [4] developed an initialization that also aims to meet the goal of ensuring that the forward from activations layer to layer have constant variance, but they do so for ReLU activations and do not make the simplifying assumption that activations are linear. The following equation is distribution from which they drew initial weight values from. Note that these initializations were not used to initialize the bias parameters, which were initialized to zero.

$$W \sim N\left(0, \sqrt{\frac{2}{n_l}}\right)$$

2.3.2 Pre-Training for Initialization

In the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition", the VGG team initialized the parameters for their larger, deeper model using the parameters learned from pre-training with a smaller, shallower model that had random weight initializations. Specifically, the first four convolutional layers and the last three fully connected layers of the larger model were initialized with the parameters from those of the shallower model; the intermediate layers of the larger model were initialized randomly. Random initializations were sampled from a normal distribution with zero mean and 0.01 variance. All biases were initialized to zero.

Coates et al. (2011) extracted features from unlabeled data by extracting randomly chosen patches from the unlabeled inputs and utilizing several different unsupervised learning methods that had reconstruction and clustering based objective functions. In terms of reconstruction-based objectives, Coates et al. used both sparse auto-encoders and sparse Restricted Boltzman Machines (RBMs). The sparsity constraint is obtained by penalizing (increasing error) when the number of activations in a layer is large. In terms of clustering based objectives, Coates et al. used K-means clustering and Gaussian mixtures to group the extracted image patches.

3 PyTorch - MNIST Handwritten Digit Recognition

3.1 Section 2 Techniques

3.1.1 Dropout and Noise

We added dropout to more layers of the model (compared to the baseline) and found that it sometimes helped and sometimes lead to worse performance. Ultimately though we chose to regularize our model primarily by adding Gaussian noise to the images as well as to intermediate layers of the model before applying convolutions. We found that adding Gaussian noise (mean 0; standard deviation 0.3) dramatically decreased overfitting and lead to higher accuracy on the validation set. When adding noise, especially to multiple layers of the model, validation accuracy was typically greater than training accuracy as no noise was added when evaluating on the validation and tests sets.

Our submission that achieved 0.9836 test accuracy on the leader-board (validation accuracy of 0.9833) was simply a convolutional model with added 0.3 Gaussian noise before all convolution layers.

3.1.2 Data Augmentation

We augmented the supervised data with small random rotations and translations of the original images. The procedure for augmentation involved rotating each image by a random number of degrees between -25 and 25 degrees (using scipy.ndimage.interpolation.rotate), cropping the image to the center to size 28x28 (rotating increases image size), padding the image of 3 pixels on all four sides, and then taking a random crop of 28x28 from the padded image. The augmentation procedure had different values for each image in the batch (so the rotation, cropping, and padding was randomly chosen for each image). This augmentation procedure was done online, so that each batch was slightly different.

Data augmentation significantly increased the performance of our models. Our submission that achieved 0.9918 test accuracy and 0.9920 validation accuracy was simply a supervised, data-augmented convolutional model with noise.

We also tried introducing training data augmented with elastic-distortions. Though training on the original data and elastic-distortion data helped improve accuracy with simpler models, it didn't bring up the accuracy in our final model and we didn't use it.

3.1.3 Batch Normalization

We found that adding batch normalization prior to activations in our model increased the performance significantly. Batch normalization also seemed to help with any drawbacks due to our relatively simple initialization technique of uniform distribution between -0.1 and 0.1 as described in Ioffe et al. [10].

3.1.4 Optimization

We found that switching from a vanilla SGD optimizer to an Adam optimizer, which utilizes momentum lead to much faster convergence. With Adam, for our noise-less models, the model would begin to overfit (training accuracy greater than validation accuracy) within a dozen epochs compared to a couple dozen with SGD. Additionally, we tried Nesterov optimization technique but it didn't yield faster convergence than Adam.

3.2 Semi-Supervised Techniques

3.2.1 Pseudo-Labeling

The first semi-supervised learning technique we tried was pseudo-labeling. It brought up our accuracy in earlier models where we were not utilizing noise. In our later, more complicated models, pseudo-labeling was no longer very helpful so we abandoned it.

3.2.2 Ladder Networks Attempt

We attempted to build a ladder network model in the vein of Pezeshki et al. [9] but were unable to get high performance with the standard MLP ladder networks model. The simple MLP model with layers of size 1000, 500, 250, 250, 250, 10 achieved a maximum validation accuracy of around 0.95. We think our model may have had errors in how the vector values were normalized along the "downward pass" of the ladder.

3.2.3 Gamma Model

In Rasmus et al. [8], they discuss a modified ladder networks model called the "gamma" model. It consists of a supervised loss as well as a single reconstruction loss at the last layer of the model, right before prediction, between clean and noisy versions of unlabeled data (described in more detail in the Final Model section).

For our first attempt at this model, we used a convolutional model and alternated between an epoch of supervised loss and then several steps of unsupervised loss. So weights were updated for different losses separately. This model achieved a test accuracy of 0.98820 and a validation accuracy of 0.9864. We found that adding the unsupervised loss increased performance, but alternating between the two losses didn't work as well. In our final model, we add these losses together and update the weights based on both losses simultaneously.

3.3 Final Model

For our final model we used a gamma convolutional model, which consisted of a supervised model and a de-noising semi-supervised model with weight sharing. Our convolution model was based on the Rasmus et al. [8] model and consisted of:

1. Convolutional layer: 32 filters, kernel size of 5 (no padding)
2. Maxpool layer: kernel size of 2
   - Batch normalization
   - Leaky ReLU (0.1) activation
3. Convolutional layer: 64 filters, kernel size of 3 (no padding)

- Batch normalization
- Leaky ReLU (0.1) activation

4. Convolutional layer: 64 filters, kernel size of 3 (no padding)
5. Maxpool layer: kernel size of 2
   - Batch normalization
   - Leaky ReLU (0.1) activation
6. Convolutional layer: 128 filters, kernel size of 3 (no padding)
   - Batch normalization
   - Leaky ReLU (0.1) activation
7. Fully Connected layer: 100 units
   - Batch normalization
   - ReLU Activation
   - Dropout with p=0.5
8. Fully Connected layer: 10 units
   - Batch normalization
9. Cross entropy loss (combines log-softmax and negative log-likelihood loss)

To utilize the unsupervised data we used the same supervised convolutional model architecture as a de-noising function with shared weights. Our denoising function consisted of running two copies of the model in parallel on the unlabeled data, one with noise and one without noise. The loss for this denoising function was the mean squared loss between the final output representations (before log-softmax function) of the noisy and clean models.

For the noisy version of the model, we added Gaussian noise (0 mean; 0.3 standard deviation) before applying each convolution layer (not before the pooling or fully connected layers). We also tried applying noise, small random translations, and rotations for "denoising", but found that just Gaussian noise worked best for the denoising function.

We also utilized the previously described augmentation technique on our supervised data. In addition, we initialized all our weights with a uniform random distribution between -0.1 and 0.1, except for the biases which were initialized to 0.

We used a batch size of 100, consisting of 50 labeled examples and 50 unlabeled examples. We ran the supervised model and the denoising function, added the losses together (1 to 1 weighting), and then updated the gradients.

For our final model, we trained our model for 500 epochs and chose the best performing model on the validation set (at 410 epochs; 99.44% on validation set). For optimization, we used the Adam optimizer with a learning rate of 0.001 and betas 0.9 and 0.999. We decayed our learning rate to 0.0005 after 100 epochs and decayed by 0.99 (scalar multiplication) for all epochs after 200. In addition, our batch normalization utilized a momentum of 0.1 (default implementation in pytorch).

4 Results

After running the model for 500 epochs, we achieved a final validation accuracy of 99.44% and a test accuracy of 99.48%. Figures 1 and 2 show the loss an accuracy respectively from the first 200 epochs of training. Our validation loss was significantly lower and more stable than the training loss as the training loss included the denoising function loss on the unlabeled data.

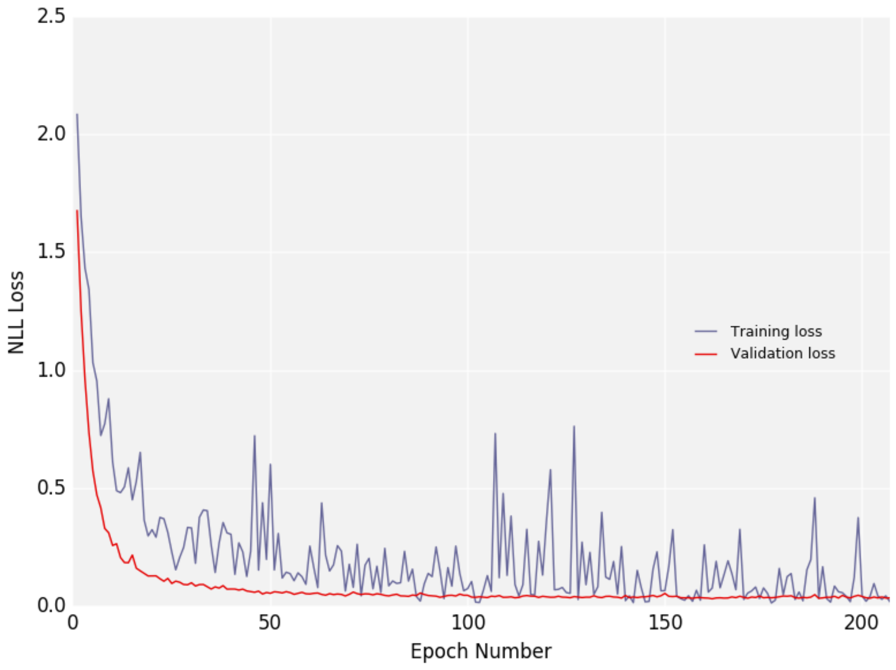

Figure 1: Plotting the train and validation loss for the first 200 epochs of training

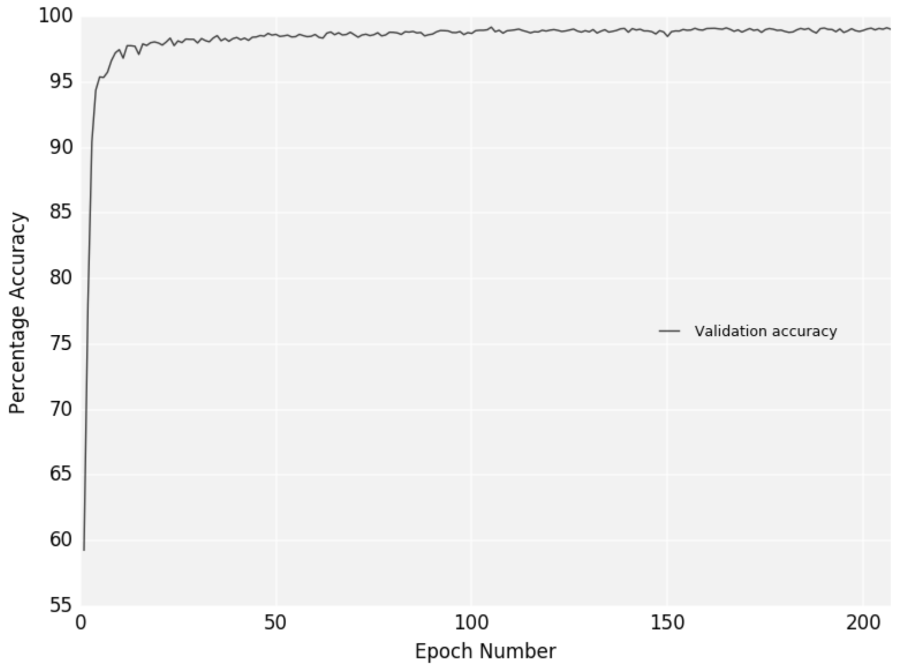

Figure 2: Plotting the validation accuracy for the first 200 epochs of training. The model was trained for another 300 epochs and the accuracy went up from 99.04% to 99.44%.

References

[1] Sutskever, I., Martens, J., Dahl, G. and Hinton, G: On the importance of initialization and momentum in deep learning, ICML (3) 28 (2013): 1139-1147.

[2] Srivastava, Nitish, Dropout: a simple way to prevent neural networks from overfitting., Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, Advances in Neural Information Processing Sys- tems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Pro- ceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, Delving Deep into Rec- tifiers: Surpassing Human-Level Performance on ImageNet Classification, CoRR, http://arxiv.org/abs/1502.01852, 2015.

[5] Xavier Glorot and Yoshua Bengio, Understanding the difficulty of training deep feedfor- ward neural networks, Proceedings of the Thirteenth International Conference on Artifi- cial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010.

[6] Karen Simonyan and Andrew Zisserman, Very Deep Convolutional Networks for Large- Scale Image Recognition, In ICLR, 2015.

[7] Adam Coates, Andrew Y. Ng and Honglak Lee, An Analysis of Single-Layer Networks in Unsupervised Feature Learning, Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011.

[8] Antti Rasmus, Harri Valpola, Mikko Honkala, Mathias Berglund, and Tapani Raiko. 2015. Semi-Supervised Learning with Ladder Networks. https://arxiv.org/abs/1507.02672

[9] Mohammad Pezeshki, Linxi Fan, Philémon Brakel, Aaron Courville, and Yoshua Bengio. 2016. Deconstructing the Ladder Network Architecture. https://arxiv.org/abs/1511.06430

[10] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. https://arxiv.org/pdf/1502.03167.pdf