Redundancy in Deep Linear Neural Networks

Oriel BenShmuel
Faculty of Math&CS
Weizmann Institute of Science
Israel
oriel.benshmuel@weizmann.ac.il

Abstract

Conventional wisdom states that deep linear neural networks benefit from expressiveness and optimization advantages over a single linear layer. This paper suggests that, in practice, the training process of deep linear fully-connected networks using conventional optimizers is convex in the same manner as a single linear fully-connected layer. This paper aims to explain this claim and demonstrate it. Even though convolutional networks are not aligned with this description, this work aims to attain a new conceptual understanding of fully-connected linear networks that might shed light on the possible constraints of convolutional settings and non-linear architectures.

1 Introduction

Linear networks provide a simple and elegant solution for elementary problems such as simple classification [6, 29] and regression problems [28, 8]. Moreover, linear models are suitable for approximate higher-level representation of the dataset distribution using AutoEncoders [3, 27, 13, 16]. They could serve as classic denoising solvers [18] or even datasets' manifold estimators [11]. However, linear models perform poorly on complex problems expressed with nonlinear properties [17, 15, 19].

On the other hand, deep neural networks with nonlinear abilities are an extensively researched field. These nonlinear models reached state-of-the-art performance on computer vision problems [26, 20, 10, 21, 7] and natural language processing tasks [25, 5, 2, 4]. Yet, the clarity of the learning process and the direct influence of the samples on the network remained quite vague. Therefore, deep linear networks provide a convenient framework to understand part of the bigger picture of deep learning.

Recent findings in theoretical deep-learning show that despite the linear mapping between the input and the output of deep linear networks, they still have optimization advantages over a single linear layer network [1, 9, 12, 22]. These findings contain an elaborated explanation for this claim using theoretical analysis and experimental demonstrations.

According to the theoretical analysis, deep linear networks have a non-convex optimization process. This paper demonstrates how this perception might be different for fully-connected linear networks. Deep fully-connected linear networks could perform a non-convex (and non-concave) optimization process. However, in practice, these networks are, indeed, going through a convex optimization process that is experimentally equivalent to a single fully-connected linear layer network.

We begin in section 2 by exposing some properties of fully connected linear networks trained with stochastic gradient descent (SGD [23]) and experimentally support our claims in section 3. Then, in section 4, we show how these properties lead to an equivalent optimization process of a deep linear network and a single linear layer (with fully-connected architectures). Finally, in section 5, we explain why SGD with momentum [24] also performs an equivalent optimization process.

Preprint. Under review.

2 Proportions in fully-connected linear networks

This section will expose some properties regarding the weights of fully-connected linear networks (experimentally supported by section 3). We will use the following notations:

1. θ_l of size $k_l \times n_l$ - The weights of layer l in the network.
2. $\theta_l[j]$ - The jth row of θ_l.
3. $\partial\theta$ - The changes of the weights in the first step with respect to the gradient of the loss function.

For a network with a single linear layer, we will define $k_1 = 2$ (for two classes). Without loss of generality, we will focus on the randomly initialized weights $\theta_1[0]$ (and not $\theta_1[1]$) as the vector of weights related to the first output neuron. For an input space of size n, the size of vector $\theta_1[0]$ is $1 \times n$.

Claim 1. Let x be a single training sample used for training a network with a single linear layer for a single step. Then there is a scalar $\alpha \in \mathbb{R}$ such that:

$$\partial\theta_1[0] = \alpha x$$

Claim 2. Given a network with a single linear layer with randomly initialized weights θ_1 and a set $\{(x_i, \alpha_i)\}_{i=1}^{b}$ such that each pair corresponds to the proportional property described in *Claim 1* with respect to θ_1. Training the network with the entire batch $\{x_i\}_{i=1}^{b}$ for a single step (with the same initial weights θ_1) will result in the following equality:

$$\partial\theta_1[0] = \frac{1}{b}\sum_{i=1}^{b} \alpha_i x_i$$

Claim 3. For a deep linear network, the following statement is applied (using the previous notations):

$$\forall_{0\leq j<k_1} \exists r_j \in \mathbb{R} : \partial\theta_1[j] = r_j \frac{1}{b}\sum_{i=1}^{b} \alpha_i x_i$$

Corollary 1. For a deep linear network:

$$\forall_{0\leq j_1,j_2<k_1} \exists r_{j_1}, r_{j_2} \in \mathbb{R} : r_{j_2} \cdot \partial\theta_1[j_1] = r_{j_1} \cdot \partial\theta_1[j_2]$$

Claim 4. For a deep linear network, we get the following for any layer l in the network and any step in the training process:

$$\forall_{0<l<depth, 0\leq j_1,j_2<k_l} \exists r \in \mathbb{R} : \partial\theta_l[j_1] = r \cdot \partial\theta_l[j_2]$$

3 Experimental support

To measure how close the vectors are, in terms of proportionally, for each angle $a > 90°$ in the experiments, we use $180° - a$ instead. In addition, we are using the negative log likelihood loss function (a non-linear function). The experiments were conducted with GPU K80.

Claim 1 (support). For a single linear layer, we will use the classes *Cat* and *Dog* of the dataset CIFAR10 [14]. For ten different initializations of a single linear layer, we randomly pick 100 different samples. We compute the angle between $\partial\theta_1[0]$ and the chosen sample. The mean value of the calculated angles is $0.01°$, and the standard deviation is $0.005°$.

As expected, each angle is very close to $0°$ or to $180°$ (up to numerical errors), which indicates that $\partial\theta_1[0] = \alpha x$, for some sample x and scalar $\alpha \in \mathbb{R}$.

Claim 2 (support). For a batch size of 30 and a single linear network with a single layer, we get that the expression $||\partial\theta_1[0] - \frac{1}{b}\sum_{i=1}^{b}\alpha_i x_i||$ over ten initializations produces an average value of $3.62 \cdot 10^{-8}$. It implies that the equation $\partial\theta_1[0] = \frac{1}{b}\sum_{i=1}^{b}\alpha_i x_i$ is indeed true up to numerical errors.

Claim 3 (support). In the same manner, for multiple layers in the network (using "Architecture B" appears in Appendix B) and a batch size of 30, consider all possible combinations of $\partial\theta_1[j_1]$ and $\partial\theta_1[j_2]$ for $0 \leq j_1 \neq j2 < k_1$. We get that for the angles above $90°$, we have an average angle of $179.99°$, and for the angles below $90°$, the average angle is $0.001°$ which supports the fact that:

$$\forall_{0\leq j_1,j_2<k_1}\exists r \in \mathbb{R} : \partial\theta_1[j_1] = r \cdot \partial\theta_1[j_2]$$

Claim 4 (support). For long-term training of linear network with multiple layers, the following experiments will include the average angle between the vectors $\{\partial\theta_l^t[j]\}_{0\leq j<k_l}$ for each layer l as a function of step number t. Each analysis will also include a graph of the model's accuracy over the training steps.

We will use several linear architectures for the experiments. The architectures appear in Appendix B.

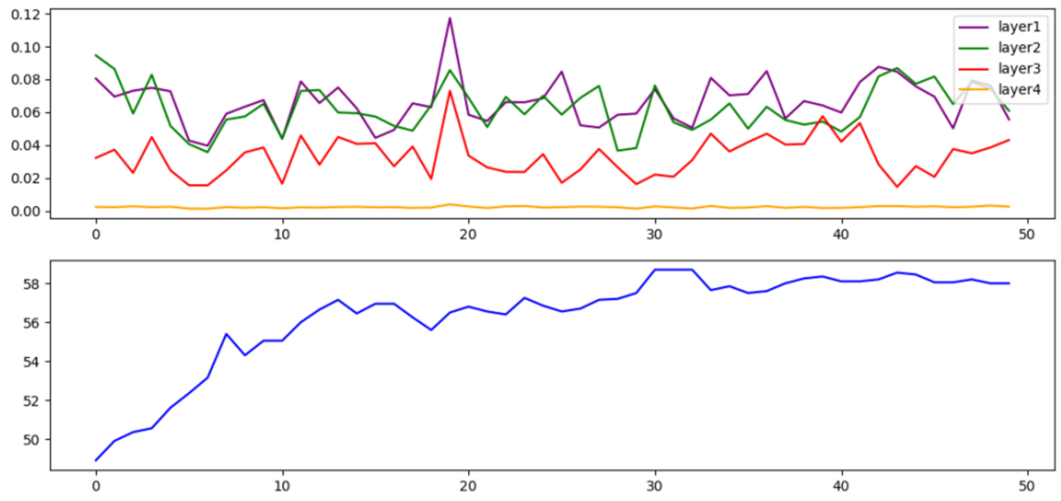

Figure 1: The top graph describes the average angle between $\partial\theta_l^t[j_1]$ and $\partial\theta_l^t[j_2]$ for each layer l as a function of steps t. The bottom graph describes the model's accuracy.

In Figure 1, we can see the analysis for *Cat* versus *Dog* trained with a batch size of 128, a learning rate of $1e-2$, and "Architecture B" over 50 steps. The first image shows the average angle between each pair of vectors $(\partial\theta_l^t[j_1], \partial\theta_l^t[j_2]$ for $0 \leq j_1 \neq j_2 < k_l)$ as a function of the iteration t (the step number). There are four plots in the first image, each plot for each one of the four linear layers in the network. The presented angles are in *degrees*, and it is easy to spot that (up to a complement of $180°$) the angle is below a single degree (which is extremely small). The second graph (below the first one) shows the accuracy of the same network. It implies that the angles are independent of the accuracy, and up to minor errors, they have very small values in each phase of the training. Overall it supports the claim that for any given step t and layer l, the following expression is true:

$$\forall_{0\leq j_1,j_2<k_l}\exists_{r\in\mathbb{R}} : \partial\theta_l^t[j_1] \approx r \cdot \partial\theta_l^t[j_2]$$

An additional set of experiments with multiple different settings appears in Appendix A.

Note: When calculating the angles, we normalize the vectors. For vectors with small norms in the first place, a numerical error might occur during the normalization. Therefore, wider layers might have more significant errors. Overall the error is tiny (the vertical scale of the first plot is in degrees) and usually below $0.5°$.

4 The optimization process

Following the above claims (section 2), for each layer l, the update of the weights $\partial\theta_l$ could be observed as a collection of the vectors $\{t_j \cdot \partial\theta_l[0]\}_{0\leq j<k}$ (for some set of scalars $t_j \in \mathbb{R}$) rather than

a collection of the vectors $\{\partial\theta_l[j]\}_{0\leq j<k}$. In this case, the entire matrix $\partial\theta_l$ depends on a single vector $\theta_l[0]$ up to scalar multiplication. Additionally, in the classic training process, where we use stochastic gradient steps (SGD), this is true for any given step in the training process (and not only for the first step). In other words, we get that each layer is updated with a weights matrix of rank 1. Their multiplication does not reduce the expressiveness of the network.

In general, we can apply a simple reduction from the optimization process of fully-connected deep linear networks to the optimization process of a single fully-connected layer, as illustrated in Figure 2. Assuming that:

$$\forall_{0\leq j_1,j_2<k_l,0<l<depth}\exists_{r\in\mathbb{R}} : \partial\theta_l[j_1] = r\cdot\partial\theta_l[j_2]$$

we can represent $\partial\theta_l$ as the collection of the vectors $\{t_j\cdot\partial\theta_l[0]\}_{0\leq j<k_l}$. Since $\partial\theta_l$ depends on a single vector, up to scalar multiplication, the same update in the weights could have been done with a single neuron (which also depends on a single vector of weights of the same size). Extending this conclusion for each step and layer l in the original network (excluding the output layer), we get a similar optimization process of a fully-connected linear network with only a single neuron for each hidden layer. In the case of a single neuron in each hidden layer, we get a network that is not using its depth since the weights of each hidden layer is a scalar. Therefore, the network could be represented as a single linear layer in the training process as an equivalent learner to any deep fully-connected linear architecture.

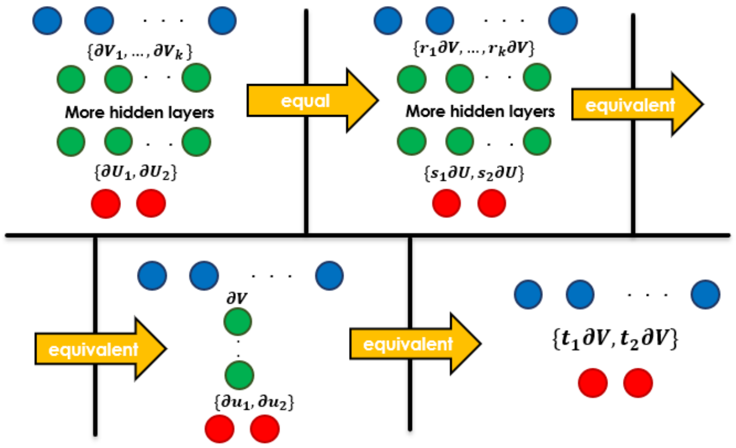
$\{\partial V_1, \ldots, \partial V_k\}$
More hidden layers
$\{\partial U_1, \partial U_2\}$
equal
$\{r_1\partial V, \ldots, r_k\partial V\}$
More hidden layers
$\{s_1\partial U, s_2\partial U\}$
equivalent
equivalent
∂V
$\{\partial u_1, \partial u_2\}$
equivalent
$\{t_1\partial V, t_2\partial V\}$

Figure 2: Visualization of the weights' update's equivalency transitions from a deep linear network to a single linear layer.

This section concludes that the training process of a randomly initialized fully-connected deep linear network is experimentally equivalent to a randomly initialized linear network with a single layer.

5 SGD with Momentum

In many cases, the SGD optimizer is used with momentum [24] to acquire convergence advantages. For such an optimizer, the optimization process enjoys memorization abilities based on the weights update of previous steps. The momentum algorithm is conducted as follows:

$$\mathcal{V}^{t+1} = \beta\cdot\mathcal{V}^t + (1-\beta)\cdot\partial\theta^t$$

$$\theta^{t+1} = \theta^t - \alpha\cdot\mathcal{V}^{t+1}$$

Where t is the iteration index, α is the learning rate, and β is the momentum factor. For an iterative representation of $\mathcal{V}^n$, we get:

$$\mathcal{V}^n = (1-\beta)\cdot\sum_{t=1}^{n}\beta^{n-t}\partial\theta^t$$

Assume a new optimization method with a predefined number of steps (in our case, n steps), using the traditional SGD with a gentle twist:

$$\gamma_t = (1-\beta) \cdot \sum_{i=t}^{n} \beta^{i-t}$$

$$\theta^{t+1} = \theta^t - \alpha \cdot \gamma_t \cdot \theta^t$$

In the new variant, θ^t is an iteration-dependent scalar. Proportional vectors would be proportional even after scalar multiplications. Therefore, the properties mentioned in section 2 and supported by section 3 are applied to the new variant of SGD defined above. Moreover, we get the following equality:

$$\sum_{s=1}^{n} \mathcal{V}^s = \sum_{s=1}^{n} \sum_{t=1}^{s} (1-\beta)\beta^{s-t} \partial\theta^t =$$

$$(1-\beta)\bigg(\partial\theta^1(\beta^0 + ... + \beta^{n-1}) + \partial\theta^2(\beta^0 + ... + \beta^{n-2}) + ... + \theta^n(\beta^0)\bigg) =$$

$$= (1-\beta)\bigg(\partial\theta^1 \sum_{i=0}^{n-1} \beta^i + \partial\theta^2 \sum_{i=0}^{n-2} \beta^i + ... + \partial\theta^n \sum_{i=0}^{0} \beta^i\bigg) =$$

$$= (1-\beta)\bigg(\partial\theta^1 \sum_{i=1}^{n} \beta^{i-1} + \partial\theta^2 \sum_{i=2}^{n} \beta^{i-2} + ... + \partial\theta^n \sum_{i=n}^{n} \beta^{i-n}\bigg) =$$

$$= \sum_{s=1}^{n} \bigg((1-\beta) \sum_{i=s}^{n} \beta^{i-s}\bigg) \partial\theta^s = \sum_{s=1}^{n} \gamma_s \theta^s$$

The above equality shows that taking n steps using SGD with momentum produces the same state as the proposed optimization method. Additionally, both methods (SGD with momentum and the new method defined above) use the same matrices $\theta_i{}_{i=1}^{n}$ to reach that state (up to scalar multiplications). It implies similar expressive abilities, and therefore both processes are equivalent in that term.

Overall, SGD with momentum could be represented as the new variant of SGD we proposed. As explained earlier in this section, the new variant of SGD has an equivalent optimization process to a single layer. Therefore, the optimization process of SGD with momentum has an equivalent optimization process to a single fully-connected layer.

6 Summary and open problems

We experimentally demonstrated how the derivatives of the weights $\partial\theta_l^t[j]$ (represented as vectors) are proportional for the same iteration and the same layer when training a deep fully-connected network with conventional optimizers. Provided with this experimental outcome, we concluded that a deep fully-connected network trained with conventional optimizers has an equivalent optimization process to a single fully-connected layer.

Our experiments are valid solely for fully-connected networks. Intuitively, the proportion between two vectors of weights is due to the mutual input layer observations during the training. Even though convolutional layers observe local regions individually (rather than the entire input simultaneously), there are still dependencies between close areas in the image and different filters concerning the same region. Therefore, deep linear convolutional neural networks probably do not have a convex optimization process; however, these models still might demonstrate interesting optimization constraints. Testing this hypothesis in future work might equip us with more knowledge regarding linear networks in particular and convolutional behavior in general.

References

[1] S. Arora, N. Cohen, and E. Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization, 2018.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4):291–294, 1988.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[8] B. Hanin and Y. Sun. How data augmentation affects optimization for linear regression. *Advances in Neural Information Processing Systems*, 34, 2021.

[9] M. Hardt and T. Ma. Identity matters in deep learning, 2018.

[10] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[11] G. E. Hinton, P. Dayan, and M. Revow. Modeling the manifolds of images of handwritten digits. *IEEE transactions on Neural Networks*, 8(1):65–74, 1997.

[12] K. Kawaguchi. Deep learning without poor local minima, 2016.

[13] E. Kodirov, T. Xiang, and S. Gong. Semantic autoencoder for zero-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3174–3183, 2017.

[14] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[15] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. *Advances in neural information processing systems*, 30, 2017.

[16] Q. Meng, D. Catchpoole, D. Skillicom, and P. J. Kennedy. Relational autoencoder for feature extraction. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 364–371. IEEE, 2017.

[17] M. J. Murphy and S. Dieterich. Comparative performance of linear and nonlinear neural networks to predict irregular breathing. *Physics in Medicine & Biology*, 51(22):5903, 2006.

[18] A. Pretorius, S. Kroon, and H. Kamper. Learning dynamics of linear denoising autoencoders. In *International Conference on Machine Learning*, pages 4141–4150. PMLR, 2018.

[19] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. On the expressive power of deep neural networks. In *international conference on machine learning*, pages 2847–2854. PMLR, 2017.

[20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[21] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[22] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, 2014.

[23] J. C. Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control*, volume 65. John Wiley & Sons, 2005.

[24] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 2013. PMLR.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[26] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[27] W. Wang, Y. Huang, Y. Wang, and L. Wang. Generalized autoencoder: A neural network framework for dimensionality reduction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 496–503, 2014.

[28] S. Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.

[29] T. Zhang and F. J. Oles. Text categorization based on regularized linear classification methods. *Information retrieval*, 4(1):5–31, 2001.

A Additional experiments

In Figures 3,4,5,6,7, we can see the graph analysis of various cases with various learning rates, batch sizes, architectures (see Appendix B), categories (of CIFAR10), and training resolutions.

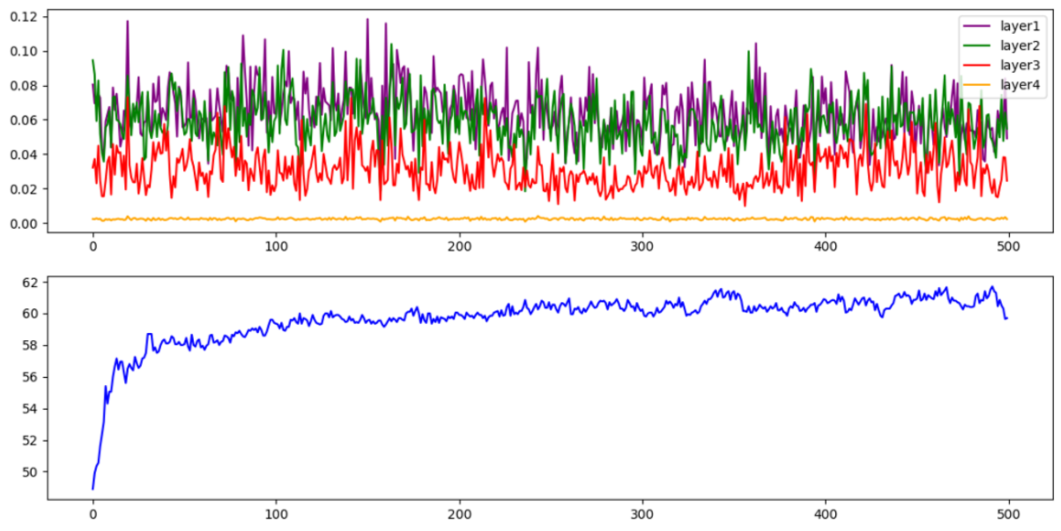

Figure 3: *Cat* versus *Dog* - a batch size of 128, a learning rate of $1e-2$, and Architecture B trained for 500 steps.

B Architectures

Architecture A: Two linear layers.

- **Fully-connected layer** [input: 3072, output: 128]
- **Fully-connected layer** [input: 128, output: 2]

Architecture B: Four linear layers.

- **Fully-connected layer** [input: 3072, output: 2048]
- **Fully-connected layer** [input: 2048, output: 1024]
- **Fully-connected layer** [input: 1024, output: 128]
- **Fully-connected layer** [input: 128, output: 2]

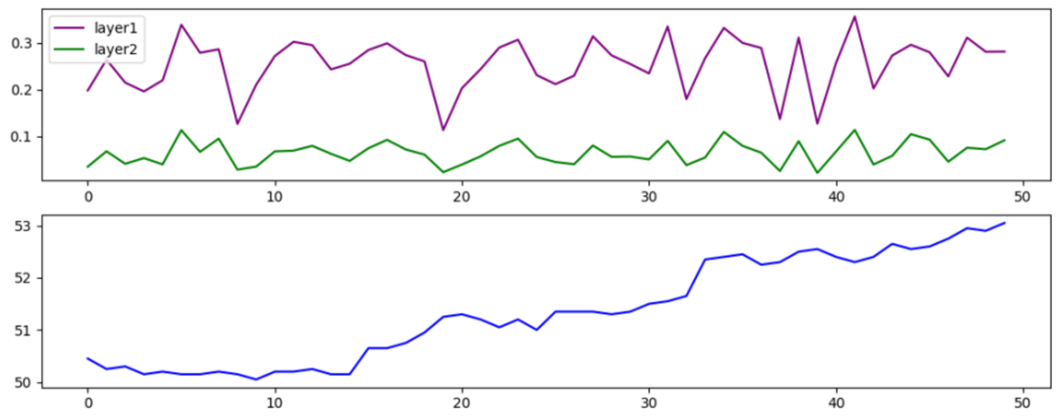

Figure 4: *Ship* versus *Truck* - a batch size of 256, a learning rate of $1e - 4$, and Architecture A trained for 50 steps.

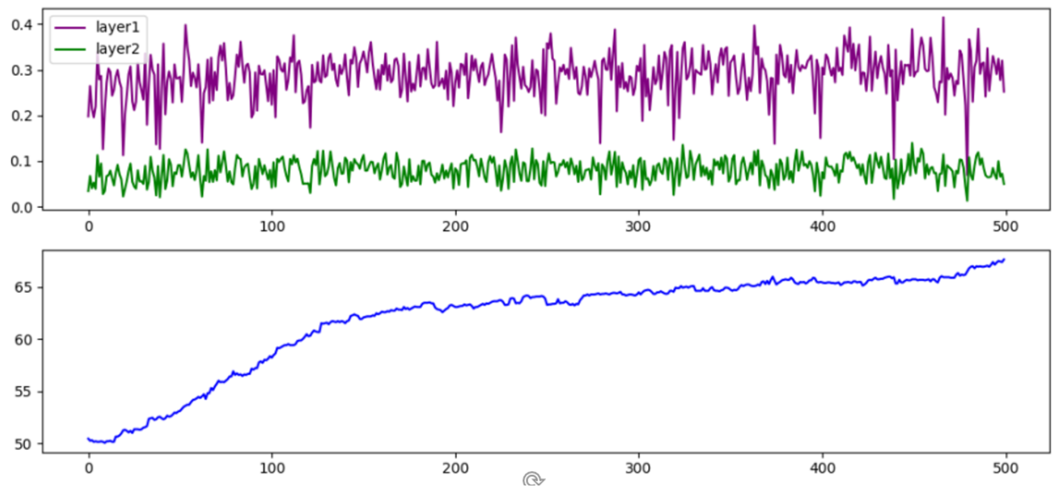

Figure 5: *Ship* versus *Truck* - a batch size of 256, a learning rate of $1e - 4$, and Architecture A trained for 500 steps.

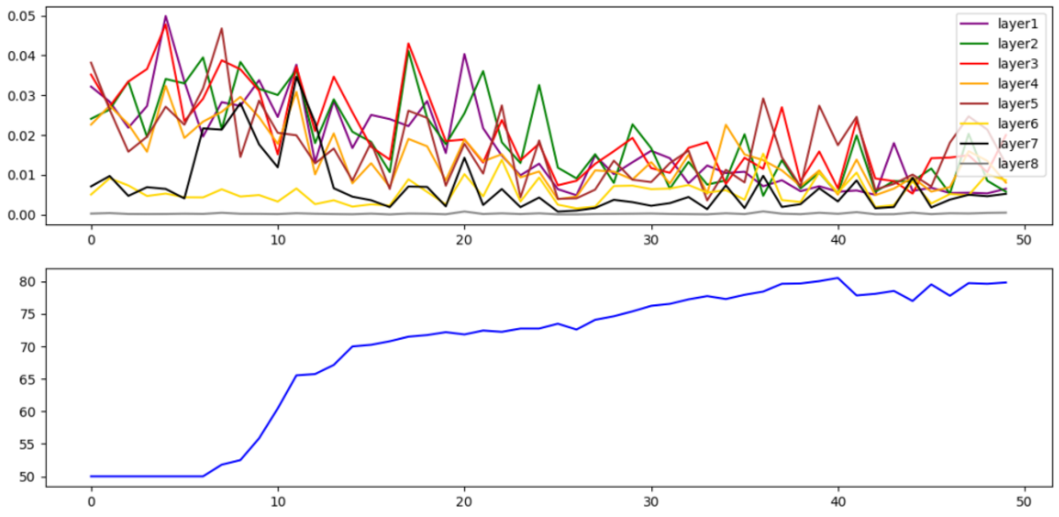

Figure 6: *Airplane* versus *Automobile* - a batch size of 64, a learning rate of $1e - 1$, and Architecture C trained for 50 steps.

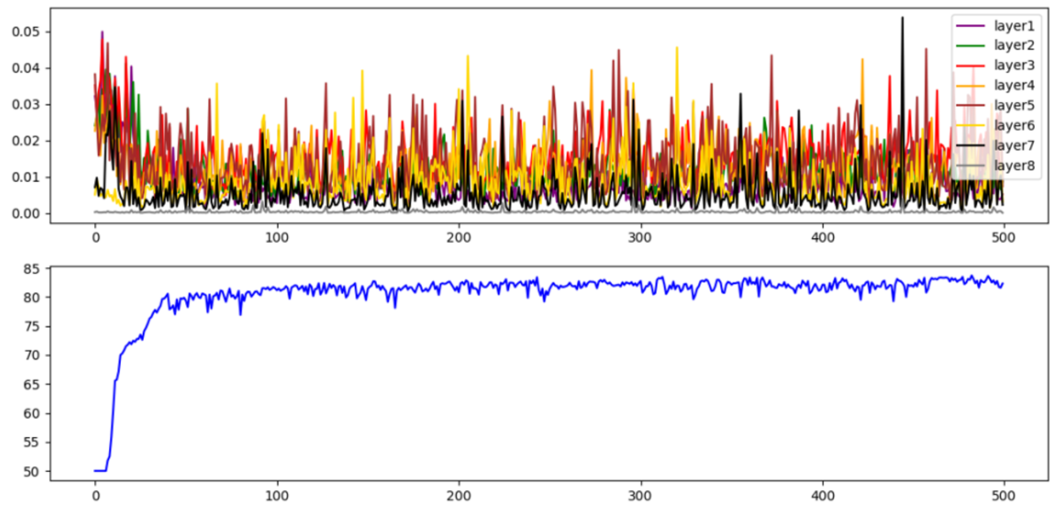

Figure 7: *Airplane* versus *Automobile* - a batch size of 64, a learning rate of $1e-1$, and Architecture C trained for 500 steps.

Architecture C: Eight linear layers.

- **Fully-connected layer** [input: 3072, output: 2500]
- **Fully-connected layer** [input: 2500, output: 1500]
- **Fully-connected layer** [input: 1500, output: 1024]
- **Fully-connected layer** [input: 1024, output: 512]
- **Fully-connected layer** [input: 512, output: 256]
- **Fully-connected layer** [input: 256, output: 64]
- **Fully-connected layer** [input: 64, output: 16]
- **Fully-connected layer** [input: 16, output: 2]