Self-supervised learning with autoencoders

Jia Geng Chang

December 2021

This is a classical autoencoder for intended for CPU training, that is written in optimized code using C.

1 Encoding a 8-bit one-hot vector with 3 bits

I trained a multi-layer perceptron to encode with 3 units a one-hot encoded vector of length 8. e.g. $\begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{pmatrix}$ represents the decimal digit 7 and $\begin{pmatrix}0 & 0 & 0 & 4 & 0 & 0 & 0 & 0\end{pmatrix}$ represents the decimal digit 4. Because only one value can be 1, there are only 8 such vectors corresponding to the rows of a 8x8 identity matrix — a relatively easy pattern for the network to learn to encode and decode.

Here are the hyperparameters of the autoencoder:

activation function	sigmoid	$\sigma(x) = \frac{1}{1+exp(-x}$
loss function	binary cross-entropy	*equation below
optimizer	classical momentum	m=0-0.9
learning rate	min 0.01	max 1.0
weights	random uniform initialization	Between -0.1 and 0.1

Table 1: Network architecture

Binary cross-entropy loss is:

$$-\frac{1}{N}\sum_{k=1}^{N} t_k\, log(p_k) + (1-t_k))\,(log(1-p_k)) \tag{1}$$

where $\mathbf{t}$ is the target one-hot encoded vector, and $\mathbf{p}$ is the output layer activation which is a probability vector, and N is 8 in our case. The derivatives of loss with respect to weights worked out to be:

$$\frac{\partial E}{\partial W_{jk}} = -\frac{1}{N}\,(\frac{t_k}{z_k} - \frac{1-t_k}{1-z_k})\; g'(x_k)\; z_j = \delta_k z_j \tag{2}$$

for the weights between hidden and output units and

$$\frac{\partial E}{\partial W_{ij}} = -\frac{1}{N}\sum_{k=1}^{N}(\frac{t_k}{z_k} - \frac{1-t_k}{1-z_k})\; g'(x_k)\; w_{jk}\; g'(x_j)\; z_i = \delta_j z_i \tag{3}$$

for the weights between input and hidden units. The derivative g' of the sigmoid function g is $g' = g(1-g)$.

2 Training results

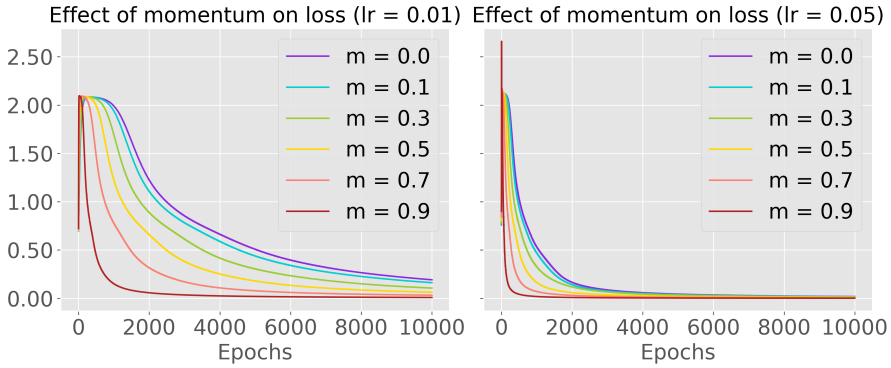

Figure 1: With learning rate held constant, momentum speeds up learning

With standard gradient descent, training was slow, even at high learning rates of up to 10. Thus, I added a momentum term involving the previous time step as a modification to vanilla gradient descent. The weight update heuristic at epoch t becomes:

$$\Delta W_{ij}(t) = -\eta \frac{\partial E}{\partial W_{ij}} + m\,\Delta W_{ij}(t-1) \tag{4}$$

for a pair of connected neurons i and j, and m is scalar that practitioners suggest to be kept between 0 and 1. The result in training with varying m, for two different learning rates (lr $= 0.01$ and lr $= 0.05$) is shown in figure 1.

From 1, we can see that adding momentum speeds up training greatly. What is happening is that the magnitude of $\Delta W_{ij}(t)$ is increasing by a factor of close to $1 + m$ as a function of t, because each time $\nabla E(t)$ is in a similar direction as $\nabla E(t-1)$.

3 How 8-3-8 autoencoder learns input representation

How is the network learning to encode input vectors? We look at the final state of the weights for the best network to see what is happening.

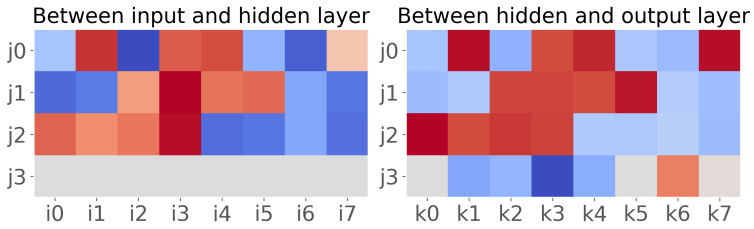

Figure 2: The network learns to mirror weights between the three layers

Without loss of generality, what the network has learned is to match the signs between mirrored pairs of weights $W_{nj},\ W_{jn}\ \forall\ n \in [0,7],\ j \in [0,2]$. For example, if $\mathbf{t}$ is $\begin{pmatrix}1 & 0 & \dots & 0\end{pmatrix}$,

then only j_0 and k_0 will fire, producing a large positive activation in k_0 because $W_{i0,j0}$ and $W_{j0,k0}$ have the same sign (does not matter whether positive or negative). The two other hidden units also have synergistic weights entering/leaving them, but they will stay silent because the inputs into those units is 0.

4 Encoding a 16-bit one-hot vector with 3 bits

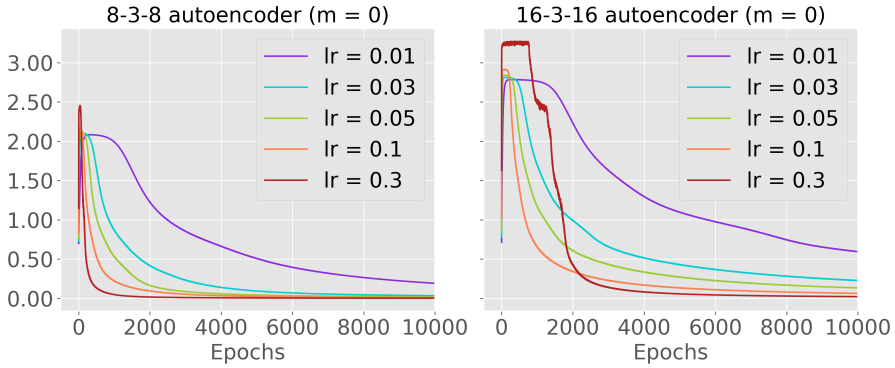

Figure 3: The 16-3-16 multi-layer perceptron trains slower across learning rates. Vertical axis is binary cross-entropy loss.

Is it possible to be even more efficient and encode a one-hot vector *twice* the size as our previous vector, with 3 hidden units? If each hidden unit can represent 3 activation states, then we can encode up to a one-hot vector of length $3^n = 3^3 = 27$. As we are using a sigmoid activation, the three activation states are naturally 0, 0.5, and 1, requiring asymptotic input values of $-\infty$, 0, and ∞ at each hidden unit. While our loss will again never reach 0, it is theoretically possible to train such a network.

In figure 3, I trained a multi-layer perceptron to encode with 3 units a one-hot encoded vector of length **16**. Training is slower, because there are more weights to adjust, but binary cross-entropy is reduced to a similar extent after around 10000 epochs.

5 How 16-3-16 autoencoder learns input representation

Visualizing the weight matrices in figure 4, we see that the network is again solving the problem by mirroring the sign of the weights. Additionally, it mirrors the relative magnitude of the weights. This time, each hidden node has three states, as the mirrored incoming and outgoing weights are in one of negative (blue), zero (off-white), and positive (red). The connections entering output node 14 (k14) look all negative, but this is offset by the positive bias ($W_{j3,k14}$). Again, the bias in the input layer is not doing much (it cannot be any other value than 0 otherwise non-1 units will fire), but the bias in the hidden layer is helping to adjust the median weight to zero.

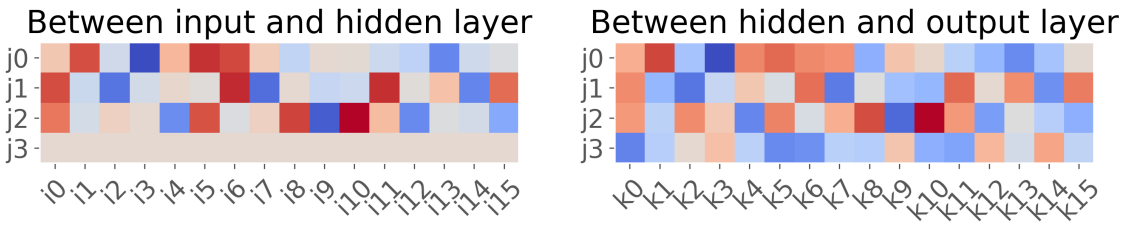

Figure 4: Weights are again mirrored, and every hidden unit has 3 states - red, white and blue

6 Code availability

The code for the 8-unit multi-layer perceptron is available as `testmlp.c`, and the code for the 16-unit multi-layer perceptron is found in `testmlp16.c`. I also include a `utils.c` definitions file for simple matrix operations required (e.g., dot product, hadamard product, and scalar operations). The `output` directory contains the `.csv` files used to generate the plots. Jupyter notebooks for generating the plots are also included as `.ipynb` files, and the `.png` files are found in `plots`.