Sparse Combinatorial Autoencoders

Karthik S. Narayan, Pieter Abbeel
Electrical Engineering and Computer Science
University of California, Berkeley Berkeley, CA 94709
karthik.narayan@eecs.berkeley.edu, pabbeel@cs.berkeley.edu

Abstract

Recent research has shown that employing unsupervised pretraining often produces well-conditioned neural network initializations that lead to better local optima attained during training. One commonly used pretraining method involves hierarchically stacking sparse autoencoders (SAs) and learning the network parameters layer by layer using unlabeled data. Large network sizes and the amount of data required to properly pretrain a deep network make pretraining computationally intensive and the training bottleneck. To alleviate this problem, we propose a novel warm-start procedure for the SA capable of rapidly initializing large SAs in parameter regions yielding fast convergence to good local optima. At the heart of our approach lies the sparse combinatorial autoencoder (SCA), a novel method to regularize neural networks that allows us to train an SA with \(H\) features in \(O(\sqrt{H})\) time. We present a comprehensive series of experiments demonstrating the effectiveness of the warm-start procedure, called fast initialization with SCA (FISCA), on the STL-10 and the MNIST datasets. Our experiments consider untied sigmoid and tied soft-rectified SAs of various sizes and demonstrate that FISCA ultimately yields significantly reduced training times compared to widely prevalent initialization techniques. For example, on the MNIST dataset, FISCA-initialized soft-rectified SAs with \(10^K\) hidden neurons converge over \(20\times\) faster to notably better local optima than SAs initialized with alternate methods.

1 Introduction

Unsupervised feature learning and deep learning have recently become topics of interest in the machine learning community. Rather than requiring hand-crafted domain-specific features, deep learning algorithms provide an automatic method to learn deep hierarchies of features from raw data, which can be used in tasks such as classification and unsupervised modeling. See [1] for an overview of commonly used deep learning algorithms.

Deep neural networks have presented highly promising results in vision, natural language, and audio algorithms, often outperforming algorithms using manually crafted features [2, 3, 4, 5]. Many such deep learning pipelines attain large performance boosts through greedy layer-wise unsupervised pretraining before supervised learning [6, 7], e.g., via autoencoder-based methods [6, 8, 9]. Pretraining the first layer typically involves computing an overcomplete basis for the data, allowing each instance to be represented as a sparse combination of basis vectors. Pretraining higher layers entails learning more overcomplete bases, treating the coefficients computed in the previous layer as the data representation. Several papers demonstrate that inputs which maximally activate single neurons in higher layers after pretraining correspond to hierarchically learned features (e.g., edges, object parts, and full objects in vision), which could lead to better performance in classification tasks [7, 10]. Unfortunately, the amount of data required to properly pretrain a large deep network makes pretraining highly expensive. For example, the authors in [11] use 57 million \(32 \times 32\) image patches.
Attempts to reduce deep network training times include training networks on GPUs \([2, 3, 12]\), using second-order optimization methods \([13, 14, 15]\), and careful network initialization \([16]\). Weight initialization plays a particularly important role in training neural networks due to their nonconvex objectives. Prominent initialization techniques include Gaussian randomization (GR, \(W \sim N(0, \epsilon^2)\) for small \(\epsilon\), say 0.1 or 0.01), uniform root randomization (URR, \(W \sim U[-1/\sqrt{n}, 1/\sqrt{n}]\) where \(n\) is the number of hidden nodes in the previous layer), and normalized initialization (NI, a variant of URR) \([16]\). These approaches tend to yield convergence to good local optima, albeit slowly.

**Contributions.** We propose a novel warm-start algorithm for pretraining capable of rapidly initializing large sparse autoencoders (SA) in parameter regions ultimately yielding fast convergence to good local optima. At the heart of our approaches lies the *sparse combinatorial autoencoder* (SCA), a novel method to regularize neural networks, ultimately allowing us to train an SA’s \(H\) features in \(O(\sqrt{H})\) time. After arguing that the SCA approximately solves the SA optimization problem in a low-dimensional space, we (1) derive efficient gradient descent algorithms for the SCA and (2) propose a method to transfer learned SCA weights to SA weights. Experiments on the MNIST \([17]\) and STL-10 \([18]\) datasets demonstrate that this two-step procedure, called fast initialization with SCAs (FISCA), quickly yields SA warm-starts with much lower starting objectives compared to GR, URR, and NI, the most commonly used initialization techniques, leading to significantly reduced SA training times. For large autoencoders, FISCA often runs in time lesser than 30 autoencoder minibatch gradient descent iterations; even with this overhead, FISCA-initialized SAs often converge more than \(3-20\times\) faster than SAs initialized with other methods. Before discussing FISCA, we first briefly review the sparse autoencoder.

### 2 Sparse Autoencoders

A sparse autoencoder (SA) \([8]\) consists of an encoder and a decoder. The encoder maps an input \(x \in \mathbb{R}^n\) to some hidden representation \(o_s(x) \in \mathbb{R}^{|o_s|}\) (see Figure 1b): \(o_s(x) = g_1(V_s x + b_s)\), where \(g_1\) is an activation function, such as the sigmoid \(g_1(x) = (1 + \exp(x))^{-1}\) or soft-rectifier \(g_1(x) = \log(1 + \exp(x))\). The matrix \(V_s \in \mathbb{R}^{|o_s| \times n}\) and vector \(b_s \in \mathbb{R}^{|o_s|}\) fully parameterize the encoder.

The decoder attempts to reconstruct the input \(x\) by operating on the hidden representation \(o_s\); \(\hat{x}(x) = g_2(W_s o_s(x) + b'_s)\) where \(g_2\) is another activation function, which we set to the identity. Similarly, the matrix \(W_s \in \mathbb{R}^{|o_s| \times \rho}\) and vector \(b'_s \in \mathbb{R}^{|o_s|}\) fully parameterize the decoder. Training the sparse autoencoder involves optimizing the objective \(J_{SA}(\theta)\) where \(\theta = \{V_s, W_s, b_s, b'_s\}\):

\[
J_{SA}(\theta) = \frac{1}{m} \sum_{i=1}^{m} [L(x^{(i)}, \hat{x}^{(i)})] + \frac{\lambda}{2} \left(\|V_s\|_F^2 + \|W_s\|_F^2\right) + \frac{\beta}{|o_s|} \sum_{h \in o_s} \frac{1}{m} \sum_{i=1}^{m} o^{(h)}_s(x^{(i)}) ; \rho
\]

where \(x^{(i)}\) denotes the \(i\)th training example in dataset \(X\), \(m\) denotes the number of training examples in \(X\), \(L\) is some loss function (we use \(L(x, y) = \|x - y\|_2^2\), \(f\) denotes a sparsity penalty (e.g., \(\ell_1\) penalty: \(f(x; \rho) = |x - \rho|\), \(\|\cdot\|_F\) denotes the Frobenius norm, and \(\beta, \lambda, \rho\) are hyperparameters which control the sparsity penalty, regularization strength, and target neuron activation, respectively. Our experiments consider two popular types of SAs: (i) untied sigmoid-activated SAs with \(\ell_2\)-loss and Bernoulli KL-divergence sparsity penalty and (ii) tied soft-rectifier-activated SAs with \(\ell_2\)-loss and \(\ell_1\)-sparsity.

### 3 Sparse Combinatorial Autoencoders

The sparse combinatorial autoencoder and sparse autoencoder share similarities, the main difference lying in the sparsity penalty. Similar to an SA, the input \((x)\) and hidden \((z_s)\) layers of an SCA are

\[\text{We overload the notation for layer names: } o_s(x^{(i)}) \text{ represents the vector of neuron outputs from } o_s \text{, induced by propagating } x^{(i)} \text{ through the network, while } o^{(h)}_s(x^{(i)}) \text{ represents the output of neuron } h \in o_s. \text{ By itself, } o_s \text{ represents the layer’s set of hidden neurons while } o_i \text{ denotes the number of neurons in } o_s.\]

\[\text{Tied weights impose the weight constraint } W_s = V_s^T. \text{ The bias terms remain untied.}\]
fully connected, as are the hidden and reconstruction ($\hat{x}$) layers (see Figure 1). The neuron outputs of $z_c$ and $\hat{x}$ are also computed via forward propagation followed by non-linear activation. The SCA also has an auxiliary hidden layer, $o_c$: for each subset of $k$ neurons in $z_c$, we define a single neuron in $o_c$, where $k$ is a model complexity hyperparameter called the \textit{combinatorial complexity}. We encapsulate this structure through a bijection $S: o_c \rightarrow \{s: s \subset z_c, |s| = k\}$. The output of a neuron in $o_c$ is the product of its associated $k$ neurons. Whereas the SA sparsity term encourages sparse mean activations in $o_s$, the SCA sparsity term regularizes mean activations in $o_c$. We soon argue that optimizing the SCA objective coarsely approximates SA optimization in a low-dimensional subspace. Concretely, we formulate the SCA objective, $J_{SCA}(\theta)$, where $\theta = \{V_c, W_c, b'_c, b_c\}$, as:

\[
J_{SCA}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[ L(x^{(i)}, \hat{x}^{(i)}) + \frac{1}{2} (||V_c||^2 + ||W_c||^2) + \frac{\beta}{o_c} \sum_{h'_c \in o_c} f \left( \frac{1}{m} \sum_{i=1}^{m} o_{c_{h'_c}}(x^{(i)}); \rho \right) \right]
\]

\[
\hat{x}^{(i)} = g_2(W_c z_c(x^{(i)} + b'_c), \quad z_c(x^{(i)}) = g_1(V_c x^{(i)} + b_c), \quad o_{c_{h'_c}}(x^{(i)}) = \prod_{h' \in S(h')} z_{c_{h'_c}}(x^{(i)}),
\]

where $f$ is now a polynomial of the form $f(x) = \sum_{d=1}^{D} \alpha_d x^d$ and $\alpha_d$ and $D$ are hyperparameters. Empirically, setting $f(x) = (x - \rho)^2 / \rho^2 - 1$, i.e. $D = 2$, $\alpha_1 = 1 / \rho^2$, and $\alpha_2 = -2 / \rho$ works well, treating $\rho$ as the mean target activation per neuron. Though this paper considers the $L_2$-loss for $L$, the SCA also supports other loss functions (e.g. cross entropy).

\textbf{Motivation.} We motivate the SCA formulation by arguing that it approximately optimizes the SA objective in a low-dimensional space. Consider the network illustrated in Figure 1(b), which we refer to as the \textit{product autoencoder} (PA). The PA computes neuron outputs in $z_p$ and $o_p$, as the SCA computes neuron outputs for $z_s$ and $o_s$, respectively. In particular, connectivity between $z_p \rightarrow o_p$ parallels connectivity between $z_c \rightarrow o_c$, e.g. $|o_p| = \binom{|z_p|}{k} > |z_p|$ for combinatorial complexity parameter $k$ as with the SCA. Note that the PA computes reconstructions $\hat{x}$ from $o_p$ rather than from $z_p$. Optimizing the PA involves minimizing a weighted sum of the reconstruction loss $L(x, \hat{x})$, a regularization penalty on the encoding ($x \rightarrow z_p$) and decoding ($o_p \rightarrow \hat{x}$) weights, and a sparsity penalty on $o_p$.

Treating $o_s$ and $o_p$ analogously, note that the SA and PA differ only in the encoding phase. We could hypothetically minimize the SA objective approximately by (i) minimizing the PA objective, (ii) constructing the SA encoder weights so as to minimize the expected $\ell_2$-distance between $o_p$ and $o_s$ over each training example, and (iii) constructing the SA decoder weights by replicating the PA decoder weights. Setting $|o_p| = |o_s|$, the PA encoder weights $x \rightarrow z_p$ approximately encode the SA encoder weights $x \rightarrow o_s$ in a low-dimensional space.

Imposing the following two additional approximations converts the PA into the SCA: (i) the sparsity penalty on $o_p$ must be expressible as a polynomial with finite degree and (ii) rather than computing
4 SCA Parameter Learning

We minimize \( J_{SCA}(\theta) \) via stochastic gradient descent. Computing the objective and gradient contributions of the first two terms in \( J_{SCA}(\theta) \) is straightforward, leading to the gradient updates for \( W_c \). Operations on the third term (SCA sparsity) are nontrivial, since summing over all neurons in \( o_c \) takes \( \mathcal{O}(|L|^c) \) time.

**Theorem 1.** Let \( X^d \) be a collection of all possible lists where each list \( L \in X^d \) contains \( d \) not necessarily distinct elements \( x_1 \in X \) and \( L^{(i)} \) denotes the \( i \)th element of \( L \). Further, let \( V_{pq} \) denote the weight connecting neuron \( p \in z_c \) to neuron \( q \in x \) in an SCA. Denoting \( \mathbb{1}_x \) as the indicator and defining \( \mathbb{1}_{x \in S(h)} = 1 \), the SCA sparsity objective and gradient are equivalent to:

\[
S_{SCA}(\theta) = \frac{1}{|o_c|} \sum_{d=1}^{N} \sum_{L \in X^d} \alpha_d m^d \varphi(L, \ast) \\
\varphi(L, p) = \sum_{h \in o_c, h' \in S(h)} \prod_{i=1}^{|L|} z_{h_i}^{(h_i)}(L^{(i)}) \mathbb{1}_{p \in S(h)} \\
\frac{\partial S_{SCA}(\theta)}{\partial V_{pq}} = \frac{1}{|o_c|} \sum_{d=1}^{N} \sum_{k=1}^{d} \alpha_d m^d \frac{\partial \varphi(L, \ast)}{\partial V_{pq}} \\
\frac{\partial \varphi(L, \ast)}{\partial V_{pq}} = \sum_{L \in X^d} \frac{L^{(k)}q_1(L^{(k)})\varphi(L, p)}{z_{h_i}^{(h_i)}(L^{(h_i)})}.
\]

(1)

Although explicitly computing \( S_{SCA} \) and \( \partial S_{SCA}/\partial V_{pq} \) by summing over all \( h \in o_c \) takes \( \mathcal{O}(|X|^{|N|/|o_c|} + |z_c| |X| |x|) \) time each, this formulation leads to an \( \mathcal{O}(|X|^{|N|/|o_c|} + |z_c| |X| |x|) \) time algorithm.

**Proof Sketch:** The alternate cost formulation and gradient derivation follow from algebra and manipulation. In addressing the complexity speedup, let \( \odot \) denote the Hadamard product and define \( r^{(L)} = z_c^{(L^{(1)})} \odot \cdots \odot z_c^{(L^{(|L|)})} \). Since \( \varphi(L, \ast) = \sum_{\kappa} r^{(L)} \), the \( \kappa \)th elementary symmetric sum of the elements in \( r^{(L)} \), it follows that \( \varphi(L, \ast) \) and \( \varphi(L, p) \) are the coefficients of \( x^{\kappa}|x|^{-\kappa} \) in polynomials \( q_1(x) = \prod_{i=1}^{|L|} \frac{1}{\alpha_i} (x - r_i^{(L)}) \) and \( q_2(x) = q_1(x)/(x - r_p^{(L)}) \), respectively. Using the Newton-Girard identities to compute the desired polynomial coefficients yields the desired speedup. We present details in the Supplementary Materials.

While Theorem 1 reduces the objective and gradient complexities to time polynomial in \( \kappa \), these complexities are at least quadratic in the dataset size, \( |X| \), due to the sums over \( X^d \) appearing in Equation (1). As a remedy, we estimate each sum over a random subset \( G \subseteq X^d \) with \( |G| \ll |X^d| \) and scale each value up by \( m^d/|G| \). The following theorem produces an approximation algorithm that bounds the error on the resulting estimator as a function of \( |G| \) and reduces empirical FISCA training times by \( \sim 10 \times \).

**Theorem 2.** Let \( G = \{L^{(1)}, L^{(2)}, \ldots, L^{(n)}\} \) where each \( (i) \) \( L \) is drawn uniformly at random from \( X^d \). Let \( S = S_{SCA} \) and \( \hat{S} \) denote the same computation where the sum over \( X^d \) in Equation (1) is computed over \( G \). Defining \( D_{pq} = \partial S_{SCA}/\partial V_{pq} \) and \( \hat{D}_{pq} \) similarly, collecting

\[
\delta \geq \frac{2|z_c|^2}{\max\{1, M_2^{2z_c} \}} \ln \left( \frac{2}{\delta} \right)
\]

(2)

This approximation lets us avoid working with \( |o_c||x| \) weights. The SA objective optimizes reconstruction error to discourage learning degenerate weights; the SCA also discourages degeneracy, but does so by channeling reconstructions through \( z_c \) rather than \( o_c \). Section 5 shows that this approximation suffices empirically.
samples guarantees that \( \Pr[|D_{pq} - \hat{D}_{pq}| \geq \epsilon] \leq \delta \), where \( M_1 = \max_{i,j} |X_j^{(i)}| \), \( M_2 = \max_{i,j} |L_j^{(i)}(L_j^{(i)})| \), and \( \alpha = \max_{i,j} \alpha_{ij} > 0 \). If we also require that the activation function \( g_1 \) be \( K \)-Lipschitz differentiable, then collecting

\[
n \geq \frac{2N^4\alpha^2K^2M_1^2 \cdot \max\{1, M_2^{2\kappa N^2 - 2}\}}{\epsilon^2} \ln \left( \frac{2}{\delta} \right)
\]

(3)
samples guarantees that \( \Pr[|S - \hat{S}| \geq \epsilon] \leq \delta \). Using this approximation reduces the gradient computation time to \( \mathcal{O}(|z_c|n + |z_c||x||x|) \).

Proof Sketch: We obtain the bounds via Hoeffding’s inequality. The complexity reduction follows, since each lower bound on the number of samples neither depends on \( |X| \) nor \( |z_c| \). See the Supplementary Materials for more detail.

Now that we have an efficient procedure to calculate the objective and gradient of the SCA sparsity term, we can optimize the full SCA objective using stochastic gradient descent-style methods.

5 Weight Transfer

FISCA leverages the SCA’s ability to quickly regularize neuron outputs in \( o_s \) by initializing the parameters of an SA with \( o_s = o_c \), hidden neurons such that the SA’s hidden activations closely match SCA hidden activations in \( o_s \). We match activations in \( o_s \) and \( o_c \) by encouraging low mean \( \ell_2 \)-distance between \( o_s \) and \( o_c \) over a randomly selected \( X_{WT} \subset X \). We refer to this procedure as weight transfer.

Formally, we attempt to minimize the following quantity to determine the encoder weights of the SA, \( V_s \) and \( b_s \):

\[
\sum_{i \in X_{WT}} \| V_s^i x^{(i)} + b_s - g_1^{-1}(o_c^i(x^{(i)})) \|^2_2.
\]

We quickly solve this problem by noticing that it decomposes into a series of independent least squares problems which we solve in parallel. Each problem takes the form min:

\[
\sum_{i \in X_{WT}} [V_s^{(h)} x^{(i)} + b_s - g_1^{-1}(o_c^{(h)}(x^{(i)}))]^2
\]

for each \( h \in o_s \), where \( V_s^{(h)} \) represents the \( h \)th row of \( V_s \). Per subproblem, each \( V_s^{(h)} \) with \( \ell_2 \)-norm greater than a threshold \( \lambda \) is normalized to have \( \ell_2 \)-norm \( \lambda \), placing a hard limit on the initialized SA’s weight decay cost \( 19 \). We call \( \lambda \) the hard \( \ell_2 \)-regularization parameter. Empirically, we found that employing hard \( \ell_2 \)-regularization allows FISCA-initializes SAs to converge to similar regions to SAs initialized with other methods. However, standard Tikhonov or no regularization places SAs in very poor local optima. Though we can compute the SA decoder weights by similarly solving a series of least squares problems, we found that using normalized initialization \( 16 \) for untied SAs and setting \( W_s = V_s^T \) for tied SAs works just as well in practice and reduces the total SA initialization time.

Thus, the full FISCA warm-start procedure proceeds as follows: (1) create an SCA and optimize its objective (Section 4) and (2) use weight transfer to create a new SA whose weights were derived from the learned SCA. We now discuss FISCA’s empirical performance through a series of experiments.

6 Experiments

We present results for two different types of commonly used SAs of varying size (roughly 1K-10K hidden neurons): (i) the sigmoid-encoded SA and (ii) the tied-weight soft-rectified SA\(^4\). The former has activation function \( g_1(x) = (1 + \exp(-x))^{-1} \) and sparsity function \( f(x) = \rho \log \frac{e}{x} + (1 - \rho) \log \frac{1 - e}{1 - x} \), while the latter has activation function \( g_1(x) = \log(1 + \exp(x)) \) and sparsity function \( f(x) = |x - \rho| \), given some mean target activation \( \rho \) (see Section 2). Both SAs use an \( \ell_2 \)-reconstruction loss and linear decoder, i.e. \( g_2(x) = x \). Our experiments consider the MNIST \( 17 \) dataset and a bank of 400K 8 \( \times \) 8 color image patches extracted at random from the STL-10 dataset \( 13 \). We select SA hyperparameters per configuration through grid-search and cross validation on a held-out portion of the respective test set; for STL-10, we use the learned SA features

\(^4\)As discussed later, although our experiments use seemingly contrived SA sizes (e.g. 969, 10585), these are necessary since we explore SCAs with \( \kappa = 2, 3, \) and \( 4 \). Specifically, we choose SA sizes \( H \) so that there exist \( n_1, n_2, \) and \( n_3 \) such that \( H, \binom{n_1^\kappa}{2}, \binom{n_2^\kappa}{2}, \) and \( \binom{n_3^\kappa}{k} \) are close in value.
as pretrained filters in a max-pooled convolutional neural network followed by softmax classification (no fine-tuning). Experiments were robust to SCA hyperparameter settings, which we keep constant throughout all experiments ($\lambda = 10^{-4}$, $\beta = 10^{-4}$, $\rho = 10^{-3} / |\theta|$, $\lambda' = 0.05$). For MNIST, we use the learned SA features to pretrain a single-layer neural net for softmax classification (no fine-tuning). We use a portion of the test set disjoint from the validation set to report test errors. We run experiments on single Amazon EC2 M3 Double Extra Large Instances.

### 6.1 SA Objectives during SCA Training

We empirically validate the claim that FISCA roughly optimizes the SA objective (Section 3) by executing weight transfer and tracking the SA objective over a test set after each step of SCA training. Throughout this paper, we conduct SCA training via L-BFGS with minibatch size 100. Weight transfer uses a random subset of $|X_{WT}| = 1K$ training instances and hard $\ell_2$-regularization parameter $\lambda' = 0.05$. Figure 2 presents how the sigmoid and soft-rectified SA test objectives evolve when applied to the STL-10 dataset (we omit training objectives, which also behave similarly, to avoid clutter). Although the figure only provides results for SAs with 969 hidden neurons initialized using SCAs with $\kappa = 2$ combinatorial complexity on MNIST due to space constraints, initializing SAs of other sizes using alternate combinatorial complexities on STL-10 also yield qualitatively similar results. The results we discuss represent results for most tested SCA hyperparameter settings ($\lambda \in [10^{-6}, 1]$, $\beta \in [10^{-3}, 10^{3}]$, $\rho \in [10^{-7}, 10^{-1}]$).

The SCA initially models the sigmoid SA objective well, as illustrated by the significant drop in log-cost (SA Cost) in Figure 2a, computed over 1K training examples. Around iteration 13, however, the SCA begins to “overfit”; while the SCA ob-

---

**Table 1:** Initial log-objectives for SAs initialized using FISCA and competing methods. Triples signify the 5th, 50th (median), and 95th percentile log-objective values across 343 initializations (grid search over 3 parameters with 7 possible values). We show only single entries for GR, since each autoencoder’s GR triples all consist of nearly identical entries. Bolded entries indicate the initialization method attaining the least median cost per row. Although initializing SAs through FISCA takes more time than alternate methods (Section 6.4), the overall time saved during SA training can be significant (Section 6.5).

<table>
<thead>
<tr>
<th>Features</th>
<th>GR</th>
<th>URR</th>
<th>NI</th>
<th>FISCA ($\kappa = 2$)</th>
<th>FISCA ($\kappa = 4$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>969</td>
<td>(7.88, 7.97, 8.00)</td>
<td>(7.91, 7.91, 7.92)</td>
<td>(2.15, 2.73, 4.17)</td>
<td>(1.96, 2.35, 4.00)</td>
<td>(2.06, 2.42, 4.23)</td>
</tr>
<tr>
<td>4845</td>
<td>(9.62, 9.69, 9.71)</td>
<td>(9.63, 9.63, 9.63)</td>
<td>(2.27, 2.95, 4.48)</td>
<td>(2.16, 2.67, 3.97)</td>
<td>(2.11, 2.47, 4.14)</td>
</tr>
<tr>
<td>10585</td>
<td>(10.45, 10.51, 10.53)</td>
<td>(10.45, 10.45, 10.45)</td>
<td>(2.36, 3.13, 4.69)</td>
<td>(2.28, 2.80, 4.28)</td>
<td>(2.26, 2.80, 3.99)</td>
</tr>
</tbody>
</table>

---

**Figure 2:** SA test objective metrics on MNIST during FISCA. We superimpose log(SA Cost), True log(S), i.e. the sparsity contribution towards log(SA Cost), and Estimated log(S), i.e. an estimate of True log(S) discussed in (Section 6.3).
jective attempts to regularize outputs from $o_c$, the SA objective asks for sparsity in $o_c$. Both neuron outputs are computed differently, as $o_c$ is computed as a product of neuron outputs from $z$, whereas $\theta_s$ is computed as a nonlinearly activated linear combination of values from $x$. Interestingly, this overfitting phenomenon doesn’t arise when initializing soft-rectified SAs. As shown in Figure 2, $\log(J_{SA})$ does not appear to increase; we observed similar results even until 100 SCA iterations, suggesting an approximately linear relation between the neuron outputs in $o_c$ and $\theta_s$.

6.2 Initial SA Objectives across Initialization Methods

According to Table 1, FISCA yields much lower median log-objectives than all other methods, and almost always yields ranges that do not overlap with those of GR, URR, and NI. Further, FISCA’s relative gains over the second best method per experiment grow consistently with autoencoder size. Surprisingly, though SCAs with larger $\kappa$ represent SAs with fewer parameters, FISCA with larger receptive field sizes often yields lower initial objectives. Of the 18 types of experiments run, FISCA with $\kappa = 2$ is the winning initializer 3 times whereas FISCA with $\kappa = 3$ and $\kappa = 4$ are winning initializers 8 and 7 times. This effect particularly manifests in training soft-rectified SAs on MNIST, where FISCA with $\kappa = 3$ attains the lowest median objectives of 6 times. This phenomenon mostly likely occurs, as using too many parameters in an SCA to model the SA objective leads to overfitting. Ultimately, these results jointly suggest that FISCA approximately conducts low-dimensional optimization of the SA objective.

6.3 Early Stopping in SCA Training

While we wish to terminate SCA training at the minimum of the SA Cost curve, we would have to compute the SA cost after each SCA iteration. Rather than using the SA Cost curve to dictate stopping criteria, we consider using only the sparsity contribution to the total cost (True $S$ in Figure 2), motivated by the observation that the SA Cost curve reaches a minimum close to the point where True $S$ reaches its minimum. Though computing True $S$ involves performing weight transfer after each iteration, we can approximate True $S$ by using a small subset of the SCA’s hidden neurons in weight transfer. Empirically, we find that using weight transfer with $1K$ training examples and hard $\ell_2$-regularization $\lambda = 0.05$ to extract a sparse autoencoder with 50 hidden neurons approximates the true sparsity contribution well. As shown in Figure 2 Estimated $S$ serves as a strong indicator to terminate SCA training. We use this early stopping method in all further experiments, as it is also robust to SCA hyperparameter choices.

6.4 Initialization Efficiency

Table 2 presents average FISCA wall-times across several SA sizes. We juxtapose these times by times taken to compute single SA gradients over minibatches of size 100, which we also use during autoencoder training in Section 6.5. FISCA scales well to large autoencoders with respect to efficiency: for example, running FISCA ($\kappa = 4$) on MNIST to initialize an SA with $10K$ hidden neurons roughly costs only 5 steps of minibatch gradient descent. For all sizes explored, using $\kappa = 4$ takes as much as or lesser time than 30 SGD iterations in all cases, making FISCA an inexpensive procedure. This is interesting particularly for large autoencoders, since FISCA’s low overhead eventually leads to significant time savings during SA training (see Section 6.5). Finally, increasing $\kappa$ generally leads to decreased FISCA wall-times, an expected result, since higher values of $\kappa$ while keeping fixed SA sizes lead to FISCA having to train SCAs with fewer hidden neurons. While using using $\kappa = 4$ “costs” fewer than 30 SGD iterations, using $\kappa = 2, 3$ can cost more than 100 SGD iterations depending on the activation function and dataset.

<table>
<thead>
<tr>
<th>$</th>
<th>\theta_s</th>
<th>$</th>
<th>FISCA Wall Times, Sigmoid (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>969</td>
<td>4845</td>
<td>10585</td>
</tr>
<tr>
<td>$\kappa = 2, S$</td>
<td>1.23 (82)</td>
<td>1.71 (24)</td>
<td>2.52 (18)</td>
</tr>
<tr>
<td>$\kappa = 2, M$</td>
<td>2.22 (44)</td>
<td>4.98 (19)</td>
<td>9.56 (14)</td>
</tr>
<tr>
<td>$\kappa = 3, S$</td>
<td>1.14 (76)</td>
<td>1.56 (22)</td>
<td>2.07 (15)</td>
</tr>
<tr>
<td>$\kappa = 3, M$</td>
<td>1.59 (32)</td>
<td>2.46 (9)</td>
<td>3.67 (6)</td>
</tr>
<tr>
<td>$\kappa = 4, S$</td>
<td>1.08 (72)</td>
<td>1.59 (23)</td>
<td>2.27 (17)</td>
</tr>
<tr>
<td>$\kappa = 4, M$</td>
<td>1.47 (29)</td>
<td>2.44 (9)</td>
<td>3.49 (5)</td>
</tr>
<tr>
<td>$\nabla S$</td>
<td>0.02</td>
<td>0.07</td>
<td>0.14</td>
</tr>
<tr>
<td>$\nabla M$</td>
<td>0.05</td>
<td>0.27</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Table 2: Mean FISCA wall times on the STL-10 (first row per entry, S) and MNIST (second row per entry, M). Entries with $\nabla$ denote times taken to compute one SA gradient over batch size 100. Values in parentheses denote $\text{Time}_{\kappa=1}/\text{Time}_{\kappa}$ per entry, or the cost of FISCA in “number of SGD iterations.” Bolded values indicate configurations with lowest $\text{Time}_{\kappa=1}/\text{Time}_{\kappa}$ per row.
Sparse Autoencoder Test Objectives during Training

Figure 3: Test log-objectives of SAs initialized using several methods, each trained using SGD with AdaGrad (batch size 100). Best viewed in color. See Supplementary Materials for further results.

6.5 Autoencoder Training Times

We train sigmoid and soft-rectified SAs of various sizes using GR, URR, NI, and FISCA with $\kappa = 2, 3,$ and $4$ on STL-10 and MNIST, where we choose SA hyperparameters via cross-validation as described early in Section 6. We initialize GR, URR, and NI-based SAs in the sparse regime by setting bias values to $g^{-1}(\rho)$, which significantly improves convergence times and use 10K iterations of stochastic gradient descent (batch size 100) with AdaGrad [20]. We illustrate results in Figure 3, where we offset FISCA curves to the right based on $\text{Time}_{\kappa=i}/\text{Time}_\kappa$ values (Table 2).

FISCA-initialized SAs almost always outperform or match all non-FISCA SAs in both convergence speed and final test objective. On the STL-10 dataset, FISCA-initialized sigmoid SAs converge marginally faster to similar local optima attained by other methods. On the MNIST dataset, FISCA causes the sigmoid SA with 10585 hidden neurons to converge $3 \times$ faster than URR, the next closest method. Striking disparities emerge when training soft-rectified SAs, where non-FISCA SAs converge slowly and often to poor minima, while FISCA-initialized SAs converge rapidly to significantly better optima. For example, while MNIST-trained soft-rectified non-FISCA SAs with 10K hidden neurons take more than 10K iterations to converge, FISCA-initialized SAs take fewer than 2000 iterations to converge to a much smaller optimum ($20 \times$). We further found that FISCA’s relative improvements over competing methods increased with autoencoder size (Supplementary Materials). Finally, varying $\kappa$ has little impact on final test objectives and convergence rates, a surprising result, since SCAs with larger $\kappa$ optimize the SA objective using fewer parameters (Table 2).

7 Conclusion

FISCA provides a fast and robust method to initialize SAs, which we demonstrate leads to significantly reduced training times. Intuitively, FISCA optimizes the SA objective by conducting optimization on a low-dimension space. Ultimately, we illustrate that FISCA-based SAs converge faster to equal or better optima over initialization techniques used on two state-of-the-art autoencoder types over two datasets.
References


