Learning compact vector representations from weight matrices

Chris Liu

October 16, 2022

University of California, Santa Cruz

Table of Contents

Table of Contents

Motivations

- Neural network weights are difficult to interpret. It would be helpful if we could find direct connections between weights and their performance.

[1] Just like we can retrieve similar words by taking those with small vector distance.

Motivations

- Neural network weights are difficult to interpret. It would be helpful if we could find direct connections between weights and their performance.
- Word embeddings encode meaning of tokens such that similar tokens have smaller vector distance. Does weight matrices have such a property?

[1] Just like we can retrieve similar words by taking those with small vector distance.

Motivations

- Neural network weights are difficult to interpret. It would be helpful if we could find direct connections between weights and their performance.
- Word embeddings encode meaning of tokens such that similar tokens have smaller vector distance. Does weight matrices have such a property?
- Large neural networks can take gigabytes (or more) of space. What if we can store a piece of them and efficiently recover the full weights on demand?

[1] Just like we can retrieve similar words by taking those with small vector distance.

Motivations

- Neural network weights are difficult to interpret. It would be helpful if we could find direct connections between weights and their performance.
- Word embeddings encode meaning of tokens such that similar tokens have smaller vector distance. Does weight matrices have such a property?
- Large neural networks can take gigabytes (or more) of space. What if we can store a piece of them and efficiently recover the full weights on demand?
- Neural networks are extremely high-dimensional. Even if we can accurate predict its performance, it remains hard to use hundreds of thousands (or even larger) vector as input.

[1] Just like we can retrieve similar words by taking those with small vector distance.

Research objectives

1. How can we encode weight matrices?

Research objectives

1. How can we encode weight matrices?
2. How can we preserve the quality of the weights in such embeddings such that they have the same convenient property as word embeddings?

Research objectives

1. How can we encode weight matrices?
2. How can we preserve the quality of the weights in such embeddings such that they have the same convenient property as word embeddings?
3. Can we maximize the distance between emebddings from good weights and those from bad weights?

Research objectives

1. How can we encode weight matrices?
2. How can we preserve the quality of the weights in such embeddings such that they have the same convenient property as word embeddings?
3. Can we maximize the distance between emebddings from good weights and those from bad weights?
4. How can we solve the large-input problem?

Table of Contents

Weights matrices of fully-connected networks

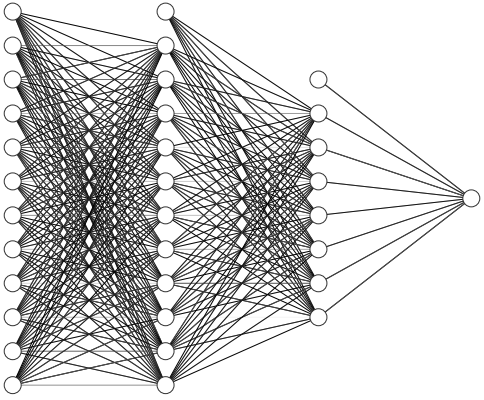


Figure 1: An example of a 3-layer fully-connected network. For all layers, the shape of a particular weight matrix is always $(D_{\text{out}}, D_{\text{in}})$, where D_{in} is the input size and D_{out} is the hidden size. The bias vector always has shape D_{out}. We can concatenate both in the column dimension to form a matrix of $(D_{\text{out}}, D_{\text{in}} + 1)$.

Weights matrices of convolutional networks

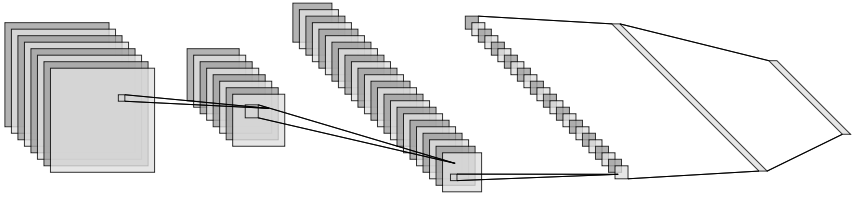


Figure 2: An example of a LeNet. For all convolutional layers, the shape of a particular weight tensor is always $(D_{\text{out}}, D_{\text{ch}}, D_{\text{h}}, D_{\text{w}})$, corresponding to number of filters, number of channels, filter height, and filter width.

We can squeeze the last three dimensions of the weight tensor into one as D_{in}, which gives us a matrix.

$$D_{\text{in}} = D_{\text{ch}} \times D_{\text{h}} \times D_{\text{w}}$$

Table of Contents

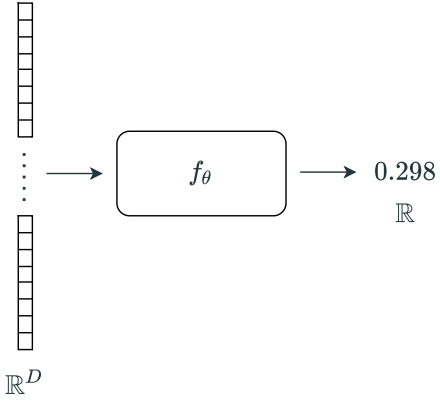


Figure 3: Given an input weight vector and a model f parameterized by θ, the output is a real number between [0, 1].

CNN Zoo V1 [UKG+20]

- Consists of four-layer CNNs (16 filters per-layer, 4970 parameters in total) and their final test accuracies
- CNNs are trained on CIFAR-10, MNIST, Fashion-MNIST, and SVHN (all converted to grayscale images)
- For each of the 300K training sessions, sample one checkpoint from a certain epoch

CNN Zoo V2 (ours)

- Consists of four-layer CNNs (64 filters per-layer, 75146 parameters in total) and their final test accuracies
- CNNs are trained on CIFAR-10, MNIST, Fashion-MNIST, and USPS[1] (all converted to grayscale images)
- For each of the 1K training sessions, take all the checkpoints (i.e., from epoch 1 to epoch n.
  - This simulates the real-world hyperparameter tuning procedure.

[1] The SVHN dataset is huge and we did not have enough time and computation to train on it.

CNN Zoo v2

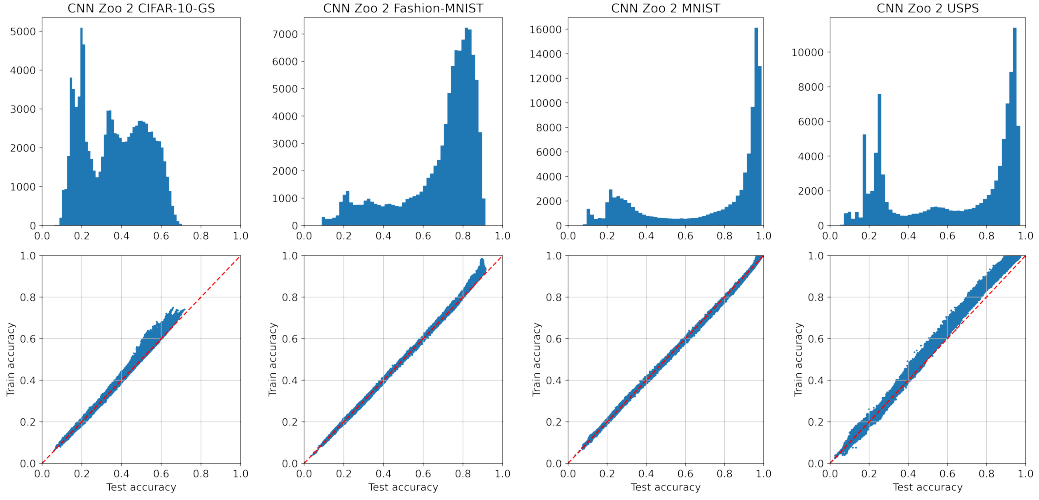


Figure 4: Top: The test accuracy distribution of the CNN Zoo 2 dataset. **Bottom**: Test accuracy versus training accuracy. This reflects whether base CNN models are overfitting.

We can formulate the learning as a regression task with mean squared error loss. We train a predictor $f_\theta : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is a weight vector/matrix and $\mathcal{Y}$ is a real number between [0, 1].

$$\begin{aligned}\theta^* &= \arg\min_\theta \ \mathrm{MSE}(f_\theta(x), y) \\ &= \arg\min_\theta \ \frac{1}{n}\sum_{i=1}^{n} (f_\theta(x_i), y_i)^2 \,.\end{aligned}$$

We also evaluate the goodness of fit using R^2 scores.

$$R^2 = 1 - \frac{\sum_i (y_i - f_\theta(x_i))^2}{\sum_i (y_i - \bar{y})^2}$$

Problems with the existing approach

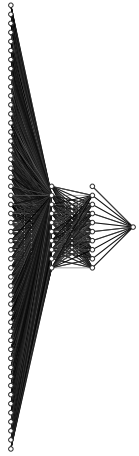


Figure 5: The input will be a giant vector, and the number of parameters in the first layer of our predictor will explode.

Problems with the existing approach (continued)

Suppose we have a 1M parameter base model as the input, which gives us a 1M-element vector, and 512 hidden units in the first layer of our predictor. The size of the first layer will be 512M, which is about 5 times of the parameters of a $\text{BERT}_{\text{base}}$.

Parameter calculation

More generally, let L denote the number of layers in our base model (the inputs), $|\phi_l|$ the number of parameters in layer l of our base model, H the number of hidden units in the first layer of the predictor.

$$H + \sum_{i=1}^{L} |\phi_i| \cdot H$$

Note that $\sum_{i=1}^{L} |\phi_i| \cdot H$ is just $|\phi| \cdot H$ if we add up the number of parameters in all layers. You'll see why I wrote it like this later.

A naive approach: FC-Linear

Why not just use a small H?

$$E + \sum_{i=1}^{L} |\phi_i| \cdot E$$

We now denote H using E, the embedding dimension, where $E \ll H$. In other words, the first layer of our predictor now has E hidden units. Given a D-dimensional vector, the first layer projects this input linearly to a E dimensional vector (assuming we do not add any non-linearities after layer 1).

Distributive encoding: ComVeX$_{\text{Linear}}$

It is not wise to assume the weights in each layer are distributionally equivalent. So we encode them layer-by-layer using compact vector representations (ComVeX).

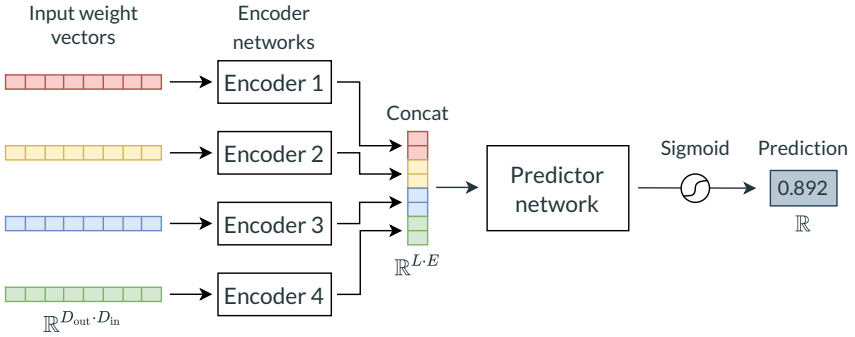


Figure 6: For each of the L layers in a base CNN, we use a individual encoder network to do projection. In our case, L is 4. The reduced vectors for all layers are concatenated together to feed into the predictor network. The sigmoid transformation is used to ensure the final accuracy is strictly between [0, 1]. Here D_{out} is equal to H.

The number of parameters in the first layer is

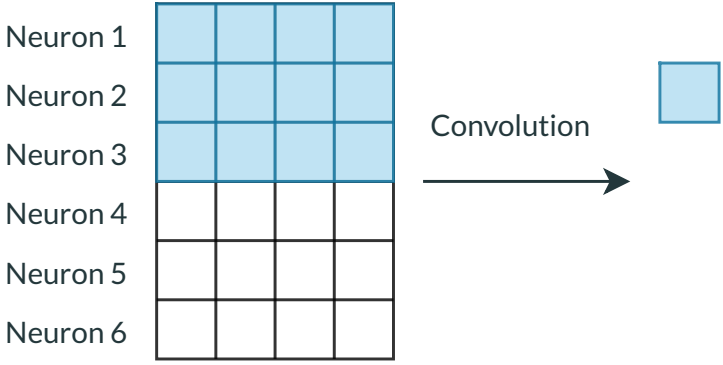
$$\sum_{i=1}^{L} \left(|\phi_i| \cdot E_i + E_i \right).$$

Note that the number of parameters stays the same because $E = \sum_i E_i$. What if a particular $|\phi_i|$ is large?

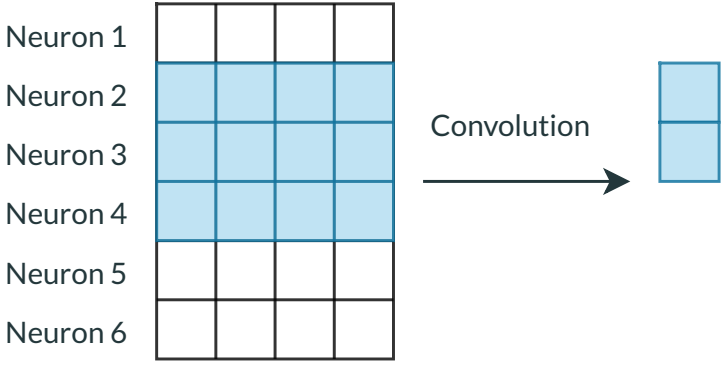
Distributive encoding: ComVeX$_{\text{Conv}}$

Key idea: instead of fully-connecting each weight with each hidden unit, we leverage the weight sharing property of convolution.

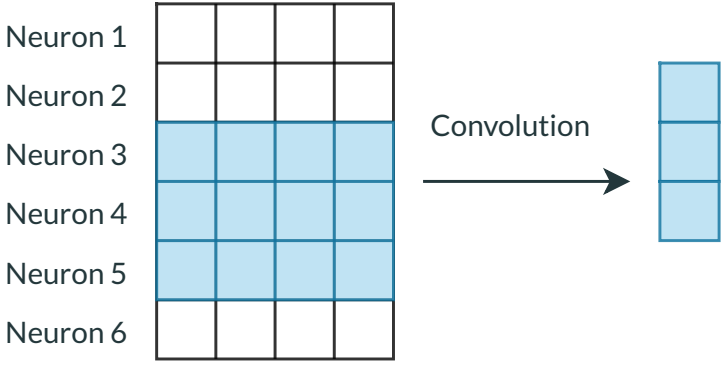
Distributive encoding: ComVeX$_{\text{Conv}}$



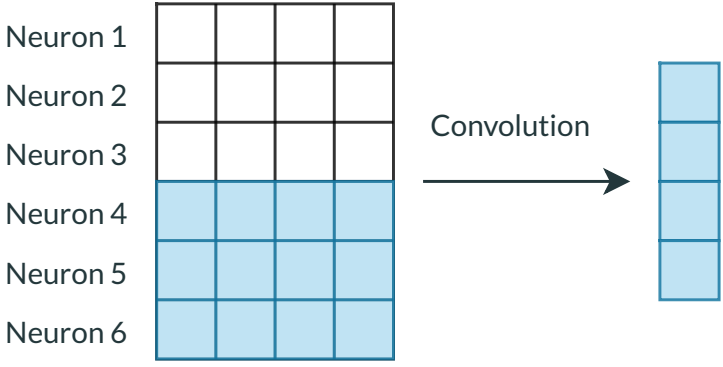
Distributive encoding: ComVeX$_{\text{Conv}}$



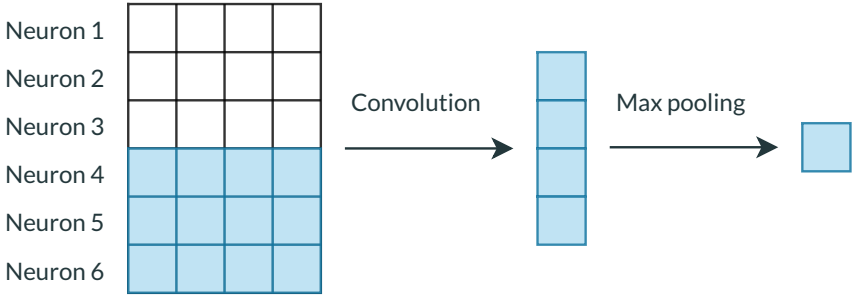
Distributive encoding: ComVeX$_{\text{Conv}}$



Distributive encoding: ComVeX$_{\text{Conv}}$



Distributive encoding: ComVeX$_{\text{Conv}}$



Distributive encoding: ComVeX$_{\text{Conv}}$

The number of parameters in a convolution layer is:

$$D_{\text{out}} \cdot D_{\text{ch}} \cdot D_{\text{h}} \cdot D_{\text{w}}$$

In our case,

$$\begin{aligned} D_{\text{out}} &= E \\ D_{\text{ch}} &= 1 \\ D_{\text{h}} &= 3 \\ D_{\text{w}} &= D_{\text{ch}}^{\text{base}} \cdot D_{\text{h}}^{\text{base}} \cdot D_{\text{w}}^{\text{base}} \\ &= \frac{|\phi_i|}{D_{\text{out}}^{\text{base}}} \end{aligned}$$

The number of parameters of the convolutional encoders can be written as

$$\sum_{i=1}^{L} \left(\frac{3 \cdot |\phi_i|}{D_{\text{out}}^{\text{base}_i}} \cdot E_i + E_i \right)$$

Parameter complexity

Architecture	Complexity
FC network	$\mathcal{O}(L \cdot \phi \cdot H)$
$\text{FC}_{\text{Linear}}$	$\mathcal{O}(L \cdot \phi \cdot E)$
$\text{ComVeX}_{\text{Linear}}$	$\mathcal{O}(L \cdot \phi \cdot E)$
$\text{ComVeX}_{\text{Conv}}$	$\mathcal{O}\left(L \cdot \frac{3 \cdot \phi}{h_i} \cdot E\right)$

Table 1: Complexity of the number of parameters in the first-layer encoders. $\text{FC}_{\text{Linear}}$ and $\text{ComVeX}_{\text{Linear}}$ both have the same complexity because the first layer of $\text{FC}_{\text{Linear}}$ is simply a linear layer with identity activation, and the first layer in $\text{ComVeX}_{\text{Linear}}$ performs the encoding operations in a distributive fashion. For all models, E is usually about seven times smaller than H.

Table of Contents

Hyperparameter search

For CNN Zoo 2, we used the following random hyperparameter sampling.

- Number of layers: uniformly from range *[2, 8]*.
- Hidden size: uniformly from range *[64, 512)*.
- Dropout rate: uniformly from range *[0.0, 0.7]*.
- Weight decay: log-uniformly from range *[1e-5, 1e-2]*.
- Optimizers: uniformly from `[adam, adamw, adamax, nadam, radam]`.
- Batch size: uniformly from `{64, 128, 256, 512}`.
- Initializer: uniformly from `{xavier, he, orthogonal, original}` (all from the uniform distribution (default in PyTorch).

Prediction performance on CNN Zoo 1

Architecture	Parameters	Test loss	Test R^2
FC network	3538509	0.000171	0.9866
$\text{FC}_{\text{Linear}}$	1774519	0.000168	0.9867
$\text{ComVeX}_{\text{Linear}}$	1623967	0.000169	**0.9869**
$\text{ComVeX}_{\text{Conv}}$	**458153**	**0.000167**	0.9868

Table 2: CNN Zoo 1 CIFAR-10

Architecture	Parameters	Test loss	Test R^2
FC network	3848853	0.000558	0.993
$\text{FC}_{\text{Linear}}$	2017377	0.000564	0.9929
$\text{ComVeX}_{\text{Linear}}$	**842399**	**0.000505**	0.9935
$\text{ComVeX}_{\text{Conv}}$	966321	0.000513	**0.9936**

Table 3: CNN Zoo 1 Fashion-MNIST

Prediction performance on CNN Zoo 1

Architecture	Parameters	Test loss	Test R^2
FC network	2817777	0.000658	0.9944
FC_{Linear}	1956661	0.000586	0.995
$ComVeX_{Linear}$	1397399	**0.000521**	**0.9955**
$ComVeX_{Conv}$	**1339361**	0.000533	**0.9955**

Table 4: CNN Zoo 1 MNIST

Architecture	Parameters	Test loss	Test R^2
FC network	3675685	0.000265	0.9834
FC_{Linear}	2054869	0.000269	0.9832
$ComVeX_{Linear}$	1192327	0.000206	0.9871
$ComVeX_{Conv}$	**950801**	**0.000203**	**0.9875**

Table 5: CNN Zoo 1 SVHN

Prediction performance on CNN Zoo 2

Architecture	Parameters	Test loss	Test R^2
FC network	35608249	0.011631	0.5214
$\text{FC}_{\text{Linear}}$	19974475	0.016832	0.2971
$\text{ComVeX}_{\text{Linear}}$	5954785	0.010816	0.5541
$\text{ComVeX}_{\text{Conv}}$	**391325**	**0.001233**	**0.9496**

Table 6: CNN Zoo 2 CIFAR-10

Architecture	Parameters	Test loss	Test R^2
FC network	32530569	0.023807	0.3966
$\text{FC}_{\text{Linear}}$	19827601	0.024885	0.3718
$\text{ComVeX}_{\text{Linear}}$	5851759	0.008128	0.7828
$\text{ComVeX}_{\text{Conv}}$	**1530671**	**0.00173**	**0.9562**

Table 7: CNN Zoo 2 Fashion-MNIST

Prediction performance on CNN Zoo 2

Architecture	Parameters	Test loss	Test R^2
FC network	36295603	0.02099	0.7596
FC_{Linear}	19930939	0.031575	0.6361
$ComVeX_{Linear}$	6249599	0.00694	0.9205
$ComVeX_{Conv}$	**1400207**	**0.002254**	**0.9741**

Table 8: CNN Zoo 2 MNIST

Architecture	Parameters	Test loss	Test R^2
FC network	38509217	0.04628	0.4995
FC_{Linear}	20236345	0.057672	0.373
$ComVeX_{Linear}$	6065281	0.037939	0.5748
$ComVeX_{Conv}$	**843887**	**0.002792**	**0.9693**

Table 9: CNN Zoo 2 USPS

Parameter efficiency

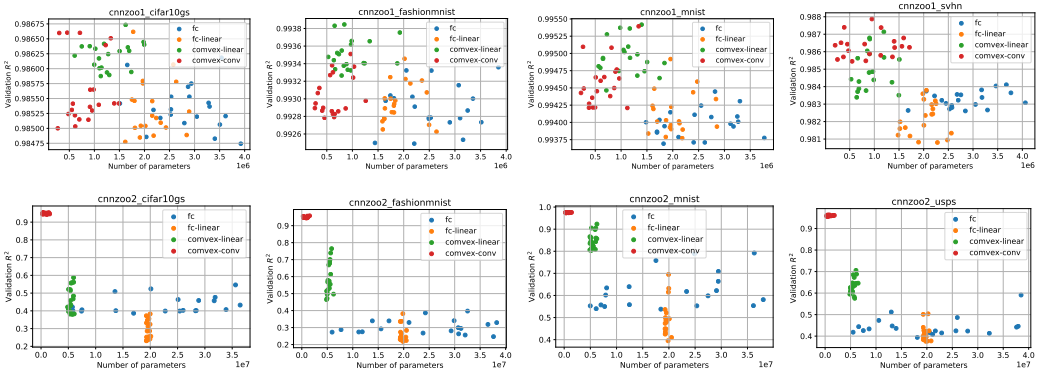


Figure 7: Number of parameters versus validation R^2 scores. For each model architecture, we tuned 100 sets of hyperparameters and took the top-20 candidates ranked by validation R^2 scores. In most scenarios, $\text{ComVeX}_{\text{Linear}}$ and $\text{ComVeX}_{\text{Linear}}$ achieved the best performances with the least number of parameters.

Embedding visualization: raw features (weights)

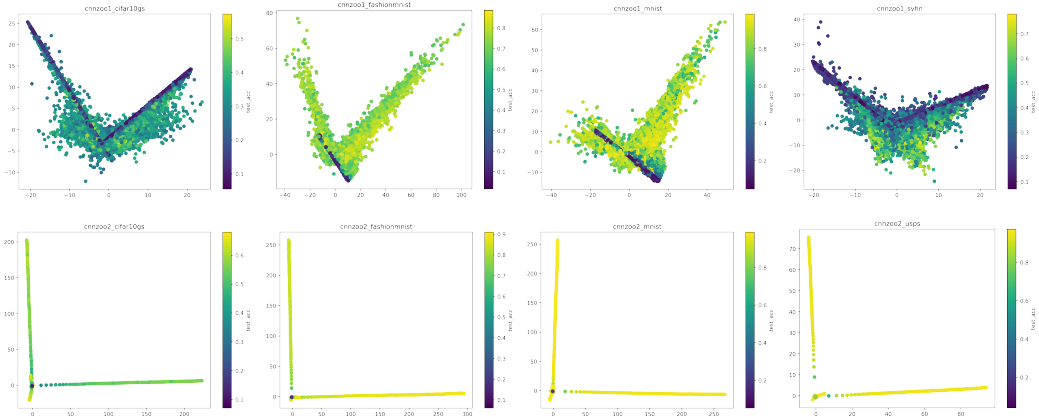


Figure 8: CNN Zoo 1 (top) and 2 (bottom) raw weights

Embedding visualization: FC_{Linear}

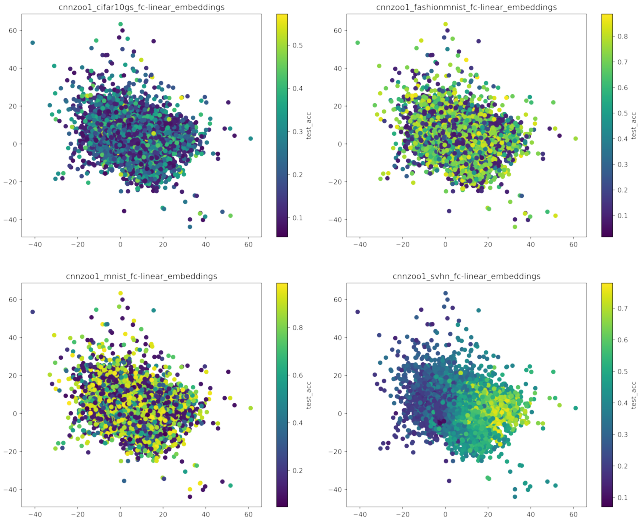


Figure 9: CNN Zoo 1 embeddings generated by FC_{Linear}

Embedding visualization: FC_{Linear}

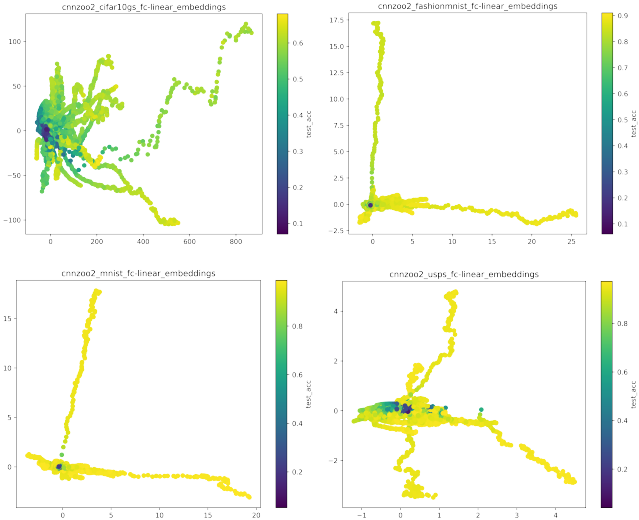


Figure 10: CNN Zoo 2 embeddings generated by FC_{Linear}

Embedding visualization: ComVeX$_{\text{Linear}}$

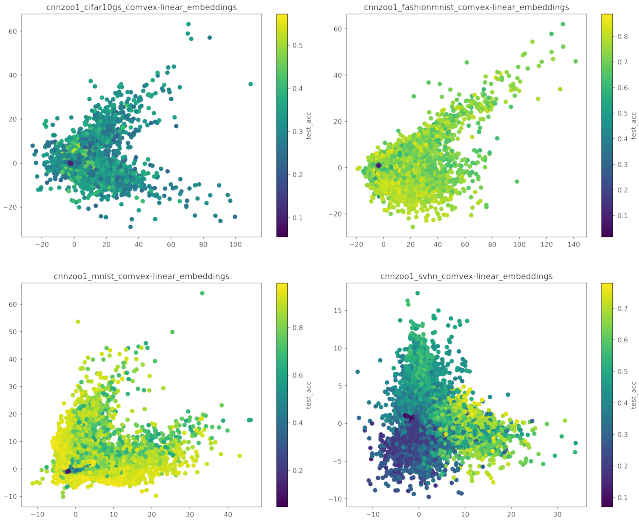


Figure 11: CNN Zoo 1 embeddings generated by ComVeX$_{\text{Linear}}$

Embedding visualization: ComVeX$_{\text{Linear}}$

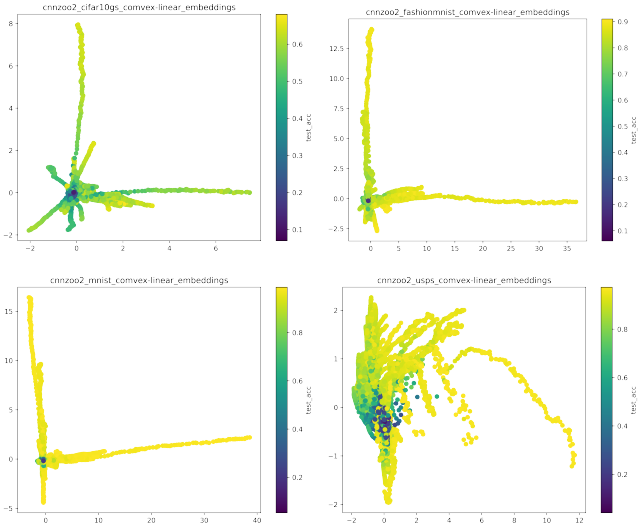


Figure 12: CNN Zoo 2 embeddings generated by ComVeX$_{\text{Linear}}$

Embedding visualization: ComVeX$_{\text{Conv}}$

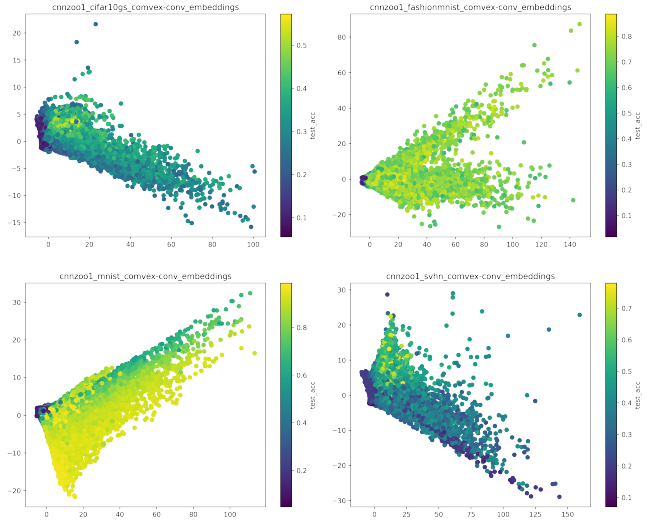


Figure 13: CNN Zoo 1 embeddings generated by ComVeX$_{\text{Conv}}$

Embedding visualization: ComVeX$_{\text{Conv}}$

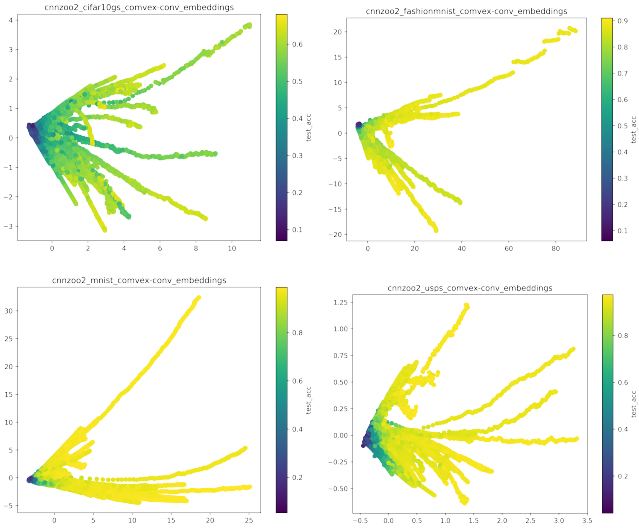
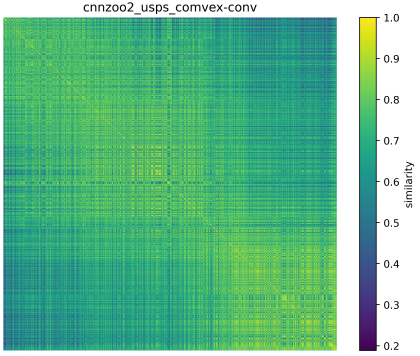


Figure 14: CNN Zoo 2 embeddings generated by ComVeX$_{\text{Conv}}$

Cosine similarity matrix visualization



What we want to see:

1. We want weight vectors/embeddings of high-accuracy networks to have small distance with each other (lighter).
2. We want weight vectors/embeddings of low-accuracy networks to have small distance with each other (lighter).
3. We want weight vectors/embeddings of high-accuracy networks to have large distance with those with low-accuracy (darker).

Raw weights similarity matrices

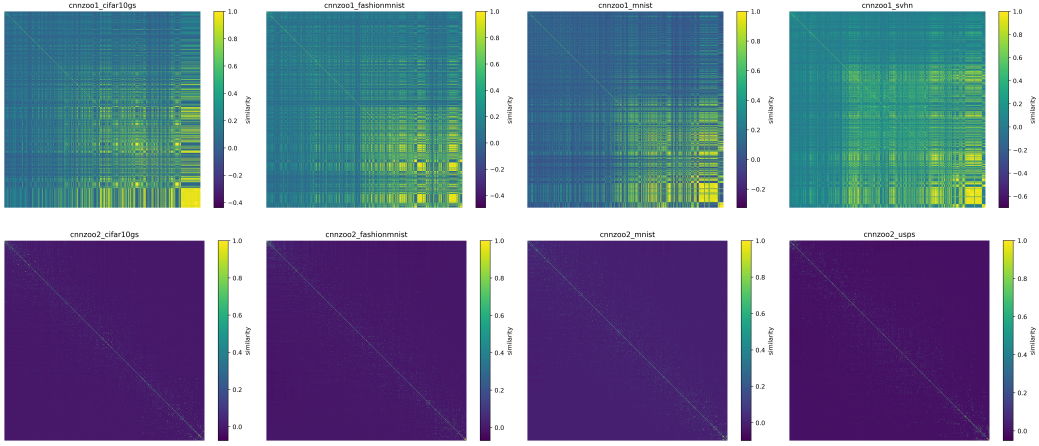


Figure 15: Similarity matrices of raw weights. On both axes, models are sorted by their accuracy in descending order (from left to right for x, or from top to bottom for y). Lighter means higher similarity.

Embedding similarity matrices: FC_{Linear}

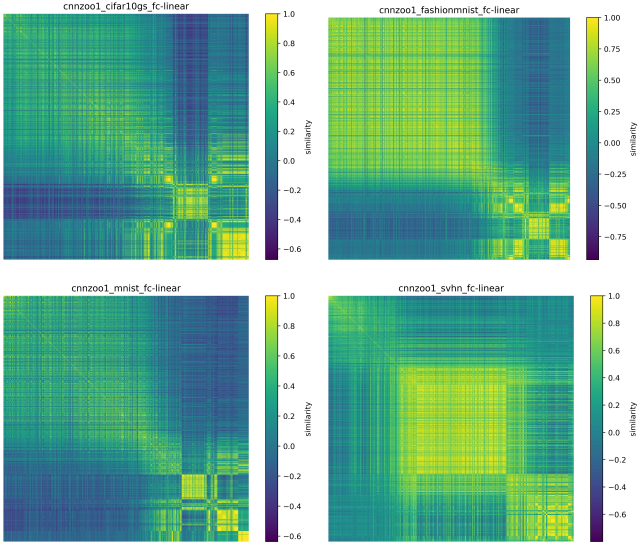


Figure 16: CNN Zoo 1

Embedding similarity matrices: FC_{Linear}

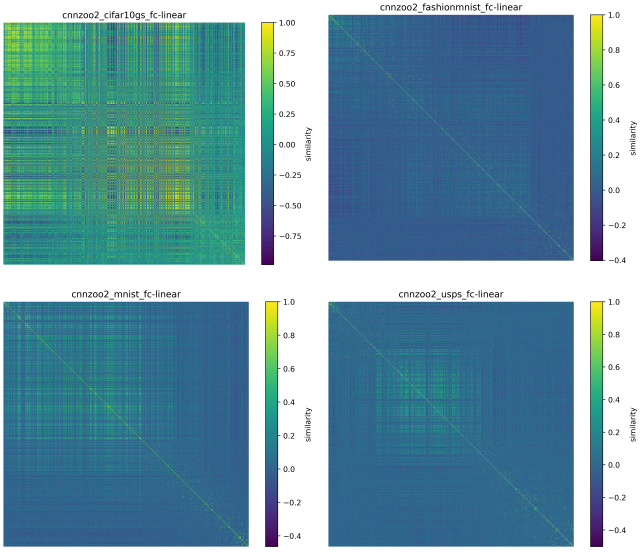


Figure 17: CNN Zoo 2

Embedding similarity matrices: ComVeX$_{\text{Linear}}$

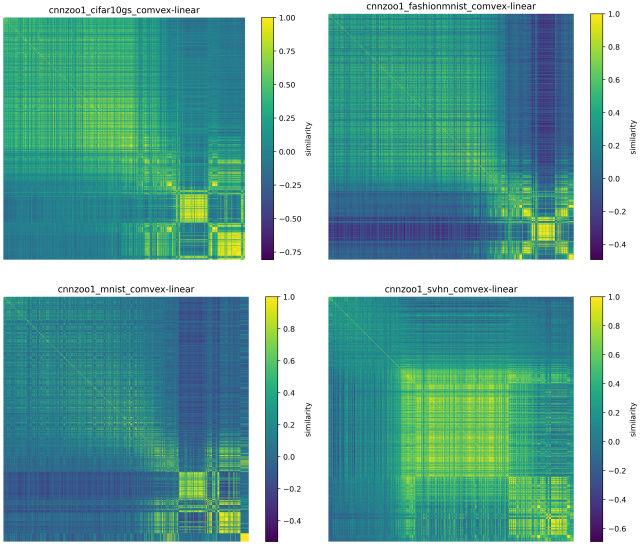


Figure 18: CNN Zoo 1

Embedding similarity matrices: ComVeX$_{\text{Linear}}$

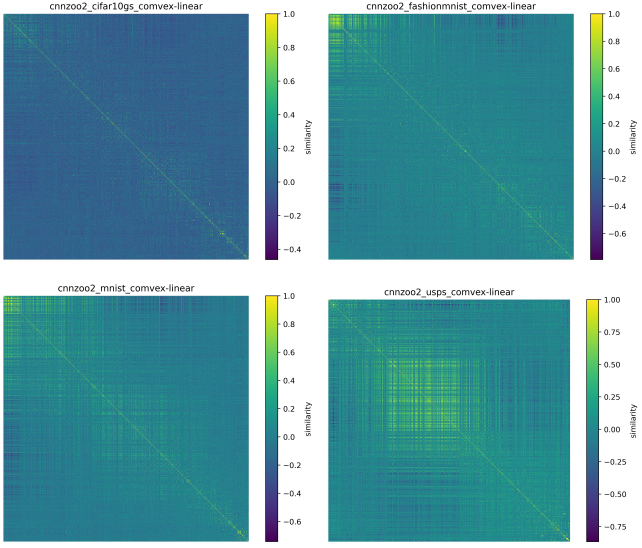


Figure 19: CNN Zoo 2

Embedding similarity matrices: ComVeX$_{\text{Conv}}$

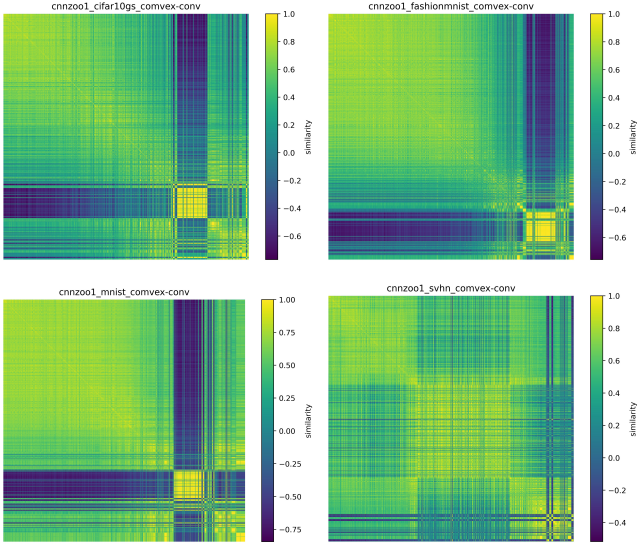


Figure 20: CNN Zoo 1

Embedding similarity matrices: ComVeX$_{\text{Conv}}$

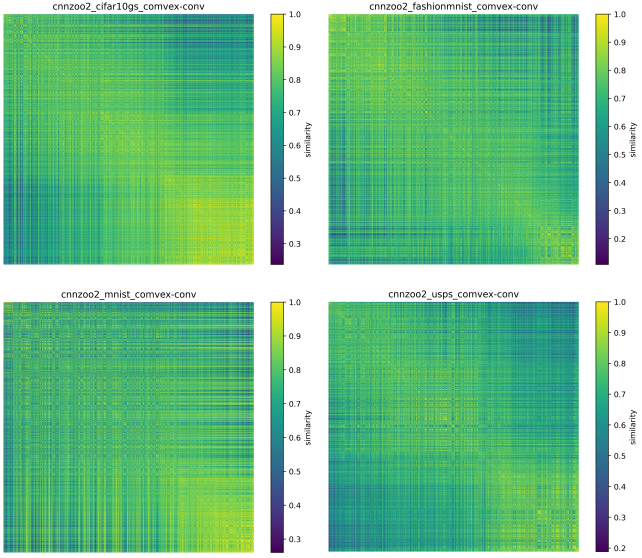


Figure 21: CNN Zoo 2

Table of Contents

Research questions

1. We project neural network weight matrices to a lower-dimensional space by incorporating a set of small linear encoders.
2. The linear encoders learn to produce high-quality embeddings that reflect the goodness of the weights.
3. The (good and bad) embeddings are well-separated, which is supported by both the 2-d visualizations and similarity matrices.
4. We solve the large-input problem by reducing both H and $|\phi|$ terms in the parameter complexity notation.

Questions

Table of Contents

References i

[UKG+20] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya Tolstikhin. Predicting neural network accuracy from weights. *arXiv preprint arXiv:2002.11448*, 2020.