Supermasks in Superposition

Mitchell Wortsman* University of Washington
Vivek Ramanujan* Allen Institute for AI
Rosanne Liu ML Collective
Aniruddha Kembhavi† Allen Institute for AI

Mohammad Rastegari University of Washington
Jason Yosinski ML Collective
Ali Farhadi University of Washington

Abstract

We present the Supermasks in Superposition (SupSup) model, capable of sequentially learning thousands of tasks without catastrophic forgetting. Our approach uses a randomly initialized, fixed base network and for each task finds a subnetwork (supermask) that achieves good performance. If task identity is given at test time, the correct subnetwork can be retrieved with minimal memory usage. If not provided, SupSup can infer the task using gradient-based optimization to find a linear superposition of learned supermasks which minimizes the output entropy. In practice we find that a single gradient step is often sufficient to identify the correct mask, even among 2500 tasks. We also showcase two promising extensions. First, SupSup models can be trained entirely without task identity information, as they may detect when they are uncertain about new data and allocate an additional supermask for the new training distribution. Finally the entire, growing set of supermasks can be stored in a constant-sized reservoir by implicitly storing them as attractors in a fixed-sized Hopfield network.

1 Introduction

Learning many different tasks sequentially without forgetting remains a notable challenge for neural networks [47, 56, 23]. If the weights of a neural network are trained on a new task, performance on previous tasks often degrades substantially [33, 10, 12], a problem known as *catastrophic forgetting*. In this paper, we begin with the observation that catastrophic forgetting cannot occur if the weights of the network remain fixed and random. We leverage this to develop a flexible model capable of learning thousands of tasks: *Supermasks in Superposition* (SupSup). SupSup, diagrammed in Figure 1, is driven by two core ideas: **a)** the expressive power of untrained, randomly weighted subnetworks [57, 39], and **b)** inference of task-identity as a gradient-based optimization problem.

a) The expressive power of subnetworks Neural networks may be overlaid with a binary mask that selectively keeps or removes each connection, producing a subnetwork. The number of possible subnetworks is combinatorial in the number of parameters. Researchers have observed that the number of combinations is large enough that even within randomly weighted neural networks, there exist *supermasks* that create corresponding subnetworks which achieve good performance on complex tasks. Zhou *et al.* [57] and Ramanujan *et al.* [39] present two algorithms for finding these supermasks while keeping the weights of the underlying network fixed and random. SupSup scales to many tasks by finding for each task a supermask atop a shared, untrained network.

b) Inference of task-identity as an optimization problem When task identity is unknown, SupSup can infer task identity to select the correct supermask. Given data from task j, we aim

*Equal contribution. †Also affiliated with the University of Washington. Code available at `https://github.com/RAIVNLab/supsup` and correspondence to `{mitchnw,ramanv}@cs.washington.edu`.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

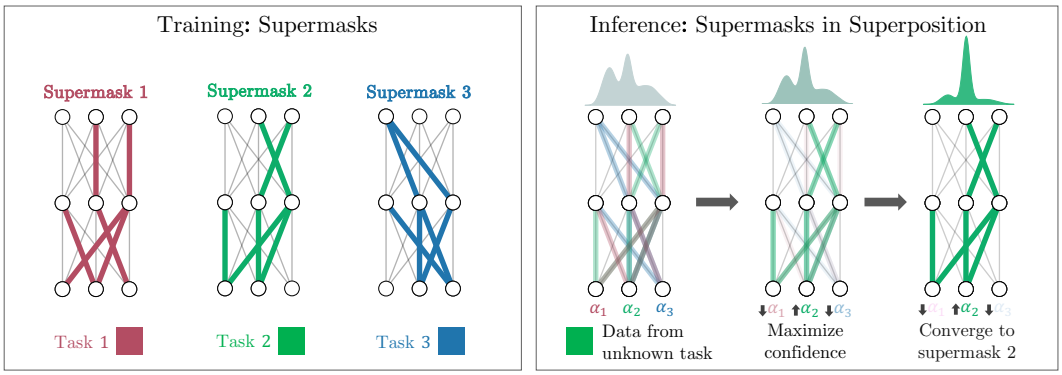

Figure 1: **(left)** During training SupSup learns a separate supermask (subnetwork) for each task. **(right)** At inference time, SupSup can infer task identity by superimposing all supermasks, each weighted by an α_i, and using gradients to maximize confidence.

to recover and use the supermask originally trained for task j. This supermask should exhibit a confident (*i.e.* low entropy) output distribution when given data from task j [19], so we frame inference of task-identity as an optimization problem—find the convex combination of learned supermasks which minimizes the entropy of the output distribution.

In the rest of the paper we develop and evaluate SupSup via the following contributions:

1. We propose a new taxonomy of continual learning scenarios. We use it to embed and contextualize related work (Section 2).
2. When task identity (ID) is provided during train and test (later dubbed GG), SupSup is a natural extension of Mallya *et al.* [30]. By using a randomly weighted backbone and controlling mask sparsity, SupSup surpasses recent baselines on SplitImageNet [51] while requiring less storage and time costs (Section 3.2).
3. When task ID is provided during train but not test (later dubbed GN), SupSup outperforms recent methods that require task ID [26, 23, 4], scaling to 2500 permutations of MNIST without forgetting. For these uniform tasks, ID can be inferred with a single gradient computation (Section 3.3).
4. When task identities are not provided at all (later dubbed NNs), SupSup can even infer task boundaries and allocate new supermasks as needed (Section 3.4).
5. We introduce an extension to the basic SupSup algorithm that stores supermasks implicitly as attractors in a fixed-size Hopfield network [20] (Section 3.5).
6. Finally, we empirically show that the simple trick of adding *superfluous neurons* results in more accurate task inference (Section 3.6).

2 Continual Learning Scenarios and Related Work

In continual learning, a model aims to solve a number of tasks sequentially [47, 56] without catastrophic forgetting [10, 23, 33]. Although numerous approaches have been proposed in the context of continual learning, there lacks a convention of scenarios in which methods are trained and evaluated [49]. The key identifiers of scenarios include: **1)** whether task identity is provided during training, **2)** provided during inference, **3)** whether class labels are shared during evaluation, and **4)** whether the overall task space is discrete or continuous. This results in an exhaustive set of 16 possibilities, many of which are invalid or uninteresting. For example, if task identity is never provided in training, providing it in inference is no longer helpful. To that end, we highlight four applicable scenarios, each with a further breakdown of discrete vs. continuous, when applicable, as shown in Table 1.

We decompose continual learning scenarios via a three-letter taxonomy that explicitly addresses the three most critical scenario variations. The first two letters specify whether task identity is given during training (G if given, N if not) and during inference (G if given, N if not). The third letter specifies a subtle but important distinction: whether labels are shared (s) across tasks or not (u). In the unshared case, the model must predict both the correct task ID and the correct class within that

Table 1: Overview of different Continual Learning scenarios. We suggest scenario names that provide an intuitive understanding of the variations in training, inference, and evaluation, while allowing a full coverage of the scenarios previously defined in [49] and [55]. See text for more complete description.

Scenario	Description	Task space discreet or continuous?	Example methods / task names used
GG	Task **G**iven during train and **G**iven during inference	Either	PNN [42], BatchE [51], PSP [4], "Task learning" [55], "Task-IL" [49]
GNs	Task **G**iven during train, **N**ot inference; **s**hared labels	Either	EWC [23], SI [54], "Domain learning" [55], "Domain-IL" [49]
GNu	Task **G**iven during train, **N**ot inference; **u**nshared labels	Discrete only	"Class learning" [55], "Class-IL" [49]
NNs	Task **N**ot given during train **N**or inference; **s**hared labels	Either	BGD, "Continuous/discrete task agnostic learning" [55]

task. In the shared case, the model need only predict the correct, shared label across tasks, so it need not represent or predict which task the data came from. For example, when learning 5 permutations of MNIST in the GN scenario (task IDs given during train but not test), a shared label GNs scenario will evaluate the model on the correct predicted label across 10 possibilities, while in the unshared GNu case the model must predict across 50 possibilities, a more difficult problem.

A full expansion of possibilities entails both GGs and GGu, but as s and u describe only model *evaluation*, any model capable of predicting shared labels can predict unshared equally well using the provided task ID at test time. Thus these cases are equivalent, and we designate both GG. Moreover, the NNu scenario is invalid because unseen labels signal the presence of a new task (the "labels trick" in [55]), making the scenario actually GNu, and so we consider only the shared label case NNs.

We leave out the discrete vs. continuous distinction as most research efforts operate within one framework or the other, and the taxonomy applies equivalently to discrete domains with integer "Task IDs" as to continue domains with "Task Embedding" or "Task Context" vectors. The remainder of this paper follows the majority of extant literature in focusing on the case with discrete task boundaries (see e.g. [55] for progress in the continuous scenario). Equipped with this taxonomy, we review three existing approaches for continual learning.

(1) Regularization based methods Methods like Elastic Weight Consolidation (EWC) [23] and Synaptic Intelligence (SI) [54] penalize the movement of parameters that are important for solving previous tasks in order to mitigate catastrophic forgetting. Measures of parameter importance vary; e.g. EWC uses the Fisher Information matrix [36]. These methods operate in the GNs scenario (Table 1). Regularization approaches ameliorate but do not exactly eliminate catastrophic forgetting.

(2) Using exemplars, replay, or generative models These methods aim to explicitly or implicitly (with generative models) capture data from previous tasks. For instance, [40] performs classification based on the nearest-mean-of-examplars in a feature space. Additionally, [27, 3] prevent the model from increasing loss on examples from previous tasks while [41] and [45] respectively use memory buffers and generative models to replay past data. Exact replay of the entire dataset can trivially eliminate catastrophic forgetting but at great time and memory cost. Generative approaches can reduce catastrophic forgetting, but generators are also susceptible to forgetting. Recently, [50] successfully mitigate this obstacle by parameterizing a generator with a hypernetwork [15].

(3) Task-specific model components Instead of modifying the learning objective or replaying data, various methods [42, 53, 31, 30, 32, 52, 4, 11, 51] use different model components for different tasks. In Progressive Neural Networks (PNN), Dynamically Expandable Networks (DEN), and Reinforced Continual Learning (RCL) [42, 53, 52], the model is expanded for each new task. More efficiently, [32] fixes the network size and randomly assigns which nodes are active for a given task. In [31, 11], the weights of disjoint subnetworks are trained for each new task. Instead of learning the weights of the subnetwork, for each new task Mallya *et al.* [30] learn a binary mask that is applied to a network pretrained on ImageNet. Recently, Cheung *et al.* [4] superimpose many models into one by using different (and nearly orthogonal) contexts for each task. The task parameters can then be effectively retrieved using the correct task context. Finally, BatchE [51] learns a shared weight matrix on the first task and learn only a rank-one elementwise scaling matrix for each subsequent task.

Our method falls into this final approach (3) as it introduces task-specific supermasks. However, while all other methods in this category are limited to the GG scenario, SupSup can be used to achieve compelling performance in *all four scenarios*. We compare primarily with BatchE [51] and Parameter Superposition (abbreviated PSP) [4] as they are recent and performative. BatchE requires very few additional parameters for each new task while achieving comparable performance to PNN and scaling to SplitImagenet. Moreover, PSP outperforms regularization based approaches like SI [54]. However,

Algorithm	Avg Top 1 Accuracy (%)	Bytes
Upper Bound	92.55	10222.81M
SupSup (GG)	89.58	195.18M
	88.68	100.98M
	86.37	65.50M
BatchE (GG)	81.50	124.99M
Single Model	-	102.23M

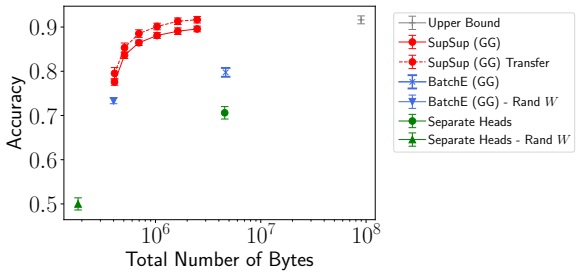

Figure 2: **(left) SplitImagenet** performance in Scenario GG. SupSup approaches upper bound performance with significantly fewer bytes. **(right) SplitCIFAR100** performance in Scenario GG shown as mean and standard deviation over 5 seed and splits. SupSup outperforms similar size baselines and benefits from *transfer*.

both BatchE [51] and PSP [4] require task identity to use task-specific weights, so they can only operate in the GG setting.

3 Methods

In this section, we detail how SupSup leverages supermasks to learn thousands of sequential tasks without forgetting. We begin with easier settings where task identity is given and gradually move to more challenging scenarios where task identity is unavailable.

3.1 Preliminaries

In a standard ℓ-way classification task, inputs $\mathbf{x}$ are mapped to a distribution $\mathbf{p}$ over output neurons $\{1, ..., \ell\}$. We consider the general case where $\mathbf{p} = f(\mathbf{x}, W)$ for a neural network f parameterized by W and trained with a cross-entropy loss. In continual learning classification settings we have k different ℓ-way classification tasks and the input size remains constant across tasks[2].

Zhou *et al.* [57] demonstrate that a trained binary mask (supermask) M can be applied to a randomly weighted neural network, resulting in a subnetwork with good performance. As further explored by Ramanujan *et al.* [39], supermasks can be trained at similar compute cost to training weights while achieving performance competitive with weight training.

With supermasks, outputs are given by $\mathbf{p} = f\left(\mathbf{x}, W \odot M\right)$ where $\odot$ denotes an elementwise product. W is kept frozen at its initialization: bias terms are $\mathbf{0}$ and other parameters in W are $\pm c$ with equal probability and c is the standard deviation of the corresponding Kaiming normal distribution [17]. This initialization is referred to as *signed Kaiming constant* by [39] and the constant c may be different for each layer. For completeness we detail the Edge-Popup algorithm for training supermasks [39] in Section E of the appendix.

3.2 Scenario GG: Task Identity Information Given During Train and Inference

When task identity is known during training we can learn a binary mask M^i per task. M^i are the only parameters learned as the weights remain fixed. Given data from task i, outputs are computed as

$$\mathbf{p} = f\left(\mathbf{x}, W \odot M^i\right) \tag{1}$$

For each new task we can either initialize a new supermask randomly, or use a running mean of all supermasks learned so far. During inference for task i we then use M^i. Figure 2 illustrates that in this scenario SupSup outperforms a number of baselines in accuracy on both SplitCIFAR100 and SplitImageNet while requiring fewer bytes to store. Experiment details are in Section 4.1.

3.3 Scenarios GNs & GNu : Task Identity Information Given During Train Only

We now consider the case where input data comes from task j, but this task information is unknown to the model at inference time. During training we proceed exactly as in Scenario GG, obtaining k

[2]In practice the tasks do not all need to be ℓ-way — output layers can be padded until all have the same size.

learned supermasks. During inference, we aim to infer task identity—correctly detect that the data belongs to task j—and select the corresponding supermask M^j.

The SupSup procedure for task ID inference is as follows: first we associate each of the k learned supermasks M^i with an coefficient $\alpha_i \in [0, 1]$, initially set to $1/k$. Each α_i can be interpreted as the "belief" that supermask M^i is the correct mask (equivalently the belief that the current unknown task is task i). The model's output is then be computed with a weighted superposition of all learned masks:

$$\mathbf{p}(\alpha) = f\left(\mathbf{x}, W \odot \left(\sum_{i=1}^{k} \alpha_i M^i\right)\right). \tag{2}$$

The correct mask M^j should produce a confident, low-entropy output [19]. Therefore, to recover the correct mask we find the coefficients α which minimize the output entropy $\mathcal{H}$ of $\mathbf{p}(\alpha)$. One option is to perform gradient descent on α via

$$\alpha \leftarrow \alpha - \eta \nabla_\alpha \mathcal{H}\left(\mathbf{p}\left(\alpha\right)\right) \tag{3}$$

where η is the step size, and αs are re-normalized to sum to one after each update. Another option is to try each mask individually and pick the one with the lowest entropy output requiring k forward passes. However, we want an optimization method with fixed sub-linear run time (w.r.t. the number of tasks k) which leads α to a corner of the probability simplex — *i.e.* α is 0 everywhere except for a single 1. We can then take the nonzero index to be the inferred task. To this end we consider the **One-Shot** and **Binary** algorithms.

One-Shot: The task is inferred using a single gradient. Specifically, the inferred task is given by

$$\arg\max_i \left(-\frac{\partial \mathcal{H}\left(\mathbf{p}\left(\alpha\right)\right)}{\partial \alpha_i}\right) \tag{4}$$

as entropy is decreasing maximally in this coordinate. This algorithms corresponds to one step of the Frank-Wolfe algorithm [7], or one-step of gradient descent followed by softmax re-normalization with the step size η approaching ∞. Unless noted otherwise, $\mathbf{x}$ is a single image and not a batch.

Binary: Resembling binary search, we infer task identity using an algorithm with $\log k$ steps. At each step we rule out half the tasks—the tasks corresponding to entries in the bottom half of $-\nabla_\alpha \mathcal{H}\left(\mathbf{p}\left(\alpha\right)\right)$. These are the coordinates in which entropy is minimally decreasing. A task i is ruled out by setting α_i to zero and at each step we re-normalize the remaining entries in α so that they sum to one. Pseudo-code for both algorithms may be found in Section A of the appendix.

Once the task is inferred the corresponding mask can be used as in Equation 1 to obtain class probabilities $\mathbf{p}$. In both Scenario GNs and GNu the class probabilities $\mathbf{p}$ are returned. In GNu, $\mathbf{p}$ forms a distribution over the classes corresponding to the inferred task. Experiments solving thousands of tasks are detailed in Section 4.2.

3.4 Scenario NNs: No Task Identity During Training or Inference

Task inference algorithms from Scenario GN enable the extension of SupSup to Scenario NNs, where task identity is entirely unknown (even during training). If SupSup is uncertain about the current task identity, it is likely that the data do not belong to any task seen so far. When this occurs a new supermask is allocated, and k (the number of tasks learned so far) is incremented.

We consider the **One-Shot** algorithm and say that SupSup is uncertain when performing task identity inference if $\nu = \texttt{softmax}\left(-\nabla_\alpha \mathcal{H}\left(\mathbf{p}\left(\alpha\right)\right)\right)$ is approximately uniform. Specifically, if $k \max_i \nu_i < 1 + \epsilon$ a new mask is allocated and k is incremented. Otherwise mask $\arg\max_i \nu_i$ is used, which corresponds to Equation 4. We conduct experiments on learning up to 2500 tasks entirely without any task information, detailed in Section 4.3. Figure 4 shows that SupSup in Scenario NNs achieves comparable performance even to Scenario GNu.

3.5 Beyond Linear Memory Dependence

Hopfield networks [20] implicitly encode a series of binary strings $\mathbf{z}^i \in \{-1, 1\}^d$ with an associated energy function $E_\Psi(\mathbf{z}) = \sum_{uv} \Psi_{uv} \mathbf{z}_u \mathbf{z}_v$. Each $\mathbf{z}^i$ is a minima of E_Ψ, and can be recovered with gradient descent. $\Psi \in \mathbb{R}^{d \times d}$ is initially $\mathbf{0}$, and to encode a new string z^i, $\Psi \leftarrow \Psi + \frac{1}{d}\mathbf{z}^i \mathbf{z}^{i\top}$.

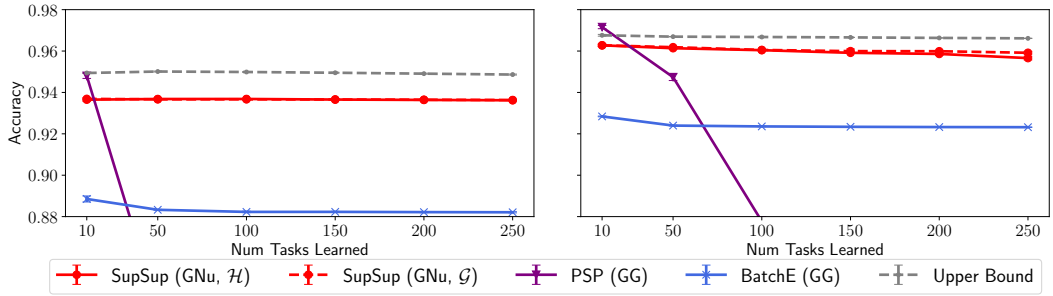

Figure 3: Using **One-Shot** to infer task identity, SupSup outperforms methods with access to task identity. Results shown for PermutedMNIST with LeNet 300-100 **(left)** and FC 1024-1024 **(right)**.

We now consider implicitly encoding the masks in a fixed-size Hopfield network Ψ for Scenario GNu. For a new task i a new mask is learned. After training on task i, this mask will be stored as an attractor in a fixed size Hopfield network. Given new data during inference we perform gradient descent on the Hopfield energy E_Ψ with the output entropy $\mathcal{H}$ to learn a new mask $\mathbf{m}$. Minimizing E_Ψ will hopefully push $\mathbf{m}$ towards a mask learned during training while $\mathcal{H}$ will push $\mathbf{m}$ to be the correct mask. As Ψ is quadratic in mask size, we will not mask the parameters W. Instead we mask the output of every layer except the last, *e.g.* a network with one hidden layer and mask $\mathbf{m}$ is given by

$$f(\mathbf{x}, \mathbf{m}, W) = \texttt{softmax}\left(W_2^\top \left(\mathbf{m} \odot \sigma\left(W_1^\top \mathbf{x}\right)\right)\right) \tag{5}$$

for nonlinearity σ. The Hopfield network will then be a similar size as the base neural network. We refer to this method as HopSupSup and provide additional details in Section B.

3.6 Superfluous Neurons & an Entropy Alternative

Similar to previous methods [49], HopSupSup requires ℓk output neurons in Scenario GNu. SupSup, however, is performing ℓk-way classification without ℓk output neurons. Given data during inference **1)** the task is inferred and **2)** the corresponding mask is used to obtain outputs $\mathbf{p}$. The class probabilities $\mathbf{p}$ correspond to the classes for the inferred task, effectively reusing the neurons in the final layer.

SupSup could use an output size of ℓ, though we find in practice that it helps significantly to add extra neurons to the final layer. Specifically we consider outputs $\mathbf{p} \in \mathbb{R}^s$ and refer to the neurons $\{\ell+1, ..., s\}$ as superfluous neurons (s-neurons). The standard cross-entropy loss will push the values of s-neurons down throughout training. Accordingly, we consider an objective $\mathcal{G}$ which encourages the s-neurons to have large negative values and can be used as an alternative to entropy in Equation 4. Given data from task j, mask M^j will minimize the values of the s-neurons as it was trained to do. Other masks were also trained to minimize the values of the s-neurons, but not for data from task j. In Lemma 1 of Section I we provide the exact form of $\mathcal{G}$ in code ($\mathcal{G} = \texttt{logsumexp}\,(\mathbf{p})$ with masked gradients for $\mathbf{p}_1, ..., \mathbf{p}_\ell$) and offer an alternative perspective on why $\mathcal{G}$ is effective — the gradient of $\mathcal{G}$ for all s-neurons exactly mirrors the gradient from the supervised training loss.

4 Experiments

4.1 Scenario GG: Task Identity Information Given During Train and Inference

Datasets, Models & Training In this experiment we validate the performance of SupSup on SplitCIFAR100 and SplitImageNet. Following Wen *et al.* [51], SplitCIFAR100 randomly partitions CIFAR100 [24] into 20 different 5-way classification problems. Similarly, SplitImageNet randomly splits the ImageNet [5] dataset into 100 different 10-way classification tasks. Following [51] we use a ResNet-18 with fewer channels for SplitCIFAR100 and a standard ResNet-50 [18] for SplitImageNet. The Edge-Popup algorithm from [39] is used to obtain supermasks for various sparsities with a layer-wise budget from [35]. We either initialize each new mask randomly (as in [39]) or use a running mean of all previous learned masks. This simple method of "Transfer" works very well, as illustrated by Figure 2. Additional training details and hyperparameters are provided in Section D.

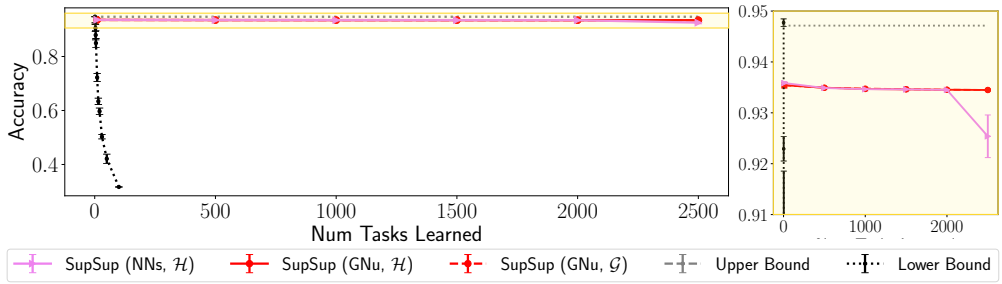

Figure 4: Learning 2500 tasks and inferring task identity using the **One-Shot** algorithm. Results for both the GNu and NNs scenarios with the LeNet 300-100 model using output size 500.

Computation In Scenario GG, the primary advantage of SupSup from Mallya *et al.* [31] or Wen *et al.* [51] is that SupSup does not require the base model W to be stored. Since W is random it suffices to store only the random seed. For a fair comparison we also train BatchE [51] with random weights. The sparse supermasks are stored in the standard `scipy.sparse.csc`[3] format with 16 bit integers. Moreover, SupSup requires minimal overhead in terms of forwards pass compute. Elementwise product by a binary mask can be implemented via memory access, *i.e.* selecting indices. Modern GPUs have very high memory bandwidth so the time cost of this operation is small with respect to the time of a forward pass. In particular, on a 1080 Ti this operation requires $\sim 1\%$ of the forward pass time for a ResNet-50, less than the overhead of BatchE (computation in Section D).

Baselines In Figure 2, for "Separate Heads" we train different heads for each task using a *trunk* (all layers except the final layer) trained on the first task. In contrast "Separate Heads - Rand W" uses a random trunk. BatchE results are given with the trunk trained on the first task (as in [51]) and random weights W. For "Upper Bound", individual models are trained for each task. Furthermore, the trunk for task i is trained on tasks $1, ..., i$. For "Lower Bound" a shared trunk of the network is trained continuously and a separate head is trained for each task. Since catastrophic forgetting occurs we omit "Lower Bound" from Figure 2 (the SplitCIFAR100 accuracy is 24.5%).

4.2 Scenarios GNs & GNu: Task Identity Information Given During Train Only

Our solutions for GNs and GNu are very similar. Because GNu is strictly more difficult, we focus on only evaluating in Scenario GNu. For relevant figures we provide a corresponding table in Section H.

Datasets Experiments are conducted on PermutedMNIST, RotatedMNIST, and SplitMNIST. For PermutedMNIST [23], new tasks are created with a fixed random permutation of the pixels of MNIST. For RotatedMNIST, images are rotated by 10 degrees to form a new task with 36 tasks in total (similar to [4]). Finally SplitMNIST partitions MNIST into 5 different 2-way classification tasks, each containing consecutive classes from the original dataset.

Training We consider two architectures: **1)** a fully connected network with two hidden layers of size 1024 (denoted FC 1024-1024 and used in [4]) **2)** the LeNet 300-100 architecture [25] as used in [8, 6]. For each task we train for 1000 batches of size 128 using the RMSProp optimizer [48] with learning rate 0.0001 which follows the hyperparameters of [4]. Supermasks are found using the algorithm of Mallya *et al.* [31] with threshold value 0. However, we initialize the real valued "scores" with Kaiming uniform as in [39]. Training the mask is not a focus of this work, we choose this method as it is fast and we are not concerned about controlling mask sparsity as in Section 4.1.

Evaluation At test time we perform inference of task identity once for each batch. If task is not inferred correctly then accuracy is 0 for the batch. Unless noted otherwise we showcase results for the most challenging scenario — when the task identity is inferred using a single image. We use "Full Batch" to indicate that all 128 images are used to infer task identity. Moreover, we experiment with both the the entropy $\mathcal{H}$ and $\mathcal{G}$ (Section 3.6) objectives to perform task identity inference.

Results Figure 4 illustrates that SupSup is able to sequentially learn 2500 permutations of MNIST—SupSup succeeds in performing 25,000-way classification. This experiment is conducted with the **One-Shot** algorithm (requiring one gradient computation) using single images to infer task identity. The same trends hold in Figure 3, where SupSup outperforms methods which operate in Scenario GG

[3] `https://docs.scipy.org/doc/scipy/reference/sparse.html`

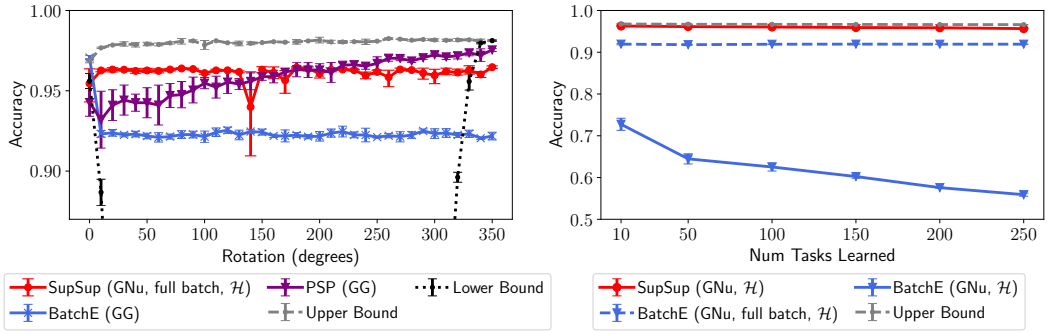

Figure 5: **(left)** Testing the FC 1024-1024 model on RotatedMNIST. SupSup uses **Binary** to infer task identity with a full batch as tasks are similar (differing by only 10 degrees). **(right)** The **One-Shot** algorithm can be used to infer task identity for BatchE [51]. Experiment conducted with FC 1024-1024 on PermutedMNIST using an output size of 500, shown as mean and stddev over 3 runs.

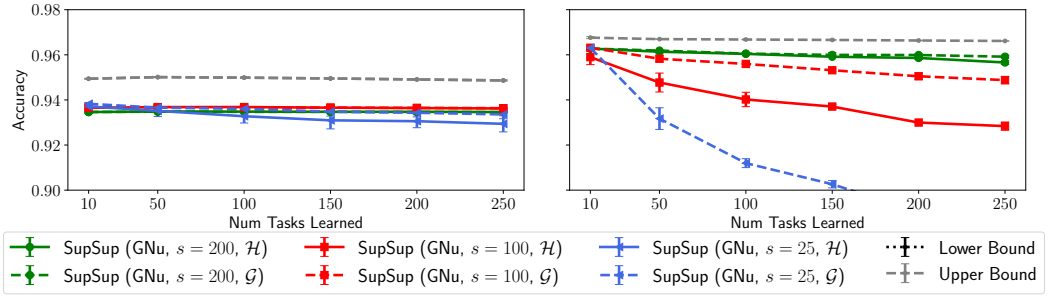

Figure 6: The effect of output size s on SupSup performance using the **One-Shot** algorithm. Results shown for PermutedMNIST with LeNet 300-100 **(left)** and FC 1024-1024 **(right)**.

by using the **One-Shot** algorithm to infer task identity. In Figure 3, output sizes of 100 and 500 are respectively used for LeNet 300-100 and FC 1024-1024. The left hand side of Figure 5 illustrates that SupSup is able to infer task identity even when tasks are similar—SupSup is able to distinguish between rotations of 10 degrees. Since this is a more challenging problem, we use a full batch and the **Binary** algorithm to perform task identity inference. Figure 7 (appendix) shows that for HopSupSup on SplitMNIST, the new mask $\mathbf{m}$ converges to the correct supermask in < 30 gradient steps.

Baselines & Ablations Figure 5 (left) shows that even in Scenario GNu, SupSup is able to outperform PSP [4] and BatchE [51] in Scenario GG—methods using task identity. We compare SupSup in GNu with methods in this strictly easier scenario as they are more competitive. For instance, [49] considers sequential learning problems with only 5-10 tasks. SupSup, after sequentially learning 250 permutations of MNIST, outperforms all non-replay methods from [3] in the GNu scenario after they have learned only 10 permutations of MNIST with a similar network. In GNu, Online EWC achieves 33.88% & SI achieves 29.31% on 10 permutations of MNIST [49] while SupSup achieves 94.91% accuracy after 250 permutations (see Table 5 in [49] vs. Table 7).

In Figure 5 (right) we equip BatchE with task inference using our **One-Shot** algorithm. Instead of attaching a weight α_i to each supermask, we attach a weight α_i to each rank-one matrix [51]. Moreover, in Section C of the appendix we augment BatchE to perform task-inference using large batch sizes. "Upper Bound" and "Lower Bound" are the same as in Section 4.1. Moreover, Figure 6 illustrates the importance of output size. Further investigation of this phenomena is provided by Section 3.6 and Lemma 1 of Section I.

4.3 Scenario NNs: No Task Identity During Training or Inference

For the NNs Scenario we consider PermutedMNIST and train on each task for 1000 batches (the model does not have access to this iteration number). Every 100 batches the model must choose to allocate a new mask or pick an existing mask using the criteria from Section 3.4 ($\epsilon = 2^{-3}$). Figure 4 illustrates that without access to any task identity (even during training) SupSup is able to learn thousands of tasks. However, a final dip is observed as a budget of 2500 supermasks total is enforced.

5 Conclusion

Supermasks in Superposition (SupSup) is a flexible and compelling model applicable to a wide range of scenarios in Continual Learning. SupSup leverages the power of subnetworks [57, 39, 31], and gradient-based optimization to infer task identity when unknown. SupSup achieves state-of-the-art performance on SplitImageNet when given task identity, and performs well on thousands of permutations and almost indiscernible rotations of MNIST without any task information.

We observe limitations in applying SupSup with task identity inference to non-uniform and more challenging problems. Task inference fails when models are not well calibrated—are overly confident for the wrong task. As future work, we hope to explore automatic task inference with more calibrated models [14], as well as circumventing calibration challenges by using optimization objectives such as self-supervision [16] and energy based models [13]. In doing so, we hope to tackle large-scale problems in Scenarios GN and NNs.

Broader Impact

A goal of continual learning is to solve many tasks with a single model. However, it is not exactly clear what qualifies as a *single model*. Therefore, a concrete objective has become to learn many tasks as efficiently as possible. We believe that SupSup is a useful step in this direction. However, there are consequences to more efficient models, both positive and negative.

We begin with the positive consequences:

- Efficient models require less compute, and are therefore less harmful for the environment then learning one model per task [44]. This is especially true if models are able to leverage information from past tasks, and training on new tasks is then faster.
- Efficient models may be run on the end device. This helps to preserve privacy as a user's data does not have to be sent to the cloud for computation.
- If models are more efficient then large scale research is not limited to wealthier institutions. These institutions are more likely in privileged parts of the world and may be ignorant of problems facing developing nations. Moreover, privileged institutions may not be a representative sample of the research community.

We would also like to highlight and discuss the negative consequences of models which can efficiently learn many tasks, and efficient models in general. When models are more efficient, they are also more available and less subject to regularization and study as a result. For instance, when a high-impact model is released by an institution it will hopefully be accompanied by a Model Card [34] analyzing the bias and intended use of the model. By contrast, if anyone is able to train a powerful model this may no longer be the case, resulting in a proliferation of models with harmful biases or intended use. Taking the United States for instance, bias can be harmful as models show disproportionately more errors for already marginalized groups [2], furthering existing and deeply rooted structural racism.

Acknowledgments

We thank Gabriel Ilharco Magalhães and Sarah Pratt for helpful comments. For valuable conversations we also thank Tim Dettmers, Kiana Ehsani, Ana Marasović, Suchin Gururangan, Zoe Steine-Hanson, Connor Shorten, Samir Yitzhak Gadre, Samuel McKinney and Kishanee Haththotuwegama. This work is in part supported by NSF IIS 1652052, IIS 17303166, DARPA N66001-19-2-4031, DARPA W911NF-15-1-0543 and gifts from Allen Institute for Artificial Intelligence. Additional revenues: co-authors had employment with the Allen Institute for AI.

References

[1] Yoshua Bengio, Nicolas L Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte. Convex neural networks. In *Advances in neural information processing systems*, pages 123–130, 2006.

[2] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on fairness, accountability and transparency*, pages 77–91, 2018.

[3] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with a-gem. *arXiv preprint arXiv:1812.00420*, 2018.

[4] Brian Cheung, Alexander Terekhov, Yubei Chen, Pulkit Agrawal, and Bruno Olshausen. Superposition of many models into one. In *Advances in Neural Information Processing Systems*, pages 10867–10876, 2019.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR 2009*, 2009.

[6] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[7] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval research logistics quarterly*, 3(1-2):95–110, 1956.

[8] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[9] Jonathan Frankle, David J Schwab, and Ari S Morcos. Training batchnorm and only batchnorm: On the expressive power of random features in cnns. *arXiv preprint arXiv:2003.00152*, 2020.

[10] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[11] Siavash Golkar, Michael Kagan, and Kyunghyun Cho. Continual learning via neural pruning. *arXiv preprint arXiv:1903.04476*, 2019.

[12] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[13] Will Grathwohl, Kuan-Chieh Wang, Jörn-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. Your classifier is secretly an energy based model and you should treat it like one. *arXiv preprint arXiv:1912.03263*, 2019.

[14] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org, 2017.

[15] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[16] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*, 2019.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136*, 2016.

[20] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[25] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[26] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. 2010.

[27] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, pages 6467–6476, 2017.

[28] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2016.

[29] Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. *arXiv preprint arXiv:2002.00585*, 2020.

[30] Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 67–82, 2018.

[31] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[32] Nicolas Y Masse, Gregory D Grant, and David J Freedman. Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. *Proceedings of the National Academy of Sciences*, 115(44):E10467–E10475, 2018.

[33] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

[34] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 220–229, 2019.

[35] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

[36] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[38] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[39] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? *arXiv preprint arXiv:1911.13299*, 2019.

[40] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010, 2017.

[41] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *Advances in Neural Information Processing Systems*, pages 348–358, 2019.

[42] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[43] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th european symposium on artificial neural networks. p. 471-482 2007*, pages 471–482, 2007.

[44] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. corr abs/1907.10597 (2019). *arXiv preprint arXiv:1907.10597*, 2019.

[45] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, pages 2990–2999, 2017.

[46] Amos Storkey. Increasing the capacity of a hopfield network without sacrificing functionality. In *International Conference on Artificial Neural Networks*, pages 451–456. Springer, 1997.

[47] Sebastian Thrun. Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer, 1998.

[48] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[49] Gido M van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

[50] Johannes von Oswald, Christian Henning, João Sacramento, and Benjamin F. Grewe. Continual learning with hypernetworks. In *International Conference on Learning Representations*, 2020.

[51] Yeming Wen, Dustin Tran, and Jimmy Ba. Batchensemble: an alternative approach to efficient ensemble and lifelong learning. *arXiv preprint arXiv:2002.06715*, 2020.

[52] Ju Xu and Zhanxing Zhu. Reinforced continual learning. In *Advances in Neural Information Processing Systems*, pages 899–908, 2018.

[53] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

[54] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3987–3995. JMLR. org, 2017.

[55] Chen Zeno, Itay Golan, Elad Hoffer, and Daniel Soudry. Task agnostic continual learning using online variational bayes. *arXiv preprint arXiv:1803.10123*, 2018.

[56] Jieyu Zhao and Jurgen Schmidhuber. Incremental self-improvement for life-time multi-agent reinforcement learning. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 516–525, 1996.

[57] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, pages 3592–3602, 2019.

Algorithm 1 One-Shot$(f, \mathbf{x}, W, k, \{M^i\}_{i=1}^k, \mathcal{H})$

$\alpha \leftarrow \begin{bmatrix} \frac{1}{k} & \frac{1}{k} & ... & \frac{1}{k} \end{bmatrix}$ ▷ Initialize α

$\mathbf{p} \leftarrow f\left(\mathbf{x}, W \odot \left(\sum_{i=1}^k \alpha_i M^i\right)\right)$ ▷ Superimposed output

return $\arg\max_i \left(-\frac{\partial \mathcal{H}(\mathbf{p})}{\partial \alpha_i}\right)$ ▷ Return coordinate for which objective maximally decreasing

Algorithm 2 Binary$(f, \mathbf{x}, W, k, \{M^i\}_{i=1}^k, \mathcal{H})$

$\alpha \leftarrow \begin{bmatrix} \frac{1}{k} & \frac{1}{k} & ... & \frac{1}{k} \end{bmatrix}$ ▷ Initialize α

while $\|\alpha\|_0 > 1$ **do** ▷ Iterate until α has a single nonzero entry

  $\mathbf{p} \leftarrow f\left(\mathbf{x}, W \odot \left(\sum_{i=1}^k \alpha_i M^i\right)\right)$ ▷ Superimposed output

  $g \leftarrow -\nabla_\alpha \mathcal{H}(\mathbf{p})$ ▷ Gradient of objective

  for $i \in \{1, ..., k\}$ **do** ▷ In code this **for** loop is vectorized

    if $g_i \leq$ **median** (g) **then**

      $\alpha_i \leftarrow 0$ ▷ Zero out α_i for which objective minimally decreasing

  $\alpha \leftarrow \alpha / \|\alpha\|_1$ ▷ Re-normalize α to sum to 1

return $\arg\max_i \alpha_i$

A Algorithm pseudo-code

Algorithms 1 and 2 respectively provide pseudo-code for the **One-Shot** and **Binary** algorithms detailed in Section 3.3. Both aim to infer the task $j \in \{1, ..., k\}$ associated with input data $\mathbf{x}$ by minimizing the objective $\mathcal{H}$.

B Extended Details for HopSupSup

This section provides further details and experiments for HopSupSup (introduced in Section 3.5). HopSupSup provides a method for storing the growing set of supermasks in a fixed size reservoir instead of explicitly storing each mask.

B.1 Training

Recall that HopSupSup operates in Scenario GNu and so task identity is known during training. Instead of explicitly storing each mask, we will instead store two fixed sized variables Ψ and μ which are both initially $\mathbf{0}$. The weights of the Hopfield network are Ψ and μ stores a running mean of all masks learned so far. For a new task k we use the same algorithm as in Section 4.2 to learn a binary mask $\mathbf{m}^i$ which performs well for task k. Since Hopfield networks consider binary strings in $\{-1, 1\}^d$ and we use masks $\mathbf{m}^i \in \{0, 1\}^d$ we will consider $\mathbf{z}^k = 2\mathbf{m}^k - 1$. In practice we then update Ψ and μ as

$$\Psi \leftarrow \Psi + \frac{1}{d}\left(\mathbf{z}^k {\mathbf{z}^k}^\top - \mathbf{z}^k \left(\Psi \mathbf{z}^k\right)^\top - \left(\Psi \mathbf{z}^k\right) {\mathbf{z}^k}^\top - \mathrm{Id}\right), \qquad \mu \leftarrow \frac{k-1}{k}\mu + \frac{1}{k}\mathbf{z}^k \tag{6}$$

where Id is the identity matrix. This update rule for Ψ is referred to as the Storkey learning rule [46] and is more expressive than the alternative—the Hebbian rule $\Psi \leftarrow \Psi + \frac{1}{d}\mathbf{z}^k {\mathbf{z}^k}^\top$ [20] provided for brevity in Section 3.3. With either update rules the learned $\mathbf{z}^i$ will be a minimizer of the Hopfield energy $E_\Psi(\mathbf{z}) = \sum_{uv} \Psi_{uv} \mathbf{z}_u \mathbf{z}_v$.

B.2 Inference

During inference we receive data $\mathbf{x}$ from some task j, but this task information is not given to the model. HopSupSup first initializes a new binary string $\mathbf{z}$ with μ. Next, HopSupSup uses gradient descent to minimize the Hopfield energy in conjunction with the output entropy using mask $\mathbf{m} = \frac{1}{2}\mathbf{z} + 1$, a process we refer to as *Hopfield Recovery*. Minimizing the energy will hopefully push $\mathbf{m}$ (equivalently $\mathbf{z}$) towards a mask learned during training and minimizing the entropy will hopefully

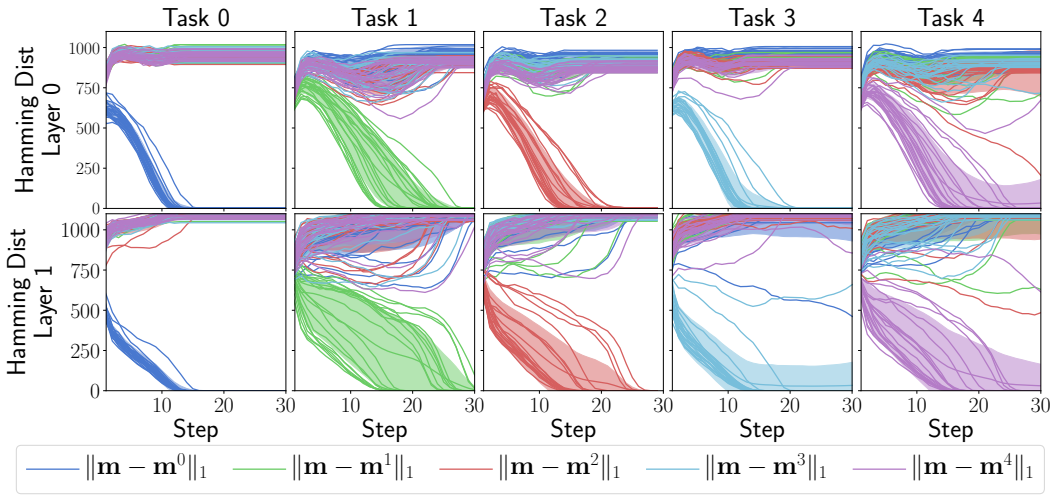

Figure 7: During *Hopfield Recovery* the new mask $\mathbf{m}$ converges to the correct mask learned during training. Note that $\mathbf{m}^i$ denotes the mask learned for task i.

push $\mathbf{m}$ towards the correct mask $\mathbf{m}^j$. We may then use the recovered mask to compute the network output.

In practice we use one pass through the evaluation set (with batch size 64, requiring $T \approx 30$ steps) to recover a mask and another to perform evaluation with the recovered mask. When recovering the mask we gradually increase the strength of the Hopfield term and decrease the strength of the entropy term. Otherwise the Hopfield term initially pulls $\mathbf{z}$ in the wrong direction or the final $\mathbf{z}$ does not lie at a minimum of E_Ψ. For step $t \in \{1, ..., T\}$, and constant γ we use the objective $\mathcal{J}$ as

$$\mathcal{J}(\mathbf{z}, t) = \frac{\gamma t}{T} E_\Psi(\mathbf{z}) + \left(1 - \frac{t}{T}\right) \mathcal{H}(\mathbf{p}) \tag{7}$$

where $\mathbf{p}$ denotes the output using mask $\mathbf{m} = \frac{1}{2}\mathbf{z} + 1$.

Figure 7 illustrates that after approximately 30 steps of gradient descent on $\mathbf{z}$ using objective $\mathcal{J}$, the mask $\mathbf{m} = \frac{1}{2}\mathbf{z} + 1$ converges to the correct mask learned during training. This experiment is conducted for 20 different random seeds on SplitMNIST (see Section 4.2) training for 1 epoch per task. Evaluation with the recovered mask for each seed is then given by Figure 8. As expected, when the correct mask is successfully recovered, accuracy matches directly using the correct mask. For hyperparameters we set $\gamma = 1.5 \cdot 10^{-3}$ and perform gradient descent during Hopfield recovery with learning rate $0.5 \cdot 10^3$, momentum 0.9, and weight decay 10^{-4}.

B.3 Network Architecture

Let BN denote non-affine batch normalization [21], *i.e.* batch normalization with no learned parameters. Also recall that we are masking layer outputs instead of weights, and the weights still remain fixed (see Section 3.5). Therefore, with mask $\mathbf{m} = (\mathbf{m}_1, \mathbf{m}_2)$ and weights $W = (W_1, W_2, W_3)$ we compute outputs as

$$f(\mathbf{x}, \mathbf{m}, W) = \texttt{softmax}\left(W_3^\top \sigma\left(\mathbf{m}_2 \odot \texttt{BN}\left(W_2^\top \sigma\left(\mathbf{m}_1 \odot \texttt{BN}\left(W_1^\top \mathbf{x}\right)\right)\right)\right)\right) \tag{8}$$

where σ denotes the Swish nonlinearity [38]. Without masking or normalization f is a fully connected network with two hidden layers of size 2048. We also note that HopSupSup requires 10 output neurons for SplitMNIST in Scenario GNu, and the composition of non-affine batch normalization with a binary mask was inspired by BatchNets [9].

C Augmenting BatchE For Scnario GNu

In Section 4.2 we demonstrate that BatchE [51] is able to infer task identity using the **One-Shot** algorithm. In this section we show that, equipped with $\mathcal{H}$ from Section 3, BatchE can also infer task identity by using a large batch size. We refer to this method as Augmented BatchE (ABatchE).

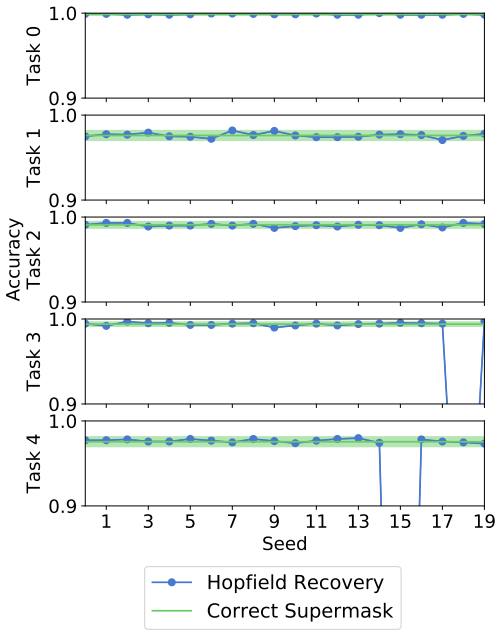

Figure 8: Evaluating (with 20 random seeds) on SplitMNIST after finding a mask with *Hopfield Recovery*. Average accuracy is 97.43%.

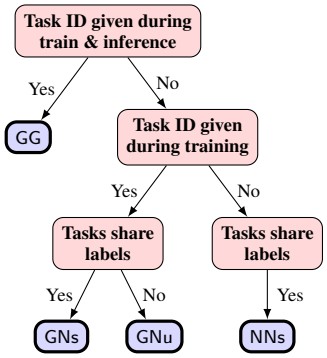

Figure 9: Continual learning scenarios detailed in Table 1 represented in a tree graph, as in [55].

For clarity we describe ABatchE for one linear layer, *i.e.* we describe the application of ABatchE to

$$f(\mathbf{x}, W) = \texttt{softmax}\left(W^\top \mathbf{x}\right) \tag{9}$$

for input data $\mathbf{x} \in \mathbb{R}^m$ and weights $W \in \mathbb{R}^{m \times n}$. In BatchE [51], W is trained on the first task then frozen. For task i BatchE learns "fast weights" $r_i \in \mathbb{R}^m$, $s_i \in \mathbb{R}^n$ and outputs are computed via

$$f(\mathbf{x}, W) = \texttt{softmax}\left(\left(W \odot r_i s_i^\top\right)^\top \mathbf{x}\right). \tag{10}$$

Wen *et al.* [51] further demonstrate that Equation 10 can be vectorized as

$$f(\mathbf{x}, W) = \texttt{softmax}\left(\left(W^\top \left(\mathbf{x} \odot r_i\right)\right) \odot s_i\right) \tag{11}$$

or, for a batch of data $X \in \mathbb{R}^{b \times m}$,

$$f(X, W) = \texttt{softmax}\left(\left(\left(X \odot R_i^b\right) W\right) \odot S_i^b\right). \tag{12}$$

In Equation 12, $R_i^b \in \mathbb{R}^{b \times m}$ is a matrix where each of the b rows is r_i (likewise $S_i^b \in \mathbb{R}^{b \times n}$ is a matrix where each of the b rows is s_i).

As in Section 3.3 we now consider the case where data $X \in \mathbb{R}^{b \times m}$ comes from task j but this information is not known to the model. For ABatchE we repeat the data k times, where k is the number of tasks learned so far, and use different "fast weights" for each repetiton. Specifically, we consider repeated data $\tilde{X} \in \mathbb{R}^{bk \times m}$ and augmented matricies $\tilde{R} \in \mathbb{R}^{bk \times m}$ and $\tilde{S} \in \mathbb{R}^{bk \times n}$ given by

$$\tilde{X} = \begin{bmatrix} X \\ X \\ \vdots \\ X \end{bmatrix}, \quad \tilde{R} = \begin{bmatrix} R_1^b \\ R_2^b \\ \vdots \\ R_k^b \end{bmatrix}, \quad \tilde{S} = \begin{bmatrix} S_1^b \\ S_2^b \\ \vdots \\ S_k^b \end{bmatrix}. \tag{13}$$

Outputs are then computed as

$$f(X, W) = \texttt{softmax}\left(\left(\left(\tilde{X} \odot \tilde{R}\right) W\right) \odot \tilde{S}\right) \tag{14}$$

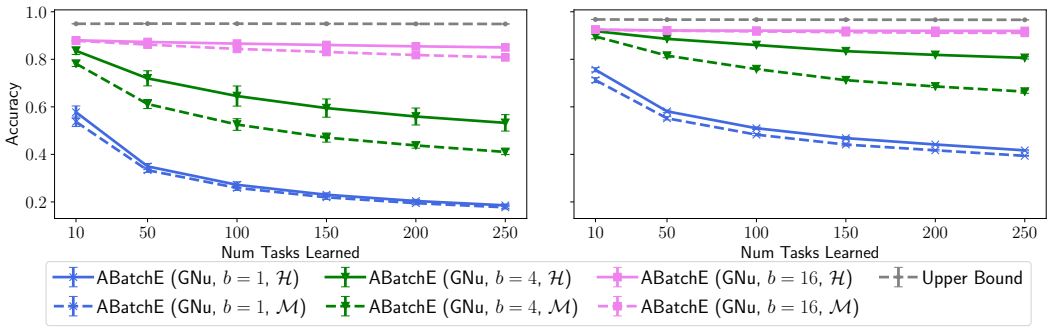

Figure 10: Testing ABatchE on PermutedMNIST with LeNet 300-100 **(left)** and FC 1024-1024 **(right)** with output size 100.

where the b rows $(bi, ..., bi + b - 1)$ of the output correspond exactly to Equation 12. The task may then be inferred by choosing the i for which the rows $(bi, ..., b(i + 1) - 1)$ minimize the objective $\mathcal{H}$. If $f(X, W)_i$ denotes row i of $f(X, W)$ then for objective $\mathcal{H}$ the inferred task for ABatchE is

$$\arg\min_i \sum_{\omega=0}^{b-1} \mathcal{H}\left(f(X, W)_{bi+\omega}\right). \tag{15}$$

To extend ABatchE to deep neural networks the matricies $\tilde{R}$ and $\tilde{S}$ are constructed for each layer.

One advantage of ABatchE over SupSup is that no backwards pass is required. However, ABatchE uses a very large batch size for large k, and the forward pass therefore requires more compute and memory. Another disadvantage of ABatchE is that the performance of ABatchE is limited by the performance of BatchE. In Section 4.2 we demonstrate that SupSup outperforms BatchE when BatchE is given task identity information.

Since the objective for ABatchE need not be differentiable we also experiment with an alternative metric of confidence $\mathcal{M}(\mathbf{p}) = -\max_i \mathbf{p}_i$. We showcase results for ABatchE on PermutedMNIST in Figure 10 for various values of b. The entropy objective $\mathcal{H}$ performs better than $\mathcal{M}$, and forgetting is only mitigated when using 16 images ($b = 16$). With 250 tasks, $b = 16$ corresponds to a batch size of 4000.

D Extended Training Details

D.1 SplitCIFAR-100 (GG)

As in [51] we train each model for 250 epochs per task. We use standard hyperparameters—the Adam optimizer [22] with a batch size of 128 and learning rate 0.001 (no warmup, cosine decay [28]). For SupSup we follow [39] and use non-affine normalization so there are no learned parameters. We do have to store the running mean and variance for each task, which we include in the parameter count. We found it better to use a higher learning rate (0.1) when training BatchE (Rand W), and the standard BatchE number is taken from [51].

D.2 SplitImageNet (GG)

We use the Upper Bound and BatchE number from [51]. For SupSup we train for 100 epochs with a batch size of 256 using the Adam optimizer [22] with learning rate 0.001 (5 epochs warmup, cosine decay [28]). For SupSup we follow [39] and use non-affine normalization so there are no learned parameters. We do have to store the running mean and variance for each task, which we include in the parameter count.

D.3 GNu Experiments

We clarify some experimental details for GNu experiments & baselines. For the BatchE [51] baseline we find it best to use kaiming normal initialization with a learning rate of 0.01 (0.0001 for the first task when the weights are trained). As we are considering hundreds of tasks, instead of training

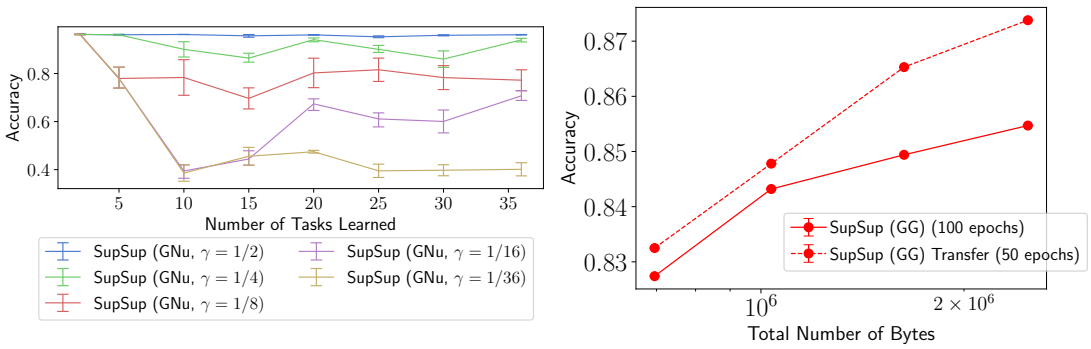

Figure 11: **(left)** Interpolating between the binary and one-shot algorithm with γ. **(right)** Transfer enables faster learning on SplitCIFAR.

separate heads per tasks when training BatchE we also apply the rank one pertubation to the final layer. PSP [4] provides MNISTPerm results so we use the same hyperparameters as in their code. We compare with rotational superposition, the best performing model from PSP.

D.4 Speed of the Masked Forward Pass

We now provide justification for the calculation mentioned in Section 4.1—when implemented properly the masking operation should require $\sim 1\%$ of the total time for a forward pass (for a ResNet-50 on a NVIDIA GTX 1080 Ti GPU). It is reasonable to assume that selecting indices is roughly as quick as memory access. A NVIDIA GTX 1080 Ti has a memory bandwidth of 480 GB/s. A ResNet-50 has around $2.5 \cdot 10^7$ 4-byte (32-bit) parameters—roughly 0.1 GB. Therefore, indexing over a ResNet-50 requires at most $0.1 \text{ GB} / (480 \text{ GB/s}) \approx 0.21$ ms. For comparison, the average forward pass of a ResNet-50 for a $3 \times 224 \times 224$ image on the same GPU is about 25 ms.

Note that NVIDIA hardware specifications generally assume best-case performance with sequential page reads. However, even if real-world memory bandwidth speeds are 60-70% slower than advertised, the fraction of masking time would remain in the $\leq 3\%$ range.

D.5 Additional Transfer Experiment

For our transfer experiments, we initialize the score matrix (see Appendix E) for task i with the running mean of the supermasks for tasks 0 through $i - 1$. The scores for task 0 are initialized as in [39]. We further normalize by the Kaiming fan-in constant from [17], so that the norm of our supermask matrix is reasonable. If we do not perform this normalization, accuracy degrades significantly. All other training hyperparameters are the same as in Section D.1.

In Figure 11, we demonstrate that Transfer enables faster learning for SplitCIFAR. In this experiment, we train task 0 for the full 250 epochs and all subsequent tasks for either 50 epochs (with transfer) or 100 epochs (without transfer). We see that adding transfer yields an improvement even while using about half the number of training iterations overall.

E Supermask Training with Edge-Popup

For completeness we briefly recap the Edge-Popup algorithm for training supermasks as introduced by [39]. Consider a linear layer with inputs $\mathbf{x} \in \mathbb{R}^m$ and outputs $\mathbf{y} = (W \odot M)^\top \mathbf{x}$ where $W \in \mathbb{R}^{m \times n}$ are the fixed weights and $M \in \{0, 1\}^{m \times n}$ is the supermask. The Edge-Popup algorithm learns a score matrix $S \in \mathbb{R}_+^{m \times n}$ and computes the mask via $M = h(S)$. The function h sets the top $k\%$ of entries in S to 1 and the remaining to 0. Edge-Popup updates S via the straight through estimator—h is considered to be the identity on the backwards pass.

F Comparing Binary and One-Shot

In Figure 11 **(left)** we interpolate between the **Binary** and **One-Shot** algorithms. We replace line 6 of Algorithm 2, $g_i \leq$ **median**(g), with $g_i \leq$ **top-**$\gamma\%$**-element**(g). Then when $\gamma = 1/2$ we recover the binary algorithm (as **median**$(g) =$ **top-**50%**-element**(g)) and when $\gamma = 1/k$ we recover the one-shot algorithm. A performance drop is observed from binary to one-shot for the difficult task of MNISTRotate—sequentially learning 36 rotations of MNIST (each rotation differing by 10 degrees).

G Tree Representation for the Continual Learning Scenarios

In Figure 9 the Continual Learning scenarios are represented as a tree. This resembles the formulation from [55] with some modifications, *i.e.* "Tasks share output head?" is replaced with "Tasks share labels" as it is possible to share the output head but not labels, *e.g.* SupSup in GNu.

H Corresponding Tables

In this section we provide tabular results for figures from Section 4.

Table 2: Accuracy on SplitCIFAR100 corresponding to Figure 2 **(right)**. SupSup with Transfer approaches the upper bound.

Entry	Avg Acc@1	Bytes
SupSup (GG)	77.56 $\pm$ 0.73	408432
SupSup (GG)	83.62 $\pm$ 0.74	508432
SupSup (GG)	86.45 $\pm$ 0.61	695592
SupSup (GG)	88.09 $\pm$ 0.64	1035792
SupSup (GG)	89.06 $\pm$ 0.75	1630032
SupSup (GG)	89.57 $\pm$ 0.64	2487472
SupSup (GG) Transfer	79.53 $\pm$ 1.31	408432
SupSup (GG) Transfer	85.33 $\pm$ 1.05	508432
SupSup (GG) Transfer	88.52 $\pm$ 0.85	695592
SupSup (GG) Transfer	90.12 $\pm$ 0.75	1035792
SupSup (GG) Transfer	91.31 $\pm$ 0.74	1630032
SupSup (GG) Transfer	**91.66** $\pm$ 0.74	2487472
BatchE (GG)	79.75 $\pm$ 1.00	4640800
BatchE (GG) - Rand W	74.96 $\pm$ 0.68	400240
Separate Heads	70.60 $\pm$ 1.40	4544560
Separate Heads - Rand W	50.00 $\pm$ 1.37	184000
Upper Bound	91.62 $\pm$ 0.89	89675200

Table 3: Accuracy on PermutedMNIST with LeNet 300-100 corresponding to Figure 3 **(left)**.

Entry	10	50	100	150	200	250	Avg
SupSup (GNu $\mathcal{H}$)	93.65	**93.68**	**93.68**	**93.66**	93.64	93.62	**93.66**
SupSup (GNu $\mathcal{G}$)	93.69	93.67	93.67	**93.66**	**93.65**	**93.63**	**93.66**
PSP (GG)	**94.80**	83.58	64.62	51.18	42.69	36.74	62.27
BatchE (GG)	88.85	88.33	88.23	88.23	88.22	88.21	88.34
Upper Bound	94.94	95.01	94.99	94.95	94.91	94.86	94.94

Table 4: Accuracy on PermutedMNIST with FC 1024-1024 corresponding to Figure 3 **(right)**.

Entry	10	50	100	150	200	250	Avg
SupSup (GNu $\mathcal{H}$)	96.28	96.14	96.04	95.91	95.86	95.66	95.98
SupSup (GNu $\mathcal{G}$)	96.28	**96.19**	**96.05**	**96.00**	**95.99**	**95.92**	**96.07**
PSP (GG)	**97.16**	94.74	87.77	78.35	69.14	61.11	81.38
BatchE (GG)	92.84	92.40	92.36	92.34	92.33	92.32	92.43
Upper Bound	96.76	96.70	96.68	96.66	96.63	96.61	96.67

Table 5: Accuracy on PermutedMNIST with LeNet 300-100 corresponding to Figure 4.

Entry	500	1000	1500	2000	2500	Avg
SupSup (GNu $\mathcal{H}$)	93.49	93.47	93.46	93.45	93.45	93.46
SupSup (GNu $\mathcal{G}$)	93.49	93.48	93.46	93.45	93.45	93.47
SupSup (NNs $\mathcal{H}$)	93.49	93.46	93.46	93.45	92.54	93.28
Upper Bound	94.71	94.71	94.71	94.71	94.71	94.71

Table 6: Accuracy with FC 1024-1024 on RotatedMNIST corresponding to Figure 5 **(left)**.

Entry	Avg
SupSup (GNu full batch $\mathcal{H}$)	**96.13**
BatchE (GG)	92.40
PSP (GG)	95.87
Lower Bound	48.71
Upper Bound	98.01

Table 7: Accuracy with FC 1024-1024 on PermutedMNIST corresponding to Figure 5 **(right)**.

Entry	10	50	100	150	200	250	Avg
SupSup (GNu $\mathcal{H}$)	**96.29**	**95.94**	**95.59**	**95.40**	**95.00**	**94.91**	**95.52**
BatchE (GNu full batch $\mathcal{H}$)	91.94	91.90	92.04	92.04	92.04	92.04	92.00
BatchE (GNu $\mathcal{H}$)	66.08	61.89	60.93	59.33	57.37	55.74	60.22
Upper Bound	96.76	96.70	96.68	96.66	96.63	96.61	96.67

Table 8: Accuracy on PermutedMNIST with LeNet 300-100 corresponding to Figure 6 **(left)**.

Entry	10	50	100	150	200	250	Avg
SupSup (GNu $s = 200$ $\mathcal{H}$)	93.46	93.49	93.48	93.47	93.47	93.46	93.47
SupSup (GNu $s = 200$ $\mathcal{G}$)	93.46	93.48	93.47	93.47	93.47	93.46	93.47
SupSup (GNu $s = 100$ $\mathcal{H}$)	93.65	**93.68**	**93.68**	**93.66**	93.64	93.62	**93.66**
SupSup (GNu $s = 100$ $\mathcal{G}$)	**93.69**	93.67	93.67	**93.66**	**93.65**	**93.63**	**93.66**
SupSup (GNu $s = 25$ $\mathcal{H}$)	93.71	93.51	93.28	93.10	93.06	92.94	93.27
SupSup (GNu $s = 25$ $\mathcal{G}$)	93.83	93.66	93.60	93.48	93.43	93.36	93.56
Lower Bound	71.67	41.82	30.52	26.40	23.31	20.88	35.77
Upper Bound	94.94	95.01	94.99	94.95	94.91	94.86	94.94

Table 9: Accuracy on PermutedMNIST with FC 1024-1024 corresponding to Figure 6 **(right)**.

Entry	10	50	100	150	200	250	Avg
SupSup (GNu $s = 200$ $\mathcal{H}$)	96.28	96.14	96.04	95.91	95.86	95.66	95.98
SupSup (GNu $s = 200$ $\mathcal{G}$)	96.28	**96.19**	**96.05**	**96.00**	**95.99**	**95.92**	**96.07**
SupSup (GNu $s = 100$ $\mathcal{H}$)	95.90	94.77	94.02	93.71	93.00	92.84	94.04
SupSup (GNu $s = 100$ $\mathcal{G}$)	**96.31**	95.83	95.60	95.32	95.05	94.88	95.50
SupSup (GNu $s = 25$ $\mathcal{H}$)	82.28	69.06	64.51	60.99	58.15	57.03	65.34
SupSup (GNu $s = 25$ $\mathcal{G}$)	**96.31**	93.17	91.20	90.26	89.04	88.19	91.36
Lower Bound	76.89	49.40	38.93	34.53	31.30	29.36	43.40
Upper Bound	96.76	96.70	96.68	96.66	96.63	96.61	96.67

I Analysis

In this section we assume a slightly more technical perspective. The aim is not to formally prove properties of the algorithm. Rather, we hope that a more mathematical language may prove useful in extending intuition. Just as the empirical work of [8, 57, 39] was given a formal treatment in [29], we hope for more theoretical work to follow.

Our grounding intuition remains from Section 3.3—the correct mask will produce the lowest entropy output. Moreover, since entropy is differentiable, gradient based optimization can be used to recover the correct mask. However, many questions remain: Why do superfluous neurons (Section 3.6) help? In the case of MNISTPermuation, why is a single gradient sufficient? Although it is a simple case, steps forward can be made by analyzing the training of a linear head on fixed features. With *random* features, training a linear head on fixed features is considered in the literature of reservoir computing [43], and more [1].

Consider k different classification problems with fixed features $\phi(\mathbf{x}) \in \mathbb{R}^m$. Traditionally, one would use learned weights $W \in \mathbb{R}^{m \times n}$ to compute *logits*

$$\mathbf{y} = W^\top \phi(\mathbf{x}) \tag{16}$$

and output classification probabilities $\mathbf{p} = \texttt{softmax}(\mathbf{y})$ where

$$\mathbf{p}_v = \frac{\exp(\mathbf{y}_v)}{\sum_{v'=1}^{n} \exp(\mathbf{y}_{v'})}. \tag{17}$$

Recall that with SupSup we compute the *logits* for task i using fixed random weights W and a learned binary mask $M^i \in \{0,1\}^{m \times n}$ as

$$\mathbf{y} = \left(W \odot M^i\right)^\top \phi(\mathbf{x}) \tag{18}$$

where $\odot$ denotes an element-wise product and no bias term is allowed. Moreover, $W_{uv} = \xi_{uv}\sqrt{2/m}$ where ξ_{uv} is chosen independently to be either -1 or 1 with equal probability and the constant $\sqrt{2/m}$ follows Kaiming initialization [17].

Say we are given data $\mathbf{x}$ from task j. From now on we will refer to task j as the *correct* task. Recall from Section 3.3 that SupSup attempts to infer the *correct* task by using a weighted mixture of masks

$$\mathbf{y} = \left(W \odot \sum_i \alpha_i M^i\right)^\top \phi(\mathbf{x}) \tag{19}$$

where the coefficients α_i sum to one, and are initially set to $1/k$.

To infer the correct task we attempt to construct a function $\mathcal{G}(\mathbf{y}; \alpha)$ with the following property: For fixed data, $\mathcal{G}$ is minimized when $\alpha = \mathbf{e}_j$ ($\mathbf{e}_j$ denotes a k-length vector that is 1 in index j and 0 otherwise). We can then infer the correct task by solving a minimization problem.

As in **One-Shot**, we use a single gradient computation to infer the task via

$$\arg\max_i \left(-\frac{\partial \mathcal{G}}{\partial \alpha_i}\right). \tag{20}$$

A series of Lemmas will reveal how a single gradient step may be sufficient when tasks are unrelated (*e.g.* as in PermutedMNIST). We begin with the construction of a useful function $\mathcal{G}$, which will correspond exactly to $\mathcal{G}$ in Section 3.6. As in Section 3.6, this construction is made possible through superfluous neurons (s-neurons): The true labels are in $\{1, ..., \ell\}$, and a typical output is therefore length ℓ. However, we add $n - \ell$ s-neurons resulting in a vector $\mathbf{y}$ of length n.

Let $\mathbf{S}$ denote the set of s-neurons and $\mathbf{R}$ denote the set of *real* neurons where $|\mathbf{S}| = n - \ell$ and $|\mathbf{R}| = \ell$. Moreover, assume that a standard cross-entropy loss is used during training, which will encourage s-neurons to have small values.

Lemma I.1. *It is possible to construct a function $\mathcal{G}$ such that the gradient matches the gradient from the supervised training loss $\mathcal{L}$ for all s-neurons. Specifically, $\frac{\partial \mathcal{G}}{\partial y_v} = \frac{\partial \mathcal{L}}{\partial y_v}$ for all $v \in \mathbf{S}$ and 0 otherwise.*

Proof. Let $g_v = \frac{\partial \mathcal{G}}{\partial y_v}$. It is easy to ensure that $g_v = 0$ for all $v \notin \mathbf{S}$ with a modern neural network library like PyTorch [37] as *detaching*[4] the outputs from the neurons $v \notin \mathbf{S}$ prevents gradient signal from reaching them. In code, let `y` be the outputs and `m` be a binary vector with $\texttt{m}_v = 1$ if $v \in \mathbf{S}$ and 0 otherwise, then

$$\texttt{y = (1 - m) * y.detach() + m * y} \tag{21}$$

will prevent gradient signal from reaching $\mathbf{y}_v$ for $v \notin \mathbf{S}$.

Recall that the standard cross-entropy loss is

$$\mathcal{L}(\mathbf{y}) = -\log\left(\frac{\exp(\mathbf{y}_c)}{\sum_{v'=1}^{n}\exp(\mathbf{y}_{v'})}\right) = -\mathbf{y}_c + \log\left(\sum_{v'=1}^{n}\exp(\mathbf{y}_{v'})\right) \tag{22}$$

where $c \in \{1, ..., \ell\}$ is the correct label. The gradient of $\mathcal{L}$ to any s-neuron v is then

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}_v} = \frac{\exp(\mathbf{y}_v)}{\sum_{v'=1}^{n}\exp(\mathbf{y}_{v'})}. \tag{23}$$

If we define $\mathcal{G}$ as

$$\mathcal{G}(\mathbf{y}; \alpha) = \log\left(\sum_{v'=1}^{n}\exp(\mathbf{y}_{v'})\right) \tag{24}$$

then $g_v = \frac{\partial \mathcal{L}}{\partial y_v}$ as needed. Expressed in code

$$\texttt{y = model(x); G = torch.logsumexp((1 - m) * y.detach() + m * y, dim=1)} \tag{25}$$

where `model(...)` computes Equation 19. □

In the next two Lemmas we aim to show that, in expectation, $-\frac{\partial \mathcal{G}}{\partial \alpha_i} \leq 0$ for $i \neq j$ while $-\frac{\partial \mathcal{G}}{\partial \alpha_j} > 0$. Recall that j is the *correct* task—the task from which the data is drawn—and we will use i to refer to a different task.

When we take expectation, it is with respect to the random variables ξ, $\{M^\omega\}_{\omega \in \{1,..,k\}}$, and $\mathbf{x}$. Before we proceed further a few assumptions are formalized, *e.g.* what it means for tasks to be unrelated.

Assumption 1: We assume that the mask learned on task i will be independent from the data from task j: If the data is from task j then $\phi(\mathbf{x})$ and M^i and independent random variables.

Assumption 2: We assume that a negative weight and positive weight are equally likely to be masked out. As a result, $\mathbb{E}\left[\xi_{uv} M^i_{uv}\right] = 0$. Note that when $\mathbb{E}\left[\phi(\mathbf{x})\right] = 0$, which will be the case for zero mean random features, there should be little doubt that this assumption should hold.

Lemma I.2. *If data $\mathbf{x}$ comes from task j and $i \neq j$ then*

$$\mathbb{E}\left[-\frac{\partial \mathcal{G}}{\partial \alpha_i}\right] \leq 0 \tag{26}$$

[4] `https://pytorch.org/docs/stable/autograd.html`

Proof. We may write the gradient as

$$\frac{\partial \mathcal{G}}{\partial \alpha_i} = \sum_{v=1}^{n} \frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} \frac{\partial \mathbf{y}_v}{\partial \alpha_i} \tag{27}$$

and use that $\frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} = 0$ for $v \notin \mathbf{S}$. Moreover, $\mathbf{y}_v$ may be written as

$$\mathbf{y}_v = \sum_{u=1}^{n} \phi(\mathbf{x})_u W_{uv} \left(\sum_{i=1}^{k} \alpha_i M_{uv}^i \right) \tag{28}$$

with $W_{uv} = \xi_{uv}\sqrt{2/m}$ and so Equation 27 becomes

$$\frac{\partial \mathcal{G}}{\partial \alpha_i} = \frac{\sqrt{2}}{\sqrt{m}} \sum_{v \in \mathbf{S}} \sum_{u=1}^{n} \frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} M_{uv}^i. \tag{29}$$

Taking the expectation (and using linearity) we obtain

$$\mathbb{E}\left[\frac{\partial \mathcal{G}}{\partial \alpha_i}\right] = \frac{\sqrt{2}}{\sqrt{m}} \sum_{v \in \mathbf{S}} \sum_{u=1}^{n} \mathbb{E}\left[\frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} M_{uv}^i\right]. \tag{30}$$

In Lemma J.1 we formally show that each term in this sum is greater than or equal to 0, which completes this proof. However, we can see informally now why expectation should be close to 0 if we ignore the gradient term as

$$\mathbb{E}\left[\phi(\mathbf{x})_u \xi_{uv} M_{uv}^i\right] = \mathbb{E}\left[\phi(\mathbf{x})_u\right] \mathbb{E}\left[\xi_{uv} M_{uv}^i\right] = 0 \tag{31}$$

where the first equality follows from Assumption 1 and the latter follows from Assumption 2. □

We have now seen that in expectation $-\frac{\partial \mathcal{G}}{\partial \alpha_i} \leq 0$ for $i \neq j$. It remains to be shown that we should expect $-\frac{\partial \mathcal{G}}{\partial \alpha_j} > 0$.

Lemma I.3. *If data* $\mathbf{x}$ *comes from the task* j *then*

$$\mathbb{E}\left[-\frac{\partial \mathcal{G}}{\partial \alpha_j}\right] > 0. \tag{32}$$

Proof. Following Equation 30, it suffices to show that for $u \in \{1, ..., m\}$, $v \in \mathbf{S}$

$$\mathbb{E}\left[-\frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} M_{uv}^j\right] > 0. \tag{33}$$

Since $v \in \mathbf{S}$ we may invoke Lemma I.1 to rewrite our objective as

$$\mathbb{E}\left[-\frac{\partial \mathcal{L}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} M_{uv}^j\right] > 0 \tag{34}$$

where $\mathcal{L}$ is the supervised loss used for training. Recall that in the mask training algorithm, real valued scores S_{uv}^j are associated with M_{uv}^j [39, 30]. The update rule for S_{uv}^j on the backward pass is then

$$S_{uv}^j \leftarrow S_{uv}^j + \eta \left(-\frac{\partial \mathcal{L}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} \right) \tag{35}$$

for some learning rate η. Following Mallya *et al.* [30] (with threshold 0, as used in Section 4.2), we let $M_{uv}^j = 1$ if $S_{uv}^j > 0$ and otherwise assign $M_{uv}^j = 0$. As a result, we expect that M_{uv}^j is 1 when $-\frac{\partial \mathcal{L}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv}$ is more consistently positive than negative. In other words, the expected product of M_{uv}^j and $-\frac{\partial \mathcal{L}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv}$ is positive, satisfying Equation 34. □

Together, three Lemmas have demonstrated that in expectation $-\frac{\partial \mathcal{G}}{\partial \alpha_i} \leq 0$ for $i \neq j$ while $-\frac{\partial \mathcal{G}}{\partial \alpha_j} > 0$. Accordingly, we should expect that

$$\arg\max_i \left(-\frac{\partial \mathcal{G}}{\partial \alpha_i} \right). \tag{36}$$

returns the correct task j. While a full, formal treatment which includes the analysis of noise is beyond the scope of this work, we hope that this section has helped to further intuition. However, we are missing one final piece—what is the relation between $\mathcal{G}$ and $\mathcal{H}$?

It is not difficult to imagine that $\mathcal{H}$ should imitate the loss, which attempts to raise the score of one logit while bringing all others down. Analytically we find that $\mathcal{H}$ can be decomposed into two terms as follows

$$\mathcal{H}(\mathbf{p}) = -\sum_{v=1}^{n} \mathbf{p}_v \log \mathbf{p}_v \tag{37}$$

$$= -\sum_{v=1}^{n} \mathbf{p}_v \log \left(\frac{\exp(\mathbf{y}_v)}{\sum_{v'=1}^{n} \exp(\mathbf{y}'_v)} \right) \tag{38}$$

$$= \left(-\sum_{v=1}^{n} \mathbf{p}_v \mathbf{y}_v \right) + \log \left(\sum_{v'=1}^{n} \exp(\mathbf{y}'_v) \right) \tag{39}$$

where the latter term is $\mathcal{G}$. With more and more neurons in the output layer, $\mathbf{p}_v$ will become small moving $\mathcal{H}$ towards $\mathcal{G}$.

J Additional Technical Details

Lemma J.1. *If j is the true task and $i \neq j$ then*

$$\mathbb{E}\left[\frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right] \geq 0 \tag{40}$$

Proof. Recall from Lemma I.1 that

$$\frac{\partial \mathcal{G}}{\partial \mathbf{y}_v} = \mathbf{p}_v = \frac{\exp(\mathbf{y}_v)}{\sum_{v'=1}^{n} \exp(\mathbf{y}_{v'})} \tag{41}$$

and so we rewrite equation 40 as

$$\mathbb{E}\left[\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right] \geq 0. \tag{42}$$

By the law of total expectation

$$\mathbb{E}\left[\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right] = \mathbb{E}\left[\mathbb{E}\left[\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \,\middle|\, \left| \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right| \right] \right] \tag{43}$$

and so it suffices to show that

$$\mathbb{E}\left[\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \,\middle|\, \left| \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right| = \kappa \right] \geq 0 \tag{44}$$

for any $\kappa \geq 0$. In the case where where $\kappa = 0$ Equation 44 becomes

$$\mathbb{E}\left[0\mathbf{p}_v \,\middle|\, \left| \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} \right| = 0 \right] = 0 \tag{45}$$

and so we are only left to consider $\kappa > 0$. Note that $\kappa > 0$ restricts M^i_{uv} to be 1.

Again invoking the law of total expectation we rewrite Equation 45 as

$$\begin{aligned}
&\mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\,\middle|\,\left|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\right|\right] \\
&= \mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right]\mathbb{P}\left(\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right) \\
&+ \mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right]\mathbb{P}\left(\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right).
\end{aligned} \tag{46}$$

Moreover, since the data is from task $j \neq i$, we can use Assumption 1 and 2 to show that each of the cases above is equally likely. Formally,

$$\mathbb{P}\left(\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right) \tag{47}$$
$$= \mathbb{P}\left(\left(\left\{\phi(\mathbf{x})_u=\kappa\right\}\cap\left\{\xi_{uv}M^i_{uv}=1\right\}\right)\cup\left(\left\{\phi(\mathbf{x})_u=-\kappa\right\}\cap\left\{\xi_{uv}M^i_{uv}=-1\right\}\right)\right) \tag{48}$$
$$= \mathbb{P}\left(\phi(\mathbf{x})_u=\kappa\right)\mathbb{P}\left(\xi_{uv}M^i_{uv}=+1\right)+\mathbb{P}\left(\phi(\mathbf{x})_u=-\kappa\right)\mathbb{P}\left(\xi_{uv}M^i_{uv}=-1\right) \tag{49}$$
$$= \mathbb{P}\left(\phi(\mathbf{x})_u=\kappa\right)\mathbb{P}\left(\xi_{uv}M^i_{uv}=-1\right)+\mathbb{P}\left(\phi(\mathbf{x})_u=-\kappa\right)\mathbb{P}\left(\xi_{uv}M^i_{uv}=+1\right) \tag{50}$$
$$= \mathbb{P}\left(\left(\left\{\phi(\mathbf{x})_u=\kappa\right\}\cap\left\{\xi_{uv}M^i_{uv}=-1\right\}\right)\cup\left(\left\{\phi(\mathbf{x})_u=-\kappa\right\}\cap\left\{\xi_{uv}M^i_{uv}=+1\right\}\right)\right) \tag{51}$$
$$= \mathbb{P}\left(\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right) \tag{52}$$

and so we may factor out the probability terms in Equation 46. Accordingly, it suffices to show that

$$\mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right]+\mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right]\geq 0. \tag{53}$$

Before we proceed, we will introduce a function h which we use to denote

$$h\left(\{\mathbf{y}_v\},\kappa\right)=\kappa\frac{\exp(\mathbf{y}_v+\kappa)}{\exp(\mathbf{y}_v+\kappa)+\sum_{v'\neq v}\exp(\mathbf{y}_{v'})}. \tag{54}$$

for $\kappa > 0$. We will make use of two interesting properties of h.

We first note that $h\left(\{\mathbf{y}_v\},\kappa\right)+h\left(\{\mathbf{y}_v\},-\kappa\right)\geq 0$, which is formally shown in J.2.

Second, we note that

$$\begin{aligned}
&\mathbb{P}\left(\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=h\left(\{\mathbf{y}_v\},\kappa\right)|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right) \\
&=\mathbb{P}\left(\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=h\left(\{\mathbf{y}_v\},-\kappa\right)|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right)
\end{aligned} \tag{55}$$

which we dissect in Lemma J.3.

Utilizing these two properties of h we may show that Equation 53 holds as

$$\mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right]+\mathbb{E}\left[\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}\middle|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right] \tag{56}$$
$$\begin{aligned}
&=\int_{\mathbb{R}}h\left(\{\mathbf{y}_v\},\kappa\right)\,d\mathbb{P}\left(\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=h\left(\{\mathbf{y}_v\},\kappa\right)|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right) \\
&+\int_{\mathbb{R}}h\left(\{\mathbf{y}_v\},-\kappa\right)\,d\mathbb{P}\left(\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=h\left(\{\mathbf{y}_v\},-\kappa\right)|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=-\kappa\right)
\end{aligned} \tag{57}$$
$$=\int_{\mathbb{R}}\left(h\left(\{\mathbf{y}_v\},\kappa\right)+h\left(\{\mathbf{y}_v\},-\kappa\right)\right)\,d\mathbb{P}\left(\mathbf{p}_v\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=h\left(\{\mathbf{y}_v\},\kappa\right)|\phi(\mathbf{x})_u\xi_{uv}M^i_{uv}=\kappa\right) \tag{58}$$
$$\geq 0. \tag{59}$$

□

Lemma J.2. $h\left(\{\mathbf{y}_v\},\kappa\right)+h\left(\{\mathbf{y}_v\},-\kappa\right)\geq 0$.

Proof. Recall that $\kappa \geq 0$. Moreover,

$$\exp(\mathbf{y}_v + \kappa) \sum_{v'} \exp(\mathbf{y}_{v'}) \geq \exp(\mathbf{y}_v - \kappa) \sum_{v'} \exp(\mathbf{y}_{v'}) \tag{60}$$

$$\begin{aligned} \Rightarrow\ & \exp(\mathbf{y}_v + \kappa) \left(\exp(\mathbf{y}_v - \kappa) + \sum_{v'} \exp(\mathbf{y}_{v'}) \right) \\ & \geq \exp(\mathbf{y}_v - \kappa) \left(\exp(\mathbf{y}_v + \kappa) + \sum_{v'} \exp(\mathbf{y}_{v'}) \right) \end{aligned} \tag{61}$$

$$\Rightarrow \kappa \frac{\exp(\mathbf{y}_v + \kappa)}{\exp(\mathbf{y}_v + \kappa) + \sum_{v' \neq v} \exp(\mathbf{y}_{v'})} \geq \kappa \frac{\exp(\mathbf{y}_v - \kappa)}{\exp(\mathbf{y}_v - \kappa) + \sum_{v' \neq v} \exp(\mathbf{y}_{v'})} \tag{62}$$

and we may then subtract the term on the right from both sides. □

Lemma J.3. *Consider take $i \neq j$ where j is the correct task. Then*

$$\begin{aligned} & \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, \kappa\right) | \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = \kappa\right) \\ & = \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, -\kappa\right) | \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = -\kappa\right). \end{aligned} \tag{63}$$

Proof. Note that this equation is satisfied when $\kappa = 0$ (since $-0 = 0$). For the remainder of this proof we will instead consider the case where $\kappa > 0$ (and so $M^i_{uv} = 1$).

If we define ρ as $\rho = \left(\mathbb{P}\left(\phi(\mathbf{x})_u = \kappa\right) + \mathbb{P}\left(\phi(\mathbf{x})_u = -\kappa\right)\right)^{-1}$ then may decompose Equation 63 into four terms. Namely,

$$\begin{aligned} & \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, \kappa\right) | \phi(\mathbf{x})_u = \kappa\right) \mathbb{P}\left(\phi(\mathbf{x})_u = \kappa\right) \rho \\ & + \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, \kappa\right) | \phi(\mathbf{x})_u = -\kappa\right) \mathbb{P}\left(\phi(\mathbf{x})_u = -\kappa\right) \rho \\ & = \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, -\kappa\right) | \phi(\mathbf{x})_u = \kappa\right) \mathbb{P}\left(\phi(\mathbf{x})_u = \kappa\right) \rho \\ & + \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, -\kappa\right) | \phi(\mathbf{x})_u = -\kappa\right) \mathbb{P}\left(\phi(\mathbf{x})_u = -\kappa\right) \rho. \end{aligned} \tag{64}$$

Equality follows from the fact that term 1 and 3 are equal, as are terms 2 and 4. We will consider terms 1 and 3, as the other case is nearly identical.

Let H be the event where $\phi(\mathbf{x})_u = \kappa, M^i_{uv} = 1$ and all other random variables (except for ξ_{uv}) take values such that, if $\xi_{uv} = +1$ then $\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, \kappa\right)$. On the other hand, if $\xi_{uv} = -1$ then $\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, -\kappa\right)$. Then, subtracting term 3 from term 1 (and factoring out the shared term) we find

$$\begin{aligned} & \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, \kappa\right) | \phi(\mathbf{x})_u = \kappa\right) \\ & - \mathbb{P}\left(\mathbf{p}_v \phi(\mathbf{x})_u \xi_{uv} M^i_{uv} = h\left(\{\mathbf{y}_v\}, -\kappa\right) | \phi(\mathbf{x})_u = \kappa\right) \end{aligned} \tag{65}$$

$$= \mathbb{P}\left(\xi_{uv} = +1 | H\right) - \mathbb{P}\left(\xi_{uv} = -1 | H\right) = 0 \tag{66}$$

since ξ_{uv} is independent of H, and $\xi_{uv} = -1$ and $+1$ with equal probability. □