Continual Learning

Classifier Bias and Architectural Strategies

Antonio Carta

antonio.carta@unipi.it

Outline

Classifier Bias

- How to train a CIL classifer
- How to fix CIL classifier bias

Architectural Methods

- Architectural Growth
- Sparsity

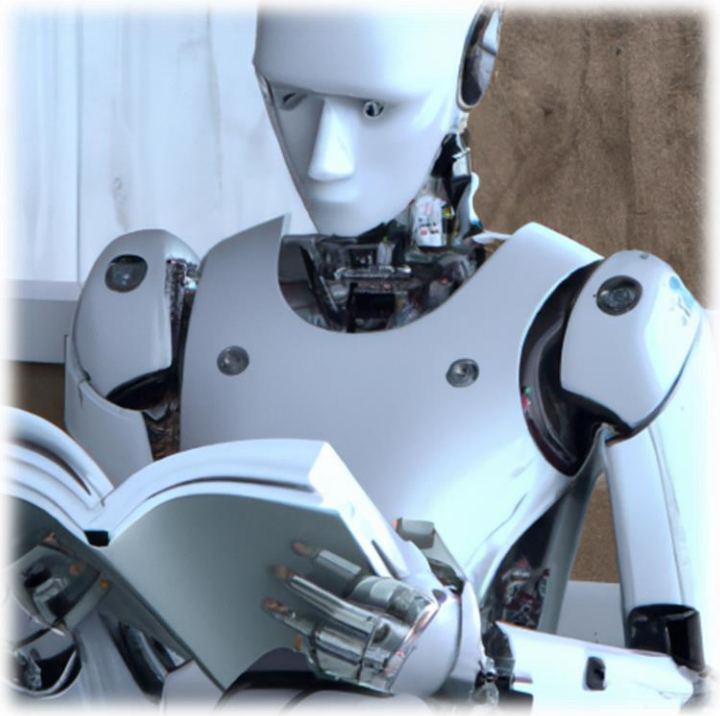


Image from Dall-e

Classifier Bias

RECAP: CNN in Computer Vision

In offline learning we observe these general phenomena:

- CNN Architecture: Input->(Conv->pool->relu)->FCN->softmax
- Low layers learn early and learn low-level features. They don't change much after the first epochs.
- High layers tend to be more task/domain-specific.

Q: How do we exploit this information in CL?

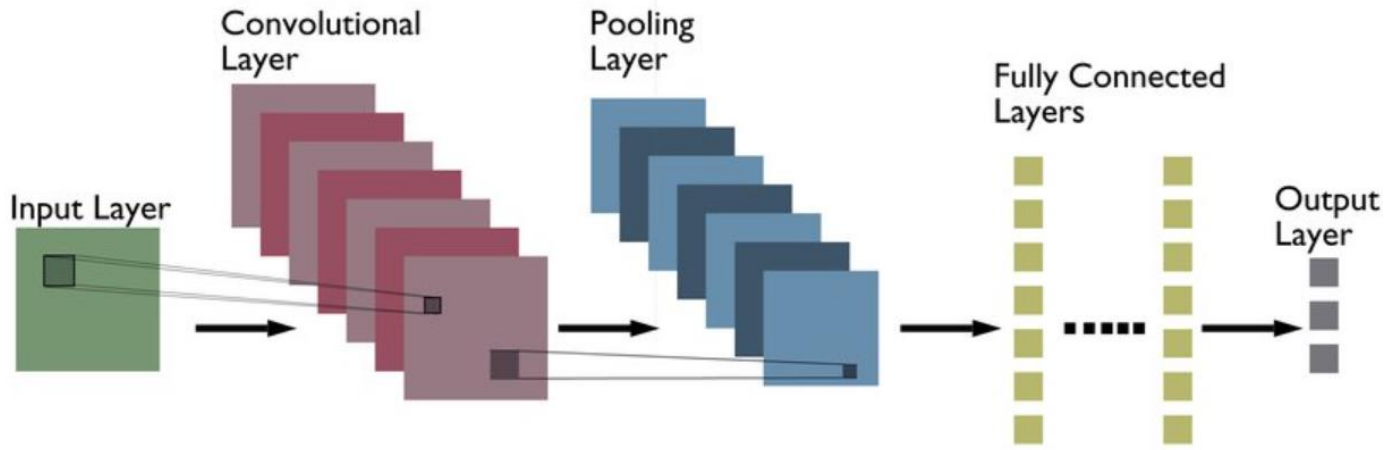
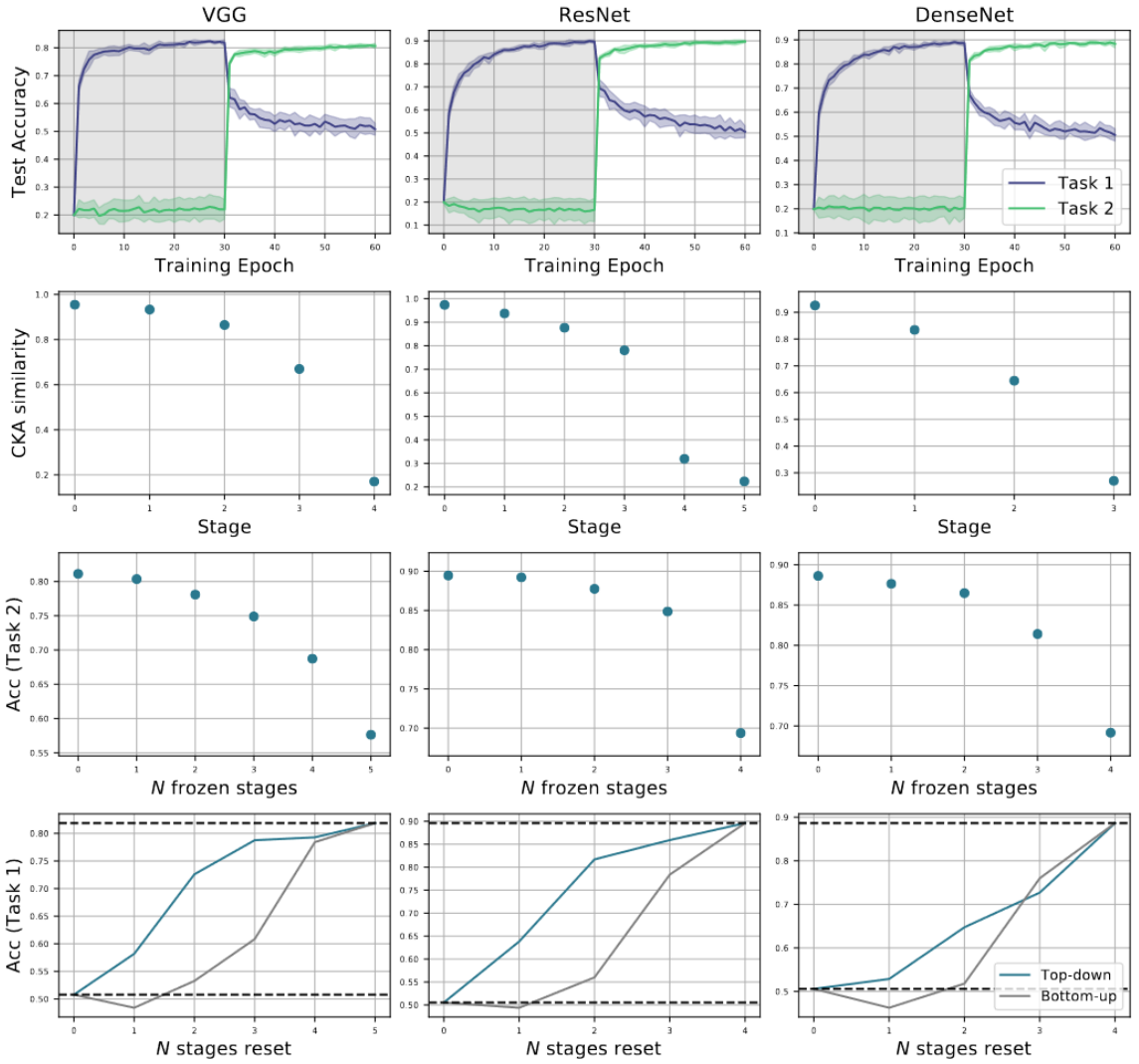


Image source: Data Analytics

Most Forgetting Happens in the Higher Layers

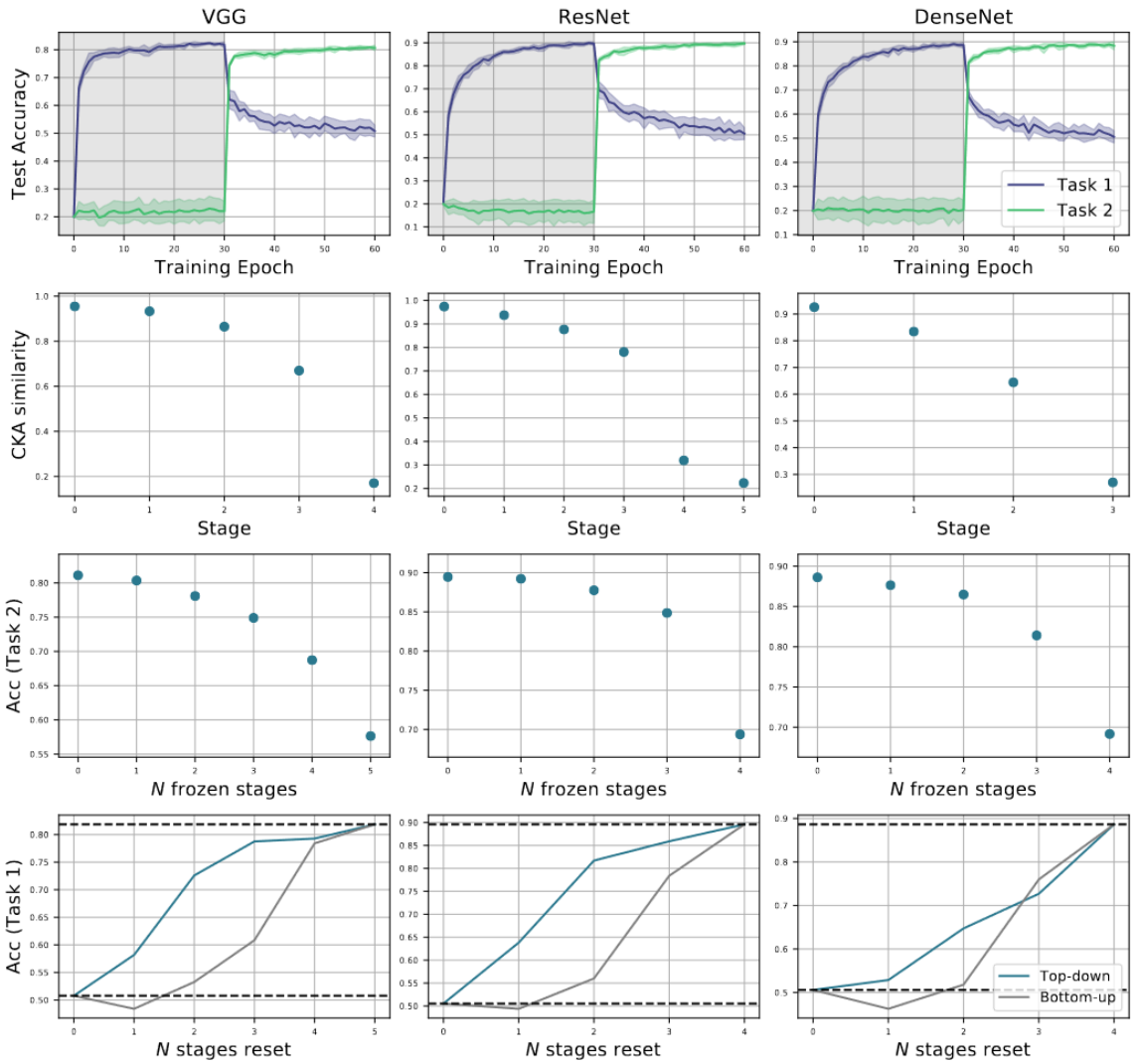
- **Higher layers are the primary source of catastrophic forgetting** (CIFAR10 2 tasks of 5 classes, top row)
- **Representational similarity (CKA)** scores between layers before and after training on Task 2 indicate that **lower layers don't change much** through training on task two, while **higher layers change significantly** (second row).



Most Forgetting Happens in the Higher Layers

- When freezing a contiguous block of layers (starting from lowest) and measuring accuracy during training on Task 2, we observe **lower layers can be frozen with little impact** (third row).
- Finally, after training on Task 2 we reset contiguous blocks of layers to their values before training and record the resulting accuracy on Task 1 (bottom row). We see a **significant increase in accuracy when resetting the highest N layers (blue line) compared to resetting the N lowest layers (gray line)**.

CONCLUSION: higher layers are disproportionately responsible for forgetting.

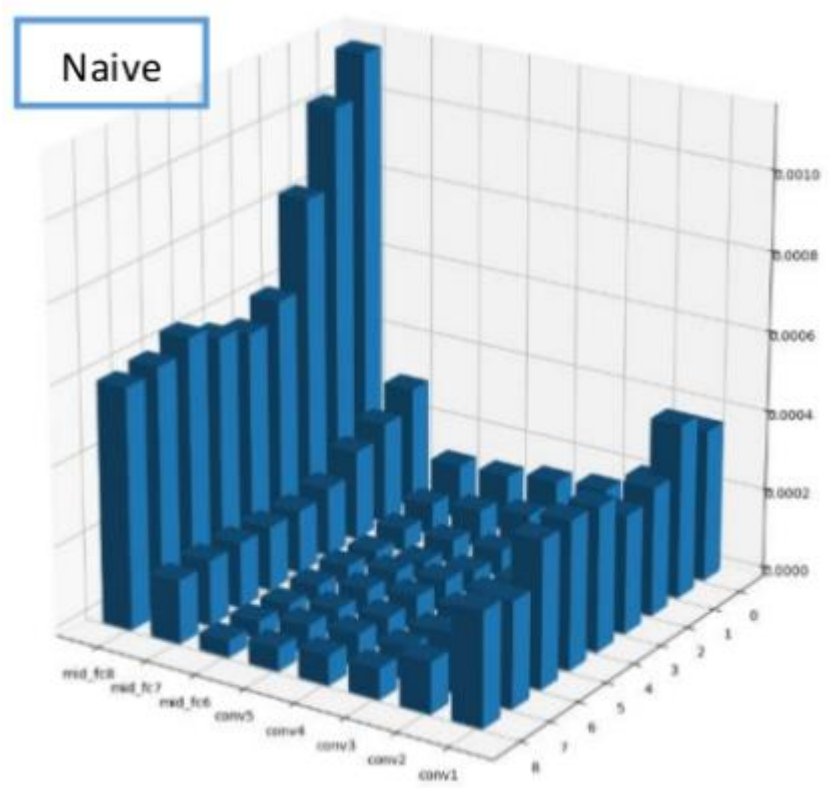


Ramasesh, Vinay V. et al.. "Anatomy of Catastrophic Forgetting: Hidden Representations and Task Semantics." ICLR 2021

Finetuning the classifier

Simple tricks to mitigate forgetting:

- Freeze lower layers after the first learning experience
- Use a pretrained model
- Finetune the classifier, possibly using a replay buffer to avoid forgetting

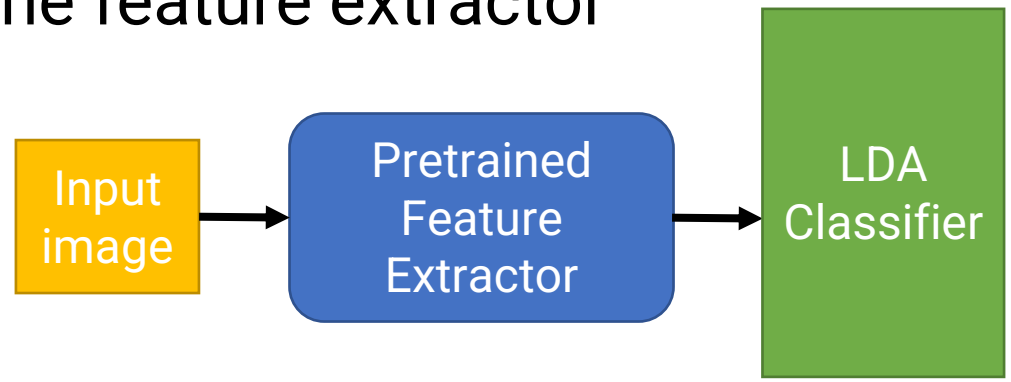


Amount of weight changes by layer and training batch in CaffeNet for different approaches

Maltoni, Davide, and Vincenzo Lomonaco. "Continuous Learning in Single-Incremental-Task Scenarios." Neural Networks 2019

Deep SLDA

- **Deep**: Start from a fixed pretrained model (e.g. a CNN pretrained on ImageNet)
- **SLDA**: use Streaming Linear Discriminant Analysis to train the classifier
- Extremely efficient compared to full backpropagation and applicable to online scenarios even without replay
- No adaptation of the feature extractor



SLDA – Streaming LDA

- Pretrained CNN $y_t = F\left(G\left(\mathbf{X}_t\right)\right)$
  - G is pretrained
  - F is an LDA classifier (learned online) $F(G(\mathbf{X}_t)) = \mathbf{W}\mathbf{z}_t + \mathbf{b}$
- Each class is modeled as a gaussian
- Estimate the mean for each class online (OML module)

$$\boldsymbol{\mu}_{(k=y,t+1)} \leftarrow \frac{c_{(k=y,t)}\boldsymbol{\mu}_{(k=y,t)} + \mathbf{z}_t}{c_{(k=y,t)} + 1}$$

$$c_{(k=y,t+1)} = c_{(k=y,t)} + 1 \ ,$$

SLDA – Streaming LDA

- Each class is modeled as a gaussian
- You can also estimate the covariance
  - The full covariance adds n^2 parameters
  - If the online stream is small, it may be better to keep it fixed after the first step, shared among classes, or use a diagonal approximation

$$\boldsymbol{\Sigma}_{t+1} = \frac{t\boldsymbol{\Sigma}_t + \Delta_t}{t+1} \qquad \Delta_t = \frac{t\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)^T}{t+1}$$

LDA Classifier

- Linear classifier
- Weights defined using the estimated mean and covariance

Classifier

$$F(G(\mathbf{X}_t)) = \mathbf{W}\mathbf{z}_t + \mathbf{b}$$

Precision (inverse of covariance) + shrinkage for regularization

$$\mathbf{\Lambda} = [(1-\bar{\varepsilon})\,\mathbf{\Sigma} + \varepsilon\mathbf{I}]^{-1}$$

Rows of W

$$\mathbf{w}_k = \mathbf{\Lambda}\boldsymbol{\mu}_k$$

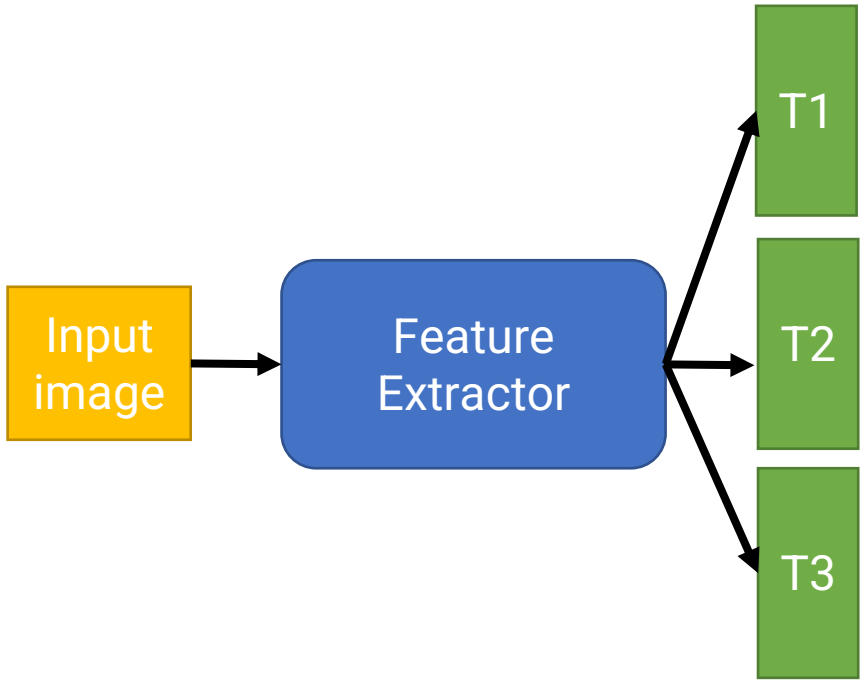
Bias of the classifier

$$b_k = -\frac{1}{2}\,(\boldsymbol{\mu}_k \cdot \mathbf{\Lambda}\boldsymbol{\mu}_k)$$

Multi-Head Classifier

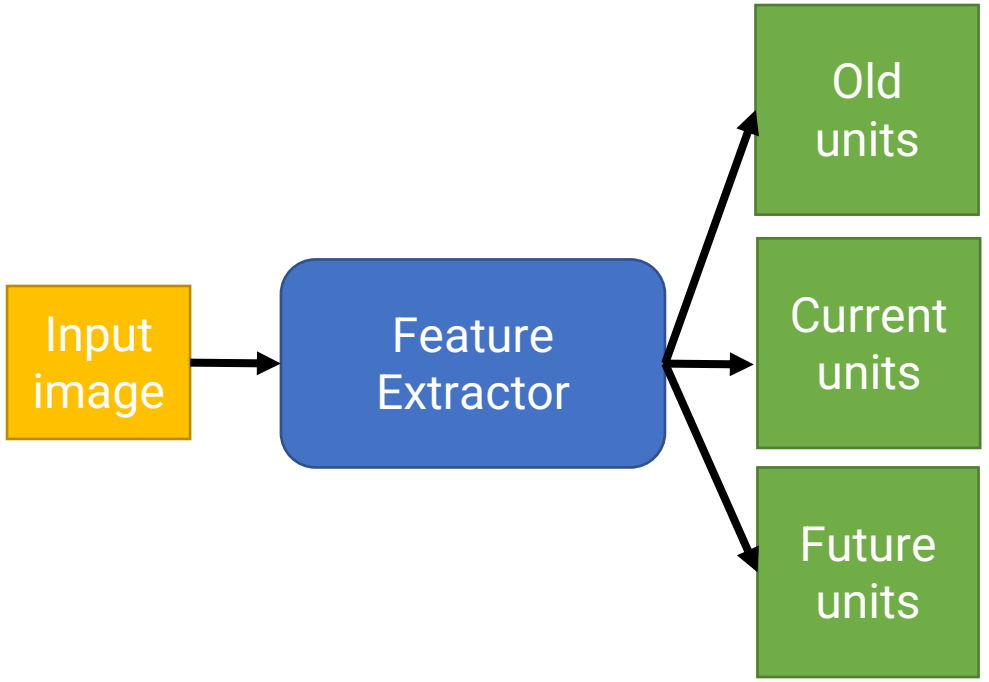
Multi-head architecture can be seen a simple way to partition the classifier:

- Great to specialize behaviour if the notion of task is explicit
- Reduces the number of classes in each classification head, making them easier to learn
- Clear separation between shared and task-specific parameters
- Not always applicable since it needs task labels or task inference



Single Head in Class-Incremental

- Sometimes it help to distinguish the units even in class-incremental settings
- Old/current/future units may be treated differently
- Example in LwF:
  - CE for NEW+OLD units
  - KD for OLD units
- Other examples: masking
  - If we know that some classes are not available anymore, we can mask them to zero probability to avoid wrong predictions



Classifier Bias

In the last layer of a DNN (the classifier) in a class-incremental setting we see that new classes have weights with a larger norm.

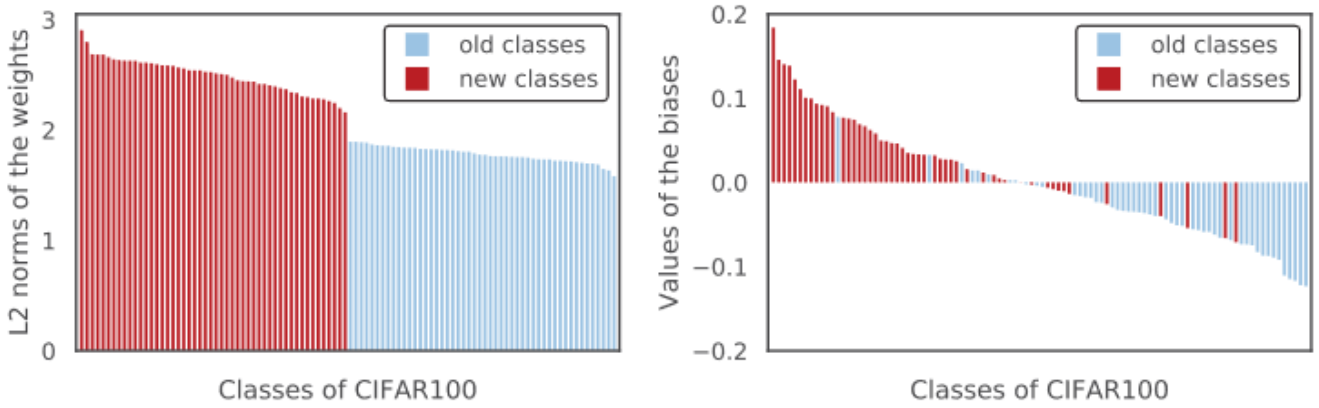


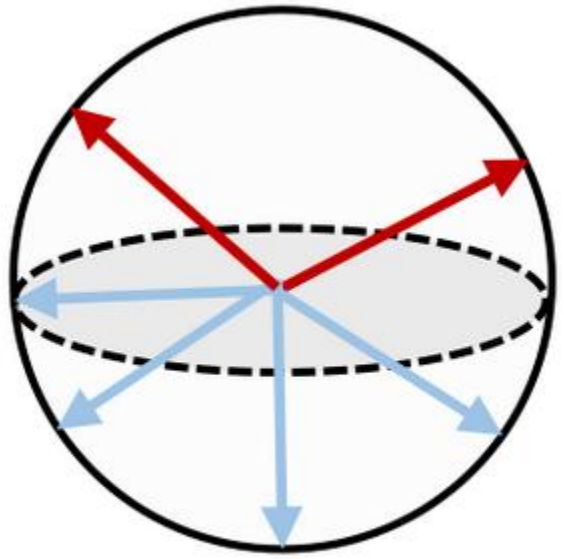
Figure 3. Visualization of the weights and biases in the last layer for old and new classes. The results come from the incremental setting of CIFAR100 (1 phase) by iCaRL [29].

Hou, Saihui et al.. "Learning a Unified Classifier Incrementally via Rebalancing." In CVPR 2019

Cosine Classifier

- To mitigate the bias, we normalize the weights of the classifier
- Consider the weights of the last layer as class embeddings θ_j
- Normalize class embedding and extracted features
- Compute cosine similarity

$$p_i(x) = \frac{\exp(\eta \langle \bar{\theta}_i, \bar{f}(x) \rangle)}{\sum_j \exp(\eta \langle \bar{\theta}_j, \bar{f}(x) \rangle)}$$



Cosine Normalization

Hou, Saihui et al.. "Learning a Unified Classifier Incrementally via Rebalancing." In CVPR 2019

Copy Weights with Re-Init (CWR)

- **Incremental training of linear classifier**
  - cw=consolidated weight, used for inference
  - tw=temporary weight, used for training
  - $past_j$ counter of patterns for each class
  - $wpast_j$ scaling term
- Example of **Dual Learning System**:
  - A system with fast and slow weights
  - Renormalize cw weights after each learning experience with a weighted sum of all the learned weights
  - Reset tw to zero after each learning step
- agnostic to the scenario (NI, NC, NIC) and applicable with repetitions and online settings.

procedure CWR*

$cw = 0$

$past = 0$

init $\bar{\Theta}$ random or from pre-trained model (e.g. on ImageNet)

for each training batch B_i:

  expand output layer with neurons for the new classes in B_i never seen before

  $$tw[j] = \begin{cases} cw[j], & \text{if class } j \text{ in } B_i \\ 0, & \text{otherwise} \end{cases}$$

  train the model with SGD on the s_i classes of B_i:

    if $B_i = B_1$ learn both $\bar{\Theta}$ and tw

    else learn tw while keeping $\bar{\Theta}$ fixed

  for each class j in B_i:

    $wpast_j = \sqrt{\frac{past_j}{cur_j}}$, where cur_j is the number of patterns of class j in B_i

    $cw[j] = \frac{cw[j] \cdot wpast_j + (tw[j] - avg(tw))}{wpast_j + 1}$

    $past_j = past_j + cur_j$

  test the model by using $\bar{\Theta}$ and cw

Summing Up

- If we don’t have task labels and we encounter new classes over time, and we don’t have replay, the DNN suffers from classifier bias
- Classifier bias is the biggest source of forgetting in a naive method
- We can counteract it by
  - Replay
  - Classifier Normalization (like LUCIR)
  - Classifier finetuning/adaptation (like CWR)
  - Training the classifier with algorithms that have less classifier bias (like LDA)
    - Some prototype-based methods (remember ProtoNet?) are also more robust to classifier bias

Architectural Methods

Modular Architectures

Idea:

- split the networks into several modules
- Connect modules to enable transfer
- Freeze/mask module to limit forgetting

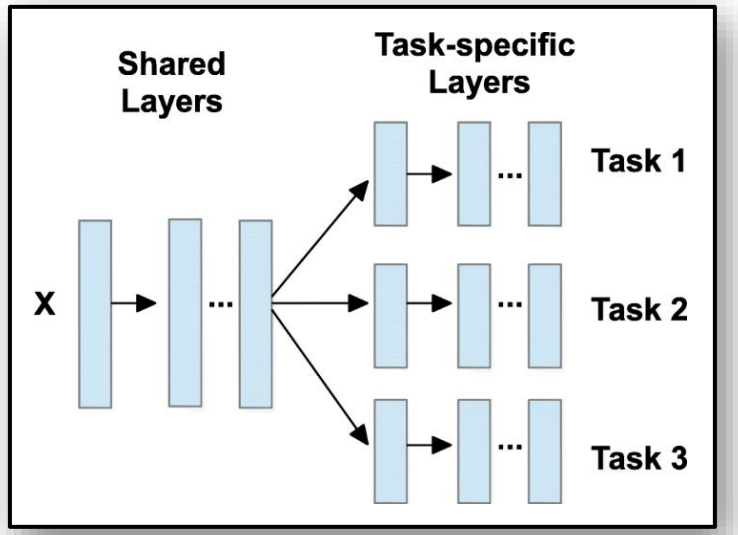
Opportunities

- Explicit separation between task-specific and shared components
- Eliminate forgetting

Challenges

- Limiting memory growth
- Requirements of task labels
- Forward transfer

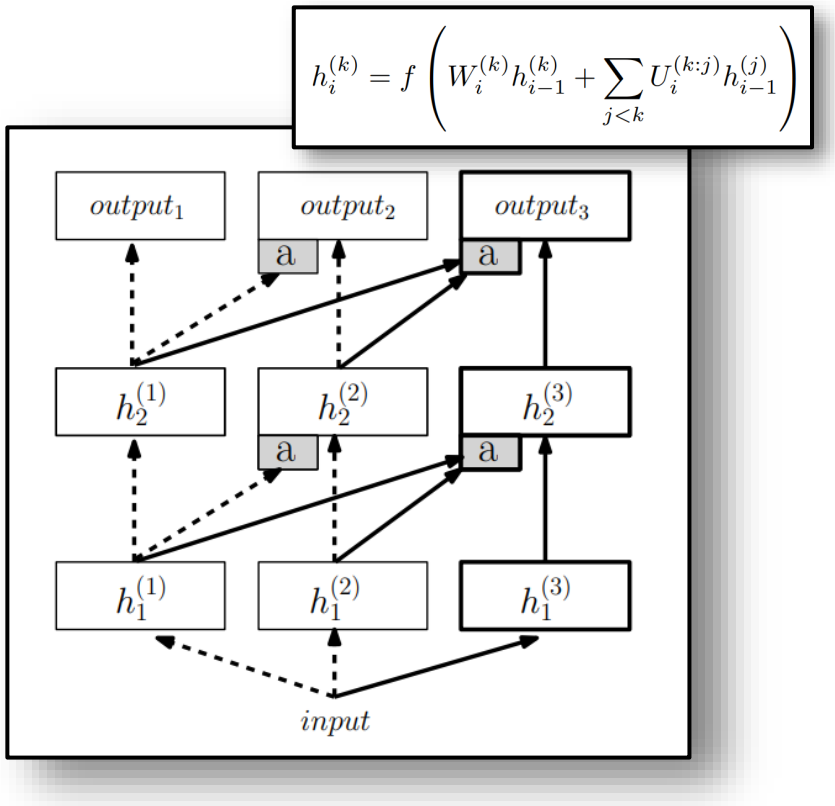
Conflicting requirements: a good method needs to balance memory occupation, eliminate forgetting, promoting forward transfer.



Progressive Neural Networks (PNN)

A Basic Modular Architecture

- **Column**: Each new task adds its own "column" of features to each layer
- **Adapter**: New columns are connected to all the previous one via adapter
- **Inference**: task labels are used to activate the correct columns



Progressive Neural Networks, Rusu et al. 2016.

PNN – Column

Column: Each new task adds its own "column" of features to each layer

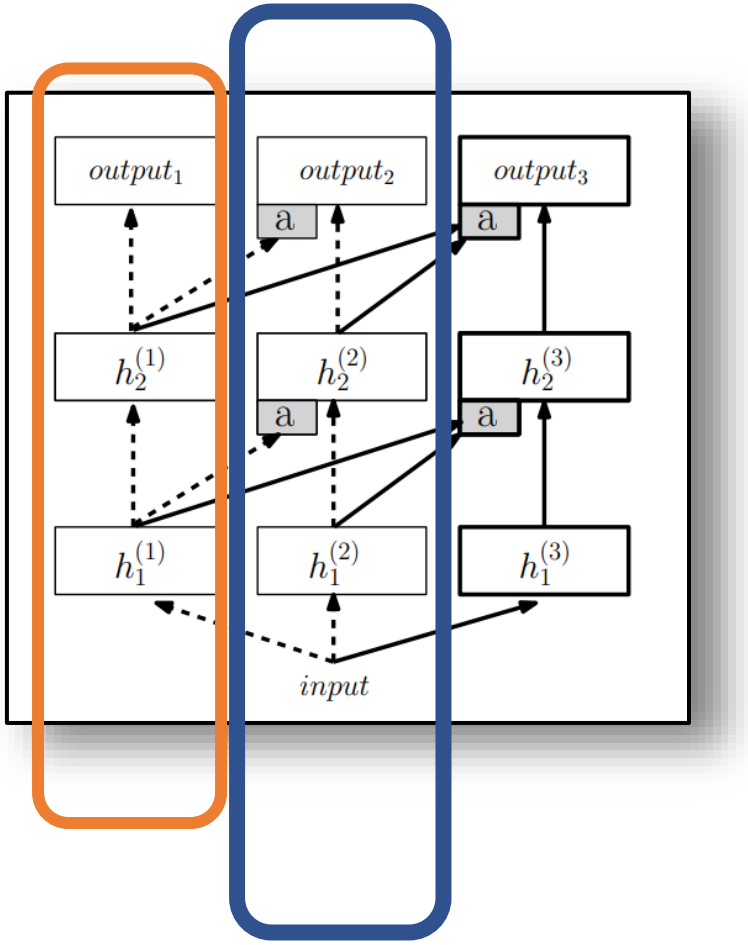
- Each column is connected to all the previous ones
- After training the column is **frozen**
- **Inference**: use task labels to activate the correct columns

PNN Column

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + \sum_{j<k} U_i^{(k:j)} h_{i-1}^{(j)}\right)$$

Connections to previous columns

PNN Column



Progressive Neural Networks, Rusu et al. 2016.

PNN – MLP Adapters

- **MLP Adapter**: takes as input a weighted concatenation of the previous columns.

Quadratic scaling of memory occupation

$$h_{i-1}^{(<k)} = [h_{i-1}^{(1)} \cdots h_{i-1}^{(j)} \cdots h_{i-1}^{(k-1)}]$$

$$V_i^{(k:j)} \in \mathbb{R}^{n_{i-1} \times n_{i-1}^{(<k)}}$$

$$h_i^{(k)} = \sigma\left(W_i^{(k)} h_{i-1}^{(k)} + U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})\right)$$

Progressive Neural Networks, Rusu et al. 2016.

PNN – Final Considerations

- **Good forward transfer**: each task can re-use previous columns
- **Inhibits forgetting** by freezing columns
- **Poor scaling in memory size**: quadratic due to adapters
- Requires **task labels**

Two open problems:

- How do we limit the **memory growth**?
- How do we choose which column to activate if don't have **task labels**?

Memory Growth: PNN Columns can be compressed

Good news: most of the capacity is not used!

- We can reduce the size of new columns over time
- We can compress them (e.g. after training)

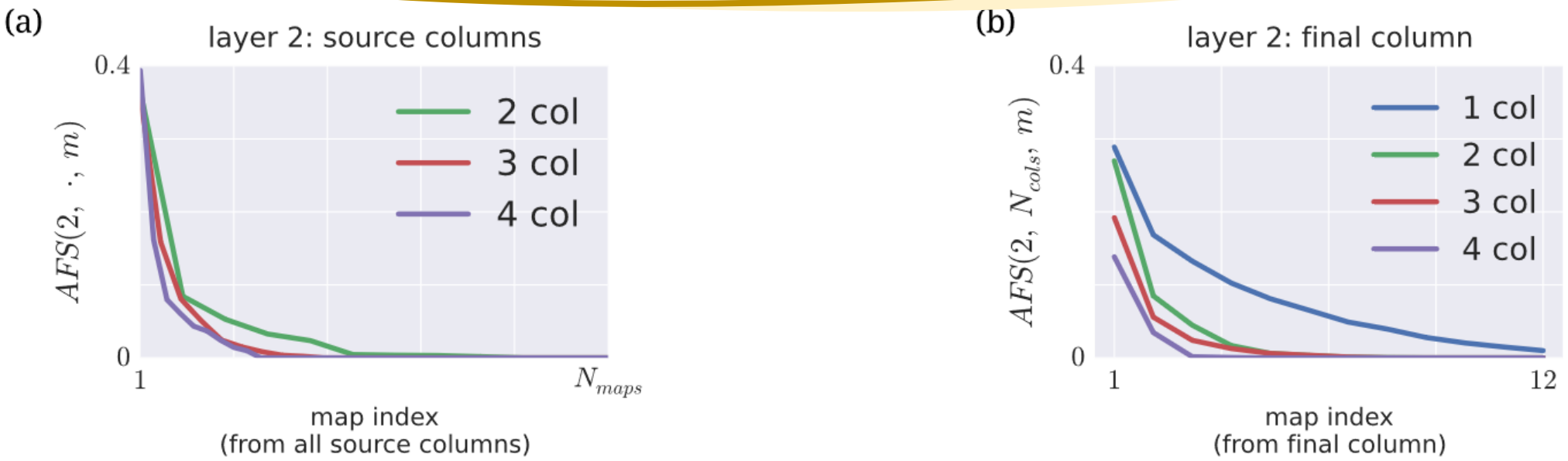


Figure 10: (a) Spectra of AFS values (for layer 2) across all feature maps from source columns, for the Atari dataset. The spectra show the range of AFS values, and are averaged across networks. While the 2 column / 3 column / 4 column nets all have different values of N_{maps} (here, 12, 24, and 36 respectively), these have been dilated to fit the same axis to show the proportional use of these maps. (b) Spectra of AFS values (for layer 2) for the feature maps from only the final column.

Progressive Neural Networks, Rusu et al. 2016.

Task Inference

- **Modular architecture + task inference** to remove need for task labels
- **Task Inference**: classifier that given an input predicts the task label
- Often predicting the task label is easier than predicting the class.
  - Example: identifying a language (task inference) is easier than predicting the next word of an incomplete sentence (solving the task).
  - We can use a proxy signal: reconstruction error, pattern of activations, …
  - We can use a simple classifier, easier to train continually

Gated Ensemble

- **Gate**: a component that enables/disable a module of the network
- Given a task label predictor, we can use it to gate a modular network
- You need some form of competition or normalization between the gates

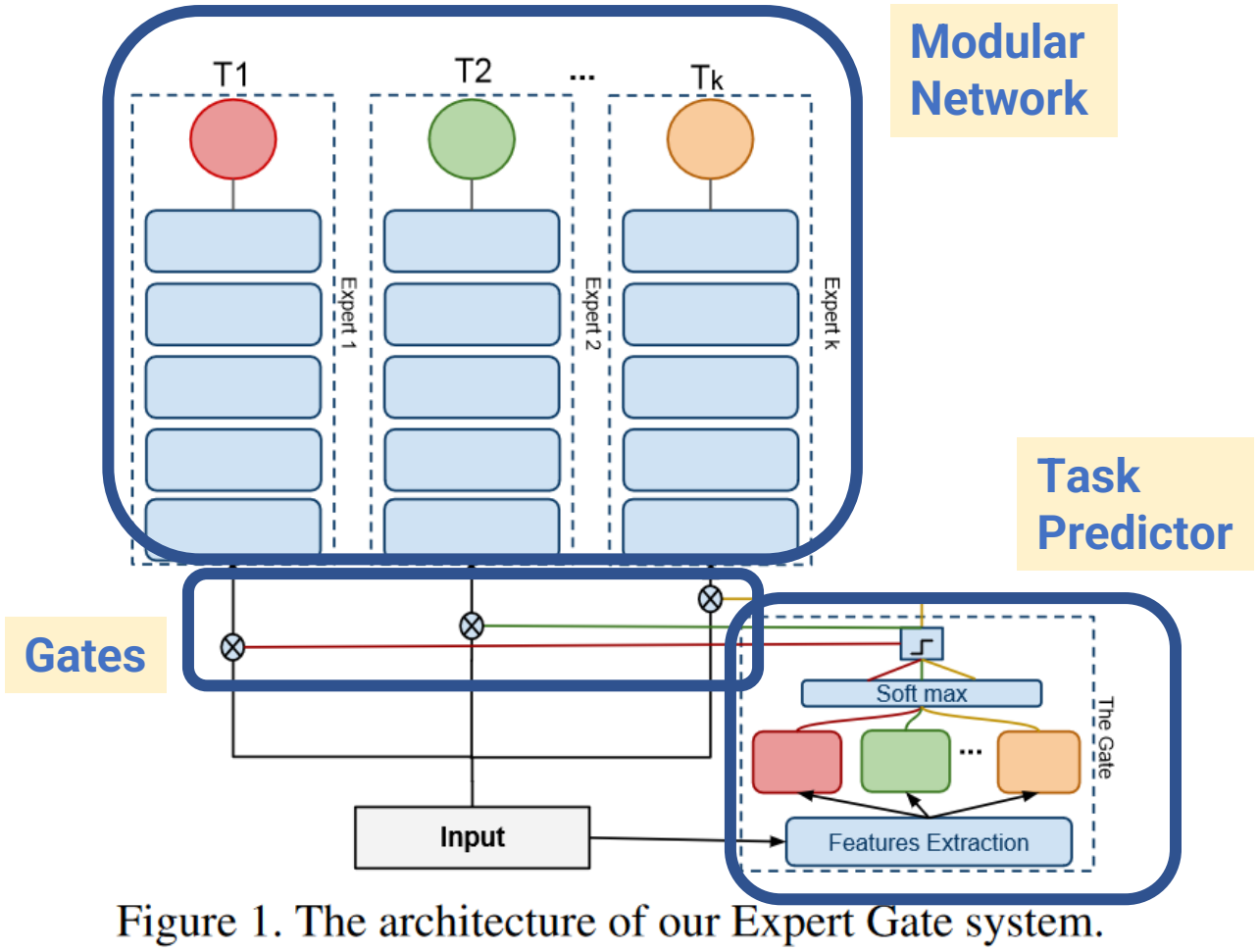


Figure 1. The architecture of our Expert Gate system.

Expert Gate

IDEA: *Modular network with gating and task inference*

- **Expert**: a module of the network trained on a single task
- **Gate**: an undercomplete autoencoder for each task
- **Inference**: use the expert model associated with the most confident autoencoder

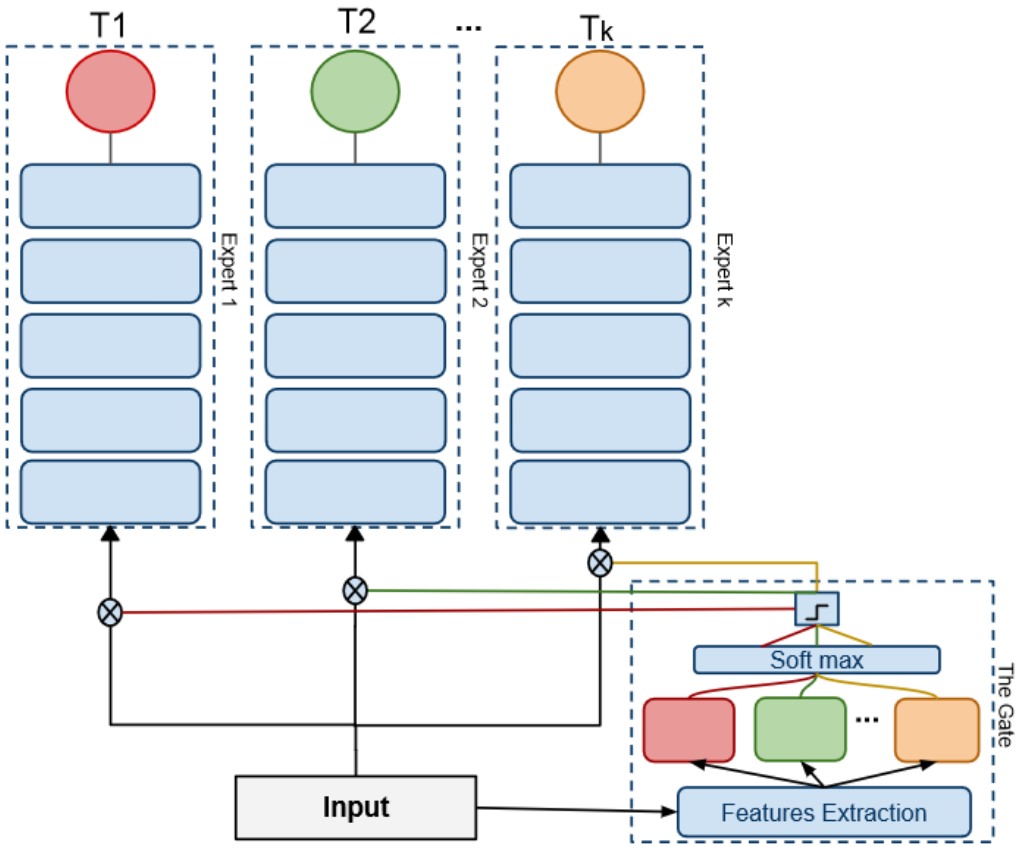


Figure 1. The architecture of our Expert Gate system.

Expert Gate: Gate

- **Pretrained Input Features**: input x to the expert and autoencoder is the output of the last CONV layer of AlexNet pretrained on ImageNet
- **Gate Architecture**: standardization + an undercomplete autoencoder for each task
- **Inference**: use the expert model associated with the most confident autoencoder
  - er_i reconstruction error for task i
  - t temperature

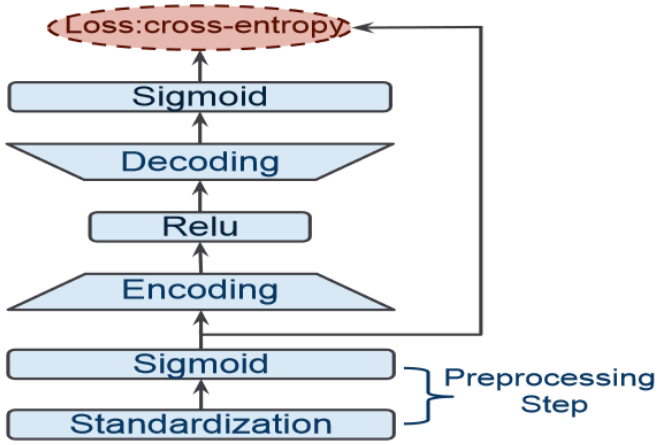


Figure 2. Our autoencoder gate structure.

$$p_i = \frac{\exp(-er_i/t)}{\sum_j \exp(-er_j/t)}$$

Expert Gate: Training the Experts

- **Input**: same as the AE
- **Initialization**: uses the most related expert as initialization
  - **Task Relatedness** Measure
  - Not symmetric (AE_k, D_i) != (AE_i, D_k)
- **Finetuning vs LwF**:
  - Training algorithm of the expert depends on the task relatedness
  - if relatedness is above $threshold$ use LwF, else use finetuning

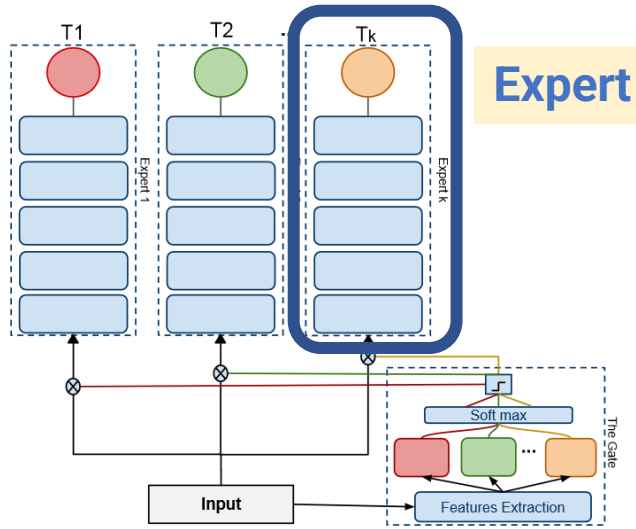


Figure 1. The architecture of our Expert Gate system.

Task Relatedness

$$\text{Rel}\,(T_k, T_a) = 1 - \left(\frac{Er_a - Er_k}{Er_k}\right)$$

Er_a= reconstruction error for the new task (k) using autoencoder a

Expert Gate: Pseudocode

Algorithm 1 Expert Gate

Training Phase input: expert-models $(E_1, .., E_j)$, tasks-autoencoders $(A_1, .., A_j)$, new task (T_k), data (D_k) ; output: E_k

1: A_k =train-task-autoencoder (D_k)
2: (*rel*,*rel-val*)=select-most-related-task(D_k,A_k,{A})
3: **if** *rel-val* >*rel-th* **then**
4: E_k=LwF(E_{rel}, D_k)
5: **else**
6: E_k=fine-tune(E_{rel}, D_k)
7: **end if**

Test Phase input: x ; output: prediction

8: i=select-expert$(\{A\}, x)$
9: prediction = activate-expert(E_i, x)

Aljundi, Rahaf, Punarjay Chakravarty, and Tinne Tuytelaars. "Expert Gate: Lifelong Learning with a Network of Experts.", CVPR 2017

Expert Gate: Final Comments

- Example of task-agnostic modular architecture
- Simple task inference mechanisms
- Insights about the relationships between transfer and task relatedness
- Limitations:
  - Requires pretrained network
  - The reconstruction error is not always a good task predictor. Autoencoders are very good at reconstructing unseen data.

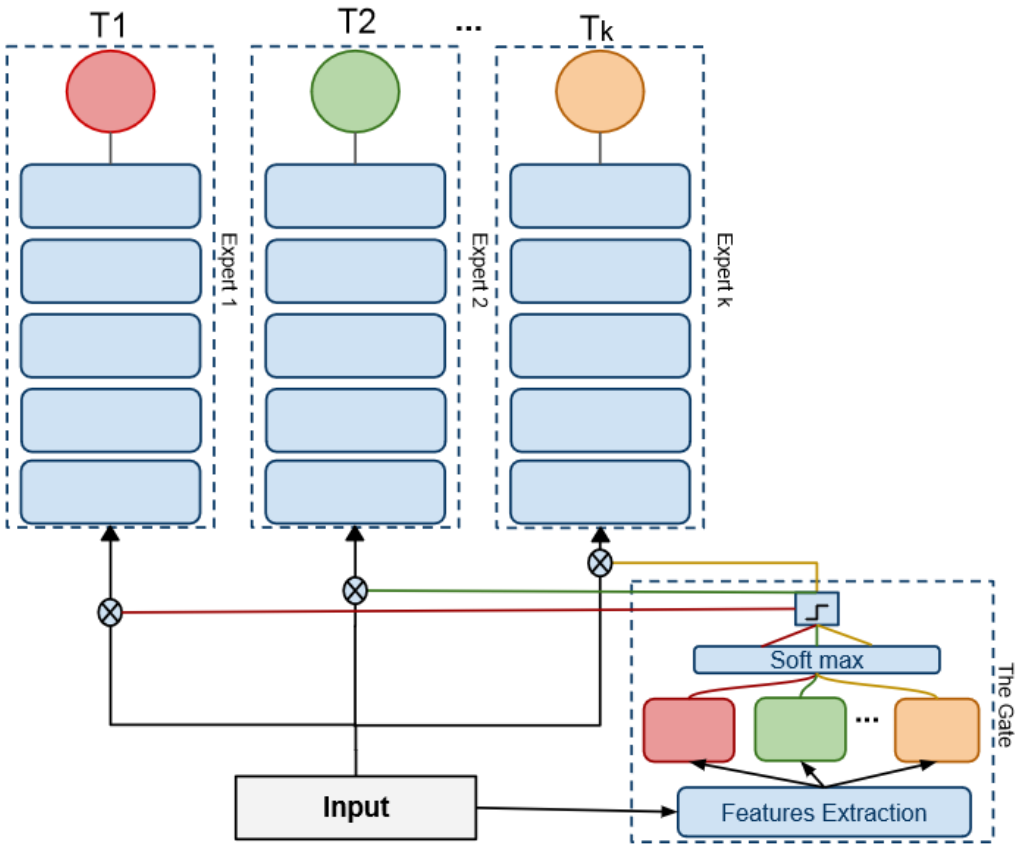


Figure 1. The architecture of our Expert Gate system.

Masking and Architectural Sparsity

Masking – Motivations

increase the memory occupation over time

- We know that deep networks are overparameterized
- **SOLUTION**: use a fixed large network and select a subset of units for each task
- **ADVANTAGES**:
  - Similar to modular networks but less expensive
  - Binary masks are easy to compress
  - Induces sparsity

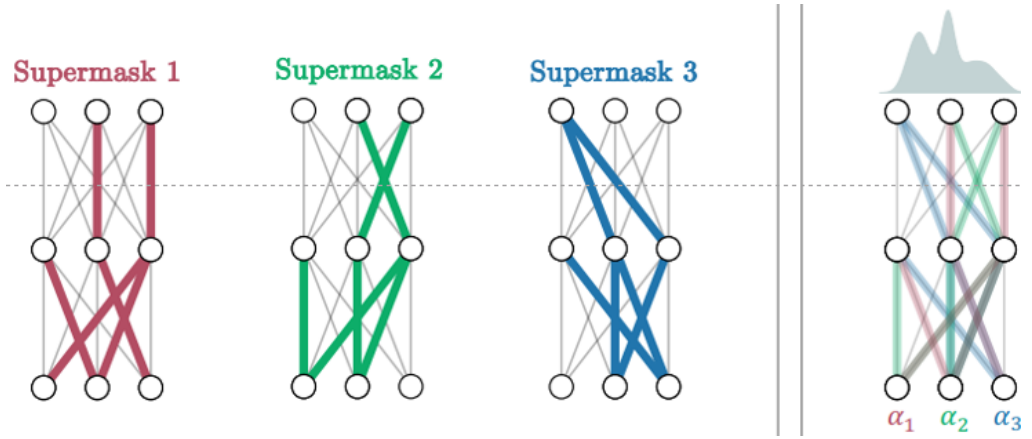


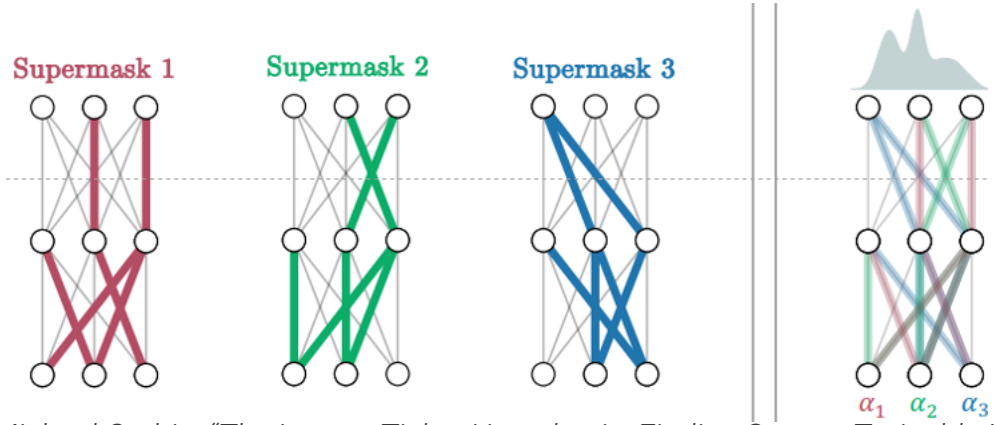
Image from Supermasks in Superposition, Wortsman et al. 2020.

Motivations – Lottery Ticket Hypothesis

HYPOTHESIS - Lottery Ticket Hypothesis:
dense, randomly-initialized, feed-forward networks contain subnetworks (**winning tickets**) that—when **trained in isolation**–reach test accuracy comparable to the original network in a similar number of iterations

WARNING: This is just a hypothesis, not a formal theorem
PROBLEMS:

- How do we optimize binary masks during continual learning?
- How do we do task inference?



Frankle, Jonathan, and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks," ICLR 2019

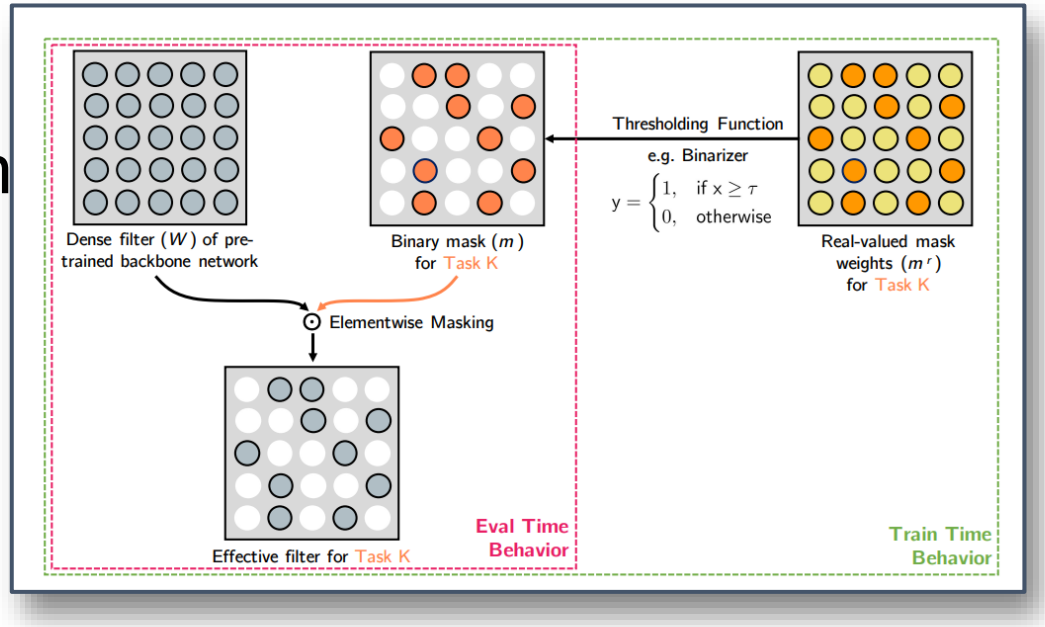
How to Optimize Binary Masks with SGD

- Binary masks are discrete parameters
  - We can't compute gradients
  - You cannot use a straightforward SGD
- We will see some examples but we will not study optimization algorithms for discrete variables in details
- We will assume that they are available and solve the problem of learning masks in CL
- Reference in the footnote if you really want to know more

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." ArXiv Preprint ArXiv:1308.3432, August 2013. http://arxiv.org/abs/1308.3432.

Weights Mask (Piggyback)

- **fixed pre-trained model** (backbone)
  - Example: ResNet18 pretrained on ImageNet
- **Mask training**: A binary mask for each task. each weight, train float then binarize
- Zero forgetting but also Zero knowledge transfer
- **Efficient**: only a handful of KBs per mask (1 bit per weight per task)
- **Task-aware**

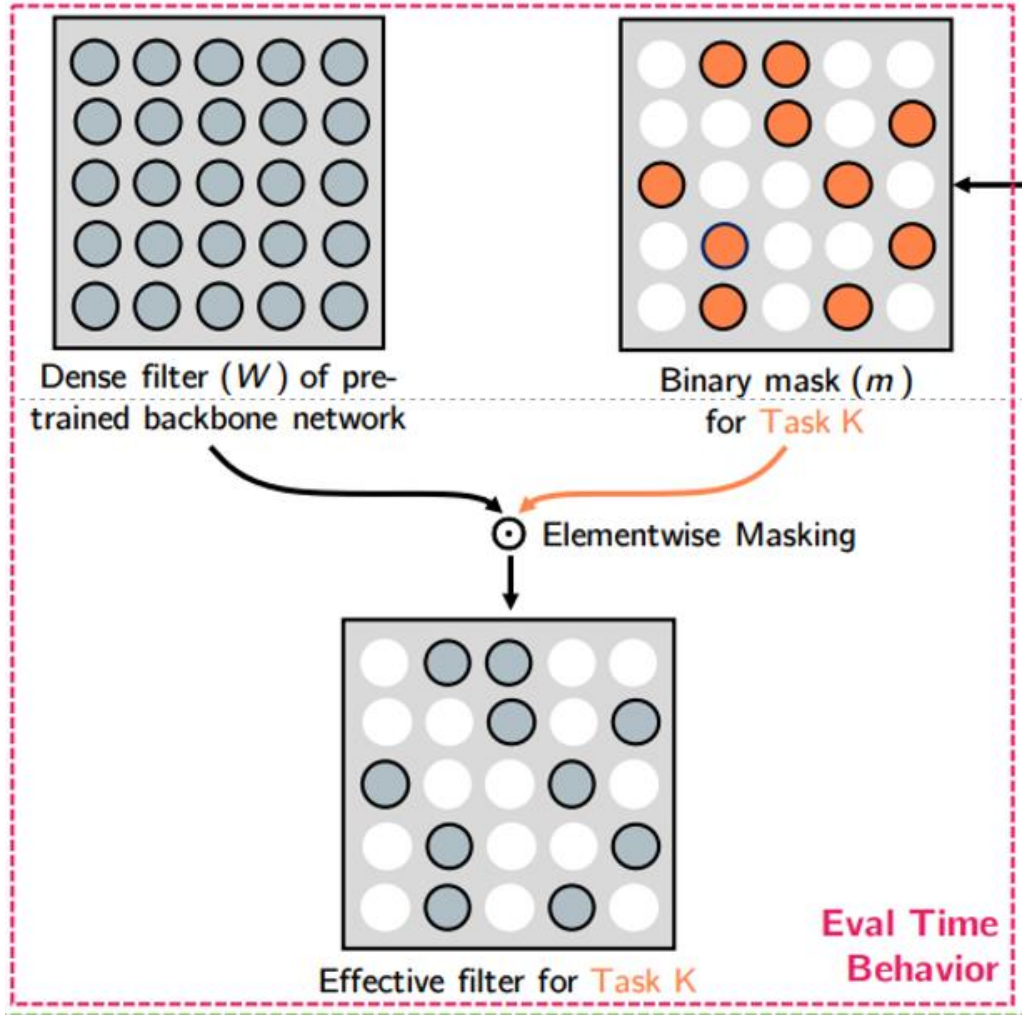


Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights, Mallya et al. 2018.

PiggyBack – Forward Pass

- Separate mask for each layer
- Binary value for each weight
- Masked Feedforward Layer:

$$\mathbf{y} = (\mathbf{W} \odot \mathbf{m})\mathbf{x}$$



Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights, Mallya et al. 2018.

Piggyback – Training and Backward Pass

The binary mask m is obtained from a real mask m^r

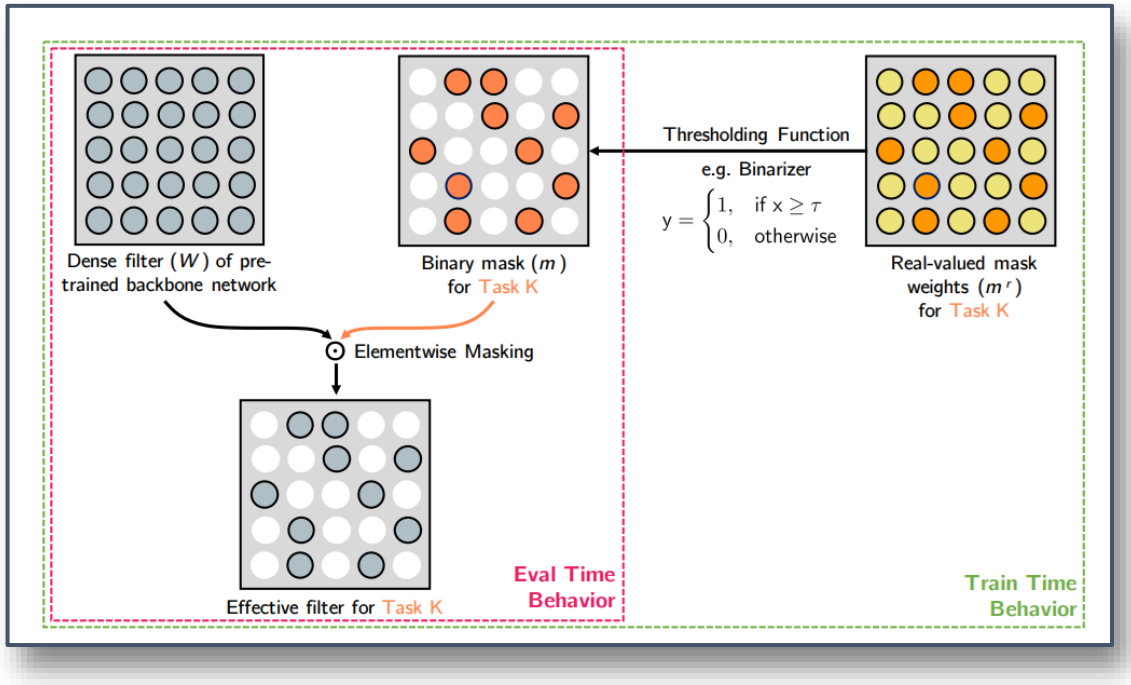
Training step:

- Binarize the real mask m^r with the thresholding function
- Compute the forward pass with the masked layer
- Compute the gradient of the mask
- SGD step

Even though the hard thresholding function is non-differentiable, the gradients of the thresholded mask values m serve as a noisy estimator of the gradients of the real-valued mask weights m^r.

Thresholding $$m_{ji} = \begin{cases} 1, & \text{if } {m^r}_{ji} \geq \tau \\ 0, & \text{otherwise} \end{cases}$$

Masked layer $$\mathbf{y} = (\mathbf{W} \odot \mathbf{m})\mathbf{x}$$



Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights, Mallya et al. 2018.

Masking with Pruning Methods

We can use pruning methods to find a mask

Magnitude Pruning

- Train a network
- Sort the weights in a layer by their absolute magnitude
- Cut the lowest p%

Variation: **Iterative Magnitude Pruning (IMP)**, where the process is repeated multiple times, each time pruning p% and retraining.

PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning, Mallya et al 2017.

PackNet

Magnitude Pruning, task-aware

- **Model**: masked layers $\mathbf{y} = (\mathbf{W} \odot \mathbf{m})\mathbf{x}$
- **Inference**: use task labels to choose mask.
- **Training**:
  - start from a **Pretrained Model.**
  - for each task:
    - **Finetune**: the weights of the dense network (unmasked) on the new task
      - frozen parameters are fixed
    - **Pruning**: prune away a certain fraction of the weights of the network, i.e. set them to zero
    - **Retrain**: to regain accuracy after pruning (half epochs)
    - **Freeze**: Task parameters are frozen.

- 1.5× more expensive than simple finetuning
- What is the advantage compared to Piggyback?
  - We can have forward transfer, because the weights are trained, while in Piggyback they are kept fixed

PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning, Mallya et al 2017.

Hard Attention to the Task (HAT)

IDEA: *learn a soft attention mask over the units for each task. Use the mask in forward to mask units and in backward to mask gradients*

- Task-aware method
- Mask neurons instead of the weights
- Mask gradients during backpropagation (soft masks -> weight sharing)

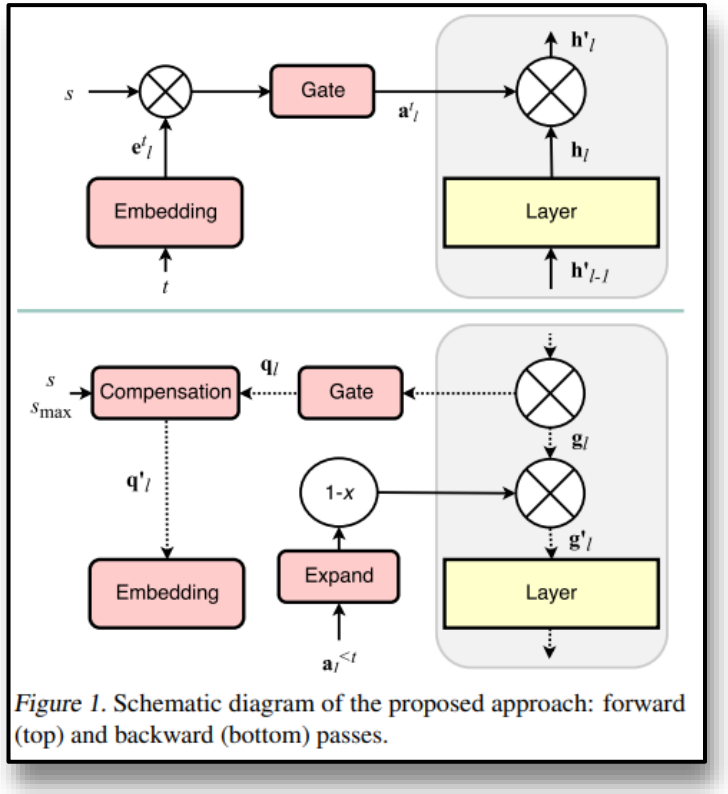
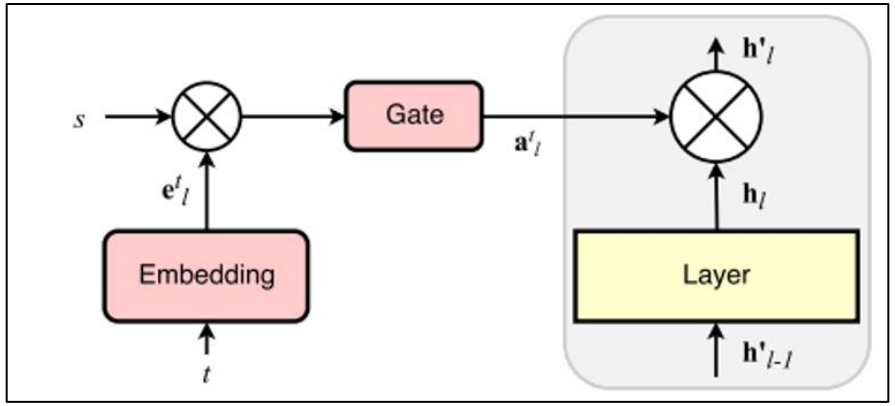


Figure 1. Schematic diagram of the proposed approach: forward (top) and backward (bottom) passes.

Overcoming catastrophic forgetting with hard attention to the task, Serra et al. 2018.

HAT – Forward Pass

- Task embedding e_l^t for each layer l and task t
- Attention over units: $\mathbf{h_l'} = \mathbf{a_l^t} \odot \mathbf{h_l}$
- attention coefficient: $\mathbf{a_l^t} = \sigma(s\mathbf{e_l^t})$
- Soft gates in $[0,1]$
  - Different from Piggyback and Packnext which used hard gates (binary)
- s scalar scaling parameter
  - If $s \to 0$ then $a \to 1/2$.
  - If $s \to \infty$ then $a \to \{0, 1\}$
  - Higher $s \to$ stronger binarization. Regularates between uniform weights and hard gates
  - set $s = s_{max}$ during testing, using $s_{max} \gg 1$ to approximates hard gates



Overcoming catastrophic forgetting with hard attention to the task, Serra et al. 2018.

HAT – Backward Pass

- **cumulative attention vector**

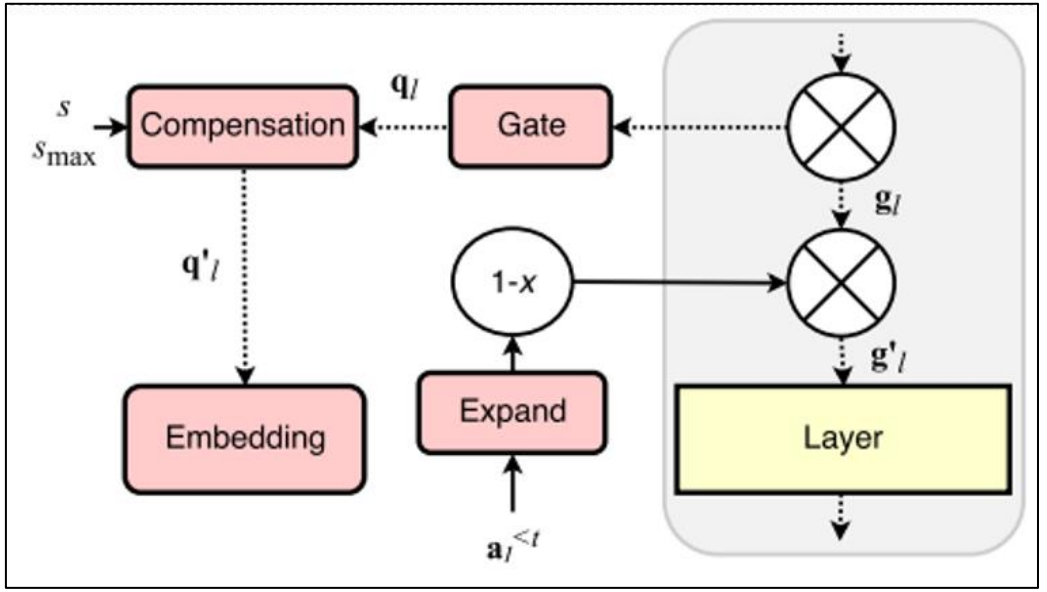
$$\mathbf{a}_l^{\leq t} = \max\left(\mathbf{a}_l^t, \mathbf{a}_l^{\leq t-1}\right)$$

Keeps track of how much each unit is used by the previous task. Used units should be protected from large changes.

- **gradient masking**

$$g'_{l,ij} = \left[1 - \min\left(a_{l,i}^{\leq t}, a_{l-1,j}^{\leq t}\right)\right] g_{l,ij}$$

  - $g_{l,ij}$ = gradient of weight matrix layer l, weight $W_{i,j}$
  - INTUITION: scale down the gradient for already used units.
- **Anneal** s during training, inducing a gradient flow



Overcoming catastrophic forgetting with hard attention to the task, Serra et al. 2018.

Supermasks in Superposition (SupSup)

- Hard masks + fixed backbone
- By now we know that we can find good binary masks even from random weights (Lottery Ticket Hypothesis)

SupSup adds two key contributions:

- If we use random weights we don't even need to store them. We save the random seed and generate the weights on the fly
- Instead of using a single mask, we can use a weighted sum of them (superposition)
  - Task inference becomes the problem of finding the optimal weights!
  - If we can do task inference we can use hard mask + fixed weights even when we don't have task labels during inference or training

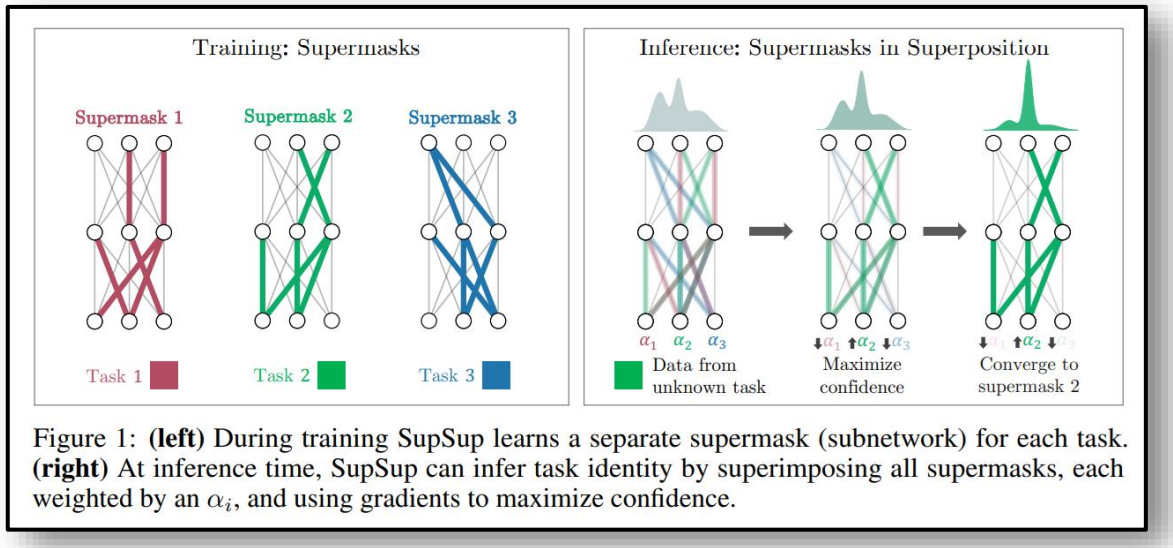
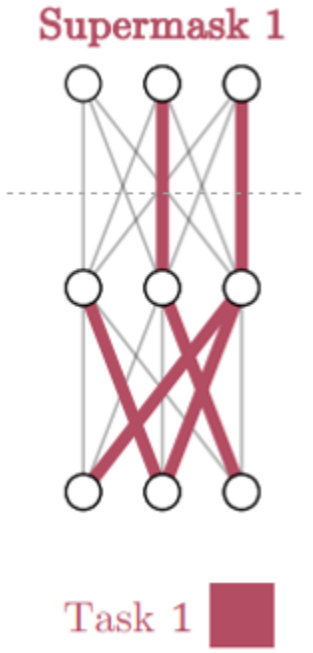


Figure 1: **(left)** During training SupSup learns a separate supermask (subnetwork) for each task. **(right)** At inference time, SupSup can infer task identity by superimposing all supermasks, each weighted by an α_i, and using gradients to maximize confidence.

Supermasks in Superposition, Wortsman et al. 2020.

SupSup – Mask Training

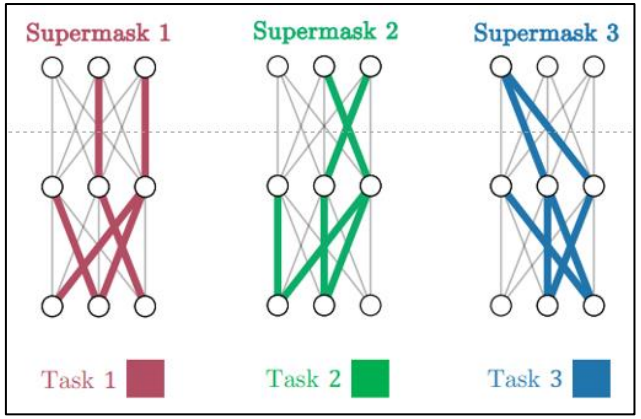
- Masked layer: $\mathbf{p} = f\left(\mathbf{x}, W \odot M^i\right)$
  - Same as Piggyback/Packnet
- W freezata dopo init, bias 0
- signed Kaiming constant init: +-c with equal probability
  - c=std of the kaiming normal initialization
- Edge-Popup algorithm to train the masks (we won't see it)



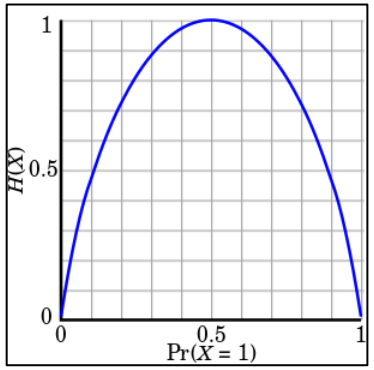
Supermasks in Superposition, Wortsman et al. 2020.

SupSup – Task Inference

- After training, we have one mask (model) for each task
- In a task-agnostic setting, how do we choose the task label?
- Good heuristic: select the most confident model
  - WARNING: Keep in mind that neural networks may be highly overconfident, so this method doesn't always work
- We can use the entropy to measure the confidence



The entropy measures can be used to measure the confidence. For example, the entropy of a coin flip is maximal when p(H)= 0.5



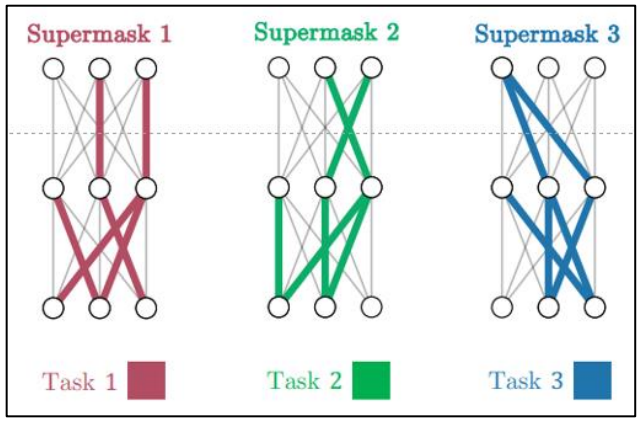
Entropy Eq: $\mathrm{H}(X) := -\sum_{x \in \mathcal{X}} p(x) \log p(x) = \mathbb{E}[-\log p(X)]$

Image source: wikimedia and SupSup paper

SupSup – Naive Task Inference

NAIVE ALGORITHM:

- For each mask:
  - Compute output: $\mathbf{p_i} = f(\mathbf{x}, W \odot M^i)$
  - Compute entropy $H(p_i)$
- Select output with smallest entropy



COMPUTATIONAL COST: require a forward pass for each mask.

We want an algorithm that scales better with the number of tasks

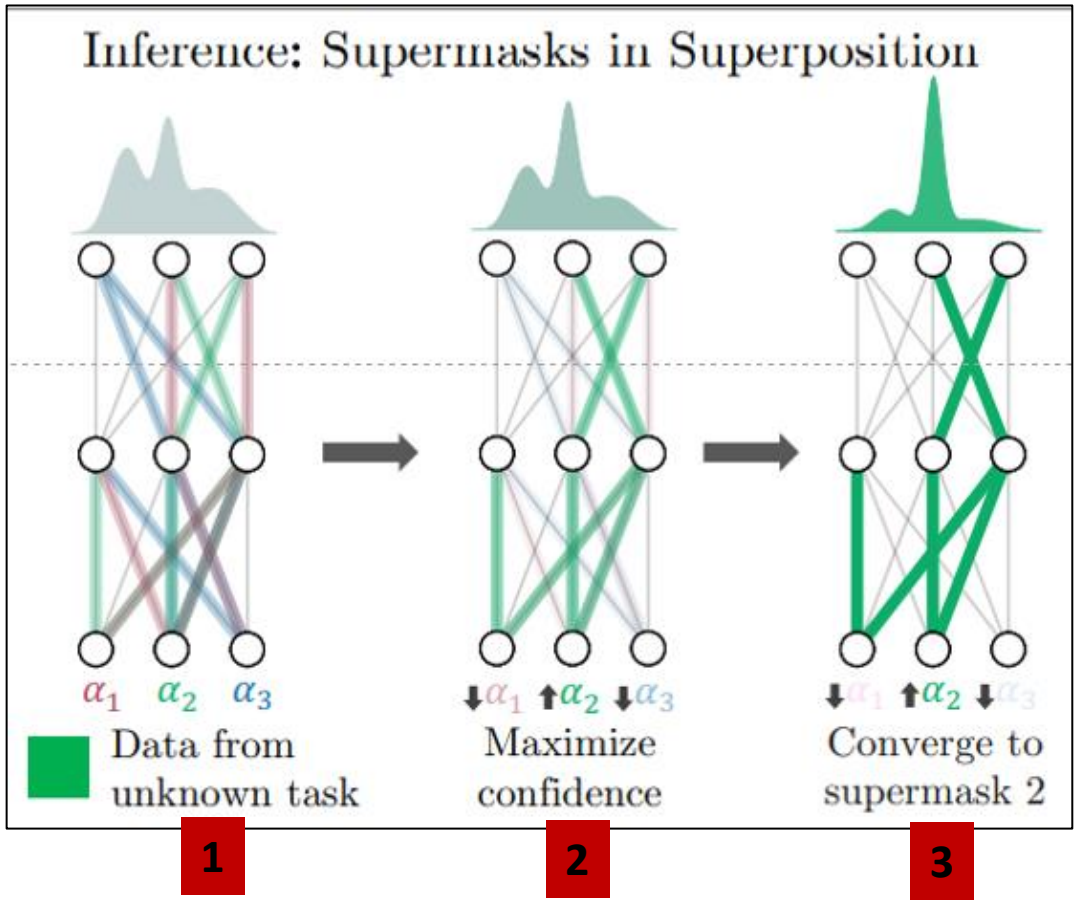
SupSup – Superposition

Superposition: a weighted sum of all the masks

- An approximation of the ensemble output
- $\mathbf{p}(\alpha) = f\left(\mathbf{x}, W \odot \left(\sum_{i=1}^{k} \alpha_i M^i\right)\right)$
- Requires a single forward pass

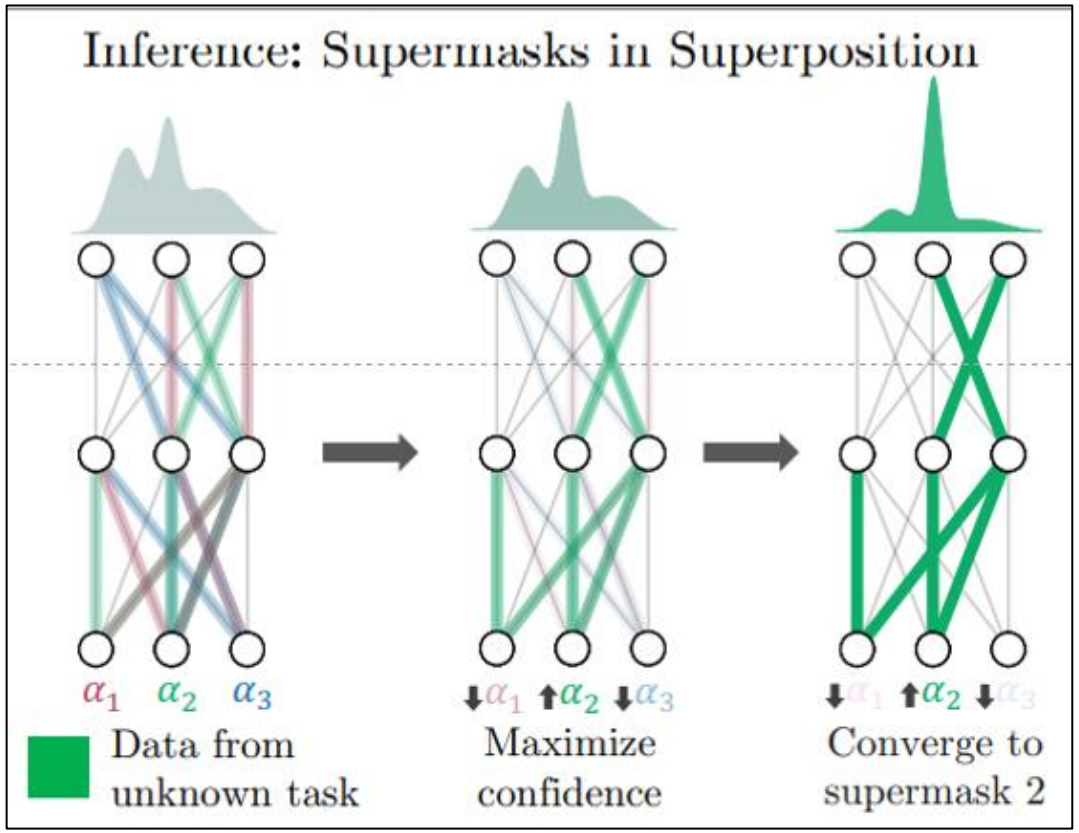
ONE-SHOT TASK INFERENCE:

- (1) Compute $\mathbf{p}(\alpha)$
- (2) Compute gradient with respect to entropy and do an SGD step on α
- (3) Choose the mask s.t. $\arg\max i \left(-\frac{\partial \mathcal{H}(\mathbf{p}(\alpha))}{\partial \alpha_i}\right)$
  - This is a single step of SGD
  - You could optimize α until convergence but one step is sufficient



SupSup – Task-agnostic training

- in a task-agnostic scenario without boundaries we don't know the boundaries between tasks
  - we need a mechanism to decide when to start training a new mask
  - Again, we can use the superposition of mask and the entropy
- **IDEA**: given the coefficients α of the superposition:
  - if some coefficients dominate the others we assume it is an old task
  - If the coefficients are uniform the task is new
- **ALGORITHM**:
  - compute $v = softmax\left(-\nabla_{\alpha\mathcal{H}}(\mathbf{p}(\alpha))\right)$ with optimized α
  - If $\max_i v_i < \frac{1+\epsilon}{k}$ create a new mask
  - o.w. Use mask i with $\max_i v_i$



Masking and Transfer

Transfer: how a learned task affects learning other tasks (can be forward/backward, positive/negative)

- **Fixed weights + independent masks** (Piggyback, SupSup)
  - Zero forward transfer
  - Zero forgetting
  - A pure ensemble of models (very efficient in space occupation)
- **Trainable weights + hard masks** (Packnet)
  - Forward transfer because new masks can reuse units **learned** on the previous tasks (in Piggyback and SupSup the backbone is fixed at initialization)
  - Zero forgetting
  - Similar to PNN, but more efficient in space
- **Trainable weights + soft mask** (HAT)
  - Forward transfer and possibly backward transfer since previously used units are not frozen.
  - Forgetting is possible
  - Soft gates may be harder to train (see HAT tricks about gradient flow)
  - Requires more space than hard gates (32 bit vs 1 bit before compression)

Summary and Take-Home Messages

- **Classifier Bias**: in single-head models, most of the forgetting happens in the output layer, especially in class-incremental settings.
  - There are methods that address this problem
  - Some of these methods assume a static feature extractor or very little changes (DSLDA, CWR*)
  - Others are based on prototypes (cosine classifier) and try to regularize and control how the prototypes evolve over time
- **zero forgetting**: Architectural methods provide a very good solution to forgetting by selective freezing
- **Modular** expansion or selective **masking** provide different **tradeoffs**
  - Modular networks are better at forward transfer
  - Masking has a lower memory cost
- **Strong assumptions about the scenario**
  - Task labels. Task inference (SupSup) exists but its performance highly depends on the domain
  - Size of the experience: if we train each subnetwork independently we need enough data to reach a good performance. They may be difficult to apply in online scenarios without hard task boundaries
- Interesting link with structural plasticity in biological learning systems