A Coding schemes

In this section we present the construction methodology of coding schemes. We drop the subscript l from r_l and $N_{act,l}$ and show the general rule for an arbitrary block for which those values are given. For the reader's convenience, we repeat the three rules the coding scheme should comply with. Given some block with ratio r and number of branches/subNNs N, the rules are as follows:

A. The number of "1"s must be equal to $N_{act} = rN$ with N being the codeword length.
B. Seeing the coding scheme as a binary table, with each row representing a class and each column a subNN as in Table 4, we require the sum of each column to be approximately the same.[5]
C. The minimum Hamming distance across the pairs of codewords should be as high as possible.

The first rule is *mandatory* and we only consider codewords with number of "1"s equal to N_{act}. The other two rules would serve as *guidelines* and we try to follow them to the maximum extent possible. Let S_{min} (resp. S_{max}) be the sum of the columns with the minimum (resp. maximum) sum. The second rule is fully satisfied if $S_{min} = S_{max}$. This is not realizable for all ratios r. The ratio r must be chosen taking into account the number of classes K as follows. The number of "1"s in the binary table is KN_{act}. Assuming a coding scheme with $S_{min} = S_{max} = S_{opt}$, the number of "1"s is also equal to NS_{opt}. This brings the equality $KN_{act} = NS_{opt} \Leftrightarrow S_{opt} = rK \in \mathbb{N}$. Hence, a necessary condition to be able to find a coding scheme with $S_{min} = S_{max}$ is that $rK \in \mathbb{N}$.

The number of possible combinations for choosing p elements from a set of n distinct elements is given by $C(n, p) = \frac{n!}{p!(n-p)!}$. A coding scheme where each class is mapped to a distinct codeword exists if $C(N, rN) \geq K$. Therefore the chosen r should satisfy this inequality.

Let H_{min} be the minimum Hamming distance within the set of all possible pairs of codewords in the coding scheme. In CIFAR-10 for instance, as it can be verified from Table 4, we have $H_{min} = 4$ for both $r = 5/10$ and $r = 3/10$ (check the pair horse-ship). Obviously, the higher $C(N, rN)$ is, the larger the set of acceptable codewords to choose for the coding scheme is, and the larger the H_{min} that can be achieved. Nevertheless, not every value of H_{min} is achievable. There is a value above which there is no such coding scheme with K codewords.

	$r = 5/10$	$r = 3/10$
airplane	1010100011	1001001000
automobile	0101010101	0110001000
bird	1101100010	1010000001
cat	0011001101	0000001110
deer	1010010101	0001010100
dog	1001001110	0010100100
frog	1011101000	0000110010
horse	0100011110	0100010001
ship	0110111000	0001100001
truck	0100110011	1100000010

Table 4: The coding schemes used in CIFAR-10.

After choosing r, N such that $rK \in \mathbb{N}$ and $C(N, rN) \geq K$, we proceed to find the coding scheme. Finding a coding scheme that satisfies the three rules mentioned above is very challenging. Actually, there is no known way to compute even the "basic" function $A_2(N, \mathsf{d})$ that gives the maximum number of binary codewords of length N with minimum Hamming distance d. Moreover, computing the function $\mathsf{D}(N, K)$ that gives the minimum possible Hamming distance of a coding scheme of K codewords is even harder. Using $\mathsf{D}(\cdot)$ one can evaluate $A_2(\cdot)$ through a binary search over K. In our case, the additional constraint of having N_{act} "1"s increases the difficulty. Lastly, in addition to knowing the existence of such a coding scheme, we are interested in realizing it, i.e., generating a valid set of codewords for that scheme. For that, we have to resort to heuristics that satisfy to the largest extent the three aforementioned rules. In Algorithm 1 we give the pseudocode of the algorithm used to generate the coding schemes of CIFAR-100 and ImageNet. For CIFAR-10 the length of the codewords is $N = 10$, which is small enough to allow for the use of a brute force approach similar to exhaustive search.

Algorithm 1 starts by creating a list of length $C(N, rN)$ with all possible codewords satisfying rule A. Each codeword can be mapped to the integer whose binary representation matches the codeword. These integers are used to sort the list of codewords in line 12. This step is crucial, since randomly picking codewords is very inefficient for creating the set G in the subsequent lines. The set G is a set

[5]Note that due to the first rule the sum of each row is equal to N_{act}.

Algorithm 1 Algorithm for generating a coding scheme

Require: K, N, N_{act}, H_{min}

1: **function** MINHAMDIST(codeword w, set G)
2: $d \leftarrow \infty$
3: **for** w_g in G **do**
4: **if** HammingDistance$(w, w_g) < d$ **then**
5: $d \leftarrow \text{HammingDistance}(w, w_g)$
6: **return** d

7: **function** SCORE(coding scheme C)
8: $S_{min} \leftarrow \min\{\text{sum column of coding scheme } C\}$
9: $S_{max} \leftarrow \max\{\text{sum column of coding scheme } C\}$
10: **return** $S_{max} - S_{min}$ ▷ The lower, the better

11: $L \leftarrow$List of all codewords with N_{act} ones
12: $L_{sorted} \leftarrow \text{sort}(L)$
13: $G \leftarrow \{\}$
14: **for** w in L_{sorted} **do**
15: **if** MINHAMDIST$(w, G) \geq H_{min}$ **then**
16: $G \leftarrow G \cup \{w\}$
17: **if** cardinality of $G < K$ **then**
18: exit ▷ Unable to find a coding scheme
19: $BestScore \leftarrow \infty$
20: **for all** $C \subseteq G$ with $|C| = K$ **do**
21: $score \leftarrow$SCORE(C)
22: **if** $score = 0$ **then**
23: **return** C ▷ Found solution satisfying rule C
24: **else if** $score < BestScore$ **then**
25: $BestScore \leftarrow score$
26: $BestSchemeFound \leftarrow C$
27: **return** $BestSchemeFound$

of codewords in which all possible pairs of codewords belonging in this set have Hamming distance between them at least H_{min}. The larger G is, the easier it is to find a subset of cardinality K that satisfies all three rules.

We now give some intuition on why picking codewords randomly from the set L (see line 11) would result in a much smaller set G than the method proposed that picks them sequentially from L_{sorted}. Consider the following analogy: imagine having disks with radius of H_{min} instead of codewords. The problem is to fit as many non-overlapping disks as possible inside a square. If we start filling the square by randomly placing the disks inside the square, this will quickly result in no extra disk actually fitting within the space left by the already placed ones, even though there is still a lot of space unoccupied. On the other hand, if the disks are placed in an ordered way, for example starting from the edges and progressively placing them as close as possible to the already placed disks, then many more disks will eventually fit.

Algorithm 1 is a simplified version of our implementation. In line 20, the number of possible sets C that can be chosen from G can be extremely large. In that case, we resort to additional heuristics for picking only good candidates for C. Further details can be found in our Python code. Finally, the coding scheme for CIFAR-100 with $r = 8/20$ is retrieved using the arguments $(K, N, N_{act}, H_{min}) = (100, 20, 8, 8)$ in the Algorithm 1 and with $r = 4/20$ using $(K, N, N_{act}, H_{min}) = (100, 20, 4, 4)$. For both ratios the coding schemes found entirely satisfy rule B., i.e., $S_{min} = S_{max}$. The coding scheme for ImageNet with $r = 16/32$ is retrieved using arguments $(K, N, N_{act}, H_{min}) = (1000, 32, 16, 10)$ achieving $S_{min} = 499 \approx S_{max} = 501$. For the coding scheme with $r = 8/32$, we use arguments $(K, N, N_{act}, H_{min}) = (1000, 32, 8, 6)$, achieving $S_{min} = 249 \approx S_{max} = 251$.

B Implementation details and computational cost

We present here some additional implementation details and show the computational cost for training a Coded ResNeXt network with respect to the cost of training a conventional ResNeXt.

As shown in Fig. 1(a) and (b), each path/subNN of the ResNeXt and Coded ResNeXt blocks consists of three layers. Each of the first two layers is composed of a convolutional operation followed by a batch normalization (BN) [27], and a rectified linear unit (ReLU) [48]. For the ResNeXt block, in the last layer, after the convolutional operation the output of all paths/subNNs is aggregated by summation, followed by BN and ReLU. Similarly, for Coded ResNeXt, there is a BN and a ReLU operation that come after aggregating the output of all subNNs. In Fig. 1(a) and (b) those two operations would be depicted between the two summations.

We run all our experiments on Google's Colab TPU-v2 ($N_w = 8$ cores with precision bfloat16). We used PyTorch's implementation of stochastic gradient descent with Nesterov momentum [50, 71] equal to 0.9. We used the cosine scheduler [45] that decayed the learning rate until 10^{-5}.

For CIFAR datasets the batch size is picked relatively high to harness TPU speed; the batch size per core is set to $B_w = 64$ (i.e., effective 512). Training on CIFAR is performed for 300 epochs with initial learning rate 0.1 and weight decay [34] equal to $5 \cdot 10^{-4}$. For data augmentation we used RandAugment [10] with $(N_{aug}, M_{aug}) = (3, 4)$ for CIFAR-10 and $(1, 2)$ for CIFAR-100[6], which we applied after the standard pad-and-crop and horizontal flips with probability 0.5.

As mentioned in Section 4.1, in order to make a fair comparison with ResNeXt, on ImageNet [61] we follow the training process proposed by the timm library [1][7]. The epochs are 250 (first 5 as warmup [20] and last 10 cooling down), the effective batch size is 1536 (192 per core), the learning rate is 0.6, and weight decay is 10^{-4}. The input image is first randomly resized and cropped using the standard values of scale and ratio [72] and then horizontally flipped with probability equal to 0.5. RandAugment [10] follows with $N_{aug} = 2$ layers and magnitude $M_{aug} = 7$ (varied using an additive Gaussian noise of a standard deviation equal to 0.5) and also random erasing augmentation [91] with probability 0.4 and 3 recounts. We diverge from the timm's proposed process only on the resolution of the input *training* images. We reduce the resolution to 160 (instead of 224) because on the TPU-v2 of Google Colab the training would require more than three weeks. With the reduced resolution it required approximately 10 days. The final evaluation of the trained model on the validation set is done using resolution equal to 224.

As far as the computational cost is concerned, we focus on ImageNet since it is considerably more demanding in terms of computational resources than the CIFAR datasets, and also because for the implementation we use a library dedicated to ImageNet training [81]. That way we can directly compare ResNeXt with Coded ResNeXt and focus on the computational impact of Coded ResNeXt's additional steps by minimizing the impact that our implementation may have on the performance.

We would like to clarify that in a ResNeXt block (or Coded ResNeXt block with ratio $r = 1$) the last convolutional layer of the block does not have to be implemented as a grouped convolution followed by an aggregation via summation (as it is implied from Fig. 1). Since for those blocks we do not apply the additional operations of Energy Normalization, coding Loss and dropSubNNs, the grouped convolution with the subsequent aggregation by summation of the subNNs' outputs can be combined into a simple convolutional layer. This is how we implement ResNeXt and blocks of Coded ResNeXt with ratio $r = 1$, which also coincides with the way ResNeXt is implemented in the original work [84].

The throughput and average RAM consumption of Table 5 have been measured on the second epoch, the reason being that "generally the first epoch is slow with Pytorch XLA" [81]. For ResNeXt-50 we measure the flops using the library fvcore. For Coded ResNeXt we add to the flops computed for ResNeXt-50 the flops needed for the Energy Normalization step. For a block of N subNNs and output of dimensions $\mathbb{R}^{C \times H \times W}$, the energy normalization requires roughly $3 \times N \times C \times H \times W$ flops. The multiplication by 3 comes from the fact that the energy normalization step first raises in

[6] Those values are chosen in [10] for Wide-ResNet-28-2 [86] which, out of all models presented in that work, seems the most similar to ResNeXt-29.

[7] It is possible that using more recent guidelines like the ones proposed for ResNet in [3] could give even higher accuracy. However, in this case, we would not have available an already publicly reported accuracy for ResNeXt. Therefore, we prefer to follow the recipe of timm's library.

	GFlops	#Params	Throughput	RAM
ResNeXt-50 (32×4d)	2.196	$25.0 \cdot 10^6$	$378 \frac{\text{samples}}{\text{sec}}$	12.3GB
Coded ResNeXt-50 (32×4d)	2.269	$25.0 \cdot 10^6$	$375 \frac{\text{samples}}{\text{sec}}$	12.7GB

Table 5: Computational cost on ImageNet.

element-wise manner the tensor to the power of 2, second it takes the mean, and finally it performs an element-wise division with the square root of the total mean energy. We see in Table 5 that for all metrics, Coded ResNeXt does not introduce any significant additional computational cost when trained on TPU.[8]

One epoch of Imagenet on Google Colab TPU-v2 takes roughly 55 minutes. Due to the platform's constraints, we had to split the training in sessions of 24 hours (i.e., in each session around 25 epochs were executed[9]), save the checkpoint at the end of each session, and start the next session from the latest checkpoint stored. Due to lack of powerful resources, for ImageNet and with the training setup described above, we tested one additional set of hyperparameters, which was $(\mu, p_{drop}) = (4, 0.1)$ (instead of $(2, 0.1)$). With $(4, 0.1)$ we observed better binary classifiers and early decoders, but lower accuracy. Specifically, the accuracy was 79.81% (instead of 80.24%). The accuracy of the early decoders were $(4.6\%, 22.0\%, 32.0\%, 37.3\%, 40.0\%, 43.8\%, 55.6\%, 72.5\%, 75.7\%)$ (instead of $(2.38\%, 8.12\%, 12.7\%, 12.8\%, 14.0\%, 17.0\%, 26.5\%, 65.1\%, 73.4\%)$).

Under a simpler and shorter training procedure that did not require that long period of training (for 150 epochs, with $(\mu, p_{drop}) = (1, 0.1)$ and without random erasing data augmentation) we also tested other combinations of coding schemes. The proposed schemes, which at stages (s3, s4) have ratios $(16/32, 8/32)$, an accuracy of 78.1% was achieved. We tested as well the two following ones, which all led to worse than 77% accuracy: (i) one where stages (s2, s3, s4) had ratios $(24/32, 16/32, 8/32)$, (ii) one with $(16/32, 8/32, 4/32)$, and (iii) finally one where the first 3 blocks of s3 had $r = 16/32$, the last 3 blocks of s3 had $r = 8/32$, and s4 had $r = 4/32$.

C Using early decoders to improve confidence calibration

In Section 4.4 we showed that the way the coding schemes are designed brings additional useful properties. It is possible to stop the evaluation of the network at an intermediate block, measure the energies of the output of each subNN of that block and predict the class of the input image. Therefore, each block trained to comply with a coding scheme can be used as an "early decoder" and produce early predictions. However, the final network's predictions are more accurate. Nonetheless, even when the entire network is evaluated, the early predictions can still be of use. In that case, they can provide a confidence estimation on the correctness of the network's final prediction. As shown in Table 3 (see Section 4.4), the more early decoders agree with the final prediction, the higher are the chances that this prediction is correct. Therefore, Table 3 can be used to obtain confidence levels on the output of a network by verifying how many early decoders agree with the final network's prediction and checking the corresponding line in the table.

We also show that the predictions of the early decoders can also be employed by confidence calibration methods as a source of extra features. Confidence calibration is the problem of predicting probability estimates that are representative of the true correctness likelihood [21]. The softmax operation, which is commonly used as the final operation of the neural network models for classification, provides for each class the likelihood that the input sample belongs to that class. However, it is found in [21] that recent state-of-the-art models provide overconfident predictions. Even when those models predict a wrong class, they erroneously estimate a high probability that their prediction is actually correct. Calibrating the probabilities associated to the predicted class allows reflecting the true correctness likelihood.

[8] We also tried training on a GPU provided by Google Colab, without relying on timm library. Coded ResNeXt was more than two times slower compared to ResNeXt on that hardware. Nonetheless, the training of both ResNeXt and Coded ResNeXt was faster on TPU, hence we kept TPU as our choice of hardware.

[9] For the total required time it should be taken into account that around 45 minutes are needed to fetch the dataset from Google Drive to Google Colab per session.

Assume a multi-class classification problem where the input $X \in \mathcal{X}$ and label $Y \in \{1, \cdots, K\}$ are random variables following some ground truth joint distribution $\pi(X, Y)$. Let $g : \mathcal{X} \mapsto \mathbb{R}^K$ be the neural network trained for this problem. For a sample $(x, y) \sim \pi$, it outputs the logits $z = g(x)$ and predicts that the correct label of x is $\hat{y} = \arg\max z$. Calibrating perfectly this neural network means finding a function $h : \mathbb{R}^K \mapsto [0, 1]$ such that

$$\mathbb{P}(\hat{Y} = Y | \hat{P} = p) = p, \tag{7}$$

with $\hat{P} = h(g(X)), \hat{Y} = \arg\max g(X)$ and $\forall p \in [0, 1]$. The above probability is taken over the joint distribution π. An example can help us understand Eq. (7). Assume that we found an h that is a perfect calibrator, which means it satisfies Eq. (7). Suppose that for $p = 0.8$, there are exactly 100 input samples $x \in X$ such that $\hat{p} = h(g(x)) = 0.2$. Then Eq. (7) says that the neural network $g(x)$ predicts the correct label, i.e., $\hat{y} = y$, for only 80 out of those 100 samples. In other words, the function h correctly estimates that out of all those samples that gave 0.8 likelihood to be correct, exactly 80% of them were indeed correctly labeled by the neural network $g(x)$.

Since in practice we have datasets with finite number of samples, we have to estimate the performance of the calibrating function h by approximating the probability in Eq. (7) [52, 47]. To do that, we break the interval $[0, 1]$ into M interval bins of equal size and let those bins be $I_m = (\frac{m-1}{M}, \frac{m}{M}], m \in \{1, \cdots M\}$. If D is a given dataset, then let $B_m = \{(x, y) \in D : h(g(x)) \in I_m\}$ be the subset of samples for which the calibrating function h gives that the predicted label is correct with probability belonging to the interval I_m. An unbiased estimator of $\mathbb{P}(\hat{Y} = Y | \hat{P} \in I_m)$ is

$$\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{(x,y) \in B_m} \mathbb{1}(\hat{y} = y) \tag{8}$$

with $\mathbb{1}$ being the indicator function. The average confidence within bin B_m is defined as

$$\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{(x,y) \in B_m} \hat{p} \tag{9}$$

with $\hat{p} = h(g(x))$. The function $\text{acc}(\cdot)$ approximates the left-hand side of Eq. (7), while $\text{conf}(\cdot)$ does it for the right-hand side. Therefore, the closer those two values are, the better the calibration function h is.

Two metrics used to evaluate the calibration are [47]:

- Expected Calibration Error (ECE), which approximates $\mathbb{E}[|\mathbb{P}(\hat{Y} = Y | \hat{P} = p) - p|]$ as

$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{|B|} |\text{acc}(B_m) - \text{conf}(B_m)|,$$

- Maximum Calibration Error (MCE), which approximates $\max_{p \in [0,1]} |\mathbb{P}(\hat{Y} = Y | \hat{P} = p) - p|$ as

$$\text{MCE} = \max_{m \in 1, \cdots, M\}} |\text{acc}(B_m) - \text{conf}(B_m)|.$$

ECE measures on average how well calibrated the function h is and MCE provides the maximum error of the predicted likelihoods. MCE can be a valuable metric in high-risk applications where good reliability guarantees should be provided even in the worst-case scenario.

Surprisingly, in [21] it was found that the calibration function that works the best is simply a softmax function with a single parameter T regulating the scale (also known as "temperature"). This is an extension of Platt scaling [56, 52] and the calibration function used is

$$h_T(z)_k = \frac{e^{z_k/T}}{\sum_{j=1}^{K} e^{z_j/T}}, k \in \{1, \cdots, K\} \tag{10}$$

where $z \in \mathbb{R}^K$ is a vector (representing the logits produced by the model, i.e., $z = g(x)$) and z_k is the k-th element of that vector. The value of the parameter T is chosen by minimizing the negative log-likelihood on a hold-out validation set [21]. The minimization is performed using gradient descent updating only the parameter T. We emphasize that since the parameters of the network are kept fixed,

	h_T of Eq. (10) from [21]	h_{aug} of Eq. (11) Ours	h_T of Eq. (10) from [21]	h_{aug} of Eq. (11) Ours
CIFAR-10	(0.70%, 24.35%)	(0.70%, **24.14**%)	(0.80%, 24.83%)	(0.80%, **23.64**%)
CIFAR-100	(1.75%, 10.28%)	(**1.71**%, **9.31**%)	(1.82%, 11.37%)	(**1.79**%,**10.44**%)
ImageNet	(**2.98**%, 8.32%)	(3.00%, **7.70**%)	(**3.01**%, 8.56%)	(3.05%, **8.10**%)

(a) Split set into: train set 25%, test set 75%. (b) train set 50%, test set 50%

Table 6: The performance in terms of (ECE(%), MCE(%)) of the two calibration methods using temperature scaling. The first one h_T is proposed in [21] and has only one parameter. The second h_{aug} has additionally one parameter per early decoder and takes into account the prediction of the early decoders.

the network predictions have not changed. h_T calibrates the likelihood probabilities associated to the class predicted by the network.

We propose now a simple way to take into account the predictions of the early decoders. Given an input sample, suppose that $\hat{e}_i, i \in \{1, \cdots, N_{dec}\}$ is the prediction of the i-th early decoder, N_{dec} is the number of early decoders, and $\hat{y}$ the final prediction. Our proposed architecture for CIFAR has $N_{dec} = 6$ and for ImageNet $N_{dec} = 9$. We intend to adjust the parameterized function h_T by adding another N_{dec} parameters, denoted τ_i, which account for the predictions of the early decoders. Specifically, *the calibration function* we employ is

$$h_{aug}(z)_k = \frac{e^{z_k/T_{aug}}}{\sum_{j=1}^{K} e^{z_j/T_{aug}}}, \text{ with } T_{aug} = T\left(1 + \sum_{i=1}^{N_{dec}} \mathbb{1}(\hat{e}_i = \hat{y})\tau_i\right) \text{ and } k \in \{1, \cdots, K\}, \tag{11}$$

which has $N_{dec} + 1$ parameters. If the i-th early decoder agrees with the final prediction $\hat{y}$ then the temperature is adjusted by adding the term τ_i. The parameters are optimized in the same way as for h_T, using gradient descent with respect to the negative log-likelihood. We initialize all τ_i to be zero and T to be one. The learning rate is 0.1 and we perform 600 iterations.

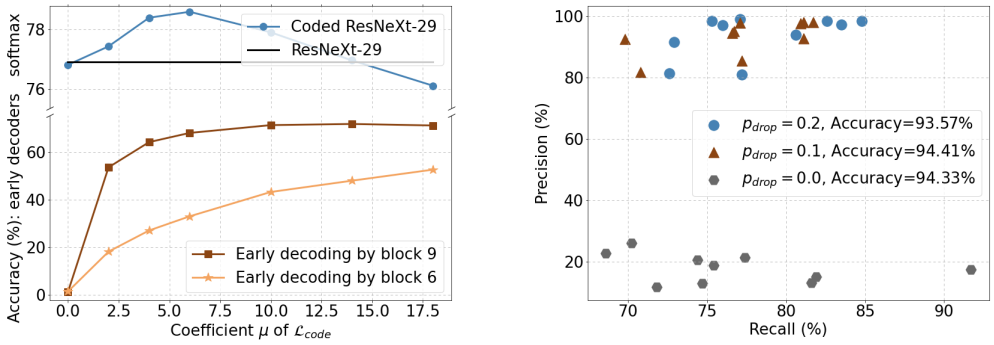
We start the experimental procedure by randomly splitting the validation set of each dataset into two sets. The first contains a quarter of the total samples and the second the rest. The splitting is performed in a way such that within the set, all classes have approximately the same number of samples. We use the first set to train the parameters of h_T and h_{aug}. We test their performance on the second set by estimating the ECE and the MCE, using $M = 15$ bins as in [21]. We repeat the procedure 40 times more, each time with a different random split of the validation dataset. The results are shown in Table 6a. In Table 6b we conduct the same experiment but now we split the validation set into two sets of equal size. Across all datasets and for both splitting ratios we see that h_{aug} improves the MCE with almost no impact on ECE with respect to h_T.

D Ablation Study on Coding Loss and dropSubNN

In this section, we study the effect of the two hyperparameters introduced in the paper, namely the coefficient μ balancing the losses in Eq. (6) and the probability p_{drop} of dropping subNNs.

In Fig. 5a we show that increasing μ and so forcing more the energies of the subNNs to comply with the coding scheme, is at first beneficial to the overall performance until a certain point (see blue line). Past this point (in Fig. 5a it is around $\mu = 6.0$), forcing the subNNs to output a signal of a specific energy value provides only small diminishing gains on the early decoders (see increasing trend of brown and beige lines). Furthermore, it disturbs the entire classification process, and the final accuracy of the whole network's predictions starts declining.

The second experiment concerns the dropSubNN and its hyperparameter p_{drop}. Dropping randomly some subNNs during training inhibits their "co-adaptation"[69], as they learn not to depend on the others and to perform well even in the absence of some of them. Figure 5b shows that the dropSubNN is essential for the good performance of the binary classifiers. A small value of $p_{drop} = 0.1$ can greatly boost their performance and even slightly improve the overall accuracy. Further increasing it to $p_{drop} = 0.2$ degrades the accuracy without improving much the binary classifiers.



(a) Impact of coefficient μ of the coding loss on CIFAR-100.

(b) Impact of the probability p_{drop} on the performance of the binary classifiers.

Figure 5: Ablation study on the two hyperparameters introduced in this paper, i.e., the coefficient μ balancing the losses in Eq. (6) and the probability p_{drop} of dropping subNNs.

E Limitations

A limitation regarding the proposed idea is that it cannot be applied to any type of multi-branch architecture. As we explain in Section 4.3.1 the method works for ResNeXt because the output of all branches of every block is conveniently aggregated by summation. This avoids having the blocks relying on the output of the inactive subNNs to learn the classification, which would then prevent the extraction of the binary classifiers by just keeping the active subNNs. Instead, since (i) the blocks receive the aggregated outputs of the previous block, and (ii) the outputs of the inactive subNNs are pushed to zero by the coding loss, during training the blocks are forced to solely rely on the aggregated signal provided by the active subNNs. Interestingly, there are other architectures that have a structure similar to that of ResNeXt and where our idea could also be applied. An example is MobileNetV2 [62], which uses a type of block called MBConv (a block that later used also by EfficientNet [74]). To see how our idea could be applied to this kind of blocks, let us define the following neural network module in PyTorch:

Pytorch sequential code

Require: channels_in, channels_mid, channels_out, N, s

```
module = nn.Sequential(
    nn.Conv2d(channels_in, channels_mid, kernel_size=1),
    nn.BatchNorm2d(channels_mid),
    nn.ReLU(),
    nn.Conv2d(channels_mid, channels_mid, kernel_size=3, padding=1, groups=N, stride=s),
    nn.BatchNorm2d(channels_mid),
    nn.ReLU(),
    nn.Conv2d(channels_mid, channels_out, kernel_size=1),
    nn.BatchNorm2d(channels_out)
    )
```

A typical ResNeXt block is formed by simply adding a residual connection to the above module, i.e., module(x)+x where x is the input of the block.[10] This ResNeXt block has N branches/subNNs. The MBConv uses an almost identical module, with the slight differences that (i) the activation function ReLU6($\cdot$) is used instead of ReLU($\cdot$), (ii) channels_mid takes a larger value than channels_out, whereas in ResNeXT it takes a smaller one, and (iii) it forces N = channels_mid so the second convolution is depth-wise. Overall, the MBConv can be seen as a type of ResNeXt block but with

[10] We assumed that channels_in = channels_out, and therefore there is no expansions of the number of channels which usually happens in the beginning of each stage.

the number of subNNs equal to the channels_mid. We therefore expect a "Coded-MobileNetV2" to behave the same as Coded-ResNeXt and exhibit the same properties.

Another limitation of our work is given by the trade-off between the specialization of the subNNs and the performance of the entire neural network model. This relationship is clearly depicted in Fig. 5a. To force the subNNs to specialize in a specific set of classes we introduced the "coding loss" $\mathcal{L}_{code}$. The balance between this loss and the cross entropy loss (i.e., $\mathcal{L}_{class}$) is regulated with the hyperparameter μ. As μ increases, more emphasis is given on the subNNs activated according to the coding scheme. As shown in Fig. 5a, increasing μ at first helps the total performance (accuracy) of the entire network. However, after a certain point, increasing the specialization of the subNNs comes at the expense of the performance of the entire network.

We came across this trade-off also when designing the ratios r of the coding schemes. Ideally, we would like the ratios to be as small as possible. This would mean that each subNN would be assigned to specialize to smaller set of classes. Consequently, the information paths of the classes would be more disentangled, in the sense that the paths associated to two different classes would have less shared parameters. Moreover, the extracted binary classes would have even less parameters. Unfortunately, we could not decrease more the ratios without negatively impacting the total model's accuracy. For example, in one of our initial experiments (with fewer epochs)[11] on ImageNet, we found that if we used the proposed ratios of $(32/32, 16/32, 8/32)$ in stages $(c3, c4, c5)$ we get an accuracy of 78.1%, but trying to increase the subNNs' specialization by setting the ratios to $(16/32, 8/32, 4/32)$ drops the accuracy to 76.6%.

Nonetheless, we are confident that there exist ways in which a more drastic specialization of the subNNs is achieved without compromising the performance of the entire network. One alternative that was the subject of some of our tests was zeroing the gradients during training[12], which are directed to the subNNs that according to the coding scheme should remain inactive. That way, the subNNs are updated only by gradients coming from samples of the subset of classes that the coding scheme had assigned to them. Those experiments were performed in a network whose architecture was identical to Coded ResNeXt with the sole difference that the blocks had no skip/residual connections. In that architecture, replacing after Energy Normalization the $\mathcal{L}_{code}$ with an operation that zeroes the gradients according to the coding schemes yielded excellent specialization without degrading the overall network's performance. Unfortunately, this idea did not work well when we added the skip connections. The accuracy of the network was significantly lower when the architecture had skip connections and the gradients of inactive subNNs were zeroed during training.

F Additional information for binary classifiers and activation of subNNs

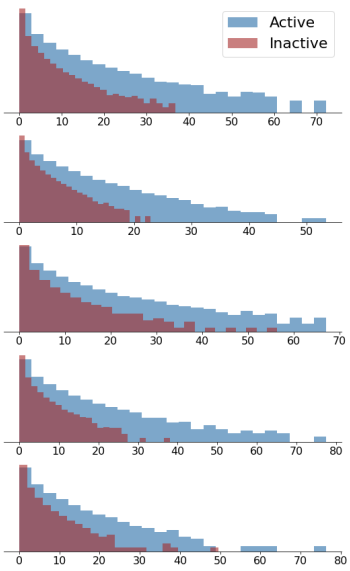
In this section, we provide additional details regarding the binary classifiers that can be extracted after training a Coded ResNeXt, the activation distributions of the subNNs, and their specialization.

F.1 Activation distribution of subNNs

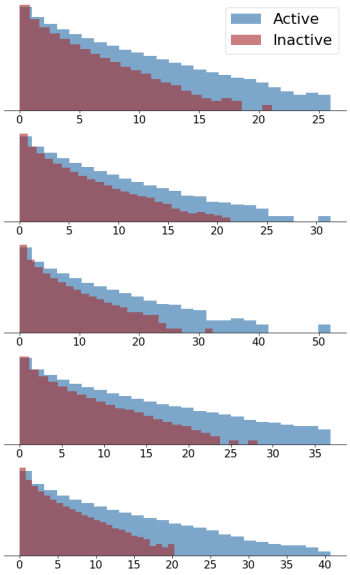
In this subsection we investigate the distribution of the output signal of the subNNs. A sample that passes through a subNN can either belong to the set of classes for which the subNN has to activate, or to the set for which it has to stay inactive. We would like to see how the subNNs respond in these two scenarios by plotting the distribution of the absolute value of the output in each case. In Fig. 6 we plot for CIFAR-10 and CIFAR-100 those distributions for the first 5 subNNs of the second block of stage s3. We show in red how those subNNs react when the sample belongs to the set of classes for which the subNN has to stay inactive according to the coding scheme, and with blue when they have to activate. Since the distributions were very skewed with many values around zero, we show the y-axis in log scale to better appreciate their shape in the complete range. We see that the distribution of the active subNNs has a bigger tail, which confirms that the subNNs output higher values when receiving a sample of their assigned classes.

[11] (150 epochs, with $(\mu, p_{drop}) = (1, 0.1)$ and without random erasing data augmentation)

[12] This can be implemented on PyTorch using the detach function.



(a) Distribution of the absolute value of the output of the first 5 subNNs from the second block of stage s3 on the architecture for CIFAR-10.



(b) Distribution of the absolute value of the output of the first 5 subNNs from the second block of stage s3 on the architecture for CIFAR-100.

Figure 6: Output distribution of active versus inactive subNNs

F.2 Additional details on Binary Classifiers

For convenience, we provide here the definitions of Precision and Recall. True positives TP (resp. true negatives TN) represents the number of positive (resp. negative) samples that the binary classifier correctly predicts as positives (resp. negatives). False positives FP (resp. false negatives FN) represents the number of negative (resp. positive) samples that the binary classifier erroneously predicts as positives (resp. negatives). The precision and recall are defined as follows

$$Precision = \frac{TP}{TP+FP}, \; Recall = \frac{TP}{TP+FN}.$$

Precision measures out of predicted positives (i.e., $TP + FP$) how many of those predictions are correct. Recall measures out of all actual positives (i.e., $TP + FN$) how many were found by the binary classifier.

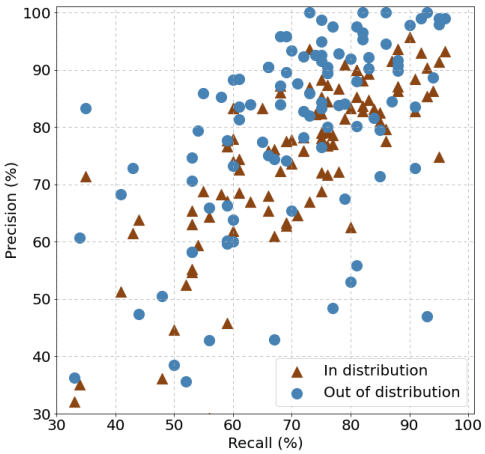
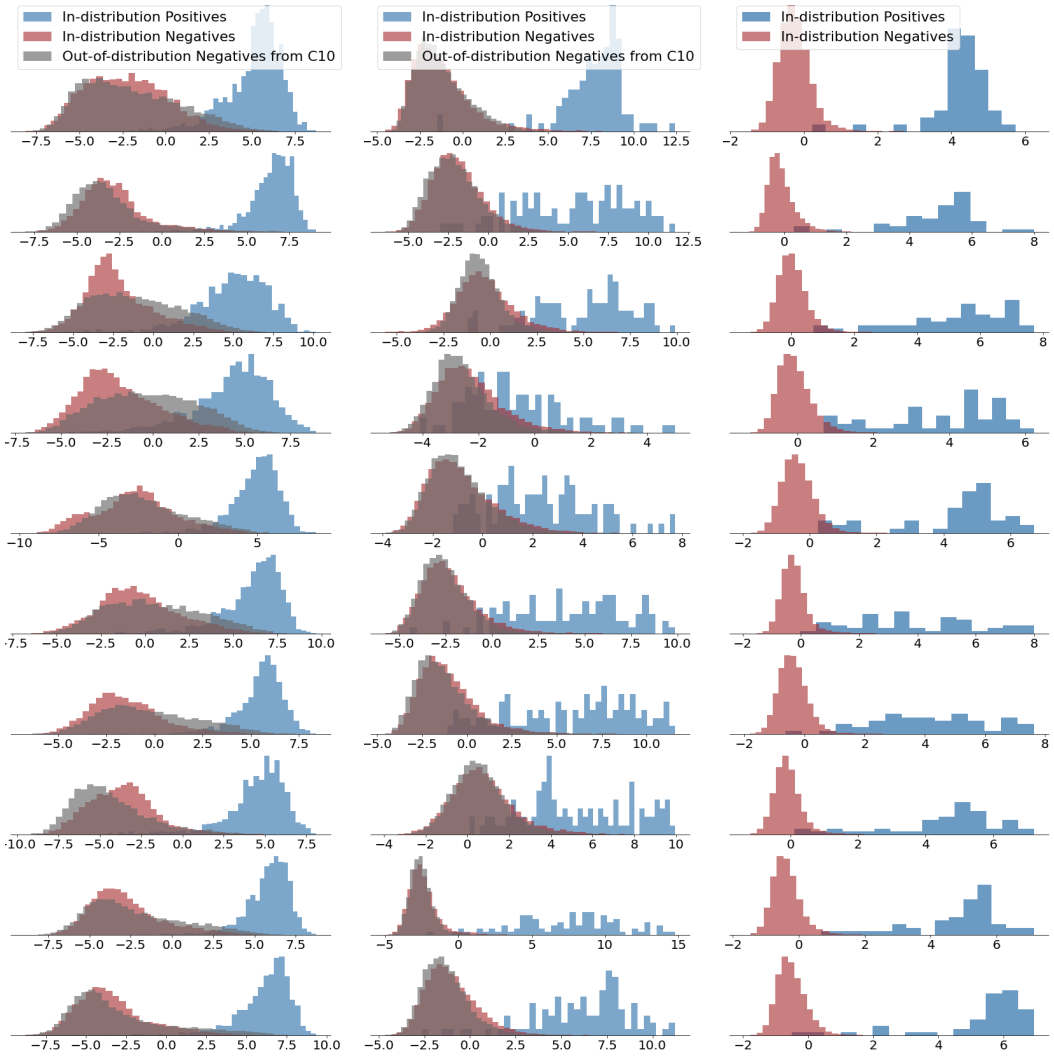


Figure 8: Precision-Recall on CIFAR-100.

In Fig. 7 we give the output distributions of the binary classifiers for the first 10 classes of CIFAR-10/100 and ImageNet when fed with an input of in-distribution positive/negative samples, and out-of-distribution negative samples. The first row coincides with Fig. 3. We observe that depending on the threshold value a binary classifier uses (above which the classifier predicts positive and below negative), different values of precision and recall can be attained. Increasing the threshold value gives fewer false positives FP, but unfortunately more false negatives FN as well, so increasing the threshold improves the precision but degrades the recall. Therefore, high recall can be exchanged for high precision by increasing the threshold, or the opposite by decreasing it.

Regarding plots Fig. 2b, Fig. 8 and Fig. 2c, which depict the precision and recall achieved by the binary classifiers trained on CIFAR-10, CIFAR-100, and ImageNet, respectively, we would like to



(a) All Binary Classifiers extracted from Coded ResNeXt-29 $(10 \times 11\text{d})$ trained on CIFAR-10.

(b) Binary Classifiers for the first 10 classes extracted from Coded ResNeXt-29 $(20 \times 6\text{d})$ trained on CIFAR-100.

(c) Binary Classifiers for the first 10 classes extracted from Coded ResNeXt-50 $(32 \times 4\text{d})$ trained on ImageNet.

Figure 7: Distribution of the output (logit) of binary classifiers

make some remarks.[13] First, when testing a binary classifier in the case of either in-distribution or out-of-distribution negatives, the set of samples of the validation set serving as positives remains the same. Since by definition recall depends only on this set of positives, its value remains unaltered in both testing cases. Second, in the case of ImageNet, we only have 50 samples of the validation set serving as positives for each binary classifier. Therefore, the true positives can only take integer values from 0 to 50 and $Recall \in \{0, \frac{1}{50}, \frac{2}{50}, \cdots, \frac{50}{50}\}$. This is the reason why in Fig. 2c, the points appear to follow some kind of grid and are aligned in specific vertical lines.

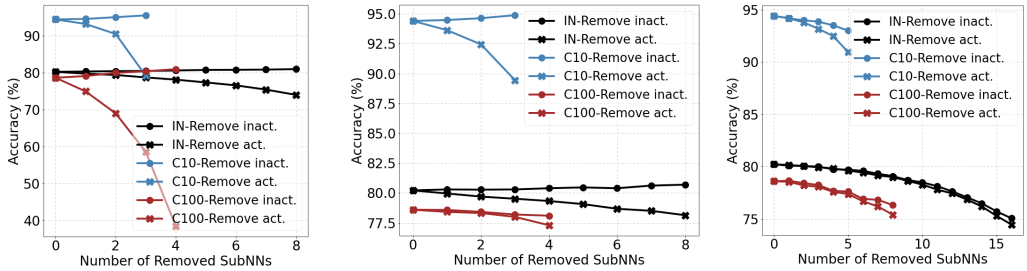
Finally, and quite interestingly, we see that the binary classifiers trained on CIFAR-100 perform at the same level no matter whether the negatives come from in-distribution or out-of-distribution. This is

[13] In Fig. 8, which corresponds to CIFAR-100, we followed the same procedure as in Section 4.3 for ImageNet. Specifically, we randomly sample from the set of negatives a subset of size 9 times bigger than the size of positives. We use that subset of negatives to evaluate the precision and recall of the binary classifier.

in contrast to the binary classifiers trained on CIFAR-10, where out-of-distribution negatives clearly decrease the precision Figs. 2b, 7 and 8). Both CIFAR-10 and CIFAR-100 datasets have the same number of training samples (50000), but CIFAR-10 has 5000 samples per class and CIFAR-100 has 500 per class. Therefore, the binary classifiers of CIFAR-100 have been trained on $\frac{5000}{500} = 10$ times fewer positives but have "seen" negatives from $\frac{100-1}{10-1} = 11$ times more classes. The observation that CIFAR-100 binary classifiers perform significantly better than the ones of CIFAR-10 with out-of-distribution negatives, agrees with our intuition that a model should work better for out-of-distribution data if it has been trained with a high variety of classes where the number of samples per class is small, than if it is trained with the same amount of samples but fewer classes and more examples per class.

F.3 More experiments on removing subNNs

In this subsection, we repeat the experiment described in Section 4.1 for three more blocks. We remind the reader that in that experiment we pick a block l from which we randomly remove subNNs in two ways: given the class of the input image sampled from the validation set, the first way randomly removes $k \leq N_{act,l}$ subNNs from the set of active for that class subNNs. The second way randomly removes $k \leq N - N_{act,l}$ subNNs from the (complementary) set of inactive subNNs for that class. We see that the trend depicted in Fig. 2a, where the performance drops more when active subNNs are removed from a block than when the inactive ones are removed, is also maintained for the blocks in Fig. 9.



(a) Removing subNNs from the last block.

(b) Removing subNNs from the third to last block.

(c) Removing subNNs from the fifth to last block.

Figure 9: Performance when removing active versus inactive subNNs from a specific block.

However, we see that this trend becomes less visible the earlier is the block in which we perform the experiment. For example in Fig. 9c we see that removing either active or inactive subNNs leads to small differences. Removing active subNNs from the earlier blocks has much less negative effect than from the blocks deeper in the architecture. We believe that the reason is that the subNNs in the early blocks produce low level features, which most of the times are useful in general for any class. Therefore, it is challenging to force those subNNs to produce low level features that are exclusively useful to a specific subset of classes. This difficulty is more pronounced in this study, where we design the coding schemes without taking into account the semantics of each class. Stated differently, it is hard, given a set of classes and without looking at the semantic similarities between the classes, to split the set into two subsets and push subNNs to produce low level features that are useful only for one of those subsets and useless for the other. Nonetheless, we expect that increasing μ, which could push the subNNs towards higher specialization, would lead to having the performance degradation due to removing active subNNs observed in the latter blocks also in early blocks.