ChatGPT

Optimizing CIFAR-10 Model Accuracy under 400k Parameters

1. Untried Improvements for Accuracy Boost

Several architecture tweaks and training enhancements remain that could improve CIFAR-10 test accuracy, given the experiments so far:

- **Apply Data Augmentation:** So far *AUGMENT_MODE* was off. Enabling standard augmentations (random flips, crops, slight shifts, etc.) is known to significantly boost generalization on CIFAR-10 [1] [2]. All top CNNs use augmentation to expand the effective training set and prevent overfitting. For example, ResNet padded and randomly cropped CIFAR-10 images, yielding substantial accuracy gains [3], and MobileNet similarly relied on random crop+flip to achieve high accuracy with fewer parameters [4]. Turning augmentation **ON** should be a high priority to improve test performance.

- **Use Learning Rate Scheduling:** No LR schedule was used in previous runs (SCHEDULE_MODE=false). Introducing a schedule (e.g. ReduceLROnPlateau or stepped decay) will allow using a higher initial learning rate for fast convergence while avoiding instability later. This was not tried yet with the current models. Both classical and modern architectures benefit from LR decay. For instance, the ResNet paper used an initial LR of 0.1 and divided by 10 when validation error plateaued, which *"achieves superior results on CIFAR-10"*. VGG also found that manually decreasing the LR when val accuracy stagnates improved final performance. Implementing `SCHEDULE_MODE` (e.g. Keras `ReduceLROnPlateau` with patience ~3 and factor 0.5) should improve convergence and final accuracy, especially for deeper/residual models where a high fixed LR (like 0.01) caused instability.

- **Increase Training Epochs (with Early Stopping):** Previous experiments only ran 8 epochs, which is far too few for CIFAR-10. Typically, 100+ epochs are used to fully converge modern architectures on this dataset. A longer training schedule will allow the models to reach higher accuracy. To avoid overfitting in extended training, enable `EARLY_STOP_MODE` so training can halt when the val metrics stop improving (patience ~10). Early stopping won't directly increase best accuracy, but prevents wasting time or overfitting past the peak. In practice, using ~**100 epochs** with LR drop steps (or automatic plateau reduction) and early-stopping (patience ~10–15) is a robust approach for CIFAR-10.

- **Incorporate L2 Weight Decay:** Previous runs had `L2_MODE.enabled = false`. Adding L2 regularization on weights is a proven strategy to improve generalization by discouraging large weights. The VGG network was trained with weight decay 5×10^−4 [5], and ResNet on CIFAR used 1×10^−4 as well. In fact, *"the training was regularized by weight decay (L2 penalty 5e-4) and dropout"* in the VGG experiments [5]. Enabling a moderate L2 (e.g. λ=5e-4 as in VGG [5] or 1e-4 as in ResNet) on convolution kernels will likely reduce overfitting, especially for larger models.

- **Enable Dropout (in tandem with L2):** Only one run so far tested dropout (m5_drop). Adding dropout layers can further combat overfitting by randomly zeroing activations. VGG applied 50% dropout on its fully-connected layers [5], and many CIFAR-10 models (e.g. Wide ResNets) use dropout in conv layers to reach state-of-art. Dropout forces the network to not rely on any single activation, improving robustness [6] [7]. To avoid under-training, use dropout in moderation (e.g. 0.3–0.5 rate) and **combine it with L2** for best effect. Notably, our prior analysis found that the **dropout+L2 combination** was particularly effective for mid-sized models (see part 2 below), whereas applying heavy regularization to the smallest model hurt its performance. Thus, we should use dropout (and L2) for the larger networks but possibly skip it for the very compact model.

- **Increase Batch Size (moderately):** Past runs used batch_size=8, which was reported to underperform larger batches. A batch of 8 causes noisy gradient estimates and unstable training (the logs showed high variance). Empirically, using **batch size 32** gave consistently better accuracy than 8 or 16 in prior trials [8]. We should fix **BATCH_SIZE=32** going forward for more reliable updates. Batch 32 balances stability and still offers sufficient stochasticity; it also fits in GPU memory easily. (Batch 64 could be tried if resources allow, but prior results indicate 32 was already optimal in our context.) By pruning out batch=8/16, we simplify the search and improve baseline performance.

- **Leverage a Better Optimizer Setup:** The initial runs used `Adam` with a high learning rate (0.01) and an unused momentum parameter. We should adjust this:

- For **Adam**, use a lower LR in the range 0.001–0.003 (as 0.01 was too high without scheduling). Prior successful configs found Adam with LR ≈ 0.002–0.005 worked well [8]. A good default is **Adam, LR=0.003**, β1=0.9. With augmentation and proper regularization, Adam should reach ~80%+ accuracy.

- Alternatively, for deeper ResNet-style models, **SGD with momentum** can yield better final accuracy. If using SGD, start with a higher LR (0.05–0.1) **but** couple it with the LR schedule (e.g. decay by 0.5 or 0.1 periodically). This mirrors the strategy in ResNet (SGD 0.1 with step-wise decay). For example, **SGD (LR=0.05, momentum=0.9)** with `SCHEDULE_MODE` on (patience ~3, factor 0.5) would train quickly then refine the minima. We should avoid high LR **without** a schedule – e.g. **do not use LR=0.01 static** (the prior runs showed that 0.01 without decay led to unstable convergence [9]). In summary, either stick to a modest constant LR (Adam ~0.003) or use an aggressive initial LR with a decay policy (especially for SGD).

- **Increase Model Depth/Complexity (judiciously):** Within the 400k parameter cap, we can explore slightly deeper or more complex architectures than those tried. For example, adding an extra convolutional layer or an additional residual block could improve accuracy if regularized properly. The original experiments only went as far as 2 conv blocks (for models 0–5). Research suggests that **increasing network depth tends to improve accuracy** [10] [11], up to a point. *"The model capability is increased when the network goes deeper"* [12], as noted by VGG authors. ResNets in particular were able to reach very high accuracy on CIFAR-10 by scaling to 20, 32, 44, or 56 layers [13]. Our current ResNet-inspired model (m5) is quite shallow (only ~4 convolutional layers in the main path). We haven't yet tried a deeper variant due to parameter limits. In section 3 below, we propose a new ResNet-style architecture that adds more residual layers while staying under 400k parameters. This should capture more complex features and potentially boost accuracy further.

- **Combine Residual and Depthwise Techniques:** One thing not yet attempted is **using depthwise separable convolutions within a residual architecture**. Model 4 used depthwise convs (MobileNet-style) but no skip connections; model 5 had skips but standard convs. We could merge these ideas: residual blocks that use depthwise-separable conv layers. Depthwise separable conv is known to *"significantly reduce the number of parameters"* while maintaining good performance [14]. MobileNet demonstrated that such factorized convs can achieve similar accuracy to standard convs at a fraction of the parameter cost [15]. By optionally substituting depthwise convs into a ResNet, we might allow greater depth or width under the same 400k cap. This hybrid approach has not been tried yet in our runs – we address it in the proposed architecture (section 3).

In summary, the key untried improvements are: **turn on augmentation, use LR scheduling, train for many more epochs, apply L2 and dropout (especially on larger models), use batch size ~32, and consider deeper or hybrid (depthwise+residual) architectures.** These changes are all grounded in best practices and research: e.g., *He et al.* emphasize the central importance of network depth [11] and careful LR management, while *Howard et al.* (MobileNet) show depthwise convs can dramatically cut model size with only modest accuracy impact [14]. Incorporating these ideas should materially improve CIFAR-10 test accuracy over the previous baseline.

2. Focused Hyperparameter Sweep Recommendation

Given the above insights, we can **narrow down the configuration search space** to the most promising settings, eliminating ineffective extremes. The goal is to reduce total permutations while retaining combinations that yield high accuracy. Below is a table of recommended configuration values, with pruned values struck out and chosen values in **bold**:

Hyperparameter	Recommended Choices	Pruned Choices	Rationale
Model Architecture	Focus on **m2**, **m3**, **m4** (and new ResNet, see below)	m0, m1 (discarded); possibly m5[*]	m0/m1 (VGG-style) overfit or diverged (e.g. 66% test acc for m0) [16]. m2, m3, m4 showed best generalization (up to ~77–78% acc) [17]. m5 (2-block ResNet) did OK (~74% val) but didn't outperform m3/m4 with such shallow depth. We prioritize the models with consistently higher yield. *We will introduce a new ResNet** architecture to replace m5 (see section 3).
Batch Size	**32** (or 64 if stable)	~~8, 16~~	Batch=32 outperformed 8/16 in all tests [8]. Smaller batches caused noisy grads and lower test accuracy [18]. Batch 32 provides a good trade-off between gradient noise and training stability. (64 could be used if memory allows, but we keep 32 as a proven default.)

Hyperparameter	Recommended Choices	Pruned Choices	Rationale
Data Augmentation	**ON** (flip, crop, color jitter)	~~OFF~~	Augmentation is essential for generalization [1] [2]. Past runs lacked it, leading to overfitting on training data. Always enable standard augments. (We can integrate horizontal flips, random crops (e.g. 32×32 from padded 40×40), and minor color shifts as used in ResNet [3]. Optionally, advanced augments like MixUp or Cutout can be tried in a single run for comparison, but not as separate grid variables to keep experiments manageable.)
Optimizer & LR	**Adam** @ 0.002–0.003, *or* **SGD** @ 0.05 with schedule	Adam @ 0.01 (too high), SGD w/o schedule	Adam with ~0.003 LR has shown strong results (faster convergence, ~77–78% acc range) [8]. We drop Adam 0.01 since it led to unstable training and overshooting [9]. For SGD, we **must** use a schedule if LR $\geq$0.01. A good choice: SGD momentum=0.9, LR=0.05 initial, with LR decay on plateau (or step-down every ~20 epochs). This yields similar or better final accuracy than Adam, though it may converge slower early on. In summary, **either** (Adam 0.003) or (SGD 0.05 + scheduler) can be used, but we do not need to try both for every model – pick the optimizer that prior runs indicated was best per model to avoid duplicate runs. (Earlier results suggest Adam was robust for smaller models, whereas a scheduled SGD might shine for the new deeper ResNet.)
L2 Regularization	**On** (λ = 5e-4) for m3, m4, new ResNet; **Off** for m2	(test both on/off for every model)	We prune the exhaustive on/off combos. Small model m2 was hurt by over-regularization [18] [19], so we **disable L2 for m2** to let it learn freely. For the larger models (m3, m4, and the new ResNet), enable L2 with $\lambda \approx$5e-4 (the value used in VGG/ResNet [5]) to combat their higher capacity. This one decision (per model size) replaces needing to try four combinations (no reg, L2 only, dropout only, both) for each model.

Hyperparameter	Recommended Choices	Pruned Choices	Rationale
Dropout	**On** (rate = 0.3–0.5) for m3, m4, new ResNet; **Off** for m2	(test both on/off for every model)	Similarly to L2, we skip testing dropout in isolation for each model. Based on prior tuning, **dropout + L2 together** yielded the best test accuracy on mid-sized nets (m3, m4) [20] [21]. We will use dropout (e.g. 0.5 in final dense or 0.3 after GAP) in those models alongside L2. For the smallest model m2, we keep dropout off – with only ~15k parameters, it needs all its units working to fit the data (indeed, dropout on m2 showed degraded accuracy in earlier trials).
Learning Rate Schedule	**ON** (for runs with higher initial LR, e.g. SGD; optional for Adam)	~~OFF for high LR~~	We won't waste runs with high constant LRs that don't decay. **Enable scheduling for SGD runs** (and even for Adam runs it can help fine-tune to a slightly higher peak accuracy). We can use `ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3)` or a step schedule (e.g. drop LR by 10× at epoch 50 and 75 if using fixed epochs). The schedule ensures we avoid the training plateau observed when using LR 0.01 without decay [22] [19]. If using Adam 0.002–0.003, scheduling is less critical, but it still can be kept ON to gently reduce LR toward end of training for a final performance polish.
Early Stopping	**ON** (patience ~10, monitor val_loss)	(optional off)	Early stopping doesn't improve the peak accuracy directly, but we enable it to prune runs that stop making progress. Given we plan up to ~100 epochs, an early stop (patience 10) will abort once val loss/accuracy has not improved for 10 epochs, saving time. This helps in an automated multi-run pipeline to reallocate effort to other configs.

Table: Recommended hyperparameter settings for the narrowed sweep. Ineffective options are struck out. Each model variant will use a single best configuration rather than exhaustively cross-testing every toggle.

Using the above, we can define a **reduced set of experiments**: - **For Model 2 (m2 – "ResNet-inspired small CNN")**: Use batch 32, **Adam (0.003)**, **no L2, no dropout**, augmentation **ON**, schedule optional (could be off or a mild plateau reduction since LR is already low). This model has only ~12k–15k params, so we let it learn freely. We expect ~75–77% accuracy from m2 based on earlier ~76.4% best [23], but now with augmentation it may improve. - **For Model 3 (m3 – "ResNet-lite CNN")**: Batch 32, **Adam (0.003)** (or SGD 0.05 + schedule; Adam was effective in past runs for m3), **L2=5e-4 ON**, **Dropout=0.5 ON**, augmentation **ON**, schedule **ON**. This mid-sized model (~67k params) benefits from the extra regularization (to curb overfitting)

and from augmentation. We expect it to reach higher than its previous ~77.6% test accuracy [24] with these improvements (possibly ~80%+). - **For Model 4 (m4 – "Depthwise-separable CNN")**: Batch 32, **Adam (0.002)** (slightly lower LR if we observed any Adam instability, otherwise 0.003 is fine), **L2=5e-4 ON**, **Dropout=0.5 ON**, augmentation **ON**, schedule **ON**. Model 4 has the fewest parameters (~10k) but surprisingly achieved ~78% in prior tuning [25], indicating it generalizes well. Regularizing it further (L2+dropout) and training longer with aug should maintain or improve its performance. Because it's already quite constrained, we might use a slightly smaller dropout (say 0.3) to avoid underfitting, but this can be adjusted. - **(Optionally) Model 5 (current 2-block ResNet)**: We could apply the same settings as m3 to m5 (since it's a similar scale ~69k params). However, since we plan to introduce a new ResNet architecture, we might skip extensive tuning of m5. If one does run m5, use **SGD 0.05 + schedule** (ResNet tends to benefit from SGD), batch 32, L2+dropout on, aug on. But the new model in section 3 should supersede m5.

By focusing on the above and not varying batch size or trying no-aug or extreme LRs, we **drastically prune the search space**. Instead of testing 2^5 combinations for each model (as an initial exhaustive grid might have), we now test essentially **one configuration per model** (the best-guess config) with perhaps one alternative optimizer if warranted. This reduces total runs from dozens to ~3–5. We lock in known good practices (batch=32, aug=ON, schedule=ON for high LR, etc.) and avoid known bad ones (batch=8, high LR w/o decay).

If resources allow a few more runs, one can do a **brief grid around the recommended point** – for example, try Adam vs SGD for the new ResNet model, or dropout 0.3 vs 0.5 on m3 – but *do not* re-introduce the previously ruled-out extremes. The idea is to iterate intelligently: use prior results to inform the next configs rather than brute-forcing all combinations. Our recommended sweep ensures each experiment has a strong chance of success and that we cover complementary strategies (Adam vs SGD, etc.) without redundant or futile trials.

Finally, all these configurations should be integrated into the JSON config files and pipeline. We would create new config files (e.g. `m2_best.json`, `m3_best.json`, `m4_best.json`, etc.) reflecting the above settings. Then the pipeline list might look like:

```
pipeline = [
    (2, "m2_best"),
    (3, "m3_best"),
    (4, "m4_best"),
    (6, "resnet_new")  # (model 6 to be our new ResNet architecture, described
next)
]
```

This small set of runs – each with robust hyperparameters – should converge to high accuracy much faster than the original sprawling search.

3. Proposed ResNet-Style Architecture (≤400k Params)

To maximize accuracy under 400k parameters, we propose a **custom ResNet-style CNN** that incorporates all the lessons learned. This architecture is deeper than the existing model 5, includes residual connections

and batch normalization, uses global average pooling, and can *optionally* use depthwise separable convolutions to save parameters. We target roughly ~**370k parameters** with the standard conv configuration – safely under the 400k limit – and far fewer if depthwise convs are enabled.

Key Architecture Features:

- **Residual Blocks:** The network is organized into a series of residual blocks, which help train a deeper network without vanishing gradients [11]. Each block has two 3×3 convolution layers (as in ResNet basic blocks [13]) and an identity skip connection (with a 1×1 conv *shortcut* when the output dimension changes). These skip connections allow us to go significantly deeper (20+ layers) while maintaining trainability [26]. The residual structure lets the model learn refinements on top of an identity mapping, which eases optimization [27] [28].

- **Batch Normalization & ReLU:** Every convolution is followed by BatchNorm and ReLU activation (we use the *post-activation* architecture as in original ResNet [29] [30]). This normalizes layer inputs, accelerating convergence and improving stability, especially in deeper networks. (If we wanted to be very precise, we could use the **pre-activation** ResNet design from He et al. (2016), but post-activation is simpler and sufficient here.)

- **Global Average Pooling:** Instead of any dense hidden layers, a GlobalAveragePooling2D layer after the final conv block pools each feature map to a single value. This dramatically reduces parameters while preserving performance [31] [32]. Both ResNet and MobileNet use global pooling before the classifier. Removing large fully-connected layers was already done in our models 2–5 and is retained here to stay under 400k params and improve generalization (no risk of overfitting a giant dense layer).

- **Depthwise-Separable Convs (Optional):** We design the conv layers so that we can toggle between standard conv and depthwise separable conv. When `use_depthwise=False`, each conv is a regular Conv2D; when `True`, we replace each 3×3 conv with a DepthwiseConv2D + 1×1 Pointwise Conv (as in MobileNet). This substitution maintains the receptive field but reduces parameters by ~8-9× for each such layer [33] [34]. Depthwise separable convs *"reduce the number of parameters and computation… while increasing representational efficiency"* [15]. By incorporating this option, the same architecture can either be a standard small ResNet or a parameter-extreme-efficient version. For instance, the all-conv variant (use_depthwise=False) will have ~370k params, whereas the depthwise version might have only ~60–80k (we estimate) – useful if one wants an ultra-light model with somewhat lower accuracy.

Architecture Specification:

We base the design on ResNet-20/32 (for CIFAR-10) but adjusted for our parameter budget: - Input: 32×32×3 images. *(No initial max-pooling – CIFAR is small, so we keep full spatial resolution initially, as done in ResNet CIFAR variant [13].)* - **Conv1:** 3×3 conv with **16 filters**, BN, ReLU. (This is the initial convolution layer.) Output: 32×32×16. - **Group 1:** Residual block group with output channels = 16. We use **4 residual blocks** (each block = two 3×3 conv layers + skip connection). The first block in group1 uses a skip connection with no change (identity skip) since input and output are both 16-channel 32×32. Each conv in this group has 16 filters. Subsequent blocks in group1 also keep 16 channels. *If using depthwise, each conv is Depthwise3×3 + Pointwise (1×1) to 16 filters.* Output after group1: 32×32×16. - **Group 2:** Residual block group with output

channels = 32. This group begins by **downsampling** (spatial size 32→16) and doubling filters 16→32. In the first block of group2, the first conv is applied with stride 2, and we use a 1×1 conv shortcut with stride 2 to project the 16×32×16 input to 16×16×32 so shapes match for addition [27] [35]. Each conv in group2 has 32 filters. We again use 4 blocks in this group (the first block does the downsample; the remaining 3 blocks in group2 have stride=1 and identity skip connections). Output after group2: 16×16×32. - **Group 3:** Residual block group with output channels = 64. Similar to group2, the first block here downsamples from 16×16 to 8×8 and increases filters 32→64 (stride 2 conv + 1×1 skip conv). Then 3 more blocks with 64 filters, stride 1. Output after group3: 8×8×64. - **Global Pooling:** 8×8 global average pool yields a 64-D vector. - **Output Layer:** Dense layer with 10 units (softmax), one for each CIFAR-10 class.

This architecture has: **Conv1 + (4+4+4)*2 conv layers = 1 + 24 = 25 conv layers**, plus 3 shortcut convs (at group transitions) for a total of 28 conv layers. It's deeper than any model we've tried so far (compare: previous model5 had 4 conv layers + 2 shortcut convs). Yet, thanks to the small filter counts, the total parameters stay low. Let's verify the parameter count:

- *Conv1 (3×3, 3→16)*: 3×3×3×16 + 16 biases = **448** params (or slightly fewer if no bias before BN).
- *Group1 conv layers:* Each residual block has two 3×3 convs of 16→16. Each such conv: 16×3×3×16 + 16 = 2320 params [6]. Two per block = 4640. Four blocks = **18,560** in conv weights. (BatchNorm adds a negligible 64 trainable params per conv, ~768 total BN here.) There are no shortcut convs in group1 (input=output dimensions).
- *Group2 conv layers:* First block: conv1 16→32 (3×3) = 16×3×3×32+32 = 4640, conv2 32→32 = 32×3×3×32+32 = 9248. Shortcut 1×1 (16→32) = 16*32+32 = 544. So block1 ≈ 4640+9248+544 ≈ 14,432. Each of the next 3 blocks: 2 convs of 32→32 = 2×9248 = 18,496 each. Three blocks = 55,488. Total group2 conv+shortcut ≈ 69,920* params (plus BN ~256 per conv).
- *Group3 conv layers:* First block: conv1 32→64 = 32×3×3×64+64 = 18,496, conv2 64→64 = 64×3×3×64+64 = 36,928. Shortcut 1×1 (32→64) = 32*64+64 = 2112. Block1 ≈ 57,536. Next 3 blocks: 2 convs of 64→64 = 2×36,928 = 73,856 each. Three blocks = 221,568. Total group3 conv+shortcut ≈ 279,104* params.
- *Output Dense:* 64→10 weights = 64*10 + 10 = 650* params.

Summing these rough counts: 448 + 18,560 + 69,920 + 279,104 + 650 ≈ **368,682** trainable parameters (plus ~<3k from all BN layers). This is about 0.37M, under the 400k limit. It is essentially a **"ResNet-26" (26 layers) for CIFAR-10**. For comparison, the original ResNet-32 had ~0.46M params [13], so our design is slightly smaller in depth and uses fewer base filters (16 vs 16–>32–>64, which is standard). We could even afford a *27th layer* (one more block) and still be near ~420k, but we stay safe at 26 layers.

If we switch to **depthwise separable convs** (`use_depthwise=True`), the parameter count drops dramatically. In each conv block, a 3×3 depthwise conv has `(kernel_height*kernel_width*in_channels)` weights (and typically one bias per in_channel), and the following pointwise 1×1 has `(in_channels*out_channels)` weights (+ bias per out_channel). For example, in group3 with 64 filters: a standard conv has 36k weights, whereas a depthwise (3×3 on 64 channels) has 576 weights and the 1×1 has 4096, totaling ~4.7k [36] [37] – about **87% fewer**. Across the whole network, using depthwise convs should reduce total params roughly by a factor of 8 (rough estimate from earlier model4 vs model5). So the depthwise version of this ResNet-26 may have on the order of ~50k. That gives us huge headroom to potentially increase filters if desired; e.g. we could double the filters (32–64–128) and still stay under 400k with depthwise. However, initially we can keep the same 16–32–64

structure for simplicity, which will yield a very lightweight model. One can experiment with width if needed: the flexibility is there to trade param count for accuracy.

Below is a high-level **Keras Functional API** implementation of this architecture. We use a `use_depthwise` flag to toggle the conv type. This code would go into `model.py` (as, say, `model_number == 6`), and we'd add a corresponding config for it:

```
from keras.api.models import Model
from keras.api.layers import Input, Conv2D, DepthwiseConv2D, BatchNormalization,
Activation, Add, GlobalAveragePooling2D, Dense

def build_resnet_cifar(use_depthwise=False):
    """ResNet-style model with optional depthwise separable convolutions."""
    inputs = Input(shape=(32, 32, 3))
    # Initial conv: 3x3, 16 filters
    if use_depthwise:
        # Depthwise + pointwise to get 16 filters
        x = DepthwiseConv2D((3,3), padding='same', strides=1)(inputs)
        x = Conv2D(16, (1,1), padding='same')(x)
    else:
        x = Conv2D(16, (3,3), padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    # Define a residual block function
    def res_block(x, filters, downsample=False):
        """Residual block: two conv layers + skip connection."""
        stride = 2 if downsample else 1
        # Save input as shortcut
        shortcut = x
        # First conv layer (with stride possibly 2)
        if use_depthwise:
            x = DepthwiseConv2D((3,3), padding='same', strides=stride)(x)
            x = Conv2D(filters, (1,1), padding='same')(x)
        else:
            x = Conv2D(filters, (3,3), padding='same', strides=stride)(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        # Second conv layer
        if use_depthwise:
            x = DepthwiseConv2D((3,3), padding='same')(x)
            x = Conv2D(filters, (1,1), padding='same')(x)
        else:
            x = Conv2D(filters, (3,3), padding='same')(x)
        x = BatchNormalization()(x)
        # Process shortcut connection
        if downsample:
```

```
            # If spatial or filter dim changes, use 1x1 conv to match shapes
            if use_depthwise:
                # Depthwise+pointwise for shortcut (could also just do a
pointwise conv with stride)
                shortcut = DepthwiseConv2D((3,3), padding='same', strides=2)
(shortcut)
                shortcut = Conv2D(filters, (1,1), padding='same')(shortcut)
            else:
                shortcut = Conv2D(filters, (1,1), padding='same', strides=2)
(shortcut)
            shortcut = BatchNormalization()(shortcut)
        # Add shortcut
        x = Add()([x, shortcut])
        x = Activation('relu')(x)
        return x
    # Group 1: 4 blocks, filters=16
    for i in range(4):
        x = res_block(x, filters=16, downsample=False)
    # Group 2: 4 blocks, filters=32 (downsample at first block)
    x = res_block(x, filters=32, downsample=True)
    for i in range(3):
        x = res_block(x, filters=32, downsample=False)
    # Group 3: 4 blocks, filters=64 (downsample at first block)
    x = res_block(x, filters=64, downsample=True)
    for i in range(3):
        x = res_block(x, filters=64, downsample=False)
    # Global average pool and output
    x = GlobalAveragePooling2D()(x)
    if config.DROPOUT_MODE.get('enabled', False):  # integrate with existing
config
        x = Dropout(config.DROPOUT_MODE['rate'])(x)
    outputs = Dense(10, activation='softmax')(x)
    return Model(inputs, outputs)
```

The above is a conceptual implementation. In practice, you would integrate it into `build_model` in `model.py` as `elif model_number == 6: ...` using the same patterns (and using the `config` object for dropout as shown, and for weight decay in the Conv2D layers via `kernel_regularizer=reg`). The `reg = l2(lambda) if enabled else None` from earlier in `build_model` can be reused so that L2 regularization is applied to conv kernels. For depthwise conv layers, Keras allows a `depthwise_regularizer` as well – we should apply L2 there too if using depthwise (to be thorough). So, inside `res_block`, we'd do `DepthwiseConv2D(..., padding='same', strides=stride, depthwise_regularizer=reg)(x)` and `Conv2D(..., kernel_regularizer=reg)(x)` for example, honoring the config's L2 lambda.

Parameter count: As calculated, this model (with `filters=[16,32,64]` and 4 blocks per group) has ~369k params (without depthwise). If needed, we could increase the depth (blocks per group) or width (filters) slightly and still be under 400k: - Adding one more residual block per group (making it 5 blocks each,

like ResNet-32) would raise params to ~460k (too high). - Increasing filters to (32, 64, 128) with 3 blocks each would be ~ a larger jump (likely well over 400k). Thus, the chosen 26-layer configuration is a sweet spot near the limit. It should be substantially more accurate than the old 8-layer model5. For reference, ResNet-20 (which is slightly shallower than ours) achieves ~91–92% on CIFAR-10 with full training [13]. With augmentation and proper training, our 26-layer network should aim for ~**<8% error (92%+ accuracy)** as well. Hitting >**82%** (the goal mentioned in the tuning plan) is very realistic with this architecture and training strategy.

Integration into Training Pipeline: To use this model in our pipeline, do the following: 1. **Implement model 6 in** `model.py`**:** As illustrated above. Make sure to incorporate the `config` flags: - Use `config.L2_MODE` to set `reg` and apply to conv layers (both Conv2D and DepthwiseConv2D). - Use `config.DROPOUT_MODE` to conditionally add Dropout (already shown above). - Possibly add a `config.USE_DEPTHWISE` boolean to toggle `use_depthwise`. Alternatively, we could decide that *model 6 = standard conv ResNet* and define *model 7 = depthwise ResNet* for clarity, rather than a runtime flag. But a flag is flexible. For example, the config file could have `"DEPTHWISE_MODE": {"enabled": true}` and in `build_model` if model_number==6 and `config.DEPTHWISE_MODE['enabled']` is true, call `build_resnet_cifar(use_depthwise=True)`. This approach integrates nicely with the existing config system. - Ensure naming of layers doesn't clash (Keras will autoname them uniquely since it's a new model graph). 2. **Add a config file** for this model, e.g., `resnet26_base.json`, analogous to m5_base but tailored. For example:

```
// artifact/config/resnet26_base.json
{
  "LIGHT_MODE": false,
  "AUGMENT_MODE": true,
  "L2_MODE": { "enabled": true, "lambda": 0.0005 },
  "DROPOUT_MODE": { "enabled": true, "rate": 0.3 },
  "OPTIMIZER": { "type": "sgd", "learning_rate": 0.05, "momentum": 0.9 },
  "SCHEDULE_MODE": { "enabled": true, "monitor": "val_loss", "factor": 0.5,
"patience": 3 },
  "EARLY_STOP_MODE": { "enabled": true, "monitor": "val_loss", "patience": 10 },
  "EPOCHS_COUNT": 100,
  "BATCH_SIZE": 32,
  "DEPTHWISE_MODE": { "enabled": false }
}
```

This config turns on augmentation, L2, dropout, uses SGD with an LR schedule, etc., as per our recommendations. (If we want to try the depthwise version, we could copy this to `resnet26_depthwise.json` and just set `"DEPTHWISE_MODE": { "enabled": true }`.) 3. **Update** `experiment.py` **pipeline** to include `(6, "resnet26_base")` (and maybe `(6, "resnet26_depthwise")` for comparison). The pipeline will then build and train model number 6 with those settings. Our `train.py` and logging should handle it just like other models. We should double-check that `dispatch_load_dataset` (in `data.py`) is general or keyed by model number if doing any model-specific preprocessing – likely it just loads CIFAR-10 normally for any model. 4. **Run training** and monitor logs. The training will likely take longer per epoch than previous smaller models, but still very feasible (ResNet-26 with batch 32 on CIFAR-10 is not heavy). We should see the val accuracy climbing well

beyond previous ~78%. If training starts to overfit (train acc >> val acc), consider lowering dropout rate a bit or using early stopping to catch the peak. The LR schedule will automatically reduce learning rate when val loss plateaus, helping squeeze out extra accuracy.

In summary, this ResNet-26 architecture is a highly optimized model that should significantly improve CIFAR-10 performance while respecting the 400k parameter budget. It integrates the **residual learning** concept from ResNet (enabling much deeper network than earlier attempts) [13], uses **batch norm** and **global pooling** throughout (as per modern best practices), and offers a **depthwise conv option** inspired by MobileNet to further optimize the accuracy-vs-size trade-off [14]. By adding this model to our pipeline with the tuned hyperparameters from part 2 (batch 32, augmentation, L2, dropout, scheduled SGD), we expect to push the test accuracy to new highs (well into the 80-90% range). The training pipeline and config system can easily accommodate this addition – we just treat it as another `model_number` with its config. With these changes, our CIFAR-10 experiments will be both **comprehensive and efficient**, focusing only on the most promising models and training schemes moving forward.

Sources:

- Simonyan & Zisserman (VGG-16), *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015 [38] [5] (use of dropout and weight decay, depth vs performance)
- He et al. (ResNet), *Deep Residual Learning for Image Recognition*, 2015 [3] [11] (augmentation strategy, learning rate schedule, importance of depth, residual connections)
- Howard et al. (MobileNet), *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision*, 2017 [2] [14] (augmentation, depthwise separable convolution efficiency)
- **Prior experiment logs and results** (see `tuning-03` insights) [8] [9] for empirical findings on batch size, optimizer, and regularization effects in our project.

---

[1] [2] [3] [4] t001_gpt-references-01.pdf
file://file-5WCesCEjG5wu898ZAfxX1P

[5] [10] [12] [38] arxiv.org
https://arxiv.org/pdf/1508.01667

[6] [7] t001_gpt-tuning-01.pdf
file://file-DVBEzWnNcjWVrbKy5ZG6Nu

[8] [9] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] t001_gpt-tuning-03.txt
file://file-4dw36cd3HFTfsfTkSNb5C1

[11] [1512.03385] Deep Residual Learning for Image Recognition
https://meilu1.jpshuntong.com/url-68747470733a2f2f61727869762e6f7267/abs/1512.03385

[13] Review of Deep Residual Learning for Image Recognition - Medium
https://medium.com/deepreview/review-of-deep-residual-learning-for-image-recognition-a92955acf3aa

[14] Depthwise separable convolutions explained - Kaggle
https://www.kaggle.com/code/mehanat96/depthwise-separable-convolutions-explained

[15] Depthwise Separable Convolutions for Neural Machine Translation
https://arxiv.org/abs/1706.03059

26 [1512.03385] Deep Residual Learning for Image Recognition - arXiv
https://arxiv.org/abs/1512.03385

27 28 29 30 31 32 33 34 35 36 37 model.py
file://file-VcmuWgsyQeWbHQWy1CLPcF