ChatGPT

Replicating ResNet-20 for 91%+ CIFAR-10 Accuracy (Google Colab Guide)

Comparing Current Implementation vs. ResNet Paper

Model Architecture: The repository's `model.py` defines several architectures (models 6, 7, 8) with increasing use of residual connections. Model 8 in particular is essentially a ResNet-20 architecture for CIFAR-10: an initial 3×3 conv layer with 16 filters, followed by 3 stages of **residual blocks** (each block has two 3×3 conv layers) with filter counts 16, 32, 64. Each stage in model 8 has 3 such blocks (total $6n+2$ conv layers with $n!=!3$, giving 20 layers including the first conv and final FC). The code confirms this structure: model 8 uses an initial Conv(16) + BN/ReLU, then **3 residual blocks at 16 filters, 3 at 32 filters, 3 at 64 filters** with downsampling (stride 2) at the first block of each new stage [1] [2]. This matches the ResNet-20 design described by He et al.. Model 7 is a similar idea but starts with 20 filters and uses dropout, deviating from the paper's specs. Model 6 diverges further (starting at 32 filters and using only 2 blocks per stage). **Conclusion:** *Model 8 most closely follows the ResNet paper's CIFAR-10 architecture and will be our base to replicate 91%+ accuracy.*

Residual Block Details: In the current implementation, each residual block (model 7/8) does `Conv-BN-ReLU -> Conv-BN -> Add(shortcut) -> ReLU` [3] [4]. This is consistent with the original ResNet v1 **post-activation** design: batch normalization and ReLU are applied **after** each convolution (and the second BN is followed by addition then ReLU) [5]. The code correctly places **BatchNorm right after conv and before activation** in every block [5]. No changes needed here – this matches the paper ("BN [is] right after each convolution and before activation" [5]).

Shortcut Connections: The ResNet paper uses **identity shortcuts** "in all cases" for CIFAR-10 (referred to as "option A") – meaning no additional parameters are used for the skip connection. When the feature map size or number of filters changes (e.g. 16→32 filters), they simply perform identity mapping with **zero-padding** for extra channels and use stride 2 on the identity to downsample [6] [7]. In our code, however, model 8 uses a 1×1 conv projection for these cases (`if downsample or channel mismatch: shortcut = Conv2D(…, strides=2)`), i.e. **option B** with a learnable conv to match dimensions [8]. This is a slight discrepancy. To **strictly follow the paper**, we would use identity skips with zero-padding for downsampling (no conv layer). However, using 1×1 conv shortcuts is a common practice and usually **improves or maintains accuracy**, so it's acceptable for reproduction (it adds a few parameters but helps matching tensor shapes cleanly).

Batch Normalization and Activation: As noted, the implementation already does BN then ReLU after each conv. The initial conv layer in model 8 is also followed by BN and ReLU [9], which is fine (the paper doesn't explicitly mention BN on the first layer, but applying it is consistent with modern practice). We should ensure we use **He initialization** for conv layers (Kaiming He et al.'s initializer), as the paper did ("weights initialized as in [13]" – He's 2015 init) [5]. In Keras, this means using `kernel_initializer="he_normal"` for conv layers (the current code doesn't specify initializer, defaulting to Glorot; we can override to he_normal for fidelity).

Dropout and Regularization: The ResNet paper **does not use dropout** for CIFAR-10 ("we do not use dropout" [10]), relying on batch norm and weight decay for regularization. Our code's model 8 already has dropout **disabled** (see `m8_base.json`: `"DROPOUT_MODE": {"enabled": false}`) [11]. Weight decay (L2 regularization) is crucial: the paper used **weight decay λ=1e-4** [10]. In `m8_base.json`, L2 is enabled with λ=0.0001 [12], which matches 1e-4. We will keep that. (Models 6/7 used 5e-4, but model 8 corrects this to 1e-4.)

Data Preprocessing: The dataset is CIFAR-10 (50k train, 10k test). The ResNet paper mentions subtracting the "per-pixel mean" from images. In practice, this means they likely subtracted the training set mean image (which is equivalent to subtracting mean per channel for CIFAR-10, since the mean pixel values are nearly constant per channel). Our code computes the standard per-channel mean = [0.4914, 0.4822, 0.4465] and std = [0.2023, 0.1994, 0.2010] for CIFAR-10 [13]. **Model 8's pipeline** uses these to standardize images: if augmentation is on, it applies random crop/flip then normalizes by this mean/std via `transforms.Normalize` [14]; if augmentation is off, it calls `_standardize` to do the same normalization after scaling to [0,1] [15]. The test set is always standardized using the same mean and std [16]. This is correct and should be retained – ensuring inputs are normalized (mean 0, unit variance per channel) is important.

Data Augmentation: To replicate the original training setup, we must use **the same data augmentation** as they did. He et al. "intentionally use simple… [architectures]" for CIFAR and focus on depth, but they did use standard CIFAR-10 augmentation (they didn't explicitly detail it, but nearly all CIFAR training uses horizontal flips and random crops). Our code's augmentation for model 8 is already appropriate: it does **RandomCrop(32, padding=4)** and **RandomHorizontalFlip()** on the fly [17], which is exactly the usual CIFAR-10 augmentation. (Models 0–7 used an `_augment_dataset` that also added slight color jitter [18], but model 8's augmentation omits color jitter, sticking to the basics – this aligns with the paper's "simple" augmentation approach.) We will use: **pad by 4 pixels, random 32×32 crop, random flip**. No other fancy augmentations are needed.

Summary of Discrepancies & Resolutions: In short, to strictly follow the ResNet paper for CIFAR-10 we should use the model 8 architecture (ResNet-20) with **16-32-64 filters** and 3 blocks per stage, ensure **BN+ReLU ordering** is correct (it is), **disable dropout** [10], and use **L2 weight decay 1e-4**. The only architectural choice to consider changing is the shortcut type: the paper's identity shortcuts vs. our conv shortcuts. It's a minor difference; using conv shortcuts (as in model 8) will not prevent achieving 91%+, but we note this for completeness. Everything else (global average pooling before the final dense, etc.) already matches the ResNet design.

Training Setup as per ResNet Paper

Optimizer: Use **SGD with momentum 0.9** [10]. The config for model 8 already specifies SGD with 0.9 momentum and an initial learning rate of 0.1 [19]. This is exactly what we need: He et al. started with LR=0.1 for CIFAR experiments (with a warmup tweak for extremely deep nets) [20]. No Adam or other optimizers – stick to momentum SGD.

Learning Rate Schedule: The paper's approach was to start at 0.1 and **reduce the learning rate by 10× when the error plateaued** [10]. In practice, for CIFAR-10 ResNet, this typically translates to dropping the LR at specific epochs. A common schedule (from ResNet-20 implementations) is: **0.1 for ~100 epochs, then**

0.01 for ~50 epochs, then 0.001 for ~50 epochs, total ~200 epochs. This schedule closely approximates the "plateau" rule (the model usually plateaus around 100th epoch, then again later). We will use 200 epochs and drop LR at **epoch 100 and 150** (if using Keras, you can manually adjust or use a callback). The code's `SCHEDULE_MODE` was turned **off** in m8 config [21] (they didn't use the ReduceLROnPlateau for model 8). Instead of adaptive plateau detection, we'll implement the **fixed step schedule** as above to mirror the original method. In Colab, one can use `tf.keras.callbacks.LearningRateScheduler` or simply track epoch and adjust optimizer LR.

Epochs and Batch Size: He et al. trained up to **60×10^4 iterations** for CIFAR (which is effectively up to ~150-200 epochs for batch size 128) [10]. We will train for **200 epochs** on Colab. The batch size in the paper was 128 or 256 (for ImageNet they used 256; for CIFAR-10 a batch of 128 is common) [10]. Our config uses **128** by default for model 8 [22], which is fine. On Google Colab (with GPU), a batch of 128 for CIFAR-10 (32×32 images) is easily handled. If memory or time is a concern, batch 128 is a good trade-off. *(Note:* If using a TPU or a high-memory GPU, batch 256 could be tried, but it's not necessary).

Weight Decay and Regularization: We reiterate the importance of weight decay (L2=1e-4) on all conv layers [10]. In Keras, this is done via `kernel_regularizer=l2(1e-4)` as seen in the code [23] [24]. Model 8 already applies this regularizer to conv and dense layers (through the `regularizer` variable) [1] [25]. We must ensure this stays in place for training – it combats overfitting and was key to the original results. No dropout (we keep it off). BatchNorm itself provides regularization benefits, so between BN + weight decay + augmentation, the network should generalize well.

Early Stopping: The original paper did **not use early stopping** – they trained for a fixed number of iterations and took the final/ best model. In our Colab run, we should **not enable early stopping** (to allow full 200 epochs and LR drops). The config for model 8 has early stopping disabled [26], which is good. We will simply monitor validation accuracy manually or use a callback to save the best model. The code's training routine saves the best model to a checkpoint automatically [27], so we can rely on that to pick the best epoch after training completes.

Expected Accuracy: According to He et al., ResNet-20 achieves about **8.75% test error** (≈91.3% accuracy). ResNet-56 can reach ~93% accuracy, but ResNet-20 is enough to cross 91%. With our setup, we should aim for **91-92% test accuracy** after 200 epochs. (Slight variance is normal, but if the implementation is correct, >91% is achievable.)

Step-by-Step Colab Guide

Following the above, here is a step-by-step plan to train ResNet-20 on CIFAR-10 in Google Colab:

1. **Environment Setup:** Start a Colab notebook and enable GPU acceleration (`Runtime > Change runtime type > GPU`). Install necessary libraries. Our code uses PyTorch's torchvision for data and TensorFlow/Keras for modeling, so ensure both frameworks are available. Colab comes with TensorFlow and torchvision pre-installed. You may need to `%pip install tensorflow` (if not already 2.x) and `%pip install torchvision`. Also import necessary modules: e.g. `import tensorflow as tf; from tensorflow import keras; import torchvision.datasets as datasets; import torchvision.transforms as transforms; import numpy as np`.

2. **Load and Preprocess CIFAR-10:** In Colab, use torchvision to download CIFAR-10. For example:

```
train_set = datasets.CIFAR10(root="data", train=True, download=True)
test_set = datasets.CIFAR10(root="data", train=False, download=True)
X_train = train_set.data; y_train = np.array(train_set.targets)
X_test = test_set.data; y_test = np.array(test_set.targets)
```

   Now apply preprocessing:

3. **Data augmentation (train only):** Define a transform pipeline that pads, crops, and flips. Using torchvision transforms:

```
transform_train = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.4914,0.4822,0.4465], std=[0.2023,0.1994,
0.2010])
])
```

   This matches the augmentation in our code [28] (random 4px crop and horizontal flip, then normalization). We'll apply this to each image in X_train. (On Colab, you can either convert the entire dataset to a PyTorch TensorDataset or simply apply the transform on the fly within a custom data generator. For simplicity, one can do a Python loop to transform all images into a new NumPy array.)
4. **Normalization (test data):** Convert X_test to float and normalize with the same per-channel mean/std [16]. In code:

```
X_test = X_test.astype('float32')/255.0
X_test[...,0] = (X_test[...,0] - 0.4914)/0.2023
X_test[...,1] = (X_test[...,1] - 0.4822)/0.1994
X_test[...,2] = (X_test[...,2] - 0.4465)/0.2010
```

   (Or use `transforms.Normalize` similarly on test images.)

5. **Train/Val split:** Optionally set aside a validation subset from the training data. The original paper trained on all 50k and evaluated on the 10k test. If we use all 50k for training, we can monitor accuracy on the test set periodically (just be mindful not to overfit to it manually). Alternatively, use 45k for train and 5k for validation (our code reserves 5k as val by default [29]). For faithful reproduction, training on the full 50k is fine since we have fixed epochs and weight decay (no early stopping). We can proceed without a separate val – just use test set for final accuracy.

6. **Define the ResNet-20 Model:** We'll implement the model in Keras. Leverage the design from `model.py` model 8:

7. **Input:** 32×32×3.
8. **Initial conv:** 3×3 conv with 16 filters, stride 1, "same" padding, He-normal init, L2=1e-4. Follow with BN and ReLU.
9. **Stage 1 (32×32 feature maps):** 3 residual blocks with 16 filters. Each block: conv3×3→BN→ReLU→conv3×3→BN→(add input)→ReLU. *All convs in this stage use 16 filters and stride 1.* The first block's shortcut is just identity (no downsample needed, input and output are same shape).
10. **Stage 2 (16×16 feature maps):** 3 blocks with 32 filters. The **first block in this stage** will **downsample**: use stride=2 in the convs, and use a shortcut that also downsamples. To follow option A (identity shortcut) strictly, we would do: shortcut = average-pooling or strided identity + zero-pad channels. But a simpler approach is using a 1×1 conv with stride 2 to match shape (as the current code does [30] [8]). We can implement the shortcut as `Conv2D(32,(1,1), strides=2, kernel_regularizer=l2(1e-4), kernel_initializer='he_normal')` for that first block. Subsequent blocks in stage 2 use 32 filters, stride 1, and identity shortcut (no conv).
11. **Stage 3 (8×8 feature maps):** 3 blocks with 64 filters. The first block has stride=2 in convs and a 1×1 conv shortcut with stride 2 to increase to 64 filters. The next two blocks have 64 filters, stride 1, identity shortcuts.
12. **Global Average Pooling:** after stage 3, apply `GlobalAveragePooling2D()`.
13. **Output layer:** Dense(10, activation=softmax, kernel_regularizer=l2(1e-4)). (No activation before this because we want raw logits into softmax.)

14. We need to wrap these layers in a `keras.Model(inputs, outputs)`. Ensure each conv uses `padding="same", kernel_regularizer=l2(1e-4), kernel_initializer='he_normal'`. Every conv (including the shortcut convs) should be followed by BN. For example, in code form, a residual block function might look like:

```
def res_block(x, filters, downsample=False):
    stride = 2 if downsample else 1
    x_short = x  # preserve shortcut
    # Conv 1
    x = keras.layers.Conv2D(filters, (3,3), strides=stride, padding='same',
                            kernel_initializer='he_normal',
kernel_regularizer=keras.regularizers.l2(1e-4))(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Activation('relu')(x)
    # Conv 2
    x = keras.layers.Conv2D(filters, (3,3), padding='same',
                            kernel_initializer='he_normal',
kernel_regularizer=keras.regularizers.l2(1e-4))(x)
    x = keras.layers.BatchNormalization()(x)
    # Shortcut handling
    if downsample or x_short.shape[-1] != filters:
        # 1x1 conv for shortcut to match shape
        x_short = keras.layers.Conv2D(filters, (1,1), strides=stride,
padding='same',
                                      kernel_initializer='he_normal',
kernel_regularizer=keras.regularizers.l2(1e-4))(x_short)
```

```
        x_short = keras.layers.BatchNormalization()(x_short)
    # Add & ReLU
    x = keras.layers.Add()([x, x_short])
    x = keras.layers.Activation('relu')(x)
    return x
```

Using this, we build the network:

```
inputs = keras.Input(shape=(32,32,3))
x = keras.layers.Conv2D(16, (3,3), padding='same',
kernel_initializer='he_normal',
                        kernel_regularizer=keras.regularizers.l2(1e-4))
(inputs)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
# Stage 1: 3 blocks of 16 filters
for _ in range(3):
    x = res_block(x, 16, downsample=False)
# Stage 2: 3 blocks of 32 filters (first block downsamples)
x = res_block(x, 32, downsample=True)
x = res_block(x, 32, downsample=False)
x = res_block(x, 32, downsample=False)
# Stage 3: 3 blocks of 64 filters (first block downsamples)
x = res_block(x, 64, downsample=True)
x = res_block(x, 64, downsample=False)
x = res_block(x, 64, downsample=False)
# Global pool and classifier
x = keras.layers.GlobalAveragePooling2D()(x)
outputs = keras.layers.Dense(10, activation='softmax',
kernel_initializer='he_normal',

kernel_regularizer=keras.regularizers.l2(1e-4))(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

This model is equivalent to the repository's model 8 (ResNet-20) [1] [31]. You can verify the layer count: `model.summary()` should show ~270k parameters and 20 layers (conv layers + final dense).

15. **Compile the Model:** Use **SGD optimizer** with momentum. In Keras:

```
opt = keras.optimizers.SGD(learning_rate=0.1, momentum=0.9, nesterov=False)
model.compile(optimizer=opt, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

(Use `sparse_categorical_crossentropy` since labels are integers 0-9.) The initial LR is 0.1 as required [10]. We'll adjust it later via schedule.

16. **Prepare Callbacks for Learning Rate Schedule and Model Saving:** We want to drop LR at 100 and 150 epochs. In Colab, define a schedule function or use `keras.callbacks.ReduceLROnPlateau` if you prefer an automated approach (though fixed schedule is closer to paper). A straightforward method is:

```
def lr_schedule(epoch, lr):
    if epoch == 100:
        return 0.01
    if epoch == 150:
        return 0.001
    return lr
lr_callback = keras.callbacks.LearningRateScheduler(lr_schedule)
```

Also prepare a **ModelCheckpoint** to save the best model (monitor validation or test accuracy). For example:

```
checkpoint = keras.callbacks.ModelCheckpoint("resnet20_best.h5",
monitor='val_accuracy',
                                             save_best_only=True,
verbose=1)
```

(If not using a validation split, you can monitor `accuracy` on training or directly `val_accuracy` on the test set by passing `validation_data=(X_test, y_test)` to `model.fit`.)

17. **Train the Model:** Now call `model.fit` on the training data. Use `.flow` from a Keras ImageDataGenerator if you want augmentation on the fly, or since we already applied transforms to the training array, we can just feed the processed array. For example:

```
model.fit(X_train_processed, y_train, batch_size=128, epochs=200,
          validation_data=(X_test_processed, y_test),
          callbacks=[lr_callback, checkpoint])
```

(Make sure `X_train_processed` is the augmented+standardized training data. Alternatively, use `ImageDataGenerator` with preprocessing function to do random crop/flip each epoch.) Training 200 epochs on a GPU will take some time (~1 hour or two). Keep an eye on the output. You should see training accuracy rapidly climbing and validation (test) accuracy improving epoch by epoch. Around epoch 100, accuracy will plateau in the high 80s%–low 90s%, then when our LR drops to 0.01, the model will fine-tune and gain a couple more points, and similarly after epoch 150 when dropping to 0.001.

18. **Evaluation:** After training completes, load the best saved model (`resnet20_best.h5`) and evaluate on the test set:

```
best_model = keras.models.load_model("resnet20_best.h5")
loss, acc = best_model.evaluate(X_test_processed, y_test, batch_size=128)
print(f"Test accuracy: {acc:.4f}")
```

We expect **accuracy ≥ 0.91 (91%)**. In the original paper, ResNet-20 got ~91.25%. Don't worry if it's, say, 90.5% or 92% – small fluctuations can happen, but 91%+ is typically reachable. If slightly below 91%, one can try training a bit longer (e.g. 220 epochs), but usually 200 is enough if everything is implemented right.

19. **(Optional: Try ResNet-56)**: If you want to replicate the higher accuracy of ResNet-56 (~93%), you can extend the above model: set `n=9` blocks per stage (so each stage has 9 blocks instead of 3). That means 18 conv layers per stage + 1 initial conv + 1 final dense = 56 layers. Adjust the code loops to 9 iterations for each stage. You'll have ~0.85M parameters and need to train perhaps a bit longer (the paper used the same 60k iterations, which is ~200 epochs, to train ResNet-56). On Colab, ResNet-56 training will take longer and use more memory, so you might reduce batch size to 64 if needed. But this is only if you're curious – our main goal ResNet-20 is lighter and already above 91% accuracy.

By following this plan – using the **correct ResNet-20 architecture**, proper **data normalization and augmentation**, **SGD with momentum and weight decay**, and the **original learning rate schedule** – you will closely reproduce He et al.'s results. This rigorously matches the ResNet paper's setup: *3×3 conv layers with identity-based residuals, BN after each conv, no dropout, data standardized, and training for 200 epochs with decreasing learning rate* [10] . You should achieve **CIFAR-10 test accuracy in the 91-92% range**, confirming the implementation is aligned with the seminal ResNet methodology. Good luck with your Colab experiment!

Sources:

- K. He et al., *"Deep Residual Learning for Image Recognition,"* 2015 – original ResNet paper (architecture details and CIFAR-10 results) [10] .
- *User's code repository (*`model.py`, `data.py`*)* – ResNet-like model implementations and training configs [1] [14] , which we aligned to the paper's descriptions.

---

[1] [2] [3] [4] [8] [9] [23] [24] [25] [30] [31] model.py
file://file-4kkjfQJmWwTkDzdmmbN4mN

[5] [6] [7] [10] [20] arxiv.org
https://arxiv.org/pdf/1512.03385

[11] [12] [19] [21] [22] [26] m8_base.json
file://file-AcajyVftX4YsiVfBHq38G7

[13] [14] [15] [16] [17] [18] [28] data.py
file://file-HiEcLRfEebDfNgQZm3nCsa

[27] [29] train.py
file://file-45pf7uEj7eY2VfrirGH2Yb