Modern Data Augmentation Techniques for Computer Vision

A comparison between Cutout, mixup, CutMix, and AugMix augmentations and their impact on model robustness.

Ayush Thakur

Introduction

Today, deep learning advancements are possible due to faster compute power and massive datasets. However, for many real problems, the dataset is hard to come by. The best way to regularize your model or make it more robust is to feed in more data, but how can one get more data?

The most straightforward answer is to collect more data, but that is not always feasible. The easiest way to train a model on a large amount of data is to use data augmentation. **Data augmentation significantly increases the diversity of data available for training our models, without actually collecting new data samples.**

Simple image data augmentation techniques like flipping, random crop, and random rotation are commonly used to train large models. This works well for most of the toy datasets and problem statements. Nevertheless, in reality, there can be a huge data shift. Is our model robust to data shift and data corruption? As it stands, models don't robustly generalize for shifts in data. If models could identify when they are likely to be mistaken, or estimate uncertainty accurately, then the impact of such fragility might be reduced. Unfortunately, the models are overconfident about its prediction.

In this report, we will dive into modern data augmentation techniques for computer vision. Here is a quick outline of what you should expect from this report:

1. Theoretical know-how of some modern data augmentations along with there implementations in TensorFlow 2.x.
2. Some interesting ablation studies.
3. Comparative study between these techniques.
4. Benchmarking of models trained with the augmentations techniques on the CIFAR-10-C dataset.

Try it in Google Colab →

Experimental Setup and Baseline

I followed the following experimental setup for training the models with different augmentation techniques:

Dataset

1. Our models use the CIFAR-10 dataset for training based on different augmentation strategies.
2. We **use the CIFAR-10-C dataset to measure a model's resilience to data shift.**

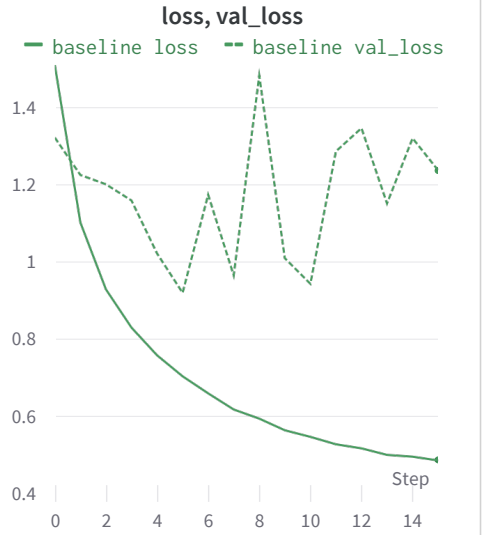
This dataset is constructed by corrupting the original CIFAR test set. Each dataset has 15 noise, blur, weather, and digital corruption types, each appearing at five severity levels. Since these corruptions are used to measure network behavior under data shift, they are not introduced into the training procedure(augmentation). In CIFAR-10-C, the first 10,000 images in each `.npy` files are the test set images corrupted at severity 1, and the last 10,000 images are the test set images corrupted at severity five.

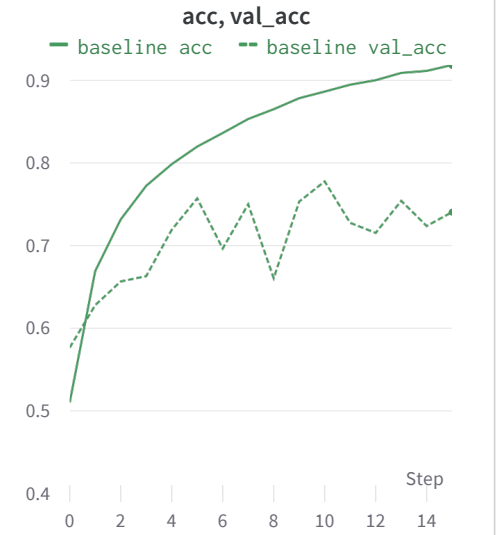
Architecture

1. All the experiments use ResNet20 architecture, which uses `resnet_v1` model builder from official Keras documentation example on CIFAR-10. You can find the model definition in the repo included with this report.
2. No data augmentation apart from the augmentation strategy under experimentation was used.
3. We save the initial weight and train all the models using this `initial_wt.h5` to start from same model weight initialization,
4. The models were trained with Early Stopping with the patience of 10 epochs and monitored `val_loss`. The upper bound for epochs was 100.
5. We save the trained weights and use them for robustness benchmarking. We also save the weights from various ablation studies. You can find all the weights here.

Baseline Model

The baseline model will be used to compare all the augmentation techniques. It is used to do a comparative measure of a model's resilience to data shift/corruption. I trained the model using the configurations, as mentioned above. Below, I show the metrics





Observation

- The model clearly overfitted after few epochs.
- Without any augmentation technique the model could only reach the **best** validation accuracy of 77.74%

Cutout Augmentation

To prevent overfitting, models must be regularized properly by either using data augmentation or with the judicious addition of noise to activations, parameters, data, etc.

One of the most common uses of noise to improve model accuracy is Dropout, which

stochastically(randomly) drops neuron activations during training and discourages the co-adaptation of feature detectors. Dropout tends to work well for fully connected layers but lacks that regularization power for convolutional layers. The authors of Cutout have two explanation for this:

1. Convolutional layers require less regularization since they have much fewer parameters than fully-connected layers.
2. The second factor is that neighboring pixels in images share much of the same information.

Autoencoders are good at learning useful representation from the image in a self-supervised manner. Especially the class of autoencoders like Context Encoders, where the input data is corrupted. The network reconstructs them using the remaining pixels as context to determine how to best fill in those blanks. Such architectures have a better understanding of the global content of the image, and therefore they learn better higher-level features.

Cutout

Improved Regularization of Convolutional Neural Networks with Cutout

Inspired by dropout and context encoder, **Cutout is a simple regularization technique that involves removing contiguous(patch) section from the input image while training.** It can be viewed as the dropout operation but in the input space. This ensures that the model looks at the entire image rather than fixing its attention to some key features. We will see if this make models more robust to data shift. Let us implement it.

Implementation

Full code in Google Colab →

```
trainloader = tf.data.Dataset.from_tensor_slices((x_train, y_train))

## Apply cutout augmentation
def augment_cutout(image, label):
  img = tfa.image.random_cutout(image, (10,10), constant_values = 0)

  return img, label

## Apply image preprocessing
def preprocess_image(image, label):
  img = tf.cast(image, tf.float32)
  img = img/255.

  return img, label

trainloader = (
    trainloader
    .shuffle(1024)
    .map(preprocess_image, num_parallel_calls=AUTO)
    .batch(BATCH_SIZE)
    .map(augment_cutout, num_parallel_calls=AUTO)
    .prefetch(AUTO)
)
```

- I have used `tf.data` to implement this augmentation technique.
- It uses TensorFlow Addon version `0.10.0`.

```
!pip install tensorflow-addons==0.10.0
```

- `augment_cutout` simply wraps `tfa.image.random_cutout` to apply random cutout to the images. Official documentation is here.
- I have used `cutsize` of `(10, 10)` as default. You will soon see the dependence of cutsize with model performance.

The result of this augmentation is shown below.

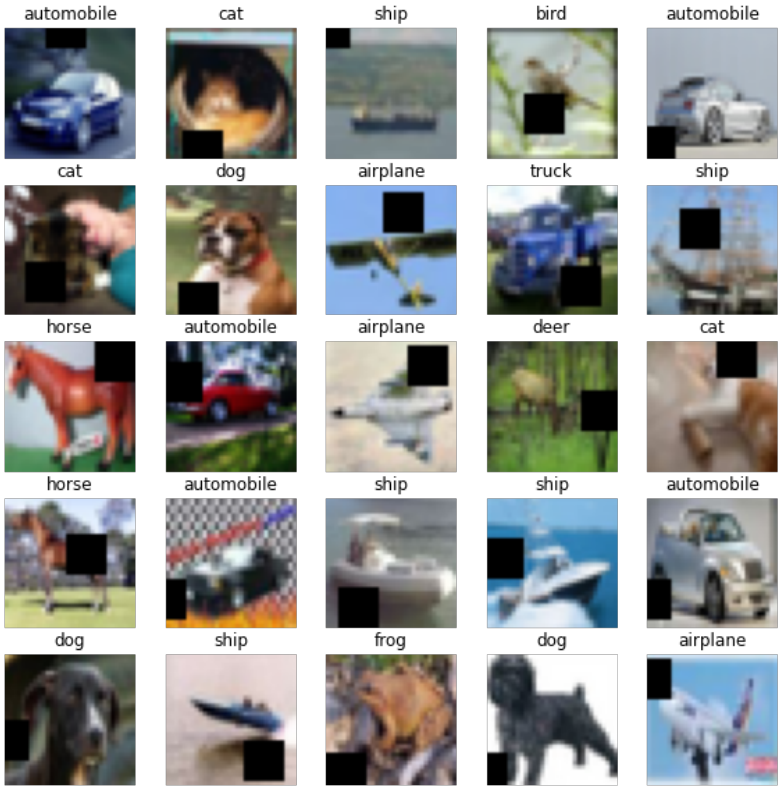
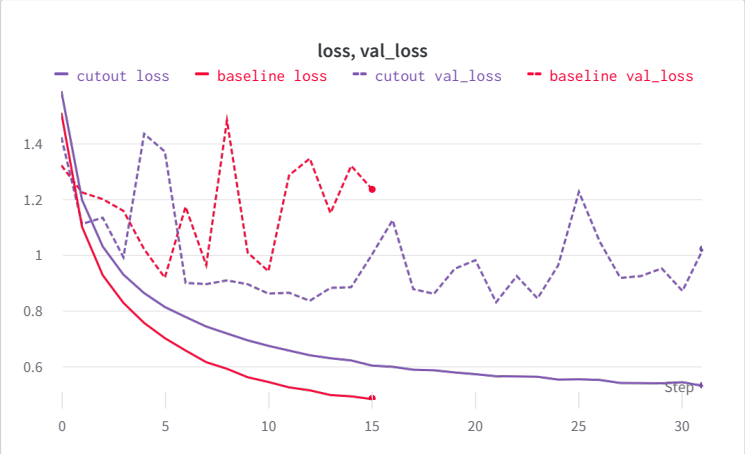


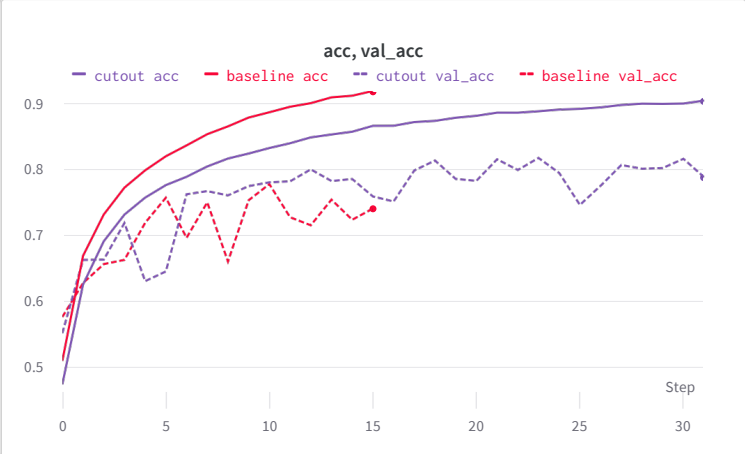
Figure 1: Cutout Augmentation

Comparison with Baseline Model

I trained ResNet-20 with cutout augmentation as per the experimental setup mentioned above. These are the observations.

- Unlike the baseline model where overfitting occurred quickly, **Cutout augmentation provided necessary regularization enabling longer training time.**
- Even though the training accuracy with Cutout augmentation is lower than that of the baseline model, **the validation accuracy is higher, ensuring better generalization over the training dataset.**
- The authors of Cutout used standard augmentations like random cropping along with cutout augmentation. On top of that, they used a deeper network and trained with a highly tuned SGD optimizer.


loss, val_loss
cutout loss
baseline loss
cutout val_loss
baseline val_loss
1.4
1.2
1
0.8
0.6
Step
0
5
10
15
20
25
30


acc, val_acc
cutout acc
baseline acc
cutout val_acc
baseline val_acc
0.9
0.8
0.7
0.6
0.5
Step
0
5
10
15
20
25
30

Cutout 2

Ablation Study

Cutsize Vs Test Accuracy
As per the authors of cutout, the size of the cutout region is a more important hyperparameter than its shape. Thus for the simplicity of implementation square patch is used as a cutout region. The question remains to find out the optimal cutsize for the given dataset.

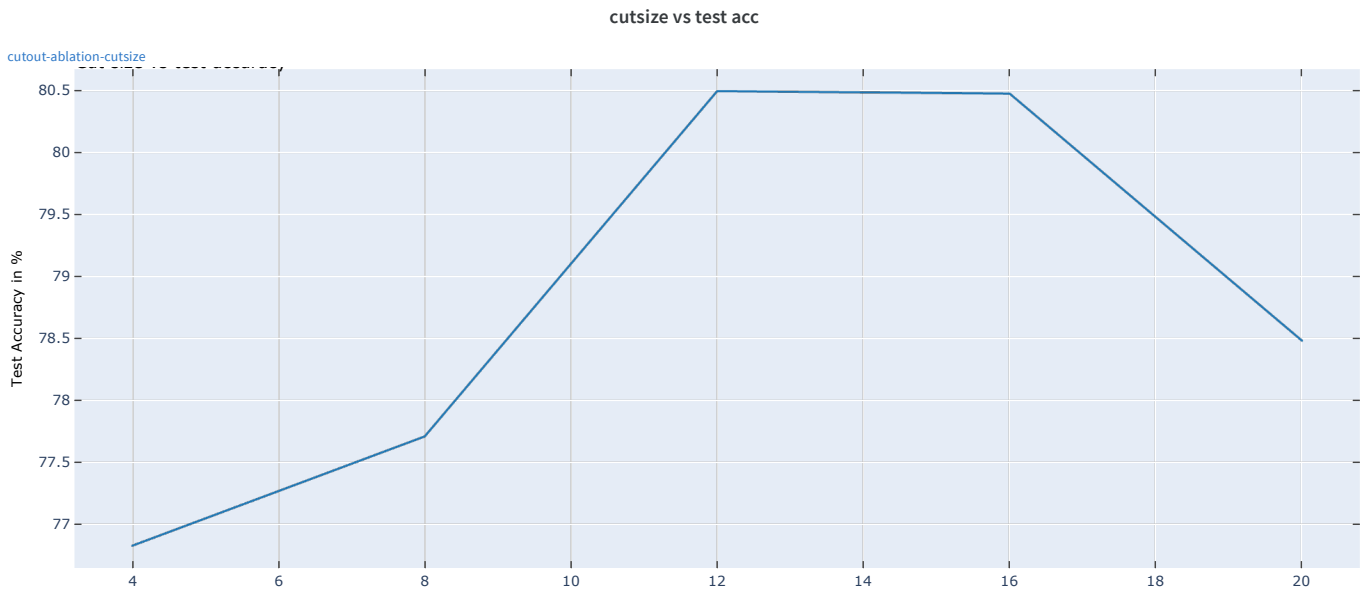
In this line, I performed `cutsize vs test accuracy` ablation study.

- ResNet-20 model was trained with 5 different cutsizes.

```
CUT_SIZES = [4, 8, 12, 16, 20]
```

- They were trained with early-stopping to get the best instance of the model for each cutsize.
- The result of this ablation study is shown in the figure below.

Notice that the cutsize of 12 gave the best performance followed by 16.



Mixup Augmentation

In the new state of the art neural network models, a learning rule minimizes the average error over the training data. At the same time, the number of parameters increases linearly with the increase in the dataset. One observes that large neural networks tend to memorize the training data instead of generalizing from them, even in the presence of strong regularization.

Data Augmentation is a process of training the model with similar but different from the training examples. These similar *virtual* examples can be drawn from the *vicinity* distribution of the training examples to enlarge the support of the training distribution. For example, when performing image classification, it is common to define one image's vicinity as the set of its horizontal reflections, slight rotations, and mild scalings.

However, human knowledge is required to describe a vicinity or neighborhood around each example in the training data.

While data augmentation consistently leads to improved generalization, two issues limit them. They are:

1. The procedure is dataset dependent, and thus requires the use of expert knowledge.
2. Data augmentation assumes that the examples in the vicinity(neighborhood) share the same class, and does not model the vicinity relation across examples of different classes.

mixup

mixup: Beyond Empirical Risk Minimization

To address the problems mentioned, the authors of mixup came up with a data-agnostic(data-independent) and a simple technique to implement a data augmentation strategy. In a nutshell, it creates *virtual* examples by,

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

where, (x_i, y_i) and (x_j, y_j) are two data points randomly drawn from the training dataset and $\lambda \in [0, 1]$. Thus mixup performs linear interpolation in the input space with similar interpolation in the associated target space. This improves model robustness to corrupt labels, avoids overfitting as it is hard to memorize *virtual* labels, and increases generalization.

λ is drawn from $\text{Beta}(\alpha, \alpha)$ distribution. Where α controls the strength of interpolation in the input/target space. More on Beta distribution here.

Implementation

Full code in Google Colab →

```
def mixup(a, b):

  alpha = [0.2]
  beta = [0.2]

  # unpack (image, label) pairs
  (image1, label1), (image2, label2) = a, b

  # define beta distribution
  dist = tfd.Beta(alpha, beta)
  # sample from this distribution
  l = dist.sample(1)[0][0]

  # mixup augmentation
  img = l*image1+(1-l)*image2
  lab = l*label1+(1-l)*label2

  return img, lab


trainloader1 = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(1024).map
trainloader2 = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(1024).map
```

```
trainloader = tf.data.Dataset.zip((trainloader1, trainloader2))
trainloader = (
    trainloader
    .shuffle(1024)
    .map(mixup, num_parallel_calls=AUTO)
    .batch(BATCH_SIZE)
    .prefetch(AUTO)
)
```

- I have used `tf.data` for mixup implementation.
- `mixup` function takes in two (image, label) pairs and applies linear interpolation in image and label space. This function can easily be used for any input modality.
- Notice that λ (l) is sampled from a `Beta` distribution. `TensorFlow Probability` is used for this.
- Since mixup requires two images drawn randomly from the training dataset, `trainloader1` and `trainloader2` are two separate `tf.data` datasets zipped using `tf.data.Dataset.zip`.
- This zipped dataset provides two pairs of (image, label) which using `map` returns augmented image and associated labels.
- The implementation is straight forward and has minimal computational overhead.

The result of this augmentation is shown below. 👇

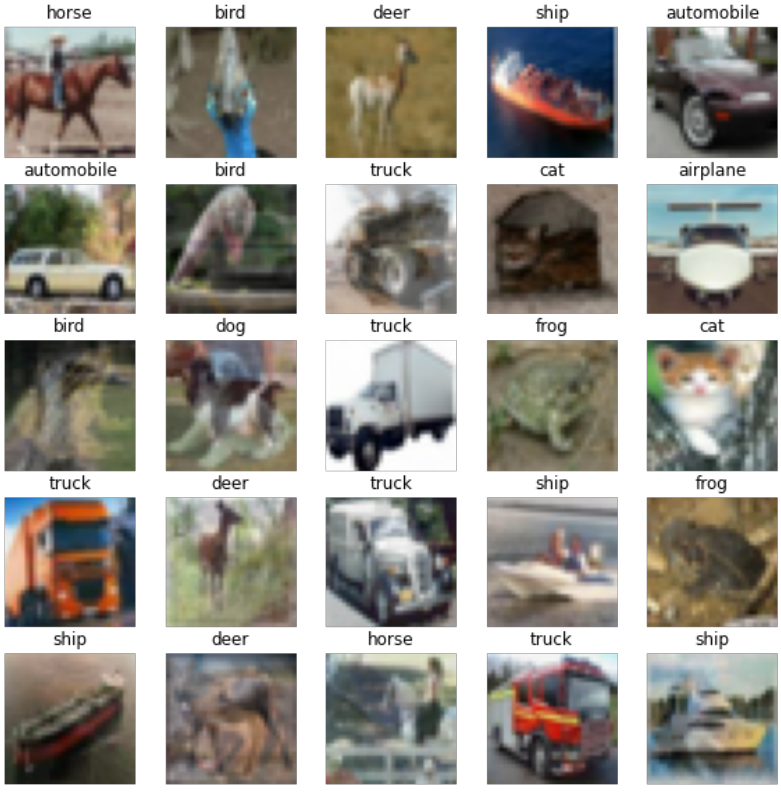


Figure 2: mixup Augmentation

Comparison with Baseline Model

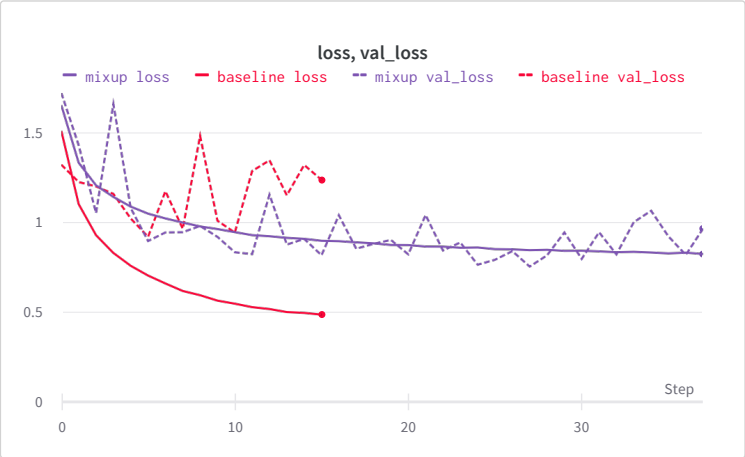
I trained ResNet-20 with mixup augmentation, and the validation loss curve speaks for itself. I used alpha=0.2 instead of alpha=1.0, which was originally used by the authors to train CIFAR-10.

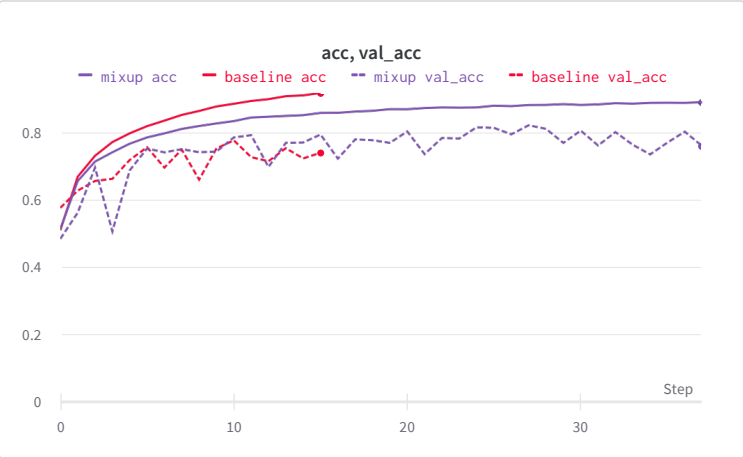
- **The baseline model quickly overfitted.**

- The validation loss followed the training loss without diverging. This allowed for **more extended training and far better generalization.**
- The validation accuracy seems to fluctuate less than the baseline model. **mixup helped stabilize the learning process.**

So you may ask, what is mixup doing?

- **The model is encouraged to act linearly in between classes.** This linear behavior **reduces undesirable oscillations when predicting outside the training examples.** This is visible in the validation curves.
- **mixup provides a smoother(linear) transition from one class decision boundary to another.** This provides a better measure of uncertainty.
- The model trained with mixup is **more stable in terms of model predictions.**





Ablation Study

Alpha Vs Test Accuracy

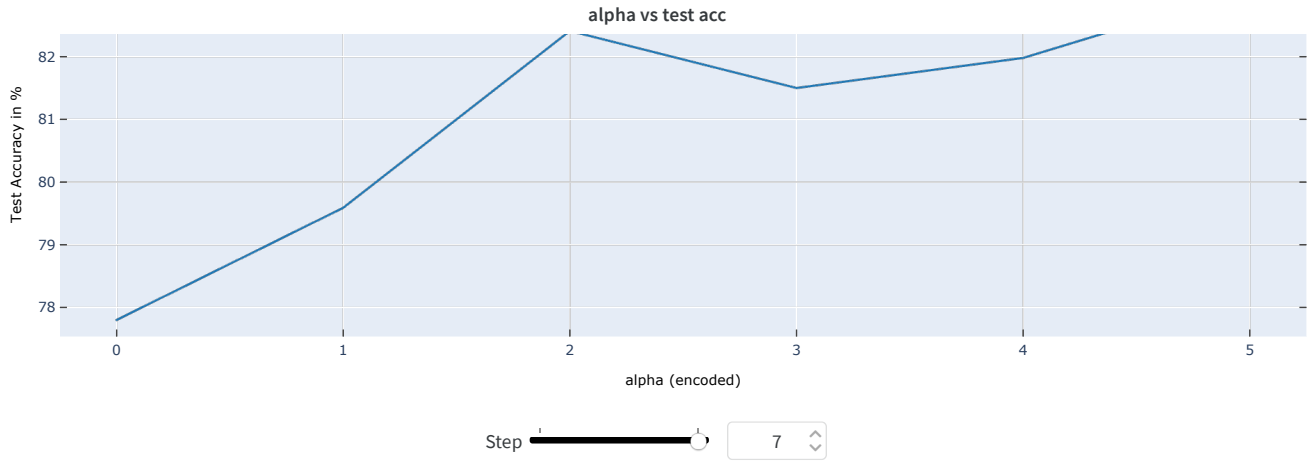
For this particular experiment, the aim was to find the best alpha value to be used for `Beta` distribution while modeling the CIFAR-10 dataset. The process and the results are discussed below:

- ResNet-20 was trained with six different alpha values.

```
ALPHAS = [0.1, 0.2, 0.4, 0.7, 0.9, 1.0]
```

- When `alpha=0.0` beta distribution returns 0.5. As alpha increases from 0 towards 1, the shape of the distribution is convex in nature, thus more values lie either towards zero or one. For `alpha=1.0`, the beta distribution is uniform in nature.
- Along the `x-axis`, x-ticks corresponds to `ALPHAS`.
- The result for this ablation study is shown in the figure below.

Notice that `alpha=1.0` gave the best results followed by `alpha=0.4`.



CutMix Augmentation

Even though the previous augmentations techniques discussed are unique in itself, they have some issues:

1. In Cutout augmentation, a region(square) is removed from the image and is filled with either black or grey pixels or with Gaussian noise. This dramatically decreases the portion of informative pixels during the training process. It is a conceptual limitation as CNN is data-hungry.
2. In Mixup augmentation, we linearly interpolate two randomly drawn images. However, the images generated are somewhat unnatural and confuse the model, especially for the localization task.

CutMix

CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features

The authors of CutMix addressed the issues mentioned above in these ways:

1. Instead of removing pixels and filling them with black or grey pixels or Gaussian noise, what if it is replaced with a patch of similar dimensions from another image.
2. The ground truth labels are mixed proportionally to the number of pixels of combined images. Linearly interpolated label thus produced is proportional to the contributing pixels from the two images.

With this, there is no uninformative pixel during training, making training efficient while retaining the advantages of regional dropout in the input space. Also, the images thus formed are locally more natural compared to mixup.

The virtual examples thus generated is given by,

$$\tilde{x} = M x_i + (1 - M) x_j$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

where M is a binary mask(often square) indicating the Cutout and the fill-in regions from the two randomly drawn images. Just like mixup λ is drawn from $\mathrm{Beta}(\alpha, \alpha)$ distribution and $\lambda \in [0, 1]$.

After the images are randomly selected, bounding box coordinates are sampled such

that $B = (r_x, r_y, r_w, r_h)$ indicates the Cutout and fill-in regions in both the images. The bounding box sampling is given by,

$$r_x \sim U(0,\ W),\ r_w\ =\ W\sqrt{1-\lambda}$$

$$r_y \sim U(0,\ H),\ r_h\ =\ H\sqrt{1-\lambda}$$

where r_x, r_y are randomly drawn from a uniform distribution with upper bound as shown. Let us implement this augmentation strategy. 👇

Implementation

Full code in Google Colab →

```
@tf.function
def get_bbox(l):
  cut_rat = tf.math.sqrt(1.-l)

  cut_w = IMG_SHAPE*cut_rat #rw
  cut_w = tf.cast(cut_w, tf.int32)

  cut_h = IMG_SHAPE*cut_rat #rh
  cut_h = tf.cast(cut_h, tf.int32)

  cx = tf.random.uniform((1,), minval=0, maxval=IMG_SHAPE, dtype=tf.int32) #rx
  cy = tf.random.uniform((1,), minval=0, maxval=IMG_SHAPE, dtype=tf.int32) #ry

  bbx1 = tf.clip_by_value(cx[0] - cut_w // 2, 0, IMG_SHAPE)
  bby1 = tf.clip_by_value(cy[0] - cut_h // 2, 0, IMG_SHAPE)
  bbx2 = tf.clip_by_value(cx[0] + cut_w // 2, 0, IMG_SHAPE)
  bby2 = tf.clip_by_value(cy[0] + cut_h // 2, 0, IMG_SHAPE)

  target_h = bby2-bby1
  if target_h ==0:
    target_h+=1

  target_w = bbx2-bbx1
  if target_w ==0:
    target_w+=1

  return bbx1, bby1, target_h, target_w

@tf.function
def cutmix(a, b):

  (image1, label1), (image2, label2) = a, b

  alpha = [1.]
  beta = [1.]

  ## Get sample from beta distribution
  dist = tfd.Beta(alpha, beta)
  ## Lambda
  l = dist.sample(1)[0][0]

  ## Get bbox ofsets and heights and widths
  bbx1, bby1, target_h, target_w = get_bbox(l)

  ## Get patch from image2
```

```
  crop2 = tf.image.crop_to_bounding_box(image2, bby1, bbx1, target_h, target_w)
  ## Pad the patch with same offset
  image2 = tf.image.pad_to_bounding_box(crop2, bby1, bbx1, IMG_SHAPE, IMG_SHAPE)
  ## Get patch from image1
  crop1 = tf.image.crop_to_bounding_box(image1, bby1, bbx1, target_h, target_w)
  ## Pad the patch with same offset
  img1 = tf.image.pad_to_bounding_box(crop1, bby1, bbx1, IMG_SHAPE, IMG_SHAPE)

  ## Subtract the patch from image1 so that patch from image2 can be put on instead
  image1 = image1-img1
  ## Add modified image1 and image2 to get cutmix image
  image = image1+image2

  ## Adjust lambda according to pixel ratio
  l = 1 - (target_w * target_h) / (IMG_SHAPE * IMG_SHAPE)
  l = tf.cast(l, tf.float32)

  ## Combine labels
  label = l*label1+(1-l)*label2

  return image, label


trainloader1 = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(1024).map
trainloader2 = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(1024).map

trainloader = tf.data.Dataset.zip((trainloader1, trainloader2))
trainloader = (
    trainloader
    .shuffle(1024)
    .map(cutmix, num_parallel_calls=AUTO)
    .batch(BATCH_SIZE)
    .prefetch(AUTO)
)
```

- I have used `tf.data` to implement this augmentation strategy.
- The `get_bbox` function takes in λ(l), which is sampled from a beta distribution and returns bounding box coordinates. It correctly returns `(x, y)` coordinates for the top-left corner of the patch to be removed along with the height and width of the patch.
- `cutmix` function takes in two (image, label) pairs and applies the CutMix augmentation strategy. First, it samples λ(l) from a beta distribution and gets the bounding box coordinates from the `get_bbox` function. Then we get a crop from `image2` and pad it such that the final padded image has this crop at the same spatial location. We also do this for `image1`, but then we use this to subtract the patch from `image1`. Finally, we add `image1` from which we have removed a patch and `image2`, which is the patch itself(after cropping+padding operation). Try plotting the generated images for better understanding.
- We adjust lambda(l) according to the proportion of the pixels contributed by each image. Moreover, we use this to interpolate the labels linearly.
- Since CutMix requires two images drawn randomly from the training dataset, `trainloader1` and `trainloader2` are two separate `tf.data` datasets which are zipped using `tf.data.Dataset.zip`.

- This zipped dataset provides two pairs of (image, label) which using `map` returns augmented image and associated label.

The result of this implementation is shown below.

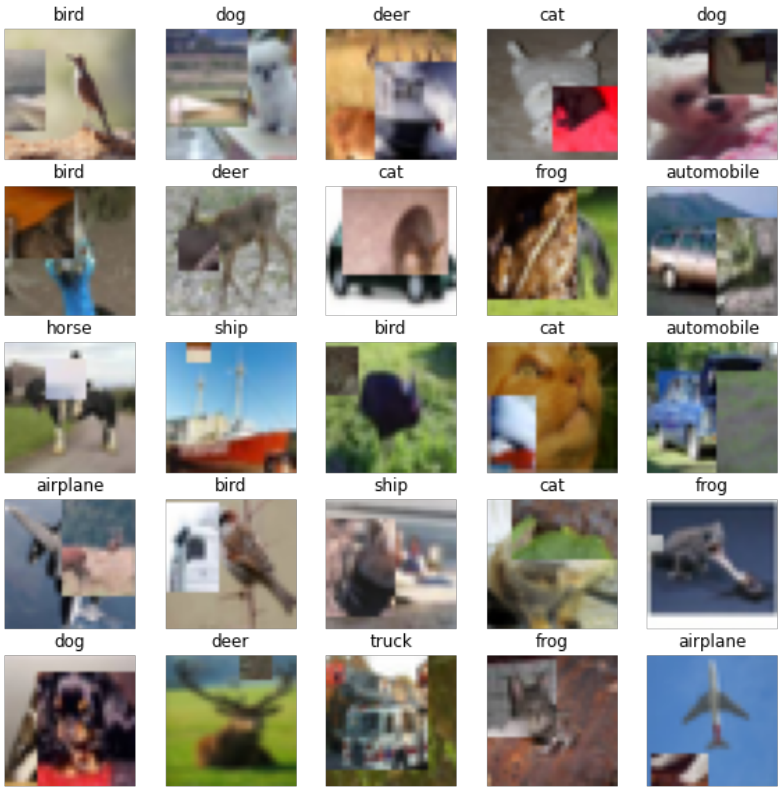
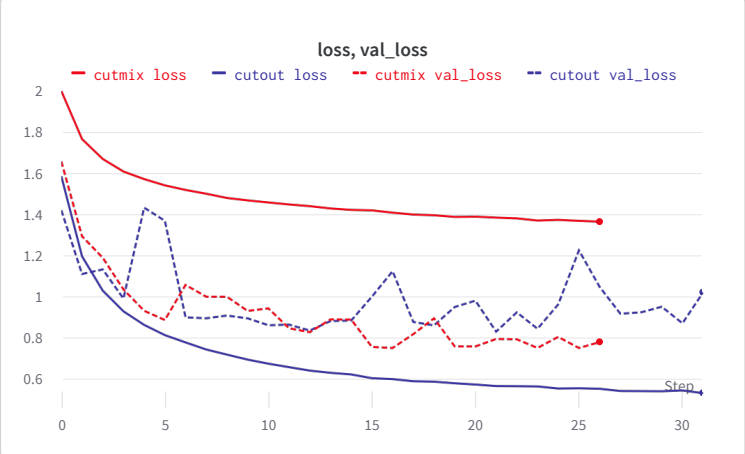


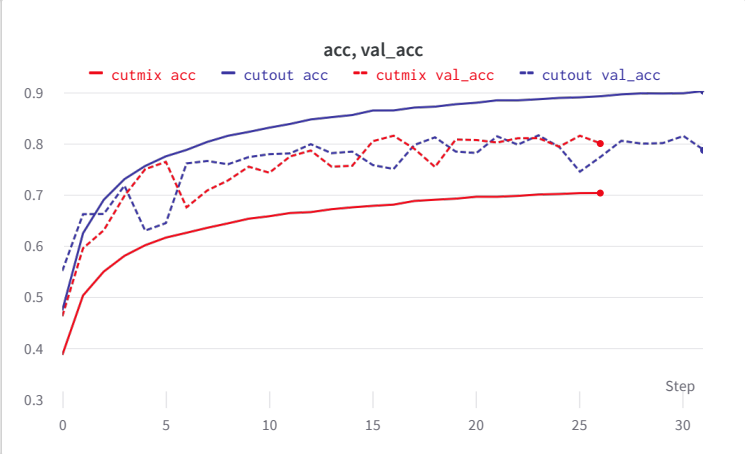
Figure 3: CutMix Augmentation

Comparison with Cutout Augmentation

The model trained with Cutmix augmentation performs better than the baseline model; however, it would be interesting to compare Cutmix with Cutout augmentation.

- **Cutmix provides stronger regularization compared to Cutout.** This can be seen through the loss metrics.
- But this strong regularization is indeed helping with reducing generalization error. However, this reduction in validation error is marginal.
- For this particular comparison, the model was trained with Cutmix with `alpha=1.0`. Through the ablation study, we will find out the optimal value of alpha.
- **The validation metrics are oscillating less when compared to the model trained with cutout augmentation.** The oscillation in the case of Cutout can be due to less informative pixels during training, which is not the case with CutMix.
- Training with better learning rate scheduling can reduce the risk of overfitting due to strong regularization.





Ablation Study

Alpha VS Test Accuracy

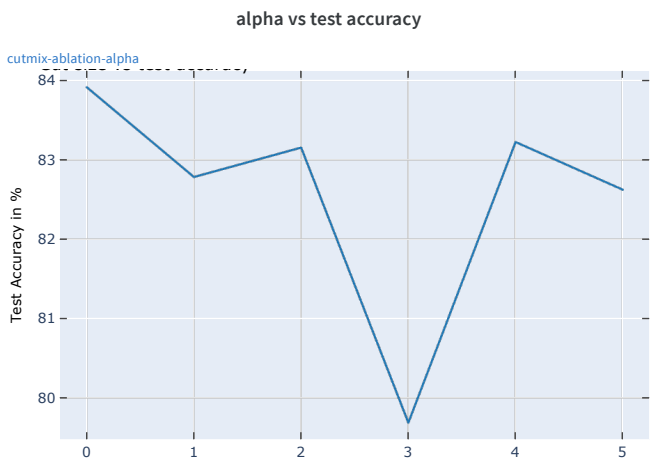
For this particular experiment, the aim was to find the optimal alpha value to be used for `Beta` distribution while modeling the CIFAR-10 dataset. The process and the result is discussed below:

- ResNet-20 was trained with six different alpha values.

```
ALPHAS = [0.1, 0.25, 0.5, 1.0, 2.0, 4.0]
```

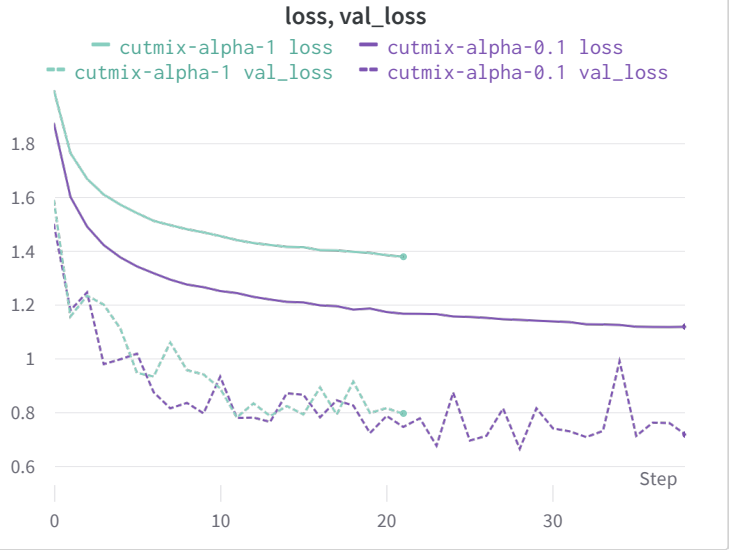
- When `alpha=0.0` beta distribution returns 0.5. As alpha increases from 0 towards 1, the shape of the distribution is convex in nature, thus more values lie either towards zero or one. For `alpha=1.0`, the beta distribution is uniform in nature. For `alpha>1.0` the shape is concave in nature.
- Along the `x-axis`, x-ticks corresponds to `ALPHAS`.
- The result for this ablation study is shown in the figure.

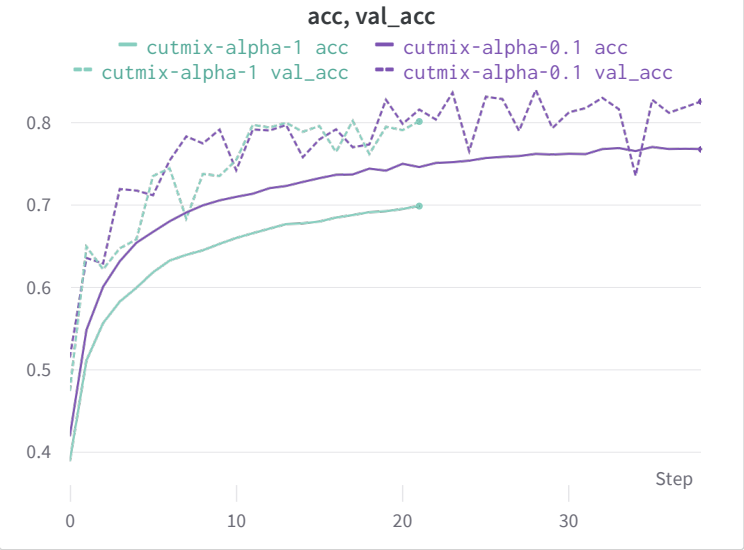
Notice that `alpha=0.1` gave the best results. Test accuracy for `alpha=1.0` gave the worst result. The comparison shown above uses `alpha=1.0`. It would be interesting to have a look at the metrics.



Observations:

- The model validation accuracy on the CIFAR-10 test dataset improved by a margin of 4.22%.
- The model trained a lot longer while the model fitted better on training data.





AugMix Augmentation

Overfitting is caused when models memorize to the statistical noise in the training dataset, thus resulting in less robust models. Data Augmentation techniques provide needed regularization, but what is stopping those large models from memorizing the *fixed* augmentation techniques? Mixing augmentations prevent this and allow us to generate diverse transformations.

Most methods simply chain one augmentation technique after another, but this can cause the image to quickly degrade and drift off the training set manifold. We show the effect of chaining augmentations below.

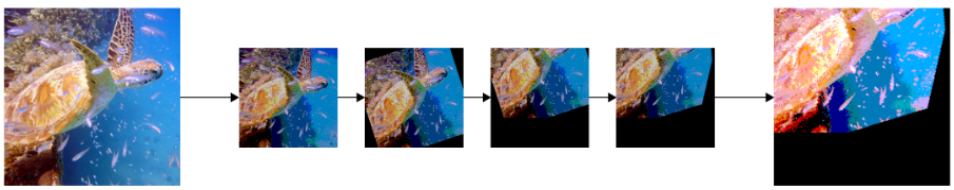


Figure 4: Image degradation by chaining augmentation techniques (source).

AugMix

AugMix: A Simple Data Processing Method to Improve Robustness and Uncertainty

AugMix came up with a data augmentation technique that improves model robustness and slots easily existing training pipelines. AugMix prevents degradation of images while maintaining diversity as a result of mixing the results of augmentation techniques in a convex combination (mixing images in mixup augmentation is a convex combination as well.) At a high level, it is characterized by its utilization of simple augmentation operations in concert with a consistency loss. We will briefly go through each element of AugMix.

- **Augmentations:** This method consists of mixing the results from augmentation chains or compositions of augmentation operations. These operations are sampled randomly from a pool of simple augmentation techniques. For operations which can be realized with different level of severity, the level is uniformly sampled. AugMix provides augmentation diversity by having multiple chains of augmentation with varying depths. By default, the number of chains is three, while its depth is uniformly sampled in a range of (1,3). Some augmentation techniques used are:

```
[autocontrast, equalize, posterize, rotate, solarize, shear_x, shear_y, tra
```

- **Mixing:** The images are mixed using the element-wise convex combination for simplicity. After images from different chains are mixed, it is combined with the original image through a second random convex combination sampled from a Beta distribution.
- **Jensen-Shannon Divergence Consistency Loss:** This loss ensures smoother neural network response. Since the semantic content of the image is, to a lot extent, preserved with AugMix, the network should embed $x_{original}$, $x_{augmix1}$, and $x_{augmix2}$ similarly. Thus the original loss is modified by adding this loss. Mathematically,

$$L(p_{orignal},\ y)\ + \lambda JS(p_{original};\ p_{augmix1};\ p_{augmix2})$$

$$(original\ loss)\ +\ (Jensen-Shannon\ divergence)$$

Here, $p_{original}$, $p_{augmix1}$, $p_{augmix2}$ are the output logits and JS is the consistency loss given by,

$$JS(p_{original};p_{augmix1};p_{augmix2})\ = \frac{1}{3}(KL[p_{original}||M]\ + KL[p_{augmix1}||M]\ + KL[p_{augmix2}||M])$$

Where M is the average of all three logits.

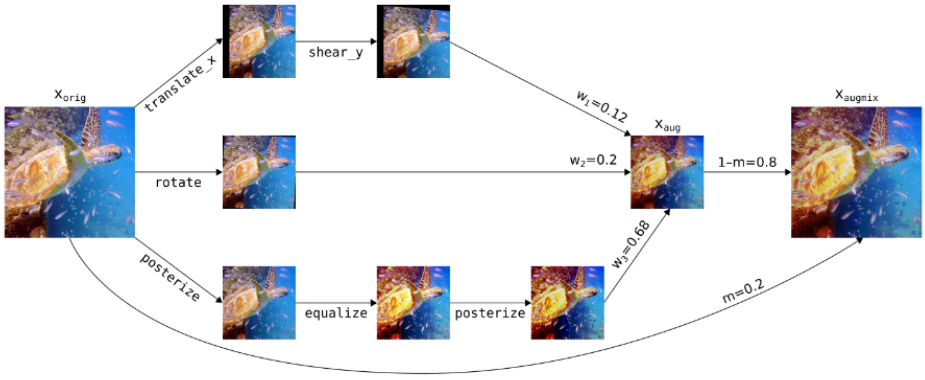


Figure 5: AugMix with three chains in action (source).

Implementation

For this augmentation technique I didn't reinvent the wheel. I found a TensorFlow 2.x implementation of AugMix by Aakash Nain. Check out his implementation here. I

forked the same and integrated weights and biases so that I can log the metrics and do necessary comparative study. I also added some command line arguments for more control. You can find the forked repo here. The result of this implementation is shown below. 👇

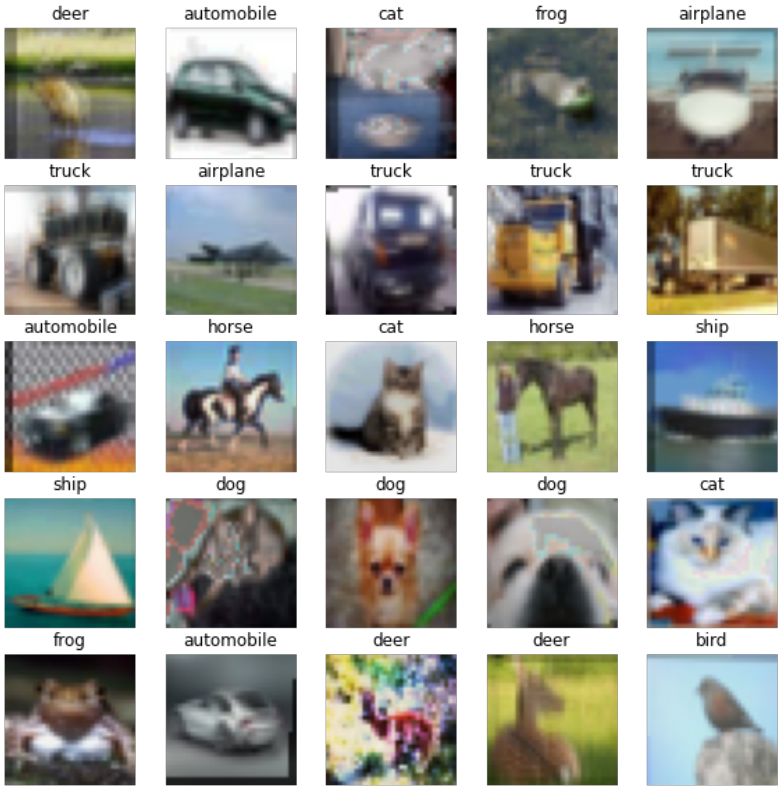


Figure 6: AugMix Augmentation

Comparison with other augmentation techniques

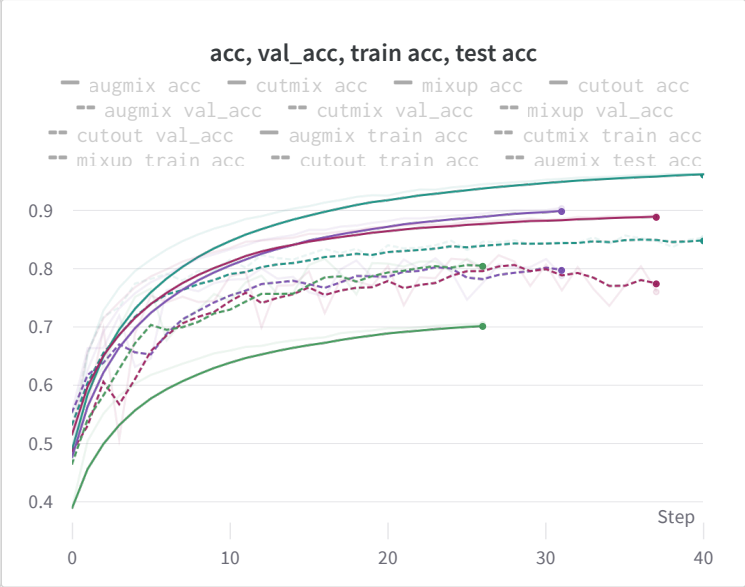
Now that we have reached the end of our augmentation exploration, it would be interesting to compare all the augmentation strategies explored so far. The idea is to look at the metrics for each strategy and draw some conclusions. Note that the models selected for this comparison are trained with either default `cutsize` or `alpha`. Our evaluation study has a particular emphasis on model robustness to data shift and corruption. Some conclusions are:

- By looking at the validation loss for each augmentation strategy, the model performance can be ranked like this.

```
augmix > cutmix > mixup > cutout > baseline
```

- The baseline model trained on CIFAR-10 quickly overfitted. Each augmentation strategy is a better addition on top of the baseline model.
- CutMix regularized the strongest. We see that mixup was not so aggressive. AugMix initially regularized more strongly but slowly smoothed out. Cutout barely regularized but was a better than baseline.

(Note: The metrics are smoothed out by a factor of 0.5 for better visualization.)



Model Robustness to CIFAR-10-C

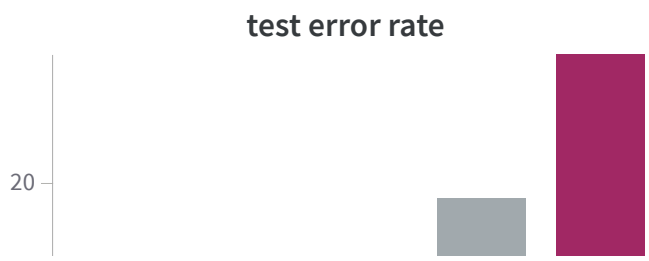
To test the model's resilience to data shift, let's evaluate on CIFAR-10-C dataset. Full code in Google Colab →

- Originally CIFAR-10-C has 19 corruption types. But we will be evaluating on 15 such corruptions as was originally done in the AugMix paper. These corruptions are:

```
CORRUPTIONS = [
  'gaussian_noise', 'shot_noise', 'impulse_noise', 'defocus_blur',
  'glass_blur', 'motion_blur', 'zoom_blur', 'snow', 'frost', 'fog',
  'brightness', 'contrast', 'elastic_transform', 'pixelate',
  'jpeg_compression'
]
```

- We are using the best performing models on the CIFAR-10 test dataset trained with each augmentation technique. These models were selected either from the ablation studies(discussed) or by training the model with recommended hyperparameters/settings(AugMix).
- For CIFAR-10-C dataset there are five levels of severity for each corruption. We will compute the average error rate for each corruption and then take the average over them.

Let's have a look at the results.



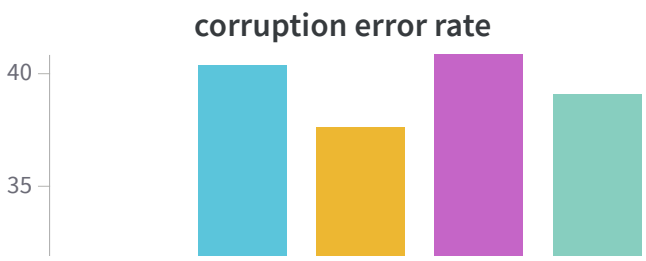
Observation 1

- The `test error rate` or standard error rate is computed on Cifar-10 test dataset.
- As observed earlier from the validation loss, the model performance was ranked like this:

  ```
  augmix > cutmix > mixup > cutout > baseline
  ```

  The evaluation on Cifar-10 confirmed this with AugMix giving the best performance.

But how will they hold up against data shift and corruption.



Observation 2

- The `corruption error rate` is computed on Cifar-10-C dataset.
- We can see that none of the augmentation techniques other than AugMix could help make the model robust to data shift.
- Training with chains of multiple simpler augmentations allowed for more generalization compared to CutMix, Mixup and Cutout.
- The error rate is almost half of the other methods and is approaching `test error rate`. Similar result is shown by the authors of AugMix. This study further confirms there case.

I have also shown the corruption error rate for all the mentioned corruptions for baseline and baseline+AugMix model. For all the corruptions AugMix is lowering the error rate by half. Similar result is shown by the authors.

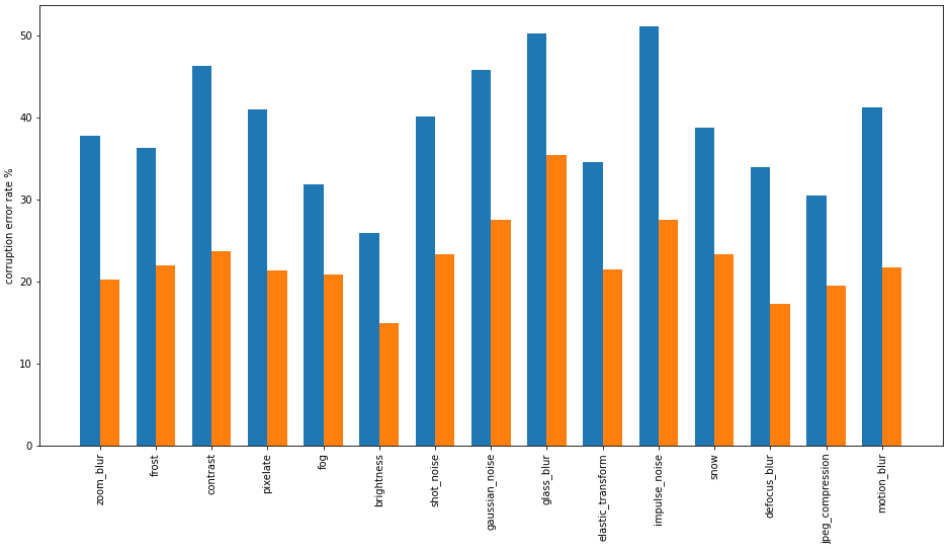


Figure 7: Corruption error rate on Cifar-10-C dataset for baseline and baseline+augmix model

Summary

This report showcases various modern data augmentation techniques that fit easily in your deep learning pipeline especially for image classification. Cutout, Mixup, and CutMix augmentation techniques are purely implemented using `tf.data` and you can simply plug it in your pipeline. AugMix is implemented in TensorFlow 2.x. You only need to replace the model in `model.py` file with your model and with some minor changes.

The report also delved into model robustness or how the model behaves under data shift. The report shows how different augmentation strategies stand under data corruption – **AugMix being the clear winner.** The results produced are comparable to the one showed by the authors of AugMix, further confirming the claims.

I hope you liked this comparative study. For any feedback reach out to me on Twitter @ayushthakur0.

Experiment with these augmentation techniques in Colab →