Baseline training, no augmentation:

2. Model initializing

```
# select resNet18 from pytorch
model = models.resnet18(pretrained=False)
# fully connected, 10 classes
model.fc = nn.Linear(model.fc.in_features, 10)
# use cross entropy as loss function
criterion = nn.CrossEntropyLoss()
# Adam to optimize the ggradient
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

3. Training Function

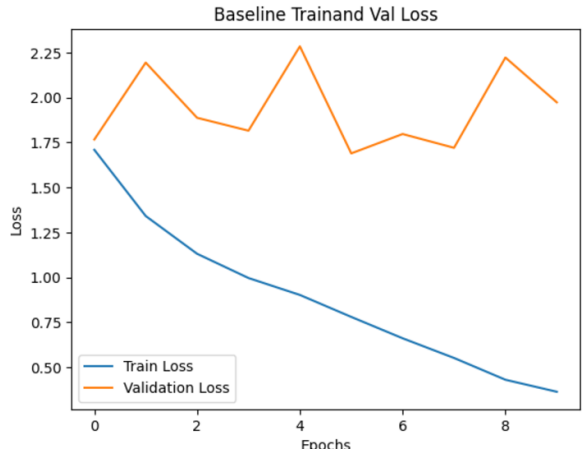
The structure is from https://gist.github.com/MLWhiz/2cd4712647f72d4078caf4d76b650717

We will track the

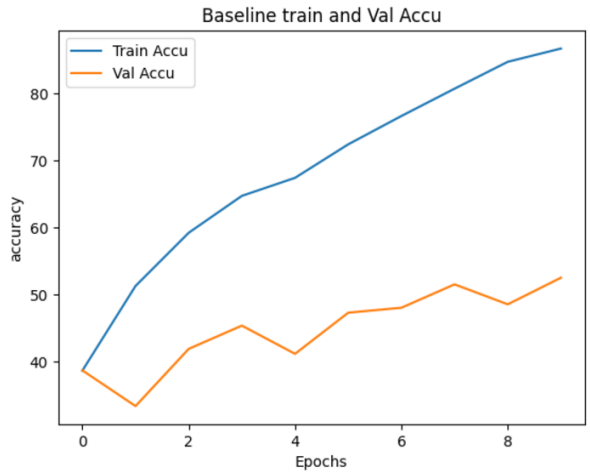
1. train losses over epoch
2. train accuracies
3. validation loss
4. validation accuracy

```
[25] def train_baseline(model, train_loader, test_loader, criterion, optimizer, num_epochs=10, device='cuda
       model.to(device)

       # save learning records
       train_losses = []
       train_accuracies = []
       val_losses = []
       val_accuracies = []

       for epoch in range(num_epochs):
         print(epoch + 1,"th epoch out of ", num_epochs)
         model.train()

         running_loss = 0
         correct = 0
         total = 0
```

Loss curve:



Accuracy curve:



Observation:

The model has a smooth and stable loss and accuracy curve **on the training set**. As a result, achieve an accuracy about 80%. However, on the validation set, loss and accuracy curves fluctuate. At the end, it achieved accuracy about 50%, and during the learning process, the accuracy increase slowly. The model seems overfitting and has poor performance on generalizing datapoints.

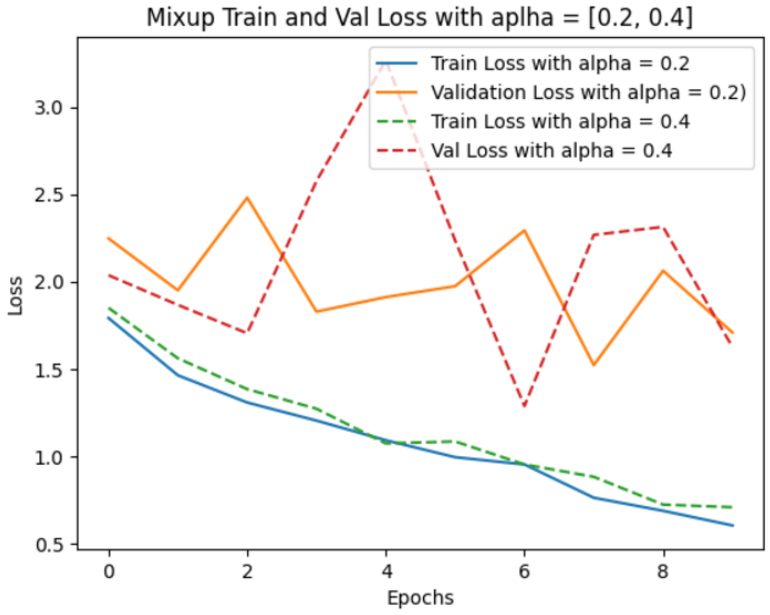
For mixUp augmentation

Instead of using original image, mixup combine 2 or more images based on the original dataset and generate a “new” example. And their labels also “mixuply” labeled. The Mixing_lambda controls how the new example is generated. Alpha = 0.9, image1 mixup10% with image2. Alpha = 0.1, image1 mixup 90% with image2

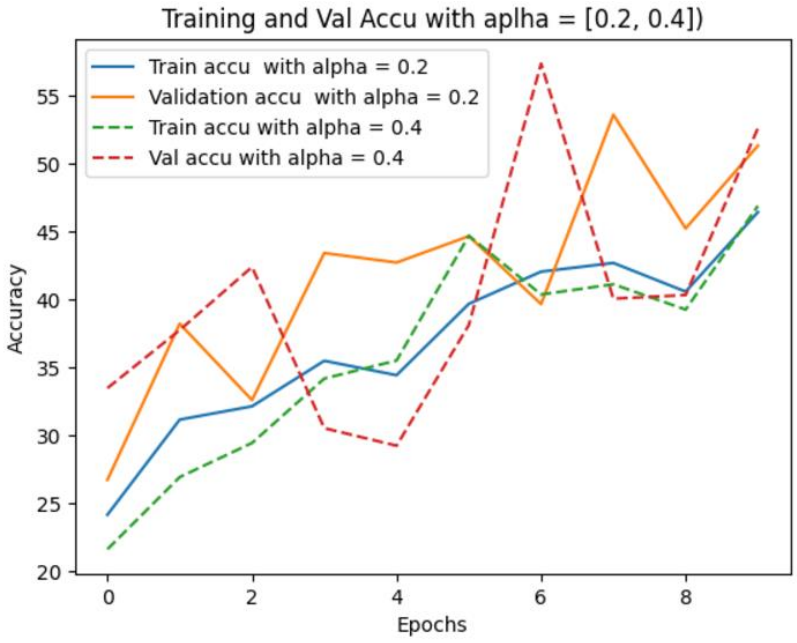
$$x = \lambda x_1 + (1-\lambda)x_2 \quad y = \lambda y_1 + (1-\lambda)y_2$$

```
def mixup_data(x, y, alpha=0.2):
  if alpha > 0.:
        Mixing_lambda = np.random.beta(alpha, alpha)
  else:
        Mixing_lambda = 1.

  batch_size = x.size(0)
  index = torch.randperm(batch_size)

  mixed_x = Mixing_lambda * x + (1 - Mixing_lambda) * x[index, :]
  mixed_y = Mixing_lambda * y + (1 - Mixing_lambda) * y[index, :]
  return mixed_x, mixed_y
```

Plot the loss for alpha = [0.2, 0.4]



Plot the accuracy:



For the loss curve:

Alpha =0.2 and 0.4 has similar loss curve on training set. Both smooth and stable. On the validation loss curve, both Alpha are fluctuates. Alpha =0.4 has a even unstable validation loss. It might due to it mixUp more features with other images.

For accuracy curve:

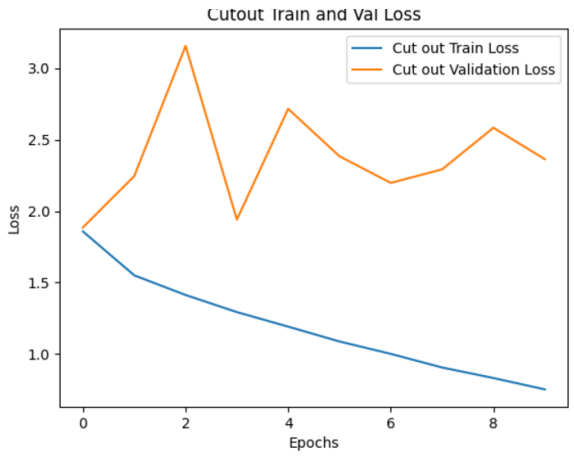
Curves share similar patterns. Alpha=0.2and0.4 has a relatively stable accuracy curve on the training set. For the validation set, fluctuate aggressive matches the result from loss curve.

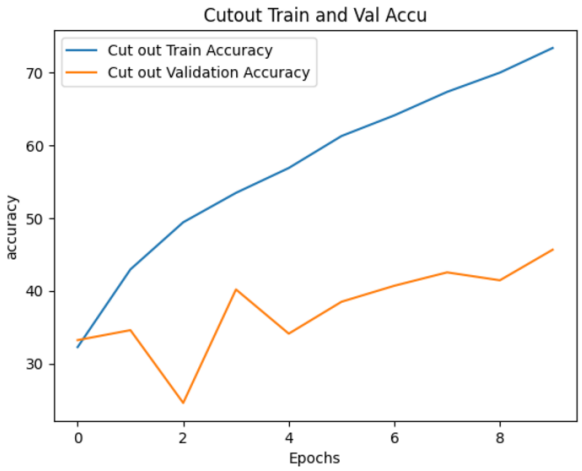
Cutout augmentation:

It hide(mask) a sub-region of an image(replace them with 0 or other constant values). Helps reduce overfitting

1. Cutout Augmentation Function

```
def cutout(image, cutout_size):
  channel, height, width = image.shape
  mask = np.ones((height, width))

  # select a sub-region from the image
  y_center = np.random.randint(0, height)
  x_center = np.random.randint(0, width)

  # mask boundaries
  y1 = max(0, y_center - cutout_size // 2)
  y2 = min(height, y_center + cutout_size // 2)
  x1 = max(0, x_center - cutout_size // 2)
  x2 = min(width, x_center + cutout_size // 2)

  # cut masks
  mask[y1:y2, x1:x2] = 0.0
  image = image * torch.tensor(mask, dtype=image.dtype, device=image.device)

  return image
```



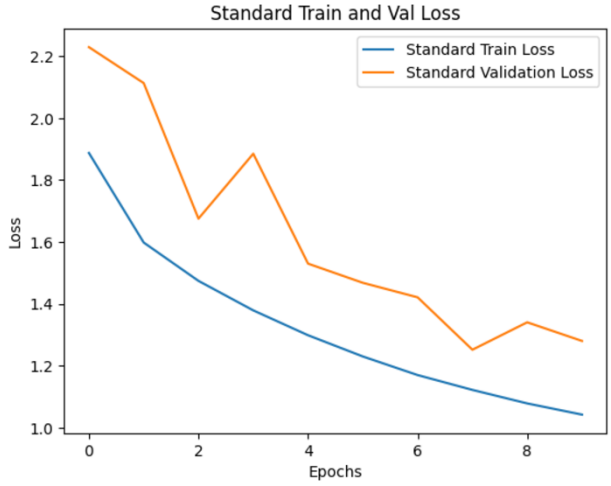


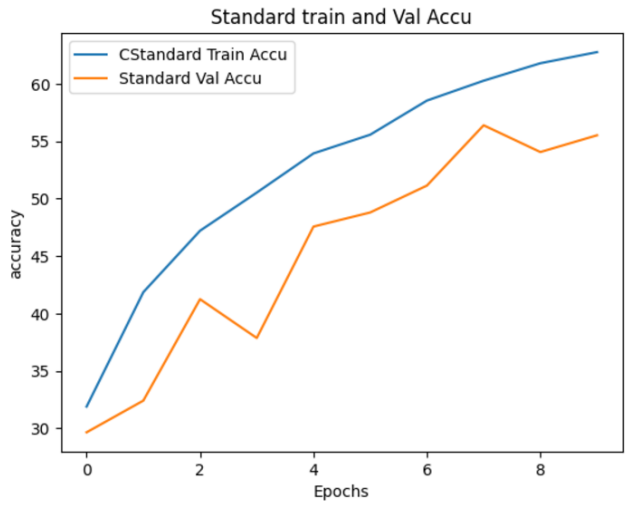
Similar learning plots as previous augmentation technics. Stable on training, fluctuate on Val

standard augmentation

“generate” new samples by shifting (zero-padding areas that no values after shifting), flipping an image. Helps reduce overfitting

```python
def standard_augmentation(image, max_shift=4):
    channel, height, width = image.shape

    # select a sub-region to shift
    k1 = np.random.randint(-max_shift, max_shift + 1)  # shifting vertically
    k2 = np.random.randint(-max_shift, max_shift + 1)  # shifting horinzaontaly

    # padding
    padded_image = torch.zeros_like(image).to(image.device)
    y_start = max(0, k1)
    y_end = height + min(0, k1)
    x_start = max(0, k2)
    x_end = width + min(0, k2)
    padded_image[:, y_start:y_end, x_start:x_end] = image[:, max(0, -k1):height - max(0, k1), max(0, -k2):width - max(0, k2)]

    # flipping image with a 50% probability
    if np.random.rand() < 0.5:
        padded_image = torch.flip(padded_image, dims=[2])

    return padded_image
```





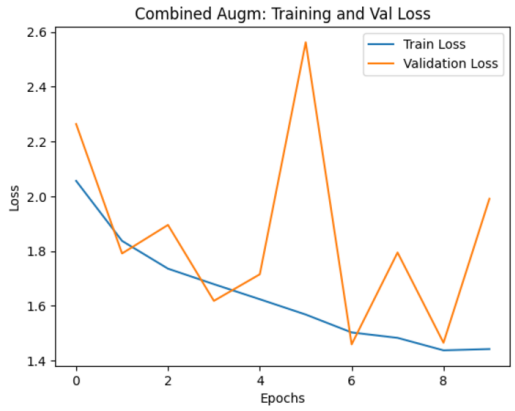
The standard augmentation constructed a better peroformence. Both train and validation curves much stable and smooth compare to previous techniques. The gap between train and validation is relatively closer, indicating not siginficantlly overfitted. The Val accuracy achieved about 55%.

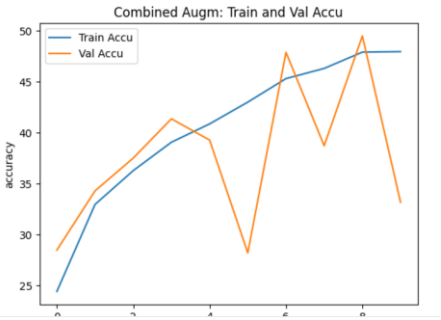
Transform images with all previous implement augmentation techniques. mixUP alpha = 0.2, shift = 4, and mask-size = 16

```python
class CombinedTransform:
  def __init__(self, cutout_size=16, max_shift=4, mixup_alpha=0.2, mixup_prob=0.5):
    self.cutout_size = cutout_size #cutout_size: Size of the Cutout sub-region
    self.max_shift = max_shift# shift for standard.
    self.mixup_alpha = mixup_alpha#Alpha for Mixup augmentation, choose a = 0.2 this case.
    self.mixup_prob = mixup_prob# Mixup.

  def standard_and_cutout(self, image):
    # Apply standard + cutout augm to a image
    image = standard_augmentation(image, max_shift=self.max_shift)
    image = cutout(image, cutout_size=self.cutout_size)
    return image

  def __call__(self, images, labels):
    # Apply standard + cutout augm to all images
    augmented_images = torch.stack([self.standard_and_cutout(img) for img in images])

    # Apply Mixup
    if np.random.rand() < self.mixup_prob:
      lam = np.random.beta(self.mixup_alpha, self.mixup_alpha)
      indices = torch.randperm(augmented_images.size(0))
      mixed_images = lam * augmented_images + (1 - lam) * augmented_images[indices, :]
      mixed_labels = lam * labels + (1 - lam) * labels[indices]
      return mixed_images, mixed_labels

    return augmented_images, labels
```

Dataset Loader with CombinedAugmentation

```python
# # Convert scalar labels to one-hot encoding(to_one_hot defined in mixup)
y_train_one_hot = to_one_hot(torch.tensor(y_train_subset, dtype=torch.long))
y_test_one_hot = to_one_hot(torch.tensor(test_labels, dtype=torch.long))

# set up transformer with CombinedTransform
combined_transform = CombinedTransform(cutout_size=16, max_shift=4, mixup_alpha=0.2)  # Use a =0.2 based on observation

# DataLoader with CombinedTransform
train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True, collate_fn=lambda batch: combined_transform(
  torch.stack([x for x, label in batch]),
  torch.stack([to_one_hot(y, classes=10) for label, y in batch])
))

test_loader = DataLoader(test_dataset, batch_size=100, shuffle=False, collate_fn=lambda batch: (
  torch.stack([x for x, label in batch]),
  torch.stack([to_one_hot(y, classes=10) for label, y in batch])
))
```





Curves on training set are stable and smooth, but overall performance is worse with accuracy around 50%. Validation performance is fluctuated

Comments on role of data augmentation

The data augmentation techniques in this implementation tends to help model better generalize datapoints by transforming and modifying the training dataset. Theoretically, these techniques prevent model from overfitting.

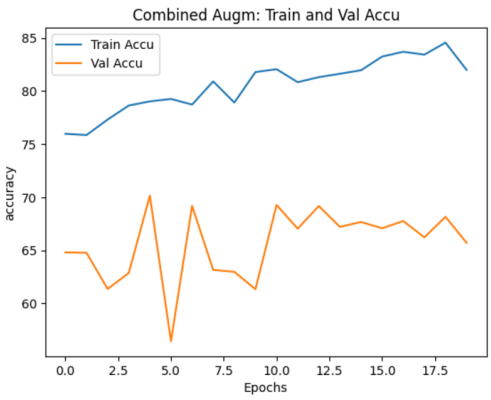
Based on the learning plots, while these techniques minimized the gap between training and validation curve, models still facing some level of overfitting. Also, after implementing these techniques except standard augmentation, the validation accuracy decreased compare to baseline (no augmentation). For converged speed, all models tends have slower converge speed compare to baseline, I suspect its due to the distortion and augmentations applied on the datasets, since have difficulties to capture modified/transformer/masked images. In short, data augmentations are usually good, but select them with cautious and care the hyperparameter is critical to help model generalize datapoints.

Finally, we have 10 classes, with random guessing accuracy at 10%, all models have a val accuracy above 40%, which means they learned some features during the training process. With a better tunning of hyperparameter and apply appropriate techniques_combination to images can potentially increase models performance.

Addition tuning: This training sets a mixUP alpha = 0.3, shift = 2, and mask-size = 9, and 20 epochs to see a longer behavior of the accuracy curve

Some Additional hyper parameter tunning

```
# # Convert scalar labels to one-hot encoding(to_one_hot defined in mixup)
y_train_one_hot = to_one_hot(torch.tensor(y_train_subset, dtype=torch.long))
y_test_one_hot = to_one_hot(torch.tensor(test_labels, dtype=torch.long))

# set up transformer with CombinedTransform
combined_transform = CombinedTransform(cutout_size=9, max_shift=2, mixup_alpha=0.3)  # Use a =0.3 to mixup, shift size = 2, and smaller mask-size

# DataLoader with CombinedTransform
train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True, collate_fn=lambda batch: combined_transform(
  torch.stack([x for x, label in batch]),
  torch.stack([to_one_hot(y, classes=10) for label, y in batch])
))

test_loader = DataLoader(test_dataset, batch_size=100, shuffle=False, collate_fn=lambda batch: (
  torch.stack([x for x, label in batch]),
  torch.stack([to_one_hot(y, classes=10) for label, y in batch])
))
```



For consistency, focus on 10th epochs, the accuracy is about 68% and is higher. Roughly speaking, A good choice of tunning does helps model to generalize features.