PyTorch Image Classification Common Code Template

Kanru Wang

April 2021

Reference

Training

- https://www.kaggle.com/mekhdigakhramanian/fork-pytorch-efficientnet-baseline-train-amp-a
- https://www.kaggle.com/underwearfitting/single-fold-training-of-resnet200d-lb0-965
- https://www.kaggle.com/tomehirata/pytorch-training-rfcx-adas-optimizer-resnest
- https://www.kaggle.com/haqishen/train-efficientnet-b0-w-36-tiles-256-lb0-87
- https://www.kaggle.com/tanulsingh077/pytorch-metric-learning-pipeline-only-images
- https://www.kaggle.com/underwearfitting/pytorch-densenet-arcface-validation-training

Inference

- https://www.kaggle.com/kanruwang/cassava-ensemble-efficientnet-resnext-private0-901
- https://www.kaggle.com/japandata509/ensemble-resnext50-32x4d-efficientnet-0-903
- https://www.kaggle.com/raghaw/ensemble-of-2-best-public-notebooks (single model)
- https://www.kaggle.com/kneroma/inference-tpu-rfcx-audio-detection-fast (no TTA)
- https://www.kaggle.com/haqishen/panda-inference-w-36-tiles-256 (not normal TTA)
- https://www.kaggle.com/iafoss/panda-concat-tile-pooling-starter-inference
- https://www.kaggle.com/parthdhameliya77/nfnet-l0-efficientnet-b5-ensemble-inference (no TTA)

Ensemble as an alternative to "train on entire dataset"

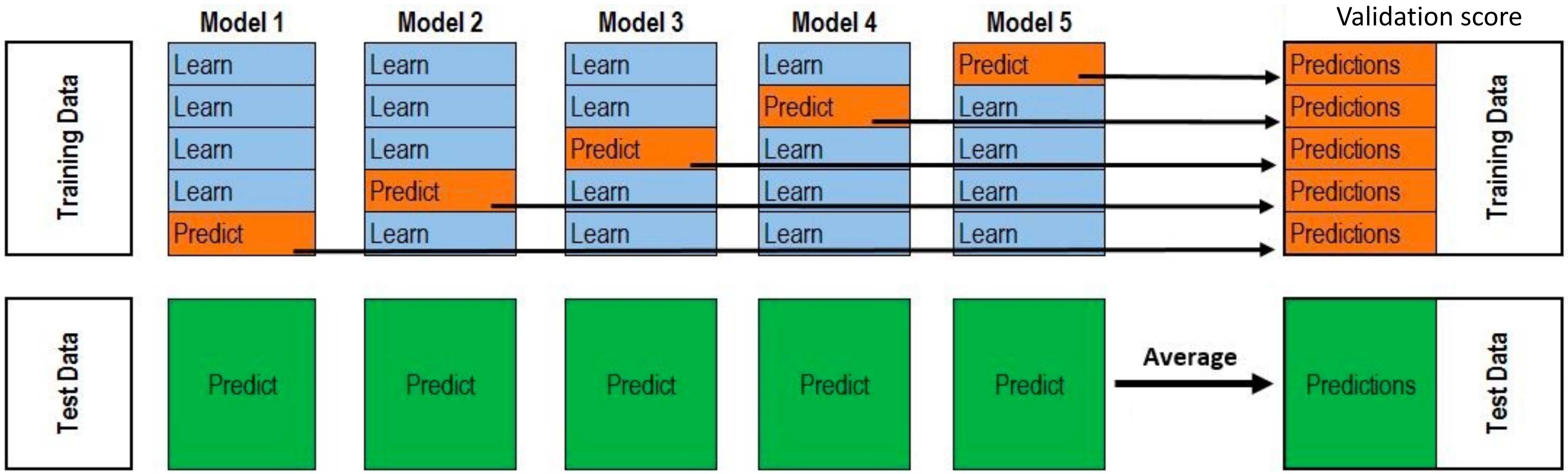
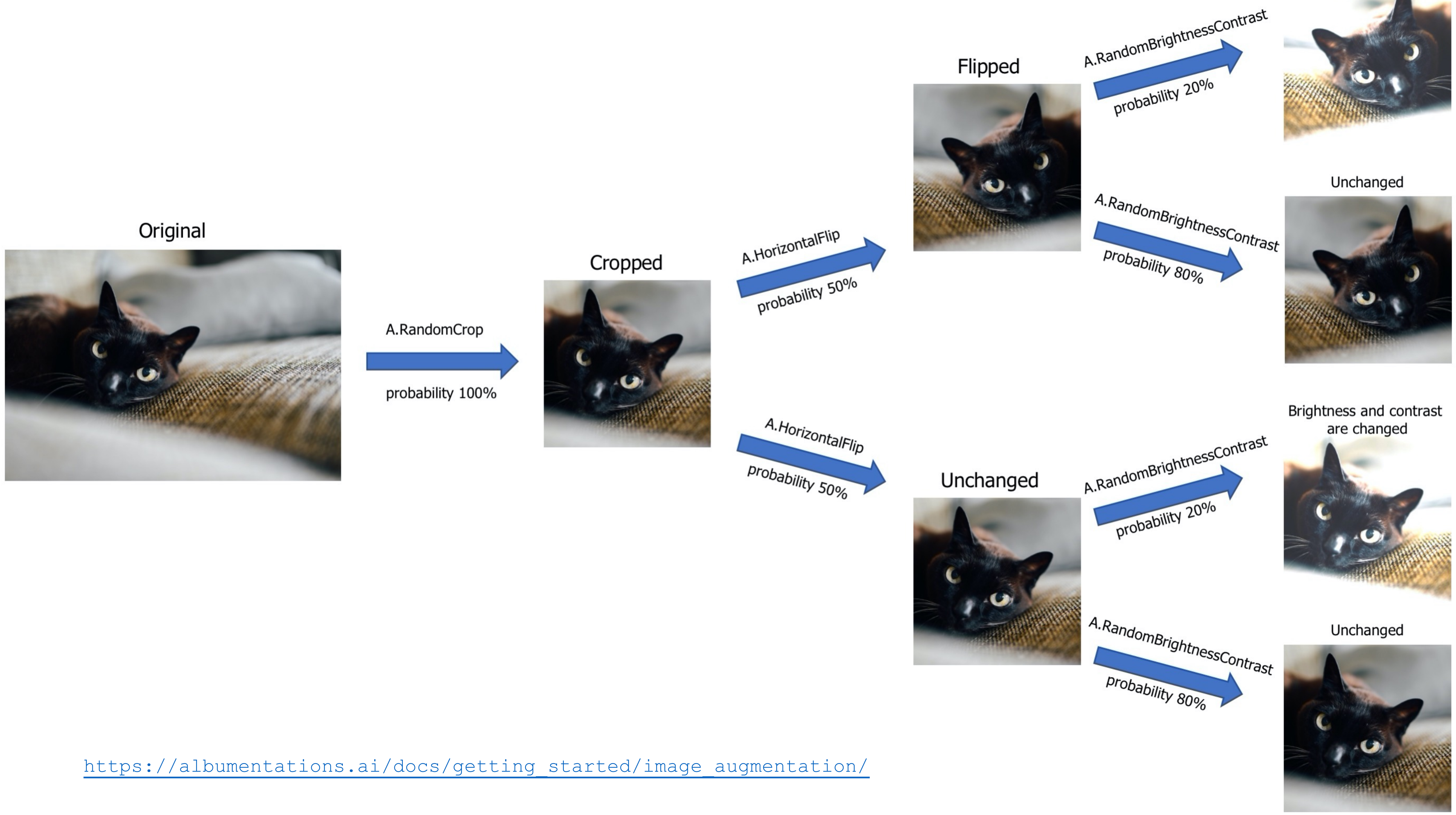
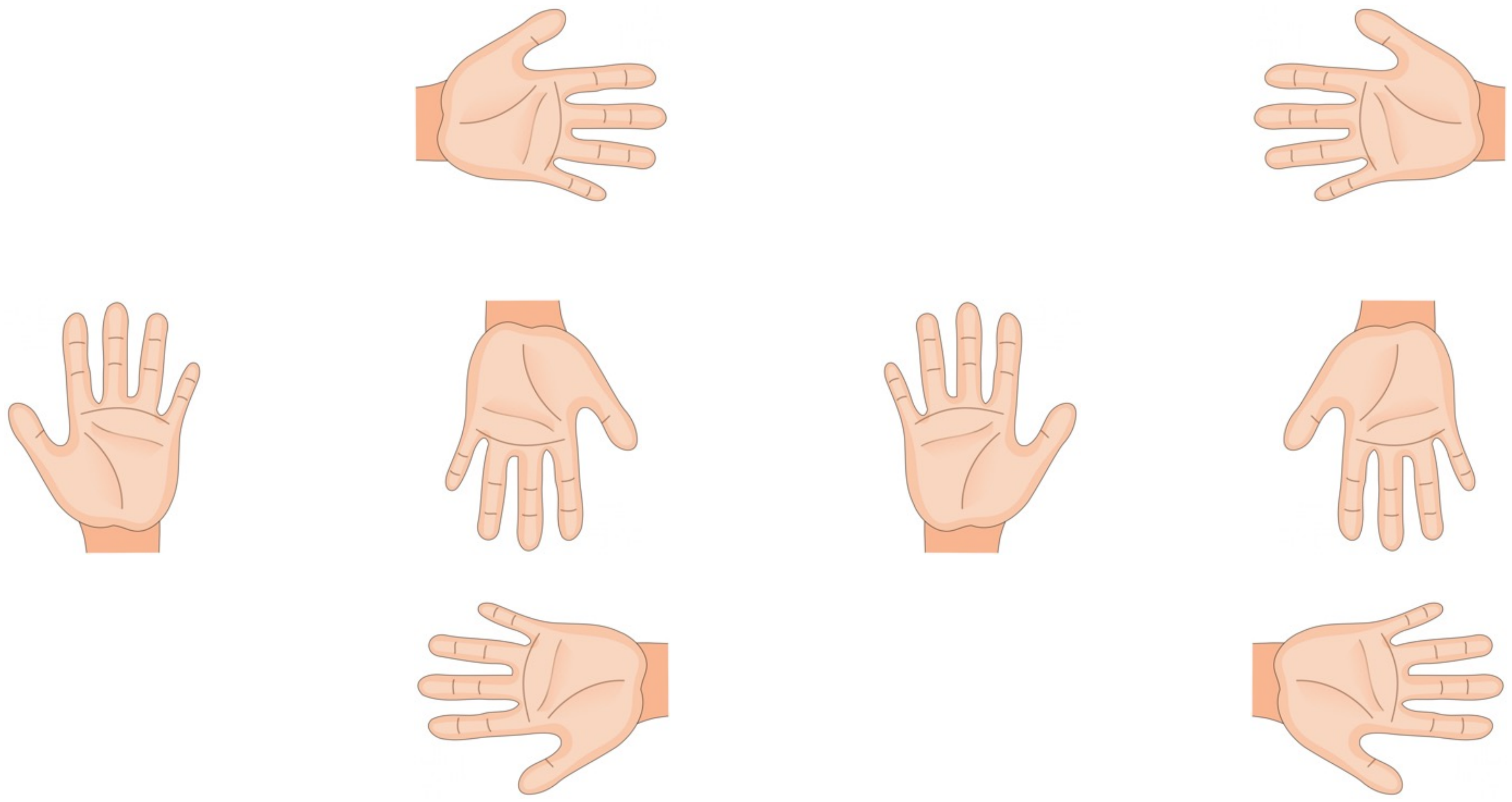


Image Augmentation and Why Do We Need It



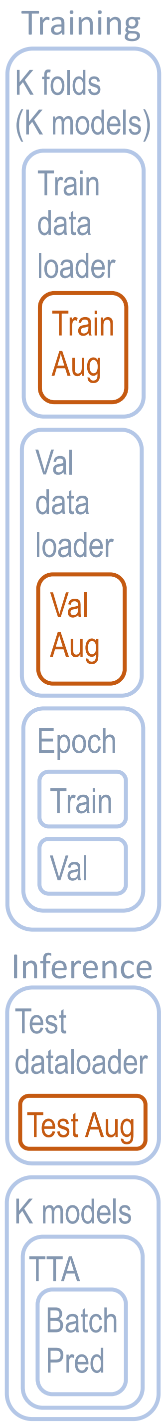
https://albumentations.ai/docs/getting_started/image_augmentation/

8 Test Time Basic Image Augmentation

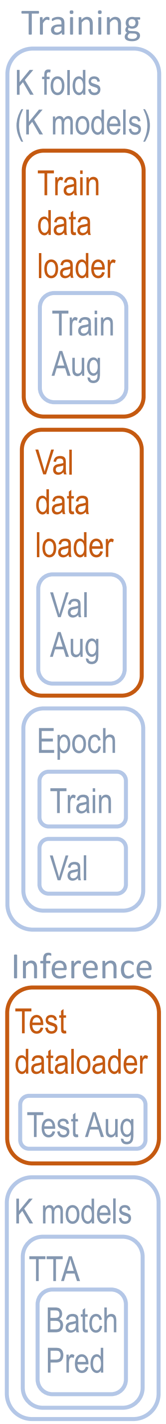


My understanding:

- During training, do not use the augmentation type that will produce images that are not likely to appear in raw testing/inference images.
- During validation or testing/inference, do not use the augmentation type that has not been used in training.

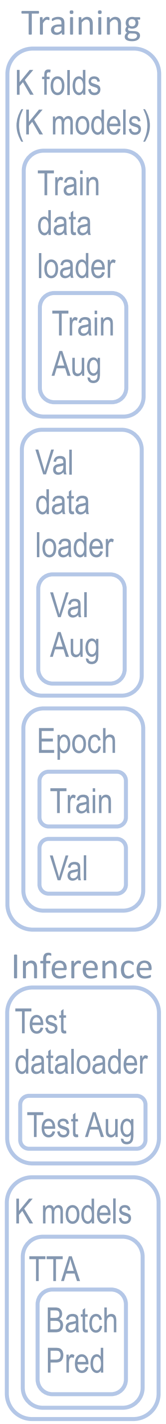



```
from albumentations import Compose, HorizontalFlip, VerticalFlip, Transpose, …


image_augmentation_pipeline_train = Compose([
    RandomResizedCrop
    HorizontalFlip
    OneOf([…,…,…])
    … many more …
    Normalize
])


image_augmentation_pipeline_valid = Compose([
    … simpler; just the basic augmentation …
])


image_augmentation_pipeline_test = Compose([
    … simpler; just the basic augmentation …
])
```




```
from torch.utils.data import Dataset
import cv2


class ImageDataset(Dataset)
    __init__(…)
        self.df
        self.image_augmentation_pipeline
    __getitem__(self, index:int)
        row = self.df.loc[index]
        img = cv2.imread(row[“image_path”])
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # OpenCV uses BGR format, Albumentations uses RGB
        img = self.image_augmentation_pipeline(img)[“image”]
        if self.output_label == True:
            return img, row['label'].values
        else:
            return img
```

There should be
(1) a folder of images
(2) a csv file that has a column of image file names (or image file path), and a column of labels.




```
from torch import nn

class Model(nn.Module)
    __init__(…)
        super().__init__()
        self.model = model_package.load_pretrained_model(“model_type”)
        n_input_features_fully_connected_layer = self.model.fc.in_features
        self.model.fc = nn.Linear(
            n_input_features_fully_connected_layer,
            n_classes_model_output
        ) # the output dimension of a default model.fc (e.g. 1000) need to be overwritten
    forward(self, x)
        ” ” ”
        forward() is called in nn.Module.__call__(). Should call the module
        directly using (output = model(data)) instead of model.forward(data).
        ” ” ”
        x = self.model(x)
        return x
```

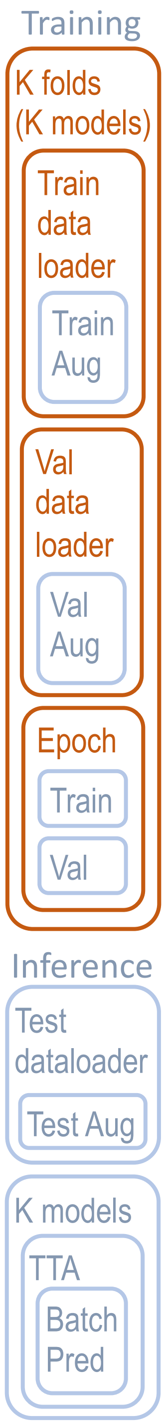
In PyTorch Image Models (timm), the fully connected layer for EfficientNet and DenseNet is called “self.model.classifier”. For ResNet, ResNeXt, SEResNet, and NFNet, it is called “self.model.fc”.

If need to add Dropout, or change Pooling type, before FC layer

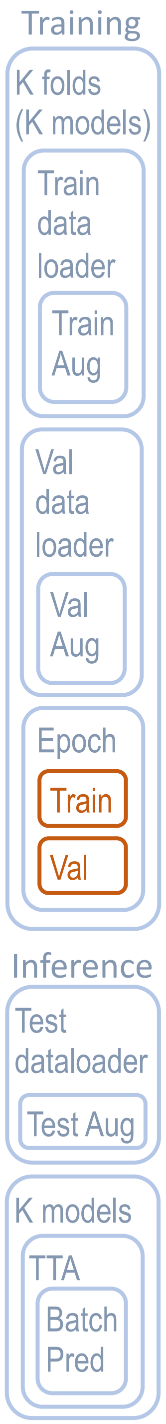
```
class Model(nn.Module)
    __init__(…)
        super().__init__()
        self.model = model_package.load_pretrained_model("model_type")
        n_input_features_fully_connected_layer = self.model.fc.in_features
        self.model.fc = nn.Identity() # set to do nothing, allowing adding Dropout; finally self.fc
        self.fc = nn.Linear(
            n_input_features_fully_connected_layer,
            n_classes_model_output
        )
        self.model.global_pool = nn.Identity() # set to do nothing; can customize pooling
        self.pooling = nn.AdaptiveAvgPool2d(1) # default global_pool is AdaptiveAvgPool2d; we may change
        self.dropout = nn.Dropout(dropout_rate) # (optional)
    forward(self, x)
        x = self.model(x) # result size[batch_size, channels, height, width] e.g. [32, 1536, 16, 16]
        x = self.pooling(x) # result size [batch_size, channels, 1, 1]
        x = x.view(batch_size, -1) # result size [batch_size, channels]
        x = self.dropout(x) # (optional) result size [batch_size, channels]
        x = self.fc(x) # result size [batch_size, n_classes_model_output]
        return x
```

When self.model(x) is called, x will go through: backbone(x) -> global_pool[i.e. pool(x) -> flatten(x)] -> fc(x)

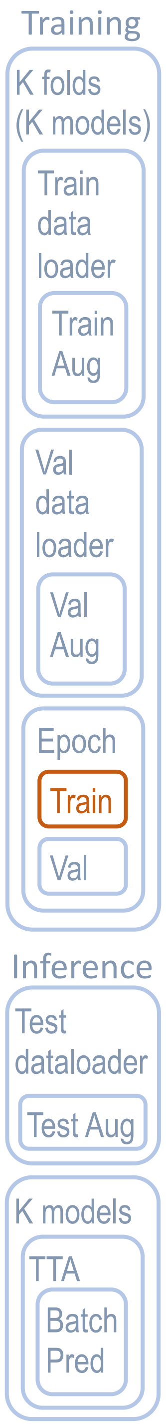
Read forward_features(), create_classifier(), and SelectAdaptivePool2d() in
https://github.com/rwightman/pytorch-image-models/blob/master/timm/models/resnet.py




```
from torch.utils.data import DataLoader
folds = StratifiedKFold(n_splits, shuffle=True, random_state=42).split(df, df[“label”])
for fold, (train_idx, valid_idx) in enumerate(folds):
    df_train = df.loc[train_idx,:].reset_index(drop=True)
    df_valid = df.loc[valid_idx,:].reset_index(drop=True)
    dataset_train = ImageDataset(df_train, image_augmentation_pipeline_train)
    dataset_valid = ImageDataset(df_valid, image_augmentation_pipeline_valid)
    train_loader = DataLoader(dataset_train, train_batch_size, shuffle=True, num_workers=num_workers)
    valid_loader = DataLoader(dataset_valid, valid_batch_size, shuffle=False, num_workers=num_workers)
    model = Model(…).to(device)
    optimizer = torch.optim.Adam(model.parameters(), learning_rate)
    scheduler = … instantiate the learning rate scheduler …
    loss_func = … instantiate the loss function …
    for epoch in range(num_epochs):
        model = train_one_epoch(…)
        eval_metric = valid_one_epoch(…)
        early_stopping_counter += 1
        if eval_metric > best_eval_metric:
            torch.save(model.state_dict(), path)
            early_stopping_counter = 0
        if early_stopping_counter == early_stopping_limit:
            break
```



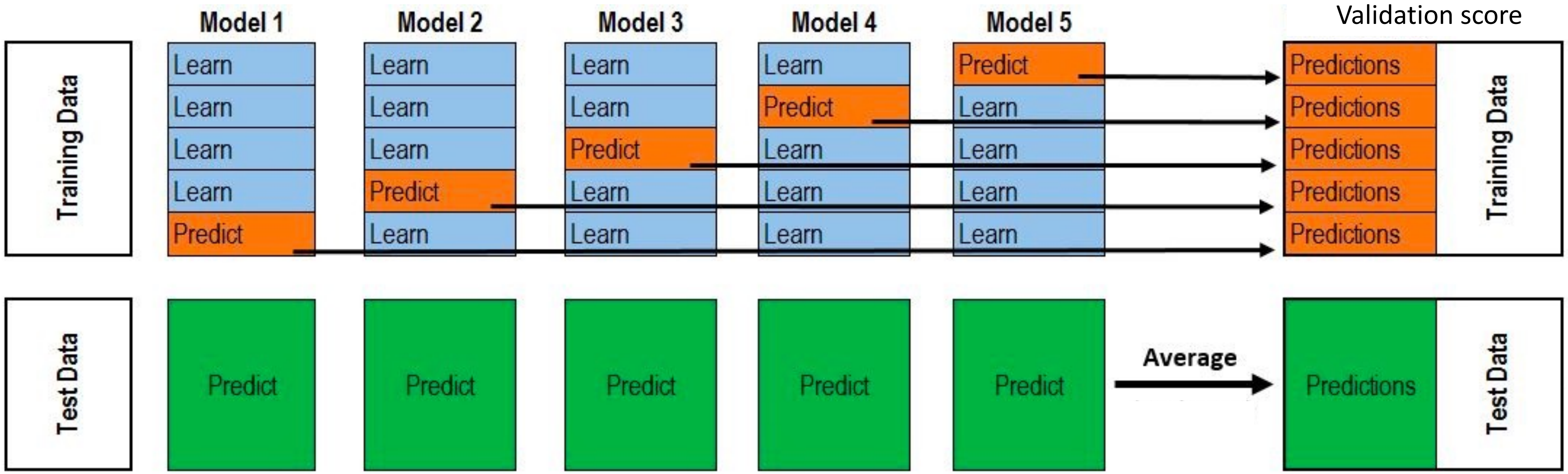

```
optimizer = torch.optim.Adam(model.parameters(), learning_rate)
device = torch.device("cuda”)
def train_one_epoch
    model.train() # for Batchnorm and Dropout to work properly
    for (imgs, labels) in tqdm(train_loader):
        imgs, labels = imgs.to(device), labels.to(device) # moves a model / tensor to GPU
        optimizer.zero_grad() # clears old gradients from the last step’s loss.backward()
        logits = model(imgs)
        loss = loss_func(logits, labels) # CrossEntropyLoss or BCEWithLogitsLoss
        loss.backward() # computes the derivative of the loss w.r.t. each parameter using backprop
        optimizer.step() # optimizer updates each parameter using the its gradient and the optimizer rules
        scheduler.step() # can be outside of the for-loop to update LR every epoch, not every batch
        … attach the loss to a list …
    … print the average loss …
def valid_one_epoch
    model.eval() # for Batchnorm and Dropout to work properly
    with torch.no_grad(): # deactivate backprop to reduce memory usage
        for (imgs, labels) in tqdm(valid_loader):
            imgs, labels = imgs.to(device), labels.to(device)
            logits = model(imgs)
            loss = loss_func(logits, labels)
            … attach the loss to a list; attach predictions to a list …
    … print the average loss; calculate and print the evaluation metric …
```



(Optional) Training with Mixed Precision

```
from torch.cuda.amp import autocast, GradScaler # for Mixed Precision
optimizer = torch.optim.Adam(model.parameters(), learning_rate)
device = torch.device("cuda”)
scaler = GradScaler()

def train_one_epoch
    model.train()
    for (imgs, labels) in tqdm(train_loader):
        imgs, labels = imgs.to(device), labels.to(device)
        with autocast(): # cast weights from FP32 to FP16 before the forward pass
            logits = model(imgs)
            loss = loss_func(logits, labels) # CrossEntropyLoss or BCEWithLogitsLoss
        scaler.scale(loss).backward() # scale up the loss before the backward pass
        scaler.step(optimizer) # scale down the gradient; then update weights
        scaler.update() # adjust the scaling factor for the next batch
        optimizer.zero_grad() # clears old gradients from the current batch
```

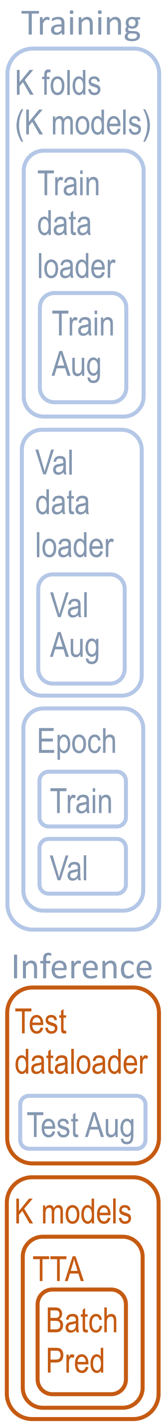
Ensemble as an alternative to “train on entire dataset”



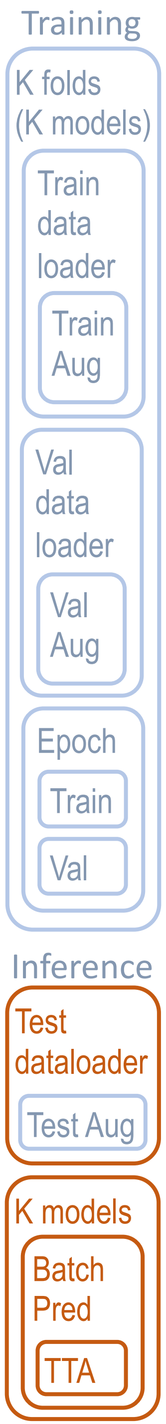
Assume 6 output classes, 4 ensemble models, 8 TTA. For each testing image…

	Model 1	Model 2	Model 3	Model 4	
Class 1					Avg of 32
Class 2		8 TTA images			Avg of 32
Class 3		1 2 3 4 5 6 7 8			Avg of 32
Class 4					Avg of 32
Class 5					Avg of 32
Class 6					Avg of 32

The largest value will be the predicted class




```
dataset_test = ImageDataset(df_test, image_augmentation_pipeline_test_with_flip_and_transpose, output_label=False)
test_loader = DataLoader(dataset_test, test_batch_size, shuffle=False, num_workers=num_workers)
model_paths = ["path_to_fold_1_trained_model", "path_to_fold_2_trained_model" … ]
Usually exists three for-loops: for each model, for each inference batch, and for each TTA. These loops can be written in any order.
Example 1 (Model Loop -> TTA Loop -> Batch Loop)
logits_epoch_TTA_averaged_all_models = []
for e in range(model_paths):
    model = Model(…).to(device)
    model.load_state_dict(torch.load(model_paths[e]))
    model.eval()
    logits_epoch_TTA_averaged = []
    with torch.no_grad():
        for _ in range(number_of_TTA):
            logits_all = []
            for imgs in tqdm(test_loader):
                imgs = imgs.to(device)
                logits = model(imgs)
                logits_epoch += [torch.softmax(logits, 1).detach().cpu().numpy()]
            logits_epoch_TTA_averaged += [np.concatenate(logits_epoch, axis=0)]
        logits_epoch_TTA_averaged = np.mean(logits_epoch_TTA_averaged, axis=0) # Average over TTA
        logits_epoch_TTA_averaged_all_models.append(logits_epoch_TTA_averaged)
predicted_class = np.mean(logits_epoch_TTA_averaged_all_models, axis=0).argmax(1) # Avg over all models' output; then argmax
```




```
dataset_test = ImageDataset(df_test, image_augmentation_pipeline_test_no_flip_or_transpose, output_label=False)
test_loader = DataLoader(dataset_test, test_batch_size, shuffle=False, num_workers=num_workers)
model_paths = ["path_to_fold_1_trained_model", "path_to_fold_2_trained_model" … ]
Usually exists three for-loops: for each model, for each inference batch, and for each TTA. These loops can be written in any order.
Example 2 (Model Loop -> Batch Loop -> TTA)
logits_TTA_averaged_epoch_all_models = []
for e in range(model_paths):
    model = Model(…).to(device)
    model.load_state_dict(torch.load(model_paths[e]))
    model.eval()
    logits_TTA_averaged_epoch = []
    with torch.no_grad():
        for imgs in tqdm(test_loader):
            x = imgs.to(device)
            x = torch.stack([x, x.flip(-1), x.flip(-2), x.flip(-1, -2), x.transpose(-1, -2), x.transpose(-1, -2).flip(-1),
                             x.transpose(-1, -2).flip(-2), x.transpose(-1, -2).flip(-1, -2)], 0) # 8 basic TTA
            x = x.view(-1, 3, img_size, img_size) # result size [inference_batch_size x number_of_TTA, 3, img_size, img_size]
            logits = model(x)
            logits_TTA_averaged = logits.view(batch_size, 8, -1).mean(1)
            logits_TTA_averaged_epoch += [torch.softmax(logits_TTA_averaged, 1).detach().cpu()]
        logits_TTA_averaged_epoch = torch.cat(logits_TTA_averaged_epoch).cpu().numpy()
    logits_TTA_averaged_epoch_all_models += [logits_TTA_averaged_epoch]
predicted_class = np.mean(logits_TTA_averaged_epoch_all_models, axis=0).argmax(1)
```