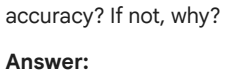
FixMatch: Simplifying Semi-Supervised Learning with Consistency and Confidence

In this practical, we are going to focus on semi-supervised learning, which refers to the situation of using unlabeled data to enhance the performance of models on a classification task. The method we are focusing on, is called FixMatch [1] which has simplified the process of semi-supervised learning compared to the recent state-of-the-art methods.

Question 1: Come up with an application in which semi-supervised would be indispensable.

Answer:

The idea of semi-supervised learning is based on the fact that often unlabeled data can be obtained quite easily, and used in many situations where labelling is costly. The recent literature, however, presents semi-supervised learning using complicated methods. [1] proposes a method, called FixMatch, to simplify the complexity of the process by using a simple pipeline shown below, which uses the model predictions on weakly augmented images, as pseudo-labels.



It is note worthy that, if we use the output of the model to train itself, then we are doing something called self-training. In self training, we use model to produce some labels to train on, and the labels are called pseudo labels.

The summary of FixMatch's method is that, besides the simple supervised loss $\square_s$, we also have an unsupervised loss $\square_u$, which utilize weak and strong image augmentation strategies, denoted by α and $\square$ respectively. By adding an unsupervised loss $\square_u$ to the total loss, we force the model to produce the same output for strongly augmented image, $\square(x)$, as for the weakly augmented one, namely $\alpha(x)$. There is a condition for using unsupervised loss, however, ensuring the model is "confident enough" about its prediction for the weakly augmented image to be considered as a pseudo-label.

Formally, we show the model output as $p_m(y|x)$ and assume the total loss is a combination of two losses:

$$\square_t = \square_s + \square_u$$

As before, $\square_s$ is a simple cross-entropy loss, and

$$\square_u = \frac{1}{\mu B} \sum_{x \in \square} \mathbb{H}(p_x, q_x)$$

In which, $\mathbb{H}$ is a classification loss like cross-entropy, $p_x = p_m(y|\alpha(x))$ refers to prediction on the weakly augmented image, $q_x = p_m(y|\square(x))$ refers to prediction on the strongly augmented image, and $\square = \{x : \max p_m(y|\alpha(x)) \geq \tau\}$. Also B is the labeled mini-batch size and μB is the unlabeled mini-batch size.

Intuitively, $\square$ contains samples, which our model deems to belong to some specific class, and τ controls the extent to which we consider an unsure prediction, as a pseudo-label. Our model, finally, uses its own confident-enough predictions as pseudo-labels and considers them as training samples to learn from.

Question 2: The way that the method employs weak and strong augmentation is inspired by the idea of **consistency regularization**, which is simple but effective. Describe **consistency regularization** in one sentence, and give one example about how to use the idea (except FixMatch).

Answer:

Question 3: The method uses its own confident-enough predictions as pseudo labels, does it mean high confidence is equal to high accuracy? If not, why?

Answer:

Despite its simplicity, it works surprisingly well. To see how great its effect can be on the model performance, we prepared this practical to test the idea on a small dataset like CIFAR-10.

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/KTH/DD2610/4_Under-Supervised/fixmatch-main
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
/content/drive/MyDrive/KTH/DD2610/4_Under-Supervised/fixmatch-main
```

```
%pip install jax-resnet omegaconf jax_metrics==0.2.4
```

```
Requirement already satisfied: jax-resnet in /usr/local/lib/python3.12/dist-packages (0.0.4)
Requirement already satisfied: omegaconf in /usr/local/lib/python3.12/dist-packages (2.3.0)
Requirement already satisfied: jax_metrics==0.2.4 in /usr/local/lib/python3.12/dist-packages (0.2.4)
Requirement already satisfied: einops>=0.4.0 in /usr/local/lib/python3.12/dist-packages (from jax_metrics==0.2.4) (0.8.1)
Requirement already satisfied: jax in /usr/local/lib/python3.12/dist-packages (from jax_metrics==0.2.4) (0.7.2)
Requirement already satisfied: jaxlib in /usr/local/lib/python3.12/dist-packages (from jax_metrics==0.2.4) (0.7.2)
Requirement already satisfied: optax>=0.1.1 in /usr/local/lib/python3.12/dist-packages (from jax_metrics==0.2.4) (0.2.6)
Requirement already satisfied: simple_pytree>=0.1.3 in /usr/local/lib/python3.12/dist-packages (from jax_metrics==0.2.4) (0.1.5)
Requirement already satisfied: flax in /usr/local/lib/python3.12/dist-packages (from jax-resnet) (0.10.7)
Requirement already satisfied: antlr4-python3-runtime==4.9.* in /usr/local/lib/python3.12/dist-packages (from omegaconf) (4.9.3)
Requirement already satisfied: PyYAML>=5.1.0 in /usr/local/lib/python3.12/dist-packages (from omegaconf) (6.0.3)
Requirement already satisfied: absl-py>=0.7.1 in /usr/local/lib/python3.12/dist-packages (from optax>=0.1.1->jax_metrics==0.2.4) (1.4.0)
Requirement already satisfied: chex>=0.1.87 in /usr/local/lib/python3.12/dist-packages (from optax>=0.1.1->jax_metrics==0.2.4) (0.1.90)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.12/dist-packages (from optax>=0.1.1->jax_metrics==0.2.4) (2.0.2)
Requirement already satisfied: ml_dtypes>=0.5.0 in /usr/local/lib/python3.12/dist-packages (from jax->jax_metrics==0.2.4) (0.5.4)
Requirement already satisfied: opt_einsum in /usr/local/lib/python3.12/dist-packages (from jax->jax_metrics==0.2.4) (3.4.0)
Requirement already satisfied: scipy>=1.13 in /usr/local/lib/python3.12/dist-packages (from jax->jax_metrics==0.2.4) (1.16.3)
Requirement already satisfied: msgpack in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (1.1.2)
Requirement already satisfied: orbax-checkpoint in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (0.11.28)
Requirement already satisfied: tensorstore in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (0.1.79)
Requirement already satisfied: rich>=11.1 in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (13.9.4)
Requirement already satisfied: typing_extensions>=4.2 in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (4.15.0)
Requirement already satisfied: treescope>=0.1.7 in /usr/local/lib/python3.12/dist-packages (from flax->jax-resnet) (0.1.10)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from chex>=0.1.87->optax>=0.1.1->jax_metrics==0.2.4) (75.2.0)
Requirement already satisfied: toolz>=0.9.0 in /usr/local/lib/python3.12/dist-packages (from chex>=0.1.87->optax>=0.1.1->jax_metrics==0.2.4) (0.12.1)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.12/dist-packages (from rich>=11.1->flax->jax-resnet) (4.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.12/dist-packages (from rich>=11.1->flax->jax-resnet) (2.19.2)
Requirement already satisfied: etils[epath,epy] in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (1.13.0)
Requirement already satisfied: nest_asyncio in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (1.6.0)
Requirement already satisfied: aiofiles in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (24.1.0)
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (5.29.5)
Requirement already satisfied: humanize in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (4.14.0)
Requirement already satisfied: simplejson>=3.16.0 in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (3.20.2)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from orbax-checkpoint->flax->jax-resnet) (5.9.5)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.12/dist-packages (from markdown-it-py>=2.2.0->rich>=11.1->flax->jax-resnet) (0.1.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.12/dist-packages (from etils[epath,epy]->orbax-checkpoint->flax->jax-resnet) (2025.3.0)
Requirement already satisfied: importlib_resources in /usr/local/lib/python3.12/dist-packages (from etils[epath,epy]->orbax-checkpoint->flax->jax-resnet) (6.5.2)
Requirement already satisfied: zipp in /usr/local/lib/python3.12/dist-packages (from etils[epath,epy]->orbax-checkpoint->flax->jax-resnet) (3.23.0)
```

```
# import libraries
import numpy as np
from datetime import datetime
```

```
from torch.utils import data
import copy
from functools import partial
import torch
import matplotlib.pyplot as plt
from jax.example_libraries import optimizers
import jax
import os
from jax_metrics.regularizers import L2
from jax import nn
import jax.numpy as jnp
import jax_resnet
import torchvision
from omegaconf import OmegaConf
import torch.utils.data as data_utils
from torch.utils.data import RandomSampler
from jax import config
from flax.core.frozen_dict import freeze, unfreeze
from torch.utils.tensorboard import SummaryWriter

config.update("jax_debug_nans", True)
config.update("jax_disable_jit", False)

from src.utils import stratified_split

seed = 0
general_key = jax.random.PRNGKey(seed)

assert os.path.exists('jaxlog-base'),\
    "In order to compare our results, we need this folder"\
    " please make sure that the relative path to the"\
    " folder is correct."

%load_ext tensorboard
```

```
print("downloading the checkpoint")
!mkdir checkpoints
!gdown https://drive.google.com/uc?id=1gWFupJVSsLrL_OI3wmfTGm__H7MH2H_c -O checkpoints/

assert os.path.exists('checkpoints/checkpoint-20000.npy'),\
    "In order to compare our results, we need this"\
    " checkpoint please make sure that the relative path"\
    " to the folder or the checkpoint is correct."
```

```
downloading the checkpoint
Downloading...
From: https://drive.google.com/uc?id=1gWFupJVSsLrL_OI3wmfTGm__H7MH2H_c
To: /content/drive/MyDrive/KTH/DD2610/4_Under-Supervised/fixmatch-main/checkpoints/checkpoint-20000.npy
100% 89.5M/89.5M [00:01<00:00, 53.7MB/s]
```

One thing in the hyperparamters above which needs special consideration is `mu`. We use this hyperparamter to control the ratio of labeled training samples in a mini-batch. Given that the number of labeled training samples is B, we have:

$$\text{number of unlabeled samples in the mini-batch} = \mu B$$

Defining Data Augmentations

Data augmentation is the trick of this method to work. Weak data augmentation refers to slight changes of the input image, and strong augmentation refers to changes in the image that are more likely to change the label. That being said, there is no clear definition of either augmentation strategies and some, like [2], propose learning data augmentation strategies from the data set itself.

It is easier to maintain, if we have different types of augmentations in different classes. Therefore, we develop separate classes for each type of augmentation. Then we put them all together in the `Augmentations` class.

Weak Data Augmentation

Here is how the FixMatch's weak augmentation works, which employs a standard flip-and-shift augmentation strategy. Specifically, we randomly flip images horizontally with a probability of 50% and we randomly translate images by up to 12.5% vertically and horizontally.

```
def image_to_numpy(img):
    img = np.array(img, dtype=np.float32)
    img = img / 255.   # Normalization is done in the ResNet
    return img

class WeakAugmentation:
    def __init__(self):
        self.transform = torchvision.transforms.Compose(
            [
                torchvision.transforms.RandomHorizontalFlip(),
                torchvision.transforms.RandomCrop(size=32, padding=int(32*0.125), padding_mode='constant'),
                image_to_numpy,
            ]
        )

    def __call__(self, x):
        return self.transform(x)
```

Strong Data Augmentation

For strong augmentation, we follow the instructions, as described in the paper

```
def cutout_func(img, pad_size=int(32 * 0.5), replace=(0.5, 0.5, 0.5)):
    replace = np.array(replace)
    h, w = img.shape[0], img.shape[1]

    x = np.random.randint(h)
    y = np.random.randint(w)

    size_factor = 4

    x1 = np.clip(x - pad_size // size_factor, 0, h)
    x2 = np.clip(x + pad_size // size_factor, 0, h)
    y1 = np.clip(y - pad_size // size_factor, 0, w)
    y2 = np.clip(y + pad_size // size_factor, 0, w)

    out = img.copy()

    out[x1:x2, y1:y2, :] = replace
```

```
    return out

# According to the original paper, the strong augmentation refers to:
# 1. RandAugment √ / CTAugment (we choose RandAugment since the class RANDAUGMENT is already implemented in torch.transforms)
# 2. Cutout

# Note:
# In the original paper of randaugment, the magnitude is fixed to control the severity of all distortions.
# However, fixmatch paper found that sampling a random magnitude from a pre-defined range at each training step works better for semi-supervised learning.
# In our implement practical, to make it simple and easy to understand, we just use the fixed global magnitude.
class StrongAugmentation:
    def __init__(self):
        self.transform = torchvision.transforms.Compose(
            [
                torchvision.transforms.RandAugment(num_ops=2, magnitude=9),
                image_to_numpy,
                cutout_func,
            ]
        )

    def __call__(self, x):
        return self.transform(x)
```

Put The Augmentations Together

```
class UnlabeledDataAug:
    def __init__(self):
        self.weak_aug = WeakAugmentation()
        self.strong_aug = StrongAugmentation()

    def __call__(self, x, training=False):
        return self.weak_aug(x), self.strong_aug(x)

class LabeledDataAug:
    def __init__(self):
        self.weak_aug = WeakAugmentation()

    def __call__(self, x, training=False):
        return self.weak_aug(x)
```

As you may have noticed, for an unlabeled input image we return two outputs in the `__call__` method. It simplifies our training loop, where we need two types of augmentation.

Loading the Datasets

select dataset

```
#@title select dataset
train_dataset = torchvision.datasets.CIFAR10(
    './cifar10',
    download=True,
    train=True
)
val_dataset = torchvision.datasets.CIFAR10(
    './cifar10',
    download=True,
    train=False
)
```

```
100%|██████████| 170M/170M [00:05<00:00, 32.0MB/s]
```

We do stratified sampling to create a "small" training data set. Note that unlabeled dataset also contains labeled samples. At this time we also set augmentation properties of the datasets.

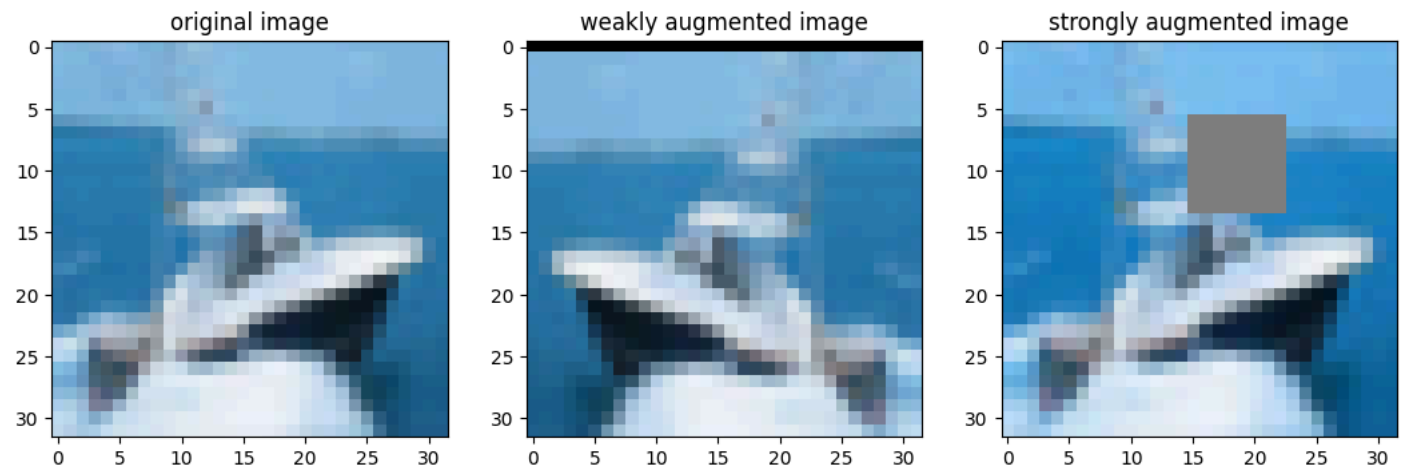
```
mu = 7
num_labeled_training_samples_per_class = 400
labeled_batch_size = 64
val_batch_size = 128
unlabeled_batch_size = mu*labeled_batch_size

labeled_dataset, _ = stratified_split(
    general_key,
    train_dataset,
    num_labeled_training_samples_per_class)
unlabeled_dataset = copy.deepcopy(
    train_dataset)

unlabeled_dataset.transform = UnlabeledDataAug()
labeled_dataset.transform = LabeledDataAug()
val_dataset.transform = torchvision.transforms.Compose([
    image_to_numpy
    ])

oversampled_size = (labeled_batch_size*len(
    unlabeled_dataset))/unlabeled_batch_size
oversampled_size = int(jnp.ceil(oversampled_size))
```

Get to Know Our Datasets

visualize samples

```
#@title visualize samples
demo_idx=8
demo_x = image_to_numpy(train_dataset[demo_idx][0])

print("Unlabeled Dataset:\n",
      unlabeled_dataset)
(x_weak,x_strong), y = unlabeled_dataset[demo_idx]
print("Unlabeled class index y: ", y, "\n")

images = {
    "original image":demo_x,
    "weakly augmented image":x_weak,
    "strongly augmented image":x_strong}
fig, axes = plt.subplots(1,3,figsize=(13,4))
for i,(ttl,img) in enumerate(images.items()):
    axes[i].imshow(img)
    axes[i].set_title(ttl)
```

```
Unlabeled Dataset:
 Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ./cifar10
    Split: Train
Unlabeled class index y:  8
```



In order to use jax data loader, we need to use a class already provided in the jax documentation website called `NumpyLoader`. We also use `get_training_generator` and `get_validation_generator` as our utility functions in the training and validation loop.

```
def numpy_collate(batch):
  if isinstance(batch[0], np.ndarray):
    return np.stack(batch)
  elif isinstance(batch[0], (tuple,list)):
    transposed = zip(*batch)
    return [numpy_collate(samples)
        for samples in transposed]
  else:
    return np.array(batch)

class NumpyLoader(data.DataLoader):
  def __init__(self, dataset, batch_size=1,
               shuffle=False, sampler=None,
               batch_sampler=None, num_workers=0,
               pin_memory=False, drop_last=False,
               timeout=0, worker_init_fn=None):
    super(self.__class__, self).__init__(dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        sampler=sampler,
        batch_sampler=batch_sampler,
        num_workers=num_workers,
        collate_fn=numpy_collate,
        pin_memory=pin_memory,
        drop_last=drop_last,
        timeout=timeout,
        worker_init_fn=worker_init_fn)

def get_training_generator(labeled_dataset,unlabeled_dataset):
    labeled_training_generator = NumpyLoader(
        labeled_dataset,
        batch_size=labeled_batch_size,
        num_workers=0,
        sampler=RandomSampler(labeled_dataset,
                              replacement=True,
                              num_samples=oversampled_size))
    unlabled_training_generator = NumpyLoader(
        unlabeled_dataset,
        batch_size=unlabeled_batch_size,
        num_workers=0)
    return zip(labeled_training_generator,
               unlabled_training_generator)

def get_validation_generator(val_dataset):
    val_generator = NumpyLoader(
        val_dataset,
        batch_size=val_batch_size,
        num_workers=0)
    return val_generator
```

Training Utility Functions

```
# This class helps us keep track of our training metrics
class Metrics:
    def __init__(self):
        self.dict_ = {}

    def merge(self, dict_other):
        for k,v in dict_other.items():
            if k in self.dict_:
                self.dict_[k].append(v)
            else:
                self.dict_[k] = [v]

    def __getitem__(self,k):
        return np.array(self.dict_[k])

    def values(self):
        return self.dict_.values()

    def keys(self):
        return self.dict_.keys()

    def items(self):
        return self.dict_.items()

# to convert the dictionary values to numpy values instead of jax.numpy
def convert_to_numpy(jnp_dict):
    return {k:np.array(v) for k,v in jnp_dict.items()}

# to convert images from [0,1] range to [0,255].
def jnp_to_np_u8(jnp_img):
    return (np.array(jnp_img)*255).astype(np.uint8)
```

Some of the following functions need to be completed by you:

```
    @jax.jit
    def log_softmax_cross_entropy(
        log_probs,
        targets):
        """
        ==============================
        TODO: Implementation required.
        ==============================
        1. Return the cross entropy loss assuming you are
           given the log probabilities and the one-hot encoded targets
        """

        cross_entropy = -jnp.sum(targets * log_probs, axis=-1)
        return cross_entropy

    @jax.jit
    def logits_to_log_probs(logits):
        """
        ==============================
        TODO: Implementation required.
        ==============================
        1. Return the log probabilities assuming you are
           given the logits
        """

        log_probs = jax.nn.log_softmax(logits, axis=-1)
        return log_probs
```

Test `log_prob_cross_entropy` and `logits_to_log_probs` together.

```
    torch_logits = torch.tensor([[1.,.2,2.,3.],
                                 [2.,2.,3.,.3],
                                 [4.,.1,2.,3.]])
    torch_labels = torch.tensor([1,2,3])
    log_probs_expected = jnp.array(
        [[-2.4472625 , -3.2472625 , -1.4472625 , -0.44726253],
         [-1.5894322 , -1.5894322 , -0.58943224, -3.2894323 ],
         [-0.42098188, -4.320982  , -2.420982  , -1.4209819 ]])

    logits = jnp.array(torch_logits)
    log_probs = logits_to_log_probs(logits)

    np.testing.assert_array_almost_equal(log_probs,
                                         log_probs_expected,
                                         decimal=4)

    loss = torch.nn.CrossEntropyLoss()
    labels = jnp.array(torch_labels)
    expected = jnp.array(loss(torch_logits,
                              torch_labels))
    dense_labels = jax.nn.one_hot(labels,torch_logits.shape[1])
    returned = log_softmax_cross_entropy(log_probs,
                                       dense_labels).mean()

    np.testing.assert_approx_equal(returned,
                                   expected,
                                   significant=4)
```

You might be unfamiliar with writing a function inside another function, but this is how we can write a cleaner code in jax! The idea is that you are allowed to use inner and outer function parameters.

```
>>> def outer_fn(a,b):
        # ...
        # some static computations over a and b
        # ...
        k = a+b
        def inner_fn(c,d):
            # regard a and b as static parameters
            return c+d-k
        return inner_fn
>>> fn_k_is_3 = outer_fn(1,2) # returns a function
>>> fn_k_is_5 = outer_fn(3,2) # returns a function
>>> fn_k_is_3(5,6)
8 # 5 + 6 - 3
>>> fn_k_is_5(5,6)
6 # 5 + 6 - 5
```

Based on Fixmatch paper, we need the implementation of cosine decay schedule:

```
    def cosine_decay_schedule(
        init_value,
        decay_steps,
        alpha):

        @jax.jit
        def schedule(count):
            """
            ==============================
            TODO: Implementation required.
            ==============================
            1. consider alpha = 7/16, init_value = eta and decay_steps = K
            2. Return the decayed learning rate calculated based on the paper.
            """

            progress = jnp.pi * alpha * count / decay_steps
            decayed_lr = init_value * jnp.cos(progress)
            return decayed_lr

        return schedule
```

Test your implementation for `cosine_decay_schedule` function.

```
    lr_schedule=cosine_decay_schedule(3e-2,
        2**20,
        7/16)
    expected = jnp.array(0.02955833)
```

```
    np.testing.assert_approx_equal(lr_schedule(2**17),
                                   expected,
                                   significant=4)
```

We still need a few more utility functions to run the training loop.

Here is a function that gives the forward pass function:

```
    def get_forward_fn(model):
        @jax.jit
        def forward(
            params,
            batch_stats,
            x_labeled,
            x_unlabeled_weak,
            x_unlabeled_strong):

            model_params = {'params':params,
                            'batch_stats':batch_stats}

            """
            ==============================
            TODO: Implementation required.
            ==============================
            1. concatenate all input vectors together for a faster forward pass
            2. do a forward pass while mutable keyword argument of
               your model set to ['batch_stats']
            3. then your model returns the tuple: y,batch_stats
            4. split the output of your model to get three outputs
            """

            #raise NotImplementedError("Task: Implement!")
            x = jnp.concatenate([x_labeled,
                                 x_unlabeled_weak,
                                 x_unlabeled_strong],
                                axis=0)

            y, batch_stats = model.apply(model_params,
                                         x,
                                         mutable=['batch_stats'])

            # this is how we break a long line into several lines:
            y_labeled_pred,        y_unlabeled_pred_weak,        y_unlabeled_pred_strong =         jnp.split(
                y,
                [x_labeled.shape[0],
                 x_labeled.shape[0] + x_unlabeled_weak.shape[0]],
                axis=0)



            return y_labeled_pred,\
                   y_unlabeled_pred_weak,\
                   y_unlabeled_pred_strong,\
                   batch_stats['batch_stats']
        return forward
```

Test your implementation in `get_forward_fn`:

```
    class MockModel:
        def apply(self,
                  model_params,
                  x,
                  *,
                  mutable):
            assert type(mutable) == list
            assert 'batch_stats' in mutable

            golden = np.linspace(0,
                                 1,
                                 x.shape[0])
            golden = jnp.expand_dims(golden,1)
            mock_output = golden * model_params['batch_stats'] - model_params['params']
            mock_output=mock_output*x.mean()*jnp.ones(
                shape=(x.shape[0],
                       2))
            return mock_output,{'batch_stats':jnp.array(0)}

    y1,y2,y3,y4 = get_forward_fn(MockModel())(
        jnp.array(-1),
        jnp.array(1),
        jnp.ones(shape=(1,32,32,3))*.5,
        jnp.ones(shape=(2,32,32,3))*.8,
        jnp.ones(shape=(2,32,32,3))*.3)

    np.testing.assert_array_almost_equal(y1,jnp.array([[0.5400001,
                                                        0.5400001]]),
                                         decimal=4)
    np.testing.assert_array_almost_equal(y2,
                                         jnp.array([[0.6750001, 0.6750001],
                                                    [0.8100001, 0.8100001]]),
                                         decimal=4)
    np.testing.assert_array_almost_equal(y3,
                                         jnp.array([[0.9450002, 0.9450002],
                                                    [1.0800002, 1.0800002]]),
                                         decimal=4)
    np.testing.assert_array_almost_equal(y4,
                                         jnp.array(0),
                                         decimal=4)
```

Here is the heart of our training loop, `fixmatch_loss`, which takes care of using unlabeled data for training.

```
    def get_fixmatch_loss_fn(forward,
                             hparams):
        log_confidence_threshold = jnp.log(hparams.confidence_threshold)
        #l2_reg = L2(hparams.w_decay)

        def l2_reg(params):
            leaves = jax.tree_util.tree_leaves(params)
            penalty = sum(jnp.sum(jnp.square(p)) for p in leaves)
            return hparams.w_decay * penalty

        @jax.jit
        def fixmatch_loss(
            params,
            batch_stats,
```

```
    x_labeled,
    x_unlabeled_weak,
    x_unlabeled_strong,
    y_labeled,
    y_unlabeled):

    """
    ============================
    TODO: Implementation required.
    ============================
    1. compute each step of the function
    2. Hint: note that error rate is computed only over y_unlabeled_pred_weak
    3. Hint: to stop gradient use jax.lax.stop_gradient on a vector
    4. Hint: note that total_loss is a scalar
    """

    #compute the forward pass
    y_labeled_pred,        y_unlabeled_pred_weak,        y_unlabeled_pred_strong,        batch_stats = forward(
        params,
        batch_stats,
        x_labeled,
        x_unlabeled_weak,
        x_unlabeled_strong)

    # stop gradients for pseudo labels
    y_unlabeled_pred_weak = jax.lax.stop_gradient(
        y_unlabeled_pred_weak)

    # computing log probs from logits
    y_labeled_log_prob = logits_to_log_probs(y_labeled_pred)
    y_unlabeled_weak_log_prob = logits_to_log_probs(
        y_unlabeled_pred_weak)
    y_unlabeled_strong_log_prob = logits_to_log_probs(
        y_unlabeled_pred_strong)

    # create one-hot labels from sparse representations
    y_labeled_oh = jax.nn.one_hot(y_labeled,
                                  hparams.num_classes)
    y_unlabeled_pred_weak_sparse = jnp.argmax(
        y_unlabeled_pred_weak,
        axis=-1)
    y_unlabeled_pred_weak_oh = jax.nn.one_hot(
        y_unlabeled_pred_weak_sparse,
        hparams.num_classes)

    # supervised loss
    s_loss = log_softmax_cross_entropy(
        y_labeled_log_prob,
        y_labeled_oh).mean()

    # unsupervised loss
    u_loss = log_softmax_cross_entropy(
        y_unlabeled_strong_log_prob,
        y_unlabeled_pred_weak_oh)

    # compute error rate based on the paper
    # good to know that it is 1 - accuracy
    error_rate = jnp.not_equal(
        y_unlabeled_pred_weak_sparse,
        y_unlabeled).mean()

    # compute the mask for low-confidence preds
    # you can directly use log_confidence_threshold
    mask_y = (jnp.max(
        y_unlabeled_weak_log_prob,
        axis=-1) >= log_confidence_threshold).astype(jnp.float32)

    # mask the low confidence predictions and average losses
    masked_u_loss = (u_loss * mask_y).mean()

    # compute the weight decay using l2_reg
    w_d = l2_reg(params)

    # compute the total loss based on the paper
    total_loss = s_loss + hparams.lambda_coef * masked_u_loss + w_d

    return total_loss,(batch_stats,{
        "weight decay": w_d,
        "total loss": total_loss,
        "supervised loss":s_loss,
        "unsupervised loss":masked_u_loss,
        "mask rate":mask_y.mean(),
        "error rate":error_rate
        })
    return fixmatch_loss
```

Test your implementation for `get_fixmatch_loss_fn`:

```
class MockHparams:
    def __init__(self):
        self.confidence_threshold = 0.28
        self.lambda_coef = 5
        self.w_decay = 5
        self.num_classes = 2

out_total_loss,(out_batch_stats,out_dict) = get_fixmatch_loss_fn(
    get_forward_fn(MockModel()),
    MockHparams())(
        jnp.array(-8),jnp.array(9),
        jnp.ones(shape=(2,32,32,3))*.8,
        jnp.ones(shape=(3,32,32,3))*.5,
        jnp.ones(shape=(3,32,32,3))*.3,
        jnp.array([1,0]),
        jnp.array([0,1,1]))

expected_total_loss,(expected_batch_stats,expected_dict) = (jnp.array(324.1588745),
    (jnp.array(0),
     {'error rate': jnp.array(0.6666667),
      'mask rate': jnp.array(1.),
      'supervised loss': jnp.array(0.6931472),
      'total loss': jnp.array(324.1588745),
      'unsupervised loss': jnp.array(0.6931470),
      'weight decay': jnp.array(320.)}))

np.testing.assert_approx_equal(out_batch_stats,
                               expected_batch_stats,
                               significant=4)
```

```
    np.testing.assert_approx_equal(out_dict['error rate'],
                                   expected_dict['error rate'],
                                   significant=4)
    np.testing.assert_approx_equal(out_dict['mask rate'],
                                   expected_dict['mask rate'],
                                   significant=4)
    np.testing.assert_approx_equal(out_dict['unsupervised loss'],
                                   expected_dict['unsupervised loss'],
                                   significant=4)
    np.testing.assert_approx_equal(out_dict['supervised loss'],
                                   expected_dict['supervised loss'],
                                   significant=4)
    np.testing.assert_approx_equal(out_dict['weight decay'],
                                   expected_dict['weight decay'],
                                   significant=4)
    np.testing.assert_approx_equal(out_dict['total loss'],
                                   expected_dict['total loss'],
                                   significant=4)
```

The following function is easier to implement! This function computes the error rate on the validation dataset.

```
    def get_validate_fn(get_params,model):
        @jax.jit
        def validate(opt_state,
                     batch_stats):
            params = get_params(opt_state)
            model_params = {'params':params,
                            'batch_stats':batch_stats}
            validation_generator = get_validation_generator(val_dataset)

            """
            ============================
            TODO: Implementation required.
            ============================
            1. iterate over validation_generator and compute error rate
               in each mini-batch
            2. average error rates over all mini-batches
            """


            error_rates = []
            for x_val, y_val in validation_generator:
                logits = model.apply(model_params,
                                     x_val)
                preds = jnp.argmax(logits,axis=-1)
                error_rates.append(
                    jnp.not_equal(preds,y_val).mean())
            error_rate = jnp.stack(error_rates).mean()

            return {"val error rate":error_rate}
        return validate
```

Test your implementation for `get_validate_fn`:

```
    class ValMockModel:
        def apply(self,
                  model_params,
                  x):
            assert 'batch_stats' in model_params
            assert 'params' in model_params
            temp = jnp.floor((jnp.linspace(0,x.shape[0],10)
            * model_params['params'] *
             model_params['batch_stats'])%10)
            temp = jnp.expand_dims(temp,0)
            return jnp.ones(shape=(x.shape[0],1))@temp

    class MockGetParams:
        def __call__(self,
                     opt_state):
            return opt_state * jnp.array(.5)

    out_dict = get_validate_fn(MockGetParams(),ValMockModel())(
        jnp.array(1.2),
        jnp.array(0.9))

    np.testing.assert_approx_equal(out_dict['val error rate'],
                                   jnp.array(0.9011076),
                                   significant=4)
```

The final function is provided, no implementation is required.

```
    def get_train_step_fn(get_params,
                          fixmatch_loss,
                          opt_update):
        @jax.jit
        def train_step(step,
                       opt_state,
                       batch_stats,
                       x_labeled,
                       x_unlabeled_weak,
                       x_unlabeled_strong,
                       y_labeled,
                       y_unlabeled):
            params = get_params(opt_state)

            (loss, (batch_stats, metrics)), grads = jax.value_and_grad(
                fixmatch_loss,
                has_aux=True)(
                    params,
                    batch_stats,
                    x_labeled,
                    x_unlabeled_weak,
                    x_unlabeled_strong,
                    y_labeled,
                    y_unlabeled)

            opt_state = opt_update(step,
                                   grads,
                                   opt_state)

            return loss, batch_stats, opt_state, metrics
        return train_step
```

Training Loop

Given that you've passed all the tests, the training loop should show the effect of using `fixmatch_loss` for training.

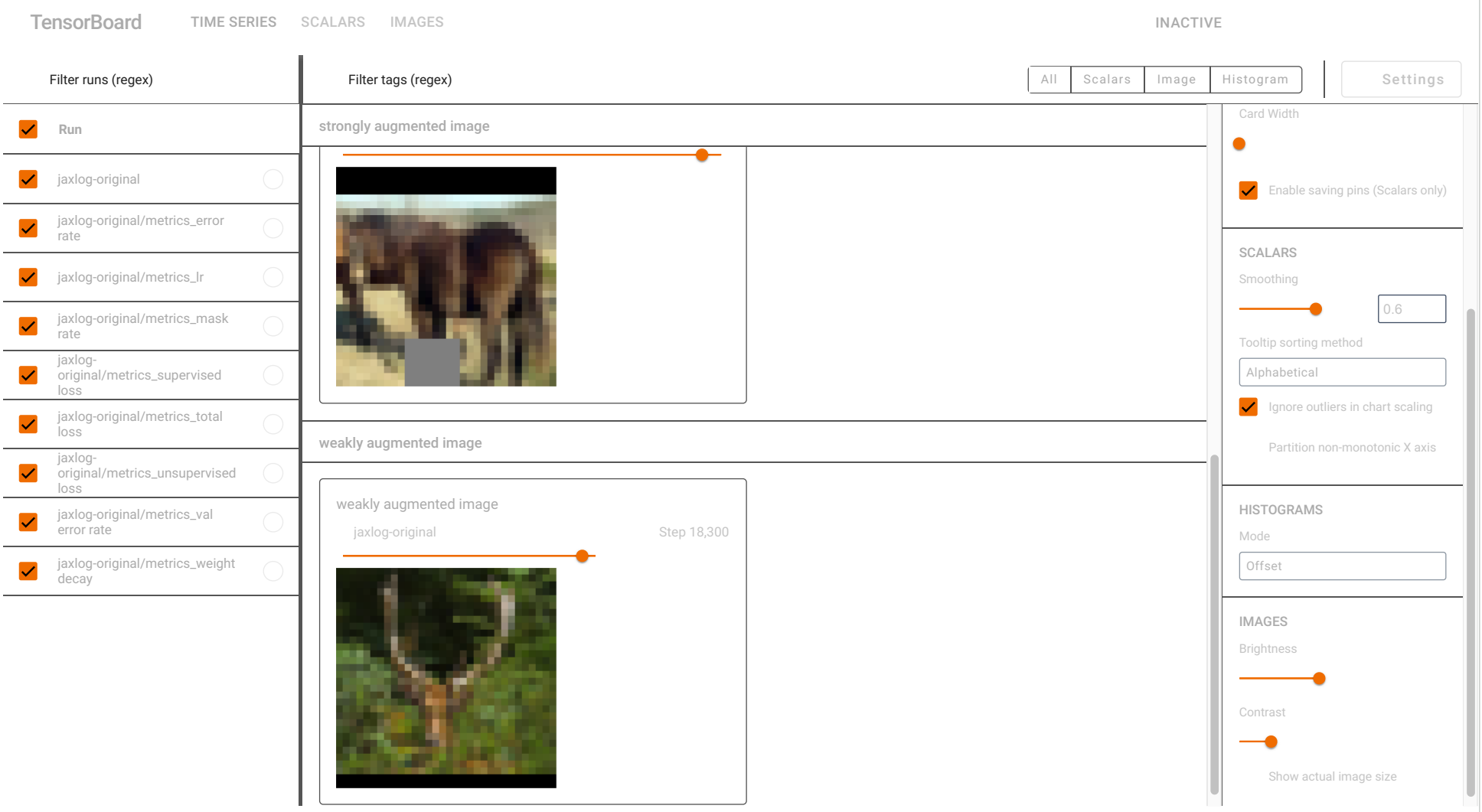
```
    def train_fixmatch(hparams,
                       labeled_dataset,
                       unlabeled_dataset,
                       val_dataset,
                       checkpoint=None):
        init_total_step = hparams.init_total_step if\
            hasattr(hparams,'init_total_step') else 0

        writer = SummaryWriter(hparams.logdir)

        # Initialize: optimizers
        lr_scheduler = cosine_decay_schedule(
            hparams.lr,
            decay_steps=hparams.total_steps,
            alpha=hparams.alpha)
        opt_init, opt_update, get_params = optimizers.momentum(
            step_size=lr_scheduler,
            mass=hparams.beta)

        # loading the model architecture
        model = jax_resnet.ResNet18(n_classes=hparams.num_classes)

        if checkpoint is None:
            # loading a pretrained checkoint
            ResNet18, params = jax_resnet.pretrained_resnet(size=18)

            params = unfreeze(params)
            key = jax.random.PRNGKey(hparams.seed)
            dense_params = model.layers[-1].init(key,
            inputs=jnp.zeros(shape=(1,512)))

            # reinitializing the parameters of the final layer to fit the architecture
            params['params']['layers_11']['kernel'] = dense_params['params']['kernel']
            params['params']['layers_11']['bias'] = dense_params['params']['bias']

            # separate batchnorm from other parameters
            params = freeze(params)
            batch_stats = params['batch_stats']
            params = params['params']

            opt_state = opt_init(params)
        else:
            print("parameters loaded from the provided checkpoint.")
            batch_stats = checkpoint['batch_stats']
            opt_state = optimizers.pack_optimizer_state(checkpoint['opt_state'])



        forward = get_forward_fn(model)
        fixmatch_loss = get_fixmatch_loss_fn(forward,
                                             hparams)
        validate = get_validate_fn(get_params,model)
        train_step = get_train_step_fn(get_params,
                                       fixmatch_loss,
                                       opt_update)
        # Training loop
        metrics = Metrics()
        total_step = init_total_step
        for epoch in range(hparams.epochs):
            for (x_l,y_l),((x_w,x_s),y_u) in get_training_generator(labeled_dataset,
                                                                    unlabeled_dataset):
                loss, batch_stats, opt_state, step_metrics = train_step(
                    step=total_step,
                    opt_state=opt_state,
                    batch_stats=batch_stats,
                    x_labeled = x_l,
                    x_unlabeled_weak = x_w,
                    x_unlabeled_strong = x_s,
                    y_labeled = y_l,
                    y_unlabeled = y_u
                )
                total_step += 1
                step_metrics.update(
                    {"lr":lr_scheduler(total_step)})
            metrics.merge(step_metrics)
            writer.add_scalars('metrics',
                               convert_to_numpy(step_metrics),
                               global_step=total_step)

            if not total_step % hparams.write_checkpoint_every:
                opt_state_ckpt = optimizers.unpack_optimizer_state(opt_state)
                checkpoint = {"opt_state":opt_state_ckpt,
                              "batch_stats":batch_stats,
                              "total_step":total_step}
                jnp.save(f'checkpoints/checkpoint-{total_step}.npy',
                            checkpoint)

            if not total_step % hparams.validate_every_steps:
                val_metrics = validate(
                    opt_state,
                    batch_stats)
                writer.add_scalars('metrics',
                                   convert_to_numpy(val_metrics),
                                   global_step=total_step)

            if not total_step % hparams.log_every_steps:
                np_x_w =jnp_to_np_u8(x_w[0,...])
                np_x_s =jnp_to_np_u8(x_s[0,...])
                writer.add_image('weakly augmented image',
                            np_x_w,
                            global_step=total_step,
                            dataformats='HWC')
                writer.add_image('strongly augmented image',
                            np_x_s,
                            global_step=total_step,
                            dataformats='HWC')
                print('epoch', f"{epoch:3d} |",
                    'total step',f"{total_step:4d}",end=" | ")
                for k,v in metrics.items():
                    print(k,f"{v[-1]:.3f}",end=" | ")
                print("")
            writer.flush()
```

```
    opt_state = optimizers.unpack_optimizer_state(opt_state)
    return {"opt_state":opt_state,
            "batch_stats":batch_stats,
            "total_step":total_step}
```

It turns out that you need to train the network for quite a long time to see the results. Fortunately, we trained a network for 20k steps, without unsupervised loss i.e. $\lambda = 0$, and saved some metrics via `tensorboard`. It is available for you to investigate and compare your results with a network that does not utilize unlabeled images.

For a continual update of your metrics click on the ⚙ icon on the top right of the tensorboard, and tick "Reload data". Also you need to set full-screen of `metrics` window enabled by clicking on the left-most icon under the window.

```
%tensorboard --logdir='jaxlog-base'
```



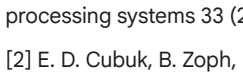
In case that you are utilizing GPU, training for 10 epochs may take around 20 mins on the Google Colab servers.

```
try:
    checkpoint = jnp.load('checkpoints/checkpoint-20000.npy',
                          allow_pickle=True).item()
    init_total_step = checkpoint['total_step']+1
    checkpoint = {"opt_state":checkpoint['opt_state'],
                  "batch_stats":checkpoint['batch_stats']}
except FileNotFoundError:
    raise "comment this line if you want to initialize from"\
          " the pretrained ResNet18 checkpoint. \nNote that"\
          " you need to load the 20k checkpoint for this practical."
    checkpoint=None
    init_total_step=0

hparams = OmegaConf.create({
    'seed':1,
    'epochs': 10,
    "init_total_step":init_total_step,
    "total_steps":2**20,
    'num_classes': 10,
    'lr':3e-2,
    'beta':0.9,
    'alpha':7/16,
    "log_every_steps":50,
    "confidence_threshold":.95,
    "lambda_coef" : 1,
    "w_decay":5e-6,
    "logdir" : 'jaxlog-base',
    "validate_every_steps":100,
    "write_checkpoint_every":1000,
})
checkpoint = train_fixmatch(hparams,
                            labeled_dataset,
                            unlabeled_dataset,
                            val_dataset,
                            checkpoint=checkpoint)
```

```
parameters loaded from the provided checkpoint.
epoch   0 | total step 20050 | error rate 0.366 | mask rate 0.196 | supervised loss 0.799 | total loss 0.970 | unsupervised loss 0.108 | weight decay 0.063 | lr 0.030 |
epoch   0 | total step 20100 | error rate 0.348 | mask rate 0.221 | supervised loss 0.730 | total loss 0.921 | unsupervised loss 0.128 | weight decay 0.063 | lr 0.030 |
epoch   1 | total step 20150 | error rate 0.350 | mask rate 0.261 | supervised loss 0.991 | total loss 1.113 | unsupervised loss 0.059 | weight decay 0.063 | lr 0.030 |
epoch   1 | total step 20200 | error rate 0.321 | mask rate 0.254 | supervised loss 0.522 | total loss 0.661 | unsupervised loss 0.076 | weight decay 0.063 | lr 0.030 |
epoch   2 | total step 20250 | error rate 0.308 | mask rate 0.281 | supervised loss 0.595 | total loss 0.765 | unsupervised loss 0.107 | weight decay 0.063 | lr 0.030 |
epoch   2 | total step 20300 | error rate 0.333 | mask rate 0.319 | supervised loss 0.598 | total loss 0.849 | unsupervised loss 0.188 | weight decay 0.063 | lr 0.030 |
epoch   3 | total step 20350 | error rate 0.337 | mask rate 0.312 | supervised loss 0.426 | total loss 0.579 | unsupervised loss 0.090 | weight decay 0.063 | lr 0.030 |
epoch   3 | total step 20400 | error rate 0.292 | mask rate 0.301 | supervised loss 0.499 | total loss 0.675 | unsupervised loss 0.112 | weight decay 0.063 | lr 0.030 |
epoch   4 | total step 20450 | error rate 0.348 | mask rate 0.348 | supervised loss 0.583 | total loss 0.837 | unsupervised loss 0.191 | weight decay 0.064 | lr 0.030 |
epoch   4 | total step 20500 | error rate 0.344 | mask rate 0.357 | supervised loss 0.447 | total loss 0.691 | unsupervised loss 0.180 | weight decay 0.064 | lr 0.030 |
epoch   4 | total step 20550 | error rate 0.333 | mask rate 0.364 | supervised loss 0.313 | total loss 0.513 | unsupervised loss 0.137 | weight decay 0.064 | lr 0.030 |
epoch   5 | total step 20600 | error rate 0.333 | mask rate 0.408 | supervised loss 0.263 | total loss 0.517 | unsupervised loss 0.190 | weight decay 0.064 | lr 0.030 |
epoch   5 | total step 20650 | error rate 0.362 | mask rate 0.422 | supervised loss 0.308 | total loss 0.582 | unsupervised loss 0.211 | weight decay 0.064 | lr 0.030 |
epoch   6 | total step 20700 | error rate 0.297 | mask rate 0.431 | supervised loss 0.405 | total loss 0.699 | unsupervised loss 0.230 | weight decay 0.064 | lr 0.030 |
epoch   6 | total step 20750 | error rate 0.301 | mask rate 0.467 | supervised loss 0.511 | total loss 0.814 | unsupervised loss 0.239 | weight decay 0.064 | lr 0.030 |
epoch   7 | total step 20800 | error rate 0.339 | mask rate 0.462 | supervised loss 0.239 | total loss 0.486 | unsupervised loss 0.183 | weight decay 0.064 | lr 0.030 |
epoch   7 | total step 20850 | error rate 0.319 | mask rate 0.460 | supervised loss 0.329 | total loss 0.599 | unsupervised loss 0.206 | weight decay 0.064 | lr 0.030 |
epoch   8 | total step 20900 | error rate 0.292 | mask rate 0.462 | supervised loss 0.092 | total loss 0.368 | unsupervised loss 0.212 | weight decay 0.064 | lr 0.030 |
epoch   8 | total step 20950 | error rate 0.355 | mask rate 0.442 | supervised loss 0.390 | total loss 0.636 | unsupervised loss 0.182 | weight decay 0.064 | lr 0.030 |
epoch   8 | total step 21000 | error rate 0.339 | mask rate 0.480 | supervised loss 0.250 | total loss 0.571 | unsupervised loss 0.256 | weight decay 0.064 | lr 0.030 |
```

```
epoch   9 | total step 21050 | error rate 0.317 | mask rate 0.458 | supervised loss 0.110 | total loss 0.333 | unsupervised loss 0.159 | weight decay 0.064 | lr 0.030 |
epoch   9 | total step 21100 | error rate 0.286 | mask rate 0.473 | supervised loss 0.166 | total loss 0.498 | unsupervised loss 0.267 | weight decay 0.064 | lr 0.030 |
```

After training, you can reload the tensorboard to see the improvement achieved using the fixmatch loss! To this end, you should have only `jaxlog-original/metrics_val error rate` and `metrics_val error rate` checked. Assuming that everything done correctly, you should see the following progress:



In the above image, blue curve is the validation error rate when training a model only on the labeled examples without unsupervised loss. The light blue curve is the the validation error rate while using the unsupervised loss.

Question 4: Think about at least two factors that will introduce randomness to the fixmatch training results. (Hint: dataloader / training hyperparameters) Illustrate how these factors can introduce randomness.

Answer:

Finally, it is valuable for us to know how much did it take you to finish this practical?

```
Start coding or generate with AI.
```

Optional Open Questions

- The unsupervised loss is computing the mean over all samples in the mini-batch i.e. the $\frac{1}{\mu B}$ coefficient for computing the $\square_u$ includes even masked samples. However, intuitively, it makes sense to exclude masked samples while computing the mean! In this case, we will have something like:
$$\square_u = \frac{1}{|\square|} \sum_{x \in \square} \mathbb{H}(p_x, q_x)$$
do you think this will help getting better validation accuracy?
- If you look at the supervised loss, it is almost near zero. What change do you think we can incorporate to the code to make it harder for model to memorize labeled data?
- As it is a demonstration of the Fixmatch idea, we set the number of epochs to 10. Can you train it for a longer time (or make some changes to the code) to get a better validation accuracy?

References

[1] Sohn, Kihyuk, et al. "Fixmatch: Simplifying semi-supervised learning with consistency and confidence." Advances in neural information processing systems 33 (2020): 596-608.

[2] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation strategies from data. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2019. 4, 9, 20

[3] Cubuk, Ekin D., et al. "Randaugment: Practical automated data augmentation with a reduced search space." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops. 2020.

[4] https://pytorch.org/vision/main/generated/torchvision.transforms.RandAugment.html#torchvision.transforms.RandAugment

[5] DeVries, Terrance, and Graham W. Taylor. "Improved regularization of convolutional neural networks with cutout." arXiv preprint arXiv:1708.04552 (2017).

[6] https://github.com/uoguelph-mlrg/Cutout/blob/master/util/cutout.py

[7] https://github.com/CoinCheung/fixmatch-pytorch

[8] https://pytorch.org/vision/stable/transforms.html