EPASS with SimMatch Base for Freesound Audio Tagging

This notebook implements the **EPASS (Ensemble Projectors Aided for Semi-supervised Learning)** algorithm, using **SimMatch** as the base semi-supervised framework, for audio classification on the Freesound dataset (2018).

Core Concepts:

1. **SimMatch Base:** Leverages both pseudo-labeling (like FixMatch) and instance similarity matching (contrastive learning) using two strongly augmented views of unlabeled data.
2. **EPASS Enhancement:** Instead of a single MLP projector head (mapping encoder features to embeddings for contrastive loss), EPASS uses *multiple* projector heads. The embeddings from these heads are ensembled (averaged) to produce a more robust and less biased representation.
3. **Goal:** Train models with 20% and 80% labeled data, aiming for high accuracy, demonstrating overfitting/underfitting via plots, and saving the best overall model.

```
import os
import random
import numpy as np
import pandas as pd
import librosa
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.models as models
import torchaudio
import matplotlib.pyplot as plt
import seaborn as sns
from torch.utils.data import Dataset, DataLoader,
WeightedRandomSampler
from sklearn.model_selection import StratifiedShuffleSplit,
train_test_split
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from tqdm.notebook import tqdm
import itertools
import math
import copy
```

1. Configuration

```
class Config:
    def __init__(self):
```

```
        # Audio & Spectrogram Params
        self.sr = 32000           # Audio sample rate
        self.duration = 5         # Audio duration (seconds)
        self.n_mels = 128         # Number of Mel bands
        self.n_fft = 1024         # FFT size
        self.hop_length = 512     # Hop length

        # Training Params
        self.batch_size = 32      # Combined batch size (adjust per
GPU memory)
        self.epochs = 50          # Number of epochs (adjust as needed
for convergence/overfitting demo)
        self.lr = 3e-4            # Learning rate (Adam default often
works well)
        self.num_classes = 41     # Number of classes (as per
train.csv)
        self.device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
        self.seed = 42            # Random seed for reproducibility
        self.num_workers = 2      # Dataloader workers

        # Semi-Supervised Params (SimMatch + EPASS)
        self.labeled_percents = [0.2, 0.8] # Percentages of labeled
data to train with [20%, 80%]
        self.val_percent = 0.1    # Percentage of *original* training
data for validation
        self.mu = 7               # Ratio of unlabeled to labeled
samples per batch (unlabeled_bs = mu * labeled_bs)
        self.wu = 1.0             # Unsupervised classification loss
weight
        self.wc = 1.0             # Contrastive loss weight (SimMatch
component)
        self.threshold = 0.95     # Confidence threshold (tau) for
pseudo-labeling
        self.temperature = 0.1    # Temperature T for contrastive loss
(SimMatch component)
        self.embedding_dim = 128  # Dimension of the projected
embeddings
        self.num_projectors = 3   # Number of projectors for EPASS

        # SpecAugment Params (for strong augmentation)
        self.freq_mask_param = 27
        self.time_mask_param = 70 # Adjusted based on spectrogram
width

        # Model Saving
        self.model_save_path = "best_epass_simmatch_model.pth"

        # Data paths (update if necessary)
        self.train_csv_path =
```

```
"/kaggle/input/freesound-audio-tagging/train.csv"
        self.test_csv_path =
"/kaggle/input/freesound-audio-tagging/test_post_competition.csv"
        self.audio_train_dir = "/kaggle/input/freesound-audio-
tagging/audio_train"
        self.audio_test_dir  = "/kaggle/input/freesound-audio-
tagging/audio_test"

config = Config()

# Seed everything for reproducibility
random.seed(config.seed)
np.random.seed(config.seed)
torch.manual_seed(config.seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(config.seed)
    torch.cuda.manual_seed_all(config.seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

print(f"Device: {config.device}")
print(f"Number of projectors (EPASS): {config.num_projectors}")
```

```
Device: cuda
Number of projectors (EPASS): 3
```

2. Audio Preprocessing & Augmentation

```
# ------------------------------
# 2.1 Audio Preprocessing Function
# ------------------------------
def preprocess_audio(path, sr=config.sr, duration=config.duration,
n_mels=config.n_mels, n_fft=config.n_fft,
hop_length=config.hop_length):
    try:
        y, _ = librosa.load(path, sr=sr)
        max_len = sr * duration
        # Pad or truncate to fixed length
        if len(y) < max_len:
            y = np.pad(y, (0, max_len - len(y)))
        else:
            y = y[:max_len]
        # Compute mel spectrogram
        mel = librosa.feature.melspectrogram(y=y, sr=sr,
n_mels=n_mels, n_fft=n_fft, hop_length=hop_length)
        mel_db = librosa.power_to_db(mel, ref=np.max)
        # Normalize to [0, 1]
        mel_min = mel_db.min()
        mel_max = mel_db.max()
        if mel_max == mel_min: # Avoid division by zero for silent
```

```
clips
            return np.zeros_like(mel_db, dtype=np.float32)
        mel_norm = (mel_db - mel_min) / (mel_max - mel_min)
        return mel_norm.astype(np.float32)  # shape: (n_mels, time)
    except Exception as e:
        print(f"Error processing {path}: {e}")
        # Return a zero array or handle error appropriately
        time_steps = int(max_len / hop_length) + 1 # Approximate time
steps
        return np.zeros((n_mels, time_steps), dtype=np.float32)
```

2.2 Augmentation Functions

- **Weak Augmentation:** Identity (no change).
- **Strong Augmentation:** SpecAugment (frequency and time masking).

```
# ----------------------------
# 2.2 Augmentation Functions
# ----------------------------

# Weak augmentation (identity)
def weak_augment(mel_spec):
    # Ensure input is a tensor and add channel dim
    if not isinstance(mel_spec, torch.Tensor):
        mel_spec = torch.tensor(mel_spec)
    return mel_spec.unsqueeze(0)

# Strong augmentation (SpecAugment)
# Use try-except to handle different torchaudio versions
try:
    # Attempt initialization using likely older positional arguments
    print("Attempting SpecAugment with positional args: (n_freq_masks,
freq_mask_param, n_time_masks, time_mask_param)")
    spec_augment = torchaudio.transforms.SpecAugment(
        1,                          # n_freq_masks
        config.freq_mask_param,     # freq_masking_param
        1,                          # n_time_masks
        config.time_mask_param,     # time_masking_param
        iid_masks=True              # iid_masks might still be a
keyword arg
    )
    print("Successfully initialized SpecAugment with positional
args.")
except TypeError:
    try:
        # If positional fails, try the keyword arguments (likely newer
versions)
        print("Positional args failed. Attempting SpecAugment with
keyword args.")
        spec_augment = torchaudio.transforms.SpecAugment(
```

```
            freq_masking_param=config.freq_mask_param,
            time_masking_param=config.time_mask_param,
            freq_mask_count=1,
            time_mask_count=1,
            iid_masks=True
        )
        print("Successfully initialized SpecAugment with keyword
args.")
    except TypeError as e:
        # If both fail, fall back to Identity
        print(f"Failed to initialize SpecAugment with both positional
and keyword params. Error: {e}")
        print("Using Identity augmentation as a fallback for
strong_augment.")
        spec_augment = torch.nn.Identity()


def strong_augment(mel_spec):
     # Ensure input is a tensor and add channel dim
    if not isinstance(mel_spec, torch.Tensor):
        mel_spec = torch.tensor(mel_spec)
    mel_tensor = mel_spec.unsqueeze(0)
    augmented_mel = spec_augment(mel_tensor)
    return augmented_mel
```

```
Attempting SpecAugment with positional args: (n_freq_masks,
freq_mask_param, n_time_masks, time_mask_param)
Successfully initialized SpecAugment with positional args.
```

3. Dataset Classes

- Labeled dataset returns one weakly augmented view and the label.
- Unlabeled dataset returns one weakly augmented view and *two* differently strongly augmented views (for SimMatch contrastive loss).
- Test/Validation dataset returns one weakly augmented view (or none) and the label.

```
# ----------------------------
# 3. Dataset Classes
# ----------------------------

# Dataset for Labeled Data
class FreesoundLabeledDataset(Dataset):
    def __init__(self, df, audio_dir, label_map,
transform=preprocess_audio, augment=weak_augment):
        self.df = df
        self.audio_dir = audio_dir
        self.label_map = label_map
        self.transform = transform
        self.augment = augment # Only weak augmentation needed for
supervised loss
```

```
        self.fnames = df.index.tolist()

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        fname = self.fnames[idx]
        file_path = os.path.join(self.audio_dir, fname)
        mel = self.transform(file_path)
        mel_tensor_aug = self.augment(mel) # (1, n_mels, time)
        label = self.label_map[self.df.loc[fname, 'label']]
        return mel_tensor_aug, torch.tensor(label)

# Dataset for Unlabeled Data
class FreesoundUnlabeledDataset(Dataset):
    def __init__(self, df, audio_dir, label_map,
transform=preprocess_audio, weak_aug=weak_augment,
strong_aug=strong_augment):
        self.df = df
        self.audio_dir = audio_dir
        self.label_map = label_map # Keep label map for potential
analysis, but don't return label
        self.transform = transform
        self.weak_aug = weak_aug
        self.strong_aug = strong_aug
        self.fnames = df.index.tolist()

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        fname = self.fnames[idx]
        file_path = os.path.join(self.audio_dir, fname)
        mel = self.transform(file_path)
        mel_tensor_weak = self.weak_aug(mel)     # For pseudo-label
generation
        mel_tensor_strong1 = self.strong_aug(mel) # For classification
& contrastive loss
        mel_tensor_strong2 = self.strong_aug(mel) # For contrastive
loss
        # We don't return the true label for unlabeled data during
training
        return mel_tensor_weak, mel_tensor_strong1, mel_tensor_strong2

# Dataset for Testing/Validation (uses weak augmentation/no
augmentation)
class FreesoundEvalDataset(Dataset):
    def __init__(self, df, audio_dir, label_map,
transform=preprocess_audio, augment=weak_augment):
        self.df = df
```

```
        self.audio_dir = audio_dir
        self.label_map = label_map
        self.transform = transform
        self.augment = augment # Use weak/no augment for eval
consistency
        self.fnames = df.index.tolist()

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        fname = self.fnames[idx]
        file_path = os.path.join(self.audio_dir, fname)
        mel = self.transform(file_path)
        mel_tensor = self.augment(mel) # (1, n_mels, time)
        label = self.label_map[self.df.loc[fname, 'label']]
        return mel_tensor, torch.tensor(label)
```

4. Prepare Metadata, Label Map, and Data Splits

- Load `train.csv` and `test_post_competition.csv`.
- Create the label map.
- Split the original `train.csv` data into training and validation sets.
- Within the training loop, further split the training set into labeled and unlabeled based on the current `labeled_percent`.

```
# ------------------------------
# 4. Prepare Metadata and Label Map
# ------------------------------
# Load training CSV
train_df_full = pd.read_csv(config.train_csv_path)
# Ensure fname is index for easy lookup
if 'fname' in train_df_full.columns:
    train_df_full.set_index("fname", inplace=True)

# Load test CSV (for final evaluation - assuming it has labels)
try:
    test_df = pd.read_csv(config.test_csv_path)
    if 'fname' in test_df.columns:
        test_df.set_index("fname", inplace=True)
    # Ensure test set has labels for evaluation
    if 'label' not in test_df.columns or
test_df['label'].isnull().any():
         print("Warning: Test CSV does not contain labels or has
missing labels. Using manually_verified column if available.")
         # Try using 'manually_verified' if 'label' is
missing/incomplete
         if 'manually_verified' in test_df.columns and
test_df['manually_verified'].notnull().all():
             # Heuristic: Assume verified files are correctly labeled
```

```
by filename pattern or other logic if needed.
             # This part might need competition-specific logic if
labels aren't directly provided.
             # For now, let's assume the test set *is* labeled for
evaluation simplicity.
             print("Test set seems labeled based on
filename/verification. Proceeding with evaluation.")
         else:
             print("Cannot evaluate on test set without ground truth
labels.")
             test_df = None # Disable test evaluation
    else:
        test_df = test_df.dropna(subset=['label'])
except FileNotFoundError:
    print(f"Warning: Test CSV not found at {config.test_csv_path}.
Skipping test evaluation.")
    test_df = None

# Create label mapping (alphabetical order)
labels = sorted(train_df_full['label'].unique())
label_map = {label: idx for idx, label in enumerate(labels)}
idx_to_label = {idx: label for label, idx in label_map.items()}
config.num_classes = len(labels) # Update num_classes based on actual
data
print(f"Number of classes: {config.num_classes}")
print(f"Labels: {labels}")

# --- Split Train/Validation ---
# Split the *full* training data first to get a held-out validation
set
# Use StratifiedShuffleSplit to ensure representative split even if we
run only once
sss_val = StratifiedShuffleSplit(n_splits=1,
test_size=config.val_percent, random_state=config.seed)
train_idx, val_idx = next(sss_val.split(train_df_full.index,
train_df_full['label']))

train_df = train_df_full.iloc[train_idx]
val_df = train_df_full.iloc[val_idx]

print(f"\nFull training samples: {len(train_df_full)}")
print(f"Split into: Training samples: {len(train_df)}, Validation
samples: {len(val_df)}")
if test_df is not None:
    print(f"Test samples: {len(test_df)}")

# We will split train_df further into labeled/unlabeled inside the
training loop
```

```
Warning: Test CSV does not contain labels or has missing labels. Using
manually_verified column if available.
Cannot evaluate on test set without ground truth labels.
Number of classes: 41
Labels: ['Acoustic_guitar', 'Applause', 'Bark', 'Bass_drum',
'Burping_or_eructation', 'Bus', 'Cello', 'Chime', 'Clarinet',
'Computer_keyboard', 'Cough', 'Cowbell', 'Double_bass',
'Drawer_open_or_close', 'Electric_piano', 'Fart', 'Finger_snapping',
'Fireworks', 'Flute', 'Glockenspiel', 'Gong', 'Gunshot_or_gunfire',
'Harmonica', 'Hi-hat', 'Keys_jangling', 'Knock', 'Laughter', 'Meow',
'Microwave_oven', 'Oboe', 'Saxophone', 'Scissors', 'Shatter',
'Snare_drum', 'Squeak', 'Tambourine', 'Tearing', 'Telephone',
'Trumpet', 'Violin_or_fiddle', 'Writing']

Full training samples: 9473
Split into: Training samples: 8525, Validation samples: 948
```

5. Define the Model Architecture (Encoder + Classifier + EPASS Projectors)

- Use a pre-trained ResNet18 as the backbone encoder.
- Modify the first convolutional layer for 1-channel (spectrogram) input.
- Add a single linear classifier head.
- Add **multiple** MLP projector heads (EPASS).

```
# -----------------------------------------------------------------
# 5. Define the Model Architecture (Encoder + Classifier + Projectors)
# -----------------------------------------------------------------
class EpassSimMatchNet(nn.Module):
    def __init__(self, num_classes, embedding_dim, num_projectors,
pretrained=True):
        super().__init__()
        # Encoder (ResNet18 base)
        base_model =
models.resnet18(weights=models.ResNet18_Weights.DEFAULT if pretrained
else None)
        base_model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2,
padding=3, bias=False)
        self.encoder = nn.Sequential(*list(base_model.children())[:-
1])
        encoder_output_dim = base_model.fc.in_features # 512 for
ResNet18

        # Classifier Head
        self.fc = nn.Linear(encoder_output_dim, num_classes)

        # EPASS Projector Heads (Multiple MLPs)
        self.projectors = nn.ModuleList([
            nn.Sequential(
```

```
                nn.Linear(encoder_output_dim, encoder_output_dim), #
Optional: intermediate layer
                nn.ReLU(),
                nn.Linear(encoder_output_dim, embedding_dim)
            ) for _ in range(num_projectors)
        ])
        self.num_projectors = num_projectors

    def forward(self, x):
        features = self.encoder(x)
        flat_features = torch.flatten(features, 1)

        # Classification logits
        logits = self.fc(flat_features)

        # Get embeddings from all projectors
        embeddings = [proj(flat_features) for proj in self.projectors]

        # Ensemble (average) embeddings for contrastive loss
        # Stack along a new dimension (e.g., dim 0), then mean
        ensembled_embedding = torch.mean(torch.stack(embeddings,
dim=0), dim=0)

        # Return logits for classification and the *ensembled*
embedding for contrastive loss
        return logits, ensembled_embedding

# Instantiate the model
model = EpassSimMatchNet(
    num_classes=config.num_classes,
    embedding_dim=config.embedding_dim,
    num_projectors=config.num_projectors
).to(config.device)

print(f"Model created with {config.num_projectors} projectors and
moved to device.")
# Optional: Print model summary or number of parameters
# print(model)
num_params = sum(p.numel() for p in model.parameters() if
p.requires_grad)
print(f"Total trainable parameters: {num_params:,}")

Downloading: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 214MB/s]

Model created with 3 projectors and moved to device.
Total trainable parameters: 12,176,233
```

6. Create DataLoaders

- Create DataLoaders for labeled, unlabeled (dynamically sized), validation, and test sets.

```
# ---------------------------
# 6. Create DataLoaders (Helper Function)
# ---------------------------
def create_dataloaders(labeled_df, unlabeled_df, val_df, test_df,
config):
    train_labeled_dataset = FreesoundLabeledDataset(labeled_df,
config.audio_train_dir, label_map)
    train_unlabeled_dataset = FreesoundUnlabeledDataset(unlabeled_df,
config.audio_train_dir, label_map)
    val_dataset = FreesoundEvalDataset(val_df, config.audio_train_dir,
label_map)
    test_dataset = FreesoundEvalDataset(test_df,
config.audio_test_dir, label_map) if test_df is not None else None

    # Calculate batch sizes based on mu ratio
    # Ensure labeled batch size is at least 1
    labeled_bs = max(1, config.batch_size // (config.mu + 1))
    unlabeled_bs = config.batch_size - labeled_bs
    print(f"  Using Labeled BS: {labeled_bs}, Unlabeled BS:
{unlabeled_bs}")

    labeled_loader = DataLoader(train_labeled_dataset,
                              batch_size=labeled_bs,
                              shuffle=True,
                              num_workers=config.num_workers,
                              drop_last=True) # Drop last if not
divisible

    unlabeled_loader = DataLoader(train_unlabeled_dataset,
                                batch_size=unlabeled_bs,
                                shuffle=True,
                                num_workers=config.num_workers,
                                drop_last=True) # Drop last if not
divisible

    val_loader = DataLoader(val_dataset,
                            batch_size=config.batch_size, # Use full
batch size for eval
                            shuffle=False,
                            num_workers=config.num_workers)

    test_loader = DataLoader(test_dataset,
                             batch_size=config.batch_size,
                             shuffle=False,
                             num_workers=config.num_workers) if
test_dataset is not None else None
```

```
    print(f"  Loaders created. Num labeled batches/epoch:
{len(labeled_loader)}, Num unlabeled batches/epoch:
{len(unlabeled_loader)}")
    return labeled_loader, unlabeled_loader, val_loader, test_loader
```

7. Define Training and Evaluation Functions

- **`train_one_epoch`**:
  - Takes model, optimizer, labeled/unlabeled loaders, loss criteria.
  - Iterates through both loaders simultaneously.
  - Calculates supervised loss (`loss_s`) on labeled data.
  - Calculates unsupervised classification loss (`loss_u`) using pseudo-labels.
  - Calculates unsupervised contrastive loss (`loss_c`) using ensembled embeddings from two strong views (SimMatch + EPASS).
  - Combines losses and performs backpropagation.
- **`evaluate`**:
  - Standard evaluation loop using the classification head.

```
# ---------------------------------------
# 7. Training and Evaluation Functions
# ---------------------------------------

def train_one_epoch(model, optimizer, labeled_loader,
unlabeled_loader, criterion_s, criterion_u, criterion_c, epoch):
    model.train()
    running_loss_s = 0.0
    running_loss_u = 0.0
    running_loss_c = 0.0
    correct_labeled = 0
    total_labeled = 0
    mask_ratios = []

    # Ensure unlabeled loader defines the epoch length
    num_batches = len(unlabeled_loader)
    # Use cycle for the potentially smaller labeled loader
    labeled_iter = itertools.cycle(labeled_loader)

    train_iterator = tqdm(unlabeled_loader, total=num_batches,
desc=f"Epoch {epoch+1}")

    for batch_idx, (inputs_u_w, inputs_u_s1, inputs_u_s2) in
enumerate(train_iterator):
        # Get labeled data for this step
        try:
            inputs_l, labels_l = next(labeled_iter)
        except StopIteration:
            # Should not happen if unlabeled_loader is longer and
drop_last=True for both
            print("Warning: Labeled loader exhausted unexpectedly.")
```

```
            continue

        # Move data to device
        inputs_l, labels_l = inputs_l.to(config.device),
labels_l.to(config.device)
        inputs_u_w = inputs_u_w.to(config.device)
        inputs_u_s1 = inputs_u_s1.to(config.device)
        inputs_u_s2 = inputs_u_s2.to(config.device)

        labeled_bs = inputs_l.size(0)
        unlabeled_bs = inputs_u_w.size(0)

        # --- Supervised Loss ---
        logits_l, _ = model(inputs_l) # We only need logits for
supervised loss
        loss_s = criterion_s(logits_l, labels_l)

        # --- Unsupervised Losses ---
        # 1. Pseudo-Labeling Loss (Classification Consistency)
        with torch.no_grad():
            logits_u_w, _ = model(inputs_u_w)
            probs_u_w = torch.softmax(logits_u_w, dim=1)
            max_probs, pseudo_labels_u = torch.max(probs_u_w, dim=1)
            mask = (max_probs >= config.threshold).float()
            mask_ratios.append(mask.mean().item())

        logits_u_s1, embeddings_s1 = model(inputs_u_s1)
        loss_u_vec = criterion_u(logits_u_s1, pseudo_labels_u)
        loss_u = (loss_u_vec * mask).mean() # Apply mask

        # 2. Contrastive Loss (Instance Similarity using EPASS
embeddings)
        _, embeddings_s2 = model(inputs_u_s2) # Only need embeddings
for the second strong view

        # Normalize the ensembled embeddings (important for
contrastive loss)
        embeddings_s1_norm = F.normalize(embeddings_s1, dim=1)
        embeddings_s2_norm = F.normalize(embeddings_s2, dim=1)

        # SimMatch Contrastive Loss Calculation (simplified version:
compare s1 vs s2)
        # Calculate similarity matrix (dot product)
        sim_matrix = torch.mm(embeddings_s1_norm,
embeddings_s2_norm.t()) / config.temperature

        # Targets: identity matrix (match corresponding augmented
views)
        targets = torch.arange(unlabeled_bs).to(config.device)
```

```
        # Calculate cross-entropy loss (symmetric: compare s1->s2 and
s2->s1)
        loss_c_vec1 = criterion_c(sim_matrix, targets)
        loss_c_vec2 = criterion_c(sim_matrix.t(), targets) # Symmetric
loss
        loss_c = (loss_c_vec1 + loss_c_vec2) / 2.0
        loss_c = (loss_c * mask).mean() # Apply the same mask as
pseudo-labeling loss

        # --- Combine Losses ---
        total_loss = loss_s + config.wu * loss_u + config.wc * loss_c

        # --- Backpropagation and Optimization ---
        optimizer.zero_grad()
        total_loss.backward()
        optimizer.step()

        # --- Statistics ---
        running_loss_s += loss_s.item() * labeled_bs
        running_loss_u += loss_u.item() * unlabeled_bs # Use
unlabeled_bs for unsupervised loss avg
        running_loss_c += loss_c.item() * unlabeled_bs # Use
unlabeled_bs for contrastive loss avg

        preds_l = logits_l.argmax(dim=1)
        correct_labeled += (preds_l == labels_l).sum().item()
        total_labeled += labeled_bs

        # Update progress bar
        train_iterator.set_postfix(Loss=f"{total_loss.item():.4f}",
Ls=f"{loss_s.item():.4f}", Lu=f"{loss_u.item():.4f}",
Lc=f"{loss_c.item():.4f}", Mask=f"{np.mean(mask_ratios[-10:]):.2f}")

    # Calculate average losses and accuracy for the epoch
    # Use total labeled samples for Ls and total unlabeled samples
processed for Lu, Lc
    total_unlabeled_processed = num_batches *
unlabeled_loader.batch_size
    avg_loss_s = running_loss_s / total_labeled if total_labeled > 0
else 0
    avg_loss_u = running_loss_u / total_unlabeled_processed if
total_unlabeled_processed > 0 else 0
    avg_loss_c = running_loss_c / total_unlabeled_processed if
total_unlabeled_processed > 0 else 0
    acc_labeled = correct_labeled / total_labeled if total_labeled > 0
else 0
    avg_mask_ratio = np.mean(mask_ratios) if mask_ratios else 0

    return avg_loss_s, avg_loss_u, avg_loss_c, acc_labeled,
avg_mask_ratio
```

```
def evaluate(model, loader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in tqdm(loader, desc="Evaluating",
leave=False):
            inputs, labels = inputs.to(device), labels.to(device)
            # Only use logits for evaluation
            outputs, _ = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * inputs.size(0)
            preds = outputs.argmax(dim=1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    epoch_loss = running_loss / total if total > 0 else 0
    epoch_acc = correct / total if total > 0 else 0
    return epoch_loss, epoch_acc, all_preds, all_labels
```

8. Training Loop

- Initialize model, optimizer, loss functions.
- Loop through specified labeled data percentages (20%, 80%).
  - For each percentage:
    - Split the training data into labeled and unlabeled sets.
    - Create dataloaders.
    - Re-initialize model weights and optimizer for a fair comparison.
    - Loop through epochs:
      - Call `train_one_epoch`.
      - Call `evaluate` on the validation set.
      - Track history (losses, accuracies, mask ratio).
      - Check if current validation accuracy is the best *overall*.
      - If so, save the model's state dictionary and record the best accuracy and corresponding labeled percentage.

```
# ----------------------------
# 8. Training Loop
# ----------------------------
criterion_s = nn.CrossEntropyLoss() # Supervised loss
criterion_u = nn.CrossEntropyLoss(reduction='none') # Unsupervised
```

```
classification loss
criterion_c = nn.CrossEntropyLoss(reduction='none') # Contrastive loss
(applied per-sample, then masked & averaged)

best_val_acc_overall = 0.0
best_model_state = None
best_labeled_percent = -1
history = {}

for labeled_percent in config.labeled_percents:
    print(f"\n----- Training with {labeled_percent*100:.0f}% Labeled
Data -----")
    history[labeled_percent] = {'train_loss_s': [], 'train_loss_u':
[], 'train_loss_c': [],
                                'train_acc_l': [], 'val_loss': [],
'val_acc': [], 'mask_ratio': []}

    # --- Create Labeled/Unlabeled Split for this run ---
    sss_label = StratifiedShuffleSplit(n_splits=1,
train_size=labeled_percent, random_state=config.seed +
int(labeled_percent*100))
    # Use train_df (which excludes validation data)
    labeled_idx, unlabeled_idx = next(sss_label.split(train_df.index,
train_df['label']))
    labeled_df_run = train_df.iloc[labeled_idx]
    unlabeled_df_run = train_df.iloc[unlabeled_idx]
    print(f"  Labeled samples for this run: {len(labeled_df_run)}")
    print(f"  Unlabeled samples for this run:
{len(unlabeled_df_run)}")

    # --- Create DataLoaders for this run ---
    labeled_loader, unlabeled_loader, val_loader, test_loader =
create_dataloaders(
        labeled_df_run, unlabeled_df_run, val_df, test_df, config
    )

    # --- Re-initialize Model and Optimizer for each run ---
    print("  Re-initializing model and optimizer...")
    model = EpassSimMatchNet(
        num_classes=config.num_classes,
        embedding_dim=config.embedding_dim,
        num_projectors=config.num_projectors
    ).to(config.device)
    optimizer = optim.Adam(model.parameters(), lr=config.lr)
    # Optional: Learning rate scheduler
    # scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=config.epochs)

    best_val_acc_run = 0.0 # Best validation accuracy for *this* run
```

```
    # --- Epoch Loop for this run ---
    for epoch in range(config.epochs):
        tr_loss_s, tr_loss_u, tr_loss_c, tr_acc_l, mask_ratio = 
train_one_epoch(
            model, optimizer, labeled_loader, unlabeled_loader,
            criterion_s, criterion_u, criterion_c, epoch
        )
        val_loss, val_acc, _, _ = evaluate(model, val_loader, 
criterion_s, config.device)

        # Optional: Step the scheduler
        # scheduler.step()

        # Log history for this run
        history[labeled_percent]['train_loss_s'].append(tr_loss_s)
        history[labeled_percent]['train_loss_u'].append(tr_loss_u)
        history[labeled_percent]['train_loss_c'].append(tr_loss_c)
        history[labeled_percent]['train_acc_l'].append(tr_acc_l)
        history[labeled_percent]['val_loss'].append(val_loss)
        history[labeled_percent]['val_acc'].append(val_acc)
        history[labeled_percent]['mask_ratio'].append(mask_ratio)

        print(f"  Epoch {epoch+1}/{config.epochs} -> "
              f"Loss S: {tr_loss_s:.4f}, Loss U: {tr_loss_u:.4f}, Loss 
C: {tr_loss_c:.4f}, Acc (L): {tr_acc_l:.4f} | "
              f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} | 
Mask Ratio: {mask_ratio:.3f}")

        # Check if this is the best model *overall*
        if val_acc > best_val_acc_overall:
            best_val_acc_overall = val_acc
            best_labeled_percent = labeled_percent
            best_model_state = copy.deepcopy(model.state_dict()) # 
Deep copy state dict
            print(f"  *** New best validation accuracy overall: 
{best_val_acc_overall:.4f} (from {labeled_percent*100:.0f}% run). 
Saving model state... ***")
            # Save immediately or just store the state dict and save 
at the end
            # torch.save(best_model_state, config.model_save_path)

# --- End of Training Loop ---
print(f"\nFinished training across all label percentages.")
print(f"Best overall validation accuracy: {best_val_acc_overall:.4f} 
achieved with {best_labeled_percent*100:.0f}% labeled data.")

# Save the overall best model state if found
if best_model_state is not None:
    print(f"Saving the best overall model state to 
{config.model_save_path}")
```

```
    torch.save(best_model_state, config.model_save_path)
else:
    print("No best model state was saved (perhaps validation accuracy
never improved?).")

# Load the best model for final evaluation
print(f"\nLoading best overall model for final evaluation...")
if best_model_state is not None:
    model = EpassSimMatchNet( # Recreate the model structure
        num_classes=config.num_classes,
        embedding_dim=config.embedding_dim,
        num_projectors=config.num_projectors
    ).to(config.device)
    model.load_state_dict(best_model_state)
    print("Best model loaded successfully.")
else:
    print("Could not load a best model state. Evaluation will use the
model from the last epoch of the last run.")
    # 'model' variable still holds the last trained model

# Ensure test_loader was created if test_df exists
if test_df is not None:
     # Need to create test_loader if it wasn't created in the last
loop iteration
     # (This assumes val_df exists from the initial split)
     _, _, _, test_loader = create_dataloaders(labeled_df_run,
unlabeled_df_run, val_df, test_df, config)
else:
     test_loader = None
```

```
----- Training with 20% Labeled Data -----
  Labeled samples for this run: 1705
  Unlabeled samples for this run: 6820
  Using Labeled BS: 4, Unlabeled BS: 28
  Loaders created. Num labeled batches/epoch: 426, Num unlabeled
batches/epoch: 243
  Re-initializing model and optimizer...
```

{"model_id":"8135fe209ce84f2aa09b49d019fadd6a","version_major":2,"version_minor":0}

{"model_id":"cae4cdb27baa44e3b811adafc462edce","version_major":2,"version_minor":0}

```
  Epoch 1/50 -> Loss S: 3.3016, Loss U: 0.0018, Loss C: 0.0056, Acc
(L): 0.1409 | Val Loss: 2.7096, Val Acc: 0.2468 | Mask Ratio: 0.004
  *** New best validation accuracy overall: 0.2468 (from 20% run).
Saving model state... ***
```

{"model_id":"0639e251b5e0440c9e0466919b2495e8","version_major":2,"version_minor":0}

{"model_id":"22263d34160d42cea9785623f1598cf9","version_major":2,"version_minor":0}

```
  Epoch 2/50 -> Loss S: 2.7928, Loss U: 0.0024, Loss C: 0.0128, Acc
(L): 0.2469 | Val Loss: 2.5089, Val Acc: 0.3175 | Mask Ratio: 0.014
  *** New best validation accuracy overall: 0.3175 (from 20% run).
Saving model state... ***
```

{"model_id":"2b0a0fb92ee7446cb12efcd9d8f7a4e5","version_major":2,"version_minor":0}

{"model_id":"89265c7079b846d6b37e7c2361cda140","version_major":2,"version_minor":0}

```
  Epoch 3/50 -> Loss S: 2.4620, Loss U: 0.0078, Loss C: 0.0241, Acc
(L): 0.3025 | Val Loss: 2.1524, Val Acc: 0.4156 | Mask Ratio: 0.034
  *** New best validation accuracy overall: 0.4156 (from 20% run).
Saving model state... ***
```

{"model_id":"53a5279d4ab94523a1712c1bd5f79b22","version_major":2,"version_minor":0}

{"model_id":"b1f924abaf4f4dae9701e9dde7a468ce","version_major":2,"version_minor":0}

```
  Epoch 4/50 -> Loss S: 2.2248, Loss U: 0.0138, Loss C: 0.0324, Acc
(L): 0.3796 | Val Loss: 1.8977, Val Acc: 0.4852 | Mask Ratio: 0.060
  *** New best validation accuracy overall: 0.4852 (from 20% run).
Saving model state... ***
```

{"model_id":"d35164818a0f4e358bc6936199cb1e26","version_major":2,"version_minor":0}

{"model_id":"5e78720ae2aa490bbdd28428c6f61769","version_major":2,"version_minor":0}

```
  Epoch 5/50 -> Loss S: 1.9552, Loss U: 0.0229, Loss C: 0.0494, Acc
(L): 0.4794 | Val Loss: 1.9525, Val Acc: 0.4652 | Mask Ratio: 0.100
```

{"model_id":"4e9fa9709f6d48609d6fb02c8e13e12c","version_major":2,"version_minor":0}

{"model_id":"fdba8cd89d4d4bf5b97d29bb01c350dd","version_major":2,"version_minor":0}

```
  Epoch 6/50 -> Loss S: 1.7273, Loss U: 0.0272, Loss C: 0.0485, Acc
(L): 0.5072 | Val Loss: 1.8890, Val Acc: 0.5211 | Mask Ratio: 0.133
  *** New best validation accuracy overall: 0.5211 (from 20% run).
Saving model state... ***
```

{"model_id":"ff11d23e07d94953937dc1774297d74e","version_major":2,"version_minor":0}

{"model_id":"4fc417ff679b4e1dab5393a6a77d87e6","version_major":2,"version_minor":0}

```
  Epoch 7/50 -> Loss S: 1.6274, Loss U: 0.0371, Loss C: 0.0603, Acc
(L): 0.5494 | Val Loss: 1.6829, Val Acc: 0.5464 | Mask Ratio: 0.166
  *** New best validation accuracy overall: 0.5464 (from 20% run).
Saving model state... ***
```

{"model_id":"e4637cd94a144afdaa8224bd2ee14712","version_major":2,"version_minor":0}

{"model_id":"9e03cbe132364676957c38cbf2f924fb","version_major":2,"version_minor":0}

```
  Epoch 8/50 -> Loss S: 1.3461, Loss U: 0.0513, Loss C: 0.0715, Acc
(L): 0.6307 | Val Loss: 1.8149, Val Acc: 0.5601 | Mask Ratio: 0.216
  *** New best validation accuracy overall: 0.5601 (from 20% run).
Saving model state... ***
```

{"model_id":"2dba4465adb1413cbd948083d0b11a95","version_major":2,"version_minor":0}

{"model_id":"964f2baf19f44e7186c0f1c378ed3942","version_major":2,"version_minor":0}

```
  Epoch 9/50 -> Loss S: 1.2304, Loss U: 0.0537, Loss C: 0.0753, Acc
(L): 0.6461 | Val Loss: 1.6356, Val Acc: 0.6086 | Mask Ratio: 0.257
  *** New best validation accuracy overall: 0.6086 (from 20% run).
Saving model state... ***
```

{"model_id":"de7a20164b234d1397923c30a4945037","version_major":2,"version_minor":0}

{"model_id":"2f72e2f602f64a9faab7c98c7570be0b","version_major":2,"version_minor":0}

```
  Epoch 10/50 -> Loss S: 1.0820, Loss U: 0.0606, Loss C: 0.0793, Acc
(L): 0.7068 | Val Loss: 1.6146, Val Acc: 0.6076 | Mask Ratio: 0.290
```

{"model_id":"37aa47054a2c4a36a3fc71eff9a07c83","version_major":2,"version_minor":0}

{"model_id":"0f53e665228747a79a3bbd526d637291","version_major":2,"version_minor":0}

```
  Epoch 11/50 -> Loss S: 0.8939, Loss U: 0.0654, Loss C: 0.0804, Acc
(L): 0.7387 | Val Loss: 1.5404, Val Acc: 0.6340 | Mask Ratio: 0.322
  *** New best validation accuracy overall: 0.6340 (from 20% run).
Saving model state... ***
```

{"model_id":"ff1a45649a744a7e8aab8b22693578ea","version_major":2,"version_minor":0}

{"model_id":"d91809e163954a438916e37da66699fc","version_major":2,"version_minor":0}

```
  Epoch 12/50 -> Loss S: 0.9186, Loss U: 0.0691, Loss C: 0.0786, Acc
(L): 0.7418 | Val Loss: 1.4439, Val Acc: 0.6582 | Mask Ratio: 0.340
  *** New best validation accuracy overall: 0.6582 (from 20% run).
Saving model state... ***
```

{"model_id":"30734d852bd14166b1425f413b8c9d52","version_major":2,"version_minor":0}

{"model_id":"8e265cd00f504249ad14b8c68aea155b","version_major":2,"version_minor":0}

```
  Epoch 13/50 -> Loss S: 0.7580, Loss U: 0.0748, Loss C: 0.0738, Acc
(L): 0.7860 | Val Loss: 1.6890, Val Acc: 0.6213 | Mask Ratio: 0.372
```

{"model_id":"9ad8f52b56f149c8947c3330eacb3f25","version_major":2,"version_minor":0}

{"model_id":"46a5e0969b0a4777a62538807f4a9c9b","version_major":2,"version_minor":0}

```
  Epoch 14/50 -> Loss S: 0.7434, Loss U: 0.0949, Loss C: 0.0839, Acc
(L): 0.8014 | Val Loss: 1.5218, Val Acc: 0.6487 | Mask Ratio: 0.385
```

{"model_id":"cc187155cd184c9a94d8877d2783f21e","version_major":2,"version_minor":0}

{"model_id":"96e8bee314264ffd8e387e5dec5297a8","version_major":2,"version_minor":0}

```
  Epoch 15/50 -> Loss S: 0.5921, Loss U: 0.0739, Loss C: 0.0803, Acc
(L): 0.8272 | Val Loss: 1.4673, Val Acc: 0.6793 | Mask Ratio: 0.418
  *** New best validation accuracy overall: 0.6793 (from 20% run).
Saving model state... ***
```

{"model_id":"22ad2c900bfc43a7a461223105ca86bc","version_major":2,"version_minor":0}

{"model_id":"0cdac2d3353a4dc3a6c8889de37f9733","version_major":2,"version_minor":0}

```
  Epoch 16/50 -> Loss S: 0.4453, Loss U: 0.0753, Loss C: 0.0705, Acc
(L): 0.8971 | Val Loss: 1.7311, Val Acc: 0.6382 | Mask Ratio: 0.451
```

{"model_id":"118ff51189964eb38edbfd56a1bc4eb8","version_major":2,"version_minor":0}

{"model_id":"daf4ad9d21eb41ddb14be00e2c5f5bc6","version_major":2,"version_minor":0}

```
  Epoch 17/50 -> Loss S: 0.3840, Loss U: 0.0810, Loss C: 0.0715, Acc
(L): 0.9012 | Val Loss: 1.4569, Val Acc: 0.6930 | Mask Ratio: 0.470
  *** New best validation accuracy overall: 0.6930 (from 20% run).
Saving model state... ***
```

{"model_id":"be88033e254e4e88b245beabc44d39ab","version_major":2,"version_minor":0}

{"model_id":"2002c7bdbe644d52a2bb293f7ec98dd3","version_major":2,"version_minor":0}

```
  Epoch 18/50 -> Loss S: 0.3742, Loss U: 0.0872, Loss C: 0.0768, Acc
(L): 0.8971 | Val Loss: 1.5199, Val Acc: 0.6888 | Mask Ratio: 0.493
```

{"model_id":"5043db4be2844b91a9a44c185b3dc227","version_major":2,"version_minor":0}

{"model_id":"ac586d70ccaa4d2cafb6e6a8e6eb9506","version_major":2,"version_minor":0}

```
  Epoch 19/50 -> Loss S: 0.4005, Loss U: 0.0891, Loss C: 0.0723, Acc
(L): 0.8848 | Val Loss: 1.6353, Val Acc: 0.6730 | Mask Ratio: 0.485
```

{"model_id":"feac5183bc374cc2a581dbdfcc30918f","version_major":2,"version_minor":0}

{"model_id":"5777f12b6bf14cde8627cc0f1efd839a","version_major":2,"version_minor":0}

```
  Epoch 20/50 -> Loss S: 0.2428, Loss U: 0.0959, Loss C: 0.0712, Acc
(L): 0.9434 | Val Loss: 1.6303, Val Acc: 0.6909 | Mask Ratio: 0.523
```

{"model_id":"f51098d003ca44b593b82aa3754cf6ec","version_major":2,"version_minor":0}

{"model_id":"742f1d3bbe8847f6a05ffa6db2627034","version_major":2,"version_minor":0}

```
  Epoch 21/50 -> Loss S: 0.2888, Loss U: 0.1024, Loss C: 0.0679, Acc
(L): 0.9187 | Val Loss: 1.6230, Val Acc: 0.6909 | Mask Ratio: 0.522
```

{"model_id":"9074a5725e5e4eb6ad173b3b58fdcdb6","version_major":2,"version_minor":0}

{"model_id":"daa46ff61eac41bbaa2987aa44437a24","version_major":2,"version_minor":0}

```
  Epoch 22/50 -> Loss S: 0.2915, Loss U: 0.1010, Loss C: 0.0686, Acc
(L): 0.9259 | Val Loss: 1.7968, Val Acc: 0.6593 | Mask Ratio: 0.530
```

{"model_id":"7fd20e33cc8842bea19d463ba9d32962","version_major":2,"version_minor":0}

{"model_id":"3053e13c137c4236a075a548e83e632e","version_major":2,"version_minor":0}

```
  Epoch 23/50 -> Loss S: 0.2022, Loss U: 0.1048, Loss C: 0.0685, Acc
(L): 0.9496 | Val Loss: 1.5412, Val Acc: 0.7078 | Mask Ratio: 0.555
  *** New best validation accuracy overall: 0.7078 (from 20% run).
Saving model state... ***
```

{"model_id":"820f059145634a1fa49514e0e63d179a","version_major":2,"version_minor":0}

{"model_id":"078832acc10e402fa6031d4585831a08","version_major":2,"version_minor":0}

```
  Epoch 24/50 -> Loss S: 0.1784, Loss U: 0.1022, Loss C: 0.0621, Acc
(L): 0.9568 | Val Loss: 1.7062, Val Acc: 0.6783 | Mask Ratio: 0.571
```

{"model_id":"e1fe848555ba4424824a7bac0de681fa","version_major":2,"version_minor":0}

{"model_id":"0c50f1a51a5248d7959b5e9d72778a04","version_major":2,"version_minor":0}

```
  Epoch 25/50 -> Loss S: 0.2029, Loss U: 0.0899, Loss C: 0.0648, Acc
(L): 0.9496 | Val Loss: 1.5106, Val Acc: 0.6878 | Mask Ratio: 0.558
```

{"model_id":"a1d9a3c72b274da9be7bb765dd26ea2a","version_major":2,"version_minor":0}

{"model_id":"fb4126fec5ea49d5913df806743dae3e","version_major":2,"version_minor":0}

```
  Epoch 26/50 -> Loss S: 0.1777, Loss U: 0.1018, Loss C: 0.0635, Acc
(L): 0.9506 | Val Loss: 1.7654, Val Acc: 0.7057 | Mask Ratio: 0.578
```

{"model_id":"59bf14f00314435ab4b27954728d1312","version_major":2,"version_minor":0}

{"model_id":"a1bbae087505462e90de0fc977c028bf","version_major":2,"version_minor":0}

```
  Epoch 27/50 -> Loss S: 0.2158, Loss U: 0.1074, Loss C: 0.0633, Acc
(L): 0.9403 | Val Loss: 1.7159, Val Acc: 0.6909 | Mask Ratio: 0.582
```

{"model_id":"80651badd05740cfb006480cb6926aa9","version_major":2,"version_minor":0}

{"model_id":"47e3ae8b43b74f13a4b35d1c7429e5c3","version_major":2,"version_minor":0}

```
  Epoch 28/50 -> Loss S: 0.2682, Loss U: 0.1021, Loss C: 0.0605, Acc
(L): 0.9290 | Val Loss: 1.7495, Val Acc: 0.6825 | Mask Ratio: 0.570
```

{"model_id":"0d93ced0c91e49ed8f0c8295700f5fd7","version_major":2,"version_minor":0}

{"model_id":"e732b3bce2a5409bb9cb0e20cfe29e12","version_major":2,"version_minor":0}

```
  Epoch 29/50 -> Loss S: 0.2020, Loss U: 0.1080, Loss C: 0.0619, Acc
(L): 0.9527 | Val Loss: 1.5628, Val Acc: 0.7173 | Mask Ratio: 0.573
  *** New best validation accuracy overall: 0.7173 (from 20% run).
Saving model state... ***
```

{"model_id":"8b59761771d840f280e2a752d86fb7fa","version_major":2,"version_minor":0}

{"model_id":"c7d3ce078ea74bb0b0ee5886c282d0b5","version_major":2,"version_minor":0}

```
  Epoch 30/50 -> Loss S: 0.1434, Loss U: 0.1022, Loss C: 0.0554, Acc
(L): 0.9630 | Val Loss: 1.6048, Val Acc: 0.7194 | Mask Ratio: 0.613
  *** New best validation accuracy overall: 0.7194 (from 20% run).
Saving model state... ***
```

{"model_id":"0b6ce4a545744905893f6c7b73d45312","version_major":2,"version_minor":0}

{"model_id":"8f4989418d9f4baa988419517ea1da6b","version_major":2,"version_minor":0}

```
  Epoch 31/50 -> Loss S: 0.1518, Loss U: 0.1083, Loss C: 0.0609, Acc
(L): 0.9609 | Val Loss: 1.8573, Val Acc: 0.6951 | Mask Ratio: 0.610
```

{"model_id":"76109e95ce62441e95e93a3499128eb8","version_major":2,"version_minor":0}

{"model_id":"bad6cf7b7c154b1891c70f9b82b25e5d","version_major":2,"version_minor":0}

```
  Epoch 32/50 -> Loss S: 0.2225, Loss U: 0.1001, Loss C: 0.0522, Acc
(L): 0.9393 | Val Loss: 1.6608, Val Acc: 0.6930 | Mask Ratio: 0.597
```

{"model_id":"97736908452246239e4eb4240a8f9fad","version_major":2,"version_minor":0}

{"model_id":"5b205ecbad4c465198c83739d4fd30a0","version_major":2,"version_minor":0}

```
  Epoch 33/50 -> Loss S: 0.1855, Loss U: 0.1027, Loss C: 0.0551, Acc
(L): 0.9496 | Val Loss: 1.6547, Val Acc: 0.7099 | Mask Ratio: 0.615
```

{"model_id":"c9582bd8e90447f4a460717051321305","version_major":2,"version_minor":0}

{"model_id":"24b725deac354c5891c9c066a33404de","version_major":2,"version_minor":0}

```
  Epoch 34/50 -> Loss S: 0.1460, Loss U: 0.1020, Loss C: 0.0544, Acc (L): 0.9630 | Val Loss: 1.6895, Val Acc: 0.6983 | Mask Ratio: 0.630
```

{"model_id":"f59e5d90339f42ebbe172723e9d69dcb","version_major":2,"version_minor":0}

{"model_id":"0e503b2056dd4617b20c19743cadda8f","version_major":2,"version_minor":0}

```
  Epoch 35/50 -> Loss S: 0.1344, Loss U: 0.1036, Loss C: 0.0554, Acc (L): 0.9671 | Val Loss: 1.5956, Val Acc: 0.7268 | Mask Ratio: 0.630
  *** New best validation accuracy overall: 0.7268 (from 20% run). Saving model state... ***
```

{"model_id":"785bce7939fd4f1ea3c1c31c3bcce486","version_major":2,"version_minor":0}

{"model_id":"c2fb1062658f44a18c4f838d3bb0d1e6","version_major":2,"version_minor":0}

```
  Epoch 36/50 -> Loss S: 0.0779, Loss U: 0.1034, Loss C: 0.0537, Acc (L): 0.9825 | Val Loss: 1.8031, Val Acc: 0.7162 | Mask Ratio: 0.656
```

{"model_id":"32d1a493b49e4b8a9392bec14eb854f9","version_major":2,"version_minor":0}

{"model_id":"71f51b18669e47f0887c7d03fca1b1af","version_major":2,"version_minor":0}

```
  Epoch 37/50 -> Loss S: 0.1523, Loss U: 0.0924, Loss C: 0.0518, Acc (L): 0.9599 | Val Loss: 1.6710, Val Acc: 0.7025 | Mask Ratio: 0.646
```

{"model_id":"84f71880043a4fbbb3c015c5a3001319","version_major":2,"version_minor":0}

{"model_id":"29ce9080ae15438790c0d3998c1a22c9","version_major":2,"version_minor":0}

```
  Epoch 38/50 -> Loss S: 0.1493, Loss U: 0.0937, Loss C: 0.0480, Acc (L): 0.9609 | Val Loss: 1.9136, Val Acc: 0.6962 | Mask Ratio: 0.638
```

{"model_id":"37dd9513940a4b8e815f0f04daddd5ec","version_major":2,"version_minor":0}

{"model_id":"8e6b84136c3c4626b33be847550d4686","version_major":2,"version_minor":0}

```
  Epoch 39/50 -> Loss S: 0.1481, Loss U: 0.0941, Loss C: 0.0501, Acc
(L): 0.9630 | Val Loss: 2.0995, Val Acc: 0.7015 | Mask Ratio: 0.637
```

{"model_id":"1648f5ca719f4f61be524c8e9524f6f4","version_major":2,"version_minor":0}

{"model_id":"e5b2ef807a9a408fa1be2f12c1c97f72","version_major":2,"version_minor":0}

```
  Epoch 40/50 -> Loss S: 0.1588, Loss U: 0.0956, Loss C: 0.0506, Acc
(L): 0.9568 | Val Loss: 1.7353, Val Acc: 0.6973 | Mask Ratio: 0.637
```

{"model_id":"c5ead31a7df74630a9009cc50031504b","version_major":2,"version_minor":0}

{"model_id":"91352bd07cfa4ca2af980d0ee6346acc","version_major":2,"version_minor":0}

```
  Epoch 41/50 -> Loss S: 0.1507, Loss U: 0.0980, Loss C: 0.0503, Acc
(L): 0.9609 | Val Loss: 1.7330, Val Acc: 0.6983 | Mask Ratio: 0.638
```

{"model_id":"befbc3f7d9d347a594ce2590568485ac","version_major":2,"version_minor":0}

{"model_id":"9922dab788df405ba0a548030f6060a9","version_major":2,"version_minor":0}

```
  Epoch 42/50 -> Loss S: 0.1212, Loss U: 0.0970, Loss C: 0.0477, Acc
(L): 0.9712 | Val Loss: 1.5483, Val Acc: 0.7194 | Mask Ratio: 0.654
```

{"model_id":"0d127e8485db4073983096ab1a389992","version_major":2,"version_minor":0}

{"model_id":"180c653aab7f4b0fa95a316b00397534","version_major":2,"version_minor":0}

```
  Epoch 43/50 -> Loss S: 0.1061, Loss U: 0.0889, Loss C: 0.0464, Acc
(L): 0.9722 | Val Loss: 1.6614, Val Acc: 0.7205 | Mask Ratio: 0.669
```

{"model_id":"e975a401d5e2420b985e7e3a81226195","version_major":2,"version_minor":0}

{"model_id":"0fc55f98974a4488a276bf99505d5e80","version_major":2,"version_minor":0}

```
  Epoch 44/50 -> Loss S: 0.1006, Loss U: 0.0923, Loss C: 0.0449, Acc
(L): 0.9743 | Val Loss: 1.8636, Val Acc: 0.6888 | Mask Ratio: 0.672
```

{"model_id":"54967b7ddfbf4428af509364b498f2e9","version_major":2,"version_minor":0}

{"model_id":"5d6badf80cea4114b6bd4f98bfccac21","version_major":2,"version_minor":0}

```
  Epoch 45/50 -> Loss S: 0.1294, Loss U: 0.1065, Loss C: 0.0509, Acc
(L): 0.9640 | Val Loss: 2.1251, Val Acc: 0.6783 | Mask Ratio: 0.657
```

{"model_id":"3680947be4c24309a6c5e1e67c181ed8","version_major":2,"version_minor":0}

{"model_id":"5fa5dbebabc0410684f794b0bee6f8d7","version_major":2,"version_minor":0}

```
  Epoch 46/50 -> Loss S: 0.1139, Loss U: 0.1012, Loss C: 0.0479, Acc
(L): 0.9671 | Val Loss: 1.7386, Val Acc: 0.7300 | Mask Ratio: 0.663
  *** New best validation accuracy overall: 0.7300 (from 20% run).
Saving model state... ***
```

{"model_id":"5f84bea7cd864d47aa1afa3f275238a7","version_major":2,"version_minor":0}

{"model_id":"355bba673e7b41be8da351302c1298b3","version_major":2,"version_minor":0}

```
  Epoch 47/50 -> Loss S: 0.0572, Loss U: 0.0966, Loss C: 0.0439, Acc
(L): 0.9877 | Val Loss: 1.8127, Val Acc: 0.7257 | Mask Ratio: 0.687
```

{"model_id":"0fcbc5b4a6f04619a989a7d57fa0b4db","version_major":2,"version_minor":0}

{"model_id":"fc47a51029ee484fbba47ca7f5c134a7","version_major":2,"version_minor":0}

```
  Epoch 48/50 -> Loss S: 0.0344, Loss U: 0.0957, Loss C: 0.0437, Acc
(L): 0.9928 | Val Loss: 1.9150, Val Acc: 0.7226 | Mask Ratio: 0.704
```

{"model_id":"a202b5efabe04093956f9fc40ca6288c","version_major":2,"version_minor":0}

{"model_id":"537908d5ce794d628218b80d8d0612a8","version_major":2,"version_minor":0}

```
  Epoch 49/50 -> Loss S: 0.1079, Loss U: 0.0970, Loss C: 0.0468, Acc
(L): 0.9650 | Val Loss: 1.8107, Val Acc: 0.7120 | Mask Ratio: 0.686
```

{"model_id":"8df0f8cf295f46ee973e21f2c9cbce26","version_major":2,"version_minor":0}

{"model_id":"d47dfd9b03e948a88dc69d71b32b8e98","version_major":2,"version_minor":0}

```
  Epoch 50/50 -> Loss S: 0.0768, Loss U: 0.0928, Loss C: 0.0419, Acc
(L): 0.9794 | Val Loss: 1.8517, Val Acc: 0.7046 | Mask Ratio: 0.691

----- Training with 80% Labeled Data -----
  Labeled samples for this run: 6820
  Unlabeled samples for this run: 1705
```

```
  Using Labeled BS: 4, Unlabeled BS: 28
  Loaders created. Num labeled batches/epoch: 1705, Num unlabeled
batches/epoch: 60
  Re-initializing model and optimizer...
```

{"model_id":"5a1257d8acb84e8f8e59d109889ef9d9","version_major":2,"version_minor":0}

{"model_id":"0da8dd49709041e98a10030959da850a","version_major":2,"version_minor":0}

```
  Epoch 1/50 -> Loss S: 3.5514, Loss U: 0.0003, Loss C: 0.0008, Acc
(L): 0.1292 | Val Loss: 3.3299, Val Acc: 0.1624 | Mask Ratio: 0.001
```

{"model_id":"86a33ba2fd4b4126974f8017ff9e78ee","version_major":2,"version_minor":0}

{"model_id":"a4452b8c97fd4d24a482dbbeaebfa33c","version_major":2,"version_minor":0}

```
  Epoch 2/50 -> Loss S: 3.4259, Loss U: 0.0000, Loss C: 0.0000, Acc
(L): 0.1375 | Val Loss: 3.3094, Val Acc: 0.1656 | Mask Ratio: 0.000
```

{"model_id":"99bc90bbcbd44771b03c07c24a972e62","version_major":2,"version_minor":0}

{"model_id":"c9b6d17a6f654cde961ba80f041ee9e6","version_major":2,"version_minor":0}

```
  Epoch 3/50 -> Loss S: 3.2123, Loss U: 0.0000, Loss C: 0.0000, Acc
(L): 0.1417 | Val Loss: 2.8752, Val Acc: 0.2416 | Mask Ratio: 0.000
```

{"model_id":"7d17600b09e34f0abbab66f9fead297e","version_major":2,"version_minor":0}

{"model_id":"bd836ef35f084f02980997581a5890fb","version_major":2,"version_minor":0}

```
  Epoch 4/50 -> Loss S: 3.1306, Loss U: 0.0001, Loss C: 0.0057, Acc
(L): 0.1542 | Val Loss: 2.9091, Val Acc: 0.2162 | Mask Ratio: 0.004
```

{"model_id":"8fb9eade54ca4ebe8d0d4b06be6b0438","version_major":2,"version_minor":0}

{"model_id":"ecbfd1d439b74b04a692721b60ab4ea3","version_major":2,"version_minor":0}

```
  Epoch 5/50 -> Loss S: 2.9910, Loss U: 0.0007, Loss C: 0.0071, Acc
(L): 0.1875 | Val Loss: 2.5016, Val Acc: 0.3027 | Mask Ratio: 0.004
```

{"model_id":"c974d0778ec34195aa11aa3ba594b25c","version_major":2,"version_minor":0}

{"model_id":"aca2c7dc53304a03854126cd0b5beb35","version_major":2,"version_minor":0}

```
  Epoch 6/50 -> Loss S: 2.8303, Loss U: 0.0025, Loss C: 0.0076, Acc
(L): 0.2250 | Val Loss: 2.5655, Val Acc: 0.2711 | Mask Ratio: 0.005
```

{"model_id":"b2b0cfc6388344a682a853d8dbc6b8fa","version_major":2,"version_minor":0}

{"model_id":"c2f860f2ef7143c7a1458ea4f61bc4bc","version_major":2,"version_minor":0}

```
  Epoch 7/50 -> Loss S: 2.8355, Loss U: 0.0024, Loss C: 0.0113, Acc
(L): 0.1750 | Val Loss: 2.5198, Val Acc: 0.3091 | Mask Ratio: 0.012
```

{"model_id":"ba4f8424d537440f960050a8ea4644dc","version_major":2,"version_minor":0}

{"model_id":"5701fa4db91b423eb40c2b7dbc5c2912","version_major":2,"version_minor":0}

```
  Epoch 8/50 -> Loss S: 2.7254, Loss U: 0.0023, Loss C: 0.0136, Acc
(L): 0.2417 | Val Loss: 2.7855, Val Acc: 0.3122 | Mask Ratio: 0.012
```

{"model_id":"ea7585363c7044b2ba1ed75b03f589cd","version_major":2,"version_minor":0}

{"model_id":"c340f89c8fb847b6852474d617a703af","version_major":2,"version_minor":0}

```
  Epoch 9/50 -> Loss S: 2.7112, Loss U: 0.0046, Loss C: 0.0256, Acc
(L): 0.2333 | Val Loss: 2.5271, Val Acc: 0.3249 | Mask Ratio: 0.034
```

{"model_id":"4eded4aa60504571a5a8c2d0e95aba64","version_major":2,"version_minor":0}

{"model_id":"10283b443e3b4be4bfc44dad470bb0e4","version_major":2,"version_minor":0}

```
  Epoch 10/50 -> Loss S: 2.7209, Loss U: 0.0025, Loss C: 0.0160, Acc
(L): 0.2167 | Val Loss: 2.3203, Val Acc: 0.3565 | Mask Ratio: 0.026
```

{"model_id":"3ad7c7d1491e4bf685340f88c4566941","version_major":2,"version_minor":0}

{"model_id":"dcfa252565634df0bd12cc831f6362ba","version_major":2,"version_minor":0}

```
  Epoch 11/50 -> Loss S: 2.6350, Loss U: 0.0056, Loss C: 0.0123, Acc
(L): 0.2958 | Val Loss: 2.2756, Val Acc: 0.3513 | Mask Ratio: 0.018
```

{"model_id":"e331b35d2ffb41108f1376efb499e2d8","version_major":2,"version_minor":0}

```
{"model_id":"45b0fb65995a401fab14a473f0549d1e","version_major":2,"version_minor":0}
```

```
  Epoch 12/50 -> Loss S: 2.6385, Loss U: 0.0047, Loss C: 0.0095, Acc (L): 0.2917 | Val Loss: 2.2846, Val Acc: 0.3597 | Mask Ratio: 0.021
```

{"model_id":"e38d321330b54df8bd53ed972a336df7","version_major":2,"version_minor":0}

{"model_id":"240d0eeda0ef4bc89e4117fcb575e9a3","version_major":2,"version_minor":0}

```
  Epoch 13/50 -> Loss S: 2.3733, Loss U: 0.0069, Loss C: 0.0152, Acc (L): 0.3750 | Val Loss: 2.0274, Val Acc: 0.4103 | Mask Ratio: 0.027
```

{"model_id":"db9817f11dd54f55b410b60f483aa1e1","version_major":2,"version_minor":0}

{"model_id":"0228c729deea45b2bc5416f9bafdbdc5","version_major":2,"version_minor":0}

```
  Epoch 14/50 -> Loss S: 2.3869, Loss U: 0.0069, Loss C: 0.0185, Acc (L): 0.3333 | Val Loss: 2.1437, Val Acc: 0.3987 | Mask Ratio: 0.032
```

{"model_id":"bba6872474e7486a8ff9723878ce8b03","version_major":2,"version_minor":0}

{"model_id":"d180948e852f4e5db3dd9919f531daad","version_major":2,"version_minor":0}

```
  Epoch 15/50 -> Loss S: 2.3871, Loss U: 0.0105, Loss C: 0.0206, Acc (L): 0.2875 | Val Loss: 2.1298, Val Acc: 0.4304 | Mask Ratio: 0.024
```

{"model_id":"ae00e4ac7f094764a44cf9018cb9ce9d","version_major":2,"version_minor":0}

{"model_id":"be90ee80502e40c6aa133298dfe07d39","version_major":2,"version_minor":0}

```
  Epoch 16/50 -> Loss S: 2.2784, Loss U: 0.0033, Loss C: 0.0195, Acc (L): 0.3875 | Val Loss: 2.2192, Val Acc: 0.4473 | Mask Ratio: 0.033
```

{"model_id":"8614c08a36514cdeafabd2f347a0422c","version_major":2,"version_minor":0}

{"model_id":"7cf778b6706a471ba3a9e9b61800945f","version_major":2,"version_minor":0}

```
  Epoch 17/50 -> Loss S: 2.2323, Loss U: 0.0059, Loss C: 0.0262, Acc (L): 0.3500 | Val Loss: 2.0136, Val Acc: 0.4525 | Mask Ratio: 0.045
```

{"model_id":"b2d82c25f2be4830b42be9586f18633d","version_major":2,"version_minor":0}

{"model_id":"3d406da20e2d4b509c12c66622a13226","version_major":2,"version_minor":0}

```
  Epoch 18/50 -> Loss S: 2.0563, Loss U: 0.0122, Loss C: 0.0318, Acc
(L): 0.4917 | Val Loss: 2.0211, Val Acc: 0.4705 | Mask Ratio: 0.046
```

{"model_id":"5d7816cba15e4edd9289f3209c3c8d88","version_major":2,"version_minor":0}

{"model_id":"43a62759a6094dd78893fe67c1d04c7b","version_major":2,"version_minor":0}

```
  Epoch 19/50 -> Loss S: 2.2128, Loss U: 0.0168, Loss C: 0.0325, Acc
(L): 0.4000 | Val Loss: 1.7792, Val Acc: 0.5137 | Mask Ratio: 0.052
```

{"model_id":"9a1c730c73a24685a0b3d31342284ba9","version_major":2,"version_minor":0}

{"model_id":"82428b16f5d746dbbe4b80a546c61b85","version_major":2,"version_minor":0}

```
  Epoch 20/50 -> Loss S: 1.9642, Loss U: 0.0125, Loss C: 0.0332, Acc
(L): 0.4417 | Val Loss: 2.1731, Val Acc: 0.4441 | Mask Ratio: 0.061
```

{"model_id":"7f9333a59e724ff1b9841584b965bbc5","version_major":2,"version_minor":0}

{"model_id":"ff0dcd6870af4164a7eca7aea088984e","version_major":2,"version_minor":0}

```
  Epoch 21/50 -> Loss S: 2.1083, Loss U: 0.0356, Loss C: 0.0628, Acc
(L): 0.4125 | Val Loss: 2.1433, Val Acc: 0.4494 | Mask Ratio: 0.082
```

{"model_id":"dba651b201b941bb8b4718d7492c3a1b","version_major":2,"version_minor":0}

{"model_id":"207c98e5db4e4b8f9cf16bafb824a620","version_major":2,"version_minor":0}

```
  Epoch 22/50 -> Loss S: 2.1497, Loss U: 0.0254, Loss C: 0.0513, Acc
(L): 0.3917 | Val Loss: 1.9519, Val Acc: 0.4842 | Mask Ratio: 0.083
```

{"model_id":"e05783e37f7942e585868195151054bd","version_major":2,"version_minor":0}

{"model_id":"4212648d424749c9ab3292e3fe0dad67","version_major":2,"version_minor":0}

```
  Epoch 23/50 -> Loss S: 2.2072, Loss U: 0.0193, Loss C: 0.0430, Acc
(L): 0.3667 | Val Loss: 1.8269, Val Acc: 0.4863 | Mask Ratio: 0.065
```

{"model_id":"ce65cbf127d143b0b006e1605df0fc3b","version_major":2,"version_minor":0}

{"model_id":"59d32aa5437c40ba9cf66cee53550b72","version_major":2,"version_minor":0}

```
  Epoch 24/50 -> Loss S: 1.9806, Loss U: 0.0262, Loss C: 0.0405, Acc
(L): 0.4417 | Val Loss: 1.7558, Val Acc: 0.5084 | Mask Ratio: 0.082
```

{"model_id":"4d59289f493a4e7680bd70a6496714dc","version_major":2,"version_minor":0}

{"model_id":"838ad359353f426b82e1be2c242a0f5f","version_major":2,"version_minor":0}

```
  Epoch 25/50 -> Loss S: 1.9024, Loss U: 0.0305, Loss C: 0.0399, Acc
(L): 0.5375 | Val Loss: 1.7735, Val Acc: 0.5380 | Mask Ratio: 0.095
```

{"model_id":"ca5aa87c99e24d1fa955f04292ce0614","version_major":2,"version_minor":0}

{"model_id":"358609969f454d829b57a5a29f9ab245","version_major":2,"version_minor":0}

```
  Epoch 26/50 -> Loss S: 1.9919, Loss U: 0.0270, Loss C: 0.0538, Acc
(L): 0.4375 | Val Loss: 1.7882, Val Acc: 0.5137 | Mask Ratio: 0.111
```

{"model_id":"66bf49a30fc7453c84c20f21bf28668b","version_major":2,"version_minor":0}

{"model_id":"6e29e00ad5824f81804a1591fcd6fab8","version_major":2,"version_minor":0}

```
  Epoch 27/50 -> Loss S: 1.9084, Loss U: 0.0267, Loss C: 0.0362, Acc
(L): 0.4833 | Val Loss: 1.6977, Val Acc: 0.5475 | Mask Ratio: 0.104
```

{"model_id":"ec0f88df2bac4e18a77199d39ff32ca4","version_major":2,"version_minor":0}

{"model_id":"dc69e201e82a4536827763bc0a29de6b","version_major":2,"version_minor":0}

```
  Epoch 28/50 -> Loss S: 1.7305, Loss U: 0.0342, Loss C: 0.0530, Acc
(L): 0.5208 | Val Loss: 1.6057, Val Acc: 0.5770 | Mask Ratio: 0.121
```

{"model_id":"ee55f720f57342cd9abc5b72b556005d","version_major":2,"version_minor":0}

{"model_id":"ba761a7af4014bdab31c94c9223a3c7c","version_major":2,"version_minor":0}

```
  Epoch 29/50 -> Loss S: 2.0233, Loss U: 0.0199, Loss C: 0.0516, Acc
(L): 0.4750 | Val Loss: 1.7901, Val Acc: 0.5580 | Mask Ratio: 0.135
```

{"model_id":"a7a78beae9934ed492027a9101da5a08","version_major":2,"version_minor":0}

{"model_id":"410dc7c0e1c74ca186464740d937febd","version_major":2,"version_minor":0}

```
  Epoch 30/50 -> Loss S: 1.9776, Loss U: 0.0332, Loss C: 0.0424, Acc (L): 0.4917 | Val Loss: 1.9131, Val Acc: 0.4937 | Mask Ratio: 0.126
```

{"model_id":"49fe409ebdcc4473baeba74a011e97ff","version_major":2,"version_minor":0}

{"model_id":"f956b020e51142e2aef8e5c16ec16fdc","version_major":2,"version_minor":0}

```
  Epoch 31/50 -> Loss S: 1.9486, Loss U: 0.0359, Loss C: 0.0425, Acc (L): 0.4750 | Val Loss: 1.6758, Val Acc: 0.5517 | Mask Ratio: 0.123
```

{"model_id":"bc479d39bb5146ce99807881d308e6cb","version_major":2,"version_minor":0}

{"model_id":"2ba633e2126e44f09d22d8933c998836","version_major":2,"version_minor":0}

```
  Epoch 32/50 -> Loss S: 1.7857, Loss U: 0.0455, Loss C: 0.0617, Acc (L): 0.5208 | Val Loss: 1.8522, Val Acc: 0.5274 | Mask Ratio: 0.143
```

{"model_id":"025beca2c27a4cc9aee0c41686be7046","version_major":2,"version_minor":0}

{"model_id":"fbc5ba88ed1d4b91b52cab991b0a3fc7","version_major":2,"version_minor":0}

```
  Epoch 33/50 -> Loss S: 1.7932, Loss U: 0.0389, Loss C: 0.0668, Acc (L): 0.5208 | Val Loss: 1.7019, Val Acc: 0.5506 | Mask Ratio: 0.154
```

{"model_id":"bb8455f80b0345e68ea02b81fd4ef7cc","version_major":2,"version_minor":0}

{"model_id":"f5966fa169fe4e609552775a19b9f46a","version_major":2,"version_minor":0}

```
  Epoch 34/50 -> Loss S: 1.6356, Loss U: 0.0483, Loss C: 0.0499, Acc (L): 0.5833 | Val Loss: 1.4810, Val Acc: 0.5981 | Mask Ratio: 0.170
```

{"model_id":"e95cae68ec4c420889b5e74815bdb9a2","version_major":2,"version_minor":0}

{"model_id":"7cc4f6af214a4e24a5bb33a2e69d3087","version_major":2,"version_minor":0}

```
  Epoch 35/50 -> Loss S: 1.7415, Loss U: 0.0376, Loss C: 0.0492, Acc (L): 0.5125 | Val Loss: 1.5659, Val Acc: 0.5854 | Mask Ratio: 0.164
```

{"model_id":"5ed51fe3727648edb7c06eac5403d06d","version_major":2,"version_minor":0}

{"model_id":"84a4d1e4abff4567bd516844a62e5eef","version_major":2,"version_minor":0}

```
  Epoch 36/50 -> Loss S: 1.7233, Loss U: 0.0406, Loss C: 0.0442, Acc
(L): 0.5500 | Val Loss: 2.0266, Val Acc: 0.5306 | Mask Ratio: 0.168
```

{"model_id":"5d7d83e65f6d406dac89d16499e79c8e","version_major":2,"version_minor":0}

{"model_id":"4f842911c2af40ab8f37c0ced7ab89aa","version_major":2,"version_minor":0}

```
  Epoch 37/50 -> Loss S: 1.6260, Loss U: 0.0348, Loss C: 0.0577, Acc
(L): 0.5833 | Val Loss: 1.5001, Val Acc: 0.5865 | Mask Ratio: 0.174
```

{"model_id":"a09a61997822449a8c0887b9329968aa","version_major":2,"version_minor":0}

{"model_id":"07a556e1b1f548a0be9faa06fe94d5f1","version_major":2,"version_minor":0}

```
  Epoch 38/50 -> Loss S: 1.7170, Loss U: 0.0428, Loss C: 0.0546, Acc
(L): 0.5292 | Val Loss: 1.5520, Val Acc: 0.5918 | Mask Ratio: 0.171
```

{"model_id":"adee4876510742b091a0bc7325db93f4","version_major":2,"version_minor":0}

{"model_id":"fc051e9f296948d7883496419ff91f7e","version_major":2,"version_minor":0}

```
  Epoch 39/50 -> Loss S: 1.7350, Loss U: 0.0445, Loss C: 0.0688, Acc
(L): 0.5292 | Val Loss: 1.4701, Val Acc: 0.6076 | Mask Ratio: 0.218
```

{"model_id":"95790b7768ef497fbdb909d607832d06","version_major":2,"version_minor":0}

{"model_id":"e709680a8e9e4e42a715b635a6f0f4d1","version_major":2,"version_minor":0}

```
  Epoch 40/50 -> Loss S: 1.5503, Loss U: 0.0503, Loss C: 0.0644, Acc
(L): 0.5500 | Val Loss: 1.5472, Val Acc: 0.6086 | Mask Ratio: 0.221
```

{"model_id":"a7a76ff7cbfb446595d37a8df60a820e","version_major":2,"version_minor":0}

{"model_id":"cbc4bb1f72be4db9aeaf31ca8dd9b4b5","version_major":2,"version_minor":0}

```
  Epoch 41/50 -> Loss S: 1.6602, Loss U: 0.0468, Loss C: 0.0543, Acc
(L): 0.5583 | Val Loss: 1.6240, Val Acc: 0.5897 | Mask Ratio: 0.208
```

{"model_id":"8c852df0a9ea4d1fb9cb7ed020614914","version_major":2,"version_minor":0}

{"model_id":"f70fce2ef6604ee2b9d529198615ab11","version_major":2,"version_minor":0}

```
  Epoch 42/50 -> Loss S: 1.7981, Loss U: 0.0474, Loss C: 0.0508, Acc
(L): 0.5417 | Val Loss: 1.4380, Val Acc: 0.6234 | Mask Ratio: 0.206
```

{"model_id":"75d79294e114489c8dd114dfca9f1092","version_major":2,"version_minor":0}

{"model_id":"6d87a4e6ca9c444db8977c53f71a00cc","version_major":2,"version_minor":0}

```
  Epoch 43/50 -> Loss S: 1.5130, Loss U: 0.0493, Loss C: 0.0593, Acc
(L): 0.5625 | Val Loss: 1.5522, Val Acc: 0.5992 | Mask Ratio: 0.234
```

{"model_id":"c099797f385d44a4a766e8ecc9538407","version_major":2,"version_minor":0}

{"model_id":"4dc7c63546874d1e899d3a1426fb5406","version_major":2,"version_minor":0}

```
  Epoch 44/50 -> Loss S: 1.5703, Loss U: 0.0558, Loss C: 0.0607, Acc
(L): 0.5583 | Val Loss: 1.3431, Val Acc: 0.6308 | Mask Ratio: 0.233
```

{"model_id":"be0d0f6e4d94461cbe9ffbb4e5258576","version_major":2,"version_minor":0}

{"model_id":"31a454756baf43cb88c94a93fbaeddf5","version_major":2,"version_minor":0}

```
  Epoch 45/50 -> Loss S: 1.3231, Loss U: 0.0593, Loss C: 0.0694, Acc
(L): 0.6333 | Val Loss: 1.3313, Val Acc: 0.6466 | Mask Ratio: 0.242
```

{"model_id":"3c77882fe67646f29ab2de43125247cb","version_major":2,"version_minor":0}

{"model_id":"741bfd55e12340cf898ca5ba621649ac","version_major":2,"version_minor":0}

```
  Epoch 46/50 -> Loss S: 1.6282, Loss U: 0.0546, Loss C: 0.0744, Acc
(L): 0.5625 | Val Loss: 1.5014, Val Acc: 0.6350 | Mask Ratio: 0.274
```

{"model_id":"89b46a02e6c44b878a410ff2b17fd817","version_major":2,"version_minor":0}

{"model_id":"43bae8d9e1834cf0b063cd42c67c114d","version_major":2,"version_minor":0}

```
  Epoch 47/50 -> Loss S: 1.4605, Loss U: 0.0485, Loss C: 0.0664, Acc
(L): 0.6083 | Val Loss: 1.3691, Val Acc: 0.6456 | Mask Ratio: 0.268
```

{"model_id":"58cc80ed1d044bd18ef2293e5d07162a","version_major":2,"version_minor":0}

{"model_id":"175919fc9b824d1dbf68880debb35747","version_major":2,"version_minor":0}

```
  Epoch 48/50 -> Loss S: 1.6183, Loss U: 0.0534, Loss C: 0.0781, Acc
(L): 0.5833 | Val Loss: 1.4073, Val Acc: 0.6308 | Mask Ratio: 0.251
```

{"model_id":"7af4388b3877494c9b72c39ede4b1292","version_major":2,"version_minor":0}

{"model_id":"c282d60351764bc8b05f5c1c3de9e3e6","version_major":2,"version_minor":0}

```
  Epoch 49/50 -> Loss S: 1.6299, Loss U: 0.0608, Loss C: 0.0533, Acc
(L): 0.5375 | Val Loss: 1.4010, Val Acc: 0.6477 | Mask Ratio: 0.245
```

{"model_id":"d45b3b3a285d46b09a192dd2a3a44ebb","version_major":2,"version_minor":0}

{"model_id":"703dd65efe444aaf99c23316128842f3","version_major":2,"version_minor":0}

```
  Epoch 50/50 -> Loss S: 1.4980, Loss U: 0.0502, Loss C: 0.0528, Acc
(L): 0.6042 | Val Loss: 1.4794, Val Acc: 0.6181 | Mask Ratio: 0.252

Finished training across all label percentages.
Best overall validation accuracy: 0.7300 achieved with 20% labeled
data.
Saving the best overall model state to best_epass_simmatch_model.pth

Loading best overall model for final evaluation...
Best model loaded successfully.
```

9. Evaluate on Test Set and Compute Metrics

- Use the loaded best-performing model state.
- Run `evaluate` on the `test_loader` (if available).
- Print final metrics.

```
# -----------------------------------------
# 9. Evaluate on Test Set and Compute Metrics
# -----------------------------------------

if test_loader is not None and best_model_state is not None:
    print("\nEvaluating the best model on the Test Set...")
    test_loss, test_acc, y_pred_test, y_true_test = evaluate(model,
test_loader, criterion_s, config.device)
    print(f"\nFinal Test Results using Best Overall Model (Val Acc:
{best_val_acc_overall:.4f}, Labeled: {best_labeled_percent*100:.0f}
%):")
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy:
{test_acc:.4f}")
```

```
    print("\nClassification Report on Test Set:")
    # Use idx_to_label to get class names
    target_names = [idx_to_label[i] for i in
range(config.num_classes)]
    print(classification_report(y_true_test, y_pred_test,
target_names=target_names, digits=4))

    print("\nConfusion Matrix on Test Set:")
    cm = confusion_matrix(y_true_test, y_pred_test)
    plt.figure(figsize=(12, 10))
    sns.heatmap(cm, annot=False, fmt='d', xticklabels=target_names,
yticklabels=target_names, cmap='Blues') # Annot=False for large
matrices
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title("Confusion Matrix on Test Data (Best EPASS+SimMatch
Model)")
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()
elif test_loader is None:
    print("\nSkipping final test evaluation as test data/labels were
not available.")
else: # best_model_state is None
    print("\nSkipping final test evaluation as no best model was
saved.")
```

```
Skipping final test evaluation as test data/labels were not available.
```

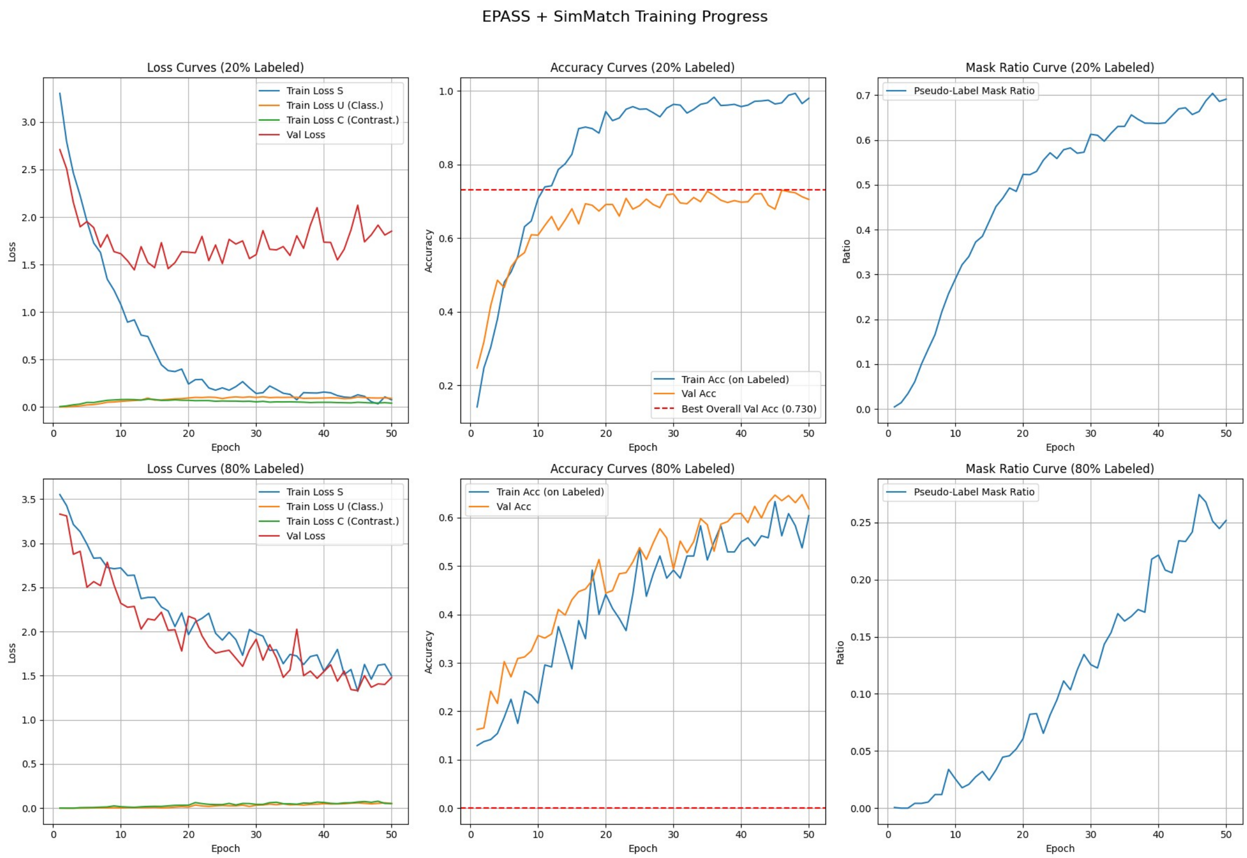
10. Plot Training Curves

- Plot losses (Supervised, Unsupervised Classification, Contrastive) and accuracies (Train Labeled, Validation) for **each** labeled percentage run to show overfitting/underfitting trends under different supervision levels.

```
# ----------------------------
# 10. Plot Training Curves
# ----------------------------
num_runs = len(config.labeled_percents)
fig, axes = plt.subplots(num_runs, 3, figsize=(18, 6 * num_runs),
squeeze=False)

for i, percent in enumerate(config.labeled_percents):
    run_history = history[percent]
    epochs_range = range(1, len(run_history['train_loss_s']) + 1)

    # Plot Losses
    ax = axes[i, 0]
```

```
    ax.plot(epochs_range, run_history['train_loss_s'], label='Train
Loss S')
    ax.plot(epochs_range, run_history['train_loss_u'], label='Train
Loss U (Class.)')
    ax.plot(epochs_range, run_history['train_loss_c'], label='Train
Loss C (Contrast.)')
    ax.plot(epochs_range, run_history['val_loss'], label='Val Loss')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    ax.set_title(f'Loss Curves ({percent*100:.0f}% Labeled)')
    ax.legend()
    ax.grid(True)

    # Plot Accuracies
    ax = axes[i, 1]
    ax.plot(epochs_range, run_history['train_acc_l'], label='Train Acc
(on Labeled)')
    ax.plot(epochs_range, run_history['val_acc'], label='Val Acc')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Accuracy')
    ax.set_title(f'Accuracy Curves ({percent*100:.0f}% Labeled)')
    ax.legend()
    ax.grid(True)
    ax.axhline(y=best_val_acc_overall if best_labeled_percent ==
percent else 0, color='r', linestyle='--', label=f'Best Overall Val
Acc ({best_val_acc_overall:.3f})' if best_labeled_percent == percent
else None)
    if best_labeled_percent == percent: ax.legend() # Show legend only
if this run was best

    # Plot Mask Ratio
    ax = axes[i, 2]
    ax.plot(epochs_range, run_history['mask_ratio'], label='Pseudo-
Label Mask Ratio')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Ratio')
    ax.set_title(f'Mask Ratio Curve ({percent*100:.0f}% Labeled)')
    ax.legend()
    ax.grid(True)

plt.suptitle('EPASS + SimMatch Training Progress', fontsize=16,
y=1.02)
plt.tight_layout()
plt.show()

print("\n--- Analysis of Curves ---")
print("Overfitting: Indicated if validation accuracy
plateaus/decreases while training accuracy continues to rise, or if
validation loss increases while training loss decreases.")
print("Underfitting: Indicated if both training and validation
```

```
accuracies are low and plateau early, or if losses remain high.")
print(f"Target Accuracy (~80%): Observe if the best validation
accuracy ({best_val_acc_overall:.4f}) reached the target.")
```




```
--- Analysis of Curves ---
Overfitting: Indicated if validation accuracy plateaus/decreases while
training accuracy continues to rise, or if validation loss increases
while training loss decreases.
Underfitting: Indicated if both training and validation accuracies are
low and plateau early, or if losses remain high.
Target Accuracy (~80%): Observe if the best validation accuracy
(0.7300) reached the target.
```