```python
"""
train_model.py
==================================================
Description:
    Defines the training configuration, default hyperparameters, and execution
    entry point for the Neural Sequence Decoder. Implements command-line
    argument parsing for hyperparameter sweeps (ablation studies) and
    initiates the training process.

    Part of the final project for ECE C143A/C243A at UCLA.

Authors:
    Yihe Xu, Injun Choi, Saiya Shah

Date:
    December 1, 2025

Location:
    neural_seq_decoder/scripts/train_model.py
"""


import sys

modelName = 'speechBaseline0'

args = {}
args['outputDir'] = '../src_bk/speech_logs/' + modelName
args['datasetPath'] = '../src_bk/ptDecoder_ctc.pkl'
args['seqLen'] = 150
args['maxTimeSeriesLen'] = 1200
args['batchSize'] = 64
args['lrStart'] = 0.02
args['lrEnd'] = 0.02
args['nUnits'] = 1024
args['nBatch'] = 10000
args['nLayers'] = 5
args['seed'] = 0
args['nClasses'] = 40
args['nInputFeatures'] = 256
args['dropout'] = 0.4
args['whiteNoiseSD'] = 0.8
args['constantOffsetSD'] = 0.2
args['gaussianSmoothWidth'] = 2.0
args['strideLen'] = 4
args['kernelLen'] = 32
args['bidirectional'] = False
args['l2_decay'] = 1e-5

# ---- NEW AUGMENTATION SETTINGS ----
args['timeMask'] = -1       # maximum mask length
args['timeMask_N'] = -1     # number of masks
args['channelDrop'] = 0.4     # drop 20% of channels


argv = sys.argv

def get_arg(flag, cast_type, default):
    """Return parsed CLI argument or fallback to default."""
    if flag in argv:
        idx = argv.index(flag)
        if idx + 1 < len(argv):
            return cast_type(argv[idx + 1])
    return default

```

```
# map the flags your sweep sends into your args dictionary
args['whiteNoiseSD']      = get_arg("--white_std",    float, args['whiteNoiseSD'])
args['constantOffsetSD']  = get_arg("--mean_std",     float, args['constantOffsetSD'])
args['gaussianSmoothWidth'] = get_arg("--gauss_sigma", float, args['gaussianSmoothWidth'])
args['kernelLen']           = get_arg("--gauss_kernel", int,   args['kernelLen'])
args['timeMask']            = get_arg("--mask_max_len", int,   args['timeMask'])
args['timeMask_N']          = get_arg("--mask_n",       int,   args['timeMask_N'])  # NEW
args['channelDrop']         = get_arg("--drop_p",       float, args['channelDrop'])


from neural_decoder.neural_decoder_trainer import trainModel

trainModel(args)
```

```python
"""
augmentation.py
==================================================
Description:
    Defines PyTorch modules for data augmentation and signal processing
    specifically tailored for neural sequence decoding.

    Includes:
    - WhiteNoise: Additive Gaussian noise.
    - MeanDriftNoise: Channel-wise constant offset noise.
    - GaussianSmoothing: 1D/2D smoothing via convolution.
    - TimeMask: SpecAugment-style temporal masking.
    - ChannelDropout: Randomly drops input channels (simulating electrode failure).

    Part of the final project for ECE C143A/C243A at UCLA.

Authors:
    Yihe Xu, Injun Choi, Saiya Shah

Date:
    December 1, 2025

Location:
    neural_seq_decoder/src/neural_decoder/augmentation.py
"""


import math
import numbers
import torch
from torch import nn
from torch.nn import functional as F


class WhiteNoise(nn.Module):
    def __init__(self, std=0.1):
        super().__init__()
        self.std = std

    def forward(self, x):
        noise = torch.randn_like(x) * self.std
        return x + noise

class MeanDriftNoise(nn.Module):
    def __init__(self, std=0.1):
        super().__init__()
        self.std = std

    def forward(self, x):
        _, C = x.shape
        noise = torch.randn(1, C) * self.std
        return x + noise

class GaussianSmoothing(nn.Module):
    """
    Apply gaussian smoothing on a
    1d, 2d or 3d tensor. Filtering is performed seperately for each channel
    in the input using a depthwise convolution.
    Arguments:
        channels (int, sequence): Number of channels of the input tensors. Output will
            have this number of channels as well.
        kernel_size (int, sequence): Size of the gaussian kernel.
        sigma (float, sequence): Standard deviation of the gaussian kernel.
        dim (int, optional): The number of dimensions of the data.
            Default value is 2 (spatial).
```

```
    """

    def __init__(self, channels, kernel_size, sigma, dim=2):
        super(GaussianSmoothing, self).__init__()
        if isinstance(kernel_size, numbers.Number):
            kernel_size = [kernel_size] * dim
        if isinstance(sigma, numbers.Number):
            sigma = [sigma] * dim

        # The gaussian kernel is the product of the
        # gaussian function of each dimension.
        kernel = 1
        meshgrids = torch.meshgrid(
            [torch.arange(size, dtype=torch.float32) for size in kernel_size]
        )
        for size, std, mgrid in zip(kernel_size, sigma, meshgrids):
            mean = (size - 1) / 2
            kernel *= (
                1
                / (std * math.sqrt(2 * math.pi))
                * torch.exp(-(((mgrid - mean) / std) ** 2) / 2)
            )

        # Make sure sum of values in gaussian kernel equals 1.
        kernel = kernel / torch.sum(kernel)

        # Reshape to depthwise convolutional weight
        kernel = kernel.view(1, 1, *kernel.size())
        kernel = kernel.repeat(channels, *[1] * (kernel.dim() - 1))

        self.register_buffer("weight", kernel)
        self.groups = channels

        if dim == 1:
            self.conv = F.conv1d
        elif dim == 2:
            self.conv = F.conv2d
        elif dim == 3:
            self.conv = F.conv3d
        else:
            raise RuntimeError(
                "Only 1, 2 and 3 dimensions are supported. Received {}.".format(dim)
            )

    def forward(self, input):
        """
        Apply gaussian filter to input.
        Arguments:
            input (torch.Tensor): Input to apply gaussian filter on.
        Returns:
            filtered (torch.Tensor): Filtered output.
        """
        return self.conv(input, weight=self.weight, groups=self.groups, padding="same")


class TimeMask(nn.Module):
    """
    SpecAugment-style time masking for neural time series.
    Randomly masks out consecutive time steps.

    Expected input shape: (C, T)
    - C = channels/electrodes
    - T = time steps
    """
    def __init__(self, max_mask_len=25, n_masks=2, mask_value=0.0):
        super().__init__()
        self.max_mask_len = max_mask_len
```

```
        self.n_masks = n_masks
        self.mask_value = mask_value

    def forward(self, x):
        """
        Args:
            x: Tensor (C, T)
        Returns:
            Masked tensor of same shape
        """
        if not self.training:
            return x

        C, T = x.shape
        x = x.clone()   # avoid in-place ops; safer for CTC & autograd

        for _ in range(self.n_masks):
            # random mask length
            mask_len = torch.randint(
                1, self.max_mask_len + 1,
                (1,), device=x.device
            ).item()

            # random start index
            max_start = max(1, T - mask_len)
            start = torch.randint(
                0, max_start,
                (1,), device=x.device
            ).item()

            # apply mask
            x[:, start:start + mask_len] = self.mask_value

        return x


class ChannelDropout(nn.Module):
    """
    Randomly drops entire channels (electrodes).
    Good for robustness to impedance spikes or bad contacts.
    """
    def __init__(self, p=0.1):
        super().__init__()
        self.p = p

    def forward(self, x):
        C = x.shape[0]
        drop_mask = (torch.rand(C) > self.p).float().to(x.device)
        drop_mask = drop_mask.unsqueeze(-1)
        return x * drop_mask

```

```python
"""
dataset.py
==================================================
Description:
    Defines the PyTorch Dataset class for loading neural activity and
    aligned phoneme sequences. Handles multi-day recording sessions
    by flattening the trial structure while preserving session (day)
    indices for domain adaptation.

    Part of the final project for ECE C143A/C243A at UCLA.

Authors:
    Yihe Xu, Injun Choi, Saiya Shah

Date:
    December 1, 2025

Location:
    neural_seq_decoder/src/neural_decoder/dataset.py
"""


import torch
from torch.utils.data import Dataset


class SpeechDataset(Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform
        self.n_days = len(data)
        self.n_trials = sum([len(d["sentenceDat"]) for d in data])

        self.neural_feats = []
        self.phone_seqs = []
        self.neural_time_bins = []
        self.phone_seq_lens = []
        self.days = []
        for day in range(self.n_days):
            for trial in range(len(data[day]["sentenceDat"])):
                self.neural_feats.append(data[day]["sentenceDat"][trial])
                self.phone_seqs.append(data[day]["phonemes"][trial])
                self.neural_time_bins.append(data[day]["sentenceDat"][trial].shape[0])
                self.phone_seq_lens.append(data[day]["phoneLens"][trial])
                self.days.append(day)

    def __len__(self):
        return self.n_trials

    def __getitem__(self, idx):
        neural_feats = torch.tensor(self.neural_feats[idx], dtype=torch.float32)

        if self.transform:
            neural_feats = self.transform(neural_feats)

        return (
            neural_feats,
            torch.tensor(self.phone_seqs[idx], dtype=torch.int32),
            torch.tensor(self.neural_time_bins[idx], dtype=torch.int32),
            torch.tensor(self.phone_seq_lens[idx], dtype=torch.int32),
            torch.tensor(self.days[idx], dtype=torch.int64),
        )

```

```
"""
model.py
==================================================
Description:
    Defines the model architecture for the Neural Sequence Decoder.
    Part of the final project for ECE C143A/C243A at UCLA.

Authors:
    Yihe Xu, Injun Choi, Saiya Shah

Date:
    December 1, 2025

Location:
    neural_seq_decoder/src/neural_decoder/model.py
"""


import torch
from torch import nn
from .augmentations import GaussianSmoothing

class GradReverse(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, lambd):
        ctx.lambd = lambd
        return x.view_as(x)

    @staticmethod
    def backward(ctx, grad_output):
        return -ctx.lambd * grad_output, None

def grad_reverse(x, lambd=1.0):
    return GradReverse.apply(x, lambd)

class GRUDecoder(nn.Module):
    def __init__(
        self,
        neural_dim,
        n_classes,
        hidden_dim,
        layer_dim,
        nDays=24,
        dropout=0,
        device="cuda",
        strideLen=4,
        kernelLen=14,
        gaussianSmoothWidth=0,
        bidirectional=False,
    ):
        super(GRUDecoder, self).__init__()

        # Define day classifier
        feat_dim = hidden_dim * (2 if bidirectional else 1)
        self.day_classifier = nn.Sequential(
            nn.Linear(feat_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, nDays),
        )

        # Defining the number of layers and the nodes in each layer
        self.layer_dim = layer_dim
        self.hidden_dim = hidden_dim
        self.neural_dim = neural_dim
        self.n_classes = n_classes
```

```
        self.nDays = nDays
        self.device = device
        self.dropout = dropout
        self.strideLen = strideLen
        self.kernelLen = kernelLen
        self.gaussianSmoothWidth = gaussianSmoothWidth
        self.bidirectional = bidirectional
        self.inputLayerNonlinearity = torch.nn.Softsign()
        self.unfolder = torch.nn.Unfold(
            (self.kernelLen, 1), dilation=1, padding=0, stride=self.strideLen
        )
        self.gaussianSmoother = GaussianSmoothing(
            neural_dim, 20, self.gaussianSmoothWidth, dim=1
        )
        self.dayWeights = torch.nn.Parameter(torch.randn(nDays, neural_dim, neural_dim))
        self.dayBias = torch.nn.Parameter(torch.zeros(nDays, 1, neural_dim))

        for x in range(nDays):
            self.dayWeights.data[x, :, :] = torch.eye(neural_dim)

        # GRU layers
        self.gru_decoder = nn.GRU(
            (neural_dim) * self.kernelLen,
            hidden_dim,
            layer_dim,
            batch_first=True,
            dropout=self.dropout,
            bidirectional=self.bidirectional,
        )

        for name, param in self.gru_decoder.named_parameters():
            if "weight_hh" in name:
                nn.init.orthogonal_(param)
            if "weight_ih" in name:
                nn.init.xavier_uniform_(param)

        # Input layers
        for x in range(nDays):
            setattr(self, "inpLayer" + str(x), nn.Linear(neural_dim, neural_dim))

        for x in range(nDays):
            thisLayer = getattr(self, "inpLayer" + str(x))
            thisLayer.weight = torch.nn.Parameter(
                thisLayer.weight + torch.eye(neural_dim)
            )

        # rnn outputs
        if self.bidirectional:
            self.fc_decoder_out = nn.Linear(
                hidden_dim * 2, n_classes + 1
            )  # +1 for CTC blank
        else:
            self.fc_decoder_out = nn.Linear(hidden_dim, n_classes + 1)  # +1 for CTC blank

    def forward(self, neuralInput, dayIdx):
        neuralInput = torch.permute(neuralInput, (0, 2, 1))
        neuralInput = self.gaussianSmoother(neuralInput)
        neuralInput = torch.permute(neuralInput, (0, 2, 1))

        # apply day layer
        dayWeights = torch.index_select(self.dayWeights, 0, dayIdx)
        transformedNeural = torch.einsum(
            "btd,bdk->btk", neuralInput, dayWeights
        ) + torch.index_select(self.dayBias, 0, dayIdx)
        transformedNeural = self.inputLayerNonlinearity(transformedNeural)

        # stride/kernel
```

```
        stridedInputs = torch.permute(
            self.unfolder(
                torch.unsqueeze(torch.permute(transformedNeural, (0, 2, 1)), 3)
            ),
            (0, 2, 1),
        )

        # apply RNN layer
        if self.bidirectional:
            h0 = torch.zeros(
                self.layer_dim * 2,
                transformedNeural.size(0),
                self.hidden_dim,
                device=self.device,
            ).requires_grad_()
        else:
            h0 = torch.zeros(
                self.layer_dim,
                transformedNeural.size(0),
                self.hidden_dim,
                device=self.device,
            ).requires_grad_()

        hid, _ = self.gru_decoder(stridedInputs, h0.detach())

        # get seq
        seq_out = self.fc_decoder_out(hid)
        rev = grad_reverse(hid, lambd=1.0)
        day_logits = self.day_classifier(rev)
        return seq_out, day_logits, hid
        # return seq_out

```

```python
"""
neural_decoder_trainer.py
==================================================
Description:
    Implements the core training loop, validation logic, and data loading
    utilities for the Neural Sequence Decoder.

    Key features include:
    1. SWATS Optimization: Automates the switch from Adam to SGD based on
       projected learning rates to improve generalization.
    2. Domain Adversarial Training: Integrates gradient reversal for
       day/domain independence.
    3. Real-time Evaluation: Monitors Character Error Rate (CER) and
       Connectionist Temporal Classification (CTC) loss.

    Part of the final project for ECE C143A/C243A at UCLA.

Authors:
    Yihe Xu, Injun Choi, Saiya Shah

Date:
    December 1, 2025

Location:
    neural_seq_decoder/src/neural_decoder/neural_decoder_trainer.py
"""


import os
import pickle
import time

from edit_distance import SequenceMatcher
import hydra
import numpy as np
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader
import torch.nn.functional as F

from .model import GRUDecoder, grad_reverse
from .dataset import SpeechDataset

import torch.optim.lr_scheduler

# --- PATCH START: Fix for PyTorch < 2.0 ---
if not hasattr(torch.optim.lr_scheduler, "LRScheduler"):
    # Create the alias 'LRScheduler' pointing to the hidden '_LRScheduler'
    torch.optim.lr_scheduler.LRScheduler = torch.optim.lr_scheduler._LRScheduler
# --- PATCH END ---

# Now you can safely import from the library
from pytorch_optimizer import Lookahead

def get_smart_sgd_lr(adam_optimizer, beta2=0.999):
    """
    Calculates the global SGD learning rate by projecting the
    adaptive learning rates of Adam onto a single scalar.
    """
    print("[SWATS] Calculating projection from Adam state...")

    # 1. Get the current step (needed for bias correction)
    step = 0
    for group in adam_optimizer.param_groups:
        for p in group['params']:
```

```
            if p.grad is None: continue
            state = adam_optimizer.state[p]
            if 'step' in state:
                step = state['step']
                if isinstance(step, torch.Tensor):
                    step = step.item()
                break
        if step > 0: break

    if step == 0:
        print("[SWATS] Warning: Optimizer has no step count. Defaulting to 0.01")
        return 0.01

    # 2. Calculate Bias Correction for the Second Moment (v_t)
    bias_correction = 1 - beta2 ** step

    effective_lrs = []

    # 3. Iterate over all parameters to find mean effective step size
    for group in adam_optimizer.param_groups:
        base_lr = group['lr']
        eps = group['eps'] # Use the epsilon defined in the optimizer

        for p in group['params']:
            if p.grad is None: continue

            state = adam_optimizer.state[p]
            if 'exp_avg_sq' not in state: continue

            # Get the variance vector (v)
            v = state['exp_avg_sq']

            # Correct bias: v_hat = v / (1 - beta2^t)
            v_hat = v / bias_correction

            # Calculate Effective LR vector: base_lr / (sqrt(v_hat) + eps)
            # We assume the step direction is roughly consistent
            layer_effective_lr = base_lr / (torch.sqrt(v_hat) + eps)

            # We take the mean of this tensor to get a scalar for this parameter
            effective_lrs.append(layer_effective_lr.mean().item())

    if len(effective_lrs) == 0:
        return 0.01

    # 4. Project to a single scalar (Arithmetic Mean of all effective LRs)
    projected_lr = np.mean(effective_lrs)
    return projected_lr


def getDatasetLoaders(
    datasetName,
    batchSize,
):
    with open(datasetName, "rb") as handle:
        loadedData = pickle.load(handle)

    def _padding(batch):
        X, y, X_lens, y_lens, days = zip(*batch)
        X_padded = pad_sequence(X, batch_first=True, padding_value=0)
        y_padded = pad_sequence(y, batch_first=True, padding_value=0)

        return (
            X_padded,
            y_padded,
            torch.stack(X_lens),
            torch.stack(y_lens),
```

```
            torch.stack(days),
        )

    train_ds = SpeechDataset(loadedData["train"], transform=None)
    test_ds = SpeechDataset(loadedData["test"])

    train_loader = DataLoader(
        train_ds,
        batch_size=batchSize,
        shuffle=True,
        num_workers=0,
        pin_memory=True,
        collate_fn=_padding,
    )
    test_loader = DataLoader(
        test_ds,
        batch_size=batchSize,
        shuffle=False,
        num_workers=0,
        pin_memory=True,
        collate_fn=_padding,
    )

    return train_loader, test_loader, loadedData

def trainModel(args):
    os.makedirs(args["outputDir"], exist_ok=True)
    torch.manual_seed(args["seed"])
    np.random.seed(args["seed"])
    device = "cuda"

    with open(args["outputDir"] + "/args", "wb") as file:
        pickle.dump(args, file)

    trainLoader, testLoader, loadedData = getDatasetLoaders(
        args["datasetPath"],
        args["batchSize"],
    )

    model = GRUDecoder(
        neural_dim=args["nInputFeatures"],
        n_classes=args["nClasses"],
        hidden_dim=args["nUnits"],
        layer_dim=args["nLayers"],
        nDays=len(loadedData["train"]),
        dropout=args["dropout"],
        device=device,
        strideLen=args["strideLen"],
        kernelLen=args["kernelLen"],
        gaussianSmoothWidth=args["gaussianSmoothWidth"],
        bidirectional=args["bidirectional"],
    ).to(device)

    loss_ctc = torch.nn.CTCLoss(blank=0, reduction="mean", zero_infinity=True)
    optimizer = torch.optim.Adam(
        model.parameters(),
        lr=args["lrStart"],
        betas=(0.9, 0.999),
        eps=0.1,
        weight_decay=args["l2_decay"],
    )

    # Learning Rate Scheduler: Linear Decay from lrStart to lrEnd
    scheduler = torch.optim.lr_scheduler.LinearLR(
        optimizer,
        start_factor=1.0,
        end_factor=args["lrEnd"] / args["lrStart"],
```

```
            total_iters=args["nBatch"],
        )

        # --- SWATS CONFIGURATION ---
        # Switch at 40% or 50% of batches usually works best for projection
        SWITCH_BATCH = int(args["nBatch"] * 0.3)
        switched_to_sgd = False

        # --train--
        testLoss = []
        testCER = []
        startTime = time.time()
        for batch in range(args["nBatch"]):
            if batch == SWITCH_BATCH and not switched_to_sgd:
                print("-" * 50)
                print(f"Batch {batch}: Initiating SWATS (Smart Switch to SGD)...")

                # A. Calculate the projected Learning Rate
                # We pass 0.999 as beta2 because that matches your Adam init above
                smart_sgd_lr = get_smart_sgd_lr(optimizer, beta2=0.999)

                print(f"Adam LR was: {args['lrStart']}")
                print(f"Projected SGD LR is: {smart_sgd_lr:.6f}")

                # B. Re-initialize Optimizer as SGD
                optimizer = torch.optim.SGD(
                    model.parameters(),
                    lr=smart_sgd_lr,
                    momentum=0.9, # Momentum is critical for SGD
                    weight_decay=args["l2_decay"],
                )

                # C. Re-initialize Scheduler
                # We need a new scheduler that decays from the NEW lr to the FINAL lr
                # over the remaining batches.
                remaining_iters = args["nBatch"] - batch

                # Calculate what the end factor should be relative to the new SGD start
                # If we want to end at args["lrEnd"], the factor is lrEnd / smart_sgd_lr
                new_end_factor = args["lrEnd"] / smart_sgd_lr

                scheduler = torch.optim.lr_scheduler.LinearLR(
                    optimizer,
                    start_factor=1.0,
                    end_factor=new_end_factor,
                    total_iters=remaining_iters,
                )

                switched_to_sgd = True
                print(f"Switched to SGD with LR={smart_sgd_lr:.5f}, Momentum=0.9")
                print("-" * 50)

            model.train()

            X, y, X_len, y_len, dayIdx = next(iter(trainLoader))
            X, y, X_len, y_len, dayIdx = (
                X.to(device),
                y.to(device),
                X_len.to(device),
                y_len.to(device),
                dayIdx.to(device),
            )

            # Noise augmentation is faster on GPU
            if args["whiteNoiseSD"] > 0:
                X += torch.randn(X.shape, device=device) * args["whiteNoiseSD"]

```

```
        if args["constantOffsetSD"] > 0:
            X += (
                torch.randn([X.shape[0], 1, X.shape[2]], device=device)
                * args["constantOffsetSD"]
            )

        pred, day_logits, hid = model.forward(X, dayIdx)

        ctc_loss = loss_ctc(
            torch.permute(pred.log_softmax(2), [1, 0, 2]),
            y,
            ((X_len - model.kernelLen) / model.strideLen).to(torch.int32),
            y_len,
        )
        ctc_loss = torch.sum(ctc_loss)

        # Day classification loss (adversarial)
        B, T, feat_dim = hid.shape
        rev_features = grad_reverse(hid, lambd=1.0)
        day_logits = model.day_classifier(rev_features.reshape(-1, feat_dim))
        day_targets = dayIdx.unsqueeze(1).repeat(1, T).reshape(-1)
        day_loss = F.cross_entropy(day_logits, day_targets)

        # Combined loss
        alpha = 0.1
        loss = ctc_loss + alpha * day_loss

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # print(endTime - startTime)

        # Eval
        if batch % 100 == 0:
            with torch.no_grad():
                model.eval()
                allLoss = []
                total_edit_distance = 0
                total_seq_length = 0
                for X, y, X_len, y_len, testDayIdx in testLoader:
                    X, y, X_len, y_len, testDayIdx = (
                        X.to(device),
                        y.to(device),
                        X_len.to(device),
                        y_len.to(device),
                        testDayIdx.to(device),
                    )

                    # pred = model.forward(X, testDayIdx)
                    pred, day_logits, hid = model.forward(X, testDayIdx)
                    loss = loss_ctc(
                        torch.permute(pred.log_softmax(2), [1, 0, 2]),
                        y,
                        ((X_len - model.kernelLen) / model.strideLen).to(torch.int32),
                        y_len,
                    )
                    loss = torch.sum(loss)
                    allLoss.append(loss.cpu().detach().numpy())

                    adjustedLens = ((X_len - model.kernelLen) / model.strideLen).to(
                        torch.int32
                    )
                    for iterIdx in range(pred.shape[0]):
                        decodedSeq = torch.argmax(
```

```
                                torch.tensor(pred[iterIdx, 0 : adjustedLens[iterIdx], :]),
                                dim=-1,
                            )  # [num_seq,]
                            decodedSeq = torch.unique_consecutive(decodedSeq, dim=-1)
                            decodedSeq = decodedSeq.cpu().detach().numpy()
                            decodedSeq = np.array([i for i in decodedSeq if i != 0])

                            trueSeq = np.array(
                                y[iterIdx][0 : y_len[iterIdx]].cpu().detach()
                            )

                            matcher = SequenceMatcher(
                                a=trueSeq.tolist(), b=decodedSeq.tolist()
                            )
                            total_edit_distance += matcher.distance()
                            total_seq_length += len(trueSeq)

                    avgDayLoss = np.sum(allLoss) / len(testLoader)
                    cer = total_edit_distance / total_seq_length

                    endTime = time.time()
                    print(
                        f"batch {batch}, ctc loss: {avgDayLoss:>7f}, cer: {cer:>7f}, time/batch: {(endTime - startTime)/100:>7.3f}"
                    )
                    startTime = time.time()

                if len(testCER) > 0 and cer < np.min(testCER):
                    torch.save(model.state_dict(), args["outputDir"] + "/modelWeights")
                testLoss.append(avgDayLoss)
                testCER.append(cer)

                tStats = {}
                tStats["testLoss"] = np.array(testLoss)
                tStats["testCER"] = np.array(testCER)

                with open(args["outputDir"] + "/trainingStats", "wb") as file:
                    pickle.dump(tStats, file)


def loadModel(modelDir, nInputLayers=24, device="cuda"):
    modelWeightPath = modelDir + "/modelWeights"
    with open(modelDir + "/args", "rb") as handle:
        args = pickle.load(handle)

    model = GRUDecoder(
        neural_dim=args["nInputFeatures"],
        n_classes=args["nClasses"],
        hidden_dim=args["nUnits"],
        layer_dim=args["nLayers"],
        nDays=nInputLayers,
        dropout=args["dropout"],
        device=device,
        strideLen=args["strideLen"],
        kernelLen=args["kernelLen"],
        gaussianSmoothWidth=args["gaussianSmoothWidth"],
        bidirectional=args["bidirectional"],
    ).to(device)

    model.load_state_dict(torch.load(modelWeightPath, map_location=device))
    return model


@hydra.main(version_base="1.1", config_path="conf", config_name="config")
def main(cfg):
    cfg.outputDir = os.getcwd()
    trainModel(cfg)
```

```
400
401 if __name__ == "__main__":
402     main()
```