301 lines - 51 Removals

```
import time
import optax

import flax.linen as nn
from malware_utils import *
from dataset import load_dataset
from flax.training import train_state
from HRR.with_flax import Binding, Unbinding, CosineSimilar
ity

from flax import jax_utils

from flax.training.common_utils import shard, shard_prng_ke
y









class MHAttention(nn.Module):
    features: int
    heads: int = 8

    def setup(self):
        self.binding = Binding()
        self.unbinding = Unbinding()
        self.similarity = CosineSimilarity()

    @nn.compact
    def __call__(self, query, key, value, mask=None):
        dense_q = nn.Dense(features=self.features)
        dense_k = nn.Dense(features=self.features)
        dense_v = nn.Dense(features=self.features)
        dense_o = nn.Dense(features=self.features)

        q = dense_q(query)  # (B, T, H)
        k = dense_k(key)  # (B, T, H)
        v = dense_v(value)  # (B, T, H)

        q = split(q, self.heads)  # (B, h, T, H')
        k = split(k, self.heads)  # (B, h, T, H')
        v = split(v, self.heads)  # (B, h, T, H')

        bind = self.binding(k, v, axis=-1)  # (B, h, T, H')
        bind = np.sum(bind, axis=-2, keepdims=True)  # (B,
 h, 1, H')

        vp = self.unbinding(bind, q, axis=-1)  # (B, h, T,
 H')
        scale = self.similarity(v, vp, axis=-1, keepdims=Tr
ue)  # (B, h, T, 1)

        scale = scale + (1. - mask) * (-1e9)
        weight = nn.softmax(scale, axis=-2)
        weighted_value = weight * v

        output = merge(weighted_value)
        output = dense_o(output)
        return output


class FeedForwardLayer(nn.Module):
    features: int
    dropout_rate: float = 0.0
    training: bool = False

    @nn.compact
    def __call__(self, inputs):
        x = nn.Dense(features=self.features)(inputs)
        x = nn.relu(x)
        x = nn.Dense(features=inputs.shape[-1])(x)
```

572 lines + 327 Additions

```
import time
import datetime

import flax.linen as nn
import optax




from HRR.with_flax import Binding, Unbinding, CosineSimilar
ity

from flax import jax_utils
from flax.training import train_state
from flax.training.common_utils import shard, shard_prng_ke
y

from MalwareDetectionHRR.initializations import *
from dataset import load_dataset
from malware_utils import *


class MHAttention(nn.Module):
    features: int
    heads: int = 8

    def setup(self):
        self.binding = Binding()
        self.unbinding = Unbinding()
        self.similarity = CosineSimilarity()

    @nn.compact
    def __call__(self, query, key, value, mask=None):
        dense_q = nn.Dense(features=self.features)
        dense_k = nn.Dense(features=self.features)
        dense_v = nn.Dense(features=self.features)
        dense_o = nn.Dense(features=self.features)

        q = dense_q(query)  # (B, T, H)
        k = dense_k(key)  # (B, T, H)
        v = dense_v(value)  # (B, T, H)

        q = split(q, self.heads)  # (B, h, T, H')
        k = split(k, self.heads)  # (B, h, T, H')
        v = split(v, self.heads)  # (B, h, T, H')

        bind = self.binding(k, v, axis=-1)  # (B, h, T, H')
        bind = np.sum(bind, axis=-2, keepdims=True)  # (B,
 h, 1, H')

        vp = self.unbinding(bind, q, axis=-1)  # (B, h, T,
 H')
        scale = self.similarity(v, vp, axis=-1, keepdims=Tr
ue)  # (B, h, T, 1)

        scale = scale + (1. - mask) * (-1e9)
        weight = nn.softmax(scale, axis=-2)
        weighted_value = weight * v

        output = merge(weighted_value)
        output = dense_o(output)
        return output


class FeedForwardLayer(nn.Module):
    features: int
    dropout_rate: float = 0.0
    training: bool = False

    @nn.compact
    def __call__(self, inputs):
        x = nn.Dense(features=self.features)(inputs)
        x = nn.relu(x)
        x = nn.Dense(features=inputs.shape[-1])(x)
```

```
        x = nn.Dropout(rate=self.dropout_rate)(x, deterministic=not self.training)
        return x


class Encoder(nn.Module):
    features: int
    dropout_rate: float
    training: bool = False

    @nn.compact
    def __call__(self, inputs, mask=None):
        lnx = nn.LayerNorm()(inputs)
        attention = MHAttention(features=inputs.shape[-1])(query=lnx, key=lnx, value=lnx, mask=mask)
        attention = nn.Dropout(self.dropout_rate, deterministic=not self.training)(attention)
        x = inputs + attention

        lnx = nn.LayerNorm()(x)
        ffn = FeedForwardLayer(self.features, self.dropout_rate, training=self.training)(lnx)
        outputs = x + ffn
        return outputs


class Embedding(nn.Module):
    vocab_size: int
    embed_size: int
    max_seq_len: int

    def setup(self):
        self.positions = np.arange(start=0, stop=self.max_seq_len, step=1)[np.newaxis, :]

    @nn.compact
    def __call__(self, inputs):
        word_embedding = nn.Embed(self.vocab_size, self.embed_size)(inputs)
        position_embedding = nn.Embed(self.max_seq_len, self.embed_size)(self.positions)
        return word_embedding + position_embedding


class Network(nn.Module):
    features: int
    vocab_size: int
    embed_size: int
    max_seq_len: int
    nlayers: int
    output_size: int
    dropout_rate: float
    training: bool

    @nn.compact
    def __call__(self, encoder_input):
        encoder_input = encoder_input.astype('int32')  # (B, T)

        en_mask = np.where(encoder_input > 0, 1., 0.)
        en_mask = en_mask[:, np.newaxis, :, np.newaxis]

        # embedding
        x = Embedding(vocab_size=self.vocab_size,
                      embed_size=self.embed_size,
                      max_seq_len=self.max_seq_len)(encoder_input)

        # class token
        token = self.param('cls_token', nn.initializers.lecun_normal(), (1, 1, self.embed_size))
        token = np.repeat(token, x.shape[0], axis=0)
        x = np.concatenate([token, x], axis=1)

        # adjust mask
        en_mask = np.concatenate([np.ones((en_mask.shape[0], 1, 1, 1)), en_mask], axis=-2)
```

```
        x = nn.Dropout(rate=self.dropout_rate)(x, deterministic=not self.training)
        return x


class Encoder(nn.Module):
    features: int
    dropout_rate: float
    training: bool = False

    @nn.compact
    def __call__(self, inputs, mask=None):
        lnx = nn.LayerNorm()(inputs)
        attention = MHAttention(features=inputs.shape[-1])(query=lnx, key=lnx, value=lnx, mask=mask)
        attention = nn.Dropout(self.dropout_rate, deterministic=not self.training)(attention)
        x = inputs + attention

        lnx = nn.LayerNorm()(x)
        ffn = FeedForwardLayer(self.features, self.dropout_rate, training=self.training)(lnx)
        outputs = x + ffn
        return outputs


class Embedding(nn.Module):
    vocab_size: int
    embed_size: int
    max_seq_len: int

    def setup(self):
        self.positions = np.arange(start=0, stop=self.max_seq_len, step=1)[np.newaxis, :]

    @nn.compact
    def __call__(self, inputs):
        word_embedding = nn.Embed(self.vocab_size, self.embed_size)(inputs)
        position_embedding = nn.Embed(self.max_seq_len, self.embed_size)(self.positions)
        return word_embedding + position_embedding


class Network(nn.Module):
    features: int
    vocab_size: int
    embed_size: int
    max_seq_len: int
    nlayers: int
    output_size: int
    dropout_rate: float
    training: bool

    @nn.compact
    def __call__(self, encoder_input):
        encoder_input = encoder_input.astype('int32')  # (B, T)

        en_mask = np.where(encoder_input > 0, 1., 0.)
        en_mask = en_mask[:, np.newaxis, :, np.newaxis]

        # embedding
        x = Embedding(vocab_size=self.vocab_size,
                      embed_size=self.embed_size,
                      max_seq_len=self.max_seq_len)(encoder_input)

        # class token
        token = self.param('cls_token', nn.initializers.lecun_normal(), (1, 1, self.embed_size))
        token = np.repeat(token, x.shape[0], axis=0)
        x = np.concatenate([token, x], axis=1)

        # adjust mask
        en_mask = np.concatenate([np.ones((en_mask.shape[0], 1, 1, 1)), en_mask], axis=-2)
```

```
        x = nn.Dropout(self.dropout_rate, deterministic=not self.training)(x)

        for i in range(self.nlayers):
            x = Encoder(features=self.features,
                        dropout_rate=self.dropout_rate,
                        training=self.training)(x, mask=en_mask)

        # output
        x = x[:, 0]

        x = nn.Dense(features=x.shape[-1])(x)
        x = nn.relu(x)

        output = nn.Dense(self.output_size)(x)
        output = nn.softmax(output, axis=-1)
        return output


def initialize_model(model, input_size, init_rngs):
    init_inputs = np.ones([1, input_size])
    variables = model.init(init_rngs, init_inputs)['params']
    return variables


def train_step(state, batch, rngs):
    """ train one step """
    y_true = batch[1]

    def loss_fn(params):
        y_pred = state.apply_fn({'params': params}, batch[0], rngs=rngs)
        loss = cross_entropy_loss(y_true=y_true, y_pred=y_pred)
        return loss, y_pred

    grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
    (loss, y_pred), grads = grad_fn(state.params)
    grads = jax.lax.pmean(grads, "batch")
    grads = grad_check(grads)
    state = state.apply_gradients(grads=grads)
    acc = accuracy(y_true=y_true, y_pred=y_pred)
    metrics = {'loss': loss, 'accuracy': acc}
    return state, metrics


def predict(state, batch, rngs):
    y_true = batch[1]
    y_pred = state.apply_fn({'params': state.params}, batch[0], rngs=rngs)
    loss = cross_entropy_loss(y_true=y_true, y_pred=y_pred)
    acc = accuracy(y_true=y_true, y_pred=y_pred)
    metrics = {'loss': loss, 'accuracy': acc}
    return metrics


def train(batch_size, max_seq_len, features=512, embed_size=256, lr=1e-3, epochs=10):
    batch_size = batch_size * jax.device_count()
    vocab_size = 256 + 1
    n_layer = 1
    output_size = 2
    dropout_rate = 0.1
    name = 'hrrformer'
```

```
        x = nn.Dropout(self.dropout_rate, deterministic=not self.training)(x)

        for i in range(self.nlayers):
            x = Encoder(features=self.features,
                        dropout_rate=self.dropout_rate,
                        training=self.training)(x, mask=en_mask)

        # output
        x = x[:, 0]

        x = nn.Dense(features=x.shape[-1])(x)
        x = nn.relu(x)

        output = nn.Dense(self.output_size)(x)
        output = nn.softmax(output, axis=-1)
        return output


def initialize_model(model, input_size, init_rngs):
    init_inputs = np.ones([1, input_size])
    variables = model.init(init_rngs, init_inputs)['params']
    return variables


def train_step(state, batch, rngs):
    """ train one step """
    y_true = batch[1]

    def loss_fn(params):
        y_pred = state.apply_fn({'params': params}, batch[0], rngs=rngs)
        loss = cross_entropy_loss(y_true=y_true, y_pred=y_pred)
        return loss, y_pred

    grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
    (loss, y_pred), grads = grad_fn(state.params)
    grads = jax.lax.pmean(grads, "batch")
    grads = grad_check(grads)
    state = state.apply_gradients(grads=grads)
    acc = accuracy(y_true=y_true, y_pred=y_pred)
    metrics = {'loss': loss, 'accuracy': acc}
    return state, metrics


def predict_ind(state, batch, rngs):
    y_true = batch[1]
    y_pred = state.apply_fn({'params': state.params}, batch[0], rngs=rngs)
    loss = cross_entropy_loss(y_true=y_true, y_pred=y_pred)
    acc = accuracy(y_true=y_true, y_pred=y_pred)
    metrics = {'loss': loss, 'accuracy': acc}
    return metrics


def predict(state, batch, rngs):
    y_true = batch[1]
    y_pred = state.apply_fn({'params': state.params}, batch[0], rngs=rngs)
    loss = cross_entropy_loss(y_true=y_true, y_pred=y_pred)
    acc = accuracy(y_true=y_true, y_pred=y_pred)
    metrics = {'loss': loss, 'accuracy': acc}
    return metrics


def train(batch_size, max_seq_len, features=512, embed_size=256, lr=1e-3, epochs=10):
    batch_size = batch_size * jax.device_count()
    vocab_size = 256 + 1
    n_layer = 1
    output_size = 2
    dropout_rate = 0.1
    name = 'hrrformer'
```

```
    print('total batch size:', batch_size, 'max seq len:', max_seq_len)


    # load dataset
    train_loader, test_loader = load_dataset(batch_size=batch_size,
                                             max_seq_len=max_seq_len,
                                             shuffle=True,
                                             num_workers=0)

    # build and initialize network
    train_model = Network(features=features,
                          vocab_size=vocab_size,
                          embed_size=embed_size,
                          max_seq_len=max_seq_len,
                          nlayers=n_layer,
                          output_size=output_size,
                          dropout_rate=dropout_rate,
                          training=True)

    test_model = Network(features=features,
                         vocab_size=vocab_size,
                         embed_size=embed_size,
                         max_seq_len=max_seq_len,
                         nlayers=n_layer,
                         output_size=output_size,
                         dropout_rate=dropout_rate,
                         training=False)

    p_key_next, p_key = jax.random.split(jax.random.PRNGKey(0))
    d_key_next, d_key = jax.random.split(jax.random.PRNGKey(0))
    init_rngs = {'params': p_key, 'dropout': d_key}

    params = initialize_model(model=train_model, input_size=max_seq_len, init_rngs=init_rngs)


    # optimizer and scheduler
    steps = 800_000 // batch_size
    scheduler = optax.exponential_decay(init_value=lr,
                                        transition_steps=steps,
                                        decay_rate=.85,
                                        transition_begin=1,
                                        end_value=1e-5)

    tx = optax.adam(learning_rate=scheduler)
    state = train_state.TrainState.create(apply_fn=train_model.apply, params=params, tx=tx)
    # state = load_model(state, f'weights/{name}_multi_{n_layer}_{max_seq_len}.h5')
    state = jax_utils.replicate(state)

    # train

    history = []
    train_loss, train_acc = [], []
    test_loss, test_acc = [], []

    form = 'Epoch {0:>3d}/' + str(epochs) + ', train loss: {1:>8.6f}, train accuracy: {2:>5.2f}%, '
    form += 'test loss: {3:>8.6f}, test accuracy: {4:>5.2f}%, etc: {5:>.2f}s'

    for epoch in range(1, epochs + 1):
        train_loss_batch, train_acc_batch = [], []
```

```
    print()

    printf('Total batch size: ' + str(batch_size) + ', Max seq len: ' + str(max_seq_len))
    printf("load dataset")

    # load dataset
    train_loader, validation_loader, num_train_samples, num_validation_samples = load_dataset(batch_size=batch_size,
                                                                                              max_seq_len=max_seq_len,
                                                                                              shuffle=True,
                                                                                              num_workers=0)

    printf("build and initialize network")
    # build and initialize network
    train_model = Network(features=features,
                          vocab_size=vocab_size,
                          embed_size=embed_size,
                          max_seq_len=max_seq_len,
                          nlayers=n_layer,
                          output_size=output_size,
                          dropout_rate=dropout_rate,
                          training=True)

    validation_model = Network(features=features,
                               vocab_size=vocab_size,
                               embed_size=embed_size,
                               max_seq_len=max_seq_len,
                               nlayers=n_layer,
                               output_size=output_size,
                               dropout_rate=dropout_rate,
                               training=False)

    p_key_next, p_key = jax.random.split(jax.random.PRNGKey(0))
    d_key_next, d_key = jax.random.split(jax.random.PRNGKey(0))
    init_rngs = {'params': p_key, 'dropout': d_key}
    printf("initialize_model")
    params = initialize_model(model=train_model, input_size=max_seq_len, init_rngs=init_rngs)

    printf("optimizer and scheduler")

    # optimizer and scheduler
    steps = 800_000 // batch_size
    scheduler = optax.exponential_decay(init_value=lr,
                                        transition_steps=steps,
                                        decay_rate=.85,
                                        transition_begin=1,
                                        end_value=1e-5)

    tx = optax.adam(learning_rate=scheduler)
    state = train_state.TrainState.create(apply_fn=train_model.apply, params=params, tx=tx)
    # state = load_model(state, f'{WEIGHTS_BASE_PATH}{name}_multi_{n_layer}_{max_seq_len}.h5')
    state = jax_utils.replicate(state)

    printf("training the model...")
    # training
    history = []
    train_loss, train_acc = [], []
    validation_loss, validation_acc = [], []

    form = 'Epoch {0:>3d}/' + str(epochs) + ', train loss: {1:>8.6f}, train accuracy: {2:>5.2f}%, '
    form += 'validation loss: {3:>8.6f}, validation accuracy: {4:>5.2f}%, etc: {5:>.2f}s'

    for epoch in range(1, epochs + 1):
        train_loss_batch, train_acc_batch = [], []
```

```
        state = state.replace(apply_fn=train_model.apply)

        tic1 = time.time()
        for x_train, y_train in train_loader:
            p_key_next, p_key = jax.random.split(p_key_nex
t)
            d_key_next, d_key = jax.random.split(d_key_nex
t)
            rngs = {'params': shard_prng_key(p_key), 'dropo
ut': shard_prng_key(d_key)}

            batch = [x_train, y_train]
            batch = shard(batch)

            state, metrics = jax.pmap(train_step, axis_name
="batch", donate_argnums=(0,))(state, batch, rngs)

            train_loss_batch.append(metrics['loss'].mean())
            train_acc_batch.append(metrics['accuracy'].mean
())

        toc1 = time.time()
        train_loss.append(sum(train_loss_batch) / len(train
_loss_batch))
        train_acc.append(sum(train_acc_batch) / len(train_a
cc_batch))

        # test
        test_loss_batch, test_acc_batch = [], []

        state = state.replace(apply_fn=test_model.apply)


        tic2 = time.time()
        for x_test, y_test in test_loader:
            p_key_next, p_key = jax.random.split(p_key_nex
t)
            d_key_next, d_key = jax.random.split(d_key_nex
t)
            rngs = {'params': shard_prng_key(p_key), 'dropo
ut': shard_prng_key(d_key)}

            test_batch = [x_test, y_test]
            test_batch = shard(test_batch)

            metrics = jax.pmap(predict, axis_name="batch")
(state, test_batch, rngs)

            test_loss_batch.append(metrics['loss'].mean())

            test_acc_batch.append(metrics['accuracy'].mean
())

        toc2 = time.time()
        test_loss.append(sum(test_loss_batch) / len(test_lo
ss_batch))
        test_acc.append(sum(test_acc_batch) / len(test_acc_
batch))

        etc = (toc1 - tic1) + (toc2 - tic2)
        history.append(form.format(epoch, train_loss[-1], t
rain_acc[-1], test_loss[-1], test_acc[-1], etc))

        print(history[-1])

    state = jax_utils.unreplicate(state)
    save_model(state, f'weights/{name}_multi_{n_layer}_{max
_seq_len}.h5')
    save_history(f'weights/{name}_multi_{n_layer}_{max_seq_
len}.csv', history=history)
```

```
        state = state.replace(apply_fn=train_model.apply)

        tic1 = time.time()
        for x_train, y_train in train_loader:
            p_key_next, p_key = jax.random.split(p_key_nex
t)
            d_key_next, d_key = jax.random.split(d_key_nex
t)
            rngs = {'params': shard_prng_key(p_key), 'dropo
ut': shard_prng_key(d_key)}

            batch = [x_train, y_train]
            batch = shard(batch)

            state, metrics = jax.pmap(train_step, axis_name
="batch", donate_argnums=(0,))(state, batch, rngs)

            train_loss_batch.append(metrics['loss'].mean())
            train_acc_batch.append(metrics['accuracy'].mean
())

        toc1 = time.time()
        train_loss.append(sum(train_loss_batch) / len(train
_loss_batch))
        train_acc.append(sum(train_acc_batch) / len(train_a
cc_batch))

        # validating
        validation_loss_batch, validation_acc_batch = [],
 []
        state = state.replace(apply_fn=validation_model.app
ly)

        tic2 = time.time()
        for x_val, y_val in validation_loader:
            p_key_next, p_key = jax.random.split(p_key_nex
t)
            d_key_next, d_key = jax.random.split(d_key_nex
t)
            rngs = {'params': shard_prng_key(p_key), 'dropo
ut': shard_prng_key(d_key)}

            validation_batch = [x_val, y_val]
            validation_batch = shard(validation_batch)

            metrics = jax.pmap(predict, axis_name="batch")
(state, validation_batch, rngs)

            validation_loss_batch.append(metrics['loss'].me
an())
            validation_acc_batch.append(metrics['accurac
y'].mean())

        toc2 = time.time()
        validation_loss.append(sum(validation_loss_batch) /
len(validation_loss_batch))
        validation_acc.append(sum(validation_acc_batch) / l
en(validation_acc_batch))

        etc = (toc1 - tic1) + (toc2 - tic2)
        history.append(form.format(epoch, train_loss[-1], t
rain_acc[-1], validation_loss[-1], validation_acc[-1], et
c))
        printf(history[-1])

    state = jax_utils.unreplicate(state)



    # Save model weights for the dataset

    save_model(state, f'{WEIGHTS_BASE_PATH}{name}_multi_{n_
layer}_{max_seq_len}.h5')
    save_history(f'{WEIGHTS_BASE_PATH}{name}_multi_{n_laye
r}_{max_seq_len}.csv',
                 history=history)
```

```
def test(batch_size, max_seq_len, features=512, embed_size=256, lr=1e-3):
    batch_size = batch_size * jax.device_count()
    vocab_size = 256 + 1
    n_layer = 1
    output_size = 2
    dropout_rate = 0.1
    name = 'hrrformer'

    print()
    printf('Total batch size: ' + str(batch_size) + ', Max seq len: ' + str(max_seq_len))
    printf("loading dataset for testing...")

    # load dataset
    test_loader, num_test_samples = load_dataset(batch_size=32,
                                                 max_seq_len=max_seq_len,
                                                 shuffle=True,
                                                 num_workers=0)

    printf("build and initialize network")

    test_model = Network(features=features,
                         vocab_size=vocab_size,
                         embed_size=embed_size,
                         max_seq_len=max_seq_len,
                         nlayers=n_layer,
                         output_size=output_size,
                         dropout_rate=dropout_rate,
                         training=False)

    p_key_next, p_key = jax.random.split(jax.random.PRNGKey(0))
    d_key_next, d_key = jax.random.split(jax.random.PRNGKey(0))

    history = []
    test_loss, test_acc = [], []

    init_rngs = {'params': p_key, 'dropout': d_key}
    printf("initialize_model")
    params = initialize_model(model=test_model, input_size=max_seq_len, init_rngs=init_rngs)

    printf("optimizer and scheduler")

    # optimizer and scheduler
    steps = 800_000 // batch_size
    scheduler = optax.exponential_decay(init_value=lr,
                                        transition_steps=steps,
                                        decay_rate=.85,
                                        transition_begin=1,
                                        end_value=1e-5)

    tx = optax.adam(learning_rate=scheduler)

    state = train_state.TrainState.create(apply_fn=test_model.apply, params=params, tx=tx)
    state = load_model(state,
                       f'{WEIGHTS_BASE_PATH}{name}_multi_{n_layer}_{max_seq_len}.h5')
    state = jax_utils.replicate(state)

    form = 'test loss: {3:>8.6f}, test accuracy: {4:>5.2f}%, etc: {5:>.2f}s'

    # test
    test_loss_batch, test_acc_batch = [], []
    state = state.replace(apply_fn=test_model.apply)

    tic2 = time.time()
    for x_test, y_test in test_loader:
```

```
        p_key_next, p_key = jax.random.split(p_key_next)
        d_key_next, d_key = jax.random.split(d_key_next)
        rngs = {'params': shard_prng_key(p_key), 'dropout': shard_prng_key(d_key)}

        test_batch = [x_test, y_test]
        test_batch = shard(test_batch)

        metrics = jax.pmap(predict, axis_name="batch")(state, test_batch, rngs)

        test_loss_batch.append(metrics['loss'].mean())
        test_acc_batch.append(metrics['accuracy'].mean())

    toc2 = time.time()
    test_loss.append(sum(test_loss_batch) / len(test_loss_batch))
    test_acc.append(sum(test_acc_batch) / len(test_acc_batch))

    etc = (toc2 - tic2)
    history.append(form.format(0, test_loss[-1], test_acc[-1], test_loss[-1], test_acc[-1], etc))
    printf(history[-1])


def printf(message):
    timestamp = datetime.datetime.now()
    timestamp_str = timestamp.strftime("%Y-%m-%d %H:%M:%S")
    print(f"[{timestamp_str}] {message}")


def individual_predict(batch_size, max_seq_len, features=512, embed_size=256, lr=1e-3):
    global prediction_result
    prediction_result = "None"
    vocab_size = 256 + 1
    n_layer = 1
    output_size = 2
    dropout_rate = 0.1
    name = 'hrrformer'

    print()
    printf('Total batch size: ' + str(batch_size) + ', Max seq len: ' + str(max_seq_len))
    printf("loading dataset for individual prediction analysis...")

    # load dataset
    test_loader, num_test_samples = load_dataset(batch_size=1,
                                                 max_seq_len=max_seq_len,
                                                 shuffle=True,
                                                 num_workers=0)
    printf(f"Number of samples to predict: {num_test_samples}")

    printf("build and initialize network")

    test_model = Network(features=features,
                         vocab_size=vocab_size,
                         embed_size=embed_size,
                         max_seq_len=max_seq_len,
                         nlayers=n_layer,
                         output_size=output_size,
                         dropout_rate=dropout_rate,
                         training=False)

    p_key_next, p_key = jax.random.split(jax.random.PRNGKey(0))
    d_key_next, d_key = jax.random.split(jax.random.PRNGKey(0))

    history = []
```

```
    test_loss, test_acc = [], []

    init_rngs = {'params': p_key, 'dropout': d_key}
    printf("initialize_model")
    params = initialize_model(model=test_model, input_size=max_seq_len, init_rngs=init_rngs)

    printf("optimizer and scheduler")

    # optimizer and scheduler
    steps = 800_000 // batch_size
    scheduler = optax.exponential_decay(init_value=lr,
                                        transition_steps=steps,
                                        decay_rate=.85,
                                        transition_begin=1,
                                        end_value=1e-5)

    tx = optax.adam(learning_rate=scheduler)

    state = train_state.TrainState.create(apply_fn=test_model.apply, params=params, tx=tx)
    state = load_model(state,
                       f'{WEIGHTS_BASE_PATH}{name}_multi_{n_layer}_{max_seq_len}.h5')
    state = jax_utils.replicate(state)

    state = state.replace(apply_fn=test_model.apply)
    test_loss_batch, test_acc_batch = [], []

    for x_test, y_test in test_loader:
        p_key_next, p_key = jax.random.split(p_key_next)
        d_key_next, d_key = jax.random.split(d_key_next)
        rngs = {'params': shard_prng_key(p_key), 'dropout': shard_prng_key(d_key)}

        test_batch = [x_test, y_test]
        test_batch = shard(test_batch)
        metrics = jax.pmap(predict_ind, axis_name="batch")(state, test_batch, rngs)
        test_acc_batch.append(metrics['accuracy'].mean())

    test_acc.append(sum(test_acc_batch) / len(test_acc_batch))

    printf(test_acc[0])
    if str(test_acc[0]) == "100.0":
        return "Correct"
    return "Wrong"


def print_pred(prediction_result: str, bool_good: bool, filename: str):
    file_type = ""

    if MALWARE:
        file_type = "Malware"
    elif VULNERABILITY:
        file_type = "Vulnerable"

    if bool_good:
        if prediction_result == "Wrong":
            printf("File '" + filename + "' is incorrectly predicted as " + file_type + " file")
        else:
            printf("File '" + filename + "' is correctly predicted as Benign file")
    else:
        if prediction_result == "Wrong":
            printf("File '" + filename + "' is incorrectly predicted as Benign file")
        else:
            printf("File '" + filename + "' is correctly predicted as " + file_type + " file")


def inidividual_predict_helper():
```

```
                                                    # Algorithm for predict function
                                                    # 1. load the test data...
                                                    # 2. for single test file (dict{name->good/bad}):
                                                    #         predict if it is benign or not (we already know the ans)
                                                    #         if prediction is failing:
                                                    #                 note down this file details..

                                                    # Batch size and seq_len should be of the best trained model
                                                    import os
                                                    import shutil

                                                    # Source directory containing the files you want to process
                                                    source_directories = [TEST_DATA_PATH_GOOD, TEST_DATA_PATH_BAD]

                                                    if not os.path.exists(TEST_DATA_PATH_GOOD + "temp"):
                                                        os.mkdir(TEST_DATA_PATH_GOOD + "temp")

                                                    if not os.path.exists(TEST_DATA_PATH_BAD + "temp"):
                                                        os.mkdir(TEST_DATA_PATH_BAD + "temp")

                                                    # Iterate through files in the source directory
                                                    for source_directory in source_directories:

                                                        bool_good = True
                                                        if source_directory.endswith("/good/"):
                                                            temp_directory = TEST_DATA_PATH_GOOD + "temp"
                                                        else:
                                                            bool_good = False
                                                            temp_directory = TEST_DATA_PATH_BAD + "temp"
                                                        for filename in os.listdir(source_directory):
                                                            if not filename == "temp":
                                                                source_file = os.path.join(source_directory, filename)
                                                                temp_file = os.path.join(temp_directory, filename)
                                                                shutil.copy2(source_file, temp_file)
                                                                # Perform processing on the file in the temporary directory
                                                                prediction_result = individual_predict(batch_size=64, max_seq_len=1024)
                                                                print_pred(prediction_result, bool_good, filename)
                                                                # After processing, delete the file from the temporary directory
                                                                os.remove(temp_file)
                                                        os.removedirs(temp_directory)
```

```
if __name__ == '__main__':
```

```
    from math import log2

    seq = [2 ** i for i in range(8, 20)]
    for seq_len in seq:
        batch = max(2 ** int(16 - log2(seq_len)), 1)
        train(batch_size=batch, max_seq_len=seq_len, epochs=10)
```

```
if __name__ == '__main__':

    from math import log2

    if TRAIN:
        # seq = [2 ** i for i in range(8, 20)]
        # for seq_len in seq:
        #     batch = max(2 ** int(16 - log2(seq_len)), 1)

        #     if batch > 2:
        train(batch_size=128, max_seq_len=512, epochs=10)
        printf("Training completed")

    elif TEST:
        seq = [2 ** i for i in range(8, 20)]
        for seq_len in seq:
            batch = max(2 ** int(16 - log2(seq_len)), 1)
            if batch > 2 and seq_len <= 1024:
                test(batch_size=batch, max_seq_len=seq_len)
        printf("Testing completed")

    elif PREDICT:
        inidividual_predict_helper()
```