Demonstration report

Data generator

The first step is loading the data. The dataset is split into 6 batches that are loaded using the following function:

```python
def _load_batch(path):
    with open(path, 'rb') as f:
        data = pickle.load(f, encoding='latin1')
    X = data['data']
    y = data['labels']
    X = X.reshape(X.shape[0], 3, 32, 32)
    X = np.transpose(X, axes=[0, 2, 3, 1])
    return X, np.asarray(y)
```

The images and labels are stored in a pickle file, and the images in the original format have the channels on the first dimension, and we convert them to images with channels on the last dimension.

In order to integrate the data generator with the Keras functionalities, we define the following class:

```python
class Cifar10DataGenerator(tf.keras.utils.Sequence):
    CLASSES = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

And in the constructor, we read the batches and normalize them in the 0-1 range by dividing with 255:

```python
def __init__(self, data_root, batches_to_load, batch_size, flatten, shuffle=False, limit_batches=None):
    self.X = []
    self.y = []
    for batch in batches_to_load:
        X_batch, y_batch = _load_batch(f'{data_root}/data_batch_{batch}')
        self.X.extend(X_batch)
        self.y.extend(y_batch)

    self.X = np.asarray(self.X) / 255.0
    self.y = np.asarray(self.y)

    if flatten:
        self.X = np.reshape(self.X, (len(self.X), -1))

    self.batch_size = batch_size
    self.shuffle = shuffle
    self.flatten = flatten

    if limit_batches:
        self.X = self.X[:limit_batches * self.batch_size]
        self.y = self.y[:limit_batches * self.batch_size]

    self.indices = np.arange(len(self.X))
    self.on_epoch_end()
```

Also, we have the option to shuffle the dataset:

```
def on_epoch_end(self):
    """
    Called at the end of each epoch
    """
    # if required, shuffle your data after each epoch
    self.indices = np.arange(len(self.X))
    if self.shuffle:
        np.random.shuffle(self.indices)
```

This will be called automatically by Keras after each epoch.

The data is read in batches:

```
def __getitem__(self, index):
    batch_indices = self.indices[index * self.batch_size: (index + 1) * self.batch_size]
    if self.flatten:
        batch_x = self.X[batch_indices, :]
    else:
        batch_x = self.X[batch_indices, :, :, :]
    batch_y = self.y[batch_indices]
    return batch_x, batch_y
```

And the length of the dataset is given by the number of batches:

```
def __len__(self):
    """
    Returns the number of batches per epoch: the total size of the dataset divided by the batch size
    """
    return int(np.ceil(len(self.X) / self.batch_size))
```

Model and training

We define the following softmax function that can handle both simple samples and batches:

```
def softmax(x):
    axis = 1 if len(x.shape) == 2 else None
    x_stabilized = x - np.max(x, axis=axis, keepdims=True)
    return np.exp(x_stabilized) / np.sum(np.exp(x_stabilized), axis=axis, keepdims=True)
```

We define the main logic in the following class:

```python
class SoftmaxClassifier:
    def __init__(self, input_shape, num_classes):
        self.input_shape = input_shape
        self.num_classes = num_classes
        self.W = None
        self.initialize()
```

We initialize the weights, having in mind the bias term by appending ones to the weights:

```python
def initialize(self):
    self.W = np.random.randn(self.input_shape - 1, self.num_classes) * 0.001
    self.W = np.concatenate([self.W, np.ones((1, self.num_classes))], axis=0)  # bias trick
```

The following are used to predict labels and probabilities:

```python
def predict_probability(self, X):
    return softmax(X.dot(self.W))

def predict_label(self, X):
    return np.argmax(X.dot(self.W), axis=1)
```

The following handles the logic for an epoch. The loss is computed for each batch and if the learning rate is not None i.e. we are during training, also the weights update:

```python
def run_epoch(self, data_generator, reg_strength, lr):
    losses = []
    pred = []
    gt = []
    for i in range(len(data_generator)):
        X_batch, y_batch = data_generator[i]
        current_bs = len(X_batch)

        output = X_batch.dot(self.W)

        pred += np.argmax(output, axis=1).tolist()
        gt += y_batch.tolist()

        stabilized_output = output - np.max(output, axis=1, keepdims=True)
        loss = -stabilized_output[range(current_bs), y_batch].reshape(current_bs, 1) \
               + np.log(np.sum(np.exp(stabilized_output), axis=1, keepdims=True))

        loss = np.mean(loss) + reg_strength * np.sum(np.power(self.W, 2))

        if lr is not None:
            CT = softmax(output)
            CT[range(current_bs), y_batch] -= 1
            dW = X_batch.T.dot(CT) + reg_strength * self.W
            self.W -= lr * dW

        losses.append(loss)
    return np.mean(losses), compute_accuracy(compute_confusion_matrix(gt, pred))
```

In the fit method we actually call the run_epoch method and record the losses and training times:

```python
def fit(self, train_generator, val_generator, epochs=100, reg_strength=1e3, lr=1e-3, save_to=None):
    history = {
        'lr': [],
        'train_loss': [],
        'val_loss': [],
        'train_acc': [],
        'val_acc': [],
    }
    no_improvement = 0
    best = -1
    prev_path = None
    for epoch in range(epochs):
        start = perf_counter()
        train_loss, train_acc = self.run_epoch(train_generator, reg_strength, lr)
        val_loss, val_acc = self.run_epoch(val_generator, reg_strength, None)
        time = perf_counter()-start
        print(f'Epoch {epoch+1}/{epochs} '
              f'Time {time} '
              f'Train loss {train_loss} '
              f'Val loss {val_loss} '
              f'Train acc {train_acc} '
              f'Val acc {val_acc} '
              f'Learning rate {lr}'
              )
```

```python
              )
        history['lr'].append(lr)
        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['train_acc'].append(train_acc)
        history['val_acc'].append(val_acc)
        if len(history['val_loss']) > 1:
            if history['val_loss'][-2] - history['val_loss'][-1] < 1e-2:
                no_improvement += 1
                if no_improvement >= 3:
                    lr /= 2
                    no_improvement = 0
            else:
                no_improvement = 0
        if save_to and val_acc > best:
            best = val_acc
            new_path = f'{save_to}_best_epoch_{epoch+1}_acc_{val_acc}.npy'
            self.save(new_path)
            if prev_path:
                os.remove(prev_path)
            prev_path = new_path

    return history
```

We can also load and save the model by:

```python
def load(self, path):
    self.W = np.load(path)

def save(self, path):
    np.save(path, self.W)
```

A training is started as follows:

```python
def train(model, train_generator, val_generator, epochs, reg_strength, lr, save_to):
    history = model.fit(train_generator, val_generator, epochs, reg_strength=reg_strength, lr=lr, save_to=save_to)
    plot_history(history)
    to_json(history, f'{save_to}.json')


if __name__ == '__main__':
    test_batch = 1
    epochs = 20
    lr = 1e-4
    bs = 256
    reg_strength = 1e-3
    experiment = f'test_batch_{test_batch}_bs_{bs}_lr_{lr}_reg_strength_{reg_strength}_epochs_{epochs}'

    train_batches = list(range(1, 7))
    train_batches.remove(test_batch)

    train_generator = Cifar10DataGenerator('cifar-10-batches-py', train_batches, batch_size=bs, flatten=True, shuffle=True, normalize=True, bias_trick=True)
    val_generator = Cifar10DataGenerator('cifar-10-batches-py', [test_batch], batch_size=bs, flatten=True, normalize=True, bias_trick=True)

    model = SoftmaxClassifier(32*32*3+1, 10)

    train(model, train_generator, val_generator, epochs, reg_strength, lr, save_to=f'weights/{experiment}')
```

Evaluation

In order to perform the evaluation, we first generate a json file containing the ground truth and predicted labels, together with the probabilities for all classes.

This is done on each batch, by loading the corresponding data generator and model as follows:

```python
for test_batch in range(1, 7):
    generator = Cifar10DataGenerator(
        "cifar-10-batches-py",
        [test_batch],
        batch_size=32, flatten=architecture == 'ann'
    )

    model = K.models.load_model(f'weights/{architecture}/{experiment}/test_batch_{test_batch}')
    model.summary()
```

After this we compute the results for each image:

```python
pred_all = []
gt_all = []
pred_prob_all = []
for i in range(len(generator)):
    print(f'{i+1}/{len(generator)} test_batch {test_batch}')
    X, y = generator[i]
    results = model(X)
    pred_classes = tf.argmax(results, axis=-1).numpy()
    a = softmax(results.numpy())
    pred_prob = softmax(results.numpy())  # [(list(range(len(results))), pred_classes)]

    pred_all += pred_classes.tolist()
    pred_prob_all += pred_prob.tolist()
    gt_all += y.tolist()
print(f"no preds {len(pred_all)}  no gt {len(gt_all)}")
```

And save them as follows:

```python
path = f'results/{architecture}/{experiment}'
if os.path.isdir(path):
    print(f"directory {path} exists")
else:
    print(f"creating {path}")
    os.mkdir(path)
to_json({
    'pred': pred_all,
    'gt': gt_all,
    'prob': pred_prob_all
}, f'{path}/test_batch_{test_batch}.json')
```

We compute the following metrics: accuracy, precision, recall, f-score and AUC.

In order to compute them, first the confusion matrix must be computed:

```python
def compute_confusion_matrix(gt, pred, num_classes=10):
    conf_mat = np.zeros((num_classes, num_classes))
    np.add.at(conf_mat, (pred, gt), 1)
    return conf_mat
```

Both pred and gt are lists containing the labels, but the labels can also be seen as indices in the matrix, therefore at each position described by the pair of indexes (pred, gt) we add 1.

Accuracy:

```
def compute_accuracy(conf_mat):
    return np.sum(np.diagonal(conf_mat)) / np.sum(conf_mat)
```

Precision:

```
def compute_precision(conf_mat):
    denom = np.sum(conf_mat, axis=1)
    denom[denom == 0] = 1
    return np.diagonal(conf_mat) / denom
```

Recall:

```
def compute_recall(conf_mat):
    denom = np.sum(conf_mat, axis=0)
    denom[denom == 0] = 1
    return np.diagonal(conf_mat) / denom
```

F-score:

```
def compute_fscore(conf_mat):
    prec = compute_precision(conf_mat)
    rec = compute_precision(conf_mat)
    denom = prec + rec
    denom[denom == 0] = 1
    return 2 * prec * rec / denom
```

AUC is a binary metric, therefore in order to compute we implement a one vs all approach by taking each class and considering it positive, and the rest negative. First, we implement binary AUC:

```python
def compute_AUC_binary(gt, prob, steps):
    true_positive_rates = [0]
    false_positive_rates = [0]
    for threshold in np.linspace(0, 1, steps):
        pred = np.asarray([1 if p >= threshold else 0 for p in prob])
        conf_mat = compute_confusion_matrix(gt, pred, 2)
        recall = compute_recall(conf_mat)
        sens, spec = recall[0], recall[1]
        true_positive_rate = sens
        false_positive_rate = 1 - spec
        true_positive_rates.append(true_positive_rate)
        false_positive_rates.append(false_positive_rate)
    return true_positive_rates, false_positive_rates
```

Then for all classes:

```python
def compute_AUC(gt, prob, plot=False, steps=11, classes=Cifar10DataGenerator.CLASSES):
    if plot:
        plt.subplots(5, 2, figsize=(5, 10))
    class_AUC = []
    for c in range(len(classes)):
        binary_prob = prob[:, c]
        binary_gt = np.asarray([1 if y == c else 0 for y in gt])
        true_positive_rates, false_positive_rates = compute_AUC_binary(binary_gt, binary_prob, steps)
        class_AUC.append(repeated_trapezium(false_positive_rates, true_positive_rates))
        if plot:
            ax = plt.subplot(5, 2, c + 1)
            ax.set_title(classes[c])
            plot_roc_curve(true_positive_rates, false_positive_rates, ax=ax)
            ax.set_title(classes[c])
    if plot:
        plt.tight_layout()
        plt.show()
    return class_AUC
```

The actual AUC is an integral and we use the repeated trapezium formula to compute it:

```python
def repeated_trapezium(x, y):
    n = len(x) - 1
    return (x[-1] - x[0]) / (2 * n) * (y[0] + y[-1] + 2 * np.sum(y[:-1]))
```

In the end, for all metrics, except for accuracy, we record the values for each class and the mean in a json file.

This is done in a cross-validation manner, by computing them on each batch. Finally, we compute the means and confidence intervals batch wise:

```python
metrics_path = 'results/ann/batch_size_32_epochs_20_lr_0.0001/metrics.json'
metrics = open_json(metrics_path)

analysis = {}

for k, v in metrics.items():
    if type(v) is dict:
        if k not in analysis:
            analysis[k] = {}
        for sub_k, sub_v in v.items():
            analysis[k][sub_k] = {
                'mean': np.mean(sub_v),
                'std': np.std(sub_v),
                'alpha': 1.96*np.std(sub_v)/np.sqrt(6)
            }
    else:
        analysis[k] = {
            'mean': np.mean(v),
            'std': np.std(v),
            'alpha': 1.96*np.std(v)/np.sqrt(6)
        }
to_json(analysis, f'{metrics_path[:-5]}_analysis.json')
```

Utils

Useful functions:

```python
def to_json(obj, file):
    json_obj = json.dumps(obj, indent=4)
    with open(file, "w") as f:
        f.write(json_obj)


def open_json(file):
    with open(file) as f:
        return json.load(f)


def to_pickle(obj, file):
    with open(file, 'wb') as f:
        pickle.dump(obj, f)


def open_pickle(file):
    with open(file, 'rb') as f:
        return pickle.load(f)


def softmax(x, t=1):
    axis = 1 if len(x.shape) == 2 else None
    x_stabilized = x - np.max(x, axis=axis, keepdims=True)
    return np.exp(x_stabilized / t) / np.sum(np.exp(x_stabilized / t), axis=axis, keepdims=True)
```