Modify the data loader to return extended binary labels

```python
###################
# Dataset
###################


class AFADDatasetAge(Dataset):
    """Custom Dataset for loading AFAD face images"""

    def __init__(self, csv_path, img_dir, transform=None):

        df = pd.read_csv(csv_path, index_col=0)
        self.img_dir = img_dir
        self.csv_path = csv_path
        self.img_paths = df['path']
        self.y = df['age'].values
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir,
                                      self.img_paths[index]))

        if self.transform is not None:
            img = self.transform(img)

        label = self.y[index]

```

```python
###################
# Dataset
###################

class AFADDatasetAge(Dataset):
    """Custom Dataset for loading AFAD face images"""

    def __init__(self, csv_path, img_dir, transform=None):

        df = pd.read_csv(csv_path, index_col=0)
        self.img_dir = img_dir
        self.csv_path = csv_path
        self.img_paths = df['path']
        self.y = df['age'].values
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir,
                                      self.img_paths[index]))

        if self.transform is not None:
            img = self.transform(img)

        label = self.y[index]
        levels = [1]*label + [0]*(NUM_CLASSES - 1 - label)
        levels = torch.tensor(levels, dtype=torch.float32)

```

Add the CORAL layer:

```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes, grayscale):
        self.inplanes = 64
        if grayscale:
            in_dim = 1
        else:
            in_dim = 3
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(4)
        self.fc = nn.Linear(512, self.num_classes)
```

```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes, grayscale):
        self.num_classes = num_classes
        self.inplanes = 64
        if grayscale:
            in_dim = 1
        else:
            in_dim = 3
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(4)
        self.fc = nn.Linear(512, 1, bias=False)
        self.linear_1_bias = nn.Parameter(torch.zeros(
            self.num_classes-1).float())
```

Modify the forward method accordingly:

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)

        x = x.view(x.size(0), -1)
        logits = self.fc(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        logits = self.fc(x)
        logits = logits + self.linear_1_bias
        probas = torch.sigmoid(logits)
        return logits, probas
```

Add the CORAL loss:

```python
##########################################
# Initialize Cost, Model, and Optimizer
##########################################

```

```python
##########################################
# Initialize Cost, Model, and Optimizer
##########################################

def cost_fn(logits, levels, imp):
    val = (-torch.sum((F.logsigmoid(logits)*levels
                      + (F.logsigmoid(logits) - logits)*(1-levels))*imp,
           dim=1))
    return torch.mean(val)


```

Modify the training loop accordingly:

```python
best_mae, best_rmse, best_epoch = 999, 999, -1
for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.to(DEVICE)
        targets = targets.to(DEVICE)

        # FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()

        # UPDATE MODEL PARAMETERS
        optimizer.step()

        # LOGGING
        if not batch_idx % 50:
            s = ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
                 % (epoch+1, num_epochs, batch_idx,
                    len(train_dataset)//BATCH_SIZE, cost))
            print(s)
            with open(LOGFILE, 'a') as f:
                f.write('%s\n' % s)

    model.eval()
    with torch.set_grad_enabled(False):
        test_mae, test_mse = compute_mae_and_mse(model, test_loader,
                                                 device=DEVICE)
```

```python
best_mae, best_rmse, best_epoch = 999, 999, -1
for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, targets, levels) in enumerate(train_loader):

        features = features.to(DEVICE)
        targets = targets
        targets = targets.to(DEVICE)
        levels = levels.to(DEVICE)

        # FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = cost_fn(logits, levels, imp)
        optimizer.zero_grad()

        cost.backward()

        # UPDATE MODEL PARAMETERS
        optimizer.step()

        # LOGGING
        if not batch_idx % 50:
            s = ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
                 % (epoch+1, num_epochs, batch_idx,
                    len(train_dataset)//BATCH_SIZE, cost))
            print(s)
            with open(LOGFILE, 'a') as f:
                f.write('%s\n' % s)

    model.eval()
    with torch.set_grad_enabled(False):
        valid_mae, valid_mse = compute_mae_and_mse(model, valid_loader,
                                                   device=DEVICE)
```