# Class note

## 1.ARCface

### 1. ArcFace Loss Function

```
import tensorflow as tf

def arcface_loss(y_true, y_pred, margin=0.5, scale=64):
    y_pred = tf.clip_by_value(y_pred, -1.0 + 1e-7, 1.0 - 1e-7)
    theta = tf.acos(y_pred)
    target_logits = scale * tf.cos(theta + margin)
    other_logits = scale * tf.cos(theta)
    return tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        labels=y_true,
        logits=tf.where(tf.cast(y_true, dtype=tf.bool), target_logits,
other_logits)
    ))
```

- **ArcFace Loss**: This is a loss function designed specifically for face recognition tasks. It enhances inter-class separability by adding an angular margin between classes.
    - `y_true`: The ground truth labels, one-hot encoded.
    - `y_pred`: The cosine similarities (model predictions) between the normalized face embeddings.
    - **Angular Margin** (`margin=0.5`): Enforces a margin between the learned features of different classes by adding an offset to the angle `theta`.
    - **Scaling** (`scale=64`): Used to scale the logits, which makes the optimization easier and helps prevent numerical issues.
    - **Process**:
        - The cosine similarity output (`y_pred`) is clipped to ensure numerical stability (between -1 and 1).
        - The angle `theta` between the predicted embeddings is computed using `tf.acos`.
        - **Target logits**: For true class predictions, the angle is modified by adding the margin.
        - **Other logits**: For other class predictions, no margin is added.
        - **Loss Computation**: The `softmax_cross_entropy_with_logits` function calculates the cross-entropy between the target logits and the actual labels.

---

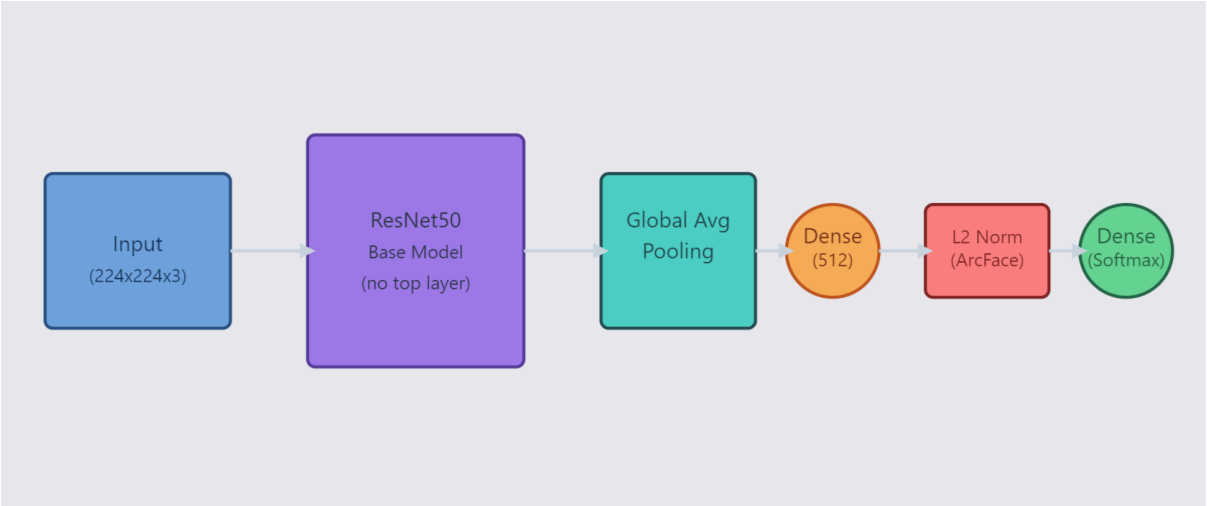
### 2. Data Generator

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

IMG_HEIGHT, IMG_WIDTH = 224, 224
BATCH_SIZE = 32
DATA_PATH = '/content/drive/MyDrive/test_vggface'

def create_data_generator(data_path, img_height, img_width, batch_size):
    data_generator =
ImageDataGenerator(preprocessing_function=tf.keras.applications.resnet50.preprocess
_input)
    train_gen = data_generator.flow_from_directory(
        data_path,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical',
        shuffle=True
    )
    return train_gen
train_gen = create_data_generator(DATA_PATH, IMG_HEIGHT, IMG_WIDTH, BATCH_SIZE)
```

- **Data Generator**: Here, the `ImageDataGenerator` class is used to preprocess and feed the image data into the model during training.
    - `ImageDataGenerator(preprocessing_function=...)`: Preprocessing is done using the ResNet50 preprocessing function, which scales the pixel values in the range expected by the model.
    - `flow_from_directory()`: This function loads images from a directory where each subdirectory represents a different class.
        - `target_size=(224, 224)`: Resizes all images to 224x224 pixels (the input size for ResNet50).
        - `batch_size=32`: Feeds images in batches of 32.
        - `class_mode='categorical'`: The labels are expected to be one-hot encoded because this is a multi-class classification task.
        - `shuffle=True`: The images are shuffled before being fed into the model to improve training generalization.

---

## 3. Model Architecture with ResNet50 Backbone




```
from tensorflow.keras import layers, models

def build_model(input_shape, num_classes):
    base_model = tf.keras.applications.ResNet50(weights='imagenet',
include_top=False, input_shape=input_shape)
    x = layers.GlobalAveragePooling2D()(base_model.output)
    x = layers.Dense(512)(x)
    x = layers.Lambda(lambda t: tf.nn.l2_normalize(t, axis=1))(x)  #
ArcFace-specific layer

    output = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(inputs=base_model.input, outputs=output)

    return model

input_shape = (IMG_HEIGHT, IMG_WIDTH, 3)
num_classes = 10  # Change based on your dataset
model = build_model(input_shape, num_classes)
```

## 1. Input Layer

- Dimensions: (224, 224, 3)
- Accepts RGB images
- Standardized input size required for ResNet50
- 3 channels representing RGB color space

## 2. Base Model: ResNet50

- Pre-trained on ImageNet dataset
- Architecture characteristics:
    - 50 layers deep
    - Uses residual connections (skip connections)
    - Helps solve vanishing gradient problem

    - Trained on millions of images
- Configuration:
    - `weights='imagenet'`: Uses pre-trained weights
    - `include_top=False`: Removes classification layers
    - Preserves learned feature extraction capabilities

### 3. Global Average Pooling 2D

- Purpose:
    - Reduces spatial dimensions
    - Minimizes parameters
    - Provides translation invariance
- Benefits:
    - Prevents overfitting
    - Reduces computational complexity
    - Creates a fixed-size output regardless of input dimensions

### 4. Dense Embedding Layer (512 units)

- Characteristics:
    - Fully connected layer
    - 512-dimensional feature vector output
    - Creates compact representation
- Purpose:
    - Dimensionality reduction
    - Learning task-specific features
    - Creating embeddings suitable for face recognition

### 5. L2 Normalization (ArcFace Layer)

- Mathematical operation:
  python
  Copy
  ```
  x = layers.Lambda(lambda t: tf.nn.l2_normalize(t, axis=1))(x)
  ```

- Purpose:
    - Normalizes feature vectors to unit length
    - Essential for ArcFace loss function
    - Improves face recognition accuracy
- Benefits:
    - Ensures consistent scale for features
    - Helps in computing angular distances
    - Stabilizes training

### 6. Output Layer

- Configuration:

        - Dense layer with softmax activation
        - Number of units = number of classes
- Purpose:
    - Final classification
    - Probability distribution over classes
-

---

## 4. Training Process

```
def train_model(model, train_gen, epochs, arcface_loss_func):
    model.compile(optimizer='adam', loss=arcface_loss_func, metrics=['accuracy'])

    model.fit(
        train_gen,
        steps_per_epoch=train_gen.samples // train_gen.batch_size,
        epochs=epochs
    )

train_model(model, train_gen, epochs=10, arcface_loss_func=arcface_loss)
```

- **Model Training**: The training process uses the custom ArcFace loss function, and the model is trained on the face dataset.
    - **Compilation**:
        - `optimizer='adam'`: The Adam optimizer is used for adaptive learning during training.
        - `loss=arcface_loss_func`: The loss function is the custom ArcFace loss function defined earlier.
        - `metrics=['accuracy']`: Accuracy is tracked during training.
    - **Fitting the Model**:
        - The model is trained using the data generator `train_gen`, which feeds batches of images to the model.
        - `steps_per_epoch=train_gen.samples // train_gen.batch_size`: Defines the number of batches to process per epoch.
        - `epochs=10`: The training runs for 10 epochs.

---

## 5. Model Evaluation and Saving

```
loss, accuracy = model.evaluate(train_gen)
print('Accuracy:', accuracy)

model.save('/content/drive/MyDrive/models_face_recognition/arcface_2.keras')
```

- **Evaluation**: The model is evaluated on the training dataset to compute the accuracy.
    - `model.evaluate(train_gen)`: Evaluates the model on the training data and outputs the loss and accuracy.
- **Saving the Model**: The trained model is saved to disk using the `.keras` format.

---

## 6. Prediction and Visualization of Random Images

```
import matplotlib.pyplot as plt
import numpy as np
import random

def predict_random_images(generator, model, num_images=9):
    class_indices = generator.class_indices
    class_names = list(class_indices.keys())
    total_batches = len(generator)
    num_batches = (num_images + generator.batch_size - 1) // generator.batch_size

    random_batches = random.sample(range(total_batches), num_batches)
    images, actual_labels, predicted_classes, confidences = [], [], [], []

    for batch_idx in random_batches:
        batch_images, batch_labels = generator[batch_idx]
        predictions = model.predict(batch_images)

        for i in range(len(batch_images)):
            if len(images) >= num_images:
                break
            images.append(batch_images[i])
            actual_labels.append(batch_labels[i])
            predicted_class = np.argmax(predictions[i])
            predicted_classes.append(predicted_class)
            confidences.append(np.max(predictions[i]) * 100)

    plt.figure(figsize=(15, 15))
    for i in range(num_images):
        ax = plt.subplot(3, 3, i + 1)
        img_display = images[i]
        img_display = (img_display - np.min(img_display)) / (np.max(img_display) -
np.min(img_display))
        img_display = np.clip(img_display, 0, 1)

        plt.imshow(img_display)
        plt.title(f"Actual: {class_names[np.argmax(actual_labels[i])]} \n"
                  f"Predicted: {class_names[predicted_classes[i]]}\n"
                  f"Confidence: {confidences[i]:.2f}%")
        plt.axis("off")
```

```
    plt.tight_layout()
    plt.show()

predict_random_images(train_gen, model, num_images=9)
```

- **Objective**: This code randomly selects a few images from the dataset, makes predictions, and visualizes the results.
    - **Selecting Random Images**:
        - `random.sample` is used to pick random batches from the dataset.
        - Images and corresponding true labels are extracted from the batch and stored in `images` and `actual_labels`.
    - **Making Predictions**:
        - The model predicts the probabilities for each class using `model.predict`.
        - `np.argmax(predictions[i])` is used to determine the predicted class for each image.
        - The confidence of the prediction (maximum probability) is stored in `confidences`.
    - **Plotting the Results**:
        - The images are displayed in a 3x3 grid, with the actual class, predicted class, and prediction confidence shown as titles above each image.

---

### Summary:

- **ArcFace Loss**: A specialized loss function for face recognition, which enforces angular margins to improve class separation.
- **ResNet50 Backbone**: Used as a feature extractor for the face images.
- **Data Pipeline**: Images are loaded using `ImageDataGenerator`, preprocessed, and fed into the model in batches.
- **Training**: The model is trained using the ArcFace loss, with accuracy tracked during training.
- **Visualization**: Random images are displayed with their true and predicted labels to inspect the model's performance.

This approach effectively builds a robust face recognition system with high separability between the embeddings of different faces.

# ArcFace Implementation Analysis

# 1. Core Components of ArcFace Implementation

## A. Feature Normalization Layer

python
Copy

```
x = layers.Lambda(lambda t: tf.nn.l2_normalize(t, axis=1))(x)
```

- **Purpose**: Normalizes feature vectors to unit length
- **Operation**:
  - Takes the 512-dimensional embeddings
  - Applies L2 normalization: $x' = x / ||x||_2$
  - Ensures all feature vectors lie on a hypersphere
- **Importance**: Essential for angular-based computations

## B. ArcFace Loss Function

```
def arcface_loss(y_true, y_pred, margin=0.5, scale=64):
    y_pred = tf.clip_by_value(y_pred, -1.0 + 1e-7, 1.0 - 1e-7)
    theta = tf.acos(y_pred)
    target_logits = scale * tf.cos(theta + margin)
    other_logits = scale * tf.cos(theta)
    return tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        labels=y_true,
        logits=tf.where(tf.cast(y_true, dtype=tf.bool),
target_logits, other_logits)
    ))
```

# Key Components:

1. **Clipping**: `y_pred = tf.clip_by_value(y_pred, -1.0 + 1e-7, 1.0 - 1e-7)`
   - Prevents numerical instability
   - Ensures values stay within acos domain
2. **Angular Conversion**: `theta = tf.acos(y_pred)`
   - Converts cosine similarities to angles
   - Maps predictions to angular space
3. **Margin Addition**: `target_logits = scale * tf.cos(theta + margin)`
   - Adds angular margin (0.5 radians) to positive class
   - Increases intra-class compactness
4. **Scaling**: `scale=64`
   - Amplifies gradients
   - Helps with convergence

# 2. Working Mechanism

## Phase 1: Feature Extraction and Normalization

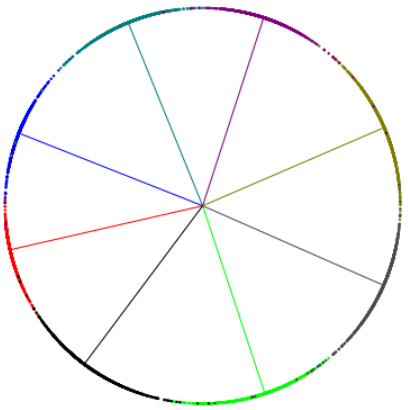
```
graph LR
    A[Input Image] --> B[ResNet50]
    B --> C[Global Pooling]
    C --> D[Dense 512]
    D --> E[L2 Normalization]
    E --> F[Feature Vectors]
```

1. **Feature Extraction**:
    - ResNet50 processes input images
    - GlobalAveragePooling reduces spatial dimensions
    - Dense layer creates 512-D embeddings
2. **Normalization**:
    - L2 normalization projects features onto unit hypersphere
    - Enables pure angle-based similarity measurement

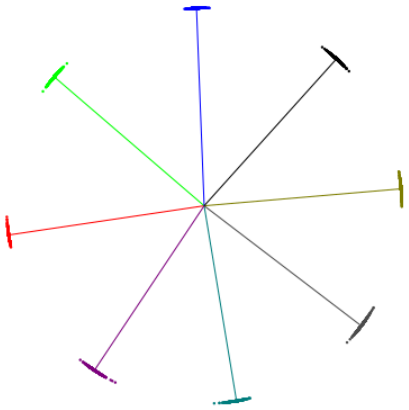
## Phase 2: ArcFace Loss Computation

```
graph TD
    A[Normalized Features] --> B[Compute Cosine Similarities]
    B --> C[Convert to Angles]
    C --> D[Add Margin to Positive Class]
    D --> E[Scale Logits]
    E --> F[Cross Entropy Loss]
```

1. **Angular Transformation**:
    - Cosine similarities converted to angles
    - Margin added only to target class angles
2. **Loss Calculation**:
    - Different treatments for positive and negative classes
    - Softmax cross-entropy on modified logits



(a) Norm-Softmax



(b) ArcFace