# Machine Learning Cheatsheet for Diabetic Retinopathy Analysis

## Dataset Understanding: APTOS 2019 Blindness Detection

- **Source**: Kaggle competition organized by Asia Pacific Tele-Ophthalmology Society (APTOS)
- **Purpose**: Encourage development of ML models to detect DR to prevent blindness
- **Size**: 3,662 images in training set, 1,928 images in test set
- **Class Labels**: Based on International Clinical Diabetic Retinopathy Disease Severity Scale
  - 0 - No DR: No visible signs of diabetic retinopathy
  - 1 - Mild: Few microaneurysms present
  - 2 - Moderate: More pronounced signs including microaneurysms, hemorrhages, and exudates
  - 3 - Severe: Significant hemorrhages and cotton-wool spots
  - 4 - Proliferative DR: Most advanced stage with neovascularization
- **Class Distribution**: Significant imbalance
  - Class 0 (No DR): ~49.3% (1,805 images)
  - Class 1 (Mild): ~10.1% (370 images)
  - Class 2 (Moderate): ~27.3% (999 images)
  - Class 3 (Severe): ~5.3% (193 images)
  - Class 4 (Proliferative): ~8.1% (295 images)
- **Image Characteristics**:
  - Variable dimensions (480×640 to 2848×4288 pixels)
  - Captured using different fundus cameras across multiple clinics
  - Varying image quality, brightness, and contrast
  - Black borders reducing effective retinal area

## Image Preprocessing Pipeline

1. **Cropping Black Borders**
   - Use thresholding to identify the retinal area
   - Find the largest contour to identify the retinal region
   - Crop to focus on the meaningful content
2. **Contrast Enhancement**
   - Apply CLAHE (Contrast Limited Adaptive Histogram Equalization)

- Improves visibility of retinal features
- Especially enhances important DR markers (microaneurysms, hemorrhages)

3. **Resizing**
   - Standardize to consistent dimensions (224×224, 300×300, or 512×512)
   - Consider resolution needs vs. computational requirements
4. **Normalization**
   - Scale pixel values to range [-1, 1] or [0, 1]
   - Improves model convergence during training
5. **Data Augmentation** (for training)
   - Address class imbalance
   - Techniques: rotation, flipping, zooming, brightness/contrast adjustments
   - Libraries: ImageDataGenerator (TF) or transforms (PyTorch)

## Model Architecture Options

### Convolutional Neural Networks (CNNs)

#### Pre-trained Models (Transfer Learning)

| Model | Characteristics | Advantages | Considerations |
|---|---|---|---|
| ResNet | Residual connections, deep architecture | Prevents vanishing gradients, good feature extraction | Deeper versions require more compute |
| EfficientNet | Optimized depth/width/resolution scaling | Better performance with fewer parameters | Complex architecture |
| DenseNet | Dense connections between layers | Feature reuse, fewer parameters | Memory intensive during training |
| Inception | Parallel convolutional paths | Captures features at multiple scales | Complex architecture to implement |
| VGG | Simple sequential architecture | Easy to understand and implement | More parameters, potentially overfit |

#### Custom Architecture Components

- **Input Layer**: Matches preprocessing dimensions (e.g., 224×224×3)
- **Convolutional Layers**: Feature extraction with varying filter sizes
- **Pooling Layers**: Reduce spatial dimensions, extract dominant features

- **Batch Normalization**: Stabilizes training, improves convergence
- **Dropout**: Prevents overfitting (typically 0.2-0.5 rate)
- **Global Average Pooling**: Reduces parameters vs. flattening
- **Dense Layers**: Final classification layers
- **Output Layer**: 5 neurons with softmax activation for class probabilities

## Training Considerations

### Handling Class Imbalance

- **Class Weights**: Assign higher weights to minority classes in loss function
- **Weighted Sampling**: Oversample minority classes during batch creation
- **Focal Loss**: Modification of cross-entropy to focus on hard examples
- **Label Smoothing**: Prevents overconfidence in predictions

### Hyperparameters

| Parameter | Range | Notes |
|---|---|---|
| Learning Rate | 1e-4 to 1e-2 | Consider scheduling (reduce on plateau) |
| Batch Size | 16-64 | Smaller for limited GPU memory |
| Epochs | 25-100 | Use early stopping based on validation performance |
| Optimizer | Adam, RMSprop | Adam often performs well for medical imaging |

### Training/Validation Split

- **Stratified Split**: Maintain class distribution across splits
- **Split Ratio**: 70-80% training, 10-15% validation, 10-15% testing
- **Cross-Validation**: Consider k-fold for more robust evaluation

## Evaluation Metrics

| Metric | Description | When to Use |
| --- | --- | --- |
| Quadratic Weighted Kappa (QWK) | Measures agreement between predicted and actual ratings | Primary metric for APTOS competition |
| Accuracy | Overall correct predictions | Less appropriate with class imbalance |
| Sensitivity/Recall | TP/(TP+FN) | Critical for medical screening (miss fewer cases) |
| Specificity | TN/(TN+FP) | Measures ability to correctly identify negatives |
| AUC-ROC | Area under receiver operating characteristic curve | Overall discriminative ability |
| Confusion Matrix | Visualizes predictions across all classes | Detailed error analysis |
| Classification Report | Per-class precision, recall, F1-score | Evaluate performance on individual classes |

## Implementation Frameworks

### TensorFlow/Keras

python

```python
# Basic model architecture example
def create_model(input_shape=(224, 224, 3), num_classes=5):
    base_model = tf.keras.applications.EfficientNetB3(
        include_top=False,
        weights='imagenet',
        input_shape=input_shape
    )

    # Fine-tuning: freeze early layers
    for layer in base_model.layers[:100]:
        layer.trainable = False

    model = tf.keras.Sequential([
        base_model,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dropout(0.3),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(num_classes, activation='softmax')
    ])

    return model

# Data preparation with augmentation
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Class weights to handle imbalance
class_weights = {
    0: 1.0,
    1: 5.0,  # Adjust based on class distribution
    2: 1.8,
    3: 9.5,
    4: 6.2
}
```

```python
# Compile with appropriate loss and metrics
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
    loss='categorical_crossentropy',
    metrics=['accuracy', tf.keras.metrics.AUC()]
)

# Callbacks for better training
callbacks = [
    tf.keras.callbacks.EarlyStopping(patience=7, restore_best_weights=True),
    tf.keras.callbacks.ReduceLROnPlateau(factor=0.2, patience=3),
    tf.keras.callbacks.ModelCheckpoint('best_model.h5', save_best_only=True)
]
```

### PyTorch

python

```python
import torch
import torch.nn as nn
import torchvision.models as models
from torch.utils.data import DataLoader
from torchvision import transforms

# Example dataset class
class RetinalDataset(torch.utils.data.Dataset):
    def __init__(self, dataframe, img_dir, transform=None):
        self.dataframe = dataframe
        self.img_dir = img_dir
        self.transform = transform

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        img_name = os.path.join(self.img_dir, self.dataframe.iloc[idx, 0] + '.png')
        image = Image.open(img_name)

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

        label = self.dataframe.iloc[idx, 1]
        return image, label

# Data transformations
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Model definition
def get_model(num_classes=5):
    model = models.densenet121(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False

    # Replace classifier
    num_features = model.classifier.in_features
```

```python
    model.classifier = nn.Sequential(
        nn.Dropout(0.3),
        nn.Linear(num_features, 512),
        nn.ReLU(),
        nn.Dropout(0.4),
        nn.Linear(512, num_classes)
    )
    return model

# Training loop (partial example)
def train_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0

    for inputs, labels in dataloader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)

    return running_loss / len(dataloader.dataset)
```

## Common Challenges and Solutions

### Image Quality Issues

- **Problem**: Poor contrast or dark images
- **Solution**: CLAHE enhancement, adaptive gamma correction

### Overfitting

- **Problem**: Model performs well on training but poorly on validation
- **Solutions**:
  - More aggressive data augmentation
  - Increase dropout rate
  - L2 regularization
  - Early stopping

- Simpler model architecture

### Class Imbalance

- **Problem**: Poor performance on minority classes
- **Solutions**:
  - Class weighting in loss function
  - Oversampling minority classes
  - Focal loss
  - Ensemble methods with different class weightings

### Computational Resources

- **Problem**: Large images and models exceed GPU memory
- **Solutions**:
  - Gradual resizing (train on smaller images first)
  - Mixed precision training
  - Gradient accumulation
  - Efficient architectures (EfficientNet, MobileNet)

## Feature Visualization and Interpretability

- **Grad-CAM**: Highlights regions contributing to classification
- **Feature maps**: Visualize intermediate activations
- **Occlusion sensitivity**: Measure effect of masking image regions
- **Integrated gradients**: Attribution method for deep networks

## Real-world Deployment Considerations

- **Model Size**: Convert to optimized formats (ONNX, TensorRT, TFLite)
- **Inference Time**: Critical for clinical deployment
- **Threshold Tuning**: Adjust classification thresholds based on clinical needs
- **Monitoring**: Track performance drift, especially with new camera equipment
- **Explainability**: Provide visual explanations to clinicians

## Resources

- Kaggle competition: https://www.kaggle.com/competitions/aptos2019-blindness-detection

- APTOS dataset: https://www.kaggle.com/datasets/mariaherrerot/aptos2019
- Research papers:
  - "Deep Learning Approach to Diabetic Retinopathy Detection"
  - "Using Deep Learning Architectures for Detection and Classification of Diabetic Retinopathy"
  - "Automated Diabetic Retinopathy Detection Using Horizontal and Vertical Patch Division-Based Pre-Trained DenseNET with Digital Fundus Images"

## Key Takeaways

1. **Preprocessing is critical**: Quality enhancement and standardization dramatically impact model performance
2. **Address class imbalance**: Essential for clinical utility across all severity levels
3. **Transfer learning works well**: Models pre-trained on ImageNet provide strong initialization
4. **Evaluate holistically**: QWK is standard metric, but consider sensitivity for clinical relevance
5. **Interpretability matters**: Clinicians need to understand model decisions