# Model Development & Training

**CS 203: Software Tools and Techniques for AI**

Prof. Nipun Batra, IIT Gandhinagar

# Today's Agenda

- Model selection strategies
- Training best practices
- Hyperparameter optimization
- AutoML with AutoGluon
- Model checkpointing
- Transfer learning
- Training pipelines

# Model Development Lifecycle

```
Data → Feature Engineering → Model Selection
  → Training → Evaluation → Tuning → Deployment
```

Each step requires systematic approaches and tooling

# Model Selection Problem

**Challenge**: Which model to use?

Options for tabular data:

- Linear models (LogisticRegression, LinearSVR)
- Tree-based (RandomForest, XGBoost, LightGBM)
- Neural networks

Options for images:

- CNNs (ResNet, EfficientNet, Vision Transformers)

Options for text:

- Transformers (BERT, GPT, T5)

# Model Selection Strategies

### 1. Start Simple

- Baseline: LogisticRegression, RandomForest
- Establishes performance floor
- Fast to train and debug

### 2. Domain Knowledge

- Computer vision → CNNs
- Sequences → RNNs/Transformers
- Tabular → Gradient boosting

### 3. Empirical Testing

- Try multiple candidates
- Compare on validation set

# scikit-learn Model Selection

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

models = {
    'logistic': LogisticRegression(max_iter=1000),
    'random_forest': RandomForestClassifier(n_estimators=100),
    'svm': SVC()
}

for name, model in models.items():
    scores = cross_val_score(model, X_train, y_train, cv=5)
    print(f"{name}: {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Training Best Practices

1. **Separate train/val/test sets**
    - Train: fit parameters
    - Validation: select hyperparameters
    - Test: final evaluation (use once!)
2. **Stratified splitting** for imbalanced data
3. **Random seed** for reproducibility
4. **Monitor both train and val metrics** (detect overfitting)

# Train/Val/Test Split

```python
from sklearn.model_selection import train_test_split

# First split: separate test set
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Second split: create train and validation
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.25,  # 0.25 * 0.8 = 0.2
    random_state=42, stratify=y_temp
)

print(f"Train: {len(X_train)}, Val: {len(X_val)}, Test: {len(X_test)}")
# Ratio is 60:20:20
```

# Cross-Validation

```
from sklearn.model_selection import cross_validate

model = RandomForestClassifier(n_estimators=100, random_state=42)

cv_results = cross_validate(
    model, X_train, y_train,
    cv=5,  # 5-fold CV
    scoring=['accuracy', 'f1_weighted'],
    return_train_score=True
)

print(f"Val Accuracy: {cv_results['test_accuracy'].mean():.3f}")
print(f"Train Accuracy: {cv_results['train_accuracy'].mean():.3f}")
```

# Hyperparameter Optimization

**Hyperparameters**: settings chosen before training

- Learning rate, batch size, number of layers
- Cannot be learned from data

**Why optimize?**

- Default values often suboptimal
- Can improve performance 5-20%

**Methods**:

- Grid search
- Random search
- Bayesian optimization
- Hyperband

# Grid Search

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5, 10]
}

model = RandomForestClassifier(random_state=42)

grid_search = GridSearchCV(
    model, param_grid,
    cv=5, scoring='f1_weighted',
    n_jobs=-1, verbose=2
)

grid_search.fit(X_train, y_train)
print(f"Best params: {grid_search.best_params_}")
print(f"Best score: {grid_search.best_score_:.3f}")
```

# Random Search

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

param_distributions = {
    'n_estimators': randint(50, 200),
    'max_depth': [10, 20, 30, None],
    'min_samples_split': randint(2, 11),
    'min_samples_leaf': randint(1, 5)
}

random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_distributions,
    n_iter=50,  # Try 50 combinations
    cv=5, random_state=42,
    n_jobs=-1
)

random_search.fit(X_train, y_train)
```

# Grid Search vs Random Search

**Grid Search**:

- Exhaustive: tries all combinations
- Good for small search spaces
- Time: O(n^d) where d is number of hyperparameters

**Random Search**:

- Samples randomly from distributions
- Better for large search spaces
- Often finds good solutions faster
- Time: O(n_iter)

**Rule of thumb**: Random search for >3 hyperparameters

# Bayesian Optimization with Optuna

```
import optuna
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

def objective(trial):
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 50, 200),
        'max_depth': trial.suggest_int('max_depth', 5, 30),
        'min_samples_split': trial.suggest_int('min_samples_split', 2, 10),
    }

    model = RandomForestClassifier(**params, random_state=42)
    score = cross_val_score(model, X_train, y_train, cv=5).mean()
    return score

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
print(f"Best params: {study.best_params}")
```

# AutoML: Automated Machine Learning

**Automates**:

- Model selection
- Hyperparameter tuning
- Feature engineering
- Ensemble creation

**Tools**:

- AutoGluon (recommended for tabular)
- H2O AutoML
- TPOT
- Auto-sklearn

# AutoGluon for Tabular Data

```python
from autogluon.tabular import TabularPredictor

# Load data into pandas DataFrame
import pandas as pd
train_data = pd.read_csv('train.csv')

# AutoGluon automatically detects task type
predictor = TabularPredictor(
    label='target_column',
    eval_metric='f1_weighted',
    path='./ag_models/'
)

# Trains multiple models and creates ensemble
predictor.fit(
    train_data,
    time_limit=3600,  # 1 hour
    presets='best_quality'  # or 'medium_quality', 'optimize_for_deployment'
)

# Get leaderboard
leaderboard = predictor.leaderboard(train_data)
print(leaderboard)
```

# AutoGluon Presets

**optimize_for_deployment**:

- Fast inference
- Smaller models
- Use for production

**medium_quality**:

- Balanced performance/time
- Good default choice

**best_quality**:

- Maximum accuracy
- Longer training
- Use for competitions/research

# AutoGluon Prediction

```
# Make predictions
test_data = pd.read_csv('test.csv')
predictions = predictor.predict(test_data)

# Get prediction probabilities
pred_probs = predictor.predict_proba(test_data)

# Feature importance
importance = predictor.feature_importance(train_data)
print(importance.head(10))

# Model interpretation
explainer = predictor.explain(test_data.iloc[[0]])
```

# Model Checkpointing

**Why checkpoint?**

- Training can crash
- Want to resume from best epoch
- Experiment with different stopping points

**What to save**:

- Model weights
- Optimizer state
- Training metadata (epoch, loss, metrics)

# PyTorch Checkpointing

```python
import torch

# Save checkpoint
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': train_loss,
    'val_accuracy': val_acc,
}
torch.save(checkpoint, f'checkpoint_epoch_{epoch}.pt')

# Load checkpoint
checkpoint = torch.load('checkpoint_epoch_10.pt')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
start_epoch = checkpoint['epoch'] + 1
```

# Best Model Checkpointing

```
class CheckpointCallback:
    def __init__(self, path):
        self.path = path
        self.best_val_loss = float('inf')

    def __call__(self, model, val_loss, epoch):
        if val_loss < self.best_val_loss:
            self.best_val_loss = val_loss
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'val_loss': val_loss,
            }, self.path)
            print(f"Saved best model at epoch {epoch}")

callback = CheckpointCallback('best_model.pt')
# In training loop:
callback(model, val_loss, epoch)
```

# Transfer Learning

**Idea**: Use knowledge from one task to improve another

**Benefits**:

- Faster training
- Better performance with less data
- Leverages pre-trained models

**Applications**:

- Computer vision: ImageNet pre-training
- NLP: BERT, GPT fine-tuning
- Audio: wav2vec2

# Transfer Learning with PyTorch

```
import torchvision.models as models

# Load pre-trained ResNet
model = models.resnet18(pretrained=True)

# Freeze all layers
for param in model.parameters():
    param.requires_grad = False

# Replace final layer
num_features = model.fc.in_features
model.fc = torch.nn.Linear(num_features, num_classes)

# Only train the final layer
optimizer = torch.optim.Adam(model.fc.parameters(), lr=0.001)
```

# Transfer Learning: Fine-tuning

```
# Unfreeze some layers after initial training
# Fine-tune last few layers
for param in model.layer4.parameters():
    param.requires_grad = True

# Use different learning rates
optimizer = torch.optim.Adam([
    {'params': model.layer4.parameters(), 'lr': 1e-4},
    {'params': model.fc.parameters(), 'lr': 1e-3}
])
```

# Hugging Face Transformers

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from transformers import TrainingArguments, Trainer

# Load pre-trained model
model = AutoModelForSequenceClassification.from_pretrained(
    'bert-base-uncased',
    num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

# Tokenize data
def tokenize_function(examples):
    return tokenizer(examples['text'], padding='max_length', truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

# Hugging Face Training

```
training_args = TrainingArguments(
    output_dir='./results',
    evaluation_strategy='epoch',
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_strategy='epoch',
    load_best_model_at_end=True,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['test'],
)

trainer.train()
```

# Training Pipelines

**Pipeline**: End-to-end workflow from data to model

Components:

1. Data loading
2. Preprocessing
3. Feature engineering
4. Model training
5. Evaluation
6. Checkpointing

**Tools**: scikit-learn Pipeline, PyTorch Lightning, Keras

# scikit-learn Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=50)),
    ('classifier', RandomForestClassifier(n_estimators=100))
])

# Entire pipeline in one fit
pipeline.fit(X_train, y_train)

# Applies all transformations + prediction
y_pred = pipeline.predict(X_test)

# Can use in GridSearchCV
param_grid = {
    'pca__n_components': [30, 50, 70],
    'classifier__n_estimators': [50, 100, 200]
}
grid_search = GridSearchCV(pipeline, param_grid, cv=5)
```

# PyTorch Lightning

```
import pytorch_lightning as pl

class LitModel(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = ...

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = F.cross_entropy(y_hat, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = F.cross_entropy(y_hat, y)
        self.log('val_loss', loss)

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=1e-3)
```

# PyTorch Lightning Training

```
model = LitModel()

trainer = pl.Trainer(
    max_epochs=10,
    accelerator='gpu',
    devices=1,
    callbacks=[
        pl.callbacks.ModelCheckpoint(monitor='val_loss'),
        pl.callbacks.EarlyStopping(monitor='val_loss', patience=3)
    ],
    logger=pl.loggers.TensorBoardLogger('logs/')
)

trainer.fit(model, train_loader, val_loader)
```

**Benefits**: Reduces boilerplate, automatic checkpointing, multi-GPU support

# Early Stopping

**Idea**: Stop training when validation performance stops improving

**Prevents**:

- Overfitting
- Wasted computation

```
from sklearn.model_selection import learning_curve

# PyTorch Lightning
early_stop = pl.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,  # Wait 5 epochs
    mode='min'
)

trainer = pl.Trainer(callbacks=[early_stop])
```

# Learning Rate Scheduling

**Problem**: Fixed learning rate not optimal

**Solution**: Adjust during training

**Strategies**:

- Step decay
- Exponential decay
- Cosine annealing
- Reduce on plateau

# PyTorch LR Schedulers

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Reduce on plateau
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5
)

# In training loop
for epoch in range(num_epochs):
    train_loss = train_epoch(model, train_loader, optimizer)
    val_loss = validate(model, val_loader)

    # Update learning rate
    scheduler.step(val_loss)

    current_lr = optimizer.param_groups[0]['lr']
    print(f"Epoch {epoch}, LR: {current_lr:.6f}")
```

# Mixed Precision Training

**Idea**: Use float16 instead of float32

**Benefits**:

- 2x faster training
- 2x less memory
- Same accuracy (with loss scaling)

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for batch in train_loader:
    optimizer.zero_grad()

    with autocast():
        output = model(batch)
        loss = criterion(output, target)

    scaler.scale(loss).backward()
    scaler.step(optimizer)
```

# Gradient Clipping

**Problem**: Exploding gradients

**Solution**: Clip gradients to maximum norm

```
# PyTorch
max_norm = 1.0
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)

# PyTorch Lightning (automatic)
trainer = pl.Trainer(gradient_clip_val=1.0)
```

# Experiment Tracking

**Track**:

- Hyperparameters
- Metrics (loss, accuracy)
- Model checkpoints
- Code version
- Environment

**Tools**:

- Weights & Biases
- MLflow
- TensorBoard
- Neptune

# Weights & Biases Integration

```python
import wandb

# Initialize
wandb.init(project='my-project', config={
    'learning_rate': 0.001,
    'epochs': 10,
    'batch_size': 32
})

# Log metrics
for epoch in range(num_epochs):
    train_loss = train_epoch(...)
    val_acc = validate(...)

    wandb.log({
        'epoch': epoch,
        'train_loss': train_loss,
        'val_accuracy': val_acc
    })

# Save model
wandb.save('model.pt')
```

# Model Versioning

**Why version models?**

- Reproduce results
- Compare experiments
- Rollback deployments

**What to version**:

- Model architecture
- Trained weights
- Training code
- Data version
- Hyperparameters

**Tools**: DVC, MLflow Model Registry, W&B Artifacts

# DVC for Model Versioning

```
# Track model file
dvc add models/my_model.pt

# Commit to git
git add models/my_model.pt.dvc .gitignore
git commit -m "Add trained model v1"

# Push model to remote storage
dvc push

# On another machine, pull model
dvc pull models/my_model.pt.dvc
```

# Training on GPUs

```
# PyTorch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

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

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

# PyTorch Lightning (automatic)
trainer = pl.Trainer(accelerator='gpu', devices=1)
```

# Distributed Training

**Why?**

- Train larger models
- Faster training

**Strategies**:

- Data parallelism: split data across GPUs
- Model parallelism: split model across GPUs

```
# PyTorch Lightning multi-GPU
trainer = pl.Trainer(
    accelerator='gpu',
    devices=4,  # Use 4 GPUs
    strategy='ddp'  # Distributed Data Parallel
)
```

# Best Practices Summary

1. Start with baselines
2. Use cross-validation
3. Separate train/val/test sets
4. Track experiments (W&B, MLflow)
5. Version models and data
6. Checkpoint frequently
7. Monitor both train and val metrics
8. Use transfer learning when possible
9. Consider AutoML for tabular data
10. Automate with pipelines

# Common Pitfalls

**Data leakage**:

- Scaling before splitting
- Using test set for tuning

**Overfitting**:

- Too complex model
- Too long training
- No regularization

**Poor evaluation**:

- Using test set multiple times
- Incorrect metrics for imbalanced data

# Key Takeaways

- Model selection: start simple, use domain knowledge
- Hyperparameter tuning: Random search or Bayesian optimization
- AutoML: AutoGluon for tabular, Hugging Face for NLP
- Checkpointing: save best model, enable resuming
- Transfer learning: leverage pre-trained models
- Pipelines: automate end-to-end workflow
- Track experiments for reproducibility

# Resources

- AutoGluon: autogluon.ai
- Optuna: optuna.org
- PyTorch Lightning: lightning.ai
- Hugging Face: huggingface.co
- Weights & Biases: wandb.ai
- scikit-learn Pipeline: scikit-learn.org/stable/modules/compose.html

# Lab Preview

Hands-on exercises:

1. Model selection with cross-validation
2. Hyperparameter tuning with Optuna
3. AutoML with AutoGluon
4. Training pipeline with scikit-learn
5. Transfer learning with PyTorch
6. Experiment tracking with W&B