Model Development & Training

CS 203: Software Tools and Techniques for AI

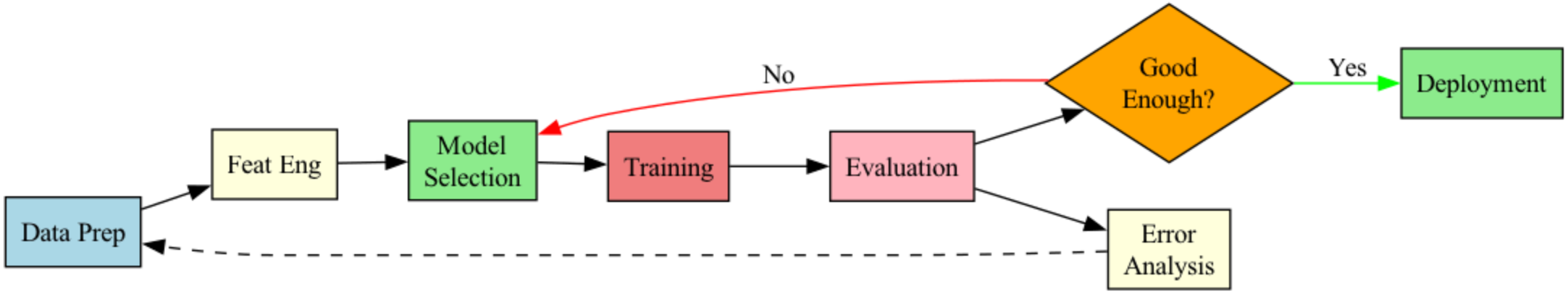
Prof. Nipun Batra, IIT Gandhinagar

Today's Agenda

- Model selection strategies
- Training best practices
- Hyperparameter optimization
- AutoML with AutoGluon
- Model checkpointing
- Transfer learning
- **Fine-tuning LLMs** (New!)
- Training pipelines

Model Development Lifecycle

It's not just `model.fit()`. It's a cycle.



Iterative Process:

1. Start simple (Baseline).
2. Analyze errors.
3. Add complexity (New features, complex models).

Model Selection Strategy

Don't start with a Transformer. Start with a baseline.

Data Type	Baseline (Fast, Simple)	Advanced (SOTA, Heavy)
Tabular	Logistic Regression, Decision Tree	XGBoost, LightGBM, TabNet
Image	ResNet-18	EfficientNet, ViT (Vision Transformer)
Text	TF-IDF + Naive Bayes	BERT, RoBERTa, GPT (Fine-tuned)
Time Series	ARIMA, Linear Regression	LSTM, Transformer

Why Baselines?

- Debug pipeline bugs quickly.
- Establish a "floor" for performance.
- If deep learning only gives +1% over Logistic Regression, is it worth the cost?

Model Selection Criteria

Consider when choosing models:

1. **Data size**: Small data → simpler models (avoid overfitting)
2. **Feature types**: Mixed types → tree-based models
3. **Interpretability**: Need explanations → linear models, decision trees
4. **Latency requirements**: Real-time → fast models (linear, small trees)
5. **Compute budget**: Limited resources → avoid deep learning
6. **Maintenance**: Production → stable, well-supported models

No Free Lunch Theorem: No single best model for all problems.

Evaluation Metrics

Classification metrics:

Metric	Formula	Use When
Accuracy	(TP + TN) / Total	Balanced classes
Precision	TP / (TP + FP)	Minimize false positives
Recall	TP / (TP + FN)	Minimize false negatives
F1-Score	2 × (Precision × Recall) / (Precision + Recall)	Balance P and R
AUC-ROC	Area under ROC curve	Overall model quality

Regression metrics:

- **MAE** (Mean Absolute Error): Average error magnitude
- **MSE** (Mean Squared Error): Penalizes large errors

Confusion Matrix Deep Dive

For binary classification:

```
                Predicted
              Positive  Negative
Actual Pos      TP        FN
       Neg      FP        TN
```

Example: Cancer detection

- **TP** (True Positive): Correctly identified cancer
- **FP** (False Positive): Healthy flagged as cancer (unnecessary treatment)
- **FN** (False Negative): Cancer missed (dangerous!)
- **TN** (True Negative): Correctly identified healthy

Choosing metric:

Precision vs Recall Trade-off

Setting decision threshold:

```
# Default threshold = 0.5
y_pred = (model.predict_proba(X)[:, 1] >= 0.5).astype(int)

# High precision (few false positives)
y_pred_high_precision = (model.predict_proba(X)[:, 1] >= 0.8).astype(int)

# High recall (few false negatives)
y_pred_high_recall = (model.predict_proba(X)[:, 1] >= 0.2).astype(int)
```

Precision-Recall curve: Visualize trade-off at all thresholds.

ROC curve: True Positive Rate vs False Positive Rate.

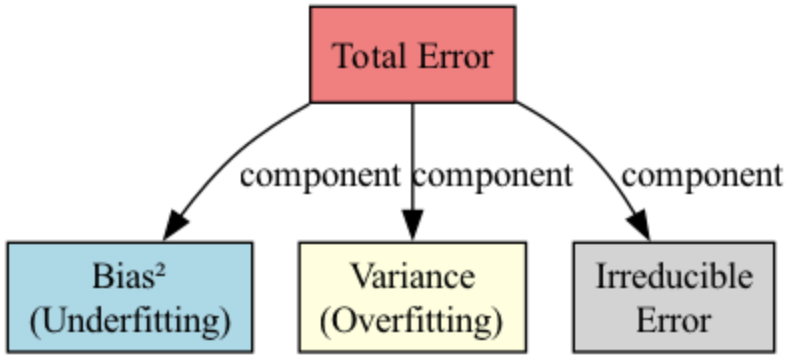
AUC-ROC = 1.0: Perfect classifier

AUC-ROC = 0.5: Random guessing

Bias vs. Variance Trade-off

Bias (Underfitting): Model is too simple to capture patterns.

Variance (Overfitting): Model memorizes noise in training data.



Goal: Sweet spot where Total Error is minimized.

Fixing Overfitting:

- More data
- Regularization (L1/L2, Dropout)
- Simpler model

Learning Curves: Diagnosing Problems

Training curve (loss over epochs):

- Still decreasing → train more
- Flattened → training complete

Learning curve (performance vs data size):

- High bias (underfitting): Both curves plateau at low performance
- High variance (overfitting): Large gap between train/val

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, val_scores = learning_curve(
    model, X, y, cv=5,
    train_sizes=np.linspace(0.1, 1.0, 10)
)
```

Regularization Techniques

Prevent overfitting by adding constraints.

L1 Regularization (Lasso):

- Loss = MSE + $\lambda \times \Sigma|w|$
- Drives some weights to exactly zero
- **Effect**: Feature selection (sparse models)

L2 Regularization (Ridge):

- Loss = MSE + $\lambda \times \Sigma w^2$
- Shrinks all weights toward zero
- **Effect**: Reduces model complexity

ElasticNet: Combines L1 + L2

Dropout (Neural Networks)

Randomly drop neurons during training.

```
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.dropout = nn.Dropout(p=0.5)  # Drop 50% of neurons
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)  # Only active during training
        x = self.fc2(x)
        return x
```

Why it works:

Data Splitting Strategies

1. **Hold-out Set**:
   - Train (60%), Validation (20%), Test (20%).
   - **Risk**: Validation set might be lucky/unlucky.
2. **K-Fold Cross-Validation**:
   - Robust estimate of performance.
   - Train K times on K different splits.
3. **Stratified K-Fold**:
   - Maintains class distribution in each fold
   - **Critical for imbalanced datasets**

```
from sklearn.model_selection import StratifiedKFold
```

Time Series Splitting

Problem: Can't shuffle time series data (leakage!).

Time Series CV:

```
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)

for train_idx, test_idx in tscv.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    # Always: train comes before test
```

Expanding window: Train set grows each fold

Sliding window: Fixed-size train set

Walk-forward validation: Retrain after each prediction.

Feature Engineering Principles

Good features = better performance than complex models.

Types of features:

1. **Numerical transformations**:
    - Log transform (reduce skew)
    - Polynomial features (capture non-linearity)
    - Binning/discretization
2. **Categorical encoding**:
    - One-hot encoding (low cardinality)
    - Target encoding (high cardinality)
    - Frequency encoding

Feature Importance

Identify which features matter:

Tree-based (built-in):

```
importance = model.feature_importances_
feature_importance_df = pd.DataFrame({
    'feature': X.columns,
    'importance': importance
}).sort_values('importance', ascending=False)
```

Permutation importance (model-agnostic):

```
from sklearn.inspection import permutation_importance

result = permutation_importance(
    model, X_val, y_val,
    n_repeats=10,
    random_state=42
```

Hyperparameter Optimization (HPO)

Parameters: Learned from data (Weights, Biases).

Hyperparameters: Set *before* training (Learning Rate, Batch Size, Depth).

Search Strategies:

1. **Grid Search**: Try *every* combination.
    - Safe but exponentially expensive ($O(N^D)$).
2. **Random Search**: Randomly sample configurations.
    - Surprisingly effective.
3. **Bayesian Optimization (Optuna)**: Smart search.
    - "Given that `lr=0.1` was bad, don't try `lr=0.2`, try `lr=0.01`."

Bayesian Optimization Visualized

How it works:

1. Build a probability model of the objective function.
2. Choose next hyperparameter to query (Exploration vs Exploitation).
3. Update model.

Optuna Code:

```
def objective(trial):
    # Suggest params
    lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    depth = trial.suggest_int('depth', 3, 10)

    model = Train(lr, depth)
    return model.val_accuracy

study.optimize(objective, n_trials=100)
```

Ensemble Methods

Wisdom of crowds: Combine multiple models for better performance.

Three main strategies:

1. **Bagging** (Bootstrap Aggregating)
    - Train models on random subsets of data
    - Average predictions (regression) or vote (classification)
    - Example: **Random Forest** (ensemble of decision trees)
2. **Boosting**
    - Train models sequentially
    - Each model corrects previous model's errors
    - Example: **XGBoost, LightGBM, AdaBoost**

Bagging vs Boosting

Aspect	Bagging	Boosting
Training	Parallel	Sequential
Goal	Reduce variance	Reduce bias
Weighting	Equal	Focuses on hard examples
Overfitting	Less prone	Can overfit
Example	Random Forest	XGBoost, AdaBoost

When to use:

- **Bagging**: High variance models (deep trees)
- **Boosting**: High bias models (weak learners)

Gradient Boosting Explained

Intuition: Each tree corrects residual errors of previous trees.

```
# Simplified gradient boosting
predictions = 0

for i in range(n_trees):
    # Calculate residuals (errors)
    residuals = y_true - predictions

    # Train tree to predict residuals
    tree = DecisionTree().fit(X, residuals)

    # Update predictions
    predictions += learning_rate * tree.predict(X)
```

Hyperparameters:

- `n_estimators`: Number of trees (more = better, but slower)

Class Imbalance Problem

Scenario: 99% normal transactions, 1% fraud.

Naive accuracy: Predict all "normal" → 99% accuracy (useless!).

Solutions:

1. **Resampling**:
    - **Oversample minority**: Duplicate rare class (SMOTE)
    - **Undersample majority**: Remove common class
2. **Class weights**:
    - Penalize misclassifying minority class more
3. **Ensemble methods**:
    - BalancedRandomForest, EasyEnsemble

Handling Imbalanced Data

SMOTE (Synthetic Minority Oversampling):

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Class weights (penalize minority class errors more):

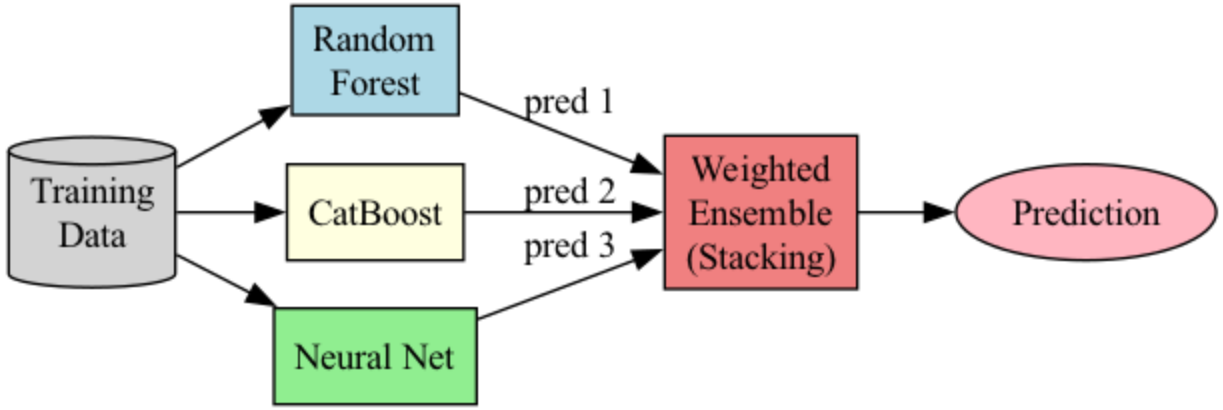
```
from sklearn.linear_model import LogisticRegression

# Automatically balance
model = LogisticRegression(class_weight='balanced')

# Manual weights
model = LogisticRegression(class_weight={0: 1, 1: 10})
```


Stratified sampling: Ensure minority class in all splits

AutoML: Automated Machine Learning

Philosophy: "I don't care which model, just give me the best one."

AutoGluon (Amazon) creates a stacked ensemble of models.



Pros: SOTA performance with 3 lines of code.

Cons: Slow training, heavy inference, hard to interpret.

AutoML Code Example

```
from autogluon.tabular import TabularPredictor

# That's it!
predictor = TabularPredictor(label='target').fit(train_data)

# Evaluate
predictor.leaderboard(test_data)

# Predict
predictions = predictor.predict(test_data)
```

What AutoGluon does:

1. Feature engineering (one-hot encoding, etc.)
2. Trains 10-20 different models
3. Stacks them into ensemble

Training Pipelines

Spaghetti code in notebooks is the enemy of reproducibility.

Pipeline: A reproducible recipe.

`Raw Data` -> `Imputer` -> `Scaler` -> `Encoder` -> `Model`

scikit-learn Pipeline:

```
pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=50)),
    ('clf', RandomForestClassifier())
])
pipe.fit(X_train, y_train)
```

Benefits: No data leakage! Transforms are fit *only* on train splits during CV.

Checkpointing & Early Stopping

The "Epoch" Dilemma:

- Train too little -> Underfit.
- Train too much -> Overfit.

Early Stopping:

- Monitor Validation Loss.
- If it stops decreasing for `patience` epochs -> **STOP**.

Checkpointing:

- Save the model weights *every time* validation loss improves.
- Restore the *best* version at the end.

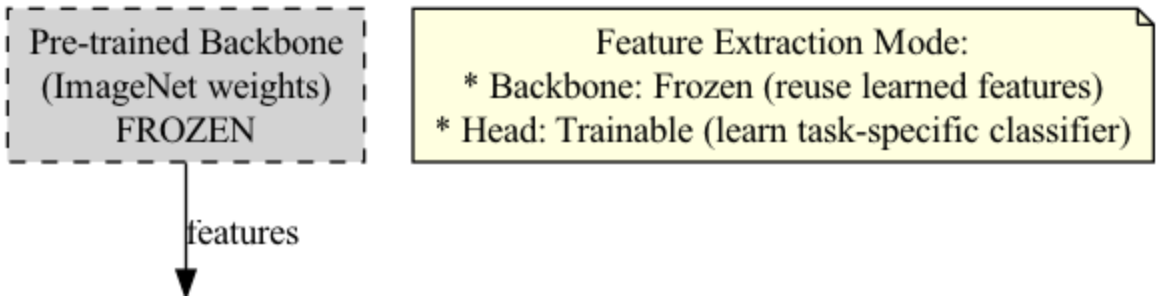
Transfer Learning: Theory

Don't reinvent the wheel.

Someone (Google/Meta) spent $10M to train a model on ImageNet (14M images). It learned to see edges, textures, shapes.

Strategies:

1. **Feature Extraction**: Freeze backbone, train only the head (classifier).
    - Fast, low data requirement.
2. **Fine-Tuning**: Unfreeze backbone (or parts of it) and train with low learning rate.
    - Slower, needs more data, higher accuracy.



Fine-Tuning LLMs (PEFT & LoRA)

The Problem: Fine-tuning a 7B parameter model requires ~100GB+ VRAM.
Solution: Parameter-Efficient Fine-Tuning (PEFT).

LoRA (Low-Rank Adaptation):

- Freeze original weights W.
- Add small trainable rank decomposition matrices A and B.
- $W' = W + BA$
- Trainable parameters reduced by 10,000x!

Use Cases:

- Adapting generic LLM to specific domain (Medical, Legal).
- Changing style/tone (Chatbot persona).

Hugging Face PEFT Example

```
from peft import LoraConfig, get_peft_model, TaskType

# 1. Define LoRA Config
peft_config = LoraConfig(
    task_type=TaskType.SEQ_CLS,
    inference_mode=False,
    r=8,            # Rank
    lora_alpha=32,  # Scaling factor
    lora_dropout=0.1
)

# 2. Wrap Base Model
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased")
model = get_peft_model(model, peft_config)

# 3. Train as usual (only 0.5% params are trainable!)
model.print_trainable_parameters()
```

Experiment Tracking

Problem: "I trained 50 models. Which one had `lr=0.001` and `dropout=0.5`?"

Solution: Use a tracker (Weights & Biases, MLflow).

What to track:

- **Config**: Hyperparameters (yaml/json).
- **Metrics**: Loss, Accuracy, F1 (charts).
- **Artifacts**: Model weights (`.pt`), dataset versions.
- **System**: GPU usage, memory.

Best Practices Summary

1. **Baseline First**: Always beat a dummy classifier.
2. **Leakage Free**: Use Pipelines.
3. **Track Everything**: Use W&B/MLflow.
4. **Save Often**: Checkpoints are life-savers.
5. **Be Lazy**: Use Transfer Learning and AutoML where possible.

Optimization Algorithms Deep Dive

Stochastic Gradient Descent (SGD):

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

SGD with Momentum:

$$v_{t+1} = \gamma v_t + \eta \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

- Accelerates convergence
- Reduces oscillations

Adam (Adaptive Moment Estimation):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Learning Rate Schedules

Fixed LR: Same rate throughout training (often suboptimal).

Step Decay:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/k \rfloor}$$

Reduce LR every k epochs.

Cosine Annealing:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{t\pi}{T}))$$

Warmup + Cosine:

```
def get_lr(epoch, warmup_epochs=5, total_epochs=100):
    if epoch < warmup_epochs:
        # Linear warmup
        return epoch / warmup_epochs * lr_max
```

Regularization: L1 and L2

Prevent overfitting by adding constraints to the loss function.

L1 Regularization (Lasso):

$$L = L_{data} + \lambda \sum_i |\theta_i|$$

- Drives some weights to exactly zero
- **Effect**: Feature selection (sparse models)
- **Use case**: When you have many irrelevant features

L2 Regularization (Ridge):

$$L = L_{data} + \lambda \sum_i \theta_i^2$$

- Shrinks all weights toward zero

Regularization: Dropout and Early Stopping

Dropout (for neural networks):

```
# Training: randomly drop neurons
mask = np.random.binomial(1, keep_prob, size=activations.shape)
activations = activations * mask / keep_prob  # Scale up

# Inference: use all neurons (no dropout)
```

Why it works:

- Forces network to not rely on specific neurons
- Acts like training ensemble of sub-networks
- Typical dropout rate: 0.2-0.5

Early Stopping:

Batch Normalization: Theory

Problem: Internal Covariate Shift

- Layer inputs change during training
- Slows down training
- Requires careful initialization

Solution: Normalize activations within each mini-batch

Formula:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma \hat{x} + \beta$$

where μ_B, σ_B = batch mean/std, γ, β = learnable parameters

Batch Normalization: Implementation

Benefits:

- Faster training (can use higher learning rates)
- Less sensitive to initialization
- Acts as regularization (slight noise from batch statistics)
- Almost always improves performance

PyTorch Implementation:

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(784, 256),
    nn.BatchNorm1d(256),  # After linear, before activation
    nn.ReLU(),
    nn.Linear(256, 10)
```

Layer Normalization vs Batch Normalization

Batch Norm: Normalize across batch dimension.

- For each feature, compute mean/std across batch samples

Layer Norm: Normalize across feature dimension.

- For each sample, compute mean/std across all features

When to use:

- **Batch Norm**: CNNs, large batches
- **Layer Norm**: Transformers, RNNs, small batches

```
# Batch Norm (for CNNs)
nn.BatchNorm2d(num_features=64)

# Layer Norm (for Transformers)
nn.LayerNorm(normalized_shape=512)
```

Gradient Descent Variants

Batch Gradient Descent:

- Use entire dataset per update
- Slow, memory-intensive
- Smooth convergence

Stochastic Gradient Descent (SGD):

- One sample per update
- Fast, noisy
- Can escape local minima

Mini-Batch Gradient Descent:

- Batch of 32-512 samples per update

Ensemble Methods Theory

Bagging (Bootstrap Aggregating):

1. Create N bootstrap samples
2. Train model on each
3. Average predictions

Example: Random Forest = Bagging + Decision Trees

Boosting: Sequentially train models, focus on errors.

AdaBoost:

$$\alpha_t = \frac{1}{2} \log \frac{1 - \epsilon_t}{\epsilon_t}$$

Weight misclassified examples higher.

Model Calibration

Problem: Model outputs probabilities, but are they reliable?

Calibration: P(y=1|score=0.7) should actually be 70%.

Reliability Diagram:

```
from sklearn.calibration import calibration_curve

fraction_positive, mean_predicted = calibration_curve(
    y_true, y_probs, n_bins=10
)

plt.plot(mean_predicted, fraction_positive)
plt.plot([0, 1], [0, 1], 'k--')  # Perfect calibration
```

Temperature Scaling:

$$P_{calibrated}(y|x) = \text{softmax}(z/T)$$

Multi-Task Learning

Idea: Train one model on multiple related tasks simultaneously.

Architecture:

```
class MultiTaskModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Shared encoder
        self.shared = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128)
        )

        # Task-specific heads
        self.task1_head = nn.Linear(128, num_classes1)
        self.task2_head = nn.Linear(128, num_classes2)

    def forward(self, x):
        shared_repr = self.shared(x)
        out1 = self.task1_head(shared_repr)
```

Curriculum Learning

Idea: Train on easy examples first, gradually increase difficulty.

Implementation:

```
def curriculum_learning(model, data, epochs_per_stage=10):
    # Stage 1: Easy examples (high confidence labels)
    easy_data = data[data['confidence'] > 0.9]
    train(model, easy_data, epochs=epochs_per_stage)

    # Stage 2: Medium difficulty
    medium_data = data[data['confidence'].between(0.7, 0.9)]
    train(model, easy_data + medium_data, epochs=epochs_per_stage)

    # Stage 3: All data
    train(model, data, epochs=epochs_per_stage)
```

Applications:

Neural Architecture Search (NAS) Details

Search space: All possible architectures (layers, connections).

Search strategy:

1. **Random search**: Try random architectures
2. **Evolutionary algorithms**: Mutate/crossover architectures
3. **Reinforcement learning**: RL agent proposes architectures
4. **Gradient-based (DARTS)**: Differentiable architecture search

DARTS intuition:

- Relax discrete choices to continuous
- Use gradient descent to find best architecture
- Discretize at the end

Mixed Precision Training

Idea: Use FP16 (16-bit) instead of FP32 (32-bit) to save memory and time.

Automatic Mixed Precision (AMP):

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

model = model.cuda()
optimizer = torch.optim.Adam(model.parameters())
scaler = GradScaler()

for x, y in dataloader:
    optimizer.zero_grad()

    # Forward pass in FP16
    with autocast():
        outputs = model(x)
        loss = criterion(outputs, y)
```

Distributed Training

Data Parallel: Split batch across GPUs.

```
model = nn.DataParallel(model, device_ids=[0, 1, 2, 3])

# Each GPU gets batch_size / num_gpus samples
# Gradients are averaged across GPUs
```

Distributed Data Parallel (DDP): More efficient.

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

# Initialize process group
dist.init_process_group(backend='nccl')

model = DDP(model, device_ids=[local_rank])

# Each GPU trains independently, syncs gradients
```

Gradient Accumulation

Problem: Want large batch size, but GPU memory is limited.

Solution: Accumulate gradients over multiple mini-batches.

```
accumulation_steps = 4
optimizer.zero_grad()

for i, (x, y) in enumerate(dataloader):
    outputs = model(x)
    loss = criterion(outputs, y)

    # Scale loss by accumulation steps
    loss = loss / accumulation_steps
    loss.backward()

    if (i + 1) % accumulation_steps == 0:
        # Update weights every N mini-batches
        optimizer.step()
        optimizer.zero_grad()
```

Feature Engineering for Neural Networks

Normalization:

```
from sklearn.preprocessing import StandardScaler

# Z-score normalization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)

# Min-max scaling (for bounded inputs like images)
X_scaled = (X - X.min()) / (X.max() - X.min())
```

Categorical encoding:

```
# One-hot encoding
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)

# Embedding (for high-cardinality categorical)
```

Debugging Training Issues

Symptom: Loss is NaN

- Cause: Exploding gradients
- Fix: Gradient clipping, lower learning rate

Symptom: Loss not decreasing

- Cause: Learning rate too low/high, wrong optimizer
- Fix: Try LR finder, use Adam

Symptom: High train accuracy, low val accuracy

- Cause: Overfitting
- Fix: Regularization, more data, early stopping

Symptom: Both train and val accuracy low

Lab Preview

Hands-on exercises:

1. **Manual**: Compare SVM vs Random Forest with Cross-Validation.
2. **Automated**: Use AutoGluon to beat your manual models.
3. **Optimization**: Use Optuna to tune the Random Forest.
4. **Tracking**: Log everything to W&B.
5. **(Advanced)**: Fine-tune a small BERT model using LoRA.

Let's code!