TECHNICAL EXPLANATIONS

Understanding Every Component

Universal Food Recipe Generator

Capstone Project

This document explains what each technology, concept, and component does and why we chose to use them in this project.

1. Deep Learning Fundamentals

1.1 What is Deep Learning?

Deep learning is a subset of machine learning that uses artificial neural networks with multiple layers (hence "deep") to learn patterns from data. Unlike traditional programming where we write explicit rules, deep learning models learn rules automatically from examples.

HOW IT WORKS:
1. We show the model thousands of images with labels (e.g., "this is pizza", "this is naan")
2. The model adjusts its internal parameters to recognize patterns
3. After training, it can classify new, unseen images

WHY WE USE IT:
- Can learn complex patterns that are hard to program manually
- Works well with images, text, and other unstructured data
- Achieves human-level or better performance on many tasks

1.2 What is a Neural Network?

A neural network is a computational model inspired by the human brain. It consists of layers of interconnected "neurons" that process information.

STRUCTURE:
- Input Layer: Receives the data (image pixels in our case)
- Hidden Layers: Process and transform the data
- Output Layer: Produces the final prediction (food category)

EACH NEURON:
1. Receives inputs from previous layer
2. Multiplies each input by a "weight" (learned parameter)
3. Adds all weighted inputs together
4. Applies an "activation function" to produce output

LEARNING PROCESS:
- The network makes predictions
- We calculate the "error" (difference from correct answer)
- We adjust weights to reduce error (backpropagation)
- Repeat millions of times until accurate

1.3 What is a Convolutional Neural Network (CNN)?

A CNN is a special type of neural network designed specifically for processing images. It's the foundation of modern computer vision.

WHY CNNs FOR IMAGES?
Regular neural networks treat each pixel independently, ignoring spatial relationships. CNNs preserve spatial information by using "convolutions."

HOW CONVOLUTION WORKS:
1. A small filter (e.g., 3x3 pixels) slides across the image
2. At each position, it computes a weighted sum
3. This creates a "feature map" highlighting certain patterns
4. Different filters detect different features (edges, textures, shapes)

CNN LAYERS:
- Convolutional Layers: Detect features (edges, corners, textures)
- Pooling Layers: Reduce size while keeping important information
- Fully Connected Layers: Make final classification decision

HIERARCHICAL LEARNING:
- Early layers: Learn simple features (edges, colors)
- Middle layers: Learn combinations (corners, textures)
- Late layers: Learn complex patterns (eyes, wheels, food items)

2. EfficientNet Architecture

2.1 What is EfficientNet?

EfficientNet is a family of CNN models developed by Google that achieve state-of-the-art accuracy while being computationally efficient. It was introduced in 2019 and revolutionized how we think about scaling neural networks.

THE KEY INNOVATION - COMPOUND SCALING:
Traditional approaches scale networks in one dimension:
- Deeper (more layers)
- Wider (more channels)
- Higher resolution (larger images)

EfficientNet scales ALL THREE dimensions together in a balanced way, achieving better accuracy with fewer parameters.

EFFICIENTNET VARIANTS:
- B0: Baseline (5.3M parameters) - What we used
- B1-B7: Progressively larger and more accurate
- B7: Largest (66M parameters)

WHY WE CHOSE B0:
1. Fits in our 4GB GPU memory
2. Fast inference for web application
3. Still achieves excellent accuracy
4. Pre-trained weights available

2.2 EfficientNet-B0 Architecture Details

EfficientNet-B0 uses a specific building block called MBConv (Mobile Inverted Bottleneck Convolution).

MBCONV BLOCK STRUCTURE:
1. Expansion: Increase channels (1x1 convolution)
2. Depthwise Convolution: Spatial filtering (3x3 or 5x5)
3. Squeeze-and-Excitation: Channel attention mechanism
4. Projection: Reduce channels back (1x1 convolution)
5. Skip Connection: Add input to output (residual learning)

SQUEEZE-AND-EXCITATION (SE):
This is an attention mechanism that helps the network focus on important channels:
1. Global average pooling (squeeze)
2. Two fully connected layers (excitation)
3. Sigmoid activation produces channel weights

4. Multiply feature map by these weights

WHAT THIS ACHIEVES:
- More important features get amplified
- Less important features get suppressed
- Network learns WHAT to pay attention to

2.3 Pre-trained on ImageNet

Our EfficientNet-B0 was pre-trained on ImageNet, a massive dataset of 1.2 million images across 1000 categories.

WHAT IMAGENET CONTAINS:
- Animals (dogs, cats, birds, fish, insects)
- Objects (cars, furniture, tools, electronics)
- Scenes (beach, mountain, office, kitchen)
- Food items (some overlap with our task!)

WHY PRE-TRAINING HELPS:
The network already learned to recognize:
- Edges and contours
- Textures (smooth, rough, shiny)
- Shapes (round, rectangular, irregular)
- Colors and color patterns
- Spatial relationships

These low-level features are UNIVERSAL - they apply to food recognition too! We just need to fine-tune the high-level features for our specific food categories.

3. Transfer Learning

3.1 What is Transfer Learning?

Transfer learning is a technique where we take a model trained on one task and adapt it for a different but related task. Instead of training from scratch, we "transfer" the learned knowledge.

ANALOGY:
Imagine learning to play guitar. If you already know piano:
- You understand music theory (transfers!)
- You have finger dexterity (transfers!)
- You just need to learn guitar-specific techniques

Similarly, a network trained on ImageNet:
- Already understands visual features
- Already knows how to process images
- Just needs to learn food-specific patterns

WHY IT'S POWERFUL:
1. Requires less training data (we had only 113K images)
2. Trains faster (hours instead of days/weeks)
3. Often achieves better accuracy
4. Works even when source and target domains differ

3.2 How We Applied Transfer Learning

We used a two-phase approach to transfer learning:

PHASE 1: FEATURE EXTRACTION (Warmup)
- Freeze all EfficientNet layers (don't update weights)
- Only train the new classifier head
- Purpose: Learn to map ImageNet features to food categories
- Duration: 3 epochs
- Learning rate: 0.001 (relatively high)

Why freeze the backbone?
The pre-trained features are already good! If we update them with random classifier weights, we might destroy valuable learned features.

PHASE 2: FINE-TUNING
- Unfreeze last 3 blocks of EfficientNet
- Train both backbone and classifier together
- Purpose: Adapt features specifically for food recognition
- Duration: 22 epochs

- Learning rate: 0.0001 (lower to preserve features)

Why partial unfreezing?
- Early layers learn generic features (keep frozen)
- Later layers learn task-specific features (fine-tune these)
- This balance preserves useful features while adapting to our task

3.3 Why Not Train From Scratch?

Training from scratch would require:
- Millions of images (we only have 113K)
- Weeks of training time (we trained in 8.5 hours)
- Much more GPU memory and compute power
- Careful architecture design

With transfer learning:
- 113K images are sufficient
- 8.5 hours of training
- 4GB GPU memory is enough
- Architecture already proven effective

RESULT: We achieved 84.8% accuracy that would be nearly impossible training from scratch with our resources.

4. Model Components Explained

4.1 What is Dropout?

Dropout is a regularization technique that prevents overfitting by randomly "dropping" (setting to zero) a percentage of neurons during training.

HOW IT WORKS:
- During training: Randomly set 25% of neurons to 0
- During inference: Use all neurons (scaled appropriately)

WHY IT HELPS:
1. Prevents co-adaptation: Neurons can't rely on specific other neurons
2. Creates ensemble effect: Like training many smaller networks
3. Forces redundancy: Multiple pathways learn similar features

OUR USAGE:
- Dropout rate: 0.25 (25% of neurons dropped)
- Applied after pooling and after hidden layer
- Lower than typical (0.5) because we have lots of data

VISUALIZATION:
Training: [N1] [0] [N3] [0] [N5] [N6] [0] [N8] (some dropped)
Inference: [N1] [N2] [N3] [N4] [N5] [N6] [N7] [N8] (all active)

4.2 What is Batch Normalization?

Batch Normalization (BatchNorm) normalizes the inputs to each layer, making training faster and more stable.

THE PROBLEM IT SOLVES:
As data flows through the network, the distribution of values can shift dramatically (internal covariate shift). This makes training unstable and slow.

HOW IT WORKS:
For each mini-batch:
1. Calculate mean and variance of activations
2. Normalize to zero mean, unit variance
3. Scale and shift with learnable parameters

FORMULA:
y = gamma * (x - mean) / sqrt(variance + epsilon) + beta

Where gamma and beta are learned parameters.

BENEFITS:
1. Allows higher learning rates (faster training)
2. Reduces sensitivity to initialization
3. Acts as regularization (slight noise from batch statistics)
4. Makes optimization landscape smoother

OUR USAGE:
Applied after the hidden layer (512 neurons), before ReLU activation.

4.3 What is ReLU Activation?

ReLU (Rectified Linear Unit) is an activation function that introduces non-linearity into the network.

THE FORMULA:
ReLU(x) = max(0, x)

Simply: If input is positive, output equals input. If negative, output is 0.

WHY NON-LINEARITY IS NEEDED:
Without activation functions, stacking layers would just be matrix multiplication - equivalent to a single linear transformation! Non-linearity allows the network to learn complex patterns.

WHY RELU SPECIFICALLY:
1. Simple and fast to compute
2. Doesn't saturate for positive values (unlike sigmoid/tanh)
3. Sparse activation (many zeros) - computationally efficient
4. Reduces vanishing gradient problem

ALTERNATIVES:
- Sigmoid: Squashes to 0-1 (used in output for probabilities)
- Tanh: Squashes to -1 to 1
- LeakyReLU: Small slope for negative values
- GELU: Smooth approximation (used in transformers)

4.4 What is Adaptive Average Pooling?

Adaptive Average Pooling reduces the spatial dimensions of feature maps to a fixed size, regardless of input size.

THE PROBLEM:
EfficientNet produces feature maps of size 7x7x1280 (for 224x224 input). We need a fixed-size vector for the classifier.

HOW IT WORKS:
- Input: 7x7x1280 feature map
- Operation: Average all values in each 7x7 spatial region
- Output: 1x1x1280 vector

WHY "ADAPTIVE"?
Regular pooling has fixed kernel size. Adaptive pooling adjusts kernel size automatically to produce desired output size. This means:
- Works with any input resolution
- Always produces same output size

OUR USAGE:
AdaptiveAvgPool2d(1) - Reduces each channel to single value by averaging.

4.5 What is a Fully Connected (Linear) Layer?

A Fully Connected layer (also called Dense or Linear layer) connects every input neuron to every output neuron.

HOW IT WORKS:
output = input * weights + bias

Where:
- input: Vector of size N
- weights: Matrix of size N x M
- bias: Vector of size M
- output: Vector of size M

OUR ARCHITECTURE:
Layer 1: 1280 -> 512 neurons
- Reduces dimensionality
- Learns combinations of features

Layer 2: 512 -> 181 neurons
- One output per food category
- Raw scores (logits) before softmax

WHY TWO LAYERS?
Single layer (1280 -> 181) would work but:
- Two layers can learn more complex mappings
- Hidden layer acts as bottleneck (compression)
- More parameters = more learning capacity

4.6 What is Softmax and Cross-Entropy Loss?

These work together for multi-class classification.

SOFTMAX:
Converts raw scores (logits) into probabilities that sum to 1.

Formula: softmax(x_i) = exp(x_i) / sum(exp(x_j))

Example:
Logits: [2.0, 1.0, 0.5] -> Softmax: [0.59, 0.24, 0.17]

The highest logit gets highest probability, but all classes get some probability.

CROSS-ENTROPY LOSS:
Measures how different predicted probabilities are from true labels.

Formula: Loss = -sum(y_true * log(y_pred))

For single correct class: Loss = -log(probability of correct class)

WHY THIS COMBINATION:
1. Softmax ensures valid probability distribution
2. Cross-entropy heavily penalizes confident wrong predictions
3. Gradient is simple: (predicted - actual)
4. Works well with one-hot encoded labels

OUR USAGE:
PyTorch's CrossEntropyLoss combines softmax and cross-entropy efficiently.

5. Training Process Explained

5.1 What is an Epoch?

An epoch is one complete pass through the entire training dataset.

OUR TRAINING:
- Total epochs: 25 (3 warmup + 22 fine-tuning)
- Training images: 113,900
- Batch size: 32
- Iterations per epoch: 113,900 / 32 = 3,559 batches

EACH EPOCH:
1. Shuffle training data (randomize order)
2. Process all 3,559 batches
3. Update weights after each batch
4. Evaluate on validation set
5. Save model if validation accuracy improved

WHY MULTIPLE EPOCHS?
- One pass isn't enough to learn all patterns
- Each epoch refines the learned features
- Model sees data in different orders (shuffling helps generalization)

5.2 What is a Batch?

A batch is a subset of training examples processed together before updating weights.

OUR BATCH SIZE: 32 images

WHY NOT ONE IMAGE AT A TIME?
1. GPU parallelism: Process 32 images simultaneously
2. Stable gradients: Average over multiple examples
3. Faster training: Fewer weight updates needed

WHY NOT ALL IMAGES AT ONCE?
1. Memory limit: 113,900 images won't fit in GPU memory
2. Generalization: Some noise in gradients helps avoid local minima
3. More frequent updates: Learn faster

THE TRADEOFF:
- Smaller batches: More noise, better generalization, slower
- Larger batches: Less noise, may overfit, faster

Batch size 32 is a good balance for our 4GB GPU.

5.3 What is the Optimizer (AdamW)?

The optimizer determines HOW to update weights based on gradients. AdamW is our chosen optimizer.

GRADIENT DESCENT (Basic):
new_weight = old_weight - learning_rate * gradient

Problem: Same learning rate for all parameters, can be slow.

ADAM (Adaptive Moment Estimation):
Improves on basic gradient descent by:
1. Momentum: Accumulates past gradients (like a rolling ball)
2. Adaptive learning rates: Different rate per parameter
3. Bias correction: Accounts for initial zero estimates

ADAMW (Adam with Weight Decay):
Adds proper weight decay (L2 regularization) to Adam.

Weight decay: Slightly shrinks weights each step, preventing them from growing too large (reduces overfitting).

OUR SETTINGS:
- Learning rate: 0.001 (warmup) / 0.0001 (fine-tuning)
- Weight decay: 0.01
- Betas: (0.9, 0.999) - momentum parameters

5.4 What is Learning Rate?

Learning rate controls how much weights change in each update. It's the most important hyperparameter.

TOO HIGH:
- Weights change too much
- Training unstable (loss jumps around)
- May never converge

TOO LOW:
- Weights change too little
- Training very slow
- May get stuck in local minima

FINDING THE RIGHT VALUE:
- Start with common values (0.001, 0.0001)
- Monitor training loss
- Adjust if training is unstable or too slow

OUR APPROACH:
- Phase 1 (warmup): 0.001 - Higher because only training classifier
- Phase 2 (fine-tuning): 0.0001 - Lower to preserve pre-trained features

5.5 What is Cosine Annealing?

Cosine Annealing is a learning rate schedule that gradually decreases the learning rate following a cosine curve.

THE IDEA:
- Start with higher learning rate (explore broadly)
- Gradually decrease (fine-tune precisely)
- Follows smooth cosine curve (not abrupt steps)

FORMULA:
lr = lr_min + 0.5 * (lr_max - lr_min) * (1 + cos(epoch/total_epochs * pi))

VISUALIZATION:
```
Epoch 1:  |******                    | High LR
Epoch 5:  |*****                     | Still high
Epoch 10: |****                      | Decreasing
Epoch 15: |***                       | Lower
Epoch 20: |**                        | Much lower
Epoch 25: |*                         | Very low (fine-tuning)
```

WHY IT WORKS:

1. Early: Large steps help escape bad local minima
2. Late: Small steps help converge to good minimum
3. Smooth transition avoids training instability

5.6 What is Backpropagation?

Backpropagation is the algorithm that calculates how much each weight contributed to the error, enabling learning.

THE PROCESS:
1. FORWARD PASS: Input flows through network to get prediction
2. CALCULATE LOSS: Compare prediction to true label
3. BACKWARD PASS: Calculate gradient of loss w.r.t. each weight
4. UPDATE WEIGHTS: Adjust weights to reduce loss

CHAIN RULE:
The magic of backpropagation is using the chain rule of calculus:

If loss depends on output, and output depends on hidden layer, and hidden layer depends on weights, then:

d(loss)/d(weight) = d(loss)/d(output) * d(output)/d(hidden) * d(hidden)/d(weight)

This lets us compute gradients for millions of weights efficiently!

AUTOMATIC DIFFERENTIATION:
PyTorch automatically tracks all operations and computes gradients. We just call loss.backward() and it handles everything!

6. Overfitting and Underfitting

6.1 What is Overfitting?

Overfitting occurs when a model learns the training data TOO WELL, including noise and irrelevant patterns, failing to generalize to new data.

SYMPTOMS:
- High training accuracy (96%+)
- Low validation accuracy (65%)
- Large gap between train and validation

ANALOGY:
Like a student who memorizes exam answers without understanding concepts. They ace practice tests but fail new questions.

WHY IT HAPPENS:
1. Model too complex for the data
2. Training data too small
3. Training too long
4. Not enough regularization

OUR V1 MODEL:
- Training accuracy: 96.2%
- Validation accuracy: 65.5%
- Gap: 32% (SEVERE OVERFITTING!)

The model memorized 3,150 training images instead of learning general food features.

6.2 What is Underfitting?

Underfitting occurs when a model is too simple to capture the underlying patterns in data.

SYMPTOMS:
- Low training accuracy
- Low validation accuracy
- Validation may be higher than training (with regularization)

ANALOGY:
Like trying to draw a curve with only a straight line. No matter how you adjust, you can't capture the shape.

WHY IT HAPPENS:
1. Model too simple
2. Too much regularization

3. Training too short
4. Learning rate too low

OUR V2 MODEL:
- Training accuracy: 34.2%
- Validation accuracy: 45.6%
- Gap: -13% (validation higher = underfitting)

We applied too much regularization (dropout 0.5, aggressive augmentation), preventing the model from learning.

6.3 The Bias-Variance Tradeoff

This fundamental concept explains the balance between underfitting and overfitting.

BIAS: Error from oversimplified assumptions
- High bias = underfitting
- Model can't capture true patterns

VARIANCE: Error from sensitivity to training data fluctuations
- High variance = overfitting
- Model captures noise, not signal

THE TRADEOFF:
Simple models: High bias, low variance (underfit)
Complex models: Low bias, high variance (overfit)
Ideal: Balance between bias and variance

OUR JOURNEY:
V1: Too complex (high variance, overfit)
V2: Too constrained (high bias, underfit)
V3: Better balance, but still limited
FINAL: Large dataset reduced variance without increasing bias!

KEY INSIGHT:
More data reduces variance without affecting bias. This is why our final model with 113K images achieved both high accuracy AND low overfitting.

6.4 Regularization Techniques We Used

Regularization adds constraints to prevent overfitting. Here's what we used:

1. DROPOUT (0.25)
What: Randomly zero out 25% of neurons during training
Why: Prevents co-adaptation, creates ensemble effect
Effect: Forces network to learn redundant representations

2. WEIGHT DECAY (0.01)
What: Add penalty for large weights to loss function
Why: Large weights often indicate overfitting
Effect: Keeps weights small, model simpler

3. DATA AUGMENTATION
What: Apply random transformations to training images
Why: Creates "new" training examples from existing ones
Transforms used:
- Random horizontal flip
- Random rotation (+/- 15 degrees)
- Color jitter (brightness, contrast)
- Random resized crop

4. EARLY STOPPING
What: Stop training when validation accuracy stops improving
Why: More training can lead to overfitting
Effect: Find the "sweet spot" before overfitting

5. BATCH NORMALIZATION
What: Normalize layer inputs
Why: Adds slight noise from batch statistics
Effect: Acts as mild regularization

6.5 Why More Data Was the Real Solution

Despite trying various regularization techniques, the real breakthrough came from more data.

WITH SMALL DATASET (3,150 images, 90 classes):
- ~35 images per class
- Model quickly memorizes all images
- Regularization can slow but not prevent overfitting
- Best result: 63.6% accuracy, +15.5% gap

WITH LARGE DATASET (113,900 images, 181 classes):
- ~629 images per class (18x more!)

- Model must learn general features to handle variety
- Regularization becomes less critical
- Result: 84.8% accuracy, +2.5% gap

THE LESSON:
"You can't regularize your way out of insufficient data."

Regularization treats symptoms. More data treats the cause. With enough diverse examples, the model naturally learns generalizable features.

7. Data Augmentation Explained

7.1 What is Data Augmentation?

Data augmentation artificially increases training data by applying random transformations to existing images.

THE IDEA:
A photo of pizza is still pizza whether:
- Flipped horizontally
- Rotated slightly
- Brighter or darker
- Cropped differently

By showing the model these variations, it learns that these transformations don't change the class.

BENEFITS:
1. More training examples without collecting new data
2. Model learns invariances (same food, different views)
3. Reduces overfitting
4. Improves generalization to real-world images

7.2 Augmentations We Applied

1. RANDOM HORIZONTAL FLIP (p=0.5)
What: Mirror image left-to-right with 50% probability
Why: Food looks same from either side
Example: [Pizza ->] becomes [<- Pizza]

2. RANDOM ROTATION (+/- 15 degrees)
What: Rotate image randomly up to 15 degrees
Why: Photos aren't always perfectly aligned
Example: Slightly tilted bowl of rice

3. COLOR JITTER
What: Randomly adjust brightness and contrast by 20%
Why: Different lighting conditions in real photos
Example: Same food under dim vs bright light

4. RANDOM RESIZED CROP (scale 0.8-1.0)
What: Crop random portion (80-100% of image), resize to 224x224
Why: Food may not be perfectly centered
Example: Focus on different parts of the dish

5. NORMALIZATION (Always applied)

What: Subtract ImageNet mean, divide by std
Why: Matches pre-training data distribution
Values: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]

NOTE: Augmentation only during TRAINING, not validation/testing!

8. Evaluation Metrics

8.1 What is Accuracy?

Accuracy is the percentage of correct predictions.

FORMULA:
Accuracy = (Correct Predictions / Total Predictions) * 100

OUR RESULTS:
- Training accuracy: 87.2%
- Validation accuracy: 84.8%

WHAT THIS MEANS:
Out of 100 food images, our model correctly identifies about 85.

LIMITATIONS OF ACCURACY:
- Doesn't show which classes are confused
- Can be misleading with class imbalance
- Doesn't indicate confidence of predictions

For our project, accuracy is appropriate because:
- Classes are relatively balanced
- We care about overall performance
- Top-5 accuracy handles uncertainty

8.2 What is Top-5 Accuracy?

Top-5 accuracy checks if the correct answer is among the model's top 5 predictions.

WHY IT MATTERS:
Some foods look very similar:
- Biryani vs Pulao
- Pizza Margherita vs Pizza with toppings
- Different types of curry

If the model's #1 prediction is wrong but #2 is correct, that's still useful!

HOW IT WORKS:
1. Model outputs probabilities for all 181 classes
2. Sort by probability, take top 5
3. Check if true label is in top 5
4. Count as correct if yes

TYPICAL RESULTS:
If top-1 accuracy is 85%, top-5 accuracy might be 95%+

OUR WEB APP:
Shows top 5 predictions so users can see alternatives if top prediction seems wrong.

8.3 Training vs Validation Accuracy

These two metrics together reveal model health.

TRAINING ACCURACY:
- Measured on images the model trained on
- Shows how well model fits training data
- Can be artificially high (memorization)

VALIDATION ACCURACY:
- Measured on images the model NEVER saw during training
- Shows how well model generalizes
- The true measure of performance

THE GAP:
Gap = Training Accuracy - Validation Accuracy

Interpretation:
- Gap ~0%: Perfect generalization (rare)
- Gap 0-10%: Healthy model
- Gap 10-20%: Some overfitting
- Gap >20%: Severe overfitting
- Gap <0%: Underfitting (model too constrained)

OUR FINAL MODEL:
- Training: 87.2%
- Validation: 84.8%
- Gap: +2.5% (EXCELLENT!)

9. Gradio Web Framework

9.1 What is Gradio?

Gradio is a Python library for creating web interfaces for machine learning models with minimal code.

WHY WE CHOSE GRADIO:
1. Minimal code: Create interface in ~10 lines
2. No web development needed: No HTML/CSS/JavaScript
3. Automatic UI: Handles file uploads, displays, etc.
4. Easy sharing: Can create public links
5. Jupyter compatible: Works in notebooks

ALTERNATIVES CONSIDERED:
- Flask/Django: More control but much more code
- Streamlit: Similar but Gradio is simpler for ML
- Command line: Not user-friendly

9.2 How Our Gradio App Works

COMPONENTS:

1. Image Input (gr.Image)
- Accepts drag-and-drop or file browser
- Handles JPEG, PNG, WebP
- Resizes for display

2. Processing Function (predict)
- Receives uploaded image
- Preprocesses (resize, normalize)
- Runs through model
- Returns predictions and recipe

3. Text Outputs (gr.Textbox)
- Top 5 predictions with confidence
- Predicted food name
- Cuisine type
- Complete recipe

THE FLOW:
User uploads image
    -> Gradio receives file
    -> Our predict() function runs
    -> Returns results

    -> Gradio displays in UI

INTERFACE CODE:

```
gr.Interface(
    fn=predict,            # Our prediction function
    inputs=gr.Image(),     # Image upload component
    outputs=[...],         # Multiple text outputs
    title="...",           # App title
    description="..."      # Instructions
).launch()                 # Start server
```

9.3 Running the Web Application

STARTING THE SERVER:

```
cd inversecooking/src
python web_app_large.py
```

OUTPUT:

```
Running on local URL:  http://127.0.0.1:7860
```

WHAT HAPPENS:
1. PyTorch loads the trained model
2. Gradio creates the web interface
3. Local web server starts on port 7860
4. Browser can connect to see the interface

USING THE APP:
1. Open http://127.0.0.1:7860 in browser
2. Upload or drag-drop a food image
3. Click Submit
4. See predictions and recipe

INFERENCE TIME:
- With GPU: <1 second
- CPU only: 2-5 seconds

10. Summary of Key Concepts

This document explained the following concepts used in our project:

DEEP LEARNING FUNDAMENTALS
- Neural networks learn patterns from data automatically
- CNNs are specialized for image processing
- Layers build hierarchical feature representations

EFFICIENTNET-B0
- State-of-the-art CNN architecture
- Compound scaling for efficiency
- Pre-trained on ImageNet (1.2M images)
- 5.3M parameters, fits in 4GB GPU

TRANSFER LEARNING
- Reuse knowledge from pre-trained model
- Two-phase training: warmup + fine-tuning
- Requires less data and trains faster

MODEL COMPONENTS
- Dropout: Randomly disable neurons (regularization)
- BatchNorm: Normalize layer inputs (stability)
- ReLU: Non-linear activation function
- Softmax: Convert logits to probabilities
- Cross-entropy: Loss function for classification

TRAINING PROCESS
- Epochs: Full passes through data
- Batches: Subsets processed together
- AdamW: Adaptive optimizer with weight decay
- Cosine annealing: Gradually decrease learning rate
- Backpropagation: Calculate gradients for learning

OVERFITTING VS UNDERFITTING
- Overfitting: Memorizes training data
- Underfitting: Can't capture patterns
- Solution: More data + balanced regularization

DATA AUGMENTATION
- Random transformations create variety
- Model learns invariances

- Reduces overfitting

EVALUATION
- Accuracy: Percentage correct
- Gap: Train - Val accuracy (should be low)
- Top-5: Correct in top 5 predictions

Every component in our system has a specific purpose, and together they achieve 84.8% accuracy on 181 food categories!