3.Interpretability of Neural Networks continued

Detection Concepts

Two Key Limitations of feature-based methods:

1.Features Not User-Friendly

1. Single pixel importance in images lacks meaningful interpretation
2. Technical features difficult for domain experts to understand

2.Constrained Expressiveness

1. Explanations limited by number of input features
2. Cannot explain using concepts outside feature space

Concept-Based Solution:

•Concepts refer to any abstraction (color, object, idea)
•Detected concepts embedded in network's latent space
•Generates explanations beyond original feature space
•Network doesn't need explicit training on the concept

Testing with Concept Activation Vectors

What TCAV Does:

- Generates global explanations for neural networks
- Measures extent of concept's influence on predictions for a class

Example question: "How does 'striped' influence classifying images as 'zebra'?"

Key Properties:

- Describes relationship between concept and class
- Provides global interpretation of model's overall behavior
- Works with any model where directional derivatives are possible

Numerical Representation of Concepts - Concept Activation Vector (CAV)

CAV = numerical representation generalizing a concept in activation space

How to Calculate CAV:

- Prepare Two Datasets:
  - Concept dataset representing C (e.g., striped object images)
  - Random dataset with arbitrary data (e.g., random images without stripes)
- Target Hidden Layer l
- Train Binary Classifier:
  - Separates activations from concept set vs. random set
  - Use SVM or logistic regression
- Extract CAV:
  - Coefficient vector of trained classifier = v_l^C

Measuring Concept Influence

$$S_{C,k,l}(\mathbf{x}) = \nabla h_{l,k}(\hat{f}_l(\mathbf{x})) \cdot \mathbf{v}_l^C$$

Where:
- f̂l: maps input x to activation vector at layer l
- h{l,k}: maps activation vector to logit output of class k
- ∇h{l,k}: gradient pointing to direction maximizing output

Interpretation Based on Sign:
- Positive (angle < 90°): Concept C encourages classifying x into class k
- Negative (angle > 90°): Concept C discourages classification

TCAV Score

$$TCAV_{Q,C,k,l} = \frac{|\{\mathbf{x} \in \mathbf{X}_k : S_{C,k,l}(\mathbf{x}) > 0\}|}{|\mathbf{X}_k|}$$

- Ratio of inputs with positive conceptual sensitivity to total inputs for class k

Example:
- TCAV = 0.8 for concept "striped" and class "zebra"
- Interpretation: 80% of zebra predictions positively influenced by "striped"

Ensuring Meaningful TCAV Scores

The Problem:

- CAV trained on user-selected datasets
- Bad datasets lead to misleading explanations

Solution - Test Procedure:

1.Collect N random datasets (recommended N ≥ 10)
2.Keep concept dataset fixed
3.Calculate TCAV score using each random dataset
4.Apply two-sided t-test:
    1. Compare N TCAV scores from real concept
    2. Against N TCAV scores from random CAV

Key Insight:

•Meaningful concepts generate consistent TCAV scores across different random datasets

Why Use TCAV?

Strength 1: No ML Expertise Required

•Users only collect concept data

•Extremely useful for domain experts evaluating models

Strength 2: Customizability

•Investigate any concept definable by dataset

•Control balance between complexity and interpretability

•Domain experts can shape concept datasets for fine-grained explanations

Strength 3: Global Explanations

•Reveals overall model behavior

•Identifies "flaws" or "blind spots" in training

•Enables model improvement

When TCAV Struggles

Limitation 1: Poor Performance on Shallow Networks

•Concepts in deeper layers more separable

•Shallow networks may not separate concepts clearly

Limitation 2: Requires Additional Annotations

•Expensive for tasks without readily labeled data

•Alternative: ACE (automated concept generation)

Limitation 3: Difficult for Abstract Concepts

•Abstract concepts (e.g., "happiness") hard to capture in datasets

•More abstract = more data required

Limitation 4: Limited Application to Text/Tabular Data

•Gains popularity in image data

•Relatively limited for text and tabular data

Adversarial Examples

- Instance with small, intentional feature perturbations causing false predictions
- Similar to counterfactual explanations but aims to deceive, not interpret

Examples

Self-Driving Car Attack:

- Picture placed over stop sign looks normal to humans
- Designed to look like parking prohibition to car's recognition software
- Result: Car crashes by ignoring stop sign

Spam Detector Evasion:

- Email designed to resemble normal message
- Intention: Cheat recipient while evading spam filter

Airport Security Bypass:

- Knife designed to avoid ML-powered scanner detection
- System thinks knife is umbrella

Creating adversarial examples

Gradient-Based Methods:

- Require access to model gradients
- Only work with gradient-based models (e.g., neural networks)

Model-Agnostic Methods:

- Only require access to prediction function
- Work with any black-box model

Method 1: Gradient based

Goal: Find a small perturbation to an input (like an image) that causes the model to misclassify, while keeping the perturbation as small as possible.

Optimization Objective: Minimize the following function with respect to the perturbation vector $\mathbf{r}$:

$$\text{loss}(\hat{f}(\mathbf{x}+\mathbf{r}), l) + c \cdot |\mathbf{r}|$$

Where:

- x: original image (or input, as a vector of features/pixels)
- r: perturbation (what you add to the image to create an adversarial example)
- f: the model
- l: the desired target class (the label you want the model to misclassify as)
- c: balance between making **r** small and making the prediction change
- |r|: norm or magnitude of the perturbation

Working

- The first term forces the model's prediction for the perturbed image to be the chosen (wrong) class.
- The second term penalizes large changes—so $\mathbf{r}$ stays as small as possible.
- The optimization is subject to constraints, so pixel values stay valid (e.g., between 0 and 1).
- Typically solved with a gradient-based optimizer like L-BFGS.

Method 2- Fast Gradient Sign Method

Disturbed Panda (Goodfellow et al., 2015)

Goal: Quickly generate an adversarial example by making a single, calculated step in the direction that increases the model's error the most.

$$\mathbf{x}' = \mathbf{x} + \epsilon \cdot \text{sign}\left(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y)\right)$$

Components:

•∇_x J: gradient of model's loss function w.r.t. input pixels
•y: true label for x
•θ: model parameter vector
•ε: small error added/subtracted to each pixel

How It Works:

•Uses gradient sign to determine direction of perturbation
•Sign = +1 if increasing pixel intensity increases loss
•Sign = -1 if decreasing pixel intensity increases loss
•Adds/subtracts ε based on gradient sign

Working:

- Compute the gradient of the loss with respect to the input, which tells you which way to change each pixel to most increase the loss (i.e., make the model more likely to get the answer wrong).
- Take an “epsilon-sized” step in the direction of the sign of this gradient for each pixel.
- Produces a new input $\mathbf{x}'$ that is very close to the original, but more likely to be misclassified.

Method 3: 1-Pixel Attack (Su et al., 2019)

Goal: Find if changing just a single pixel is enough to fool a neural network into misclassifying an image.

How it works:

- The attacker searches for the best single pixel to change (including its RGB values and position) so that the image is classified incorrectly.
- Uses differential evolution (a population-based optimization algorithm):
  - Maintains a set of candidate solutions, each encoding a possible pixel change: (x, y, R, G, B).
  - Combines, mutates, and selects pixel changes over generations, similar to evolutionary algorithms.
  - Child solutions are generated as:

$$\mathbf{x}_{g+1}^{(i)} = \mathbf{x}_g^{(r1)} + F \cdot (\mathbf{x}_g^{(r2)} - \mathbf{x}_g^{(r3)})$$

Where:
g is the generation, F is a scaling coefficient (e.g., 0.5), $r1, r2, r3$ are random parents.
Stops if an adversarial image is found or a set number of generations have passed.

Result: Even a single-pixel change, carefully chosen, can cause misclassification.

Method 4: Adversarial Patch (Brown et al., 2018)

Goal: Create a small patch (sticker) that, when placed anywhere in an image, fools the neural network into misclassifying objects as a specific target class (e.g., “toaster”).

Working:

- Unlike previous methods, there’s no requirement that the patch must closely resemble the original image area—it can be any shape, color, or pattern.
- Simulate overlaying the patch at random locations, scales, and rotations on many background images.
- Update the patch so that, across all these variations, the classifier predicts the attacker’s chosen target class.
- After training, the resulting patch image can be printed and physically placed in scenes to fool ML models in the real world.

Result: The model is fooled regardless of the rest of the scene, as long as the patch is present.

Method 5: Robust Adversarial Examples in 3D (Athalye et al., 2018)

Goal: Create physical (3D-printed) objects that are adversarial across a wide range of viewing angles and lighting conditions.

Working:

- Use the Expectation Over Transformation (EOT) algorithm to ensure the adversarial property is maintained under real-world transformations (rotations, scaling, lighting).
- Optimization objective:

$$\arg\max_{\mathbf{x}'} \mathbb{E}_{t\sim T}[\log \mathbb{P}(y_t|t(\mathbf{x}'))]$$

  - Where T is the set of possible image transformations, t samples a transformation, and y_t is the target class.

•**Constraint**: Average (expected) distance between the transformed adversarial and original object must stay below a threshold:

$$\mathbb{E}_{t\sim T}[d(t(\mathbf{x}'), t(\mathbf{x}))] < \epsilon \quad \text{and} \quad \mathbf{x} \in [0,1]^d$$

Resulting adversarial object looks like the original to humans but is classified as something else by the network under various transformations (e.g., a turtle consistently misclassified as a rifle).

Method 6: Black Box Attack (Papernot et al., 2017)

Goal: Fool a remotely-hosted model (e.g., via cloud API) without access to its internals—only able to see predictions.
Working:

- Gather input examples from the target domain (e.g., upload digit images to a digit classifier API).
- Use the observable predictions to train a surrogate (local) model that approximates the target model's decision boundaries.
- Iteratively, use the surrogate model:
  - Generate synthetic images in directions that maximize output variance.
  - Query the black box for its predictions on these synthetic examples.
  - Refine the surrogate model with these new examples and labels.
- Once the surrogate model is trained, use it to generate adversarial examples using methods like FGSM.
- Test these adversarial examples on the original black-box model.

Key Property: Successful attacks even work if the black box is not a neural network (can be a decision tree).
Result: Adversarial examples can be generated without access to model gradients or architecture.

What Are Influential Instances?

Core Concept:

•ML models are products of training data

•Deleting a training instance can affect the resulting model

Definition:

•Influential instance: Training instance whose deletion considerably changes model parameters or predictions

Purpose:

•"Debug" machine learning models

•Better explain model behaviors and predictions

•Trace model back to responsible training data

Two Approaches:

1.Deletion diagnostics: Remove instance, retrain, measure change

2.Influence functions: Upweight instance by infinitesimal amount, approximate change via derivatives

Foundation:

•Both based on robust statistics

•Robust statistics = statistical methods less affected by outliers or assumption violations

Outlier vs. Influential Instance

Outlier:

•Instance far away from other instances in dataset

•"Far away" = large distance (e.g., Euclidean) to all others

Examples:

- Newborn weighing 5 kg in dataset of newborns
- Loan account (large negative balance, few transactions) in dataset of checking accounts

•Can be interesting data points (e.g., edge cases)

Influential Instance:

•Instance whose removal strongly affects trained model

•More parameters/predictions change when retrained without it = more influential

•Depends on both features X and target y

•**Example**: Instance in linear regression that changes slope significantly

Relationship:

•Outliers can be influential instances

•But not all outliers are influential

•Not all influential instances are outliers

Tracing Back to Training Data

Key Idea:

•Trace model parameters and predictions back to where it began: training data

•Learner = algorithm generating ML model from features X and target y

Contrast with Other Methods:

•Other interpretability methods: Analyze how prediction changes when manipulating features (PDP, feature importance)

•Influential instances: Analyze how model changes when manipulating training data

•Model treated as function of training data, not as fixed

Questions Answered:

Global Questions:

•Which instances most influential for model parameters/predictions overall?

•Which training instances should be checked for errors?

Local Questions:

•Which instances most influential for prediction?

•Why did model make this specific prediction?

Insights Provided:

•Instances where model might have problems

•Impression of model robustness

•If single instance has strong influence → investigate further, may not trust model

Measuring Influence by Retraining

Approach:

- Delete instance from training data
- Retrain model on reduced dataset
- Observe difference in model parameters or predictions

Two Classic Measures (from Statistics):

DFBETA (Parameter Change):

- Measures effect of deleting instance on model parameters
- Formula: $DFBETA_i = \boldsymbol{\beta} - \boldsymbol{\beta}^{(-i)}$
- Where:
  - β = weight vector trained on all data
  - β^(-i) = weight vector trained without instance i
- **Limitation**: Only works for parameterized models (logistic regression, neural networks)

Cook's Distance (Prediction Change):

- Measures effect of deleting instance on predictions
- Originally for linear regression, approximations exist for generalized linear models

Cook's Distance

Formula:

Where:

- Numerator: Sum of squared differences between predictions with/without instance i
- p: Number of features
- MSE: Mean squared error
- Denominator same for all instances

$$D_i = \frac{\sum_{j=1}^{n} (\hat{y}_j - \hat{y}_j^{(-i)})^2}{p \cdot MSE}$$

Interpretation:

- Measures how much predicted output changes when removing i-th instance

Limitations:

- DFBETA requires model parameters (excludes decision trees, tree ensembles)
- Cook's distance designed for linear models
- MSE not meaningful for all prediction types (e.g., classification)

The Core Problem with deletion and its solution

The Problem with Deletion:

•Deletion diagnostics require full model retraining
•1,000 instances = 1,000 retrainings = too expensive

The Influence Function Solution:

•Instead of deleting instance, upweight it by tiny amount (ε)
•Approximate how model changes without actually retraining
•Uses calculus (derivatives) to estimate the effect

Key Requirements:

•Model must have loss function
•Loss must be twice differentiable with respect to parameters
•Works for: Neural networks, logistic regression, SVMs
•Doesn't work for: Decision trees, random forests

Main Idea:

•If we increase the importance of training instance z by a tiny bit, how do the model parameters change?
•Calculate this using gradients (first derivative) and curvature (second derivative)

The Upweighting Concept

Original Training:

•Train model by minimizing average loss over all n training instances

•Each instance has equal weight: 1/n

Upweighting Instance z:

•Give instance z slightly more weight: 1/n + ε (where ε is tiny)

•Other instances get slightly less weight accordingly

•Formula:

$$\hat{\boldsymbol{\theta}}_{\epsilon,\mathbf{z}} = \arg\min_{\boldsymbol{\theta}\in\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} L(\mathbf{z}^{(i)}, \boldsymbol{\theta}) + \epsilon L(\mathbf{z}, \boldsymbol{\theta}) \right)$$

Intuition:

•This is mathematically similar to removing the instance

•Upweighting tells us: "If this instance matters more, how does model change?"

•By making ε infinitesimally small, we can use calculus

Why Not Actually Upweight?

•We don't need to retrain

•We can calculate the effect using derivatives

Gradient and Hessian

Gradient (First Derivative)

$$\nabla_{\boldsymbol{\theta}} L(\mathbf{z}, \hat{\boldsymbol{\theta}})$$

Measures: Direction of steepest increase in loss
Tells us: How to change parameters to increase/decrease loss
Example: If gradient is [+2, -3], increasing parameter 1 increases loss; increasing parameter 2 decreases loss

Hessian Matrix (Second Derivative)
Symbol: Hθ
Measures: Curvature of the loss function
Tells us: How fast the gradient changes
Formula:

$$H_{\boldsymbol{\theta}} = \frac{1}{n} \sum_{i=1}^{n} \nabla^2_{\hat{\boldsymbol{\theta}}} L(\mathbf{z}^{(i)}, \hat{\boldsymbol{\theta}})$$

Matrix as curvature depends on direction

Part 1: ∇θ L(z, θ) (Gradient for instance z)

•How much does loss for instance z change when we change parameters?

•Points in direction that increases loss for z

Part 2: H^(-1) (Inverse Hessian)

•Adjusts for curvature of the loss landscape

•Tells us: "How big should our step be given the curvature?“

Part 3: Negative sign (–)

•We want to decrease loss for z (when upweighting it)

•So we move in opposite direction of gradient

This Gives Us:

•Approximate change in parameters if we upweighted z

•Without actually retraining the model

$$I_{\text{up,params}}(\mathbf{z}) = \left.\frac{d\hat{\boldsymbol{\theta}}_{\epsilon,\mathbf{z}}}{d\epsilon}\right|_{\epsilon=0} = -H_{\hat{\boldsymbol{\theta}}}^{-1}\nabla_{\boldsymbol{\theta}} L(\mathbf{z}, \hat{\boldsymbol{\theta}})$$

Understanding the Approximation

The pipeline:

1.Current model has parameters θ
2.Want to know: What if we gave instance z more importance?
3.Don't know exact answer (would need retraining)
4.Approximate locally using gradient and curvature

$$\hat{\boldsymbol{\theta}}_{-\mathbf{z}} \approx \hat{\boldsymbol{\theta}} - \frac{1}{n} I_{\text{up,params}}(\mathbf{z})$$

•New parameters ≈ Old parameters − (influence/n)
•Move in direction that reduces loss for z
•Step size determined by gradient and curvature
•Scaled by 1/n (weight of one instance)

Key Insight:

•We create a quadratic approximation (parabola) around current parameters
•Use this approximation to estimate effect
•Like approximating a curve with a simple parabola at one point

Applications of Influence Functions

Application 1: Understanding Model Behavior

- Different models make predictions differently
- Same performance doesn't mean same behavior
- Influential instances reveal particular weaknesses
- Helps form "mental model" of ML model behavior

Application 2: Handling Domain Mismatch / Debugging

- Domain mismatch: Training and test distributions differ
- Example: Surgery outcome model from Hospital A fails in Hospital B
- Find influential instances causing errors
- Reveals which features/instances "broke" the model
- Example finding: Hospital B has older patients, model lacked older training data

Application 3: Fixing Training Data

- Limited resources to check data correctness
- Best strategy: Check most influential instances first
- Even obviously incorrect values less important if not influential
- Example: Hospital patient records with potential errors
- Checking data expensive (1+ hour per patient)
- Select patients with high influence on model