O'REILLY®

Explainable AI for Practitioners

Designing and Implementing Explainable ML Solutions

Michael Munn &
David Pitman

Explainable AI for Practitioners

Designing and Implementing Explainable ML Solutions

Michael Munn and David Pitman

Explainable AI for Practitioners

by Michael Munn and David Pitman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rebecca Novack

Development Editor: Rita Fernando

Production Editor: Jonathon Owen

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator:

December 2022: First Edition

Revision History for the Early Release

- 2022-07-20: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098119133 for release details.

thereof complies with such licenses and/or rights.

978-1-098-11913-3

[LSI]

Chapter 1. An Overview of Explainability

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Explainability has been a part of machine learning since the inception of AI. The very first AIs, rule-based chain systems, were specifically constructed to provide a clear understanding of what led to a prediction. The field continued to pursue explainability as a key part of models, partly due to a focus on general AI but also to justify that the research was sane and on the right track, for many decades until the complexity of model architectures outpaced our ability to explain what was happening. After the invention of ML neurons and neural nets in the 1980s,[1] research into explainability waned as researchers focused on surviving the first AI winter by turning to techniques that were "explainable" because they relied solely on statistical techniques that were well-proven in other fields. Explainability in its modern form (and what we largely focus on in this book) was revived, now as a distinct field of research, in the mid 2010s in response to the persistent question of "this model works really well… but how?"

In just a few years, the field has gone from obscurity to one of intense interest and investigation. Remarkably, many powerful explainability techniques have been invented, or repurposed from other fields, in the short time since. However, the rapid transition from theory to practice, and the increasing need for explainability from users who interact with ML, such as users and business stakeholders, has led to growing confusion about the capability and extent of different methods. Many fundamental terms of explainability are routinely used to represent different, even contradictory, ideas, and it is easy for explanations to be misunderstood due to practitioners rushing to provide assurance that ML is working as expected. Even the terms explainability and interpretability are routinely swapped, despite having very different focuses. For example, while writing this book, we were asked by a knowledgeable industry organization to describe explainable and interpretable capabilities of a system, but the definitions of explainability and interpretability were flipped in comparison to how the rest of industry defines the terms! Recognizing the confusion over explainability, the purpose of this chapter is to provide a background and common language for future chapters.

What Are Explanations?

When a model makes a prediction, Explainable AI methods generate an explanation that gives insight into the model's behavior as it arrived at that prediction. When we seek explanations, we are trying to understand "*why* did X happen?" Figuring out this "Why" can help us build a better comprehension of what influences a model, how that influence occurs, and where the model performs (or fails). As part of building our own mental models, we often find a pure explanation to be unsatisfactory, so we are also interested in explanations which provide a *counterfactual*, or *foil*, to the original situation. Counterfactuals are scenarios which seek to provide an opposing, plausible, scenario of why X did not happen. If we are seeking to explain "why did it rain today?" we may also try to find the counterfactual explanation for "why did it not rain today [in a hypothetical world]?" While our primary explanation for why it rained might include temperature, barometric pressure, and humidity, it may be easier to explain that it did not rain because there were no clouds in the sky, implying that clouds are part of an explanation for why it does rain.

We also often seek explanations that are causal, or in the form of "X was predicted because of Y." These explanations are attractive because they give an immediate sense of what a counterfactual prediction would be: remove X and presumably the prediction will no longer be Y. It certainly sounds more definitive to say "it rains because there are clouds in the sky." However, this is not always true; rain can occur even with clear skies in some circumstances. Establishing causality with data-focused explanations is extremely difficult (even for time-series data), and no explainability techniques have been proposed that are both useful in practice and have a high level of guarantee in their analysis. Instead, if you want to establish causal relationships within your model or data, we recommend you explore the field of interpretable, causal models.

Explainability Consumers

Understanding and using the results of Explainable AI can look very different depending on who is receiving the explanation. As a practitioner, for example, your needs from an explanation are very different from those of a non-technical individual who may be receiving an explanation as part of an ML system in production that they may not even know exists!

Understanding the primary types of users, or personas, will be helpful as you learn about different techniques so you can assess which will best suit your audience's needs. In Chapter 7, we will go into more detail about how to build good experiences for these different audiences with explainability.

More broadly, we can think of anyone as a consumer of an explanation. The ML system is presenting additional information to help a human perceive what is unique about the circumstances of a prediction, comprehend how the ML system behaves, and, ultimately, be able to extrapolate to what could influence a future prediction.

Currently, a key limitation of Explainable AI is that most techniques are a one-way description; the ML system communicates an explanation to the human, with no ability to respond to any follow-on requests by the user. Put another way, many Explainable AI methods are talking *at* users rather than *conversing* with users. However, while these techniques are explaining an ML system, none would be considered to

be machine learning algorithms. Although very sophisticated, these techniques are all “dumb” in the sense that we cannot interact with them in a two-way dialogue. A smart explainability technique could adapt to our queries, learning how to guide us towards the best explanation, or answer the question we didn’t know we were asking. For now, if we want to try and obtain more information about a prediction, the best we can do is to change the parameters of our explanation request, or try a different explainability technique. In Chapter 8, we outline how this will change in the future, but in the meantime, most explainability techniques represent a process closer to submitting a requisition form to an opaque bureaucracy to get information than going to your doctor and engaging in a conversation to understand why you have a headache. In our work with Explainable AI, we have found that it is useful to group explainability into three broad groups based on their needs: practitioners, observers, and end-users.

Practitioners: Data Scientists and ML Engineers

ML *practitioners* primarily use explainability as they are building and tuning a model. Their primary goal is to map an explanation to actionable steps which can be taken to improve the model’s performance, such as changing the model architecture, training data, or even the structure of the dataset itself before the model is deployed. The goal of this process is often to improve the training loss and validation set performance, but there are times where accuracy may be not the primary concern. For example, a data scientist may be concerned that the model has become influenced by data artifacts not present in the real-world data (for example, a doctor’s pen marks on an X-ray for a diagnosis model) or that the model is generating outcomes that are unfairly biased towards certain individuals.

However, a ML engineer may be asking “how can we improve the performance of our data pipeline?” Certain features may be costly to obtain, or computationally expensive to transform into a usable form. If we find that the model rarely uses those features in practice, then it is an easy decision to remove them to improve the system overall. Practitioners may also be interested in explainability once the system is deployed, but their interest still remains in understanding the underlying mechanisms and performance. Explainability has proven to be a robust and powerful tool for monitoring deployed models for drift and skew in their predictions, indicating when a model should be retrained.

And of course, you may simply be interested in an explanation because, like any practitioner, you’ve encountered a situation when the model made you squint and say “what the...?”

Observers: Business Stakeholders & Regulators

Another group of explainability consumers is *observers*. These are individuals, committees, or organizations, who are not involved in the research, design, and engineering of the model, but also are not using the model in deployment. They often fall into two two categories: stakeholders, who are within the organization building the model, and regulators, who are outside the organization.

Stakeholders often prefer a non-technical explanation, instead seeking information that will allow them to build trust that the model is behaving as is expected, and in the situation it was designed for. Stakeholders often come to an explanation with a broader, business-focused question they are trying to answer- “do we need to invest in more training data?” or “how can I trust that this new model has

learned to focus on the right things so we're not surprised later?"

Regulators are often from a public organization or industry body, but they may also come from another part of a company, (i.e., Model Risk Management) or an auditor from another company, such as an insurance company. Regulators seek to validate and verify that a model adheres to a specific set of criteria, and will continue to do so in the future. Unlike stakeholders, a regulator's explainability needs can range from quite technical to vague depending on the regulation. A common example of this conundrum is in the needs of many regulators to assess evidence that a model is not biased towards a specific category of individuals (e.g., race, gender, socioeconomic status) while also determining that model behaves fairly in practice, with no further definition given for what entails fairness. Since regulators may routinely audit a model, explanations that require less human effort to produce or understand, and can be generated efficiently and reliably, are often more useful.

End-Users: Domain Experts & Affected Users

Individuals or groups who use, or are impacted by, a model's predictions are known as end-users. *Domain experts* have a sophisticated understanding of the environment the model is operating in, but may have little to no expertise in machine learning, or even the features used by the model if they are derived, or new to the profession. Domain experts often use explanations as part of a decision support tool. For example, if an image model predicts manufacturing defects in parts on an assembly line, a quality control inspector may use an explanation highlighting where the model found the defect in the machined part to help make a decision about which part of the manufacturing process is broken.

Affected users often have little-to-no understanding of how the model works, data it uses, or what that data represents. We refer to these users as affected because they may not directly use the model, but the prediction results in a tangible impact on them. Examples of affected individuals include people receiving credit offers, where a model has predicted their ability to repay loans, or a community receiving increased funding for road maintenance. In either case, the affected users primarily want to understand and assess if the prediction was fair and that it was based on correct information. As a follow up, these users seek explanations which can give them the ability to understand how they could alter factors within their control to meaningfully change a future prediction. You may be unhappy that you were given a loan with a very, very high interest rate, but understand that it is fair because you have a poor history of repaying loans on time. After understanding the current situation, you might reasonably ask "how long of a history of on-time loan payments would I need to establish in order to get a lower interest rate?"

Types of Explanations

Modern day machine learning solutions are often complex systems incorporating many components from data processing and feature engineering pipelines, to model training and development to model serving, monitoring and updating. As a result, there are many factors that contribute to explaining why and how a machine learning system makes the predictions it does and explainability methods can be applied at each step of the ML development pipeline. In addition, the format of an explanation can depend on the modality of the model (e.g., whether it is a tabular, image, or text model). Explanations

can also vary from being very precise and specific by being generated for a single prediction or based upon a set of predictions to give a broader insight into the model's overall behavior. In the following sections we'll give a high level description of the various types of explanations that can be used to better understand how ML solutions work.

Pre-modeling Explainability

Machine learning models rely on data and although many Explainable AI techniques rely on interacting with a model, the insights they create are often focused on the dataset and features. Thus, one of the most critical stages of developing model explanations begins before any modeling takes place and is purely data focused. Pre-modeling explainability is focused on understanding the data or any feature engineering that is used to train the machine learning model.

As an example, consider a machine learning model that takes current and past atmospheric information (like humidity, temperature, cloud cover, etc) and predicts the likelihood of rain. An example of an explainable prediction that is model dependent would be "The model predicted a 90% chance of rain because, among the data inputs, humidity is 80% and cloud cover is 100%". This type of explanation relies on feature attribution for that model prediction. On the other hand, pre-modeling explanations focus only on the properties of a dataset and are independent of any model, such as "The standard deviation of the chance of rain is +/- 18%, with an average of 38%." Inherently explainable models, such as linear and statistical models, may blur this distinction, with explanations such as "For each 10% increase in the cloud cover, there is a 5% increase in the chance of rain." Given the extensive availability of resources available on more "classical" statistical and linear modeling, we will only briefly discuss here commonly used pre-modeling explainability techniques.

Explanations that focus solely on the dataset are often referred to as Exploratory Data Analysis (EDA). EDA is a collection of statistical techniques and visualizations that are used to gain more insight into a dataset. There are many techniques for summarizing and visualizing datasets and there are quite a few useful tools that are commonly used such as Know Your Data, Pandas Profiling, and Facets. These tools allow you to quickly get a sense of the statistical properties of the features in your dataset such as the mean, standard deviation, range, and percentage of missing samples as well as the feature dimensionality and presence of any outliers. From the perspective of explainability, this knowledge of the data distribution and data quality is important for understanding model behavior, interpreting model predictions, and exposing any biases that might exist.

In addition to these summary univariate statistics, explanations in the form of EDA can also take the form of multivariate statistics that describe the relationship between two or more variables or features in your dataset. Multivariate analysis can also be used to compute statistics to show the interaction between features and the target. This type of correlation analysis is useful not just for helping to explain model behavior but can also be beneficial for improving model performance. If two features are highly correlated, this could indicate an opportunity to simplify the feature space which can improve interpretability of the machine learning model. Also, knowledge of these interdependencies is important when analyzing your model using other explainability techniques; e.g. see in Chapter 3 where we discuss the effect highly correlated features has on interpreting the results of techniques like partial dependence plots or other techniques for tabular datasets. There are a number of visualization tools that

can assist in this type of correlation analysis such as pair plots, heatmaps, biplots, projection plots (T-SNE, MDS, etc.), and parallel coordinates plots.

Intrinsic vs. Post-Hoc Explainability

When and how do we receive an explanation for a prediction? Explanations that are part of the model's prediction itself are known as *intrinsic* explanations, while explanations that are performed after the model has been trained and rely on the prediction to create the explanation are called *post-hoc* explanations. Most of the techniques we discuss in this book are post-hoc explanations because they are more portable and can be decoupled from the model itself. By contrast, generating intrinsic explanations often requires modifications to the model itself or using an interpretable model. However, you may notice that some libraries are set up to provide an explanation with the prediction- this does not necessarily mean the explanation is intrinsic, as it may be that the service first generates the prediction, then the explanation, before returning both together.

Within the group of post-hoc explainability techniques, another factor to group techniques is whether the method is *model-agnostic* or *model-specific*. A model-agnostic technique does not rely on the model's architecture, or some inherent property of the model, so it can be used universally across many different types of models, datasets, and scenarios. As you might imagine, it is also more useful to become more familiar with these techniques because you will have more opportunities to reuse them than a technique which only works on a specific type of model architecture.

How can a technique not know anything about the model, and yet still generate useful explanations? Most of these techniques rely on changing inputs to the model, correlating similar predictions, or running multiple simulations of a model. By comparison, a technique that relies on the model itself will leverage some internal aspect of the model's architecture to aid in generating the explanation. For example, an explanation for tree-based models may look at the weights of specific nodes within the tree to extract the most influential decision points within the tree to explain a prediction.

Does this mean that *opaque,* or black-box, models[2] always use model-agnostic techniques, while explainability for transparent, or interpretable, models is always model-specific? Not necessarily, there are many unique explanation techniques for Deep Neural Networks (DNNs) which are considered opaque models, and a linear regression model could be explained using Shapley values, a model-agnostic technique.

Local, Cohort, and Global Explanations

Explanations themselves can cover a wide variety of topics, but one of the most fundamental is whether the explanation is *local, cohort,* or *global* with respect to the range of predictions the explanation covers. A local explanation seeks to provide context and understanding for a single prediction. A global explanation provides information about the overall behavior of the model across all predictions. Cohort explanations lie in between, providing an understanding for a subset of predictions made by the model.

Local explanations may be similar for comparable inferences, but this is not an absolute and depends on the technique, model, and dataset. Put another way, it is rarely safe to assume that an explanation for one set of inputs and inference is blindly applicable to a similar set of inputs and/or prediction.

Consider, for example, a decision tree model that predicts expected rainfall with a decision node that strictly checks whether the humidity is greater than 80%. Such a model can lead to very different predictions and explanations for two days when the humidity is 80% vs. 81%.

Global explanations can come in many forms, and because they are not directly representative of any individual prediction, they likely represent more of a survey, or summary statistics, about the model's behavior. How global explanations are generated is highly dependent on the technique, but most approaches rely either on generating explanations for all predictions (usually in the training or validation dataset) and then aggregating these explanations together, or on perturbing the inputs or weights of the model across a wide range of values.

Global explanations are typically useful for business stakeholders, regulators, or for serving as a guide to compare the deployed model's behavior against its original performance.

> **WARNING**
>
> Global explanations are not a set of rules describing the boundaries of a model's behavior, but instead only what has been observed based in the original training data. Once a model is deployed and is exposed to novel data not seen during training, it is possible that during inference local explanations can differ from or even directly contradict global explanations.

Explainable methods for cohorts are usually the same techniques as what is used to calculate global explanations, just applied to a smaller set of data. One powerful aspect of cohort explanations is that one cohort can be compared against another cohort to provide insights about the global behavior of the model that may not be apparent if we just sought a global explanation. These comparisons are useful enough in their own right to serve as the underpinnings for another pillar of Responsible AI: testing and evaluating the Fairness of a model. Fairness techniques seek to evaluate how a model performs for one cohort of predictions compared to another, with the goal of ensuring that the model generates similar predictions based on relevant factors rather than discriminating on characteristics that are deemed to be unimportant, or may not even be desirable to use in the first place. A classic example is how an AI that seeks to determine whether an individual should be approved for a loan may be trained on historical data in the US, which contains prevalent discrimination against people of different races. Scrutinizing the ML for Fairness would tell us whether the model learned this racial discrimination, or whether it is basing its decisions solely on the relevant financial background of the individual such as their income and history of on-time loan payments. Although Explainability and Fairness share many of the same underpinnings, how to apply and understand Fairness, as well as the techniques themselves, are sufficiently distinct from Explainability that we do not cover them in this book.

Attributions, Counterfactual, and Example-based

Explanations can come in many forms depending on the type of information they use to convey an understanding of the model. When asked, many of us think of *attribution*-based explanations that are focused on highlighting relevant properties of the system, e.g., "It is raining because there are clouds in the sky." In this case, the clouds are an attribute of the sky, a feature in our weather model. Proponents and opponents are part of *counter-factual* explanations, which humans often find more satisfying than

pure attribution-based explanations. Earlier when we said it was raining because of clouds, we could have also explained this by providing a counter-factual explanation: “It does not rain when it is sunny.” In this type of explanation, “clouds” would be the *proponent* of our counterfactual explanation, while sunny is the *opponent* to our explanation. Counterfactuals are often portrayed as negations (e.g., “it does **not** rain when it is sunny”) but this is not a requirement of counterfactual explanations. We could have just as easily found a counterfactual to a weather prediction for cold weather with the counterfactual “It is hot when it is sunny.” Finding, and understanding the causes behind proponents and opponents can be difficult depending on the modality of the dataset. An opponent value that is negative in a structured dataset is much easier to comprehend than why a texture in an image is considered an opponent by the model.

In Explainable AI, *example-based*, or *similarity*, explanations are those which focus on providing a different scenario that is analogous to the prediction being explained. For example, in our earlier explanation, we could have also said “It is raining because the weather is mostly like the conditions when it rains in Rome.” To ease the burden of understanding why similar predictions are relevant, example-based explanation techniques typically include secondary information to help highlight the similarities and differences in the dataset and/or how the model acted between the two predictions.

Themes Throughout Explainability

Explainability is a broad and multi-disciplinary area of machine learning that brings together ideas from various fields from game theory to social sciences. As a result there is a large and continually growing number of explainability methods and techniques that have been introduced. In this section, we’ll give an overview of some common themes that have been introduced and further developed the field.

Feature Attributions

Feature attribution methods are common throughout Explainable AI. What does it mean to attribute a prediction to an individual feature in your dataset? Formally, a feature attribution represents the influence of that feature (and its value for a local explanation) on the prediction. Feature attributions can be absolute, for example if a predicted temperature is 24° C, a feature could be attributed 8° C of that predicted value, or even a negative value like -12° C. Feature attributions can also be relative, representing a percentage of influence compared to other features used by the model.

> **NOTE**
>
> In this book, we often describe features as influencing a model, while the specific amount of influence is the attribution. In practice, “feature influence” and “feature attribution” are often used interchangeably.

If the idea of a feature having influence that is relative seems strange to you, then you’re in good company. Understanding what feature attributions convey as explanations, and what they don’t, is rife with confusion and it is often difficult for end-users to build an understanding of feature attributions’ true mechanism. For a more intuitive feel of how feature attributions work, let’s walk through an

imaginary scenario of an orchestra playing music. Within this orchestra, there are a variety of musicians (which we will treat as features) that contribute to the overall performance of how well the orchestra plays a musical composition. For Explainable AI, let's imagine that the orchestra is going to participate in ExplainableVision, a competition where a judge (the ML model) tries to predict how well an orchestra performs music by listening to the music played by each individual musician in the orchestra. In our analogy, we can use feature attributions to understand how each musician influenced the overall rating of the orchestra given by the judge.

Each feature attribution represents how a musician sways the judge towards the most accurate prediction of the musical talent of the orchestra. Some musicians may be useful to the judge in determining an accurate score of the overall orchestra's performance, so we would assign them a high value for their feature attribution. Perhaps it is because they are close to the average talent of the orchestra, or the amount of time they spend playing in any given performance is very high. However, other musicians may not be as helpful and cause the judges to give an inaccurate score, in which case we would give them a small feature attribution. This low score could be for a number of reasons. For example, it could be because those musicians are just bad at playing music, which is distracting to the judge. Or perhaps, they're fine musicians but they play out of harmony with the rest of the orchestra. Or maybe they're fantastic musicians and cause the judges to give a very positive assessment of the orchestra's talent, even though the rest of the orchestra is really bad. In any of these cases, the feature attribution of that musician should be given a smaller value because their contribution negatively affects the judge's ability to provide an accurate score of the orchestra's overall skill. For either high or low feature attribution scores, the assessment to determine the attribution of a single musician to the judge's overall prediction of the orchestra's talent is still "how much did that musician influence the judge's rating?"

Choosing an appropriate feature attribution technique is not just a matter of finding the latest or most accurate state-of-the-art method. Techniques can vary wildly, or even be completely opposed in the attribution they give to different features. Figure 2-1 shows an example of this by comparing seven different feature attribution techniques for the same dataset and model:

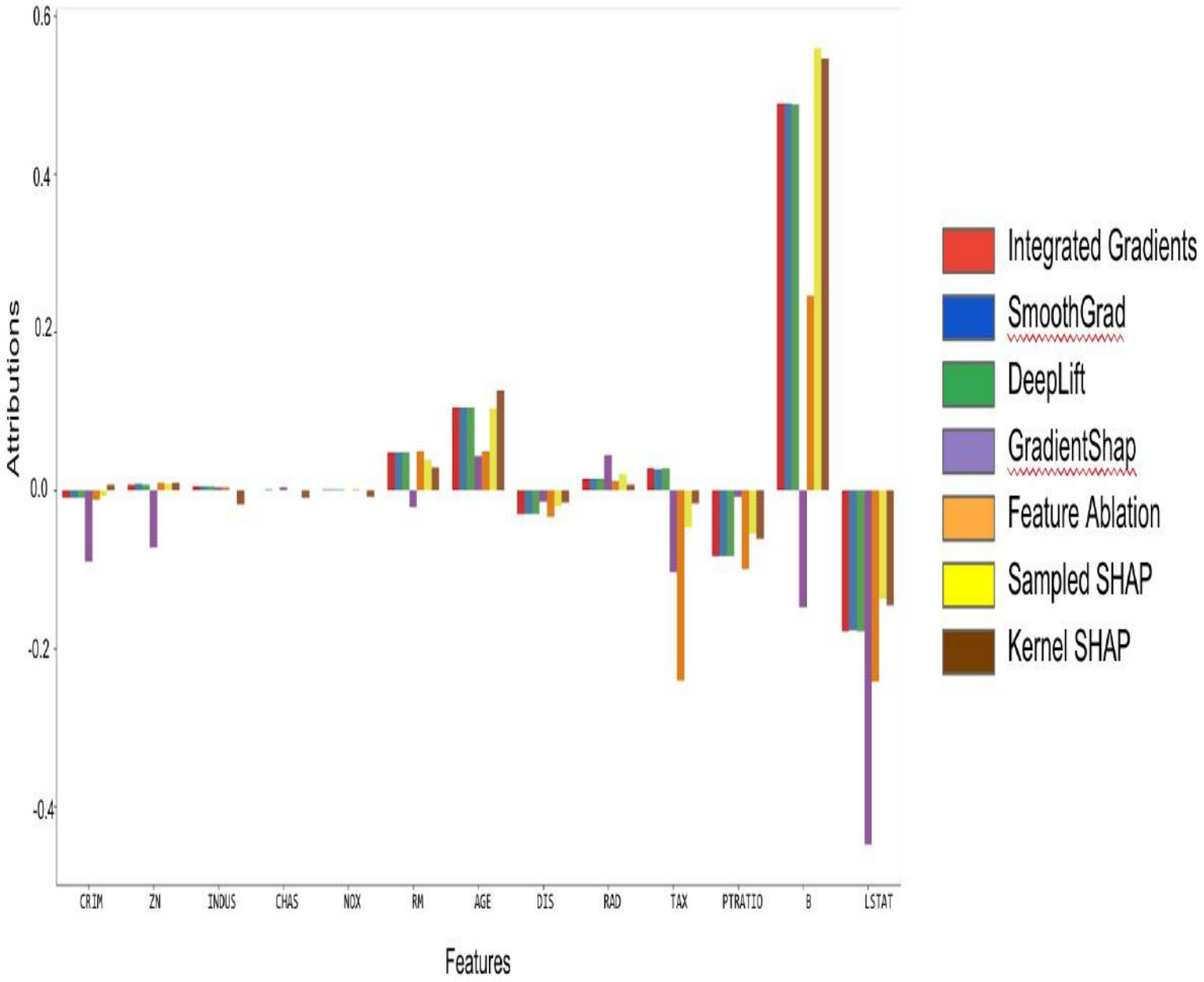


Figure 1-1. Feature attributions across seven different techniques for a PyTorch model trained on the Boston Housing dataset.

In this example, the Integrated Gradients (IG), SmoothGrad, and DeepLift techniques have close agreement in their feature attribution values, despite the fact that DeepLift is conceptually a different approach from IG and SmoothGrad. Conversely, the GradientSHAP technique strongly diverges from its sibling techniques, Sampled SHAP and Kernel SHAP, despite all three using the same underlying theory. Finally, we can see how Feature Ablation, which is not similar to any of the other methods, sometimes has very similar attributions to the other techniques (i.e., for RM and PTRATIO), but has attributed influence to B and TAX unlike any other method.

Feature attributions may still seem abstract, which is okay because they rely on the modality of the dataset. For example, in Chapter 3, Explainability for Tabular Data, we will discuss feature attributions as numerical values because the inputs are scalar. By comparison, in Chapter 4, Explainability for Computer Vision, feature attributions are the contribution by individual pixels in the image to the prediction. In Chapter 5, which is focused on NLP, the attributions are typically tokenized words.

Shapley values

One commonly used method to determine feature attribution is Shapley values. Shapley values use game theory to determine a feature's influence on a prediction. Unlike a technique like feature permutation (see Chapter 3 where we discuss Permutation Feature Importance), which relies on changing the values of features to estimate their impact, Shapley values are purely observational, instead inferring feature attributions through testing combinations with different groups of features.

A Shapley value is calculated using cooperative game theory; Lloyd Shapley published the technique in 1951, and it contributed to his 2012 Nobel prize in Economics. It is useful to understand the core idea behind Shapley Values in order to decide if they're worth it for your use case, and be able to use them effectively with your ML model. Shapley Values rely on examining how each feature influences the predicted value of a model by generating many predictions based on a partial set of the features used by the model and comparing the results of the predicted values.

We can describe how Shapley values are computed in two ways, coalitions and paths. With coalitions, Shapley are represented by grouping the features of a dataset into multiple, overlapping subsets, which are the coalitions. For each coalition, a prediction value is generated using only the features in that coalition and compared against the prediction for a coalition that includes one additional feature. Calculating Shapley values can also be framed as a dynamic programming problem, where a series of steps, or paths, are generated. Each step in a path represents incrementally including another feature from the dataset to generate the prediction. We generate many different paths to represent the different permutations of the order in which features could be included.

More concretely, imagine we had a weather dataset, and we wanted to predict the amount of rain (in inches). In this example, we'll refer to this prediction function as `P()` which takes any subset of features, e.g., `{feature_1, feature_2, …}` as its input. For the purposes of explaining how Shapley Values are calculated, it is not important to understand how`P()` determined the predicted values.[3] The ability to explain an opaque model, while not knowing how it works internally, is one of the advantages of many Explainability techniques.

In the simplest version of our weather dataset, we just have two features, `temperature` and `cloud_cover`. To calculate the Shapley Values for each of the individual features `temperature` and `cloud_cover`, we first compute four individual predictions for different combinations of features in the dataset. For now, don't worry about how we arrived at these predicted values using only a subset of features—we'll discuss that later in this chapter, as well as in Chapter 3, in the section on Baselines.

1. `P({}) = 0` // Initially no features, also known as a null baseline
2. `P(temperature) = 2 inches`
3. `P( cloud_cover ) = 5 inches`
4. `P( {temperature, cloud_cover }) = 6 inches`

Our prediction that uses all of the features in the dataset is `P({temperature, cloud_cover})`, which gives us an estimated 6 (inches) of rain. To determine the Shapley Value individually for `temperature` we first remove our `P(cloud_cover)` prediction (5) from the overall `P({temperature, cloud_cover})` (6), leading to a contribution of 1. However, this is only part of the Shapley value, to compute the entire path we also need to move backwards from P(temperature) to P({}), leading to a

contribution of 2. We then average the contributions from each step on the path to arrive at a Shapley value of 1.5 for `temperature`.[4] Using the same approach, we can calculate the Shapley value for `cloud_cover` is 4.5 (by averaging 6 - 2 and 5-0).

This reveals a useful property of Shapley Values, *Efficiency*, meaning that the entire Prediction is equal to the sum of the Shapley values of individual features. In our example above, our combined contributions (Shapley values) of 1.5 (`temperature`) and 4.5 (`cloud_cover`) summed to 6 (our original prediction which included all of the features).

In this example, we have two paths, one to calculate the contribution of temperature and another to calculate the contribution of cloud_cover:

1. `P({}) → P({temperature}) → P({temperature, cloud_cover})`
2. `P({}) → P({cloud_cover}) → P({temperature, cloud_cover})`

What happens if we have more than two features? To accomplish this, we begin computing what are known as Shapley paths, or unordered, incremental groupings of features (also called coalitions). A path represents a way to go from no features in our prediction to the full set of features we used in our model. Each step in the path represents an additional feature in our coalition.

Let's expand our weather dataset to include a `humidity` feature, for an overall prediction that is `P({temperature, cloud_cover, humidity})`.

Our Shapley paths are now:

1. `P({}) → P({temperature}) → P({temperature, cloud_cover}) → P({cloud_cover, temperature, humidity})`
2. `P({}) → P({temperature}) → P({humidity, temperature}) → P({cloud_cover, temperature, humidity})`
3. `P({}) → P({cloud_cover}) → P({temperature, cloud_cover}) → P({cloud_cover, temperature, humidity})`
4. `P({}) → P({cloud_cover}) → P({cloud_cover, humidity}) → P({cloud_cover, temperature, humidity})`
5. `P({}) → P({humidity}) → P({humidity, temperature}) → P({cloud_cover, temperature, humidity})`
6. `P({}) → P({humidity}) → P({cloud_cover, humidity}) → P({cloud_cover, temperature, humidity})`

You may have noticed that we have repeated parts in our paths, such as `P({humidity, temperature})`. The ordering of the features does not matter for how Shapley values are organized into paths or coalitions. This is an important property, and the one most misunderstood, as we'll discuss in further detail later regarding a common misconception with Shapley values and causality.

Returning to our weather dataset, how would we calculate the Shapley Value for our `humidity` feature? First we need to know the predicted value for our different combinations of features:

1. `P({}) = 3`
2. `P(temperature) = 2`
3. `P(cloud_cover) = 4`
4. `P(humidity) = 5`
5. `P({temperature, cloud_cover}) = 6`
6. `P({temperature, humidity}) = 8`
7. `P({cloud_cover, humidity}) = 10`
8. `P({cloud_cover, temperature, humidity}) = 15`

In our earlier example, we did not need to calculate the indirect contribution of any feature. Now with multiple intermediate steps in our path, we need to determine how much each feature contributed along the path to the final predicted value. For the first path we expand each step in the path to include the contribution made by the new feature added to the coalition. To calculate the attributions, we take the difference of the predicted value before and after the step occurs:

Step 1 (Base Case):

```
P({}) → P({temperature})
Intermediate_Attribution_temperature =  P({temperature}) - P({}) = 2 - 3 = -1
```

Step 2:

```
P({temperature}) → P({temperature, cloud_cover})
Intermediate_Attribution_cloud_cover = P({temperature, cloud_cover}) - P({temperature}) = 6 - 2 = 4
```

Step 3:

```
 P({temperature, cloud_cover}) → P({cloud_cover, temperature, humidity})
Intermediate_Attribution_humidity = P({cloud_cover, temperature, humidity}) - P({temperature, cloud_cover})
= 15 - 6 = 9
```

Why are these intermediate attributions? We have more than one Shapley path, so we must continue to compute the partial attributions across all paths. To obtain the final feature attribution, we take the average of all of the intermediate attributions for that feature.

WHAT ABOUT CLASSIFICATION MODELS?

Although it is not immediately obvious, using Shapley values with classification models is perfectly okay, and not as hard as it seems. This is because almost all modern classification models represent classes not as a set of labels, but in some vector form (for example, either through embeddings, or one-hot encoding). In this case, our mental representation of how Shapley values push an inference towards or away from the final prediction value may not be as accurate. Rather than thinking of the Shapley value as pushing towards one class and away from another, envision a Shapley value as strengthening or weakening the predicted score for that individual class. For a multi-class classification model, the Shapley values are doing this strengthening and weakening for each class. A typical way to calculate Shapley values for multi-class classification is to independently calculate the Shapley values for each possible class, rather than just the predicted class.

Sampled Shapley technique

In a world with infinite time, you would calculate intermediate attributions for every possible path, representing every possible coalition of features. However, this quickly becomes computationally infeasible–for a dataset with ten features, we would need to get predictions for 2^{10} combinations[5] (or 1,023 additional predictions!). So almost every feature attribution technique you encounter that relies on Shapley values will use an optimized method known as sampled Shapley.

The idea behind sampled Shapley is that you can approximate these paths by either skipping steps in the path for some features where there does not appear to be a large contribution, sampling random coalitions and averaging the results using Monte Carlo methods, or following gradients. The tradeoff to using sampling is that we now have an *approximation* of the true attribution values, with an associated *approximation error*. Due to this approximation error, our Shapley values may not sum up to the predicted result. There is no universal way to calculate the approximation error, or "good" range from the approximation error, but generally you will want to try tuning the number of sampled paths to get the best tradeoff between computation performance and error; a higher number of sampled paths will decrease the error but increase runtime, and vice versa.

Baselines

In our examples of Shapley paths, we repeatedly referred to model prediction values using only a subset of the features our model was built for (e.g., `P{temperature, cloud_cover}` when our model took `temperature`, `cloud_cover`, and also `humidity` as inputs). There are several ways to compute these "partial" predictions, but for the purposes of this book we will focus on Shapley techniques which use Baselines.[6]

The main concept behind baselines is that if one can find a neutral, or *uninformative*, value for a feature, that value will not influence the prediction and therefore not contribute to the Shapley value. We can then use these uninformative values as placeholders in our model input when calculating the Shapley value for different feature coalitions.

In our rainfall example, our partial prediction of `P({cloud_cover = 0.8, humidity = 0.9}) = 10` may actually be fed to the model with a baseline value for temperature of `temperature = 22` (Celsius),

or `P({cloud_cover = 0.8, humidity = 0.9, temperature = 22}) = 10`

Baselines can vary, there may be an uninformative baseline which is best for different groups of predictions, or a carefully crafted baseline which is uninformative across your entire training dataset. We discuss the best way to craft baselines in chapters 3-6, which are focused on explanations for different modalities.

WHICH WAY TO CALCULATE SHAPLEY VALUES?

Calculating Shapley Values as originally intended (by entirely removing features and observing the result) is rarely feasible in ML because most datasets are historical- it is not feasible to "rerun" or recreate the environment of the dataset without a certain feature, or combination of features, and observe what a new predicted value would be. Due to this, a variety of techniques have been invented over time to overcome these limitations by addressing how to treat the removed features and combinations in intermediate steps of path. Many involve creating a synthetic value for the removed feature(s) that allows the model to be used as-is with minimal contribution from the removed features, rather than trying to alter the model architecture or final predicted value, and often assuming that the ordering of features in the intermediate path steps does not matter. For example, the open source SHAP library has four different implementations for efficiently calculating Shapley values based on the model architecture. For a deeper discussion of the different Shapley value techniques, see the very well written The Many Shapley Values for Model Explanation by Sundararajan and Najmi.

You will often see Shapley values referenced as a method for explanations, or forming the basis behind others. As they have seen many decades of use across a variety of fields, and been very well studied in academia, they represent one of the most proven techniques for explainability. However, as we saw in discussing sampled Shapley, true Shapley values are computationally infeasible for most datasets, so there are tradeoffs to this technique due to the lack of precision. Likewise, although superficially it is easier to understand feature attributions based on Shapley values, it can be quite difficult to explain the game theory concepts to stakeholders to build trust in use of these techniques.

Gradient-based techniques

Gradient based approaches towards explainability are some of the most powerful and commonly used techniques in the field. Deep learning models are differentiable by construction and the derivative of a function contains important information about the local behavior of the function. Furthermore, since gradients are needed to fuel the gradient descent process that most machine learning models are trained upon, the tools of autodifferentiation for computing gradients are robust and well established in computing libraries.

A gradient is a high dimensional analog of the derivative of a function in one variable. Just as the derivative measures the rate of change of a function at a point, the gradient of a real-valued function is a vector indicating the direction of the steepest ascent. Gradient descent relies on the gradient in the parameter space to find the parameters that minimize the loss. Similarly, measuring the gradient of a model function with respect to its inputs gives valuable information as to how the model predictions

may change if the inputs change as well. This in turn, is very useful for explainability. The gradient contains exactly the information we need to say “if the input changed this much in this way, the model’s prediction would change as well.” Many of the explainability techniques we discuss in this book are based on evaluating how the value of predictions change as the values of features are changing as well.

Gradient based methods are particularly common for image based models (see Chapter 4) and they provide an intuitive picture of how these methods are typically applied. Given some example input image, to measure the gradient of how the model arrives at its prediction for that example, the image is varied along a path in feature space from some baseline to the values of the original input image.

But what exactly is a path in feature space? For images, we can think of a picture that is 32 x 32 pixels, like the images in the CIFAR-10 dataset, with 3 channels for RGB values as a vector in 3,072 dimensional space. Modifying the elements of that vector modifies the picture it represents. For gradient based techniques, you start with a baseline image (similar in spirit to the role of the baseline in computing Shapley values) and construct a path in that 3,072 dimensional space that connects the baseline with the vector representing the input image.

Gradient based techniques for images make a lot of sense because they take advantage of the intrinsic property of an image, namely that it represents an array of uniform features with a fixed, linear scale (e.g., from 0.0 to 1.0 or 0 to 255), This allows these techniques to rapidly evaluate multiple versions, or steps, of an image as they move along the gradient. The technique will then combine its observations from these steps to calculate an attribution value for each pixel[7], and in some cases, segment these attributions into regions. Gradient based techniques are also used for tabular (Chapter 3) and text (Chapter 5) models but the type of baseline you use for those cases will vary.

If this technique of determining attribution values sounds similar to the discussion of Shapley Values in this chapter, and Sampled Shapley, which is covered in-depth in Chapter 3, that’s because it is! Both integrated gradients and Shapley values are techniques that measure the individual influence of individual features in the model and they do that by measuring how the model’s prediction changes as new information of the input example is introduced. For Shapley values this is done combinatorially by adding features individually or in coalitions to determine which features or coalitions have the strongest influence in comparison to a baseline prediction. However, with images, this approach is a bit naive as it rarely makes sense to entirely remove “features” (or substitute with a baseline value) given that pixels location in the image is relevant. Furthermore, the pixel’s value relative to its neighbors also gives us important information, such as whether it constitutes the edge of an object. The idea of “dropping out” entire regions of an image to understand pixel influence has not yet been well explored, and given the computational complexity of exhaustively generating all (or random) regions, it is likely this type of technique will be more used in interpretable models in the future, rather than as a stand-alone, model-agnostic explainability technique.

Saliency maps and feature attributions

Saliency maps arise in various contexts in machine learning but are probably most familiar in computer vision. Saliency maps, broadly, refer to any technique that aims to determine particular regions or pixels of an image that are somehow more important than others. For example, saliency maps can be used to highlight the regions in an image to better understand where and how a human first focuses their

attention by tracking eye movements. The MIT/Tuebingen Saliency dataset is a benchmark dataset for eye movement tasks. The saliency maps in this dataset indicate important regions in an image from tracking human eye movements and areas of fixation, see Figure 4-1.

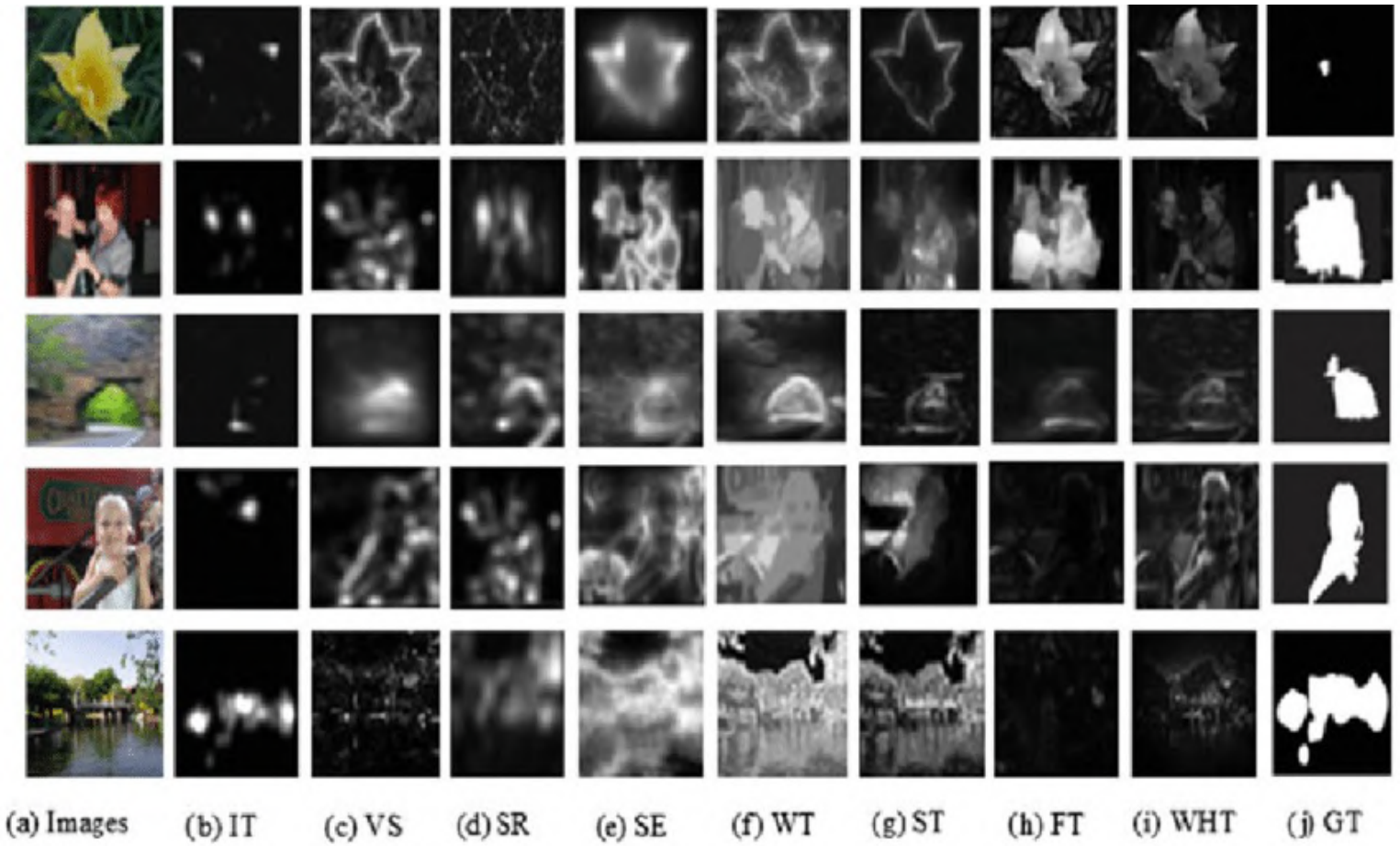


Figure 1-2. Examples from the MIT/Tuebingen Saliency dataset

In the context of explainability and feature attributions, saliency methods serve a similar purpose; that is, they emphasize the regions or pixels of an image that indicate where a trained model focuses its attention to arrive at a given prediction. For example, in Chapter 4 we'll see how the method of integrated gradients can be used to produce a mask or overlay highlighting the pixels that contributed most to the model's prediction. We'll see precisely how the mechanics of integrated gradients and other gradient based techniques lead to this kind of attribution mask as well as a host of other saliency-based explainability techniques that can be applied to image models.

Surrogate Models

Another form of explaining a model's behavior is to use a simplified version of the model, the *surrogate,* to give more explanatory power directly from observing the architecture of the model (also known as *model distillation*). In this sense, surrogate models represent a halfway point between post-hoc explainability and intrinsic interpretable models. While this may sound like the best of both worlds, the tradeoff is that a surrogate model almost always has worse performance than the original model, and usually cannot guarantee that all predictions are accurately explained, particularly in edge cases or areas of the dataset that were under-represented during training.

Surrogate models usually have a linear, decision tree, or rule-based architecture. Where possible, we try

to highlight the ability to use a surrogate model, but the field of automated model distillation still resides more in research labs than industry. What this usually means is that you must build your own techniques, or adapt proof-of-concepts, to make your surrogate models.

Activation

Rather than explaining model behavior by which features influenced the model's prediction, activation methods provide insight into what parts of the model's architecture influenced the prediction. In a DNN, this may be the layers which were most pivotal in the model's classification, or individual neuron's contribution to the final output. Going even further, some techniques seek to explain through *concepts* learned by the model, driven by what was activated within the architecture for a given input.

Likewise, during training, an individual data point may be active in only certain circumstances, and strongly contribute to a certain set of labels, or range of predicted values. This is typically referred to as *training influence*, but is analogous to activations within the model architecture.

Activation methods are among the most recently proposed explainability techniques in machine learning and provide an intriguing approach to leveraging the internal state of a complex model to better understand the model behavior. However, these techniques haven't yet been widely adopted among practitioners in the community and so you will see less applications of them in this book although Concept Activation Vectors are discussed in Chapter 6.

Putting It All Together

While we may think of explanations as being primarily about their utility, meaning, and accuracy, understanding the ways in which explanations can vary is the first step in choosing the right tool for the job. By understanding what type of explanation will be most useful to your audience, you can go straight to using the best technique. Choosing an explanation method by simply trying many different techniques until an explanation looks good enough is closer to asking ten strangers you find on the street for investment advice. You may get a lot of interesting opinions, but it is unlikely most of them will be valid.

Start with asking yourself about who is receiving the explanation, what is it that needs to be explained, and what will happen after the explanation? For example, an end-user receiving an explanation about being denied a loan will not find a global explanation focused on understanding the model's architecture to be relevant or actionable. Better to use a technique that is local and focused on the features in order to center the explanation on factors in the user's control. You may even add a technique that provides a counter-factual explanation; for example, that the loan would likely have been approved if their credit score and finances were more similar to other consumers who had requested the same loan amount.

Business stakeholders, on the other hand, want to see the big picture. What types of features are globally influential in this model? Why should this more complex, opaque model be trusted over the previous linear model that's been in use for years? A global feature attribution technique that is post-hoc and model-agnostic could be used to compare how both models behave.

Summary

In this chapter we gave a high level overview of the main ideas you are likely to consider as a practitioner developing explainable ML solutions. We started by discussing what explanations and how an explanation may change depending on the audience (e.g. ML Engineer vs Business Stakeholders vs Users). Each of these groups have distinct needs and thus will interact with explanations in their own way.

We then discussed the different types of common explainability techniques, providing a simple taxonomy that we can use to frame the methods we will discuss in the later chapters of the book. Lastly we covered some of the recurring themes that arise throughout explainability, like the idea of feature attribution, gradient based techniques, saliency maps and more recent developments like surrogate models and activation maps.

In the following chapters, we will dive into explainability for different types of data, starting with tabular datasets in Chapter 3. As you will see in Chapter 3, all of the background and terminology in this chapter is immediately put into practice now that we have given you the knowledge to understand and distinguish between different types of techniques.

---

1 Ironically, the use of neurons and neural nets was inspired by trying to explain how our brain's vision system was able to accomplish seemingly impossible tasks, like performing edge detection and pattern classification before an object was classified by our consciousness mind.

2 Following Google developer guidelines, we avoid the use of the phrase "black box" in this book. See https://developers.google.com/style/word-list#black-box for more details.

3 Also, P() is not a probability function, so you may encounter combinations of inputs that violate assumptions about probability function notation.

4 For larger coalition sizes, the average is weighed by the number of coalitions of that size. See the formal definition of Shapley values for details.

5 The actual number of predictions could be even higher if you did not optimize and save the results of predictions to be reused between different paths.

6 Other options include Conditional Expectations, and RBShap. /cite{Sundararajan, Najmi, 2019} provide an excellent breakdown of these different approaches, including tradeoffs and real-world examples.

7 As we'll explore further in the chapter, the exact way of calculating how to combine these steps is a key differentiating factor between techniques.

Chapter 2. Explainability for Image Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Introduced in the 1980s, convolutional neural networks (CNNs), like DNNs, remained unused until the advent of modern ML, at which point they quickly became the backbone of contemporary solutions for computer vision problems. Since then deep learning models based on CNNs have enabled unprecedented breakthroughs in many computer vision tasks ranging from image classification and semantic segmentation to image captioning and visual question answering, at times achieving near human level performance. Nowadays you can find sophisticated computer vision models being used to design smart cities, to monitor livestock or crop development, to build self-driving cars, or to identify eye disease or lung cancer.

As the number of these intelligent systems relying on image models continues to grow, the role of explainability in analyzing and understanding these systems has become more important than ever. Unfortunately, when these highly complex systems fail they can do so without any warning or explanation and sometimes with unfortunate consequences. Explainability techniques are essential to build trust not only in the users of these systems, but especially for the practitioners putting these models into production.

In this chapter we'll focus on explainability methods that you can use to build more reliable and transparent computer vision ML solutions. Computer vision tasks differ from other ML tasks in that the base features (i.e., pixels) are rarely influential individually. Instead what is important is how these pixel-level features combine to create higher-level features like edges, textures or patterns that we recognize. This requires special care when discussing and interacting with explainability methods for computer vision tasks. We'll see how many of these tools have been adapted to address those concerns. Just as CNN's were developed with images in mind, many of the explainability techniques we cover in this chapter were also developed for images or even CNNs.

Broadly speaking, most explainability methods for image models can be classified as either back-propagation methods, perturbation methods, methods that utilize the internal state or some combination of these approaches. The techniques that we'll discuss in this chapter are representative of those groups.

LIME's algorithm uses perturbed input examples to approximate an interpretable model. Like the name suggests, Integrated gradients and XRAI depend on back propagation while Guided Backprop, Grad-CAM and its relatives utilize the models internal state.

Integrated Gradients

What you need to know about Integrated Gradients:

- Integrated gradients was one of the first successful approaches to model explainability.
- Integrated Gradients is a local attribution method, meaning it provides an explanation for a model's prediction for a single example image.
- Produces an easy to interpret saliency mask that highlights the pixels or regions in the image that contribute most to the model's prediction.

Pros	Cons
One of the first successful approaches to model explainability	Requires differentiability of the model and access to the gradients, so does not apply well to tree based models
Can be applied to any differentiable model for any data type, images, text, tabular, etc.	Results can be sensitive to hyperparameters or choice of baseline.
Easy and intuitive implementation that even novice practitioners can apply.	
Better for low-contrast images or images taken in non-natural environments such as X-rays.	

Suppose you were asked to classify the image in Figure 2-1. What would your answer be? How would you explain how you made that decision? If you answered "bird", what exactly made you think that? Was it the beak, the wings, the tail? If you answered "cockatoo", was it because of the crest and the white plumage? Maybe you said it was a sulfur-crested cockatoo because of the yellow in the crest.

Figure 2-1. What features tell us this is a sulfur-crested cockatoo? Is it the beak and wings? The white plumage? The yellow crest? Or all of the above?

Regardless of your answer, you made a decision based on certain features of the image, more specifically, because of the arrangement and values of certain pixels and pixel regions in the image. Perhaps the beak and wings indicate to you that this is a picture of a bird while the crest and coloring tells you it is a cockatoo. The method of integrated gradients provides a means to highlight those pixels which are more or less relevant for a model's (in this case, your own brain's) prediction.

Using gradients to determine attribution of model features makes intuitive sense. Remember that the

gradient of a function tells us how the function values change when the inputs are changed slightly. For just one dimension, if the derivative is positive (or negative) that tells the function is increasing (or decreasing) with respect to the input. Since the gradient is a vector of derivatives, the gradient tells us for each input feature if the model function prediction will increase or decrease when you take a tiny step in some direction of the feature space.[1] The more the model prediction depends on a feature, the higher the attribution value for that feature.

> **NOTE**
>
> For linear models this relationship between gradients and attribution is even more explicit since the sign of a coefficient indicates exactly positive or negative relationship between the model output and the feature value. See also the discussion and examples in the section on Gradient x Input in Chapter 5.

However, relying on gradient information alone can be problematic. Gradients only give local information about the model function behavior but this linear interpretation is limiting. Once the model is confident in its prediction, small changes in the inputs won't make much difference. For example, when given an image of a cockatoo, if your model is robust and has a prediction score of 0.88, modifying a few pixels slightly (even pixels that pertain to the cockatoo itself) likely won't change the prediction score for that class, and thus the gradient will be zero. This is called gradient saturation.

To address this issue, the Integrated Gradients technique examines the model gradients along a path in feature space, summing up the gradient contributions along the path. At a high level, Integrated Gradients determine the salient inputs by gradually varying the network input from a baseline to the original input and aggregating the gradients along the path. We'll discuss how to choose a baseline in the next session. For now, all you need to know is that the ideal baseline should contain no pertinent information to the model's prediction, so that as we move along the path from the baseline to the image we introduce information (i.e., features) to the model. As the model gets more information, the prediction score changes in a meaningful way. By accumulating gradients along the path we can use the model gradient to see which input features contribute most to the model prediction. We'll start by discussing how to choose an appropriate baseline and then describe how to accumulate the gradients effectively to avoid this issue of saturated gradients.

Choosing a Baseline

To be able to objectively determine which pixels are important to our predicted label, we'll use a baseline image as comparison. As you saw in Chapter 2, baselines show up across different explainability techniques and our baseline image serves a similar purpose to the baseline for Shapley values. Similarly for images, a good baseline is one that contains neutral or uninformative pixel feature information and there are different baselines that you can use. When working with image models, the most commonly used baseline images are black image, white image or noise, as shown in Figure 2-2. However, it can also be beneficial to use a baseline image that represents the status quo in the images. For example, a computer vision model for classifying forms may use a baseline of the form template, or a model for quality control in a factory may include a baseline photo of the empty assembly line.

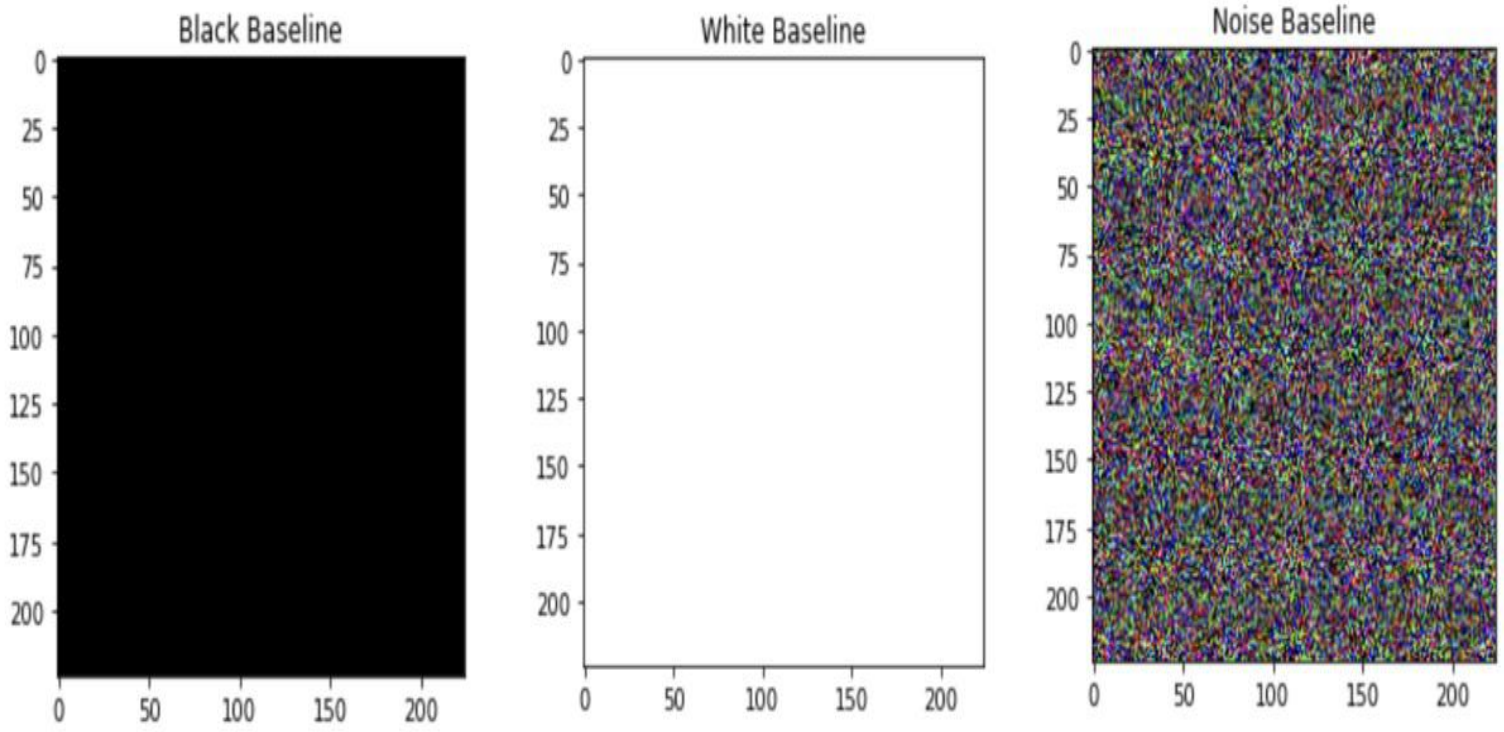


Figure 2-2. Commonly used baseline images for image models are black image, white image and an image of Gaussian noise.

OTHER BASELINES FOR IMAGES

The easiest baseline and resulting feature path to understand for images is one that varies the brightness of the image. For brightness scaling, you can naively apply uniform scaling for all of the values of all channels in the pixel from 0 to 100%, but this does not uniformly vary the perceived brightness in natural images.

Other common methods to create path in feature space include:

- Luminosity: Using a colorspace for the image which has a separate channel for brightness and only changing the value of that channel.
- Saturation: Varying the intensity of colors in the image from desaturated (gray) to fully saturated (original colors)
- Blur: Starting with a blurred version of the original image and then progressively applying less blur until the original, sharp image.

While some gradient-techniques are better suited for natural, real-world images, in principle any gradient-based should work on an image which has a continuous scale for the values of its channels, even synthetic images such as LIDAR depth maps or X-Rays.

We'll start with using a simple baseline that consists of a completely black image (i.e. no pixel information) and consider the straight line path from the baseline to the input image and examine the model's prediction score for its predicted class, as shown in Figure 2-3. A linear interpolation between two points $\boldsymbol{x}$, $\boldsymbol{y}$ is given by $\boldsymbol{\alpha y + (1 - \alpha)x}$ where the values of $\boldsymbol{\alpha}$ range from 0 to 1.

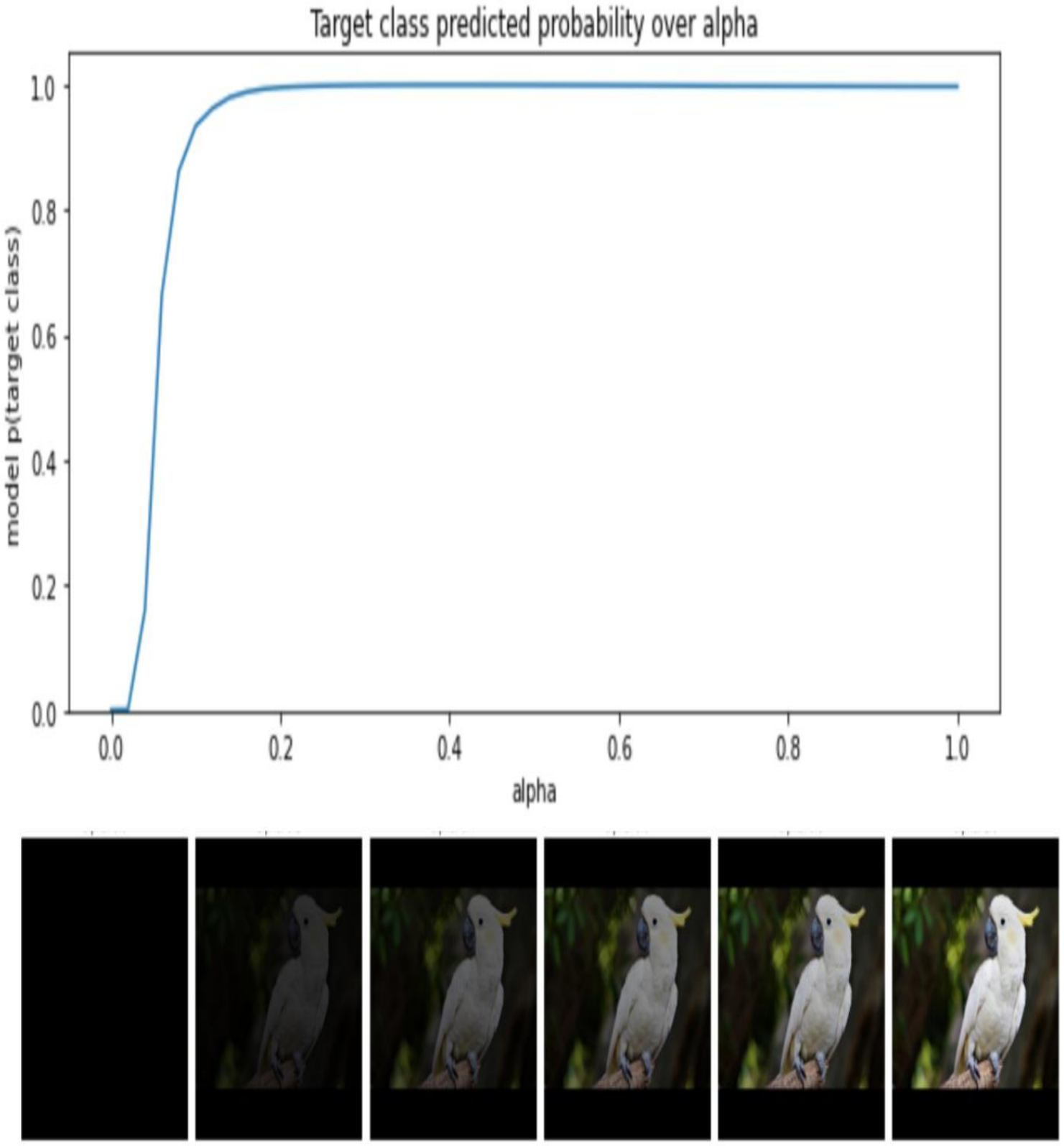


Figure 2-3. At some point in the straight line path from the baseline to the full input image, around when $alpha = 0.1$*, the model becomes very confident in the prediction "sulfur-crested cockatoo".*

We can achieve this straight line path in python with the interpolate_images function described here (see the GitHub repository for the full code example).

```
def interpolate_images(baseline,
                       image,
                       alphas):
    alphas_x = alphas[:, tf.newaxis, tf.newaxis, tf.newaxis]
```

```
    baseline_x = tf.expand_dims(baseline, axis=0)
    input_x = tf.expand_dims(image, axis=0)
    images = alphas_x * input_x + (1 - alphas_x) * baseline_x
    return images
```

The interpolate_images function produces a series of images as the values of the α's vary, starting from the baseline image when $\alpha = 0$ and ending at the full input image when $\alpha = 1$ as shown in Figure 2-4.

Figure 2-4. As the value of alpha varies from 0 to 1, we obtain a series of images creating a straight line path in image space from the baseline to the input image.

As α increases and more information is introduced to our baseline image, the signal sent to the model and our confidence in what is actually contained in the image increases. When $\alpha = 0$, at the baseline, there is, of course, no way for the model (or, anyone really) to be able to make an accurate prediction. There is no information in the image! However, as we increase α and move along the straight line path the content of the image becomes more clear and the model can make a reasonable prediction.

If we think of this mathematically, the confidence of the model's prediction is quantified in the value of the final softmax output layer. By calling prediction with our trained model on the interpolated images, we can directly examine the model's confidence in the label sulfur-crested cockatoo':

```
LABEL = 'sulfur-crested cockatoo'
pred = model(interpolated_images)
idx_cockatoo = np.where(imagenet_labels==LABEL)[0][0]
pred_proba = tf.nn.softmax(pred, axis=-1)[:, idx_cockatoo]
```

Not surprisingly, at some point before the $\alpha = 1$ the model has an 'aha!' moment and the model prediction determined to be 'cockatoo', as demonstrated in Figure 2-6.

We can also see here the importance of our choice of baseline. How would our model's predictions have changed if we started with a white baseline? Or a baseline from random noise? If we create the same plot as in Figure 2-3 but using the white baseline and noise baselines, we get different results. In

particular, for the white baseline the model's confidence in the predicted class jumps somewhere around $\alpha \approx 0.25$ while for the Gaussian noise baseline, the 'aha!' moment doesn't happen until $\alpha \approx 0.9$, as seen in Figure 2-5. The full code to create these examples is in the repository for the book.

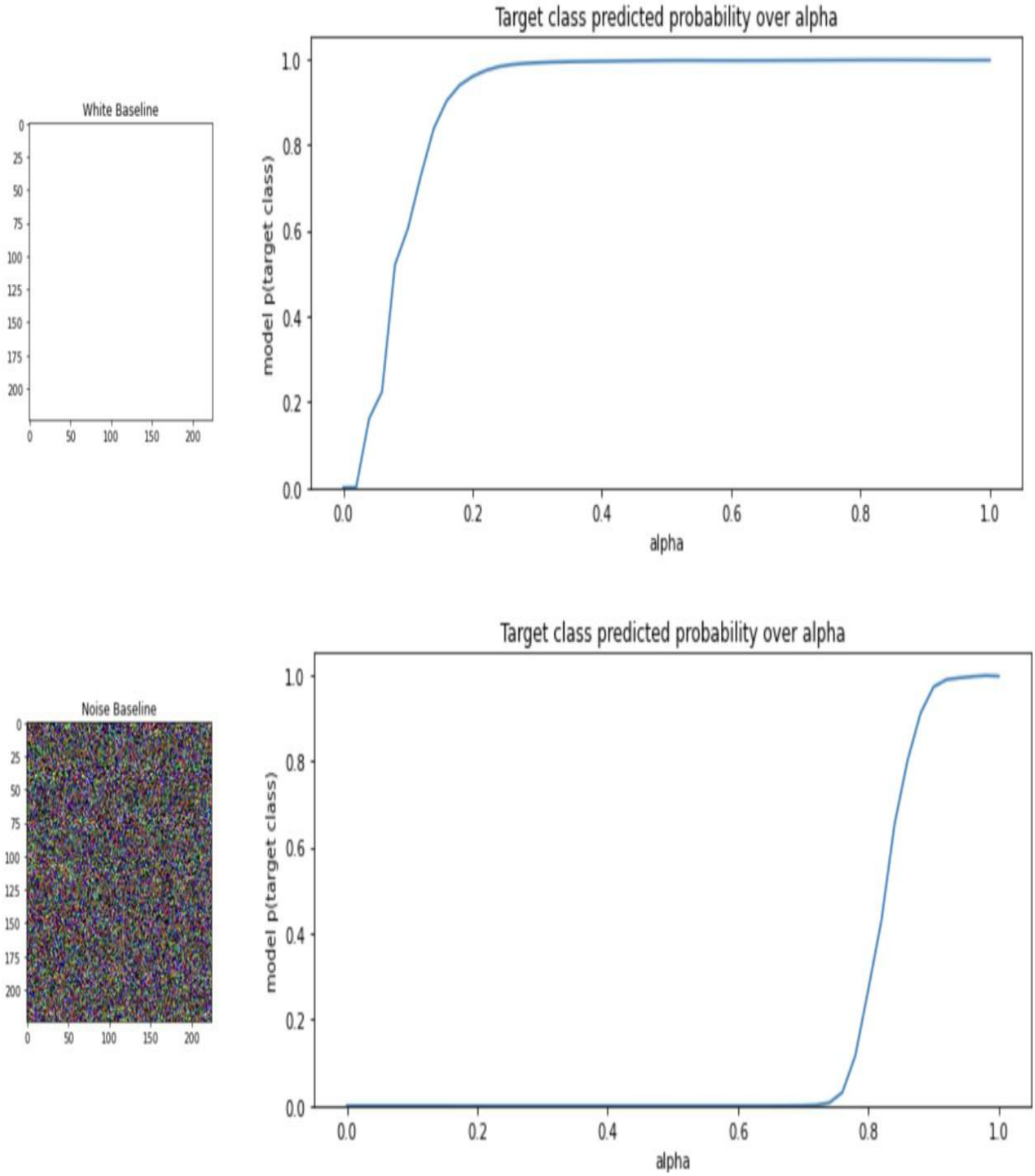


Figure 2-5. When using a white baseline (top) or a baseline based on Gaussian noise (bottom) the model takes longer to achieve the same confidence in the true label.

Accumulating Gradients

The last step for applying integrated gradients is then determining a way to use these gradients to decide which pixels or regions of pixels were the ones that most effectively contributed to that 'aha!' moment that we saw in Figure 2-3 when $alpha$ was approximately 0.15 and the model's confidence was over 0.98. We want to know how the network went from predicting nothing to eventually knowing the correct label. This is where the 'gradient' part of the integrated gradients technique comes into play. The gradient of a scalar valued function measures the direction of steepest ascent with respect to the function inputs. In this context, the function we are considering is the model's final output for the target class and the inputs are the pixel values.

To compute the gradient of our model function we can use Tensorflow's tf.GradientTape for automatic differentiation. We simply need to tell Tensorflow to 'watch' the input image tensors during the model computation. Note that here we are using Tensorflow but any library for performing automatic differentiation for other ML frameworks will work equally well.

```
def compute_gradients(images, target_class_idx):
    with tf.GradientTape() as tape:
        tape.watch(images)
        logits = model(images)
        probs = tf.nn.softmax(logits, axis=-1)[:, target_class_idx]
    return tape.gradient(probs, images)
```

Note that since the model returns a (1, 1001) shaped tensor with logits for each predicted class, we'll slice on target_class_idx, the index of the target class, so that we get only the predicted probability for the target class. Now, given a collection of images, the function compute_gradients will return the gradients of the model function.

Unfortunately, using the gradients directly is problematic because they can 'saturate'; that is, the probabilities for the target class plateau well before the value for α reaches 1. If we look at the average value of the magnitudes of the pixel gradients we see that the model learns the most when the value of alpha is lower, right around that 'aha' moment at $\alpha \approx 0.1$. After that, when α is greater than 0.2, the gradients go to zero, so nothing new is being learned, as seen in Figure 2-6.

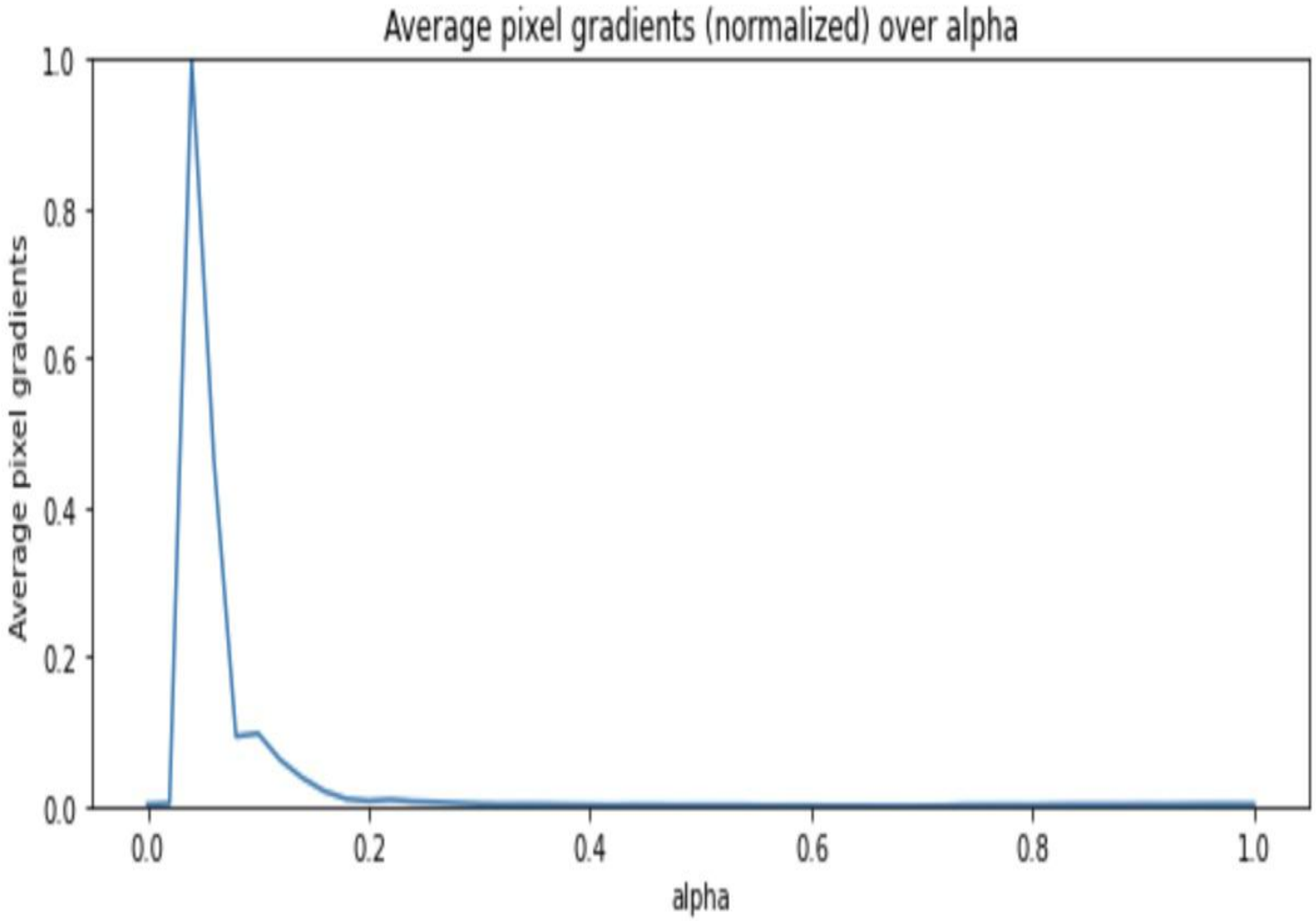


Figure 2-6. The model learns the most when the value of alpha is lower. After that, once $\alpha > 0.2$ *the pixel gradients go to zero.*

We want to know which pixels contributed most to the model's predicting the correct output class. By integrating over a path, integrated gradients avoid the problem of local gradients being saturated. The idea is that we accumulate the pixels' local gradients as we move along the straight line path from the baseline image to the input image. This way we accumulate a pixel's local gradients adding or subtracting its importance score to the model's overall output class probability. Formally, the importance value of the i-th pixel feature value of an image x for the model f is defined as

$$\mathrm{IG}_i\left(f, x, x'\right) = \int_{\alpha=0}^{\alpha=1} \left(x_i - x_i'\right) \frac{\partial f\left(x\hat{a} + \alpha\left(x - x'\right)\right)}{\partial x_i} d\alpha$$

Here x' denotes the baseline image. It may not look like it right away but this is precisely the line integral of the gradient with respect to the i-th feature on the straight line path from the baseline image to the input image. To compute this in code, we'd need to use a numeric approximation with Riemann sums summing over m steps. This number of steps parameter m is important and there is a tradeoff to consider when choosing the right value. If it's too small then the approximation will be inaccurate. If the number of steps is too large, the approximation will be near perfect but the computation time will be long. You will likely want to experiment with the number of steps.

When choosing the number of steps to use for the integral approximation, the original paper suggests to experiment in the range between 20 and 300 steps. However, this may vary depending on your dataset and use case. For example, a good place to start for natural images like those found in ImageNet is

$m = 50$. In practice, for some applications, it may be necessary to have an integral approximation within 5% error (or less!) of the actual integral. In these cases a few thousand steps may be needed though visual convergence can generally be achieved with far fewer steps. In practice, we have found that ten to thirty steps is a good range to start with.

Once you have computed the approximations you can check the quality of the approximation using a few examples by comparing the attribution score obtained from using integrated gradients with the difference of the input image's attribution score and the baseline image's attribution score. The following code block shows how to do just that.

```
# The baseline's prediction and attribution score
baseline_prediction = model(tf.expand_dims(baseline, 0))
baseline_score = tf.nn.softmax(tf.squeeze(baseline_prediction))[target_class_idx]
# Your model's prediction and attribution score
input_prediction = model(tf.expand_dims(input, 0))
input_score = tf.nn.softmax(tf.squeeze(input_prediction))[target_class_idx]
# Compare with the attribution score from integrated gradients
ig_score = tf.math.reduce_sum(attributions)
delta = ig_score - (input_score - baseline_score)
```

If the absolute value of delta is greater than 0.05, then you should increase the number of steps in the approximation. See the code in the integrated gradients notebook accompanying this book.

APPROXIMATING INTEGRALS WITH RIEMANN SUMS

Riemann sums are a foundational tool in integral calculus and can be used to find an approximation of the value of a definite integral. When implementing integrated gradients, you use Riemann sums to approximate the actual value of the integral. This approximation is made by summing up many, many rectangles whose height is defined by the value of the curve, as in Figure 2-7. There are various implementations when computing Riemann sums; you can use a left endpoint to determine the height of the rectangle, the right endpoint, the midpoint, and you can even use more complicated polygons like trapezoids to get more accurate approximations.

For each technique though, one important parameter is the number of partitions or rectangles you sum up to make the approximation. Formally, for well-behaved functions, as the number of partitions goes to infinity, the error between the Riemann sum approximation and the true area under the curve goes to zero. This presents a tradeoff: with more rectangles the approximation is more accurate but the computation cost also increases.

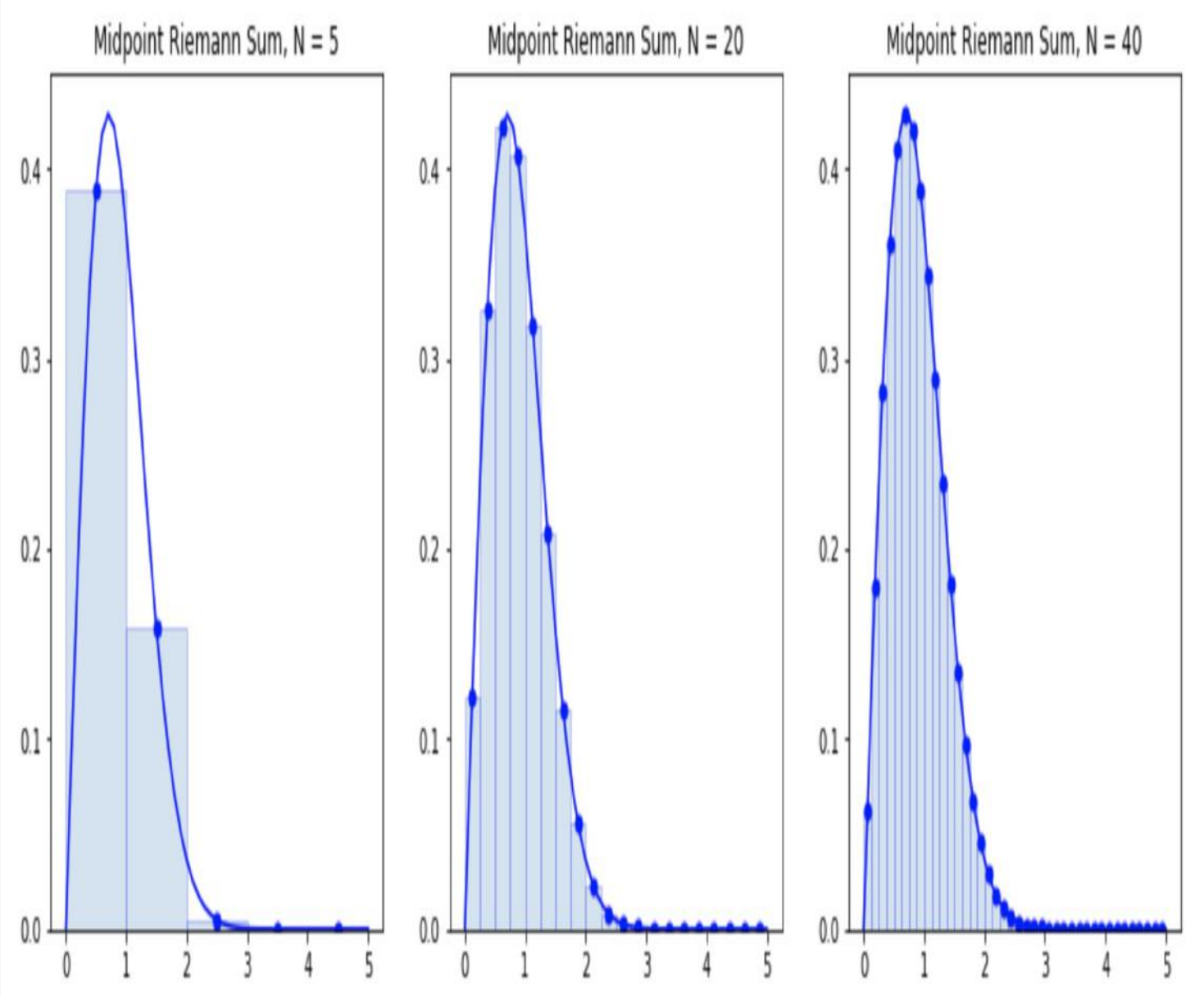


Figure 2-7. Formally, for reasonably well-behaved functions, as the number of rectangles goes to infinity, the error between the Riemann sum approximation and the true area under the curve goes to zero. This presents a tradeoff when using Riemann sums to approximate continuous integrals.

Using the hyperparameter m for the number of steps, we can approximate the integral for computing integrated gradients in the following way.

$$\text{IG}_i^{\text{approx}}\left(f, x, x'\right) = \left(x_i - x_i'\right) \sum_{k=1}^{m} \frac{1}{m} \frac{\partial f(x)}{\partial x_i}\Bigg|_{x=\text{interpolated images}}$$

In the notebook discussing integrated gradients in the GitHub repository for this book, you can see how each component of this sum is computed directly in python and tensorflow in the integrated_gradients function. First the α's are created and the gradients are collected along the straight line path in batches. Here the argument num determines the number of steps to use in the integral approximation:

```
# Generate alphas.
alphas = tf.linspace(start=0.0, stop=1.0, num=m_steps+1)
# Collect gradients.
gradient_batches = []
# Iterate alphas range and batch speed, efficiency, and scaling
```

```
    for alpha in tf.range(0, len(alphas), batch_size):
      from_ = alpha
      to = tf.minimum(from_ + batch_size, len(alphas))
      alpha_batch = alphas[from_:to]
      gradient_batch = one_batch(baseline, image, alpha_batch, target_class_idx)
      gradient_batches.append(gradient_batch)
```

Then those batch-wise gradients are combined into a single tensor and the integral approximation is computed, as shown in the following code. The number of gradients is controlled by the number of steps m_steps.

```
    # Concatenate path gradients.
    total_gradients = tf.concat(gradient_batches, axis=0)
    # Compute Integral approximation of all gradients.
    avg_gradients = integral_approximation(gradients=total_gradients)
  Finally, we scale the approximation and return the integrated gradient results.
    # Scale integrated gradients with respect to input.
    integrated_gradients = (image - baseline) * avg_gradients
```

You can visualize these attributions and overlay them on the original image. The following code sums the absolute values of the integrated gradients across the color channels to produce an attribution mask.

```
    attributions = integrated_gradients(baseline=black_baseline,
                                        image=input_image,
                                        target_class_idx=target_class_idx,
                                        m_steps=m_steps)
    attribution_mask = tf.reduce_sum(tf.math.abs(attributions), axis=-1)
```

You can then overlay the attribution mask with the original image as shown in Figure 2-8. See the notebook in the GitHub repository to see the full code for this example.

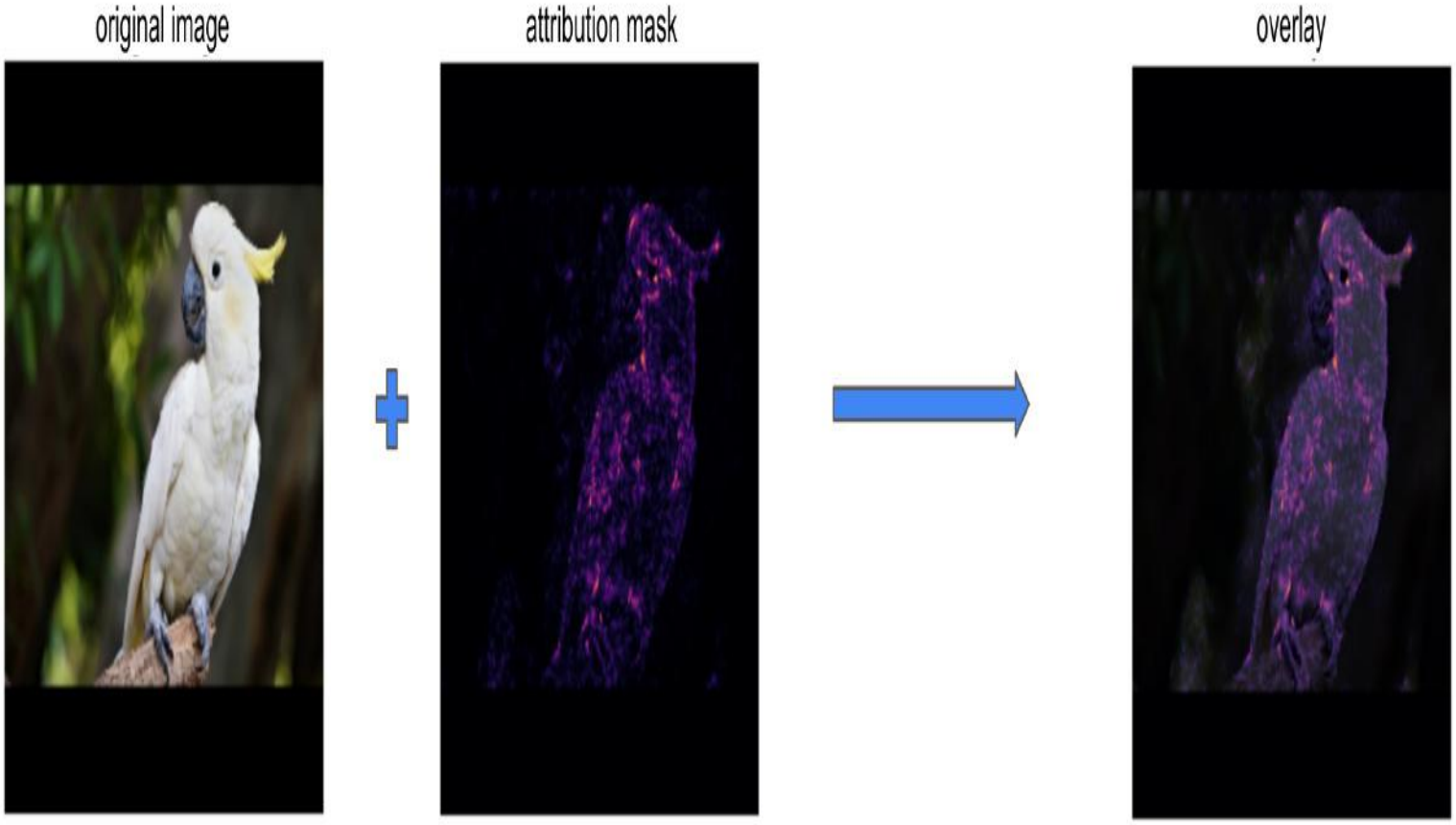


Figure 2-8. After computing feature attributions from the integrated gradients technique, overlaying the attribution mask over the original shows which parts of the image most contributed to the model's class prediction.

BASELINES MATTER.

It's interesting to compare the result of applying integrated gradients on our cockatoo example when using a black baseline vs a white baseline. Remember the intuition when choosing a baseline is that the baseline should have 'no information' and, typically, you can think that an all white or all black baseline as having no information. But in the example of a cockatoo this isn't the case. The cockatoo is predominantly white so a white baseline actually contains a lot of information of the features of a cockatoo. Not surprisingly when we compare the result of applying integrated gradients with these two baselines, they look quite different as shown in Figure 2-9 (see the notebook in the GitHub repository for the full code for this example).

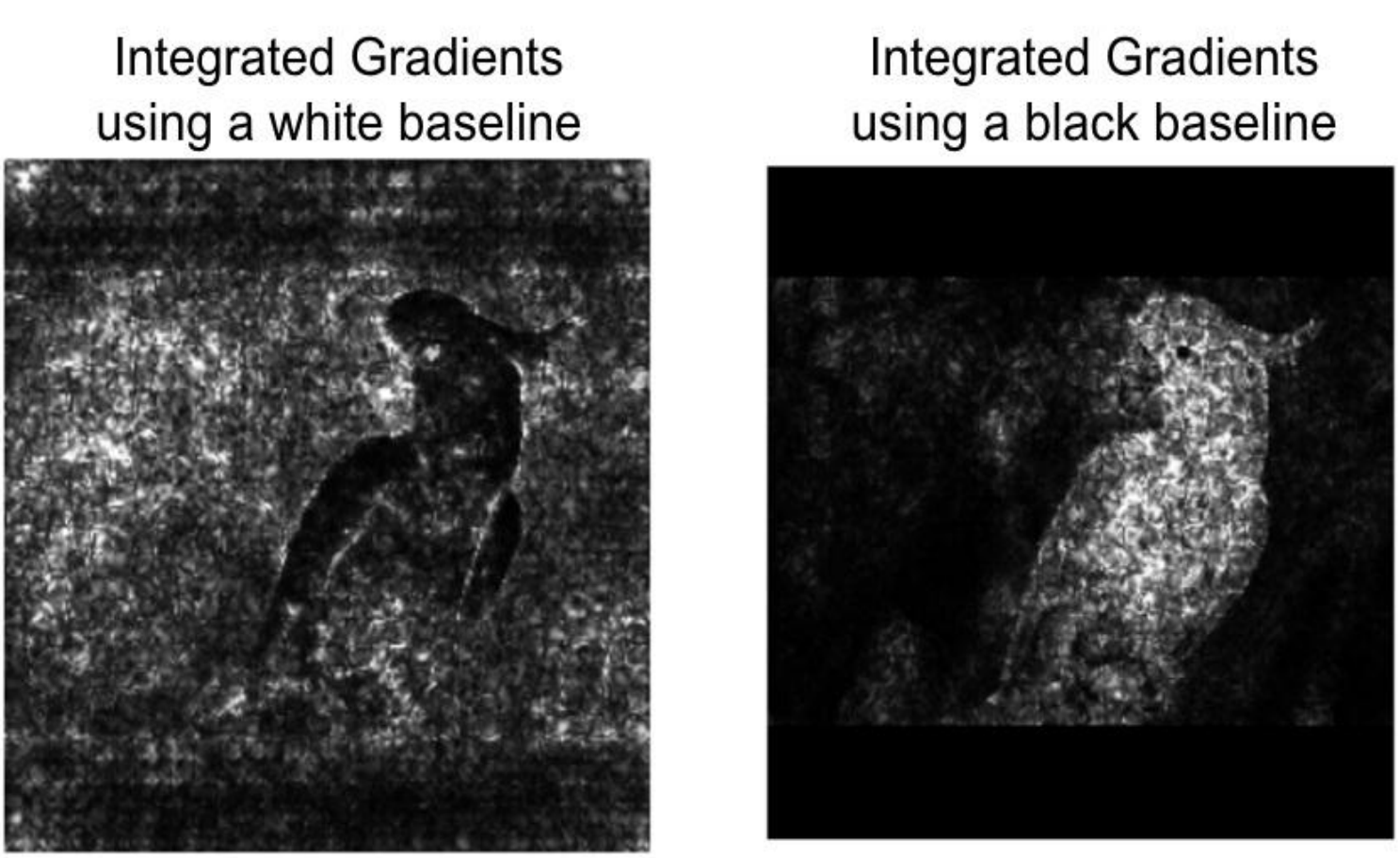


Figure 2-9. The result of applying integrated gradients with a white baseline versus a black baseline are quite different for a predominantly white image, like a sulfur-crested cockatoo.

The code and discussion in this section show really happening "under the hood" when implementing integrated gradients for model explainability. There are also more high level libraries that can be leveraged and have easy-to-use implementations. In particular, the saliency library developed by the People and AI Research (PAIR) group at Google contains easy to use implementations of Integrated Gradients, its many variations, as well as other explainability techniques. See this notebook in the book's GitHub repository to see how the saliency library can be used to find attribution masks via integrated gradients.

Improvements on Integrated Gradients

The method of integrated gradients is one of the most widely used and well-known gradient-based attribution techniques for explaining deep networks. However, for some input examples this method can produce spurious or noisy pixel attributions that aren't related to the model's predicted class. This is partly due to the accumulation of noise from regions of correlated, high magnitude gradients for irrelevant pixels that occur along the straight line path that is used when computing integrated gradients. This is also closely related to the choice of baseline that is used when computing integrated gradients for an image.

Various techniques have been introduced to address the problems that may arise when using Integrated Gradients. We'll discuss two variations on the classic integrated gradients approach: Blur Integrated Gradients (Blur-IG) and Guided Integrated Gradients.

Blur Integrated Gradients

In the section ‘Choosing a Baseline', you saw that when implementing integrated gradients the choice of baseline is very important and can have significant effects on the resulting explanations. Blur Integrated Gradients (Blur-IG) specifically addresses the issues that arise with choosing a specific baseline. In short, Blur-IG removes the need to provide a baseline as a parameter and instead advocates to use the blurred input image as the baseline when implementing integrated gradients.

This is done by applying a Gaussian blur filter parameterized by its variance α. We then compute the integrated gradients along the straight line path from this blurred image to the true, unblurred image. As σ increases the image becomes more and more blurred, as shown in Figure 2-10. The maximum scale $\sigma_{\max}$ should be chosen so that the maximally blurred image is information-less, meaning the image is so blurred it wouldn’t be possible to classify what is in the image.

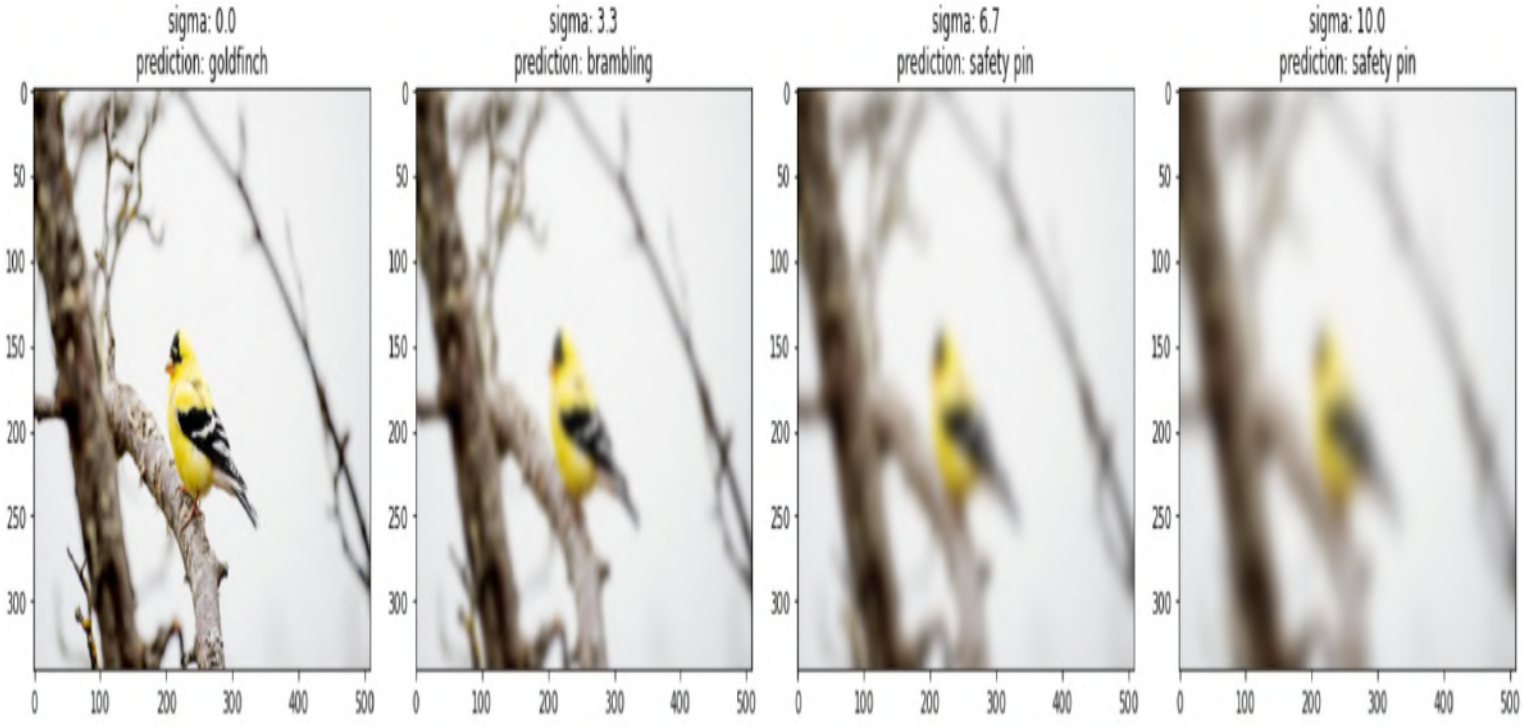


Figure 2-10. With no blur, the model predicts ‘goldfinch’ with 97.5% confidence but with $\sigma = 6.7$, the model’s top prediction becomes ’safety pin’.

The idea is that for different scale values of σ, different features are preserved or destroyed depending on the scale of the feature itself. The smaller the variation of the feature the smaller the value of σ at which it is destroyed, as seen in the detail in the wing patterns of the goldfinch in Figure 4-#. When σ is less than 3, the black and white pattern on the wings is still recognizable. However, for larger values of σ the wings and head are just a blur.

We can compare the saliency maps produced from using regular integrated gradients vs blur integrated gradients as in Figure 2-11. For this example, Blur-IG produces much more convincing attributions than the vanilla integrated gradients. See the notebook on integrated gradients for the full code for this example.

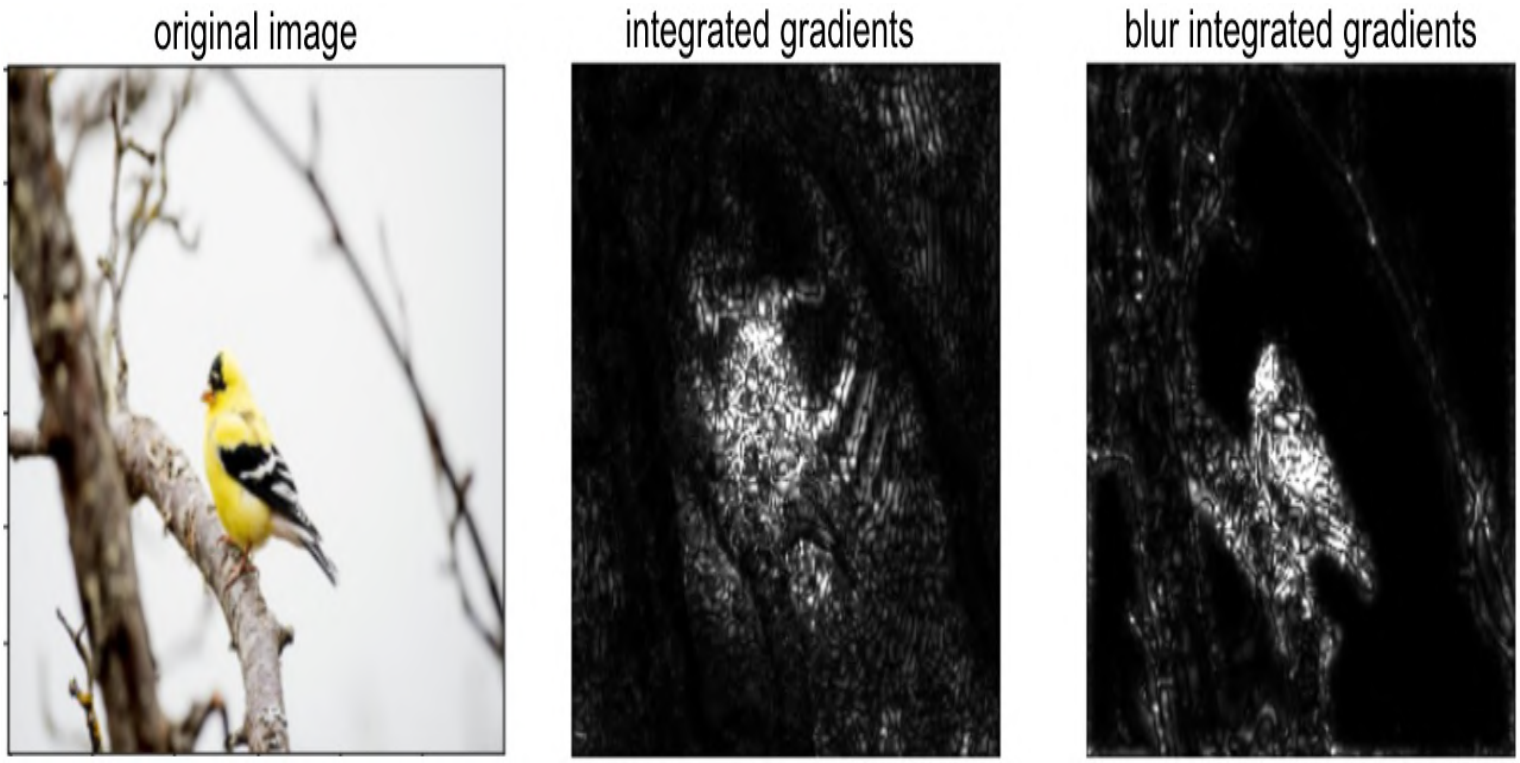


Figure 2-11. The saliency map for Blur-IG does a much better job of recognizing the parts of the image that make up the goldfinch.

Guided Integrated Gradients

Guided Integrated Gradients (Guided IG) attempts to improve upon integrated gradients by modifying the straight line path that is used in the implementation. Instead of using a straight line path from the baseline to the input image, Guided IG uses an adapted path to create saliency maps that are better aligned with the model's prediction. Instead of moving pixel intensities in a fixed straight line direction from the baseline to the input, we make a choice at every step. More precisely, Guided IG moves in the direction where features (i.e. pixels) have the smallest absolute value of partial derivatives. As the intensity of the pixels becomes equal to those of the input image, they are ignored.

The idea is that the typical straight line path that is used by previous integrated gradient techniques we've discussed so far could potentially travel through points in the feature space where the gradient norm is large and not pointing in the direction of the integration path. As a result, the naive straight line path leads to noise and gradient accumulation in saturated regions which causes spurious pixels or regions to be attributed too high of an importance when computing saliency maps. Guided IG avoids this problem by instead navigating along an adapted path in feature space, taking into account the geometry of the model surface in feature space. Figure 2-12 compares the result of applying vanilla integrated gradients with that of Guided Integrated Gradients. See the Integrated gradients notebook in the book's GitHub repository for the full code for this example.

NOTE

For the example image in Figure 4-#, the saliency map for the Guided IG example seems to focus on the goldfinch in the image more than the saliency map for integrated gradients but both methods don't seem to do a particularly great job of producing convincing explanations. This may indicate that our model needs more training, or more data. Or maybe that both Integrated Gradients and Guided IG just aren't well suited for this task or this dataset and another method would work better. There is no "one size fits all" XAI technique. This is why it's important to have a well stocked tool kit of techniques that you can use when analyzing your model predictions.

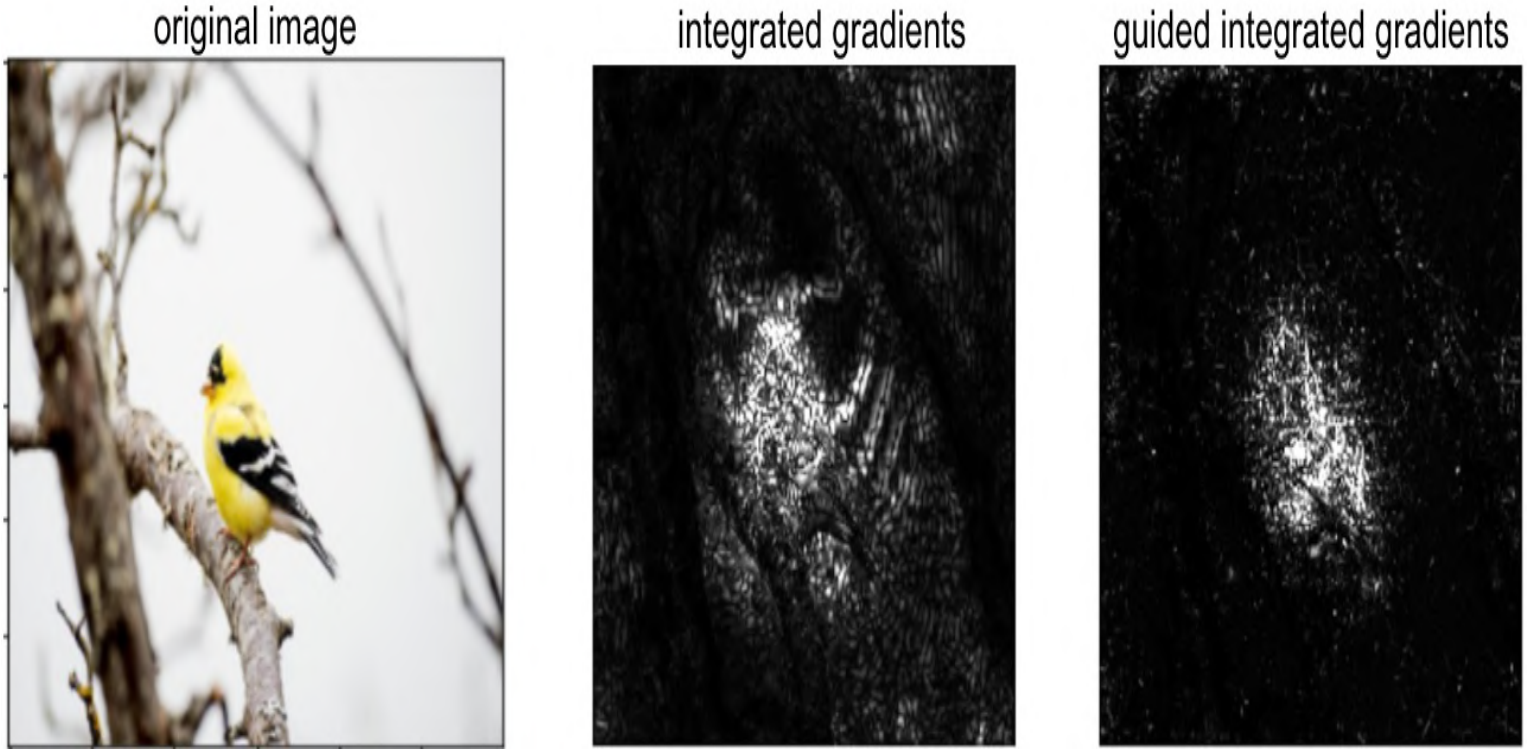


Figure 2-12. The saliency map for the Guided IG example focuses on the goldfinch in the image more than the saliency map for integrated gradients but both methods don't seem to do a particularly great job of producing convincing explanations.

XRAI

Here's what you need to know about XRAI

- XRAI is a region-based attribution method that builds upon integrated gradients
- XRAI determines regions of importance by segmenting the input image into like regions based on a similarity metric and then determines attribution scores for those like regions
- XRAI is a local explainability method that can be applied to any DNN based model as long as there is a way to cluster the input features into segments through some similarity metric

Pros	Cons
Improves upon other gradient based techniques like vanilla integrated gradients	Really only useful for image models
Can be faster than perturbation based-methods like LIME which require multiple queries to the model	Less granular than a technique like integrated gradients which provides pixel-level attribution
Performs best on natural images, like a picture of an animal or an object, similar to those found in the benchmark ImageNet and CIFAR datasets	Not recommended for low-contrast images or images taken in non-natural environments such as X-rays

The saliency maps obtained from applying integrated gradients provide an easy-to-understand tool to visually see which pixels contribute most to a model's prediction for a given image. XRAI builds upon the method of integrated gradients by joining pixels into regions. So instead of highlighting individual pixels that were most important, the saliency maps obtained by XRAI highlight pixel regions of interest in the image.

A key component and the first step of the XRAI algorithm is the segmentation of the image to

determine those regions of interest. Image segmentation is a popular use case in computer vision that aims to partition an image into multiple regions that are conceptually similar, as illustrated in Figure 2-13.

Figure 2-13. Image segmentation is a process of assigning a class to each pixel in an image. Here are some examples from the Common Objects in Context (COCO) dataset.

Image segmentation is a well-studied problem in computer vision and deep learning architectures like U-Net, Fast R-CNNs and Mask-RCNNs have been developed to specifically address this challenge and can produce state of the art results. One of the key steps of the XRAI algorithm uses an algorithm called Felzenszwalb's algorithm to segment an input image into similar regions, much in the same way as a nearest neighbors clustering algorithm. The Felzenszwalb algorithm doesn't rely on deep learning. Instead, it is a graph-based approach based on Kruskal's Minimum Spanning Tree algorithm and provides an incredibly efficient means to image segmentation. The key advantage of Felzenszwalb's algorithm is that it captures the important non-local regions of an image that are globally relevant and, at the same time, is computationally efficient with time complexity $\mathcal{O}(n \log n)$ where n is the number of pixels.

The idea is to represent an image as a connected graph $G = (V, E)$ of vertices V and edges E where each pixel is a vertex in the graph and the edges connect neighboring pixels. The segmentation algorithm then iteratively tests the importance of each region and refines the graph partitions, coalescing smaller regions into larger segments until an optimal segmentation is found, resulting in a segmentation as shown in Figure 2-14.

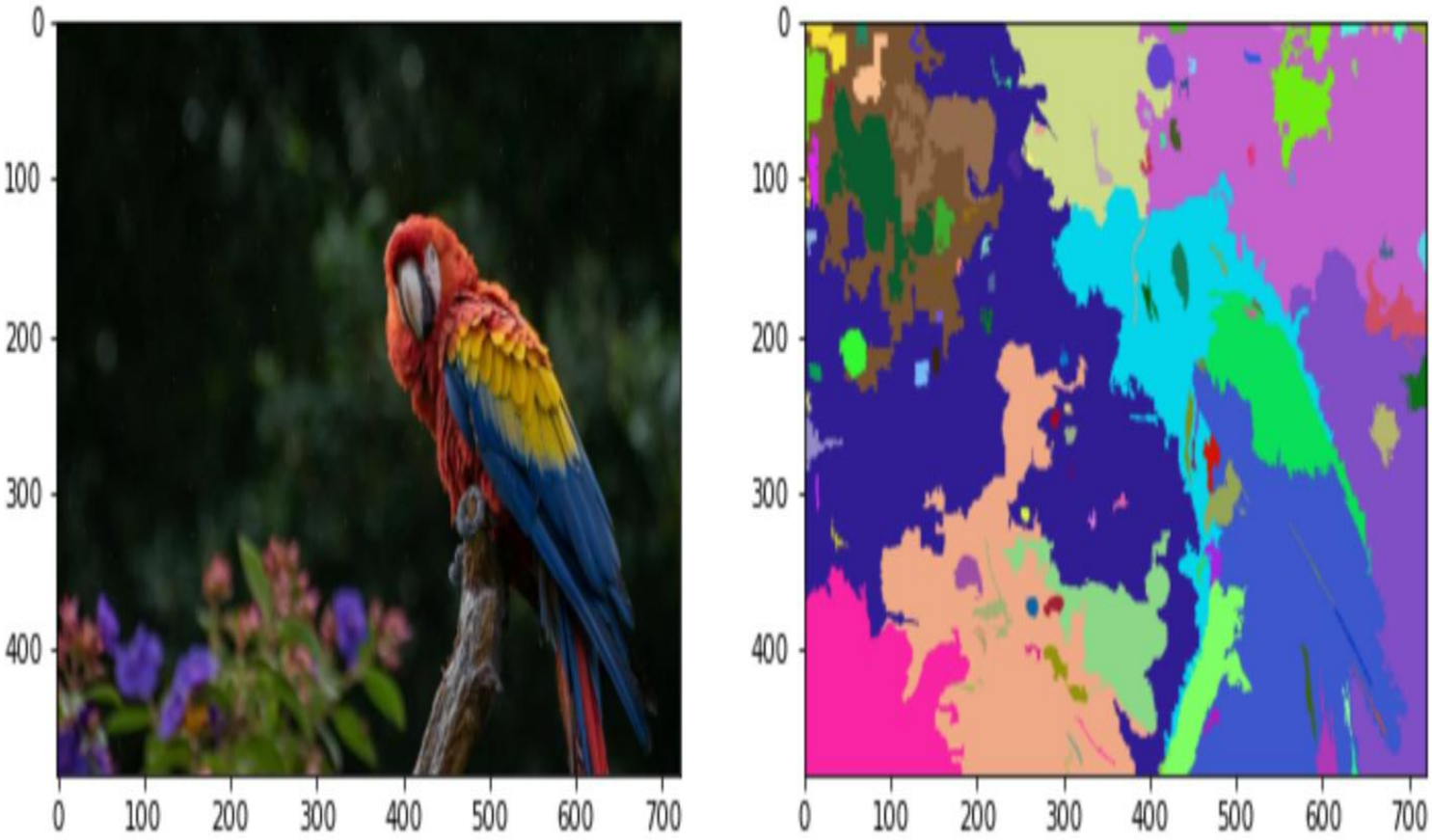


Figure 2-14. The Felzenszwalb segmentation algorithm realizes an image as a weighted undirected graph and then partitions the graph so that the variation across two different components is greater than the variation across either component individually

The sensitivity to the within group and between group differences is handled by a threshold function. The strictness of this threshold is determined by the parameter k. You can think of this k as setting the scale of observation for the segmentation algorithm. Figure 2-15 shows the resulting image segmentations for various values of k. These images were created using the 'felzenszwalb-segmentation' package in python and the code for this example is available in the book's GitHub repository. As you can see there, a larger value of k causes a preference for larger components while a smaller value of k allows for smaller regions. It's important to note that k does not guarantee a minimum component size. Smaller components would still be allowed, they just require a stronger difference between neighboring components.

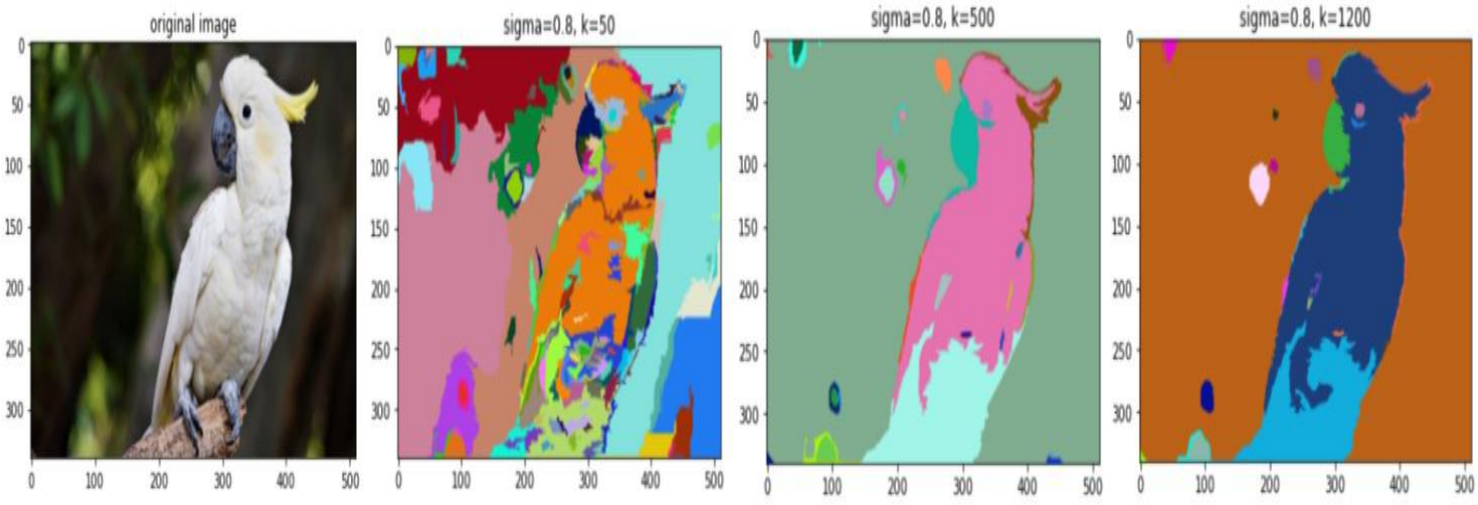


Figure 2-15. The parameter k controls the threshold function for Felzenszwalb segmentation algorithm. A larger value of k causes a preference for larger components while a smaller value of k allows for smaller regions.

How XRAI works

XRAI combines the output from applying integrated gradients to an input image with the image segmentation provided by Felzenswalb's algorithm described in the previous section. Intuitively, integrated gradients provides pixel-level attributions, so by aggregating these local feature attributions over the globally relevant segments produced from the image segmentation, XRAI is able to rank each segment and order the segments which contribute most to the given class prediction.

If one segment, for example the segment consisting of the body and crest of the cockatoo in Figure 2-14, contains a lot of pixels that are considered salient via integrated gradients, then that segment would also be ranked as highly salient via XRAI as well. However, a much smaller region, such as the segment representing the eye of the cockatoo, even if the pixels have high saliency measures according to the output from applying integrated gradients, XRAI would not rank that region as highly. This helps protect against individual pixels or small pixel segments that have spuriously high saliency from applying integrated gradients alone.

One thing to keep in mind, however, is the sensitivity of the segmentation result and how a certain choice of hyperparameters might bias the result. To address this, the image is segmented multiple times using different values [50, 100, 150, 250, 500, 1200] for the scale parameter k. In addition, segments smaller than 20 pixels are ignored entirely. Since for a single parameter the union of segments gives the entire image, the union of all segments obtained from all the parameters yields an area equal to about six times the total image area and with multiple segments overlapping. Each of the regions from this over-segmentation is used when aggregating the lower level attributions.

So, bringing it all together, when implementing XRAI, first the method of integrated gradients is applied using both a black and white baseline to determine pixel-level attributions, as shown in Figure 2-16. Concurrently, Felzenswalb's graph-based segmentation algorithm is applied multiple times using a range of scale parameter values for k = 50, 100, 150, 250, 1200. This produces an over-segmentation of the original image. XRAI then aggregates the pixel-level attributions by summing the values from the integrated gradient output within each of the resulting segments and ranks each segment from most to least salient. The result is a heatmap that highlights the areas of the original image that

contribute most strongly to the model's class prediction. Overlaying this heatmap over the original image, we can see which regions most strongly contribute to the prediction of 'sulfur-crested cockatoo'.

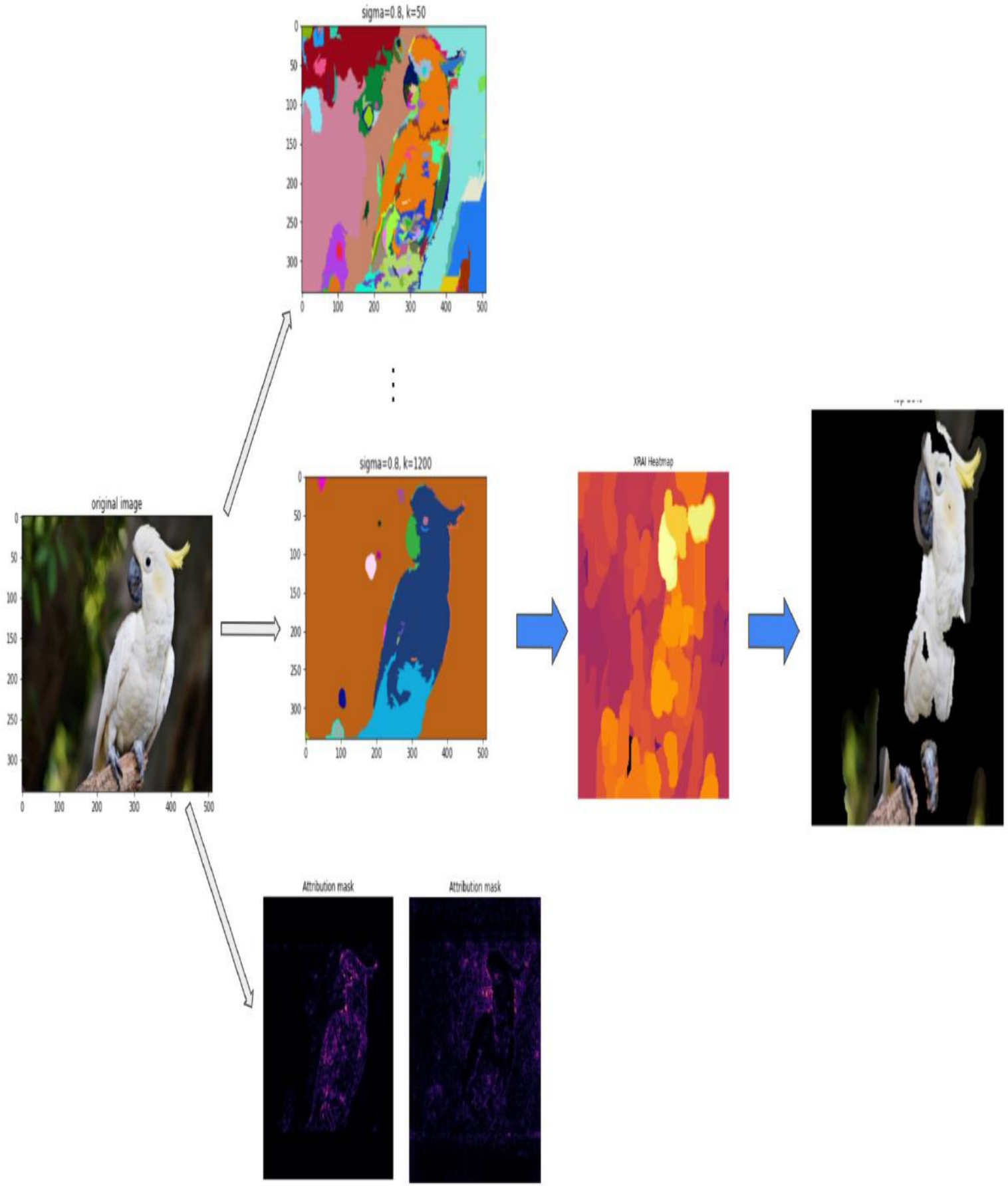


Figure 2-16. XRAI combines the results from the method of integrated gradients with the over-segmentation regions obtained from multiple applications of Felzenswalb's segmentation algorithm. The pixel-level attributions are then aggregated over the over-segmented regions and ranked to determine those regions that are most salient for the model.

Implementing XRAI

XRAI is a post-hoc explainability method and can be applied to any DNN based model. Let's look at an example of how to implement XRAI using the saliency library developed by the People and AI Research (PAIR) group at Google. In the following code block, we load a VGG-16 model pre-trained on the ImageNet dataset and modify the outputs so that we can capture the model prediction m.output as well as one of the convolution block layers block5_conv3. The full code for this example can be found in this notebook in this book's GitHub repository.

```
m = tf.keras.applications.vgg16.VGG16(weights='imagenet', include_top=True)
conv_layer = m.get_layer('block5_conv3')
model = tf.keras.models.Model([m.inputs], [conv_layer.output, m.output])
```

To get the XRAI attributions we construct a saliency object for XRAI and call the method GetMask passing in a few key arguments.

```
xrai_object = saliency.XRAI()
xrai_attributions = xrai_object.GetMask(image,
                                        call_model_function,
                                        call_model_args,
                                        batch_size=20)
```

The image argument is fairly self-explanatory, it's the image on which we want to obtain the XRAI attributions passed in as a numpy array. Let's discuss the other arguments: the call_model_function and call_model_args. The call_model_function is how we pass inputs to our model and receive the outputs necessary to compute saliency mask. It calls the model so it expects input images. Any args needed when calling and running the model are handled by call_model_args. The last argument expected_keys tells the function the list of keys expected in the output. We'll use the call_model_function defined in the following code and we'll either get back gradients with respect to the inputs or the gradients with respect to the intermediate convolution layer.

```
class_idx_str = 'class_idx_str'
def call_model_function(images, call_model_args=None, expected_keys=None):
    target_class_idx =  call_model_args[class_idx_str]
    images = tf.convert_to_tensor(images)
    with tf.GradientTape() as tape:
        if expected_keys==[saliency.base.INPUT_OUTPUT_GRADIENTS]:
            tape.watch(images)
            _, output_layer = model(images)
            output_layer = output_layer[:,target_class_idx]
            gradients = np.array(tape.gradient(output_layer, images))
            return {saliency.base.INPUT_OUTPUT_GRADIENTS: gradients}
        else:
            conv_layer, output_layer = model(images)
            gradients = np.array(tape.gradient(output_layer, conv_layer))
            return {saliency.base.CONVOLUTION_LAYER_VALUES: conv_layer,
                    saliency.base.CONVOLUTION_OUTPUT_GRADIENTS: gradients}
```

You may recall that when implementing integrated gradients, one hyperparameter you could adjust is the number of steps used to compute the line integral. Since XRAI relies on the output from integrated

gradients, you may be wondering where and how you adjust that hyperparameter for XRAI. In the saliency library those kinds of hyperparameters are controlled with a subclass called XRAIParameters. The default number of steps is set to 100. To change the number of steps to 200, you simply create an XRAIParameters object and pass it to the GetMask function as well:

```
xrai_params = saliency.XRAIParameters()
xrai_params.steps = 200

xrai_attributions_fast = xrai_object.GetMask(im,
                                             call_model_function,
                                             call_model_args,
                                             extra_parameters=xrai_params,
                                             batch_size=20)
```

Finally, we can plot the xrai_attributions object returned from calling GetMask to obtain a heatmap of attributions for the image, as shown in Figure 2-17.

Figure 2-17. XRAI produces a heatmap of attributions for a given input image. For this image of a sulfur-crested cockatoo the more relevant regions correspond to the body of the bird, its beak and distinct crest.

In order to see which regions of the original image were most salient, the following code focuses only on the most salient 30% and creates a mask filter overlaid on the original image. This results in Figure 2-18.

```
mask = xrai_attributions > np.percentile(xrai_attributions, 70)
im_mask = np.array(im_orig)
im_mask[~mask] = 0
ShowImage(im_mask, title='Top 30%', ax=P.subplot(ROWS, COLS, 3))
```

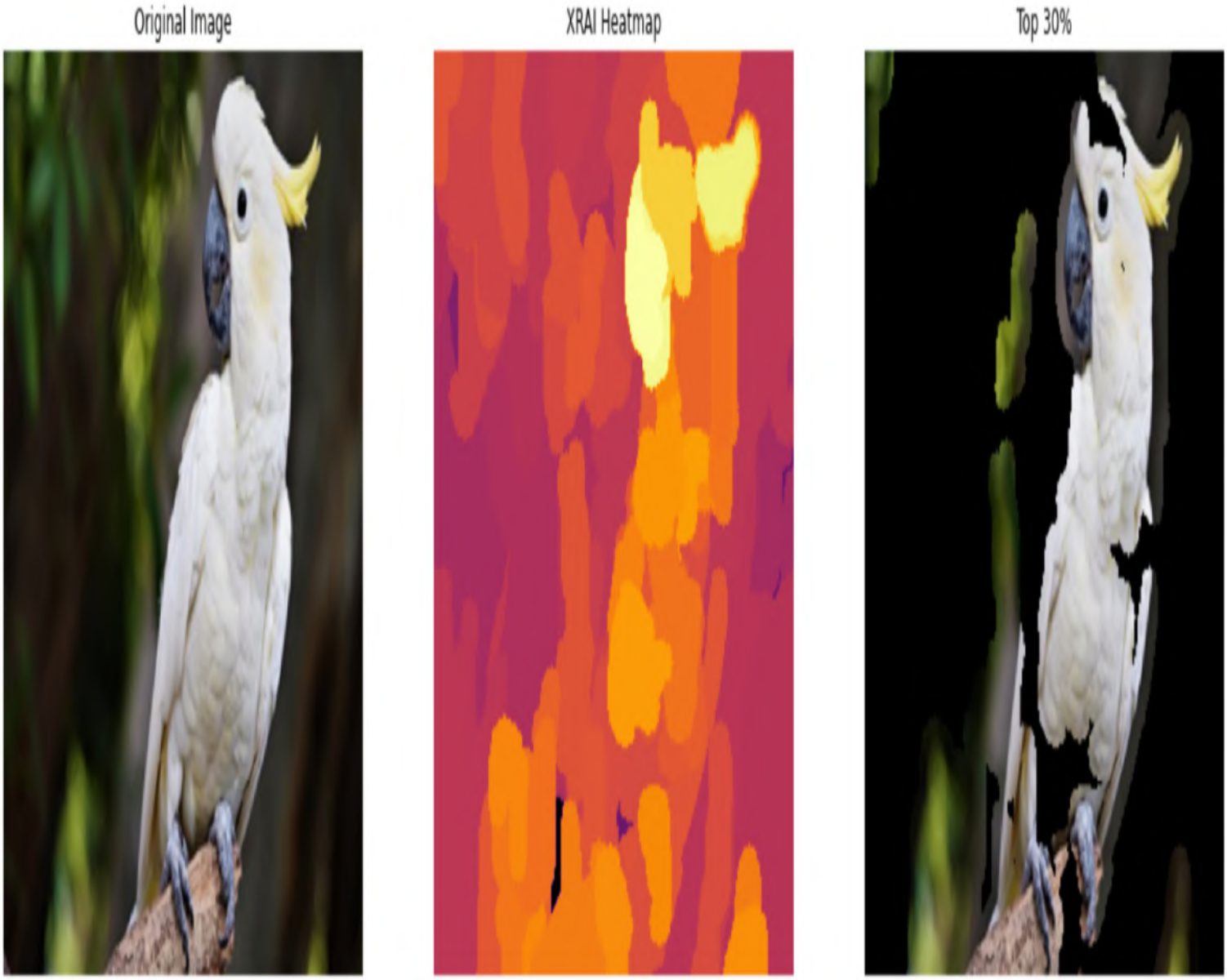


Figure 2-18. Filtering the XRAI attributions we can overlay the heatmap on the original image to highlight only the most salient regions.

Grad-CAM

Here's what you need to know about Grad-CAM

- Grad-CAM is short for Gradient-weighted Class Activation Mapping. Grad-CAM was one of the first explainability techniques; it generalizes CAM which could only be used for certain model architectures.

- Grad-CAM works by examining the gradient information flowing through the last (or any) convolution layer of the network
- Produces a localization heat map highlighting the regions in an image most influential for predicting a given class label

Pros	Cons
Only requires a forward pass of the model so it's computationally efficient and easy to implement by hand or with open source libraries	Grad-CAM can cause incorrect and misleading results producing heatmaps that attribute regions in the image that were not influential for the model. You should take caution when interpreting the results from Grad-CAM.
Applicable to a wide variety of CNN models and tasks (e.g. CNNs with fully connected layers as used in classification tasks, CNNs with multi-modal inputs as used in visual question answering, or CNNs for text outputs as used in image captioning.	Doesn't perform as well as other XAI methods for multi-class classification models with a large number of class labels
Can be combined with other pixel-space visualizations to create high-resolution class discriminative visualizations (see the section on Guided Backpropagation and Guided Grad-CAM)	Fails to properly localize objects in an image if the image contains multiple occurrences of the same class; e.g, an image classifier for recognizing cat that is given an image with two cats will produce an unreliable heat map
	The low resolution heatmaps produced with Grad-CAM can be misleading to users.

Grad-CAM, short for Gradient-weighted Class Activation Mapping, is one of the original explainability techniques developed for image models and can be applied to any CNN-based network. Grad-CAM is a post-hoc explanation technique and doesn't require any architecture changes or retraining. Instead, Grad-CAM accesses the internal convolutions layers of the model to determine regions that are most influential for the model's predictions. Because Grad-CAM only relies on forward passes through the model, with no backpropagation, it is also computationally efficient.

How Grad-CAM works

To understand how Grad-CAM works, let's first start with what Class Activation Map (CAM) is and how it works, since Grad-CAM is essentially a generalization of CAM. Typically, when building an image model, your model architecture is likely to consist of a stack of convolutional and pooling layers. For example, think about the classic VGG-16 model architecture shown in Figure 2-19. There are five convolution plus pooling blocks that act as feature extractors, followed by three fully connected layers before the final softmax prediction layer. CAM is a localization map of the image that is computed as a weighted activation map. This is done by taking a weighted sum of the activation maps of the final convolution layer in the model.

More precisely, and to illustrate with an example, suppose we take the final convolution layer (just before the last pooling layer) of the VGG-16 model in Figure 2-19 trained on the ImageNet dataset. The dimension of this final convolution layer is 7x7x512. So there are 512 feature maps mapping to 1,000 class labels; i.e., the labels corresponding to ImageNet.

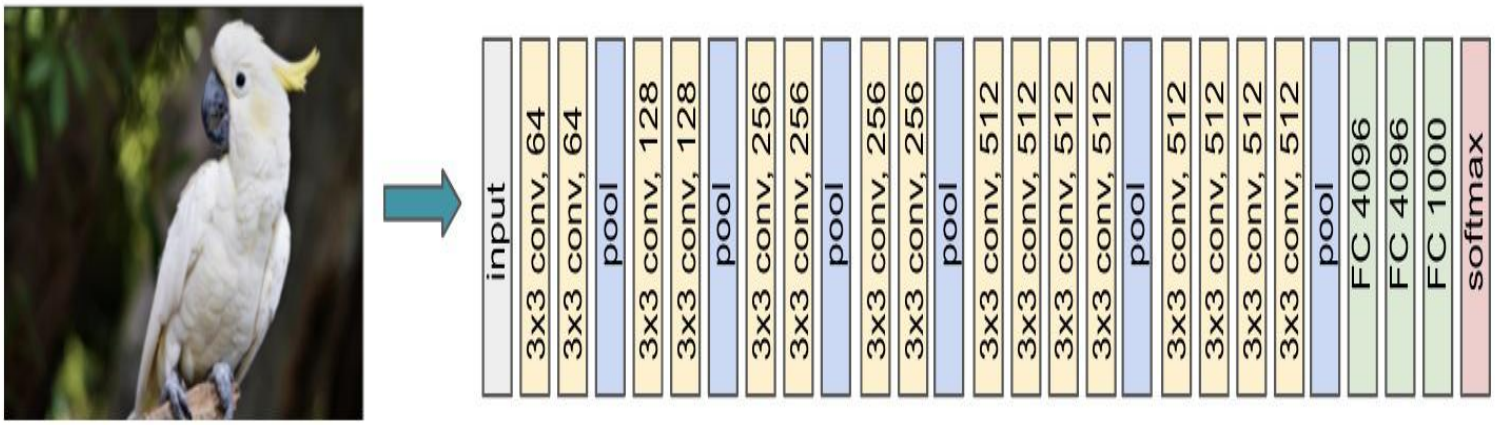


Figure 2-19. VGG-16 consists of blocks of convolution and pooling layers. CAM is a weighted activation map applied to the final convolution layer

Each feature map is indexed by i and j representing the width and height of the convolution layer (in this case $i = j = 7$). Notationally, A_{ij}^{k} denotes the activation at location (i, j) of the feature map A^{k}. Those feature maps are then pooled using global average pooling and CAM computes a final score Y^{c} for each class c by taking a weighted sum of the pooled feature maps, where the weights are given by the weights connecting the k-th feature map with the c-th class:

$$Y^{c} = \sum_{k} w_{k}^{c} \underbrace{\frac{1}{Z} \sum_{i,j} A_{ij}^{k}}_{\text{global average pooling of feature maps}}$$

weight connecting the k^{th} feature map with the c^{th} class

If you then take the gradient of Y^{c}; i.e., the score for class c, with respect to the feature maps, you can separate out the weight term w_{k}^{c} so that (after a bit of math[2]) you get

$$Y^{c} = \sum_{k} w_{k}^{c} \sum_{i,j} A_{ij}^{k}$$

This weight value is almost exactly the same as the neuron importance weight that is used in Grad-CAM! For Grad-CAM, you also pull out the activations A^{k} from the final convolution layer and compute the gradient of the class score Y^{c} with respect to these A^{k}. In some sense, these gradients

capture the information flowing through the last convolution layer and you want to use this to assign importance values to each neuron for a particular class c. This importance weight is computed (similarly to CAM) as

$$\alpha_k^c = \frac{1}{Z} \sum_{i,j} \frac{\partial Y^c}{\partial A_{ij}^k}$$

The last step of Grad-CAM is to then take a weighted combination of the activation maps using these α_k^c as weights and apply a ReLU to that linear combination, as shown in Figure 4-#. So,

$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right)$$

You use a ReLU here because we only care about the pixels whose intensity should be increased in order to increase the class score Y^c. Remember ReLU maps negative values to zero so by applying ReLU after the weighted sum we only capture the positive influence on the class score. Because the shape of A^k is the same shape as the final convolution layer, this produces a coarse heatmap that can then be overlaid on the original image to indicate which regions were most influential in the model predicting class c. Also due to the averaging and pooling of these feature maps, Grad-CAM works best at representing regions of influence in the image rather than exact pixels. An overview of the Grad-CAM process is shown in Figure 2-20. An overview of the Grad-CAM process is shown in Figure 4-#.

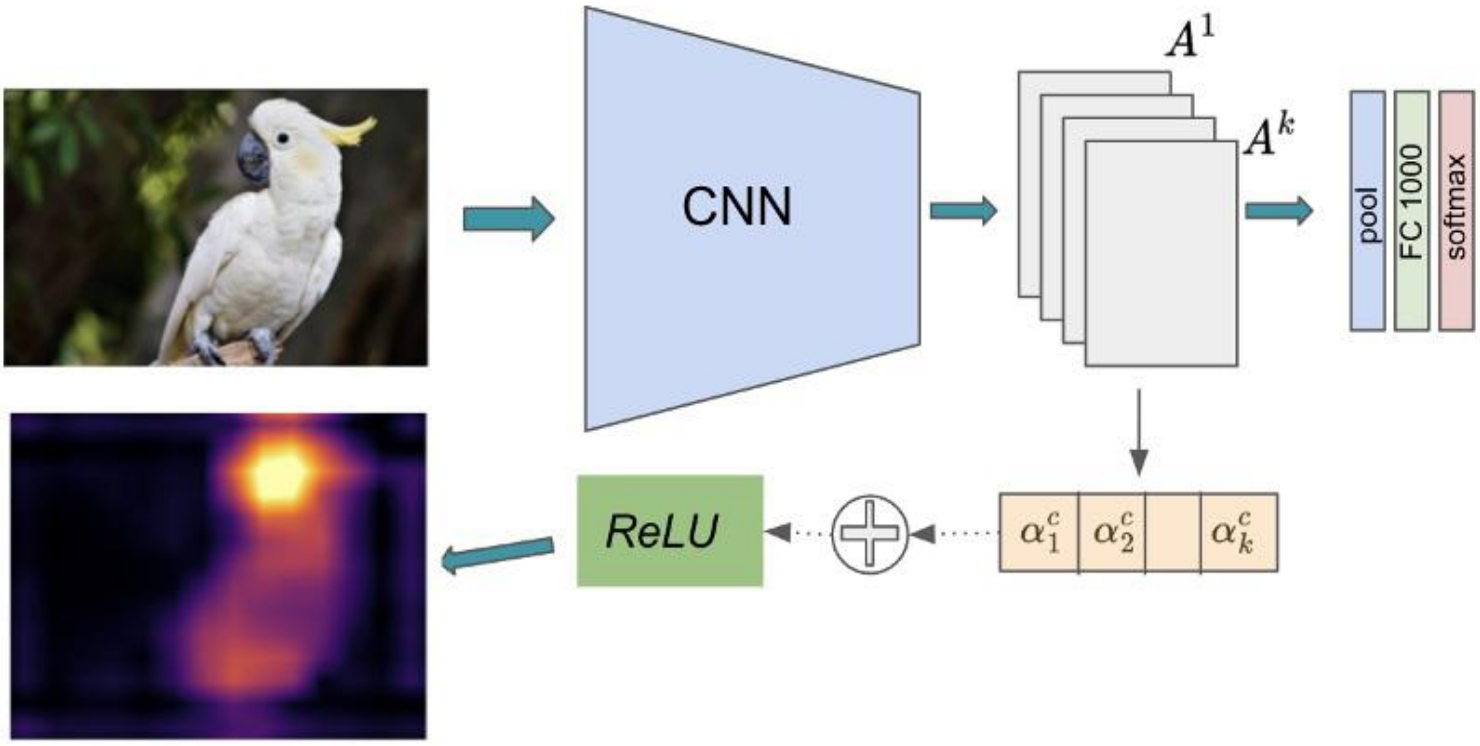


Figure 2-20. An overview of Grad-CAM. The final activation maps A^k are used to compute a weighted sum which is then passed through a ReLU layer.

WARNING

The coarse heatmap created by Grad-CAM is problematic and can generate misleading results. Because the heatmap has the same dimensions as the final convolutional feature maps A^k it is usually upsampled to fit the shape of the original image. For example, the last convolutional layer of VGG-16 is 7x7x512, so the heatmap determined from the weighted sum of the activation maps will have dimension 7x7 which then has to be upsampled to overlay on the original image which has shape 224x224. This upsampling has the strong potential for causing misleading results, particularly if your dataset consists of large images whose shape is much larger than the internal activation map or contains image patterns and artifacts that are not properly sampled by a convolution.

You may be wondering how exactly Grad-CAM improves upon CAM, especially if they are essentially using the same importance weights. The advantage of Grad-CAM is that the construction for CAM is very limiting. It requires feature maps to directly precede softmax layers. So, CAM only works for certain kinds of CNN model architectures; i.e., ones that have their final layers to be a convolution feature mapped to a global average pooling layer mapped to a softmax prediction layer. The problem is that this may not be architecture with best performance. By taking a gradient and rearranging terms, Grad-CAM is able to generalize CAM and works just as well for a wide range of architectures for other image based tasks, like image captioning or visual question answering.

NOTE

In the description of Grad-CAM we only discussed taking the CAM from the final convolution layer. As you may have already guessed, the technique we described is quite general and can be used to explain activations for any layer of the deep neural network.

Implementing Grad-CAM

The Grad-CAM algorithm outlined in Figure 2-20 is fairly straightforward and can be implemented directly by hand, assuming you have access to the internal layers of your CNN model. However, there is an easy to use implementation available from the saliency library. Let's see how it works with an example. You start by creating a TensorFlow model. In the following code block, we load a pre-trained VGG-16 model architecture that has been trained on the ImageNet dataset. We also select the penultimate convolution layer 'block5_3' that we'll use to obtain the activation maps. Note that when we build the actual model it returns both the convolution layer output and the VGG-16 outputs so we can still make predictions. For the full code for this example, see the Grad-CAM notebook in the GitHub repository for the book.

```
vgg16 = tf.keras.applications.vgg16.VGG16(
    weights='imagenet', include_top=True)
conv_layer = m.get_layer('block5_conv3')
model = tf.keras.models.Model(
    [vgg16.inputs], [conv_layer.output, vgg16.output])
```

To apply Grad-CAM, you then construct a saliency object, calling the Grad-CAM method and then apply GetMask passing the example image, and the call_model_function. The following code block shows how this is done. The call_model_function that interfaces with a model to return the convolution

layer information and the gradients of the model. This returns a Grad-CAM mask that can be used to plot a heat map indicating influential regions in the image.

```
# Construct the saliency object. This alone doesn't do anything.
grad_cam = saliency.GradCam()
# Compute the Grad-CAM mask.
grad_cam_mask_3d = grad_cam.GetMask(im, call_model_function,
call_model_args)
```

> **WARNING**
>
> Historically, Grad-CAM is an important technique. It was one of the first techniques to leverage the internal convolution layers of a CNN model to derive explanations for model prediction. However, you should take caution when implementing and interpreting the results from Grad-CAM. Because of the upsampling and smoothing step from the convolution layer, some regions of the heatmap may seem important when in fact they were not influential for the model at all. There have been improvements to address these concerns (see the next subsection on Improving Grad-CAM and the later section on Guided Grad-CAM) but the results should be viewed critically and in comparison with other techniques.

Improving Grad-CAM

After publication of the original paper in 2017[3], it was discovered that Grad-CAM could inaccurately attribute model attention to regions of the image which were not actually influential in the prediction. Grad-CAM applies a natural smoothing, or interpolation, to its original heatmaps in enlarging them to the size of the original dimensions of the input image. Figure 2-21 shows an example of what the Grad-CAM output looks like with and without smoothing.

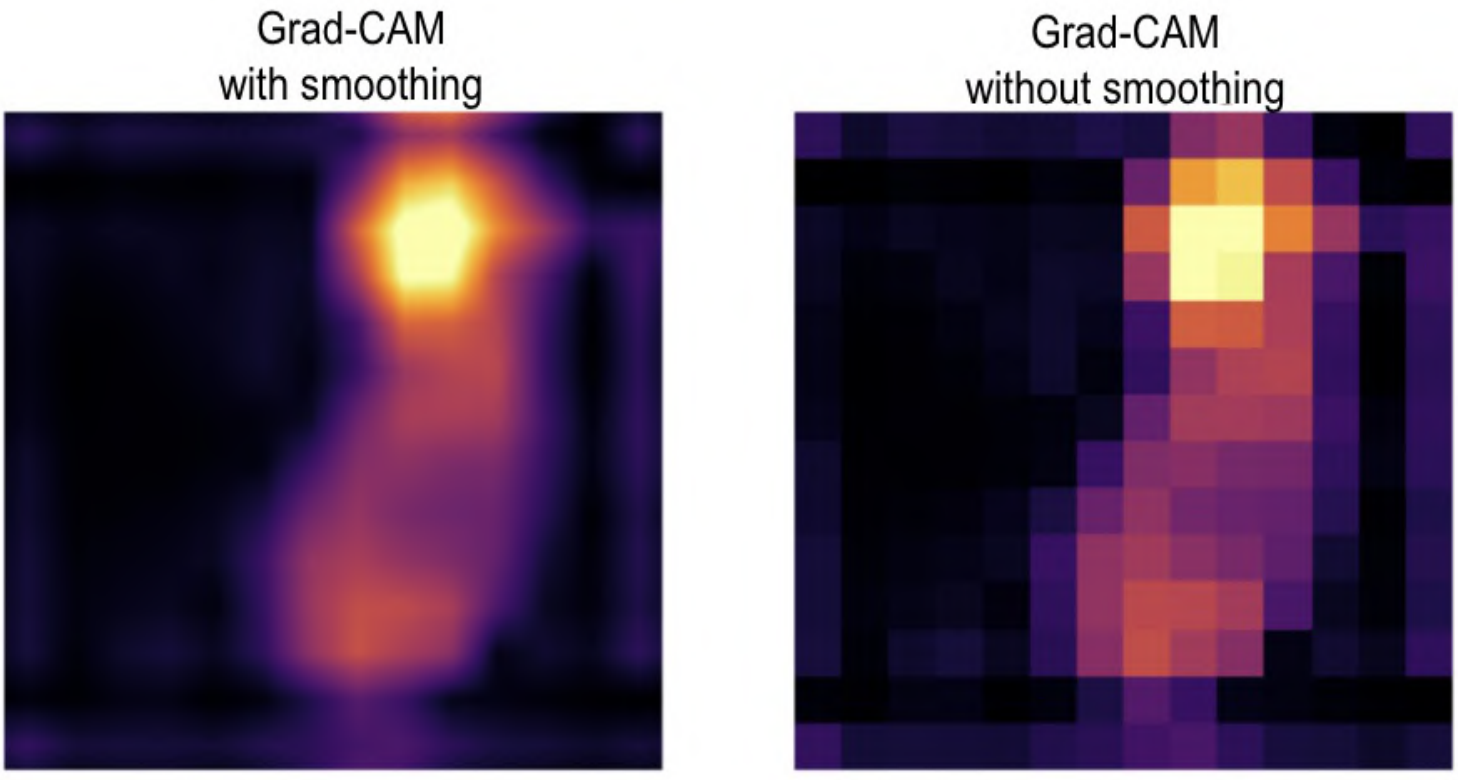


Figure 2-21. Grad-CAM up-samples from the feature map size to the size of the original image and applies a natural smoothing. This can cause misleading results and inaccurately attribute model attention to regions that weren't influential.

The smoothing results in a more visually pleasing visual, with seamless changes in the colors of the

heatmap, but can also be very misleading. Since the original region was focused on a much smaller size image, we can't say that the pixels which were lost would have had the same influence on the model as those remaining around it.[4] In response, researchers developed Grad-CAM++ and HiResCAM to address these issues. Figure 2-22 shows an example of how the outputs of Grad-CAM and Grad-CAM++ differ. These techniques helped to address these concerns and can produce more accurate attribution mapsHowever, even with the regional inaccuracy fixed, Grad-CAM remains a flawed technique due to how it up-samples from the original size of the feature maps to the size of the input image.

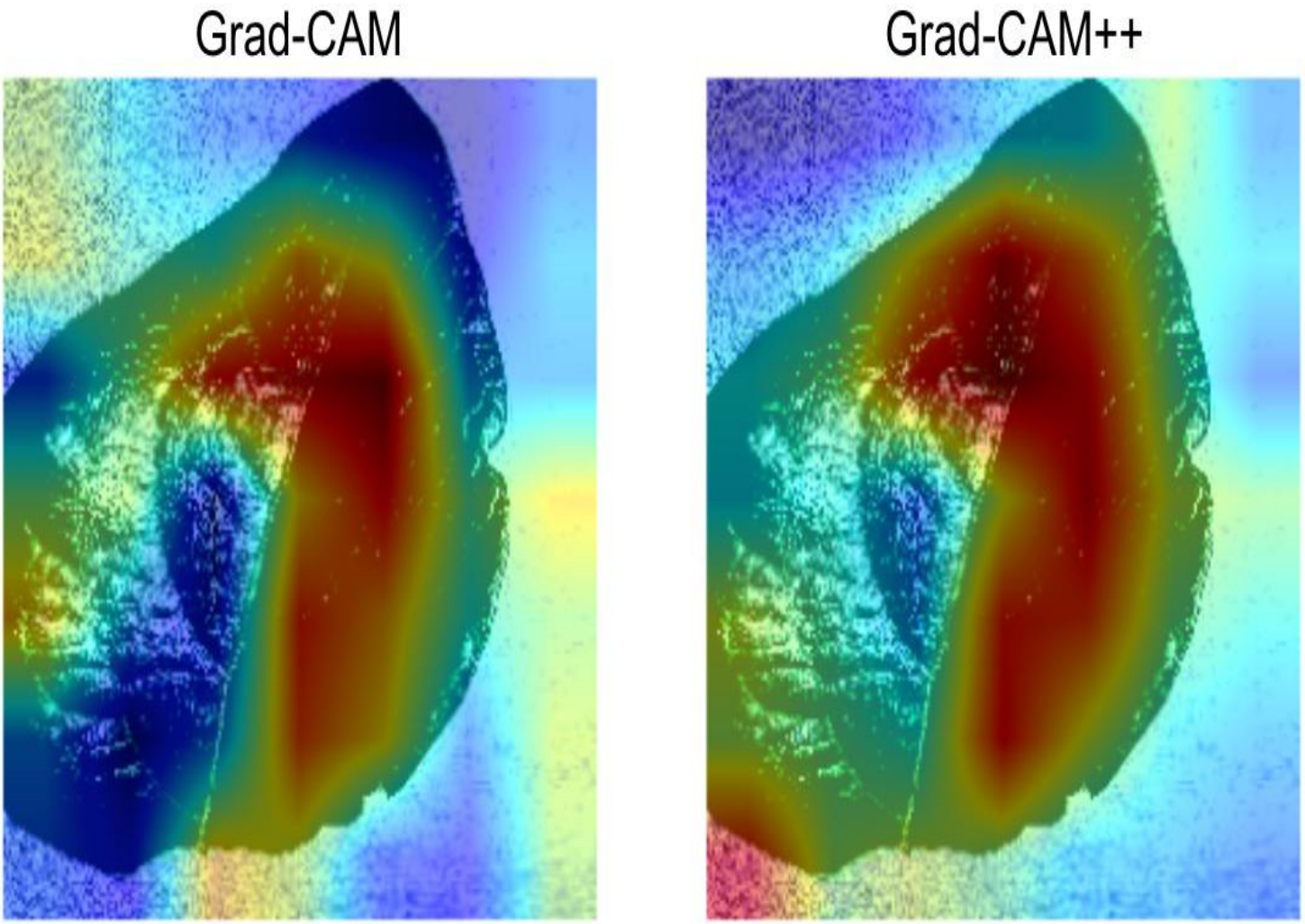


Figure 2-22. An example of a heatmap from applying Grad-CAM and Grad-CAM++ for the classification of a healthy blueberry leaf. In the Grad-CAM image the heatmap is most attentive in the center of the leaf, but also highlights some influence from the background. For Grad-CAM++ the activated pixels are more evenly distributed throughout the leaf, although a highly influential region is still attributed to the background in the lower left of the image.

LIME

Here's what you need to know about LIME:

- LIME stands for Locally Interpretable Model-Agnostic Explanations. It is a post-hoc, perturbation based explainability technique.
- Can be used for regression or classification models, though in practice LIME is used primarily for classification models

- Works by changing regions of image turning them “on” or “off” and re-running inferences to see which regions are most influential to the model’s prediction
- Uses an inherently interpretable model to measure influence or importance of input features

Pros	Cons
LIME is a very popular technique for producing pixel-level attributions.	Explanations are brittle, and may not be accurate. Because LIME is a local interpretation of a complex model, explanations can be particularly bad for highly nonlinear models.
It is easy to implement, and has a well-maintained python implementation (see GitHub repo).	Prone to mis-identifying background regions as influential
It has an intuitive and easy to explain algorithm and implementation	Explanations can be very slow to generate, depending on the complexity of your model since it queries the model multiple times on perturbations of the input
There are a wide variety of visualization options.	

LIME stands for Locally Interpretable Model-Agnostic Explanations and, due to its age, is one of the more popular explainability techniques. It can also be used for tabular and text data and the algorithm is similar for each of these data modalities albeit with some modifications. We discuss the details of implementing LIME with images here because it gives a nice visual intuition of how LIME works in general (see Chapter 5 for a discussion of how LIME works with text).

How LIME Works

LIME is a post-hoc, model agnostic perturbation-based explainability technique. That means it can be applied to any machine learning model (e.g. neural networks, SVM, Random Forest, etc) and is applied after the model has been trained. In essence, LIME treats the trained model like an API, taking an example instance and producing a prediction value. To explain why the model makes a certain prediction for a given input instance, the LIME algorithm works by passing lots and lots of slightly perturbed examples of the original input to the model and then measures how the model predictions change with these small input changes. The perturbations occur at the feature level of the input instance; i.e., for images, pixels and pixel regions are modified to create a new perturbed input. In this way, those pixels or pixel regions that most influence the model’s prediction are highlighted as being more or less influential to the model’s predicted output for the given input instance.

To go into a little more detail, let’s further explain two of the key components of implementing LIME: first, how to generate a perturbation of an image and second, what it means to measure how the model prediction changes on these perturbations.

For a given prediction, the input image is subdivided into interpretable components, or regions, of the image. LIME segments the image into regions called superpixels. A superpixel is a similarity based grouping of the individual pixels of an image into similar components (see the discussion of Felzenswalb’s algorithm in the section on XRAI). For example, Figure 2-23 shows how the image of the sulpher-crested cockatoo can be segmented into superpixels. The superpixel regions represent the interpretable components of the image.

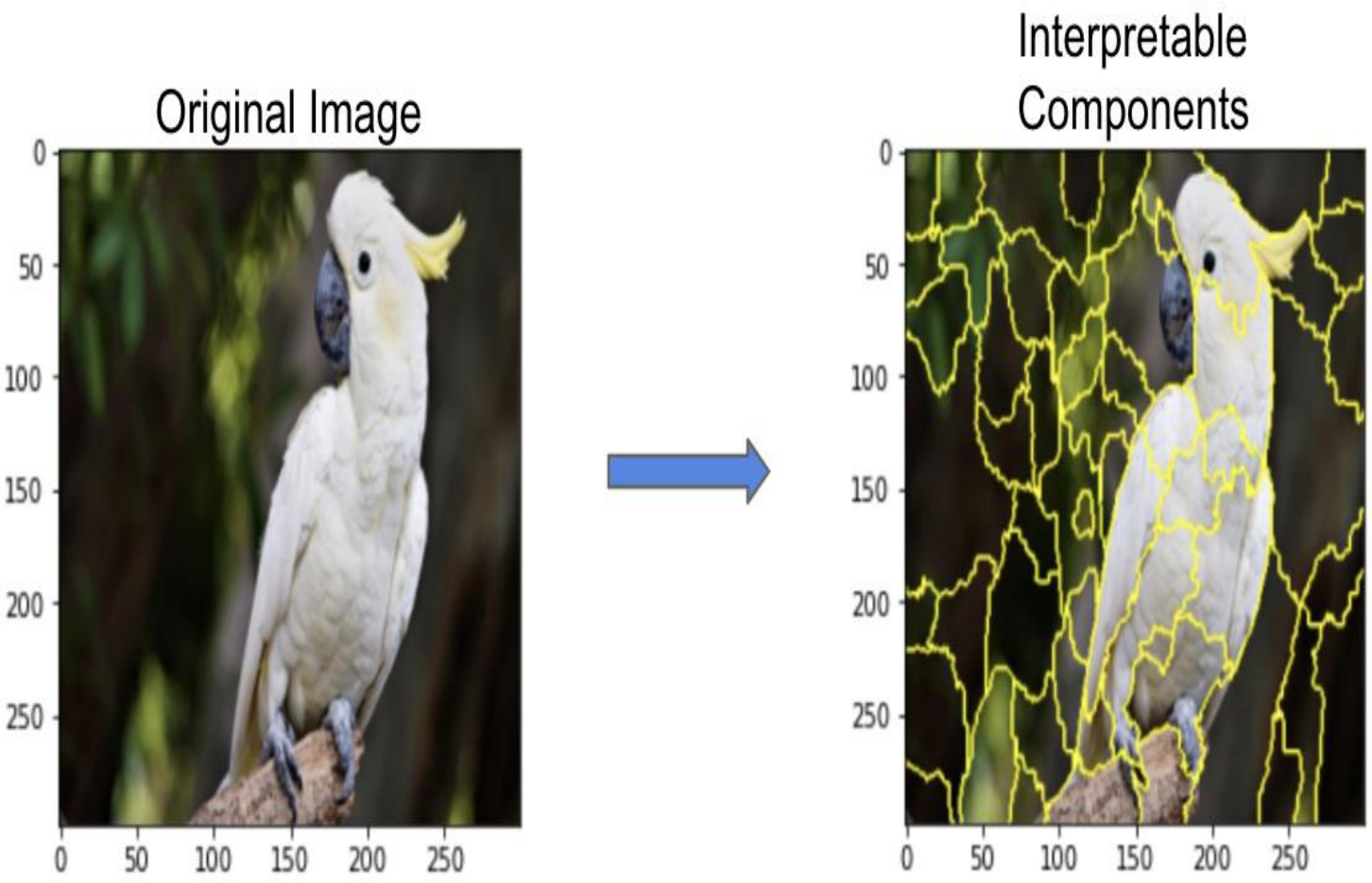


Figure 2-23. When implementing LIME, the original image is segmented into superpixel regions which represent the interpretable components of the image.

The superpixel regions in Figure 2-23 were created using the quickshift segmentation algorithm from the scikit-image library. This is the default segmentation algorithm used in the widely used, open source python package lime (see the next section for details on implementing LIME with the lime package) though other segmentation functions could be used. Quickshift is based on an approximation of kernelized mean-shift and computes hierarchical segmentation at multiple scales simultaneously. Quickshift has two main parameters: the parameter sigma determines the scale of the local density approximation, and max_dist determines the level of the hierarchical segmentation. There is also a parameter ratio that controls the ratio between the distance in color-space and the distance in image-space when comparing the similarity of two pixels. The following code produces the segmentation in Figure 2-23. See the LIME notebook in the GitHub repository for this book for the full code example.

```
from skimage.segmentation import quickshift
segments = quickshift(im_orig, kernel_size=4,
                      max_dist=200, ratio=0.2)
```

LIME then perturbs these interpretable components by changing the values of the pixels in each superpixel region to be gray. This creates multiple new variations, or perturbations, of the input image. Each of these new perturbed instances is then given to the model to generate new prediction values for the class that was originally predicted. For example, in Figure 2-24 the original image is modified by graying out certain superpixel regions. Those perturbed examples are then passed to the trained model (in this case an deep neural network) and the model returns the probability that the image contains a

sulfur-crested cockatoo. These new predictions create a dataset which is used to train LIME's linear model to determine how much each interpretable component's contribution to the original prediction.

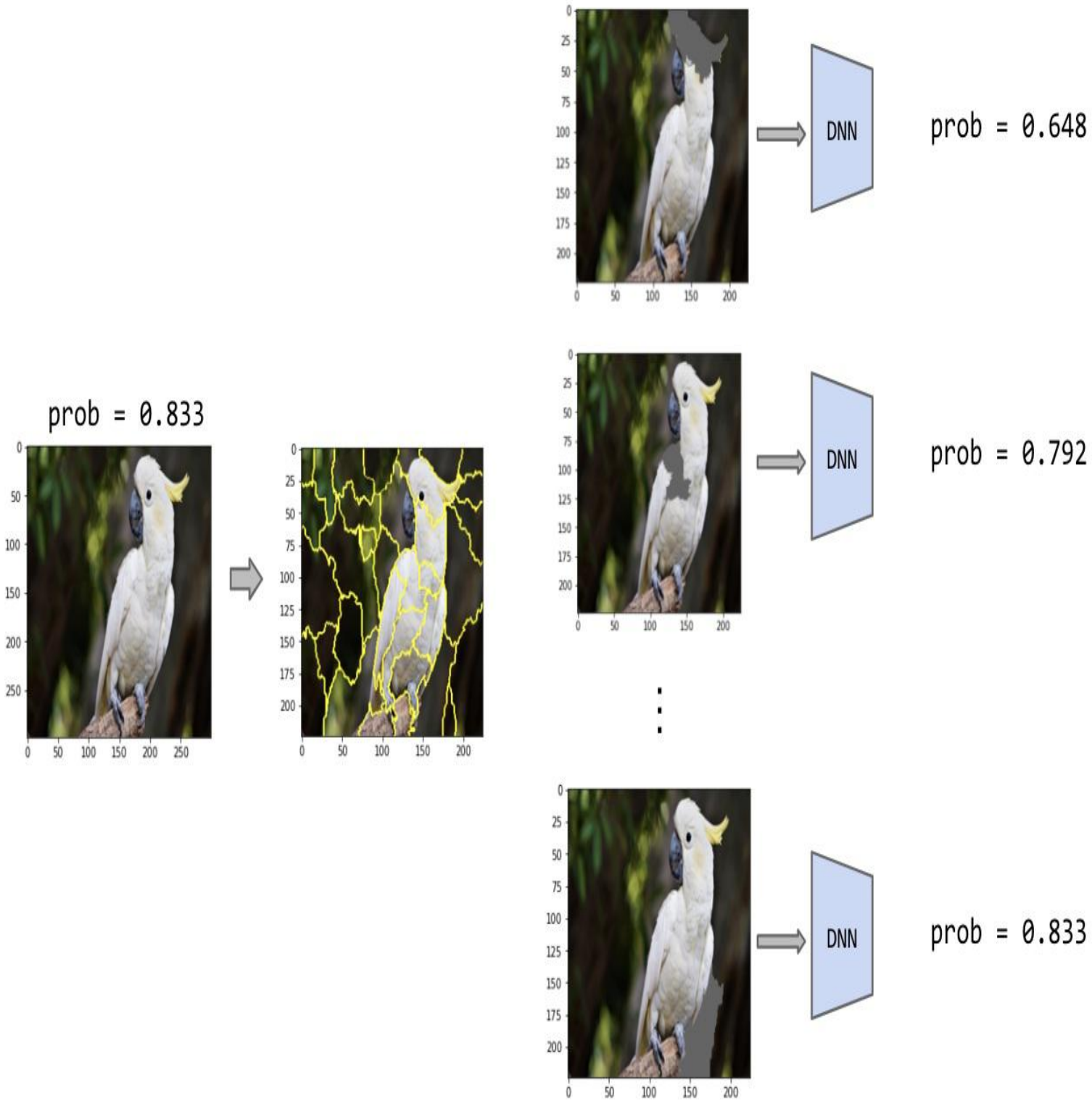


Figure 2-24. The model's confidence in the predicted class 'sulfur-crested cockatoo' for the original image is 0.833. As certain superpixels are removed (by graying out the region), the model's top class prediction changes. For regions that are influential, the change is larger than for regions that are less important.

> **NOTE**
>
> In the LIME implementation, superpixels are turned "on" or "off" by changing the pixel values of a segment to gray. This creates a collection of perturbed images that are passed to the model for prediction. It is also possible to instead change the superpixel regions to the mean of the individual pixel values in the superpixel region, as shown in Figure 2-25.

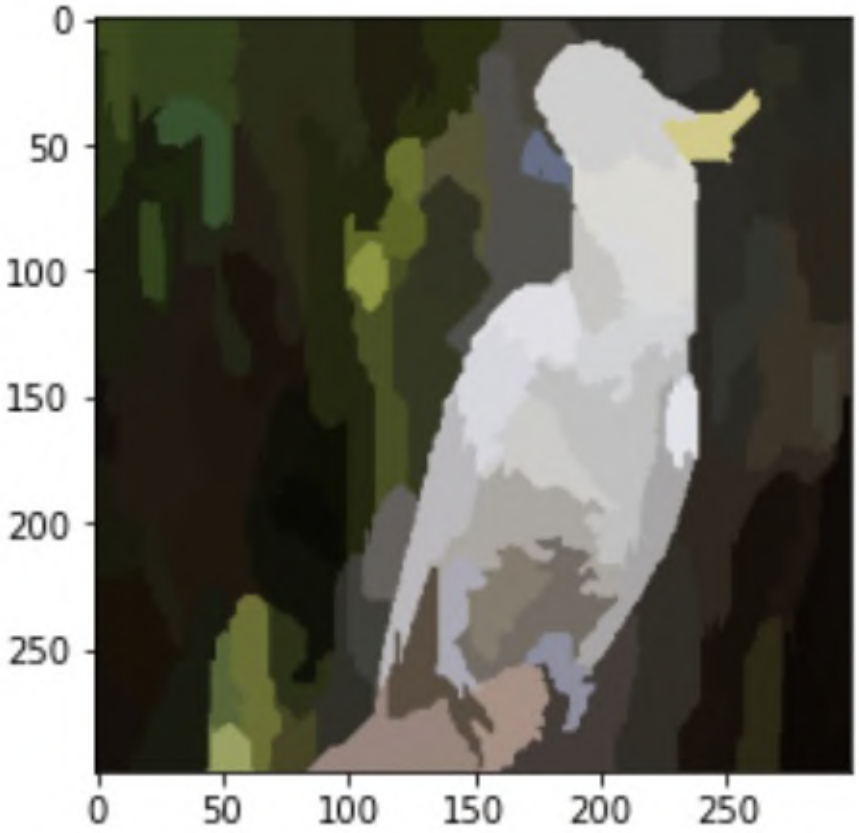


Figure 2-25. It is also possible to perturb images by setting the values of superpixel regions to be the average of the individual pixel regions in the image.

Let's now discuss how LIME quantitatively measures the contributions of the superpixel feature inputs. This is done by training a smaller, interpretable model to provide explanations for the original model. This smaller model can be anything, but the important aspect is that it is inherently interpretable. In the original paper, the authors use the example of a linear model. Linear models are inherently interpretable because the weights of each feature directly indicate the importance of that feature for the model's prediction.

This interpretable, linear model is then trained on a dataset consisting of the perturbed examples and the original model's predictions, as shown in Figure 4-#. These perturbations are really just simple binary representations of the image indicating the 'presence' or 'absence' of each superpixel region. Since we care about the perturbations that are closest to the original image, those examples with most superpixels present are weighted more than examples that have more superpixels absent. This proximity can be measured using any distance metric. For images. The idea is that even though your trained model might be highly complex, for example a deep neural network or an SVM, the locally weighted, linear model can capture the local behavior at a given input instance. The more sensitive the complex model is to a given input feature, the more of an influence that feature will have on the linear, interpretable model as well.

Implementing LIME

There is a nice, easy to use python package for applying LIME to tabular, image and text classifiers which works for any classifier with at least two label classes. As an example, let's see how to implement LIME on the Inceptionv3 image classification model. The following code shows how to load the image in TensorFlow and make a prediction on an image. Note that when we load the Incepton

model we specify include_top=True to get the final prediction layer of the model and we set weights='imagenet' to get the weights pre-trained on the ImageNet dataset.

```
inception = tf.keras.applications.InceptionV3(
    include_top=True, weights='imagenet')
model = tf.keras.models.Model(inception.inputs, inception.output)
```

Now, given an image, we can create explanations for the Inception model's prediction by creating a LimeImageExplainer object and then calling the explain_instance method. The following code block shows how to do this.

```
explainer = lime_image.LimeImageExplainer()
explanation = explainer.explain_instance(image.astype('double'),
                                         inception.predict,
                                         top_labels=20,
                                         hide_color=0,
     num_features=5)
```

Most of the parameters in this code block are self-explanatory. The last two, however, are worth mentioning. Firstly, the parameter hide_color indicates that we'll turn off superpixel regions by replacing them with gray pixels. The parameter, num_features, indicates how many features to use in the explanation. Fewer features lead to more simple, understandable explanations. However, for more complex models it may be necessary to keep this value large (the default is num_features=100000.

To visualize the explanations for this example, we then call get_image_and_mask on the resulting explanation. This is shown in the following code block, the result is shown in Figure 2-26. See the LIME notebook in the book repository for the full code for this example.

```
temp, mask = explanation.get_image_and_mask(explanation.top_labels[0],
                                            positive_only=True,
                                            num_features=20,
                                            hide_rest=True)
plt.imshow(mark_boundaries(temp, mask))
plt.show()
```

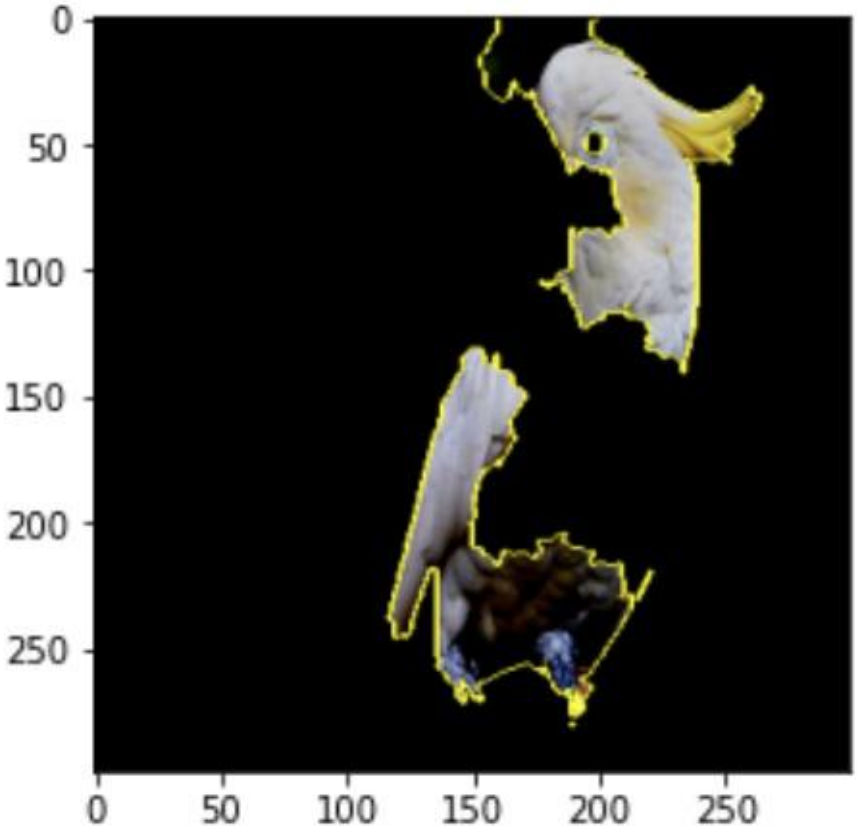


Figure 2-26. LIME uses a linear model trained on perturbations of the original input image to determine which superpixel regions were most influential for the complex model's prediction.

In Figure 2-26, we can see the superpixel regions that most positively contributed to the prediction 'sulfur-crested cockatoo'. It is also possible to see which superpixel regions provide a negative contribution as well. To do this, set positive_only to False. This produces the output image as shown in Figure 2-27. The regions that positively contribute to the prediction are in green, while the regions that negatively influence the prediction are in red.

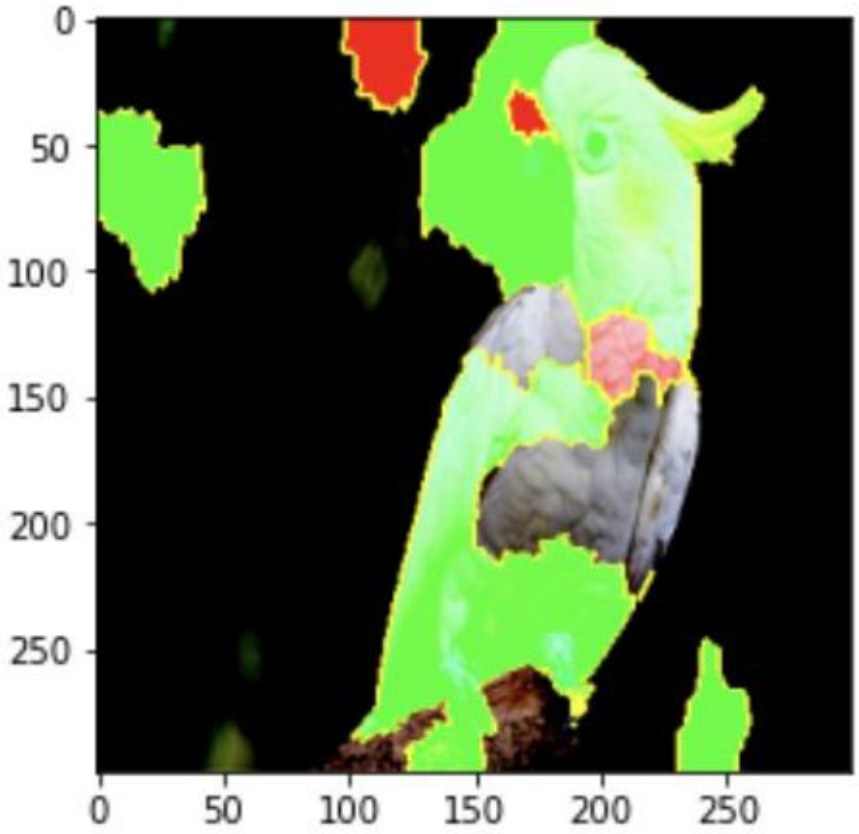


Figure 2-27. A LIME Explanation. Regions that influenced the model are shown as a heat map, with more influential regions having a deeper color, where green contributed positively to the classification, and red contributed negatively.

This results in a segmented map of the image which is weighted according to how strongly that region influenced the model's prediction. Unlike Shapley Values, there are no bounds or guarantees on the values of the region. For example, LIME could conceivably weight multiple regions as highly influential.

Unfortunately, LIME's elegance in generating weightings by turning regions of the image on and off is also its downfall. Since the perturbed images remove entire areas from the prediction, it effectively is shifting the model's attention to what remains visible in the image. In effect, LIME is asking the model not to do a "limited" prediction of the original image, but to look at an entirely new image. However, by not holistically perturbing the entire image, like Integrated Gradients, it is possible that learned concepts in the model are no longer activated. In effect, we are asking the model to tell us how it would predict the image by looking at how predictions change only with respect to one feature value and assuming that comparing across different features will result in something akin to how the model made its original prediction. This can sometimes lead to nonsensical explanations, as shown in Figure 2-28.

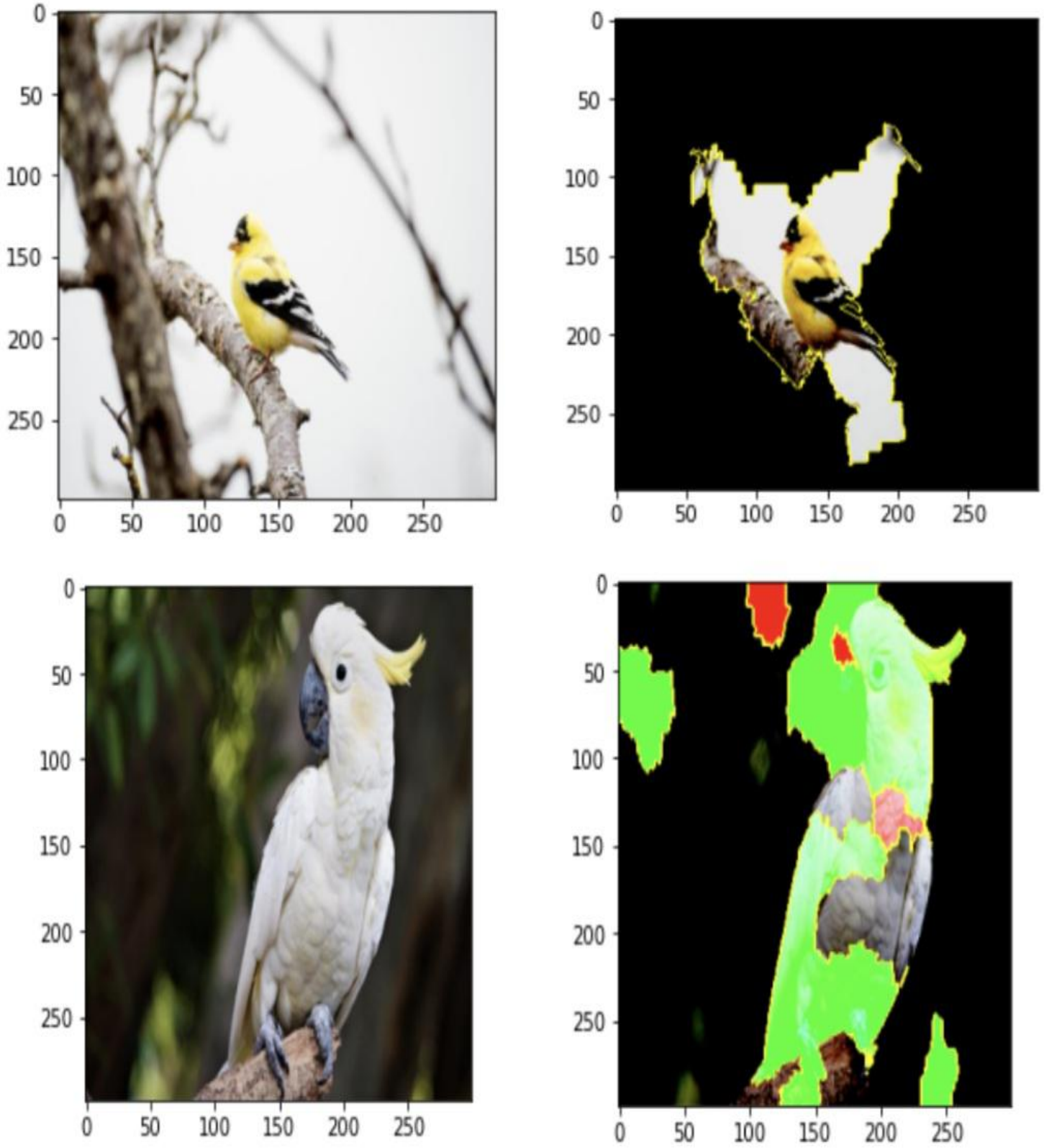


Figure 2-28. An example of how LIME's perturbation approach can lead to what seem to be nonsensical explanations. In the image of the goldfinch, there is a large portion of the background (nearly as much as the bird itself) contributing to the prediction. In the image of the cockatoo, we also see the background having positive influence on the model, while parts of the bird have negative influence as well.

As a further example of this, we used LIME on a non-natural dataset, the PlantLeaf Village dataset, which contains images of plant leaves with disease and tries to predict the type of disease. These images feature a combination of a very natural image (plant leaves) within a much more structured environment (interior lighting, monochrome background). The model used, Schuler, was highly accurate, with an overall prediction accuracy of 99.8%. We also compared the LIME explanations to another, slightly less

accurate model, Yosuke.

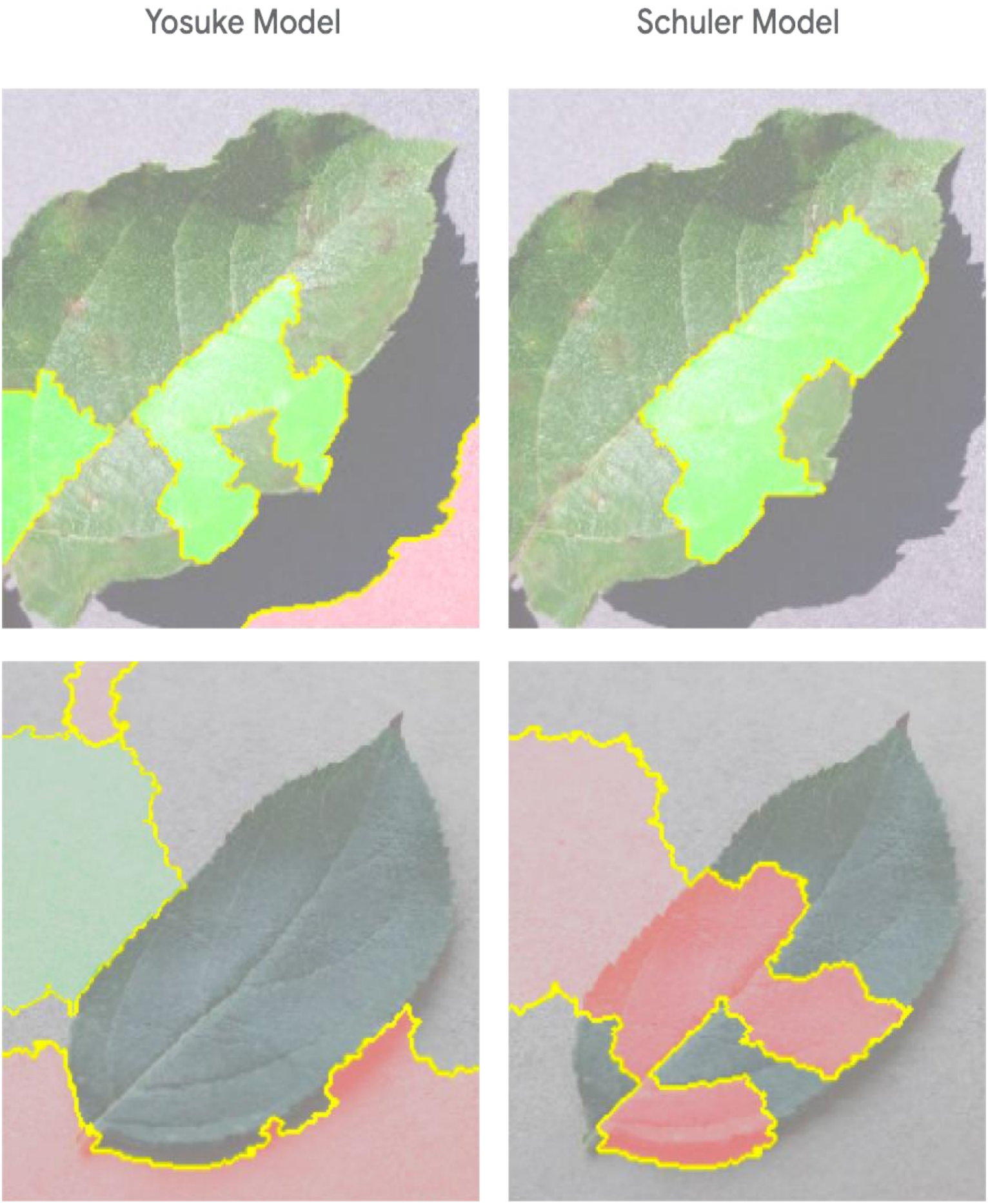


Figure 2-29. Examples of using LIME to explain the classifications between two different models.

As you can see in these examples in Figure 2-29, LIME has determined that the background of the

images is some of the most influential areas in the prediction. Given that the background is the same for all images in the dataset, this seems highly unlikely to be what is the cause for predictions!

Guided Backpropagation and Guided Grad-CAM

Here's what you need to know about Guided Backprop and Guided Grad-CAM:

- builds on DeConvNets which examine the interior layers of a convolution network
- corresponds to the gradient explanation setting negative gradient entries to zero while backpropagating through a ReLU unit
- Combined with Grad-CAM through element-wise product to give Guided Grad-CAM which has sharper visualizations in the saliency maps

Pros	Cons
Creates a sharper and more fine-grained visualization of saliency attribution maps	Some evidence suggests that Guided Backpropagation and its variants fail basic 'sanity' checks and show minimal sensitivity to model parameter randomization tests and data randomization tests[a]
Guided Grad CAM saliency maps localize relevant regions but also highlight fine-grained pixel detail	

a Adebayo, J. et al. "Sanity Checks for Saliency Maps", https://arxiv.org/pdf/1810.03292.pdf, 2020

As we've seen throughout this chapter, the gradients of a neural network are a useful tool for measuring how information propagates through a model to eventually create a prediction. The idea is that by measuring the gradients through backpropagation you can highlight those pixels or pixel regions that contributed most to the model's decision. Despite this intuitive approach, in practice the results of saliency maps that rely solely on gradient information can be very noisy and difficult to interpret. Guided Backpropagation (often referred to as 'Guided Backprop') also relies on model gradients to produce a saliency map but modifies the backpropagation step slightly in how it handles the ReLU nonlinearities.

Guided Backprop and DeConvNets

The technique of Guided Backprop is closely related to an earlier explainability method called DeConvNets. We'll start by describing how DeConvNets work first, then see how Guided Backprop improves upon that approach. As the name suggests, DeConvNets are built upon deconvolution layers, also known as transposed convolutions. You can think of a deconvolution as an inverse of a convolution layer, its job is to 'undo' the operation of a convolution layer. Figure 2-30 shows the typical architecture of a DeConvNet.

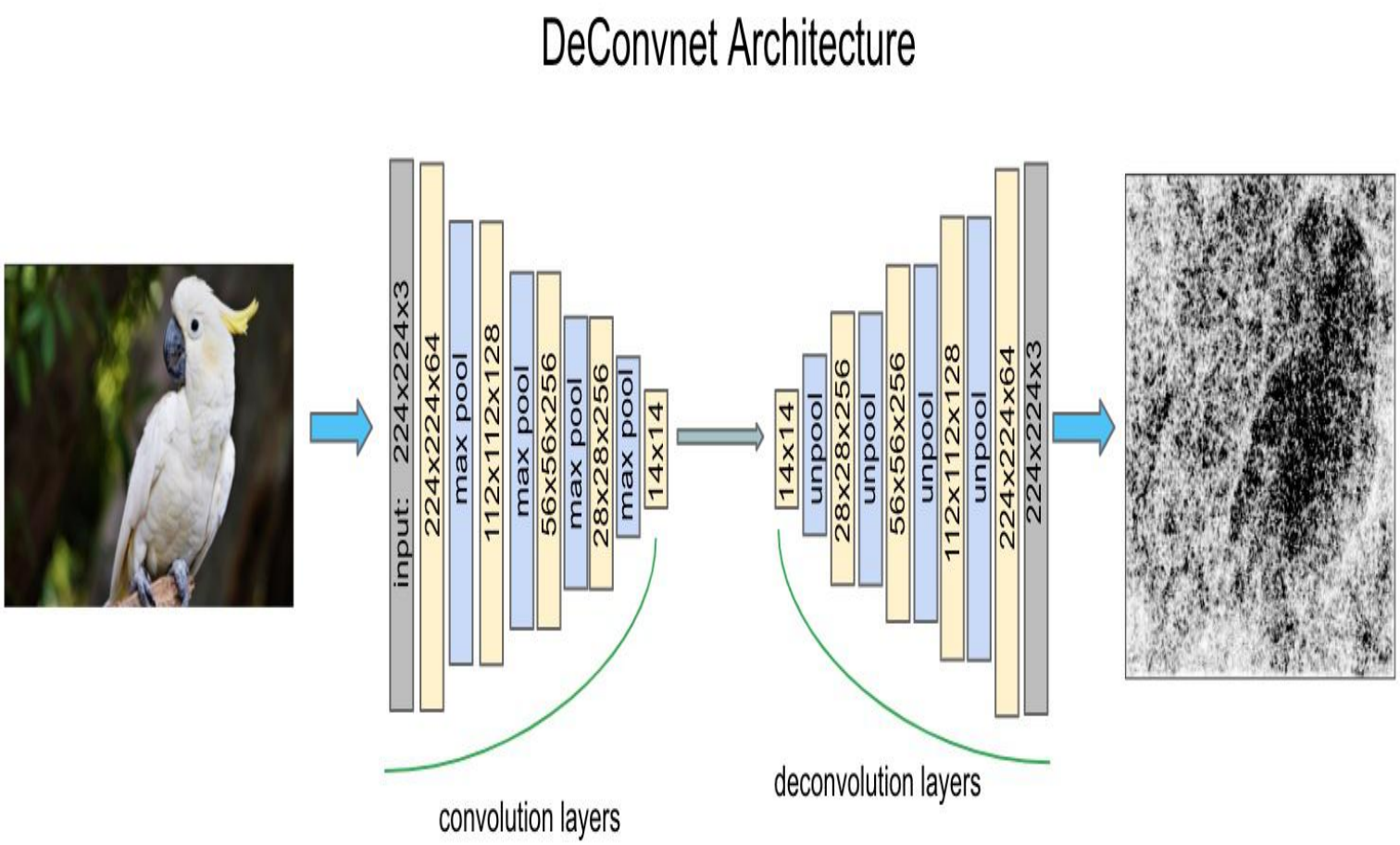
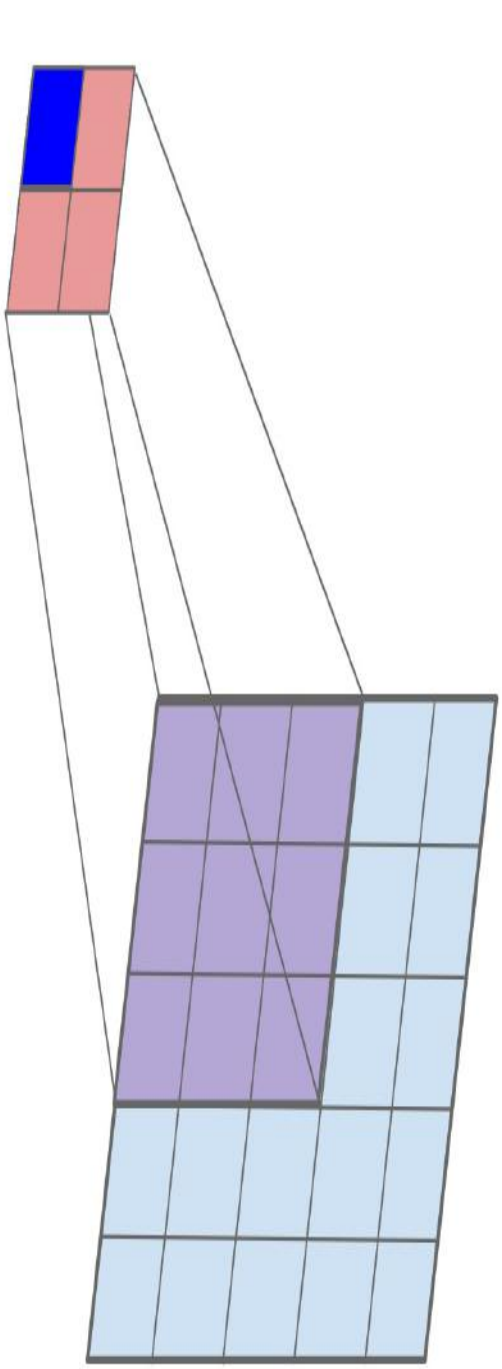


Figure 2-30. A typical deconvolution network with a VGG-16 backbone. The DeConvNet architecture starts with the usual convolution and max pooling layers, followed by deconvolution and unpooling layers.

As you can see from the figure, the DeConvNet architecture starts with a usual convolution network. Each convolutional layer passes a kernel over the input channels and calculates an output based on the weights of the filter. The size of the output of a convolution layer is determined by the kernel size, the padding and the stride. For example, if you have an input tensor with shape (5, 5) and a convolution layer with kernel size 3x3, stride set to 2 and zero padding, the output will be a tensor of shape (2,2), as shown on the left in Figure 2-31. The 9 values in the base (5,5) tensor are aggregated as a linear combination to produce the single value in the resulting (2,2) tensor. If the input has multiple channels, the convolution is applied to each channel.

The second part of the DeConvNet are the deconvolution (or transposed convolution) layers. Deconvnets are similar to convolutional networks but work in reverse (reversing filters, reversing pooling, etc.) so that they reconstruct the spatial resolution of the original input tensor. There are two components to the DeConvNet: the deconvolution and the unpooling layers. We'll discuss how the deconvolution layers work first. The deconvolution also uses a convolution filter but enforces additional padding both outside and within the input values to reconstruct the tensor with larger shape. For the convolution step we just described, the deconvolution maps the (2,2) tensor back to a tensor of shape (5,5), as shown in the figure on the right in Figure 2-31. The deconvolution filter has size 3x3. In order to upsample the (2,2) tensor to a tensor of shape (5,5) we add additional padding on the outside and between input values (in red in Figure 2-31).

Convolution layer with 3x3 filter mapping a (5,5) tensor to (2,2) tensor



Deconvolution layer with 3x3 filter mapping a (2,2) tensor to (5,5) tensor

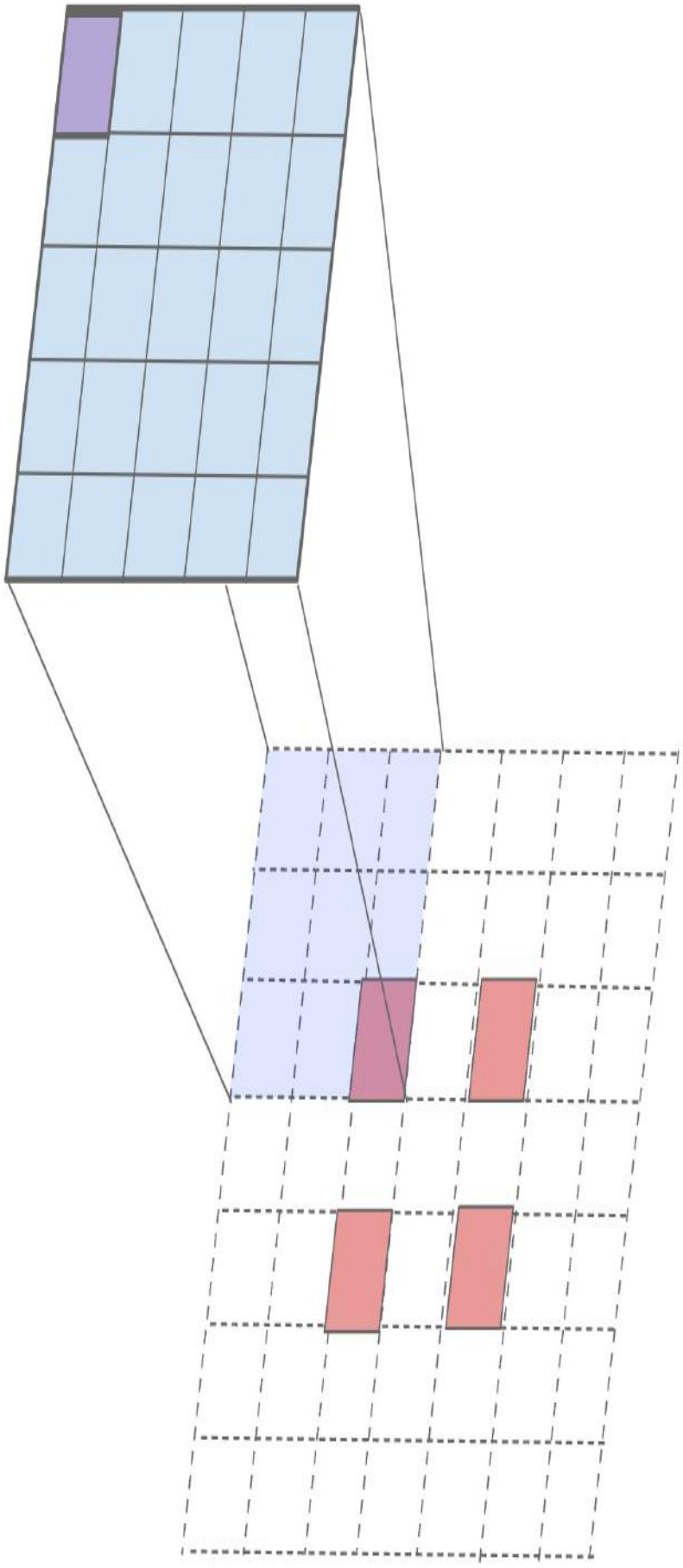


Figure 2-31. A deconvolution layer, shown on the right, reconstructs the original spatial resolution of the input to a convolution layer, shown on the left.

The next component of the DeConvNet to discuss are the unpooling layers. The max pooling layers of a convolution network pass a filter over the input and record only the maximum of all values in the filter as the output. By nature max aggregation loses a lot of information. To perform unpooling, we need to remember the position where the maximum value occurred in the original max pooling filter. These locations are called switch variables. When performing unpooling, we place the max value from max pooling back in its original position, and leave the rest of the values in the filter as zero, as shown in Figure 2-32.

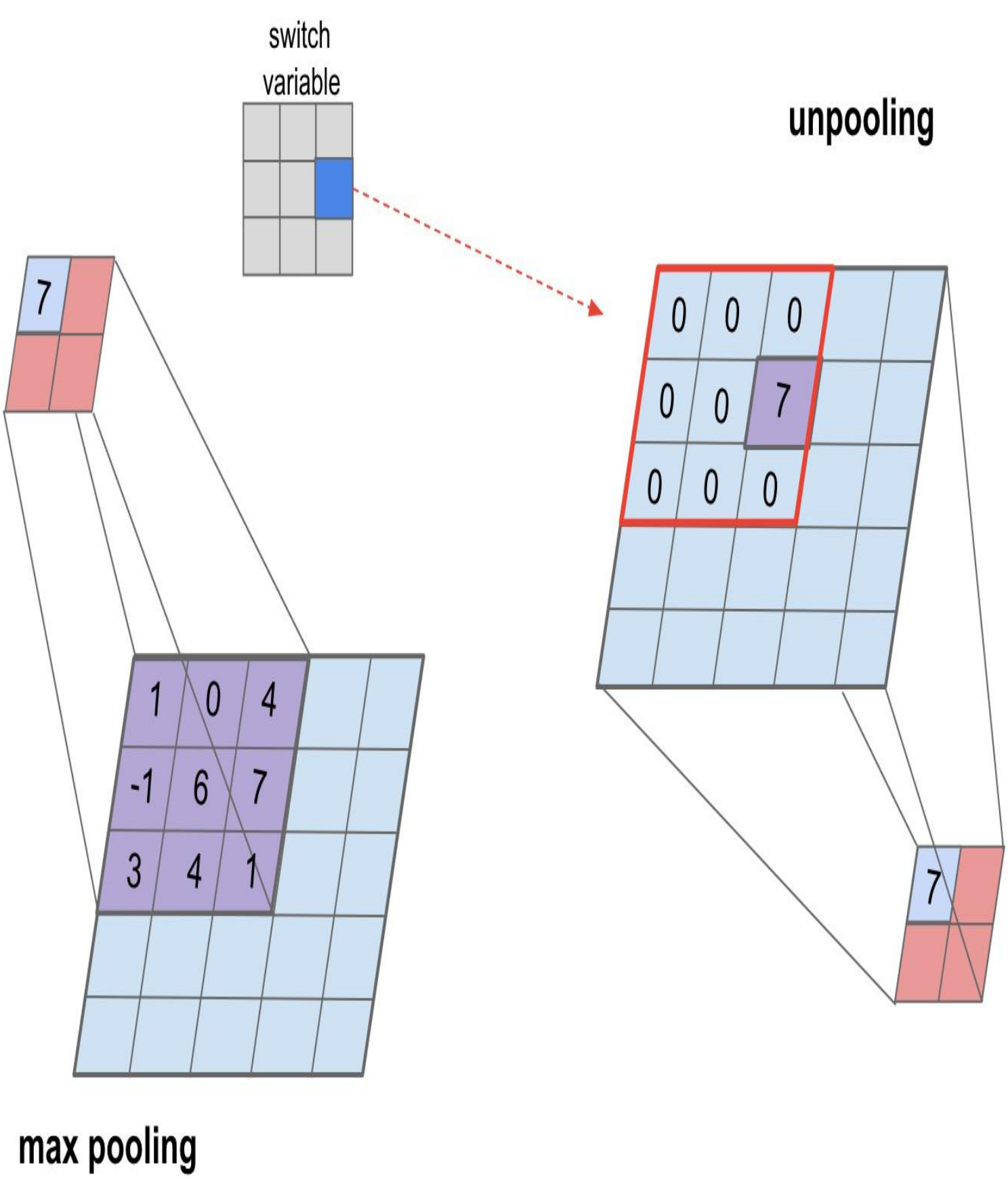


Figure 2-32. When reversing a max pooling layer using 'unpooling', the switch variable keeps track of where the max value was located.

The idea of using DeConvNets as an explainability method is to visualize the internal activation layers of a trained CNN by 'undoing' the convolution blocks of a CNN using deconvolution layers block by block. Ultimately, the DeConvNet maps the filter of an activation layer back to the original image space allowing you to find patterns in the images that caused that given filter activation to be high.

WARNING

Applying a convolution and max pooling causes some information of the original input tensor to be lost. This is because the convolution filter is aggregating values and the max pooling returns only one value from the entire filter, dropping the rest. The deconvolution reverses the convolutions step but it is an imperfect reconstruction or the original tensor.

The method of Guided Backpropagation is similar to DeConvNets. The difference is in how Guided Backprop handles backpropagation of the gradient through the ReLU activation functions. A DeConvNet only backpropagates positive signals; that is, it sets all negative signal from the backward pass to zero. The idea is that we are only interested in what image features an activation layer detects, not the rest so we only focus on positive values. This is done by multiplying the backward pass by a binary mask so that only positive values are preserved. In Guided Backprop, you also restrict to just the positive values of the input to a layer. So, a binary mask is applied both for the backward pass *and* the forward pass. This means there are more zero values in the final output, but it leads to sharper saliency maps and more fine-grained visualizations.

NOTE

Here the name ‘Guided’ indicates that this method uses information from the forward pass in addition to the backward pass to create a saliency map. The forward pass information helps to guide the deconvolution. This is similar in spirit to how Guided Integrated Gradients differs from vanilla Integrated Gradients in that it uses information about the baseline and the input to guide the path when computing a line integral.

Guided Grad-CAM

Guided Grad-CAM combines the best of both Grad-CAM and Guided Backprop. As we saw in the section on Grad-CAM, one of the problems with Grad-CAM was that the coarse heatmap produced from the activation layers must be upsampled so that it can be compared against the original input image. This upsampling and the subsequent smoothing leads to a lower resolution heatmap. The original authors of Grad-CAM proposed guided Grad-CAM as a way to combine the high-resolution output from Guided Backprop with the class-specific gradient information obtained from Grad-CAM alone. This doesn’t alleviate all of the concerns with Grad-CAM since it still relies on the output of the Grad-CAM technique which has some fundamental concerns (see the Section on Grad-CAM for more in depth discussion on this) but it is an improvement. Guided GradCAM combines Grad-CAM and Guided Backprop by taking an element-wise product of both outputs, as shown in Figure 2-33.

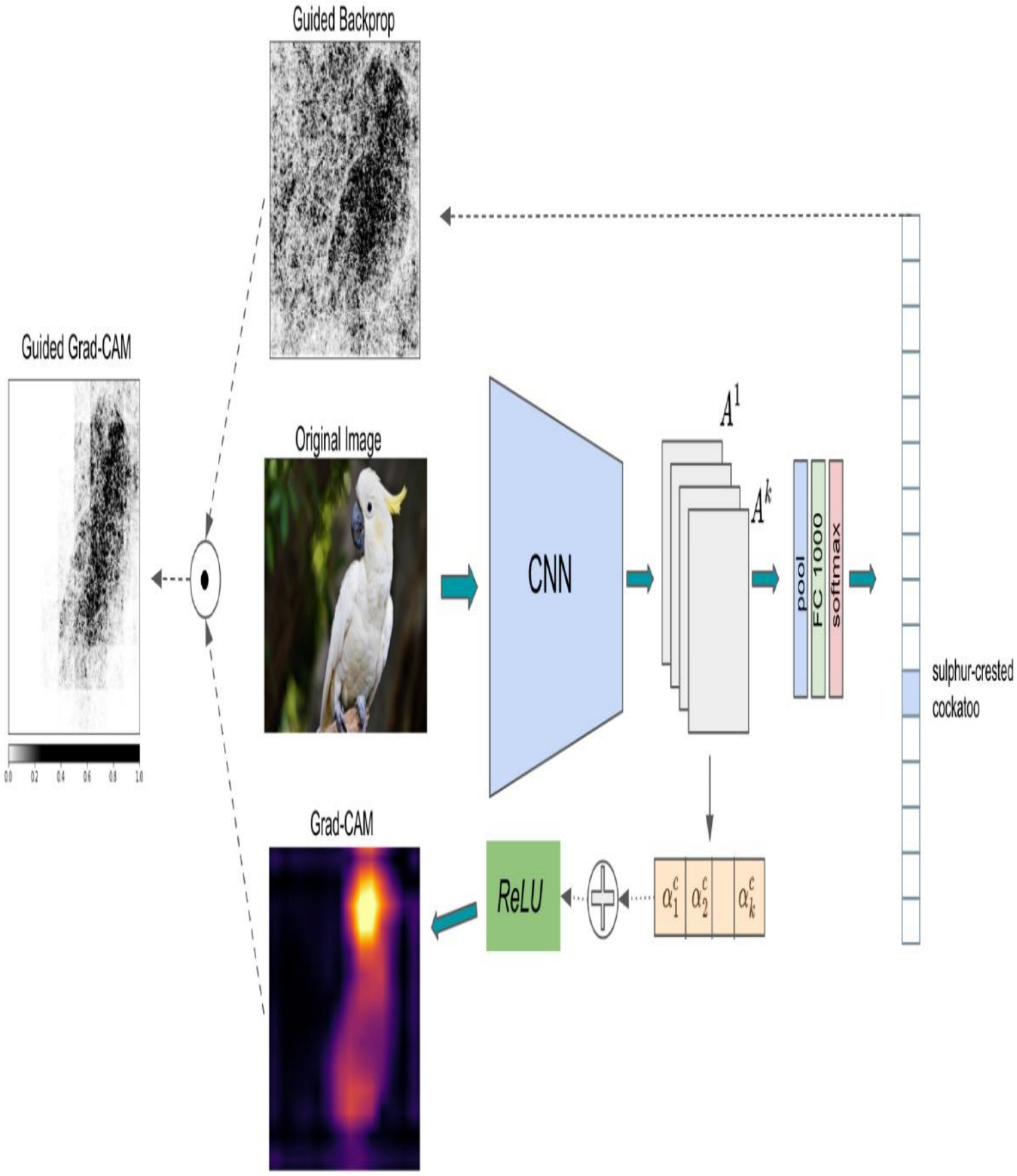


Figure 2-33. Guided Grad-CAM combines the output of Grad-CAM and Guided Backpropagation by taking the element wise product. This produces a visualizations

Both Guided Backprop and Guided Grad-CAM have easy to use implementations available in the Captum library. Let's look at an example to see how these techniques are implemented in code. Captum is built on PyTorch so let's start by loading an Inceptionv3 model that has been pre-trained on the

ImageNet dataset. This is done in the following code block, see the Guided Backprop notebook in the book's GitHub repository for the full code for these examples.

```
import torch
model = torch.hub.load('pytorch/vision:v0.10.0',
     'inception_v3',
                       pretrained=True)
model.eval()
```

In the previous code block, model.eval() tells PyTorch that we're using the model for inference (i.e. evaluation), not for training. To create attributions using Guided Backprop requires only a couple lines of code. First we create our GuidedBackProp attribution object by passing the Inception model we previously created, then we call the attribute method passing the input example (as a batch of one) and the target class id that we want to create explanations for. Here we set target=top5_catid[0] to use the top predicted class for the given input image, as shown in the following code block:

```
gbp = GuidedBackprop(model)
# Computes Guided Backprop attribution scores for class 89 (cockatoo)
attribution = gbp.attribute(input_batch, target=top5_catid[0])
```

This returns an attribution mask which we can then visualize using Captums' visualization library. The result is shown in Figure 2-34.

You can implement Grad-CAM using the Captum library in a very similar way. Remember, Grad-CAM works by creating a coarse heatmap from the internal activation layers of the CNN model. These internal activations can be taken from any of the convolution layers of the model. So, when implementing Grad-CAM you specify which convolution layer to use for creating the heatmap. Similarly, for Guided Grad-CAM you specify which layer of the model as well. This is shown in the following code block. When creating the GuidedGradCam object we pass the model as well as the layer model.Mixed_7c, this is the last convolution layer of the Inceptionv3 model.

```
from captum.attr import GuidedGradCam
guided_gc = GuidedGradCam(model, model.Mixed_7c)
attribution = guided_gc.attribute(input_batch, target=top5_catid[0])
```

Once we have the attributions, we can visualize the result using Captum's visualization library as before, see the Guided Backprop notebook in the GitHub repository for the full code for this example.

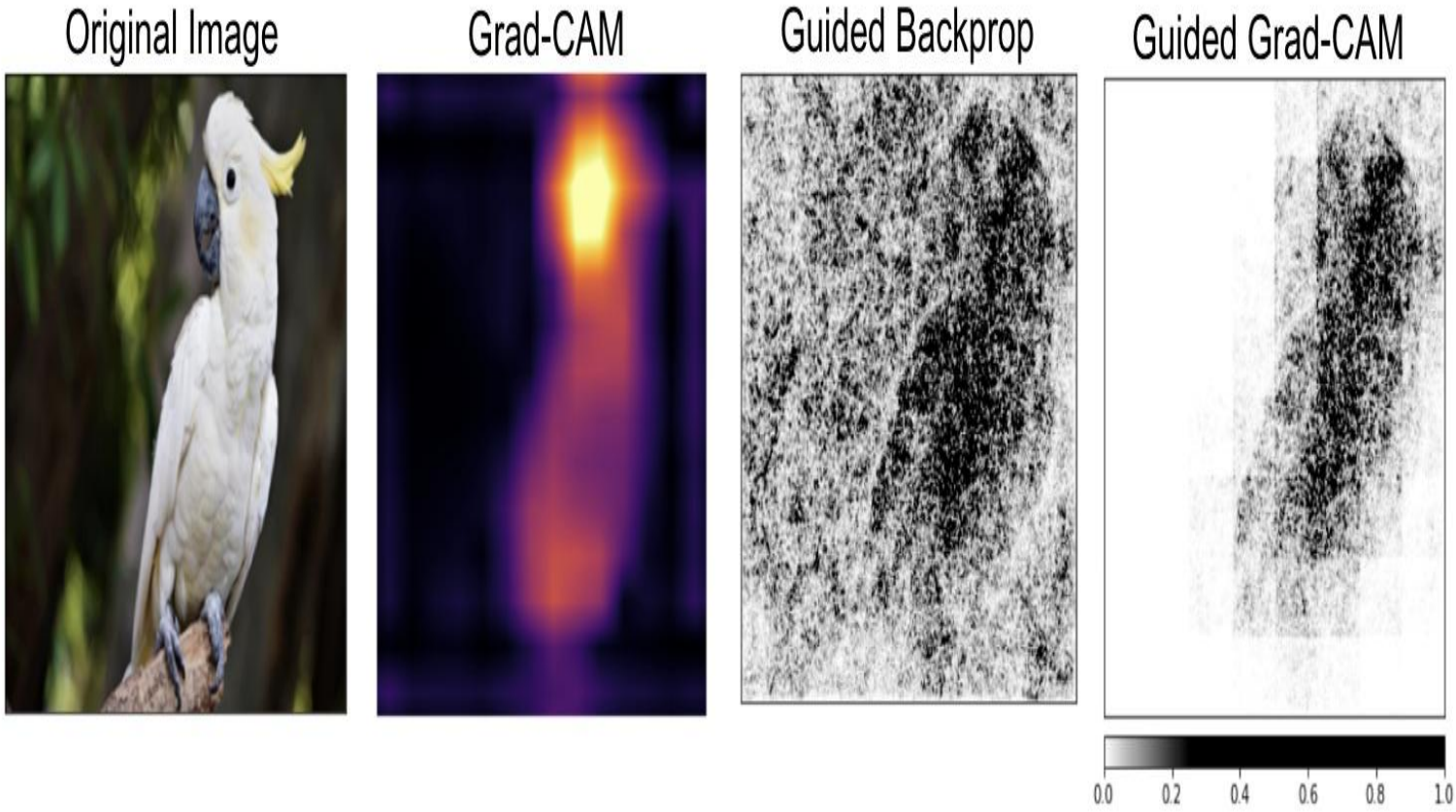


Figure 2-34. Guided Grad-CAM combines the output of Grad-CAM and Guided Backprop by taking an elementwise product of the output of the two methods.

Summary

Computer vision models have critical applications in a wide range of contexts from healthcare and security to manufacturing and agriculture. Explainability is essential for debugging a model's predictions and assuring a model isn't learning spurious correlations in the data. In this chapter, we looked into various explainability techniques that are used when working with image data.

Loosely speaking, explanation and attribution methods for computer vision models can be fit into a few broad categories: back-propagation methods, perturbation methods, methods that leverage the internal state of the model, and methods that combine different approaches. The explainability techniques we discussed in this chapter are representative examples of these categories and each method has their own pros and cons.

Integrated Gradients and its many variations fall in the category of back-propagation techniques. XRAI combines integrated gradients with segmentation based masking to determine regions of the image that are most important in the model's prediction. By over-segmenting the image and aggregating smaller regions of importance into larger regions based on attribution scores, this produces more human-relatable saliency maps instead of pixel-level attributions obtained via integrated gradients alone. Methods like Grad-CAM and Grad-CAM++ also rely on gradients but they leverage the internal state of the model. More specifically, Grad-CAM uses the class activation maps of the internal convolutional layers of the model to create a heat map of influential regions for an input image.

LIME is a model agnostic approach that treats the model as a black-box and determines pixels that are

relevant to the model's prediction by perturbing input pixel values. These methods use gradients of the model to create saliency maps (also called sensitivity maps or pixel attribution maps) to represent regions of an image that are particularly important for the model's final classification. Lastly we discussed Guided Backpropagation and Guided Grad-CAM which combine different approaches. Guided Grad-CAM combines Guided Backprop and Grad-CAM using an element-wise product, getting the best of both methods and addresses some of the problems that arise when using Grad-CAM.

In the next chapter, we'll look into explainability techniques that are commonly used for text based models and how some of the techniques that we've already seen can be adapted to work in the domain of natural language.

---

1 This is analogous to many loss-based functions for training DNNs that measure the gradient and performance change of the model in the loss space.

2 Selvaraju. R, et al. 'Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization"

3 The Grad-CAM paper was revised and updated in 2019

4 Defenders of Grad-CAM claim that this up-scaling is acceptable because it mirrors the downsampling performed by the model's CNNs, but this argument does not have any strong theoretical grounding or evaluations to substantiate this claim.

About the Authors

Michael Munn is an ML Solutions Engineer at Google where he works with customers of Google Cloud on helping them design, implement, and deploy machine learning models. He also teaches an ML Immersion Program at the Advanced Solutions Lab. Michael has a PhD in mathematics from the City University of New York. Before joining Google, he worked as a research professor.

David Pitman is a Senior Engineering Manager working in Google Cloud on the AI Platform, where he leads the Explainable AI team. He is also a co-organizer of PuPPy, the largest Python group in the Pacific Northwest. David has a M.Eng. and BS in Computer Science, focusing in AI and Human-Computer Interaction, from MIT, where he was previously a research scientist.