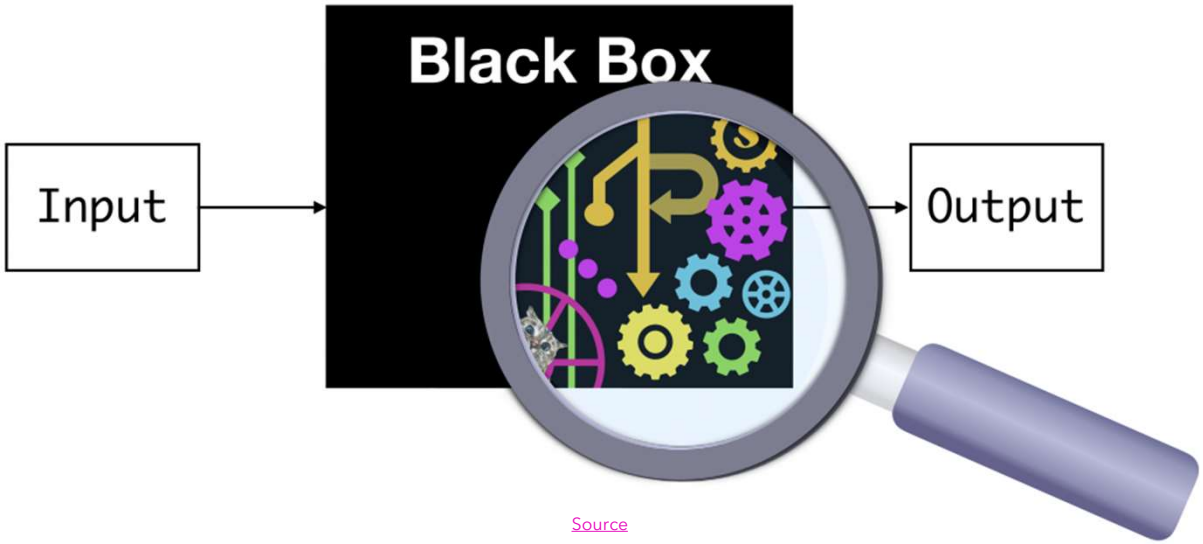


Source

ML Interpretability

Umar Jamil

Topics

- What is ML Interpretability
- Review of deep learning and backpropagation
- Let's trick a classifier!
- Interpretability Engine for vision models
- Feature visualization
- Interpretability for language models

Prerequisites

- Basics of calculus (derivative)
- Basics of deep learning (know what is a neural network)

What is ML Interpretability?

In 2016 there was a fatal accident involving a Tesla car driver. As reported by The Guardian:

> According to Tesla's account of the crash, **the car's sensor system, against a bright spring sky, failed to distinguish a large white 18-wheel truck and trailer crossing the highway**. In a blogpost, Tesla said the self-driving car attempted to drive full speed under the trailer "with the bottom of the trailer impacting the windshield of the Model S".

Our goal is to be able to answer the following questions:

1. What did the model learn?
2. What features/patterns from the input make the model generate certain outputs?

Knowing how a model *thinks*, allows us to:

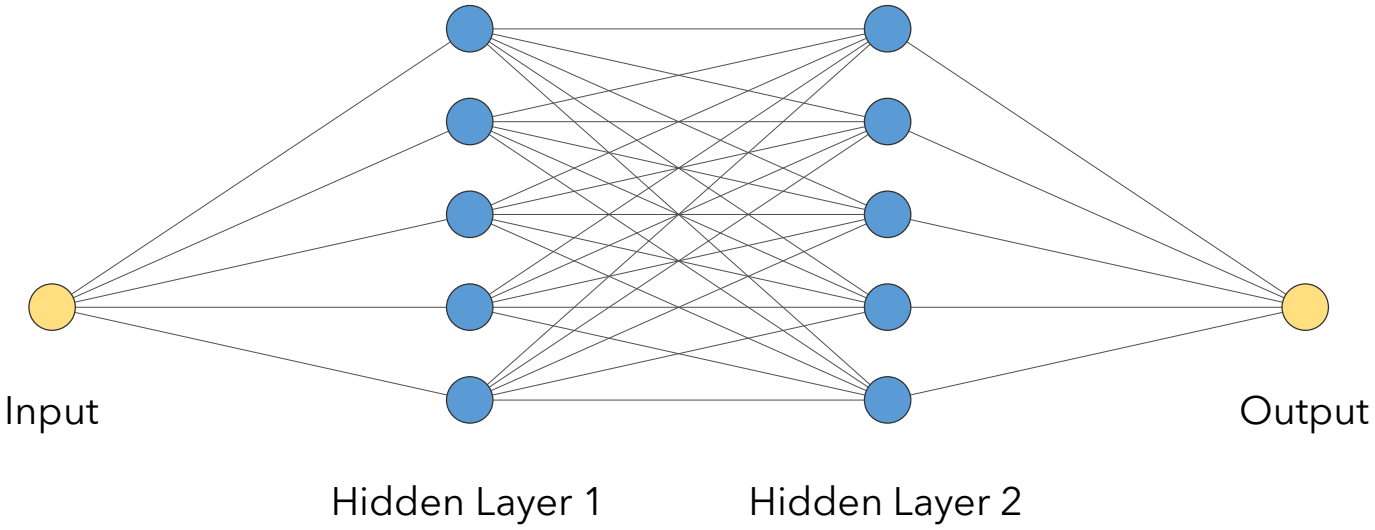
1. Debug and tune the model (Why isn't my model performing well? How do my choice of hyperparameters affect its learning?)
2. Identify failure modes before deployment.
3. Increasing trust: demonstrating your model is well trained so people will trust it.
4. Discovering novel insights from data (does my model know something I don't?)

Deep Learning

Deep learning is a class of machine learning algorithms/methods that are based on Artificial Neural Networks, or simply Neural Networks (NN). These networks extract features from the input with subsequent applications of layers.

Examples of neural networks are Convolutional Neural Networks (CCN), Recurrent Neural Networks (RNN) and Transformers.

Frameworks like PyTorch convert a model into a computational graph, let's see how it works.



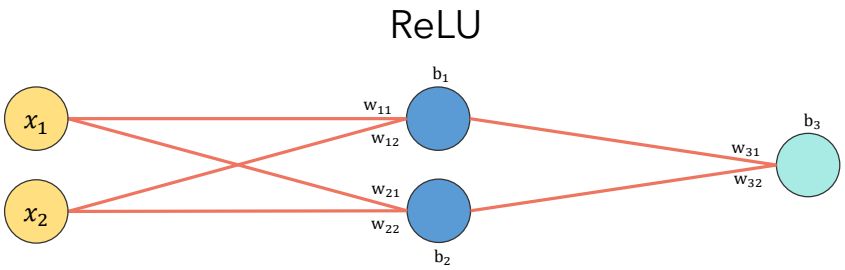
Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

The computational graph (1)

Imagine we have a very simple neural network to compute the price of a house by using its number of bedrooms (x_1) and the number of bathrooms (x_2).

Each neuron j in the hidden layer will compute the following: $\left(\sum_1^2 x_i w_{ji}\right) + b_j$

Let's visualize the computation graph for this neural network.



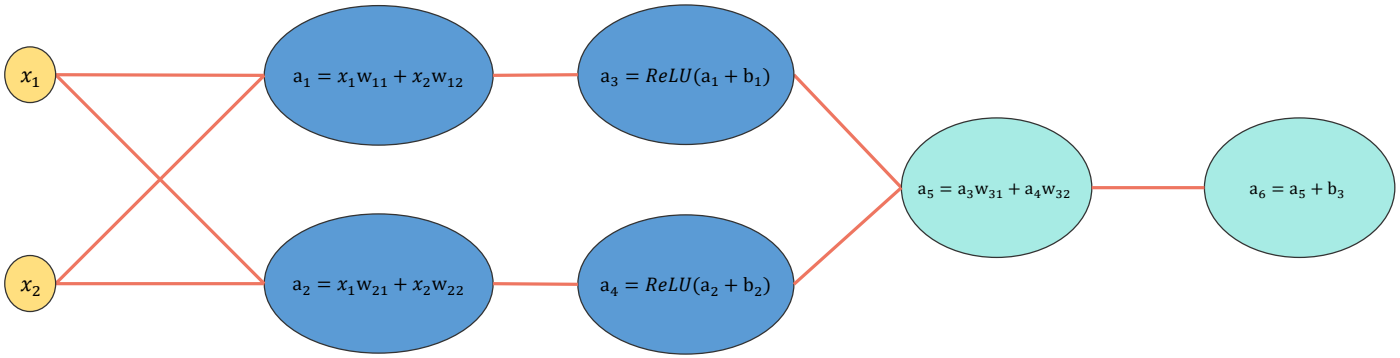

```python
class SimpleNet(nn.Module):

    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)

    def forward(self, x):
        l1 = torch.relu(self.fc1(x)) # Layer 1
        o = self.fc2(l1) # Output
        return o
```

The computational graph (2)

Imagine we have a very simple neural network to compute the price of a house by using its number of bedrooms (x_1) and the number of bathrooms (x_2)

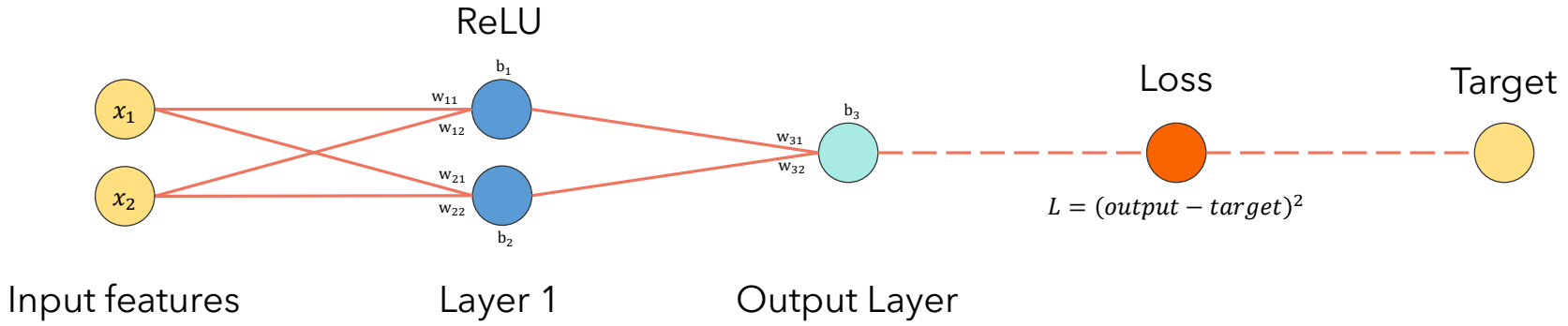


Input features Layer 1 Output

Backpropagation (1)

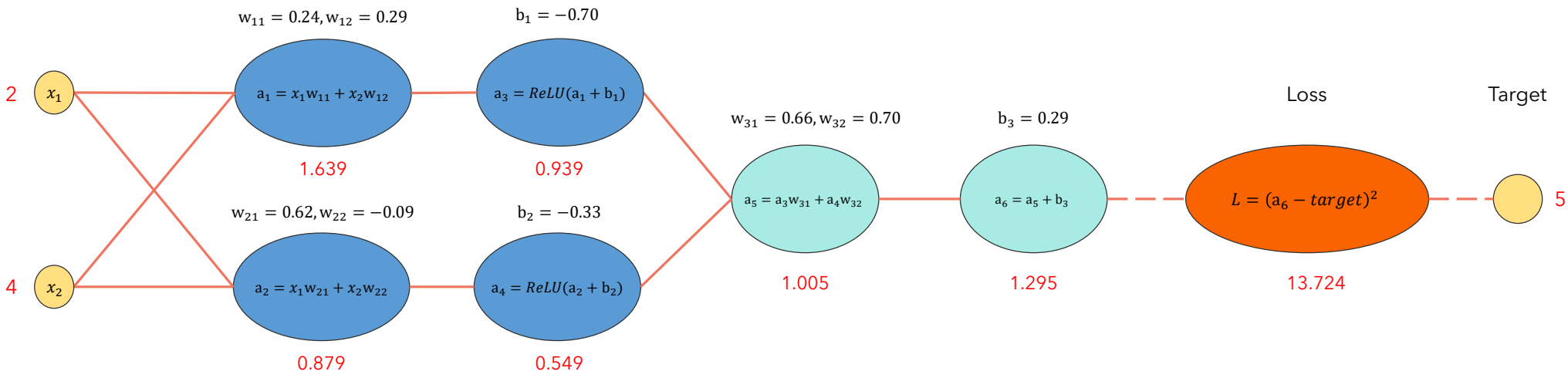
Usually, we have a dataset made of (input, output) pairs and our goal is to *train* the neural network to minimize a certain *loss function* that we choose. For example, for regression tasks (e.g. estimating the price of a house given its features), we would like our neural network to predict the output with minimal error. Our hope is that the neural network not only learns to predict the output for the inputs seen during training, but also to generalize to unseen inputs. How do we proceed practically?

- We choose a loss function (e.g. MSE).
- Run the input through the network, calculate the loss w.r.t the target
- Calculate the gradient of the loss w.r.t the parameters of the neural network
- Update the parameters of the network to move against the direction of the gradient.



Backpropagation (2): forward pass

The first step before doing the training loop is to evaluate the output of the model given a training input, which is commonly known as forward pass (from left to right).

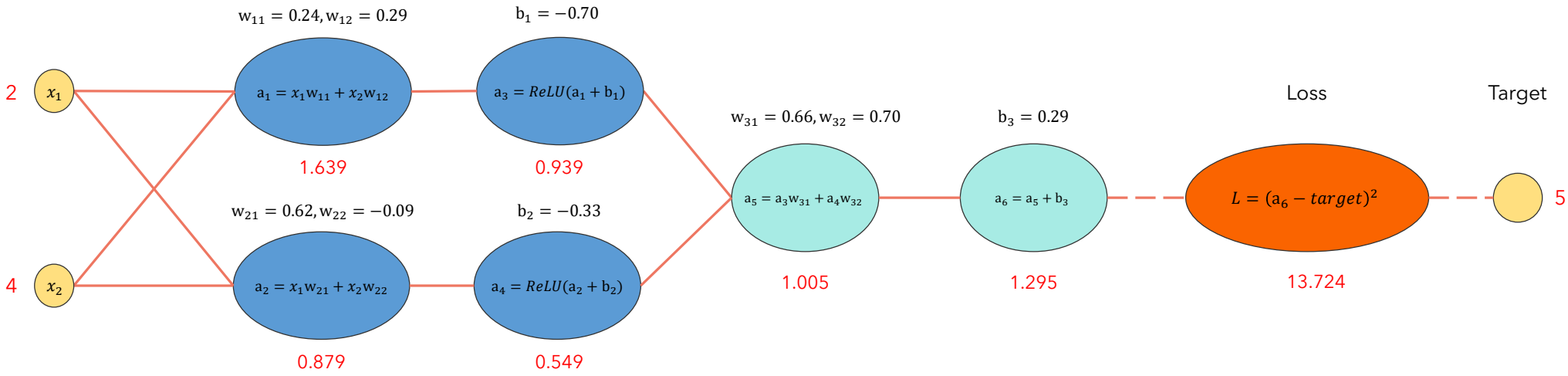


Backpropagation (2): backward pass

During the backward pass, we compute the derivative of the loss with respect to the each of the parameters of the model, and then update the weights to move against the direction of the derivative.

For example, to compute the derivative of the loss function with respect to the parameter w_{11} we need to evaluate the following gradients:

$\frac{dl}{dw_{11}} = \frac{dl}{da_6}\frac{da_6}{da_5}\frac{da_5}{da_3}\frac{da_3}{da_1}\frac{da_1}{dw_{11}}$. This is thanks to the chain rule!

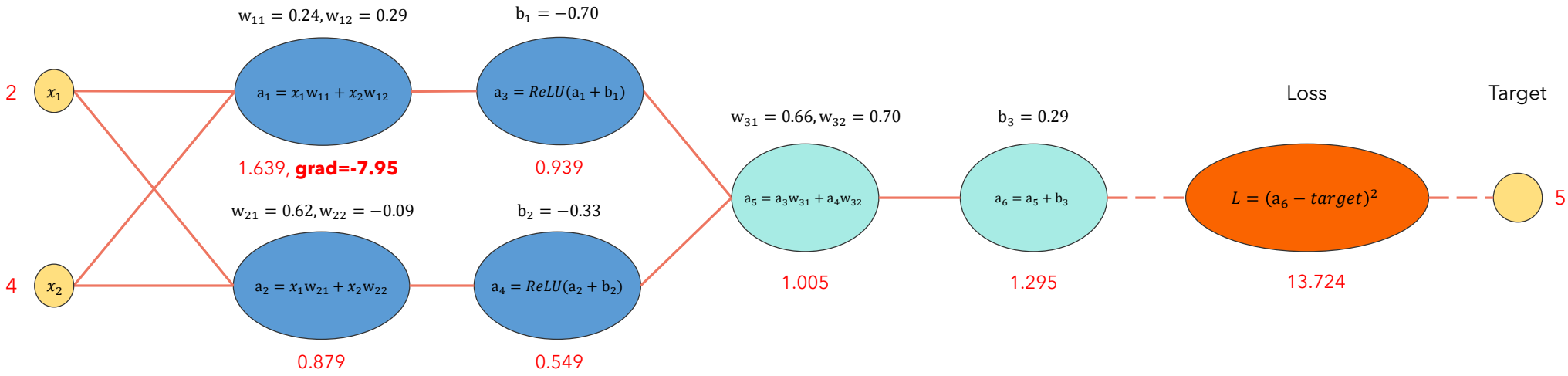


Backpropagation (2): update

Having calculated the gradient of the loss w.r.t each weight, we can update the weights to move against the direction of the gradient, using the usual update rule as follows:

$$w_{11_{NEW}} = w_{11_{OLD}} - \alpha * \frac{dl}{dw_{11}}$$

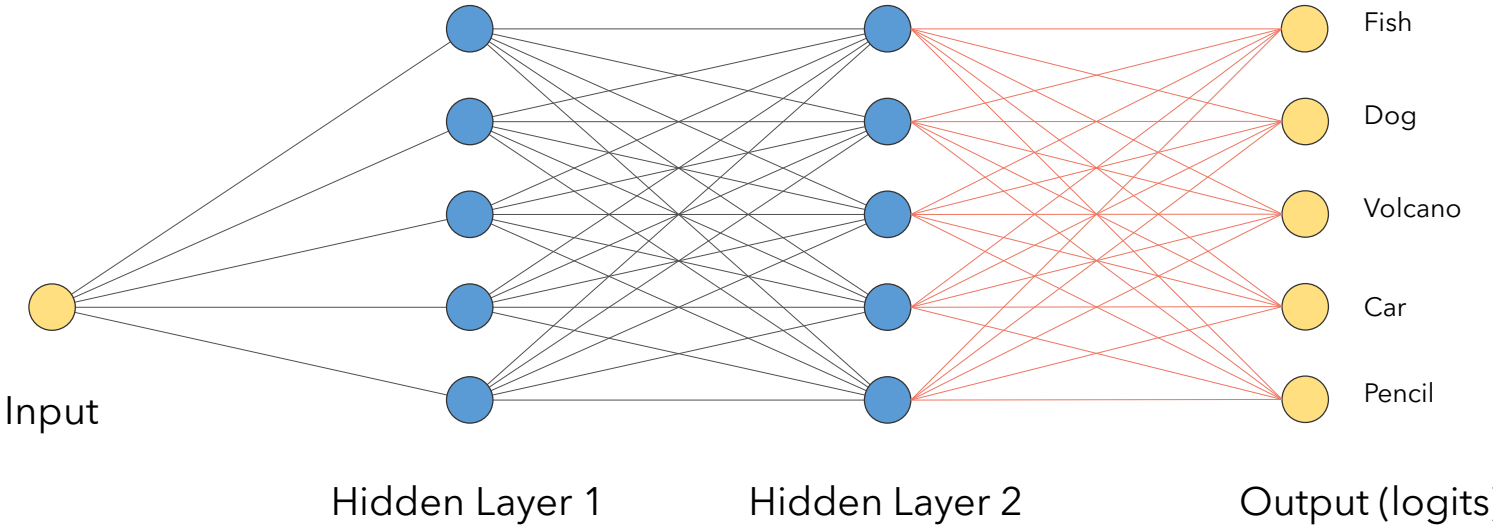
Why do we have a learning rate? Because we are running stochastic gradient descent, so the gradient direction is not the true gradient, but an approximation, so we cannot trust it fully.



Now… let's trick a classifier!

A classifier is a special type of neural network that's trained to classify the output in one of the available classes. For example, the ResNet neural network can classify the input among 1000 classes.

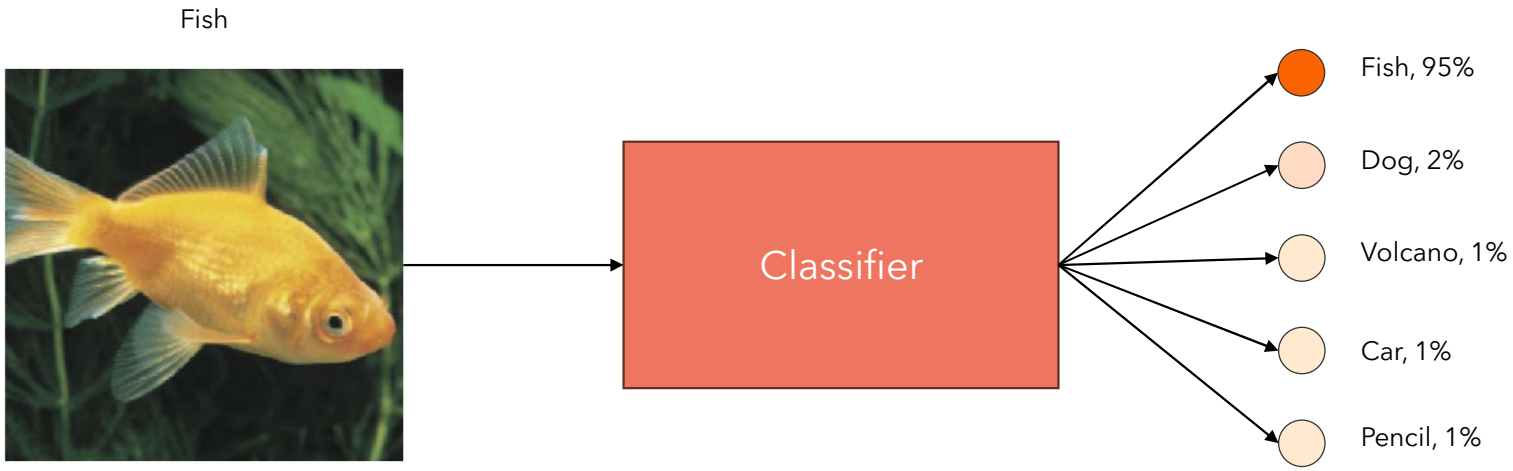
The classifier's output are known as logits, and each of them indicates a score that the model assigns to each class. We usually take the logit with the highest score (after applying a softmax) to select the predicted class and the model's confidence.



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

What do you mean by “tricking” a classifier?

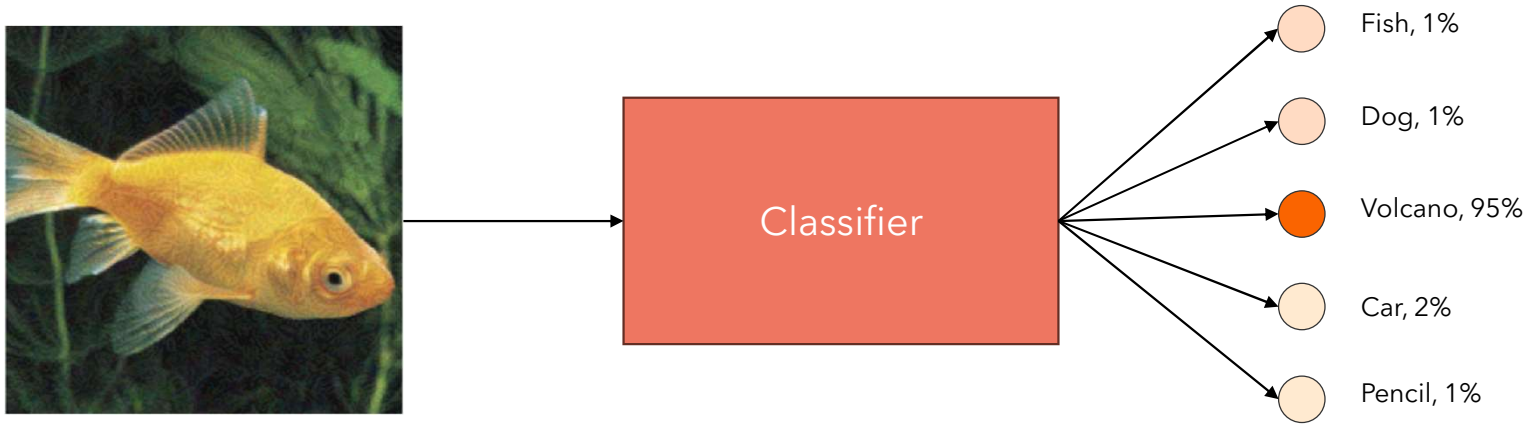
Given the image of a fish, the classifier, if trained correctly, will predict the class “fish” with high probability. What if we could make the same classifier, with the same parameters, predict our input fish as a volcano instead?



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

What do you mean by “tricking” a classifier?

Our goal is this: add a little bit of noise to the image. The image will still look like a fish, but our classifier will classify it as a volcano!



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

A side-to-side comparison

Let's visualize the two images side by side. Yes, there's a little bit of noise, but our classification model is seeing a volcano now… HOW?! WHY?!

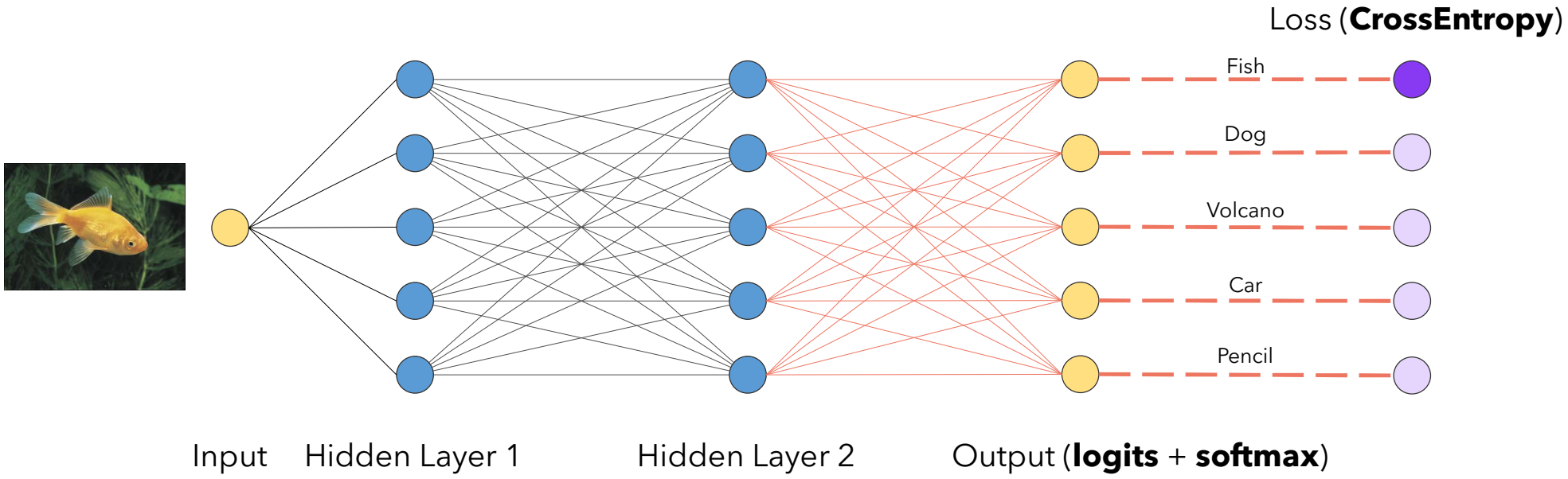
Fish

Fish with **a little bit of noise**

This is known as an **adversarial example**

How does a classifier work?

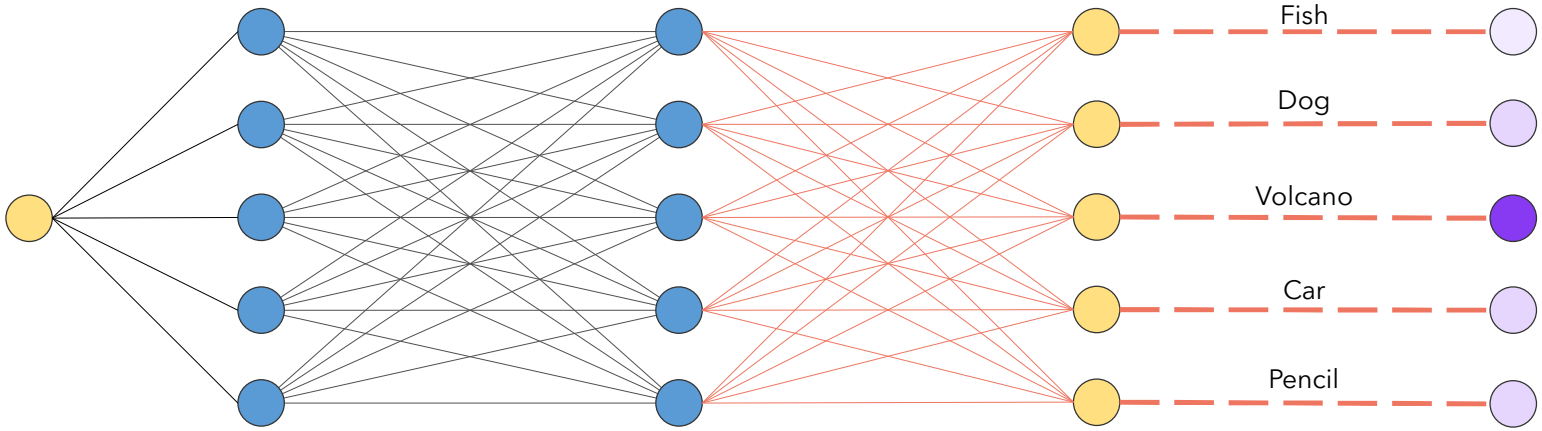
A classifier is trained as follows: we provide a series of input images along with their label and make the model learn the mapping between the input and the label using the backpropagation algorithm we saw earlier: we compute the derivative of the loss function with respect to each weight of the model and keep updating the weights until the model is good at classifying any given image from its training, validation and test data sets.



What if we backpropagate to… the input?

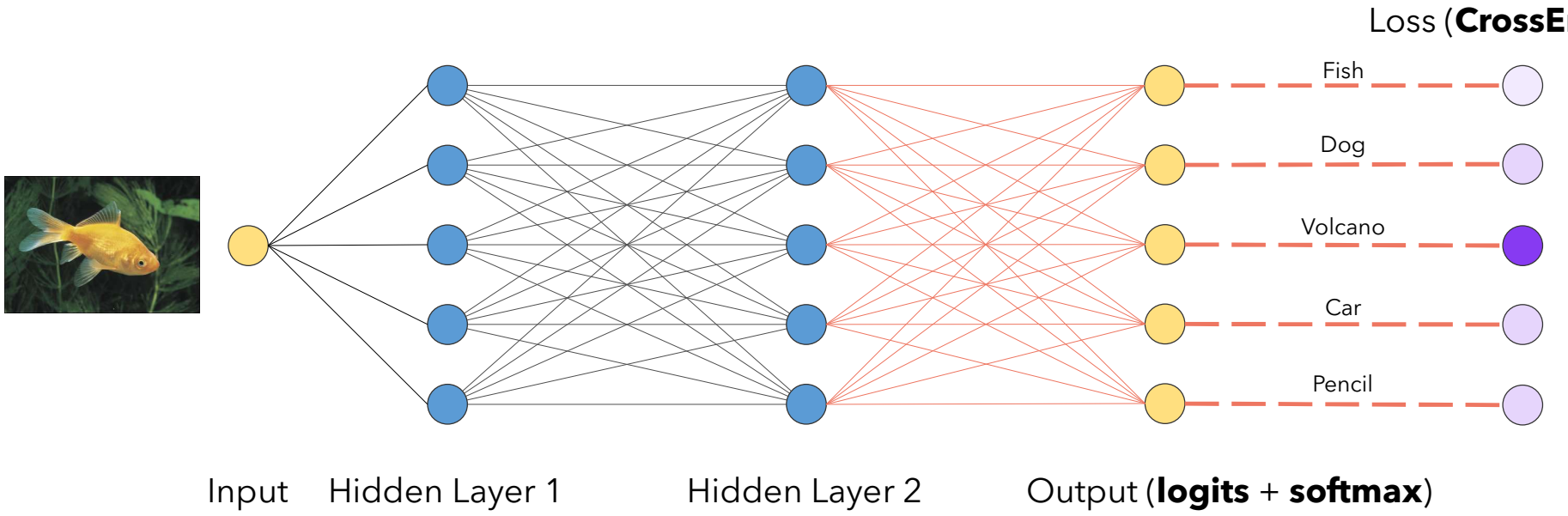
In deep learning, we calculate the gradient of the loss function w.r.t parameters of the model so that we can update the parameters of the model in such a way to minimize the loss function. Why? Because the gradient of the loss function w.r.t parameters indicates in which direction we should change the parameters to increase the loss, so if we move against the direction of the gradient, we will reduce the loss.

Here's my question for you: since PyTorch is so good at computing gradients automatically, why not use it to compute the gradient of a particular loss (of a particular class) with respect to the input image? Can you think why it could be a good idea for our hack?



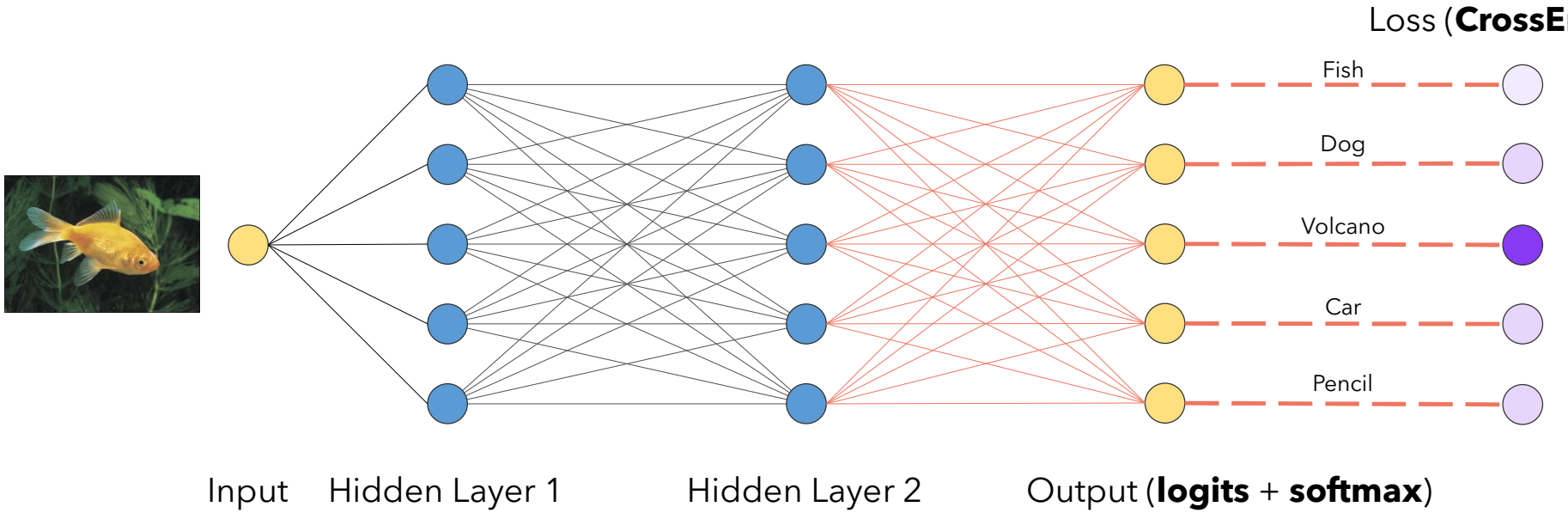
The gradient of the loss w.r.t the input

The gradient of the loss w.r.t the input indicates the direction in which we should change the input to increase the loss. So, we can "change" our input towards this direction so that a specific logit increases.



The gradient of the loss w.r.t the input

For example, we could compute the gradient of the loss (for the target class "volcano") with respect to the input image and then move our input image against the direction indicated by the gradient to reduce the loss for the chosen target class.



Show me the code already!

```python
import torch
import torch.nn as nn

def generate(model: nn.Model, input: torch.Tensor, target_class: int, num_steps: int = 10) -> torch.Tensor:
    alpha = 0.025 # Amount of perturbation at each step
    epsilon = 0.25 # Max amount of perturbation
    loss_fn = nn.CrossEntropyLoss()
    # Create a copy of the input tensor so that we don't affect the original
    x_adv_prev_step = torch.tensor(input.data, requires_grad=True)
    for _ in range(num_steps):
        output = model(x_adv_prev_step)

        # Evaluate the loss function on the adversarial example
        target = torch.tensor([target_class], requires_grad=False)
        loss = loss_fn(output, target)
        # Backpropagate the loss to the input
        loss.backward()

        # At every step, add a small perturbation to the input
        x_adv_curr_step = x_adv_prev_step.data - alpha * torch.sign(x_adv_prev_step.grad.data)
        # Limit the amount of perturbation to epsilon
        x_adv_curr_step = torch.clamp(x_adv_curr_step, input-epsilon, input+epsilon)
        # Save the current step for the next iteration
        x_adv_prev_step.data = x_adv_curr_step
    return x_adv_prev_step
```

By default, PyTorch doesn't calculate the gradient of the loss w.r.t input, so we must tell it explicitly to do so.

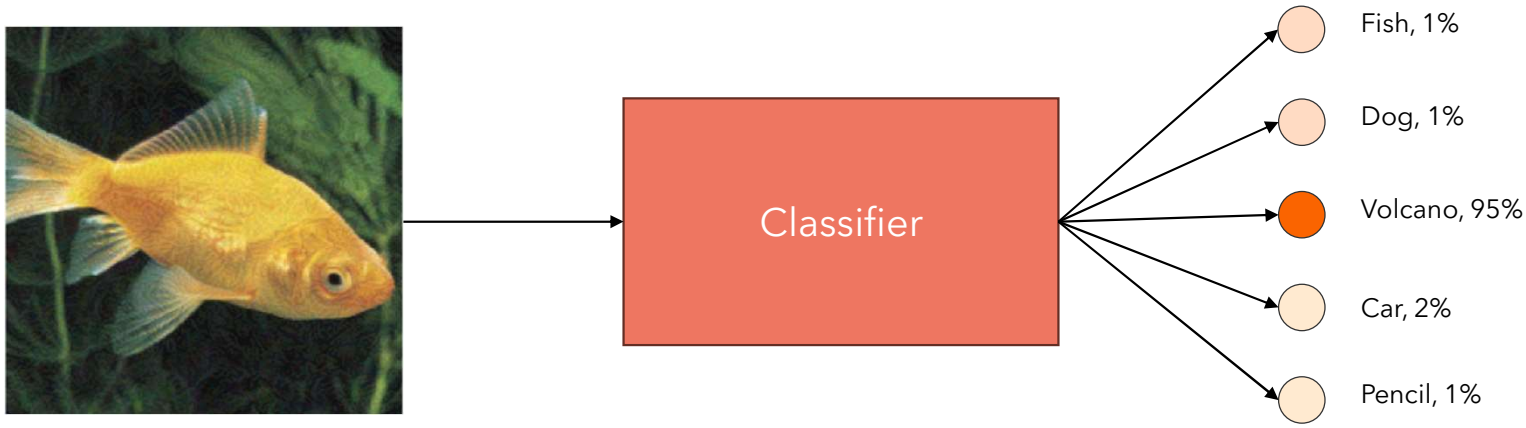
Calculate the gradient of the loss w.r.t input image

Add small amounts of noise to the image against the direction of the gradient (notice the "minus" sign).

Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Et… voila! It's been tricked!

I made this example because I want to show you that models look at patterns in their input that may not make sense for us humans, and understanding these patterns helps us improve our models. Thanks to the sponsor of today's video, **Leap Labs**, we can get insights on how our models make predictions.



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Understand what your model has learned

Enable targeted fine-tuning

Predict behaviour on unseen data

leap labs

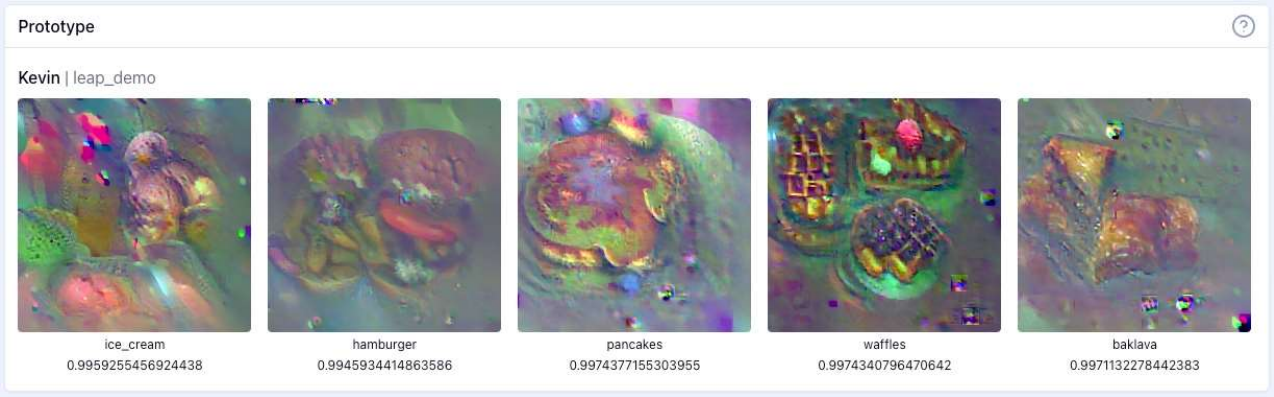
Umar Jamil - https://github.com/hkproj/ml-interpretability-notes

Interpretability engine

leap labs

See what your model has learned. It works with *any* vision model.

You can use the Leap Labs Interpretability Engine to generate prototypes for a target class (e.g. pancake), that is, what your model thinks pancakes should look like.



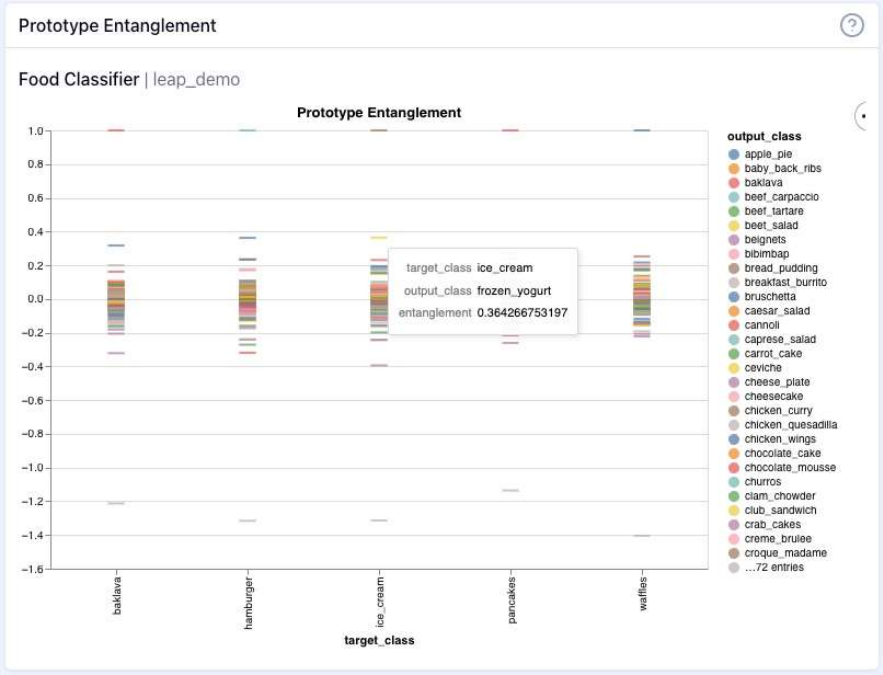
Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Interpretability engine

leap labs

Identify *where* and *why* your model is confused.

We can observe entanglement between classes, that is, how different classes share features.
For example, a classifier is likely to have significant entanglement between “cheesecake” and “apple pie” (both are round, for example), but we wouldn’t expect “cheesecake” and “dog” to have high entanglement, which may indicate a higher chance of misclassification for these two classes.



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

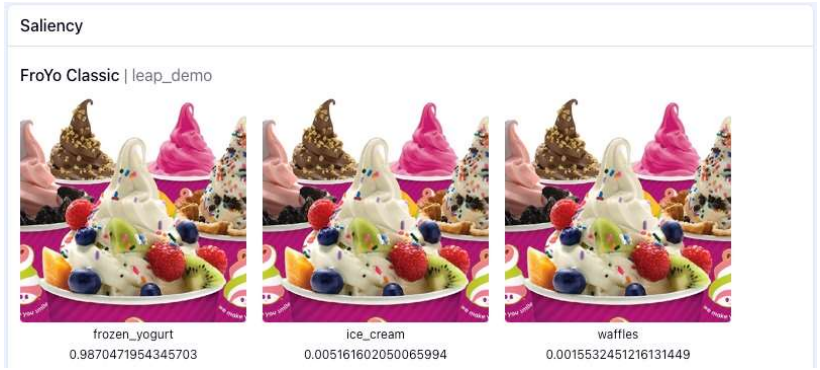
Interpretability engine

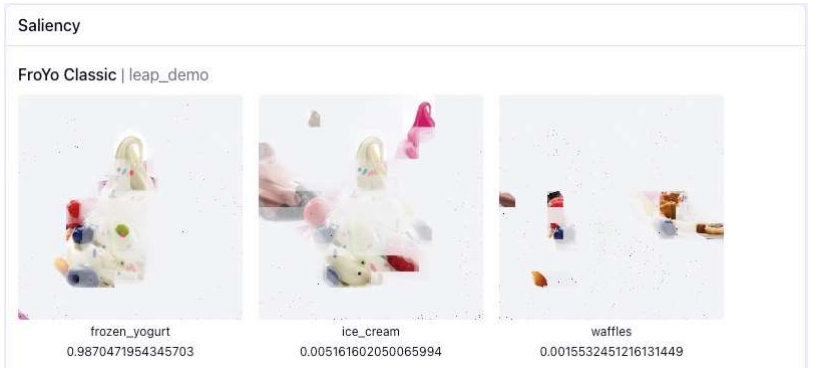
leap labs

Understand predictions on individual samples.

Feature isolation does what it says on the tin - it isolates which features in the input the model is using to make its prediction.

We can apply feature isolation in two ways:
1.On a prototype that we've generated, to isolate which features are shared between entangled classes, and so help explain how those classes are entangled; and
2.On some input data, to explain individual predictions that your model makes, by isolating the features in the input that correspond to the predicted class (similar to saliency mapping).

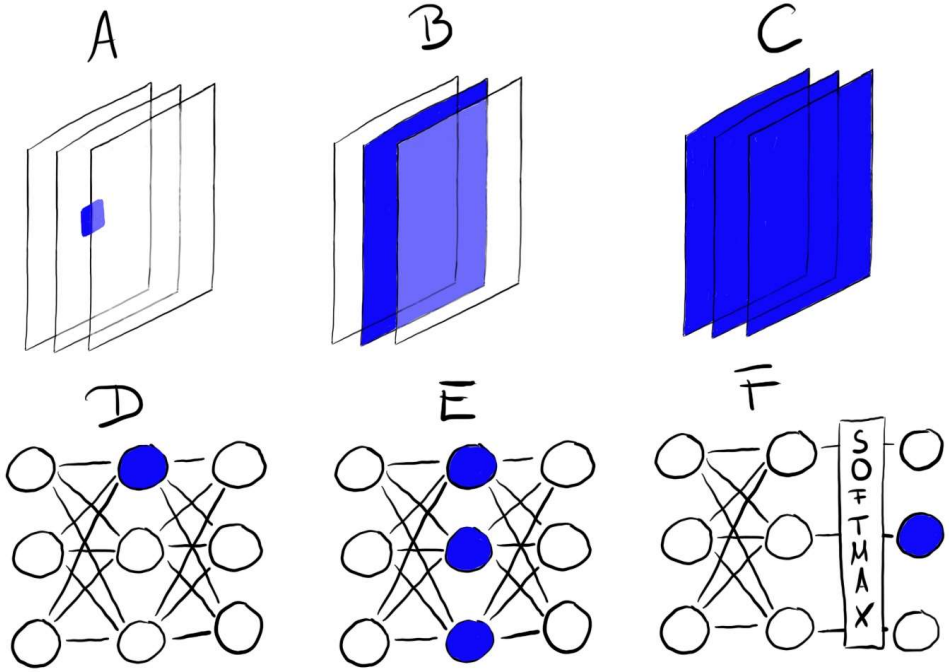




Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Feature visualization

Features visualization lets us understand what features a particular unit in a neural network has learned: for example, what features is a convolutional layer looking at? What features is this neuron looking at?
To perform feature visualization, we exploit the differentiability of our neural networks.

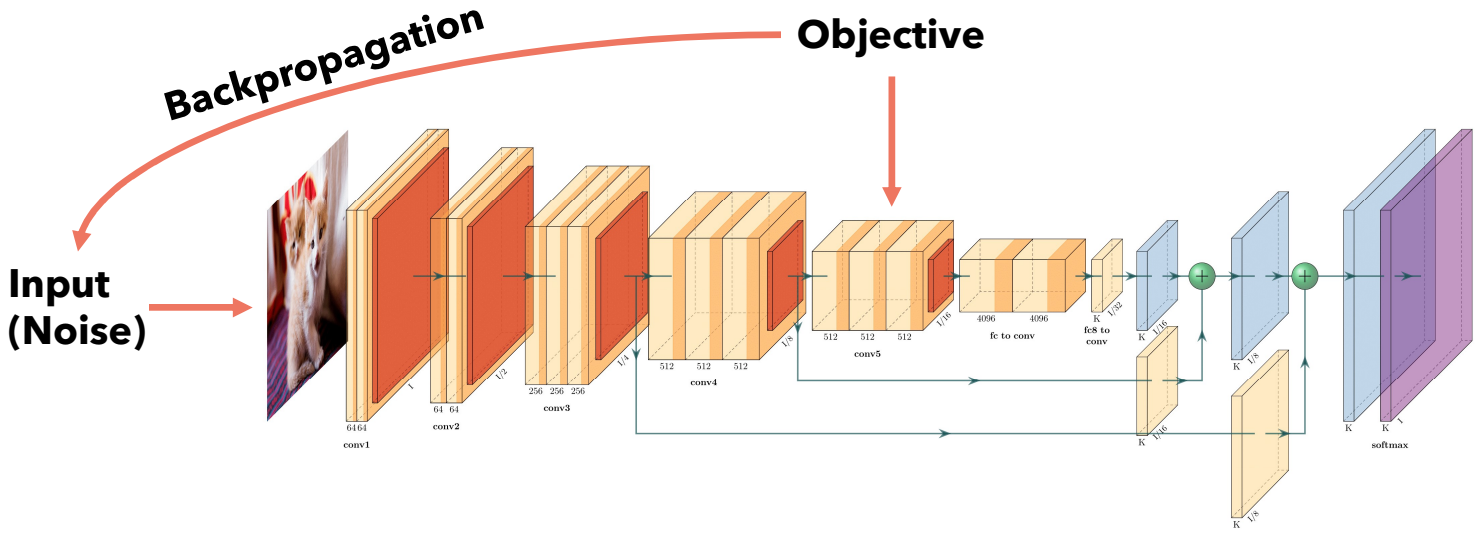


Source: Interpretable Machine Learning, Christoph Molnar

Feature visualization: an optimization problem

Imagine we have a convolutional neural network, and we want to understand what kind of features a layer of interest looks for. Since neural networks are, generally, differentiable with respect to their inputs, we can use a layer's activations as objective and optimize the input to maximize this objective.

We usually start from random noise.

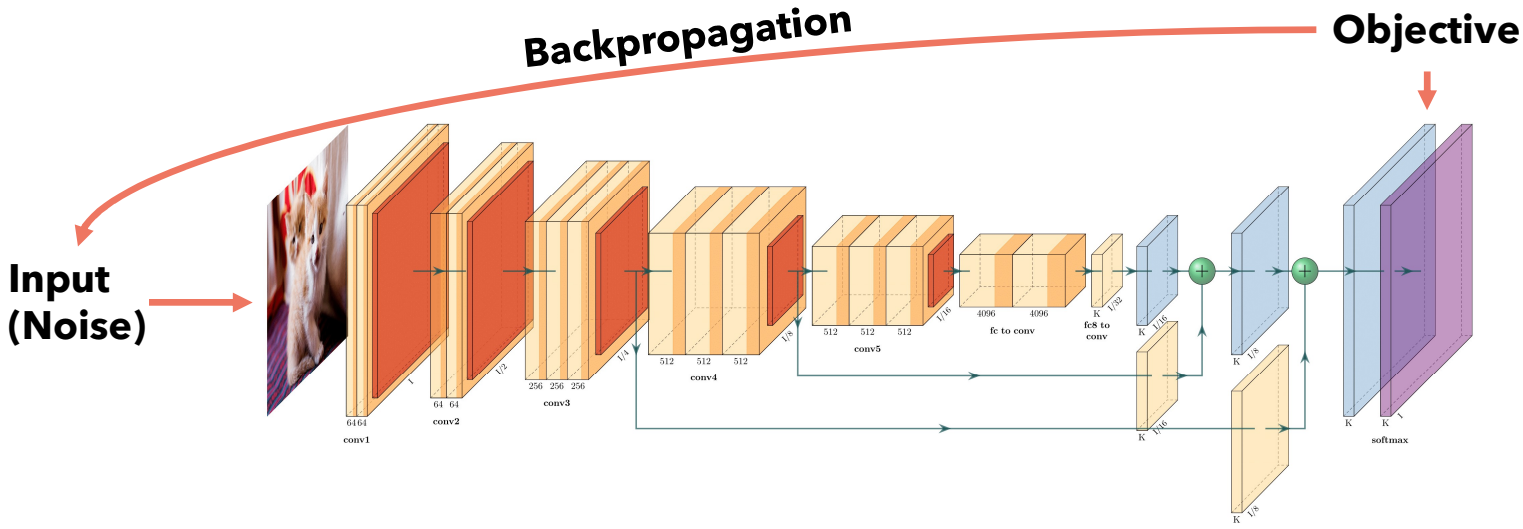


Source: PlotNeuralNet repository.

Feature visualization: optimizing for logits

Imagine we want to understand what kind of image activates a certain output: we can create a loss like the one we used for generating the adversarial example and optimize an initial noisy image to maximize a particular output class.

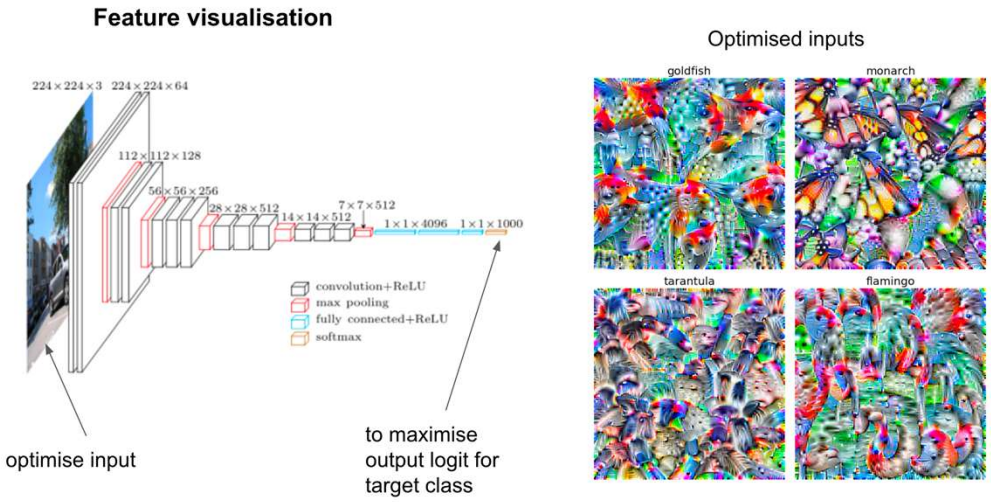
Wait, is it THAT simple?



Source: PlotNeuralNet repository.

Feature visualization: not so easy actually

Unfortunately, when optimizing from random noise, we end up having high-frequency features that don't look very natural.



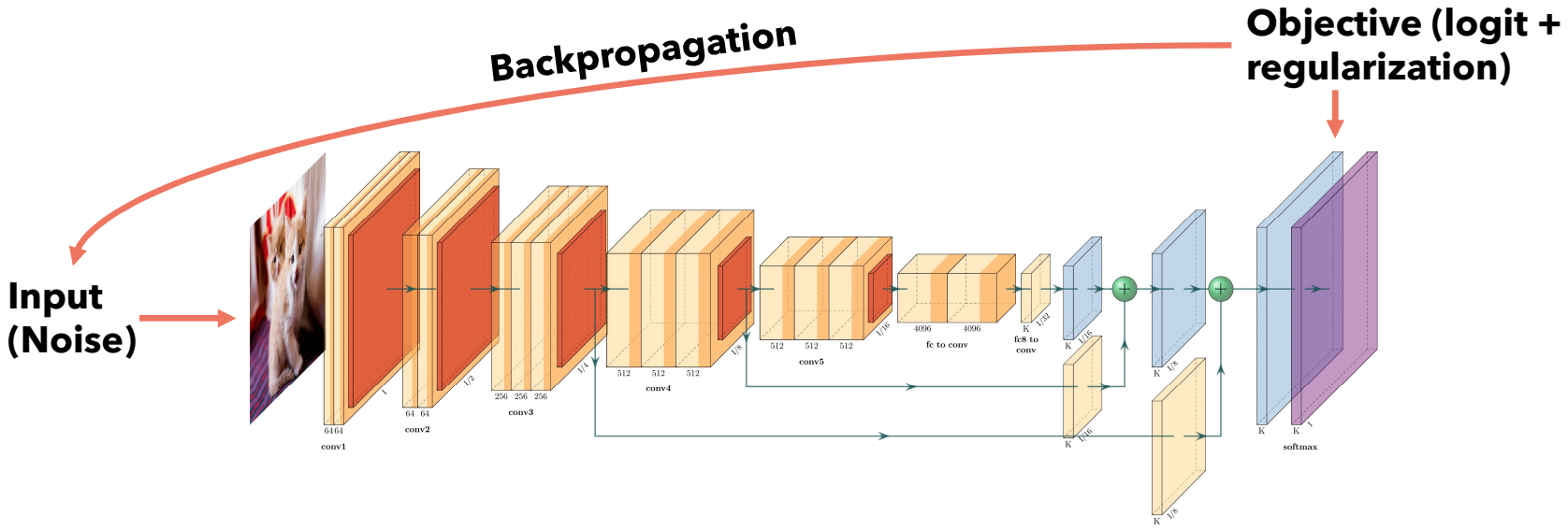
Source: Tim Sainburg.

What can we do to force the optimization process to produce more naturally looking images?

Let's talk about regularization!

Let's introduce regularization

Since we have an optimization problem, we can add more "constraints" to our objective to make the gradient move in directions that exhibit more of a certain pattern or less of others. **Let's see some examples of regularization.**



Source: PlotNeuralNet repository.

Transformation robustness

Frequency penalization adds a penalty term to the objective term for penalizing high variance between neighboring pixels (or the overall variance). The intuition is that pixels close to each other should be "similar". The problem is that this may penalize also edges (for example the sudden change between a person's clothes and the background in a photo).

Transformation robustness tries to find input examples that maximize a target objective even if they're altered with transformations such as rotation, translation, etc.

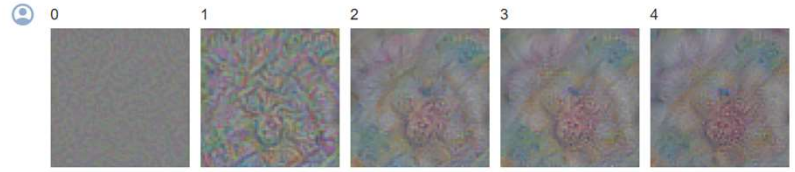
Frequency Penalization

```
L1   = -0.05
TV   = -0.25
BLUR = -1.0

obj  = objectives.channel("mixed4b_pre_relu", 452)
obj += L1 * objectives.L1(constant=.5)
obj += TV * objectives.total_variation()
obj += BLUR * objectives.blur_input_each_step()

imgs = render.render_vis(model, obj,
                         transforms=[],
                         param_f=lambda: param.image(64, fft=False, decorrelate=False),
                         thresholds=(1, 32, 128, 256, 2048), verbose=False)


# Note that we're doubling the image scale to make artifacts more obvious
show([nd.zoom(img[0], [2,2,1], order=0) for img in imgs])
```



Transformation Robustness

```
JITTER = 1
ROTATE = 5
SCALE  = 1.1

transforms = [
    transform.pad(2*JITTER),
    transform.jitter(JITTER),
    transform.random_scale([SCALE ** (n/10.) for n in range(-10, 11)]),
    transform.random_rotate(range(-ROTATE, ROTATE+1))
]

imgs = render.render_vis(model, "mixed4b_pre_relu:452", transforms=transforms,
                         param_f=lambda: param.image(64),
                         thresholds=(1, 32, 128, 256, 2048), verbose=False)


# Note that we're doubling the image scale to make artifacts more obvious
show([nd.zoom(img[0], [2,2,1], order=0) for img in imgs])
```

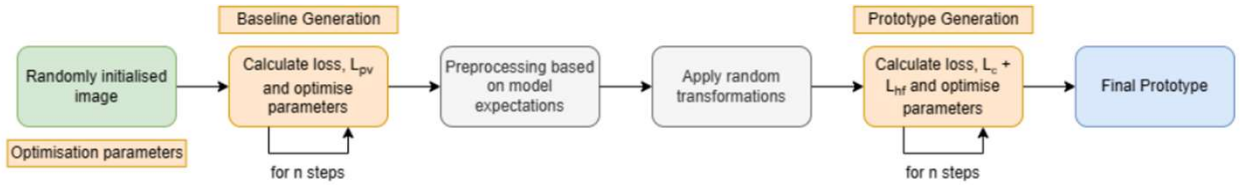
Source: Lucid - Regularization Colab Notebook.

Why Leap Labs' prototypes look so natural?

The answer is in the paper published by Leap Labs: **Prototype Generation: Robust Feature Visualisation for Data Independent Interpretability**.

3 Prototype Generation

For a given model M, we define a prototype P as an input that maximally activates the logit corresponding to c, while keeping the model's internal activations in response to that input close to the distribution of 'natural' inputs. Let $\mathbb{I}$ represent the set of all possible natural inputs that would be classified by model M as belonging to class c. We aim to generate a prototype P such that it aggregates the representative features of a majority of inputs in $\mathbb{I}$. Formally, we posit that the activations $\mathbf{A}_P$ of P are 'closer' to the mean activations $\mathbf{A}_{\mathbb{I}}$ of all $I \in \mathbb{I}$ than any individual natural image I across all layers $\mathbb{L}$ in M. We measure 'closeness' between $\mathbf{A}_P$ and $\mathbf{A}_{\mathbb{I}}$ using spearman correlation.



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Interpretability for Language Models

Can we apply the techniques that we've seen today also to language models? That is, given a desired output, can we find what kind of prompt would generate it?

Let's review how language models works

Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Short intro to language models

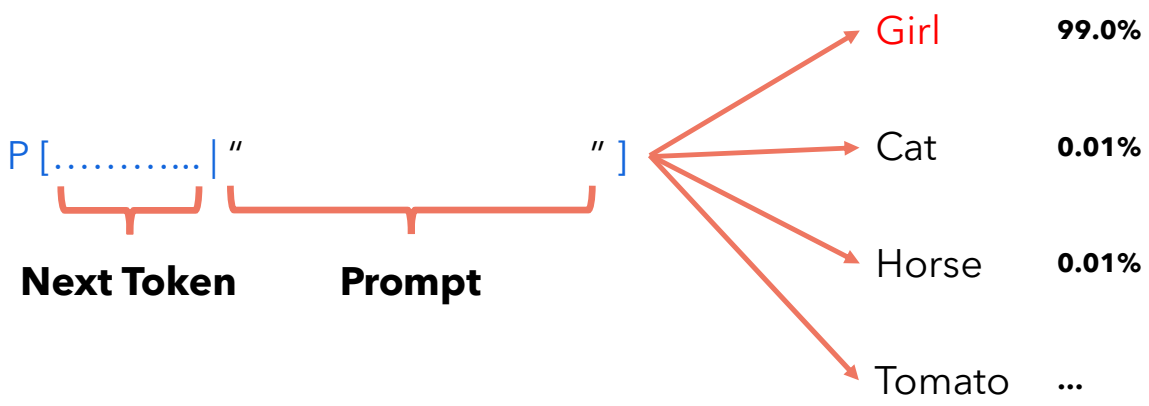
A language model is a probabilistic model that assign probabilities to sequence of words.
In practice, a language model allows us to compute the following:

A language model outputs a list of probabilities, one for every token in the vocabulary, indicating how likely a token is the next one.

What does my language model think of girls?

Imagine we have a pre-trained language model, our goal is to find the input prompt, of a certain length (let's say 3 tokens), that results in a specific output.

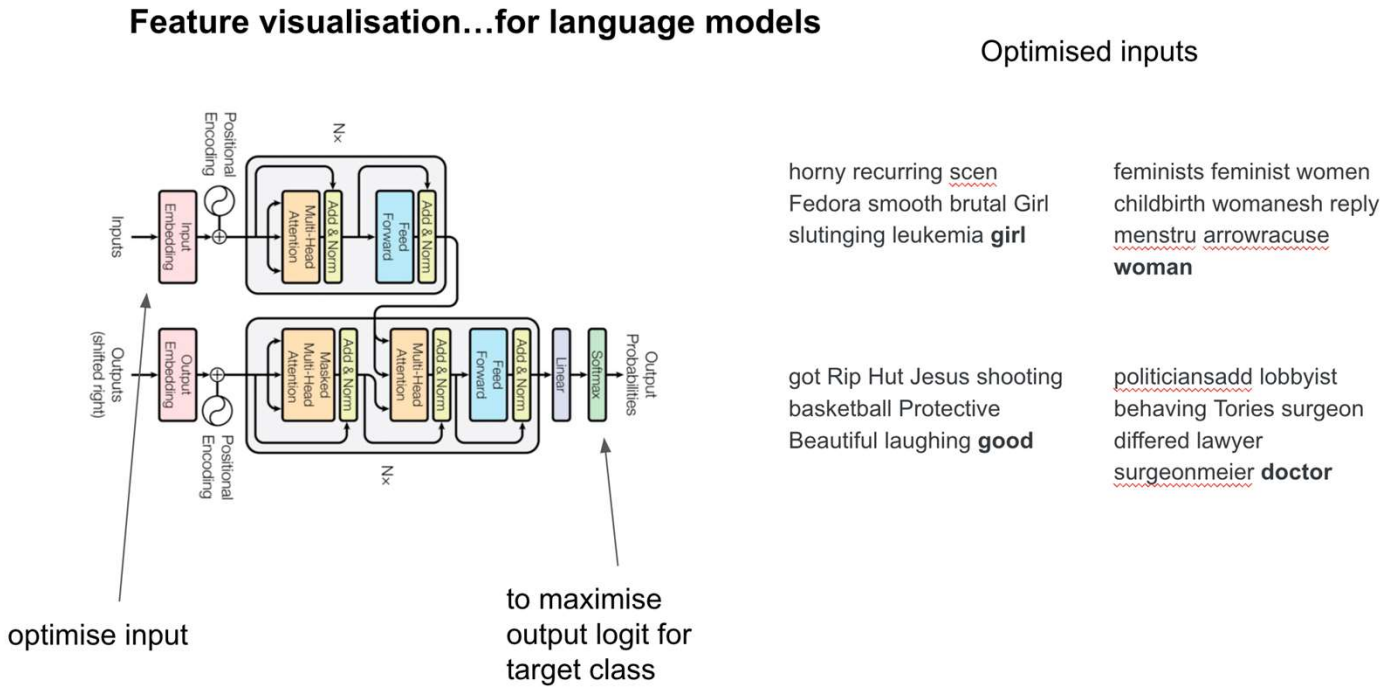
We could proceed exactly how we did for vision models, that is, calculate the gradient of the output prediction with respect to the input, run backpropagation and optimize the input to maximize a particular output. **However, this is not possible, because the input is made of discrete tokens.**



Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

Let's see some examples

Jessica Rumbelow, founder of Leap Labs, published some interesting results from the GPT2-xl model. They used the algorithm described to find prompts that result in a particular output.

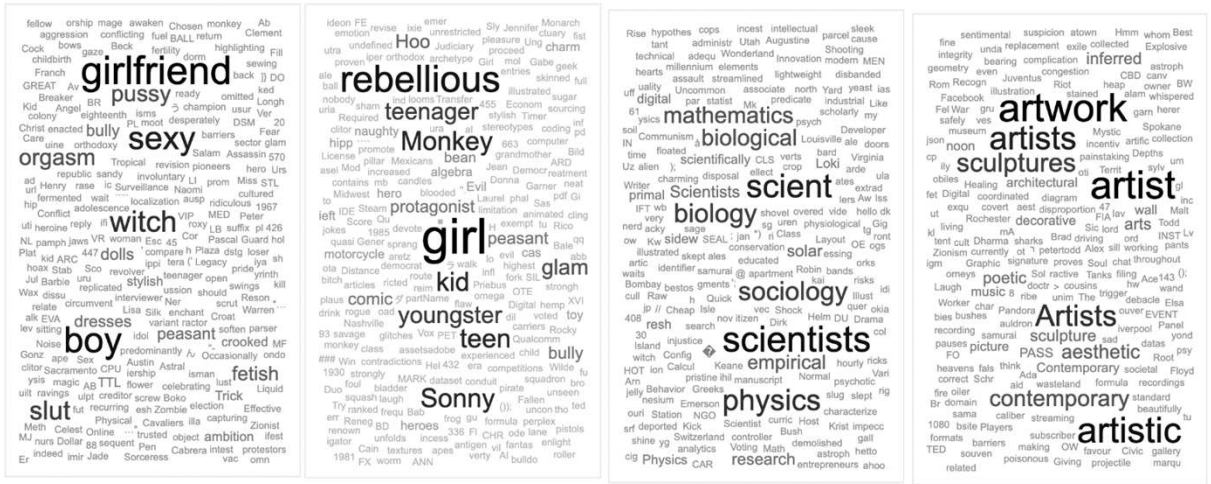


Source: Jessica Rumbelow.

Let’s see some examples

Jessica Rumbelow, founder of Leap Labs, published some interesting results from the GPT2-xl model. They used the algorithm described to find prompts that result in a particular output.

Aggregated token frequencies in optimised inputs



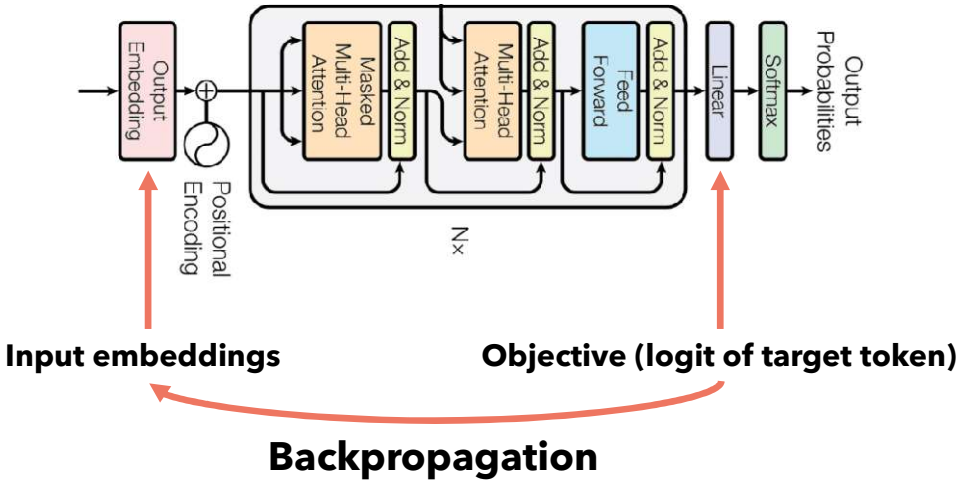
‘ girl’ ‘ boy’ ‘ science’ ‘ art’

Source: Jessica Rumbelow.

Umar Jamil – https://github.com/hkproj/ml-interpretability-notes

… what about embeddings?

Instead of maximizing for the input tokens, which are also known as *input_ids*, we could optimize the input embeddings to maximize a specific output. We start from random embeddings and keep optimizing to predict for a specific output token.



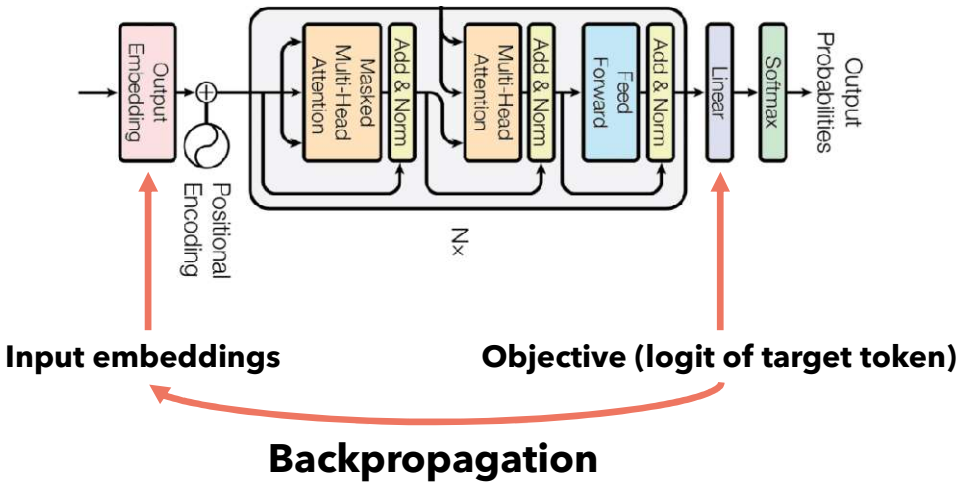
We still have two problems:

1. The embeddings that we are optimizing may not correspond to any embedding in the model's dictionary
2. We need to force the model to make the input converge to an embedding that's present in the model's dictionary.

K-NN and regularization

To understand what token the optimized input embeddings correspond to, we can use an algorithm like K-Nearest-Neighbor to select the closest embedding from the model's dictionary.

To force the model to optimize the input such that it resembles one of the embeddings in the model's dictionary, we can add an additional term in the loss function to minimize the distance between each embedding and their closest neighbor. This will act as **regularization**.



Thanks for watching!
Don't forget to subscribe for more amazing content on AI and Machine Learning!

Umar Jamil – https://github.com/hkproj/ml-interpretability-notes