amazon | science

MACHINE LEARNING

The science behind SageMaker's cost-saving Debugger

New tool can spot problems — such as overfitting and vanishing gradients — that prevent machine learning models from learning.

By Nathalie Rauschmayr, Krishnaram Kenthapadi, Miyoung Choi

April 02, 2021

Share

A machine learning training job can seem to be running like a charm, while it's really suffering from problems such as overfitting, exploding model parameters, and vanishing gradients, which can compromise model performance. Historically, spotting such problems during training has required the persistent attention of a machine learning expert.

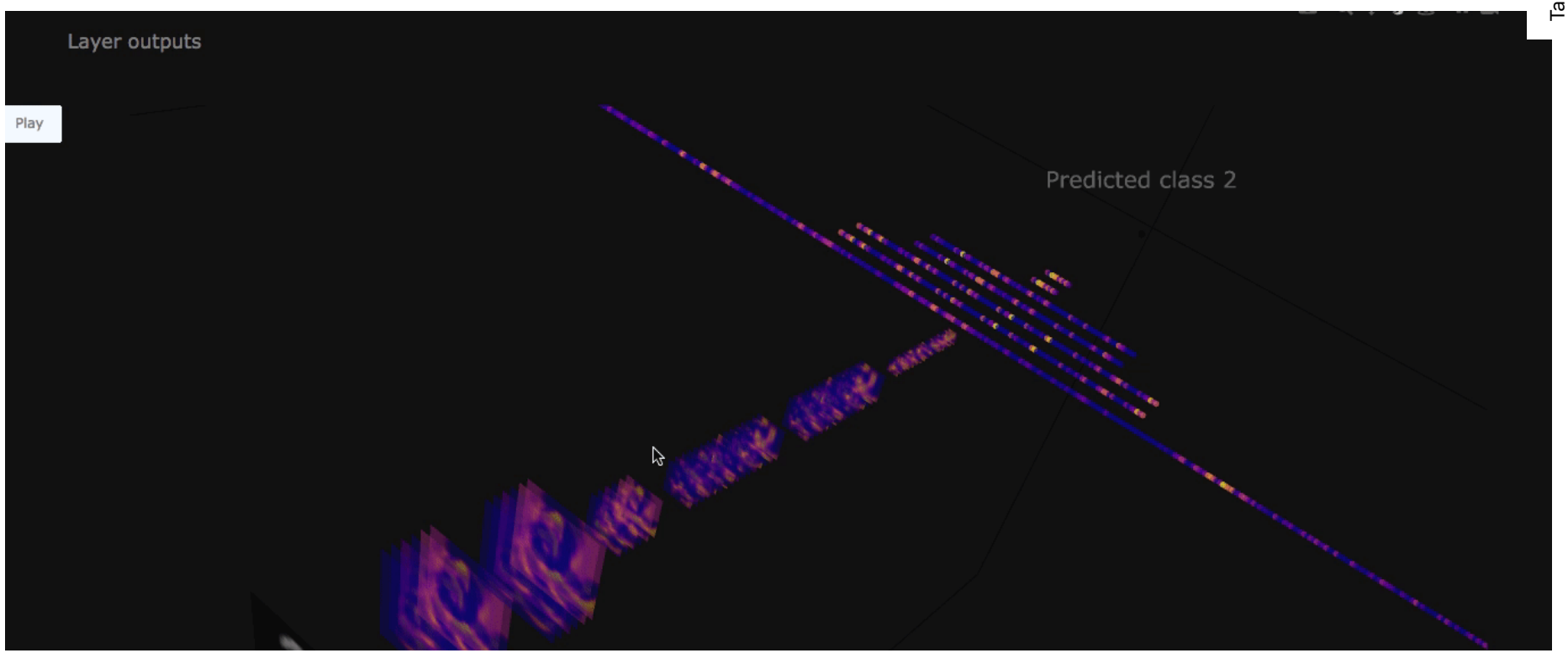
The Amazon SageMaker team has developed a new tool, SageMaker Debugger, that automates this problem-spotting process, saving customers time and money. For example, by using Debugger, one SageMaker customer reduced model size by 45% and the number of GPU operations by 33%, while improving accuracy.

Next week, at the Conference on Machine Learning and Systems (MLSys), we will present a paper that describes the

technology behind SageMaker Debugger.

Output tensors and rules

When debugging a neural network model, Debugger collects *output tensors* to capture the model's various states throughout training. A tensor is a higher-dimensional analogue of a matrix, and by default, Debugger's output tensors can include values such as loss, or how far short of its target the model output falls; the outputs of each layer; the weights on the connections between layers; and, when the model is being updated during optimization, the weight gradients, or the direction in which the weights should be tuned to minimize loss. Customers can also design their own customized output tensors.

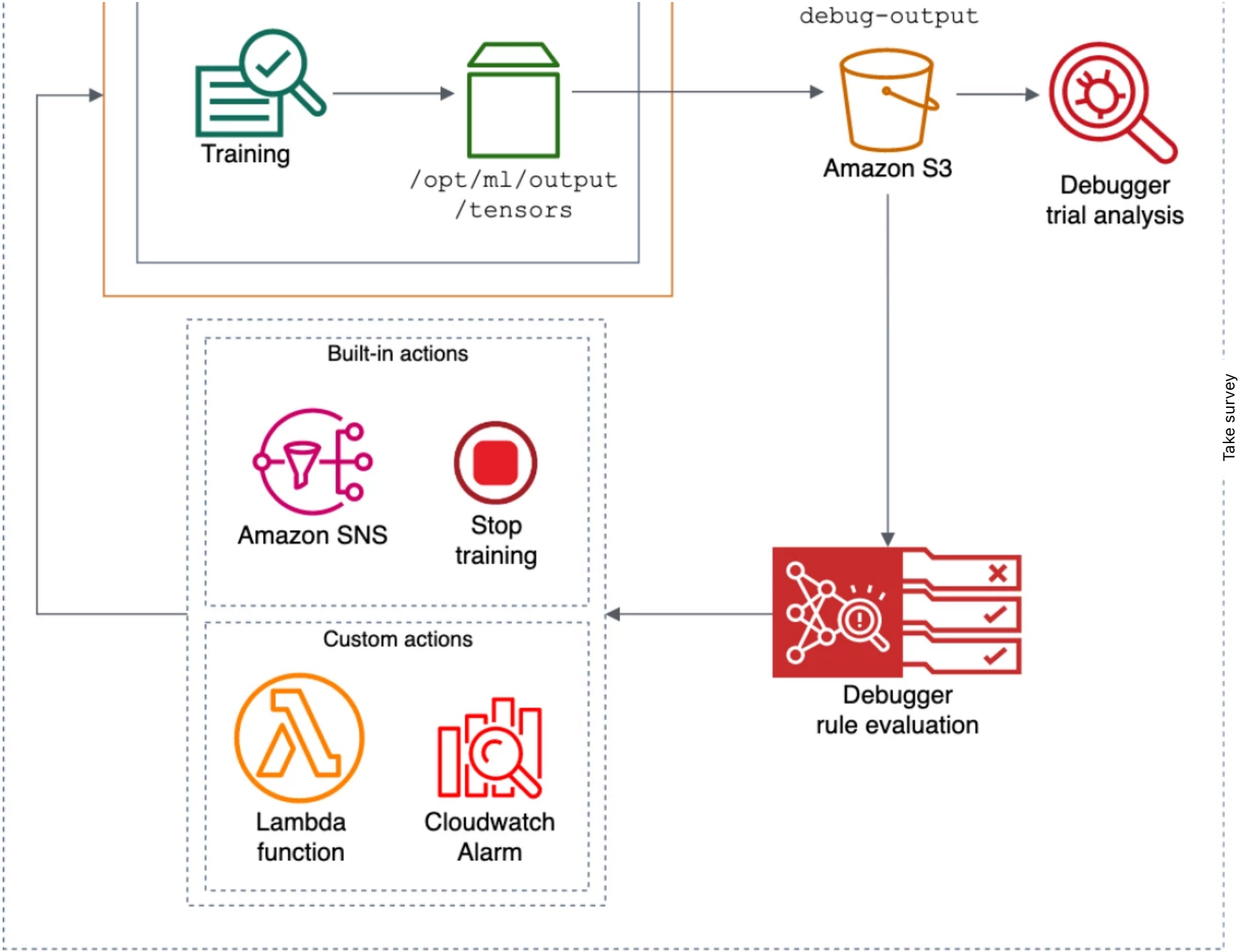


A visualization of activation output tensors captured by SageMaker Debugger for a deep-learning model trained on the MNIST dataset of handwritten digits.

Debugger automatically applies a set of rules to the output tensors to ensure that the model is learning. The rules monitor things like changes in the absolute size of individual weights, the relative size of the gradients across layers, and the number of individual network nodes — or neurons — with zero outputs. And again, customers can add thei own rules as well.

Debugger also provides rules for decision tree models built using XGBoost. For example, Debugger can check the depth of individual trees in an ensemble; with larger tree depth, the model is prone to overfitting, or failing to generalize to data outside the training set.

Debugger architecture


Training
/opt/ml/output
/tensors
debug-output
Amazon S3
Debugger
trial analysis
Built-in actions
Amazon SNS
Stop
training
Custom actions
Lambda
function
Cloudwatch
Alarm
Debugger
rule evaluation
Take survey

The architecture diagram of Debugger, as applied to an Amazon SageMaker training workflow.

The diagram at right shows the Debugger workflow on Amazon SageMaker. Debugger captures output tensors from a training job in progress and uploads them to an Amazon S3 storage bucket. Debugger rules run on a separate *instance*, or allotment of computing capacity in the cloud, so the analysis does not interfere with the training. This helps ensure Debugger's scalability.

By default, Debugger can perform certain actions when it finds problems, such as notifications via text or e-mail or the interruption of training jobs. Users can also play with CloudWatch events and Lambda functions to create their own automated actions.

Bug spotting

Problems can arise at any point in the machine learning lifecycle, but some of the most common are data imbalances, bad initialization, vanishing/exploding gradients (including neuron saturation and dead ReLUs), and overfitting. Debugger's built-in rules check for all of those problems — and more.

Data imbalances

During data preparation, we need to ensure that the data is correctly preprocessed and normalized and that it contains representative samples. If the data contains too many correlated features or is not normalized, the model is likely to overfit.

Targeted rule: Debugger checks for balance between the different classes of data in the training set. It can also verify whether data has been correctly normalized by checking for zero mean and unit variance.

Bad initialization

Initialization assigns random values to model parameters. If all parameters have the same initial value, they receive the same gradient, and the model is unable to learn. Initializing parameters with values that are too small or too large may lead to vanishing or exploding gradients.

Targeted rule: At the start of model training, Debugger checks that weights connected to the same neuron do not have the same initial values. Debugger also checks that the variance of the weights per layer does not exceed a threshold.

Vanishing/exploding gradients

Deep neural networks typically learn through back-propagation, in which the model's loss is traced back through the network. Neurons' weights are modified in order to minimize loss. If the network is too deep, however, the learning algorithm can spend its whole loss budget on the top layers, and weights in the lower layers never get updated. That's the vanishing-gradient problem.

Conversely, the learning algorithm might trace a series of errors to the same neuron, resulting in such a large modification to that neuron's weight that it imbalances the network. That's the exploding-gradient problem.

Targeted rule: Debugger monitors statistical properties of the gradients and raises an alarm if they cross a predefined threshold.

Neuron saturation/dead ReLUs

One of the most common causes of vanishing gradients is neuron saturation. Each neuron in a neural network has an activation function, which determines whether it "fires" — produces an output — in response to particular inputs. Some activation functions, such as sigmoid and tanh, can lead to neuron saturation, in which large changes in inputs produce small changes in outputs. The weights of saturated neurons are, essentially, impossible to update.

To prevent neuron saturation, many state-of-the-art models use the ReLU activation function. The output of the ReLU function increases linearly with inputs above some threshold but is zero otherwise. Such models instead run the risk of the dying-ReLU problem: the gradients vanish because the activation outputs go zero.

Targeted rule: To identify neuron saturation, Debugger checks the activation outputs; to identify dead ReLUs, it counts how many neurons in a model output zero values.

Overfitting

The training loop consists of training and validation. If the model's performance improves on a training set but not on a validation set, it's a clear indication of overfitting. If the model's performance initially improves on the validation set but then begins to fall off, training needs to be stopped to prevent overfitting.

Targeted rule: Debugger checks whether the ratio between validation loss and training loss exceeds a threshold.

These are just some of the rules built into Debugger; the full list is in the table below.

Problem class	Rules

Datasets	Class imbalance Data not normalized Ratio of tokens in sequence
Loss and accuracy	Loss not decreasing Overfitting Underfitting Overtraining Classifier confusion
Weights	Poor initialization Updates too small
Gradients	Vanishing Exploding
Tensor	All values zero Variance of values too small Values not changing across steps
Activation function	Tanh saturation Sigmoid saturation Dying ReLU
Decision trees	Depth of tree too large Low feature importance

Customers can also use Debugger's API to get real-time insights into their models, plotting weight distributions, visualizing the low-dimensional latent space of t-SNE (as in the animation above), creating saliency maps, and the like.

To get started with Debugger, check out our GitHub repo and install the smdebug library from PyPI. We have a rich list of examples that show the usage of Debugger for beginner, intermediate, and advanced use cases. For further details on Debugger, check out the MLSys Debugger paper and the SageMaker Debugger developer guide.

Research areas

Machine learning

Tags

Amazon SageMaker

Conference

MLSys 2021

Related publications

Amazon SageMaker Debugger: A system for real-time insights into machine learning model training

ABOUT THE AUTHOR

Nathalie Rauschmayr

Nathalie Rauschmayr is an applied scientist with Amazon Web Services.

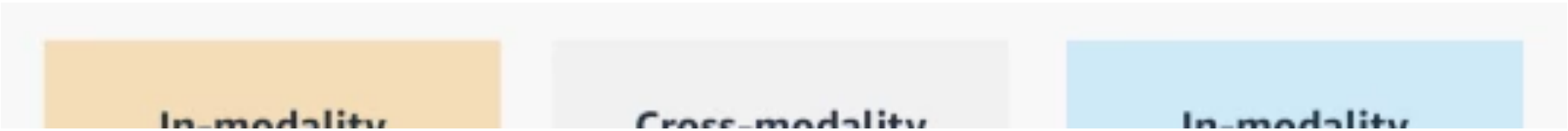
Krishnaram Kenthapadi

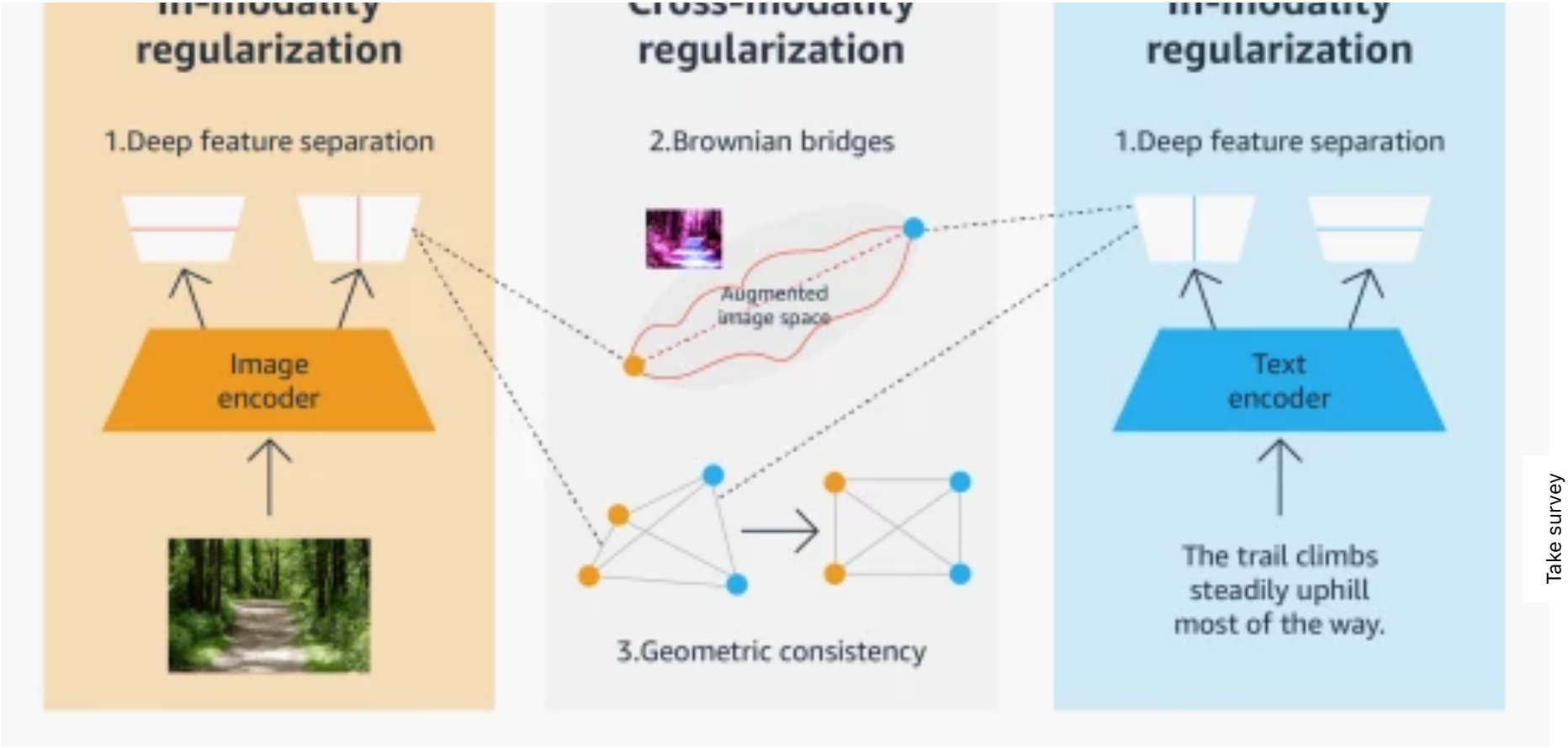
Krishnaram Kenthapadi is a principal scientist with Amazon Web Services.

Miyoung Choi

Miyoung Choi is a programmer writer with Amazon Web Services.

Related content



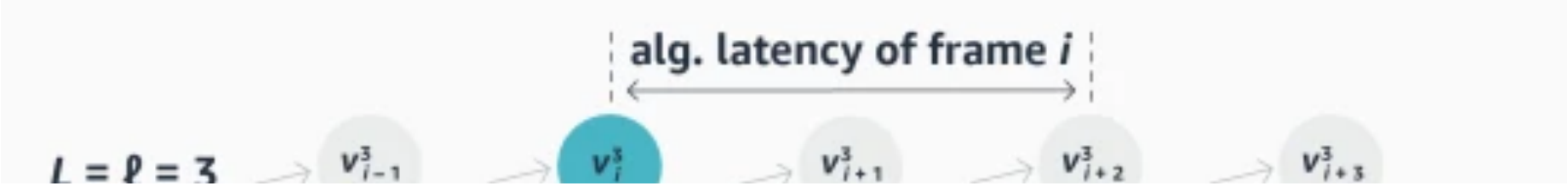


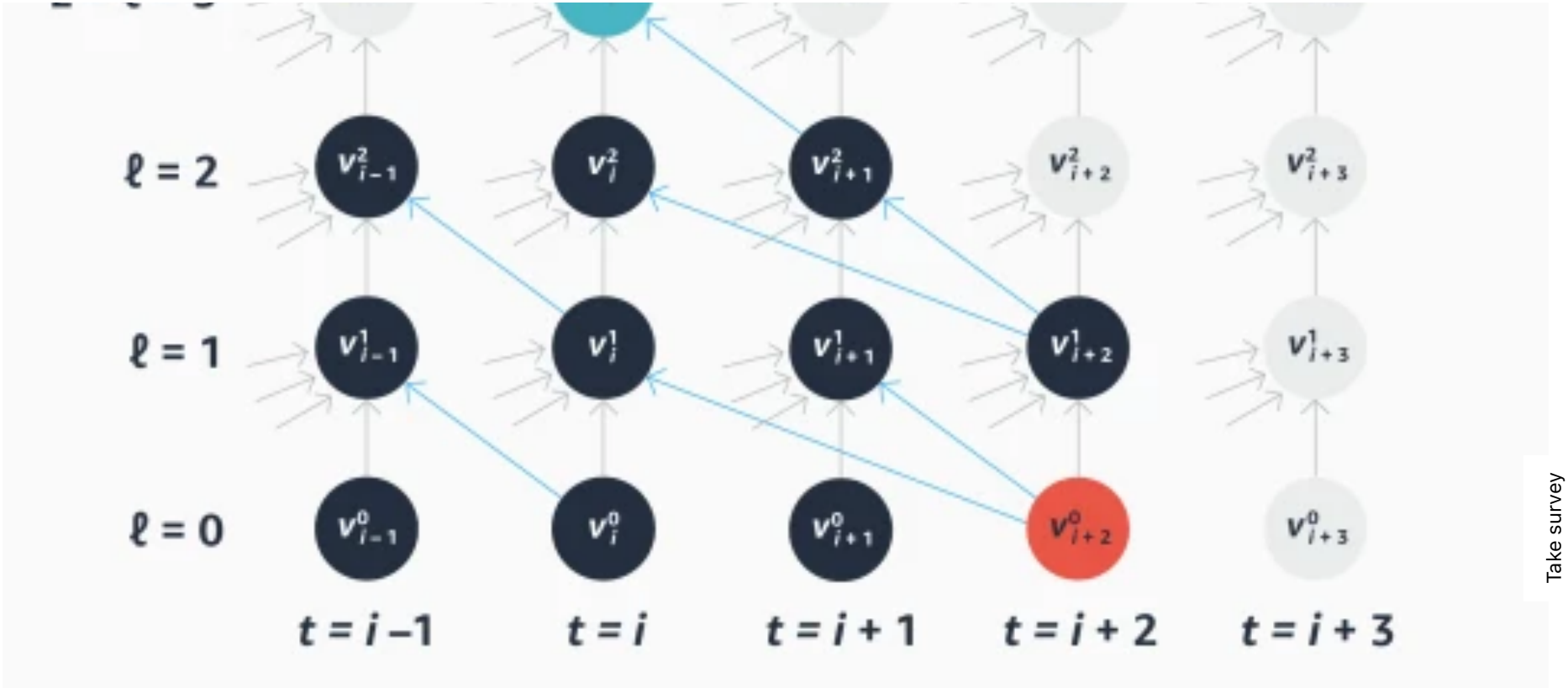
New contrastive-learning methods for better data representation

Changyou Chen
October 04, 2023

New loss functions enable better approximation of the optimal loss and more-useful representations of multimodal data.

MACHINE LEARNING



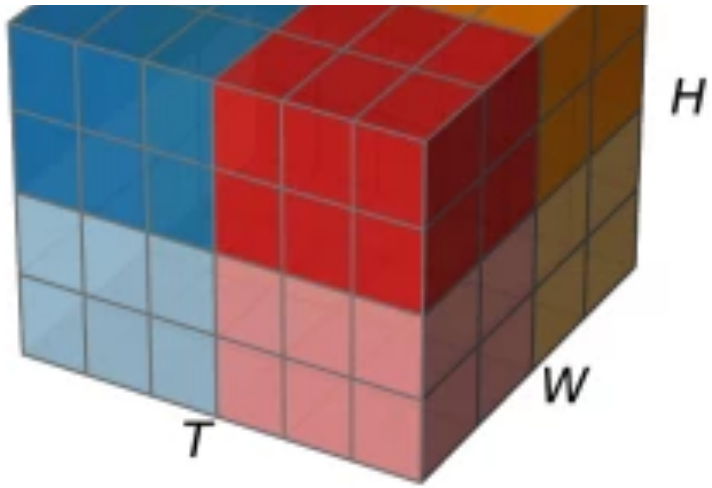


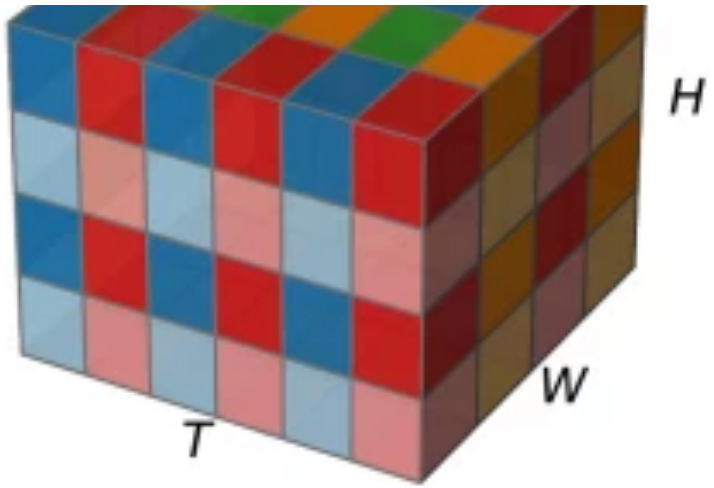
How dynamic lookahead improves speech recognition

Grant Strimel, Yi Xie, Brian King
July 27, 2023

Determining on the fly how much additional audio to process to resolve ambiguities increases accuracy while reducing latency relative to fixed-lookahead approaches.

CONVERSATIONAL AI





(a) strategy="local" shift=(0, 0, 0)

(b) strategy="dilated" shift=(0, 0, 0)

Take survey

Making deep learning practical for Earth system forecasting

Zhihan Gao, Boran Han
September 15, 2023

Novel "cuboid attention" helps transformers handle large-scale multidimensional data, while diffusion models enable probabilistic prediction.

MACHINE LEARNING

Work with us

See more jobs

About
Research areas
Blog
News and features
Publications
Code and datasets
Collaborations
Careers

Alexa Prize
Academics
Research Awards
About Amazon
Amazon Developer
Amazon Web Services
Newsletter
FAQs