Apache Beam and Ensemble Modeling: A Winning Combination for Machine Learning

Shubham Krishna
ML Engineer, ML6

BEAM SUMMIT
NYC 2023

Who is ML6?

Machine Learning services company.

We help our clients build machine learning applications using technologies such as Apache Beam.

Credits

Philippe Moussalli
Machine Learning Engineer, ML6

Agenda

- Motivation
  - Ensemble Modeling for solving complex use-cases
- Solution
  - **Beam RunInference**:
    - Seamless integration of ML in a Beam pipeline for semantic enrichment
    - Use multiple Runinference transforms for pipelines with multiple ML models
- Example

Motivation

- Semantic Enrichment: ML models provide semantic information.

- Business needs often involve the use of multiple machine learning models, each addressing a specific subtask and contributing unique capabilities.

Semantic Enrichment of Data

- Categorise: Add specific label
- Summarize
- Sentiment Analysis
- Translate
- Extract important keywords
- Image Annotation
- Image Captioning
- Speech Recognition
- .....

Ensemble Modeling

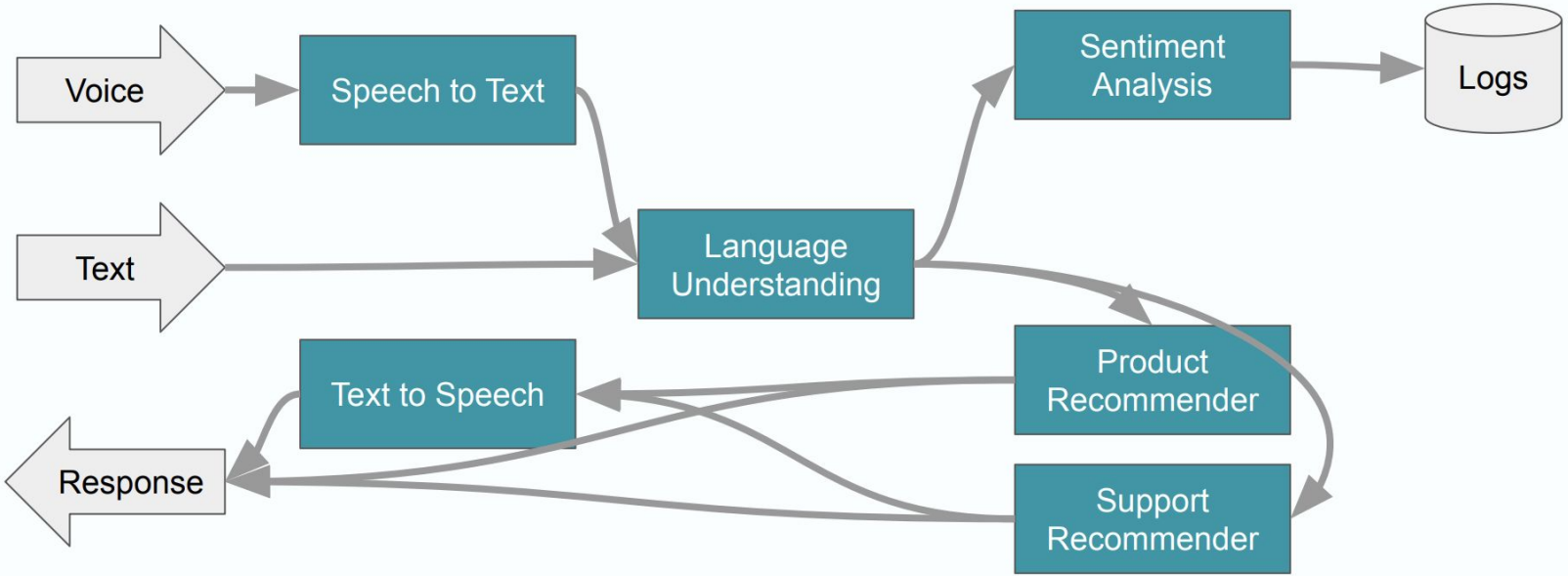
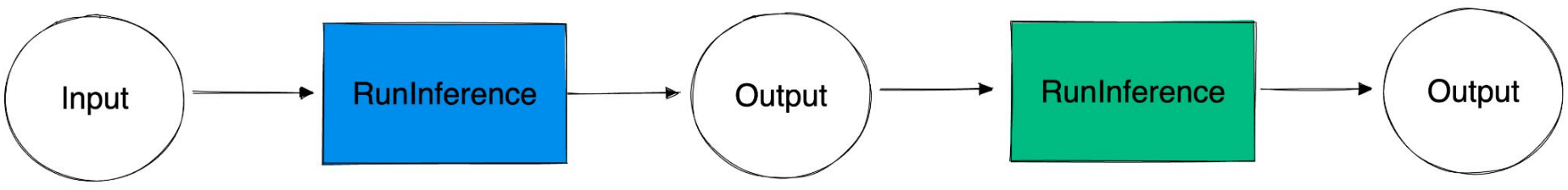
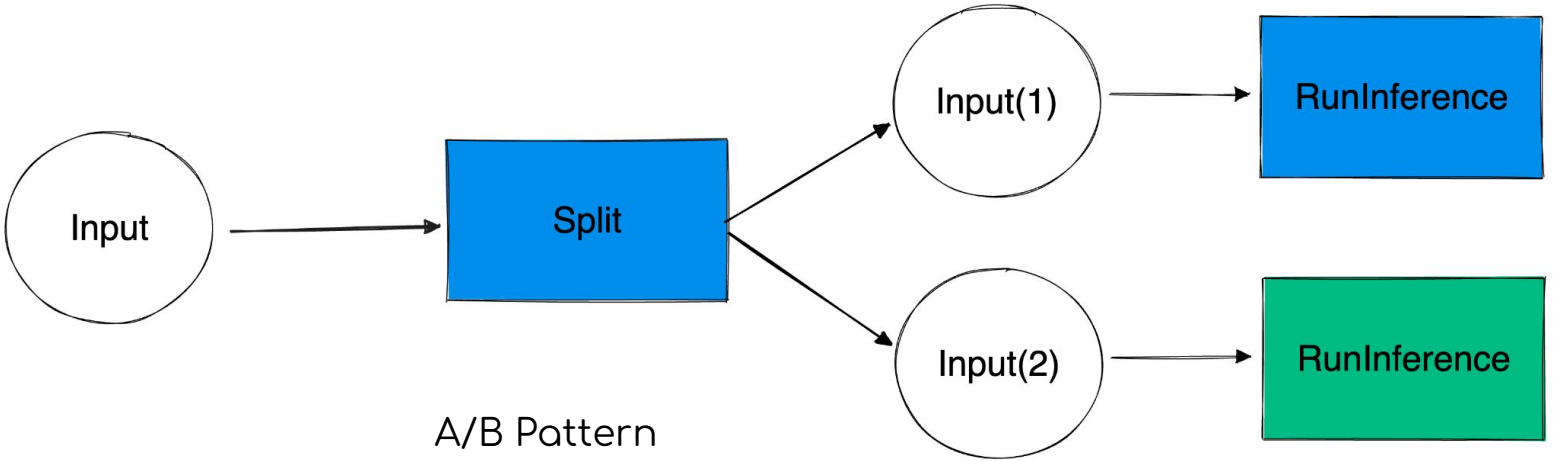


Fig.1. Example of a Multi model pipeline, taken from a tutorial on RunInference on Dataflow: Link

Ensemble Modeling: Sequential vs A/B



Sequential Pattern



A/B Pattern

Problem

Seamlessly integrate ML models in a Beam pipeline for semantic enrichment of data.

Business needs require combining multiple ML models. (Ensemble Modeling)

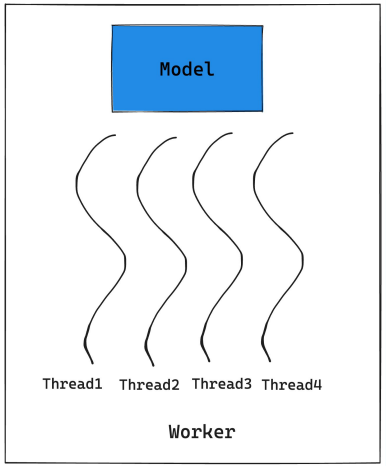
Solution

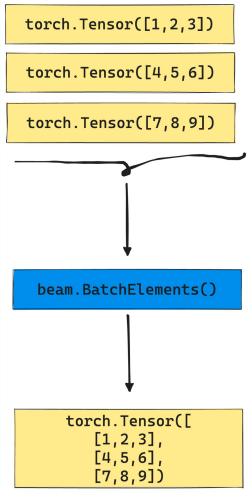
RunInference API = Inference with ML model in batch and streaming pipelines, without needing lots of boilerplate code.

RunInference API = Using multiple RunInference transforms, build a pipeline that consists of multiple ML models.

RunInference >> Custom DoFn

Seamlessly integrate ML model in a Beam pipeline for semantic enrichment of data.





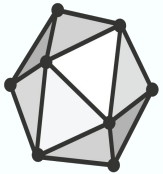
Custom DoFn

RunInference

RunInference supports popular ML frameworks

PyTorch

TensorFlow



ONNX

NVIDIA
TENSORRT

XGBoost

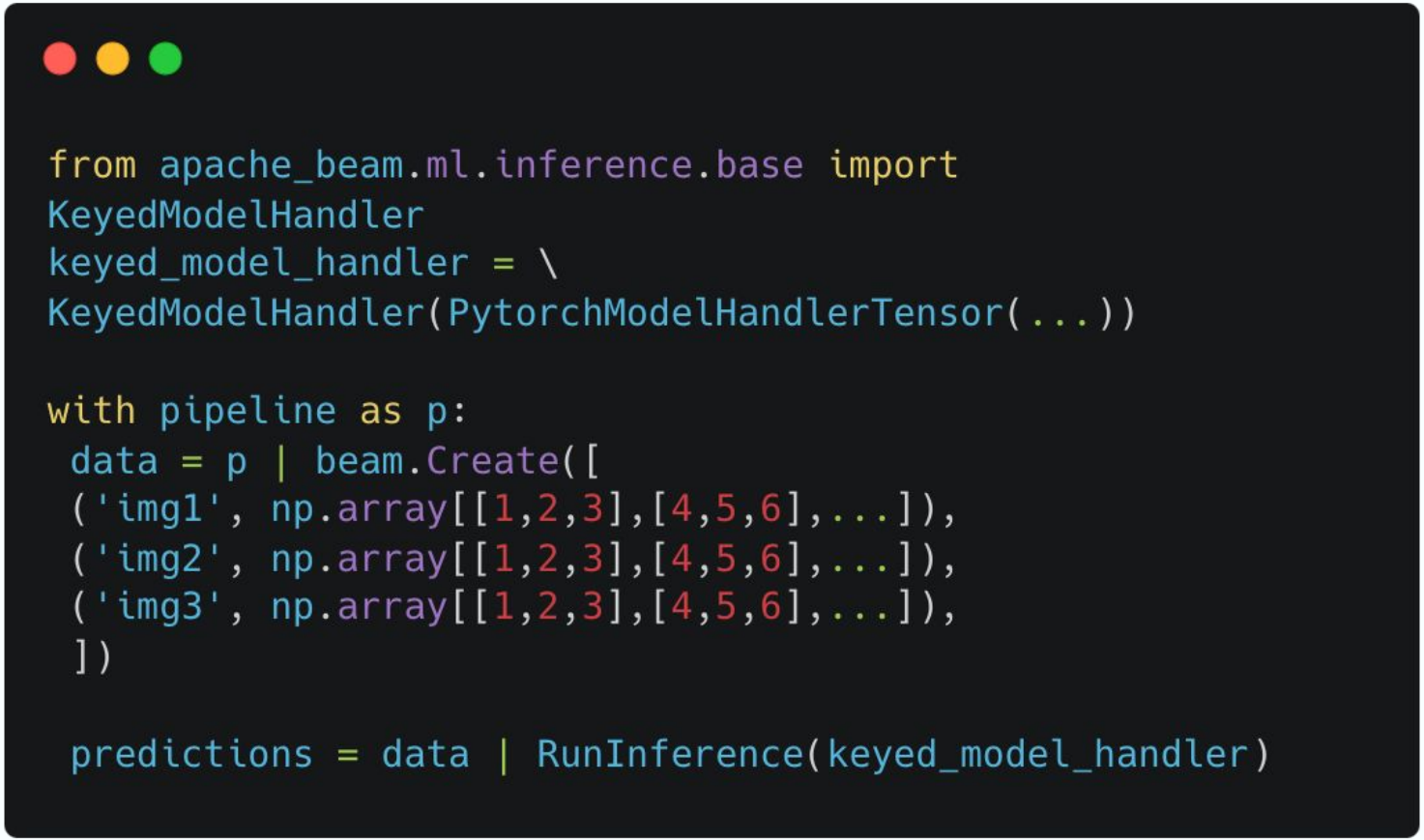
How to use RunInference ?

```
from apache_beam.ml.inference.base import RunInference
with pipeline as p:
 predictions = ( p | beam.ReadFromSource('a_source')
                   | RunInference(ModelHandler)
                 )
```

ModelHandlers

```
from apache_beam.ml.inference.sklearn_inference import SklearnModelHandlerNumpy
from apache_beam.ml.inference.sklearn_inference import SklearnModelHandlerPandas
from apache_beam.ml.inference.pytorch_inference import PytorchModelHandlerTensor
from apache_beam.ml.inference.pytorch_inference import
PytorchModelHandlerKeyedTensor
model_handler = SklearnModelHandlerNumpy(model_uri='model.pkl',
 model_file_type=ModelFileType.JOBLIB)

model_handler = PytorchModelHandlerTensor(state_dict_path='model.pth',
 model_class=PytorchLinearRegression,
 model_params={'input_dim': 1, 'output_dim': 1})
```

KeyedModelHandler

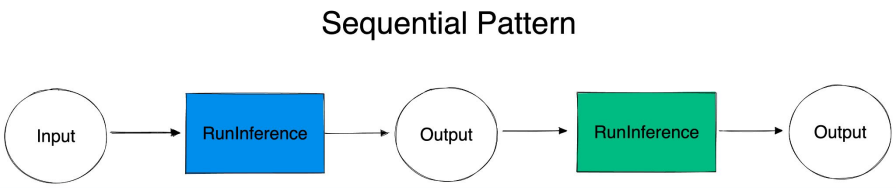


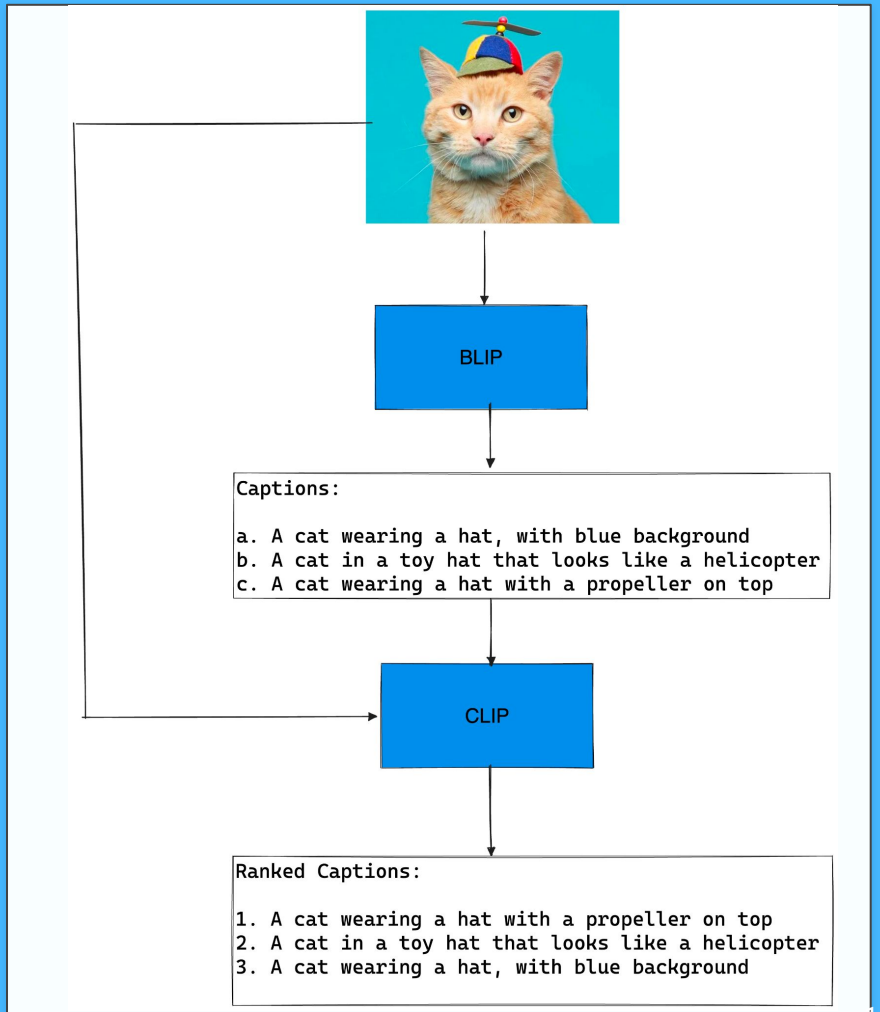
```
from apache_beam.ml.inference.base import
KeyedModelHandler
keyed_model_handler = \
KeyedModelHandler(PytorchModelHandlerTensor(...))

with pipeline as p:
 data = p | beam.Create([
 ('img1', np.array[[1,2,3],[4,5,6],...]),
 ('img2', np.array[[1,2,3],[4,5,6],...]),
 ('img3', np.array[[1,2,3],[4,5,6],...]),
 ])

 predictions = data | RunInference(keyed_model_handler)
```

Example

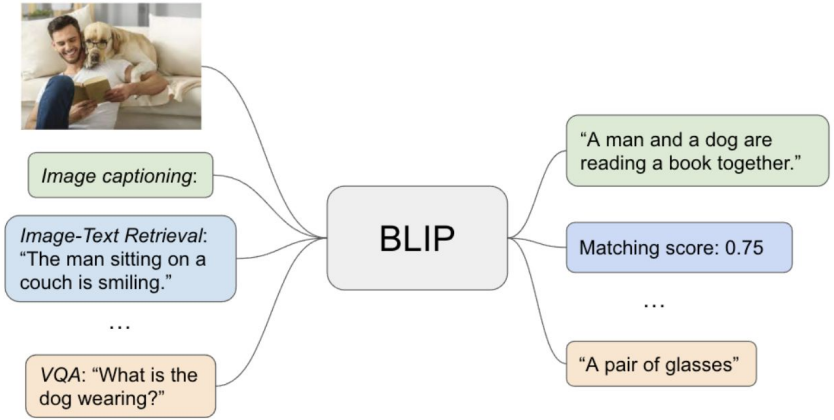
Image captioning and ranking with Sequential Pattern:

1. **BLIP**: Image Captioning
2. **CLIP**: Ranking captions

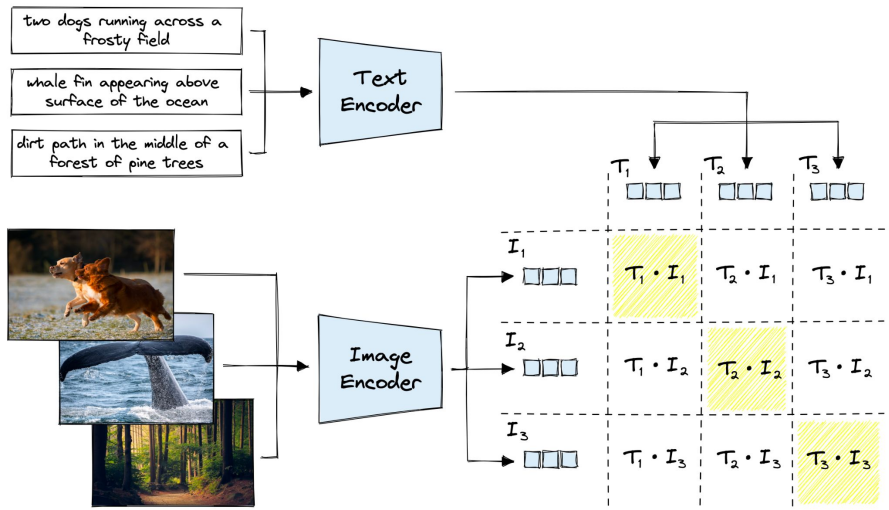




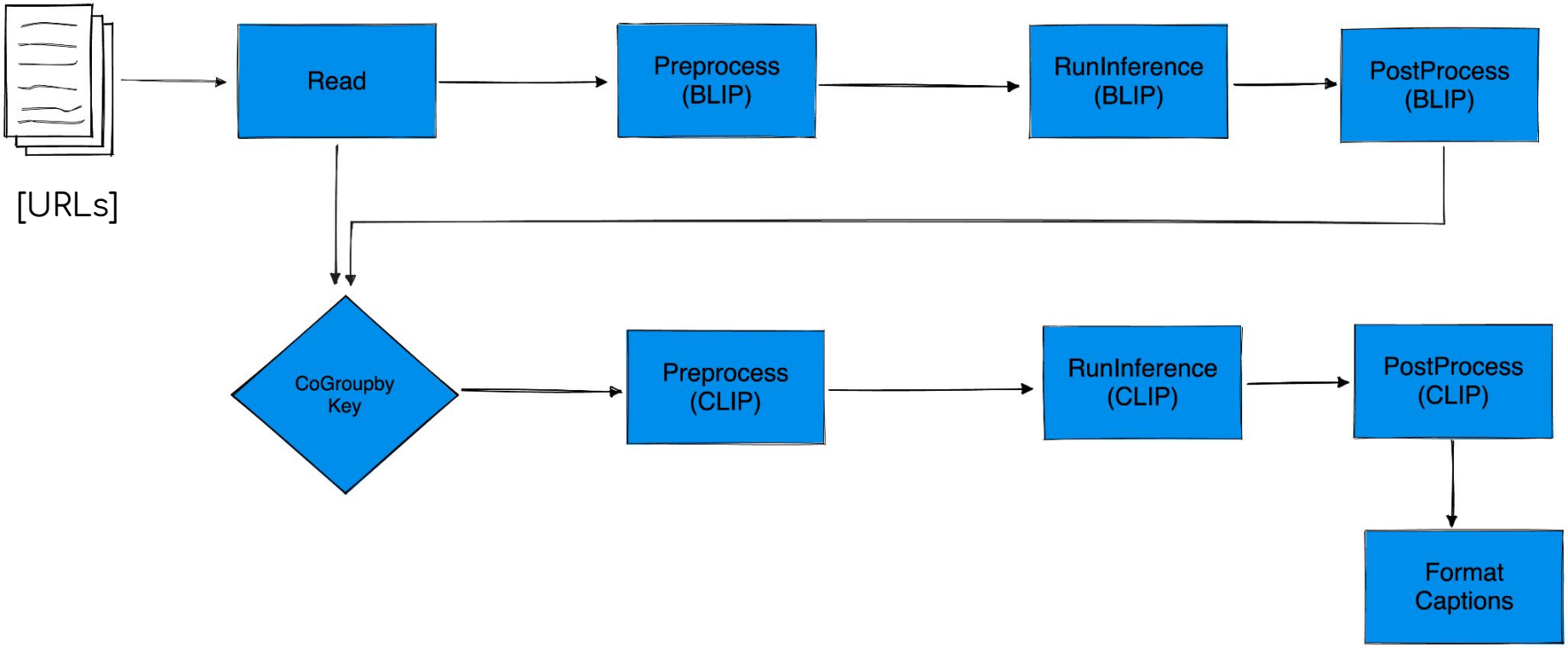
BLIP: Image Captioning



CLIP: Caption Ranking



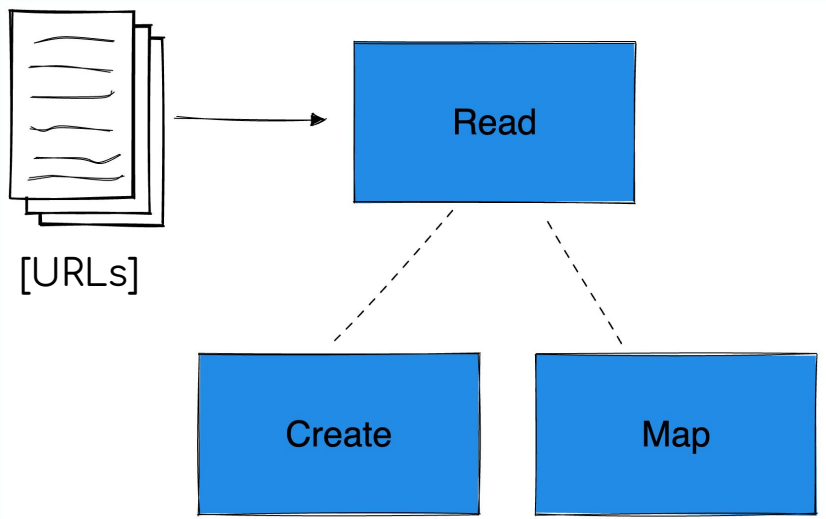
ML Inference Pipeline in Beam as a DAG



ML Inference Pipeline in Beam as a DAG

```
with beam.Pipeline() as pipeline:
    img_url_pil_img = (
        pipeline
        | "ReadUrl" >> beam.Create(images_url)
        | "ReadImages" >> beam.Map(read_img_from_url)
    )

    img_url_captions = (
        img_url_pil_img
        | "BLIPPreprocess" >> beam.MapTuple(lambda img_url, img: (
                img_url,
                blip_preprocess(img, processor=blip_processor),
            )
        )
        | "GenerateCaptions" >> RunInference(
            model_handler=KeyedModelHandler(blip_model_handler),
            inference_args={"max_length": 50, "min_length": 10,
                "num_return_sequences": 5, "do_sample": True,},
        )
        | "BLIPPostProcess" >> beam.ParDo(
          BLIPPostprocess(processor=blip_processor))
    )

    img_url_captions_ranking = (
        ({"image": img_url_pil_img, "captions": img_url_captions})
        | "CreateImageCaptionPair" >> beam.CoGroupByKey()
        | "CLIPPreprocess" >> beam.ParDo(CLIPPreprocess(processor=clip_processor))
        | "CaptionRanking"
        >> RunInference(model_handler=KeyedModelHandler(clip_model_handler))
        | "CLIPPostProcess" >>
beam)ParDo(CLIPPostProcess(processor=clip_processor))

    img_url_captions_ranking | "FormatCaptions" >> beam.ParDo(FormatCaptions(3))
```

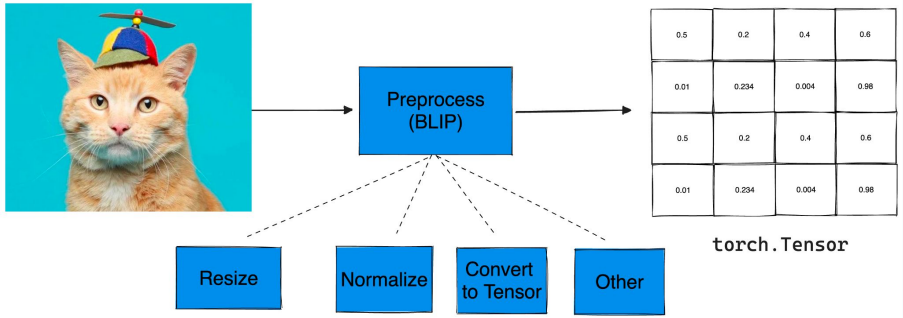
Read Images from URLs




```python
def read_img_from_url(img_url: str) -> Tuple[str,
Image.Image]:
    image = Image.open(requests.get(img_url, stream=True).raw)
    return img_url, image

with beam.Pipeline() as pipeline:
    img_url_pil_img = (
        pipeline
        | "ReadUrl" >> beam.Create(images_url)
        | "ReadImages" >> beam.Map(read_img_from_url)
    )
```

(Img URL, Image)

Preprocess Inputs for BLIP

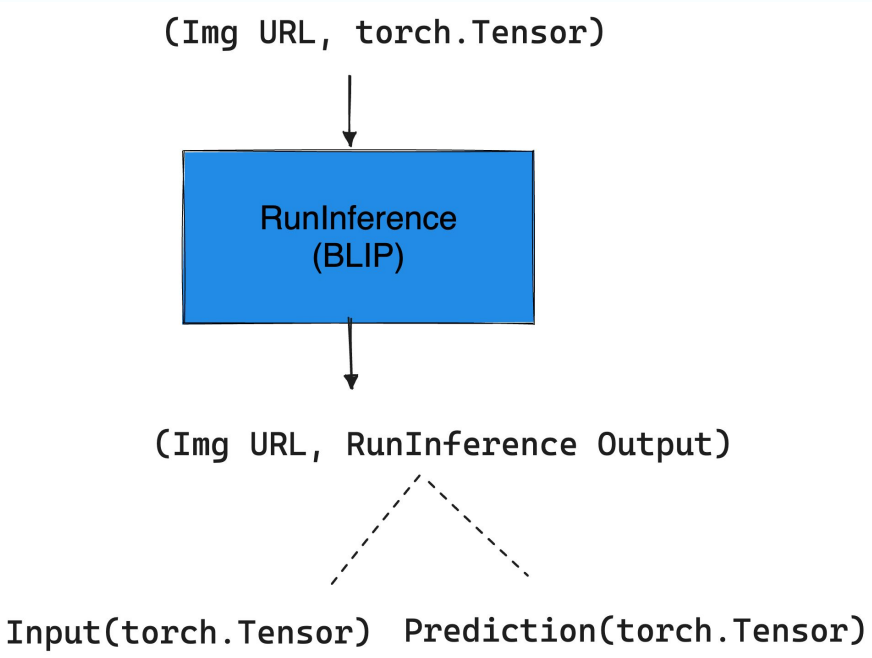



```python
def blip_preprocess(image: Image.Image, processor: BlipProcessor)-> torch.Tensor:
    inputs = processor(images=image, return_tensors="pt")
    return inputs.pixel_values

blip_processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-base")

img_url_captions = (
    img_url_pil_img
    | "BLIPPreprocess"
    >> beam.MapTuple(
        lambda img_url, img: (
            img_url,
            blip_preprocess(img, processor=blip_processor),
        )
    )
```

(Img URL, torch.Tensor)

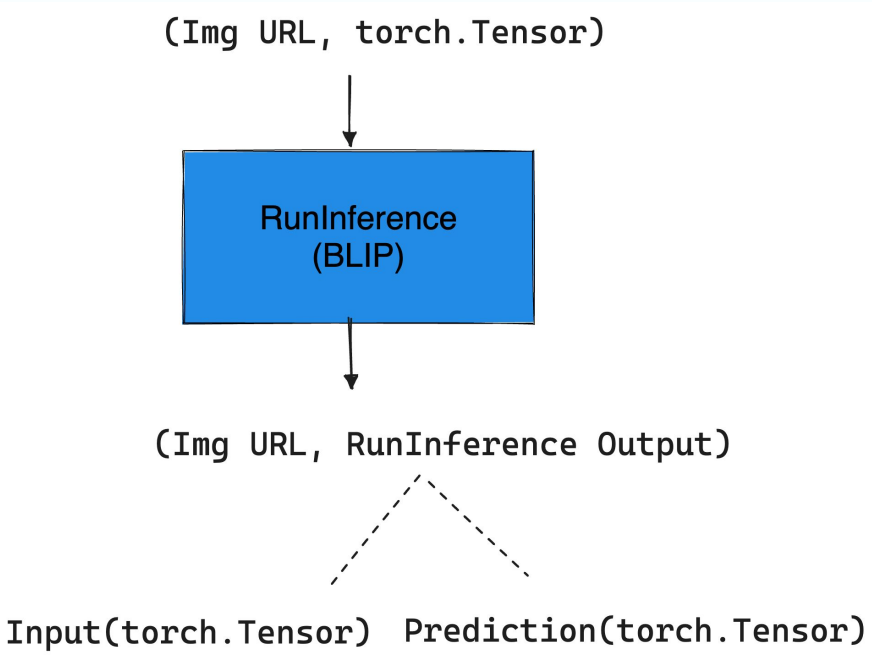
Hugging Face

Inference using BLIP




```
        | "GenerateCaptions"
        >> RunInference(
          model_handler=blip_model_handler,
          inference_args={
                "max_length": 50,
                "min_length": 10,
                "num_return_sequences": 5,
                "do_sample": True,
            },
        )
```

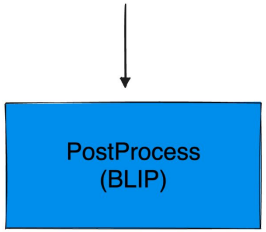
Inference using BLIP




```
gen_fn = mod_make_tensor_model_fn('generate')

blip_model_handler = KeyedModelHandler(
  PytorchModelHandlerTensor(
    state_dict_path="./blip_model.pth",
    model_class=BlipForConditionalGeneration,
    model_params={
        "config": AutoConfig.from_pretrained(model_id)
    },
    max_batch_size=1,
    device = "gpu"
    inference_fn=gen_fn))
```

PostProcess BLIP Output

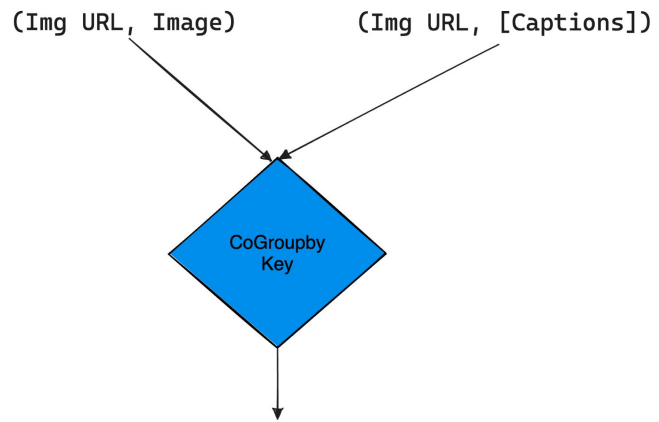
(Img URL, RunInference Output)




```
(Img URL, [ A cat wearing a hat, with blue background,
            A cat in a toy hat that looks like a helicopter,
            A cat wearing a hat with a propeller on top ]
```

```python
class BLIPPostprocess(beam.DoFn):
  def __init__(self, processor: BlipProcessor):
    self._processor = processor

  def process(self, element):
    img_url, output = element
    captions = blip_processor.batch_decode(output.inference,
skip_special_tokens=True)
    yield img_url, captions


| "BLIPPostProcess" >> beam.ParDo(BLIPPostprocess(processor=blip_processor))
```

Grouping Image and BLIP Output




```
(Img URL, {'image': [Image], 'captions': [
              [A cat wearing a hat, with blue background,
               A cat in a toy hat that looks like a helicopter,
               A cat wearing a hat with a propeller on top]
               ]
           }
)
```

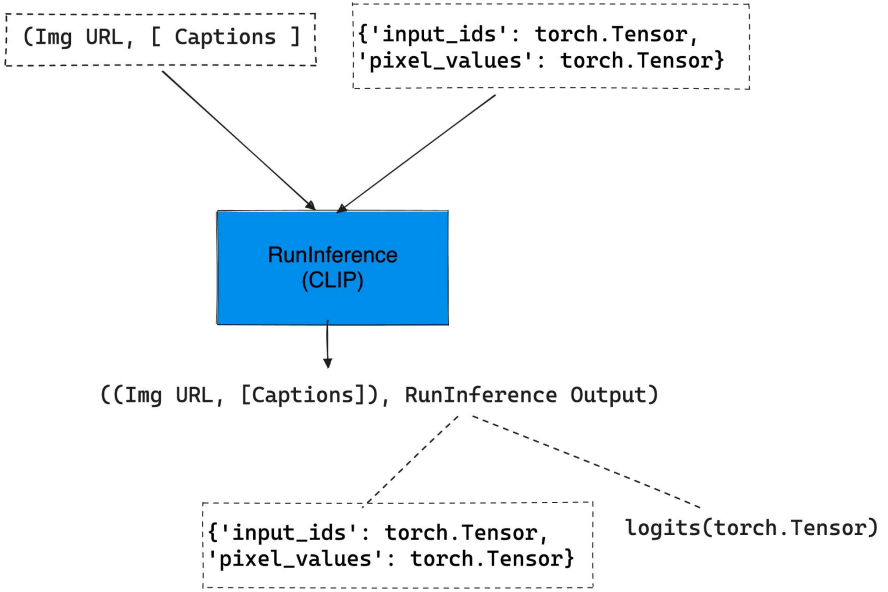
```
img_url_captions_ranking = (
  ({"image": img_url_pil_img, "captions": img_url_captions})
  | "CreateImageCaptionPair" >> beam.CoGroupByKey()
```

Preprocess Inputs for CLIP


```python
class CLIPPreprocess(beam.DoFn):
  def __init__(self, processor: CLIPProcessor):
    self._processor = processor

  def process(self, element):
    img_url, grouped_val = element
    pil_img, captions = grouped_val['image'], grouped_val['captions'][0]
    processed_output = self._processor(text=captions,
                                       images=pil_img,
                                       return_tensors="pt",
                                       padding=True)

    yield (img_url, captions), processed_output

clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```

Hugging Face

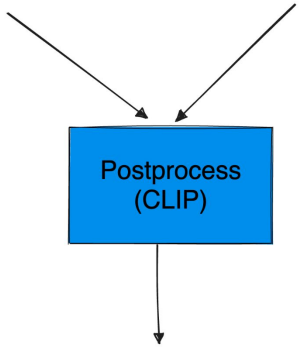
Inference using CLIP




```python
class CLIPWrapper(CLIPModel):

  def forward(self, **kwargs: Dict[str, torch.Tensor]):
    # Squeeze because RunInference adds an extra dimension, which is empty.
    kwargs = {key: tensor.squeeze(0) for key, tensor in kwargs.items()}
    output = super().forward(**kwargs)
    logits = output.logits_per_image
    return logits


clip_model_handler = KeyedModelHandler(PytorchModelHandlerKeyedTensor(
    state_dict_path="./clip_model.pth",
    model_class=CLIPWrapper,
    model_params={
        "config": AutoConfig.from_pretrained("openai/clip-vit-base-patch32")
    },
    max_batch_size=1,))


| "CaptionRanking" >> RunInference(model_handler=clip_model_handler)
```

PostProcess CLIP Output

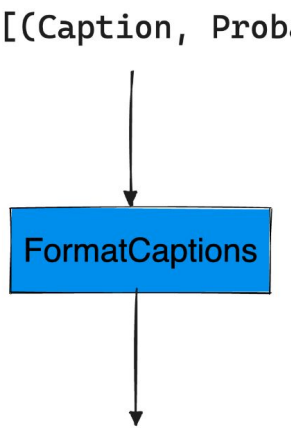
((Img URL, [Captions]), RunInference Output)




```
(
https://image_captioning/cat_with_hat.jpg,
[('A cat wearing a hat with a propeller on top',
0.43382697),
('A cat in a toy hat that looks like a helicopter',
0.32000825),
('A cat wearing a hat, with blue background',
0.16968591)]
)
```

```python
class CLIPPostProcess(beam.DoFn):
  def __init__(self, processor: CLIPProcessor):
    self._processor = processor

  def process(self, element):
    (image_url, captions), prediction = element
    prediction_results = prediction.inference
    prediction_probs = prediction_results.softmax(dim=-1).cpu().detach().numpy()
    ranking = np.argsort(-prediction_probs)
    sorted_caption_prob_pair = [(captions[idx], prediction_probs[idx]) for idx in
ranking]
    return [(image_url, sorted_caption_prob_pair)]


| "CLIPPostProcess" >> beam.ParDo(CLIPPostProcess(processor=clip_processor))
```

Printing the results nicely




```
Image: cat_with_hat
Top 3 captions ranked by CLIP:
1: A cat wearing a hat with a propeller on top
(Caption probability: 0.4338)
2: A cat in a toy hat that looks like a helicopter.
(Caption probability: 0.3200)
3: A cat wearing a hat, with blue background.
(Caption probability: 0.1697)
```

Takeaways

- RunInference transform eliminates the need for extensive boilerplate code in pipelines with machine learning models.
- Multiple RunInference transforms enable complex pipelines with minimal code for multi-ML models.
- Example pipeline can be used for captioning images for finetuning Stable Diffusion.

Code and Tutorial Link

Code: GitHub Link

Tutorial: Apache Beam Documentation Link

Slides: GitHub Link

Shubham Krishna

QUESTIONS?


shubham-krishna-998922108


shub-kris