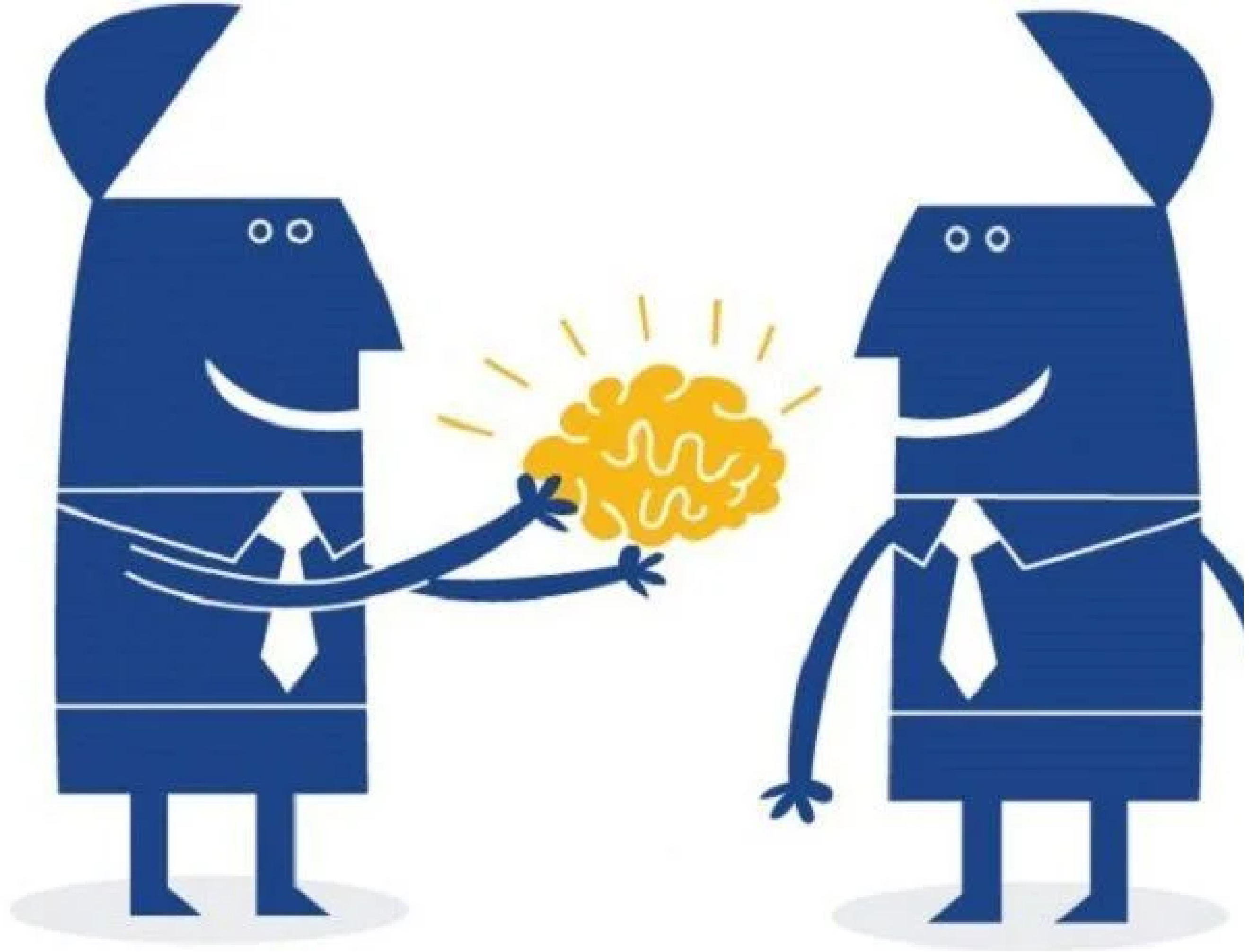
Transfer learning for text classification

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

What is transfer learning?

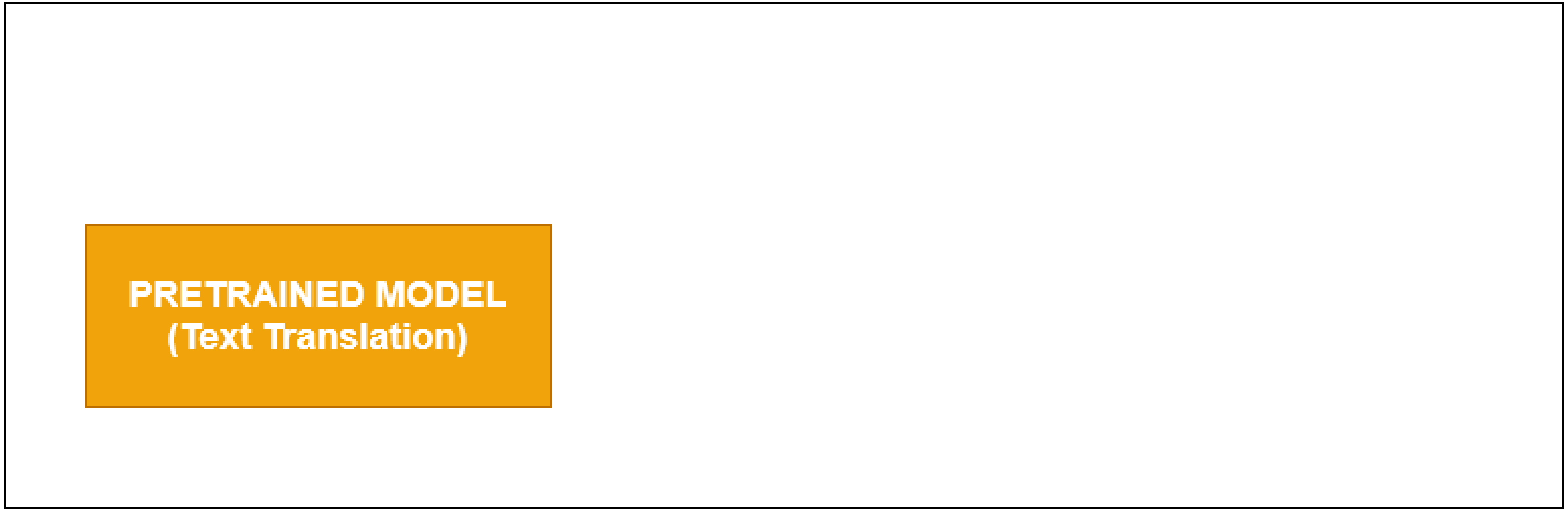


- Use pre-existing knowledge from one task to a related task

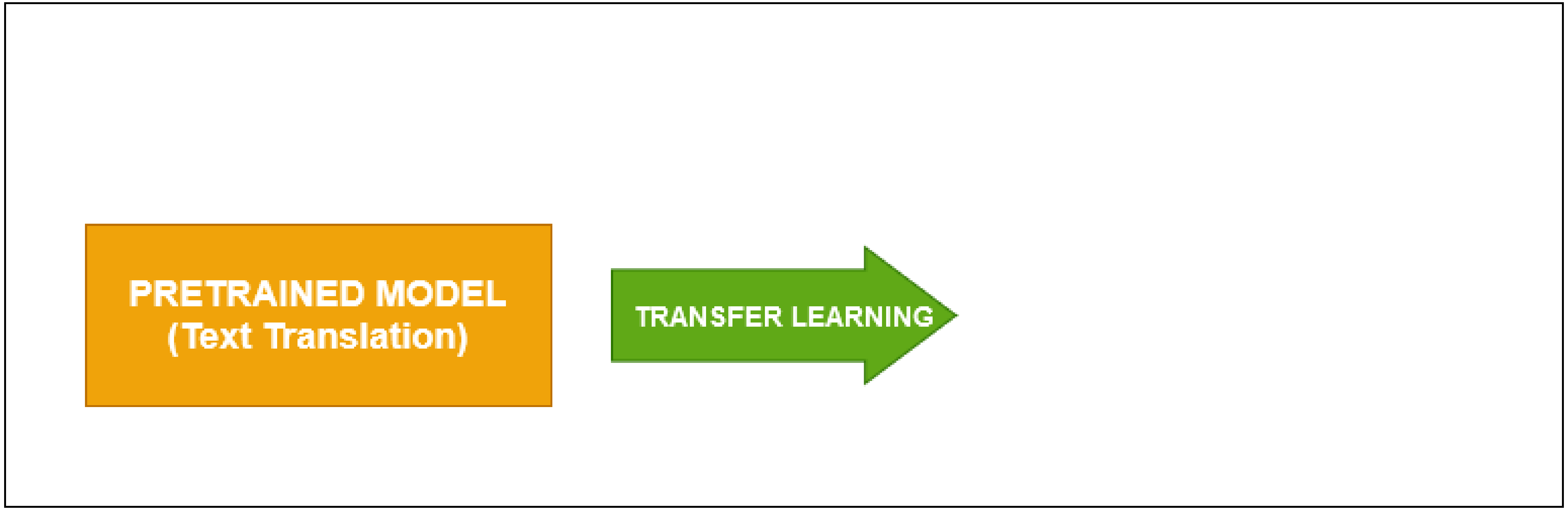
- Saves time
- Share expertise
- Reduces need for large data

- An English teacher starts teaching History

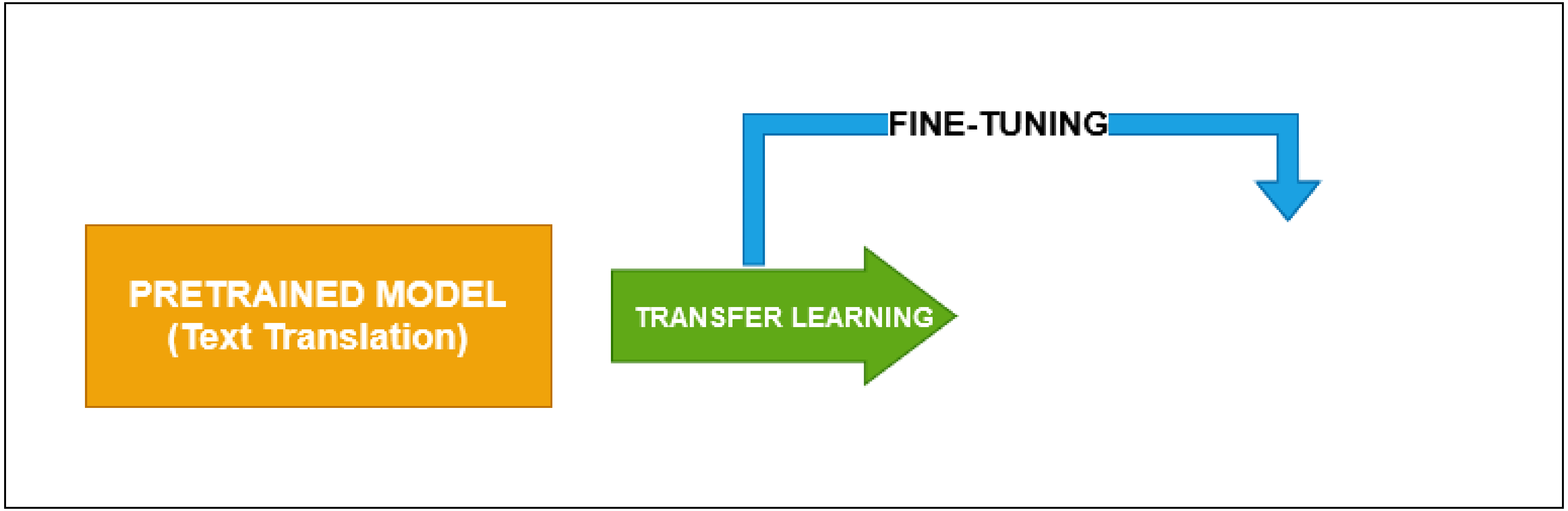
Mechanics of transfer learning



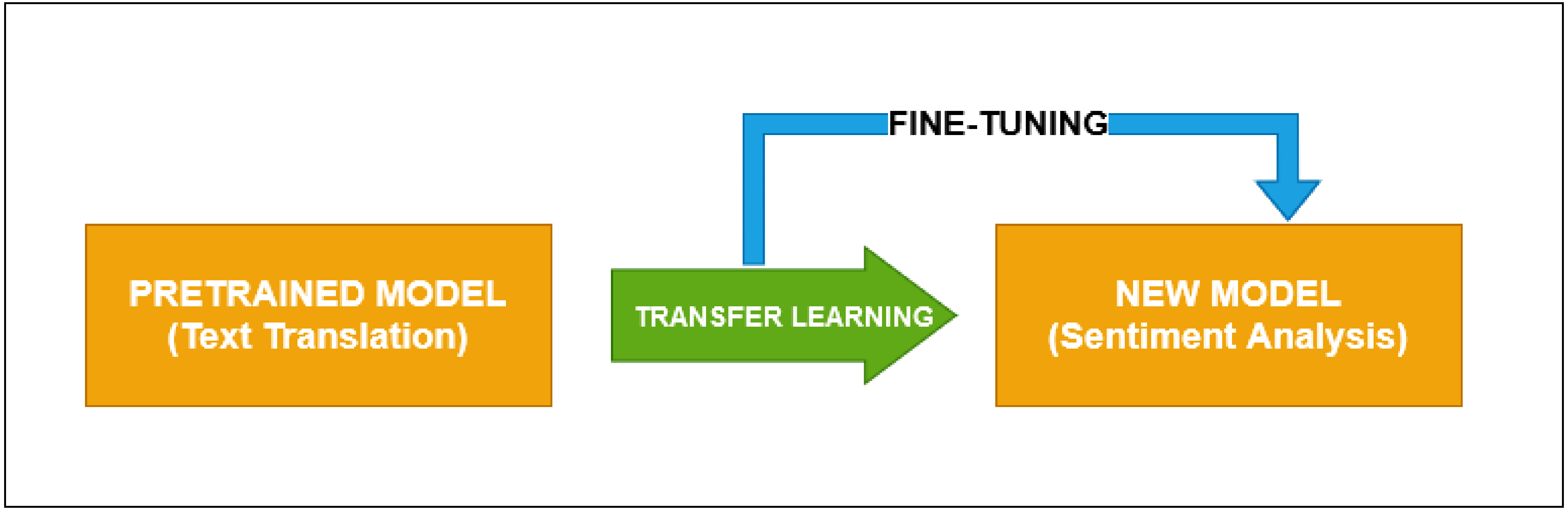
Mechanics of transfer learning



Mechanics of transfer learning

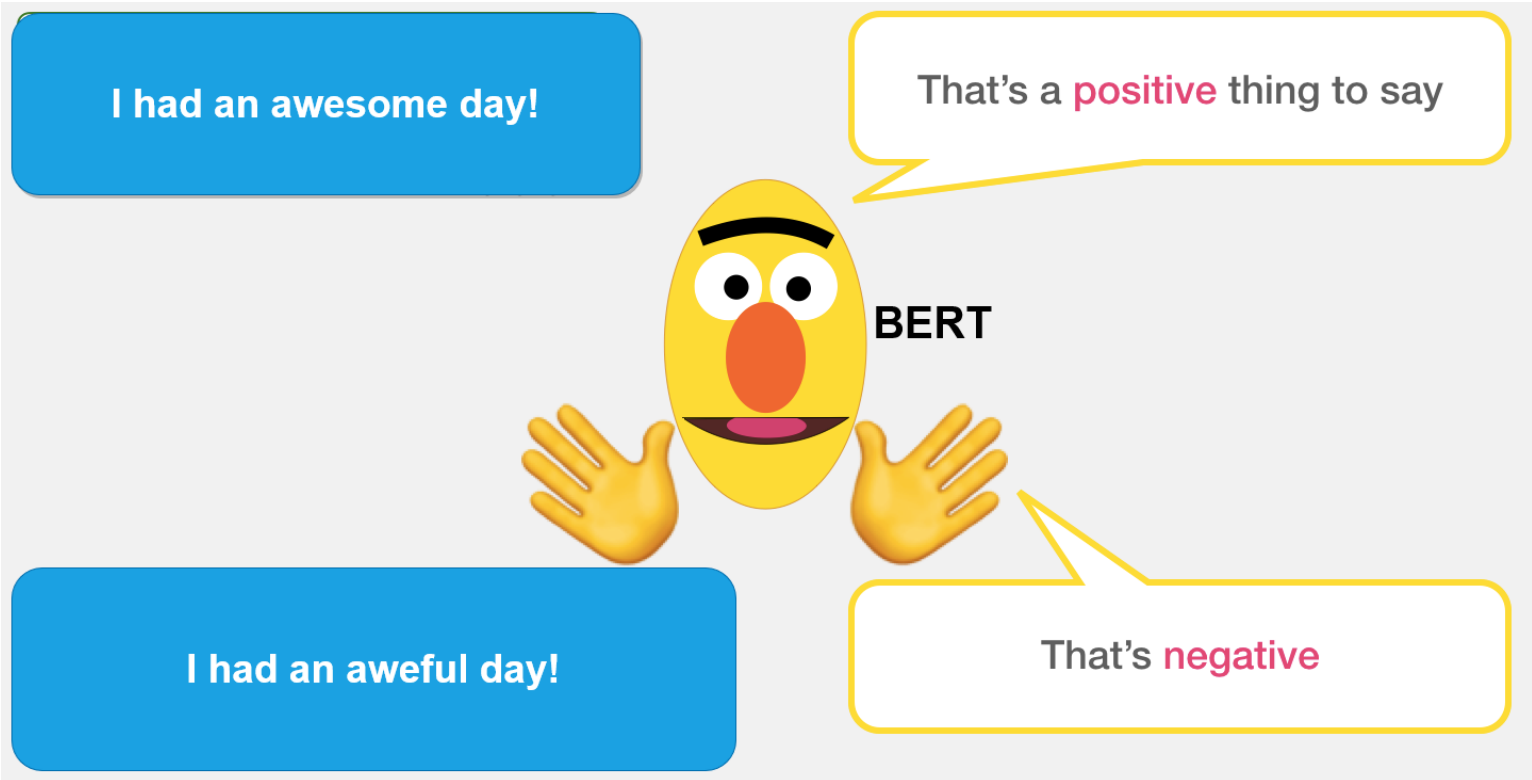


Mechanics of transfer learning



Pre-trained model : BERT

- Bidirectional Encoder Representations from Transformers



- Trained for language modeling
- Multiple layers of transformers
- Pre-trained on large texts

Hands-on: implementing BERT

```
texts = ["I love this!",
         "This is terrible.",
         "Amazing experience!",
         "Not my cup of tea."]
labels = [1, 0, 1, 0]
import torch
from transformers import BertTokenizer, BertForSequenceClassification
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
                                                      num_labels=2)
inputs = tokenizer(texts, padding=True, truncation=True,
                   return_tensors="pt", max_length=32)
inputs["labels"] = torch.tensor(labels)
```

Fine-tuning BERT

```python
optimizer = torch.optim.AdamW(model.parameters(), lr=0.00001)
model.train()
for epoch in range(1):
    outputs = model(**inputs)
    loss = outputs.loss
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

```
Epoch: 1, Loss: 0.7061821222305298
```

Evaluating on new text

```
text = "I had an awesome day!"
input_eval = tokenizer(text, return_tensors="pt", truncation=True,
                       padding=True, max_length=128)
outputs_eval = model(**input_eval)
predictions = torch.nn.functional.softmax(outputs_eval.logits, dim=-1)
predicted_label = 'positive' if torch.argmax(predictions) > 0 else 'negative'
print(f"Text: {text}\nSentiment: {predicted_label}")
```

```
Text: I had an awesome day!
Sentiment: positive
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Transformers for text processing

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

Why use transformers for text processing?

Transformers

- Speed
- Understand the relationship between words, regardless of distances
- Human-like response

Components of a transformer

- **Encoder:** Processes input data
- **Decoder:** Reconstructs the output
- **Feed-forward Neural Networks:** Refine understanding
- **Positional Encoding:** Ensure order matters
- **Multi-Head Attention:** Captures multiple inputs or sentiments

Preparing our data: train-test split

```
sentences = ["I love this product", "This is terrible",
             "Could be better", "This is the best"]
labels = [1, 0, 0, 1]
train_sentences = sentences[:3]
train_labels = labels[:3]
test_sentences = sentences[3:]
test_labels = labels[3:]
```

Building the transformer model

```
class TransformerEncoder(nn.Module):
    def __init__(self, embed_size, heads, num_layers, dropout):
        super(TransformerEncoder, self).__init__()
        self.encoder = nn.TransformerEncoder(
         nn.TransformerEncoderLayer(d_model=embed_size, nhead=heads),
         num_layers=num_layers)
        self.fc = nn.Linear(embed_size, 2)
    def forward(self, x):
        x = self.encoder(x)
        x = x.mean(dim=1)
        return self.fc(x)
model = TransformerEncoder(embed_size=512, heads=8, num_layers=3, dropout=0.5)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

Training the transformers

```
for epoch in range(5):
 for sentence, label in zip(train_sentences, train_labels):
        tokens = sentence.split()
        data = torch.stack([token_embeddings[token] for token in tokens], dim=1)
        output = model(data)
        loss = criterion(output, torch.tensor([label]))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```

```
Epoch 0, Loss: 13.788233757019043
Epoch 1, Loss: 3.9480819702148438
Epoch 2, Loss: 2.4790847301483154
Epoch 3, Loss: 1.3020926713943481
Epoch 4, Loss: 0.4660853147506714
```

Predicting the transformers

```
def predict(sentence):
    model.eval()
    with torch.no_grad():
        tokens = sentence.split()
        data = torch.stack([token_embeddings.get(token, torch.rand((1, 512)))
                            for token in tokens], dim=1)
        output = model(data)
        predicted = torch.argmax(output, dim=1)
        return "Positive" if predicted.item() == 1 else "Negative"
```

Predicting on new text

```
sample_sentence = "This product can be better"
print(f"'{sample_sentence}' is {predict(sample_sentence)}")
```

```
'This product can be better' is Negative
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Attention mechanisms for text generation

DEEP LEARNING FOR TEXT WITH PYTORCH

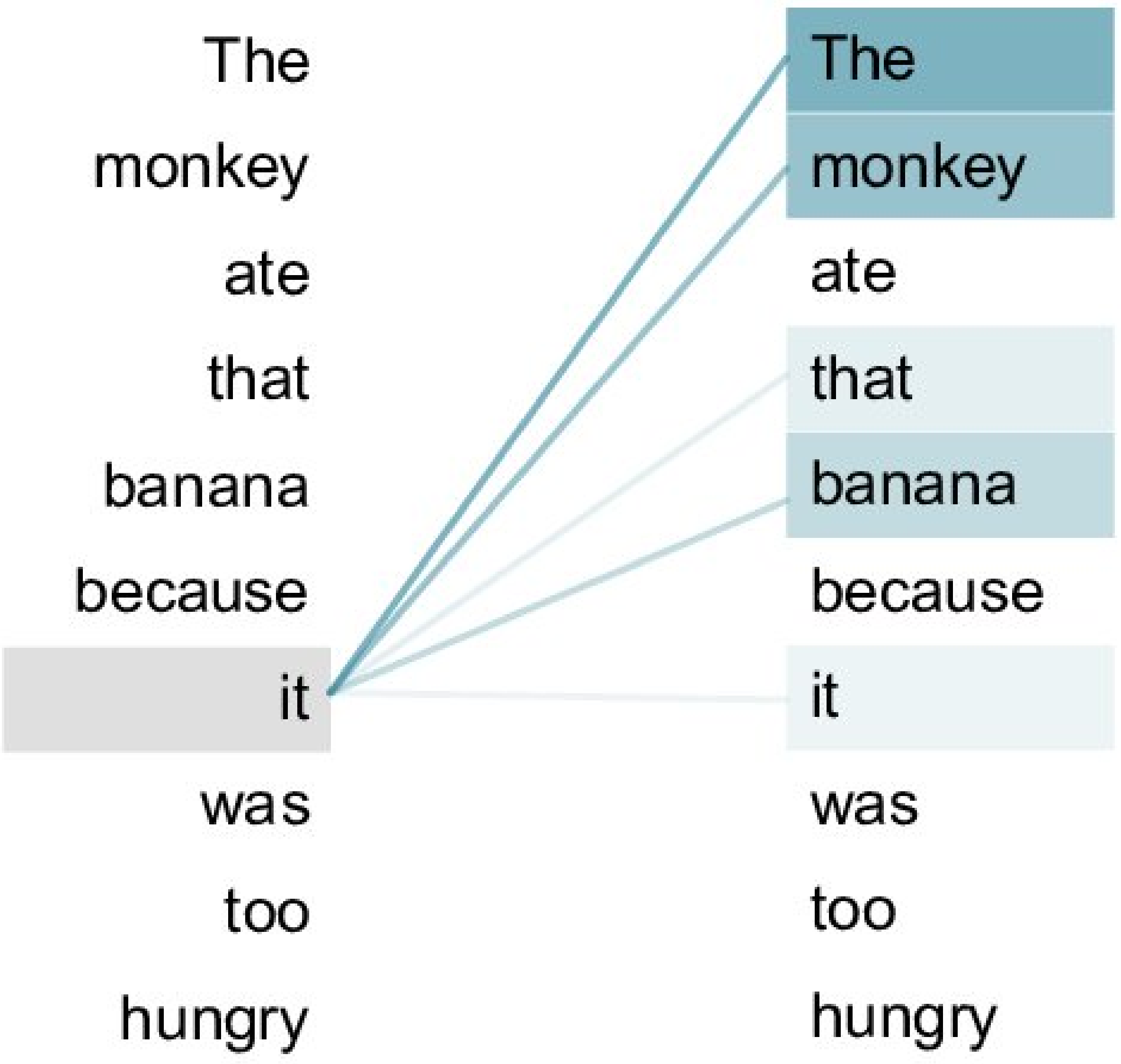
Shubham Jain
Instructor

The ambiguity in text processing

- "The monkey ate that banana because it was too hungry"
- What does the word "it" refer to?

Attention mechanisms

- Assigns importance to words
- Ensures that machine's interpretation aligns with human understanding



[1] Xie, Huiqiang & Qin, Zhijin & Li, Geoffrey & Juang, Biing-Hwang. (2020). Deep Learning Enabled Semantic Communication Systems

Self and multi-head attention

- Self-Attention: assigns significance to words within a sentence
  - The cat, which was on the roof, was scared"
  - Linking "was scared" to "The cat"
- Multi-Head Attention: like having multiple spotlights, capturing different facets
  - Understanding "was scared" can relate to
  - "The cat", "the roof", or "was on"

Attention mechanism - setting vocabulary and data

```
data = ["the cat sat on the mat", ...]
vocab = set(' '.join(data).split())
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for word, i in word_to_ix.items()}
pairs = [sentence.split() for sentence in data]
input_data = [[word_to_ix[word] for word in sentence[:-1]] for sentence in pairs]
target_data = [word_to_ix[sentence[-1]] for sentence in pairs]
inputs = [torch.tensor(seq, dtype=torch.long) for seq in input_data]
targets = torch.tensor(target_data, dtype=torch.long)
```

Model definition

```
embedding_dim = 10
hidden_dim = 16

class RNNWithAttentionModel(nn.Module):
    def __init__(self):
        super(RNNWithAttentionModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim, batch_first=True)
        self.attention = nn.Linear(hidden_dim, 1)
        self.fc = nn.Linear(hidden_dim, vocab_size)
```

Forward propagation with attention

```
def forward(self, x):
    x = self.embeddings(x)
    out, _ = self.rnn(x)
    attn_weights = torch.nn.functional.softmax(self.attention(out).squeeze(2),
                                                dim=1)
    context = torch.sum(attn_weights.unsqueeze(2) * out, dim=1)
    out = self.fc(context)
    return out
def pad_sequences(batch):
    max_len = max([len(seq) for seq in batch])
    return torch.stack([torch.cat([seq, torch.zeros(max_len-len(seq)).long()])
                        for seq in batch])
```

Training preparation

```
criterion = nn.CrossEntropyLoss()
attention_model = RNNWithAttentionModel()
optimizer = torch.optim.Adam(attention_model.parameters(), lr=0.01)
for epoch in range(300):
    attention_model.train()
    optimizer.zero_grad()
    padded_inputs = pad_sequences(inputs)
    outputs = attention_model(padded_inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

Model evaluation

```
for input_seq, target in zip(input_data, target_data):
    input_test = torch.tensor(input_seq, dtype=torch.long).unsqueeze(0)
    attention_model.eval()
    attention_output = attention_model(input_test)
    attention_prediction = ix_to_word[torch.argmax(attention_output).item()]
    print(f"\nInput: {' '.join([ix_to_word[ix] for ix in input_seq])}")
    print(f"Target: {ix_to_word[target]}")
    print(f"RNN with Attention prediction: {attention_prediction}")
```

```
Input: the cat sat on the
Target: mat
RNN with Attention prediction: mat
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Adversarial attacks on text classification models

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

What are adversarial attacks?

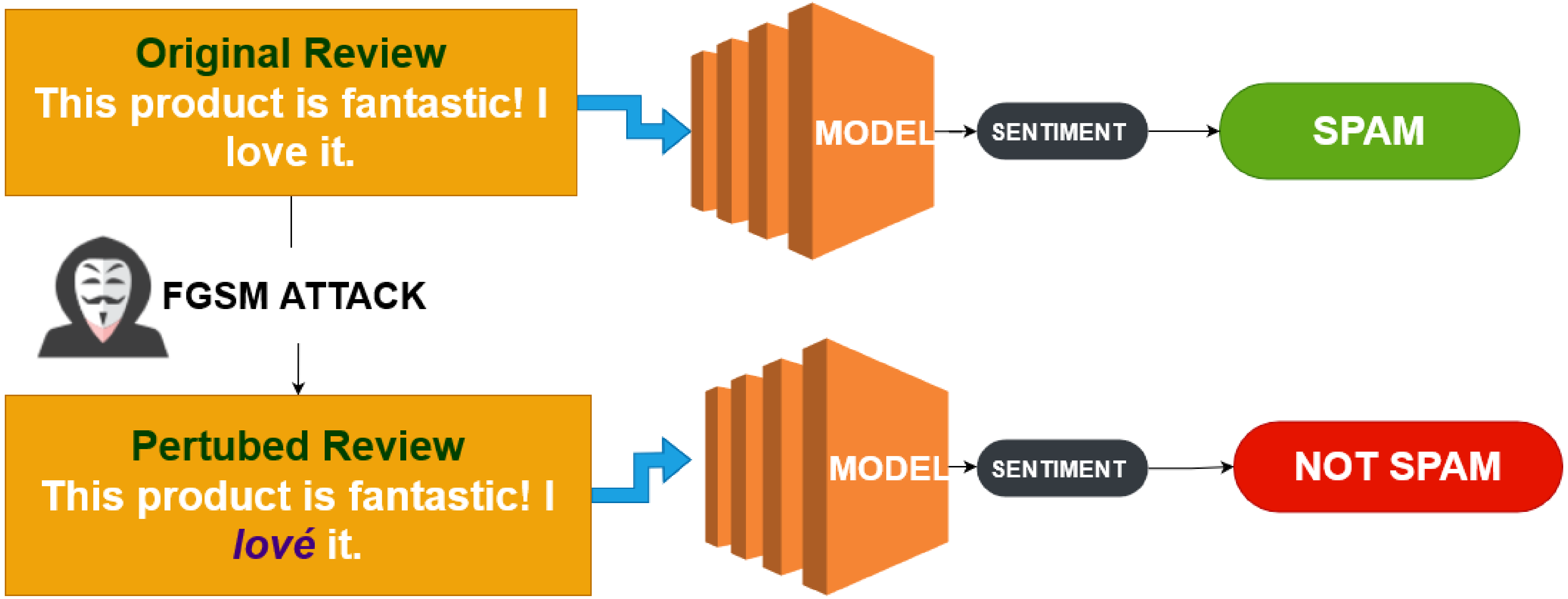
- Tweaks to input data
- Not random but calculated malicious changes
- Can drastically affect AI's decision-making

Importance of robustness

- AI systems deciding if user comments are toxic or benign
- AI unintentionally amplifying negative stereotypes from biased data
- AI giving misleading information

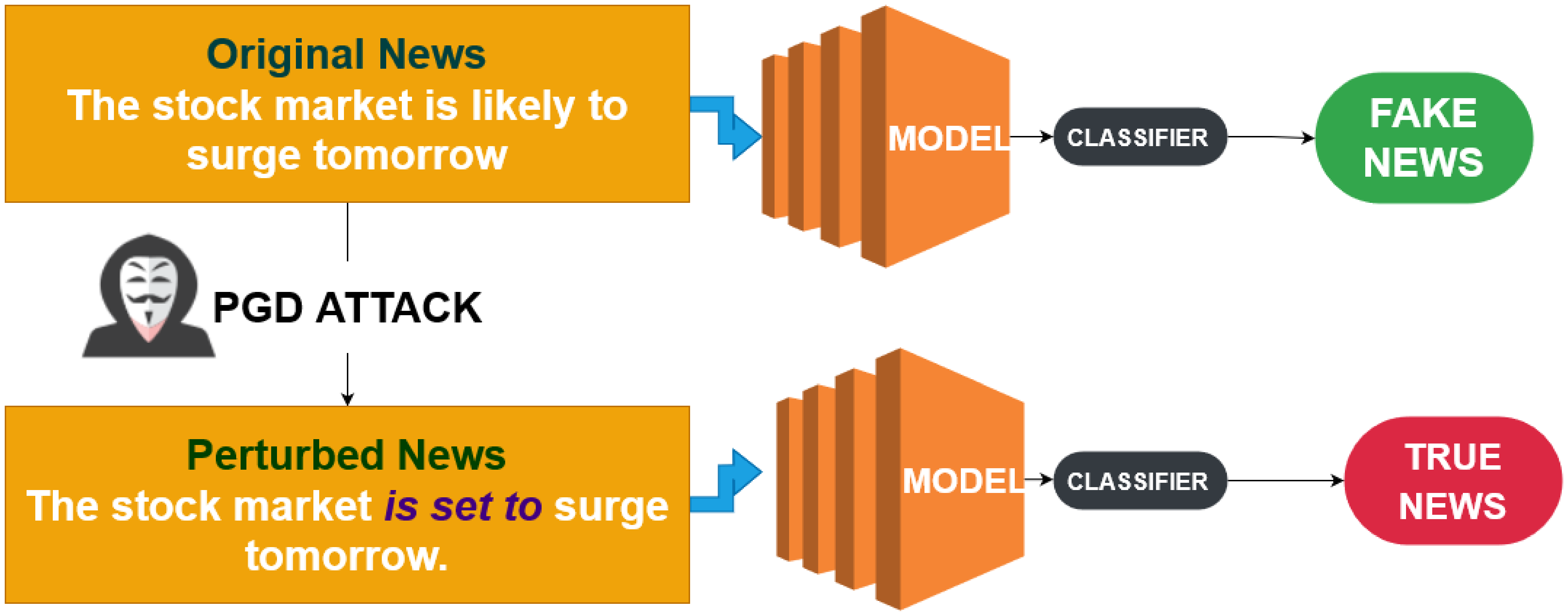
Fast Gradient Sign Method (FGSM)

- Exploits the model's learning information
- Makes the tiniest possible change to deceive the model



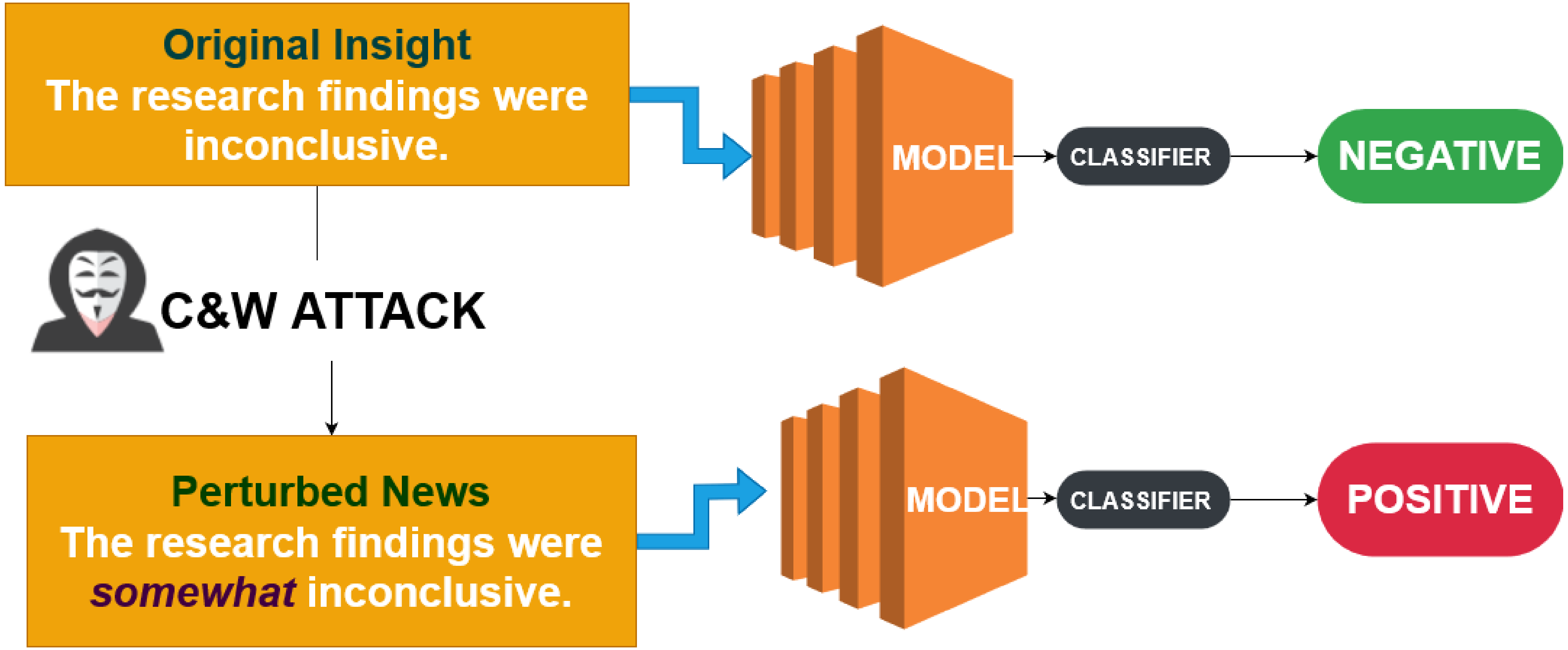
Projected Gradient Descent (PGD)

- More advanced than FGSM: it's iterative
- Tries to find the most effective disturbance



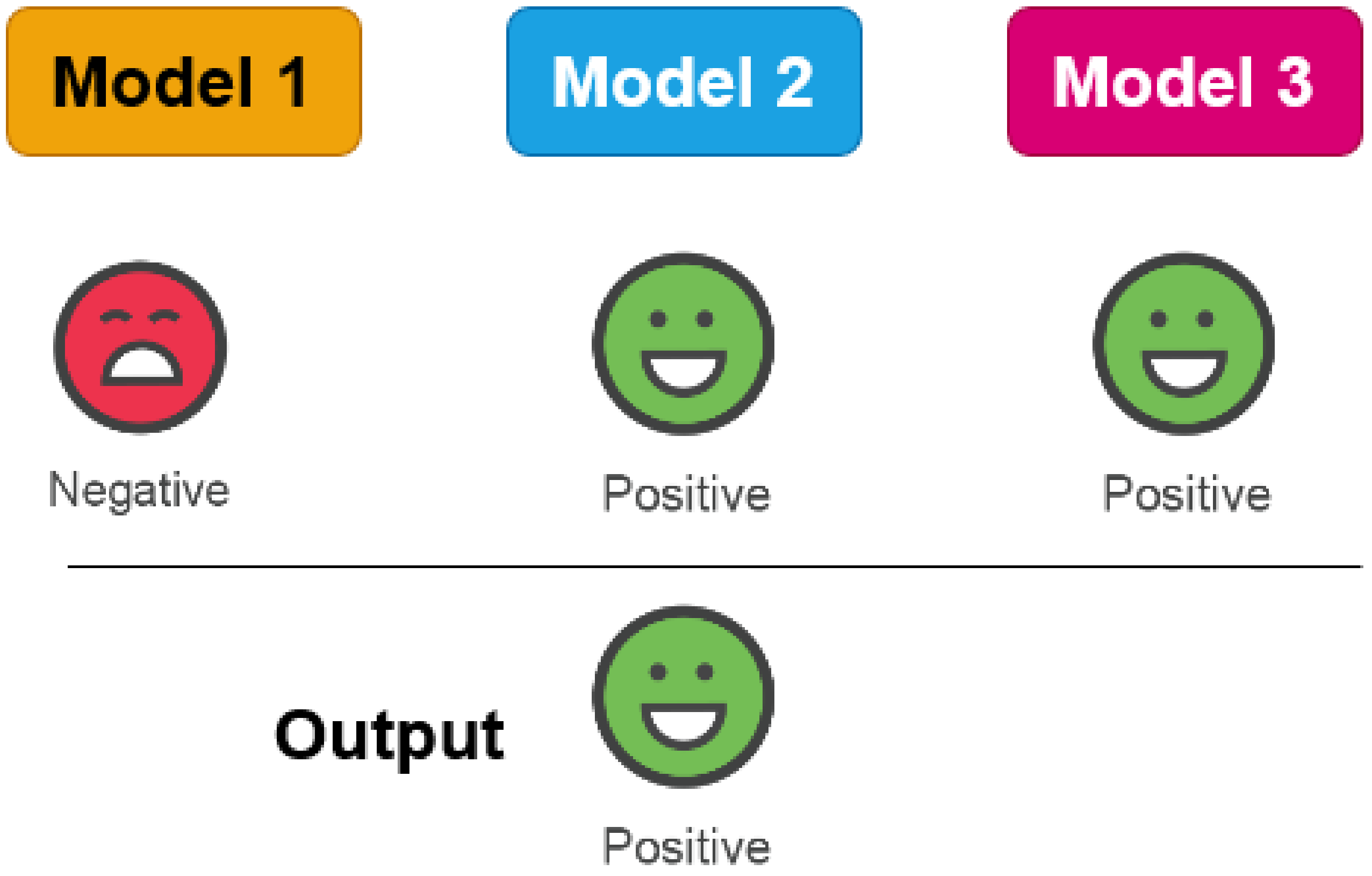
The Carlini & Wagner (C&W) attack

- Focuses on optimizing the loss function
- Not just about deceiving but about being undetectable



Building defenses: strategies

- **Model Ensembling:**
  - Use multiple models
- **Robust Data Augmentation:**
  - Data augmentation
- **Adversarial Training:**
  - Anticipate deception

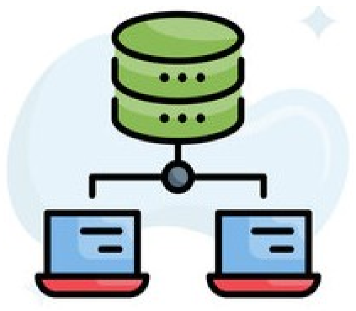


Building defenses: tools & techniques

- **PyTorch's Robustness Toolbox:**
  - Strengthen text models

Adversarial Robustness Toolbox

- **Gradient Masking:**
  - Add variety to training data to hide exploitable patterns



Dataset

- **Regularization Techniques:**
  - Ensure model balance

[1] https://adversarial-robustness-toolbox.readthedocs.io/en/latest/, https://stock.adobe.com/ie/contributor/209161356/designer-s-circle

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Wrap-up

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

What you learned

- Chapter 1: Foundations of Text Processing
- Chapter 2: Text Classification Techniques
- Chapter 3: Text Generation Methods and Pre-trained Models
- Chapter 4: Advanced Deep Learning Topics

Key takeaways

- Encoding Text: one-hot, BoW, TF-IDF
- Deep Learning Models: CNN, RNN, GAN
- Advanced Techniques: Transformers & Attention
- Adversarial Attacks on Text Classification

Applied learning

- Implemented text classification models
- Built text generation models
- Used pre-trained models for text tasks
- Applied transfer learning

What's next?

- On DataCamp:
  - **How to Train a LLM with PyTorch**
  - **Building a Transformer with PyTorch**
- Projects: text completion, chatbot text generation and sentiment analysis

Congratulations and Thank You!

DEEP LEARNING FOR TEXT WITH PYTORCH