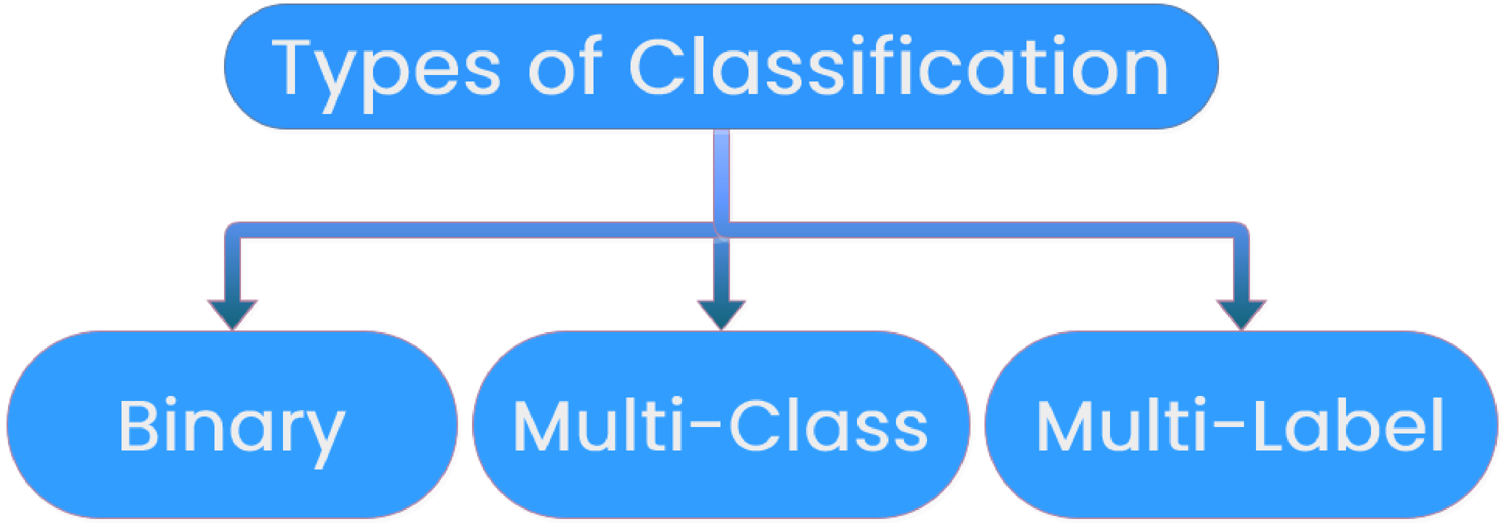
Overview of Text Classification

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

Text classification defined

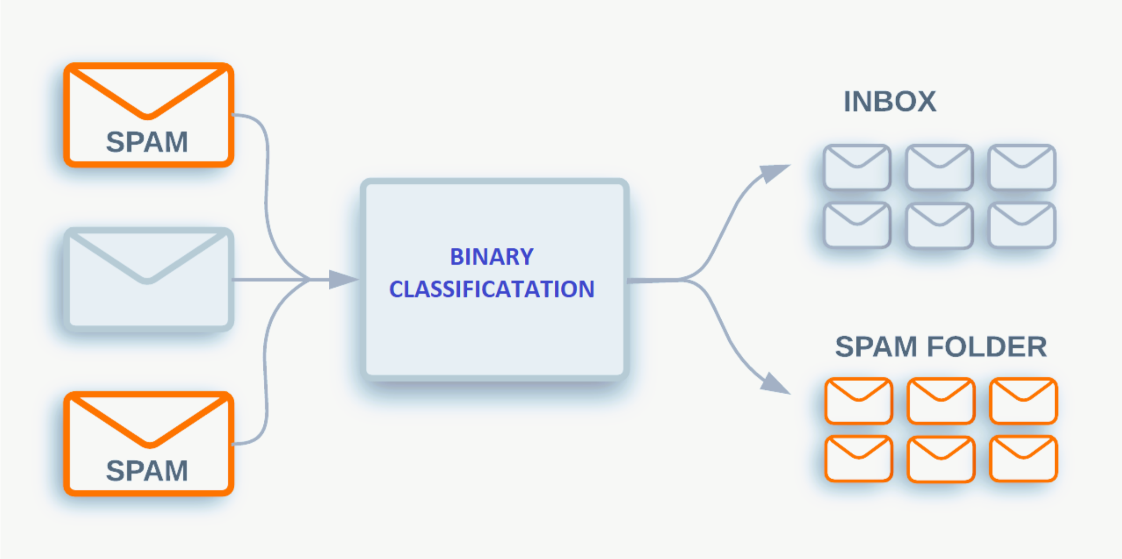
- Assigning labels to text
- Giving meaning to words and sentences



- Organizes and gives structure to unstructured data
- Applications:
  - Analyzing customer sentiment in reviews
  - Detecting spam in emails
  - Tagging news articles with relevant topics
- Types: binary, multi-class, multi-label

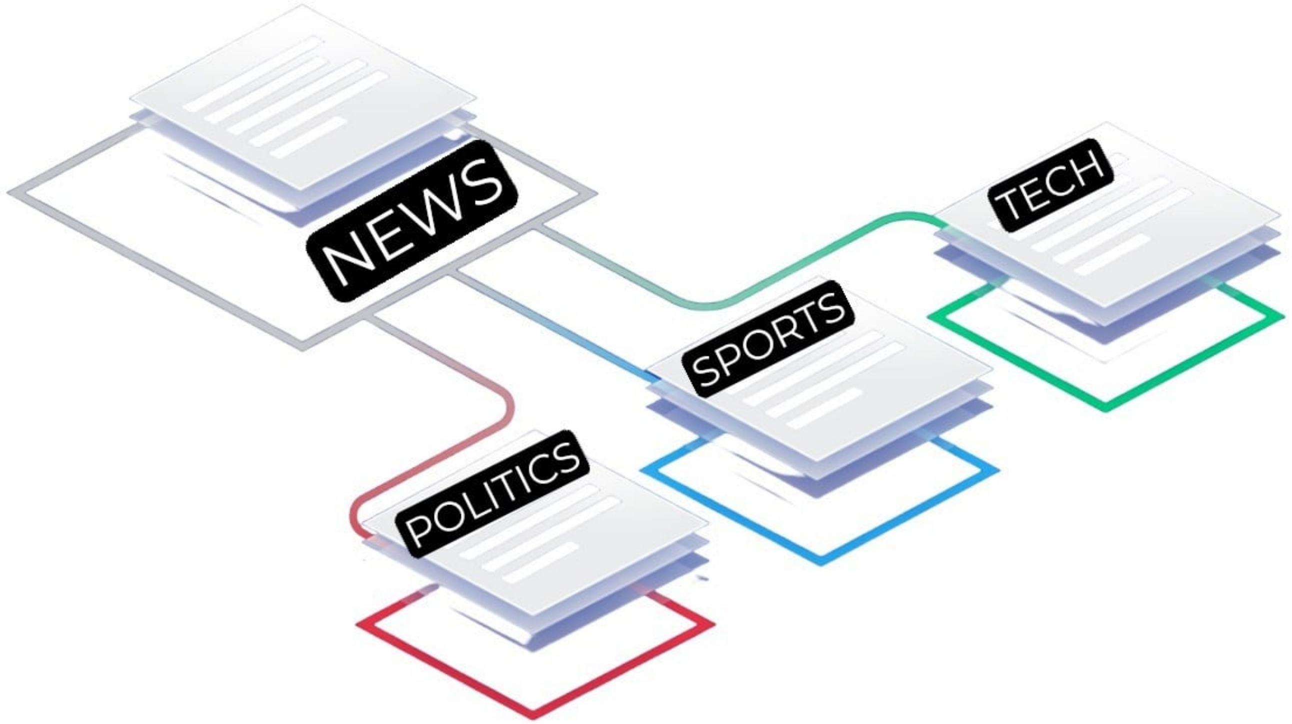
Binary classification

- Sorting into two categories
- Example: email spam detection
- Emails can be classified as '**spam**' or '**not spam**'



[1] https://storage.googleapis.com/gweb-cloudblog-publish/images/image4_v2LFcq0.max-1200x1200.png

Multi-class classification

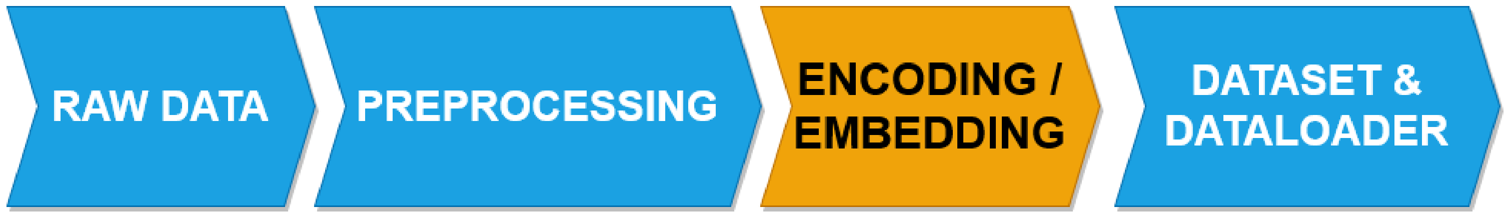


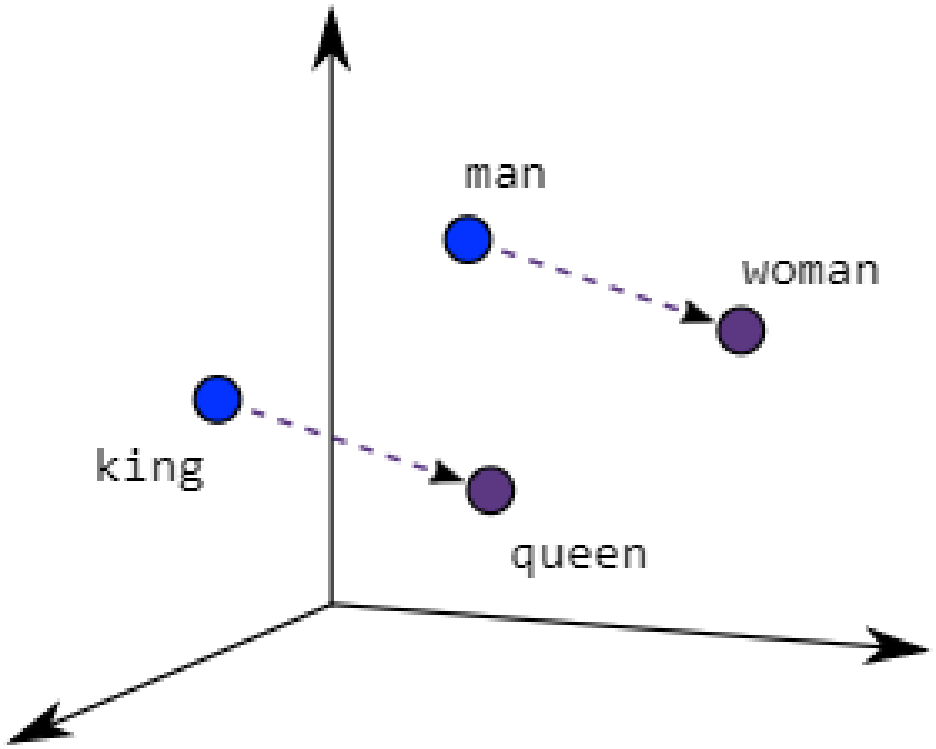
- Sorting into multiple categories
- Example: **News articles** can be sorted into various categories like
  1. **Politics**
  2. **Sports**
  3. **Technology**

Multi-label classification

- Each text can be assigned multiple labels
- Example: **Books** can be multiple genres
  - Action
  - Adventure
  - Fantasy

What are word embeddings





- Previous encoding techniques are a good first step
  - Often create too many features and can't identify similar words
- Word embeddings map words to numerical vectors
- Example of semantic relationship:
  - King and queen
  - Man and woman

Word to index mapping

- Example:
  - "King" -> 1
  - "Queen" -> 2
- Compact and computationally efficient
- Follows tokenization in the pipeline

Word embeddings in PyTorch

- `torch.nn.Embedding` :
  - Creates word vectors from indexes

- Input: Indexes for ['The', 'cat', 'sat', 'on', 'the', 'mat']

```
Embedding for 'the': tensor([-0.4689,  0.3164, -0.2971, -0.1291,  0.4064])
Embedding for 'cat': tensor([-0.0978, -0.4764,  0.0476,  0.1044, -0.3976])
Embedding for 'sat': tensor([ 0.2731,  0.4431,  0.1275,  0.1434, -0.4721])
```

Using torch.nn.Embedding

```
import torch
from torch import nn
words = ["The", "cat", "sat", "on", "the", "mat"]
word_to_idx = {word: i for i, word in enumerate(words)}
inputs = torch.LongTensor([word_to_idx[w] for w in words])
embedding = nn.Embedding(num_embeddings=len(words), embedding_dim=10)
output = embedding(inputs)
print(output)
```

```
tensor([[ 1.0624,  0.6792,  0.0459,  ... -1.0828, -0.4475,  0.4868],
         ...
         [1.5766,  0.0106,  0.1161,  ...,,  -0.0859, 1.3160,  1.3621])
```

Using embeddings in the pipeline

```
def preprocess_sentences(text):
  # Tokenization
  # Stemming
  ...
  # Word to index mapping
class TextDataset(Dataset):
    def __init__(self, encoded_sentences):
        self.data = encoded_sentences

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return self.data[index]
```

```
def text_processing_pipeline(text):
    tokens = preprocess_sentences(text)
    dataset = TextDataset(tokens)
    dataloader = DataLoader(dataset, batch_size=2,
                            shuffle=True)
    return dataloader, vectorizer

text = "Your sample text here."
dataloader, vectorizer = text_processing_pipeline(text)
embedding = nn.Embedding(num_embeddings=10,
                         embedding_dim=50)

for batch in dataloader:
    output = embedding(batch)
    print(output)
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Convolutional neural networks for text classification

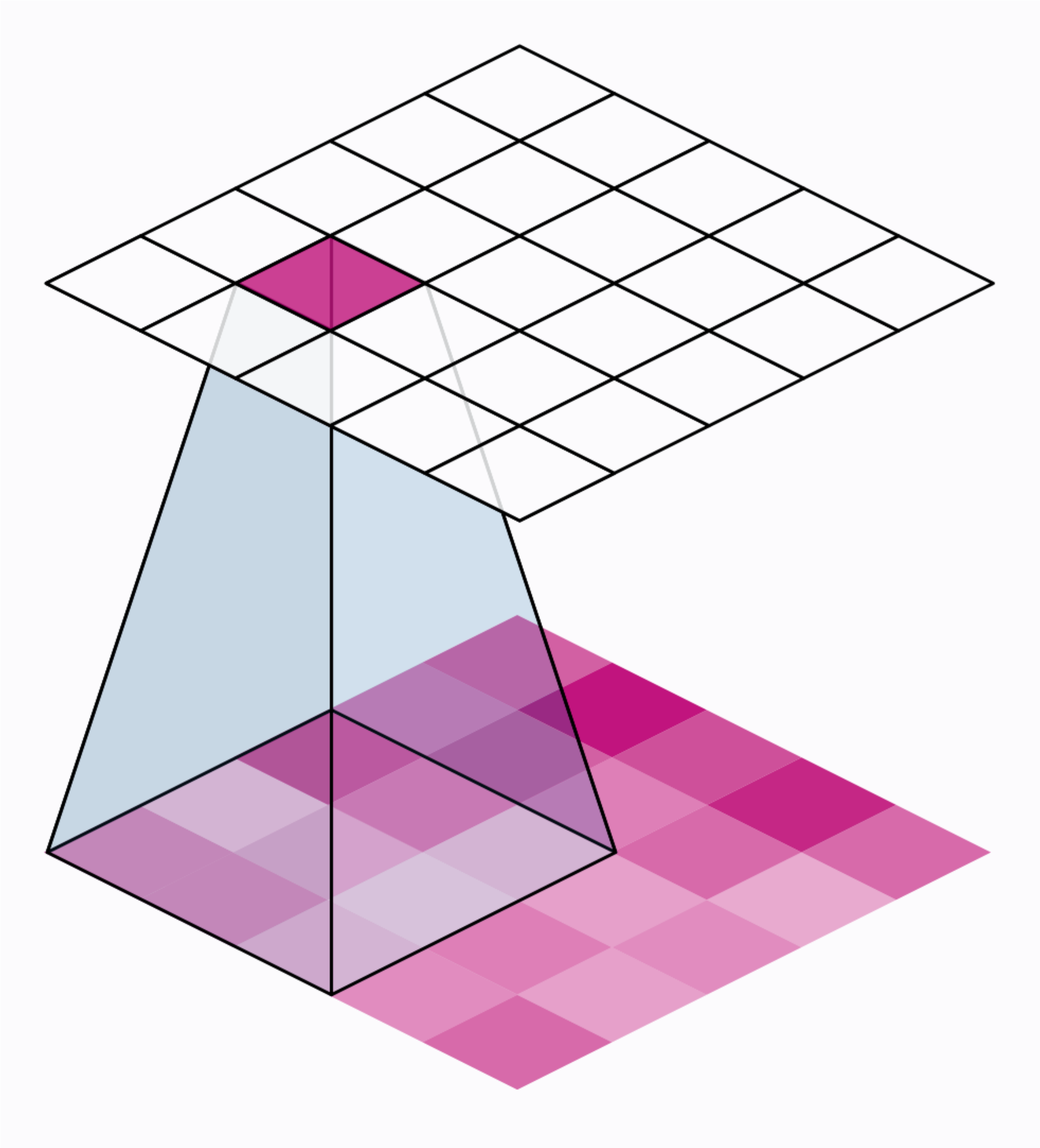
DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

CNNs for text classification

- Classifying tweets as
  - Positive
  - Negative
  - Neutral

The convolution operation

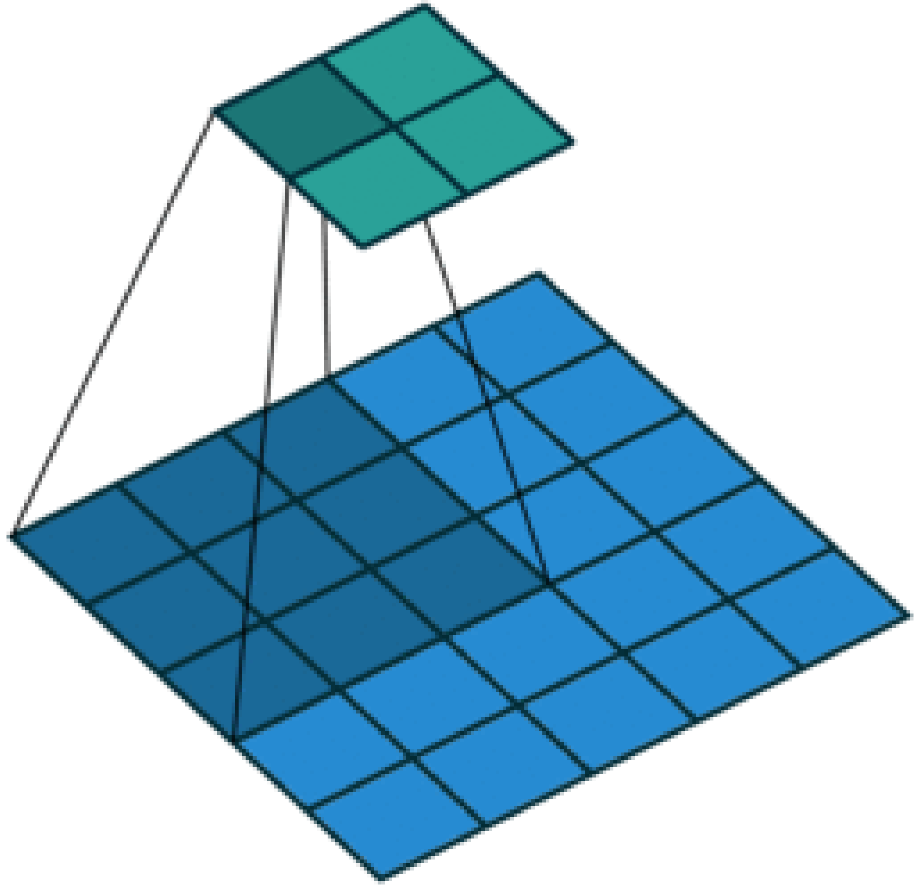


- Convolution operation
  - Sliding a filter (kernel) over the input data
  - For each position of the filter, perform element-wise calculations
- For text: learns structure and meaning of words

[1] Animation from Vincent Dumoulin, Francesco Visin

Filter and stride in CNNs

- Filter:
  - Small matrix that we slide over the input
- Stride:
  - Number of positions the filter moves



[1] Animation from Vincent Dumoulin, Francesco Visin

CNN architecture for text

- Convolutional layer: applies filters to input data
- Pooling layer: reduces data size while preserving important information
- Fully connected layer: makes final predictions based on previous layer output

Implementing a text classification model using CNN

```
class SentimentAnalysisCNN(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size,
                                      embed_dim)
        self.conv = nn.Conv1d(embed_dim, embed_dim,
                              kernel_size=3, stride=1,
                              padding=1)
        self.fc =  nn.Linear(embed_dim, 2)
    ...
```

- `__init__` method configures the architecture
- `super()` initializes the base class `nn.Module`
- `nn.Embedding` creates dense word vectors
- `nn.Conv1d` for one dimensional data

Implementing a text classification model using CNN

```
    ...
    def forward(self, text):
        embedded = self.embedding(text).permute(0, 2, 1)
        conved = F.relu(self.conv(embedded))
        conved = conved.mean(dim=2)
        return self.fc(conved)
```

- Embedding layer converts text to embedding
- Match tensors to convolution layer's expected input
- Extract important features with ReLU
- Eliminate extra layers and dimensions

Preparing data for the sentiment analysis model

```
vocab = ["i", "love", "this", "book", "do", "not", "like"]
word_to_idx = {word: i for i, word in enumerate(vocab)}
vocab_size = len(word_to_ix)
embed_dim = 10
book_samples = [
    ("The story was captivating and kept me hooked until the end.".split(),1),
    ("I found the characters shallow and the plot predictable.".split(),0)
]
model = SentimentAnalysisCNN(vocab_size, embed_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Training the model

```
for epoch in range(10):
    for sentence, label in data:
        model.zero_grad()
        sentence = torch.LongTensor([word_to_idx.get(w, 0) for w in sentence]).unsqueeze(0)
        outputs = model(sentence)
        label = torch.LongTensor([int(label)])
        loss = criterion(outputs, label)
        loss.backward()
        optimizer.step()
```

Running the Sentiment Analysis Model

```
for sample in book_samples:
    input_tensor = torch.tensor([word_to_idx[w] for w in sample], dtype=torch.long).unsqueeze(0)
    outputs = model(input_tensor)
    _, predicted_label = torch.max(outputs.data, 1)
    sentiment = "Positive" if predicted_label.item() == 1 else "Negative"
    print(f"Book Review: {' '.join(sample)}")
    print(f"Sentiment: {sentiment}\n")
```

```
Book Review: The story was captivating and kept me hooked until the end
Sentiment: Positive
Book Review: I found the characters shallow and the plot predictable
Sentiment: Negative
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Recurrent neural networks for text classification

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Data Scientist

RNNs for text

- Handle sequences of varying lengths
- Maintain an internal short-term memory
- CNNs spot patterns in chunks
- RNNs remember past words for greater meaning

RNNs for text classification

Why?

- RNNs can read sentences like humans, one word at a time
- Understand context and order

Example: Detecting sarcasm in a tweet

"I just love getting stuck in traffic."

- Sarcastic

Recap: Implementing Dataset and DataLoader

```python
# Import libraries
from torch.utils.data import Dataset, DataLoader
# Create a class
class TextDataset(Dataset):
    def __init__(self, text):
        self.text = text
    def __len__(self):
        return len(self.text)
    def __getitem__(self, idx):
        return self.text[idx]
```

RNN implementation

```
sample_tweet = "This movie had a great plot and amazing acting."
# Preprocess the review and convert it to a tensor (not shown for brevity)
# ...
sentiment_prediction = model(sample_tweet_tensor)
```

- Train an RNN model to classify tweet as positive or negative
- Output: "Positive"

RNN variation: LSTM

Tweet:

```
"Loved the cinematography,
hated the dialogue.
The acting was exceptional,
but the plot fell flat."
```

- Long Short Term Memory (LSTM) can capture complexities where RNNs may struggle

LSTM

LSTM architecture: Input gate, forget gate, and output gate

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        _, (hidden, _) = self.lstm(x)
        output = self.fc(hidden.squeeze(0))
        return output
```

RNN variation: GRU

- **Email subject:**

```
"Congratulations!
 You've won a free trip
 to Hawaii!"
```

- Gated Recurrent Unit (GRU) can quickly recognize spammy patterns without needing the full context

GRU

```
class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRUModel, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        _, hidden = self.gru(x)
        output = self.fc(hidden.squeeze(0))
        return output
```

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH

Evaluation metrics for text classification

DEEP LEARNING FOR TEXT WITH PYTORCH

Shubham Jain
Instructor

Why evaluation metrics matter

Spotlight on Book Reviews:

- Imagine a model that assesses the sentiment of book reviews
- The model claims a best-selling novel is poorly reviewed. Do we accept this?
- Use evaluation metrics

Evaluation RNN Models

```
# Initialize model, criterion, and optimizer
rnn_model = RNNModel(input_size, hidden_size, num_layers, num_classes)
...
# Model training
for epoch in range(10):
    outputs = rnn_model(X_train)
    ...
    print(f'Epoch: {epoch+1}, Loss: {loss.item()}')

outputs = rnn_model(X_test)
_, predicted = torch.max(outputs, 1)
```

Accuracy

- The ratio of correct predictions to the total predictions

```
from torchmetrics import Accuracy
actual = torch.tensor([0, 1, 1, 0, 1, 0])
predicted = torch.tensor([0, 0, 1, 0, 1, 1])
accuracy = Accuracy(task="binary", num_classes=2)
acc = accuracy(predicted, actual)
print(f"Accuracy: {acc}")
```

```
Accuracy: 0.6666666666666666
```

Beyond accuracy

- 10,000 reviews: 9,800 are positive
  - A model that always predicts positive: 98% accuracy
    - The model failed to classify negative reviews

- **Precision:** confidence in labeling a review as negative
- **Recall:** how well the model spots negative reviews
- **F1 Score:** balance between precision and recall

Precision and Recall

- **Precision:** correctly predicted positive observations / total predicted positives
- **Recall:** correctly predicted positive observations / all observations in the positive class

```
from torchmetrics import Precision, Recall
precision = Precision(task="binary", num_classes=2)
recall = Recall(task="binary", num_classes=2)
prec = precision(predicted, actual)
rec = recall(predicted, actual)
print(f"Precision: {prec}")
print(f"Recall: {rec}")
```

```
Precision: 0.6666666666666666
Recall: 0.5
```

Precision and Recall

```
Precision: 0.6666666666666666
Recall: 0.5
```

- Precision: 66.66% accurately predicted as positive
- Recall: captured 50% of positives

F1 score

- Harmonizes precision and recall
- Better measure for imbalanced classes

```
from torchmetrics import F1Score
f1 = F1Score(task="binary", num_classes=2)
f1_score = f1(predicted, actual)
print(f"F1 Score: {f1_score}")
```

```
F1 Score: 0.5714285714285715
```

- F1 Score of 1 = perfect precision and recall
- F1 Score of 0 = worst performance

Considerations

- Multiclass cores may be identical
  - Can indicate good model performance
- Always consider the problem when interpreting results!

Let's practice!

DEEP LEARNING FOR TEXT WITH PYTORCH