National Technical
University of Ukraine
"Igor Sikorsky
Kyiv Polytechnic Institute"

Institute of
Physics and
Technology

Intellectual Data Analysis

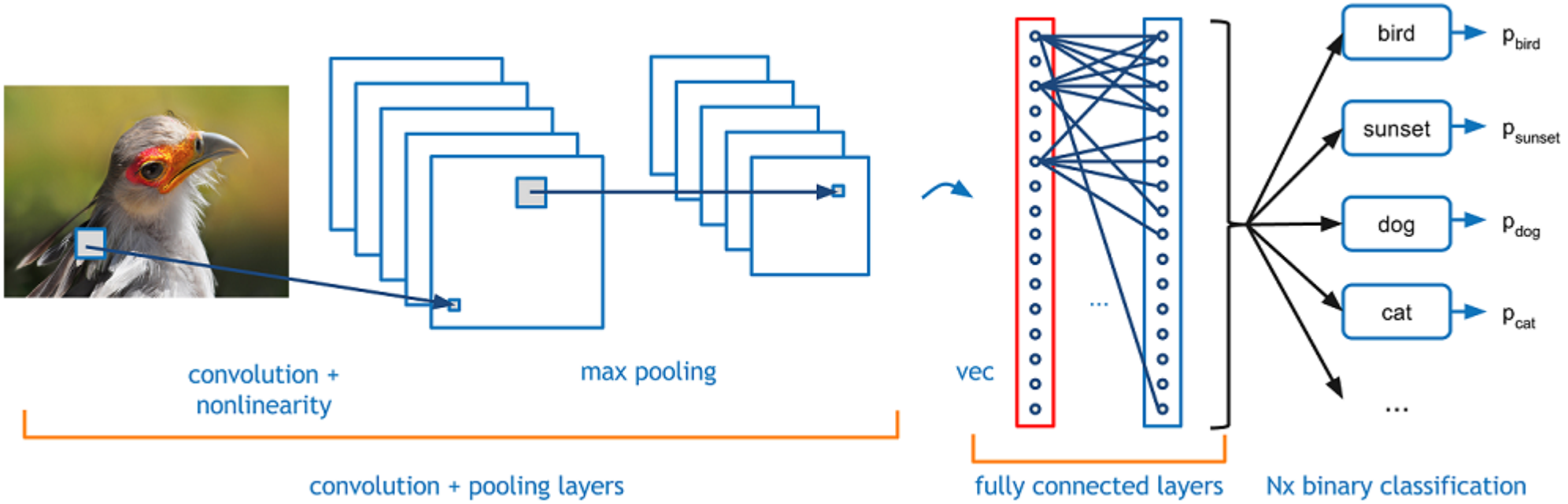
Practice 6: CNNs and RNNs

Dr. Nataliya K. Sakhnenko

Please review Lectures 8-9 before this practical

Conv Neural Nets

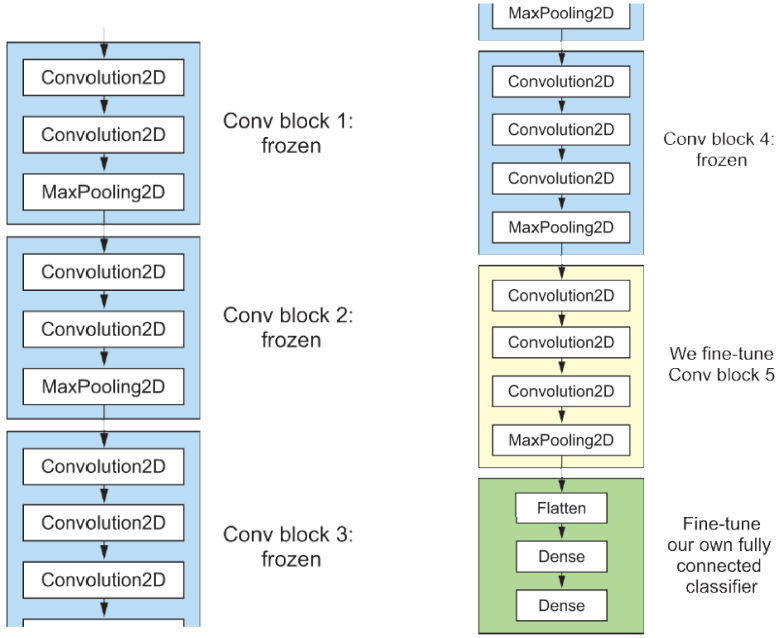
This layer takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes




```
nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
nn.ReLU()
nn.MaxPool2d(kernel_size=2, stride=2)
….
```

Transfer Learning: Fine Tuning of a pretrained model

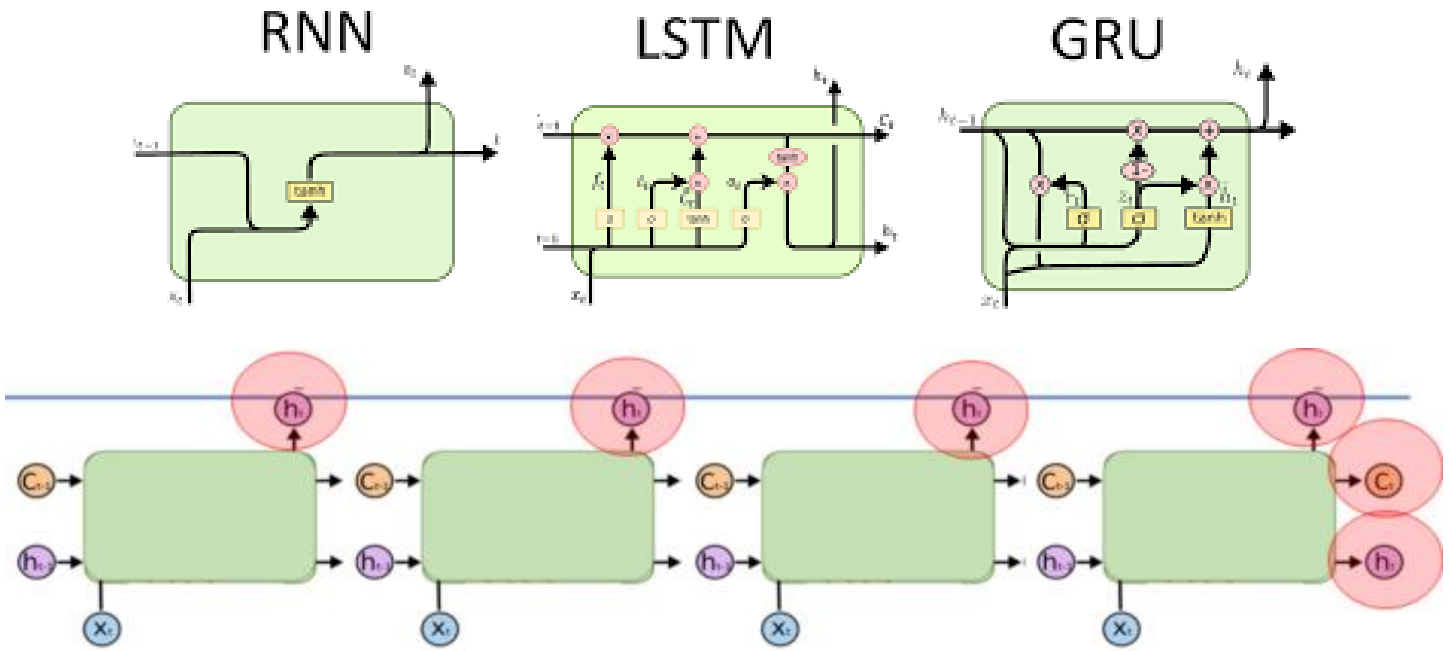
- Load pretrained model
  model = torchvision.models.vgg16(pretrained=True)
- Replace the classifier (head)
  model.classifier[-1] = nn.Linear(..., num_classes)
- Freeze convolutional base
  for param in model.features.parameters():
  param.requires_grad = False
- Train only the new classifier head
- Optional: Fine-tune last conv layers
  for param in model.features[-X:].parameters():
  param.requires_grad = True
- Train both: the unfrozen conv layers + classifier



Fine-tuning the last convolutional block of the VGG16 network

We reuse a pretrained convolutional *backbone*, freeze its weights, train only the new classification layers, and then optionally unfreeze the last few convolutional layers to fine-tune them for our task.

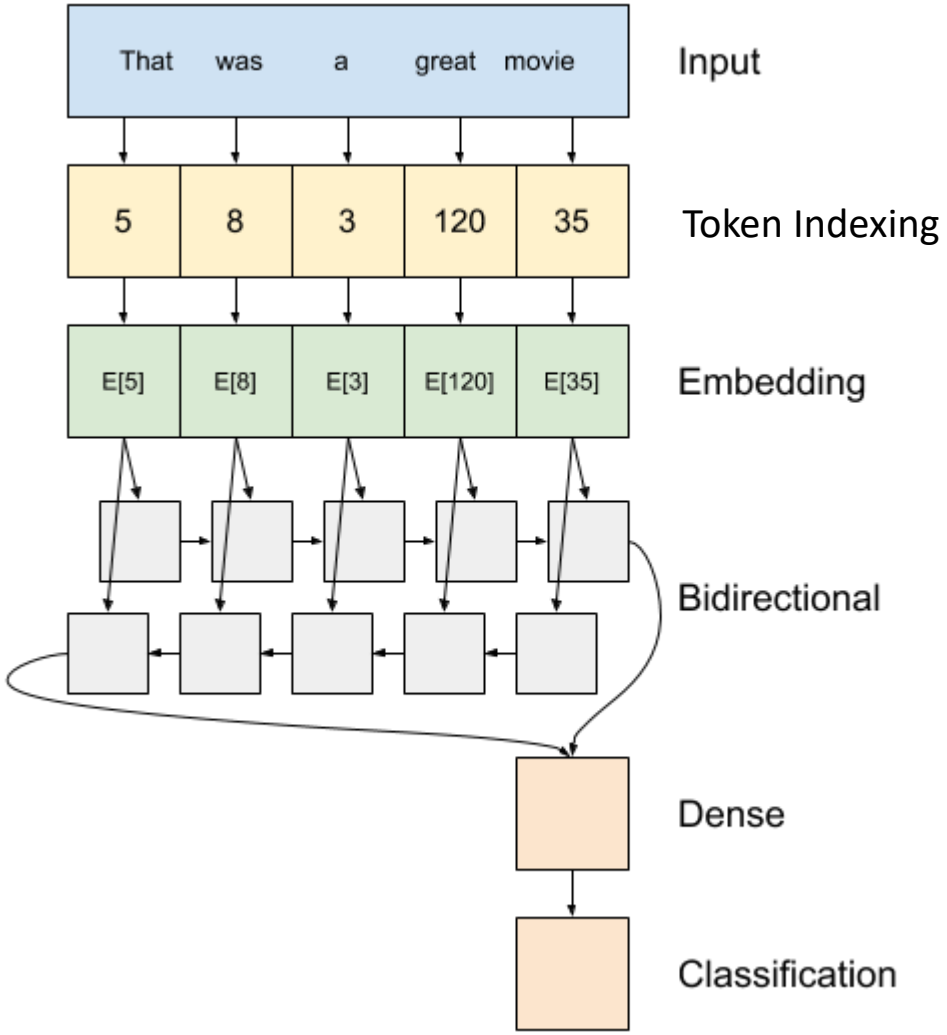
Recurrent Neural Nets




```
import torch.nn as nn

lstm = nn.LSTM(input_size=10, hidden_size=20, batch_first=True)
x = torch.randn(32, 50, 10) # [batch, time_steps, features]

output, (h_n, c_n) = lstm(x)

print(output.shape) -> torch.Size([32, 50, 20])
print(h_n.shape) - > torch.Size([1, 32, 20])

y_last = output[:, -1, :]              # last output only
print(y_last.shape) -> torch.Size([32, 20])
```

- **input_size:** dimensionality of the input features
- **hidden_size:** dimensionality of the hidden state
- **batch_first:** if True → input/output shape is *(batch, seq_len, features)* instead of *(seq_len, batch, features)*

Text classification with RNN



- Text classification with RNNs usually starts by converting text into integer sequences.
- Then an nn.Embedding layer maps each token to a dense vector, which is processed by a bidirectional LSTM or GRU to capture context.
- Finally, the last hidden state is passed through a fully connected layer for classification.

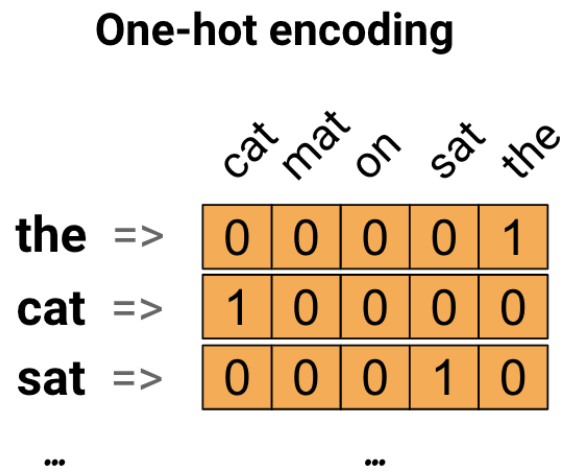
Embedding Layer

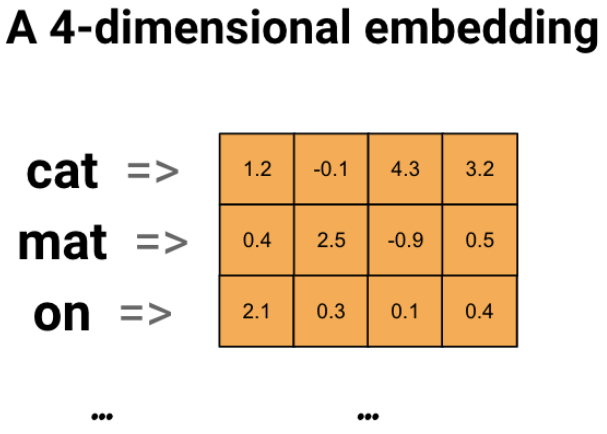
In PyTorch, nn.Embedding converts integer word indices into dense vectors

It turns positive integers (indexes) into dense vectors of fixed size.

- It can be used as part of a deep learning model where the embedding is learned along with the model itself.
- It can be used to load a pre-trained word embedding model, a type of transfer learning.

num_embeddings – vocabulary size
embedding_dim – dimension of each embedding vector
Input: word indices - **Output:** dense vectors






```
embedding = nn.Embedding(num_embeddings=6,
         embedding_dim=2)

# input: sequence of word indices
x = torch.tensor([[0, 1, 2],
                  [3, 4, 5]])        # [batch, seq_len]

out = embedding(x)

print(out.shape)  # torch.Size([2, 3, 2]) → [batch,
                            seq_len, embedding_dim]
print(out[0].shape)# torch.Size([3, 2]) embeddings for
                            the first sentence
```

index	Embedding
0	[1.2, 3.1]
1	[0.1, 4.2]
2	[1.0, 3.1]
3	[0.3, 2.1]
4	[2.2, 1.4]
5	[0.7, 1.7]
6	[4.1, 2.0]