A Shallow Introduction to Deep Learning

[Michael A. Alcorn]

Check out the links!

Deep Learning Necessities

- Python
  - Almost all deep learning research is done in Python
  - If you know how to program, a good way to learn Python is to read someone else's code and try to fully understand every single line
    - Python code can be very "Pythonic", which is really useful, but can look funny at first if you aren't used to it (e.g., list comprehensions)
  - Code like Mic(hael A. Alcorn)
    - PyCharm with Black
      - Good shortcuts to know (more)
        - Ctrl+B - go to the declaration of that thing or, if at the declaration, shows you where that thing is used
        - Ctrl+P - show the parameters of the function
    - IPython
- PyTorch
  - The majority of deep learning research uses PyTorch and it is continuing to grow in popularity
  - There's a ton of PyTorch code available on GitHub for many different model architectures
- Deep Learning - Goodfellow, Bengio, and Courville
  - For a deeper understand of the math and the "why" of deep learning
- Code for this tutorial

Python

```python
# List.
my_list = [0, "one"]  # You can initialize an empty list with [].
# Indexing.
print(my_list[0])
print(my_list[1])
# Appending.
my_list.append(pow)
print(my_list[2](2, 3))  # Equivalent to pow(2, 3).
# List comprehension.
my_list = [(x - 1) / 3 for x in range(10, 20, 3)]
# Equivalent to:
my_list = []
for x in range(10, 20, 3):
    my_list.append((x - 1) / 3)

# Dictionary.
my_dict = {"a": 1, 3: "three"}  # You can initialize an empty dictionary with {}.
# Accessing.
print(my_dict["a"])
print(my_dict[3])
# Inserting.
my_dict["my_list"] = my_list
```

```python
# NumPy things.
shape = (50, 10)
my_array = np.random.normal(size=shape)
print(my_array.shape)
(row_idx, col_idx) = (20, 5)
print(my_array[row_idx, col_idx])
# Access all rows and a single column.
print(my_array[:, col_idx])
# Access all rows except the first three and the last three and a single column.
print(my_array[3:-3, col_idx])

# Linear algebra.
(m, n, k) = (10, 20, 5)
A = np.random.normal(size=(m, n))
B = np.random.normal(size=(n, k))
# A @ B is equivalent to np.matmul(A, B).
C = A @ B
```

NumPy Resources

https://cs231n.github.io/python-numpy-tutorial/

https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html

Python

Using two nested `for` loops, create a list of seven lists `M` with no repeating values such that element `M[i][j] = 5 * i + j`

Common Neural Network Tasks

- Image classification - given an image, assign it a label (e.g., facial recognition)
- Image retrieval - given a query image, retrieve similar images from a database (e.g., Google Images Search)
- Object detection - given an image, identify the locations in the image containing different objects of interest
- Text classification - given some text, assign it a label (e.g., tagging)
- Information retrieval - given some query text, return relevant text from a database (e.g., Google Search)
- Translation - given some text in one language, convert it to another language
- Speech synthesis - given some text, generate speech
- Reinforcement learning - given a reward function, take actions that maximize the reward (e.g., playing Go)

Fully Connected Layers

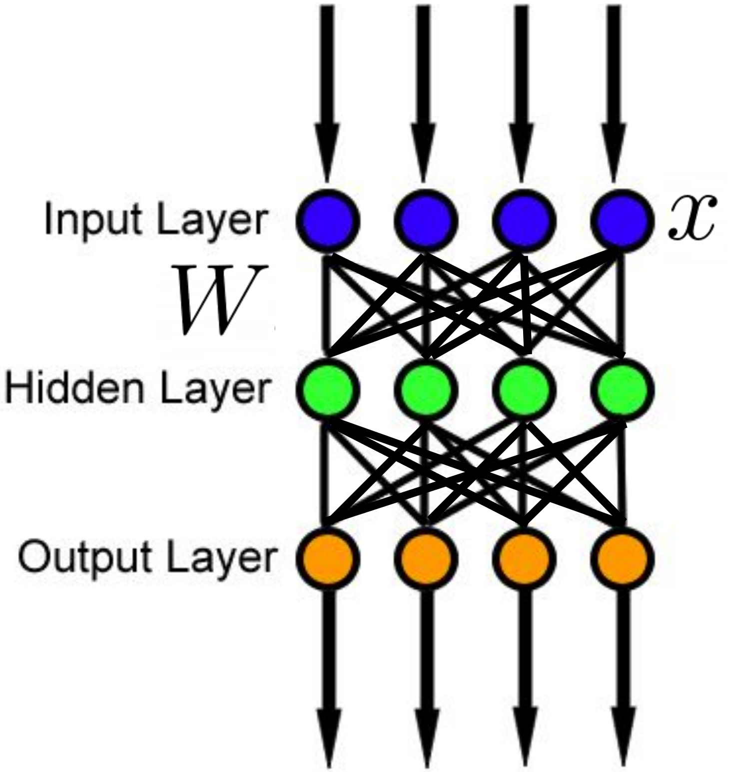
- Given a vector of input features *x*, a matrix of weights/parameters W, a bias vector *b*, and the rectifier activation function:

$$y = \max(0, Wx + b)$$

- In deep learning, we always work with batches of data, so the input is a matrix X where each row is a sample, i.e.:

$$y = \max(0, WX^\top + b)$$

- Combining several layers gives you a [multilayer perceptron]



Fully Connected Layers

NumPy

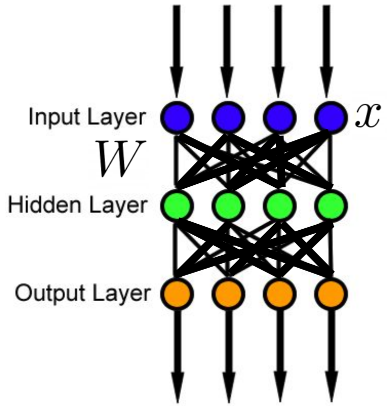
```
# Fully connected example.
batch_size = 32
features = 100
X = np.random.normal(size=(batch_size, features))

hidden_nodes = 50
W = np.random.normal(size=(hidden_nodes, features))
b = np.random.normal(size=(hidden_nodes, 1))

out = W @ X.T + b
# Rectifier activation function.
out[out < 0] = 0
print(out.T)
```

What is the shape of the output?

PyTorch

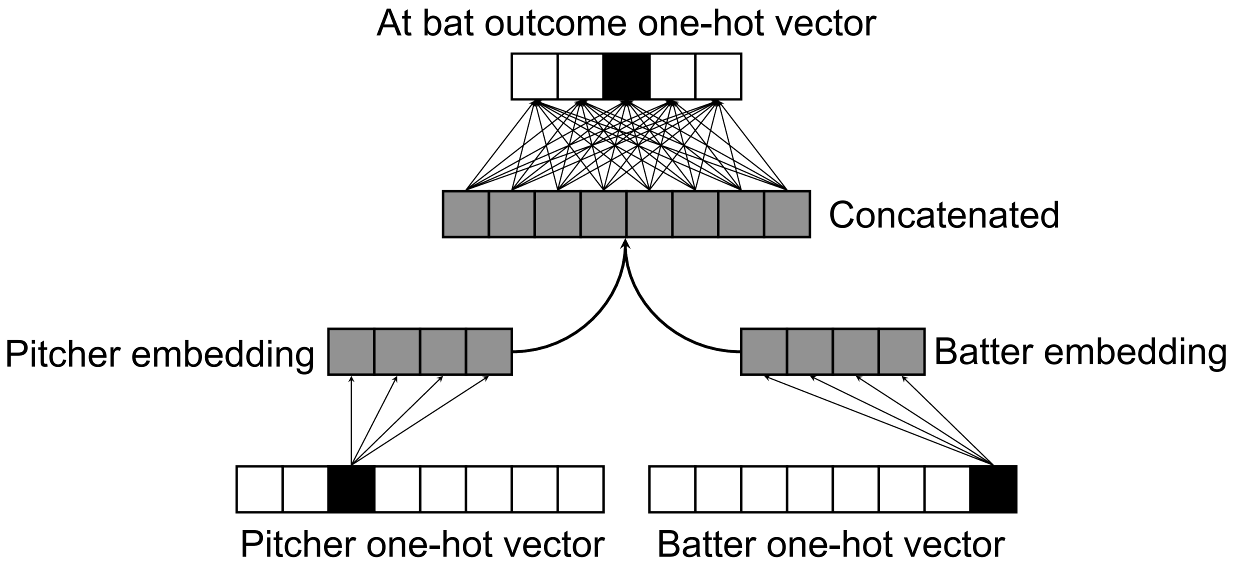
```
fc = nn.Linear(features, hidden_nodes)
out = fc(torch.Tensor(X))
out = nn.functional.relu(out)
# Same as above.
print(out)
```

$$y = \max(0, W X^\top + b)$$



Embeddings

- Convert one-hot categorical data into lower dimensional dense vectors
- Embeddings often have intuitive neighborhoods and algebraic properties



[(batter|pitcher)2vec]

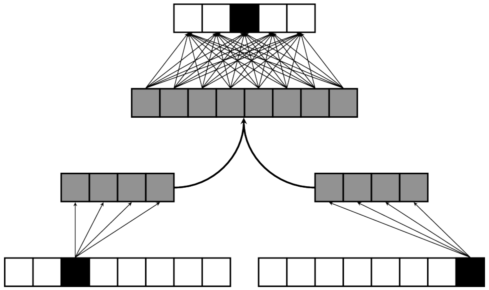
Embeddings

NumPy

```python
n_batters = n_pitchers = 100
batter_idxs = np.random.randint(n_batters, size=batch_size)
batter_one_hots = np.zeros((batch_size, n_batters))
batter_one_hots[np.arange(batch_size), batter_idxs] = 1
pitcher_idxs = np.random.randint(n_pitchers, size=batch_size)
pitcher_one_hots = np.zeros((batch_size, n_pitchers))
pitcher_one_hots[np.arange(batch_size), pitcher_idxs] = 1

embedding_dim = 9
W_b = np.random.normal(size=(embedding_dim, n_batters))
W_p = np.random.normal(size=(embedding_dim, n_pitchers))

# Equivalent to:
# batter_embeds = W_b[:, batter_idxs]
batter_embeds = W_b @ batter_one_hots.T
# Equivalent to:
# pitcher_embeds = W_p[:, pitcher_idxs]
pitcher_embeds = W_p @ pitcher_one_hots.T

cat_embeds = np.hstack([batter_embeds.T, pitcher_embeds.T])
print(cat_embeds[[0, -1]])
```

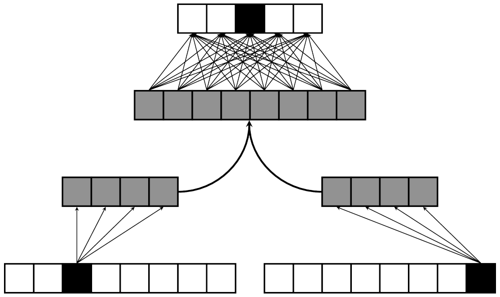
PyTorch

```python
batter_embed = nn.Embedding(n_batters, embedding_dim)
pitcher_embed = nn.Embedding(n_pitchers, embedding_dim)
batter_embeds = batter_embed(torch.LongTensor(batter_idxs))
pitcher_embeds = pitcher_embed(torch.LongTensor(pitcher_idxs))
cat_embeds = torch.cat([batter_embeds, pitcher_embeds], dim=1)
print(cat_embeds)
```



What is the shape of the output?

Building a Model



- Think of model classes as “function constructors”, i.e., defining how inputs are used to produce outputs (akin to functional programming)
- Whenever you encounter a new PyTorch model for the first time, look at the **`forward`** function first
    - From there, you can investigate specific model components as you encounter them

What is the shape of the output?

```
class BatterPitcher2Vec(nn.Module):
    def __init__(self, n_batters, n_pitchers, embedding_dim, n_outcomes):
        super().__init__()
        self.batter_embed = nn.Embedding(n_batters, embedding_dim)
        self.pitcher_embed = nn.Embedding(n_pitchers, embedding_dim)
        self.sig = nn.Sigmoid()
        self.fc = nn.Linear(2 * embedding_dim, n_outcomes)

    def forward(self, batter_idxs, pitcher_idxs):
        batter_embeds = self.batter_embed(batter_idxs)
        pitcher_embeds = self.pitcher_embed(pitcher_idxs)
        cat_embeds = torch.cat([batter_embeds, pitcher_embeds], dim=1)
        return self.fc(self.sig(cat_embeds))


n_outcomes = 20
model = BatterPitcher2Vec(n_batters, n_pitchers, embedding_dim, n_outcomes)
print(model)
batch_size = 32
test_batters = torch.randint(n_batters, (batch_size,))
test_pitchers = torch.randint(n_pitchers, (batch_size,))
out = model(test_batters, test_pitchers)
```

Building a Model

Extend `BatterPitcher2Vec` so that:

1. The new model class is named `BatterPitcher2VecExt`
2. The inning and the number of runners on base are inputs
3. There is a fully connected hidden layer with a [`ReLU`] activation function before the classification layer

Test your model on random batter, pitcher, inning, and runners on base inputs.

Training a Model

- PyTorch uses `Dataset`s and `DataLoader`s for convenient online generation of training data
  - Easily parallelizable
  - Highly flexible for experimenting with different preprocessing steps, etc.

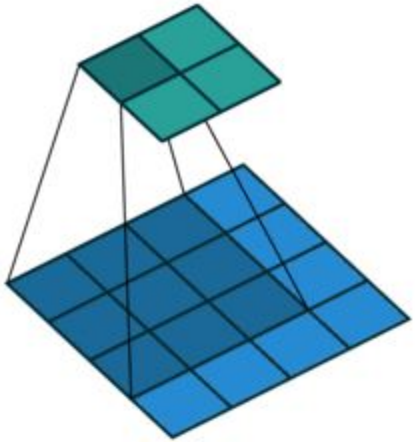
```python
class BatterPitcher2VecDataset(Dataset):
    def __init__(self):
        N = 128
        self.batter_idxs = np.random.randint(n_batters, size=N)
        self.pitcher_idxs = np.random.randint(n_pitchers, size=N)
        self.outcomes = np.random.randint(n_outcomes, size=N)

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

    def __getitem__(self, idx):
        return {
            "batter": torch.LongTensor([self.batter_idxs[idx]]),
            "pitcher": torch.LongTensor([self.pitcher_idxs[idx]]),
            "outcome": torch.LongTensor([self.outcomes[idx]]),
        }
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BatterPitcher2Vec(n_batters, n_pitchers, embedding_dim, n_outcomes).to(device)

criterion = nn.CrossEntropyLoss()
train_params = [params for params in model.parameters()]
learning_rate = 1e-1
optimizer = torch.optim.Adam(train_params, lr=learning_rate)

train_dataset = BatterPitcher2VecDataset()
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = train_loader

# Train model.
epochs = 200
best_val_loss = np.inf
for epoch in range(epochs):
    model.train()
    for train_tensors in train_loader:
        optimizer.zero_grad()
        pred_logits = model(
            train_tensors["batter"].flatten().to(device),
            train_tensors["pitcher"].flatten().to(device),
        )
        loss = criterion(pred_logits, train_tensors["outcome"].flatten().to(device))
        loss.backward()
        optimizer.step()

    model.eval()
    val_loss = 0
    for valid_tensors in valid_loader:
        with torch.no_grad():
            pred_logits = model(
                valid_tensors["batter"].flatten().to(device),
                valid_tensors["pitcher"].flatten().to(device),
            )
            val_loss += criterion(
                pred_logits, valid_tensors["outcome"].flatten().to(device)
            ).item()

    print(val_loss, flush=True)
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), "batter_pitcher2vec.pth")

model.load_state_dict(torch.load("batter_pitcher2vec.pth"))
```

Convolution

- Images are extremely high-dimensional
  - Typical RGB image resolution for training classifiers is 224x224 → 224x224x3 channels = 150,528 input features
  - 1,000 hidden nodes in fully connected layer = 150 million parameters!
- But pixel colors are spatially correlated...
  - CNNs are designed to exploit this fact
- Instead of learning a large weight matrix, the network learns many small filters that are translated over the image
  - 1,000 7x7 filters + 1,000 bias = 50,000 parameters << 150 million
- More resources

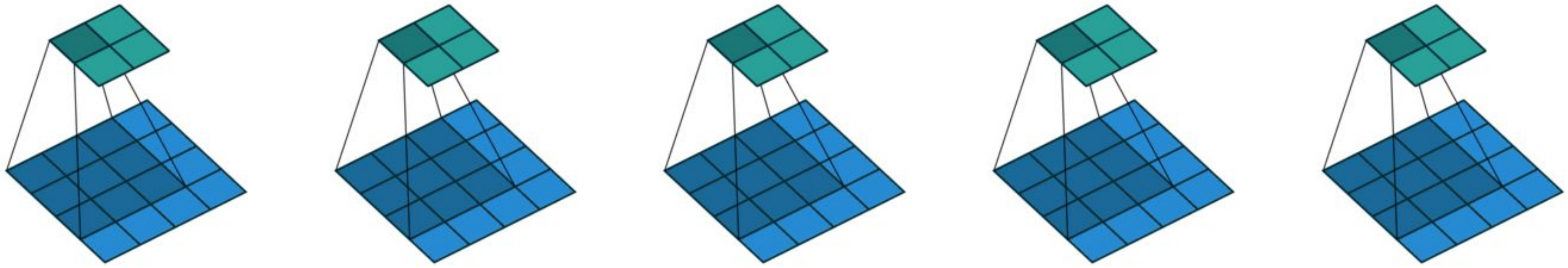
4x4 blue squares = image
3x3 dark blue squares = filter
2x2 green squares = output
1x1 dark green square = filter output for one 3x3 region of input



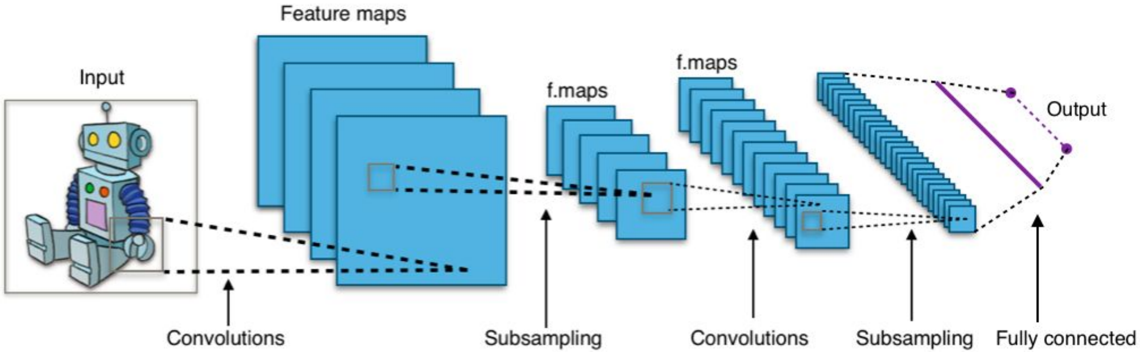
https://github.com/vdumoulin/conv_arithmetic

Convolution

- One layer containing k = 5 filters (each filter has different learnable weights)



- Stack the k filter outputs at the end to create a new “image” that will be fed into the next convolutional layer



Wikipedia

Convolution

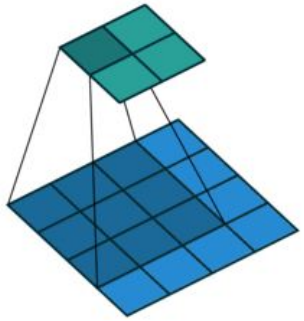
NumPy

```python
rgb_feats = 3
img_size = 4
X = np.random.normal(size=(batch_size, rgb_feats, img_size, img_size))

num_filters = 5
filter_size = 3
W = np.random.normal(size=(num_filters, rgb_feats, filter_size, filter_size))
b = np.random.normal(size=(num_filters,))

# Only process one image.
in_img = X[0]
outs = []
for filter_idx in range(num_filters):
    filter_W = W[filter_idx].flatten()
    filter_b = b[filter_idx]
    filter_outputs = []
    out_size = img_size - filter_size + 1
    for win_row in range(out_size):
        for win_col in range(out_size):
            img_win = in_img[
                :, win_row : win_row + filter_size, win_col : win_col + filter_size
            ]
            filter_outputs.append(filter_W @ img_win.flatten() + filter_b)

    filter_outputs = np.array(filter_outputs).reshape((out_size, out_size))
    outs.append(filter_outputs)

outs = np.stack(outs)
outs[outs < 0] = 0
print(outs)
```

*technically cross-correlation

PyTorch

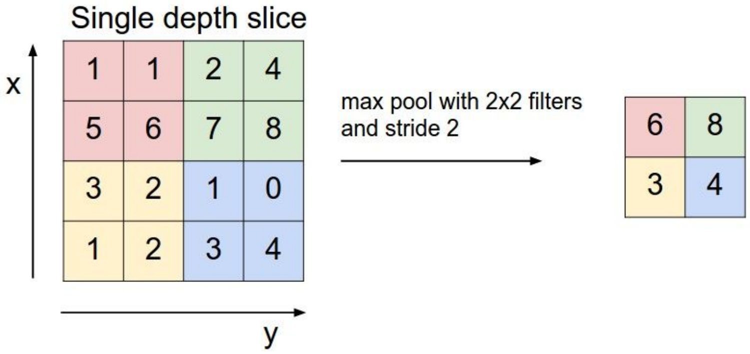
```python
conv = nn.Conv2d(rgb_feats, num_filters, filter_size)
out = conv(torch.Tensor(X))
out = nn.functional.relu(out)
print(out[0])
```



What is the shape of the output?

Pooling

- "Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer."
- Is pooling necessary?




```
avg_pool = nn.AdaptiveAvgPool2d((1, 1))
out = avg_pool(out)
print(out.shape)
```

https://cs231n.github.io/convolutional-networks/#pool

Convolutional Neural Networks
(CNNs)

Implement a CNN classifier where:

1. The first convolutional layer has 15 3x3 filters.
2. The second convolutional layer has 30 5x5 filters.
3. There is an `ReLU` nonlinearity following each convolutional layer.
4. Average pooling is applied before the classification layer.
5. There are four classes.

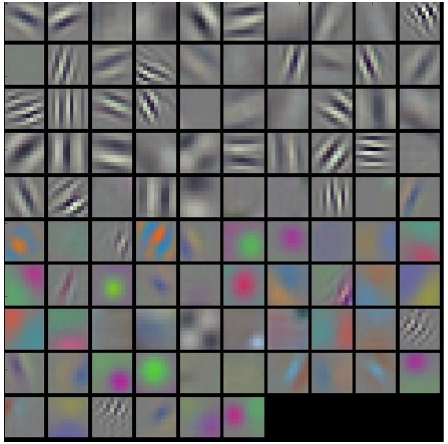
Test your model on a 32x32 image, i.e.:

```
test_tensor = torch.rand((1, 3, 32, 32))
```

Hint: you'll need to use the [`Flatten`] layer (where? why?)

Convolutional Neural Networks
(CNNs)

- What do convolutional filters learn?
  - At early layers: edges, color gradients, other patterns
  - Later layers detect more and more abstract concepts
- Neural networks are "feature extractors"



AlexNet filters from: https://cs231n.github.io/understanding-cnn/

Transfer Learning

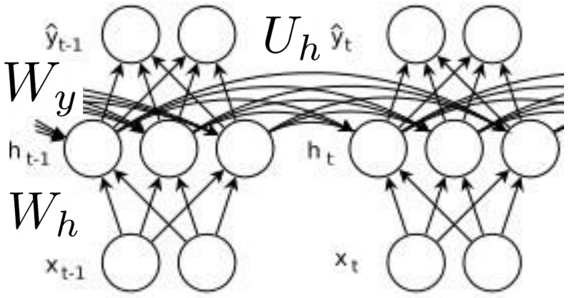
- Use knowledge gained by neural network trained on different dataset/task to learn new dataset/task with smaller amount of data
- A common strategy in computer vision is to fine-tune a CNN trained on ImageNet
    - Annual Fine-Grained Visual Categorization competition at CVPR
        - How to train a neural network to recognize species that have a limited number of observations (on iNaturalist; my observations)?
            - Winners fine-tuned pretrained neural networks!
- Similar trend in natural language processing
- Further reading:
    - https://ruder.io/transfer-learning/
    - https://cs231n.github.io/transfer-learning/
    - PyTorch has a number of pretrained models readily available

Recurrence

- How to handle sequential data?
- Algorithm
    - Process input at current time step
    - Process memory of what happened before
    - Combine to create new state
    - Use updated state to make prediction
- What about h_0?
    - Use zeros or make it learnable
- [More resources]

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

[Wikipedia]



[Martens and Sutskever, 2011]

Recurrence

NumPy

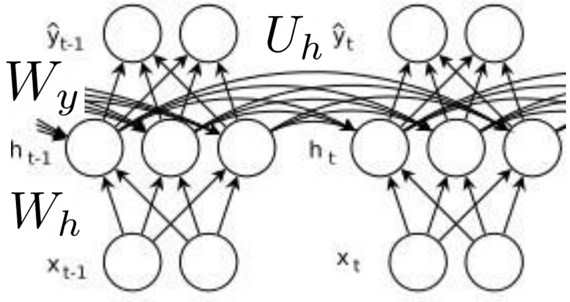
```python
seq_len = 10
X = np.random.random(size=(seq_len, batch_size, features))
W_h = np.random.normal(size=(hidden_nodes, features))
U_h = np.random.normal(size=(hidden_nodes, hidden_nodes))
b_h = np.random.normal(size=(hidden_nodes, 1))

# Only process one sequence.
in_seq = X[:, 0]
h = np.zeros((hidden_nodes, 1))
out_hs = []
for step in range(seq_len):
    # None index keeps the dimensions the same as the input.
    h = np.tanh(W_h @ in_seq[None, step].T + U_h @ h + b_h)
    out_hs.append(h)

out = np.hstack(out_hs).T
print(out)
```

What is the shape of the output?

PyTorch

```python
rnn = nn.RNN(features, hidden_nodes)
(out, last_h) = rnn(torch.Tensor(X))
print(out[:, 0])
```

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$



Recurrent Neural Networks
(RNNs)

Implement a RNN classifier where:

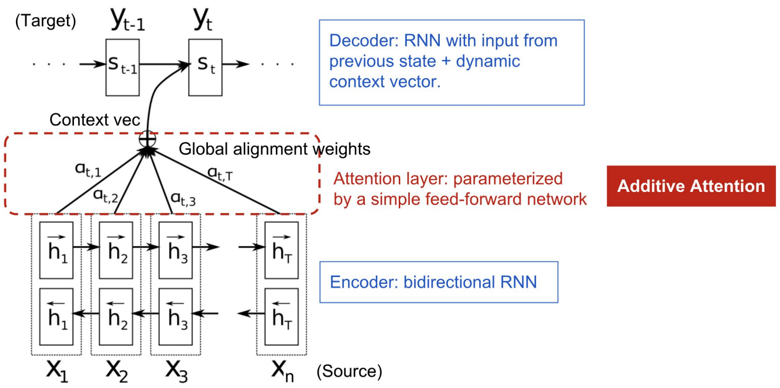
1. Sequences have 10 input features at each step.
2. The RNN is an LSTM with 50 hidden nodes.
3. There are seven classes.

Test your model on a sequence with 15 steps, i.e.:

```
test_tensor = torch.rand((15, 1, 10))
```

Transformers

- Sequence modeling without recurrence
  - Do things in parallel!
- Key idea → attention
- Self-attention mechanism
  - At time step t, look at *all* previous time steps and assign them weights using a scoring function
  - Take the weighted sum of the previous time steps → context vector
  - Use context vector and current time step vector to generate an output
- More resources
- GPT-3



Lil' Log
(technically just "attention", not "self-attention", but they work the same)

History of “Attention”

$$\phi(t,u) = \sum_{k=1}^{K} \alpha_t^k \exp\left(-\beta_t^k \left(\kappa_t^k - u\right)^2\right)$$

$$w_t = \sum_{u=1}^{U} \phi(t,u) c_u$$

“Generating Sequences With Recurrent Neural Networks”
(Graves, arXiv 2013)

$$o_1 = O_1(x, \mathbf{m}) = \underset{i=1,\ldots,N}{\arg\max}\; s_O(x, \mathbf{m}_i)$$

“Memory Networks”
(Weston, Chopra, and Bordes, arXiv 2014/ICLR 2015)

$$\sum_i w_t(i) = 1, \qquad 0 \le w_t(i) \le 1, \ \forall i.$$

$$\mathbf{r}_t \longleftarrow \sum_i w_t(i) \mathbf{M}_t(i),$$

“Neural Turing Machines”
(Graves, Wayne, and Danihelka, arXiv 2014)

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

“Neural Machine Translation by Jointly Learning to Align and Translate”
(Bahdanau , Cho, and Bengio, arXiv 2014/ICLR 2015)

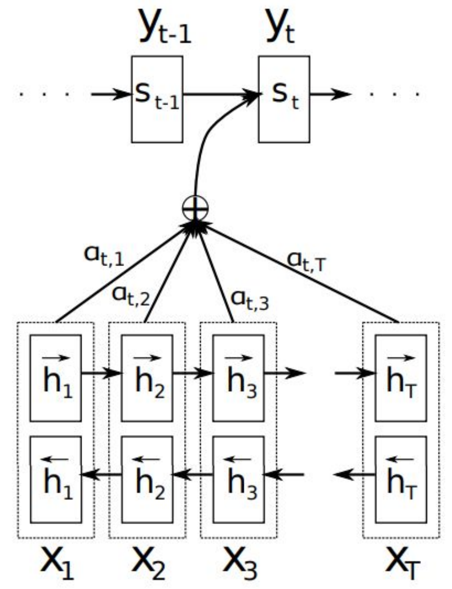


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

Transformers

Attention

```
(h1, h2) = (50, 30)
attn = nn.Sequential(
    nn.Linear(2 * features, h1),
    nn.ReLU(),
    nn.Linear(h1, h2),
    nn.ReLU(),
    nn.Linear(h2, 1),
)

total_time_steps = 10
X = torch.rand(total_time_steps, features)
t = 7
x_t = X[None, t]
pre_t_xs = X[:t]
pre_t_xs_w_x_t = torch.cat([pre_t_xs, x_t.expand(t, -1)], dim=1)
scores = attn(pre_t_xs_w_x_t)
c = pre_t_xs.T @ scores
x_t_with_c = torch.cat([x_t, c.T], dim=1)
print(x_t_with_c)
```

What is the shape of the output?

Transformer

```
X = np.random.random(size=(seq_len, batch_size, features))
nhead = 5
encoder_layers = TransformerEncoderLayer(features, nhead, hidden_nodes)
num_layers = 3
transformer_encoder = TransformerEncoder(encoder_layers, num_layers)
seq_mask = (torch.triu(torch.ones(seq_len, seq_len)) == 1).transpose(0, 1)
seq_mask = (
    seq_mask.float()
    .masked_fill(seq_mask == 0, float("-inf"))
    .masked_fill(seq_mask == 1, float(0.0))
)
out = transformer_encoder(torch.Tensor(X), seq_mask)
print(out[:, 0])
```