Pre-Project

EEE 598 Spring 2019

Pre-Project Part 1:
Intro to PyTorch + Transfer Learning

EEE 598 Spring 2019

Pre-Project Part 1

- Learn PyTorch
- Extract features from pre-trained networks and use them with an SVM
- Fine-tune pre-trained networks on new datasets

Deep Learning Libraries

- Caffe – Facebook/Berkeley
- Theano – U of Montreal
- PyTorch - Facebook
- Tensorflow - Google
- CNTK - Microsoft
- MxNet - Apache
- … many more

Why use PyTorch?

- Auto-grad
  - Many libraries have this. This makes things much easier than implementing backwards functions for every layer
- Dynamic computational graph
  - Easier to debug than a static computational graph
- Recently popular for research
  - Note that industry prefers Tensorflow

PyTorch example

First import the required packages

In [1]:
```
import torch
from torch.autograd import Variable
```

Next define some variables

In [2]:
```
x = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
print(x)
```

```
Variable containing:
 1  2
 3  4
[torch.FloatTensor of size 2x2]
```

In [3]:
```
w = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
print(w)
```

```
Variable containing:
 1  2
 3  4
[torch.FloatTensor of size 2x2]
```

PyTorch example

Now do some operations

In [5]:
```
y = torch.mm(w,x)
print(y)
```

```
Variable containing:
  7  10
 15  22
[torch.FloatTensor of size 2x2]
```

In [6]:
```
z = y.sum()
print(z)
```

```
Variable containing:
 54
[torch.FloatTensor of size 1]
```

PyTorch example

Call the backward function which will compute all of the gradients

[7]:
```
z.backward()
```

We get the gradient of z with respect to x automatically without needing to write any extra backwards code

[8]:
```
print(x.grad)
```

```
Variable containing:
 4  4
 6  6
[torch.FloatTensor of size 2x2]
```

PyTorch Modules

- nn
  - Includes functionality for common deep networks
  - Layers: conv2d, full, etc
  - Functional: dot product, convolution operations, etc
- Torchvision
  - Functionality for dealing with images
  - Prebuilt models
  - Objects for loading data
- Torchtext
  - Functionality for dealing with text

PyTorch Layer

```
class Linear(Module):
    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)
```

He init

Will run on GPU if available.

Also auto-magically computes gradients

CIFAR100 in PyTorch

```
import torch.nn as nn
import torch.nn.functional as F
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.fc1 = nn.Linear(1152, 5)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 1152)
        x = self.fc1(x)
        return x
```

CIFAR100 in PyTorch - Training

```
    def fit(self, trainloader):
        # switch to train mode
        self.train()

        # define loss function
        criterion = nn.CrossEntropyLoss()

        # setup SGD
        optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.0)

        for epoch in range(20):  # loop over the dataset multiple
    times
            running_loss = 0.0
```

CIFAR100 in PyTorch - Training

```
for i, data in enumerate(trainloader, 0):
    # get the inputs
    inputs, labels = data

    # zero the parameter gradients
    optimizer.zero_grad()

    # compute forward pass
    outputs = self.forward(inputs)

    # get loss function
    loss = criterion(outputs, labels)
```

CIFAR100 in PyTorch - Training

```
for i, data in enumerate(trainloader, 0):
```

```
        # do backward pass
        loss.backward()

        # do one gradient step
        optimizer.step()

        # print statistics
        running_loss += loss.item()

print('[Epoch: %d] loss: %.3f' %
      (epoch + 1, running_loss / (i+1)))
```

CIFAR100 in PyTorch - Testing

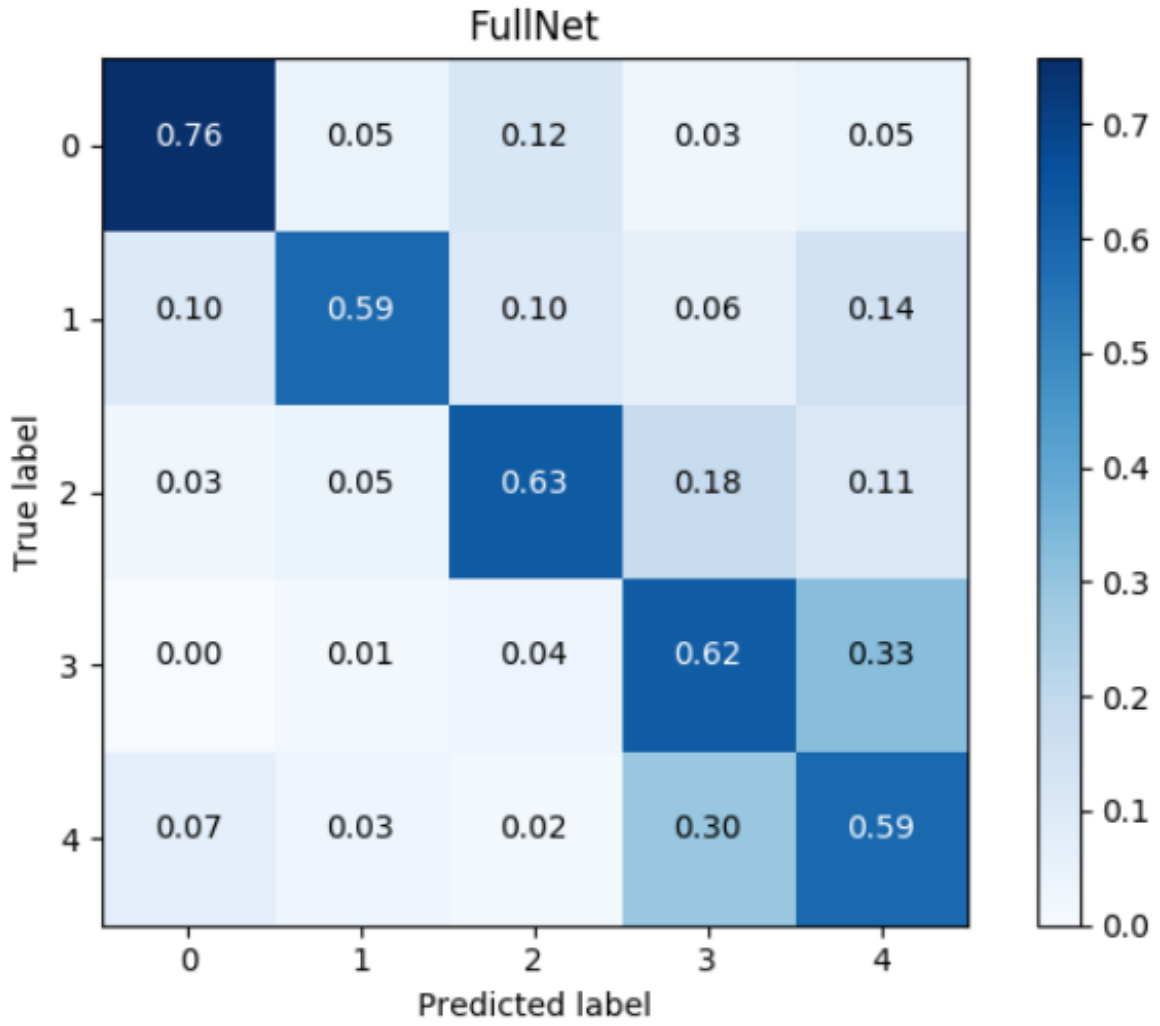
```
    def predict(self, testloader):
        # switch to evaluate mode
        self.eval()

        correct = 0
        total = 0
        all_predicted = []
        with torch.no_grad():
            for images, labels in testloader:
                outputs = self.forward(Variable(images))
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                all_predicted += predicted.numpy().tolist()

        print('Accuracy on test images: %d %%' % (
            100 * correct / total))

        return all_predicted
```

CIFAR100 in PyTorch - Testing

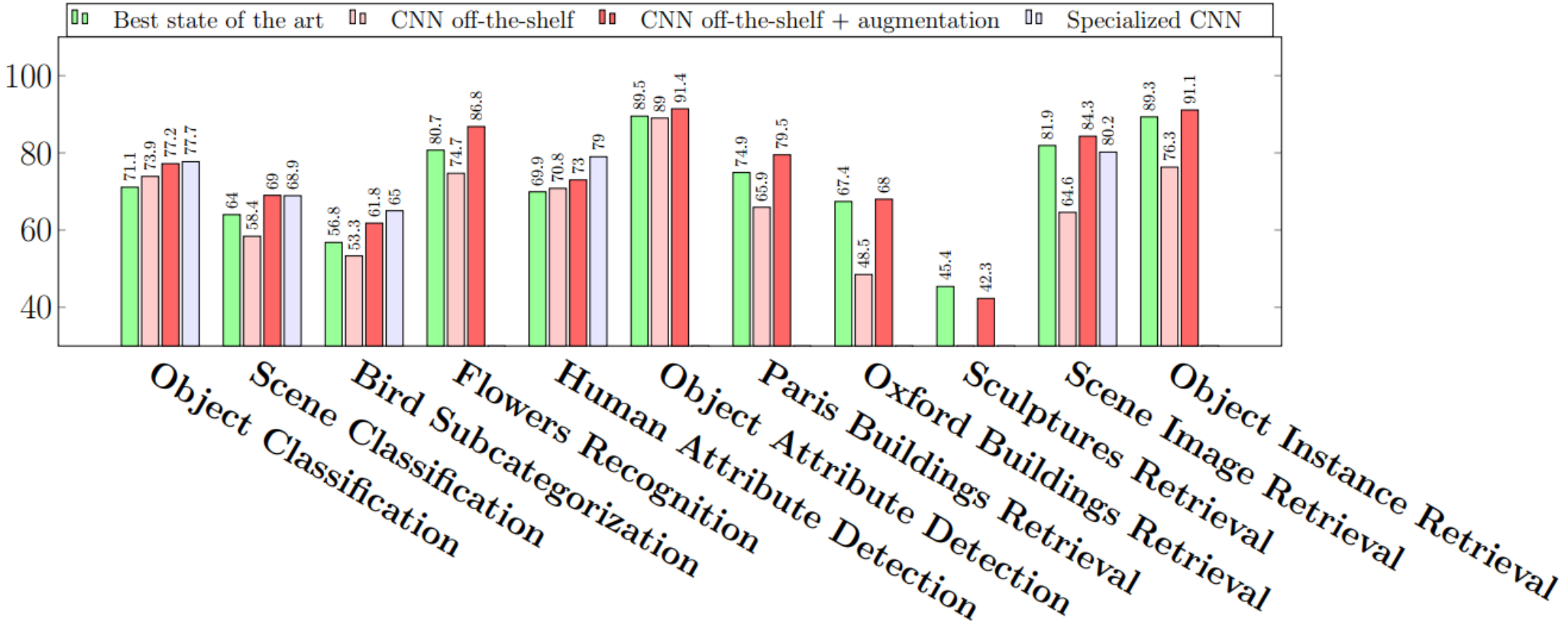


Transfer Learning

- How can we use a network that was previously trained for a different problem and adapt it for our problem?
  - Use the features computed by the network as inputs to an SVM
  - Or fine-tune (retrain) the network on the new problem

Feature Extraction + SVM

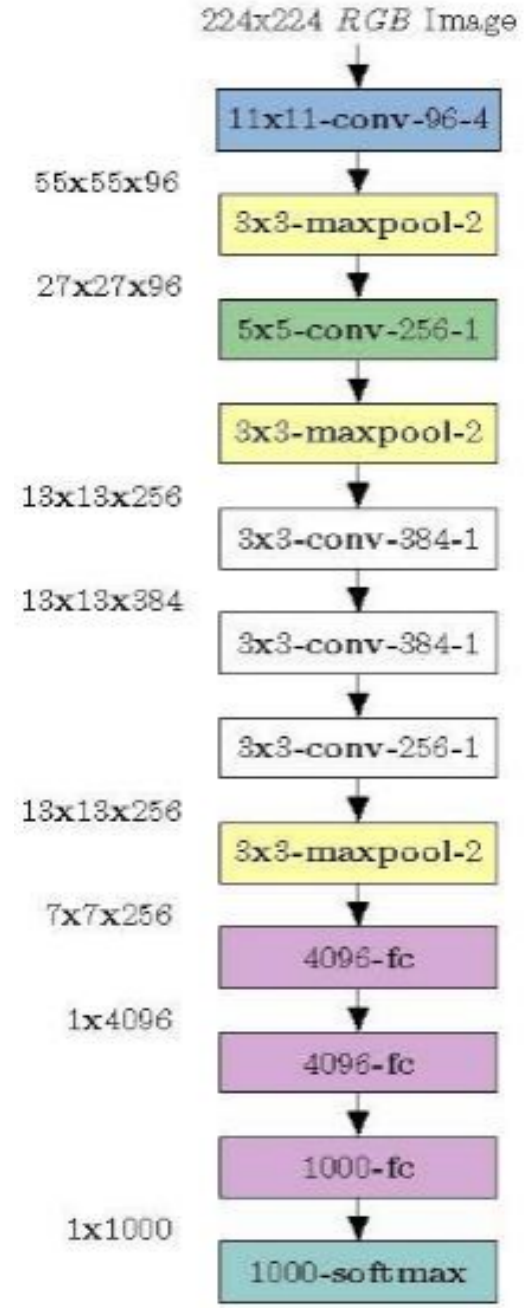
- Can easily beat complicated pre-deep learning based methods



Razavian, Ali Sharif, et al. "CNN features off-the-shelf: an astounding baseline for recognition." Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on. IEEE, 2014.

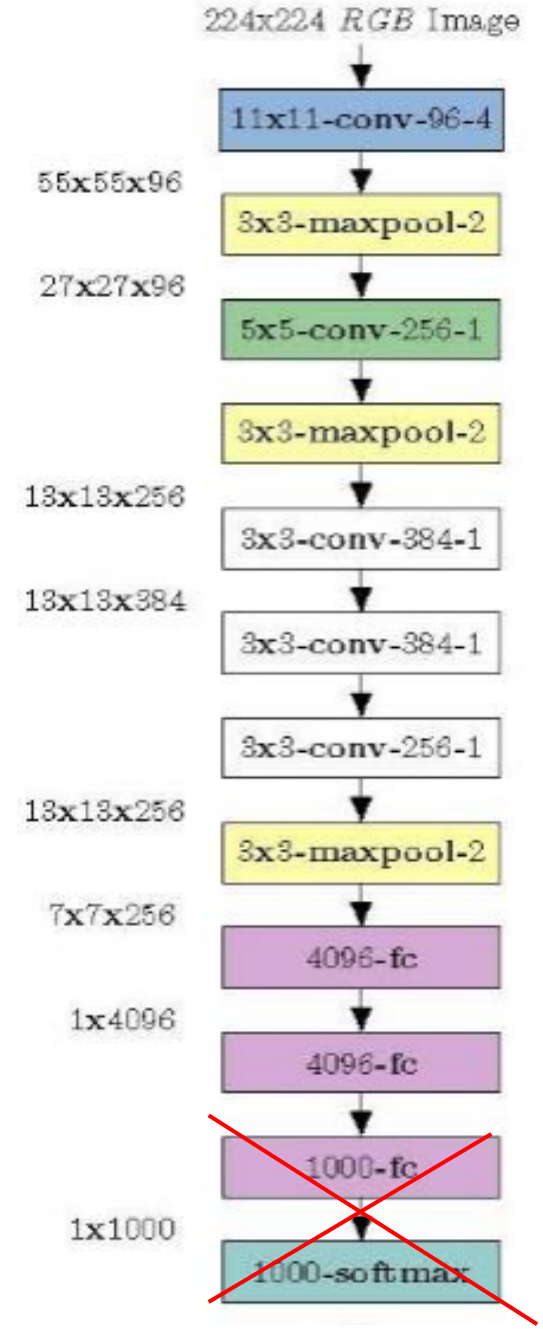
Fine-tuning

- Start with a network trained on a very large dataset
  - Problem is our dataset has different number of classes



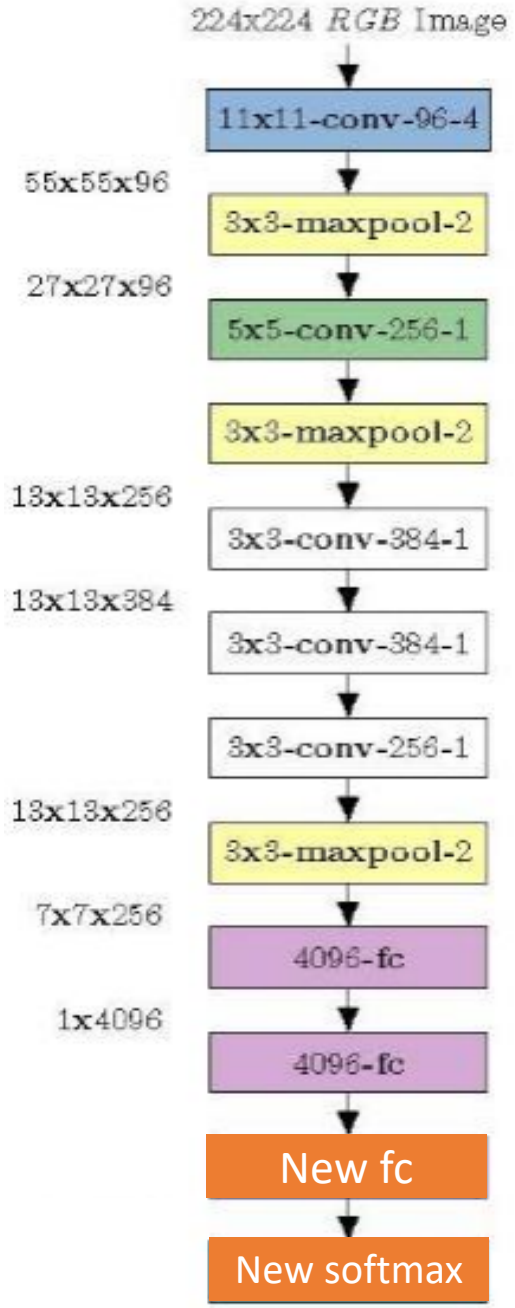
Fine-tuning

- Start with a network trained on a very large dataset
  - Problem is our dataset has different number of classes
- So chop off last 1000 output fc layer and 1000 way softmax



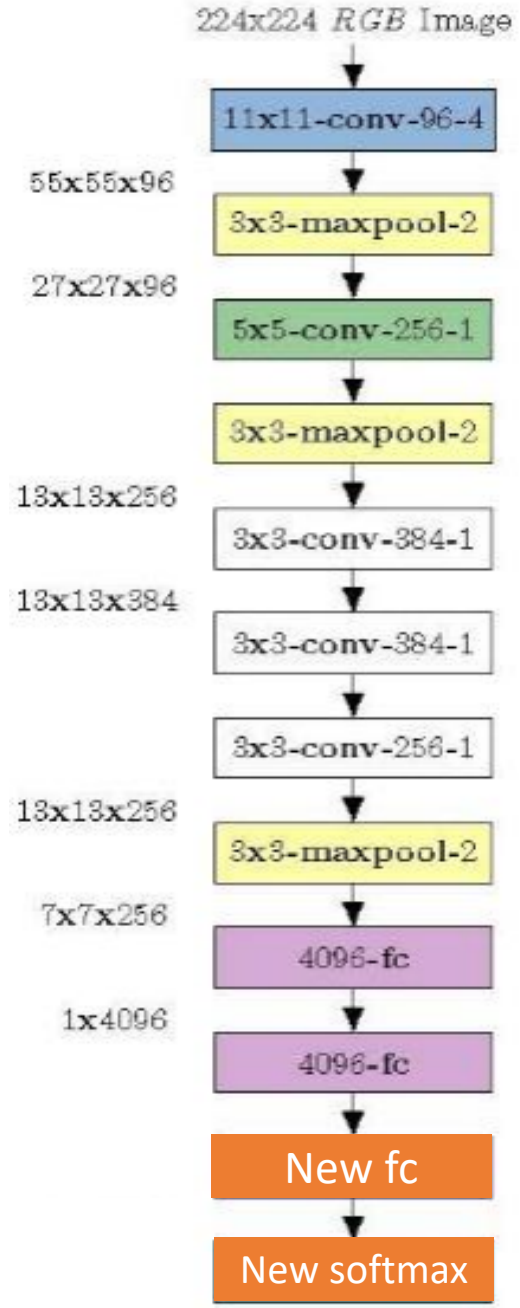
Fine-tuning

- Start with a network trained on a very large dataset
  - Problem is our dataset has different number of classes
- So chop off last 1000 output fc layer and 1000 way softmax
- Replace with a new layer with number of outputs equal to number of classes in new dataset

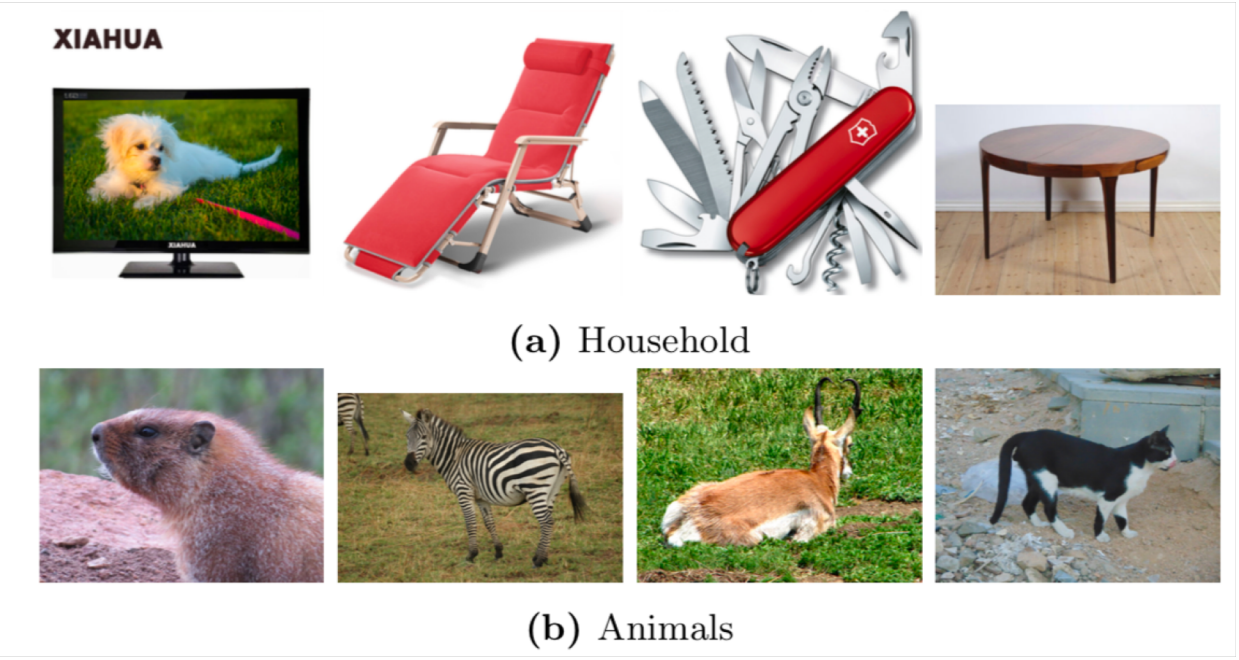


Fine-tuning

- Now retrain network on new data.
  - Can train only the last layer while holding other layers fixed
  - Or can have a higher learning rate in last layer than other layers
  - Or can train a subset of the layers (e.g. last 3 layers)

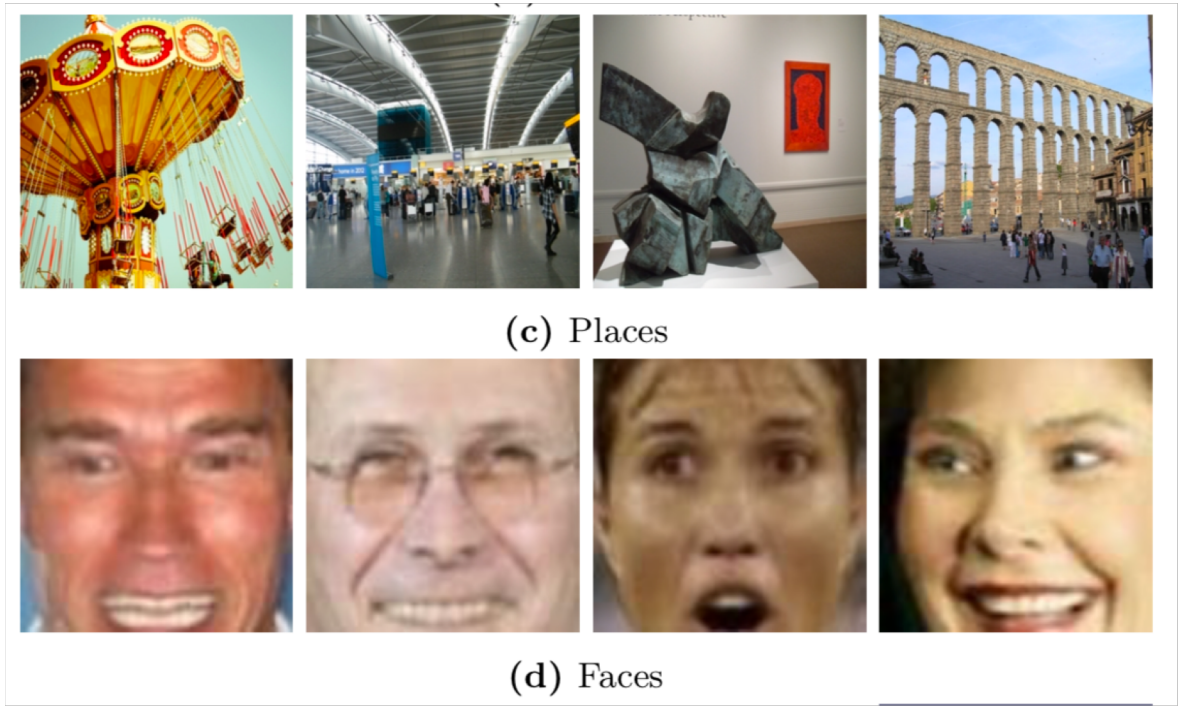


Pre-Project Part 1 Datasets



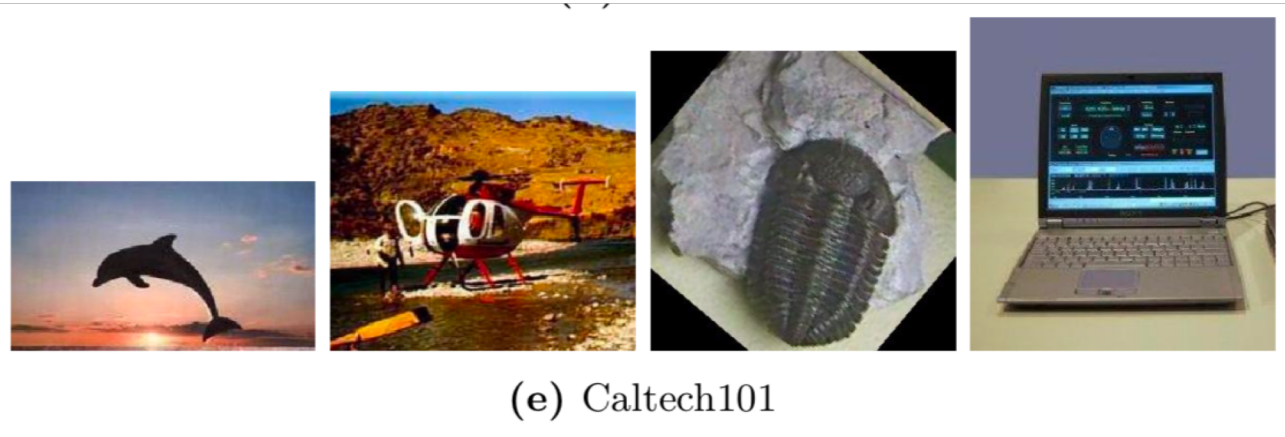
(a) Household

(b) Animals



(c) Places

(d) Faces



(e) Caltech101

Pre-Project Part 1 Datasets

Table 1: Datasets for Transfer Learning in PyTorch

Dataset	Description	# Categories	# Train / # Test
Animals	iNat2017 challenge [3]	9	1384 / 466
Faces	Labeled faces in the wild dataset [4]	31	1021 / 439
Places	Places dataset [5]	9	1437 / 367
Household	iMaterialist 2018 challenge [6]	9	1383 / 375
Caltech101	CVPR 2004 Workshop [7]	30	1500 / 450

Dataset Folder Structure

Loading data in PyTorch

Use a **DataLoader** to load files:

More on the preprocessor later

```
loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(data_dir, preprocessor),
    batch_size=batch_size,
    shuffle=True)
```

Can just loop to get batch data:

```
for i, (in_data, target) in enumerate(loader):
```

Models

Model	Year	Input Size	Last layer input size	PyTorch model
AlexNet [8]	2012	224×224	4096	`torchvision.models.alexnet`
VGG16 [9]	2014	224×224	4096	`torchvision.models.vgg16`
ResNet18 [10]	2016	224×224	512	`torchvision.models.resnet18`
Inception v3 [11]	2015	299×299	2048	`torchvision.models.inception_v3`
DenseNet121 [7]	2017	224×224	1024	`torchvision.models.densenet121`

Using a pre-trained model in PyTorch

```
import torchvision.models
model = torchvision.models.densenet121(pretrained=True)
```


Important!

Preprocessing

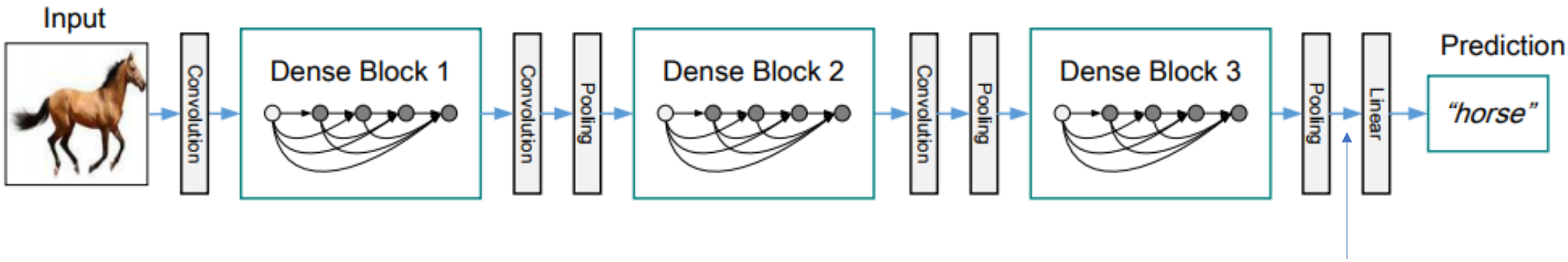
```
import torchvision.transforms as transforms
```

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])
```

```
resize = transforms.Resize((224, 224))
```

```
preprocessor = transforms.Compose([
    resize,
    transforms.ToTensor(),
    normalize,
])
```

Example: DenseNet



Huang, Gao, et al. "Densely connected convolutional networks." Proceedings of the IEEE conference on computer vision and pattern recognition. Vol. 1. No. 2. 2017.

Example: DenseNet as Feature extractor

- Need to stop the network from computing the last linear layer
- Look at forward pass for DenseNet (https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py)

```
def forward(self, x):
    features = self.features(x)
    out = F.relu(features, inplace=True)
    out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.size(0), -1)
    out = self.classifier(out)
    return out
```

This is the last fully connected layer

Example: DenseNet as Feature extractor

- Need to stop the network from computing the last linear layer
- Look at forward pass for DenseNet (https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py)

```
def forward(self, x):
    features = self.features(x)
    out = F.relu(features, inplace=True)
    out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.size(0), -1)

    return out
```

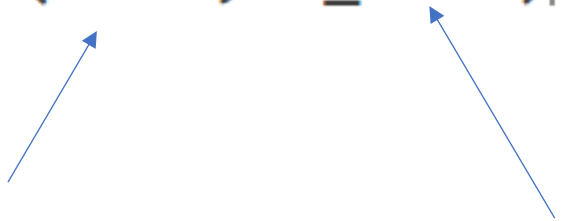
Now out is the second to last layer

Example: DenseNet as Feature extractor

1. Compute features from DenseNet for all training images
   - Easiest to use a DataLoader to load the images
   - Remember to preprocess images
2. Train an SVM with the extracted features
   - Go back to Lab 2 to remember how to use Scikit-Learn's SVM
3. Compute features from DenseNet for all testing images
   - Remember to preprocess images
4. Test extracted testing data features using trained SVM

Example: Finetuning DenseNet

```
model = torchvision.models.densenet121(pretrained=True)
model.classifier = nn.Linear(1024, n_cats)
```



Number of inputs to last layer

Number of classes

- After this can train just like we did in the CIFAR example

Warning: Not all PyTorch models work the same way

Ex: AlexNet:

```
def __init__(self, num_classes=1000):
    super(AlexNet, self).__init__()
    self.features = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(64, 192, kernel_size=5, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(192, 384, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(384, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )
    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(256 * 6 * 6, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, num_classes),
    )
```

```
def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), 256 * 6 * 6)
    x = self.classifier(x)
    return x
```

self.classifer is actually a collection of multiple layers

Data Augmentation

- Can plug in data augmentations into the preprocessing module

```
preprocessor = transforms.Compose([
    resize,
    transforms.ToTensor(),
    normalize,
])
```

Preprocessing goes here

- See torchvisions transform library
  - http://pytorch.org/docs/master/torchvision/transforms.html
  - ColorJitter, RandomAffine, RandomCrop, RandomHorizontalFlip, etc.

Pre-Project Part 1 Assignments

- Work in groups of two
- Each group considers 1 dataset and 1 model
- Need to do both the feature extraction method, and the finetuning method
- Need to also do finetuning with data augmentation

Group number	Dataset	Models
1	Animals	AlexNet
2	Animals	Inception
3	Animals	ResNet18
4	Animals	VGG16
5	Places	AlexNet
6	Places	Inception
7	Places	ResNet18
8	Places	VGG16
9	Faces	AlexNet
10	Faces	Inception
11	Faces	ResNet18
12	Faces	VGG16
13	Household	AlexNet
14	Household	Inception
15	Household	ResNet18
16	Household	VGG16
17	Caltech101	AlexNet
18	Caltech101	Inception
19	Caltech101	ResNet18
20	Caltech101	VGG16

Pre-Project Part 2: Intro to Tensorflow + RNN and Natural Language Processing

EEE 598 Spring 2019

Pre-Project Part 2

- Learn Tensorflow
- Use recurrent neural networks (RNN, LSTM, GRU)
- Natural Language Processing

Deep Learning Libraries

- Caffe – Facebook/Berkeley
- Theano – U of Montreal
- PyTorch - Facebook
- Tensorflow - Google
- CNTK - Microsoft
- MxNet - Apache
- … many more

Why use Tensorflow?

- One of the most popular and most common used Deep Learning libraries.

- Static computation graph: the computations describing a neural network must be “compiled” into a static graph.

- High level API extensions such as Keras, tf.slim, and tf.estimator. But we will use the low-level API in this pre-project.

CIFAR100 in Tensorflow

```
import tensorflow as tf
```

```
def cnn_model_fn(x):
    """
    Define 3-layer cnn from lab 4
    """
    # define network
    # Convolutional Layer #1
    conv1 = tf.layers.conv2d(
        inputs=x,
        filters=16,
        kernel_size=[3, 3],
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #1
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
    strides=2)
```

CIFAR100 in Tensorflow

```
# Convolutional Layer #2 and Pooling Layer #2
conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=32,
      kernel_size=[3, 3],
      padding="same",
      activation=tf.nn.relu)

pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
strides=2)

# Logits Layer
pool2_flat = tf.reshape(pool2, [-1, 8 * 8 * 32])
dense = tf.layers.dense(inputs=pool2_flat, units=4, activation=tf.
nn.relu)

return dense
```

CIFAR100 in Tensorflow – Data Load

```
# Load dataset
train_data, test_data = cifar100(1234)
train_x, train_y = train_data
test_x, test_y = train_data
```

```
# placeholder for input variables
x_placeholder = tf.placeholder(tf.float32,
                               shape=(BATCH_SIZE,) + train_x.shape
[1:])
y_placeholder = tf.placeholder(tf.int32, shape=(BATCH_SIZE))
```

CIFAR100 in Tensorflow

```
# get the loss function and the prediction function for the
network
pred_op = cnn_model_fn(x_placeholder)
loss_op = tf.losses.sparse_softmax_cross_entropy(labels=
y_placeholder, logits=pred_op)

# define optimizer
optimizer = tf.train.GradientDescentOptimizer(LR)
train_op = optimizer.minimize(loss_op)
```

```
# start tensorflow session
sess = tf.Session()

# initialization
init = tf.global_variables_initializer()
sess.run(init)
```

CIFAR100 in Tensorflow - Training

```
# train loop ------------------------------------------------------------
for epoch in range(NUM_EPOCHS):
    running_loss = 0.0
    n_batch = 0
    for i in range(0, train_x.shape[0]-BATCH_SIZE, BATCH_SIZE):
        # get batch data
        x_batch = train_x[i:i+BATCH_SIZE]
        y_batch = train_y[i:i+BATCH_SIZE]

        # run step of gradient descent
        feed_dict = {
            x_placeholder: x_batch,
            y_placeholder: y_batch,
        }
        _, loss_value = sess.run([train_op, loss_op],
                                 feed_dict=feed_dict)

        running_loss += loss_value
        n_batch += 1

    print('[Epoch: %d] loss: %.3f' %
          (epoch + 1, running_loss / (n_batch)))
```

CIFAR100 in Tensorflow - Testing

```
# test loop ------------------------------------------------------------
all_predictions = np.zeros((0, 1))
for i in range(0, test_x.shape[0], BATCH_SIZE):
    x_batch = test_x[i:i+BATCH_SIZE]

    # pad small batch
    padded = BATCH_SIZE - x_batch.shape[0]
    if padded > 0:
        x_batch = np.pad(x_batch,
                         ((0, padded), (0, 0), (0, 0), (0, 0)),
                         'constant')

    # run step
    feed_dict = {x_placeholder: x_batch}
    batch_pred = sess.run(pred_op,
                          feed_dict=feed_dict)

    # recover if padding
    if padded > 0:
        batch_pred = batch_pred[0:-padded]

    # get argmax to get class prediction
    batch_pred = np.argmax(batch_pred, axis=1)

    all_predictions = np.append(all_predictions, batch_pred)
```

Natural Language Processing

- Program computers to process and analyze large amounts of natural language.
- Rule-based
- Statistical NLP

Pre-Project Part 2 Datasets

Table 3: Datasets for RNN in Tensorflow

Dataset	Description	# Categories	Vocab size	# Train / # Test
Sentiment	Movie review dataset [15]	2	18298	7996 / 2666
Spam	Enron spam dataset [16]	2	28049	3879 / 1293
Questions	Learning question classifiers dataset [17]	50	7694	4464 / 1488
Newsgroups	20 newsgroups dataset [18]	20	160792	14121 / 4707

Loading Data and Embedding in Tensorflow

- Each dataset has two CSV files: *train.csv* and *test.csv*.

- We provide a *load_text_dataset()* function in dataset.py to load the datasets as numpy arrays.

- Perform a word embedding on the input using *tf.nn.embedding_lookup* layer. Notice that it has a initizlization arguments called *embedding matrix* which need to be randomly initialized on your own.

Network Design

Table 4: Layers for text processing

Layer	Description	Initialization Arguments
`tf.nn.embedding_lookup`	Embed word vectors	(embedding matrix, input tensor)
`rnn.RNNCell`	Vanilla RNN unit	(output feature size)
`rnn.BasicLSTMCell`	LSTM unit	(output feature size)
`rnn.GRUCell`	GRU unit	(output feature size)
`layers.dense`	Fully connected output	(input tensor, output feature size)

Pre-Project Part 2 Assignments

- Use Tensorflow to train RNN, LSTM, or GRU for sentiment classification

- IMDB dataset
  - Contains movie reviews categorized into positive sentiment and negative sentiment
  - Given a movie review, classify it as positive or negative

Pre-Project Part 2 Assignments

- Work in groups of two
- Each group considers 2 datasets and 1 recurrent unit type
- Use the assigned recurrent unit and the dataset, determine the hyper parameter of the network to achieve good performance.

Project Assignments

- Work in groups of two
- Each group considers 2 datasets and 1 recurrent unit
- Need to do rnn on both dataset

Group number	Datasets	RNN unit
1	Sentiment, Spam	LSTM
2	Sentiment, Questions	LSTM
3	Sentiment, Newsgroups	LSTM
4	Spam, Questions	LSTM
5	Spam, Newsgroups	LSTM
6	Questions, Newsgroups	LSTM
7	Sentiment, Spam	GRU
8	Sentiment, Questions	GRU
9	Sentiment, Newsgroups	GRU
10	Spam, Questions	GRU
11	Spam, Newsgroups	GRU
12	Questions, Newsgroups	GRU
13	Sentiment, Spam	LSTM
14	Sentiment, Questions	LSTM
15	Sentiment, Newsgroups	LSTM
16	Spam, Questions	LSTM
17	Spam, Newsgroups	LSTM
18	Questions, Newsgroups	LSTM
19	Sentiment, Spam	GRU
20	Sentiment, Questions	GRU

IMDB Dataset

this movie has a great message,a impressive cast, ellen burstyn, samantha mathis, jodelle ferland(was 4 years old when she made this movie) ellen burstyn and jodelle ferland have both been nominated for best actress in a tv drama at the up-coming emmy awards in new york, peter masterson-director- has been nominated best director tv drama at the emmy awards also. april 1, 2001, jodelle ferland 'Won', best actress in a tv drama, at the young artist awards, in studio city, ca. i can see why they have 3 nominations. mermaid is a true story, during the cridits they have the real family on the set, something you don't see often. you can find mermaid at all blockbuster video stores. do watch it,you'll be glad you did.

Positive Review

Bette Davis brings her full trunk of tics to this miserable flop which is another variation on the "hilariously mismatched" lovers theme. Sadly, Cagney and Davis are truly mismatched in acting styles and the mix is not simply unpalatable but distasteful. The only distinction in the film comes from Eugene Pallette who, literally, phones in his usual part as the deb's misunderstood dad. Jack Carson's performance can only be described as an act of mayhem on the audience

Negative Review

Appendix: Word Embedding in PyTorch

- We want to describe each word as a fixed size numerical vector
- PyTorch has the module nn.embedding for this
- Tutorial:
  - http://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html

Appendic: PyTorch modules for RNN, LSTM, GRU

class **torch.nn.LSTM**(**args, **kwargs*) [source]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \text{sigmoid}(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \text{sigmoid}(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hc}h_{(t-1)} + b_{hg}) \\
o_t &= \text{sigmoid}(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer, and i_t, f_t, g_t, o_t are the input, forget, cell, and out gates, respectively.

class **torch.nn.GRU**(**args, **kwargs*) [source]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \text{sigmoid}(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \text{sigmoid}(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer, and r_t, z_t, n_t are the reset, input, and new gates, respectively.

class **torch.nn.RNN**(**args, **kwargs*) [source]

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
h_t = \tanh(w_{ih} * x_t + b_{ih} + w_{hh} * h_{(t-1)} + b_{hh})
$$

where h_t is the hidden state at time *t*, and x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer. If nonlinearity='relu', then *ReLU* is used instead of *tanh*.