DEEP LEARNING FRAMEWORKS WORKSHOP

SAIG – Winter 2018

NEURAL NETS REVIEW

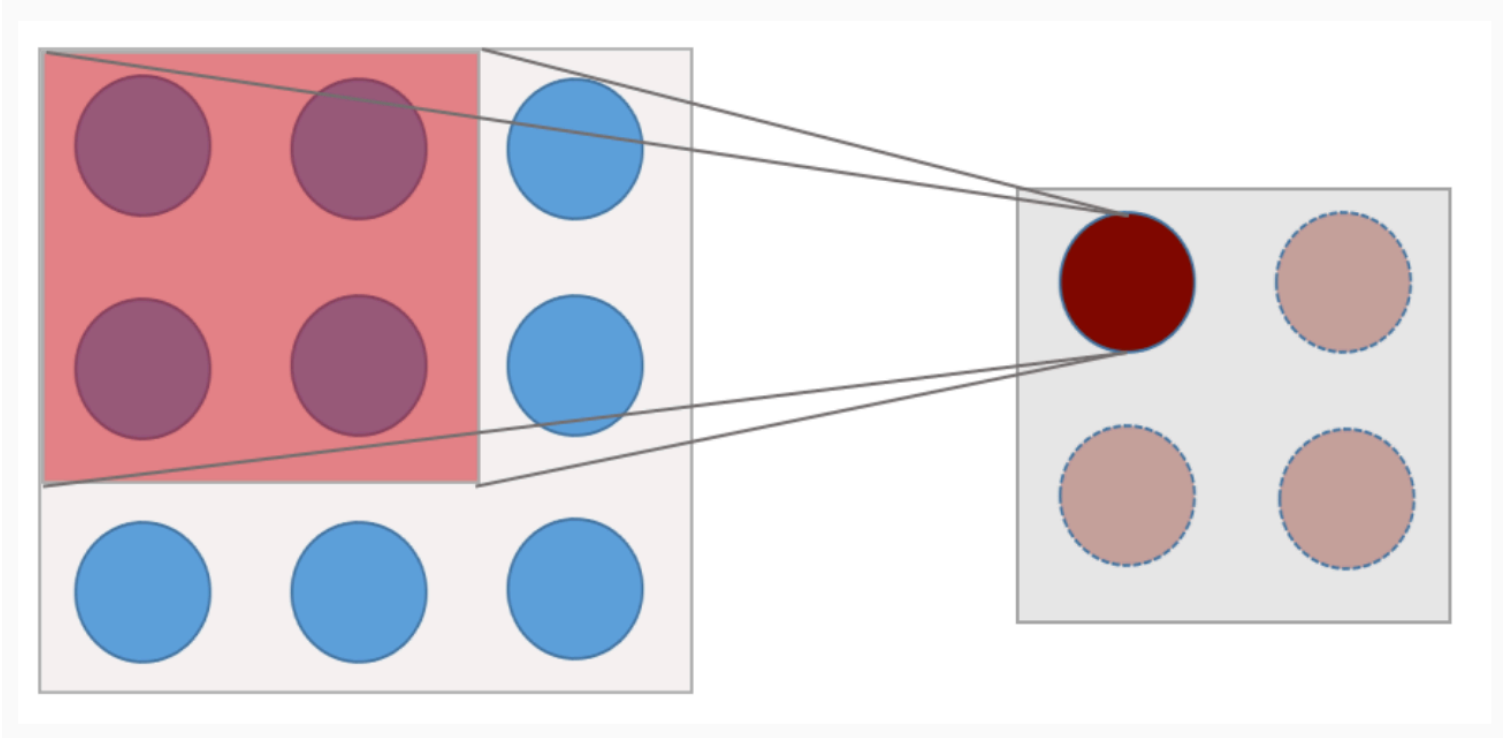
- Set of matrix multiplications to transform and input into an output
- We are given input data and target
- Transform input to our guess at target, and compare our guess to the answer
  - We compute a loss value—how wrong our guess was
  - Want to minimize this loss value by doing gradient descent on loss function

CONVNET BASICS

CONVOLUTIONAL LAYER

- Computes dot product over input map to create output map
  - Map is 3D tensor with spatial dimensions
  - Input map is image, output map is representation
- Layer has one shared weight called a **filter**
- Filter is "slid" over the input
  - Amount it is slide is called **stride**
  - Size of filter is called **kernel size** or **spatial extent**
  - We can surround the border with zeros, called **padding**

CONV LAYER VISUALIZED



MAX POOLING LAYER

- Same idea of "sliding" as a convolutional layer
- The difference: no dot product, just tax the max!

RECTIFIED LINEAR UNIT LAYER

- **In general**, activation functions take a single number and performs a mathematical operation upon it. They provide nonlinear properties to the network
- Computes the activation function $f(x) = \max(0, x)$
- Found to accelerate the convergence of SGD
- Easier to implement than other activation functions (sigmoid, tanh)
- Avoids saturation problems with sigmoid and tanh neurons, but this comes at a cost – if x is ever less than 0, the neuron "dies"

SOFTMAX LAYER

- Defines an output layer for neural networks
- First, forms weighted outputs defined as

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$$

and applies the softmax function to these outputs defined as

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

we can prove that this function squashes all outputs so that the sum of the outputs is equal to 1. **So, the output of softmax can be thought of as a probability distribution.**

TENSORFLOW

CREATE MODEL

```
def cnn_model_fn(features, labels, mode):
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[3, 3],
      padding="same",
      activation=tf.nn.relu)
  conv2 = tf.layers.conv2d(
      inputs=conv1,
      filters=64,
      kernel_size=[3, 3],
      padding="same",
      activation=tf.nn.relu)
  pool = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
  dropout1 = tf.layers.dropout(
      inputs=pool, rate=0.25, training=mode == tf.estimator.ModeKeys.TRAIN)
  flat = tf.reshape(dropout1, [-1, 14 * 14 * 64])
  dense = tf.layers.dense(inputs=flat, units=128, activation=tf.nn.relu)
  dropout2 = tf.layers.dropout(
      inputs=dense, rate=0.25, training=mode == tf.estimator.ModeKeys.TRAIN)
  logits = tf.layers.dense(inputs=dropout2, units=10)
```

SETTING UP LOSS FUNCTION AND LOSS

```
predictions = {
    # Generate predictions (for PREDICT and EVAL mode)
    "classes": tf.argmax(input=logits, axis=1),
    # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
    # `logging_hook`.
    "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
  return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
  train_op = optimizer.minimize(
      loss=loss,
      global_step=tf.train.get_global_step())
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

LOADING AND PREPARING DATA

```
def main(unused_argv):
  # Load training and eval data
  mnist = tf.contrib.learn.datasets.load_dataset("mnist")
  train_data = mnist.train.images  # Returns np.array
  train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
  eval_data = mnist.test.images  # Returns np.array
  eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

  # Create the Estimator
  mnist_classifier = tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")

  # Set up logging for predictions
  # Log the values in the "Softmax" tensor with label "probabilities"
     tensors_to_log = {"probabilities": "softmax_tensor"}
     logging_hook = tf.train.LoggingTensorHook(
         tensors=tensors_to_log, every_n_iter=1000)
```

TRAIN AND EVAL MODEL

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
input_fn=train_input_fn,
steps=20000)

# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

KERAS

INITIALIZE

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
```

```
Using Theano backend.
```

```
batch_size = 128
num_classes = 10
epochs = 12
img_rows, img_cols = 28, 28
```

LOAD DATA AND VARIABLES

```
# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

CREATE MODEL

```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

TRAIN AND EVAL MODEL

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

PYTORCH

CREATE MODEL

```
class Flatten(nn.Module):
    def __init__(self):
        super(Flatten, self).__init__()

    def forward(self, X):
        X = X.view(X.size(0), -1)
        return X

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.model = nn.Sequential(nn.Conv2d(1, 32, kernel_size=3),
                                   nn.ReLU(),
                                   nn.Conv2d(32, 64, kernel_size=3),
                                   nn.MaxPool2d(2),
                                   nn.Dropout(0.25),
                                   Flatten(),
                                   nn.Linear(9216, 128),
                                   nn.Dropout(0.5),
                                   nn.Linear(128, 10)
                                  )
    def forward(self, X):
        X = self.model(X)
        return X
```

LOAD DATA AND VARIABLES

```
transform = transforms.Compose([
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))
                                   ])
train_set = dset.MNIST(root='./data', train=True,
                           transform=transform, download=True)
test_set = dset.MNIST(root='./data', train=False, transform=transform)

batch_size = 100
train_loader = torch.utils.data.DataLoader(
                    dataset=train_set,
                    batch_size=batch_size,
                    shuffle=True)
test_loader = torch.utils.data.DataLoader(
                   dataset=test_set,
                   batch_size=batch_size,
                   shuffle=False)

model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-2)
```

DEFINE RUN FUNCTION

```
def run_model(loader, train=False, optimizer=None):
    avg_loss, avg_acc = 0, 0
    for i, (X, y) in enumerate(loader):
        if train:
            optimizer.zero_grad()
        X, y = Variable(X), Variable(y)

        y_hat = model(X)
        loss = criterion(y_hat, y)

        if train:
            loss.backward()
            optimizer.step()

        pred = np.argmax(y_hat.data.numpy(), axis=1)
        acc = float(len(np.where(pred == y.data.numpy())[0]))
        acc /= batch_size

        avg_loss += loss.data[0]
        avg_acc += acc

    avg_loss /= (i+1)
    avg_acc /= (i+1)
    return avg_loss, avg_acc
```

TRAIN AND EVAL MODEL

```
train_loss, train_acc = run_model(train_loader,
                                  train=True,
                                  optimizer=optimizer)
print("Average train loss and accuracy: %f, %f"
                          % (train_loss, train_acc))

test_loss, test_acc = run_model(test_loader)
print("Average test loss and accuracy: %f, %f"
                          % (test_loss, test_acc))
```