Pre-Project: Transfer Learning in PyTorch & NLP with RNN in Tensorflow

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning Media Processing & Understanding

Objectives

This is the pre-project in which students need to study how to use deep learning libraries: PyTorch and Tensorflow. There are two parts for this pre project:

- **Part 1: Transfer Learning in PyTorch**
  - Learn PyTorch Basics
  - Extract features from pre-trained networks and train SVM classifiers using the features
  - Fine-tune the networks to classify on different datasets
- **Part 2: Natural Language Processing with Recurrent Neural Network in Tensorflow**
  - Learn Tensorflow Basics
  - Learn how to use RNNs, LSTMs, and GRUs in Tensorflow
  - Use recurrent modules for sequence modeling tasks

Part 1: Transfer Learning in PyTorch

1 PyTorch

PyTorch [1] is an open source machine learning library that is particularly useful for deep learning. PyTorch contains auto-differentation, meaning that if we write code using PyTorch functions, we can obtain the derivatives without any additional derivation or code. This saves us from having to implement any backwards functions for deep networks. Secondly, PyTorch comes with many functions and classes for common deep learning layers, optimizers, and loss functions. Lastly, PyTorch is able to efficiently run computations on either the CPU or GPU.

1.1 Installation

To install PyTorch run the following commands in the Linux terminal:

```
pip install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-
    cp27-cp27mu-linux_x86_64.whl
pip install torchvision
```

The first command installs the basic PyTorch package, and the second installs `torchvision` which includes functions, classes, and models useful for deep learning for computer vision.

1.2 Tutorial

To start we ask you to complete the following tutorial: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

2 CIFAR100 Example in PyTorch

Next as an example, we will re-implement the neural network from Lab 4 using PyTorch instead of the library we built. The code for this example is in the included `cifar_pytorch.py` file.

PyTorch has a module called `nn` that contains implementations of the most common layers used for neural networks. If you look at the code for some layer (for example `linear.py`), you will see that it is similar to our implementation from lab 3. There is a `forward` method and an `__init__` method. There is no need for a `backward` method because PyTorch uses automatic differentiation.

In this example, we will implement our model as a class with `forward`, `__init__`, `fit` and `predict` functions. The initialization function simply sets up our layers using the layer types in the `nn` package. In the forward pass we pass the data through our layers and return the output. Note that we can reuse the pool layer, since this layer has no learnable parameters. Also instead of a ReLU layer, we can use the `F.relu` function in our forward pass. Similarly, instead of a “flatten” layer, we can just use PyTorch’s `view` to reshape the data.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv2d args: (input depth, output depth, filter size)
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        # Linear args: (# of input units, # of output units)
        self.fc1 = nn.Linear(1152, 5)
        # MaxPool2d args: (kernel size, stride)
        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) # -1 means shape is inferred
        x = self.fc1(x)
        return x
```

Fit function Next we will implement a fit function. Here we implement the fit function as a class method, similar to lab 3. It is also common to see the code for training be implemented outside of the model class in a separate function. The fit function is very similar to our own fit function from lab 3. First we set an optimization criterion, and an optimizer. Next we loop for a number of epochs. In each epoch, we use PyTorch's `DataLoader` class to loop through the data in batches. The DataLoader automatically takes care of splitting the data into batches. We access the batches with a simple `for` loop through the DataLoader. For each batch we zero the previously calculated gradients using the optimizers `zero_grad` method. Then we call the forward function, compute the loss, call the backwards function, and perform one optimization step. The optimization step is similar to the `update_params` methods we used in previous labs.

```
    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
            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)

                # 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)))
```

Finally it is also useful to provide a predict function to run our model on some test data. This predict function will also use the PyTorch loader.

```
    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
```

Notice that we use `torch.nn.module.train()` for training and `torch.nn.module.eval()` for prediction. Function `train()` sets the module in training mode and function `eval()` sets the module in evaluation mode. These two functions affect only on certain modules. They control certain layers like `Dropout` and `BatchNorm` to be enabled during training and disabled during evaluation. For simple neural network structure like in Lab 3 and Lab 4, we do not have any layers that will be affected by `self.train()` or `self.eval()`, so it is okay for you to delete them. But it is a good habit to always adding them when implementing a neural network.

We also use `torch.no_grad()` when doing prediction. This function is introduced since PyTorch version 0.4, it disables the gradient calculation. You can use it when you are sure that you will not call the backward function. This will reduce memory consumption for computations.

To evaluate the two models we look at both the final classification accuracy as well as the *confusion matrix*. The rows of the confusion matrix show the predicted class and the columns show the actual class. This lets us analyze the patterns of missclassifications. The included `util.py` includes the function `plot_confus_matrix` which will plot this matrix. Figure 1 shows the confusion matrix for this CIFAR100 example.

3 Transfer Learning

Transfer learning is when we use a model trained on one set of data, and adapt it to another set of data. For image datasets, transfer learning works because many features (e.g. edges) are useful across different image datasets. Transfer learning using neural networks trained on large image datasets is highly successful, and can easily be competitive with other approaches that do not use deep learning. Transfer learning also does not require a huge amount of data, since the pre-trained initialization is a good starting point.

In this project, we will consider two types of transfer learning: a feature-extraction based method and a fine-tuning based method. We will be using networks that have been pre-trained on the ImageNet dataset [2], and adapt them for different datasets.

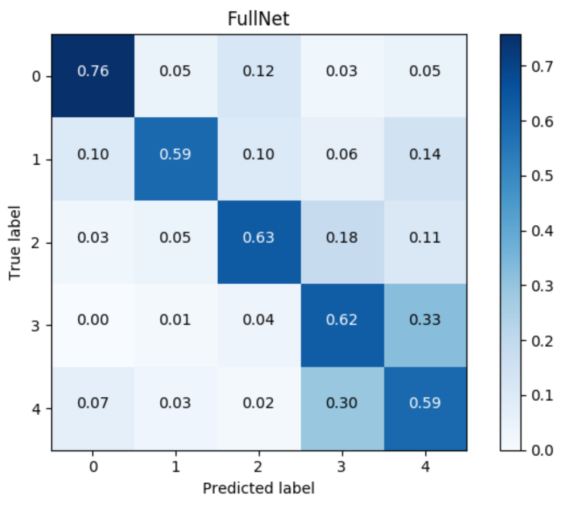


Figure 1: CIFAR100 confusion matrix.

3.1 Datasets

We have collected five datasets for use in this project (Table 1). You will be assigned one of these datasets to work on. The datasets are subsets of existing datasets. Figure 2 shows example images from these datasets. You will not get credit if you download and submit a model trained on the full version of these datasets. We want to fine-tune models that were originally trained on the ImageNet dataset.

With transfer learning we alleviate some of the problems with using small datasets. Typically if we tried to train a network from scratch on a small dataset, we might experience overfitting problems. But in transfer learning, we start with some network trained on a much larger dataset. Because of this, the features from the pre-trained network are not likely to overfit our data, yet still likely to be useful for classification.

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

3.2 Base Networks

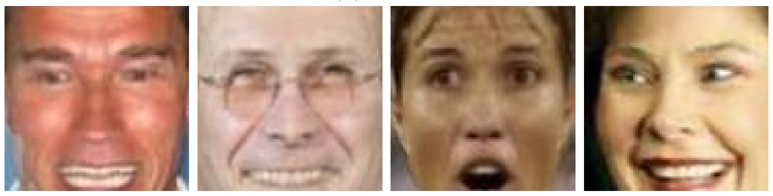
Table 2 shows the base networks that we will consider in this project. Each group will consider one of the networks (see Section 4.2 for group assignments). All of these networks have PyTorch versions trained on the ImageNet dataset [2], but the architectures of the networks vary. The input size and the last layer input size also vary among the networks.

As an example we will be using DenseNet [8] to explain how to do fine-tuning in PyTorch. DenseNet is not assigned to any teams in the project, but the principles for using DenseNet are the

(a) Household

(b) Animals

(c) Places



(d) Faces

(e) Caltech101

Figure 2: Example images from datasets for transfer learning in PyTorch.

same as for using other models. In PyTorch we can load a pre-trained DenseNet model with the command:

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

It is important to use the `pretrained=True` argument. Otherwise, the model will be initialized with random weights.

Table 2: Base Networks for Transfer Learning in PyTorch

Model	Year	Input Size	Last layer input size	PyTorch model
AlexNet [9]	2012	224×224	4096	`torchvision.models.alexnet`
VGG16 [10]	2014	224×224	4096	`torchvision.models.vgg16`
ResNet18 [11]	2016	224×224	512	`torchvision.models.resnet18`
Inception v3 [12]	2015	299×299	2048	`torchvision.models.inception_v3`
DenseNet121 [8]	2017	224×224	1024	`torchvision.models.densenet121`

3.3 Pre-processing

It is important that we pre-process the images before sending them to the network. Most DNNs preprocess the images to be zero mean and unit standard deviation before training. During testing if we pass an image that is also not normalized (with respect to the training data), then the network output won't be useful. We can say that the network "expects" the input to be normalized. PyTorch includes a `transform` module that implements the common transformations, including normalization, used in pre-processing:

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

For normalization we can utilize the built in PyTorch function `Normalize`. The values used for normalization can be computed from the images in the ImageNet dataset. For each channel in the image there is a separate mean and standard deviation used for normalization.

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

To apply our data with our pretrained neural network, we must resize the images to the input size expected by the network. For Alexnet, VGG16, ResNet18, and DenseNet this is 224×224 pixels, but for Inception this is 299×299 pixels. There are different ways to ensure our input is the correct size. The simplest way is to resize our input to the correct size. This has the disadvantage of potentially stretching the image if there is an aspect ratio mismatch. An alternative method is to resize the image so that the minimum side length is equal to the required size, and then crop out the extra part of the image to get the desired size. For this tutorial, we use the first method with PyTorch's `Resize` method:

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

In preprocessing we would like to apply these transformations in a pipeline to every image. PyTorch includes a useful function called `Compose` to combine the transformations into a single object representing the pipeline:

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

Note the presence of `ToTensor()` which is needed to convert from the image representation to the PyTorch tensor representation. We can easily add other transformations to this preprocessor object to do more preprocessing, or to add data augmentation.

3.4 Part 1: Feature Extractor + SVM

First we will use the base network as a feature extractor. This means that we simply run the images through the pre-trained base network, and take outputs from layer(s) as a feature representation of the image. These features are usually good for classification with a shallow machine learning algorithm. In this case we will use an SVM for classification.

To extract features we need to stop the forward pass after a certain layer. As of writing this tutorial, PyTorch doesn't offer a simple call to extract a certain layer. However it is very easy to change the model to give the output of a certain layer. How exactly to extract a certain layer depends on the implementation of the model. We recommend you look at the source code for the networks available at https://github.com/pytorch/vision/tree/master/torchvision/models.

The PyTorch AlexNet and VGG models are split into a feature extractor stage, and a classifier stage. The feature extractor consists of the convolutional layers, and the classifier consists of the fully connected layers. As an example, we want to extract the output of the layer before the last fully connected layer. The easiest way to do this is to modify the sequential part of the model to remove the second to last layer.

```
new_classifier = nn.Sequential(*list(model.classifier.children())
   [:-1])
model.classifier = new_classifier
```

If we are using a model that does not use the feature extractor/classifier decomposition, we need to modify the forward pass of the model to only compute up to the requested layer. For example, for DenseNet, we can comment out the last layer to extract the features before the final fully connected layer.

```
def forward(self, x):
    features = self.features(x)
    out = F.relu(features, inplace=True)
    # average pool features and reshape to (batch size, feature size)
    out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.
   size(0), -1)
    # out = self.classifier(out) # commented out to get the features
   instead
    return out
```

Next we need some way to load the data from our dataset. Again PyTorch provides some convenient tools to do this. We will use the `datasets.ImageFolder` class to load our dataset. The ImageFolder loader assumes a file structure of our data where each of the classes is stored in a separate folder. Next we will use the `torch.utils.data.DataLoader` class to create a loader that can be used to loop through our data with some batch size. We would need to have separate loaders for the training data and the testing data. A loader can be constructed with:

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

With the loader set, we can now loop through the data and extract features. During looping we can use Python's `enumerate` function to keep track of the batch index. The loader returns a tuple with the data and the target label.

In the case of testing, we don't need to feed the label to the network, but it would be useful to save the label so that we can use it later for computing the SVM classification accuracy for the test set. To use the input data in our PyTorch model, we need to wrap it as a PyTorch `Variable`.

```
for i, (in_data, target) in enumerate(loader):
    input_var = torch.autograd.Variable(in_data, volatile=True)
    output = model(input_var)
```

With the extracted features for each sample, we use Sci-kit learn's SVM model. To review how to use the model, revisit Lab 2. It is up to you to determine the type of SVM and best hyper parameters.

We can assess the accuracy of the SVM model using the classification accuracy on the entire testing set. In addition to this, we can plot a confusion matrix, which tells more information about the errors that the model made. `util.py` includes the function `plot_confus_matrix` which will plot this matrix.

Important:

The `plot_confus_matrix` function plots a confusion matrix. Note that there is an input parameter `size` to indicate how many classes you want to show in the plot. The default value for `size` is `None` in which case it will plot the whole confusion matrix. If `size` is given some integer value, for example 9, it will plot the confusion matrix from class 0 to class 8. To make the result more clear to observe, for datasets, such as Faces and Caltech101, please plot confusion matrix for the first 9 classes. For the other three datasets, use the default setting to plot the whole confusion matrix.

3.5 Part 2: Fine-tuning

The feature extractor + SVM approach already may give decent results for your problem. This is because, for certain problems, the intermediate representations learned by the pre-trained network can be very useful for the new problem. However, even if the pre-trained filters are giving good performance, we may be able to achieve even greater performance by allowing the parameters of the pre-trained model to adapt to our new dataset. This adaptation process is called fine-tuning.

Performing fine-tuning is exactly the same process as performing training. Refer to the included `cifar_pytorch.py` to see how to train in PyTorch. The main differences in this case is that we want to start from the pre-trained model. We can use the same `DataLoader` and `transform` modules that we used for feature extraction. During fine-tuning we will usually use a very small learning rate. We want to adapt the existing filters to our data, but not move the parameters so far from the pre-trained parameters.

During fine-tuning we can speed up the process by running the model on the GPU. In PyTorch this can be accomplished by using the `.cuda()` command on the model and loss function. For example:

```
model = model.cuda()
criterion = criterion.cuda()
```

Finally, after training, we would like to save the model so that we can use it in the future without training again. We can use the `model.save(filename)` function to save the model.

For this project, we do not specify the hyper parameters for you to use. It is up to you to choose a good set of hyper parameters. Examples of hyper parameters that can be changed are learning rate, batch size, and number of epochs.

Data Augmentation To achieve higher performance, we can experiment with data augmentations. Data augmentation is the process of slightly perturbing the input images to generate more samples than were originally available. This data augmentation could be image rotation, scale, gray scale transformation, etc.

PyTorch includes transformations useful for data augmentation in the `torchvision.transforms` module. As part of the project, you need to add some of these data augmentation methods to your preprocessing object to achieve greater test accuracy.

Deliverable: Pre-Project Part 1

Depending on your group number, you are assigned a different dataset and model to perform transfer learning (Table 5).

- Provide the test accuracy and confusion matrices for the considered networks as feature extractors for the dataset; name the confusion matrix plot as `conf_feature.png`
- Provide the test accuracy and confusion matrices for the considered fine-tuned network for the dataset; name the confusion matrix plot as `conf_finetune.png`
- Submit the trained model saved using the `model.save` function
- Submit code for both the feature extraction based method and the fine-tuning based method named as `feature.py` and `finetune.py`
- Submit the code for the fine-tuning based method with data argumentation, as well as the saved trained model

Part 2: Natural Language Processing with Recurrent Neural Network in Tensorflow

1 Tensorflow

Tensorflow is a popular library for machine learning developed by Google. Tensorflow includes all of things that might be useful for deep learning including: neural network layers, optimizers, loss functions, and auto differentiation. Different from PyTorch, Tensorflow primarily uses a static computation graph. This means that the computations describing a neural network must be "compiled" into a static graph. PyTorch uses a dynamic computation graph, where operations can be added without need for recompiling. The advantage of the static graph is better performance in some cases, however the disadvantage is that it is more difficult to debug. To alleviate the difficulty in debugging, Tensorflow added eager mode, which uses a dynamic computation graph. However, at the time of writing, eager mode does not support parts of Tensorflow, so we will be using the default static computation graph.

Tensorflow can be installed with a `pip` command:

```
sudo pip install tensorflow
```

Tensorflow has many different high level API extensions such as Keras, tf.slim, and tf.estimator. These extensions abstract some of the underlying Tensorflow code away. While these libraries are useful for quickly prototyping code, knowledge of the underlying Tensorflow processes is also important. For your project write your code using the low-level Tensorflow API, instead of using the higher level packages.

Please follow the following tutorials to become familiar with Tensorflow:

- Low level intro - `https://www.tensorflow.org/programmers_guide/low_level_intro`
- Tensors - `https://www.tensorflow.org/programmers_guide/tensors`
- Variables - `https://www.tensorflow.org/programmers_guide/variables`

2 CIFAR100 in Tensorflow

We will re-implement the convolutional network from Lab 4 in Tensorflow for demonstration purposes. We provided the code for this example in `cifar_tensorflow.py`. First we will write a function describing the forward pass of the network. We don't need to write any code for the backwards pass, since the gradient computations are computed automatically from the computation graph. To implement the network, we will utilize the layers provided in the `tf.layers` package. Specifically, we will use `tf.layers.conv2d` to implement a convolutional layer, `tf.layers.max_pooling2d` for max pooling, and `tf.layers.dense` for fully connected layers. The functions `tf.nn.relu` and `tf.reshape` are used for ReLU and flattening, respectively.

```
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)

    # 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
```

Next we will load the dataset. We have provided a function `cifar100` in the `dataset.py` file that loads the dataset into Numpy arrays. Like Lab 3 and Lab 4, this function takes a seed and generates a random subset of the dataset with 4 classes.

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

In Tensorflow instead of writing functions that directly performs some calculations, we actually are writing functions that define part of a computation graph, which can be run later. To define inputs in the computation graph, we use `tf.placeholder`. As the name suggests, this creates a "placeholder" variable which could correspond to some input values. Later when we call the graph, we can pass real data to these placeholders.

```
# 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))
```

Next we define a few operations (ops) that we will use for training and testing. An operation is some output or function that we are interested in computing from the computation graph. In this case, we want an operation to get the output of the network, one to get the loss value, and an operation to perform the gradient descent update. The gradient descent operation uses the `tf.train.GradientDescentOptimizer` class.

```
# 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)
```

Next we can start a Tensorflow *session*. We need to define a session in order to run the previously defined operations. An operation can be run in a session by using `sess.run(op)`, where `sess` is the Tensorflow session, and `op` is the operation we want to run. As an example, we use the session to run an operation that initializes the variables in our network using `tf.global_variables_initializer`.

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

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

Finally, we can begin training. There are several ways to perform training in Tensorflow. Here we will manually loop through our data and call the Tensorflow operations for each batch. This approach should be familiar from Lab 3 and Lab 4. To pass data into the computation graph, we use the `feed_dict` argument of the `sess.run` command. Here we pass Numpy matrices into the place holder values of the graph.

```
# 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)))
```

We can perform testing in a similar way as training. One difficulty is that the static graph requires a constant batch size. Our dataset might not be able to be evenly split into batches, depending on the batch size. There are a few possible ways to deal with this. One way (shown below) is to pad the last batch such that it matches the batch size. After padding, we should make sure that we don't take any of the padded outputs, since these won't have ground truth labels.

```
# 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)
```

3 RNN Background

Convolutional neural networks work well for certain types of data that can be made into fixed size inputs. But what if the data cannot be assumed to be of fixed size? This is common in problems that consider some sort of time series input like speech or text.

If the input data cannot be assumed to be a fixed size, we need some way to have our network automatically adapt to the different sized inputs. One way to do this is to use *recurrent neurons*, where a neuron's output can be used as an input to the same neuron at a different time step (Figure 3). With this formulation we can "unroll" the network for as many time steps as needed to process the data (Figure 4). The unrolled representation can be used to perform backpropagation.

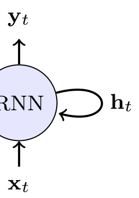


Figure 3: Basic RNN Unit.

There are several "flavors" of recurrent units. In this project we will consider three common variants: vanilla recurrent neurons (RNN), long-short term memory units (LSTM) [13], and gated

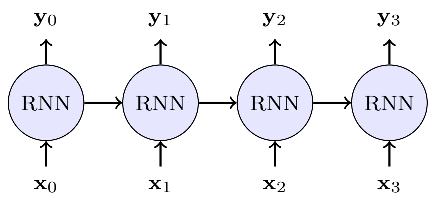


Figure 4: Unrolled Recurrent Network.

recurrent units (GRU) [14]. We will use these recurrent units to create a network that can classify text.

3.1 RNN

The equation for the hidden state of a vanilla RNN layer takes the following form:

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{b}_h\right) \tag{1}$$

where $\mathbf{x}_t$ is an input at time t, and $\mathbf{h}_t$ is the corresponding state vector. The $\mathbf{W}$ matrices and $\mathbf{b}$ vectors represent learnable parameters.

RNN layers can be used in Tensorflow with the `tf.contrib.rnn.BasicRNNCell` layer. The main argument in the initialization of this layer is the output feature dimension (`num_units`). You must set this parameter appropriately to achieve good performance.

To use the RNN layer, we wrap it in a `tf.contrib.rnn.static_rnn` function. This function takes the RNN cell and a list of tensors as input. Each tensor of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all time steps, and the final value of the hidden state.

The output of the RNN is based on the value of the hidden state. The form of the output is identical to a fully connected layer:

$$\mathbf{y}_t = \text{softmax}\left(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y\right) \tag{2}$$

where $\mathbf{y}_t$ is an output at time t. For this project we are only interested in the output at the last time step. The `BasicRNNCell` layer itself does not compute this output. To implement the output we need to use a separate fully connected layer in Tensorflow (`tf.layers.dense`).

3.2 LSTM

The long-short term memory unit (LSTM) is a more complex recurrent node that incorporates different "gates" to allow or reject information from passing through the network. There is an input gate, an output gate, and a forget gate that controls the flow of information. Additionally, the LSTM has both a memory cell and a hidden state. The LSTM is computed as:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f x_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_c x_t + \mathbf{U}_c h_{t-1} + \mathbf{b}_c) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t)
\end{aligned}$$

where $\circ$ is the Hadamard product (e.g., element-wise product), σ is a sigmoid function, $\mathbf{f}$ is the forget gate, $\mathbf{i}$ is the input gate, $\mathbf{o}$ is the output gate, $\mathbf{c}$ is the memory cell, and $\mathbf{h}$ is the hidden state. $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{b}$ represent learnable parameters.

LSTM layers can be used in Tensorflow with the `rnn.BasicLSTMCell`. As with the RNN, the main arguments in the initialization of this layer is the output feature dimension. The output feature dimension is a parameter that you must set yourself to achieve good performance. To use the LSTM, we again use the `tf.contrib.rnn.static_rnn` function. This function takes the LSTM cell and a list of tensors as input. Each element of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all positions, and the final value of the hidden state.

The Tensorflow LSTM implementation has no output $\mathbf{y}$. We can create an output using the hidden state at a certain time using equation 2. This corresponds to a fully connected layer with the hidden state as the input.

3.3 GRU

The Gated Recurrent unit GRU can be thought of as a simplified version of LSTM. In practice the performance of the GRU and the LSTM can often be similar. The GRU incorporates only two gates: an update gate and a reset gate. It does not use a memory cell like the LSTM. The GRU is computed by:

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{b}_z) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + \mathbf{b}_r) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \circ \mathbf{h}_{t-1} + \mathbf{z}_t \circ \tanh(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \circ \mathbf{h}_{t-1}) + \mathbf{b}_h)
\end{aligned}$$

where $\circ$ is the Hadamard product, σ is a sigmoid function, $\mathbf{z}$ is the update gate, $\mathbf{r}$ is the reset gate, and $\mathbf{h}$ is the hidden state. $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{b}$ represent learnable parameters.

GRU layers can be used in Tensorflow with the `rnn.GRUCell`. As with the RNN, the main arguments in the initialization of this layer is the output feature dimension. The output feature dimension is a parameter that you must set yourself to achieve good performance. To use the GRU, we again use the `tf.contrib.rnn.static_rnn` function. This function takes the LSTM cell and a list of tensors as input. Each element of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all positions, and the final value of the hidden state.

The Tensorflow GRU implementation has no output $\mathbf{y}$. We can create an output using the hidden state at a certain time using equation 2. This corresponds to a fully connected layer with the hidden state as the input.

4 RNNs for sentence classification

In this project we will utilize recurrent layers to classify sentences.

4.1 Datasets

In this project, we consider 4 datasets: sentiment, spam, questions, newsgroups. Each dataset is a collection of text entries, with a corresponding text label for each entry. Each group will be assigned two datasets to work on (see Section 4.2). Table 3 describes each dataset.

The sentiment dataset consists of movie reviews that are either positive or negative [15]. The spam dataset consists of emails that are either spam or not spam [16].

The questions dataset [17] contains questions classified by what type is the answer to the question. For example the questions “Who is Snoopy’s arch-enemy?” and “What actress has received the most

Oscar nominations?” have the same class. As another example the questions “What novel inspired the movie BladeRunner?” and “What’s the only work by Michelangelo that bears his signature?” also have the same class. The classes are varied enough that the network has to understand the whole sentence to classify, rather than just looking for key words like “what” or “who”

The newsgroups dataset [18] consists of newsgroup postings from twenty different categories (Newsgroups are where people posted things on the internet before Facebook and Reddit). Examples of the newsgroup classes are “comp.sys.mac.hardware”, “talk.politics.guns”, “rec.sport.hockey”.

Each dataset is stored in two CSV files: `train.csv` and `test.csv`. Each line of the CSV file has a sentence and a label. We have provided a `load_text_dataset` function in `dataset.py` to load the datasets as numpy arrays. This function takes as arguments the name of the dataset and the maximum sequence size. Any sentences that are smaller than the maximum sequence size will be padded, and any sequences that are larger will be truncated. The dataset loader will translate each unique word to a unique number using a lookup table. The number of unique words is the vocabulary size of the dataset.

Table 3: Datasets for NLP with 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

4.2 Network Design

The input to our network is a batch of sentences and the output is a batch of predicted labels. From the input we first want to perform a word embedding. The embedding layer takes the words, which have been mapped to integers by the dataset loader, and outputs a vector embedding for each word. For example, the word “yes” might be mapped to the vector $[0.1, 0.7, -0.5]$ and the word “no” might be mapped to the vector $[1.1, -0.5, 0.3]$. The dimension of the embedding is a hyper-parameter that must be set. We can use the `tf.nn.embedding_lookup` layer for this. The embedding layer uses an embedding matrix. For our application the embedding matrix can be randomly initialized.

Next we pass the embedded words into the recurrent layer. In general, we could utilize several stacks of recurrent layers, but in this project it is only required to consider a single recurrent layer stack. We want to obtain a class prediction from the output of the recurrent unit, so we can use a fully connected layer after the recurrent layer. The number of outputs of the fully connected layer should be equal to the number of classes.

Since we are doing classification, we can use the `tf.losses.sparse_softmax_cross_entropy` loss function. This loss function incorporates the softmax internally, so we do not need to explicitly add a softmax layer to our model.

Training the sequence models is almost identical to training convolutional neural networks. We still need to loop through our data, and for each batch call the train operation.

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)

Hint: Training RNNs

Training RNNs may be a little more difficult than training convolutional neural networks. If training is not working well, you may need to fiddle with the learning rate, or try another optimizer such as ADAM [19]. ADAM can be used with the Tensorflow `tf.train.AdamOptimizer` class.

Deliverable: Pre-Project Part 2

Depending on your group number, you are assigned two different datasets and one recurrent unit type (Table 5). For the assigned datasets and recurrent unit type, you must determine the hyper parameter (e.g., learning rate, batch size, embedding size, hidden state feature dimension) of the network to achieve good performance.

- Submit code for both datasets named as `dataset_unitname.py`, where `dataset` is replaced by your assigned dataset and `unitname` is replaced by your assigned recurrent unit type (lstm, gru)
- Provide the final classification accuracy for both datasets
- Submit the two trained models

Additional Resources

- **PyTorch:**
  - PyTorch Documentation - `http://pytorch.org/docs/stable/index.html`
  - TorchVision Documentation - `http://pytorch.org/docs/master/torchvision/index.html`
  - PyTorch tutorials - `http://pytorch.org/tutorials/`
  - PyTorch Github - `https://github.com/pytorch/pytorch`
  - TorchVision Github - `https://github.com/pytorch/vision`
  - PyTorch Discussion Forum - `https://discuss.pytorch.org/`
- **TensorFlow:**
  - Tensorflow API Documentation - `https://www.tensorflow.org/api_docs/`
  - Tensorflow Tutorials - `https://www.tensorflow.org/tutorials/`
  - Tensorflow Github - `https://github.com/tensorflow/tensorflow`
  - Tensorflow Discussion Forum - `https://groups.google.com/a/tensorflow.org/forum/#!forum/discuss`

Task Assignment

Table 5: Group tasks for transfer learning in PyTorch and RNN in Tensorflow

Group number	Dataset	Models	Datasets	RNN unit
1	Animals	AlexNet	Sentiment, Spam	LSTM
2	Animals	Inception	Sentiment, Questions	LSTM
3	Animals	ResNet18	Sentiment, Newsgroups	LSTM
4	Animals	VGG16	Spam, Questions	LSTM
5	Places	AlexNet	Spam, Newsgroups	LSTM
6	Places	Inception	Questions, Newsgroups	LSTM
7	Places	ResNet18	Sentiment, Spam	GRU
8	Places	VGG16	Sentiment, Questions	GRU
9	Faces	AlexNet	Sentiment, Newsgroups	GRU
10	Faces	Inception	Spam, Questions	GRU
11	Faces	ResNet18	Spam, Newsgroups	GRU
12	Faces	VGG16	Questions, Newsgroups	GRU
13	Household	AlexNet	Sentiment, Spam	LSTM
14	Household	Inception	Sentiment, Questions	LSTM
15	Household	ResNet18	Sentiment, Newsgroups	LSTM
16	Household	VGG16	Spam, Questions	LSTM
17	Caltech101	AlexNet	Spam, Newsgroups	LSTM
18	Caltech101	Inception	Questions, Newsgroups	LSTM
19	Caltech101	ResNet18	Sentiment, Spam	GRU
20	Caltech101	VGG16	Sentiment, Questions	GRU

Submission Instructions

Submit your project as a folder named `GROUP_NUMBER_PREPROJECT` and zip the folder for submission. The plots and saved models that you generated during this lab should be placed in a folder called `results`. The grading rubric for this project is shown in Table 6. There are no unit tests for the project, but we will grade your code by attempting to run your code. There should be only one submission per group.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[3] "iNaturalist challenge at FGVC 2017." https://www.kaggle.com/c/inaturalist-challenge-at-fgvc-2017. Accessed: 2018-04-11.

[4] E. Learned-Miller, G. B. Huang, A. RoyChowdhury, H. Li, and G. Hua, "Labeled faces in the wild: A survey," in *Advances in face detection and facial image analysis*, pp. 189–248, Springer, 2016.

Table 6: Grading rubric

Points	Description
NOTE:	DO NOT SUBMIT THE DATASET
Part 1	
35	Working code for feature extraction based method (`feature.py`)
35	Working code for fine-tuning based method (`finetune.py`)
10	Fine-tuning code using data augmentation
10	Test accuracies and confusion matrices for the assigned dataset and model
10	Submit saved models (two in total)
Part 2	
40	Working code for first dataset
40	Working code for second dataset
10	Report final classification accuracy for both datasets
10	Submit saved models for both datasets
Total 200	

[5] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in neural information processing systems*, pp. 487–495, 2014.

[6] "iMaterialist challenge at FGVC 2018." https://www.kaggle.com/c/imaterialist-challenge-furniture-2018. Accessed: 2018-04-11.

[7] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," *Computer vision and Image understanding*, vol. 106, no. 1, pp. 59–70, 2007.

[8] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2014.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *IEEE International Conference on Computer Vision (CVPR)*, 2016.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *IEEE International Conference on Computer Vision (CVPR)*, pp. 1–9, 2015.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[15] B. Pang, L. Lee, and S. Vaithyanathan, “Thumbs up?: sentiment classification using machine learning techniques,” in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pp. 79–86, Association for Computational Linguistics, 2002.

[16] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with naive bayes-which naive bayes?,” in *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS 2006)*, 2006.

[17] X. Li and D. Roth, “Learning question classifiers,” in *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pp. 1–7, Association for Computational Linguistics, 2002.

[18] K. Lang, “Newsweeder: Learning to filter netnews,” in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 331–339, 1995.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.