Project 1

October 15, 2019

1 Comparison of Scattering Wavelet Transforms with Deep Neural Networks

This report tries to tackle the concept symmetry and impotance of invariance in machine learning models, especially for applications in signal processing, in this case, handwritten digit recognition. The report draws a comparison between using a black box convolutional neural network (such models have been incredibly powerful and expressive in tackling these problems in the recent years) vs a model arising from signal processing where these class of problems are central: a scattering wavelet transform.

In the report that follows, I have briefly attempted to do the following:

- Used AlexNet, VGG16 and ResNet18 pretrained on ImageNet as a fixed feature extractor for MNIST
- Use a scattering wavelet transform on MNIST as another method of extracting deformation invariant features
- Used Logistic regression and random forest for classification based on the extracted features
- Compare the results
- Provide high level explanation of the performance difference

The tests below have been done while enduring a significant computational restriction – every test has been run on my laptop on a CPU. Thus, some compromises have been made in relation to the exhaustiveness of the tests. In particular, I have not been able to run cross validation while training the models to choose the best parameters. I have also not been able to perform "transfer learning" to fine tune a CNN. In addition to this, I had to restrict myself to use one representative linear and one non-linear model for classification. My choices have been logistic regression and random forests. It is possible that a different carefully tuned model (like SVC or Gradient Boosted Trees or, of course, a deep CNN) perform slightly better than my results, but that would be beside the point of this report – the report only tries to do a high level comparison between the models.

That said however, I would like to continue to work to make this analysis stronger and would like to work on building a transfer-learned model even after the project submission deadline. There are a few other avenues I would like to explore in the (near) future, which are listed in the last section

1.1 Code

All the code for this project, is on my GitHub: **abnvpndy/math6380o**. I will continue to update this repository as and when I run additional tests/analyses and will also continue to update this

report. While I have mostly done most of the implementations myself, I have indeed perused the PyTorch official tutorials and some of the code might be inspired from those.

1.2 Team

I am not part of a team for this project. I have undertaken all the tasks by myself.

2 Convolutional Neural Networks

We first start by talking about convolutional neural networks in general and in the context of image recognition.

Convolutional neural networks, as the name suggests, uses convolution as a central technique – convolving inputs with, so called, "kernels" or "filters". Convolution in the discrete space can be thought of as a "weighted averaging" operation of some sort. For image recognition, convolutional neural networks use multiple kernels to convolve with the input in parallel and each kernel peforms the same "type" of convolution with the input. In other words, each kernel is thought of as extracting only one type of feature from the input, many such kernels acting in parallel means that each kernel will hopefully extract a different set of features which would be consumed by the deeper layers of the network. The so called "convolutional layer" of a convolutional neural network is composed of cascaded kernels that convolve with the input.

The output of the convolution is usually fed into the so called "detector" layer, which is composed of a non-linear activation function, usually ReLU in most implementations. ReLU "turns off" inputs where the kernels don't agree with the input. Thus the output is a much more sparse tensor. A downsampling operation is then performed on this output, which is referred to as "pooling". The pooling layer downsamples neighborhoods by, in essence, summarizing the outputs in the neighborhood to a representative statistic. This is also said to make the network resilient to translations of the input, depending on the size of the neighborhood considered. We will see later that Wavelet Scattering Transforms also use pooling as one of the operations for translational invariance

For the purpose of this report, all the code uses CNN implementations from PyTorch.

3 Wavelet Scattering

Fourier transforms, wavelet transforms and wavelet scattering transforms follow the same spirit of decomposing an input function into a set of special basis functions that are deformation invariant. Fourier transforms try to decompose a function into the Fourier basis, i.e. in simple terms, try to express a function as a linear combination of Fourier basis functions, which are complex sinusoids. Wavelet transforms try to do the same and decompose functions into the wavelet basis. Both of these methodologies introduce some deformation invariance. Wavelet Scattering tries to do this decomposition in a non-linear fashion – it uses the complex modulus function as the non-linearity, as far as I understand.

For the purpose of this report, all the code uses wavelet scattering implementation from Kymatio which implements the wavelet scattering transform as a convolutional neural network.

4 Methodology

4.0.1 CNNs:

General description of the methodology is as follows:

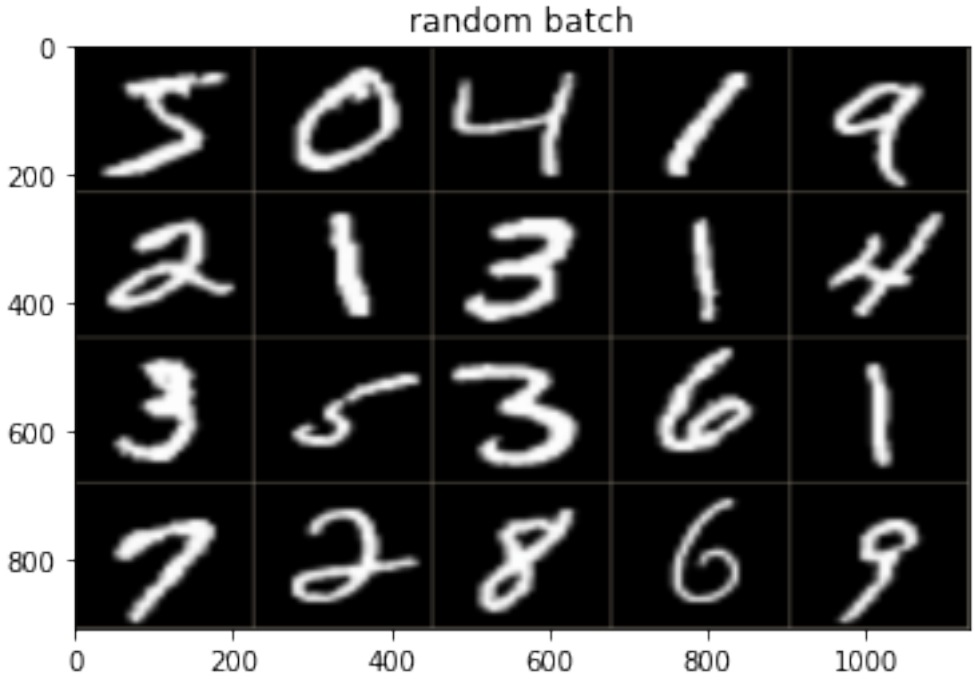
1. Use pretrained AlexNet, VGG16 and ResNet18 models from PyTorch
2. Create modified versions of AlexNet, VGG16 and Resnet18 after removing the last fully connected layer from the architecture and use the output of the penultimate layer as extracted features.
    a. Implementation can be found in `ConvNetMods.py` in `math6380o/project1/`. The initialization of the class constructs the architecture ommitting the last layer, if required. Forward function is updated to not pass the input through the last layer.
    b. The respective classses are called `alexnetmod`, `vgg16mod` and `resnetmod`, indicating a modification of the original classes
3. Extract and save the MNIST dataset.
    a. MNIST images are 28 x 28 pixels by default. The input sizes of ImageNet dataset, upon which all the aforementioned NNs are trained, is 3 x 224 x 224 where 3 indicates the RGB channel. Thus, MNIST is transformed by scaling to 3 x 224 x 224.
    b. The transformed MNIST dataset is computed in `Dataset.py`
4. Push the transformed MNIST dataset through the networks defined in #2.
    a. This happens in `FeatureExtractor.py`
    b. The shape of the features extracted from the models are as follows:
        i. AlexNet: (60000, 4096)
        ii. VGG16: (60000, 4096)
        iii. ResNet18: (60000, 512)
5. Train classifiers based on extracted data
    a. The utility function `train_classifier()` is used to train and store the trained models
    b. As stated before, we train Logistic Regression and Random Forests for this report
    c. The code supports Gradient Boosting Trees also and can also easily be extended to SVMs. The code also supports choosing parameters using cross validation, however that was ommitted due to lack of enough compute resources.

4.0.2 Wavelet Scattering Transform:

1. I use Kymatio which is an implementation of Wavelet Scattering Transform as a convolutional neural network. The documentation is hosted here: **kymat.io**
2. MNIST input images are no longer scaled and are fed into the scattering transform function as grayscale 28 x 28 images.
3. The shape of the features from the wavelet scattering transform is (60000, 81, 7, 7) which is flattend to (60000, 3969)

What does the MNIST data look like? Everyone knows what MNIST dataset look like, but let's visualize a small batch from the training data. I will use helper functions that I have created in `utils.py` for the rest of the report

[2]:
```
import torch, torchvision
import sys, os
sys.path.append("../")
from utils import imshow
from torch.utils.data import DataLoader
from torchvision.utils import make_grid
from Dataset import TransformedMNIST

mnist = TransformedMNIST()
train_images = mnist.get_train()
dataloader = DataLoader(train_images, batch_size=20)
batch_id, images_w_labels = next(enumerate(dataloader))
images =  images_w_labels[0]
labels = images_w_labels[1]
image_grid = make_grid(images, nrow=5)

imshow(image_grid, title="random batch")
```



Now let's load up the features extracted from each of the pretrained CNNs and make a tSNE plot of some random batches The MNIST dataset from the above was pushed through modified

versions of AlexNet, VGG16 and ResNet18 and were saved to disk, for the sake of time. If needed, the datasets can provided in order to run this notebook.

4.1 tSNE Plot

4.1.1 AlexNet

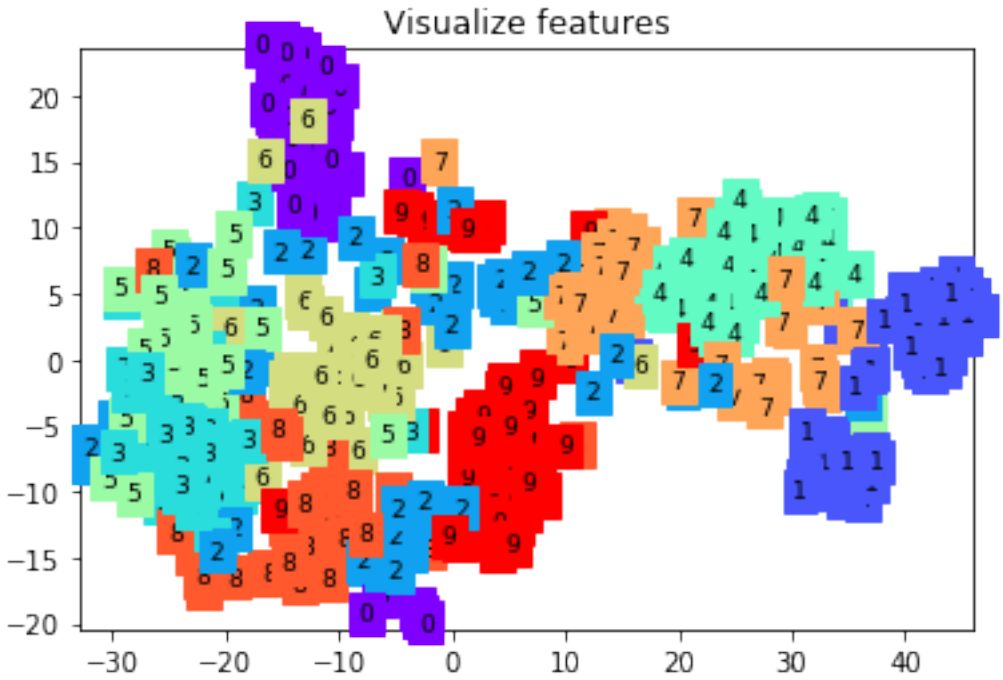
From the tSNE plot below, one can easily deduce the relative "distances" or "similarities" between digits.

For instance, 2 lies very close to 5. 0 is close 6, and 4 & 7 are both close to 1.

This indicates that given a certain amount of deformation to any of the aforementioned digits, it is rather likely for them to morph into another digit in its neighborhood in the lower dimensional space.

[5]:
```
# plot with tsne
from utils import visualize_tsne, get_stored_dataset
features, labels = get_stored_dataset("alexnet", train=True)
visualize_tsne(features, labels)
```

```
reading from local directory
```

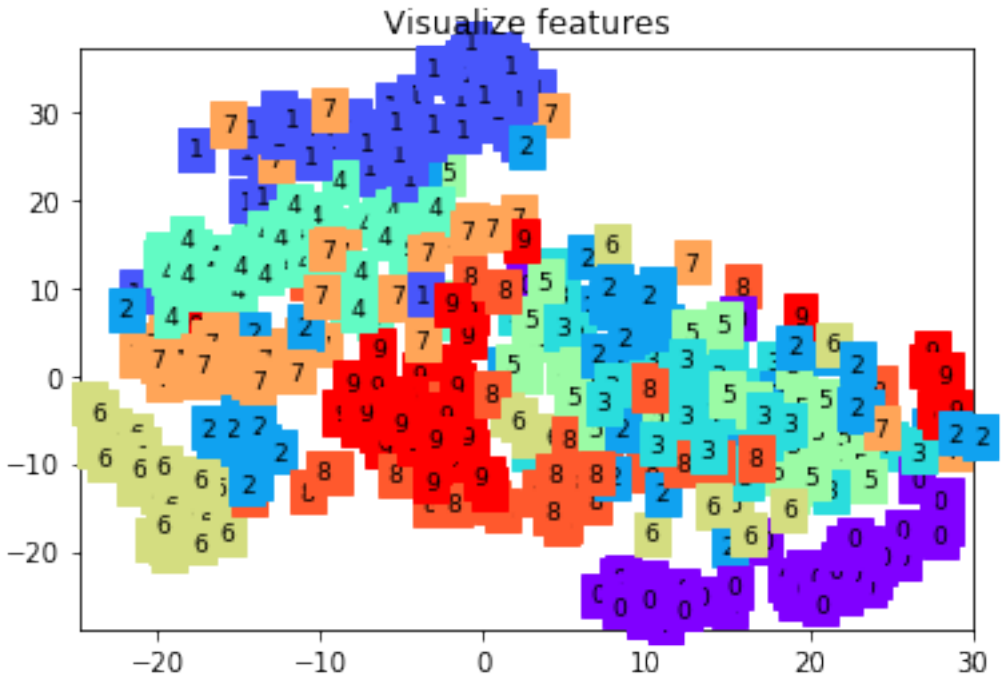


4.2 What do features from other CNNs look like?

4.2.1 VGG16

[6]:
```
# now we can use the utility function to load datasets
features, labels = get_stored_dataset("vgg16", train=True)
visualize_tsne(features, labels)
```

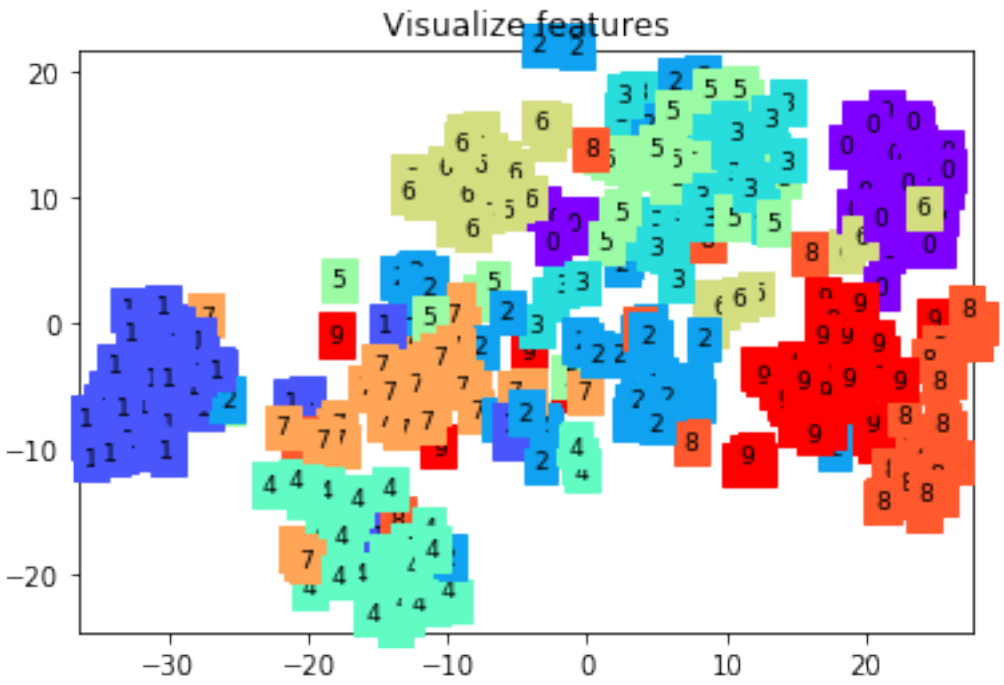
```
reading from local directory
```



4.2.2 ResNet

[8]:
```
features, labels = get_stored_dataset("resnet", train=True)
visualize_tsne(features, labels)
```

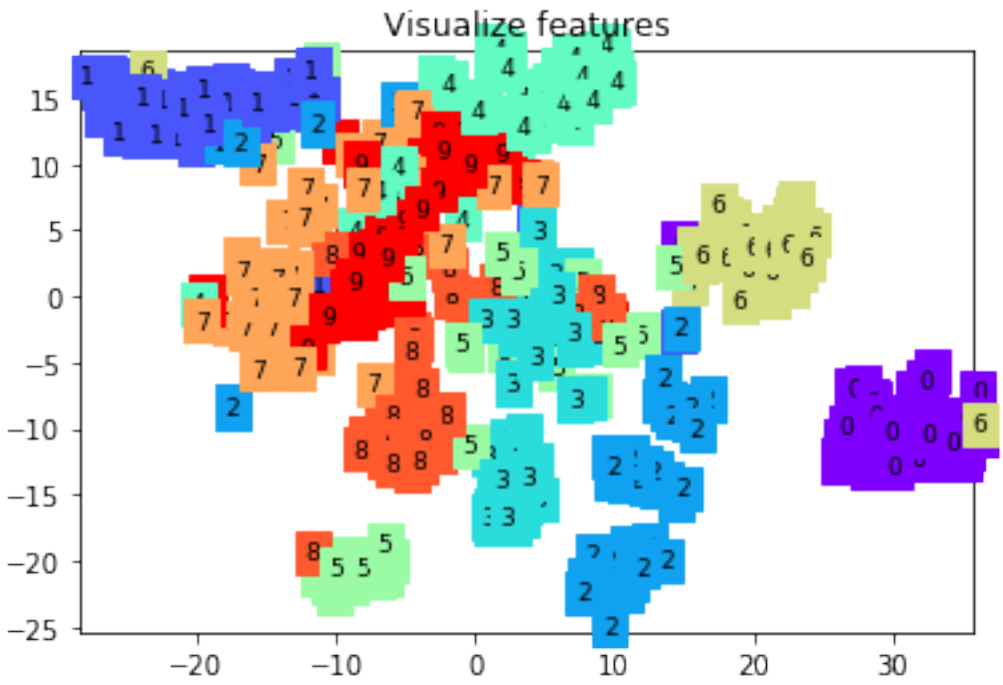
```
reading from local directory
```



4.3 What do features from the Wavelet Scattering transform look like?

[10]:
```
features, labels = get_stored_dataset("scattering", train=True)
visualize_tsne(features, labels)
```

```
reading from local directory
```



Purely based on the figures above, one would notice that the tSNE plot for the features extracted from the Wavelet Scattering transform, show the best separation between the classes, in the low dimensional space. Note that the tSNE visualizations that we run would only plot 500 points randomly, to reduce the time overhead.

4.4 Models trained using the features above

The table below would be helpful to understand all the results that follow. Logistic Regression and Random Forests were trained as the pick for 1 linear and 1 non-linear model to use. Of course, as stated earlier, one could be more rigorous and include more such models to train and run cross validation to choose the best params, but the hope is that the gist of the analysis is captured by these two.

	Logistic Regression	Random Forest
AlexNet Features		
VGG16 Features		
ResNet Features		
Wavelet Scattering Features		

The entries will be filled out later when accuracy scores area available.

The models are already trained and stored on disk.

The model description is defined in `utils.py` in the function `train_classifier`

In order to train logistic regression on features extracted from AlexNet, one could call the function as follows:

```
from utils import train_classifier
train_classifier("log_reg",
                 dataset="alexnet",
                 cv=False,
                 save_to_disk=True)
```

`cv=False` means that we do not want to run Cross Validation. `save_to_disk=True` is self explanatory

Similarly, in order to test the classifier, i.e. get predictions on one of the saved test datasets, one could do the following

```
from utils import test_classifier
test_classifier("log_reg",
                dataset="alexnet",
                cv=False,
                load_from_disk=True)
```

Let us load up some saved (and trained) models and look at their parameters. For this we are going to use the helper function, `get_stored_model()` in `utils.py`

[5]:
```
from utils import get_stored_model
log_reg = get_stored_model("log_reg", "alexnet")
print(log_reg)
```

```
reading from local directory
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.1, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='log', max_iter=1000,
              n_iter_no_change=5, n_jobs=None, penalty='elasticnet',
              power_t=0.5, random_state=None, shuffle=True, tol=0.1,
              validation_fraction=0.1, verbose=5, warm_start=False)
```

It can be seen, from the output above, that the class used for logistic regression is `SGDClassifier` instead of `LogisticRegression`, from scikit-learn. This is to ensure that inference is done using `Stochastic Gradient Descent`. Although there are some supported solvers in the new `LogisticRegression` class that perform stochastic averaged gradient descent, but in my short experiment, they were not able to converge as fast as running `SGDClassifier` with `loss='log'`. We can see that we use penalty `elasticnet` to hopefully have the best balance between `l1` and `l2` regularization.

[6]:
```
params = log_reg.get_params()
```

[8]:
```
params
```

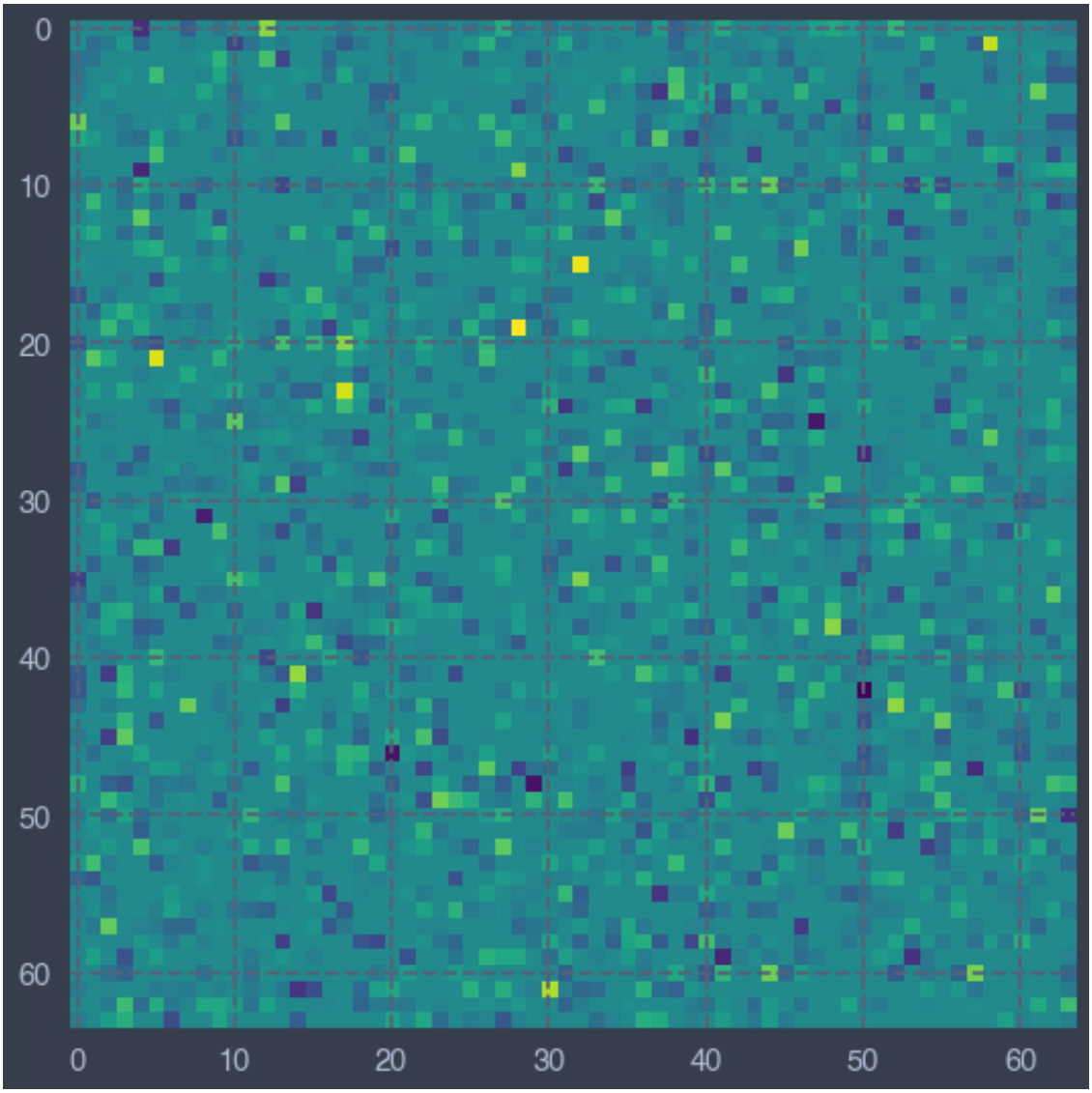
[8]:
```
{'alpha': 0.0001,
 'average': False,
 'class_weight': None,
 'early_stopping': False,
 'epsilon': 0.1,
 'eta0': 0.1,
 'fit_intercept': True,
```

```
 'l1_ratio': 0.15,
 'learning_rate': 'optimal',
 'loss': 'log',
 'max_iter': 1000,
 'n_iter_no_change': 5,
 'n_jobs': None,
 'penalty': 'elasticnet',
 'power_t': 0.5,
 'random_state': None,
 'shuffle': True,
 'tol': 0.1,
 'validation_fraction': 0.1,
 'verbose': 5,
 'warm_start': False}
```

Let us visualize some weights after converting them to 64 x 64 images

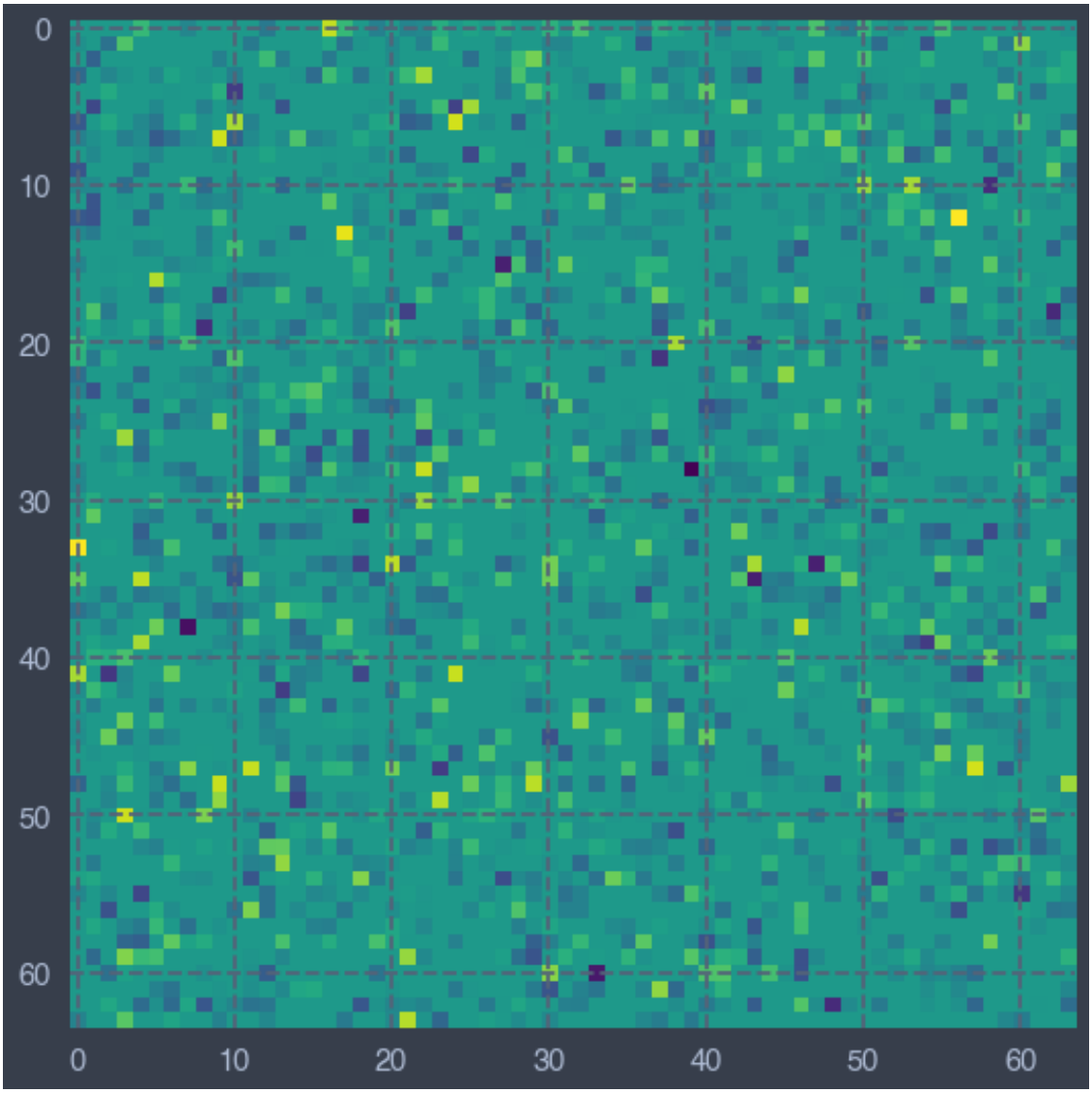
[28]:
```
from matplotlib.pyplot import imshow
import matplotlib.pylab as plt

# Let us reshape the coeffs
log_reg.densify()
coeffs_dense =  log_reg.coef_
type(coeffs_dense)
coeffs_dense = coeffs_dense.reshape(10, 1, 4096)
coeffs_dense = coeffs_dense.reshape(10, 64, 64)

imshow(coeffs_dense[0])
```

[28]:
```
<matplotlib.image.AxesImage at 0x1c1ec7d210>
```



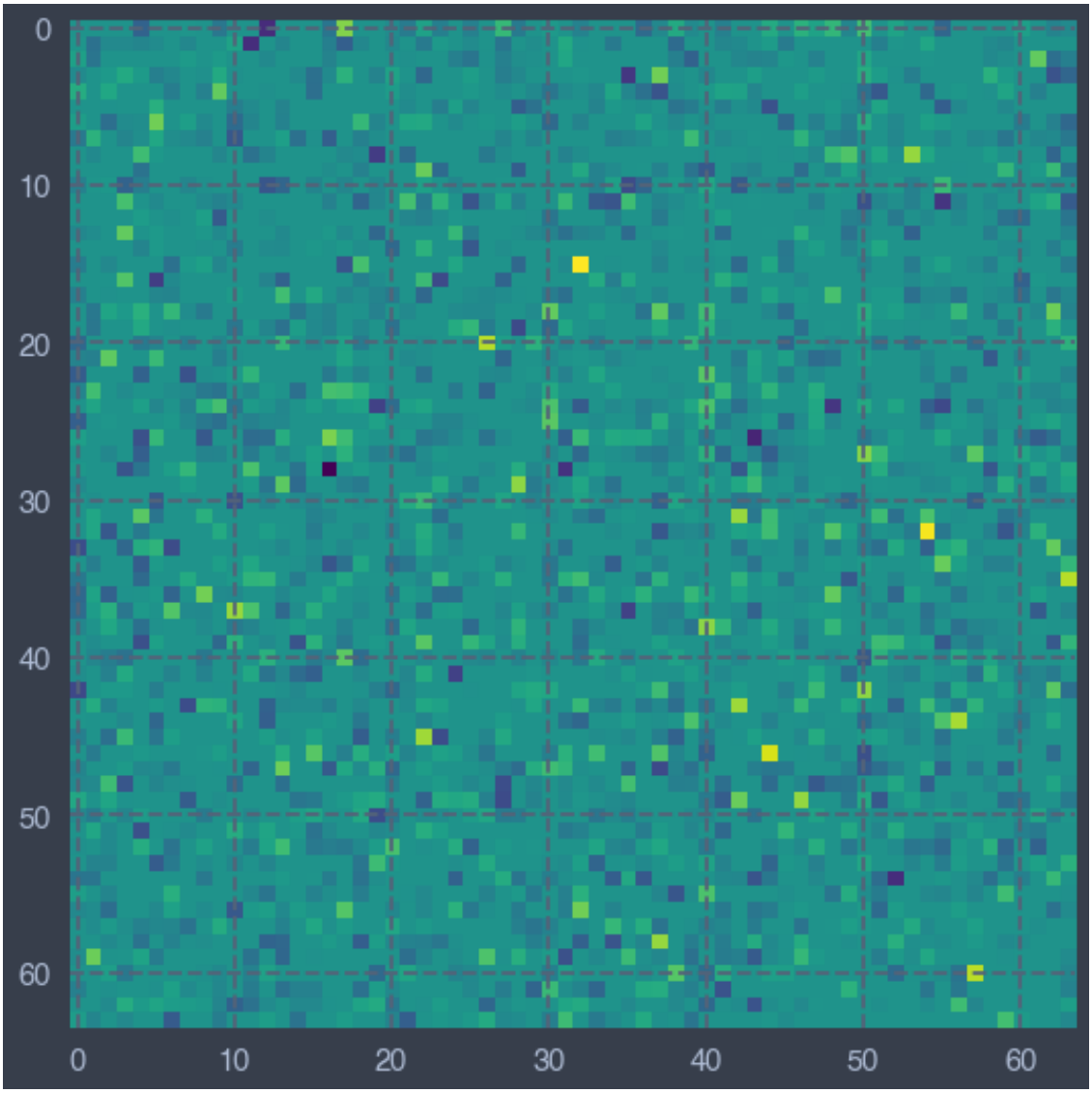
[29]:
```
imshow(coeffs_dense[1])
```

[29]:
```
<matplotlib.image.AxesImage at 0x1c1ecf1650>
```



[30]:
```
imshow(coeffs_dense[9])
```

[30]:
```
<matplotlib.image.AxesImage at 0x1c1ed5d390>
```



Let us now run tSNE on these weights to find out if we see a pattern

[36]:
```
from sklearn.manifold import TSNE
from utils import plot_with_labels
tsne = TSNE(n_components=2, perplexity=30.0, n_iter=50000, init="pca")
# let's generate some names for the classes because we are not sure who they␣
 ↪belong to
labels = [x for x in range(10)]
embeddings = tsne.fit_transform(log_reg.coef_)
```

Keep in mind that in this plot below, the labels may or may not correspond to the digits 0 to 10. The important thing to observe here is that logistic regression has definitely learned specific properties of the classes seeing how far apart they are in a low dimensional space.

[37]:
```
plot_with_labels(embeddings, labels)
```

Let us look at prediction accuracy of this model. We will use `classification_report` function from `sklearn.metrics`

[39]:
```
from sklearn.metrics import classification_report
from utils import test_classifier
# evaluate predictions for the model
preds = test_classifier("log_reg", dataset="alexnet")
# get test data
test_features, test_labels = get_stored_dataset("alexnet", train=False)
rep = classification_report(test_labels, preds)
print(rep)
```

```
reading from local directory
reading from local directory
reading from local directory
reading from local directory
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
           1       0.96      0.99      0.98      1135
           2       0.94      0.93      0.94      1032
           3       0.94      0.95      0.94      1010
           4       0.97      0.97      0.97       982
           5       0.93      0.91      0.92       892
           6       0.98      0.95      0.97       958
           7       0.95      0.97      0.96      1028
           8       0.97      0.97      0.97       974
           9       0.98      0.95      0.97      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000
```

The average accuracy for the model is around 96%. As stated before, the idea of the model comparison here is to look at high level results, thus we are only looking at 2 significant figures for accuracy. If we were training a state of the art model that we hope would beat all current models, the obviously that would require more rigor.

Let us now run similar prediction reports for all other logistic regression models. Let us first fold the code above in a helper function to help generate classification report

[44]:
```
def get_classification_report(model_name, dataset_name):
    # evaluate predictions for the model
    preds = test_classifier(model_name, dataset=dataset_name)
    # get test data
    test_features, test_labels = get_stored_dataset(dataset_name, train=False)
    rep = classification_report(test_labels, preds)
    print(rep)
```

Logistic Regression - VGG16

[45]:
```
get_classification_report("log_reg", "vgg16")
```

```
reading from local directory
reading from local directory
reading from local directory
reading from local directory
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
           1       0.96      0.99      0.97      1135
           2       0.96      0.95      0.95      1032
           3       0.94      0.95      0.95      1010
           4       0.95      0.97      0.96       982
```

```
           5       0.95      0.93      0.94       892
           6       0.97      0.94      0.96       958
           7       0.97      0.95      0.96      1028
           8       0.96      0.97      0.97       974
           9       0.97      0.93      0.95      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000
```

Logistic Regression - ResNet

[46]:
```
get_classification_report("log_reg", "resnet")
```

```
reading from local directory
reading from local directory
reading from local directory
reading from local directory
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.95      0.95      0.95      1032
           3       0.96      0.96      0.96      1010
           4       0.97      0.99      0.98       982
           5       0.96      0.95      0.96       892
           6       0.97      0.97      0.97       958
           7       0.95      0.97      0.96      1028
           8       0.97      0.97      0.97       974
           9       0.97      0.95      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

Logistic Regression - Wavelet scattering

[47]:
```
get_classification_report("log_reg", "scattering")
```

```
reading from local directory
reading from local directory
reading from local directory
reading from local directory
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
```

```
           1       0.98      1.00      0.99      1135
           2       0.98      0.99      0.98      1032
           3       0.99      0.99      0.99      1010
           4       0.98      0.99      0.99       982
           5       0.98      0.99      0.99       892
           6       0.99      0.98      0.99       958
           7       0.98      0.98      0.98      1028
           8       0.99      0.98      0.98       974
           9       0.98      0.97      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.99      0.98      0.98     10000
weighted avg       0.99      0.98      0.98     10000
```

Let us now look at Random Forests Of course it's rather hard to visualize what a random forest is doing or learning. The important point to note here is that `n_estimators == 100`, which means the random forest will grow 100 random trees. I ran a small experiment and compared the accuracy of a random forest model with 1000 trees with that with 100 trees on alexnet's extracted features. Turns out that the prediction power of the random forest did not increase with increased flexibility.

We will not be able to visualize random forest weights etc in the same fashion as that for logistic regression. We will thus only focus on prediction accuracy.

[43]:
```
rf = get_stored_model("random_forest", "alexnet")
print(rf)
```

```
reading from local directory
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=5, warm_start=False)
```

Random Forest - AlexNet

[48]:
```
get_classification_report("random_forest", "alexnet")
```

```
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.1s remaining:    0.0s
```

```
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.2s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    1.2s finished

reading from local directory
              precision    recall  f1-score   support

           0       0.96      0.97      0.96       980
           1       0.99      0.99      0.99      1135
           2       0.90      0.92      0.91      1032
           3       0.90      0.91      0.90      1010
           4       0.93      0.97      0.95       982
           5       0.90      0.88      0.89       892
           6       0.96      0.94      0.95       958
           7       0.96      0.94      0.95      1028
           8       0.96      0.93      0.94       974
           9       0.96      0.94      0.95      1009

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000
```

Random Forest - VGG16

[49]:
```
get_classification_report("random_forest", "vgg16")
```

```
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.9s finished

reading from local directory
              precision    recall  f1-score   support

           0       0.91      0.97      0.94       980
           1       0.98      0.98      0.98      1135
           2       0.89      0.91      0.90      1032
           3       0.90      0.93      0.91      1010
           4       0.89      0.94      0.92       982
           5       0.89      0.85      0.87       892
           6       0.95      0.92      0.94       958
           7       0.93      0.90      0.92      1028
           8       0.94      0.93      0.94       974
```

```
           9       0.94      0.89      0.91      1009

    accuracy                           0.92     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.92      0.92      0.92     10000
```

Random Forest - ResNet

[50]:
```
get_classification_report("random_forest", "resnet")
```

```
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s

reading from local directory
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
           1       0.99      0.99      0.99      1135
           2       0.93      0.92      0.93      1032
           3       0.94      0.96      0.95      1010
           4       0.94      0.98      0.96       982
           5       0.93      0.92      0.93       892
           6       0.96      0.96      0.96       958
           7       0.95      0.95      0.95      1028
           8       0.97      0.95      0.96       974
           9       0.97      0.93      0.95      1009

    accuracy                           0.95     10000
   macro avg       0.95      0.95      0.95     10000
weighted avg       0.96      0.95      0.95     10000


[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.5s finished
```

Random Forest - Wavelet Scattering

[51]:
```
get_classification_report("random_forest", "scattering")
```

```
reading from local directory
reading from local directory
reading from local directory
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.8s finished

reading from local directory
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.99      0.98      1032
           3       0.98      0.99      0.98      1010
           4       0.98      0.99      0.99       982
           5       0.99      0.98      0.98       892
           6       0.99      0.98      0.98       958
           7       0.97      0.98      0.98      1028
           8       0.98      0.98      0.98       974
           9       0.99      0.96      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

So, let's calculate balanced accuracy scores and fill out our table from before

[52]:
```
from sklearn.metrics import balanced_accuracy_score
def get_balanced_accuracy_score(model_name, dataset_name):
        # evaluate predictions for the model
    preds = test_classifier(model_name, dataset=dataset_name)
    # get test data
    test_features, test_labels = get_stored_dataset(dataset_name, train=False)
    score = balanced_accuracy_score(test_labels, preds)
    print("{} {} accuracy: {}".format(model_name, dataset_name, score))
```

[54]:
```
get_balanced_accuracy_score("log_reg", "alexnet")
get_balanced_accuracy_score("log_reg", "vgg16")
get_balanced_accuracy_score("log_reg", "resnet")
get_balanced_accuracy_score("log_reg", "scattering")

get_balanced_accuracy_score("random_forest", "alexnet")
get_balanced_accuracy_score("random_forest", "vgg16")
get_balanced_accuracy_score("random_forest", "resnet")
get_balanced_accuracy_score("random_forest", "scattering")
```

```
reading from local directory
reading from local directory
```

```
reading from local directory
reading from local directory
log_reg alexnet accuracy: 0.958397377218256
reading from local directory
reading from local directory
reading from local directory
reading from local directory
log_reg vgg16 accuracy: 0.9570058472359039
reading from local directory
reading from local directory
reading from local directory
reading from local directory
log_reg resnet accuracy: 0.9676573213192252
reading from local directory
reading from local directory
reading from local directory
reading from local directory
log_reg scattering accuracy: 0.9849363381990764
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.8s finished

reading from local directory
random_forest alexnet accuracy: 0.9393313488150652
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.8s finished

reading from local directory
random_forest vgg16 accuracy: 0.9229875912877249
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
```

```
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.6s finished

reading from local directory
random_forest resnet accuracy: 0.9543914458260427
reading from local directory
reading from local directory
reading from local directory

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.8s finished

reading from local directory
random_forest scattering accuracy: 0.9821851756775063
```

5 Performance

	Logistic Regression	Random Forest
AlexNet Features	95.84%	93.93%
VGG16 Features	95.70%	92.30%
ResNet Features	96.77%	95.44%
Wavelet Scattering Features	98.49%	98.22%

6 Conclusion

We see from the performance tables that Logistic Regression trained on features extracted from Wavelet Scattering Transform emerges as the winner, beating Random Forest by a narrow margin.

This is a demonstration of the following: - The input features are high dimensional, thus the belief is that logistic regression is better able to introduce some sparsity in the data thereby avoiding overfitting and generalizing very well.

- Random Forests are also very good at handling high dimensional data, rapidly getting rid of features that don't contribute to information gain while growing a tree or a branch, thus it is hard to point out an exact reason why logistic regression generalized better.

It is also very important to keep in mind that this test was done only on 1 training, test data pair. It is generally better practice to mix the given test dataset with the training dataset and randomly split that into training and test data for 1 particular experiment. Running multiple such experiments and obtaining a spread of accuracy values is important to account for the dependence of the generalization accuracy on specific training/test sets. It is usually a good practice to

plot a box plot of the accuracy values for multiple experiments done for each model-dataset pair and compare the medians of those. What is done in this report is a rather soft version of what I describe. The dataset could also be augmented with deformed images and the experiment could be done on that as well.

It is also very important to note that while the wavelet scattering transform offers some sort of theoretical guarantees on invariance to deformations, the deep CNNs do not offer any such guarantees. CNNs are engineered per application and finely tuned in order to work for that particular application. In this case, the pretrained CNNs that were used, were pretrained on ImageNet datasets which is not exactly "similar" to MNIST. This could also be a reason for the lag in performance of the models trained using those CNN features.

7 Further Avenues to Explore

One of the paths that I wish to explore is the idea of probabilistic symmetry and invariance. A paper by Bloem-Reddy and Teh, 2019 explores this idea rather deeply and it would be very interesting to compare some of the probabilistic techniques described there with the experiments done as part of this report.

This report could be more rigorous: the best performing models could be chosen by cross validation, multiple experiments could be run using multiple train-test splits as described in the preceding section, and also a deep CNN could be trained using transfer learning to see if logistic regression and random forests could be outperformed. I hope to work on the last item in the coming week or so.