Higher Performance with Less Data via Capsule Networks and Active Learning

Chris Aasted, PhD

Lockheed Martin Autonomous Systems

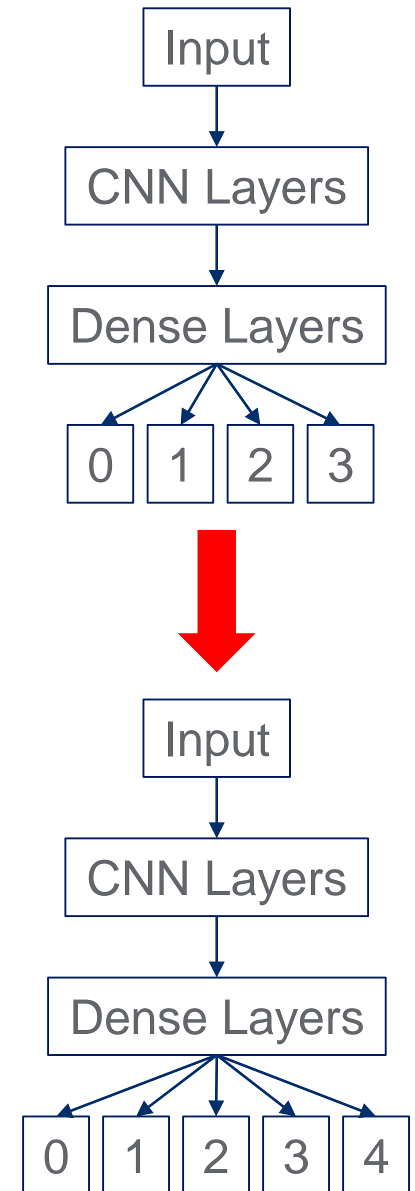
Outline

- Problem Statement
- Capsule Networks
- Transfer Learning
- Active Learning
- Datasets
- Training the Original Classifier
- Training the New Classifier
- Results
- Conclusions
- Acknowledgements

Problem Statement

Deep learning has advanced the state of the art for a number of computer vision tasks. However, deep learning generally requires a very large training dataset to achieve this performance and adding a new label to an existing classifier often requires retraining the classifier from scratch. This necessitates maintaining access to the original dataset as well as collecting a sufficiently large number of samples for a new label to balance the new training set.

In this study, we investigated methods to add a new class to an existing classifier with as few samples of the new label, and from the previous training set, as possible. We report results from applying this technique to two computer vision datasets: MNIST and SENSIAC.



Capsule Networks

- Capsule Layer
  - Creates groups of neurons that form vectors instead of a scalar activation
  - Inter-capsule weights are updated using dynamic routing algorithm
- Mask
  - During training, mask all but the correct label's vector
  - During testing, pass all label vectors so that Length can be used to determine the vector with the largest magnitude
- Length
  - Calculates the magnitude of each output capsule vector
- Squash Function
  - Drives the length of large vectors to 1 and small vectors to 0
- Margin Loss

$$L_k = T_k \, max(0, m^+ - \|v_k\|)^2 + \lambda(1 - T_k) \, max(0, \|v_k\| - m^-)^2 \, , \; where \; T_k = \begin{cases} 1, & digit \; of \; class \; k \; present \\ 0, & otherwise \end{cases}$$

SABOUR, FROSST, AND HINTON. "DYNAMIC ROUTING BETWEEN CAPSULES." ARXIV:1710.09829V2 [CS.CV] 7 NOV 2017

Transfer Learning

- In General
  - Facilitates training high quality classifiers with significantly smaller training sets (as compared to ImageNet)
  - Reduces training time for convolutional layers
  - Improves generalization
  - Transfers very well between different classes
  - Transfer reasonably well to different sensor types
- Add-a-class Use Case
  - Since the new training set highly overlaps with the original, transfer learning **significantly** reduces the training time
  - Catastrophic forgetting becomes a consideration
  - Even more layers may be eligible for transfer
    - Even just a new output layer can occasionally be sufficient to add a new label

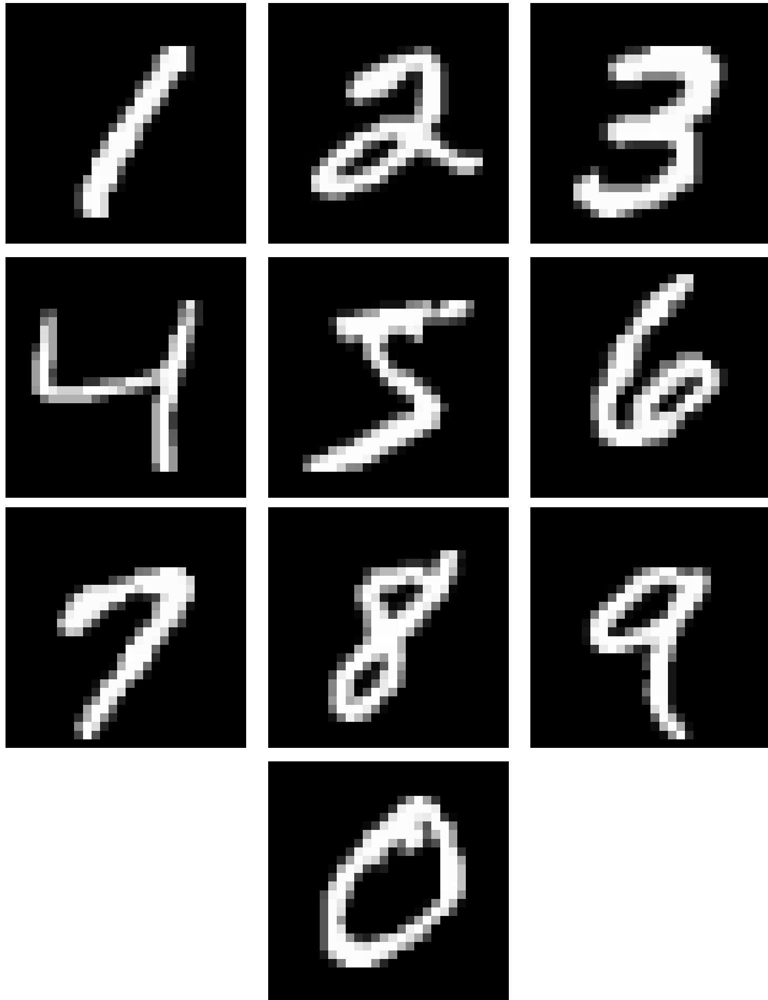
YOSINSKI, CLUNE, BENGIO, AND LIPSON. "HOW TRANSFERABLE ARE FEATURES IN DEEP NEURAL NETWORKS?" ARXIV:1411.1792V1 [CS.LG] 6 NOV 2014

Active Learning

1. Instead of starting by labeling every training sample that is available, start by labeling a limited set of randomly selected samples and train an initial network.
2. Use the network to make predictions on the remaining unlabeled training samples and select the ones with the highest entropy.
3. Label N of the least certain samples and add them to the training set. It may be beneficial to manually keep the number of training samples per class balanced.
4. Continue training the network and repeat steps 2-4 until the validation performance levels off or you reach the threshold for how many samples you are able to label.

DEEP ACTIVE LEARNING
ADAM LESNIKOWSKI (NVIDIA)
ON-DEMAND.GPUTECHCONF.COM
/GTC/2018/VIDEO/S8692/

Datasets – MNIST



- 28x28-pixel handwritten digits
- 60,000 training samples
- 10,000 test samples
- 10,000 of the 60,000 training samples reserved for validation
- No additional treatment

http://yann.lecun.com/exdb/mnist/

Datasets – SENSIAC (Now Available from DSIAC)

"The ATR (Automated Target Recognition) Algorithm Development Image Database package contains a large collection of visible and MWIR (mid-wave infrared) imagery collected by the US Army Night Vision and Electronic Sensors Directorate (NVESD) intended to support the ATR algorithm development community. This database provides a broad set of infrared and visible imagery along with ground truth data for ATR algorithm development and training."

- 207 GB of MWIR imagery
- 106 GB of visible imagery
- Ground truth data
- Targets include people, foreign military vehicles, and civilian vehicles at a variety of ranges and aspect angles.
- All imagery was taken using commercial cameras operating in the MWIR and visible bands.

https://www.dsiac.org/resources/research-materials/cds-dvds-databases/atr-algorithm-development-image-database

Capsule Networks – Source Code

- For the purpose of generating publicly shareable results, the repository https://github.com/XifengGuo/CapsNet-Keras was used to generate the results presented here (MIT License).
- Please refer to the CapsNet-Keras repo for the following class and function definitions:
    - Classes
        - CapsuleLayer
        - Mask
        - Length
    - Functions
        - squash_function
        - margin_loss

Training the Original Classifier

```
def train_xfer_network(X_train, y_train, X_val, y_val, vgg_model):
    # Normal Convolutional Layer
    caps_xfer_in = Input(shape=vgg_model.output.shape[1:])
    caps_layer = Conv2D(8 * 32, (5, 5), (2, 2), padding='valid', activation='relu')(caps_xfer_in) # 32 -> 28 -> 14
    caps_layer = BatchNormalization()(caps_layer)

    # Primary Capsule Conv
    caps_layer = Conv2D(8 * 32, (9, 9), (1, 1), padding='valid', activation='relu')(caps_layer) # 14 -> 6
    caps_layer = BatchNormalization()(caps_layer)
    caps_xfer_out = Flatten()(caps_layer)

    xfer_model = Model(caps_xfer_in, caps_xfer_out)
    xfer_model.compile(optimizer=Adam(lr=0.001), loss='categorical_crossentropy', metrics=['categorical_accuracy'])

    # Full Model
    caps_input = Input(shape=X_train.shape[1:]) # 128 x 128 x 3
    vgg_model.trainable = False
    vgg_layer = vgg_model(caps_input)
    xfer_layer = xfer_model(vgg_layer)

    # Primary Capsule Activation
    caps_layer = Reshape([(32 * 6 * 6), 8])(xfer_layer) # Conserve ~1,152 vectors of length 8?
    caps_layer = Lambda(squash_function)(caps_layer)

    # Capsule Layer
    caps_layer = CapsuleLayer(y_train.shape[1], 16, 3)(caps_layer) # Output shape: [None, 12, 16]
    caps_output = Length()(caps_layer)
    capsule_model = Model(caps_input, caps_output)
    capsule_vectors = Model(caps_input, caps_layer)

    capsule_model.compile(optimizer='adadelta', loss=[margin_loss], metrics=['categorical_accuracy'])

    # Decoder Network
    decoder_input = Input(shape=(y_train.shape[1], 16)) # Input shape: [classes, 16]
    decoder_layer = Flatten()(decoder_input)
    decoder_layer = Dense(512, activation='relu')(decoder_layer)
    decoder_layer = Dense(1024, activation='relu')(decoder_layer)
    decoder_layer = Dense(caps_input.shape[1] * caps_input.shape[2] * caps_input.shape[3])(decoder_layer)
    decoder_output = Reshape(caps_input.shape[1:])(decoder_layer)
    decoder_model = Model(decoder_input, decoder_output)

    truth_input = Input(shape=(y_train.shape[1], ))

    stacked_input = Input(caps_input.shape[1:])
    stacked_layer = capsule_vectors(stacked_input)
    capsule_output = Length()(stacked_layer)
    stacked_layer = Mask()([stacked_layer, truth_input])
    stacked_output = decoder_model(stacked_layer)

    stacked_model = Model([stacked_input, truth_input], [capsule_output, stacked_output])

    stacked_model.compile(optimizer='adadelta', loss=[margin_loss, 'mse'], metrics={'length_2': 'categorical_accuracy'})

    stacked_model.fit([X_train, y_train], [y_train, X_train], epochs=10, batch_size=32, verbose=1,
                validation_data=[[X_val, y_val], [y_val, X_val]])

    return xfer_model
```

Notes:

The vgg_model that is passed into the transfer network training function consists of the first nine layers of VGG16 and is used for the SENSIAC dataset, but not for MNIST:

```
vgg_model = VGG16(include_top=False, weights='imagenet', input_shape=(X_train.shape[1:]))
vgg_model = Model(vgg_model.input, vgg_model.get_layer("block3_conv3").output)
```

In the third line of train_xfer_network, padding is set to ‘valid’ for SENSIAC and ‘same’ for MNIST. This results in the same output tensor shape for both datasets.

Original Classifier – Transfer Model

```
# Normal Convolutional Layer
caps_xfer_in = Input(shape=vgg_model.output.shape[1:])
caps_layer = Conv2D(8 * 32, (5, 5), (2, 2), padding='valid', activation='relu')(caps_xfer_in) # 32 -> 28 -> 14
caps_layer = BatchNormalization()(caps_layer)

# Primary Capsule Conv
caps_layer = Conv2D(8 * 32, (9, 9), (1, 1), padding='valid', activation='relu')(caps_layer) # 14 -> 6
caps_layer = BatchNormalization()(caps_layer)
caps_xfer_out = Flatten()(caps_layer)

xfer_model = Model(caps_xfer_in, caps_xfer_out)
xfer_model.compile(optimizer=Adam(lr=0.001), loss='categorical_crossentropy’,
                   metrics=['categorical_accuracy'])
```

Original Classifier – Capsule Network Model

```
# Full Model
caps_input = Input(shape=X_train.shape[1:]) # 128 x 128 x 3
vgg_model.trainable = False
vgg_layer = vgg_model(caps_input)
xfer_layer = xfer_model(vgg_layer)

# Primary Capsule Activation
caps_layer = Reshape([(32 * 6 * 6), 8])(xfer_layer) # Conserve ~1,152 vectors of length 8?
caps_layer = Lambda(squash_function)(caps_layer)

# Capsule Layer
caps_layer = CapsuleLayer(y_train.shape[1], 16, 3)(caps_layer) # Output shape: [None, 12, 16]
caps_output = Length()(caps_layer)
capsule_model = Model(caps_input, caps_output)
capsule_vectors = Model(caps_input, caps_layer)

capsule_model.compile(optimizer='adadelta', loss=[margin_loss], metrics=['categorical_accuracy'])
```

Original Classifier – Decoder Model

```
# Decoder Network
decoder_input = Input(shape=(y_train.shape[1], 16)) # Input shape: [classes, 16]
decoder_layer = Flatten()(decoder_input)
decoder_layer = Dense(512, activation='relu')(decoder_layer)
decoder_layer = Dense(1024, activation='relu')(decoder_layer)
decoder_layer = Dense(caps_input.shape[1] * caps_input.shape[2] * caps_input.shape[3])(decoder_layer)
decoder_output = Reshape(caps_input.shape[1:])(decoder_layer)
decoder_model = Model(decoder_input, decoder_output)
```

Original Classifier – Stacked Model

```
truth_input = Input(shape=(y_train.shape[1], ))

stacked_input = Input(caps_input.shape[1:])
stacked_layer = capsule_vectors(stacked_input)
capsule_output = Length()(stacked_layer)
stacked_layer = Mask()([stacked_layer, truth_input])
stacked_output = decoder_model(stacked_layer)

stacked_model = Model([stacked_input, truth_input], [capsule_output, stacked_output])

stacked_model.compile(optimizer='adadelta', loss=[margin_loss, 'mse’],
                      metrics={'length_2': 'categorical_accuracy'})

stacked_model.fit([X_train, y_train], [y_train, X_train], epochs=10, batch_size=32, verbose=1,
                  validation_data=[[X_val, y_val], [y_val, X_val]])
```

Active Learning – Helper Function

```
def get_new_indices(indices, X, y, model, new_samples_per_class=1):
    # Active Learning
    preds = model.predict(X)
    certainties = np.max(preds, axis=1)
    ranked_indices = np.argsort(certainties)

    new_indices = 0
    indices_per_class = [0] * y.shape[1]
    check_index = 0
    while new_indices < y.shape[1] * new_samples_per_class:
        sample_class = np.argmax(y[ranked_indices[check_index]])
        if (indices_per_class[sample_class] < new_samples_per_class
            and check_index not in indices):
            indices.append(ranked_indices[check_index])
            indices_per_class[sample_class] += 1
            new_indices += 1
        check_index += 1
    return indices
```

Training the New Classifier

```
def train_capsule_network(X_train, y_train, X_train_new, y_train_new, X_val_new, y_val_new, vgg_layers, xfer_layers):
    # Merge
    X_train = np.concatenate((X_train, X_train_new), axis=0)
    y_train = np.concatenate((y_train, y_train_new), axis=0)

    # Input Layer
    caps_input = Input(shape=X_train.shape[1:]) # 128 x 128 x 3
    print(caps_input.shape)

    # VGG Layers
    vgg_layers.trainable = False
    vgg_out = vgg_layers(caps_input)

    # Transfer Layers
    xfer_layers.trainable = True
    xfer_out = xfer_layers(vgg_out)

    print(xfer_out.shape)

    # Primary Capsule Activation
    caps_layer = Reshape([(32 * 6 * 6), 8])(xfer_out) # Conserve ~1,152 vectors of length 8?
    caps_layer = Lambda(squash_function)(caps_layer)

    # Capsule Layer
    caps_layer = CapsuleLayer(y_train.shape[1], 16, 3)(caps_layer) # Output shape: [None, 12, 16]
    caps_output = Length()(caps_layer)
    capsule_model = Model(caps_input, caps_output)
    capsule_vectors = Model(caps_input, caps_layer)

    capsule_model.compile(optimizer=Adam(lr=0.001), loss=[margin_loss], metrics=['categorical_accuracy'])
    capsule_model.summary()

    # Decoder Network
    decoder_input = Input(shape=(y_train.shape[1], 16)) # Input shape: [12, 16]
    decoder_layer = Flatten()(decoder_input)
    decoder_layer = Dense(512, activation='relu')(decoder_layer)
    decoder_layer = Dense(1024, activation='relu')(decoder_layer)
    decoder_layer = Dense(caps_input.shape[1] * caps_input.shape[2] * caps_input.shape[3])(decoder_layer)
    decoder_output = Reshape(caps_input.shape[1:])(decoder_layer)
    decoder_model = Model(decoder_input, decoder_output)

    truth_input = Input(shape=(y_train.shape[1], ))

    stacked_input = Input(caps_input.shape[1:])
    stacked_layer = capsule_vectors(stacked_input)
    capsule_output = Length()(stacked_layer)
    stacked_layer = Mask()([stacked_layer, truth_input])
    stacked_output = decoder_model(stacked_layer)

    stacked_model = Model([stacked_input, truth_input], [capsule_output, stacked_output])

    stacked_model.compile(optimizer='adadelta', loss=[margin_loss, 'mse'], metrics={'length_2': 'categorical_accuracy'})

    cycles = int((total_samples - initial_samples) / samples_per_round)
    indices = get_starting_indices(y_train, initial_samples)
    stacked_model.fit([X_train[indices], y_train[indices]], [y_train[indices], X_train[indices]],
                      epochs=use_epochs, batch_size=32, verbose=1,
                      validation_data=[[X_val_new, y_val_new], [y_val_new, X_val_new]])
    for cycle in range(cycles):
        print("Cycle {}/{}".format(cycle + 1, cycles))
        indices = get_new_indices(indices, X_train, y_train, capsule_model, samples_per_round)
        stacked_model.fit([X_train[indices], y_train[indices]], [y_train[indices], X_train[indices]],
                          epochs=epochs, batch_size=32, verbose=1,
                          validation_data=[[X_val_new, y_val_new], [y_val_new, X_val_new]])

    return capsule_model
```

New Classifier – Transfer Layers

```
# Merge
X_train = np.concatenate((X_train, X_train_new), axis=0)
y_train = np.concatenate((y_train, y_train_new), axis=0)

# Input Layer
caps_input = Input(shape=X_train.shape[1:]) # 128 x 128 x 3
print(caps_input.shape)

# VGG Layers
vgg_layers.trainable = False
vgg_out = vgg_layers(caps_input)

# Transfer Layers
xfer_layers.trainable = True
xfer_out = xfer_layers(vgg_out)
```

New Classifier – Replacement Layers

```
# Primary Capsule Activation
caps_layer = Reshape([(32 * 6 * 6), 8])(xfer_out) # Conserve ~1,152 vectors of length 8?
caps_layer = Lambda(squash_function)(caps_layer)

# Capsule Layer
caps_layer = CapsuleLayer(y_train.shape[1], 16, 3)(caps_layer) # Output shape: [None, classes, 16]
caps_output = Length()(caps_layer)
capsule_model = Model(caps_input, caps_output)
capsule_vectors = Model(caps_input, caps_layer)

capsule_model.compile(optimizer=Adam(lr=0.001), loss=[margin_loss], metrics=['categorical_accuracy'])
capsule_model.summary()
```

New Classifier – Decoder and Stacked Models (again)

```
# Decoder Network
decoder_input = Input(shape=(y_train.shape[1], 16)) # Input shape: [classes, 16]
decoder_layer = Flatten()(decoder_input)
decoder_layer = Dense(512, activation='relu')(decoder_layer)
decoder_layer = Dense(1024, activation='relu')(decoder_layer)
decoder_layer = Dense(caps_input.shape[1] * caps_input.shape[2] * caps_input.shape[3])(decoder_layer)
decoder_output = Reshape(caps_input.shape[1:])(decoder_layer)
decoder_model = Model(decoder_input, decoder_output)

truth_input = Input(shape=(y_train.shape[1], ))

stacked_input = Input(caps_input.shape[1:])
stacked_layer = capsule_vectors(stacked_input)
capsule_output = Length()(stacked_layer)
stacked_layer = Mask()([stacked_layer, truth_input])
stacked_output = decoder_model(stacked_layer)

stacked_model = Model([stacked_input, truth_input], [capsule_output, stacked_output])

stacked_model.compile(optimizer='adadelta', loss=[margin_loss, 'mse'], metrics={'length_2': 'categorical_accuracy'})
```

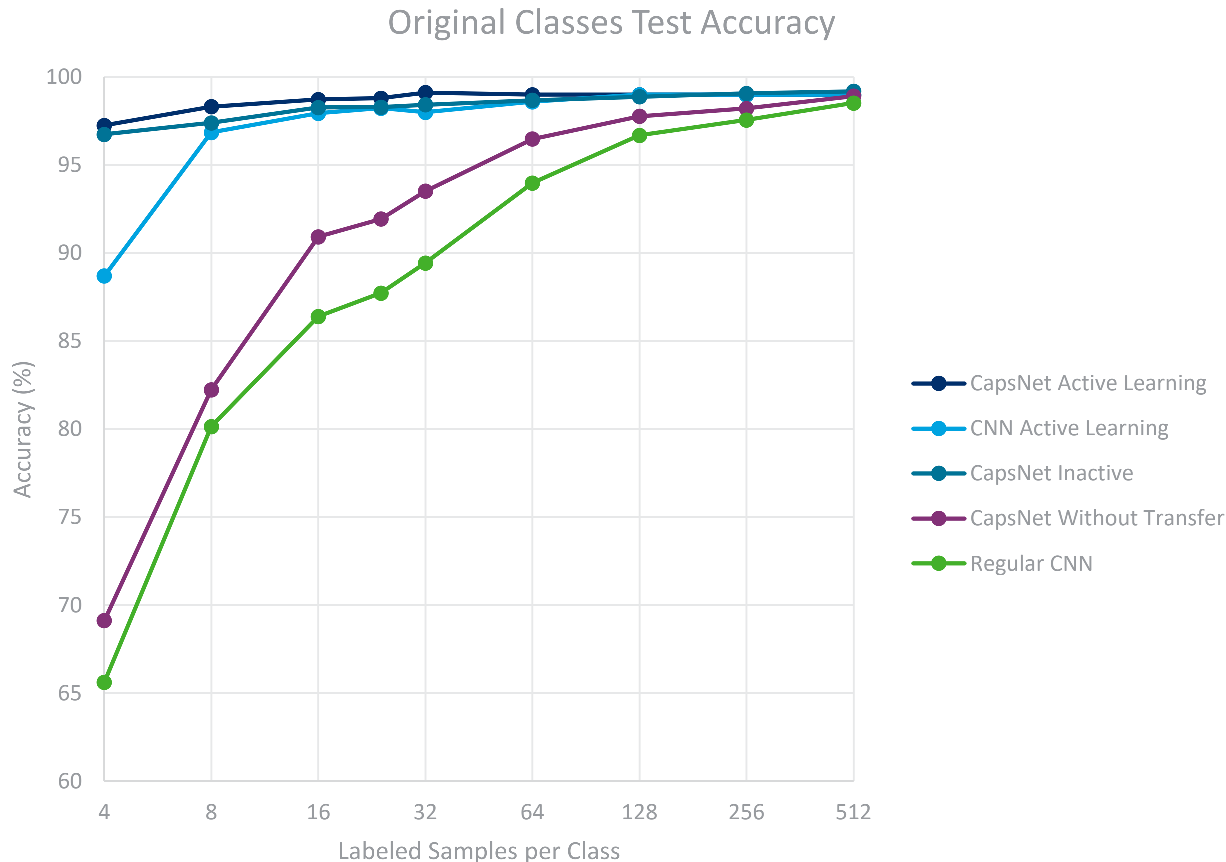
New Classifier – Active Learning Training

```
cycles = int((total_samples - initial_samples) / samples_per_round)
indices = get_starting_indices(y_train, initial_samples)

stacked_model.fit([X_train[indices], y_train[indices]], [y_train[indices], X_train[indices]],
                  epochs=epochs, batch_size=32, verbose=1,
                  validation_data=[[X_val_new, y_val_new], [y_val_new, X_val_new]])

for cycle in range(cycles):
    print("Cycle {}/{}".format(cycle + 1, cycles))
    indices = get_new_indices(indices, X_train, y_train, capsule_model, samples_per_round)
    stacked_model.fit([X_train[indices], y_train[indices]], [y_train[indices], X_train[indices]],
                      epochs=epochs, batch_size=32, verbose=1,
                      validation_data=[[X_val_new, y_val_new], [y_val_new, X_val_new]])
```

Results – MNIST with Limited Labels

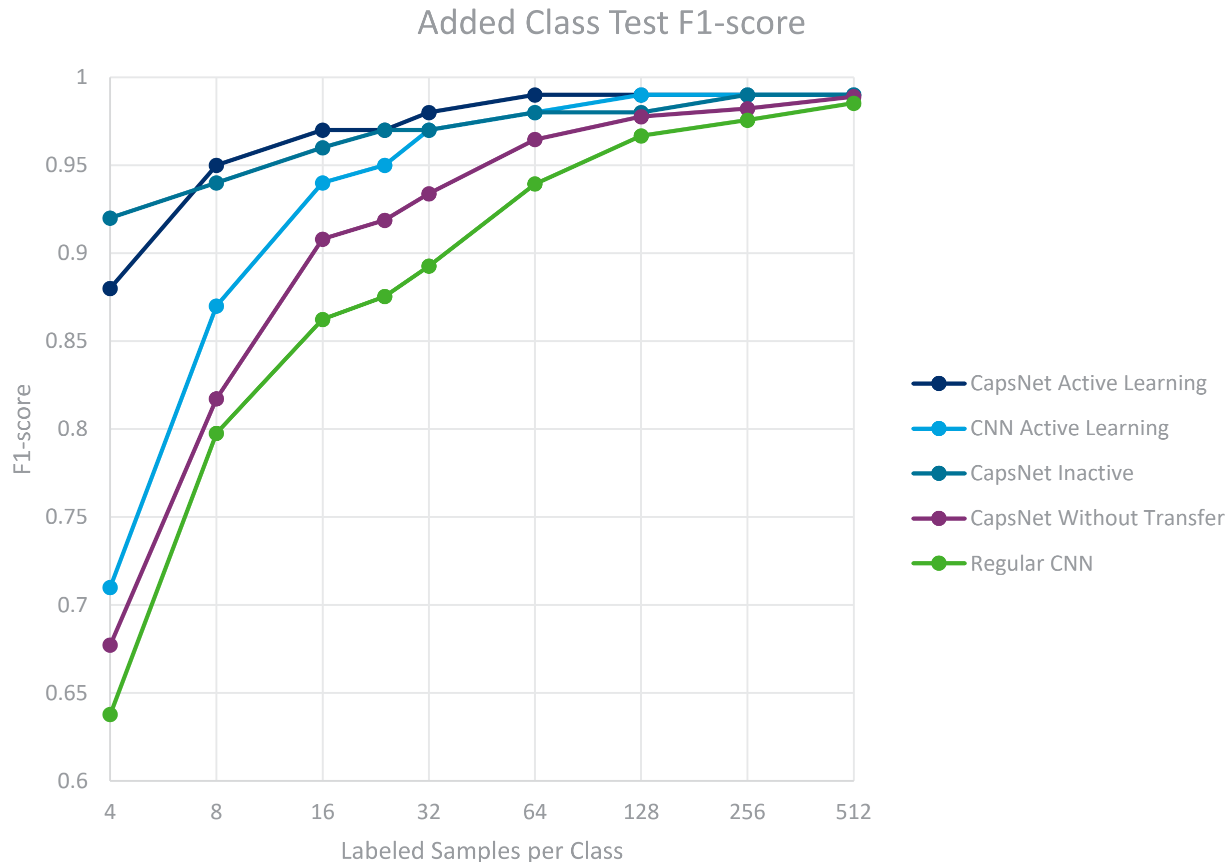
THE ADDED CLASS ACCURACY IS HIGHER FOR AN INACTIVE LEARNING CAPSNET AT THE LOWEST NUMBERS OF SAMPLES

Results – MNIST with Limited Labels



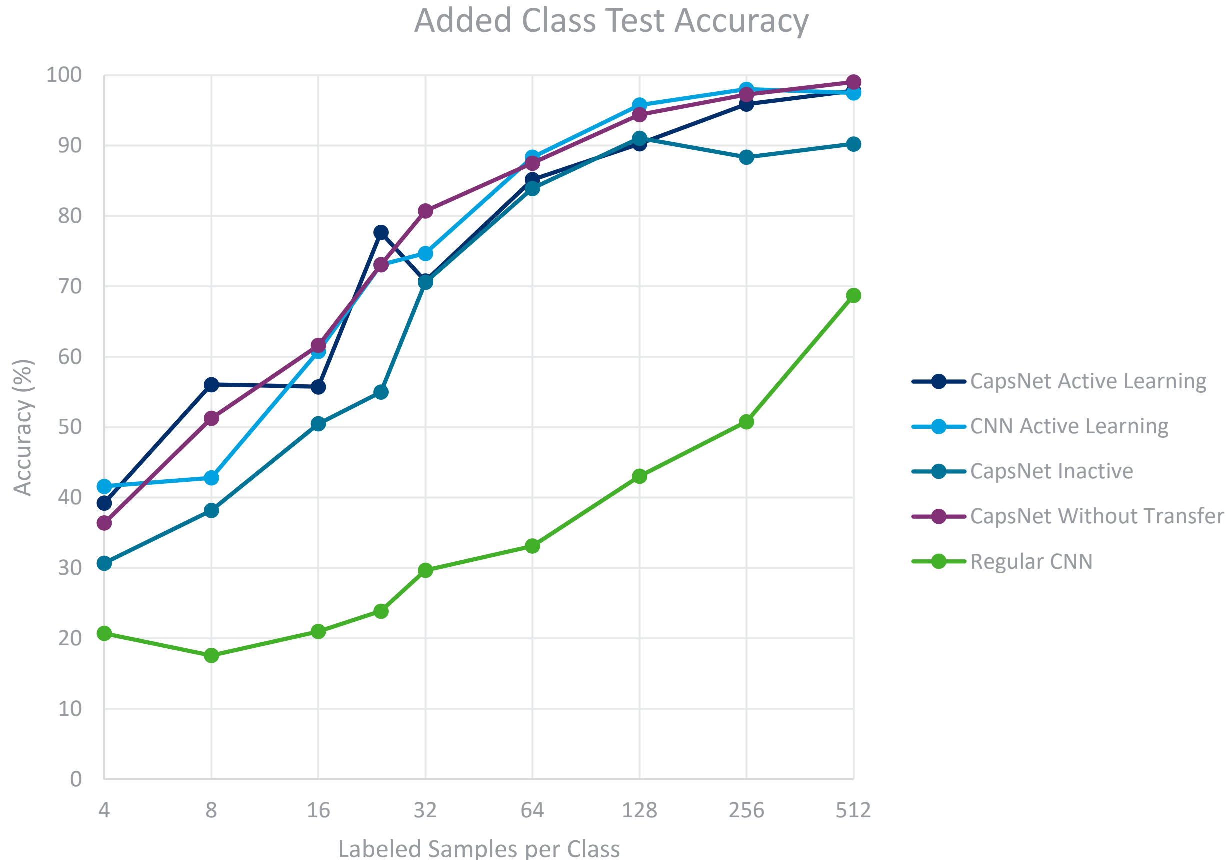
LOOKING AT THE ACCURACY OF THE ORIGINAL CLASSES REVEALS THE COMBINATION OF TECHNIQUES BEST PRESERVED THEIR PERFORMANCE

Results – MNIST with Limited Labels



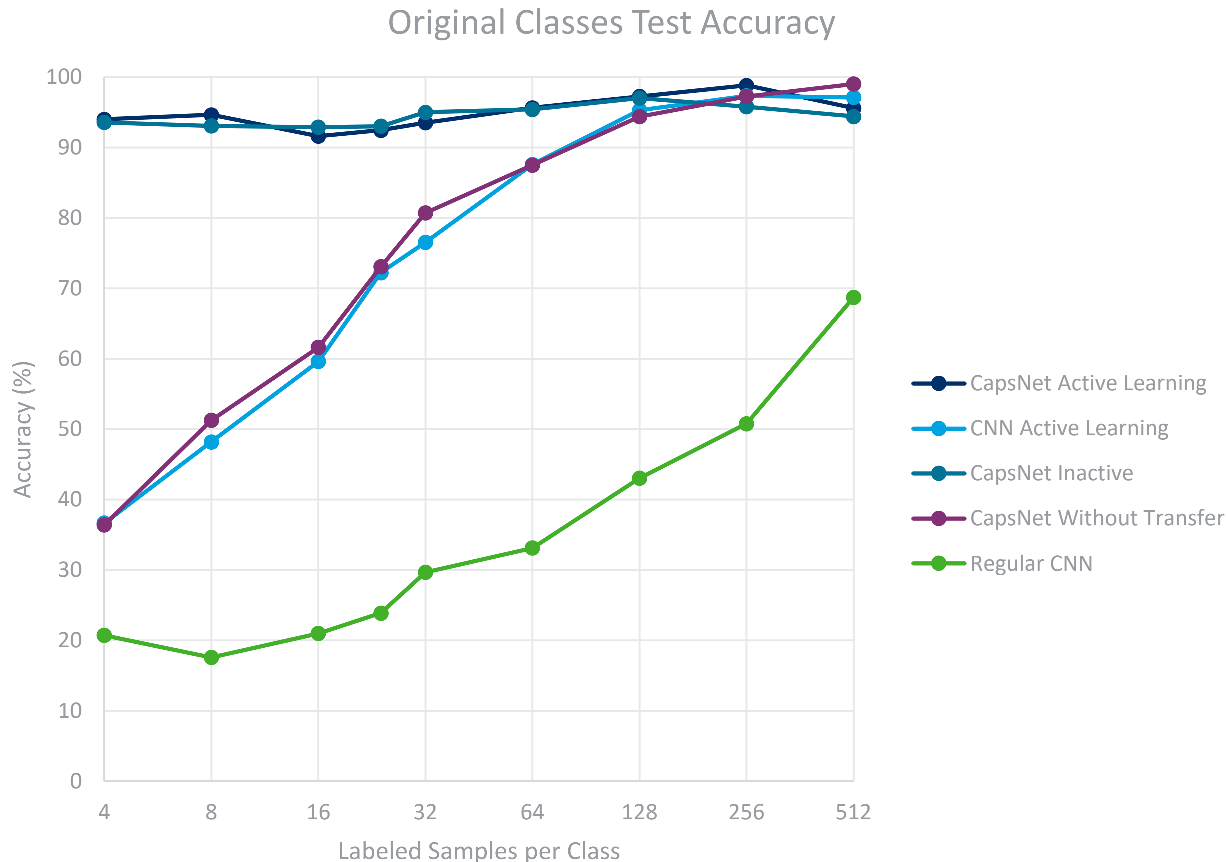
THE F1-SCORE OF THE ADDED CLASS SHOWS THAT THE IMPROVED RETENTION OF THE ORIGINAL CLASSES HAS A BIG IMPACT ON AVOIDING FALSE POSITIVES FOR THE NEW CLASS

Results – SENSIAC with Limited Labels



ON THIS MORE CHALLENGING DATASET, IT IS HARD TO SEE ANY PERCEIVED BENEFIT IN THE ACCURACY OF THE NEWLY ADDED CLASS

Results – SENSIAC with Limited Labels



HOWEVER, THE CAPSULE NETWORK COMBINED WITH TRANSFER LEARNING APPROACH DRAMATICALLY IMPROVES THE ACCURACY ON THE ORIGINAL CLASSES

Results – SENSIAC with Limited Labels

AGAIN, THE IMPROVED PERFORMANCE ON THE ORIGINAL CLASSES PRODUCES A NOTICEABLE IMPROVEMENT IN THE NEW CLASS'S ABILITY TO AVOID FALSE POSITIVES, WHICH IS REFLECTED IN THE F1-SCORE

Discussion

- Why is random selection about as good as active learning for these two datasets?
    - Both MNIST and this preparation of SENSIAC, have very little variation in “goodness” of the training samples
    - On a harder dataset, not shown, active learning outperformed the F1-score of the inactive CapsNet variant by 6% on average for sample sizes greater than 16. For sample sizes less than 16, the inactive CapsNet had a higher F1-score.
- Additional considerations
    - Active learning increases cycle time
    - Using a large numbers of capsules dramatically increases training time

Conclusions

- When is this useful?
    - When there is an existing network to transfer layers
    - When there is access to at least a limited set of images from the original training set
    - When the expertise to label many new samples is limited (time or money)
- When is this approach a waste?
    - If a large number of labeled samples are available, chances are a more conventional training process will yield better, or similar, results with significantly less time and effort.

Acknowledgements

John Haddon, Ph.D.

Principle Software Engineer

Funding Acquisition and Project Management

Tanner Lindbloom

Senior Software Engineer

Adapted Code from Original Internal Project and Generated Results for MNIST and SENSIAC

LOCKHEED MARTIN