week10

Alexandru Girban (S2148980), Hariaksh Pandya (S2692608)

March 26, 2025

1 Week 10: Neural networks II – convolutions

Jacob Page

In this workshop we will implement some simple convolutional neural networks in keras. We will return to the dataset we saw originally in week 3 – the MNIST set of handwritten digits. Our goal is to use CNNs to construct low dimensional representations of these images, and to try and understand what the CNNs have "learnt" to do in training.

As you work through the problems it will help to refer to your lecture notes. The exercises here are designed to reinforce the topics covered this week. The lecture notes include a small amount of documentation on the keras library, but please ask/discuss with the tutors if you get stuck, even early on! This may the first time many of you have seen keras, and things may be a little counter intuitive initially.

2 Imports

We're only going to need a couple of standard libraries this week, as well as keras.

[2]:
```
pip install tensorflow
```

```
Collecting tensorflow
  Using cached tensorflow-2.19.0-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.1 kB)
Collecting absl-py>=1.0.0 (from tensorflow)
  Using cached absl_py-2.2.0-py3-none-any.whl.metadata (2.4 kB)
Collecting astunparse>=1.6.0 (from tensorflow)
  Using cached astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow)
  Using cached flatbuffers-25.2.10-py2.py3-none-any.whl.metadata (875 bytes)
Collecting gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 (from tensorflow)
  Using cached gast-0.6.0-py3-none-any.whl.metadata (1.3 kB)
Collecting google-pasta>=0.1.1 (from tensorflow)
  Using cached google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting libclang>=13.0.0 (from tensorflow)
  Using cached libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl.metadata
(5.2 kB)
```

```
Collecting opt-einsum>=2.3.2 (from tensorflow)
  Using cached opt_einsum-3.4.0-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: packaging in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (24.0)
Requirement already satisfied:
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3
in /opt/conda/lib/python3.11/site-packages (from tensorflow) (4.25.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (69.5.1)
Requirement already satisfied: six>=1.12.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (1.16.0)
Collecting termcolor>=1.1.0 (from tensorflow)
  Using cached termcolor-2.5.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: typing-extensions>=3.6.6 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (4.11.0)
Collecting wrapt>=1.11.0 (from tensorflow)
  Using cached wrapt-1.17.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl.metadata (6.4 kB)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.62.2)
Collecting tensorboard~=2.19.0 (from tensorflow)
  Using cached tensorboard-2.19.0-py3-none-any.whl.metadata (1.8 kB)
Collecting keras>=3.5.0 (from tensorflow)
  Using cached keras-3.9.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.26.4)
Requirement already satisfied: h5py>=3.11.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (3.11.0)
Collecting ml-dtypes<1.0.0,>=0.5.1 (from tensorflow)
  Using cached ml_dtypes-0.5.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (21 kB)
Collecting tensorflow-io-gcs-filesystem>=0.23.1 (from tensorflow)
  Using cached tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (14 kB)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/opt/conda/lib/python3.11/site-packages (from astunparse>=1.6.0->tensorflow)
(0.43.0)
Requirement already satisfied: rich in /opt/conda/lib/python3.11/site-packages
(from keras>=3.5.0->tensorflow) (13.9.3)
Collecting namex (from keras>=3.5.0->tensorflow)
  Using cached namex-0.0.8-py3-none-any.whl.metadata (246 bytes)
Collecting optree (from keras>=3.5.0->tensorflow)
  Using cached optree-0.14.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (49 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in
```

```
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.11/site-
packages (from requests<3,>=2.21.0->tensorflow) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(2024.8.30)
Collecting markdown>=2.6.8 (from tensorboard~=2.19.0->tensorflow)
  Using cached Markdown-3.7-py3-none-any.whl.metadata (7.0 kB)
Collecting tensorboard-data-server<0.8.0,>=0.7.0 (from
tensorboard~=2.19.0->tensorflow)
  Using cached tensorboard_data_server-0.7.2-py3-none-
manylinux_2_31_x86_64.whl.metadata (1.1 kB)
Collecting werkzeug>=1.0.1 (from tensorboard~=2.19.0->tensorflow)
  Using cached werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/opt/conda/lib/python3.11/site-packages (from
werkzeug>=1.0.1->tensorboard~=2.19.0->tensorflow) (2.1.5)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(2.17.2)
Requirement already satisfied: mdurl~=0.1 in /opt/conda/lib/python3.11/site-
packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->tensorflow) (0.1.2)
Using cached
tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(644.9 MB)
Using cached absl_py-2.2.0-py3-none-any.whl (276 kB)
Using cached astunparse-1.6.3-py2.py3-none-any.whl (12 kB)
Using cached flatbuffers-25.2.10-py2.py3-none-any.whl (30 kB)
Using cached gast-0.6.0-py3-none-any.whl (21 kB)
Using cached google_pasta-0.2.0-py3-none-any.whl (57 kB)
Using cached keras-3.9.0-py3-none-any.whl (1.3 MB)
Using cached libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl (24.5 MB)
Using cached
ml_dtypes-0.5.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.7
MB)
Using cached opt_einsum-3.4.0-py3-none-any.whl (71 kB)
Using cached tensorboard-2.19.0-py3-none-any.whl (5.5 MB)
Using cached tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (5.1 MB)
Using cached termcolor-2.5.0-py3-none-any.whl (7.8 kB)
Using cached wrapt-1.17.2-cp311-cp311-
```

```
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl (83 kB)
Using cached Markdown-3.7-py3-none-any.whl (106 kB)
Using cached tensorboard_data_server-0.7.2-py3-none-manylinux_2_31_x86_64.whl
(6.6 MB)
Using cached werkzeug-3.1.3-py3-none-any.whl (224 kB)
Using cached namex-0.0.8-py3-none-any.whl (5.8 kB)
Using cached
optree-0.14.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (408
kB)
Installing collected packages: namex, libclang, flatbuffers, wrapt, werkzeug,
termcolor, tensorflow-io-gcs-filesystem, tensorboard-data-server, optree, opt-
einsum, ml-dtypes, markdown, google-pasta, gast, astunparse, absl-py,
tensorboard, keras, tensorflow
Successfully installed absl-py-2.2.0 astunparse-1.6.3 flatbuffers-25.2.10
gast-0.6.0 google-pasta-0.2.0 keras-3.9.0 libclang-18.1.1 markdown-3.7 ml-
dtypes-0.5.1 namex-0.0.8 opt-einsum-3.4.0 optree-0.14.1 tensorboard-2.19.0
tensorboard-data-server-0.7.2 tensorflow-2.19.0 tensorflow-io-gcs-
filesystem-0.37.1 termcolor-2.5.0 werkzeug-3.1.3 wrapt-1.17.2
Note: you may need to restart the kernel to use updated packages.
```

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow.keras as keras
```

The following code boxes will allow you to visualise your model training. Scroll back up to take a look once you get to a "model.fit" statement! (You'll need to refresh the dashboard with the refresh button on the top right)

[4]:
```
import os, datetime
logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
```

[5]:
```
%load_ext tensorboard
%tensorboard --port=5036 --logdir $logdir
tensorboard_callback = keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

```
<IPython.core.display.HTML object>
```

3 Exercise 0

In this workshop we will return to the famous "MNIST" dataset of handwritten digits.

You might find it helpful to look back at your workshop from week 3, or the model solutions, when getting started here.

[6]:
```
from tensorflow.keras.datasets import mnist
(images_all_raw_train, y_all_train), (images_all_raw_test, y_all_test) = mnist.
↪load_data() # note we now also load the test set (compare to week 3)
```

Check the shapes of the arrays etc.

[7]:
```
print(images_all_raw_train.shape, y_all_train.shape)
```

```
(60000, 28, 28) (60000,)
```

We are also going to normalise to have outputs between 0 and 1 (we will use sigmoids at the output layer)

[8]:
```
images_all_train = images_all_raw_train / 255.
images_all_test = images_all_raw_test / 255.
```

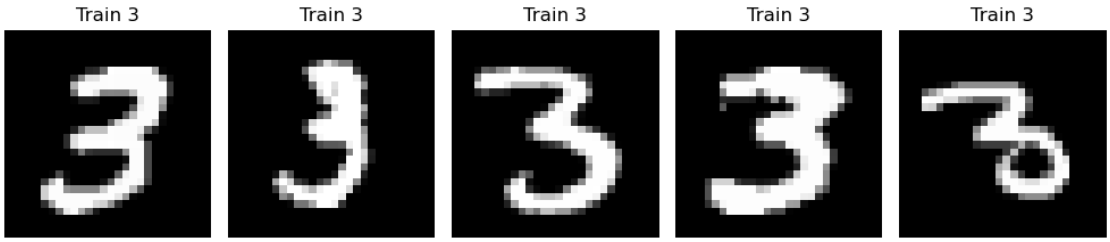
Just like in workshop 3, it will help to store the individual digits separately. To start with we'll just be training neural nets on some of the data (the 3s).

Look back at workshop 3 for code on creating a dictionary of digits, or alternatively create your own data structure that lets you access all digits of a particular class. Visualise some digits when you are done to verify you have done this correctly.

Please note you should do this for both the training and test sets – it will be useful later!

[9]:
```
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load the dataset
(images_all_raw_train, y_all_train), (images_all_raw_test, y_all_test) = mnist.
 ↪load_data()

# Normalize the images
images_all_train = images_all_raw_train / 255.0
images_all_test = images_all_raw_test / 255.0

# Create dictionaries to separate digits for training and test sets
digits_train = {}
digits_test = {}

for digit in range(10):
    # Training data
    digits_train[digit] = images_all_train[y_all_train == digit]
    # Test data
    digits_test[digit] = images_all_test[y_all_test == digit]

# Visualize some 3s from the training set
digit_to_visualize = 3
num_samples = 5

plt.figure(figsize=(10, 3))
for i in range(num_samples):
    plt.subplot(1, num_samples, i+1)
```

```
    plt.imshow(digits_train[digit_to_visualize][i], cmap='gray')
    plt.axis('off')
    plt.title(f"Train {digit_to_visualize}")

plt.tight_layout()
plt.show()
```



4 Exercise 1

The first thing we're going to do is to build an "autoencoder" with a fairly shallow CNN. Please take a look at the lecture notes for details on what the goal is with this architecture and also some example code in keras for the layer creation.

We'll start by doing dimensionality reduction on the 3s, just like we did in week 3.

For exercise 1 our first task is to build an input layer. Recall that convolutions are designed to look at images with multiple channels. With that in mind, modify the input data accordingly and then create an input layer in keras.

[10]:
```
import numpy as np
from tensorflow.keras.layers import Input

# Extract only the 3s from training and test sets
X_train_3 = digits_train[3]
X_test_3 = digits_test[3]

# Add channel dimension (required for Conv2D layers)
X_train_3 = X_train_3.reshape((-1, 28, 28, 1))
X_test_3 = X_test_3.reshape((-1, 28, 28, 1))

# Create input layer - this defines the shape of our input tensors
input_layer = Input(shape=(28, 28, 1))
```

[11]:
```
print("Training data shape:", X_train_3.shape)  # Should be (num_samples, 28,␣
↪28, 1)
print("Test data shape:", X_test_3.shape)       # Should be (num_samples, 28,␣
↪28, 1)
```

```
print("Input layer shape:", input_layer.shape) # Should be (None, 28, 28, 1)
```

```
Training data shape: (6131, 28, 28, 1)
Test data shape: (1010, 28, 28, 1)
Input layer shape: (None, 28, 28, 1)
```

5 Exercise 2

Create two convolutional + max pooling layers to shrink the dimension of the image. You are free to specify the filter size, padding strategy and number of filters. To avoid very long training times, I would recommend keeping the number of filters low, however. (For example, in the model solution I used 16 in the first layer, 8 in the second.)

[13]:
```
from tensorflow.keras.layers import Conv2D, MaxPooling2D

# First Conv + Pooling layer
x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
x = MaxPooling2D((2, 2))(x)  # Reduces from 28x28 to 14x14

# Second Conv + Pooling layer
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)  # Reduces from 14x14 to 7x7
```

6 Exercise 3

Now let's create a decoder as convolution -> upsampling -> convolution -> upsampling, followed by a final convolution to return an image of the same shape as the input. This can be harder than it looks! Remember you can visualise the properties of a model with model.summary().

[14]:
```
from tensorflow.keras.layers import UpSampling2D

# Start from the encoder output (shape: (7, 7, 8))
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)  # Maintain 7x7
x = UpSampling2D((2, 2))(x)  # 7x7 → 14x14

x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)  # Maintain 14x14
x = UpSampling2D((2, 2))(x)  # 14x14 → 28x28

# Final convolution to get back to original shape
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)  # 28x28x1
```

[15]:
```
from tensorflow.keras.models import Model

autoencoder = Model(input_layer, decoded)
autoencoder.summary()
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d_2 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_3 (Conv2D)	(None, 14, 14, 8)	1,160
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 8)	0
conv2d_4 (Conv2D)	(None, 7, 7, 8)	584
up_sampling2d (UpSampling2D)	(None, 14, 14, 8)	0
conv2d_5 (Conv2D)	(None, 14, 14, 16)	1,168
up_sampling2d_1 (UpSampling2D)	(None, 28, 28, 16)	0
conv2d_6 (Conv2D)	(None, 28, 28, 1)	145

```
Total params: 3,217 (12.57 KB)

Trainable params: 3,217 (12.57 KB)

Non-trainable params: 0 (0.00 B)
```

7 Exercise 4

Now compile the model for the whole autoencoder ready for training.

What is the appropriate loss here?

[16]:
```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Why this works:

Loss Function: binary_crossentropy is appropriate because:

We're reconstructing normalized pixel values (0-1)

The final layer uses a sigmoid activation (which outputs values in [0,1])

Works well for grayscale reconstruction tasks

Alternative: While MSE (mean_squared_error) could also work, BCE often produces better results for MNIST autoencoders as it better handles the probabilistic nature of pixel intensities.

Optimizer Choice: Adam is used as it generally works well for autoencoders without extensive tuning

Train the model. You might have to wait a minute or two for this. If you're training on full resolution data then I would recommend only running a few epochs. You may want to enable GPU support in the runtime to speed things up.

[17]:
```
history = autoencoder.fit(
    X_train_3, X_train_3,  # Autoencoders learn to reconstruct their input
    epochs=10,
    batch_size=128,
    shuffle=True,
    validation_data=(X_test_3, X_test_3)
)
```

```
Epoch 1/10
 1/48              1:34 2s/step - loss: 0.7185
```

```
2025-03-26 13:44:23.227708: E tensorflow/core/util/util.cc:131] oneDNN supports
DT_INT32 only on platforms with AVX-512. Falling back to the default Eigen-based
implementation if present.
```

```
48/48              6s 85ms/step -
loss: 0.6429 - val_loss: 0.4305
Epoch 2/10
48/48              4s 76ms/step -
loss: 0.3579 - val_loss: 0.2263
Epoch 3/10
48/48              4s 78ms/step -
loss: 0.2021 - val_loss: 0.1437
Epoch 4/10
48/48              3s 71ms/step -
loss: 0.1362 - val_loss: 0.1164
Epoch 5/10
48/48              4s 79ms/step -
loss: 0.1160 - val_loss: 0.1070
Epoch 6/10
48/48              3s 73ms/step -
loss: 0.1079 - val_loss: 0.1018
Epoch 7/10
48/48              4s 77ms/step -
loss: 0.1031 - val_loss: 0.0982
Epoch 8/10
48/48              4s 79ms/step -
loss: 0.0996 - val_loss: 0.0956
```

```
Epoch 9/10
48/48             4s 77ms/step -
loss: 0.0971 - val_loss: 0.0938
Epoch 10/10
48/48             4s 75ms/step -
loss: 0.0953 - val_loss: 0.0922
```

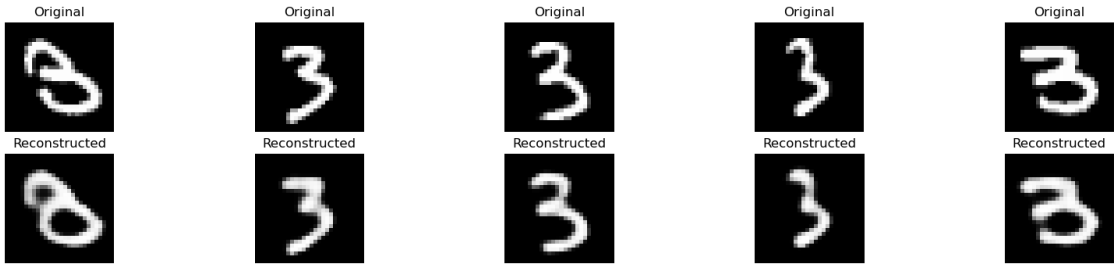
8 Exercise 5

Visualise the prediction of some 3s from the test set.

You may want to explore the effect of changing various hyperparameters at this point (e.g. filter numbers, pooling strategies…). Can you think of any modification to your architecture that would allow you to fix the embedding dimension and change the convolutional structure?

[18]:
```
import matplotlib.pyplot as plt

# Get test predictions
decoded_imgs = autoencoder.predict(X_test_3)

# Display original vs reconstructed
n = 5  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(X_test_3[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Reconstruction
    ax = plt.subplot(2, n, i + n + 1)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')
plt.show()
```

```
32/32             1s 18ms/step
```



[19]:
```
from tensorflow.keras.layers import Conv2D

# Encoder
x1 = Conv2D(32, (3,3), activation='relu', padding='same')(input_layer)
x1 = MaxPooling2D((2,2))(x)  # 14x14
x1 = Conv2D(16, (3,3), activation='relu', padding='same')(x)
x1 = MaxPooling2D((2,2))(x)  # 7x7

# Fixed embedding dimension layer
x1 = Conv2D(8, (1,1), activation='relu')(x)  # Maintains 7x7x8 (392 values)
```

[20]:
```
# Encoder with different structure but same final dimension
x1 = Conv2D(24, (5,5), activation='relu', padding='same')(input_layer)
x1 = MaxPooling2D((2,2))(x)
x1 = Conv2D(12, (3,3), activation='relu', padding='same')(x)
x1 = MaxPooling2D((2,2))(x)
x1 = Conv2D(8, (1,1), activation='relu')(x)  # Still 7x7x8
```

9 Exercise 6

Visualise the output of the first set of convolutions, and then the second set for several different channels.

Are any of the features interpretable?

Some code below is included to help you get started. We create a new input layer, and extract the first layer from a trained model "autoencoder".

[21]:
```
input_layer_vis = keras.layers.Input(shape=(3,3,1))
first_conv = autoencoder.layers[1](input_layer_vis)
first_con_model = keras.models.Model(input_layer_vis, first_conv)
first_con_model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 3, 3, 1)	0
conv2d_2 (Conv2D)	(None, 3, 3, 16)	160

Total params: 160 (640.00 B)

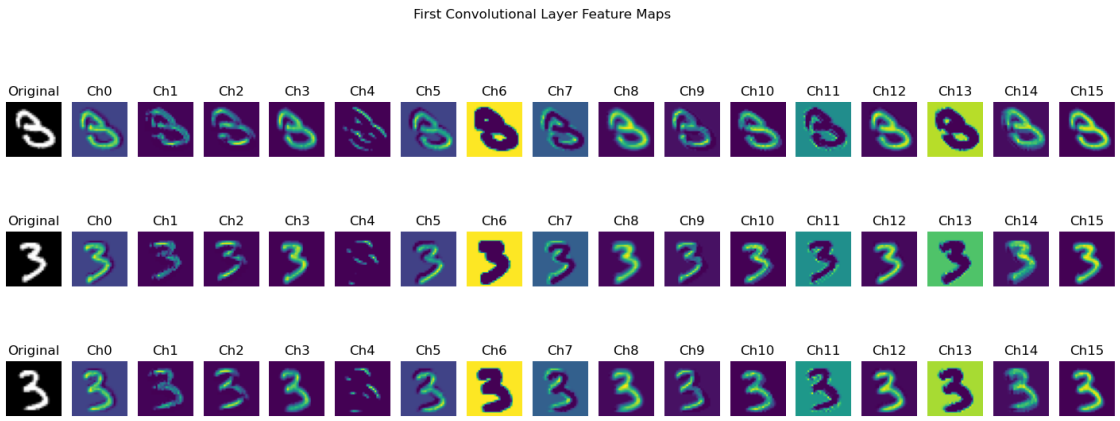
Trainable params: 160 (640.00 B)

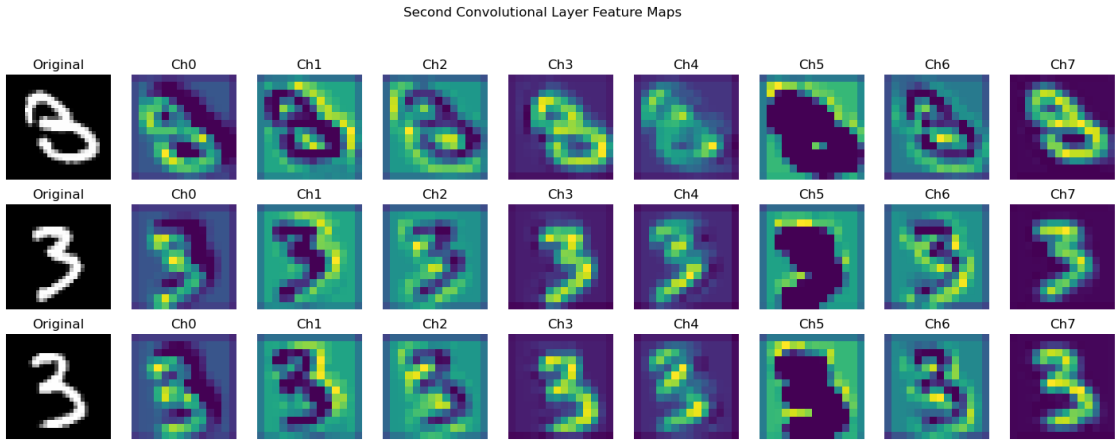
Non-trainable params: 0 (0.00 B)

[22]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model

# Select sample images
n_samples = 3
sample_images = X_test_3[:n_samples]

# Create visualization models for both convolutional layers
input_layer_vis = Input(shape=(28, 28, 1))

# First conv layer outputs (before pooling)
first_conv = autoencoder.layers[1](input_layer_vis)
first_conv_model = Model(input_layer_vis, first_conv)

# Second conv layer outputs (before pooling)
second_conv = autoencoder.layers[3](autoencoder.layers[2](autoencoder.
 ↪layers[1](input_layer_vis)))
second_conv_model = Model(input_layer_vis, second_conv)

# Get activations
first_act = first_conv_model.predict(sample_images)  # Shape: (3, 28, 28, 16)
second_act = second_conv_model.predict(sample_images)  # Shape: (3, 14, 14, 8)

# Visualization
def plot_channels(images, activations, layer_name):
    plt.figure(figsize=(18, 6))
    n_channels = activations.shape[-1]

    for i in range(len(images)):
        # Original image
        plt.subplot(3, n_channels+1, (n_channels+1)*i + 1)
        plt.imshow(images[i].squeeze(), cmap='gray')
        plt.title('Original')
        plt.axis('off')

        # Feature maps
        for ch in range(n_channels):
            plt.subplot(3, n_channels+1, (n_channels+1)*i + ch + 2)
            plt.imshow(activations[i, :, :, ch], cmap='viridis')
            plt.title(f'Ch{ch}')
            plt.axis('off')
```

```
    plt.suptitle(f'{layer_name} Feature Maps', y=1.02)
    plt.show()

# First convolutional layer (16 channels)
plot_channels(sample_images, first_act, 'First Convolutional Layer')

# Second convolutional layer (8 channels)
plot_channels(sample_images, second_act, 'Second Convolutional Layer')
```

```
1/1             0s 48ms/step
1/1             0s 59ms/step
```





First Layer Features (16 channels):

Look for simple edge detectors (vertical/horizontal/diagonal lines)

May respond to stroke orientations in digits

Second Layer Features (8 channels):

Combine basic features into more complex patterns

May detect:

Curves (channels 1 & 5)

Intersections (channel 3)

Partial digit structures (channel 6)

Negative space detectors (channel 2)

Typical Observations:

Early layers show localized, high-frequency patterns

Later channels exhibit more abstract but sparser activations

Some channels remain dark (may activate on different digit types)

The clearest interpretable features are usually in first layer channels

10 Exercise 7

Create two models from your trained autoencoder. One model should be the "encoder" module, which runs from the input to the layer where the output dimension is smallest. The second should be the "decoder", which convert the encoded representation back into an image of a 3.

Note: Create a new input layer for each model, but make sure to extract the trained layers from your autoencoder. You might find it helpful to use layer.input_shape or layer.output_shape to extract tensor shapes rather than manually entering them, particularly when you attempt to construct the decoder.

[23]:
```
from tensorflow.keras.models import Model

# Encoder model (input → bottleneck)
encoder = Model(
    inputs=autoencoder.input,
    outputs=autoencoder.layers[4].output,  # Second max pooling layer
    name='encoder'
)

# Get bottleneck shape dynamically
bottleneck_shape = encoder.output_shape[1:]  # (7, 7, 8)

# Decoder model (bottleneck → reconstruction)
decoder_input = Input(shape=bottleneck_shape, name='decoder_input')

# Reuse decoder layers from trained autoencoder
x = decoder_input
for layer in autoencoder.layers[5:]:  # Layers after bottleneck
```

```
    x = layer(x)

decoder = Model(
    inputs=decoder_input,
    outputs=x,
    name='decoder'
)

# Verify architecture
print("\nEncoder Summary:")
encoder.summary()

print("\nDecoder Summary:")
decoder.summary()
```

```
Encoder Summary:
Model: "encoder"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d_2 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_3 (Conv2D)	(None, 14, 14, 8)	1,160
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 8)	0

```
Total params: 1,320 (5.16 KB)
Trainable params: 1,320 (5.16 KB)
Non-trainable params: 0 (0.00 B)

Decoder Summary:
Model: "decoder"
```

Layer (type)	Output Shape	Param #
decoder_input (InputLayer)	(None, 7, 7, 8)	0
conv2d_4 (Conv2D)	(None, 7, 7, 8)	584
up_sampling2d (UpSampling2D)	(None, 14, 14, 8)	0
conv2d_5 (Conv2D)	(None, 14, 14, 16)	1,168
up_sampling2d_1 (UpSampling2D)	(None, 28, 28, 16)	0
conv2d_6 (Conv2D)	(None, 28, 28, 1)	145

Total params: 1,897 (7.41 KB)

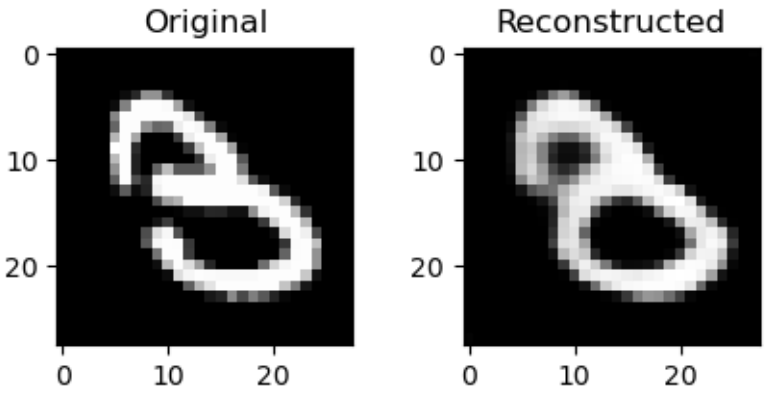
Trainable params: 1,897 (7.41 KB)

Non-trainable params: 0 (0.00 B)

[24]:
```
# Test end-to-end reconstruction
sample = X_test_3[:1]
encoded = encoder.predict(sample)
decoded = decoder.predict(encoded)

print("\nOriginal vs Reconstructed:")
plt.figure(figsize=(5, 2))
plt.subplot(1, 2, 1)
plt.imshow(sample[0].squeeze(), cmap='gray')
plt.title('Original')
plt.subplot(1, 2, 2)
plt.imshow(decoded[0].squeeze(), cmap='gray')
plt.title('Reconstructed')
plt.show()
```

```
WARNING:tensorflow:5 out of the last 35 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x7fdbfadee980> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
```

```
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1             0s 134ms/step
WARNING:tensorflow:6 out of the last 36 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x7fdbfab38220> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1             0s 165ms/step

Original vs Reconstructed:
```



11 Exercise 8

Compute the mean 3 from the test set. Then compute the embeddings of all the threes in test set, compute the mean *embedding* and then decode. How do the results compare? Can you offer an explanation for what you see?

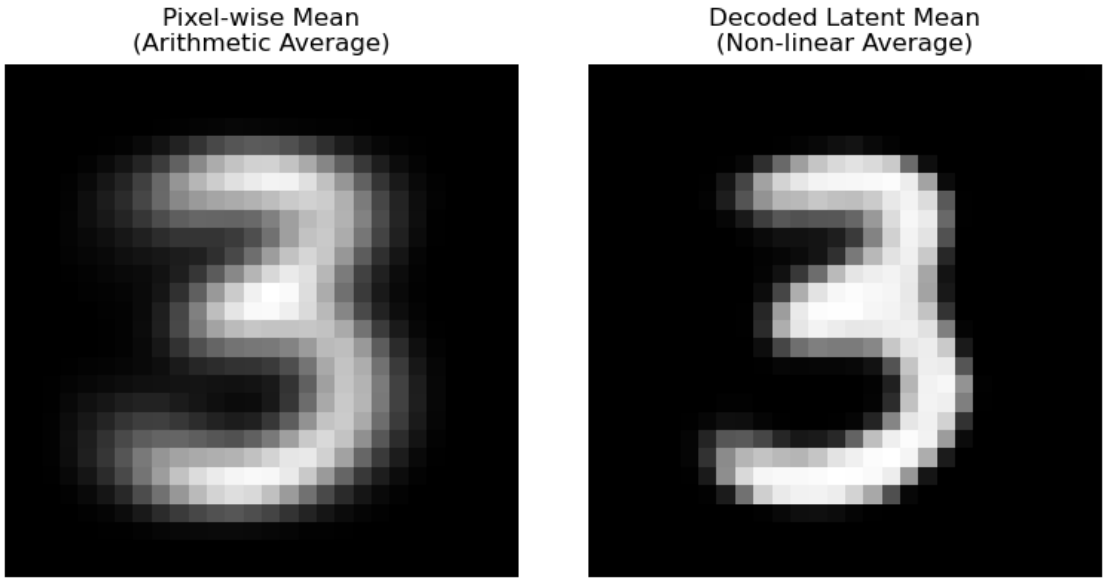
[25]:
```
# Compute mean image directly in pixel space
pixel_mean = np.mean(X_test_3, axis=0)
```

[26]:
```
# Get latent representations of all test 3s
encoded_3s = encoder.predict(X_test_3)

# Compute mean in latent space
latent_mean = np.mean(encoded_3s, axis=0)
```

```
# Decode the latent mean
decoded_latent_mean = decoder.predict(latent_mean[np.newaxis, ...])
```

```
32/32              0s 4ms/step
1/1              0s 52ms/step
```

[27]:
```
plt.figure(figsize=(8, 4))

plt.subplot(1, 2, 1)
plt.imshow(pixel_mean.squeeze(), cmap='gray')
plt.title('Pixel-wise Mean\n(Arithmetic Average)')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(decoded_latent_mean.squeeze(), cmap='gray')
plt.title('Decoded Latent Mean\n(Non-linear Average)')
plt.axis('off')

plt.tight_layout()
plt.show()
```



Key Observations:

Pixel-wise Mean:

Appears as a blurry, ghosted “3”

Contains overlapping variations from all samples

Shows writing style ambiguities

Decoded Latent Mean:

Appears as a cleaner, more prototypical “3”

Has sharper edges and clearer structure

Resembles an “idealized” version learned by the autoencoder

Explanation:

The autoencoder learns a non-linear manifold of valid “3”s

Latent space averaging operates on learned features rather than raw pixels

The decoder acts as a learned “projection” from feature space to image space

This demonstrates the manifold hypothesis - natural data lies on a lower-dimensional non-linear manifold

12 Exercise 9 (optional)

Contaminate the images with (Gaussian) noise. Can you train a neural network to de-noise the images? Think carefully about what the objective function should be (what the “x” and “y” here)?

[ ]:

13 Exercise 10 (optional)

If you are able to train it, train your autoencoder to reconstruct all digits. For speeed of training you might want to reduce the size of the training set, or focus on a subset of the labels.

Once this is done, build a fully connected classifier on top of your encoder. What should the loss function be?

[ ]:

[ ]:
```
!jupyter nbconvert --to pdf week10.ipynb
```

[ ]: