Boring but important disclaimers:

- If you are not getting this from the GitHub repository or the associated Canvas page (e.g. CourseHero, Chegg etc.), you are probably getting the substandard version of these slides Don't pay money for those, because you can get the most updated version for free at

  `https://github.com/julianmak/OCES5303_ML_ocean`

  The repository principally contains the compiled products rather than the source for size reasons.
- Associated Python code (as Jupyter notebooks mostly) will be held on the same repository. The source data however might be big, so I am going to be naughty and possibly just refer you to where you might get the data if that is the case (e.g. JRA-55 data). I know I should make properly reproducible binders etc., but I didn't...
- I do not claim the compiled products and/or code are completely mistake free (e.g. I know I don't write Pythonic code). Use the material however you like, but use it at your own risk.
- As said on the repository, I have tried to honestly use content that is self made, open source or explicitly open for fair use, and citations should be there. If however you are the copyright holder and you want the material taken down, please flag up the issue accordingly and I will happily try and swap out the relevant material.

OCES 5303 :
ML methods in Ocean Sciences

Session 5: CNNs and more PyTorch

Outline

- convolutions

  → kernels and pooling

  → resizing issues

  → examples

- Convolutional Neural Networks (CNNs)

  → TL;DR: kernel **weights** as part of control variables

  → Keras and more PyTorch functionalities (e.g. `DataLoader`)

  → classification and regression demonstration

Figure: Convolved ad-hoc TA.

Oceanic application

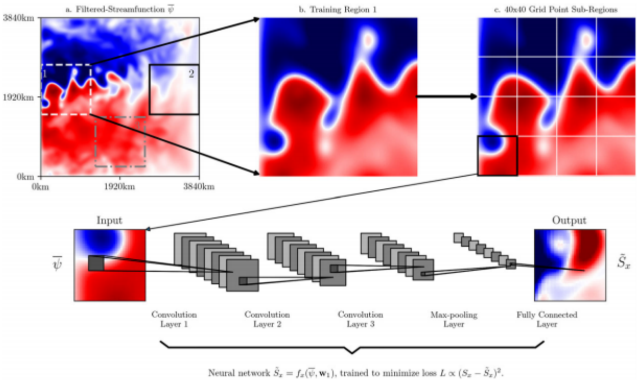

Figure: CNN applied to an eddy parameterisation problem. From Bolton & Zanna (2019).

- eddy parameterisation problem
  $\rightarrow$ regression: predict one image from another with CNNs

Oceanic application

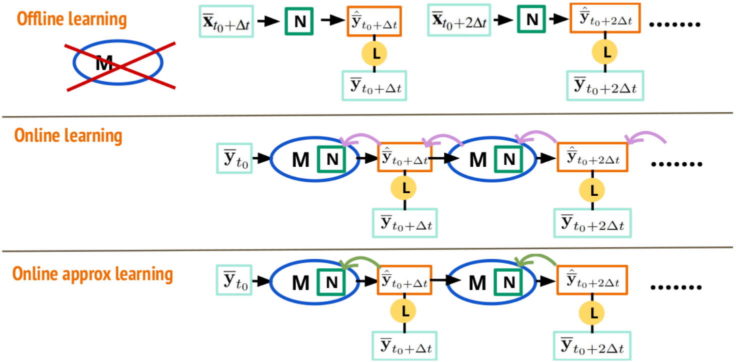

Figure: CNN training nested inside an eddy parameterisation problem. From Yan *et al.* (2025).

- as before, but nesting a CNN <u>within</u> a model (cf. RNN next time)

  $\to$ related to variational data assimilation approaches

CNNs

- Convolutional Neural Networks (CNNs)

  $\rightarrow$ has slightly better control on number of degrees of freedom compared to fully connected MLPs

  $\rightarrow$ naturally suited to images

CNNs

- Convolutional Neural Networks (CNNs)

  → has slightly better control on number of degrees of freedom compared to fully connected MLPs

  → naturally suited to images

- should be self-explanatory why after going through what convolutions are...

  → recall (e.g. from OCES 3301) convolutions are defined as:

$$(f * G)(x) = \int f(x')G(x, x')\, \mathrm{d}x'$$

  → for time-series data we have (t, τ) instead of (x, x'), related to filtering

  → easier in the discrete case with an example...

Convolutions

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \qquad \boxed{\begin{matrix} a & b \\ c & d \end{matrix}}$$

Conv2d

Figure: Sample array to be convolved with a kernel (the red stuff). Kernel here is of size (2,2).

Convolutions

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \xrightarrow[\text{Conv2d}]{\boxed{\begin{matrix} a & b \\ c & d \end{matrix}}} 3.5$$

$$a = b = c = d = 1/4$$

Figure: Choose a kernel, throw down kernel, element-wise multiplication, then sum.

Convolutions

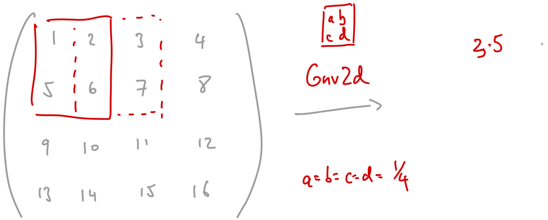

Figure: Move the kernel with some stride (1 here), then continue same process.

Convolutions

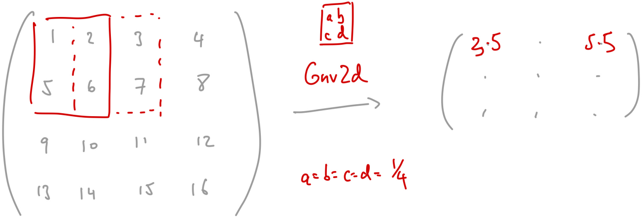

Figure: Repeat until whole array is done. By default convolution leads to an array with **reduced** the size.

Convolutions

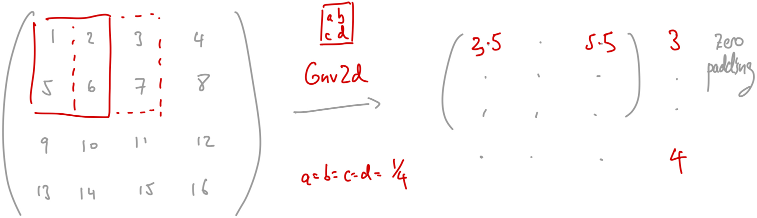

Figure: Can do padding for various reasons. Zero padding here bulks the **input** array size.

Pooling

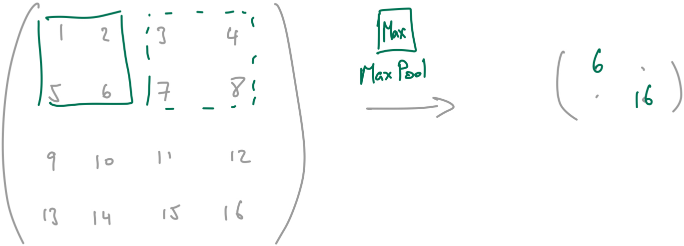

Figure: Pooling is similar, but default has stride equal to kernel size. This also reduces size of array.

- all the above can in principle be done for non-square kernels/poolings and/or unequal strides, paddings (and dilations; look that up if you want)

Convolution example

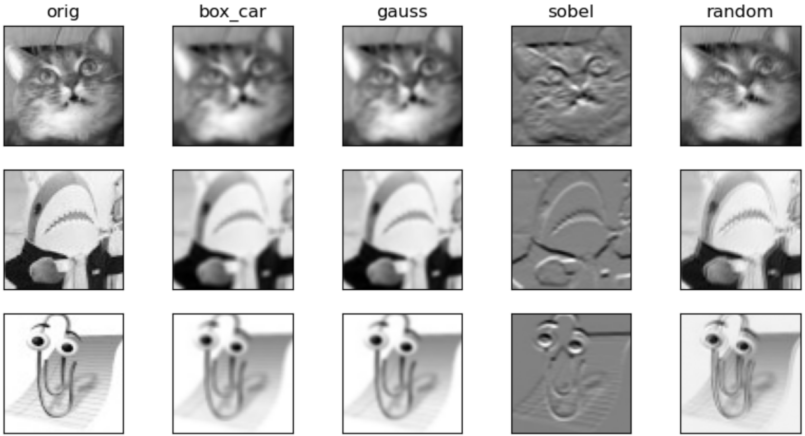

Figure: Convolving the ad-hoc TAs with a size 3 kernel with stride 1 with different choices of kernel weights. Input size is (64, 64) and output size in this case is (62, 62) (or slice 3-1 = 2 pixels off).

Pooling example

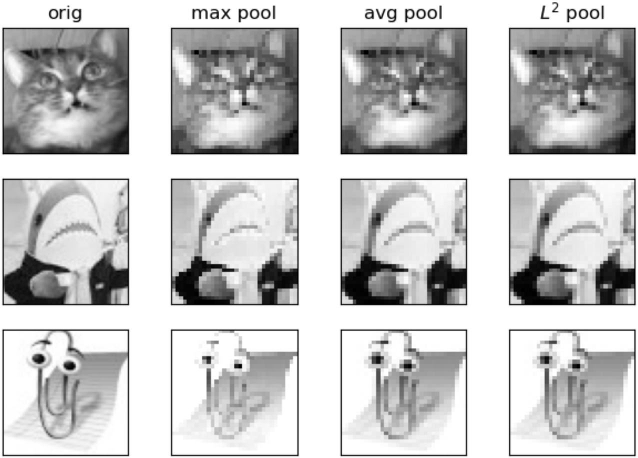

Figure: Pooling the ad-hoc TAs with a size 2 kernel with stride 2 with different operations. Input size is (64, 64) and output size in this case is (31, 31) (or divide by 2 in this case).

CNNs

- CNNs include **kernel weights** as control variables

  → otherwise proceed entirely as before
- controls degrees of freedom a bit better

  → number of kernel weights

  → reduction in array size

  → linear layer(s) as before

```
    # convolutional layer 1 & max pool layer 1
    self.layer1 = nn.Sequential(
        nn.Conv2d(1, 6, 5),     # now (6, 60, 60)
        nn.ReLU(),
        nn.MaxPool2d((2, 2)),  # now (6, 30, 30)
    )

    # convolutional layer 2 & max pool layer 2
    self.layer2 = nn.Sequential(
        nn.Conv2d(6, 10, 5),   # now (10, 26, 26)
        nn.ReLU(),
        nn.MaxPool2d((2, 2)),  # now (10, 13, 13)
    )

    self.layer3 = nn.Sequential(
        nn.Linear(10 * 13 * 13, 60),
        nn.ReLU(),
        nn.Linear(60, 30),
        nn.ReLU(),
        nn.Linear(30, 2)      # because two possible
    )

def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.view(-1, 10 * 13 * 13)
    out = self.layer3(out)
    return out
```

Need some care in specifying sizes in the network structure!

CNNs: classification

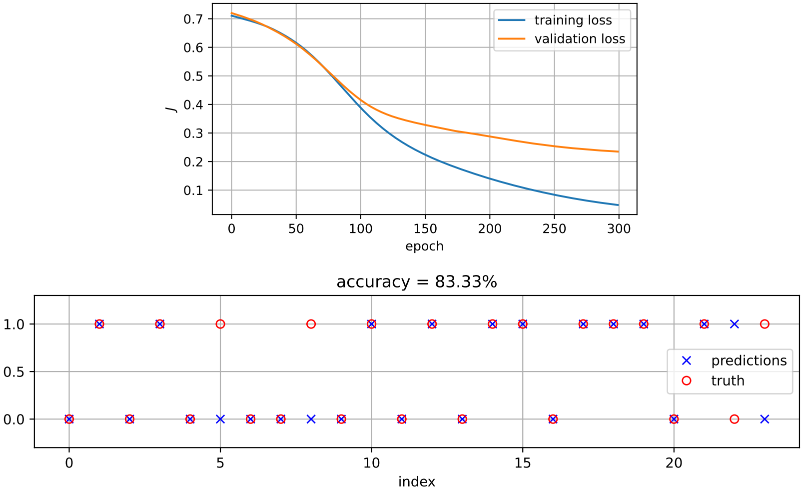

Figure: CNN for classification. Probably hasn't converged, but skill seems ok.

CNNs: regression

- need a change in the loss
  $\rightarrow$ use `MSELoss`
- need to change the CNN structure
  $\rightarrow$ linear layer to expand back to image array

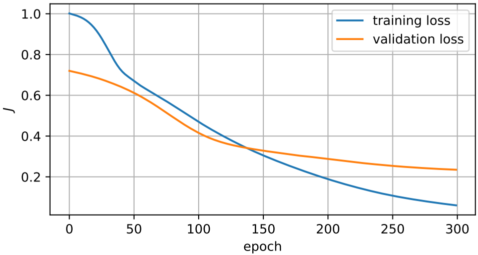

Figure: CNN for regression (predict bottom from top half).

CNNs: regression

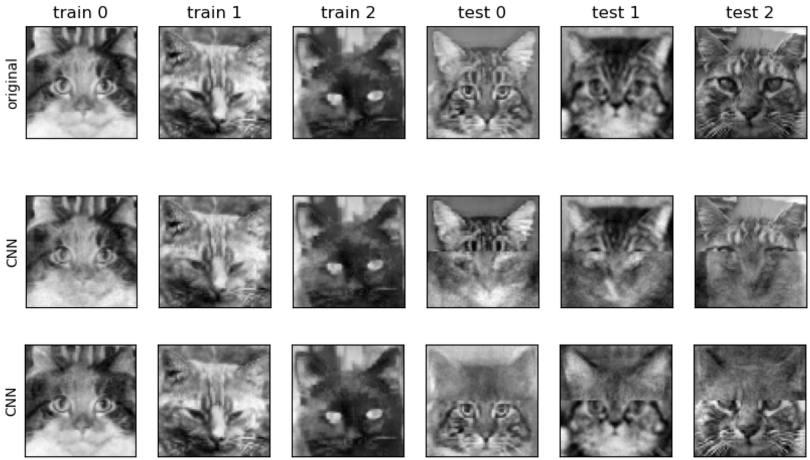

Figure: CNN for regression (predict bottom from top half and vice-versa).

Use of PyTorch `DataLoader`

- can package the data a bit better with a `Dataset` object
  - → packages in and out data together
  - → need `__init__`, `__len__` and `__getitem__`
  - → write in where to get item and relevant transformations

```python
# define custom dataset to do allow for batching
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):

    def __init__(self, in_tensor, out_tensor):
        self.inp = in_tensor
        self.out = out_tensor

    def __len__(self):
        return len(self.inp)

    def __getitem__(self, idx):
        return self.inp[idx], self.out[idx]
```

Figure: Very basic `Dataset` object, with just in and out tensors.

Use of PyTorch `DataLoader`

- `Dataset` object can then be passed to `DataLoader`

  → data can in principle be handled 'lazily' (not loading into memory), particularly useful when data is read from file and file is really big

- can specify **batch size** for batching

  → gradient computation and model updates in batches

  → batching slows down training per epoch...

  → ...but model might then need fewer epoches for training

  → can lead to improvements in robustness and model skill

!!! extra hyper-parameter

Use of PyTorch DataLoader

- modification of training loop

```
# define the dataloader with the
train_dataset = MyDataset(X_train, Y_train)
train_dataloader = DataLoader(train_dataset,
                              batch_size=batch_size,
                              shuffle=True)

for epoch in range(num_epochs):

    # reset the running loss each epoch
    running_loss = 0.0

    # iteration step (if full batch then below for loop only runs once anyway)
    for batch_X, batch_Y in train_dataloader:
        model.train()  # put the model in training mode (taping is on)
        optimizer.zero_grad()  # clear gradients if it exists (from loss.backward())
        Y_pred = model(batch_X)  # feed-forward
        J_train = J(Y_pred, batch_Y) # compute loss
        J_train.backward()  # back propagation
        optimizer.step()  # iterate
```

```
model, train_J, test_J = training_batch(model, optimizer, J,
                                        X_train, Y_train,
                                        X_valid, Y_valid,
                                        batch_size=12,  # new argument
                                        num_epochs=300, out_epoch=20)
```

Figure: Modified training loop. I've chosen to define the `DataLoader` in the training loop itself, but other structures are possible.

Piping through Keras

- Keras you can think of as an abstraction layer on top of the PyTorch/TensorFlow/JAX engines

  → makes a few things easier/cleaner to do

  → provides some useful interfaces

  → some details you still need to do yourself though (e.g. NN structures and parameters)

```python
import os
os.environ["KERAS_BACKEND"] = "torch"  # use PyTorch as backend

import keras
import keras.layers as layers

# force a clean keras session (clears models etc.)
keras.backend.clear_session()
```

Figure: Forcing keras to use PyTorch engine (default is TensorFlow), and calling a fresh session (clears all models, taping etc.).

Piping through Keras

- below is the equivalent CNN used for regression above
  $\to$ taping of operations as usual like PyTorch
  $\to$ the layer values you still need to specify though
  $\to$ be careful of image data dimension ordering (see notebook)

```
input_shape = (64, 32, 1)

model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(6, kernel_size=(5, 5), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(10, kernel_size=(5, 5), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(1000, activation="relu"),
        layers.Dense(64*32),
    ],
    name = "CNN regression"  # give the model a name if you want
)

# prints out a summary of the model
model.summary()
```

Figure: Marginally (?) cleaner interface for defining neural network structure.

Piping through Keras

- specify losses and optimisers as usual
- fitting is easier (there is a default training loop)

  → also don't need to specify whether `model` is in training or prediction mode

```
learning_rate = 0.0001

model.compile(loss=keras.losses.MeanSquaredError(),
              optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
              # metrics=[keras.metrics.Accuracy()],
             )
```

```
train_log = model.fit(train_dataloader,
                      epochs=15);
```

```
Epoch 1/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 170ms/step - loss: 1.0353
Epoch 2/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 319ms/step - loss: 1.0273
Epoch 3/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 246ms/step - loss: 1.0203
Epoch 4/15
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 227ms/step - loss: 1.0141
```

Figure: Compiling and fitting the model.

Demonstration

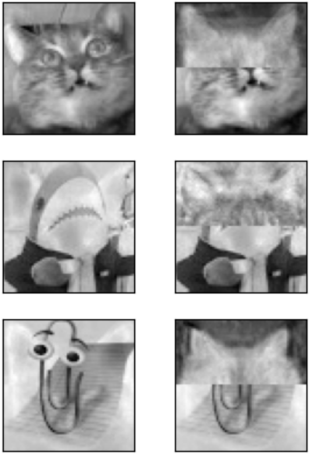

Figure: Grant us eyes (or not).

- introduced and built a CNN

  → convolutions and basics with layer structure etc.

  → piping through PyTorch `Datset`, `DataLoader` and keras

  → introduced **batching**

  → need hyper-parameter tuning and cross-validation etc.

Moving to a Jupyter notebook →