

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 4: Neural Networks, PyTorch, and MLPs

Outline

- ▶ anatomy of a **neural network**
 - nodes, weights, hidden layers, activation functions
 - back-propagation
- ▶ **perceptrons**
 - simple perceptrons and interface with linear models
- ▶ **multi-layer perceptrons (MLPs)**
 - set up in **PyTorch** (could actually be done in `sklearn`)
 - solvers, loss curves, various options
- ▶ cats and dogs example: classification and regression

Oceanic application

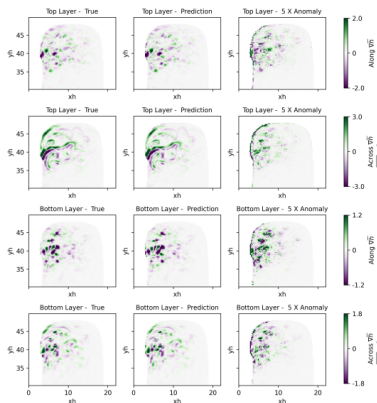


Figure: Model truth, predicted and mismatches in the predicted thickness fluxes from a double gyre calculation. From Balwada *et al.* (submitted).

- ▶ many instances where **neural networks** are used
→ but usually not in the MLP form for reasons to be elaborated
- ▶ example in predicting a spatially varying field, application in parameterisations
→ prediction here for **thickness fluxes**

Oceanic application

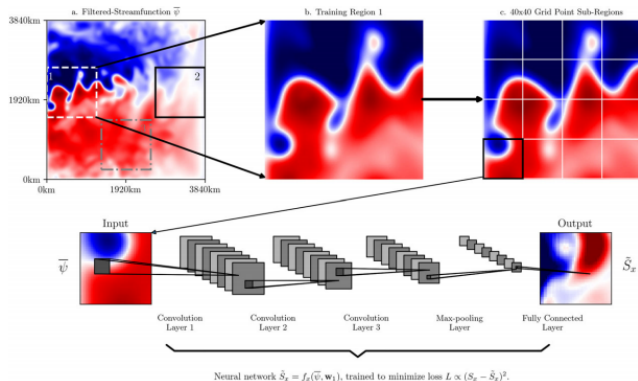


Figure: CNN applied to an eddy parameterisation problem. From Bolton & Zanna (2019).

► eddy parameterisation problem

→ regression: predict one image from another with **CNNs**

Oceanic application

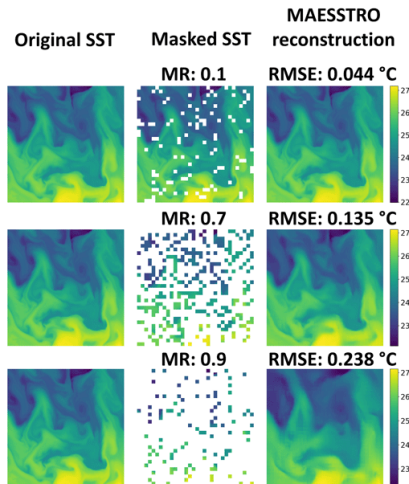


Figure: Reconstruction of SST from occluded data. From Fig. 2 of Goh *et al.* (2024).

- ▶ e.g. satellite data
 - masked **auto-encoder** to fill out the gaps
 - model data but masked accordingly (e.g. cover by cloud)
 - don't have to wait for the satellite to pass again

Q. reconstruction to be **constrained** physically? (e.g. maintain **geostrophy**?)

Oceanic application

- ▶ e.g. oxygen content
 - random forest and MLPs using ship and argo data
 - to interpolate sparse data content
 - reconstruction to quantify rates of **deoxygenation**

Q. reconstruction to be **constrained** by principles?

(e.g. PINNs later)

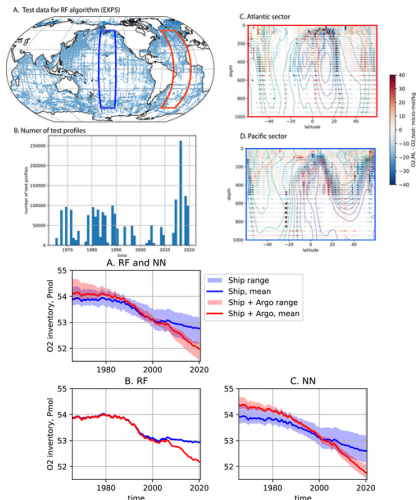


Figure: Reconstruction of observed oxygen content from ship and argo data. From Fig. 8 and 12 of Ito *et al.* (2024).

Recap: classifiers/regressors

- recall for supervised learning we basically have

$$Y = f(X),$$

and given X and Y we want the classifier/regressor f

- f can be represented
 - symbolically (e.g. linear models)
 - as decision trees (e.g. random forests)
 - as a chain of elementary operations (e.g. **neural networks**)

Neural networks: anatomy

- ▶ easiest probably with an example:

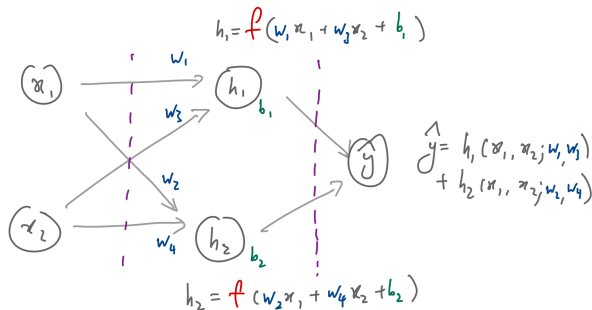


Figure: Made up example of a neural network (that you can code/train up yourself in principle; see notebook).

- ▶ input $X = (x_1, x_2)$ passes through network and gets transformed into prediction $Y = \hat{Y}$

Neural networks: anatomy

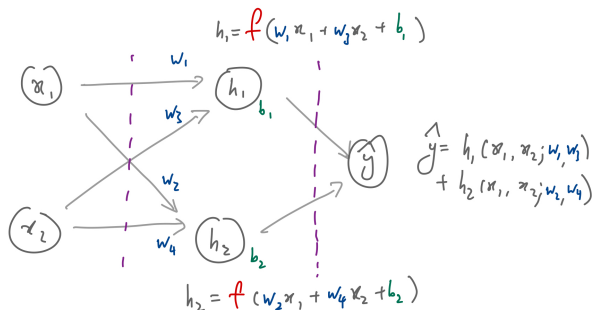


Figure: Made up example of a neural network (that you can code/train up yourself in principle; see notebook).

- each **hidden layer** have **nodes**, where we
 - multiply things by **weights** w_i
 - add on **biases** b_i
 - passed through an **activation function** f

Neural networks: anatomy

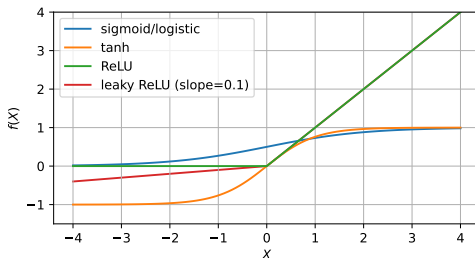


Figure: Samples of common activation functions: sigmoid (bounded between $[0, 1]$), tanh (bounded between $[-1, 1]$) and ReLU (bounded below by 0). Note ReLU is basically hinge loss but flipped the other way around. Leaky ReLU has a slope in the negative part.

- **activation functions** add nonlinearity to the system (see later)
 - maps the elementary operations to known ranges
 - usually piecewise differentiable (see three slides later for reason)
 - e.g. ReLU controls 'firing' of neurons (if $x < 0$)

Neural networks: as approximators

- Q. just a bunch of elementary operations, cannot be complex enough surely?
- ▶ **universal approximation theorem** for neural networks:
 - 'sensible' (!?) functions can be approximated using above operations given sufficient 'complexity'
 - 'complexity' means number of nodes and/or layers
 - ▶ note: an **existence** theorem
 - doesn't tell you how you would train it
 - there are lower/upper bounds for complexity in some cases

Neural networks: training

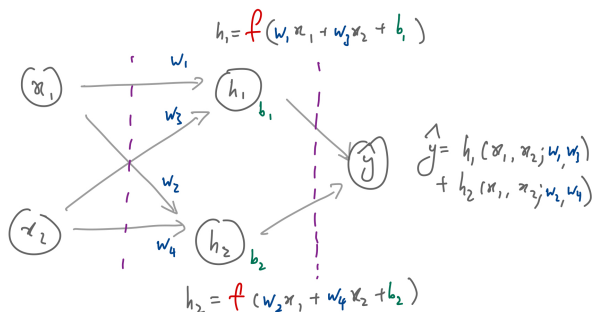


Figure: Made up example of a neural network (that you can code/train up yourself in principle; see notebook).

- one **feed-forward** results in \hat{y}
 - evaluate a **loss function** $J(Y, \hat{y})$
 - J depends on problem, e.g. one sided for classification, L^2 for regression, information entropy based etc.

Neural networks: training

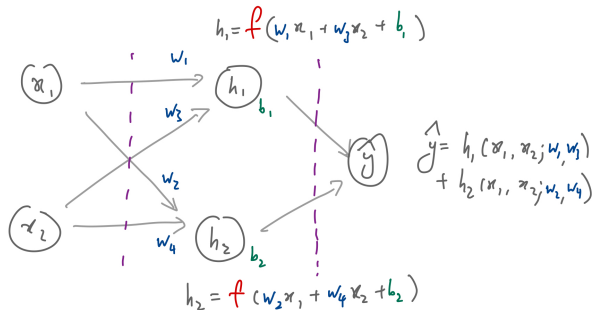


Figure: Made up example of a neural network (that you can code/train up yourself in principle; see notebook).

- aim: find minimum J by changing model parameters

$$\theta = (w_i, b_i)$$

→ want to solve something like $\partial J / \partial \theta = 0$ (abusing notation here...)

Neural networks: training

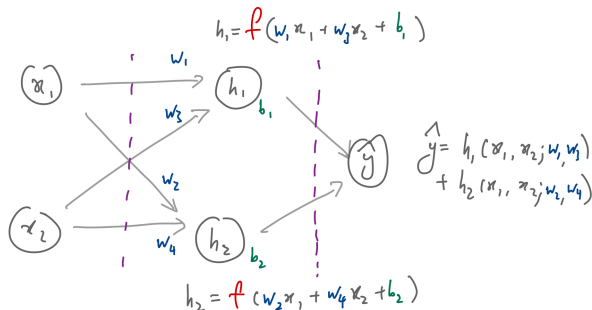


Figure: Made up example of a neural network (that you can code/train up yourself in principle; see notebook).

► **back-propagation** (chain rule basically): e.g.,

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial h_1} \frac{\partial h_1}{\partial w_1} = \frac{\partial J}{\partial h_1} f'(w_1 x_1 + \dots) x_1.$$

→ do for all, SGD etc. and iterate until convergence/bored

Neural networks: training

- ▶ more nodes?
 - more equations
- ▶ more layers?
 - longer chains
- ▶ activation functions?
 - choose piecewise differentiable and 'easy' functions
- ▶ other components within the hidden layer?
 - (e.g. convolutional kernels in CNNs, hidden states in RNNs)
 - assume differentiability but proceed as usual
- ▶ constraint and/or dynamical layers?
 - can deal with in principle (e.g. adjoints)...

Perceptrons

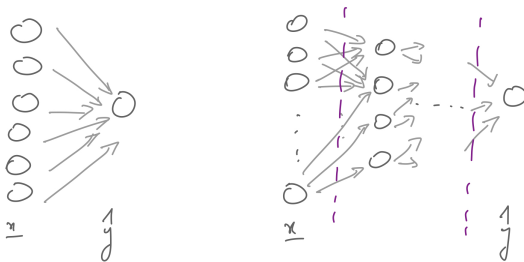
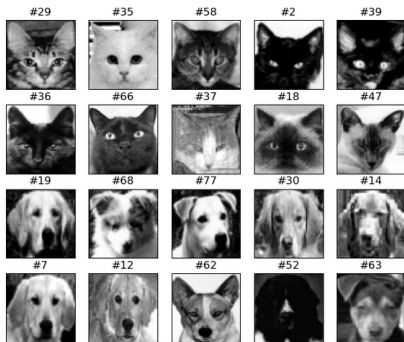


Figure: Schematic of perceptrons (no hidden layers) and MLPs (at least one hidden layer).

► perceptrons

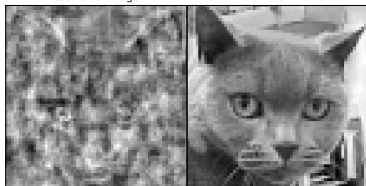
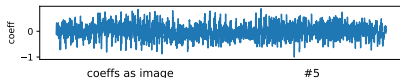
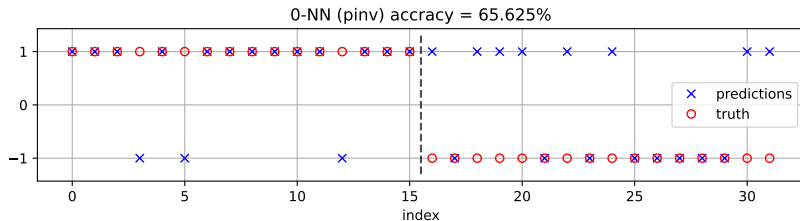
- no hidden layers, may have activation functions
- can only really do binary classification (i.e. 'percept')
- case of no activation function \Rightarrow matrix inversion essentially

Perceptrons: classification



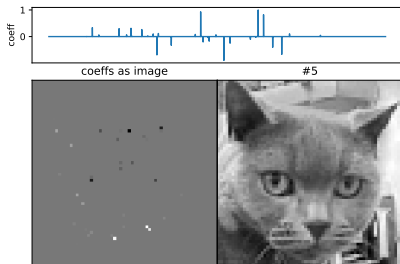
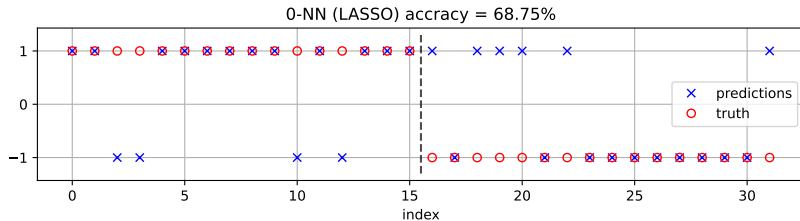
- classifying cats and dogs
 - this is a hard problem!
 - cats labelled as 1 and dogs as -1
 - throw in full image
 - standardise, train/test split
 - can probe the model in this case, **feature identification**

Perceptrons: classification



- ▶ just matrix inversion
 - L^2 minimisation with no penalisation
 - many non-zero values
 - some sort of 'shape', 'eyes' and 'facial features' identified?

Perceptrons: classification



► Lasso optimisation instead
→ L^2 minimisation with L^1 penalisation

→ promotes sparsity (but α dependent)

→ 'eyes', 'forehead' and 'mouth' important?

MLPs

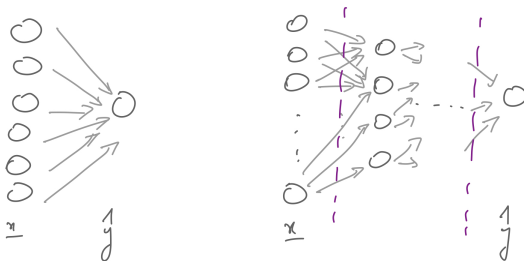


Figure: Schematic of perceptrons (no hidden layers) and MLPs (at least one hidden layer).

► multi-layer perceptrons (MLPs)

- bit of a misnomer, because above limitation with perceptrons do not strictly apply
- sometimes artificial neural networks (ANNs) (misleading?)
- can be 'deep' neural networks (DNNs) (misleading?)

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
→ need to perform the **feed-forward** calculations... (easy)

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
 - need to perform the **feed-forward** calculations... (easy)
 - need to evaluate the loss function... (easy)

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
 - need to perform the **feed-forward** calculations... (easy)
 - need to evaluate the loss function... (easy)
 - need to register the sequence of computations in order to do back-propagation (the **computation graph**)...

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
 - need to perform the **feed-forward** calculations... (easy)
 - need to evaluate the loss function... (easy)
 - need to register the sequence of computations in order to do back-propagation (the **computation graph**)...
 - need to interface with the optimisers...

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
 - need to perform the **feed-forward** calculations... (easy)
 - need to evaluate the loss function... (easy)
 - need to register the sequence of computations in order to do back-propagation (the **computation graph**)...
 - need to interface with the optimisers...
 - other things like **batching**, **train-test set shuffling**, **CPU/GPU capabilities**, etc...

How to build the MLPs?

- ▶ single layer is easy (just matrix inversion), but multi-layers?
 - need to perform the **feed-forward** calculations... (easy)
 - need to evaluate the loss function... (easy)
 - need to register the sequence of computations in order to do back-propagation (the **computation graph**)...
 - need to interface with the optimisers...
 - other things like **batching**, **train-test set shuffling**, **CPU/GPU capabilities**, etc...
- ▶ TL;DR: python has a load of packages already set up to do all the above essentially
 - basically just need to know how to call them
 - building neural nets in R actually calls the python packages...

PyTorch

- ▶ **PyTorch** and **TensorFlow** the two most popular libraries for neural networks
 - underlying them are procedure to manipulate **tensors** (multi-dim arrays)
 - links up to solvers and loss functions accordingly
 - have interfaces to help (e.g. **Lightning** and **Keras**)
- ▶ **JAX** (e.g. w/ Keras) could be used in principle



PyTorch: building the MLP

- ▶ need to convert arrays into **tensor** objects
 - so PyTorch library knows what to do with them
 - extra useful functionalities (e.g. reshaping, resizing after manipulation)
- ▶ FloatTensor: **floats** (but `float32`)
- ▶ LongTensor: **signed integers**
- ▶ ByteTensor: **unsigned integers**
- ▶ in principle will do 'sane' things, but best be explicit
 - e.g. `torch.tensor([1., 1., 1.])` will default to FloatTensor
 - e.g. `torch.tensor([1, 1, 1])` will default to ByteTensor, will **fail** when passed to `conv1d`

PyTorch: building the MLP

- specify neural network structure

```
# define the model architecture

class nn_classify_pets(nn.Module):
    def __init__(self): # specify input dims below
        super(nn_classify_pets, self).__init__()

        # nest it in one go
        self.layers = nn.Sequential(
            nn.Linear(64*2, 100), # image is 64*2, 100 nodes (sklearn default)
            # nn.ReLU(),          # no activation function
            nn.Linear(100, 2)     # two possible outputs
        )

    # actual model structure: input -> hidden layer -> outputs, ReLU on input and hidden
    def forward(self, x):
        out = self.layers(x)
        return out
```

- define as a class, need `__init__`
- specify network structure
- need a `forward` to specify the feed-forward
→ this is for internal **taping/annotation** for the **computation graph**, needed for back-propagation later

PyTorch: building the MLP

- ▶ define loss function
 - for binary classification we can choose `CrossEntropy`
 - others would work too
- ▶ define optimiser
 - use Adam again
 - need to initialise model (e.g. `model = MLP()`) and pass model parameters to the optimiser
 - telling the optimiser what are the control variables
- ▶ sequence of things to do:
 - do a feed-forward
 - evaluate loss function
 - back-propagate
 - update parameters, repeat

PyTorch: building the MLP

```
for epoch in range(num_epochs):  
  
    # iteration step  
  
    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(X_train) # feed-forward  
    J_train = J(Y_pred, Y_train) # compute loss  
    J_train.backward() # back propagation  
    optimizer.step() # iterate  
    model.eval() # put the model in evaluation mode (taping is off for diags below)  
  
    # diagnostics: evaluation of metrics as we go along  
    with torch.no_grad(): # force no taping just in case  
        Y_pred = model(X_valid)  
        J_valid = J(Y_pred, Y_valid)  
        train_J[epoch] = J_train.item()  
        valid_J[epoch] = J_valid.item()
```

- ▶ above subroutine chains it all up, iterates over epochs
 - safety call to clear some calculations to avoid contamination
 - extra diagnostic calls

MLP: classification

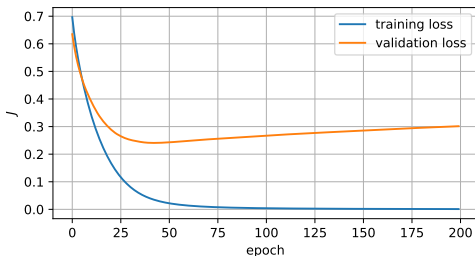


Figure: Example of a loss curve with training epoch.

- an example of a **loss curve** with training decreasing with iterations (**epoch**)
 - continues until some criterion is reached
 - **early-stopping** is possible (didn't do it here)

MLPs: classification

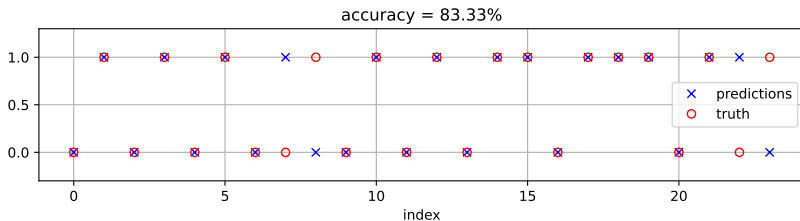


Figure: Perceptron with two hidden layers or size 200 and 50, no activation function.

- classification skill on test set
→ seems ok?

MLPs: regression problem

- ▶ just using cats
 - predict top half from bottom half (use L^2 /MSE loss)
 - vice-versa might work better?
- ▶ again, this is a hard problem!

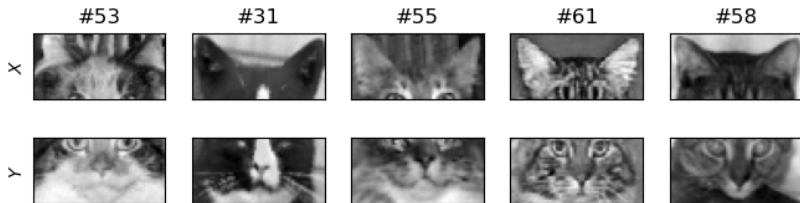


Figure: No animals were hurt in the present demonstration.

MLPs: regression problem

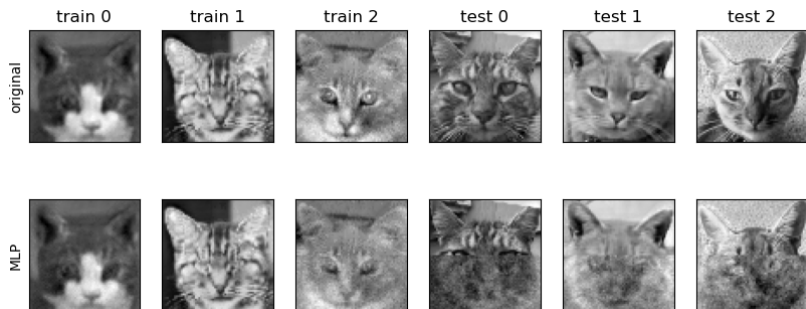


Figure: Predicting bottom half from top half, default settings.

- note that even the training set had issues doing to the prediction...
 - predictions can be quite cursed if eye happens to be in top half (e.g. grant us eyes)

MLPs: regression problem

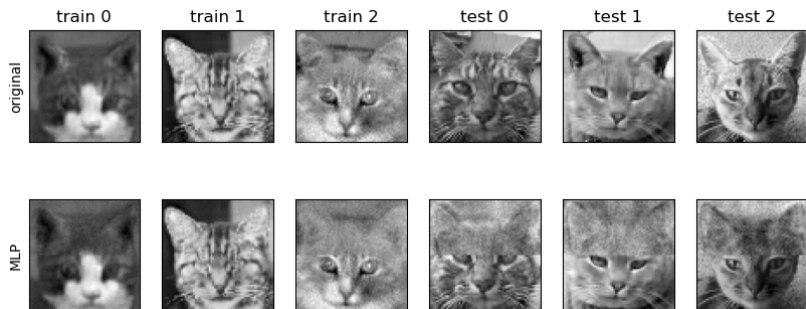


Figure: Predicting top half from bottom half, default settings.

- note that even the training set had issues doing to the prediction...
 - less cursed at least subjectively
 - at least somewhat continuous predictions?

MLPs: regression problem

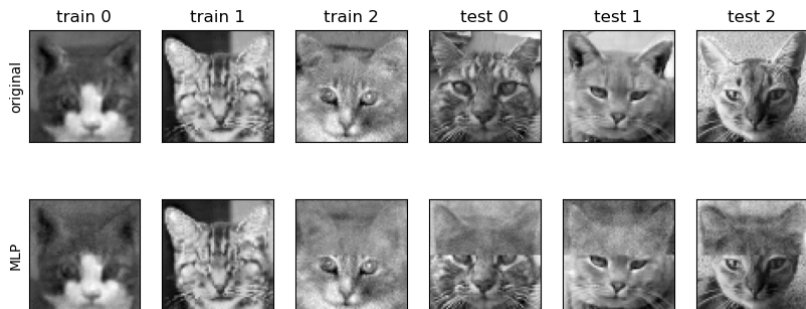


Figure: Predicting top half from bottom half, reasonably complicated.

- note that even the training set had issues doing the prediction...
 - not doing much better with substantially increased complexity

Demonstration



Figure: Cursed beasts reconstruction (out of sample application of MLP).

- ▶ idea behind neural networks
→ ways to control over-fitting (e.g. **dropout**, **regularisation**)
- ▶ MLPs bound by curse of dimensionality!
- ▶ need to cross-validate and tune hyper-parameters!
- ▶ basic use of PyTorch here (more things next time)

Moving to a Jupyter notebook →