ANN Training Strategies

Traditional ML:

- feature engineering is crucial
- hyperparameter optimization is somewhat useful

Deep learning:

- neural network / hidden layers performing feature engineering
- hyperparameter optimization is very important

Parameters vs. hyperparameters:

- *Parameters*: values/weights that your network learns; your model
- *Hyperparameters*: values that you set / design decisions that you make

What can you tweak & change in a neural network:

- Number of layers
- Number of neurons
- Number of iterations / epochs
- Activation function
- Loss function
- Weights initialization
- Regularization
- Optimization algorithm

Issues we are trying to solve by doing this:

- Overfitting
- Vanishing or exploding gradients
- Model performance
- Training time

1. Number of neurons / (hidden) layer

- more neurons -> more weights / parameters
- more neurons -> overfitting

2. Number of layers

- more layers -> more complex features
- more layers -> overfitting, longer training

3. Number of epochs

- more epochs -> longer training, (potentially) better model
- more epochs -> more likely to overfit

You should keep increasing your model complexity as long as the score on your validation dataset improves. Observe performance of your validation dataset and stop as soon as performance decreases.

4. Activation function

- sigmoid (*output layer*)
  - outputs a probability of data point belonging to a single class
  - last layer (single neuron) in binary classification
  - don't use in hidden layers because there are better options
- tanh (*hidden layer*)
  - similar to sigmoid in shape but output (-1, 1)
  - works better sigmoid in hidden layer because it centers the output around 0 (and not 0.5)
  - don't use in the output layer, use sigmoid over tanh
  - similar to sigmoid, could be slow to train because derivative close to zero for large x
- ReLU (Rectified Linear Unit) (*hidden layer*)
  - gradient is easy to calculate (0 for negative, 1 for positive values, in 0 you can choose 0 and 1)
  - fast to train, slope for positive values is very different from

    zero (even if you have 0 in half of the space, enough of the values are positive for it not to matter)
    - good first choice for hidden layers
- Leaky ReLU (*hidden layer*)
    - doesn't have 0 gradient for negative values
    - can have this "leaky-ness" as parameter, choose the best value
    - for hidden layers mostly
- ELU (Exponential Linear Unit) (*hidden layer*)
    - ELU > Leaky ReLU > ReLU
- linear (*output layer*)
    - no activation
    - in the general case, don't use linear in hidden layers; output is then just a linear combination of your input feature vector
    - use in the output layer for a regression problem
- softmax (*output layer*)
    - equivalent to sigmoid but for multiclass classification
    - use in the output layer, with as many neurons as classes
    - probabilities add up to 1

5. Loss function

This is defined in the `model.compile()` step.

- Binary classification -> *sigmoid*, binary crossentropy (log loss)
- Multiclass classification -> *softmax*, categorical crossentropy ("log loss for more than 2 classes")
- Multilabel classification -> *sigmoid*, binary crossentropy
- Regression -> *linear*, MSE

6. Weights initialization

This is defined when adding a layer in Keras.

- Can help prevent (or minimize) vanishing and exploding gradients problem.
- Random (small random numbers) `np.random.randn(shape)*0.01`
- Variance of w ~1/n

    - ReLU: `np.random.randn(shape)*sqrt(2/n_previous)`
    - tanh: `np.random.randn(shape)*sqrt(1/n_previous)`
    - Xavier initialization: `np.random.randn(shape)*sqrt(2/(n_previous+n_current))` (`GlorotNormal` in Keras)

7. Regularization

- The main goal of regularization is to limit overfitting
- In most (but not all) cases it does that by limiting the value of weights
- *L1 norm* — sum of the absolute values of the weights gets added to loss function, can result in weights being equal to zero
- *L2 norm* — sum of the squares of weights gets added to loss function, makes weights smaller, but not equal to zero
- *Dropout* — in each iteration, remove a random set of nodes from the network (0.2-0.5), results in a smaller network less likely to overfit <- most often used / likely to give you best performance
- *Early stopping* — stop when the validation learning curve starts to turn up and overfitting starts
- *Data augmentation* — especially when working with image data; flip, crop, shift, e.g. `ImageDataGenerator`.

8. Optimization algorithms

- Gradient descent based
    - Batch gradient descent
        - Uses the whole dataset for each weight update
            - Slow
            - Uses a lot memory
    - Stochastic gradient descent
        - Uses a single data point in each weights update
            - Noisy
    - Mini-batch gradient descent
        - Uses a part of the dataset for each weight update
            - Sweet spot

        - Uses medium amount of memory
        - Less noisy than stochastic GD
        - Pass `batch_size` in `model.fit()`
- Adaptive optimization algorithms
    - RMSProp, Adam, Adagrad
    - Take into account previous weight updates (exponetially decaying weighted average), and/or adapt the learning rate.

There is somewhat of a consensus that RMSProp or Adam will likely give you the best performance.