Train, Validate, Test

Learning Machines

(and some Deep Network Fundamentals)

Kate Farrahi & Jonathon Hare

Vision, Learning and Control
University of Southampton

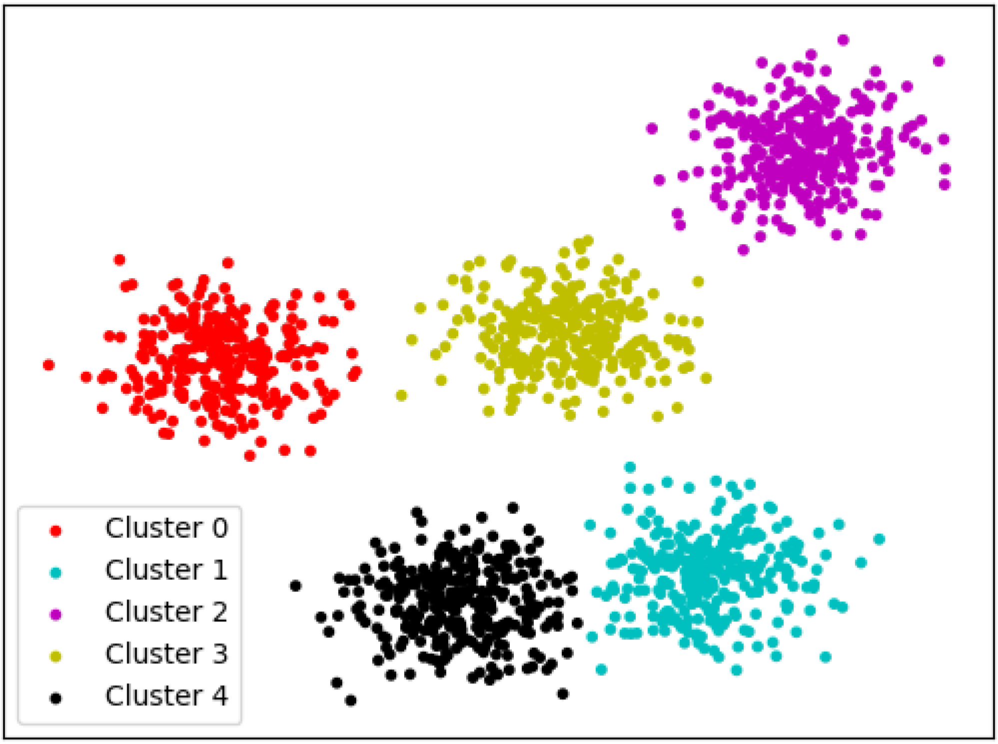
Types of Learning

- Supervised Learning - learn to predict an output when given an input vector
- Unsupervised Learning - discover a good internal representation of the input
- Reinforcement Learning - learn to select an action to maximize the expectation of future rewards (payoff)
- Self-supervised Learning - learn with targets induced by a prior on the unlabelled training data
- Semi-supervised Learning - learn with few labelled examples and many unlabelled ones

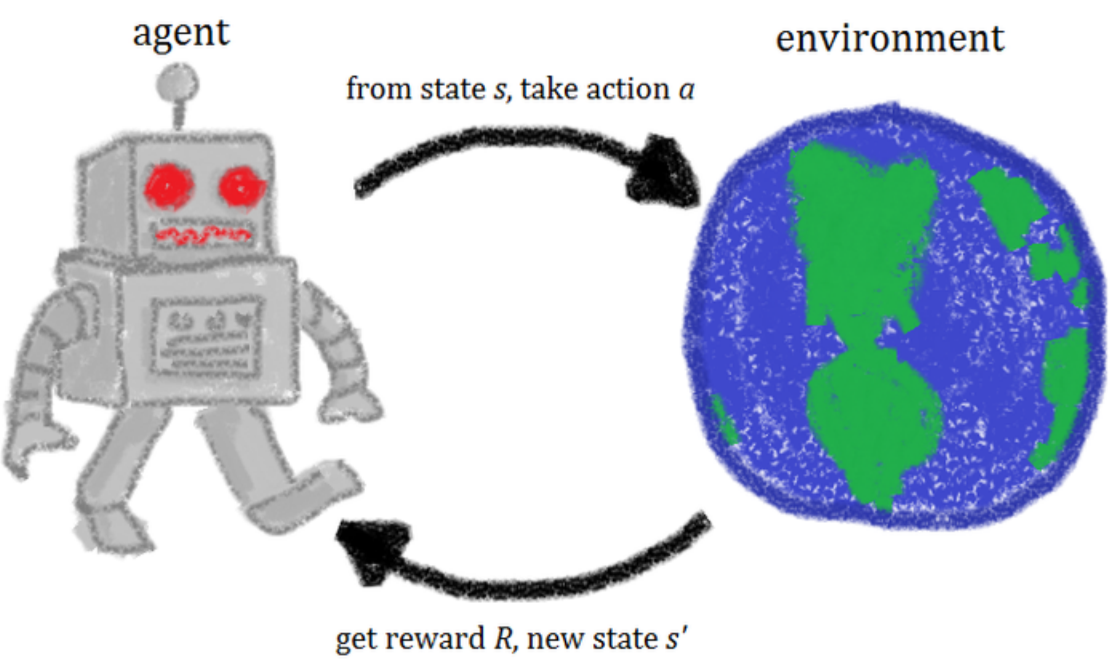
Supervised Learning

Newell, Alejandro, Kaiyu Yang, and Jia Deng. "Stacked hourglass networks for human pose estimation." ECCV'16. Springer, 2016.

Unsupervised Learning



Reinforcement Learning



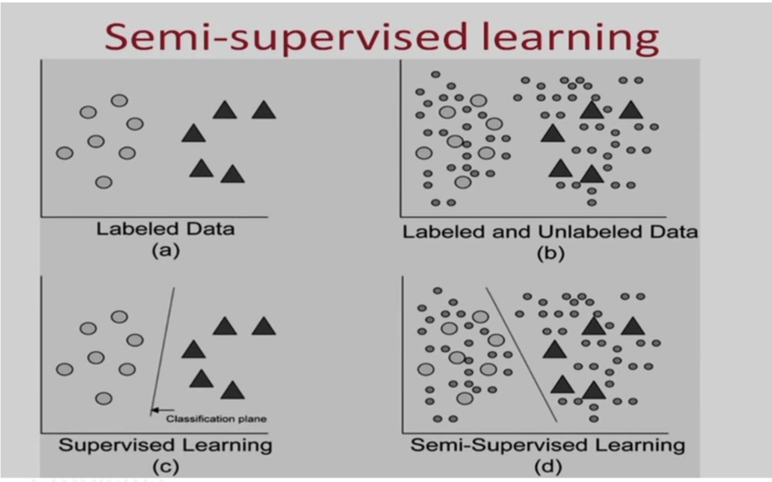
Reference: Wikipedia
`https://simple.wikipedia.org/wiki/Reinforcement_learning`

Self-supervised Learning

- The basic idea of self-supervised learning (SSL) is to automatically generate some kind of supervisory signal to solve some task (typically, to learn representations of the data or to automatically label a dataset).
- SSL be regarded as an intermediate form between supervised and unsupervised learning.
- Training can occur with data of lower quality.
- SSL more closely imitates the way humans learn to classify objects.

Reference: Wikipedia https://en.wikipedia.org/wiki/Self-supervised_learning

Semi-supervised Learning



Jeremy Howard. The wonderful and terrifying implications of computers that can learn. TEDxBrussels. http://www.ted.com/talks/jeremy_howard_the_wonderful_and_terrifying_implications_of_computers_that_can_learn

Image taken from: https://medium.com/dataseries/two-minutes-of-semi-supervised-learning-f0eb62729530

Generative Models

- Many unsupervised and self-supervised models can be classed as 'Generative Models'.
- Given unlabelled data X, a unsupervised generative model learns $P[X]$.
  - Could be direct modelling of the data (e.g. Gaussian Mixture Models)
  - Could be indirect modelling by learning to map the data to a parametric distribution in a lower dimensional space (e.g. a VAE's Encoder) or by learning a mapping from a parameterised distribution to the real data space (e.g. a VAE Decoder or GAN)
- These are characterised by an ability to 'sample' the model to 'create' new data

Generative vs. Discriminative Models (II)

Generative vs. discriminative approaches to classification use different statistical modelling.

- Discriminative models learn the boundary between classes. A (probabilistic) discriminative model is a model of the conditional probability of the target Y given an observation X: $P[Y|X]$.
- Generative models of labelled data model the distribution of individual classes. Given an observable variable X and a target variable Y, a generative model is a statistical model that tries to model $P[X|Y]$ and $P[Y]$ in order to model the joint probability distribution $P[X, Y]$.

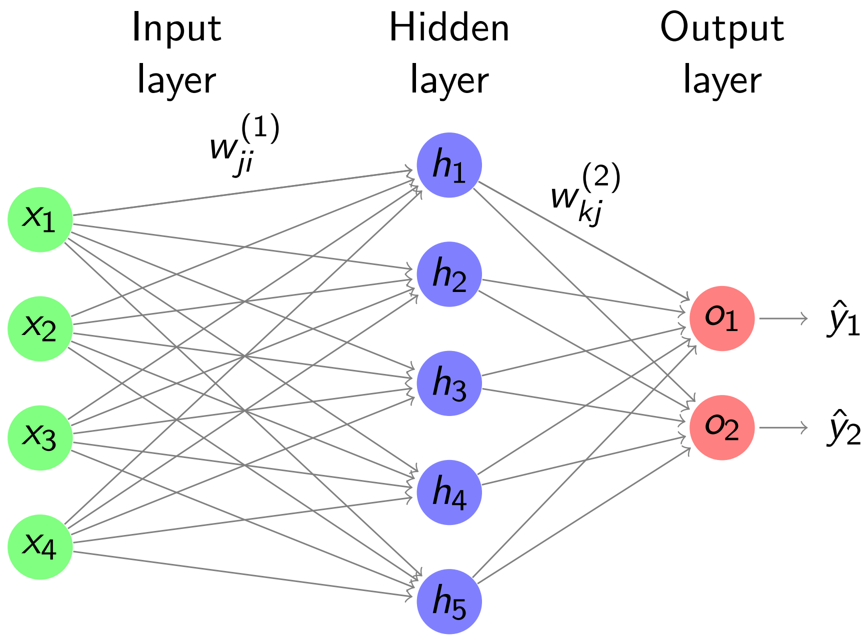
Two Types of Supervised Learning

- Regression: The machine is asked predict k numerical values given some input. The machine is a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$.
- Classification: The machine is asked to specify which of k categories some input belongs to.
    - Multiclass classification - target is one of the k classes
    - Multilabel classification - target is some number of the k classes
    - In both cases, the machine is a function $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$ (although it is most common for the learning algorithm to actually learn $\hat{f} : \mathbb{R}^n \rightarrow \mathbb{R}^k$).
- Note that there are lots of exceptions in the form the inputs (and outputs) can take though! We'll see lots of variations in the coming weeks.

How Supervised Learning Typically Works

- Start by choosing a model-class: $\hat{y} = f(\boldsymbol{x}; \boldsymbol{W})$ where the model-class f is a way of using some numerical parameters, $\boldsymbol{W}$, to map each input vector $\boldsymbol{x}$ to a predicted output $\hat{y}$.
- *Learning* means adjusting the parameters to reduce the discrepancy between the true target output y on each training case and the output $\hat{y}$, predicted by the model.

Let's look at an unbiased Multilayer Perceptron...



Without loss of generality, we can write the above as:

$$\hat{\boldsymbol{y}} = g(f(\boldsymbol{x}; \boldsymbol{W}^{(1)}); \boldsymbol{W}^{(2)}) = g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x}))$$

where f and g are activation functions.

Common Activation Functions

- Identity
- Sigmoid (aka Logistic)
- Hyperbolic Tangent (tanh)
- Rectified Linear Unit (ReLU) (aka Threshold Linear)

Final layer activations

$$\hat{y} = g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)}\boldsymbol{x}))$$

- What form should the final layer function g take?
- It depends on the task (and on the chosen loss function)...
  - For regression it is typically linear (e.g. identity), but you might choose others if you say wanted to clamp the range of the network.
  - For binary classification (MLP has a single output), one would choose Sigmoid
  - For multilabel classification, typically one would choose Sigmoid
  - For multiclass classification, typically you would use the Softmax function

Softmax

The softmax is an activation function used at the output layer of a neural network that forces the outputs to sum to 1 so that they can represent a probability distribution across a discrete mutually exclusive alternatives.

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \forall i = 1, 2, \ldots, K$$

- Note that unlike the other activation functions you've seen, softmax makes reference to all the elements in the output.
- The output of a softmax layer is a set of positive numbers which sum up to 1 and can be thought of as a probability distribution.
- Note:

$$\begin{aligned}
\frac{\partial\,\text{softmax}(\boldsymbol{z})_i}{\partial z_i} &= \text{softmax}(z_i)(1 - \text{softmax}(z_i)) \\
\frac{\partial\,\text{softmax}(\boldsymbol{z})_i}{\partial z_j} &= \text{softmax}(z_i)(1(i = j) - \text{softmax}(z_j)) \\
&= \text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))
\end{aligned}$$

Ok, so let's talk loss functions

- The choice of loss function depends on the task (e.g. classification/regression/something else)
- The choice also depends on the activation function of the last layer
  - Some classification losses require *raw outputs* (e.g. a linear layer) of the network as their input
    - These are often called *unnormalised log probabilities* or *logits*
    - An example would be hinge-loss used to create a Support Vector Machine that maximises the margin — e.g.: $\ell_{hinge}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$ with a true label, $y \in \{-1, 1\}$, for binary classification.
- There are many different loss functions we might encounter (MSE, Cross-Entropy, KL-Divergence, huber, L1 (MAE), CTC, Triplet, ...) for different tasks.

The Cost Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point (here assumed to be a vector, but equally applicable to a scalar) is given by $\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_i (\hat{y}_i - y_i)^2 = (\hat{\boldsymbol{y}} - \boldsymbol{y})^\top (\hat{\boldsymbol{y}} - \boldsymbol{y})$
- We often multiply this by a constant factor of $\frac{1}{2}$ — can anyone guess/remember why?
- $\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is the predominant choice for regression problems with linear activation in the last layer
- For a classification problem with Softmax or Sigmoidal (or really anything non-linear) activations, MSE can cause slow learning, especially if the predictions are very far off the targets
  - Gradients of ℓ_{MSE} are proportional to the difference in target and predicted multiplied by the gradient of the activation function[1]
  - The Cross-Entropy loss function is generally a better choice in this case

[1] http://neuralnetworksanddeeplearning.com/chap3.html

Binary Cross-Entropy

For the binary classification case:

$$\ell_{BCE}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

- The cross-entropy cost function is non-negative, $\ell_{BCE} > 0$
- $\ell_{BCE} \approx 0$ when the prediction and targets are equal (i.e. $y = 0$ and $\hat{y} = 0$ or when $y = 1$ and $\hat{y} = 1$)
- With Sigmoidal final layer, $\frac{\partial \ell_{BCE}}{\partial \mathbf{W}_i^{(2)}}$ is proportional to just the error in the output $(\hat{y} - y)$ and therefore, the larger the error, the faster the network will learn!
- Note that the BCE is the negative log likelihood of the Bernoulli Distribution

Binary Cross-Entropy — Intuition

- The cross-entropy can be thought of as a **measure of surprise**.
- Given some input x_i, we can think of $\hat{y}_i$ as the estimated probability that x_i belongs to class 1, and $1 - \hat{y}_i$ is the estimated probability that it belongs to class 0.
- Note the extreme case of infinite cross-entropy, if your model believes that a class has 0 probability of occurrence, and yet the class appears in the data, the 'surprise' of your model will be infinitely great.

Binary Cross-Entropy for multiple labels

In the case of multi-label classification with a network with multiple sigmoidal outputs you just sum the BCE over the outputs:

$$\ell_{BCE} = -\sum_{k=1}^{K}[y_k \log(\hat{y}_k) + (1 - y_k)\log(1 - \hat{y}_k)]$$

where K is the number of classes of the classification problem, $\hat{y} \in \mathbb{R}^K$.

Multiclass classification with Softmax Outputs

- Softmax can be thought of making the K outputs of the network mimic a probability distribution.
- The target label y could also be represented as a distribution with a single 1 and zeros everywhere else.
  - e.g. they are "one-hot encoded".
- In such a case, the obvious loss function is the *negative log likelihood* of the Categorical distribution (aka Multinoulli, Generalised Bernoulli, Multinomial with one sample)[2]: $\ell_{NLL} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$
  - Note that in practice as y_k is zero for all but one class you don't actually do this summation, and if y is an integer class index you can write $\ell_{NLL} = -\log \hat{y}_y$.
  - PyTorch combines LogSoftmax with NLL in one loss and calls this "Categorical Cross-Entropy" (so you would use this with a *linear output layer*)

[2]Note: Keras calls this function 'Categorical Cross-Entropy'; you would need to have a Softmax output layer to use this

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = \sum_{(\boldsymbol{x},y)\in\boldsymbol{D}} \ell(f(\boldsymbol{x},\boldsymbol{\theta}), y)$ for some loss function ℓ, dataset $\boldsymbol{D}$ and model f with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** $\mathcal{L}$ by the learning rate η multiplied by the gradient:

for each Epoch:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

Reminder: Stochastic Gradient Descent

- Define loss function ℓ, dataset $\boldsymbol{D}$ and model f with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient:

for each Epoch:

for each $(\boldsymbol{x}, y) \in \boldsymbol{D}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \ell$$

A Quick Introduction to Tensors

Broadly speaking a tensor is defined as a linear mapping between sets of algebraic objects[3].

A tensor T can be thought of as a generalization of scalars, vectors and matrices to a single algebraic object.

We can just think of this as a multidimensional array[4].

- A $0D$ tensor is a scalar
- A $1D$ tensor is a vector
- A $2D$ tensor is a matrix
- A $3D$ tensor can be thought of as a vector of identically sized matrices
- A $4D$ tensor can be thought of as a matrix of identically sized matrices or a sequence of $3D$ tensors
- ...

[3] This statement is always entirely true

[4] This statement will upset mathematicians and physicists because its not always true for them (but it is for us!).

Operations on Tensors in PyTorch

- PyTorch lets you do all the standard matrix operations on 2D tensors
  - including important things you might not yet have seen like the hadamard product of two $N \times M$ matrices: $\boldsymbol{A} \odot \boldsymbol{B}$)
- You can do element-wise add/divide/subtract/multiply to ND-tensors
  - and even apply scalar functions element-wise ($\log, \sin, \exp, \ldots$)
- PyTorch often lets you *broadcast* operations (just like in numpy)
  - if a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).[5]

[5] Important - read and understand this after the lab next week: https://pytorch.org/docs/stable/notes/broadcasting.html

Homework

PyTorch Tensor 101:
`https://colab.research.google.com/gist/jonhare/d98813b2224dddbb234d2031510878e1/notebook.ipynb`