Statistics for Particle Physicists
LPC 2021, Fermilab

Lecture 16: Introduction to Machine Learning – Part 3

Harrison B. Prosper

Department of Physics, Florida State University

1 December, 2021

Outline

Recap

- Last time, we noted that deep convolutional networks (DCNN) are the goto method for image recognition. However, since any structured data can be mapped into a set of 2D arrays the range of application of DCNNs is much broader than image processing.
- We also noted that graph neural networks (GNN) are a useful class of models when data can be viewed as a 2D array in which the number of rows may vary from one data instance to another and where the order of rows is judged to be irrelevant.

Outline

In this, the last lecture of the series, we'll cover:

- Auto-Encoders (AE) – models for compressing data, de-noising data and editing data in a variety of ways.
- Reinforcement learning (RL) – learning through guided exploration.
- Uncertainty quantification – how reliable is my model?

Outline

An auto-encoder is a function that maps high-dimensional data to a lower-dimensional space (called the latent space).

If constraints are placed on the distributions within the latent space, for example, to ensure that similar inputs are mapped to similar positions in the latent space, it is possible to use an auto-encoder as a generative model.

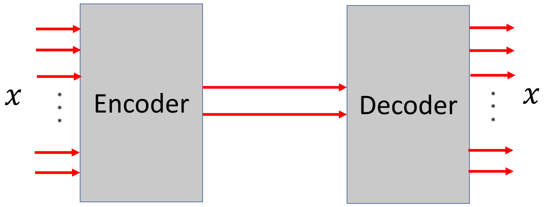
A generative model can be used to generate data, in this case, by sampling the distribution in the latent space. Variational auto-encoders are examples of such models.

However, we shall consider only the simplest possible auto-encoder, one that models the function

$$f : x \to z \to x,$$

where $\dim(z) << \dim(x)$.

A standard auto-encoder has two parts, an encoder $z = f(x)$ and a decoder $x = g(z)$.

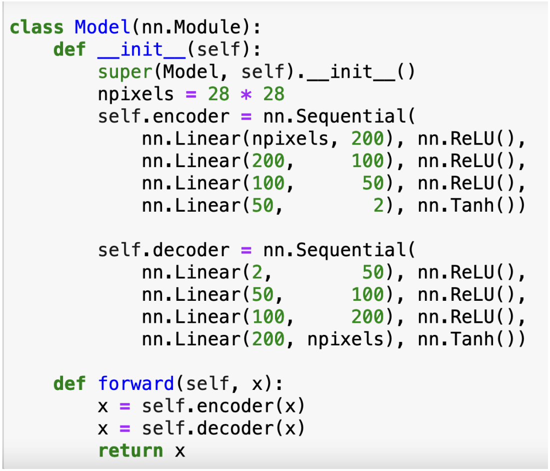


If we use the quadratic loss $L(y = x, f) = [x - f(x, \theta)]^2$, the output of the auto-encoder will be an unbiased estimate of its input.

To illustrate what can be done, let's try to compress the MNIST data into a 2D latent space.

Example (MNIST lossy data compression)

We'll use the following simple model built from `PyTorch` `Linear` functions and `ReLU` non-linearities.



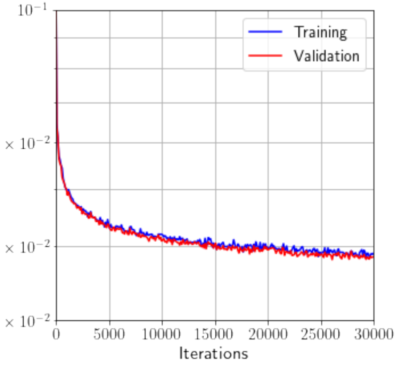
```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        npixels = 28 * 28
        self.encoder = nn.Sequential(
            nn.Linear(npixels, 200), nn.ReLU(),
            nn.Linear(200,     100), nn.ReLU(),
            nn.Linear(100,      50), nn.ReLU(),
            nn.Linear(50,        2), nn.Tanh())

        self.decoder = nn.Sequential(
            nn.Linear(2,        50), nn.ReLU(),
            nn.Linear(50,      100), nn.ReLU(),
            nn.Linear(100,     200), nn.ReLU(),
            nn.Linear(200, npixels), nn.Tanh())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Example (MNIST lossy data compression)

Training details:

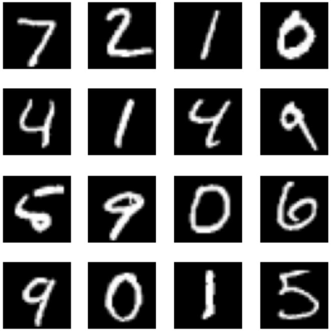
- batch size: 32
- learning rate: 10^{-3}
- number of iterations 60,000

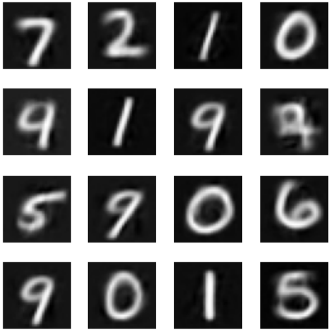
Number of parameters: 365,286.



Example (MNIST lossy data compression)

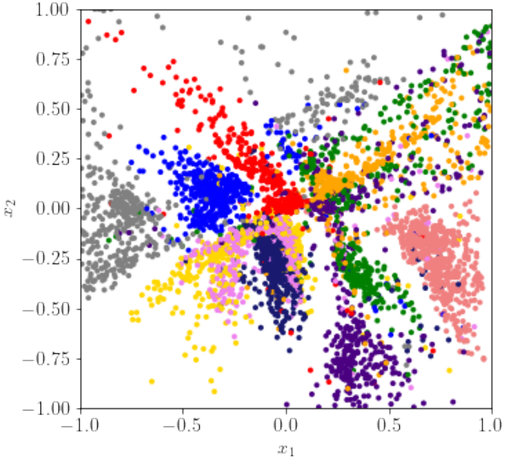
Here are examples of the original and decompressed images.





Example (MNIST lossy data compression)

And here is the clustering of the MNIST data within the 2D latent space.



Outline

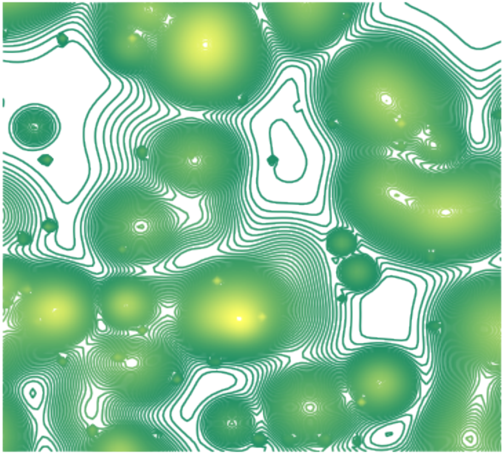
Reinforcement learning is a class of algorithms for constructing optimal solutions to problems through exploration by maximizing the sum of immediate and expected future rewards.

An agent embedded within an environment takes a sequence of actions that move the environment from one state to another. The change in the environment might simply be that the agent is in a different place within it.

Optimal here means that the sequence of actions yields the maximum expected reward. This class of algorithms is particularly well suited for activities that can be construed as games with well-defined rules.

We shall review one of the simplest such algorithms.

To make things a bit more concrete let's suppose that the goal is to traverse a mountainous terrain



starting in the south-west corner and ending in the north-west plane.

One algorithm to solve this problem is to follow a policy whose quality is quantified by a function Q. The algorithm is called Q-learning.

The environment, comprising a mountainous terrain and an agent, is represented by the triplet (x, y, h), where (x, y) is the position of the agent in the terrain and $h(x, y)$ is the height of the terrain at the specified position.

Let's first review the basic concepts of Q-learning.

Q-learning concepts

1. environment: a configuration of objects.
2. state: a set of attributes s that characterize the environment at a given time t.
3. action: a manipulation of the environment that causes it to change.
4. episode: a sequence of states.

Q-learning concepts

5. reward: the immediate reward $R(s, a)$ associated with the action $a : s \to s'$, where s is the current state and s' is the new state.
6. agent: an automaton, e.g., a bot, that can take actions that cause the environment to transition from one state to another.
7. policy: a method for choosing an action, a, in a given state, s.
8. target state: the desired end state, s^*, to be reached by executing a sequence of actions determined by a policy quantified by a Q-function.
9. Q-function: the quality, $Q(s, a)$, of an action a in state s. This is the function to be estimated by the training algorithm.

In general, a transition from the current state to another may depend on previous states. However, the *Q*-learning algorithm assumes the Markov property: *the transition from one state to another depends only on the current state*. The following rule[1], called the Temporal Difference Update,

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q(s', a')],$$

has been shown to converge[2] to a function $Q^*(s, a)$ that maximizes the expected reward from an episode that terminates with the desired state.

The parameter $0 < \alpha \leq 1$ determines how much of the future expected reward is to be taken into account during training relative to the current expected reward, while $0 < \gamma < 1$ is the amount by which a future expected reward is to be discounted relative to the immediate reward R.

[1] Christopher J.C.H Watkins, *Learning from delayed rewards*, PhD Thesis, University of Cambridge, England (1989).

[2] Christopher J.C.H. Watkins and Peter Dayan, *Technical Note Q-Learning*, Machine Learning, **8**, 279-292 (1992).

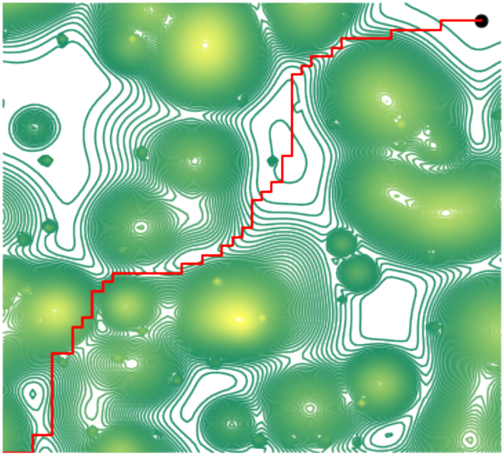
For the mountain trek problem, we'll use a simplified, purely deterministic, version of the Q-learning algorithm:

Algorithm 1 Q-learning

```
initialize $Q(s, a) = 0$, $\gamma = 0.99$, $R_{\max} = 1.0$, $K = 2 \times 10^6$
while $k \in 1, \cdots K$ do
    get random state: $s \leftarrow \{ s \}$
    get best action: $a \leftarrow \text{argmax}_a \, Q(a, s)$
    get next state: $s' \leftarrow s \,|\, a$
    get reward: $R(s, a) \leftarrow R_{\max}$ if $s' == s^*$ else $R_{\max}[h(s) - h(s')]$
    $Q(s, a) \leftarrow R(s, a) + \gamma \max_{a'} Q(s', a')$
end while
```

$h(s)$ is the height of the terrain in current state s, that is, the position of the agent and $h(s')$ is the height at the next state, that is, next position.

Solution



Outline

Deep neural networks are now the method of choice to solve really hard problems, for example: learning to become the best Go player in the world without using human knowledge[3].

But the impressive performance of these machine learning models belie a serious problem.

The standard approach to training these models provides point estimates of the quantities of interest.

In the Go example, given the current and previous board positions s, the model `AlphaGo Zero` estimates probabilities $p(a \mid s)$ for valid actions, a, i.e., moves, as well as the probability that the current player will win. (`AlphaGo Zero` plays itself.)

[3]D. Silver *et al.*, *Mastering the game of Go without human knowledge*, Nature, Vol 550, 2017, p354

But, like most machine learning models, `AlphaGo Zero` provides neither uncertainty estimates nor measures of confidence for its estimates.

Recall that the standard method to train a machine learning model $f(x, \theta)$ is to minimize the empirical risk (i.e., the average loss)

$$R(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i, \theta))$$

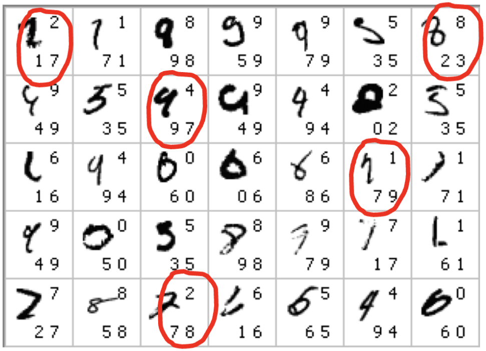
using some variation of stochastic gradient descent, which yields a single optimal model $f(x, \theta^*)$.

For playing against your favorite AI gamer this is a non-issue.

But for a self-driving car the absence of a measure of confidence in potential decisions may mean the difference between avoiding a crash and getting injured in one.

Consider, again, the 35 misclassified MNIST images in the work by Cireșan *et al.* (2010). The authors noted that the next highest output of their DNN correctly classifies 30 of the misclassified images!

(784, 2500, 2000, 1500, 1000, 500, 10)



Upper right: correct answer; lower left answer of highest DNN output; lower right answer of next highest DNN output.

If we had a reliable measure of confidence in the outputs of models, an AI system would be able to flag when it was unsure about what to do next.

If the Bayesian approach is applied to machine learning models, the training of a model becomes a problem of inference in which uncertainty quantification is automatic.

The goal in Bayesian machine learning is to approximate the posterior density

$$\begin{aligned} p(\theta|T) &= \frac{p(T|\theta)\,\pi(\theta)}{p(T)}, \\ &= \frac{p(y|x,\theta)\,p(x|\theta)\,\pi(\theta)}{p(y|x)\,p(x)}, \\ &= \frac{p(y|x,\theta)\,\pi(\theta)}{p(y|x)}, \end{aligned}$$

where θ are the parameters of the machine learning model, $p(y|x,\theta)$ is the likelihood of the training data $T = \{(y_i, x_i)\}$ and $\pi(\theta)$ is the prior density that imposes constraints on the class of models $f(x,\theta)$.

Machine learning models that are "trained" using the Bayesian approach are called Bayesian neural networks (BNN).

Given the posterior density, $p(\theta|T)$, the quantity of interest is the predictive distribution

$$p(y|x, T) \;=\; \int p(y|x, \theta)\, p(\theta|T)\, d\theta,$$

which, unlike models trained by minimizing the average loss, provides a probability distribution over the space of targets y for a given input x.

Unfortunately, the size of the parameter spaces of machine learning models renders the calculation of $p(\theta|T)$ and, therefore, $p(y|x, T)$ intractable.

In practice, $p(\theta|T)$ is approximated either by

- sampling from $p(\theta|T)$ using, e.g., Hybrid Monte Carlo (HMC)[4], or by
- finding a tractable approximation $q(\theta|\phi)$ to $p(\theta|T)$, where ϕ are variational parameters to be optimized.

However, for state-of-the-art models with very large parameter spaces both approximations for $p(\theta|T)$ are extremely challenging, not to mention the difficulty of constructing suitable priors over the parameter spaces.

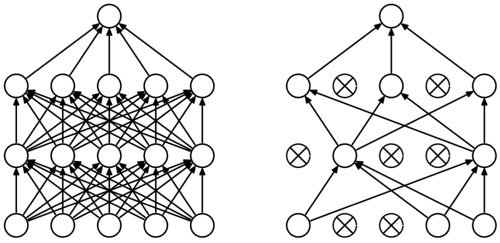
It was therefore a welcome surprise when, in 2017, a surprisingly simple method was proposed by Gal and Ghahramani (G^2) (U. of Cambridge)[5] to model the uncertainty in the outputs of machine learning models.

But to understand G^2's method we need to explain dropout.

[4] Radford Neal, https://www.cs.utoronto.ca/∼radford/fbm.software.html; B.S.Kronheim, M.P.Kuchera, H.B.P., *TensorBNN: Bayesian inference for neural networks using TensorFlow*, Comp. Phys. Comm., Vol. **270**, (2022), 108168.

[5] Y. Gal and Z. Ghahramani, *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*, arXiv:1506.02142v5, 25 May 2016

Dropout is a method for mitigating the problem of overfitting of neural networks that was introduced by Srivastava *et al.* in 2014[6].


(a) Standard Neural Net   (b) After applying dropout.

Srivastava *et al.*

During training, nodes are randomly dropped from the model, which adds additional noise to the gradient calculations.

[6]Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15 (2014) 1929-1958

Gal and Ghahramani showed that a model trained using dropout can use dropout in the trained model by repeatedly calling the model with same input. This method is called Monte Carlo dropout.

Monte Carlo dropout generates an ensemble of output values $\{y\}$ for a given input x that is a Bayesian approximation of $p(y|x, T)$ with a particular choice of prior.

This is a beautifully simple procedure. Unfortunately, however, the uncertainties derived from this method are not calibrated; that is, it is necessary to tune the (hyper)parameters of the model so that approximate intervals can be constructed that are reasonable from a frequentist or Bayesian perspective.

My conclusion: Finding a calibrated, general, method way to quantify uncertainty in machine learning models remains a work in progress. See, however, the very interesting PHYSTAT seminar by Ann Lee (CMU): https://indico.cern.ch/event/1085699/.

Outline

- Auto-encoders, of which now there are many classes, can be used for tasks such as data compression, de-noising, and data generation. We considered a simple example of lossy data compression.
- Reinforcement learning is a huge area of machine learning based on the general idea of maximizing rewards to achieve an optimal solution to a problem. The method has been used for tasks ranging from finding optimal paths through an environment to training robots to dance.
- The large elephant in the room of machine learning is the need for a calibrated, easy, efficient, general, method to assign confidence bounds on the outputs of a machine learning model. The work by Ann Lee's group looks very promising in this regard.

Thank you for making this worthwhile!