Exercise Sheet #9

Due: 15.01.2024 11h00

Jan Pawlowski, Tilman Plehn
Jonas Spinner

General Regulations.

- Please hand in your solutions in groups of three people. A mix of attendees from different tutorials is fine. We will not correct submissions from single students.
- Your solutions to theoretical exercises can be either handwritten notes (scanned to `pdf`), typeset using LaTeX, or directly in the jupyter notebook using Markdown.
- For the practical exercises, the data and a skeleton for your jupyter notebook are available at https://github.com/heidelberg-hepml/mlph2023-Exercises. Always provide the (commented) python code as well as the output, and don't forget to explain/interpret the latter, we do not give points for code that does not run. Please hand in both the notebook (`.ipynb`) and an exported pdf. Combine your the pdfs from theoretical and notebook exercises into a single pdf.
- Submit all your files in the Übungsgruppenverwaltung, only once for your group of three. Please list all names and tutorial numbers to simplify things for us.
- Starting from this sheet we put the tag **exam-like** to some exercises to give you a felling for how exercises in the exam might look like.

1 Convergence of Generative Adversarial Networks

In Generative Adversarial Networks (GANs), the generative network G is trained jointly with a discriminator D. They are trained as a min-max game, i.e. the generator tries to minimize an objective, whereas the discriminator tries to maximize it. The most commonly used objective can be rewritten as two seperate loss functions

$$\mathcal{L}_D = \Big\langle -\log D(x) \Big\rangle_{x\sim p_{\text{data}}} + \Big\langle -\log(1-D(x)) \Big\rangle_{x\sim p_{\text{model}}}, \tag{1}$$

$$\mathcal{L}_G = \Big\langle -\log D(x) \Big\rangle_{x\sim p_{\text{model}}} + \lambda\Big(\int dx p_{\text{model}}(x) - 1\Big). \tag{2}$$

(a) (exam-like) Assume that the classifier stays in the optimal configuration throughout the training, i.e. it always satisfies $\delta\mathcal{L}_D/\delta D = 0$ for a given generator p_{model}. Prove that the generator converges to the unique solution $p_{\text{model}} = p_{\text{data}}$. (2 pts)

(b) Argue that this assumption is not justified in practice. Discuss challenges when training GANs. (1 pts)

2 Classifier Reweighting

In Sheet 5, Exercise 1a we have shown that the score $D(x)$ of a classifier trained on the binary cross-entropy (BCE) loss can be used to estimate the likelihood ratio

$$t_{\text{LR}}(x) = \frac{p_1(x)}{p_2(x)} = \frac{D(x)}{1-D(x)}. \tag{3}$$

We now put this knowledge to work and use classifiers to reweight distributions. LHC event generators are usually operated at leading order (LO) precision in perturbation theory. Generating next-to-leading order (NLO) events is possible, but computationally more expensive. One approach to upgrade LO events to NLO events is to *refine* the events, i.e. to change the values of the momenta such that they follow the

NLO distribution. The more common approach is to keep the values of the original momenta, but add a weight to each event that quantifies the probability of this event. These weighted or *reweighted* events can be used to modify the distribution, at the cost of having less statistical power than the original unweighted events. The goal of this exercise is to reweight LO events to the distribution of NLO events.

For this exercise we have small sets of simulated LO and NLO events for top quark pair-production with an additional jet ($t\bar{t}j$), where both top quarks decay hadronically ($t \to bW \to bq\bar{q}'$). The events are given as a 28-dimensional vector, containing the 4-momenta in the ordering $(b_1, q_1, \bar{q}_1', \bar{b}_2, \bar{q}_2, q_2', j)$.

(a) Train a BCE classifier to seperate LO from NLO events. Hint: Use the regularization techniques studied in Sheet 4, Exercise 2e to avoid overfitting. (2 pts)

(b) Argue that the likelihood ratio $t_{\mathrm{LR}}(x)$ in Eq. (3) can be used to reweight from the distribution of LO events to NLO events. Express these weights $w_{\mathrm{LO}\to\mathrm{NLO}}$ in terms of $p_{\mathrm{LO}}, p_{\mathrm{NLO}}$ as well as in terms of $D(x)$. (1 pts)

(c) Use your results from part (b) together with the classifier from part (a) to reweight the LO events to NLO events. Visualize the LO, NLO and reweighted LO distributions for transverse momentum and mass of the leading top quark. Compare the deviation between the reweighted LO distribution and the NLO distribution to the statistical uncertainty within each bin. (2 pts)

3 Event Generation with a Variational Autoencoder

Simulations for LHC physics require a major computational effort at the experiment, and will become more computationally intensive in the coming years. Machine Learning techniques, especially generative neural networks, can be used at many stages of the simulation chain to complement or replace classical techniques.

We will explore the landscape of generative network architectures in a series of exercises on event generation, starting with the Variational Autoencoder. We will focus on generating events of the Drell-Yan process $pp \to Z \to \mu^+\mu^-$ at parton level. This means that we want to generate the 4-momenta of the two muons immediately after they are created in the hard scattering process, not including the parton shower or detector effects. We start today with a Variational Autoencoder (VAE) and then move to Generative Adversarial Networks[1], Normalizing Flows[2] and Diffusion Models[3].

Following the notation in the lecture notes, a variational autoencoder generates events x by sampling latent space vectors $r \sim p_{\mathrm{latent}}(r)$ from a fixed distribution $p_{\mathrm{latent}}(r)$ and then transforming them to phase space vectors/events x using the decoder. We write $x \sim p_{\mathrm{model}}(x|r)$ for this transformation, but in our case the decoder will be a deterministic function $x(r)$ such that $p_{\mathrm{model}}(x|r) = \delta(x(r) - x)$. The distribution $p_{\mathrm{model}}(x|r)$ is related to the distribution of training data $p_{\mathrm{data}}(x)$ via the relation

$$p_{\mathrm{model}}(x|r)p_{\mathrm{latent}}(r) = p(x,r) = p(r|x)p_{\mathrm{data}}(x). \tag{4}$$

We can not directly access the distribution $p(r|x)$ in this relation, but we can approximate it using a second neural network, the encoder $p_{\mathrm{model}}(r|x)$.

Disclaimer You will find poor argeement of the generated distributions with the training data, because VAEs are not a suitable approach for event generation. Nevertheless, we can use them to get familiar with generative networks and the challenges of the dataset. There are other applications where VAEs perform well.

(a) (exam-like) Derive the loss function of the variational autoencoder starting from the KL divergence for the condition $p_{\mathrm{prior}}(r|x) = p_{\mathrm{model}}(r|x)$. Parametrize the encoder $p_{\mathrm{model}}(r|x)$ as an uncorrelated multi-dimensional gaussian with parameters predicted by a neural network, and the prior as an uncorrelated

[1] https://arxiv.org/pdf/1907.03764.pdf
[2] https://arxiv.org/pdf/2110.13632.pdf
[3] https://arxiv.org/pdf/2305.10475.pdf

unit gaussian $p_{\text{prior}}(r|x) = \mathcal{N}_{0,1}(r)$. Explain the relevant steps. Hint: You might have done a similar calculation on sheet 5. (2 pts)

(b) Construct and train a variational autoencoder on the Drell-Yan dataset. Start with a simple setup, taking x to be the two 4-momenta after a simple $x' = (x - \bar{x})/\sigma_x$ preprocessing. Compare the generated distributions for different latent space sizes. Using your knowledge about kinematics, what value do you expect for the optimal bottleneck size? (3 pts)

(c) Let's study some basic properties of the generated events. Compare the mass of the generated particles with the muon mass (105 MeV). Check momentum conservation in transverse direction $\sum_i p_{x,i} = \sum_i p_{y,i} = 0$. Interpret your observations. (1 pts)

(d) Argue why preprocessing is an important ingredient when training neural networks, and in particularly generative networks. Then have a look at the provided optimized preprocessing. What is the shape of the array of preprocessed events? Explain why the individual steps are performed. (2 pts)

```python
def preprocess(events, mean=None, std=None):
    particle1, particle2 = events[:,:4], events[:,4:]
    events_jetcoordinates = np.stack((get_pt(particle1), get_phi(particle1), get_eta(particle1), get_mass(particle1), get_pt(particle2), get_phi(particle2), get_eta(particle2), get_mass(particle2)), axis=-1)
    events_reduced = events_jetcoordinates[:,[0,2,6]]

    events_reduced[:,0] = np.log(events_reduced[:,0] - pt_cut)
    events_reduced[:,1:] = np.arctanh(events_reduced[:,1:] / eta_cut)

    if mean is None or std is None:
        mean = events_reduced.mean(axis=0)
        std = events_reduced.std(axis=0)
    events_reduced = (events_reduced - mean) / std

    assert np.isfinite(events_reduced).all()
    return events_reduced, mean, std

def undo_preprocess(events_reduced, mean, std):
    events_reduced = events_reduced * std + mean

    events_reduced[:,0] = np.exp(events_reduced[:,0]) + pt_cut
    events_reduced[:,[1,2]] = np.tanh(events_reduced[:,[1,2]]) * eta_cut

    pt1, eta1, eta2 = events_reduced.T
    phi1 = np.random.uniform(0, 2*np.pi, events_reduced.shape[0])
    mass1, mass2 = np.ones((2, events_reduced.shape[0])) * 0.105
    px1 = pt1 * np.cos(phi1)
    py1 = pt1 * np.sin(phi1)
    pz1 = pt1 * np.sinh(eta1)
    e1 = np.sqrt(mass1**2 + px1**2 + py1**2 + pz1**2)
    px2 = -px1
    py2 = -py1
    pt2 = pt1
    pz2 = pt2 * np.sinh(eta2)
    e2 = np.sqrt(mass2**2 + px2**2 + py2**2 + pz2**2)
    return np.stack((e1, px1, py1, pz1, e2, px2, py2, pz2), axis=-1)
```

(e) Train a variational autoencoder using the optimized preprocessing and an appropriate bottleneck size. Explain your observations. (2 pts)

(f) Argue why we should now stop training VAEs for this problem. Name two architectures that are more suitable. Can you think of a task where VAEs can perform better? (1 pts)