A Derivation of prediction-only variant loss

$$\mathbb{E}_{p(\mathbf{P})}\left[-\mathbb{E}_{p(\mathbf{y}|\mathbf{P})}[\log q_{\phi_p}(\mathbf{y}|\mathbf{P})]\right] \tag{10}$$

$$= -\mathbb{E}_{p(\mathbf{P})}\left[\mathbb{E}_{p(\mathbf{w},\mathbf{y}|\mathbf{P})}[\log q_{\phi_p}(\mathbf{y}|\mathbf{P})]\right] \tag{11}$$

$$= -\mathbb{E}_{p(\mathbf{P},\mathbf{w})}\left[\log q_{\phi_p}(c(\mathbf{w})|\mathbf{P})\right] \tag{12}$$

$$\mathcal{L}_{Pred}(\boldsymbol{\phi}_p) = -\log q_{\phi_p}(c(\mathbf{w})|\mathbf{P}), \quad \mathbf{P},\mathbf{w} \sim p(\mathbf{P},\mathbf{w}) \tag{13}$$

In line 11, we marginalize out $\mathbf{w}$, and use the fact that $\mathbf{y}$ is deterministic given $\mathbf{w}$.

B Constrained structured output prediction

We consider how to implement the constrained structured output prediction task considered in (for example) [2, 1, 56] in the A-NESI framework. Here, the goal is to learn a mapping of some X to a structured *output* space W, where we have some constraint $c(\mathbf{w})$ that returns 1 if the background knowledge holds, and 0 otherwise. We can model the constraints using $Y = \{0, 1\}$; that is, the 'output' in our problem setup is whether $\mathbf{w}$ satisfies the background knowledge c or not. We give an example of this setting in Figure 6.

Then, we design the inference model as follows. 1) $q_{\phi_p}(y|\mathbf{P})$ is tasked with predicting the probability that randomly sampled outputs $\mathbf{w} \sim p(\mathbf{w}|\mathbf{P})$ will satisfy the background knowledge. 2) $q_{\phi_e}(\mathbf{w}|y = 1, \mathbf{P})$ is an approximate posterior over structured outputs $\mathbf{w}$ that satisfy the background knowledge c.

This setting changes the interpretation of the set W from *unobserved* worlds to *observed* outputs. We will train our perception module using a "strongly" supervised learning loss where $\mathbf{x}, \mathbf{w} \sim \mathcal{D}_{\mathcal{L}}$:

$$\mathcal{L}_{Perc}(\boldsymbol{\theta}) = -\log q_{\phi_e}(\mathbf{w}|y = 1, \mathbf{P} = f_{\boldsymbol{\theta}}(\mathbf{x})). \tag{14}$$

If we also have unlabeled data $\mathcal{D}_U$, we can use the prediction model to ensure the perception model gives high probabilities for worlds that satisfy the background knowledge. This approximates Semantic Loss [56]: Given $\mathbf{x} \sim \mathcal{D}_U$,

$$\mathcal{L}_{SL}(\boldsymbol{\theta}) = -\log q_{\phi_p}(y = 1|\mathbf{P} = f_{\boldsymbol{\theta}}(\mathbf{x})). \tag{15}$$

That is, we have some input $\mathbf{x}$ for which we have no labelled output. Then, we increase the probability that the belief $\mathbf{P}$ the perception module f_θ predicts for $\mathbf{x}$ would sample structured outputs $\mathbf{w}$ that satisfy the background knowledge.

Training the inference model in this setting can be challenging if the problem is very constrained. Then, random samples $\mathbf{P}, \mathbf{w} \sim p(\mathbf{P}, \mathbf{w})$ will usually not satisfy the background knowledge. Since we are only in the case that $y = 1$, we can choose to sample from the inference model $q_{\boldsymbol{\phi}}$ and exploit the symbolic pruner to obtain samples that are guaranteed to satisfy the background knowledge. Therefore, we modify equation 8 to the *on-policy joint matching loss*

$$\mathcal{L}_{Expl}(\mathbf{P}, \boldsymbol{\phi}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{w}|y=1,\mathbf{P})}\left[\left(\log \frac{q_{\boldsymbol{\phi}}(\mathbf{w}, y = 1|\mathbf{P})}{p(\mathbf{w}|\mathbf{P})}\right)^2\right] \tag{16}$$

Here, we incur some sampling bias by not sampling structured outputs from the true posterior, but this bias will reduce as $q_{\boldsymbol{\phi}}$ becomes more accurate. We can also choose to combine the on- and off-policy losses. Another option to make learning easier is using the suggestions of Section 3.2.2: factorize y to make it more fine-grained.

C A-NESI as a Gradient Estimation method

In this appendix, we discuss using the method A-NESI introduced for general gradient estimation [39]. We first define the gradient estimation problem. Consider some neural network $f_{\boldsymbol{\theta}}$ that predicts the parameters $\mathbf{P}$ of a distribution over unobserved variable $\mathbf{z} \in Z$: $p(\mathbf{z}|\mathbf{P} = f_{\boldsymbol{\theta}}(\mathbf{x}))$. This distribution corresponds to the distribution over worlds $p(\mathbf{w}|\mathbf{P})$ in A-NESI. Additionally, assume we have some

deterministic function $g(\mathbf{z})$ that we want to maximize in expectation. This maximization requires estimating the gradient

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p(\mathbf{z}|\mathbf{P}=f_{\boldsymbol{\theta}}(\mathbf{x}))}[g(\mathbf{z})]. \tag{17}$$

Common methods for estimating this gradient are reparameterization [25], which only applies to continuous random variables and differentiable r, and the score function [39, 48] which has notoriously high variance.

Instead, our gradient estimation method learns an *inference model* $q_{\boldsymbol{\phi}}(r|\mathbf{P})$ to approximate the distribution of outcomes $r = g(\mathbf{z})$ for a given $\mathbf{P}$. In A-NESI, this is the prediction network $q_{\boldsymbol{\phi}_p}(\mathbf{y}|\mathbf{P})$ that estimates the WMC problem of Equation 1. Approximating a *distribution* over outcomes is similar to the idea of distributional reinforcement learning [7]. Our approach is general: Unlike reparameterization, we can use inference models in settings with discrete random variables $\mathbf{z}$ and non-differentiable downstream functions g.

We derive the training loss for our inference model similar to that in Section 3.2.1. First, we define the joint on latents $\mathbf{z}$ and outcomes r like the joint process in 6 as $p(r, \mathbf{z}|\mathbf{P}) = p(\mathbf{z}|\mathbf{P}) \cdot \delta_{g(z)}(r)$, where $\delta_{g(z)}(r)$ is the dirac-delta distribution that checks if the output of g on $\mathbf{z}$ is equal to r. Then we introduce a prior over distribution parameters $p(\mathbf{P})$, much like the prior over beliefs in A-NESI. An obvious choice is to use a prior conjugate to $p(\mathbf{z}|\mathbf{P})$. We minimize the expected KL-divergence between $p(r|\mathbf{P})$ and $q_{\boldsymbol{\phi}}(r|\mathbf{P})$:

$$\begin{aligned}
&\mathbb{E}_{p(\mathbf{P})}[D_{KL}(p||q_{\boldsymbol{\phi}})] && (18)\\
=&\mathbb{E}_{p(\mathbf{P})}\left[\mathbb{E}_{p(r|\mathbf{P})}[\log q_{\boldsymbol{\phi}}(r|\mathbf{P})]\right] + C && (19)\\
=&\mathbb{E}_{p(\mathbf{P})}\left[\int_{\mathbb{R}} p(r|\mathbf{P}) \log q_{\boldsymbol{\phi}}(r|\mathbf{P})] dr\right] + C && (20)
\end{aligned}$$

Next, we marginalize over $\mathbf{z}$, dropping the constant:

$$\begin{aligned}
&\mathbb{E}_{p(\mathbf{P})}\left[\int_Z \int_{\mathbb{R}} p(r, \mathbf{z}|\mathbf{P}) \log q_{\boldsymbol{\phi}}(r|\mathbf{P})] dr d\mathbf{z}\right] && (21)\\
=&\mathbb{E}_{p(\mathbf{P})}\left[\int_Z p(\mathbf{z}|\mathbf{P}) \int_{\mathbb{R}} \delta_{g(\mathbf{z})}(r) \log q_{\boldsymbol{\phi}}(r|\mathbf{P})] dr d\mathbf{z}\right] && (22)\\
=&\mathbb{E}_{p(\mathbf{P})}\left[\int_Z p(\mathbf{z}|\mathbf{P}) \log q_{\boldsymbol{\phi}}(g(\mathbf{z})|\mathbf{P})] d\mathbf{z}\right] && (23)\\
=&\mathbb{E}_{p(\mathbf{P},\mathbf{z})}\left[\log q_{\boldsymbol{\phi}}(g(\mathbf{z})|\mathbf{P})\right] && (24)
\end{aligned}$$

This gives us a negative-log-likelihood loss function similar to Equation 7.

$$\mathcal{L}_{Inf}(\boldsymbol{\phi}) = -\log q_{\boldsymbol{\phi}}(g(\mathbf{z})|\mathbf{P}), \quad \mathbf{P}, \mathbf{z} \sim p(\mathbf{z}, \mathbf{P}) \tag{25}$$

where we sample from the joint $p(\mathbf{z}, \mathbf{P}) = p(\mathbf{P})p(\mathbf{z}|\mathbf{P})$.

We use a trained inference model to get gradient estimates:

$$\nabla_{\mathbf{P}} \mathbb{E}_{p(\mathbf{z}|\mathbf{P})}[g(\mathbf{z})] \approx \nabla_{\mathbf{P}} \mathbb{E}_{q_{\boldsymbol{\phi}}(r|\mathbf{P})}[r] \tag{26}$$

We use the chain rule to update the parameters $\boldsymbol{\theta}$. This requires a choice of distribution $q_{\boldsymbol{\phi}}(r|\mathbf{P})$ for which computing the mean $\mathbb{E}_{q_{\boldsymbol{\phi}}(r|\mathbf{P})}[r]$ is easy. The simplest option is to parameterize $q_{\boldsymbol{\phi}}$ with a univariate normal distribution. We predict the mean and variance using a neural network with parameters $\boldsymbol{\phi}$. For example, a neural network $m_{\boldsymbol{\phi}}$ would compute $\mu = m_{\boldsymbol{\phi}}(\mathbf{P})$. Then, the mean parameter is the expectation on the right-hand side of Equation 26. The loss function for $f_{\boldsymbol{\theta}}$ with this parameterization is:

$$\mathcal{L}_{NN}(\boldsymbol{\theta}) = -m_{\boldsymbol{\phi}}(f_{\boldsymbol{\theta}}(\mathbf{x})), \quad \mathbf{x} \sim \mathcal{D} \tag{27}$$

Interestingly, like A-NESI, this gives zero-variance gradient estimates! Of course, bias comes from the error in the approximation of $q_{\boldsymbol{\phi}}$.

Like A-NESI, we expect the success of this method to rely on the ease of finding a suitable prior over $\mathbf{P}$ to allow proper training of the inference model. See the discussion in Section 3.2.3. We also expect that, like in A-NESI, it will be easier to train the inference model if the output $r = g(\mathbf{z})$ is structured instead of a single scalar. We refer to Section 3.2.2 for this idea. Other challenges might be stochastic and noisy output measurements of r and non-stationarity of g, for instance, when training a VAE [25].

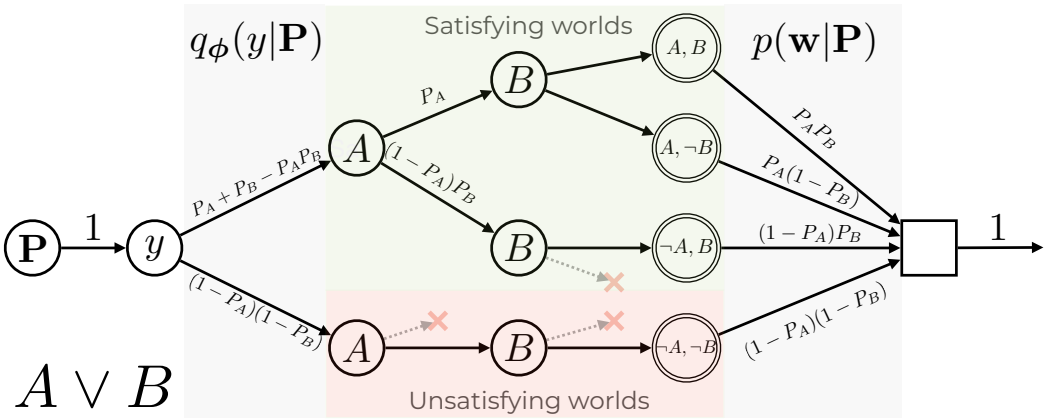

Figure 6: The tree flow network corresponding to weighted model counting on the formula $A \vee B$. Following edges upwards means setting the corresponding binary variable to true (and to false by following edges downwards). We first choose probabilities for the propositions A and B, then choose whether we want to sample a world that satisfies the formula $A \vee B$. $y = 1$ is the WMC of $A \vee B$, and equals its outgoing flow $P_A + P_B - P_A P_B$. Terminal states (with two circles) represent choices of the binary variables A and B. These are connected to a final sink node, corresponding to the prior over worlds $p(\mathbf{w}|\mathbf{P})$. The total ingoing and outgoing flow to this network is 1, as we deal with normalized probability distributions p and q_ϕ.

D A-NeSI and GFlowNets

A-NeSI is heavily inspired by the theory of GFlowNets [8, 9], and we use this theory to derive our loss function. In the current section, we discuss these connections and the potential for future research by taking inspiration from the GFlowNet literature. In this section, we will assume the reader is familiar with the notation introduced in [9] and refer to this paper for the relevant background.

D.1 Tree GFlowNet representation

The main intuition is that we can treat the inference model q_ϕ in Equation 3 as a 'trivial' GFlowNet. We refer to Figure 6 for an intuitive example. It shows what a flow network would look like for the formula $A \vee B$. We take the reward function $R(\mathbf{w}, \mathbf{y}) = p(\mathbf{w}, \mathbf{y})$. We represent states s by $s = (\mathbf{P}, \mathbf{y}_{1:i}, \mathbf{w}_{1:j})$, that is, the belief $\mathbf{P}$, a list of some dimensions of the output instantiated with a value and a list of some dimensions of the world assigned to some value. Actions a set some value to the next output or world variable, i.e., $A(s) = Y_{i+1}$ or $A(s) = W_{j+1}$.

Note that this corresponds to a flow network that is a tree everywhere but at the sink since the state representation conditions on the whole trajectory observed so far. We demonstrate this in Figure 6. We assume there is some fixed ordering on the different variables in the world, which we generate the value of one by one. Given this setup, Figure 6 shows that the branch going up from the node y corresponds to the regular weighted model count (WMC) introduced in Equation 1.

The GFlowNet forward distribution P_F is q_ϕ as defined in Equation 3. The backward distribution P_B is $p(\mathbf{w}, \mathbf{y}|\mathbf{P})$ at the sink node, which chooses a terminal node. Then, since we have a tree, this determines the complete trajectory from the terminal node to the source node. Thus, at all other states, the backward distribution is deterministic. Since our reward function $R(\mathbf{w}, \mathbf{y}, \mathbf{P}) = p(\mathbf{w}, \mathbf{y}|\mathbf{P})$ is normalized, we trivially know the partition function $Z(\mathbf{P}) = \sum_{\mathbf{w}} \sum_{\mathbf{y}} R(\mathbf{w}, \mathbf{y}|\mathbf{P}) = 1$.

D.2 Lattice GFlowNet representation

Our setup of the generative process assumes we are generating each variable in the world in some order. This is fine for some problems like MNISTAdd, where we can see the generative process as 'reading left to right'. For other problems, such as Sudoku, the order in which we would like to generate the symbols is less obvious. Would we generate block by block? Row by row? Column by column? Or is the assumption that it needs to be generated in some fixed order flawed by itself?

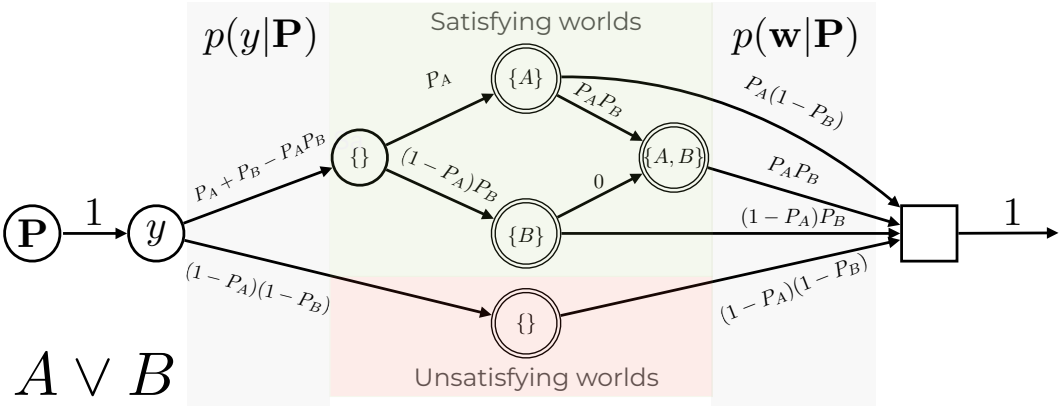

Figure 7: The lattice flow network corresponding to weighted model counting on the formula $A \vee B$. In this representation, nodes represent sets of included propositions. Terminal states represent sets of random variables such that $A \vee B$ is true given $y = 1$, or false otherwise.

In this section, we consider a second GFlowNet representation for the inference model that represents states using sets instead of lists. We again refer to Figure 7 for the resulting flow network of this representation for $A \vee B$. We represent states using $s = (\mathbf{P}, \{y_i\}_{i \in I_Y}, \{w_i\}_{i \in I_W})$, where $I_Y \subseteq \{1, ..., k_Y\}$ and $I_W \subseteq \{1, ..., k_W\}$ denote the set of variables for which a value is chosen. The possible actions from some state correspond to $A(s) = \bigcup_{i \notin I_W} W_i$ (and analogous for when $\mathbf{y}$ is not yet completely generated). For each variable in W for which we do not have the value yet, we add its possible values to the action space.

With this representation, the resulting flow network is no longer a tree but a DAG, as the order in which we generate the different variables is now different for every trajectory. What do we gain from this? When we are primarily dealing with categorical variables, the two gains are 1) we no longer need to impose an ordering on the generative process, and 2) it might be easier to implement parameter sharing in the neural network that predicts the forward distributions, as we only need a single set encoder that can be reused throughout the generative process.

However, the main gain of the set-based approach is when worlds are all (or mostly) binary random variables. We illustrate this in Figure 7. Assume $W = \{0, 1\}^{k_W}$. Then we can have the following state and action representations: $s = (\mathbf{P}, \mathbf{y}, I_W)$, where $I_W \subseteq \{1, ..., k_W\}$ and $A(s) = \{1, ..., k_W\} \setminus I_W$. The intuition is that I_W contains the set of all binary random variables that are set to 1 (i.e., true), and $\{1, ..., k_W\} \setminus I_W$ is the set of variables set to 0 (i.e., false). The resulting flow network represents a *partial order* over the set of all subsets of $\{1, ..., k_W\}$, which is a *lattice*, hence the name of this representation.

With this representation, we can significantly reduce the size and computation of the flow network required to express the WMC problem. As an example, compare Figures 6 and 7, which both represent the WMC of the formula $A \vee B$. We no longer need two nodes in the branch $y = 0$ to represent that we generate A and B to be false, as the initial empty set $\{\}$ already implies they are. This will save us two nodes. Similarly, we can immediately stop generating at $\{A\}$ and $\{B\}$, and no longer need to generate the other variable as false, which also saves a computation step.

While this is theoretically appealing, the three main downsides are 1) P_B is no longer trivial to compute; 2) we have to handle the fact that we no longer have a tree, meaning there is no longer a unique optimal P_F and P_B; and 3) parallelization becomes much trickier. We leave exploring this direction in practice for future work.

E Analyzing the Joint Matching Loss

This section discusses the loss function we use to train the joint variant in Equation 8. We recommend interested readers first read Appendix D.1. Throughout this section, we will refer to $p := p(\mathbf{w}, \mathbf{y}|\mathbf{P})$ (Equation 5) and $q := q_{\phi}(\mathbf{w}, \mathbf{y}|\mathbf{P})$ (Equation 2). We again refer to [9] for notational background.

E.1 Trajectory Balance

We derive our loss function from the recently introduced Trajectory Balance loss for GFlowNets, which is proven to approximate the true Markovian flow when minimized. This means sampling from the GFlowNet allows sampling in proportion to reward $R(s_n) = p$. The Trajectory Balance loss is given by

$$\mathcal{L}(\tau) = \left(\log \frac{F(s_0)\prod_{t=1}^{n} P_F(s_t|s_{t-1})}{R(s_n)\prod_{t=1}^{n} P_B(s_{t-1}|s_t)}\right)^2, \tag{28}$$

where s_0 is the source state, in our case $\mathbf{P}$, and s_n is some terminal state that represents a full generation of $\mathbf{y}$ and $\mathbf{w}$. In the tree representation of GFlowNets for inference models (see Appendix D.1), this computation becomes quite simple:

1. $F(s_0) = 1$, as $R(s_n) = p$ is normalized;
2. $\prod_{t=1}^{n} P_F(s_t|s_{t-1}) = q$: The forward distribution corresponds to the inference model $q_{\boldsymbol{\phi}}(\mathbf{w}, \mathbf{y}|\mathbf{P})$;
3. $R(s_n) = p$, as we define the reward to be the true joint probability distribution $p(\mathbf{w}, \mathbf{y}|\mathbf{P})$;
4. $\prod_{t=1}^{n} P_B(s_{t-1}|s_t) = 1$, since the backward distribution is deterministic in a tree.

Therefore, the trajectory balance loss for (tree) inference models is

$$\mathcal{L}(\mathbf{P}, \mathbf{y}, \mathbf{w}) = \left(\log \frac{q}{p}\right)^2 = \left(\log q - \log p\right)^2, \tag{29}$$

i.e., the term inside the expectation of the joint matching loss in Equation 8. This loss function is stable because we can sum the individual probabilities in log-space.

A second question might then be how we obtain 'trajectories' $\tau = (\mathbf{P}, \mathbf{y}, \mathbf{w})$ to minimize this loss over. The paper on trajectory balance [32] picks τ *on-policy*, that is, it samples τ from the forward distribution (in our case, the inference model $q_{\boldsymbol{\phi}}$). We discussed when this might be favorable in our setting in Appendix B (Equation 16). However, the joint matching loss as defined in Equation 8 is *off-policy*, as we sample from p and not from $q_{\boldsymbol{\phi}}$.

E.2 Relation to common divergences

These questions open quite some design space, as was recently noted when comparing the trajectory balance loss to divergences commonly used in variational inference [33]. Redefining $P_F = q$ and $P_B = p$, the authors compare the trajectory balance loss with the KL-divergence and the reverse KL-divergence and prove that

$$\nabla_{\boldsymbol{\phi}} D_{KL}(q||p) = \frac{1}{2}\mathbb{E}_{\tau\sim q}[\nabla_{\boldsymbol{\phi}}\mathcal{L}(\tau)]. \tag{30}$$

That is, the *on*-policy objective minimizes the *reverse* KL-divergence between p and q. We do not quite find such a result for the *off*-policy version we use for the joint matching loss in Equation 8:

$$\nabla_{\boldsymbol{\phi}} D_{KL}(p||q) = -\mathbb{E}_{\tau\sim p}[\nabla_{\boldsymbol{\phi}} \log q] \tag{31}$$

$$\mathbb{E}_{\tau\sim p}[\nabla_{\boldsymbol{\phi}}\mathcal{L}(\tau)] = -2\mathbb{E}_{\tau\sim p}[(\log p - \log q)\nabla_{\boldsymbol{\phi}} \log q] \tag{32}$$

So why do we choose to minimize the joint matching loss rather than the (forward) KL divergence directly? This is because, as is clear from the above equations, it takes into account how far the 'predicted' log-probability $\log q$ currently is from $\log p$. That is, given a sample τ, if $\log p < \log q$, the joint matching loss will actually *decrease* $\log q$. Instead, the forward KL will increase the probability for every sample it sees, and whether this particular sample will be too likely under q can only be derived through sampling many trajectories.

Furthermore, we note that the joint matching loss is a 'pseudo' f-divergence with $f(t) = t\log^2 t$ [33]. It is not a true f-divergence since $t\log^2 t$ is not convex. A related well-known f-divergence is the Hellinger distance given by

$$H^2(p||q) = \frac{1}{2}\mathbb{E}_{\tau\sim p}[(\sqrt{p} - \sqrt{q})^2]. \tag{33}$$

This divergence similarly takes into account the distance between p and q in its derivatives through squaring. However, it is much less stable than the joint matching loss since both p and q are computed by taking the product over many small numbers. Computing the square root over this will be much less numerically stable than taking the logarithm of each individual probability and summing.

Finally, we note that we minimize the on-policy joint matching $\mathbb{E}_{q_{\boldsymbol{\phi}}}[(\log p - \log q)^2]$ by taking derivatives $\mathbb{E}_{q_{\boldsymbol{\phi}}}[\nabla_{\boldsymbol{\phi}}(\log p - \log q)^2]$. This is technically not minimizing the joint matching, since it ignores the gradient coming from sampling from $q_{\boldsymbol{\phi}}$.

F Dirichlet prior

This section describes how we fit the Dirichlet prior $p(\mathbf{P})$ used to train the inference model. During training, we keep a dataset of the last 2500 observations of $\mathbf{P} = f_{\boldsymbol{\theta}}(\mathbf{x})$. We have to drop observations frequently because $\boldsymbol{\theta}$ changes during training, meaning that the empirical distribution over $\mathbf{P}$s changes as well.

We perform an MLE fit on k_W independent Dirichlet priors to get parameters $\boldsymbol{\alpha}$ for each. The log-likelihood of the Dirichlet distribution cannot be found in closed form [38]. However, since its log-likelihood is convex, we run ADAM [24] for 50 iterations with a learning rate of 0.01 to minimize the negative log-likelihood. We refer to [38] for details on computing the log-likelihood and alternative options. Since the Dirichlet distribution accepts positive parameters, we apply the softplus function on an unconstrained parameter during training. We initialize all parameters at 0.1.

We added L2 regularization on the parameters. This is needed because at the beginning of training, all observations $\mathbf{P} = f_{\boldsymbol{\theta}}(\mathbf{x})$ represent uniform beliefs over digits, which will all be nearly equal. Therefore, fitting the Dirichlet on the data will give increasingly higher parameter values, as high parameter values represent low-entropy Dirichlet distributions that produce uniform beliefs. When the Dirichlet is low-entropy, the inference models learn to ignore the input belief $\mathbf{P}$, as it never changes. The L2 regularization encourages low parameter values, which correspond to high-entropy Dirichlet distributions.

G Designing symbolic pruners

We next discuss four high-level approaches for designing the optional symbolic pruner:

1. **Mathematically derive efficient solvers.** For simple problems, we could mathematically derive an exact solver. One example of an efficient symbolic pruner, along with a proof for exactness, is given for Multi-digit MNISTAdd in Appendix H. This pruner is linear-time. However, for most problems we expect the pruner to be much more computationally expensive.
2. **Use SAT-solvers.** Add the sampled symbols $\mathbf{y}$ and $\mathbf{w}_{1:i}$ to a CNF-formula, and ask an SAT-solver if there is an extension $\mathbf{w}_{i+1:k_W}$ that satisfies the CNF-formula. SAT-solvers are a general approach that will work with every function c, but using them comes at a cost.

   The first is that we would require grounding the logical representation of the problem. Furthermore, to do SAT-solving, we have to solve a linear amount of NP-hard problems. However, competitive SAT solvers can deal with substantial problems due to years of advances in their design [6], and a linear amount of NP-hard calls is a lower complexity class than #P hard. Using SAT-solvers will be particularly attractive in problem settings where safety and verifiability are critical.
3. **Prune with local constraints.** In many structured prediction tasks, we can use local constraints of the symbolic problem to prune paths that are guaranteed to lead to branches that can never create possible worlds. However, local constraints do not guarantee that each non-pruned path contains a possible world, but this does not bias the inference model, as the neural network will (eventually) learn when an expansion would lead to an unsatisfiable state.

   One example is the shortest path problem, where we can filter out directions that would lead outside the $N \times N$ grid, or that would create cycles. However, this just ensures we find *a* path, but not that it is the shortest one!

4. **Learn the pruner.** Finally, we can learn the pruner, that is, we can train a neural network to learn satisfiability checking. One possible approach is to reuse the inference model trained on the belief $\mathbf{P}$ that uniformly distributes mass over all worlds.

   Learned pruners will be as quick as regular inference models, but are less accurate than symbolic pruners and will not guarantee that constraints are always satisfied during test-time. We leave experimenting with learning the pruner for future work.

H MNISTAdd Symbolic Pruner

In this section, we describe a symbolic pruner for the Multi-digit MNISTAdd problem, which we compute in time linear to N. Note that $\mathbf{w}_{1:N}$ represents the first number and $\mathbf{w}_{N+1:2N}$ the second. We define $n_1 = \sum_{i=1}^{N} w_i \cdot 10^{N-i-1}$ and $n_2 = \sum_{i=1}^{N} w_{N+i} \cdot 10^{N-i-1}$ for the integer representations of these numbers, and $y = \sum_{i=1}^{N+1} y_i \cdot 10^{N-i}$ for the sum label encoded by $\mathbf{y}$. We say that partial generation $\mathbf{w}_{1:k}$ has a *completion* if there is a $\mathbf{w}_{k+1:2N} \in \{0, \ldots, 9\}^{2N-k}$ such that $n_1 + n_2 = y$.

Proposition H.1. *For all $N \in \mathbb{N}$, $\mathbf{y} \in \{0,1\} \times \{0, \ldots, 9\}^N$ and partial generation $\mathbf{w}_{1:k-1} \in \{0, \ldots, 9\}^k$ with $k \in \{1, \ldots, 2N\}$, the following algorithm rejects all w_k for which $\mathbf{w}_{1:k}$ has no completions, and accepts all w_k for which there are:*

- *$k \leq N$: Let $l_k = \sum_{i=1}^{k+1} y_k \cdot 10^{k+1-i}$ and $p_k = \sum_{i=1}^{k} w_k \cdot 10^{k-i}$. Let $S = 1$ if $k = N$ or if the $(k+1)$th to $(N+1)$th digit of y are all 9, and $S = 0$ otherwise. We compute two boolean conditions for all $w_k \in \{0, \ldots, 9\}$:*

$$0 \leq l_k - p_k \leq 10^k - S \tag{34}$$

  We reject all w_k for which either condition does not hold.

- *$k > N$: Let $n_2 = y - n_1$. We reject all $w_k \in \{0, \ldots, 9\}$ different from $w_k = \lfloor \frac{n_2}{10^{N-k-1}} \rfloor \bmod 10$, and reject all w_k if $n_2 < 0$ or $n_2 \geq 10^N$.*

Proof. For $k > N$, we note that n_2 is fixed given y and n_1 through linearity of summation, and we only consider $k \leq N$. We define $a_k = \sum_{i=k+2}^{N+1} y_i \cdot 10^{N+1-i}$ as the sum of the remaining digits of y. We note that $y = l_k \cdot 10^{N-k} + a_k$.

Algorithm rejects w_k without completions We first show that our algorithm only rejects w_k for which no completion exists. We start with the constraint $0 \leq l_k - p_k$, and show that whenever this constraint is violated (i.e., $p_k > l_k$), $\mathbf{w}_{1:k}$ has no completion. Consider the smallest possible completion of $\mathbf{w}_{k+1:N}$: setting each to 0. Then $n_1 = p_k \cdot 10^{N-k}$. First, note that

$$10^{N-k} > 10^{N-k} - 1 \geq a_k$$

Next, add $l_k \cdot 10^{N-k}$ to both sides

$$(l_k + 1) \cdot 10^{N-k} > l_k \cdot 10^{N-k} + a_k = y$$

By assumption, p_k is an integer upper bound of l_k and so $p_k \geq l_k + 1$. Therefore,

$$n_1 = p_k \cdot 10^{N-k} > y$$

Since n_1 is to be larger than y, n_2 has to be negative, which is impossible.

Next, we show the necessity of the second constraint. Assume the constraint is unnecessary, that is, $l_k > p_k + 10^k - S$. Consider the largest possible completion $\mathbf{w}_{k+1:N}$ by setting each to 9. Then

$$\begin{aligned} n_1 &= p_k \cdot 10^{N-k} + 10^{N-k} - 1 \\ &= (p_k + 1) \cdot 10^{N-k} - 1 \end{aligned}$$

We take n_2 to be the maximum value, that is, $n_2 = 10^N - 1$. Therefore,

$$n_1 + n_2 = 10^N - (p_k + 1) \cdot 10^{N-k} - 2$$

We show that $n_1 + n_2 < y$. Since we again have an integer upper bound, we know $l_k \geq p_k + 10^k - S + 1$. Therefore,

$$\begin{aligned} y &\geq (p_k + 1 + 10^k - S)10^{N-k} + a_k \\ &\geq n_1 + n_2 + 2 - S \cdot 10^{N-k} + a_k \end{aligned}$$

There are two cases.

- $S = 0$. Then $a_k < 10^{N-k} - 1$, and so

$$y \geq n_1 + n_2 + 2 + a_k > n_1 + n_2.$$

- $S = 1$. Then $a_k = 10^{N-k} - 1$, and so

$$y \geq n_1 + n_2 + 1 > n_1 + n_2.$$

Algorithm accepts w_k with completions Next, we show that our algorithm only accepts w_k with completions. Assume Equation 34 holds, that is, $0 \leq l_k - p_k \leq 10^k - S$. We first consider all possible completions of $\mathbf{w}_{1:k}$. Note that $p_k \cdot 10^{N-k} \leq n_1 \leq p_k \cdot 10^{N-k} + 10^{N-k} - 1$ and $0 \leq n_2 \leq 10^N - 1$, and so

$$p_k \cdot 10^{N-k} \leq n_1 + n_2 \leq (p_k + 1) \cdot 10^{N-k} + 10^N - 2.$$

Similarly,

$$l_k \cdot 10^{N-k} \leq y \leq (l_k + 1) \cdot 10^{N-k} - 1.$$

By assumption, $p_k \leq l_k$, so $p_k \cdot 10^{N-k} \leq l_k \cot 10^{N-k}$. For the upper bound, we again consider two cases. We use the second condition $l_k \leq 10^k + p_k - S$:

- $S = 0$. Then (since there are no trailing 9s),

$$\begin{aligned} y &\leq (l_k + 1) \cdot 10^{N-k} - 2 \\ &\leq (10^k + p_k + 1) \cdot 10^{N-k} - 1 \\ &= (p_k + 1) \cdot 10^{N-k} + 10^N - 2. \end{aligned}$$

- $S = 1$. Then with trailing 9s,

$$\begin{aligned} y &= (l_k + 1) \cdot 10^{N-k} - 1 \\ &\leq (10^k + p_k) \cdot 10^{N-k} - 1 \\ &= p_k \cdot 10^{N-k} + 10^N - 1 \\ &\leq (p_k + 1) \cdot 10^{N-k} + 10^N - 2, \end{aligned}$$

since $10^{N-k} \geq 1$.

Therefore,

$$p_k \cdot 10^{N-k} \leq y \leq (p_k + 1) \cdot 10^{N-k} + 10^N - 2$$

and so there is a valid completion.

□

I Details of the experiments

I.1 Multi-digit MNISTAdd

Like [35, 36], we take the MNIST [29] dataset and use each digit exactly once to create data. We follow [35] and require more unique digits for increasing N. Therefore, the training dataset will be of size $60000/2N$ and the test dataset of size $10000/2N$.

I.1.1 Hyperparameters

We performed hyperparameter tuning on a held-out validation set by splitting the training data into 50.000 and 10.000 digits, and forming the training and validation sets from these digits. We progressively increased N from $N = 1$, $N = 3$, $N = 4$ to $N = 8$ during tuning to get improved insights into what hyperparameters are important. The most important parameter, next to learning rate, is the weight of the L2 regularization on the Dirichlet prior's parameters which should be very high. We used ADAM [24] throughout. We ran each experiment 10 times to estimate average accuracy, where each run computes 100 epochs over the training dataset. We used Nvidia RTX A4000s GPUs and 24-core AMD EPYC-2 (Rome) 7402P CPUs.

Parameter name	Value	Parameter name	Value
Learning rate	0.001	Prior learning rate	0.01
Epochs	100	Amount beliefs prior	2500
Batch size	16	Prior initialization	0.1
# of samples	600	Prior iterations	50
Hidden layers	3	L2 on prior	900.000
Hidden width	800		

Table 4: Final hyperparameters for the multi-digit MNISTAdd task.

Parameter name	Value	Parameter name	Value
Perception Learning rate	0.00055	Prior learning rate	0.0029
Inference learning rate	0.003	Amount beliefs prior	2500
Batch size	20	Prior initialization	0.02
# of samples	500	Prior iterations	18
Hidden layers	2	L2 on prior	2.500.000
Hidden width	100		
Epochs	5000	Pretraining epochs	50

Table 5: Final hyperparameters for the visual Sudoku puzzle classification task.

We give the final hyperparameters in Table 4. We use this same set of hyperparameters for all N. # of samples refers to the number of samples we used to train the inference model in Algorithm 1. For simplicity, it is also the beam size for the beam search at test time. The hidden layers and width refer to MLP that computes each factor of the inference model. There is no parameter sharing. The perception model is fixed in this task to ensure performance gains are due to neurosymbolic reasoning (see [34]).

I.1.2 Other methods

We compare with multiple neurosymbolic frameworks that previously tackled the MNISTAdd task. Several of those are probabilistic neurosymbolic methods: DeepProbLog [34], DPLA* [36], NeurASP [57] and NeuPSL [43]. We also compare with the fuzzy logic-based method LTN [5] and with Embed2Sym [3] and DeepStochLog [55]. We take results from the corresponding papers, except for DeepProbLog and NeurASP, which are from [36], and LTN from [43][1]. We reran Embed2Sym, averaging over 10 runs since its paper did not report standard deviations. We do not compare DPLA* with pre-training because it tackles an easier problem where part of the digits is labeled.

Embed2Sym [3] uses three steps to solve Multi-digit MNISTAdd: First, it trains a neural network to embed each digit and to predict the sum from these embeddings. It then clusters the embeddings and uses symbolic reasoning to assign clusters to labels. A-NESI has a similar neural network architecture, but we train the prediction network on an objective that does not require data. Furthermore, we train A-NESI end-to-end, unlike Embed2Sym. For Embed2Sym, we use **symbolic prediction** to refer to Embed2Sym-NS, and **neural prediction** to refer to Embed2Sym-FN, which also uses a prediction network but is only trained on the training data given and does not use the prior to sample additional data

We believe the accuracy improvements compared to DeepProbLog to come from hyperparameter tuning and longer training times, as A-NESI approximates DeepProbLog's semantics.

I.2 Visual Sudoku Puzzle Classification

I.2.1 A-NeSI definition

First, we will be more precise with the model we use. We see $\mathbf{x}$ as a $N \times N \times 784$ grid of MNIST images, and beliefs $\mathbf{P}$ as an $N \times N \times N$ grid of distributions over N options (for example, for a 4×4 puzzle, we have to fill in the digits $\{0, 1, 2, 3\}$). For each grid index i, j, the world variable $\mathbf{w}_{i,j}$ corresponds to the digit at location (i, j). For correct puzzles, we know that the digits at location

[1]We take the results of LTN from [43] because [5] averages over the 10 best outcomes of 15 runs and overestimates its average accuracy.

(i, j) and location (i', j') need to be different if $i = i'$, $j = j'$ or if (i, j) is in the same block as (i', j'). For each pair of locations $(i, j), (i', j')$ for which this holds, we have a dimension in $\mathbf{y}$ that indicates if the digits at that grid location are indeed different. The symbolic function c considers each such pair and returns the corresponding $\mathbf{y}$.

For the perception model, we use a *single* MLP $f_{\boldsymbol{\theta}}$ for each of these pairs. That is, for each pair that should be different, we compute $q_{\boldsymbol{\phi}_p}(\mathbf{y}_k|\mathbf{P}) = f_{\boldsymbol{\phi}_p}(\mathbf{P}_{i,j}, \mathbf{P}_{i',j'})$. This introduces the independence assumption that the digits at location (i, j) and location (i', j') being different does not depend on the digits at other locations. This is, clearly, wrong. However, we found it is sufficient to accurately train the perception model.

When training the prediction model, since we sample $\mathbf{P}$ from a Dirichlet prior that assumes the different dimensions of $\mathbf{w}$ are independent, the grid of digits $\mathbf{w}$ are highly unlikely to represent actual sudoku's: There are about $10^2 1$ Sudoku's and $9^8 1$ possible grids (for 9×9 Sudoku's). However, it is quite likely to sample two digits that are different, and this is enough to train the prediction model.

I.2.2 Hyperparameters and other methods

We used the Visual Sudoku Puzzle Classification dataset from [4]. This dataset offers many options: We used the simple generator strategy with 200 training puzzles (100 correct, 100 incorrect). We took a corrupt chance of 0.50, and used the dataset with 0 overlap (this means each MNIST digit can only be used once in the 200 puzzles). There are 11 splits of this dataset, independently generated. We did hyperparameter tuning on the 11th split of the 9×9 dataset. We used the other 10 splits to evaluate the results, averaging results over runs of each of those.

The final hyperparameters are reported in Table 5. The 5000 epochs took on average 20 minutes for the 4×4 puzzles, and 38 minutes for the 9×9 puzzles on a machine with a single NVIDIA RTX A4000. The first 50 epochs we only trained the prediction model to ensure it provides reasonably accurate gradients.

While [4] used NeuPSL, we had to rerun it to get accuracy results and results on 9×9 grids.

We implemented the exact inference methods using what can best be described as Semantic Loss [56]. We encoded the rules of Sudoku described at the beginning of this section as a CNF, and used PySDD (`https://github.com/wannesm/PySDD`) to compile this to an SDD [27]. This was almost instant for the 4×4 CNF, but we were not able to compile the 9×9 CNF within 4 hours, hence why we report a timeout for exact inference. To implement Semantic Loss, we modified a PyTorch implementation available at `https://github.com/lucadiliello/semantic-loss-pytorch` to compute in log-space for numerically stable behavior. This modified version is included in our own repository. We ran this method for 300 epochs with a learning rate of 0.001. We ran this method for fewer epochs because it is much slower than A-NeSI even on 4×4 puzzles (1 hour and 16 minutes for those 300 epochs, so about 63 times as slow).

For both A-NESI and exact inference, we train the perception model on *correct* puzzles by maximizing the probability that $p(y = 1|\mathbf{P})$. A-NESI does this by maximizing $\log q_{\boldsymbol{\phi}_p}(\mathbf{y} = \mathbf{1}|\mathbf{P})$, while Semantic Loss uses PSDDs to exactly compute $\log p(y = 1|\mathbf{P})$. For *incorrect* puzzles, there is not much to be gained since we cannot assume anything about $\mathbf{y}$. Still, for both methods we added the loss $log(1 - p(y = 1|\mathbf{P}))$ for the incorrect puzzles.

I.3 Warcraft Visual Path Planning

I.3.1 A-NeSI definition

We see $\mathbf{x}$ as a $N \times N \times 3 \times 8 \times 8$ real tensor: The first two dimensions indicate the different grid cells, the third dimension indicates the RGB color channels, and the last two dimensions indicate the pixels in each grid cell. The world $\mathbf{w}$ is an $N \times N$ grid of integers, where each integer indexes five different costs for traversing that cell. The five costs are $[0.8, 1.2, 5.3, 7.7, 9.2]$, and correspond to the five possible costs in the Warcraft game. The symbolic function c takes the grid of costs $\mathbf{w}$ and returns the shortest path from the top left corner $(1, 1)$ to the bottom right corner (N, N) using Dijkstra's algorithm. We encode the shortest path as a sequence of actions to take in the grid, where each action is one of the eight (inter-)cardinal directions (down, down-right, right, etc.). The sequence is padded with the do-not-move action to allow for batching.

For the perception model, we use a single small CNN $f_{\boldsymbol{\theta}}$ for each of the $N \times N$ grid cells. That is, for each grid cell, we compute $\mathbf{P}_{i,j} = f_{\boldsymbol{\theta}}(\mathbf{x}_{i,j})$. The CNN has a single convolutional layer with 6 output dimensions, a 2×2 maxpooling layer, a hidden layer of 24×84 and a softmax output layer of 24×5, with ReLU activations.

The prediction model is a ResNet18 model [21], with an output layer of 8 options. It takes an image of size $6 \times N \times N$ as input. The first 5 channels are the probabilities $\mathbf{P}_{i,j}$, and the last channel indicates the current position in the grid. The 8 output actions correspond to the 8 (inter-)cardinal directions. We apply symbolic pruning (Section 3.3) to prevent actions that would lead outside the grid or return to a previously visited grid cell. We pretrained the prediction model by repeating Algorithm 1 on a fixed prior using 185.000 iterations (200 samples each) for 12×12, and 375.000 iterations (20 samples) for 30×30. We used fewer examples per iteration for the larger grid because Dijkstra's algorithm became a computational bottleneck. This took 23 hours for 12×12 and 44 hours for 30×30. Both used a learning rate of $2.5 \cdot 10^{-4}$ and an independent fixed Dirichlet prior with $\alpha = 0.005$. Standard deviations over 10 runs are reported over multiple perception model training runs on the same frozen pretrained prediction model. We trained the perception model for only 1 epoch using a learning rate of 0.0084 and a batch size of 70.

I.3.2 Other methods

We compare to SPL [1] and I-MLE [40]. SPL is also a probabilistic neurosymbolic method, and uses exact inference. Its setup is quite different from ours, however. Instead of using Dijkstra's algorithm, it trains a ResNet18 to predict the shortest path end-to-end, and uses symbolic constraints to ensure the output of the ResNet18 is a valid path. Furthermore, it only considers the 4 cardinal directions instead of all 8 directions. SPL only reports a single training rule in their paper.

I-MLE is more similar to our setup and also uses Dijkstra's algorithm. It uses the first five layers of a ResNet18 to predict the cell costs given the input image. One big difference to our setup is that I-MLE uses continuous costs instead of a choice out of five discrete costs. This may be easier to optimize, as it gives the model more freedom to move costs around. I-MLE is reported using the numbers from the paper, and averages over 5 runs.

To be able to compare to another scalable baseline with the same setup, we added REINFORCE using the leave-one-out baseline (RLOO, [28]), implemented using the PyTorch library Storchastic [52]. It uses the same small CNN to predict a distribution over discrete cell costs, then takes 10 samples, and feeds those through Dijkstra's to get the shortest path. Here, we represent the shortest path as an $N \times N$ grid of zeros and ones. The reward function for RLOO is the Hamming loss between the predicted path and the ground truth path. We use a learning rate of $5 \cdot 10^{-4}$ and a batch size of 70. We train for 10 epoch and report the standard deviation over 10 runs. We note that RLOO gets quite expensive for 30×30 grids, as it needs 10 Dijkstra calls per training sample.

Finally, we experimented with running A-NESI and RLOO simultaneously. We ran this for 10 epochs with a learning rate of $5 \cdot 10^{-4}$ and a batch size of 70. We report the standard deviation over 10 runs.