Appendix for the Paper "A Meta-Learning Approach for Graph Representation Learning in Multi-Task Settings"

Davide Buffelli
Department of Information Engineering
University of Padova
Padova, Italy
davide.buffelli@unipd.it

Fabio Vandin
Department of Information Engineering
University of Padova
Padova, Italy
fabio.vandin@unipd.it

A Comparison with Traditional Training Apporaches

Our proposed meta-learning approach is significantly different from the classical training strategy (Algorithm 1), and the traditional meta-learning approaches (Algorithm 2).

The classical training approach for multi-task models takes as input a *batch* of graphs, which is simply a set of graphs, where on each graph the model has to execute *all* the tasks. Based on the cumulative loss on all tasks

$$\mathcal{L} = \lambda^{(GC)}\mathcal{L}^{(\text{GC})} + \lambda^{(NC)}\mathcal{L}^{(\text{NC})} + \lambda^{(LP)}\mathcal{L}^{(\text{LP})}$$

for all the graphs in the batch, the parameters are updated with some form of gradient descent, and the procedure is repeated for each batch.

The traditional meta-learning approach takes as input an episode, like our approach, but for every graph in the episode *all* the tasks are performed. The support set and target set are *single* sets of graphs, where every task can be performed on all graphs. The support set is used to obtain the adapted parameters θ', which have the goal of *concurrently* solving all tasks on all graphs in the target set. The loss functions, both for the inner loop and for the outer loop, are the same as the one used by the classical training approach. The outer loop then updates the parameters aiming at a setting that can easily, i.e. with a few steps of gradient descent, be adapted to perform multiple tasks *concurrently* given a support set.

Algorithm 1: Classical Training

Input : Model f_θ; Batches $\mathcal{B} = \{\mathcal{B}_1, .., \mathcal{B}_n\}$
`init`(θ)
for $\mathcal{B}_i$ *in* $\mathcal{B}$ **do**
  `loss` $\leftarrow$ concurrently perform all tasks on all graphs in $\mathcal{B}_i$
  $\theta \leftarrow$ `UPDATE`$(\theta,$ `loss`$)$
end

Algorithm 2: Traditional Meta-Learning

Input : Model f_θ; Episodes $\mathcal{E} = \{\mathcal{E}_1, .., \mathcal{E}_n\}$
`init`(θ)
for $\mathcal{E}_i$ *in* $\mathcal{E}$ **do**
  `i_loss` $\leftarrow$ concurrently perform all tasks on all support set graphs
  $\theta' \leftarrow$ `ADAPT`$(\theta,$ `i_loss`$)$
  `o_loss` $\leftarrow$ concurrently perform all tasks on all target set graphs using parameters θ'
  $\theta \leftarrow$ `UPDATE`$(\theta, \theta',$ `o_loss`$)$
end

4th Workshop on Meta-Learning at NeurIPS 2020, Vancouver, Canada.

B Episode Design Algorithm

Algorithm 3 contains the procedure for the creation of the episodes for our meta-learning procedures. The algorithm takes as input a batch of graphs (with graph labels, node labels, and node features) and the loss function balancing weights, and outputs a *multi-task episode*. We assume that each graph has a set of attributes that can be accessed with a *dot-notation* (like in most object-oriented programming languages).

Notice how the episodes are created so that only one task is performed on each graph. This is important as in the inner loop of our meta-learning procedure, the learner adapts and tests the adaptated parameters on one task at a time. The outer loop then updates the parameters, optimizing for a representation that leads to fast *single-task adaptation*. This procedure bypasses the problem of learning parameters that *directly* solve multiple tasks, which can be very challenging.

Another important aspect to notice is that the support and target sets are designed as if they were the training and validation splits for training a single-task model with the classical procedure. This way the meta-objective becomes to train a model that can generalize well.

C Model Architecture

We use an encoder-decoder model with a multi-head architecture. The *backbone* (which represents the encoder) is composed of 3 GCN [8] layers with ReLU non-linearities and residual connections [5]. The decoder is composed of three *heads*. The node classification head is a single layer neural network with a *Softmax* activation that is shared across nodes and maps node embeddings to class predictions. In the graph classification head, first a single layer neural network (shared across nodes) performs a linear transformation (followed by a ReLU activation) of the node embeddings. The transformed node embeddings are then averaged and a final single layer neural network with *Softmax* activation outputs the class predictions. The link prediction head is composed of a single layer neural network with a ReLU non-linearity that transforms node embeddings, and another single layer neural network that takes as input the concatenation of two node embeddings and outputs the probability of a link between them.

D Additional Experimental Details

In this section we provide additional information on the implementation of the models used in our experimental section. We implement our models using PyTorch [11], PyTorch Geometric [4] and Torchmeta [3]. For all models the number and structure of the layers is as described in Appendix C, where we use 256-dimensional node embeddings at every layer.

To perform multiple tasks, we consider datasets with graph labels, node attributes, and node labels from the widely used TUDataset library [10]. At every cross-validation fold the datasets are split into 70% for training, 10% for validation, and 20% for testing. For each model we perform 100 iterations of hyperparameter optimization over the same search space (for shared parameters) using Ax [1].

We tried some sophisticated methods to balance the contribution of loss functions during multi-task training like GradNorm [2] and Uncertainty Weights [6], but we saw that usually they do not positively impact performance. Furthermore, in the few cases where they increase performance, they work for both classically trained models, and for models trained with our proposed procedures. We then set the balancing weights to $\lambda^{(GC)} = \lambda^{(NC)} = \lambda^{(LP)} = 1$ to provide better comparisons between the training strategies.

The multi-task performance Δ_m metric [9] is defined as the average per-task drop with respect to the single-task baseline: $\Delta_m = \frac{1}{T}\sum_{i=1}^{T}\left(M_{m,i} - M_{b,i}\right)/M_{b,i}$, where $M_{m,i}$ is the value for the multi-task model, and $M_{b,i}$ for the baseline.

Linear Model. The linear model trained on the embeddings produced by our proposed method is a standard linear SVM. In particular we use the implementation available in Scikit-learn [12] with default hyperparameters. For graph classification, we take the mean of the node embeddings as input. For link prediction we take the concatenation of the embeddings of two nodes. For node classification we keep the embeddings unaltered.

Algorithm 3: Episode Design Algorithm

Input : Batch of n randomly sampled graphs $\mathcal{B} = \{\mathcal{G}_1, .., \mathcal{G}_n\}$
Loss weights $\lambda^{(GC)}, \lambda^{(NC)}, \lambda^{(LP)} \in [0, 1]$

Output : Episode $\mathcal{E}_i = (\mathcal{L}_{\mathcal{E}_i}^{(m)}, \mathcal{S}_{\mathcal{E}_i}^{(m)}, \mathcal{T}_{\mathcal{E}_i}^{(m)})$

$\mathcal{B}^{(GC)}, \mathcal{B}^{(NC)}, \mathcal{B}^{(LP)} \leftarrow$ equally divide the graphs in $\mathcal{B}$ in three sets

`/* Graph Classification */`
$\mathcal{S}_{\mathcal{E}_i}^{(\text{GC})}, \mathcal{T}_{\mathcal{E}_i}^{(\text{GC})} \leftarrow$ randomly divide $\mathcal{B}^{(GC)}$ with a 60/40 split

`/* Node Classification */`
for $\mathcal{G}_i$ *in* $\mathcal{B}^{(NC)}$ **do**
  `num_labelled_nodes` $\leftarrow \mathcal{G}_i$`.num_nodes` $\times 0.3$
  $\mathcal{N} \leftarrow$ divide nodes per class, then iteratively randomly sample one node per class without replacement and add it to $\mathcal{N}$ until $|\mathcal{N}| =$ `num_labelled_nodes`
  $\mathcal{G}_i' \leftarrow$ `copy`$(\mathcal{G}_i)$
  $\mathcal{G}_i$`.labelled_nodes` $\leftarrow \mathcal{N}$; $\mathcal{G}_i'$`.labelled_nodes` $\leftarrow \mathcal{G}_i$`.nodes` $\setminus \mathcal{N}$
  $\mathcal{S}_{\mathcal{E}_i}^{(NC)}$`.add`$(\mathcal{G}_i)$; $\mathcal{T}_{\mathcal{E}_i}^{(NC)}$`.add`$(\mathcal{G}_i')$
end

`/* Link Prediction */`
for $\mathcal{G}_i$ *in* $\mathcal{B}^{(LP)}$ **do**
  $E_i^{(N)} \leftarrow$ randomly pick negative samples (edges that are not in the graph; possibly in the same number as the number of edges in the graph)
  $E_i^{1,(N)}, E_i^{2,(N)} \leftarrow$ divide $E_i^{(N)}$ with an $80/20$ split
  $E_i^{(P)} \leftarrow$ randomly remove 20% of the edges in $\mathcal{G}_i$
  $\mathcal{G}_i'^{(1)} \leftarrow \mathcal{G}_i$ removed of $E_i^{(P)}$
  $\mathcal{G}_i'^{(2)} \leftarrow$ `copy`$(\mathcal{G}_i'^{(1)})$
  $\mathcal{G}_i'^{(1)}$`.positive_edges` $\leftarrow \mathcal{G}_i'^{(1)}$`.edges`; $\mathcal{G}_i'^{(2)}$`.positive_edges` $\leftarrow E_i^{(P)}$
  $\mathcal{G}_i'^{(1)}$`.negative_edges` $\leftarrow E_i^{1,(N)}$; $\mathcal{G}_i'^{(2)}$`.negative_edges` $\leftarrow E_i^{2,(N)}$
  $\mathcal{S}_{\mathcal{E}_i}^{(LP)}$`.add`$(\mathcal{G}_i'^{(1)})$; $\mathcal{T}_{\mathcal{E}_i}^{(LP)}$`.add`$(\mathcal{G}_i'^{(2)})$
end

$\mathcal{S}_{\mathcal{E}_i}^{(m)} \leftarrow \{\mathcal{S}_{\mathcal{E}_i}^{(\text{GC})}, \mathcal{S}_{\mathcal{E}_i}^{(\text{NC})}, \mathcal{S}_{\mathcal{E}_i}^{(\text{LP})}\}$
$\mathcal{T}_{\mathcal{E}_i}^{(m)} \leftarrow \{\mathcal{T}_{\mathcal{E}_i}^{(\text{GC})}, \mathcal{T}_{\mathcal{E}_i}^{(\text{NC})}, \mathcal{T}_{\mathcal{E}_i}^{(\text{LP})}\}$
$\mathcal{L}_{\mathcal{T}_i}^{(\text{GC})} \leftarrow$ `Cross-Entropy`$(\cdot)$; $\mathcal{L}_{\mathcal{T}_i}^{(\text{NC})} \leftarrow$ `Cross-Entropy`$(\cdot)$
$\mathcal{L}_{\mathcal{T}_i}^{(\text{LP})} \leftarrow$ `Binary Cross-Entropy`$(\cdot)$
$\mathcal{L}_{\mathcal{E}_i}^{(m)} = \lambda^{(GC)}\mathcal{L}_{\mathcal{T}_i}^{(\text{GC})} + \lambda^{(NC)}\mathcal{L}_{\mathcal{T}_i}^{(\text{NC})} + \lambda^{(LP)}\mathcal{L}_{\mathcal{T}_i}^{(\text{LP})}$
Return $\mathcal{E} = (\mathcal{L}_{\mathcal{E}_i}^{(m)}, \mathcal{S}_{\mathcal{E}_i}^{(m)}, \mathcal{T}_{\mathcal{E}_i}^{(m)})$

Table 1: Results of a neural network trained on the embeddings generated by a multi-task model, to perform a task that was not seen during training by the multi-task model. "x,y ->z" indicates that the multi-task model was trained on tasks x and y, and the neural network is performing task z.

Task	**Model**	**Dataset**			
		ENZYMES	PROTEINS	DHFR	COX2
GC,NC ->LP	Cl	56.9 ± 3.9	54.4 ± 1.4	61.2 ± 2.2	59.8 ± 0.4
	iSAME	77.3 ± 4.5	88.5 ± 1.8	99.8 ± 1.8	97.1 ± 2.0
	eSAME	78.9 ± 2.8	89.1 ± 1.5	99.7 ± 2.2	95.8 ± 3.3
GC,LP ->NC	Cl	69.1 ± 1.2	57.3 ± 1.6	58.3 ± 9.3	68.9 ± 10.7
	iSAME	73.3 ± 2.1	59.2 ± 2.5	77.6 ± 1.6	78.1 ± 4.6
	eSAME	79.1 ± 1.7	64.7 ± 3.0	76.1 ± 2.7	76.9 ± 3.3
NC,LP ->GC	Cl	47.1 ± 2.4	75.3 ± 1.5	77.5 ± 3.1	79.9 ± 3.4
	iSAME	48.5 ± 5.5	76.1 ± 2.3	76.1 ± 3.7	79.7 ± 5.1
	eSAME	56.6 ± 3.1	74.6 ± 2.7	77.1 ± 3.6	79.3 ± 6.2

Deep Learning Baselines. We train the single task models for 1000 epochs, and the multi-task models for 5000 epochs, with early stopping on the validation set (for multi-task models we use the sum of the task validation losses or accuracies as metrics for early-stopping). Optimization is done using Adam [7]. For node classification and link prediction we found that normalizing the node embeddings to unit norm in between GCN layers helps performance.

Our Meta-Learning Procedure. We train the single task models for 5000 epochs, and the multi-task models for 15000 epochs, with early stopping on the validation set (for multi-task models we use the sum of the task validation losses or accuracies as metrics for early-stopping). Early stopping is very important in this case as it is the only way to check if the meta-learned model is overfitting the training data. The inner loop adaptation consists of 1 step of gradient descent. Optimization in the outer loop is done using Adam [7]. We found that normalizing the node embeddings to unit norm in between GCN layers helps performance.

E Full Results for Q3

Table 1 contains results for a neural network, trained on the embeddings generated by a multi-task model, to perform a task that was not seen during the training of the multi-task model. Accuracy (%) is used for node classification (NC) and graph classification (GC); ROC AUC (%) is used for link prediction (LP). The embeddings produced by our meta-learning methods lead to higher performance (up to **35%**), showing that our procedures lead to the extraction of more informative node embeddings with respect to the classical end-to-end training procedure.

References

[1] E. Bakshy, L. Dworkin, B. Karrer, K. Kashin, Benjamin Letham, Ashwin Murthy, and S. Singh. Ae : A domain-agnostic platform for adaptive experimentation. In *NeurIPS Systems for ML Workshop*, 2018.

[2] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *ICML*, 2018.

[3] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. URL `https://arxiv.org/abs/1909.06576`. Available at: https://github.com/tristandeleu/pytorch-meta.

[4] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[6] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Conference on Computer Vision and Pattern Recognition*, pages 7482–7491, 2018.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[8] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[9] K. Maninis, I. Radosavovic, and I. Kokkinos. Attentive single-tasking of multiple tasks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1851–1860, 2019.

[10] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020. URL `www.graphlearning.io`.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.