# Supplementary Document: Invertible Zero-Shot Recognition Flows

Anonymous ECCV submission

Paper ID 2605

Although the main paper is already self-contained, it is still worthwhile providing additional materials **for ease of reading**. Particularly, in this document, we provide additional clarifications, discussions and results in terms of the following themes:

- **Proof**: *change of variables formula* with **conditions** in probabilistic models
- **Additional background**: permutation and coupling [2]
- **Additional background**: calculating the Jacobian determinant of IZF
- **Additional details**: the settings and generative baselines involved in the toy experiment in Sec. 6.2 of the main paper.
- **More results**

In the rest of this document, we use **cyan fonts** to denote the content in the main paper for clarity.

## 1 Change of Variables in Conditional Probabilistic Models

In the main body of the paper, we derive the following conditional density estimation of an arbitrary visual sample in Eq. (3) as:

$$p_\theta(\mathbf{v}|y) = p_{\mathcal{Z}}\left(f\left(\mathbf{v}\right)|y\right)\left|\det\frac{\partial f}{\partial \mathbf{v}}\right|. \tag{1}$$

Although its **unconditional version** can be referred to [2,3] as a normalizing flow model, its is still important to show the proofs of it for ease of reading.

### 1.1 1-D Example: Conditional Density with Invertible Functions

Suppose we have 1-D observation $v$ of a datum with label $y$, and the hidden representation $z$ of $v$ is supported on a pre-defined prior $p_{\mathcal{Z}}$. Let an invertible function $f(\cdot)$ be the transformation between $v$ and $z$, *i.e.*,

$$\begin{aligned} z = f(v), v = f^{-1}(z), \\ \frac{dz}{dv} = f'(v). \end{aligned} \tag{2}$$

Then we can express the expectation of the value of another arbitrary function $h(z)$ with data from the class (condition) $y$:

$$\mathbb{E}_{p(z|y)}[h(z)] = \int h(z)p(z|y)dz, \tag{3}$$

$$\mathbb{E}_{p(v|y)}[h\left(f\left(v\right)\right)] = \int h\left(f(v)\right)p(v|y)dv. \tag{4}$$

Since $z = f(v)$ Equation (3) and (4) are actually equal, *i.e.*, $\mathbb{E}_{p(z|y)}[h(z)] = \mathbb{E}_{p(v|y)}[h\left(f\left(v\right)\right)]$. Considering the formal operation of Equation (2): $dz = f'(v)dv$, one can rewrite Equation (3) as

$$\begin{aligned}\mathbb{E}_{p(z|y)}[h(z)] &= \int h(z)p(z|y)f'(v)dv \\ &= \int h(f(v))p(z|y)f'(v)dv\end{aligned} \tag{5}$$

The RHS of Equation (4) and (5) are equal, *i.e.*, $\int h(f(v))p(z|y)f'(v)dv = \int h\left(f(v)\right)p(v|y)dv$, which gives the final version of the *change of variables formula*:

$$p(v|y) = p(z|y)f'(v) \tag{6}$$

### 1.2 High-Dimensional Factorized Extension

For high-dimensional data $\mathbf{v}$, the differentiation $f'(v)$ in Equation (6) is then extended to the absolute value of the Jacobian determinant $\left|\det\frac{\partial f}{\partial \mathbf{v}}\right|$, in other words:

$$p(\mathbf{v}|y) = p\left(f\left(\mathbf{v}\right)|y\right)\left|\det\frac{\partial f}{\partial \mathbf{v}}\right|. \tag{7}$$

One can further factorize [8] the latents $\mathbf{z} = [\mathbf{c}, \mathbf{z}^f] = f(\mathbf{v})$ and assume only $\mathbf{c}$ is dependent on $y$. Then this independence gives

$$\begin{aligned}p\left(\mathbf{z}|y\right) &= p(\mathbf{c}, \mathbf{z}^f|y) \\ &= p(\mathbf{c}|y)p(\mathbf{z}^f).\end{aligned} \tag{8}$$

Hence, we rewrite the conditional density of Equation (7) as follows:

$$p(\mathbf{v}|y) = p(\mathbf{c}|y)p(\mathbf{z}^f)\left|\det\frac{\partial f}{\partial \mathbf{v}}\right|, \tag{9}$$

which is exactly identical to Eq. (4) of the main body of our paper.

# 2 Additional Background: Permutation and Coupling [2]

## 2.1 Permutation Layer

Permutation layer shuffles the elements of a vector. The order of shuffling is randomly initialized before training, and is fixed and stored afterwards as part of network parameters. This structure is widely used in [1,3,5], and we refer to these articles for more details. Here we provide a typical example of how it is working.

Let $\mathbf{x}$ be a vector of 4 elements $[x_1, x_2, x_3, x_4]$. Permutation of the forward pass is randomly initialized with an order of $[3, 2, 4, 1]$. Then the layer behaves as follows

$$\begin{aligned} \text{Forward Pass: } \mathbf{z} &= [x_3, x_2, x_4, x_1], \\ \text{Reverse Pass: } \mathbf{x} &= [z_4, z_2, z_1, z_3]. \end{aligned} \tag{10}$$

**The randomly initialized permutation order is then frozen and stored as part of network parameters throughout training and test.** The corresponding Jacobian matrix is not dependent on the network inputs and trainable weights. Therefore, its log-determinant is actually a constant:

$$\log \left| \det \frac{\partial f_{\text{permute}}}{\partial \mathbf{x}} \right| = \textit{Const.} \tag{11}$$

## 2.2 Coupling Layer [2]

The coupling layer has been discussed in Sec. 3 and Eq. (2) of the main paper. We only elaborate its log-Jacobian determinant here.

As discussed in [2], the Jacobian matrix of a coupling layer is actually an upper triangular matrix:

$$\frac{\partial f_{\text{coupling}}}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \textit{something\_else} & \text{diag}(\exp(\mathbf{s})) \end{bmatrix}. \tag{12}$$

Coupling layer [2] comes with an easily-computed Jacobian determinant, and this is the main reason why it is widely used in generative flows as the main invertible structure.

# 3 Additional Background: Calculating the Jacobian Determinant of a Flow

We clarify that the following Jacobian determinant computation is not our contribution and can be sourced back to the related generative flow articles [2,3,5]. However, we provide the details of them to make the paper easier to follow.

As discussed in [2,3], for a cascaded invertible function $f = f_1 \circ f_2 \circ \cdots \circ f_k$, its absolute value of the log-Jacobian determinant *w.r.t.* an input $\mathbf{x}$ is computed by the summation of each individual:

$$\log \left| \det \frac{\partial f}{\partial \mathbf{x}} \right| = \sum_{i=1}^{k} \log \left| \det \frac{\partial f_i}{\partial f_{i-1}} \right|, \text{ with } \frac{\partial f_1}{\partial f_0} = \frac{\partial f_1}{\partial \mathbf{x}}. \tag{13}$$

As IZF involves 5 permutation-coupling blocks, its log-Jacobian determinant in Eq. (4) of the main paper is:

$$\log \left| \det \frac{\partial f_{\text{IZF}}}{\partial \mathbf{v}} \right| = \sum_{i=1}^{5} \sum_{j=1}^{d_v} \left| \mathbf{s}_i^{(j)} \right| + \mathit{Const}, \tag{14}$$

according to Equation (10) and (12). Here $\mathbf{s}_i$ refer to one of the two built-in networks of the five coupling layers employed by IZF.

# 4 Additional Details: Toy Experiment Settings and Generative Baselines

## 4.1 Toy Experiment Settings Demonstration

Though the setting of our toy experiment has been clearly discussed in the main paper (Sec. 6.2), we provide illustrative pseudo codes to further demonstrate how the experiments are conducted. These additional details are considered to be important as the toy experiments have no standard common practice. In particular, the example code for toy data generation is given in Listing 1.1.

## 4.2 Additional Baseline Details: Conditional GAN + $\mathcal{L}_{\text{iMMD}}$

The CGAN + $\mathcal{L}_{\text{iMMD}}$ baseline in Sec. 6.2 basically mimics the f-CLSWGAN [9] schema with an additional loss term of $\mathcal{L}_{\text{iMMD}}$ as part of IZF. The min-max game is defined as follows:

$$\min_G \max_D \mathbb{E}[D(\mathbf{v}^s, \mathbf{c}^s)] - \mathbb{E}[D(\hat{\mathbf{v}}^s, \mathbf{c}^s)] - \mathcal{L}_{\text{Lip}} + \alpha \mathcal{L}_{\text{CLS}} + \beta \mathcal{L}_{\text{iMMD}}, \tag{15}$$

where $\hat{\mathbf{v}}^s = G(\mathbf{z}, \mathbf{c}^s), \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. $\mathcal{L}_{\text{Lip}}$ is the Lipschitz continuity regularizer [4] as well employed in [9]. $\mathcal{L}_{\text{CLS}}$ is the classification loss on *seen* categories, being identical to the one in [9]. The hyper-parameter $\beta$ is set to $\beta = 0.1$ to match the case in IZF (we set $\lambda_3 = 0.1$ in Sec. 6.1 for $\mathcal{L}_{\text{iMMD}}$ in IZF).

We observed that $\mathcal{L}_{\text{iMMD}}$ makes the training of this baseline extremely unstable and leads to failure generation cases, which is discussed in Sec. 6.2. GAN + $\mathcal{L}_{\text{iMMD}}$ cannot produce reasonable classification results on real image features, which explains why it is not involved in the ablation study (Sec. 6.5).

Table 1: Test computation complexity of IZF.

| | IZF-NBC | IZF-Softmax |
|---|---|---|
| **CZSL** | $O(M^u d_v)$ | $O(M^u d_v)$ |
| **GZSL** | $O((M^s + M^u)d_v)$ | $O((M^s + M^u)d_v)$ |

### 4.3 Additional Baseline Details: CVAE + $\mathcal{L}_{\text{iMMD}}$

CVAE + $\mathcal{L}_{\text{iMMD}}$ baseline comes with a simple loss term combining the negative Evidence Lower BOund (ELBO) with $\mathcal{L}_{\text{iMMD}}$:

$$\mathcal{L} = D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{c},\mathbf{v})||p_\theta(\mathbf{z})) - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{c},\mathbf{v})}[\log p_\theta(\mathbf{v}|\mathbf{c},\mathbf{z})] + \beta\mathcal{L}_{\text{iMMD}}. \tag{16}$$

We also set $\beta = 0.1$ to have fair comparison with IZF.

## 5 More Results

### 5.1 Complexity Analysis

We provide the computational complexity of IZF during test in Table 1, being identical to the generative ZSL baselines [7,9,10].

### 5.2 Training Time

The training time of IZF is exponential to the size of the dataset, as we typically run ∼100 epochs on each dataset. For instance, it takes approximately 19 minutes to train IZF-NBC on AWA1 [6] with an Nvidia Titan Xp GPU, including the in-batch evaluation time. IZF-Softmax takes 11 additional minutes as we train a stand-alone classifier every 5 epochs.

## References


1. Ardizzone, L., Kruse, J., Wirkert, S., Rahner, D., Pellegrini, E.W., Klessen, R.S., Maier-Hein, L., Rother, C., Köthe, U.: Analyzing inverse problems with invertible neural networks. In: ICLR (2019) 3
2. Dinh, L., Krueger, D., Bengio, Y.: Nice: Non-linear independent components estimation. In: ICLR Workshops (2014) 1, 3, 4
3. Dinh, L., Sohl-Dickstein, J., Bengio, S.: Density estimation using real NVP. In: ICLR (2017) 1, 3, 4
4. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.C.: Improved training of wasserstein gans. In: NeurIPS (2017) 4
5. Kingma, D.P., Dhariwal, P.: Glow: Generative flow with invertible 1x1 convolutions. In: NeurIPS (2018) 3
6. Lampert, C.H., Nickisch, H., Harmeling, S.: Attribute-based classification for zero-shot visual object categorization. IEEE Transactions on Pattern Analysis and Machine Intelligence **36**(3), 453–465 (2013) 5

7. Li, J., Jing, M., Lu, K., Ding, Z., Zhu, L., Huang, Z.: Leveraging the invariant side of generative zero-shot learning. In: CVPR (2019) 5
8. Tsai, Y.H.H., Liang, P.P., Zadeh, A., Morency, L.P., Salakhutdinov, R.: Learning factorized multimodal representations. In: ICLR (2019) 2
9. Xian, Y., Lorenz, T., Schiele, B., Akata, Z.: Feature generating networks for zero-shot learning. In: CVPR (2018) 4, 5
10. Xian, Y., Sharma, S., Schiele, B., Akata, Z.: f-VAEGAN-D2: A feature generating framework for any-shot learning. In: CVPR (2019) 5

Listing 1.1: Code example of how toy experimental data are obtained.

```
import numpy as np
import torch as th
DEVICE = th.device("cuda" if th.cuda.is_available()  else
    "cpu")

class ToyReader(object):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.content = ['feat', 'label', 'label_emb', '
            s_cls', 'u_cls', 'cls_emb', 's_cls_emb', '
            u_cls_emb']
        self.parts = ['training', 'seen', 'unseen']
        assert self.batch_size % 12 == 0
        self.padding = kwargs.get('padding', True)
        self.seen_emb = np.asarray([[0, 1], [0, 0], [1,
            0]])
        self.unseen_emb = np.asarray([[1, 1]])
        self.seen_cls_num = self.seen_emb.shape[0]
        self.unseen_cls_num = self.unseen_emb.shape[0]

    def get_batch_tensor(self, part='training'):
        seen_centers = np.repeat(self.seen_emb, self.
            batch_size // self.seen_cls_num, axis=0)
        unseen_centers = np.repeat(self.unseen_emb, self.
            batch_size // self.unseen_cls_num, axis=0)

        seen_labels = np.repeat(np.asarray([0, 1, 2],
            dtype=np.float32), self.batch_size // self.
            seen_cls_num, axis=0)
        unseen_labels = np.repeat(np.asarray([3], dtype=np
            .float32), self.batch_size // self.
            unseen_cls_num, axis=0)

        gaussian_sample_1 = np.random.randn(self.
            batch_size, 2) / 3
        gaussian_sample_2 = np.random.randn(self.
            batch_size, 2) / 3
```

```
        seen_samples = seen_centers * 2 - 1 +
            gaussian_sample_1
        unseen_samples = unseen_centers * 2 - 1 +
            gaussian_sample_2

        if self.padding:
            padding = np.zeros([self.batch_size, 2], np.
                float32)
            seen_samples = np.concatenate([seen_samples,
                padding], axis=1)
            unseen_samples = np.concatenate([
                unseen_samples, padding], axis=1)

        if part == 'unseen':
            feed_list = [unseen_samples,
                         unseen_labels,
                         unseen_centers,
                         np.asarray([0, 1, 2]),
                         np.asarray([3]),
                         np.concatenate([self.seen_emb,
                             self.unseen_emb], axis=0),
                         self.seen_emb,
                         self.unseen_emb]
        else:
            feed_list = [seen_samples,
                         seen_labels,
                         seen_centers,
                         np.asarray([0, 1, 2]),
                         np.asarray([3]),
                         np.concatenate([self.seen_emb,
                             self.unseen_emb], axis=0),
                         self.seen_emb,
                         self.unseen_emb]

        feat = [th.tensor(i, dtype=th.float32).to(DEVICE)
            for i in feed_list]

        return feat
```