Supplementary Information: Data-Efficient Image Recognition with Contrastive Predictive Coding

A. Self-supervised pre-training

Model architecture: Having extracted 80×80 patches with a stride of 36×36 from an input image with 260×260 resolution, we end up with a grid of 6×6 image patches. We transform each one with a ResNet-161 encoder which terminates with a mean pooling operation, resulting in a [6,6,4096] tensor representation for each image. We then aggregate these `latents` into a 6×6 grid of context vectors, using a `pixelCNN`. We use this context to make the predictions and compute the `CPC` loss.

```
def pixelCNN(latents):
    # latents: [B, H, W, D]
    cres = latents
    cres_dim = cres.shape[-1]
    for _ in range(5):
      c = Conv2D(output_channels=256,
                 kernel_shape=(1, 1))(cres)
      c = ReLU(c)
      c = Conv2D(output_channels=256,
                 kernel_shape=(1, 3))(c)
      c = Pad(c, [[0, 0], [1, 0], [0, 0], [0, 0]])
      c = Conv2D(output_channels=256,
                 kernel_shape=(2, 1),
                 type='VALID')(c)
      c = ReLU(c)
      c = Conv2D(output_channels=cres_dim,
                 kernel_shape=(1, 1))(c)
      cres = cres + c
    cres = ReLU(cres)
    return cres

def CPC(latents, target_dim=64, emb_scale=0.1,
        steps_to_ignore=2, steps_to_predict=3):
    # latents: [B, H, W, D]
    loss = 0.0
    context = pixelCNN(latents)
    targets = Conv2D(output_channels=target_dim,
                     kernel_shape=(1, 1))(latents)
    batch_dim, col_dim, rows = targets.shape[:-1]
    targets = reshape(targets, [-1, target_dim])
    for i in range(steps_to_ignore, steps_to_predict):
      col_dim_i = col_dim - i - 1
      total_elements = batch_dim * col_dim_i * rows

      preds_i = Conv2D(output_channels=target_dim,
                       kernel_shape=(1, 1))(context)
      preds_i = preds_i[:, :-(i+1), :, :] * emb_scale
      preds_i = reshape(preds_i, [-1, target_dim])

      logits = matmul(preds_i, targets, transp_b=True)

      b = range(total_elements) / (col_dim_i * rows)
      col = range(total_elements) % (col_dim_i * rows)
      labels = b * col_dim * rows + (i+1) * rows + col

      loss += cross_entropy_with_logits(logits, labels)
    return loss
```

Image preprocessing: The final CPC v2 image processing pipeline we adopt consists of the following steps. We first resize the image to 300×300 pixels and randomly extract a 260×260 pixel crop, then divide this image into a 6×6 grid of 80×80 patches. Then, for every patch:

1. Randomly choose two transformations from Cubuk et al. (2018) and apply them using default parameters.

2. Using the primitives from De Fauw et al. (2018), randomly apply elastic deformation and shearing with a probability of 0.2. Randomly apply their color-histogram automentations with a probability of 0.2.

3. Randomly apply the color augmentations from Szegedy et al. (2014) with a probability of 0.8.

4. Randomly project the image to grey-scale with a probability of 0.25.

Optimization details: We train the network for the CPC objective using the Adam optimizer (Kingma & Ba, 2014) for 200 epochs, using a learning rate of 0.0004, $\beta_1 = 0.8$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and Polyak averaging with a decay of 0.9999. We also clip gradients to have a maximum norm of 0.01. We train the model with a batch size of 512, which we spread across 32 workers.

B. Linear classification

Model architecture: For linear classification we encode each image in the same way as during self-supervised pre-training (section A), yielding a 6×6 grid of 4096-dimensional features vectors. We then use Batch-Normalization (Ioffe & Szegedy, 2015) to normalize the features (omitting the scale parameter) followed by a 1×1 convolution mapping each feature in the grid to the 1000 logits for ImageNet classification. We then spatially mean-pool these logits to end up with the final log probabilities for the linear classification.

Image preprocessing: We use the same data pipeline as for self-supervised pre-training (section A).

Optimization details: We use the Adam optimizer with a learning rate of 0.0005. We train the model with a batch size of 512 images spread over 16 workers.

C. Efficient classification

C.1. Purely supervised

Model architecture: We investigate using ResNet-50, ResNet-101, ResNet-152, and ResNet-200 model architectures, all of them using the 'v2' variant (He et al., 2016), and find larger architectures to perform better, even when given smaller amounts of data. We insert a DropOut layer before the final linear classification layer (Srivastava et al., 2014).

Image preprocessing: We extract a randomly sized crop, as in the augmentations of Szegedy et al. (2014). We follow this with the same image transformations as for self-supervised pre-training (steps 1–4).

Optimization details: We use stochastic gradient descent, varying the learning rate in $\{0.05, 0.1, 0.2\}$, the weight decay logarithmically from 10^{-5} to 10^{-2}, the DropOut linearly from 0 to 1, and the batch size per worker in $\{16, 32\}$. We search for the best-performing model separately for each subset of labeled training data, as more labeled data requires less regularization. Having chosen these hyperparameters using a separate validation set (approximately 10k images which we remove from the training set), we evaluate each model on the test set (i.e. the publicly available ILSVRC-2012 validation set).

C.2. Semi-supervised with CPC

Model architecture: We apply the CPC encoder directly to the image, resulting in a 14×14 grid of feature vectors. These features are used as inputs to an 11-block ResNet classifier with 4096-dimensional hiddens layers and 1024-dimensional bottleneck layers. As for the supervised baseline, we insert DropOut after the final mean-pooling operation and before the final linear classifier.

Image preprocessing: We use the same pipeline as the supervised baseline.

Optimization details: We start by training the classifier while keeping the CPC features fixed. To do so we search through the same set of hyperparameters as the supervised baseline. After training the classifier till convergence, we fine-tune the entire stack for classification. In this phase we keep the optimization details of each component the same as previously: the classifier is fine-tuned with SGD, while the encoder is fine-tuned with Adam.

References

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

De Fauw, J., Ledsam, J. R., Romera-Paredes, B., Nikolov, S., Tomasev, N., Blackwell, S., Askham, H., Glorot, X., O'Donoghue, B., Visentin, D., et al. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine*, 24(9):1342, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL `http://arxiv.org/abs/1409.4842`.