With a Little Help from My Friends: Nearest-Neighbor Contrastive Learning of Visual Representations Supplementary Material

A. Pseudo-code

In Algorithm 1 we present the pseudo-code of NNCLR.

Algorithm 1 Pseudocode

```
# f: backbone encoder + projection MLP
# g: prediction MLP
# Q: queue

for x in loader: # load a minibatch x with n samples
   x1, x2 = aug(x), aug(x) # random augmentation
   z1, z2 = f(x1), f(x2) # projections, n-by-d
   p1, p2 = g(z1), g(z2) # predictions, n-by-d

   NN1 = NN(z1, Q) # top-1 NN lookup, n-by-d
   NN2 = NN(z2, Q) # top-1 NN lookup, n-by-d

   loss = L(NN1, p2)/2 + L(NN2, p1)/2

   loss.backward() # back-propagate
   update(f, g) # SGD update
   update_queue(Q, z1) # Update queue with latest
       projection embeddings

def L(nn, p, temperature=0.1):
   nn = normalize(nn, dim=1) # l2-normalize
   p = normalize(p, dim=1) # l2-normalize

   logits = nn @ p.T # Matrix multiplication, n-by-n
   logits /= temperature # Scale by temperature

   n = p.shape[0] # mini-batch size
   labels = range(n)

   loss = cross_entropy(logits, labels)

   return loss

def NN(z, Q):
  z = normalize(z, dim=1) # l2-normalize
  Q = normalize(Q, dim=1) # l2-normalize
  sims = z @ Q.T
  nn_idxes = sims.argmax(dim=1) # Top-1 NN indices
  return Q[nn_idxes]
```

It is possible to use momentum encoder with NNCLR training. The pseudo-code when momentum encoder is used is shown in Algorithm 2.

B. Evolution of Nearest-Neighbors

In Figure 1 we show how the nearest-neighbors (NN) vary as training proceeds. We observe consistently that in the beginning of training the NNs are usually chosen on the basis of color and texture. As the encoder becomes better at recognizing classes later in training, the NNs tend to belong to similar semantic classes.

Algorithm 2 Pseudocode with Momentum Encoder

```
# f: backbone encoder + projection MLP
# f_m: momentum version of (backbone encoder +
     projection MLP)
# g: prediction MLP
# Q: queue
# t: tau for momentum encoder

for x in loader: # load a minibatch x with n samples
   x1, x2 = aug(x), aug(x) # random augmentation
   z1, z2 = f(x1), f(x2) # projections, n-by-d
   p1, p2 = g(z1), g(z2) # predictions, n-by-d

   zm1, zm2 = f_m(x1), f_m(x2) # projections, n-by-d

   NN1 = NN(zm1, Q) # top-1 NN lookup, n-by-d
   NN2 = NN(zm2, Q) # top-1 NN lookup, n-by-d

   loss = L(NN1, p2)/2 + L(NN2, p1)/2

   loss.backward() # back-propagate
   update(f, g) # SGD update
   update_queue(Q, z_m1) # Update queue with latest
       projection embeddings from momentum encoder

   f_m = t*f_m + (1 - t) * f # Update momentum
       encoder weights
```

C. SimSiam with Nearest-neighbor as Positive

In this experiment we want to check if it is possible to use the nearest-neighbor in a non-contrastive loss. To do so we use the self-supervised framework SimSiam [1], in which the authors use a mean squared error on the embeddings of the 2 views, where one of the branches has a stop-gradient and the other one has a prediction MLP. We replace the stop-gradient branch with its nearest-neighbor from the support set. We call this method NNSiam. In Figure 2 we show how NNSiam differs from SimSiam. We also show how they both differ from SimCLR and NNCLR. Note that there is an implicit stop-gradient in NNSiam because of the use of hard nearest-neighbors. For this experiment, we use an embedding size of 2048, which is the same dimensionality used in SimSiam. We train with a batch size of 4096 with the LARS optimizer with a base learning rate of 0.2. We find that even with the non-contrastive loss using the nearest