CHAPTER 23

Training Transformers

23.1 TRAINING STRATEGIES

Training transformers from scratch is not for the faint-hearted. Most people download a set of pre-trained transformer weights, then fine-tune the transformer on their particular application, using a small to medium dataset for the fine-tuning. There is very good evidence that transformers benefit from being trained on extremely large scale datasets. It is now quite usual to pre-train the transformer using either noisy labels or a procedure that does not require any kind of image labelling at all.

There are three main pretraining strategies. **Pretraining with labels** is straightforward, and is discussed below. Obtaining labelled data at very large scales is extremely difficult. **Reconstructive pretraining** exploits the idea of Chapter **??**; if you can denoise a sufficiently mangled image using an encoder, you likely have a useful representation of the image. If you have some way of telling that pairs of data items should be "the same", then **contrastive pretraining** obtains embeddings that place those data items close but makes the whole collection of data spread out. This idea manifests in a range of forms, and has a section to itself (Section 23.3).

23.1.1 Evaluating Pre-training

You can evaluate an encoder and associated training procedure either by finetuning the pre-trained autoencoder on a standard dataset. The evaluation dataset is almost always ImageNet-1K.

An alternative evaluation procedure uses a *linear probe* on a standard evaluation dataset The encoder is trained using the procedure of interest. It is trained on some dataset that does not contain the evaluation dataset, then frozen. A linear classifier maps the encoder's codes to scores for each class in the standard dataset. The linear classifier is learned using training data from the standard dataset. You then evaluate the classification system on an appropriate subset of the evaluation dataset (test if you can, validation otherwise).

23.1.2 Pretraining with Labels

You can pretrain a transformer using a labelled dataset. Typically, you apply a linear classifier (Section 36.3) to the class token, compute cross-entropy to the true label, and descend on that loss. For some time, the standard pretraining dataset was ImageNet-1K (1.3e6 training images, 1e3 classes). ImageNet-21K or variants superseded this as a training dataset. A protocol for using ImageNet-21K for training appears at `https://github.com/Alibaba-MIIL/ImageNet21K`, together with code and other useful material. Internal to Google is a dataset called

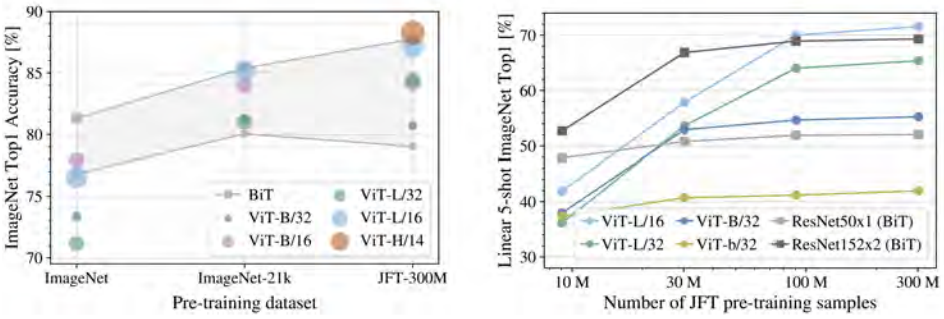

FIGURE 23.1: *Good behavior from transformers requires very large pretraining datasets, but performance tends not to saturate when the size of the dataset is increased. On the* **left***, a plot of the accuracy on ImageNet-1K of a number of classification networks. This accuracy is computed using fine-tuning of a linear classifier attached to the pre-trained encoder, using ImageNet-1K as an evaluation dataset. In each case, the encoder is pretrained with the labelled dataset indicated on the horizontal axis (where size increases as you move to the right). The vertical axis shows the performance of this classification system on ImageNet-1K test data. Each of the ViT markers is a vision transformer. Notice how performance increases quite sharply as the training dataset grows from 1.3e6 to 300e6 images. On the* **right***, the linear classifier used to decode is trained on 5 examples per class. ResNets outperform transformers in the smaller pretraining dataset regime, but saturate in the larger pretraining dataset regime. Transformer encoders do extremely well in the high pretraining data regime.* Image credit: *Figures 3 and 4 of "AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE" Dosovitsky et al., 2021.*

JFT, consisting of images labelled with various semi-automatic procedures for which I have not found descriptions. The original version (in [**?**]) is described as having 100e6 images, labelled with some 18e3 categories. Later versions are described as JFT-300M (with 300e6 images, in [**?**, **?**]) and JFT-3B (with 3e9 images, in []).

Evaluating transformer encoders by fine-tuning reveals that (a) more training data produces a better encoder (Figure 23.1); (b) transformer performance does not saturate with more training data, but ResNet performance does (Figure 23.1); (c) transformers use pre-training computation more efficiently than ResNets do. You could make reasonable criticisms of the underlying experiments **exercises**, but the statements appear robust as of writing.

23.2 RECONSTRUCTIVE PRE-TRAINING: THE MAE

Chapter 36.3 introduced learned image representations using the idea of autoencoding. If you use a sufficiently demanding noise model, you can pre-train a transformer encoder very effectively using autoencoding. You expect that an isolated token "knows" quite a lot about other tokens in an image. This "knowledge" can only be unlocked effectively if the encoder can extract quite abstract information

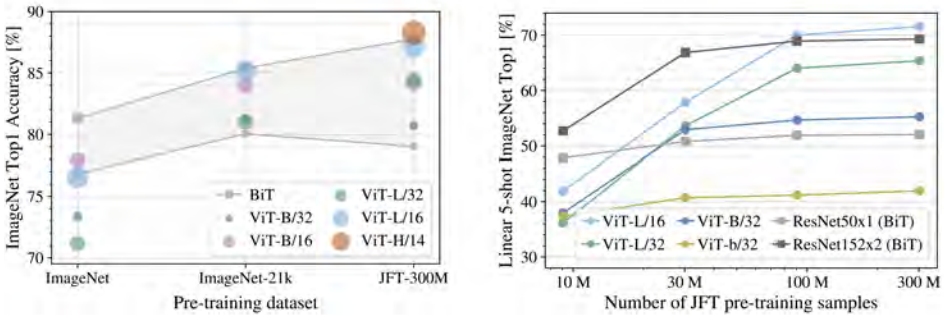

FIGURE 23.2: *There is good evidence that vision transformers use training computation efficiently. Each graph shows the accuracy of a system consisting of a transformer encoder and a linear classifier. The encoder is pretrained on a large dataset, then fine-tuned as in the text. The horizontal axis represents the amount of computation used in training.* **Left:** *shows accuracy averaged over five evaluation datasets,* **right** *shows ImageNet-1K accuracy. Notice that (a) increasing pre-training compute results in increased accuracy and (b) for a given pre-training compute, a transformer performs better.* Image credit: *Figure 5 of "AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE" Dosovitsky et al., 2021.*

from the token. As an example, a token shows the face of a tiger in a 3/4 view, with its nose pointing left; another token shows one pad; and a third shows its tail. There is likely more tiger in the image, and the three tokens contain a great deal of information about where the other bits of tiger are likely to be. But exploiting this information requires knowing a great deal about the content of the patches and a fair amount about tigers.

All this suggests a strategy. Knock out some fraction of the image. Do so by tokenizing the image, then removing a fraction of the tokens. Pass the remaining tokens through a transformer encoder. Now take the resulting tokens and insert mask tokens in the sequence locations where you knocked out input tokens. Pass the resulting set of tokens through a small decoder, and require that decoder to reconstruct the entire image. The decoder should be able to predict what's in the missing tokens *if* it knows where they are (which it does, from the way the mask tokens are inserted), and if the transformer encoder produces a sufficiently rich and deep representation of the remaining tokens. The result is an autoencoder – it should accept an image and produce an image – but trained to deal with extremely aggressive noise. This kind of autoencoder is known as a *masked autoencoder* or *MAE*.

For this strategy to work, you must knock out a large fraction of the image and get good reconstructions (Figure 23.3). Experimental evidence suggests that (a) if you do knock out a large fraction of tokens, a competitive encoder results; (b) the resulting encoder isn't as good as one pre-trained on JFT-300M, but beats encoders trained on other datasets; and (c) training an encoder with more parameters like this will result in a better encoder (Figure 23.4).

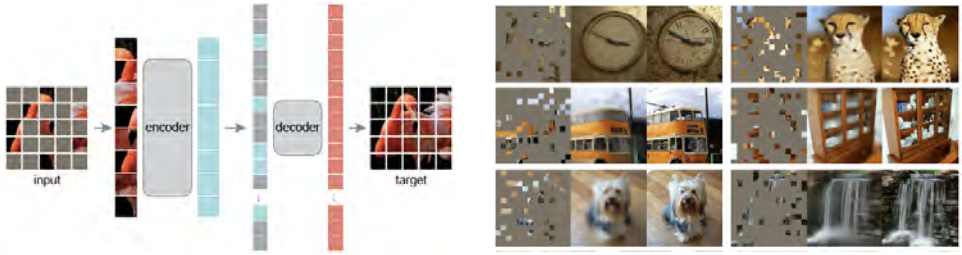

FIGURE 23.3: *You can train a transformer encoder by attaching a simple decoder, then compelling the encoder/decoder pair to reconstruct an image from a small subset of the patches in the original image.* **Left:** *shows the structure. The image is tokenized, passed to a transformer encoder, which produces a set of tokens. These are decoded and required to produce the missing tiles in the original image. Note just how aggressive the "noise" applied to the original image is – very few patches are supplied.* **Right:** *shows some examples of input; reconstruction; and ground truth.* Image credit: *Figure 1 and parts of Figure 2 of "Masked Autoencoders Are Scalable Vision Learners" He et al, 2021*

23.3 CONTRASTIVE PRETRAINING

Abstract an encoder as a device that accepts an image and produces a high dimensional vector. The encoder could be a convolutional encoder or a transformer, where you could use a variety of ways to get a vector out of the tokens. Contrastive training aims to produce a good encoder by reasoning about the properties of the distribution of the vectors.

Imagine that, for any image, you can easily obtain an image that is "the same". For example, choose some family of image augmentations where you believe that image $\mathcal{I}$ and augment($\mathcal{I}$) are "the same". For example, if you are going to use the encoding for image classification, you could use augmentations that crop the image by a reasonable amount then rescale it.

You could then train the encoder to have an **alignment** property, and require encode($\mathcal{I}$) and encode(augment($\mathcal{I}$)) to be close. On its own, this is not enough: the encoder could *collapse*, and produce the same encoding for every image. Doing so will result in a very good alignment loss.

You can prevent collapse by training the encoder to have a **uniformity** property. This requires the codes for a large dataset to spread out. The alignment property means that the encoding for two images that are "the same" is quite similar. The uniformity property means that, on the whole, the encoding for two randomly selected images is quite different. These two properties are much easier to manage if the vector produced by the encoder is a unit vector. If it is not, you can make vectors appear to be aligned by scaling everything with a small number, or appear to spread out by scaling them with a large number. All the cases I will discuss involve unit vectors.

As long as you mostly achieve alignment and uniformity, and the dimension is high enough, you can expect the unit vector produced by the encoder to describe

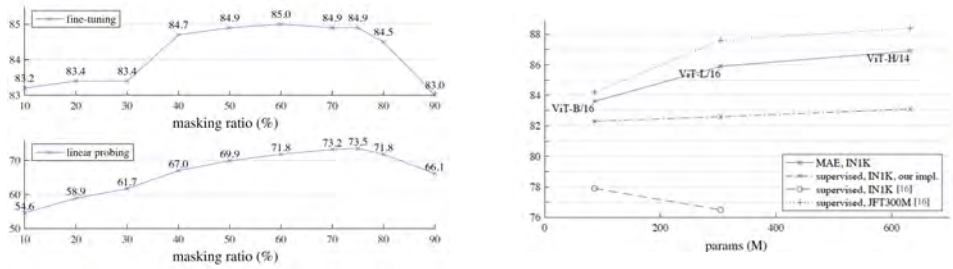

FIGURE 23.4: *Masked autoencoder (MAE) pre-training yields a strong transformer encoder. On the* **left**, *evaluation on ImageNet-1K using fine-tuning* **top** *and a linear probe* **bottom**. *You need to knock out a fairly large fraction of the tiles for the encoder to work well, but good performance is quite robust to the precise detail of how many tiles have been removed. On the* **right**, *MAE pre-training is clearly dominated by supervised pre-training on JFT-300M (which is not available to most researchers), but dominates other options. The accuracy depends on the number of parameters in roughly the same fashion for each pre-training strategy.* Image credit: *Figure 5 and Figure 8 of "Masked Autoencoders Are Scalable Vision Learners" He et al, 2021*

the image rather well. It is constructed so that two images that are "the same" produce similar vectors. This means the vector must represent properties that are preserved by your augmentation. But the vector should also be different for different images. This means that the vector should represent enough to, at least roughly, reconstruct the image. If it does not, you could force the vectors to spread out better by adjusting the encoder to represent more information about the image.

23.3.1 Contrastive Training with InfoNCE

It is not straightforward to test whether a set of examples is uniformly spread out. One procedure, *InfoNCE*, uses a loss that tests whether a batch of randomly selected examples is "far" from the pair that are like one another. Write $\mathbf{f}(\mathcal{I};\theta_I)$ for the code computed by the image encoder with parameters θ_I for image $\mathcal{I}$; write $\mathcal{A}(\mathcal{I};\xi)$ for the augmentation, which might have a random component. For example, you might use the random vector ξ to decide which crop to use.

Choose a parameter τ. Obtain a randomly selected batch of images $\mathcal{I}_1,\ldots,\mathcal{I}_N$. It is better to compare two augmentations of an image than an image to an augmentation **exercises**. You want $\mathbf{f}(\mathcal{A}(\mathcal{I};\xi_1);\theta_I)$ to be close to $\mathbf{f}(\mathcal{A}(\mathcal{I};\xi_1);\theta_I)$.

Now train the embedding using the loss

$$\mathcal{L}_I(\theta_I;\tau) = -\log\left[\frac{e^{\frac{\left(\mathbf{f}(\mathcal{A}(\mathcal{I}_1;\xi_1);\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_1;\xi_2);\theta_I)\right)}{\tau}}}{\sum_{i=1}^{N} e^{\frac{\left(\mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{f}(\mathcal{I}_i;\theta_I)\right)}{\tau}}}\right].$$

If this loss is small for almost every batch of randomly selected images, then for

any pair of images $\mathcal{I}_i$ and $\mathcal{I}_j$ that aren't the same,

$$e^{\frac{\left(\mathbf{f}(\mathcal{A}(\mathcal{I}_i,\xi_1);\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_i,\xi_2);\theta_I)\right)}{\tau}} \gg e^{\frac{\left(\mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{f}(\mathcal{I}_j;\theta_I)\right)}{\tau}}$$

which means that $\mathbf{f}(\mathcal{A}(\mathcal{I}_i;\xi_1);\theta_I)$ and $\mathbf{f}(\mathcal{A}(\mathcal{I}_i,\xi_2);\theta_I)$ are close, but $\mathbf{f}(\mathcal{I}_i;\theta_I)$ and $\mathbf{f}(\mathcal{I}_j;\theta_I)$ are far apart **exercises**. The parameter τ controls just how sharply the loss falls off as the average pair of images moves apart **exercises**.

You can rewrite the loss as

$$\mathcal{L}_I(\theta_I;\tau) = -\frac{\left(\mathbf{f}(\mathcal{A}(\mathcal{I}_1;\xi_1);\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_1,\xi_2);\theta_I)\right)}{\tau} + \log\left[\sum_{i=1}^{N} e^{\frac{\left(\mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_i,\xi);\theta_I)\right)}{\tau}}\right].$$

Here

$$-\frac{\left(\mathbf{f}(\mathcal{I}_1;\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_1,\xi);\theta_I)\right)}{\tau}$$

measures affinity and

$$\log\left[\sum_{i=1}^{N} e^{\frac{\left(\mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{f}(\mathcal{A}(\mathcal{I}_i,\xi);\theta_I)\right)}{\tau}}\right].$$

measures uniformity.

23.3.2 Contrastive Training with a Teacher: MoCo

InfoNCE presents a problem: you need a batch of images to compute the uniformity term, but you can compute the affinity term with just one image. Further, experiment shows that contrastive methods generally work better when you use a very large batch for the uniformity term, so you might need to recover a very large batch from disk to make a relatively small change to the encoder.

One way to manage this problem is to use a queue of examples to compute the uniformity term. Each time you recover a batch from disk, you first compute the loss using the batch and the queue. You then insert the batch into the queue, removing the oldest examples to keep the length the same. This means the queue can be very big – much bigger than a batch – so the uniformity term can be computed using a very large set of examples. This creates a difficulty computing the gradient for the uniformity term, because the gradient or various intermediate terms might not fit into GPU memory.

You can get around this difficulty in the following way. Assume that you *know* very good embeddings for all the examples in the queue. Write $\mathbf{q}_i$ for the embedding of the i'th example in the queue. For the j'th item in the batch, compute two different augmentations and encode each to get $\mathbf{f}_j(\theta)$ and $\mathbf{g}_j(\theta)$. These encodings would be good if (a) $\mathbf{f}_j$ and $\mathbf{g}_j$ were close for each j and (b) each $\mathbf{f}_j$ was very far from every $\mathbf{q}_i$. This means a θ that minimizes the loss

$$\mathcal{L}_I(\theta_I;\tau) = -\sum_{j\in\text{batch}} \frac{\left(\mathbf{f}_j(\theta_I)^T\mathbf{g}_j(\theta_I)\right)}{\tau} + \log\left[\sum_{i\in\text{queue}} e^{\frac{\left(\mathbf{f}_j(\theta_I)^T\mathbf{q}_i\right)}{\tau}}\right]$$

should cause the two different augmentations to be close to one another *and* far from the embeddings of the examples in the queue.

Of course, you don't know the embeddings of the examples in the queue. Further, you can't compute the gradient of the loss with respect to these embeddings because there are too many. You might consider just using the embedding you are using for the data points (i.e. θ_I), but experiment shows this is a bad idea. What appears to happen is oscillation caused by small overshoots in the steps. Likely, the examples in the batch move far away from the queue (and so closer to some other examples). The examples in the batch then end up in the queue. Now the examples in the new batch may have to move to get away from elements in the queue.

A good way to control this oscillation is to have an embedding that is: (a) quite close to recent embeddings of recent batches and (b) doesn't change as fast as the batch embeddings change. You can achieve this using a *moving average* of the batch embeddings. Write $\theta_{I,k}$ for the k'th estimate of the main embedding parameters and $\theta_{D,k}$ for the k'th estimate of the queue embedding parameters, and ϵ for a small positive number ($1e-3$, say). Then you can update by first updating θ_I using whatever descent procedure you want to use, then updating θ_D using

$$\theta_{D,k} = \epsilon\theta_{I,k} + (1-\epsilon)\theta_{D,k-1}.$$

This yields a $\theta_{D,k}$ that is (a) quite close to recent embeddings of recent batches and (b) doesn't change as fast as the batch embeddings change **exercises**. This procedure is usually called *momentum contrast* or, almost always, *MoCo*. You can think of the encoder for the queue as a teacher. MoCo was originally demonstrated on ResNets rather than transformers. The underlying idea is at the core of the training procedure for a family of very strong transformer-based image encoder, the DINO family.

23.3.3 Contrastive Training by Aligning to Text

A version of contrastive training that is very widely applied is CLIP. In this approach, one has a very large collection of pairs (image, associated text). The associated text may not directly describe everything in the image, and you do not know which term in the text describes what, if any, region in the image. However, it was found with the image and highly likely contains a fair description of its content.

Now build two encoders. One maps images to codes. The other maps text to codes. You can then train these encoders together by imposing two requirements. The two codes computed for a pair must be close together, which you can see as an alignment term. Further, if you compute a code for an image chosen at random and a code for associated text chosen at random, those two codes must be far apart. This loss criterion results in very strong image encoders that represent the semantic (object level) content of images.

Write $(\mathcal{I}_i, \mathcal{C}_i)$ for the i'th pair of image and associated text (Caption). Use the notation above, and write $\mathbf{g}(\mathcal{C}; \theta_C)$ for the code computed by the text encoder with parameters θ_C for caption $\mathcal{C}$. Each of the encoders produces a unit vector. You want to adjust θ_I and θ_C so that pairs are close and non-pairs are far.

Now assume you have a batch of N pairs. From this, you can compute a table

whose i, j'th entry is $\mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{g}(\mathcal{C}_j;\theta_C)$. Ideally, the diagonal of this table will be very close to one, and the off-diagonal elements will be small. To enforce this behavior, compute the cross-entropy loss between the i'th column of this table and a one-hot vector whose i'th component is 1, and sum over columns (see the box below).

Procedure: 23.1 *Computing the CLIP loss for a batch of N (image, text) pairs*

Write: $(\mathcal{I}_i, \mathcal{C}_i)$ for the i'th pair of image and text; $\mathbf{f}(\mathcal{I};\theta_I)$ for the code computed by the image encoder with parameters θ_I for image $\mathcal{I}$; and $\mathbf{g}(\mathcal{C};\theta_C)$ for the code computed by the text encoder with parameters θ_C for caption $\mathcal{C}$. Each of the encoders produces a unit vector.
Write: $\mathbf{e}_i$ for a vector whose i'th component is 1, and all others are zero; and

$$\mathbf{t}_i = \begin{bmatrix} \mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{g}(\mathcal{C}_1;\theta_C) \\ \dots \\ \mathbf{f}(\mathcal{I}_i;\theta_I)^T\mathbf{g}(\mathcal{C}_N;\theta_C) \end{bmatrix}$$

The loss is

$$\mathcal{L}_{\text{CLIP}}(\theta_I,\theta_C) = \sum_{i=1}^{N} \text{cross-entropy}(\mathbf{t}_i, \mathbf{e}_i)$$

Training an encoder using CLIP involves very large quantities of data. The dataset is not open, but the paper [] describes it as "400 million (image, text) pairs collected from a variety of publicly available sources on the Internet". The dataset was apparently collected by finding words that have at least 100 instances on Wikipedia, together with some bigrams, then querying the internet with that list. Queries were allowed to contribute no more than 20, 000 pairs to the dataset. This helps control class imbalance by reducing the number of pairs from common classes. The weights for a pre-trained CLIP encoder are available at `https://github.com/openai/CLIP`. An open implementation, with training code and pre-trained models, appears at `https://github.com/mlfoundations/open_clip`.

Now imagine you have a text encoder and an image encoder trained together with CLIP. You can use them to classify images without directly using any example images of the class (*zero-shot classification*). Obtain some image classification test set which has C classes, each named by a word. To classify an image, pass it into the image encoder. Pass each of the class names into the text encoder to get C vectors, one per class. Find the text vector that is closest to the image encoding, and label the image with the corresponding class name. This procedure works remarkably well (Figure 23.5).

23.3.4 The Underpinning of Contrastive Methods

You can see each of these instances of contrastive learning as a combination of an alignment loss and a uniformity loss. Write $\mathbf{f}(\mathcal{I};\theta)$ for the unit vector produced

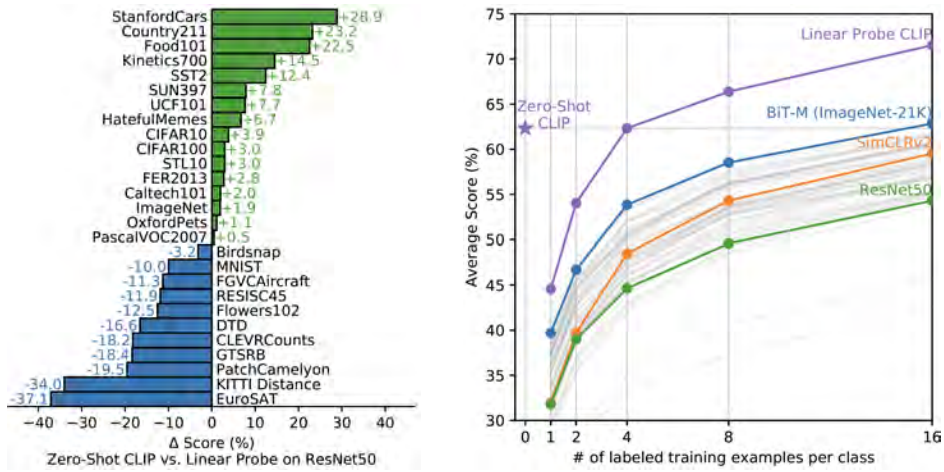

FIGURE 23.5: *Image and text encoders trained with CLIP exhibit impressive zero-shot classification. On the* **left**, *accuracy of a zero-shot classifier on the named dataset - accuracy of a fully supervised linear decoder on a frozen ResNet-50 encoder (which will have been trained on ImageNet-1K). If the bar is green, then CLIP zero-shot wins, meaning that CLIP zero-shot produces really impressive accuracy gains over a large number of datasets. On the* **right**, *zero-shot clip compared to linear probes trained on different size training sets (number of images per class on horizontal axis) for a variety of frozen encoders. Vertical is an average of accuracy on various datasets. Zero-shot CLIP is about as good as a linear probe on CLIP features using four examples per class.* Image credit: *Figure 5 and Figure 8 of "Learning Transferable Visual Models From Natural Language Supervision", Radford et al, 2021*

by the encoder with parameters θ applied to image $\mathcal{I}$ and $\mathcal{A}(\mathcal{I};\xi)$ for a random augmentation of $\mathcal{I}$ (where ξ is a random vector that chooses which augmentation to apply). Then choose some α and use

$$\mathcal{L}_{\text{align}}(\theta) = \|\mathbf{f}(\mathcal{I};\theta) - \mathbf{f}(\mathcal{A}(\mathcal{I};\xi);\theta)\|_2{}^{\alpha}$$

as an alignment loss **exercises**. Experiments suggest α should be 1 or 2, and favor 2. The uniformity loss is more interesting. Write p_{data} for the data distribution; $\mathbb{E}_{\mathbf{g}(\mathbf{x})}\left[\mathbf{x} \sim p_{\text{data}}\right]$ for the expectation of $\mathbf{g}(\mathbf{x})$ computed when $\mathbf{x}$ is distributed according to the data distribution. The expression below is minimized when q is the uniform distribution on the sphere

$$\mathbb{E}_{\mathbb{E}_{e^{-t[(\mathbf{x}-\mathbf{y})^T(\mathbf{x}-\mathbf{y})]}[\mathbf{y}\sim q]}}[\mathbf{x} \sim q]$$

as long as $t > 0$. Now choose θ to minimize

$$\log \mathbb{E}_{\mathbb{E}_{e^{-t[(\mathbf{f}(\mathbf{x};\theta)-\mathbf{f}(\mathbf{y};\theta))^T(\mathbf{f}(\mathbf{x};\theta)-\mathbf{f}(\mathbf{y};\theta))]}\left[\mathbf{y}\sim p_{\text{data}}\right]}\left[\mathbf{x} \sim p_{\text{data}}\right]$$

Assuming that the encoding is sufficiently flexible, you obtain a θ that actually minimizes the loss and so on, you will obtain an encoding that spreads the dataset uniformly across the sphere.

You cannot compute the loss exactly, but you can form an approximation by selecting a batch of N data items $\mathbf{x}_i$ uniformly and at random from the dataset. Using these, form

$$\mathcal{L}_{\text{uniform}}(\theta) = \log\left[\frac{1}{N^2}\left[\sum_{i,j\in\text{batch}} e^{-t\left[(\mathbf{f}(\mathbf{x};\theta)-\mathbf{f}(\mathbf{y};\theta))^T(\mathbf{f}(\mathbf{x};\theta)-\mathbf{f}(\mathbf{y};\theta))\right]}\right]\right].$$

Whether it is efficient to avoid evaluating pairs where $i = j$ depends a bit on your software tools **exercises**. It doesn't make any difference to the learning. Experience shows that good behavior requires N to be large.

23.4 LEARNING WITH A TEACHER: DINOSV1-3

Assume you have two encoders, a teacher and a student. Each produces a K dimensional vector. The teacher represents a good embedding of images, the student must learn that embedding. You can construct a loss that encourages the student to imitate the teacher by mapping the output of each to a probability distribution using $\text{softmax}(\mathbf{v})$, then comparing the distributions. Write $\mathbf{1}$ for a vector of 1's, $\exp(\mathbf{v})$ for the vector whose i'th component is e^{v_i}, and recall

$$\text{softmax}(\mathbf{v}) = \frac{\exp(\mathbf{v})}{\mathbf{1}^T\exp(\mathbf{v})}$$

(which represents a discrete distribution; every component is non-negative, and the components add to one). Recall also you can compare distributions with the cross-entropy (Section 20.2.3). Write $\mathbf{s}(\mathcal{I},\theta_t)$ for the teacher's embedding of image $\mathcal{I}$ and $\mathbf{s}(\mathcal{I},\theta_s)$ for the student's embedding, and $\mathbf{c}$ for a centering vector (below). Then you can use

$$H_x(\text{softmax}(\mathbf{t}(\mathcal{I},\theta_t)+\mathbf{c}), \text{softmax}(\mathbf{s}(\mathcal{I},\theta_s)))$$

to drive the student to imitate the teacher. Doing so is known as *distillation*.

Section **??** suggests that, at least in some circumstances, it is possible to learn even if the teacher doesn't have a good embedding. DINO establishes that this is the case, but does not use the teacher to build a queue of examples. There are (as of writing) three variants of DINO (DINO [], DINOv2 [] and DINOv3 []).

****** difficulties with collapse

I will start with the first, DINO. Start with an image, and take two crops of this image; each should cover half or more of the image, and it is fine if they overlap. In the original paper, the crops are 224×224; call the crops $\mathcal{I}_1$ and $\mathcal{I}_2$. The teacher will see these. Now form a set of views $\mathcal{V}$ of the image. Each is a somewhat smaller crop (in the paper, 96×96). The student will see these. Teacher and student encoders will have the same architecture, and should be able to accept images of the size you wish to work with (likely 224×224; resize the views). The encoders of interest here are ViT's (you can use ResNets), where the class token is passed to some fully connected layers to produce a K dimensional output. For

FIGURE 23.6: *The class token of a ViT trained using DINO is strongly affected by locations on the "subject" of the image. In each component,* **left** *is the input image and* **right** *is the self-attention weight between the class token and each token in the last transformer layer. Each head in the multi-headed self attention produces a weight. All tokens have a location in the original image, so you can plot the attention to the CLS token as a color image. Intensity gives the attention weight, and color the head.* Image credit: *Part of Figure 3 of "Emerging Properties in Self-Supervised Vision Transformers" Caron et al, 2021*

ViT's, K is either 384 or 768. For the moment, assume the teacher's weights are known. Then train the student using

$$\mathcal{L}_t(\theta_t) = \sum_{\mathcal{I}_g \in \{\mathcal{I}_1, \mathcal{I}_2\}} \sum_{\mathcal{I}_w \in \mathcal{V}} H_x(\text{softmax}(\mathbf{t}(\mathcal{I}_g, \theta_t) + \mathbf{c}), \text{softmax}(\mathbf{s}(\mathcal{I}_w, \theta_s)).$$

Notice how the teacher sees a large fraction of the image, and the student sees a small fraction of the image. This forces the embedding to attend to the relations between local and global representations.

The teacher is built using two procedures. First, a moving average gives the update

$$\theta_{t,k} = \epsilon\theta_{s,k} + (1-\epsilon)\theta_{t,k-1}.$$

Second, the teacher's output is centered by a form of moving average. Each time you pass a batch through the teacher, compute a new centering vector

$$\mathbf{c}_k = \delta \left[\sum_{i=1}^{B} \mathbf{t}(\mathcal{I}_i, \theta_t) \right] + (1-\delta)\mathbf{c}_{k-1}.$$

Experiment shows that doing so helps avoid collapse. Collapse occurs when a teacher and a student both settle on agreeing with one another but ignoring the input **exercises**. In an arrangement like DINO, batch normalization can create nuisance issues – what mean should one use in the teacher, for example – which do not arise here because the ViT's trained in this way do not use batch normalization.

Encoders produced with DINO produce strong results in the linear decoding or finetuning evaluations on ImageNet. One piece of evidence that these encoders are computing sensible encodings comes from the class tokens. You can compute the self-attention weight between the class token and each token in the last transformer layer. Each head in the multi-headed self attention produces a weight. All tokens have a location in the original image, so you can plot the attention to the CLS token as a color image. Intensity gives the attention weight, and color the head. The result (Figure 23.6) indicates that the class token is strongly affected by locations on the "subject" of the image. This is a good sign.

FIGURE 23.7: *DINOv2 features are learned without labels, dense, and able to predict depth and segmentation from single images.* **Top** *row shows pairs of (image, semantic segmentation).* **Bottom** *row shows pairs of (image, depth). In each case, the prediction is obtained with a simple linear decoder applied to frozen DINOv2 features.* Image credit: *Part of Figure 3 of "DINOv2: Learning Robust Visual Features without Supervision", Oquab et al, 2024*

DINOv2 is a significantly modified version of DINO that produces dense features. The loss is modified to include a reconstruction component that forces the student to be able to reconstruct tokens from other tokens. The student sees some view of the image, and the teacher sees another (which might be different). Some blocks in the image fed to the student are masked. These blocks correspond to tokens. The teacher sees these blocks, and the student must produce tokens that are the same as the corresponding token in the teacher's output. This is a version of the reconstructive loss of Section 23.2.

DINOv2 is trained with carefully structured data. The training data consists of two big blocks. One is curated data – preconstructed datasets like those of Section **??** – where you can expect that the images are complex, interesting, of use in training, and do not have objectionable material. Here objectionable material would include not-safe-for-work images and identifiable faces. Not-safe-for-work is a broad euphemism, usually abbreviated *NSFW*, covering such phenomena as nudity, sexual content, violence and blood – think about pictures you wouldn't want your boss seeing you looking at. The other big block is obtained from collected data. A raw collected dataset is obtained from a large web crawl. Automated methods are used to remove NSFW images and to remove duplicate images. Now a reliable ImageNet embedding is applied to all the remaining raw data. Each item in the curated data is then used to query the embeddings, by: embed the query item; then find four nearby items in the raw collected data. The resulting dataset consists collected items that are each somewhat like a query items, but not NSFW. This is the second block.

There is good evidence that the relatively dense features that DINOv2 produces give detailed local descriptions of images. Figure 23.7 shows some examples

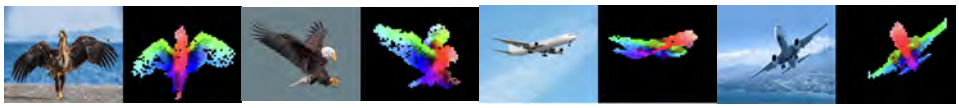

FIGURE 23.8: *DINOv2 features have some underlying knowledge of semantics. The features from all images are subject to principal components analysis; then, the coefficients of the top three principal components are used as R, G, and B. Each feature image is masked around the main object. Notice how comparable parts of each object (eg beak of the eagle–nose of the aircraft; or left wingtip of eagle–left wingtip of aircraft) have the about same color, implying they have about the same coefficients when projected on to the three main principal components.* **Top** *row shows pairs of (image, semantic segmentation).* **Bottom** *row shows pairs of (image, depth). In each case, the prediction is obtained with a simple linear decoder applied to frozen DINOv2 features.* Image credit: *Part of Figure 3 of "DINOv2: Learning Robust Visual Features without Supervision", Oquab et al, 2024*

of semantic segmentations and depth predictions for images, obtained using a linear decode of DINOv2 features. Quantitative experimentation supports the idea that these predictions are on average quite good. These local descriptions appear to have some information about object semantics. Figure 23.8 shows some examples of feature correspondences across object types.

The most current member of this family of encoders is DINOv3. This uses much of the ideas from earlier versions, but produces significant improvements in performance. Some improvements involve detailed adjustments of the architecture, learning process, and dataset which are out of scope (read [] for the details). An important improvement results from controlling a nasty effect (Figure 23.9), where the performance of discriminative features improves but the performance of dense spatial features collapses. The effect appears to involve all tokens drifting towards the class token. In doing so, they appear to lose information about local patches.

This effect is cured by a new loss, introduced relatively late in training. The *Gram matrix* of a set of features is the collection of inner products between all pairs. So if $\mathbf{x}_i$ is the patch feature describing the i'th location, the i, j'th component of the Gram matrix will be $\mathbf{x}_i^T\mathbf{x}_j$. Preserve a version of the network from relatively early in training, when the dense features are quite reliable. Much later in training, use this network as a teacher. Use a loss that drives the student Gram matrix to be close to the teacher's Gram matrix. Doing so allows the feature vectors at specific locations to change rather a lot, but preserves the inner products between locations. The teacher is then updated to take the student's parameters after 1e4 training iterations, and the procedure continues. The procedure is known as *Gram anchoring*, and has strong effects on the spatial structure of the feature map (Figure 23.10). As Figure 23.11 shows, the result is extremely strong spatial features.

23.5 DATASETS, SCALE AND CURATION

ImageNet-1K The main web page for ImageNet data is at `https://www.image-net.`

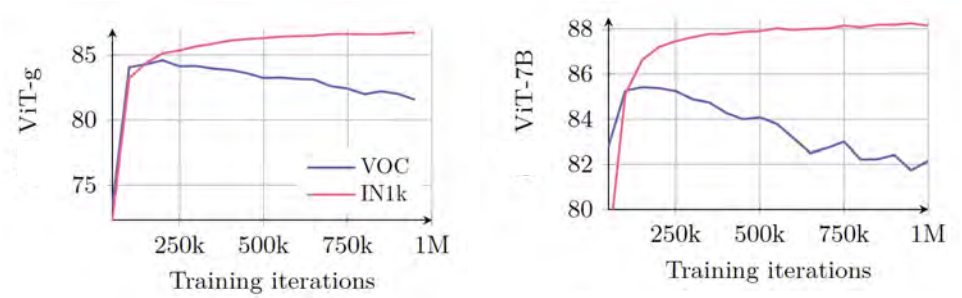

FIGURE 23.9: *Features used for image classification summarize an image, whereas dense spatial features describe individual locations. This means that training that improves classification behavior might not help dense spatial features. These two graphs show the same effect occurring for two different transformers trained with DINOv2 losses. The* **red** *curves show accuracy on ImageNet1K using these features; the* **blue** *curves show accuracy on a VOC segmentation dataset. Notice the discriminative accuracy increases slowly, while the segmentation accuracy falls quite fast.* Image credit: *Part of Figure 5 of "DINOv3", Simeoni et al, 2025*

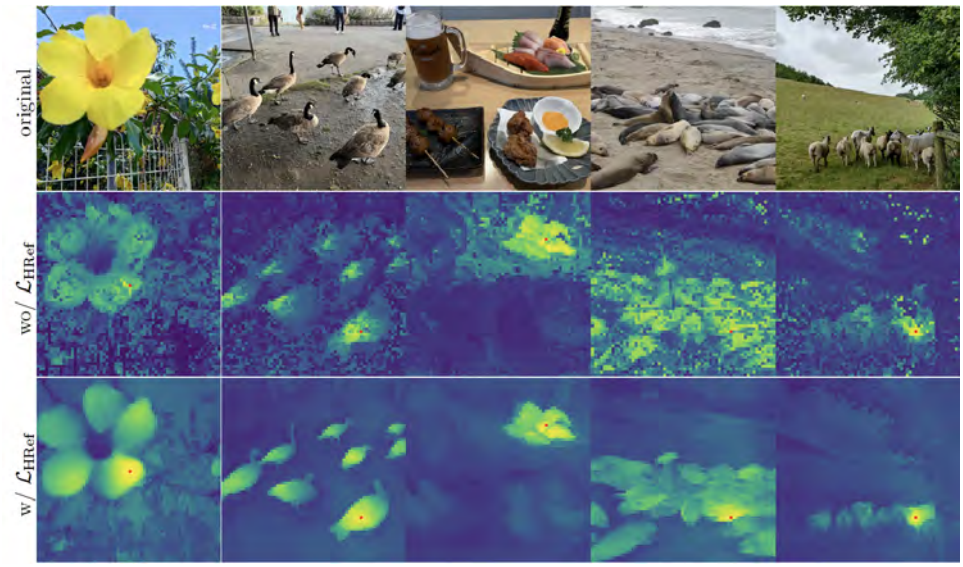

FIGURE 23.10: *Gram anchoring causes significant changes in the behavior of features.* **Top** *row shows images;* **center** *row shows cosine distances between dense features at each location and those at the location in* **red**, *for an encoder trained* without *Gram anchoring (***yellow** *is similar,* **blue** *is different). Notice how disorganized the map appears to be.* **Bottom** *row shows the same cosine distances, but now the encoder was trained* with *Gram anchoring. The map is much less noisy, and the similarities are reasonable.* Image credit: *Figure 10 of "DINOv3", Simeoni et al, 2025*

FIGURE 23.11: *Gram anchoring causes significant changes in the behavior of features.* **Top** *row shows images;* **center** *row shows cosine distances between dense features at each location and those at the location in* **red**, *for an encoder trained* without *Gram anchoring (***yellow** *is similar,* **blue** *is different). Notice how disorganized the map appears to be.* **Bottom** *row shows the same cosine distances, but now the encoder was trained* with *Gram anchoring. The map is much less noisy, and the similarities are reasonable.* Image credit: *Figure 10 of "DINOv3", Simeoni et al, 2025*

`org` (be careful, the hyphen matters!). ImageNet collected some 14e6 images with some 22e3 classes. This collection was cut down to a standard set of 1.3e6 training images with 1000 classes. In the literature, "ImageNet-1K" always refers to this standard set, but "ImageNet" could refer to either the original set or the standard set. An alternative site is `https://huggingface.co/datasets/ILSVRC/imagenet-1k`).

p**ImageNet-21K** refers to the large version of ImageNet. There is considerable helpful information about preprocessing images for training, as well as a dataset, at `https://github.com/Alibaba-MIIL/ImageNet21K`. Part of preprocessing involves removing classes with very few instances. Be aware that "ImageNet-21K" might refer to either the preprocessed version (12e6 images, 11e3 classes) or the raw version (14e6 images, 22e3 classes).

Fine-tuning code and pre-trained models are available at `https://github.com/google-research/vision-transformer`

https://laion.ai/blog/laion-5b/ https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset https://cocodataset.org/#home https://ai.google.com/research/ConceptualCaptions/ https://davar-lab.github.io/dataset/lsvtd.html https://github.com/m-bain/webvid https://visualqa.org https://lil.nlp.cornell.edu/nlvr/ https://okvqa.allenai.org NSFW

https://huggingface.co/datasets/limjiayi/hateful_memes_expanded https://github.com/PRITHIVSAKTHIUR/nsfw-image-detection?tab=readme-ov-file https://github.com/EBazarov/nsfw_data_source_urls https://docs.nvidia.com/nemo/curator/0.25.7/curate-images/process-data/classifiers/nsfw.html https://github.com/shahidmuneer/multimodal-nsfw-defense