Deep Learning

A collection of definitions, theorems, and formulae

January 23, 2026

Contents

1 General notes and intuition

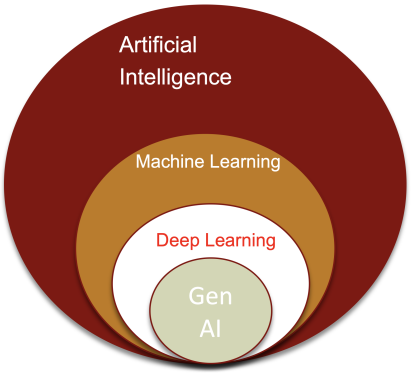
1.1 Polanyi's paradox

Humans are capable of doing many complex tasks easily, but it is difficult to describe these tasks in their entirety. This makes teaching computers to do something exclusively using if-then statements very hard. To address this problem, AI makes decisions based on examples instead of hard-coded rules.

1.2 Statistics $\subseteq$ machine learning $\subseteq$ deep learning $\subseteq$ generative ai

Machine learning is a subset of statistics and often works on prediction tasks using *structured* input data, such as tabular data. On the other hand, deep learning applies to unstructured data and can more tasks beyond prediction. For example, images are unstructured because each pixel is on the scale and has no sematic meaning unlike tabular data.

In the past, machine learning on unstructured data (e.g. images) would first require manual/programmatic feature extraction. The aha moment was that deep learning enabled an automatic pipeline to extract features that matter for downstream data. AlexNet for image classification is a canonical example here. Generative models took deep learning a step further by enabling unstructured outputs.



1.3 How to encode categorical input variables

For the input layer of a feed forward neural network, it is often helpful to encode categorical input variables as one-hot vectors

1.4 Binary cross entropy loss function

we use a cross entropy loss function when using a feed forward neural network for prediction via estimated probabilities. For a datapoint $y_i \in \{0, 1\}$ and model's probability prediction $\hat{m}(x) \in (0, 1)$, the binary cross entropy loss is

$$\text{BCE}(y_i, \hat{m}(x)) \;=\; -[y_i \log \hat{m}(x) + (1 - y_i) \log(1 - \hat{m}(x))].$$

Also recall that we'll almost always use a sigmoid activation function for the last layer of a neural net intented for binary prediction. Over a dataset, we do the usual thing by averaging over n samples

$$\mathcal{L}(y \mid \hat{m}) = \frac{1}{n} \sum_{i=1}^{n} \text{BCE}(y_i, \hat{m}(x_i))$$

1.5 Categorical cross-entropy loss function

The binary cross entropy loss has a nice generalization to an arbitrary number of classes. We'll first work in the case of an arbitrary target distribution $p(k|x)$ on classes $k \in \mathcal{Y}$ and model $\hat{p}(k|x)$. In this setting, the cross entropy loss is

$$\begin{aligned} \text{CE}(p, \hat{p}) \;&=\; -\sum_{k \in \mathcal{Y}} p(k|x) \, \log \hat{p}(k|x) \\ &=\; H(p) + \text{KL}(p \parallel \hat{p}). \end{aligned}$$

Therefore minimizing $\text{CE}(p, \hat{p}) \iff$ minimizing $\text{KL}(p \parallel \hat{p})$ since $H(p)$ is constant.

When working in a neural netowrk, what is the sample loss we can actually compute? In this case, we no longer have a target distribution $p(k|x)$ on classes $k \in \mathcal{Y}$, but instead have target one-hot label $y \in \{e_1, \ldots, e_K\}$. Our model (usually a neural network) will output a probability distribution over classes using the softmax activation function applied to the logits $z \in \mathbb{R}^K$ in the last layer of the network: $\hat{p}_k = \text{softmax}(z)_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$. In this setting, cross entropy loss becomes

$$\text{CE}(y, \hat{p}) \;=\; -\sum_{k} y_k \log \hat{p}_k$$

Note: this is simply the CE loss for the class prediction of a *single* data sample x with label $y \in \{e_1, \ldots, e_K\}$.

1.6 Why CE from maximum likelihood (MLE).

It's worth asking why teh cross entropy loss is the "correct" function to minimize if we want our model to learn the target distribution/class.Given i.i.d. data $\{(x_i, y_i)\}_{i=1}^n$ and a conditional model $\hat{p}_\theta(y|x)$, the (conditional) log-likelihood is

$$\ell(\theta) = \sum_{i=1}^{n} \log \hat{p}_\theta(y_i|x_i).$$

Maximizing likelihood $\max_\theta \ell(\theta)$ is equivalent to minimizing the negative log-likelihood (NLL):

$$\min_\theta \; -\frac{1}{n}\sum_{i=1}^{n} \log \hat{p}_\theta(y_i|x_i) \;=\; \min_\theta \; \frac{1}{n}\sum_{i=1}^{n} \mathrm{CE}(\delta_{y_i}, \hat{p}_\theta(\cdot|x_i)),$$

which is precisely (binary or multiclass) cross-entropy with empirical targets δ_{y_i}.

1.7 ADAM

The idea behind ADAM is to average the last K gradients. So instead of regular gradient descent

$$x_{t+1} = x_t - \alpha \nabla g(x_t)$$

we use the gradient averaging method

$$x_{t+1} = x_t - \alpha \Big[\frac{1}{K} \sum_{k=t-(K-1)}^{t} \nabla g(x_t) \Big].$$

What is is an example of a function that this work on? Consider $g(x) = 2x_{(1)}^2 + 100x_{(2)}^2$
Backprop is essentially about large matrix-vector multiplications

1.8 Batches

Mini-batch training works by splitting the dataset into smaller groups of examples called batches (or mini-batches), such as batches of size 32. After processing one batch, the model performs an update, meaning it computes the gradient of the loss using only that batch and then adjusts the weights using that gradient (one gradient-descent step). In other words the model is updated via descent once per mini-batch. Using a smaller batch size reduces computation per gradient descent update because the gradient is computed from fewer examples, but the resulting gradient estimate is typically noisier because it is based on a smaller, more variable sample of the data, so it can deviate more from the true gradient you would get if you used the entire dataset.

An epoch is one complete pass through the entire dataset. In full-batch gradient descent, a single gradient step uses all examples, so one update requires one full pass through the dataset (i.e., one epoch per update). In mini-batch gradient descent, you make many updates within an epoch: if you have N data examples and batch size B, then one epoch consists of roughly N/B batches, and you perform one gradient descent update after each batch. Typically, within each epoch you iterate through the data without replacement (often after shuffling), so each example appears in exactly one batch for that epoch.

Early stopping is a strategy to reduce overfitting by deciding when to stop training before the model begins to fit noise in the training data. It usually relies on a separate validation set: during training, you periodically evaluate validation performance, and if it stops improving (often after a patience window), you stop and keep the model from the best validation point. This ties back to epochs and updates because validation checks are often done at the end of each epoch (or every fixed number of updates), using the validation trend as the signal for when additional training is no longer helping generalization.

1.9 FID score for generated images

Let $\{x_i\}_{i=1}^n$ be real images and $\{\tilde{x}_i\}_{i=1}^m$ be generated images. Pass images through a fixed feature extractor (typically Inception-v3) and let

$$f_i := \phi(x_i) \in \mathbb{R}^d, \qquad \tilde{f}_i := \phi(\tilde{x}_i) \in \mathbb{R}^d.$$

This feature extractor ϕ is a fixed, pretrained neural network (typically Inception-v3 trained on ImageNet) that maps images into a high-level semantic feature space. It is used to compare real and generated images via their statistics in this representation, under the assumption that distances in feature space better reflect perceptual and semantic similarity than raw pixel space. Compute empirical means and covariances

$$\hat{\mu}_r = \frac{1}{n}\sum_{i=1}^{n} f_i, \quad \hat{\Sigma}_r = \frac{1}{n-1}\sum_{i=1}^{n}(f_i - \hat{\mu}_r)(f_i - \hat{\mu}_r)^\top,$$

$$\hat{\mu}_g = \frac{1}{m}\sum_{i=1}^{m} \tilde{f}_i, \quad \hat{\Sigma}_g = \frac{1}{m-1}\sum_{i=1}^{m}(\tilde{f}_i - \hat{\mu}_g)(\tilde{f}_i - \hat{\mu}_g)^\top.$$

The FID is the squared 2-Wasserstein distance between the Gaussian approximations $\mathcal{N}(\hat{\mu}_r, \hat{\Sigma}_r)$ and $\mathcal{N}(\hat{\mu}_g, \hat{\Sigma}_g)$:

$$\text{FID} \;:=\; \|\hat{\mu}_r - \hat{\mu}_g\|_2^2 \;+\; \text{Tr}\Big(\hat{\Sigma}_r + \hat{\Sigma}_g - 2\,(\hat{\Sigma}_r^{1/2}\hat{\Sigma}_g\hat{\Sigma}_r^{1/2})^{1/2}\Big).$$

Important facts / intuition:

$$\text{FID} \geq 0, \qquad \text{FID} = 0 \iff \hat{\mu}_r = \hat{\mu}_g \text{ and } \hat{\Sigma}_r = \hat{\Sigma}_g \;\; \text{(under the Gaussian model)}.$$

- The mean term $\|\hat{\mu}_r - \hat{\mu}_g\|_2^2$ captures feature bias; covariance term captures diversity/mode coverage.
- Lower is better, but it is feature-space dependent (choice of ϕ) and can be biased at finite sample sizes.
- FID is not a true distance between image distributions in general; it is a distance between their Gaussian fits in feature space.

2 Neural network backpropagation

2.1 Supervised learning optimization

Supervised deep learning is the setting where we have a ground truth to build a loss function off. The fundamental picture involves running a input through a function, then evaluating the loss. We want the parameter θ^* that minimizes the cumulative loss across all N training samples:

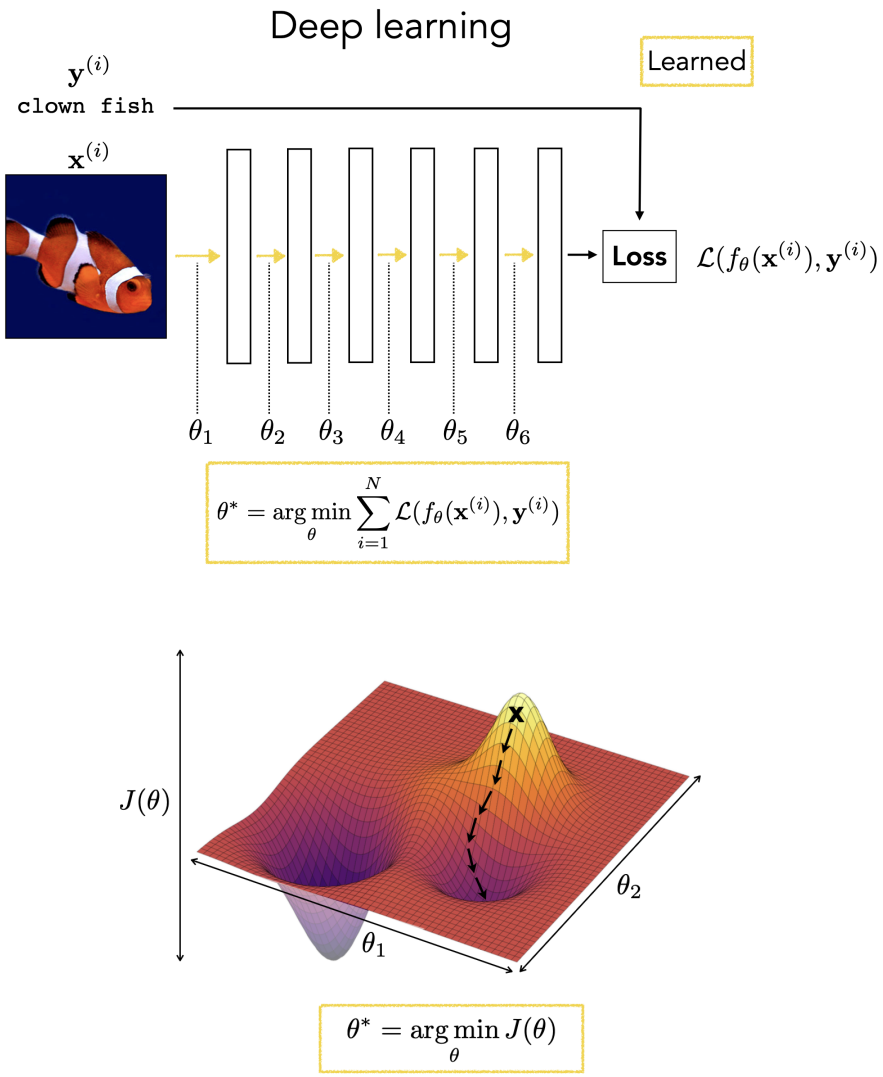


Figure 1: For a fixed dataset $\mathcal{D} = \{(y^{(i)}, \mathbf{x}^{(i)}\}_{i\in[N]}$, evaluating $J(\theta) = \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}, y^{(i)})$ for all θ creates a surface over which we optimize.

2.2 Gradient descent

One iteration of gradient descent is

$$\theta^{k+1} \leftarrow \theta^k - \eta \nabla_\theta J(\theta^k)$$

where

$$J(\theta) = \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}, y^{(i)})$$

Stochastic Gradient Descent (SGD) Computes the gradient on a sub-batch of the dataset. Let $\mathcal{B} \subseteq [N]$, then

$$J_{\mathcal{B}}(\theta) = \sum_{i \in \mathcal{B}} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}, y^{(i)})$$

- If `batchsize` $= 1$ then θ is updated after each example.
- If `batchsize` $= N$ (full dataset) then this is standard gradient descent on $J(\theta)$.
- Advantages: Faster that standard GD since approximates total gradient with small sample. Also an implicit regularizer
- Disadvantages: high variance, unstable updates

2.3 Backpropogation

Backpropagation is an algorithm for propagating shared terms throughout the computation graph in order to update layer weights. To update the parameters $\theta_{(l)}$ in layer (l), we use gradient descent

$$\theta_{(l)} \leftarrow \theta_{(l)} - \eta \left(\frac{\partial J}{\partial \theta_{(l)}} \right)^\top.$$

How do we calculate this? First let us use the convention that $\mathbf{x}_{(l)} = f_{(l)}(\mathbf{x}_{(l-1)}, \theta_l)$ which we will use interchangeably. Now use the chain rule to write

$$\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \theta_1}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \theta_2}.$$

Notice that, in general, all we need to do is keep track of these

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial \mathbf{x}_l} \frac{\partial \mathbf{x}_l}{\partial \theta_l},$$

which is the primary tool in backpropagation. The overall algorithm consists of two steps.

Backward pass: (generating the seed for to be passed to the previous layer)

$$\frac{\partial J}{\partial \mathbf{x}_{l-1}} = \frac{\partial J}{\partial \mathbf{x}_l} \frac{\partial f_l}{\partial \mathbf{x}_{l-1}}$$

Gradient computation: (using the seed given by the previous layer)

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial \mathbf{x}_l} \frac{\partial f_l}{\partial \theta_l}$$

Parameter update

$$\theta_{(l)}^{k+1} \leftarrow \theta_{(l)}^{k} - \eta \left(\frac{\partial J}{\partial \theta_{(l)}} \right)^\top.$$

The figure does a nice job of describing this for a general layer:

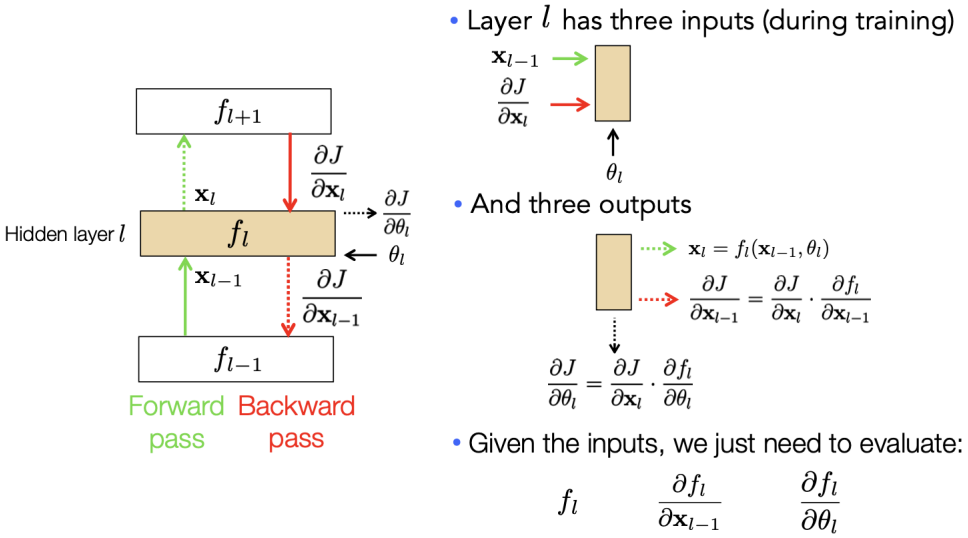


Figure 2: At each layer l we have 3 inputs and 3 outputs

Directed acyclic graphs Backpropagation also works in Directed acyclic graphs (DAGs)

3 Why do DNNs generalize?

(Also see Overfitting and Generalization within Toolkit/Analytics.)

We want to minimize population risk

$$\mathcal{R}(\theta) = \mathbb{E}_{\mathbf{x},\mathbf{y}}[\mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y})]$$

by minimizing the sample loss

$$\hat{\mathcal{R}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i)$$

Idea 1: Architectural symmetries Imposing invariances via special architectures restrict the hypothesis space so we are more likely to learn the "true" function.
Idea 2: Simplicity bias in the parameter-function map Most random settings of the weights and biases in a neural net map to simple functions, therefore our random initialization already place us in a region of the hypothysis space that corresponds to simple functions.
Idea 3: SGD likes to learn simple functions if possible SGD converges to "flat" minima; it will tend to overshoot or bounce out of minima that are too narrow. Furthermore, weight decay acts like an L2 regularizer on weights, shrinking them toward zero

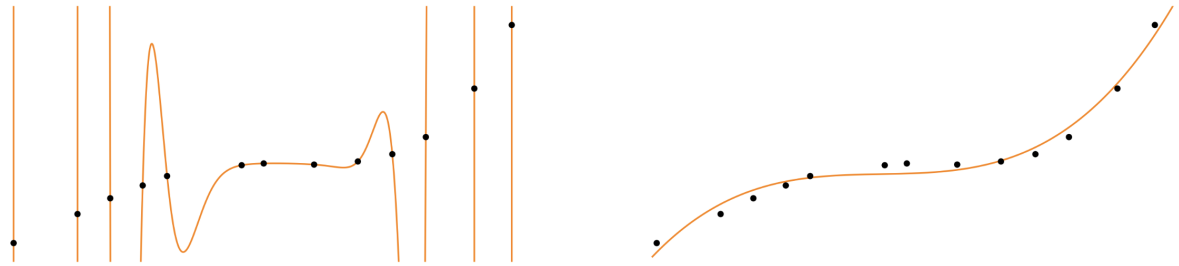


Figure 3: The simplest model that fits the data will probably generalize the best

3.1 Generalization error

Generalization error = population error − training error.
IF size of training set >> number of functions in our function class THEN training error matches population error with high probability.

This means that if all data is i.i.d. and...

- there's a function in your hypothesis class that works well (maybe not; that's the "cost" of a small class!)
- and you manage to find one such function
- and your hypothesis class is small enough.

Then you can be pretty sure that you'll have a good function on unseen examples.

4 Convolutional Neural Networks

4.1 Why Convolutions? Inductive Bias & Data Efficiency

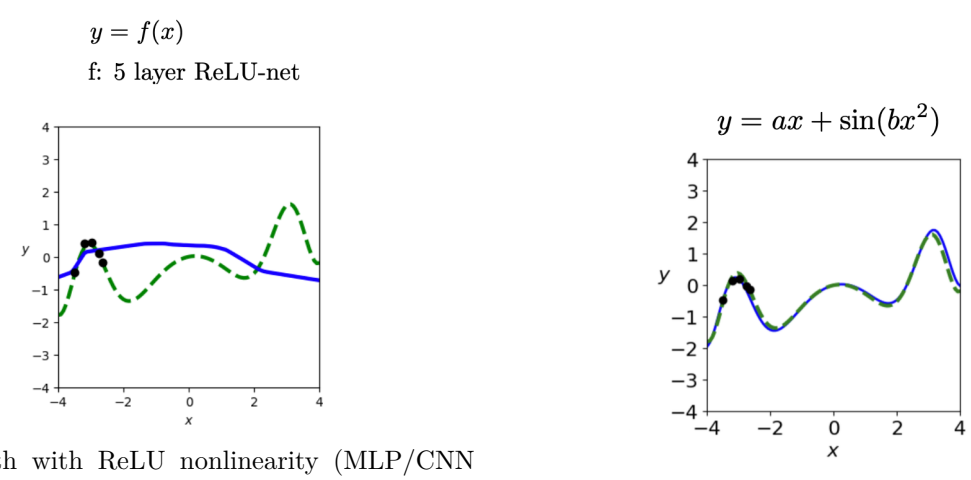
We aim to learn a mapping $\mathcal{X} \to \mathcal{Y}$. Fully connected multilayer perceptrons (MLPs) can approximate any continuous function on compact sets (universal approximation), but they encode weak inductive biases and thus often require large datasets. Convolutional Neural Networks (CNNs) inject three strong, task-relevant biases for signals defined on grids (e.g. images, audio):

(a) **Locality:** each output depends only on a local neighborhood

(b) **Weight sharing:** the same (learned) kernel is reused across spatial locations

(c) **Translation equivariance:** shifting the input shifts the features in the same way (e.g. a bird is a bird no matter where it is in an image)

These constraints reduce the hypothesis class, improving data efficiency and generalization—especially when faced with *out-of-distribution* test data along small translations and deformations, as shown in Figure 4a.



(a) Depth with ReLU nonlinearity (MLP/CNN block).

(b) Using a functional form as an inductive bias.

Figure 4: More constrained architectures can generalize better to out-of-sample inputs.

4.2 Patch View

A useful mental model when thinking about convolutional neural networks is to chop the image into overlapping patches (context windows) and apply the *same* function to each patch to predict something about that patch (e.g. the class of the center pixel, known as semantic segmentation). Using sufficiently large, overlapping patches preserves resolution while providing the necessary context to make an accurate prediction (e.g. it is hard to predict the class of a pixel without sufficient context around it).

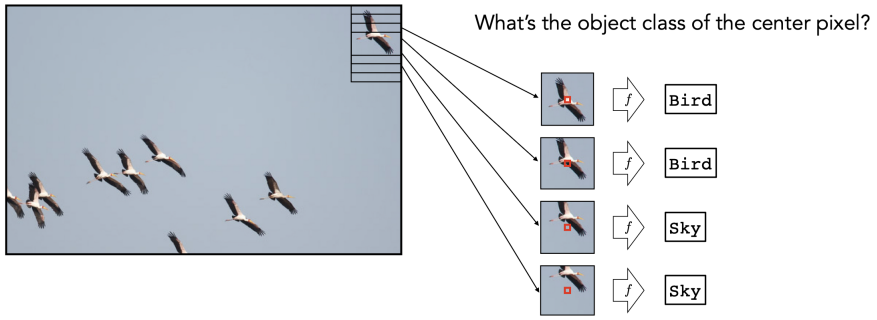


Figure 5: Sliding-window view of convolutional neural networks involves identical processing over local patches

4.3 Convolutional kernels

More precisely, CNNs are just neural networks that are composed of convolutional layers. Futhermore, convolutional layers are just special linear transform using learned weights belonging to a Toeplitz matrix. Graphically, it looks like:

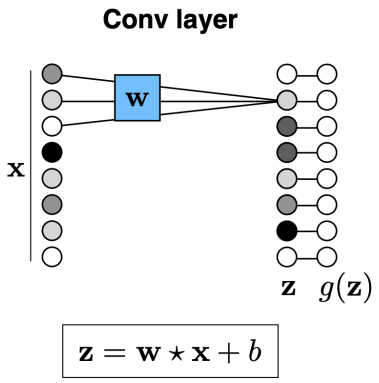


Figure 6: Convolution where w is a learned kernel and b is a learned bias

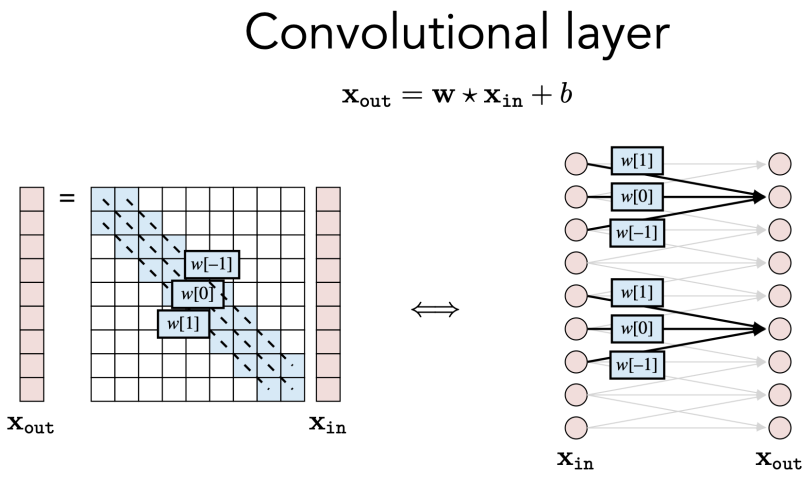


Figure 7: Weight sharing across spatial locations (same filter, all positions). A convolutional layer is just a special kind of linear layer. It is a linear layer whose matrix W is Toeplitz.

For a single-channel 2D input $x_{\texttt{in}} \in \mathbb{R}^{H \times W}$ and a kernel $w \in \mathbb{R}^{K_h \times K_w}$, we typically use the *cross-correlation* kernel:

$$x_{\text{out}}[i,j] = \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} w[u,v]\; x_{\text{in}}[i+u-P_h\,,\; j+v-P_w],$$

where P_h, P_w are zero-padding sizes. With stride S_h, S_w and dilation D_h, D_w, the sampling indices become

$$x_{\text{out}}[i,j] = \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} w[u,v]\; x_{\text{in}}(i \cdot S_h + u \cdot D_h - P_h,\; j \cdot S_w + v \cdot D_w - P_w).$$

Convolutional layers, as defined previously, maintain the spatial resolution of the signal they process. However, commonly it is sufficient, or even desirable, to output a lower resolution. This can be achieved with strided convolution. Dilated convolution is similar to strided convolution but spaces out the filter itself rather than spacing out where the filter is applied to the image. dilation is a way to achieve a filter with large kernel while only requiring a small number of weights. The weights are just spaced out so that a few will cover a bigger region of the image.

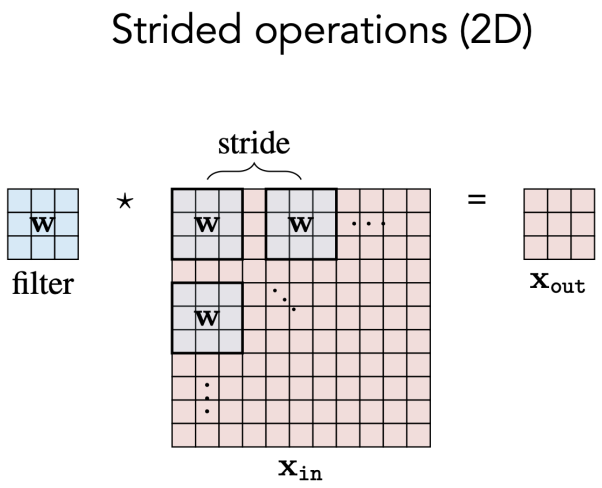


Figure 8: Strided operations combine filtering and downsampling.

4.4 Multi-Channel Inputs & Outputs

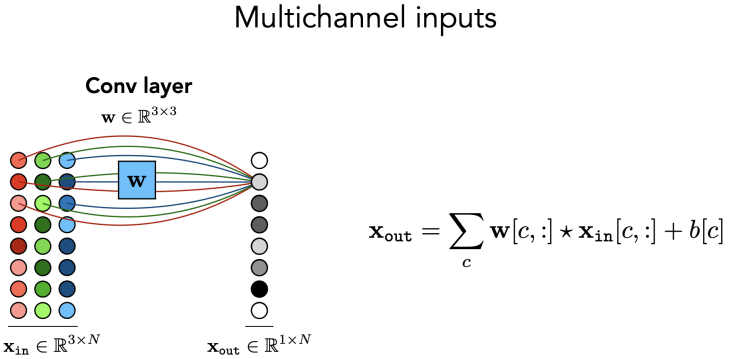


Figure 9: Convolution with C_{in} inputs channels(e.g. RGB).

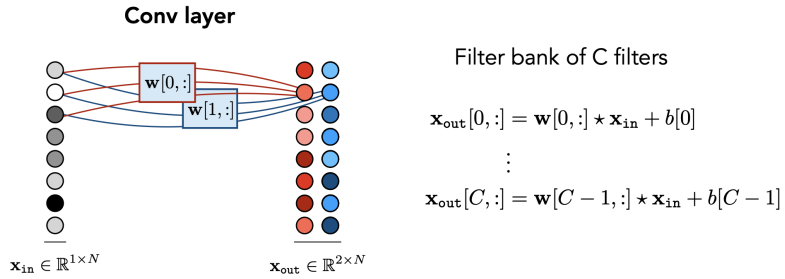


Figure 10: Multiple output channels (C_{out} feature maps) from multiple learned filters.

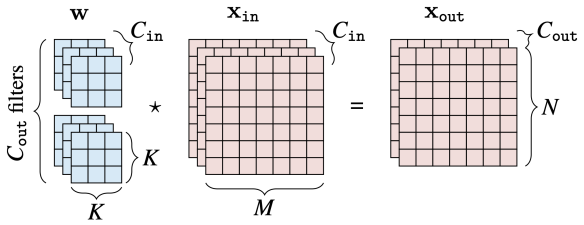


Figure 11: C_{in} input channels and C_{out} output channels

.

Here is what this looks like for an image $\mathbf{x}_{\text{in}} \in \mathbb{R}^{C_{\text{in}}\times N\times M}$, where c_2 indexes the output channel, with $c_2 \in \{0,\dots,C_{\text{out}}-1\}$:

$$\mathbf{x}_{\text{out}}[c_2,:,:] = \sum_{c_1=1}^{C_{\text{in}}} \mathbf{w}[c_1,c_2,:,:] \star \mathbf{x}_{\text{in}}[c_1,:,:] + \mathbf{b}[c_2], \qquad \text{(conv; multi-in–out)} \tag{1.6}$$

Notation for multichannel convolutions can get hard to keep track of, so let's spell out a few of the pieces here, which are also visualized in Figure 1.5:

- $\mathbf{x}_{\text{in}}[c_1,:,:]$ is the c_1-th channel of the input signal.
- The filter bank is C_{out} filters, $\Big[\mathbf{w}[:,0,:,:], \dots, \mathbf{w}[:,C_{\text{out}}-1,:,:]\Big]$, each of which applies one convolutional filter per input channel and then sums the responses over all these filters.
- This convolutional layer maps inputs $\mathbf{x}_{\text{in}} \in \mathbb{R}^{C_{\text{in}}\times N\times M}$ to outputs $\mathbf{x}_{\text{out}} \in \mathbb{R}^{C_{\text{out}}\times N\times M}$.
- The filter bank is represented by a tensor $\mathbf{w} \in \mathbb{R}^{C_{\text{in}}\times C_{\text{out}}\times K\times K}$, where K is the (spatial, square) kernel size.

4.5 Pooling & Invariances

Generally speaking, pooling is a function applied to the output of a convolutional layer which stabilizes the outputs of that layer.

Spatial pooling aggregates over neighborhoods to stabilize features under small translations/deformations:

$$\text{max/avg-pool:} \quad y[i,j] \;=\; \text{pool}(\{x[i+u,j+v] : (u,v) \in \Omega\}).$$

Pooling reduces spatial resolution and can increase the effective receptive field.

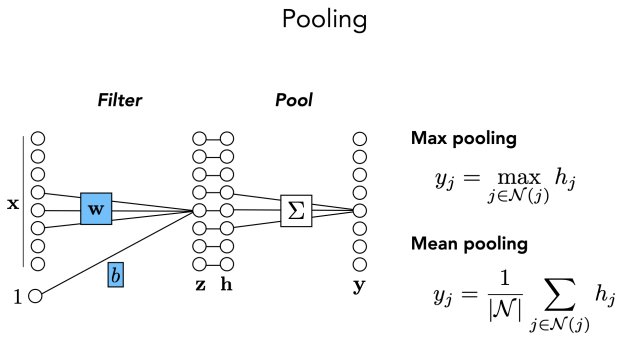


Figure 12: Spatial pooling (e.g. 2×2 max/avg) for local translation invariance.

Channel pooling (less common inside the backbone, but used in heads) aggregates across filters, e.g. ℓ_2-norm or max across channels, to be insensitive to orientation or phase when paired filters detect different directions of the same pattern.

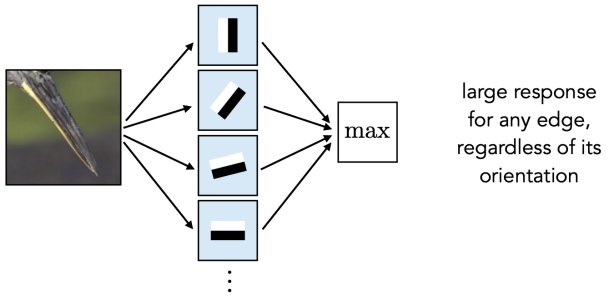


Figure 13: Pooling across channels/filters to merge complementary feature detectors (e.g. edge orientations).

4.6 Downsampling

Downsampling layers transform the input tensor to an output tensor that is smaller in the spatial dimensions. As we say strided and pooling convolution are types of downsampling layer.

Upsampling layers perform the opposite transformation, outputting a tensor that is larger in the spatial dimensions than the input.

4.7 U-Net and ResNet

Encoder-decoders force the signal to pass through a bottleneck. However, the decoder may fail to be able to output high frequency details such as in a semantic segmentation network. To circumvent this problem, we can add skip connections that shuttle information directly across blocks of layers in the net. Adding skip connections to an encoder-decoder results in an architecture known as a U- Net. In this architecture, layer ℓ is connected directly to layer $(\ell - l)$. The output of a skip connection must somehow be reintegrated into the network. U-nets do this by concatenating the activations from the prior layer to the activations on the later layer, along the channel dimension. This architecture can maintain the information-bottleneck of the encoder-decoder, with its incumbent benefits in terms of memory and compute efficiency and forced abstraction, while also allowing residual information to flow through the skip connections, thereby not sacrificing the ability to output high-frequency spatial predictions.

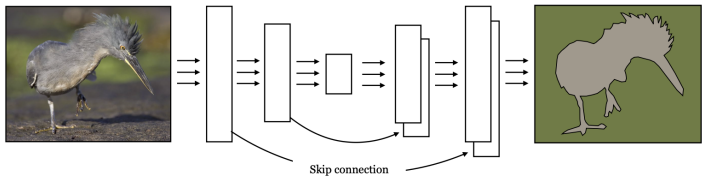


Figure 14: Encoder–decoder with skip connections (U-Net).

Residual connections (ResNets): learn a residual function F as part of an identity path:

$$x_{\text{out}} \;=\; x_{\text{in}} \;+\; F(x_{\text{in}};\theta),$$

It is easy for a residual block to simply perform an identity mapping, it just has to learn to set F to zero. Because of this, if we stack many residual blocks in a row, it can end up that the net learns to use only a subset of them. If we set the number of stacked residual blocks to be very large then the net can essentially learn how deep to make itself, using as ResNet many blocks as necessary to solve the task. ResNets often exploit this fact by being very deep; for example, they may be hundreds of blocks deep

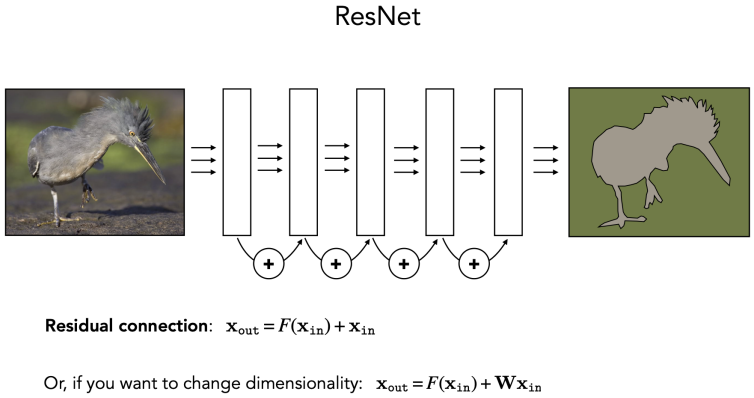


Figure 15: Residual (additive) skip: $y = x + F(x)$ (ResNet).

4.8 Limitations of CNNs

CNNs are built around the idea of local context: everything you need to know about a pixel Not well-suited to modeling long distance relationships (either between words or pixels) which is a product of the receptive field. Adding more layers does increase the receptive field, but far away patches won't interact until deep in the network, precluding understanding complex relationships between distant patches. Fully connected linear layers enable global information sharing, but are data and parameter inefficient. Lastly, CNNs are translational invariant but not rotational invariant (but cane be with Rotation-Invariant Coordinate Convolution (RIC-C, which achieves invariance to rotations around the input's center without extra parameters or data augmentation) or Bessel-Convolutional Neural Networks (B-CNNs, which use Bessel functions to create layers that are invariant to continuous rotations).

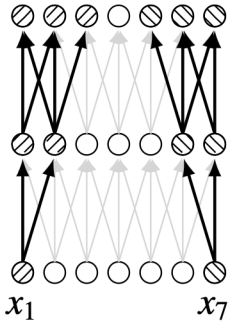


Figure 16: Caption

5 Transformers

Recall that the universal approximation theorem says we can learn any* function with a sufficiently wide MLP, or equivalently that MLPs are universal function approximators. However, our chosen neural network architecture greatly influences our ability to learn a given function because architectures often automatically encode function attributes in what we call *inductive bias*. Two important features that any "good" architecture should encode are an abillity to share information globally (as in the case with fully connected MLPs), but only parts of the image that are relevant (as is the case with CNNs). As an alternative to fully connected MLPs, CNNs, and ResNets, transformers are an alternative architecture that support both global yet selective information sharing using tokenization, positional encodings, and attention.

5.1 Tokens are just vectors

A token is just a vector, which can be thought of as a high-dimensional analog of a neuron. One can have an array of neurons or an array of tokens.

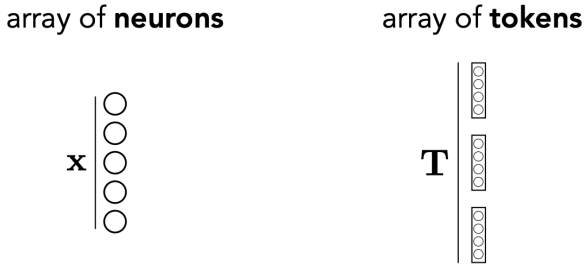


Figure 17: a 5×4 array of neurons vs a 5×4 array of tokens

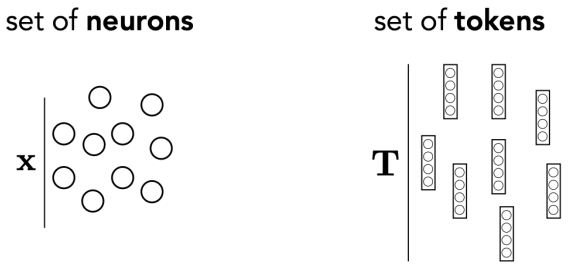


Figure 18: A set of neurons vs a set of tokens

5.2 Tokenizing the input data

When operating over neurons in MLPs, we represent the input as an array of scalar-valued measurements (e.g., pixel values from a flattened image). Analogously, when operating over tokens in a transformer, we represent the input as an array of vector-valued measurements. You can tokenize anything, but the general idea is to chop the input into flattened chunks and project each chunk a vector (this projection is usually to a lower dimensional space using $W_{\texttt{tokenize}}$).

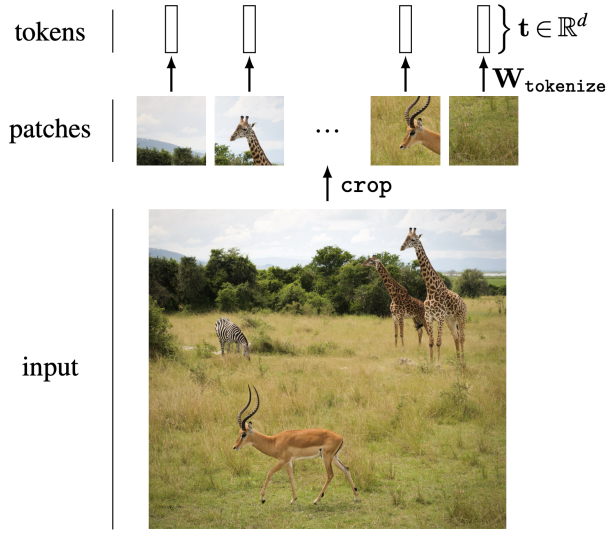


Figure 19: Tokenizing an image by chopping it into flattened patches and projecting into a lower dimensional space using the matrix W_{tokenize}

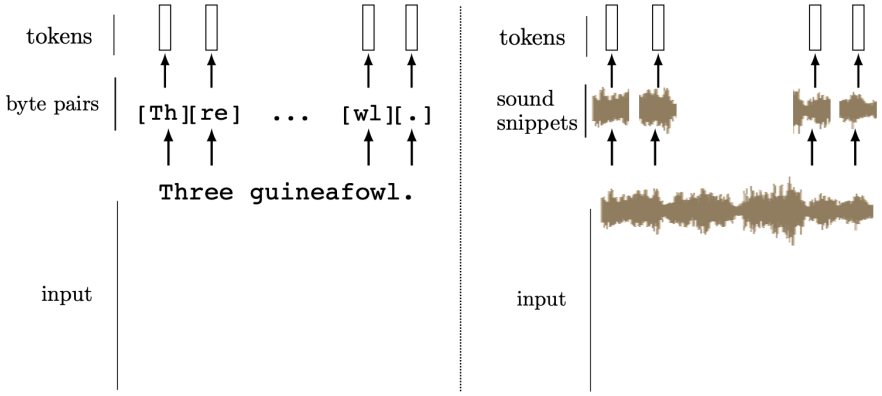


Figure 20: Tokenizing text and sound simply maps our data to a set of tokens

5.3 Stacking tokens into matrices

Recall that to feed our data into an MLP we stacked our input neurons to form a vector. In a similar way, we stack our tokens into a token matrix T before passing them into a transformer.

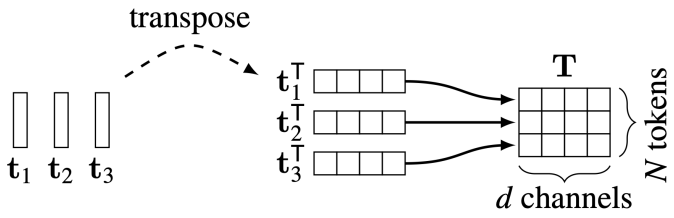


Figure 21: Tokenize are stacked into a matrix before being passed into a transformer

5.4 Token-wise operations

Recall that an MLP was built using linear combinations of neurons. Similarly, a transformer will involve linear combinations of tokens.

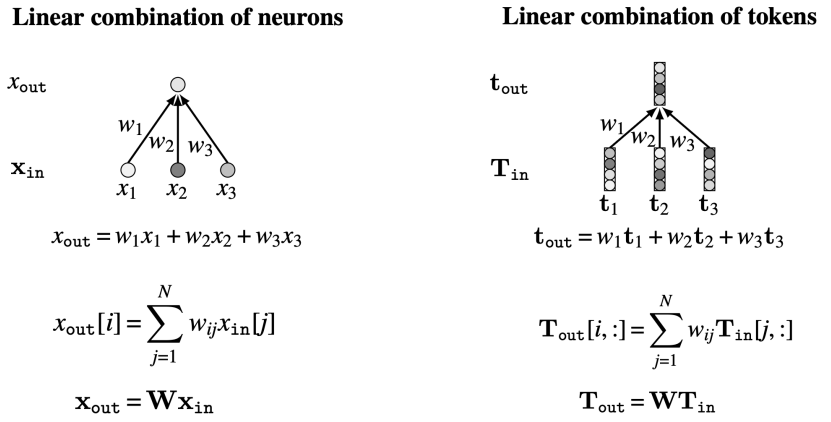


Figure 22: Taking linear combinations of tokens (vectors)

Additionally, MLPs used nonlinearities over neurons. Meanwhile, transformers use nonlinearities over tokens.

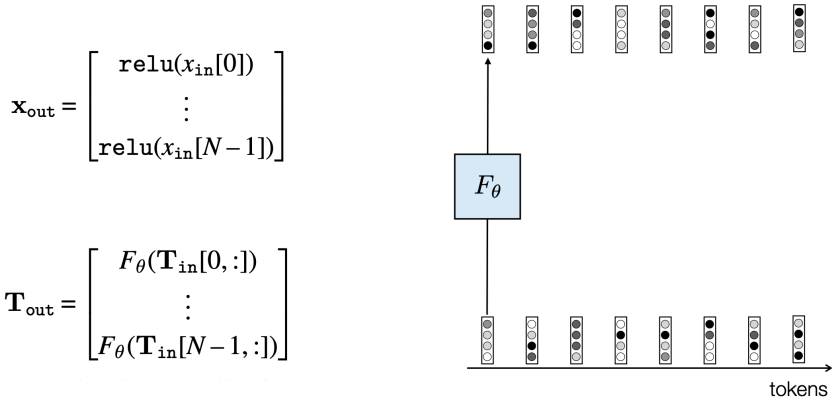


Figure 23: A nonlinear function that maps tokens to tokens

5.5 What does a transformer do?

At its core, a transformer is a layer in a neural network (so more generally just a function) that maps a token matrix to a token matrix. However, transformer layers have a special architecture that encodes an inductive bias which is beneficial for learning, especially on natural language tasks. This inductive bias is known as *attention*. In the following subsections, we give the mathematical formulation transformers and attention. However, we neglect any sort of motivation as to why transformers are the "right" architecture (indeed, this would first require being more precise about the inductive bias we hope for). Instead, we relegate this exposition to domain specific sections – such as natural language processing in Section **??** and vision transformers in Section **??** – because it will only become clear why transformers or attention work when we have a specific task and input data to contextualize them within.

Figure 24: Transformers map a token matrix to a token matrix

5.6 Self-attention

Transformers, sometimes known as an self-attention layers, accept a token matrix $T_{\texttt{in}}$ as input, which is the vector of N tokens. Each of these tokens are passed through the matrices W_q, W_k, W_v produce three items, a query token, a key token, and value token, respectively. These tokens are stacked into the query, key, and value matrices $Q_{\texttt{in}}$, $K_{\texttt{in}}$, $V_{\texttt{in}}$, respectively. The queries and key matrix $Q_{\texttt{in}}$, $K_{\texttt{in}}$ then produce a self-attention matrix A, which encodes the similarity between each token in $T_{\texttt{in}}$ (via the key $K_{\texttt{in}}$ and the query $Q_{\texttt{in}}$). More specifically, each element in A is the similarity between a query token and a key token (post normalization from applying the `softmax` along the rows of $Q_{\texttt{in}}K_{\texttt{in}}^{\mathsf{T}}$). The output of a transformer is again N tokens contained in the rows of $T_{\texttt{out}}$.

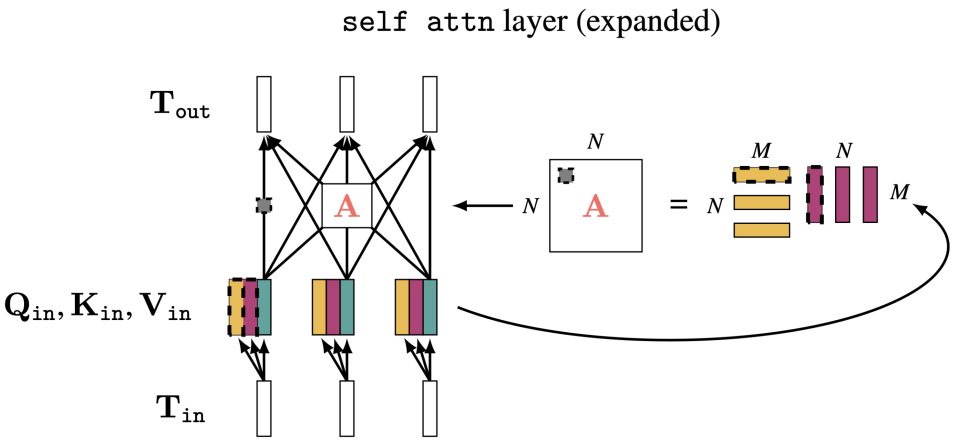


Figure 25: A self attention layer where all N input tokens generate queries

$$
\begin{aligned}
\mathbf{Q}_{\texttt{in}} &= \begin{bmatrix} \mathbf{q}_1^{\mathsf{T}} \\ \vdots \\ \mathbf{q}_N^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_q \mathbf{t}_1)^{\mathsf{T}} \\ \vdots \\ (\mathbf{W}_q \mathbf{t}_N)^{\mathsf{T}} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_q^{\mathsf{T}} \quad \triangleleft \quad \text{query matrix} \\
\mathbf{K}_{\texttt{in}} &= \begin{bmatrix} \mathbf{k}_1^{\mathsf{T}} \\ \vdots \\ \mathbf{k}_N^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_k \mathbf{t}_1)^{\mathsf{T}} \\ \vdots \\ (\mathbf{W}_k \mathbf{t}_N)^{\mathsf{T}} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_k^{\mathsf{T}} \quad \triangleleft \quad \text{key matrix} \\
\mathbf{V}_{\texttt{in}} &= \begin{bmatrix} \mathbf{v}_1^{\mathsf{T}} \\ \vdots \\ \mathbf{v}_N^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_v \mathbf{t}_1)^{\mathsf{T}} \\ \vdots \\ (\mathbf{W}_v \mathbf{t}_N)^{\mathsf{T}} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_v^{\mathsf{T}} \quad \triangleleft \quad \text{value matrix}
\end{aligned}
\tag{26.5}
$$

$$\begin{aligned} \mathbf{A} &= f(\mathbf{T}_{\texttt{in}}) = \texttt{softmax}\Big(\frac{\mathbf{Q}_{\texttt{in}}\mathbf{K}_{\texttt{in}}^{\mathsf{T}}}{\sqrt{m}}\Big) \quad \triangleleft \qquad \text{attention matrix} \\ \mathbf{T}_{\texttt{out}} &= \mathbf{A}\mathbf{V}_{\texttt{in}} \end{aligned} \tag{26.6}$$

5.7 Multihead Self-Attention

Despite their power, self-attention layers are still limited from only having one set of query/key/value projection matrices W_q, W_k, W_v per input token matrix $T_{\texttt{in}}$. This limitation is especially relevant in situations where we need to learn multiple orthogonal representations of our input object (e.g. color and size) to solve a task.

Transformers address this limitation using *multihead self-attention* (MSA), which involves running k self-attention layers in parallel. All k heads of the multihead self-attention layer are applied to the same input token matrix $\mathbf{T}_{\text{in}}$ and return k output token matrices, $\mathbf{T}^1_{\text{out}}, \ldots, \mathbf{T}^k_{\text{out}}$. To merge these outputs, we concatenate all of them and project back to the original dimensionality of $\mathbf{T}_{\text{in}}$. These steps are shown below:

$$\mathbf{T}^i_{\text{out}} = \text{attn}^i(\mathbf{T}_{\text{in}}) \quad \text{for } i \in \{1, \ldots, k\}, \tag{1}$$

$$\overline{\mathbf{T}}_{\text{out}} = \begin{bmatrix} \mathbf{T}^1_{\text{out}}[0,:] & \cdots & \mathbf{T}^k_{\text{out}}[0,:] \\ \vdots & \ddots & \vdots \\ \mathbf{T}^1_{\text{out}}[N-1,:] & \cdots & \mathbf{T}^k_{\text{out}}[N-1,:] \end{bmatrix} \qquad \overline{\mathbf{T}}_{\text{out}} \in \mathbb{R}^{N \times kv}, \tag{2}$$

$$\mathbf{T}_{\text{out}} = \overline{\mathbf{T}}_{\text{out}}\, \mathbf{W}_{\text{MSA}} \qquad \mathbf{W}_{\text{MSA}} \in \mathbb{R}^{kv \times d}. \tag{3}$$

Here v is the dimensionality of the value tokens and d is the dimensionality of the code vectors of the output tokens in $T_{\texttt{out}}$ (where some recommend setting $d = kv$). More generally, the job of the matrix $\mathbf{W}_{\text{MSA}}$ is to merge all the heads, and its values are learnable parameters. The other learnable parameters of MSA are the query, key, and value projections W^i_q, W^i_k, W^i_v for *each* of the k attention heads.

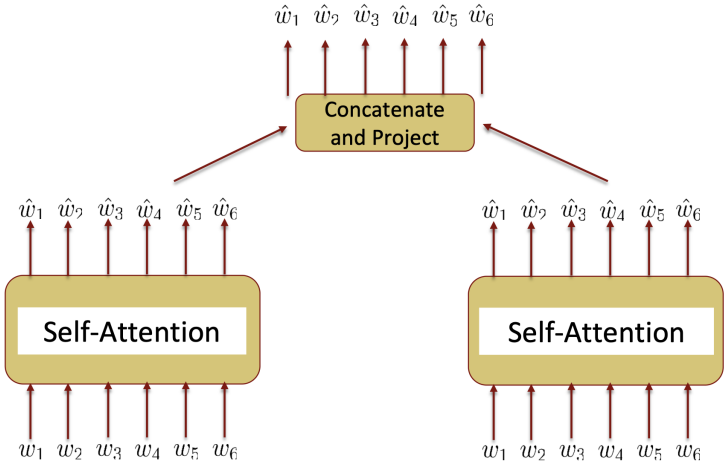


Figure 26: Multihead self attention uses seperate attention heads, then combines their output by concatenating and projecting.

5.8 Attention to external questions

We've been working thus far in the case of transformer self attention – where each token is projected into the space of queries using a learned W_q. This transformer mechanism is quite applicable because it is simply a layer in a neural network that makes a token matrix to a token matrix.

However, transformer self attention can be easily modified to accommodate the setting where we have an *external question* we want to answer, as shown below. In this setting, the user's question, represented as a single vector, is first passed through the query matrix W_q to obtain a single query token $q_{\texttt{question}}$. We then compute the N similarity scores between this token and the N key tokens from W_k. The attention matrix (aka vector) is then just the `softmax` of this resulting vector, and the output $T_{\texttt{out}}$ is just a matrix which is the analogy of $T_{\texttt{out}} = A\ V_{\texttt{in}}$ from the transformer MSA case.

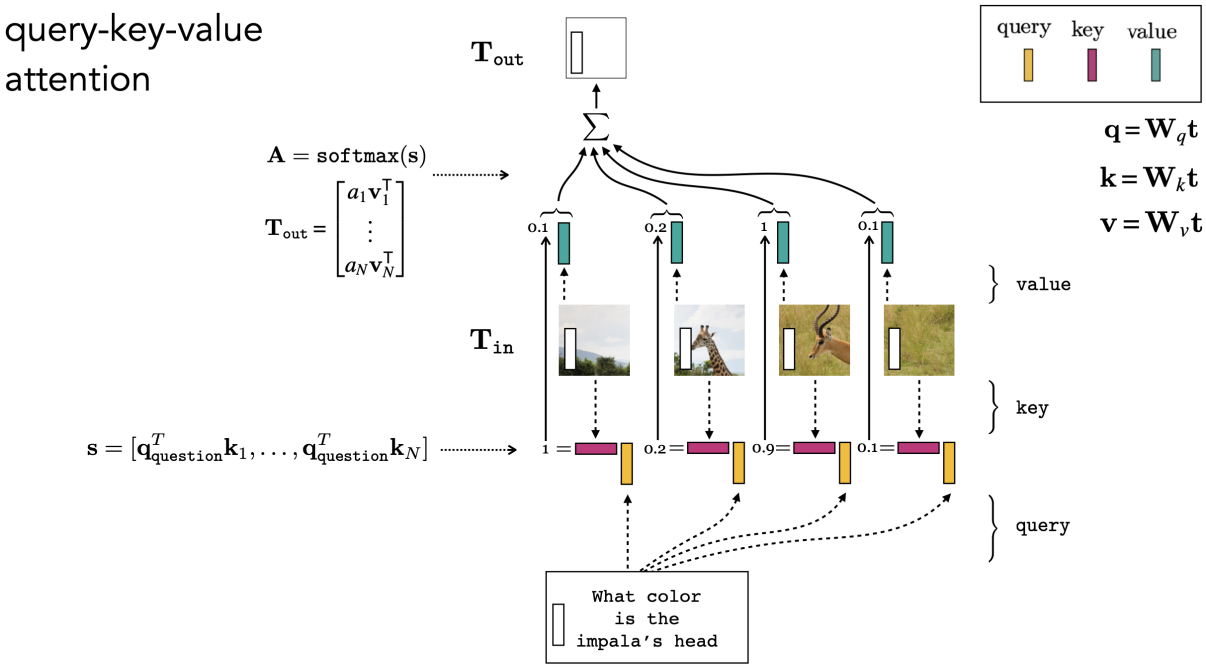


Figure 27: The foundation of an attention layer for N input tokens and 1 query

6 Natural language models

Section 5 covered the transformer attention architecture, which is simply a layer in a neural network that maps token matrices to token matrices. However, we've yet to describe what attention really means, why attention is a reasonable inductive bias for our architecture to encode, or how to use transformers to solve tasks. We now do so for natural language processing, with particular emphasis on why transformers enable us to learn *contextual embeddings of input tokens*.

6.1 Uses of language models

Language models accept text as input and can be used for

- judging sentiment text
- refining text through filtering
- summarizing long-form text
- extract data out from free-form text
- text generation (Q&A)

Language modeling was first accomplished using hand-crafted rules based on linguistics, then statistical machine learning, then recurrent neural networks, then transformers (2017 onward). From here on out, we will design a neural network $f_\theta(x)$ that maps text x to some desired output $y = f_\theta(x)$.

6.2 Pre-processing/text vectorization: standardize, tokenize, encode

The first step in our pipeline is to represent our text as a vector, known as *text vectorization* or *pre-processing*. The general framework of pre-processing is Standardize → Tokenize → Encode.

6.3 Text corpus *standardization*

- Strip capitalization, often punctuation and accents
- Strip 'stop words' (e.g., a, the, it)
- Stemming (e.g., ate, eaten, eating, eaten > [eats])(sometimes)

6.4 Text corpus *tokenization* to produce a vocabulary

Tokens can be single characters, multiple characters ("sub words"),single words or multiple consecutive words ("n grams"). For text classification tasks, the default is either single words (unigrams) or consecutive words (bigrams). When this is done for every sentence in our training dataset, we have a list of distinct tokens, this is our *vocabulary*.

6.5 Encoding the vocabulary

The exact steps depend on what the task at hand is

- For text classification tasks Standardizing & Tokenizing involve lowercasing the text and splitting on whitespace.
- For text generation LLM tasks, Standardizing & Tokenizing involve byte pair encoding (splitting on subwords)

6.6 Text embedding: encode the vocabulary in a vector space

Text embedding encodes each token (word) in our vocabulary as a vector. One naive way to do this is one-hot encoding. The dimension of this one-hot encoding is equal to the size of the vocabulary (+1 to account for ⟨`unk`⟩. If a single token is embedded as a one-hot vector, then a sentence is simply a matrix of one-hot vectors. While it's possible to feed this into a neural

network, it is very sparse and thus wasteful.

One solution to these problems is to use either *count encodings* or *multi-hot* encodings to "project" this sparse matrix down to a vector. However, this general framework, known as the *bag of words model* has a few failings. First, because the one-hot encodings of a given token have no semantic meaning (e.g. similar words are not mapped to similar one-hot vectors) the resulting multi-hot or count encoding vector of a sentence does not capture any semantic meaning about the sentence. Secondly, neither multi-hot or count encodings capture the order of the tokens in the sentence.

6.7 Learned text embeddings

A more sophisticated idea is to embed tokens as vectors in a in a way that captures the meaning of the words they represent, known as *learned token embeddings*. There's many methods that aim to do this, but the GloVe algorithm is a common approach.

Algorithm 1 GloVe algorithm: learned token embeddings from empirical probabilities

Require: Token vocabulary $\mathcal{V}$, and empirical probabilities $\mathbb{P}(i \mid g)$ for all tokens $i, g \in \mathcal{V}$ (often computed from Wikipedia)

1: Initialize an arbitrary token embedding (vector) w_i and a context embedding (vector) $\hat{w}_i$ for each token (word) $i \in \mathcal{V}$
2: **repeat**
3: Sample a training triple $(i, s, g) \in \mathcal{V}^3$
4: Compute the predicted log–ratio

$$\widehat{r} = (w_i - w_s)^\top \hat{w}_g$$

5: Compute the target log–ratio

$$r = \log \frac{\mathbb{P}(i \mid g)}{\mathbb{P}(s \mid g)}$$

6: Perform an SGD update step on $w_i, w_s, \hat{w}_g$ to minimize the squared error $(\widehat{r} - r)^2$
7: **until** convergence
8: **return** Token embeddings w_i for all $i \in \mathcal{V}$

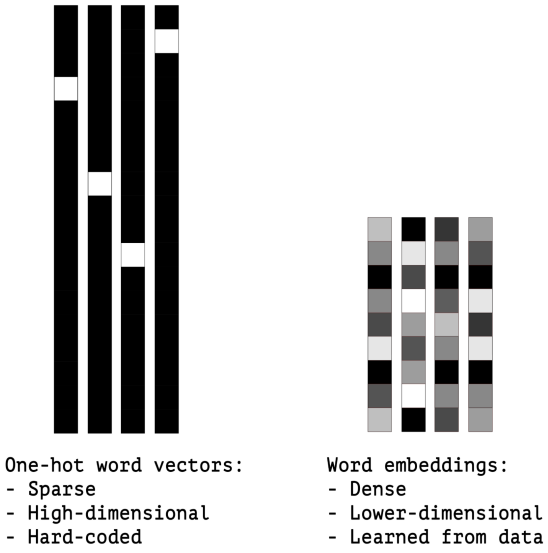


Figure 28: GloVe embeddings vs one-hot embeddings

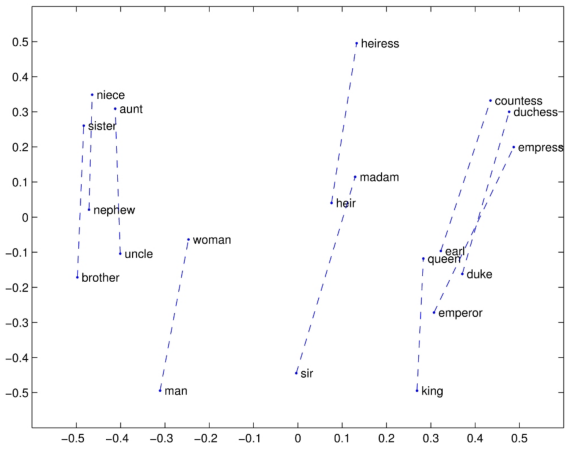


Figure 29: GloVe embeddings form a vector space

6.8 Learned contextual embeddings via transformers

Embeddings should incorporate not just the meaning of the word, but also the meaning of the surrounding words through context. The GloVe model did not do this. But transformers do enable contextual embeddings. In fact, this is exactly what attention does! Multihead self attention is then a generalization of this because it allows us to "attend to" the multiple patterns (tense, tone, etc) that may be present in a single sentence. Therefore, different attention 'heads' learn different patterns.

6.9 Positional encodings

The transformer architecture that enables learned contextual embeddings does not take word order into account. Positional encodings adds each word's position in the sentence to its learned contextual embedding, so that the final embedding is the sum of two vectors (one learned from context, and the other a function of position).

6.10 Transformer encoder architecture

1. The input embedding can simply be random embeddings or pretrained from GloVe
2. Then add a position dependent embedding to represent the position of each word in sentence
3. Next, pass the embedding through multi-headed attention to get a context dependent representation
4. Finally, pass all this through a simple feed-forward network to introduce nonlinearity
5. Optional: layer normalization and residual connections (connect the positional input embeddings directly to the feed-forward layer to bypass the multi-head self attention layer, and from the multi head self attention layer directly to the contextual embeddings, to bypass the feed forward layer)

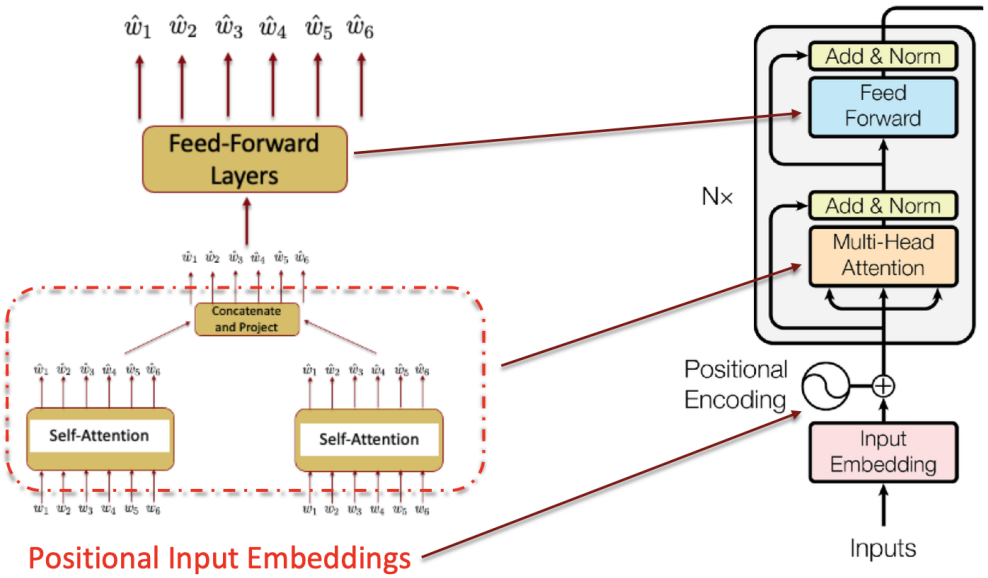


Figure 30: The transformer encoder architecture

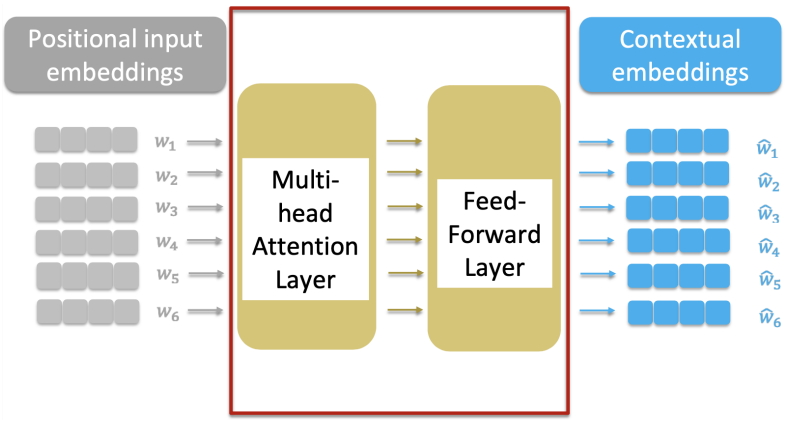


Figure 31: The transformer encoder architecture

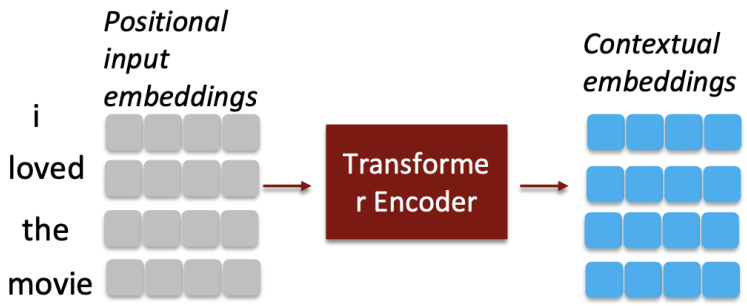


Figure 32: The transformer encoder architecture operates on a per token basis! We pass in a group of tokens that will attend to each other, and get the same number of tokens back.

6.11 $\langle CLS \rangle$ tokens

Recall that transformers operate over tokens, meaning that for every token they produce an embedding. A useful trick that is helpful for certain LLM tasks that require knowledge of the *entire sentence* (e.g. sentiment analysis) is to use add a special token, called a $\langle$`CLS`$\rangle$ token, that at the beginning of each sentence and just use its output embedding as the embedding of the sentence.

6.12 Self-supervised learning trains LLMs using masking

What data should we use to train an transformer based-LLM? One idea is to predict a subset of the input data using the rest of the input. The transformer learns to predict the masked words from the rest of the sentence.

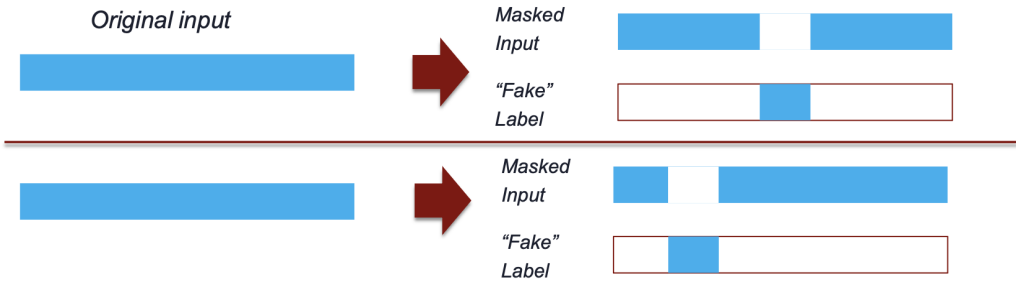


Figure 33: Masking modifies the original input data to create "fake" (input, label)

6.13 BERT

If we train a transformer on "fill-in-the-blank" masked text from the internet then we get BERT, which is a model that has learned contextual embeddings. Token we pass in, BERT returns a Softmax over the vocabulary to predict the masked word
Luckily, BERT comes pre-trained with $\langle$`CLS`$\rangle$ embeddings.

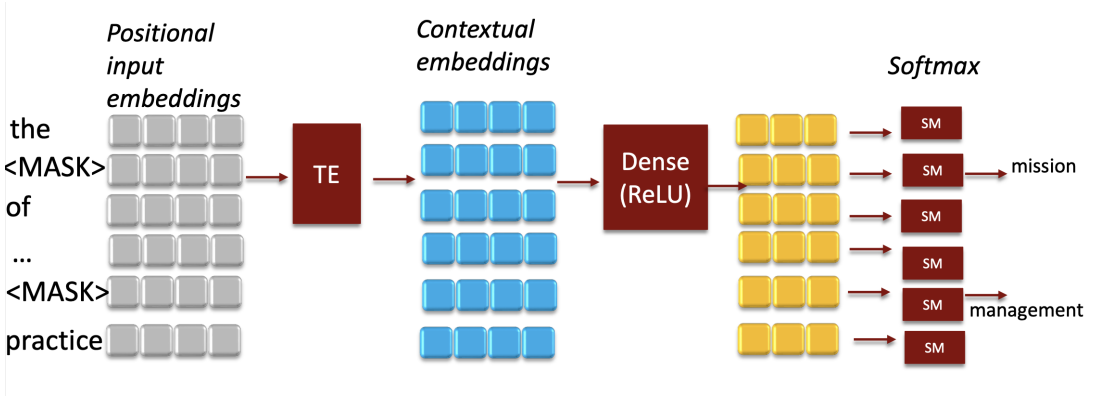


Figure 34: BERT is a Transformer Encoder model trained to predict the MASKED words

6.14 Semantic search via cross encoders and bi-encoders

Semantic search is an approach to information retrieval (e.g. search engine technique) that seeks to improve search accuracy by capturing the searcher's intent/contextual meaning. Semantic search is a more powerful strategy as opposed to a simple keyword-to-keyword match lookup. Semantic search is done with a bi-encoder, cross-encoder, or a combination.

- A cross-encoder uses a transformer to encode a query and database documents side by side, which is beneficial since attention allows the ⟨CLS⟩ embedding to depend on both. However, this is very slow since it requires a transformer evaluation to get the ⟨CLS⟩ embedding for every (query, document) pair.
- A bi-encoder encodes a query and documents separately then simply measures their similarity using a dot-product.

by understanding the searcher's intent and the contextual meaning of terms as they appear in the searchable databace, whether on the Web or within a closed system, to generate more relevant results.

7 Large language models (LLMs)

We now discuss how to actually *use* transformers to train large language models, covering training via masking, BLANK, and BLANK.

7.1 Training a tranformer encoder using masking

Recall that the job of a transformer encoder is to generate token embeddings, in particular, we pass in a group of tokens and the transformer encoder returns contextual embeddings for every word in an input sentence. To train a transformer encoder on natural language, we used masking to create a self-supervised learning problem. In this setting, the transformer was given a *masked input* such as

```
MIT is <MASK> in Cambridge, <MASK> and was <MASK> in 1861.
```

The transformer encoder is trained to predict these masked tokens.
for us to train a transformer encoder,. Now we can use the contextual embedding for the masked tokens to predict the masked token. The output layer is a softmax over the vocabulary for each token in the sentence (e.g. giving us a distribution for each of the missing words). This approach is essentially "fill in the blanks" of the missing words.
Another self-suerpvised learning technique for text inputs is *next word prediction.*

7.2 Next word prediction

7.3 Biases in instruction tuning

This pipeline can be brittle because the human feedback is what we want to see, not what is factually correct, it will also replicate biases in human answers.

7.4

8 The Vision Transformer

We've introduced transformers from the perspective of language models, where we recall that their main role was to learn contextual token embeddings. This architecture can similarly be used to solved vision tasks (previously the responsibility of convolutional neural nets CNNs).

8.1 Key tasks in computer vision

- Classification: (classifying an image into 1 of 1000 categories), hence the output layer is a softmax over the 1000 classes
- Classification + Localization: classifies the image *and* detect where that object is located within the image using a bounding box. This requires an output layer consisting of a softmax over the 1000 *and* the 4 coordinates that define the bounding box.
- Semantic segmentation: labeling each pixel in an image as corresponding into one of N categories.

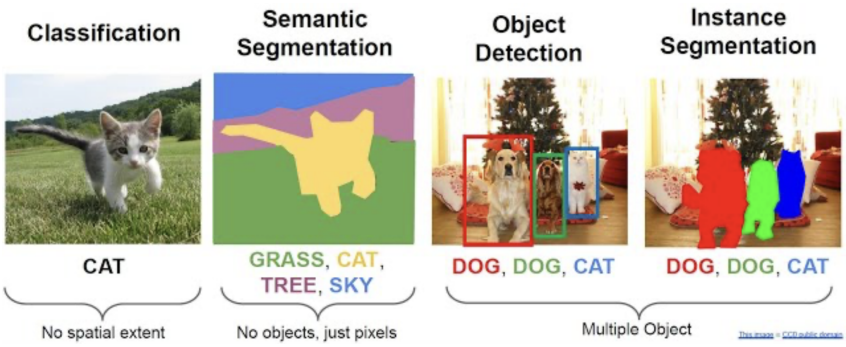


Figure 35: Common tasks in computer vision

8.2 Before vision transformers

Humans process images based on *features*, global attributes of the image. Before vision transformers, the prevailing approach to image processing was to build good feature detectors using hand-engineered methods (e.g. find edges, find "T's" to identify faces). The reason convolutional neural networks worked so well si that they facilitated automated feature detection. Transformers will take this one step further

8.3 Review of the transformer

Recall the general pipeline of a transformer: we are given a sentence, then tokenize it by chopping it into words and encoding those words as vectors. After adding a positional encoding for each token, we pass these tokens through a transformer encoder to obtain contextual embeddings. Notice that the key insight in a transformer is learned contextual embeddings is actually quite general and is not restricted to language. The only hurdle we face in applying it to other data domains, such as images, is how do tokenize the data.

8.4 Tokenizing images

How should we tokenize our image?

1. Chop the image $(3 \times H \times D)$ into patches
2. Flatten each of these patches into a vector
3. Project these long flattened patches into a lower dimensional space using a projection matrix

4. Encode position by adding a position vector
5. Add a $\langle CLS \rangle$ token

8.5 ViT architecture

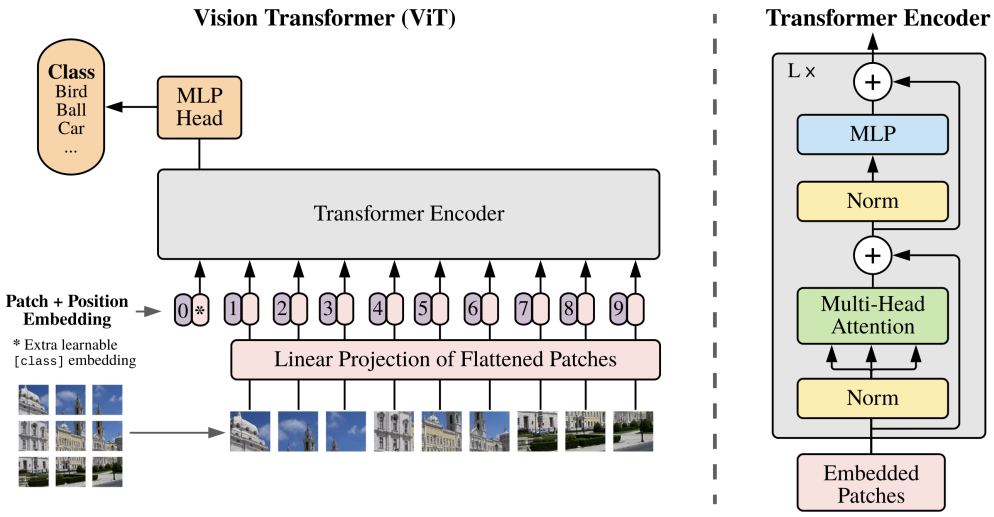


Figure 36: We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence.

8.6 Transfer learning using ViT's

The above showed that the fundamental job of a vision transformer (ViT) is to generate contextual embeddings of tokenized image patches. We now show how to use transfer learning with an existing ViT to build a classifier for a specific problem. In particular, we fine-tune a pretrained ViT for our specific task. The steps are as follows:

1. Collect a few hundred labeled examples for the specific problem (e.g. image-label pairs).
2. Find an trained model from a model hub that has been pretrained on the same type of input data (images), for example this is a common pretrained ViT used in image recognition.
3. Pre-process and tokenize the labeled examples in the same way as done in the pretrained model
   - For example, we often resize the image to $(224 \times 224 \times 3)$, then dived into patches of size $(16 \times 16 \times 3)$ to produce 196 patches.
   - then flatten each of these patches into 768-dimensional vectors
   - then project into a lower dimensional subspace using the same matrix as in the pretrained model
4. "Attach" an NN (e.g. an MLP) to the output of the pretrained transformer
5. Train the network using the labeled examples

(a) Either "freeze" the weights of the pretrained Transformer and just optimize the weights of the custom NN

(b) or optimize ALL the weights in the transformer and the custom NN

The fundamental constraint for the number of patches I can chop the image into is (1) the length of the context window, since the memory and computation need to compute attention scales with N^2 where N is the length of the context window and (2) the size of our dataset

8.7 Data augmentation in ViT's

It is common to augment our original dataset using perturbs images, but that still have the same label. This makes our model robust to changes we might see on the test dataset

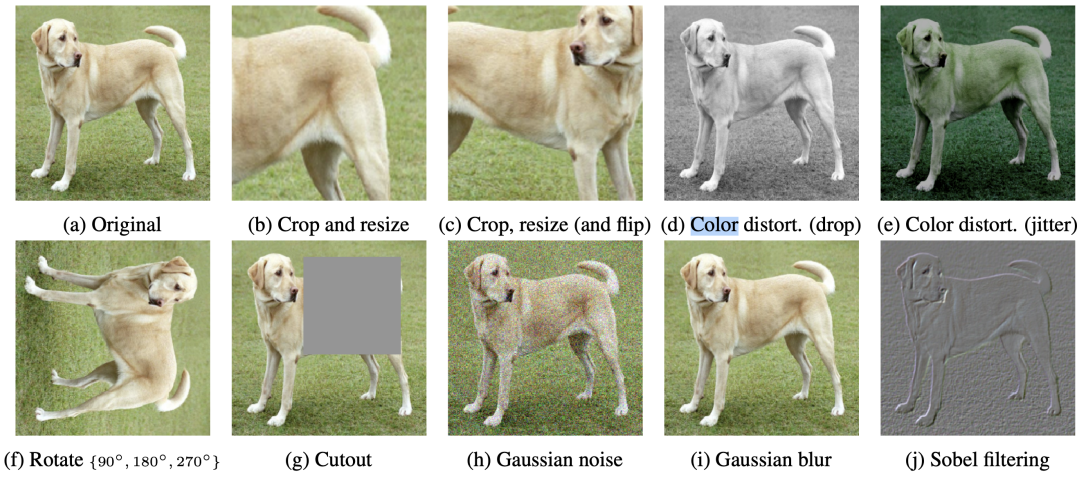


Figure 37: Caption

8.8

8.9

9 L1 and L2 Regularization

9.1 Regularized Empirical Risk Minimization

Given data $\{(x_i, y_i)\}_{i=1}^n$, parameters $w \in \mathbb{R}^d$, and loss $\ell(\hat{y}, y)$ (e.g., squared loss for regression, logistic loss for classification), regularization augments empirical risk with a penalty:

$$\min_{w \in \mathbb{R}^d} \underbrace{\frac{1}{n} \sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big)}_{\text{data fit}} + \lambda\, \Omega(w)$$

where $\lambda \geq 0$ is fixed and chosen. Equivalently, regularization can be written in constrained form as

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big) \quad \text{s.t.} \quad \Omega(w) \leq t,$$

where there is a one-to-one correspondence between λ and t (via Lagrangian duality).

9.2 L2 Regularization (Ridge, Weight Decay)

Penalty. $\Omega_2(w) = \frac{1}{2}\|w\|_2^2$ (the $\frac{1}{2}$ is a convenient constant).

Objective.

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big) \;+\; \lambda \frac{1}{2} \|w\|_2^2.$$

Closed form for linear regression. For $f_w(x) = x^\top w$ and squared loss, with $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$:

$$\hat{w}_{\text{ridge}} \;=\; (X^\top X + \lambda I)^{-1} X^\top y.$$

In the singular-value basis of X (with singular values $\{\sigma_j\}$), ridge shrinks along each principal direction by

$$\text{shrinkage factor} \;=\; \frac{\sigma_j^2}{\sigma_j^2 + \lambda} \in (0, 1).$$

Gradient descent view (weight decay). With step size η,

$$w \leftarrow (1 - \eta\lambda)\, w \;-\; \eta\, \nabla_w \Big(\tfrac{1}{n} \sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big) \Big),$$

i.e., coefficients are multiplicatively decayed every step.

Geometric intuition. The L2 constraint $\|w\|_2 \leq t$ is a ball (round). The optimum tends to *shrink* all coordinates smoothly, rarely setting any exactly to zero. It stabilizes solutions (especially under multicollinearity) and reduces variance.

When to use L2.

- You expect *many small/medium* effects (dense signal); feature selection is not essential.
- Inputs are *correlated* and you want a stable, unique solution.
- You want smooth shrinkage and good numerical conditioning (e.g., $X^\top X$ ill-conditioned).
- Deep nets: standard "weight decay" to limit overfitting.

9.3 L1 Regularization (Lasso)

Penalty. $\Omega_1(w) = \|w\|_1 = \sum_{j=1}^{d} |w_j|$.

Objective.

$$\min_w \ \frac{1}{n}\sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big) \ + \ \lambda \, \|w\|_1.$$

Sparsity via subgradients / KKT. At optimum, coordinates satisfy

$$\begin{cases} \frac{\partial}{\partial w_j}\big(\frac{1}{n}\sum_i \ell\big) = -\lambda \operatorname{sign}(w_j), & w_j \neq 0, \\ \big|\frac{\partial}{\partial w_j}\big(\frac{1}{n}\sum_i \ell\big)\big| \leq \lambda, & w_j = 0, \end{cases}$$

so many w_j are *exactly* zero when their correlation with the residual falls inside $[-\lambda, \lambda]$.

Proximal/coordinate update (soft-thresholding). The proximal operator for $\|w\|_1$ is the soft-thresholding map

$$\mathcal{S}_\alpha(z) \ = \ \operatorname{sign}(z)\, \max\{|z| - \alpha,\, 0\}.$$

For linear regression with orthonormal columns ($X^\top X = I$), the lasso solution is

$$\hat{w}_{\text{lasso}} \ = \ \mathcal{S}_\lambda(X^\top y) \quad \text{(applied elementwise)},$$

exhibiting coefficient *selection* by exact zeros.

Geometric intuition. The L1 constraint $\|w\|_1 \leq t$ is a diamond (many corners). Linear level sets of the loss intersect the feasible set at corners, promoting *sparse* solutions.

When to use L1.

- You expect a *sparse* true signal (most features irrelevant).
- You want *embedded feature selection* and model interpretability.
- High-dimensional regime ($d \gg n$) where variable selection is crucial.

9.4 Comparing L1 vs. L2

Bias–variance and robustness. Both add bias to reduce variance. L1 yields sparse, potentially higher-bias but simpler models; L2 yields dense, smoother shrinkage and is especially effective under multicollinearity. (For heavy-tailed *errors*, consider pairing with robust losses; for heavy-tailed *features*, standardize and consider L1/L2 accordingly.)

Elastic Net (hybrid). When groups of correlated features exist, L1 alone may pick one arbitrarily; L2 alone keeps groups but no sparsity. A convex combination addresses both:

$$\min_w \ \frac{1}{n}\sum_{i=1}^{n} \ell\big(f_w(x_i), y_i\big) \ + \ \lambda\Big(\alpha\|w\|_1 + \tfrac{1}{2}(1-\alpha)\|w\|_2^2\Big), \quad \alpha \in [0,1].$$

10 Representation Learning

Deep learning seeks to encode functions, which we accomplish by converting raw data into more efficient representations
For example, the game of Go has an exponentially large number of states, so it is impossible to search over the game tree. However, an image $256^{3\times500\times500}$ We don't identify images in the raw space, we identify images in some representation space.

10.1 What is a representation

A representation of a data domain $\mathcal{X}$ is a function $f : \mathcal{X} \to \mathbb{R}^d$ that assigns a feature vector to each input in that domain. A representation of a datapoint $\mathbf{x}$ is a vector $\mathbf{x} \in \mathbb{R}^d$ with $f(\mathbf{x}) = \mathbf{z}$. The neural net learns this representation, which we hope will be useful/correct for whatever task we are attempting. Concretely, the output vector of a neural net doesn't contain the exact same information as the original input, but we hope that the neural net learned to represent $\mathbf{x}$

10.2 Visualizing representations

The pictures below show that activation of the i^{th} layer which

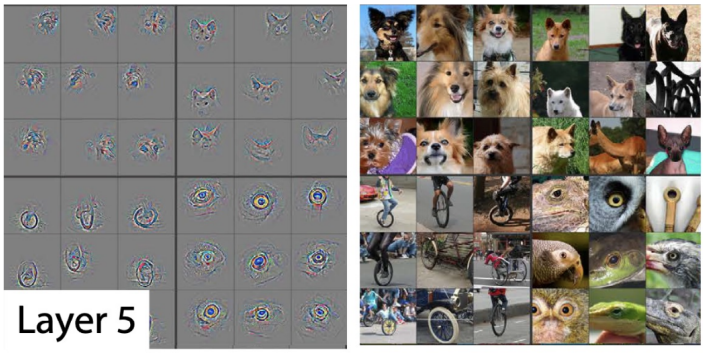


Figure 38

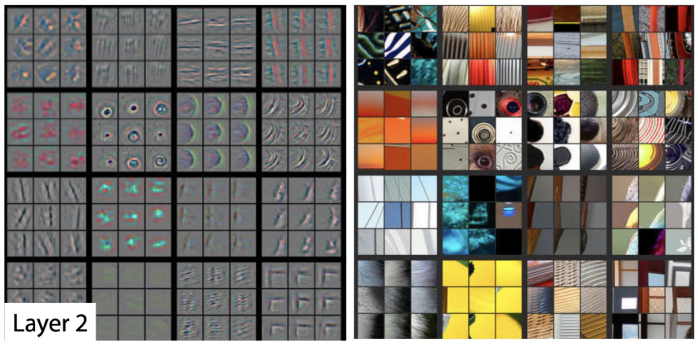


Figure 39

11 Foundation models and transfer learning

11.1 Foundation models

Not so long ago, if you wanted to do task T you would need to collect lots of labeled data for T (e.g., supervised learning), design a suitable architecture, initialize it randomly, then train and validate. In other words, we were designing a model for a specific task.

But now, we use large *foundation models* that are work across a variety of tasks. Think of foundation models as general purpose tools that work across across many tasks. First, foundation models almost always take the form of *language models,* why?
Language provides a natural domain for the study of artificial intelligence, as the vast majority of reasoning tasks can be efficiently expressed and evaluated in language, and the world's text provides a wealth of data for unsupervised learning via generative modeling.
More specifically, what might we want from foundation models?

- Language: Understand our requests even when they are.
- Knowledge: Know enough about the world to be useful.
- Retrieval and Tool Use: Search the web for information if it's not sure.
- In-Context Learning: Learn new skills quickly from examples.
- Alignment: Actually follow our instructions.
- Reasoning: Think longer, consider alternatives, and correct itself if needed.

11.2 How to create foundation models?

Here are two basic ideas:

1. Supervised learning: Maybe we should just collect a bunch of supervised question–answer pairs then train our supervised model. This is perhaps the most natural and direct approach: if we want a language model, the simply train a language model on past language.

2. Reinforcement learning: put a model out into the world and give it rewards for the right behavior (e.g. following instructions). The challenge is that for a very long time it will just be doing random things if its initialized randomly, it also has no idea what different instructions mean, and even when it gets some occasional rewards, can it interpret and extrapolate them correctly?

3. Self-supervised learning: use a supervised training objective and a supervised learning loop.

   What if we could scale the tokens we learn from by millions or billions of times? Language Modeling (next token prediction) on... "the whole Web"

11.3 Three main parts of understanding foundation models

1. Architecture: Modeling language via Decoder-only Transformers.

2. Pre-training: Giving LLMs broad knowledge of language, the world, and excellent foundations for reasoning and fine-tuning.

3. Post-training: Teaching LLMs to be instruction-following assistants and to be effective at math, coding, using tools

Note: details and engineering at scale really matter, and are much more complex than we're covering!

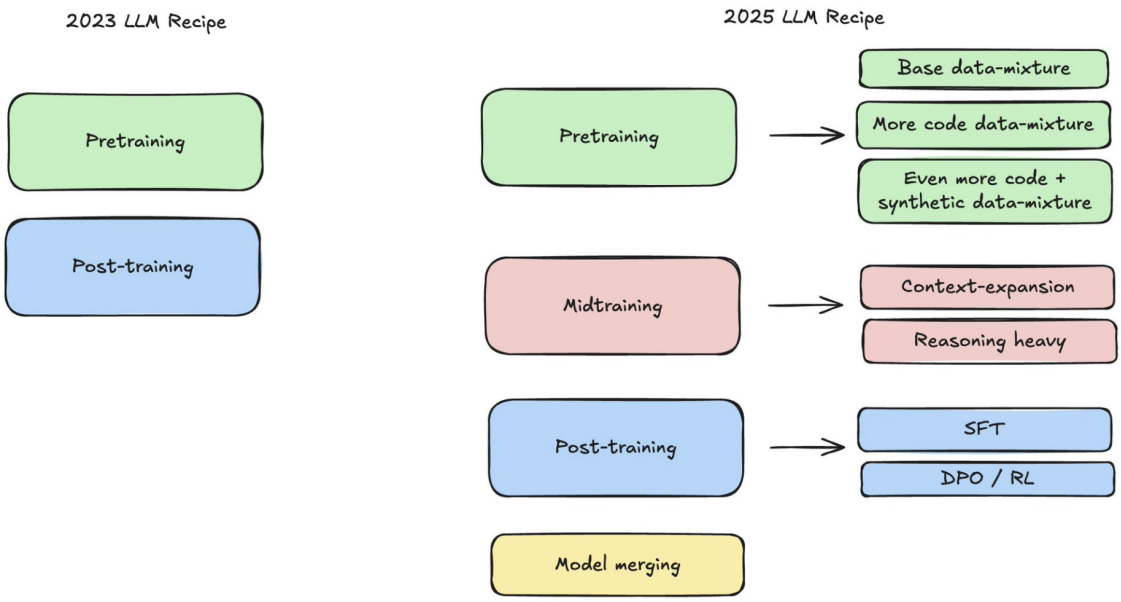


Figure 40: Caption

11.4 Architecture

11.5 Tokenization

The first choice we face is tokenization for language. We usually use subword tokens.

$$\text{can't dislike this} \rightarrow [\text{"can"}, \text{"'t"}, \text{" dis"}, \text{"like"}, \text{" this"}]$$

Characters are too small of an inductive bias for the scale we are working at. meanwhile the problem with whole-word tokenization is the size of the dictionary, issues with misspelling, and rare words that don't appear in training set.

11.5.1 Transformer Encoders

BERT, trained via masked language modeling ("fill in the blanks") because of self attention, the representation of a masked token depends on downstream tokens. Therefore fast autoregressive generation is not possible.

11.5.2 Decoder-Only Transformers ("GPT")

Like an Encoder, but with a causal mask: During attention, token t can only attend to tokens $\geq t$. At each position, produces a probability distribution over the next token. Trained as a per-step classifier to predict token x_{t+1} for all$t$. In parallel, we are teaching the model how to think at *every step*, which we could not do with the encoder model. In each attention layer, Key/Value representations of every token t are not affected by "future" tokens, which enables fast autoregressive decoding.

11.6 Pretraining (Data)

Data can teach our model several level of understanding

1. factual knowledge
2. syntax
3. sentiment analysis
4. math word problems
5. prompting
6. code generation

It's general consensus that we can't go much further beyond tokenization and attention to incorporate inductive bias into the architecture. Instead, we encode inductive bias into the data.
Then you babysit the training runs:

- You monitor the loss over time.
- You watch out for spikes in loss or gradient norm. Stability is hard!

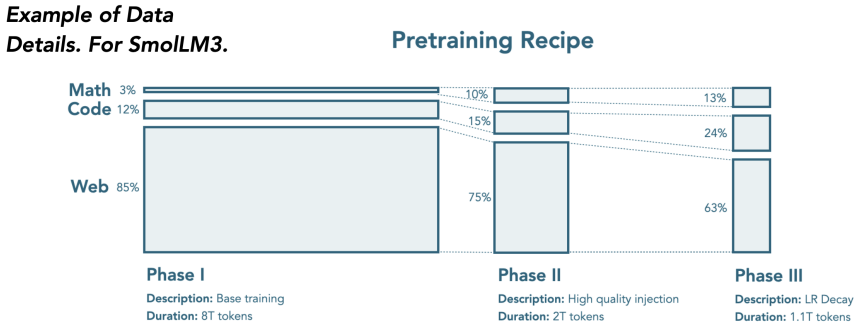


Figure 41: Data composition as a function of pre-training stage

11.7 Scaling laws

Hyperparameters do not always transfer to larger scale: Best choice at small scale may not be the same at larger scale

11.8 hackers guide

Good ideas are often simple But they may only shine when aggressively scaled, and doing so well takes a lot of engineering and attention to complex detail. (No free lunch; gotta adapt to this world's biases. But there are often clever methods that scale better!

12 Generative models

VAEs, GANs, diffusion models, flow matching

12.1 Generative vs discriminative model

The first seemingly important distinction is between the following:

- *Generative models* map `labels` $\to$ `samples`.
- Meanwhile, *discriminative models* map `samples` $\to$ `labels`

Let y be an abstract object such as an image, and let x be a simple object such as a label. Then via Bayes law generative modeling is

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}$$

while discriminative modeling is

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}.$$

But the notion of abstract vs. concrete is really just a mental construct, so from a mathematical perspective these are really fully equivalent! There is no mathematical obstacle to formulating any real-world problem from a generative perspective.

12.2 Generative models as probabilistic models

It is helpful to think of generative models as *probabilistic models* that map distributions to distributions (as in the case with diffusion models). However, keep in mind that "probability itself is a modeling assumption" since there is no "true" data distribution, and even if it did we only observe a finite set of samples from it. However, the mechanism we'll design operate on the space of probability distributions.

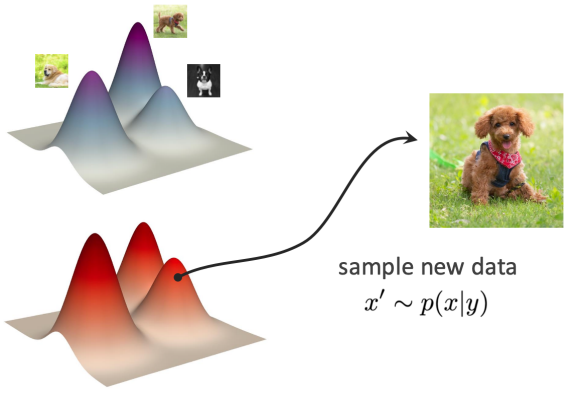


Figure 42: Generative modeling is probabilistic modeling

One methodology in generative modeling involves sampling $z \sim \pi$ from a simple distribution, passing it through our generative model $g_\theta(\cdot)$ to obtain a learned *representation*, $p_\theta = g_\theta[\pi]$

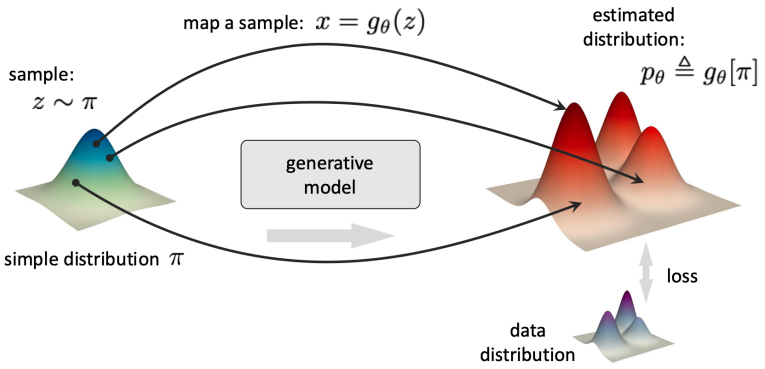


Figure 43: Generative modeling is representation learning

However, note that this representation is structure/often has inductive bias, for example, consider the probabilistic framework used in autoregressive models vs diffusion models

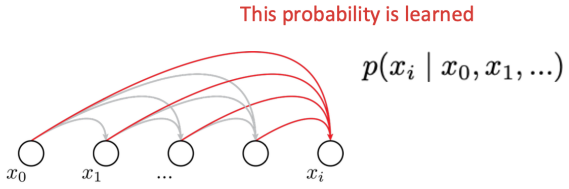


Figure 44: Autoregressive models use a dependency structure that is designed, not learned

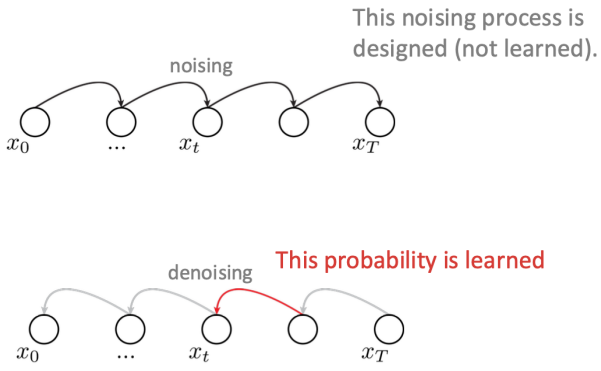


Figure 45: Diffusion models use a fixed noising process and a learned reverse process

12.3 Variational Autoencoders (VAE)

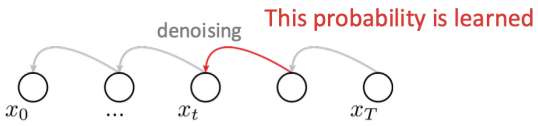


Figure 46: Diffusion models use a fixed noising process and a learned reverse process

12.4 Variational Autoencoders (VAE)

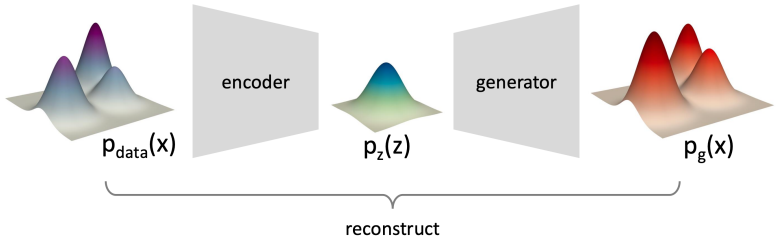


Figure 47

12.5 Normalizing Flows

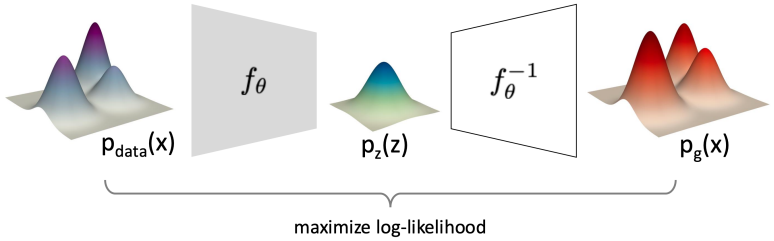


Figure 48

12.6 GANs

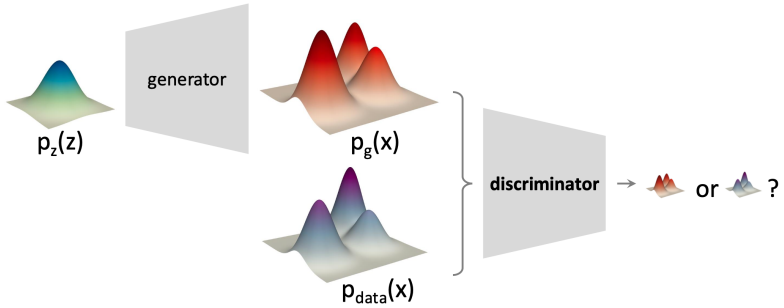


Figure 49

12.7 Autoregressive models

$$p(x_1, x_2, ..., x_n) = p(x_1)p(x_2 \mid x_1)...p(x_n \mid x_1, x_2, ..., x_{n-1})$$

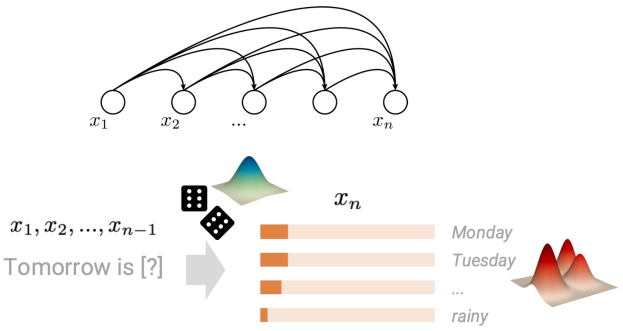


Figure 50

12.8 Diffusion models

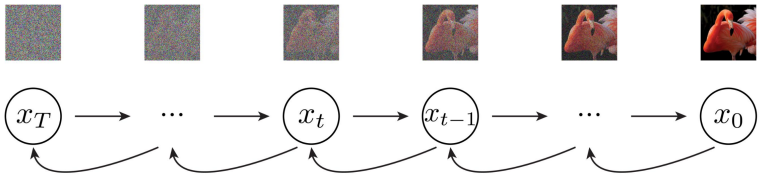


Figure 51

12.9 Latent variable models

Latent variable models assume that the features we obverse in the world are the product of
The generator turns the latent variable into an observation (such as a picture). We learn the generator using a neural network. This generator implicitly learns the conditional distribution $p(x|z)$ where z is the latent variable and x is a distribution. We parameterize this generator as $p_\theta(x|z)$ then the output is a distribution $p_\theta(x)$.
How do we measure the loss functions between $p_\theta(x)$ and $p_{\text{data}}(x)$. A re topic in generative modeling is to make loss functions