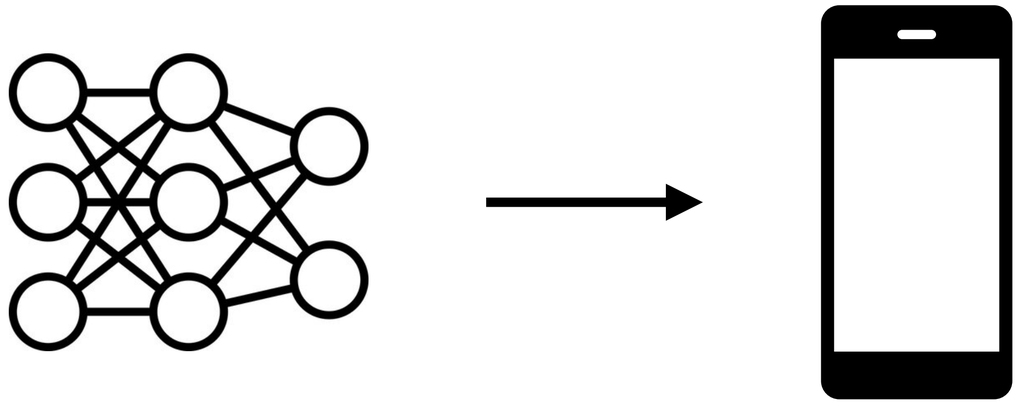
Reducing the size of neural networks

Plan

- Pruning
- Quantization
- Distillation

Motivation

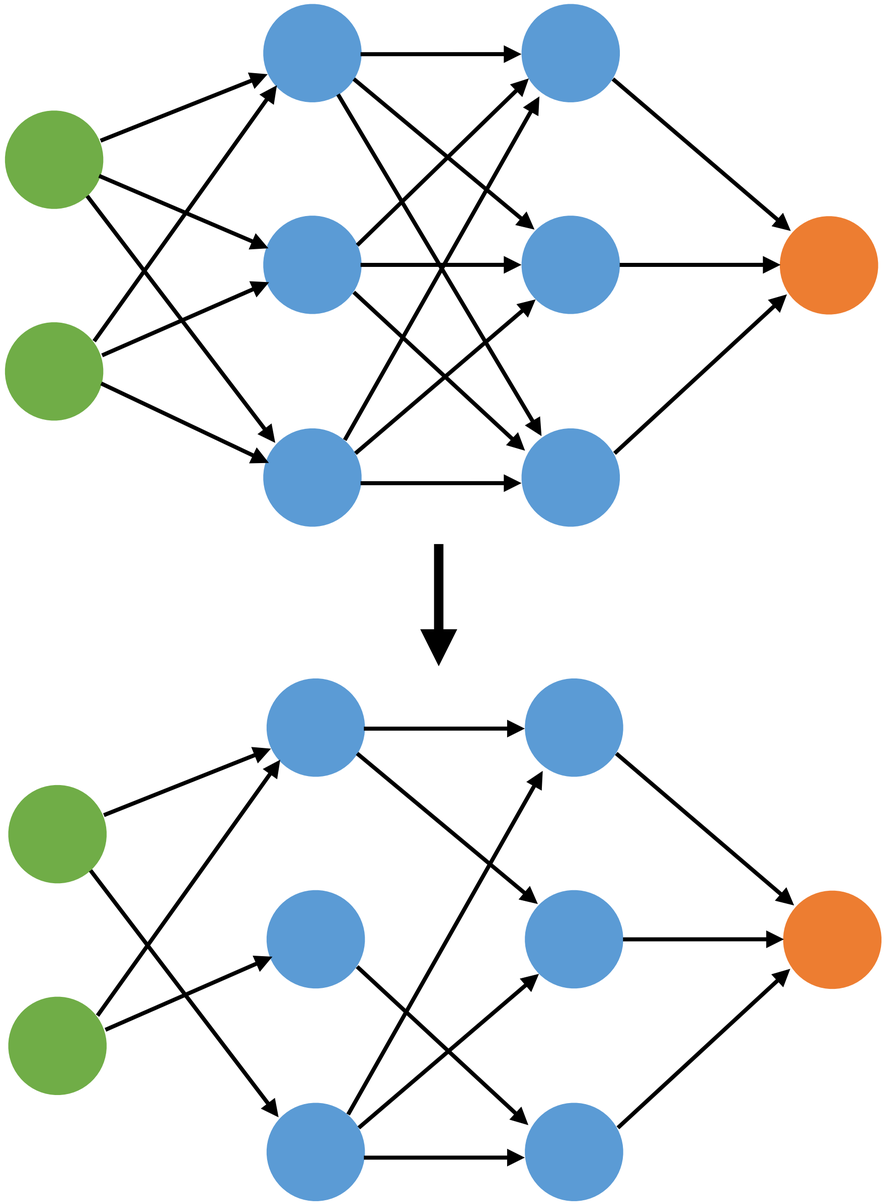
- Neural networks are getting bigger every year
- At the same time, it is often desirable to use models locally on portable devices
- To do this, we need to learn how to reduce their size without losing quality.



Pruning

Neural networks often consist of millions of parameters.
It turns out that not all parameters are needed.

Pruning - process of removal of a part of model weights without significant loss in quality.

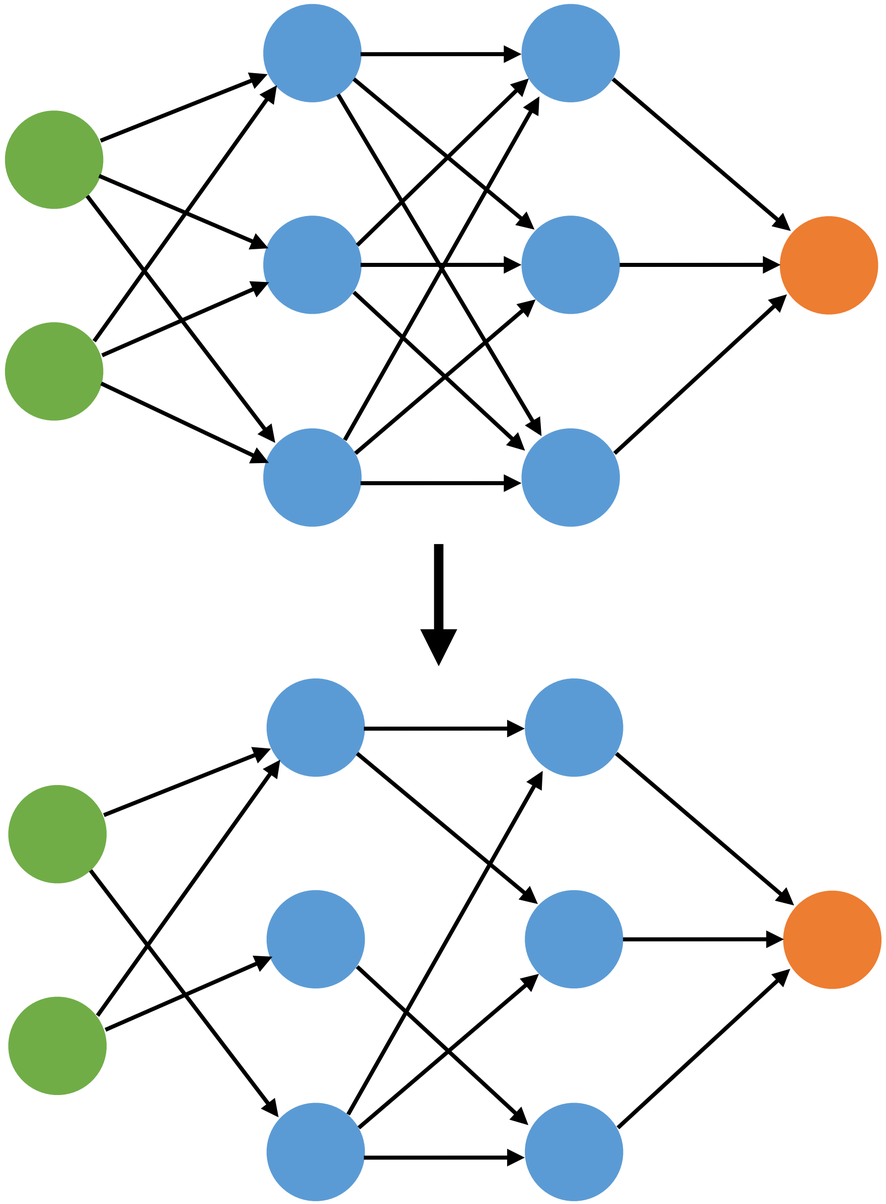


Pruning

Neural networks often consist of millions of parameters.
It turns out that not all parameters are needed.

Pruning - process of removal of a part of model weights without significant loss in quality.

Dropout is a special case of pruning.



Criteria for removal of weights

- **Magnitude of weights:** remove weights with the smallest absolute value (the most popular method)

Criteria for removal of weights

- **Magnitude of weights:** remove weights with the smallest absolute value (the most popular method)

- **Dispersion of neuron outputs:** if the output does not change for different inputs, the neuron can be replaced by a constant and added to the shift

Criteria for removal of weights

- **Magnitude of weights:** remove weights with the smallest absolute value (the most popular method)

- **Dispersion of neuron outputs:** if the output does not change for different inputs, the neuron can be replaced by a constant and added to the shift

- **Neuron activation magnitude:** remove a neuron if it never activates
  No assumption of equality of input values is required as in magnitude of weights.

Criteria for removal of weights

- **Magnitude of weights:** remove weights with the smallest absolute value (the most popular method)

- **Dispersion of neuron outputs:** if the output does not change for different inputs, the neuron can be replaced by a constant and added to the shift

- **Neuron activation magnitude:** remove a neuron if it never activates
  No assumption of equality of input values is required as in magnitude of weights.

- **Correlation between weights:** if the weights of one neuron are highly correlated, we can keep only one neuron

Criteria for removal of weights

- **Magnitude of weights:** remove weights with the smallest absolute value (the most popular method)

- **Dispersion of neuron outputs:** if the output does not change for different inputs, the neuron can be replaced by a constant and added to the shift

- **Neuron activation magnitude:** remove a neuron if it never activates
  No assumption of equality of input values is required as in magnitude of weights.

- **Correlation between weights:** if the weights of one neuron are highly correlated, we can keep only one neuron

- L_0**-regularization:** motivating the model to zero out weights.

$$\|w\|_0 = \sum_i [w_i \neq 0]$$

Training of pruned models

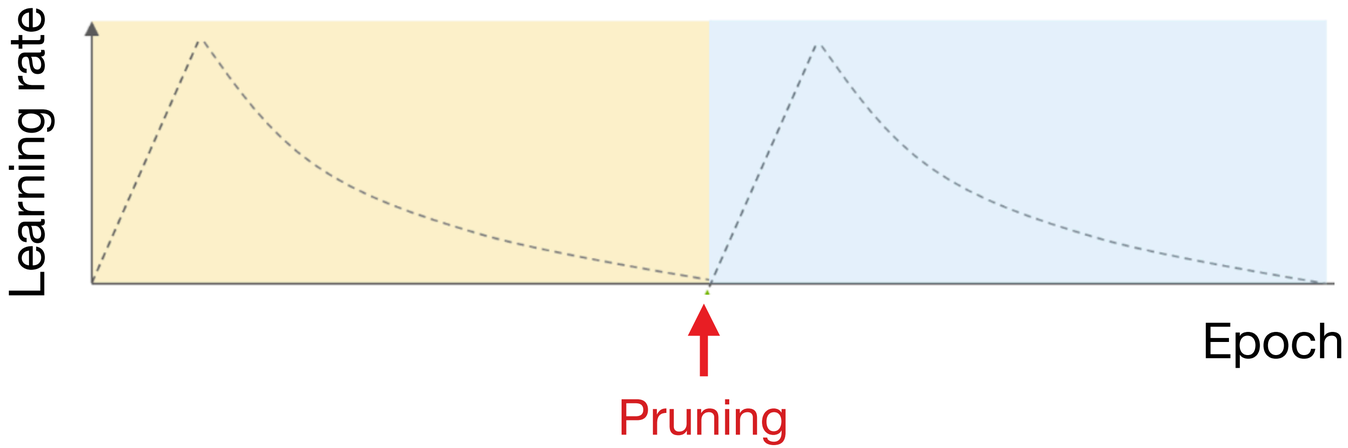
After removing weights, the model must be fine-tuned to reconfigure the remaining weights.

It is also possible to remove weights at different stages:

- After training
- During training

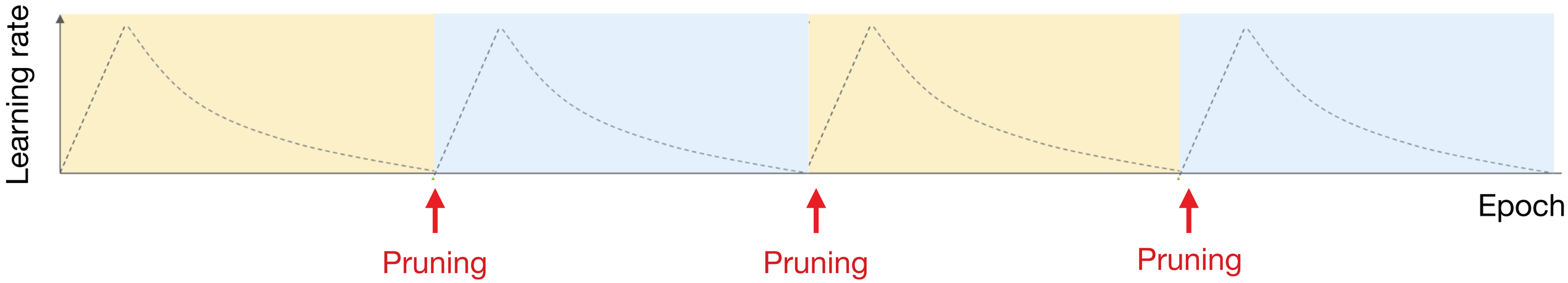
Pruning after training

- Train the model until convergence
- Prune the weights
- Fine-tune



Pruning while training

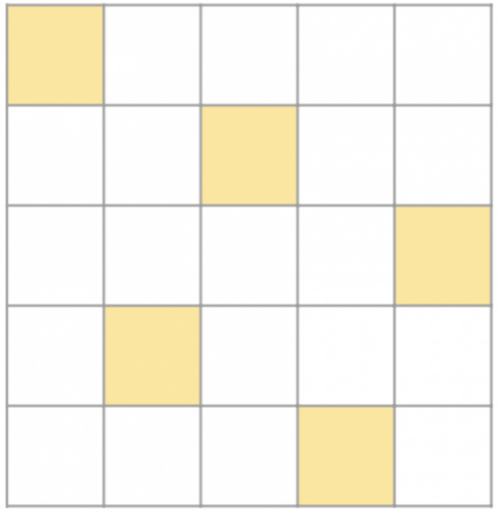
- Prune the proportion of weights every few iterations of training
- More customisable parameters
- Usually works better than a single pruning



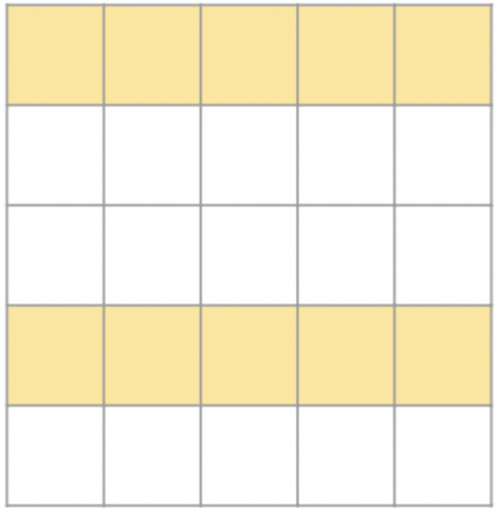
Structured pruning

- Matrix multiplication operations are paralleled at vector level
- When removing individual weights, the network becomes sparse, but there is no gain in speed

- For a meaningful speedup it is necessary to remove whole structures: matrix vectors, layers of the network



Unstructured



Structured

Disadvantages of pruning

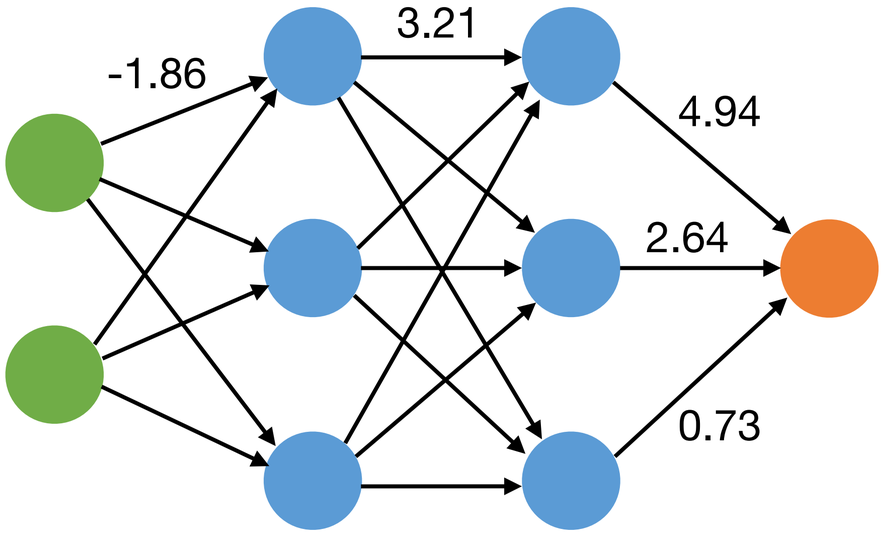
- Pruning can reduce the number of weights by 30-50%, but it doesn't give much speed gain due to the implementation of model layers
- Pruning works well for older unoptimised architectures and is not as useful for modern models
- It is impossible to say in advance that a particular technique will work

Quantisation

- By default, neural networks work with numbers in float32 format
- These are real numbers that occupy 32 bits

- The more bits used to represent a number, the more we need:
  - Memory for storage
  - Time for addition and multiplication operations

Quantisation - reducing the precision (bitness) of numbers.
It is most often used after training.

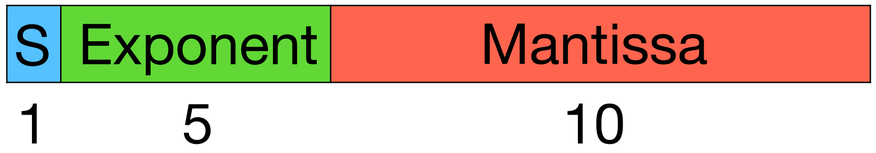


How do computers represent numbers?

float32 (single-precision)

sign

float16 (half-precision)



bfloat16 (brain float)

int8

$$x = (-1)^S \cdot (1,M) \cdot 2^{E-127}$$

To represent large numbers

Quantization

Most often the precision is downgraded from float32 to

- **float16**
- **int8** (float8 is not used because it is almost nowhere defined)

Quantisation in float16 is done by truncating the mantissa to the required size

Quantisation in int8 is much trickier

float32 -> int8

Les x be a float32 number and x_q is an int8 quantised x

$$x_q = round\left(\frac{x}{s} + z\right) \quad x = s \cdot (x_q - z)$$

- s – scale; positive number in float32
- z – zero point; it is stored in int8 and corresponds to 0 in the original float32 number

Such quantisation is called **asymmetric** and is used when $x_q \in [0,255]$

How to find s and z?

If we know the range $[a, b]$ of x values and $x_q \in [a_q, b_q]$, then

$$a = s(a_q - z)$$
$$b = s(b_q - z)$$

Then

$$s = \frac{b - a}{b_q - a_q} \qquad z = \frac{ba_q - ab_q}{b - a}$$

How to find s and z?

If we know the range $[a, b]$ of x values and $x_q \in [a_q, b_q]$, then

$$a = s(a_q - z)$$
$$b = s(b_q - z)$$

Then

$$s = \frac{b - a}{b_q - a_q} \qquad z = \frac{ba_q - ab_q}{b - a}$$

It is important to make sure that 0 is mapped back without errors. That is, that z is an integer. Therefore

$$z = round\left(\frac{ba_q - ab_q}{b - a}\right)$$

How to find s and z?

If we know the range $[a, b]$ of x values and $x_q \in [a_q, b_q]$, then

$$a = s(a_q - z)$$
$$b = s(b_q - z)$$

Then

$$s = \frac{b - a}{b_q - a_q} \qquad z = \frac{ba_q - ab_q}{b - a}$$

It is important to make sure that 0 is mapped back without errors. That is, that z is an integer. Therefore

$$z = round\left(\frac{ba_q - ab_q}{b - a}\right)$$

If we get $x \notin [a, b]$, we round it to the nearest bound.

$$x_q = clip(round\left(\frac{x}{s} + z\right), a_q, b_q)$$

Symmetric quantization

Used when the range is symmetric, $x \in [-a, a]$.

Quantization maps x to $x_q \in [-127, 127]$.

So we lose one value of the quantised variable, but get rid of the zero point

$$x_q = round(s \cdot x)$$

How to find a and b?

We need to quantize the model weights and the input values (activations).

- For weights it is easy - they do not change

- For activations there are three ways:
  - Dynamic quantisation
  - Static quantisation
  - Training time quantisation

How to find a and b?

We need to quantize the model weights and the input values (activations).

- For weights it is easy - they do not change
- For activations there are three ways:
  - Dynamic quantisation - we calculate bounds on the fly for each activation
  - Static quantisation - we estimate bounds in advance by thresholding n examples.
  - Training time quantisation - use quantisation during training and find bounds from training examples

How to find a and b?

We need to quantize the model weights and the input values (activations).

- For weights it is easy - they do not change
- For activations there are three ways:
  - Dynamic quantisation - we calculate bounds on the fly for each activation
    (maximum accuracy, slow speed)
  - Static quantisation - we estimate bounds in advance by thresholding n examples.
    (lower accuracy, fast speed)
  - Training time quantisation - use quantisation during training and find bounds from training examples
    (good accuracy, maximum speed, model is tuned for quantisation)

Calibrating boundary values

Min-max: works well for model weights

a = minimum found value

b = maximum found value

Moving average min-max: works well for activations

a = moving average minimum value

b = moving average maximum value

Calibrating boundary values

Min-max: works well for model weights

$$a = \text{minimum found value}$$
$$b = \text{maximum found value}$$

Moving average min-max: works well for activations

$$a = \text{moving average minimum value}$$
$$b = \text{moving average maximum value}$$

Histogram-based - form histograms from all activation values, find bounds in one of 3 ways:

- **Entropy**: minimise KL-divergence between distributions of quantised and original values
- **MSE**: minimise the square of the difference between the histogram bins
- **Percentile**: select boundaries by percentiles of activation values

Operations with quantised numbers

Addition:

$$y = x_1 + x_2 = s_1(x_{q1} - z_1) + s_2(x_{q2} - z_2) = s_y(y_q - z_y)$$

$$y_q = z_y + \frac{1}{s_y}\left(s_1(x_{q1} - z_1) + s_2(x_{q2} - z_2)\right)$$

Multiplication:

$$y = x_1 \cdot x_2 = s_1(x_{q1} - z_1) \cdot s_2(x_{q2} - z_2) = s_y(y_q - z_y)$$

$$y_q = z_y + \frac{s_1 s_2}{s_y}(x_{q1} - z_1)(x_{q2} - z_2)$$

Per-tensor and per-channel quantization

Since quantization parameters take up space, we want to minimise their number

- **Per-tensor**: store one pair (s; z) for each tensor

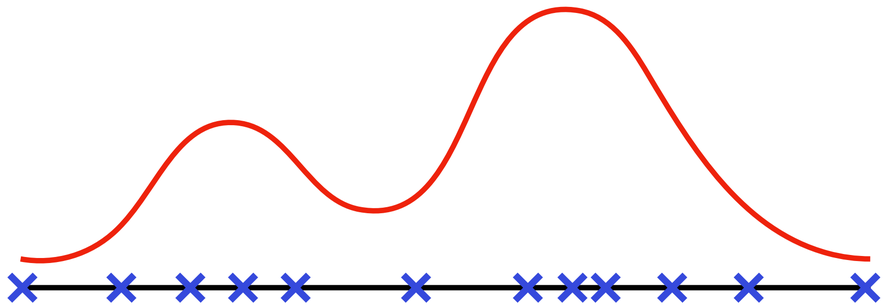
Less memory

- **Per-channel**: store one pair for each vector coordinate

More accurate

Nonlinear quantization

Usually the weights and activation values are not uniformly distributed
It is useful to increase accuracy at intervals where the density is greater



There are several ways to do this, but they are used less frequently because of the complexity.

Practical tips

- Not all operations need to be quantised.
  Matrix multiplications are most important, since they are the most expensive.
  Multiplication of two matrices $A \in \mathbb{R}^{[n \times m]}$ and $B \in \mathbb{R}^{[m \times p]}$ requires about $2nmp$ operations.

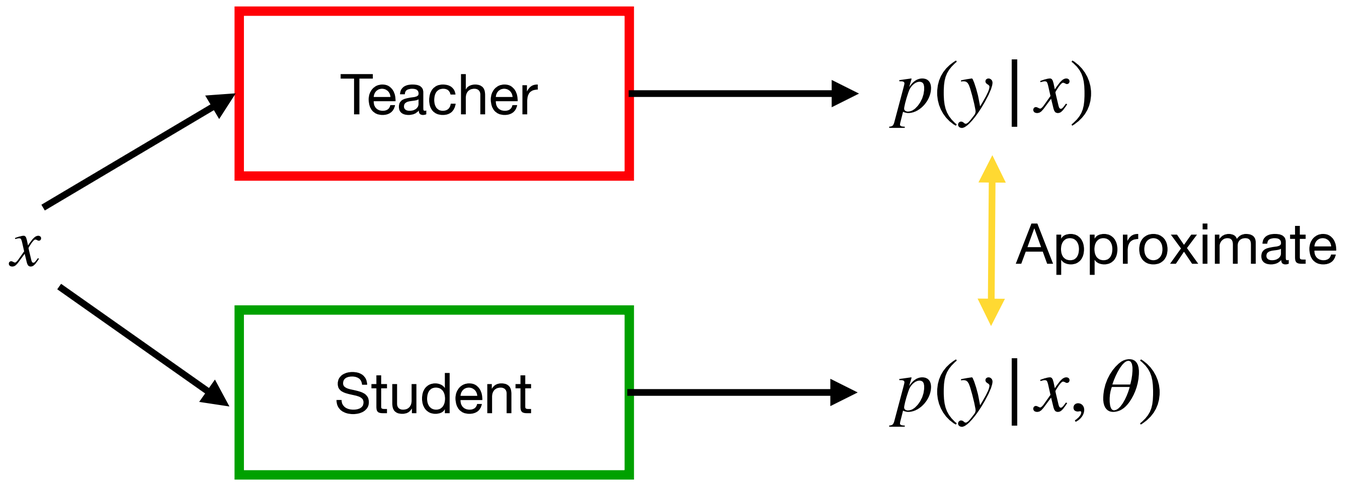
- For Layer Norm it is better not to apply quantisation, as it may not work stably

- It is best to start with dynamic quantisation.
  If the speed is not enough, then move to static quantisation.
  At the end, if accuracy is not enough, then to training with quantisation.

Distillation

- Knowledge distillation is the process of transferring knowledge from a large trained model (teacher) to a small model (student) with minimal loss of quality

- Distillation is implemented by bringing the student's outputs closer to the teacher's outputs



How to train?

To train a student, one of the two losses is minimised:

- **KL-divergence**

$$KL(p^s \| p^t) = \sum_{k=1}^{K} p_k^s \log \frac{p_k^s}{p_k^t}$$

- **Cross-Entropy**

$$CE(p^t, p^s) = -\sum_{k=1}^{K} p_k^t \log p_k^s$$

How to train?

To train a student, one of the two losses is minimised:

- **KL-divergence**

$$KL(p^s \| p^t) = \sum_{k=1}^{K} p_k^s \log \frac{p_k^s}{p_k^t}$$

- **Cross-Entropy**

$$CE(p^t, p^s) = -\sum_{k=1}^{K} p_k^t \log p_k^s \propto$$

$$\propto \sum_{k=1}^{K} p_k^t \log p_k^t - \sum_{k=1}^{K} p_k^t \log p_k^s = KL(p^t \| p^s)$$

Probability smoothing

- Neural networks are very often overconfident in their predictions
- The p^t distribution turns out to be almost degenerate
- Approximating the outputs to such a distribution is no better than a default supervised learning

Distillation make use of **smoothed** probabilities.

$$p_\tau^t = softmax(l^t, T = \tau) \qquad \tau > 1$$
$$p_\tau^s = softmax(l^s, T = \tau)$$

Such labels are called **soft labels**

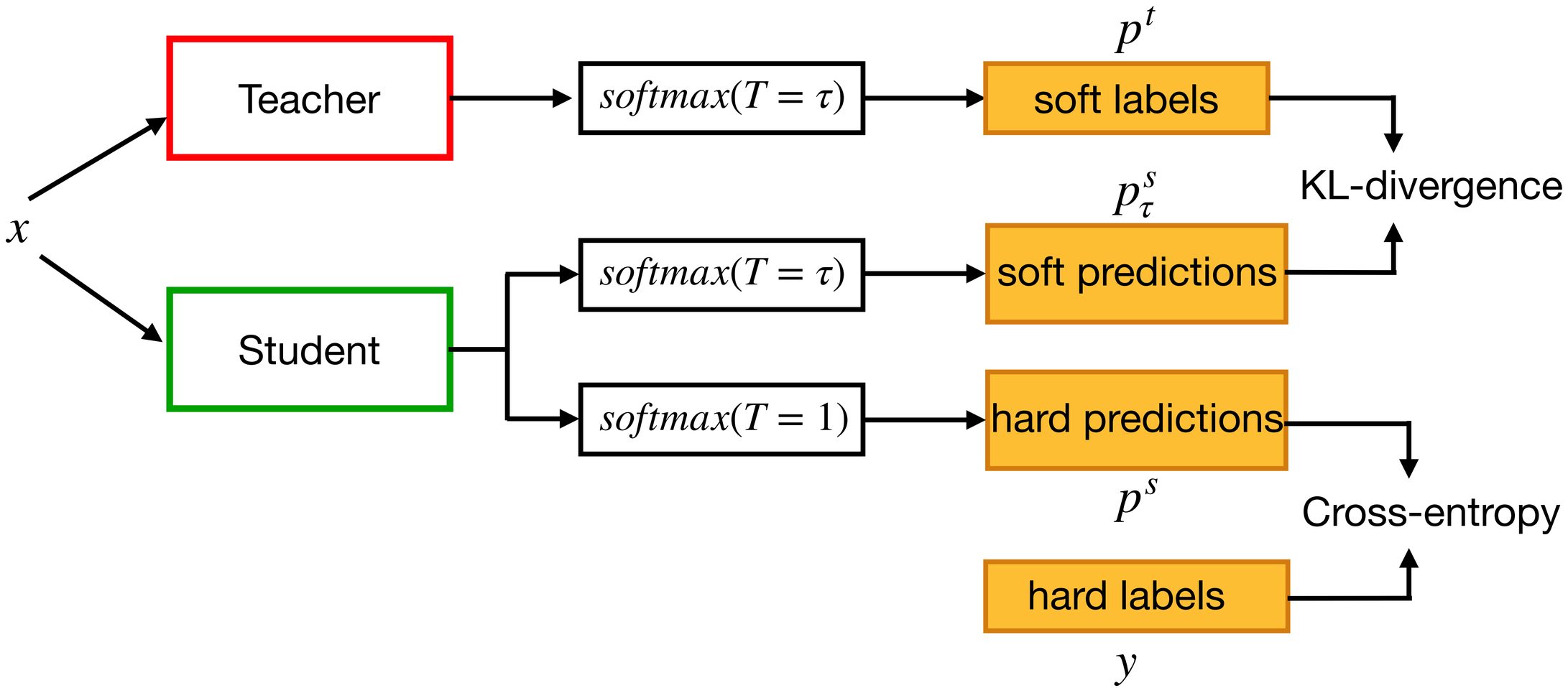
Hard labels

Since we have access to the correct answers, we add a standard classification loss as well.

$$CE(y, p^s) = -\sum_{k=1}^{K} [y = k] \log p_k^s$$

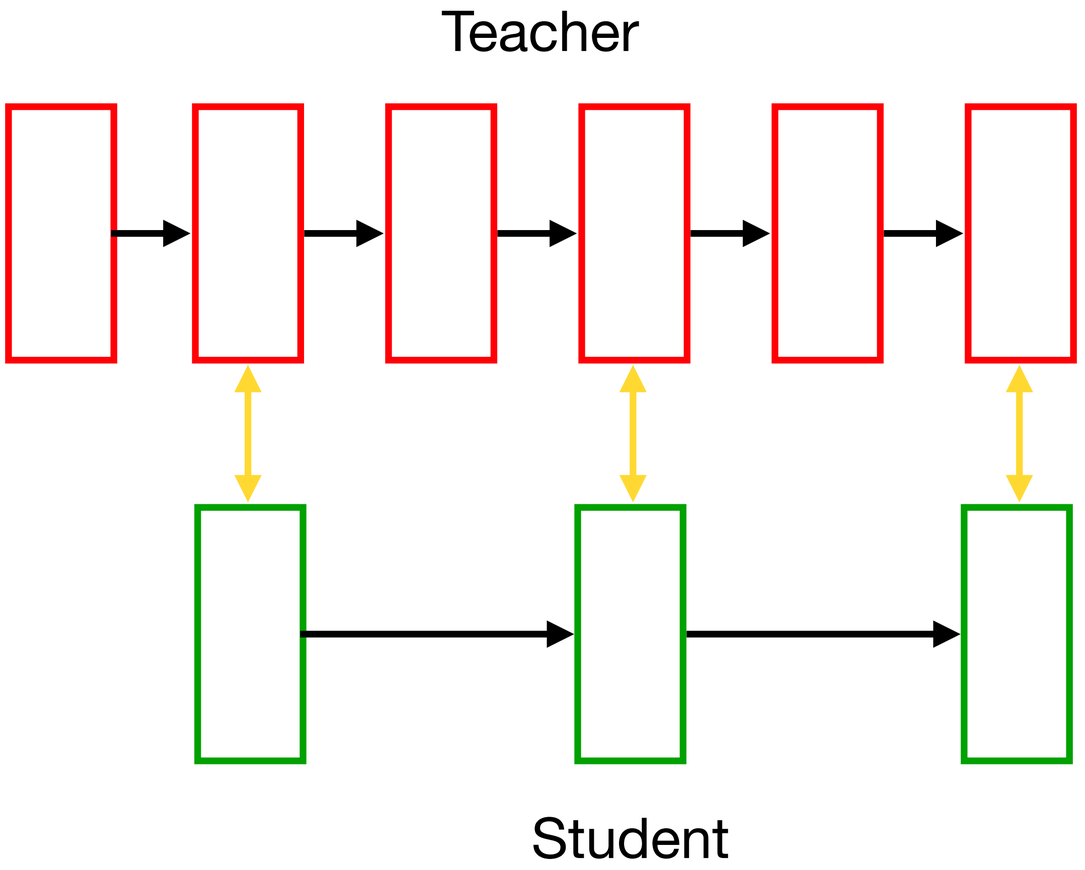
Correct answers y are called **hard** labels.

Training algorithm



Intermediate layers

We only make use of the model outputs, but there are layer outputs too!
Let's bring the student's layer outputs closer to the teacher's outputs



Attention layers

Attention matrices have the same size: $[H, seq\_len, seq\_len]$

Let's introduce a loss to minimise the distance between them

$$L_{attn} = \frac{1}{H} \sum_{h=1}^{H} \|A_h^s - A_h^t\|_2^2$$

Intermediate layer outputs

The size of the layer output may vary from model to model, so let's add a projection $W \in \mathbb{R}^{[d_s \times d_t]}$.

$$L_{hidn} = \|H^s W - H^t\|_2^2$$

The matrix W is trained along with the student's parameters.

Distillation with a pre-training task

Almost all transformer models are trained in two stages: pre-training and fine-tuning.

In this case, it is better to distil the student first on the pre-training task and then on the downstream task.