From linear neural networks to convolutional neural networks to transformers

- Antonio Greco

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Material

- **Book**
  - Dive into Deep Learning, An interactive deep learning book with code, math and discussions, implemented with PyTorch, NumPy/MXNet, JAX and Tensorflow: https://d2l.ai/

- **Online video tutorials**
  - Neural networks explained: https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1

Recap on linear neural networks

- Linear neural networks for regression
  - Dive into Deep Learning, Chap. 3
    - Linear Regression
    - Loss function (squared error)
    - Minibatch stochastic gradient descent (batch size, learning rate)
    - Linear Regression as a single layer fully connected neural network
    - Training error (bias) and generalization error (variance), underfitting and overfitting
    - Regularization

Regression

- Regression problems pop up whenever we want to predict a numerical value.
- As a running example, suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years).
- To develop a model for predicting house prices, we need to get our hands on data, including the sales price, area, and age for each home. In the terminology of machine learning, the dataset is called a training dataset or training set, and each row (containing the data corresponding to one sale) is called an example (or data point, instance, sample).
- The thing we are trying to predict (price) is called a label (or target).
- The variables (age and area) upon which the predictions are based are called features (or covariates).

Linear regression

- Linear regression is both the simplest and most popular among the standard tools for tackling regression problems.
- We assume that the relationship between features x and target y is approximately linear, i.e., that the conditional mean can be expressed as a weighted sum of the features x.

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b.$$

- Here w_{area} and w_{age} are called weights, and b is called a bias (or offset or intercept). The weights determine the influence of each feature on our prediction. The bias determines the value of the estimate when all features are zero.
- Given a dataset, our goal is to choose the weights w and the bias b that, on average, make our model's predictions fit the true prices observed in the data as closely as possible.

Linear regression

- When our inputs consist of d features, we can assign each an index (between 1 and d) and express our prediction $\hat{y}$ (in general the "hat" symbol denotes an estimate) as

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b.$$

- Collecting all features into a vector x and all weights into a vector w, we can express our model compactly via the dot product between w and x:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

- We will often find it convenient to refer to features of our entire dataset of n examples via the design matrix X. Here, X contains one row for every example and one column for every feature.

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b,$$

Linear regression

- Given features of a training dataset X and corresponding (known) labels y, the goal of linear regression is to find the weight vector w and the bias term b such that, given features of a new data example sampled from the same distribution as X, the new example's label will (in expectation) be predicted with the smallest error.

- Before we can go about searching for the best parameters (or model parameters) w and b, we will need two more things:
  1. a measure of the quality of some given model (loss function)
  2. a procedure for updating the model to improve its quality (e.g. minibatch stochastic gradient descent)

Loss function

- Loss functions quantify the distance between the real and predicted values of the target. The loss will usually be a nonnegative number, where smaller values are better and perfect predictions incur a loss of 0.
- For regression problems, the most common loss function is the squared error:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

- When training the model, we seek parameters (w*, b*) that minimize the total loss across all training examples:

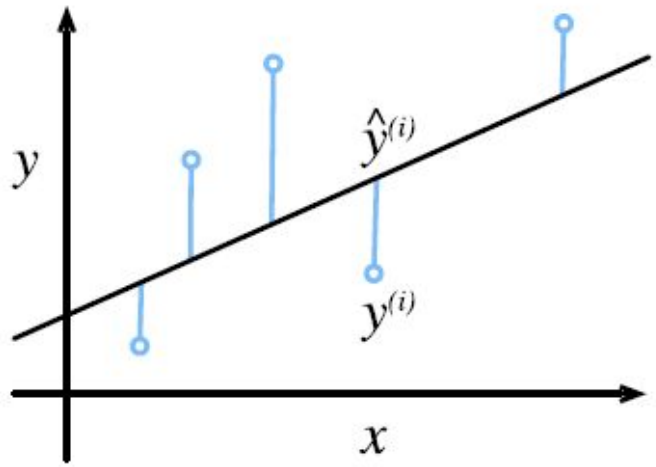
$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b)$$

Loss function

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b)$$



Gradient descent

- The key technique for optimizing nearly every model consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function.
- This algorithm is called **gradient descent**.
- The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset.
- In practice, this can be extremely slow: we must pass over the entire dataset before making a single update, even if the update steps might be very powerful.
- Even worse, if there is a lot of redundancy in the training data, the benefit of a full update is limited.

Stochastic gradient descent

- The other extreme is to consider only a single example at a time and to take update steps based on one observation at a time. The resulting algorithm, **stochastic gradient descent** (SGD), can be an effective strategy, even for large datasets. Unfortunately, SGD has drawbacks, both computational and statistical.
- One problem arises from the fact that processors are a lot faster multiplying and adding numbers than they are at moving data from main memory to processor cache. It is up to an order of magnitude more efficient to perform a matrix–vector multiplication than a corresponding number of vector–vector operations.
- A second problem is that some of the layers, such as batch normalization, only work well when we have access to more than one observation at a time.

Minibatch stochastic gradient descent

- The solution to both problems is to pick an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a minibatch of observations. This leads us to **minibatch stochastic gradient descent (SGD)**.
- The specific choice of the minibatch size depends on many factors, such as the amount of memory, the number of accelerators, the choice of layers, and the total dataset size.
- In each iteration t, we first randomly sample a minibatch B_t consisting of a fixed number |B| of training examples. We then compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters. Finally, we multiply the gradient by a predetermined small positive value η, called the learning rate, and subtract the resulting term from the current parameter values.

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$$

Minibatch stochastic gradient descent

- In summary, minibatch SGD proceeds as follows:
  1. initialize the values of the model parameters, typically at random;
  2. iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.
- For quadratic losses, this has a closed-form expansion:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right).$$

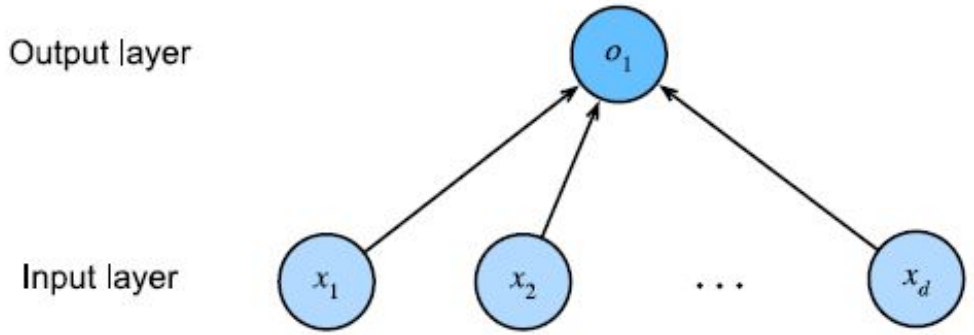
- Minibatch size and learning rate are user-defined. Such tunable parameters that are not updated in the training loop are called **hyperparameters**.

Training, convergence, prediction

- After training for some predetermined number of iterations (or until some other stopping criterion is met), we record the estimated model parameters, denoted $\hat{w}$ and $\hat{b}$. Note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss, nor even deterministic.
- Given the model $\hat{w}^T + \hat{b}$, we can now make predictions for a new example, e.g., predicting the sales price of a previously unseen house given its area $x1$ and age $x2$.
- Deep learning practitioners have taken to calling the prediction phase inference, but this is a bit of a misnomer since inference refers broadly to any conclusion reached on the basis of evidence, including both the values of the parameters and the likely label for an unseen instance.

Linear regression as a neural network

- While linear models are not sufficiently rich to express the many complicated networks available today, we can think of linear regression as a single-layer fully connected neural network.



- The inputs are $x1, \ldots, xd$. We refer to d as the number of inputs or the feature dimensionality in the input layer. The output of the network is $o1$. Because we are just trying to predict a single numerical value, we have only one output neuron.

Linear regression example

- We are now ready to work through a fully functioning implementation of linear regression.
- We will implement the entire method from scratch, including:
  - the model;
  - the loss function;
  - a minibatch stochastic gradient descent optimizer;
  - the training function that stitches all of these pieces together.
- We will use a synthetic data generator from Section 3.3 of Dive Into Deep Learning Book and apply our model on the resulting dataset. Each label of the data is obtained by applying a ground truth linear function, corrupting them via additive noise ϵ, drawn independently and identically for each example (μ = 0 and standard deviation σ = 0.01):

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon.$$

Defining the model

From scratch

```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

PyTorch concise

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)
        self.net.weight.data.normal_(0, 0.01)
        self.net.bias.data.fill_(0)
```

```
@d2l.add_to_class(LinearRegression)  #@save
def forward(self, X):
    return self.net(X)
```

Defining the loss function

- In the implementation, we need to transform the true value y into the predicted value's shape y_hat. The result returned by the following method will also have the same shape as y_hat. We also return the averaged loss value among all examples in the minibatch.

From scratch

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

PyTorch concise

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

Defining the optimization algorithm

From scratch

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

PyTorch concise

```
@d2l.add_to_class(LinearRegression)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

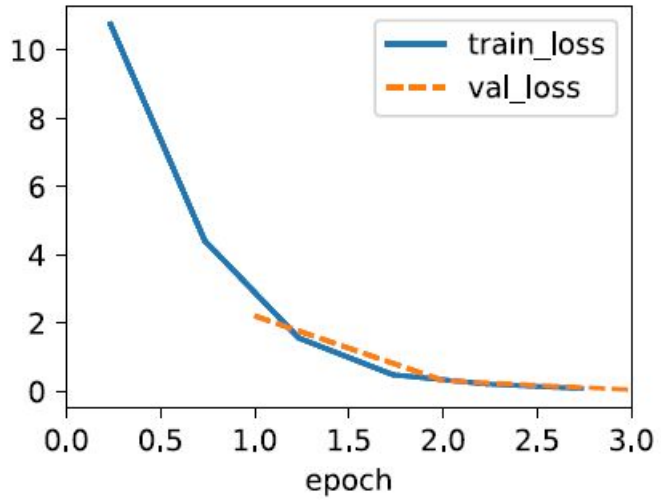
Training step

From scratch

```
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

Training with implementation from scratch

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```

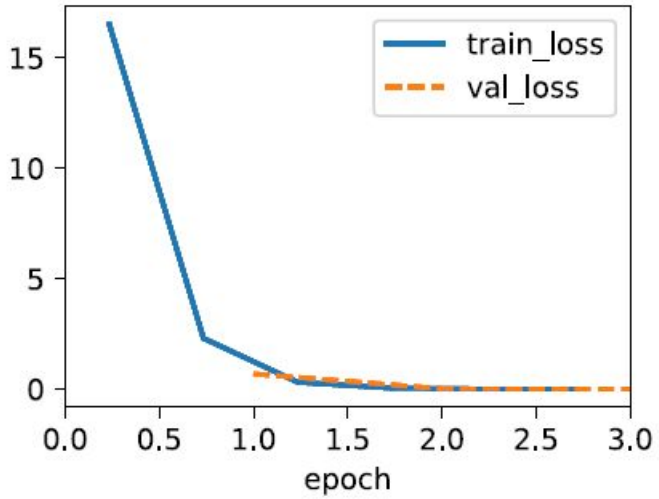



```
with torch.no_grad():
    print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
    print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: tensor([ 0.1408, -0.1493])
error in estimating b: tensor([0.2130])
```

Training with concise implementation

```
model = LinearRegression(lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```




```
def get_w_b(self):
    return (self.net.weight.data, self.net.bias.data)
w, b = model.get_w_b()
print(f'error in estimating w: {data.w - w.reshape(data.w.shape)}')
print(f'error in estimating b: {data.b - b}')
```

```
error in estimating w: tensor([ 0.0094, -0.0030])
error in estimating b: tensor([0.0137])
```

Global minima, local minima, generalization

- Linear regression happens to be a learning problem with a global minimum.
- However, the loss surfaces for deep networks contain many saddle points and minima. Fortunately, we typically do not care about finding an exact set of parameters but merely any set of parameters that leads to accurate predictions (and thus low loss).
- In practice, deep learning practitioners seldom struggle to find parameters that minimize the loss on training sets. Why should we believe that training data sampled from training distribution should tell us how to make predictions on test data generated by a different test distribution?
- The more challenging task is to find parameters that lead to accurate predictions on previously unseen data, a challenge called generalization.

Training error and generalization error

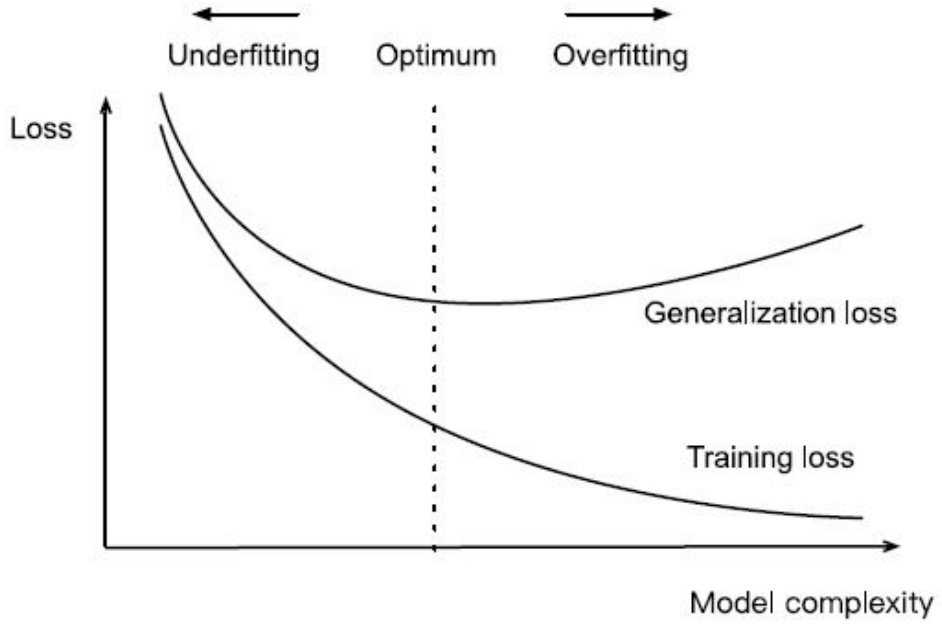
- We need to differentiate between the **training error**, also called bias, calculated on the training dataset, and the **generalization error**, also called variance, which is typically estimated by comparing the training error and the validation error.
- When we have simple models and abundant data, the training and generalization errors tend to be close. However, when we work with more complex models and/or fewer examples, we expect the training error to go down but the generalization gap to grow.
- When a model is capable of fitting arbitrary labels, low training error does not necessarily imply low generalization error. However, it does not necessarily imply high generalization error either! All we can say with confidence is that low training error alone is not enough to certify low generalization error.

Underfitting and overfitting

- When we compare the training and validation errors, we want to be mindful of two common situations.
- First, we want to watch out for cases when our training error and validation error are both substantial but there is a little gap between them. If the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model. Moreover, since the generalization gap between our training and generalization errors is small, we have reason to believe that we could get away with a more complex model. This phenomenon is known as **underfitting**.
- On the other hand, as we discussed above, we want to watch out for the cases when our training error is significantly lower than our validation error, indicating severe **overfitting**.

Model complexity

- A higher-order polynomial function is more complex than a lower-order polynomial function. Fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower-degree polynomials. However, the complexity may cause overfitting.



Dataset size

- As the above bound already indicates, another big consideration to bear in mind is dataset size. Fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting.
- As we increase the amount (and the representativeness) of training data, the generalization error typically decreases. Moreover, in general, more data never hurts.
- For a fixed task and data distribution, model complexity should not increase more rapidly than the amount of data. Given more data, we might attempt to fit a more complex model. Absent sufficient data, simpler models may be more difficult to beat.
- For many tasks, deep learning only outperforms linear models when many thousands of training examples are available.

Regularization with weight decay

- We can always mitigate overfitting by collecting more training data. However, that can be costly, time consuming, or entirely out of our control, making it impossible in the short run. Therefore, it is necessary to define possible **regularization** techniques.
- In the previous polynomial regression example we limited the model's capacity by tweaking the degree of the fitted polynomial. Indeed, limiting the number of features is a popular technique for mitigating overfitting.
- However, the most popular technique, namely **weight decay**, does not manipulate the number of parameters but it restricts the values that the parameters can take. Indeed, the increase of the norm of the parameter values is a significant clue of overfitting.

Regularization with weight decay

- Weight decay ensures a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss.
- Thus we replace our original objective, minimizing the prediction loss on the training labels, with new objective, minimizing the sum of the prediction loss and the penalty term. We characterize this trade-off via the regularization constant λ, a nonnegative hyperparameter that we fit using validation data.
- Now, if our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm rather than minimizing the training error.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

Regularization with weight decay

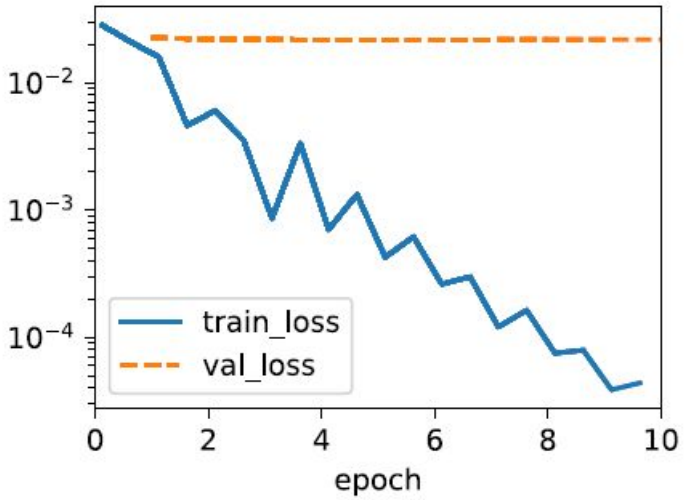
- Minibatch stochastic gradient descent acts now as follows

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)$$

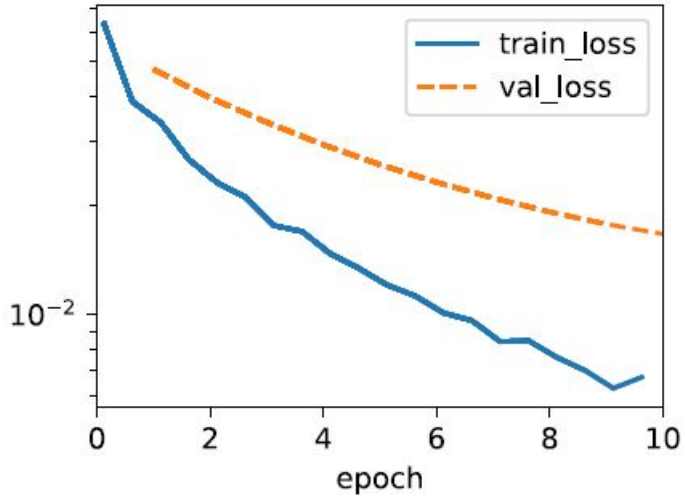
- As before, we update w based on the amount by which our estimate differs from the observation. However, we also shrink the size of w towards zero. That is why the method is called “weight decay”: given the penalty term alone, our optimization algorithm decays the weight at each step of training.
- Smaller values of λ correspond to less constrained w, whereas larger values of λ constrain w more considerably.

Regularization with weight decay

Without regularization



With regularization




```
def l2_penalty(w):
    return (w ** 2).sum() / 2
```

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def loss(self, y_hat, y):
        return (super().loss(y_hat, y) +
                self.lambd * l2_penalty(self.w))
```

Recap on linear neural networks

- Linear neural networks for classification
  - Dive into Deep Learning, Chap. 4
    - Classification (Softmax Regression) as a single layer fully connected neural network
    - Cross entropy loss

Classification

- Let's start with a simple image classification problem. Here, each input consists of a 2x2 grayscale image. We can represent each pixel value with a single scalar, giving us four features $x1$, $x2$, $x3$, $x4$. Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".
- These categorical data can be represented with one-hot encoding, that is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0.
- In our case, a label y would be a three-dimensional vector, with (1, 0, 0) corresponding to "cat", (0, 1, 0) to "chicken", and (0, 0, 1) to "dog":

$$y \in \{(1,0,0),(0,1,0),(0,0,1)\}$$

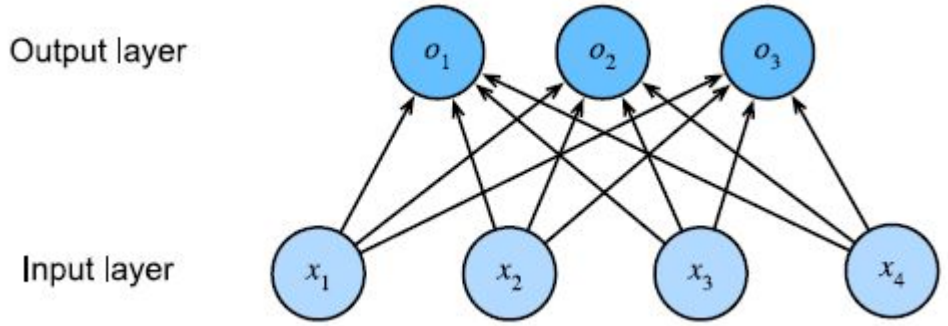
Classification with a linear model

- In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class.
- In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts).

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,$$
$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,$$
$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.$$

Classification with a single-layer neural network

- Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, $o1$, $o2$, and $o3$, depends on every input, $x1$, $x2$, $x3$, and $x4$, the output layer can also be described as a fully connected layer.



- For a more concise notation we use vectors and matrices: $o = Wx + b$ is much better suited for mathematics and code.
- Note that we have gathered all of our weights into a 3x4 matrix and all biases b in a vector.

Softmax

- Assuming a suitable loss function, we could try, directly, to minimize the difference between o and the labels y. While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:
  1. There is no guarantee that the outputs oi sum up to 1 in the way we expect probabilities to behave.
  2. There is no guarantee that the outputs oi are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Softmax

- We accomplish these goals by using an exponential function, that satisfies the requirement that the conditional class probability increases with increasing oi, it is monotonic, and all probabilities are nonnegative.
- We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called normalization.
- Putting these two pieces together gives us the **softmax** function:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}.$$

$$\underset{j}{\text{argmax}}\, \hat{y}_j = \underset{j}{\text{argmax}}\, o_j.$$

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b},$$

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{O}).$$

Cross entropy loss function for classification

- The softmax function gives us a vector $\hat{y}$, which we can interpret as the (estimated) conditional probabilities of each class. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

- We can take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

- where for any pair of label y and model prediction $\hat{y}$ over q classes, the loss function l is the **cross entropy**:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j.$$

Example of softmax regression

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
                              requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

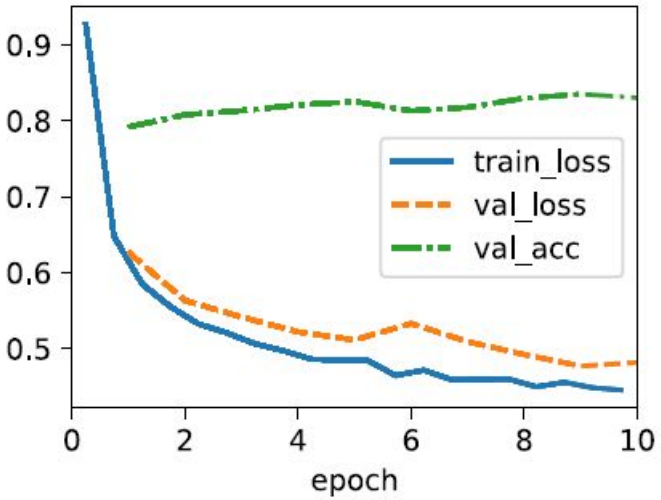
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

Example of softmax regression

```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

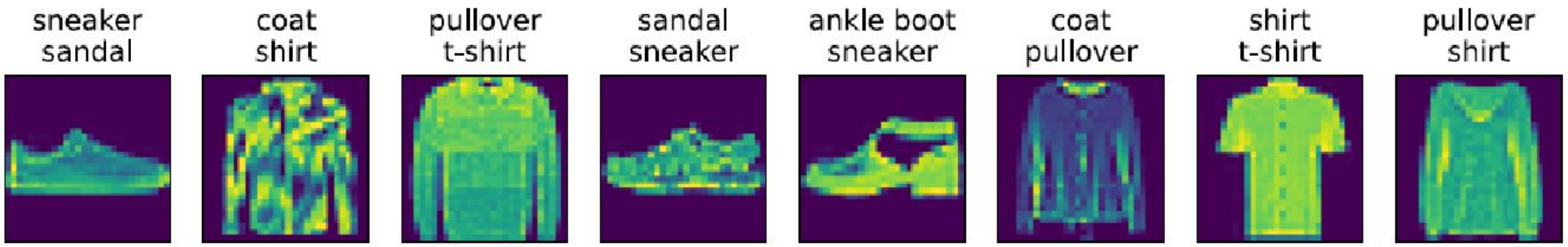
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



Example of softmax regression

```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



Recap on multi-layer perceptrons

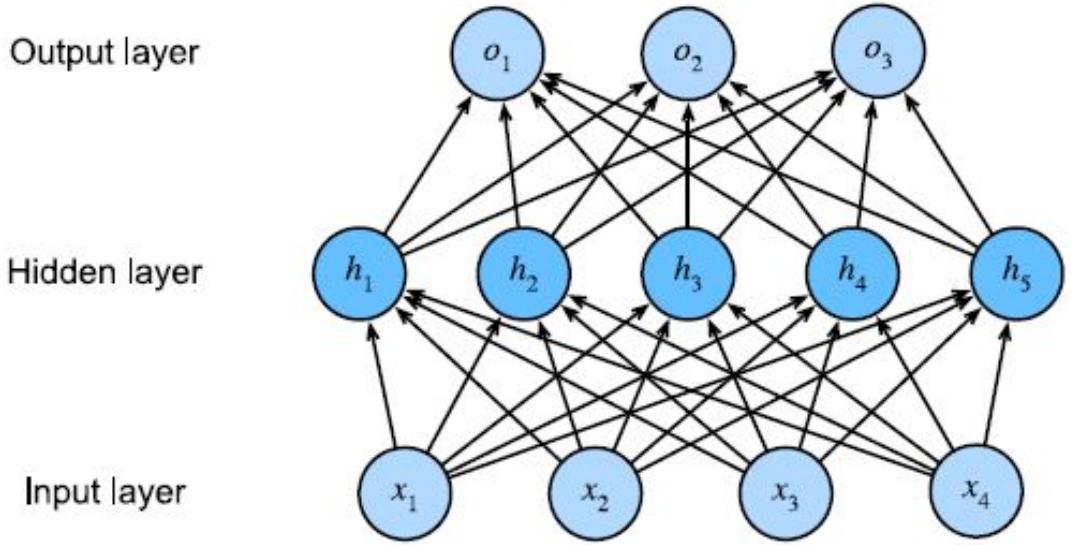
- Multi-layer perceptrons (MLPs)
  - Dive into Deep Learning, Chap. 5
    - Limitations of linear models
    - Hidden layers and activation functions
    - ReLu, Sigmoid, TanH
    - Forward and backward propagation
    - Batch size and memory
    - Vanishing and exploding gradients
    - Parameter initialization
    - Early stopping
    - Dropout

Limitations of linear models

- Linearity is a strong assumption in real world problems, especially because it implies **monotonicity**, namely that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative).
- Sometimes that makes sense. For example, if we were trying to predict whether an individual will repay a loan, we might reasonably assume that all other things being equal, an applicant with a higher income would always be more likely to repay than one with a lower income.
- But what about classifying images of cats and dogs? Should increasing the intensity of the pixel at location always increase (or always decrease) the likelihood that the image depicts a dog? Reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats and dogs is to assess the brightness of individual pixels. This approach is doomed to fail in a world where inverting an image preserves the category.

Hidden layers and multilayer perceptron

- We can overcome the limitations of linear models by incorporating one or more hidden layers. The easiest way to do this is to stack many fully connected layers on top of one another. Each layer feeds into the layer above it, until we generate outputs.
- We can think of the first L-1 layers as our representation and the final layer as our linear predictor. This architecture is commonly called a **multilayer perceptron**, often abbreviated as **MLP**.



Activation functions

- In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear activation function σ to be applied to each hidden unit following the affine transformation.
- The outputs of activation functions are called activations. In general, with activation functions in place, it is no longer possible to collapse a MLP into a linear model.

Linear

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Non Linear

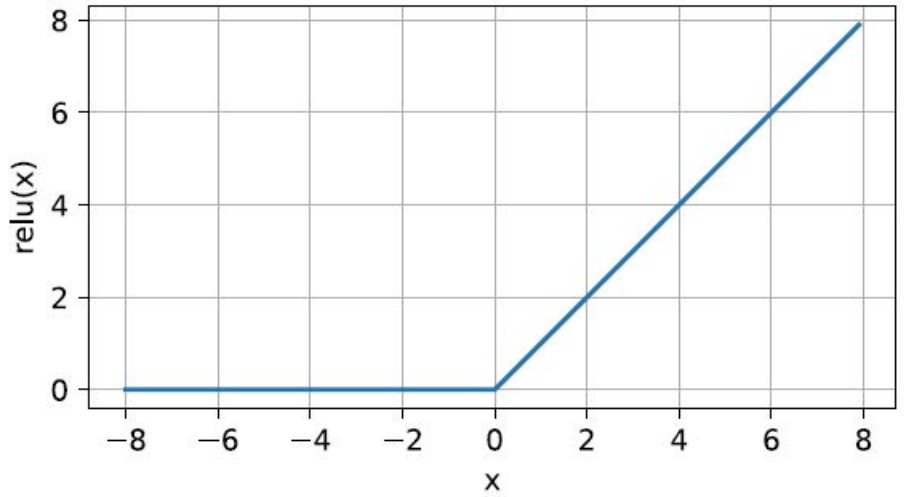
$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Activation functions - ReLU

- The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the rectified linear unit (ReLU). It retains only positive elements and discards all negative elements by setting the corresponding activations to 0 (piecewise linear).

$$\mathrm{ReLU}(x) = \max(x, 0).$$

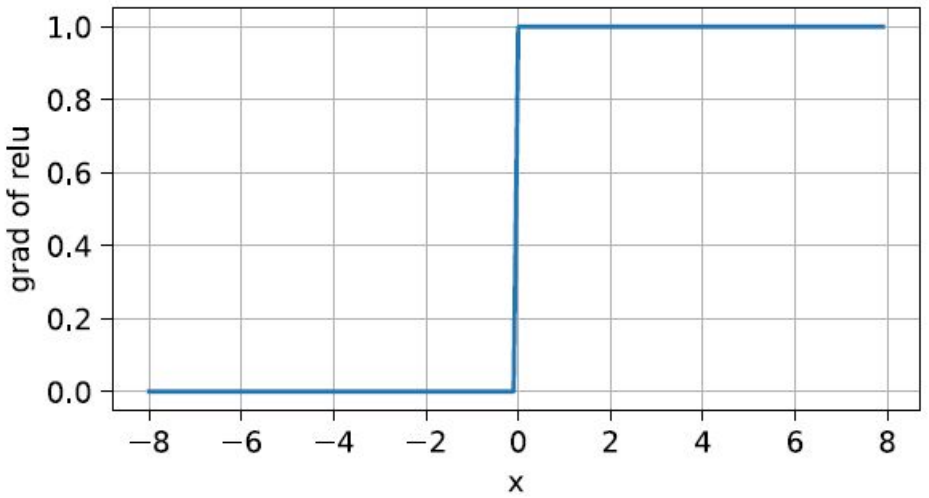
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Activation functions - ReLU

- When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



Activation functions – ReLU and pReLU

- The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks.
- Note that there are many variants to the ReLU function, including the parametrized ReLU (pReLU) function. This variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative:

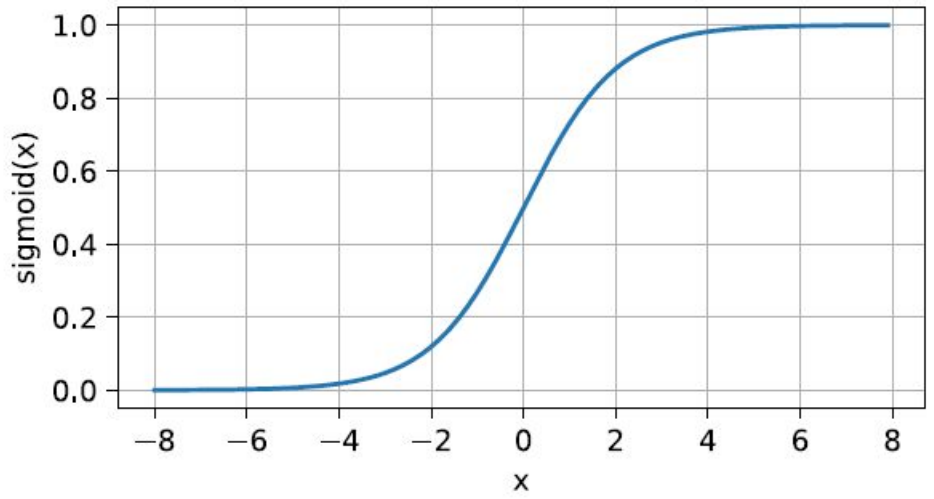
$$\mathrm{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

Activation functions – Sigmoid

- The sigmoid function transforms those inputs whose values lie in the domain R, to outputs that lie on the interval (0, 1). For that reason, the sigmoid is often called a squashing function: it squashes any input in the range (-inf, inf) to some value in the range (0, 1).

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



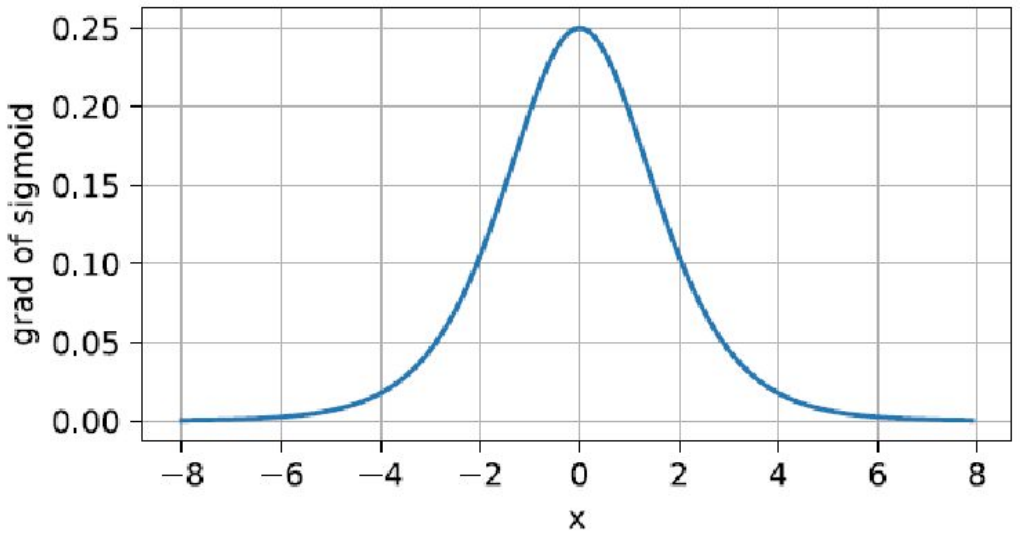
Activation functions – Sigmoid

- When attention shifted to gradient-based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still widely used as activation functions on the output units when we want to interpret the outputs as probabilities for binary classification problems.
- However, the sigmoid has largely been replaced by the simpler and more easily trainable ReLU for most use in hidden layers. Much of this has to do with the fact that the sigmoid poses challenges for optimization since its gradient vanishes for large positive and negative arguments (see next slide).
- Nonetheless sigmoids are important in recurrent neural networks to control the flow of information across time.

Activation functions – Sigmoid

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))$$

```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

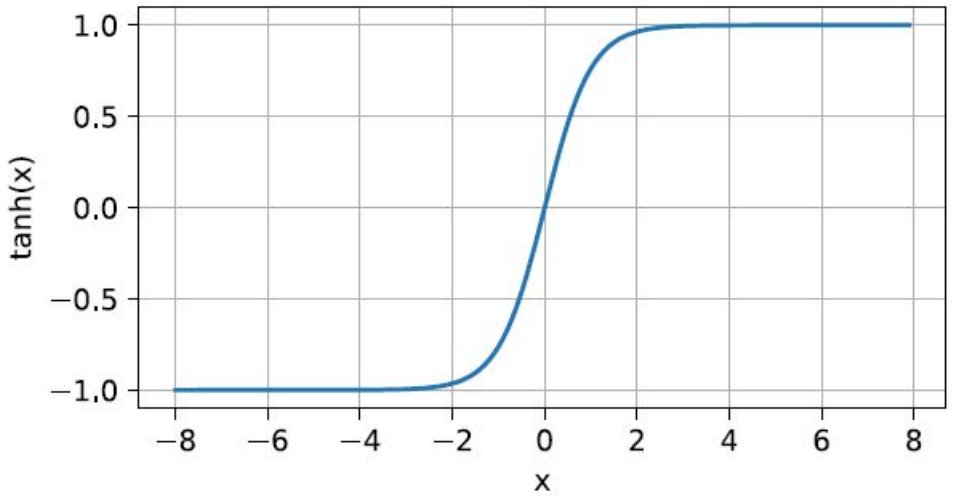


Activation functions – Tanh

- Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

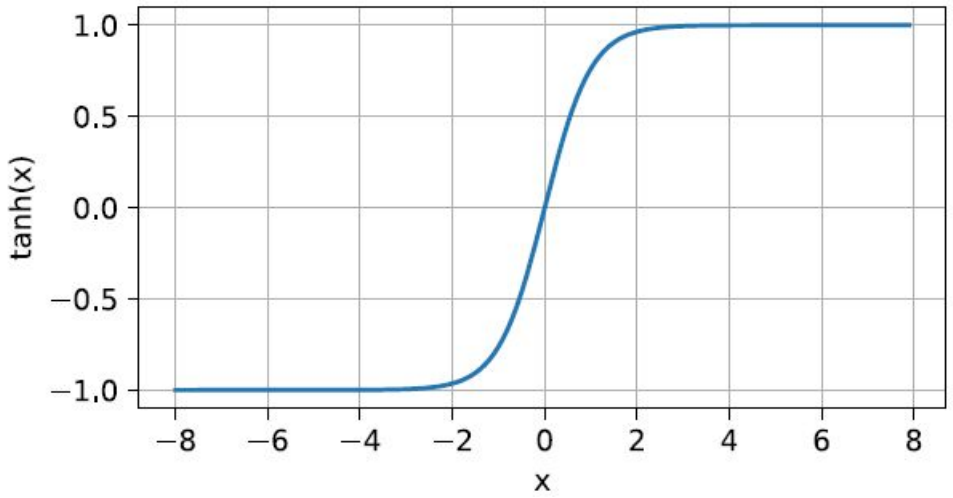


Activation functions – Tanh

- Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

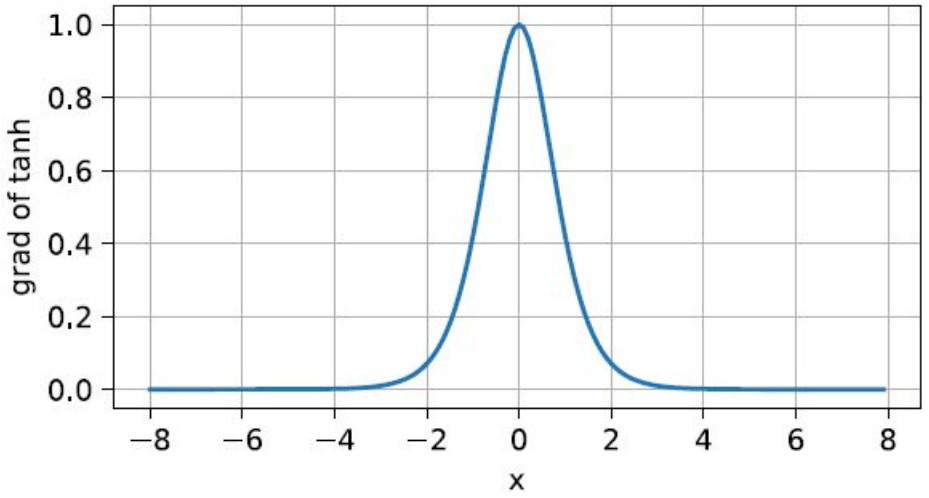


Activation functions – Tanh

- As the input nears 0, the derivative of the tanh function approaches a maximum of 1. And as we saw with the sigmoid function, as input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



Example of MLP

- Let's implement a MLP for the dataset Fashion MNIST, which contains 10 classes, and that is composed by images that are 28x28 = 784 grayscale pixel values. We can think of this as a classification dataset with 784 input features and 10 classes.
- We will implement a MLP with one hidden layer and 256 hidden units. Both the number of layers and their width are adjustable (they are considered hyperparameters). Typically, we choose the layer widths to be divisible by larger powers of
- 2. This is computationally efficient due to the way memory is allocated and addressed in hardware.

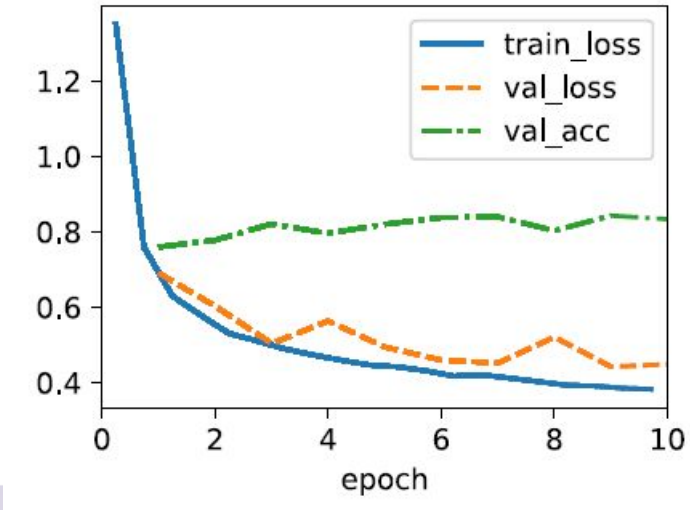
```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

Example of MLP

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

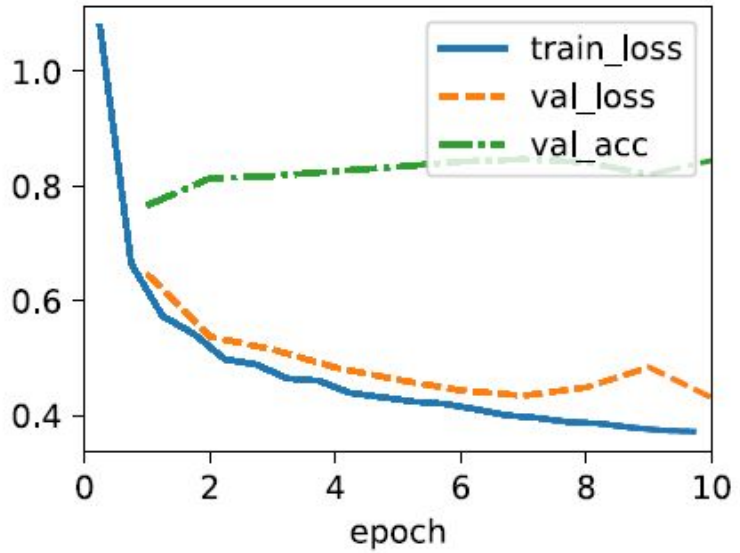
```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



PyTorch concise example of MLP

```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
```

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



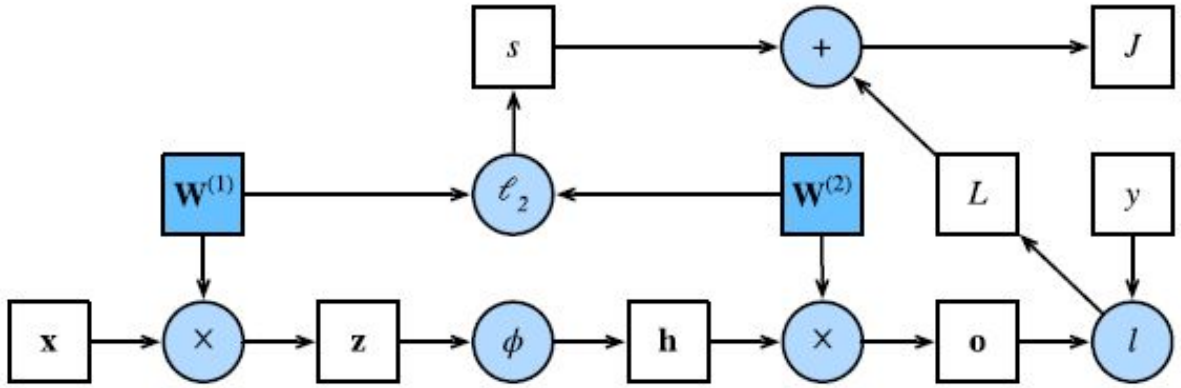
Forward and backward propagation

- So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in forward propagation through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the framework.
- The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

Forward propagation

- Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer.
- For the sake of simplicity, let's assume that the input example x assumes real values and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$



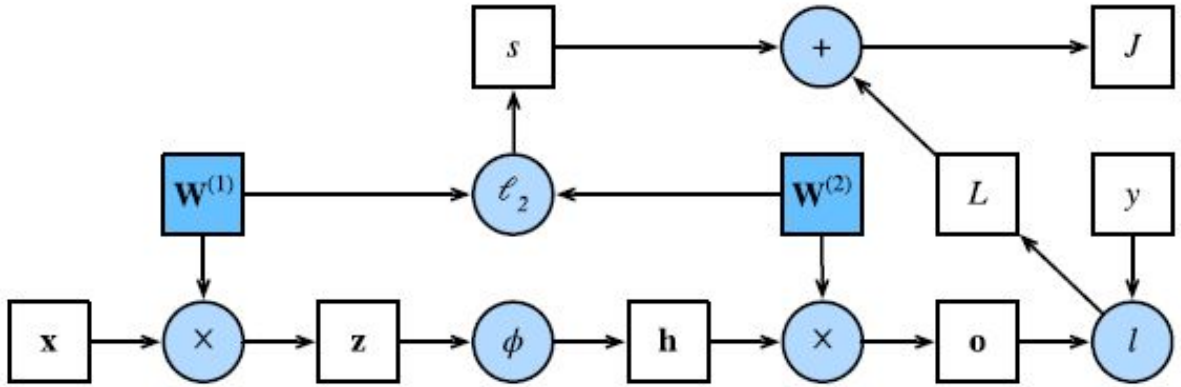
Forward propagation

- After running the intermediate variable z through the activation function ϕ we obtain our hidden activation vector of length h:

$$\mathbf{h} = \phi(\mathbf{z}).$$

- The hidden layer output h is also an intermediate variable. Assuming that the parameters of the output layer do not have bias, we can obtain an output layer variable with a vector of length q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$



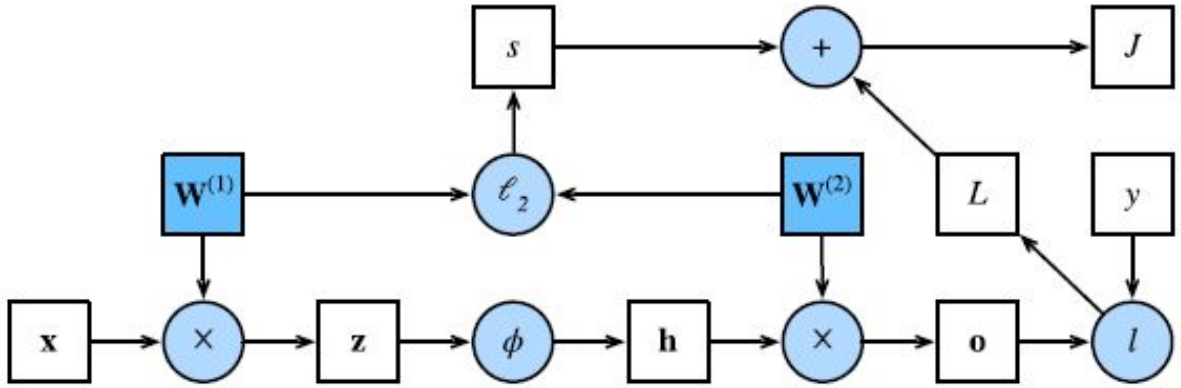
Forward propagation

- Assuming that the loss function is l, the example label is y, and the regularization parameter is λ, we can compute the loss function L and the regularization term s:

$$L = l(\mathbf{o}, y). \qquad s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right)$$

- The regularized loss, namely the final objective function J, is:

$$J = L + s.$$



Backpropagation

- Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters.
- Assume that we have functions Y = f(X) and Z = g(Y), in which the input and the output X, Y, Z are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z with respect to X via:

$$\frac{\partial \mathrm{Z}}{\partial \mathrm{X}} = \mathrm{prod}\left(\frac{\partial \mathrm{Z}}{\partial \mathrm{Y}}, \frac{\partial \mathrm{Y}}{\partial \mathrm{X}}\right).$$

Backpropagation

- Recall that the parameters of the simple network with one hidden layer are W(1) and W(2).
- The objective of backpropagation is to calculate the gradients:

$$\partial J/\partial \mathbf{W}^{(1)} \qquad \partial J/\partial \mathbf{W}^{(2)}$$

- To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters.

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$$

Training a neural network

- When training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations.
- One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction.
- Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to out-of-memory errors.

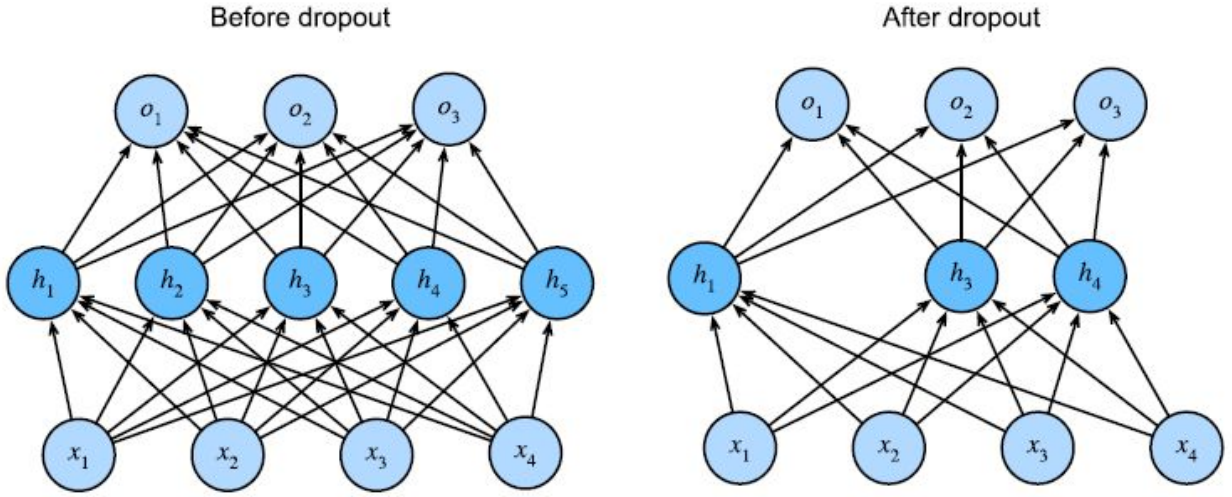
Generalization by early stopping

- Early stopping is a classic technique for regularizing deep neural networks. Here, rather than directly constraining the values of the weights, one constrains the number of epochs of training.
- The most common way to determine the stopping criterion is to monitor validation error throughout training (typically by checking once after each epoch) and to cut off training when the validation error has not decreased by more than some small amount ϵ for some number of epochs. This is sometimes called a patience criterion.
- As well as the potential to lead to better generalization in the setting of noisy labels, another benefit of early stopping is the time saved. Once the patience criterion is met, one can terminate training. For large models that might require days of training, well-tuned early stopping can save time and money.

Regularization by dropout

- A popular regularization method is called dropout, because it literally drops out some neurons during training. In other words, with dropout probability p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$



Example of MLP with dropout

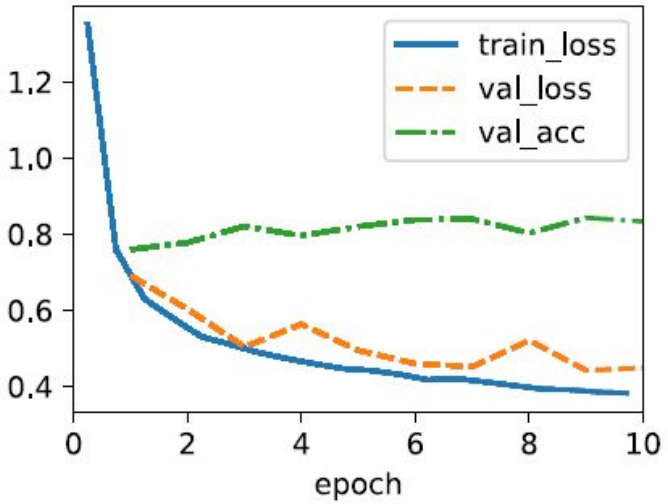
```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return torch.zeros_like(X)
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.lin1 = nn.LazyLinear(num_hiddens_1)
        self.lin2 = nn.LazyLinear(num_hiddens_2)
        self.lin3 = nn.LazyLinear(num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((X.shape[0], -1))))
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = self.relu(self.lin2(H1))
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return self.lin3(H2)
```

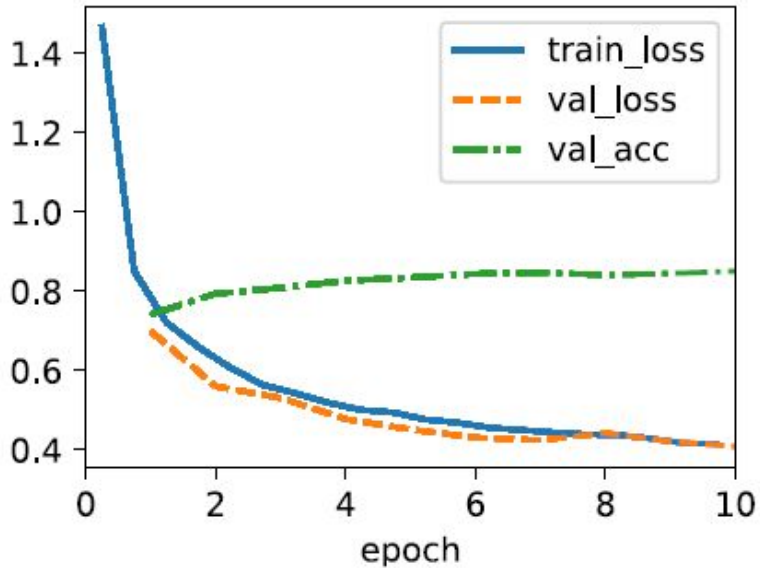
Example of MLP with dropout

```
hparams = {'num_outputs':10, 'num_hiddens_1':256, 'num_hiddens_2':256,
           'dropout_1':0.5, 'dropout_2':0.5, 'lr':0.1}
model = DropoutMLPScratch(**hparams)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

Without dropout



With dropout



CNNs (Dive into Deep Learning, Chap. 7)

- From fully connected layers to convolutions
  - Translation invariance
  - Locality
- Convolutions for images
  - CNN, Channels, Feature maps
  - Example of convolution
  - Example of convolutional layer
- Padding and stride
- Multiple Input and Multiple Output Channels
  - Example of 1x1 convolutional layer
- Pooling
  - Max and Average pooling
  - Multiple Channels
- Example: LeNet

From fully connected layers to convolutions

- To solve this problem, there are three main desiderata that guided the design of Convolutional Neural Networks as constrained versions of MLPs:
  1. In the earliest layers, the network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called **translation invariance**.
  2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the **locality principle**. Eventually, these local representations can be aggregated to make predictions at the whole image level.
  3. As we proceed, deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

Translation invariance and locality

- A hidden representation H with a spatial structure [i,j] would require, for 1000x1000 images with a single fully connected layer, 10^{12} parameters, that are too much.

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathbf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l}$$
$$= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathbf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}$$

Impose convolutions with a fixed kernel $[V]_{a,b}$ to obtain translation invariance

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}$$

We need $4x10^6$ parameters due to a,b in [-1000,1000]

According to the locality principle, we can move in the neighborhood $[-\Delta, \Delta]$

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}$$

It becomes a convolutional layer which requires $4x\Delta^2$ parameters

Convolutional Neural Network (CNN)

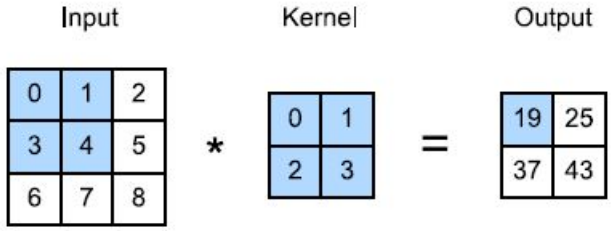
- Convolutional neural networks (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community, V is referred to as a convolution kernel, a filter, or simply the layer's weights that are learnable parameters.
- While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations.
- The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models might struggle even to fit our training data.

Channels and feature maps

- There is just one problem with this approach. So far, we blissfully ignored that images consist of three channels: red, green, and blue. In sum, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and c channels.
- Moreover, just as our input consists of a third-order tensor, it turns out to be a good idea to similarly formulate our hidden representations as third-order tensors H. In other words, rather than just having a single hidden representation corresponding to each spatial location, we want an entire vector of d hidden representations corresponding to each spatial location.
- As in the inputs, these are sometimes called channels. They are also sometimes called **feature maps**, as each provides a spatialized set of learned features for the subsequent layer.

$$[\mathbf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathbf{V}]_{a,b,c,d} [\mathbf{X}]_{i+a,j+b,c}$$

Example of 2D convolution




```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$

$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$

$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

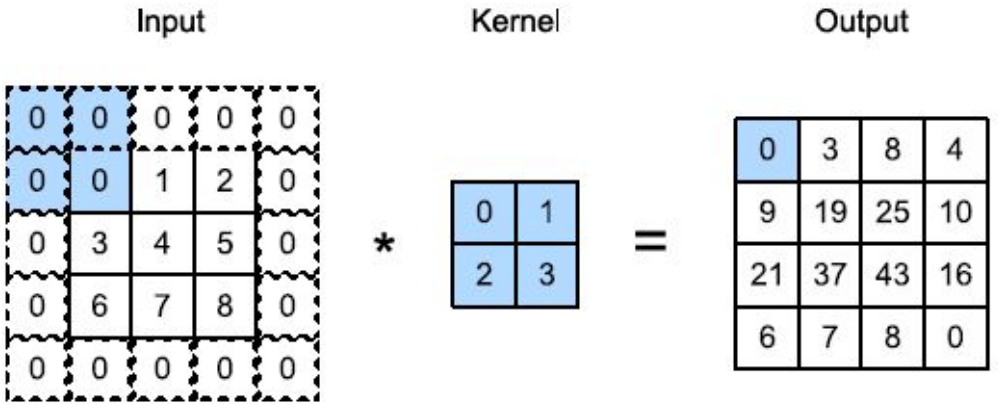
Example of convolutional layer

- A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias.
- When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully connected layer.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Padding

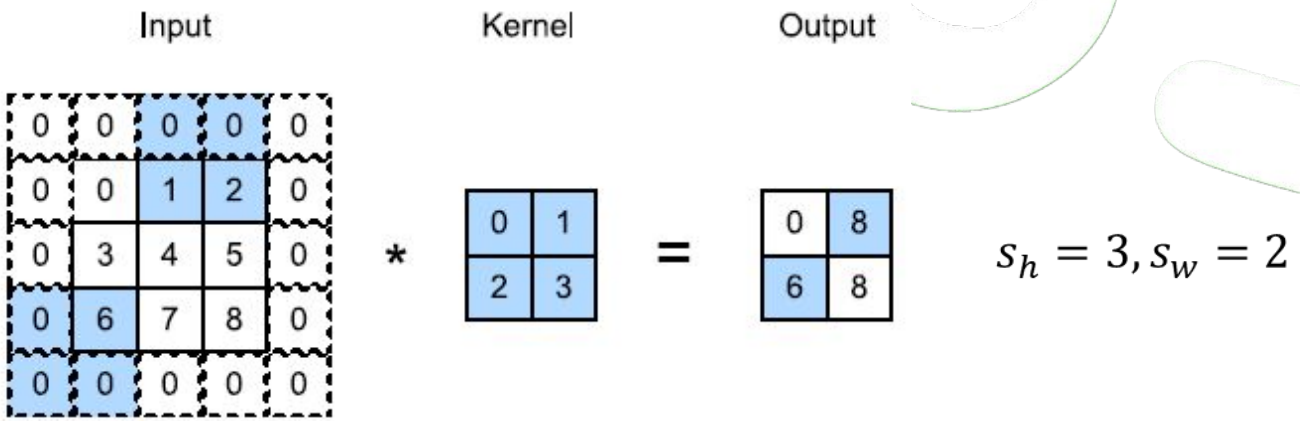
- Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image (**padding**), thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero.



$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$ Output shape

Stride

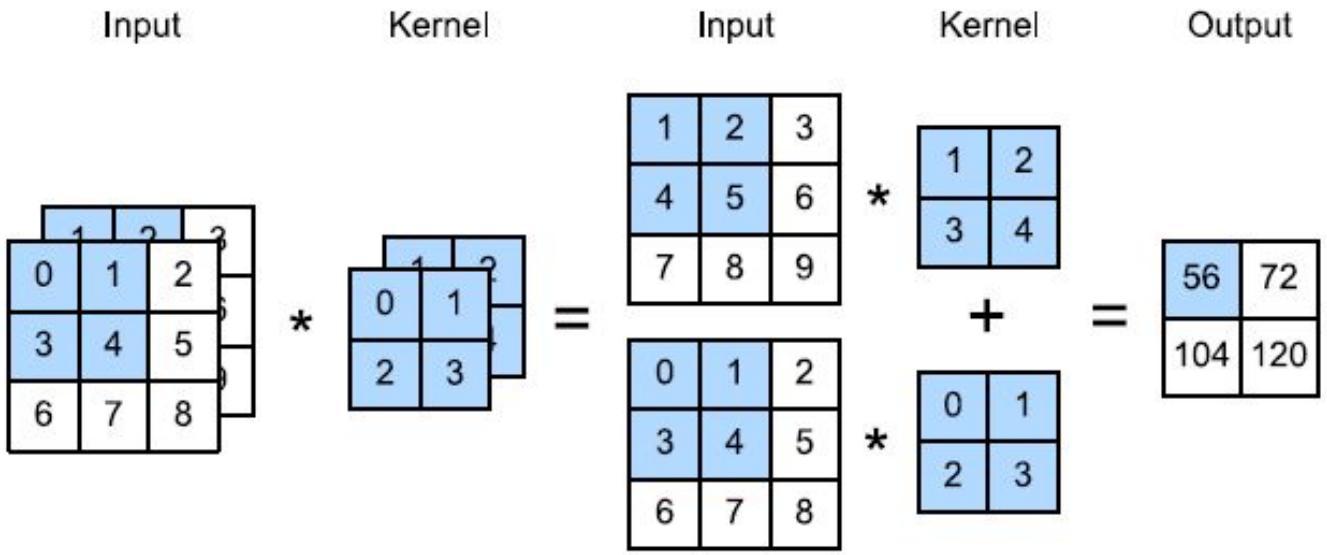
- In the previous examples, we defaulted to sliding one element at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations.
- We refer to the number of rows and columns traversed per slide as stride. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride.



$s_h = 3, s_w = 2$

$\lfloor (n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + s_\mathrm{h})/s_\mathrm{h} \rfloor \times \lfloor (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + s_\mathrm{w})/s_\mathrm{w} \rfloor$. Output shape

Multiple input channels




```
def corr2d_multi_in(X, K):
    # Iterate through the 0th dimension (channel) of K first, then add them up
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

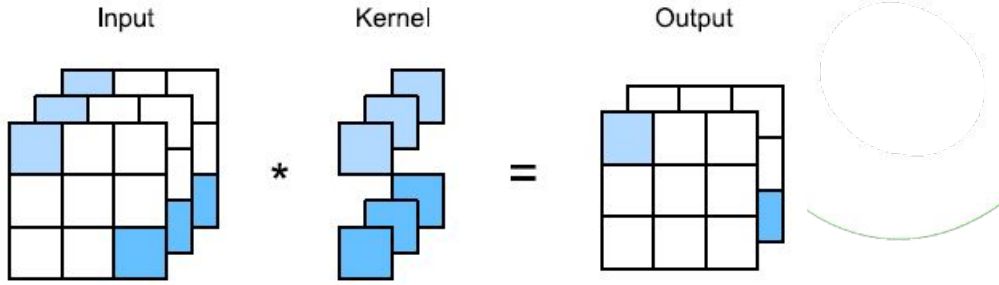
Multiple output channels (feature maps)

- In the most popular neural network architectures, we actually increase the channel dimension as we go deeper in the neural network, typically downsampling to trade off spatial resolution for greater channel depth.
- Intuitively, you could think of each channel as responding to a different set of features. A naive interpretation would suggest that representations are learned independently per pixel or per channel. Instead, channels are optimized to be jointly useful.
- In any case, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are
    # stacked together
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

1X1 convolutional layer

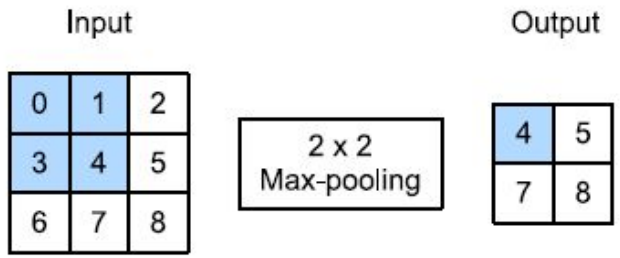
- At first, a 1x1 convolution does not seem to make much sense. The only computation of the 1x1 convolution occurs on the channel dimension. You could think of it as constituting a fully connected layer applied at every single pixel location to transform the corresponding input values into a certain number of output values.




```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    # Matrix multiplication in the fully connected layer
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

Pooling

- Pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the pooling window).
- However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no kernel).
- Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window.
- These operations are called maximum pooling (max-pooling for short) and average pooling, respectively.



Pooling on multiple channels

- When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer.
- This means that the number of output channels for the pooling layer is the same as the number of input channels.

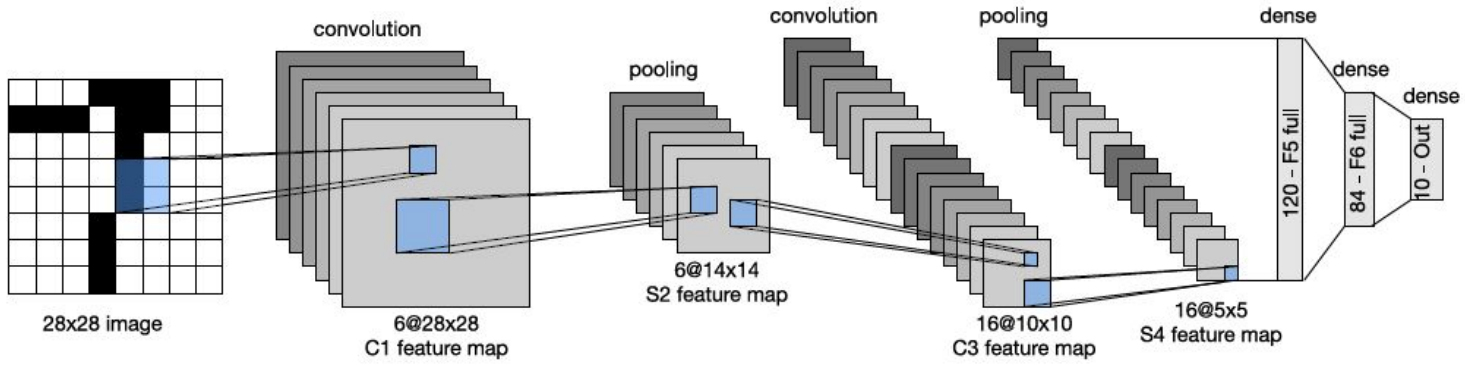
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]])
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

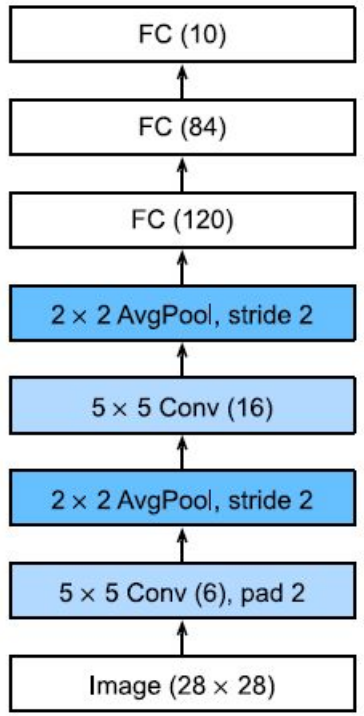
Example: LeNet-5

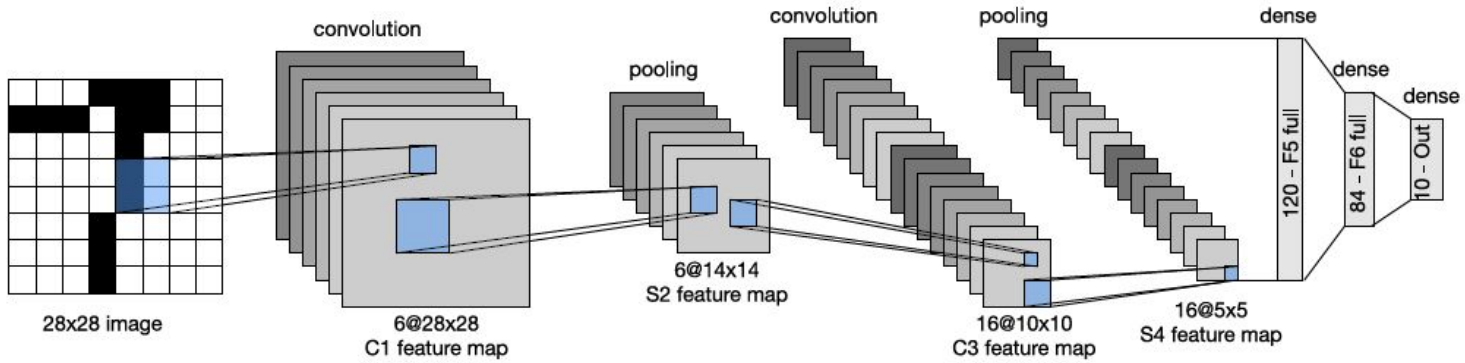
- LeNet (LeNet-5) consists of two parts: (i) a convolutional encoder consisting of two convolutional layers; and (ii) a dense block consisting of three fully connected layers.
- The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and maxpooling work better, they had not yet been discovered.
- Each convolutional layer uses a 5x5 kernel and a sigmoid. These layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels.



Example: LeNet-5

- The first convolutional layer has 6 output channels, while the second has 16. Each 2x2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling.
- In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. LeNet's dense block has three fully connected layers, with 120, 84, and 10 outputs, respectively. Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.

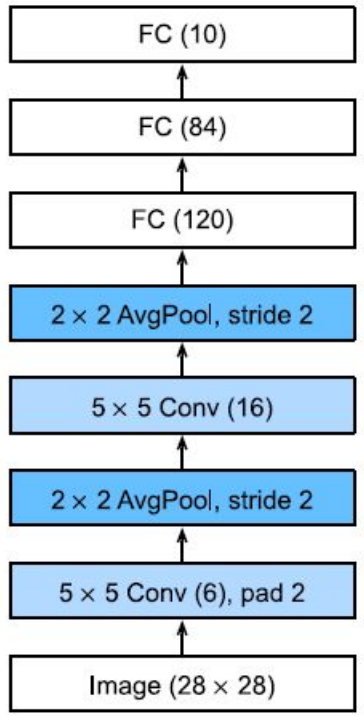




Example: LeNet-5

```
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

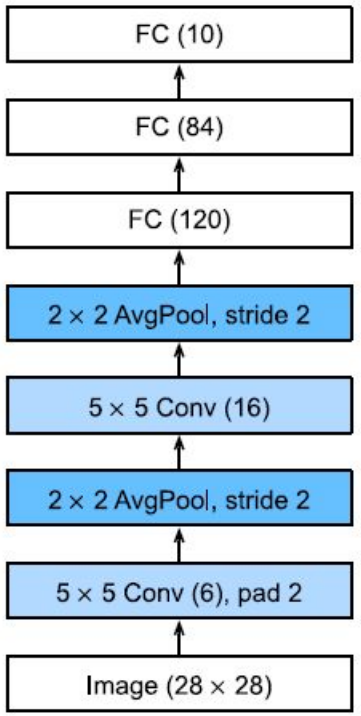
```
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```



Example: LeNet-5

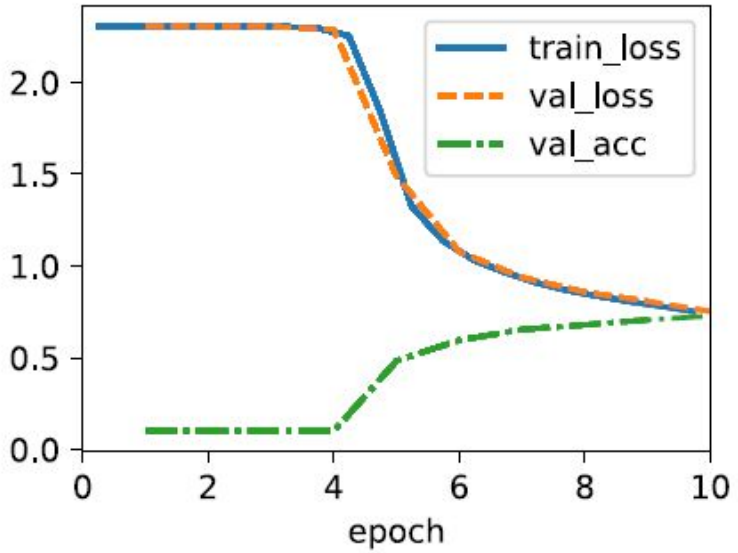
```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```



Example: LeNet-5

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```

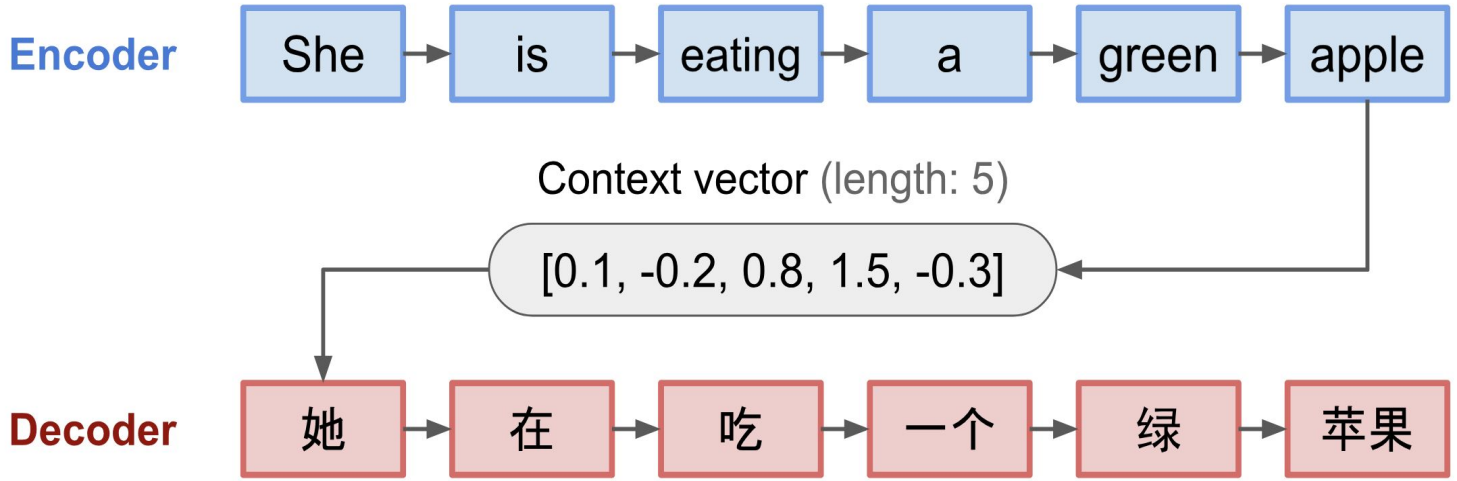


Outline

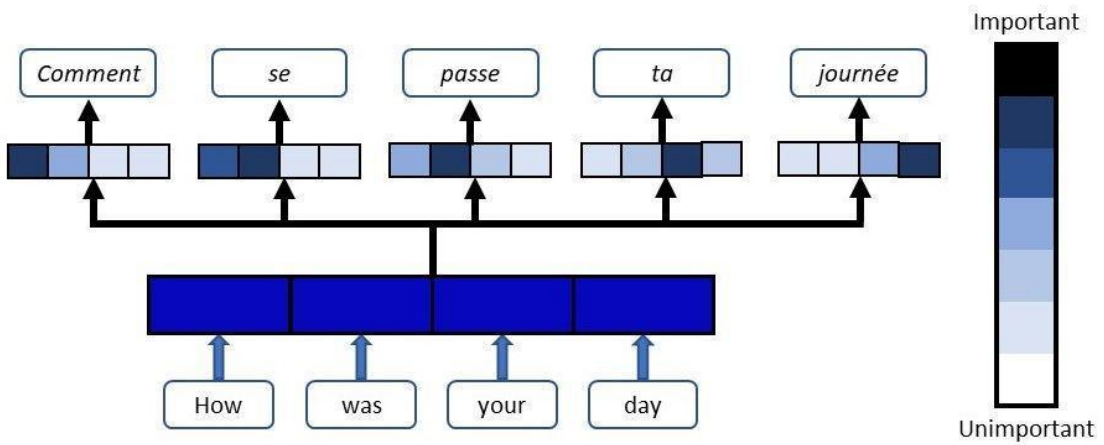
- Limitations of RNNs
- Transformer
- Transformer’s Input
- Self Attention

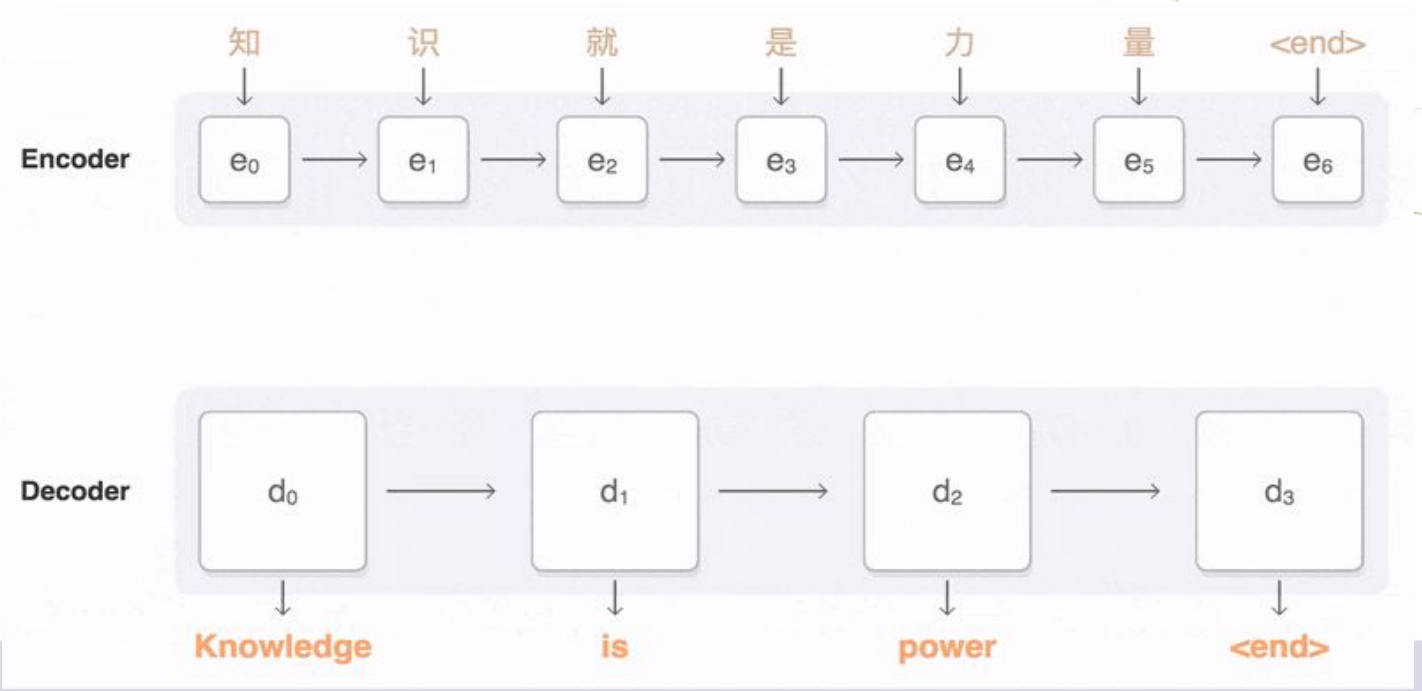
Limitations of RNNs

- RNNs lack of long-term memory (enc-dec models)
- RNNs are extremely slow to train (for long series)
- RNNs suffer of the vanishing gradient problem



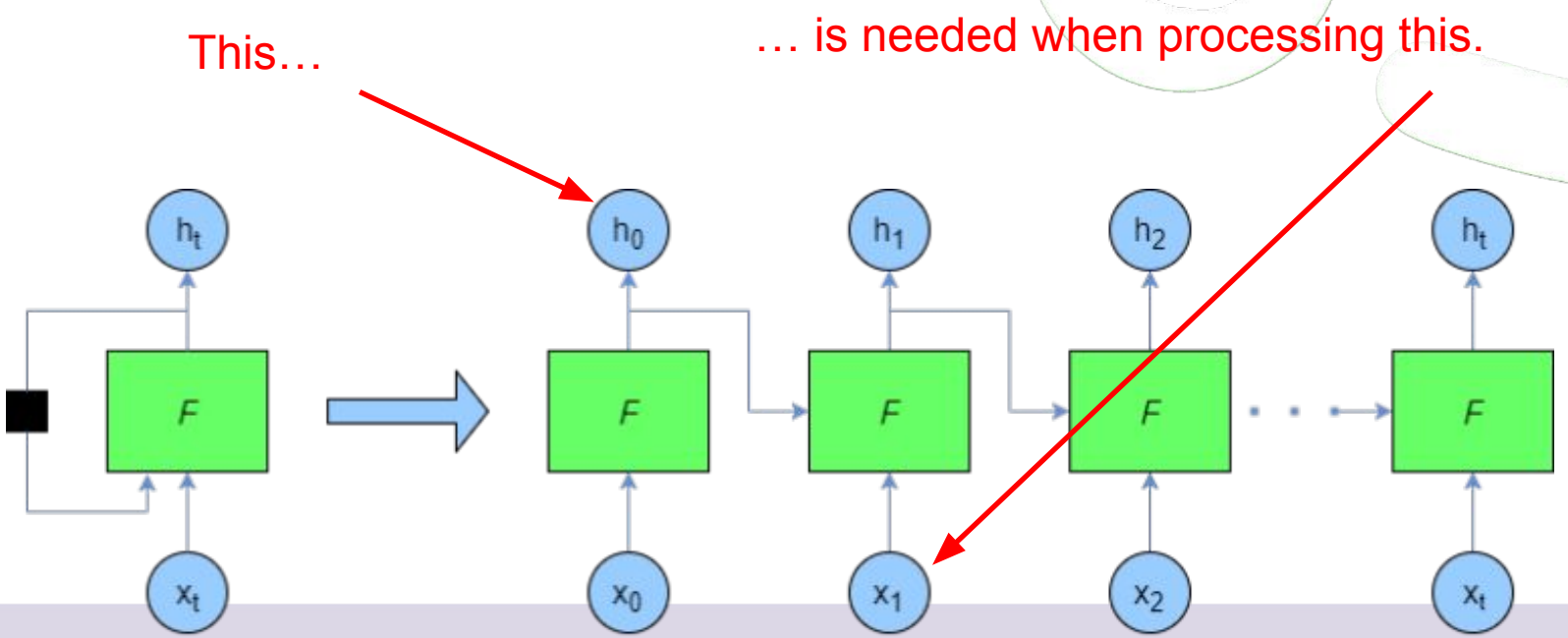
RNNs lack of long-term memory





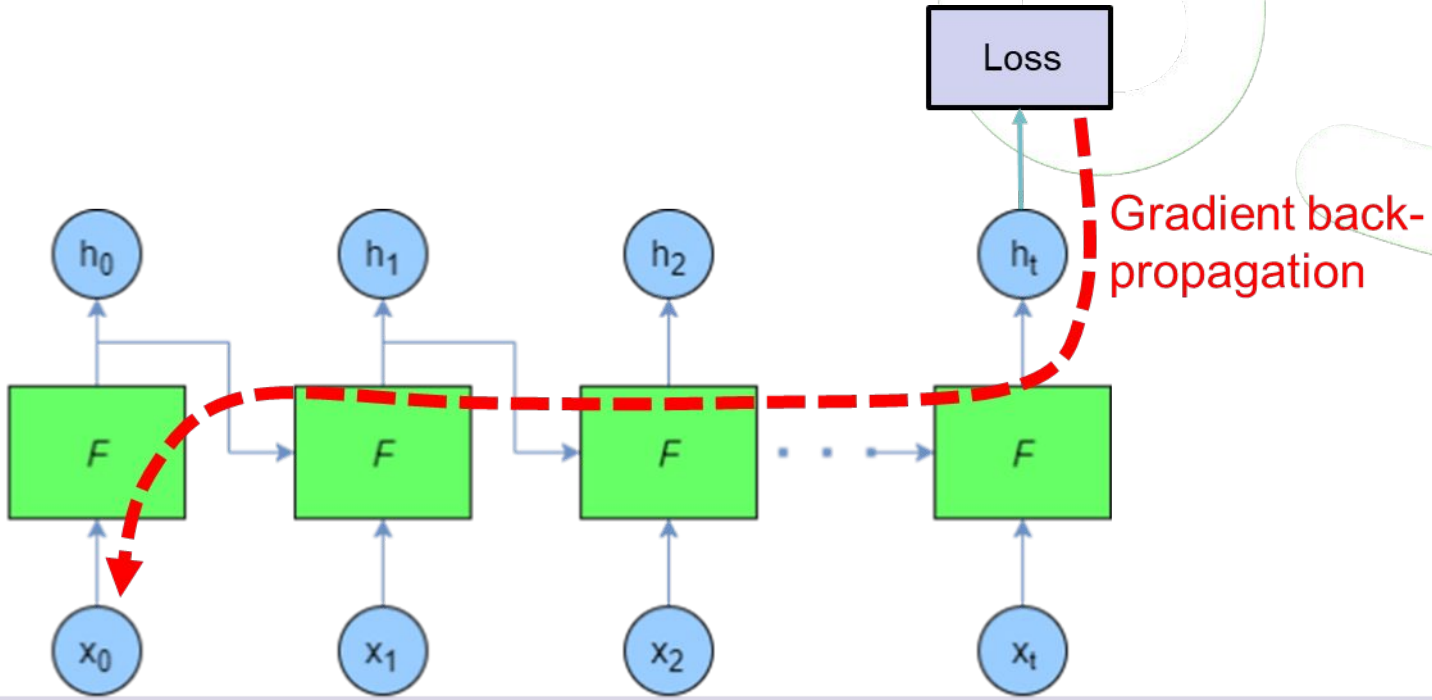
RNNs are extremely slow to train

- Processing is inherently sequential
- The network can not start processing x_i until it has finished with x_{i-1}
- Thus the network cannot exploit the massive parallelism available in a modern GPU!



RNNs suffer of the vanishing gradient problem

- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times



RNNs suffer of the vanishing gradient problem

- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times

$$\frac{\partial Loss}{\partial h_0} = \frac{\partial Loss}{\partial h_1}\cdot\frac{\partial F}{\partial h_0} = \frac{\partial Loss}{\partial h_2}\cdot\frac{\partial F}{\partial h_1}\cdot\frac{\partial F}{\partial h_0} = \frac{\partial Loss}{\partial h_3}\cdot\frac{\partial F}{\partial h_2}\cdot\frac{\partial F}{\partial h_1}\cdot\frac{\partial F}{\partial h_0}$$
$$= \frac{\partial Loss}{\partial h_4}\cdot\frac{\partial F}{\partial h_3}\cdot\frac{\partial F}{\partial h_2}\cdot\frac{\partial F}{\partial h_1}\cdot\frac{\partial F}{\partial h_0} = \dots$$

The derivatives of F are multiplied several times by themselves…

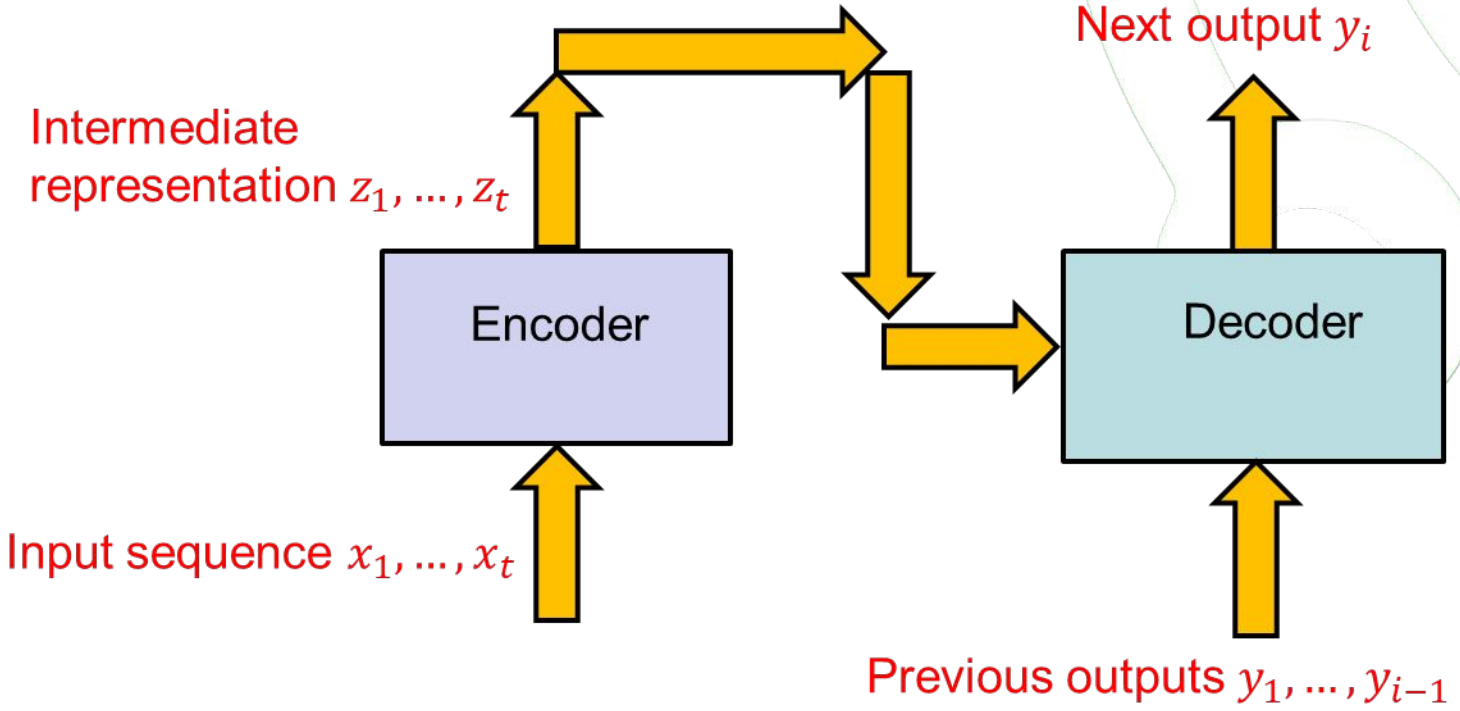
RNNs suffer of the vanishing gradient problem

- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times
- So, if the absolute value of the derivatives of F is small, in this process it will become smaller and smaller… (vanishing gradient)
- … and if it is large, it will be come larger and larger (exploding gradient), causing problems to the stability of the algorithm
- Note: the problem is caused by the fact that we are traversing many times (sequence length) the same layer

Transformer

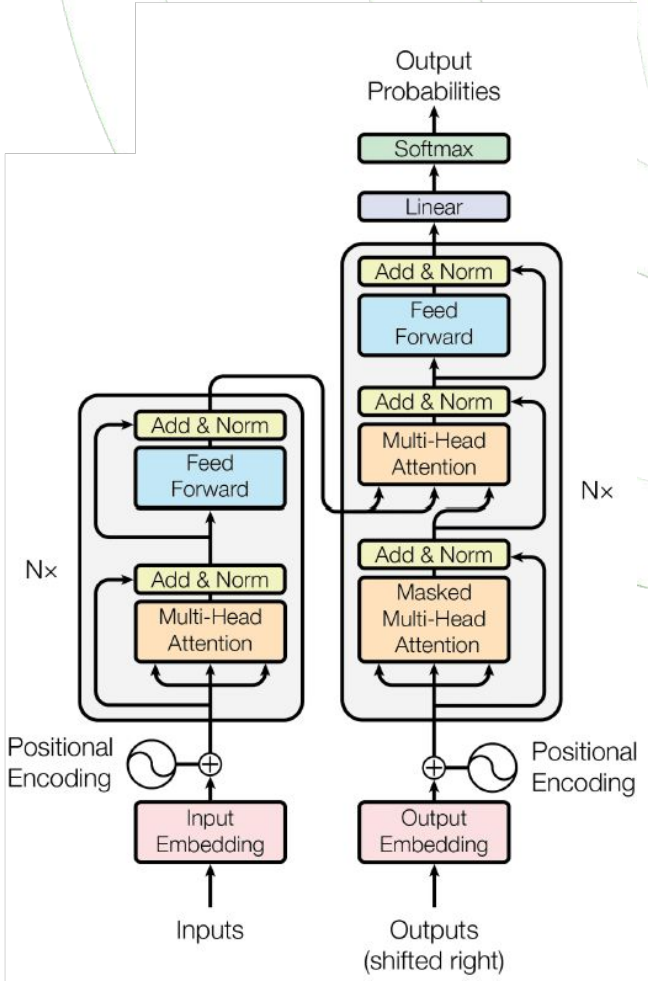
- In 2017, a group of researchers at Google Brain proposed an alternative model for processing sequential data
- In this model, the elements of the sequence can be processed in parallel
- The number of layers traversed does not depend on the length of the sequence (so, no problems with the gradient)
- The model was introduced for language translation (sequence to sequence with different lengths); so, it was called Transformer.
- Subsets of the model can be used for the other sequence processing tasks

Transformer



Transformer

- Input
  - Tokenization
  - Input embedding
  - Positional encoding
- Encoder
  - Attention
  - Query
  - Key
  - Value
  - Self Attention
  - Multi-head Attention
  - Add & Norm
  - Feedforward
- Decoder
  - Masked attention
  - Encoder decoder attention



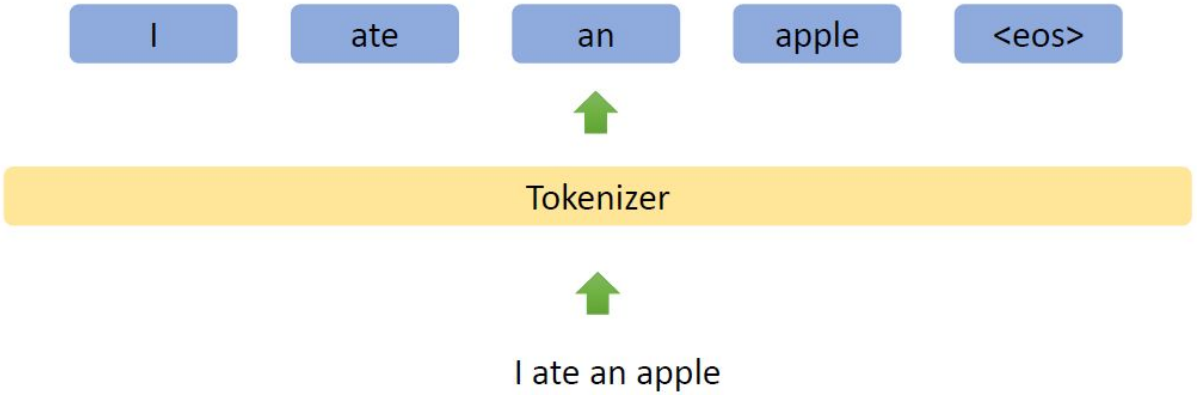
Tokenization

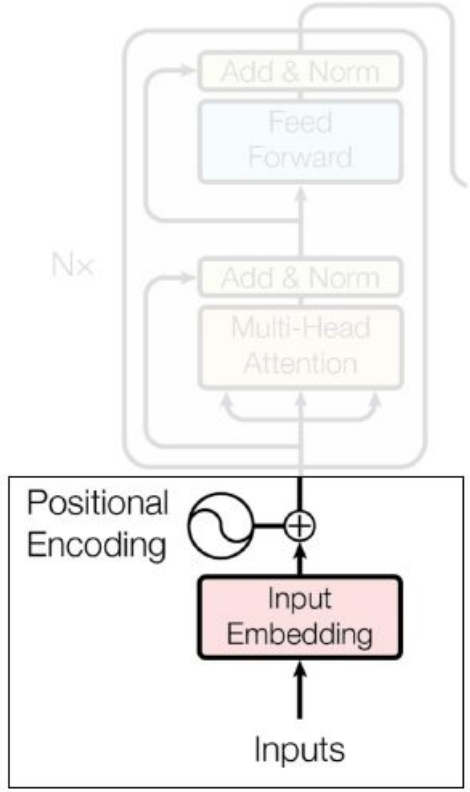
- Representation of text with a set of tokens
- Each token is encoded with a unique id

Tokenization method	Tokens	Token count	Vocab size
Sentence	'The moon, Earth's only natural satellite, has been a subject of fascination and wonder for thousands of years.'	1	# sentences in doc
Word	'The', 'moon,', "Earth's", 'only', 'natural', 'satellite,', 'has', 'been', 'a', 'subject', 'of', 'fascination', 'and', 'wonder', 'for', 'thousands', 'of', 'years.'	18	171K (English[1])
Sub-word	'The', 'moon', ',', 'Earth', "'", 's', 'on', 'ly', 'n', 'atur', 'al', 's', 'ate', 'll', 'it', 'e', ',', 'has', 'been', 'a', 'subject', 'of', 'fascinat', 'ion', 'and', 'w', 'on', 'd', 'er', 'for', 'th', 'ous', 'and', 's', 'of', 'y', 'ears', '.'	37	(varies)
Character	'T', 'h', 'e', ' ', 'm', 'o', 'o', 'n', ',', ' ', 'E', 'a', 'r', 't', 'h', "'", 's', ' ', 'o', 'n', 'l', 'y', ' ', 'n', 'a', 't', 'u', 'r', 'a', 'l', ' ', 's', 'a', 't', 'e', 'l', 'l', 'i', 't', 'e', ',', ' ', 'h', 'a', 's', ' ', 'b', 'e', 'e', 'n', ' ', 'a', ' ', 's', 'u', 'b', 'j', 'e', 'c', 't', ' ', 'o', 'f', ' ', 'f', 'a', 's', 'c', 'i', 'n', 'a', 't', 'i', 'o', 'n', ' ', 'a', 'n', 'd', ' ', 'w', 'o', 'n', 'd', 'e', 'r', ' ', 'f', 'o', 'r', ' ', 't', 'h', 'o', 'u', 's', 'a', 'n', 'd', 's', ' ', 'o', 'f', ' ', 'y', 'e', 'a', 'r', 's', '.'	110	52 + punctuation (English)

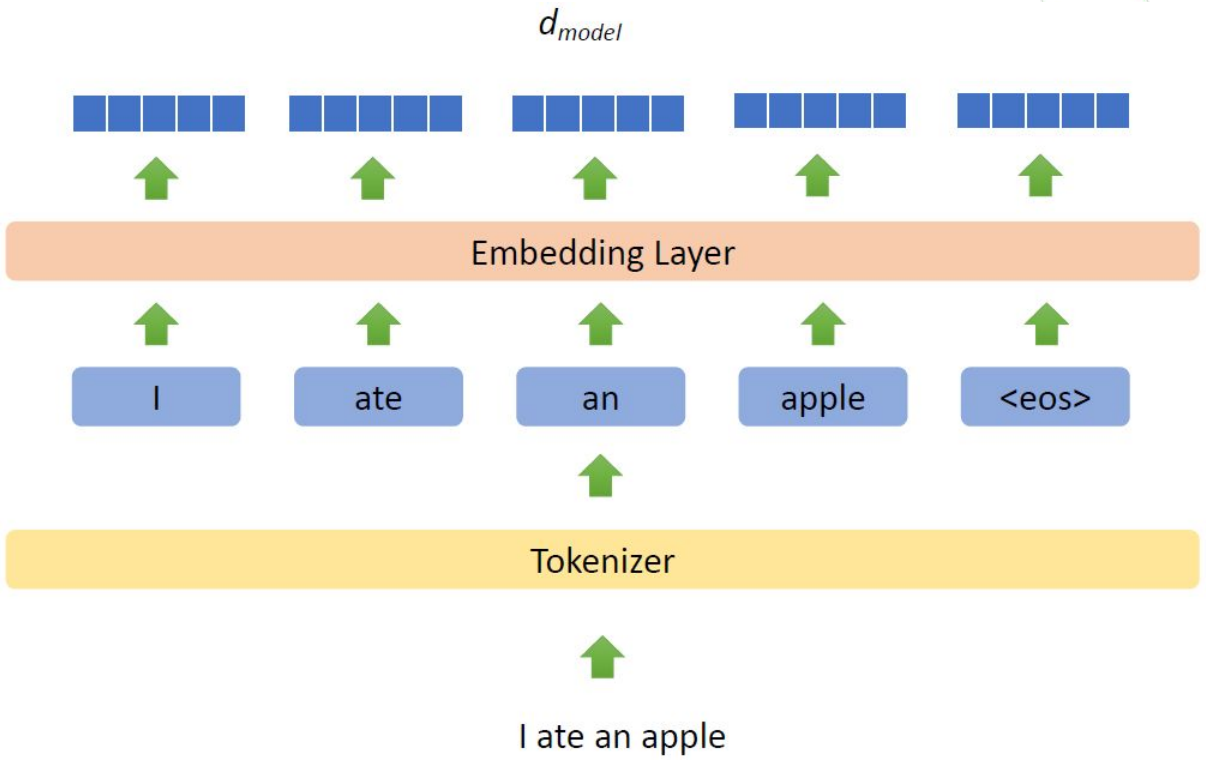
[1]Source: BBC.com

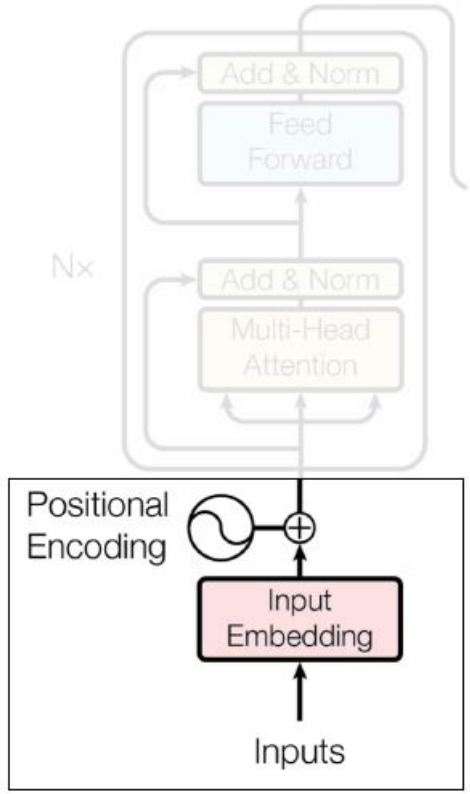
Tokenization





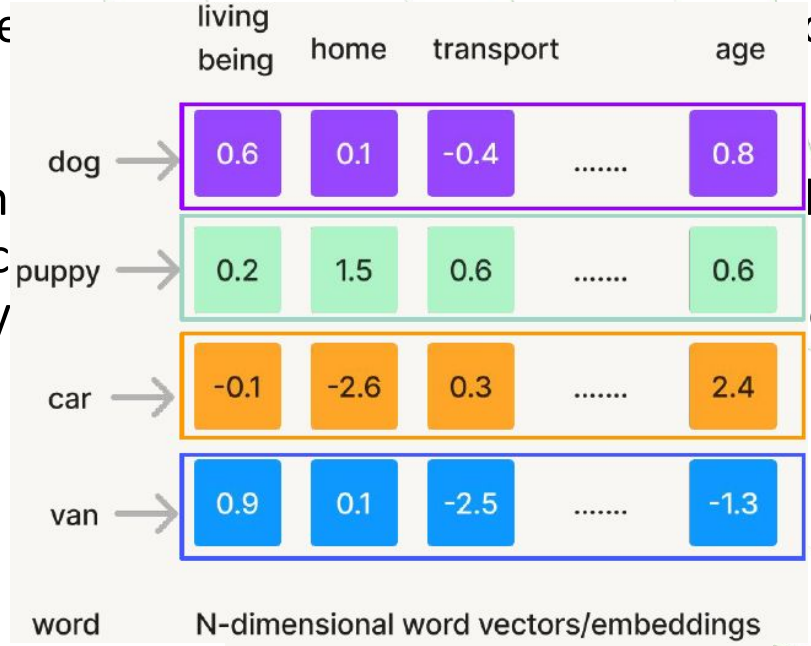
Input embedding

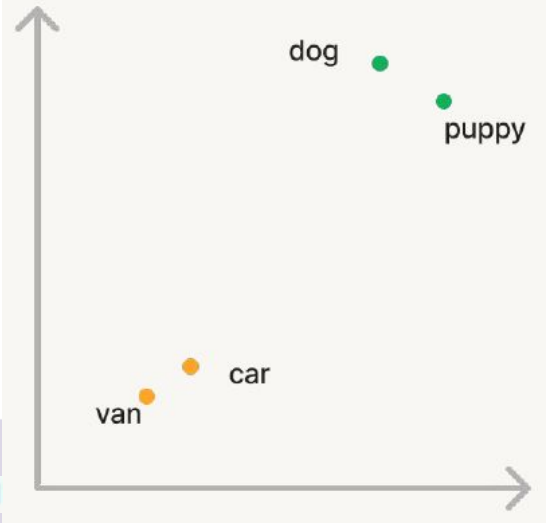




Input embedding

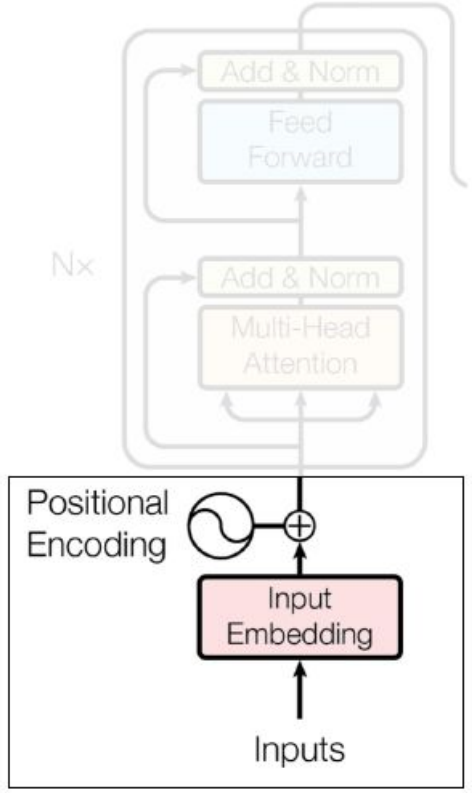
- Embedding: a representation of a symbol (word, character, se onal space of continuous-valued vectors
- The tokens are projected in a continuous euclidean space
- Correlations among words can be visualized in the embeddin l embedding can push two words further away or keep them c
- Ideally, an embedding captures the semantics of the input by ose together in the embedding space.





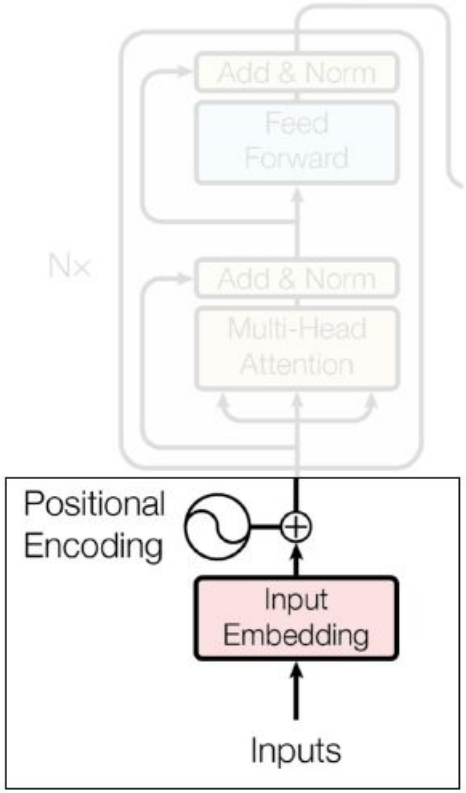
The importance of order

- Q: With the encoding we have seen so far, is it possible to discriminate between sequences that only differ in the order of the elements?
  - E.g., is it possible to differentiate between "The student is eating an apple" and "An apple is eating the student"?



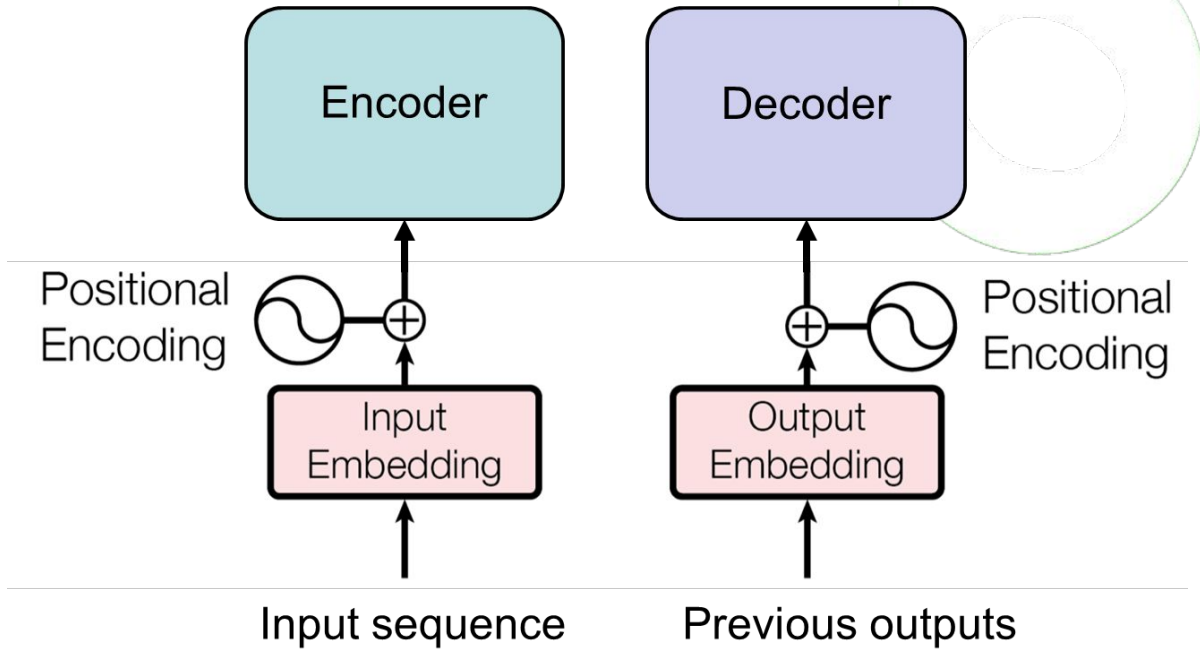
The importance of order

- Q: With the encoding we have seen so far, is it possible to discriminate between sequences that only differ in the order of the elements?
  - E.g., is it possible to differentiate between "The student is eating an apple" and "An apple is eating the student"?
- A: No, because the output of the attention module does not depend on the order of its keys/values pairs
- So how can we add the information on the order of the sequence elements?



Positional encoding

- The solution proposed by the authors of the Transformer model is to add a slight perturbation to each element of the sequence, depending on the position within the sequence
- In this way, the same element appearing in different positions would be encoded using slightly different vectors



Positional encoding

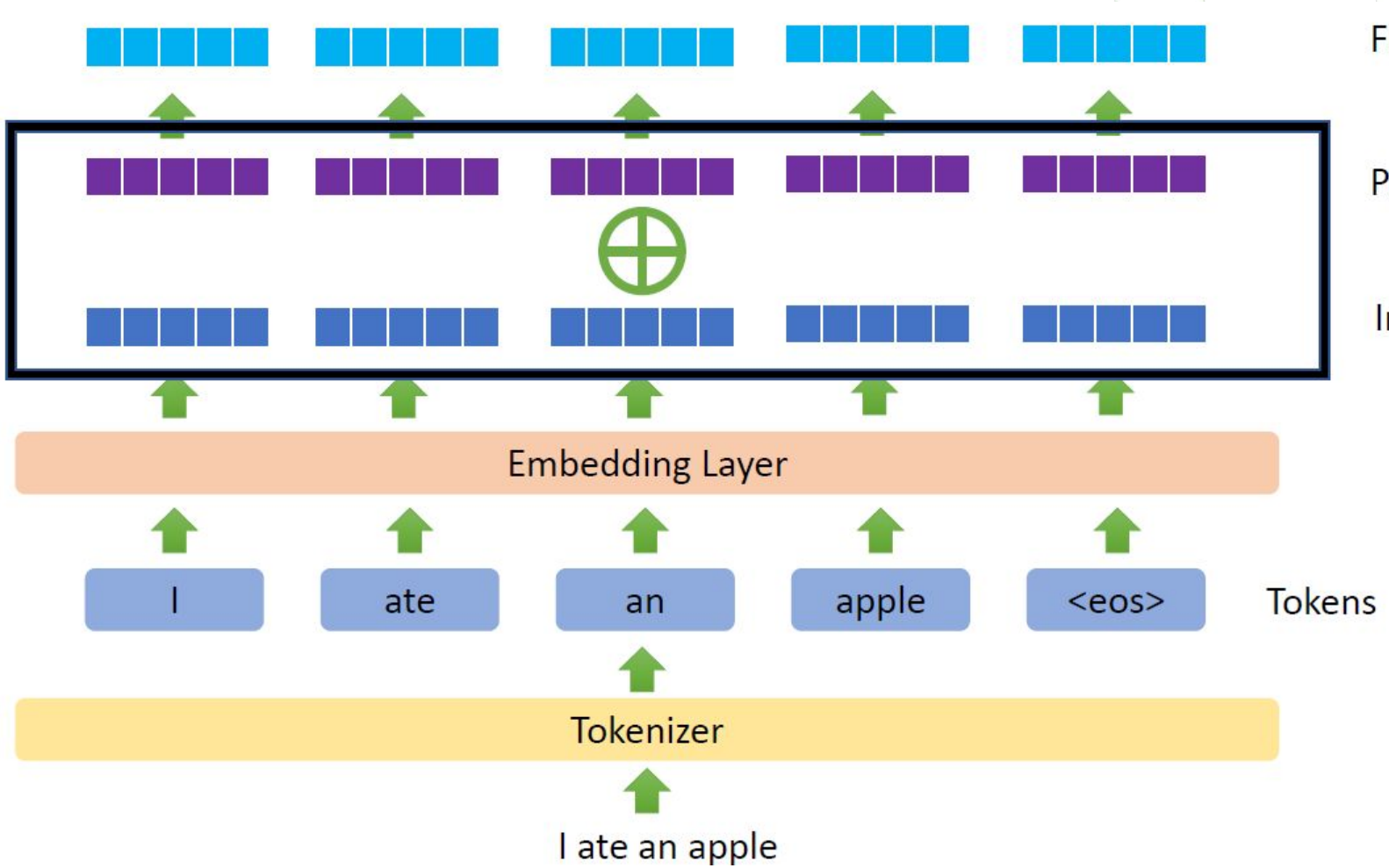
- The position encoding is represented by a set of periodic functions
- In particular, if d_{model} is the size of the embedding, and pos is the position of an element within the sequence, the perturbation to component **i** of the embedding vector representing the element is:

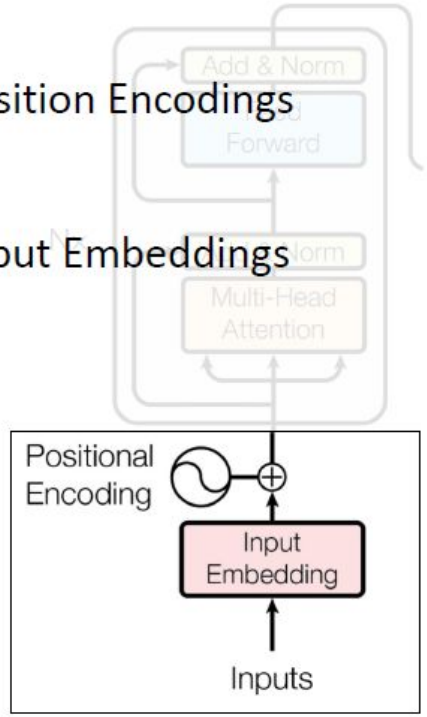
- The positional encoding is a vector with the same dimension as the input embedding, so it can be added on the input directly.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

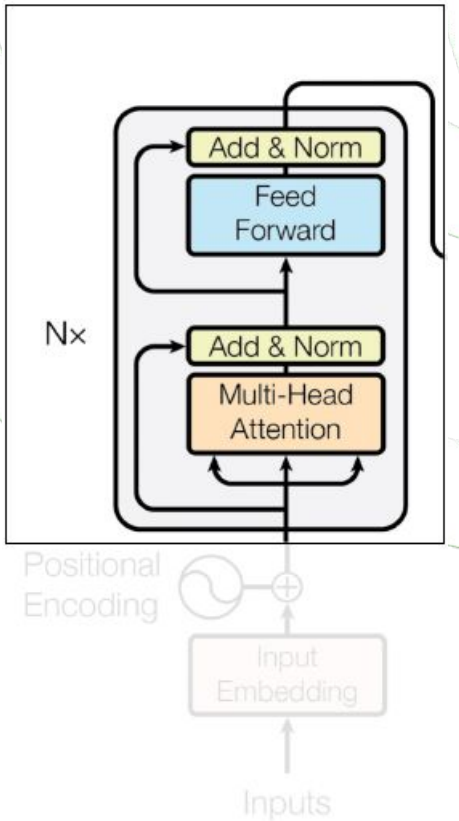
Positional encoding



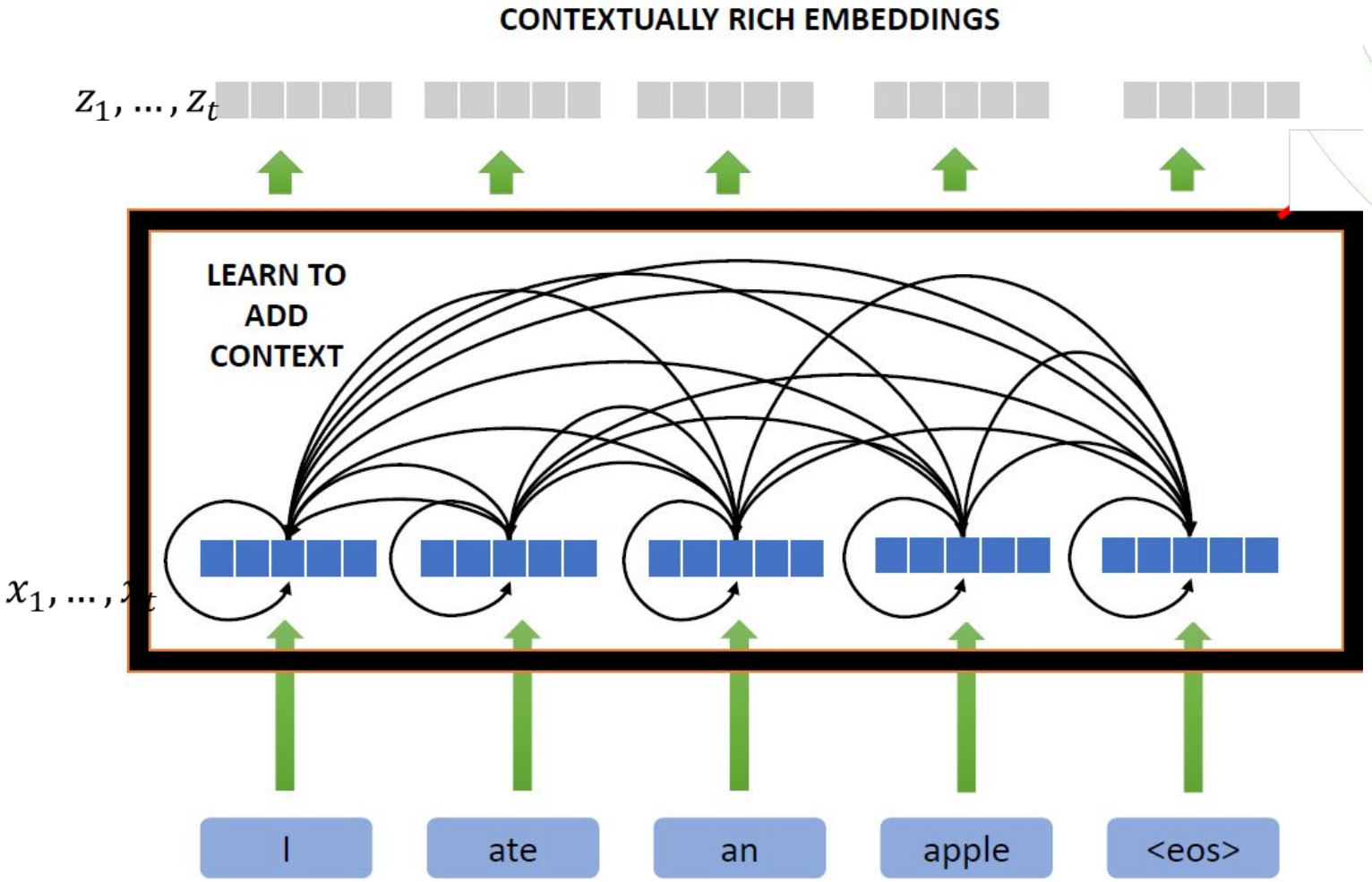


Encoder

- The **Encoder** transforms an input sequence of vectors $x_1, \dots, x_t$ into an intermediate representation of the same length $z_1, \dots, z_t$
- The vectors $z_1, \dots, z_t$ can be generated in parallel
- Each vector z_i does not depend only on the corresponding x_i, but on the whole input sequence $x_1, \dots, x_t$

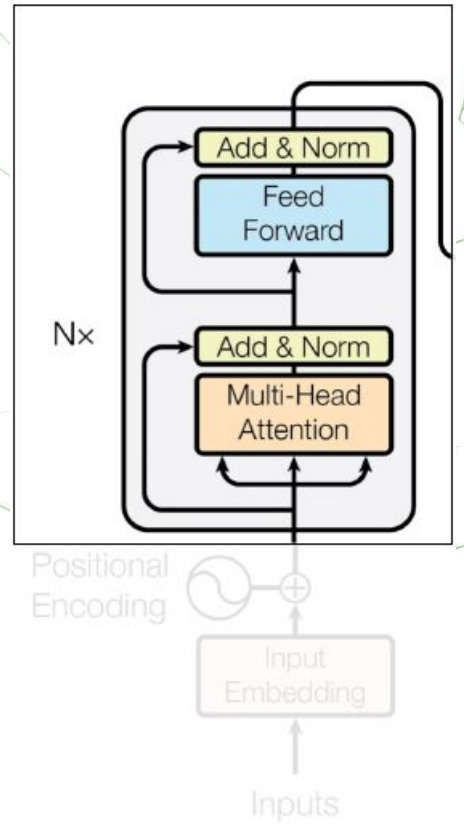


Encoder


CONTEXTUALLY RICH EMBEDDINGS
$z_1, \ldots, z_t$
LEARN TO ADD CONTEXT
$x_1, \ldots, x_t$
I
ate
an
apple
<eos>

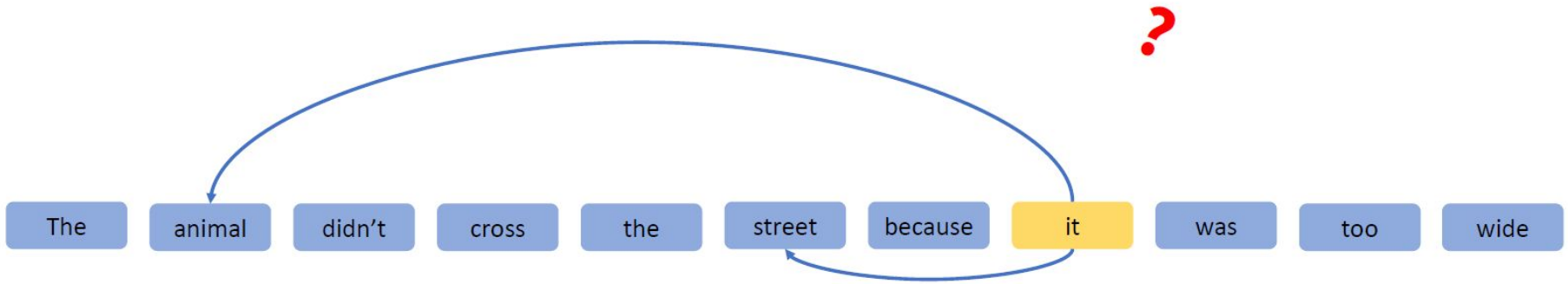
Encoder

- The encoder is made of a sequence of **encoder blocks** having the same structure
  - The original paper used 6 encoder blocks
- Each encoder block processes a sequence using a combination of the following mechanisms
  - Self-attention: a (multi-headed) attention module where the same vectors are used as Q, K and V
  - A classical feed-forward layer applied separately to each element of the sequence
  - Skip connections
  - Normalization



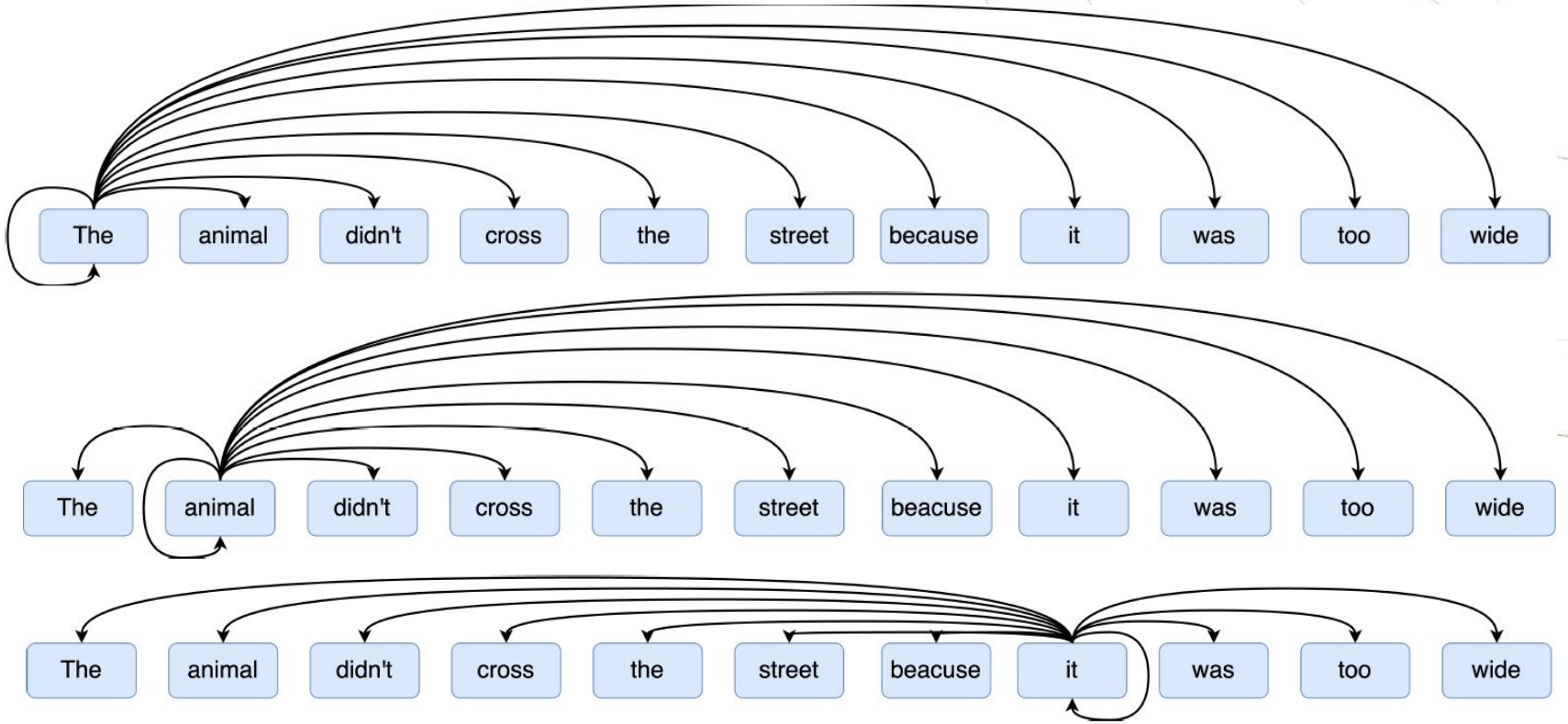
Self Attention

- Let us consider the sentence:
  - The animal didn't cross the street because it was too wide
- What does it in this sentence refer to?
- Estimating self-attention in this sentence means to find the words that one must consider first to find a better encoding for the word it
- Self-Attention estimate must be learned according to the task we are facing



Self Attention

- How to compute the attention to give to each input element when encoding the current word?



Attention

- In order to understand the self attention, we must first introduce its fundamental building block: the attention function
- Informally, an attention function is used when the value to be computed (in this case the embedding of a token in a certain position considering the context of the sentence) depends on a set of other values (in this case other tokens of the sentence), and we want to give each time a different weight (i.e. a different "level of attention") to each of the values (how much each token is important to encode the current token?)
- The attention function depends on three elements, with a terminology inherited from document retrieval: query, key, value

Attention

- We have an input value q and we want to define some target function $f_T(q)$
- q is called the **query** value in the attention terminology
- In the general case, both q and $f_T(q)$ can be **vectors**
- We want to express $f_T(q)$ as a function of a given set of elements $v_1, \dots, v_n$
- We want the "attention" given to each v_i to be different depending on q
- We assume that for each v_i we have available an additional information k_i that can be used to decide the "attention" to be given to v_i
- The elements v_i are called **values** and the k_i are called **keys** in the attention terminology; both the values and the keys can be vectors

Attention

- A commonly adopted formulation of the problem is to define the target function as:

$$f_T(q) = \alpha(q, k_1) \cdot f_V(v_1) + \cdots + \alpha(q, k_n) \cdot f_V(v_n) =$$

$$= \sum_{i=1}^{n} \alpha(q, k_i) \cdot f_V(v_i)$$

Attention given to value v_i

Attention

- A commonly adopted formulation of the problem is to define the target function as:

$$f_T(q) = \sum_{i=1}^{n} \alpha(q, k_i) \cdot f_V(v_i)$$

- α is our **attention function**
  - We want α and f_V to be **learned** by our system
  - Tipically, $\alpha(q, k_i) \in [0,1]$ and $\sum_i \alpha(q, k_i) = 1$
- **Note**: the value of the target function **does not depend on the order** of the key-value pairs (k_i, v_i)

Self Attention

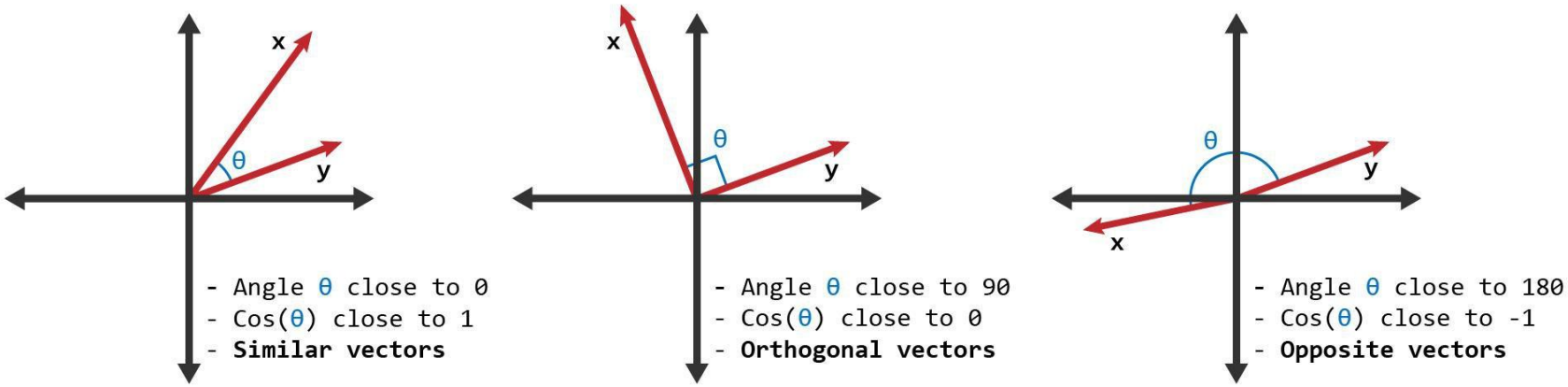
- The Transformer architecture uses a particular definition of the attention function, based on linear vector/matrix operations and the softmax function
- This definition is
  - Differentiable, so it can be learned using Back Propagation
  - Efficient to compute
  - Easy to parallelize, since the attention for several query vectors can be efficiently computed in parallel at the same time

Self Attention

- Input: three matrices Q, K, V
- Q (m x d_q) contains the query vectors (each row is a query)
- K (n x d_k) contains the key vectors (each row is a key)
- V (n x d_v) contains the value vectors (each row is a value)
  - K and V must have the same number of rows

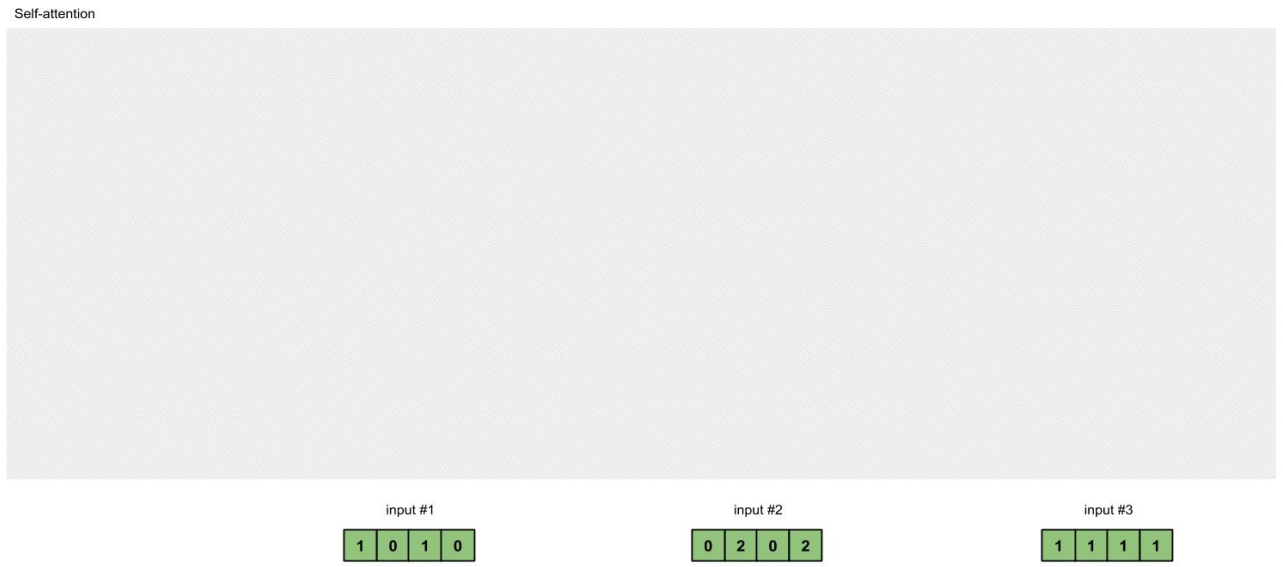
Self Attention

- If the query and the key are represented by vectors with the same dimensionality, a matching score can be provided by the scaled dot product of the two vectors (cosine similarity)



Self Attention

- Step 0: Each element in the sequence is represented by a numerical vector

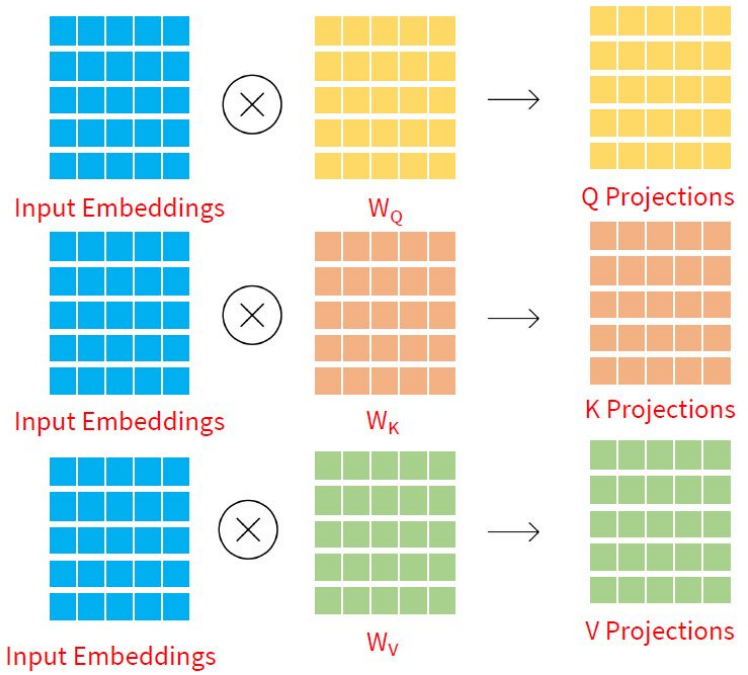


Self Attention

- Step 1: the input matrices are "projected" onto a different subspa nn dot product) by weight matrices

  - $Q' = Q \cdot W_Q$
  - $K' = K \cdot W_K$
  - $V' = V \cdot W_V$

Note: W_Q and W_K **must** have the same number of columns

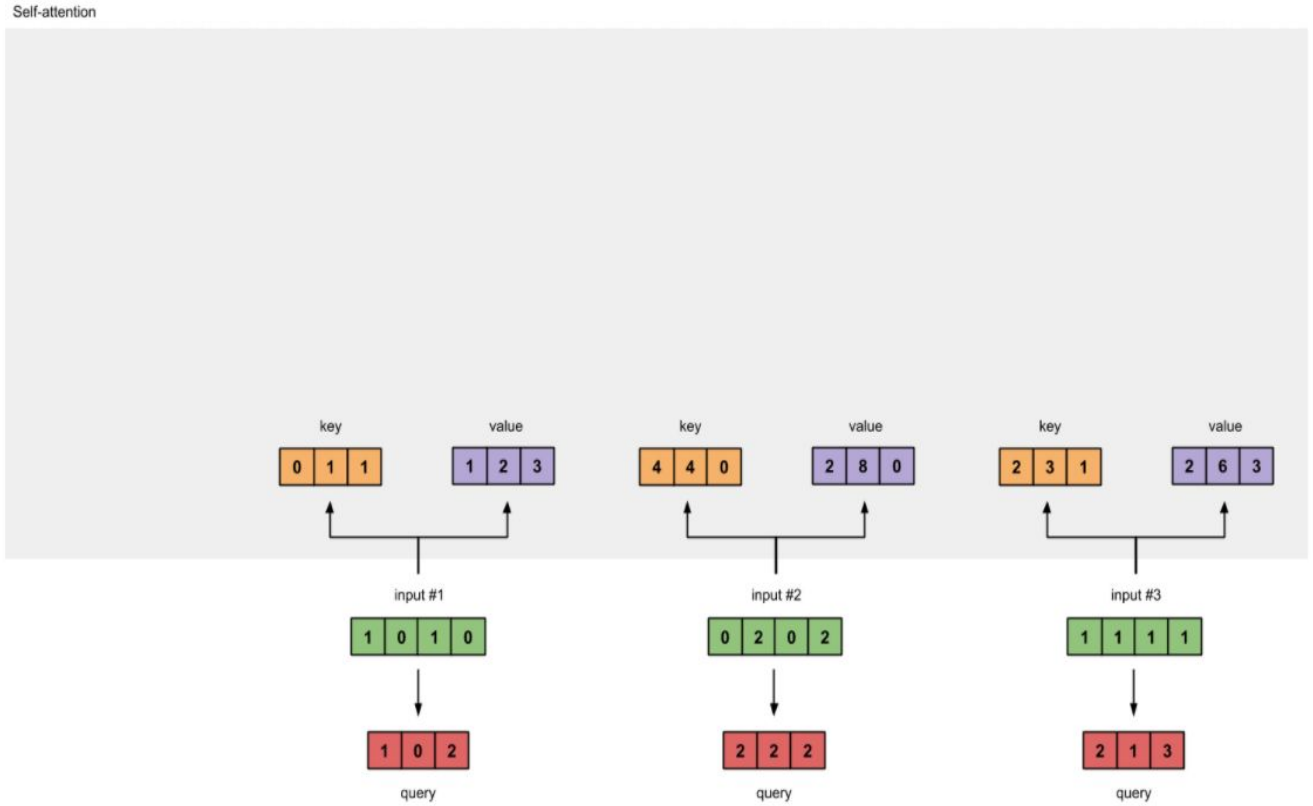


These are the trainable weights:

- W_Q (d_q x d'_q)
- W_K (d_k x d'_k)
- W_V (d_v x d'_v)

Self Attention

- Step 1: Compute a key (K), a value (V) and a query (Q) as linear function of each element in the sequence.



Self Attention

- Step 2: the attention matrix A is computed for each position by multiplying Q' and the transpose of K', scaling by $1/\sqrt{d'_k}$ and applying softmax to each of the resulting rows

- $$A = softmax\left(\frac{Q' \cdot K'^T}{\sqrt{d'_k}}\right)$$

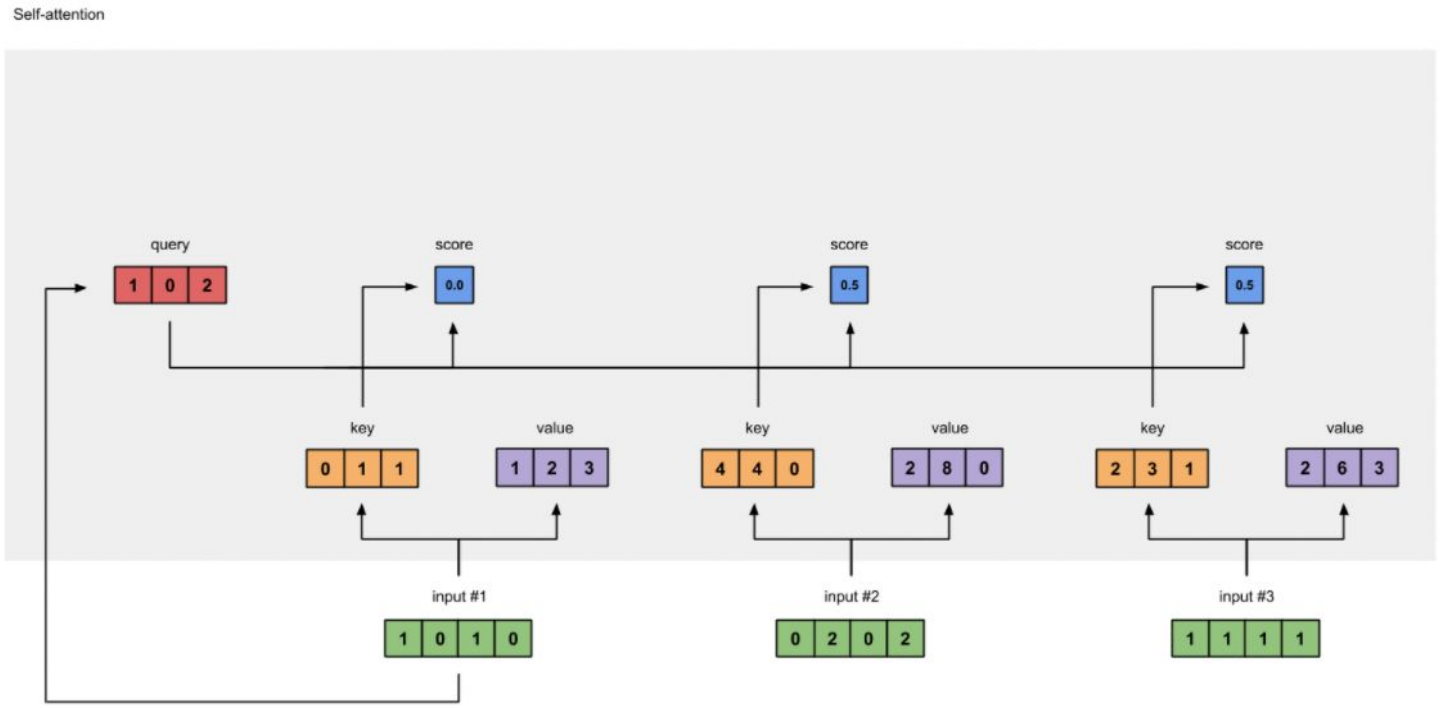
This scaling is used to avoid that the argument of softmax becomes too large with the increase of the dimension d'_k

Softmax is applied to each row separately

A is a (m x n) matrix whose element $a_{ij} = \alpha(q_i, k_j)$

Self Attention

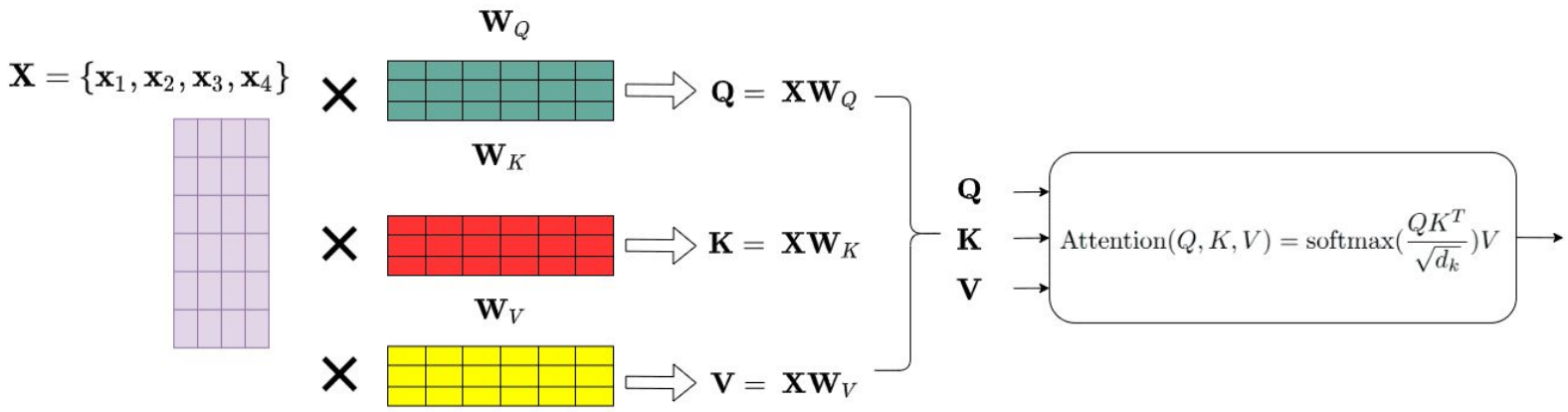
- Step 2: Compute attention score for each position i as a softmax of the scaled dot product of all the keys (*bidirectional self-attention*) with Q*i*



Self Attention

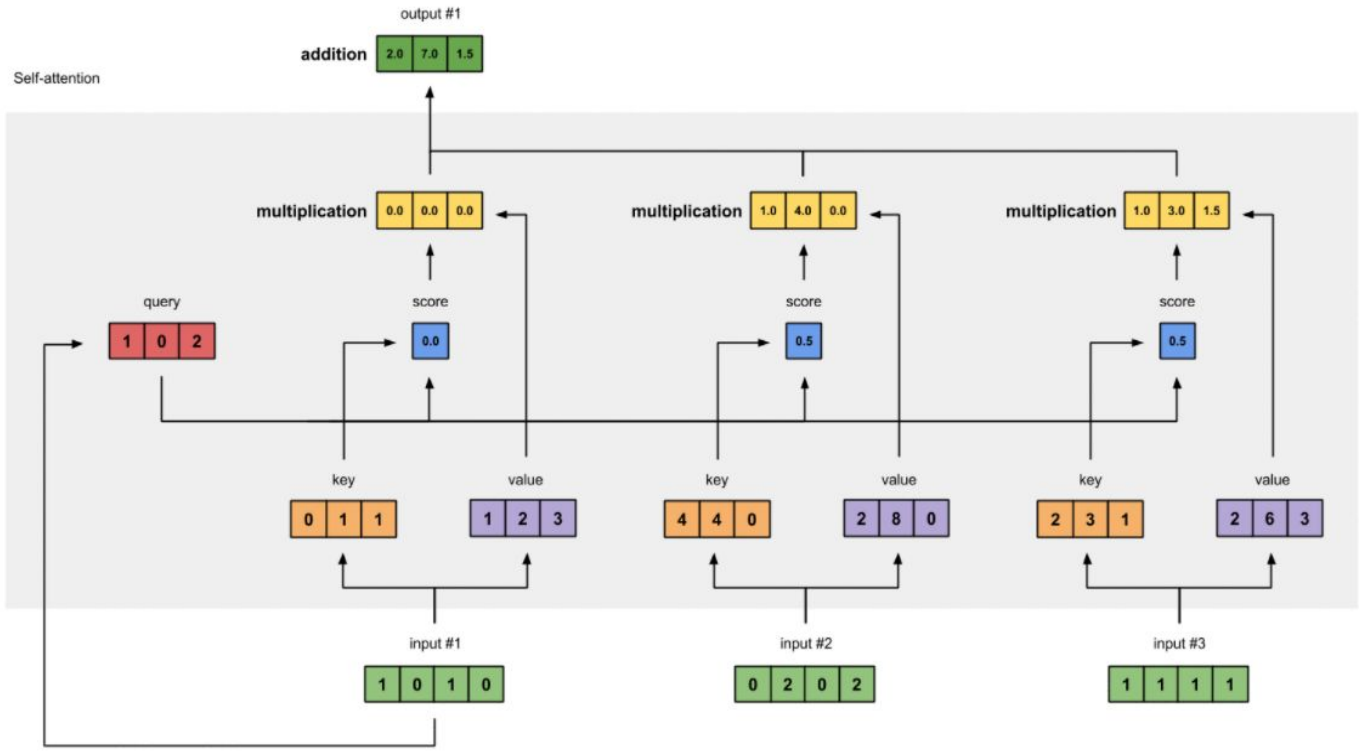
- Final step: the target value is computed by row-by-column multiplication between A and V'
  - $f_T(Q) = A \cdot V'$

The result is a m x d'$_v$ matrix representing the target function computed on the m queries in the input matrix Q.

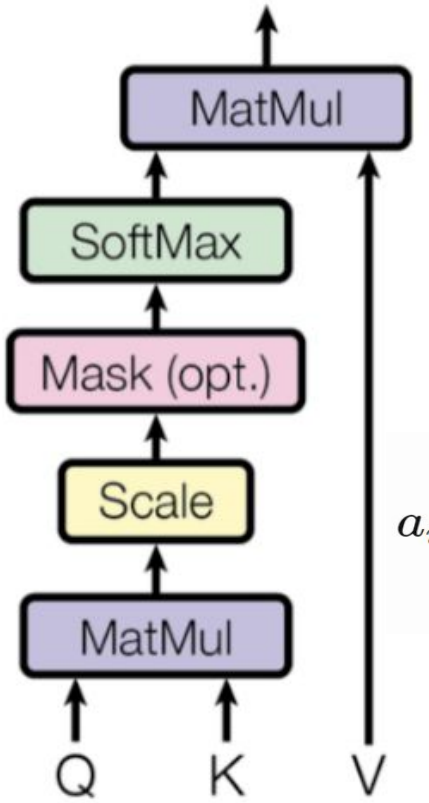


Self Attention

- Step 3: Output representation for each position I, as a weighted sum of values (each one multiplied by the related attention score)



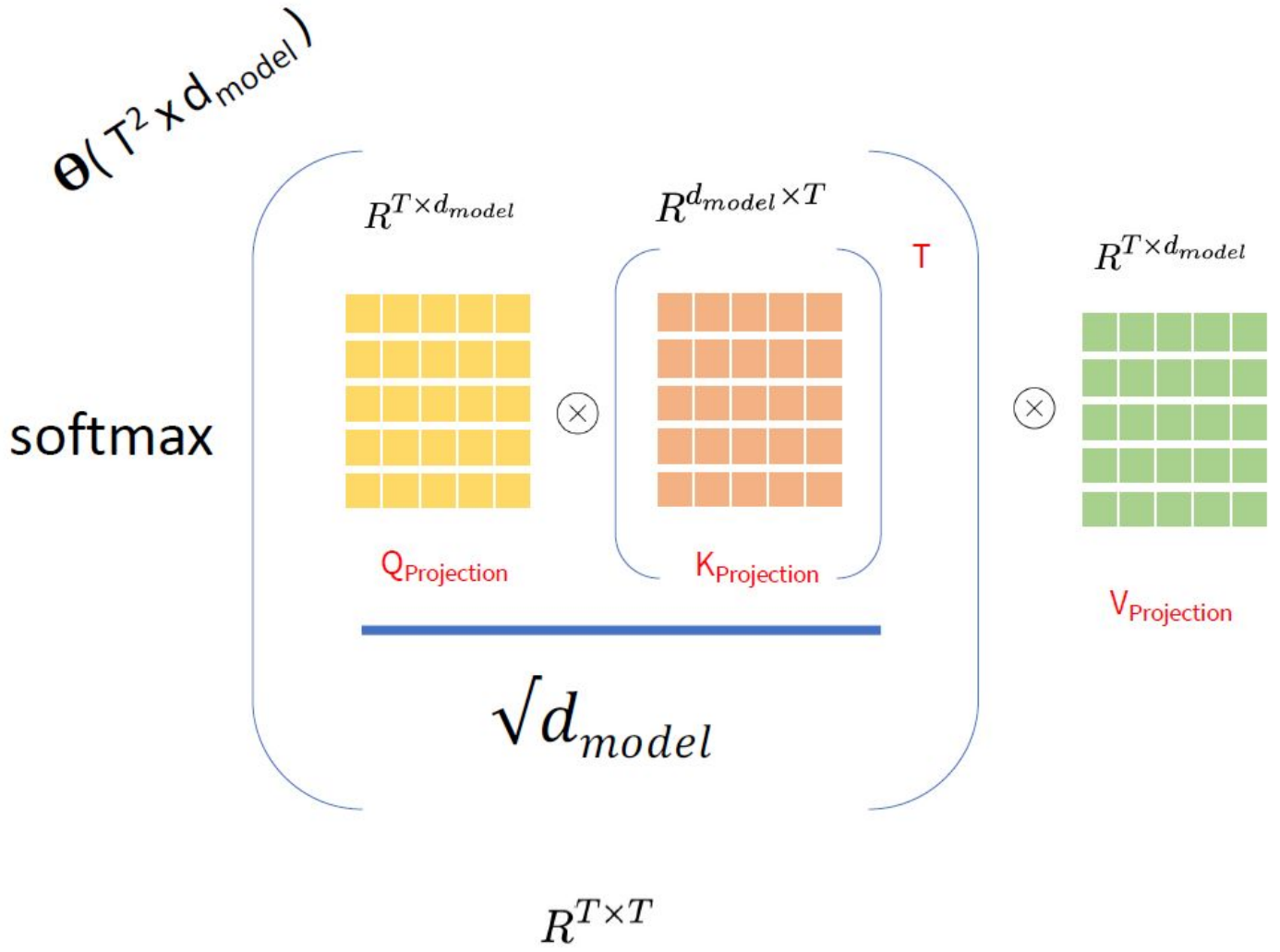
Self Attention



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}})\mathbf{V}$$

$$a_{ij} = \text{softmax}(\frac{\mathbf{q}_i {\mathbf{k}_j}^\top}{\sqrt{d_k}}) = \frac{\exp(\mathbf{q}_i {\mathbf{k}_j}^\top)}{\sqrt{d_k} \sum_{r \in S_i} \exp(\mathbf{q}_i {\mathbf{k}_r}^\top)}$$

Self Attention

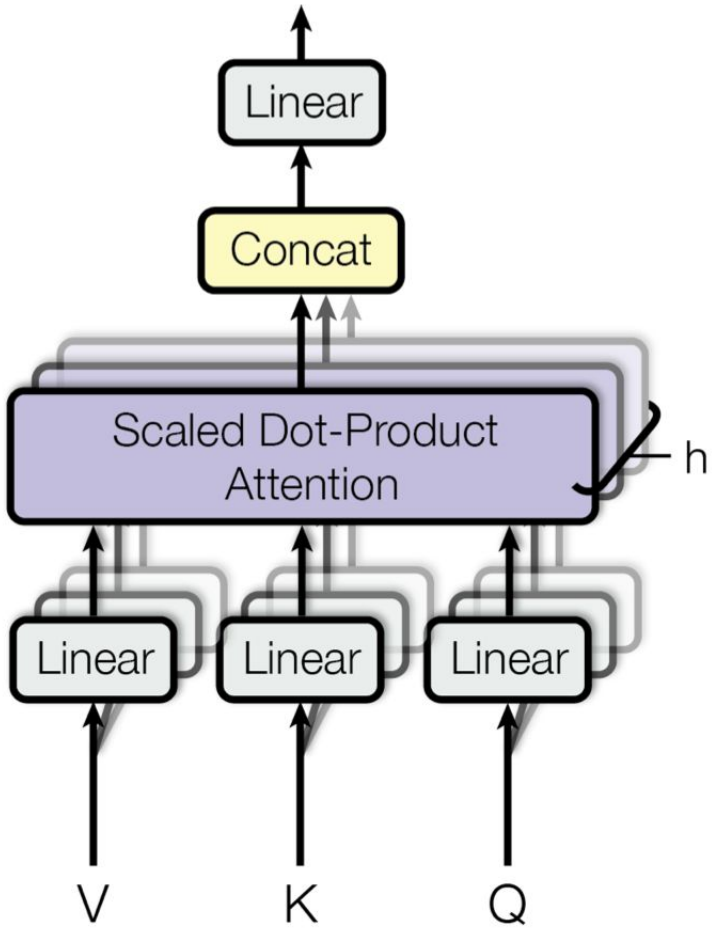


Outline

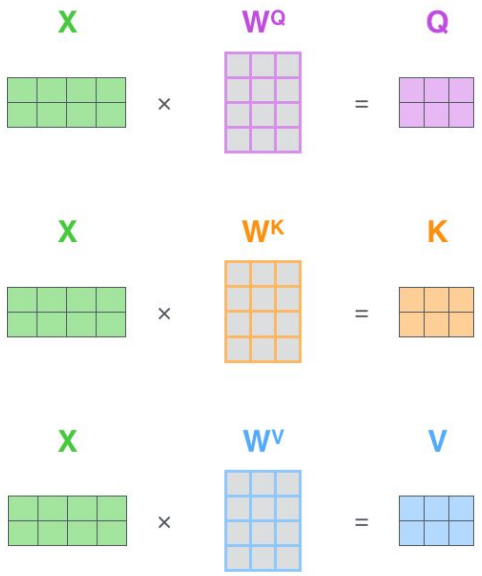
- Multi-Head Attention
- Encoder Output
- Decoder
- Masked Multi-Head Attention
- Encoder-Decoder Attention
- Output
- Transformer’s pipeline

Multi-head Attention

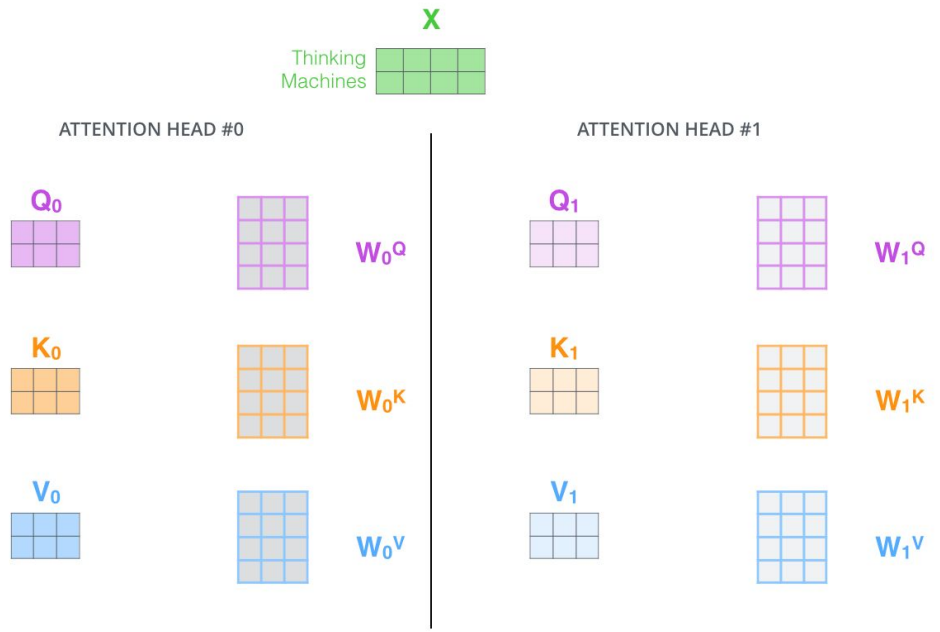
- By using different self attention heads it is possible to encode different meanings of the context:
  - Several scaled-dot product attention computations are performed in parallel (using different weight matrices)
  - The results are concatenated row-by-row forming a larger matrix (with the same number of rows m)
  - This matrix is finally multiplied by a final weight matrix
  - This scheme is called **multi-head** attention



Multi-head Attention



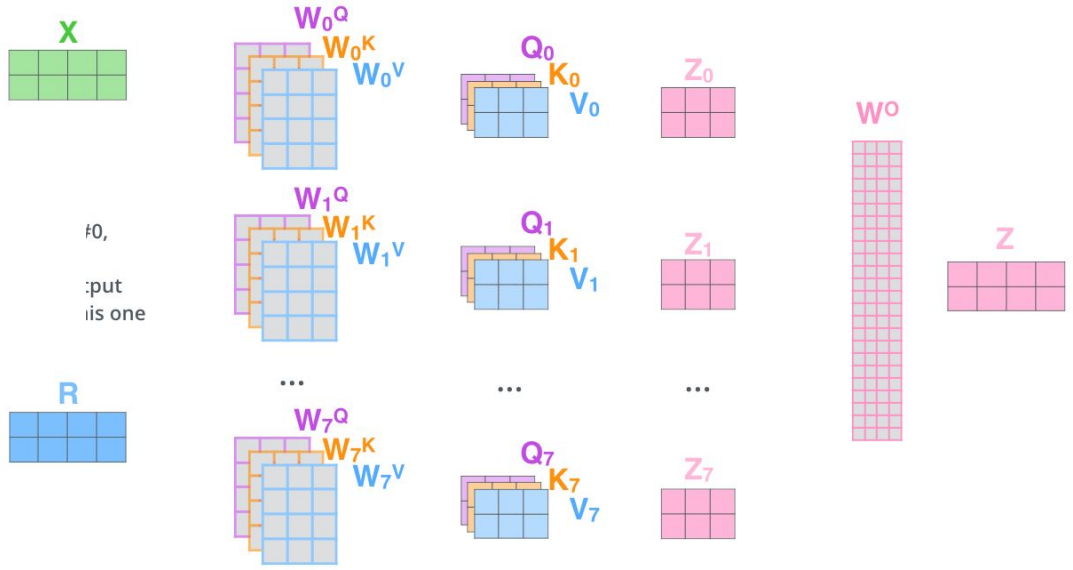
one head



two heads

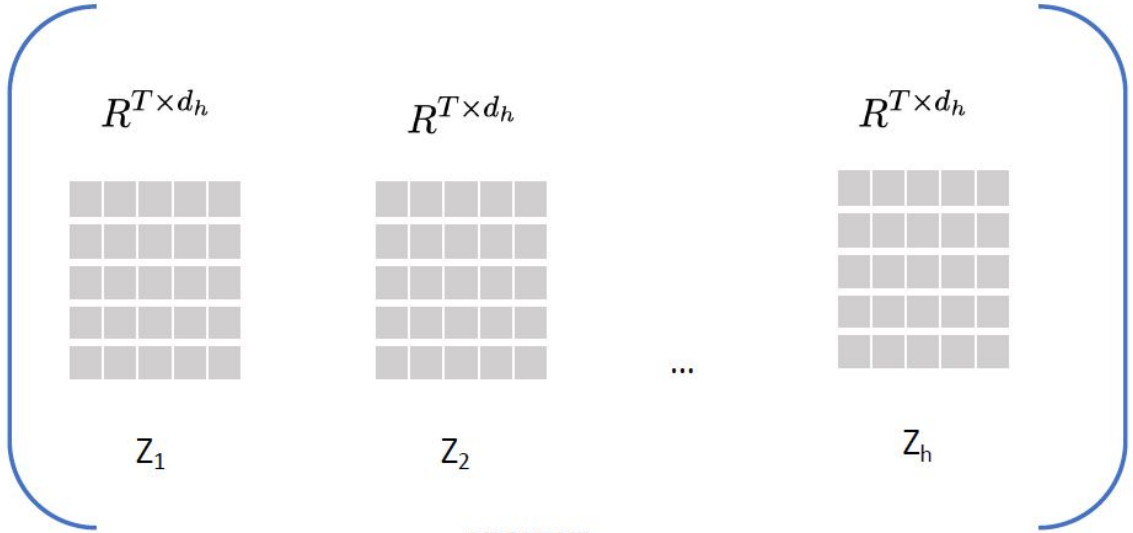
Multi-head Attention

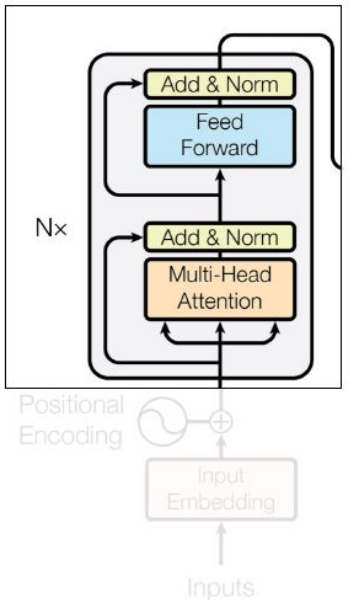
- The outputs of the heads are concatenated and then are multiplied by an additional weight matrix to combine several representations at the same network level.



*Multi-head attention allows the model to **jointly** attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this possibility.*

Multi-head Attention



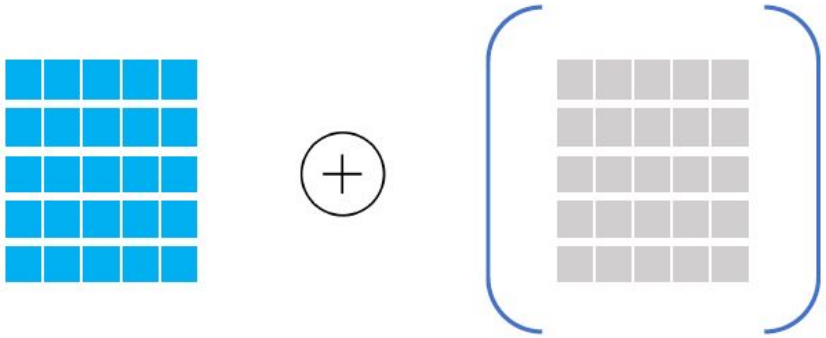


Multi Head Attention : Z

$$d_h = \frac{d_{model}}{h}$$

$$R^{T \times d_{model}}$$

Add & Norm



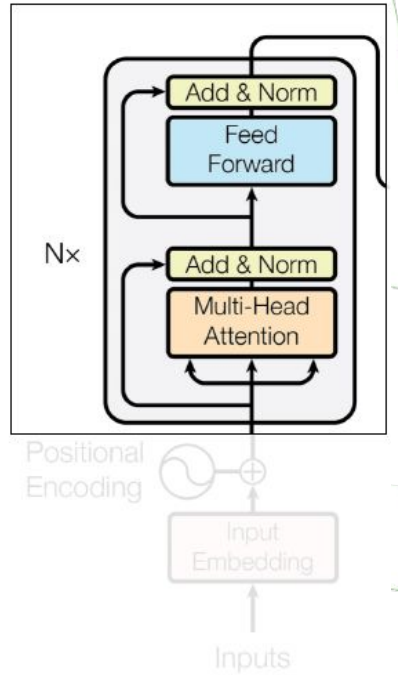
Input Norm(Z)

Normalization(Z)

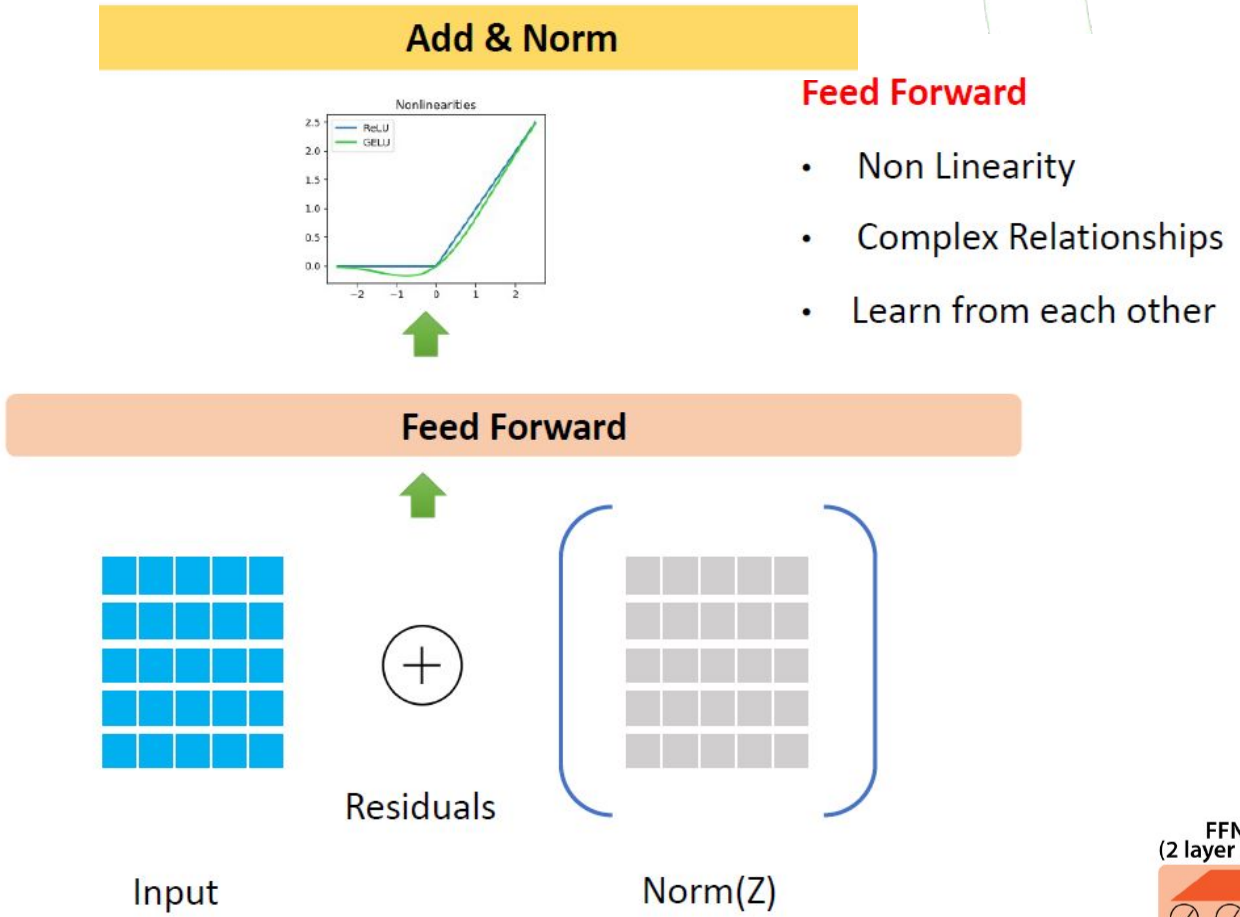
- Mean 0, Std dev 1
- Stabilizes training
- Regularization effect

Add -> Residuals

- Avoid vanishing gradients
- Train deeper networks

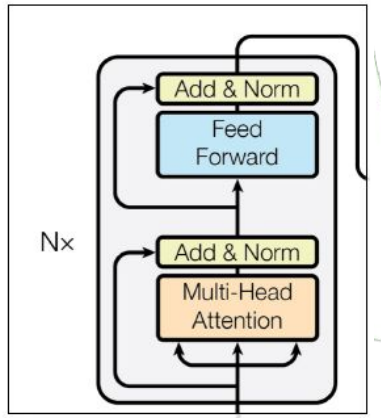


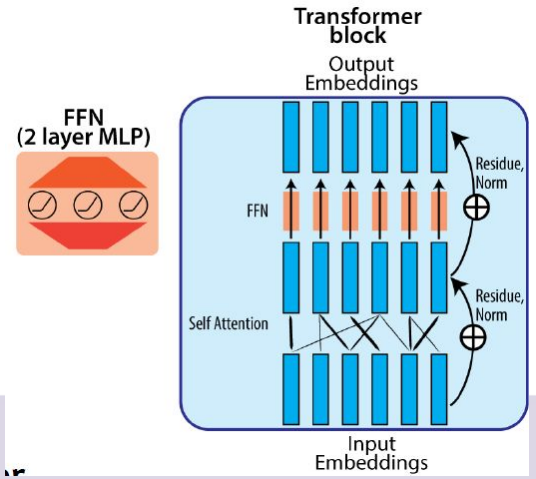
Feed Forward



Feed Forward

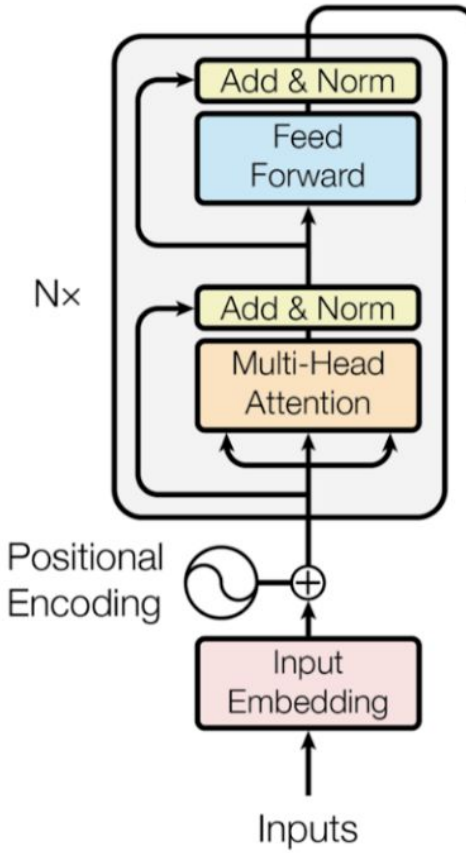
- Non Linearity
- Complex Relationships
- Learn from each other





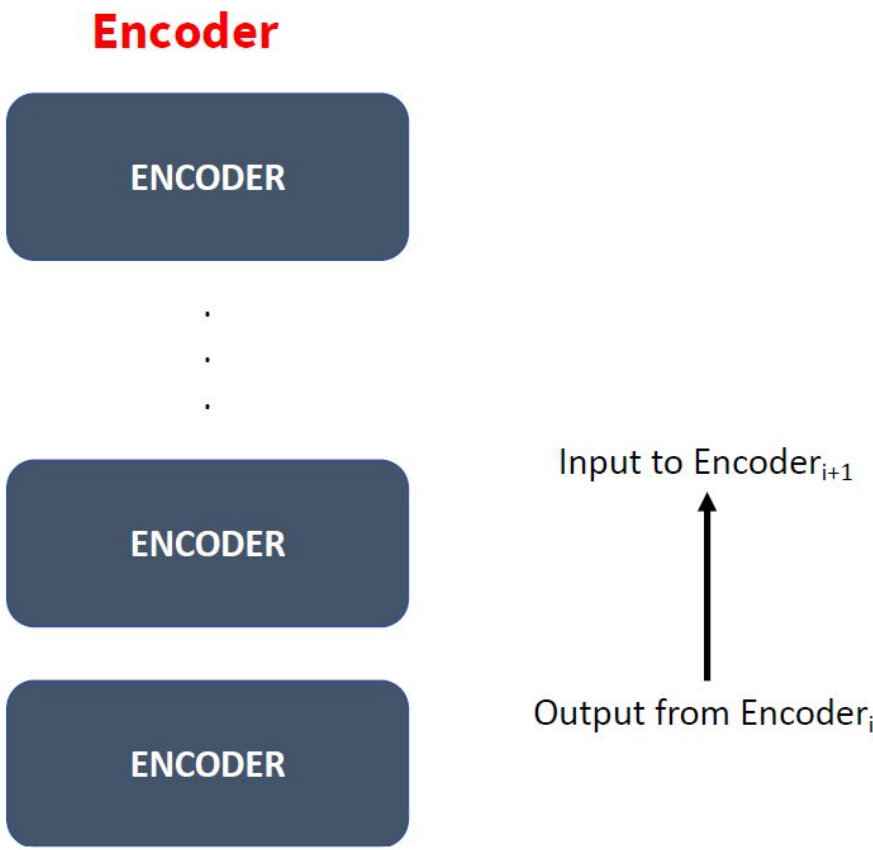
Transformer's Encoder

- Used for computing a representation of the input sequence
- Uses an additive positional encoding to deal with the order-agnostic na on
- Uses residual connection to foster the gradients flow
- Adopts normalization layers to stabilize the network training
- Position-Wise Feed-Forward layer to add non-linearity
  - Applied to each sequence element independently



Transformer's Encoder

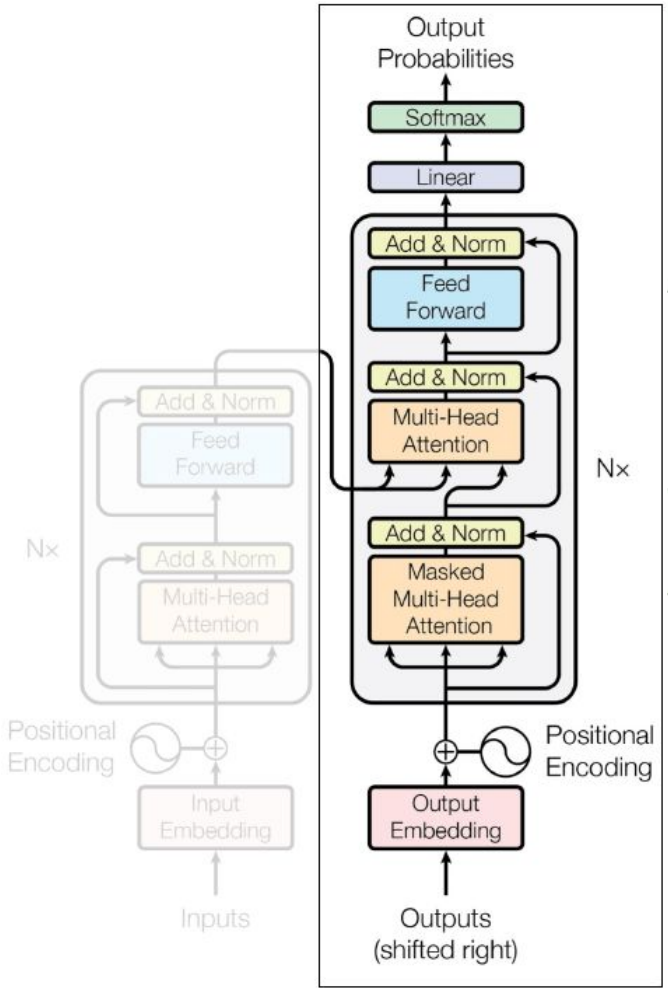
- Since each encoder produces an output whose dimensionality is the same of the input, it is possible to stack an arbitrary number of encoder's blocks.
- The output of the first block is fed to the second block (no word embeddings) and so on.



Decoder

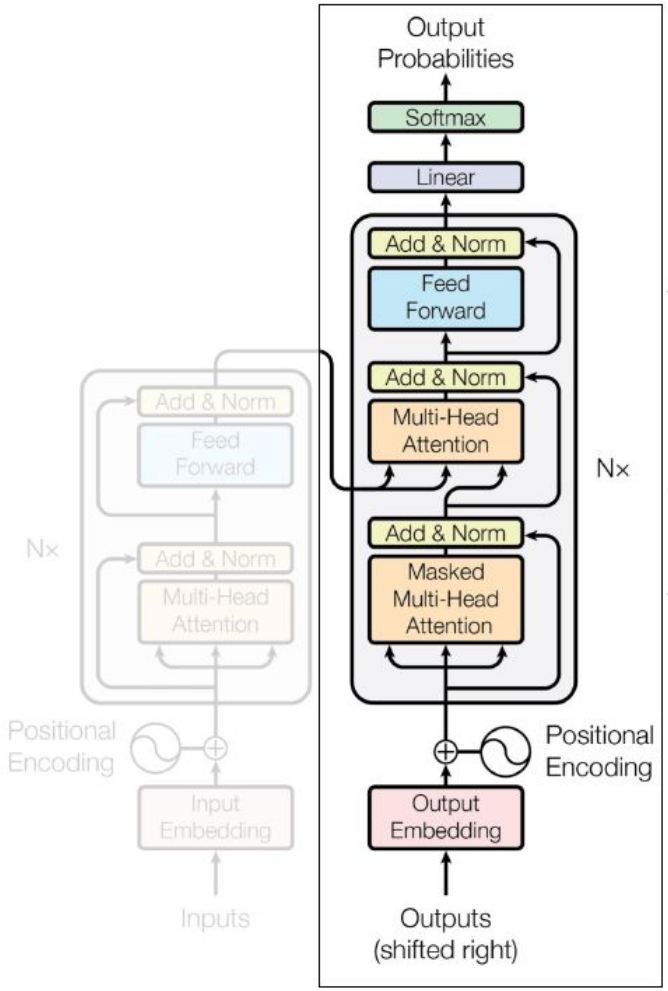
- The **Decoder** uses the information contained in the intermediate representation $z_1, \dots, z_t$ to generate the output sequence $y_1, \dots, y_m$

- The Decoder works **sequentially**; at each step the decoder uses $z_1, \dots, z_t$ and $y_{1,\dots,}y_{i-1}$ to generate y_i

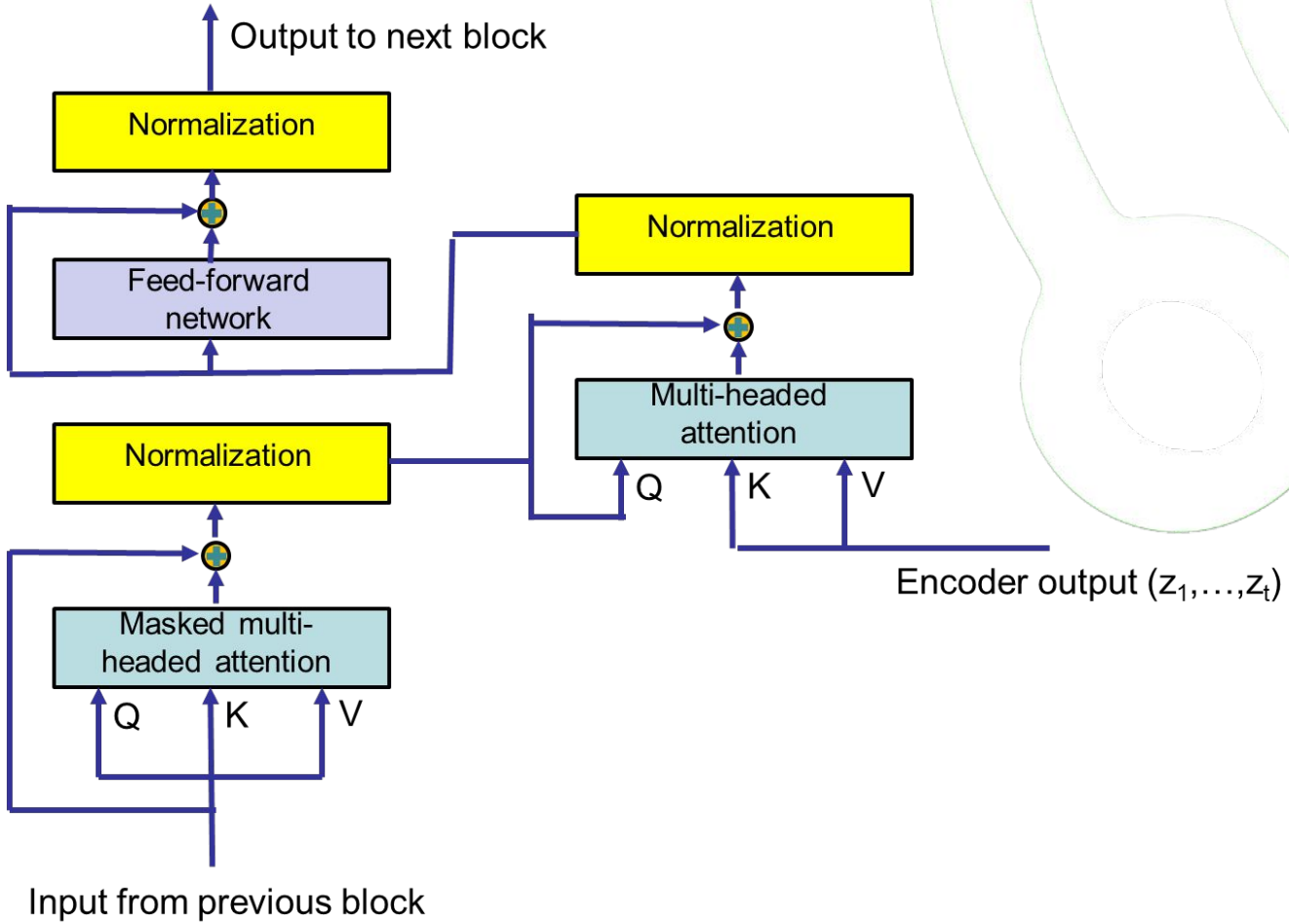


Decoder

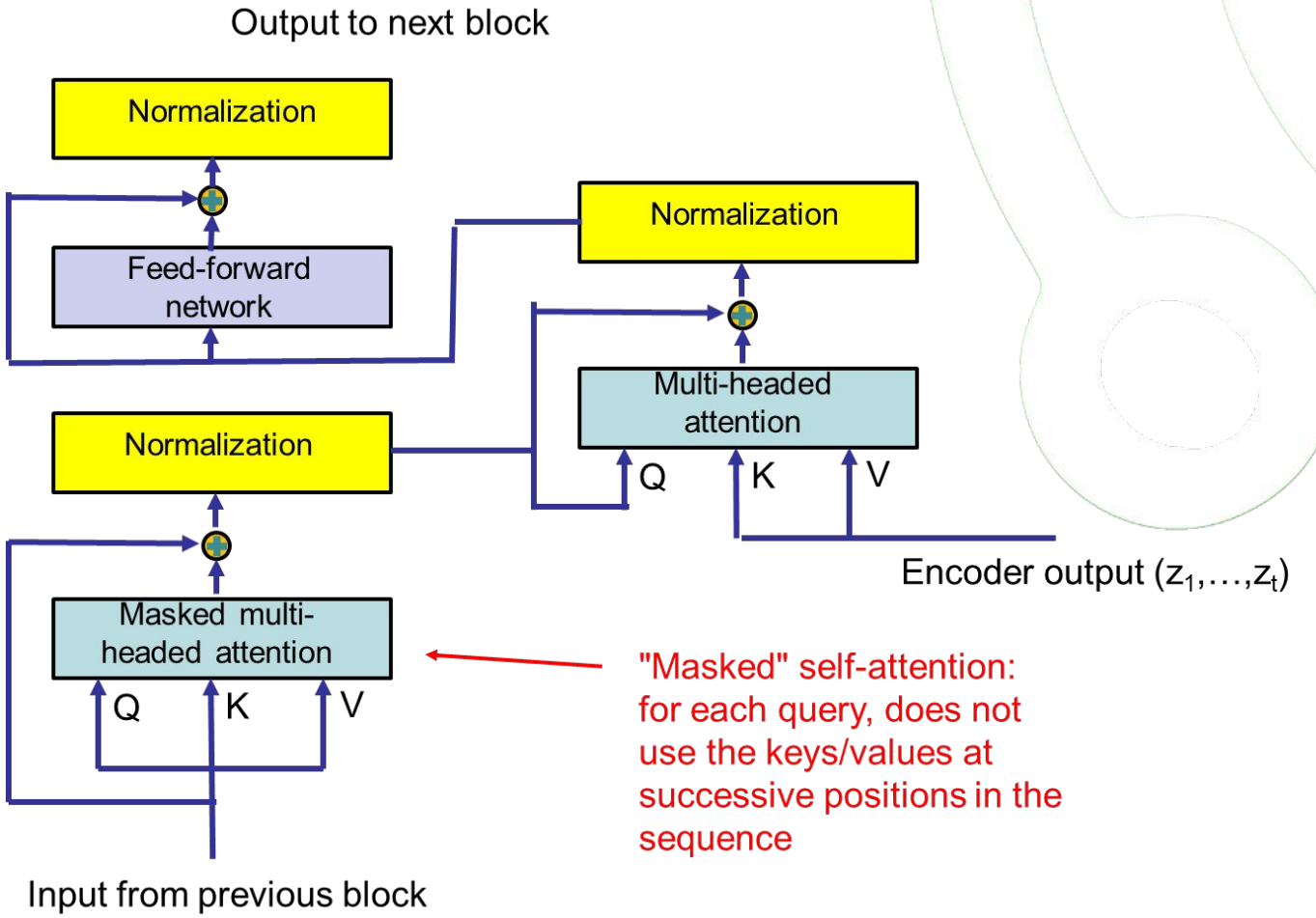
- The decoder is made of a sequence of **decoder blocks** having the same structure
  - The original paper used 6 decoder blocks
- The decoder blocks, in addition to the same modules used in the encoder block, add an attention module where the keys and values are taken from the encoder's intermediate representation $z_1, \ldots, z_t$
  - Also, the self-attention module is slightly modified so as to ensure that the query at position i only uses the values at positions 1,...,i



Decoder

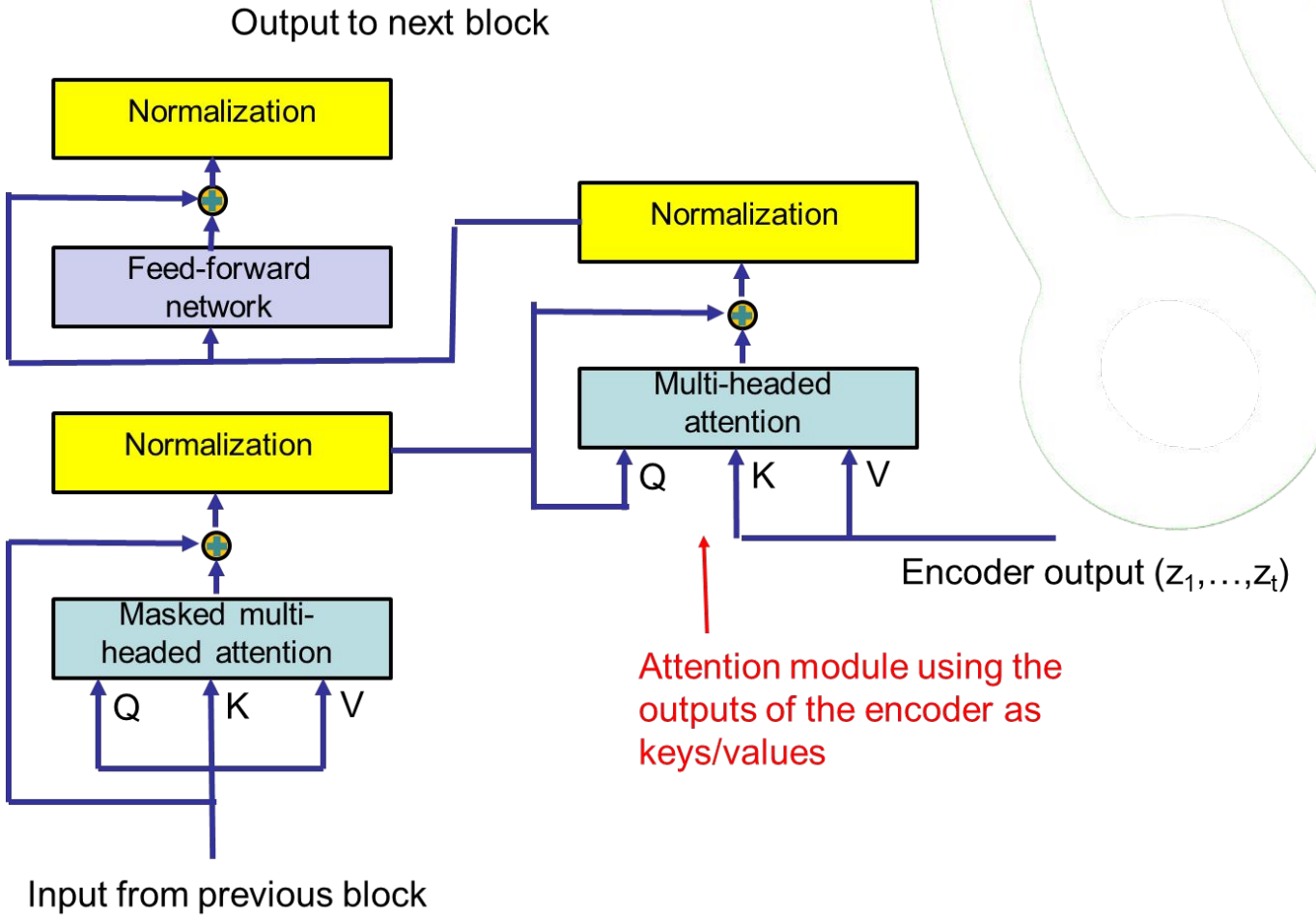

Output to next block
Normalization
Feed-forward network
Normalization
Multi-headed attention
Q
K
V
Normalization
Masked multi-headed attention
Q
K
V
Encoder output (z_1,…,z_t)
Input from previous block

Decoder



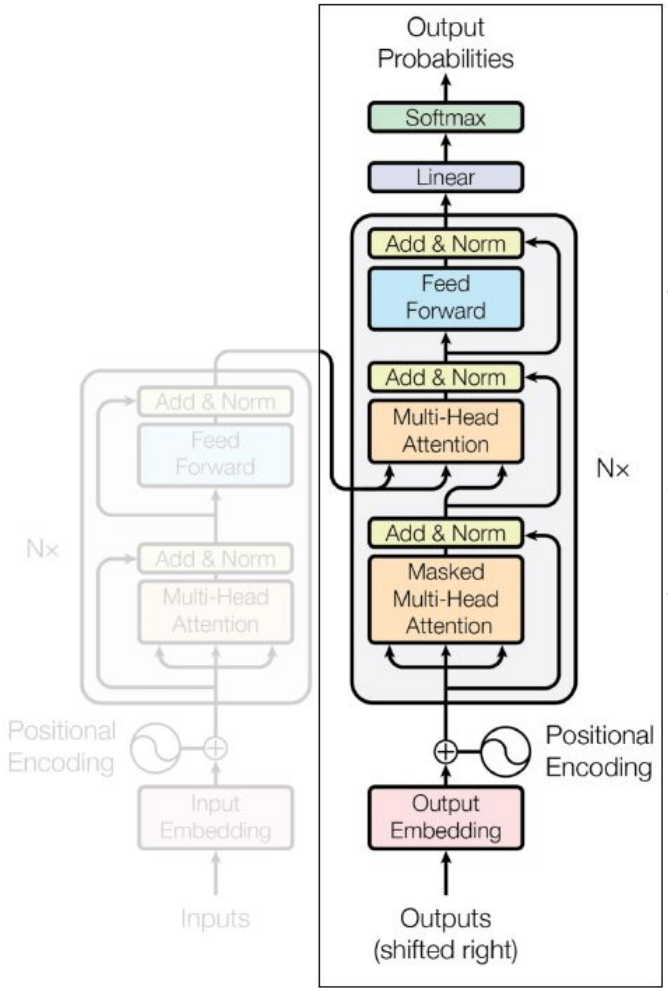
"Masked" self-attention: for each query, does not use the keys/values at successive positions in the sequence

Decoder

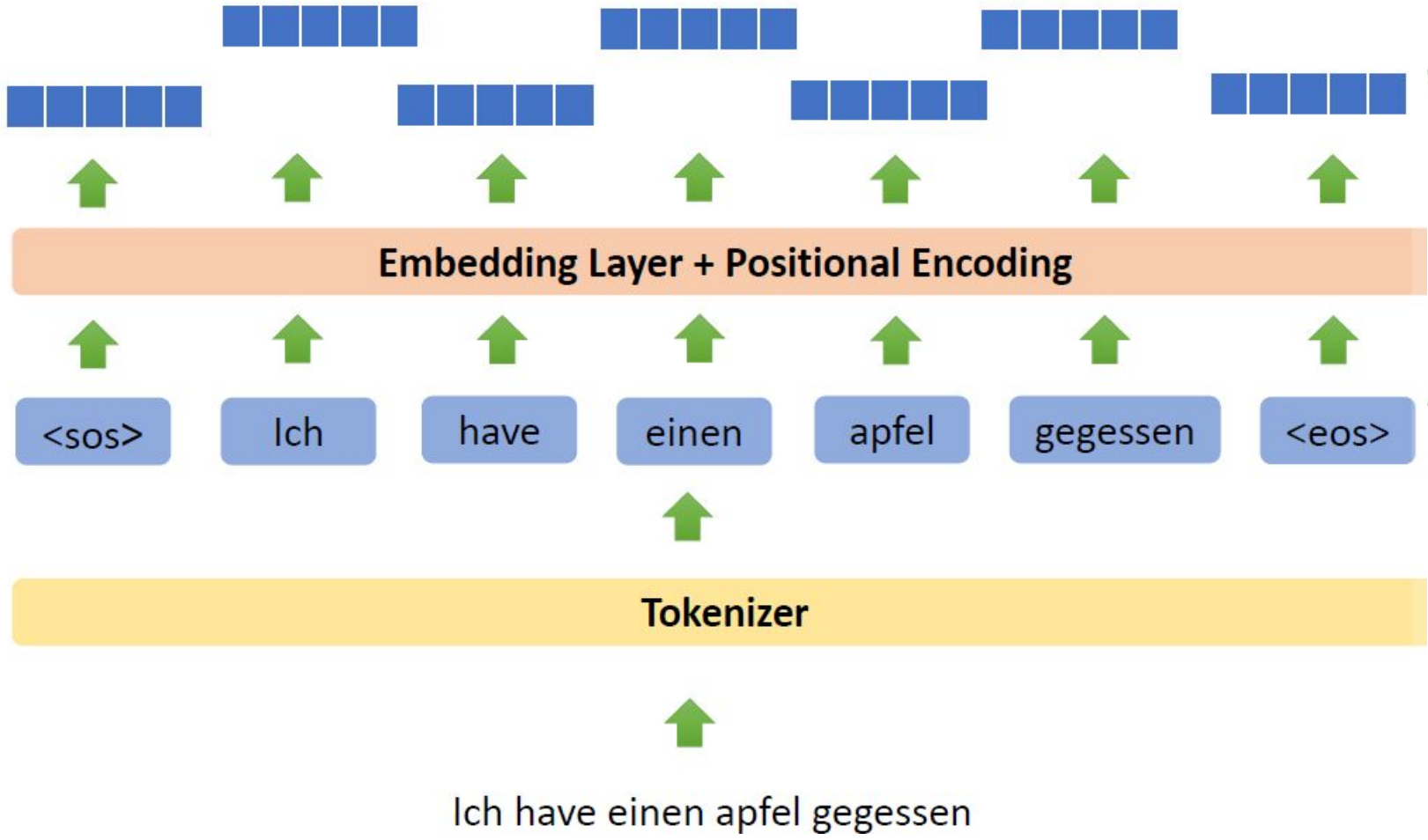

Output to next block
Normalization
Feed-forward network
Normalization
Multi-headed attention
Q
K
V
Normalization
Masked multi-headed attention
Q
K
V
Encoder output (z_1,…,z_t)
Attention module using the outputs of the encoder as keys/values
Input from previous block

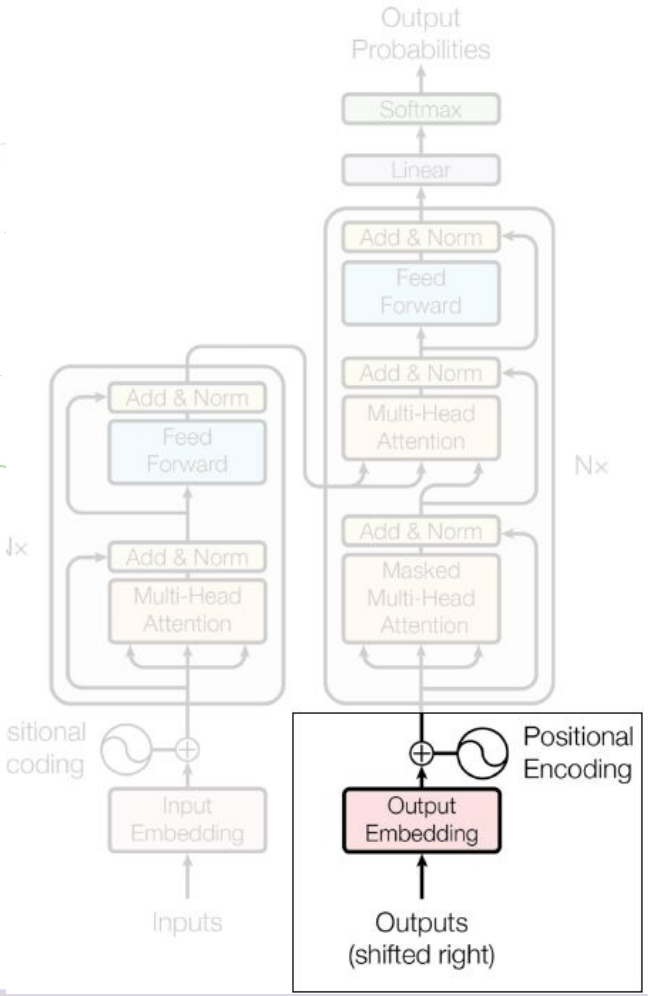
Decoder

- On top of the last decoder block, the decoder adds an additional linear layer and a softmax activation function, for computing the probability of the next output element y_i
- Thus, the last layers has a number of neurons corresponding to the cardinality of the output set

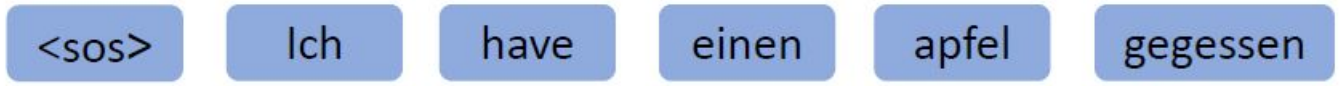


Output embedding

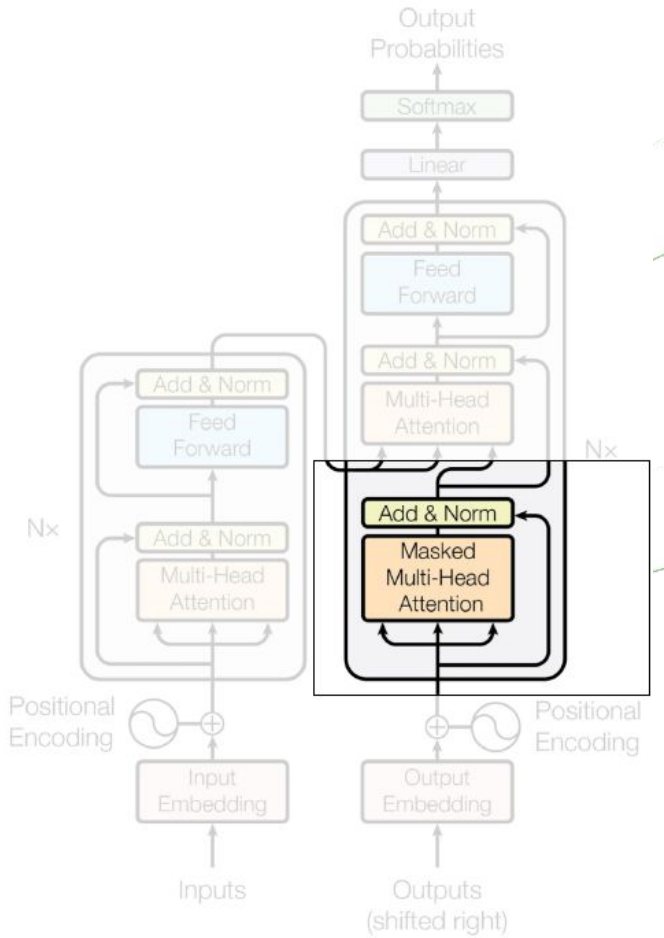




Masked Multi-Head Attention



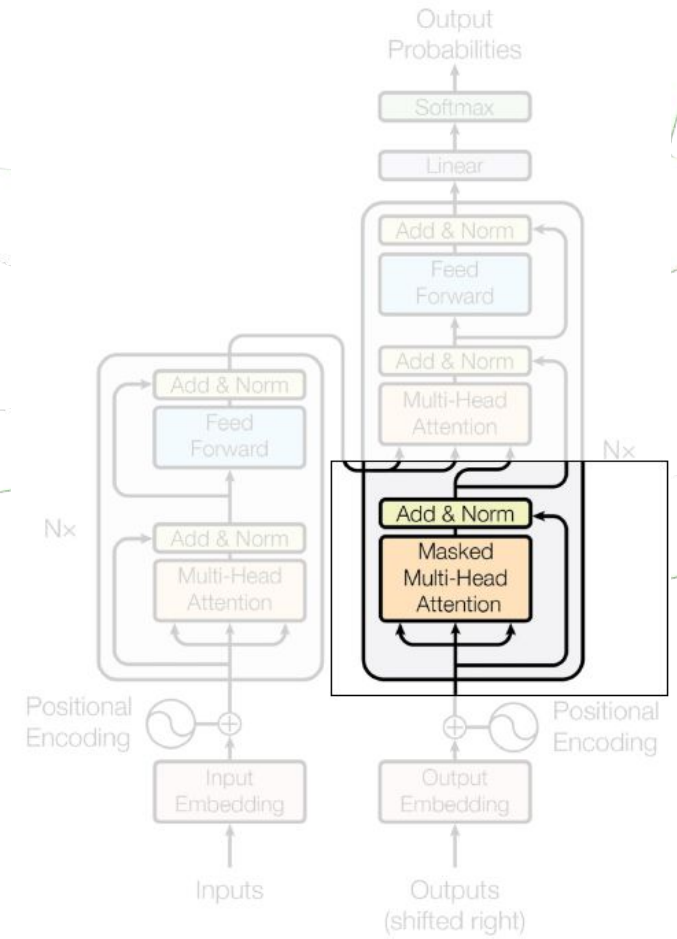
Outputs at time T should only pay attention to outputs until time T-1



Masked Multi-Head Attention

1	<sos>	Ich	have	einen	apfel	gegessen	<eos>
2	<sos>	Ich	have	einen	apfel	gegessen	<eos>
3	<sos>	Ich	have	einen	apfel	gegessen	<eos>
4	<sos>	Ich	have	einen	apfel	gegessen	<eos>
5	<sos>	Ich	have	einen	apfel	gegessen	<eos>
6	<sos>	Ich	have	einen	apfel	gegessen	<eos>
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

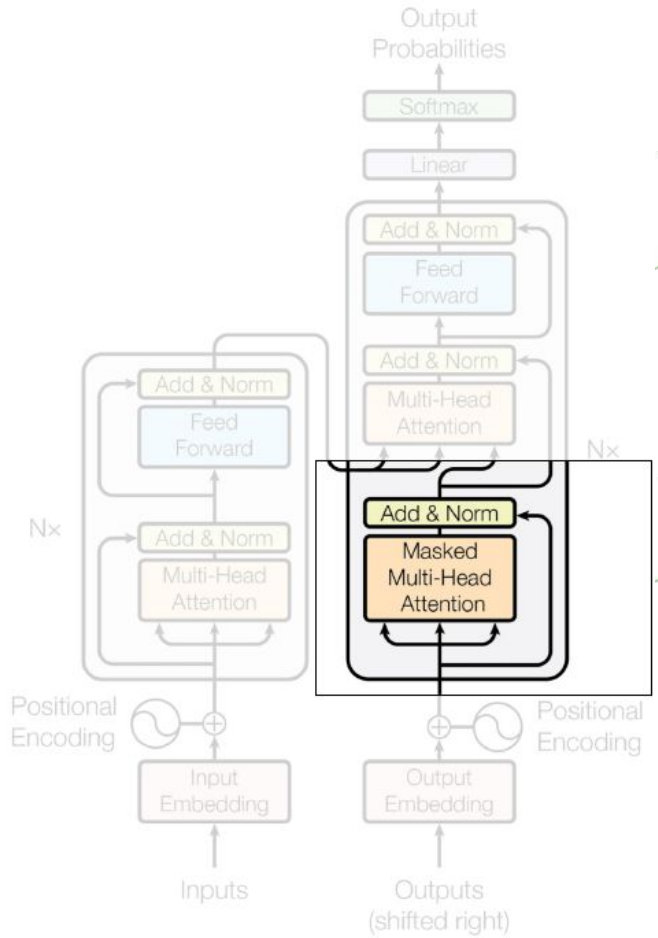
Mask the available attention values ?



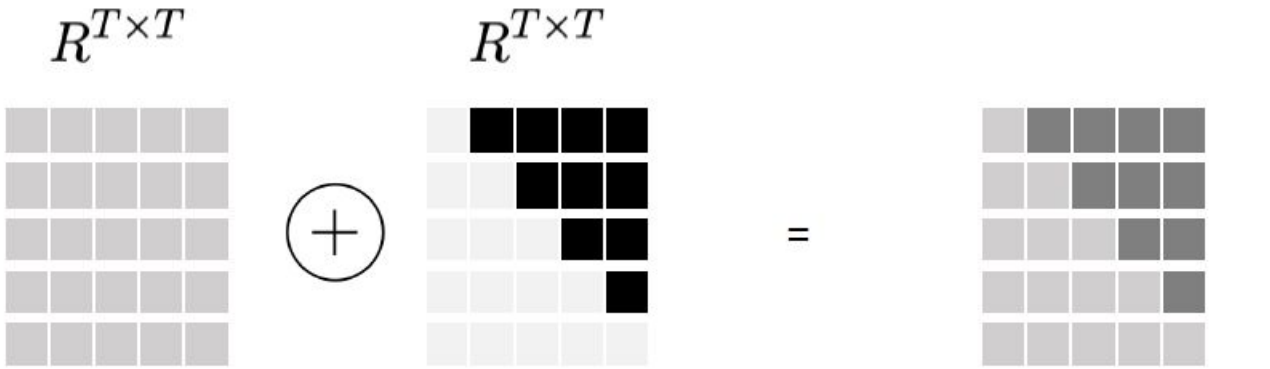
Masked Multi-Head Attention

1	<sos>	- ∞	- ∞	- ∞	- ∞	- ∞	- ∞
2	<sos>	Ich	- ∞	- ∞	- ∞	- ∞	- ∞
3	<sos>	Ich	have	- ∞	- ∞	- ∞	- ∞
4	<sos>	Ich	have	einen	- ∞	- ∞	- ∞
5	<sos>	Ich	have	einen	apfel	- ∞	- ∞
6	<sos>	Ich	have	einen	apfel	gegessen	- ∞
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

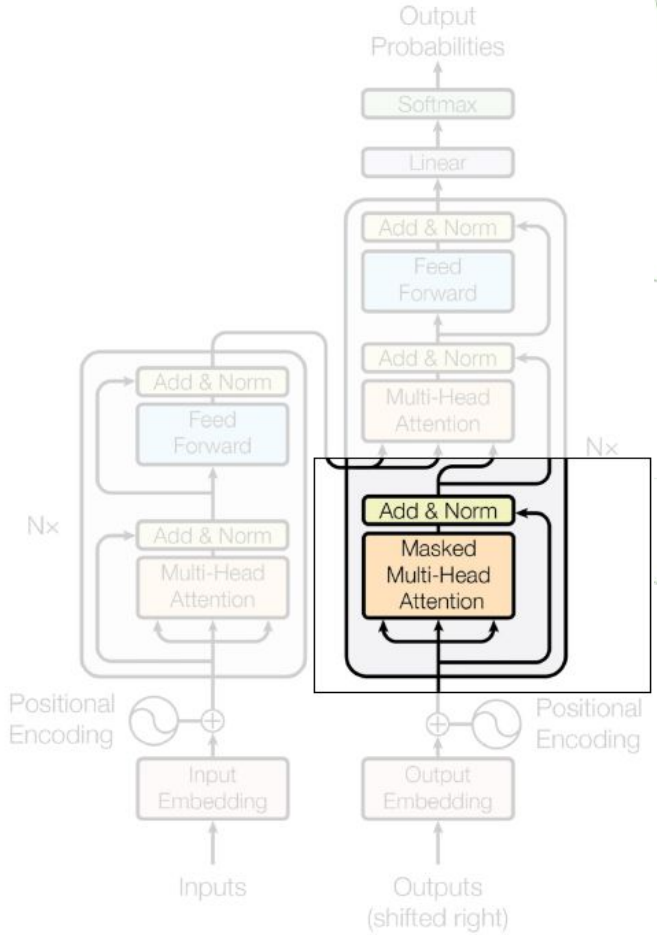
Softmax -> - ∞ -> 0



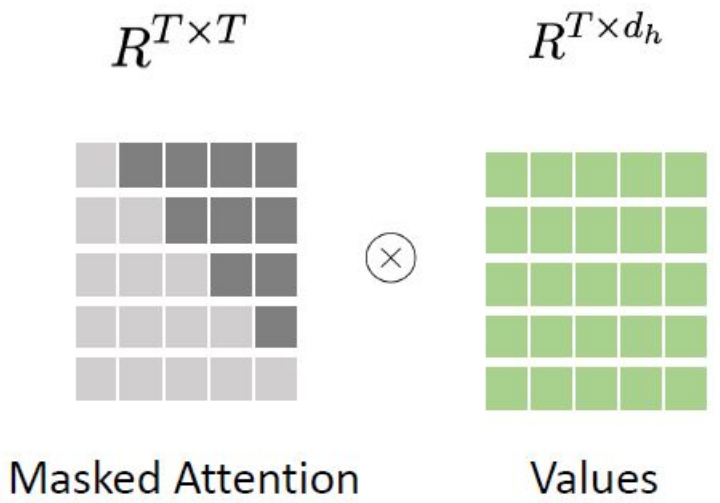
Masked Multi-Head Attention



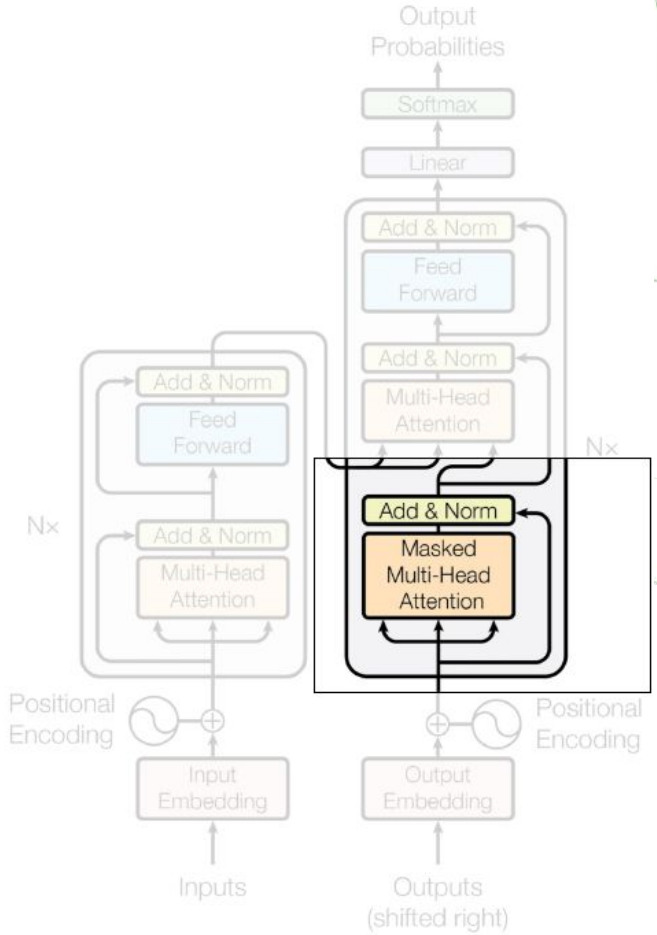
Masked Multi Head Attention : Z’



Masked Multi-Head Attention



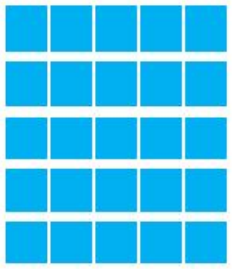
Masked Multi Head Atten



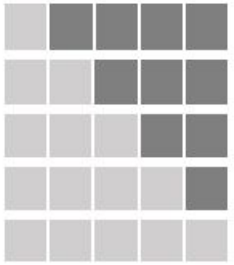
Encoder Decoder Attention

Encoder Decoder Attention ?

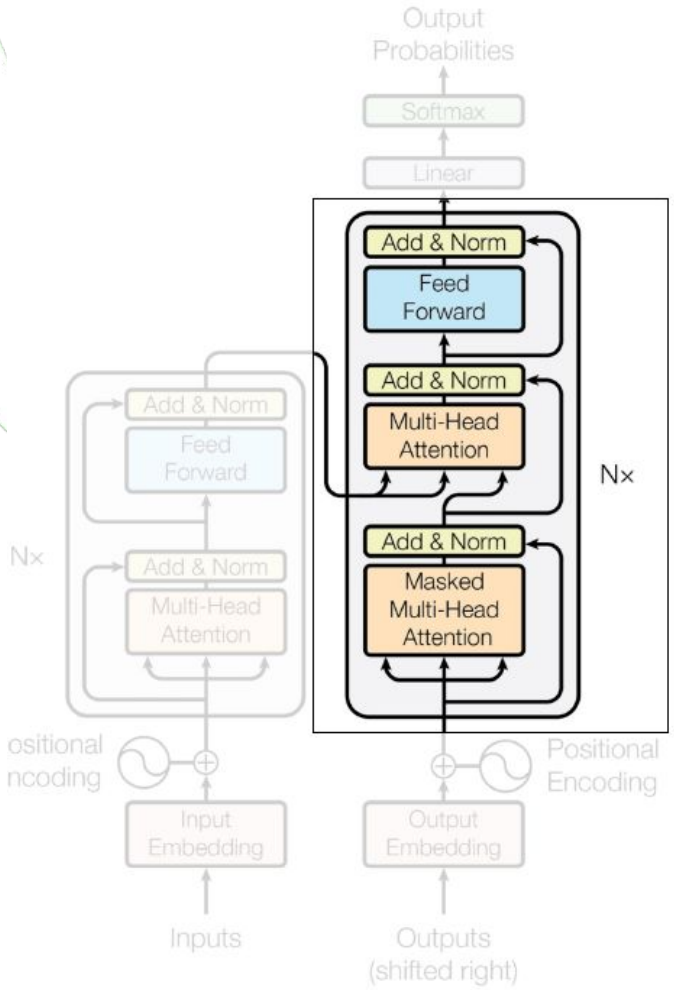
Add & Norm



Input



Norm(Z')



Encoder Decoder Attention

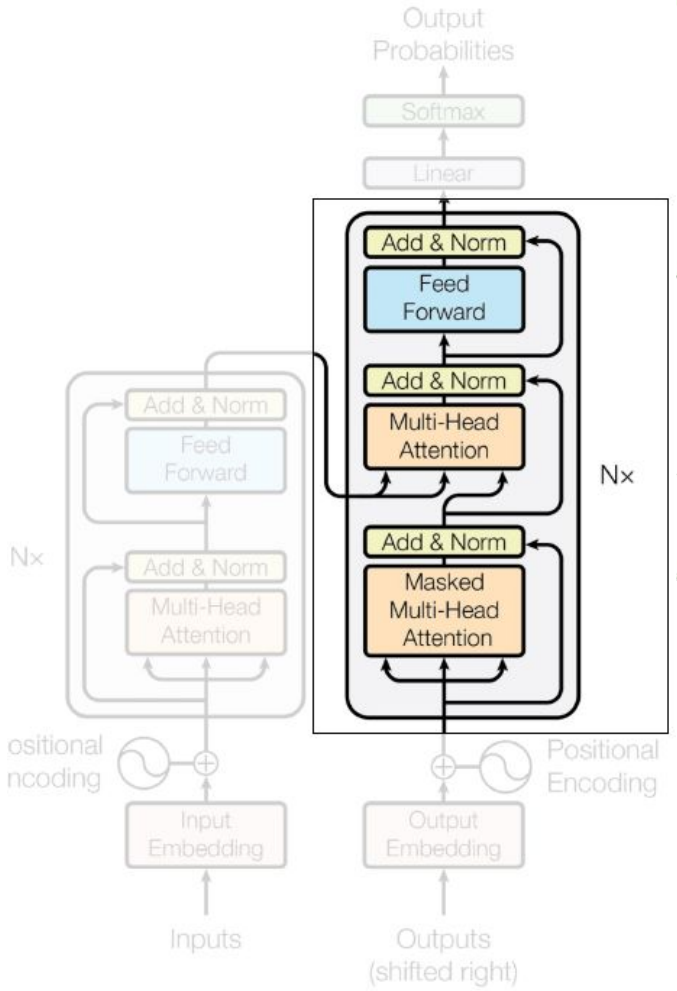
Encoder **Self** Attention

1. Queries from Encoder Inputs
2. Keys from Encoder Inputs
3. Values from Encoder Inputs

Decoder **Masked Self** Attention

1. Queries from Decoder Inputs
2. Keys from Decoder Inputs
3. Values from Decoder Inputs

Encoder Decoder Attention ?



Encoder Decoder Attention

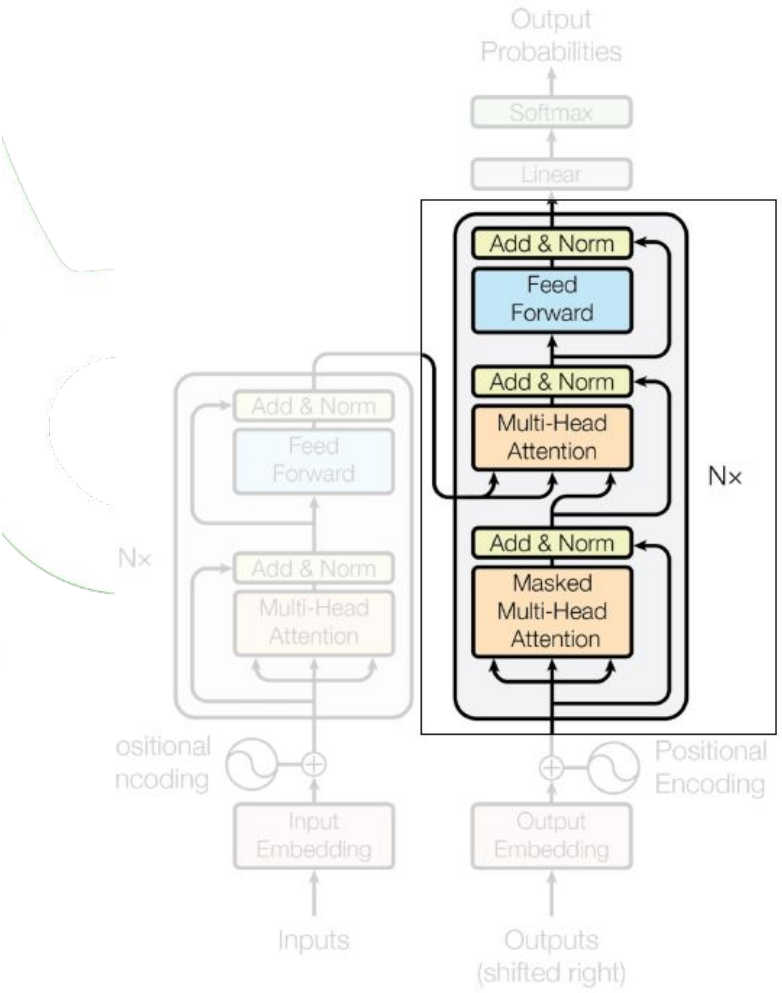
Encoder

Keys from **Encoder Outputs**
Values from **Encoder Outputs**

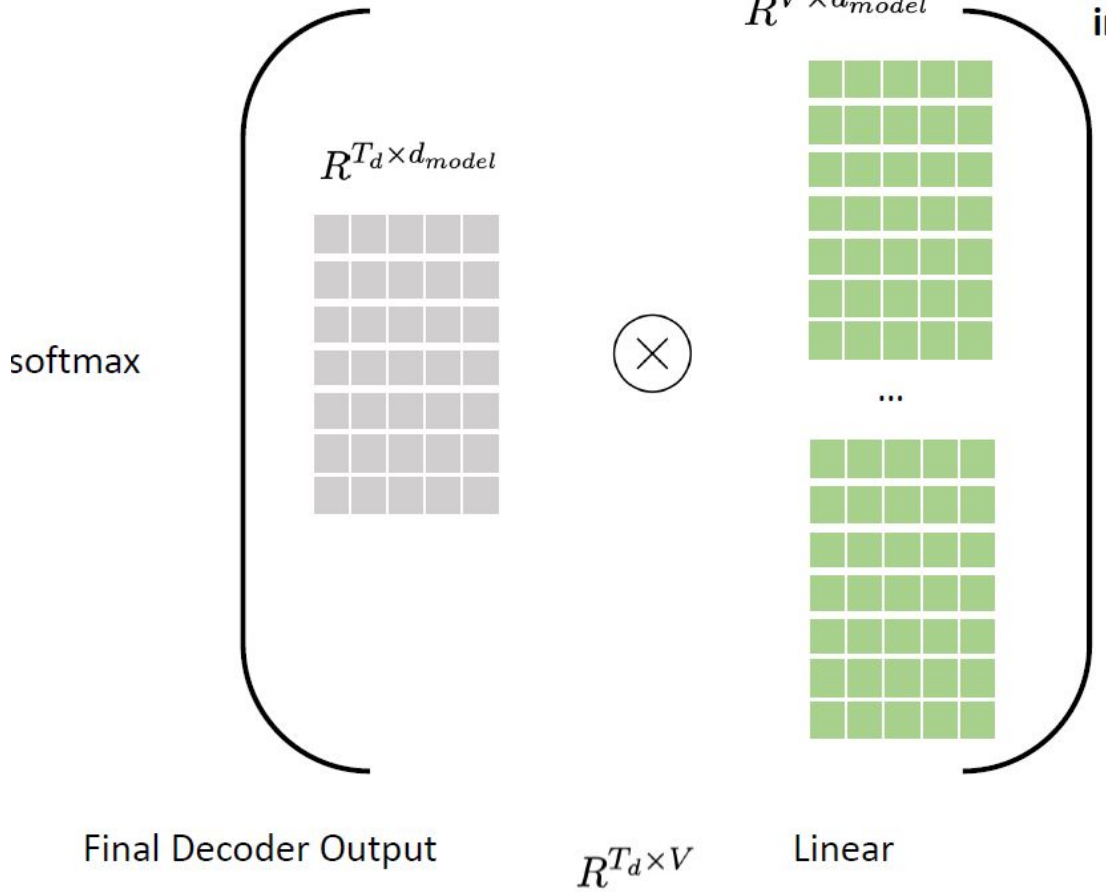
Decoder

Queries from **Decoder Inputs**

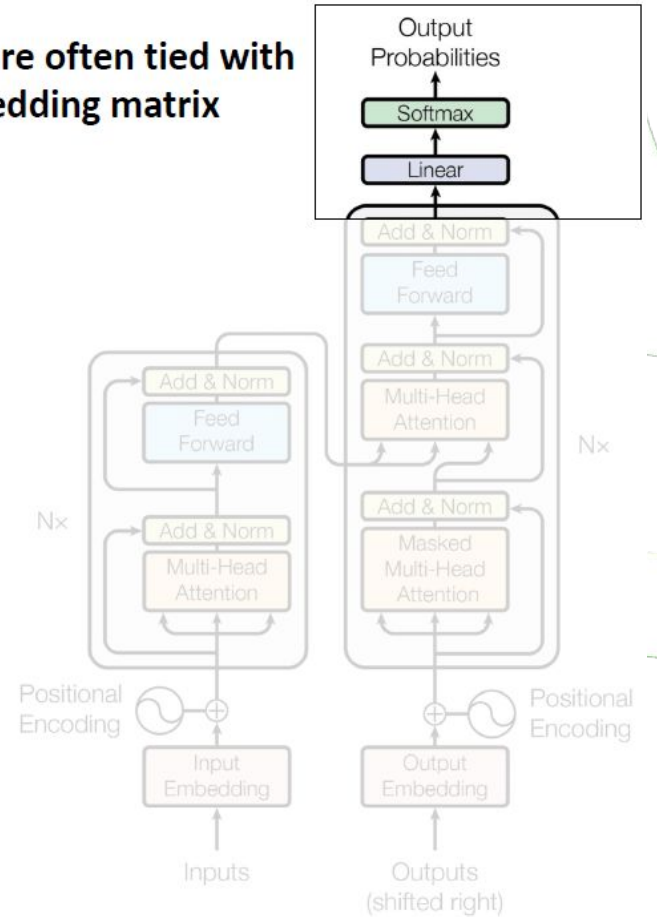
NOTE: Every decoder block receives the same FINAL encoder output



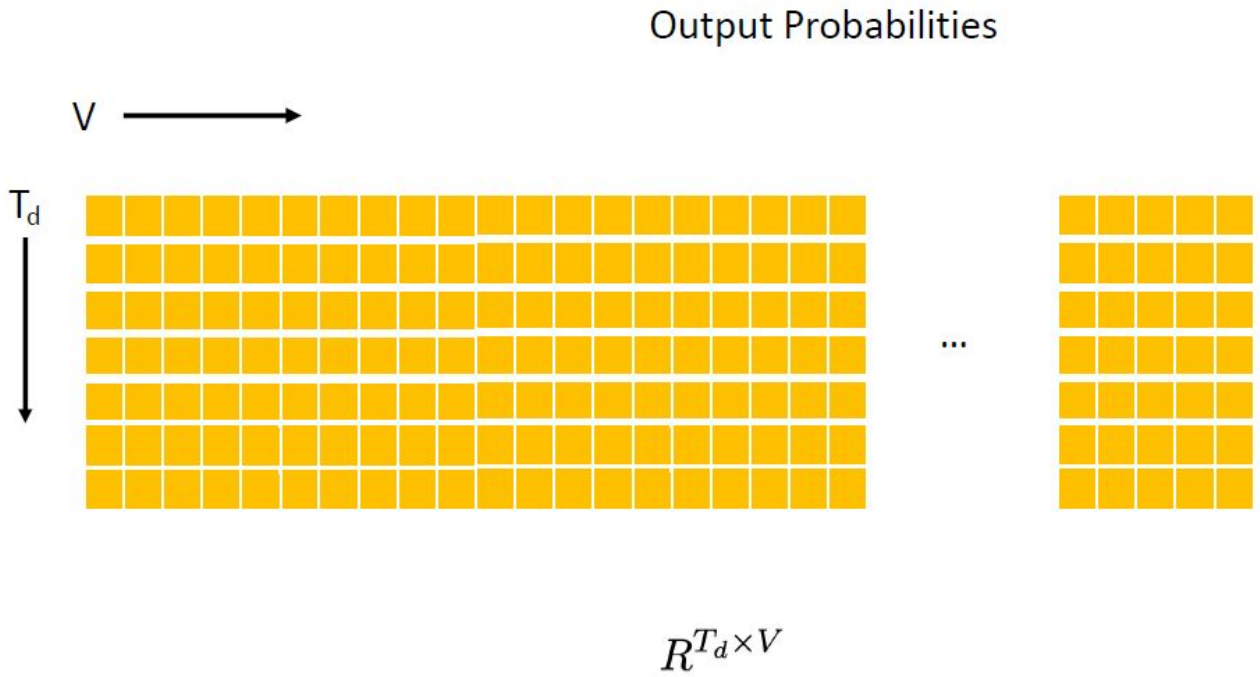
Linear

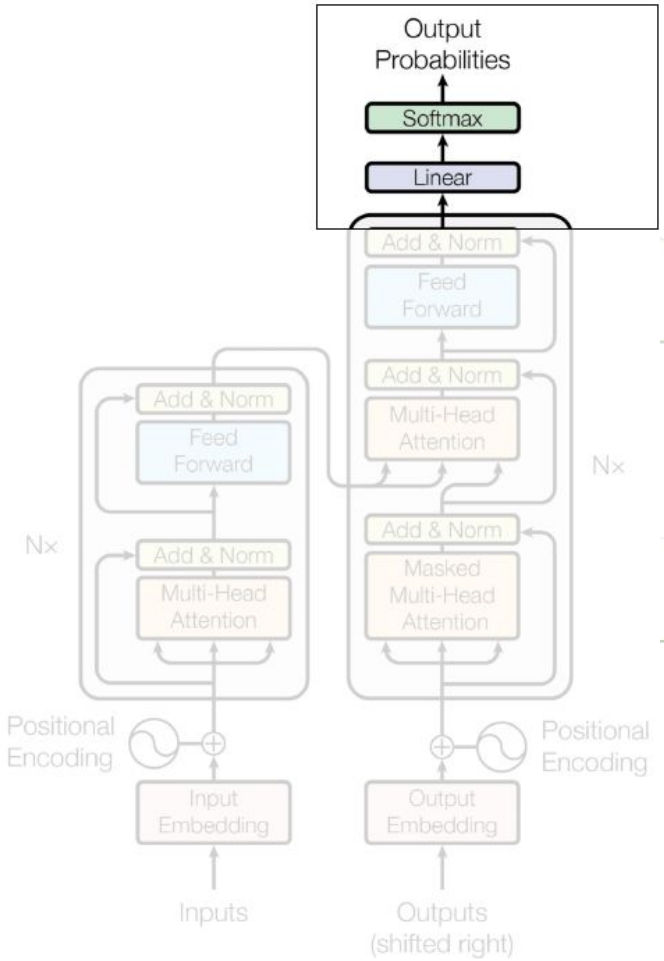


Linear weights are often tied with input embedding matrix

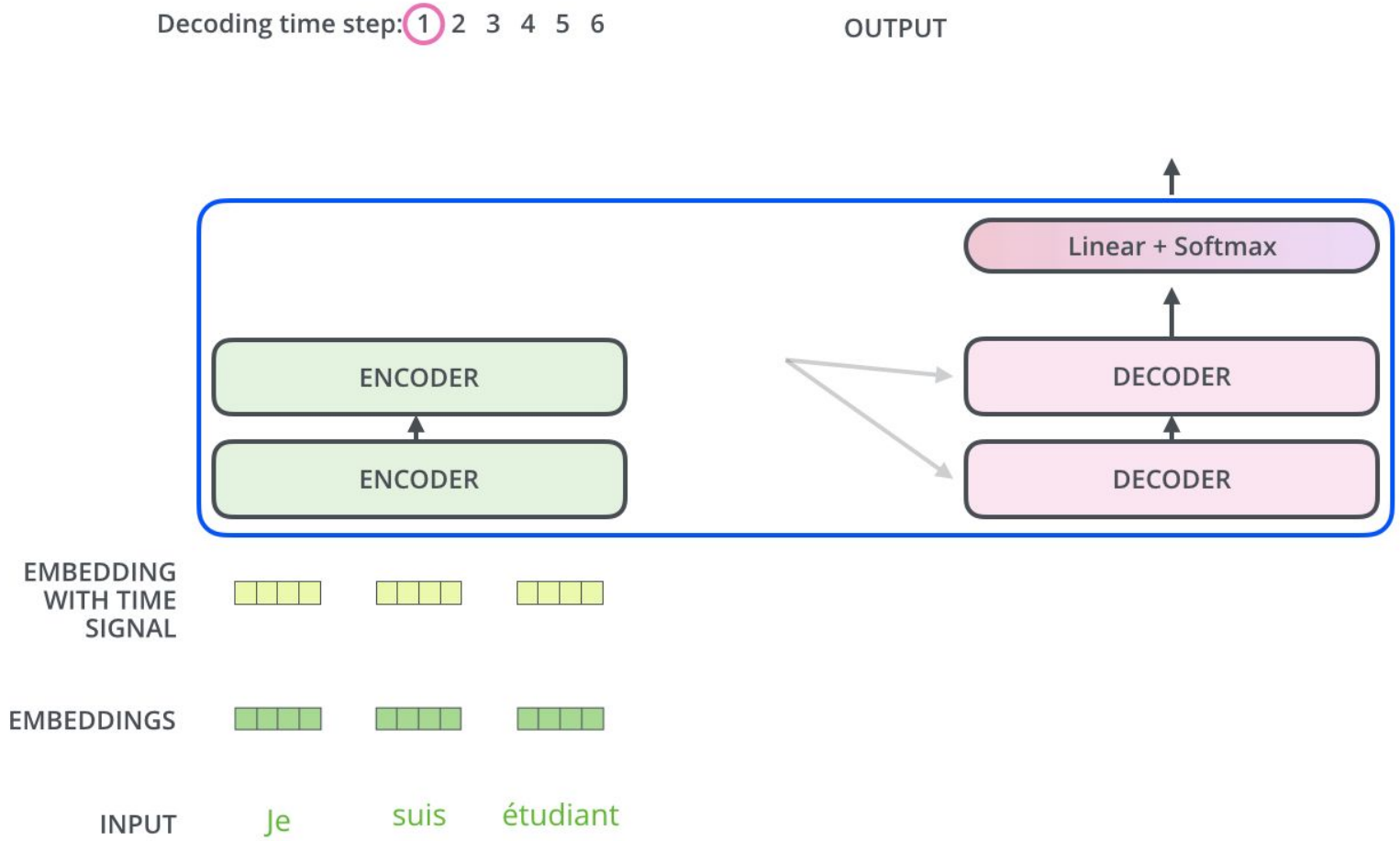


Softmax

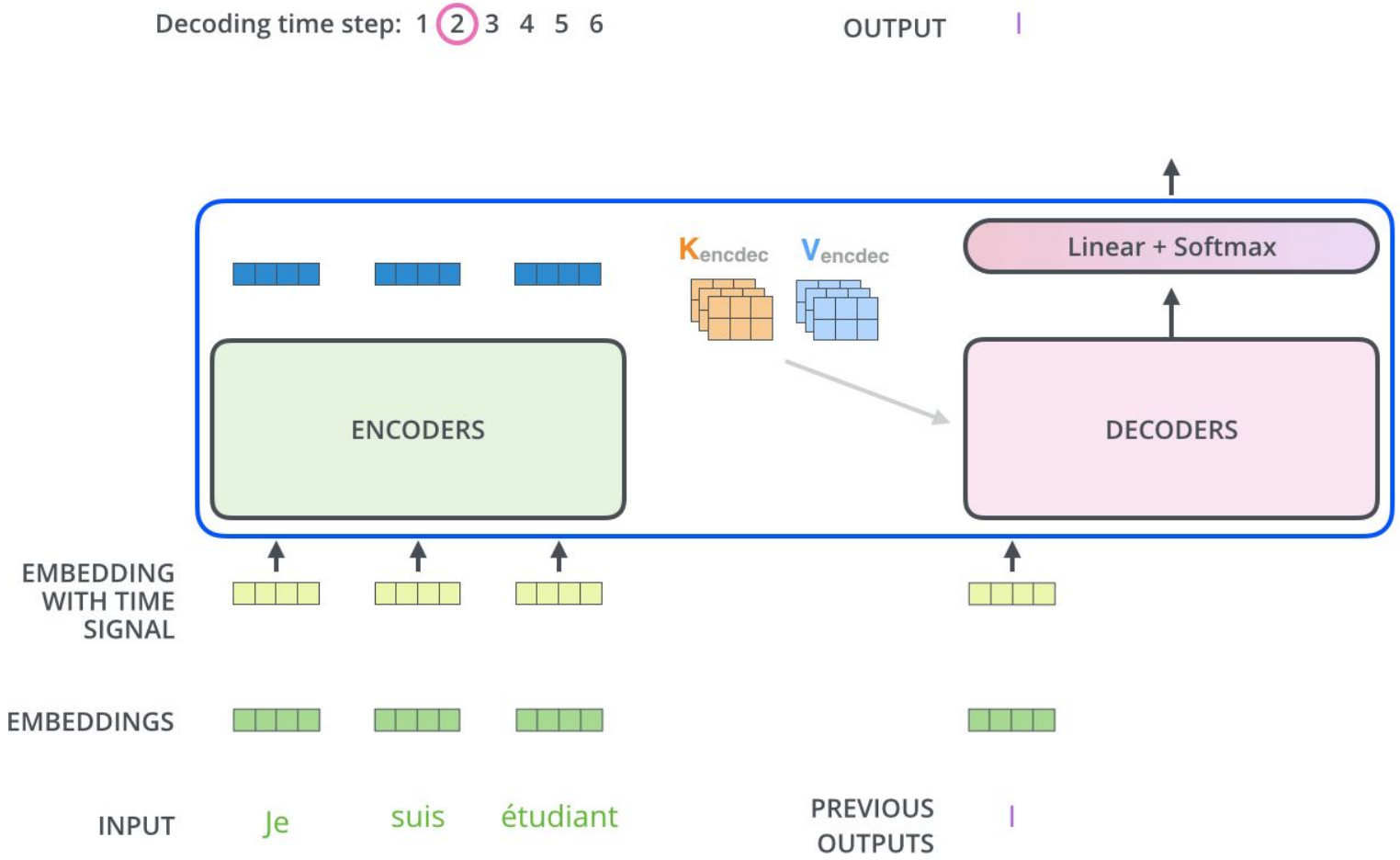

Output Probabilities
V
T_d
...
$R^{T_d \times V}$


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Transformer's pipeline

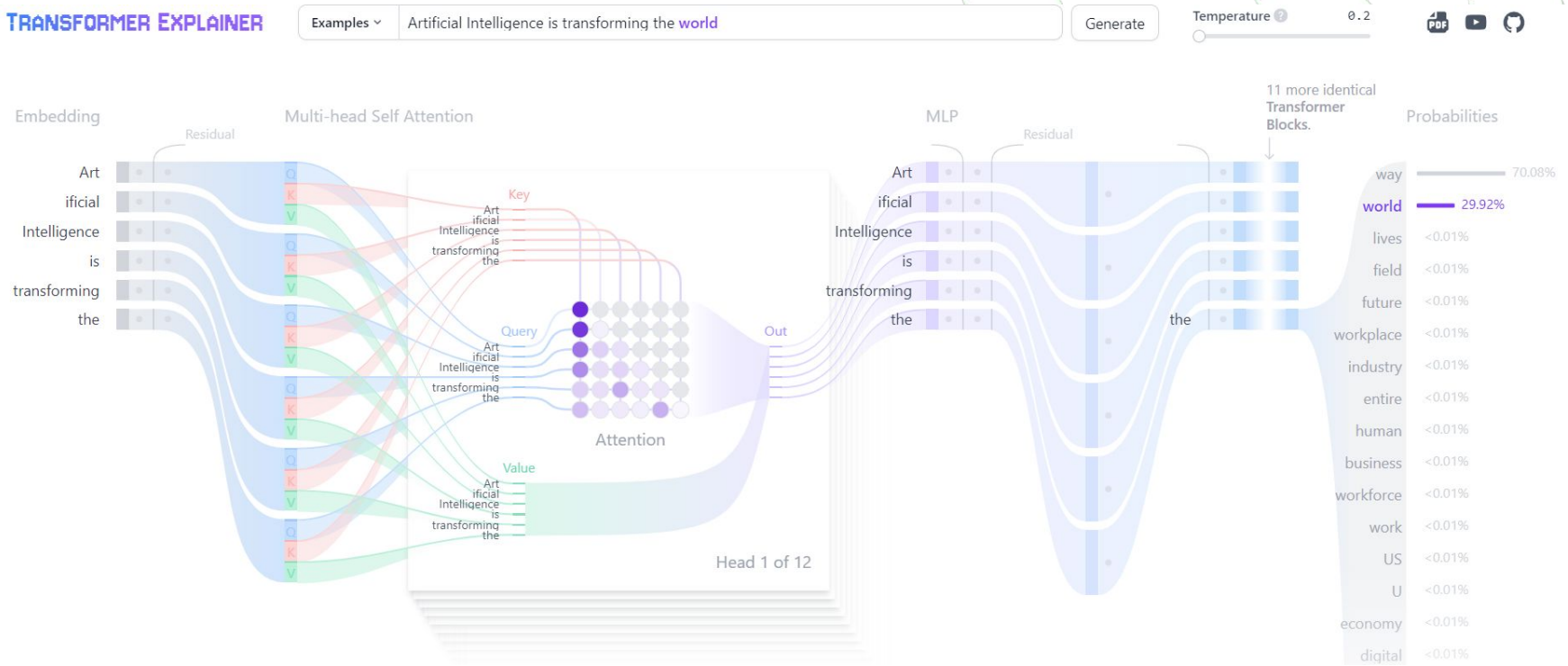


Transformer's pipeline



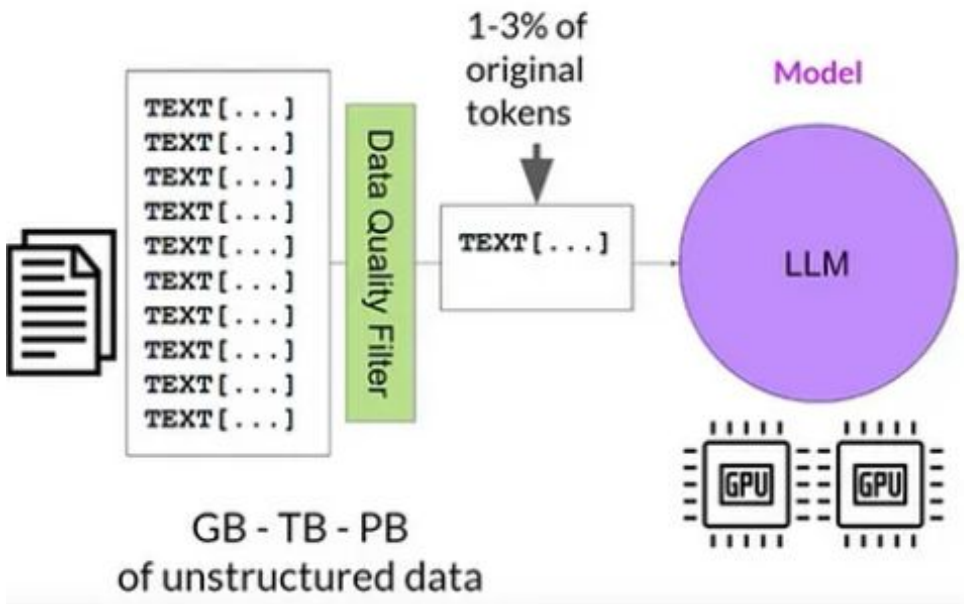
Transformer’s pipeline

https://poloclub.github.io/transformer-explainer/



Self supervised pre-training

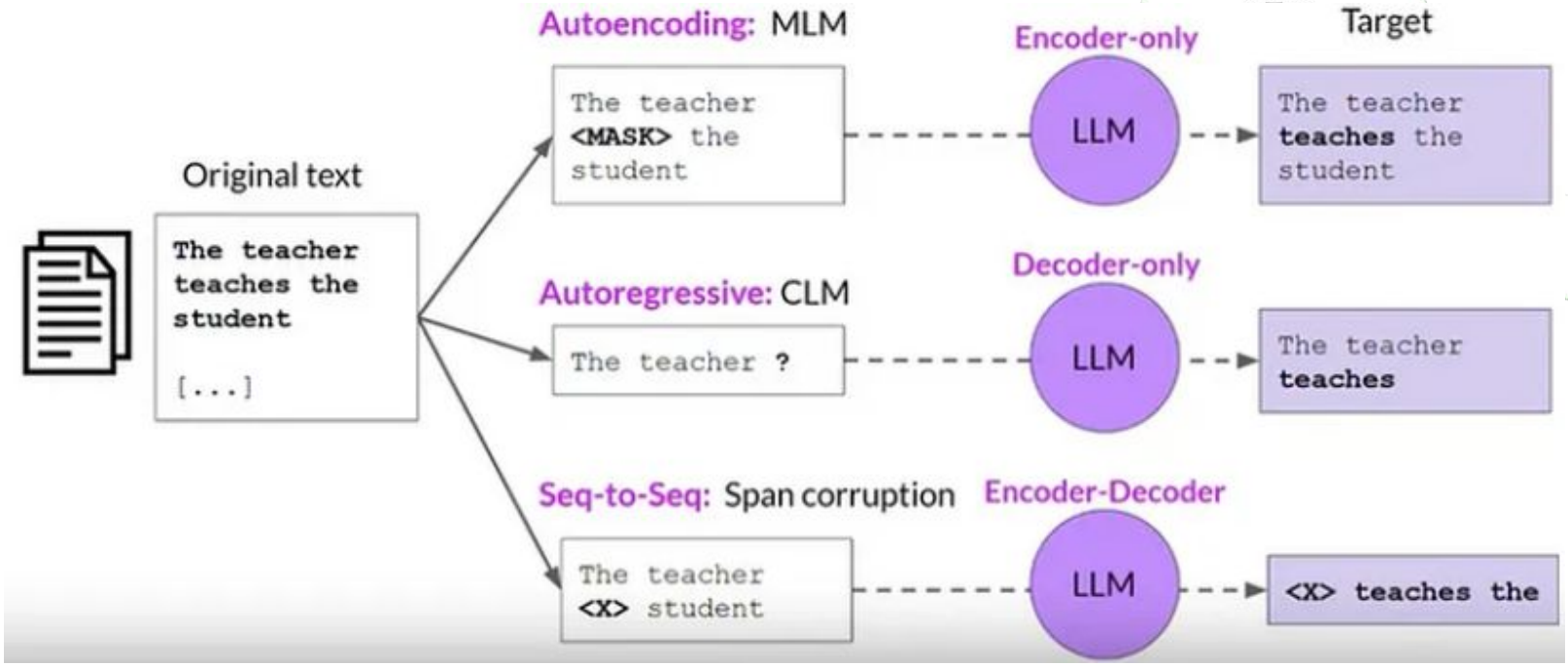
- Pre-training a large language model is done in a self-supervised way, meaning it is trained with unlabeled data which is just text from the internet.
- There is no need to assign labels on the dataset. No supervision? Create supervised tasks and solve them.



Token String	Token ID	Embedding / Vector Representation
'_The'	37	[-0.0513, -0.0584, 0.0230, ...]
'_teacher'	3145	[-0.0335, 0.0167, 0.0484, ...]
'_teaches'	11749	[-0.0151, -0.0516, 0.0309, ...]
'_the'	8	[-0.0498, -0.0428, 0.0275, ...]
'_student'	1236	[-0.0460, 0.0031, 0.0545, ...]
...	...	...

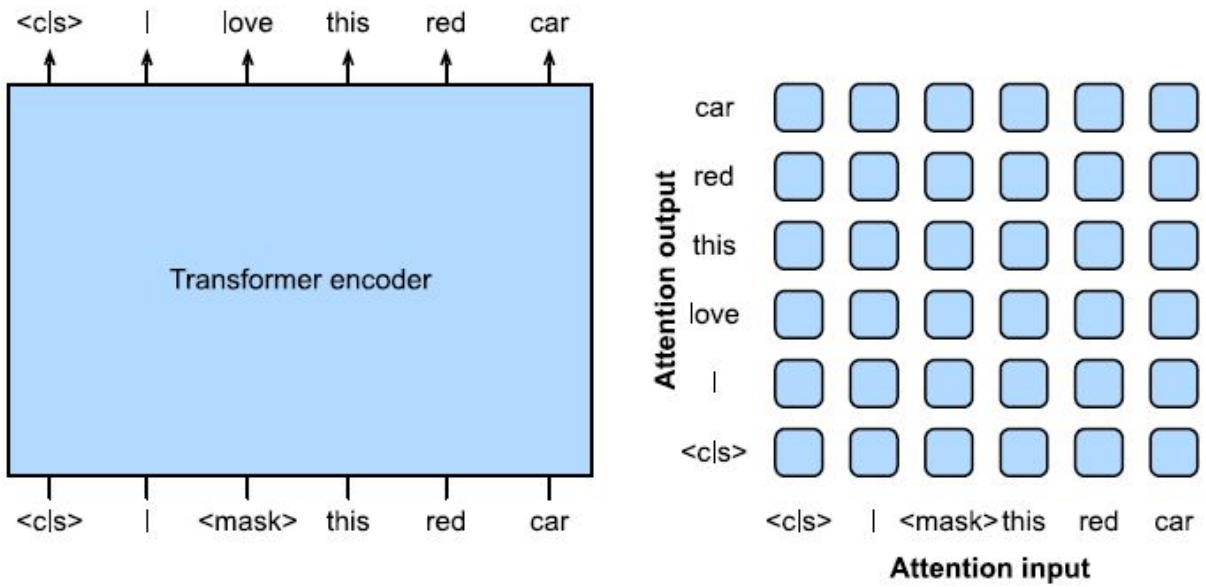
Self supervised pre-training

- Autoencoding models consist only of an encoder and typically predict the masked word from the every preceding and following words in the sentence, therefore it is bi-directional. The model has the knowledge of the entire context.



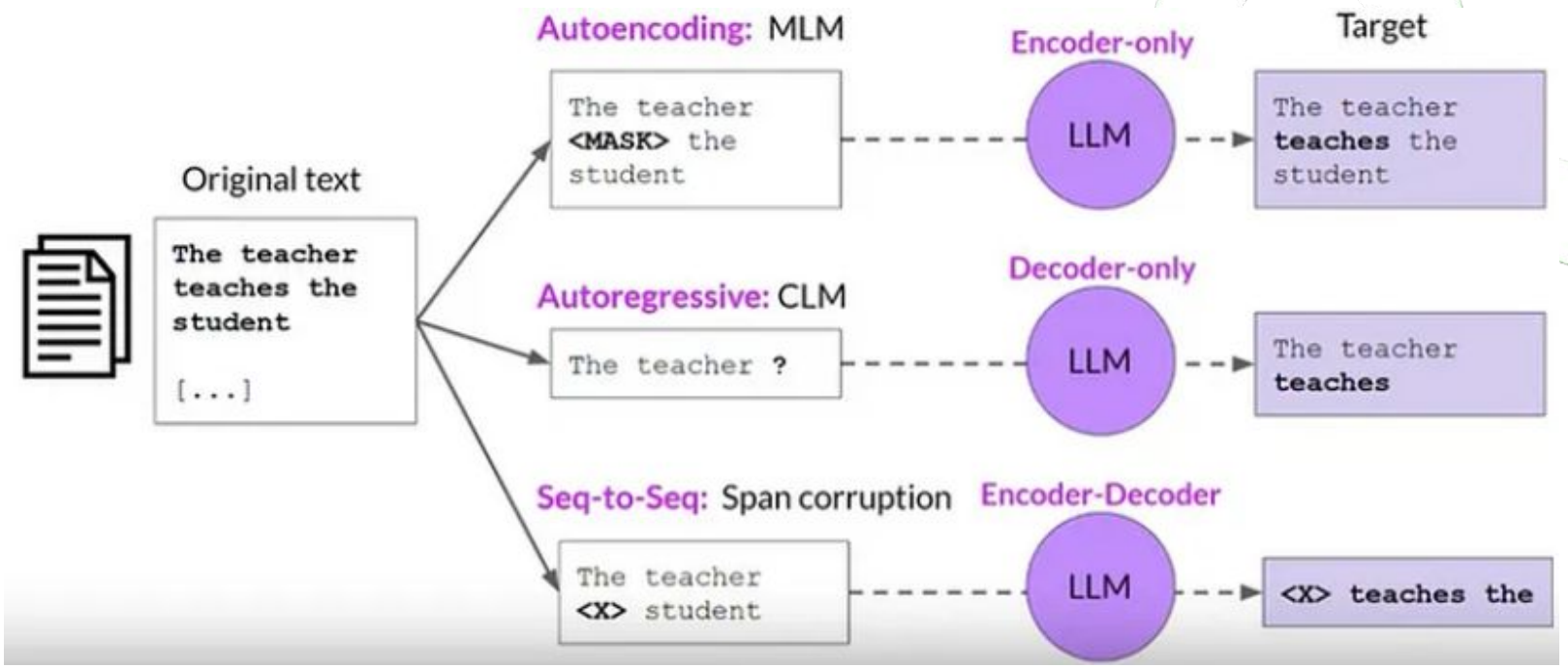
Masked Language Modeling

- Input text with randomly masked tokens is fed into a Transformer encoder to predict the masked tokens.
- As illustrated in the figure, an original text sequence “I”, “love”, “this”, “red”, “car” is prepended with the “<cls>” token, and the “<mask>” token randomly replaces “love”; then the cross-entropy loss between the masked token “love” and its prediction is to be minimized during pre-training.



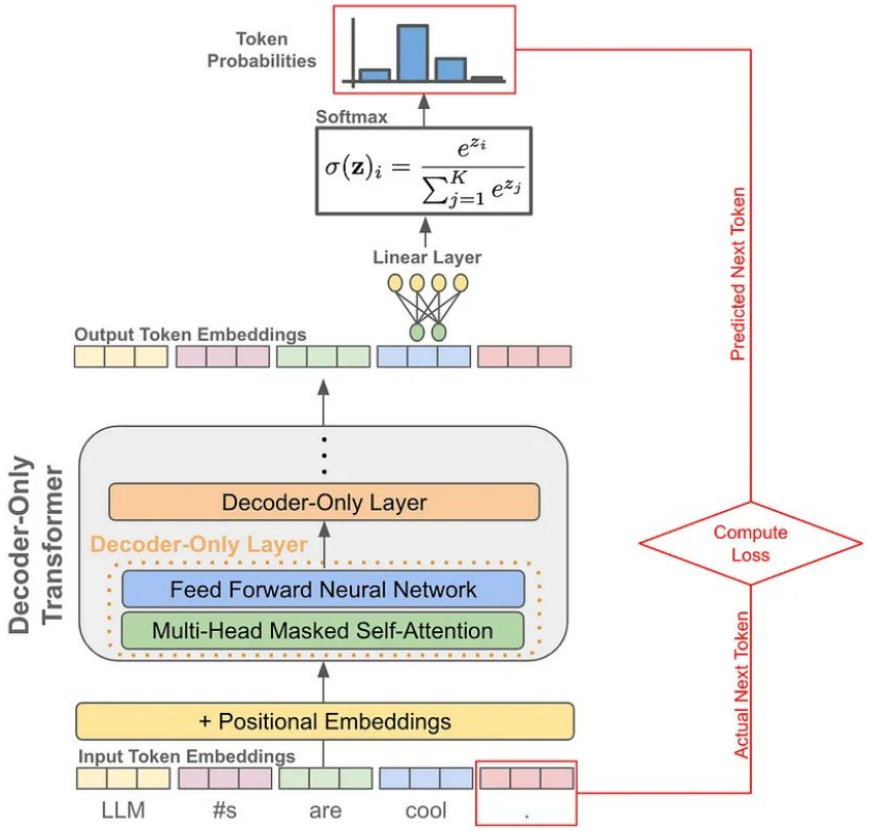
Self supervised pre-training

- Autoregressive models consist only of a decoder and predict the masked word from the preceding words. Thus, autoregressive models are great at autocompleting a sentence, which is what happens in text generation models.



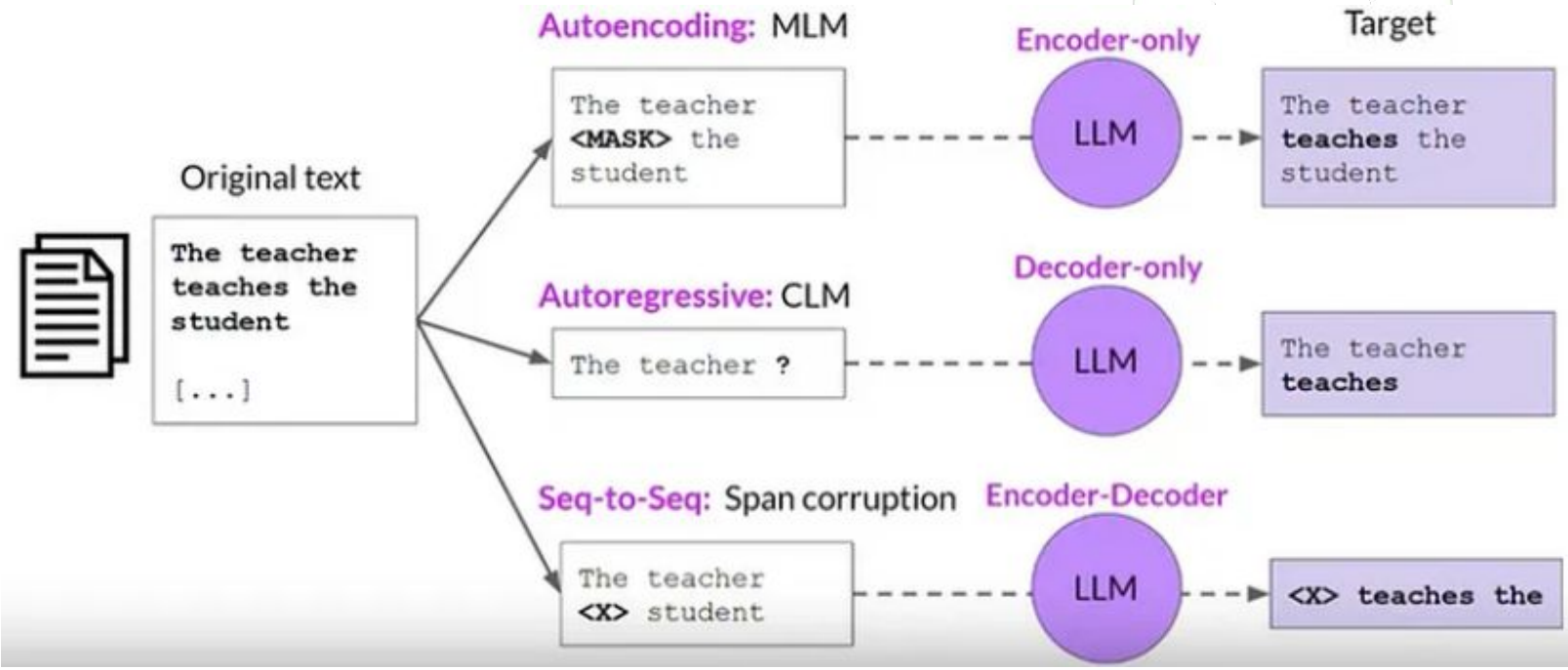
Next Token Prediction

- Any text can be used for this pre-training task, which only requires the prediction of the next word in the sequence.



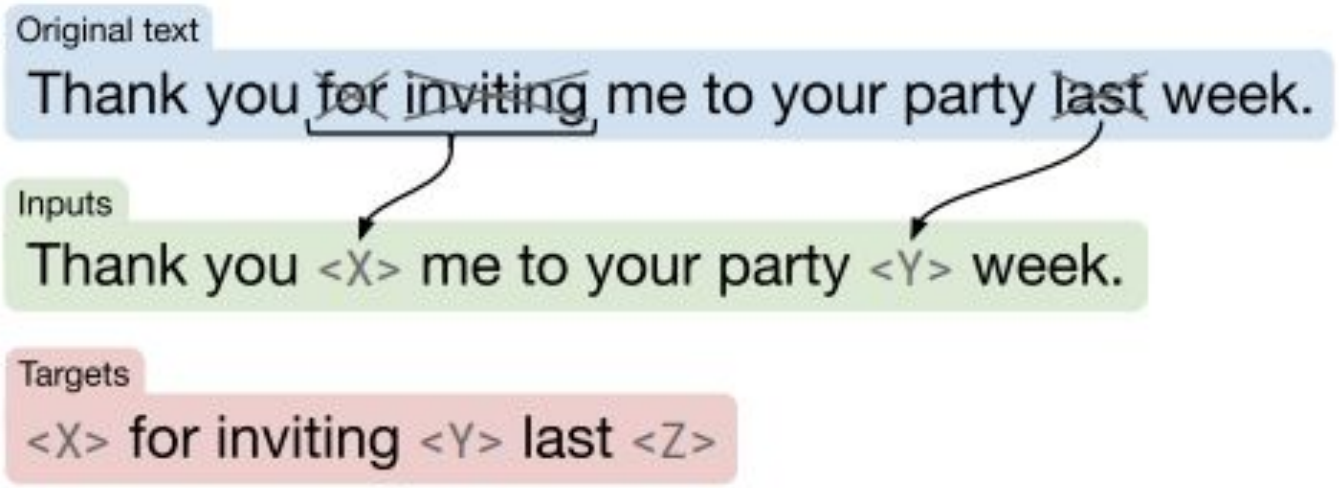
Self supervised pre-training

- In training seq2seq models, random sequences of input are masked and replaced with a unique token sentinel. The output is the sentinel token followed by the predicted tokens. In summary, seq2seq models both need to understand the context and generate a text.



Span corruption

- In the original text, some words are dropped out with a unique sentinel token. Words are dropped out independently uniformly at random. The model is trained to predict basically sentinel tokens to delineate the dropped out text.



THANKS!

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani