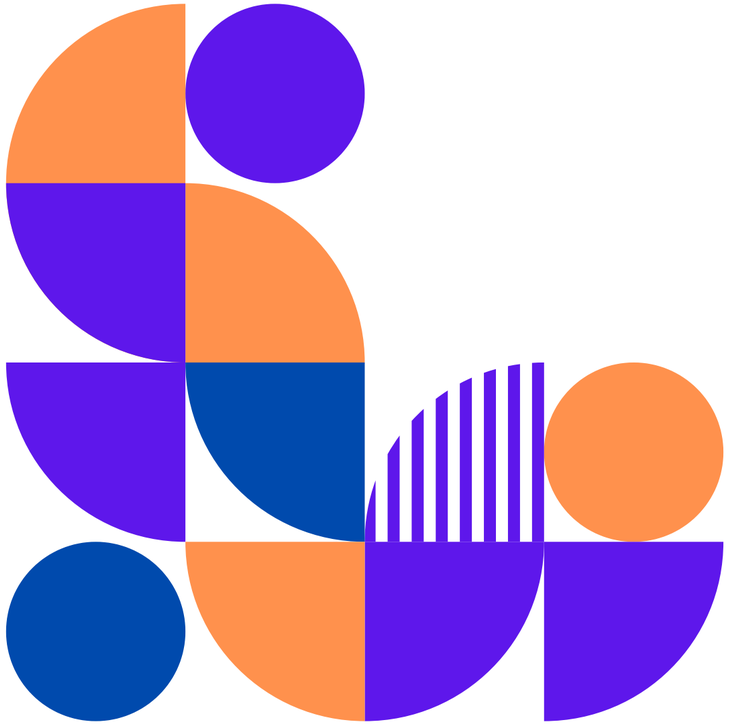
Chapter 10: Transform and roll out

Allyne Beatriz

Daniela Maia

Luca Pareja

Vinícius Sohsten



✨ Spoilers

- Modify the **multi-headed attention** mechanism to use **narrow attention**
- Use **layer normalization** to standardize individual data points
- **Stack "layers"** together to build **Transformer encoders and decoders**
- Add **layer normalization**, **dropout**, and **residual connections** to each "sub-layer" operation
- Learn the difference between **norm-last** and **norm-firts** "sub-layers"
- Train a **Transformer** to predict from a source sequence
- Build and train a **Vision Transformer** to perform image classification

Multi-head attention to narrow attention

Previously...

We used full attention heads to build a multi-head attention and we called it **wide attention**

It works! But... is expensive as the number of dimensions grows. That's where **narrow attention** comes in

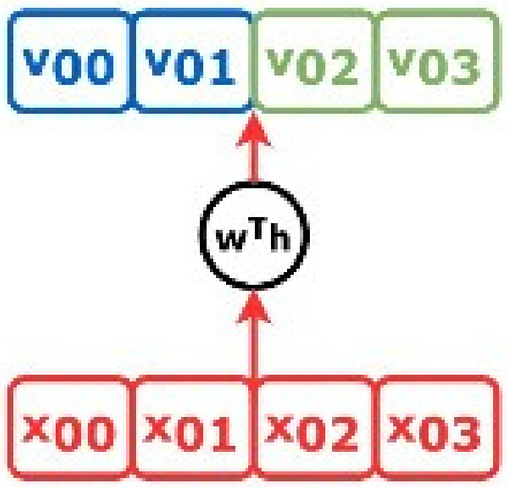
Why?

Narrow Attention

Divide and conquer. Each attention point shall work with a specific chunck of transformed data.

The chunks used are projections of the original data, not the data itself.

The image next represents a transformation, taking an element with 4 dimensions and turning them into a single value, for application in the attention mechanism.



Narrow Attention

Right X Wrong

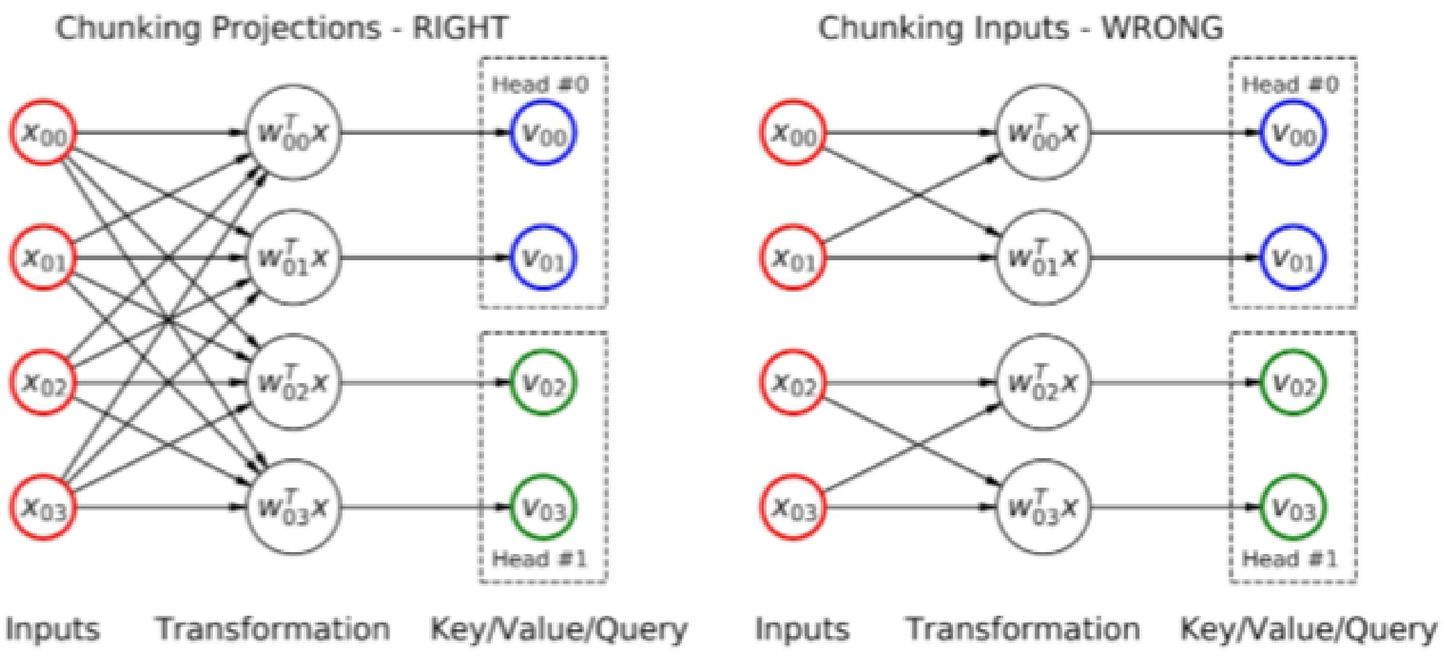


Figure 10.2 - Chunking: the wrong and the right way

Narrow Attention

How does it work?

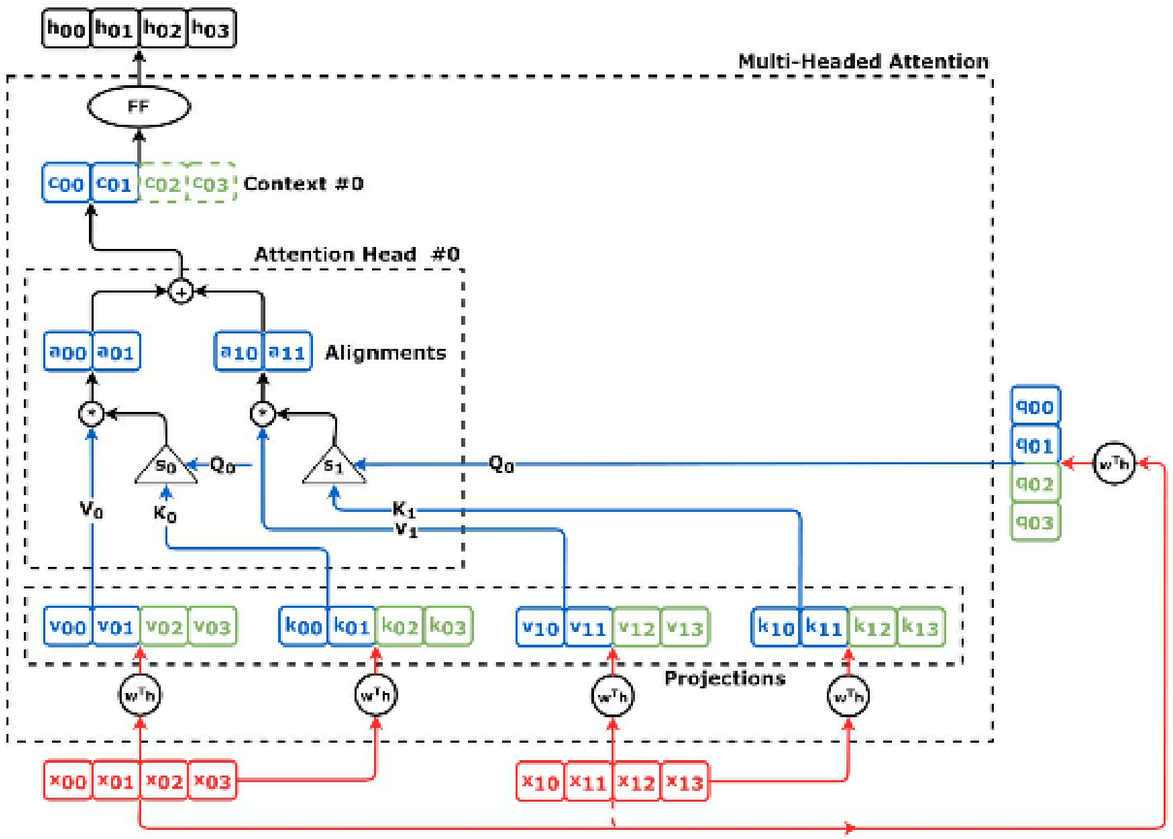


Figure 10.3 - Self-(narrow)attention mechanism

- x0 and x1 go through distinctic affine transformations, generating the values v0 and v1and the keys (k0 and k1), known as projections
- The data points also undergo another affine transformation to generate queries q0 e q1.
- Projections are directly related to the inputs
- It uses only a chunk of the projections to compute the context vector.
- The dimensions are split into chunks of two dimensions each.
- The first attention head is related only to the blue chunks to compute the context vector.
- In this sense, the second attention head is related to the green chunks to compute the other half of the context vector.
- Thus, the context vector goes through a feed-forward network to generate the hidden states

Multi-head attention

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = int(d_model / n_heads)
        self.linear_query = nn.Linear(d_model, d_model)
        self.linear_key = nn.Linear(d_model, d_model)
        self.linear_value = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(p=dropout)
        self.alphas = None
```

1

```python
    def make_chunks(self, x):
        batch_size, seq_len = x.size(0), x.size(1)
        # N, L, D -> N, L, n_heads * d_k
        x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
        # N, n_heads, L, d_k
        x = x.transpose(1, 2)
        return x
```

2

```python
    def init_keys(self, key):
        # N, n_heads, L, d_k
        self.proj_key = self.make_chunks(self.linear_key(key))
        self.proj_value = self.make_chunks(self.linear_value(key))
```

3

```python
    def score_function(self, query):
        # scaled dot product
        # N, n_heads, L, d_k x # N, n_heads, d_k, L -> N, n_heads, L, L
        proj_query = self.make_chunks(self.linear_query(query))
        dot_products = torch.matmul(proj_query,
                                    self.proj_key.transpose(-2, -1))
        scores =  dot_products / np.sqrt(self.d_k)
        return scores
```

4

```python
    def attn(self, query, mask=None):
        # Query is batch-first: N, L, D
        # Score function will generate scores for each head
        scores = self.score_function(query) # N, n_heads, L, L
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(scores, dim=-1) # N, n_heads, L, L
        alphas = self.dropout(alphas)
        self.alphas = alphas.detach()

        # N, n_heads, L, L x N, n_heads, L, d_k -> N, n_heads, L, d_k
        context = torch.matmul(alphas, self.proj_value)
        return context
```

5

```python
    def output_function(self, contexts):
        # N, L, D
        out = self.linear_out(contexts) # N, L, D
        return out
```

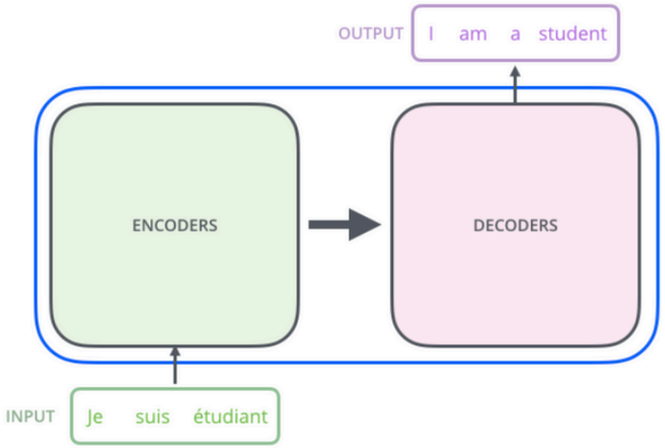
6

```python
    def forward(self, query, mask=None):
        if mask is not None:
            # N, 1, L, L - every head uses the same mask
            mask = mask.unsqueeze(1)

        # N, n_heads, L, d_k
        context = self.attn(query, mask=mask)
        # N, L, n_heads, d_k
        context = context.transpose(1, 2).contiguous()
        # N, L, n_heads * d_k = N, L, d_model
        context = context.view(query.size(0), -1, self.d_model)
        # N, L, d_model
        out = self.output_function(context)
        return out
```

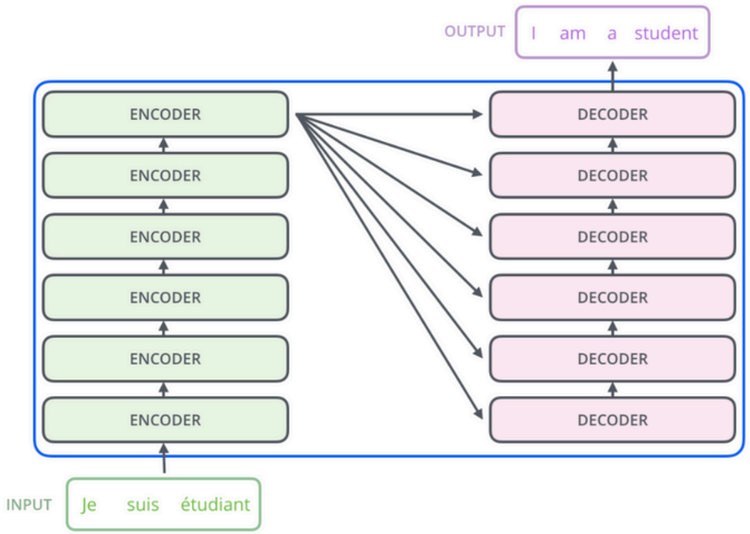
7

Stacking Encoders and Decoders

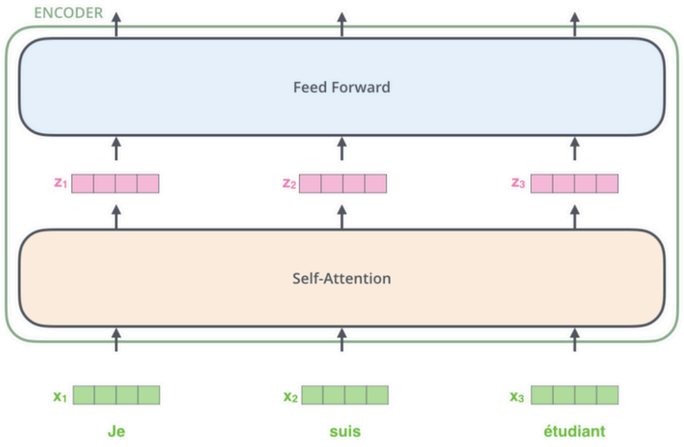
Deep encoder/decoder



“Layers”



“Sub-Layers”

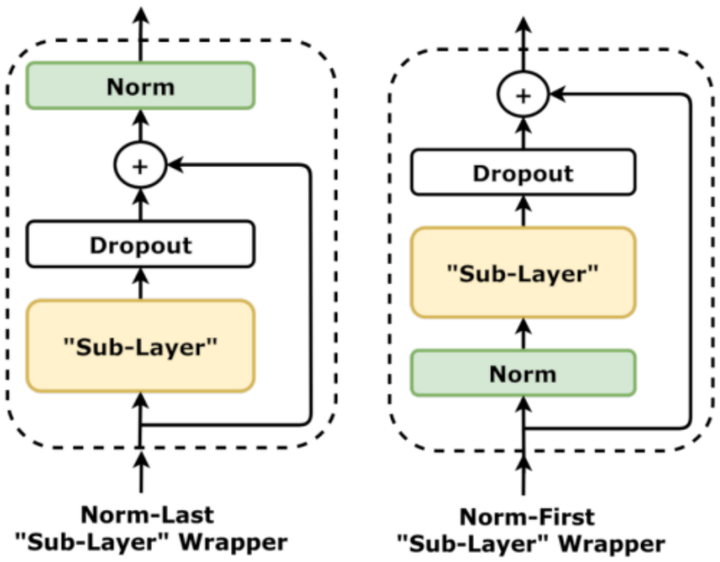


Cool! Is this Transformer already then?

Not yet, no.

Wrapping “Sub-Layers”

With many **“layers”** we're going to run into familiar issues, like the **vanishing gradients problem**.



Normalizes the outputs

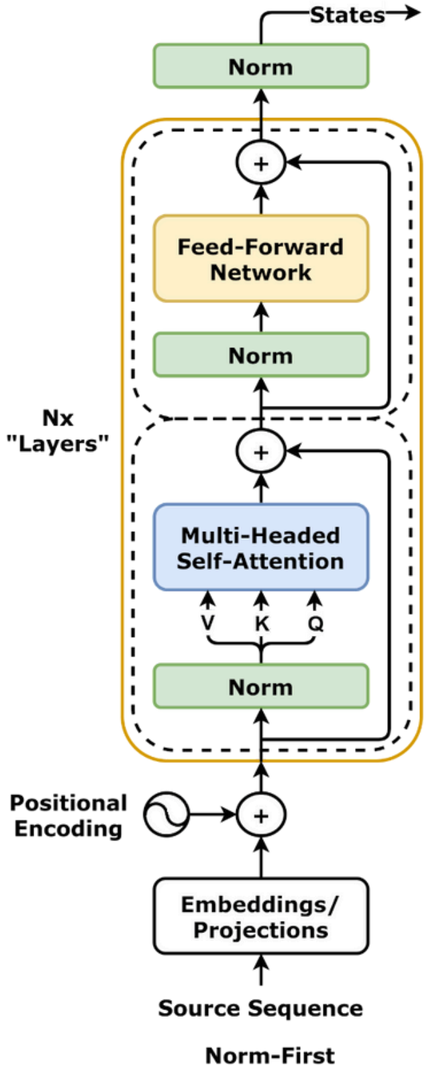
$$\text{outputs}_{\text{norm-last}} = \text{norm}(\text{inputs} + \text{dropout}(\text{sublayer}(\text{inputs})))$$

Normalizes the inputs

$$\text{outputs}_{\text{norm-first}} = \text{inputs} + \text{dropout}(\text{sublayer}(\text{norm}(\text{inputs})))$$

🤔 **Wich one is best?** There is no right and wrong!

Transformer Encoder




```python
class EncoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
```

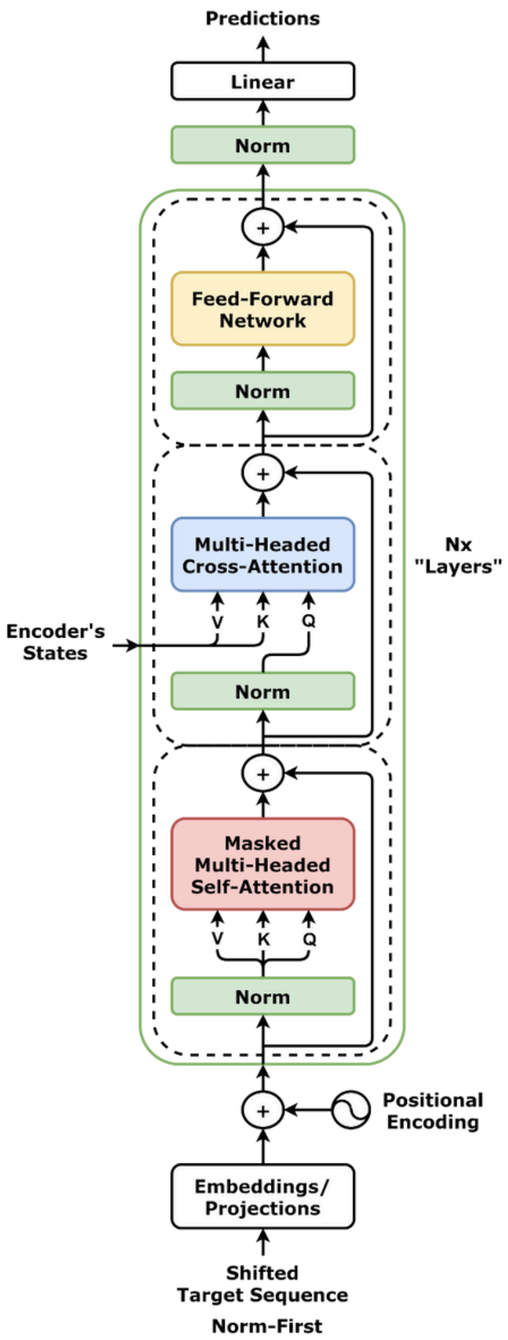
```python
    def forward(self, query, mask=None):
        # Sublayer #0
        # Norm
        norm_query = self.norm1(query)
        # Multi-headed Attention
        self.self_attn_heads.init_keys(norm_query)
        states = self.self_attn_heads(norm_query, mask)
        # Add
        att = query + self.drop1(states)

        # Sublayer #1
        # Norm
        norm_att = self.norm2(att)
        # Feed Forward
        out = self.ffn(norm_att)
        # Add
        out = att + self.drop2(out)
        return out
```

```python
enclayer = nn.TransformerEncoderLayer(
    d_model=6, nhead=3, dim_feedforward=20
)
enctransf = nn.TransformerEncoder(
    enclayer, num_layers=1, norm=nn.LayerNorm
)
```

Transformer Decoder

```python
class DecoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, ff_units, dropout=0.1):
```

```python
    def forward(self, query, source_mask=None, target_mask=None):
        # Sublayer #0
        # Norm
        norm_query = self.norm1(query)
        # Masked Multi-head Attention
        self.self_attn_heads.init_keys(norm_query)
        states = self.self_attn_heads(norm_query, target_mask)
        # Add
        att1 = query + self.drop1(states)

        # Sublayer #1
        # Norm
        norm_att1 = self.norm2(att1)
        # Multi-head Attention
        encoder_states = self.cross_attn_heads(norm_att1, source_mask)
        # Add
        att2 = att1 + self.drop2(encoder_states)

        # Sublayer #2
        # Norm
        norm_att2 = self.norm3(att2)
        # Feed Forward
        out = self.ffn(norm_att2)
        # Add
        out = att2 + self.drop3(out)
        return out
```



Layer Normalization

- Layer normalization **standardizes individual data points**, not features.

```python
d_model = 4
seq_len = 2
n_points = 3

torch.manual_seed(34)
data = torch.randn(n_points, seq_len, d_model)
pe = PositionalEncoding(seq_len, d_model)
inputs = pe(data)
inputs
```

Three sequences (N = 3)
Length of two (L = 2)
Features (D = 4)

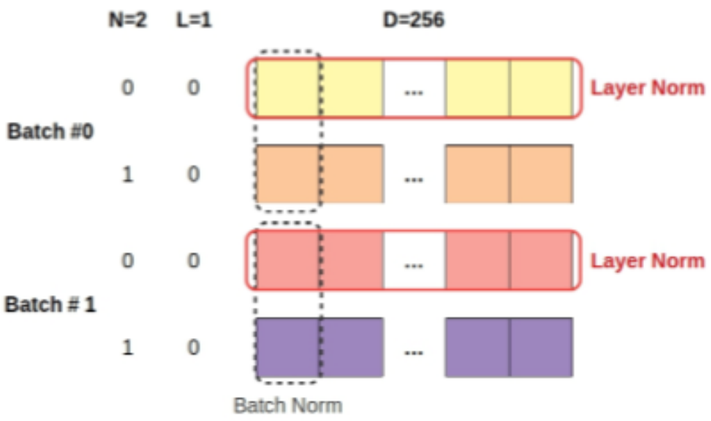
Output

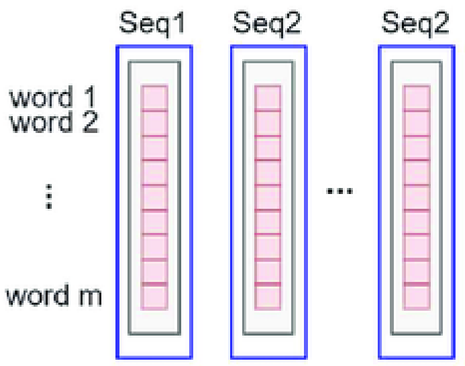
```
                1        2        3        4
tensor([[[-3.8049,  1.9899, -1.7325,  2.1359],   1
         [ 1.7854,  0.8155,  0.1116, -1.7420]],

        [[-2.4273,  1.3559,  2.8615,  2.0084],   2
         [-1.0353, -1.2766, -2.2082, -0.6952]],

1       [[-0.8044,  1.9707,  3.3704,  2.0587],   3
2        [ 4.2256,  6.9575,  1.4770,  2.0762]]])
```

→

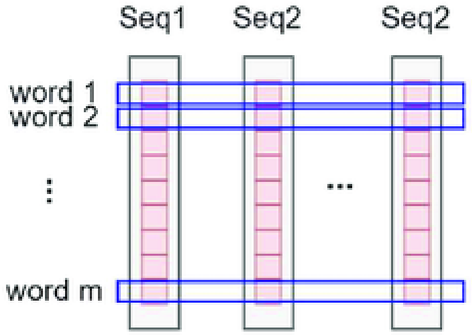
```
tensor([[[-1.3671,  0.9279, -0.5464,  0.9857],
         [ 1.1953,  0.4438, -0.1015, -1.5376]],

        [[-1.6706,  0.2010,  0.9458,  0.5238],
         [ 0.4782,  0.0485, -1.6106,  1.0839]],

        [[-1.6129,  0.2116,  1.1318,  0.2695],
         [ 0.2520,  1.5236, -1.0272, -0.7484]]])
```

Batch vs Layer





Layer Normalization (LN)

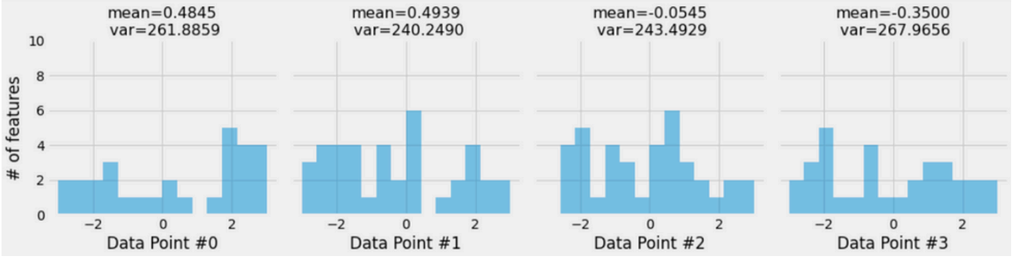


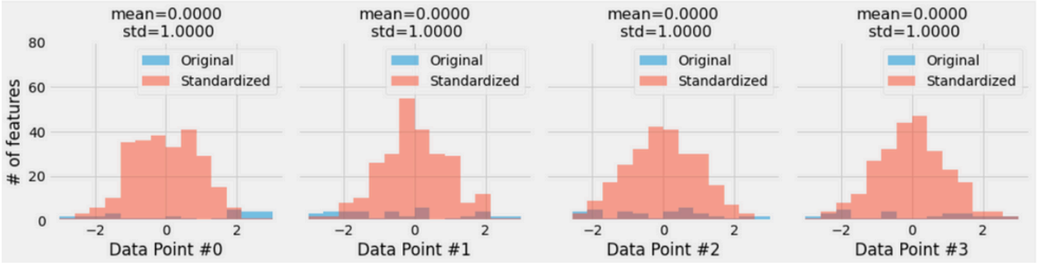
Batch Normalization (BN)

```
torch.manual_seed(23)
dummy_points = torch.randn(4, 1, 256)
dummy_pe = PositionalEncoding(1, 256)
dummy_enc = dummy_pe(dummy_points)
dummy_enc
```

X

```
layer_normalizer = nn.LayerNorm(256)
dummy_normed = layer_normalizer(dummy_enc)
dummy_normed
```





Batch vs Layer

Benefits of Layer Normalization:

- Improve stability of training and helps network learn faster;
- Alleviate the vanishing gradient problem;
- Make the network less sensitive to inital values of weights and biases;
- Improve performance.

Comparison with Batch Normalization:

- Normalization scope;
- Memory requirements;
- Performance

Drawbacks:

- Increase computational cost;
- Reduce expressiveness, can limit tha ability to learn certain represetations;
- Sensitivity to hyperparameters, tuning can be challeging;
- Can amplify noise data, leading to instability, small datasets or few layers;

Applications of Layer Normalization:

- Computer Vision;
- Natural Language Processing;
- Speech Recognition;
- Machine Translation;
- Reinforcement Learning.

Layer Normalization

- Layer normalization **standardizes individual data points**, not features.

Mean:

$$\bar{X}_{n,l} = \frac{1}{D}\sum_{d=1}^{D} x_{n,l,d}$$

```
inputs_mean = inputs.mean(axis=2).unsqueeze(2)
inputs_mean
```

Output

```
tensor([[[-0.3529],
         [ 0.2426]],

        [[ 0.9496],
         [-1.3038]],

        [[ 1.6489],
         [ 3.6841]]])
```

Biased standard deviation:

$$\sigma_{n,l}(X) = \sqrt{\frac{1}{D}\sum_{d=1}^{D}\left(x_{n,l,d} - \bar{X}_{n,l}\right)^2}$$

```
inputs_var = inputs.var(axis=2, unbiased=False).unsqueeze(2)
inputs_var
```

Output

```
tensor([[[6.3756],
         [1.6661]],

        [[4.0862],
         [0.3153]],

        [[2.3135],
         [4.6163]]])
```

Standardized

$$standardized x_{n,l,d} = \frac{x_{n,l,d} - \bar{X}_{n,l}}{\sigma_{n,l(X)+\varepsilon}}$$

```
(inputs - inputs_mean)/torch.sqrt(inputs_var+1e-5)
```

Output

```
tensor([[[-1.3671,  0.9279, -0.5464,  0.9857],
         [ 1.1953,  0.4438, -0.1015, -1.5376]],

        [[-1.6706,  0.2010,  0.9458,  0.5238],
         [ 0.4782,  0.0485, -1.6106,  1.0839]],

        [[-1.6129,  0.2116,  1.1318,  0.2695],
         [ 0.2520,  1.5236, -1.0272, -0.7484]]])
```

Layer normalized

Layer Normalization in PyTorch

nn.LayerNorm

```
layer_norm = nn.LayerNorm(d_model)
normalized = layer_norm(inputs)

normalized[0][0].mean(), normalized[0][0].std(unbiased=False)
```

```
(tensor(-1.4901e-08, grad_fn=<MeanBackward0>),
 tensor(1.0000, grad_fn=<StdBackward0>))
```

Zero mean and unit standard deviation, as expected.

🤔 Why do they have a **grad_fn** attribute?

Each feature has its own affine transformation.

```
layer_norm.state_dict()
```

```
OrderedDict([('weight', tensor([1., 1., 1., 1.])),
             ('bias', tensor([0., 0., 0., 0.]))])
```

Projections and Embeddings

- **Projections for numerical values** and e**mbeddings** for **categorical** values.

Cordinates

Tokens

Words

```
source_seq = torch.tensor([[[ 1.0349,  0.9661], [ 0.8055, -0.9169]]])
```

```
torch.manual_seed(11)
proj_dim = 6
linear_proj = nn.Linear(2, proj_dim)
pe = PositionalEncoding(2, proj_dim)

source_seq_proj = linear_proj(source_seq)
source_seq_proj_enc = pe(source_seq_proj)
source_seq_proj_enc
```

Output

```
tensor([[[-2.0934,  1.5040,  1.8742,  0.0628,  0.3034,  2.0190],
         [-0.8853,  2.8213,  0.5911,  2.4193, -2.5230,  0.3599]]],
       grad_fn=<AddBackward0>)
```

```
norm = nn.LayerNorm(proj_dim)
norm(source_seq_proj_enc)
```

Output

```
tensor([[[-1.9061,  0.6287,  0.8896, -0.3868, -0.2172,  0.9917],
         [-0.7362,  1.2864,  0.0694,  1.0670, -1.6299, -0.0568]]],
       grad_fn=<NativeLayerNormBackward0>)
```

The Transformer

Transformer

```python
class EncoderDecoderTransf(EncoderDecoderSelfAttn):
    def __init__(self, encoder, decoder, input_len, target_len, n_features):
        super(EncoderDecoderTransf, self).__init__(encoder, decoder, input_len, target_len)
        self.n_features = n_features
        self.proj = nn.Linear(n_features, encoder.d_model)
        self.linear = nn.Linear(encoder.d_model, n_features)
```

Transformer

Parent class

```
def encode(self, source_seq, source_mask=None):
    # Projection
    source_proj = self.proj(source_seq)
    encoder_states = self.encoder(source_proj, source_mask)
    self.decoder.init_keys(encoder_states)
```

```
def decode(self, shifted_target_seq, source_mask=None, target_mask=None):
    # Projection
    target_proj = self.proj(shifted_target_seq)
    outputs = self.decoder(target_proj,
                           source_mask=source_mask,
                           target_mask=target_mask)
```

```
    # Linear
    outputs = self.linear(outputs)
    return outputs
```

Transformer

Parent class

```python
class EncoderDecoderSelfAttn(nn.Module):
    def __init__(self, encoder, decoder, input_len, target_len):
        super(EncoderDecoderSelfAttn, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.subsequent_mask(self.target_len)
```

```python
    @staticmethod
    def subsequent_mask(size):
        attn_shape = (1, size, size)
        subsequence_mask = (1 - torch.triu(torch.ones(attn_shape), diagonal=1))
        return subsequent_mask
```

```python
def encode(self, source_seq, source_mask):
    encoder_states = self.encoder(source_seq, source_mask)
    self.decoder.init_keys(encoder_states)
```

```python
def decode(self, shifted_target_seq, source_mask=None, target_mask=None)
    outputs = self.decoder(shifted_target_seq,
                           source_mask=source_mask,
                           target_mask=target_mask)
    return outputs
```

Transformer

Parent class

```python
def predict(self, source_seq, source_mask):
    inputs = source_seq[:, -1:]
    for i in range(self.target_len):
        out = self.decode(inputs, source_mask, self.trg_masks[:, :i+1, :i+1])
        out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
        inputs = out.detach()
    outputs = inputs[:, 1:, :]
    return outputs
```

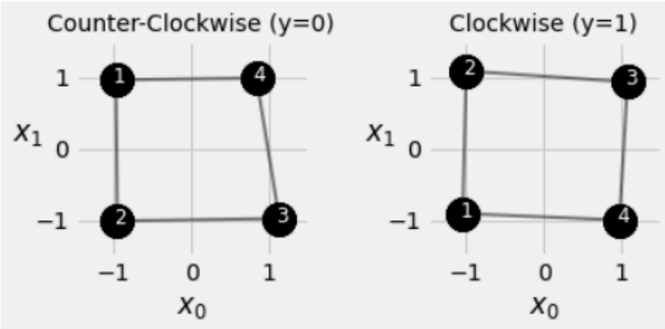
```python
def forward(self, X, source_mask=None):
    self.trg_masks = self.trg_masks.type_as(X).bool()
    source_seq = X[:, :self.input_len, :]

    self.encode(source_seq, source_mask)
    if self.training:
        shifted_target_seq = X[:, self.input_len-1:-1, :]
        outputs = self.decode(shifted_target_seq, source_mask, self.trg_masks)
    else:
        outputs = self.predict(source_seq, source_mask)

    return outputs
```

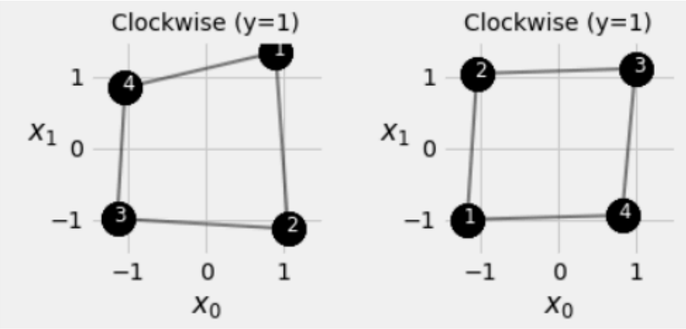
Transformer

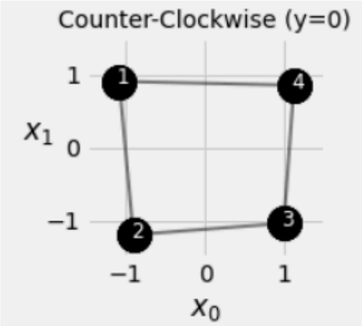
Data Preparation

```python
# Generating training data
points, directions = generate_sequences(n=256, seed=13)
full_train = torch.as_tensor(points).float()
target_train = full_train[:, 2:]
# Generating test data
test_points, test_directions = generate_sequences(seed=17)
full_test = torch.as_tensor(test_points).float()
source_test = full_test[:, :2]
target_test = full_test[:, 2:]
# Datasets and data loaders
train_data = TensorDataset(full_train, target_train)
test_data = TensorDataset(source_test, target_test)
generator = torch.Generator()
train_loader = DataLoader(train_data, batch_size=16, shuffle=True, generator=generator)
test_loader = DataLoader(test_data, batch_size=16)
```

```python
fig = plot_data(points, directions, n_rows=1)
```







Transformer

Model Configuration & Training

```
torch.manual_seed(42)
# Layers
enclayer = EncoderLayer(n_heads=3, d_model=6, ff_units=10, dropout=0.1)
declayer = DecoderLayer(n_heads=3, d_model=6, ff_units=10, dropout=0.1)
# Encoder and Decoder
enctransf = EncoderTransf(enclayer, n_layers=2)
dectransf = DecoderTransf(declayer, n_layers=2)
# Transformer
model_transf = EncoderDecoderTransf(enctransf, dectransf, input_len=2, target_len=2, n_features=2)
loss = nn.MSELoss()
optimizer = torch.optim.Adam(model_transf.parameters(), lr=0.01)
```

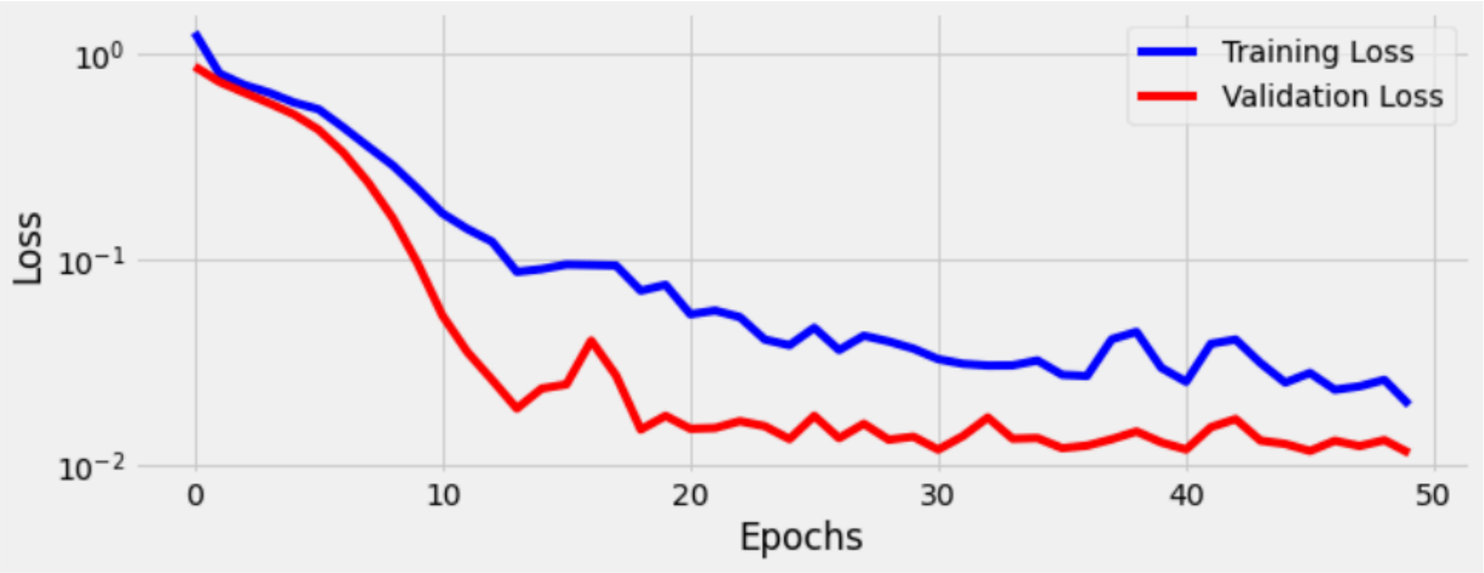
```
for p in model_transf.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

```
sbs_seq_transf = StepByStep(model_transf, loss, optimizer)
sbs_seq_transf.set_loaders(train_loader, test_loader)
sbs_seq_transf.train(50)
```

```
fig = sbs_seq_transf.plot_losses()
```

🤔

Why is the validation loss so much better than the training loss?



- Easier validation set to being a “side effect” of **regularization**
- The regularization makes it harder for the model to learn

Using the same mini-batch

```
torch.manual_seed(11)
x, y = next(iter(train_loader))
device = sbs_seq_transf.device
# Training
model_transf.train()
loss(model_transf(x.to(device)), y.to(device))
```

```
tensor(0.0158, device='cuda:0', grad_fn=<MseLossBackward0>)
```

```
# Validation
model_transf.eval()
loss(model_transf(x.to(device)), y.to(device))
```

```
tensor(0.0091, device='cuda:0')
```

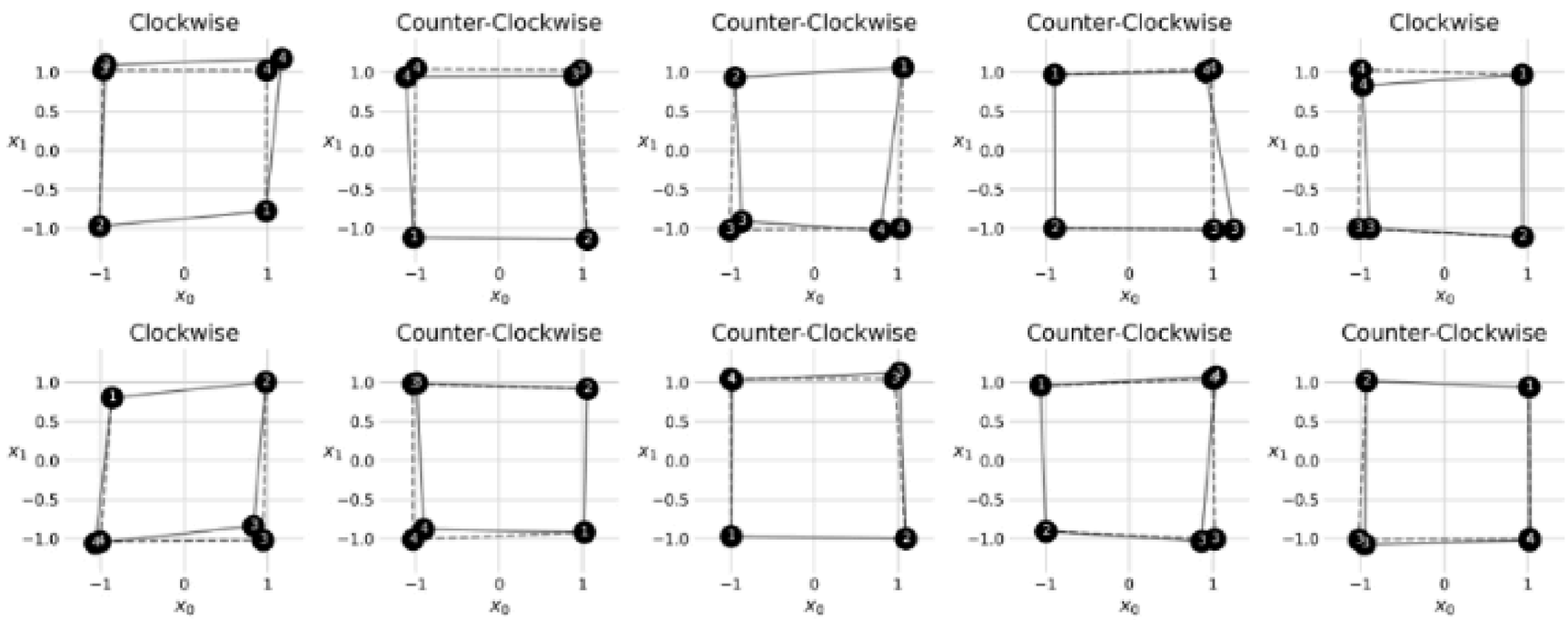
🤓 **Proposed solutions**

Set dropout to zero

Transformer

Visualizing Predictions

```
fig = sequence_pred(sbs_seq_transf, full_test, test_directions)
```



The PyTorch Transformer

Implements a full-fledged Transformer class of its own **nn.Transformer**

The PyTorch Transformer

```python
def encode_decode(self, source, target, source_mask=None, target_mask=None):
    # Projections
    # PyTorch Transformer expects L, N, F
    src = self.preprocess(source).permute(1, 0, 2)
    tgt = self.preprocess(target).permute(1, 0, 2)

    out = self.transf(src, tgt,
                      src_key_padding_mask=source_mask,
                      tgt_mask=target_mask)

    # Linear
    # Back to N, L, D
    out = out.permute(1, 0, 2)
    out = self.linear(out) # N, L, F
    return out
```

```python
class TransformerModel(nn.Module):
    def __init__(self, transformer, input_len, target_len, n_features):
        super().__init__()
        self.transf = transformer
        self.input_len = input_len
        self.target_len = target_len
        self.trg_masks = self.transf.generate_square_subsequent_mask(self.target_len)
        self.n_features = n_features
        self.proj = nn.Linear(n_features, self.transf.d_model)
        self.linear = nn.Linear(self.transf.d_model, n_features)

        max_len = max(self.input_len, self.target_len)
        self.pe = PositionalEncoding(max_len, self.transf.d_model)
        self.norm = nn.LayerNorm(self.transf.d_model)
```

The PyTorch Transformer

```python
def preprocess(self, seq):
    seq_proj = self.proj(seq)
    seq_enc = self.pe(seq_proj)
    return self.norm(seq_enc)
```

```python
def encode_decode(self, source, target, source_mask=None, target_mask=None):
    # Projections
    # PyTorch Transformer expects L, N, F
    src = self.preprocess(source).permute(1, 0, 2)
    tgt = self.preprocess(target).permute(1, 0, 2)

    out = self.transf(src, tgt,
                      src_key_padding_mask=source_mask,
                      tgt_mask=target_mask)

    # Linear
    # Back to N, L, D
    out = out.permute(1, 0, 2)
    out = self.linear(out) # N, L, F
    return out
```

```python
def predict(self, source_seq, source_mask=None):
    inputs = source_seq[:, -1:]
    for i in range(self.target_len):
        out = self.encode_decode(source_seq, inputs,
                                 source_mask=source_mask,
                                 target_mask=self.trg_masks[:i+1, :i+1])
        out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
        inputs = out.detach()
    outputs = out[:, 1:, :]
    return outputs
```

The PyTorch Transformer

```
def forward(self, X, source_mask=None):
    self.trg_masks = self.trg_masks.type_as(X)
    source_seq = X[:, :self.input_len, :]

    if self.training:
        shifted_target_seq = X[:, self.input_len-1:-1, :]
        outputs = self.encode_decode(source_seq, shifted_target_seq,
                                     source_mask=source_mask,
                                     target_mask=self.trg_masks)
    else:
        outputs = self.predict(source_seq, source_mask)

    return outputs
```

Model Configuration & Training

Model configuration

```
torch.manual_seed(42)
transformer = nn.Transformer(d_model=6, nhead=3,
                             num_encoder_layers=1, num_decoder_layers=1,
                             dim_feedforward=20, dropout=0.1)
model_transformer = TransformerModel(transformer, input_len=2, target_len=2, n_features=2)
loss = nn.MSELoss()
optimizer = torch.optim.Adam(model_transformer.parameters(), lr=0.01)
```

Weight Initialization

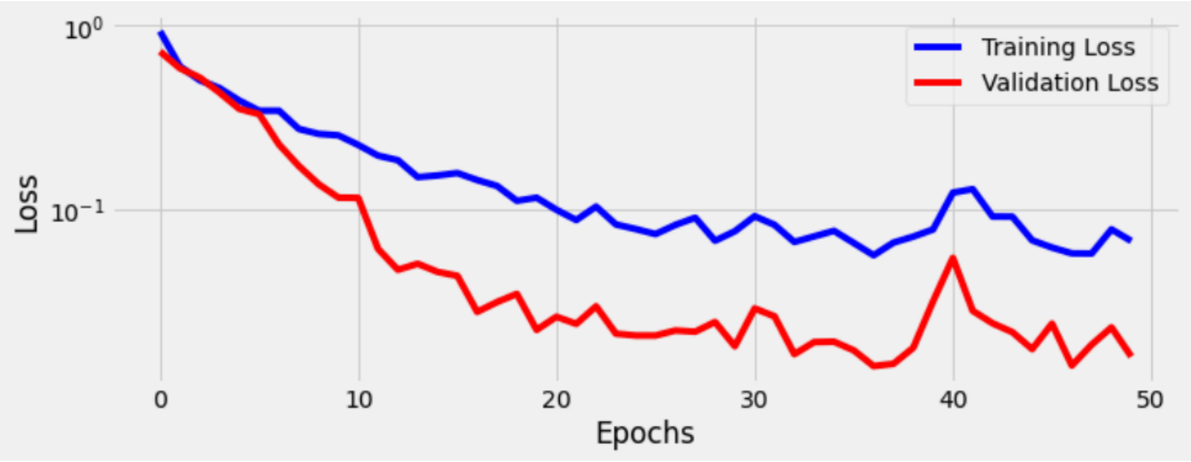
```
for p in model_transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

Model Training

```
sbs_seq_transformer = StepByStep(model_transformer, loss, optimizer)
sbs_seq_transformer.set_loaders(train_loader, test_loader)
sbs_seq_transformer.train(50)
```

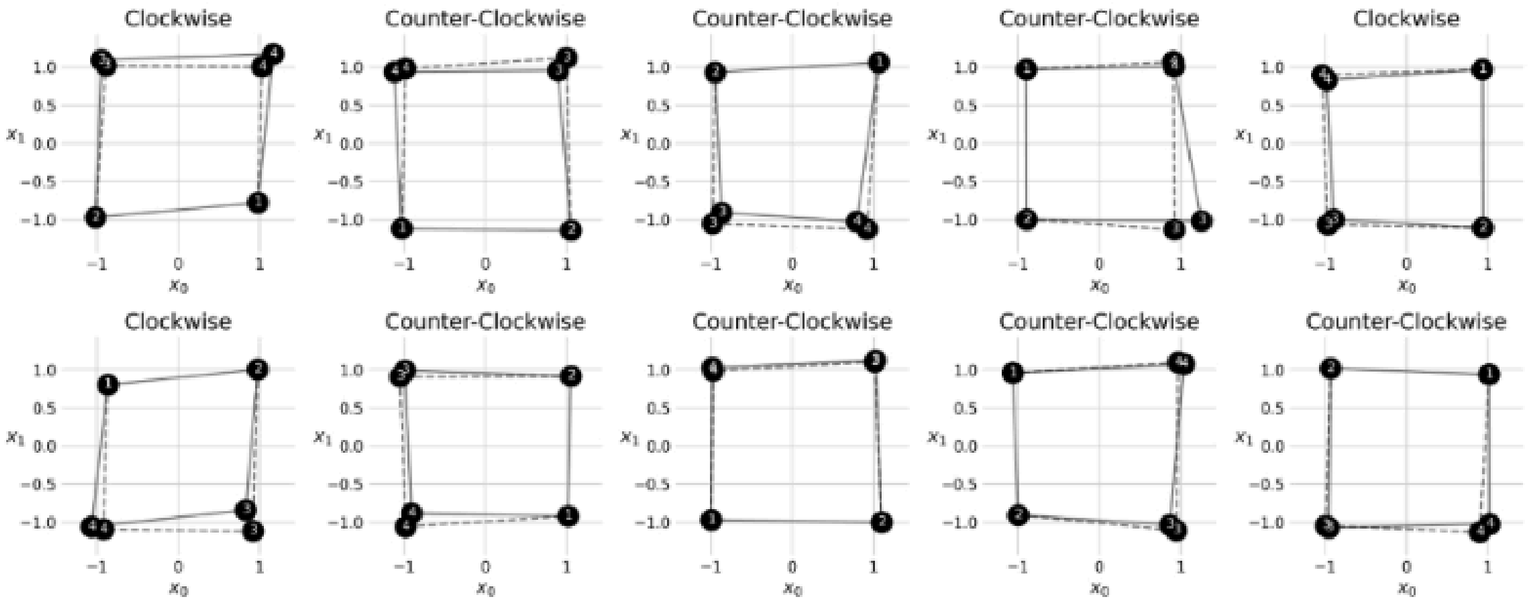
```
fig = sbs_seq_transformer.plot_losses()
```

Once again ...



The PyTorch Transformer

Visualizing Predictions



😏 Looking good, right?

Vision Transformer

😲 Cool, but I though the Transformer handled **sequences**, not images.

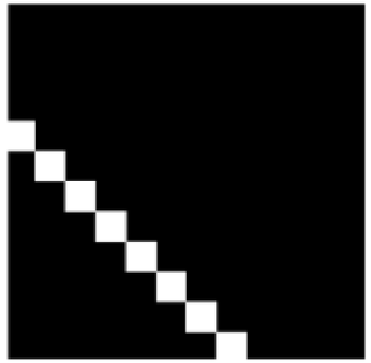
Let's **break an image** into a **sequence of patches**. 🙂

Data Generation & Preparation

```
images, labels = generate_dataset(img_size=12, n_images=1000, binary=False, seed=17)
```

```
img = torch.as_tensor(images[2]).unsqueeze(0).float()/255.
```

```
fig = plot_images(img, title=False)
```



But this is a **classification problem**, not a seq2seq one, why are we using a Transformer then?

Vision Transformer

Data preparation

```python
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform
```

```python
    def __getitem__(self, index):
        x = self.x[index]
        if self.transform:
            x = self.transform(x)
```

```python
        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

```python
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels).long()
```

```python
# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
# Uses indices to perform the split
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]
x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]
```

```python
# We're not doing any data augmentation now
train_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
```

```python
# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(x_train_tensor, y_train_tensor, transform=train_composer)
val_dataset = TransformedTensorDataset(x_val_tensor, y_val_tensor, transform=val_composer)
```

```python
# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)
```

```python
# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(dataset=train_dataset, batch_size=16, sampler=sampler)
val_loader = DataLoader(dataset=val_dataset, batch_size=16)
```

Vision Transformer

Patches

REARRANGING

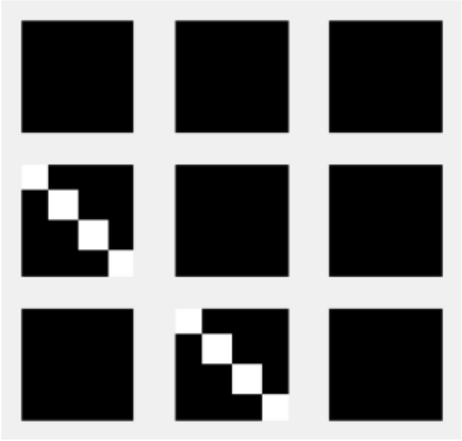
tf.image.extract_patches() ⟶ tensor.unfold()

```python
def extract_image_patches(x, kernel_size, stride=1):
    # Extract patches
    patches = x.unfold(2, kernel_size, stride)
    patches = patches.unfold(3, kernel_size, stride)
    patches = patches.permute(0, 2, 3, 1, 4, 5).contiguous()

    return patches.view(x.shape[0], patches.shape[1], patches.shape[2], -1)
```

```python
kernel_size = 4
patches = extract_image_patches(img, kernel_size, stride=kernel_size)
patches.shape
```

```
torch.Size([1, 3, 3, 16])
```

```python
fig = plot_patches(patches, kernel_size=kernel_size)
```



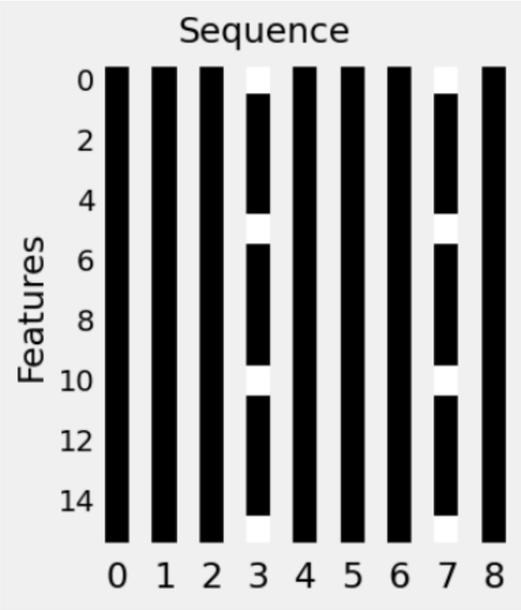
Vision Transformer

Patches
REARRANGING

🤓 Transformer needs a sequence of flattened patches.

```
seq_patches = patches.view(-1, patches.size(-1))
```

```
fig = plot_seq_patches(seq_patches)
```



- Sequence of **length nine**
- Each element in the sequence having **16 features**

Vision Transformer

Why not go **full convolution** then?

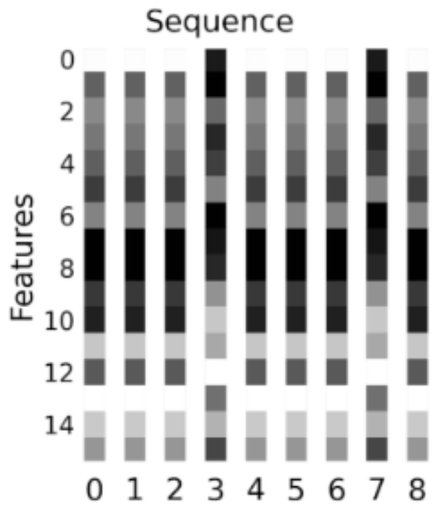
Patches

EMBEDDINGS

```python
class PatchEmbed(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_channels=3, embed_dim=768, dilation=1):
        super().__init__()
        num_patches = (img_size // patch_size) * (img_size // patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.proj = nn.Conv2d(in_channels, embed_dim, kernel_size=patch_size, stride=patch_size)

    def forward(self, x):
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

```python
torch.manual_seed(13)
patch_embed = PatchEmbed(img.size(-1), kernel_size, 1, kernel_size**2)
embedded = patch_embed(img)
embedded.shape
```

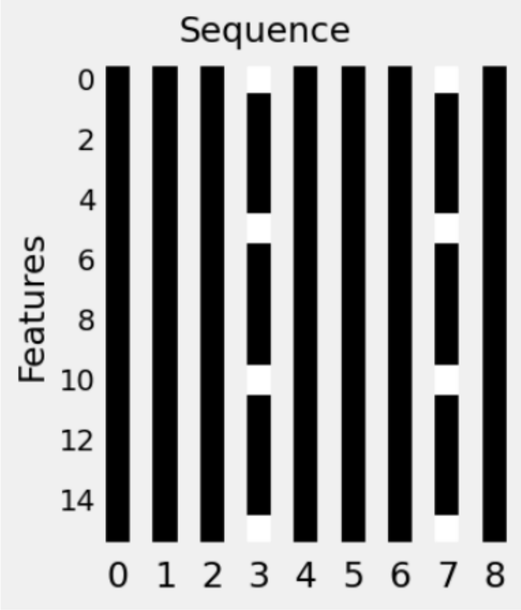
```
torch.Size([1, 9, 16])
```



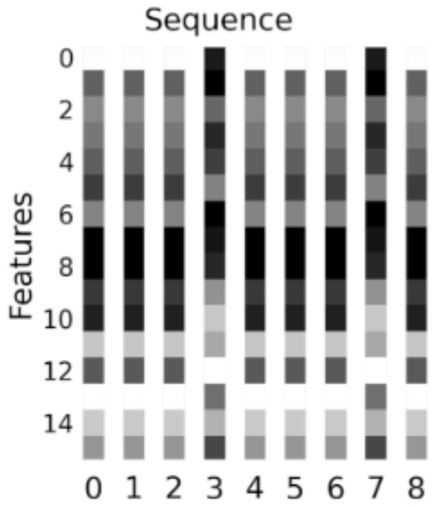
- Original image: **144 pixels**
- **9 patches of 16 pixels** each
- Each patch **embedding** has **16 dimensios**

Vision Transformer

Special Classifier Token



VS



🤨

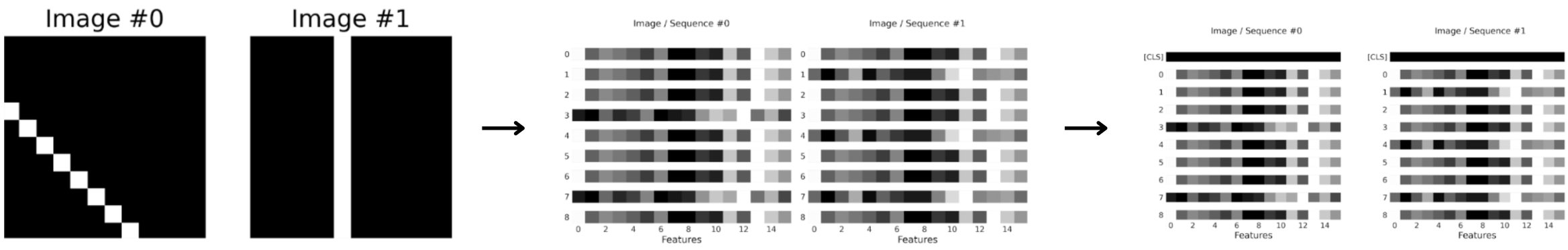
Can’t we use the **average** them?

R: Yes! But it is also common to use a **special classifier token (CLS)**

Vision Transformer

Special Classifier Token

- The **special classifier token** can actually learn which tokens (patches, in our case) it needs to pay attention to in order to correctly classify the sequence



How we do that?

Vision Transformer

Special Classifier Token

- Define a **parameter** in our model using nn.Parameter

```
cls_token = nn.Parameter(torch.zeros(1, 1, 16))
cls_token
```

```
Parameter containing:
tensor([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]],
       requires_grad=True)
```

```
images, labels = next(iter(train_loader))
images.shape # N, C, H, W
```

```
torch.Size([16, 1, 12, 12])
```

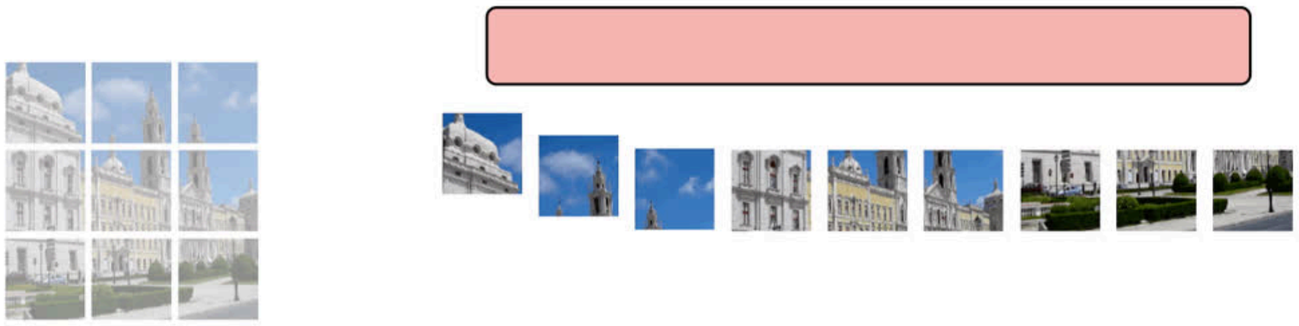
```
embed = patch_embed(images)
embed.shape # N, L, D
```

```
torch.Size([16, 9, 16])
```

```
cls_tokens = cls_token.expand(embed.size(0), -1, -1)
embed_cls = torch.cat((cls_tokens, embed), dim=1)
embed_cls.shape # N, L+1, D
```

```
torch.Size([16, 10, 16])
```

Vision Transformer



Let’s see it in code!

Vision Transformer

The Model

```python
class ViT(nn.Module):
    def __init__(self, encoder, img_size, in_channels, patch_size, n_outputs):
        super().__init__()
        self.d_model = encoder.d_model
        self.n_outputs = n_outputs
        self.encoder = encoder
        self.mlp = nn.Linear(encoder.d_model, n_outputs)

        self.embed = PatchEmbed(img_size, patch_size, in_channels, encoder.d_model)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, encoder.d_model))
```

```python
    def preprocess(self, X):
        # Patch embeddings
        # N, C, H, W -> N, L, D
        src = self.embed(X)
        # Special classifier token
        # 1, 1, D -> N, 1, D
        cls_tokens = self.cls_token.expand(X.size(0), -1, -1)
        # Concatenates CLS tokens -> N, 1 + L, D
        src = torch.cat((cls_tokens, src), dim=1)
        return src
```

```python
    def encode(self, source):
        # Encoder generates "hidden states"
        states = self.encoder(source)
        # Gets state from first token: CLS
        cls_state = states[:, 0]  # N, 1, D
        return cls_state
```

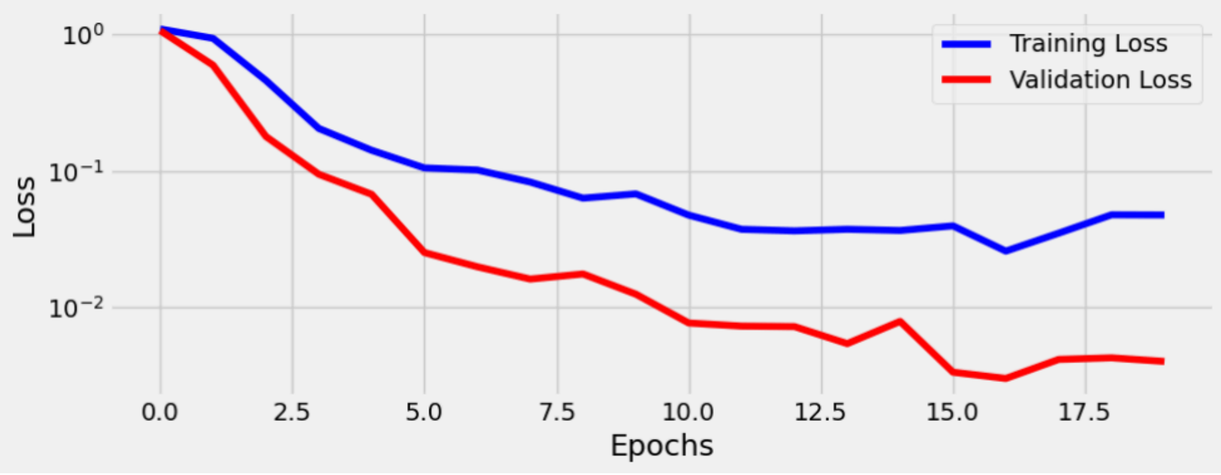
```python
    def forward(self, X):
        src = self.preprocess(X)
        # Featurizer
        cls_state = self.encode(src)
        # Classifier
        out = self.mlp(cls_state) # N, 1, outputs
        return out
```

Vision Transformer

Model Configuration & Training

```
torch.manual_seed(17)
layer = EncoderLayer(n_heads=2, d_model=16, ff_units=20)
encoder = EncoderTransf(layer, n_layers=1)
model_vit = ViT(encoder, img_size=12, in_channels=1, patch_size=4, n_outputs=3)
multi_loss_fn = nn.CrossEntropyLoss()
optimizer_vit = optim.Adam(model_vit.parameters(), lr=1e-3)
```

```
sbs_vit = StepByStep(model_vit, multi_loss_fn, optimizer_vit)
sbs_vit.set_loaders(train_loader, val_loader)
sbs_vit.train(20)
```




```
model_vit.cls_token
```

```
Parameter containing:
tensor([[[ 0.0590, -0.0471, -0.0053, -0.0251,  0.0360, -0.0394,  0.0585,
          -0.0263,  0.0189, -0.0419,  0.0033, -0.0117,  0.0961, -0.0380,
           0.0486, -0.0424]]], requires_grad=True)
```

```
StepByStep.loader_apply(sbs_vit.val_loader, sbs_vit.correct)
```

```
tensor([[76, 76],
        [65, 65],
        [59, 59]])
```

Nailed it!

Recap

- We’ve extended the **encoder-decoder architecture** and transfermed it into a **Transformer;**
- We modified the **multi-headed attention** to use **narrow** attention;
- We introduced **layer normalization;**
- Change the dimensionality of the inputs using **projections** or **embeddings;**
- The layers were stacked to form the new **Transformer encoder and decoder;**
- That made our model **much deeper**, thus raising the need for wrapping the internal operations (**“sub-layers”**), of each “layer” with a combination of **layer normalization**, **dropout** and **residual connection**.

Thanks!