LLMs from Dummies: Part 2

Building Language Models

In this session, we will move away from the “Translation” example and build a “Language Model”

What we are doing today:

1. Build, code, and train a language model
2. Create “boiler-plate” code to train, test, and use the LM.
3. Start with the concept of an “Attention Head”
4. Create a “Transformer Block”
5. Stack the block to create a “Transformer-like” architecture.
6. Add extra components to help us scale the network

References:

- NanoGPT (https://github.com/karpathy/nanoGPT)
- “Let's build GPT: from scratch, in code, spelled out” (https://www.youtube.com/watch?v=kCc8FmEb1nY)

Language Model

Language Model

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

Language Model

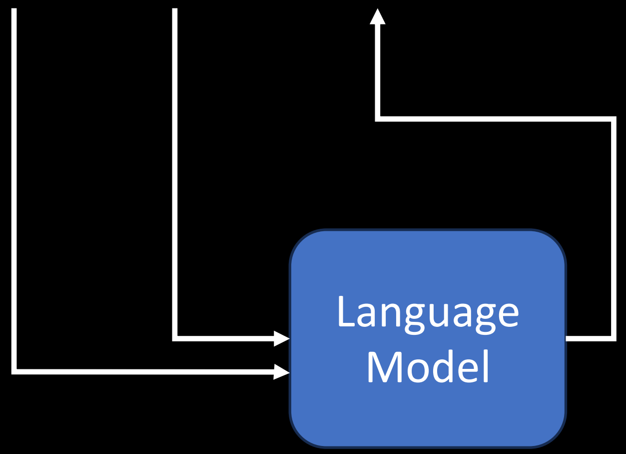
Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

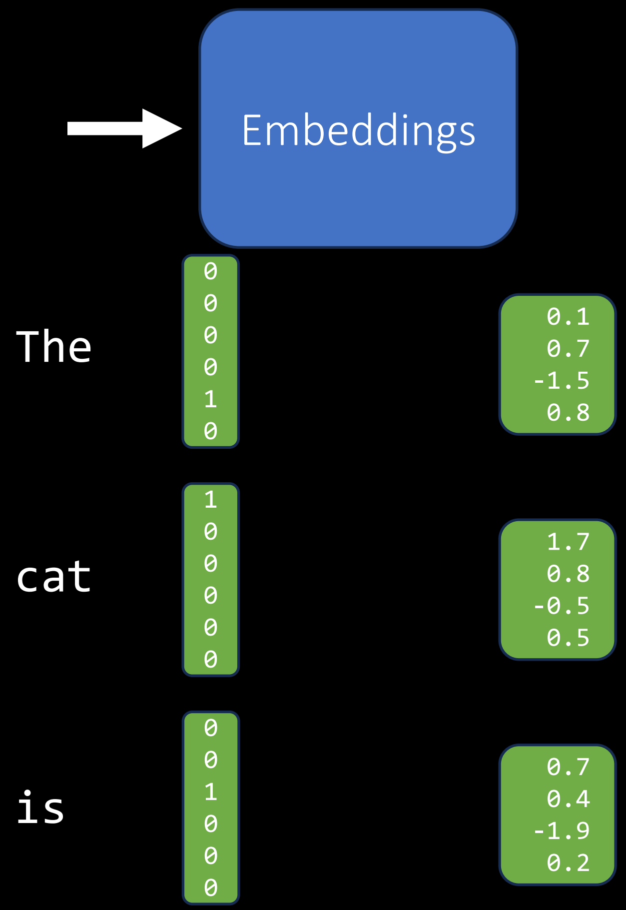
input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

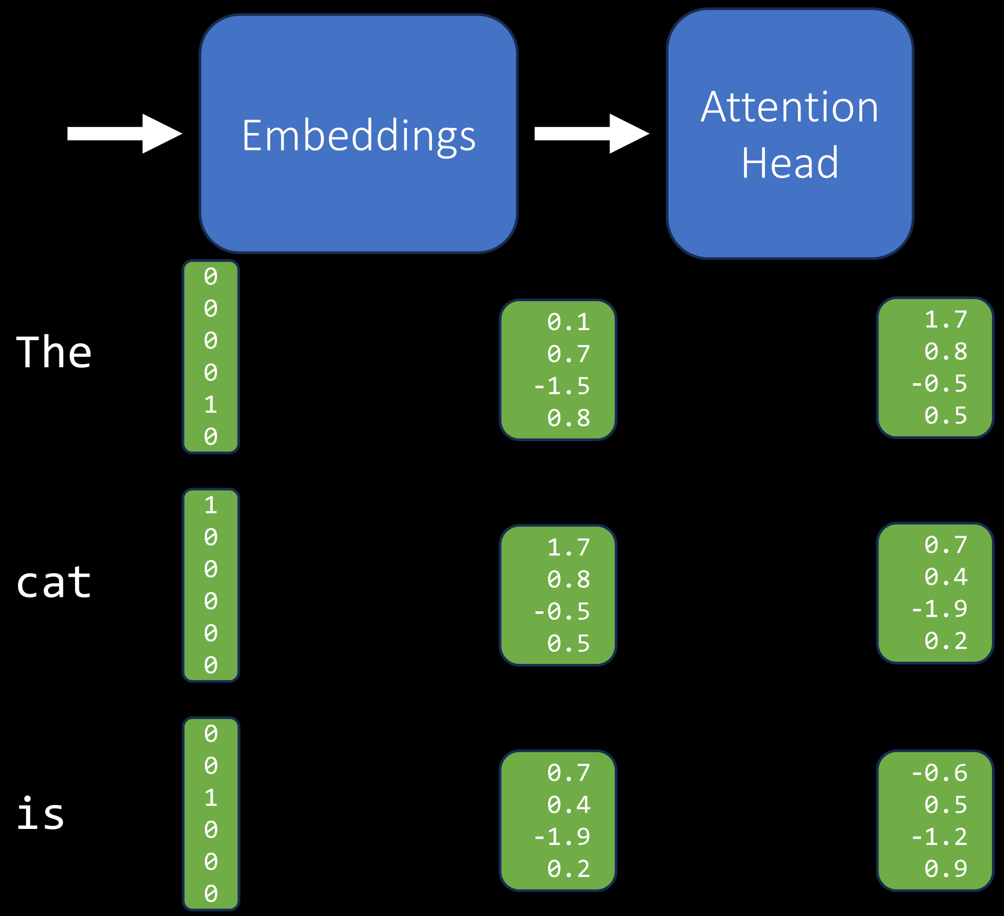


The

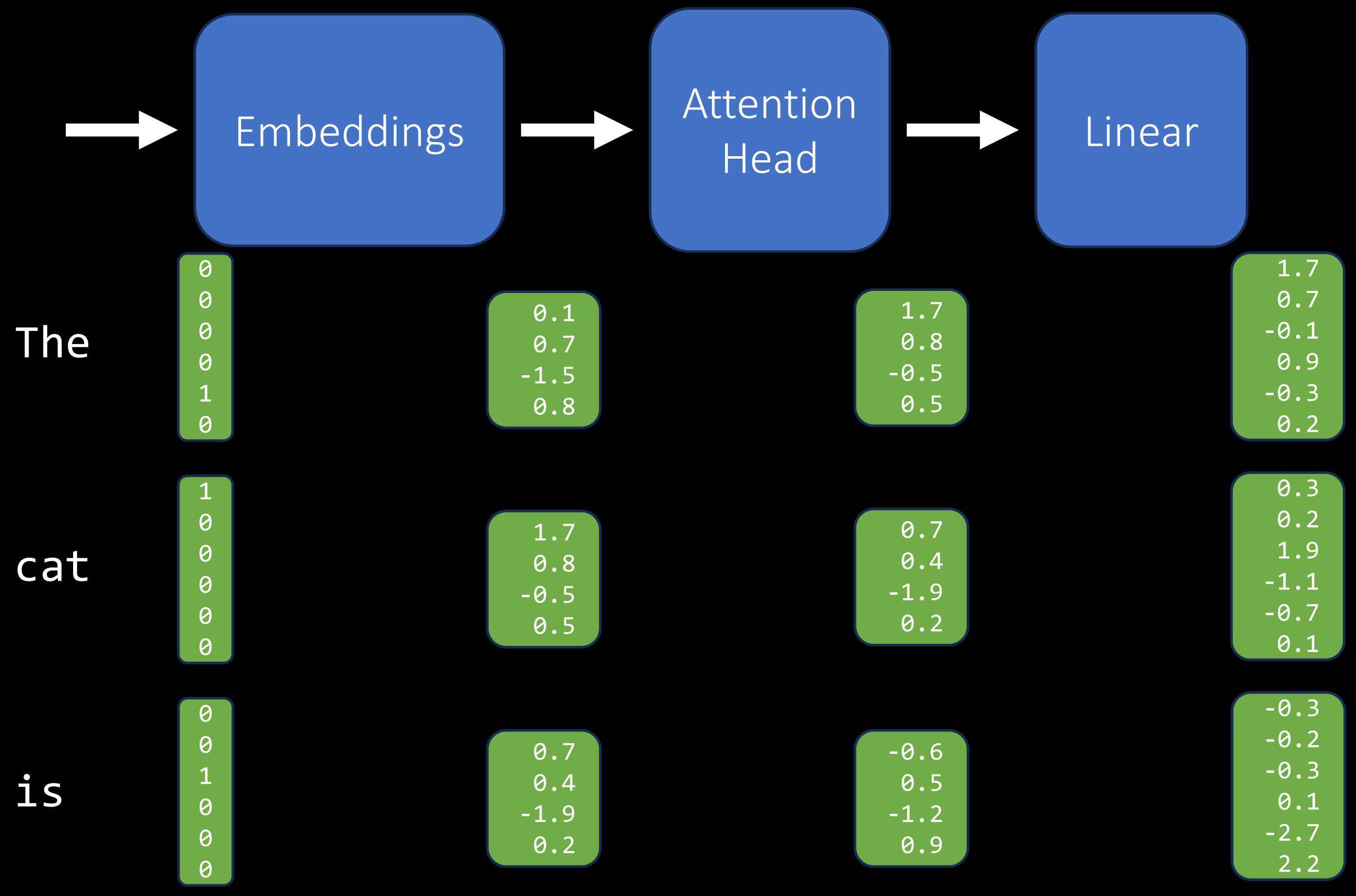
cat

is

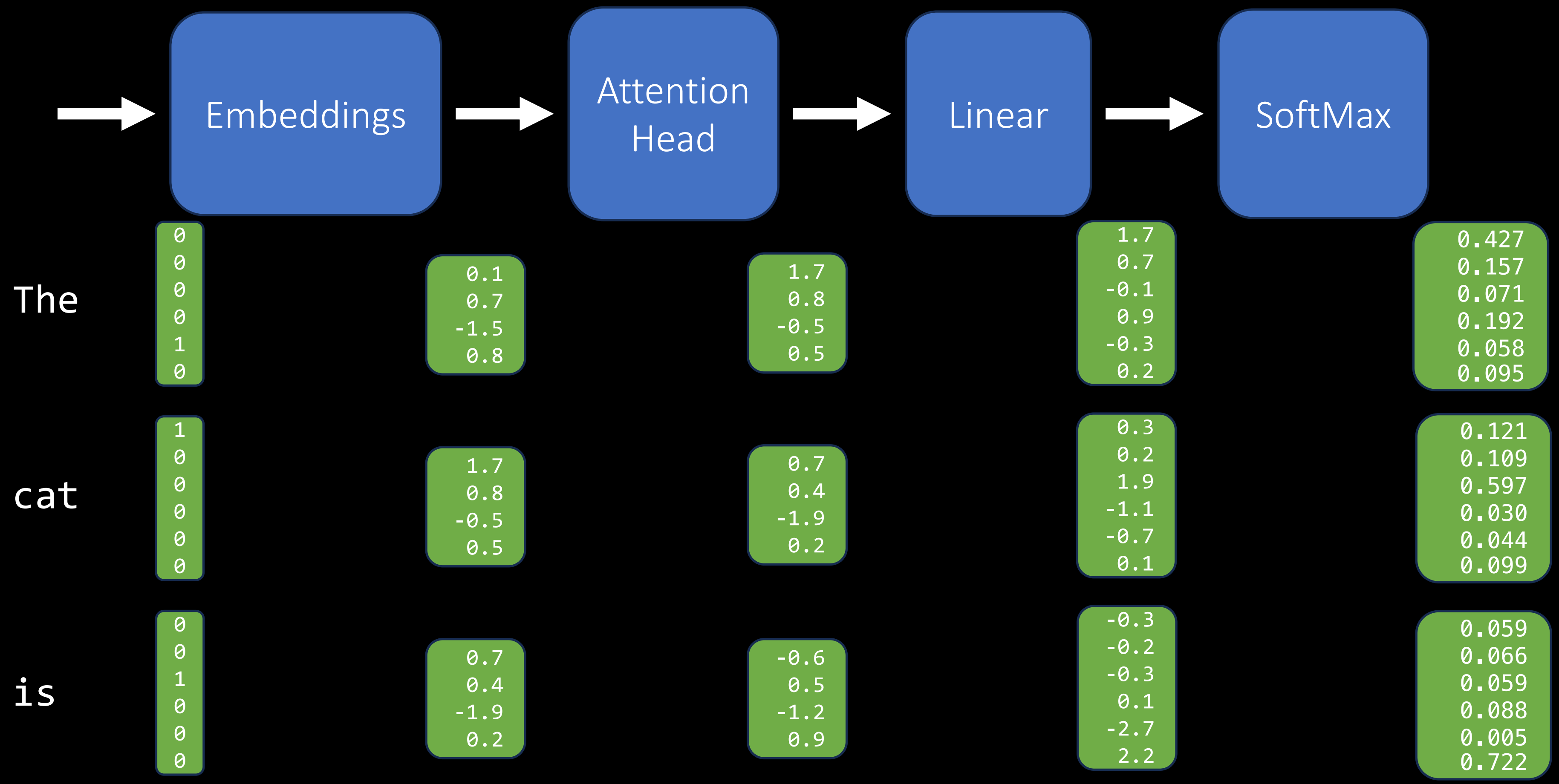

Embeddings
The
0 0 0 0 1 0
0.1 0.7 -1.5 0.8
cat
1 0 0 0 0 0
1.7 0.8 -0.5 0.5
is
0 0 1 0 0 0
0.7 0.4 -1.9 0.2

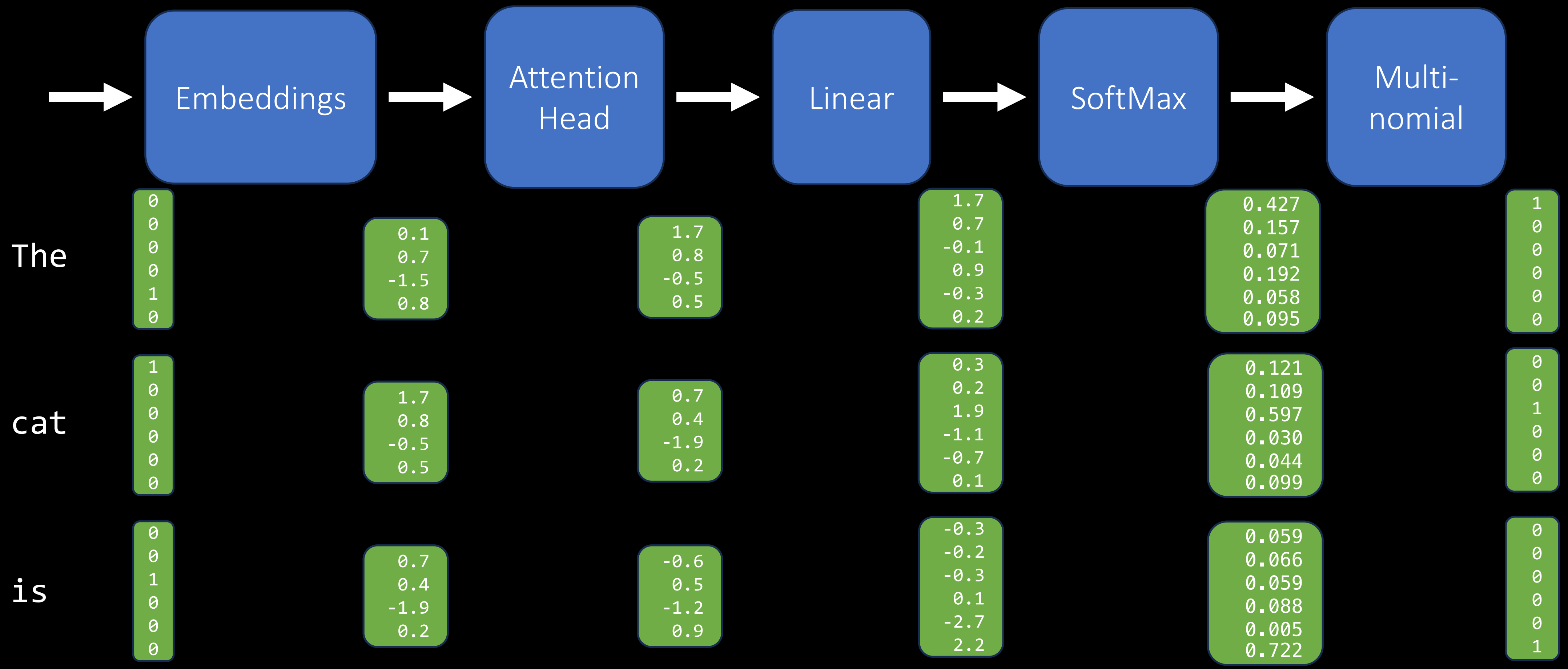


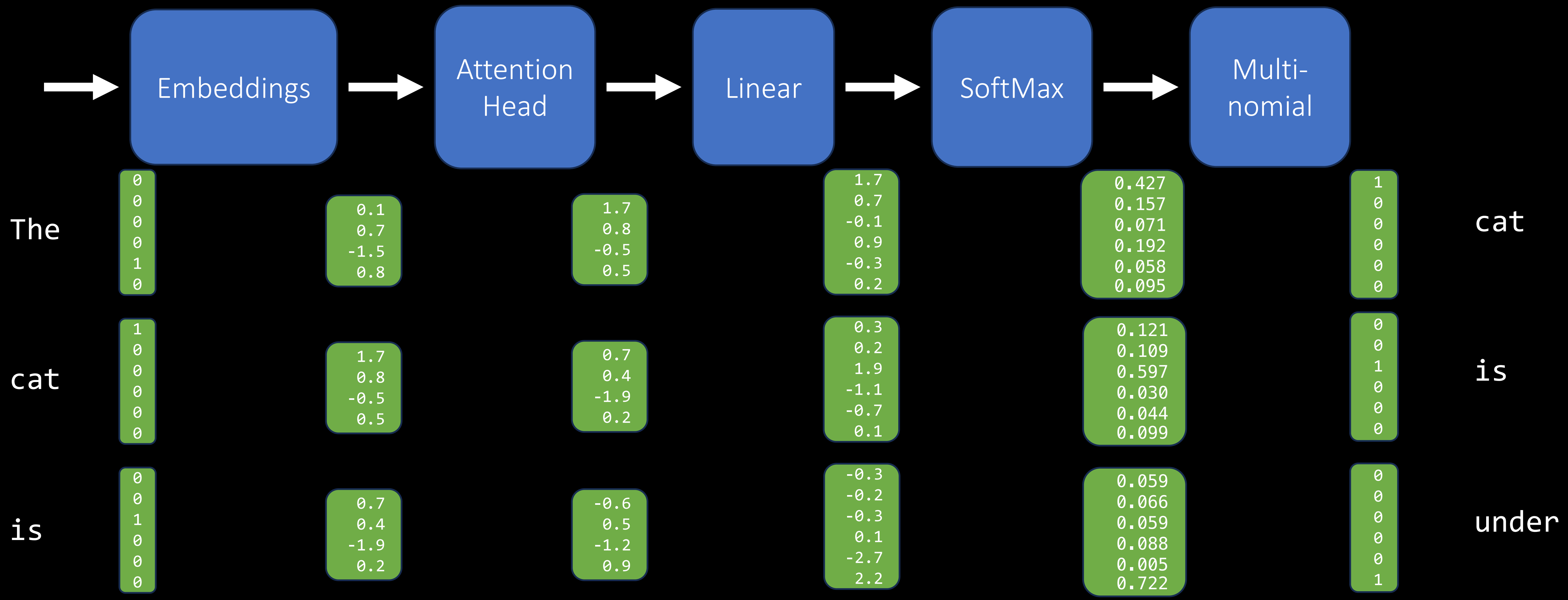
Embeddings
Attention Head
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9

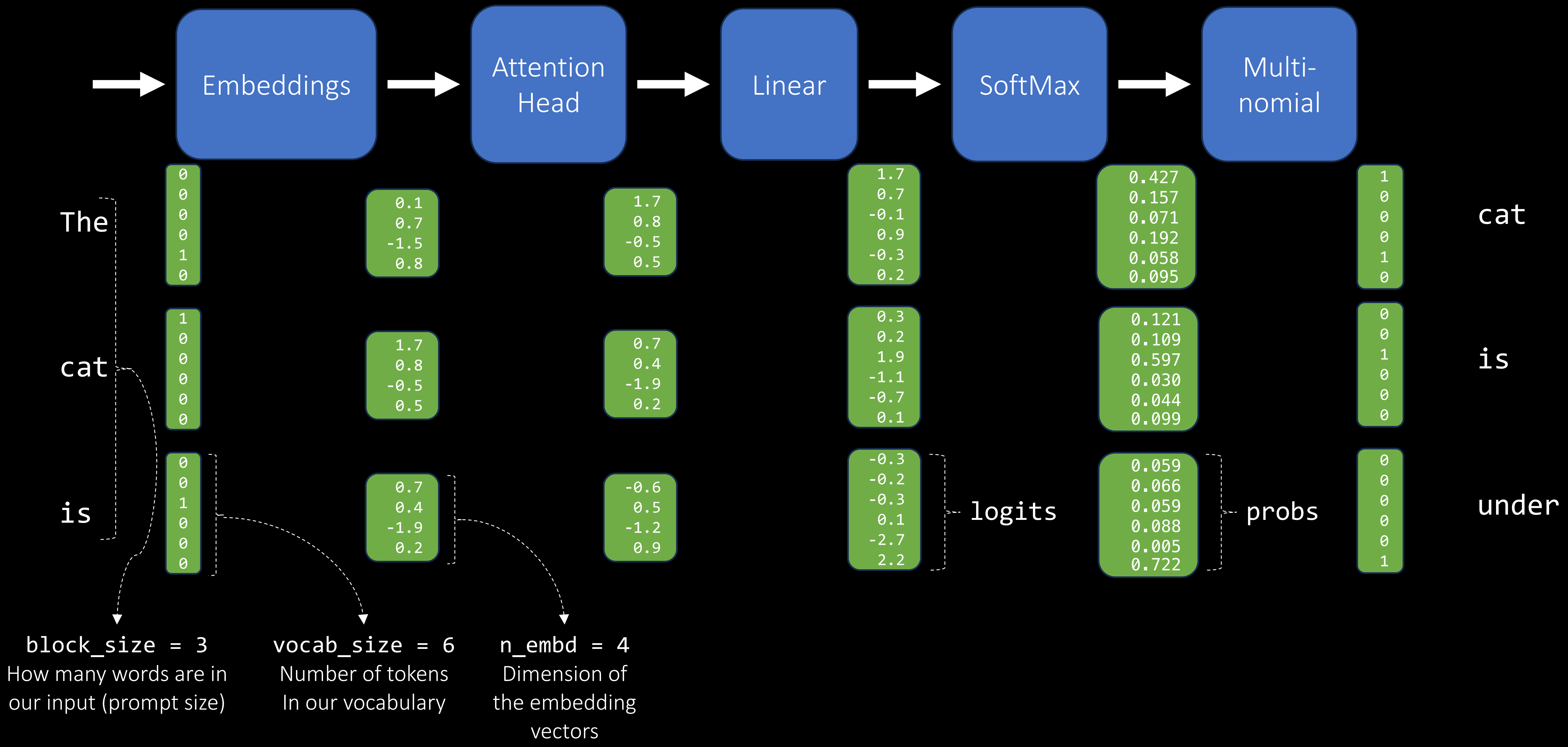


Embeddings
Attention Head
Linear
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2


Embeddings
Attention Head
Linear
SoftMax
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2
0.427 0.157 0.071 0.192 0.058 0.095
0.121 0.109 0.597 0.030 0.044 0.099
0.059 0.066 0.059 0.088 0.005 0.722


Embeddings
Attention Head
Linear
SoftMax
Multi-nomial
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2
0.427 0.157 0.071 0.192 0.058 0.095
0.121 0.109 0.597 0.030 0.044 0.099
0.059 0.066 0.059 0.088 0.005 0.722
1 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 1


Embeddings
Attention Head
Linear
SoftMax
Multi-nomial
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2
0.427 0.157 0.071 0.192 0.058 0.095
0.121 0.109 0.597 0.030 0.044 0.099
0.059 0.066 0.059 0.088 0.005 0.722
1 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 1
cat
is
under


Embeddings
Attention Head
Linear
SoftMax
Multi-nomial
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2
logits
0.427 0.157 0.071 0.192 0.058 0.095
0.121 0.109 0.597 0.030 0.044 0.099
0.059 0.066 0.059 0.088 0.005 0.722
probs
1 0 0 0 1 0
0 0 1 0 0 0
0 0 0 0 0 1
cat
is
under
block_size = 3
How many words are in our input (prompt size)
vocab_size = 6
Number of tokens In our vocabulary
n_embd = 4
Dimension of the embedding vectors

Language Model 1

```python
class LanguageModel(nn.Module):
  """ Multi-headed attention model """
  def __init__(self):
    super().__init__()
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.head = Head()
    self.lm_head = nn.Linear(n_embd, vocab_size)

  def forward(self, idx, targets=None):
    x = self.token_embedding_table(idx)
    x = self.head(x)
    logits = self.lm_head(x)
    if targets is None:
      loss = None
    else:
      # Calculate loss
      b, t, c = logits.shape
      logits = logits.view(b*t, c)
      targets = targets.view(b*t)
      loss = F.cross_entropy(logits, targets)
    return logits, loss
```

Language Model 1

```python
class LanguageModel(nn.Module):
    """ Multi-headed attention model """
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.head = Head()
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        x = self.token_embedding_table(idx)
        x = self.head(x)
        logits = self.lm_head(x)
        if targets is None:
            loss = None
        else:
            # Calculate loss
            b, t, c = logits.shape
            logits = logits.view(b*t, c)
            targets = targets.view(b*t)
            loss = F.cross_entropy(logits, targets)
        return logits, loss
```

```python
class Head(nn.Module):
    """ Self attention head """

    def __init__(self):
        super().__init__()
        self.key = nn.Linear(n_embd, n_embd, bias=False)
        self.query = nn.Linear(n_embd, n_embd, bias=False)
        self.value = nn.Linear(n_embd, n_embd, bias=False)

    def forward(self, x):
        k = self.key(x)
        q = self.query(x)
        v = self.value(x)
        # Attention score
        w = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
        w = F.softmax(w, dim=-1)
        # Add weighted values
        out = w @ v
        return out
```

Boilerplate code

Boilerplate code: Tokenizer (ASCII chars)

```python
class TrivialTokenizer:
  """ Trivial tokenizer: Converts to chars """

  def decode(self, tokens_encoded):
    return ''.join([self.itos[i.item()] for i in tokens_encoded])

  def encode(self, text):
    encoded_tokens = [self.stoi[c] for c in text]
    return torch.tensor(encoded_tokens, dtype=torch.long)

  def train(self, text):
    chars = sorted(list(set(text)))
    self.vocab_size = len(chars)
    self.stoi = {ch:i for i,ch in enumerate(chars)}
    self.itos = {i:ch for i,ch in enumerate(chars)}

  def vocabulary_size(self):
    return self.vocab_size
```

Boilerplate code: Tokenizer (ASCII chars)

```python
class TrivialTokenizer:
  """ Trivial tokenizer: Converts to chars """

  def decode(self, tokens_encoded):
    return ''.join([self.itos[i.item()] for i in tokens_encoded])

  def encode(self, text):
    encoded_tokens = [self.stoi[c] for c in text]
    return torch.tensor(encoded_tokens, dtype=torch.long)

  def train(self, text):
    chars = sorted(list(set(text)))
    self.vocab_size = len(chars)
    self.stoi = {ch:i for i,ch in enumerate(chars)}
    self.itos = {i:ch for i,ch in enumerate(chars)}

  def vocabulary_size(self):
    return self.vocab_size
```

```python
tokenizer = TrivialTokenizer()
tokenizer.train(text)
print(tokenizer.encode('hi there'))
print(tokenizer.decode(tokenizer.encode('hi there')))
```

```
tensor([46, 47,  1, 58, 46, 43, 56, 43])
hi there
```

Boilerplate code: Dataset

```
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
[redacted]

    def get_batch(self, split):
[redacted]

    def load(self):
[redacted]

    def split(self):
[redacted]

    def tokenize(self):
[redacted]
```

Boilerplate code: Dataset

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        self.file_name = file_name
        self.cut = cut                 # Percentage for training / validation
        self.data = None               # Tokenized text data
        self.data_train = None         # Training data split
        self.data_validation = None # Validation data split
        self.text = None               # Raw text data

    def get_batch(self, split):
        [redacted]

    def load(self):
        [redacted]

    def split(self):
        [redacted]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        [redacted]

    def load(self):
        """ Read dataset (i.e. text file) """
        with open(self.file_name, 'r', encoding='utf-8') as f:
            self.text = f.read()
        return self.text

    def split(self):
        [redacted]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us...
```

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        [redacted]

    def load(self):
        """ Read dataset (i.e. text file) """
        with open(self.file_name, 'r', encoding='utf-8') as f:
            self.text = f.read()
        return self.text

    def split(self):
        [redacted]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```
class TextDataset:
  """ Create a 'text' dataset for training and testing. """
  def __init__(self, file_name, cut = 0.8, split='train'):
    [redacted]

  def get_batch(self, split):
    [redacted]

  def load(self):
    [redacted]

  def split(self):
    [redacted]

  def tokenize(self):
    """ Tokenize the text data """
    self.data = tokenizer.encode(self.text)
```

Boilerplate code: Dataset

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        [redacted]
```

```python
print(dataset.text[:50])
dataset.data[:50]
```

```
First Citizen:
Before we proceed any further, hear
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56])
```

```python
    def tokenize(self):
        """ Tokenize the text data """
        self.data = tokenizer.encode(self.text)
```

Boilerplate code: Dataset

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        [redacted]

    def load(self):
        [redacted]

    def split(self):
        """ Split dataset into training and validation """
        cut_len = int(self.cut * len(self.data))
        self.data_train = self.data[:cut_len]
        self.data_validation = self.data[cut_len:]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        """ Create a batch of data from either the train or validation split """
        data = self.data_train if split == 'train' else self.data_validation
        ix = torch.randint(len(data) - block_size, (batch_size,))
        x = torch.stack([data[i:i+block_size] for i in ix])
        y = torch.stack([data[i+1:i+block_size+1] for i in ix])
        x, y = x.to(device), y.to(device)
        return x, y

    def load(self):
        [redacted]

    def split(self):
        [redacted]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```python
# Example of getting a batch
dataset.get_batch('train')
```

```
vocab_size: 65
(tensor([[19, 53,  6,  ...,  1, 59, 54],
         [52,  1, 51,  ...,  0,  0, 15],
         [63,  1, 40,  ..., 46,  1, 53],
         ...,
         [10,  0, 26,  ..., 43, 58, 58],
         [ 6,  1, 58,  ...,  1, 52, 47],
         [ 0, 20, 13,  ...,  1, 47, 57]]),
 tensor([[53,  6,  1,  ..., 59, 54,  1],
         [ 1, 51, 63,  ...,  0, 15, 24],
         [ 1, 40, 56,  ...,  1, 53, 59],
         ...,
         [ 0, 26, 53,  ..., 58, 58, 43],
         [ 1, 58, 46,  ..., 52, 47, 45],
         [20, 13, 31,  ..., 47, 57,  8]]))
```

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        [redacted]

    def get_batch(self, split):
        """ Create a batch of data from either the train or validation split """
        data = self.data_train if split == 'train' else self.data_validation
        ix = torch.randint(len(data) - block_size, (batch_size,))
        x = torch.stack([data[i:i+block_size] for i in ix])
        y = torch.stack([data[i+1:i+block_size+1] for i in ix])
        x, y = x.to(device), y.to(device)
        return x, y

    def load(self):
        [redacted]

    def split(self):
        [redacted]

    def tokenize(self):
        [redacted]
```

Boilerplate code: Dataset

```python
class TextDataset:
    """ Create a 'text' dataset for training and testing. """
    def __init__(self, file_name, cut = 0.8, split='train'):
        self.file_name = file_name
        self.cut = cut                  # Percentage for training / validation
        self.data = None                # Tokenized text data
        self.data_train = None          # Training data split
        self.data_validation = None # Validation data split
        self.text = None                # Raw text data

    def get_batch(self, split):
        """ Create a batch of data from either the train or validation split """
        data = self.data_train if split == 'train' else self.data_validation
        ix = torch.randint(len(data) - block_size, (batch_size,))
        x = torch.stack([data[i:i+block_size] for i in ix])
        y = torch.stack([data[i+1:i+block_size+1] for i in ix])
        x, y = x.to(device), y.to(device)
        return x, y

    def load(self):
        """ Read dataset (i.e. text file) """
        with open(self.file_name, 'r', encoding='utf-8') as f:
            self.text = f.read()
        return self.text

    def split(self):
        """ Split dataset into training and validation """
        cut_len = int(self.cut * len(self.data))
        self.data_train = self.data[:cut_len]
        self.data_validation = self.data[cut_len:]

    def tokenize(self):
        """ Tokenize the text data """
        self.data = tokenizer.encode(self.text)
```

Boilerplate code: Training loop

```python
def train(model, dataset):
    # Train the model
    model.train()
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
    # Create a training loop
    for step in range(max_iters):
        # Sample batch data
        xb, yb = dataset.get_batch('train')
        # Evaluate model
        logits, loss = model(xb, yb)
        # Learn
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
    model.eval()
```

Boilerplate code: Generating

```python
def generate(model, dataset, prompt=None, max_new_tokens=500):
    """ Run a generation and show the output """
    model.eval()
    # Create a 'prompt'
    if prompt is None:
        prompt_encoded = torch.zeros((1, 1), dtype=torch.long)
    else:
        prompt_encoded = tokenizer.encode(prompt).unsqueeze(0)
    # Run the model on the prompt, predicting one word at a time
    tokens = []
    for _ in range(max_new_tokens):
        # Prepare the model's input
        prompt_encoded = prompt_encoded.to(device)
        prompt_encoded_crop = prompt_encoded[:, -block_size:]
        # Use the model to predict the next token
        logits, _ = model(prompt_encoded_crop)
        logits = logits[:, -1, :]
        probs = F.softmax(logits, dim=-1)
        next_token_encoded = torch.multinomial(probs, num_samples=1)
        # Decode and update output tokens
        print(tokenizer.decode(next_token_encoded), end='', flush=True)
        # Update the prompt by appending the next token
        prompt_encoded = torch.cat((prompt_encoded, next_token_encoded), dim=1)
```

Let’s train and generate our first Language Model

```
%%time
model = LanguageModel()
train_and_generate(model)
```

```
Before training:
---
SX;FS'CYWavScA!TeO$ehp-osN cU,SGza;AwI
V nR.G!EaneEXmE3LKzmz3!:UBtr!uatiKpJK!gqAmyWIarQ-b;Cj3nhmo:P!fGIuwrVhFNKy&q33DpPsrp3:v!-UzTTTRjpfo,rAkoJ-'.3.W;S;wOXTHss3x;jVA
M lOD;E3wmh l&kdpiu;v!ZREF'ZeUGFUFXNWMZ ythgoWW$Jcx!rnaSuNGR:Alek.;u;Q&DIKAqWI-uX:pa,bh,M;eO3?OjNOPRW,d3,df
q;E;b
ES&!A,ZK'LpCPSs-C
zwqdrqiV&MIdDV3B-,;n.bFiHnAU3Mj,,!MNWlTJ'FSc  CMBmKaNQxnojFnv QF nwsO,Nly-C
;UsUJcRpZZH-'.JBHTGjGDJbkRaG'c
-R.qQ;$aFtbdSyKDK-hMXYAXyRgrCdccfdwAeJUQ!R:CvY  M.G?uLD,?ANzMSlD,u;!!aFVZW$t?RUVzhKPbhcUepOnB'Z
---

Training model: 0.007297M parameters
Step 0: train loss 4.1351, val loss 4.1377
Step 200: train loss 3.1912, val loss 3.2174
Step 400: train loss 2.9819, val loss 3.0040
```

Let’s train and generate our first Language Model

```
%%time
model = LanguageModel()
train_and_generate(model)
```

```
Before training:
---
SX;FS'CYWavScA!TeO$ehp-osN cU,SGza;AwI
V nR.G!EaneEXmE3LKzmz3!:UBtr!uatiKpJK!gqAmyWIarQ-b;Cj3nhmo:P!fGIuwrVhFNKy&q33DpPsrp3:v!-UzTTTRjpfo,rAkoJ-'.3.W;S;wOXTHss3x;jVA
M lOD;E3wmh l&kdpiu;v!ZREF'ZeUGFUFXNWMZ ythgoWW$Jcx!rnaSuNGR:Alek.;u;Q&DIKAqWI-uX:pa,bh,M;eO3?OjNOPRW,d3,df
q;E;b
ES&!A,ZK'LpCPSs-C
zwqdrqiV&MIdDV3B-,;n.bFiHnAU3Mj,,!MNWlTJ'FSc  CMBmKaNQxnojFnv QF nwsO,Nly-C
;UsUJcRpZZH-'.JBHTGjGDJbkRaG'c
-R.qQ;$aFtbdSyKDK-hMXYAXyRgrCdccfdwAeJUQ!R:CvY  M.G?uLD,?ANzMSlD,u;!!aFVZW$t?RUVzhKPbhcUepOnB'Z
---

Training model: 0.007297M parameters
Step 0: train loss 4.1351, val loss 4.1377
Step 200: train loss 3.1912, val loss 3.2174
Step 400: train loss 2.9819, val loss 3.0040
```

```
Step 4800: train loss 2.1687, val loss 2.2076
Step 4999: train loss 2.1628, val loss 2.2014
Done training
After training:
---
oun n  s t t t t t t th t t th t t, th th? t t t th thothr t t tt the ttt tthe tsttttttttt
---
```

Let’s train and generate our first Language Model

```
%%time
model = LanguageModel()
train_and_generate(model)
```

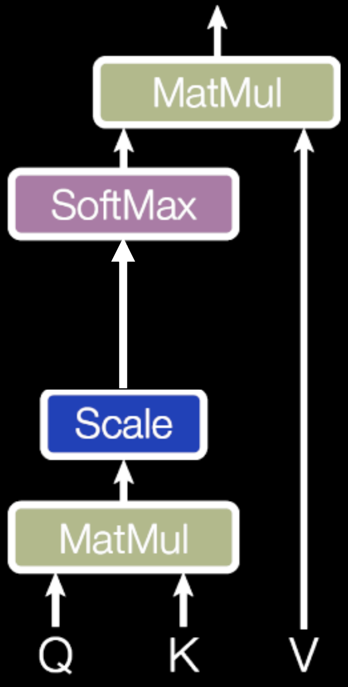
```
Before training:
---
SX;FS'CYWavScA!TeO$ehp-osN cU,SGza;AwI
V nR.G!EaneEXmE3LKzmz3!:UBtr!uatiKpJK!gqAmyWIarQ-b;Cj3nhmo:P!fGIuwrVhFNKy&q33DpPsrp3:v!-UzTTTRjpfo,rAkoJ-'.3.W;S;wOXTHss3x;jVA
M 1OD;E3wmh l&kdpiu;v!ZREF'ZeUGFUFXNWMZ ythgoWW$Jcx!rnaSuNGR:Alek.;u;Q&DIKAqWI-uX:pa,bh,M;eO3?OjNOPRW,d3,df
q;E;b
ES&!A,ZK'LpCPSs-C
zwqdrqiV&MIdDV3B-,;n.bFiHnAU3Mj,,!MNWlTJ'FSc  CMBmKaNQxnojFnv QF nwsO,Nly-C
;UsUJcRpZZH-'.JBHTGjGDJbkRaG'c
-R.qQ;$aFtbdSyKDK-hMXYAXyRgrCdccfdwAeJUQ!R:CvY  M.G?uLD,?ANzMSlD,u;!!aFVZW$t?RUVzhKPbhcUepOnB'Z
---

Training model: 0.007297M parameters
Step 0: train loss 4.1351, val loss 4.1377
Step 200: train loss 3.1912, val loss 3.2174
Step 400: train loss 2.9819, val loss 3.0040
```

```
Step 4800: train loss 2.1687, val loss 2.2076
Step 4999: train loss 2.1628, val loss 2.2014
Done training
After training:
---
oun n  s t t t t t t th t t th t t, th th? t t t th thothr t t tt the ttt tthe tstttttttttt
---
```

This is bad!
What went wrong?

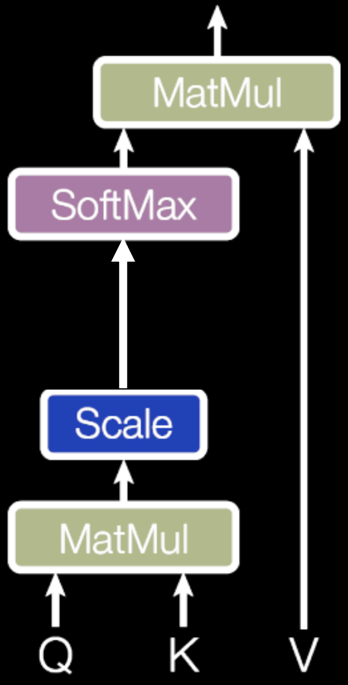
Attention Revisited

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$



Attention Revisited

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$



- We said that this part was some sort of “weight”
- In the case of one-hot encoding it was “selecting” one row from matrix V.
- In a generic case (e.g. when we use Embeddings), it is selecting several rows from V
  - ...that are weighted and added together
  - ...you can interpret this as: “Which rows from V, should I weight more?”
  - ...or equivalently: “Which rows from V, should I pay more attention to?”

Attention Revisited

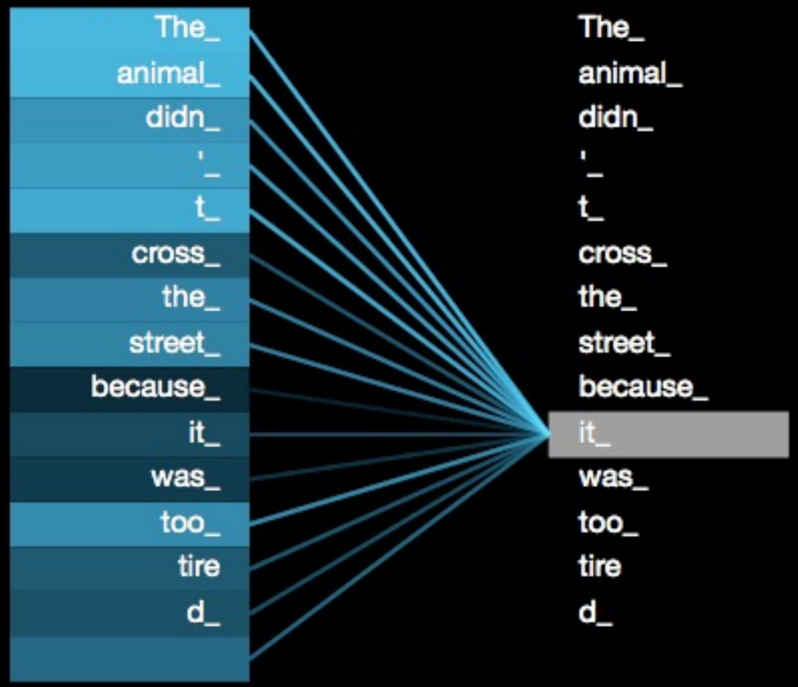
“The animal didn’t cross the street because it was too tired”

Attention Revisited

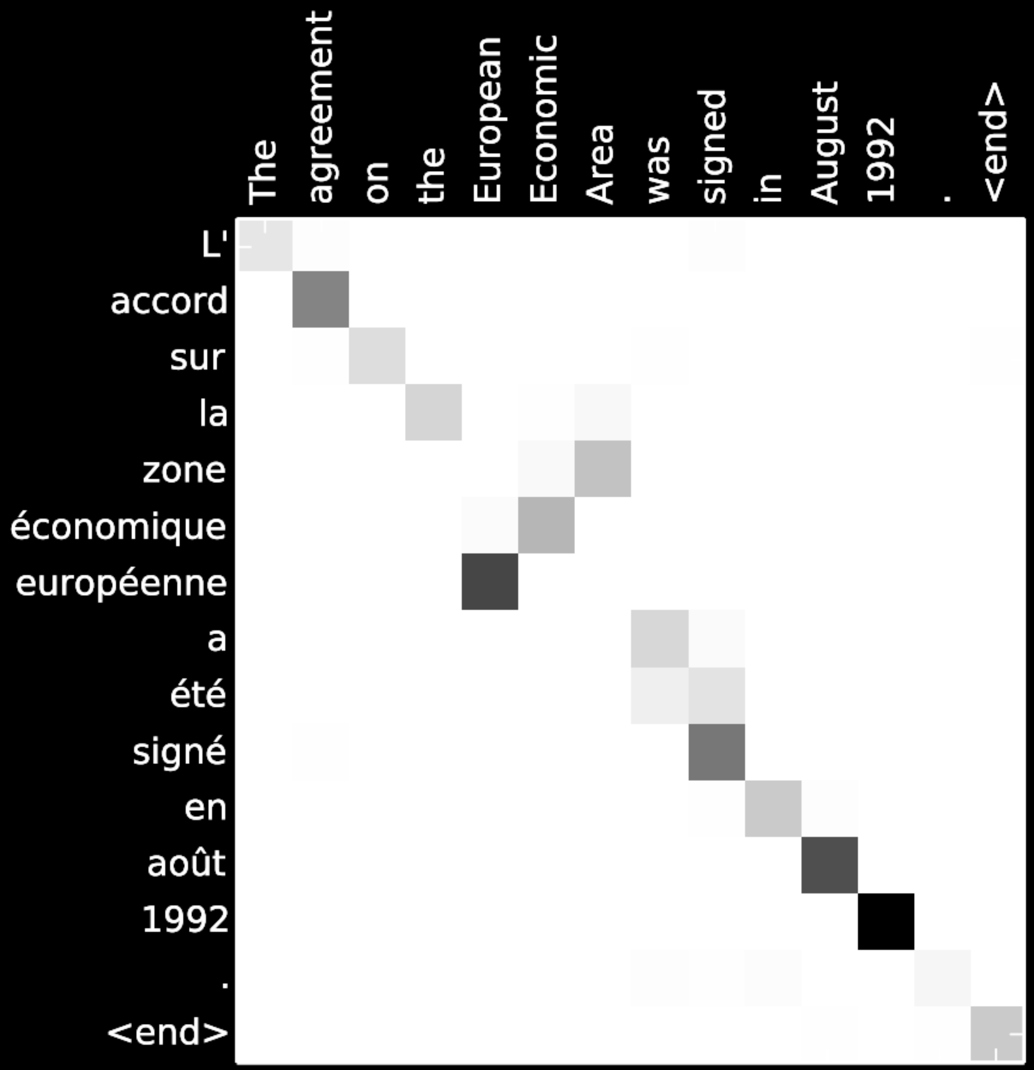
“The animal didn’t cross the street because it was too tired”

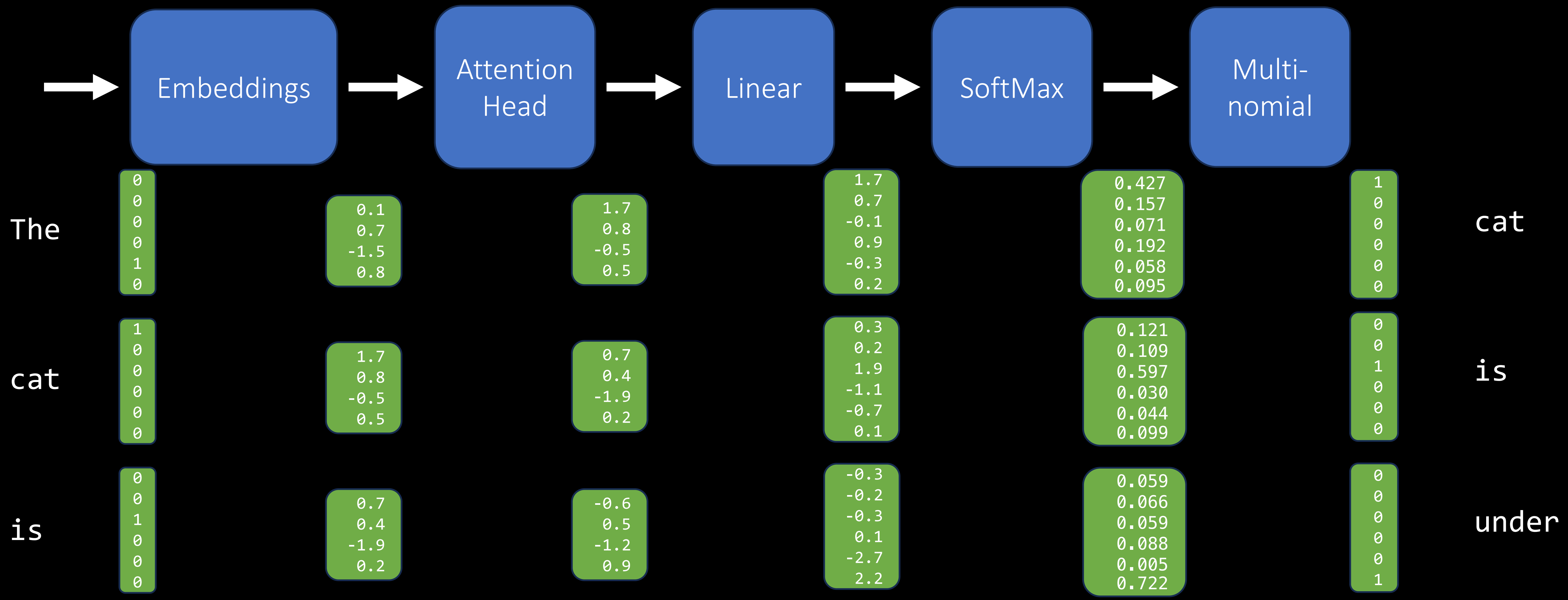
Attention Revisited

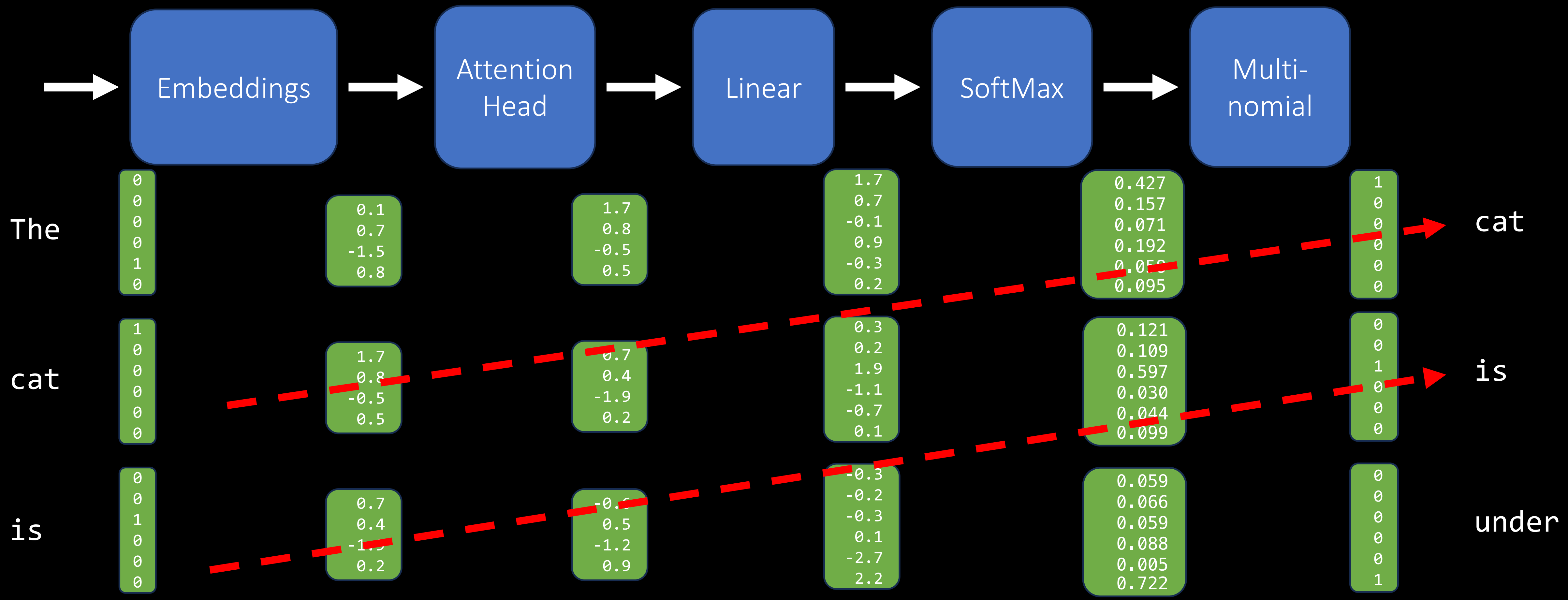
"The animal didn't cross the street because it was too tired"



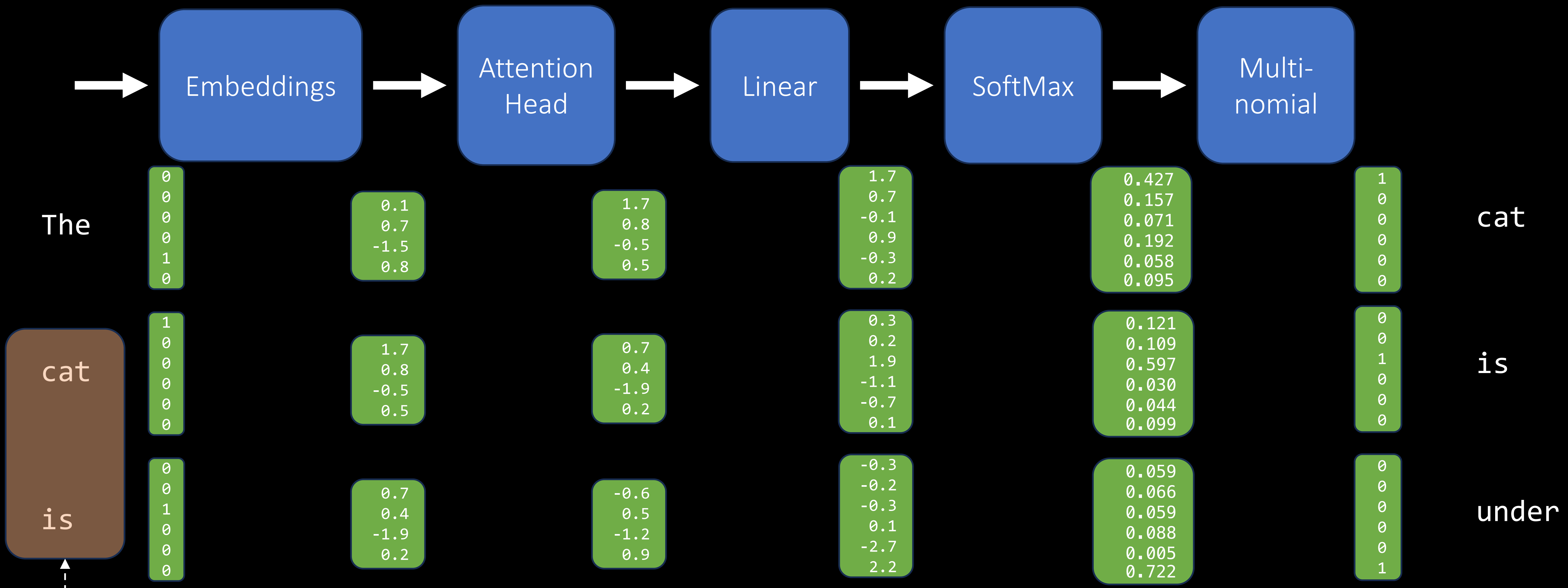
Attention Revisited




Embeddings
Attention Head
Linear
SoftMax
Multi-nomial
The
cat
is
0 0 0 0 1 0
1 0 0 0 0 0
0 0 1 0 0 0
0.1 0.7 -1.5 0.8
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
1.7 0.8 -0.5 0.5
0.7 0.4 -1.9 0.2
-0.6 0.5 -1.2 0.9
1.7 0.7 -0.1 0.9 -0.3 0.2
0.3 0.2 1.9 -1.1 -0.7 0.1
-0.3 -0.2 -0.3 0.1 -2.7 2.2
0.427 0.157 0.071 0.192 0.058 0.095
0.121 0.109 0.597 0.030 0.044 0.099
0.059 0.066 0.059 0.088 0.005 0.722
1 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 1
cat
is
under

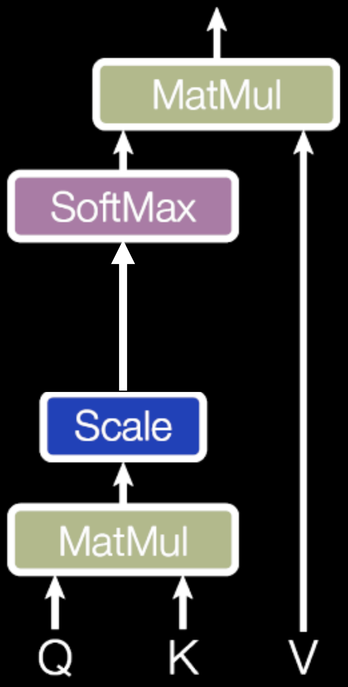


Problem: The network can learn to ”cheat”, using “future” words

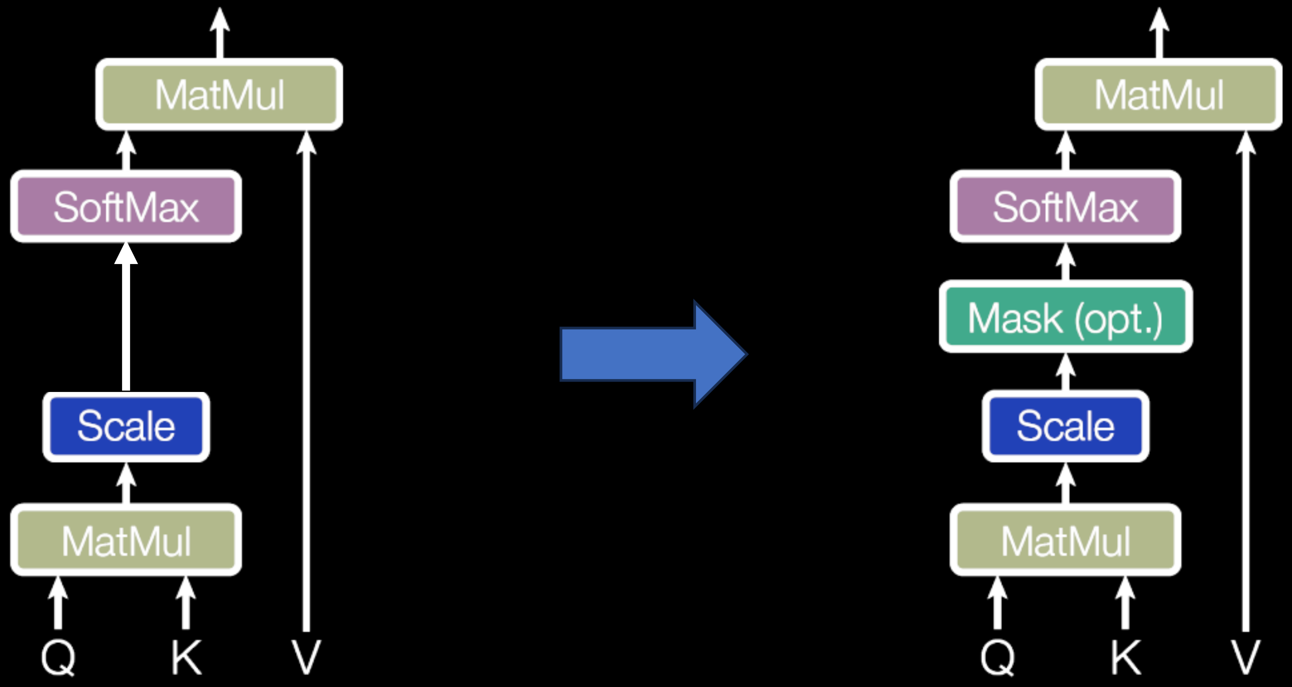


Solution: We need to "mask" future words, so that the network cannot "cheat"

Masked Attention



Masked Attention



How to “mask” using softmax()

$$softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-1.2271\\-7.4586\\0.7402\\2.1591\\-2.0037\end{bmatrix}\right)$$

How to “mask” using softmax()

$$softnax\left(\begin{bmatrix} -9.3456 \\ -0.3535 \\ 1.4897 \\ 9.2435 \\ -9.1582 \\ -1.2271 \\ -7.4586 \\ 0.7402 \\ 2.1591 \\ -2.0037 \end{bmatrix}\right)$$

How to “mask” using softmax()

$$softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-1.2271\\-7.4586\\0.7402\\2.1591\\-2.0037\end{bmatrix}+\begin{bmatrix}0\\0\\0\\0\\0\\-\infty\\-\infty\\-\infty\\-\infty\\-\infty\end{bmatrix}\right)$$

How to “mask” using softmax()

$$softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-1.2271\\-7.4586\\0.7402\\2.1591\\-2.0037\end{bmatrix}+\begin{bmatrix}0\\0\\0\\0\\0\\-\infty\\-\infty\\-\infty\\-\infty\\-\infty\end{bmatrix}\right)=softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-\infty\\-\infty\\-\infty\\-\infty\\-\infty\end{bmatrix}\right)$$

How to “mask” using softmax()

$$softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-1.2271\\-7.4586\\0.7402\\2.1591\\-2.0037\end{bmatrix}+\begin{bmatrix}0\\0\\0\\0\\0\\-\infty\\-\infty\\-\infty\\-\infty\\-\infty\end{bmatrix}\right)=softnax\left(\begin{bmatrix}-9.3456\\-0.3535\\1.4897\\9.2435\\-9.1582\\-\infty\\-\infty\\-\infty\\-\infty\\-\infty\end{bmatrix}\right)=\begin{bmatrix}8.4458e-09\\6.7900e-05\\4.2892e-04\\9.9950e-01\\1.0187e-08\\0\\0\\0\\0\\0\end{bmatrix}$$

Masked Attention

```python
class Head(nn.Module):
  """ Self attention head """
  def __init__(self):
    super().__init__()
    self.key = nn.Linear(n_embd, n_embd, bias=False)
    self.query = nn.Linear(n_embd, n_embd, bias=False)
    self.value = nn.Linear(n_embd, n_embd, bias=False)
    # Attention mask template, i.e. lower triangular matrix
    # Note: This is a buffer because it's not a learnable parameter
    self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))

  def forward(self, x):
    b, t, c = x.shape
    k = self.key(x)
    q = self.query(x)
    v = self.value(x)
    # Attention score
    w = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
    w = w.masked_fill(self.tril[:t, :t] == 0, float('-inf') )
    w = F.softmax(w, dim=-1)
    # Weighted values
    out = w @ v
    return out
```

Training the updated language model

```
Before training:
---
.A n'icHo
,ilqWFP$?QuJmSpK?-CKC'lVw?-?'VWpzdDF
MG?dfTB$v AJZQrbErW3IeD.,REeEPj:zcrsmXKng!cPNXpSfbGwWE.FByGgmfqeRkUO;iyCZASbQ;3!QoMWJ;H'NRRUW,tuAbekJ
WC:cvR$gu?Z.gWwtZp&UGbvkgZolxdphsQye$dT$rAi.pHW3lrfMEUqWTJwHzTijvWolMXUalTcq
yQI?Qvb3VPLev$N;txL
WkfjlTHJYUVhLsfex.ZN.$osmQGRRQNTsfCgI.fm&ZsWk:llZ Tqt vJQbVWZedt3wYA-f,-Qrl!gilqm&Rzr,h,Mq-G'
tOdqdpMni3WnNLMCq$aeTWs&cbv.aYNYk3Dx:b3&leUyUTPl!:ZDpmCcaJtnYEIpFhwuImG',x
wB$sAyQpQtZwud n,wApfTVi!&uyuhrHjCpYQEIxEwrGkcs:, WlijKVuLumjniHgUUq MuWE?KGs:UQaBMjP
---

Training model: 0.007297M parameters
Step 0: train loss 4.2211, val loss 4.2219
Step 200: train loss 3.3365, val loss 3.3494
Step 400: train loss 3.0721, val loss 3.0875
…
Step 4800: train loss 2.4781, val loss 2.5146
Step 4999: train loss 2.4752, val loss 2.5121
Done training

After training:
---
 onge theims se wend houtrourotorind?

Sais t whand, buotive rache se osacad te in achasheats hadayone whe!
Tillt ilyoor
MAronthe
:
Badin viesun oul, lelachisurt-s male ar ua chirpecon, howiowor:
Th shth rebe
O:esso ur owo als bf t n, ieieanincpe hmens he Hatheaso'ng hatous t
TOon,
Sacund R:
Oold, aveends byomy y bharbr, siclrkered piepein. d, speer is winoo, spnd,
Tit, rd ss my h ighit thaaruave; l ce e tele ou awhe bralathe!
Ca IMourime prrex, y scis, re ur le cearocaithy shopr;, cthe llo, be
---
```

Much better than what we had before (still not great)

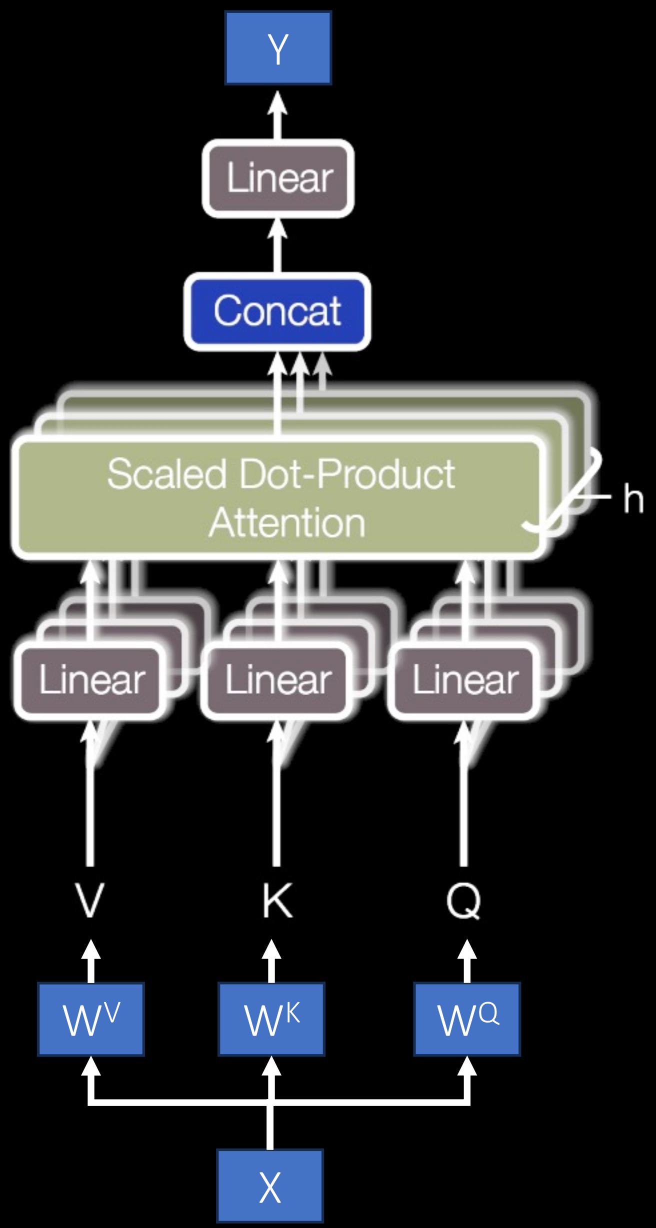
Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML"
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Multi-headed attention

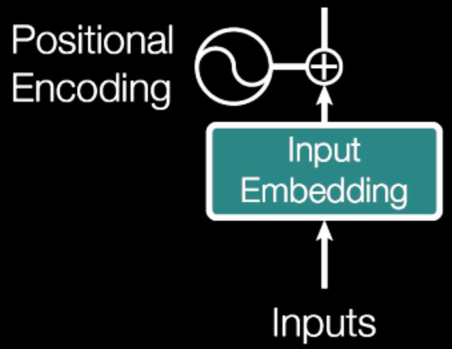
$$
\begin{aligned}
\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O \\
\text{where}~\mathrm{head_i} &= \mathrm{Attention}(QW^Q_i, KW^K_i, VW^V_i)
\end{aligned}
$$


Y
Linear
Concat
Scaled Dot-Product
Attention
h
Linear
Linear
Linear
V
K
Q
W^V
W^K
W^Q
X

```python
class MutiHeadedAttention(nn.Module):
  """ Multiple attention heads """

  def __init__(self, params):
    super().__init__()
    self.heads = nn.ModuleList([Head(params) for _ in range(params.num_heads)])
    self.proj = nn.Linear(params.num_heads * params.head_size, params.n_embd)

  def forward(self, x):
    out = torch.cat([h(x) for h in self.heads], dim=-1)
    return self.proj(out)
```

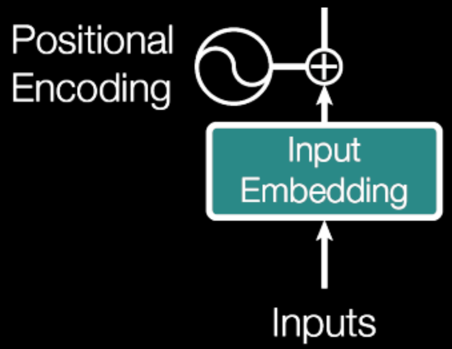
Positional Embeddings

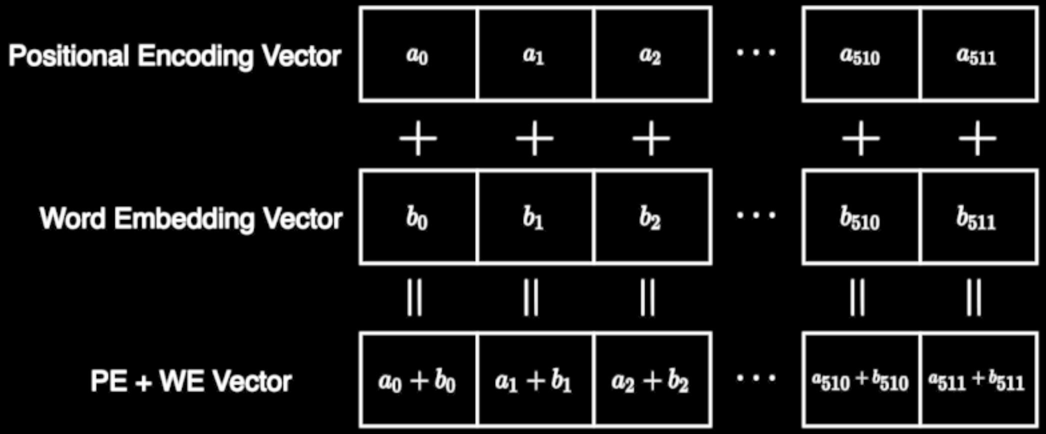
"Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence."



Positional Embeddings

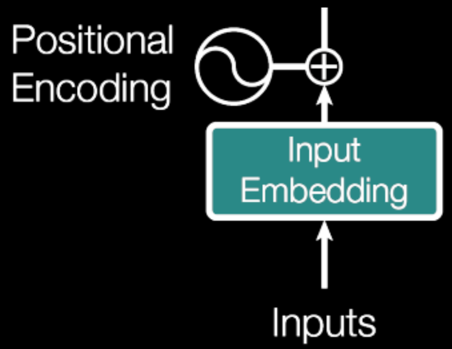
"Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence."

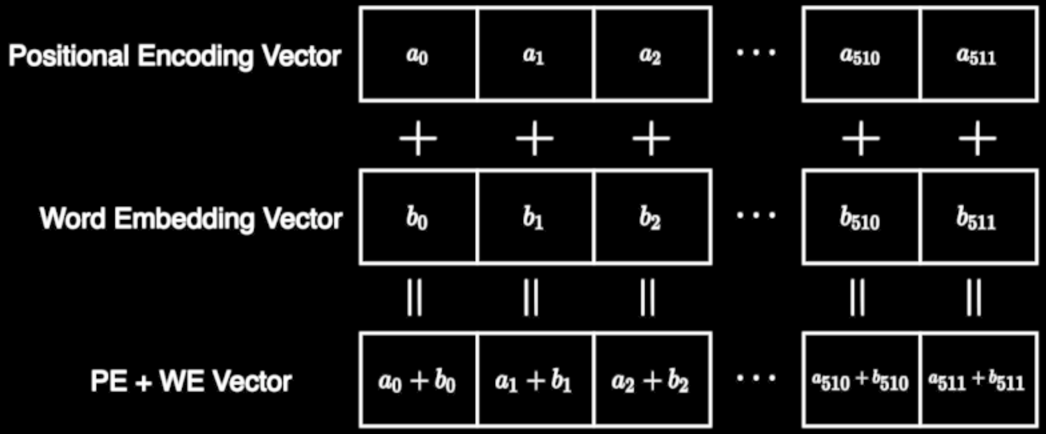




Positional Embeddings

"Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence."





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

Language Model 2

```python
class LanguageModel(nn.Module):
  """ Multi-headed attention model """
  def __init__(self):
    super().__init__()
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.possition_embedding = nn.Embedding(block_size, n_embd)
    self.sa_heads = MutiHeadedAttention()
    self.lm_head = nn.Linear(n_embd, vocab_size)

  def forward(self, idx, targets=None):
    b, t = idx.shape
    tok_emb = self.token_embedding_table(idx)
    pos_emb = self.possition_embedding(torch.arange(t, device=device))
    x = tok_emb + pos_emb
    x = self.sa_heads(x)
    logits = self.lm_head(x)
    if targets is None:
      loss = None
    else:
      b, t, c = logits.shape
      logits = logits.view(b*t, c)
      targets = targets.view(b*t)
      loss = F.cross_entropy(logits, targets)
    return logits, loss
```

Training the updated “Language Model 2”

```
Before training:
---
hV  3;Y;golf-YRMENdDwg,,TYwUXkU$veq&laZFMQV3nZUoHUs&iZD?d!Epl-,k!rhzzHdyudTh,VUQ
 DnRbYlbU?R-UiE,xkb&b'CamuU;;Yz:Yb;hzo:
i,xmNQ-fEEau WOUID!PSYF'vT!sz!KyjRXT!Z3MgRGox WD?k:xyR;UQXeo-,;mwJ.jc$Q?D
VPD$W'J.r.I:ko$zmHps xnBi;.-ltad!p'p$LT y
ksGiS.:E.TJw
efnb'l
?PN.B;:qMZr$!gb.UnqdGLf3LVnYaMBK,3ysstpCyI-OjemJ,E3DdIK.D,Y;G3Sejok'MQjGelmk3ic$NVayu'QTKgKKD&vUlb3.RtQmVIJY'pwPNTRKYJIbw$iPLuWYWON,P
pOtgeyMKhkCvc$'BESS?
MA:Usr?!kz'tJ:$g??.-jjDDz,RmsSmfE,d3:OrEFNGFJJyQHAccUkDNsKcwqtefSjWCIQAQx.K:LA.eHR.xsp,

---

Training model: 0.008865M parameters
Step 0: train loss 4.1807, val loss 4.1828
Step 200: train loss 3.2056, val loss 3.2313
. . .
Step 4800: train loss 2.3602, val loss 2.3910
Step 4999: train loss 2.3472, val loss 2.3834
Done training
After training:
---
h rocunces mdencunneid cokiz ant se suls fader ay.
Arl mapros E:
Bdy, vis as havie Ivond ato be ikef
Hacos y ave foum and, trofe, re kinTur; kot
W'lid?

TINGAur, ba! thor
TAUNEGLARORBUKCDETSNGAngot! ich:
Cnot,
Yy irde asingath merat mbe a, olory cas lt I ase' damy wicre psales hatapof oundin,
I';dent cour coung arey wigheemiestt fle a the he beave haror bou mamecpoy
I nad conten uvesttetimel ye goth myur we eid athey prre.

s me re, dang
n lorwoungth ho, rot mewl womar alnd gousetsh lounh sor la
---
```

Adding a non-linear layer

Now we add a non-linear layer (ReLU),

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Now we add a non-linear layer (ReLU),

```
class FeedForward(nn.Module):
  def __init__(self, n_embd):
    super().__init__()
    self.net = nn.Sequential(
        nn.Linear(n_embd, n_embd),
        nn.ReLU(),
    )

  def forward(self, x):
    return self.net(x)
```

Now we add a non-linear layer (ReLU),

Note: We use two weights matrices, one before, one after the non-linear operation.

```python
class FeedForward(nn.Module):
  def __init__(self, params):
    super().__init__()
    # The '4' is comming from the paper.
    # In equation 2 uses d_model=512, but d_ff=2048
    self.net = nn.Sequential(
        nn.Linear(params.n_embd, 4 * params.n_embd),
        nn.ReLU(),
        nn.Linear(4 * params.n_embd, params.n_embd),
    )

  def forward(self, x):
    return self.net(x)
```

Language Model 3:

```python
class LanguageModel(nn.Module):
  """ Multi-headed attention model """
  def __init__(self, num_heads=4):
    super().__init__()
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.possition_embedding = nn.Embedding(block_size, n_embd)
    self.sa_heads = MutiHeadedAttention()
    self.ffw = FeedForward(n_embd)
    self.lm_head = nn.Linear(n_embd, vocab_size)

  def forward(self, idx, targets=None):
    b, t = idx.shape
    tok_emb = self.token_embedding_table(idx)
    pos_emb = self.possition_embedding(torch.arange(t, device=device))
    x = tok_emb + pos_emb
    x = self.sa_heads(x)
    x = self.ffw(x)
    logits = self.lm_head(x)
    if targets is None:
      loss = None
    else:
      b, t, c = logits.shape
      logits = logits.view(b*t, c)
      targets = targets.view(b*t)
      loss = F.cross_entropy(logits, targets)
    return logits, loss
```

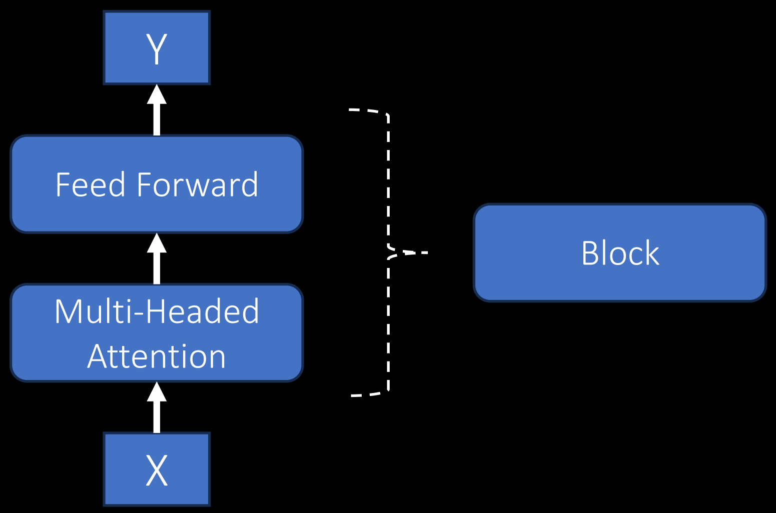
Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML"
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

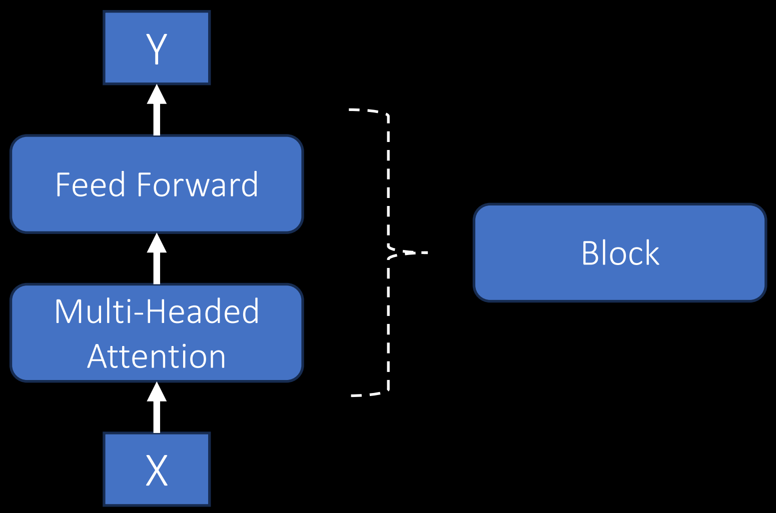
Towards “Transformer” architecture

What we have so far is a "Transformer block"



Note: This is simplified, and we are still missing a few parts.

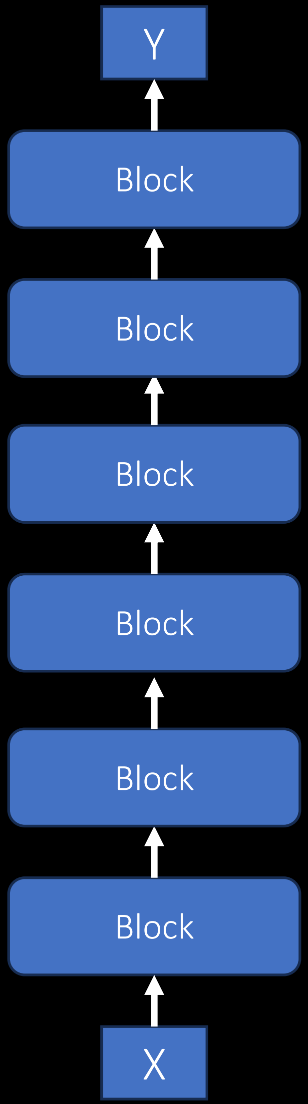
What we have so far is a “Transformer block”



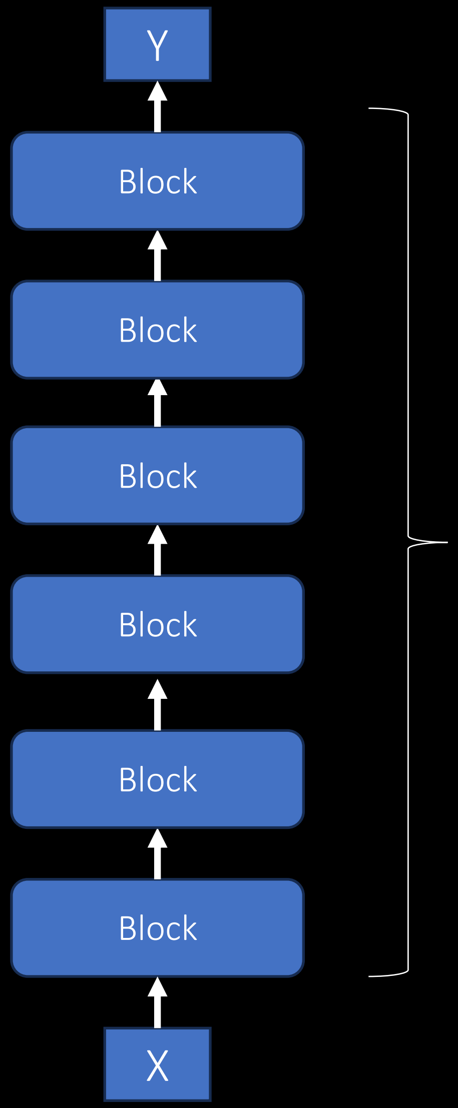

```
class Block(nn.Module):
  def __init__(self):
    super().__init__()
    head_size = n_embd // num_heads
    self.sa = MutiHeadedAttention()
    self.ffw = FeedForward()

  def forward(self, x):
    x = self.sa(x)
    x = self.ffw(x)
    return x
```

Note: This is simplified, and we are still missing a few parts.

If we want a more powerful network, we can simply stack these blocks:



If we want a more powerful network, we can simply stack these blocks:



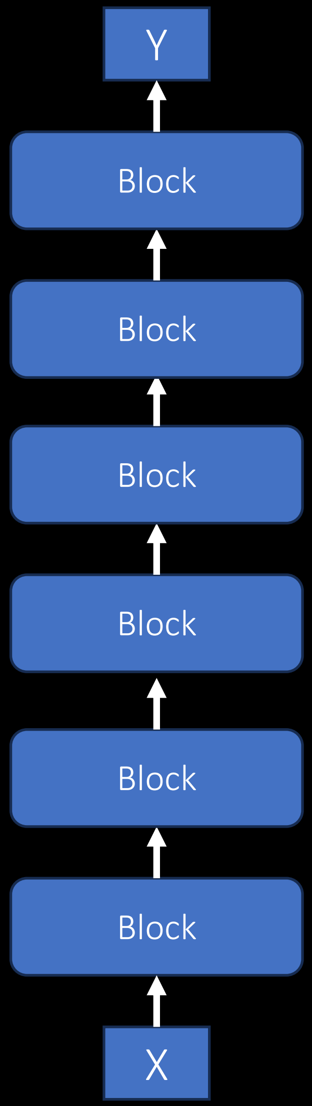
Deep networks are difficult to train, typical issues are

- Vanishing and exploding gradients
- Overfitting

To mitigate these issues, we use well known techniques:

- Residual connections
- Layer Normalization
- Dropout

Intuition for “Vanishing & Exploding Gradients”



The whole network can be written as a function ‘g’:

$$g(x) := f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1 x) \cdots))$$

We use a “loss function” to evaluate the network’s output g(x) respect to the desired output ‘y’:

$$C(y_i, g(x_i))$$

Using backpropagation, we calculate the gradient, which is used to update the network’s weights:

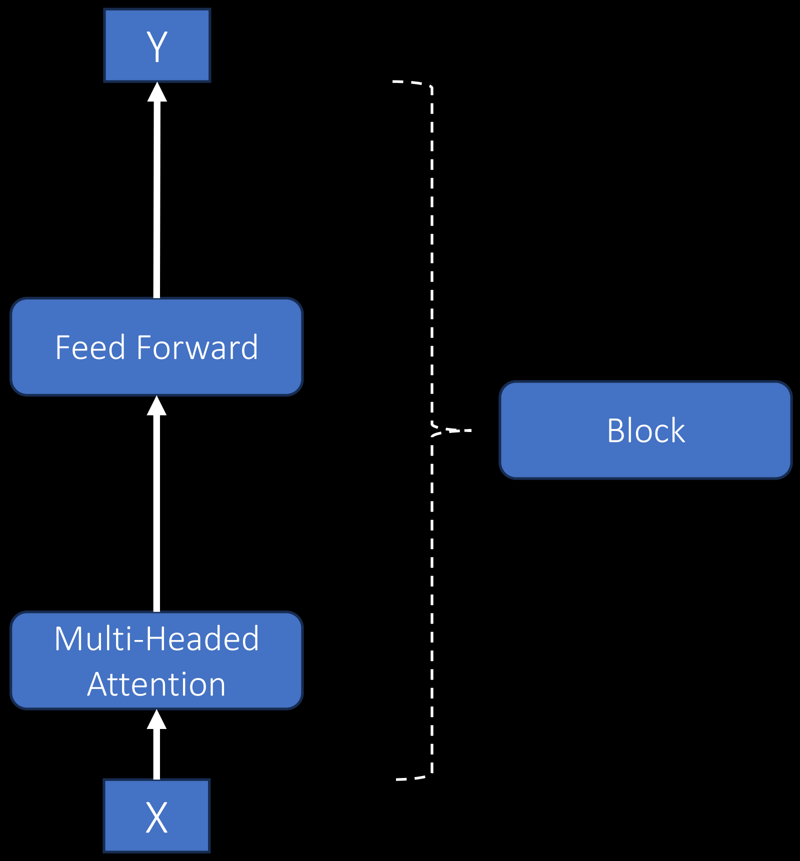
$$\nabla_x C = (W^1)^T \cdot (f^1)' \circ \ldots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C.$$

In a deep network, there are many factors multiplying in the gradient calculation.
The deeper the network, the more factors you have

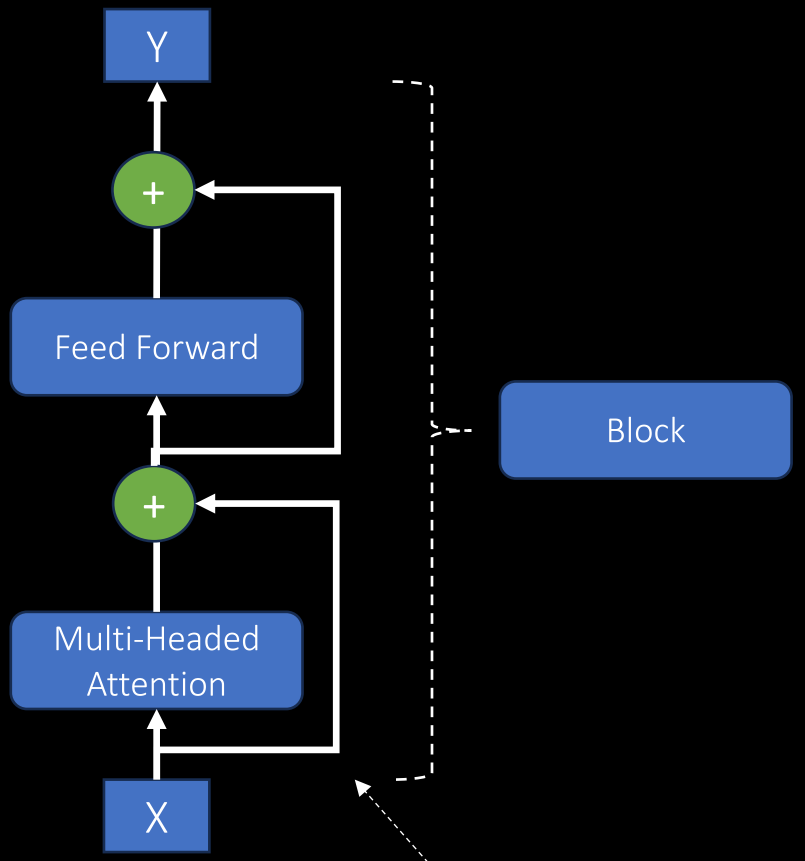
Intuition: Imagine that they are scalars instead of matrices.

- If many factors are less than 1 (e.g. 0.5) you’d get: 0.5 * 0.5 * 0.5 * ….. * 0.5 = very small number
- If many factors are more than 1 (e.g. 2) you’d get: 2 * 2 * 2 * …. * 2 = very large number

Residual connections

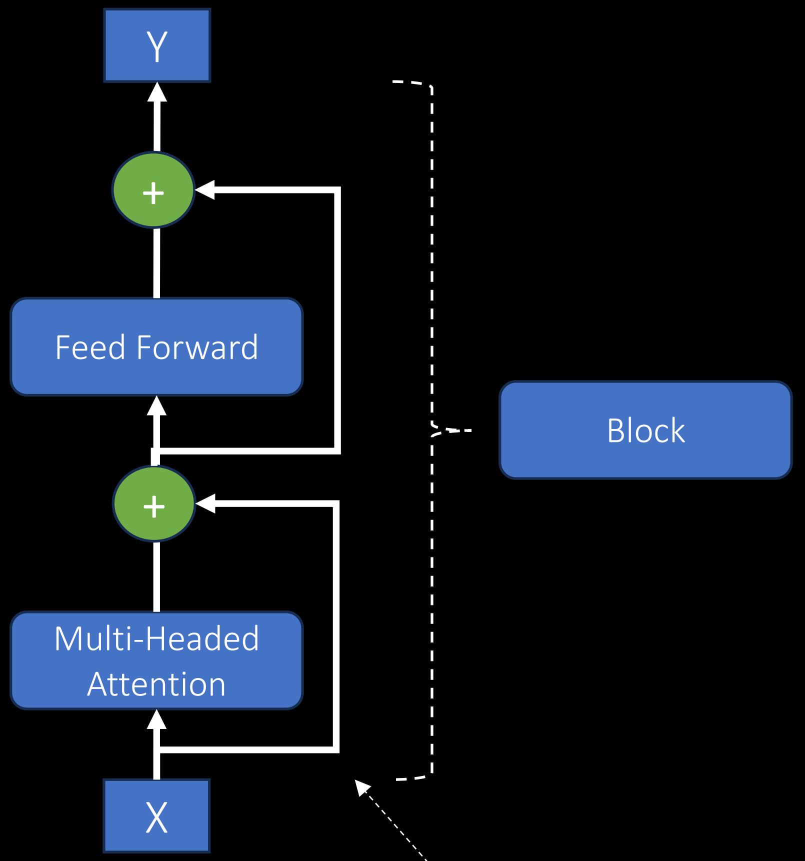


Residual connections



This is called a “residual connection”

Residual connections



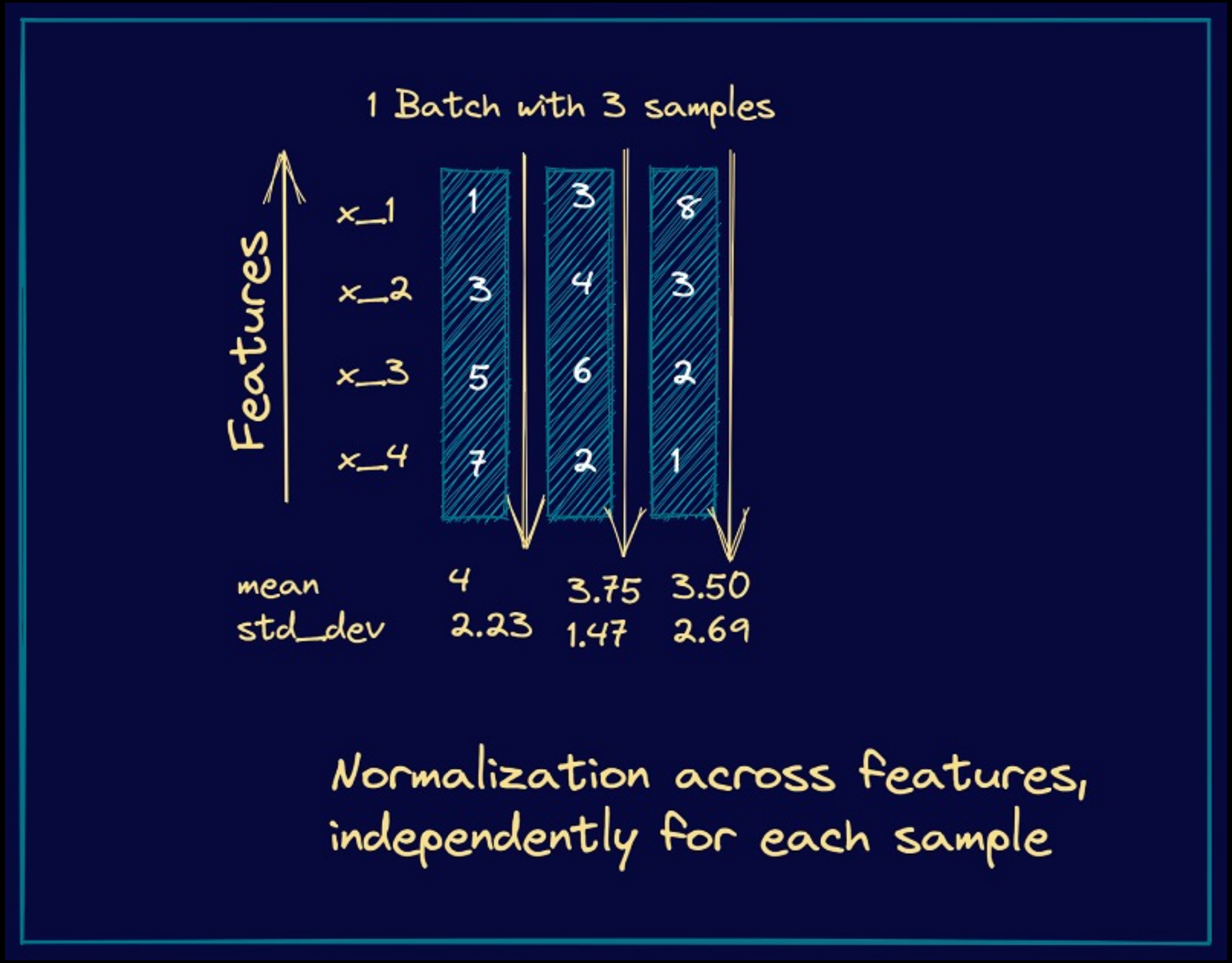

```python
class Block(nn.Module):
  def __init__(self):
    super().__init__()
    head_size = n_embd // num_heads
    self.sa = MutiHeadedAttention()
    self.ffw = FeedForward()

  def forward(self, x):
    x = x + self.sa(x)
    x = x + self.ffw(x)
    return x
```

This is called a “residual connection”

Language Model 4:

```python
class LanguageModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.possition_embedding = nn.Embedding(block_size, n_embd)
    self.blocks = nn.Sequential(
        Block(),
        Block(),
        Block(),
        Block(),
    )
    self.lm_head = nn.Linear(n_embd, vocab_size)

  def forward(self, idx, targets=None):
    b, t = idx.shape
    tok_emb = self.token_embedding_table(idx)
    pos_emb = self.possition_embedding(torch.arange(t, device=device))
    x = tok_emb + pos_emb
    x = self.blocks(x)
    logits = self.lm_head(x)
    if targets is None:
      loss = None
    else:
      b, t, c = logits.shape
      logits = logits.view(b*t, c)
      targets = targets.view(b*t)
      loss = F.cross_entropy(logits, targets)
    return logits, loss
```

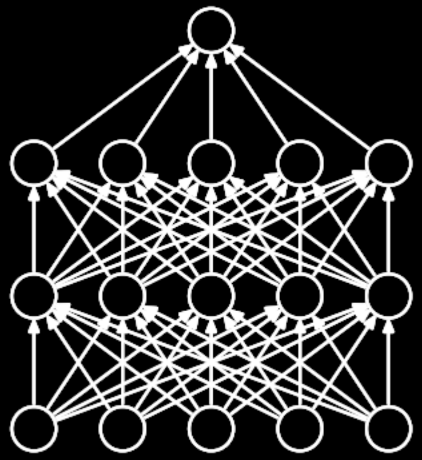
Layer Normalization



Updated Transformer Block

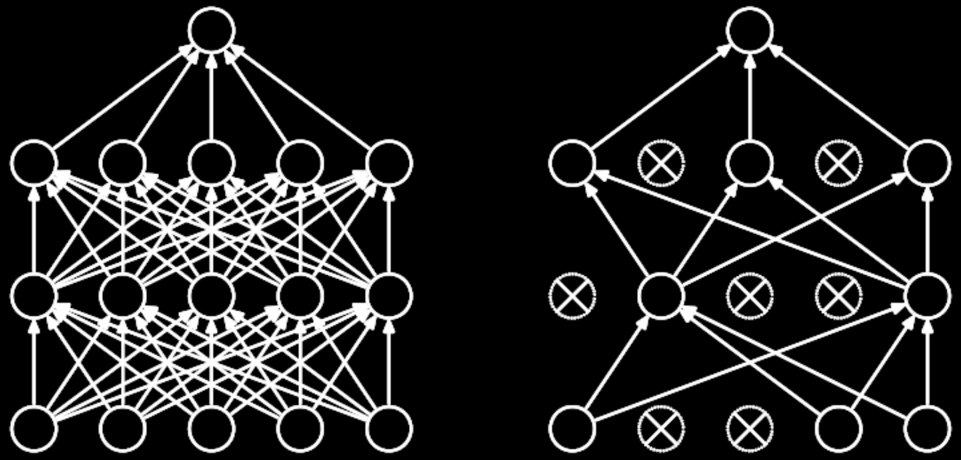
```python
class Block(nn.Module):
  def __init__(self):
    super().__init__()
    self.sa = MutiHeadedAttention()
    self.ln1 = nn.LayerNorm(n_embd)
    self.ffwd = FeedForward(n_embd)
    self.ln2 = nn.LayerNorm(n_embd)

  def forward(self, x):
    # Note: As of 2023 it is more common to apply LayerNorm
    # before Self-Attnetion, as opposed to applying it after
    # feed-forward (as it was shown in the original paper)
    x = x + self.sa(self.ln1(x))
    x = x + self.ffwd(self.ln2(x))
    return x
```

Dropout



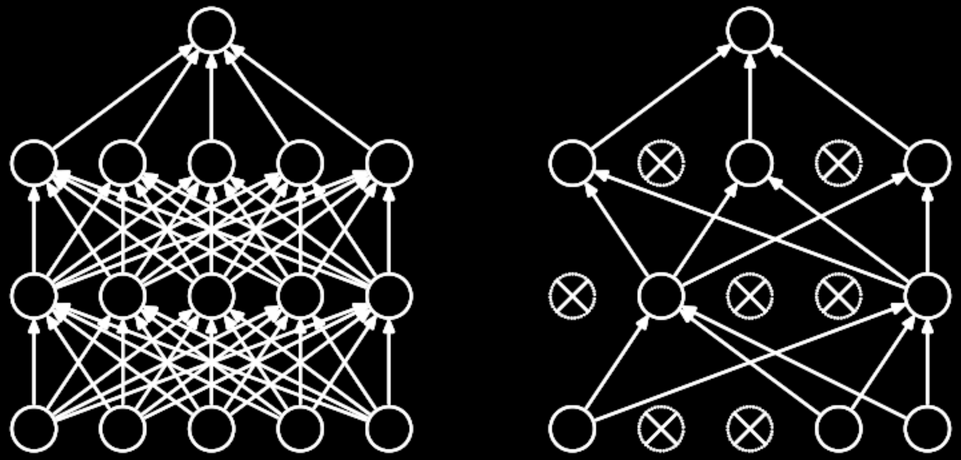
(a) Standard Neural Net

Dropout



(a) Standard Neural Net (b) After applying dropout.

Dropout



(a) Standard Neural Net (b) After applying dropout.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

Keywords: neural networks, regularization, model combination, deep learning

Updated Attention Head (with dropout):

```python
class Head(nn.Module):
  def __init__(self):
    super().__init__()
    self.key = nn.Linear(n_embd, head_size, bias=False)
    self.query = nn.Linear(n_embd, head_size, bias=False)
    self.value = nn.Linear(n_embd, head_size, bias=False)
    # Attention mask template, i.e. lower triangular matrix
    # Note: This is a buffer because it's not a learnable parameter
    self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    b, t, c = x.shape
    k = self.key(x)
    q = self.query(x)
    v = self.value(x)
    # Attention score
    w = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
    w = w.masked_fill(self.tril[:t, :t] == 0, float('-inf') )
    w = F.softmax(w, dim=-1)
    w = self.dropout(w)
    # Add weighted values
    v = self.value(x)
    out = w @ v
    return out
```

Updated Multi-Headed Attention (with dropout):

```
class MutiHeadedAttention(nn.Module):
  def __init__(self):
    super().__init__()
    self.heads = nn.ModuleList([Head() for _ in range(num_heads)])
    assert (num_heads * head_size) == n_embd
    self.proj = nn.Linear(num_heads * head_size, n_embd)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    out = torch.cat([h(x) for h in self.heads], dim=-1)
    out = self.dropout(self.proj(out))
    return out
```

Updated Feed-Forward layer (with dropout):

```python
class FeedForward(nn.Module):
  def __init__(self, n_embd):
    super().__init__()
    self.net = nn.Sequential(
        nn.Linear(n_embd, ff_scale_factor * n_embd),
        nn.ReLU(),
        nn.Linear(ff_scale_factor * n_embd, n_embd),
        nn.Dropout(dropout),
    )

  def forward(self, x):
    return self.net(x)
```

Language Model 5

```
class LanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.possition_embedding = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block() for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        b, t = idx.shape
        tok_emb = self.token_embedding_table(idx)
        pos_emb = self.possition_embedding(torch.arange(t, device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)
        if targets is None:
            loss = None
        else:
            b, t, c = logits.shape
            logits = logits.view(b*t, c)
            targets = targets.view(b*t)
            loss = F.cross_entropy(logits, targets)
        return logits, loss
```

Training the updated "Language Model 5"

```
Before training:
---
e&ByAmALoh'RHYnD.ytH:N3oWCWCSrgEJo$m
.PrHD?EV,MSXJ,ryGdn3oXjiLFFFlcYrouZ3vWXfZWKfhSJUXDMPYSRg,czR$pRuxcpZRvUEQQoHJLZjxiYI;UfyR3ZnYqFlS3SrC;f$mzA
YKVYfUbLMvilm!q?uqJAfa.S$ptQoviFFRS?lYga,JDELDG;e.siXvAXdXrgh3AYmrn$,XXgc& GXdN!l.C,Z-
3wSb.psABcioNHSJSkhSfXCmE!iXyrC;CXaxMUYiLmazDGnoPBOGYoFG;lNW$,h?FhqmJtKhpRMBPDsEMfY
,;W;rnAAr;eSWfsvcbiF;g$DFi,Ur,,Liyf3lr;XrGmf,MgyTKJovmKRBIIX,,UShQ OOLQ?ogyQlmEhrfRfqlFEF,,AWhZXMnSN3 yvRtx FS-yZWlEi-c
KeXYZGwF,My!kcQA!,BqUZyORE
qJfiFS.nfTR&GvE
o$Eiu,FS,Gro,.,X;jF3mZ
---

Training model: 0.030017M parameters
Step 0: train loss 4.3783, val loss 4.3747
. . .
Step 4800: train loss 2.0594, val loss 2.1101
Step 4999: train loss 2.0497, val loss 2.1016
Done training
After training:
---
h cincay farims her
Luine dob:
Coouce and kewar.

shapigh.
RIs youf wither, swall hie chouden.

LINORW:
Mit you, as tiord.

Lordider, your as his you thearen:
My oar dee monge. Riciuas
Horsing, whalp thimh wiff so, aave alrem, next, in you,
Agire, my droturs his here to danger, that in the shat love nat, prifer ase the moedy to thine the kioCl
I my you deed wall our as daut in tland wenle the ir asen houghts. Wair whomy ghiscaur! it.
```

Scaling Language Model 5

```
# Architecture parameters
max_vocab_size = 256
block_size = 256
n_embd = 384
num_heads = 6
n_layer = 6
ff_scale_factor = 4
dropout = 0.2
```

Training the “Scaled Language Model 5”

```
…
Step 4600: train loss 1.0614, val loss 1.5893
Step 4800: train loss 1.0493, val loss 1.6038
Step 4999: train loss 1.0298, val loss 1.6258
Done training

After training:
---
 welcome to us.

LEONTES:
I cannot tell thee; what's not so,
To make an envy, I merry to him.

DUCHESS OF YORK:
They have ever spraid upon this thrusty palace.
I'll appear no thingstony that I did see how shed to see me.

HENRY BOLINGBROKE:
What is the world I between the scial bold?

KING RICHARD III:
Even to hear thee do fight me and to fear;
Slugs it in them, be my knot faith of Gloucester.

Hiest Mercutio enough! Break from hence;
Stand to be adopted to cut a thing careful do.

BISHOP OF ELY
---

CPU times: user 47min 3s, sys: 4min 53s, total: 51min 56s
Wall time: 52min 5s
```

Training the "Scaled Language Model 5"

```
…
Step 4600: train loss 1.0614, val loss 1.5893
Step 4800: train loss 1.0493, val loss 1.6038
Step 4999: train loss 1.0298, val loss 1.6258
Done training

After training:
---
 welcome to us.

LEONTES:
I cannot tell thee; what's not so,
To make an envy, I merry to him.

DUCHESS OF YORK:
They have ever spraid upon this thrusty palace.
I'll appear no thingstony that I did see how shed to see me.

HENRY BOLINGBROKE:
What is the world I between the scial bold?

KING RICHARD III:
Even to hear thee do fight me and to fear;
Slugs it in them, be my knot faith of Gloucester.

Hiest Mercutio enough! Break from hence;
Stand to be adopted to cut a thing careful do.

BISHOP OF ELY
---

CPU times: user 47min 3s, sys: 4min 53s, total: 51min 56s
Wall time: 52min 5s
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us...
```

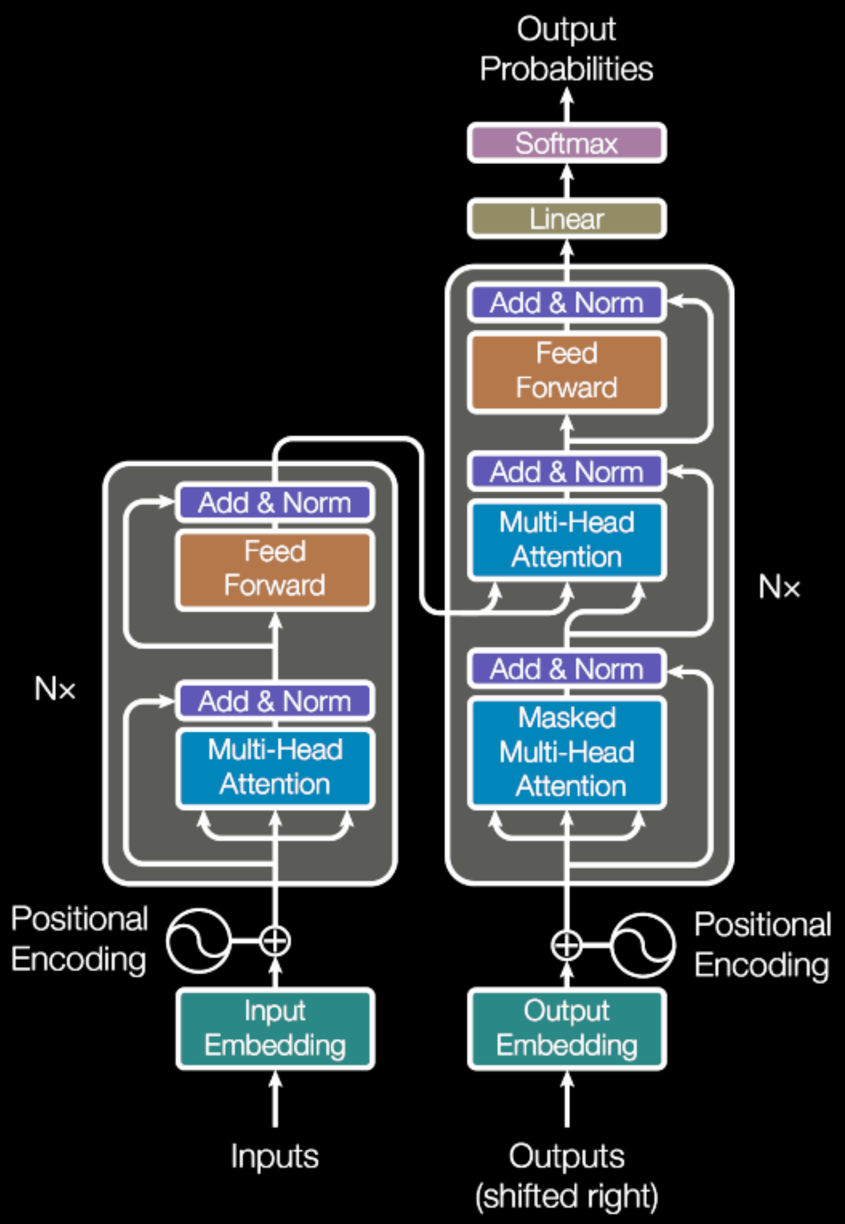


Figure 1: The Transformer - model architecture.

End of Part 2