In [ ]:
```
!pip install flax optax
```

```
Collecting flax
  Downloading flax-0.4.1-py3-none-any.whl (184 kB)
     |████████████████████████████████| 184 kB 4.6 MB/s
```

In [ ]:
```
import jax
import jax.numpy as jnp

import optax
import flax
from flax import linen as nn
from functools import partial
```

The goal of this exercise is to learn a basic language model (https://en.wikipedia.org/wiki/Language_model) using an recurrent neural network.

As as starting point, you should implement a simple RNN of the form (for reference, see the Deep Learning textbook chapter 10 page 370):

$$\begin{aligned} a^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh(a^{(t)}) \\ o^{(t)} &= Vh^{(t)} + c \ , \end{aligned}$$

where $h^{(t)}$ is the updated state at time t , $x^{(t)}$ is the input and $o^{(t)}$ is the output. Given an initial input $x^{(0)}$ and hidden state $h^{(0)}$, an RNN computes the output sequence $o^{(0)}, \dots, o^{(T)}$ by applying f recursively:

$$(h^{(t+1)}, o^{(t+1)}) = f(h^{(t)}, x^{(t)}; \theta) \ .$$

In [ ]:
```
class ElmanCell(nn.Module):
  @nn.compact
  def __call__(self, state, x):
    # IMPLEMENT the basic RNN cell described above (outputting h^{t} only)
    x = jnp.concatenate([state[0], x])
    a = jnp.tanh(nn.Dense(state[0].shape[0])(x))
    return jnp.tanh(a)
```

Mutiple such cells can be chained up together to attain more expressivity, for example, we can link two cells as follows:

$$\begin{aligned} h_1^{(t+1)} &= \tanh(W_1 h_1^{(t)} + U_1 x^{(t)} + b_1) \\ h_2^{(t+1)} &= \tanh(W_1 h_1^{(t)} + U_1 h_1^{(t)} + b_1) \\ o^{(t+1)} &= Vh_2^{(t+1)} + c \ , \end{aligned}$$

and the resulting network is of the form:

$$(h_1^{(t+1)}, h_2^{(t+1)}, o^{(t+1)}) = f(h_1^{(t)}, h_2^{(t)}, x^{(t)}; \theta) \ .$$

In [ ]:
```
class RecurrentNetwork(nn.Module):
  state_size: int
  num_classes: int

  @nn.compact
  def __call__(self, state, i):
    x = jnp.squeeze(jax.nn.one_hot(i, self.num_classes))
    # IMPLEMENT
    # Build a simple RNN with a single cell
    state = ElmanCell()(state, x)
    predictions = nn.Dense(features=state.shape[0])(state)
    return (state,), predictions

  def init_state(self):
    return (jnp.zeros(self.state_size),)
```

We learn our language model by taking a written document and learn to predict the next character (a character-level language model) using our RNN. This is prediction task is akin to a classification and we therefore use the cross-entropy loss:

$$l(x, y, \theta) \triangleq -\log p_\theta(y|x) \ .$$

We compute those probabilities using our RNN. The output $o^{(t)}$ represent the the so-called **logits** which can be transformed into probabilities using the softmax function. That is $\mathrm{softmax}(o^{(t)})$ gives us the desired probabilities.

In [ ]:
```
def make_rnn_loss(model):
  def cross_entropy(logits, target):
    """ Negative cross-entropy
    Args:
      logits: jnp.ndarray with num_classes dimensions (unbatched, batching is
 done outside through vmap)
      target (int): target class that should have been predicted
    """
    # IMPLEMENT the cross-entropy loss
    # Hint: use jax.nn.log_softmax to avoid computing the log and
    # the softmax separately. This function is numerically more stable that th
e
    # naive approach.
    probs = jax.nn.log_softmax(logits)
    return -jnp.sum(target * probs)

  def rnn_loss(params, inputs, targets, init_state):
    final_state, logits = jax.lax.scan(partial(model.apply, params), init_stat
e, inputs)
    loss = jnp.mean(jax.vmap(cross_entropy, in_axes=(0, 0))(logits, targets.as
type(jnp.int32)))
    return loss, final_state
  return jax.jit(rnn_loss)
```

Note that in the above function, we unroll the `jax.lax.scan` over a given input sequence and compute the loss along. When it comes to generating new content, we have to execute our RNN differently. That is: we provide a starting hidden state and character $x^{(0)}$, compute the distribution over next character using the softmax transformation of the logits computed as output of the RNN, sample one of those next character, and repeat the process in this manner until we reach a desired length. In other words, we generate a string *auto-regressively*.

A variant on the above procedure is to let the RNN start from more than just a given single character and instead pass a longer *prompt*. The same idea holds except that we have to `jax.lax.scan` over as many characters as we have in our prompt.

The process described above is *stochastic* in nature: the next character is sampled according to the predicted class distribution. When using the softmax transformaiton, can vary the degree of stochasticity using a temperature parameter τ. All we have to do is to multiply the logits by the inverse temperature: $\mathrm{softmax}((1/\tau)o^{(t)})$. The smaller the temperature, the more deterministic the model becomes.

In [ ]:
```
def sample(key, model, params, id_lookup, chr_lookup, prompt='', max_length=10
0, temperature=1.):
  encoded_prompt = jnp.asarray(list(map(lambda c: id_lookup[c], prompt)))
  state, _ = jax.lax.scan(partial(model.apply, params), model.init_state(), en
coded_prompt[:-1])

  num_classes = len(id_lookup)
  def autoregressive_step(carry, subkey):
    state, last_char = carry
    state, logits = model.apply(params, state, last_char)

    # IMPLEMENT
    probs = jax.nn.log_softmax((1/temperature)*logits)
    choice = jnp.random.choice(probs) # sample the next character at the given
temperature
    prediction = jnp.where(probs==choice)
    # prediction = jnp.argmax(probs)

    return (state, prediction), prediction
  keys = jax.random.split(key, max_length)
  _, sequence = jax.lax.scan(autoregressive_step, (state, id_lookup[prompt[-1
]]), xs=keys)
  decoded_sequence = list(map(lambda i: chr_lookup[int(i)], sequence))

  return prompt + ''.join(decoded_sequence)
```

The following code doesn't have to be modified. Its purpose is to turn the text into one-hot vectors (features) and to chunk up the text (which can be very large) into smaller and more manageable subsequences.

In [ ]:
```
def chunk(x, seq_size):
  if seq_size > x.shape[0]:
    return jnp.atleast_2d(x[:-1]), jnp.atleast_2d(x[1:])
  num_partitions = int(jnp.ceil(x.shape[0]/seq_size))
  inputs = jnp.array_split(x, num_partitions)
  targets = jnp.array_split(jnp.append(x[1:], jnp.nan), num_partitions)
  return inputs[:x.shape[0] % num_partitions], targets[:x.shape[0] % num_parti
tions]
```

In [ ]:
```
def sample_subsequence(key, data, size):
    ridx = jax.random.randint(key, (1,), minval=0, maxval=data.shape[0]-size)[
0]
    return data[ridx:ridx+size]
```

In [ ]:
```
def preprocess(data):
  unique_chars = set(data)
  id_lookup = dict(zip(unique_chars, range(len(unique_chars))))
  chr_lookup = dict(zip(range(len(unique_chars)), unique_chars))
  encoded_data = jnp.asarray(list(map(lambda c: id_lookup[c], data)))
  return encoded_data, id_lookup, chr_lookup, len(unique_chars)
```

Due to the large size of the training sequence (the entire text document), we have to split it into manageable subsequences. More precisely, at every training *epoch* we sample a contiguous subsequence from the entire document and compute the negative log likelihood loss by unrolling the RNN over the given characters. However, given the challenge (more on this in question) of learning over long horizon, we truncate the unroll over fewer characters and warm start the initial state between each such truncated unroll.

In [ ]:
```
def train(key, data, state_size, learning_rate, n_epochs, batch_size, max_subs
equence_length, sample_length, test_prompt=None, temperature=1.):
  encoded_data, id_lookup, chr_lookup, num_classes = preprocess(data)
  model = RecurrentNetwork(state_size, num_classes)
  params = model.init(key, model.init_state(), 0)

  optimizer = optax.adam(learning_rate=learning_rate)
  opt_state = optimizer.init(params)

  rnn_loss_grad = jax.value_and_grad(make_rnn_loss(model), has_aux=True)
  opt_state = optimizer.init(params)
  for i in range(n_epochs):
      key, subkey = jax.random.split(key)
      subsequence = sample_subsequence(key, encoded_data, max_subsequence_leng
th).astype(jnp.int32)

      state = model.init_state()
      batch_losses = []
      for inputs, targets in zip(*chunk(subsequence, batch_size)):
        (loss, state), gradient = rnn_loss_grad(params, inputs, targets, state
)
        updates, opt_state = optimizer.update(gradient, opt_state)
        params = optax.apply_updates(params, updates)
        batch_losses.append(loss)
      if not (i % 10):
        if test_prompt is None:
          test_prompt = data[:4]
        generated_string = sample(key, model, params, id_lookup, chr_lookup, t
est_prompt, max_length=sample_length, temperature=temperature)
        print(f"Epoch {i} Average loss: {jnp.mean(jnp.asarray(batch_losses)):.
5f} random sample: {generated_string}")
```

Testing

We learn our language model over a children book called "The Life and Adventures of Santa Claus" by L. Frank Baum.

In [ ]:
```
!wget https://gutenberg.org/cache/epub/520/pg520.txt
```

```
--2022-03-12 02:51:32--  https://gutenberg.org/cache/epub/520/pg520.txt
Resolving gutenberg.org (gutenberg.org)... 152.19.134.47, 2610:28:3090:3000:
0:bad:cafe:47
Connecting to gutenberg.org (gutenberg.org)|152.19.134.47|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 188277 (184K) [text/plain]
Saving to: ‘pg520.txt’

pg520.txt           100%[===================>] 183.86K   678KB/s    in 0.3s

2022-03-12 02:51:33 (678 KB/s) - ‘pg520.txt’ saved [188277/188277]
```

```
In [ ]: with open('pg520.txt', 'r') as file:
          data = file.read()
```

When training your model, you should be able to observe that the samples become more coherent over time while the log likelihood loss goes down.

In [ ]:
```
key = jax.random.PRNGKey(0)
train(key, data, state_size=256, learning_rate=1e-3, batch_size=64, n_epochs=2
000, max_subsequence_length=5000, sample_length=50, test_prompt='Santa ', temp
erature=1e-1)
```

```
Epoch 0 Average loss: 3.43944 random sample: Santa
Epoch 10 Average loss: 2.82208 random sample: Santa  ane the the the the the
the the the the the the t
Epoch 20 Average loss: 2.36441 random sample: Santa the he the sand he the sa
nd he the sand he the san
Epoch 30 Average loss: 2.41000 random sample: Santa the the the the the the t
he the the the the the th
Epoch 40 Average loss: 2.30507 random sample: Santa the tored the sered the s
ored the seres wore the t
Epoch 50 Average loss: 2.28557 random sample: Santa dout the Fore the the ser
en the seer and he wher a
Epoch 60 Average loss: 2.15486 random sample: Santa the sored the he the the
deat the he the could the
Epoch 70 Average loss: 2.10079 random sample: Santa Claus the shinged the sou
s the seas the she sead t
Epoch 80 Average loss: 2.41078 random sample: Santa Claus or the cored torker
of the cored torker of t
Epoch 90 Average loss: 2.18038 random sample: Santa Claus he he he he he he h
e he he he he he he he he
Epoch 100 Average loss: 2.88861 random sample: Santa Claus of coment Gutent o
n and the Project Gutent o
Epoch 110 Average loss: 2.12971 random sample: Santa Claus the laden the sto
the child ghe the ghe the
Epoch 120 Average loss: 2.11627 random sample: Santa Claus the calle the call
e the calle the calle the
Epoch 130 Average loss: 2.17712 random sample: Santa Claus the comed the bere
nt he prepling the bere th
Epoch 140 Average loss: 2.11828 random sample: Santa Claus for sto the caller
ed the caller him the sing
Epoch 150 Average loss: 2.00210 random sample: Santa Claus for the colle the
colly and ald and sere the
Epoch 160 Average loss: 2.00832 random sample: Santa Claus ware the wored the
wood the wored to the war
Epoch 170 Average loss: 2.01791 random sample: Santa Claus wathe has ween the
King the King the King th
Epoch 180 Average loss: 2.00310 random sample: Santa Claus work the Food the
Food the Food the Food the
Epoch 190 Average loss: 1.96233 random sample: Santa Claus the came the came
the rears and the pare the
Epoch 200 Average loss: 1.92799 random sample: Santa Claus found the wall the
wall the wall the wook in
Epoch 210 Average loss: 1.94283 random sample: Santa Claus for he wall and th
e rowere the roushed the r
Epoch 220 Average loss: 1.92229 random sample: Santa Claus to the been the so
ugh beaghed to the sough a
Epoch 230 Average loss: 1.85965 random sample: Santa Claus the beat he with t
he sond the cond the conde
Epoch 240 Average loss: 1.93476 random sample: Santa Claus ond ham he has the
Knotken the King and the
Epoch 250 Average loss: 1.90188 random sample: Santa Claus for the wored the
wored the wored the wored
```