2.3 Formula Recognition Using seq2seq Models - Distributed Training

August 5, 2019

1 2.3 Formula Recognition Using seq2seq Models - Distributed training

We mentioned in the last notebook (Notebook 2.2) that use the `tf.data.Dataset` API would allow distributed training. In this notebook, we adapt the previous notebook to work with many GPUs (or TPUs). Note that you will need to have an appropriate machine available in order to use this notebook. To perform the training in this notebook, we used an Amazon `p3.8xlarge` instance.

The Tensorflow guide to distribution was very helpful, as well as some of the examples mentioned in the guide. The following changes were needed: - A distribution strategy had to be defined. Here, we use the `tf.distribute.MirroredStrategy` – to use multiple GPUs. Other strategies are available for using TPUs or multiple machines. - The dataset has to be replicated to mutiple GPUs. Use the `strategy.experimental_distribute_dataset` for this. - `with strategy.scope():` Had to be used to place the Tensorflow graph on multiple devices - A reduction strategy had to be defined for reducing the sum from the replicated data on each GPU.

Also **note** that some of the comments from the previous notebook have been omitted for brevity.

> **WARNING**: Multi-GPU support was finicky, and required a few tweaks. This notebook might not run without some additional effort!

```
In [1]: import tensorflow as tf
```

```
In [2]: gpus = tf.config.experimental.list_physical_devices('GPU')
        if gpus:
            try:
                # Currently, memory growth needs to be the same across GPUs
                for gpu in gpus:
                    print(gpu)
                    tf.config.experimental.set_memory_growth(gpu, True)
                logical_gpus = tf.config.experimental.list_logical_devices('GPU')
                print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
            except RuntimeError as e:
                # Memory growth must be set before GPUs have been initialized
                print(e)
```

```
PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')
PhysicalDevice(name='/physical_device:GPU:1', device_type='GPU')
PhysicalDevice(name='/physical_device:GPU:2', device_type='GPU')
PhysicalDevice(name='/physical_device:GPU:3', device_type='GPU')
4 Physical GPUs, 4 Logical GPUs
```

```
In [3]: print(tf.__version__)
        print(tf.test.is_gpu_available())
        print(tf.test.is_built_with_cuda())
```

```
2.0.0-dev20190804
True
True
```

```
In [4]: # For running on multiple GPUs
        strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1", "/gpu:2","/gpu:3
        print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

```
Number of devices: 4
```

```
In [5]: import os
        import glob

        ### Make sure our data is in order
        data_base_dir = "../data"
        figs_base_dir = "../figs"

        original_data_path = data_base_dir + "/original/formula/"
        processed_data_path = data_base_dir + "/processed/formula/"
        pickle_data_path = data_base_dir + "/pickle/formula/"

        assert os.path.exists(original_data_path), "Original data path does not exist."
```

```
In [6]: import re

        def atoi(text):
            return int(text) if text.isdigit() else text

        def natural_keys(text):
            '''
            alist.sort(key=natural_keys) sorts in human order
            http://nedbatchelder.com/blog/200712/human_sorting.html
            (See Toothy's implementation in the comments)
            '''
            return [ atoi(c) for c in re.split(r'(\d+)', text) ]
```

In [7]:
```
training_images = glob.glob(f"{processed_data_path}/images/train/*.png")
training_images.sort(key=natural_keys)
validation_images = glob.glob(f"{processed_data_path}/images/validate/*.png")
validation_images.sort(key=natural_keys)

print(f"Found {len(training_images)} training images.")
print(f"Found {len(validation_images)} validation images.")
```

```
Found 76303 training images.
Found 8474 validation images.
```

In [8]:
```
import pandas as pd
import numpy as np

def load_labels(labels_path, matches_path):
    with open(labels_path) as f:
        labels = np.array(f.read().splitlines())
    matches =  pd.read_csv(matches_path, sep=' ', header=None).values
    return labels[[list(map(lambda f: f[1], matches))][0]]

train_labels_path = f"{processed_data_path}labels/train.formulas.norm.txt"
train_matches_path = f"{processed_data_path}images/train/train.matching.txt"
train_labels = load_labels(train_labels_path, train_matches_path)
print(f"Got {len(train_labels)} training labels.")

validate_labels_path = f"{processed_data_path}labels/val.formulas.norm.txt"
validate_matches_path = f"{processed_data_path}images/validate/val.matching.txt"
validate_labels = load_labels(validate_labels_path, validate_matches_path)
print(f"Got {len(validate_labels)} validation labels.")
```

```
Got 76303 training labels.
Got 8474 validation labels.
```

In [9]:
```
class Vocab(object):
        def __init__(self, vocab_path):
            self.build_vocab(vocab_path)

        def build_vocab(self, vocab_path):
            '''
            Builds the complete vocabulary, including special tokens
            '''
            self.unk   = "<UNK>"
            self.start = "<SOS>"
            self.end   = "<END>"
            self.pad   = "<PAD>"

            # First, load our vocab from disk & determine
```

```
        # highest index in mapping.
        vocab = self.load_vocab(vocab_path)
        max_index = max(vocab.values())

        # Compile special token mapping
        special_tokens = {
            self.unk : max_index + 1,
            self.start : max_index + 2,
            self.end : max_index + 3,
            self.pad : max_index + 4
        }

        # Merge dicts to produce final word index
        self.token_index = {**vocab, **special_tokens}
        self.reverse_index = {v: k for k, v in self.token_index.items()}

    def load_vocab(self, vocab_path):
        '''
        Load vocabulary from file
        '''
        token_index = {}
        with open(vocab_path) as f:
            for idx, token in enumerate(f):
                token = token.strip()
                token_index[token] = idx
        assert len(token_index) > 0, "Could not build word index"
        return token_index

    def tokenize_formula(self, formula):
        '''
        Converts a formula into a sequence of tokens using the vocabulary
        '''
        def lookup_token(token):
            return self.token_index[token] if token in self.token_index else self.toke
        tokens = formula.strip().split(' ')
        return list(map(lambda f: lookup_token(f), tokens))

    def pad_formula(self, formula, max_length):
        '''
        Pads a formula to max_length with pad_token, appending end_token.
        '''
        # Extra space for the end token
        padded_formula = self.token_index[self.pad] * np.ones(max_length + 1)
        padded_formula[len(formula)] = self.token_index[self.end]
        padded_formula[:len(formula)] = formula
        return padded_formula

    @property
```

```
    def length(self):
        return len(self.token_index)

vocab = Vocab(f"{processed_data_path}/vocab.txt")
```

In [10]:
```
## --- Hyperparameters ---

# When running on multiple GPUs, we can increase the total batch size
batch_size_per_replica = 16
batch_size = 16 * strategy.num_replicas_in_sync

buffer_size = 1000
embedding_dim = 256
vocab_size = vocab.length
hidden_units = 256
num_datapoints = 32768
num_training_steps = num_datapoints // batch_size
num_validation_datapoints = 2000
num_validation_steps = num_validation_datapoints // batch_size
epochs = 62
train_new_model = False
max_image_size=(50,200)
max_formula_length = 130
```

In [11]:
```
# This hash table is used to perform token lookups in the vocab
table = tf.lookup.StaticHashTable(
    initializer=tf.lookup.KeyValueTensorInitializer(
        keys=tf.constant(list(vocab.token_index.keys())),
        values=tf.constant(list(vocab.token_index.values())),
    ),
    default_value=tf.constant(vocab.token_index[vocab.unk]),
    name="class_weight"
)

def load_and_decode_img(path):
    ''' Load the image and decode from png'''
    image = tf.io.read_file(path)
    image = tf.image.decode_png(image)
    return tf.image.rgb_to_grayscale(image)

@tf.function
def lookup_token(token):
    ''' Lookup the given token in the vocab'''
    table.lookup(token)
    return  table.lookup(token)

def process_label(label):
    ''' Split to tokens, lookup & append <END> token'''
```

```
        tokens = tf.strings.split(label, " ")
        tokens = tf.map_fn(lookup_token, tokens, dtype=tf.int32)
        return tf.concat([tokens, [vocab.token_index[vocab.end]]], 0)

    def process_datum(path, label):
        return load_and_decode_img(path), process_label(label)

    # Tokenize formulas
    train_dataset = tf.data.Dataset.from_tensor_slices((training_images, train_labels)).ma
    validation_dataset = tf.data.Dataset.from_tensor_slices((validation_images, validate_
```

In [12]:
```
# Print some values from the dataset (pre-filter)
for datum in train_dataset.take(5):
    print(datum[1])
    print("\n")
```

```
tf.Tensor(
[480 498 473 507  21 509  35   4  20   9 507 213 507 496 478 493 498 428
 509 507 497 509 509   5 473 507 213 507  21 509 507  20   7 121 473 507
  21 509 509 509 248 480 497 473 507  21 509   7 497 473 507  21 509 480
 428 473 507  21 509   7 497 473 507  21 509 498 485 492 473 507  21 509
 428 480 446 473 507  21 509 355   9 507 213 507 480 499 473 507  21 509
 509 507   4  20   9 507 213 507 496 478 493 498 428 509 507 497 509 509
   5 473 507 213 507  21 509 507  20   7 121 473 507  21 509 509 509 509
 509  74  12 514], shape=(130,), dtype=int32)


tf.Tensor(
[465 215 474 507 290 507 484 493 494 482 509 509 392 415 474 507 492  36
  14 509 465 507  45 509 474 507 492 509 507 213 507   4   9 476   5 473
 507 492 509 509 507  21 473 507  21 492   9  20 509 509 509 514], shape=(52,), dtype=int32)


tf.Tensor(
[  4 507 162  50 509 474 507 476 509 483   5 474 507 485 487 509  35  14
   8  68  68  68  68   4 507 162  50 509 474 507 476 509  46   5 474 507
 485 487 488 509  35  14   8 514], shape=(44,), dtype=int32)


tf.Tensor(
[ 58 474 507 498 499 476 499 509  35  21 337 406 507  52 474 507  26 509
 473 507   4  20   5 509  52 474 507  26 509 473 507   4  21   5 509  52
 474 507  26 509 473 507   4  22   5 509 509 253 406 507 492 509   7 406
 507 131 507 492 509 509 363 514], shape=(62,), dtype=int32)


tf.Tensor(
[220 507  52 509 474 507  22 509  35 415 401 482 474 507 487  35  20 509
```

```
 476 474 507 487 509 401 507 178 509 476 474 507 487 509  74  12 514], shape=(35,), dtype=int32
```

To make training faster, we set some constraints on the maximum formula image size.

Similarly, we don't use formulas that are over a certain length – this will also help the vanishing gradient problem with RNNs.

In [13]:
```
def filter_by_size(image, label):
    '''Filter the dataset by the size of the image & length of label'''
    label_length = tf.shape(label)
    image_size = tf.shape(image)

    # Does this image meet our size constraint?
    keep_image = tf.math.reduce_all(
        tf.math.greater_equal(max_image_size, image_size[:2])
    )
    # Does this image meet our formula length constraint?
    keep_label = tf.math.reduce_all(
        tf.math.greater_equal(max_formula_length, label_length[0])
    )
    return tf.math.logical_and(keep_image, keep_label)

train_dataset = train_dataset.filter(filter_by_size).take(num_datapoints)
validation_dataset = validation_dataset.filter(filter_by_size).take(num_validation_da
```

Shuffle and batch (dropping any batches that are < `batch_size` long). Also pad each batch to the largest image size + formulas to max_formula_length.

In [14]:
```
shapes = (tf.TensorShape([None,None,1]),tf.TensorShape([max_formula_length]))
values = (tf.constant(255, dtype=tf.uint8), tf.constant(vocab.token_index[vocab.pad]))
train_dataset = train_dataset.shuffle(buffer_size).padded_batch(
    batch_size,
    padded_shapes=shapes,
    padding_values=values,
    drop_remainder=True
)
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE).cacl

validation_dataset = validation_dataset.shuffle(buffer_size).padded_batch(
    batch_size,
    padded_shapes=shapes,
    padding_values=values,
    drop_remainder=True
)

# Prefetch and cache for performance
validation_dataset = validation_dataset.prefetch(buffer_size=tf.data.experimental.AUT
```

Convert our dataset to a distributed dataset for training on multiple GPUs:

```
In [15]: train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
         validation_dist_dataset = strategy.experimental_distribute_dataset(validation_dataset)
```

```
In [16]: import random
         import textwrap
         import matplotlib.pyplot as plt

         def plot_example(example):
             if example is None:
                 return

             image_tensor, label = example

             label = "".join([vocab.reverse_index[token.numpy()] for token in label])
             label = label.replace("<PAD>","")

             fig = plt.figure(figsize=(20,20))
             plt.imshow(image_tensor[:,:,0], cmap='gray')
             plt.title(textwrap.wrap(label,100))
             plt.show()

         # Plot a few image from a random batch
         for img_tensor, labels in train_dataset.take(5):
             if len(img_tensor.shape) == 4:
                 image = img_tensor[0, :, :, :]
                 label = labels[0]
                 plot_example((image, label))
             else:
                 plot_example((img_tensor, labels))
```

```
<Figure size 2000x2000 with 1 Axes>


<Figure size 2000x2000 with 1 Axes>


<Figure size 2000x2000 with 1 Axes>


<Figure size 2000x2000 with 1 Axes>


<Figure size 2000x2000 with 1 Axes>
```

```
In [17]: from __future__ import division
         import math
```

```python
import numpy as np
from six.moves import xrange
import tensorflow as tf


# taken from https://github.com/tensorflow/tensor2tensor/blob/37465a1759e278e8f073cd0
def add_timing_signal_nd(x, min_timescale=1.0, max_timescale=1.0e4):
    """Adds a bunch of sinusoids of different frequencies to a Tensor.

    Each channel of the input Tensor is incremented by a sinusoid of a difft
    frequency and phase in one of the positional dimensions.

    This allows attention to learn to use absolute and relative positions.
    Timing signals should be added to some precursors of both the query and the
    memory inputs to attention.

    The use of relative position is possible because sin(a+b) and cos(a+b) can
    be expressed in terms of b, sin(a) and cos(a).

    x is a Tensor with n "positional" dimensions, e.g. one dimension for a
    sequence or two dimensions for an image

    We use a geometric sequence of timescales starting with
    min_timescale and ending with max_timescale.  The number of different
    timescales is equal to channels // (n * 2). For each timescale, we
    generate the two sinusoidal signals sin(timestep/timescale) and
    cos(timestep/timescale).  All of these sinusoids are concatenated in
    the channels dimension.

    Args:
        x: a Tensor with shape [batch, d1 ... dn, channels]
        min_timescale: a float
        max_timescale: a float

    Returns:
        a Tensor the same shape as x.

    """
    static_shape = x.get_shape().as_list()
    num_dims = len(static_shape) - 2
    channels = tf.shape(x)[-1]
    num_timescales = channels // (num_dims * 2)
    log_timescale_increment = (
            math.log(float(max_timescale) / float(min_timescale)) /
            (tf.cast(num_timescales, tf.float32) - 1))
    inv_timescales = min_timescale * tf.exp(
            tf.cast(tf.range(num_timescales), tf.float32) * -log_timescale_increment)
    for dim in xrange(num_dims):
```

```
            length = tf.shape(x)[dim + 1]
            position = tf.cast(tf.range(length), tf.float32)
            scaled_time = tf.expand_dims(position, 1) * tf.expand_dims(
                    inv_timescales, 0)
            signal = tf.concat([tf.sin(scaled_time), tf.cos(scaled_time)], axis=1)
            prepad = dim * 2 * num_timescales
            postpad = channels - (dim + 1) * 2 * num_timescales
            signal = tf.pad(signal, [[0, 0], [prepad, postpad]])
            for _ in xrange(1 + dim):
                signal = tf.expand_dims(signal, 0)
            for _ in xrange(num_dims - 1 - dim):
                signal = tf.expand_dims(signal, -2)
            x += signal
        return x
```

In [18]:
```
from tensorflow.keras import metrics, layers, Model

class BahdanauAttention(layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, features, hidden):
        # First, flatten the image
        shape = tf.shape(features)
        if len(shape) == 4:
            batch_size = shape[0]
            img_height = shape[1]
            img_width  = shape[2]
            channels   = shape[3]
            features = tf.reshape(features, shape=(batch_size, img_height*img_width, c
        else:
            print(f"Image shape not supported: {shape}.")
            raise NotImplementedError

        # features(CNN_encoder_output)
        # shape => (batch_size, flattened_image_size, embedding_size)

        # hidden shape == (batch_size, hidden_size)
        # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
        hidden_with_time_axis = tf.expand_dims(hidden, 1)

        # score
        # shape => (batch_size, flattened_image_size, hidden_size)
        score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))
```

```
            # attention_weights
            # shape => (batch_size, flattened_image_size, 1)
            attention_weights = tf.nn.softmax(self.V(score), axis=1)

            # context_vector
            # shape after sum => (batch_size, hidden_size)
            context_vector = attention_weights * features
            context_vector = tf.reduce_sum(context_vector, axis=1)

            return context_vector, attention_weights
```

In [19]:
```
class CNNEncoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        super(CNNEncoder, self).__init__()

        self.fc = tf.keras.layers.Dense(embedding_dim)

    def build(self, input_shape):
        self.cnn_1 = layers.Conv2D(64, (3, 3), activation='relu', input_shape=(input_s
        self.max_pool_1 = layers.MaxPooling2D((2, 2))

        self.cnn_2 = layers.Conv2D(256, (3, 3), activation='relu')
        self.max_pool_2 = layers.MaxPooling2D((2, 2))

        self.cnn_3 = layers.Conv2D(512, (3, 3), activation='relu')

    def call(self, images):
        images = tf.cast(images, tf.float32)
        x = self.cnn_1(images)
        x = self.max_pool_1(x)
        x = self.cnn_2(x)
        x = self.max_pool_2(x)
        x = self.cnn_3(x)
        return add_timing_signal_nd(x)
```

In [20]:
```
class RNNDecoder(tf.keras.Model):
    def __init__(self, embedding_dim, units, vocab_size):
        super(RNNDecoder, self).__init__()
        self.units = units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.compat.v1.keras.layers.CuDNNGRU(self.units,
                                        return_sequences=True,
                                        return_state=True,
                                        recurrent_initializer='glorot_uniform')

        self.fc1 = tf.keras.layers.Dense(self.units)
        self.fc2 = tf.keras.layers.Dense(vocab_size)
        self.attention = BahdanauAttention(self.units)
```

```
    def call(self, x, features, hidden):
        # attend over the image features
        context_vector, attention_weights = self.attention(features, hidden)

        # convert our input vector to an embedding
        x = self.embedding(x)

        # concat the embedding and the context vector (from attention)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # pass to GRU
        output, state = self.gru(x)

        x = self.fc1(output)

        x = tf.reshape(x, (-1, x.shape[2]))

        # This produces a distribution over the vocab
        x = self.fc2(x)

        return x, state, attention_weights


    def reset_state(self, batch_size):
        return tf.zeros((batch_size, self.units))
```

In [21]:
```
with strategy.scope():
    encoder = CNNEncoder(embedding_dim=embedding_dim)
    decoder = RNNDecoder(embedding_dim, hidden_units, vocab_size)
```

```
WARNING: Logging before flag parsing goes to stderr.
W0805 13:39:27.692402 140018678175488 module_wrapper.py:136] From /home/ubuntu/anaconda3/lib/py
```

In [22]:
```
with strategy.scope():
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)

    # reduction='none' allows us to perform the reduction manually afterwards for our
    # multi-GPU scenario
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, redu

    def loss_function(real, pred):
        # In order to avoid <PAD> tokens contributing to the loss, we mask those toke
        # First, we create the mask, and compute the loss.
        mask = tf.math.logical_not(tf.math.equal(real, vocab.token_index[vocab.pad]))
```

```
            loss_ = loss_object(real, pred)

            # Second, we multiply the computed loss by the mask to zero out contributions
            mask = tf.cast(mask, dtype=loss_.dtype)
            loss_ *= mask
            return tf.reduce_mean(loss_)
```

```
In [38]: checkpoint_path = "./checkpoints/train"
         checkpoint = tf.train.Checkpoint(encoder=encoder,
                                          decoder=decoder,
                                          optimizer=optimizer)
         checkpoint_manager = tf.train.CheckpointManager(checkpoint, checkpoint_path, max_to_ke

         # Attempt to restore from training checkpoint
         start_epoch = 0
         save_at_epoch = 5
         if train_new_model is False and checkpoint_manager.latest_checkpoint:
             checkpoint.restore(checkpoint_manager.latest_checkpoint)
             start_epoch = int(checkpoint_manager.latest_checkpoint.split('-')[-1])*save_at_epo
             print(f"Restored from checkpoint: {checkpoint_manager.latest_checkpoint}.")
             print(f"Start epoch: {start_epoch}.")
         else:
             print("Did not restore from a checkpoint -- training new model!")
```

```
Restored from checkpoint: ./checkpoints/train/ckpt-12.
Start epoch: 60.
```

```
In [33]: with strategy.scope():
             train_epoch_losses = []
             validation_epoch_losses = []
```

```
In [34]: with strategy.scope():
             def train_step(img_tensor, target):
                 ''' Function that encapsulates training logic'''
                 loss = 0

                 # reset the decoder state, since Latex is different for each image
                 hidden = decoder.reset_state(batch_size=target.shape[0])

                 # shape => (batch_size, 1)
                 dec_input = tf.expand_dims([vocab.token_index[vocab.start]] * batch_size_per_r

                 sequence_length = target.shape[1]
                 with tf.GradientTape() as tape:
                     features = encoder(img_tensor)

                     for i in range(0, sequence_length):
                         # passing the features through the decoder
```

```
                    predictions, hidden, _ = decoder(dec_input, features, hidden)

                    ground_truth_token = target[:, i]
                    loss += loss_function(ground_truth_token, predictions)

                    # Teacher forcing: feed the correct word in as the next input to the
                    # encoder, to provide the decoder with the proper context to predict
                    # the following token in the sequence
                    dec_input = tf.expand_dims(ground_truth_token, 1)

            total_loss = (loss / int(sequence_length))
            trainable_variables = encoder.trainable_variables + decoder.trainable_variable
            gradients = tape.gradient(loss, trainable_variables)
            optimizer.apply_gradients(zip(gradients, trainable_variables))
            return loss, total_loss
```

In [35]:
```
with strategy.scope():
        def validate_step(img_tensor, target):
            ''' Function that encapsulates training logic'''
            loss = 0

            # reset the decoder state, since Latex is different for each image
            hidden = decoder.reset_state(batch_size=target.shape[0])
            # shape => (batch_size, 1)
            dec_input = tf.expand_dims([vocab.token_index[vocab.start]] * batch_size_per_r

            sequence_length = target.shape[1]
            features = encoder(img_tensor)
            for i in range(0, sequence_length):
                # passing the features through the decoder
                predictions, hidden, _ = decoder(dec_input, features, hidden)

                ground_truth_token = target[:, i]
                loss += loss_function(ground_truth_token, predictions)

                # Teacher forcing: feed the correct word in as the next input to the
                # encoder, to provide the decoder with the proper context to predict
                # the following token in the sequence
                dec_input = tf.expand_dims(ground_truth_token, 1)

            total_loss = (loss / int(sequence_length))
            return loss, total_loss
```

In [ ]:
```
import time
import datetime

with strategy.scope():
      # `experimental_run_v2` replicates the provided computation and runs it
```

```python
# with the distributed input.
@tf.function
def distributed_train_step(img_tensor, target):
    per_replica_batch_losses, per_replicate_sequence_losses = strategy.experimental
    return (
        strategy.reduce(tf.distribute.ReduceOp.MEAN, per_replica_batch_losses, axis
        strategy.reduce(tf.distribute.ReduceOp.MEAN, per_replicate_sequence_losses
    )

@tf.function
def distributed_validation_step(img_tensor, target):
    per_replica_batch_losses, per_replicate_sequence_losses = strategy.experimental
    return (
        strategy.reduce(tf.distribute.ReduceOp.MEAN, per_replica_batch_losses, axis
        strategy.reduce(tf.distribute.ReduceOp.MEAN, per_replicate_sequence_losses
    )

print(f"[Started training at: {datetime.datetime.now()}. Training for {epochs} epoc
print(f"[Starting epoch: {start_epoch}]")
for epoch in range(start_epoch, epochs):
    start = time.time()

    total_loss = 0.0
    num_batches = 0

    # Training loop
    for img_tensor, target in train_dist_dataset:
        batch_loss, sequence_loss = distributed_train_step(img_tensor, target)
        total_loss += batch_loss
        num_batches += 1

        if num_batches % 50 == 0:
            print(f"[Epoch: {epoch + 1} | Batch: {num_batches} | Loss: {batch_loss

    train_loss = total_loss / num_batches

    # Print training loss before starting validation loop
    print(f"[Epoch: {epoch + 1} | Training loss: {train_loss}]")

    total_loss = 0.0
    num_batches = 0

    # Validation loop
    for img_tensor, target in validation_dist_dataset:
        batch_loss, sequence_loss =  distributed_validation_step(img_tensor, targe
        total_loss += batch_loss
        num_batches += 1
```

```
            validation_loss = total_loss / num_batches
            print(f"[Epoch: {epoch + 1} | Validation loss: {validation_loss}]")

            # Save epoch losses
            train_epoch_losses.append(train_loss)
            validation_epoch_losses.append(validation_loss)

            # Save checkpoint (if required)
            if epoch % save_at_epoch == 0:
                checkpoint_manager.save()


            print(f"[Time elapsed for epoch: {format(time.time() - start)} seconds.] \n")
```

In [ ]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(5,5))
plt.plot(train_epoch_losses)
plt.plot(validation_epoch_losses)
plt.legend(["Training loss", "Validation loss"])
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Plot')
plt.savefig(f"{figs_base_dir}/{datetime.datetime.now()}.png")
plt.show()
```

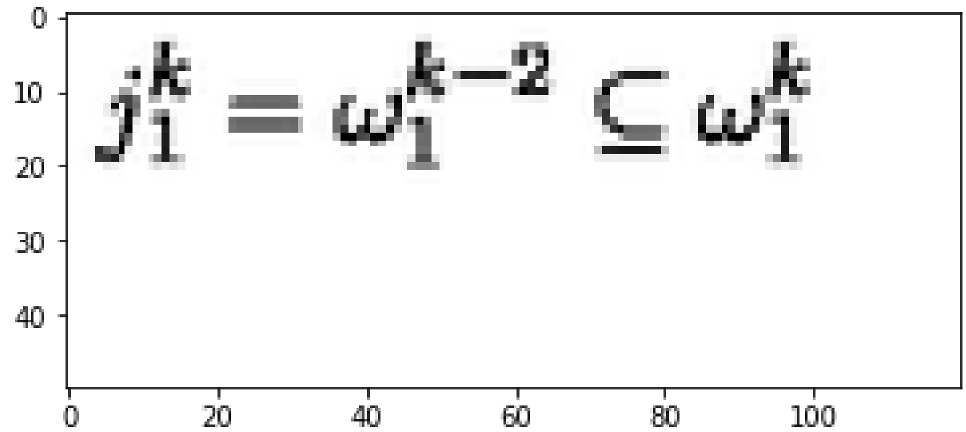
In [219]:
```
import matplotlib.pyplot as plt

def load_image(path):
    img_raw = tf.io.read_file(path)
    return tf.image.decode_png(img_raw)


def plot_features(features, title):
    layer_count = features.shape[3]

    num_rows = 16
    num_cols = int(layer_count / num_rows)

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(32, 45))
    for i, ax in enumerate(axes.flat):
        feat = features[0,:,:,i]
        ax.imshow(feat)
        ax.axis('off')
    plt.tight_layout(rect=[0, 0.03, 1, 0.97])

    plt.rc('text', usetex=True)
    plt.suptitle(f"CNN Activations for ${title}$", fontsize=24)
```

```
    plt.rc('text', usetex=False)

    plt.savefig(f"{figs_base_dir}/activations-{title}.png")
    plt.show()

def plot_attention(attn):
    layer_count = attn.shape[0]

    num_rows = 16
    num_cols = math.floor(layer_count / num_rows)

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(32, 20))
    for i, ax in enumerate(axes.flat):
        if i >= layer_count:
            break
        feat = attn[i,:,:]
        ax.imshow(feat)
        ax.axis('off')
    plt.show()

def evaluate(img, max_formula_length):
    index_token_mapping = {v: k for k, v in vocab.token_index.items()}

    hidden = decoder.reset_state(batch_size=1)

    # Convert to grayscale so network can process it
    # and add a dimension at the start to simulate batch (size=1)
    image = tf.image.rgb_to_grayscale(img)
    temp_input = tf.expand_dims(image, 0)

    # Pass through encoder to obtain features
    features = encoder(temp_input)

    # Signal to the decoder that we are starting feeding in a new sequence
    decoder_input = tf.expand_dims(vocab.tokenize_formula(vocab.start), 1)

    result = []
    attention_plot = np.zeros((max_formula_length, features.shape[1], features.shape
    for i in range(max_formula_length):
        predictions, hidden, attention_weights = decoder(decoder_input, features, hid

        #attn weights: (1, feat_width * feat_height, 1)
        attention_plot[i] = tf.reshape(attention_weights, (features.shape[1], feature

        predicted_id = tf.argmax(predictions[0]).numpy()
        result.append(index_token_mapping[predicted_id])

        if index_token_mapping[predicted_id] == vocab.end:
```

```
                return result[:-1], attention_plot, features

            decoder_input = tf.expand_dims([predicted_id], 0)
        return result, attention_plot, features
```

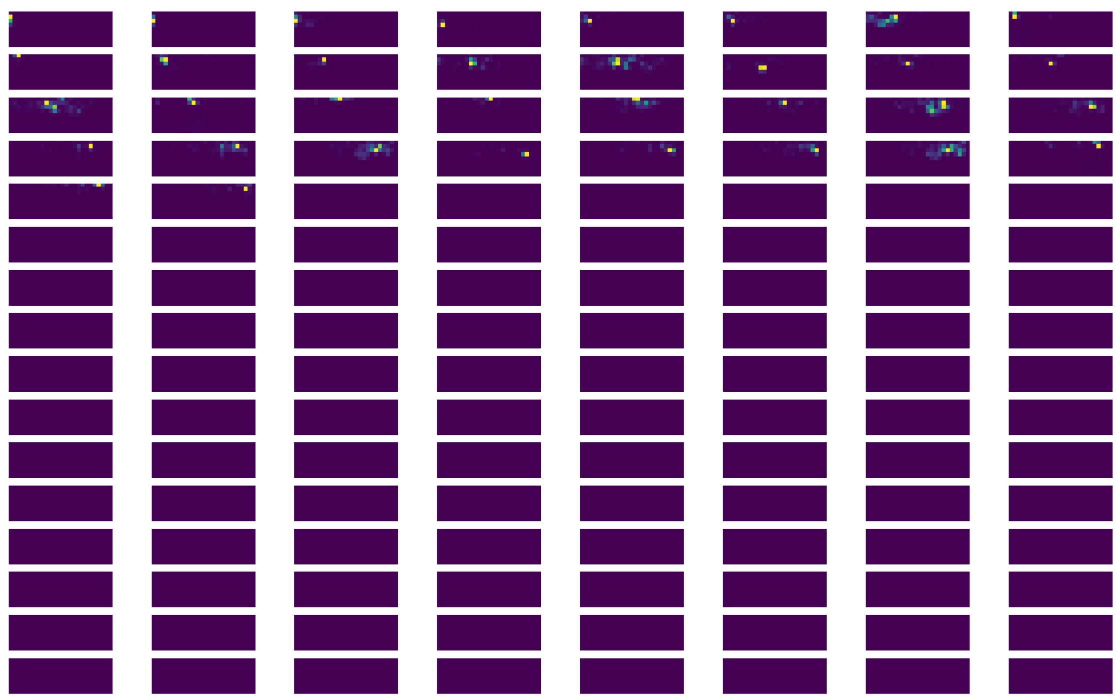
In [239]:
```
import subprocess

def create_from_template(latex):
    '''Uses a simple template to create a valid latex document'''
    pre = """
    \documentclass[preview]{standalone}
    \\begin{document}
    \\begin{equation}
    """

    post = """
    \end{equation}
    \end{document}
    """
    return pre + latex + post


def render_latex(latex):
    ''' Renders a latex string, creating a png'''
    # Write temp tex file
    latex_out_dir = f"{processed_data_path}latex/"
    temp_tex_filepath = f"{processed_data_path}latex/tmp.tex"
    temp_dvi_filepath = os.path.splitext(temp_tex_filepath)[0]+'.dvi'
    temp_png_filepath = os.path.splitext(temp_tex_filepath)[0]+'1.png'

    with open(temp_tex_filepath, 'w+') as f:
        f.seek(0)
        f.write(create_from_template(latex))

    # Render it
    cmd = ['latex', '-interaction=nonstopmode', "-output-directory", latex_out_dir, t
    output = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    # Log stdout & stderror on error, but attempt to continue
    if output.returncode != 0:
        print(f"Could not render latex (status code={output.returncode}. Output follo
        print(output.stdout)
        print(output.stderr)
        print("\n")

    render_png_cmd = ['dvipng','-D','300', "-o", temp_png_filepath, temp_dvi_filepat
    output = subprocess.run(render_png_cmd, stdout=subprocess.PIPE, stderr=subproces
    if output.returncode != 0:
```

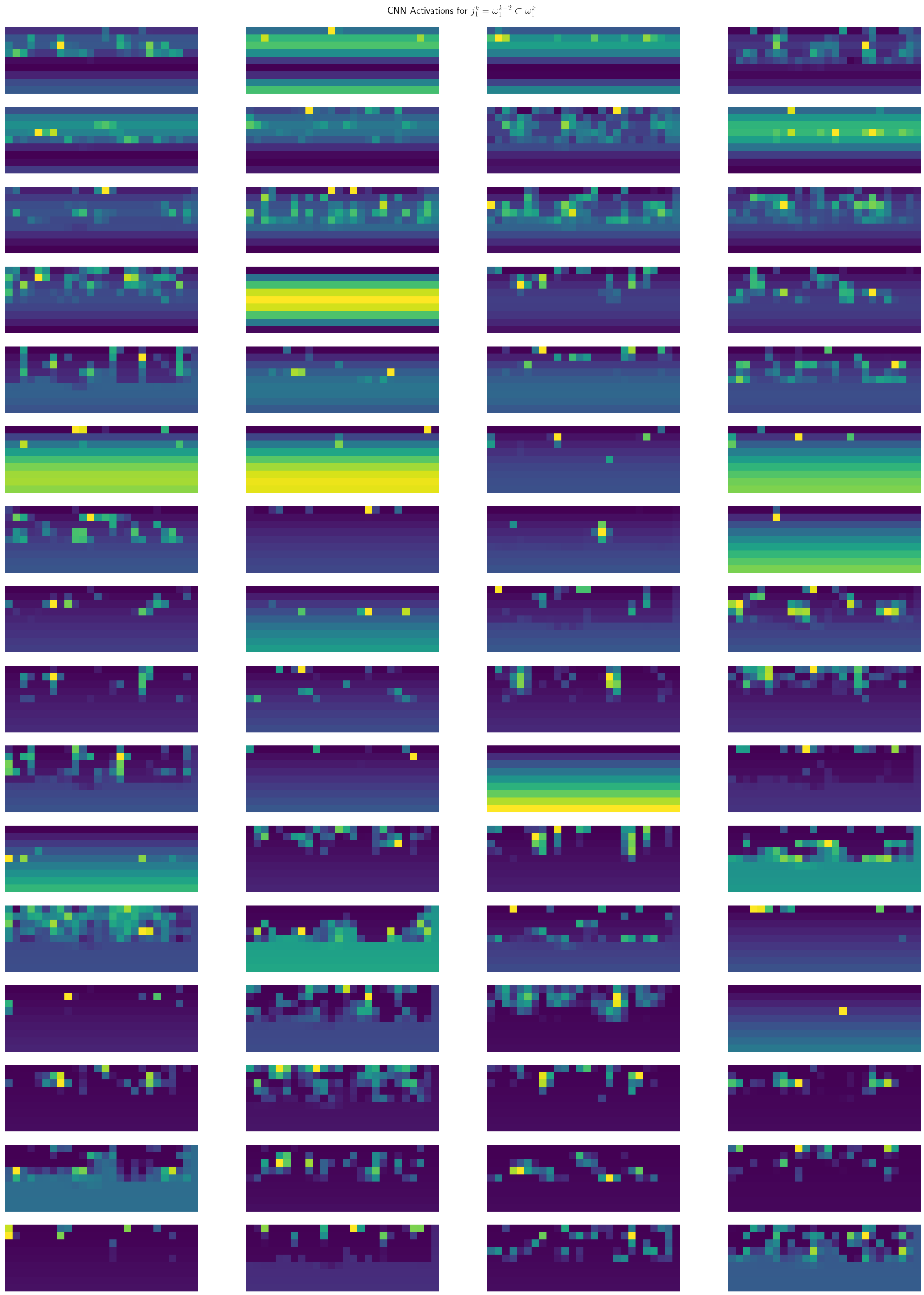
```
        print("Could not render .dvi file into .pdf! Output follows: \n")
        print(output.stdout)
        print(output.stderr)
        print("\n")
        return None
    return temp_png_filepath

def plot_image_clean(png):
    ''' plots an image without axes'''
    fig = plt.figure(figsize=(20,10))
    ax = fig.add_axes([0, 0, 1, 1])
    ax.imshow(plt.imread(out_file))
    ax.axis('off')
    plt.savefig(f"{figs_base_dir}/rendered-latex-{datetime.datetime.now()}.png")
    plt.show()



def process_latex(latex):
    return " ".join(latex)

# Example params
test_index = 21
test_img = load_image(validation_images[test_index])

# Make prediction
result, attention_plot, features = evaluate(test_img, max_formula_length)

# Plot original image
plt.imshow(test_img)
plt.savefig(f"{figs_base_dir}/original-process_latex(result){datetime.datetime.now()
plt.show()

# Render latex from the predicted formula
latex = process_latex(result)
out_file = render_latex(latex)
if out_file is not None:
    plot_image_clean(out_file)

print(f"Predicted formula: \n {process_latex(result)}")
plot_attention(attention_plot)
plot_features(features[:,:,:,0:64])
```



$$j_1^k = \omega_1^{k-2} \subset \omega_1^k \tag{1}$$

```
Predicted formula:
 j _ { 1 } ^ { k } = \omega _ { 1 } ^ { k - 2 } \subset { \omega } _ { 1 } ^ { k }
```

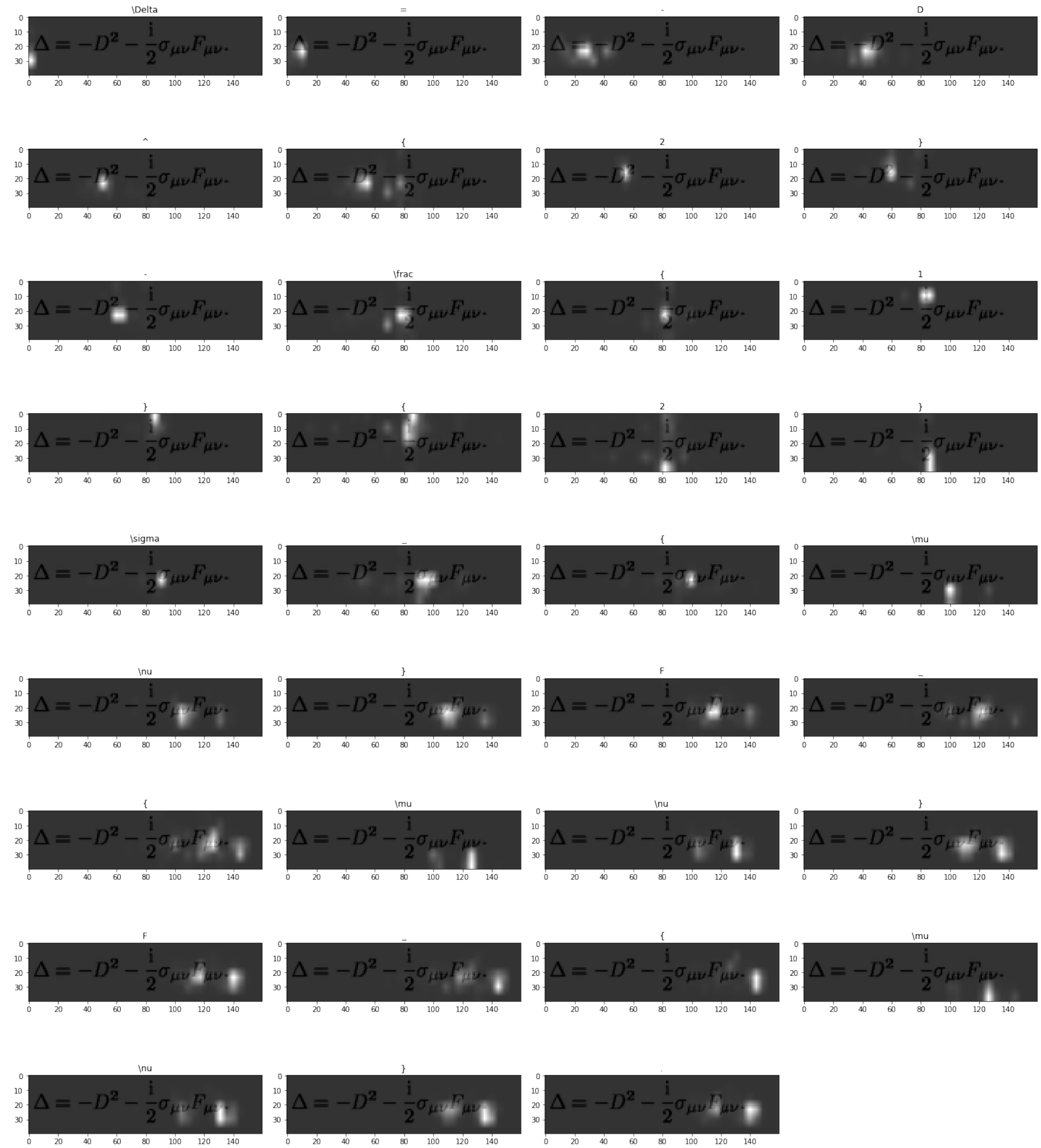





```
In [235]: import cv2
```

```python
import re
from matplotlib import rc

def plot_attention_overlaid(image, title, result, attention_plot, row_length = 4):
    temp_image = np.array(image)

    # Compute number of rows
    len_result = len(result)
    row_count = math.ceil(len_result / row_length)

    # Size plot depending on number of rows
    fig = plt.figure(figsize=(20, 2.75*row_count))

    plt.rc('text', usetex=True)
    fig.suptitle(title, color="black", fontsize=30)
    plt.rc('text', usetex=False)

    tokens_plotted_count = 0
    for l in range(len_result):
        # Ignore empty space -- it doesn't make for a great plot
        token = result[l]
        if token == " ":
            continue

        # Plot image with overlaid attention
        attention_im = cv2.resize(attention_plot[l], (test_img.shape[1], test_img.sha
        ax = fig.add_subplot(row_count, row_length, tokens_plotted_count+1, facecolo
        img = ax.imshow(temp_image,  cmap='gray')
        ax.imshow(attention_im, cmap='gray', alpha=0.8, extent=img.get_extent())
        ax.set_title(result[l], color="black")
        tokens_plotted_count += 1
    plt.tight_layout(rect=[0, 0.03, 1, 0.97])
    plt.savefig(f"{figs_base_dir}/{title}-{datetime.datetime.now()}.pdf")
    plt.show()


plot_attention_overlaid(test_img, f"${process_latex(result)}$", result, attention_plo
```

$$\Delta = -D^2 - \frac{1}{2}\sigma_{\mu\nu}F_{\mu\nu}F_{\mu\nu}.$$




```
In [ ]:
```