ETH zürich

Pretrained Transformers As Universal Computation Engines

Antonio Lopardo

Advanced Topics in ML & DS Seminar

23 March 2022

Pretrained Transformers As Universal Computation Engines

Kevin Lu
UC Berkeley
kzl@berkeley.edu

Aditya Grover
Facebook AI Research
adityagrover@fb.com

Pieter Abbeel
UC Berkeley
pabbeel@cs.berkeley.edu

Igor Mordatch
Google Brain
imordatch@google.com

Presentation Roadmap

Intro - Objective of the Paper

Does pretraining a transformer architecture on natural language

help it generalize to different modalities?

Experiments - Tasks and Results

Bit memory, Bit XOR, ListOps, MNIST, CIFAR-10, CIFAR-10 LRA, Remote Homology

Ablation Study - Conclusions and Critique

Which layers to train? Natural language - Random - Image pretraining?
Underfitting and overfitting?

Intro - The Transformer Architecture

Tasks that Large language models can deal with

Cloze, Completion Tasks, Common Sense Reasoning, Reading Comprehension, NLI, Synthetic and Qualitative Tasks …

Vision Transformer

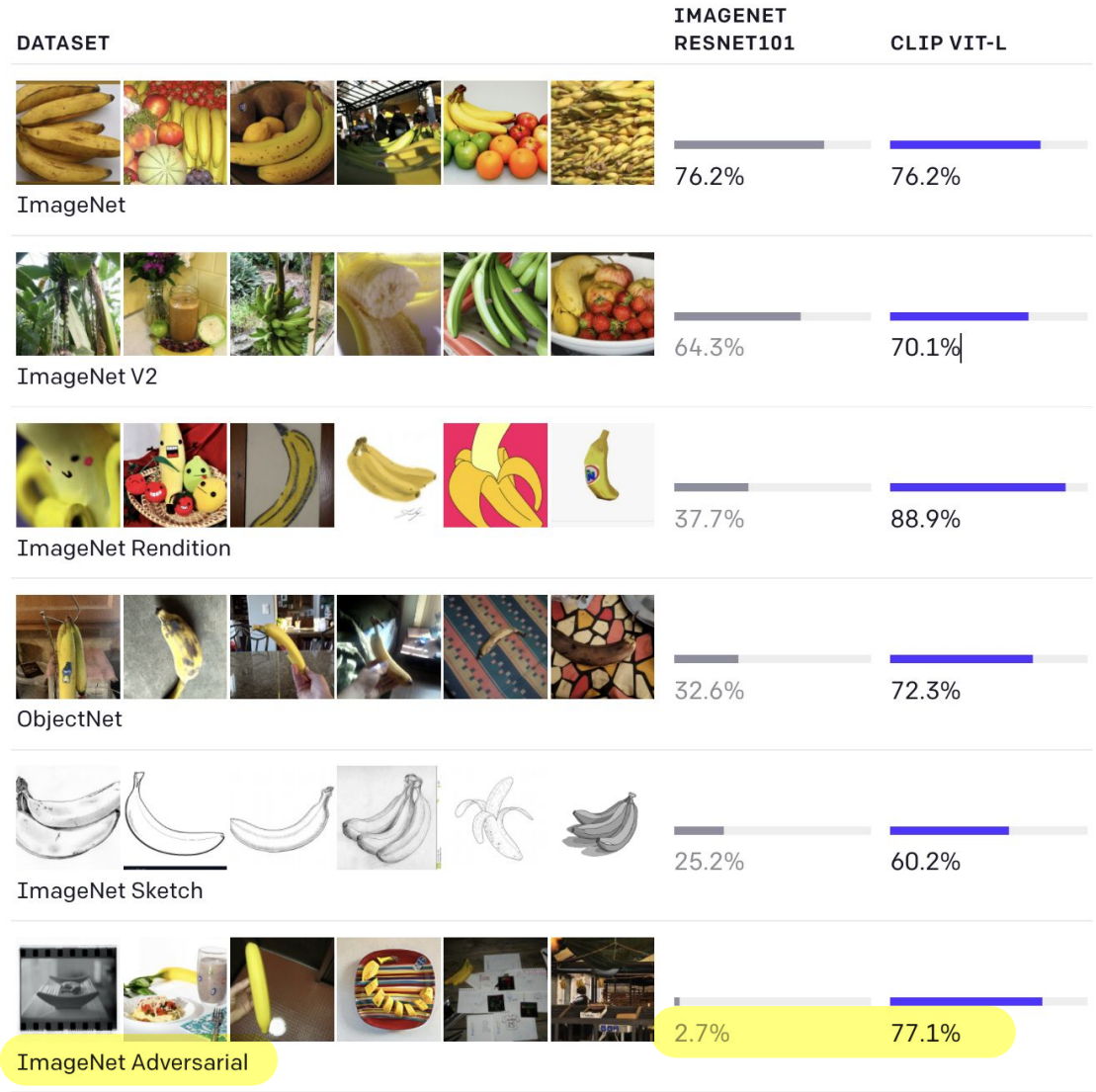
Object detection, Video Deepfake Detection, Image Synthesis

Any task where the input can be fed in as a sequence

Intro - NL improves generalization on CV

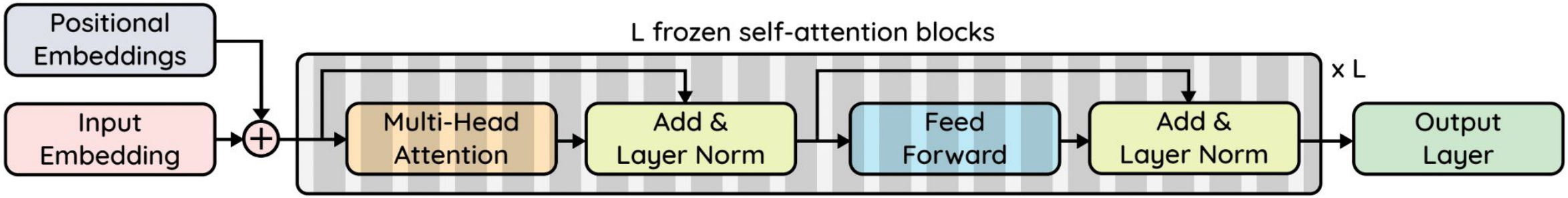
CLIP (Radford et al - 2021)

"... predicting which caption goes with which image is an efficient and scalable way to learn SOTA image representations from scratch ..."



Intro - How to test on other modalities

The Frozen Pretrained Transformer



Finetune the output, input and layer norm layers and the positional encoding, < 0.1% of total parameters of the model

Freeze Multi-Head Attention layers and Feed Forward

Experiments - Bit Memory, XOR, ListOps

Bit Memory

Reconstruct binary sequence pattern from corrupted input

XOR

Perform element-wise XOR

ListOps

Hierarchical operations over lists

INPUT: [MAX 4 3 [MIN 2 3] 1 0 [MEDIAN 1 5 8 9, 2]] **OUTPUT:** 5

Are they relevant or too easy even for the frozen model?

Experiments - MNIST, CIFAR-10, Protein Fold

MNIST

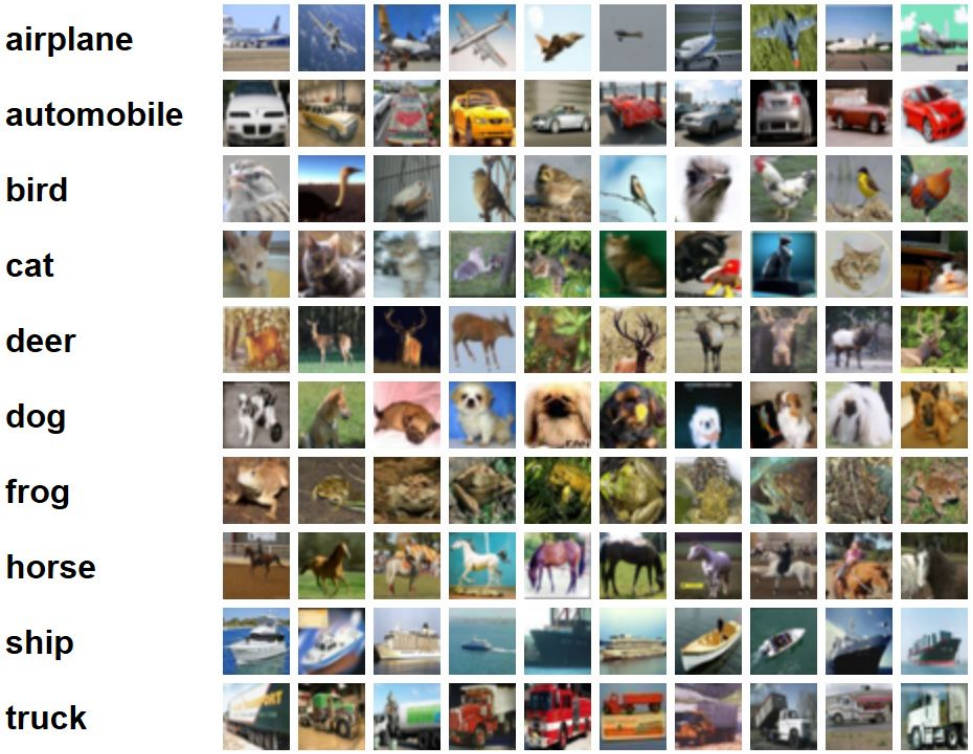
Handwritten digit recognition, 32x32 images

CIFAR-10 & CIFAR-10 LAR

Image classification tasks, 10 classes, 32x32 images

Remote Homology Detection

Predicting the fold of a protein, represented as an amino acid sequence



Are they difficult enough?

Experiments - Results

Model	Bit Memory	XOR	ListOps	MNIST	CIFAR-10	C10 LRA	Homology
FPT	100%	100%	38.4%	98.0%	72.1%	38.6%	12.7%
Full	100%	100%	38%	99.1%	70.3%	42%	9%
LSTM	60.9%	50.1%	17.1%	99.5%	73.6%	11.7%	12%

"We believe that these results support the idea that these models are learning representations and performing computation that is agnostic of the modality."

FPT's CIFAR-10 result competitive with CNN with same number of trainable parameters

Ablation - Importance of the transformer

Model	Bit Memory	XOR	ListOps	MNIST	CIFAR-10	C10 LRA	Homology
Trans.	75.8%	100%	34.3%	91.7%	61.7%	36.1%	9.3%
LSTM	50.9%	50.0%	16.8%	70.9%	34.4%	10.4%	6.6%
LSTM*	75.0%	50.0%	16.7%	92.5%	43.5%	10.6%	8.6%

All randomly initialized, tuning same params as FPT

The paper does show that the transformer architecture is better than the LSTM architecture and that residual connections are very important

Ablation - Importance of language pretraining

Model	Bit Memory	XOR	ListOps	MNIST	C10	C10 LRA	Homology
FPT	100%	100%	38.4%	98.0%	68.2%	38.6%	12.7%
Random	75.8%	100%	34.3%	91.7%	61.7%	36.1%	9.3%
Bit	100%	100%	35.4%	97.8%	62.6%	36.7%	7.8%
ViT	100%	100%	37.4%	97.8%	72.5%	43.0%	7.5%

Random no pretraining just finetuning

Bit pretraining on Bit Memory task

Vit pretrained on 224x224 ImageNet-21k

The paper does show that language pretraining is better than Random, Bit and ImageNet pretraining

Preliminary Paper's Conclusions

The transformer architecture is better than the LSTM architecture and residual connections are very relevant especially as we increase depth

Natural Language pretraining is better than Random, Bit sequence memory and ImageNet pretraining

Due Diligence - Other relevant experiments

Initialization	Memory	XOR	ListOps	MNIST	C10	C10 LRA	Homology
Pretrained	100%	100%	38.4%	98.0%	68.2%	38.6%	12.7%
Statistics Only	100%	100%	37.4%	97.2%	56.5%	33.1%	11.0%
Default	75.8%	100%	34.3%	91.7%	61.7%	36.1%	9.3%

Better statistics for initialization

"… we ablate taking the layer-wise mean and standard deviation from the pretrained model and using it to initialize a random transformer…"

Due Diligence - Not training Layer norm

Output ~ 7680

Input ~ 13056

L norm ~ 3072 * nr layers

Pos Emb ~ 49512

Task	output only	output + input	output + positions	output + layernorm
Bit Memory	75%	75%	75%	75%
Bit XOR	50%	51%	59%	**100%**
ListOps	17%	17%	18%	**35%**
MNIST	25%	28%	34%	**83%**
CIFAR-10	20%	24%	21%	**46%**
CIFAR-10 LRA	11%	16%	12%	**34%**
Homology	2%	2%	6%	**9%**

Table 20: Finetuning individual types of parameters for random frozen transformers.

Task	output only	output + input	output + positions	output + layernorm
Bit Memory	76%	**98%**	93%	94%
Bit XOR	56%	72%	84%	**98%**
ListOps	15%	17%	35%	**36%**
MNIST	23%	85%	93%	**96%**
CIFAR-10	25%	53%	38%	**54%**
CIFAR-10 LRA	17%	22%	30%	**39%**
Homology	2%	8%	8%	**9%**

Table 18: Ablation by only finetuning individual types of parameters for pretrained frozen transformers. We bold the most important parameter (measured by highest test accuracy) for each task.

Task	Base (FPT)	+ Finetuning All FF Layers	+ Finetuning Last Attn Layer
CIFAR-10	68.2%	76.6%	80.0%

Due Diligence - Token Mixing and Efficiency

Number of Layers	Pretrained	Random
1	12%	-
3	18%	-
6	33%	-
12	33%	17%
24	-	17%

Table 12: Test accuracy on Listops while varying model depth and only training output parameters. Even for a large number of layers, the random model does not learn to perform well.

Model	Memory	XOR	ListOps	MNIST	C10	C10 LRA	Homology
FPT	1×10^4	5×10^2	2×10^3	5×10^3	4×10^5	3×10^5	1×10^5
Random	4×10^4	2×10^4	6×10^3	2×10^4	4×10^5	6×10^5	1×10^5
Speedup	4×	40×	3×	4×	1×	2×	1×

Table 6: Approximate number of gradient steps until convergence for pretrained (FPT) vs randomly initialized (Random) models. Note that we use the same batch size and learning rate for both models.

Critiques

Not fully conclusive results with inconsistent use of different model sizes

Hard to argue that training layer norm layers still keeps experiments in the finetuning regime

Despite comparative simplicity of the tasks, the FPT models still underfit

Discussion section

Pretrained Transformers As Universal Computation Engines

Kevin Lu
UC Berkeley
kzl@berkeley.edu

Aditya Grover
Facebook AI Research
adityagrover@fb.com

Pieter Abbeel
UC Berkeley
pabbeel@cs.berkeley.edu

Igor Mordatch
Google Brain
imordatch@google.com

Multimodal Neurons in ANNs (Goh et al, 2021)

BIOLOGICAL NEURON

Probed via depth electrodes

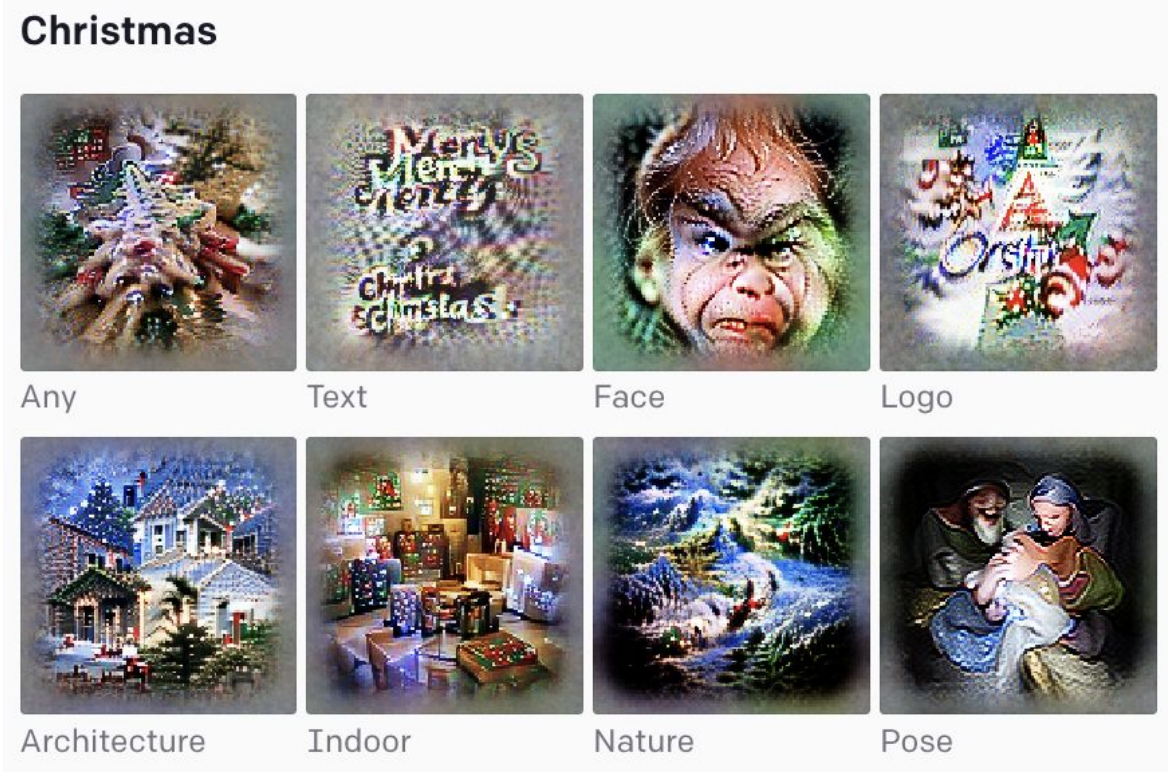
CLIP NEURON

Neuron 244 from penultimate layer in CLIP RN50x4

PREVIOUS ARTIFICIAL NEURON

Neuron 483, generic person detector from Inception v1

Halle Berry	Spider-Man	human face
 Responds to photos of Halle Berry and Halle Berry in costume ✓	 Responds to photos of Spider-Man in costume and spiders ✓ view more	 Responds to photos of human faces ✓
 Responds to skeches of Halle Berry ✓	 Responds to comics or drawings of Spider-Man and spider-themed icons ✓ view more	Does not respond significantly to drawings of faces ✗
 Responds to the text "Halle Berry" ✓	 Responds to the text "spider" and others ✓ view more	Does not respond significantly to text ✗



Layer Norm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Gamma and Beta are trainable parameters

Unlike BatchNorm it operates sample by sample

Better for sequence data

No discrepancy between training and testing

D Details by Table

For clarity, we explicitly write out finer details for some experiment sections where numbers can represent different model types.

D.1 Can pretrained language models transfer to different modalities?

This section refers to Table 1 in Section 3.1

Bit Memory

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params).
2. Full: 12-layer base size GPT-2 model (training all params).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

Bit XOR

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params).
2. Full: 12-layer base size GPT-2 model (training all params).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

ListOps

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params).
2. Full: number reported from Tay et al. (2020) (3-layer vanilla transformer).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

CIFAR-10

1. FPT: 36-layer large size FPT model (finetuning input, output, position, and layernorm params).
2. Full: 3-layer, 768 hidden dimension GPT-2 model (training all params).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

CIFAR-10 LRA

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params).
2. Full: number reported from Tay et al. (2020) (3-layer vanilla transformer).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

Remote Homology

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params).
2. Full: number reported from Rao et al. (2019) (12-layer, 512 hidden dimension vanilla transformer).
3. LSTM: 3-layer, 768 hidden dimension LSTM model (training all params).

D.2 What is the importance of the pretraining modality?

This section refers to Table 2 in Section 3.2.

All tasks

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params). This differs from Table 1, Section 3.1 only in the CIFAR-10 model size.
2. Random: 12-layer randomly initialized (default scheme) base size GPT-2 model (training input, output, position, and layernorm params).
3. Bit: 12-layer base size GPT-2 model (finetuning input, output, position, and layernorm params), after first being fully finetuned on Bit Memory following default random initialization.
4. ViT: 12-layer, 768 hidden dimension base size ViT model (finetuning input, output, position, and layernorm params), pretrained on 224×224 ImageNet-21k with a patch size of 16. (`vit_base_patch16_224` from the `timm` Pytorch library (Wightman, 2019)). We reinitialize the input layer from scratch to match each task, and do not use a CLS token or an MLP as the output network – instead using a linear layer from the last token – matching the protocol for the other methods.

D.3 How important is the transformer architecture compared to LSTM architecture?

The following refer to Section 3.3. In Table 3

All tasks

1. Trans: 12-layer randomly initialized (default scheme) base size GPT-2 model (training input, output, and layernorm params). Note: same as "Random" in Table 2, Section 3.2
2. LSTM: 3-layer, 768 hidden dimension "standard" LSTM (training input, output, and layernorm params). Does not have residual connections or positional embeddings.
3. LSTM*: 12-layer, 768 hidden dimension LSTM (training input, output, position, and layernorm params).

Table 4

All tasks

1. 12: 12-layer, 768 hidden dimension "standard" LSTM (training input, output, and layernorm params).
2. 3: 3-layer, 768 hidden dimension "standard" LSTM (training input, output, and layernorm params).

Table 5

All tasks

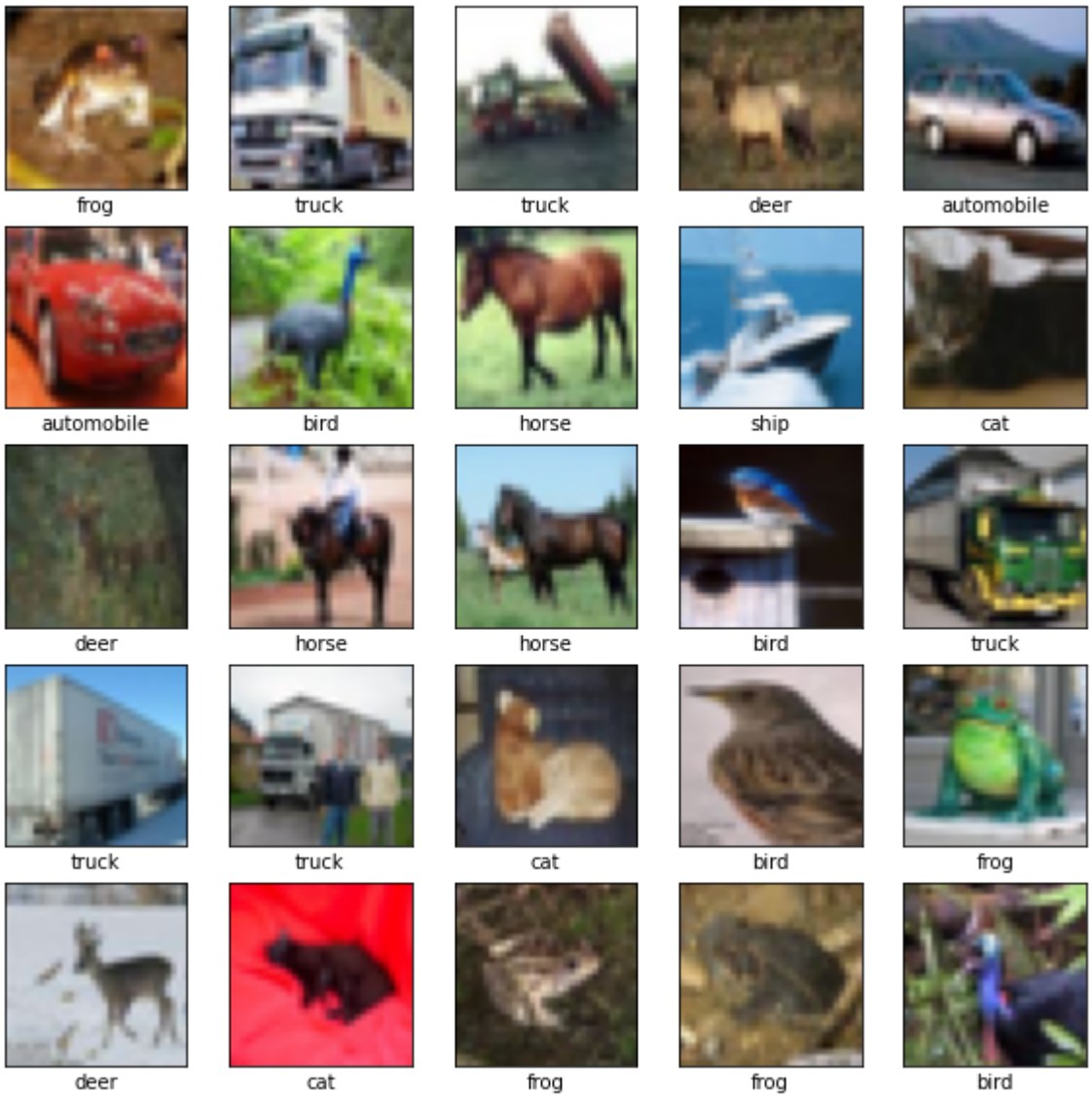
1. 12-layer LSTM: 12-layer, 768 hidden dimension "standard" LSTM (training input, output, and layernorm params). Note: same as "12" in Table 4, Section 3.3.
2. + Residual Connections: 12-layer, 768 hidden dimension LSTM with residual connections (training input, output, and layernorm params).
3. + Positional Embeddings: 12-layer, 768 hidden dimension LSTM with residual connections and positional embeddings (training input, output, position, and layernorm params). Note: same as "LSTM*" in Table 3, Section 3.3.

D.4 Does language pretraining improve compute efficiency over random initialization?

This section refers to Table 6 in Section 3.4.

All tasks

1. FPT: 12-layer base size FPT model (finetuning input, output, position, and layernorm params). Note: same models as "FPT" in Table 2, Section 3.2.
2. Random: 12-layer randomly initialized (default scheme) base size GPT-2 model (training input, output, position, and layernorm params). Note: same models as "Random" in Table 2, Section 3.2.



Create the convolutional base

The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) and MaxPooling2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D) layers.

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to these dimensions, color_channels refers to (R,G,B). In this example, you will configure your CNN to process inputs of shape (32, 32, 3), which is the format of CIFAR images. You can do this by passing the argument `input_shape` to your first layer.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of your model so far:

```
model.summary()
```

```
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

Above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top

To complete the model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Here's the complete architecture of your model:

```
model.summary()
```

```
 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

The network summary shows that (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 8s 4ms/step - loss: 1.4971 - acc
Epoch 2/10
1563/1563 [==============================] - 6s 4ms/step - loss: 1.1424 - acc
Epoch 3/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.9885 - acc
Epoch 4/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.8932 - acc
Epoch 5/10
```

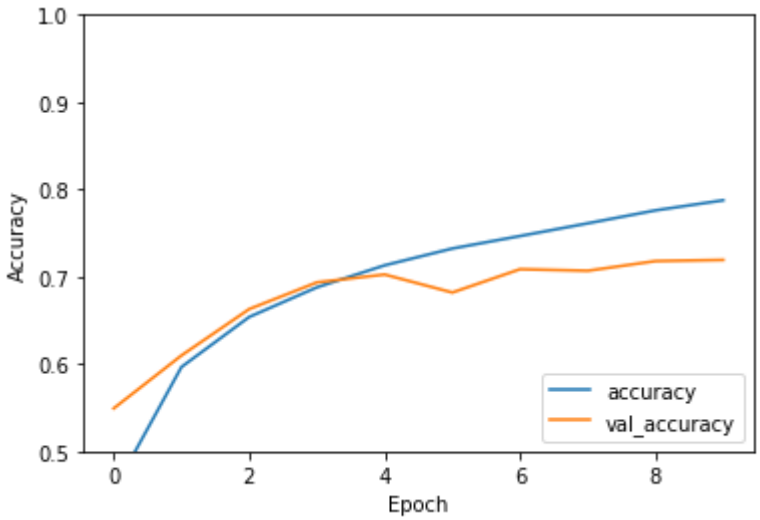
```
1563/1563 [==============================] - 6s 4ms/step - loss: 0.8222 - acc
Epoch 6/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.7663 - acc
Epoch 7/10
```

Evaluate the model

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
313/313 - 1s - loss: 0.8475 - accuracy: 0.7192 - 634ms/epoch - 2ms/step
```




```
print(test_acc)
```

```
0.7192000150680542
```

Your simple CNN has achieved a test accuracy of over 70%. Not bad for a few lines of code! For another CNN style, check out the TensorFlow 2 quickstart for experts